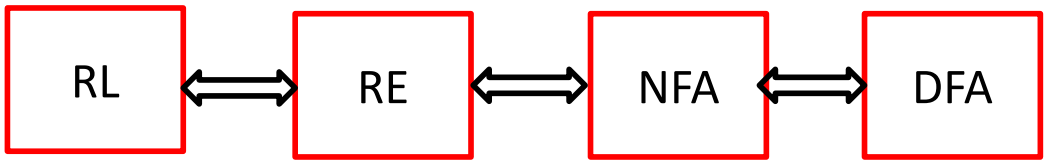
CS F351: Theory of Computation

05 – Context Free Grammar & Push Down Automata

BITS Pilani
Dubai Campus

Dr Elakkiya R, AP/CS

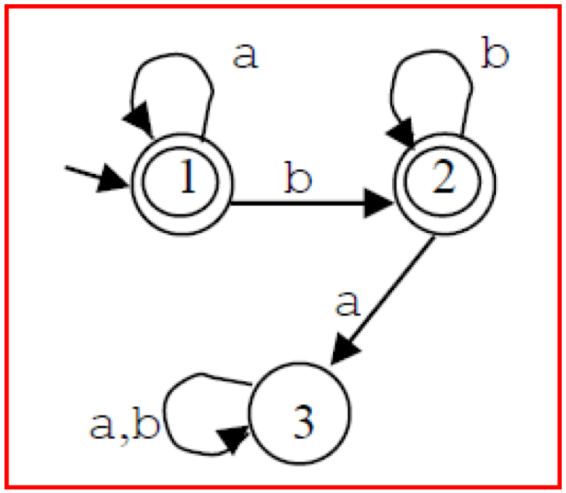
Recap (Regular Lang)



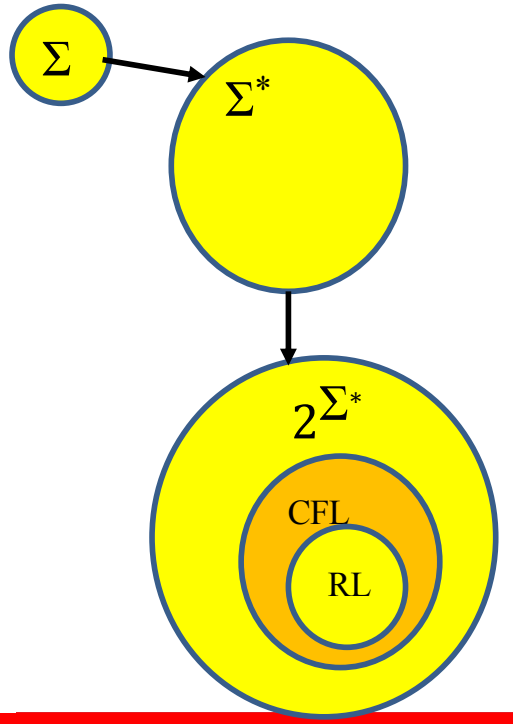
- L = any no of a's followed by b's

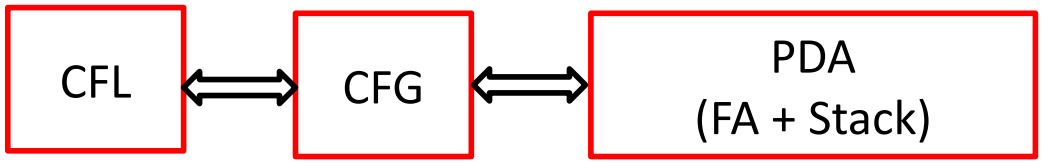
  = $\{\varepsilon, a, b, aa, ab, bb, ..\}$

- $L = \{ a^*b^* \}$



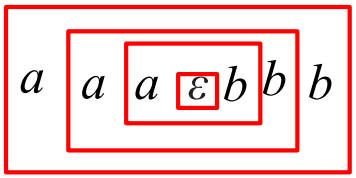
- $L = \{ a^n b^n | n \geq 0 \}$
  - No DFA, No Regular Expression => not a regular language.
- How can such languages be described/recognized?
  - Note: description has to be finite!
- Context-Free Languages (CFL)
- Context-Free Grammar(CFG)

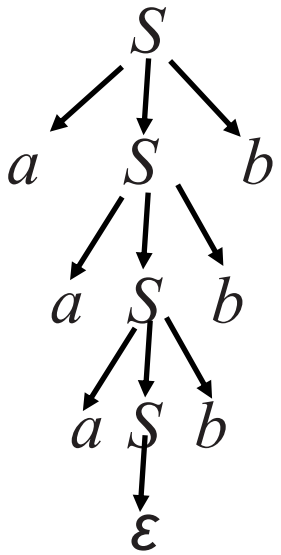




CFG Introduction

- $L = \{ a^n b^n | n \geq 0 \}$
  - No DFA, No Regular Expression => not a regular language.
- How can such languages be described/recognized?
  - Note: description has to be finite!
- $S = aaabbb$ S *is the start variable*





- Set of "substitution/production rules" applied *k* times
  - $S \rightarrow aSb$ applied 3 times
  - $S \rightarrow \varepsilon$ applied 1 time
- Context-Free **Grammar** (CFG)
- Derivation of string a^3b^3

  $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaa\varepsilon bbb \Rightarrow aaabbb$
- Context-Free Language(CFL) (which can be generated by CFG)

Fragment of English Grammar

- Notions of Context-Free Grammar can be compared with English Grammar
- Grammar is way to
  - Describe a language : All possible legally/syntactically/grammatically correct strings
  - Derive a sentence : Generate a legal string of the Language
  - Analyze a sentence : Check if a given string is valid. (Opposite of Derivation)

- Grammar

sentence	–>	**<subject> <verb-phrase> <object>**
subject	–>	This | Computers | I
verb-phrase	–>	**<adverb> <verb>** | **<verb>**
adverb	–>	never
verb	–>	is | run | am | tell
object	–>	the **<noun>** | a **<noun>** | **<noun>**
noun	–>	university | world | cheese | lies

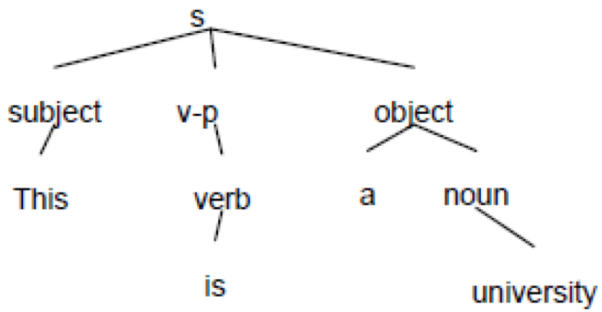
- Some valid stings.

This is a university.
Computers run the world.
I am the cheese.
I never tell lies.

- Derive a sentence/string

sentence	–>	**<subject> <verb-phrase> <object>**
	–>	This **<verb-phrase> <object>**
	–>	This **<verb> <object>**
	–>	This is **<object>**
	–>	This is a **<noun>**
	–>	This is a university

- Parse Tree representation of Derivation



Definition of CFG

innovate achieve lead

- **Definition**: Context-free grammar (CFG) $G = (V, \Sigma, R, S)$ where
  - V is finite set of variables (or non-terminals)
  - Σ is finite set of terminals (with $V \cap \Sigma = \emptyset$)
  - R is finite set of substitution rules (AKA productions), each of the form

    $X \rightarrow Y$

    where
    - $X \in V$ (i.e. a single non-terminal)
    - $Y \in (V \cup \Sigma)^*$ (can be sequence of terminals/non-terminals)
  - S is the start symbol, where $S \in V$

- **Context-Free** Grammar: ***X*** is a **single** non-terminal
  - Substitution is independent of the context X appears in
- **Context-Sensitive** Grammar: **X** $\in (V \cup \Sigma)^+$
  - Substitution depends of the context X appears in.

Derivations in CFG

- ***Definition***: *u* ***derives*** *v, written* $u \overset{*}{\Rightarrow} v$, *if*
  - $u = v$, *or*
  - $\exists\ u_1, u_2, \ldots, u_k$ *for some* $k \geq 0$ *such that*

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$$

- Remark: $\overset{*}{\Rightarrow}$ denotes a sequence of ≥ 0 single-step derivations
- Example: With the rules "$A \rightarrow B1 \mid D0C$",

$$0AA \overset{*}{\Rightarrow} 0D0CB1$$

- **Definition**: The **language** of CFG $G = (V, \Sigma, R, S)$ is

  $L(G) = \{ w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w \}$.

  i.e. From the Starting symbol S, the string w can be derived in 0 or more single-step derivations.

Such a language is called **context-free**, and satisfies $L(G) \subseteq \Sigma^*$

Eg. CFG for $L = \{ a^n \mid n \geq 0 \}$

- CFG $G = (V, \Sigma, R, S)$ with
  - $V = \{S\}$
  - $\Sigma = \{a\}$
  - Rules R: $S \rightarrow aS \mid \varepsilon$
  - $L(G) = \{ a^n \mid n \geq 0 \}$.
  - S derives a^3

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaaS \Rightarrow aaa\varepsilon = aaa$$

- Note that $\rightarrow$ and $\Rightarrow$ are different.

  $\rightarrow$ used in defining rules (productions)

  $\Rightarrow$ used in derivation

Eg. CFGs

- CFG $G = (V, \Sigma, R, S)$ with
  1. $V = \{S\}$
  2. $\Sigma = \{0, 1\}$
  3. Rules R:

$$S \to 0S \mid 1S \mid \varepsilon$$

- Then $L(G) = \Sigma^*$.
- For example, S derives 0100

$$S \Rightarrow 0S \Rightarrow 01S \Rightarrow 010S \Rightarrow 0100S \Rightarrow 0100$$

- CFG $G = (V, \Sigma, R, S)$ with
  1. $V = \{S\}$
  2. $\Sigma = \{0, 1\}$
  3. Rules R:

$$S \to 0S \mid 1S \mid 1$$

- Then $L(G) = \{ w \in \Sigma^* \mid w = s1 \text{ for some } s \in \Sigma^* \}$, i.e., strings that end in 1.
- For example, S derives 011

$$S \Rightarrow 0S \Rightarrow 01S \Rightarrow 011$$

- CFG $G = (V, \Sigma, R, S)$ with
  1. $V = \{S, Z\}$
  2. $\Sigma = \{0, 1\}$
  3. Rules R:

$$S \to 0S1 \mid Z$$
$$Z \to 0Z \mid \varepsilon$$

- Then $L(G) = \{ 0^i 1^j \mid i \geq j \}$.
- For example, S derives $0^5\,1^3$

$$\begin{aligned} S &\Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 000\,Z\,111 \\ &\Rightarrow 000\,0Z\,111 \Rightarrow 000\,00Z\,111 \Rightarrow 000\,00\varepsilon\,111 \\ &= 00000111 \end{aligned}$$

Eg CFG for Palindrome

- PALINDROME $= \{ w \in \Sigma^* \mid w = w^{\mathcal{R}} \}$, where $\Sigma = \{a, b\}$.
- CFG $G = (V, \Sigma, R, S)$ with
  1. $V = \{S\}$
  2. $\Sigma = \{a, b\}$
  3. Rules R:

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \varepsilon$$

- Then $L(G) =$ PALINDROME
- S derives $bbaabb$

$$S \Rightarrow bSb \Rightarrow bbSbb \Rightarrow bbaSabb \Rightarrow bba\varepsilon abb = bbaabb$$

- S derives $aabaa$

$$S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabaa$$

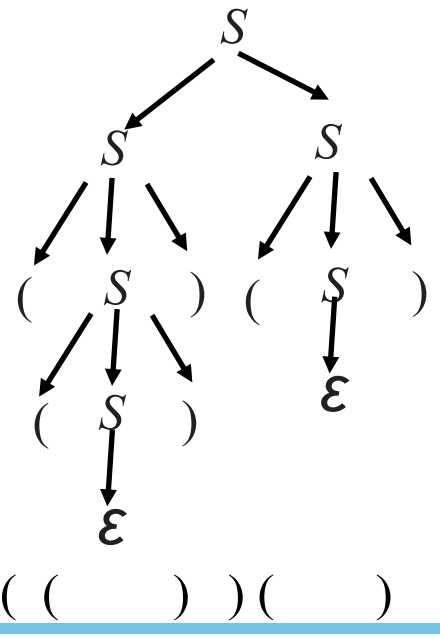
E.g. CFG for $L = \{ w \mid n_a(w) = n_b(w) \}$

- $L = \{ w \in \{a,b\}^* \mid n_a(w) = n_b(w) \}$
- e.g. strings: ε, *ab, ba, aabb, abba, abab, baba, baab, bbaa*
- $S \rightarrow aSb$ (Starting with a, $n_a = n_b = 1$, e.g. $a...b$)
- $S \rightarrow bSa$ (Starting with b, $n_a = n_b = 1$, e.g. $b...a$)
- $S \rightarrow SS$ (For repeating stings, e.g. $a..ba..b/a..bb..a$)
- $S \rightarrow \varepsilon$ (Empty String/Terminating recursion)
- Same as $S \rightarrow aSb \mid bSa \mid SS \mid \varepsilon$

  - $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaSbb \Rightarrow aa\varepsilon bb = aabb$
  - $S \Rightarrow bSa \Rightarrow bbSaa \Rightarrow bbSaa \Rightarrow bb\varepsilon aa = bbaa$
  - $S \Rightarrow SS \Rightarrow aSbaSb \Rightarrow a\varepsilon ba\varepsilon b \Rightarrow abab$
  - $S \Rightarrow SS \Rightarrow bSabSa \Rightarrow b\varepsilon ab\varepsilon a \Rightarrow baba$
  - $S \Rightarrow SS \Rightarrow aSbbSa \Rightarrow a\varepsilon bb\varepsilon a \Rightarrow abba$
  - $S \Rightarrow SS \Rightarrow bSaaSb \Rightarrow b\varepsilon aa\varepsilon b \Rightarrow baab$

E.g. CFG for $L = \{ w \mid w \text{ has balanced parenthesis} \}$

- $L = \{ w \in \Sigma^* \mid w \text{ has balanced parenthesis} \}$, $\Sigma = \{(,)\}$
- e.g. strings: ε, (), (()), (())(), (()())(())
- $S \rightarrow (S)$: Ensure every opening '(' has a closing ')'
- $S \rightarrow SS$: For repeating pattern, e.g. (…)(…)
- $S \rightarrow \varepsilon$:Empty String/Terminating recursion
- Same as $S \rightarrow (S) \mid SS \mid \varepsilon$
  - $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((\varepsilon))S \Rightarrow (())(S) \Rightarrow (())(\varepsilon) = (())()$



Regular Grammar Eg.

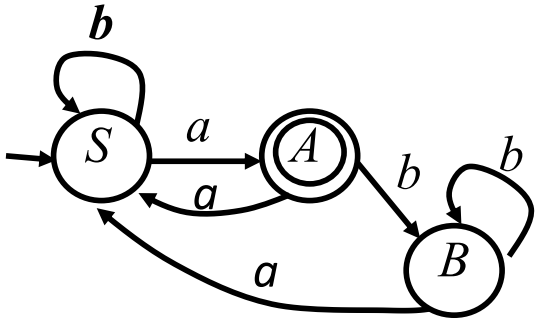
- $L = \{ w \in \{a,b\}^* \mid w$ contains an odd no of ***a***'s and w ends in ***a*** $\}$
  - egs: a, abaa, abbabba (Accepting)
  - b, aba, abaab, abbabaa (Rejecting)

$$S \rightarrow aA \mid bS$$
$$A \rightarrow aS \mid bB \mid \varepsilon$$
$$B \rightarrow aS \mid bB$$

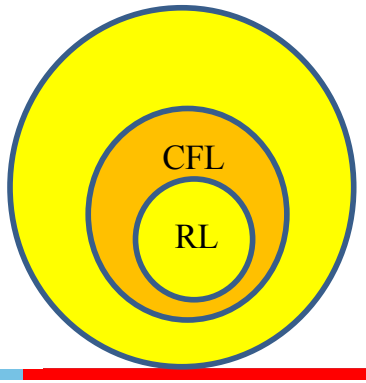
- S – Start state & received even of ***a***'s
  - a : Odd ***a***'s, Goto State A — $S \rightarrow aA$
  - b : Remain in S — $S \rightarrow bS$

- A – received odd of ***a***'s, Also ends in ***a***
  - a : Even ***a***'s, Goto State S — $A \rightarrow aS$
  - b Odd ***a***'s, Goto State B — $A \rightarrow bB$
  - ε : End of input, Accepting input — $A \rightarrow \varepsilon$

- B – received odd of ***a***'s, Got ***b***
  - a : Even ***a***'s, Goto State S — $B \rightarrow aS$
  - b : Odd ***a***'s, Remain in B — $B \rightarrow bB$



R = $b^*(ab^*ab^*)^*a$



CFG for Arithmetic Expressions

- CFG $G = (V, \Sigma, R, S)$ with
  - $V = \{S\}$
  - Σ= $\{+, *, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - Rules R:

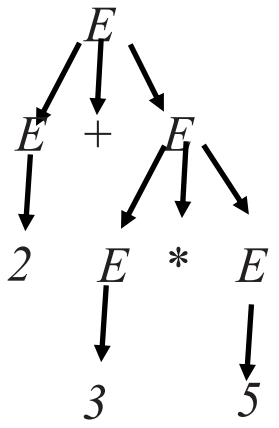
    $E \rightarrow E + E \mid E * E \mid 0 \mid 1 \mid 2 \mid .. \mid 9$
  - L(G) is the set of all valid single digit arithmetic expression with +, *

- $E \Rightarrow E + E \Rightarrow 2 + E \Rightarrow 2 + E * E$

  $\Rightarrow 2 + 3 * E \Rightarrow 2 + 3 * 5$

- $E \Rightarrow E * E \Rightarrow E + E * E$

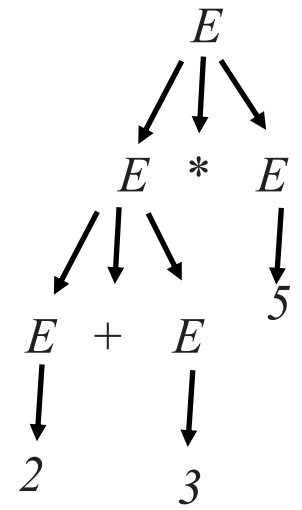
  $\Rightarrow 2 + E * E \Rightarrow 2 + 3 * E \Rightarrow 2 + 3 * 5$



$2 + (3 * 5) = 17$

Right for humans

- Multiple parses tree for a given string means there are 2 interpretations for the string.
- This mean that Grammar G is **ambiguous**



$(2 + 3) * 5 = 25$

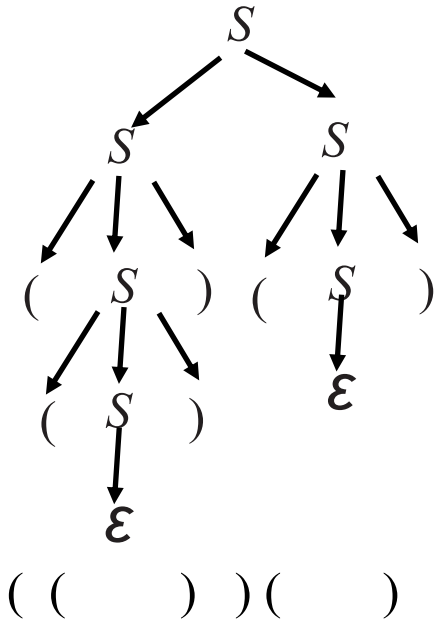
Wrong for humans

E.g. CFG for $L = \{ w \mid w \text{ has balanced parenthesis} \}$

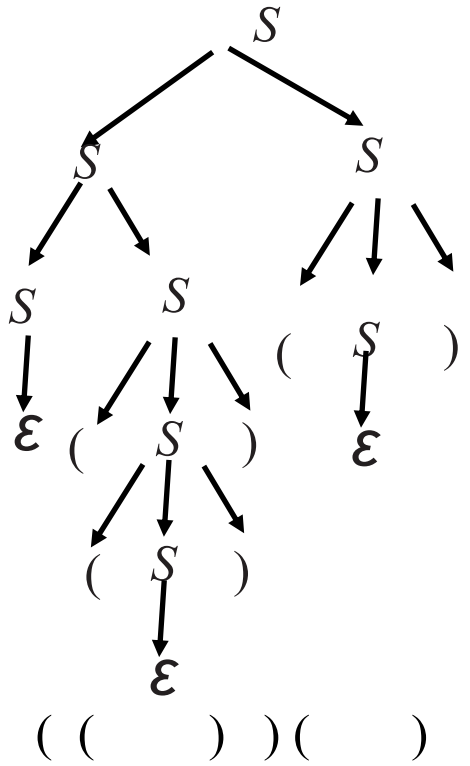
- $L = \{ w \in \Sigma^* \mid w \text{ has balanced parenthesis} \}, \Sigma = \{(,)\}$
- e.g. strings: ε, (), (()), (())(), (()())(())
- $S \rightarrow (S) \mid SS \mid \varepsilon$

- $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S$
  $\Rightarrow ((\varepsilon))S \Rightarrow (())(S) \Rightarrow (())(\varepsilon)$
  $= (())()$

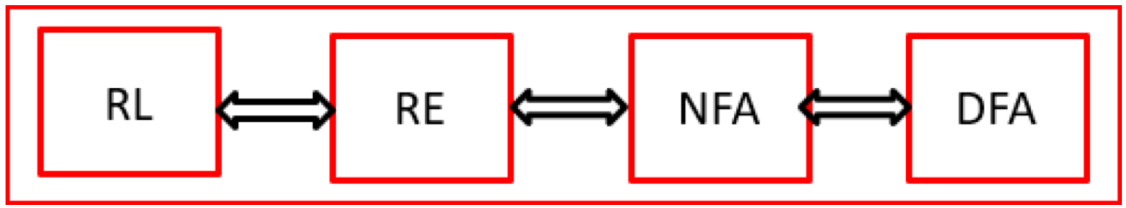
- $S \Rightarrow SS \Rightarrow SSS \Rightarrow \varepsilon SS \Rightarrow \varepsilon SS \Rightarrow \varepsilon(S)S$
  $\Rightarrow ((S))S \Rightarrow ((\varepsilon))S \Rightarrow (())(S) \Rightarrow (())(\varepsilon)$
  $= (())()$

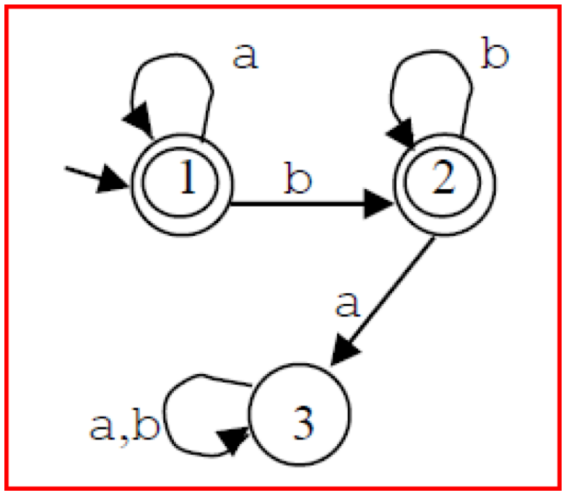


The Grammar G is **ambiguous** because it has multiple parse trees for the same string.



RL/RE/FA -Recap



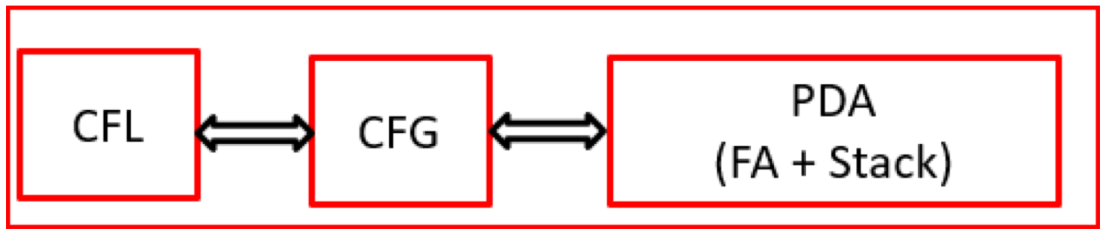


	a	b
1	1	2
2	2	2
3	3	3

- $L = \{ a^*b^* \}$

```
def dfa_execute(cur_state, inp):
    trans = {  ('1','a'):'1', ('1','b'):'2',('2','a'):'3',
               ('2','b):'2',('3','a'):'3', ('3','b'):'3'}

    next_state = trans[(cur_state,inp)]
    print ((cur_state,inp), next_state)
    return next_state
#Driver Program
cur_state = '1'
for i in ['a', 'b', 'b']:
    next_state = dfa_execute(cur_state, i)
    print(next_state, ' ')
    cur_state = next_state
```



Motivation for Stack

- Eg. $L = \{ 0^n 1^n \mid n \geq 0 \}$ (CFL)

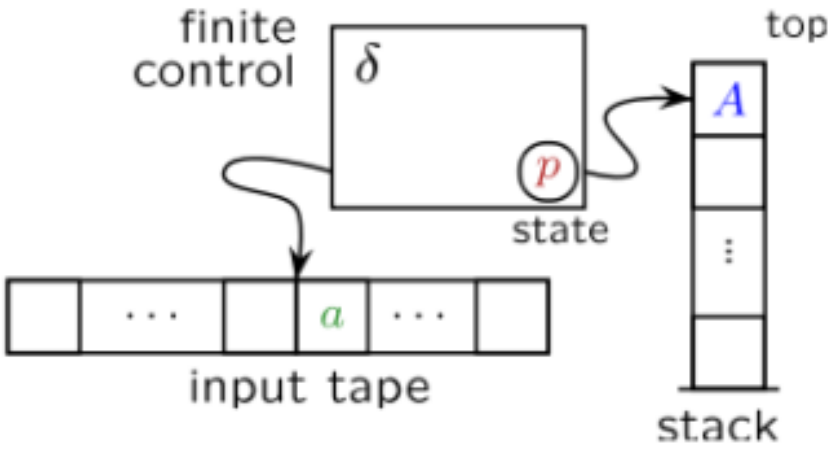
- How can we use a stack to recognize w ∈ L?
  - Push a special bottom of stack symbol $ to the stack
  - As long as you are seeing **0**'s in the input, push an **0** onto the stack.
  - While there are **1**'s in the input AND there is a corresponding **0** on the top of the stack, pop **0** from the stack

  - If at any point there is no **0** on the stack (encounter $), reject the string – not enough **0**'s!
  - If at the end of w, the top of the stack is NOT $ reject the string – not enough **1**'s.
  - Otherwise accept the string.

PDA

- Pushdown automata (PDAs) are for CFLs what finite automata are for regular languages.
  - PDA is presented with a string w over an alphabet Σ.
  - PDA accepts or doesn't accept w.

- Key Differences Between PDA and DFA:
  - PDAs have a single stack.
  - PDAs allow for nondeterminism.



- Input is read left-to-right
- Control has finite memory (NFA)
- State transition depends on
  - current state
  - input
  - top of stack
- Control can
  - push and pop symbols to/from the infinite stack

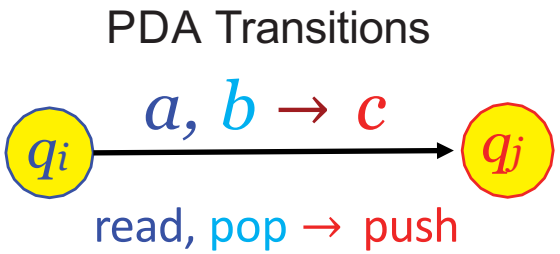
PDA

- PDA (NFA + Stack)
  - States (Q), Input Alphabet (Σ) ($\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$)
  - Stack Alphabet (Γ) $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$. $\$ \in \Gamma$ to mark bottom of stack
    - Symbols pushed onto and popped off the stack
  - Transitions

    •If PDA currently in state q_i,
    - reads $a \in \Sigma_\varepsilon$, and
    - pops $b \in \Gamma_\varepsilon$ off the stack,

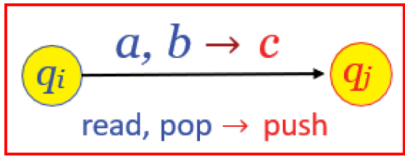
    •then PDA can
    - move to state q_j
    - push $c \in \Gamma_\varepsilon$ onto top of stack

    •If $a = \varepsilon$, then no input symbol is read.
    •If $b = \varepsilon$, then nothing is popped off stack.
    •If $c = \varepsilon$, then nothing is pushed onto stack.



PDA

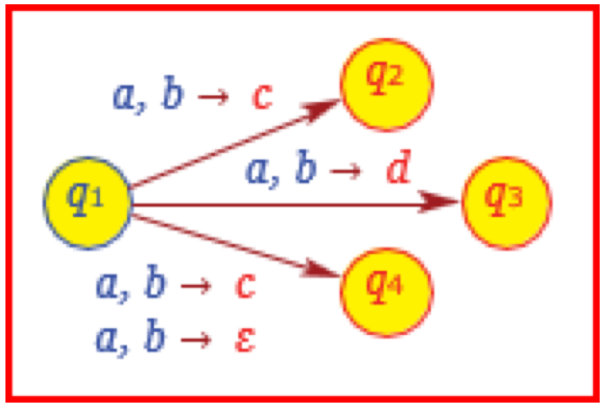
- PDA starts in start state with input string $w \in \Sigma^*$
  - stack initially empty
- PDA makes transitions among states
  - Based on current state, what from Σ_ε is next read from w, and what from Γ_ε is popped from stack.
  - Nondeterministically move to state and push from Γ_ε onto stack.
- If possible to end in accept state $\in F \subseteq Q$ after reading entire input w without crashing, then M accepts w.



Definition of PDA

Def: Pushdown automaton (PDA) $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$:

- Q is finite set of states
- Σ is (finite) input alphabet
- Γ is (finite) stack alphabet
- q_0 is start state, $q_0 \in Q$
- F is set of accept states, $F \subseteq Q$
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow \mathrm{P}(Q \times \Gamma_\varepsilon)$ is transition function



Nondeterministic:

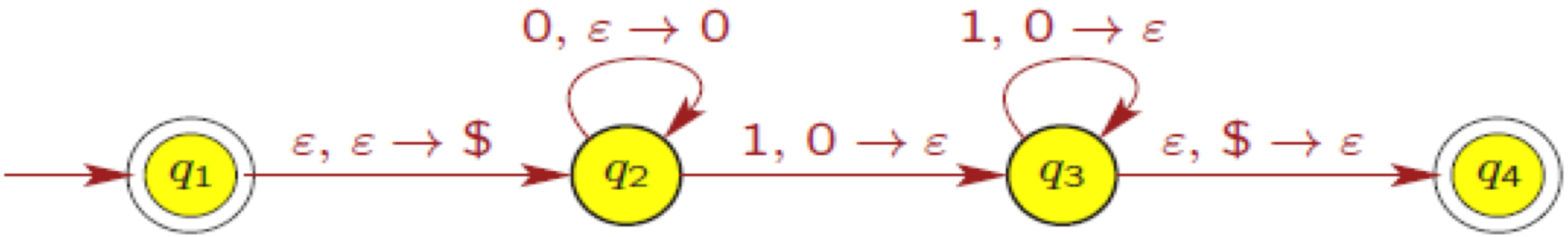
multiple choices when in state q_1,

read $a \in \Sigma_\varepsilon$, and pop $b \in \Gamma_\varepsilon$;

$\delta(q_1, a, b) = \{ (q_2, c), (q_3, d), (q_4, c), (q_4, \varepsilon) \}$

PDA for 0^n1^n $n \geq 0$

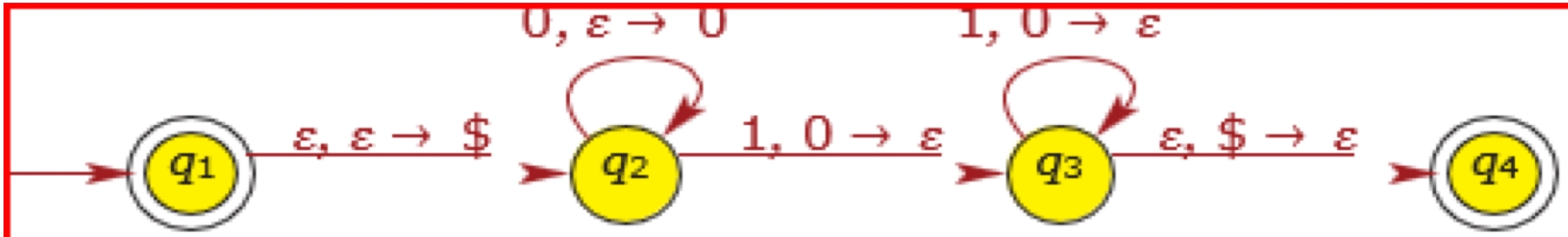
Example: PDA $M = (Q, \Sigma, \Gamma, \delta, q_1, F)$



- $Q = \{q_1, q_2, q_3, q_4\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{0, \$\}$ (use \$ to mark bottom of stack)
- q_1 is the start state
- $F = \{q_1, q_4\}$

Will see that M recognizes language $\{ 0^n1^n \mid n \geq 0 \}$.

PDA for 0^n1^n $n \geq 0$



- transition function $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow P(Q \times \Gamma_\varepsilon)$

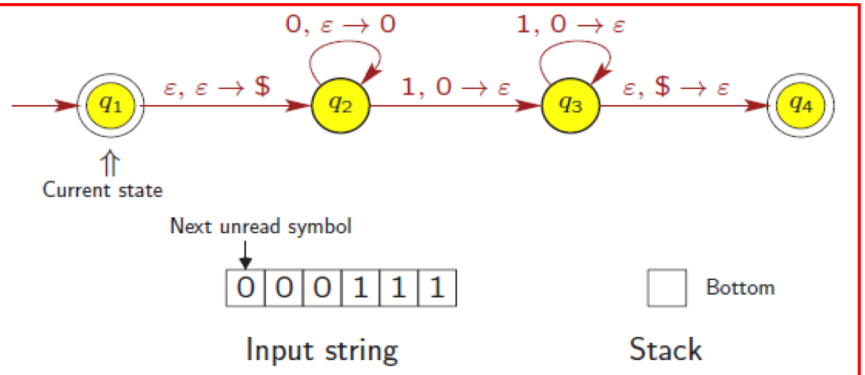
Input:	0			1			ε		
Stack:	0	$	ε	0	$	ε	0	$	ε
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$	$\{(q_3, \varepsilon)\}$					
q_3				$\{(q_3, \varepsilon)\}$				$\{(q_4, \varepsilon)\}$	
q_4									

  - e.g., $\delta(q_2, 1, 0) = \{(q_3, E)\}$.
  - Blank entries are ∅.

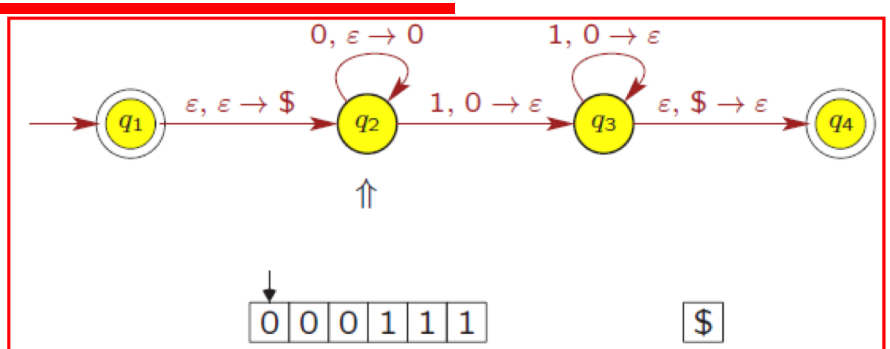
- Let's process string 000111 on our PDA.

  PDA uses stack to match each 0 to a 1.

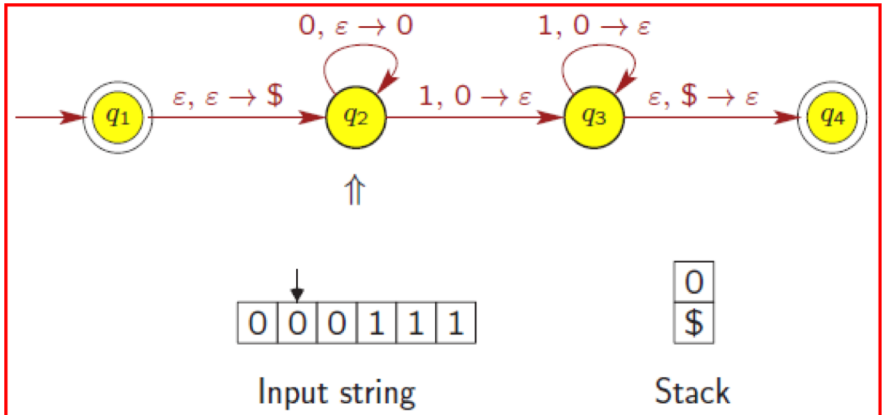
PDA for 0^n1^n $n \geq 0$



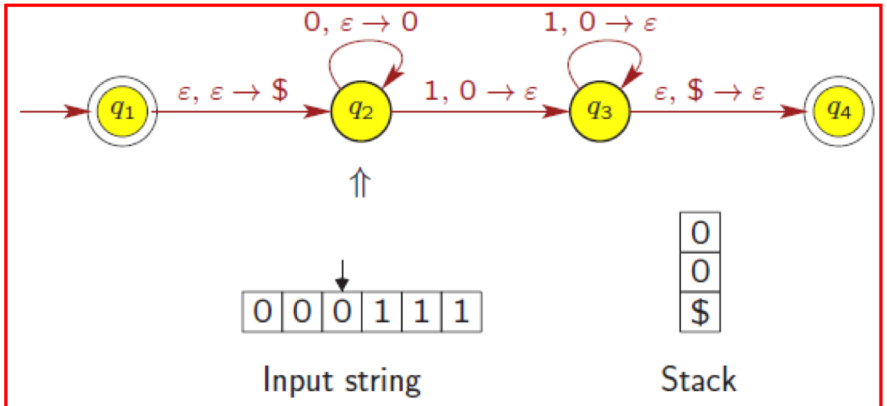
- Start in start state q_1 with stack empty.
- No input symbols read so far.
- Next go to state q_2
  - reading nothing, popping nothing, and pushing $ on stack.



- Next return to state q_2
  - reading input symbol 0
  - popping nothing from stack
  - pushing 0 on stack.

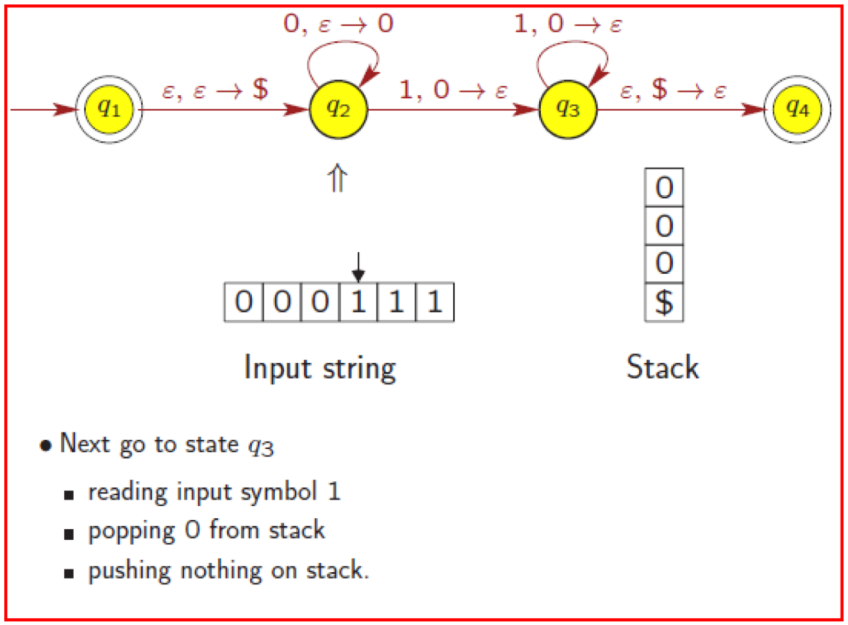


- Next return to state q_2
  - reading input symbol 0
  - popping nothing from stack
  - pushing 0 on stack.

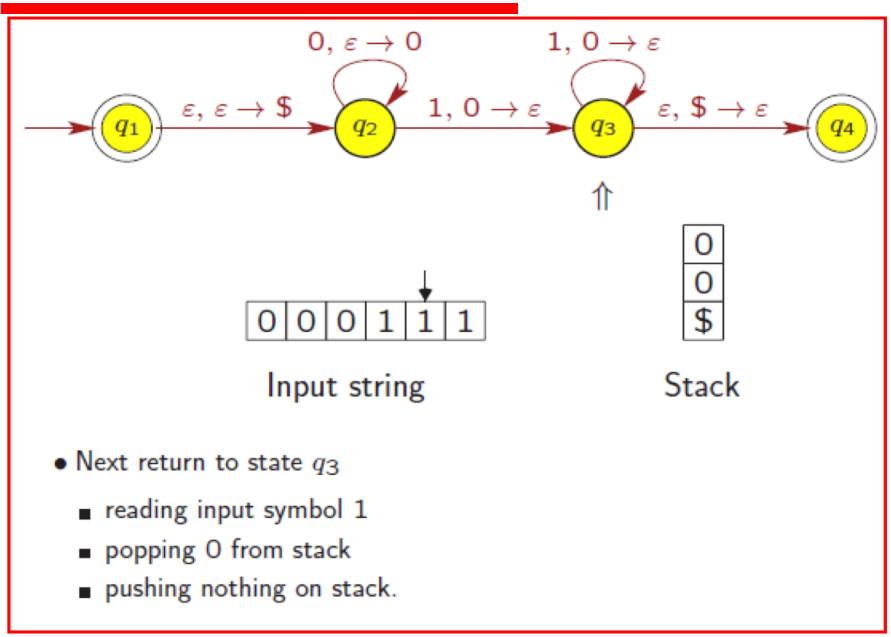


- Next return to state q_2
  - reading input symbol 0
  - popping nothing from stack
  - pushing 0 on stack.

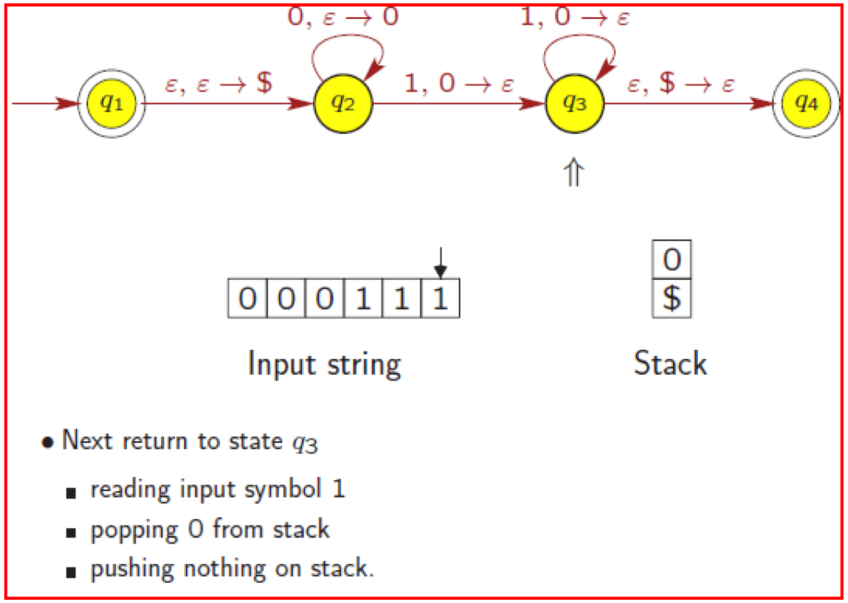
PDA for 0^n1^n $n \geq 0$



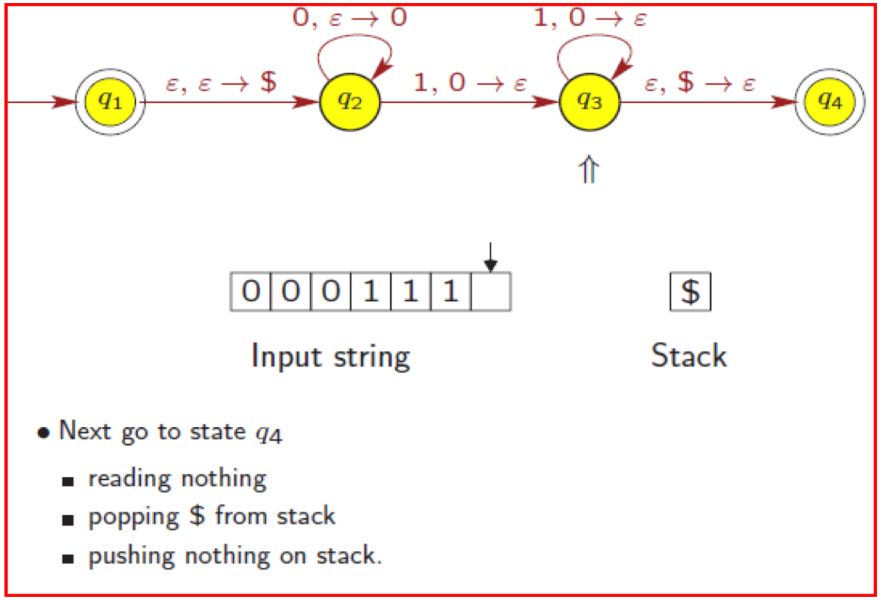
- Next go to state q_3
  - reading input symbol 1
  - popping 0 from stack
  - pushing nothing on stack.



- Next return to state q_3
  - reading input symbol 1
  - popping 0 from stack
  - pushing nothing on stack.

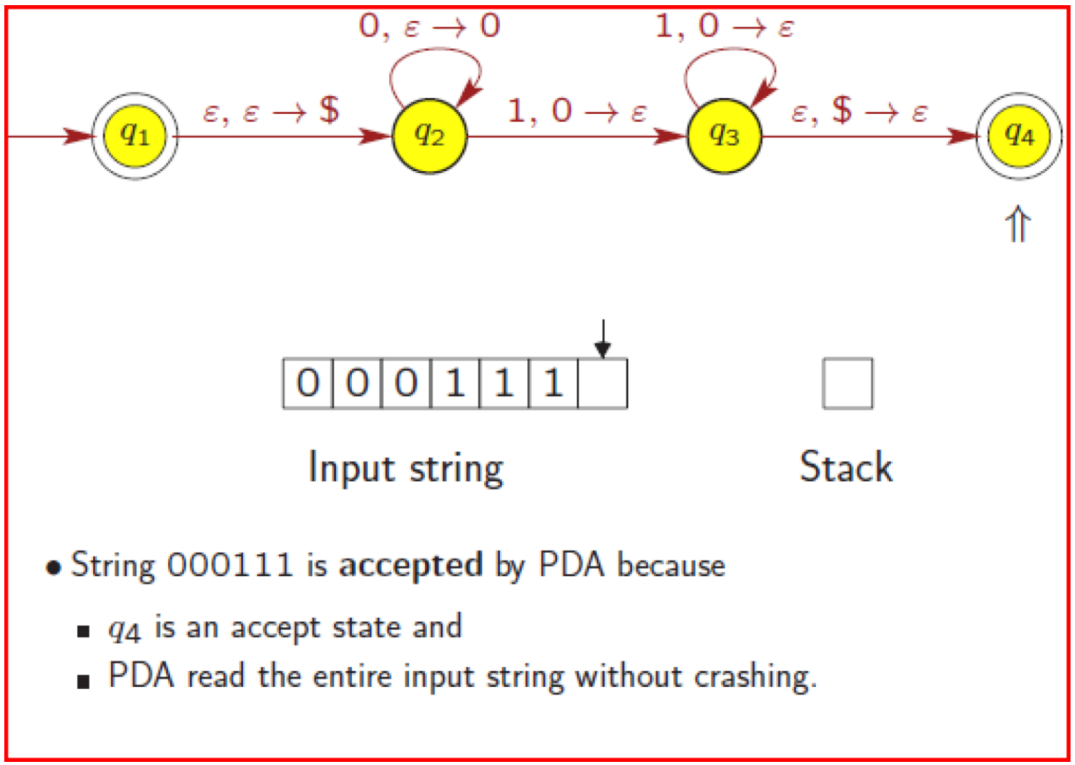


- Next return to state q_3
  - reading input symbol 1
  - popping 0 from stack
  - pushing nothing on stack.



- Next go to state q_4
  - reading nothing
  - popping $ from stack
  - pushing nothing on stack.

PDA for 0^n1^n $n \geq 0$



- String 000111 is **accepted** by PDA because
  - q_4 is an accept state and
  - PDA read the entire input string without crashing.

PDA for language L = {w ∈ Σ* }

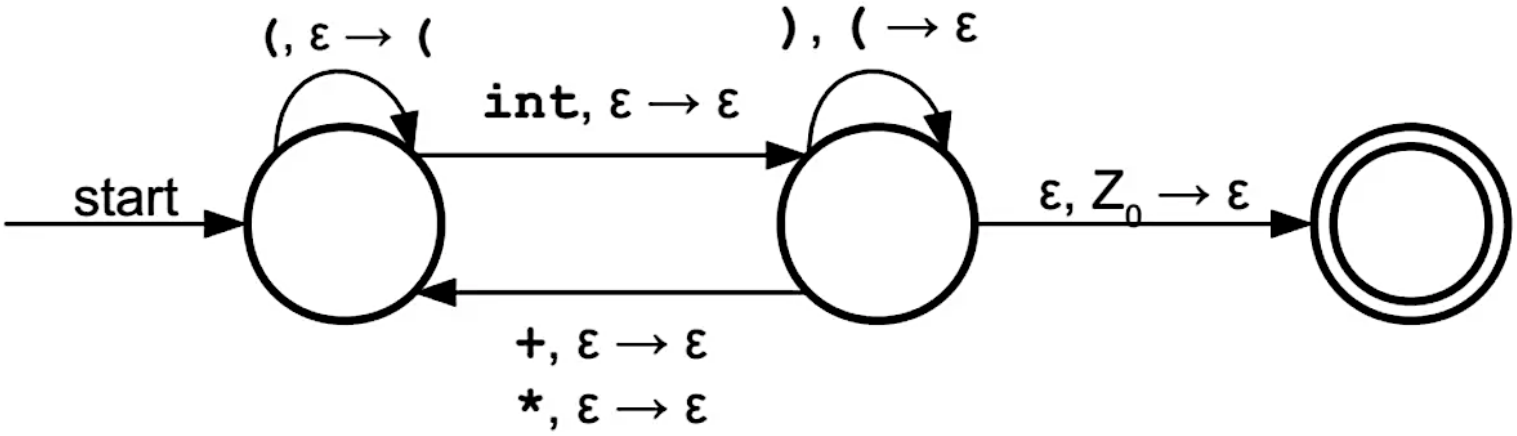
$\Sigma = \{int, +, *, (,)\}$

L = {w ∈ Σ* | w is a legal arithmetic expression}

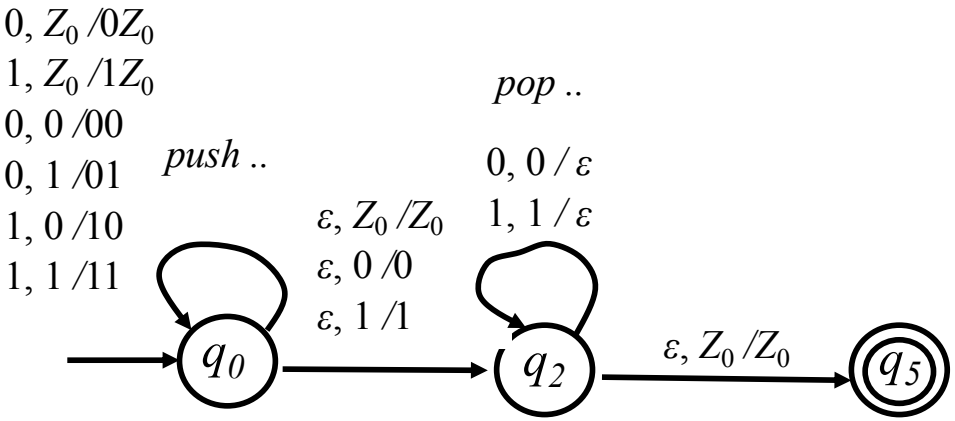
Write the transition function for the given PDA and check the strings

- int + int * int and
- ((int+int)*(int+int)+int)

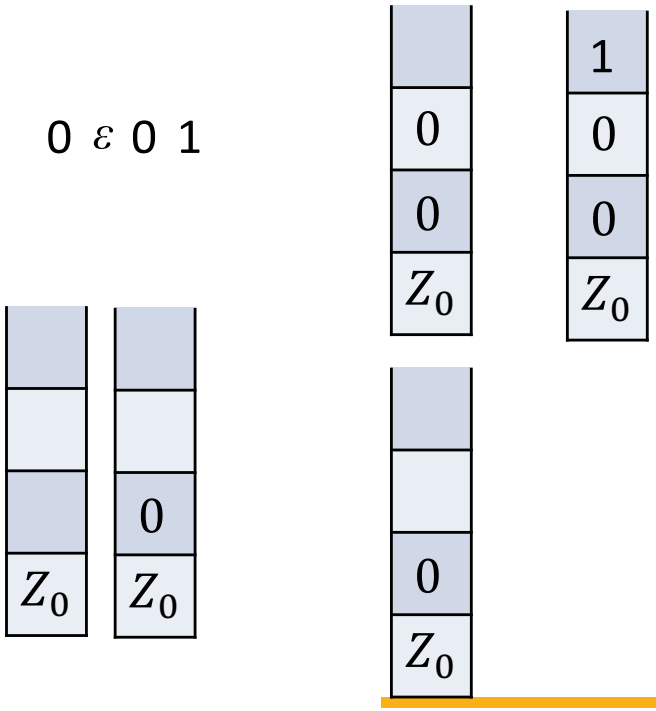
will be accepted by this PDA.



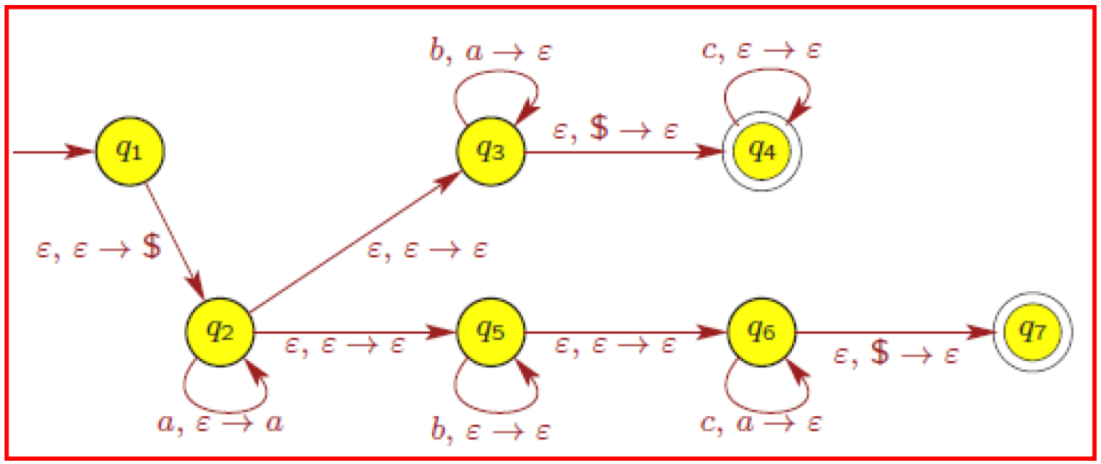
Example: PDA for language $\{ ww^{\mathcal{R}} \mid w \in \{0,1\}^* \}$



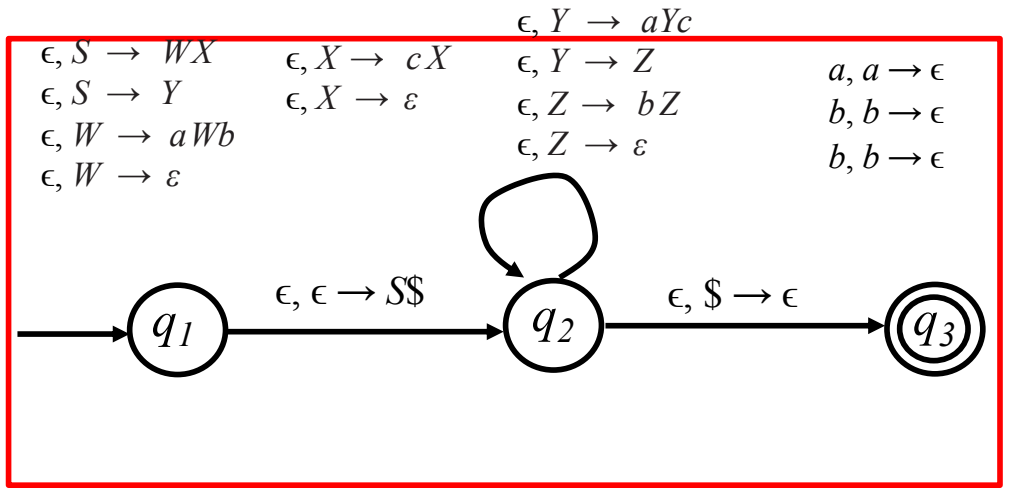
0 ε 0 1



Example: PDA for language $\{ a^i b^j c^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } i = k \}$



$$
\begin{aligned}
S &\to WX \mid Y \\
W &\to aWb \mid \varepsilon \\
X &\to cX \mid \varepsilon \\
Y &\to aYc \mid Z \\
Z &\to bZ \mid \varepsilon
\end{aligned}
$$



CFG => PDA

innovate achieve lead

Lemma

If $L = L(G)$ for some CFG G, then $L = L(M)$ for some PDA M.

Proof Idea:

- Given CFG G, convert it into PDA M with $L(M) = L(G)$.
- Basic idea: build PDA that simulates a leftmost derivation.
- For example, consider CFG $G = (V, \Sigma, R, S)$

  Variables $V = \{S, T\}$

  Terminals $\Sigma = \{0, 1\}$

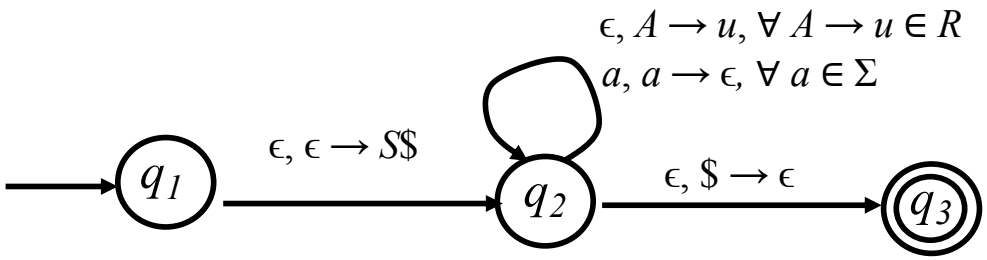
  Rules: $S \rightarrow 0TS1 \mid 1T0, \qquad T \rightarrow 1$

- Leftmost derivation of string $011101 \in L(G)$:

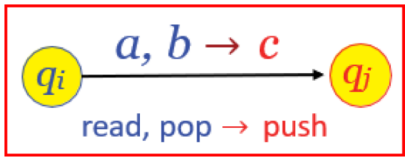
$$S \Rightarrow 0TS1 \Rightarrow 01S1 \Rightarrow 011T01 \Rightarrow 011101$$

CFG => PDA

Construction: CFG $G = (V, \Sigma, R, S)$ ⟹ PDA M



$a \in \Sigma_\varepsilon$ $b,c \in \Gamma_\varepsilon$

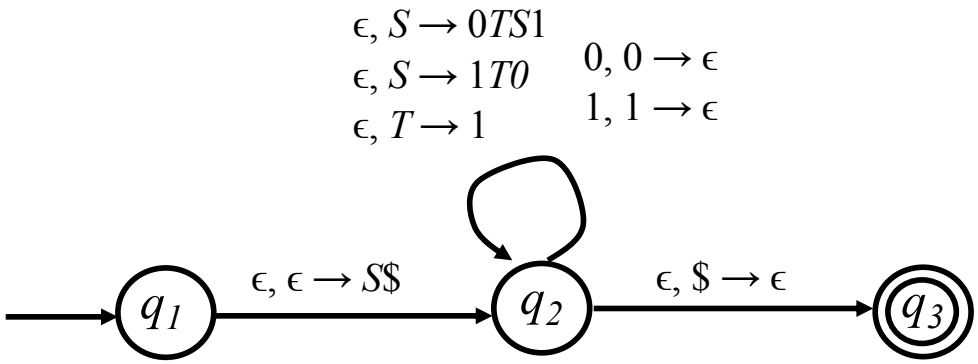


- PDA works as follows: On an input string w

1. Pushes \$ and then S on the stack, where S is start variable.
2. Repeats following until stack empty
   (a) If top of stack is variable $A \in V$, then replace A by some $u \in (\Sigma \cup V)^*$, where $A \to u$ is a rule in R.
   (b) If top of stack is terminal $a \in \Sigma$ and next input symbol is a, then read and pop a.
   (c) If top of stack is \$, then pop it and accept.

CFG => PDA

- Recall CFG rules: $S \rightarrow 0TS1 \mid 1T0,\ T \rightarrow 1$



- PDA is non-deterministic.
- Input alphabet of PDA is the terminal alphabet of CFG $\Sigma = \{0, 1\}$.
- Stack alphabet (Γ) consists of all variables, terminals and \$. i.e.$\Gamma = \{S, T, 0, 1, \$\}$.

PDA simulates a leftmost derivation using CFG

- Pushes RHS of rule in **reverse** order onto stack.

- PDA simulates a leftmost derivation using CFG
  - Pushes RHS of rule in **reverse** order onto stack.

CFL => CFG => PDA

$L = \{ 0^n 1^n \mid n \geq 0 \}$

CFG G

PDA M

Example: Language $\{ 0^n 1^n \mid n \geq 0 \}$ has CFG $G = (V, \Sigma, R, S)$

- Variables $V = \{S\}$
- Terminals $\Sigma = \{0, 1\}$
- Start variable S
- Rules R:

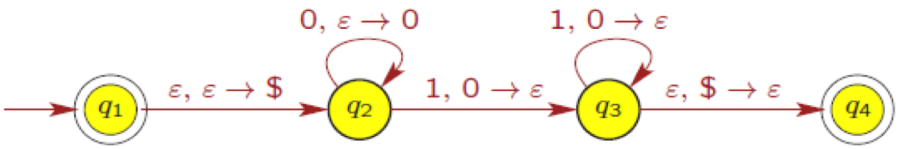
$$S \to 0S1$$
$$S \to \varepsilon$$

- Combine rules with same left-hand side in **Backus-Naur** (or **Backus Normal**) **Form (BNF)**:

$$S \to 0S1 \mid \varepsilon$$

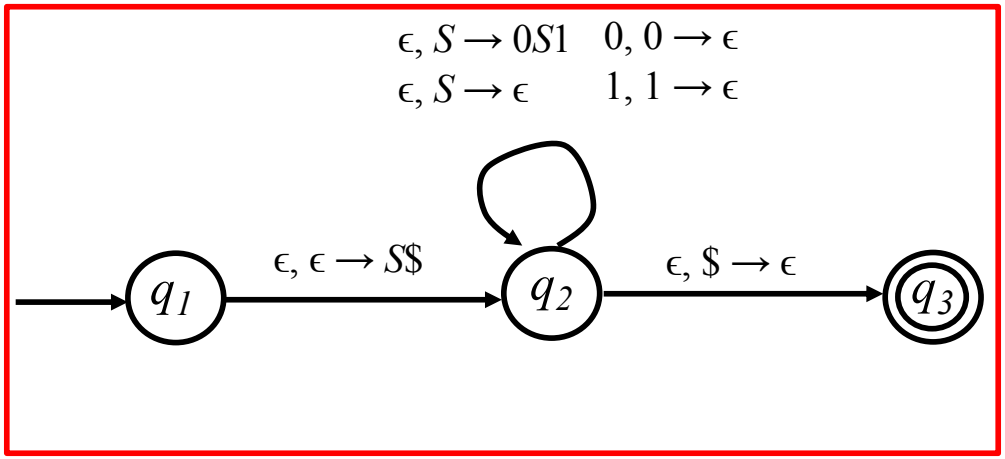
PDA M

Example: PDA $M = (Q, \Sigma, \Gamma, \delta, q_1, F)$



- $Q = \{q_1, q_2, q_3, q_4\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{0, \$\}$ (use \$ to mark bottom of stack)
- q_1 is the start state
- $F = \{q_1, q_4\}$

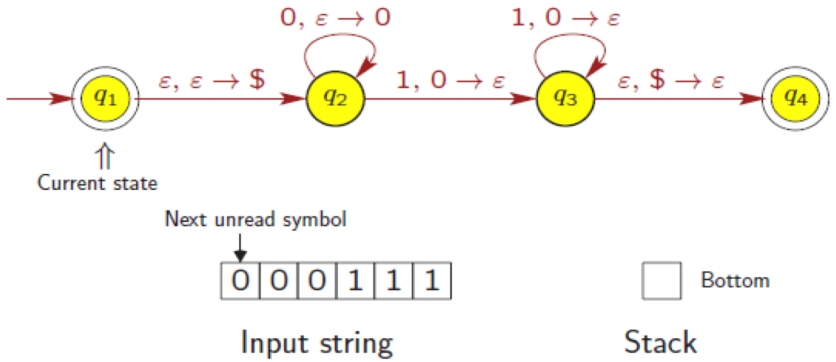
Will see that M recognizes language $\{ 0^n 1^n \mid n \geq 0 \}$.



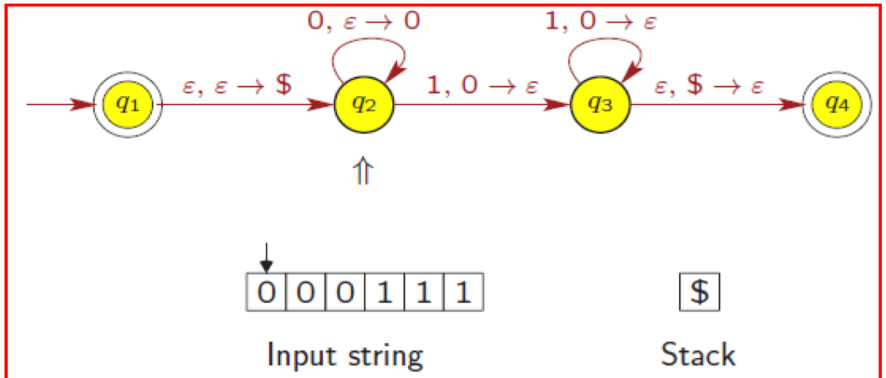
CFL => PDA vs CFG=>PDA : (Execution)

innovate achieve lead

CFL => PDA vs CFG => PDA

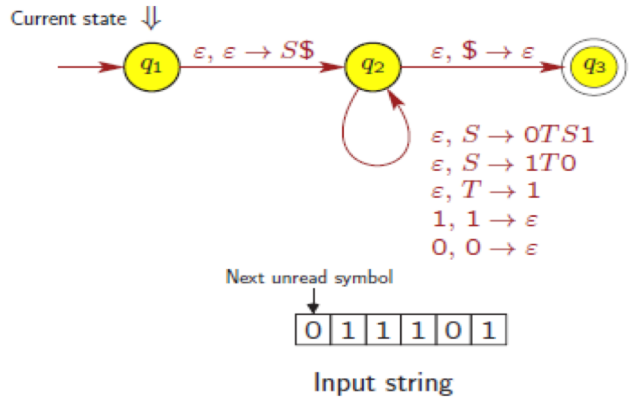


- Start in start state q_1 with stack empty.
- No input symbols read so far.
- Next go to state q_2
  - reading nothing, popping nothing, and pushing \$ on stack.



- Next return to state q_2
  - reading input symbol 0
  - popping nothing from stack
  - pushing 0 on stack.

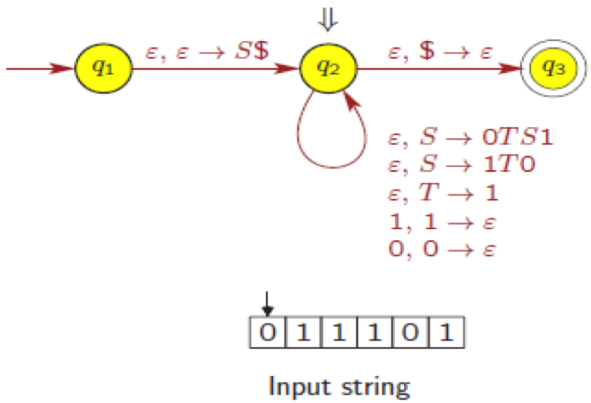
0. Start in state q_1 with 011101 on input tape and empty stack.



Leftmost derivation of string $011101 \in L(G)$:

$$S \Rightarrow 0TS1 \Rightarrow 01S1 \Rightarrow 011T01 \Rightarrow 011101$$

1. Read nothing, pop nothing, move to q_2, and push \$ and then S.

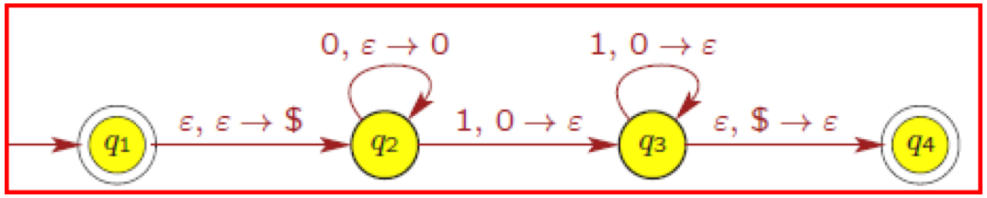


Leftmost derivation of string $011101 \in L(G)$:

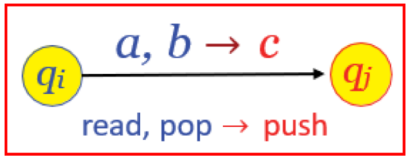
$$\underline{S} \Rightarrow 0TS1 \Rightarrow 01S1 \Rightarrow 011T01 \Rightarrow 011101$$

CFG => PDA

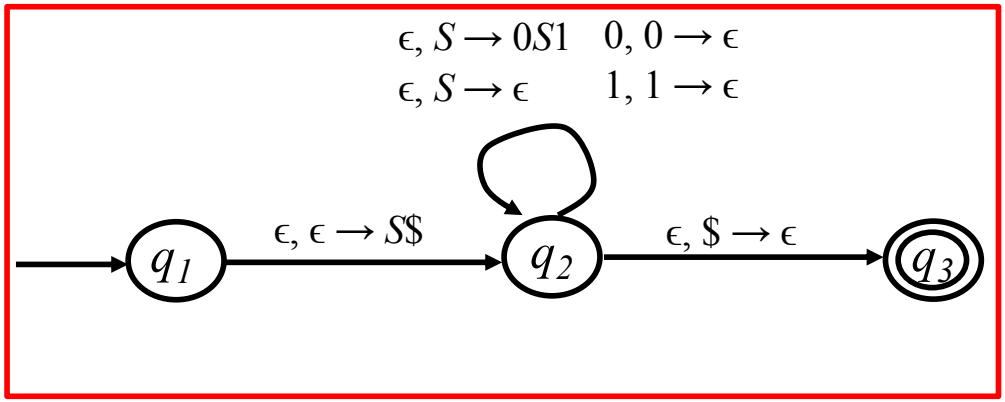
CFL => PDA



$a \in \Sigma_\varepsilon$ $b,c \in \Gamma_\varepsilon$

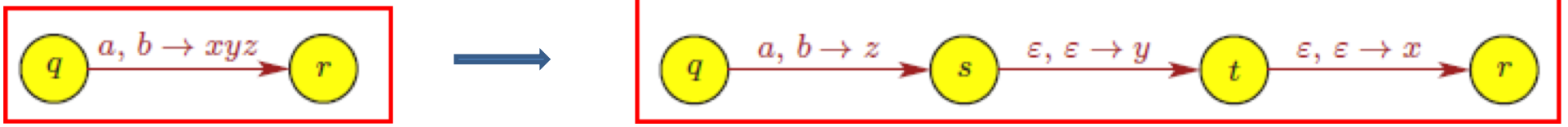


CFG => PDA

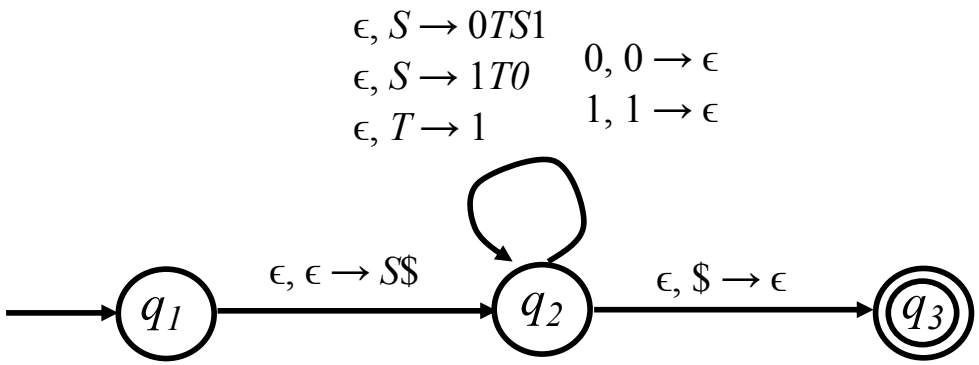


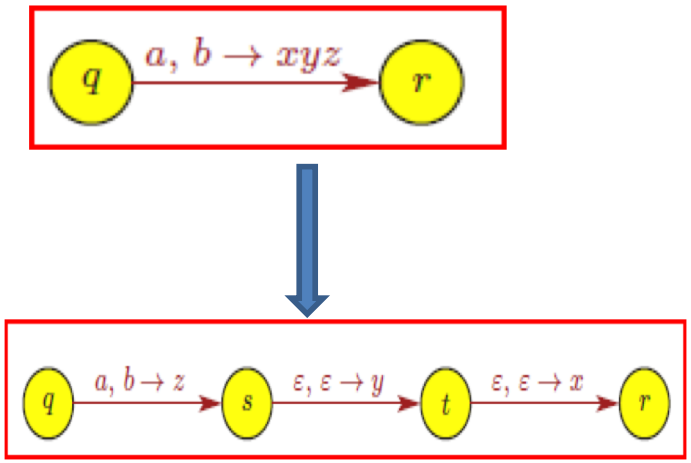
Pushing **strings** onto stack instead of ≤ 1 symbols, which is not allowed in PDA specification

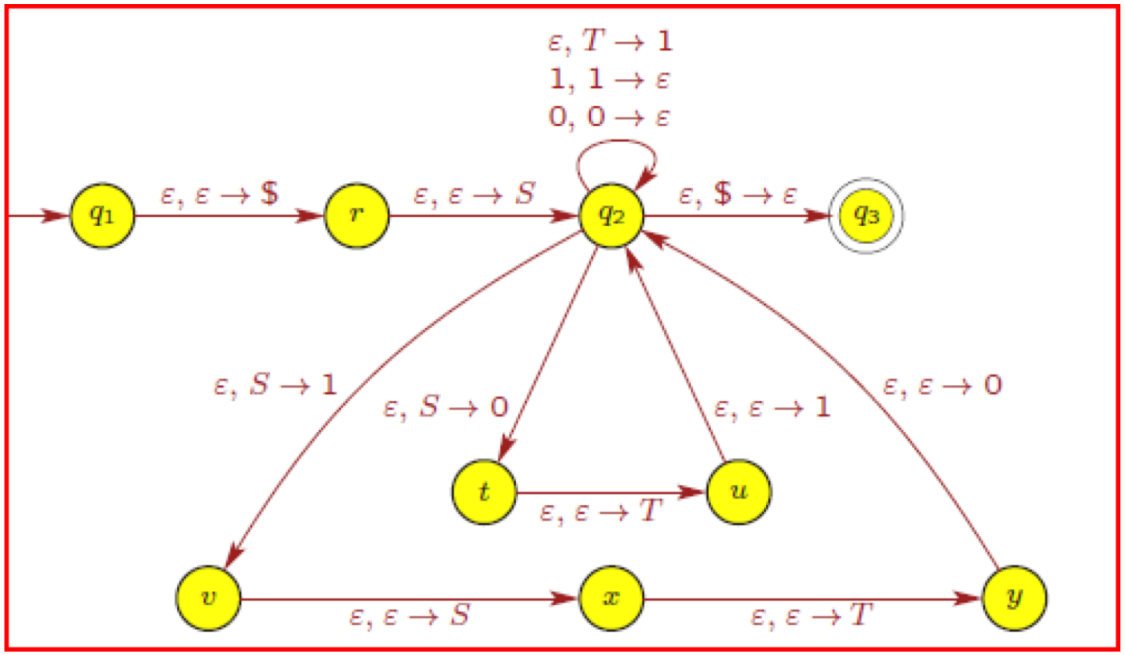
Solution:



CFG => PDA

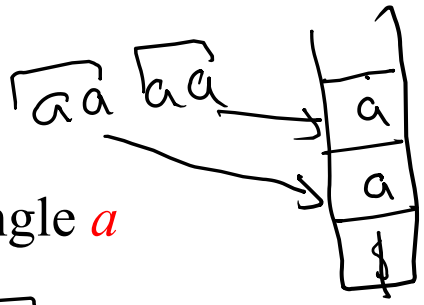

ϵ, S → 0TS1
ϵ, S → 1T0
ϵ, T → 1
0, 0 → ϵ
1, 1 → ϵ
ϵ, ϵ → S$
ϵ, $ → ϵ
q1
q2
q3


q
a, b → xyz
r
q
a, b → z
s
ε, ε → y
t
ε, ε → x
r

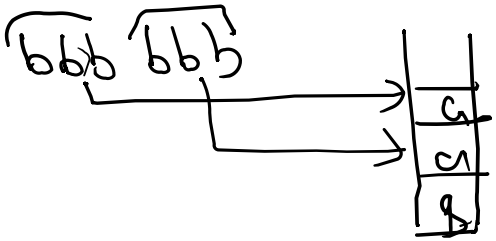

ε, T → 1
1, 1 → ε
0, 0 → ε
q1
ε, ε → $
r
ε, ε → S
q2
ε, $ → ε
q3
ε, S → 1
ε, S → 0
ε, ε → 1
ε, ε → 0
t
u
ε, ε → T
v
ε, ε → S
x
ε, ε → T
y

$L = \{ a^{2n}b^{3n} \mid n \geq 0 \}$

- Eg strings: ϵ, *aabbb*, *aaaabbbbbb*



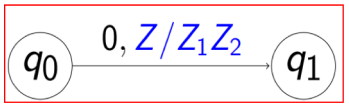
- For every 2 *a*'s, push a single *a* on to the stack
  - For the 1st *a* tos remains the same, for the 2nd *a*, push a single *a*



- For every 3 *b*'s, pop a single *a* from the stack
  - For the 1st & 2nd *b*, tos remains the same,
  - For the 3rd *b*, pop a single *a*

  - If at any point there is no *a* on the stack (encounter $), reject the string – more ***b***'s than ***a***'s
  - If at the end of w, the top of the stack is NOT $ reject the string – more ***a***'s than ***b***'s .
  - Otherwise accept the string.

$L = \{ a^{2n}b^{3n} \mid n \geq 0 \}$



innovate achieve lead

Transition Function

$\delta(q_0, a, Z_0) = (q_1, Z_0)$

$\delta(q_0, a, a) = (q_1, a)$

$\delta(q_0, \varepsilon, Z_0) = (q_2, Z_0)$

$\delta(q_0, \varepsilon, a) = (q_2, a)$

$\delta(q_1, a, Z_0) = (q_0, aZ_0)$

$\delta(q_1, a, a) = (q_0, aa)$

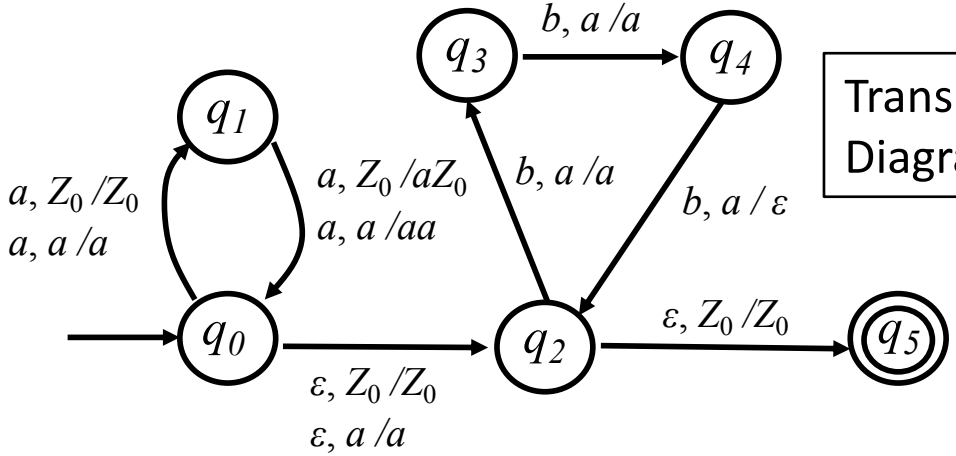
$\delta(q_2, b, a) = (q_3, a)$

$\delta(q_3, b, a) = (q_4, a)$

$\delta(q_4, b, a) = (q_2, \varepsilon)$

$\delta(q_2, \varepsilon, Z_0) = (q_5, Z_0)$

Transition/State Diagram



Instantaneous Description

$(q_0, aabbb, Z_0)$

$\vdash (q_1, abbb, Z_0)$

$\vdash (q_0, \varepsilon bbb, aZ_0)$

$\vdash (q_2, bbb, aZ_0)$

$\vdash (q_3, bb, aZ_0)$

$\vdash (q_4, b, aZ_0)$

$\vdash (q_2, \varepsilon, Z_0)$

$\vdash (q_5, Z_0)$

$\delta(q_0, a, Z_0) = (q_1, Z_0)$

$\delta(q_1, a, Z_0) = (q_0, aZ_0)$

$\delta(q_0, \varepsilon, a) = (q_2, a)$

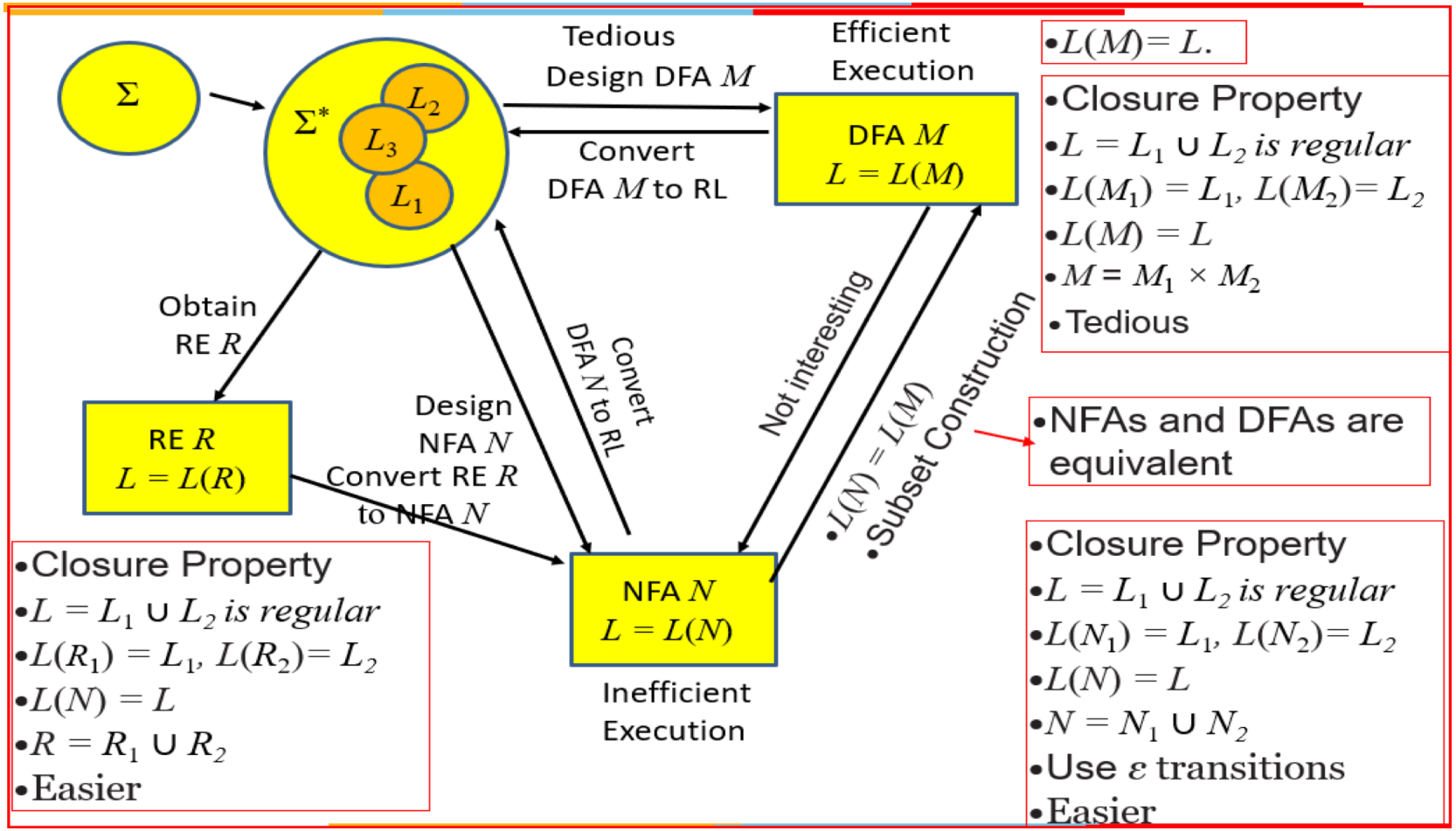
$\delta(q_2, b, a) = (q_3, a)$

$\delta(q_3, b, a) = (q_4, a)$

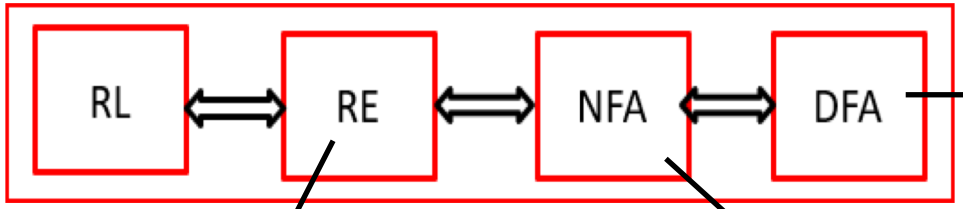
$\delta(q_4, b, a) = (q_2, \varepsilon)$

$\delta(q_2, \varepsilon, Z_0) = (q_5, Z_0)$

RL ⇔ RE ⇔ NFA ⇔ DFA


Σ
Σ*
L_2
L_3
L_1
Tedious
Design DFA M
Convert
DFA M to RL
Efficient
Execution
DFA M
$L = L(M)$
Obtain
RE R
RE R
$L = L(R)$
Design
NFA N
Convert RE R
to NFA N
Convert
DFA N to RL
Not interesting
$L(N) = L(M)$
Subset Construction
NFA N
$L = L(N)$
Inefficient
Execution
•$L(M) = L$.
•Closure Property
•$L = L_1 \cup L_2$ is regular
•$L(M_1) = L_1$, $L(M_2) = L_2$
•$L(M) = L$
•$M = M_1 \times M_2$
•Tedious
•NFAs and DFAs are equivalent
•Closure Property
•$L = L_1 \cup L_2$ is regular
•$L(R_1) = L_1$, $L(R_2) = L_2$
•$L(N) = L$
•$R = R_1 \cup R_2$
•Easier
•Closure Property
•$L = L_1 \cup L_2$ is regular
•$L(N_1) = L_1$, $L(N_2) = L_2$
•$L(N) = L$
•$N = N_1 \cup N_2$
•Use ε transitions
•Easier

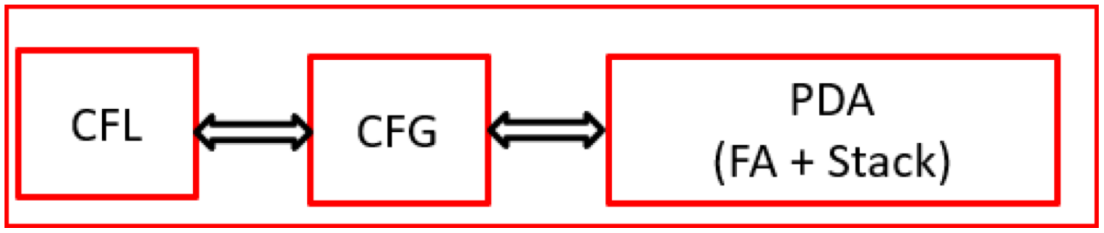
Closure Properties of RLs vs CFLs



- Closure Property
- ∪, ∘, *
- $L = L_1 \cup L_2$ *is regular*
- $L(R_1) = L_1,\ L(R_2) = L_2$
- $L(N) = L$
- $R = R_1 \cup R_2$
- Easier

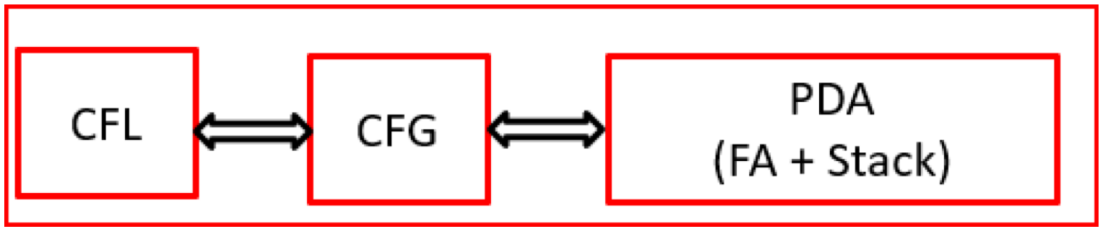
- Closure Property
- ∪, ∘, *, ∩
- $L = L_1 \cup L_2$ *is regular*
- $L(N_1) = L_1,\ L(N_2) = L_2$
- $L(N) = L$
- $N = N_1 \cup N_2$
- Use ε transitions
- Easier

- Closure Property
- ∪, ∩, $\bar{L}$
- $L = L_1 \cup L_2$ *is regular*
- $L(M_1) = L_1,\ L(M_2) = L_2$
- $L(M) = L$
- $M = M_1 \times M_2$ (Cross Product)
- Tedious



- Closure Property
- ∪, ∘, *
- A

Closure Properties of CFLs



- Closure Property
- ∪, ∘, *
- $L = L_1 \cup L_2$ *is CFL*
- Easier

Is class of CFLs closed under Union? : Yes

Theorem:
If L_1 and L_2 are CFLs, then union $L_1 \cup L_2$ is CFL.

Proof.
- Assume
  L_1 has CFG $G_1 = (V_1, \Sigma, R_1, S_1)$
  L_2 has CFG $G_2 = (V_2, \Sigma, R_2, S_2)$.
- Assume that $V_1 \cap V_2 = \emptyset$.
- $L_1 \cup L_2$ has CFG $G = (V, \Sigma, R, S)$ with
  $V = V_1 \cup V_2 \cup \{S\}$, where $S \notin V_1 \cup V_2$ is new start variable
- $R = R_1 \cup R_2 \cup \{S \rightarrow S_1 \mid S_2\}$.

Example of Union of CFLs
- Suppose L_1 has CFG G_1 with rules:
  $S \rightarrow aS \mid bXb$
  $X \rightarrow ab \mid baXb$
- Suppose L_2 has CFG G_2 with rules:
  $S \rightarrow Sbb \mid aXba$
  $X \rightarrow b \mid XaX$
- Then $L_1 \cup L_2$ has CFG G with start variable S and rules:
  $S \rightarrow S_1 \mid S_2$
  $S_1 \rightarrow aS_1 \mid bX_1b$
  $X_1 \rightarrow ab \mid baX_1b$
  $S_2 \rightarrow S_2bb \mid aX_2ba$
  $X_2 \rightarrow b \mid X_2aX_2$

Closure Properties of CFLs

Is class of CFLs closed under concatenation? : Yes

Theorem:
If L_1 and L_2 are CFLs, then concatenation $L_1 \circ L_2$ is CFL.

Proof.
- Assume
  L_1 has CFG $G_1 = (V_1, \Sigma, R_1, S_1)$
  L_2 has CFG $G_2 = (V_2, \Sigma, R_2, S_2)$.
- Assume that $V_1 \cap V_2 = \emptyset$.
- $L_1 \circ L_2$ has CFG $G = (V, \Sigma, R, S)$ with
  $V = V_1 \cup V_2 \cup \{S\}$, where $S \notin V_1 \cup V_2$ is new start variable
- $R = R_1 \cup R_2 \cup \{S \rightarrow S_1 S_2\}$.

Example of Concatenation of CFLs
- Suppose L_1 has CFG G_1 with rules:
  $S \rightarrow aS \mid bXb$
  $X \rightarrow ab \mid baXb$
- Suppose L_2 has CFG G_2 with rules:
  $S \rightarrow Sbb \mid aXba$
  $X \rightarrow b \mid XaX$
- Then $L_1 \circ L_2$ has CFG G with start variable S and rules:
  $S \rightarrow S_1 S_2$
  $S_1 \rightarrow aS_1 \mid bX_1b$
  $X_1 \rightarrow ab \mid baX_1b$
  $S_2 \rightarrow S_2bb \mid aX_2ba$
  $X_2 \rightarrow b \mid X_2aX_2$

Note: The variable X in G_1 and G_2 have been renamed as X_1 and X_2 respectively. This ensures $V_1 \cap V_2 = \emptyset$

Closure Properties of CFLs

Is class of CFLs closed under Klenee-Star? : Yes

Theorem:
If L_1 is CFL, then A_1^* is CFL.

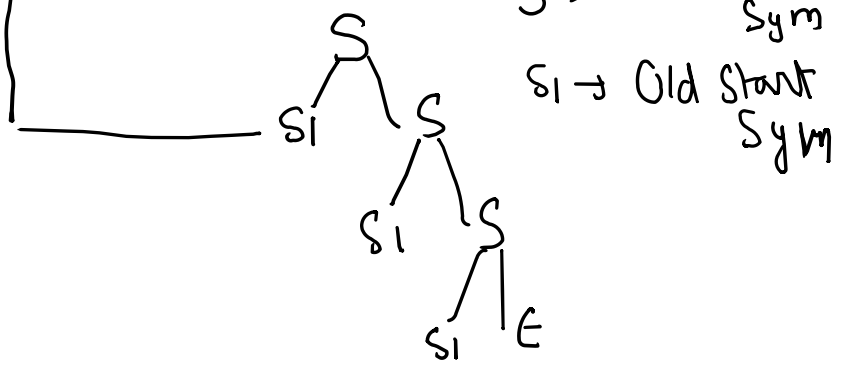
Proof.

- Assume
  L_1 has CFG $G_1 = (V_1, \Sigma, R_1, S_1)$

- has CFG $G = (V, \Sigma, R, S)$ with
  $V = V_1 \cup \{S\}$, where $S \notin V_1$ is new start variable
- $R = R_1 \cup \{S \rightarrow S_1 S, S \rightarrow \varepsilon\}$.

Example of Kleene Start of CFLs

- Suppose L_1 has CFG G_1 with rules:
  $S \rightarrow aS \mid bXb$
  $X \rightarrow ab \mid baXb$
- Then L_1^* has CFG G with start variable S and rules:
  $S \rightarrow S_1 S \mid \varepsilon$
  $S_1 \rightarrow aS_1 \mid bXb$
  $X \rightarrow ab \mid baXb$



Closure Properties of CFLs

Is class of CFLs closed under intersection? : **No**

Theorem:
If L_1 and L_2 are CFLs, then intersection
$L_1 \cap L_2$ is not necessarily CFL.

Proof. (by Contradiction)

- Assume
  $L_1 = \{ a^n b^n c^k \mid n \geq 0, k \geq 0 \}$
  $L_2 = \{ a^k b^n c^n \mid n \geq 0, k \geq 0 \}$
- $L = L_1 \cap L_2 = \{ a^n b^n c^n \mid n \geq 0 \}$
- Illustration :
- $L_1 = \{ \ldots, a^3b^3c^2, a^3b^3c^3, \ldots \}$
- $L_2 = \{ \ldots, a^2b^3c^3, a^3b^3c^3, \ldots \}$
- $L = L_1 \cap L_2 = \{ \ldots, a^3b^3c^3, \ldots \}$

Closure Properties of CFLs

Is class of CFLs closed under Complement? :
No

Theorem:
If L_1 is CFL, then complement
$L_1 \cap L_2$ is not necessarily CFL.

Proof.

- $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$

Another method.
A : Union is Closed
B: Complement is closed
C: Intersection is closed.

A and B -> C
¬C -> ¬(A and B)
Since A is true, B has to be false for ¬(A and B) to be true

Application of Closure Properties.

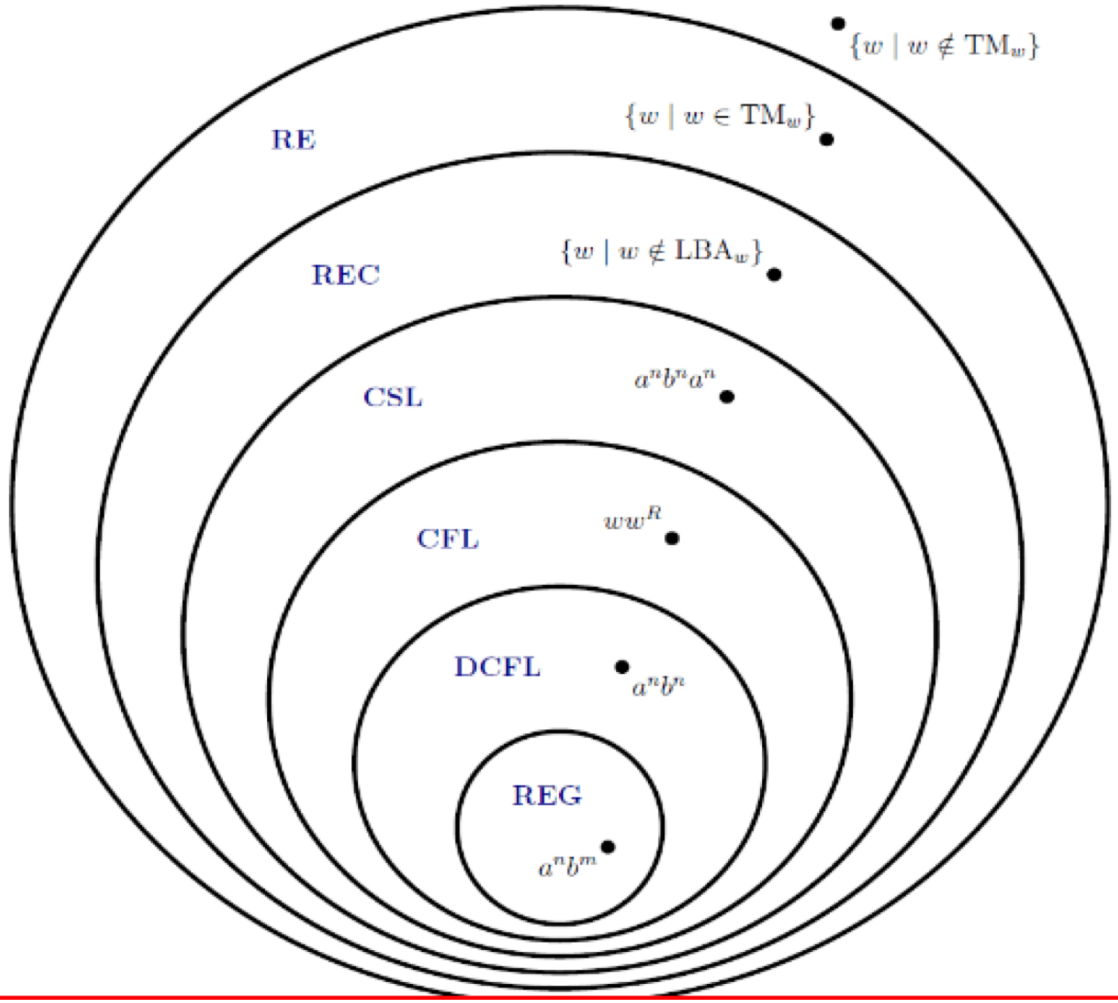
Example.
Show that L is CFL
$L = \{ a^n b^n \mid n \geq 0, n \neq \mathbf{50} \}$

Ans:
Assume
$L_1 = \{ a^n b^n \mid n \geq 0 \}$
$L_2 = \{ a^{50} b^{50} \}$
We know
L_1 is CF
L_2 is RL, so $\overline{L_2}$ is RL

$L = L_1 \cap \overline{L_2}$
CFL ∩ RL is CFL
So L is CFL

Chomsky Hierarchy

As a concise reference to the entire subject, we put some of the formal languages in the levels of the Chomsky hierarchy and summarize of the closure properties of the levels.



https://cs.fit.edu/~ryan/cse4083/

Thank You!

Tutorial VI

innovate achieve lead

1)Consider the following languages over the alphabet {0,1} and the strings $w \in \{0,1\}^*$. Construct a context-free grammar that generates strings within the specified language.

a)L_1={w contains at least three 1s}

b)L_2={$w=w^R$ and |w| is even}

c)L_3={w | the length of w is odd and the middle symbol is 0}

2) Construct a pushdown automata that recognize the given languages and represent each PDA with 6-tuple specification and state transition diagram. Consider the following languages over the alphabet {0,1} and the strings $w \in \{0,1\}^*$.

- L_1= {w contains at least three 1s}
- L_2= { $w=w^R$ and |w| is odd }

Tutorial VI

innovate achieve lead

1) Let T = { 0, 1, (,), ∪, ∗ , ∅, e }. We may think of T as the set of symbols used by regular expressions over the alphabet {0, 1}; the only difference is that we use e for symbol ε, to avoid potential confusion in what follows.

a) Your task is to design a CFG G with set of terminals T that generates exactly the regular expressions with alphabet {0, 1}.

b) Using your CFG G, give a derivation and the corresponding parse tree for the string (0 ∪ (10)∗1)∗

2) Consider the following CFG G = (V, Σ, R, S), where V = {S, T, X}, Σ = {a, b}, the start variable is S, and the rules R are S → aT Xb T → XTS | ε X → a | b Convert G to an equivalent PDA