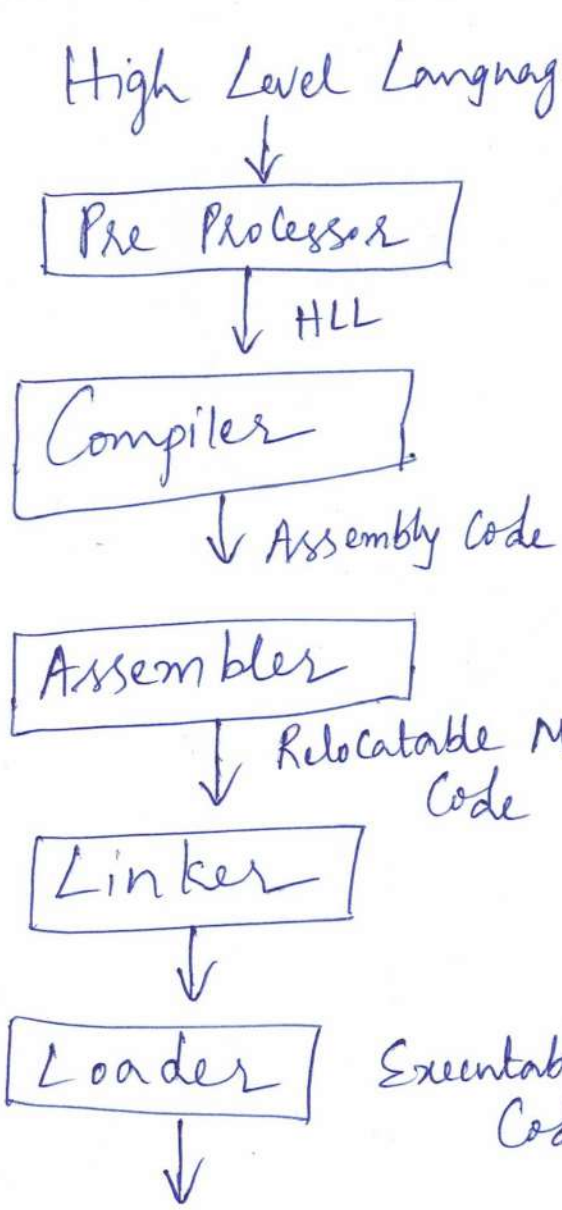
Language Processing System



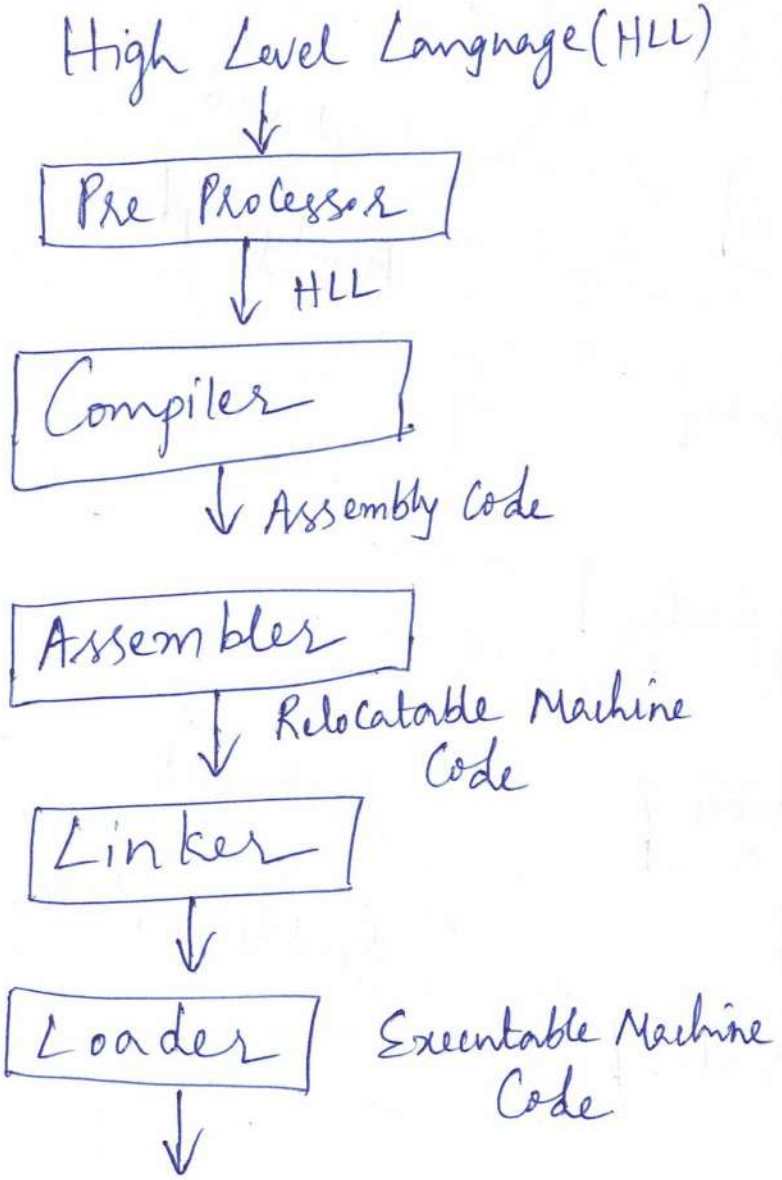
① → Linker :-

② → Loader :-

③ → Cross Compiler :-

④ → Source to Source Compiler :-

Language Processing System



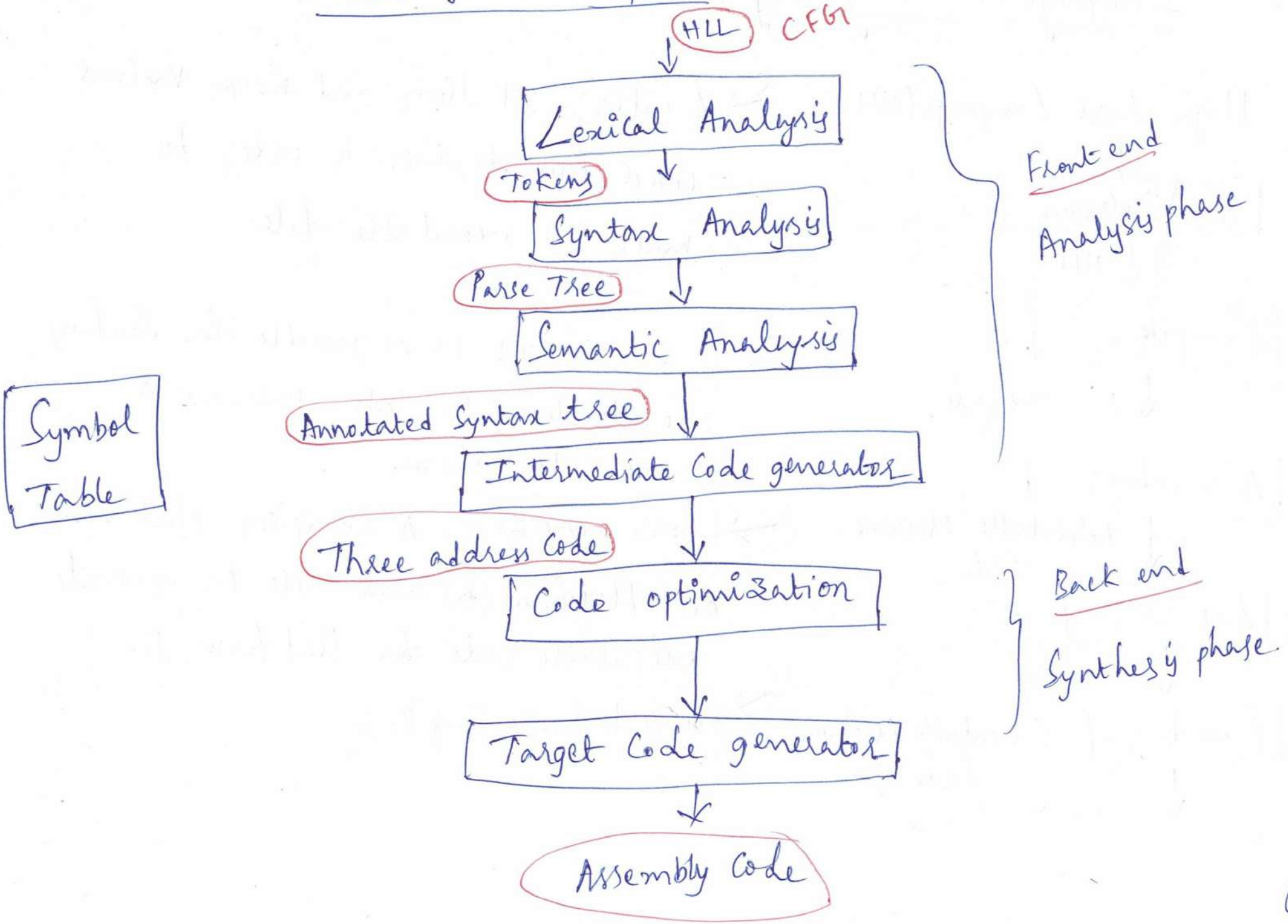
① Linker :- It links and merges various object files together in order to make an executable file.

② Loader :- It is responsible for loading executable files into memory & execute them.

③ Cross Compiler :- A Compiler that runs on Platform (A) and able to generate executable Code for Platform B.

④ Source to Source Compiler :-

Phases of a Compiler



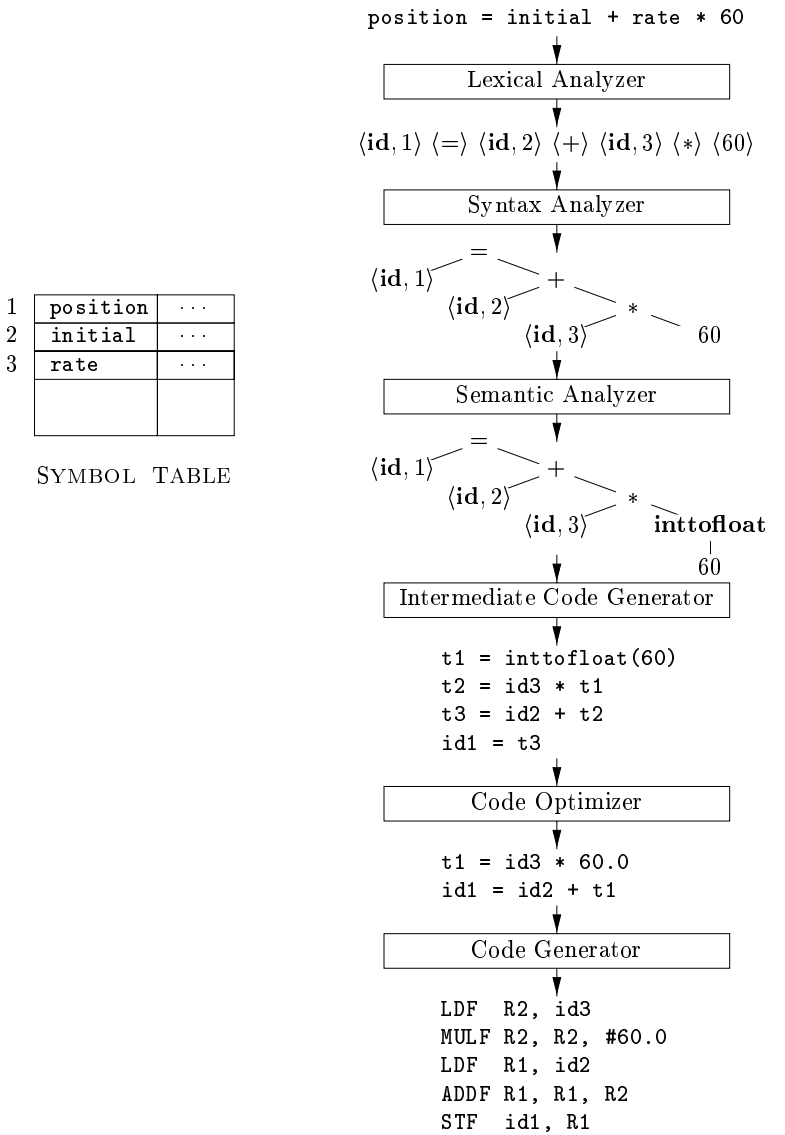


Figure 1.7: Translation of an assignment statement

Deterministic Finite State Machine (DFA)

→ 5- tuple $(Q, \Sigma, \delta, q_0, F)$

→ (1) A finite set of states - Q

→ (2) A " " of input symbols - Σ

→ (3) A transition function - $\delta : Q \times \Sigma \to Q$ Important

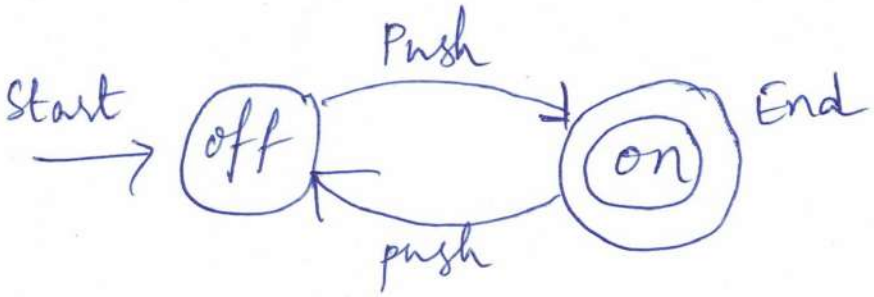
→ (4) An initial or start state - $q_0 \in Q$

→ (5) A set of accept states - $F \subseteq Q$

$\Sigma = \{0, 1\}$ — Binary Σ = Set of all ASCII values

$\Sigma = \{a, b, .. z\}$ - Alphabets

Example of A finite Automate modeling on/off switch.



$Q = \{off, on\}$

$\Sigma = \{push\}$

$q_0 = \{off\}$

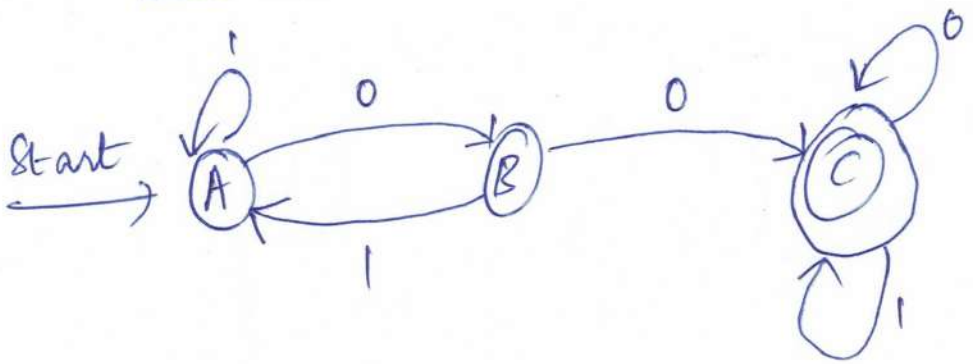
$F = \{on\}$

$\delta(on, push) = off$

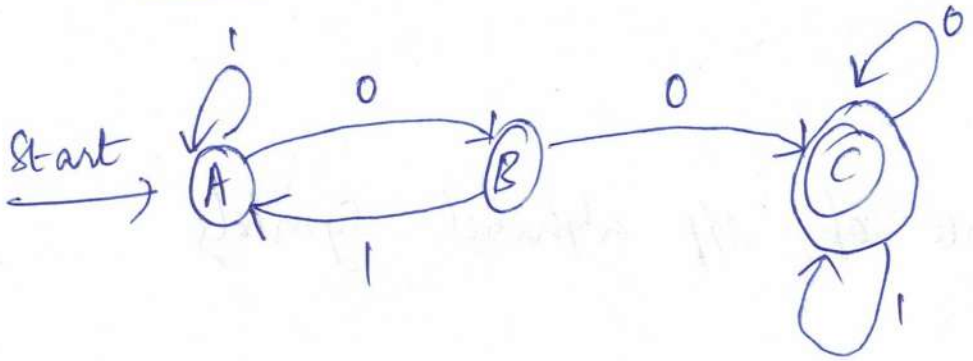
$\delta(off, push) = on$

δ	push
on	~~off~~ off
off	on

Another example of DFA



Another example of DFA



$Q = \{A, B, C\}$

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

$q_0 = \{A\}$

$F = \{C\}$

δ	0	1
A	B	A
B	C	A
C	C	C

State transition table

Alphabet

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

Strings/Word:- A finite sequence of i/p alphabet symbols

Empty String:- $\epsilon \rightarrow$ Epsilon
Zero occurence

$$w = w\epsilon = \epsilon w$$

$$001 = 001\epsilon = \epsilon 001 = 0\epsilon 01$$

$x = a_1 a_2 \ldots a_n, \quad a_i \in \Sigma$

$y = b_1 b_2 \ldots b_m, \quad b_i \in \Sigma$

$|xy| = |a_1 a_2 \ldots a_n b_1 b_2 \ldots b_m| = n + m$

} Concatenation.

Power of Alphabet

Σ = Input alphabet = $\{a, b\}$

$\Sigma^k = \{ (x_1, x_2, \ldots x_k) \mid x_i \in \Sigma \}$

$\Sigma^0 = \epsilon$

$\Sigma^1 = \Sigma = \{a, b\}$

$\Sigma^2 = \{aa, ab, ba, bb\}$

$\Sigma^3 = \{aaa, aab, aba, abb \ldots\}$

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$

$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \ldots$

$\Sigma^* = \Sigma^+ \cup \{\epsilon\}$.

Language $L \subseteq \Sigma^*$

→ Any subset of Σ^* is called a language over Σ.

Example 1:

L = language of strings consisting n 0's followed by n 1's, $n \geq 0$

$= \{\epsilon, 01, 0011, 000111 - - -\}$

$= \{0^n 1^n / n \geq 0\} \subseteq \Sigma^*$

Example 2:- L = set of all strings having equal 0's and 1's

$= \{\epsilon, 01, 10, 0011, 0101, 0110, 1100, - -\}$.

Language $L \subseteq \Sigma^*$

May Contain an Infinite number of strings drawn from a finite i/p alphabet set Σ.

Problem:- Decision problem

$\Sigma \rightarrow$ An alphabet

$L \subseteq \Sigma^*$

Given a string $w \in \Sigma^*$

decide whether $w \in L$ or not ??

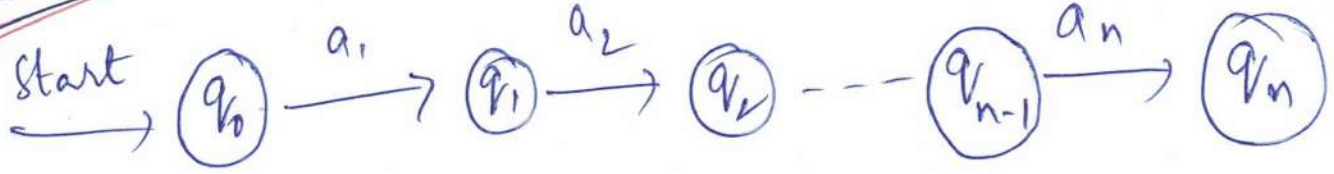
How a DFA processing Works

Given $A = \{Q, \Sigma, \delta, q_0, F\}$

$\delta : Q \times \Sigma \to Q$

$W = a_1 a_2 \cdots a_n$, $a_i \in \Sigma$

Processing

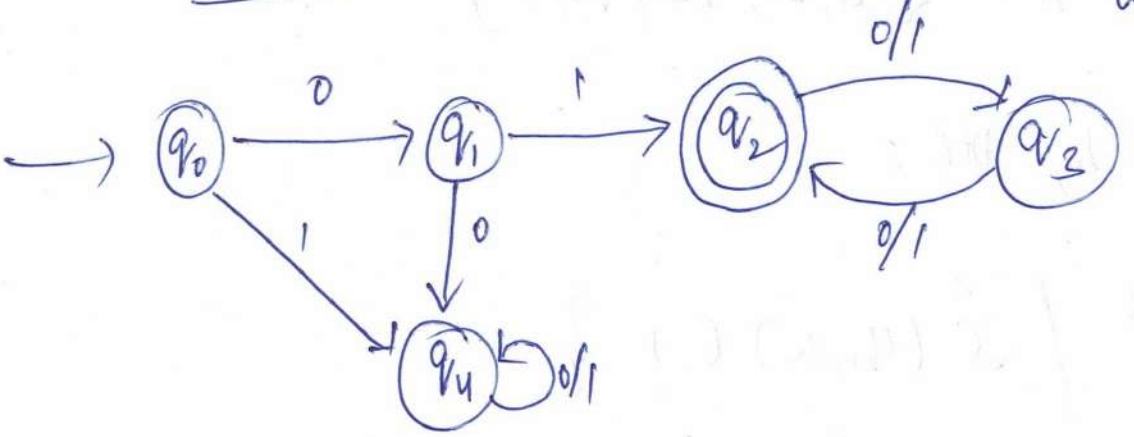


If $q_n \in F \iff W$ is accepted by DFA.

$q_n \notin F \iff W$ is Rejected.

Example

$Q = \{q_0, q_1, q_2, q_3, q_4\}$ $F = \{q_2\}$



δ	0	1
q_0	q_1	q_4
q_1	q_4	q_2
q_2	q_3	q_3
q_3	q_2	q_2
q_4	q_4	q_4

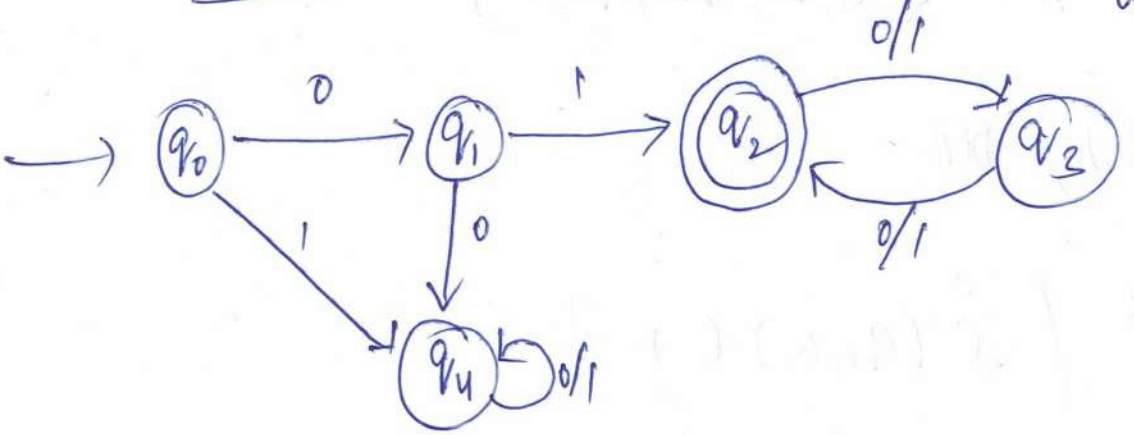
$W = 01110$

$\hat{\delta}(q_0, W) \in F$??

W accepted or not ??

Example

$Q = \{q_0, q_1, q_2, q_3, q_4\}$ $F = \{q_2\}$

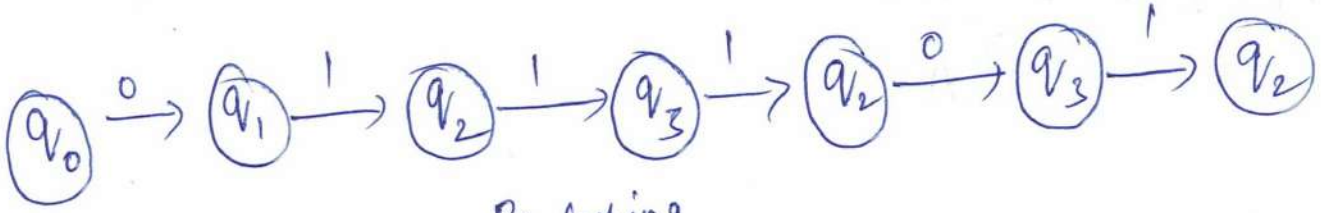


δ	0	1
q_0	q_1	q_4
q_1	q_4	q_2
q_2	q_3	q_3
q_3	q_2	q_2
q_4	q_4	q_4

$W = 01110$

$\hat{\delta}(q_0, W) \in F$??

W accepted or not ??



Processing.

This $q_2 \in F$

$\therefore$ W is accepted by DFA.

$\hat{\delta}(q_0, w) \in F$ $\quad A = \{Q, \Sigma, \delta, q_0, F\}$

→ w is accepted by DFA

$L(A) = \{ w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F \}$

⇒ Language of the DFA

$\Sigma \rightarrow$ I/p alphabet

$L \subseteq \Sigma^*$

→ Regular Language

If $L = L(A)$

DFA.

Building DFA

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

$L = \{w \in \Sigma^* / w \text{ is binary string with odd number of 1's}\}$

$1 \in L$, $01 \in L$, $001 \in L$

$111 \in L$, $01001 \in L$ -- etc.

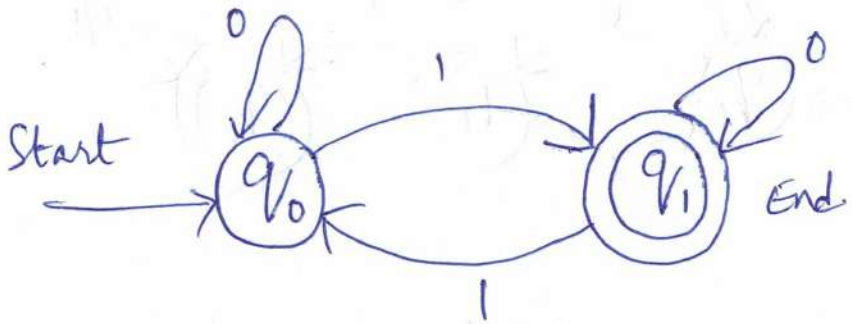
Building DFA

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

$L = \{w \in \Sigma^* / w \text{ is binary string with odd number of 1's}\}$

$1 \in L$, $01 \in L$, $001 \in L$

$111 \in L$, $01001 \in L$ -- etc.



Trial and error Method.

$L = L(A)$, A is a DFA

$\therefore$ L is a Regular Language.

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

$L = \{w \mid w$ is a binary string Containing '00' as substring$\}$

$= \{x00y \mid x, y \in \Sigma^*\}$

$00 \in L$, $0010 \in L$, $100 \in L$, $001 \in L$ -- etc

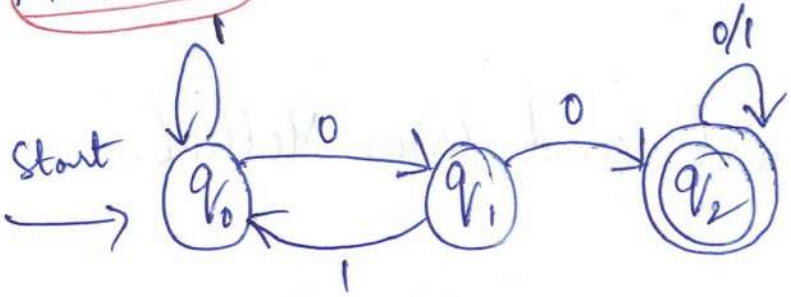
$\Sigma = \{0, 1\}$

$L = \{w \mid w$ is a binary string Containing '00' as substring$\}$

$= \{x00y \mid x, y \in \Sigma^*\}$

$00 \in L$, $0010 \in L$, $100 \in L$, $001 \in L$ -- etc

First DFA



DFA

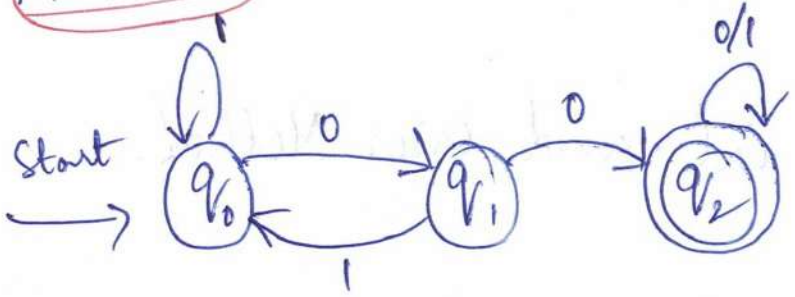
$\Sigma = \{0, 1\}$

$L = \{w \mid w$ is a binary string Containing '00' as substring$\}$

$= \{x00y \mid x, y \in \Sigma^*\}$

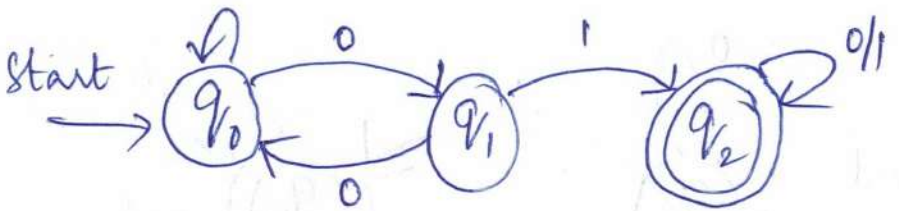
$00 \in L$, $0010 \in L$, $100 \in L$, $001 \in L$ -- etc

First DFA



DFA

Second DFA



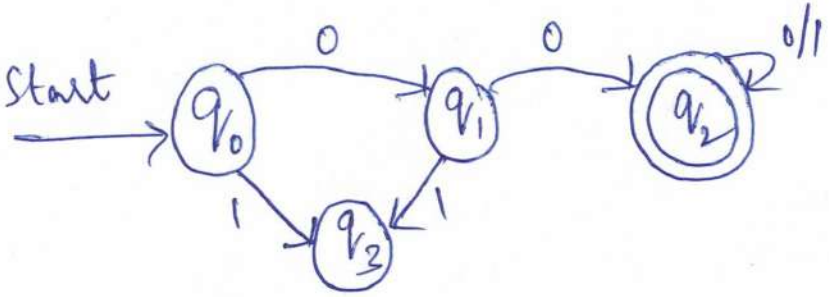
DFA

For both DFA's

$L = L(A)$ $\therefore L$ is Regular Language.

Example ① $\Sigma = \{0, 1\}$

DFA that accepts all binary strings with starting '00'.



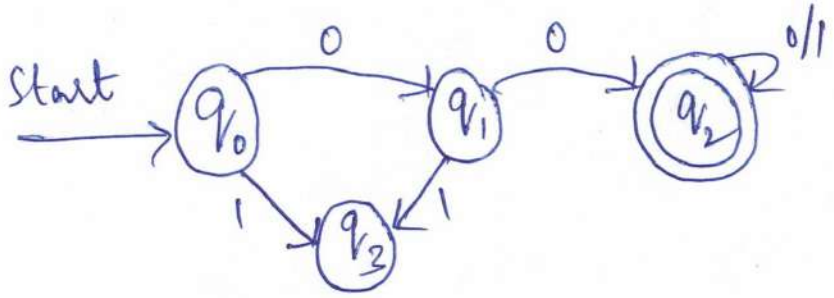
Example ② Example ① + even length

Accepts all binary strings starting '00' and length is even.

Example ① $\Sigma = \{0, 1\}$

DFA that accepts all binary strings with starting '00'.

$00 \in L$, $001 \in L$, $000 \in L$ -- etc.

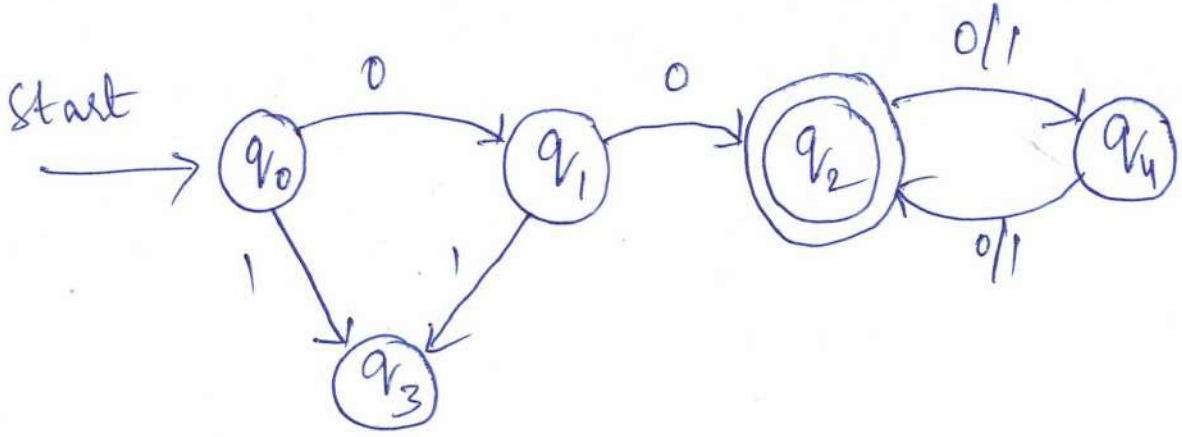


Example ② Example ① + even length

Accepts all binary strings starting '00' and length is even.

$0000 \in L$, $0001 \notin L$

$0010 \in L$ -- etc.



DFA that accepts all binary strings where 0's and 1's are alternate.

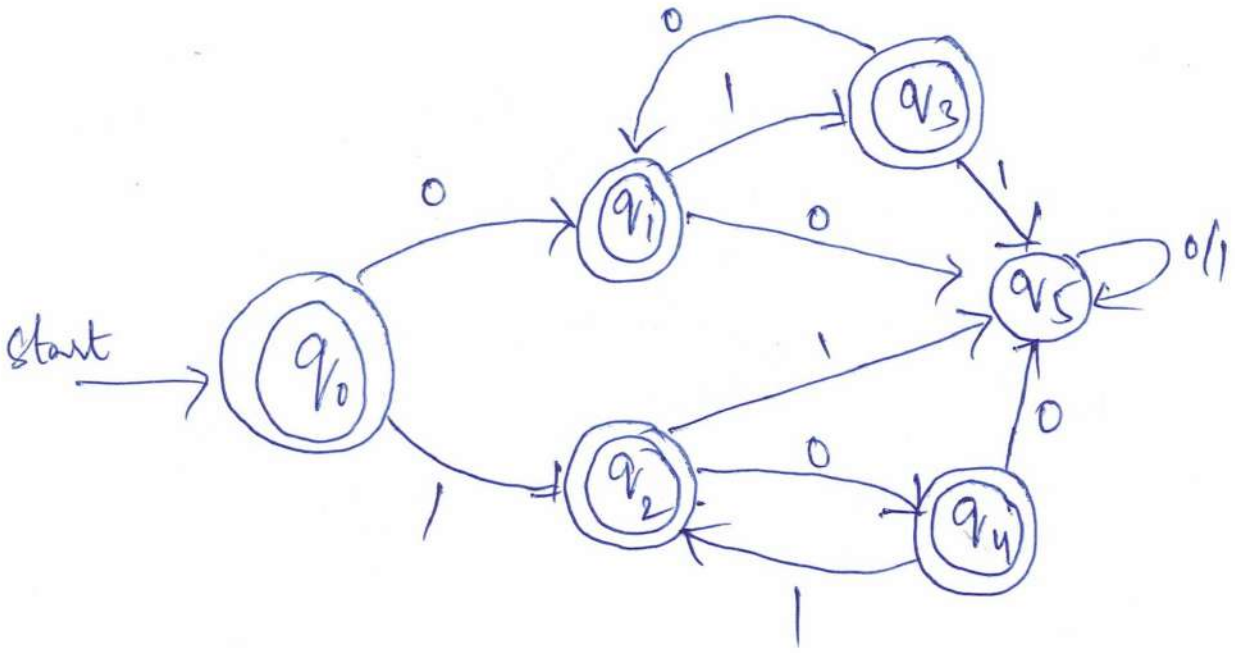
$\epsilon \in L$, $0 \in L$, $1 \in L$, $01 \in L$, $10 \in L$, $010 \in L$, $101 \in L$ --- etc.

$L = \{\epsilon, 0, 1, 01, 10, 010, 101, 0101, 1010, --\}$.

DFA that accepts all binary strings where 0's and 1's are alternate.

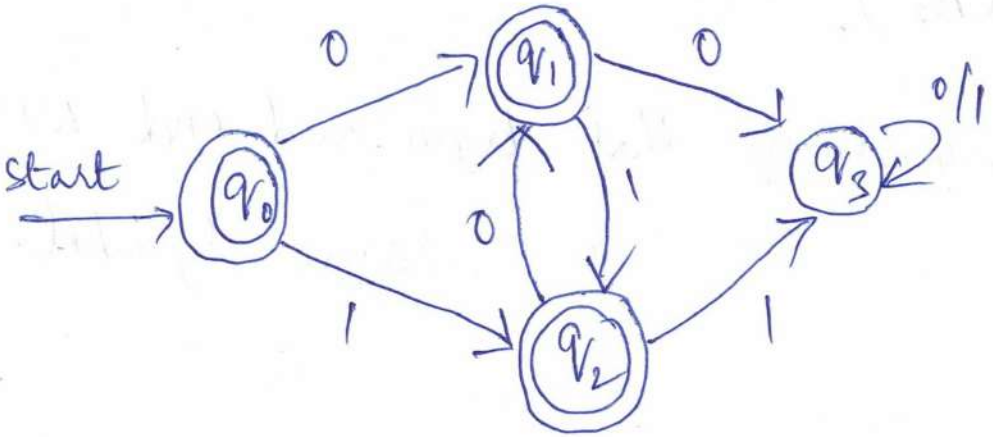
$\epsilon \in L$, $0 \in L$, $1 \in L$, $01 \in L$, $10 \in L$, $010 \in L$, $101 \in L$ --- etc.

$L = \{\epsilon, 0, 1, 01, 10, 010, 101, 0101, 1010, --\}$.



$L = \{\epsilon, 0, 1, 01, 10, 010, -- \}$

DFA that accept all binary strings where 0's and 1's are alternate.



$A = (Q, \Sigma, \delta, q_0, F)$

$Q = \{q_0, q_1, q_2, q_3\}$

$F = \{q_0, q_1, q_2\}$

δ	0	1
q_0	q_1	q_2
q_1	q_3	q_2
q_2	q_1	q_3
q_3	q_3	q_3

$\delta : Q \times \Sigma \to Q$

$\forall q \in Q$

$\forall a \in \Sigma$

$(q) \xrightarrow{a} (p)$

Note:-

1) Given a DFA $\Rightarrow$ Unique $L(A)$

2) Given a Regular language L $\Rightarrow$ $A_1, A_2, \ldots A_n$

$$L(A_1) = L = L(A_2)$$

Example: DFA that accept all binary strings that begin and end with Same symbol.

Note:-

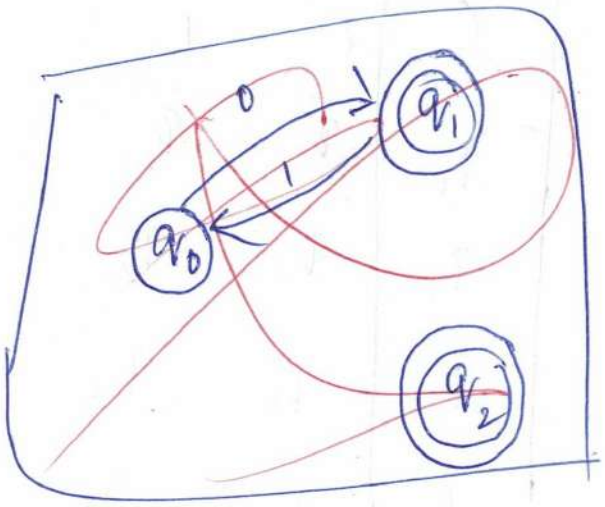
1) Given a DFA ⇒ Unique $L(A)$

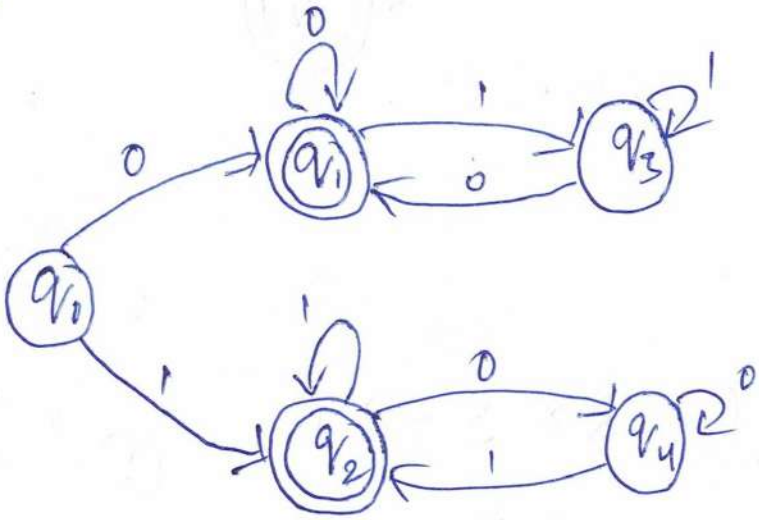
2) Given a Regular language L ⇒ $A_1, A_2, \ldots A_n$

$$L(A_1) = L = L(A_2)$$

Example: DFA that accept all binary strings that begin and end with same symbol.

$$L = \{axa / a \in \Sigma, x \in \Sigma^*\}$$





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

Regular ? $\Rightarrow$ DFA ?

Ans: No

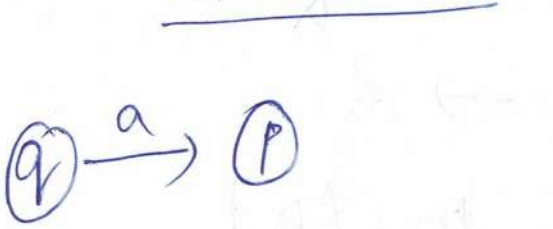
$L = \{0^n 1^n / n \geq 0\}$

Regular ? $\Rightarrow$ DFA ?

Ans: NO

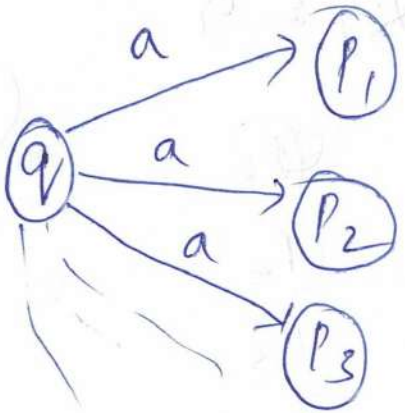
NFA $\rightarrow$ Non deterministic Finite Automata

$(Q, \Sigma, \delta, q_0, F)$



$\delta(q, a) = p$

DFA: $\delta : Q \times \Sigma \rightarrow Q$



$\delta(q, a) = \{p_1, p_2, p_3, \ldots p_k\} \in Q.$

NFA $= \delta : Q \times \Sigma \rightarrow 2^Q = \text{Pow}(Q)$

NFA

$$N = \{Q, \Sigma, \delta, q_0, F\}$$

$Q \rightarrow$ Set of possible states

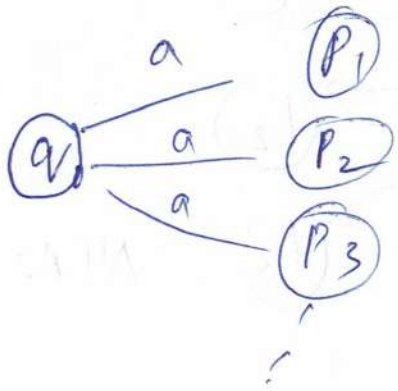
Finite

$\Sigma \rightarrow$ Input alphabet

$q_0 \rightarrow$ Starting state

$F \rightarrow$ Final state $F \subseteq Q$.

$$\delta(q, a) = \{p_1, p_2, \ldots p_k\} \subseteq Q$$



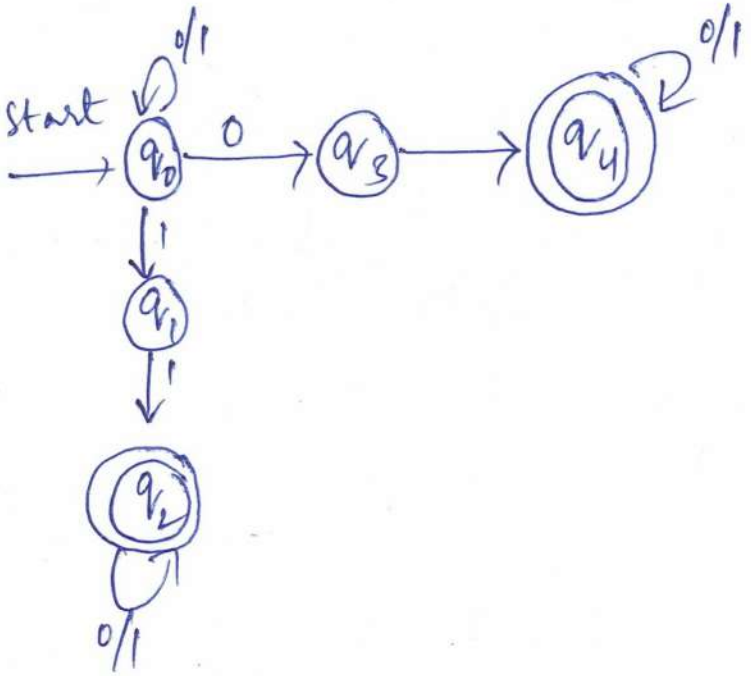
$$\delta : Q \times \Sigma \rightarrow 2^Q$$

2^Q or $POW(Q)$

= Set of all possible subsets of Q.

P

$\Sigma = \{0, 1\}$ Example of NFA

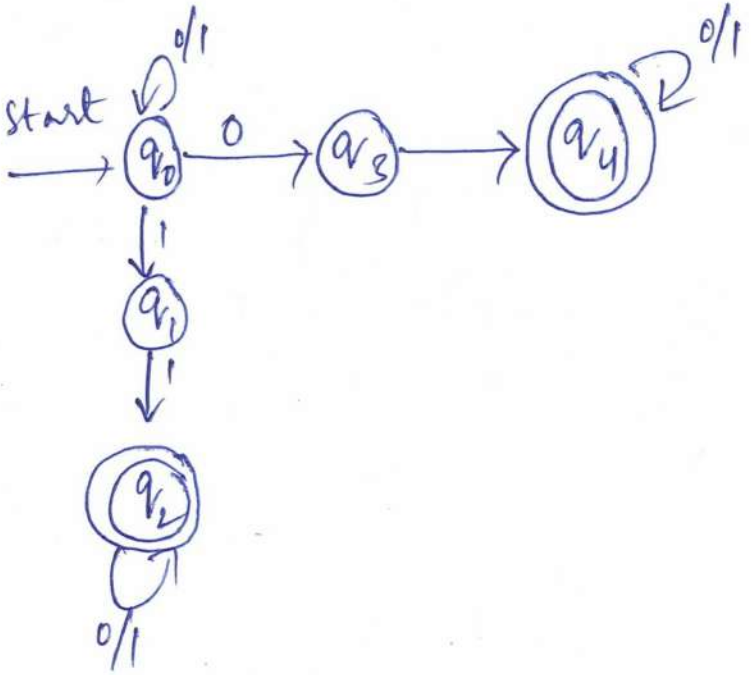


$N = (Q, \Sigma, \delta, q_0, F)$

$Q = \{q_0, q_1, q_2, q_3, q_4\}$

$F = \{q_2, q_4\}$

$\Sigma = \{0, 1\}$ Example of NFA



$N = (Q, \Sigma, \delta, q_0, F)$

$Q = \{q_0, q_1, q_2, q_3, q_4\}$

$F = \{q_2, q_4\}$

δ	0	1
q_0	$\{q_0, q_3\}$ 1	$\{q_0, q_1\}$ 1
q_1	$\emptyset$ 2	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	$\emptyset$ 2
q_4	$\{q_4\}$	$\{q_4\}$

① Multiple

② Empty.

Building NFA

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

$L \subseteq \Sigma^*$

$A \rightarrow DFA$

$N \rightarrow NFA \qquad L(A) = L$

$L(N) = L$

* Every DFA is NFA.

Building NFA

Example:- $\Sigma = \{0, 1\}$

{ NFA accepting all binary strings that end with pattern 101

$$L = \{ x\,101 \mid x \in \Sigma^* \}$$

$$= \{ 101, 0101, 1101, 00101, \text{---} \}$$

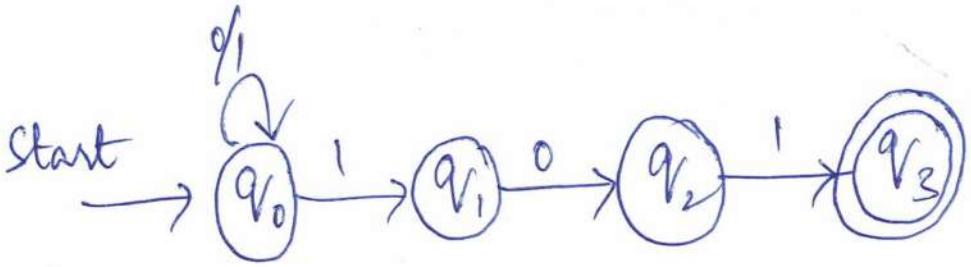
Building NFA

Example:- $\Sigma = \{0, 1\}$

{ NFA accepting all binary strings that end with pattern 101

$$L = \{ x\,101 \mid x \in \Sigma^* \}$$

$$= \{ 101, 0101, 1101, 00101, \cdots \}$$



Building NFA

Example:- $\Sigma = \{0, 1\}$

Two patterns simultaneously

$$L = 0^* + (01)^*$$

$$L = \{ 0, 00, 000, \text{---} \}$$
$$\{ 01, 0101, 010101, \text{---} \}$$

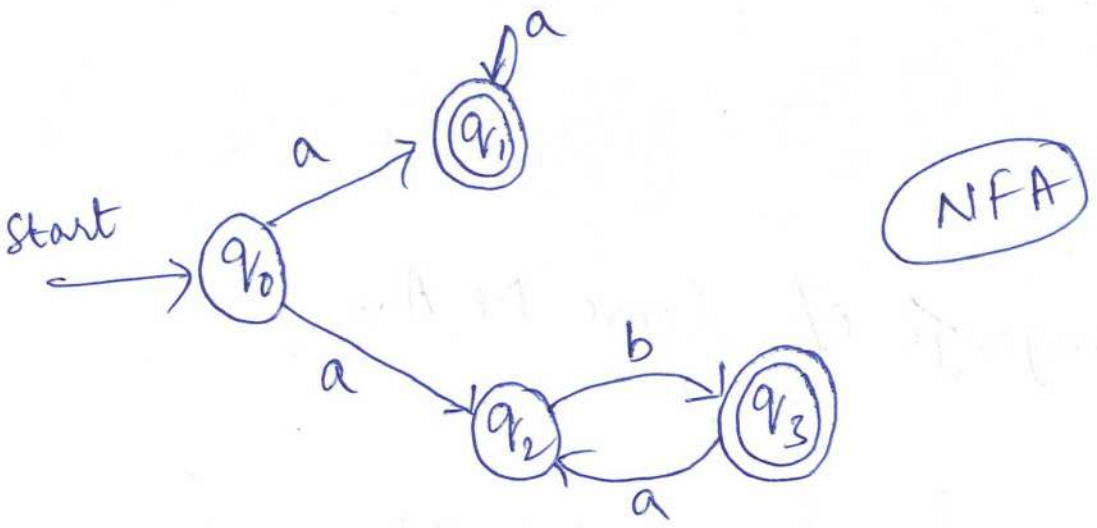
Building NFA

Example:- $\Sigma = \{0, 1\}$

Two patterns simultaneously

$$L = 0^* + (01)^*$$

$$L = \{ 0, 00, 000, \text{---} \}$$
$$\{ 01, 0101, 010101, \text{---} \}$$



NFA

The equivalence of DFA and NFA

- DFA Can be treated as NFA

↓

$(Q, \Sigma, \delta, q_0, F)$

↓

$(Q, \Sigma, \delta, q_0, F)$

$\delta : Q \times \Sigma \to Q$

$\delta : Q \times \Sigma \to 2^Q$

$\delta(q, a) = p = \{p\} \in 2^Q$

→ Language of a NFA

$L(N)$ is also a language of some DFA.

↓

Regular Language.

NFA accepts only Regular languages.

NFA to DFA

→ Given NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$

Design :- $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ Such that $L(N) = L(D)$.

$Q_N = \{q_0, q_1, -- q_n\}$

$$\left\{ \begin{array}{l} \{q_0\}\ \{q_1\} - - \{q_n\} \\ \{q_0, q_1\}\ \{q_0, q_2\} - - - \end{array} \right\}$$

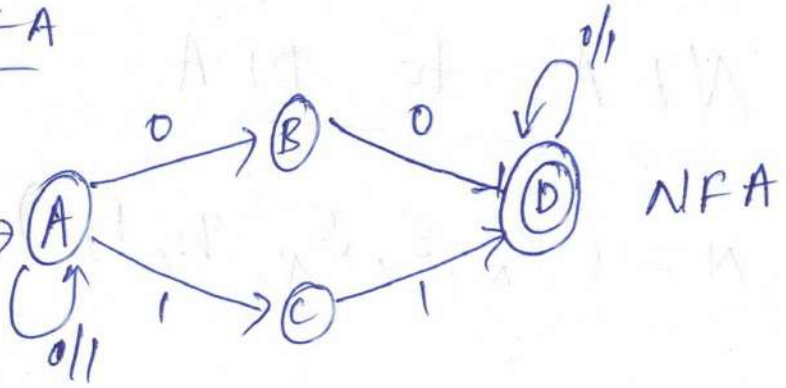
$Q_D = 2^{Q_N}$

$|Q_D| = 2^n$

$Q_D \subseteq 2^{Q_N}$

NFA to DFA

Example:- $Q_N = \{A, B, C, D\}$

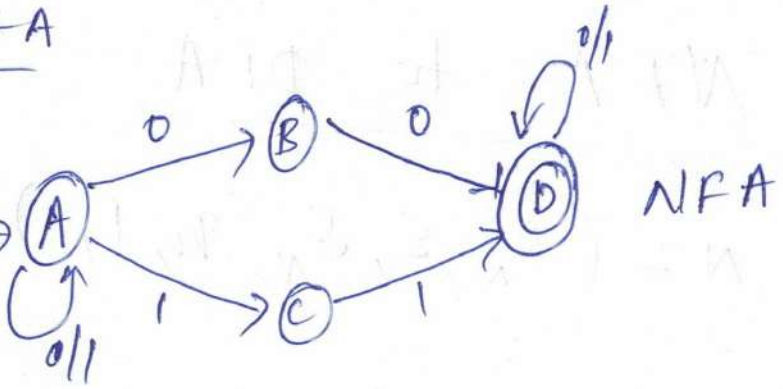


DFA

	0	1
→ {A}	{A, B}	{A, C}

NFA to DFA

Example:- $Q_N = \{A, B, C, D\}$

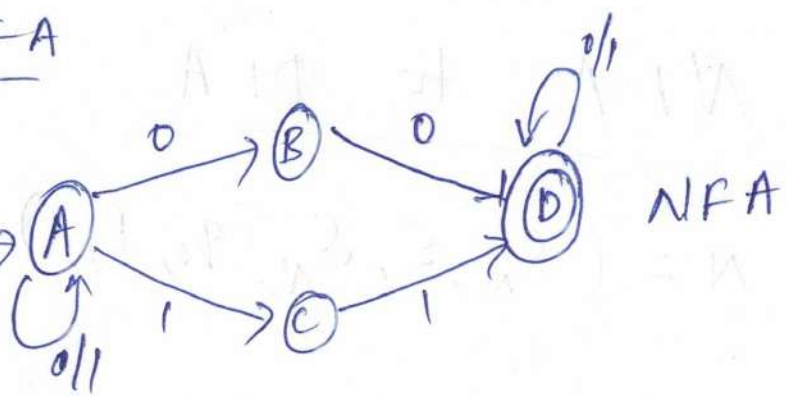


DFA

	0	1
→ {A}	{A, B}	{A, C}
{A, B}	{A, B, D}	{A, C}

NFA to DFA

Example:- $Q_N = \{A, B, C, D\}$

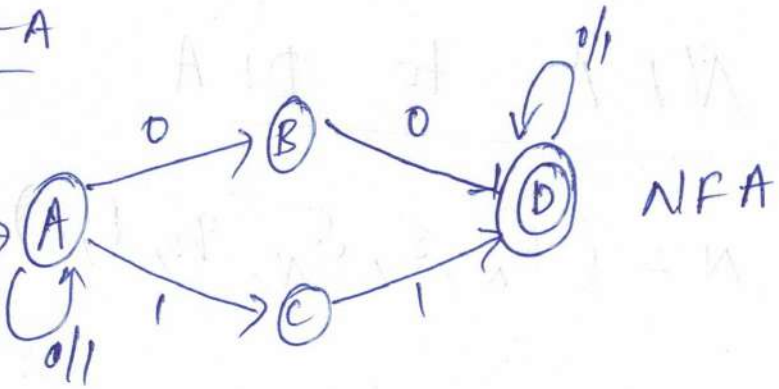


DFA

	0	1
→ {A}	{A, B}	{A, C}
{A, B}	{A, B, D}	{A, C}
{A, C}	{A, B}	{A, C, D}

NFA to DFA

Example:- $Q_N = \{A, B, C, D\}$

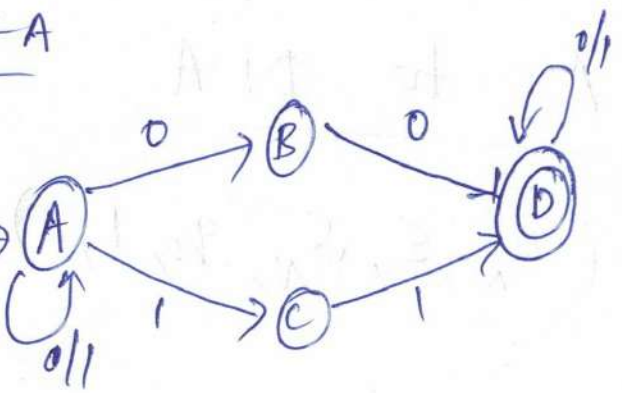


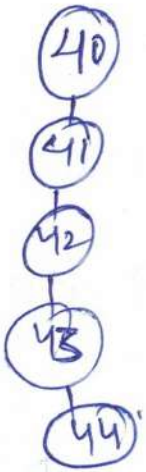
DFA

	0	1
→ {A}	{A, B}	{A, C}
{A, B}	{A, B, D}	{A, C}
{A, C}	{A, B}	{A, C, D}
*{A, B, D}	{A, B, D}	{A, C, D}
{A, C, D}	{A, B, D}	{A, C, D}

NFA to DFA

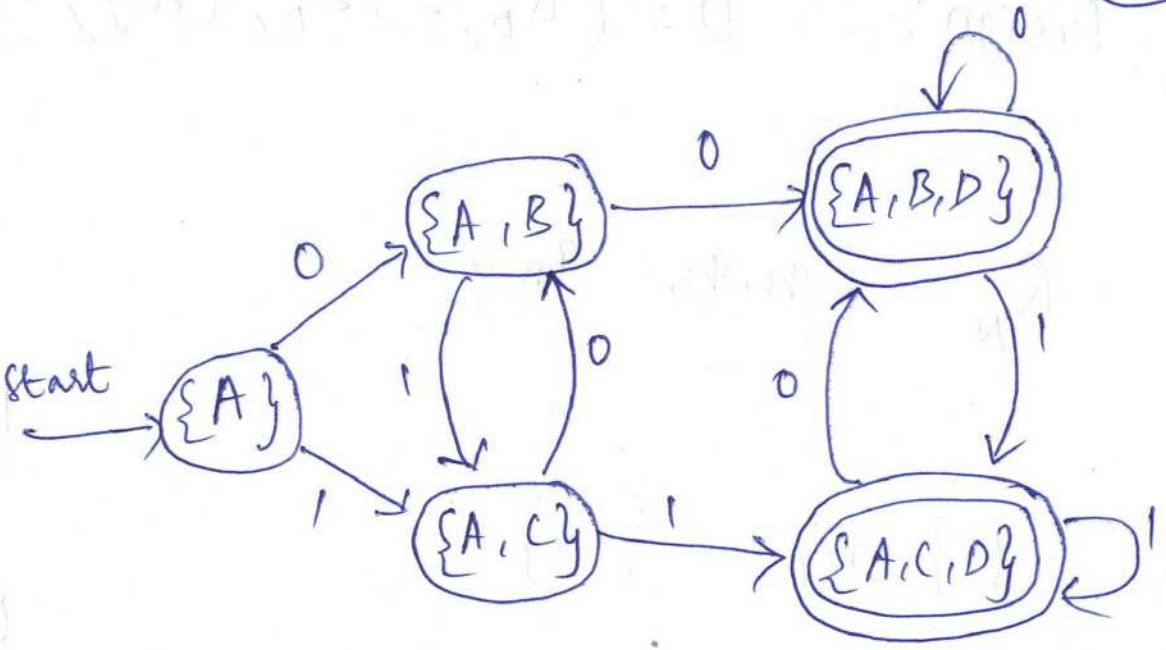
Example:- $Q_N = \{A, B, C, D\}$





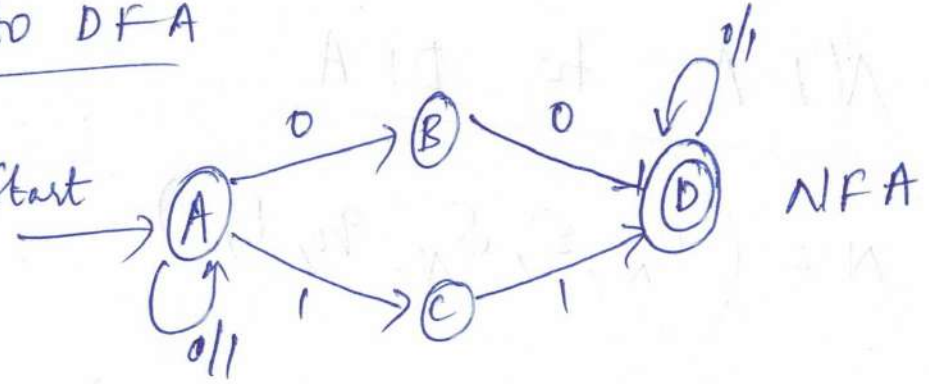
DFA

	0	1
→ {A}	{A, B}	{A, C}
{A, B}	{A, B, D}	{A, C}
{A, C}	{A, B}	{A, C, D}
*{A, B, D}	{A, B, D}	{A, C, D}
*{A, C, D}	{A, B, D}	{A, C, D}



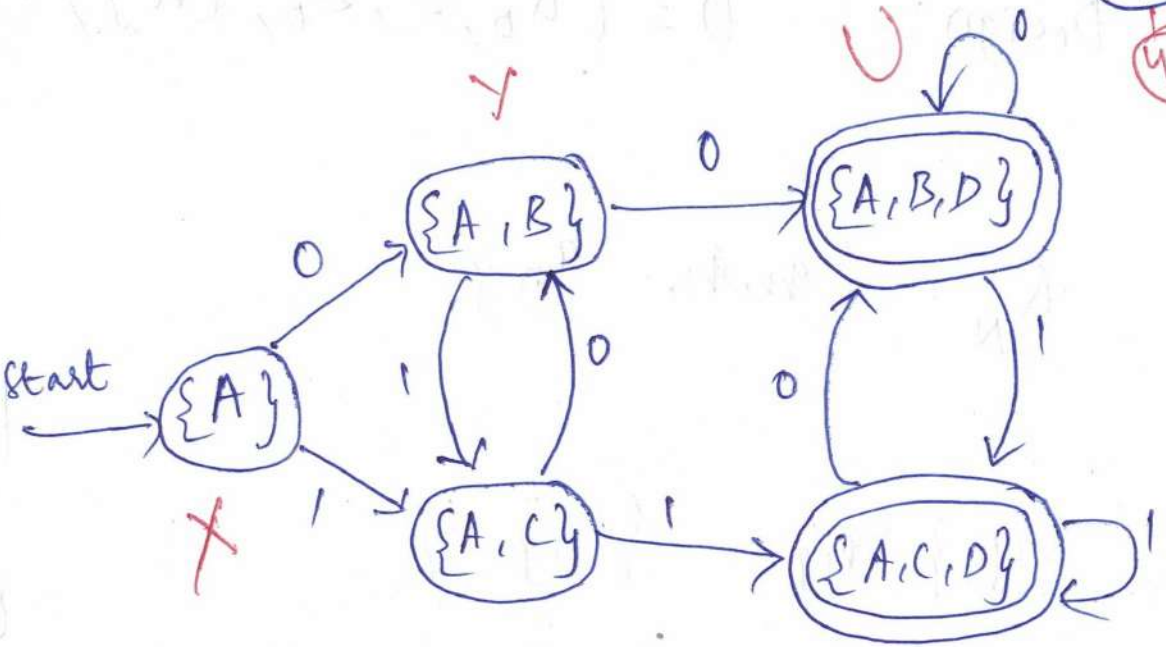
NFA to DFA

Example:- $Q_N = \{A, B, C, D\}$

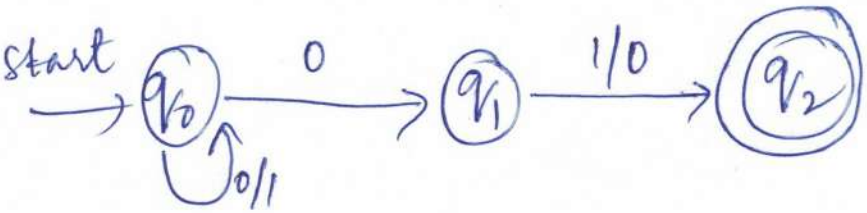


DFA

	0	1
→ {A}	{A, B}	{A, C}
{A, B}	{A, B, D}	{A, C}
{A, C}	{A, B}	{A, C, D}
*{A, B, D}	{A, B, D}	{A, C, D}
*{A, C, D}	{A, B, D}	{A, C, D}



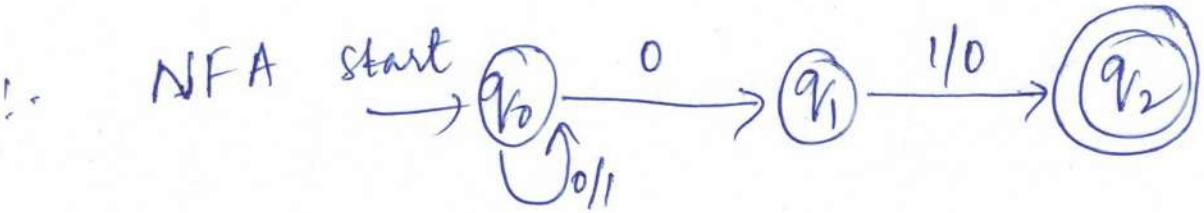
Example:- NFA



$$\delta_D(S, a) = \bigcup_{p \in S} \delta_N(p, a)$$

δ_D	0	1
→ $\{q_0\}$	$\{q_0\}$	$\{q_0, q_1\}$

Example:- NFA

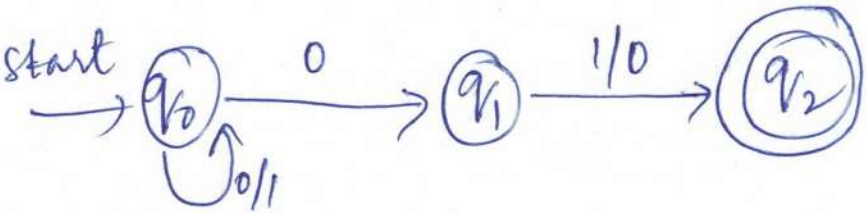


NFA to DFA

$$\delta_D(S, a) = \bigcup_{P \in S} \delta_N(P, a)$$

δ_D	0	1
$\rightarrow\{q_0\}$	$\{q_0\}$	$\{q_0, q_1\}$
$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$

Example:- NFA

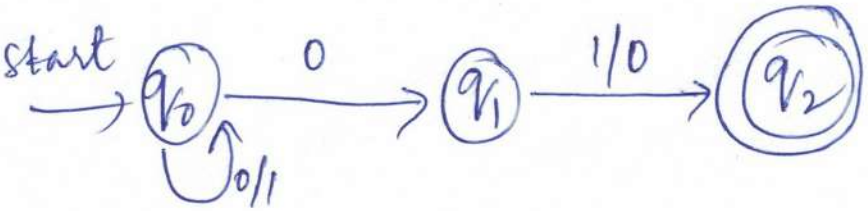


NFA to DFA

$$\delta_D(S, a) = \bigcup_{P \in S} \delta_N(P, a)$$

δ_D	0	1
$\rightarrow\{q_0\}$	$\{q_0\}$	$\{q_0, q_1\}$
$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$
$*\{q_0, q_2\}$	$\{q_0\}$	$\{q_0, q_1\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$

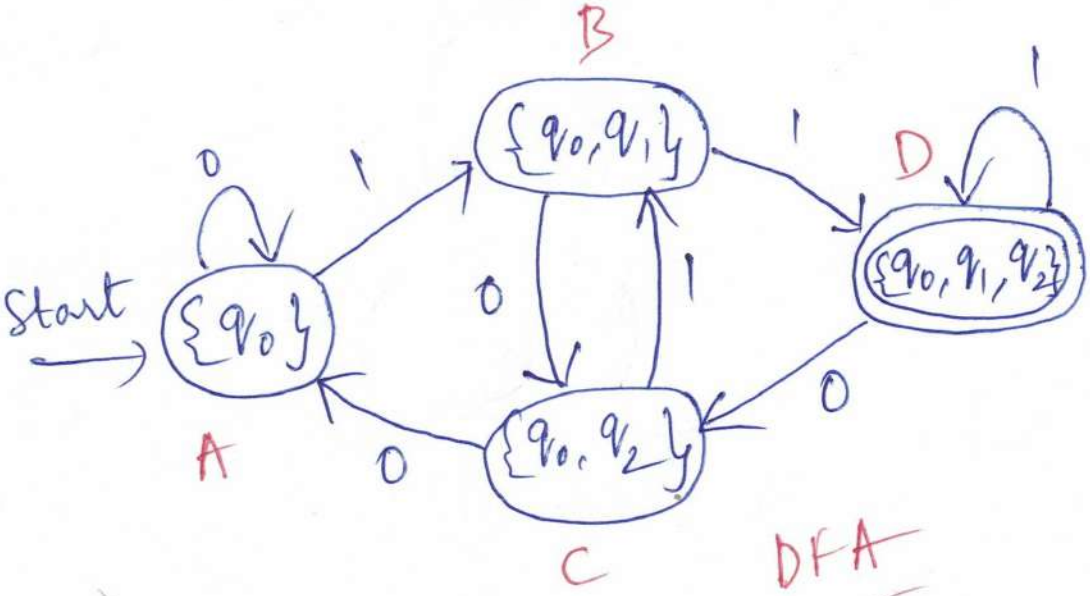
Example:- NFA



NFA to DFA

$$\delta_D(S,a) = \bigcup_{p \in S} \delta_N(p,a)$$

δ_D	0	1
$\rightarrow\{q_0\}$	$\{q_0\}$	$\{q_0, q_1\}$
$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$
$*\{q_0, q_2\}$	$\{q_0\}$	$\{q_0, q_1\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$



DFA ✓

NFA ↓ DFA ↓

$L(N) = L(D)$

Finite Automata with ϵ-moves.

(ϵ-NFA)

$(Q, \Sigma, \delta, q_0, F)$

Transition

$\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow Pow(Q)$

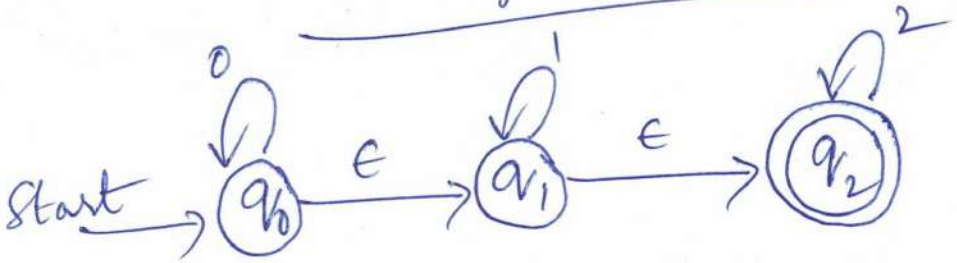
$Q \rightarrow$ Set of all state

$\Sigma \rightarrow$ alphabets

$q_0 \rightarrow$ Initial

$F \rightarrow$ Final / acceptance.

Example of ϵ-NFA



$Q = \{q_0, q_1, q_2\}$

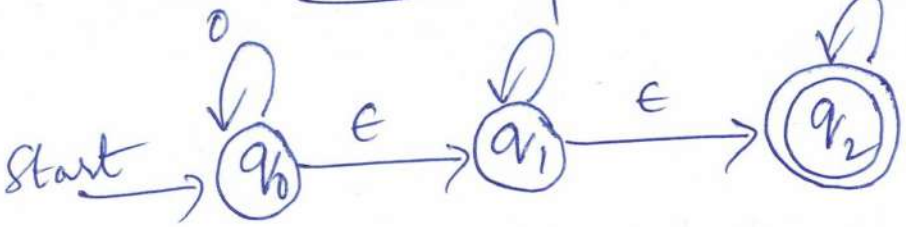
$\Sigma = \{0, 1, 2\}$

$F = \{q_2\}$

Transition function (δ)

State	0	1	2	ϵ
q_0	$\{q_0\}$	ϕ	ϕ	$\{q_1\}$
q_1	ϕ	$\{q_1\}$	ϕ	$\{q_2\}$
q_2	ϕ	ϕ	$\{q_2\}$	ϕ

Example of ϵ-NFA



$Q = \{q_0, q_1, q_2\}$

$\Sigma = \{0, 1, 2\}$

$F = \{q_2\}$

Transition function (δ)

State	0	1	2	ϵ
q_0	$\{q_0\}$	ϕ	ϕ	$\{q_1\}$
q_1	ϕ	$\{q_1\}$	ϕ	$\{q_2\}$
q_2	ϕ	ϕ	$\{q_2\}$	ϕ

$L = 0^* 1^* 2^*$

$0^* = \{\epsilon, 0, 00, 000, \dots\}$

$1^* = \{\epsilon, 1, 11, 111, \dots\}$

$2^* = \{\epsilon, 2, 22, 222, \dots\}$

$L = \{\epsilon, 0, 1, 2, \dots\}$

ϵ-NFA to NFA

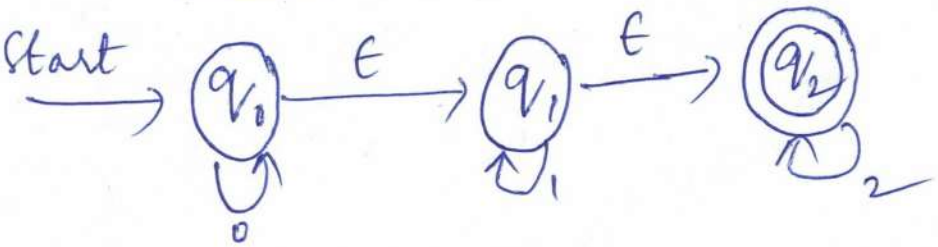
$E = (Q, \Sigma, \delta, q_0, F)$

$N = (Q, \Sigma, \delta', q_0, F')$

$\delta'(q, a) = \hat{\delta}(q, a)$

$$F' = \begin{cases} F \cup \{q_0\}, & \text{if } \epsilon\text{-closure}(q_0) \cap F \neq \phi \\ F & \text{otherwise} \end{cases}$$

ϵ-NFA to NFA

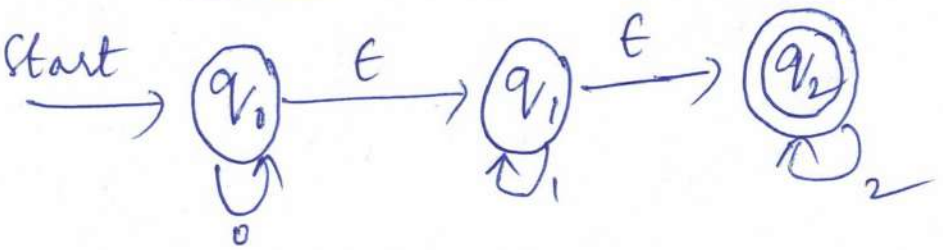
E: 

$L(E) = 0^*1^*2^*$

$N = (Q, \Sigma, \delta', q_0, F')$ $\quad \delta'(q_0, 0) = \hat{\delta}(q_0, \epsilon 0 \epsilon) = ?$

δ'	0	1	2
$\rightarrow q_0$			
q_1			
$* q_2$			

ϵ-NFA to NFA

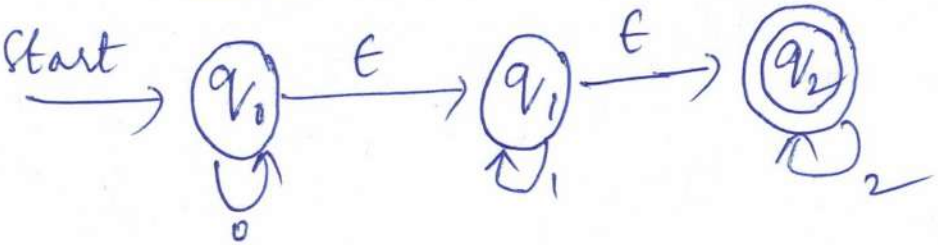
E: 

$L(E) = 0^*1^*2^*$

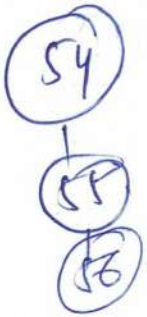
$N = (Q, \Sigma, \delta', q_0, F')$ $\quad \delta'(q_0, 0) = \hat{\delta}(q_0, \epsilon 0 \epsilon) = ? \ (q_0, q_1, q_2)$

δ'	0	1	2
$\rightarrow q_0$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
q_1			
$* q_2$			

ϵ-NFA to NFA

E: 

$L(E) = 0^*1^*2^*$

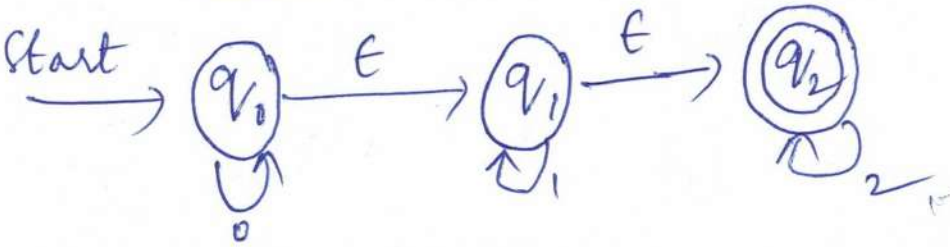


$N = (Q, \Sigma, \delta', q_0, F')$

$\delta'(q_0, 0) = \hat{\delta}(q_0, \epsilon 0 \epsilon) = ? \ (q_0, q_1, q_2)$

δ'	0	1	2
→ q_0	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
q_1	ϕ	$\{q_1, q_2\}$	$\{q_2\}$
* q_2	ϕ	ϕ	$\{q_2\}$

ϵ-NFA to NFA

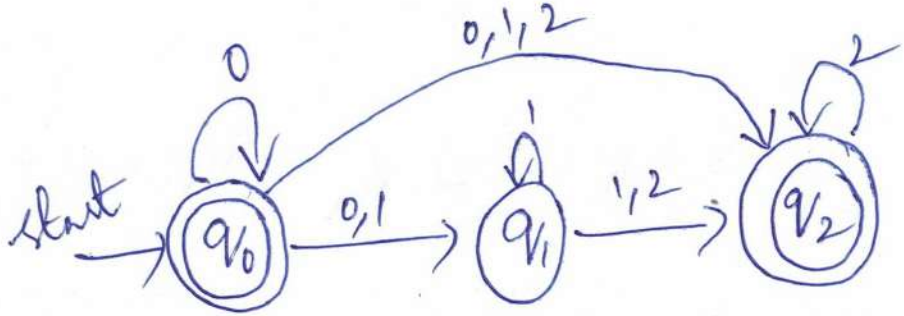
E: 

$L(E) = 0^*1^*2^*$

$N = (Q, \Sigma, \delta', q_0, F')$

$\delta'(q_0, 0) = \hat{\delta}(q_0, \epsilon 0 \epsilon) = ?\ (q_0, q_1, q_2)$

δ'	0	1	2
$\rightarrow q_0$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
q_1	ϕ	$\{q_1, q_2\}$	$\{q_2\}$
$*q_2$	ϕ	ϕ	$\{q_2\}$



NFA

Regular Expression

Σ = Set of alphabets = $\{0,1\}$ or $\{a, b, c \ldots z\}$

Σ^* = Set of all possible strings over Σ

} operations — $(\cdot, +, *)$

$L_1, L_2 \rightarrow$ 2 languages

$L = L_1 \cdot L_2 = \{ xy \mid x \in L_1 \text{ and } y \in L_2 \}$

for eg:- $L_1 = \{10, 1\}$, $L_2 = \{011, 11\}$

Regular Expression

Σ = Set of alphabets = $\{0,1\}$ or $\{a, b, c \ldots z\}$

Σ^* = Set of all possible strings over Σ

} operations — $(\cdot, +, *)$

L_1, L_2 → 2 languages

$L = L_1 \cdot L_2 = \{ xy \mid x \in L_1 \text{ and } y \in L_2 \}$

For eg:- $L_1 = \{10, 1\}$, $L_2 = \{011, 11\}$

$L = L_1 \cdot L_2 = \{10011, 1011, 1011, 111\}$.

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

↓

Kleene closure

$$L^+ = L^* / \{\epsilon\}$$

$$= \bigcup_{i=1}^{\infty} L^i$$

$$= L \cup L^2 \cup L^3$$

↘ positive closure

Eg:- $L = \{10, 11\}$

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

$L^0 = \{\epsilon\}$, $L^1 = L = \{10, 11\}$, $L^2 = L \cdot L = \{1010, 1011, 1110, 1111\}$

$L^3 = L \cdot L^2 =$

etc.

L^+ will not have L^0

Regular Expression

+, *, Concatenation (·).

$r, s \rightarrow$ regular expressions

$r+s$, $r \cdot s$, r^* will also be regular expressions.

Denotions related to regular expression (RE)

(1) ϕ is a RE and its language $L(\phi)$ is empty set.

(2) ϵ is a RE $\quad L(\epsilon) = \{\epsilon\}$.

(3) $a \in \Sigma$ is a RE $\quad L(a) = \{a\}$

(4) If 'r' and 's' are RE, then $r+s$, rs, r^* are RE

$\Sigma = \{0,1\}$ $\quad 0+1$, 01, 0^*

$r \rightarrow R = L(r)$

$s \rightarrow S = L(s)$

(i) $r+s = R \cup S = L(r) \cup L(s) = L(r+s)$

(ii) $L(rs) = L(r)\,L(s) = RS$

(iii) $L(r^*) = (L(r))^* = R^* = \bigcup_{i=0}^{\infty} R^i$

Eg:- $\Sigma = \{0, 1\}$ $r = 0$ and $s = 1$

$L(0+1) = L(0) \cup L(1) = \{0, 1\} = \Sigma$ ✓

$L(01) = L(0) \cdot L(1) = \{01\}$. ✓

$0^* = (L(0))^* = \{\epsilon, 0, 00, 000, \text{---}\}$

$$(0+1)^* = \Sigma^*$$

$$= \{ \epsilon, 0, 1, 00, 01, 10, 11, \text{---} \}.$$

$$\therefore L((0+1)^*) = \Sigma^*$$

$(0+1)^* = \Sigma^*$

$= \{ \epsilon, 0, 1, 00, 01, 10, 11, --- \}$.

$\therefore L((0+1)^*) = \Sigma^*$

$\Sigma = \{a, b\}$

$r = (a+b)^* \; aab \; (a+b)^*$

$L(r) =$ All binary strings Containing aab as Substring

$= \{ aab, aaab, aaaba, --- \}$.

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

$L = (1+01)$

$L^* \rightarrow ?$

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

$R = (1 + 01)$ $\qquad R^* \rightarrow ?$

$L(R) = R_1 + R_2 = 1 + 01 = \{1, 01\}$

$R^* = (L(R))^* = \{1, 01\}^*$

$= \{\epsilon, 1, 01, 101, 011, \ldots\}$

Precedence of the operators

+, Concatenation (·), *

* > Concatenation > +

Eg: $((0(1^*))+0) = 01^*+1$

$= \{0, 1, 01, 011, 0111, \ldots\}$

Precedence of the operators

+, Concatenation (·), *

* > Concatenation > +

Eg1 $((0(1^*))+0) = 01^*+1$

$= \{0, 1, 01, 011, 0111, \ldots\}$

Eg2:- $0^*1^*2^*$ → Set of all strings with any 0's followed by any 1's followed by any 2's.

Algebraic laws of the RE operators

(1) Commutativity of '+' :- $R + S = S + R$

Eg $0+1 = 1+0$

(2) Associativity of '+' :- R, S, T

$(R+S)+T = R+(S+T)$

(3) Associativity of concatenation(·) :- $(RS)T = R(ST)$

$\{ xyz \mid x \in R, y \in S, z \in T \}$

(4) Is Concatenation Commutative ??

Eg $R = 0$ and $S = 1$

$$L(RS) = 01 = \{xy \mid x \in R, y \in S\}.$$

$$L(SR) = 10 = \{xy \mid x \in S, y \in R\}.$$

$$L(RS) \neq L(SR) \quad \therefore \text{Not Commutative.}$$

(5) $0 + R = R + \phi = R$

$$L(\phi) \cup L(R) = \phi \cup L(R) = L(R)$$

(6) $\in R = R \in = R$

$$L(\in R) = L(R \in) = L(R).$$

Distributive laws

$$R(S+T) = RS + RT$$

$$= \{ xy \mid x \in L(R),\ y \in L(S+T) \text{ and } y \in L(S) \text{ or } y \in L(T) \}$$

$$= \{ x t_1 \mid x \in L(R),\ t_1 \in L(S) \}$$

$$\cup \{ x t_2 \mid x \in L(R),\ t_2 \in L(T) \}$$

$$= L(RS) \cup L(RT)$$

distributive laws (Contd...)

$(R+S)\,t = Rt + St$

$L(R) \cup L(S)$

$= \{ xz \mid x \in L(R) \text{ or } x \in L(S) \text{ and } z \in L(t) \}$

$= \{ xz \mid x \in L(R) \text{ and } z \in L(t) \}$

$\cup \{ xz \mid x \in L(S) \text{ and } z \in L(t) \}$

$= Rt + St$

Idempotent law :-

$$R + R = R$$

$$= L(R) \cup L(R) = L(R)$$

Laws on * :-

$$(R^*)^* = R^*$$

Eg:- $(0^*)^* = 0^*$

Eg:- $((01)^*)^* = (01)^*$

$$\epsilon^* = \epsilon$$

$$L(\epsilon^*) = \bigcup_{i=0}^{\infty} (L(\epsilon))^i$$

$$= \epsilon$$

$$\phi^* = \epsilon$$

$$L(\phi) = \phi$$

$$L(\phi^*) = \bigcup_{i=0}^{\infty} (L(\phi))^i$$

$$= \epsilon \cup \phi \cup \phi \cdots = \epsilon$$

(1) $(R+S)^* = (R^* S^*)^*$

(2) $R^* = R^* R^*$

(3) $R + S.R \neq (R+S).R$

Eg1:- $(0+1)^* = (0^* 1^*)^* = \Sigma^*$ $\quad \Sigma = \{0,1\}$

All Binary strings.

Eg2:- $0^* = 0^*.0^*$

Eg3:- $0+10 \neq (0+1).0$

① $0 \cdot 1^* + 2 = \Lambda$

$L(\Lambda) =$

② $(0^*, 1^*)^* = \Lambda$

$L(\Lambda) =$

③ $(0^*, 1^* + 0^*, 1^*)^* = \Lambda$

$L(\Lambda) =$

④ $(0+1)^* \cdot (0+1)^* = \Lambda$

$L(\Lambda) =$

⑤ $(0+1) \cdot 001 = \Lambda$

$L(\Lambda) =$

① $0 \cdot 1^* + 2 = R$

$L(R) = \{0, 2, 01, 011, 0111, \dots\}$

② $(0^*, 1^*)^* = R$

$L(R)$ = Set of all binary strings = $\{\epsilon, 0, 1, 00, 01, 10, 11, \dots\}$

③ $(0^*, 1^* + 0^*, 1^*)^* = R$

$L(R)$ = Same as ②

② = ③ = ④

④ $(0+1)^* \cdot (0+1)^* = R$

$L(R)$ = Same as ②

⑤ $(0+1) \cdot 001 = R$

$L(R) = \{0001, 1001\}$

Equivalence between Regular Expression and ϵ-NFA

$R \rightarrow L(R)$

ϵ-NFA $\rightarrow E$ $\quad L(E) = L(R)$

Theorem:- Let R be a RE, then $\exists$ ϵ-NFA, E that accept $L(R)$

$$L(E) = L(R)$$

Proof:-

Equivalence between Regular Expression and ϵ-NFA

$R \rightarrow L(R)$

ϵ-NFA $\rightarrow E$ $\quad L(E) = L(R)$

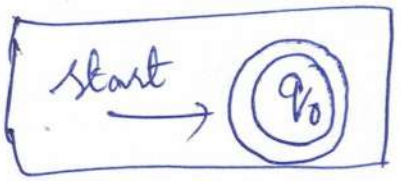
Theorem:- Let R be a RE, then $\exists$ ϵ-NFA, E that accepts $L(R)$

$$L(E) = L(R)$$

Proof:- By Induction on all operators in R, we can show that there is a ϵ-NFA E, having one final state and no transition out from this final state s.t. $L(R) = L(E)$

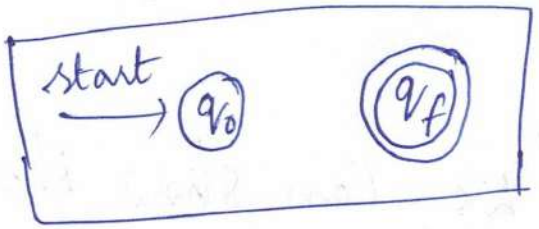
Base (Zero operators)

1) $R = \epsilon \Rightarrow L(R) = \{\epsilon\}$



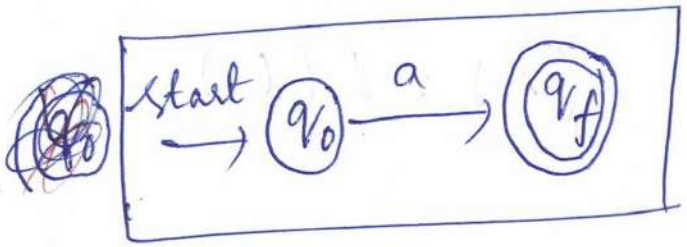
ϵ-NFA

2) $R = \phi \Rightarrow L(R) = \phi$



ϵ-NFA

3) $R = a, \quad a \in \Sigma \Rightarrow L(R) = \{a\}$



ϵ-NFA

Induction (one or more operators)

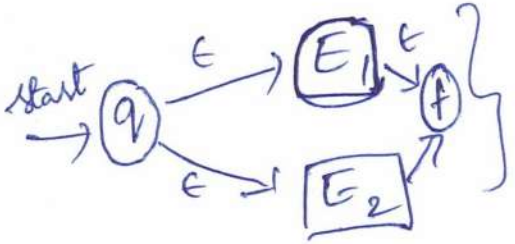
I.H { Assume that the theorem is true for RE with fewer than i operators, $i \geq 1$

r = i operators

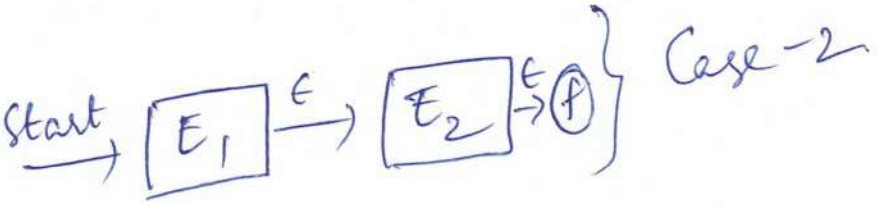
$L(r_1) = L(E_1)$

$L(r_2) = L(E_2)$

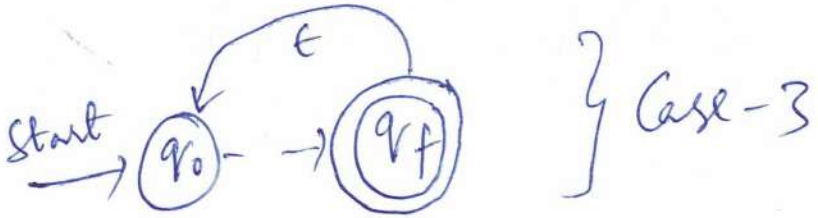
Case-1 $r = r_1 + r_2$



Case-2 $r = r_1 \cdot r_2$



Case-3 $r = r_1^*$

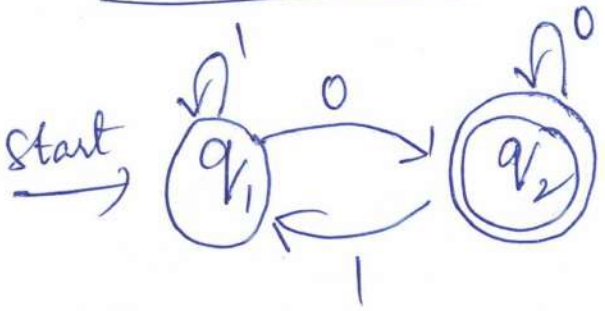


E_1 = 

∴ Every RE will have ϵ-NFA

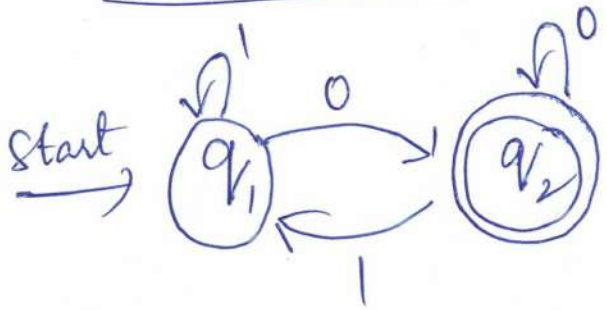
DFA to Regular Expression

Eg:-



DFA to Regular Expression

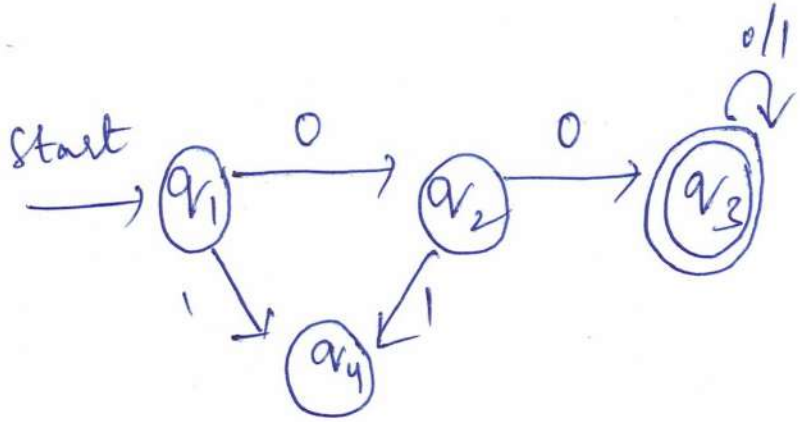
Eg1:-



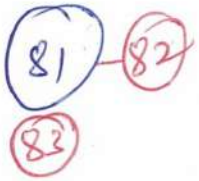
$L(\Lambda) = \{x0 / x \in \Sigma^*\}$

$R = (0+1)^* 0$

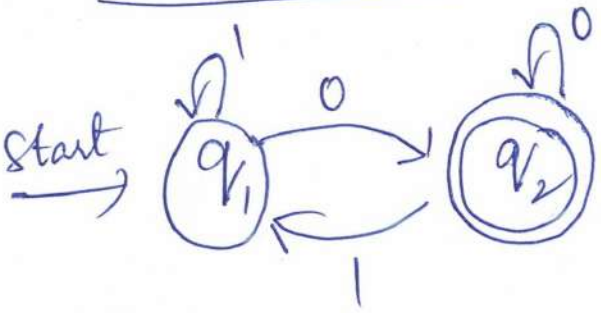
Eg2:-



DFA to Regular Expression



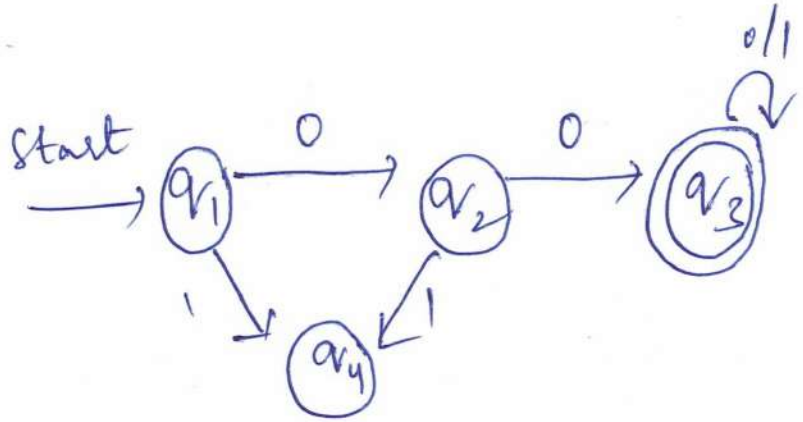
Eg1:-



$L(\Lambda) = \{x0 / x \in \Sigma^*\}$

$R = (0+1)^* 0$

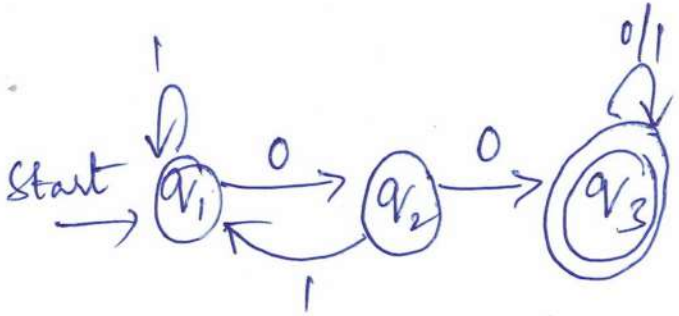
Eg2:-



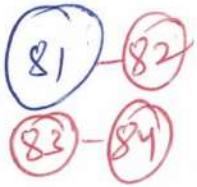
$L(\Lambda) = \{00x / x \in \Sigma^*\}$

$R = 00 (0+1)^*$

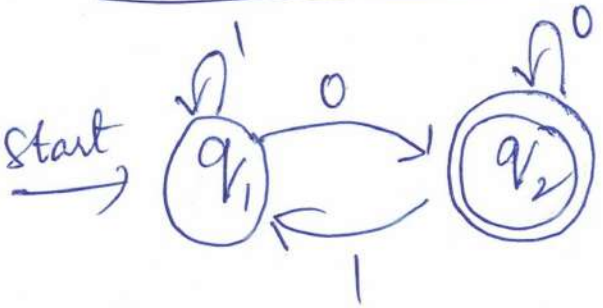
Eg3:-



DFA to Regular Expression



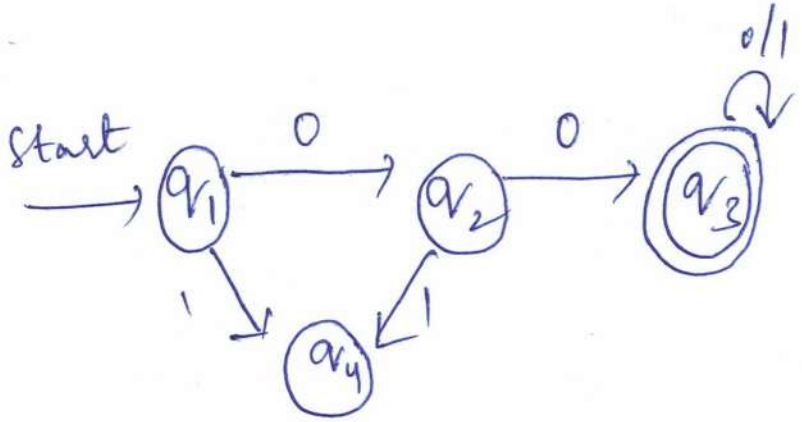
Eg1:-



$L(\Lambda) = \{x0 / x \in \Sigma^*\}$

$R = (0+1)^* 0$

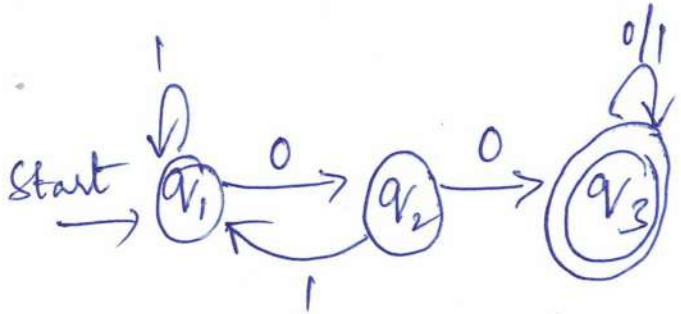
Eg2:-



$L(\Lambda) = \{00x / x \in \Sigma^*\}$

$R = 00 (0+1)^*$

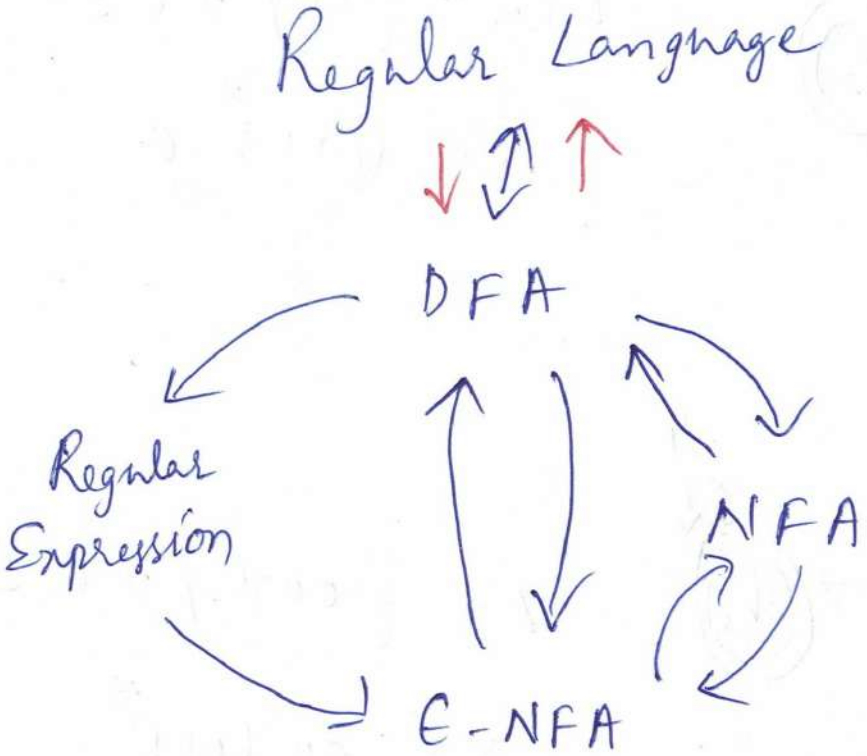
Eg3:-



$L(R) = \{x00x / x \in \Sigma^*\}$

$R = (0+1)^* 00 (0+1)^*$

Summary Diagram



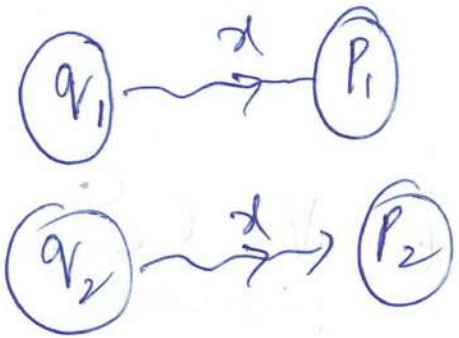
Minimization of FA

$(Q, \Sigma, \delta, q_0, F)$ DFA

$q_1 \equiv q_2$

if $\hat{\delta}(q_1, x)$ and $\hat{\delta}(q_2, x)$ are both $\in F$

or both $\notin F$



either $p_1, p_2 \in F$ (Final)

or

$p_1, p_2 \notin F$ (Final)

(1) $P \equiv P$ Equivalence (reflexive, symmetric, transitive)

(2) $P \equiv q \Leftrightarrow q \equiv P$

(3) $P \equiv q$ and $q \equiv r \Rightarrow P \equiv r$

Recursive Construction of $(k+1)^{th}$ equivalence.

(i) q_1, q_2 $(k+1)$ th equivalent $\Rightarrow$ q_1, q_2 must be k-equivalent

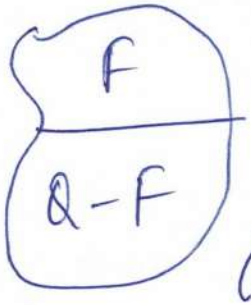
(ii) q_1, q_2 $(k+1)^{th}$ equivalent if

$\delta(q_1, a)$, $\delta(q_2, a)$ are k-equivalent $\forall a \in \Sigma$

Construction of Minimum DFA

1. $\pi_0 \rightarrow$ 0th equivalent class

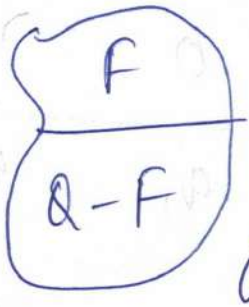
$\pi_0 = (Q_1^0, Q_2^0)$ i.e. $Q_1^0 = F$, $Q_2^0 = Q - F$



Construction of Minimum DFA

1. $\Pi_0 \rightarrow$ 0th equivalent class

$\Pi_0 = (Q_1^0, Q_2)$ i.e. $Q_1^0 = F$, $Q_2^0 = Q - F$



2. Construction of Π_{k+1} from Π_k

$q_1, q_2 \in Q_i^k$ and

$q_1, q_2 \in Q_j^{k+1}$ if $\delta(q_1, a)$, $\delta(q_2, a)$ are k-equivalent $\forall\ a \in \Sigma$

Repeat until

$\Pi_n = \Pi_{n+1}$ ✓

Example:-

δ	0	1
$\rightarrow q_0$	q_1	q_5
q_1	q_6	q_2
$*q_2$	q_0	q_2
q_3	q_2	q_6
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_6	q_4
q_7	q_6	q_2

$\Pi_0 = \{ \{q_2\}, \{q_0, q_1, q_3, q_4, q_5, q_6, q_7\} \}$

$\Pi_1 = ??$

$q_0 \equiv q_1$??

$\delta(q_0, 0) = q_1 \quad \delta(q_0, 1) = q_1 \notin F$ $\therefore q_0 \equiv q_1$

$\delta(q_1, 0) = q_6 \quad \delta(q_1, 1) = q_2 \in F$

$q_0 \equiv q_4$???

$\delta(q_0, 0) = q_1 \quad \delta(q_4, 0) = q_7$

$\delta(q_0, 1) = q_5 \quad \delta(q_4, 1) = q_5$

Example:-

δ	0	1
$\rightarrow q_0$	q_1	q_5
q_1	q_6	q_2
$*q_2$	q_0	q_2
q_3	q_2	q_6
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_6	q_4
q_7	q_6	q_2

$\Pi_0 = \{ \{q_2\}, \{q_0, q_1, q_3, q_4, q_5, q_6, q_7\} \}$

$\Pi_1 = ??$

$q_0 \equiv q_1 ??$

$\delta(q_0, 0) = q_1 \quad \delta(q_0, 1) = q_1 \notin F$ $\therefore q_0 \equiv q_1$

$\delta(q_1, 0) = q_6 \quad \delta(q_1, 1) = q_2 \in F$

$q_0 \equiv q_4 ???$

$\delta(q_0, 0) = q_1 \quad \delta(q_4, 0) = q_7$

$\delta(q_0, 1) = q_5 \quad \delta(q_4, 1) = q_5$

$\Pi_1 = \{ \{q_2\}, \{q_0, q_4, q_6\}, \{q_1, q_7\}, \{q_3, q_5\} \}$

$\Pi_2 = ??$

$\delta(q_4, 0) = q_7$ } belongs to diff. classes
$\delta(q_6, 0) = q_6$ } in Π_1

Example:-

δ	0	1
$\rightarrow q_0$	q_1	q_5
q_1	q_6	q_2
$*q_2$	q_0	q_2
q_3	q_2	q_6
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_6	q_4
q_7	q_6	q_2

$\Pi_0 = \{\{q_2\}, \{q_0, q_1, q_3, q_4, q_5, q_6, q_7\}\}$

$\Pi_1 = ??$

$q_0 \equiv q_1 ??$

$\delta(q_0, 0) = q_1 \quad \delta(q_0, 1) = q_1 \in F$ $\therefore q_0 \equiv q_1$

$\delta(q_1, 0) = q_6 \quad \delta(q_1, 1) = q_2 \in F$

$q_0 \equiv q_4 ???$

$\delta(q_0, 0) = q_1 \quad \delta(q_4, 0) = q_7$

$\delta(q_0, 1) = q_5 \quad \delta(q_4, 1) = q_5$

$\Pi_1 = \{\{q_2\}, \{q_0, q_4, q_6\}, \{q_1, q_7\}, \{q_3, q_5\}\}$

$\Pi_2 = ??$

$\delta(q_4, 0) = q_7$ } belongs to diff. classes
$\delta(q_6, 0) = q_6$ } in Π_1

$\Pi_2 = \{\{q_2\}, \{q_6\}, \{q_0, q_4\}, \{q_1, q_7\}, \{q_3, q_5\}\}$

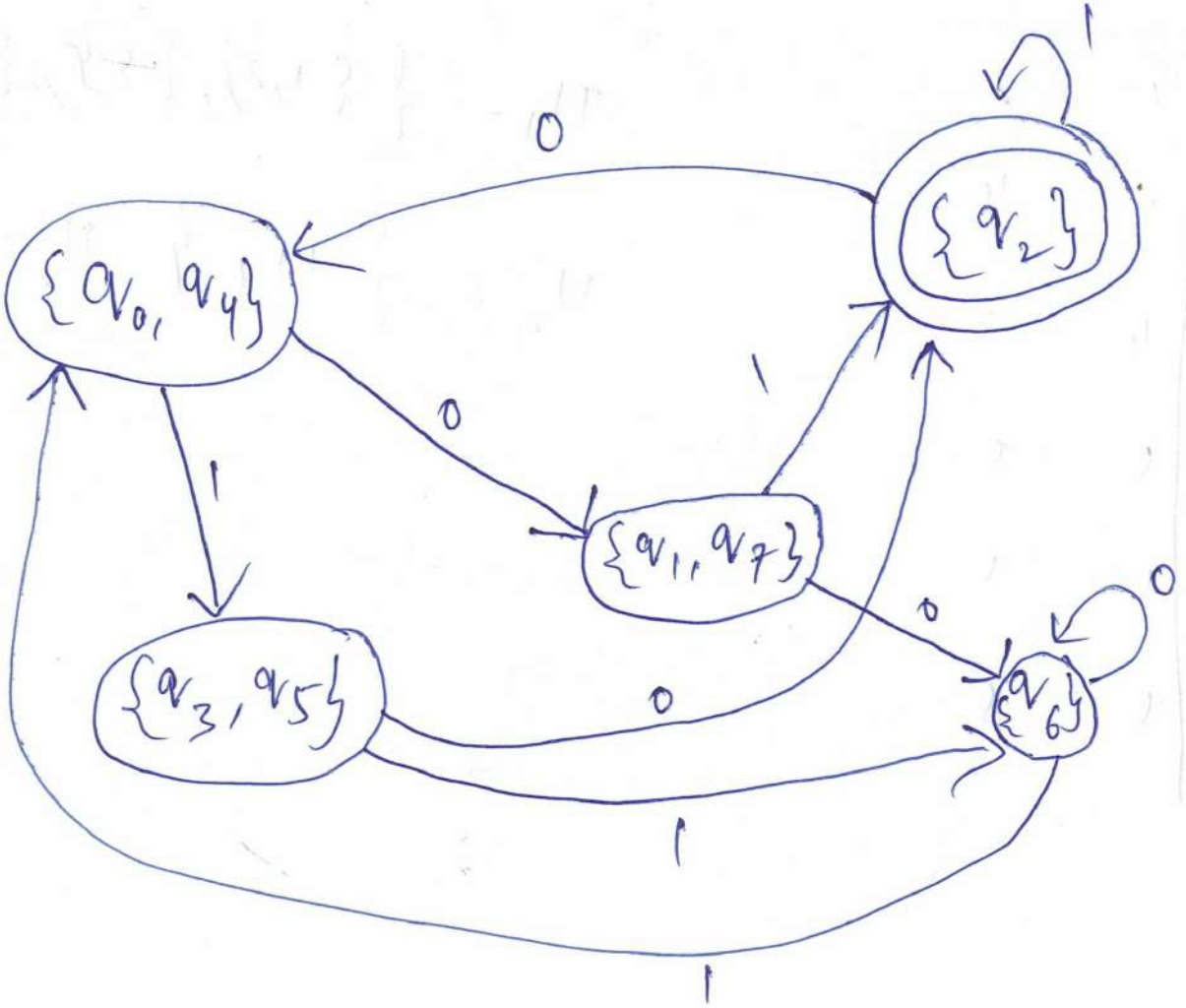
$\Pi_3 = \Pi_2$

δ	0	1
→ q_0	q_1	q_5
q_1	q_6	q_2
* q_2	q_0	q_2
q_3	q_2	q_6
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_6	q_4
q_7	q_6	q_2

$\Pi_3 = \Pi_2 = \{\{q_2\}, \{q_6\}, \{q_0, q_4\}, \{q_1, q_7\}, \{q_3, q_5\}\}$

δ	0	1
→ q_0	q_1	q_5
q_1	q_6	q_2
* q_2	q_0	q_2
q_3	q_2	q_6
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_6	q_4
q_7	q_6	q_2

$\pi_3 = \pi_2 = \{\{q_2\}, \{q_6\}, \{q_0, q_4\}, \{q_1, q_7\}, \{q_3, q_5\}\}$ (q3) (q4)



Example $\Sigma = \{a, b\}$

δ	a	b
→ 0	1	2
* 1	3	4
* 2	4	3
3	5	5
4	5	5
* 5	5	5

$\pi_0 = \{\{1, 2, 5\}, \{0, 3, 4\}\}$

$\pi_1 = \{\{1, 2\}, \{5\}, \{0\}, \{3, 4\}\}$

$\pi_2 = \{\pi_1\}$ $\pi_2 = \pi_1$

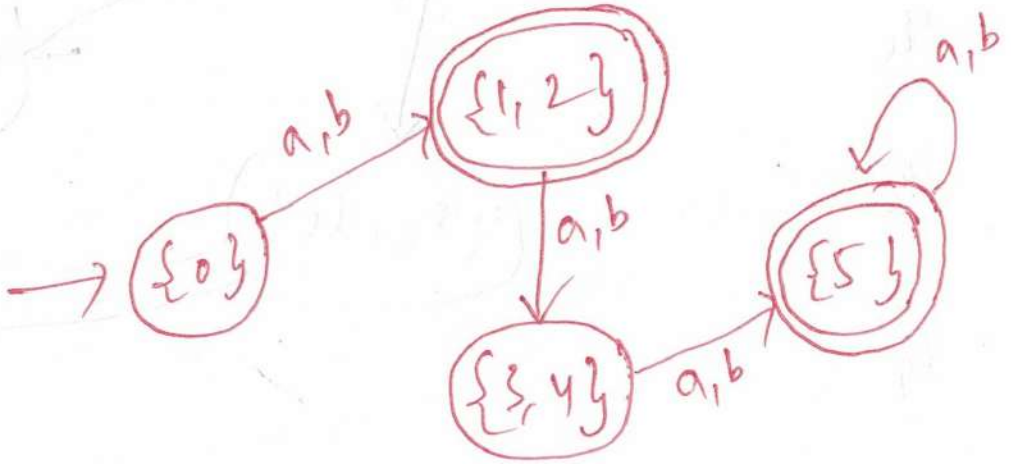
Example $\Sigma = \{a, b\}$

δ	a	b
→ 0	1	2
* 1	3	4
* 2	4	3
3	5	5
4	5	5
* 5	5	5

$\pi_0 = \{\{1, 2, 5\}, \{0, 3, 4\}\}$

$\pi_1 = \{\{1, 2\}, \{5\}, \{0\}, \{3, 4\}\}$

$\pi_2 = \{\pi_1\}$ $\pi_2 = \pi_1$



$1 \approx 2, \; 3 \approx 4$

Context Free Grammar (CFG)

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

① $V \rightarrow$ Set of Variables

$V \neq \phi$

$|V|$ = finite

② T = Set of terminals

$T \neq \phi$

$|T|$ = finite

③ P = finite set of productions / rules.

$V \cap T = \phi$

④ S = Starting Variable

$S \in V$

Example:- $G_1 = (\{E\}, \{+, *, (,), id\}, P, E)$ (CFG)

P :-

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow id$

Example:- $G_1 = (\{E\}, \{+, *, (,), id\}, P, E)$ (CFG)

P:-

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow id$

$E \Rightarrow E*E$

$\Rightarrow (E)*E$

$\Rightarrow (E)*id$

$\Rightarrow (E+E)*id$

$\Rightarrow$ ~~$(E*$~~ $(E+id)*id$

$\Rightarrow (id+id)*id$

Example:- Palindromes

$\epsilon, 0, 1$

$\epsilon, 0, 1, 00, 11, 010, 0110, 101, 1001$ -- etc.

$G_{pal} = (\{E\}, \{0,1\}, P, E)$

P:- ??

Example:- Palindromes

$\epsilon, 0, 1$

$\epsilon, 0, 1, 00, 11, 010, 0110, 101, 1001$ -- etc.

$G_{pal} = (\{E\}, \{0,1\}, P, E)$

P:- ??

E $\Rightarrow$ ϵ

E $\rightarrow$ 0

E $\rightarrow$ 1

E $\rightarrow$ 0E0

E $\rightarrow$ 1E1

P:- $E \rightarrow \epsilon / 0 / 1 / 0E0 / 1E1$ ✓

CFG $G = (V, T, P, S)$

$A \to \not{R} A \alpha / \epsilon$

$A \Rightarrow \not{R} A \alpha \Rightarrow \not{R}\not{R} A \alpha\alpha \Rightarrow \not{R}\not{R}\alpha\alpha$

$A \overset{*}{\underset{G}{\Rightarrow}} \not{R}\not{R}\alpha\alpha$

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

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

L) Language of the CFG G.

Example:- $G = (V, T, P, S)$

$V = \{S\}$ $T = \{a, b\}$

$P = \{S \rightarrow asb \mid ab\}$

$L(G) = ??$

Example:- $G = (V, T, P, S)$

$V = \{S\}$ $T = \{a, b\}$

$P = \{S \to asb \mid ab\}$

$S \Rightarrow ab$

$S \overset{*}{\Rightarrow} a^2 b^2$

$S \overset{*}{\Rightarrow} a^3 b^3$

$L(G) = ??$

$L(G) = \{ab, a^2b^2, a^3b^3, \ldots a^n b^n\}$ $n \geq 1$

$\therefore L(G) = \{a^n b^n / n \geq 1\}$

Example:- Find a CFG for all binary strings with

L {even number of 0's} $\Sigma = \{0, 1\}$

$L = \{\epsilon, 00, 010, 001, 100, -- etc\}$

Solut:-

Cases:

1) 1st Symbol start with 1, followed by even number of 0's

2) 1st Symbol start with 0, then followed by even 0's.

Example:- Find a CFG for all binary strings with

L {even number of 0's $\Sigma = \{0, 1\}$

$L = \{ \epsilon, 00, 010, 001, 100 - - \text{etc} \}$

Solut:-

Case:

1) 1st Symbol start with 1, followed by even number of 0's and having even 0's.

2) 1st Symbol start with 0, ~~then followed by~~ and having even 0's.

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

$S \rightarrow 1S \mid 0A0S \mid \epsilon$

$A \rightarrow 1A \mid \epsilon$

$L = \{ w \in (0,1)^* / w \text{ has even number of 0's} \}$ (11)

$G_1 = (\{S, A\}, \{0,1\}, P_1, S\}$

$P_1 :\ S \rightarrow 1S / 0A0S / \epsilon$

$A \rightarrow 1A / \epsilon$

$L = \{ w \in (0,1)^* / w \text{ has even number of } 0\text{'s} \}$ (11)-(12)

$G_1 = (\{S, A\}, \{0,1\}, P_1, S\}$

$P_1 :\ S \to 1S / 0A0S / \epsilon$

$A \to 1A / \epsilon$

$G_2 = \{ \{S, T\}, \{0,1\}, P_2, S\}$

$P_2 :\ S \to 1S / 0T / \epsilon$

$T \to 1T / 0S$

$L(G_1) = L = L(G_2)$

Two Equivalent CFGs.

Context Free Grammar

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

$P: \{ A \rightarrow \alpha \}$

$A \in V, \quad \alpha \in (V \cup T)^*$

$A \in (V \cup T)^*$

Example:- $L(G) = \{ 0^n 1^n 2^n \mid n \geq 1 \}$

$G = (\{S, A, \}, \{0, 1, 2\}, P, S)$

P: $S \rightarrow 0SA2$

$S \rightarrow 012$

$2A \rightarrow A2$

$1A \rightarrow 11$

Example:- $L(G) = \{ 0^n 1^n 2^n \mid n \geq 1 \}$

$G = (\{S, A\}, \{0, 1, 2\}, P, S)$

P:

$S \rightarrow 0SA2$

$S \rightarrow 012$

$2A \rightarrow A2$

$1A \rightarrow 11$

$S \rightarrow 012 \in L(G)$

$S \rightarrow 0SA2$

$S \rightarrow 0012A2$

$S \rightarrow 001A22$

$S \rightarrow 00112 2$

Derivation tree / Parse tree

$G = (V, T, P, S)$, a Parse tree for G if :-

(i) Every vertex has label which is Symbol of $V \cup T \cup \{\epsilon\}$

(ii) root is S.

(iii) If a vertex is interior and has label by $A \in V$.

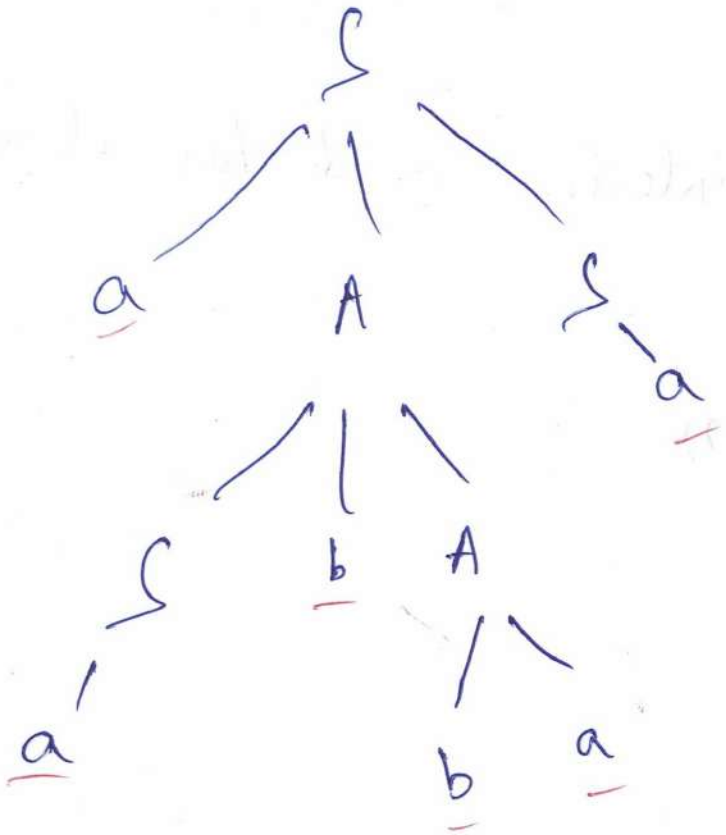
(iv) $A \rightarrow X_1 X_2 \cdots X_n$.

Example $G = (\{S, A\}, \{a, b\}, P, S)$

P: $S \rightarrow aAS / a$

$A \rightarrow SbA / SS / ba$

Parse tree



yields

a a b b a a

Parse tree of this string

a a b b a a

yield $\in (V \cup T)^*$

string

$\alpha \in (V \cup T)^*$

$S \overset{*}{\underset{G}{\Rightarrow}} \alpha$ iff we have a derivation tree that yield α.

Example: P: $S \rightarrow aAS \mid a$

$A \rightarrow SbA \mid SS \mid ba$

$S \Rightarrow aAS \Rightarrow aSSS \Rightarrow asbas \Rightarrow aSbAbas$

$\alpha = a\underline{S}b\underline{A}bas \in (V \cup T)^*$

Example $G = (\{S, A\}, \{a, b\}, P, S)$

$P: S \rightarrow aAS \mid a$

$A \rightarrow SbA \mid SS \mid ba$

$\alpha = aabbaa$

Leftmost derivation ??

Example $G = (\{S, A\}, \{a, b\}, P, S)$

P: $S \to aAS \mid a$

$A \to SbA \mid SS \mid ba$

$\alpha = aabbaa$

Leftmost derivation ??

$S \Rightarrow aAS \Rightarrow aSbAS \Rightarrow aabAS \Rightarrow aabbaS \Rightarrow aabbaa$

aabbaa

α.

$S \overset{*}{\underset{G}{\Rightarrow}} aabbaa$

Example:- $G = (\{S, A\}, \{a, b\}, P, S)$

P: $S \rightarrow aAS \mid a$

$A \rightarrow SbA \mid SS \mid ba$

$\alpha = aabbaa$

Right most Derivation ??

Example:- $G = (\{S, A\}, \{a, b\}, P, S)$

P: $S \rightarrow aAS \mid a$

$A \rightarrow SbA \mid SS \mid ba$

$\alpha = aabbaa$

Right most derivation ??

$S \Rightarrow aAS \Rightarrow aAa \Rightarrow aSbAa \Rightarrow aSbbaa$

aabbaa α.

$S \overset{*}{\underset{G}{\Rightarrow}} aabbaa$

Example $S \to 0B/1A$, $A \to 0/0S/1AA$, $B \to 1/1S/0BB$

$\alpha = 0011010|$

$G = (\{S, A, B\}, \{0, 1\}, P, S)$

Derivation tree for $\alpha = 0011010|$

Left most derivation ??

Sample $S \to 0B/1A$, $A \to 0/0S/1AA$, $B \to 1/1S/0BB$

$\alpha = 0011010|$

$G = (\{S, A, B\}, \{0, 1\}, P, S)$

Derivation tree for $\alpha = 0011010|$

Left most derivation.!?

$S \Rightarrow 0\underline{B} \Rightarrow 00\underline{BB} \Rightarrow 001\underline{S}B \Rightarrow 0011\underline{A}B \Rightarrow 00110\underline{S}B$

$00110101 \Leftarrow 0011010\underline{B} \Leftarrow 001101\underline{A}B$

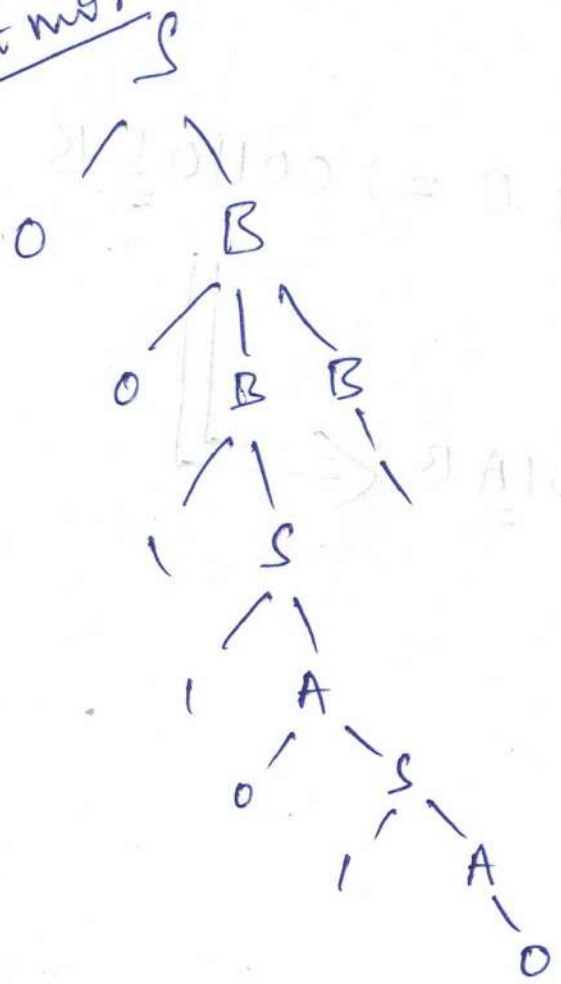
α.

$S \to 0B/1A$, $A \to 0/0S/1AA$, $B \to 1/1S/0BB$

$\alpha = 00110101$

derivation tree for $\alpha = 00110101$

Left most



$S \rightarrow 0B \mid 1A$, $A \rightarrow 0 \mid 0S \mid 1AA$, $B \rightarrow 1 \mid 1S \mid 0BB$

$\alpha = 00110101$

Right most derivation ??

$S \Rightarrow 0\underline{B} \Rightarrow 00B\underline{B} \Rightarrow 00\underline{B}1 \Rightarrow 001\underline{S}1 \Rightarrow 0011\underline{A}1$

$\Leftarrow 00110\underline{A}1 \Leftarrow 00110\underline{S}1$

00110101

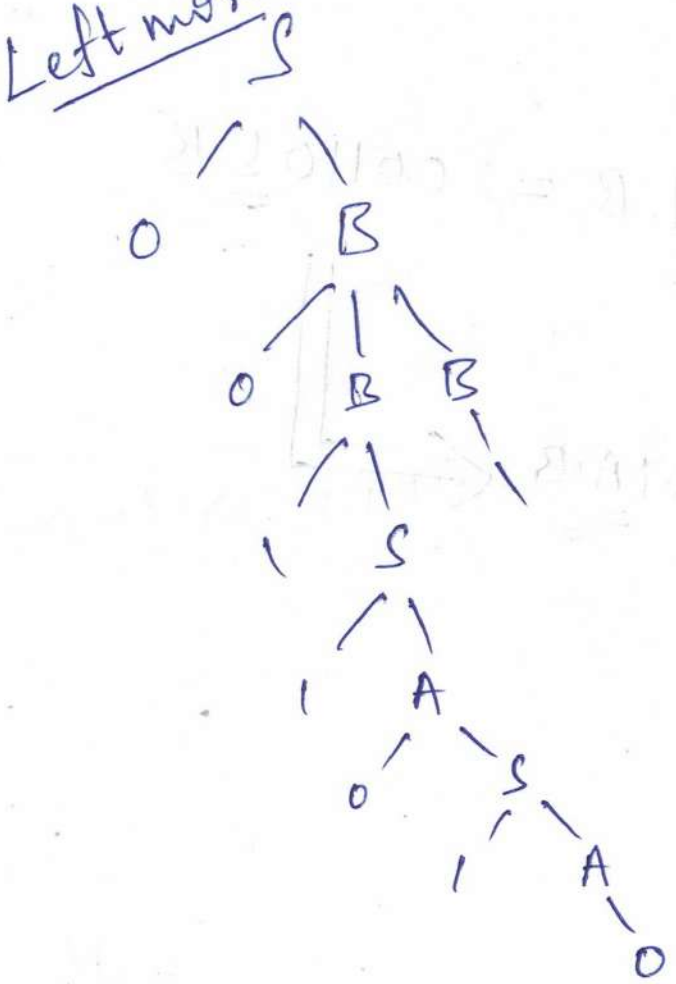
α

$S \to 0B / 1A$, $A \to 0 / 0S / 1AA$, $B \to 1 / 1S / 0BB$

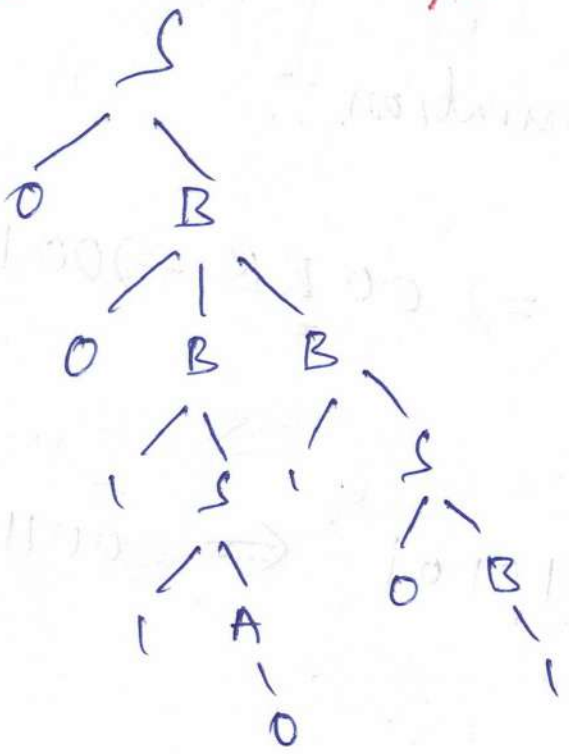
$\alpha = 00110101$

derivation tree for $\alpha = 00110101$

Left most



~~Right most~~ Another Left most derivation tree.



$S \to 0B / 1A$, $A \to 0/0S/1AA$, $B \to 1/1S/0BB$

$\alpha = 00110101$

Another left most derivation for $\alpha = 00110101$??

$S \Rightarrow 0\underline{B} \Rightarrow 00\underline{BB} \Rightarrow 001\underline{S}B \Rightarrow 0011\underline{A}B$

$00110101 \Leftarrow 00110\underline{1}B \Leftarrow 00110\underline{1}S \Leftarrow 00110\underline{B}$

α.

Same way Another Rightmost derivation also Exists for Same derivation tree.

Ambiguity in CFG.

Example:- "In books selected information is given."

selected — books
selected — Information.

Definition:- A terminal string $w \in L(G)$ is ambiguous if

$\exists$ two or more derivation tree for w.

($\exists$ two or more leftmost derivation of w)

- A CFG G is ambiguous if $\exists$ some $w \in L(G)$ which is ambiguous.
- A CFL for which every CFG is ambiguous then CFL is called inherently ambiguous.

Example :- $G = (\{S\}, \{a, b, +, *\}, P, S)$

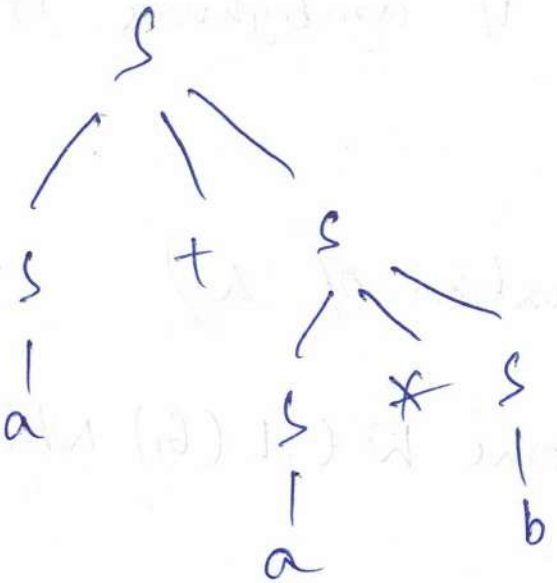
P: $S \rightarrow S+S \mid S*S \mid a \mid b$ $\quad w = a+a*b.$

one derivation tree.

Example :- $G = (\{S\}, \{a, b, +, *\}, P, S)$

P: $S \rightarrow S+S \mid S*S \mid a \mid b$ $\quad w = a+a*b.$

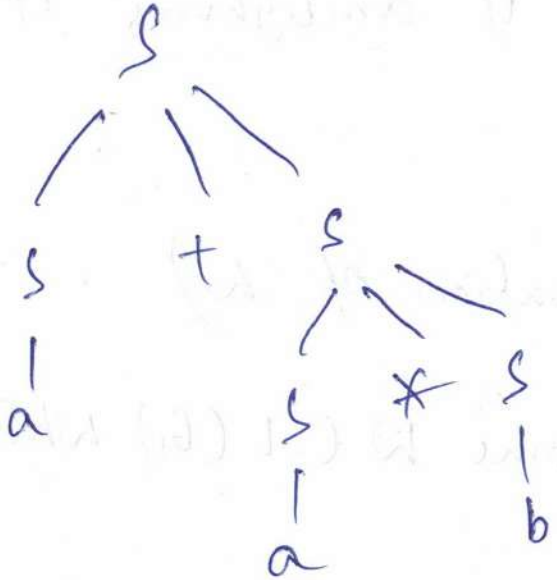
One derivation Tree.



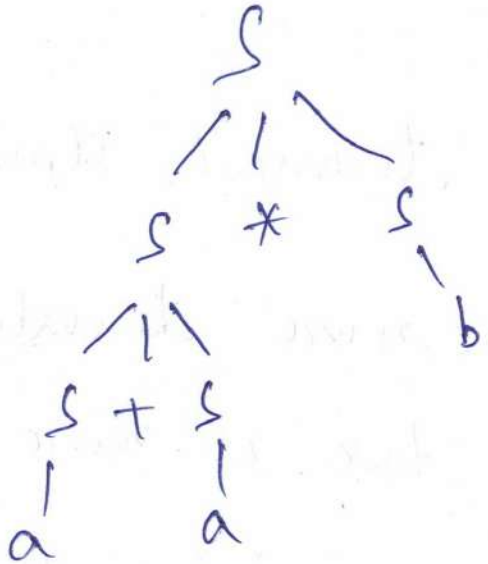
Example :- $G_1 = (\{S\}, \{a, b, +, *\}, P, S)$

P: $S \rightarrow S+S \mid S*S \mid a \mid b$ $\quad w = a+a*b$.

One derivation tree.



Another derivation tree



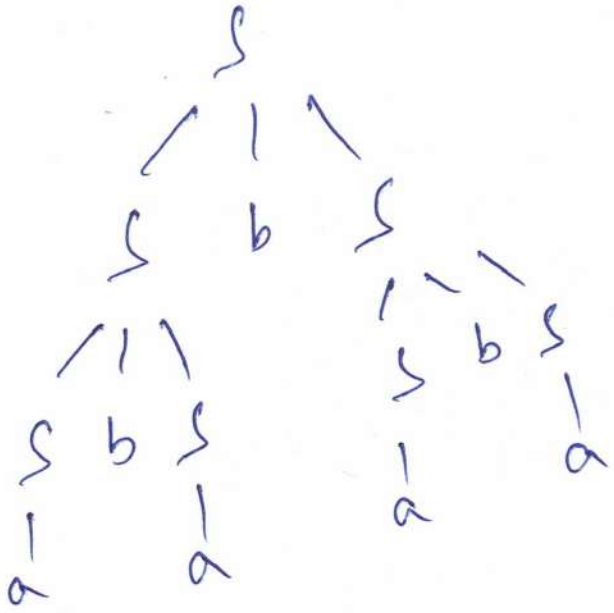
Two derivation trees exist so $w = a+a*b$ is ambiguous

and the G_1 is also Ambiguous.

Another Example:- $G = (\{S\}, \{a, b\}, P, S)$

P: $S \rightarrow SbS \mid a$ $\quad w = ababaa$

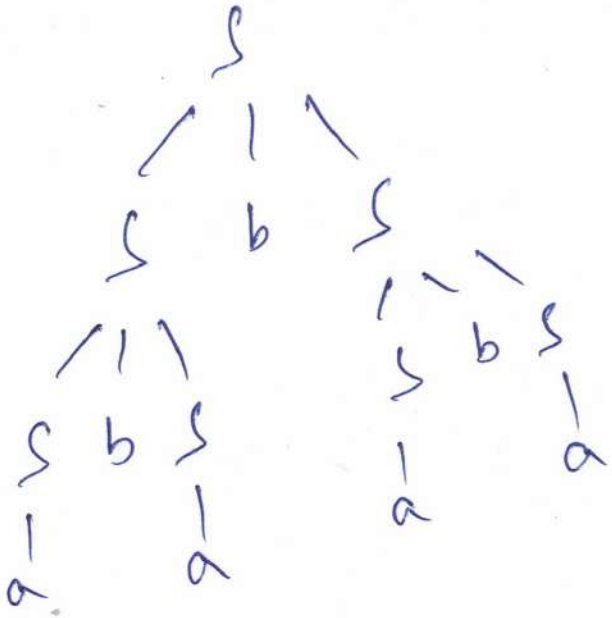
One derivation tree



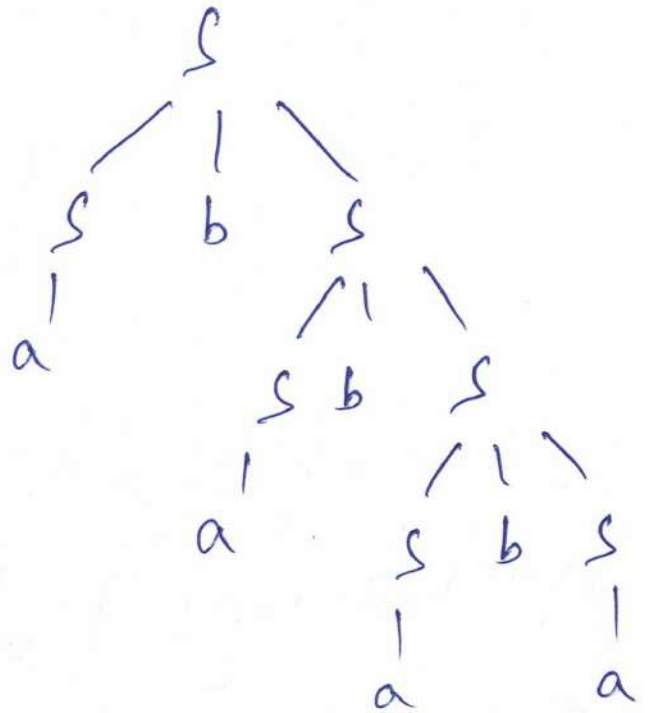
Another Example:- $G = (\{S\}, \{a, b\}, P, S)$

P: $S \rightarrow SbS \mid a$ $\quad w = ababa$

One derivation tree



Another derivation tree



$\therefore w = ababa$ is Ambiguous $\Rightarrow G$ is ambiguous

Simplification of CFG

$G = (V, T, P, S)$ → starting variable $S \in V$

Variables, terminals, Productions

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

↳ May be not all symbols $V \cup T$ are used

There is a possibility for simplifications.

Example:- $G_1 = (\{S, A, B, D, E\}, \{a, b, c\}, P, S)$

$P = \{S \to AB,\ A \to a,\ B \to b,\ B \to D,\ E \to c/\epsilon\}$

$L(G_1) = \{w \in T^* / S \overset{*}{\underset{G}{\Rightarrow}} w\} = \{ab\}$

Example:

$G_1 = (\{S, A, B, D, E\}, \{a, b, c\}, P, S)$

$P = \{S \to AB,\ A \to a,\ B \to b,\ B \to D,\ E \to c/\epsilon\}$

$L(G_1) = \{w \in T^* / S \overset{*}{\underset{G_1}{\Rightarrow}} w\} = \{ab\}$

derivation

$S \Rightarrow AB \Rightarrow aB \Rightarrow ab$

$G_1' = (V', T', P', S)$ Remove Revamp V, T, P which are not useful.

$V' = \{S, A, B\}$, $T' = \{a, b\}$

P' = remove red colored from P

$= \{S \to AB,\ A \to a,\ B \to b\}$.

Syntax Analysis:

Context-free Grammars, Pushdown Automata and Parsing
Part - 1

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is syntax analysis?
- Specification of programming languages: context-free grammars
- Parsing context-free languages: push-down automata
- Top-down parsing: LL(1) and recursive-descent parsing
- Bottom-up parsing: LR-parsing

Grammars

- Every programming language has precise grammar rules that describe the syntactic structure of well-formed programs
  - In C, the rules state how functions are made out of parameter lists, declarations, and statements; how statements are made of expressions, etc.
- Grammars are easy to understand, and parsers for programming languages can be constructed automatically from certain classes of grammars
- Parsers or syntax analyzers are generated *for* a particular grammar
- Context-free grammars are usually used for syntax specification of programming languages

What is Parsing or Syntax Analysis?

- A parser for a grammar of a programming language
  - verifies that the string of tokens for a program in that language can indeed be generated from that grammar
  - reports any syntax errors in the program
  - constructs a parse tree representation of the program (not necessarily explicit)
  - usually calls the lexical analyzer to supply a token to it when necessary
  - could be hand-written or automatically generated
  - is based on *context-free* grammars
- Grammars are generative mechanisms like regular expressions
- Pushdown automata are machines recognizing context-free languages (like FSA for RL)

Context-free Grammars

- A CFG is denoted as $G = (N, T, P, S)$
  - N: Finite set of non-terminals
  - T: Finite set of terminals
  - $S \in N$: The start symbol
  - P: Finite set of productions, each of the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$
- Usually, only P is specified and the first production corresponds to that of the start symbol
- Examples

(1)
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow id$

(2)
$S \rightarrow 0S0$
$S \rightarrow 1S1$
$S \rightarrow 0$
$S \rightarrow 1$
$S \rightarrow \epsilon$

(3)
$S \rightarrow aSb$
$S \rightarrow \epsilon$

(4)
$S \rightarrow aB \mid bA$
$A \rightarrow a \mid aS \mid bAA$
$B \rightarrow b \mid bS \mid aBB$

Derivations

- $E \Rightarrow^{E \to E+E} E + E \Rightarrow^{E \to id} id + E \Rightarrow^{E \to id} id + id$ is a derivation of the terminal string $id + id$ from E
- In a derivation, a production is applied at each step, to replace a nonterminal by the right-hand side of the corresponding production
- In the above example, the productions $E \to E + E$, $E \to id$, and $E \to id$, are applied at steps 1,2, and, 3 respectively
- The above derivation is represented in short as, $E \Rightarrow^* id + id$, and is read as S **derives** $id + id$

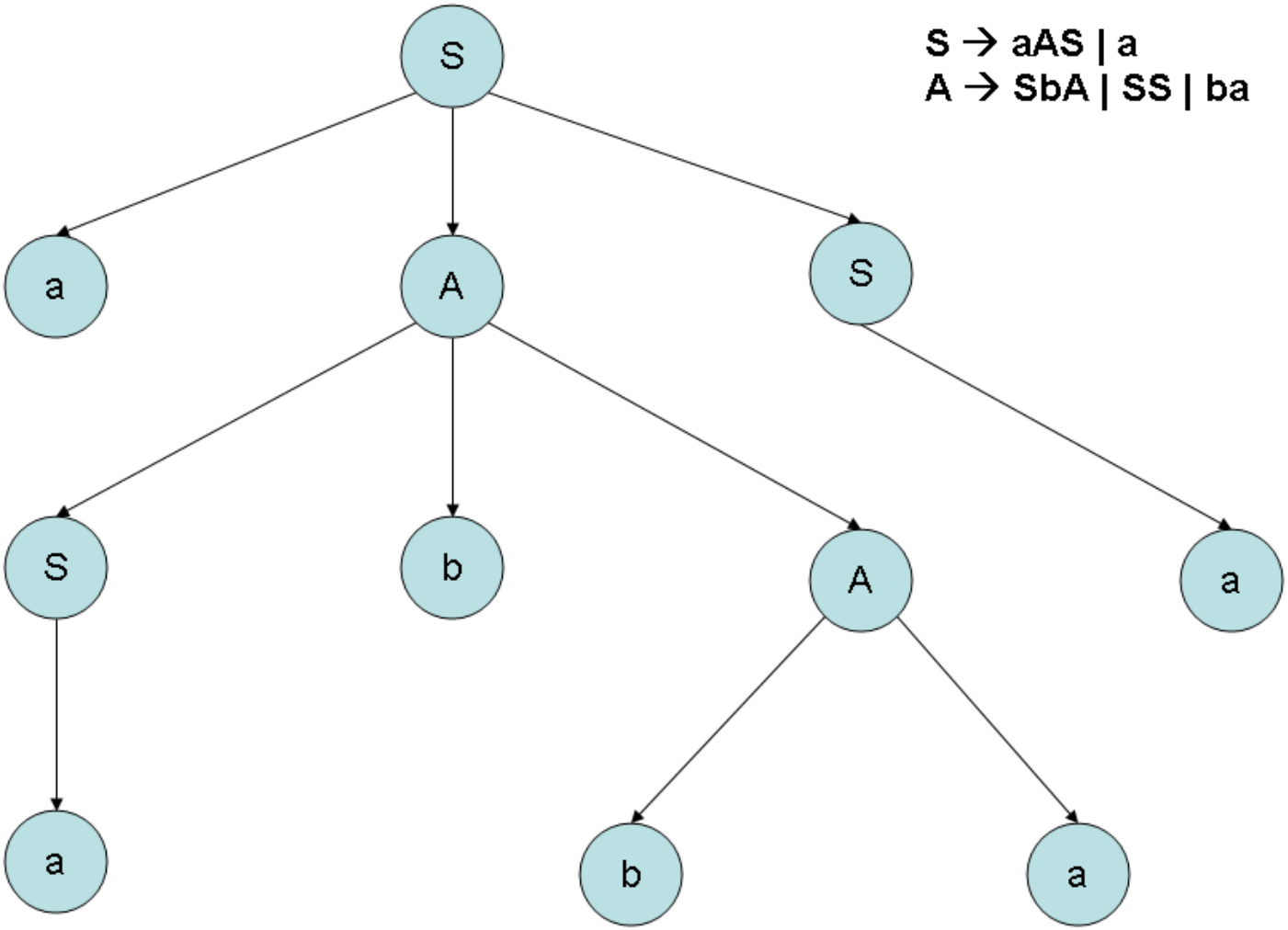
Context-free Languages

- Context-free grammars generate context-free languages (grammar and language resp.)
- The *language generated by G*, denoted $L(G)$, is
  $L(G) = \{w \mid w \in T^*, \text{ and } S \Rightarrow^* w\}$
  i.e., a string is in $L(G)$, if
  1. the string consists solely of terminals
  2. the string can be derived from S
- Examples
  1. $L(G_1)$ = Set of all expressions with +, *, names, and balanced '(' and ')'
  2. $L(G_2)$ = Set of palindromes over 0 and 1
  3. $L(G_3) = \{a^n b^n \mid n \geq 0\}$
  4. $L(G_4) = \{x \mid x \text{ has equal no. of } a's \text{ and } b's\}$
- A string $\alpha \in (N \cup T)^*$ is a **sentential form** if $S \Rightarrow^* \alpha$
- Two grammars G_1 and G_2 are equivalent, if $L(G_1) = L(G_2)$

Derivation Trees

- Derivations can be displayed as trees
- The internal nodes of the tree are all nonterminals and the leaves are all terminals
- Corresponding to each internal node A, there exists a production $\in$ P, with the RHS of the production being the list of children of A, read from left to right
- The **yield** of a derivation tree is the list of the labels of all the leaves read from left to right
- If α is the yield of some derivation tree for a grammar G, then $S \Rightarrow^* \alpha$ and conversely

Derivation Tree Example

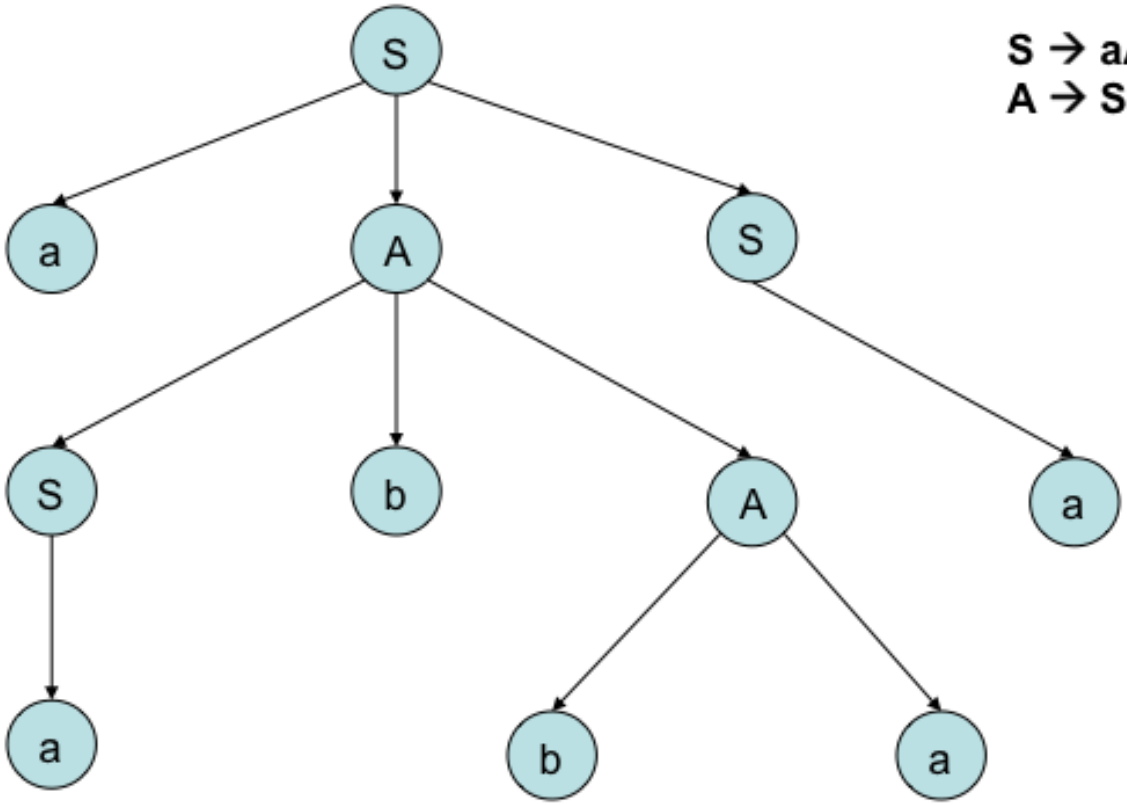


S => aAS => aSbAS => aabAS => aabbaS => aabbaa

Leftmost and Rightmost Derivations

- If at each step in a derivation, a production is applied to the leftmost nonterminal, then the derivation is said to be **leftmost**. Similarly **rightmost derivation**.
- If $w \in L(G)$ for some G, then w has at least one parse tree and corresponding to a parse tree, w has unique leftmost and rightmost derivations
- If some word w in $L(G)$ has two or more parse trees, then G is said to be **ambiguous**
- A CFL for which every G is ambiguous, is said to be an **inherently ambiguous** CFL

Leftmost and Rightmost Derivations: An Example



S → aAS | a
A → SbA | SS | ba

Leftmost derivation: **S => aAS => aSbAS => aabAS => aabbaS => aabbaa**

Rightmost derivation: **S => aAS => aAa => aSbAa => aSbbaa => aabbaa**

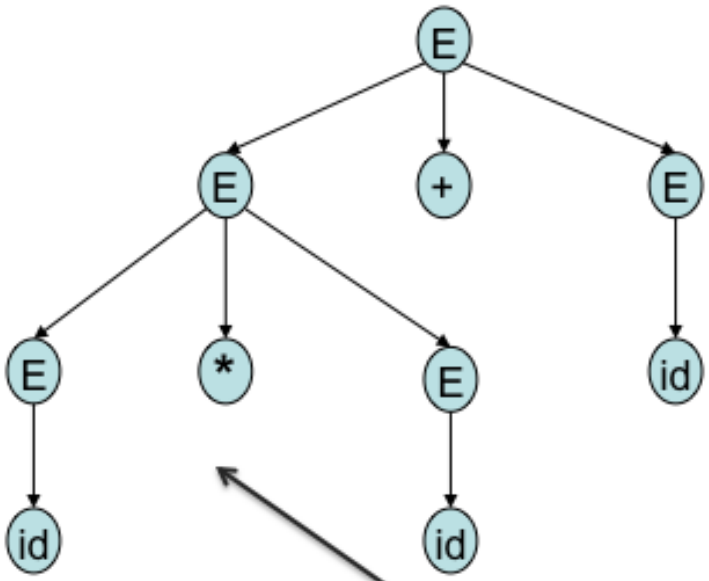
Ambiguous Grammar Examples

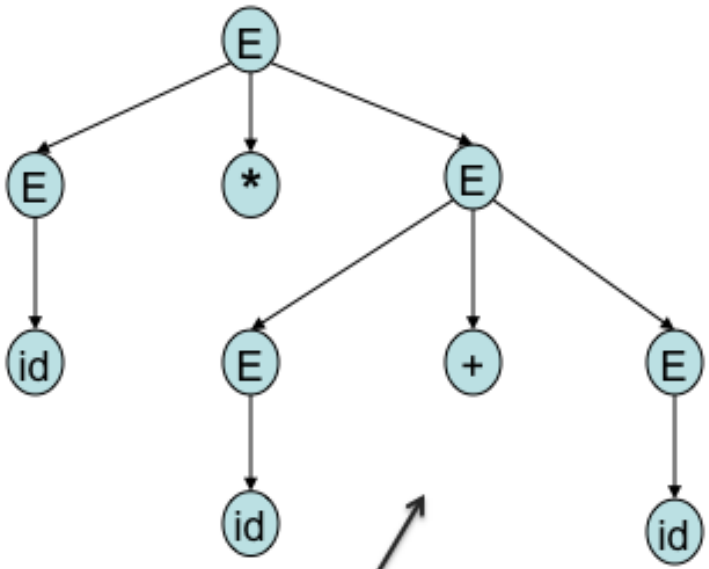
- The grammar, $E \rightarrow E + E | E * E | (E) | id$
  is ambiguous, but the following grammar for the same language is unambiguous
  $E \rightarrow E + T | T$, $T \rightarrow T * F | F$, $F \rightarrow (E) | id$
- The grammar,
  $stmt \rightarrow IF\ expr\ stmt | IF\ expr\ stmt\ ELSE\ stmt | other\_stmt$

  is ambiguous, but the following equivalent grammar is not

  $stmt \rightarrow IF\ expr\ stmt | IF\ expr\ matched\_stmt\ ELSE\ stmt$
  $matched\_stmt \rightarrow$
  $IF\ expr\ matched\_stmt\ ELSE\ matched\_stmt | other\_stmt$
- The language,
  $L = \{a^n b^n c^m d^m \mid n, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n, m \geq 1\}$,
  is inherently ambiguous

Ambiguity Example 1



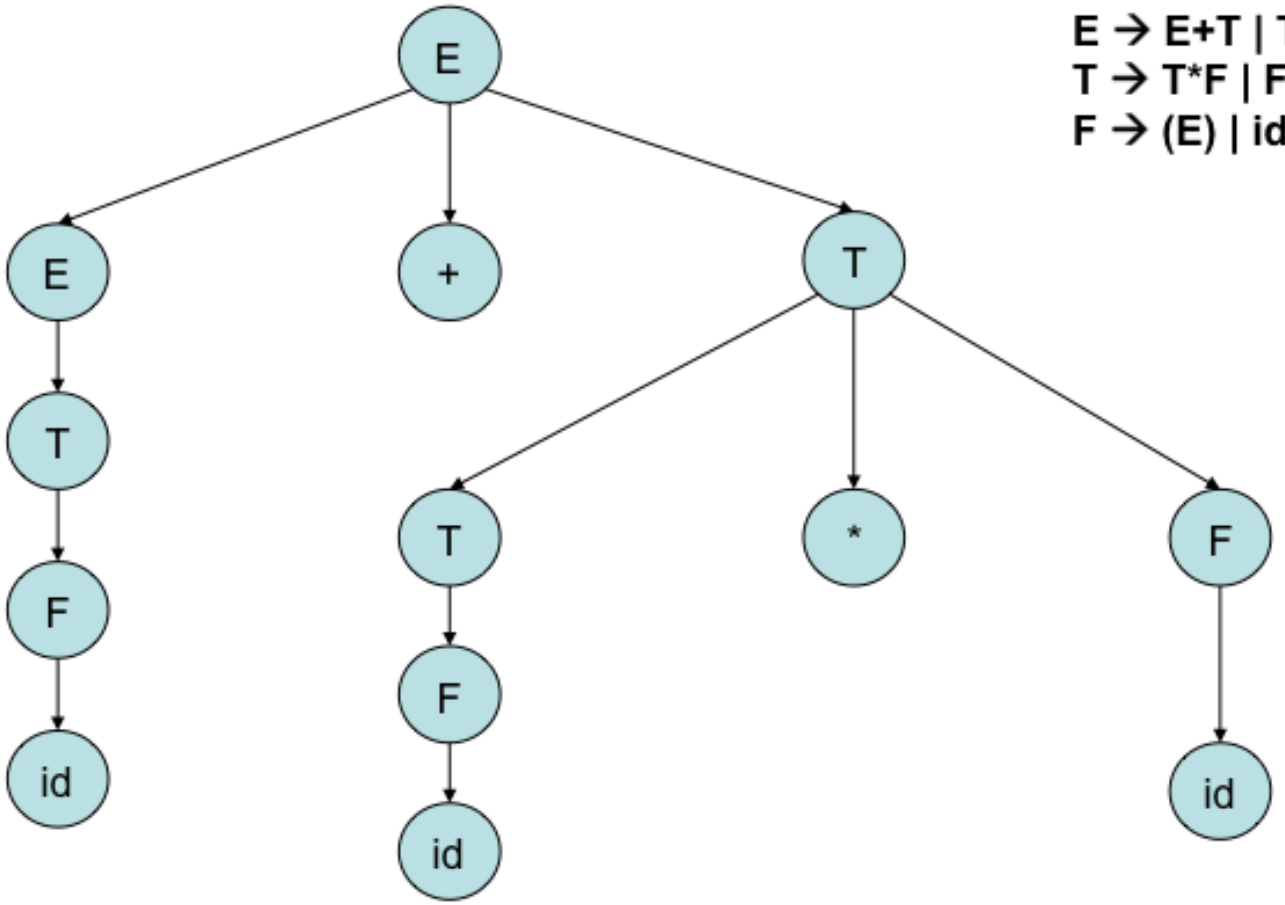


E => E+E => E*E+E => id*E+E => id*id+E => id*id+id

E => E*E => id*E => id*E+E => id*id+E => id*id+id

E → E+E | E*E | (E) | id

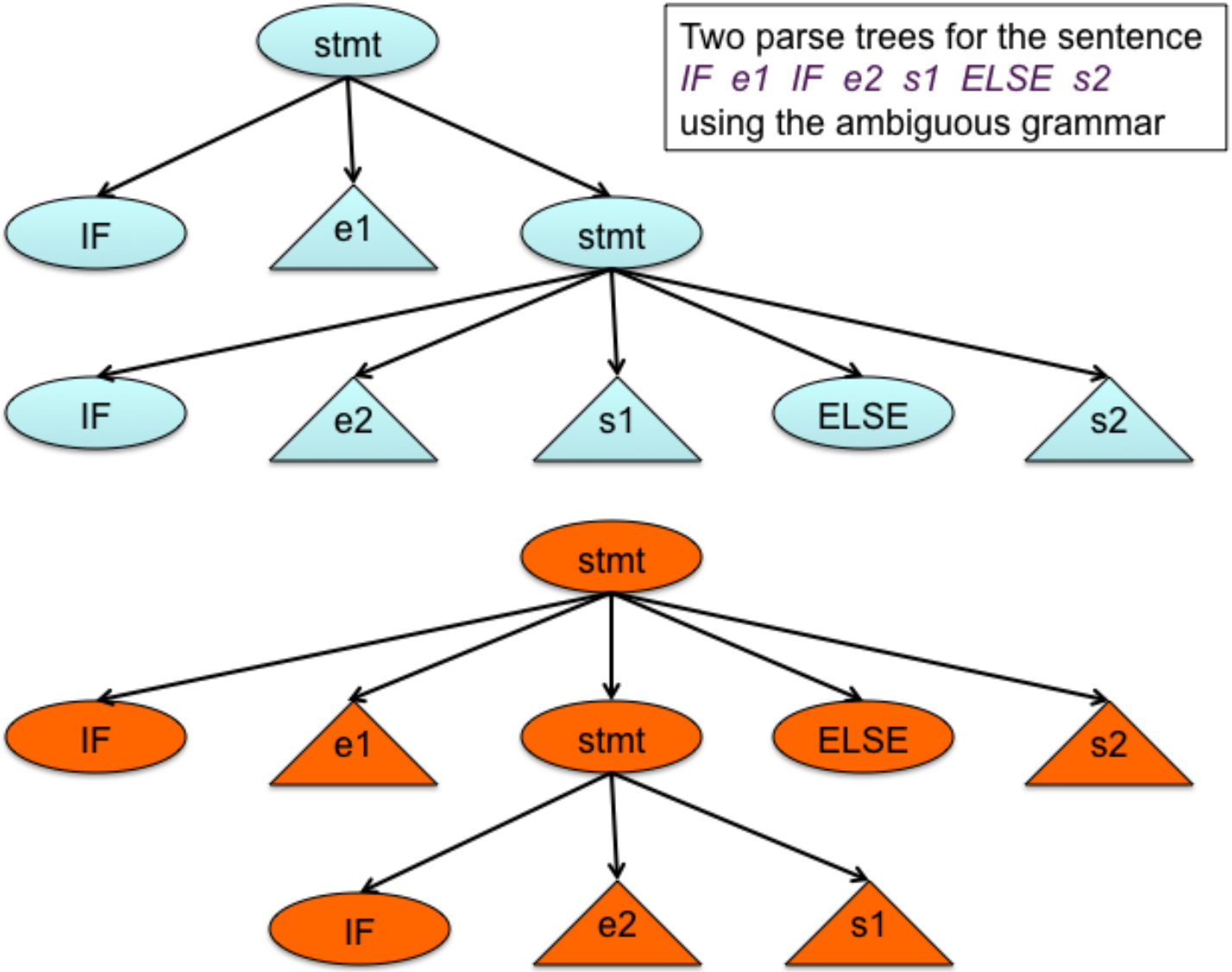
Equivalent Unambiguous Grammar



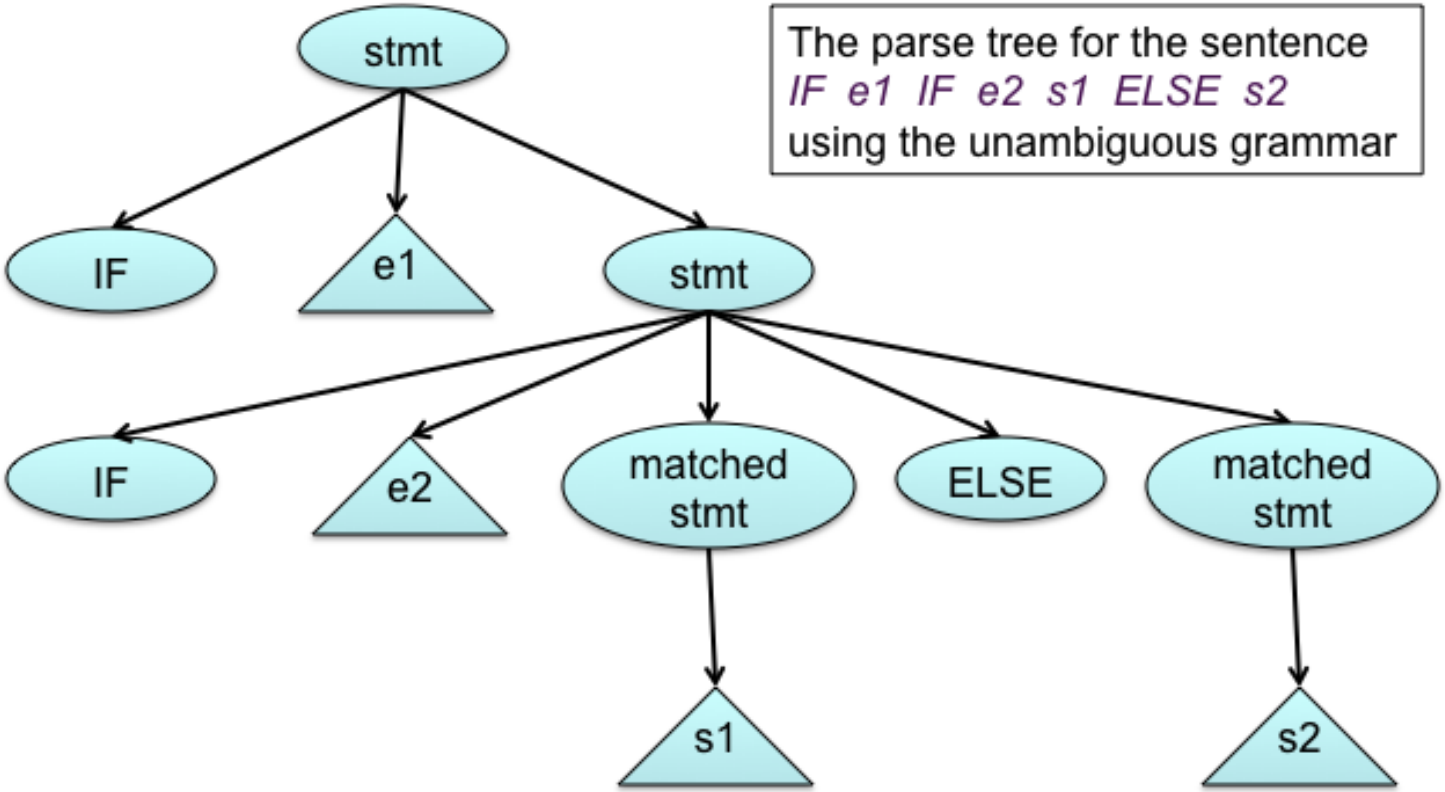
E => E+T => T+T => F+T => id+T => id+T*F => id+F*F => id+id*F => id+id*id

E => T*F => F*F => (E)*F => (E+T)*F => (T+T)*F => (F+T)*F => (id+T)*F
=> (id+F)*id => (id+id)*F => (id+id)*id

Ambiguity Example 2



Ambiguity Example 2 (contd.)



The parse tree for the sentence
IF e1 IF e2 s1 ELSE s2
using the unambiguous grammar

s→ IF e s | IF e ms ELSE s
ms → IF e ms ELSE ms | other_s

Fragment of C-Grammar (Statements)

```
program --> VOID MAIN '(' ')' compound_stmt
compound_stmt --> '{' '}' | '{' stmt_list '}'
          | '{' declaration_list stmt_list '}'
stmt_list --> stmt | stmt_list stmt
stmt --> compound_stmt| expression_stmt
             | if_stmt | while_stmt
expression_stmt --> ';'| expression ';'
if_stmt --> IF '(' expression ')' stmt
   | IF '(' expression ')' stmt ELSE stmt
while_stmt --> WHILE '(' expression ')' stmt
expression --> assignment_expr
              | expression ',' assignment_expr
```

Fragment of C-Grammar (Expressions)

```
assignment_expr --> logical_or_expr
        | unary_expr  assign_op  assignment_expr
assign_op --> '='| MUL_ASSIGN| DIV_ASSIGN
                | ADD_ASSIGN| SUB_ASSIGN
                | AND_ASSIGN| OR_ASSIGN
unary_expr --> primary_expr
        | unary_operator   unary_expr
unary_operator --> '+'| '-'| '!'
primary_expr --> ID| NUM| '(' expression ')'
logical_or_expr --> logical_and_expr
        | logical_or_expr   OR_OP logical_and_expr
logical_and_expr --> equality_expr
        | logical_and_expr   AND_OP  equality_expr
equality_expr --> relational_expr
            | equality_expr EQ_OP relational_expr
            | equality_expr NE_OP relational_expr
```

Fragment of C-Grammar (Expressions and Declarations)

```
relational_expr --> add_expr
                  | relational_expr '<' add_expr
                  | relational_expr '>' add_expr
                  | relational_expr LE_OP add_expr
                  | relational_expr GE_OP add_expr
add_expr --> mult_expr| add_expr '+' mult_expr
                      | add_expr '-' mult_expr
mult_expr --> unary_expr| mult_expr '*' unary_expr
                        | mult_expr '/' unary_expr
declarationlist --> declaration
                  | declarationlist  declaration
declaration --> type  idlist ';'
idlist --> idlist ',' ID | ID
type --> INT_TYPE | FLOAT_TYPE | CHAR_TYPE
```

Pushdown Automata

A PDA M is a system $(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$, where

- Q is a finite set of states
- Σ is the input alphabet
- Γ is the stack alphabet
- $q_0 \in Q$ is the start state
- $z_0 \in \Gamma$ is the start symbol on stack (initialization)
- $F \subseteq Q$ is the set of final states
- δ is the transition function, $Q \times \Sigma \cup \{\epsilon\} \times \Gamma$ to finite subsets of $Q \times \Gamma^*$

A typical entry of δ is given by
$\delta(q, a, z) = \{(p_1, \gamma_1), ((p_2, \gamma_2), ..., (p_m, \gamma_m)\}$
The PDA in state q, with input symbol a and top-of-stack symbol z, can enter any of the states p_i, replace the symbol z by the string γ_i, and advance the input head by one symbol.

Pushdown Automata (contd.)

- The leftmost symbol of γ_i will be the new top of stack
- a in the above function δ could be ϵ, in which case, the input symbol is not used and the input head is not advanced
- For a PDA M, we define $L(M)$, the language accepted by M **by final state**, to be
  $L(M) = \{w \mid (q_0, w, Z_0) \vdash^* (p, \epsilon, \gamma)$, for some $p \in F$ and $\gamma \in \Gamma^*\}$
- We define $N(M)$, the language accepted by M **by empty stack**, to be
  $N(M) = \{w \mid (q_0, w, Z_0) \vdash^* (p, \epsilon, \epsilon)$, for some $p \in Q$
- When acceptance is by empty stack, the set of final states is irrelevant, and usually, we set $F = \phi$

PDA - Examples

- $L = \{0^n 1^n \mid n \geq 0\}$
  $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \{Z, 0\}, \delta, q_0, Z, \{q_0\})$, where δ is defined as follows
  $\delta(q_0, 0, Z) = \{(q_1, 0Z)\}$, $\delta(q_1, 0, 0) = \{(q_1, 00)\}$,
  $\delta(q_1, 1, 0) = \{(q_2, \epsilon)\}$, $\delta(q_2, 1, 0) = \{(q_2, \epsilon)\}$,
  $\delta(q_2, \epsilon, Z) = \{(q_0, \epsilon)\}$
- $(q_0, 0011, Z) \vdash (q_1, 011, 0Z) \vdash (q_1, 11, 00Z) \vdash (q_2, 1, 0Z) \vdash (q_2, \epsilon, Z) \vdash (q_0, \epsilon, \epsilon)$
- $(q_0, 001, Z) \vdash (q_1, 01, 0Z) \vdash (q_1, 1, 00Z) \vdash (q_2, \epsilon, 0Z) \vdash$ *error*
- $(q_0, 010, Z) \vdash (q_1, 10, 0Z) \vdash (q_2, 0, Z) \vdash$ *error*

Syntax Analysis:

Context-free Grammars, Pushdown Automata and Parsing
Part - 2

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is syntax analysis? (covered in lecture 1)
- Specification of programming languages: context-free grammars (covered in lecture 1)
- Parsing context-free languages: push-down automata
- Top-down parsing: LL(1) and recursive-descent parsing
- Bottom-up parsing: LR-parsing

Pushdown Automata

A PDA M is a system $(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$, where

- Q is a finite set of states
- Σ is the input alphabet
- Γ is the stack alphabet
- $q_0 \in Q$ is the start state
- $z_0 \in \Gamma$ is the start symbol on stack (initialization)
- $F \subseteq Q$ is the set of final states
- δ is the transition function, $Q \times \Sigma \cup \{\epsilon\} \times \Gamma$ to finite subsets of $Q \times \Gamma^*$

A typical entry of δ is given by
$\delta(q, a, z) = \{(p_1, \gamma_1), ((p_2, \gamma_2), ..., (p_m, \gamma_m)\}$
The PDA in state q, with input symbol a and top-of-stack symbol z, can enter any of the states p_i, replace the symbol z by the string γ_i, and advance the input head by one symbol.

Pushdown Automata (contd.)

- The leftmost symbol of γ_i will be the new top of stack
- a in the above function δ could be ϵ, in which case, the input symbol is not used and the input head is not advanced
- For a PDA M, we define $L(M)$, the language accepted by M **by final state**, to be
  $L(M) = \{w \mid (q_0, w, Z_0) \vdash^* (p, \epsilon, \gamma)$, for some $p \in F$ and $\gamma \in \Gamma^*\}$
- We define $N(M)$, the language accepted by M **by empty stack**, to be
  $N(M) = \{w \mid (q_0, w, Z_0) \vdash^* (p, \epsilon, \epsilon)$, for some $p \in Q$
- When acceptance is by empty stack, the set of final states is irrelevant, and usually, we set $F = \phi$

PDA - Examples

- $L = \{0^n 1^n \mid n \geq 0\}$
  $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \{Z, 0\}, \delta, q_0, Z, \{q_0\})$, where δ is defined as follows
  $\delta(q_0, 0, Z) = \{(q_1, 0Z)\}$, $\delta(q_1, 0, 0) = \{(q_1, 00)\}$,
  $\delta(q_1, 1, 0) = \{(q_2, \epsilon)\}$, $\delta(q_2, 1, 0) = \{(q_2, \epsilon)\}$,
  $\delta(q_2, \epsilon, Z) = \{(q_0, \epsilon)\}$
- $(q_0, 0011, Z) \vdash (q_1, 011, 0Z) \vdash (q_1, 11, 00Z) \vdash (q_2, 1, 0Z) \vdash (q_2, \epsilon, Z) \vdash (q_0, \epsilon, \epsilon)$
- $(q_0, 001, Z) \vdash (q_1, 01, 0Z) \vdash (q_1, 1, 00Z) \vdash (q_2, \epsilon, 0Z) \vdash$ *error*
- $(q_0, 010, Z) \vdash (q_1, 10, 0Z) \vdash (q_2, 0, Z) \vdash$ *error*

PDA - Examples (contd.)

- $L = \{ww^R \mid w \in \{a, b\}^+\}$
  $M = (\{q_0, q_1, q_2\}, \{a, b\}, \{Z, a, b\}, \delta, q_0, Z, \{q_2\})$, where δ is defined as follows
  $\delta(q_0, a, Z) = \{(q_0, aZ)\}$, $\delta(q_0, b, Z) = \{(q_0, bZ)\}$,
  $\delta(q_0, a, a) = \{(q_0, aa), (q_1, \epsilon)\}$, $\delta(q_0, a, b) = \{(q_0, ab)\}$,
  $\delta(q_0, b, a) = \{(q_0, ba)\}$, $\delta(q_0, b, b) = \{(q_0, bb), (q_1, \epsilon)\}$,
  $\delta(q_1, a, a) = \{(q_1, \epsilon)\}$, $\delta(q_1, b, b) = \{(q_1, \epsilon)\}$,
  $\delta(q_1, \epsilon, Z) = \{(q_2, \epsilon)\}$
- $(q_0, abba, Z) \vdash (q_0, bba, aZ) \vdash (q_0, ba, baZ) \vdash (q_1, a, aZ) \vdash (q_1, \epsilon, Z) \vdash (q_2, \epsilon, \epsilon)$
- $(q_0, aaa, Z) \vdash (q_0, aa, aZ) \vdash (q_0, a, aaZ) \vdash (q_1, \epsilon, aZ) \vdash$ *error*
- $(q_0, aaa, Z) \vdash (q_0, aa, aZ) \vdash (q_1, a, Z) \vdash$ *error*

Nondeterministic and Deterministic PDA

- Just as in the case of NFA and DFA, PDA also have two versions: NPDA and DPDA
- However, NPDA are strictly more powerful than the DPDA
- For example, the language, $L = \{ww^R \mid w \in \{a, b\}^+\}$ can be recognized only by an NPDA and not by any DPDA
- In the same breath, the language, $L = \{wcw^R \mid w \in \{a, b\}^+\}$, can be recognized by a DPDA
- In practice we need DPDA, since they have exactly one possible move at any instant
- Our parsers are all DPDA

Parsing

- Parsing is the process of constructing a parse tree for a sentence generated by a given grammar
- If there are no restrictions on the language and the form of grammar used, parsers for context-free languages require $O(n^3)$ time (n being the length of the string parsed)
  - Cocke-Younger-Kasami's algorithm
  - Earley's algorithm
- Subsets of context-free languages typically require $O(n)$ time
  - Predictive parsing using $LL(1)$ grammars (top-down parsing method)
  - Shift-Reduce parsing using $LR(1)$ grammars (bottom-up parsing method)

Top-Down Parsing using LL Grammars

- Top-down parsing using predictive parsing, traces the left-most derivation of the string while constructing the parse tree
- Starts from the start symbol of the grammar, and “predicts” the next production used in the derivation
- Such “prediction” is aided by parsing tables (constructed off-line)
- The next production to be used in the derivation is determined using the next input symbol to lookup the parsing table (look-ahead symbol)
- Placing restrictions on the grammar ensures that no slot in the parsing table contains more than one production
- At the time of parsing table constrcution, if two productions become eligible to be placed in the same slot of the parsing table, the grammar is declared unfit for predictive parsing

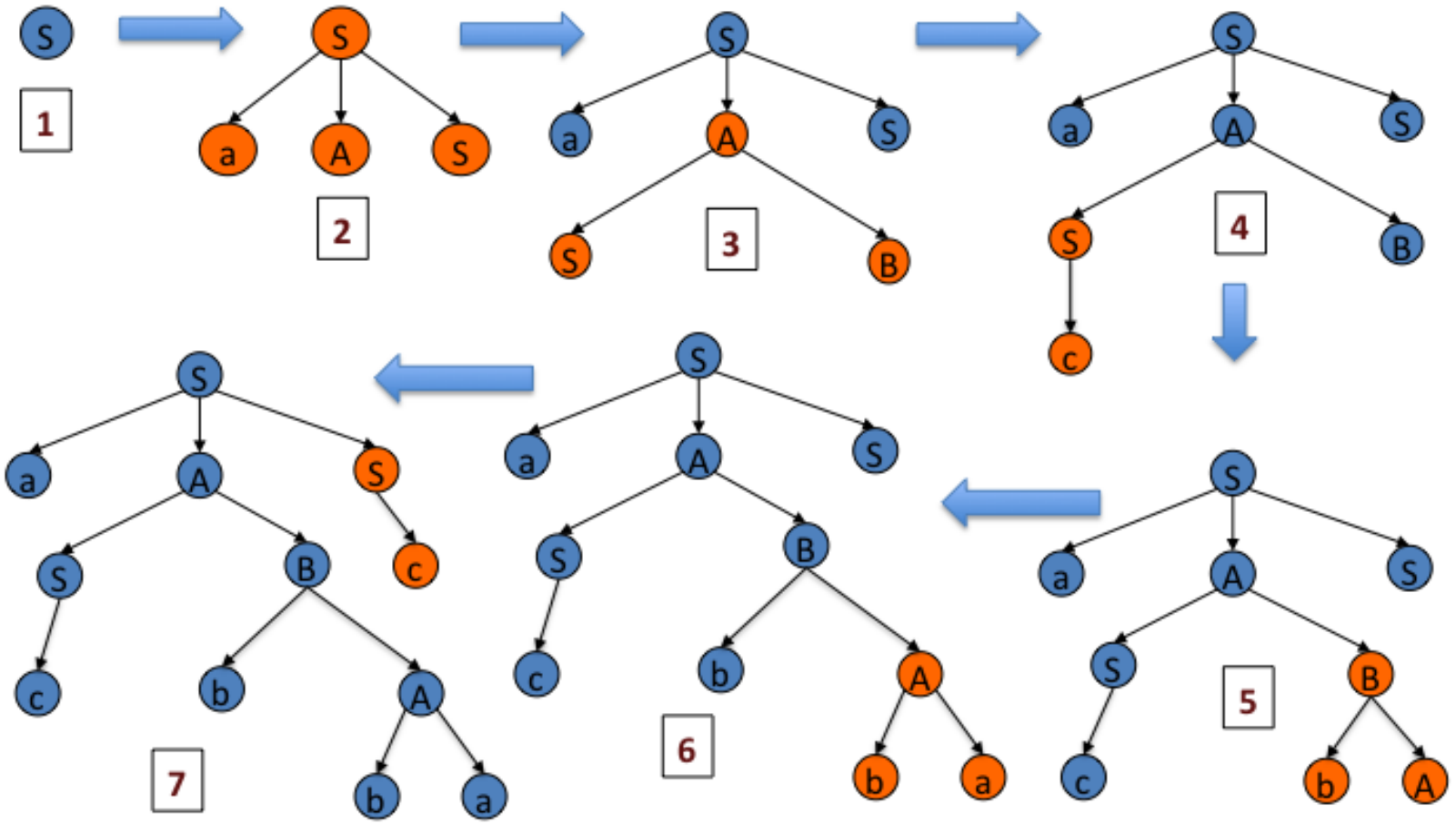
Top-Down LL-Parsing Example

S → aAS | c
A → ba | SB
B → bA | S

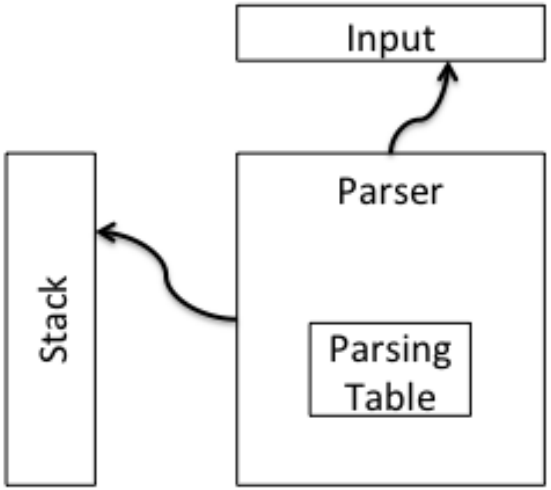
Leftmost derivation of the string *acbbac*

S => aAS => aSBS => acBS => acbAS => acbbaS => acbbac

1 2 3 4 5 6 7



LL(1) Parsing Algorithm



Initial configuration: Stack = S, Input = w$,
where, S = start symbol, $ = end of file marker
repeat {
  let X be the top stack symbol;
  let a be the next input symbol /*may be $*/;
  if X is a terminal symbol or $ then
    if $X == a$ then {
      pop X from Stack;
      remove a from input;
    } else ERROR();
  else /* X is a non-terminal symbol */
    if $M[X,a] == X \rightarrow Y_1 Y_2 \ldots Y_k$ then {
      pop X from Stack;
      push $Y_k, Y_{k-1}, \ldots, Y_1$ onto Stack;
        (Y_1 on top)
    }
} until Stack has emptied;

LL(1) Parsing Algorithm Example

Grammar

S' → S$

S → aAS | c

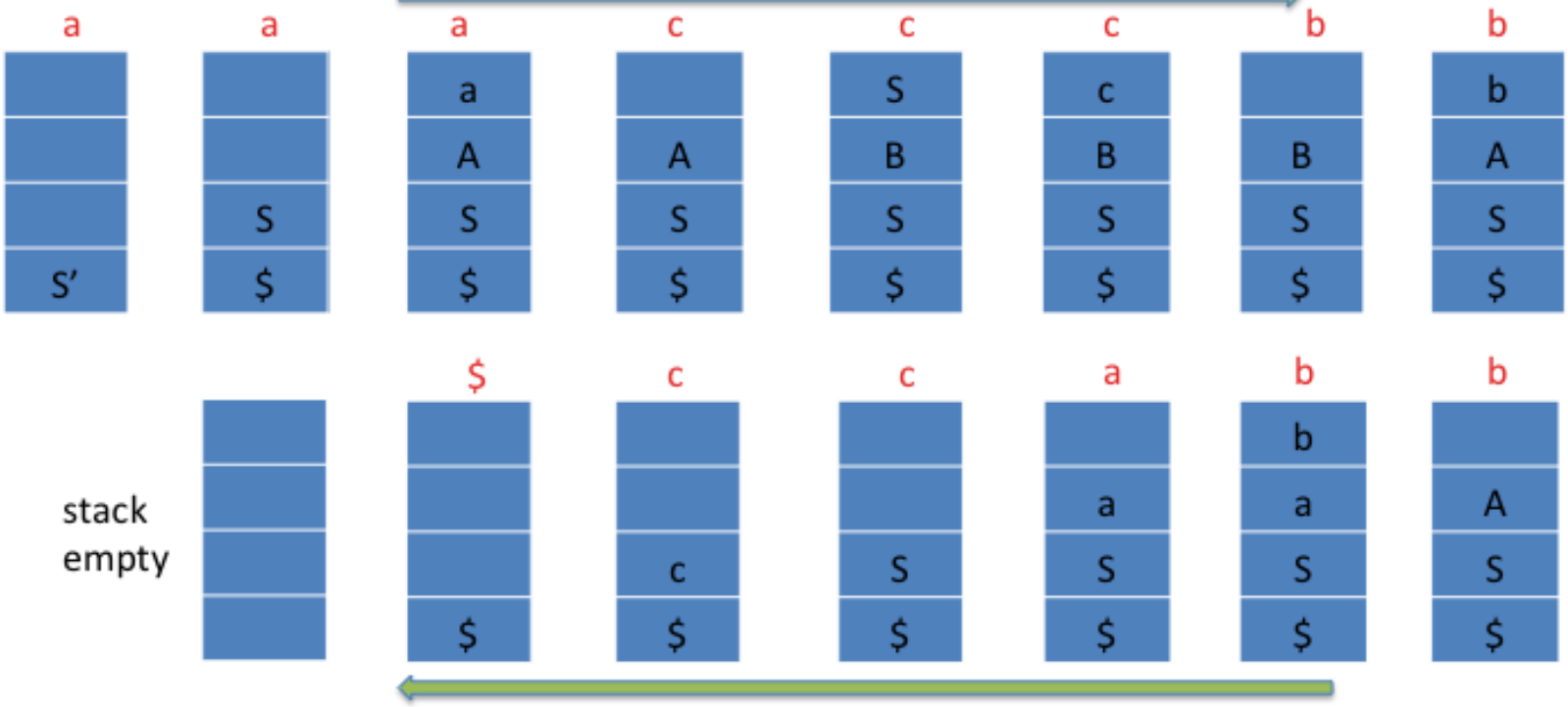
A → ba | SB

B → bA | S

string: *acbbac*

LL(1) Parsing Table

	a	b	c	$
S'	S' → S$		S' → S$	
S	S → aAS		S → c	
A	A → SB	A → ba	A → SB	
B	B → S	B → bA	B → S	



Strong LL(k) Grammars

Let the given grammar be G

- Input is extended with k symbols, $\$^k$, k is the lookahead of the grammar
- Introduce a new nonterminal S', and a production, $S' \rightarrow S\$^k$, where S is the start symbol of the given grammar
- Consider leftmost derivations only and assume that the grammar has no *useless symbols*
- A production $A \rightarrow \alpha$ in G is called a *strong LL*(k) production, if in G
  $S' \Rightarrow^* wA\gamma \Rightarrow w\alpha\gamma \Rightarrow^* wzy$
  $S' \Rightarrow^* w'A\delta \Rightarrow w'\beta\delta \Rightarrow^* w'zx$
  $|z| = k,\ z \in \Sigma^*, w$ *and* $w' \in \Sigma^*$, then $\alpha = \beta$
- A grammar (nonterminal) is strong LL(k) if all its productions are strong LL(k)

Strong LL(k) Grammars (contd.)

- Strong LL(k) grammars do not allow different productions of the same nonterminal to be used even in two different derivations, if the first k symbols of the strings produced by $\alpha\gamma$ and $\beta\delta$ are the same
- Example: $S \rightarrow Abc|aAcb,\ \ A \rightarrow \epsilon|b|c$
  S is a strong LL(1) nonterminal
  - $S' \Rightarrow S\$ \Rightarrow Abc\$ \Rightarrow bc\$,\ bbc\$$, and $cbc\$$, on application of the productions, $A \rightarrow \epsilon$, $A \rightarrow b$, and, $A \rightarrow c$, respectively. $z = b,\ b$, or c, respectively
  - $S' \Rightarrow S\$ \Rightarrow aAcb\$ \Rightarrow acb\$,\ abcb\$$, and $accb\$$, on application of the productions, $A \rightarrow \epsilon$, $A \rightarrow b$, and, $A \rightarrow c$, respectively. $z = a$, in all three cases
  - In this case, $w = w' = \epsilon$, $\alpha = Abc$, $\beta = aAcb$, but z is different in the two derivations, in all the derived strings
  - Hence the nonterminal S is strong LL(1)

Strong LL(k) Grammars (contd.)

A is not strong LL(1)

- $S' \Rightarrow^* Abc\$ \Rightarrow \underline{b}c\$$, $w = \epsilon$, $z = b$, $\alpha = \epsilon$ $(A \rightarrow \epsilon)$
  $S' \Rightarrow^* Abc\$ \Rightarrow \underline{b}bc\$$, $w' = \epsilon$, $z = b$, $\beta = b$ $(A \rightarrow b)$
- Even though the lookaheads are the same ($z = b$), $\alpha \neq \beta$, and therefore, the grammar is not strong LL(1)

A is not strong LL(2)

- $S' \Rightarrow^* Abc\$ \Rightarrow \underline{bc}\$$, $w = \epsilon$, $z = bc$, $\alpha = \epsilon$ $(A \rightarrow \epsilon)$
  $S' \Rightarrow^* aAcb\$ \Rightarrow a\underline{bc}b\$$, $w' = a$, $z = bc$, $\beta = b$ $(A \rightarrow b)$
- Even though the lookaheads are the same ($z = bc$), $\alpha \neq \beta$, and therefore, the grammar is not strong LL(2)

A is strong LL(3) because all the six strings (*bc$, bbc, cbc, cb$, bcb, ccb*) can be distinguished using 3-symbol lookahead (details are for home work)

Testable Conditions for LL(1)

- We call strong LL(1) as LL(1) from now on and we will not consider lookaheads longer than 1
- The classical condition for LL(1) property uses *FIRST* and *FOLLOW* sets
- If α is any string of grammar symbols ($\alpha \in (N \cup T)^*$), then
  $FIRST(\alpha) = \{a \mid a \in T, \text{ and } \alpha \Rightarrow^* ax, x \in T^*\}$
  $FIRST(\epsilon) = \{\epsilon\}$
- If A is any nonterminal, then
  $FOLLOW(A) = \{a \mid S \Rightarrow^* \alpha A a \beta, \alpha, \beta \in (N \cup T)^*, a \in T \cup \{\$\}\}$
- $FIRST(\alpha)$ is determined by α alone, but $FOLLOW(A)$ is determined by the "context" of A, i.e., the derivations in which A occurs

FIRST and *FOLLOW* Computation Example

- Consider the following grammar
  $S' \rightarrow S\$,\ S \rightarrow aAS \mid c,\ A \rightarrow ba \mid SB,\ B \rightarrow bA \mid S$
- $FIRST(S') = FIRST(S) = \{a, c\}$ because
  $S' \Rightarrow S\$ \Rightarrow \underline{c}\$$, and $S' \Rightarrow S\$ \Rightarrow \underline{a}AS\$ \Rightarrow \underline{a}baS\$ \Rightarrow \underline{a}bac\$$
- $FIRST(A) = \{a, b, c\}$ because
  $A \Rightarrow \underline{b}a$, and $A \Rightarrow SB$, and therefore all symbols in $FIRST(S)$ are in $FIRST(A)$
- $FOLLOW(S) = \{a, b, c, \$\}$ because
  $S' \Rightarrow \underline{S\$}$,
  $S' \Rightarrow^* aAS\$ \Rightarrow a\underline{S}BS\$ \Rightarrow aS\underline{b}AS\$$,
  $S' \Rightarrow^* a\underline{S}BS\$ \Rightarrow a\underline{S}SS\$ \Rightarrow aS\underline{a}ASS\$$,
  $S' \Rightarrow^* a\underline{S}SS\$ \Rightarrow aS\underline{c}S\$$
- $FOLLOW(A) = \{a, c\}$ because
  $S' \Rightarrow^* a\underline{A}S\$ \Rightarrow aA\underline{a}AS\$$,
  $S' \Rightarrow^* a\underline{A}S\$ \Rightarrow aA\underline{c}$

Computation of *FIRST*: Terminals and Nonterminals

{
  for each ($a \in T$) FIRST(a) = $\{a\}$; FIRST(ϵ) = $\{\epsilon\}$;
  for each ($A \in N$) FIRST(A) = $\emptyset$;
  while (FIRST sets are still changing) {
      for each production p {
          Let p be the production $A \rightarrow X_1 X_2 ... X_n$;
          FIRST(A) = FIRST(A) $\cup$ (FIRST(X_1) - $\{\epsilon\}$);
          i = 1;
          while ($\epsilon \in$ FIRST(X_i) && $i \leq n - 1$) {
              FIRST(A) = FIRST(A) $\cup$ (FIRST($X_{i+1} - \{\epsilon\}$); $i++$;
          }
          if ($i == n$) && ($\epsilon \in$ FIRST(X_n))
              FIRST(A) = FIRST(A) $\cup \{\epsilon\}$
  }
}

Syntax Analysis:

Context-free Grammars, Pushdown Automata and Parsing
Part - 3

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is syntax analysis? (covered in lecture 1)
- Specification of programming languages: context-free grammars (covered in lecture 1)
- Parsing context-free languages: push-down automata (covered in lectures 1 and 2)
- Top-down parsing: LL(1) and recursive-descent parsing
- Bottom-up parsing: LR-parsing

Testable Conditions for LL(1)

- We call strong LL(1) as LL(1) from now on and we will not consider lookaheads longer than 1
- The classical condition for LL(1) property uses *FIRST* and *FOLLOW* sets
- If α is any string of grammar symbols ($\alpha \in (N \cup T)^*$), then
  $FIRST(\alpha) = \{a \mid a \in T, \text{ and } \alpha \Rightarrow^* ax, x \in T^*\}$
  $FIRST(\epsilon) = \{\epsilon\}$
- If A is any nonterminal, then
  $FOLLOW(A) = \{a \mid S \Rightarrow^* \alpha A a \beta, \alpha, \beta \in (N \cup T)^*, a \in T \cup \{\$\}\}$
- $FIRST(\alpha)$ is determined by α alone, but $FOLLOW(A)$ is determined by the "context" of A, i.e., the derivations in which A occurs

FIRST and *FOLLOW* Computation Example

- Consider the following grammar
  $S' \rightarrow S\$,\ S \rightarrow aAS \mid c,\ A \rightarrow ba \mid SB,\ B \rightarrow bA \mid S$
- $FIRST(S') = FIRST(S) = \{a, c\}$ because
  $S' \Rightarrow S\$ \Rightarrow \underline{c}\$$, and $S' \Rightarrow S\$ \Rightarrow \underline{a}AS\$ \Rightarrow \underline{a}baS\$ \Rightarrow \underline{a}bac\$$
- $FIRST(A) = \{a, b, c\}$ because
  $A \Rightarrow \underline{b}a$, and $A \Rightarrow SB$, and therefore all symbols in $FIRST(S)$ are in $FIRST(A)$
- $FOLLOW(S) = \{a, b, c, \$\}$ because
  $S' \Rightarrow \underline{S\$}$,
  $S' \Rightarrow^* aAS\$ \Rightarrow a\underline{S}BS\$ \Rightarrow aS\underline{b}AS\$$,
  $S' \Rightarrow^* a\underline{S}BS\$ \Rightarrow a\underline{S}SS\$ \Rightarrow aS\underline{a}ASS\$$,
  $S' \Rightarrow^* a\underline{S}SS\$ \Rightarrow aS\underline{c}S\$$
- $FOLLOW(A) = \{a, c\}$ because
  $S' \Rightarrow^* a\underline{A}S\$ \Rightarrow aA\underline{a}AS\$$,
  $S' \Rightarrow^* a\underline{A}S\$ \Rightarrow aA\underline{c}$

Computation of *FIRST*: Terminals and Nonterminals

{
  for each ($a \in T$) FIRST(a) = $\{a\}$; FIRST(ϵ) = $\{\epsilon\}$;
  for each ($A \in N$) FIRST(A) = $\emptyset$;
  while (FIRST sets are still changing) {
      for each production p {
          Let p be the production $A \rightarrow X_1 X_2 ... X_n$;
          FIRST(A) = FIRST(A) $\cup$ (FIRST(X_1) - $\{\epsilon\}$);
          i = 1;
          while ($\epsilon \in$ FIRST(X_i) && $i \leq n - 1$) {
              FIRST(A) = FIRST(A) $\cup$ (FIRST($X_{i+1} - \{\epsilon\}$); $i++$;
          }
          if ($i == n$) && ($\epsilon \in$ FIRST(X_n))
              FIRST(A) = FIRST(A) $\cup \{\epsilon\}$
  }
}

Computation of $FIRST(\beta)$: β, a string of Grammar Symbols

{ /* It is assumed that FIRST sets of terminals and nonterminals are already available /*
FIRST(β) = $\emptyset$;
while (FIRST sets are still changing) {
Let β be the string $X_1 X_2 ... X_n$;
FIRST(β) = FIRST(β) $\cup$ (FIRST(X_1) - {ϵ});
i = 1;
while ($\epsilon \in$ FIRST(X_i) && $i \leq n - 1$) {
FIRST(β) = FIRST(β) $\cup$ (FIRST($X_{i+1} - \{\epsilon\}$); $i++$;
}
if ($i == n$) && ($\epsilon \in$ FIRST(X_n))
FIRST(β) = FIRST(β) $\cup \{\epsilon\}$
}
}

FIRST Computation: Algorithm Trace - 1

- Consider the following grammar
  $S' \rightarrow S\$,\ S \rightarrow aAS \mid \epsilon,\ A \rightarrow ba \mid SB,\ B \rightarrow cA \mid S$
- Initially, FIRST(S) = FIRST(A) = FIRST(B) = $\emptyset$
- Iteration 1
  - FIRST(S) = $\{a, \epsilon\}$ from the productions $S \rightarrow aAS \mid \epsilon$
  - FIRST(A) = $\{b\} \cup$ FIRST(S) - $\{\epsilon\} \cup$ FIRST(B) - $\{\epsilon\}$ = $\{b, a\}$
    from the productions $A \rightarrow ba \mid SB$
    (since $\epsilon \in$ FIRST(S), FIRST(B) is also included;
    since FIRST(B)=ϕ, ϵ is not included)
  - FIRST(B) = $\{c\} \cup$ FIRST(S) - $\{\epsilon\} \cup \{\epsilon\}$ = $\{c, a, \epsilon\}$
    from the productions $B \rightarrow cA \mid S$
    (ϵ is included because $\epsilon \in$ FIRST(S))

FIRST Computation: Algorithm Trace - 2

- The grammar is
  $S' \rightarrow S\$,\ S \rightarrow aAS \mid \epsilon,\ A \rightarrow ba \mid SB,\ B \rightarrow cA \mid S$
- From the first iteration,
  FIRST(S) = $\{a, \epsilon\}$, FIRST(A) = $\{b, a\}$, FIRST(B) = $\{c, a, \epsilon\}$
- Iteration 2
  (values stabilize and do not change in iteration 3)
  - FIRST(S) = $\{a, \epsilon\}$ (no change from iteration 1)
  - FIRST(A) = $\{b\} \cup$ FIRST(S) - $\{\epsilon\} \cup$ FIRST(B) - $\{\epsilon\} \cup \{\epsilon\}$
    = $\{b, a, c, \epsilon\}$ (changed!)
  - FIRST(B) = $\{c, a, \epsilon\}$ (no change from iteration 1)

Computation of *FOLLOW*

{ for each ($X \in N \cup T$) FOLLOW(X) = $\emptyset$;
  FOLLOW(S) = {\$}; /* S is the start symbol of the grammar */
  repeat {
    for each production $A \rightarrow X_1 X_2 ... X_n$ {/* $X_i \neq \epsilon$ */
      FOLLOW(X_n) = FOLLOW(X_n) $\cup$ FOLLOW(A);
      REST = FOLLOW(A);
      for $i = n$ downto 2 {
        if ($\epsilon \in$ FIRST(X_i)) { FOLLOW(X_{i-1}) =
            FOLLOW(X_{i-1}) $\cup$ (FIRST (X_i) $-$ {ϵ})$\cup$ REST;
            REST = FOLLOW(X_{i-1});
        } else { FOLLOW(X_{i-1}) = FOLLOW(X_{i-1}) $\cup$ FIRST (X_i) ;
                 REST = FOLLOW(X_{i-1}); }
      }
    }
  } until no FOLLOW set has changed
}

FOLLOW Computation: Algorithm Trace

- Consider the following grammar
  $S' \rightarrow S\$,\ S \rightarrow aAS \mid \epsilon,\ A \rightarrow ba \mid SB,\ B \rightarrow cA \mid S$
- Initially, $follow(S) = \{\$\}$; $follow(A) = follow(B) = \emptyset$
  $first(S) = \{a, \epsilon\}$; $first(A) = \{a, b, c, \epsilon\}$; $first(B) = \{a, c, \epsilon\}$;
- Iteration 1 /* In the following, $x \cup = y$ means $x = x \cup y$ */
  - $S \rightarrow aAS$: $follow(S) \cup = \{\$\}$; $rest = follow(S) = \{\$\}$
    $follow(A) \cup = (first(S) - \{\epsilon\}) \cup rest = \{a, \$\}$
  - $A \rightarrow SB$: $follow(B) \cup = follow(A) = \{a, \$\}$
    $rest = follow(A) = \{a,\$\}$
    $follow(S) \cup = (first(B) - \{\epsilon\}) \cup rest = \{a, c, \$\}$
  - $B \rightarrow cA$: $follow(A) \cup = follow(B) = \{a,\$\}$
  - $B \rightarrow S$: $follow(S) \cup = follow(B) = \{a, c,\$\}$
  - At the end of iteration 1
    $follow(S) = \{a, c,\$\}$; $follow(A) = follow(B) = \{a, \$\}$

FOLLOW Computation: Algorithm Trace (contd.)

- $first(S) = \{a, \epsilon\}$; $first(A) = \{a, b, c, \epsilon\}$; $first(B) = \{a, c, \epsilon\}$;
- At the end of iteration 1
  $follow(S) = \{a, c, \$\}$; $follow(A) = follow(B) = \{a, \$\}$
- Iteration 2
- $S \rightarrow aAS$: $follow(S) \cup = \{a, c, \$\}$;
  $rest = follow(S) = \{a, c, \$\}$
  $follow(A) \cup = (first(S) - \{\epsilon\}) \cup rest = \{a, c, \$\}$ (changed!)
- $A \rightarrow SB$: $follow(B) \cup = follow(A) = \{a, c, \$\}$ (changed!)
  $rest = follow(A) = \{a, c, \$\}$
  $follow(S) \cup = (first(B) - \{\epsilon\}) \cup rest = \{a, c, \$\}$ (no change)
- At the end of iteration 2
  $follow(S) = follow(A) = follow(B) = \{a, c, \$\}$;
- The *follow* sets do not change any further

LL(1) Conditions

- Let G be a context-free grammar
- G is LL(1) iff for every pair of productions $A \rightarrow \alpha$ and $A \rightarrow \beta$, the following condition holds
  - $dirsymb(\alpha) \cap dirsymb(\beta) = \emptyset$, where
    $dirsymb(\gamma)$ = if $(\epsilon \in first(\gamma))$ then
    $((first(\gamma) - \{\epsilon\}) \cup follow(A))$ else $first(\gamma)$
    (γ stands for α or β)
  - *dirsymb* stands for "direction symbol set"
- An equivalent formulation (as in ALSU's book) is as below
  - $first(\alpha.follow(A)) \cap first(\beta.follow(A)) = \emptyset$
- Construction of the LL(1) parsing table

for each production $A \rightarrow \alpha$
  for each symbol $s \in dirsymb(\alpha)$
  /* s may be either a terminal symbol or $ */
    add $A \rightarrow \alpha$ to $LLPT[A, s]$
Make each undefined entry of *LLPT* as *error*

LL(1) Table Construction using *FIRST* and *FOLLOW*

for each production $A \rightarrow \alpha$
  for each terminal symbol $a \in first(\alpha)$
    add $A \rightarrow \alpha$ to $LLPT[A, a]$
  if $\epsilon \in first(\alpha)$ {
    for each terminal symbol $b \in follow(A)$
      add $A \rightarrow \alpha$ to $LLPT[A, b]$
    if $\$ \in follow(A)$
      add $A \rightarrow \alpha$ to $LLPT[A, \$]$
  }
Make each undefined entry of *LLPT* as *error*

- After the construction of the LL(1) table is complete (following any of the two methods), if any slot in the LL(1) table has two or more productions, then the grammar is NOT LL(1)

Simple Example of LL(1) Grammar

- P1: $S \rightarrow$ *if* (*a*) *S else S* | *while* (*a*) *S* | *begin SL end*
  P2: $SL \rightarrow S\ S'$
  P3: $S' \rightarrow ; SL \mid \epsilon$
- {if, while, begin, end, a, (,), ;} are all terminal symbols
- Clearly, all alternatives of P1 start with distinct symbols and hence create no problem
- P2 has no choices
- Regarding P3, dirsymb(;SL) = {;}, and dirsymb(ϵ) = {end}, and the two have no common symbols
- Hence the grammar is LL(1)

LL(1) Table Construction Example 1

LL(1) Parsing Table for the original grammar

	if	id	else	a	$
S'	S' → S$			S' → S$	
S	S → if id S S → if id S else S			S → a	

Original Grammar

Grammar is not LL(1)

S' → S$
S → if id S |
if id S else S |
a

tokens: if, id, else, a

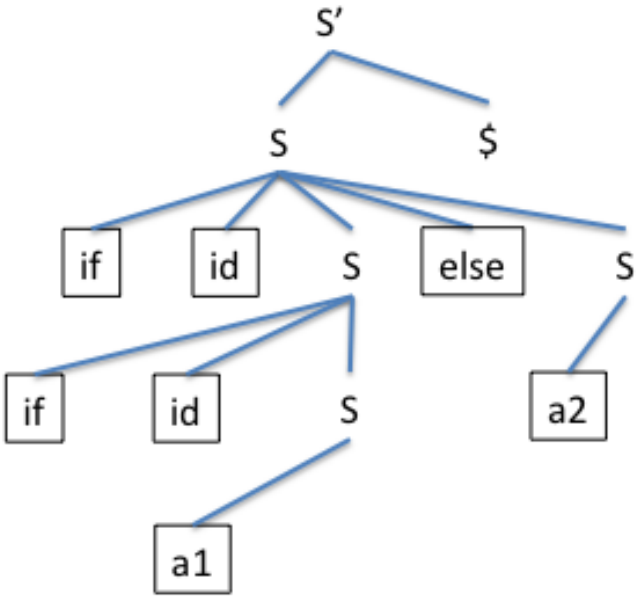
dirsymb(S$) = {if,a}; dirsymb (a) = {a}
dirsymb(if id S) = {if}
dirsymb(if id S else S) = {if}

$$dirsymb(if\ id\ S) \cap dirsymb(a) = \varnothing$$

$$dirsymb(if\ id\ S\ else\ S) \cap dirsymb(a) = \varnothing$$

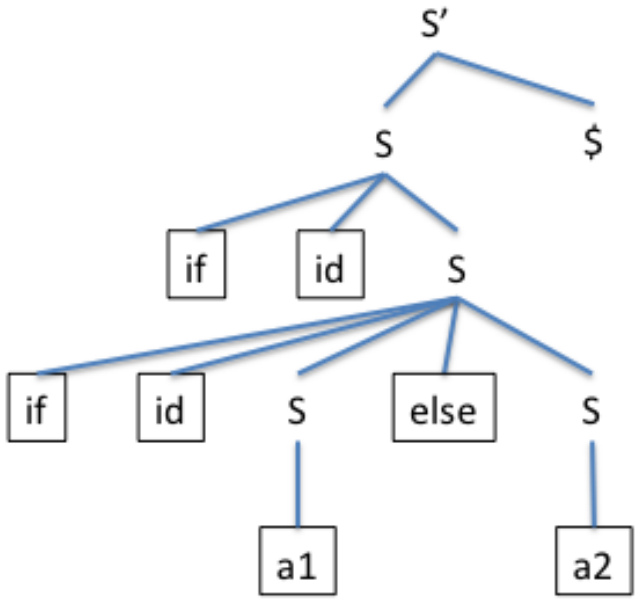
$$dirsymb(if\ id\ S) \cap dirsymb(if\ id\ S\ else\ S) \neq \varnothing$$

LL(1) Table Problem Example 1



string: if id (if id a1) else a2

parentheses are not part of the string



string: if id (if id a1 else a2)

parentheses are not part of the string

LL(1) Table Construction Example 2

Original Grammar

S' → S$
S → if id S |
if id S else S |
a

LL(1) Parsing Table for modified grammar

	if	else	a	$
S'	S' → S$		S' → S$	
S	S → if id S S1		S → a	
S1		S1 → ε S1 → else S		S1 → ε

dirsymb(S$) = {if, a}; dirsymb (a) = {a}
dirsymb(if id S S1) = {if}
dirsymb(else S) = {else}
dirsymb(ε) = {else, $}

Grammar is not LL(1)

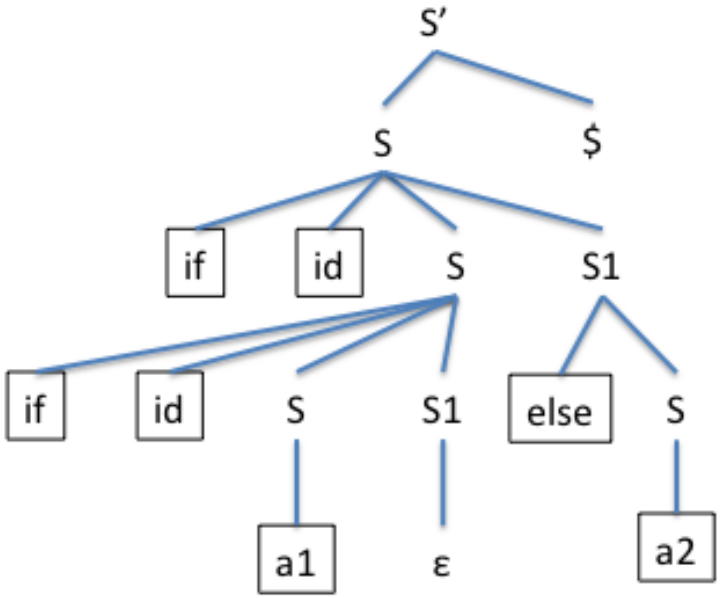
Left-Factored Grammar

S' → S$
S → if id S S1 | a
S1 → ε | else S

tokens: if, id, else, a

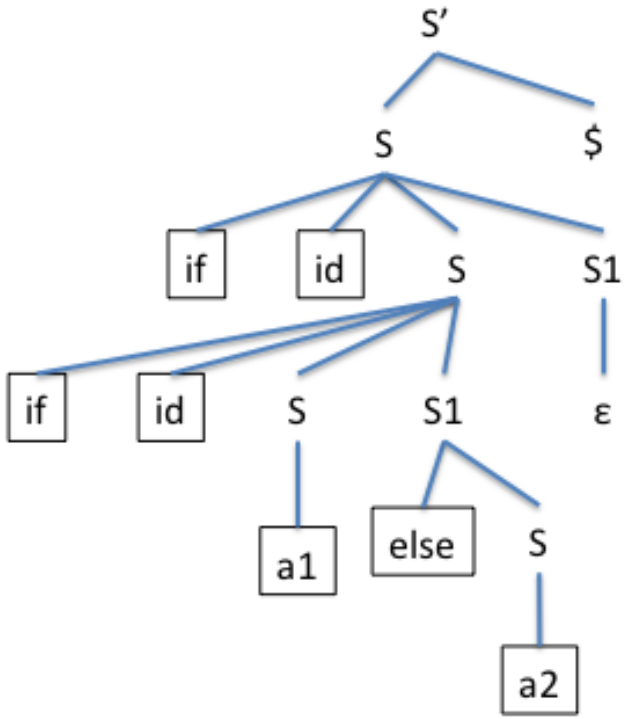
$$dirsymb(if\ id\ S\ S1) \cap dirsymb(a) = \varnothing$$
$$dirsymb(\varepsilon) \cap dirsymb(else\ S) \neq \varnothing$$

LL(1) Table Problem Example 2



string: if id (if id a1) else a2

parentheses are not part of the string



string: if id (if id a1 else a2)

parentheses are not part of the string

LL(1) Table Construction Example 3

S' → S$
S → aAS | c
A → ba | SB
B → bA | S
Grammar is LL(1)

LL(1) Parsing Table

	a	b	c	$
S'	S' → S$		S' → S$	
S	S → aAS		S → c	
A	A → SB	A → ba	A → SB	
B	B → S	B → bA	B → S	

first(S) = {a,c}
first(A) = {a,b,c}
first(B) = {a,b,c}

$$dirsymb(aAS) \cap dirsymb(c) = \varnothing$$
$$dirsymb(ba) \cap dirsymb(SB) = \varnothing$$
$$dirsymb(bA) \cap dirsymb(S) = \varnothing$$

follow(S) = {a,b,c,$}
follow(A) = {a,c}
follow(B) = {a,c}

dirsymb(S$) = {a,c}
dirsymb(aAS) = {a}
dirsymb(c) = {c}

dirsymb(ba) = {b}
dirsymb(SB) = {a,c}
dirsymb(bA) = {b}
dirsymb(S) = {a,c}

LL(1) Table Construction Example 4

Left-Recursive Grammar for Statement List

S' → SL $
SL → SL S | S
S → a

dirsymb(SL $) = {a}
dirsymb (a) = {a}
dirsymb(SL S) = {a}
dirsymb(S) = {a}

$$dirsymb(SL\ \ S) \bigcap dirsymb(S) \neq \varnothing$$

dirsymb(SL $) = {a}
dirsymb (a) = {a}
dirsymb(S A) = {a}
dirsymb(ε) = {$}

LL(1) Parsing Table for Left-Recursive Grammar

	a
S'	S' → SL $
SL	SL → SL S SL → S
S	S → a

Grammar is not LL(1)

Grammar is LL(1)

Right-Recursive Grammar for Statement List

S' → SL $
SL → S A
A → S A | ε
S → a

LL(1) Parsing Table for Right-Recursive Grammar

	a	$
S'	S' → SL $	
SL	SL → S A	
A	A → S A	A → ε
S	S → a	

$$dirsymb(S\ A) \bigcap dirsymb(\varepsilon) = \varnothing$$

Elimination of Useless Symbols

Now we study the *grammar transformations*, elimination of useless symbols, elimination of left recursion and left factoring

- Given a grammar $G = (N, T, P, S)$, a non-terminal X is *useful* if $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$, where, $w \in T^*$
  Otherwise, X is useless
- Two conditions have to be met to ensure that X is useful
  1. $X \Rightarrow^* w$, $w \in T^*$ (X derives some terminal string)
  2. $S \Rightarrow^* \alpha X \beta$ (X occurs in some string derivable from S)
- Example: $S \rightarrow AB \mid CA,\ B \rightarrow BC \mid AB,\ A \rightarrow a,$
  $C \rightarrow aB \mid b,\ D \rightarrow d$
  1. $A \rightarrow a,\ C \rightarrow b,\ D \rightarrow d,\ S \rightarrow CA$
  2. $S \rightarrow CA,\ A \rightarrow a,\ C \rightarrow b$

Testing for $X \Rightarrow^* w$

G' = (N',T',P',S') is the new grammar
N_OLD = ϕ;
N_NEW = $\{X \mid X \rightarrow w, w \in T^*\}$
while N_OLD $\neq$ N_NEW do {
  N_OLD = N_NEW;
  N_NEW = N_OLD $\cup \{X \mid X \rightarrow \alpha, \alpha \in (T \cup N\_OLD)^*\}$
}
N' = N_NEW; T' = T; S' = S;
P' = $\{p \mid$ all symbols of p are in $N' \cup T'\}$

Testing for $S \Rightarrow^* \alpha X \beta$

G' = (N',T',P',S') is the new grammar
N' = {S};
Repeat {
  for each production $A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$ with $A \in N'$ do
    add all nonterminals of $\alpha_1, \alpha_2, ..., \alpha_n$ to N' and
    all terminals of $\alpha_1, \alpha_2, ..., \alpha_n$ to T'
} until there is no change in N' and T'
P' = {p | all symbols of p are in $N' \cup T'$}; S' = S

Syntax Analysis:

Context-free Grammars, Pushdown Automata and Parsing
Part - 4

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is syntax analysis? (covered in lecture 1)
- Specification of programming languages: context-free grammars (covered in lecture 1)
- Parsing context-free languages: push-down automata (covered in lectures 1 and 2)
- Top-down parsing: LL(1) parsing (covered in lectures 2 and 3)
- Recursive-descent parsing
- Bottom-up parsing: LR-parsing

Elimination of Left Recursion

- A *left-recursive* grammar has a non-terminal A such that $A \Rightarrow^+ A\alpha$
- Top-down parsing methods (LL(1) and RD) cannot handle reft-recursive grammars
- Left-recursion in grammars can be eliminated by transformations
- A simpler case is that of grammars with *immediate left recursion*, where there is a production of the form $A \rightarrow A\alpha$
  - Two productions $A \rightarrow A\alpha \mid \beta$ can be transformed to $A \rightarrow \beta A'$, $A' \rightarrow \alpha A' \mid \epsilon$
  - In general, a group of productions:
    $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid ... \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid ... \mid \beta_n$
    can be transformed to
    $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid ... \mid \beta_n A'$, $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid ... \mid \alpha_m A' \mid \epsilon$

Left Recursion Elimination - An Example

$A \to A\alpha \mid \beta \Rightarrow A \to \beta A', A' \to \alpha A' \mid \epsilon$

- The following grammar for regular expressions is ambiguous:
  $E \to E + E \mid E\ E \mid E* \mid (E) \mid a \mid b$
- Equivalent left-recursive but unambiguous grammar is:
  $E \to E + T \mid T, T \to T\ F \mid F, F \to F* \mid P, P \to (E) \mid a \mid b$
- Equivalent non-left-recursive grammar is:
  $E \to T\ E', E' \to +T\ E' \mid \epsilon, T \to F\ T', T' \to F\ T' \mid \epsilon,$
  $F \to P\ F', F' \to *F' \mid \epsilon, P \to (E) \mid a \mid b$

Left Factoring

- If two alternatives of a production begin with the same string, then the grammar is not LL(1)
- Example: $S \rightarrow 0S1 \mid 01$ *is not LL(1)*
  - After left factoring: $S \rightarrow 0S'$, $S' \rightarrow S1 \mid 1$ *is LL(1)*
- General method: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \Rightarrow A \rightarrow \alpha A'$, $A' \rightarrow \beta_1 \mid \beta_2$
- Another example: a grammar for logical expressions is given below
  $E \rightarrow T$ *or* $E \mid T$, $T \rightarrow F$ *and* $T \mid F$,
  $F \rightarrow$ *not* $F \mid (E) \mid$ *true* $\mid$ *false*
  - This grammar is not LL(1) but becomes LL(1) after left factoring
  - $E \rightarrow TE'$, $E' \rightarrow$ *or* $E \mid \epsilon$, $T \rightarrow FT'$, $T' \rightarrow$ *and* $T \mid \epsilon$,
    $F \rightarrow$ *not* $F \mid (E) \mid$ *true* $\mid$ *false*

Grammar Transformations may not help!

Original Grammar

S' → S$
S → if id S |
if id S else S |
a

LL(1) Parsing Table for modified grammar

	if	else	a	$
S'	S' → S$		S' → S$	
S	S → if id S S1		S → a	
S1		S1 → ε S1 → else S		S1 → ε

dirsymb(S$) = {if, a}; dirsymb (a) = {a}
dirsymb(if id S S1) = {if}
dirsymb(else S) = {else}
dirsymb(ε) = {else, $}

Grammar is not LL(1)

Left-Factored Grammar

S' → S$
S → if id S S1 | a
S1 → ε | else S

tokens: if, id, else, a

$$dirsymb(if\ id\ S\ S1) \bigcap dirsymb(a) = \varnothing$$
$$dirsymb(\varepsilon) \bigcap dirsymb(else\ S) \neq \varnothing$$

Choose $S1 \rightarrow$ *else S* instead of $S1 \rightarrow \epsilon$ on lookahead *else*. This resolves the conflict. Associates *else* with the innermost *if*

Recursive-Descent Parsing

- Top-down parsing strategy
- One function/procedure for each nonterminal
- Functions call each other recursively, based on the grammar
- Recursion stack handles the tasks of LL(1) parser stack
- LL(1) conditions to be satisfied for the grammar
- Can be automatically generated from the grammar
- Hand-coding is also easy
- Error recovery is superior

An Example

Grammar: $S' \rightarrow S\$,\ S \rightarrow aAS \mid c,\ A \rightarrow ba \mid SB,\ B \rightarrow bA \mid S$

```
/* function for nonterminal S' */
void main(){/* S' --> S$ */
   fS(); if (token == eof) accept();
         else error();
}
/* function for nonterminal S */
void fS(){/* S --> aAS | c */
   switch token {
     case a : get_token(); fA(); fS();
              break;
     case c : get_token(); break;
     others : error();
   }
}
```

An Example (contd.)

```
void fA(){/* A --> ba | SB */
    switch token {
      case b : get_token();
               if (token == a) get_token();
               else error(); break;
      case a,c : fS(); fB(); break;
      others : error();
    }
}
void fB(){/* B --> bA | S */
    switch token {
      case b : get_token(); fA(); break;
      case a,c : fS(); break;
      others : error();
    }
}
```

Automatic Generation of RD Parsers

- Scheme is based on structure of productions
- Grammar must satisfy LL(1) conditions
- function *get_token*() obtains the next token from the lexical analyzer and places it in the global variable *token*
- function *error*() prints out a suitable error message
- In the next slide, for each grammar component, the code that must be generated is shown

Automatic Generation of RD Parsers (contd.)

1. ϵ : ;
2. $a \in T$: if ($token == a$) $get\_token()$; else $error()$;
3. $A \in N$: $fA()$; /* function call for nonterminal A */
4. $\alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$:

```
switch token {
  case dirsym(α1): program_segment(α1); break;
  case dirsym(α2): program_segment(α2); break;
  ...
  others: error();
}
```

5. $\alpha_1\alpha_2 \ ... \ \alpha_n$:
   program_segment(α_1); program_segment(α_2); ... ;
   program_segment(α_n);
6. $A \rightarrow \alpha$: void $fA()$ { program_segment(α); }

Bottom-Up Parsing

- Begin at the leaves, build the parse tree in small segments, combine the small trees to make bigger trees, until the root is reached
- This process is called *reduction* of the sentence to the start symbol of the grammar
- One of the ways of "reducing" a sentence is to follow the rightmost derivation of the sentence in *reverse*
  - *Shift-Reduce* parsing implements such a strategy
  - It uses the concept of a *handle* to detect when to perform reductions

Shift-Reduce Parsing

- **Handle**: A *handle* of a right sentential form γ, is a production $A \rightarrow \beta$ *and* a position in γ, where the string β may be found and replaced by A, to produce the previous right sentential form in a rightmost derivation of γ
  That is, if $S \Rightarrow^*_{rm} \alpha Aw \Rightarrow_{rm} \alpha\beta w$, then $A \rightarrow \beta$ in the position following α is a handle of $\alpha\beta w$
- A handle will always eventually appear on the top of the stack, never submerged inside the stack
- In S-R parsing, we locate the handle and reduce it by the LHS of the production repeatedly, to reach the start symbol
- These reductions, in fact, trace out a rightmost derivation of the sentence in reverse. This process is called handle pruning
- *LR-Parsing* is a method of shift-reduce parsing

Examples

1. $S \rightarrow aAcBe$, $A \rightarrow Ab \mid b$, $B \rightarrow d$
   For the string = *abbcde*, the rightmost derivation marked with handles is shown below

$$
\begin{aligned}
S &\Rightarrow \underline{aAcBe} \quad (aAcBe,\ S \rightarrow aAcBe) \\
&\Rightarrow aAc\underline{d}e \quad (d,\ B \rightarrow d) \\
&\Rightarrow a\underline{Ab}cde \quad (Ab,\ A \rightarrow Ab) \\
&\Rightarrow a\underline{b}bcde \quad (b,\ A \rightarrow b)
\end{aligned}
$$

The handle is unique if the grammar is unambiguous!

Examples (contd.)

2. $S \rightarrow aAS \mid c,\ A \rightarrow ba \mid SB,\ B \rightarrow bA \mid S$
   For the string = *acbbac*, the rightmost derivation marked with handles is shown below

$$
\begin{aligned}
S &\Rightarrow \underline{aAS} \ (aAS,\ S \rightarrow aAS) \\
&\Rightarrow aA\underline{c} \ (c,\ S \rightarrow c) \\
&\Rightarrow a\underline{SB}c \ (SB,\ A \rightarrow SB) \\
&\Rightarrow aS\underline{bA}c \ (bA,\ B \rightarrow bA) \\
&\Rightarrow aSb\underline{ba}c \ (ba,\ A \rightarrow ba) \\
&\Rightarrow a\underline{c}bbac \ (c,\ S \rightarrow c)
\end{aligned}
$$

Examples (contd.)

3. $E \rightarrow E + E$, $E \rightarrow E * E$, $E \rightarrow (E)$, $E \rightarrow id$
   For the string = $id + id * id$, two rightmost derivation marked with handles are shown below

$$
\begin{aligned}
E &\Rightarrow \underline{E + E} \quad (E + E,\ E \rightarrow E + E) \\
&\Rightarrow E + \underline{E * E} \quad (E * E,\ E \rightarrow E * E) \\
&\Rightarrow E + E * \underline{id} \quad (id,\ E \rightarrow id) \\
&\Rightarrow E + \underline{id} * id \quad (id,\ E \rightarrow id) \\
&\Rightarrow \underline{id} + id * id \quad (id,\ E \rightarrow id) \\
E &\Rightarrow \underline{E * E} \quad (E * E,\ E \rightarrow E * E) \\
&\Rightarrow E * \underline{id} \quad (id,\ E \rightarrow id) \\
&\Rightarrow \underline{E + E} * id \quad (E + E,\ E \rightarrow E + E) \\
&\Rightarrow E + \underline{id} * id \quad (id,\ E \rightarrow id) \\
&\Rightarrow \underline{id} + id * id \quad (id,\ E \rightarrow id)
\end{aligned}
$$

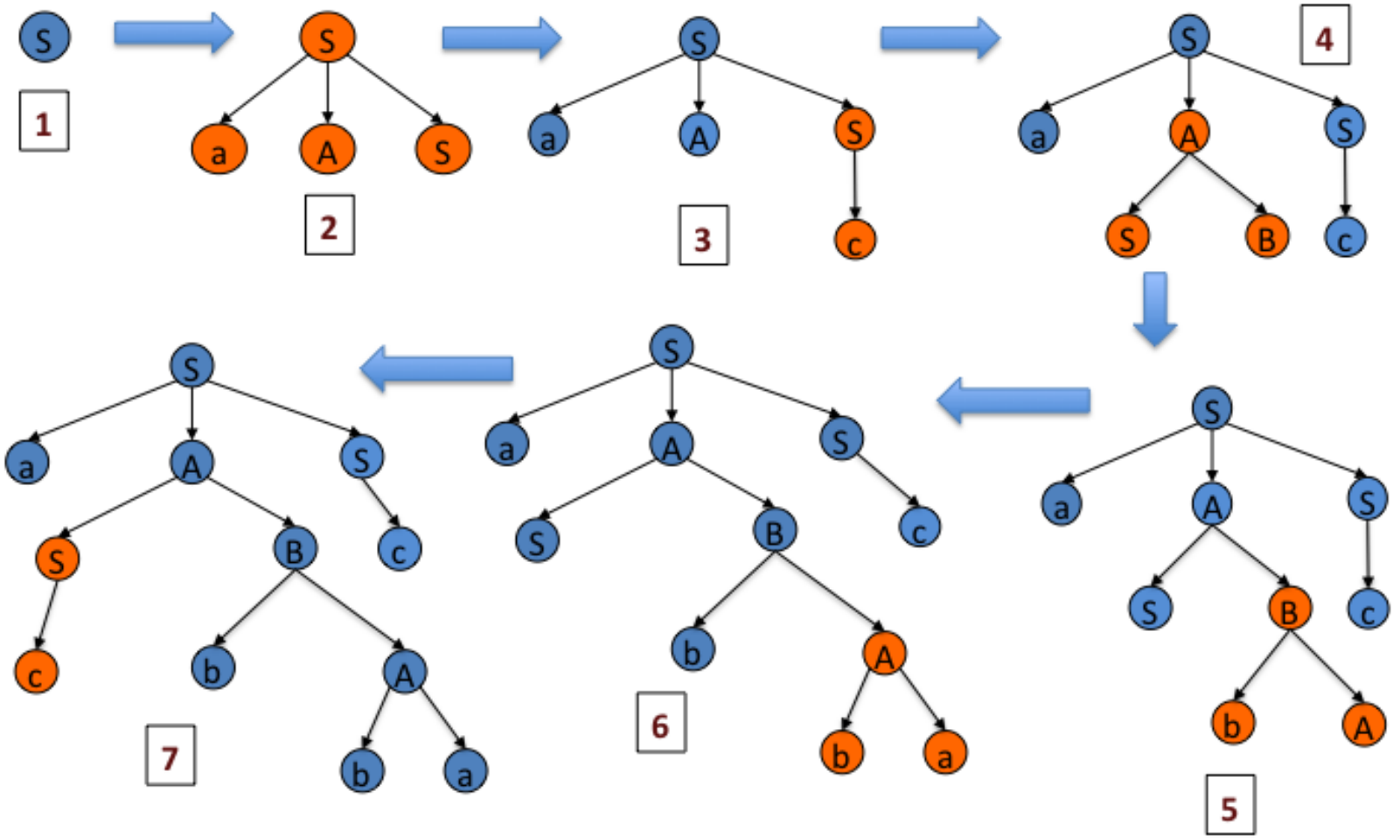
Rightmost Derivation and Bottom-UP Parsing

S → aAS | c
A → ba | SB
B → bA | S

Rightmost derivation of the string *acbbac*

S => aA**S** => a**A**c => aS**B**c => aS**b**Ac => a**S**bbac => acbbac

1 2 3 4 5 6 7



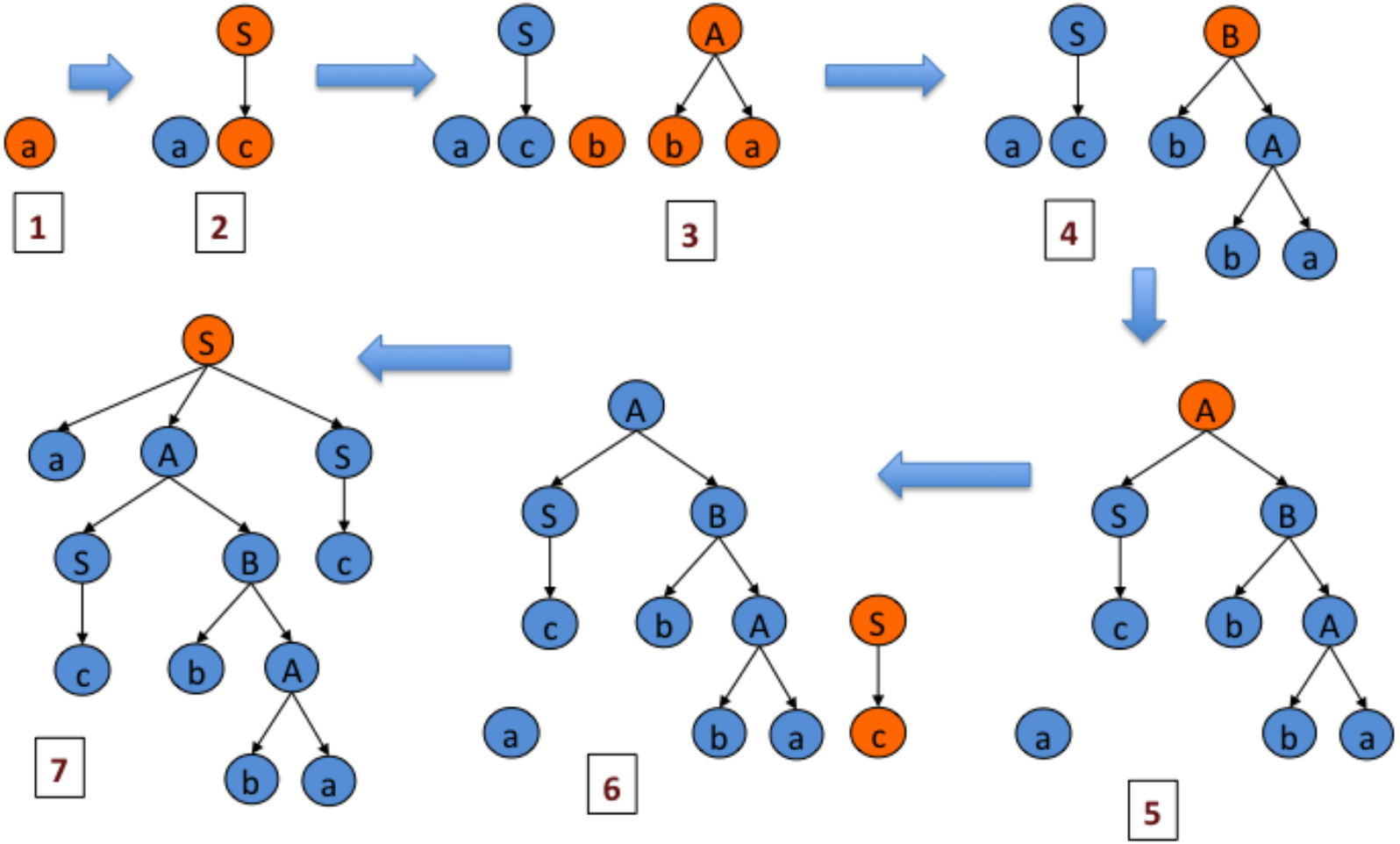
Rightmost Derivation and Bottom-UP Parsing (contd.)

S → aAS | c
A → ba | SB
B → bA | S

Rightmost derivation of the string *acbbac* in reverse

S <= a**A**S <= a**A**c <= a**S**B**B**c <= a**S**b**A**c <= a**S**bbac <= acbbac

7 6 5 4 3 2 1



Shift-Reduce Parsing Algorithm

- How do we locate a handle in a right sentential form?
  - An LR parser uses a DFA to detect the condition that a handle is now on the stack
- Which production to use, in case there is more than one with the same RHS?
  - An LR parser uses a parsing table similar to an LL parsing table, to choose the production
- A stack is used to implement an S-R parser, The parser has four actions
  1. **shift**: the next input symbol is shifted to the top of stack
  2. **reduce**: the right end of the handle is the top of stack; locates the left end of the handle inside the stack and replaces the handle by the LHS of an appropriate production
  3. **accept**: announces successful completion of parsing
  4. **error**: syntax error, error recovery routine is called

S-R Parsing Example 1

$ marks the bottom of stack and the right end of the input

Stack	**Input**	**Action**
$	*acbbac*$	shift
$ *a*	*cbbac*$	shift
$ *ac*	*bbac*$	reduce by $S \rightarrow c$
$ *aS*	*bbac*$	shift
$ *aSb*	*bac*$	shift
$ *aSbb*	*ac*$	shift
$ *aSbba*	*c*$	reduce by $A \rightarrow ba$
$ *aSbA*	*c*$	reduce by $B \rightarrow bA$
$ *aSB*	*c*$	reduce by $A \rightarrow SB$
$ *aA*	*c*$	shift
$ *aAc*	$	reduce by $S \rightarrow c$
$ *aAS*	$	reduce by $S \rightarrow aAS$
$ *S*	$	accept

S-R Parsing Example 2

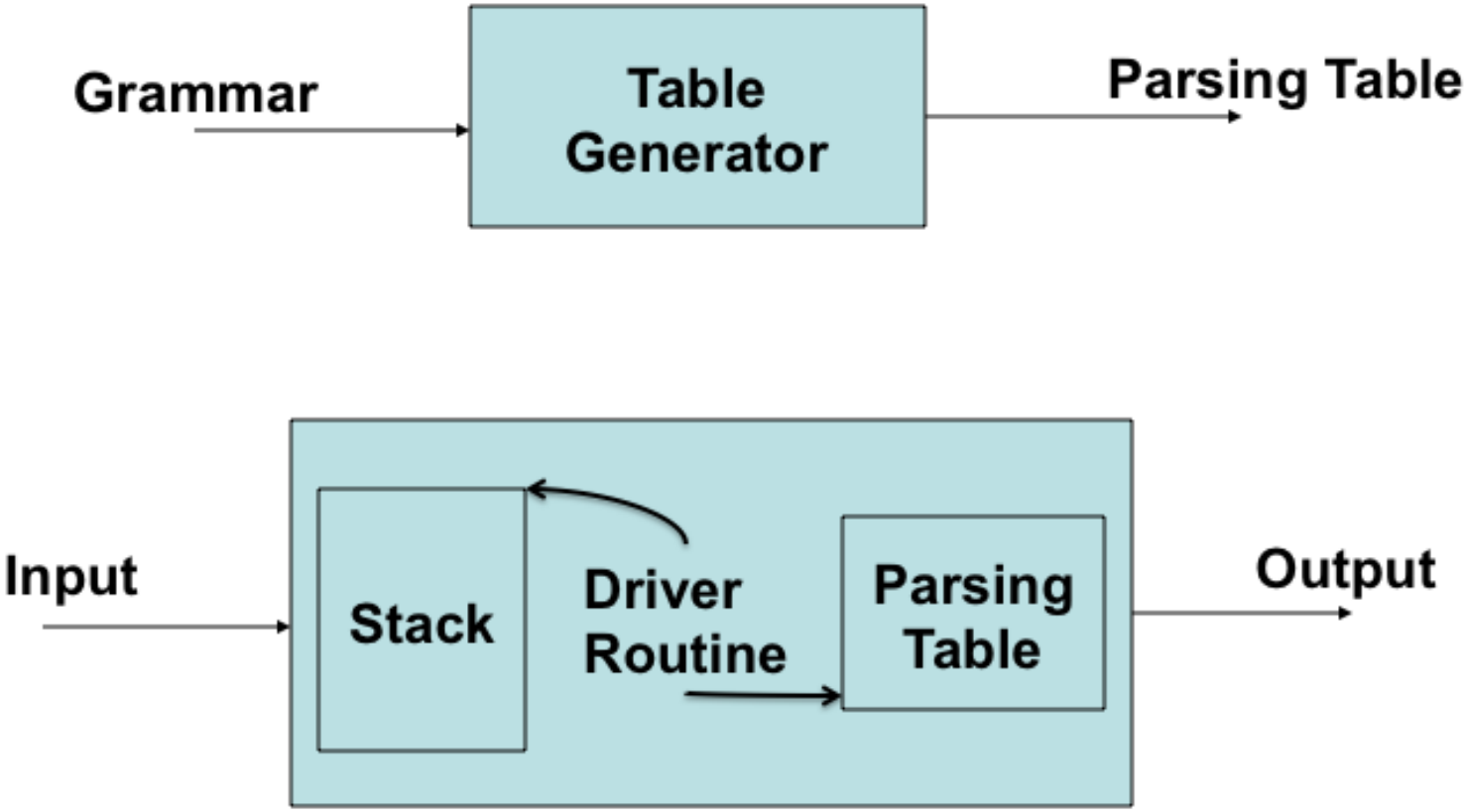
$ marks the bottom of stack and the right end of the input

Stack	Input	Action
$	$id_1 + id_2 * id_3$	shift
$ id_1	$+id_2 * id_3$	reduce by $E \rightarrow id$
$ E	$+id_2 * id_3$	shift
$ $E+$	$id_2 * id_3$	shift
$ $E + id_2$	$*id_3$	reduce by $E \rightarrow id$
$ $E + E$	$*id_3$	shift
$ $E + E*$	id_3	shift
$ $E + E * id_3$	$	reduce by $E \rightarrow id$
$ $E + E * E$	$	reduce by $E \rightarrow E * E$
$ $E + E$	$	reduce by $E \rightarrow E + E$
$ E	$	accept

LR Parsing

- LR(k) - *L*eft to right scanning with *R*ightmost derivation in reverse, k being the number of lookahead tokens
  - $k = 0, 1$ are of practical interest
- LR parsers are also automatically generated using parser generators
- LR grammars are a subset of CFGs for which LR parsers can be constructed
- LR(1) grammars can be written quite easily for practically all programming language constructs for which CFGs can be written
- LR parsing is the most general non-backtracking shift-reduce parsing method (known today)
- LL grammars are a strict subset of LR grammars - an LL(k) grammar is also LR(k), but not vice-versa

LR Parser Generation

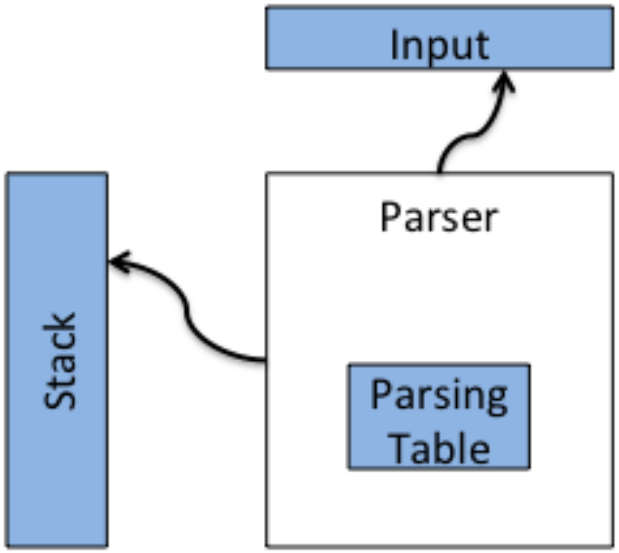


LR Parser Generator

LR Parser Configuration

- A configuration of an LR parser is:
  $(s_0 X_1 s_2 X_2 ... X_m s_m, \quad a_i a_{i+1} ... a_n \ \$)$, where,
  stack **unexpended input**
  $s_0, s_1, ..., s_m$, are the states of the parser, and $X_1, X_2, ..., X_m$, are grammar symbols (terminals or nonterminals)
- Starting configuration of the parser: $(s_0, a_1 a_2 ... a_n \$)$, where, s_0 is the initial state of the parser, and $a_1 a_2 ... a_n$ is the string to be parsed
- Two parts in the parsing table: *ACTION* and *GOTO*
  - The *ACTION* table can have four types of entries: **shift, reduce, accept**, or **error**
  - The *GOTO* table provides the next state information to be used after a *reduce* move

LR Parsing Algorithm



Initial configuration: Stack = *state 0*, Input = *w$*,
a = first input symbol;
repeat {
  let *s* be the top stack state;
  let *a* be the next input symbol;
  if (*ACTION[s, a] == shift p*) {
    push *a* and *p* onto the stack (in that order);
    advance input pointer;
  } else if (*ACTION[s,a] == reduce A* → *α*) then {
    pop *2*|α|* symbols off the stack;
    let *s'* be the top of stack state now;
    push *A* and *GOTO[s', A]* onto the stack
    (in that order);
  } else if (*ACTION[s, a] == accept*) break;
    /* parsng is over */
    else *error()*;
} until true; /* for ever */

LR Parsing Example 1 - Parsing Table

STATE	ACTION				GOTO		
	a	b	c	$	S	A	B
0	S2		S3		1		
1				R1 acc			
2	S2	S6	S3		8	4	
3	R3	R3	R3	R3			
4	S2		S3		5		
5	R2	R2	R2	R2			
6	S7						
7	R4	R4	R4	R4			
8	S2	S10	S3		12		9
9	R5	R5	R5	R5			
10	S2	S6	S3		8	11	
11	R6	R6	R6	R6			
12	R7	R7	R7	R7			

1. S' → S
2. S → aAS
3. S → c
4. A → ba
5. A → SB
6. B → bA
7. B → S

Syntax Analysis:

Context-free Grammars, Pushdown Automata and Parsing
Part - 5

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

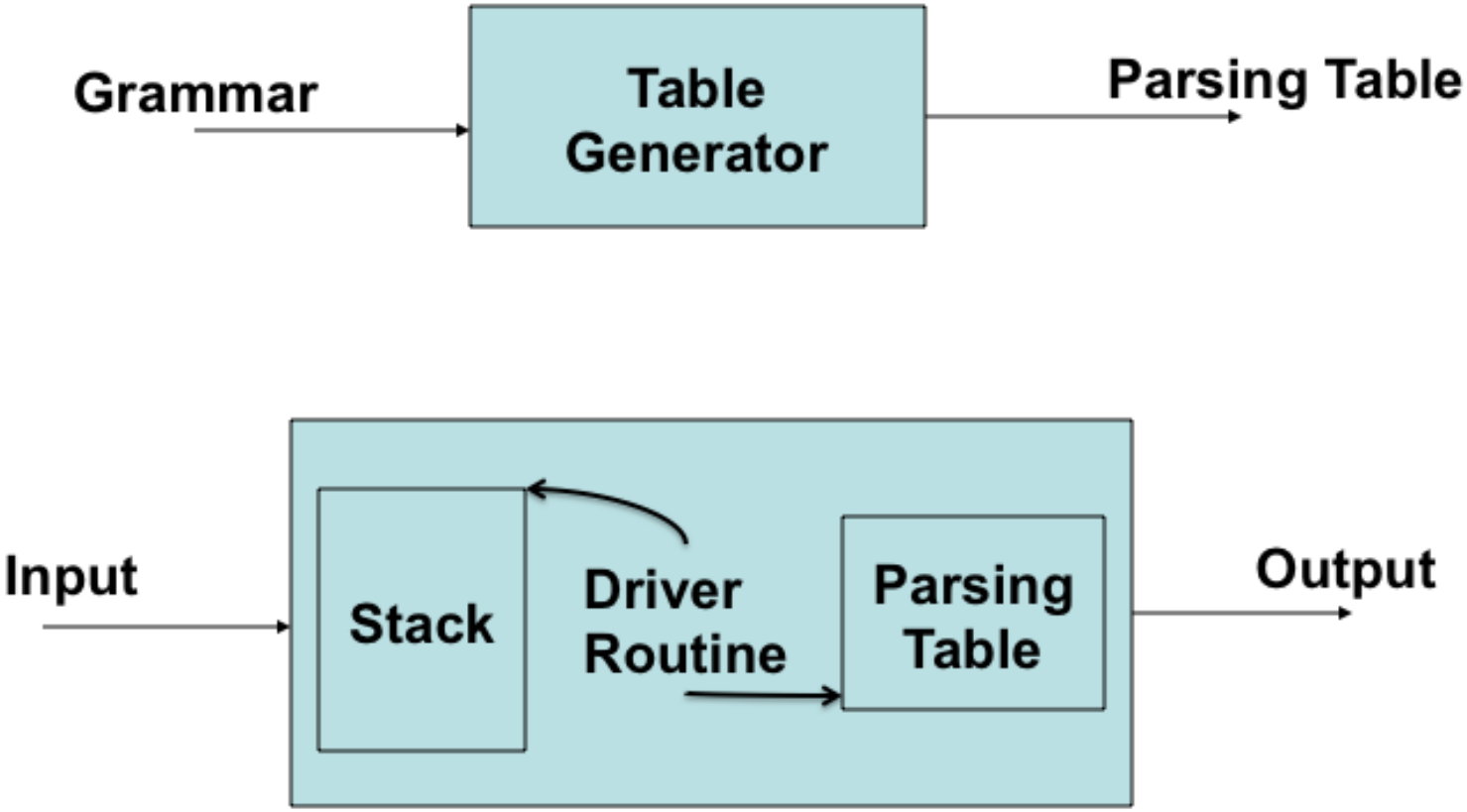
Outline of the Lecture

- What is syntax analysis? (covered in lecture 1)
- Specification of programming languages: context-free grammars (covered in lecture 1)
- Parsing context-free languages: push-down automata (covered in lectures 1 and 2)
- Top-down parsing: LL(1) parsing (covered in lectures 2 and 3)
- Recursive-descent parsing (covered in lecture 4)
- Bottom-up parsing: LR-parsing

LR Parsing

- LR(k) - *L*eft to right scanning with *R*ightmost derivation in reverse, k being the number of lookahead tokens
  - $k = 0, 1$ are of practical interest
- LR parsers are also automatically generated using parser generators
- LR grammars are a subset of CFGs for which LR parsers can be constructed
- LR(1) grammars can be written quite easily for practically all programming language constructs for which CFGs can be written
- LR parsing is the most general non-backtracking shift-reduce parsing method (known today)
- LL grammars are a strict subset of LR grammars - an LL(k) grammar is also LR(k), but not vice-versa

LR Parser Generation

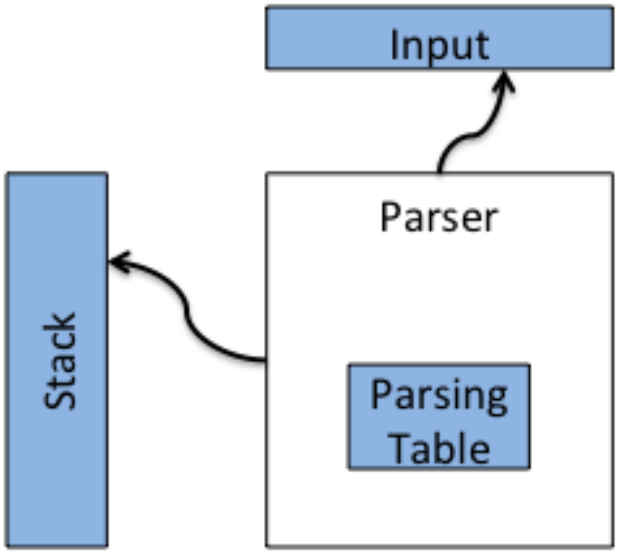


LR Parser Generator

LR Parser Configuration

- A configuration of an LR parser is:
  $(s_0 X_1 s_2 X_2 ... X_m s_m, \quad a_i a_{i+1} ... a_n \ \$)$, where,
  stack **unexpended input**
  $s_0, s_1, ..., s_m$, are the states of the parser, and $X_1, X_2, ..., X_m$, are grammar symbols (terminals or nonterminals)
- Starting configuration of the parser: $(s_0, a_1 a_2 ... a_n \$)$, where, s_0 is the initial state of the parser, and $a_1 a_2 ... a_n$ is the string to be parsed
- Two parts in the parsing table: *ACTION* and *GOTO*
  - The *ACTION* table can have four types of entries: **shift, reduce, accept**, or **error**
  - The *GOTO* table provides the next state information to be used after a *reduce* move

LR Parsing Algorithm



Initial configuration: Stack = *state 0*, Input = *w$*,
a = first input symbol;
repeat {
  let *s* be the top stack state;
  let *a* be the next input symbol;
  if (*ACTION[s, a] == shift p*) {
    push *a* and *p* onto the stack (in that order);
    advance input pointer;
  } else if (*ACTION[s,a] == reduce A* → *α*) then {
    pop *2*|α|* symbols off the stack;
    let *s'* be the top of stack state now;
    push *A* and *GOTO[s', A]* onto the stack
    (in that order);
  } else if (*ACTION[s, a] == accept*) break;
    /* parsng is over */
    else *error()*;
} until true; /* for ever */

LR Parsing Example 1 - Parsing Table

STATE	ACTION				GOTO		
	a	b	c	$	S	A	B
0	S2		S3		1		
1				R1 acc			
2	S2	S6	S3		8	4	
3	R3	R3	R3	R3			
4	S2		S3		5		
5	R2	R2	R2	R2			
6	S7						
7	R4	R4	R4	R4			
8	S2	S10	S3		12		9
9	R5	R5	R5	R5			
10	S2	S6	S3		8	11	
11	R6	R6	R6	R6			
12	R7	R7	R7	R7			

1. S' → S
2. S → aAS
3. S → c
4. A → ba
5. A → SB
6. B → bA
7. B → S

LR Parsing Example 1 (contd.)

Stack	Input	Action
0	*acbbac*\$	S2
0*a*2	*cbbac*\$	S3
0*a*2*c*3	*bbac*\$	R3 ($S \rightarrow c$, goto(2,S) = 8)
0*a*2*S*8	*bbac*\$	S10
0*a*2*S*8*b*10	*bac*\$	S6
0*a*2*S*8*b*10*b*6	*ac*\$	S7
0*a*2*S*8*b*10*b*6*a*7	*c*\$	R4 ($A \rightarrow ba$, goto(10,A) = 11)
0*a*2*S*8*b*10*A*11	*c*\$	R6 ($B \rightarrow bA$, goto(8,B) = 9)
0*a*2*S*8*B*9	*c*\$	R5 ($A \rightarrow SB$, goto(2,A) = 4)
0*a*2*A*4	*c*\$	S3
0*a*2*A*4*c*3	\$	R3 ($S \rightarrow c$, goto(4,S) = 5)
0*a*2*A*4*S*5	\$	R2 ($S \rightarrow aAS$, goto(0,S) = 1)
0*S*1	\$	R1 ($S' \rightarrow S$), and accept

LR Parsing Example 2 - Parsing Table

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				R7 acc			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

1. E → E+T
2. E → T
3. T → T*F
4. T → F
5. F → (E)
6. F → id
7. S → E

LR Parsing Example 2(contd.)

Stack	Input	Action
0	$id + id * id \$$	S5
0 id 5	$+id * id \$$	R6 ($F \rightarrow id$, G(0,F) = 3)
0 F 3	$+id * id \$$	R4 ($T \rightarrow F$, G(0,T) = 2)
0 T 2	$+id * id \$$	R2 ($E \rightarrow T$, G(0,E) = 1)
0 E 1	$+id * id \$$	S6
0 E 1 $+$ 6	$id * id \$$	S5
0 E 1 $+$ 6 id 5	$*id \$$	R6 ($F \rightarrow id$, G(6,F) = 3)
0 E 1 $+$ 6F3	$*id \$$	R4 ($T \rightarrow F$, G(6,T) = 9)
0 E 1 $+$ 6T9	$*id \$$	S7
0 E 1 $+$ 6T9 $*$ 7	$id \$$	S5
0 E 1 $+$ 6T9 $*$ 7 id 5	$\$$	R6 ($F \rightarrow id$, G(7,F) = 10)
0 E 1 $+$ 6T9 $*$ 7F10	$\$$	R3 ($T \rightarrow T * F$, G(6,T) = 9)
0 E 1 $+$ 6T9	$\$$	R1 ($E \rightarrow E + T$, G(0,E) = 1)
0 E 1	$\$$	R7 ($S \rightarrow E$) and accept

LR Grammars

- Consider a rightmost derivation:
  $S \Rightarrow^*_{rm} \phi Bt \Rightarrow_{rm} \phi\beta t$,
  where the production $B \rightarrow \beta$ has been applied
- A grammar is said to be **LR(k)**, if for any given input string, at each step of any rightmost derivation, the handle β can be detected by examining the string $\phi\beta$ and scanning *at most*, first *k* symbols of the unused input string *t*

LR Grammars (contd.)

- Example: The grammar, $\{S \rightarrow E,\ E \rightarrow E + E \mid E * E \mid id\}$, is not LR(2)
  - $S \Rightarrow^{1} \underline{E} \Rightarrow^{2} \underline{E + E} \Rightarrow^{3} E + \underline{E * E} \Rightarrow^{4} E + E * \underline{id} \Rightarrow^{5} E + \underline{id} * id \Rightarrow^{6} \underline{id} + id * id$
  - $S \Rightarrow^{1'} \underline{E} \Rightarrow^{2'} \underline{E * E} \Rightarrow^{3'} E * \underline{id} \Rightarrow^{4'} \underline{E + E} * id \Rightarrow^{5'} E + \underline{id} * id \Rightarrow^{6'} \underline{id} + id * id$
  - In the above two derivations, the handle at steps 6 & 6' and at steps 5 & 5', is $E \rightarrow id$, and the position is underlined (with the same lookahead of two symbols, $id+$ and $+id$)
  - However, the handles at step 4 and at step 4' are different ($E \rightarrow id$ and $E \rightarrow E + E$), even though the lookahead of 2 symbols is the same ($*id$), and the stack is also the same ($\phi = E + E$)
  - That means that the handle cannot be determined using the lookahead

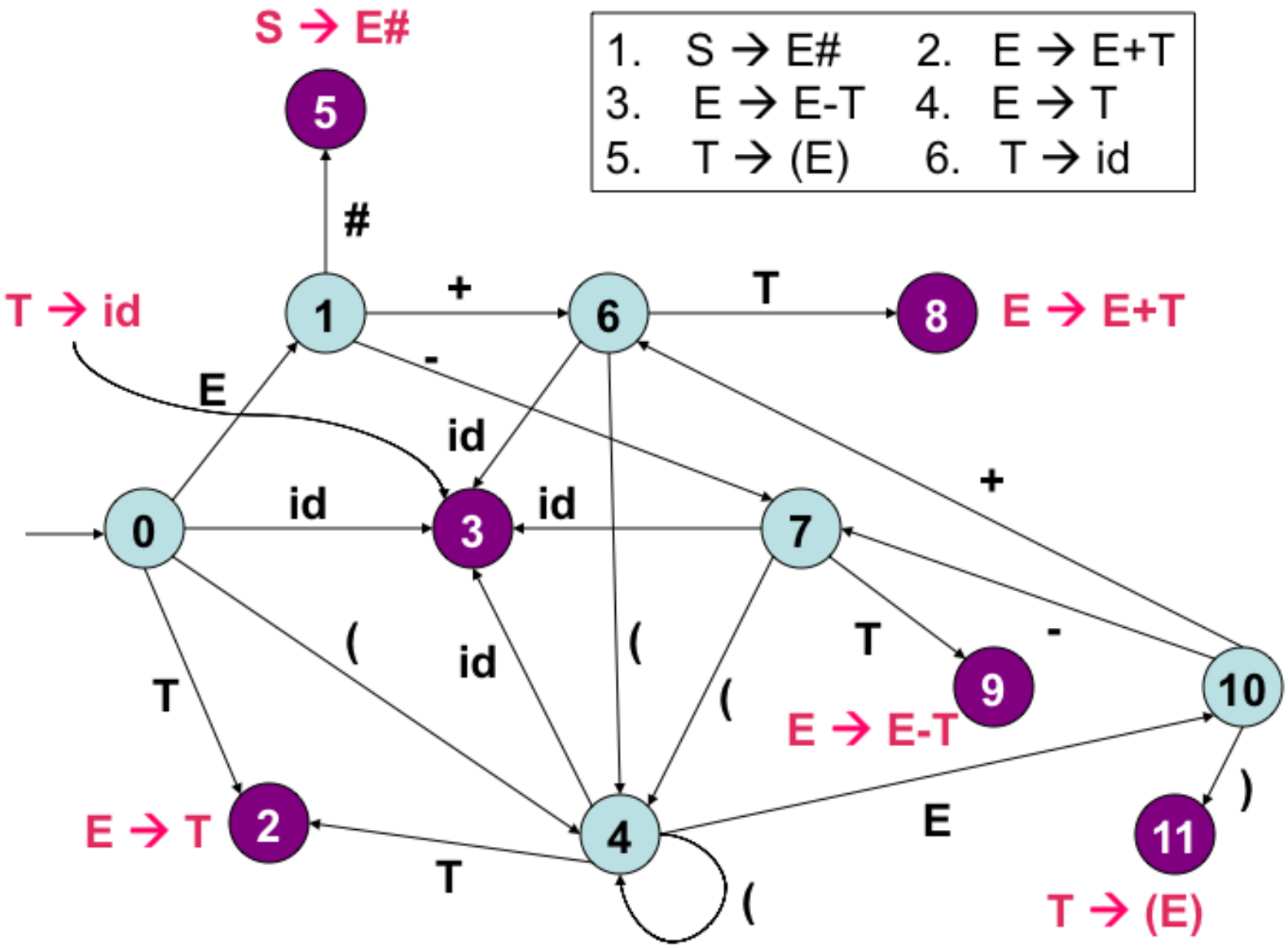
LR Grammars (contd.)

- A **viable prefix** of a sentential form $\phi\beta t$, where β denotes the handle, is any prefix of $\phi\beta$. A viable prefix cannot contain symbols to the right of the handle
- Example: $S \rightarrow E\#$, $E \rightarrow E + T \mid E - T \mid T$, $T \rightarrow id \mid (E)$
  $S \Rightarrow E\# \Rightarrow E + T\# \Rightarrow E + (E)\# \Rightarrow E + (T)\#$
  $\Rightarrow E + (id)\#$
  $E,\ E+,\ E + (,$ and $E + (id,$ are all viable prefixes of the right sentential form $E + (id)\#$
- It is always possible to add appropriate terminal symbols to the end of a viable prefix to get a right-sentential form
- Viable prefixes characterize the prefixes of sentential forms that can occur on the stack of an LR parser

LR Grammars (contd.)

- **Theorem**: The set of all viable prefixes of all the right sentential forms of a grammar is a regular language
- The DFA of this regular language can detect handles during LR parsing
- When this DFA reaches a “reduction state”, the corresponding viable prefix cannot grow further and thus signals a reduction
- This DFA can be constructed by the compiler using the grammar
- All LR parsers have such a DFA incorporated in them
- We construct an augmented grammar for which we construct the DFA
  - If S is the start symbol of G, then G' contains all productions of G and also a new production $S' \rightarrow S$
  - This enables the parser to halt as soon as S' appears on the stack

DFA for Viable Prefixes - LR(0) Automaton



Items and *Valid* Items

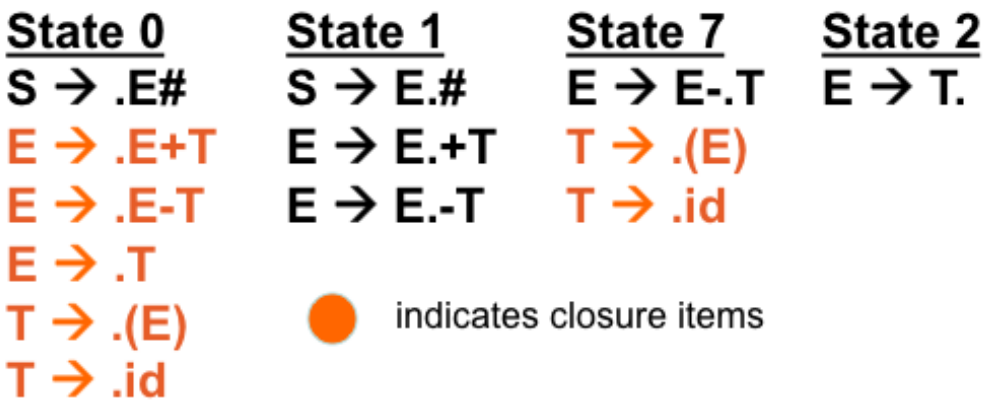
- A finite set of *items* is associated with each state of DFA
  - An *item* is a marked production of the form $[A \rightarrow \alpha_1.\alpha_2]$, where $A \rightarrow \alpha_1\alpha_2$ is a production and '.' denotes the mark
  - Many items may be associated with a production
    e.g., the items $[E \rightarrow .E + T]$, $[E \rightarrow E. + T]$, $[E \rightarrow E + .T]$, and $[E \rightarrow E + T.]$ are associated with the production $E \rightarrow E + T$
- An item $[A \rightarrow \alpha_1.\alpha_2]$ is *valid* for some viable prefix $\phi\alpha_1$, iff, there exists some rightmost derivation
  $S \Rightarrow^* \phi At \Rightarrow \phi\alpha_1\alpha_2 t$, where $t \in \Sigma^*$
- There may be several items valid for a viable prefix
  - The items $[E \rightarrow E - .T]$, $[T \rightarrow .id]$, and $[T \rightarrow .(E)]$ are all valid for the viable prefix "$E-$" as shown below
    $S \Rightarrow E\# \Rightarrow E - T\#$, $S \Rightarrow E\# \Rightarrow E - T\# \Rightarrow E - id\#$,
    $S \Rightarrow E\# \Rightarrow E - T\# \Rightarrow E - (E)\#$

Valid Items and States of LR(0) DFA

- An item indicates how much of a production has already been seen and how much remains to be seen
  - $[E \rightarrow E - .T]$ indicates that we have already seen a string derivable from "$E-$" and that we hope to see next, a string derivable from T
- Each state of an LR(0) DFA contains only those items that are valid for the *same set of viable prefixes*
  - All items in state 7 are valid for the viable prefixes "$E-$" and "$(E-$" (and many more)
  - All items in state 4 are valid for the viable prefix "(" (and many more)
  - In fact, the set of all viable prefixes for which the items in a state s are valid is the set of strings that can take us from state 0 (initial) to state s
- Constructing the LR(0) DFA using sets of items is very simple

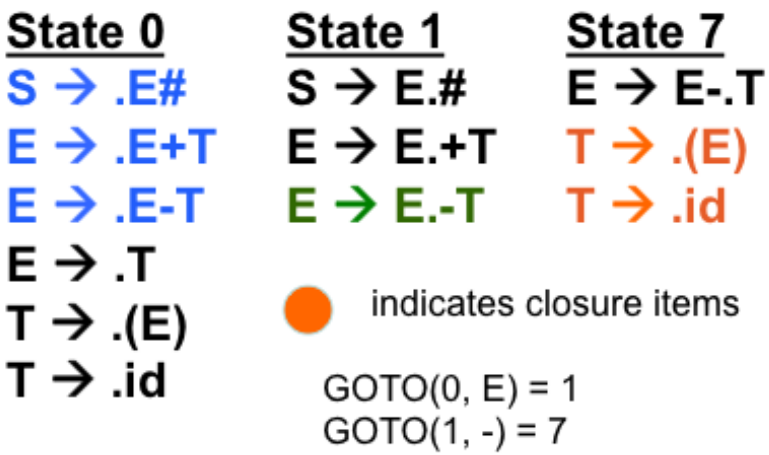
Closure of a Set of Items

Itemset closure(*I*){ /* I is a set of items */
  while (more items can be added to I) {
    for each item $[A \rightarrow \alpha.B\beta] \in I$ {
      /* note that *B* is a nonterminal and is right after the "." */
      for each production $B \rightarrow \gamma \in G$
        if (item $[B \rightarrow .\gamma] \notin I$) add item $[B \rightarrow .\gamma]$ to *I*
  }
  return *I*
}



GOTO set computation

Itemset GOTO(I, X){ /* I is a set of items
X is a grammar symbol, a terminal or a nonterminal */
Let $I' = \{[A \rightarrow \alpha X.\beta] \mid [A \rightarrow \alpha.X\beta] \in I\}$;
return (*closure*(I'))
}



Intuition behind *closure* and *GOTO*

- If an item $[A \rightarrow \alpha.B\delta]$ is in a state (i.e., item set I), then, some time in the future, we expect to see in the input, a string derivable from $B\delta$
  - This implies a string derivable from B as well
  - Therefore, we add an item $[B \rightarrow .\beta]$ corresponding to each production $B \rightarrow \beta$ of B, to the state (i.e., item set I)
- If I is the set of items valid for a viable prefix γ
  - All the items in *closure*(I) are also valid for γ
  - *GOTO*(I, X) is the set items valid for the viable prefix γX
    - If $[A \rightarrow \alpha.B\delta]$ (in item set I) is valid for the viable prefix $\phi\alpha$, and $B \rightarrow \beta$ is a production, we have $S \Rightarrow^* \phi At \Rightarrow \phi\alpha B\delta t \Rightarrow^* \phi\alpha Bxt \Rightarrow \phi\alpha\beta xt$ demonstrating that the item $[B \rightarrow .\beta]$ (in the closure of I) is valid for $\phi\alpha$
    - The above derivation also shows that the item $[A \rightarrow \alpha B.\delta]$ (in GOTO(I, B) is valid for the viable prefix $\phi\alpha B$

Construction of Sets of Canonical LR(0) Items

void Set_of_item_sets(G'){ /* G' is the augmented grammar */
  $C = \{closure(\{S' \rightarrow .S\})\}$;/* C is a set of item sets */
  while (more item sets can be added to C) {
    for each item set $I \in C$ and each grammar symbol X
    /* X is a grammar symbol, a terminal or a nonterminal */
      if (($GOTO(I, X) \neq \emptyset$) && ($GOTO(I, X) \notin C$))
        $C = C \cup GOTO(I, X)$
  }
}

- Each set in C (above) corresponds to a state of a DFA (LR(0) DFA)
- This is the DFA that recognizes viable prefixes

Construction of an LR(0) Automaton - Example 1

State 0
S → .E#
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 1
S → E.#
E → E.+T
E → E.-T

State 2
E → T.

State 3
T → id.

State 4
T → (.E)
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 5
S → E#.

State 6
E → E+.T
T → .(E)
T → .id

State 7
E → E-.T
T → .(E)
T → .id

State 8
E → E+T.

State 9
E → E-T.

State 10
T → (E.)
E → E.+T
E → E.-T

State 11
T → (E).

indicates closure items

indicates kernel items

Shift and Reduce Actions

- If a state contains an item of the form $[A \rightarrow \alpha.]$ ("reduce item"), then a reduction by the production $A \rightarrow \alpha$ is the action in that state
- If there are no "reduce items" in a state, then shift is the appropriate action
- There could be shift-reduce conflicts or reduce-reduce conflicts in a state
  - Both shift and reduce items are present in the same state (S-R conflict), or
  - More than one reduce item is present in a state (R-R conflict)
  - It is normal to have more than one shift item in a state (no shift-shift conflicts are possible)
- If there are no S-R or R-R conflicts in any state of an LR(0) DFA, then the grammar is LR(0), otherwise, it is not LR(0)

Syntax Analysis:

Context-free Grammars, Pushdown Automata and Parsing
Part - 6

Y.N. Srikant

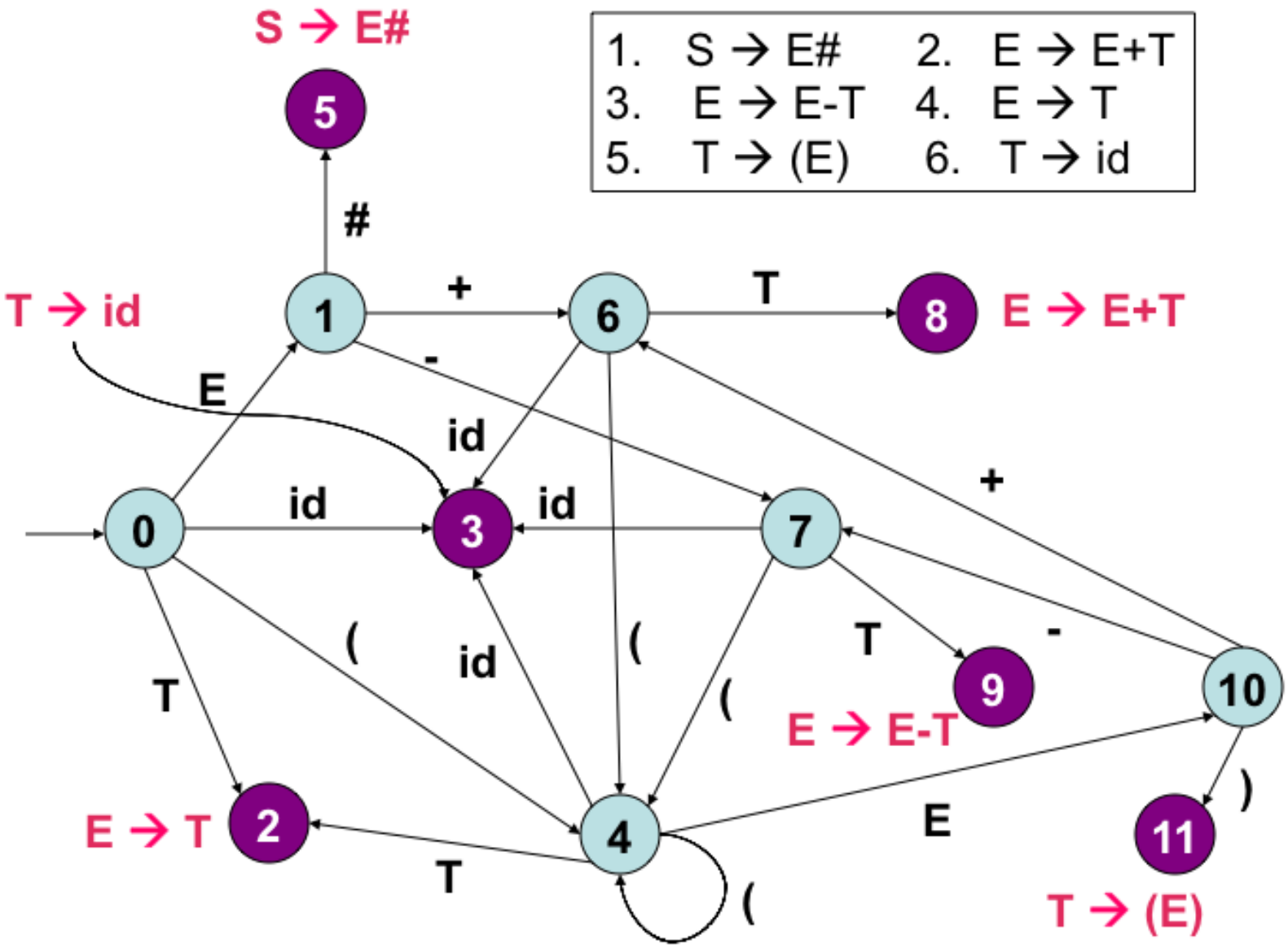
Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is syntax analysis? (covered in lecture 1)
- Specification of programming languages: context-free grammars (covered in lecture 1)
- Parsing context-free languages: push-down automata (covered in lectures 1 and 2)
- Top-down parsing: LL(1) parsing
  (covered in lectures 2 and 3)
- Recursive-descent parsing (covered in lecture 4)
- Bottom-up parsing: LR-parsing (continued)

DFA for Viable Prefixes - LR(0) Automaton



Construction of Sets of Canonical LR(0) Items

void Set_of_item_sets(G'){ /* G' is the augmented grammar */
  $C = \{closure(\{S' \rightarrow .S\})\}$;/* C is a set of item sets */
  while (more item sets can be added to C) {
    for each item set $I \in C$ and each grammar symbol X
    /* X is a grammar symbol, a terminal or a nonterminal */
      if (($GOTO(I, X) \neq \emptyset$) && ($GOTO(I, X) \notin C$))
        $C = C \cup GOTO(I, X)$
  }
}

- Each set in C (above) corresponds to a state of a DFA (LR(0) DFA)
- This is the DFA that recognizes viable prefixes

Construction of an LR(0) Automaton - Example 1

State 0
S → .E#
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 1
S → E.#
E → E.+T
E → E.-T

State 2
E → T.

State 3
T → id.

State 4
T → (.E)
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 5
S → E#.

State 6
E → E+.T
T → .(E)
T → .id

State 7
E → E-.T
T → .(E)
T → .id

State 8
E → E+T.

State 9
E → E-T.

State 10
T → (E.)
E → E.+T
E → E.-T

State 11
T → (E).

indicates closure items

indicates kernel items

Shift and Reduce Actions

- If a state contains an item of the form $[A \rightarrow \alpha.]$ ("reduce item"), then a reduction by the production $A \rightarrow \alpha$ is the action in that state
- If there are no "reduce items" in a state, then shift is the appropriate action
- There could be shift-reduce conflicts or reduce-reduce conflicts in a state
  - Both shift and reduce items are present in the same state (S-R conflict), or
  - More than one reduce item is present in a state (R-R conflict)
  - It is normal to have more than one shift item in a state (no shift-shift conflicts are possible)
- If there are no S-R or R-R conflicts in any state of an LR(0) DFA, then the grammar is LR(0), otherwise, it is not LR(0)

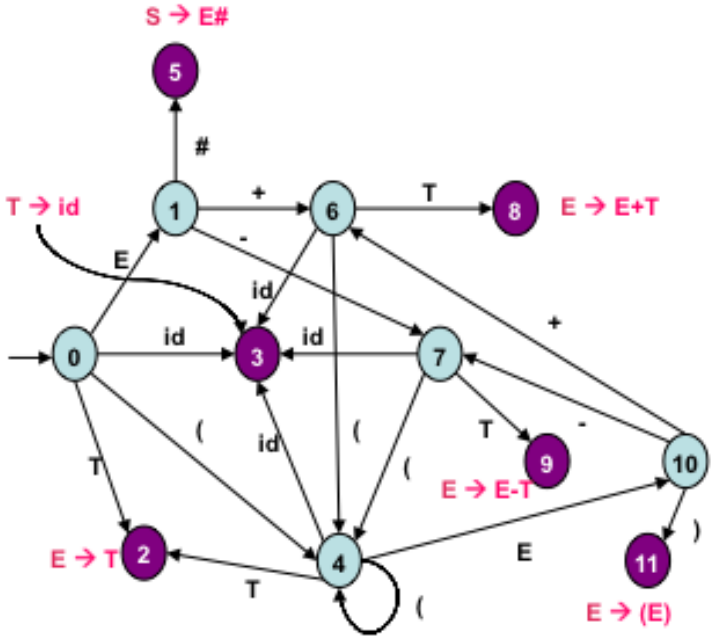
LR(0) Parser Table - Example 1

STATE	ACTION						GOTO		
	+	-	(	)	id	#	S	E	T
0			S4		S3			1	2
1	S6	S7				S5			
2	R4	R4	R4	R4	R4	R4			
3	R6	R6	R6	R6	R6	R6			
4			S4		S3			10	2
5	R1 acc	R1 acc	R1 acc	R1 acc	R1 acc	R1 acc			
6			S4		S3				8
7			S4		S3				9
8	R2	R2	R2	R2	R2	R2			
9	R3	R3	R3	R3	R3	R3			
10	S6	S7		S11					
11	R5	R5	R5	R5	R5	R5			

1. S → E#
2. E → E+T
3. E → E-T
4. E → T
5. T → (E)
6. T → id

Construction of an LR(0) Parser Table - Example 1

STATE	ACTION						GOTO		
	+	-	(	)	id	#	S	E	T
0			S4		S3			1	2
1	S6	S7				S5			
2	R4	R4	R4	R4	R4	R4			
3	R6	R6	R6	R6	R6	R6			
4			S4		S3			10	2
5	R1 acc	R1 acc	R1 acc	R1 acc	R1 acc	R1 acc			
6			S4		S3				8
7			S4		S3				9
8	R2	R2	R2	R2	R2	R2			
9	R3	R3	R3	R3	R3	R3			
10	S6	S7		S11					
11	R5	R5	R5	R5	R5	R5			



1. S → E#
2. E → E+T
3. E → E-T
4. E → T
5. T → (E)
6. T → id

State 0
S → .E#
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 1
S → E.#
E → E.+T
E → E.-T

State 2
E → T.

State 3
T → id.

State 4
T → (.E)
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 5
S → E#.

State 6
E → E+.T
T → .(E)
T → .id

State 7
E → E-.T
T → .(E)
T → .id

State 8
E → E+T.

State 9
E → E-T.

State 10
T → (E.)
E → E.+T
E → E.-T

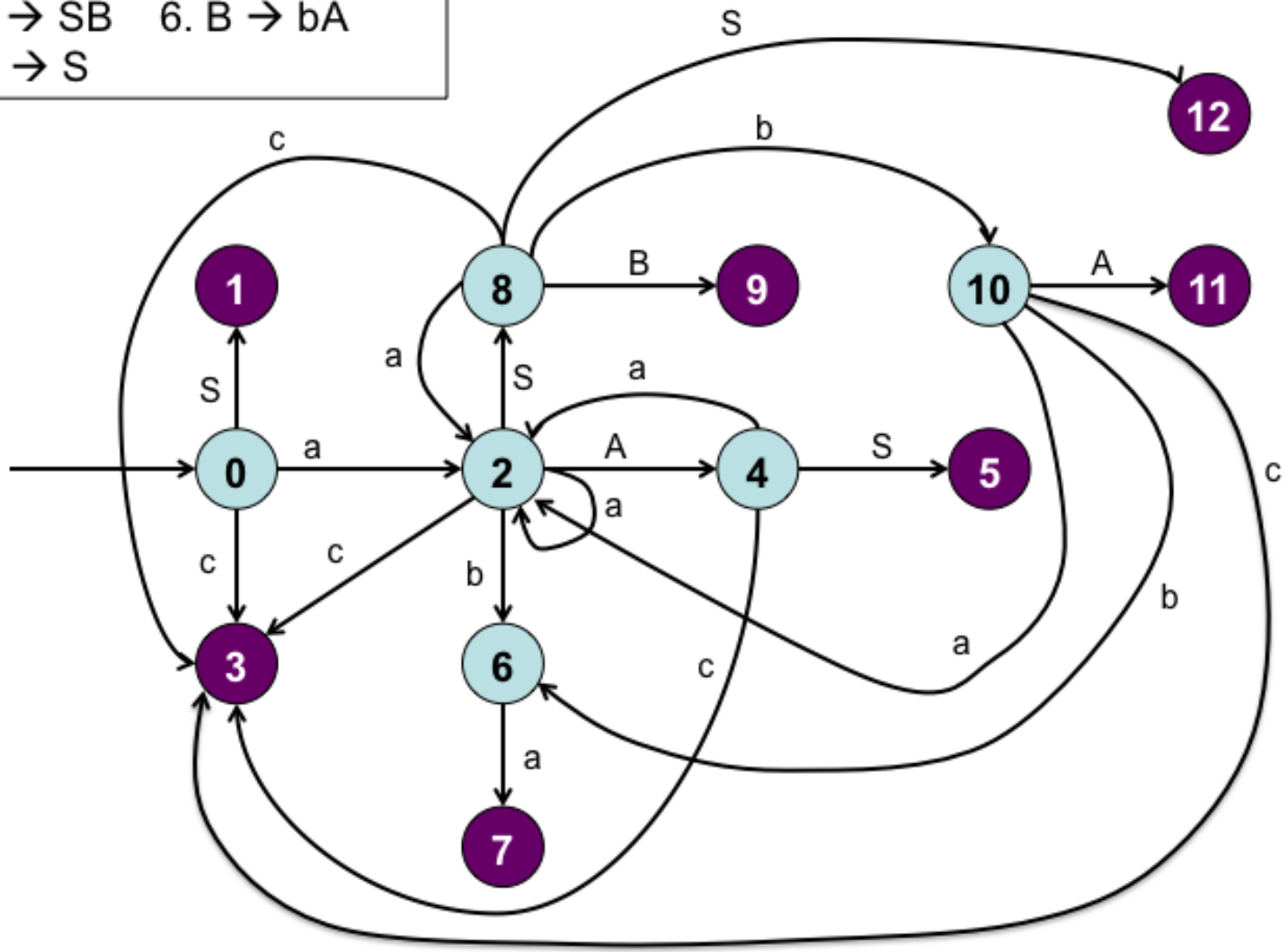
State 11
T → (E).

indicates closure items

indicates kernel items

LR(0) Automaton - Example 2

1. S' → S
2. S → aAS
3. S → c
4. A → ba
5. A → SB
6. B → bA
7. B → S



Construction of an LR(0) Automaton - Example 2

State 0
S' → .S
S → .aAS
S → .c

State 1
S' → S.

State 2
S → a.AS
A → .ba
A → .SB
S → .aAS
S → .c

State 3
S → c.

State 4
S → aA.S
S → .aAS
S → .c

State 5
S → aAS.

State 6
A → b.a

State 7
A → ba.

State 8
A → S.B
B → .bA
B → .S
S → .aAS
S → .c

State 9
A → SB.

State 10
B → b.A
A → .ba
A → .SB
S → .aAS
S → .c

State 11
B → bA.

State 12
B → S.


indicates closure items


indicates kernel items

1. S' → S	2. S → aAS	3. S → c
4. A → ba	5. A → SB	
6. B → bA	7. B → S	

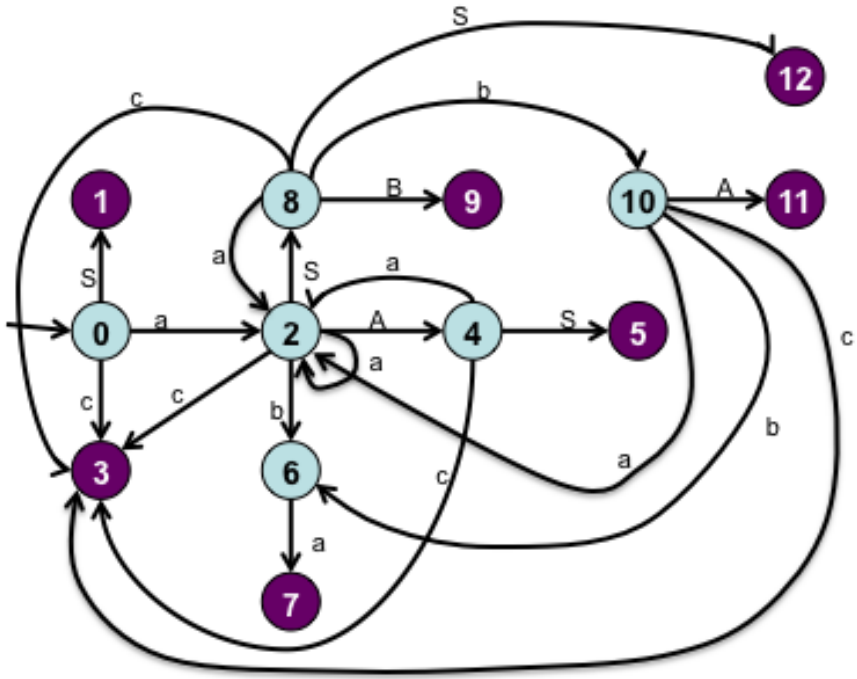
LR(0) Parser Table - Example 2

STATE	ACTION				GOTO		
	a	b	c	$	S	A	B
0	S2		S3		1		
1				R1 acc			
2	S2	S6	S3		8	4	
3	R3	R3	R3	R3			
4	S2		S3		5		
5	R2	R2	R2	R2			
6	S7						
7	R4	R4	R4	R4			
8	S2	S10	S3		12		9
9	R5	R5	R5	R5			
10	S2	S6	S3		8	11	
11	R6	R6	R6	R6			
12	R7	R7	R7	R7			

1. S' → S
2. S → aAS
3. S → c
4. A → ba
5. A → SB
6. B → bA
7. B → S

Construction of an LR(0) Parser Table - Example 2

STATE	ACTION				GOTO		
	a	b	c	$	S	A	B
0	S2		S3		1		
1				R1 acc			
2	S2	S6	S3		8	4	
3	R3	R3	R3	R3			
4	S2		S3		5		
5	R2	R2	R2	R2			
6	S7						
7	R4	R4	R4	R4			
8	S2	S10	S3		12		9
9	R5	R5	R5	R5			
10	S2	S6	S3		8	11	
11	R6	R6	R6	R6			
12	R7	R7	R7	R7			



1. S' → S
2. S → aAS
3. S → c
4. A → ba
5. A → SB
6. B → bA
7. B → S

State 0
S' → .S
S → .aAS
S → .c

State 1
S' → S.

State 2
S → a.AS
A → .ba
A → .SB
S → .aAS
S → .c

State 3
S → c.

State 4
S → aA.S
S → .aAS
S → .c

State 5
S → aAS.

State 6
A → b.a

State 7
A → ba.

State 8
A → S.B
B → .bA
B → .S
S → .aAS
S → .c

State 9
A → SB.

State 10
B → b.A
A → .ba
A → .SB
S → .aAS
S → .c

State 11
B → bA.

State 12
B → S.

indicates closure items indicates kernel items

A Grammar that is not LR(0) - Example 1

State 0
S → .E
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 1
S → E.
E → E.+T
E → E.-T

shift-reduce conflicts in state 1

State 2
E → T.

State 3
T → id.

State 4
T → (.E)
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 5
E → E+.T
T → .(E)
T → .id

State 6
E → E-.T
T → .(E)
T → .id

State 7
E → E+T.

State 8
E → E-T.

State 9
T → (E.)
E → E.+T
E → E.-T

State 10
T → (E).

● indicates closure items

● indicates kernel items

follow(S) = {$}, where $ is EOF
Reduction on $, and shifts on + and - , will resolve the conflicts
This is similar to having an end marker such as #

Grammar is not LR(0), but is SLR(1)

SLR(1) Parsers

- If the grammar is not LR(0), we try to resolve conflicts in the states using one look-ahead symbol
- Example: The expression grammar that is not LR(0)
  The state containing the items $[T \rightarrow F.]$ and $[T \rightarrow F. * T]$ has S-R conflicts
  - Consider the reduce item $[T \rightarrow F.]$ and the symbols in $FOLLOW(T)$
  - $FOLLOW(T) = \{+,), \$\}$, and reduction by $T \rightarrow F$ can be performed on seeing one of these symbols in the input (look-ahead), since shift requires seeing $*$ in the input
  - Recall from the definition of $FOLLOW(T)$ that symbols in $FOLLOW(T)$ are the only symbols that can legally follow T in any sentential form, and hence reduction by $T \rightarrow F$ when one of these symbols is seen, is correct
  - If the S-R conflicts can be resolved using the FOLLOW set, the grammar is said to be SLR(1)

A Grammar that is not LR(0) - Example 2

State 0
S → .E
E → .E+T
E → .T
T → .F*T
T → .F
F → .(E)
F → .id

State 1
S → E.
E → E.+T
Shift-reduce conflict

State 2
E → T.

State 3
T → F.*T
T → F.
Shift-reduce conflict

State 4
F → (.E)
E → .E+T
E → .T
T → .F*T
T → .F
F → .(E)
F → .id

State 5
F → id.

State 6
E → E+.T
T → .F*T
T → .F
F → .(E)
F → .id

State 7
T → F*.T
T → .F*T
T → .F
F → .(E)
F → .id

State 8
F → (E.)
E → E.+T

State 9
E → E+T.

State 10
E → F*T.

State 11
F → (E).

follow(S) = {$}, Reduction on $ and shift on +, eliminates conflicts
follow(T) = {$,), +}, where $ is EOF
Reduction on $,), and +, and shift on *, eliminates conflicts

Grammar is not LR(0), but is SLR(1)

Construction of an SLR(1) Parsing Table

Let $C = \{I_0, I_1, ..., I_i, ..., I_n\}$ be the canonical LR(0) collection of items, with the corresponding states of the parser being 0, 1, ... , i, ... , n
Without loss of generality, let 0 be the initial state of the parser (containing the item $[S' \rightarrow .S]$)
Parsing actions for state *i* are determined as follows

1. If $([A \rightarrow \alpha.a\beta] \in I_i)$ && $([A \rightarrow \alpha a.\beta] \in I_j)$
   set ACTION[i, a] = *shift j* /* *a* is a terminal symbol */
2. If $([A \rightarrow \alpha.] \in I_i)$
   set ACTION[i, a] = *reduce* $A \rightarrow \alpha$, for all $a \in$ *follow*(A)
3. If $([S' \rightarrow S.] \in I_i)$ set ACTION[i, $] = *accept*

S-R or R-R conflicts in the table imply grammar is not SLR(1)

4. If $([A \rightarrow \alpha.A\beta] \in I_i)$ && $([A \rightarrow \alpha A.\beta] \in I_j)$
   set GOTO[i, A] = *j* /* *A* is a nonterminal symbol */

All other entries not defined by the rules above are made *error*

A Grammar that is not LR(0) - Example 3

Grammar
S' → S, S → aSb, S → ε

follow(S) = {$, b}

State 0
S' → .S
S → .aSb
S → .

State 1
S' → S.

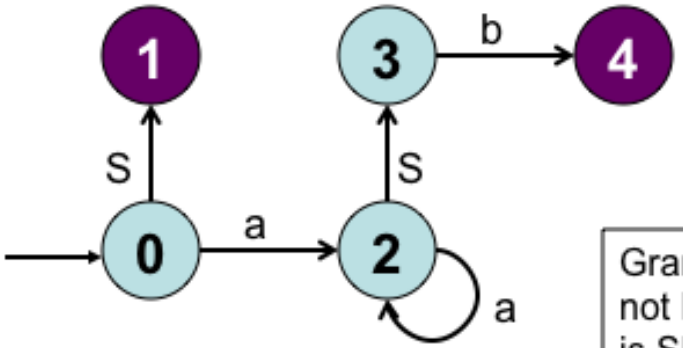
State 2
S → a.Sb
S → .aSb
S → .

shift-reduce conflict in states 0, 2

State 3
S → aS.b

State 4
S → aSb.

	a	b	$	S
0	S2	reduce S → ε	reduce S → ε	1
1			accept	
2	S2	reduce S → ε	reduce S → ε	3
3		S4		
4		reduce S → aSb	reduce S → aSb	



Grammar is not LR(0), but is SLR(1)

● indicates closure items

● indicates kernel items

A Grammar that is not SLR(1) - Example 1

Grammar: S' → S,
S → aSb, S → ab, S → ε

follow(S) = {$, b}
State 0: Reduction on *$* and *b*, by S → ε, and shift on *a* resolves conflicts
State 2: S-R conflict on *b* still remains

State 0
S' → .S
S → .aSb
S → .ab
S → .

State 1
S' → S.

State 2
S → a.Sb
S → a.b
S → .aSb
S → .ab
S → .

State 3
S → aS.b

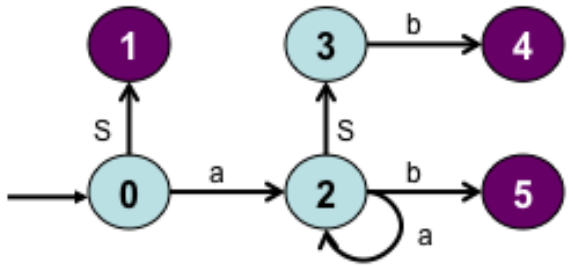
State 4
S → aSb.

State 5
S → ab.

shift-reduce conflict in states 0, 2

Grammar is neither LR(0) nor SLR(1)

	a	b	$	S
0	S2	R: S → ε	R: S → ε	1
1			accept	
2	S2	S5, R: S → ε	R: S → ε	3
3		S4		
4		R: S → aSb	R: S → aSb	
5		R: S → ab	R: S → ab	



A Grammar that is not SLR(1) - Example 2

Grammar

S' → S
S → L=R
S → R
L → *R
L → id
R → L

Grammar is neither LR(0) nor SLR(1)

State 0

S' → .S
S → .L=R
S → .R
L → .*R
L → .id
R → .L

State 1

S' → S.

State 3

S → R.

State 2

S → L .=R
R → L .
shift-reduce conflict

State 4

L → *.R
R → .L
L → .*R
L → .id

State 5

L → id.

State 6

S → L=.R
R → .L
L → .*R
L → .id

State 7

L → *R.

State 8

R → L .

State 9

S → L=R.

Follow(R) = {$,=} does not resolve S-R conflict

The Problem with SLR(1) Parsers

- SLR(1) parser construction process does not remember enough left context to resolve conflicts
  - In the "$L = R$" grammar (previous slide), the symbol '=' got into follow(R) because of the following derivation:
    $S' \Rightarrow S \Rightarrow L = R \Rightarrow L = L \Rightarrow L = id \Rightarrow *\underline{R =} id \Rightarrow ...$
  - The production used is $L \rightarrow *R$
  - The following rightmost derivation in *reverse* does not exist (and hence reduction by $R \rightarrow L$ on '=' in state 2 is illegal)
    $id = id \Leftarrow L = id \Leftarrow R = id ...$
- Generalization of the above example
  - In some situations, when a state *i* appears on top of the stack, a viable prefix $\beta\alpha$ may be on the stack such that βA cannot be followed by '*a*' in any right sentential form
  - Thus, the reduction by $A \rightarrow \alpha$ would be invalid on '*a*'
  - In the above example, $\beta = \epsilon$, $\alpha = L$, and $A = R$; L cannot be reduced to R on '=', since it would lead to the above illegal derivation sequence

LR(1) Parsers

- LR(1) items are of the form $[A \rightarrow \alpha.\beta,\ a]$, a being the "lookahead" symbol
- Lookahead symbols have no part to play in shift items, but in reduce items of the form $[A \rightarrow \alpha.,\ a]$, reduction by $A \rightarrow \alpha$ is valid only if the next input symbol is 'a'
- An LR(1) item $[A \rightarrow \alpha.\beta,\ a]$ is *valid* for a viable prefix γ, if there is a derivation $S \Rightarrow^*_{rm} \delta Aw \Rightarrow_{rm} \delta\alpha\beta w$, where, $\gamma = \delta\alpha$, $a = first(w)$ or $w = \epsilon$ and $a = \$$
- Consider the grammar: $S' \rightarrow S,\ S \rightarrow aSb \mid \epsilon$
  - $[S \rightarrow a.Sb,\ \$]$ is valid for the VP a, $S' \Rightarrow S \Rightarrow aSb$
  - $[S \rightarrow a.Sb,\ b]$ is valid for the VP aa, $S' \Rightarrow S \Rightarrow aSb \Rightarrow aaSbb$
  - $[S \rightarrow .,\ \$]$ is valid for the VP ϵ, $S' \Rightarrow S \Rightarrow \epsilon$
  - $[S \rightarrow aSb.,\ b]$ is valid for the VP $aaSb$, $S' \Rightarrow S \Rightarrow aSb \Rightarrow aaSbb$

LR(1) Grammar - Example 1

Grammar
S' → S, S → aSb, S → ε

State 0
S' → .S , $
S → .aSb , $
S → . , $

State 1
S' → S. , $

State 2
S → a.Sb , $
S → .aSb , b
S → . , b

State 3
S → aS.b , $

State 4
S → a.Sb , b
S → .aSb , b
S → . , b

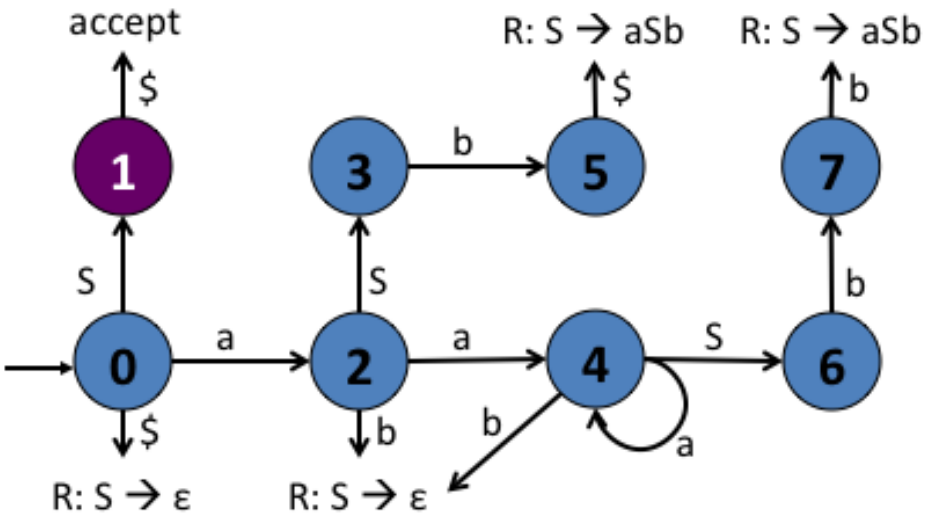
State 5
S → aSb. , $

State 6
S → aS.b , b

State 7
S → aSb. , b

Grammar is LR(1)

	a	b	$	S
0	S2		R: S → ε	1
1			accept	
2	S4	R: S → ε		3
3		S5		
4	S4	R: S → ε		6
5			R: S → aSb	
6		S7		
7		R: S → aSb		



Closure of a Set of LR(1) Items

Itemset closure(*I*){ /* I is a set of LR(1) items */
  while (more items can be added to I) {
    for each item $[A \rightarrow \alpha.B\beta,\ a] \in I$ {
      for each production $B \rightarrow \gamma \in G$
        for each symbol $b \in \mathit{first}(\beta a)$
          if (item $[B \rightarrow .\gamma,\ b] \notin I$) add item $[B \rightarrow .\gamma,\ b]$ to *I*
  }
  return *I*
}

Grammar
S' → S
S → aSb | ε

State 0
S' → .S , $
S → .aSb , $
S → . , $

State 3
S → aS.b , $

State 4
S → a.Sb , b
S → .aSb , b
S → . , b

State 7
S → aSb. , b

GOTO set computation

Itemset GOTO(I, X){ /* I is a set of LR(1) items
X is a grammar symbol, a terminal or a nonterminal */
Let $I' = \{[A \rightarrow \alpha X.\beta,\ a] \mid [A \rightarrow \alpha.X\beta,\ a] \in I\}$;
return (*closure*(I'))
}

Grammar
S' → S
S → aSb | ε

State 0
S' → .S , $
S → .aSb , $
S → . , $

State 1
S' → S. , $

State 2
S → a.Sb , $
S → .aSb , b
S → . , b

State 4
S → a.Sb , b
S → .aSb , b
S → . , b

GOTO(0, S) = 1, GOTO(0,a) = 2, GOTO(2,a) = 4

Construction of Sets of Canonical of LR(1) Items

void Set_of_item_sets(G'){ /* G' is the augmented grammar */
  $C = \{closure(\{S' \rightarrow .S,\ \$\})\}$;/* C is a set of LR(1) item sets */
  while (more item sets can be added to C) {
    for each item set $I \in C$ and each grammar symbol X
    /* X is a grammar symbol, a terminal or a nonterminal */
      if (($GOTO(I, X) \neq \emptyset$) && ($GOTO(I, X) \notin C$))
        $C = C \cup GOTO(I, X)$
  }
}

- Each set in C (above) corresponds to a state of a DFA (LR(1) DFA)
- This is the DFA that recognizes viable prefixes

LR(1) DFA Construction - Example 1

Grammar
S' → S, S → aSb, S → ε

State 0
S' → .S , $
S → .aSb , $
S → . , $

State 1
S' → S. , $

State 2
S → a.Sb , $
S → .aSb , b
S → . , b

State 3
S → aS.b , $

State 4
S → a.Sb , b
S → .aSb , b
S → . , b

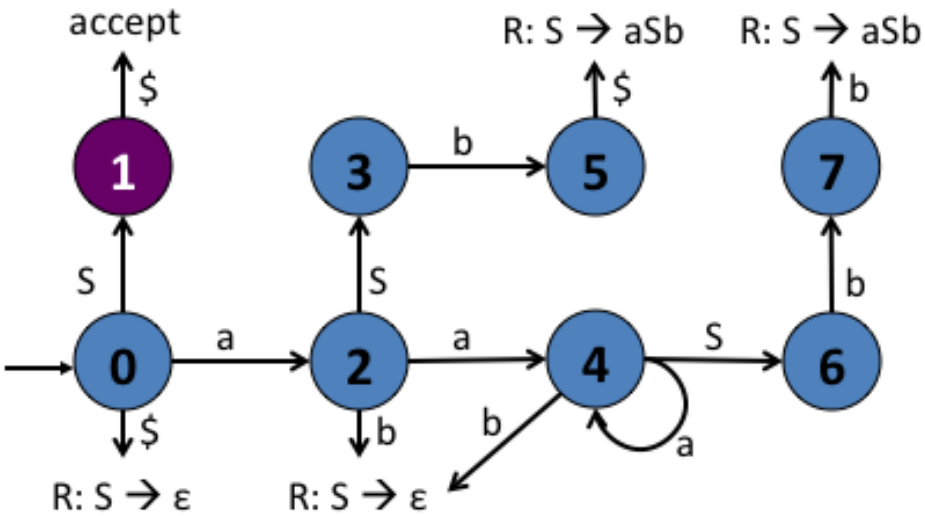
State 5
S → aSb. , $

State 6
S → aS.b , b

State 7
S → aSb. , b

Grammar is LR(1)

	a	b	$	S
0	S2		R: S → ε	1
1			accept	
2	S4	R: S → ε		3
3		S5		
4	S4	R: S → ε		6
5			R: S → aSb	
6		S7		
7		R: S → aSb		



Construction of an LR(1) Parsing Table

Let $C = \{I_0, I_1, ..., I_i, ..., I_n\}$ be the canonical LR(1) collection of items, with the corresponding states of the parser being 0, 1, ... , i, ... , n
Without loss of generality, let 0 be the initial state of the parser (containing the item $[S' \rightarrow .S,\ \$]$)
Parsing actions for state *i* are determined as follows
1. If $([A \rightarrow \alpha.a\beta,\ b] \in I_i)$ && $([A \rightarrow \alpha a.\beta,\ b] \in I_j)$
   set ACTION[i, a] = *shift j* /* *a* is a terminal symbol */
2. If $([A \rightarrow \alpha.,\ a] \in I_i)$
   set ACTION[i, a] = *reduce* $A \rightarrow \alpha$
3. If $([S' \rightarrow S.,\ \$] \in I_i)$ set ACTION[i, $] = *accept*

S-R or R-R conflicts in the table imply grammar is not LR(1)

4. If $([A \rightarrow \alpha.A\beta,\ a] \in I_i)$ && $([A \rightarrow \alpha A.\beta,\ a] \in I_j)$
   set GOTO[i, A] = *j* /* *A* is a nonterminal symbol */

All other entries not defined by the rules above are made *error*

Semantic Analysis with Attribute Grammars Part 1

Y.N. Srikant

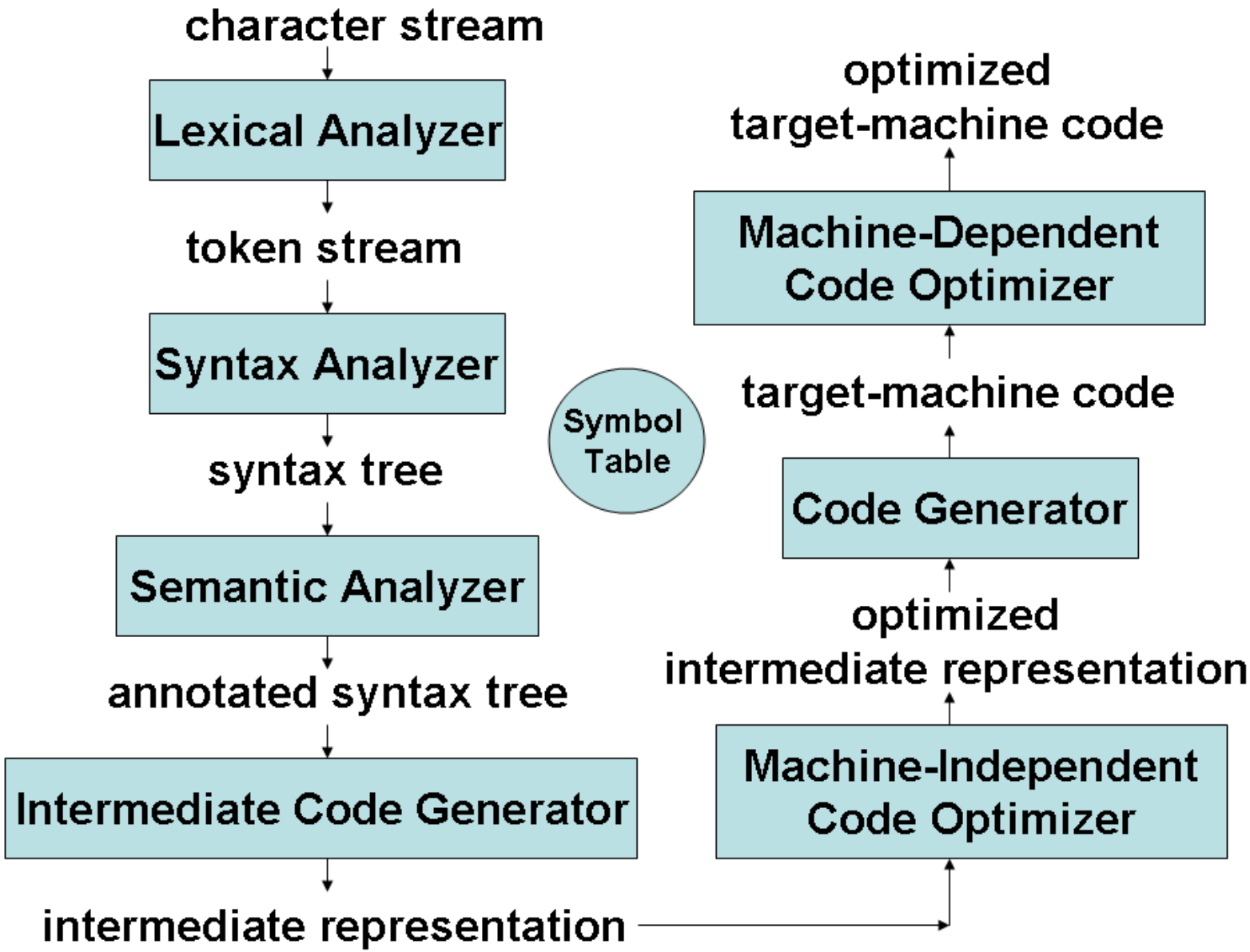
Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Introduction
- Attribute grammars
- Attributed translation grammars
- Semantic analysis with attributed translation grammars

Compiler Overview



Semantic Analysis

- Semantic consistency that cannot be handled at the parsing stage is handled here
- Parsers cannot handle context-sensitive features of programming languages
- These are *static semantics* of programming languages and can be checked by the semantic analyzer
  - Variables are declared before use
  - Types match on both sides of assignments
  - Parameter types and number match in declaration and use
- Compilers can only generate code to check *dynamic semantics* of programming languages at runtime
  - whether an overflow will occur during an aritmetic operation
  - whether array limits will be crossed during execution
  - whether recursion will cross stack limits
  - whether heap memory will be insufficient

Static Semantics

```
int dot_prod(int x[], int y[]){
  int d, i; d = 0;
  for (i=0; i<10; i++) d += x[i]*y[i];
  return d;
}
main(){
  int p; int a[10], b[10];
  p = dot_prod(a,b);
}
```

Samples of static semantic checks in *main*

- Types of *p* and return type of *dot_prod* match
- Number and type of the parameters of *dot_prod* are the same in both its declaration and use
- *p* is declared before use, same for *a* and *b*

Static Semantics: Errors given by gcc Compiler

```
int dot_product(int a[], int b[]) {...}

1 main(){int a[10]={1,2,3,4,5,6,7,8,9,10};
2 int b[10]={1,2,3,4,5,6,7,8,9,10};
3 printf("%d", dot_product(b));
4 printf("%d", dot_product(a,b,a));
5 int p[10]; p=dotproduct(a,b); printf("%d",p);}

In function ‘main’:
error in 3: too few arguments to fn ‘dot_product’
error in 4: too many arguments to fn ‘dot_product’
error in 5: incompatible types in assignment
warning in 5: format ‘%d’ expects type ‘int’, but
              argument 2 has type ‘int *’
```

Static Semantics

```
int dot_prod(int x[], int y[]){
  int d, i; d = 0;
  for (i=0; i<10; i++) d += x[i]*y[i];
  return d;
}
main(){
  int p; int a[10], b[10];
  p = dot_prod(a,b);
}
```

Samples of static semantic checks in *dot_prod*

- *d* and *i* are declared before use
- Type of *d* matches the return type of *dot_prod*
- Type of *d* matches the result type of "$*$"
- Elements of arrays *x* and *y* are compatible with "$*$"

Dynamic Semantics

```
int dot_prod(int x[], int y[]){
  int d, i; d = 0;
  for (i=0; i<10; i++) d += x[i]*y[i];
  return d;
}
main(){
  int p; int a[10], b[10];
  p = dot_prod(a,b);
}
```

Samples of dynamic semantic checks in *dot_prod*

- Value of *i* does not exceed the declared range of arrays *x* and *y* (both lower and upper)
- There are no overflows during the operations of "$*$" and "$+$" in `d += x[i]*y[i]`

Dynamic Semantics

```
int fact(int n){
   if (n==0) return 1;
   else return (n*fact(n-1));
}
main(){int p; p = fact(10); }
```

Samples of dynamic semantic checks in *fact*

- Program stack does not overflow due to recursion
- There is no overflow due to "*" in `n*fact(n-1)`

Semantic Analysis

- Type information is stored in the symbol table or the syntax tree
  - Types of variables, function parameters, array dimensions, etc.
  - Used not only for semantic validation but also for subsequent phases of compilation
- If declarations need not appear before use (as in C++), semantic analysis needs more than one pass
- Static semantics of PL can be specified using attribute grammars
- Semantic analyzers can be generated semi-automatically from attribute grammars
- Attribute grammars are extensions of context-free grammars

Attribute Grammars

- Let $G = (N, T, P, S)$ be a CFG and let $V = N \cup T$.
- Every symbol X of V has associated with it a set of *attributes* (denoted by $X.a$, $X.b$, etc.)
- Two types of attributes: *inherited* (denoted by $AI(X)$)and *synthesized* (denoted by $AS(X)$)
- Each attribute takes values from a specified domain (finite or infinite), which is its *type*
  - Typical domains of attributes are, integers, reals, characters, strings, booleans, structures, etc.
  - New domains can be constructed from given domains by mathematical operations such as *cross product, map*, etc.
  - *array*: a map, $\mathcal{N} \rightarrow \mathcal{D}$, where, $\mathcal{N}$ and $\mathcal{D}$ are domains of natural numbers and the given objects, respectively
  - *structure*: a cross-product, $A_1 \times A_2 \times \ldots \times A_n$, where n is the number of fields in the structure, and A_i is the domain of the i^{th} field

Attribute Computation Rules

- A production $p \in P$ has a set of attribute computation rules (functions)
- Rules are provided for the computation of
  - Synthesized attributes of the LHS non-terminal of p
  - Inherited attributes of the RHS non-terminals of p
- These rules can use attributes of symbols from the production p only
  - Rules are strictly local to the production p (no side effects)
- Restrictions on the rules define different types of attribute grammars
  - L-attribute grammars, S-attribute grammars, ordered attribute grammars, absolutely non-circular attribute grammars, circular attribute grammars, etc.

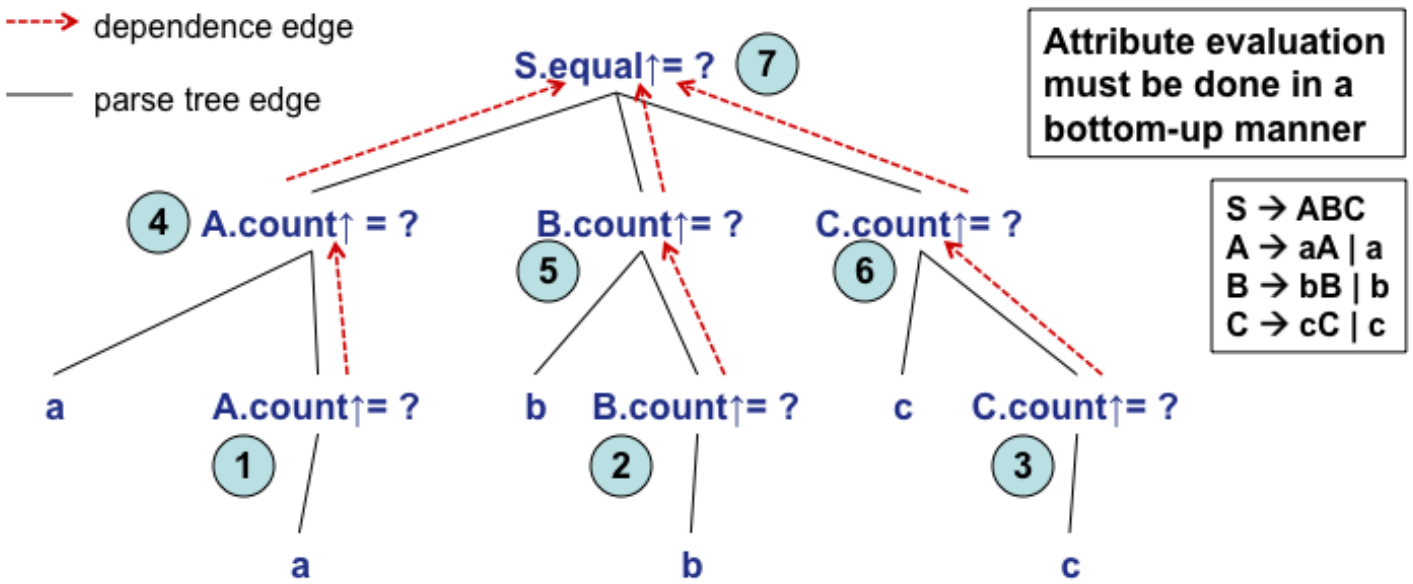
Synthesized and Inherited Attributes

- An attribute cannot be both synthesized and inherited, but a symbol can have both types of attributes
- Attributes of symbols are evaluated over a parse tree by making passes over the parse tree
- Synthesized attributes are computed in a bottom-up fashion from the leaves upwards
  - Always synthesized from the attribute values of the children of the node
  - Leaf nodes (terminals) have synthesized attributes initialized by the lexical analyzer and cannot be modified
  - An AG with only synthesized attributes is an *S-attributed grammar (SAG)*
  - YACC permits only SAGs
- Inherited attributes flow down from the parent or siblings to the node in question

Attribute Grammar - Example 1

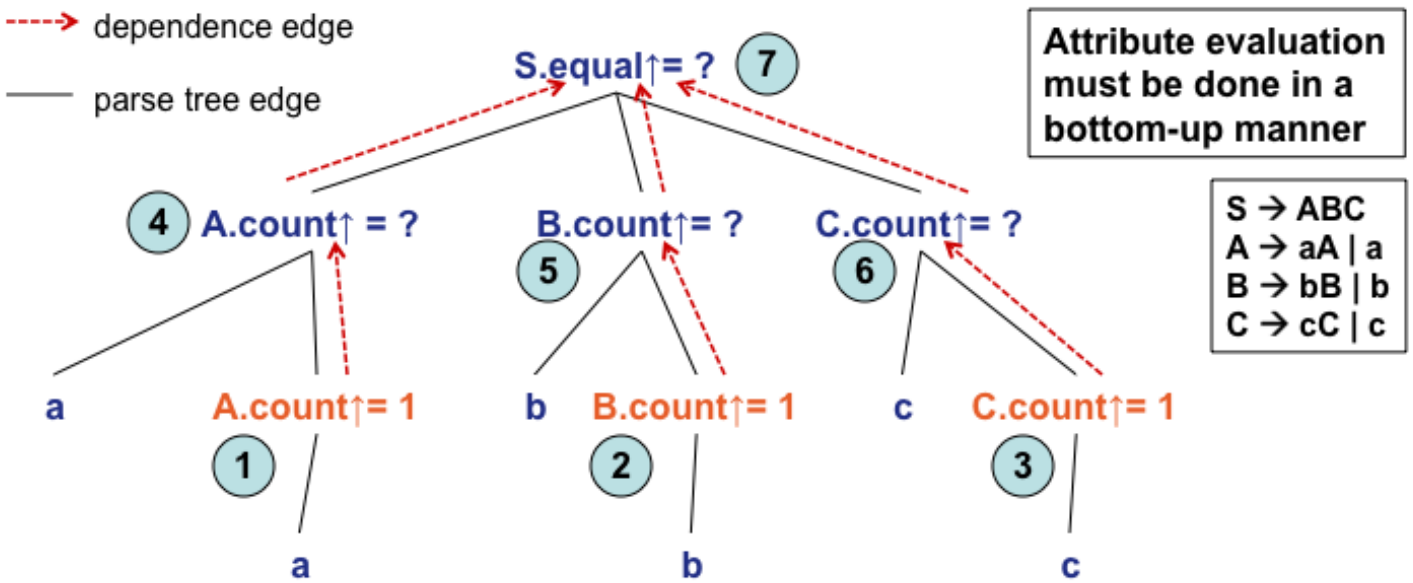
- The following CFG
  $S \rightarrow A\ B\ C,\ A \rightarrow aA \mid a,\ B \rightarrow bB \mid b,\ C \rightarrow cC \mid c$
  generates: $L(G) = \{a^m b^n c^p \mid m, n, p \geq 1\}$
- We define an AG (attribute grammar) based on this CFG to generate $L = \{a^n b^n c^n \mid n \geq 1\}$
- All the non-terminals will have only synthesized attributes
  - $AS(S) = \{equal \uparrow:\ \{T, F\}\}$
  - $AS(A) = AS(B) = AS(C) = \{count \uparrow:\ integer\}$

Attribute Grammar - Example 1 (contd.)



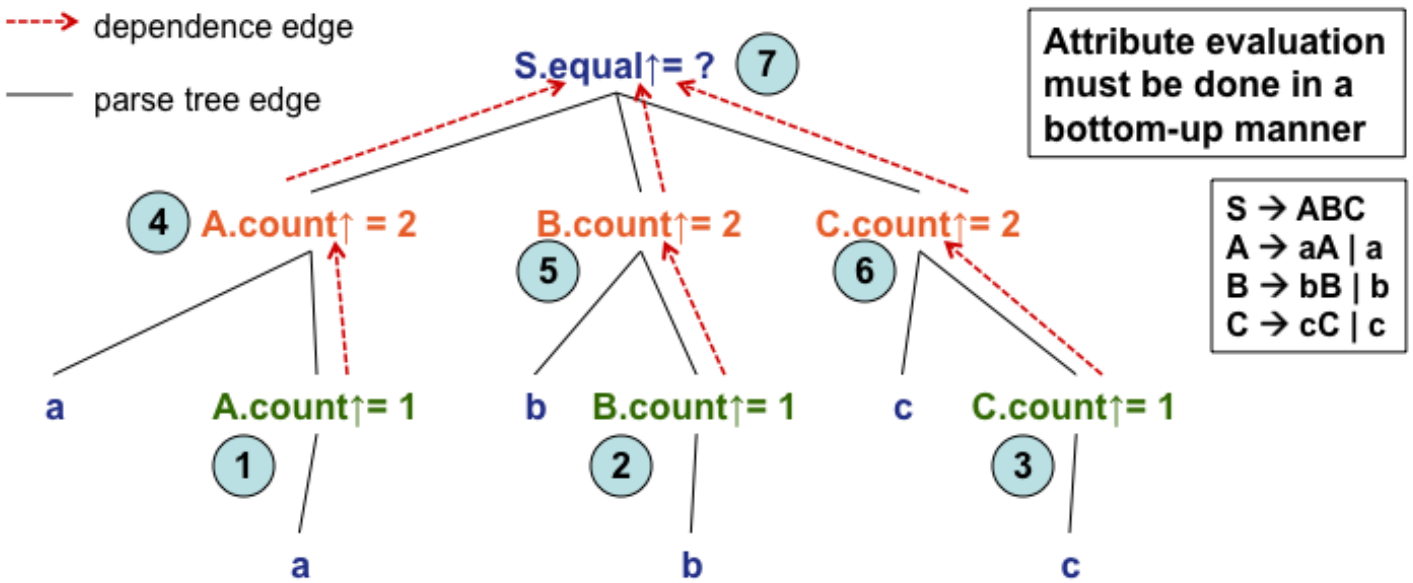
1. $S \rightarrow ABC \ \{S.equal \uparrow := \textit{if } A.count \uparrow = B.count \uparrow \ \& \ B.count \uparrow = C.count \uparrow \ \textit{then } T \textit{ else } F\}$
2. $A_1 \rightarrow aA_2 \ \{A_1.count \uparrow := A_2.count \uparrow +1\}$
3. $A \rightarrow a \ \{A.count \uparrow := 1\}$
4. $B_1 \rightarrow bB_2 \ \{B_1.count \uparrow := B_2.count \uparrow +1\}$
5. $B \rightarrow b \ \{B.count \uparrow := 1\}$
6. $C_1 \rightarrow cC_2 \ \{C_1.count \uparrow := C_2.count \uparrow +1\}$
7. $C \rightarrow c \ \{C.count \uparrow := 1\}$

Attribute Grammar - Example 1 (contd.)



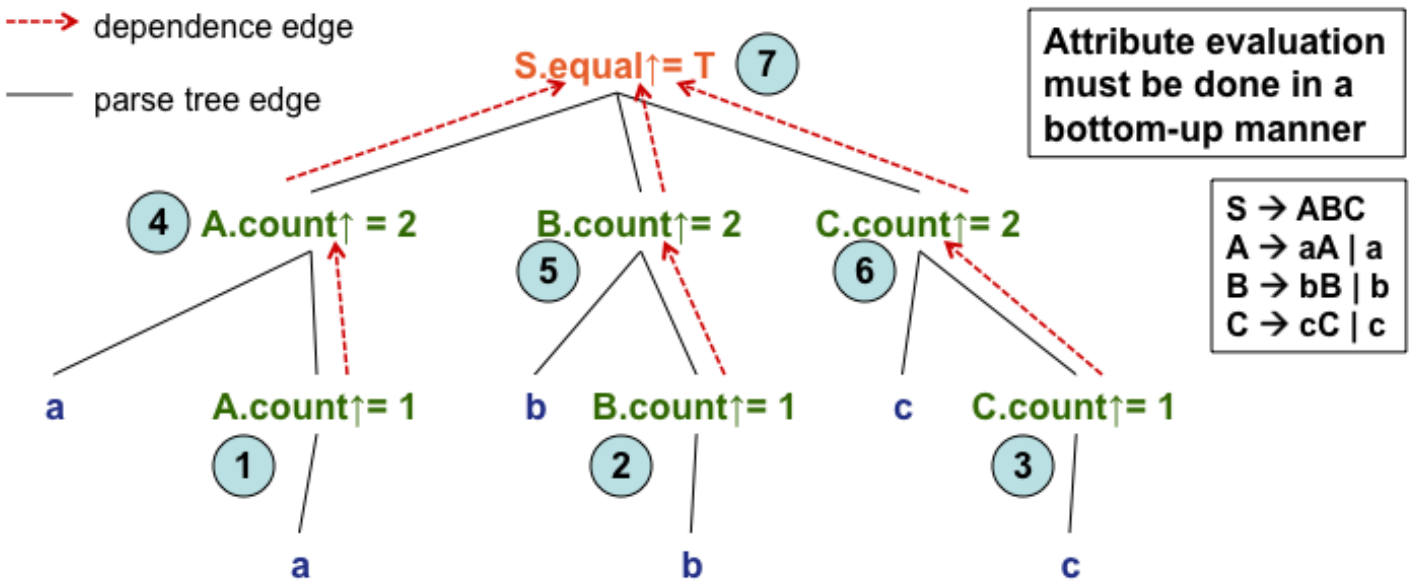
1. $S \rightarrow ABC \ \{S.equal \uparrow := \text{if } A.count \uparrow = B.count \uparrow \ \& \ B.count \uparrow = C.count \uparrow \text{ then } T \text{ else } F\}$
2. $A_1 \rightarrow aA_2 \ \{A_1.count \uparrow := A_2.count \uparrow +1\}$
3. $A \rightarrow a \ \{A.count \uparrow := 1\}$
4. $B_1 \rightarrow bB_2 \ \{B_1.count \uparrow := B_2.count \uparrow +1\}$
5. $B \rightarrow b \ \{B.count \uparrow := 1\}$
6. $C_1 \rightarrow cC_2 \ \{C_1.count \uparrow := C_2.count \uparrow +1\}$
7. $C \rightarrow c \ \{C.count \uparrow := 1\}$

Attribute Grammar - Example 1 (contd.)



1. $S \rightarrow ABC \ \{S.equal \uparrow := if\ A.count \uparrow = B.count \uparrow\ \&\ B.count \uparrow = C.count \uparrow\ then\ T\ else\ F\}$
2. $A_1 \rightarrow aA_2 \ \{A_1.count \uparrow := A_2.count \uparrow +1\}$
3. $A \rightarrow a \ \{A.count \uparrow := 1\}$
4. $B_1 \rightarrow bB_2 \ \{B_1.count \uparrow := B_2.count \uparrow +1\}$
5. $B \rightarrow b \ \{B.count \uparrow := 1\}$
6. $C_1 \rightarrow cC_2 \ \{C_1.count \uparrow := C_2.count \uparrow +1\}$
7. $C \rightarrow c \ \{C.count \uparrow := 1\}$

Attribute Grammar - Example 1 (contd.)



1. $S \rightarrow ABC$ $\{S.equal\uparrow := if\ A.count\uparrow = B.count\uparrow\ \&\ B.count\uparrow = C.count\uparrow\ then\ T\ else\ F\}$
2. $A_1 \rightarrow aA_2$ $\{A_1.count\uparrow := A_2.count\uparrow + 1\}$
3. $A \rightarrow a$ $\{A.count\uparrow := 1\}$
4. $B_1 \rightarrow bB_2$ $\{B_1.count\uparrow := B_2.count\uparrow + 1\}$
5. $B \rightarrow b$ $\{B.count\uparrow := 1\}$
6. $C_1 \rightarrow cC_2$ $\{C_1.count\uparrow := C_2.count\uparrow + 1\}$
7. $C \rightarrow c$ $\{C.count\uparrow := 1\}$

Attribute Dependence Graph

- Let T be a parse tree generated by the CFG of an AG, G.
- The *attribute dependence graph* (dependence graph for short) for T is the directed graph, $DG(T) = (V, E)$, where

  $V = \{b | b$ is an attribute instance of some tree node$\}$, and

  $E = \{(b, c) | b, c \in V$, b and c are attributes of grammar symbols in the same production p of B, and the value of b is used for computing the value of c in an attribute computation rule associated with production $p\}$

Attribute Dependence Graph

- An AG G is *non-circular*, iff for all trees T derived from G, $DG(T)$ is acyclic
  - Non-circularity is very expensive to determine (exponential in the size of the grammar)
  - Therefore, our interest will be in subclasses of AGs whose non-circularity can be determined efficiently
- Assigning consistent values to the attribute instances in DG(T) is *attribute evaluation*

Attribute Evaluation Strategy

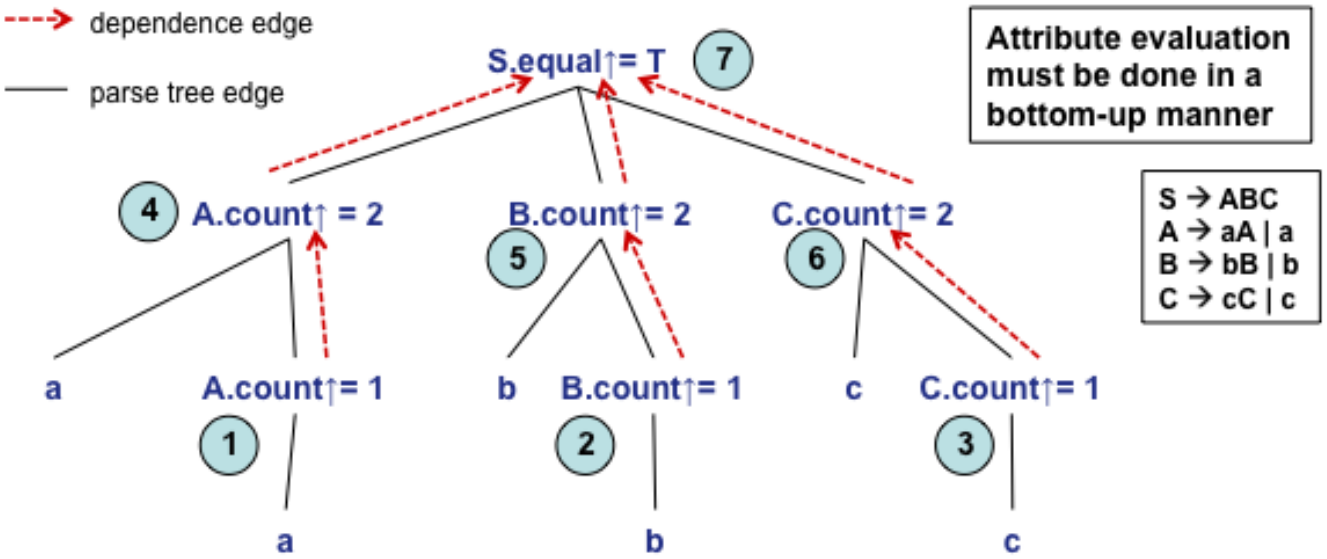
- Construct the parse tree
- Construct the dependence graph
- Perform topological sort on the dependence graph and obtain an evaluation order
- Evaluate attributes according to this order using the corresponding attribute evaluation rules attached to the respective productions
- Multiple attributes at a node in the *parse tree* may result in that node to be visited multiple number of times
  - Each visit resulting in the evaluation of at least one attribute

Attribute Evaluation Algorithm

Input: A parse tree T with unevaluated attribute instances
Output: T with consistent attribute values

```
{  Let (V, E) = DG(T);
   Let W = {b | b ∈ V & indegree(b) = 0};
   while W ≠ φ do
     {  remove some b from W;
        value(b) := value defined by appropriate attribute
                    computation rule;
        for all (b, c) ∈ E do
         {  indegree(c) := indegree(c) − 1;
            if indegree(c) = 0 then W := W ∪ {c};
         }
     }
}
```

Dependence Graph for Example 1



1,2,3,4,5,6,7 and 2,3,6,5,1,4,7 are two possible evaluation orders. 1,4,2,5,3,6,7 can be used with LR-parsing. The right-most derivation is below (its reverse is LR-parsing order)

S => ABC => ABcC => ABcc => AbBcc => Abbcc => aAbbcc => aabbcc

1. A.count = 1 {A → a, {A.count := 1}}
4. A.count = 2 {A_1 → aA_2, {A_1.count := A_2.count + 1}}
2. B.count = 1 {B → b, {B.count :=1}}
5. B.count = 2 {B_1 → bB_2, {B_1.count := B_2.count + 1}}
3. C.count = 1 {C → c, {C.count :=1}}
6. C.count = 2 {C_1 → cC_2, {C_1.count := C_2.count + 1}}
7. S.equal = 1 {S → ABC, {S.equal := *if* A.count = B.count & B.count = C.count *then* T else F}}

Attribute Grammar - Example 2

- AG for the evaluation of a real number from its bit-string representation
  Example: 110.101 = 6.625
- $N \rightarrow L.R,\ L \rightarrow BL \mid B,\ R \rightarrow BR \mid B,\ B \rightarrow 0 \mid 1$
- $AS(N) = AS(R) = AS(B) = \{value\uparrow: real\}$,
  $AS(L) = \{length\uparrow: integer,\ value\uparrow: real\}$
  1. $N \rightarrow L.R\ \{N.value\uparrow := L.value\uparrow + R.value\uparrow\}$
  2. $L \rightarrow B\ \{L.value\uparrow := B.value\uparrow;\ L.length\uparrow := 1\}$
  3. $L_1 \rightarrow BL_2\ \{L_1.length\uparrow := L_2.length\uparrow + 1;$
     $L_1.value\uparrow := B.value\uparrow * 2^{L_2.length\uparrow} + L_2.value\uparrow\}$
  4. $R \rightarrow B\ \{R.value\uparrow := B.value\uparrow / 2\}$
  5. $R_1 \rightarrow BR_2\ \{R_1.value\uparrow := (B.value\uparrow + R_2.value\uparrow)/2\}$
  6. $B \rightarrow 0\ \{B.value\uparrow := 0\}$
  7. $B \rightarrow 1\ \{B.value\uparrow := 1\}$