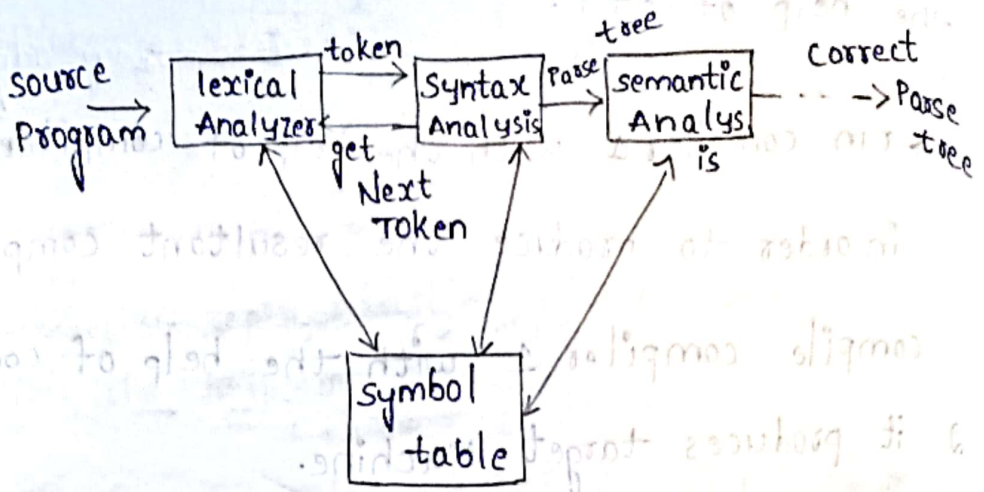
Unit-II

Role of a parser in syntax Analysis:



→ The major task of the parser is it checks whether the token is following the Grammar of the language or not.

→ The parser accepts the token from lexical Analyzer and it sends another request to lexical Analyzer in order to get the next token from the lexical analyzer

→ The tokens are following the Grammar of the language then it produces a parsetree and it is given as input to the next phase of the compilation Process.

→ If there are some syntax errors in corresponding tokens. The parser will report error messages to the user and now it is the duty of user to Correct those error messages.

Context free Grammar:

A context free Grammar is defined as

contxable contexts.

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

where V is a set of variables of non-terminals (uppercase letters)

T is a set of Terminal symbols (lower case letters or operators like (,) (,))

P is a production rules

$A \rightarrow \alpha$

where 'A' is a non terminal

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

S is a start symbol (non terminal).

Derivation:

→ The process of Derivation always start from the start symbol.

→ It is the process of applying a sequence of Production rules in order to derive a string of the corresponding language.

→ It is classified into two types.

→ left most Derivation

→ right most Derivation

left most Derivation:

In each step left most non terminal will

be expanded.

Right most derivation:

In each step the Right Most non-terminal will be expanded.

consider the following Grammar

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

obtain Lmd, Rmd for the string id+id*id.

i) Lmd

$E \rightarrow E+E$ ($\because E \rightarrow E+E$)

$\rightarrow id+E$ ($\because E \rightarrow id$)

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

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

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

Rmd

$E \rightarrow E+E \rightarrow E+E*E$

$\rightarrow E+E*id$

$\rightarrow E*id*id$

+

$\rightarrow E+id*id$

$\rightarrow id+id*id$

Parse tree / derivation tree.

It is a pictorial representation of derivation Process.

The rules for constructing a parse tree:

→ A root note most be a start symbol.

→ Internal nodes are nothing but non-terminals.

→ The terminals are nothing but leaf nodes.

→ Yield of the tree is nothing but the connection of nodes from left to right. → "Yield"

→ Ex:

consider the following Grammar

$S \rightarrow 0A|1B|0|1$

$A \rightarrow 0S|1B|1$

$B \rightarrow 0A|1S$

construct LMD and parse tree for (i) 0101 (ii) 1100101.

LMD

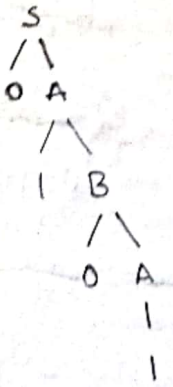
$S \rightarrow 0A$

$\rightarrow 01B$

$\rightarrow 010A$

$\rightarrow 0101$

Parse tree:



yield: 0101

RMD

$S \rightarrow 1B$

$\rightarrow 11S$

$\rightarrow 110A$

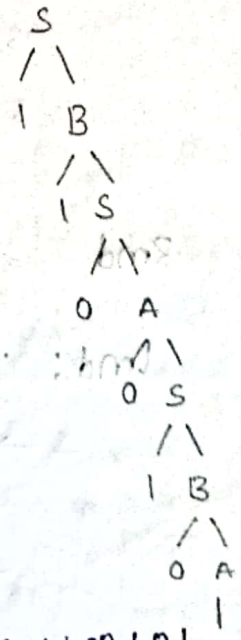
$\rightarrow 1100S$

$\rightarrow 11001B$

$\rightarrow 110010A$

$\rightarrow 1100101$

Parse tree:



yield: 1100101

Ex: consider the Grammar

$S \rightarrow (L) \mid a$

$L \rightarrow L,S \mid S$

a) what are the Non terminals, Terminals and start symbol.

b) Find Lmd, Rmd & parse tree for

i) (a,a) ii) (a,(a,a)).

c) what language does the grammar generate.

Any: The Grammar generate the strings which means well paranthesis.

a) Any: Non-terminals = {S, L}

Terminals = {(,), a, ,}

start symbol = S

b)
Sol:

Lmd

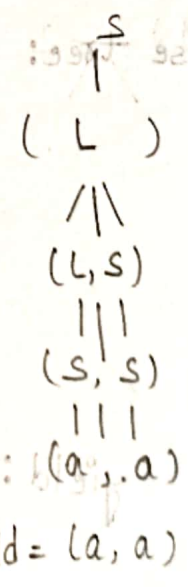
$S \rightarrow (L)$

$S \rightarrow (L,S)$

$S \rightarrow (S,S)$

$S \rightarrow (a,a)$

Parse tree



yield = (a, a)

Rmd

Lmd:

$S \rightarrow (L)$

$\rightarrow (L,S)$

$\rightarrow (S,S)$

$\rightarrow (a,S)$

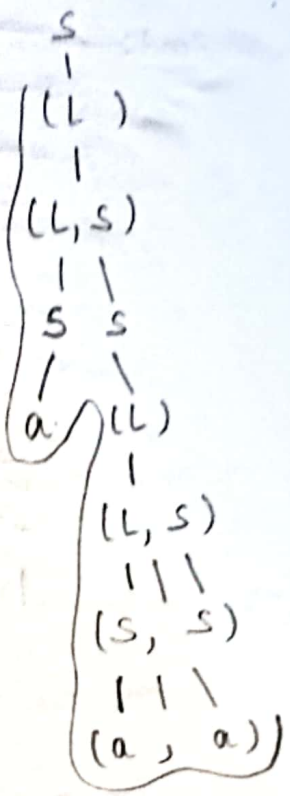
$\rightarrow (a,(L))$

→ (a, (L,S))

→ (a, (S,S))

→ (a, (a,a))

Parse tree



Yield = (a, (a,a))

Ex: Design lmd, Rmd, for the string aaabaab

grammar is $S \rightarrow aS \mid aSbS \mid \epsilon$

lmd: $S \rightarrow aS$

$S \rightarrow aSbS$

$S \rightarrow \epsilon$

$S \rightarrow aS$

$\rightarrow aaS$

$\rightarrow aaaSbS$

$\rightarrow aaa\epsilon bS$

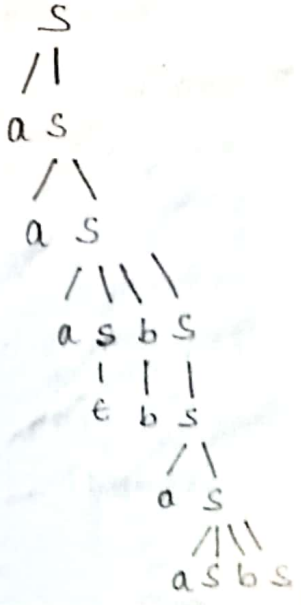
$\rightarrow aaabaS$

$\rightarrow aaabaaSbS$

$\rightarrow aaabaa\epsilon bS$

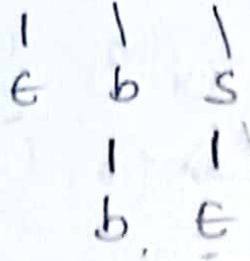
$\rightarrow aaabaabS$

Parse tree



$\rightarrow aaabaab\epsilon$

$\Rightarrow aaabaab$



Yield: aaabaab

Rmd:

$S \rightarrow aS$

$\rightarrow aasbS$

$\rightarrow aasbasbs$

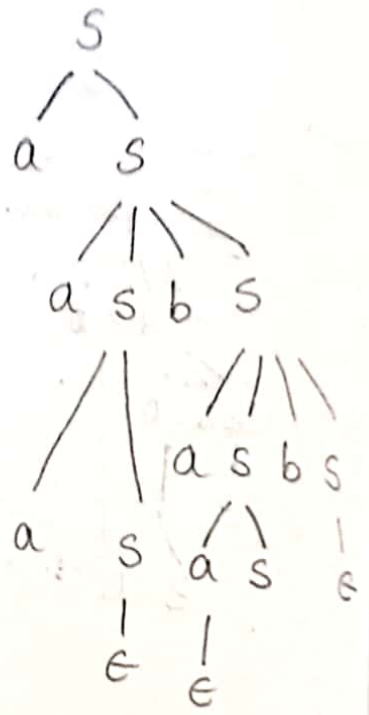
$\rightarrow aasbasb\epsilon$

$\rightarrow aaasbaasb$

$\rightarrow aaa\epsilon baasb$

$\rightarrow aaabaa\epsilon b$

$\rightarrow aaabaab.$



check whether the string, aabb is accepted or not

$S \rightarrow aAB \mid bA \mid \epsilon$

$A \rightarrow aAb \mid \epsilon$

$B \rightarrow bB \mid \epsilon$

$S \rightarrow aAB$

$S \rightarrow bA$

$S \rightarrow \epsilon$

$A \rightarrow aAb$

$A \rightarrow \epsilon$

$B \rightarrow bB$

$B \rightarrow \epsilon$

$S \rightarrow aAB$

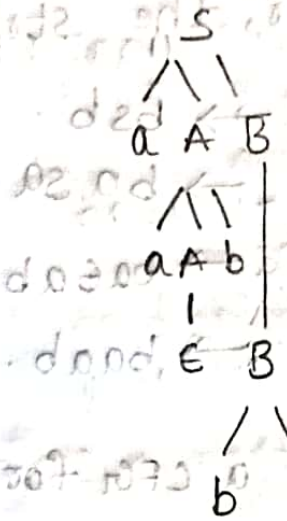
$S \rightarrow aaAbB$

$S \rightarrow aa\epsilon bB$

$S \rightarrow aabB$

$S \rightarrow aabb\epsilon$

$S \rightarrow aabb$



Yes, the given string aabb is accepted.

Ex: construct a CFG to generate palindromes are over $\{a, b\}$.

$\{aa, bb, abba, aba, baab, \ldots\}$

$S \rightarrow bSb$

$S \rightarrow aSa$

$S \rightarrow a|b|\epsilon$

consider, the string baab

$S \rightarrow bSb$

$\rightarrow baSab$

$\rightarrow ba\epsilon ab$

$\rightarrow baab.$

Ex: construct a CFG for set of strings with equal no. of a's & b's.

$\{a, b, abba, aabb, \ldots\}$

$S \rightarrow aSbS$

$S \rightarrow bSaS$

$S \rightarrow \epsilon$

Consider, the string abba

$S \rightarrow aSbS$

$\rightarrow a\epsilon bS$

$\rightarrow abbSaS$

$\rightarrow abb\epsilon aS$

$\rightarrow abb\epsilon a\epsilon$

$\rightarrow abba$

Ex: construct a CFG for $a^n b^n$

$\{ab, aabb, aaabbb, \ldots\}$

$S \rightarrow aAb$

$A \rightarrow aAb$

$A \rightarrow \epsilon$

consider, the string aabb

$S \rightarrow aAb$

$\rightarrow aaAbAb$

$\rightarrow aa\epsilon b\epsilon b$

$\rightarrow aa\epsilon b\epsilon b$

$\rightarrow aabb.$

Ex: construct a CFG for the language

$L = \{wc\ w^R / \text{where } w \in (a,b)^*\}$

{aacaa, abcba, abbcbba ....}

$S \rightarrow asa$

$S \rightarrow bsb$

$S \rightarrow \epsilon / c$

consider, the string abbcbba.

$S \rightarrow asa$

$\rightarrow absba$

$\rightarrow abbsbba$

$\rightarrow abbcbba$

Ex: construct a CFG for the language having any no. of a's over the set $\Sigma = \{a\}$.

$RE = a^*$

$\{a, \epsilon, aa, aaa, \ldots\}$

$S \rightarrow aS$

$S \rightarrow \epsilon$

consider, the string "aa"

$S \rightarrow aS$

$\rightarrow aaS$

$\rightarrow aa\epsilon$

$\rightarrow aa$

1) Regular expression $(0+1)^*$

$L = \{\epsilon, 0, 1, 01, 0001, 0011 - - -\}$

$S \rightarrow 0S/1S$

$S \rightarrow \epsilon$

2) Atleast one occurence of 000 over $\{0,1\}$.

$R.E = (0+1)^* 000 (0+1)^*$

$S \rightarrow ABA$

$B \rightarrow 000$

$A \rightarrow 0A|1A|\epsilon$

consider, the string = 000001

$S \rightarrow ABA$

$\rightarrow 0ABA$

$\rightarrow 0\epsilon 000A$

$\rightarrow 0\epsilon 0000A$

$\rightarrow 0\epsilon 00001A$

$\rightarrow 000001\epsilon$

$\rightarrow 000001$

3) Atleast two a's over $\{a,b\}$

$S \rightarrow AaAaA$

$A \rightarrow aA|bA|\epsilon$

Ambiguous Grammar:

A grammar is said to be ambiguous grammar, if it is generates more than one Parse tree of the corresponding input string.

Prove given Grammar is ambiguous or not.

$S \to asbs$

$S \to bsas$

$S \to \epsilon$

$S \to asbs$

$\to absasbs$

$\to abab$

$S \to asbs$

$\to a\epsilon bs$

$\to abasbs$

$\to abab$

This grammar is ambiguous.

Elimination of left Recursion:

→ A Grammar is in a left Recursion if the production Rules are in the form

$A \to A\alpha / B$

where 'A' is a non terminal,

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

→ The top-down parser can not handle the Grammar which contains left Recursive production

→ we have to eliminate the left Recursive Production by using two rules.

$A \to BA'$

$A' \to \alpha A' / \epsilon$

Ex:

Eliminate left Recursion for the following grammar:

$E \to E+T/T$

$T \to T* F/F$

$F \to (E) / id.$

$$\underset{A}{E} \rightarrow \underset{A}{E} \underset{\alpha}{+T} \mid \underset{\beta}{T}$$

$$\epsilon \rightarrow T\epsilon'$$

$$E' \rightarrow +TE' \mid \epsilon$$

$$\underset{A}{T} \rightarrow \underset{A}{T} \underset{\alpha}{*F} \mid \underset{\beta}{F}$$

$$T \rightarrow FT'$$

$$T \rightarrow *FT' \mid \epsilon$$

$$F \rightarrow (\epsilon) \mid id$$

After, Eliminating the left recurssion.

$$E \rightarrow TE'$$

$$E' \rightarrow TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

$$F \rightarrow (E) \mid id$$

Elimination of left Factoring:

left factoring is a process to pa Transform: mar with common prefixes.

we, can say that a Grammar contains left factoring, the production Rules are in the form of

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \ldots\ldots \mid \gamma_1 \mid \gamma_2 \mid \gamma_3 \ldots\ldots$$

The top down parsers cannot handled the Grammars which contains left factoring have to eliminate the left factoring by using two rules.

$A \longrightarrow \alpha A' \mid \gamma_1 \mid \gamma_2$

where $A' \longrightarrow \beta_1 \mid \beta_2 \mid \ldots$

Ex: $S \longrightarrow iEtS \mid iEtSeS \mid a$

$E \longrightarrow b$

$S \longrightarrow \underset{\alpha}{iEtS}\underset{\beta_1}{\epsilon} \mid \underset{\alpha}{iEtS}\underset{\beta_1}{eS} \mid \underset{\gamma_1}{a}$

A

The Grammar after eliminating the left factoring.

$S \longrightarrow iEtSS' \mid a$

$S' \longrightarrow \epsilon \mid eS$

$E \longrightarrow b.$

Ex: $A \longrightarrow aAB \mid aA \mid a$

$B \longrightarrow bB \mid b$

$A \longrightarrow \underset{\alpha}{a}\underset{\beta_1}{AB} \mid \underset{\alpha}{a}\underset{\beta_2}{A\epsilon} \mid \underset{\gamma_1}{a}$

$A \longrightarrow aA' \mid a$

$A' \longrightarrow AB \mid A$

or $B \longrightarrow \underset{\alpha}{b}\underset{\beta_1}{B} \mid \underset{\alpha}{b}$

$B \longrightarrow b\beta$

$B' \longrightarrow B \mid \epsilon$

Ex: $X \longrightarrow X+X \mid X*X \mid D$

$D \longrightarrow 1 \mid 2 \mid 3$

$X \longrightarrow \underset{\alpha}{X}\underset{\beta_1}{+X} \mid \underset{\alpha}{X*X}\underset{\beta_2}{} \mid \underset{\gamma_1}{D}$

$X \longrightarrow XX' \mid D$

$X' \longrightarrow +X \mid *X$

$D \longrightarrow 1 \mid 2 \mid 3$

Ex: $E \rightarrow T+E \mid T$

$T \rightarrow int \mid int * T \mid (E)$

$E \rightarrow \underset{\alpha}{T} \underset{\beta_1}{+E} \mid \underset{\alpha}{T}$

$T \rightarrow \underset{\alpha}{int} \underset{\beta_1}{\epsilon} + \underset{\alpha}{int} \underset{\beta_2}{* T} \mid \underset{\gamma_1}{(E)}$

$T \rightarrow TE' \mid E$

$E' \rightarrow +E \mid \epsilon$

$T \rightarrow int\ T' \mid (E)$

$T' \rightarrow E \mid * T$

First and Follow:-

First:

To calculate first we need to follow the follo rules.

1) If $A \rightarrow a\alpha$, $\alpha \in (V \cup T)^*$

then first(A) = {a}

2) If $A \rightarrow \epsilon$ them first(A) = {ε}

3) If $A \rightarrow BC$ then first (A) = First (B). if First(B doesnot contains ε.

4) If First (B) contains ε then First (A) = First (B) ∪ First (C).

Follow:

1) If 'B' 's' is a start symbol the follow of = {$}

2) If $A \rightarrow \alpha B\beta$ then follow(B) = first(β) if first(β doesn't contain (β) ε.

3) → If $A \to \alpha B$ then Follow (B) = Follow (A).

4) If $A \to \alpha B \beta$ where $B \to \epsilon$ then Follow (B) = Follow (A)

Ex: calculate First and Follow:

$E \to TE'$

$E' \to +TE' \mid \underline{E}\ \epsilon$

$T \to FT'$

$T' \to *FT' \mid \underline{E}\ \epsilon$

$F \to (E) \mid id$

Sol:- First (E) = { (, id } Follow (E) = { $,) }

First (E') = { +, ε } Follow (E') = { $,) }

First (T) = { (, id } Follow (T) = { +, $,) }

First (T') = { *, ε } Follow (T') = { +, $,) }

First (F) = { (, id } Follow (F) = { +, $,), * }.

Ex: calculate First & Follow

$S \to ABCDE$

$A \to a \mid \epsilon$

$B \to b \mid \epsilon$

$C \to c$

$D \to d \mid \epsilon$

$E \to e \mid \epsilon$

Sol: First (S) = { a, b, c } Follow (S) = { $ }

First (A) = { a, ε } Follow (A) = { b, c }

First (B) = { b, ε } Follow (B) = { c }

First (C) = { c } Follow (C) = { d, e, $ }

First (D) = { d, ε } Follow (D) = { e, $ }

First (E) = { e, ε } Follow (E) = { $ }

Ex: calculate First & Follow

$S \rightarrow Bd/cd$

$B \rightarrow aB/\epsilon$

$C \rightarrow cC/\epsilon$

First (S) = {a, b, c, d} Follow(S) = {$}

First (B) = {a, ε} Follow(B) = {b}

First (C) = {c, ε} Follow(C) = {d}

Ex: calculate First & Follow.

$S \rightarrow ACB/CbB/Ba$

$A \rightarrow da/BC$

$B \rightarrow g/\epsilon$

$C \rightarrow h/\epsilon$

First (S) = {d, g, h, b, a} Follow(S) = {$}

First (A) = {d, g, h, ε} Follow(A) = {h, g, $}

First (B) = {g, ε} Follow(B) = {$, a, h}

First (C) = {h, ε} Follow(C) = {g, $, b, h}

Ex: calculate First & Follow.

$S \rightarrow aABb$

$A \rightarrow C/\epsilon$

$B \rightarrow d/\epsilon$

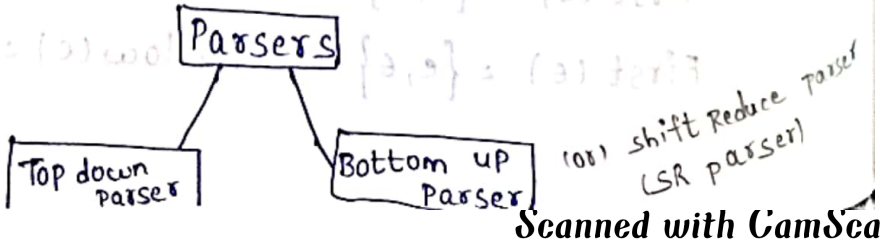
First(S) = {a} Follow(S) = {$}

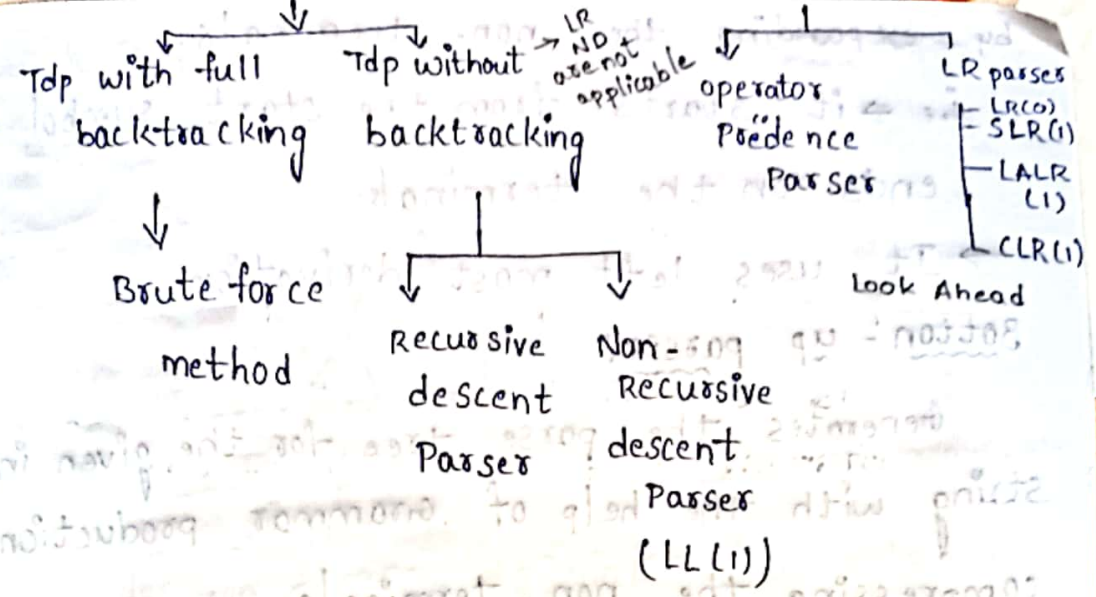
First (A) = {c, ε} Follow(A) = {c, d, $}

First (B) = {d, ε} Follow(B) = {d, $}

Types of parsers:

Parsers are derived into two types.





variety of grammers	Topdown	bottom up
1) Ambiguous	X	X
2) unambiguous	✓	✓
3) LR	X	✓
4) RR	✓	✓
5) ND	X	✓
6) Deterministic	✓	✓

Parser:

- It is a phase of a compiler which takes token as a input and with the help of context free Grammar converts it into corresponding tokens into parse tree.
- Parser is also called syntax analyzer

Top-down parser:

Generates parse tree for the given input string with the help of Grammar productions

by expanding the non-terminals.

i.e → it starts from the start symbol and ends on the terminals.

→ It uses left most derivation.

Botton-up parser:

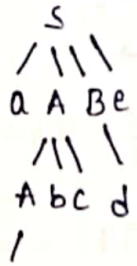
Generates the parse tree for the given input string with the help of Grammar productions by compressing the non-terminals.

i.e., It starts from the terminals and ends in the start symbol.

→ It uses Reverse of Right most Derivation.

Top down parsing:

Ex:



abbcde

S → aABe
A → Abc/b
B → d

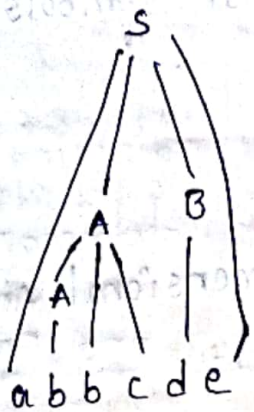
String: abbcde

S → aABe
→ aAbcBe
→ abbcBe
→ abbcde

At Every point we have to decide what is the next production we should use.

→ After matter.

Bottom-up parsing:



Main decision we have to make is when to reduce the given terminal.

scanning from left to right - and - one character at a time

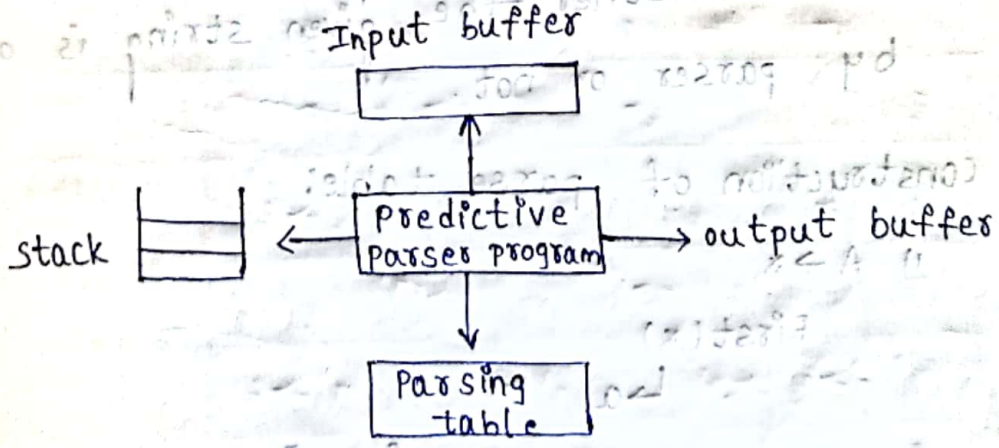
LL(1) or Non-Recursive descent parser

RMD: S → aABe
S → aAde
→ aAbcde
→ abbcde

(or)

Predictive parser:

Model of LL(1) parser



Input buffer:

It stores the Input string that is to be Parsed.

Stack:

It is a data structure which works on the

principle of

stack contains Grammar symbols initially '$' is stored in the stack.

Parsing table:

It is a two dimensional Array which is in the form of M[A, a]

where, A → non-terminals

a → Terminals.

output Buffer:

Stores the Actions which are performed by the user.

construction of LL(1) parser:

1) → Elimination of left Recursion.
2) → Elimination of left factoring.
3) → calculating First and Follow.
4) → construction of parsing table.
5) → check whether the given string is accepted by parser or not.

construction of parse table:

1) A → α

First(α)

↳ a

Add A → α to M[A, a]

2) If First(α) contains ∈ or A → ∈

Follow(A) = b

A → ∈ to M[A, b]

3) The Remaining entries of the parsing table are filled with errors.

Ex: $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id.$

Step:1 Eliminating the left Recursion.

Given Grammar is $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id.$

It is in the form of $A \to A\alpha \mid \beta$

Rules:

$A \to \beta A'$

$A' \to \alpha A' \mid \epsilon$

$E \to \underbrace{E}_{A} \underbrace{+T}_{\alpha} \mid \underbrace{T}_{\beta}$

consider, $E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to \underbrace{T}_{A} \underbrace{*F}_{\alpha} \mid \underbrace{F}_{\beta}$

Consider, $T \to FT'$

$T \to *FT' \mid \epsilon$

Consider $F \to (E) \mid id$

After, Eliminating the left recursion.

The Resultant Grammar is

$E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

$F \to (E) \mid id$

Step-2: Eliminating the Left factoring.

The Grammar is

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

The Answer is $E \to TE'$, $E' \to +TE' \mid \epsilon$, $T \to FT'$, $T' \to *FT' \mid \epsilon$, $F \to (E) \mid id$

Here, There is no left factoring process to eliminate in the Grammar.

step-3: calculating First and Follow.

First, we have note the Given Grammar.

The given Grammar is $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id.$

First(E) = { (, id }

First(E') = { +, ϵ }

First(T) = { (, id }

First(F) = { (, id }

First(T') = { *, ϵ }

Follow(E) = { $,) }

Follow(E') = { $,) }

Follow(T) = { +, $ }

Follow(T') = { +, $,) }

Follow(F) = { *, +, $ }

	+	*	(	)	id	$
E			$E \to TE'$		$E \to TE'$	
E'	$E' \to +TE'$			$E' \to \epsilon$		$E' \to \epsilon$
T			$T \to FT'$		$T \to FT'$	
T'	$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$		$T' \to \epsilon$
F			$F \to (E)$		$F \to id$	

1) $\underset{A}{E} \rightarrow \underset{\alpha}{TE'}$

First $(TE') = \{(, id\}$

Add $E \rightarrow TE'$ to $M[(, (]$

$M[(, id]$

2)a) $\underset{A}{E'} \rightarrow \underset{\alpha}{+TE'}$

First $(+TE') = \{+\}$

Add $E' \rightarrow +\ TE'$ to $M[E', +]$

2) b) $\underset{A}{E'} \rightarrow \underset{\epsilon}{\epsilon}$

Follow $(E') = \{\$,)\}$

Add $E' \rightarrow \epsilon$ to $M[E', \$]$

& $M[E',)]$

3) $\underset{A}{T} \rightarrow \underset{\alpha}{FT'}$

First $(FT') = \{(, id\}$

Add $T \rightarrow FT'$ to $M[T, (,]$

& $M[T, id]$

4)a) $\underset{A}{T'} \rightarrow \underset{\alpha}{*FT'} \mid \epsilon$

First $(*FT') = \{*\}$

Add $T' \rightarrow *FT'$ to $M[T', *]$

4)b) $\underset{A}{T'} \rightarrow \underset{\alpha}{\epsilon}$

Follow $(T') = \{+, \&,)\}$

Add $T' \rightarrow \epsilon$ to $M[T', +]$

$M[T', \$]$

$M[T',)]$

5)a) $F \to (E)$

First((E)) = { (}

Add $F \to (E)$ to M[F, (]

5)b) $F \to id$

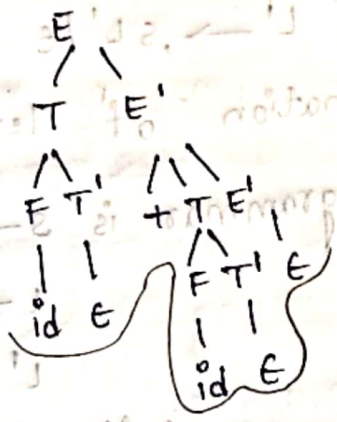
First(id) = { id }

Add $F \to id$ to M[F, id]

step:5. check whether the Given string is Accepted or not.

stack	input string	Action
$E	id + id$	$E \to TE'$
$E'T	id + id$	$T \to FT'$
$E'TF	id + id$	$F \to id$
$E'T'id	id + id$	POP
$E'T'	+ id$	$T' \to \epsilon$
$E'	+ id$	$E' \to +TE'$
$E'T+	+ id$	POP
$E'T	id$	$T \to FT'$
$E'T'F	id$	$F \to id$
$E'T'id	id$	POP
$E'T'	$	$T' \to \epsilon$
$E'	$	$E' \to \epsilon$
$	$	Accepted

Here, After Parsing the entire input string with the stack contains only $ 'dollar' symbol Then we can say that our string is accepted by the parser.



id + id ϵ ϵ
= id + id.

Ex: construct predictive parser for

$S \rightarrow (L) / a$

$L \rightarrow L,S / S$.

Step-1: Elimination of left recursion

$A \rightarrow A\alpha / \beta$

replace with $A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \epsilon$

$S \rightarrow (L) / a$ - no left recursion

consider, $L \rightarrow L, S / S$

replace with $A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \epsilon$

where, $A = L$

$\alpha = ,S$

$\beta = S$

$A' = L'$

$L \rightarrow SL'$

$L' \rightarrow ,SL' / \epsilon$

∴ The resultant Grammar is

$S \rightarrow (L) / a$

$L \rightarrow SL'$

$L' \rightarrow ,SL' / \epsilon$

Step-2: Elimination of left factoring.

The grammar is $S \rightarrow (L) / a$

$L \rightarrow SL'$

$L' \rightarrow , SL' / \epsilon$

Here, There is no left factoring

Step-3: calculating the first & Follow

The grammar is: $S \rightarrow (L) / a$

$L \rightarrow SL'$

$L' \rightarrow , SL' / \epsilon$

First (S) = { (, a } Follow (S) = { $, ,) }

First (L) = { (, a } Follow (L) = {) }

First (L') = { , ϵ } Follow (L') = {) }

Step-4: constructing the parse tree

1) a) $S \rightarrow (L)$

first (L) = { (}

Add $S \rightarrow (L)$ to M[S, (]

1) b) $S \rightarrow a$

First (a) = {a}

Add $S \rightarrow a$ to M[S, a]

2) $L \rightarrow SL'$

First (SL') = { (, a }

Add $L \rightarrow SL'$ to

M[L, (]

M[L, a]

3)a) $L' \to ,SL'$

First(,SL') = {,}

Add $L' \to ,SL'$ to M[L',]

3)b) $L' \to \epsilon$

Follow(L') = {)}

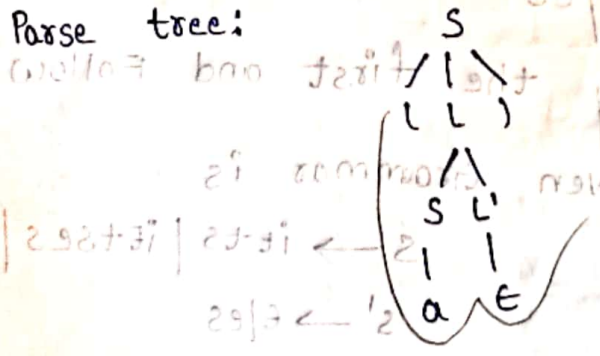
Add $L' \to \epsilon$ to M[L',)]

	(	)	a	,	$
S	$S \to (L)$		$S \to a$		
L	$L \to SL'$		$L \to SL'$		
L'		$L' \to \epsilon$		$L' \to ,SL'$	

Step-5: check whether the input string is accepted or not.

Stack	Input string	Action
$S	(a)$	$S \to (L)$
$)L(	(a)$	pop
$)L	a)$	$L \to SL'$
$)L'S	a)$	$S \to a$
$)L'a	a)$	pop
$)L'	)$	$L' \to \epsilon$
$)	)$	pop
$	$	accepted.

Parse tree:



Ex: construct predictive parser.

$S \rightarrow iEts \mid iEtses \mid a$

Step:1 Elimination of left recursion.

$A \rightarrow A\alpha \mid \beta$

replace with $A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

consider $S \rightarrow iEts$

$S \rightarrow iEtses$

$S \rightarrow a$

The Given string does not contain left recursion.

Step-2: Eliminating left factoring.

$S \rightarrow iEts$

$S \rightarrow iEtses$

$S \rightarrow a$

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \ldots\ldots \mid \gamma_1 \mid \gamma_2 \mid \gamma_3 \ldots$

$A \rightarrow \alpha A' \mid \gamma_1 \mid \gamma_2$

where $A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots$

where $A = S$

$\alpha = iEts$

$\beta_2 = es$

$\gamma = a$

$\beta_1 = \epsilon$

$S \rightarrow iEtss' \mid a$

$(S' \rightarrow iEts)_x$

$S' \rightarrow \epsilon \mid es.$

Step-3: calculating the first and Follow.

The Given Grammar is

$S \rightarrow iEts \mid iEtses \mid a$

$S' \rightarrow \epsilon \mid es$

First(S) = {i, a} Follow(S) = {$, e}

First(S') = {ε, e} Follow(S') = {$, e}

Step-4: constructing the parse tree

1) a) $S \to iEtSS'$

First(iEtSS') = {i}

Add $S \to iEtSS'$ to M[S, i]

1) b) $S \to a$

First(a) = {a}

Add $S \to a$ to M[S, a]

1) a) $S' \to \epsilon$

Follow(S') = {$, e}

Add $S' \to \epsilon$ to M[S', $]

M[S', e]

2) b) $S' \to eS$

First(eS) = {e}

Add $S' \to eS$ to M[S', e].

	i	a	e	$
S	$S \to iEtSS'$	$S \to a$		
S'			$S' \to \epsilon$ $S' \to eS$	$S' \to \epsilon$

When, two productions are placed in a single one, then it is not the (L,(1)) parser.

Recursive Descent parser:

- It contains the Recursive procedures we have to write x every non-terminal that is available Recursive procedure for each and in the Grammar.

To construct Recursive Descent parser we have to follow the following Rules:

→ The Input is non-terminal then call the corresponding procedure of the non-terminal.

2) → If the Input is terminal then compare the Input terminal with Input string. If they are same then we have to increment the Input pointer

3) → If a non-terminal produces three productions then we have to write those three productions in corresponding non-terminal.

4) → No need to define any main Function as well as No need to declare any variables. If we define the main function then we have to call start symbol function from the main function

Ex: $E \rightarrow iE'$

$E' \rightarrow +iE' \mid E$

Sol:

```
E()
{
  if (input == 'i')
    input++;
    EPRIME();
}
EPRIME()
{
  if(input == '+')
    input++;
    EPRIME();
}
else
  return;
}
```

Input string;

i+i$

Accepted.

Ex: $E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

```
E()
{
 T();
 EPRIME();
}
EPRIME()
{
  if(input == '+')
   {
    input++;
    T();
    EPRIME();
   }
   else
     return;
   }
   T()
    {
     F();
     TPRIME();
    }
   TPRIME()
    {
    if(input == '*')
    {
    input++;
    F();
    TPRIME();
    }
    else
     return;
    }
    F()
    {
    if(input == '(')
    {
     input++;
     E();
    if(input == ')')
     {
      input++;
     }
    else if(input == 'i')
     input++;
    }
```

input string: id+id$

Accepted.

Backtracking:

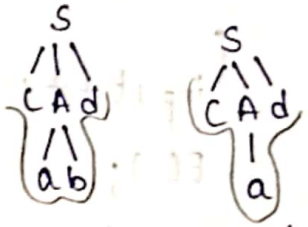
If a non terminal produces several productions then inorder to pass the given string we uses Backtracking.

In Backtracking we try to analyze all the possibilities & if any possibility is wrong then we choose next possibility.

Ex: S → CAd

A → ab/a

Input string: cad



cabd — not accepted

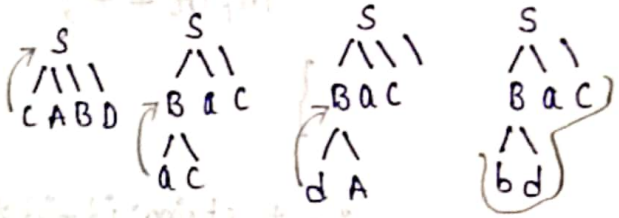
cad — Accepted

Ex: S → CABd / Bac

A → Bcd / aab

B → ac / bd

Input string : bdac.



bdac

Accepted

Bottom-up parsing:

→ Handle & Handle pruning:

Handle is a substring which matches with the right side of the production, if the (with) handle

matches with the right side of the production, then it is replaced with the corresponding left hand side non-terminal. This replacement is called reduction.

Ex: S → aABe
A → Abc / b
B → d

input

Reverse RMD:

S → aABe
→ aAde
→ aAbcde
→ abbcde

Right sentential form	Handle	Reducing production
abbcde	b	A → b
aAbcde	Abc	A → Abc
aAde	d	B → d
aABe	aABe	S → aABe
S		

Handle purning is obtained by producing rightmost derivation in Reverse order.

Ex: $E \rightarrow E+T / T$

$T \rightarrow T*F / F$

$F \rightarrow (E)\ id$

i/p string : id * id

Right Sentential-form	Handle	Reducing production
id * id	id	$F \rightarrow id$
F * id	F	$T \rightarrow F$
T * id	id	$F \rightarrow id$
T * F	T * F	$T \rightarrow T*F$
T	T	$E \rightarrow T$
E		

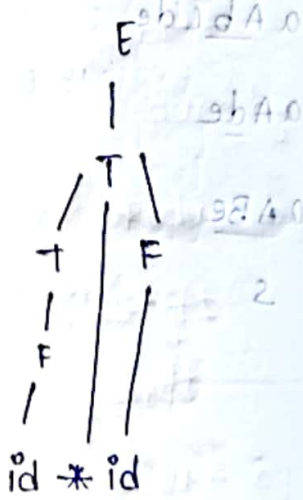
Reverse RMD: $E \rightarrow T$

$\rightarrow T*F$

$\rightarrow T*id$

$\rightarrow F*id$

$\rightarrow id*id$



Shift Reduce parser:

It is a Bottom up parsing technique in shift Reduce parser mainly we uses two datastructures.

1) stack

2) Input Buffer

Actions of shift Reduce parser:

→ shift: It is nothing but push operation it shifts the input symbol on the top of the stack.

• Initially bottom of the stack contains '$'.

Reduce:

Top of the stack matches with right of the Production then it is reduced to the corresponding left handside non-terminal.

Accept:

After parsing an entire input string. if the stack contains '$' and start symbol whereas input string contains '$' then we can say corresponding inputstring is Accepted by the Parser.

i.e; The corresponding inputstring Belongs to the language of the Grammar.

Error:

• If there are some syntax errors in the Grammar or with the inputstring not belongs to the language of the Grammar then the parser will generate error messages.

→ Those error messages are send to the user.

Shift Reduct conflict:

if the parser had a choice to select both shift action as well as Reduce action.

Reduce Reduce conflict:

It occurs more than one Reduction is Possible for corresponding Handle.

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Input string id * id

stack	input Buffer	Action
$	id * id $	shift
$ id	* id $	Reduce by F → id
$ F	* id $	Reduce by T → F
$ T	* id $	shift
$ T *	id $	shift
$ T * id	$	Reduce by F → id
$ T * F	$	Reduce by T → T*F
$ T	$	Reduce by E → T
$ E	$	Accept

Ex: $S \to (L) / a$

$L \to L,S / S$

Parse the input string (a, (a,a)) using shift Reduce parser.

Stack	input Buffer	Action
$	(a, (a,a))$	shift
$(	a, (a,a))$	shift
$(a	, (a,a))$	Reduce by $S \to a$
$(S	, (a,a))$	Reduce by $L \to S$
$(L	, (a,a))$	shift
$(L,	(a,a))$	shift
$(L,(	a,a))$	Reduce by $S \to a$
$(L,(a	,a))$	Reduce by $L \to S$
$(L,(S	,a))$	shift
$(L,(L	,a))$	shift
$(L,(L,	a))$	Reduce by $S \to a$
$(L,(L,a	))$	Reduce by $L \to L,S$
$(L,(L,S	))$	shift
$(L,(L	))$	Reduce by $S \to (L)$
$(L,L	))$	shift
$(L,L)	)$	Reduce by $L \to L,S$
$(L,S	)$	
$(L,S)	$	
$		

SLR parser : construction of SLR parser:

In order to construct the SLR parser first we have to write the agumented Grammer.

In Augumented Grammar we have to add the extra production called $s' \rightarrow s$.

where s, is the start symbol.

To construct SLR parse table we have to follow four steps.

→ calculating canonical collection of LR(0) items.
→ Designing of finite automata.
→ construction of SLR parse table.
→ parsing the i/p string.

Ex: consider the following Grammar

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

construct SLR parse table for the Grammar & show the Actions of one parser for the string 'abab'.

Step-1: Augumented Grammar:

$s' \rightarrow s$
$S \rightarrow AS$
$S \rightarrow b$
$A \rightarrow SA$
$A \rightarrow a$

step-1: $s' \rightarrow .s$
$S \rightarrow .AS$
$S \rightarrow .b$
$A \rightarrow .SA$
} To state (I_0, S I_0, A I_0, b I_0, a)

$A \to .a$ Io state Goto operations:

whenever we apply goto

goto(Io, S) then '.' will be shifted one position to the right.

I_1 : $S' \to S.$ (This item is known a final item that means dot should be present at right end of the Production).

$A \to S.A$ (If there is non terminal after dot Symbol we have to write productions of that non-terminal).

$A \to .SA$
$A \to .a$
$S \to .AS$
$S \to .b$ } I_1

I_2 goto (I_0, A)

$S \to A.S$
$S \to .AS$
$S \to .b$
$A \to .SA$
$A \to .a$

I_3 goto (I_0, b)

$S \to b.$

I_4 goto (I_0, a)

$A \to a.$

I_5 goto (I_1, A)

$A \to SA.$
$S \to .AS$
$S \to .b$

goto (I_1, S) I_6

$A \to S.A$
$A \to .SA$
$A \to .a$
$S \to .AS$
$S \to .b$

goto (I_1, a) I_4

$A \to a.$

goto (I_1, b) I_3

$S \to b.$

goto (I_2, S) I_7

$S \to AS.$

$(S \to)$

$A \to S.A$
$A \to .SA$
$A \to .a$
$S \to .AS$
$S \to .b$

goto (I_2, A)

$S \to A.S$ } I_2

goto (I_2, b)

$S \to b.$ } I_3

goto (I_2, a)

$A \to a.$ } I_4

goto (I_5, S)

$S \to AS.$
$A \to S.A$
$A \to .SA$
$A \to .a$
$A \to .a$ } I_7

goto (I_5, A)

$S \to A.S$
$S \to A.S$
$S \to A.S$ } I_2

goto (I_5, a)

$A \to a.$ } I_4

goto (I_5, b)

$S \to b.$ } I_3

goto (I_6, A)

$A \to \cdot SA$

$A \to \cdot a$

$S \to A \cdot S$
$S \to A \cdot S$
$S \to A \cdot S$
} I_2

$A \to \cdot SA$
$A \to \cdot SA$
$A \to \cdot SA$
} I_5

goto (I_6, S)

$A \to S \cdot A$
$A \to S \cdot A$
$A \to S \cdot A$
} I_6

goto (I_6, a)

I_4

goto (I_6, b)

I_7

goto (I_7, A)

$A \to SA \cdot$ } I_5

goto (I_7, S)

$A \to S \cdot A$ } I_6

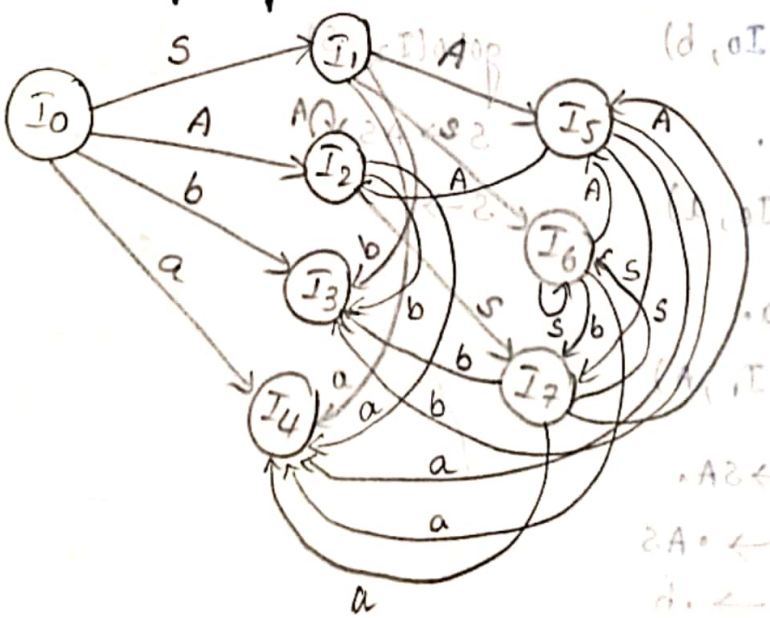
goto (I_7, a)

I_4

goto (I_7, b)

I_3

Step-2: Designning of Finite automata



Step-3: construction of SLR parse-table.

	Action			Goto	
	a	b	$	S	A
0	S_4	S_3		1	2
1	S_4	S_3	Accepted	6	5
2	S_4	S_3		7	2
3	r_2	r_2	r_2		
4	r_4	r_4			
5	S_4/r_3	S_4/r_3		7	2
6	S_4	S_3		6	5
7	S_4/r_1	S_3/r_1	r_1	6	5

I_0: $0 \to S' \to \cdot S$

$1 \to S \to \cdot AS$

$2 \to S \to \cdot b$

$3 \to A \to \cdot SA$

$4 \to A \to \cdot a$

$I_1 : S' \to S.$

$I_3 : S \to b$ (r_2)

Follow(S) = {$, a, b}

$I_4 : A \to a.$ (r_4)

Follow(A) = {a, b}

$I_5 : A \to SA.$ (r_3)

Follow(A) = {a, b}

$I_7 : S \to AS.$ (r_1)

Follow(S) = {$, a, b}

Step-4 parsing table aba b.

stack	i/p buffer	Action
$0	aba b$	shift 4
$0a4	bab$	reduce A→.a
$0A2	bab$	shift 3
$0A2b3	ab$	reduce S→.b
$0A2S7	ab$	reduce S→.AS
$0S1	ab$	shift 4
$0S1a4	b$	reduce A→.a
$0S1A5	b$	reduce A→.SA
$0A2	b$	shift 3
$0A2b3	$	reduce S→.b
$0A2S7	$	reduce S→.AS
$0S1	$	Accepted.

Ex: construct canonical LR(0) items for the Grammar

S → L = R | R

L → * R | id

R → L

Also construct SLR parse tree table & parse the input string * id = id.

Augmented Grammar:

$S' \to S$
$S \to L=R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$

Step-1:

I_0 $S' \to \cdot S$
$S \to \cdot L=R$
$S \to \cdot R$ 2
$L \to \cdot *R$ 3
$L \to \cdot id$ 4
$R \to \cdot L$ 5

goto (I_0, S)
$S' \to S. \} I_1$

goto (I_0, L)
$S \to L. = R \} I_2$
$R \to L.$

goto (I_0, R)
$S \to R. \} I_3$

goto $(I_0, *)$
$L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot id$
$\} I_4$

goto (I_0, id)
$L \to id. \} I_5$

goto $(I_2, =)$
$S \to L = \cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot id$

goto (I_4, R)
$L \to * R. \} I_7$

goto (I_4, L)
$R \to L. \} I_8$

goto $(I_4, *)$
$L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot id$
$\} I_4$

go(I_4, id)
$L \to id. \} I_5$

goto(I_6, R)
$S \to L=R. \} I_9$

goto(I_6, L)
$R \to L. \} I_8$

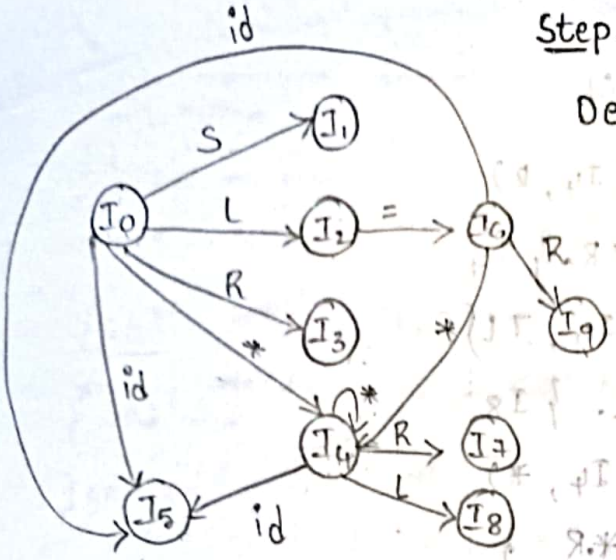
goto $(I_6, *)$
$L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot id$
$\} I_4$

goto (I_6, id)
$L \to id. \} I_5$

Follow $(S) = \{\$\}$
Follow$(L) = \{=, \$\}$
Follow$(R) = \{\$, =\}$

$S \to L=R \mid R$
$L \to *R \mid id$
$R \to L$

First $(S) = \{*, id\}$
First $(L) = \{*, id\}$
First $(R) = \{*, id\}$



Step-2: Designing a Finite Automata.

Step-3: construction of SLR parse table.

	Actions				Goto		
	*	id	=	$	S	L	R
0	s_4	s_5			1	2	3
1				Accepted			
2			s_6/r_5	r_5			
3				r_2			
4	s_4	s_5			8	7	
5			r_4	r_4			
6	s_4	s_5			8	9	
7			r_3	r_3			
8			r_5	r_5			
9				r_1			

Final states:

I_1: $S' \to S.$

I_2: $R \to L.$ (r_5)

follow(R) = { $, = }

I_3: $S \to R.$ (r_2)

F(S) = { $ }

I_5: $L \to id.$ (r_4)

F(L) = { =, $ }

I_7: $L \to *R.$ (r_3)

Follow(L) = { =, $ }

I_8: $R \to L\bullet$ $\quad r_5$

Follow(R) = { $, = }

I_9: $S \to L = R\bullet$ $\quad r_1$

Follow(S) = { $ }

Step-4: Parsing the i/p string:

Stack	I/p buffer	Action.
$0	*id = id$	shift 4
$0*4	id = id$	shift 5
$0*4id5	= id$	reduce L → id⊙
$0*4L8	= id$	reduce R → L
$0*4R7	= id$	reduce L → *R
$0L2	= id$	shift 6
$0L2=6	id$	shift 5
$0L2=6id5	$	reduce L → id
$0L2=6L8	$	reduce R → L
$0L2=6R9	$	reduce S → L = R
$0S1	$	Accepted.