SIRIRAJ

Unit - II [Syntax Analyzer / Parser]

Page No.:
Date:
YOUVA

* **Symbols:-** Basic building blocks of any language which can be any character or tokens
  Example:- A, B, C, ---Z, a, b, c, ---z, @, $, &, !, [ ], +, !, -, *
* **Alphabets:-** An alphabet is a finite collection of symbols
  $\Sigma = \{a, b, c, ---z, 0, 1, 2, 3, ---9\}$.
* **Strings:-** String is a finite collection of symbols from an alphabet set. $\Sigma = \{a, b\}$
  $\omega = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, ----\}$
* **Length of string ($|\omega|$) ⇒** Number of symbols involved in the string ω is called length of string.
  $\omega = ab$ $|\omega| = 2$ $\omega = aaa$ $|\omega| = 3$
  $\omega = \epsilon$ $|\omega| = 0$ (null string)
* **Language:-** collection of strings is called Language.
  $\Sigma = \{a, b\}$ $\omega_1 = \epsilon$, $\omega_2 = a$, $\omega_3 = b$, $\omega_4 = aa$, ---
  $L = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, baa, bbb, ----\}$
* **Formal Language:-** collection of strings we put some conditions & restrictions in the formation of string is called formal language.
  $\Sigma = \{a, b\}$
  length = 1, $\Sigma^1 = L_1 = \{a, b\}$
  length = 0, $\Sigma^0 = L_2 = \{\epsilon\}$
  length = 2, $\Sigma^2 = L_3 = \{aa, ab, ba, bb\}$
  length = 3, $\Sigma^3 = L_4 = \{aaa, aab, aba, baa, abb, bab, bba, bbb\}$

  These all languages are formal language with length condition

* **Grammar ⇒** Grammar is a set of rules (formal rules) which checks the ~~correctins~~ correctness of the ~~sence~~ sentances (OR) we can construct the correct sentance using these rules.

(OR)

Language are associated with rules which helps to make a valid string (sentance) of input symbols. Grammer is a collection of 4-tuples :-

$G = (V_n, \Sigma, P, S)$

Where

$V_n \Rightarrow$ It is a finite non empty set whose elements are called variable or non variable.

$\Sigma \Rightarrow$ It is a finite non empty set whose elements are called terminals.

$S \Rightarrow$ It is a starting variable ($S \in V_n$) or special variables.

$P \Rightarrow$ It is finite set of production whose elements are $\alpha \rightarrow \beta$ where $\alpha, \beta \in (V_n \cup \Sigma)$

* Types of Grammar ⇒ [Chomosky classification]

There are mainly four types of Grammer

- Type-0 grammar (REG) (unrestricted grammar / ~~Regular~~ unrestricted lang. Recursive enumerable grammar.
- Type-1 grammar (CSG) (context sensitive grammar / context sensetive lang
- Type-2 grammar (CFG) (Context free Grammar / context free language)
- Type-3 grammar (RLG) (Regular grammar / Regular language)

* Type-0 Grammar ⇒ $\phi A \psi \rightarrow \phi \alpha \psi$

(At least one variable on LHS of production)

(i) $ab\underline{A}bcd \rightarrow ab\underline{AB}bcd$ compair with $\phi A \psi \rightarrow \phi \alpha \psi$

$\phi = ab$, $\psi = bcd$, $A = A$ & $\alpha = AB$

↑ left context, ↑ Right context

(ii) $AC \rightarrow A$

$\phi = A$, $\psi = \wedge$ [~~left~~ Right context are $\wedge / \epsilon / d$ (null)

$A = C$ $\alpha = \wedge$

(iii) $C \rightarrow \wedge$

$\phi = \psi = \wedge$ (left & Right context are null $\wedge$)

$A = C$ $\alpha = \wedge$

A Production without any restriction is called type-0 grammar (or) unrestricted grammar (or) Recursive enumerable grammar.

(CSG)
(Context sensitive grammar)

Page No.:
Date:
YOUVA

→ Type-1 Grammar ⇒ $\phi A \psi \rightarrow \phi \alpha \psi$

where (i) $\alpha \neq \wedge$

(ii) $S \rightarrow \wedge$ is allow in type-1 grammar but S is not appear in RHS of any production.

$S \rightarrow Aa$ But $A \rightarrow \wedge$ is not allowed because Variable A is appear in RHS of S production.

$S \rightarrow Aa$ & $A \rightarrow a$ is allowed.

→ Type-2 Grammar ⇒ $\phi A \psi \rightarrow \phi \alpha \psi$ [context free Grammar] (CFG)

where (i) ϕ & ψ are null ($\wedge$/e/d) on left side of production means left side is only single Variable on left side of production

(ii) $S \rightarrow Aa$ & $A \rightarrow \wedge$ are allowed in type-2 grammar but not in type-1 Grammar.

→ Type-3 Grammar (Regular grammar) ⇒

where ~~(i) start with only single variable (Production start)~~

(i) $A \rightarrow aB$ (Right side of any production start with terminal & After that multiple terminals are not ~~variables~~ allowed on RHS of the production

Example $A \rightarrow abc$ X)

(ii) $S \rightarrow \wedge$ is allowed but S is not appear in RHS of any production.

Q.1) check the production type?

(i)

$S \rightarrow Aa$	type-2
$A \rightarrow Ba$	type-2
$B \rightarrow abc$	type-2
$A \rightarrow c$	type-3

(ii)

$S \rightarrow ASB$	type-2
$S \rightarrow d$	type-3
$A \rightarrow aA$	type-3

(iii) $S \rightarrow aS$ type-3 & $S \rightarrow ab$ type-2

(iv) $A \rightarrow abA$ is type-2.

* **Context free Grammar (CFG)** ⇒ As the name indicates context free means the grammar have no context (right & left context) is called context free grammar. It is type-2 Grammar. This grammar is accepted by PDA (Push-down Automata) context free languages are applied in parse design. It is also useful for describing block structure in programming languages.

A CFG is defined by 4-tuples.

$G = (V_n, \Sigma, P, S)$ where

V_n is the set of Variables

Σ is the set of input symbols

S is a special Variable called start symbol.

P is a production rules

∴ G is a CFL (context free Language) if every production is the form of $A \to \alpha$.

where $A \in V_n$ & $\alpha \in (V_n \cup \Sigma)^*$

It means that this is not contain left & Right context.

Q.1) Construct CFG that Generates the Language

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

Sol") $S \to AcA / BcB$ & $A \to aB / Ba / \wedge$ — null symbol

$B \to bA / Ab / \wedge$ ← null symbol

Q.2) Given ~~$G = \{ \{S\}$~~. $G = (\{S\}, \{a,b\}, P, S)$

where $P = \{ S \to aSa, \; S \to bSb \text{ & } S \to \wedge \}$

~~$L = \{ aa, bb, abba, baab, aaaabbbb, \ldots \}$~~ $L = \{aa, bb, abba, baab, \ldots$

$S \to aSa \quad S \to \wedge \quad \therefore S \to aa$

$S \to bSb \quad S \to \wedge \quad \therefore S \to bb$

$S \to aSa \quad S \to bSb \text{ & } S \to \wedge \quad \therefore S \to abSba = abba$

$S \to bSb \quad S \to aSa \text{ & } S \to \wedge \quad \therefore S \to baSab = baab$

* **Derivation tree/Parse tree** ⇒ Parse tree is also called a derivation tree.

when deriving a string w from S. if every derivation is considered to be a step in the tree construction then we get the graphical representation of derivation of string w as a tree. This is called derivation tree or Parse tree of strings

Rules:-

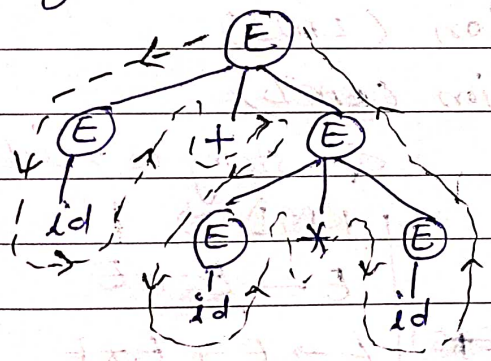
(i) All the leaf nodes of the tree are labeled by terminal of the grammar.

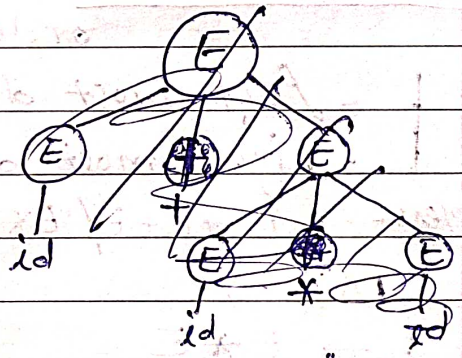
(ii) The root node of the tree is labeled by the start symbol of the grammar

(iii) The interior nodes are labeled by the variables.

Example:-(i) $E \rightarrow E+E / E*E / id$

String $w = id + id * id$

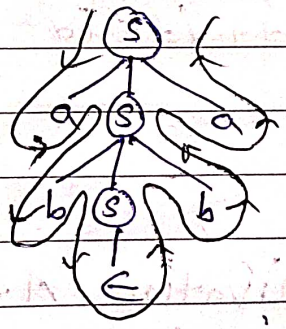




$w = id + id * id$

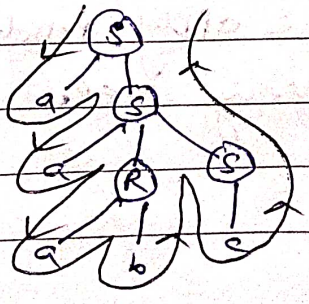
(ii) $S \rightarrow aSa / bSb / \epsilon$

$w = abba$ ⟶

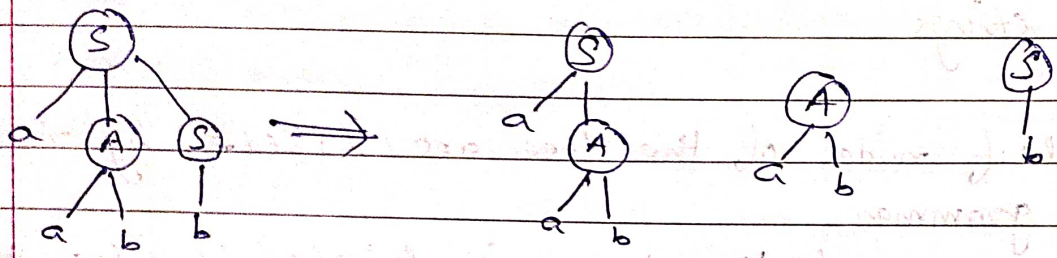


* **Yield** ⇒ (पैदा करना) The Yield of a Parse tree is the concatenation of the labels of the leaves without ~~repition~~ repitition (पुनरावृत्ति) in the left or right ordering.

Yield will be = aaabc



* **Subtree of a Parse tree** ⇒ A subtree of a Parse tree T is a tree whoose root is same vertex V of T & whose vertices are the descendants of V together with their labels & whose edges are those connecting the descendants of V.



* **Derivation** ⇒ There are mainly two types of order of derivations:-
  - Left most derivation (LMD)
  - Right most derivation (RMD)

Examples:- $E \to E+E / E*E / id$ & string $w = id + id * id$.

LMD ⇒

$E \to E*E$

$E \to E+E*E$ ($\because E \to E*E$)

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

RMD ⇒

$E \to E+E$

$E \to E+E*E$ ($\because E \to E*E$)

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

or

$E \to E+E*id$ ⟨ $E \to E+id*id$, $E \to id+id*id$

definition

LMD

A derivation $A \xrightarrow{*} w$ is called a LMD if we apply a production only to the left most variable at every step

RMD

A derivation $A \xrightarrow{*} w$ is a right most derivation if we apply production to the right most variable at every step.

Q.1 Let G be the grammar $S \to aB/bA$, $A \to a/aS/bAA$ & $B \to b/bS/aBB$ for string "aaabbabbba" then find LMD ~~& RMD~~ & also construct parse tree

Sol^n =

$S \to aB$

$S \to aaBB$

$S \to aaaBBB$

$S \to aaabBB$

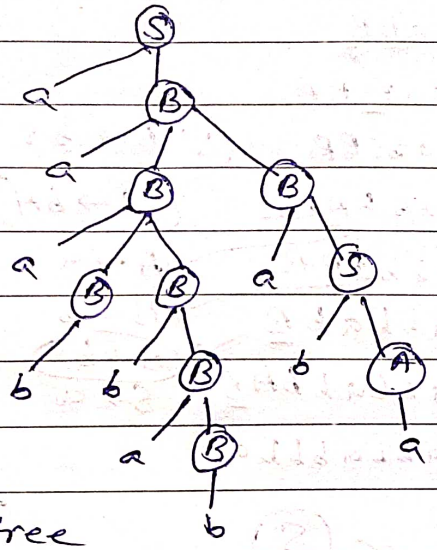
$S \to aaabbaBB$

$S \to aaabbabBB$

$S \to aaabbabbBS$

$S \to aaabbabbbA$

$S \to aaabbabbba$



Parse tree

Q.2 Let G be the grammar $S \to 0B/1A$, $A \to 0/0S/1AA$ & $B \to 1/1S/0BB$ for the string "00110101" then find LMD

Sol^n similarly Q.1

Q.3 The Given grammar G is $S \to aA/bB$, $A \to a/bA/aA$ & $B \to a/bS$ for string "aaabbba" then find LMD & Parse tree

Sol^n =

$S \to aA$

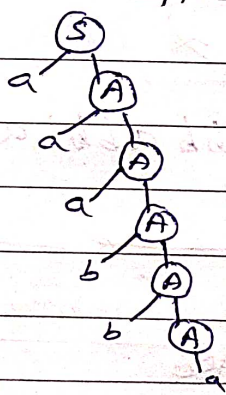
$S \to aaA$

$S \to aaaA$

$S \to aaabA$

$S \to aaabbA$

$S \to aaabbba$



RMD

Q.4 Let G be the given grammar $S \to 0B/1A$, $A \to 0/0S/1AA$ & $B \to 1/1S/0BB$ for string "00110101" find RMD & Parse tree

Sol^n =

$S \to 0B$

$S \to 00BB$

$S \to 00B1S$

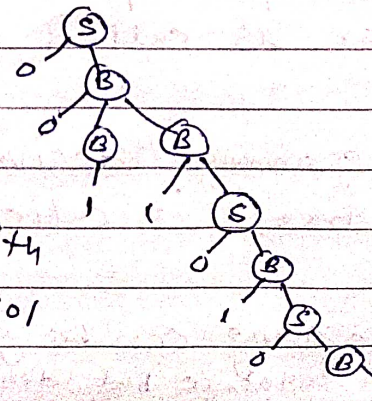
$S \to 00B10B$

$S \to 00B101S$

$S \to 00B1010B$

$S \to 00B10101$

$S \to 00110101$



Parse tree with yield = 00110101

Q.5) The Given grammar is $S \to aB/bA$, $A \to a/aS/bAA$ & $B \to b/bS/aBB$ Find LMD & RMD for the string "aaabbabbba"

LMD

$S \to aB$

$S \to aaBB$

$S \to aaaBBB$

$S \to aaabSBB$ ($\because B \to bS$)

$S \to aaabbABB$ ($S \to bA$)

$S \to aaabbaBB$ ($A \to a$)

$S \to aaabbabB$ ($B \to b$)

$S \to aaabbabbBS$ ($B \to bS$)

$S \to aaabbabbbS$ ($S \to a$)

$S \to aaabbabbba$

RMD

$S \to aB$ ($B \to aBB$)

$S \to aaBB$ ($B \to bS$)

$S \to aaBbbA$ ($S \to bA$)

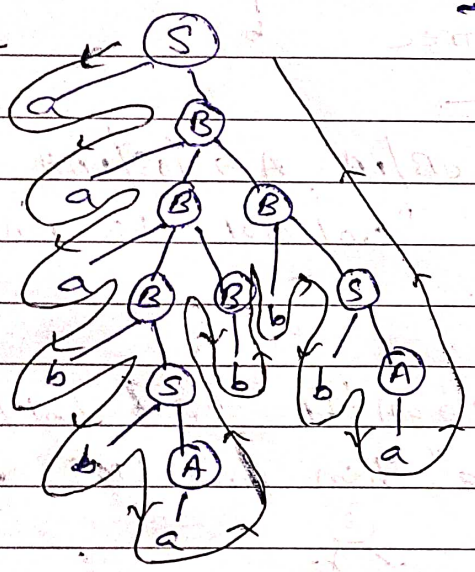
$S \to aaBbba$ ($A \to a$)

$S \to aaaBbbba$ ($B \to aBB$, $B \to b$)

$S \to aaabbAbbba$ ($B \to bS$ & $S \to bA$)

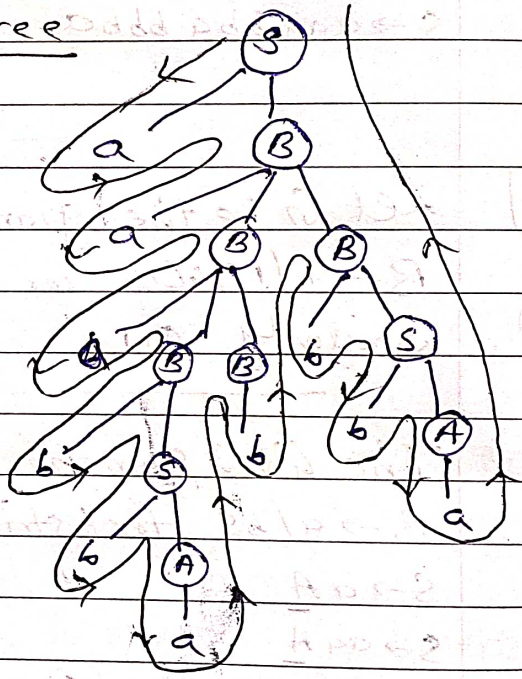
$S \to aaabbabbba$ ($A \to a$)

Parse tree



String $w = aaabbabbba$

Parse tree



String $w = aaabbabbba$

Ambiguous Grammar ≠ Ambiguity in CFG ⇒

Any grammar construct more than one parse tree for a single string (grammar) then these grammar is called ambiguous grammar.

OR

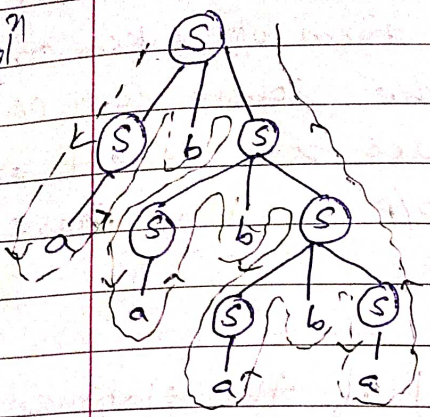
If any grammar derived more than one derivation tree (Parse tree) is called ambiguous grammar

OR

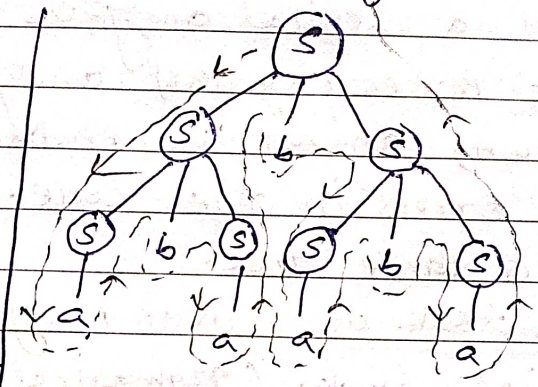
If same string can be generate in two different ways (LMD/RMD) derivation sequence but derivation will be different (LMD/RMD)

Q.1) If the productions are $S \to SbS/a$ & string $w = abababa$ then show that G (Grammar) is ambiguous.

Sol^n



$w = abababa$



$w = abababa$

So given grammar is ambiguous grammar.

Q.2) Check whether the following grammar is ambiguous $S \to iCts / iCtSes / b / a$ & string $w = ibtibtaea$

LMD ⇒

$S \to iCtSeS$

$S \to ibtiCtses$

$S \to ibtibtaea$

RMD

$S \to iCts$

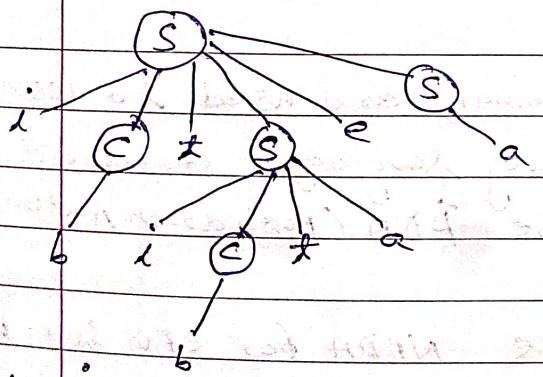
$S \to ibts$

$S \to ibtiCtSes$

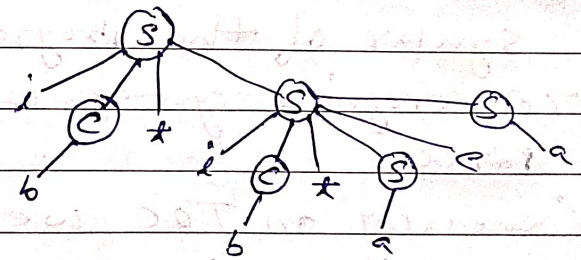
$S \to ibtibtses$

$S \to ibtibtaes$

$S \to ibtibtaea$



$w = ibtibtaea$



$w = ibtibtaea$

So given grammar is ambiguous grammar

Q.3) $S \to SAS/b$, $A \to ba$ string $w = bbabbbbab$

Q.4) $S \to aAs/a$, $A \to SbA/SS/ba$ string $w = aababaabbaaa$

Q.5)

* Syntax analyzer (Parser) ⇒ A syntax analyzer or Parser takes the input from a lexical analyzer in the form of token stream. The Parser (syntax) analyzes the source code (token stream) against the production rules to detect any ~~ero~~ errors in the code. The o/p of this Phase is a Parse tree.

main task of syntax Analyzer :-

→ Varifies Syntax (gramatical checking) of the statement.
→ It constructs the parse tree.
→ It detects syntax errors.
→ Error recovery.

Syntax Errors :-

→ Less number of tokens required for the statement
Example :- a = b+c
→ More number of tokens
Example :- a = (b+c) * (d * e))
→ Tokens are not arranged properly.
Example :- ; int a, b

Important point :-

~~→~~ CFG (Context free grammar) are used to describe syntax of the programming language statement.
→ To recognize syntax we use PDA (Push down Automata) Parser
→ Generally on TOC, we make NPDA for CFG but here we make DPDA which are very difficult for this we restrict in CFG.

Example ①

int a, b;
S → TL;
T → int | float
L → L, id | id

Example ②

a = b + c * d;
S → id = E;
E → E+E | E*E | id

GIRIRAJ

Page No.:
Date:
YOUVA

Example (5) I am talking to him

S → subject → verb → object

S → I / we / you

verb → talking

object → he / she / her / him

✗ Parser ⇒ A syntax analyzer or Parser is nothing a program that takes string w & CFG ~~Gfor~~ grammar G and varifies whether the string derivable from the grammar or not.

If it is derivable we construct derivation tree for that string otherwise, it returns Syntax Error to the error handler.

(OR)

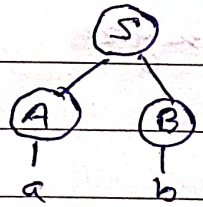
A syntax analyzer or Parser is a program that performs syntax analysis.

A Parser obtains a string of tokens from the lexical analyzer & verifies whether or not the string is valid construct of the source language.

Example:- string w = ab

Grammar:- S → AB

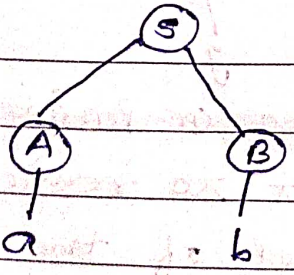
A → a & B → b



↓ TOP-down Parsing

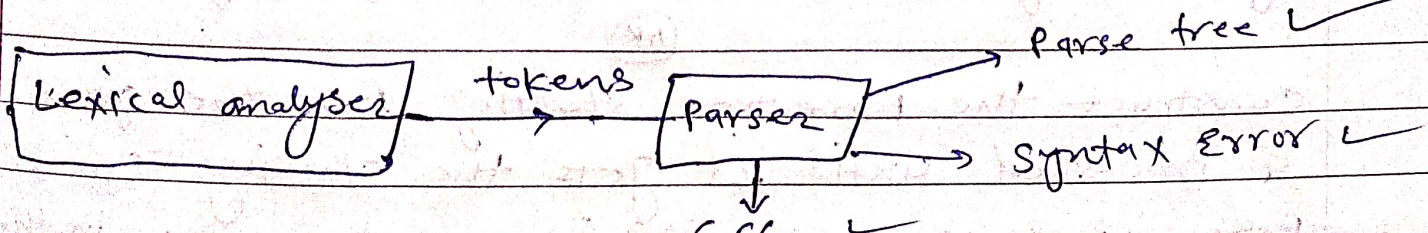
∴ string ab is a member of given grammar

Parse tree



write string Ist & you see the terminal & it is in CFG replace it by it's LHS Part it in RHD

↑ Bottom-up Parsing



Rujdambh [खलदंभ ~~ह~~ दमन को नाश करना / दमन को नाश करने वाला]

Page No.:
Date:
YOUVA

* **Parsing** ⇒ Parsing is the technique or process used to determine whether string of tokens can be generated by grammar.

Classification of Parsing (Parser) :-

- TOP-Down Parsing
- Bottom-UP Parsing

→ **TOP-Down Parsing** ⇒ TOP-down Parsing attempts to find the left most derivations for generating Parse tree from root to leaf nodes.

(OR)

In top-down Parsing, we will construct the Parse tree starting from the starting ~~symbol~~ symbol the required string. At every steps will replace LHS ~~by~~ non-terminal by it's corresponding RHS part.

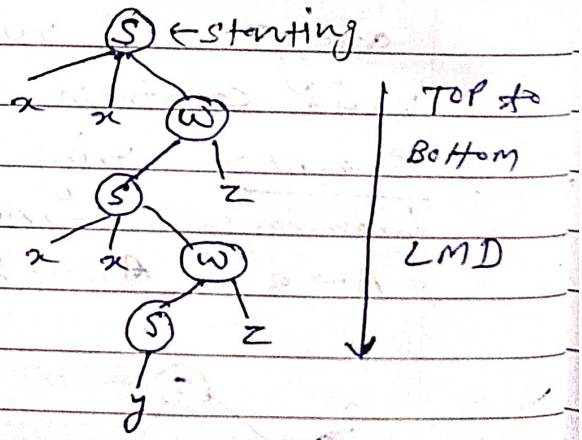
TOP-Down Parser uses LMD to construct the Parse tree.

Example:- $S \rightarrow xxW$

$S \rightarrow y$

$W \rightarrow Sz$

String $w = xxxyzz$



→ **Bottom-UP Parsing** ⇒ Bottom-UP Parser or Parsing can be defined as attempt to reduce the i/p string (w) to the starting symbol of the grammar by tracking of the RHS (RMD) of string (w) of reverse!

(OR)

construct the Parse tree starting from the given string & processed until it gets the starting symbol.

Bottom-UP Parser uses RMD in reverse order in the construction of the Parse tree.

Bottom-UP parser design from leaf node to root node.

RGG ⇒ Rakesh Giri Goswami

S

xxW

xxSz

xx(xxW)z

xxx(S)z

starting → xxxxyzz

Grammar

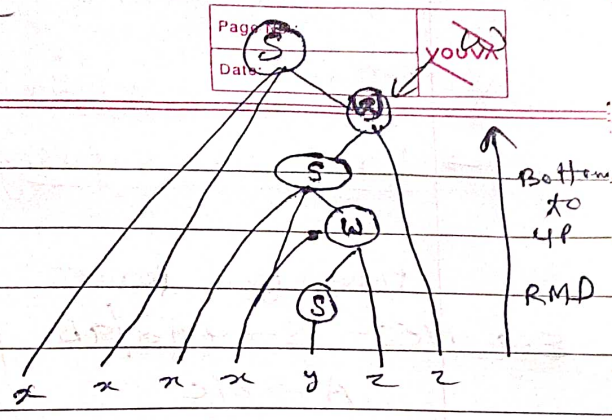
S → xxW

S → y

W = Sz

String

xxxxyzz

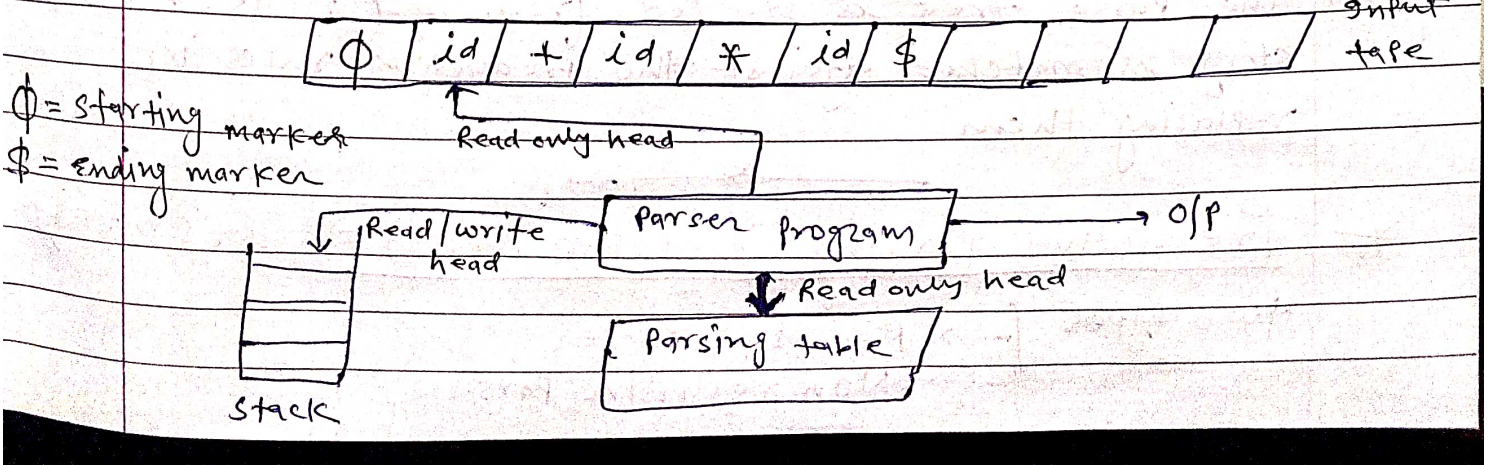


* Parsing techniques (Parsing techniques classification)

Parsing techniques

- Top-Down Parsing
  - (Back-tracking) With back tracking (Brute force Parsing)
  - (Non-back tracking) without Back-tracking (Predictive) Parsing
    - Recursive descent Predictive Parsing (RDPP)
    - Non recursive descent predictive Parsing (NRDPP)
    - Applicable for (on) LL grammar
- Bottom-up Parsing
  - Shift-reduce Parsing
    - Operator Precedance
      - Applicable on operator grammar
    - LR Parser
      - SLR
      - CLR
      - LALR
      - Applicable on LR grammar

* Basic Architecture of Parser ⇒ All the Parser (Top-Down & bottom-up) are used same architecture



$ = starting marker

$ = Ending marker

✳ Top-Down Parsing techniques ⇒

→ **Brute force Parsing ⇒** It is based on back-tracking concept. Brute force Parsing is a Powerful Parser compare to all other Parser.

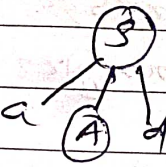
Example:- $S \to aAd/aB$

$A \to b/c$

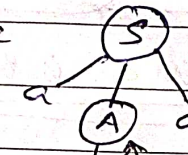
$B \to ccd/ddc$

$w = addc$

$S \to aAd.$



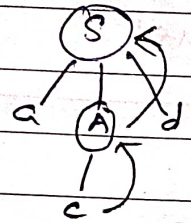
$A \to b/c$



$w = abd \notin w$

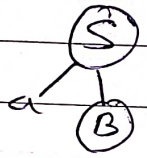
back track

Again

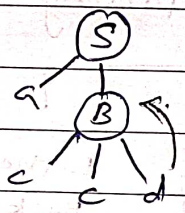


$w = acd \notin w$

Again $S \to aB$

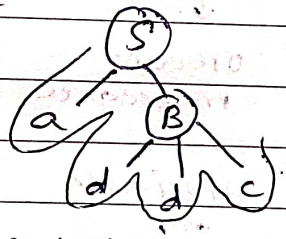


$B \to ccd.$



$w = accd \notin w$

$B \to ddc$



$w = addc$ match with w (string)

yield = addc matched with string.

→ The parsing technique working good for the number of grammers is known as power of that parser.

→ Brute force Parsing is a powerful Parser compare to all other Parser.

time complexity $O(n^3)$ it takes Exponential time

→ Back tracking require exponential time complexity, hence no practical compiler is design with back tracking.

→ The Parse tree is started from root node & if string is matched against the production rules in replacing them

✳ Predictive Parsing [predictive Parser] ⇒ It's mainly two types

- → Recursive Parser
- → Non recursive Parser

Predictive Parser ⇒ whenever a non-terminal having multiple production on the RHS part, we can predict using production to select by using Look ahead symbol. To construct Predictive Parser grammar should not contain left recursion & it should be left factoring (factored)

⇒ Non Recursive Descent Predictive Parser (NRDPR) ⇒ LL(1)

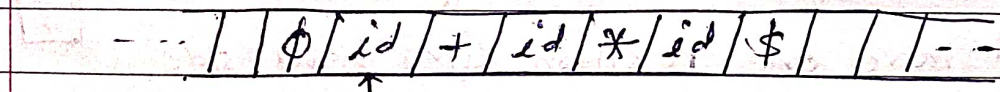
we discuss the Non-recursive Descent Predictive Parser in two ways:-

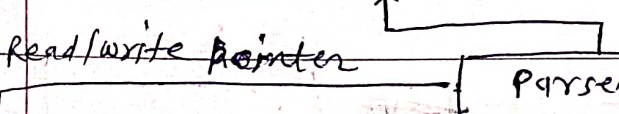
(I) Check the validity of string assume predictive table is given
(II) Grammar given & we have to design predictive table

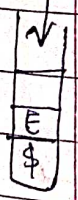
Q.1) Grammar :⇒ $E \to TE'$ $E' \to +TE'/\epsilon$
$T \to FT'$ $T' \to *FT'/\epsilon$ & $F \to (E)/id$
String ⇒ id + id * id

Sol^n Using LMD (left most derivation) ⇒

$E \to TE'$ put $T \to FT'$
$E \to FT'E'$ $F \to id$
$E \to id\ T'E'$ $T' \to \epsilon$
$E \to id\ E'$ $E' \to +TE'$
$E \to id + TE'$ $T' \to FT'$
$E \to id + FT'E'$ $F \to id$
$E \to id + id * FT'$ $T' \to *FT'$
$E \to id + id * idT'$ $F \to id$
$E \to id + id * id \epsilon$ $T' \to \epsilon$
$E \to id + id * id$







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

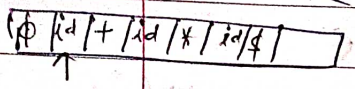
→ Two markers put in input tape :- starting marker (ϕ) & end marker ($)

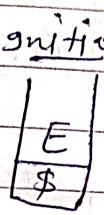
→ Stack contains initial two element $ (doller) & starting variable E

Shift operation means:- Push into stack corresponding entry from parsing table.

Example:- E Vs id ⇒ different value & corresponding entry will be E → TE' then push into stack first push E', second T. similarly all entry.

Reduce operation means:- Pop from stack & head move one possition Right hand side.

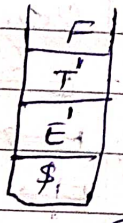




Step-I :- E Vs id ⇒ ~~E → id~~ E → TE'

Push E → TE' into stack

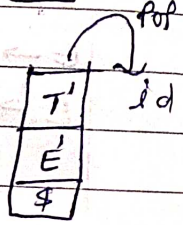
Step-II :- T Vs id ⇒ FT' push into stack

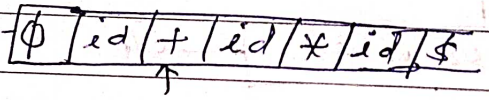


Step-III :- F Vs id ⇒ F → id

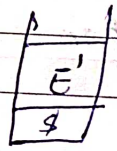
Push into stack

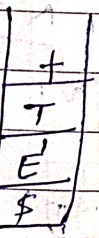
Step-IV :- id Vs id ⇒ Pop id from the stack & head move one possition Right hand side



Step-V : 

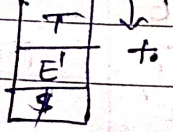
T' Vs + ⇒ T' → ∈ push



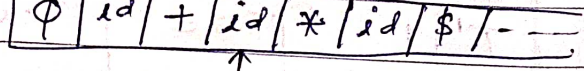


Step-VI :- E' Vs + ⇒ E' → +TE' push into stack

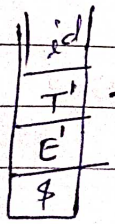
+ Vs + ⇒ Pop from stack top & head move Right side



Step-VII :- T Vs id

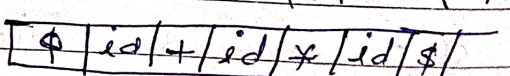


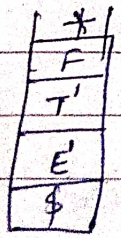
⇓ T → FT' push into stack



→ Step-VIII :- F Vs id ⇒ F → id Push into stack

Step-IX :- id Vs id ⇒ Pop from stack TOP & head move Right side





Step-X :- T' Vs * ⇒ T' → *FT'

Step-XI :- * Vs * ⇒ Pop stack TOP & head move Right side

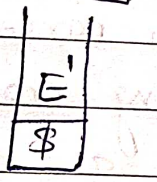
$	id	+	id	*	id	$

Step-XII :- F Vs id ⇒ F → id Push in to stack

Step-XIII :- id Vs id ⇒ Pop from stack TOP
Head move one possition Right side

Step-XIV :- | $ | id | + | id | * | id | $ |

T' Vs $ ⇒ T' → ∈ Push ∈



Step-XV :- E' Vs. $ ⇒ E' → ∈ Push ∈

Step-XVI :- $ Vs $ ⇒ String is valid

Note :- Invalid string ⇒ (i) If no corresponding entry found in table for stack TOP Vs I/P tape

(ii) If it contains different terminals like
+ Vs id or * Vs id or + Vs * or * Vs +

(iii) Both above case comes (existence) then string is invalid.

Consider a string (id) + id find the given string is valid or invalid. Grammars are:

E → TE' E' → +TE'/∈ T → FT'

T' → *FT'/∈ F → (E)/id

self करे Parsing table पिछले question में use करे।

⇒ Recursive Descent Predictive Parser (RDPP)

Working steps :-

1. Remove left Recursion from grammar if it is present in the grammar
2. Do left facutoring if needed
3. calculate the first & follow of given grammar
4. construct predictive parsing table.

⇒ Eliminating Left Recursion ⇒ If a grammar contains a pair of productions of the form $A \to A\alpha / \beta$ then the grammar is called left recursive grammar.

If we donot remove the left recursion then TOP-down Parser enter into an infinite loop during the Parsing Process.

There are following types of recursions

(1) Recursive (2) left recursion (3) Right recursion (4) Direct recursion (5) Indirect recursion

Rule:- $E \to E\alpha / \beta$ or $E \to E\alpha$ & $E \to \beta$

then $E \to \beta E'$ & $E' \to \alpha E' / \epsilon$

In general form:-

$$A \to A\alpha_1 / A\alpha_2 / A\alpha_3 / \dots A\alpha_m / \beta_1 / \beta_2 / \beta_3 / \dots / \beta_m$$

then

$$A \to \beta_1 A' / \beta_2 A' / \beta_3 A' / \dots / \beta_m A'$$

$$A' \to \alpha_1 A' / \alpha_2 A' / \alpha_3 A' / \dots / \alpha_n A' / \epsilon$$

Q.1) consider the following grammar

$A \to ABb / Aa / a / b$ $\quad B \to Bb / b$ remove left recursion

Soln

$A \to aA'$ & $A' \to BbA' / \epsilon$ $\quad \therefore A \to aA' / bA'$

$A \to bA'$ & $A' \to aA' / \epsilon$ $\quad A' \to BbA' / aA' / \epsilon$

Similarly $B \to Bb / b$

$B \to bB'$ & $B' \to bB' / \epsilon$

Q.2) $A \to ABd / Aa / a$ Soln $A \to aA'$ & $A' \to BdA' / aA' / \epsilon$

Q.3) $S \to Aa / b$ & $A \to Ac / Sd / \epsilon$

$A \to Ac / Sd / \epsilon$ Put $S \to Aa / b$ $\quad A \to Ac / bd / \epsilon$

$A \to Ac / Aad / \epsilon$ then $\quad A \to bdA'$

$A \to \epsilon A'$ or $A \to A'$ $\quad A' \to cA' / \epsilon$

$A' \to adA' / \epsilon$

then $A \to bdA' / A'$ & $A' \to cA' / adA' / \epsilon$

Remove left recursion

Date:

Q.4) $E \to E+T / T$, $T \to T*F / F$ & $F \to (E) / id$.

Sol^n $E \to E+T$, $E \to T$ $\Rightarrow$ $E \to TE'$ & $E' \to +TE' / \epsilon$

$T \to T*f$, $T \to F$ $\Rightarrow$ $T \to FT'$ & $T' \to *FT' / \epsilon$

& $f \to (E) / id$.

Q.5) $S \to (L) / a$ & $L \to L,S / S$

$L \to L,S$, $L \to S$ $\Rightarrow$ $L \to SL'$, $L' \to SL' / \epsilon$

$\Rightarrow$ Left factoring $\Rightarrow$ If $A \to \alpha / \alpha A$ generate "aaaa" we design the grammar that has no production starting symbol same called left factoring grammar Result of given grammar $A \to \alpha A'$ & $A' \to A / \epsilon$

(OR)

If you have more than one string on the grammar are started with the same symbol then left factoring. In parser we can not see forward otherwise we have to use look ahead

Q.1 Calculate the left factoring [Do the left factoring]

(i) $A \to aA / aB / a$ & $B \to b / bB$

Sol^n $A \to aA'$, $A' \to A / B / \epsilon$ | $B \to bB'$, $B' \to \epsilon / B$

(ii) $A \to aB / aG$, $B \to bB / b$, $G \to cG / \epsilon$

$A \to aA'$, $A' \to B / G$ | $B \to bB'$, $B' \to B / \epsilon$ | $G \to cG'$, $G' \to G / \epsilon$

(iii) $A \to aB / aA / aBA / a$ & $B \to bB / b$

$A \to aA'$, $A' \to B / A / BA / \epsilon$

(OR)

$A' \to B / BA$, $A' \to BA''$, $A'' \to A / \epsilon$ or ϵ / A

$B \to bB'$, $B' \to B / \epsilon$

(iv) $A \to aA / aAB / aABG$

$A \to aA'$, $A' \to A / AB / ABG$, $A' \to AA''$, $A'' \to \epsilon / B / BG$, $A'' \to B / BG$, $A'' \to BA''' / \epsilon$, $A''' \to \epsilon / G$

Calculate the first & follow ⇒

⇒ First :- Rules for first :- If $\alpha = XYZ$

Rule-1 :- if First(α) = First(XYZ) = {X} if X is terminal otherwise

First(α) = First(XYZ) = First(X) if X does not derive to an empty string (ϵ/^/d)

If X derives to an empty string (ϵ) (∵ $X \to \epsilon$) then

First(α) = First(XYZ) = First(X) − {ϵ} ∪ First(YZ)

P ⇒ Production Variable T → Terminal V → Variable

(i) $P \to TV$ then First(P) = {T}

(ii) $P \to VT$ & $V \to T'$ then First(P) = First(V) = {T'}

(iii) $P \to VT$ & $V \to T'/\epsilon$ then First(P) = First(V) − {ϵ} ∪ First(T)

= {T', ϵ} − {ϵ} ∪ {T} ⇒ {T, T'}

Q.1) Consider the following grammar & calculate the first

$S \to aABh$ $A \to c/\epsilon$ $B \to d/\epsilon$

Sol^n First(S) = {a} or First(S) = First(aABh) = {a}

First(A) = First(c) ∪ First(ϵ) = {c, ϵ}

First(B) = First(d) ∪ First(ϵ) = {d, ϵ}

Q.2) Calculate the first of the following grammar

$E \to TE'$ $E' \to +TE'/\epsilon$ $T \to FT'$ $T' \to *FT'/\epsilon$

$F \to (E)/id$

Sol^n First(E) = First(T) = First(F) = {c, id}

First(E') = {+, ϵ}

First(T') = {*, ϵ}

V	First
E	{c, id}
E'	{+, ϵ}
T	{c, id}
T'	{*, ϵ}
F	{c, id}

⇒ Follow ⇒

(i) follow of starting variable is $

(ii) $P \to VT$ or $P \to T'VT$

follow (V) = { T }

(iii) $P \to VV'$ or $P \to T'VV'$ & $V' \to T'$

follow (V) = first (V') = { T' }

(iv) $P \to VV'$ & $V' \to T'/\epsilon$

follow (V) = first (V') − { ∈ } ∪ follow (P)

(v) $P \to VV'$

follow (V') = follow (P)

Note:- We can not use ∈ in follow of any variable.

Q.1) Calculate the follow

$S \to aABb$ $A \to c/\epsilon$ $B \to d/\epsilon$

Sol^n

follow (S) = { $ }

follow (A) = first (B) − { ∈ } ∪ first (b)

= { d, ∈ } − { ∈ } ∪ { b } = { d, b } or = { b, d }

follow (B) = first (b) = { b }

Q.2) Calculate the follow for the following grammar

$E \to TE'$ $E' \to +TE'/\epsilon$ $T \to FT'$

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

Sol^n

follow (E) = { $ } ∪ first ( )) = {), $ }

follow (E') = follow (E) = {), $ }

follow (T) = first (E') − { ∈ } ∪ follow (E)

= { +, ∈ } − { ∈ } ∪ {), $ } ⇒ { +,), $ }

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

follow (F) = first (T') − { ∈ } ∪ follow (T')

= { *, ∈ } − { ∈ } ∪ { +,), $ } = { +, *,), $ }

Q.3) $S \to ACB/CbB/Ba$

$A \to da/BC$

$B \to g/\epsilon$ & $C \to h/\epsilon$ Calculate the First & follow

Predictive

Date:

⇒ Construct Parsing table →

$E \to TE'$ $E' \to +TE' / \epsilon$, $T \to FT'$

$T' \to *FT' / \epsilon$ ~~$F \to (E) / id$~~ $F \to (E) / id$

Variable	First	follow
E	{ (, id }	{), $ }
E'	{ +, ϵ }	{), $ }
T	{ (, id }	{ +,), $ }
T'	{ *, ϵ }	{ +,), $ }
F	{ (, id }	{ +, *,), $ }

V \ T	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	$F \to id$			$F \to (E)$		

Q.2) calculate First & follow & construct predictive parsing table

$S \to iEtSS_1 / a$ $S_1 \to eS / \epsilon$, $E \to b$

Variable	First	follow
S	{ a, i }	{ e, $ }
S_1	{ e, ϵ }	{ e, $ }
E	{ b }	{ t }

V \ T	a	b	e	i	t	$
S	$S \to a$			$S \to iEtSS_1$		
S_1			$S_1 \to eS$ $S_1 \to \epsilon$			$S_1 \to \epsilon$
E		$E \to b$				

time complexity $O(n)$