SPCC

Chapter 8 : Syntax Analysis (Parser)

Role of Parser :-

1] The parser obtains a string of tokens from the lexical analyzer, & verifies that the string can be generated by the grammar for the source language.

2] If parser unable to generate the string, then it display appropriate error message.

3] Parser generate parse tree.

Regular Expressions Vs. Context-Free Grammars :-

Regular Expression	Context-Free Grammar
1] A limitted amount of syntax analysis is done by a lexical analyzer using RE	1] CFG are capable of describing most, but not all of the syntax of programming languages.
2] Every construct that can be described by a regular expression can also be described by a grammar. i.e every regular set is a context-free language.	2] Not possible.

Why use regular expressions to define the lexical syntax of a language?

→ (1) The lexical rules of a language are frequently quite simple & to describe them we do not need a notation as powerful as grammars.

(2) RE generally provide a more concise & easier to understand notation for tokens than grammars.

(3) More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

(4) RE are most useful for describing the structure of lexical constructs such as identifiers, constants, keywords & so forth.

(5) Grammars are most useful in describing nested structures such as balanced parentheses, matching begin - end's, corresponding if-then-else's & so on.

Context - Free Grammar:-

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

V = set of non terminal.

T = set of terminal.

P = set of production rules.

S = start symbol.

production rules are of the form $X \rightarrow \alpha$.

where $X \in V$ & $\alpha \in (V \cup T)^*$.

ex $E \rightarrow E+E$

$/ E*E$

$/ id.$

consider string id + id * id.

Left Most derivation

$E \rightarrow E+E$

$\rightarrow id+E$

$\rightarrow id+E*E$

$\rightarrow id+id*E$

$\rightarrow id+id*id$

Rightmost Derivation.

$E \rightarrow E+E$

$\rightarrow E+E*E$

$\rightarrow E+E*id$

$\rightarrow E+id*id$

$\rightarrow id+id*id$

$E \rightarrow E*E$

$\rightarrow E+E*E$

$\rightarrow id+E*E$

$\rightarrow id+id*E$

$\rightarrow id+id*id$

$E \rightarrow E*E$

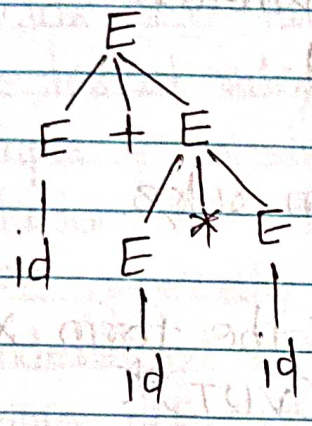
$\rightarrow E*id$

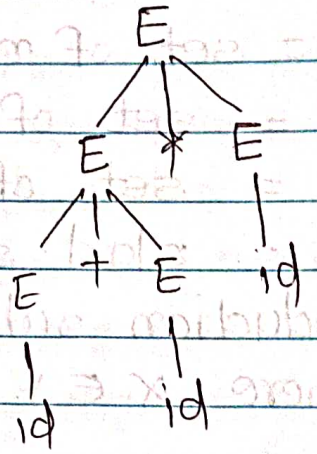
$\rightarrow E+E*id$

$\rightarrow E+id*id$

$\rightarrow id+id*id.$

Parse Tree



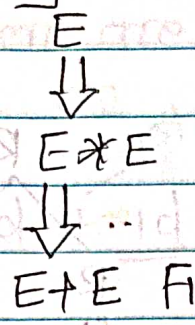


Ambiguous Grammar :- If For any string if we are getting more than one Leftmost derivation, or rightmost derivation OR parse tree then given grammar is ambiguous.

Parser does not allow ambiguous grammar because it get confused that which one parse tree follow.

ex 2+3*4. expected o/p is 14.
By considering above first parse tree.
In order to evaluate E we need to evaluate find E+E, In order to get answer we need to evaluate Right side E as E*E
∴ Ans is 14

by considering 2nd parse tree. to evaluate



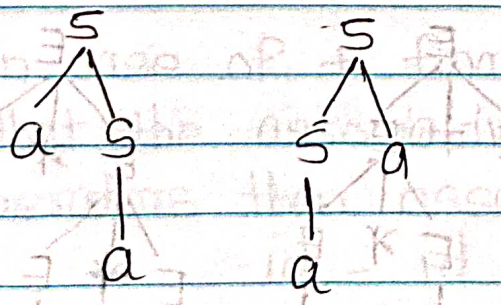
∴ Ans is = 20 Which is not correct.

Therefore, it is neccessary to remove ambiguity From grammar :-

Practically, there is no algorithm to remove ambiguity or to identify ambiguity except for error.

or. $S \rightarrow aS|Sa|a$

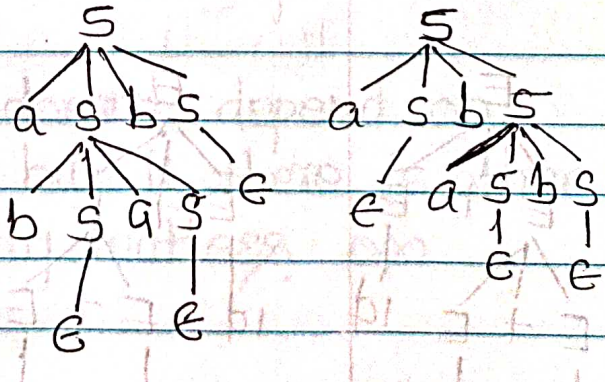
Consider string = aa



∴ Ambiguous.

$S \rightarrow aSbS|bSaS|\epsilon$

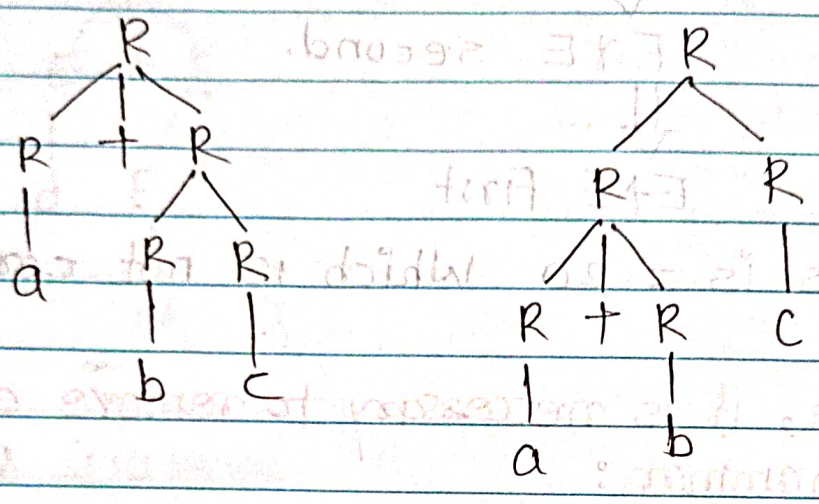
string = abab.



∴ Ambiguous

3] $R \rightarrow R+R \mid RR \mid R* \mid a \mid b \mid c$

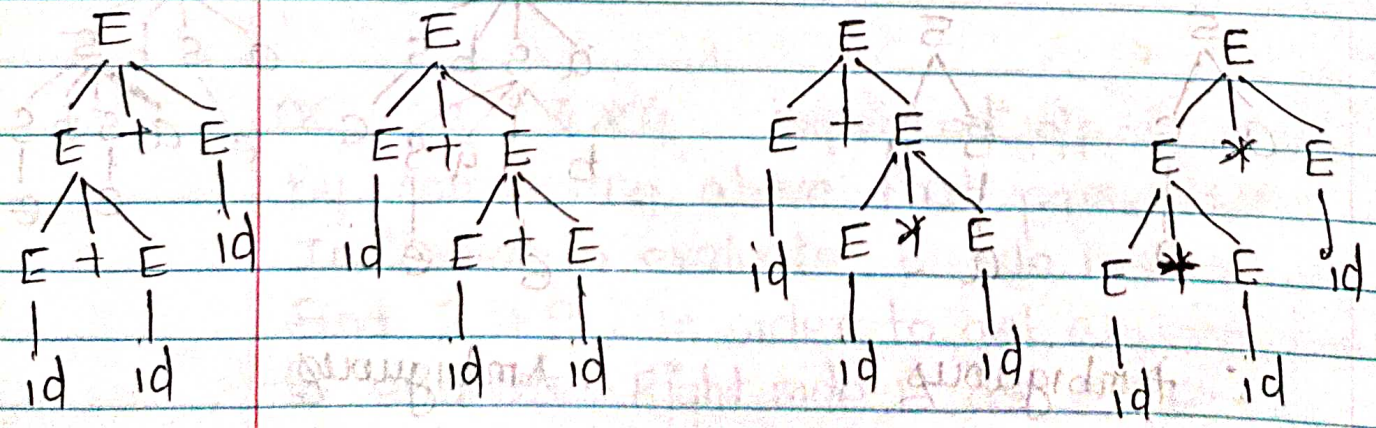
string = a+bc



Remove Ambiguity :-

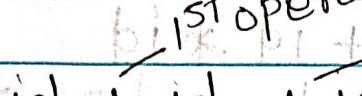
ex $E \rightarrow E+E \mid E*E \mid id$

string id+id+id string = id+id*id



id + id + id.

Whenever there are two operator on both side of operand. ex



then the problem is which operator should i associate with that operand (id). If i associate that operand with left side operator like

(id + id) + id

then it is called left associativity

If i associated that operand with right side operator like

id + (id + id)

then it is called right associativity

In case of + final o/p doesn't depend on what the associativity is but if there is other operators then associativity matters ex.

id - id - id

still + has left associativity, therefore leftmost + must be evaluated first.

∴ First parse tree is right.

Grammar fails because rules of associativity fails.

Consider, id + id * id
here operand id is associated with two operators i.e + and *, but * has highest precedance than +.
∴ * must be evaluated first.

∴ Grammar fails because precedance of operator is not taken care.

For any grammar, if grammar or parse tree grow in left side then it is going to be <u>left associative</u>.

If grammar or parse tree grow in right direction then it is going to be <u>right associative</u>.

<u>ex</u> id + id + id here we want left associativity so we restrict grammar to grow only in left direction.

+ is left associative.

∴ $E \rightarrow E + id \mid id$

Above grammar is defined as Left recursive. i.e left most symbol in RHS is equal to LHS.

When there are two different operators then highest precedance operator should be at the higest level (in parse tree)

∴ $E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id.$

Here $E \rightarrow \underline{E+T} \mid T$ indicate + is left associative.
& $T \rightarrow \underline{T*F} \mid F$ indicate * is left associative.
If there are many + then it will be at lowest level & once i reach at * then there is no way to go to +.

In above grammar if i want to add ↑

ex $2^{3^2} \Rightarrow 2\uparrow3\uparrow2$

↑ is right associative.

i.e $(2^{(3^2)})$

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow G\uparrow F \mid G$

$G \rightarrow id.$

Operator which is closest to the start symbol will have the least precedance & operator which is far away from start symbol will have the highest precedance.

ex

bExp → bExp or bExp.
/ bExp and bExp
/ nor bExp
/ true
/ false

Above grammar is ambiguous, because all operators are at same level & associativity of operators are also not clear.
∴ Unambiguous grammar is as follows.

E → E or F | F
F → F and G | G
G → nor G | true | false.

ex. R → R+R
/ RR
/ R*
/ a
/ b
/ c

given grammar all operators are at same precedance & associativity of operators are also not clear

$E \to E+T \mid T$

$T \to TF \mid F$

$F \to F* \mid a \mid b \mid c$

ex. $A \to A\$B \mid B$

$B \to B\#C \mid C$

$C \to C@D \mid D$

$D \to d$

$\$$ and $\#$ and $@$ are left associative. i.e. expression contain two $ or # or @

$\$ \Rightarrow \$$ indicate leftside $ has highest precedance than right side $ like $\# \Rightarrow \#$ or $@ \Rightarrow @$

$\$ < \# < @$

∴ Above grammar is unambiguous.

ex. $E \to E*F \mid F+E \mid F$

$F \to F-F$

$\mid id$

→ Grammar is ambiguous, because + & * at same level & − is far away from start symbol i.e − has highest precedance.

$* = +$, $* \Rightarrow *$

$+ < + $

Unambiguous grammar $E \to E+F \mid E-F \mid F$

$F \to F*G \mid G$

$G \to id$

In order to avoid ambiguity of grammar, we have introduce recursion of a grammar.

Recursion of grammar is of two types.

1] Left recursion.

2] Right Recursion.

1 **Left recursion :-**

$$A \rightarrow A\alpha / \beta$$

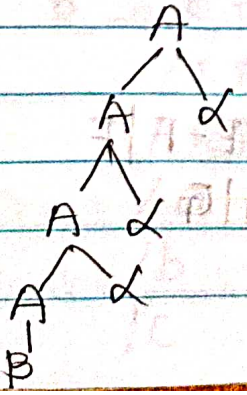
In Left recursion Leftmost symbol in RHS is equal to LHS

If We construct the function for above grammar

```
A ()
{
   A()
   α
}
```

Above code segment indicate the infinite loop situation.

Let us see parse tree for above grammar.



$$L(G) = \beta\alpha^*$$

Most of the parser perticularly Top Down parser don't allow Left recursive grammar because there are chances of falling Top down parser in infinite loop.

Therefore it is necessary to eliminate left recursion from the grammar without changing the language generated.

2) Right Recursive :-

ex

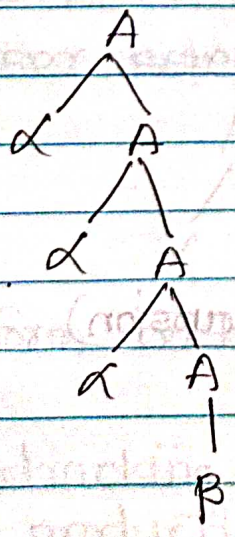
$$A \longrightarrow \alpha A / \beta$$

function :-

```
A ()
{
  α
  A()
}
```

here α is going to act as a condition checker, therefore there is no way through which parser falls in infinite loop.

parse tree :-



Regular expression for L(G)

$$\alpha^* \beta$$

Elimination of Left recursion :-

Language generated by Left recursive grammar

$$\beta\alpha^*$$

Above Language can be generated by following non left recursive grammar.

$$\boxed{\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \epsilon \mid \alpha A' \end{aligned}}$$

The above right recursive grammar is equivalent to $A \rightarrow A\alpha \mid \beta$

ex. 1] $E \rightarrow E+T \mid T$

where, $A = E$, $\alpha = +T$, $\beta = T$.

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

2] $S \rightarrow SOS1S \mid 01$

$$S \rightarrow 01S'$$
$$S' \rightarrow \epsilon \mid OS1SS'$$

3] $S \rightarrow (L) \mid x$ — (No Left recursion)

$L \rightarrow L,S \mid S$

→ $S \rightarrow (L) \mid x$

$L \rightarrow SL'$

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

4] $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots$
$\mid \beta_1 \mid \beta_2 \mid \beta_3$

$\rightarrow$

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \ldots\ldots \mid \epsilon$

Third category of grammar :-

1] Deterministic (D)

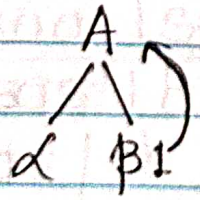
2] Non Deterministic (ND)

1] Non-Deterministic :-

ex $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$

suppose, given string is $\alpha\beta_3$.

In order to generate the string $\alpha\beta_3$ parser initially find production for α, but for α production, we are having multiple options, In such case parser select 1st production, After selecting this production parser came to know that second string is not matching. Therefore parser perform backtracking (parser fails).



Backtracking required because of common prefixes for productions

backtracking problem get arise because decision of selecting production is depends on seeing few i/p.

Therefore it is necessory to remove Non-determize & to remove non-determism we are going to apply Left factoring

Left factoring :-

means conversion of non-deterministic grammar to deterministic grammar.

ex suppose given ND is as follow.

$$A \rightarrow \alpha\beta1 \mid \alpha\beta2 \mid \alpha\beta3$$

then equivalent deterministic grammar or left factoring is as follows.

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta1 \mid \beta2 \mid \beta3$$

[We are writing common prefixes only once]

ex $S \rightarrow$ iEts
/ iEtSes
/a (assignment stm)
$E \rightarrow$ b (boolean stm)

→ Above G is ND, because iEts in common

∴ equivalent deterministic grammar.

$S \rightarrow iEtSS' | a$

$S' \rightarrow \epsilon | eS$

$E \rightarrow b$

Eliminiting LD or applying left factoring does not eliminate ambiguity.

2] $S \rightarrow aSSbS$
$| aSaSb$
$| abb$
$| b$

$\rightarrow$ $S \rightarrow aS' | b$

$S' \rightarrow SSbS | SaSb | bb$

$S' \rightarrow SS'' | bb$

$S'' \rightarrow SbS | aSb$

3] $S \rightarrow bSSaaS | bSSaSb | bSb | a$

$\rightarrow$ $S \rightarrow bSS' | a$

$S' \rightarrow SaaS | SaSb | b$

$S' \rightarrow SaS'' | b$

$S'' \rightarrow aS | Sb$