MODULE - II

SYNTAX ANALYSIS

Introduction:-

- Every programming language has some syntactic structure that should be accompanied by the statements of the program written in that particular language.
- This unit checks whether the syntax of the statements are up to the mark of source programming language and if it is thus, it constructs a parse tree.
- In this phase, we check the syntax of the grammar to construct the abstract syntax tree. This phase is modelled through Context-free Grammars (CFG).
- Grammars offer significant advantages to both language designers and compiler writers.

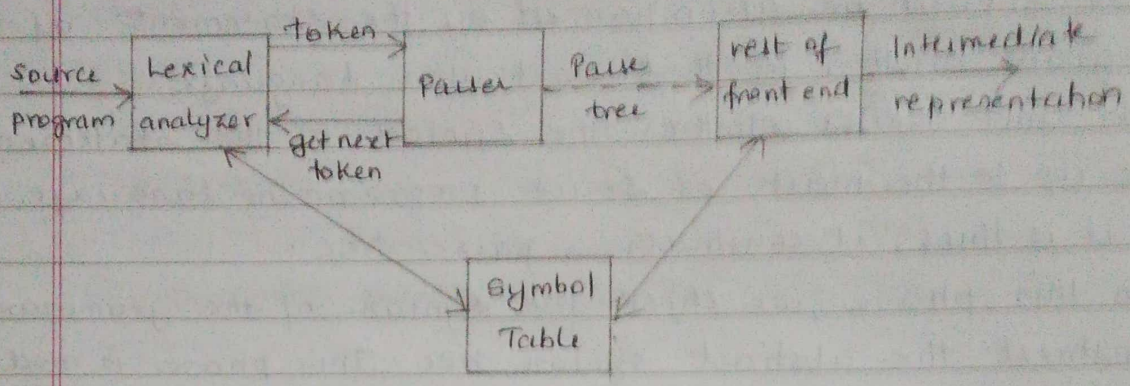
(1) A grammar gives a precise, yet easy-to-understand, syntactic specification of a programming language.

(2) From a certain classes of grammars we can automatically construct an efficient parser that determines if a source program is syntactically well formed. As an additional benefit, the parser construction process can reveal syntactic ambiguities and other difficult-to-parse constructs that might otherwise go undetected in the initial design phase of a language and its compiler.

(3) A properly designed grammar imparts a structure to a programming language that is useful for the translation of source programs into correct object code and for the detection of errors.

(4) Languages evolve over a period of time, acquiring new constructs and performing additional tasks. These new constructs can be added to a language more easily when there is an existing implementation based on a grammatical description of the language.

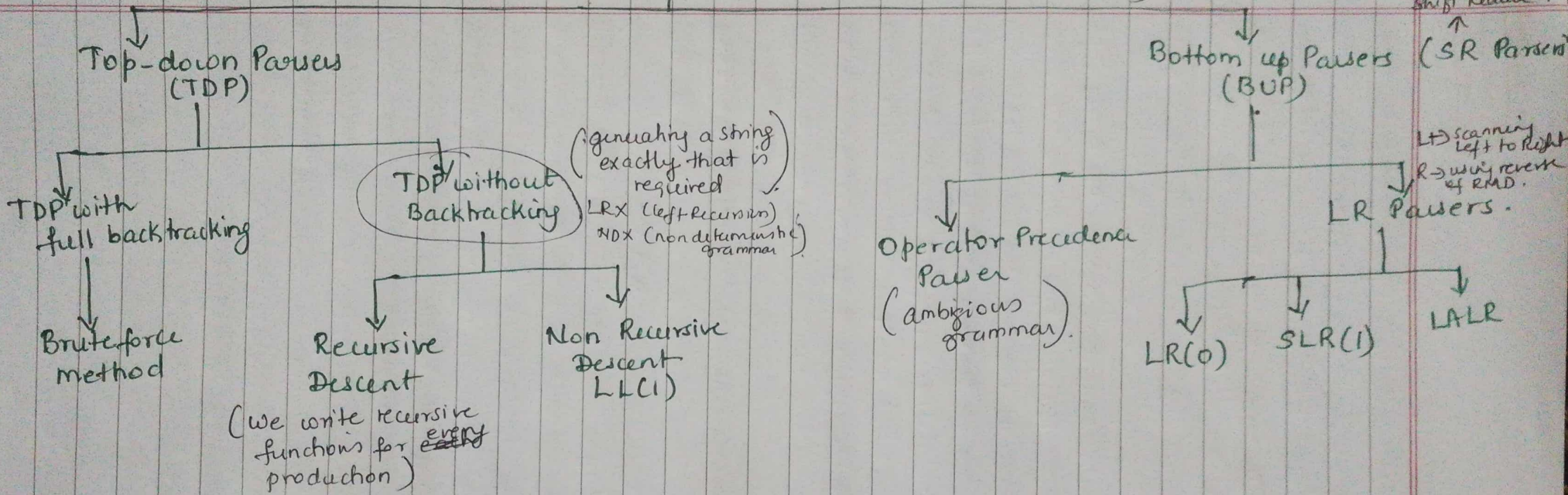
* The Role of the Parser



- The parser obtains a string of tokens from LA and verifies that the string can be generated by the grammar of the src language and the parser reports if any syntax error occurs.
- The parser receives valid tokens from the scanner and checks them against the grammar and produces the valid syntactical constructs.
- This phase checks syntax and constructs abstract syntax tree.
- There are different types of parsers: namely Top-down Parsers and bottom-up parsers.
- Top-down parsers :- build parse trees from the top (root) to the bottom (leaves).
  Bottom-up parsers :- Start from the leaves and work up the root.
- In both cases, the i/p to the parser is scanned from left to right, one symbol at a time.

PARSERS

(No Ambiguous grammar for any of below Parsers) AX.



⇒ Parsers are Syntax Analyzer

Backtracking → we try to go and analyze all possibilities and then if any of the possibility is wrong we go and choose the next possibility.

PARSING

- Parsing is the reverse of doing a derivation. It receives tokens from LA and verifies them against the grammar and produces the parse tree.

Types of Parsing :

- Top-down parsing
- Bottom-up parsing

(1) Top-down Parsing (LL(K))

- In top-down parsing the parse tree is generated from top to bottom (ie from root to leaves).
- The derivation terminates when the required i/p string is obtained.
- Top-down parsing finds leftmost derivation for an i/p string. It also constructs a parse tree for the i/p starting from the root and creating the nodes of the parse tree in preorder.

LL(K)

L → Left to right

L → Leftmost derivation

K → K stands for the number of look-ahead tokens.

Normally use 1 → scan one input at a time.

Problem with top-down parsing

(1) Backtracking
(2) Left recursion
(3) Left factoring
(4) Ambiguous.

In order to implement the parsing, we need to eliminate these problems.

Backtracking : Backtracking is a technique in which for expansion of a NT (non-terminal) symbol, we choose one alternative and if some mismatch occurs we try another alternative if any.

Eg:- $S \to xPz$

$P \to yw \mid y$

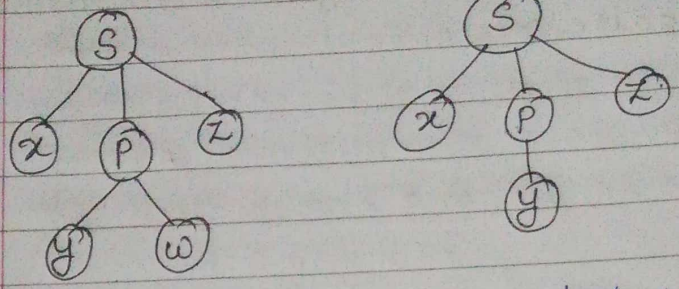


fig:- Backtracking.

- If, for a NT, there are multiple production rules beginning with the same input symbol, to get the correct derivation, we need to try all these alternatives.
- In backtracking, we need to move some levels upward in order to check the possibilities. This increases a lot of overheads in the implementation of parsing. And thus, it becomes necessary to eliminate the backtracking by modifying the grammar

- overhead ↑
- modify grammar

→ infinite loop

Left recursion:- The left-recursive grammar is a grammar which is as given below.

$$A \overset{+}{\Rightarrow} A\alpha$$

+ → deriving in one or more steps.

Here + means deriving the input in one or more steps. The A can be a NT and α denotes some input string.

- If left recursion is present in the grammar, it creates serious problems. Because of left recursion, the top-down parser can enter in infinite loop.
- Thus expansion of A causes further expansion of A only and due to generation of $A, A\alpha, A\alpha\alpha, A\alpha\alpha\alpha, \ldots$, the i/p pointer will not be sophisticated.
- This causes major problem in top-down parsing and therefore elimination of left recursion is a must.
- To eliminate left recursion, we need to modify the grammar having a production rule with left recursion.

$$A \to A\alpha \mid \beta$$

Then we eliminate left recursion by re-writing the production rules as

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

Thus, a new symbol A' is introduced. We can abovecify

whether the modified grammar is equivalent to the original or not.

- In general, to eliminate immediate left recursion among all A-production, we group the A-production as.

$$A \to A\alpha_1 | A\alpha_2 | A\alpha_3 | A\alpha_4 || A\alpha_m | \beta_1 | \beta_2 | \dots | \beta_m$$

where no β_i begins with an A. Then we replace A-prod by

$$A \to \beta_1 A' | \beta_2 A' | \dots | \beta_n A'$$

$$A' \to \alpha_1 A' | \alpha_2 A' | \dots | \alpha_m A' | \epsilon$$

Eg. Consider the grammar

1. $E \to E+T \mid T$
2. $T \to T*F \mid F$
3. $F \to (E) \mid id$

we can map this production 1 with the rule $A \to A\alpha | \beta$

$$E \to E+T \mid T$$

such that $A = E$

$\alpha = +T$

$\beta = T$

Then the production 1 becomes,

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

similarly for the production 2,

$$T \to T*F | F$$

Then the production 2 becomes

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

In production 3, there is no left recursion.

- After eliminating the immediate left recursive, the grammar for arithmetic expression can be equivalently written as

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) | id.$

Left factoring : Basically, left factoring is used when it is not clear as to which of the two alternatives is used to expand the same NT.

- By left factoring we may be able to rewrite the production in which the decision can be delayed until enough of the input is seen to make the right choice. In general, if

$$A \to \alpha\beta_1 \mid \alpha\beta_2$$

is a production, then it is not possible for us to take a decision whether to choose the first rule or the second. In such a situation the above grammar can be left factored as,

$$A \to \alpha A'$$
$$A' \to \beta_1 \mid \beta_2$$

Eg. Consider the following grammar.

$$S \to iEtS \mid iEtSeS \mid a$$
$$E \to b$$

The left factored grammar becomes

$$S \to iEtSS' \mid a$$
$$S' \to eS \mid \epsilon$$
$$E \to b$$

Ambiguity :- If there is more than one parse tree for a single input string, it is known as ambiguity. The ambiguous grammar is not desirable in top-down parsing. Hence we need to remove the ambiguity from the grammar if present.

Eg. Consider $E \to E+E \mid E*E \mid id$ as an ambiguous grammar. Designing the parse tree for id + id * id is as follows:

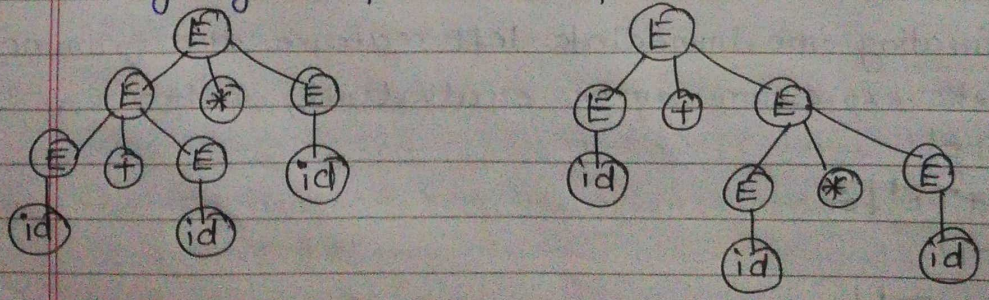


fig:- Parse tree for same string.

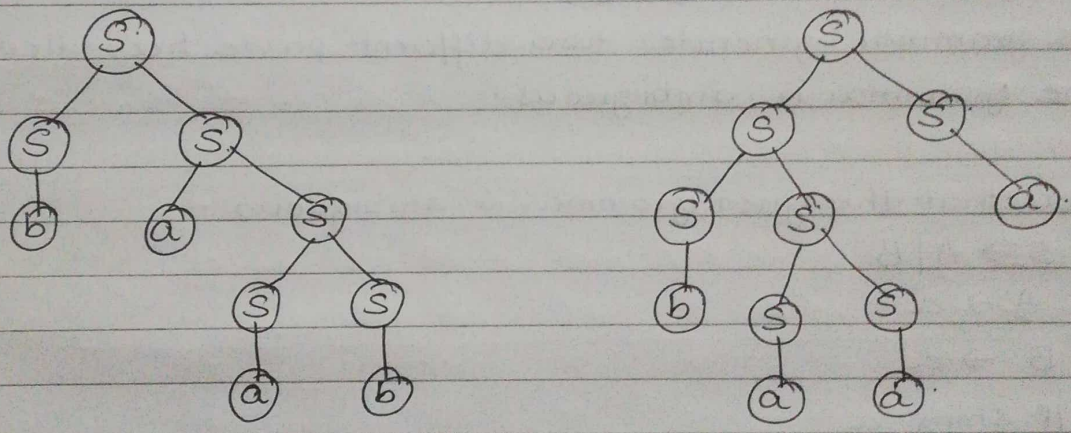
→ For removing the ambiguity, we will apply one rule:
If the grammar has a left-associative operator (such as +, -, *, /), then induce the left recursion and if the grammar has a right-associative operator (exponential operator) then induce the right recursion.
The equivalent unambiguous grammar is,

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow id$

NOTE: The grammar is unambiguous but it is left-recursive and elimination of such left recursion is again a must.

Eg: ① Show that the grammar $S \rightarrow SS \mid a \mid b$ is ambiguous.

Soln Take the string from the given grammar. Let $\omega = baaa$. Draw the parse tree.

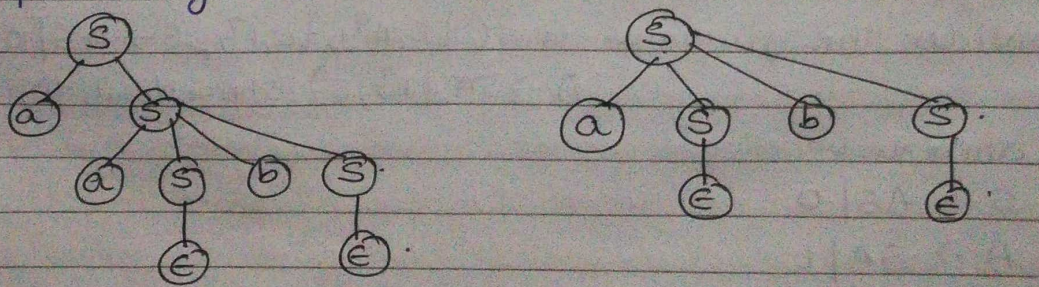


The grammar generates two different parse trees. Then, the grammar is ambiguous grammar.

Eg: ② Prove that the grammar $S \rightarrow aS \mid aSbS \mid \epsilon$ is ambiguous. Find unambiguous grammar.

Soln Input string aab.

Parse trees.



The given grammar is ambiguous, because it has more than

one parse tree for a single input string.

Un ambiguous grammar.

$S \to E$

$E \to T$

$T \to F$

$F \to a | ab | \epsilon$

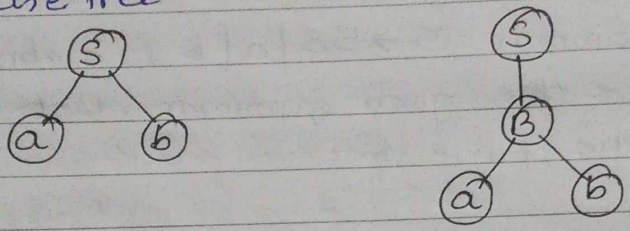
3). P.T the grammar is ambiguous.

$S \to B | ab$

$B \to ab$.

Soln Take the string $\omega = ab$

Parse tree.



The grammar generates two different parse trees. Hence the grammar is ambiguous.

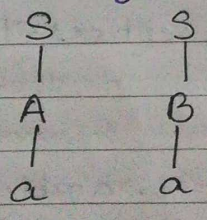
4) Show that the given grammar is ambiguous.

$S \to A | B$

$A \to a$

$B \to a$

Soln i/p string a



It generates 2 different parse trees, hence grammar is ambiguous.

5) Consider the grammar $G = (\{S, A\}, \{0, 1\}, S \to AS | 0, A \to SA | 1\}, S)$. Show that this grammar is ambiguous.

$S \to AS | 0$

$A \to SA | 1$

Ambiguous → for a single i/p string, we have diff parse

For string 00110