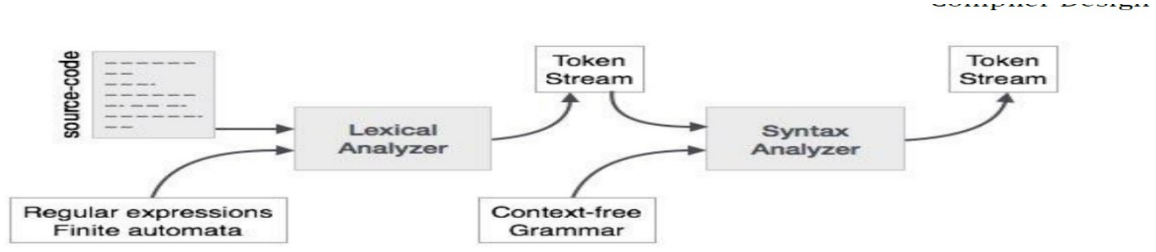
Q.1 Syntax analyzer.

Ans.

- A grammar gives precise and easy-to-understand syntactic specification of programming language.
- Syntax analysis or parsing is the second phase of a compiler.
- lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG).
- CFG is a helpful tool in describing the syntax of programming languages.
- A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams.
- The parser analyses the source code (token stream) against the production rules to detect any errors in the code.
- The output of this phase is a parse tree.

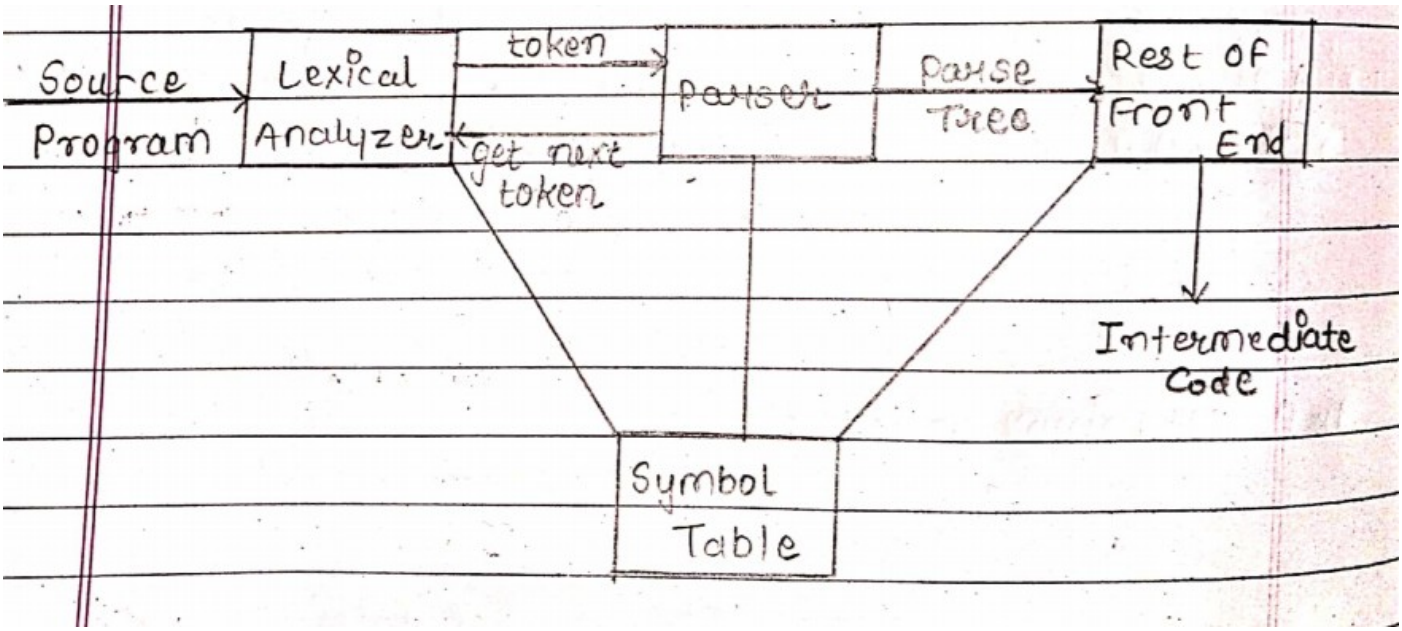


This way, the parser accomplishes two tasks, i.e., parsing the code, looking for errors, and generating a parse tree as the output of the phase.

Parsers are expected to parse the whole code even if some errors exist in the program. Parsers use error recovering strategies, which we will learn later in this chapter.

Q.2 Role of parser.

Ans.



➢ Role of parser is given below:

1. performs context-free syntax analysis

- In the compiler model, the parser obtains a string of tokens from the lexical analyser, and verifies that the string can be generated by the grammar for the source language.

2. guides context-sensitive analysis and constructs an intermediate representation

- After The parser collects sufficient number of tokens and it builds a parse tree.

3. produces meaningful error messages

- Then by building the parse tree, parse smartly finds the syntactical errors if any.

4. attempts error correction

- It is also necessary that the parse should recover from commonly occurring errors so that remaining task of process the input can be continued.

  - *Parsing* is the process where we take a particular sequence of words and check to see if they correspond to the language defined by some grammar.
  - Basically, what we have to do is to show how we can get:
    1. From the start symbol of the grammar
    2. To the sequence of words that supposedly belong to the language
    3. By using the production rules.

Q.3 *Syntax Error Handling*.

Ans.

⇨ *Syntax Error Handling:*

- Planning the error handling right from the start can both simplify the structure of a compiler and improve its response to errors.

➢ The program can contain errors at many different levels. e.g.,

- *Lexical* – such as misspelling an identifier, keyword, or operator.
- *Syntax* – such as an arithmetic expression with unbalanced parenthesis.
- *Semantic* – such as an operator applied to an incompatible operand.

- *Logical* – such as an infinitely recursive call.

➢ Much of the error detection and recovery in a compiler is cantered on the syntax analysis phase.

- One reason for this is that many errors are syntactic in nature or are exposed when the stream of tokens coming from the lexical analyser disobeys the grammatical rules defining the programming language.
- Another is the precision of modern parsing methods; they can detect the presence of syntactic errors in programs very efficiently.

➢ The error handler in a parser has simple goals:

- It should the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect subsequent errors.
- It should not significantly slow down the processing of correct programs.

➢ *Error-Recovery Strategies:*

- There are many different general strategies that a parser can employ to recover from a syntactic error.

1. Panic mode

- This is used by most parsing methods.
- On discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens (delimiters; such as; semicolon or end) is found.
- Panic mode correction often skips a considerable amount of input without checking it for additional errors.

- It is simple.

2. Phrase level

- On discovering an error; the parser may perform local correction on the remaining input; i.e., it may replace a prefix of the remaining input by some string that allows the parser to continue.
- e.g., local correction would be to replace a comma by a semicolon, deleting an extraneous semicolon, or insert a missing semicolon.
- Its major drawback is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection.

3. Error productions:

- If an error production is used by the parser, can generate appropriate error diagnostics to indicate the erroneous construct that has been recognized in the input.

Q.3 Context free grammar.

Ans.

- CFG consists of set of terminals, set of non-terminals, a start symbol and set of productions.
- In other word Formally a CFG *G* is made up with a 4-tuple (*Vt* ,*Vn*,*S*,*P*) or G=(N,**Σ**,p,s).

1. Terminals: are the basic symbols form which string are formed.

- Using Vt or Σ is representing in the grammar as a set of terminals.
- For our purposes, Vt is the set of tokens returned by the scanner.
- The token is also synonyms for terminal.
- Terminals can be small case latter's[a,b,c,…,z], digit, operator, punctuation mark or separator.

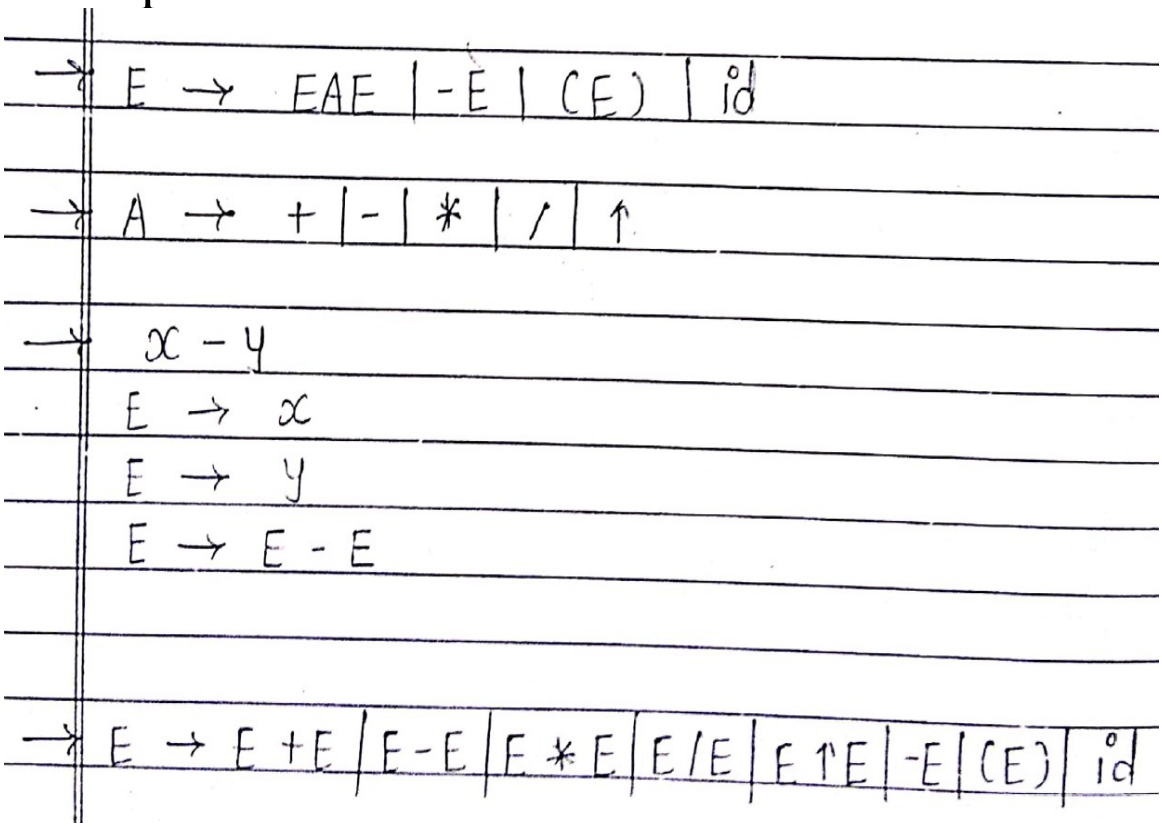
2. Non-terminals: they are syntactic variables that denote sets of string.
   - Vn or N is use for the nonterminal
   - It is a set of syntactic variables that denote sets of (sub)strings occurring in the language.
   - These are used to impose a structure on the grammar.
   - Non-terminal can be uppercase latter's [A,B,C, …..,Z]

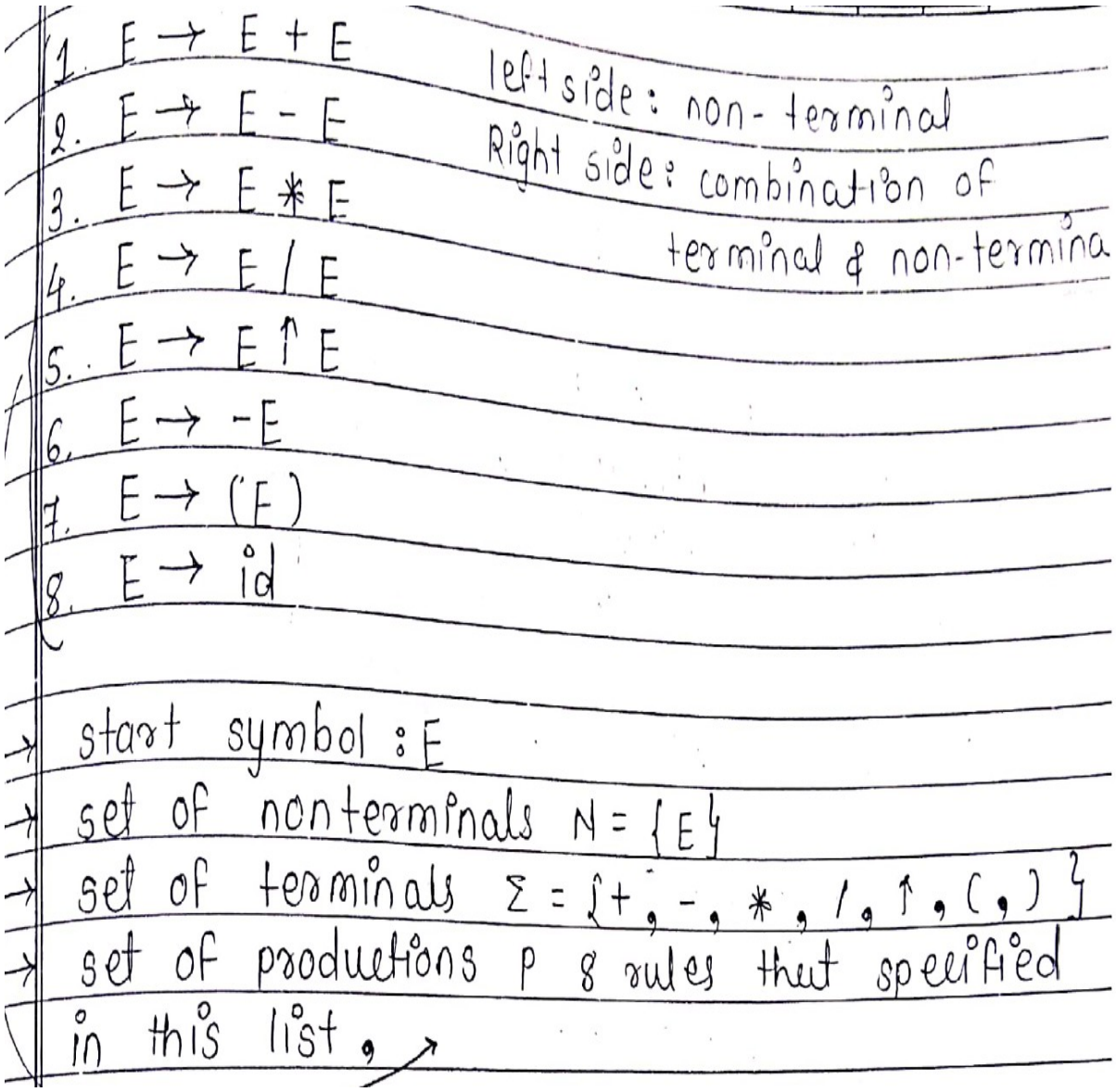
3. Start symbol: S is a distinguished nonterminal (S 2 Vn) denoting the entire set of strings in L(G).
   - In simple word sometimes called a goal/start symbol.
   - And also, it denotes the language define by the grammar.
   - Non-terminal symbol act as start symbol.

4. Production: p is a finite set of productions specifying how terminals and non-terminals can be combined to form strings in the language.
   - Each production must have a single non-terminal on its left-hand side.

- In other word each production consists of a non-terminal followed by an arrow followed by a string of no-terminals and terminals.
- terminal symbol act as production.

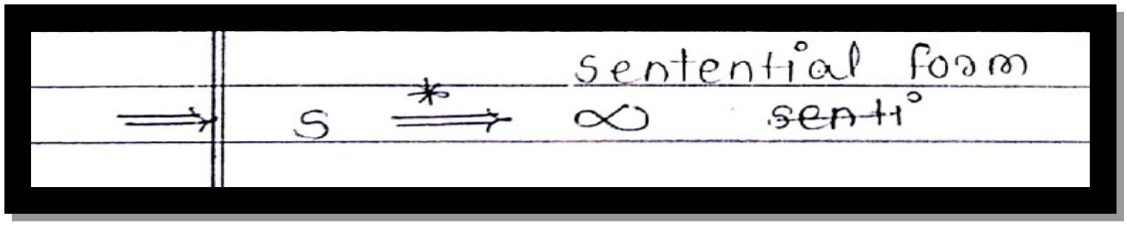
- Example:





⇨ Why context free gamer is use (advantage):

- precise syntactic specification of a programming language
- easy to understand, avoids ad hoc definition
- easier to maintain, add new language features
- can automatically construct efficient parser
- parser construction reveals ambiguity, other difficulties
- imparts structure to language
- supports syntax-directed translation



⇨ Sentence contains only terminal (E) symbols where sentinel form contains both symbols

⇒ Given a grammar G with start symbol S, a language generated by G is defined as follows:

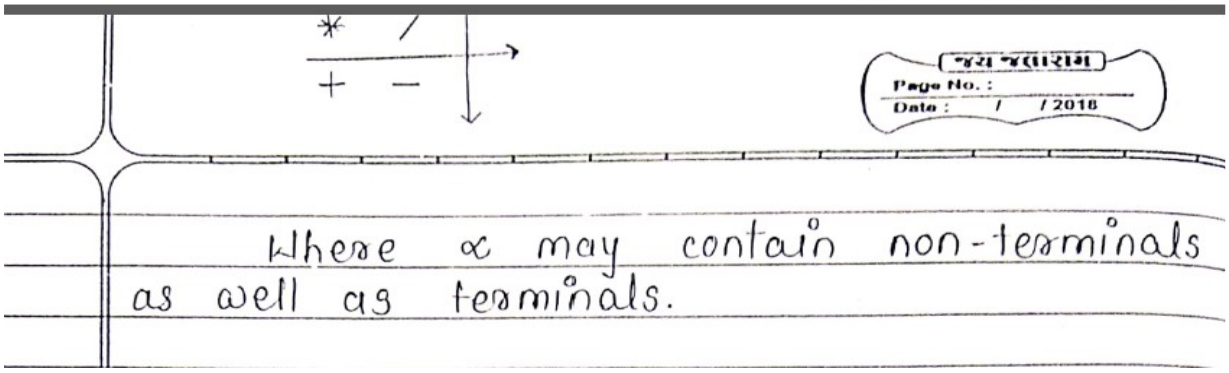
$L(G) = \{ \omega / S \overset{+}{\Rightarrow} \omega$ and ω consists of terminals only $\}$

⇒ Sentence of G: $S \overset{+}{\Rightarrow} \omega$
A sentence of a given grammar G is a string of terminals derived from the start symbol S in one or steps.

⇒ Sentential form is a string α of terminals / Non-terminals derived from the start symbol S after zero or more steps:

$S \overset{*}{\Rightarrow} \alpha$

Scanned by CamScanner



Where α may contain non-terminals as well as terminals.

⇨ **C**ontext **F**ree **G**rammars(CFGs) are classified based on:

- Number of Derivation (parse) trees
- Number of strings

⇨ Depending on Number of Derivation trees, CFGs are sub-divided into 2 types:

- Ambiguous grammars
- Unambiguous grammars

Q.4 Derivation.

Ans.

- Derivation is a sequence of production rules.
- It is used to get the input string through the production rules.
- During parsing we have to take two decisions.

⇨ Two decision we take are as follows:

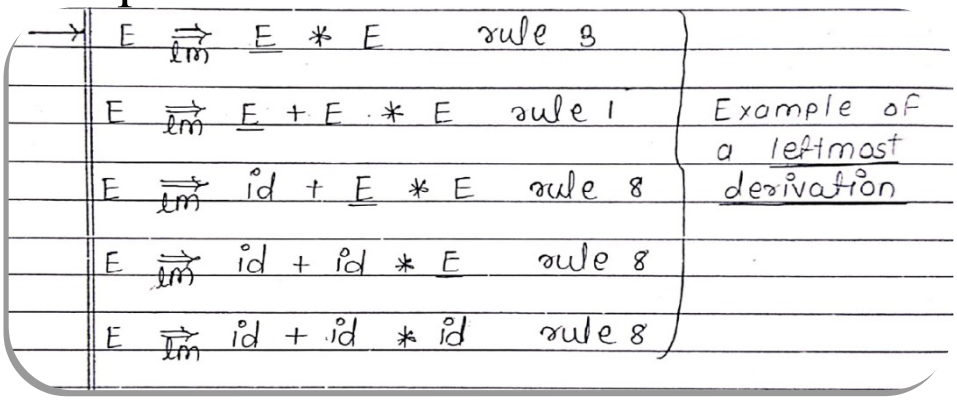
- We have to decide the non-terminal which is to be replaced.
- We have to decide the production rule by which the non-terminal will be replaced.

- Derivation is used to find whether the string belongs to a given grammar.
- Derivation Checking is done by rewriting rule in which the non-terminal on left is replaced by the string on right side of the production.
- We can apply changes one at a time in sequence.
- A derivation can be draw as a tree in that start symbol is root.
- Derivation tree (parse tree) have terminal at the leaves and non-terminal at the interior nodes.
- Derivation or parse tree show the association of operations with input string.
- The process of discovering derivation is called parsing.

⇨ **Note:** Right-most and left-most derivation have the same parse tree. If that not happen then the grammar is ambiguous.

⇨ We have two options to decide which non-terminal to be replaced with production rule.

1. Left most derivation.:
   - In the left most derivation, the input is scanned and replaced with the production rule from left to right.
   - So, in left most derivatives we read the input string from left to right.
   - Example:



2. Right most derivation.:
   - In the right most derivation, the input is scanned and replaced with the production rule from right to left.
   - So in right most derivatives we read the input string from right to left.
   - Example:

* Derive id + id * id using the given grammar. rm = rightmost lm = leftmost

→ $E \underset{rm}{\Rightarrow} E + E$

$E \underset{rm}{\Rightarrow} E + E * E$ rule 3

$E \underset{rm}{\Rightarrow} E + E * id$ rule 8

$E \underset{rm}{\Rightarrow} E + id * id$ rule 8

$E \underset{rm}{\Rightarrow} id + id * id$ rule 8

Example of a rightmost derivation

Q.4 Parse tree and Derivation.

Ans.

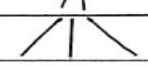
** Parse Tree and Derivation **

A parse tree may be viewed as a graphical representation a derivation that filters out the choice regarding replacement order.

Each interior node of a parse tree is labeled by some non-terminal. (A)

$A \rightarrow xyz$

[If A is a non-terminal and derives xyz.]



children of this node are labeled from left to right, by the symbols in the right side of the production by which this A is replaced in the derivation.

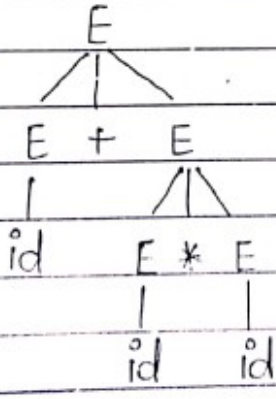
The leaves of the parse tree are labeled by non-terminals or terminals, and read from left to right.

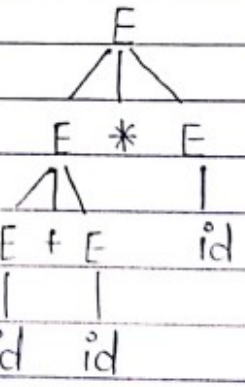
Example :

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid -E \mid (E) \mid id$

The sentence id + id * id has to distinct leftmost derivations :

$E \underset{lm}{\Rightarrow} E+E$	$E \underset{lm}{\Rightarrow} E*E$
$\underset{lm}{\Rightarrow} id+E$	$\underset{lm}{\Rightarrow} E+E*E$
$\underset{lm}{\Rightarrow} id+E*E$	$\underset{lm}{\Rightarrow} id+E*E$
$\underset{lm}{\Rightarrow} id+id*E$	$\underset{lm}{\Rightarrow} id+id*E$
$\underset{lm}{\Rightarrow} id+id*id$	$\underset{lm}{\Rightarrow} id+id*id$
parse tree for this derivation	parse tree for the given derivation





Q.5 what is ambiguity and how to remove it

Ans.

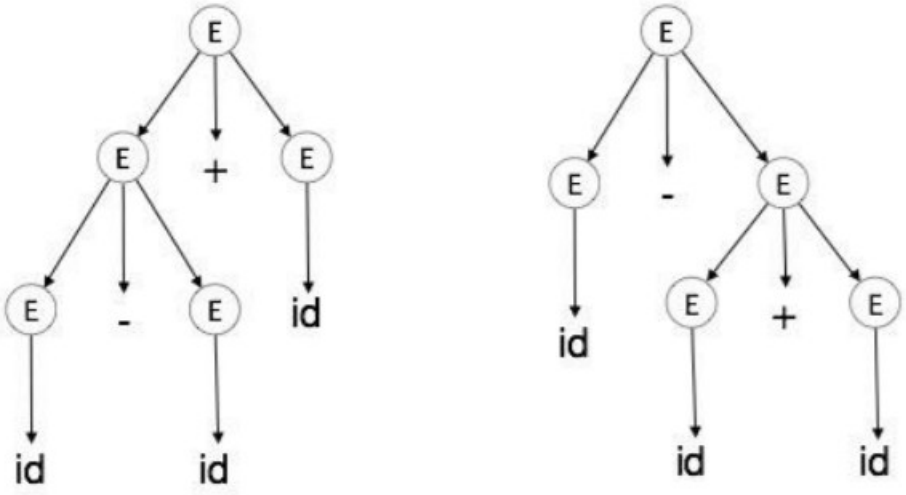
- If a grammar has multiple left most or right most derivation for a single sentinal form, the grammar is ambiguous.
- Also, we can say a grammar produce more than one parse tree for some sentence then it said to be an ambiguous grammar.
- In simple word if one line (sentential form) have two derivation then its context-free grammar ambiguity.
- Ambiguity generally refers to confusion in the context-free grammar.
- Ambiguity is not efficient in compilation process.

➢Example:

Example

```
E → E + E
E → E – E
E → id
```

For the string id + id – id, the above grammar generates two parse trees:



⇨ How to remove ambiguity:

- No method can detect and remove ambiguity automatically.
- but it can be removed by either re-writing the whole grammar without ambiguity.
- or by setting and following associativity and precedence constraints.

⇨ **Associativity:**

- If an operand has operators on both sides, the side on which the operator takes this operand is decided by the associativity of those operators.
- If the operation is left-associative, then the operand will be taken by the left operator.
- or if the operation is right-associative, the right operator will take the operand.

⇨ **Precedence:**

- If two different operators share a common operand, the precedence of operators decides which will take the operand.
- That is, id+id-id can have two different parse trees, one corresponding to (id+id)-4 and another corresponding to id+(id-id).
- By setting precedence among operators, this problem can be easily removed.
- As in the previous example, mathematically + (addition) has precedence over - (subtraction), so the expression id+id-id will always be interpreted as: (id+id)-id These methods decrease the chances of ambiguity in a language or its grammar.

- Parse tree without ambiguate.

⇨ Example:

Example: Eliminate ambiguity from the following "dangling-else" grammar:

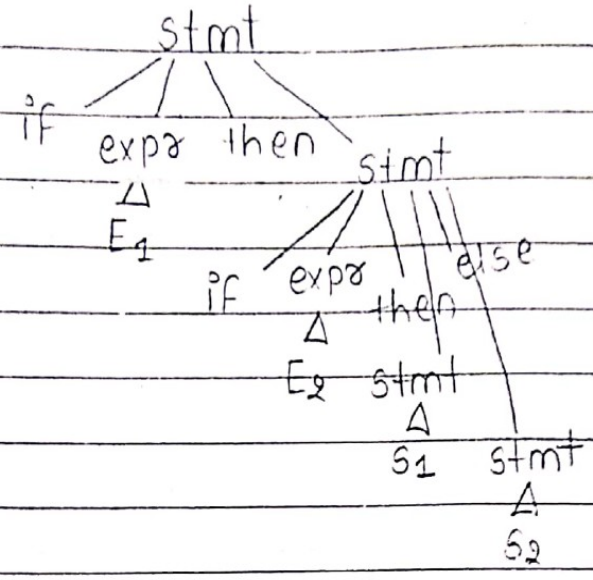
ambiguous grammar

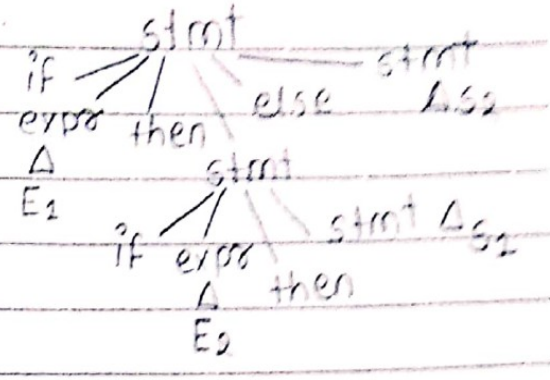
stmt → if expr then stmt |
if expr then stmt else stmt |
other

other → stands for any other statement

if E_1 then if E_2 then S_1 else S_2 has two parse trees.

Two Different possible ways to deriving statement





This is parse tree1

This is parse tree2

for the same sentence

Solution: Rewrite the "dangling-else" grammar as follows to eliminate ambiguity:

stmt → matched_stmt | unmatched_stmt

where,

matched_stmt → if expr then matched_stmt else matched_stmt | other

unmatched_stmt → if expr then stmt | if expr then matched_stmt else unmatched_stmt

⇨ General rule: "Match each else with the closet previous unmatched then" -disambiguate rule

Q.5 Difference between ambiguate and dis-ambiguate.

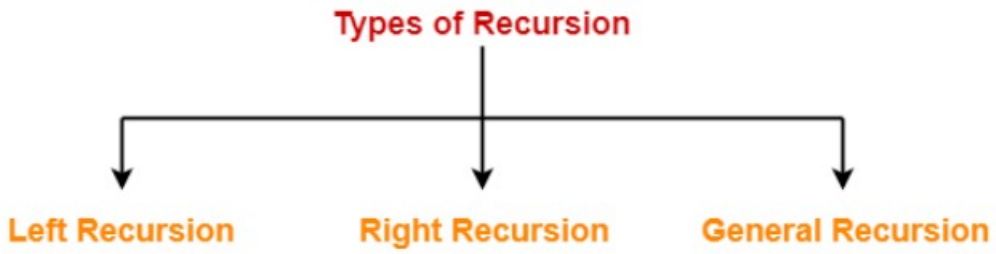
Ans.

Ambiguous Grammar	Unambiguous Grammar
A grammar is said to be ambiguous if for at least one string generated by it, it produces more than one- • parse tree • or derivation tree • or syntax tree • or leftmost derivation • or rightmost derivation	A grammar is said to be unambiguous if for all the strings generated by it, it produces exactly one- • parse tree • or derivation tree • or syntax tree • or leftmost derivation • or rightmost derivation
For ambiguous grammar, leftmost derivation and rightmost derivation represents different parse trees.	For unambiguous grammar, leftmost derivation and rightmost derivation represents the same parse tree.
Ambiguous grammar contains less number of non-terminals.	Unambiguous grammar contains more number of non-terminals.
For ambiguous grammar, length of parse tree is less.	For unambiguous grammar, length of parse tree is large.
Ambiguous grammar is faster than unambiguous grammar in the derivation of a tree. (Reason is above 2 points)	Unambiguous grammar is slower than ambiguous grammar in the derivation of a tree.
Example- $E \rightarrow E + E / E \times E / id$ (Ambiguous Grammar)	**Example-** $E \rightarrow E + T / T$ $T \rightarrow T \times F / F$ $F \rightarrow id$ (Unambiguous Grammar)

Q.6 what is recursion.

Ans.

Recursion can be classified into following three types-



1. Left Recursion-

- A production of grammar is said to have **left recursion** if the leftmost variable of its RHS is same as variable of its LHS.
- A grammar containing a production having left recursion is called as Left Recursive Grammar.
- Left recursion is considered to be a problematic situation for Top down parsers.
- Top-down parsers start parsing from the Start symbol, which in itself is non-terminal.
- So, when the parser encounters the same non-terminal in its derivation, it becomes hard for it to judge when to stop parsing the left non-terminal and it goes into an infinite loop.
- Therefore, left recursion has to be eliminated from the grammar.

⇨ **Example:** $S \rightarrow Sa / \in$

Example:

```
(1) A => Aα | β

(2) S => Aα | β
    A => Sd
```

2. Right Recursion-

- A production of grammar is said to have **right recursion** if the rightmost variable of its RHS is same as variable of its LHS.
- A grammar containing a production having right recursion is called as Right Recursive Grammar.
- Right recursion does not create any problem for the Top down parsers.
- Therefore, there is no need of eliminating right recursion from the grammar.

⇨ **Example:** $S \rightarrow aS / \in$

3. General Recursion-

- The recursion which is neither left recursion nor right recursion is called as general recursion.

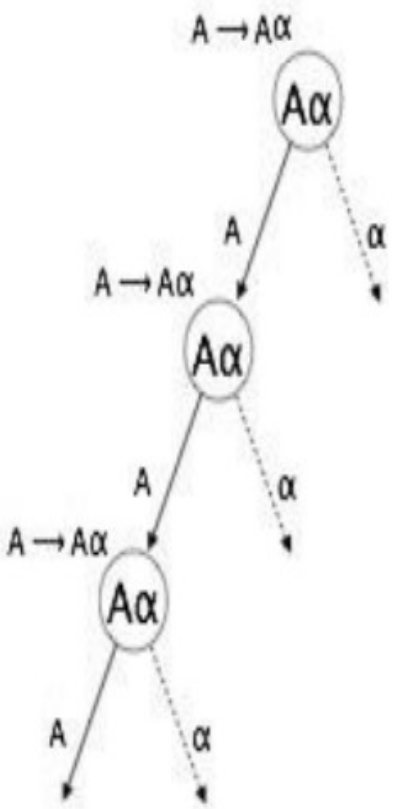
⇨ **Example:** $S \rightarrow aSb / \in$

Q.8 How to remove left recursion and what is factoring.

Ans.

(1) is an example of immediate left recursion, where A is any non-terminal symbol and α represents a string of non-terminals.

(2) is an example of indirect-left recursion.



A top-down parser will first parse A, which in-turn will yield a string consisting of A itself and the parser may go into a loop forever.

Elimination of Left Recursion

Left recursion is eliminated by converting the grammar into a right recursive grammar.

If we have the left-recursive pair of productions-

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

(Left Recursive Grammar)

where β does not begin with an A.

Then, we can eliminate left recursion by replacing the pair of productions with-

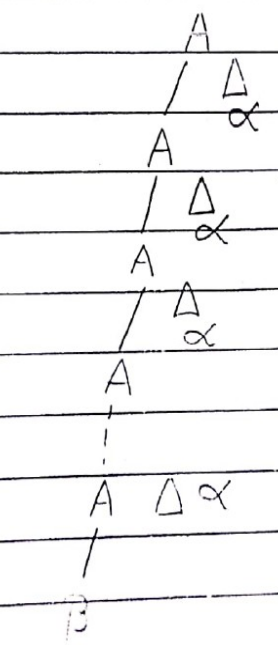
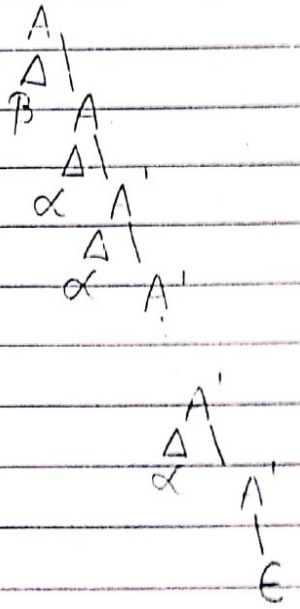
$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' / \in$$

(Right Recursive Grammar)

This right recursive grammar functions same as left recursive grammar.

- This does not impact the string derived from the grammar, but it removes immediate left recursion.
- And We can eliminate the left-recursion by introducing new non-terminals and new productions rules.

Derived using the given grammar	Derived using the new grammar
	
$\beta\alpha\alpha\alpha\alpha \ldots \alpha$	$\beta\alpha\alpha\alpha\alpha \ldots \alpha$

Example:

1. Consider a following grammar for arithmetic expressions:

left recursive grammar:

$E \rightarrow E + T \mid T$ where T is another non-Terminal

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$; id = identifier

Terminal symbols: +, *, (,), id

According to $A \rightarrow \beta A'$ / $A' \rightarrow \alpha A' \mid \epsilon$ this rule

$E \rightarrow TE'$

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

$F \rightarrow (E)$

$F \rightarrow id$

$T \rightarrow FT'$

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

Non-left recursive

2. $A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ can be transformed to be a new grammar

$$A \to \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$

$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \epsilon$$

3. consider the following grammar

$S \to Aa \mid b$ indirect left recursive

A → Non-terminal

a, b → terminal

$A \to Ac \mid Sd \mid \epsilon$

1. We order the non-terminals S, A.
2. i=1 nothing happens, for i=2 we substitute the S-productions (S deriving something) in $A \to Sd$ to obtain

$A \to Ac \mid Aad \mid bd \mid \epsilon$

Eliminate now the immediate left recursion among the A-productions, to obtain the following grammar.

$S \to Aa \mid b$

$A \to bdA' \mid \epsilon A'$ (ε is nothing so no need to write)

$A \to cA' \mid adA' \mid \epsilon$

⇨ Second method is to use the following algorithm, which should eliminate all direct and indirect left recursions.

Algorithm for eliminating left recursion.

INPUT: Grammar G with no cycles for ϵ-productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD:

1. Arrange the non-terminals in some order $A_1, A_2, A_3, \ldots, A_n$.

2. for i := 1 to n do
   begin
      for j := 1 to i-1 do
      begin
         Replace each production of the form
         $A_i \rightarrow A_j\gamma$ by the productions
         $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$
         where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
      end
      Eliminate the immediate left recursion
      among the A_i productions.
   end

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \epsilon$

SOLUTION:

1. We order the non-terminals S, A
2. i = 1 nothing happens
   for i = 2 we substitute the S-production in $A \rightarrow Sd$ to obtain
   $A \rightarrow Ac \mid Aad \mid bd \mid c$

Eliminate now the immediate left recursion among the A-production to obtain the following grammar.

$S \rightarrow Aa \mid b$
$A \rightarrow bdA' \mid A'$
$A' \rightarrow cA' \mid adA' \mid \epsilon$

This is the new grammar.

⇨ Left Factoring:

- If more than one grammar production rules has a common prefix string, then the top-down parser cannot make a choice as to which of the production it should take to parse the string in hand.
- Example If a top-down parser encounters a production like

  $A \Longrightarrow \alpha\beta \mid \alpha\gamma \mid \ldots$
- Then it cannot determine which production to follow to parse the string, as both productions are starting from the same terminal (or non-terminal).
- To remove this confusion, we use a technique called left factoring.
- Left factoring transforms the grammar to make it useful for top-down parsers.
- In this technique, we make one production for each common prefixes and the rest of the derivation is added by new productions.

Example

The above productions can be written as

```
A => αA'
A'=> β | γ | ...
```

Now the parser has only one production per prefix which makes it easier to take decisions.

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.

If $A \rightarrow \alpha\beta_1 | \alpha\beta_2$ are two A-productions, and the input begins with a non-empty string derived from α, we do not know wheather to expand A to $\alpha\beta_1$ or to $\alpha\beta_2$.

However, we may defer the decision by expanding A to $\alpha A'$. Then, after seeing the input derived from α, we expand A' to β_1 or β_2.

left-Factored grammar:

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_2 | \beta_1$

જય જલારામ
Page No. :
Date : / / 2018

Algorithm : left Factoring a grammar

INPUT : Grammar G

OUTPUT : An equivalent left-Factored grammar.

Method : For each non terminal A, find the longest prefix α common ~~to~~ to two or more ~~atter~~ alternatives.
Replace all A-productions.

original grammar $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$

When γ represents by all alternatives that do not begin with α

$$A \to \alpha A' \mid \delta$$
$$A' \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

Here, A' in a new non-terminal.

Parser

Top-down — Bottom-up

handle turning Pa...

- Recursive descent parser
- Predictive Parser

- shift Reduce parser
- Operator Precedence parser

LR Parser
- SLR Parser
- LALR Parser
- Cannonical LR Parser

Q.7 Parsing Technique difference.

Ans.

Top-down versus bottom-up

Top-down parsers

- start at the root of derivation tree and fill in
- picks a production and tries to match the input
- may require backtracking
- some grammars are backtrack-free (*predictive*)

Bottom-up parsers

- start at the leaves and fill in
- start in a state valid for legal first tokens
- as input is consumed, change state to encode possibilities (*recognize valid prefixes*)
- use a stack to store both state and sentential forms

Q.8 Parsing Technique Top-Down parsing.

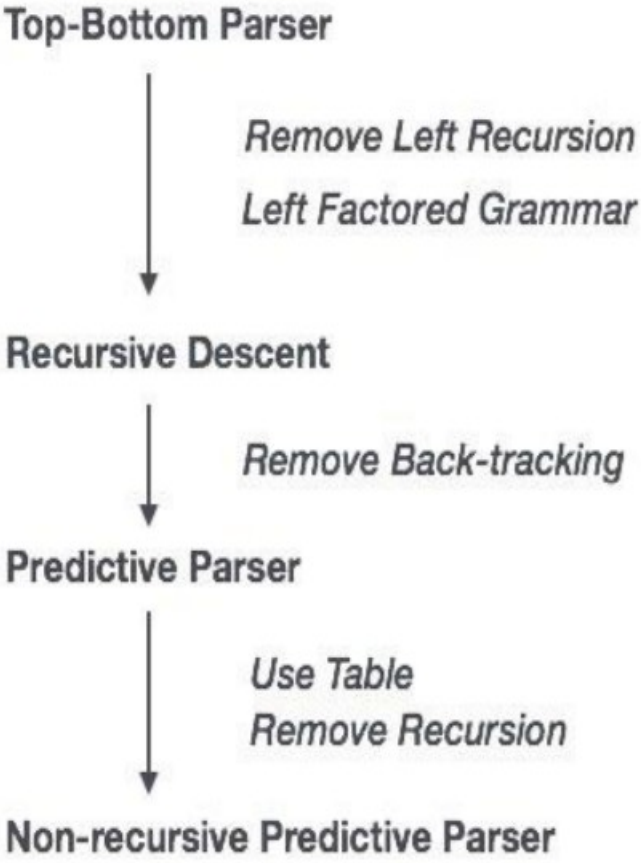
Ans.

⇨ Top-Down Parsing:

- When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.
- The top-down parsing technique parses the input, and starts constructing a parse tree from the root node gradually moving down to the leaf nodes.

⇨ The types of top-down parsing are depicted below:

1. **Recursive descent parsing**:
   - It is called recursive, as it uses recursive procedures to process the input.
   - Recursive descent parsing suffers from backtracking.



* Recursive - Descent Parsing *

- Top - down parsing technique

- Top - down parsing can be viewed as an attempt to find a leftmost derivation for an input string.

It can be viewed as an attempt to construct a parse tree for the input string starting from the root and creating the nodes of the parse tree in preorder.

Recursive - descent parsing is a general form of top - down parsing which may involve backing (making repeated scans of the input.

Example:

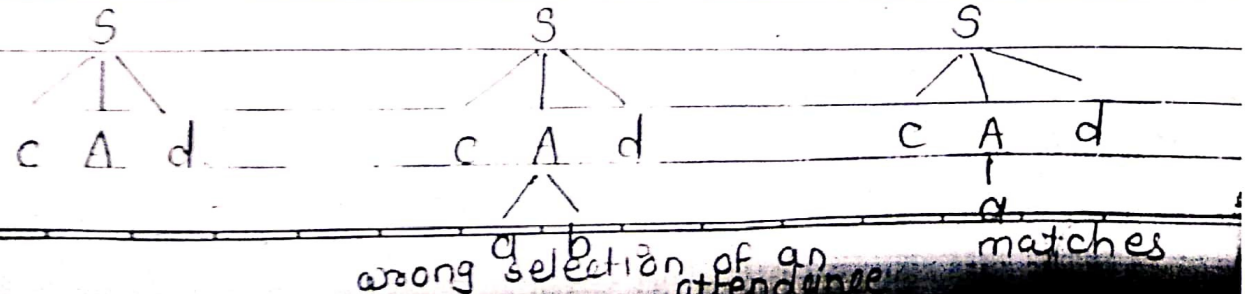
consider the following grammar:

$S \rightarrow CAd$

$A \rightarrow ab \mid a$

input string $\omega = cad$

construct the parse tree for the given input string in top - down fashion.



wrong selection of an attendance

matches

2. **Predictive Parser**:

- Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string.
- The predictive parser does not suffer from backtracking.
- To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols.
- To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.

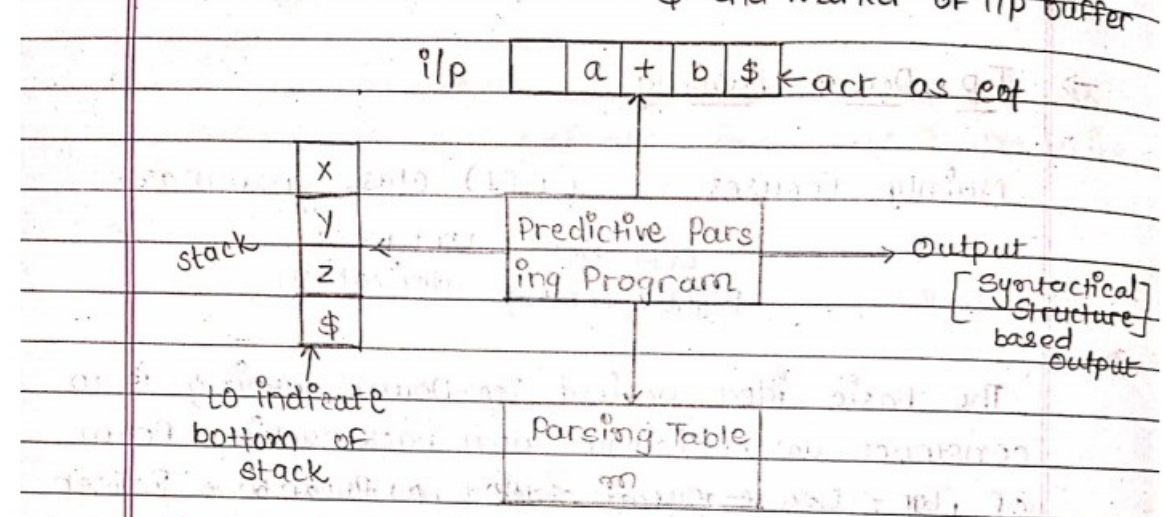
→ Predictive Parser.

In many cases, by carefully writing a grammar, eliminating (L.R.) from it, and left

Date
Page 52

Factoring the resulting grammar, we can obtain a grammar that can be parsed by a recursive descent parser that needs no backtracking i.e., a predictive parser.

Non-Recursive Predictive Parsing.



Model of N-R-P-P

★. Note :- Model of N-R-P-P = P-P = Top-down.

A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream.

→ The input buffer contains the string to be parsed, followed by $, a symbol used as right end marker to indicate the end of the input string.

→ The stack contains a sequence of grammar symbols with $ at the bottom of stack.
$ indicates the bottom of the stack.
Initially the stack contains the start symbol of the given grammar on top of $

→ The parsing table is a two-dimensional array M[A, a], where A is non-termina and a is terminal or $.

→ The parser is controlled by a program that behaves as follows:

- The program considers X, the symbol on top of the stack, and a, the current input symbol.
- These two symbols determine the action of the parser.

→ These are 3 posibilities:

1. If $X = a = \$$, the parser halts and announces successful completion of parsing.

2. If $X = a \neq \$$, the parser POPs X from the stack and advances the input pointer to the next input symbol.

3. If X is a non-terminal, the program consults entry $M[X, a]$ of the parsing table M.
This entry will be either an X-production of the grammar or an error entry.
If, for example, $M[X, a] = \{ X \rightarrow UVW \}$,
The parser replaces X on top of the stack by WVU (with U on top)
If $M[X, a] =$ error, parser calls on error recovery routine

2017
2018

Algorithm: Non-recursive predictive

INPUT: A string ω and parsing table M for grammar G.

OUTPUT: If ω is in L(G), a leftmost derivation of ω; otherwise, an error indication

METHOD: Initially, the parser is in a configuration in which it has \$S on the stack with S on top.
Moreover $\omega\$$ is in the input buffer.

INITIAL CONFIGURATION	INPUT BUFFER	STACK
	$\omega\$$	\$S

ALGORITHM: set ip to point to the first symbol of $\omega\$$;
repeat
Let X be the top stack symbol and a the symbol pointed to by ip;
if X is a terminal or \$ then
if X = a then
POP X from the stack and advance ip;
else
error();
/* X is a non-terminal */
if M[X, a] = $X \rightarrow Y_1 Y_2 Y_3 \ldots Y_K$ then
begin
POP X from the stack;
PUSH $Y_K Y_{K-1} \ldots Y_1$ onto the stack with Y_1 on top

OUTPUT the production $X \rightarrow Y_1 Y_2 \ldots Y_K$
END
Else
error()
X = \$
until

Q.9 Explain Back-tracking first and follow

Ans.

⇨ Back-tracking

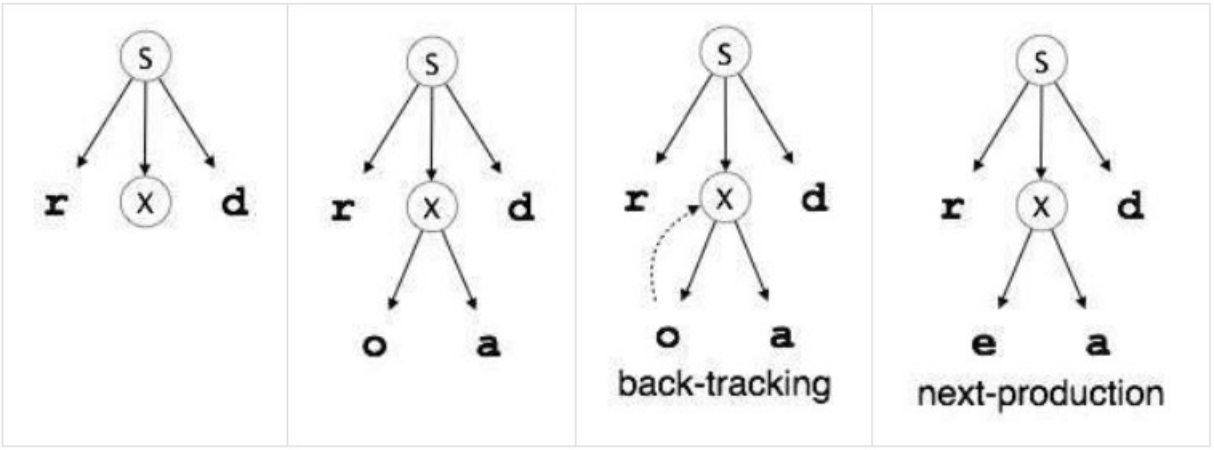
- Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched)
- To understand this, take the following example of CFG:

```
S → rXd | rZd
X → oa | ea
Z → ai
```

For an input string: read, a top-down parser, will behave like this:

It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. 'r'. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. 'e'). The parser tries to expand non-terminal 'X' and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).

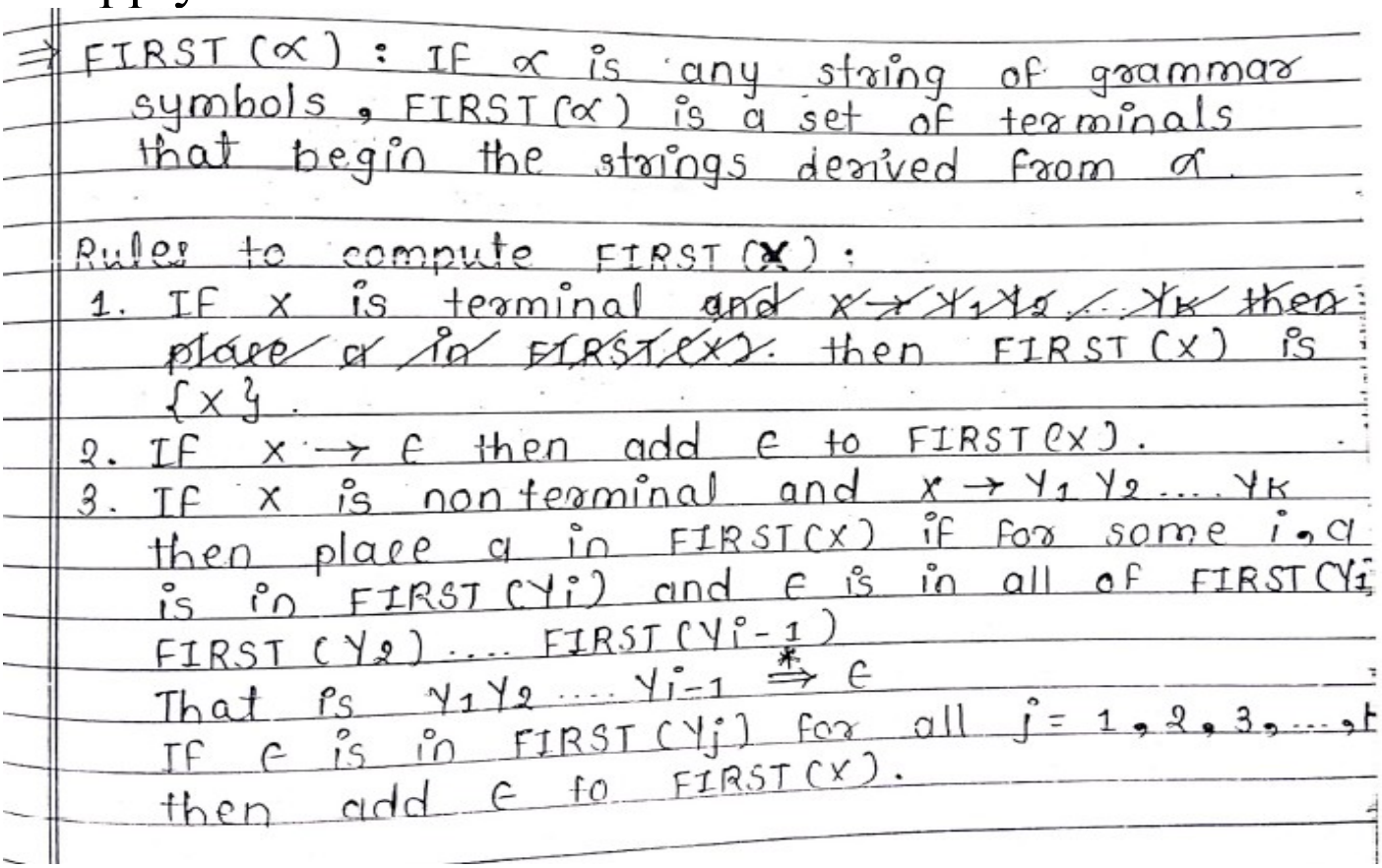
Now the parser matches all the input letters in an ordered manner. The string is accepted.



⇨ First:

- We saw the need of backtrack in parsing, which is really a complex process to implement.
- There can be easier way to sort out this problem.
- If the compiler would have come to know in advance, that what is the "first character of the string produced when a production rule is applied", and comparing it to

the current character or token in the input string it sees, it can wisely take decision on which production rule to apply.



⇨ Follow:

The parser faces one more problem. Let us consider below grammar to understand this problem.

```
A -> aBb
B -> c | ε
And suppose the input string is “ab” to parse.
```

As the first character in the input is a, the parser applies the rule A->aBb.

```
      A
    / | \
   a  B  b
```

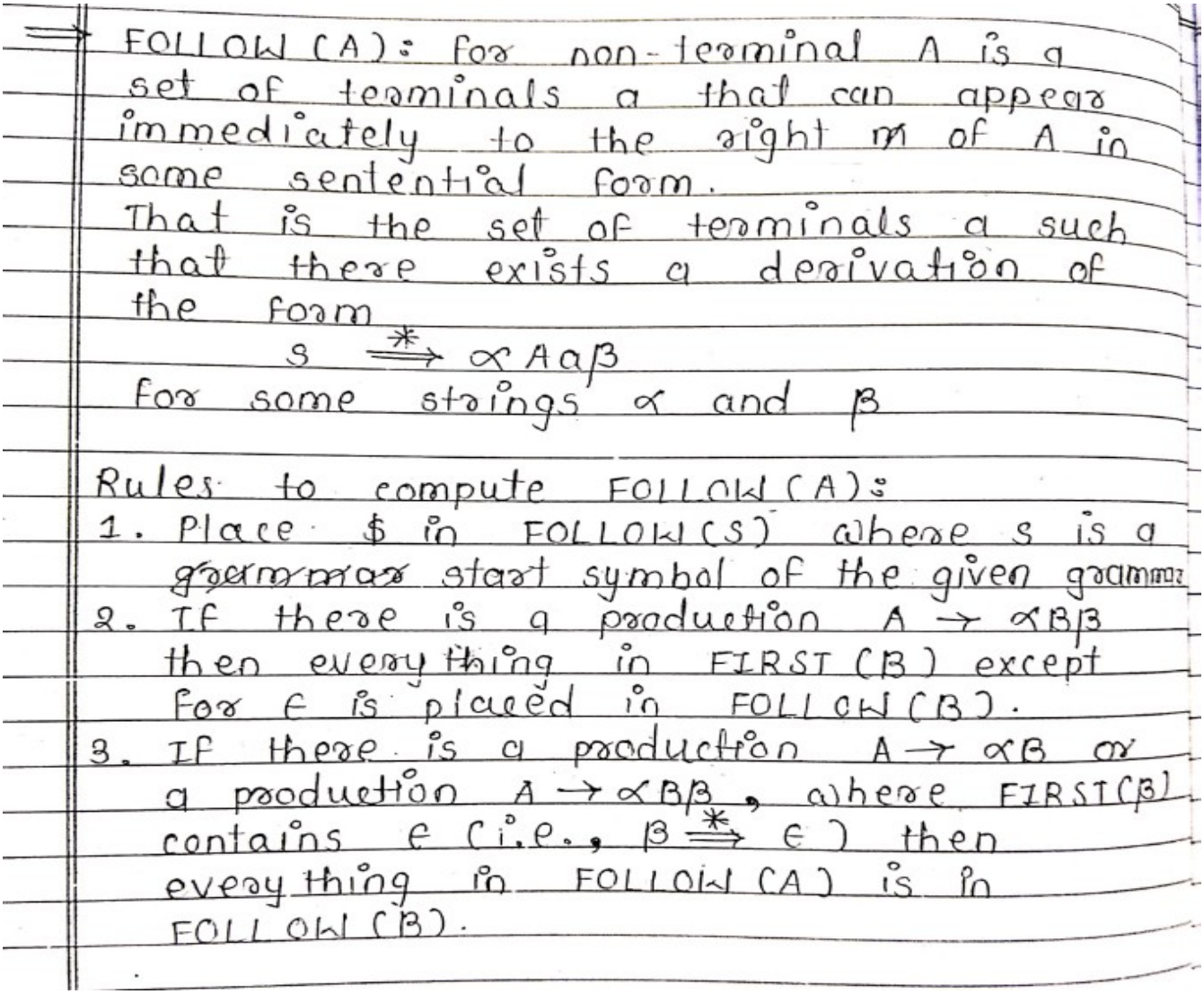
Now the parser checks for the second character of the input string which is b, and the Non-Terminal to derive is B, but the parser can't get any string derivable from B that contains b as first character. But the Grammar does contain a production rule B -> ε, if that is applied then B will vanish, and the parser gets the input “ab” , as shown below. But the parser can apply it only when it knows that the character that follows B is same as the current character in the input.

In RHS of A -> aBb, b follows Non-Terminal B, i.e. FOLLOW(B) = {b}, and the current input character read is also b. Hence the parser applies this rule. And it is able to get the string “ab” from the given grammar.

```
        A                       A
    /   |   \               /       \
   a    B    b     =>      a         b
        |
        ε
```

So FOLLOW can make a Non-terminal to vanish out if needed to generate the string from the parse tree.

The conclusions is, we need to find FIRST and FOLLOW sets for a given grammar, so that the parser can properly apply the needed rule at the correct position.



⇨ **Note**: that unlike the computation of FIRST sets for non-terminals, where the focus is on *what a non-terminal generates*, the computation of FOLLOW sets depends upon *where the non-terminal appears on the RHS of a production*.

- Handles and handle pruning
- Shift-reduce parsing
- Operator precedence parsing
- LR parsing
- Construction of SLR parsing tables