Syntax Analysis:

unit-3

* The syntax analysis phase is second phase in compilation.
* The syntax analyser (parser) basically checks for syntax of the language
* A syntax analyser takes the tokens from the lexical analyser and groups them in such a way that some programming structure (syntax) can be Recognized.
* After Grouping the tokens if at all, any syntax can not Recognized then syntatic error will be generated.
* This overall process is called syntax checking of the language.

Definition of Parser: A parsing (or) syntax analysis is a process which takes the input string 'w' and produces either Parse tree (syntatic structure) (or) generate the syntatic errors.

* The syntax of programming language constructs can be specified by context-free Grammars.

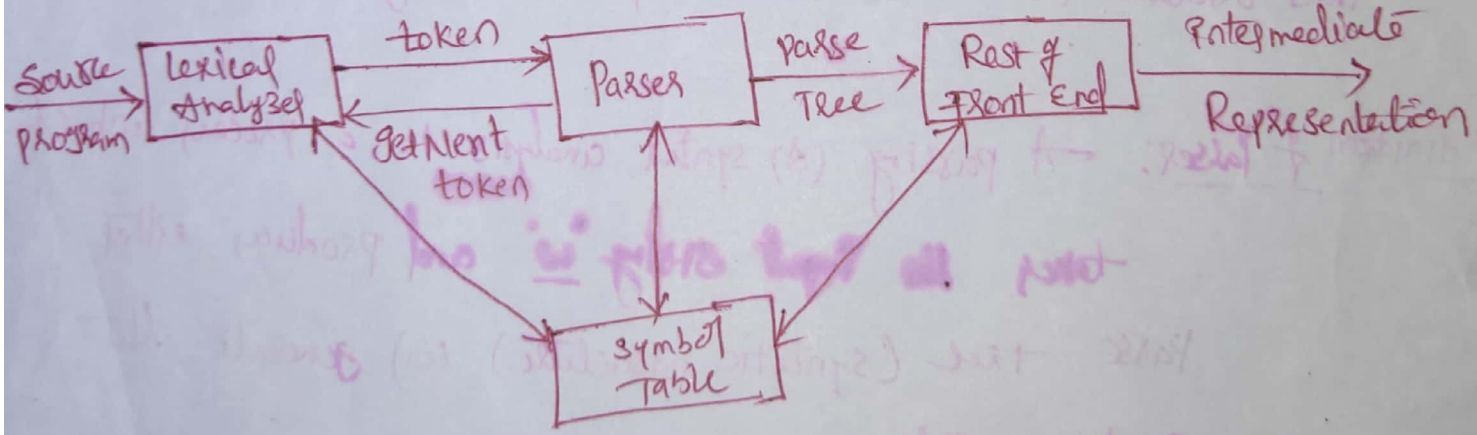
A Grammar gives a precise, yet easy to understand, syntactic specification of Programming language.

The Role of the Parser:-

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

* We expect the parser to Report any syntax errors in an intelligible fashion and to Recover from commonly occurring errors to continue processing the Remainder of the program.

* The parser constructs a parse Tree and passes it to the Rest of the compiler for further processing.



* The methods commonly used in compilers can be classified as being either top-down or bottom-up.

* top-down methods build parse Trees from the (top) top (root) to the bottom (leaves).

while bottom-up methods start from the leaves and work their way up to the Root..

* In either case, the input to the parser is scanned from left to Right, one symbol at a time.

* the most efficient top-down and bottom-up methods work only for sub classes of grammers, i.e, LL and LR grammers

left-to-Right Leftmost left Right most

LL - it is predective parsing approach, their parser are used for construct small class of grammars

LR - Parser for the large class of LR grammars.

* we assume that the output of the parser is some Representation of the parse Tree for the stream of tokens that comes from the lexical analyser.

* there are no of tasks that might conducted during parsing, such as collecting information about various token into the symbol Table, perform Type checking and other kinds of semantic analysis, and generating intermediate code.

28/06/18 f_3 - sec-4 - 3, 4, 5, 6, 7, 9, 10, 11, 26, 27, 28, 36, 40, 43, 47, 48, 52, 55, 56, 59, 61, 63, 64.

f_3-sec-3 - 3, 5, 6, 2, 8, 11, 14, 17, 27, 28, 29, 31, 34, 35, 37, 40, 42, 48, 48, 52, 53, 54, 55, 58, 59, 60, 61, 64, 63.

Context free Grammars:

Representative Grammars:-

* Associativity and precedence are captured in the following Grammar, which is similar to describing expressions, terms, and factors.
* 'E' Represents expression consisting of terms separated by + signs, T Represents terms consisting of factors separated by * signs, and F Represents factors that can be either parenthesized expressions or) identifiers.

$$E \rightarrow E+T \mid T$$

$$T \rightarrow T*F \mid F$$

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

* the above Grammar belongs to the class of LR grammars that are suitable for bottom-up parsing. this grammar can be adapted to handle additional operators and additional levels of precedence.
* However, it can not be used for top-down parsing because it is left Recursive.

the following non-left-Recursive variant of the expression grammar be used for top-down parsing.

Context free Grammar:-

* Grammars describe the syntax of programming language constructs like expressions and statements.

using a syntatic variable stmt to denote statements and variable expr to denote expressions, the production.

stmt → if (expr) stmt else stmt

* this section Reviews the definition of CFG and introduces terminology for talking about parsing.

The Formal Definition of CFG:-

Ex: The following Grammar defines simple arithmetic expressions in this terminal symbols are

id + - * / ( )

The non terminal symbols are expression, term and factor. of expr is the start symbol.

expr → expr + term
expr → expr - term
expr → term
term → term * factor
term → term / factor
term → factor
factor → (expr)

factor → id.

simple

E → E+T | E-T | T
T → T*F | T/F | F
F → (E) | id

CFG: A context free grammer (or) simple Grammer describes the programming language constructs. It consists of terminals, Non-terminals, a start symbol and productions. It is denoted by

$$G = \{V_n, V_t, P, S\}$$

left-lineal Grammar: A CFG is said to be left-lineal if all productions of the 'G' are of the form.

$A \to Aa$
$A \to a$

(or) $A \to Bw$
$A \to w$

Right-lineal Grammar: A CFG is said to be Right-lineal if all productions of the 'G' are of the form.

$A \to wB$
$A \to w.$

Ex: left lineal

$A \to Bw$
$A \to w$

} in this A & B are Non-terminal and w is a string of terminals including 'ε'.

$G = (\{S\}, \{a, b\}, P, S)$

where P, is given by

$S \to Sab \mid a$ is left-lineal

The sequence Generated by 'G' are

$S \Rightarrow Sab$
$\Rightarrow Sabab$
$\Rightarrow aabab$

Ex: Right linear: $A \to wB$, $A \to w$ } is a Right linear.

$S \to abs / a$ is a Right linear grammar

$S \Rightarrow abs$
$\Rightarrow ababs$
$\Rightarrow ababa$.

Ex:- obtain the CFG generating the set of all string of balanced parenthesis.

Sol Let the CFG be, $G = \{V_n, V_t, P, S\}$

$V_n = S$
$V_t = \{(,)\}$

(i.e. balanced parenthesis (), ()(), (()))

the productions P is given by

P: { $S \to SS$
$S \to (S)$
$S \to \varepsilon$

$S \Rightarrow SS$
$\Rightarrow (S)S$
$\Rightarrow ()SS$
$\Rightarrow ()(S)(S)$
$\Rightarrow ()(())(S)$
$\Rightarrow ()(())()$

---

obtain the language $L = a^n b^n$, where $n \geq 1$

P: { $S \to asb$
$S \to ab$
}

string = ab, aabb, aaabbb

Palindrome over {a,b}

$S \to asa / bsb$
$S \to a / b / \varepsilon$

string {abaaba}

---

even no of a's & b's

$S \to asbs$
$S \to bsas$
$S \to \varepsilon$

string aabb, abba, bbabaa...

Context-free Grammars: (CFG)

* A CFG is a set of Recursive rewriting Rules (or) productions used to generate patterns of strings.

* A CFG consists of the following components = (V_T, V_N, P, S)

  a set of terminal symbols, which are characters of the alphabet that appear in the strings generated by the grammar.

* a set of Non-terminal symbols, which are the ~~characters~~ placeholders for patterns of terminal symbols that can be generated by the non-terminals symbols.

* a set of productions, which are Rules for Replacing (or) Rewriting non terminal symbols (on the left side of the production) in a string with other terminal (or) terminal symbols (R.H.S)

* a start symbol, which a special non terminal symbol that appears in the initial string generated by the grammar

To Generate a string of terminal symbols from a CFG, we:

* Begin with a string consisting of the start symbol
* Apply one of the productions with the start symbol on the left hand side, Replacing the start symbol with the Right hand side of the production.

* Repeat the process of selecting non-terminal symbols in the string, and replacing them with the R.H.S of some corresponding production, until all Non-terminals have been replaced by terminal symbols.

A CFG for Arithmetic Expressions:

→ An example grammar that generates strings representing arithmetic expressions with the four operators +, −, *, /, and numbers as operands is:

1. <expression> → number.
2. <expr> → (<expr>) ← Nested structures
3. <expr> → <expr> + <expr>
4. <expr> → <expr> − <expr>
5. <expr> → <expr> * <expr>
6. <expr> → <expr> / <expr>

only non-terminal symbol in this grammar is <expr>, which is also the start symbol. the terminal symbols are {+, −, *, /, (,), number}

CFG Vs Regular Expressions:-

* CFG grammars are strictly more powerful than Regular expressions.

bcz

* Any language that can be generated using R.E can be generated by a CFG.

* There are languages that can be generated by CFG that can not be generated by any R.E.

(N, T, P, S) - quadruple

Ex:

The Grammar $(\{A\}, \{a,b,c\}, P, A)$, $P: A \to aA, A \to abc$

The Grammar $(\{S,a,b\}, \{a,b\}, P, S)$, $P: S \to aSa, S \to bSb, S \to \varepsilon$

The Grammar $(\{S,F\}, \{0,1\}, P, S)$, $P: S \to 00S \mid 11F, F \to 00F \mid \varepsilon$

Generation of Derivation Tree:-

A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a CFG.

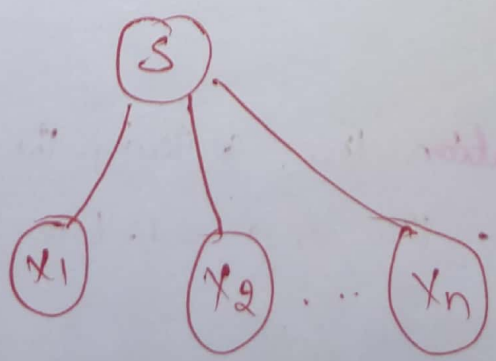
Representation Technique:

Root vertex - must be labeled by the start symbol.

vertex - labeled by a non-terminals symbol.

Leaves - labeled by a terminal symbol of Σ.

if $S \to X_1 X_2 \cdots X_n$ is a production Rule in a CFG, then the parse tree / derivation tree will be as follows



There are two different approaches to draw a derivation tree.

Top-down Approach:-

* Starts with the starting symbol 'S'
* Goes down to tree leaves using productions.

Bottom-up Approach:-

* starts from tree leaves
* Proceeds upward to the Root which is the starting symbol 'S'.

Sentential Form and Partial Derivation Tree:

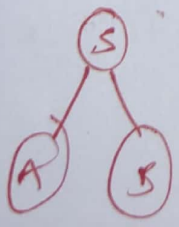
A Partial derivation tree is a sub-tree of a derivation tree/ parse tree such that either all of its children are in the sub-tree (or) none of them are in sub-tree.

Ex:

If any CFG the productions are

$S \rightarrow AB$, $A \rightarrow aaA \mid \varepsilon$, $B \rightarrow Bb \mid \varepsilon$

the partial derivation tree can be the following.



* If a partial derivation tree containing the Root S, it is called a sentential form. The above sub-tree is also in sentential form.

Left most derivation:- A left most derivation is obtained by applying production to the left most variable.

Right most derivation: A Right most derivation is obtained by applying production to the Right most variable.

Key points: during compilation, the parser uses the Grammar of the language to make a parse Tree (or) derivation tree out of the source code.

* The Grammar used must be unambiguous.
* An ambiguous Grammar must not be used for Parsing.

Based on no of strings it generates:

If CFG is generating finite number of strings, then CFG is Non-Recursive

Examples of Recursive and Non-recursive:-

Recursive Grammar:

$S \rightarrow SaS$

$S \rightarrow b$

the language (set of strings) generated by the above grammar is $\{b, bab, babab, \ldots\}$, which is Infinite.

2. $S \to Aa$
   $A \to Ab \mid c$

→ the language generated by the above grammar is: {ca, cba, cbba, ..} which is infinite.

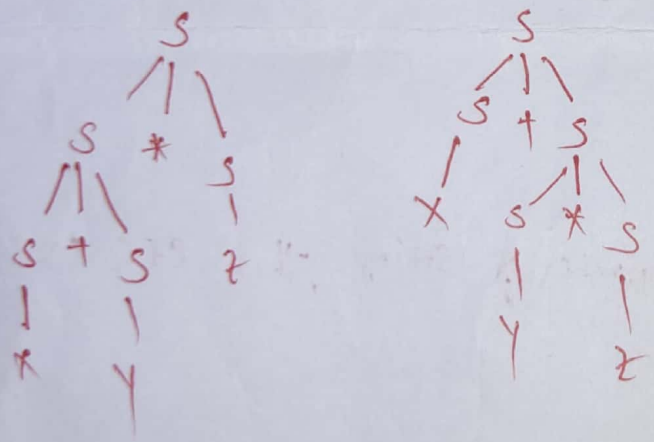
Non-Recursive Grammars:

$S \to Aa$
$A \to b \mid c$

→ the language generated by the above grammar is {ba, ca}, which is finite.

CFG: A CFG is a set of Recursive Rules used to generate patterns of strings.

"x+y*z"



Parse Tree: A CFG provide a structure to a string such structure assigns meaning to a string.

* Parse Trees are a successful data-structures to Represent and store such structures.

* A tree is a directed acyclic graph (DAG) where every node has at most one incoming edge.

* edge Relationship as Parent-child Relationship

* Every node has at most one parent, and zero or more children

* A grammar is called ambiguous if there is at least one string with two different leftmost (or) Rightmost derivations

* CFG is more powerful than finite automata (or) RE's, but still can not define all possible languages.

* CFG useful to nested structures ex:- parenthesis in programming languages.

Derivation:- we derive strings in the languages of a CFG by starting with the start symbol, and Repeatedly Replacing some variable A by the Right side of one of its productions.

Minimization of CFG:

* A Reduced grammar G' has the following properties

1, Each terminal and non-terminal that appears in G, must appear at least once in the derivation of some string accepted by L(G). (useless symbol)

2, It does not contain the null production i.e; $A \rightarrow \varepsilon$

3, It does not contain unit production i.e; $A \rightarrow B$

where A & B are Non-Terminals.

1, A symbol of G' is useful, when it appears on R.H.S of a production and can generate some terminal string. otherwise it is said to be useless symbol.

$S \rightarrow \alpha A \beta$

$\alpha A \beta \overset{*}{\Rightarrow} w$ (∵ A doesn't contain any productions)

∴ Here 'A' is said to be useless symbol.

Ex:

$S \rightarrow AB \mid C$

$A \rightarrow a$

$B \rightarrow b$

∴ Here, $S \rightarrow C$, C can not move further because there are no Rules (or) Production to C.

Then the Grammal become

$S \to AB$

$A \to a$

$B \to b$.

2, Then these can be Removed without changing the meaning of the Grammal.

Ex: $S \to aS \mid bS \mid \varepsilon$

the above grammal contains only one 'ε' production ie; $S \to \varepsilon$, it can be Removed by placing $S \to \varepsilon$ at the place of S.

later take place $S \to \varepsilon$ in $S \to aS$, we get $S \to a$ and

when we place $S \to \varepsilon$ in $S \to bS$, we get $S \to b$

After eliminating null productions the grammal becomes

$S \to aS \mid bS \mid a \mid b$.

3, $S \to AB$

$A \to a$

$B \to C$

$C \to D$

$D \to b$

Here the productions $B \to C$, $C \to D$ are unit productions we can eliminate them by $C \to D$ can be write as $C \to b$

$B \to C$ can be write as $B \to b$

the grammal becomes.

$S \to AB$

$A \to a$

$B \to b$

$C \to b$

$D \to S$.

Derivations:

* The construction of a parse tree can be made precise by taking a derivational view, in which productions are treated as rewriting rules.
* Begining with the start symbol, each rewriting step replaces a nonterminal by the body of one of its production.
* Each nonterminal is Replaced by the same body in the two derivations, but the order of Replacements is different.
* To understand how parser work, we shall consider derivations in which the nonterminal to be Replaced at each step.

Derivations are Two Types:

1, **Left most derivations:-** The left most non-terminal in α is Replaced the sentential way is $\alpha \underset{lm}{\Rightarrow} \beta$

2, **Right most derivations:-** The Right most nonterminal is always Replaced in α:

The sentential form is $\alpha \underset{rm}{\Rightarrow} \beta$

Example L.m.d of string abababa

$S \to SbS$
$S \to b$

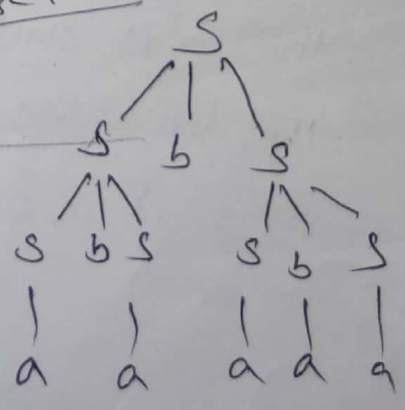
$S \underset{lm}{\Rightarrow} SbS$
$\underset{lm}{\Rightarrow} SbSbS$
$\underset{lm}{\Rightarrow} sbsbsbs$

$\underset{lm}{\Rightarrow} abSbSbS$
$\underset{lm}{\Rightarrow} ababSbS$
$\underset{lm}{\Rightarrow} ababaSb$
$\underset{lm}{\Rightarrow} abababa$

R.m.d:

$S \underset{rm}{\Rightarrow} SbS$
$\underset{rm}{\Rightarrow} SbSbS$
$\underset{rm}{\Rightarrow} SbSbE$
$\underset{rm}{\Rightarrow} SbSbSba.$

Parse Tree



Derivation Tree :- (or) Parse Tree

* A Parse Tree is a graphical Representation ; that filters out the order in which productions are applied to replace non terminals.

* Parse Tree has the following properties.

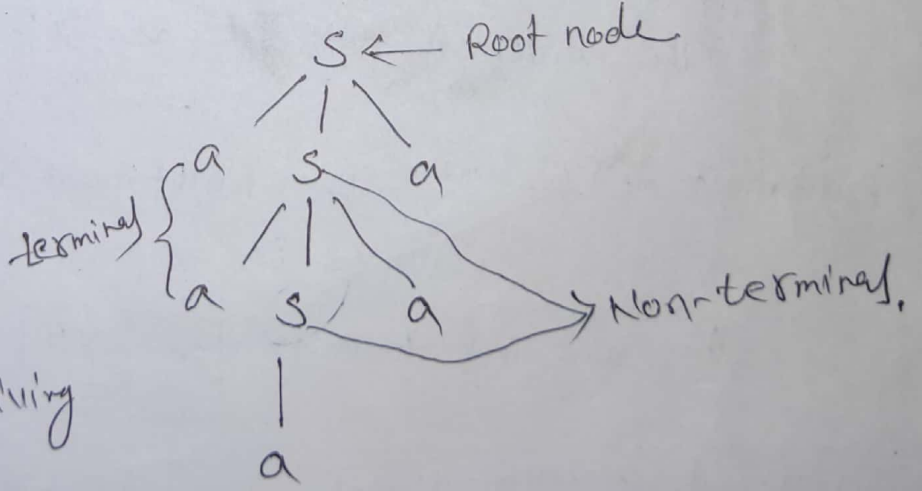
1, the Root node is always the start symbol. of G ie, S.

2, the leaf nodes are always the terminals a ie
$a \in (V_t \cup \{\varepsilon\})$

3, the interior nodes are always the non-terminals i.e.
$A_i, A_i \in V_n$.

4, if the childrens of a node $A \in V_n$ are $X_1 X_2 \cdots X_k$ the productions $A \to X_1 X_2 \cdots X_k$ is in P.

Ex:- $S \to aSa$
$S \to a$



Here 'S' is start symbol
the derivation for deriving string a^5, ie, aaaaa.

Ambiguity: A Grammar is said to be ambiguous, if it produces more than one parse tree some string generated by it.

* in other words, There is more than one left most derivation (or) more than one Rightmost derivation for given string.

An Ambiguous grammar contain productions of the form

$$S \rightarrow S \alpha S$$

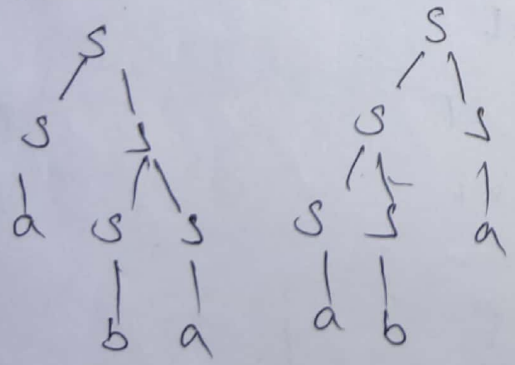
∴ where α may be string of terminals (or) Non-terminals i.e, non-terminals appears twice on R.H.S of a production.

Given Grammar

$S \rightarrow SS|a|b$

$S \Rightarrow SS$
$\Rightarrow SSS$
$\Rightarrow aSS$
$\Rightarrow abS$
$\Rightarrow aba.$

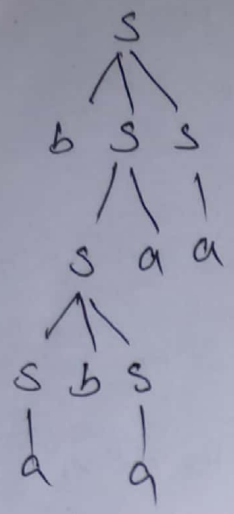
Now we construct derivation tree for string aba.

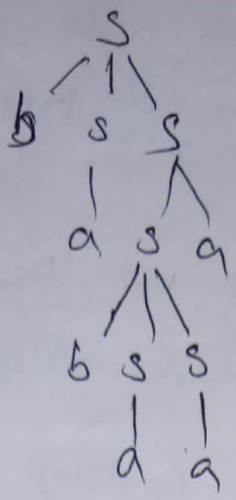


∴ Two derivation trees for the string **aba**.

what is meant by ambiguous test whether Following Grammar is Ambiguous or not

$S \rightarrow a|Sa|bSS|SSb|SSa$ $\quad w = babaa$





a grammar that produces more than one parse tree for some sentence is said to be ambiguous.

* another way, an ambiguous grammar is one that produces more than one Leftmost (or) more than one Rightmost derivations. For same sentence (or) string.

Ex: The Arithmetic expression grammar, permits two distinct leftmost derivations for sentence **id+id*id**

E → E+E / E-E / E-E / identifier / E*E

id+(id*id) — ①

(id+id)*id — ②

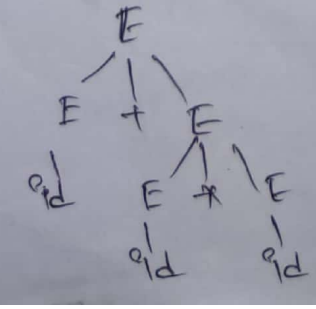
$E \xrightarrow[lm]{} E+E$

$\xrightarrow[lm]{}$ id+E

$\xrightarrow[lm]{}$ id+E*E

$\xrightarrow[lm]{}$ id+id*E

$\xrightarrow[lm]{}$ id+id*id



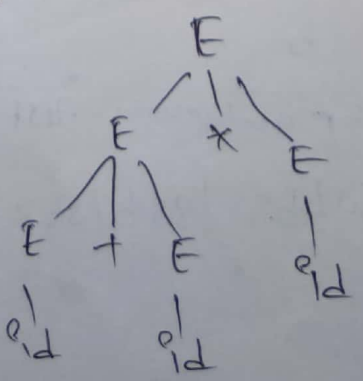
$E \xrightarrow[lm]{} E*E$

$\Rightarrow$ E+E*E

$\Rightarrow$ id+E*E

$\Rightarrow$ id+id*E

$\Rightarrow$ id+id*id



∴ Two parse trees for **id+id*id**

Consider the following grammar

$S \rightarrow 0A \mid 1B \mid 011$

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

$B \rightarrow 0A \mid 1S$

Construct left most derivation and parse tree for the following sentence

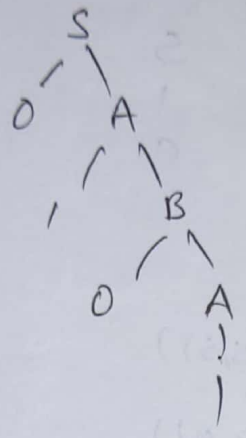
i) 0101 ii) 110010

$S \underset{lm}{\Rightarrow} 0A$

$\underset{lm}{\Rightarrow} 01B$

$\underset{lm}{\Rightarrow} 010A$

$\Rightarrow 0101$



$S \underset{lm}{\Rightarrow} 1B$

$\underset{lm}{\Rightarrow} 11S$

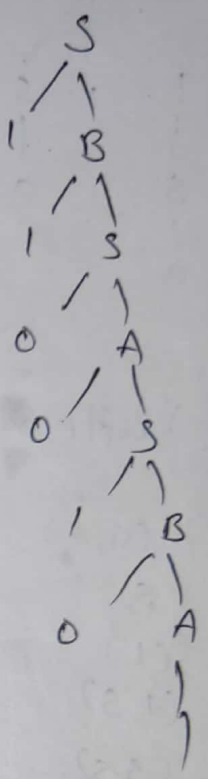
$\underset{lm}{\Rightarrow} 110A$

$\underset{lm}{\Rightarrow} 1100S$

$\underset{lm}{\Rightarrow} 11001B$

$\underset{lm}{\Rightarrow} 110010A$

$\underset{lm}{\Rightarrow} 1100101$



~~Prove that~~ Consider the grammar

$S \rightarrow (L) \mid a$

$L \rightarrow L,S \mid S$

a) what are the terminal, non-terminal and start symbol

b) find parse tree for the following sentence

i) (a, a)

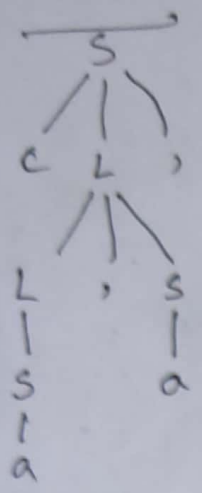
ii) (a, (a, a))

iii) (a, ((a, a), (a, a)))

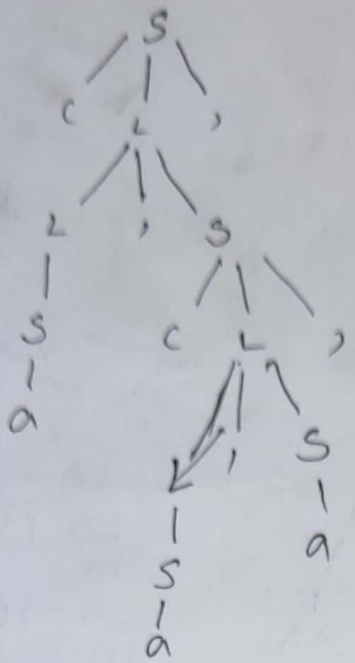
c) Construct left most derivation for each of the sentence in (b) & right most.

d) what language does the grammar generate.

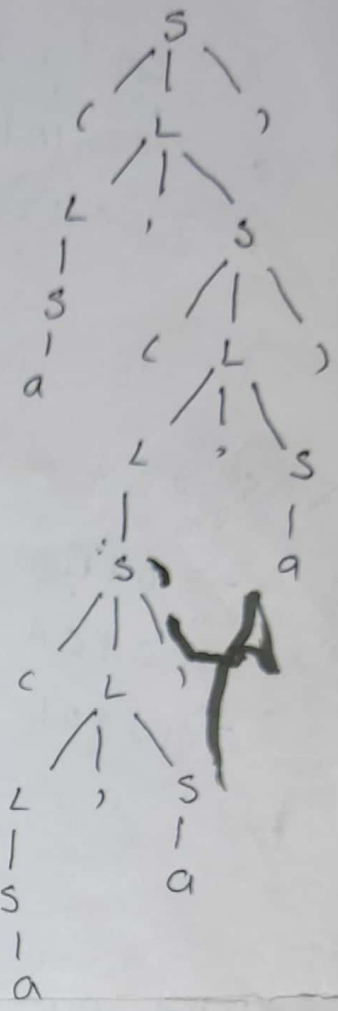
i) (a,a) is



ii) (a,(a,a)):



(a,((a,a),(a,a))):



c) Leftmost derivation:

i) (a,a)

S
(L)
(L,S)
(S,S)
(a,S)
(a,a)

ii) (a,(a,a))

S
(L)
(L,S)
(L,(L))
(S,(L))
(a,(L))
(a,(L,S))
(a,(S,S))
(a,(a,S))
(a,(a,a))

(a,(S,S))

(a,(a,a))

iii) (a,((a,a),(a,a)))

S
(L)
(L,S)
(S,S)
(a,S)
(a,(L))
(a,(L,S))
(a,(S,S))
(a,((L),S))
(a,((L,S),S))
(a,((S,S),S))
(a,((a,a),S))

(a,((a,a),(L)))
(a,((a,a),(L,S)))
(a,((a,a),(S,S)))
(a,((a,a),(a,S)))

d) Rightmost derivation

i) (a,a)

S
(L)
(L,S)
(L,a)
(S,a)
(a,a)

ii) (a,(a,a))

S
(L)
(L,S)
(L,(L))
(L,(L,a))
(L,(L,a))
(L,(S,a))
(L,(a,a))

(S,(a,a))
(a,(a,a))

iii) (a,((a,a),(a,a)))

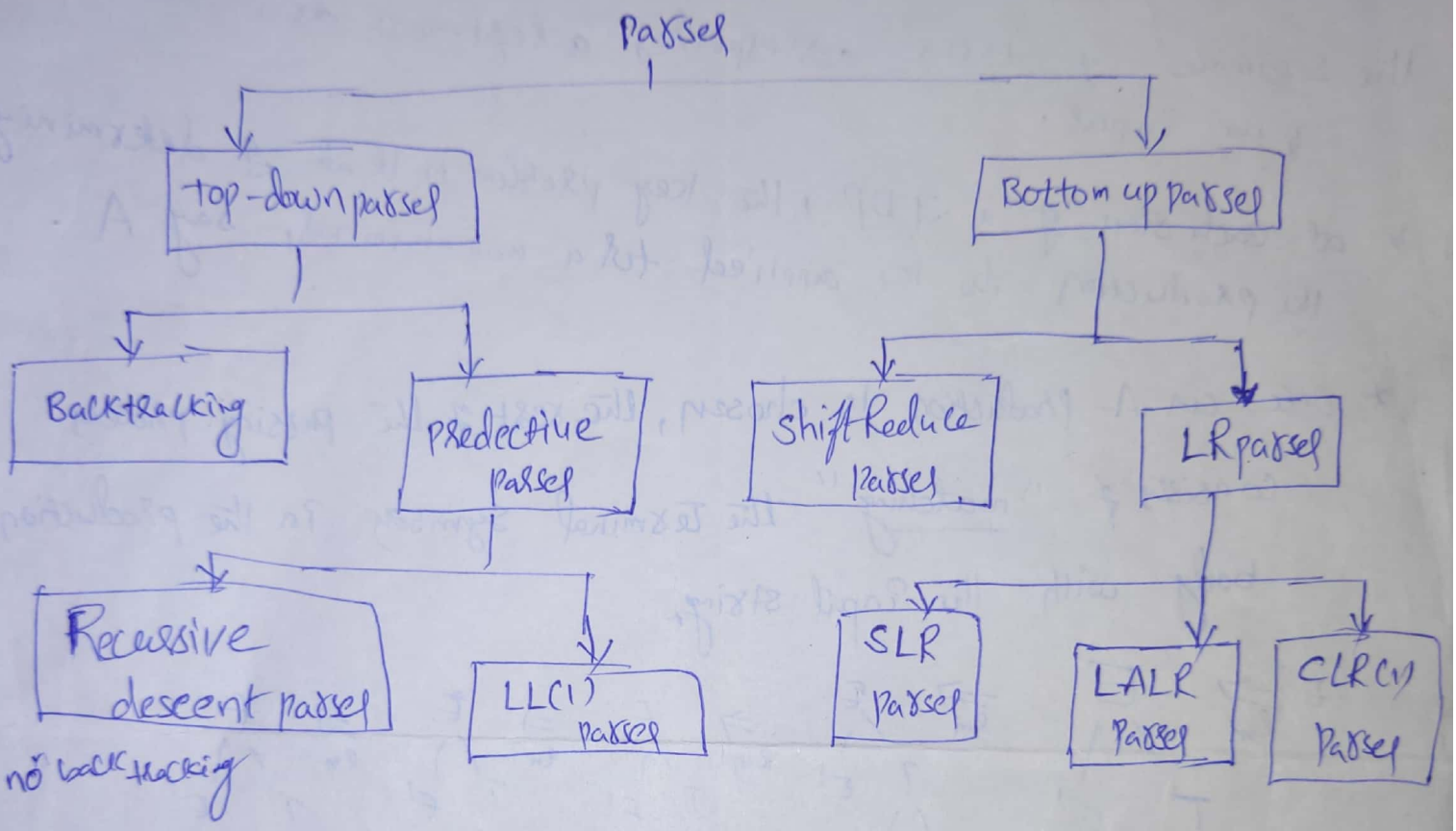
S
(L)
(L,S)
(L,(L))
(L,(L,S))
(L,(L,(L)))

(L,(L,(L,S)))
(L,(L,(L,a)))
(L,(L,(S,a)))
(L,(L,(a,a)))
(L,(S,(a,a)))

(L,((L),(a,a)))
(L,((L,S),(a,a)))
(L,((L,a),(a,a)))
(L,((S,a),(a,a)))
(L,((a,a),(a,a)))
(S,((a,a),(a,a)))
(a,((a,a),(a,a)))

Top-Down Parsing :-

* The Parser scans the input string from left to right and identifies that the derivation is leftmost (or) rightmost.



* Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the Root and creating the nodes of the parse tree in preorder (depth first way)

* Equivalently, top-down parsing can be viewed as finding a left most derivation for a parse input string.

Ex: The sequence of parse trees, for the input **id+id*id** is a top-down parse according to grammar

$E \to TE'$

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

$T \to FT'$

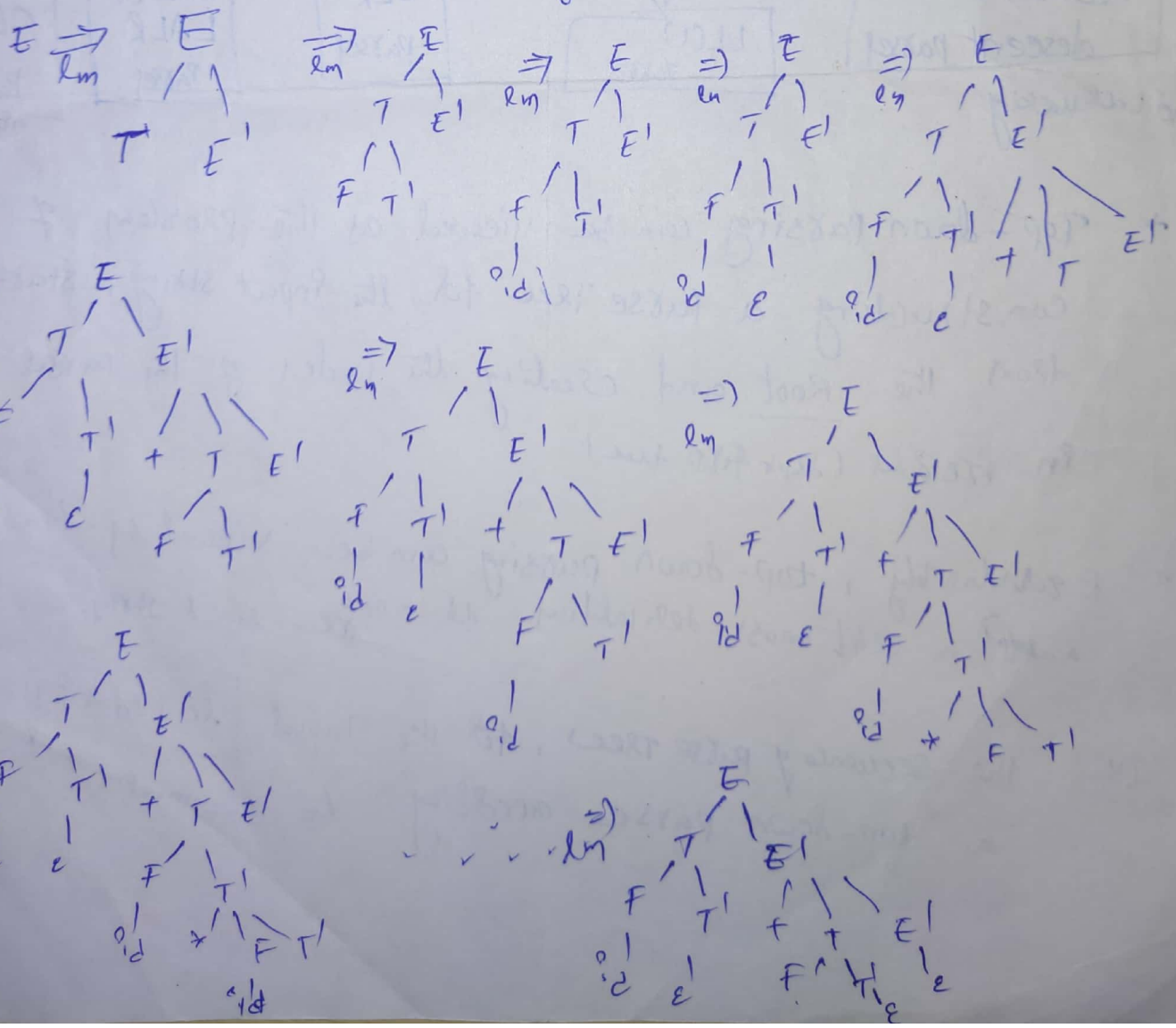
$T' \to *FT' \mid \varepsilon$

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

The sequence of trees corresponds a leftmost derivation of the input.

* At each step of a TDP, the key problem is that of determining the production to be applied for a non-terminal, say A.

* Once an A-production is chosen, the rest of the parsing process consists of "matching" the terminal symbols in the production body with the input string.



Back-Tracking: (Brute-force-method)

* Back-tracking is a technique in which for expansion of non-terminal symbol we choose one alternative and if some mismatch occurs then we try another alternatives if any.

* If for a non-terminal there are multiple production Rules begining with the same input symbol then to get the correct derivation we need to try all those alternatives secondly,

* In backtracking we need to move some levels upward in order to check the possibilities.

* this increases lot of overhead in implementation of parsing And hence it becomes necessary to eliminate the backtracking by modifying the grammar.

Consider the grammar:

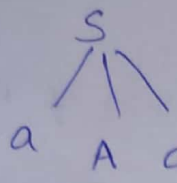
$S \rightarrow aAd \mid aB$

$A \rightarrow b \mid c$

$B \rightarrow ccd \mid ddc$

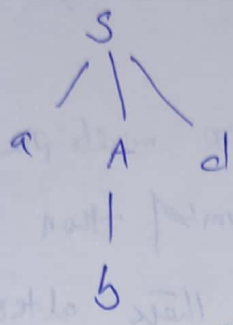
The following are the sequence of syntax trees generated during the parse tree of the string "accd" by using brute-force Parsing Technique.

select the first production of 'S', which yields



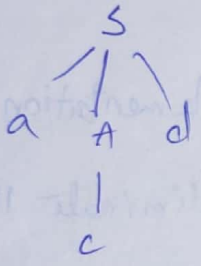
, symbol 'a' is matched with the string to be parsed.

→ next we choose the first production for 'A', we get



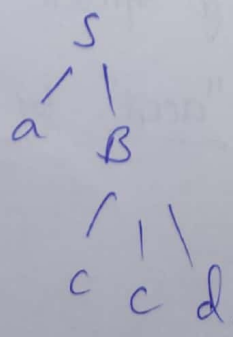
There is mismatch b/w the second symbol of 'c' of the input string and the second symbol 'b' in the sentential form abd.

At this point we must backup. the previous production application for A must be deleted and Replaced with next choice

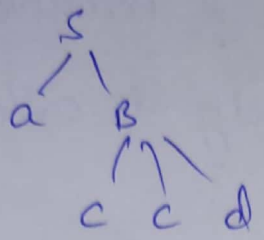


, Now the leftmost two characters of the given input string be matched.

when the third symbol 'c' of the input string is compared with the last symbol 'd' of the current sentential form, however, a mismatch again occurs. the previously choosen production for A must be deleted. since there are no more rules for A which can be selected, we must also delete the production for 'S'.



Next production of S is selected. This sequence of operations (5) yields the following parse tree.



in this Rule 'B' is applied. Now the Remaining I/p symbols are matched with Remaing symbols of sentential form which is the desired parse Tree.

* Because of this property, backTracking parsers are rarely used to parse the programming language construct.

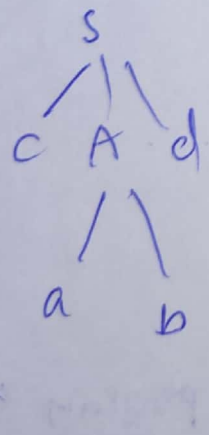
* The selection of a production for a nonterminal may involve "trial-and-error" i.e., we may have to try a production and backtrack to try another production if the first is found unsuitable

Ex:-2 Consider the grammar

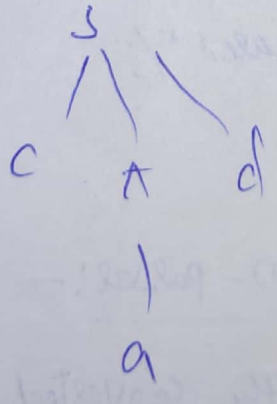
$S \rightarrow cAd$
$A \rightarrow ab \mid a$

$w = cad$

S1



X



Ex:-3

$A \rightarrow abC \mid aBd \mid cAD$

$B \rightarrow bB \mid \epsilon$

$C \rightarrow d \mid \epsilon$

$D \rightarrow a \mid b \mid \epsilon$

$w = acaba.$

Recursive Descent Parser: (RDP)

* A Recursive-descent program consists of a set of procedures, one for each nonterminal.

* Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string.

* A typical procedure for a nonterminal in a top-down parser. is

```
void A()
{
  choose an A-production, A → X1X2 ... Xk;
for (i = 1 to k)
 {
if (Xi is a nonterminal)
   call procedure Xi();
else if (Xi equals the current input symbol a)
        advance the input to the next symbol;
else /* an error has occurred */;
}
}
```

* In this type of parser the CFG is used to build the Recursive Routines.

* The R.H.S of the production Rule is directly converted to a program.

Basic steps for construction of RD parser:

* The R.H.S of the Rule is directly converted into program code symbol by symbol.

1. If the input symbol is non-terminal then a call to procedure corresponding to the non-terminal is made.
2. If the input symbol is terminal then it is matched with the lookahead from input. The lookahead pointer has to be advanced on matching of the input symbol.
3. If the production Rule has many alternates then all these alternate has to be combined into a single body of procedure.
4. The parser should be activated by a procedure corresponding to the start symbol.

Advantages of Recursive descent parser:

* Recursive descent parsers are simple to build
* Recursive descent parsers can be constructed with the help of parse Tree.

Limitations of Recursive descent parsers:

1. RDP are not very efficient as compared to other parsing Techniq
2. There are chances that the program for RDP may enter in an infinite loop for some input.

3. RDP can not provide good error messaging.

4. It is difficult to parse the string if lookahead symbol is arbitrarily long.

Construct Recursive descent parser for the following Grammar.

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) | id.$

ANS:

```
E()
{
    T();
    E'();
}
E'()
{
  if(l=='+')
   {
    match('+');
    T();
    E'();
   }
  else
    return;
}
```

```
T()
{
   F();
   T'();
}
T'()
{
  if(l=='*')
   {
    match('*');
    F();
    T'();
   }
  else
    return;
}
F()
 {
   if(l=='c')
    {
    match('c');
    E();
```

```
if(l==';')
 {
 match(';');
 }
}
else if(l=='id')
 {
 match('id');
 }
match(char t)
 {
 if(l==t)
 l=getchar();
 else
 printf("error");
 }
main()
 {
  E()
 if(l=='$');
 Printf("Parsing successfully completed");
 }
```

Ex!

$E \rightarrow T+E \mid T$

$T \rightarrow V*T \mid V$

$V \rightarrow id$

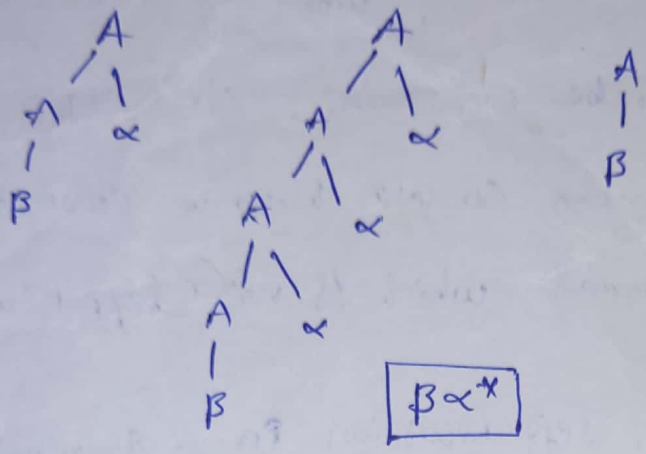
Ex!

$E \rightarrow num\ T$

$T \rightarrow * num\ T \mid \varepsilon$

Recursion: A function call it self, when we want to stop the Recursion simply we can Replace the terminal string 'β' by a non-terminal string.

Recursion like:



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

Recursion are Two Types:

1, left most Recursion (LR) $(A \to A\alpha \mid \beta)$

2, Right Recursion (RR) $(A \to \alpha A \mid \beta)$

left Recursion: If the left most symbol of R.H.S is equal to L.H.S then it's called left Recursion.

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

Right Recursion: If the Right most symbol of R.H.S is equal to L.H.S then it's called Right Recursion.

Left Recursion: Grammar $G = \langle V, T, P, S \rangle$ is said to be "left Recursion" if it has a non-terminal A such that there is a derivation.

$$A \xrightarrow{+} A\alpha \text{ , where } \alpha = \text{some string}$$

* Left Recursion has to be eliminated, since TDP can not handle left-Recursive Grammar, so we have to convert LRG Pn to an equivalent Grammar, which is not "left Recursive".

* In order to eliminate left Recursion Pn a grammar G, Consider the A-productions of the form

$$A \to A\alpha_1 | A\alpha_2 | \cdots | A\alpha_n | \beta_1 | \cdots | \beta_m$$

where β_i is do not start with A.

$$A \to \beta_1 A' | \beta_2 A' | \cdots | \beta_m A'$$
$$A' \to \alpha_1 A' | \alpha_2 A' | \cdots | \alpha_n A' | \varepsilon$$

$$A \to \beta A'$$
$$A' \to \alpha A' | \varepsilon$$

Consider the Grammar:

$$E \to E+T | T$$
$$T \to T*F | F$$
$$F \to (E) | a | b | id$$

$$E \to E+T | T$$

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

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

$T \rightarrow T*F \mid F$

$A = T,\ \alpha = *F,\ \beta = F$

$T \Rightarrow FT'$

$T' \Rightarrow *FT' \mid \varepsilon$

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

The Grammar becomes, after eliminating left Recursion

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow a \mid b \mid id$

2, Consider the Rule

$A \rightarrow ABd \mid Aa \mid a$

$B \rightarrow Be \mid b$

the Grammar becomes, after eliminating left Recursion

$A \rightarrow ABd \mid Aa \mid a$

$B \rightarrow Be \mid b$

$\Downarrow$

$A \rightarrow ABd \mid a$

$A \rightarrow Aa \mid a$

$\Leftrightarrow$

$A \Rightarrow aA'$

$A' \Rightarrow BdA' \mid \varepsilon \mid aA'$

$B \rightarrow bB'$

$B' \rightarrow eB' \mid \varepsilon$

Consider the Grammar,

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

There is no immediate left Recursion among the S-productions, so nothing happens during the loops. To eliminate the immediate left Recursion among the A-productions, we apply, we get

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid A'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Left Factoring.

In general if

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

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

$A \rightarrow \alpha A'$

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

Ex:1 Consider the following grammar

$S \rightarrow iEts \mid iEtSes \mid a$

$E \rightarrow b$

The left factored grammar becomes

$S \rightarrow iEtss' \mid a$

$s' \rightarrow es \mid \varepsilon$

$E \rightarrow b$

Ex:2 $A \rightarrow aAB \mid aA \mid a$

$B \rightarrow bB \mid b$

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

$A \rightarrow \alpha AB \mid aA \mid a$

$\alpha \; \beta_1 \; \alpha \; \beta_2 \; \alpha \; \beta_3$

$A \rightarrow \alpha A \quad A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2 \quad A' \rightarrow AB \mid A \mid \varepsilon$

similarly $B \rightarrow bB \mid b$ | $B \rightarrow bB'$

$\alpha \; \beta_1 \; \alpha \; \beta_2$ | $B' \rightarrow B \mid \varepsilon$

Left factoring:- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive (or) Top-down parsing. (5)

* Left factoring is used when it is not clear which of the 2- alternatives (or) productions used to expand the non-terminal

* By left factoring we may be able to re-write the productions. The productions are in the form of $A \to \alpha\beta_1 | \alpha\beta_2$

After eliminating the left factoring the productions are

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

* Consider the grammar $A \to aAB | aA | a$
$B \to bB | b$

eliminating the left-factoring.

$A \to aAB | aA | a\epsilon$

$A \to \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3$

$A = A$, $\alpha = a$, $\beta_1 = AB$, $\beta_2 = A$, $\beta_3 = \epsilon$ $\rightarrow$

$A \to aA'$
$A' \to AB | A$

Dangling else ambiguity: An else keyword always associates with the nearest preceding if keyword that does not cause syntax error.

Left factoring is Removing the common left factor that appears in two productions of the same non-terminal. it is done to avoid "back-tracking" by the parser. Suppose the parser has a look-ahead

* the process of factoring out the common prefixes of alternatives.

Left factoring is a grammar transformation technique. it consists in "factoring out" prefixes which are common to two or more productions.

Ex: $A \to \alpha\beta \mid \alpha\gamma$

to

$A \to \alpha A'$

$A' \to \beta \mid \gamma$

∵ left factoring is Resulted to eliminate non-determinism of a grammar

∵ it mean production should not non deterministic

3, we can see that, for every production, there is a common prefix and if we choose any production, here it is not contained that we will no need to backtrack.

2, it is non deterministic, because we can not choice any production and be assured that we will Reach at our desired string by making the correct parse tree. but if we write the grammar in a way that is deterministic and also leaves us to flexible enough to make it any string that may be possible without backTracking ie,

$A \to aA'$, $A' \to b_1 | b_2 | b_3$ now if we are asked to make the parse Tree for string ab_2 .. we don't need back tracking. because we can always choose the correct production when we get 'A' then we will generate the correct parse Tree.

The class of grammars for which we can construct predictive parsers looking 'k' symbols ahead in the input is sometimes called LL(k) class. We discuss the LL(1) class, it introduces certain computations, called FIRST and FOLLOW sets for a grammar, we shall construct "predictive parsing tables", which make explicit the choice of production during top-down parsing. These sets are also useful during bottom-up parsing.

FIRST and FOLLOW:

* The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a Grammar G.
* During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next i/p symbol.
* During panic-mode error Recovery, sets of tokens produced by FOLLOW can be used as synchronising tokens.

Define FIRST(α), where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α.

* if $\alpha \overset{*}{\Rightarrow} \varepsilon$, then ε is also in FIRST(α)

Define FOLLOW(A), for a nonterminal A, to be the set of terminals 'a' that can appear immediately to the Right of A in some sentential form. that is, the set of terminals 'a' such that there exists a derivation of the form $S \Rightarrow^* \alpha A a \beta$ for some α & β. b/w A and a.

but if so, they derived 'ε' and disappear.

* In addition, if A can be the Rightmost symbol in some sentential form, then $ is in FOLLOW(A): Recall the $ is a special "endmarker" symbol. that is assumed not be a symbol of any grammar.

TO compute FIRST(X) for all grammar symbols X, apply the following Rules until no more terminals (or) ε can be added to any FIRST set

1. If X is a terminal, then FIRST(X) = {X}
2. if X is a non-terminal and $X \to a\alpha$ is a production, then add to FIRST(X), if $X \to \epsilon$ is a production, then add ε to FIRST(X).

TO compute FOLLOW(A) for all non-terminals A, apply the following Rule until nothing can be added to any FOLLOW set

1) place $ in FOLLOW(S), where S is the start symbol, and $ is the input Right endmarker.

2, If there is a production $A \to \alpha B\beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B). (3)

3, if there is a production $A \to \alpha B$ or) a production $A \to \alpha B\beta$, where FIRST(β) containing ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

FIRST and FOLLOW sets for Grammars:

	FIRST	FOLLOW
$S \to ABCDE$	{a, b, c}	{$}
$A \to a/\epsilon$	{a, ε}	{$, b, c}
$B \to b/\epsilon$	{b, ε}	{$, c}
$C \to c$	{c}	{$, d, e, $}
$D \to d/\epsilon$	{d, ε}	{$, e}
$E \to e/\epsilon$	{e, ε}	{$}
$S \to Bb/Cd$	{a, b, c, d}	{$}
$B \to aB/\epsilon$	{a, ε}	{b}
$C \to cC/\epsilon$	{c, ε}	{d}

	FIRST	FOLLOW
$E \to TE'$	$\{id, (\}$	$\{\$,)\}$
$E' \to +TE'/\epsilon$	$\{+, \epsilon\}$	$\{\$,)\}$
$T \to FT'$	$\{id, (\}$	$\{+, \$,)\}$
$T' \to *FT'/\epsilon$	$\{*, \epsilon\}$	$\{+, \$,)\}$
$F \to id/(E)$	$\{id, (\}$	$\{*, +, \$,)\}$
$S \to ACB \mid CbB \mid Ba$	$\{d, g, h, \epsilon, b, a\}$	$\{\$\}$
$A \to da \mid BC$	$\{d, g, h, \epsilon\}$	$\{\$, h, g, b\}$
$B \to g \mid \epsilon$	$\{g, \epsilon\}$	$\{\$, a, h, g\}$
$C \to h \mid \epsilon$	$\{h, \epsilon\}$	$\{\$, g, b, h\}$
$S \to aABb$	$\{a\}$	$\{\$\}$
$A \to C/\epsilon$ (and)	$\{c, \epsilon\}$	$\{\$, d, b\}$
$B \to d \mid \epsilon$	$\{d, \epsilon\}$	$\{b\}$
$S \to aBDh$	$\{a\}$	$\{\$\}$
$B \to cC$	$\{c\}$	$\{\$, g, f, h\}$
$C \to bC \mid \epsilon$	$\{b, \epsilon\}$	$\{\$, g, f, h\}$
$D \to EF$	$\{g, f, \epsilon\}$	$\{\$, h\}$
$E \to g \mid \epsilon$	$\{g, \epsilon\}$	$\{\$, f, h\}$
$F \to f \mid \epsilon$	$\{f, \epsilon\}$	$\{\$, h\}$

S → AaAb | BbBa

A → ε

B → ε

FIRST

FIRST(S) = {ε, a, b}

FIRST(A) = {ε}

FIRST(B) = {ε}

Follow

Follow(S) = {$}

Follow(A) = {a, b}

Follow(B) = {a, b}

(6)

S → aAB | bA | ε

A → aAb | ε

B → bB | ε

FIRST(S) = {a, b, ε}

FIRST(A) = {a, ε}

FIRST(B) = {b, ε}

Follow(S) = {$}

Follow(A) = {$, b}

Follow(B) = {$, }

S → iCtSA | a

A → eS | ε

C → b

FIRST(S) = {i, a}

FIRST(A) = {e, ε}

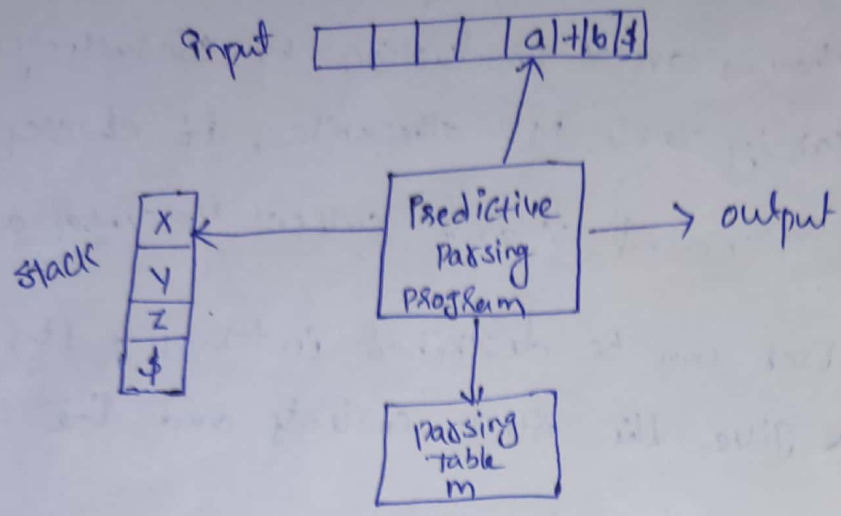
FIRST(C) = {b}

{$, e, }

{$, e}

{$, t}

Predictive parsing or) Table - Driven predictive parser:-



* A Non-Recursive predective parser can be built by maintaing a stack explicitly, rather than implicitly via Recursive calls.
* The parser mimics the left most derivation, if w is the input that has been matched so far, then the stack holds a seqence of grammer symbols α such that

$$S \xRightarrow[lm]{*} w\alpha$$

* The Table-driven parser has input buffer, a stack containing a sequence of grammer symbols, a parsing table constructed by Algorithm and an output stream.
* The input buffer contains the string to be parsed, followed by the end marker $. We reuse the symbol $ to mark the bottom of the stack, which initially contains the start symbol of the grammer on top of $.

The parser is controlled by a program that considers X, the symbol on top of the stack, and a, the current i/p symbol. if X is a non-terminal, the parser chooses an X-production by consulting entry M[X,a] of the parsing table M. otherwise, it checks for a match between the terminal X and current i/p symbol a.

** The behaviour of the parser can be described in terms of its configurations, which give the stack contents and the Remaining input.

Method: Initially, the parser is in a configuration with w$ in the input buffer and the start symbol S of G on top of the stack, above $. it uses predictive parsing table M to produce a predictive parse for the input.

```
set ip to point to the first symbol of w;
set X to the top stack symbol;
while (X ≠ $) /* stack is not empty */
{
if (X is a) pop the stack and advance ip;
else if (X is a terminal) error();
else if (M[X,a] is an error entry) error();
else if (M[X,a] = X → Y1 Y2 ... Yk) {
    output the production X → Y1 Y2 ... Yk;
    pop the stack;
    push Yk, Yk-1, ... Y1 onto the stack, with Y1 on top;
}
set X to the top stack symbol;
}
```

Construction of a predictive Parsing Table:-

I/P: Grammar G

O/P: Parsing table M.

Method: For each production $A \to \alpha$ of the Grammar, do the following

1. For each terminal a in FIRST(A), add $A \to \alpha$ to M[A, a]
2. if ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to M[A, b].
   - if ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to M[A, \$] as well.
   - if, after performing the above, there is a no production at all in M[A, a], then set M[A, a] to error (which normally Represent by an empty in the table).

$E \to E+T$

$T \to T*F$

$F \to (E) \mid id$

	FIRST	FOLLOW
$E \to TE'$	{id, (}	{\$,)}
$E' \to +TE' \mid \epsilon$	{+, ϵ}	{\$,)}
$T \to FT'$	{id, (}	{+,), \$}
$T' \to *FT' \mid \epsilon$	{*, ϵ}	{+,), \$}
$F \to id \mid (E)$	{id, (}	{+, *,), \$}

Non-Terminal	Input symbol.					
	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	$F \to id$			$F \to (E)$		

on Input id+id*id, the non-Recursive predictive parser Algorithm makes the sequence of moves, These moves correspond to a leftmost derivation

$$E \underset{lm}{\Rightarrow} TE' \underset{lm}{\Rightarrow} FT'E' \underset{lm}{\Rightarrow} idT'E' \underset{lm}{\Rightarrow} idE' \underset{lm}{\Rightarrow} id+TE' \underset{lm}{\Rightarrow} \dots$$

MATCHED	STACK	INPUT	ACTION
	E$	id+id*id$	
	TE'$	id+id*id$	output E→TE'
	FT'E'$	id+id*id$	output T→FT'
	idT'E'$	id+id*id$	output F→id
id	T'E'$	+id*id$	match id
id	E'$	+id*id$	output T'→ε
id	+TE'$	+id*id$	output E'→+TE'
id+	TE'$	id*id$	match +
id+	FT'E'$	id*id$	output T→FT'
id+	idT'E'$	id*id$	output F→id
id+id	T'E'$	*id$	match id
id+id	*FT'E'$	*id$	output T'→*FT'
id+id*	FT'E'$	id$	match *
id+id*	idT'E'$	id$	output F→id
id+id*id	T'E'$	$	match id
id+id*id	E'$	$	output T'→ε
id+id*id	$	$	output E'→ε

Error Recovery in Predictive Parsing:

* An error is detected during predictive parsing when the terminal on the stack does not match the next input symbol, or) when a nonterminal 'A' is on top of stack and 'a' is the next input symbol and the parsing table entry M[A, a] is empty, indicating an error.

Panic Mode Recovery:

* This is based on the idea of "skipping" the input symbol, until a token in a set of synchronizing tokens appear.
* The synchronizing sets should be chosen so that the parser Recovers Quickly from errors.
** At the starting, all the symbols in FOLLOW(A) are placed in the synchronizing set for non-terminal A. if we skip tokens until an element of FOLLOW(A) is seen and POP A from the stack, it is likely that parsing can continue.
* if a terminal on top of the stack does not match, a simple idea is to pop the terminal and issue a message saying that the terminal was inserted and continue parsing.

Consider the Parsing table for the Grammar:

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

using FOLLOW symbols on non-terminals as synchronising symbols, we have

synchronizing set = FOLLOW(E) = {), $ } = FOLLOW(E')

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

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

→ The Parsing table is shown in below, in which "synch" indicates the synchronising tokens obtained from the FOLLOW set of the nonterminals.

Nonterminal	Input symbol					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'			$E' \rightarrow +TE'$		$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$	synch	synch	$F \rightarrow (E)$	synch	synch

⇒ If the parser looks up an entry M[A, b] for which it is blank, then the input symbol 'b' is skipped.

* If the entry is "synch", then the non-terminal on top of the stack is popped so that the Parsing can continue.

* if a token on top of the stack does not match the input symbol, then the token is popped out of the stack.

The stack implementation for the erroneous input)id*+id and the error recovery mechanism is shown below.

STACK	Input	ACTION
E	)id*+id$	error, skip)
E$	id*+id$	
TE'$	id*+id$	E → TE'
FT'E'$	id*+id$	T → FT'
idT'E'$	id*+id$	F → id
T'E'$	*+id$	match id
*FT'E'$	*+id$	T' → *FT'
FT'E'$	+id$	match *
FT'E'$	+id$	error, M[F, +] = synch
T'E'$	+id$	F has been popped
E'$	+id$	T' → ε
+TE'$	+id$	E' → +TE'
TE'$	id$	match +
FT'E'$	id$	T → FT'
idT'E'$	id$	F → id
T'E'$	$	match id
E'$	$	T' → ε
$	$	E' → ε.