CIE-2

→ Sharan
1919102023

1. How can a lexical analyzer be constructed using lex tool?

→ By using regular expressions lex would be able to describe the tokens of one language more easily. This is part of a program that produce a automation table that supports finite languages to be parsing by lex compiler. If a token is found to have full values, it will be executed for outputs created based on user-specific parameters in C.

2. Write short notes on LEX.

→ Lex is a program that generates lexical analyzer. It is used with YACC parser generator. The lexical analyzer is a program that transforms an input stream into a sequence of tokens. It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

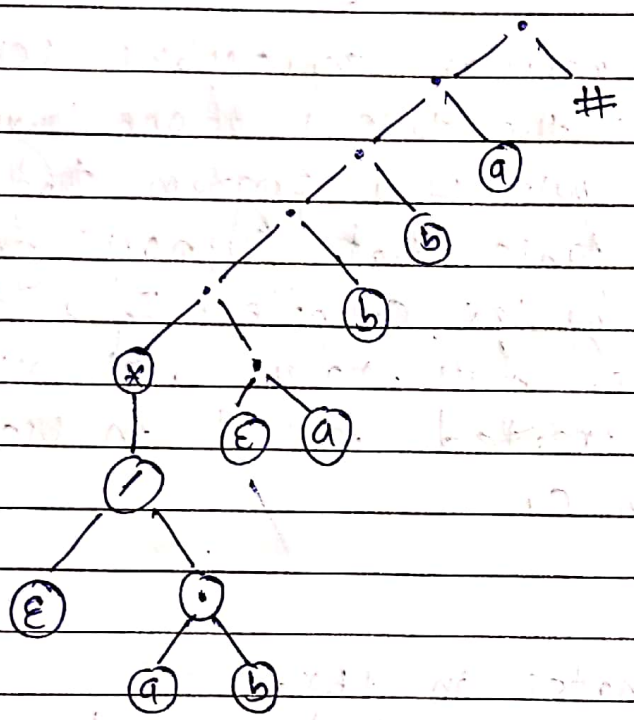
3. Draw the syntax tree for the augmented regular Expression.

$(\epsilon|ab)^*(\epsilon a)bba\#$

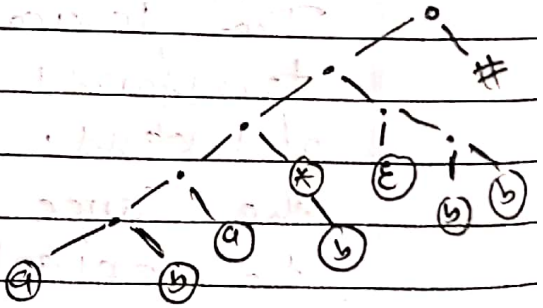
Soln

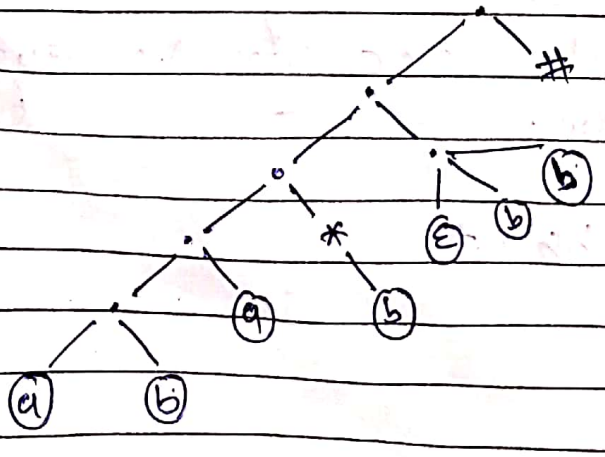
Syntax tree

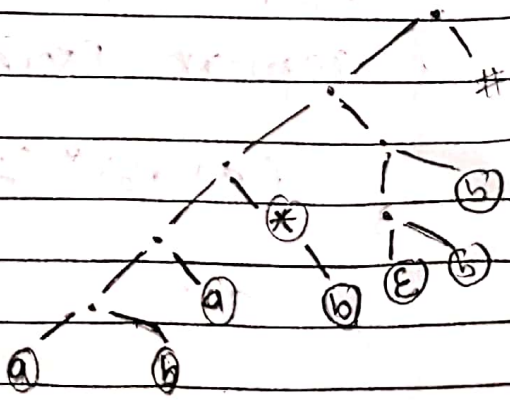
$(\varepsilon|ab)^*(\varepsilon a)bba\#$



4. Draw the syntax tree for the augmented expression. $(ab)ab^*(\varepsilon bb)\#$.







5. Write a context free grammar that generates the set of all strings of a's and b's which is a palindrome.

→ $S \rightarrow aSa$

$S \rightarrow bSb$

$S \rightarrow a$

$S \rightarrow b$

$S \rightarrow \varepsilon$

$\therefore S \rightarrow aSa \mid bSb \mid a \mid b \mid \varepsilon$

6. Mention the basic issues in parsing.

→ The two main basic issues in parsing are given below:-

(i) Specification of syntax.

(ii) Representation of input after parsing.

7. How will you eliminate left factor in a grammar?

→ Left factor is eliminated from a grammar by following below steps:-

(i) we make one production for each common prefixes.

(ii) The common prefix may be a terminal or a non-terminal or a combination of both.

(iii) Rest of the derivation is added by new productions.

8. What is the role of the parser in a compiler model?

→

(i) A parser obtains a string of tokens from the lexical analyzer and verifies the string can be generated by the grammar for the source language.

(ii) We expect the parser to report any syntax errors in an intelligible fashion.

(iii) It should also recover from commonly occurring errors so that it can continue processing the remainder of its input.

9. Draw transition diagrams for predictive parsers for the grammar.

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Soln

This grammar contains left recursion. After eliminating left recursion.

$E \rightarrow TE'$

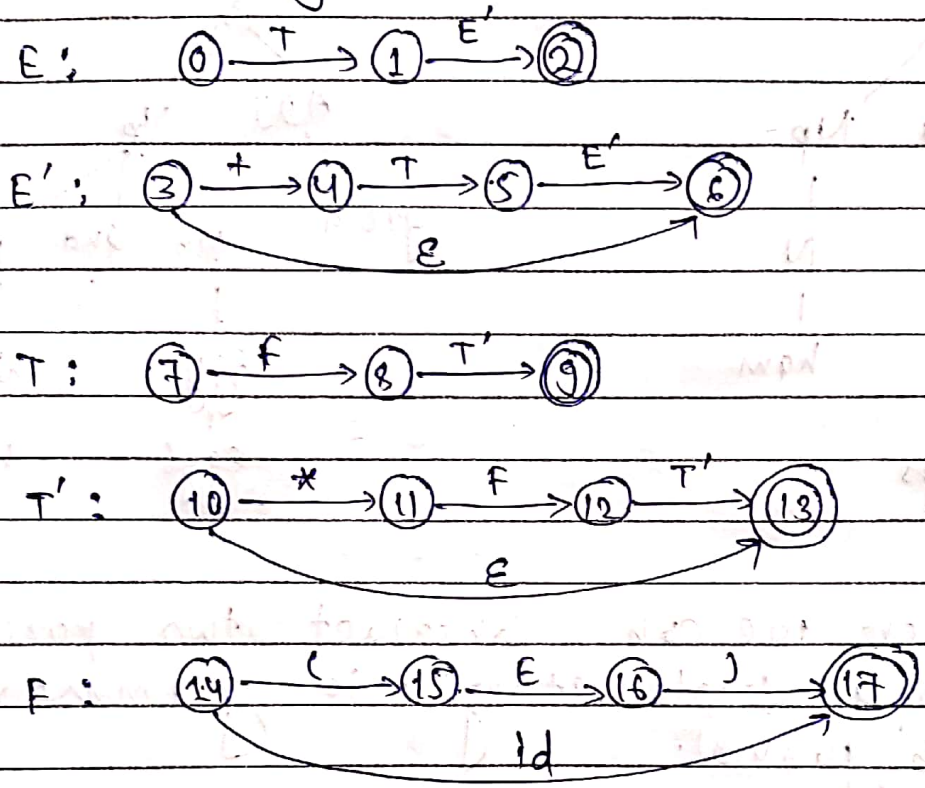
$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Transition diagram:

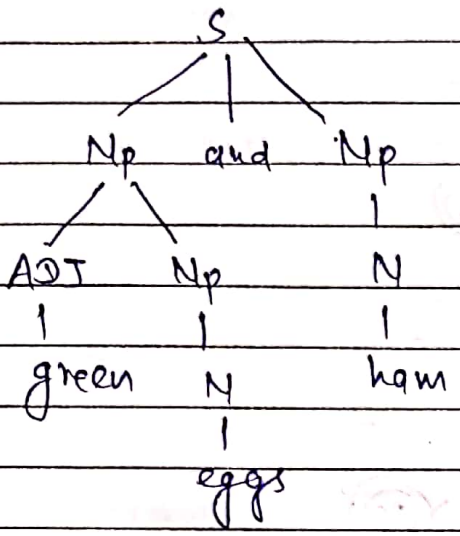


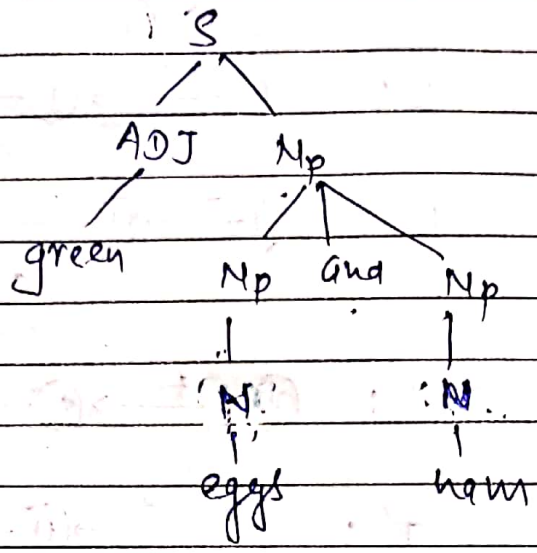
10. Consider the grammar $g = (V, T, P, S)$. Here $V = \{S, N, Np, ADJ\}$ and $T = \{$and, eggs, ham, pencil, green, cold, tasty$\}$. The set contains the following rules:

$Np \rightarrow Np$ and Np
$Np \rightarrow ADJ\ Np$
$Np \rightarrow N$

$N \rightarrow$ eggs | ham | pencil
$ADJ \rightarrow$ green | cold | tasty

Show that grammar is ambiguous by constructing two different parse trees for the sentence "green eggs and ham".





Since here we can construct two parse tree for a single string. So, grammar is a ambiguous.

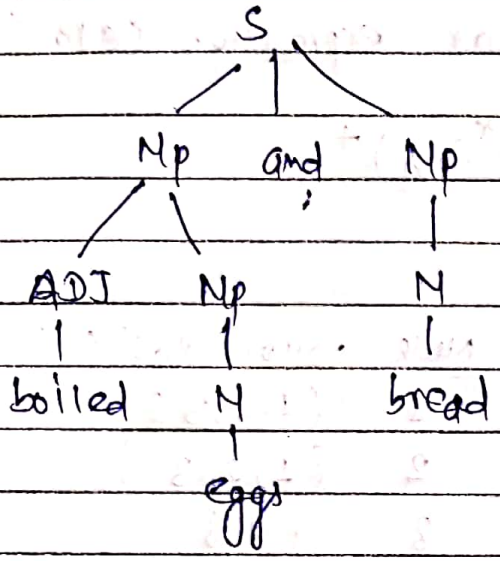
11. Consider the grammar $G = (V, T, P, S)$. Here $V = \{S, N, Np, ADJ\}$ and $T = \{$and, eggs, ham, pencil, green, cold, tasty$\}$. The set contains the following rules.

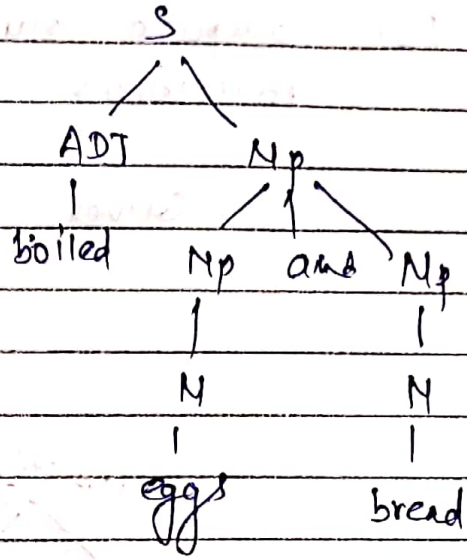
$Np \rightarrow Np$ and Np
$Np \rightarrow ADJ\ Np$
$Np \rightarrow N$

$N \rightarrow$ eggs | bread | pencil
$ADJ \rightarrow$ boiled | cold | tasty.

Is this grammar ambiguous for the sentence "boiled eggs and bread".

Sol^n





Since, we can construct two parse tree for a single string. So, grammar is a ambiguous.

12. Eliminate left recursion for the following grammar

$S \rightarrow aB \mid aC \mid Sd \mid Se$

$B \rightarrow bBc \mid f$

$C \rightarrow g$

Soln

$S \rightarrow aBS' \mid aCS'$

$S' \rightarrow dS' \mid eS' \mid \epsilon$

$B \rightarrow bBc \mid f$

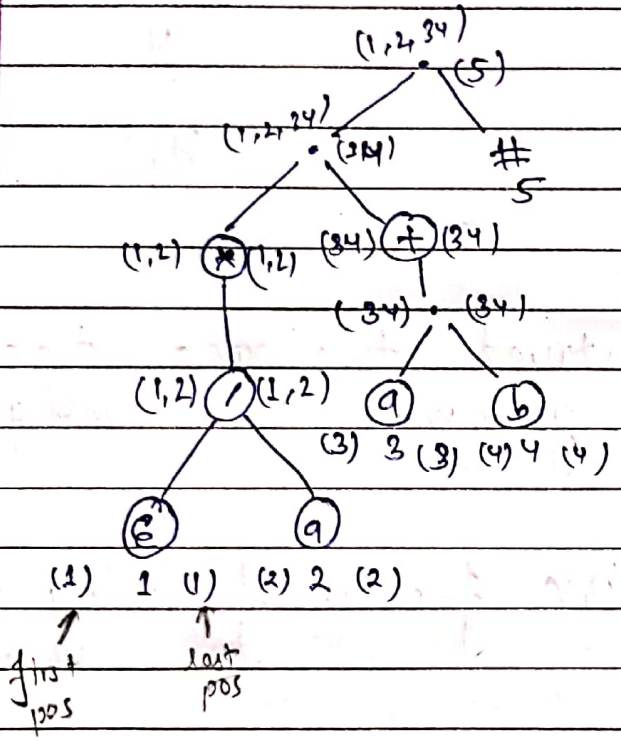
$C \rightarrow g.$

This is required grammar after eliminating left recursion.

13. Compute nullable (n), FIRSTPOS, LASTPOS and FOLLOWPOS for the regular expression $(\epsilon|a)^*(ab)^+$.

Soln:

Given, $R.E = (\epsilon|a)^*(ab)^+$



Node	Follow pos (i)
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{3,4}
5	ϕ

14. Write the rules for computing the FIRST and FOLLOW functions for constructing predictive parsing table.

⇒ The rules for computing the FIRST and FOLLOW functions for constructing predictive parsing table are given below:-

FIRST: functions

(1) If X is a terminal then FIRST(X) = {X}.

(ii) If $X \to \varepsilon$ is a production then add ε to FIRST(X).

(iii) If X is a non-terminal and $X \to Y_1, Y_2, \ldots Y_k$ is a production then add FIRST(Y_1) to FIRST(X). If $Y_1 \to \varepsilon$ then add FIRST(Y_2) to FIRST(X).

Follow function:

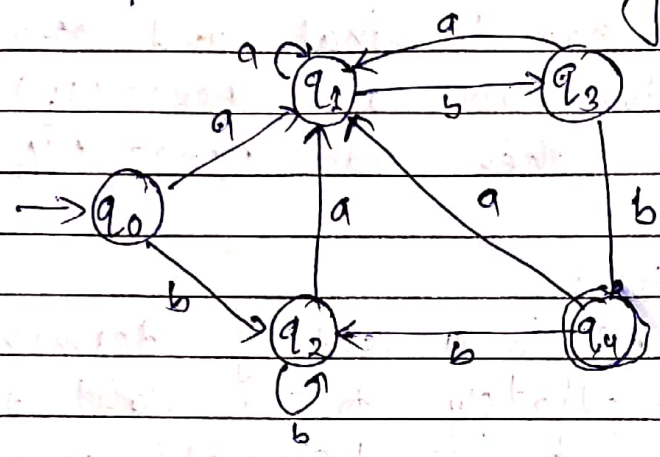
Follow(A) is a set of terminals α that appear immediately to the right of A. For right most sentencial form of A, $ will be placed in follow of A.

(i) placed $ in follow(S), where S is a start symbol and $ is the input right in marker.

(ii) If there is a production $A \to \alpha B \beta$ then, everything in first(B) except for ε is placed in follow(B).

(iii) If there is a production, $A \to \alpha B$ or a production $A \to \alpha B \beta$, where FIRST(B) contains ε then everything in follow(A) equal to follow(B).

15. What is the purpose of minimizing the states of DFA ? Minimize the given DFA.



Soln:

⇒ purpose

(i) Reduce the amount of space required to store the DFA.

(ii) Reduce the complexity of understanding how it works, and increase the performing speed of the program.

(iii) When we minimize the expression, we merge two or more states into a single equivalent state. Merging states like this should produce a smaller automation that accomplishes exactly the same task as our original state.

Transition table:

states	a	b
q_0	q_1	q_2
q_1	q_1	q_3
q_2	q_1	q_2
q_3	q_1	q_4
q_4	q_1	q_2

$\pi_0 = \{ Q_1^0, Q_2^0 \}$

where Q_1^0 is the set of all final states.

$Q_2 = Q - Q_1^0$, where Q is a set of all the states in DFA.

$Q_1^0 = f = \{q_4\} \rightarrow$ final state

$Q_2^0 = Q - Q_1^0 = \{q_0, q_1, q_2, q_3\}$

$\pi_0 = \{ \underbrace{\{q_4\}}_{1}, \underbrace{\{q_0, q_1, q_2, q_3\}}_{2} \}$

we cannot partition $\{q_4\}$ further so,

$Q_1^1 = \{q_4\}$

	q_0	q_1	q_2	q_3
a	2	2	2	2
b	2	2	2	1

$\pi_1 = \{ \underbrace{\{q_4\}}_{1}, \underbrace{\{q_0, q_1, q_2\}}_{3}, \underbrace{\{q_3\}}_{4} \}$

	q_0	q_1	q_2	q_2
a	3	3	3	3
b	3	4	3	1

$\therefore \pi_2 = \{ \underbrace{\{q_4\}}_{1}, \underbrace{\{q_0, q_2\}}_{5}, \underbrace{\{q_1\}}_{6}, \underbrace{\{q_3\}}_{4} \}$

	q_0	q_1	q_2	q_3
a	6	6	6	6
b	5	4	5	1

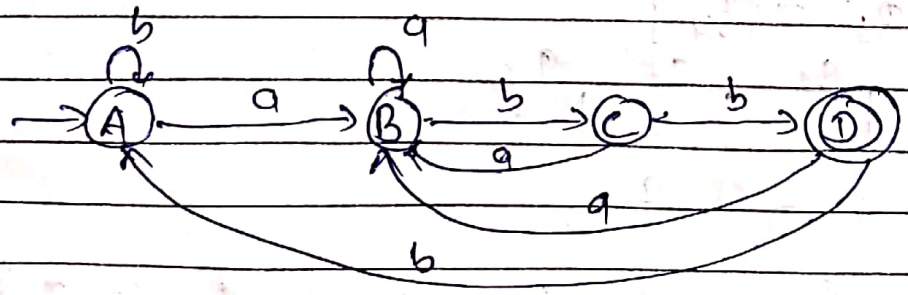
$\pi_3 = \{ \{q_4\}, \{q_0, q_2\}, \{q_1\}, \{q_3\} \}$

$\therefore \pi_2 = \pi_3$

Transition Table:

States	a	b
→ $\{q_0, q_2\}$ (A)	$\{q_1\}$ (B)	$\{q_0, q_2\}$ (A)
$\{q_1\}$ (B)	$\{q_2\}$ (B)	$\{q_3\}$ (C)
$\{q_3\}$ (C)	$\{q_1\}$ (B)	$\{q_4\}$ (D)
* $\{q_4\}$ (D)	$\{q_1\}$ (B)	$\{q_0, q_2\}$ (A)

Transition Diagram:



This is the required transition diagram.

16. Illustrate the heuristic techniques for error-recovery in predictive parsing.

⇒

An error is detected during predictive parsing,

(i) When the terminal on the top of stack doesn't match the next input symbol.

(ii) When non-terminal A is on the top of stack, a is the next input symbol, and the parsing table entry M[A,a] is empty.

Panic-mode error recovery:

Skipping input symbols on the input until a token in a selected set of synchronizing tokens appears.

Heuristics:

(i) Place all symbols in FOLLOW(A) into the synchronizing set for non-terminal A. -Skip tokens until an element of FOLLOW(A) is seen & pop A from the stack, i.e. parsing can continue.

(ii) If semicolons terminates statements then keywords that begin statements may not appear

in the follow set of the non-terminals generating expressions. Add to the synchronizing set of a lower constructs that begin higher constructs.

(iii) Add symbols in FIRST(A) to the synchronizing set for non-terminals A. It may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.

(iv) If a non-terminal can generate the empty string, then the production(s) deriving ε can be used as a default. Doing so may postpone the error detection, but cannot cause an error to be missed. Reduces the number of non-terminals that have to be considered during error recovery.

(v) If a terminal on the top of stack can't be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing.

This approach takes the synchronizing set of a token to consist all other tokens.

17. Distinguish between Context free grammar and regular expressions.

→

Regular Expressions	Context free grammar
(i) Lexical rules are quite simple in case of regular expressions.	(i) Lexical rules are difficult in case of context free grammar.
(ii) Notations in regular expressions are easy to understand.	(ii) Notations in context free grammar are quite complex.
(iii) A set of string is defined in case of regular expression.	(iii) In CFG, the language is defined by the collection of productions.
(iv) It cannot specify the recursive structure of the programming language.	(iv) It can specify the recursive structure of the programming language.
(v) It is less powerful than CFG.	(v) It is powerful than regular expression because they have more expressive power.
(vi) It only defines regular languages.	(vi) CFG can define both regular and irregular language.
(vii) It is use for describing the structure of lexical construct like identical keywords, constant etc.	(vii) It is use for used to generate patterns of strings.

18. Write an algorithm for eliminating left recursion from a grammar. Using the algorithm eliminate left recursion for the following grammar

$S \rightarrow Aa \mid b$. $A \rightarrow Ac \mid Sd \mid e$

⇒ Algorithm:

(i) If we have the grammar of the form $A \rightarrow A\alpha \mid \beta$. the rule for removing the left recursion is

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$.

This is the equivalent non-recursive grammar

(ii) Any immediate left-recursion can be eliminated by generalizing the above. Any production of the form,

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$ then

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

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

⇒ Given,

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid e$

Let us order the non terminals as S, A.

For S:

- Don't enter the inner loop
- Also there is no immediate left recursion to remove in S as outer loop says.

For A:

- Replace $A \rightarrow Sd$ with $A \rightarrow Aad \mid bd$
- Then we have, $A \rightarrow Ac \mid Aad \mid bd \mid f$

- Now remove the immediate left recursions.

$A \rightarrow bdA' \mid fA'$

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

So the resulting equivalent grammar with no left recursion is

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid fA'$

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

19. Show the stack implementation of non-recursive predictive parsing actions for the input id+id*id using the grammar given below

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Soln

Productions	first	follow
$E \rightarrow TE'$	{ (, id }	$,)
$E' \rightarrow +TE' \mid \varepsilon$	{ +, ε }	$,)
$T \rightarrow FT'$	{ (, id }	+,), $
$T' \rightarrow *FT' \mid \varepsilon$	{ *, ε }	+,), $
$F \rightarrow (E) \mid id$	{ (, id }	*, +,), $

The final table look like

Non Terminals	Input Symbols					
	id	*	+	(	)	$
E	E→TE'			E→TE'		
E'			E'→+TE'		E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→*FT'	T'→ε		T'→ε	T'→ε
F	F→id			F→(E)		

Stack operations:

Stack	Input	Production Applied
$E	id+id*id$	E→TE'
$E'T	id+id*id$	T→FT'
$E'T'F	id+id*id$	F→id
$E'T'id	id+id*id$	match id, pop
$E'T'	+id*id$	T'→ε
$E'	+id*id$	E'→+TE'
$E'T+	+id*id$	match +, pop
$E'T	id*id$	T→FT'
$E'T'F	id*id$	F→id
$E'T'id	id*id$	match id, pop
$E'T'	*id$	T'→*FT'
$E'T'F*	*id$	match *, pop
$E'T'F	id$	F→id
$E'T'id	id$	mach id, pop
$E'T'	$	T'→ε
$E'	$	E'→ε
$	$	Accept & Successful completion

20. Describe the language accepted by the following grammar.

$S \to AS \mid B$

$A \to aAc \mid Aa \mid \varepsilon$

$B \to bBb \mid \varepsilon$

is this grammar ambiguous? provide proof for ambiguity.

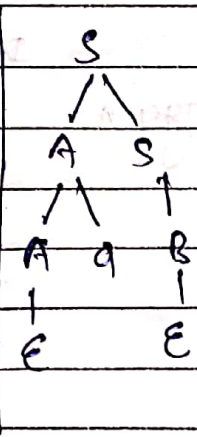
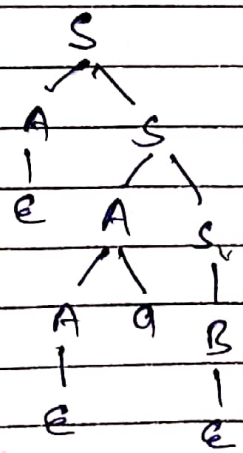
Soln:

⇒

$S \to AS$	$S \to AS$	$S \to B$
$\to aAc$	$\to aAc$	$\to bBb$
$\to aaAcc$	$\to aAac$	$\to bbBbb$
$\to aacc$	$\to aac$	$\to bbbb$
no of a = no of c	no. of a > no of c	i.e, even number's of b's.

∴ Strings of a & c with same or fewer c's than a's and no prefix has more c's than a's, followed by an even number of b's.

⇒

$S \to AS$	$S \to AS$		
$\to Aas$	$\to S$		
$\to as$	$\to AS$		
$\to aB$	$\to Aas$		
$\to a$	$\to as$		
	$\to aB$		
	$\to a$		

Hence, the given grammar is ambiguous.

21. Rewrite the following grammars so they can be parsed by a predictive parser by eliminating left recursion and applying left factoring where necessary

$S \rightarrow S+a \mid S+b \mid c$

⇒

Given $S \rightarrow S+a \mid S+b \mid c$

After eliminating left recursion.

$S \rightarrow cS'$

$S' \rightarrow +aS' \mid +bS' \mid \epsilon$

Now,

After eliminating left factoring.

$S \rightarrow cS'$

$S' \rightarrow +D \mid \epsilon$

$D \rightarrow aS' \mid bS'$

Again,

$S \rightarrow cS'$

$S' \rightarrow +D \mid \epsilon$

$D \rightarrow ES'$

$E \rightarrow a \mid b$

Hence, this is the req. grammar that can be parsed by a predictive parser

22. Consider the CFG G = {NT = {E, T, F}, T = {a, b, +, *}, P, E} with the set of production as follows.

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow F*$

$F \rightarrow a$

$F \rightarrow b$

Compute FIRST and FOLLOW functions.

Sol:

Given,

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow F* \mid a \mid b$

After eliminating left recursion.

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow aF' \mid bF'$

$F' \rightarrow *F' \mid \varepsilon$

After eliminating left factoring.

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow DF'$

$D \rightarrow a \mid b$

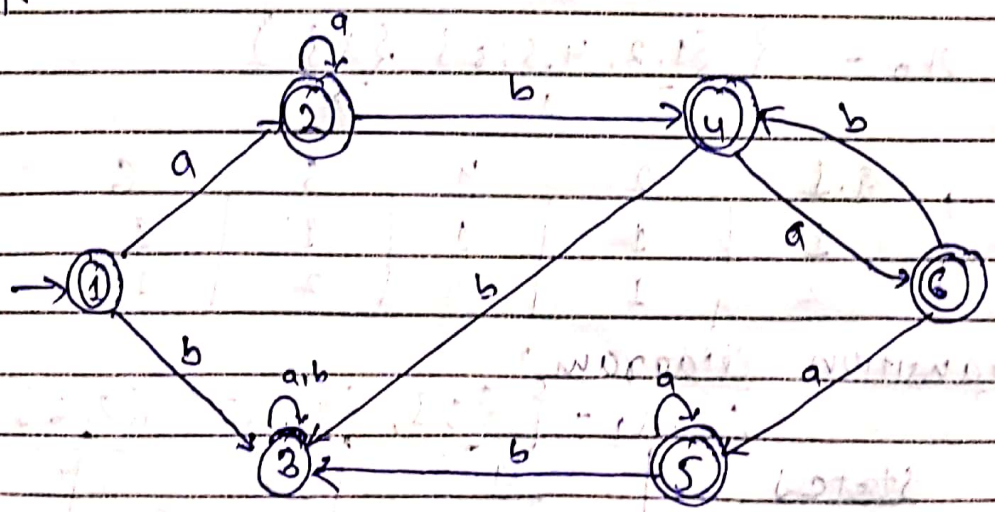
$F' \rightarrow *F' \mid \varepsilon$

First and Follow functions:

Productions	First	Follow
$E \rightarrow TE'$	$\{a, b\}$	$\{\$\}$
$E' \rightarrow +TE' \mid \epsilon$	$\{+, \epsilon\}$	$\{\$\}$
$T \rightarrow FT'$	$\{a, b\}$	$\{+, \$\}$
$T' \rightarrow *FT' \mid \epsilon$	$\{*, \epsilon\}$	$\{+, \$\}$
$F \rightarrow DF'$	$\{a, b\}$	$\{*, +, \$\}$
$D \rightarrow a \mid b$	$\{a, b\}$	$\{*, +, \$\}$
$f' \rightarrow *F' \mid \epsilon$	$\{*, \epsilon\}$	$\{*, +, \$\}$

Hence, this is the required FIRST and FOLLOW functions of a given grammar.

23. Minimize the number of states of the following DFA.



Soln

Transition Table

states	a	b
→ * 1	2	3
* 2	2	4
3	3	3
* 4	6	3
* 5	5	3
* 6	5	4

$\pi_0 = \{ q_1^0, q_2^0 \}$

where, q_1^0 is set of all final states

$q_2^0 = Q - q_1^0$, Q is a set of all the states in DFA

$q_1^0 = \{ 1, 2, 4, 5, 6 \}$

$q_2^0 = \{ 3 \}$

$\pi_0 = \{ \underbrace{\{1, 2, 4, 5, 6\}}_{1}, \underbrace{\{3\}}_{2} \}$

	1	2	4	5	6
a	1	1	1	1	1
b	2	1	2	2	1

$\therefore \pi_1 = \{ \underbrace{\{3\}}_{2}, \underbrace{\{1, 4, 5\}}_{3}, \underbrace{\{2, 6\}}_{4} \}$

Again;

	1	2	4	5	6
a	4	4	4	3	3
b	2	3	2	2	3

$\therefore \pi_2 = \{ \underset{2}{\{3\}}, \underset{5}{\{1,4\}}, \underset{6}{\{2\}}, \underset{7}{\{5\}}, \underset{8}{\{6\}} \}$

	1	2	4	5	6
a	6	6	8	7	7
b	2	5	2	2	5

$\therefore \pi_3 = \{ \underset{2}{\{3\}}, \underset{9}{\{1\}}, \underset{6}{\{2\}}, \underset{10}{\{4\}}, \underset{7}{\{5\}}, \underset{8}{\{6\}} \}$

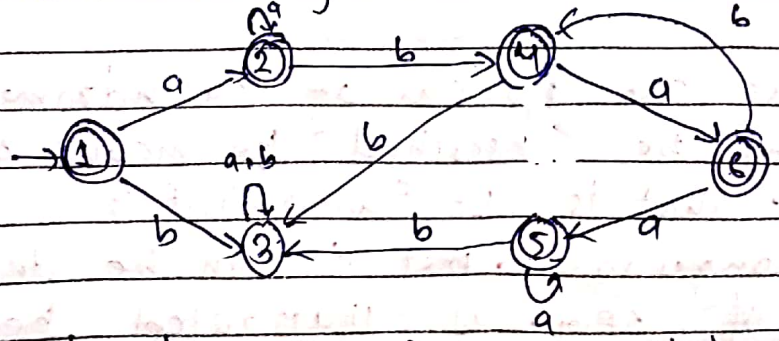
	1	2	4	5	6
a	6	6	8	7	7
b	2	10	2	2	10

$\therefore \pi_4 = \{ \{3\}, \{1\}, \{2\}, \{4\}, \{5\}, \{6\} \}$

Transition Table

state	input a	input b
* → 1	2	3
* 2	2	4
3	3	3
* 4	6	3
* 5	5	3
* 6	5	4

Transition Diagram:



This is the required minimized DFA.

24. To verify the syntax of programming language constructs, compilers use CFG. why not Regular expression ? Justify with examples.

⇒ Regular expressions are most useful for describing the structure of lexical construct such as identifiers, constant etc. But the lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expression cannot check balancing tokens, such as parenthesis, if else etc.

Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

CFG, on the other hand, is a superset of Regular Grammar.

It implies that every Regular Grammar is also context-free, but there exists some problems, which are beyond the scope of Regular Grammar. CFG is a helpful tool in describing the syntax of programming languages.

Example:

Let take the problem of palindrome language, which cannot be described by means of Regular Expression. That is, $L = \{w | w = w^R\}$ is not a regular language. But it can be described by means of CFG, as illustrated below:-

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

where,

$V = \{S, Z, N\}$

$\Sigma = \{0, 1\}$

$P = \{S \to Z \mid S \to N \mid S \to \varepsilon \mid Z \to 0S0 \mid N \to 1S1\}$

This grammar describes palindrome languages, such as: 1001, 11100111, 11111 etc.

25. Is it possible to directly obtain a DFA from a regular expression? How is it different from obtaining DFA from an intermediate NFA? obtain DFA directly for the regular expression. $(a|b)^+c(c|d)$.

Sol:

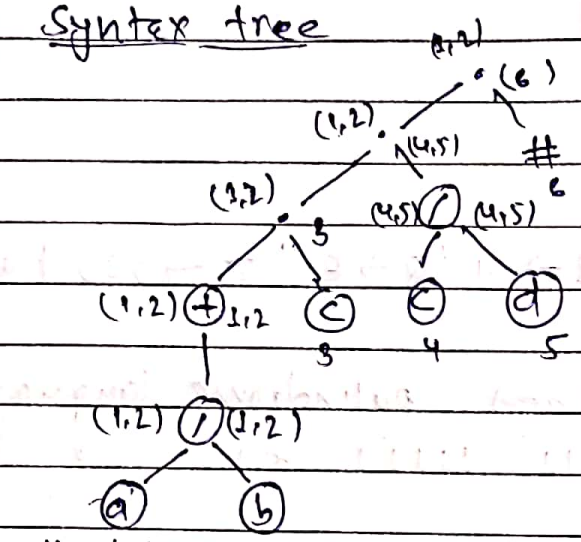
→ Yes, it is possible to directly obtain a DFA from a regular expression.

In the case of direct method, we are converting the regular expression to syntax tree and then obtain the first and follow function to construct a DFA. But in the case of NFA conversion, we are going through the constructing the NFA transition, removing epsilon and then we obtain the DFA.

→ Given,

$R.E = (a|b)^+c(c|d)$.

Syntax tree



Node	Follow (i)
1	{1, 2, 3}
2	{1, 2, 3}
3	{4, 5}
4	{6}
5	{6}
6	ϕ

Now, we construct DFA for the given regular expression. The start states of DFA is firstpos of root.

firstpos of root = {1, 2}

let the set be A.

A = {1, 2}

Consider the symbol 'a' on set A.

position 1 has a transition.

followpos(1)

⇒ {1, 2, 3}

⇒ B

Consider the input symbol 'b' on set A.

position 2 has b transition.

followpos(2)

⇒ {1, 2, 3}

⇒ B

Now, set A has no position left for c & d. so, on input c & d, it will go to dead state (z).

D.Tran [A, a] = B
D. Tran [A, b] = B
D. Tran [A, c] = Zd
D. Trans [A, d] = Zd

Again.

Consider the symbol 'a' on set B.
position 1 has a a transition.

followpos(1)
⇒ B.

Consider the symbol 'b' on set B.

followpos(2)
⇒ B

Consider the symbol 'c' on set B.

followpos(3)
⇒ {4, 5}
⇒ C

D. Tran [B, a] = B
D. Tran [B, b] = B
D. Tran [B, c] = C
D. Tran [B, d] = Zd

Again,

Consider the symbol 'c' on set C.

followpos(4)
⇒ {6}
⇒ D

Consider the symbol d on set C

followpos(5)
⇒ {6} = D

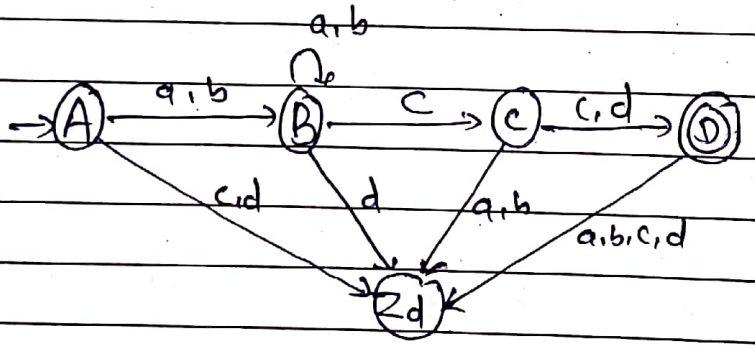
D. Tran [C, a] = Zd
D. Tran [C, b] = Zd
D. Tran [C, c] = D
D. Tran [C, d] = D

∴

Note:

∴ On input any symbol on set D, it will go to dead state (Zd).

DFA Diagram:



↑
Dead State
or
Trap state

Hence, this is the required DFA for given regular expression.

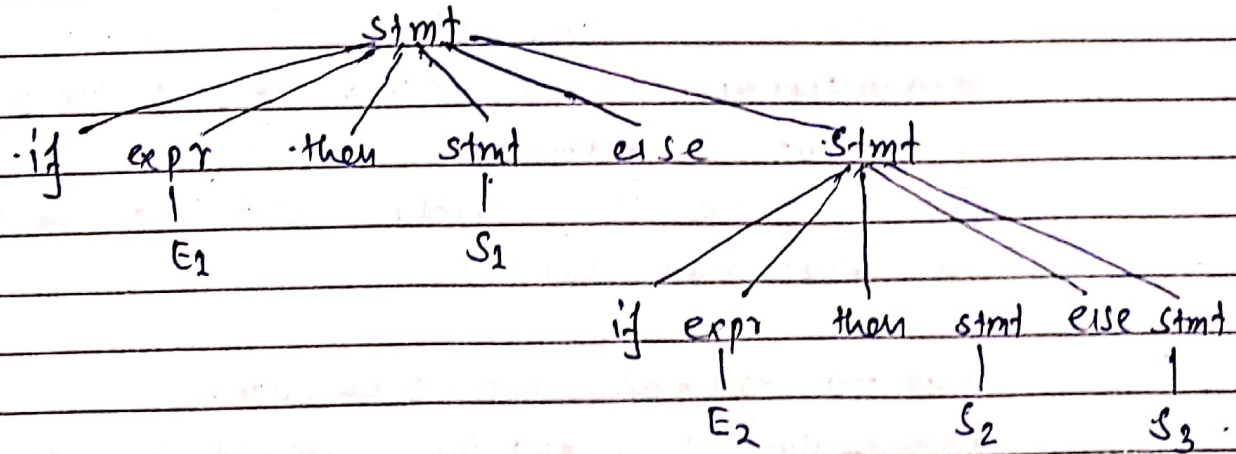
26. How do we eliminate ambiguity from the dangling the grammar?

⇒ Consider the grammar is ambiguous and ambiguity can be resolved. The given grammar is ambiguous and ambiguity can be resolved.

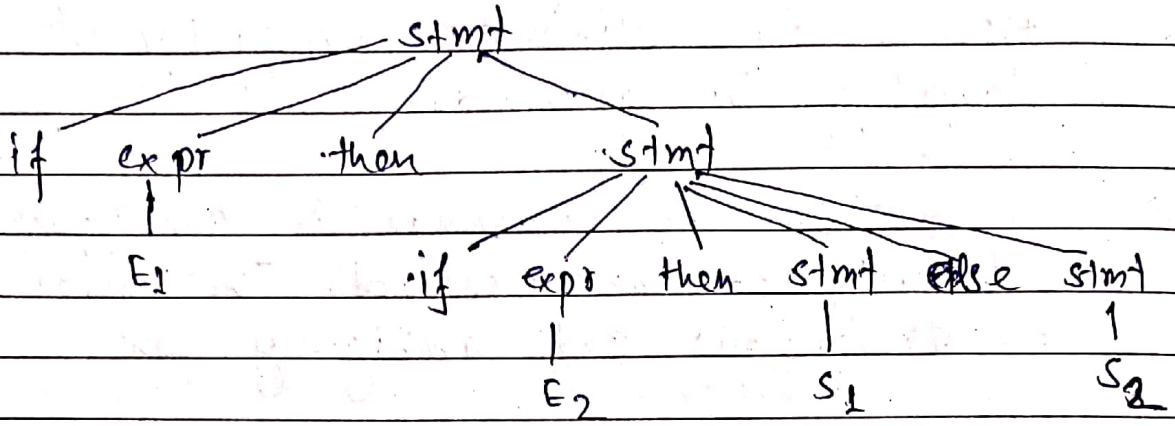
stmt → stmt → if expr then stmt
| if expr then stmt else stmt
| other

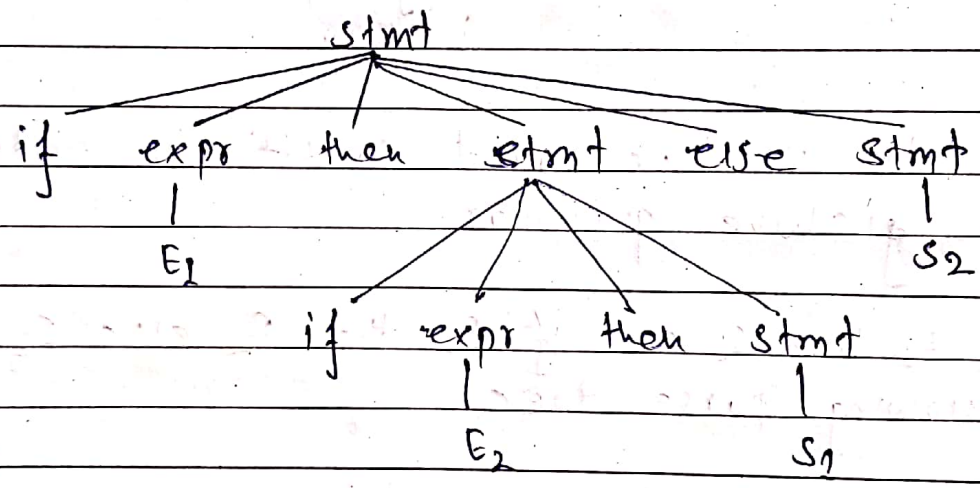
consider the compound conditional statement for the above grammar.

If E_1 then S_1 else if E_2 then S_2 else S_3 has the following parse tree.



well this is ambiguous due to the statement
If E_1 then if E_2 then S_1 else S_2
The two parse tree are





Ambiguity can be elimated as follows for unambiguous grammar.

In practice it is rarely built into the productions.

The grammar will be

Stmt → matched_stmt | open_stmt

matched_stmt → if expr then matched_stmt else matched_stmt
| other

open_stmt → if expr then stmt
| if expr then matched_stmt else open_stmt.

27. Explain the structure of lex program. write lex specifications for a desk calculator application.

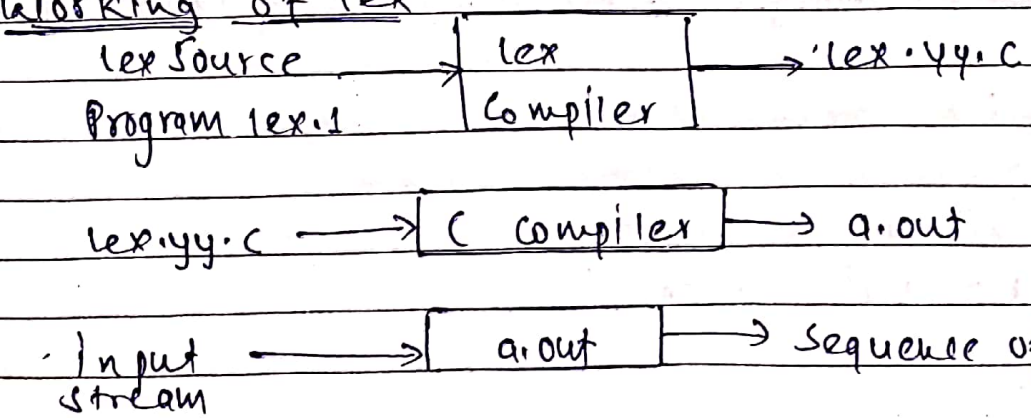
⇒ Lex is a program that generates lexical analyzer. It is used with YACC parser generator. The lexical analyzer is a program that transforms an input stream into a sequence of tokens. It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

structure of lex program:

A lex program is seperated into three sections by %%% delimiters.

```
{ definitions }
  %%%
{ rules }
  %%%
{ user subroutines }
```

Working of lex



lex program to implement a calculator.

```
%{
  int op = 0, i;
  float a, b;
%}
dig [0-9]+("."[0-9]+)
add "+"
sub "-"
mul "*"
div "/"
pow "^"
ln "ln"
%%
/* digi() is a user defined function */
{dig} {digi(); }
{add} { op = 1; }
{sub} { op = 2; }
{mul} { op = 3; }
{div} { op = 4; }
{pow} { op = 5; }
{ln} { printf("\n The Answer : %f \n\n", a); }
%%
digi()
{
if (op = 0)
  a = atof(yytext);
switch (op)
 {
```

```
case 1: a=a+b;
    break;
case 2: a=a-b;
    break;
case 3: a=a*b;
    break;
case 4: a=a|b;
    break;
case 5: for(i=a; b>1; b--)
    a=a*i;
    break;
}
  op=0;
}
}
main(int argv, char *argc[])
{
    yylex(); }
  yywrap()
  {
    return 1;
  }
```

28. describe the different strategies that a parser can employ to recover from a syntactic error.

⇒ The different strategies that a parser can employ to recover from syntactic error are given below:

(i) Panic mode:

Once an error is found, the parser intends to find designated set of synchronizing tokens by discarding input symbols one at a time. Synchronizing tokens are delimiters, semicolon or } whose role in source program is clear.

- When parser finds an error in the statement, it ignores the rest of the statement by not processing the input.
- This is the easiest way of error-recovery.
- It prevents the parser from developing infinite loops.

Advantage:

(i) simplicity
(ii) never get into infinite loop.

Disadvantage:

(i) Additional errors cannot be checked as some of the input symbols will be skipped.

(ii) Phrase Level:

Parser performs local correction on the remaining input when an error is detected.

- When a parser finds an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead.
- One wrong correction will lead to an infinite loop.

Advantage

- It can correct any input strings.

Disadvantages:

- It is difficult to cope up with actual error if it has occurred before the point of detection.

(iii) Error production:

productions which generates erroneous constructs are augmented to the grammar by considering common errors that occur. These productions detect the anticipated errors during parsing.

Error diagnostics about the erroneous constructs are generated by the parser.

-- self correction + diagnostic messages.

(iv) Global correction:

There are algorithms which make changes to modify an incorrect string into a correct string.

These algorithms perform minimal sequence of changes to obtain globally least-cost correction. When a grammar G and an incorrect string p is given, these algorithms find a parse tree for a string q related to p with smaller number of transformations.

29. Consider the following CFG $G = (N = \{S, A, B, C, D\}, T = \{a, b, c, d\}, P, S)$ where the set of productions P is given below:

$S \rightarrow A$

$A \rightarrow BC \mid DBC$

$B \rightarrow Bb \mid \varepsilon$

$C \rightarrow c \mid \varepsilon$

$D \rightarrow a \mid d$

Is this grammar suitable to be parsed using the recursive descendant parsing method? Justify and modify the grammar if needed.

Sol^n

Recursive descent parsing is actually a technique. It cannot handle left-recursion because it is a top-down parsing technique and also it should be left-factored grammar.

Eliminating the left recursion:

$S \rightarrow A$

$A \rightarrow BC \mid DBC$

$B \rightarrow B'$

$B' \rightarrow bB' \mid \varepsilon$

$C \rightarrow c \mid \varepsilon$

$D \rightarrow a \mid d$

Now,

Eliminating left-factoring

$S \rightarrow A$
$A \rightarrow EBC$
$E \rightarrow D \mid \varepsilon$
$B \rightarrow B'$
$B' \rightarrow bB' \mid \varepsilon$
$C \rightarrow c \mid \varepsilon$
$D \rightarrow a \mid d$

Hence, this is the required grammer suitable to be parsed using the recursive descendent parsing method.

30. Consider the grammar

bexpr → bexpr or bterm | bterm
bterm → bterm and bfactor | bfactor
bfactor → not bfactor | (bexpr) | true | false

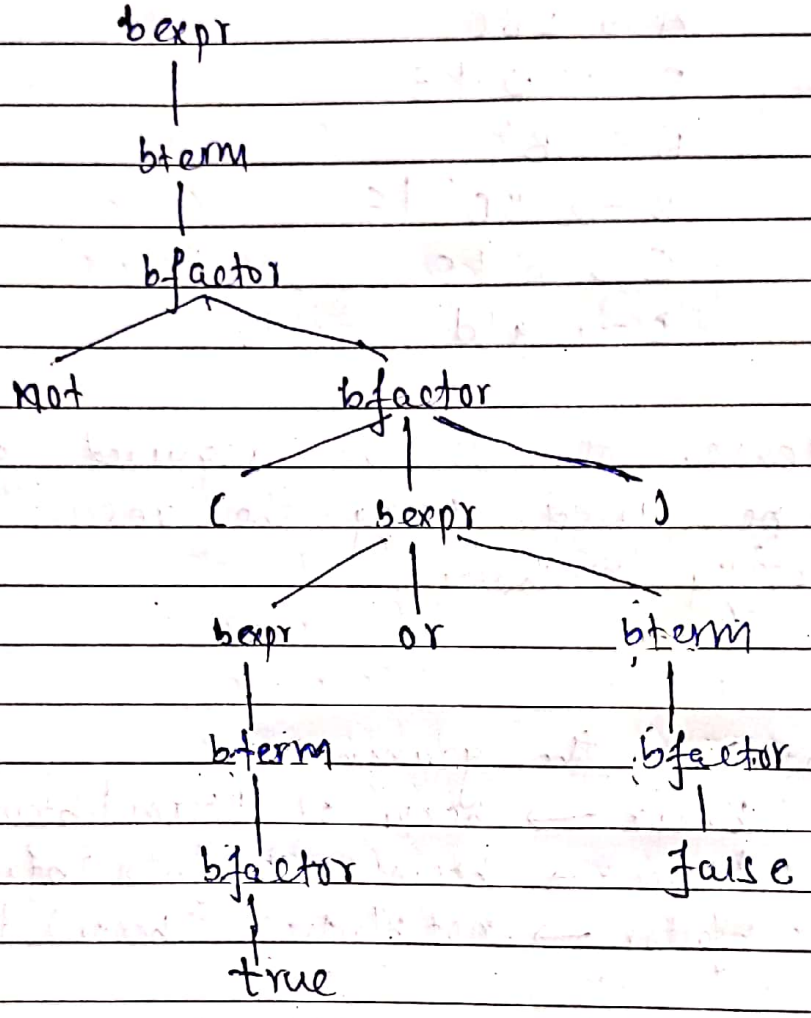
a) construct a parse tree for the input string not(true or false)
b) Show that this grammar generates all boolean expressions.

Soln

Removing left recursion

bexpr → bterm E'
E' → or bterm E' | ε
bterm → bfactor T'
T' → and bfactor T' | ε
bfactor → not bfactor | (bexpr) | true | false

a) Parse tree



b) By traversing the tree, the tree generated will lead to bfactor which will generates all boolean expression.

- bexpr → bterm → bfactor → true
- bexpr → bterm → bfactor → false
- bexpr → bterm → bfactor → not bfactor → not true
- bexpr → bterm → bfactor → not bfactor → not false
- bexpr → bexpr or bterm → bterm or bterm → bfactor or bfactor → true or false

- bexpr → bexpr and bterm → bterm and bterm → bfactor and bfactor
  ↓
  true and false

- bexpr → bterm → bfactor → not bfactor → not (expr)
  ↓
  not (bfactor or bfactor) ← not(bterm or bterm) ← not (b expr or bterm)
  ↓
  not (true or false

31. Consider the grammar

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

(i) Construct FIRST and FOLLOW sets.
(ii) Construct the predictive parsing table.
(iii) Show the moves of the predictive parser for input (a, (a, a)).

Sol^n

At first remove the left recursion.

$S \rightarrow (L) \mid a$

$L \rightarrow SL'$

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

(i)

Productions	First	Follow
$S \rightarrow (L) \mid a$	{ (, a }	{ $, , ,) }
$L \rightarrow SL'$	{ (, a }	{) }
$L' \rightarrow ,SL' \mid \varepsilon$	{ , , ε }	{) }

(ii) Parse table

Non Terminals	Input symbols				
	(	)	a	,	$
S	S → (L)		S → a		
L	L → SL'		L → SL'		
L'		L' → ε		L' → , SL'	

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

Stack	Input	Production Applied.
$S	(a, (a, a))$	S → (L)
$)L(	(a, (a, a))$	matched (, pop
$)L	a, (a, a))$	L → SL'
$)L'S	a, (a, a))$	S → a
$)L'a	a, (a, a))$	matched a, pop
$)L'	, (a, a))$	L' → , SL'
$)L'S,	, (a, a))$	matched , , pop
$)L'S	(a, a))$	S → (L)
$)L')L(	(a, a))$	matched (, pop
$)L')L	a, a))$	L → SL'
$)L')L'S	a, a))$	S → a
$)L')L'a	a, a))$	matched a, pop
$)L')L'	, a))$	L' → , SL'
$)L')L'S,	, a))$	matched , , pop
$)L')L'S	a))$	S → a
$)L')L'a	a))$	matched a, pop
$)L')L'	))$	L' → ε
$)L')	))$	matched), pop
$)L'	)$	L' → ε
$)	)$	matched), pop
$	$	Accept & successful completion.

32. Left factoring is a technique to remove non-determinism. How left factoring does avoid backtracking in grammars? Apply left factoring in the following grammar to make it deterministic.

$A \rightarrow aBaC \mid aBb \mid aB \mid a$

$B \rightarrow \varepsilon$

$C \rightarrow \varepsilon$

So,

⇒ Left factoring is a technique to remove non-determinism. sometime we find common prefix in many productions like $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$, where α is common prefix, while processing α we cannot decide whether to expan A by $\alpha\beta_1$ or by $\alpha\beta_2$, so this needs backtracking. To avoid such problem, grammar can be left factoring.

Left factoring is a process of factoring the common prefixes of the alternatives of grammar rule. If the production of the form $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$ has α as common prefix, by left factoring we get the equivalent grammar as

$A \rightarrow \alpha A'$

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

So, we can immediately expand A to $\alpha A'$.

⇒ <u>After left factoring grammar is,</u>

$A \rightarrow aA'$

$A' \rightarrow BA'' \mid \varepsilon$

$A'' \rightarrow ac \mid b \mid \varepsilon$

$B \rightarrow \varepsilon$

$C \rightarrow \varepsilon$

Hence, this is the required grammer.