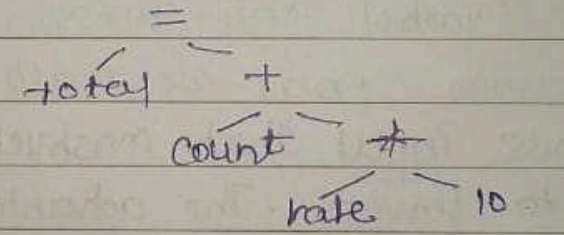
Assignment -3.

1. Write a note on roll of parser in syntax analysis

Syntax analysis →

The syntax analysis is called parsing in this phase the tokens generated by the lexical analyser are grouped together to form a hierchical structure. The syntax analysis determines the structure of the source string by grouping the token toghether The hierachical structure ~~generating~~ in the phase is called parse three or syntax tree for the expension

total = count + rate * 100 the parse tree can be generate as follows.



In the statement ~~note~~ total= count+rate*100 first of all rate * 10. will be considered because in arithmatic expression the multiplication operation should be performed before addition & the addition operation will be considerated for building such type of Syntax tree the production rules are to be designed the rule are usually expressed by conlex free grammar for above style

E ← identifier
E ← number
E ← E1 + E2
E ← E1 * E2
E ← (E)

(d)
Where E stands for expression ~~By rule 4~~
By Rule (1) count & state are expression
By Rule (2) 10 is also expression.
By Rule (4) we get rate * 10 expression
and finally count + rate * 10 is an exp

2 Explain top down parsing in example.

→ Top-down parsing in the construction of a parse tree by the starting at start symbol and guessing each derivation until we reach a string that matches input that construct tree from root to leaves. The advantage of top-down parsing in that parsers can directly be written as program table-driven top-down parsers. bottom-up parsing is practically relevant.

eg -

S → / F E then S else S/ ~~while~~
while E does ~~not~~ print.
E → true / false / id

The input taken string is : if id then while true do print else print

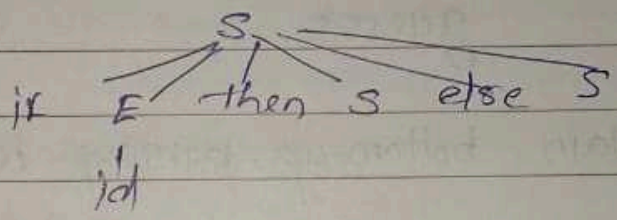
1. Tree

S

Input : if id then while true do print else print.

Action : Guess for S.

2. Tree

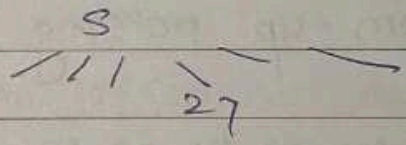


input : id then while true do print else print

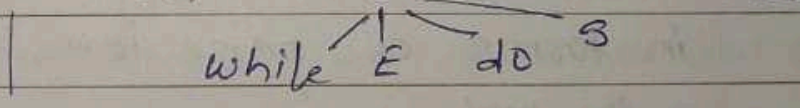
Action : id

matches : then matches : givess for S

4. Tree



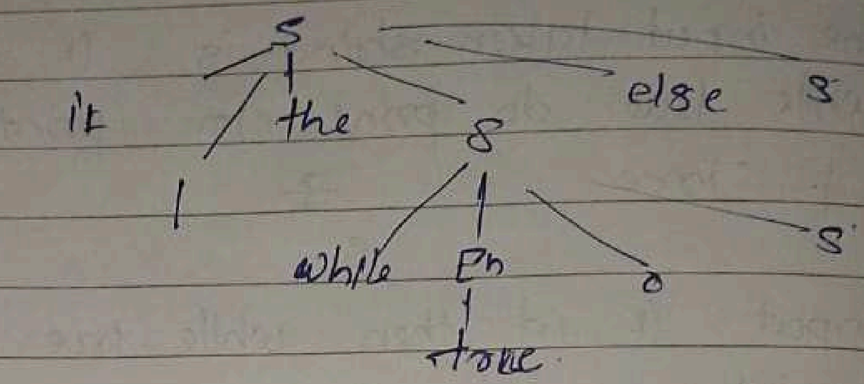
if then S else



input : while true do print else print

Action : while matches, guess for E

5. Tree



input: true do print else print
Action: true matches; do matches;
guesss

8 Explain bottom-up parsing techniques in eg.

Bottom up parser builds a derivation by working from the input sentence back towards the start symbol S Right most derivation in reverse order is done in bottom-up parsing

$S \Rightarrow r_0 \Rightarrow r_1 \Rightarrow r_2 \Rightarrow \cdots \Rightarrow r_{n-1} \Rightarrow r_n \Rightarrow$ sentence

$\longleftarrow$

in terms of parse tree working from leaves to root.

eg.

$S \longrightarrow aAcBe$

$A \longrightarrow Ab/b.$

$B \longrightarrow d$

Input abbcde

Right most derivation

$\leftarrow$ aAcBe

$\leftarrow$ aAcde

$\leftarrow$ aAb cde

$\leftarrow$ abb cde.

Bottom-up Approach

Right sentencial form.	Reduction
abb cde	
aAb cde	A $\rightarrow$ b
A a cde	A $\rightarrow$ Ab
A a cBe	B $\rightarrow$ d
S	S $\rightarrow$ aACBe

4 While a note on recursive decent method-

Top-down parsing can be viewed at an attempt to find a left most derivation for an input string equivalently, it can be viewed as a attempt to construct a parse tree for the input starting from the root and creating the nodes of the parser tree in preorder.

The special case of recursive-decent parsing, called predictive parsing

where no backtracking is required
The general form of top-down parsing called recursive descent, that may involve backtrack that is masking repeated scans of the input.
Recursive descent parser is top-down parser involving backing It makes a repeated seen of the input Backtracking parsi er are not see frequently as back tracking is very needed to parser programming language construct.

$F \rightarrow OF'$

$F' \rightarrow *qF' / E$

```
F()
{ if (look_ahead == 'a')
  { match ('a');
    F'();
  }
}
F'()
{ if (look_ahead == '*')
  { match ('*');
    match ('a');
    F'();
  }
else
```

```
{ return ; }
match ( char c)
{
  if ( look-ahead == c)
  {     look. ahead = getchar ();
  else
        print (" Error! ");
}
;  main ()
{
     f();
    if (look_ahead == '$')
        print (" parsing successfully");
    }
```

Input ⇒ - a * b $

Solve the following example

S → aAB | bB

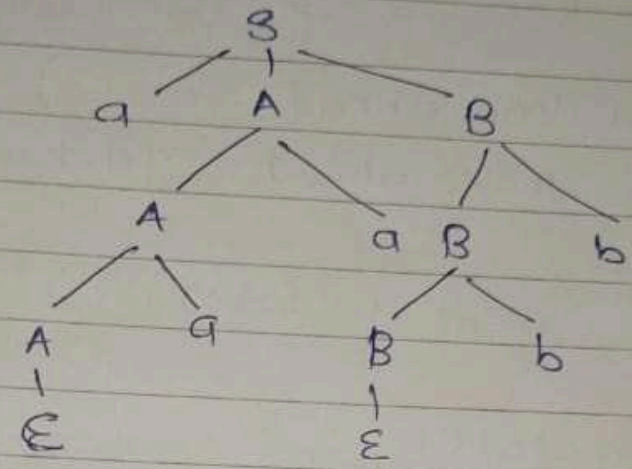
A → Aa | ε

B → Bb | ε

5- for topdown & bottomUp parsing drive string by using above pute aaabbb.

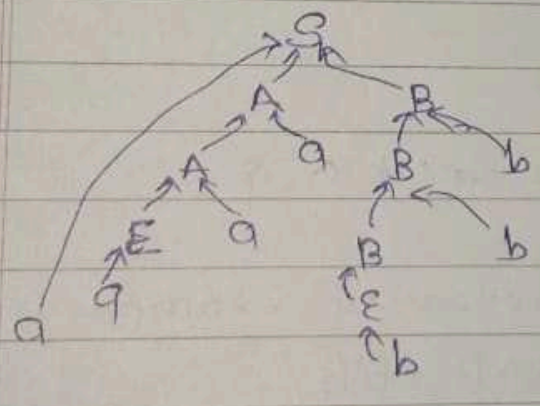
string aaa bbb

Top Down approach.



S → aAB
S → aAaB
S → aAaaB
S → aaaB
S → aaaBb
S → aaaBbb
S → aaaBbbb
S → aaabbb

Bottom up Approach



S → aAB
S → aABb
S → aABbb
S → aABbbb
S → aAbbb
S → aAabbb
S → aAaabbb
S → aaabbb

6. Explain predictive parser Construct the following grammer.

S → AB | ε
A → aAB | ε
B → bA

Finding first () sets

1. first (A) = first (a) ∪ first (E) = {a, ε}
2. First (B) = First (b) = {b}
3. first (S) = First (A) − {ε} = {a, ε, b}

Finding follow () sets

1. follow (S) = {$}
2. follow (A) = first (B) − {ε} ∪ first (B) − {ε}
   = {b, $}
3. follow (B) = follow (S) ∪ follow (A)
   = {$, b}

8 Explain the working of operator precidance grammar with example

id * id * id.

E → E+E | E-E | E*E | E/E | E^E | (E) | -E | id.

1. No. of production right side is should Countain ε.
2. NO. production sight side should contain two adjacent non terminal.
   is called an operator grammer.

operator-precedences parsing has three distain t precedence relations <, = and .> between certain pair of terminals. these

precedence relations guide the selection of handles and have following meaning.

RELATION	MEANING
$a <\cdot b$	a yield precedence to b
$a = b$	a has the same prece
$a \cdot> b$	a takes precedence over

Disadvantages

L(G) ≠ L (parser)

errer detection

usage is limited

They are easy to analyse manual

string id + id * id

	id	+	*	$
id		$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	

solution

operator grammer is

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

The input string with precedence relation inserted is-

$ <· id ·> + <· id ·> * <· id ·> $

The reduce id to E At the point we have

E + id * id

By repeating the process & proceding in the same way $ + <· id ·> * <· id ·> $

substitude E → id

After reducing the other id to E by the same process

E + E * E

Now, the I/p string after deleting the

=> $ + * $

Inserting the precedence relation we get

$ <· + <· * ·> $

Reducing by E → E * E we get

E + E

Now the input string $ + $

again inserting precedence relation we get

-> $ <· + ·> $

reducing By E → E + we get

$ $

& Finally we are left with.

E

10 consider the following Context free grammar

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

where P is:

$S \rightarrow AaAb$

$S \rightarrow Bb$

$A \Rightarrow \varepsilon$

$B \rightarrow \varepsilon$

12

a) Computer the first sets for A, B & S

b) Computer the follow sets for A, B & S

c) is the CFG of LL(1) ? Justify.

$S \rightarrow Aa\ Ab$

$S \rightarrow Bb$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

First (B) = $\{\varepsilon\}$

first (A) = $\{\varepsilon\}$

first (S) = $\{a, b\}$

follow (S) = $\{\$\}$

follow (A) = $\{a, b\}$

follow (B) = $\{b\}$

	a	b
S	1	2
A	3	3
B		4

It is a LL(1).

7. ~~What is Bottom up~~

7. How to implement shift reduce parser.

$S \rightarrow xPy$

$P \rightarrow xP \mid Qy$

$Q \rightarrow y$

Shift reduce parsing uses a stack to hold grammar symbols and input buffer to hold string to be parsed, because handles always appear at the top of the stack i.e., there's no need to look deeper into the stack.

A shift-reduce parser has just four actions:

1. Shift - next word is shifted onto the stack (input symbols) until a handle is formed.
2. Reduce - right end handle is at top of stack, locate left end of handle within the stack, pop handle off stack and push appropriate LHS.
3. Accepts - stop parsing on successful completion of parser and report success.

4. error - call an error reporting / recovery routine.

Possible Conflicts

Ambiguous grammars leads to parsing Conf

1. Shift-reduce → Both a shift action and a reduce action are possible in the state
   Ex→ dangling-else problem.

2. Reduce-reduce → Two or more distinct reduce actions are possible in the same state.