Unit-II

Q.1.What is Context Free Grammar. Give an example.

Ans : A context-free grammar (grammar for short) consists of **terminals, nonterminals, a start symbol,** and **productions**.

• **Terminals** are the basic symbols from which strings are formed. The term "token name" is a synonym for "terminal“.

• **Nonterminals** are syntactic variables that denote sets of strings.

• In a grammar, one nonterminal is distinguished as the **start symbol**, and the set of strings it denotes is the language generated by the grammar.

• The **productions** of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings.

• Each production consists of: (a) A nonterminal called the head or left side of the production.

• A body or right side consisting of zero or more terminals and nonterminals.

Example:

L= {wcw^R | w € (a, b)*}

Production rules:

1. S → aSa
2. S → bSb
3. S → c

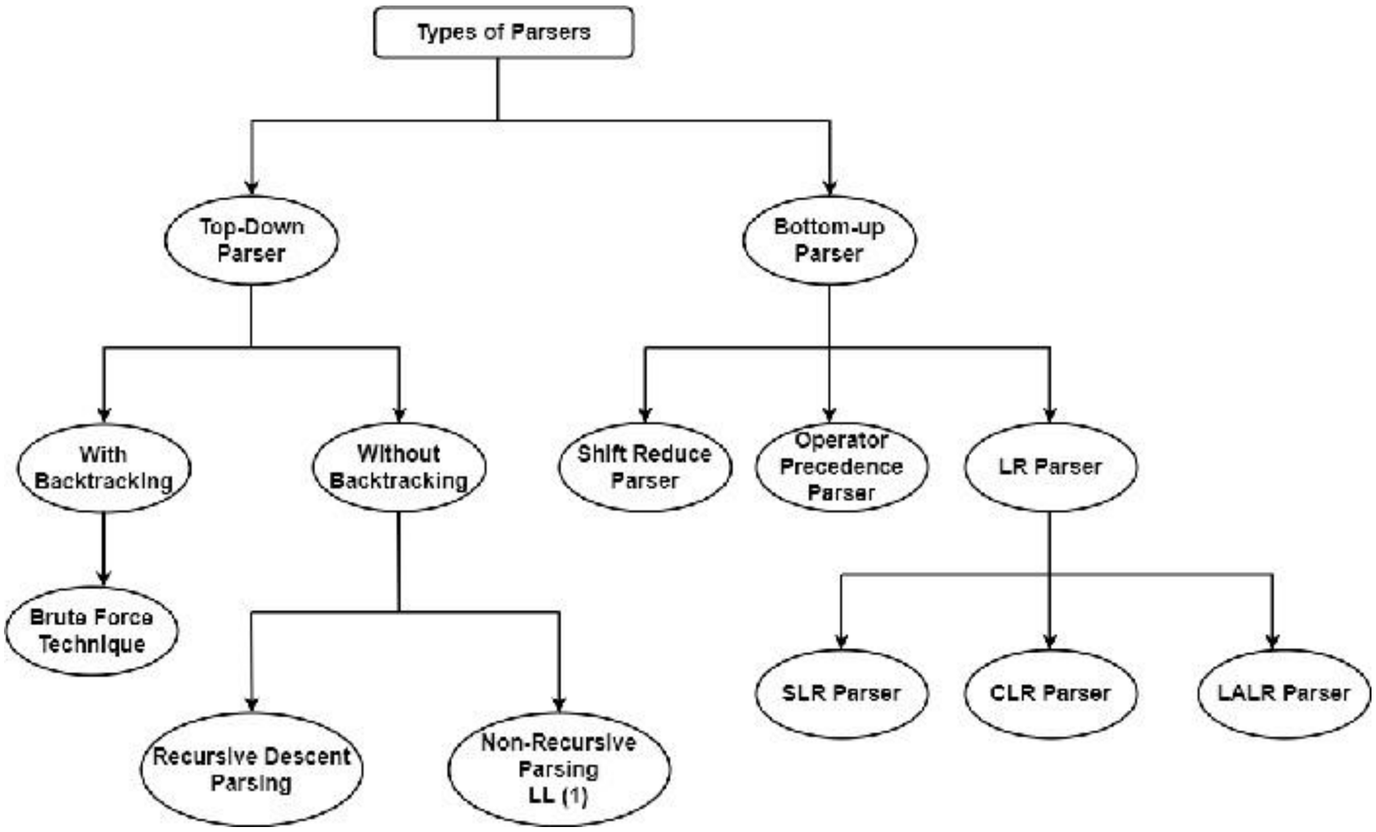
Now check that abbcbba string can be derived from the given CFG.

1. S ⇒ aSa
2. S ⇒ abSba
3. S ⇒ abbSbba
4. S ⇒ abbcbba

By applying the production S → aSa, S → bSb recursively and finally applying the production S → c, we get the string abbcbba.

Q.2. Give the classification of parsing techniques and briefly explain each.

Ans : Parsing is known as Syntax Analysis. It contains arranging the tokens as source code into grammatical phases that are used by the compiler to synthesis output generally grammatical phases of the source code are defined by parse tree. There are various types of parsing techniques which are as follows −



- **Top-Down Parser**

It generates the Parse Tree from root to leaves. In top-down parsing, the parsin begins from the start symbol and changes it into the input symbol.

An example of a Top-Down Parser is Predictive Parsers, Recursive Descent Parser.

Predictive Parser − Predictive Parser is also known as Non-Recursive Predictive Parsing. A predictive parser is an effective approach of implementing recursivedescent parsing by manipulating the stack of activation records explicitly. The predictive parser has an input, a stack, a parsing table, and an output. The input includes the string to be parsed, followed by $, the right-end marker.

Recursive Descent Parser − A top-down parser that executes a set of recursive procedures to process the input without backtracking is known as recursive-descent parser, and parsing is known as recursive-descent parsing.

- **Bottom-Up Parser**

It generates the Parse Tree from leaves to root for a given input string. In Grammar, the input string will be reduced to the starting symbol.

Example of Bottom-Up Parser is Shift Reduce Parser, Operator Precedence Parser, and LR Parsers.

Shift Reduce Parser − Shift reduce parser is a type of bottom-up parser. It uses a stack to influence the grammar symbols. A parser goes on changing the input symbols onto the stack until a handle comes on the top of the stack. When a handle occurs on the top of the stack, it implements reduction.

Operator Precedence Parser − The shift-reduce parsers can be generated by hand for a small class of grammars. These grammars have the property that no production on the right side is ϵ or has two adjacent non-terminals. Grammar with the latter property is known as operator grammar.

LR Parsers − The LR Parser is a shift-reduce parser that creates use of deterministic finite automata, identifying the set of all viable prefixes by reading the stack from bottom to top. It decides what handle, if any, is available.

A viable prefix of a right sequential form is that prefix that includes a handle, but no symbol to the right of the handle. Thus, if a finite state machine that identifies viable prefixes of the right sentential form is generated, it can guide the handle selection in the shift-reduce parser.

There are three types of LR Parsers which are as follows −

- **Simple LR Parser (SLR)** − It is very easy to implement but it fails to produce a table for some classes of grammars.
- **Canonical LR Parser (CLR)** − It is the most powerful and works on large classes of grammars.
- **Look Ahead LR Parser (LALR)** − It is intermediate in power between SLR and CLR.

Q.3. Differentiate between Top down and Bottom-up parsing.

Ans :

Top-Down Parsing	**Bottom-Up Parsing**
It is a parsing strategy that first looks at the highest level of the parse tree and works down the parse tree by using the rules of grammar.	It is a parsing strategy that first looks at the lowest level of the parse tree and works up the parse tree by using the rules of grammar.
Top-down parsing attempts to find the left most derivations for an input string.	Bottom-up parsing can be defined as an attempt to reduce the input string to the start symbol of a grammar.
In this parsing technique we start parsing from the top (start symbol of parse tree) to	In this parsing technique we start parsing from the bottom (leaf node of the parse

down (the leaf node of parse tree) in a top-down manner.	tree) to up (the start symbol of the parse tree) in a bottom-up manner.
This parsing technique uses Left Most Derivation.	This parsing technique uses Right Most Derivation.
The main leftmost decision is to select what production rule to use in order to construct the string.	The main decision is to select when to use a production rule to reduce the string to get the starting symbol.
Example: Recursive Descent parser.	Example: ItsShift Reduce parser.

Q.4. “Top-down parser is equivalent to left most derivation”. Justify this with an example.

Ans : The statement "Top-down parser is equivalent to leftmost derivation" refers to the fact that when a top-down parser constructs the parse tree for a given input string, it simulates a leftmost derivation of that string. This means that at each step, the parser selects the leftmost non-terminal in the current sentential form and expands it according to the production rules of the grammar.

Let's illustrate this with an example:

Consider the following grammar:

```
S -> aAb
A -> b | ε
```

And the input string: `aab`

Now, let's demonstrate how a top-down parser (specifically, a recursive descent parser) constructs the parse tree for this input string by simulating a leftmost derivation:

1. Step 1: Start with the start symbol `S`.
2. Step 2: Apply production `S -> aAb`, so the parse tree becomes:
   ```
   ```

```
    S
   / \
  a   Ab
```

3. Step 3: Expand the leftmost non-terminal `A` according to the production `A -> ε`, so the parse tree becomes:

```
    S
   / \
  a   Ab
       \
        ε
```

4. Step 4: Expand the leftmost non-terminal `A` according to the production `A -> b`, so the parse tree becomes:

```
    S
   / \
  a   Ab
       \
        b
```

At each step, the top-down parser selects the leftmost non-terminal (`A` in this case) and expands it according to the production rules. This process exactly mirrors a leftmost derivation of the input string. Therefore, a top-down parser constructs a parse tree equivalent to a leftmost derivation of the input string.

Q.5. What is Recursive Descent Parser. Explain with an Example.

Ans : A general form of top-down parsing, called recursive descent parsing, which may require backtracking to find the correct production to be applied.

Typically, top-down parsers are implemented as a set of recursive functions that descent through a parse tree for a string. This approach is known as recursive descent parsing, also known as LL(k) parsing where the first L stands for left-to-right, the second L stands for leftmost-derivation, and k indicates k-symbol lookahead.

Recursive descent parsing has the following benefits:

1. Ease of use: Because recursive descent parsing closely mimics the grammar rules of the language being parsed, it is simple to comprehend and use.
2. Readability: The parsing code is usually set up in a structured and modular way, which makes it easier to read and maintain.
3. Recursive descent parsers can produce descriptive error messages, which make it simpler to find and detect syntax mistakes in the input. 3. Error reporting.
4. Predictability: The predictable behavior of recursive descent parsers makes the parsing process deterministic and clear.

Recursive descent parsing, however, also has certain drawbacks:

1. Recursive descent parsers encounter difficulties with left-recursive grammar rules since they can result in unbounded recursion. To effectively handle left recursion, care must be made to avoid it or employ methods like memoization.
2. Recursive descent parsers rely on backtracking when internal alternatives to a grammar rule are unsuccessful. This could result in inefficiencies, especially if the grammar contains a lot of ambiguity or options.
3. Recursive descent parsers frequently adhere to the LL(1) constraint, which requires that they only use one token of lookahead. The grammar's expressiveness is constrained by this restriction because it is unable to handle some ambiguous or context-sensitive languages.

This parsing method may involve backtracking.

<u>Example for</u> : Backtracking

Consider the grammar G :

S → cAd

A→ ab | a

and the input string **w=cad**.

Q.6. Define FIRST and FOLLOW and illustrate with an Example.

Ans : FIRST and FOLLOW are two functions associated with grammar that help us fill in the entries of an M-table.

FIRST ()− It is a function that gives the set of terminals that begin the strings derived from the production rule.

A symbol c is in FIRST (α) if and only if α ⇒ cβ for some sequence β of grammar symbols.

A terminal symbol a is in FOLLOW (N) if and only if there is a derivation from the start symbol S of the grammar such that S ⇒ αNαβ, where α and β are a (possible empty) sequence of grammar symbols. In other words, a terminal c is in FOLLOW (N) if c can follow N at some point in a derivation.

Benefit of FIRST ( ) and FOLLOW ( )

- It can be used to prove the LL (K) characteristic of grammar.
- It can be used to promote in the construction of predictive parsing tables.
- It provides selection information for recursive descent parsers.

Computation of FIRST

FIRST (α) is defined as the collection of terminal symbols which are the first letters of strings derived from α.

FIRST (α) = {α |α →∗ αβ for some string β }

If X is Grammar Symbol, then First (X) will be −

- If X is a terminal symbol, then FIRST(X) = {X}
- If X → ε, then FIRST(X) = {ε}
- If X is non-terminal & X → a α, then FIRST (X) = {a}
- If X → Y_1, Y_2, Y_3, then FIRST (X) will be

(a) If Y is terminal, then

FIRST (X) = FIRST (Y_1, Y_2, Y_3) = {Y_1}

(b) If Y_1 is Non-terminal and

If Y_1 does not derive to an empty string i.e., If FIRST (Y_1) does not contain ε then, FIRST (X) = FIRST (Y_1, Y_2, Y_3) = FIRST(Y_1)

(c) If FIRST (Y_1) contains ε, then.

FIRST (X) = FIRST (Y_1, Y_2, Y_3) = FIRST(Y_1) − {ε} ∪ FIRST(Y_2, Y_3)

Similarly, FIRST (Y_2, Y_3) = {Y_2}, If Y_2 is terminal otherwise if Y_2 is Non-terminal then

- FIRST (Y_2, Y_3) = FIRST (Y_2), if FIRST (Y_2) does not contain ε.
- If FIRST (Y_2) contain ε, then
- FIRST (Y_2, Y_3) = FIRST (Y_2) − {ε} ∪ FIRST (Y_3)

Similarly, this method will be repeated for further Grammar symbols, i.e., for Y_4, Y_5, Y_6 … . Y_K.

Computation of FOLLOW

Follow (A) is defined as the collection of terminal symbols that occur directly to the right of A.

FOLLOW(A) = {a|S ⇒* αAaβ where α, β can be any strings}

Rules to find FOLLOW

- If S is the start symbol, FOLLOW (S) ={$}
- If production is of form A → α B β, β ≠ ε.

(a) If FIRST (β) does not contain ε then, FOLLOW (B) = {FIRST (β)}

Or

(b) If FIRST (β) contains ε (i. e. , β ⇒* ε), then

FOLLOW (B) = FIRST (β) − {ε} ∪ FOLLOW (A)

∵ when β derives ε, then terminal after A will follow B.

- If production is of form A → αB, then Follow (B) ={FOLLOW (A)}.

(Example koi bhi dal dena like koi bhi grammer consider krke FIRST FOLLOW nikal lena)

Q.7. Construct the Predictive Parser for the following grammar:-

E→E+T/T

T→T*F/F

F→id

Q.8. What are the conditions to be satisfied for the grammar to be LL(1)?

Ans : LL(1) Parsing: Here the 1st L represents that the scanning of the Input will be done from the Left to Right manner and the second L shows that in this parsing technique, we are going to use the Left most Derivation Tree. And finally, the 1 represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.

Essential conditions to check first are as follows:

1. The grammar is free from left recursion.
2. The grammar should not be ambiguous.
3. The grammar has to be left factored in so that the grammar is deterministic grammar.

These conditions are necessary but not sufficient for proving a LL(1) parser.

Algorithm to construct LL(1) Parsing Table:

Step 1: First check all the essential conditions mentioned above and go to step 2.

Step 2: Calculate First() and Follow() for all non-terminals.

1. **First():** If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.
2. Follow(): What is the Terminal Symbol which follows a variable in the process of derivation.

Step 3: For each production A –> α. (A tends to alpha)

1. Find First(α) and for each terminal in First(α), make entry A –> α in the table.
2. If First(α) contains ε (epsilon) as terminal, then find the Follow(A) and for each terminal in Follow(A), make entry A –> ε in the table.
3. If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> ε in the table for the $.
   To construct the parsing table, we have two functions:

In the table, rows will contain the Non-Terminals and the column will contain the Terminal Symbols. All the **Null Productions** of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of the First set.

Now, let's understand with an example.

Example 2: Consider the Grammar

S-->A|a
A-->a
Solution :

Step 1: The grammar does not satisfy all properties in step 1, as the grammar is ambiguous. Still, let’s try to make the parser table and see what happens

Step 2: Calculating first() and follow()

Find their First and Follow sets:

	First	**Follow**
S –> A/a	{ a }	{ $ }
A –>a	{ a }	{ $ }

Step 3: Make a parser table.

Parsing Table:

	a	**$**
S	S –> A, S –> a	
A	A –> a	

Here, we can see that there are two productions in the same cell. Hence, this grammar is not feasible for LL(1) Parser.

Q.9. Is following grammar LL(1)?Why?

S->iEtSS’|a

S’->eS|∈

E->b

Q.10. "Bottom up Parsing is equivalent to Rightmost Derivation in reverse." Justify with an example**.**

Ans : The statement "Bottom-up parsing is equivalent to rightmost derivation in reverse" means that when a bottom-up parser constructs the parse tree for a given input string, it simulates a rightmost derivation of that string in reverse order. In a rightmost derivation, at each step, the rightmost non-terminal in the current sentential form is replaced by its production.

Let's illustrate this with an example:

Consider the following grammar:

```

S -> AB

A -> a

B -> b | ε
```

And the input string: `aab`

Now, let's demonstrate how a bottom-up parser (specifically, an LR parser) constructs the parse tree for this input string by simulating a rightmost derivation in reverse:

1. Step 1: Start with the input string `aab`.

2. Step 2: Shift the first terminal `a` onto the stack.

3. Step 3: Shift the second terminal `a` onto the stack.

4. Step 4: Reduce `aa` to `A` using production `A -> a`. Now the stack becomes `[A]`.

5. Step 5: Reduce `A` to `S` using production `S -> AB`. Now the stack becomes `[S]`.

6. Step 6: Shift the last terminal `b` onto the stack.

7. Step 7: Reduce `b` to `B` using production `B -> b`. Now the stack becomes `[S, B]`.

8. Step 8: Reduce `SB` to `S` using production `S -> AB`. Now the stack becomes `[S]`.

9. Step 9: Finally, reduce `S` to the start symbol using the production `S -> AB`, resulting in the parse tree:

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

At each step, the bottom-up parser performs shift and reduce actions to mimic the rightmost derivation of the input string in reverse order. Therefore, bottom-up parsing constructs a parse tree equivalent to a rightmost derivation of the input string.

Q.11. What do you mean by handle? Explain with an example

Ans : A handle is a substring that connects a right-hand side of the production rule in the grammar and whose reduction to the non-terminal on the left-hand side of that grammar rule is a step along with the reverse of a rightmost derivation.

Finding Handling at Each Step

Handles can be found by the following process −

- It can scan the input string from left to right until first .> is encountered.
- It can scan backward until <. is encountered.
- The handle is a string between <. and .>

Example− Compute FIRST & LAST terminals of non-terminals E, T, and F in the following Grammar.

E → E + T| T

T → T * F | F

F → (E)| id

Solution

On seeing the production, we can judge

+ is the first terminal of E

* is the first terminal of T

(, id is the first terminals of F.

But E → T → F

∴ The First Terminal of F is contained in the First terminal of T and the First Terminal of T is contained in the First Terminal of E.

∴ First(F) = {(, id}

∴ First(T) =*∪ First (F) = {*, (, id}

First(E) = + ∪ First (T) = {+,*, (, id}

Similarly, the Last Terminals can be found.

Non-Terminal	FIRST Terminal	LAST Terminal
F	(, id	), id
T	*, (, id	*,), id
E	+,*, (, id	+,*,), id.

Q.12.Explain the working of Shift Reduce Parser with an Example.

Ans : Shift Reduce parser attempts for the construction of parse in a similar manner as done in bottom-up parsing i.e. the parse tree is constructed from leaves(bottom) to the root(up). A more general form of the shift-reduce parser is the LR parser.

This parser requires some data structures i.e.

- An input buffer for storing the input string.
- A stack for storing and accessing the production rules.

Basic Operations –

- Shift: This involves moving symbols from the input buffer onto the stack.
- Reduce: If the handle appears on top of the stack then, its reduction by using appropriate production rule is done i.e. RHS of a production rule is popped out of a stack and LHS of a production rule is pushed onto the stack.
- Accept: If only the start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accepted action is obtained, it is means successful parsing is done.

- Error: This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

Example – Consider the grammar
E –> 2E2
E –> 3E3
E –> 4
Perform Shift Reduce parsing for input string “32423”.

Stack	Input Buffer	Parsing Action
$	32423$	Shift
$3	2423$	Shift
$32	423$	Shift
$324	23$	Reduce by E --> 4
$32E	23$	Shift
$32E2	3$	Reduce by E --> 2E2
$3E	3$	Shift
$3E3	$	Reduce by E --> 3E3
$E	$	Accept

Q.13.Construct SLR Parser for the following grammar.

S->CC

C->cC

C->d

Q.14.What is left recursion and left factoring? Give example of each.

Ans : Left recursion and left factoring are common issues encountered when designing context-free grammars.

1. Left Recursion:**

Left recursion occurs in a grammar when a non-terminal can directly derive itself through one or more production rules. Left-recursive productions can lead to infinite recursion and ambiguity in parsing.

Example of left recursion:

Consider the following grammar:

```

A -> Aα | β

```

Here, the production `A -> Aα` is left-recursive because `A` directly derives itself. If we try to expand `A`, we can create an infinite loop: `A -> Aα -> Aαα -> Aααα -> ...`.

2. Left Factoring:

Left factoring is a technique used to eliminate common prefixes in alternative productions of a non-terminal. It helps in reducing ambiguity and simplifying parsing.

Example of left factoring:

Consider the following grammar:

```

A -> abX | abY

```

Here, both alternatives `abX` and `abY` start with the common prefix `ab`. To left factor this grammar, we can introduce a new non-terminal to represent the common prefix:

```
```

A -> abZ

Z -> X | Y

```

Now, the alternatives `X` and `Y` are factored out into the non-terminal `Z`, simplifying the grammar and reducing redundancy.

**Q.15.Why left recursion has to be eliminated from grammar before constructing top down parser?**
**Eliminate left recursion from the following grammar:**
**A → ABd | Aa | a**
**B → Be | b**
```