Decipher

1. Briefly explain a Language processing system

Ans: This flow chart represents the typical stages in the compilation process of a high-level programming language. Here's a brief explanation of each stage:

1. **Source Program:**
   - The original code written by a programmer in a high-level programming language.

2. **Preprocessor:**
   - The preprocessor takes the source program and performs text manipulations or modifications, such as macro expansion and inclusion of header files.

3. **Modified Source Program:**
   - The output of the preprocessor, which includes the modifications made during the preprocessing stage.

4. **Compiler:**
   - The compiler translates the modified source program into an intermediate representation or assembly language.

5. **Target Assembly Program:**
   - The output of the compiler is an assembly language program specific to the target architecture. It represents a **low-level version of the original code.**

6. **Assembler:**
   - The assembler converts the assembly language program into **relocatable machine code**, which includes machine instructions and memory addresses.

7. **Relocatable Machine Code:**
   - This is the output of the assembler and consists of machine code instructions along with information about memory addresses that may need to be adjusted during the linking process.

8. **Linker/Loader:**

- The linker combines multiple relocatable machine code files and resolves references between them. The loader then places the combined code into memory, making it ready for execution.

9. Target Machine Code:

- The final output, which is the machine code specific to the target computer architecture. This is the executable code that can be run on the target machine.

This process is essential for translating human-readable high-level code into machine-executable instructions, allowing software to be executed on a specific computer system. Each stage plays a crucial role in the overall compilation and linking process.

2.Write short notes on Regular Expression operators and mention the order of precedence of those operators.

Ans: Regular Expression: Regular Expressions are used for representing certain sets of strings in an algebraic fashion.

Regular Expression Operators:

These operations are:

1. UNION: The union of two languages L and M, denoted L ∪ M, is the set of strings that are in either L or M, or both. For example, if L = {001, 10, 111} and M = {ε, 001}, then L ∪ M = {ε, 10, 001, 111}.

2. CONCATENATION: The concatenation of languages L and M is the set of strings that can be formed by taking any string in L and concatenating it with any string in M. For example, if L = {001, 10, 111} and M = {ε, 001}, then **L.M or just LM**, is {001, 10, 111, 001001, 10001, 111001}.

3. STAR: The star (or, closure or Kleene closure) of a language L is denoted L * and represents the set of those strings that can be formed by taking any number of strings from L, possibly with repetitions (i.e., the same string may be selected more than once) and concatenating all of them.

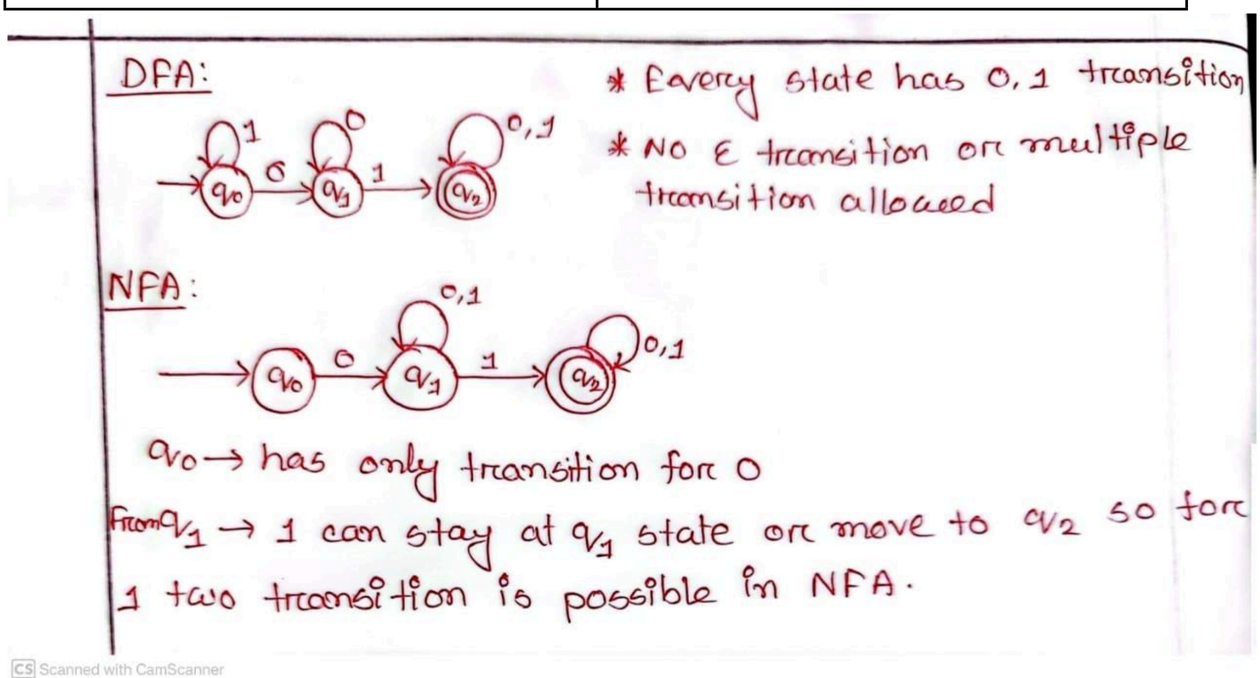
If L = {0, 11}, then L * consists of those strings of 0's and 1's such that the 1's come in pairs, e.g. **011, 11110, and ε but not 01011 or 101.**

Order of precedence of operators: Star, Concatenation and Union; From highest to lowest.

3.What is a Finite automaton? How is Deterministic Finite Automata different from NonDeterministic Finite Automata? Explain with appropriate examples?

Ans: The finite automaton (FA) is a mathematical model of a system, with discrete inputs and outputs. It consists of a finite **set of states**, and a **set of transition**s from states to states that occur in response to external "inputs" chosen from an alphabet.

DFA	NFA
For each symbolic representation of the alphabet, there is only one state transition in DFA.	There can be more than one state transition for each symbol.
DFA cannot use Empty (epsilon) String transition.	NFA can use the Empty String transition.
Dead configuration is not allowed. eg: if we give input as 0 on q0 state so we must give 1 as input to q0 as self loop.	Dead configuration is allowed. eg: if we give input as 0 on q0 state so we can give next input 1 on q1 which will go to the next state.
Backtracking is allowed in DFA.	Backtracking is not allowed in NFA.
Conversion of Regular expression to DFA is difficult.	Simpler compared to DFA



4.Explain the concept of Syntax Directed Translation and Attribute Grammar

Ans:

FLC

Q: Explain the concept of Syntax Directed Translation and Attribute Grammar:

⇒ Syntax Directed Grammar: is a techinique used compiler design where translation rules or action are embedded within the grammar of a programming language. These rules are associated with production of grammar. The goal of SDT is to associate semantic (meaningful) information with syntactic structure of the source code, faciliating the generation of intermediate code or target code during compilation.

$\boxed{\text{Grammar + Symantic Rules = SDT}}$

$S \to S \# A \mid A$	$\{S.val = S.val * A.val\}$
$A \to A \& B \mid B$	$\{A.val = A.val + B.val\}$
$B \to id$	$\{B.val = id.val\}$
G.	Rules

Attribute Grammar:

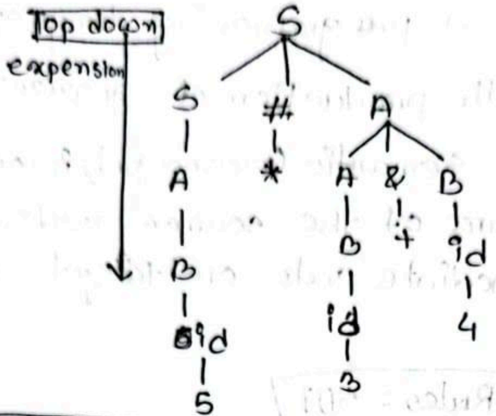
AG is a specific formalism (का विशिष्ट structure) within SDT that focuses on the use of attributes to carry and compute semantic information during the translation process.

There are two attributes:

- Synthesized → Parent gets value from child. $S \to AB$ (S will take value from B or A)
- Inherited → Child gets value from parent or siblings (for L-attribute STD, only take from Left siblings)

Example: input = 5#3&4

S-Attribute SDT: If an SDT uses only synthesized attributes, it is called as S-Attribute SDT.



5*3+4 = 15+4 = 19

it is like SDT using Rules.

L-Attribute SDT: This form of SDT uses both synthesi and inherited attributes with restriction of not taking value from right siblings → "only take from Left"

~~input~~ int

5.Illustrate the common error recovery techniques in parsing.

1. **Panic Mode:**
    - In this method, when an error is encountered anywhere in the statement, successive characters from the input are ignored one at a time until a designated set of synchronizing tokens is found. Synchronizing tokens are delimiters such as ; or } .
    - The advantage of this method is that it is the easiest to implement and guarantees not to go to infinite loop.
    - The disadvantage is that a considerable amount of input is skipped without checking it for additional errors.

2. **Statement Mode:**
    - In this method, when a parser encounters an error, it performs necessary correction on remaining input so that the rest of input statement allow the parser to parse ahead.
    - The correction can be deletion of extra semicolons, replacing comma by semicolon or inserting missing semicolon.
    - While performing correction, utmost care should be taken for not going in infinite loop.|
    - Disadvantage is that it finds difficult to handle situations where actual error occurred before point of detection.

6.Describe the preliminary simplifications that are needed to be made to get a normal form of CFG.

Ans: The goal is to try to get a Normal Form of the CFG like Chomsky Normal Form or Greibach Normal Form. To get there, we need to make a number of preliminary simplifications, which are themselves useful in various ways:

1. We must eliminate **useless symbols**, those variables or terminals that do not appear in any derivation of a terminal string from the start symbol.

2. We must eliminate **-epsilon productions**, those of the form A -> epsilon for some variable A.

3. We must eliminate **unit productions**, those of the form A B for variables A and B.

7.Explain two representations of the three-address code with appropriate examples.

Ans:

Q: Two representation of 3 address Code. (TAC)

Quadruples:

→ Quadruples are a straight forword representation of T[AC] where each operation is broken down into four feilds: Operator, Operand 1, Operand 2, Result.

Example: $a = b*(-c) + d*(-e)$

Generat 3 Address code:

$t_1 =$ ~~unary~~ unary op (-) c

$t_2 = b * t_1$

$t_3 =$ unary minus e

$t_4 = d * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

Op	Operand1	Operand2	Res[ult]
unary minus	c	–	t_1
*	t_1	b	t_2
unary minus	e	–	t_3
*	t_3	d	t_4
+	t_4	t_2	t_5
=	a	t_5	

(**) খালি space বুঝাচ্ছে op2=(–) এরকম। এটা minus না।

Triple:

→ Triples represent TAC by breaking down operations into 3: Operator, Operand 1 & Operand 2. Results are implicitly represented by the position of the instruction in the sequence.

*** arbitary position to hold the result.

Location	Operator	Op1	Op2
(38)	unary minus	c	–
(34)	*	(38)	b
(23)	uninary minus	e	–
(49)	*	(23)	d
(116)	+	(49)	(34)
(118)	=	a	(116)

(*) Indirect Triple: (এটা included না। Just বুঝার জন্য)

# (loc)	Statement
(0)	(38)
(5)	(34)
(1)	(23)
(9)	(49)
(17)	(116)
(2)	(118)

Pointer এর মত কাজ করে এটা

CFG

Q1:

Explain all the components of a Context Free Grammar (CFG) with an example and describe the importance of CFG in compiler design.

Ans:

A Context-Free Grammar (CFG) is a formal system that **describes a language by defining allowable patterns of strings**. A CFG is a quadruple (4-tuple), that is, a system which consists of 4 elements. We describe a CFG, G as follows:
G = (V, T, P, S), whereThe components of a CFG include:

- **V**ariables (Non-terminal Symbols): These are symbols that can be replaced by groups of other symbols. They are abstract and do not appear in the final strings of the language. They are used to build the structure of the language.

- **T**erminals: These are the actual symbols of the language that appear in the strings. They are the **basic units from which strings are constructed** and cannot be replaced by other symbols.

- **P**roductions (Grammar Rules): These are the rules that define how variables can be replaced by combinations of terminals and other variables. Each rule has a head (the variable being defined) and a body (what it can be replaced with).

- **S**tart Symbol: This is a special variable from which the derivation of strings starts. It represents the whole language and is used to derive all strings in the language.

In the **example** provided in the second image, the CFG consists of:

Variables: S, A
Terminals: a, b
Productions:
S → Abb
A → ε (where ε represents the empty string)
A → aA
A → bA
Start Symbol: S
Derivation example using the given CFG:

S can be replaced by Abb (using the rule S → Abb).
A can be replaced by aA, bA, or ε (using the rules for A).
Through a series of replacements using the rules, we can derive strings like babb from S.

The importance of CFG in compiler design is significant:

- **Parsing:** CFGs are used to write the syntax of programming languages. The compiler uses these grammars to parse and understand the structure of programs written in that language.

- **Syntax Checking:** By defining the syntax of a language strictly, CFGs allow compilers to detect syntax errors in source code.

- **Code Generation:** Once the structure of the code is understood, the compiler can translate the parsed structure into machine code or intermediate representations.

- **Optimization:** Understanding the grammar of a language allows compilers to apply optimization techniques to generate efficient code.

Q2: Why are they called CONTEXT FREE GRAMMAR?

Ans:

The term "context-free" in Context-Free Grammars (CFG) comes from the fact that the rules for producing strings (or sentences) in these grammars do not depend on the context of the non-terminal they are replacing. Each production rule in a CFG applies to a non-terminal symbol irrespective of what symbols precede or follow it.

$S \rightarrow Abb,$ $A \rightarrow \varepsilon$ $A \rightarrow aA$ $A \rightarrow bA.$	OR	$S \rightarrow Abb,$ $A \rightarrow \varepsilon \mid aA \mid$ $bA.$

In the provided example, we can see that the non-terminal `A` can be replaced by `ε` (the empty string), `aA`, or `bA` without considering any surrounding symbols. There is no requirement that `A` be surrounded by particular symbols for the rules to apply. This is the essence of being "context-free."

For instance, the rule:
$A \rightarrow bA$ can be applied to $S \Rightarrow Abb \Rightarrow bAbb$
can be applied to `A` in any context. Whether `A` is at the beginning, in the middle, or at the end of a string, and no matter what the surrounding symbols are, you can replace `A` with `bA`. The rule is applied in the same way every time, regardless of the surrounding 'context'.

Q3:

For a CFG to be in CNF, some conditions are needed to be met. Describe the conditions in your own words.

Ans:
To put a Context-Free Grammar (CFG) in Chomsky Normal Form (CNF), The conditions are:

- **Eliminate Useless Symbols:** Get rid of any symbols (variables or terminals) that cannot be used to derive a string of terminals (characters in the language). A symbol is useful if it can appear in some derivation starting from the start symbol and ending with a string composed only of terminal symbols.

- **Eliminate ε-productions:** Remove production rules that allow a variable to produce an empty string (ε), except for the start symbol, which can produce an empty string if necessary.

- **Eliminate Unit Productions:** Remove production rules where a variable produces another single variable, like $A \to B$.

After these preliminary steps, the grammar should have rules that either:

- **Produce exactly one terminal, like $A \to a$, or**
- **Produce exactly two variables, like $A \to BC$.**

These steps ensure that the grammar is in a standardized form that can be easily processed by algorithms, particularly those used in parsing, like the CYK algorithm. It's crucial to follow the steps in the order listed to prevent reintroducing constructs you've already eliminated.

Q4:

How can you compute the set of Generating Symbols and the set of Reachable Symbols of a grammar? Write the basis and the induction for both the cases.

Ans: Generating symbols are those which can derive a string consisting solely of terminal symbols. **Reachable symbols** are those that can be derived from the start symbol of the grammar.

To compute the set of generating symbols (often denoted as V) of a grammar G = (V, T, P, S), where V is the set of variables, T is the set of terminals, P is the set of production rules, and S is the start symbol.

- **Basis:** Every symbol of T is obviously generating; it generates itself.
- **Induction:** Suppose there is a production A-> α, and every symbol of α is already known to be generating; then A is generating.

To compute the reachable symbols of G, the following induction is performed.

- **Basis:** S is surely reachable.
- **Induction:** Suppose we have discovered that some variable A is reachable. Then for all productions with A in the head, all the symbols of the bodies of those productions are also reachable.

Example:

Consider the grammar:

$$S \to AB \mid a$$
$$A \to b$$

Eliminating non-generating symbols: All symbols but B are generating. If we eliminate B, we must eliminate the production $S \to AB$, leaving the grammar:

$$S \to a$$
$$A \to b$$

Eliminating non-reachable symbols: Only S and a are reachable from S. Eliminating A and b leaves only the production:

$$S \to a$$

Eliminating epsilon Productions

o The strategy is to begin by discovering which variables are "nullable". A variable A is nullable if A* .

o If A is nullable, then whenever A appears in a production body, say B CAD, A might (or might not) derive . So, two versions of the production are possible, B CD or B CAD. Let G = (V, T, P, S) be a CFG. We can find all the nullable symbols of G by the following iterative algorithm.

Basis: If A is a production **epsilon** of G, then A is nullable.
Induction: If there is a production B -> C1C2 …Ck, where each Ci is nullable, then B is nullable. Note that each **Ci must be a variable to be nullable,** so we only have to consider productions with all-variable bodies.

Eliminating Unit Productions:

Basis: (A, A) is a unit pair for any variable A. That is A ->* A by zero steps.
Induction: Suppose it is determined that (A, B) is a unit pair, and B-> C is a production, where C is a variable. Then (A,C) is a unit pair.

Parsing:

Q: Properties of LL1 Grammer.

Ans:

① This conditions specifies that for two different productions of a non-terminal 'A', the first symbol of the strings derived from right hand side should not both start with the same symbol.

$A \to \alpha B$
$A \to \alpha C$

② At most one production can derive the empty string.

$A \to \epsilon$	×	$A \to \epsilon$
$A \to \epsilon C$		$A \to bC$

③ If α can derive 'ε', then no string derived from 'β' should start with a terminal symbol that is in the follow set of 'A'.

α — $A \to Bc$ — β ; Here, α can derive 'ε' and 'β' is 'Bc', wher B can also derive 'ε'. For this, if B derives 'ε' then β essentially starts with c, which is in Follow(A). The parser would not be able to decide between reducing 'A' to 'ε' or continuing with 'Bc' which 'c' is the lookahead symbol

$A \to \epsilon$
$B \to \epsilon$

Q: Illustrate the purpose of Left factoring and the role of look-ahead symbol in predictive parsing

Ans:

The purpose of **left factoring** in grammar is to eliminate the common prefixes of the alternatives of a non-terminal, which helps to make the grammar suitable for predictive (top-down) parsing. This is necessary because top-down parsers, such as LL(1) parsers, cannot decide between two alternatives that begin with the same prefix based on just one lookahead symbol.

Example:

A → αB | αC

In this grammar, both alternatives for A start with the common prefix α. When the parser sees α in the input, it doesn't know whether to predict B or C after α without seeing more of the input.

After applying Left factoring:
A → αA'
A' → B | C
Now, when the parser sees α, it knows to predict A' immediately. Once it has seen α, it can decide between B and C as the next symbol based on the lookahead.

The role of the **lookahead** symbol in predictive parsing is to help the parser decide which production rule to apply. It's the next input symbol that hasn't been consumed yet. In LL(1) parsing, "1" stands for one lookahead symbol.

A → αA' | βA''
When the parser is at A and the lookahead symbol is x, if x can be derived from α, then it predicts αA'. If x can be derived from β, then it predicts βA''. The lookahead symbol helps to disambiguate the choice between multiple alternatives.

Q: Top down parsing techniques cannot work on some classes of cfg's. Justify this statement with appropriate example

Ans:

Top-down parsing techniques can indeed have difficulties with certain classes of context-free grammars (CFGs). The inherent limitations come from the constraints of top-down parsing strategies, particularly when they rely on only one symbol of lookahead (as in LL(1) parsers). Here are a few specific reasons, with examples:

- **Left Recursion:**
  Top-down parsers cannot handle left-recursive productions because they lead to infinite recursion.

  Example:
  A → Aa | b
  In this grammar, a top-down parser would try to match A by expanding it to Aa, which would again require matching A, leading to an infinite loop.

- **Common Prefixes:(Left factoring)**
  When two productions for a non-terminal begin with the same sequence of symbols, an LL(1) parser cannot decide which production to use based on a single lookahead symbol.

  Example:
  A → abc | abd
  Here, the parser cannot decide between the two rules with just one lookahead symbol after seeing ab because it doesn't know whether the next symbol will be c or d.

- **Non-Determinism:**
  A CFG that requires looking ahead more than one symbol to make a decision can't be parsed by an LL(1) parser because it's designed to make parsing decisions with only one symbol of lookahead.

  Example:
  A → aA | aB
  B → b
  In this case, after seeing the initial a, the parser can't decide whether to pursue the A → aA or A → aB production until it sees whether a b follows. But by then, it's already consumed the lookahead a.

- **Ambiguity:**
  Ambiguous grammars, where a single string can be derived in more than one way, cannot be reliably parsed by deterministic top-down parsers.

  Example:
  A → Aa | Aa
  This grammar is ambiguous because there are two ways to derive a string with multiple as, which confuses a top-down parser as to which rule to apply.

- **Backtracking Requirement:**
  Some CFGs require backtracking, which means a parser might need to go back and try different rules if it goes down the wrong path. Non-backtracking top-down parsers, like LL parsers, cannot handle these grammars.

  Example:
  A → aB | aC
  B → b
  C → c
  After seeing the initial a, the parser must decide whether to go to B or C. If the next symbol is not what the parser expects, it would need to backtrack, which LL(1) parsers do not do.

Error

Briefly describe the error recovery techniques that can be applied during parsing and explain how an error recovery technique can develop infinite loop within the procedure itself.

Error recovery techniques during parsing are essential for creating robust compilers that can handle unexpected or incorrect input gracefully. The main objective is for the parser to detect errors, report them to the user, and continue parsing the rest of the input.

Here is a brief description of the error recovery techniques from the text provided:

- **Panic Mode Recovery:** Upon detecting an error, the parser discards input tokens until it encounters a synchronizing token, such as a semicolon or a closing brace. This method is straightforward and avoids infinite loops but can lead to loss of significant input that might contain other errors.

- **Phrase Level Recovery:** This method involves making corrections to the remaining input to allow the parser to continue. Corrections might include deleting or inserting tokens. Care must be taken to ensure that the corrections do not result in an infinite loop, such as repeatedly inserting and then deleting a token.

- **Error Productions:** The grammar is augmented with productions that specifically recognize common errors. This allows the parser to generate appropriate error messages and continue parsing. However, maintaining these error productions can be challenging.

- **Global Correction:** The parser attempts to modify the entire program to find the closest error-free version. This requires a minimal set of token insertions, deletions, and substitutions. Due to its high computational cost, it is not commonly used in practice.

An error recovery technique can lead to an infinite loop if the correction mechanism repeatedly applies the same incorrect correction. For example, if a phrase level recovery strategy incorrectly inserts a token that it then identifies as an error in the next cycle, leading to its deletion, and then re-inserts it in the following cycle, the parser can become trapped in this insert-delete cycle indefinitely. To avoid this, error recovery strategies must be designed to ensure that once an error has been addressed, the same error is not reintroduced by the recovery process.

Question 6. [Marks: 12.5]

a) What is Parse Tree? Describe the relationship between Parsers and Context Free Grammars. **[1+2.5]**

b) Consider the following LL(1) parsing table and show the moves a top-down parser makes for the input string ***accd***. **[3]**

	a	**b**	**c**	**d**	**$**
S	$S \rightarrow aXd$				
X		$X \rightarrow YZ$	$X \rightarrow YZ$	$X \rightarrow \varepsilon$	
Y		$Y \rightarrow b$	$Y \rightarrow \varepsilon$	$Y \rightarrow \varepsilon$	
Z			$Z \rightarrow cX$	$Z \rightarrow \varepsilon$	

c) Construct DFAs from the following regular expressions: **[3+3]**

i. 11*01*01*

ii. 1*01*0(0|1)*

6.a.

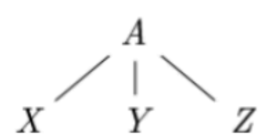
- A parse tree pictorially shows how the start symbol of a grammar derives a string in the language.
- From left to right, the leaves of a parse tree form the *yield* of the tree, which is the string generated or derived from the nonterminal at the root of the parse tree.
- Formally, given a context-free grammar, a parse tree according to the grammar is a tree with the following properties:
  - The root is labeled by the start symbol.
  - Each leaf is labeled by a terminal or by ε.
  - Each interior node is labeled by a nonterminal.
  - If A is the nonterminal labeling some interior node and X, Y, Z are the labels of the children of that node from left to right, then there must be a production $A \rightarrow XYZ$. Here, X, Y, Z each stand for a symbol that is either a terminal or nonterminal. As a special case, if $A \rightarrow \varepsilon$ is a production, then a node labeled A may have a single child labeled ε.

So, an interior node and its children correspond to a production; the interior node corresponds to the head of the production, the children to the body.

A Production in CFG

$A \rightarrow XYZ$

An Interior Node of a Parse Tree



How Syntax Analysis and Semantic Analysis of a compiler are different from each other? Write down the importance of Error Recovery during parsing. **[2+1.5]**

The primary goal of syntax analysis, also known as parsing, is to check whether the source code follows the rules of the programming language's grammar.

Semantic analysis aims to check the meaning of the syntactically correct code. It involves verifying that the operations in the code make sense in terms of the language semantics.

Key Differences

- **Focus:** Syntax analysis focuses on the form of the code, while semantic analysis focuses on the meaning.
- **Output:** The output of syntax analysis is typically an AST(abstract syntax tree), which is used by the semantic analysis phase to perform further checks.
- **Errors:** Syntax analysis deals with syntax errors, whereas semantic analysis deals with semantic errors.
- **Complexity:** Semantic analysis is generally more complex than syntax analysis because it involves more intricate checks that require a deeper understanding of the code's context and meaning.

Question 1. [Marks: 14]

a) Explain the relationships among Token, Lexeme & Pattern from the perspective of a lexical analyzer.

Token:
A **token** is a categorized block of text that is a syntactic unit to the parser in the compilation process. It is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol that represents a type of lexical unit, which can be a keyword, an identifier, or a literal, among others. The optional attribute value provides additional information, such as the specific spelling of an identifier or the value of a numerical constant. For example, for the keyword "while", the token might be represented as `<KEYWORD_WHILE, _>` indicating it is recognized as the keyword "while" without needing an additional attribute.

Pattern:
A **pattern** describes the set of rules or the format that lexemes of a token may take. For instance, the pattern for a keyword token is straightforward—it is the exact sequence of characters that constitute the keyword. For more complex tokens, like identifiers or numbers, the pattern could be a regular expression. This regular expression defines the valid sequences of characters that constitute a lexeme for that token. For example, the pattern for an identifier in many programming languages is a letter followed by zero or more letters or digits.

Lexeme:
A **lexeme** is a specific sequence of characters from the source code that matches a pattern for a token and is identified by the lexical analyzer as an instance of that token. When the lexical analyzer reads the source code, it uses patterns to group the characters into lexemes. For example, the specific identifier "sum" in the code is a lexeme that matches the pattern for an identifier token.

Relationship:
- The **pattern** is a rule or a set of rules that define how lexemes can be formed for a specific token.
- A **lexeme** is an actual sequence of characters from the source code that has been identified as a match for a pattern and is thus recognized as a token.
- A **token** is the abstract representation of a lexeme, categorized for the parser's use, often with an associated attribute value to provide further semantic information.

In the process of lexical analysis, the analyzer reads the source program's characters, groups them into lexemes using patterns, and then categorizes these lexemes into tokens. If a lexeme does not match any pattern, the lexical analyzer will flag an error, indicating that the lexeme is not part of the language's lexicon and is, therefore, not valid in the source code.

- **Sentential form**: It is a string that consists of both terminals and non-terminals (or just terminals) that can be derived from the start symbol S through a series of productions. This means we start with S and apply the grammar's rules until we get the string. If you can get to a string by repeatedly replacing symbols according to the grammar's rules, starting from S, then that string is a sentential form.

A grammar is said to be ambiguous if there is at least one string of terminal symbols that can be generated by the grammar in more than one way, meaning it has more than one distinct parse tree or derivation. In other words, if a single string can be interpreted in multiple ways according to the rules of the grammar, leading to multiple different structures or meanings without changing the sequence of terminals, then the grammar is considered ambiguous. This ambiguity can make it difficult to understand or predict the grammar's behavior, especially in contexts like programming language compilers, where a clear, unambiguous structure of sentences (programs) is crucial for correct interpretation and execution.