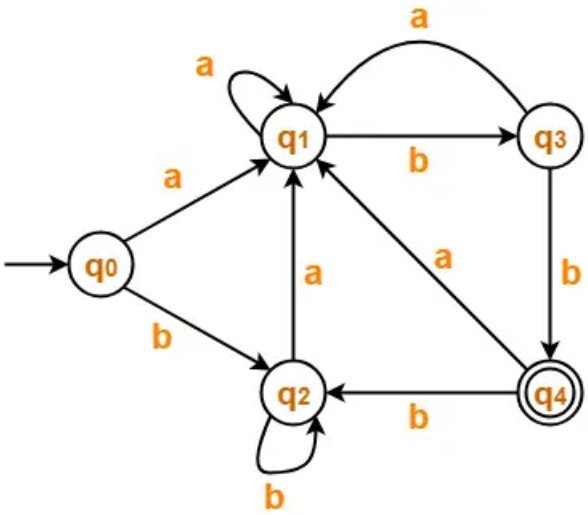
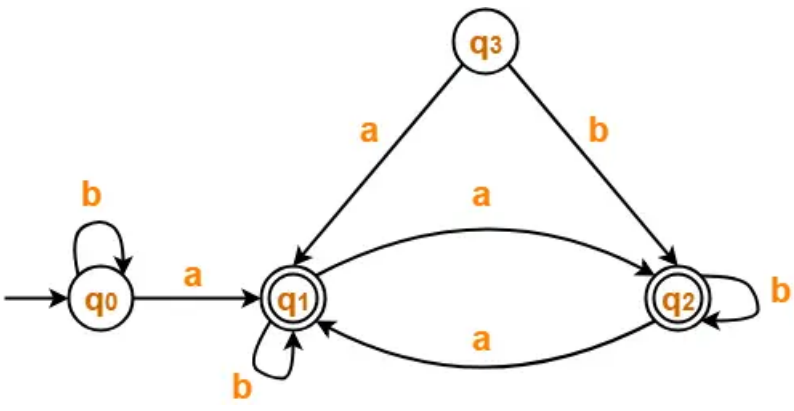
COMPILER DESIGN UNIT – 1 QUESTIONS

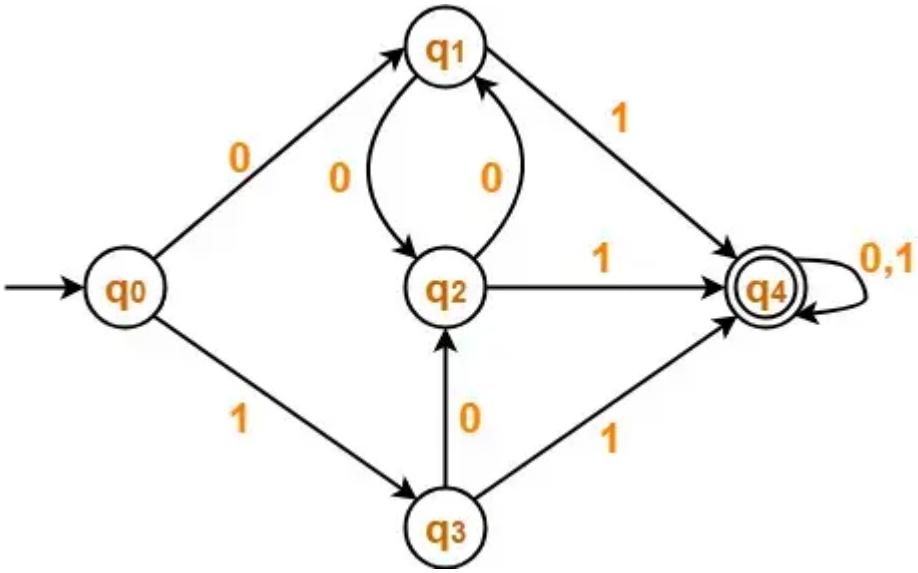
1. What is Compiler?
2. What are Translators? Why do we need Translators?
3. What does a Compiler do?
4. What is assembler?
5. What is assembly language?
6. What are the different phases of compiler?
7. What are the types of phases in compiler?
8. What are compiler construction tools. Explain any three.
9. What phases of compiler comes under front-end and back-end? Why are they called so?
10. Explain all phases of compiler in brief?
11. Draw a neat diagram of compiler with all its phases.
12. What are cross compiler? Explain with example.
13. What is bootstrapping? Explain with help of a diagram? Give example.
14. What is quick and dirty compiler?
15. What is a lexical token?
16. What is a symbol table in compiler? Explain with the help of an example.
17. Explain the role of Lexical Analysis in detail with help of a diagram.
18. What is FLEX?
19. What is YACC?
20. What is Input Buffering? What are is disadvantages?
21. Design a FA from given regular expression 10 + (0 + 11)0* 1.
22. Design a NFA from given regular expression 1 (1* 01* 01*)*.
23. Construct the FA for regular expression 0*1 + 10.
24. Minimize the given DFA-



25. Minimize the given DFA-



26. Minimize the given DFA-



PYQ:

27. Generate a pattern matcher which can recognise the following patters: 01* and 1*0

28. Write short note on LEX and LEX tool.

29. What is Single pass and Multi pass Compiler?

30. What are Pattern in Lexical Analyser?

31. What is interpreter?

32. Difference between compiler and interpreter.

COMPILER DESIGN UNIT – 2 QUESTIONS

1) What is the Role of the parser?
2) What is Annotated Parse Tree?
3) What is handle pruning? With example.
4) Discuss Context Free Grammar taking a suitable example.
5) A left recursive grammar cannot be LL(1). justify.
6) Discuss all the three methods used to perform LR parsing. Which is most powerful and why?

7) How to remove ambiguity from grammar? Remove ambiguity from the following grammar:

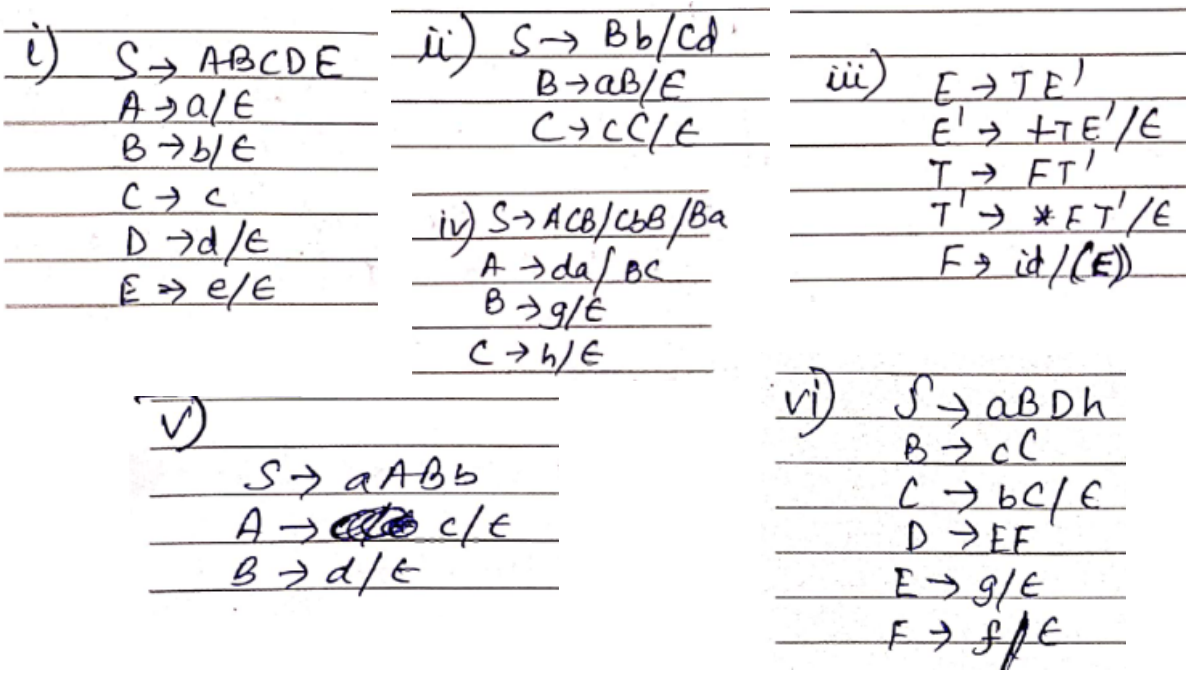
E -> E+E/E*E/(E)/id

8) How to remove left recursion from grammar? Remove Left Recursion from the following grammar:

E-> E+&/Q

9) How to convert NFA to DFA?

10) Find the FIRST and FOLLOW of the following grammar:



11) Draw the parse tree using LL(1) Parser, string is id*id+id:

E -> E+E/E*E/(E)/id

12) Draw the parse tree using LL(1) Parser, string is abd:

S-> A
A-> aB/Ad
B-> b
C-> g

13) Draw the parse tree using LL(1) Parser:

S-> aSbS/bSaS/null

14) Perform Shift Reduce Parsing on the following Grammar for input a1-(a2+a3):

S → S+S
S → S-S
S → (S)
S → a

15) Consider the following Grammar and construct the Operator Precedence Parsing. Given string is id+id*id:

E -> EAE/id
A-> +/*

16) Make a parse tree using LR(0) parsing with following grammar:

a) E-> BB
B-> CB/d
input = ccdd$

b) S-> AA
A-> aA/b

17) Check whether the grammar is LR(0) or SLR(1) or not.

E-> T+E/T
T-> id

18) Make the parse tree for grammar:

E-> BB
B-> CB/d

Using CLR(1) and LALR(1)

PYQ:

19) Improve the grammar by removing left recursion:

S-> AaB/b
A-> Aab/Aba/a
B-> a

20) Calculate and FIRST and FOLLOW for non-terminal then create predictive parsing table and prove that this grammar is not LL(1).

X-> 0Y1/1Z2
Y-> 2/null
Z-> 1Y2/1/null

21) Consider the Grammar:

S-> L=R/R
L-> *R/id
R-> L

Design LALR Parser for this Grammar.

All Answers: Unit – 1

1. Ans.

A compiler is **a computer program that helps in translating the computer code from one programming language into another language**

2. Ans.

A translator is **a program that converts instructions written in source code to object code or from high-level to machine language**.

3. Ans.

A compiler is a software that **converts the source code to the object code**. In other words, we can say that it converts the high-level language to machine/binary language. Moreover, it is necessary to perform this step to make the program executable. This is because the computer understands only binary language.

4. Ans.

For every platform (Hardware + OS) we will have an assembler. They are not universal since for each platform we have one. The output of assembler is called object file. Its translates assembly language to machine code.

5. Ans.

An assembly language is **a type of low-level programming language that is intended to communicate directly with a computer's hardware**. Unlike machine language, which consists of binary and hexadecimal characters, assembly languages are designed to be readable by humans.

6 & 7. Ans.

(i) Analysis Phase – An intermediate representation is created from the give source code :
1. Lexical Analyser
2. Syntax Analyser
3. Semantic Analyser

Lexical analyser divides the program into “tokens”, Syntax analyser recognizes “sentences” in the program using syntax of language and Semantic analyser checks static semantics of each construct.

(ii) Synthesis Phase – Equivalent target program is created from the intermediate representation. It has three parts:
1. Intermediate Code Generator
2. Code Optimizer
3. Code Generator

Intermediate Code Generator generates “abstract” code, Code Optimizer optimizes the abstract code, and final Code Generator translates abstract intermediate code into specific machine instructions.

8. Ans.

Compiler Construction Tools
Compiler construction involves specialized tools that simplify the creation of compilers, helping in various phases such as lexical analysis, syntax analysis, and code generation.

1. Parser Generators

- **Purpose:** Generate parsers from context-free grammar.
- **Use:** Automates syntax analysis, saving time and reducing complexity.
- **Examples:** Yacc, Bison, ANTLR.

2. Scanner Generators

- **Purpose:** Generate lexical analyzers from regular expressions.
- **Use:** Tokenizes source code into meaningful symbols.
- **Examples:** Lex, Flex, JFlex.

3. Syntax Directed Translation Engines

- **Purpose:** Produce intermediate code from parse trees.
- **Use:** Associates semantic actions with syntax rules to generate intermediate code.
- **Examples:** SableCC, ANTLR.

4. Automatic Code Generators

- **Purpose:** Convert intermediate code to machine code.
- **Use:** Translates high-level instructions to machine-specific code.
- **Examples:** GCC, LLVM.

5. Data-flow Analysis Engines

- **Purpose:** Optimize code by analyzing data flow.
- **Use:** Improves code efficiency by identifying redundancies.
- **Examples:** LLVM optimization passes, SSA form.

6. Compiler Construction Toolkits

- **Purpose:** Provide integrated routines for building compiler components.
- **Use:** Streamlines compiler development with reusable tools.
- **Examples:** LLVM, ANTLR.

9. Ans.

Compiler Phases

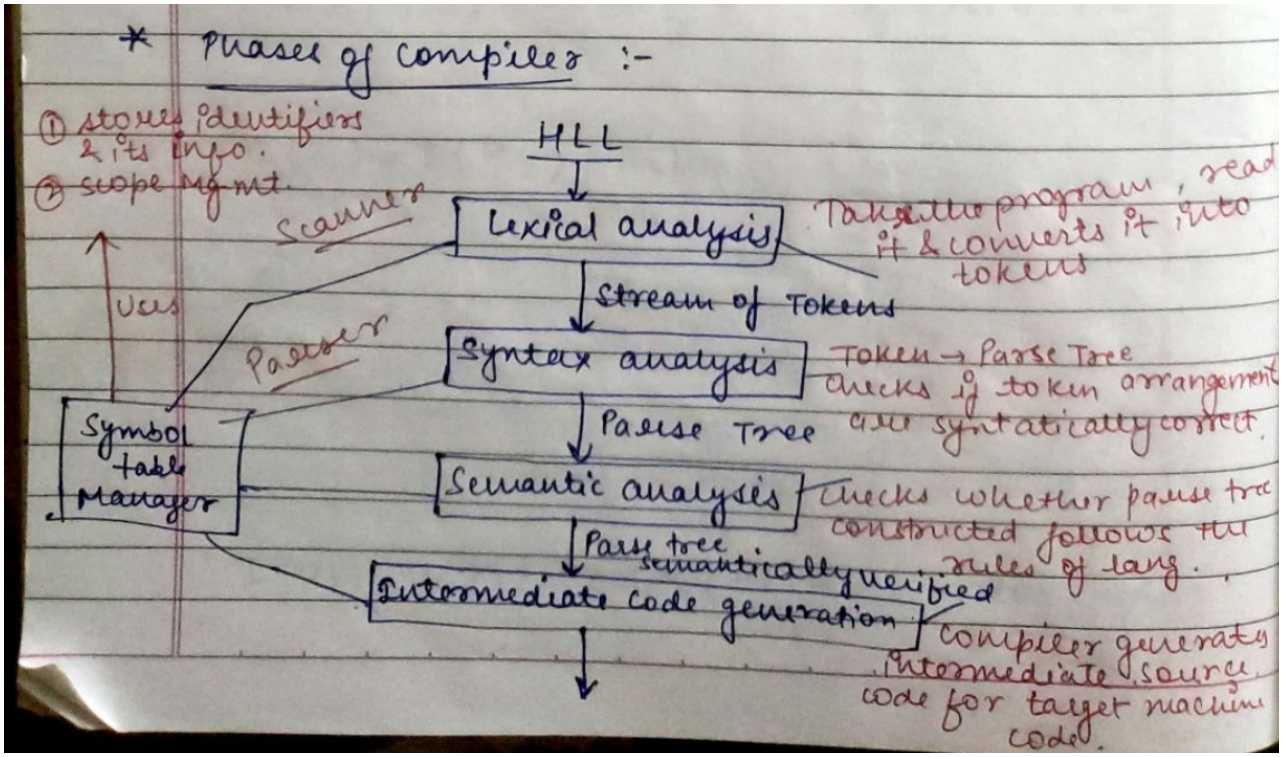
- **Analysis Phase: (Front-End)**
    - Converts source code to an intermediate representation (IR) through lexical, syntactic, and semantic analysis.
    - Tools: Scanner Generators (Lex), Parser Generators (Yacc), Translation Engines.

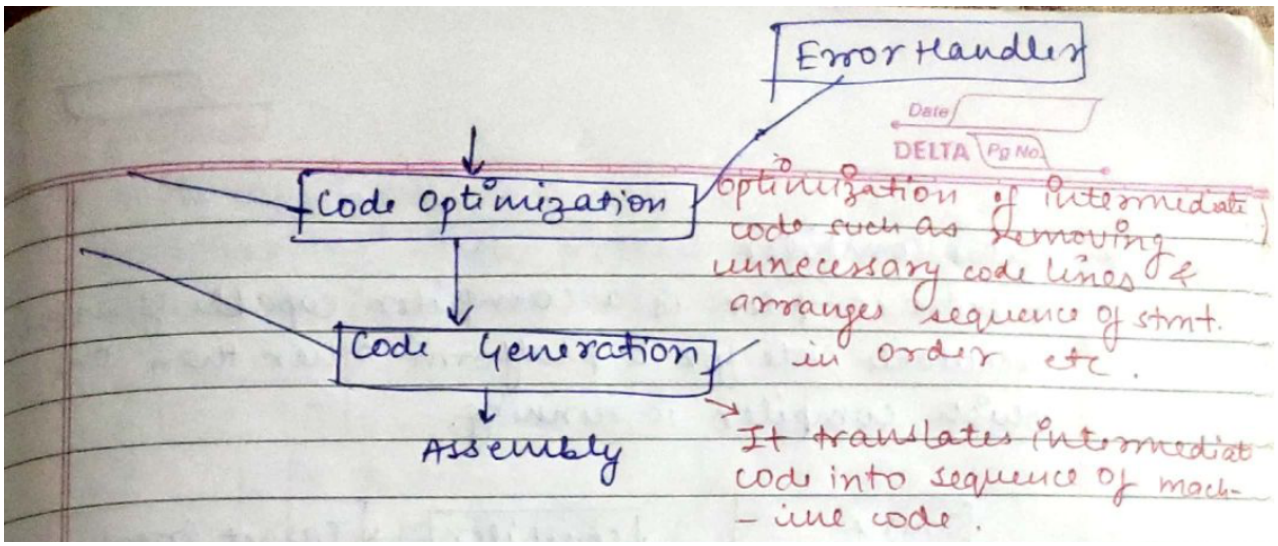
- **Synthesis Phase: (Back-End)**
    - Translates IR into machine code, applying optimizations.
    - Tools: Data-flow Engines, Automatic Code Generators.

10. Ans.

Refer to answer 6 & 7.

11. Ans.



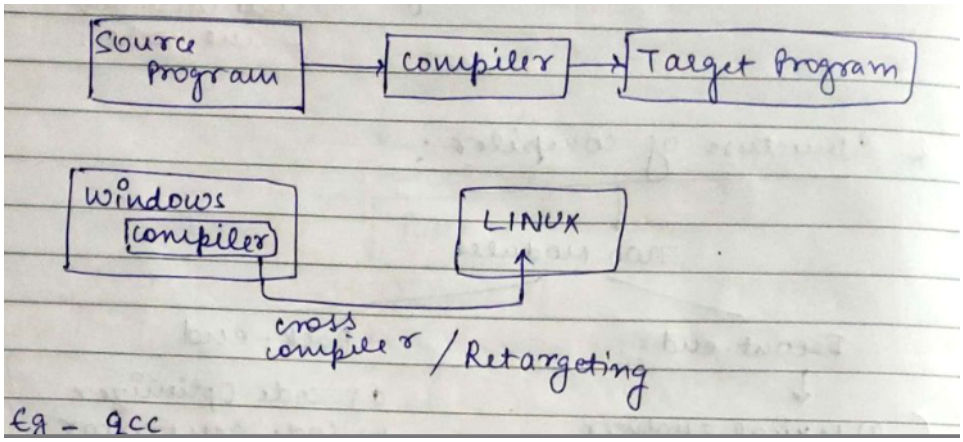


12. Ans.

A **Cross Compiler** is a type of compiler that runs on one machine (called the host machine) but produces executable code for a different machine (called the target machine).

For example, if a cross compiler is running on **Machine A** (host machine) but generates code that can execute on **Machine B** (target machine), it allows developers to build software for different architectures or platforms without requiring the target hardware during development.

Widely used in embedded systems development, where the target machine might be a different hardware architecture (e.g., ARM) from the development environment (e.g., x86).



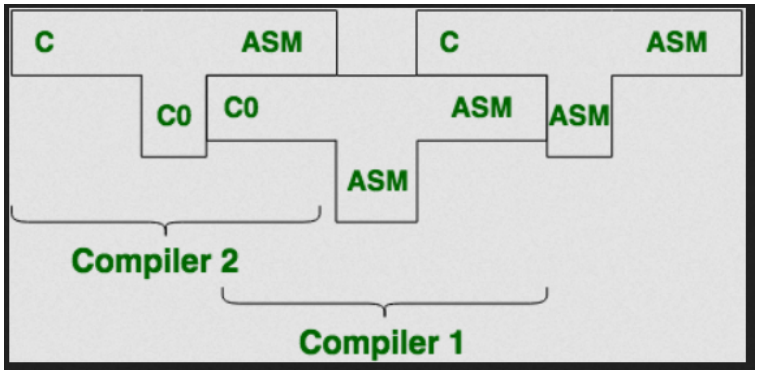
Advantages of Cross Compilers

1. **Targeting Different Architectures:** Allow development of code for a platform different from the host machine, essential for embedded systems.
2. **Development Flexibility:** Enable use of powerful host machines while targeting less capable hardware.
3. **Time Efficiency:** Reduce build times by compiling on a powerful machine before deploying to the target.
4. **Enhanced Debugging:** Improve debugging capabilities by allowing testing in a familiar environment.
5. **Support for Multiple Platforms:** Facilitate development for various platforms, enabling code reusability.
6. **Legacy Support:** Maintain and update applications for older systems where native compilers may not be available.

13. Ans.

Bootstrapping is a process in which simple language is used to translate more complicated program which in turn may handle for more complicated program. This complicated program can further handle even more complicated program and so on. Writing a compiler for any high level language is a complicated process. It takes lot of time to write a compiler from scratch.

First we write a compiler for a small of C in assembly language. Then using with small subset of C i.e. C0, for the source language c the compiler is written. Finally we compile the second compiler. using compiler 1 the compiler 2 is compiled.



we get a compiler written in ASM which compiles C and generates code in ASM.

14. Ans.

A **Quick and Dirty Compiler** is a simple, often hastily developed compiler that prioritizes getting basic functionality working over optimization and robustness. These compilers are usually created to achieve a working solution rapidly, often sacrificing performance, error handling, and comprehensive features for the sake of speed in development.

15. Ans.

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages

Lexeme: The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme. eg- “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;”

16. Ans.

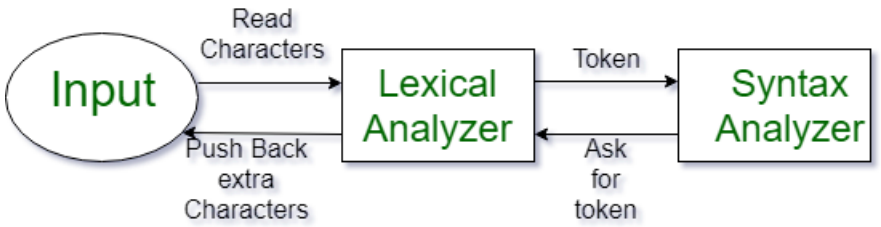
The **Symbol Table** is a crucial data structure used by the compiler during both the analysis and synthesis phases. It stores important information about the identifiers (such as variables, functions, objects, etc.) in the source code, including their names, types, scopes, and memory locations.

Why is the Symbol Table Important?

- **Fast Access:** Enables the compiler to find identifiers and their attributes quickly during parsing, semantic analysis, and code generation.
- **Error Checking:** Helps in type checking, ensuring that variables and functions are used correctly.
- **Efficient Code Generation:** Facilitates efficient code generation by storing information about memory locations and variable types.

17. Ans.

A **Lexical Analyzer** (also called a scanner) is the first phase of a compiler. It reads the source code character by character and converts it into meaningful tokens, which are the basic building blocks for syntax analysis. The lexical analyzer simplifies the source code for further processing by breaking it down into tokens.



What does the Lexical Analyzer do?

- **Input:** It takes raw source code as input (a stream of characters).
- **Output:** It generates a sequence of tokens, each representing a specific category (e.g., keywords, operators, identifiers, literals).
- **Role:** It removes unnecessary details like whitespace and comments, leaving only the core meaningful symbols.

Step-by-Step Process of Lexical Analysis:

1. **Reading Source Code:**
    - The lexical analyzer reads the source code character by character from left to right.
    - It groups characters into tokens, such as keywords (`int`, `while`), identifiers (`variableName`), operators (`+`, `=`), and literals (`123`, `"string"`).
2. **Tokenization:**
    - The primary job of the lexical analyzer is **tokenization**. Each token is a sequence of characters representing a meaningful unit in the programming language.
    - **Example:**
        - Source code: `int x = 10;`
        - Tokens: `int` (keyword), `x` (identifier), `=` (operator), `10` (literal), `;` (delimiter).
3. **Pattern Matching Using Regular Expressions:**
    - The lexical analyzer matches sequences of characters against predefined patterns (usually regular expressions) to identify valid tokens.
    - **Example Patterns:**
        - Identifiers: `[a-zA-Z_][a-zA-Z0-9_]*`
        - Integers: `[0-9]+`
        - Operators: `+`, `-`, `=`, etc.
4. **Removing Whitespace and Comments:**
    - Whitespace (spaces, tabs, newlines) and comments are not relevant to syntax analysis, so they are ignored by the lexical analyzer.
5. **Error Handling:**
    - The lexical analyzer must detect **lexical errors**, such as invalid characters or improperly formed tokens.
    - Example of errors:
        - Unrecognized characters like # in a language that doesn't support it.
        - Malformed numbers or identifiers, like `123abc` (if not allowed).
6. **Symbol Table Insertion:**
    - The lexical analyzer often interacts with the **symbol table**, particularly when it encounters identifiers (variable names, function names, etc.).
    - When a new identifier is found, the lexical analyzer may add it to the symbol table with information like its type and scope.
7. **Return Tokens to Parser:**

- After identifying and creating tokens, the lexical analyzer sends them to the **parser** (the next phase) for syntactic analysis.
- Each token consists of:
    - **Token name/type:** Classifies the token (e.g., `KEYWORD`, `IDENTIFIER`, `NUMBER`).
    - **Token value:** The actual string/character sequence from the source code (e.g., `int`, `x`, `10`).
    - **Location (optional):** The line number or position in the source code for error reporting.

Example of Lexical Analysis Process:

Given the following source code snippet:

```
int a = 5;
```

The lexical analyzer performs the following steps:

1. Reads characters: `i`, `n`, `t`, recognizes `int` as a keyword.
2. Reads whitespace and skips it.
3. Reads `a`, recognizes it as an identifier.
4. Reads `=`, recognizes it as an assignment operator.
5. Reads `5`, recognizes it as a numeric literal.
6. Reads `;`, recognizes it as a delimiter.
7. Generates tokens for the parser:
    - `KEYWORD (int)`
    - `IDENTIFIER (a)`
    - `ASSIGNMENT (=)`
    - `LITERAL (5)`
    - `DELIMITER (;)`

18. Ans.

Flex (Fast Lexical Analyzer Generator)
FLEX (fast lexical analyzer generator) is a tool/computer program for generating lexical analyzers (scanners or lexers) written by Vern Paxson in C around 1987. It is used together with Berkeley Yacc parser generator or GNU Bison parser generator.

Flex and Bison both are more flexible than Lex and Yacc and produces faster code. Bison produces parser from the input file provided by the user. The function yylex() is automatically generated by the flex when it is provided with a .l file and this yylex() function is expected by parser to call to retrieve tokens from current/this token stream.

19. Ans.

YACC is an LALR parser generator developed at the beginning of the 1970s by Stephen C. Johnson for the Unix operating system. It automatically generates the LALR(1) parsers from formal grammar specifications. YACC plays an important role in compiler and interpreter development since it provides a means to specify the grammar of a language and to produce parsers that either interpret or compile code written in that language.

20 Ans.

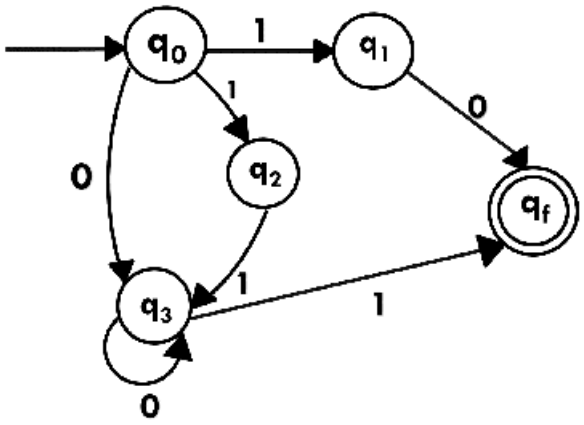
Input Buffering is a technique used in compiler design to improve the efficiency of reading and processing characters from the source code during the **lexical analysis** phase. It is used by the lexical analyzer (or scanner) to avoid reading characters from the source program one by one, which can be slow and inefficient. Instead, the source code is loaded into a buffer, typically in larger chunks, allowing the lexical analyzer to process characters more quickly.

Advantages of Input Buffering:

- **Reduces I/O Operations:** Instead of reading characters one by one from disk or memory, the source code is read in chunks, which reduces costly I/O operations.
- **Faster Lexical Analysis:** By loading multiple characters at once, the lexical analyzer can process tokens more quickly.
- **Efficient Memory Management:** The dual-buffer approach allows continuous processing without waiting for input.

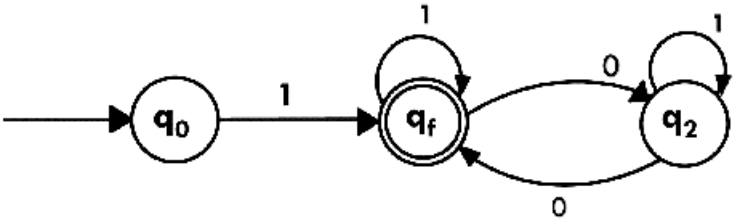
21. Ans.

10 + (0 + 11)0* 1.

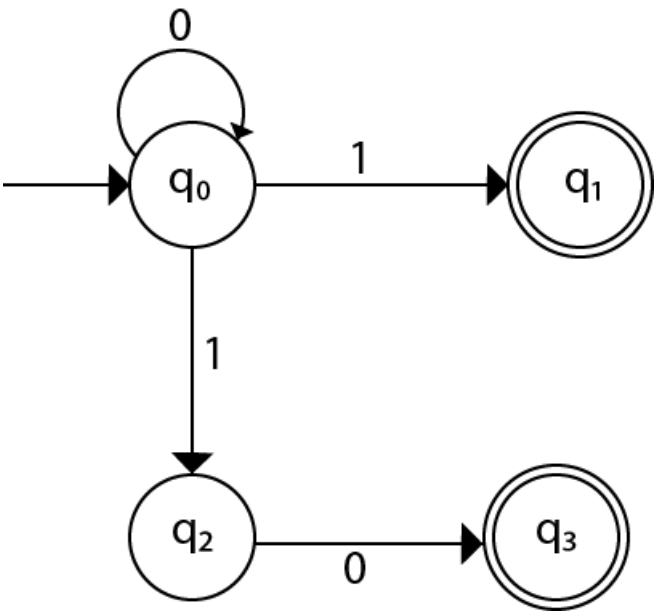


22. Ans.

1 (1* 01* 01*)*.



23. Ans



24. Ans.

	a	b
→q0	q1	q2
q1	q1	q3
q2	q1	q2
q3	q1	*q4
*q4	q1	q2

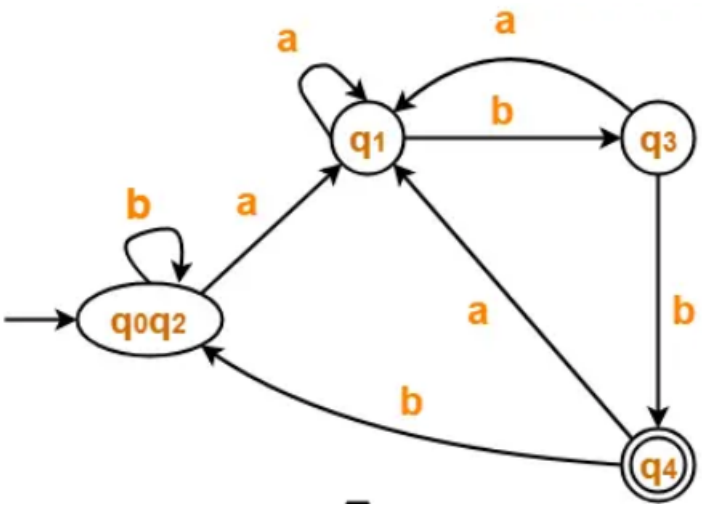
Now using Equivalence Theorem, we have-

P_0 = { q_0 , q_1 , q_2 , q_3 } { q_4 }

P_1 = { q_0 , q_1 , q_2 } { q_3 } { q_4 }

P_2 = { q_0 , q_2 } { q_1 } { q_3 } { q_4 }

P_3 = { q_0 , q_2 } { q_1 } { q_3 } { q_4 }



28. Ans.

Single-Pass Compiler

- **Definition:** Processes source code in one pass, generating code immediately.
- **Characteristics:**
  - **One Pass:** Reads and compiles the code once.
  - **Immediate Code Generation:** Generates target code while scanning.
- **Advantages:**
  - Faster compilation and lower memory usage.
- **Disadvantages:**
  - Limited error detection and complex implementations for complex languages.
- **Example:** Used for simple programming languages.

Multi-Pass Compiler

- **Definition:** Processes source code in multiple passes, analyzing and generating code separately.
- **Characteristics:**
  - **Multiple Passes:** Reads and processes the code several times.
  - **Intermediate Representation:** Generates intermediate code for further analysis.
- **Advantages:**
  - Thorough analysis and optimizations; easier implementation of complex features.
- **Disadvantages:**
  - Slower compilation and higher memory usage.
- **Example:** Used for complex programming languages like C, C++, or Java.

29. Ans.

LEX is a powerful tool used for generating **lexical analyzers** or **scanners**, which are essential components of compilers and interpreters. It is a type of **scanner generator** that takes a set of regular expressions and generates C code for a lexical analyzer that recognizes the specified patterns in the input text.

LEX is an essential tool in the field of compiler design and programming language development, enabling the efficient creation of lexical analyzers through the use of regular expressions and automated code generation. It plays a crucial role in processing source code, facilitating the development of complex programming languages and applications

Advantages of LEX:

- **Ease of Use:** LEX simplifies the process of creating lexical analyzers by automating token generation.
- **Regular Expressions:** It uses regular expressions, which are a powerful and concise way to describe patterns.
- **Integration:** Easily integrates with YACC for full compiler design.

Disadvantages of LEX:

- **C Language Dependency:** LEX generates C code, making it less suitable for projects that use other programming languages.
- **Learning Curve:** Although powerful, understanding regular expressions and how to effectively use LEX can require some time and practice.

30. Ans.

In the context of a lexical analyzer, a **pattern** refers to a specific sequence or structure that defines a category of tokens in the source code. Patterns are typically described using **regular expressions**, which allow the lexical analyzer to recognize and classify different types of input characters or strings.

Ex. int x = 10;
if (x > 5) {
  printf("x is greater than 5");
}

Patterns:

- **Keyword (int, if):** `int|if`
- **Identifier (x):** `[a-zA-Z_][a-zA-Z0-9_]*`
- **Integer Literal (10):** `[0-9]+`
- **Operator (=, >):** `=|>`
- **String Literal ("x is greater than 5"):** `"[^"]*"`

31. An **interpreter** is a program that executes code line by line, translating high-level code into machine code at runtime. This allows for immediate execution but may be slower than compiled programs, as each line is processed every time the code runs.

32. **Differences between a compiler and an interpreter**:

- **Execution**: A compiler translates the entire code into machine code before execution, creating an executable file. An interpreter translates and runs code line by line.

- **Speed**: Compiled code generally runs faster since it's pre-translated, while interpreted code can be slower due to real-time translation.
- **Error Handling**: Compilers catch all syntax errors before execution, while interpreters stop at the first error encountered during execution.
- **Usage**: Languages like C and C++ use compilers, while languages like Python and JavaScript typically use interpreters.

All Answers: Unit – 2:

1. Ans.

In the syntax analysis phase of a compiler, the parser plays a crucial role in ensuring that the tokens generated by the lexical analyzer conform to the syntactic rules of the programming language. Here’s a concise overview of the syntax analysis process and the types of parsers:

The parser plays a crucial role in the syntax analysis phase of a compiler. Its primary responsibilities include:

1. **Token Verification:**
    - The parser checks whether the sequence of tokens generated by the lexical analyzer adheres to the syntactic rules of the programming language.
2. **Grammar Compliance:**
    - It verifies that the token string can be generated by the language's grammar, ensuring that the code follows the correct structure.
3. **Error Detection:**
    - The parser identifies and reports syntax errors in the source code, providing feedback on where and what kind of errors occur.
4. **Parse Tree Construction:**
    - It generates a parse tree (or syntax tree) that represents the hierarchical structure of the token sequence, illustrating the relationship between different components of the code.
5. **Derivation Process:**
    - The parser derives the input string from the grammar using a systematic approach (top-down or bottom-up), translating the linear sequence of tokens into a structured format.
6. **Intermediate Code Generation:**
    - The parse tree produced by the parser serves as a foundation for generating intermediate code, which will be used in subsequent compilation phases.

2. Ans.

The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree.

Features of Annotated Parse Trees

- High level specification
- Hides implementation details
- Explicit order of evaluation is not specified

3. Ans.

Removing the children of the left-hand side non-terminal from the parse tree is called **Handle Pruning**.

A rightmost derivation in reverse can be obtained by handle pruning.

In compiler design, handle pruning is the technique used to optimise the parsing process of a grammar. In order to replace nonterminal symbols to handle, parsing algorithms apply reduction. In LR parsing handle is a substring of the right side of the production rule. Thus, it can be reduced to nonterminal on the left-side of the rule.

4. Ans.

Context Free Grammar is formal grammar, the syntax or structure of a formal language can be described using context-free grammar (CFG), a type of formal grammar. The grammar has four tuples: (V,T,P,S).

V - It is the collection of variables or non-terminal symbols.
T - It is a set of terminals.
P - It is the production rules that consist of both terminals and non-terminals.
S - It is the starting symbol.

A grammar is said to be the Context-free grammar if every production is in the form of :

G -> $(V \cup T)^*$, where G ϵ V

eg. S-> aS

5. Ans.

First of all you are finding FIRST and FOLLOW over the grammar in which you have removed left recursion. Therefore surely if you try to create LL(1) parsing table there won't be any 2 entries as left recursion is removed and grammar is unambiguous.

Grammar[S->SA|A A->a] is not LL(1) as left recursion exists. To prove it by constructing LL(1) parsing table you need to find FIRST and FOLLOW on this grammar only without modifying it.

6. Ans.

1. Simple LR (SLR) Parsing

- **Description:**
  - SLR parsing uses a simple approach where it constructs a parsing table based on the grammar and uses a single lookahead token to make parsing decisions.
- **Table Construction:**
  - The SLR parsing table is built using the states derived from the items of the grammar and follows a standard set of rules.
- **Advantages:**
  - Easier to implement than other LR parsing methods.
  - Sufficient for many grammars, especially those without ambiguities or conflicts.

- **Disadvantages:**
    - Limited power; cannot handle all grammars, particularly those requiring more lookahead to resolve conflicts.

2. Look-Ahead LR (LALR) Parsing

- **Description:**
    - LALR parsing improves on SLR by using the same states as SLR but combines similar states based on the lookahead symbol.
- **Table Construction:**
    - The LALR parsing table is constructed by merging states from the SLR parsing table, allowing more context by considering the lookahead token.
- **Advantages:**
    - More powerful than SLR; can handle a wider range of grammars, including some that produce conflicts in SLR parsing.
    - Efficient in terms of space, as it often results in smaller parsing tables.
- **Disadvantages:**
    - Still cannot handle all possible conflicts, but it is much more effective than SLR.

3. Canonical LR Parsing

- **Description:**
    - Canonical LR parsing is the most powerful of the three methods, using a complete set of states derived from the items of the grammar, which include the full context of the grammar.
- **Table Construction:**
    - The canonical LR parsing table is constructed using all possible states of the grammar, allowing it to handle any context-free grammar with no conflicts.
- **Advantages:**
    - Can handle the largest class of grammars, including all LR(1) grammars, which are grammars that can be parsed with one lookahead token.
    - Capable of resolving ambiguities and conflicts effectively.
- **Disadvantages:**
    - More complex to implement than SLR and LALR.
    - Generally results in larger parsing tables, requiring more memory.

Most Powerful Method

Canonical LR parsing is considered the most powerful of the three methods because it can handle the most complex grammars without conflicts. It incorporates a complete set of states derived from the grammar, providing a comprehensive understanding of the syntax being parsed. While SLR and LALR have their advantages in terms of simplicity and efficiency, they are limited in the range of grammars they can process effectively. Canonical LR parsing's ability to resolve conflicts makes it suitable for more complex programming languages, making it the preferred choice in many compiler implementations.

7. Ans.
The given grammar:

E -> E + E
E -> E * E
E -> (E)
E -> id

is ambiguous because expressions can be parsed in multiple ways, especially with operators like `+` and `*`, which can lead to different interpretations of the order of operations.

Identify the Ambiguities: The ambiguities in this grammar arise from the lack of clear precedence and associativity rules for the operators.

Introduce Non-terminals for Different Levels of Precedence: We can create separate non-terminals to reflect operator precedence. In this case, multiplication (`*`) has higher precedence than addition (`+`).

Define the Grammar with Clear Precedence and Associativity: We'll create a new grammar that respects the precedence and left associativity of the operators.

Revised Grammar :
E -> E + T | T
T -> T * F | F
F -> (E) | id

8. Ans.

Left recursion in a grammar occurs when a non-terminal directly or indirectly references itself as the leftmost symbol in one of its productions. In the given grammar:
E -> E + aT | T

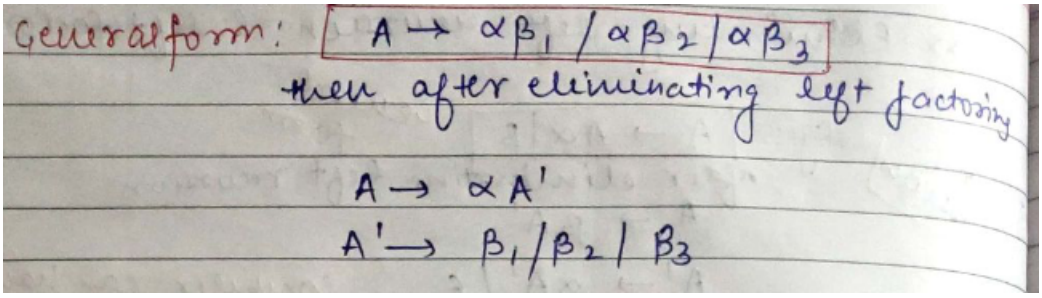
Steps to Remove Left Recursion

1. **Identify Left Recursive Productions:**
    - The left recursive production is `E -> E + aT`.
2. **Introduce a New Non-terminal:**
    - We will create a new non-terminal to help remove the left recursion. Let's call this new non-terminal `E'`.
3. **Reformulate the Productions:**
    - The left recursion can be removed by restructuring the productions as follows:
    - The non-recursive part will be moved to the original non-terminal.
    - The recursive part will be placed in the new non-terminal.

Transformed Grammar

The left-recursive grammar can be transformed into a right-recursive grammar like this

E -> T E'
E' -> + aT E' | ε



9. Ans.

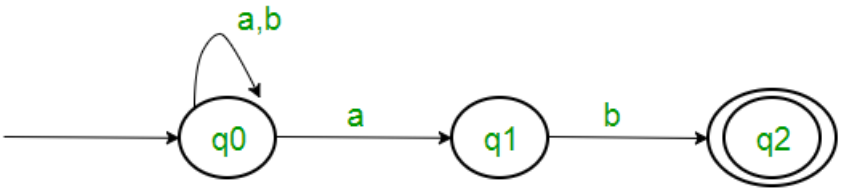


Figure 1

Following are the various parameters for NFA. Q = { q0, q1, q2 } ? = (a, b) F = { q2 }

State	a	b
q0	q0,q1	q0
q1		q2
q2		

DFA Table:

State	a	B
q0	{q0,q1}	q0
{q0,q1}	{q0,q1}	{q0,q2}
{q0,q2}	{q0,q1}	q0

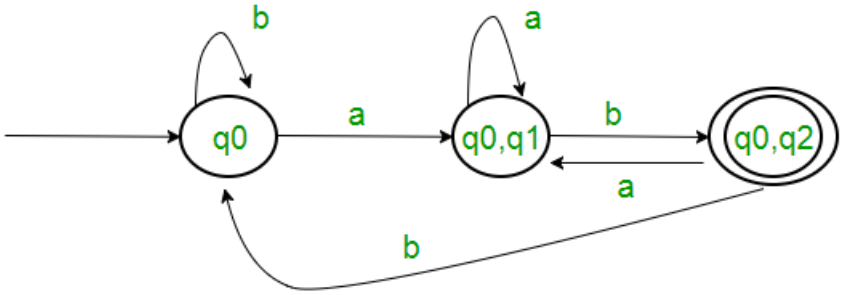


Figure 2

10. Ans.

i)

	FIRST	FOLLOW
S	{a, b, c}	{$}
A	{a, ∈}	{b, c}
B	{b, ∈}	{c}
C	{c}	{d, e, $}
D	{d, e}	{e, $}
E	{e, ∈}	{$}

ii)

	First	Follow
S	{a, b, c, d}	{$}
B	{a, ∈}	{b}
C	{c, ∈}	{d}

iii)

	FIRST	FOLLOW
E	{id, (}	{$,)}
E'	{+, ∈}	{$,)}
T	{id, (}	{+, $,)}
T'	{*, ∈}	{+, $,)}
F	{id, (}	{*, +, $,)}

	FIRST	FOLLOW
S	{d, g, h, ∈, b, n}	{$}
A	{d, g, h, ∈}	{h, g, $}
B	{g, ∈}	{$, a, h, g}
C	{h, ∈}	{g, $, b, n}

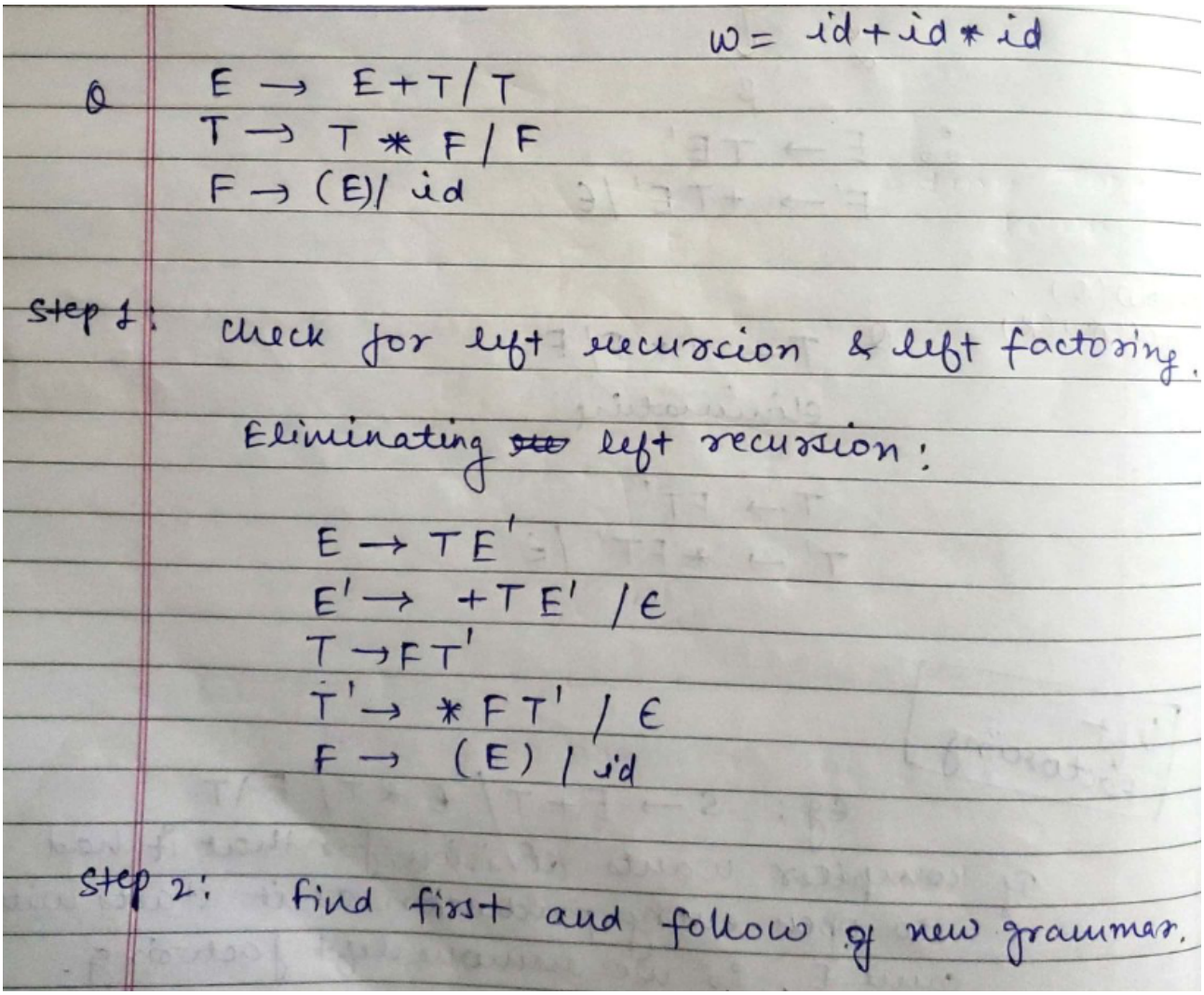
	First	Follow
S	{a}	{$}
A	{c, ∈}	{d, b}
B	{d, ∈}	{b}

vi)

S → aBDh
B → cC
C → bC / ∈
D → EF
E → g / ∈
F → f / ∈

	First	Follow
S	{a}	{$}
B	{c}	{g, f, h}
C	{b, ∈}	{g, h, f}
D	{g, ∈, f}	{h}
E	{g, ∈}	{f, h}
F	{f, ∈}	{h}

11. Ans.


w = id + id * id
Q
E → E+T / T
T → T * F / F
F → (E) / id
Step 1: Check for left recursion & left factoring.
Eliminating left recursion:
E → TE'
E' → +TE' / ϵ
T → FT'
T' → *FT' / ϵ
F → (E) / id
Step 2: find first and follow of new grammar.

	first	follow
E	(, id	), $
E'	+, ϵ	), $
T	(, id	+,), $
T'	*, ϵ	+,), $
F	(, id	*, +,), $

Step 3: Parsing Table generation

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

Step 4: Stack Implementation

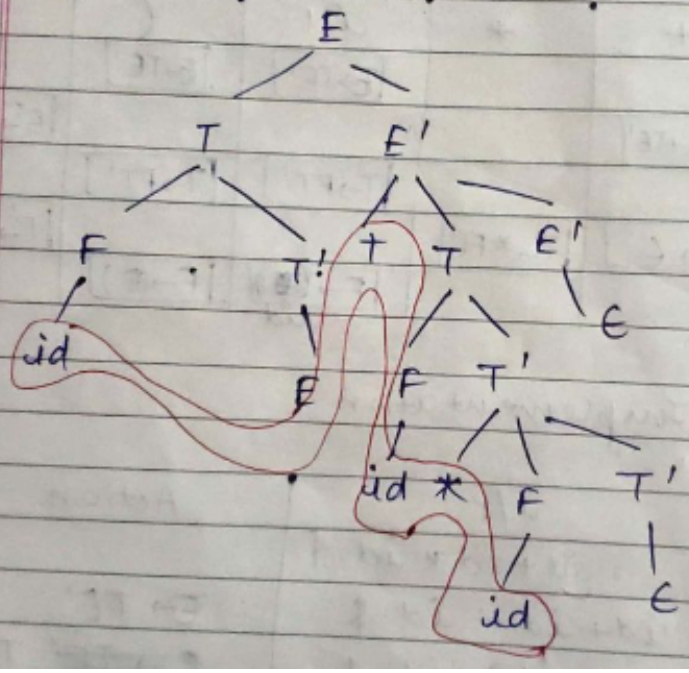
Not in case of Top-down

Stack	i/p	Action
E $	id + id * id $	
TE' $	id + id * id $	E→TE'
FT'E' $	id + id * id $	T→FT'
id T'E' $	id + id * id $	F→id
T'E' $	+ id * id $	
E' $	+ id * id $	T'→ϵ
+TE' $	+ id * id $	E'→+TE'
TE' $	id * id $	
FT'E' $	id * id $	T→FT'
id T'E' $	id * id $	F→id
T'E' $	* id $	
*FT'E' $	* id $	T'→*FT'

Parse Tree

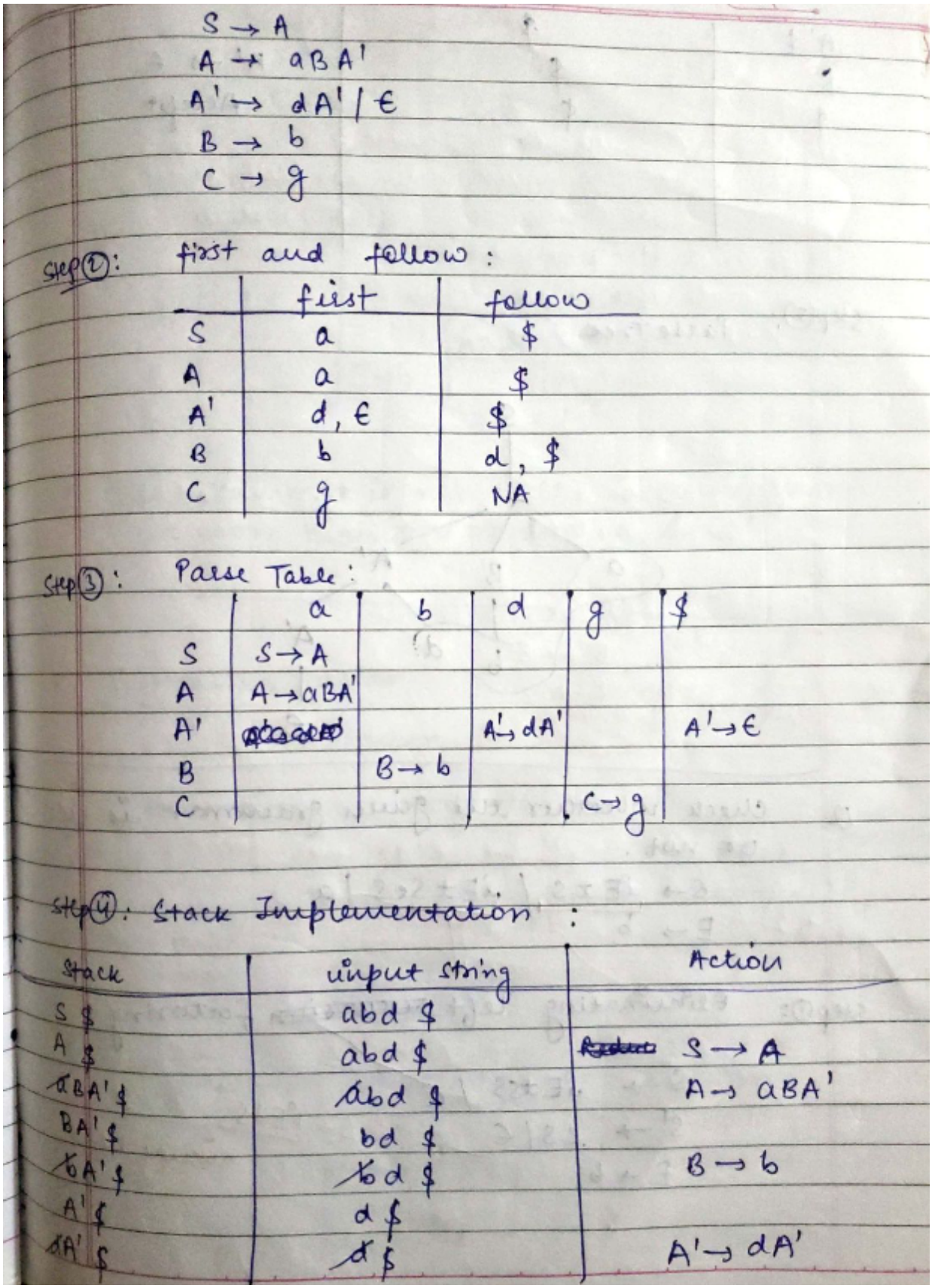
Date
DELTA Pg No

FT'E'$	id $	• F → id
~~id~~ T'E'$	~~id~~ $	~~E' → ϵ~~ T' → ϵ
T'E'$	$	E' → ϵ
E'$	$	
$	$	



Action Top-down

12. Ans.



$S \rightarrow A$
$A \rightarrow aBA'$
$A' \rightarrow dA' / \epsilon$
$B \rightarrow b$
$C \rightarrow g$

Step ①: first and follow :

	first	follow
S	a	$
A	a	$
A'	d, ϵ	$
B	b	d, $
C	g	NA

Step ③: Parse Table :

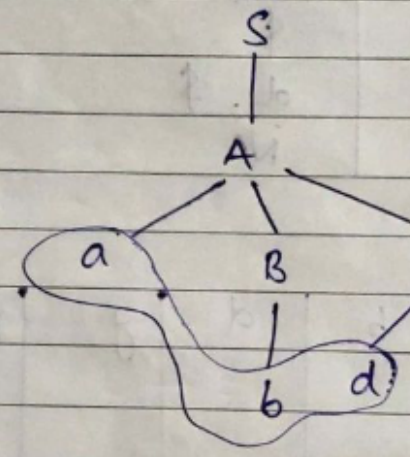
	a	b	d	g	$
S	S→A				
A	A→aBA'				
A'			A'→dA'		A'→ϵ
B		B→b			
C				C→g	

Step ④: Stack Implementation :

Stack	input string	Action
S $	abd $	
A $	abd $	S→A
aBA' $	abd $	A→aBA'
BA' $	bd $	
bA' $	bd $	B→b
A' $	d $	
dA' $	d $	A'→dA'

A'	$	
$	$	A' → ϵ Accept
$	$	

step (5): Parse Tree



13. Ans.

	FIRST( )	FOLLOW( )
S → aSbS / bSaS / ϵ	{a, b, ϵ}	{$, b, a}

	a	b	$
S → aSbS / bSaS / ϵ	S → aSbS	S → bSaS	
	S → ϵ	S → ϵ	S → ϵ

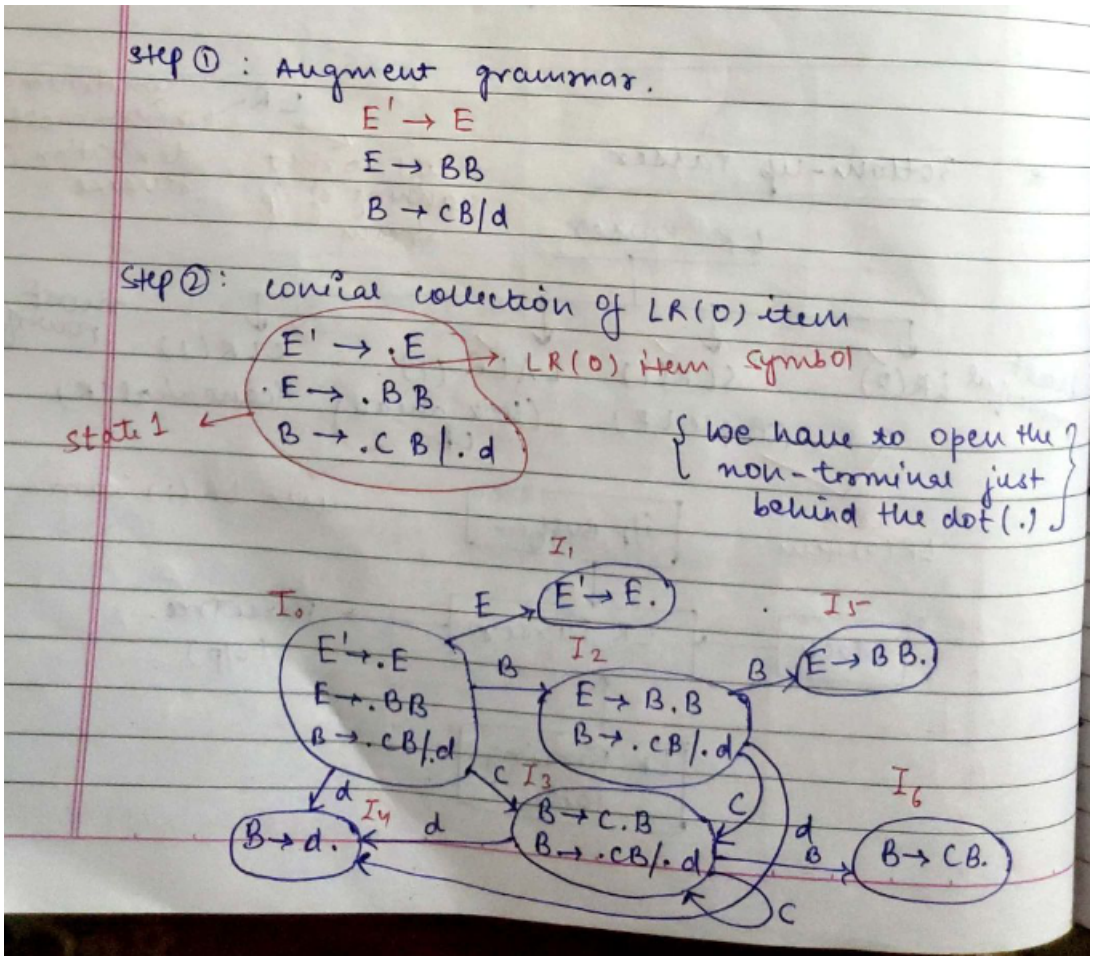
As same variable S has 2 answers for a & for b hence, its not LL(1).

14. Ans.

Stack	Input Buffer	Parsing Action
$	id+id+id$	Shift
$id	+id+id$	Reduce S->id
$S	+id+id$	Shift
$S+	id+id$	Shift
$S+id	+id$	Reduce S->id
$S+S	+id$	Reduce S->S+S
$S	+id$	Shift
$S+	id$	Shift
$S+id	$	Reduce S->id
$S+S	$	Reduce S->S+S
$S	$	Accept

15. Ans.

16. Ans.



Step ③: Number the Production

$E' \rightarrow E$

$E \rightarrow BB$ ①

$B \rightarrow cB$ ② / d ③

Step 4: Parsing Table

State	Action (Terminals)			Goto (Non-Terminals)	
	c	d	$	E	B
I_0	S_3	S_4		1	2
I_1			(self-created) Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

Step ⑤: stack implementation

stack (first state)	input	Action
$0	ccdd $	Shift c in stack and goto state 3
$0c3	cdd $	Shift c in stack and goto state 3
$0c3c3	dd $	Shift d in " & " " S_4.
$0c3c3d4	d $	Reduce r_3 i.e. $B \rightarrow d$
$0c3c3B6	d $	Reduce r_2 i.e $B \rightarrow cB$
$0c3B6	d $	Reduce r_2 i.e. $B \rightarrow cB$
$0B2	d $	shift d in stock & goto S_4.
$0B2d4	$	Reduce r_3 i.e. $B \rightarrow d$
$0B2B5	$	Reduce r_1 i.e. $E \rightarrow BB$

$ 0 E 1	$	Accept
↑ Bottom up		

Step ⑥ : Parse Tree