Question 1:

1. (a) Phases of a Compiler with Inputs and Outputs (5 Marks)

The compiler translates high-level code into machine code through several phases:

1. **Lexical Analysis:**
    - **Input:** `a = p + r * 36.0` (source code)
    - **Output:** Tokens like `ID(a)`, `=`, `ID(p)`, `+`, `ID(r)`, `*`, `NUM(36.0)`
2. **Syntax Analysis (Parsing):**
    - **Input:** Tokens from the lexical analyzer
    - **Output:** Parse tree representing the grammatical structure of the code
3. **Semantic Analysis:**
    - **Input:** Parse tree
    - **Output:** Annotated tree with type checking and semantic rules validation
4. **Intermediate Code Generation:**
    - **Input:** Annotated syntax tree
    - **Output:** Intermediate code, e.g.,
    - `t1 = r * 36.0`
    - `t2 = p + t1`
    - `a = t2`
5. **Code Optimization:**
    - **Input:** Intermediate code
    - **Output:** Optimized code with reduced instructions for efficiency
6. **Code Generation:**
    - **Input:** Optimized intermediate code
    - **Output:** Machine code or assembly code
7. **Symbol Table Management:**
    - **Input/Output:** Manages variable names like `a`, `p`, `r` during all phases

---

1. (b) Role of Lexical Analyzer with Program (5 Marks)

- **Role:**
    - Breaks source code into tokens (identifiers, keywords, literals)
    - Removes whitespace and comments
    - Detects lexical errors

Simple Program (in C) to Demonstrate Lexical Analysis:

```
#include <stdio.h>
#include <ctype.h>

int main() {
    char code[] = "a = b + 10;";
    int i = 0;
```

```
    while (code[i]) {
        if (isalpha(code[i]))
            printf("Identifier: %c\n", code[i]);
        else if (isdigit(code[i]))
            printf("Number: %c\n", code[i]);
        else if (code[i] == '+' || code[i] == '=' || code[i] == ';')
            printf("Operator: %c\n", code[i]);
        i++;
    }
    return 0;
}
```

- **Output:**

```
Identifier: a
Operator: =
Identifier: b
Operator: +
Number: 1
Number: 0
Operator: ;
```

---

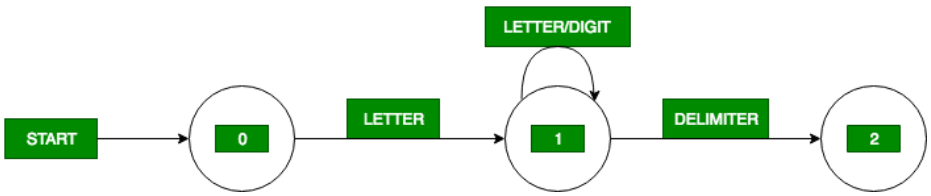
2. (a) Data Structures Used in Lexical Analysis (5 Marks)

1. **Symbol Table:**
   - Stores identifiers (variables, functions) and attributes (data type, scope).
2. **Finite Automata:**
   - Helps recognize patterns in tokens using state transitions.
3. **Hash Table:**
   - Fast lookup for reserved keywords and identifiers.
4. **Input Buffer:**
   - Handles the source code input efficiently, allowing lookahead operations.
5. **Stack:**
   - Used for managing nested structures like brackets, braces, etc.

---

2. (b) Regular Expressions & Transition Diagrams (5 Marks)

1. **Regular Expressions:**
   - **Identifier:** `[a-zA-Z_][a-zA-Z0-9_]*`
   - **Reserved Words:** `if | else | while | return | int | float`
   - **Relational Operators:** `== | != | <= | >= | < | >`



1. (a) Bootstrapping Process (5 Marks)

Definition:
Bootstrapping is the process of starting a computer from a powered-down or reset state, loading the operating system into memory. It involves a sequence of steps starting from the Basic Input/Output System (BIOS) or firmware to load the OS.

Steps Involved:

1. **Power On** – The system is powered on.
2. **BIOS/UEFI Execution** – BIOS/UEFI performs the Power-On Self-Test (POST) to check hardware components.
3. **Bootloader Activation** – BIOS/UEFI locates the bootloader in the bootable device (HDD, SSD, USB).
4. **OS Loading** – The bootloader loads the kernel of the operating system into RAM.
5. **System Initialization** – OS initializes hardware drivers, system processes, and user interfaces.

Diagram:

```
Power On → BIOS/UEFI → POST → Bootloader → Load OS Kernel → System Ready
```

Example:
When you press the power button on your laptop, it performs POST, finds the OS from your hard disk, and loads Windows/Linux into RAM.

---

1. (b) FA Equivalent to Regular Expression (0+1)*(00+11)(0+1*) (5 Marks)

Explanation:
The regular expression **(0+1)*(00+11)(0+1)*** represents strings that:

- Can have any combination of 0s and 1s,
- Must contain **00** or **11** somewhere in the string,
- Can be followed by any combination of 0s and 1s.

Finite Automaton (FA) Construction:

- **States:** q0 (start), q1, q2 (for 00), q3, q4 (for 11), qf (final state).

- **Transitions:**
  - From q0, loop on 0 and 1 (for (0+1)*).
  - From q0:
    - On 0 → q1, then on 0 → q2 (accepting 00).
    - On 1 → q3, then on 1 → q4 (accepting 11).
  - From q2 and q4, loop on 0 and 1 (for (0+1)*).

Diagram:

```
(q0) --0--> (q1) --0--> (q2*)
 |                ↑
 1                1
 |                ↓
(via 1) --> (q3) --1--> (q4*)

*q2 and q4 are final (accepting) states
```

---

2. (a) Tokens Generated by Lexical Analyzers, Lexical Errors & Recovery Strategies (5 Marks)

Tokens:
Tokens are the smallest units of meaningful code generated by the lexical analyzer. They are categorized as:

- **Keywords:** e.g., `if`, `else`, `while`
- **Identifiers:** e.g., variable names like `count`, `sum`
- **Operators:** e.g., `+`, `-`, `==`
- **Literals:** e.g., `100`, `'A'`
- **Punctuation:** e.g., `;`, `()`, `{}`

Lexical Errors:
Errors occurring during token generation, such as:

- **Misspelled keywords:** `els` instead of `else`
- **Invalid characters:** `@var` (invalid '@')
- **Unclosed literals:** `"Hello` without a closing quote

Error Recovery Strategies:

1. **Panic Mode:** Skip invalid tokens until a valid delimiter is found.
2. **Phrase-Level Recovery:** Attempt minor corrections (e.g., adding missing `;`).
3. **Error Production:** Add specific rules for common errors.
4. **Global Correction:** Tries to minimize changes to correct the error.

Example:
For code `int x @ 5;`, the `@` is an invalid character. The lexer may skip `@` and continue analyzing the rest.

---

2. (b) Regular Expression and Its Properties with Examples (5 Marks)

Definition:
A **Regular Expression (RE)** is a sequence of characters defining a search pattern, primarily for string matching.

Properties of Regular Expressions:

1. **Closure under Union:** If `R1` and `R2` are REs, then `R1 + R2` is also an RE.
   - Example: `(0+1)` matches either `0` or `1`.
2. **Closure under Concatenation:** If `R1` and `R2` are REs, `R1R2` is an RE.
   - Example: `01` matches `0` followed by `1`.
3. **Closure under Kleene Star:** If `R` is an RE, `R*` represents zero or more repetitions.
   - Example: `0*` matches ε, `0`, `00`, `000`, etc.
4. **Associativity:** `(R1 + R2) + R3 = R1 + (R2 + R3)`
5. **Distributive Property:** `R1(R2 + R3) = R1R2 + R1R3`

Example:
Regular expression `(a+b)*ab` matches strings ending with `ab` like `ab`, `aab`, `bab`, `aaab`, etc.

If you'd like detailed explanations or more examples for any of these, let me know! □

1. (a) Various Building Blocks Used to Design a Language Translator (5 Marks):

Language translators (like compilers and interpreters) are designed using the following building blocks:

1. **Lexical Analyzer (Lexer):**
   - Breaks the source code into tokens (keywords, identifiers, operators).
   - Removes white spaces and comments.
2. **Syntax Analyzer (Parser):**
   - Analyzes token sequences to ensure proper syntax based on grammar rules.
   - Generates a **parse tree**.
3. **Semantic Analyzer:**
   - Checks for semantic errors like type mismatches.
   - Uses symbol tables for variable/function validation.
4. **Intermediate Code Generator:**

    - Converts the syntax tree into intermediate code (between high-level and machine code).
5. **Code Optimizer:**
    - Improves intermediate code for efficiency without changing functionality.
6. **Code Generator:**
    - Converts optimized code into target machine code.
7. **Error Handler:**
    - Detects, reports, and handles errors in different phases (lexical, syntax, semantic).

---

1. (b) Differences Between Phase & Pass, Single-Pass & Multi-Pass Compiler (5 Marks):

Aspect	Phase	Pass
Definition	Logical steps in compilation (e.g., Lexical Analysis).	A complete traversal of the source code during compilation.
Dependency	Phases can be part of the same pass.	Each pass can include multiple phases.
Examples	Lexical Analysis, Syntax Analysis.	First pass (Syntax checking), Second pass (Optimization).
Execution	Sequentially within a pass.	Multiple passes may be needed for complex tasks.
Focus	Focuses on a specific task in compilation.	Focuses on processing the entire code per pass.

Aspect	Single-Pass Compiler	Multi-Pass Compiler
Definition	Compiler processes source code in one pass.	Compiler processes source code in multiple passes.
Speed	Faster as code is scanned once.	Slower due to multiple scans.
Memory Usage	Requires less memory.	Requires more memory for intermediate data.
Error Detection	Limited error detection.	Better error detection and optimization.
Example	Simple compilers for small applications.	Complex language compilers like C++ compilers.

---

2. (a) What is LEX? Usage of LEX in Lexical Analyzer Generation (5 Marks):

- **LEX:** A tool used to generate lexical analyzers. It processes input patterns (regular expressions) and generates C code to identify tokens.
- **Usage in Lexical Analyzer:**
    1. **Token Definition:** Define patterns using regular expressions.

2. **Input Processing:** LEX reads the input, matches patterns, and identifies tokens like identifiers, keywords, constants.
3. **C Code Generation:** Converts regular expressions into C code for token recognition.
4. **Integration:** Works with parsers like YACC for syntax analysis.
5. **Error Handling:** Detects invalid tokens and reports lexical errors.

---

2. (b) Finite Automata & Scanning Algorithm for Recognizing Identifiers & Numerical Constants in C (5 Marks):

1. **Finite Automata for Identifiers (e.g., variable names):**
   - **Rules:** Starts with a letter (A–Z, a–z) or underscore (_), followed by letters, digits, or underscores.
   - **Automata:**
     - **State 0:** Start → on letter/_ → **State 1** (valid start)
     - **State 1:** Accepting → on letter/digit/_ → stay in **State 1**
     - Invalid input → reject.
2. **Finite Automata for Numerical Constants:**
   - **Rules:** Sequence of digits (0–9), optionally with a decimal point for floating-point numbers.
   - **Automata:**
     - **State 0:** Start → on digit → **State 1** (integer)
     - **State 1:** Accepting → on digit → stay in **State 1**
     - **State 1:** on '.' → **State 2** (for decimals)
     - **State 2:** on digit → **State 3** (valid decimal)
     - Invalid characters → reject.
3. **Scanning Algorithm:**
   - **Input:** Source code string.
   - **Step 1:** Read character by character.
   - **Step 2:** Use automata to classify as identifier, number, or error.
   - **Step 3:** Tokenize valid sequences.
   - **Step 4:** Report errors if no match.

Here are the answers tailored to fit a 5-mark response for each question:

---

1. a) Phases of a Compiler (5 Marks)

A compiler translates source code into machine code through several phases:

1. **Lexical Analysis:** Breaks the source code into tokens (keywords, identifiers, operators).
   - *Example:* In `int a = 5;`, tokens are `int`, `a`, `=`, `5`, `;`

2. **Syntax Analysis (Parsing):** Checks the grammatical structure of tokens based on language rules.
    - *Example:* Validates if `int a = 5;` follows the correct declaration syntax.
3. **Semantic Analysis:** Ensures semantic correctness like type checking.
    - *Example:* Detects errors like adding an integer to a string (`int + "text"`).
4. **Intermediate Code Generation:** Converts code to an intermediate representation (IR).
    - *Example:* `t1 = 5`, `a = t1`
5. **Code Optimization:** Improves performance without changing output.
    - *Example:* Removing redundant calculations.
6. **Code Generation:** Converts IR to machine code.
    - *Example:* Generates assembly instructions like `MOV a, 5`.
7. **Symbol Table & Error Handling:** Maintains identifiers' data and handles errors during compilation.

---

1. b) Input Buffering in Lexical Analyzer (5 Marks)

Input Buffering is used to speed up reading characters from the source code. The source is divided into buffers to minimize reading time.

- **Double Buffering:** Two buffers are used. While one buffer is processed, the other is being loaded.
- **Pointers:**
    - *Forward Pointer:* Scans characters to form tokens.
    - *Lexeme Begin:* Marks the start of the current token.

Example:
For code `int x = 10;`,

- First buffer: `int x =`
- Second buffer: `10;`

The forward pointer moves character by character to identify tokens like `int`, `x`, `=`, `10`, `;`.

---

2. a) Need and Functionality of Linkers, Assemblers, and Loaders (5 Marks)

1. **Assembler:** Converts assembly code into machine code.
    - *Example:* Converts `MOV A, B` into binary code.
2. **Linker:** Combines multiple object files and resolves external references.
    - *Example:* Links code from `math.h` library with the main program.
3. **Loader:** Loads the executable code into memory for execution.
    - *Example:* When running a `.exe` file, the loader places it into RAM.

Need:

- Assembler translates code, linker integrates code, and loader enables execution.

---

2. b) Steps to Convert Regular Expression to NFA (5 Marks)

To convert a Regular Expression (RE) to a Non-deterministic Finite Automaton (NFA):

1. **Basic Symbols:**
   - For `a`, create an NFA with a start and end state connected by `a`.
2. **Union (R1 + R2):**
   - Create ε-transitions to NFA of `R1` and `R2`.
3. **Concatenation (R1R2):**
   - Connect the final state of `R1` to the start of `R2`.
4. *Kleene Star (R):**
   - Add ε-transitions for repetition, allowing 0 or more occurrences.

Example: For `a|b`:

- Create states with ε-transitions splitting into `a` and `b` branches, both connecting to a final state.

Let me know if you'd like more detailed explanations or diagrams!

Here are concise 5-mark answers for each question:

---

1. a) Token, Lexeme, and Pattern

- **Token:** A token is a category of lexical units that the compiler identifies. It represents keywords, operators, identifiers, literals, etc.
- **Lexeme:** A lexeme is the actual text or string from the source code that matches a token pattern.
- **Pattern:** A pattern is a rule, often described using regular expressions, that defines how lexemes are structured for a specific token.

Example:
Consider the statement: `int count = 10;`

- **Tokens:** `int` (Keyword), `count` (Identifier), `=` (Operator), `10` (Literal), `;` (Punctuation)
- **Lexemes:** Exact character sequences like `int`, `count`, `=`, `10`, `;`

- **Patterns:**
  - Keywords → specific reserved words like `int`, `if`, `while`
  - Identifiers → pattern like `[a-zA-Z_][a-zA-Z0-9_]*`
  - Numbers → pattern like `[0-9]+`

---

1. b) Steps to Convert NDFA to DFA

1. **Start State Identification:** The start state of the DFA is the ε-closure of the start state of the NDFA.
2. **State Representation:** Each DFA state represents a set of NDFA states (subset construction).
3. **Transitions:** For each input symbol, compute the ε-closure of the set of NDFA states reachable from the current DFA state.
4. **Mark Accepting States:** Any DFA state that includes at least one accepting state of the NDFA is an accepting state in the DFA.
5. **Repeat:** Continue until no new DFA states are generated.
6. **Minimize (Optional):** Simplify the DFA by merging equivalent states.

---

2. a) Phases of a Compiler with Example

1. **Lexical Analysis:** Converts source code into tokens.
   - *Example:* `int a = 5;` → tokens: `int`, `a`, `=`, `5`, `;`
2. **Syntax Analysis (Parsing):** Checks the syntax using grammar rules.
   - *Example:* Verifies `int a = 5;` fits declaration syntax.
3. **Semantic Analysis:** Ensures logical correctness, like type checking.
   - *Example:* Checks if `a` is declared before use.
4. **Intermediate Code Generation:** Converts code to an intermediate form.
   - *Example:* `a = 5` might become `MOV a, 5`
5. **Optimization:** Improves performance without changing output.
   - *Example:* Removing redundant code.
6. **Code Generation:** Converts intermediate code to machine code.
   - *Example:* Converts to binary instructions.
7. **Code Linking & Assembly:** Combines object code with libraries.

---

2. b) General Format of a LEX Program

A LEX program has three sections:

```
%{
    // C declarations (optional)
%}
```

```
%%
    pattern1   action1
    pattern2   action2
%%

int main() {
    yylex(); // Start lexical analysis
}

int yywrap() {
    return 1;
}
```

Explanation:

- **Declarations Section (`%{ %}`):** Includes C code and header files.
- **Rules Section (`%%`):** Patterns with corresponding actions.
- **Main Function:** Calls `yylex()` to initiate lexical analysis.

Example Pattern:

```
[0-9]+   { printf("Number detected: %s\n", yytext); }
```

This matches numbers and prints them when detected.

1. a) What are the different phases of compilation and explain the phases of the compilation with the following C language statement as an input:

position = initial + rate * 60;
Where position, initial, and rate are variables of type double.

Answer:
The compilation process has several phases:

1. **Lexical Analysis:** Converts the source code into tokens. For the given statement, tokens are: `position`, `=`, `initial`, `+`, `rate`, `*`, `60`, `;`
2. **Syntax Analysis (Parsing):** Analyzes the token sequence to check the grammatical structure based on language rules. It forms a syntax tree for the expression.
3. **Semantic Analysis:** Ensures semantic correctness, like type checking. It verifies that `position`, `initial`, and `rate` are compatible with operations (all are `double`).
4. **Intermediate Code Generation:** Translates syntax tree into intermediate code, e.g.,
5. `t1 = rate * 60`
6. `t2 = initial + t1`
7. `position = t2`
8. **Code Optimization:** Optimizes the intermediate code for efficiency, e.g., constant folding for `rate * 60` if possible.
9. **Code Generation:** Converts the optimized code into machine code.
10. **Code Linking and Assembly:** Links code with libraries and converts it to executable.

---

1. b) The programming language may be case sensitive or case insensitive. Show how to write a regular expression for a keyword in a case-insensitive language, explain with "select" in SQL.

Answer:
In case-insensitive languages like SQL, keywords can be written in any combination of uppercase and lowercase letters.

Regular Expression for "SELECT":
To recognize all case combinations of "SELECT":

```
([Ss][Ee][Ll][Ee][Cc][Tt])
```

Explanation:

- `[Ss]` matches either 'S' or 's'.
- `[Ee]` matches 'E' or 'e'.
- `[Ll]` matches 'L' or 'l'.
- `[Tt]` matches 'T' or 't'.

Examples matched: `SELECT`, `select`, `SeLeCt`, `sElEcT`

---

2. a) What is sentinel character in input buffering and explain its significance.

Answer:
A **sentinel character** is a special character used in input buffering to indicate the end of input. It helps avoid checking for the end-of-buffer condition after every character read.

Significance:

- Improves efficiency in lexical analysis.
- Reduces overhead of boundary checks.

Example:
In a buffer with size `n`, the last character can be a sentinel like `$`.

```
Input: "position=initial+rate*60;$"
```

When the lexical analyzer encounters `$`, it knows the end of input is reached.

---

2. b) Write a regular expression for recognizing the following patterns where input alphabet {a, b, c}:

i) Which begins with 'a' and ends with 'c':
Regular Expression:

```
a(a|b|c)*c
```

Explanation:

- Starts with `a`.
- `(a|b|c)*` matches any combination of 'a', 'b', 'c'.
- Ends with `c`.

ii) Which begins and ends with the same symbol:
Regular Expression:

```
(a(a|b|c)*a)|(b(a|b|c)*b)|(c(a|b|c)*c)
```

Explanation:

- `a(a|b|c)*a` matches strings starting and ending with 'a'.
- Similarly for 'b' and 'c'.

Examples:

- Matches: `abcba`, `ac`, `bab`, `cbc`
- Does not match: `ab`, `bc`, `ca`

1. a) What is boot strapping in the context of compiler and explain how it helps in language independence and reducing development time. [5M]

Answer:
Bootstrapping in the context of compilers refers to the process of writing a compiler in the source programming language that it is intended to compile. This involves using an existing simple compiler to compile a more sophisticated version of the compiler.

How it helps:

- **Language Independence:** Since the compiler can be written in its own language, it makes the language independent of any specific machine or environment.
- **Reducing Development Time:** Developers can write an initial simple compiler in a different language and then progressively improve it, reducing the time needed to develop a full compiler.

---

1. b) Write a regular expression for recognizing the following tokens in C. [5M]

i) Identifier:
Regular Expression: `[a-zA-Z_][a-zA-Z0-9_]*`

- Starts with a letter (a-z, A-Z) or underscore (_) followed by any combination of letters, digits, or underscores.

ii) Integer Constant:
Regular Expression: `[0-9]+`

- Represents one or more digits.

iii) String Constant:
Regular Expression: `"(\\.|[^\\"])*"`

- Matches double-quoted strings with possible escaped characters.

---

2. a) What is input buffering in the context of lexical analysis and explain why we take a pair of input buffers instead of a single buffer in lexical analysis. [5M]

Answer:
Input buffering is a technique used in lexical analysis to efficiently read characters from the source code. It minimizes the number of I/O operations by reading large blocks of input into memory.

Why Pair of Buffers:

- **Efficiency:** Reduces the overhead of frequent I/O operations.
- **Lookahead Capability:** Allows the lexer to backtrack if needed, which is essential for recognizing tokens that share prefixes.
- **Seamless Transition:** When one buffer is exhausted, the other buffer is already loaded, ensuring smooth reading without delays.

---

2. b) Explain the output of the following Lex specification on input 'abbbabaa'. [5M]

Lex Specification:

```
%%
a*b {printf("1"); }
ab* {printf("2"); }
b*a {printf("3"); }
ba* {printf("4"); }
```

```
%%
```

Input: `abbbabaa`

Token Matching Process:

1. `abbbabaa` matches `a*b` as `ab` → prints `1`.
2. Remaining: `bbabaa` matches `b*a` as `bba` → prints `3`.
3. Remaining: `baa` matches `ba*` as `ba` → prints `4`.
4. Remaining: `a` matches `a*b` as `a` → prints `1`.

Output: `1 3 4 1`

1. a) Compare and contrast Compiler and Interpreter. [5M]

Answer:

- **Compiler:**
  - Translates the entire source code into machine code at once.
  - Execution is faster after compilation as the code is already translated.
  - Errors are shown after the complete program is compiled.
  - Examples: GCC, Clang.
- **Interpreter:**
  - Translates and executes the source code line-by-line.
  - Execution is slower as it translates code during runtime.
  - Errors are shown immediately after the problematic line is encountered.
  - Examples: Python Interpreter, JavaScript Engine.

Key Differences:

- Compilation is done once, while interpretation happens every time the code runs.
- Compilers provide faster performance for large programs, while interpreters offer easier debugging.

---

1. b) Define Token, Pattern, and Lexeme. Identify them in the given C statement. [9M]

Answer:

- **Token:** The smallest unit in a program that has meaning to the compiler (e.g., keywords, identifiers, operators).
- **Pattern:** A rule describing how tokens are formed (often defined using regular expressions).
- **Lexeme:** The actual sequence of characters that matches the pattern for a token.

Example C Statement:

```
printf("sum of %d and %d is %d", a, b, a+b);
```

Token	Lexeme	Pattern
Keyword	printf	Identifier Pattern
Separator	(,) , , ;	Delimiter Pattern
String	"sum of %d and %d is %d"	String Literal Pattern
Identifier	a, b	Identifier Pattern
Operator	+	Operator Pattern

(OR)

2. a) What are the different ways of designing a lexical analyzer? Explain with examples. [6M]

Answer:

1. **Using Finite Automata (FA):**
   - Converts regular expressions into finite automata (DFA/NFA) to recognize tokens.
   - Example: Recognizing identifiers (`[a-zA-Z_][a-zA-Z0-9_]*`).
2. **Table-Driven Method:**
   - Uses transition tables where rows represent states and columns represent input characters.
   - Example: A state table for parsing numeric constants.
3. **Using Generator Tools:**
   - Tools like Lex generate lexical analyzers based on regular expressions.
   - Example: Writing Lex code to identify keywords, identifiers, and operators.

---

2. b) Write regular expressions for recognizing the following patterns where input alphabet = {a, b, c}. [6M]

i) Contains at least one 'a' and at most one 'b':
Regular Expression:

```
c*ac*(b?c*ac*)*
```

- Explanation: Ensures at least one 'a' and optionally one 'b' with any number of 'c's.

ii) Contains at least two 'a's and at most two 'b's:
Regular Expression:

`(c*ac*ac*)(b?c*|b?c*b?)`

- Explanation: Ensures at least two 'a's with optional occurrences of up to two 'b's.

1. (a) Define compiler and explain which phases of the compilation are machine-dependent and which are machine-independent. [5M]

Answer:
A **compiler** is a software program that translates source code written in a high-level programming language into machine code, bytecode, or another programming language. It performs several stages of analysis and code generation to produce an executable program.

Phases of Compilation:

1. **Lexical Analysis** *(Machine-Independent)* - Breaks the source code into tokens.
2. **Syntax Analysis** *(Machine-Independent)* - Checks the grammatical structure.
3. **Semantic Analysis** *(Machine-Independent)* - Ensures semantic correctness.
4. **Intermediate Code Generation** *(Machine-Independent)* - Produces intermediate code that is not machine-specific.
5. **Code Optimization** *(Machine-Independent)* - Improves intermediate code for efficiency.
6. **Code Generation** *(Machine-Dependent)* - Converts intermediate code to machine-specific code.
7. **Code Linking and Assembly** *(Machine-Dependent)* - Finalizes machine code with system-specific details.

Machine-Dependent Phases: Code Generation, Code Linking, and Assembly.
Machine-Independent Phases: Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Code Generation, Code Optimization.

1. (b) How does a lexical analyzer differentiate between DO in the following Fortran statements? [5M]
A: `DO 5 I = 1.50`
B: `DO 5 I = 1,50`

Answer:
In Fortran, spaces are insignificant, making it challenging to differentiate between constructs like these. The **lexical analyzer** uses tokenization rules to distinguish between keywords and identifiers.

- **Statement A:** `DO5I=1.50` is treated as a single identifier `DO5I` because there's no comma, making `1.50` a floating-point number.
- **Statement B:** `DO 5 I = 1,50` is recognized as a DO loop because the comma `,` indicates a list separator, matching the DO loop syntax `DO label variable = start, end`.

How Lexical Analyzer Differentiates:

- **Context and Lookahead:** Lexical analyzer checks for patterns. In A, `1.50` (floating-point) matches an identifier context, while in B, the presence of a comma matches the DO-loop pattern.
- **Finite Automata:** The analyzer transitions between states based on tokens and delimiters like `,`.

(OR)

2. (a) Explain the purpose of lexeme beginning and forward pointers in buffer pairs with an example. [5M]

Answer:
The **lexeme beginning pointer (LB)** and **forward pointer (FP)** are used in lexical analysis to identify tokens efficiently in a buffer.

- **Lexeme Beginning Pointer (LB):** Points to the start of the current lexeme.
- **Forward Pointer (FP):** Moves ahead to find the end of the current lexeme.

Example: Consider the string: `int a = 10;`

- **Initial State:** `LB` points to `i`, `FP` moves ahead.
- **Token Detection:** As `FP` moves, when it hits a space, it identifies `int` as a token (`LB` to `FP-1`).
- **Next Token:** `LB` moves to `a`, and `FP` scans ahead until the next space or delimiter.

This mechanism allows efficient **buffer management**, reducing the need to backtrack.

---

2. (b) Explain the output of the following Lex specification on input `bbaabaab`. [5M]

Lex Specification:

```
%%
a*b {printf("1");}
ab* {printf("2");}
b*a {printf("3");}
ba* {printf("4");}
%%
```

Input: `bbaabaab`

Explanation: Lex matches the longest possible substring (maximal munch rule):

1. `bb` → `b*b`**? No match.** Move ahead.
2. `bba` → **Matches** `b*a` → **Output** `3`
3. `baab` → `ba*` **matches** `baa` → **Output** `4`
4. **Remaining** `b` **matches** `b*a` → **Output** `3`

Final Output: `3 4 3`