**Section 3.1 The Role of the Lexical Analyzer**

**3.1.1 C++ Lexeme Division**  
This exercise involved breaking a small C++ program into lexemes and assigning each a token type, such as keywords, identifiers, operators, and separators. It demonstrates the basic purpose of a lexical analyzer. Implementing it in Python shows how regular expressions can be used to split source code into meaningful components.

**3.1.2 HTML Lexeme Division**  
The task required parsing an HTML snippet into its elements: tags, attributes, attribute values, and text content. It highlights that lexical analysis is not limited to programming languages but also applies to markup languages. Regex-based parsing was used to analyze the tag structures effectively.

**3.1.3 Detecting Lexical Errors**  
The objective here was to identify illegal characters in code input. The lexical analyzer must flag characters that do not match any valid token pattern. This exercise stresses the importance of handling errors during the scanning phase.

**3.1.4 Panic-Mode Error Recovery**  
This problem extended lexical error detection by implementing panic-mode recovery, where the analyzer skips over invalid symbols and continues scanning. It shows how compilers can recover from errors without halting completely, allowing further token processing.

**Section 3.2 Input Buffering**

**3.2.1 Buffer Pairs**  
This exercise demonstrated the double-buffering technique used in compilers. By alternately filling two buffers, large inputs can be processed more efficiently. It emphasizes performance considerations in lexical analysis.

**3.2.2 Sentinel Technique**  
The sentinel method places a special marker at the end of the buffer to eliminate repeated boundary checks. This approach illustrates a practical optimization that reduces scanning overhead for each character.

**Section 3.3 Specification of Tokens**

**3.3.1 Regex for Token Classes**  
In this task, regular expressions were defined for typical token categories, including identifiers, integers, floats, operators, and keywords. It demonstrates how token patterns can be directly expressed using regex in a scanner.

**3.3.2 Regex for Binary Numbers Divisible by 2**  
This problem required a regex that matches binary numbers ending in 0. It shows how specific constraints in a language can be represented with concise regular expressions.

**3.3.3 Regex for Identifiers**  
The goal was to formalize the structure of identifiers, which must start with a letter or underscore and may include digits. This illustrates practical regex design for programming language syntax rules.

**3.3.4 Regex for Floating Numbers**  
The task involved defining floating-point constants with optional exponent parts. It demonstrates regex’s ability to capture numeric formats used in programming languages.

**3.3.5 Regex for C-style Comments**  
This exercise focused on matching multi-line C-style comments (/\* ... \*/) using regex. It highlights the complexity of handling multi-line constructs and introduces concepts like greedy versus non-greedy matching.

**3.3.6 Regex for Signed Integers**  
The problem required matching integers with optional plus or minus signs. It shows how simple regex patterns can be extended to cover broader language definitions.

**Section 3.4 – Recognition of Tokens**  
These problems focus on identifying tokens such as identifiers, numbers, keywords, and operators. By using regex and transition diagrams, we simulate a lexical analyzer that categorizes input. The exercises include distinguishing reserved words from identifiers, tokenizing arithmetic expressions, and handling invalid tokens.

Problem 1: Recognize identifiers and numbers

Problem 2: Recognize keywords vs identifiers

Problem 3: Tokenize a simple expression

**Section 3.5 – Lexical Analyzer Generator (Lex)**  
This section simulates Lex tool behavior. Token rules are specified with regex, and a scanner is produced to process input. Conflict resolution, such as the longest-match rule, is shown (e.g., differentiating = from ==). Lookahead operators are also simulated to ensure keywords like if are not misinterpreted as part of longer identifiers.

Problem 1: Define rules like Lex and tokenize

Problem 2: Conflict resolution (Longest match rule) Lex uses the longest match rule. Problem 3: Lookahead operator simulation

**Section 3.6 – Finite Automata**  
Here, problems focus on simulating DFAs and NFAs. Exercises include constructing transition tables, testing string acceptance, and building automata for simple languages. Examples include a DFA for strings with an even number of symbols and an NFA for strings ending in “ab.” These exercises lay the groundwork for converting regex into automata.

Problem 1: DFA simulation

Problem 2: Transition table for DFA

Problem 3: NFA simulation (subset construction)

Problem 4: Convert regex to simple NFA (basic example)

**Section 3.7 – From Regular Expressions to Automata**  
Problems in this section illustrate converting regular expressions into NFAs and then DFAs via subset construction. Simple NFAs for regex patterns like a\* and (a|b)\*ab were built. Python implementations generated DFA transition tables, and simulations were performed on example strings.

Problem 1: Convert regex a\* to NFA (Thompson’s construction idea)

Problem 2: NFA simulation with ε-closure

Problem 3: Convert NFA → DFA (subset construction)

Problem 4: DFA simulation after conversion

**Section 3.8 – Design of a Lexical Analyzer Generator**  
This section covers building a mini Lex-like generator in Python. Token rules are defined, and the generator automatically creates a scanner. Both regex-based and DFA-based scanning methods are demonstrated. The final exercise shows an automated scanner builder, similar to real lexical analyzer generators.

Problem 1: Define token rules (like Lex) and build regex

Problem 2: NFA-based matcher (pattern matching simulation)

Problem 3: DFA-based scanner (direct conversion idea)

Problem 4: Generate scanner automatically from rules