North South University

HW2

[https://ece.northsouth.edu/~shahriar.karim/CSE_425/CSE_425_HOMEWORK_3.pdf]

Submitted by:

Arka Karmoker

ID: 2112343642

Course: CSE425

Section: 1

Submitted to:

Dr. Md Shahriar Karim [MSK1]

Department of Electrical and Computer Engineering (ECE)

North South University

Date of submission: 26/03/2025

Problem 1: Describe the alternative design approaches of a lexical analyzer.

Answer:

A lexical analyzer (lexer) is responsible for converting a sequence of characters into a sequence of tokens. There are several design approaches for implementing a lexical analyzer:

1. **Manual Implementation / Hand-Coded Lexical Analyzer:**
   - Description: The programmer manually writes code to recognize tokens using conditional statements and loops. It typically involves character-by-character processing with explicit rules for token identification.
   - Advantages: Offers fine-grained control, can be optimized for specific languages, and avoids external tool dependencies.
   - Disadvantages: Time-consuming to develop, error-prone, and difficult to maintain or modify for complex grammars.
2. **Table-Driven Lexical Analyzer:**
   - Description: Uses a predefined transition table (based on finite automata) to drive the token recognition process. The table encodes states and transitions for each character input.
   - Advantages: Systematic and easier to modify by updating the table rather than code. Suitable for automation.
   - Disadvantages: Requires more memory for the table and may be slower due to table lookups.
3. **Lexer Generator Tools (e.g., Lex, Flex):**
   - Description: A tool takes a specification file containing regular expressions for tokens and generates a lexical analyzer automatically (typically as a table-driven implementation).
   - Advantages: Fast development, reliable, and maintainable. Ideal for complex languages.
   - Disadvantages: Dependency on external tools, less control over the generated code, and potential inefficiency for simple tasks.
4. **Regular Expression-Based Approach:**
   - Description: Tokens are defined using regular expressions, and the lexer matches input against these patterns using a regex engine or converted finite automata.

- Advantages: Clean and formal definition of tokens, reusable across implementations.
- Disadvantages: May require additional processing to convert regex to executable code, potentially slower for simple cases.

Each approach varies in complexity, performance, and maintainability, with the choice depending on the language's requirements and development constraints.

Problem 2: Implement a lexical analyzer system for simple arithmetic expression using C-programming, as
demonstrated in your course textbook (11th edition, Fig. 4.1, Page 191). Explain each block of codes being
used. Show the output for the source code segment (sum + 47)/total.

Answer:

C Code:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <windows.h>
#include <psapi.h> // For process memory info

// Ensure Psapi.lib is linked (for MSVC users)
#pragma comment(lib, "Psapi.lib")

#define MAX 100

// Token types
typedef enum {
   LEXER_IDENTIFIER, LEXER_NUMBER, LEXER_PLUS, LEXER_MINUS,
   LEXER_MULTIPLY, LEXER_DIVIDE, LEXER_LPAREN, LEXER_RPAREN, LEXER_END
} LexerTokenType;

// Global variables
char input[MAX];
int pos = 0;

// Function to get the next token
LexerTokenType getToken(char *token) {
   char ch;
   int i = 0;

   // Skip whitespace
```

```
    while (isspace(input[pos])) pos++;

    ch = input[pos];
    if (ch == '\0') return LEXER_END;

    // Check for operators and parentheses
    if (ch == '+') { pos++; return LEXER_PLUS; }
    if (ch == '-') { pos++; return LEXER_MINUS; }
    if (ch == '*') { pos++; return LEXER_MULTIPLY; }
    if (ch == '/') { pos++; return LEXER_DIVIDE; }
    if (ch == '(') { pos++; return LEXER_LPAREN; }
    if (ch == ')') { pos++; return LEXER_RPAREN; }

    // Check for identifiers
    if (isalpha(ch)) {
        while (isalnum(input[pos])) token[i++] = input[pos++];
        token[i] = '\0';
        return LEXER_IDENTIFIER;
    }

    // Check for numbers
    if (isdigit(ch)) {
        while (isdigit(input[pos])) token[i++] = input[pos++];
        token[i] = '\0';
        return LEXER_NUMBER;
    }

    return LEXER_END; // Unknown character
}

// Main function
int main() {
    char token[MAX];
    LexerTokenType type;
    LARGE_INTEGER frequency, start, end;
    PROCESS_MEMORY_COUNTERS pmc;

    strcpy(input, "(sum + 47) / total");
    printf("Input: %s\n", input);
    printf("Tokens:\n");

    // Get frequency for high-resolution timing
    if (!QueryPerformanceFrequency(&frequency)) {
        printf("Error: QueryPerformanceFrequency failed.\n");
        return 1;
    }
    QueryPerformanceCounter(&start);
    pos = 0;
    while ((type = getToken(token)) != LEXER_END) {
        switch (type) {
            case LEXER_IDENTIFIER: printf("Identifier: %s\n", token); break;
            case LEXER_NUMBER: printf("Number: %s\n", token); break;
            case LEXER_PLUS: printf("Operator: +\n"); break;
            case LEXER_MINUS: printf("Operator: -\n"); break;
            case LEXER_MULTIPLY: printf("Operator: *\n"); break;
            case LEXER_DIVIDE: printf("Operator: /\n"); break;
            case LEXER_LPAREN: printf("Left Parenthesis: (\n"); break;
            case LEXER_RPAREN: printf("Right Parenthesis: )\n"); break;
            case LEXER_END: break; // Not reached, but added to silence warning
        }
    }

    // Measure end time
    QueryPerformanceCounter(&end);

    // Calculate execution time
    double time_taken = (double)(end.QuadPart - start.QuadPart) / frequency.QuadPart;
    printf("\nExecution Time: %.6f seconds\n", time_taken);

    // Measure process-specific memory usage
    HANDLE hProcess = GetCurrentProcess();
    if (GetProcessMemoryInfo(hProcess, &pmc, sizeof(pmc))) {
        printf("Memory Usage: %lu KB\n", (unsigned long)(pmc.WorkingSetSize / 1024));
    } else {
        printf("Memory Usage: Unable to retrieve\n");
    }

    return 0;
}
```

Explanation of Code Blocks:

1. **Header Files and Preprocessor Directives:**
    a. #include <stdio.h>, <ctype.h>, <string.h>: Standard libraries for input/output, character classification, and string manipulation.
    b. #include <windows.h>, <psapi.h>: Windows-specific libraries for performance timing and memory usage.
    c. #pragma comment(lib, "Psapi.lib"): Links the Psapi library for memory info (Windows-specific).

2. **Token Type Definition:**
    a. typedef enum { ... } LexerTokenType: Defines token types (e.g., identifier, number, operators) as an enumeration for clarity and type safety.

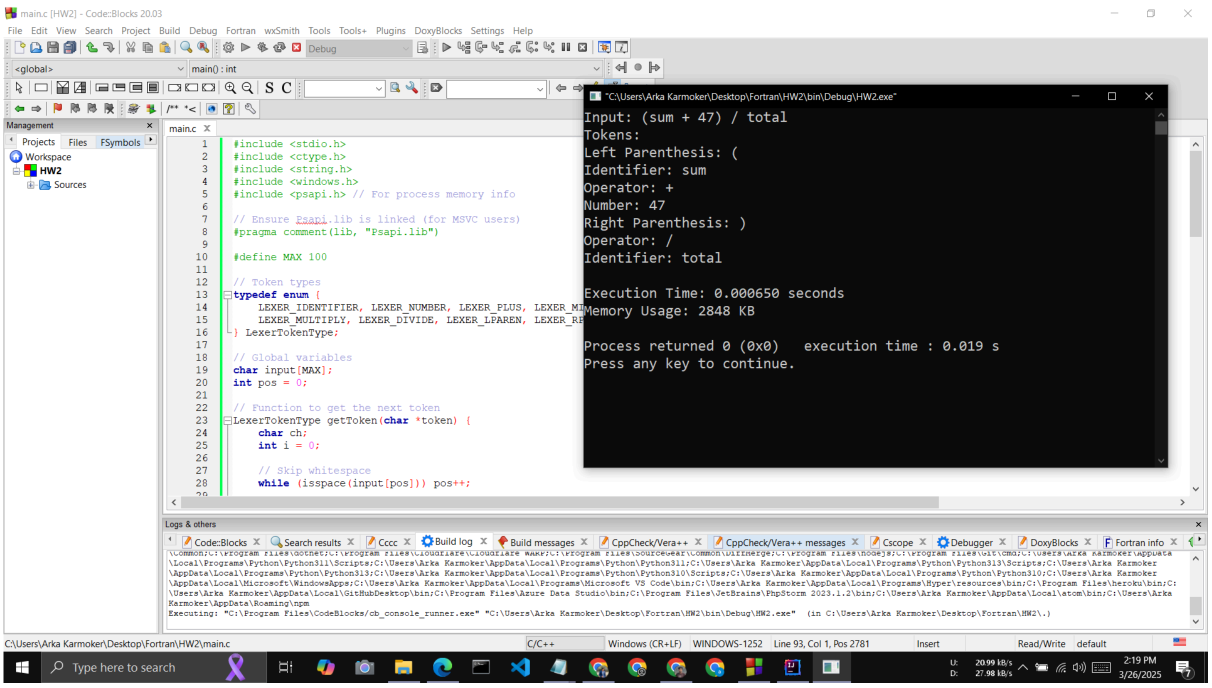
3. **Global Variables:**
    a. char input[MAX]: Stores the input string (e.g., "(sum + 47) / total").
    b. int pos = 0: Tracks the current position in the input string.

4. **getToken Function:**
    a. Skips whitespace using isspace().
    b. Checks for end of input (\0) and returns LEXER_END.
    c. Identifies single-character tokens (e.g., +, -, (,)) and increments pos.
    d. For identifiers (letters), builds the token using isalnum() and returns LEXER_IDENTIFIER.
    e. For numbers (digits), builds the token using isdigit() and returns LEXER_NUMBER.

5. **Main Function:**
    a. Initializes the input string and prints it.
    b. Uses QueryPerformanceFrequency and QueryPerformanceCounter for high-resolution timing.
    c. Loops through the input, calling getToken() and printing each token type and value.
    d. Calculates and prints execution time.
    e. Uses GetProcessMemoryInfo to measure and print memory usage.

Output for `(sum + 47) / total`:

```
Input: (sum + 47) / total
Tokens:
Left Parenthesis: (
Identifier: sum
Operator: +
Number: 47
Right Parenthesis: )
Operator: /
Identifier: total

Execution Time: 0.000650 seconds
Memory Usage: 2848 KB
```



```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <windows.h>
#include <psapi.h> // For process memory info

// Ensure Psapi.lib is linked (for MSVC users)
#pragma comment(lib, "Psapi.lib")

#define MAX 100

// Token types
typedef enum {
    LEXER_IDENTIFIER, LEXER_NUMBER, LEXER_PLUS, LEXER_MI
    LEXER_MULTIPLY, LEXER_DIVIDE, LEXER_LPAREN, LEXER_RP
} LexerTokenType;

// Global variables
char input[MAX];
int pos = 0;

// Function to get the next token
LexerTokenType getToken(char *token) {
    char ch;
    int i = 0;

    // Skip whitespace
    while (isspace(input[pos])) pos++;
```

```
Input: (sum + 47) / total
Tokens:
Left Parenthesis: (
Identifier: sum
Operator: +
Number: 47
Right Parenthesis: )
Operator: /
Identifier: total

Execution Time: 0.000650 seconds
Memory Usage: 2848 KB

Process returned 0 (0x0)   execution time : 0.019 s
Press any key to continue.
```

```
\Common;C:\Program Files\dotnet;C:\Program Files\Cloudflare\Cloudflare WARP;C:\Program Files\SourceGear\Common\DiffMerge;C:\Program Files\nodejs;C:\Program Files\Git\cmd;C:\Users\Arka Karmoker\AppData
\Local\Programs\Python\Python311\Scripts;C:\Users\Arka Karmoker\AppData\Local\Programs\Python\Python311;C:\Users\Arka Karmoker\AppData\Local\Programs\Python\Python313\Scripts;C:\Users\Arka Karmoker
\AppData\Local\Programs\Python\Python313;C:\Users\Arka Karmoker\AppData\Local\Programs\Python\Python310\Scripts;C:\Users\Arka Karmoker\AppData\Local\Programs\Python\Python310;C:\Users\Arka Karmoker
\AppData\Local\Microsoft\WindowsApps;C:\Users\Arka Karmoker\AppData\Local\Programs\Microsoft VS Code\bin;C:\Users\Arka Karmoker\AppData\Local\Programs\Hyper\resources\bin;C:\Program Files\heroku\bin;C:
\Users\Arka Karmoker\AppData\Local\GitHubDesktop\bin;C:\Program Files\Azure Data Studio\bin;C:\Program Files\JetBrains\PhpStorm 2023.1.2\bin;C:\Users\Arka Karmoker\AppData\Local\atom\bin;C:\Users\Arka
Karmoker\AppData\Roaming\npm
Executing: "C:\Program Files\CodeBlocks/cb_console_runner.exe" "C:\Users\Arka Karmoker\Desktop\Fortran\HW2\bin\Debug\HW2.exe"  (in C:\Users\Arka Karmoker\Desktop\Fortran\HW2\.)
```

main.c [HW2] - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
"C:\Users\Arka Karmoker\Desktop\Fortran\HW2\bin\Debug\HW2.exe"
C:\Users\Arka Karmoker\Desktop\Fortran\HW2\main.c
C/C++ Windows (CR+LF) WINDOWS-1252 Line 93, Col 1, Pos 2781 Insert Read/Write default
2:19 PM 3/26/2025

Problem 3: Implement a lexical analyzer stated in Problem 2 using Java programming language and compare
its execution speed and memory usage with that of C-programming.

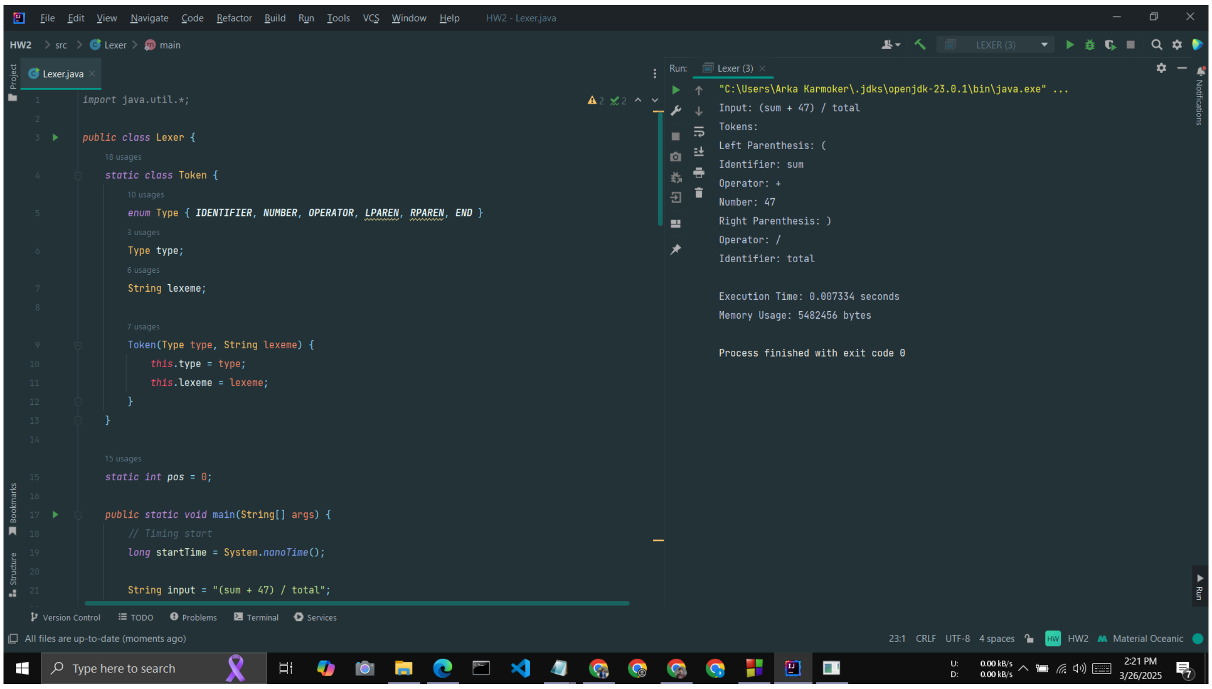
<u>Answer:</u>

<u>Java Code</u>:

```
import java.util.*;

public class Lexer {
    static class Token {
        enum Type { IDENTIFIER, NUMBER, OPERATOR, LPAREN, RPAREN, END }
        Type type;
        String lexeme;

        Token(Type type, String lexeme) {
            this.type = type;
            this.lexeme = lexeme;
        }
    }

    static int pos = 0;

    public static void main(String[] args) {
        // Timing start
        long startTime = System.nanoTime();
        String input = "(sum + 47) / total";
        pos = 0;

        System.out.println("Input: " + input);
        System.out.println("Tokens:");

        // Process tokens
        Token token;
        while (!(token = getNextToken(input)).type.equals(Token.Type.END)) {
            printToken(token);
        }

        // Timing end
        long endTime = System.nanoTime();
        double timeSpent = (endTime - startTime) / 1_000_000_000.0;

        // Memory usage
        Runtime runtime = Runtime.getRuntime();
        runtime.gc();
        long memoryUsed = runtime.totalMemory() - runtime.freeMemory();

        System.out.printf("\nExecution Time: %.6f seconds\n", timeSpent);
        System.out.println("Memory Usage: " + memoryUsed + " bytes");
    }

    static Token getNextToken(String input) {
        StringBuilder lexeme = new StringBuilder();

        // Skip whitespace
        while (pos < input.length() && Character.isWhitespace(input.charAt(pos))) pos++;

        // End of input
        if (pos >= input.length()) return new Token(Token.Type.END, "");

        char c = input.charAt(pos);

        // Check for parentheses
        if (c == '(') { pos++; return new Token(Token.Type.LPAREN, "("); }
        if (c == ')') { pos++; return new Token(Token.Type.RPAREN, ")"); }

        // Check for operators
        if (c == '+' || c == '-' || c == '*' || c == '/') {
            pos++;
            return new Token(Token.Type.OPERATOR, String.valueOf(c));
        }

        // Check for identifiers
        if (Character.isLetter(c)) {
            while (pos < input.length() && Character.isLetterOrDigit(input.charAt(pos))) {
                lexeme.append(input.charAt(pos++));
            }
            return new Token(Token.Type.IDENTIFIER, lexeme.toString());
        }

        // Check for numbers
        if (Character.isDigit(c)) {
            while (pos < input.length() && Character.isDigit(input.charAt(pos))) {
                lexeme.append(input.charAt(pos++));
            }
            return new Token(Token.Type.NUMBER, lexeme.toString());
        }

        return new Token(Token.Type.END, "");
    }

    static void printToken(Token token) {
        switch (token.type) {
            case IDENTIFIER: System.out.println("Identifier: " + token.lexeme); break;
            case NUMBER: System.out.println("Number: " + token.lexeme); break;
            case OPERATOR: System.out.println("Operator: " + token.lexeme); break;
            case LPAREN: System.out.println("Left Parenthesis: " + token.lexeme); break;
            case RPAREN: System.out.println("Right Parenthesis: " + token.lexeme); break;
            default: break;
        }
    }
}
```

Output for `(sum + 47) / total`:

```
Input: (sum + 47) / total
Tokens:
Left Parenthesis: (
Identifier: sum
Operator: +
Number: 47
Right Parenthesis: )
Operator: /
Identifier: total

Execution Time: 0.007334 seconds
Memory Usage: 5482456 bytes
```



```java
import java.util.*;

public class Lexer {
    static class Token {
        enum Type { IDENTIFIER, NUMBER, OPERATOR, LPAREN, RPAREN, END }
        Type type;
        String lexeme;

        Token(Type type, String lexeme) {
            this.type = type;
            this.lexeme = lexeme;
        }
    }

    static int pos = 0;

    public static void main(String[] args) {
        // Timing start
        long startTime = System.nanoTime();

        String input = "(sum + 47) / total";
```

```
"C:\Users\Arka Karmoker\.jdks\openjdk-23.0.1\bin\java.exe" ...
Input: (sum + 47) / total
Tokens:
Left Parenthesis: (
Identifier: sum
Operator: +
Number: 47
Right Parenthesis: )
Operator: /
Identifier: total

Execution Time: 0.007334 seconds
Memory Usage: 5482456 bytes

Process finished with exit code 0
```

Comparison of C and Java Implementations:

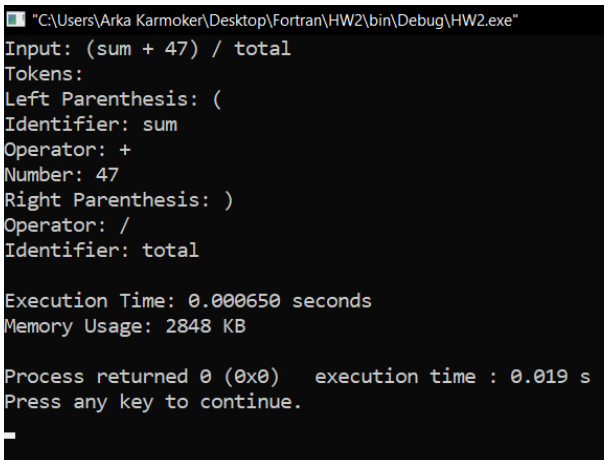
- **Execution Speed:**
  - C: 0.000650 seconds
  - Java: 0.007334 seconds
  - Analysis: The C implementation is significantly faster (approximately 11 times) due to its compiled nature, direct memory access, and lack of JVM overhead. Java's slower performance stems from JVM startup time and garbage collection, even for a small task.

- **Memory Usage:**
  - C: 2848 KB (2.848 MB)
  - Java: 5482456 bytes (≈5.48 MB)
  - Analysis: Java uses more memory due to the JVM's runtime environment, object overhead (e.g., String, Token objects), and garbage collection. C's memory usage is lower as it operates closer to the hardware with minimal runtime support.

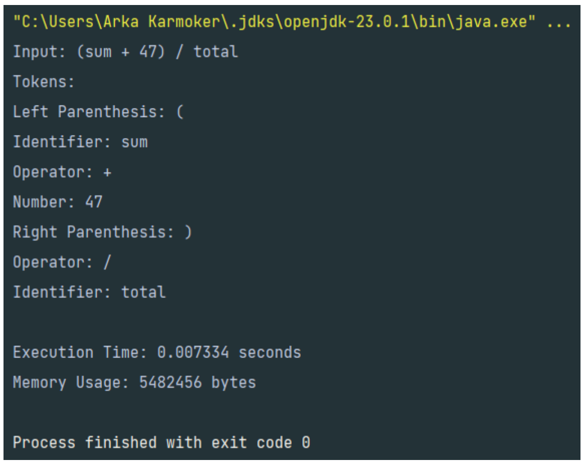
- **Other Factors:**
  - Portability: Java is more portable due to the JVM, while the C code relies on Windows-specific libraries (windows.h, psapi.h).
  - Development: Java's object-oriented design may be easier to extend, while C requires manual memory management.

Summary of Comparison

Metric	C Implementation	Java Implementation
Execution Speed	Faster (0.000650 seconds) - 11x faster than Java	Slower (0.007334 seconds)
Memory Usage	Low (2848 KB = 2.848 MB)) - Almost 2x less memory usage than Java	High (5482456 bytes = 5.48 MB)



```
"C:\Users\Arka Karmoker\Desktop\Fortran\HW2\bin\Debug\HW2.exe"
Input: (sum + 47) / total
Tokens:
Left Parenthesis: (
Identifier: sum
Operator: +
Number: 47
Right Parenthesis: )
Operator: /
Identifier: total

Execution Time: 0.000650 seconds
Memory Usage: 2848 KB

Process returned 0 (0x0)   execution time : 0.019 s
Press any key to continue.
```



```
"C:\Users\Arka Karmoker\.jdks\openjdk-23.0.1\bin\java.exe" ...
Input: (sum + 47) / total
Tokens:
Left Parenthesis: (
Identifier: sum
Operator: +
Number: 47
Right Parenthesis: )
Operator: /
Identifier: total

Execution Time: 0.007334 seconds
Memory Usage: 5482456 bytes

Process finished with exit code 0
```

Conclusion: C outperforms Java in speed and memory efficiency for this simple lexical analyzer, but Java offers better portability and maintainability.

Problem 4: Define finite automata and explain how finite automata can be used to detect source code string.

Answer:

Definition: A finite automaton (FA) is a mathematical model for recognizing patterns in strings, defined as a 5-tuple M=(S, Σ, δ, s0, F), where:

- ➔ **S**: A finite set of states.
- ➔ **Σ**: An alphabet (set of input symbols).
- ➔ **δ**: A transition function that maps a state and input symbol to a next state (for DFA: $\delta: S\times\Sigma\rightarrow S$; for NFA: $\delta: S\times(\Sigma\cup\{\epsilon\})\rightarrow P(S)$).
- ➔ **s0**: A start state. ($s0 \in S$)
- ➔ **F**: A set of accept (final) states ($F \subseteq S$).

There are two types:

- ❖ **Deterministic Finite Automaton (DFA)**: Each state has exactly one transition per input symbol.
- ❖ **Nondeterministic Finite Automaton (NFA)**: States may have multiple transitions or ε-transitions (no input).

Use in Detecting Source Code Strings:

Finite automata are widely used in lexical analysis to recognize patterns (tokens) in source code:

1. **Token Specification**: Each token type (e.g., identifier, number, operator) is defined by a regular expression, which can be converted into an FA.

2. **State Transitions**: The FA processes the input string character by character, transitioning between states based on the input. For example:
   - For an identifier (`[a-zA-Z][a-zA-Z0-9]*`):
     - Start state → Letter → Accept state (if followed by non-alphanumeric).
     - Loop on letters/digits.
   - For a number (**[0-9]+**):
     - Start state → Digit → Accept state (if followed by non-digit).
     - Loop on digits.

3. **Acceptance**: When the FA reaches an accept state and the input is fully consumed (or delimited), the token is recognized.

4. **Implementation**: The FA can be implemented as a table (table-driven lexer) or hardcoded logic (handwritten lexer). They ensure efficient, linear-time (O(n)) token recognition.

Example: Detecting the String "`sum`"

States:
- S0 (start)
- S1 (after 's')
- S2 (after 'su')
- S3 (after 'sum', accepting)
- S4 (trap state)

Alphabet: {a, b, ..., z}

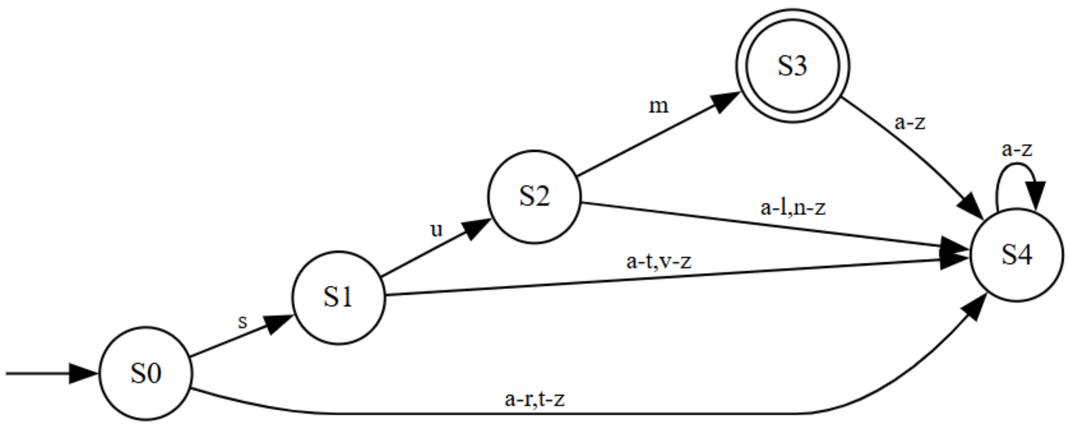
Transitions:
- S0: 's' → S1, else → S4
- S1: 'u' → S2, else → S4
- S2: 'm' → S3, else → S4
- S3: any letter → S4
- S4: any letter → S4

Start State: S0

Accepting State: S3

The DFA accepts "sum" if it reaches S3.

DFA to Detect the String 'sum'



Finite automata enable efficient token detection in lexical analysis by modeling token patterns as state machines. The DFA for "sum" demonstrates this, accepting the string when it reaches S3, as visualized in the diagram.

Problem 5: Design a state diagram to recognize comments both in Java and C programming.

Answer:

Java and C support different comment styles:

- C: `/* multi-line comment */` and `// single-line comment`.
- Java: Same as C (`/* */` and `//`).

To recognize these comments, we design a Deterministic Finite Automaton (DFA) that processes the input character by character. The DFA will transition between states based on the input characters and accept when a complete comment is recognized.

DFA Design:

States:

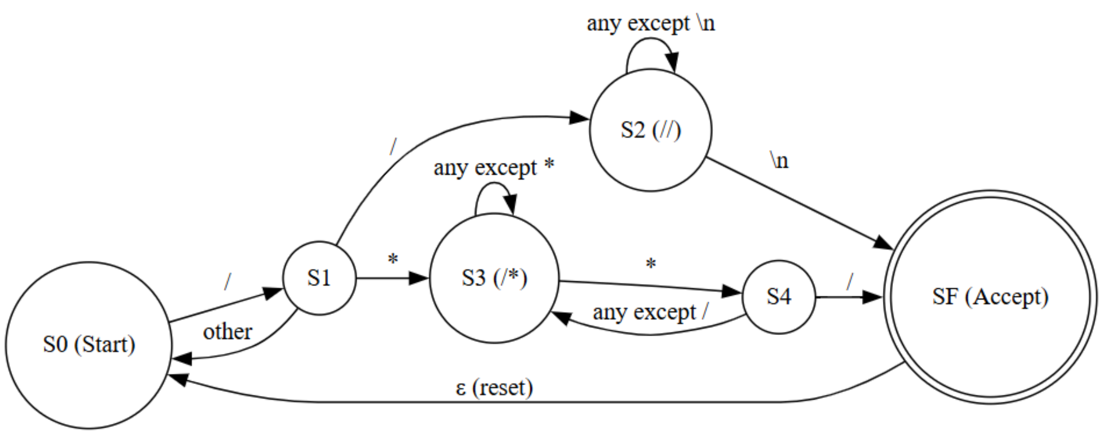
- S0: Initial state (no comment detected yet).
- S1: Intermediate state after seeing /.
- S2: Inside a single-line comment (after seeing //).
- S3: Inside a multi-line comment (after seeing /*).
- S4: Potential end of a multi-line comment (after seeing * in /*...*).
- SF: Final state (comment fully recognized, reset to S0).

Alphabet: All characters, including /, *, \n, and others.

Transitions:

1. From S0 to S1 on / (potential comment start).
2. From S1 to S2 on / (confirms //, start of single-line comment).
3. From S1 to S3 on * (confirms /*, start of multi-line comment).
4. From S1 to S0 on any character except / or * (not a comment).
5. From S2 to S2 on any character except \n (stay in single-line comment).
6. From S2 to SF on \n (end of single-line comment).
7. From S3 to S3 on any character except * (stay in multi-line comment).
8. From S3 to S4 on * (potential end of multi-line comment).
9. From S4 to S3 on any character except / (not yet ending multi-line comment).
10. From S4 to SF on / (end of multi-line comment with */).
11. From SF to S0 automatically (reset to start, no input consumed, denoted as ε-transition).

State Diagram:



This DFA recognizes both single-line (//) and multi-line (/* */) comments, returning to the initial state after completion.