Tutorial Sheet : Implementing Lexical Analyzer

Problem Statement:
Let us learn how to implement a DFA for a toy language. Assume the toy language has the following lexical elements to be recognized.

(1) Keyword in.
(2) Keyword out
(3) Operators: This are arithmetic symbols namely `+`, `-`, `*`, `/`
(4) Identifiers: Words that begin “id”, like “id12a”. These can represent variables. These include all words with “id” as a prefix followed by some alphanumeric character. Anything that does not match the patterns above will be declared as an Unknown Lexical Element.

In this tutorial, we will learn how to design a lexical analyser that recognizes the tokens in the toy languages. We will also look at the C implementation of this lexical analyser, screenshots of which have been used for reference in this tutorial.

The following steps will take you through the process one step at a time. By the end of this tutorial, you’ll understand how basic tokenization works in compilers or interpreters.

1.1 On the Alphabet for our language

The language considers all the letters (`a-z & A-Z`), digits (`0-9`) and operators (`+`, `-`, `*`, `/`). All other characters will be treated as unknown.

1.2 Token Types in our language

```c
// Token types that the lexer will recognise
typedef enum
{
    TOKEN_KEYWORD_IN,  // The keyword "in"
    TOKEN_KEYWORD_OUT, // The keyword "out"
    TOKEN_OPERATOR,    // Arithmetic operators (+, -, *, /)
    TOKEN_IDENTIFIER,  // Identifiers that start with "id"
    TOKEN_UNKNOWN      // Unknown token type
} TokenType;
```

1.3 Character Types

Based on the alphabet we have considered, we will group the characters into 4 distinct groups:

A. CHAR_LETTER - This includes the Alphabet (`a-z & A-Z`)
B. CHAR_DIGIT - This includes all the digits (`0-9`)

C. CHAR_OPERATOR - The four Arithmetic Operators (+, -, *, /)

D. CHAR_UNKNOWN - Any other character apart from the above mentioned

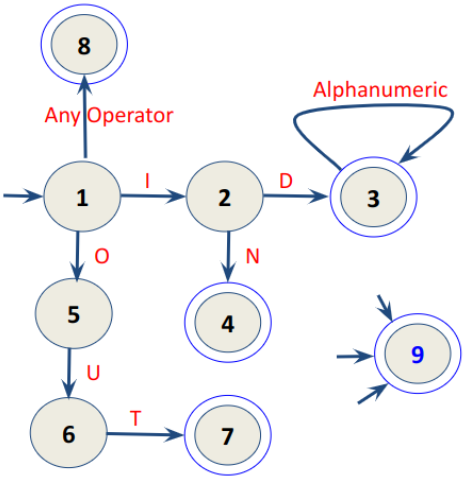
We can create an enumeration in C with these character classes/character types as below.

```c
// Character classes based on character type (letters, digits, operators, unknown)
enum
{
    CHAR_LETTER,   // Alphabetic characters
    CHAR_DIGIT,    // Digits (0-9)
    CHAR_OPERATOR, // Arithmetic operators (+, -, *, /)
    CHAR_UNKNOWN   // Any other character
};
```

Using character classes or types allows us to simplify transitions by having a single edge represent any alphabet symbol in a set. Without this, we would need to create separate edges for each individual symbol in {a,b,c, ..,z}, which would make the diagram overly complex. Imagine a DFA where a single edge corresponds to each character type, streamlining the design.

1.4 DFA For Our Lexer

This is the DFA we have designed for analysing each lexeme in our toy language:



States	Label
1	Start
2	Identifier Prefix
3	Identifier
4	IN Keyword
5,6	OUT Prefix
7	OUT Keyword
8	OPERATOR
9	ERROR STATE

Note:Transfer the DFA into state 9 from 1, 2,3,4,5,6,7 and 8 whenever an unexpected input symbol/symbols not handled is received.

```
// DFA states to process each character of the input
enum
{
    START,              // Starting state
    IN_KEYWORD,         // State for recognising the "in" keyword
    OUT_KEYWORD,        // State for recognising the "out" keyword
    IDENTIFIER_PREFIX, // State when "id" is recognised as an identifier prefix
    IDENTIFIER,         // State for continuing identifier recognition
    OPERATOR,           // State for recognising operators
    ERROR               // Error state for invalid inputs
};
```

We have defined 7 distinct states in our DFA, whose details are as below:

1. **START (State #1)** : Initial State of the DFA. Based on the character detected, this state can branch off to:
    a. **IDENTIFIER_PREFIX**: If it detects something similar to the identifier prefix
    b. **OPERATOR**: If it detects one of the four valid arithmetic operators
    c. **IN_KEYWORD**: If it detects the **in** keyword
    d. **OUT_KEYWORD**: If it detects the **out** keyword
    e. **ERROR**: If it detects an unknown character
2. **IN_KEYWORD(State #4)**: If this is our final state, then our tokenisation has detected the **in** keyword, and we report it. This is a valid acceptance state.
3. **OUT_KEYWORD(State #7)**: If this is our final state, then our tokenisation has detected the **out** keyword, and we report it. This is a valid acceptance state.
4. **IDENTIFIER_PREFIX(State #2)**: It detects the initial part of the `"id"` prefix. If the full identifier prefix is detected then we move to the IDENTIFIER state. This is an invalid acceptance state.
5. **IDENTIFIER (State #3)**: This state continues as long as the input contains valid identifier characters (letters or digits). This is a valid acceptance state and reports to us the identifier we have tokenised.
6. **OPERATOR(State #8)**: Recognizes operators like `+`, `-`, `*`, `/`. This is a valid acceptance state.
7. **ERROR(State #9)**: If something unexpected is found, the lexer transitions to this state. This is a valid acceptance state.

The transition table in the code corresponds to DFA transitions for these states in this order for the different character types mentioned before.

```
// Transition table (rows: states, columns: character classes)
// Defines how the DFA transitions between states for each input character type
int transitionTable[7][4] = {
    // CHAR_LETTER,  CHAR_DIGIT,  CHAR_OPERATOR, CHAR_UNKNOWN
    {IDENTIFIER_PREFIX, ERROR, OPERATOR, ERROR}, // START
    {ERROR, ERROR, ERROR, ERROR},                // IN_KEYWORD
    {ERROR, ERROR, ERROR, ERROR},                // OUT_KEYWORD
    {IDENTIFIER, ERROR, ERROR, ERROR},           // IDENTIFIER_PREFIX
    {IDENTIFIER, IDENTIFIER, ERROR, ERROR},      // IDENTIFIER
    {ERROR, ERROR, ERROR, ERROR},                // OPERATOR
    {ERROR, ERROR, ERROR, ERROR}                 // ERROR
};
```

Lexer Transition Table

INPUT TYPE → STATE ↓	ALPHABETS	DIGITS	OPERATORS	OTHER CHARACTERS
S_1 (START)	S_2	S_9	S_8	S_9
S_4 (IN_KEYWORD)	S_9	S_9	S_9	S_9
S_7 (OUT_KEYWORD)	S_9	S_9	S_9	S_9
S_2 (IDENTIFIER_PREFIX)	S_3	S_9	S_9	S_9
S_3 (IDENTIFIER)	S_3	S_3	S_9	S_9
S_8 (OPERATOR)	S_9	S_9	S_9	S_9
S_9 (ERROR)	S_9	S_9	S_9	S_9

1.5 Code Snippets Explained

(A) **getCharClass**: This function helps us categorise each character into it's character type. This will helps us easily implement the transition table.

```
// Function to classify characters into CHAR_LETTER, CHAR_DIGIT, CHAR_OPERATOR, or CHAR_UNKNOWN
int getCharClass(char c)
{
    if (isalpha(c)) // Check if the character is an alphabet
        return CHAR_LETTER;
    if (isdigit(c)) // Check if the character is a digit
        return CHAR_DIGIT;
    if (c == '+' || c == '-' || c == '*' || c == '/') // Check for arithmetic operators
        return CHAR_OPERATOR;
    return CHAR_UNKNOWN; // Any other character is classified as unknown
}
```

(B) **recogniseToken**: This is the core function which processes each token, moves through the DFA states, and finally determines what kind of token it is.

```c
// Function to recognise the type of token from the input string
TokenType recogniseToken(const char *input)
{
    int state = START; // Start at the initial state
    int i = 0;

    // Process each character of the input string
    while (input[i] != '\0')
    {
        char c = input[i];
        int charClass = getCharClass(c); // Get character class

        // Transition to the next state based on current state and character class
        state = transitionTable[state][charClass];

        i++;
    }

    // Determine token type based on the final state after processing all characters
    if (state == IDENTIFIER && strncmp(input, "id", 2) == 0)
        return TOKEN_IDENTIFIER; // If the string starts with "id", it's an identifier
    if (state == OPERATOR && strlen(input) == 1)
        return TOKEN_OPERATOR; // If it is a single operator, return operator type
    if (strcmp(input, "in") == 0)
        return TOKEN_KEYWORD_IN; // recognise the "in" keyword
    if (strcmp(input, "out") == 0)
        return TOKEN_KEYWORD_OUT; // recognise the "out" keyword

    return TOKEN_UNKNOWN; // If no rules match, return unknown token type
}
```

(C) **lexer**: This is the driver function that reads the input string, splits it into tokens, and uses **recogniseToken** to identify the type of each token. We have to take into account the different types of delimiters that can occur between tokens in our input string.

```c
// Main function to tokenise input
void lexer(const char *input)
{
    const char *delimiters = " \t\n"; // Delimiters to split the input string (space, tab, newline)
    char buffer[100];
    strcpy(buffer, input); // Copy input to buffer to avoid modifying the original string

    // Tokenise the input using strtok, splitting by the delimiters
    char *token = strtok(buffer, delimiters);
    while (token != NULL)
    {
        // recognise the type of the token
        TokenType type = recogniseToken(token);
        const char *tokenName;

        // Map the recognised token type to a human-readable name
        switch (type)
        {
        case TOKEN_KEYWORD_IN:
            tokenName = "Keyword 'in'";
            break;
        case TOKEN_KEYWORD_OUT:
            tokenName = "Keyword 'out'";
            break;
        case TOKEN_OPERATOR:
            tokenName = "Operator";
            break;
        case TOKEN_IDENTIFIER:
            tokenName = "Identifier";
            break;
        default:
            tokenName = "Unknown"; // If no match, label the token as unknown
            break;
        }

        // Print the token type and its value
        printf("Token: %s; String: %s\n", tokenName, token);

        // Move to the next token
        token = strtok(NULL, delimiters);
    }
}
```

(D) **main**: This function simply takes input from the user and passes it to the **lexer**.

```c
int main()
{
    char input[100];

    // Prompt user to enter a string for tokenisation
    printf("Enter a string to tokenise: ");
    fgets(input, sizeof(input), stdin); // Read the input string

    // Call the lexer to tokenise the input string
    lexer(input);

    return 0;
}
```

1.6 Results

Let's see the output of our lexer.

Example 1:

```
Enter a string to tokenise: id123 + in out
Token: Identifier; String: id123
Token: Operator; String: +
Token: Keyword 'in'; String: in
Token: Keyword 'out'; String: out
```

Example 2:

```
Enter a string to tokenise: id12313 i144 + / in id8
Token: Identifier; String: id12313
Token: Unknown; String: i144
Token: Operator; String: +
Token: Operator; String: /
Token: Keyword 'in'; String: in
Token: Identifier; String: id8
```

You can modify the input to try out different combinations of tokens and see how the lexer handles them.

1.7 Summary

In this lab sheet, we built a basic lexer to recognize keywords, identifiers, and operators, and used a DFA to manage how the lexical analyser transitions between different states. We examined each part of the code, from token types and character classes to specific functions and the DFA's structure. By understanding the basics of lexical analysis, we build a solid foundation for more advanced parsers and compilers.