Tutorial Sheet: Extending Lexical Analyzer

Problem Statement:

In this tutorial, we'll modify our lexical analyser to recognize **Unsigned Integers** (numbers without any + or - sign) as a new token type. Right now, our lexer can handle keywords, operators, and identifiers. Let's add a new token type called `TOKEN_UNSIGNED_INTEGER` to make it recognize plain numbers like 123, 456, etc.

In the course of this tutorial we will see how we should change our DFA and the transition table to include the new token type. We will also look at the modifications needed in the C implementation of the lexical analyser.

1.1 On the Alphabet for our modified language

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

1.2 Token Types in our modified language

We have included the new token type `TOKEN_UNSIGNED_INTEGER` in our language. We will have to update the lexical analyser and DFA to identify this type of token.

```c
// Token types that the lexer will recognise
typedef enum
{
    TOKEN_KEYWORD_IN,        // The keyword "in"
    TOKEN_KEYWORD_OUT,       // The keyword "out"
    TOKEN_UNSIGNED_INTEGER,  // A sequence of digits
    TOKEN_OPERATOR,          // Arithmetic operators (+, -, *, /)
    TOKEN_IDENTIFIER,        // An identifier starting with "id"
    TOKEN_UNKNOWN            // An unknown token that doesn't match any rule
} 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`)

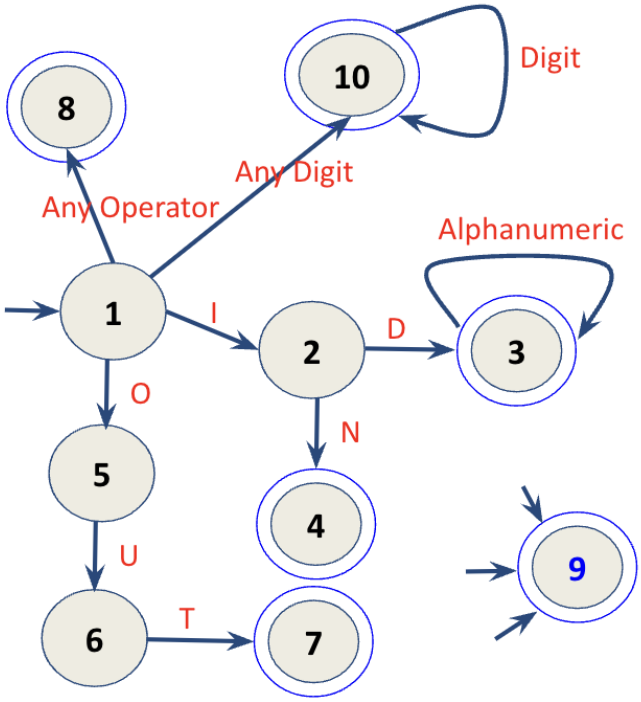
C. CHAR_OPERATOR - The four Arithmetic Operators (+, -, *, /)

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

The character types remain the same as the first tutorial and we create an enumeration in C with these character classes/character types as done earlier.

1.4 DFA For Our Lexer

This is the modified DFA after including a state for recognising `TOKEN_UNSIGNED_INTEGER`.



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

Note:Transfer the DFA into state 9 from 1, 2,3,4,5,6,7,8 and 10 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
};
```

Our modified DFA is very similar to the DFA in tutorial 1. The **START** state is modified to include a transition to the **UNSIGNED_INTEGER** state on detecting a digit, and we have defined the new **UNSIGNED_INTEGER** state with its transitions. The changes made to the DFA are detailed 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. **UNSIGNED_INTEGER**: If it detects a digit
    c. **OPERATOR**: If it detects one of the four valid arithmetic operators
    d. **IN_KEYWORD**: If it detects the **in** keyword
    e. **OUT_KEYWORD**: If it detects the **out** keyword
    f. **ERROR**: If it detects an unknown character
2. **UNSIGNED_INTEGER(State #10)**: This state continues as long as the input contains valid digits (`0-9`). This is a valid acceptance state and reports to us the unsigned integer we have tokenised.

The states not mentioned above remain the same as the earlier version.

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[8][4] = {
    // CHAR_LETTER,  CHAR_DIGIT,  CHAR_OPERATOR, CHAR_UNKNOWN
    {IDENTIFIER_PREFIX, UNSIGNED_INTEGER, OPERATOR, ERROR}, // START
    {ERROR, ERROR, ERROR, ERROR},                           // IN_KEYWORD
    {ERROR, ERROR, ERROR, ERROR},                           // OUT_KEYWORD
    {ERROR, UNSIGNED_INTEGER, ERROR, ERROR},                // UNSIGNED_INTEGER
    {IDENTIFIER, ERROR, ERROR, ERROR},                      // IDENTIFIER_PREFIX
    {IDENTIFIER, IDENTIFIER, ERROR, ERROR},                 // IDENTIFIER
    {ERROR, ERROR, ERROR, ERROR},                           // OPERATOR
    {ERROR, ERROR, ERROR, ERROR}                            // ERROR
};
```

Modified Lexer Transition Table

INPUT TYPE → STATE ↓	ALPHABETS	DIGITS	OPERATORS	OTHER CHARACTERS
S_1 (START)	S_2	S_{10}	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_{10} (UNSIGNED INTEGER)	S_9	S_{10}	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 Changes to Code Explained

(A) **getCharClass**: The getChaClass function does not change as we have not changed our character types.

(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. We simply add another IF statement to determine whether the token is `TOKEN_UNSIGNED_INTEGER` which we decide based on the final state of the DFA.

```
// 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 == UNSIGNED_INTEGER) // IMPLEMENTATION CHANGE
        return TOKEN_UNSIGNED_INTEGER;  // If final state is UNSIGNED_INTEGER, return that type
    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 will be modified in the same way as the **recogniseToken** function. We modify the switch case to include the possibility of `TOKEN_UNSIGNED_INTEGER` and report it if such a token detected

(D) **main**: No changes made to this function.

1.6 Results

Let's see the output of our modified lexer.

Example 1:

```
Enter a string to tokenise: id1 + id2 * 145
Token: Identifier; String: id1
Token: Operator; String: +
Token: Identifier; String: id2
Token: Operator; String: *
Token: Unsigned Integer; String: 145
```

Example 2:

```
Enter a string to tokenise: 1124 + 78 * id87a + i81 /
Token: Unsigned Integer; String: 1124
Token: Operator; String: +
Token: Unsigned Integer; String: 78
Token: Operator; String: *
Token: Identifier; String: id87a
Token: Operator; String: +
Token: Unknown; String: i81
Token: Operator; String: /
```

You can further experiment with the code and see how the tokenisation happens for various strings.

1.7 Summary

B.Sc. (Computer Science) | BS ZC222 - Formal Languages & Applications

Tutorial Sheet: Extending Lexical Analyzer

In the extension, we added support for unsigned integers as a new token type (`TOKEN_UNSIGNED_INTEGER`). We:

1. Defined a new token type.

2. Updated the DFA states and transition table to handle digit sequences.
3. Modified the **recogniseToken** and **lexer** functions to identify and display unsigned integers.

We have gained some experience in extending our DFA to recognise additional tokens, and this will be useful in Assignment 1 of the course.