Usman Institute of Technology

Department of Computer Science

Course Code: CS412

Course Title: Compiler Construction

Fall 2022

Lab 03

Objective:

This experiment introduces the students to implement the concept of regular expressions in programming. Also how to extract the regular expressions for a given DFA then do code.

Student Information

Student Name	
Student ID	
Date	

Assessment

Marks Obtained	
Remarks	
Signature	

Usman Institute of Technology
Department of Computer Science
CS412 – Compiler Construction

Lab 03

Instructions

- Come to the lab in time. Students who are late more than 10 minutes, will not be allowed to attend the lab.
- Students have to perform the examples and exercises by themselves.
- Raise your hand if you face any difficulty in understanding and solving the examples or exercises.
- Lab work must be submitted on or before the submission date.
- Do not copy the work of other students otherwise both will get zero marks.

1. Objective

This experiment introduces the students to implement the concept of regular expressions in programming. Also, how to extract the regular expressions for a given DFA then do code.

2. Labs Descriptions

What is Regular Expression?

Just as finite automata are used to recognize patterns of strings, regular expressions are used to generate patterns of strings. A regular expression is an algebraic formula whose value is a pattern consisting of a set of strings, called the language of the expression.

Operands in a regular expression can be:

characters from the alphabet over which the regular expression is defined.
variables whose values are any pattern defined by a regular expression.
epsilon which denotes the empty string containing no characters.
null which denotes the empty set of strings.
Operators used in regular expressions include:

Union: If R1 and R2 are regular expressions, then R1 | R2 (also written as R1 U R2 or R1 + R2) is also a regular expression.
L(R1|R2) = L(R1) U L(R2).

Concatenation: If R1 and R2 are regular expressions, then R1R2 (also written as R1.R2) is also a regular expression.
L(R1R2) = L(R1) concatenated with L(R2).

Kleene Closure: If R1 is a regular expression, then R1* (the Kleene closure of R1) is also a regular expression.
L(R1*) = epsilon U L(R1) U L(R1R1) U L(R1R1R1) U ...

Closure has the highest precedence, followed by concatenation, followed by union.

Examples

The set of strings over {0,1} that end in 3 consecutive 1's.

(0 | 1)* 111

The set of strings over {0,1} that have at least one 1.

0* 1 (0 | 1)*

The set of strings over {0,1} that have at most one 1.

0* | 0* 1 0*

The set of strings over {A..Z,a..z} that contain the word "main".

Let <letter> = A | B | ... | Z | a | b | ... | z

<letter>* main <letter>*

The set of strings over {A..Z,a..z} that contain 3 x's.
<letter>* x <letter>* x <letter>* x <letter>*

The set of identifiers in Pascal.

Let <letter> = A | B | ... | Z | a | b | ... | z
Let <digit> = 0 | 1 | 2 | 3 ... | 9

<letter> (<letter> | <digit>)*

The set of real numbers in Pascal.

Let <digit> = 0 | 1 | 2 | 3 ... | 9
Let <exp> = 'E' <sign> <digit> <digit>* | epsilon
Let <sign> = '+' | '-' | epsilon
Let <decimal> = '.' <digit> <digit>* | epsilon

<digit> <digit>* <decimal> <exp>

Unix Operator Extensions

Regular expressions are used frequently in Unix:

In the command line
Within text editors
In the context of pattern matching programs such as grep and egrep
To facilitate construction of regular expressions, Unix recognizes additional operators. These operators can be defined in terms of the operators given above; they represent a notational convenience only.

character classes: '[' <list of chars> ']'

start of a line: '^'
end of a line: '$'
wildcard matching any character except newline: '.'
optional instance: R? = epsilon | R
one or more instances: R+ == RR*
Equivalence of Regular Expressions and Finite Automata
Regular expressions and finite automata have equivalent expressive power:

For every regular expression R, there is a corresponding FA that accepts the set of strings generated by R.
For every FA A there is a corresponding regular expression that generates the set of strings accepted by A.

The proof is in two parts:

i. an algorithm that, given a regular expression R, produces an FA A such that L(A) == L(R).
ii. an algorithm that, given an FA A, produces a regular expression R such that L(R) == L(A).

Our construction of FA from regular expressions will allow "epsilon transitions" (a transition from one state to another with epsilon as the label). Such a transition is always possible, since epsilon (or the empty string) can be said to exist between any two input symbols. We can show that such epsilon transitions are a notational convenience; for every FA with epsilon transitions there is a corresponding FA without them.

Constructing an FA from an RE

We begin by showing how to construct an FA for the operands in a regular expression.

If the operand is a character c, then our FA has two states, s0 (the start state) and sF (the final, accepting state), and a transition from s0 to sF with label c.

If the operand is epsilon, then our FA has two states, s0 (the start state) and sF (the final, accepting state), and an epsilon transition from s0 to sF.

If the operand is null, then our FA has two states, s0 (the start state) and sF (the final, accepting state), and no transitions.

Given FA for R1 and R2, we now show how to build an FA for R1R2, R1|R2, and R1*. Let A (with start state a0 and final state aF) be the machine accepting L(R1) and B (with start state b0 and final state bF) be the machine accepting L(R2).

The machine C accepting L(R1R2) includes A and B, with start state a0, final state bF, and an epsilon transition from aF to b0.

The machine C accepting L(R1|R2) includes A and B, with a new start state c0, a new final state cF, and epsilon transitions from c0 to a0 and b0, and from aF and bF to cF.

The machine C accepting L(R1*) includes A, with a new start state c0, a new final state cF, and epsilon transitions from c0 to a0 and cF, and from aF to a0, and from aF to cF.

Eliminating Epsilon Transitions

If we can eliminate epsilon transitions from an FA, then our construction of an FA from a regular expression (which yields an FA with epsilon transitions) can be completed.

Observe that epsilon transitions are similar to nondeterminism in that they offer a choice: an epsilon transition allows us to stay in a state or move to a new state, regardless of the input symbol.

If starting in state s1, we can reach state s2 via a series of epsilon transitions followed by a transition on input symbol x, we can replace all of the epsilon transitions with a single transition from s1 to s2 on symbol x.

Algorithm for Eliminating Epsilon Transitions

We can build a finite automaton F2 with no epsilon transitions from a finite automaton F1 containing epsilon transitions as follows:

The states of F2 are all the states of F1 that have an entering transition labeled by some symbol other than epsilon, plus the start state of F1, which is also the start state of F2.

For each state in F1, determine which other states are reachable via epsilon transitions only. If a state of F1 can reach a final state in F1 via epsilon transitions, then the corresponding state is a final state in F2.

For each pair of states i and j in F2, there is a transition from state i to state j on input x if there exists a state k that is reachable from state i via epsilon transitions in F1, and there is a transition in F1 from state k to state j on input x.

Constructing an RE from an FA

To construct a regular expression from a DFA (and thereby complete the proof that regular expressions and finite automata have the same expressive power), we replace each state in the DFA one by one with a corresponding regular expression.

Just as we built a small FA for each operator and operand in a regular expression, we will now build a small regular expression for each state in the DFA.

The basic idea is to eliminate the states of the FA one by one, replacing each state with a regular expression that generates the portion of the input string that labels the transitions into and out of the state being eliminated.

Algorithm for Constructing an RE from an FA

Given a DFA F we construct a regular expression R such that
L(F) == L(R).

We preprocess the FA, turning the labels on transitions into regular expressions. If there is a transition with label {a,b}, then we replace the label with the regular expression a | b. If there is no transition from a state to itself, we can add one with the label NULL.

For each accepting state sF in F, eliminate all states in F except the start state s0 and sF.

To eliminate a state sE, consider all pairs of states sA and sB such that there is a transition from sA to sE with label R1, a transition from sE to sE with label R2 (possibly null, meaning no transition), and a transition from sE to sB with label R3. Introduce a transition from sA to sB with label R1R2*R3. If there is already a transition from sA to sB with label R4, then replace that label with R4|R1R2*R3.
After eliminating all states except s0 and sF:

If s0 == sF, then the resulting regular expression is R1*, where R is the label on the transition from s0 to s0.

If s0 != sF, then assume the transition from s0 to s0 is labeled R1, the transition from s0 to sF is labeled R2, the transition from sF to sF is labeled R3, and the transition from sF to s0 is labeled R4. The resulting regular expression is R1*R2(R3 | R4R1*R2)*

Let RFi be the regular expression produced by eliminating all the states except s0 and sFi. If there are n final states in the DFA, then the regular expression that generates the strings accepted by the original DFA is RF1 | RF2 | ... RFn.

Regular Expressions Library (since C++11)

The regular expressions library provides a class that represents regular expressions, which are a kind of mini-language used to perform pattern matching within strings. Almost all operations with regexes can be characterized by operating on several of the following objects:

Target Sequence: The character sequence that is searched for a pattern. This may be a range specified by two iterators, a null-terminated character string or a std::string.

Pattern: This is the regular expression itself. It determines what constitutes a match. It is an object of type std::basic_regex, constructed from a string with special syntax. See regex_constants::syntax_option_type for the description of supported syntax variations.

Matched Array: The information about matches may be retrieved as an object of type std::match_results.

Replacement String: This is a string that determines how to replace the matches, see regex_constants::match_flag_type for the description of supported syntax variations.

Main Classes

These classes encapsulate a regular expression and the results of matching a regular expression within a target sequence of characters.

basic_regex (C++11)	regular expression object (class template)
sub_match (C++11)	identifies the sequence of characters matched by a sub-expression (class template)
match_results (C++11)	identifies one regular expression match, including all sub-expression matches (class template)

Algorithms

These functions are used to apply the regular expression encapsulated in a regex to a target sequence of characters.

regex_match (C++11)	attempts to match a regular expression to an entire character sequence (function template)
regex_search (C++11)	attempts to match a regular expression to any part of a character sequence (function template)
regex_replace (C++11)	replaces occurrences of a regular expression with formatted replacement text (function template)

Iterators

The regex iterators are used to traverse the entire set of regular expression matches found within a sequence.

regex_iterator (C++11)	iterates through all regex matches within a character sequence (class template)
regex_token_iterator (C++11)	iterates through the specified sub-expressions within all regex matches in a given string or through unmatched substrings (class template)

Exceptions

This class defines the type of objects thrown as exceptions to report errors from the regular expressions library.

regex_error (C++11)	reports errors generated by the regular expressions library (class)

Traits

The regex traits class is used to encapsulate the localizable aspects of a regex.

regex_traits (C++11)	provides metainformation about a character type, required by the regex library (class template)

Constants

Defined in namespace std::regex_constants

syntax_option_type (C++11)	general options controlling regex behavior (typedef)
match_flag_type (C++11)	options specific to matching (typedef)
error_type (C++11)	describes different types of matching errors (typedef)

Program to use regex_search() for finding the pattern in a given string with other parameters.

```
#include <iostream>
#include <iterator>
#include <string>
#include <regex>

int main()
{
    std::string s = "Some people, when confronted with a problem, think "
        "\"I know, I'll use regular expressions.\" "
        "Now they have two problems.";

    std::regex self_regex("REGULAR EXPRESSIONS",
            std::regex_constants::ECMAScript | std::regex_constants::icase);
    if (std::regex_search(s, self_regex)) {
        std::cout << "Text contains the phrase 'regular expressions'\n";
    }

    std::regex word_regex("(\\w+)");
    auto words_begin =
        std::sregex_iterator(s.begin(), s.end(), word_regex);
    auto words_end = std::sregex_iterator();

    std::cout << "Found "
              << std::distance(words_begin, words_end)
              << " words\n";

    const int N = 6;
    std::cout << "Words longer than " << N << " characters:\n";
    for (std::sregex_iterator i = words_begin; i != words_end; ++i) {
        std::smatch match = *i;
        std::string match_str = match.str();
        if (match_str.size() > N) {
            std::cout << "  " << match_str << '\n';
        }
    }

    std::regex long_word_regex("(\\w{7,})");
    std::string new_s = std::regex_replace(s, long_word_regex, "[$&]");
    std::cout << new_s << '\n';

    return 0;
}
```

i. Write your observations after executing this piece of code.
ii. What is the purpose of using [$&], \\w{7, } and \\w+.
iii. What you understand by ECMAScript?

Program to use regex_search() for finding the pattern in a given string.

```
// C++ program to demonstrate working of regex_search()
#include <iostream>
#include <regex>
#include<string.h>
using namespace std;

int main()
{
    // Target sequence
    string s = "Pakistan is my Love, I love Pakistan";

    // An object of regex for pattern to be searched
    regex r("Pakistan[a-zA-Z]+");

    // flag type for determining the matching behavior
    // here it is for matches on 'string' objects
    smatch m;

    // regex_search() for searching the regex pattern
    // 'r' in the string 's'. 'm' is flag for determining
    // matching behavior.
    regex_search(s, m, r);

    // for each loop
    for (auto x : m)
        cout << x << " ";

    return 0;
}
```

After executing the above code, answer the following:

i. What is the purpose of the above regex r?
ii. What if we need that this regular expression will cater space, how does the regular expression changes?
iii. What is the purpose of auto in this code?
iv. What does the function regex_search() will do?
v. Sentence without space
vi. Change the sentence according to your favorite quote and then check.

Program to demonstrate the working of regex_match() for a given string.

```
// C++ program to demonstrate working of regex_match()
#include <iostream>
#include <regex>

using namespace std;
int main()
{
    string a = "Pakistan is my Love, I love Pakistan";

    // Here b is an object of regex (regular expression)
    regex b("(Pakistan)(.*)"); // Pakistan followed by any character

    // regex_match function matches string a against regex b
    if ( regex_match(a, b) )
        cout << "String 'a' matches regular expression 'b' \n";

    // regex_match function for matching a range in string
    // against regex b
    if ( regex_match(a.begin(), a.end(), b) )
        cout << "String 'a' matches with regular expression "
                "'b' in the range from 0 to string end\n";

    return 0;
}
```

After executing the above code, answer the followings:

i. What is the purpose of using regex_match?
ii. What is the purpose of using a.begin() and a.end() in this code?
iii. Can we change the above code? If yes enter the regex according to your criteria and show the result.

Program of using regex_replace()

```
// C++ program to demonstrate working of regex_replace()
#include <iostream>
#include <string>
#include <regex>
#include <iterator>
using namespace std;

int main()
{
    string s = "I am looking for Pakistanifamily \n";

    // matches words beginning by "Pakis"
    regex r("Pakis[a-zA-z]+");

    // regex_replace() for replacing the match with 'friend'
    cout << std::regex_replace(s, r, "friend");

    string result;

    // regex_replace( ) for replacing the match with 'friend'
    regex_replace(back_inserter(result), s.begin(), s.end(),
                  r, "friend");

    cout << result;

    return 0;
}
```

Change the conditions in the given sentence and then check the outcomes. Observe the difference between them.

i. Change the 2nd with some other word and observe the difference.
ii. Change the sentence in terms of grammar and see if this can work.
iii. Write the purpose of string in regex r? What other conditions we can use? Give five examples.

Lab 03: Design a code for regular expression to DFA and vice versa

Program to demonstrate the validation of incoming data.

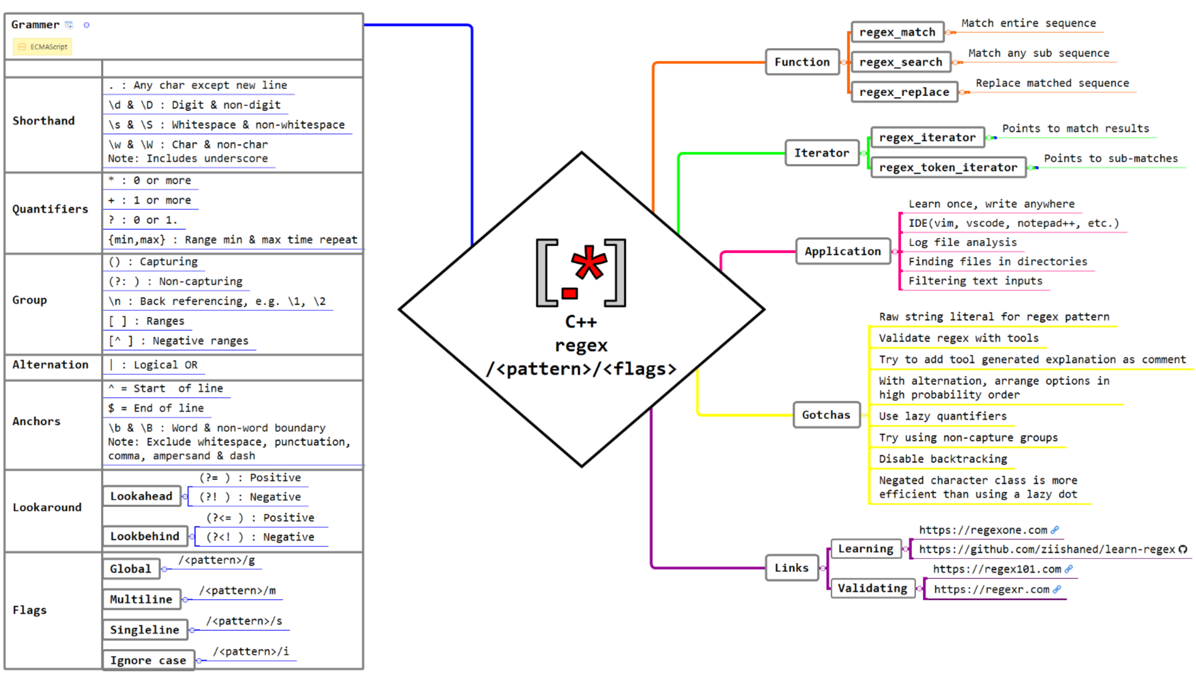
```
//Check the below program to demonstrate how you can use the regex to validate incoming data.

#include <iostream>
#include <regex>
#include <string>
using namespace std;

int main()
{
    string input;
    regex integer_expr("(\\+|-)?[[:digit:]]+");
    //As long as the input is correct ask for another number
    while(true)
    {
        cout<<"Enter the input: ";
        cin>>input;
        if(!cin) break;
        //Exit when the user inputs q
        if(input=="q")
            break;
        if(regex_match(input,integer_expr))
            cout<<"Input is an integer"<<endl;
        else
        {cout<<"Invalid input : Not an integer"<<endl;}
    }

    return 0;
}
```

.

Lab 03: Design a code for regular expression to DFA and vice versa

Another C++ program to demonstrate the regex_replace() function.

```
#include <iostream>
#include <string>
#include <regex>
#include <iterator>
using namespace std;

int main()
{
    string mystr = "This is software testing Help portal \n";

    cout<<"Input string: "<<mystr<<endl;

    // regex to match string beginning with 'p'
    regex regexp("p[a-zA-z]+");
    cout<<"Replace the word 'portal' with word 'website' : ";
    // regex_replace() for replacing the match with the word 'website'
    cout << regex_replace(mystr, regexp, "website");

    string result;

    cout<<"Replace the word 'website' back to 'portal': ";
    // regex_replace( ) for replacing the match back with 'portal'
    regex_replace(back_inserter(result), mystr.begin(), mystr.end(),
                  regexp,  "portal");

    cout << result;

    return 0;
}
```



3. Lab tasks

Task 1

Construct the automata machines for any five of the following given conditions of regular expressions:

regular language	*regular expression*	*in the language*	*not in the language*
binary alphabet			
fifth-to-last symbol is **a**	(a|b)*a(a|b)(a|b)(a|b)(a|b)	aaaaa bbbabbbb bbbbbbababababa	a bbbbbbbba aaaaaaaaaaaabaaaa
contains the substring **abba**	(a|b)*abba(a|b)*	abba aababbabbababbba bbbbbbbbabbabbbbb	abb bbabaab aaaaaaaaaaaaaaaa
does not contain the substring **bbb**	(bba|ba|a*)*(a*|b|bb)	aa ababababbaba aaaaaaaaaaaaab	bbb ababbbbabab bbbbbbbbbbbbb
number of **b** *symbols is a multiple of 3*	a*|(a*ba*ba*ba*)*	bbb aaa bbbaababbaa	b baaaaaaab baabbbaaaaab
decimal digits			
positive integer divisible by 5	5|(1|2|...|9)(0|1|...|9)*(0|5)	5 200 9836786785	1 0005 3452345234
positive ternary number	(1|2)(0|1|2)*	11 2210221	011 19 9836786785
lowercase letters			
contains the trigraph **spb**	(a|b|c|...|z)*spb(a|b|c|...|z)	raspberry crispbread	subspace subspecies
uses only the top row of the keyboard	(q|w|e|r|t|y|u|i|o|p)*	typewriter reporter	alfalfa paratrooper
genetic code			
fragile X syndrome pattern	GCG(CGG|AGG)*CTG	GCGCTG GCGCGGCTG GCGCGGAGGCTG	GCGCGG CGGCGGCGGCTG GCGCAGGCTG

Task 2

Write a program in C++ that can help user to enter the options using switch case to perform the operations of regular expressions based on their requirements.

Such as: match, replace, etc.

4. Homework Tasks

1. Design any ten of the following regular expressions:

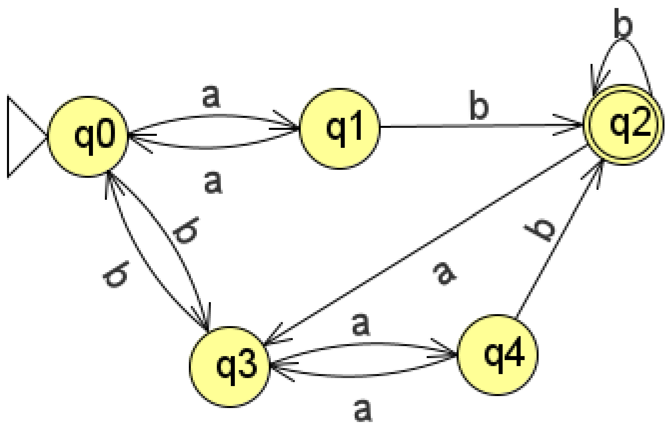
	in the language	*not in the language*
second-to-last symbol is a	aa bbbab bbbbbbbbbababab	a aaaba bbbbbbbbbbbbbb
equal numbers of as and bs	ba bbaaba aaaabbbbbbbbaaaba	a bbbaa abababababababa
palindromes	a aba abaabaabaaba	ab bbbba abababababababab
contain the pattern abba	abba abaababbabbababbba bbbbbbbbbbabbabbbbb	abb bbabaab aaaaaaaaaaaaaaaa
number of bs is divisible by 3	bbb baaaaabaaaab bbbabbaaabaaabababaaa	bb abababab aaaaaaaaaaaaaaab

	in the language	*not in the language*
palindromes	madamimadam amanaplanacanalpanama amoraroma	madamimbob madam, i'm adam not a palindrome
odd integers	3 101 583805233	2 100 2147483648
prime integers	3 101 583805233	0003 100 2147483648
integers z such that $x^2+y^2=z^2$ *for some integers x, y*	5 13 9833	2 16 9999
integers z such that $x^n+y^n=z^n$ *for some integers x, y,* $n>2$	no integers	all integers
amino acid encodings	AAA ACA ATA AGA CAA CCA CTA CGA TCA TTA GAA GCA GTA GGA AAC ACC	CCC AAAAAAAAA ABCDE
U.S. telephone number	(609) 258-3000 (800) 555-1212	(99) 12-12-12 2147483648
English words	and middle computability	abc niether misunderestimate
legal English sentences	This is a sentence. I think I can.	xya a b.c.e?? Cogito ergo sum.
legal Java identifiers	a class $xyz3_XYZ	12 123a a((BC))*
legal Java programs	public class Hi { public static void main(String[] args) { } }	int main(void) { return 0; }

2. Use JFLAP to prove that the above regular expressions are valid.
3. Write a code to check any four regular expressions that are defined in question 1. Generate the right and wrong answer or validate the correct or incorrect input.
4. For the following FA's find the regular expression and use regex to code.

Lab 03: Design a code for regular expression to DFA and vice versa

a.



b.