Lab Assignment
Compiler Construction (UCS802)

Regular Expression: ***Regular expressions*** are widely used to specify patterns. We use regular expressions to describe tokens of a ***programming language***. A regular expression is built up of simpler regular expressions (using defining rules). A regular expression can be created with a set of Alphabets defined for a language and set of operations for defining the strings in the language. $(a|b)^*abb$ is a regular expression representing a language of all the strings ending with string abb with alphabet set $\{a, b\}$, and set of operations defined as $\{*, |, concatenation\}$, with precedence set in the same order as defined above. The parenthesis can be used to simplify the regular expressions.

A ***recognizer*** for a language is a program that takes a string x, and answers "yes" if x is a sentence of that language, and "no" otherwise. We call the recognizer of the tokens as a finite automaton. We may use a deterministic or non-deterministic automaton as a lexical analyzer.

Both deterministic and non-deterministic finite automaton recognize regular sets. Deterministic automatons are widely used lexical analyzers.

First, we define regular expressions for tokens; then we convert them into a DFA to get a lexical analyzer for our tokens.

Programming Assignment 1:

Regular Expression ➔ NFA ➔ DFA, two steps: first to NFA(5 marks), then to DFA(5 marks)

Non-Deterministic Finite Automata (NFA)

A non-deterministic finite automaton (NFA) is a mathematical model that consists of:

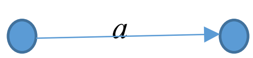
- S - a set of states
- Σ - a set of input symbols (alphabet)
- move – a transition function move to map state-symbol pairs to sets of states.
- s_0 - a start (initial) state
- F – a set of accepting states (final states)

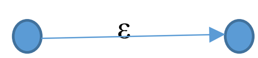
Implementation of NFA

Define a *Node* structure for defining one state of NFA, we should be able to move from this *Node* to another *Node* for any input symbol defined in Σ, and ε.

Implementation of Regular Expression

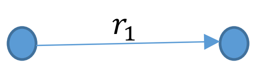
A Simple regular expression can be created using two *Nodes.* For example,
A regular expression for *ε* and *a*

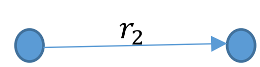




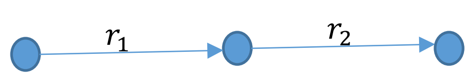
Operations on Regular Expressions

Each regular expression has one *start node* and one *end node.* Let the two regular Expressions be represented by r_1 and r_2.

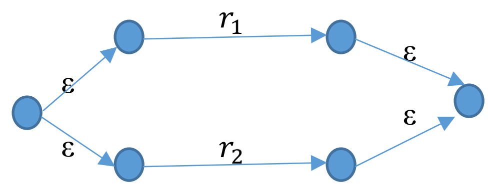




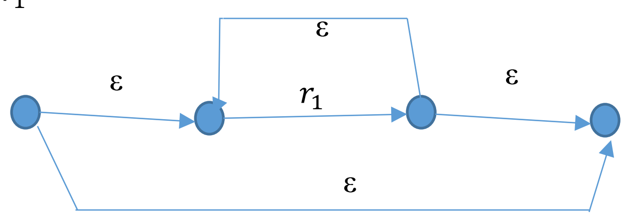
a. Concatenation of two regular Expressions: (r_1r_2): The new regular expression will be represented by:



b. $r_1 | r_2$



c. Kleene Clousure r_1^*



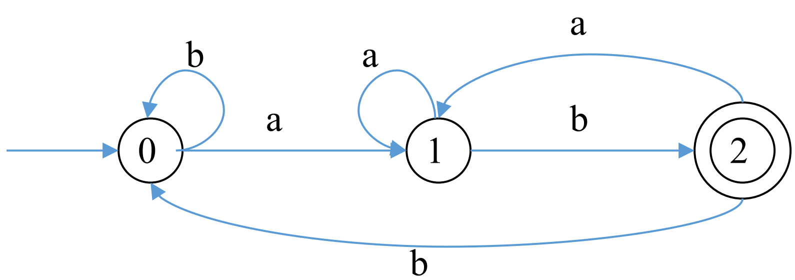
Programming Assignment –II

Given a NFA for a regular expression created using Thomson's construction rules defined in programming assignment-I create a Deterministic automata DFA. The DFA will match all the words of grammar generated by the regular expression.

Deterministic Finite Automata (DFA)

A Deterministic Finite Automaton (DFA) is a special form of a NFA. No state in DFA has ε-transition. Ffor each symbol a and state s, there is at most one labeled edge a leaving s i.e. the transition function is from pair of state-symbol to state (not set of states).

DFA for regular Expression $(\boldsymbol{a}|\boldsymbol{b})^{*}\ \boldsymbol{ab}$



Implementation of DFA

```
s ← s_o                          \\{ start from the initial state }
c ← nextchar                     \\{ get the next character from the input string }
while (c != eos)do               \\{ do until the end of the string }
        begin
          s ← move(s,c)          \\{ transition function }
          c ← nextchar
        end
if (s in F)then                  \\{ if s is an accepting state }
        return "yes"
else
        return "no"
```

Conversion of NFA to DFA

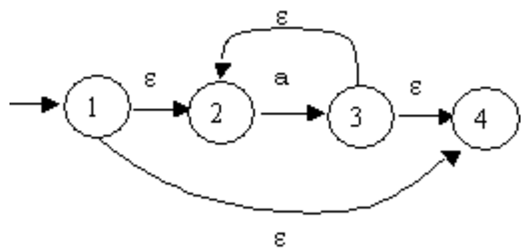
To convert and Non-deterministic Finite automata (NFA) into a Deterministic Finite Automata (DFA)

The ∈-closure of a state is the set of all states, including S itself, that you can get to via ∈-transitions. The ∈-closure of state S is denoted: $\overline{S}$

```
push all states of T onto stack
initialize ϵ − closure(T) to T
while (stack is not empty) do
begin
        pop t, the top element, off stack;
        for (each state u with an edge from t to u labelled ϵ do
```

```
        begin
                if (u is not in ϵ − closure(T)) do
                begin
                        add u to ϵ − closure(T)
                        push u onto stack
                end
        end
end
```

Example:



The $\in -$closure of state 1: $\overline{1} = \{ 1, 2, 4 \}$
The $\in -$closure of state 3: $\overline{3} = \{ 3, 2, 4 \}$

The $\in -$closure of a set of states $S_1, S_2, \ldots, S_n$ is $\overline{S_1} \cup \overline{S_2} \ldots \cup \overline{S_n}$

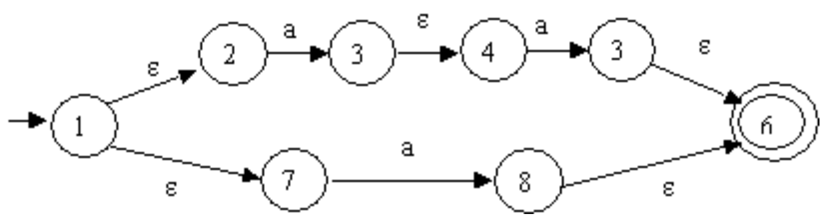
Example: The e-closure for above states 1 and 3 is

$$\{ 1, 2, 4 \} \cup \{ 3, 2, 4 \} = \{ 1, 2, 3, 4 \}$$

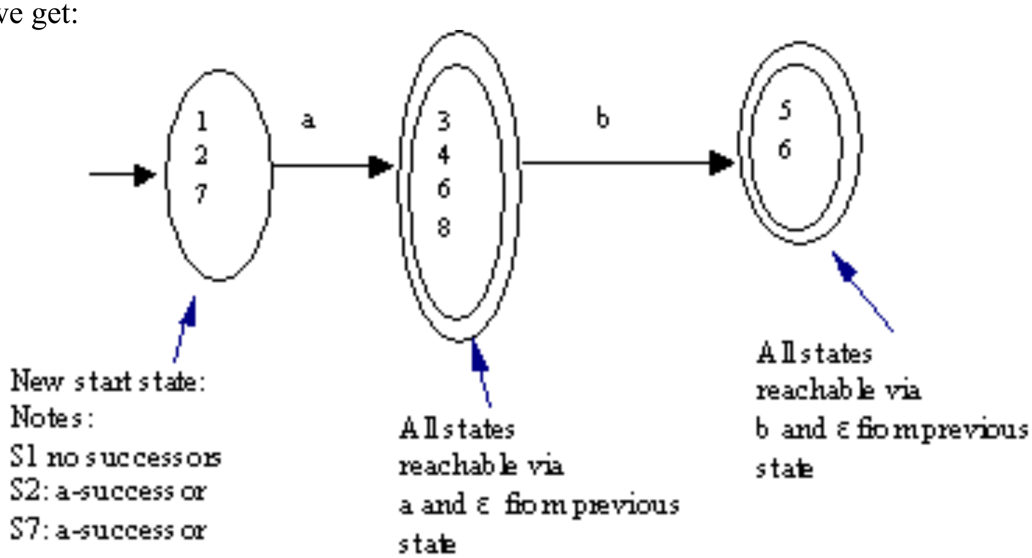
To construct a DFA from NFA the following procedure is followed:

```
put ε− closure({s0}) as an unmarked state into the set of DFA (DS)
while (there is one unmarked S1 in DS) do
begin
 mark S1
 for each input symbol a do
   begin
     S2 ← ε− closure(move(S1, a))
     if (S2 is not in DS) then
          add S2 into DS as an unmarked state
          transfunc[S1, a] ← S2
   end
end
```

Example 1: To convert the following nfa:

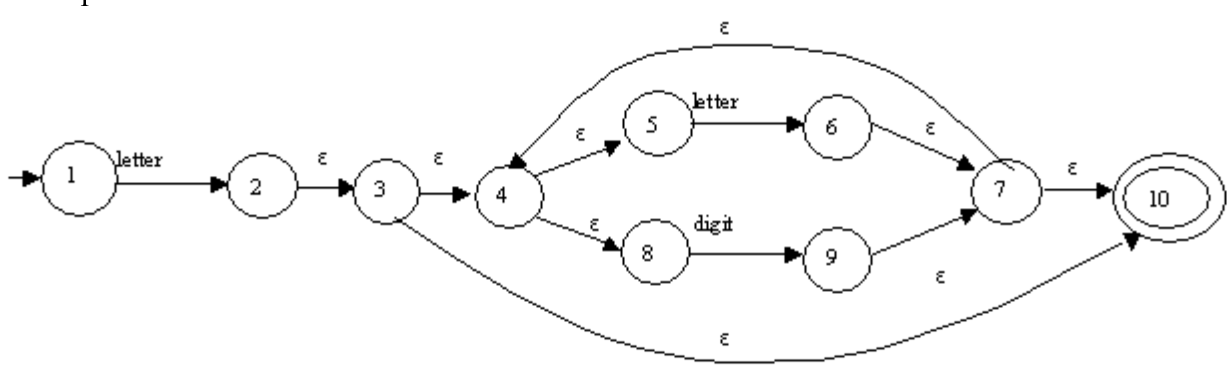


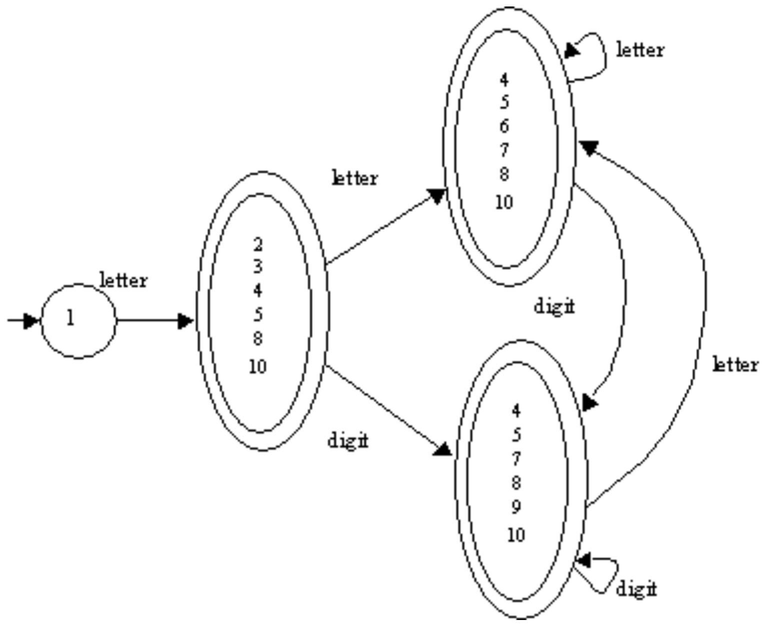
we get:



This constructs a dfa that has no epsilon-transitions and a single accepting state.

Example 2: To convert the nfa for an identifier to a dfa




letter
1
2
3
4
5
8
10
letter
4
5
6
7
8
10
letter
digit
digit
letter
4
5
7
8
9
10
digit

we get: