Lab Assignment 1

Varun Kumar

Roll No: 102217105

Subject: Compiler Construction

Programming Assignment I & II

Objective: C++ code for Regex to NFA and DFA Conversion

```cpp
#include <iostream>
#include <vector>
#include <set>
#include <map>
#include <unordered_map>
#include <unordered_set>
#include <deque>
#include <sstream>
#include <algorithm>
#include <cctype>
#include <stdexcept>
#include <string>
using namespace std;

struct NFA {
    int start, accept;
    unordered_map<int, unordered_map<char, unordered_set<int>>> trans;
    unordered_set<char> alphabet;
};

struct DFA {
    set<int> start;
    set<set<int>> accept;
    map<set<int>, map<char, set<int>>> trans;
    unordered_set<char> alphabet;
```

```
};

struct StateFactory {
    int next_id = 0;
    int New() { return next_id++; }
};

static void add_edge(unordered_map<int, unordered_map<char,
unordered_set<int>>>& trans,
                     int u, char sym, int v) {
    trans[u][sym].insert(v);
}

// Base Thompson fragment
static NFA nfa_symbol(char sym, StateFactory& sf, const unordered_set<char>&
baseAlphabet) {
    int s = sf.New(), t = sf.New();
    NFA n; n.start = s; n.accept = t; n.alphabet = baseAlphabet;
    add_edge(n.trans, s, sym, t);
    if (sym != '\0') n.alphabet.insert(sym);
    return n;
}

// Concatenation
static NFA nfa_concat(NFA a, const NFA& b) {
    add_edge(a.trans, a.accept, '\0', b.start);
    for (auto& kv1 : b.trans)
        for (auto& kv2 : kv1.second)
            for (int v : kv2.second)
                add_edge(a.trans, kv1.first, kv2.first, v);
    for (char c : b.alphabet) a.alphabet.insert(c);
    a.accept = b.accept;
    return a;
}
```

```
// Union
static NFA nfa_union(const NFA& a, const NFA& b, StateFactory& sf) {
    int s = sf.New(), t = sf.New();
    NFA n; n.start = s; n.accept = t; n.alphabet = a.alphabet;
    for (char c : b.alphabet) n.alphabet.insert(c);

    for (auto& kv1 : a.trans)
        for (auto& kv2 : kv1.second)
            for (int v : kv2.second)
                add_edge(n.trans, kv1.first, kv2.first, v);

    for (auto& kv1 : b.trans)
        for (auto& kv2 : kv1.second)
            for (int v : kv2.second)
                add_edge(n.trans, kv1.first, kv2.first, v);

    add_edge(n.trans, s, '\0', a.start);
    add_edge(n.trans, s, '\0', b.start);
    add_edge(n.trans, a.accept, '\0', t);
    add_edge(n.trans, b.accept, '\0', t);
    return n;
}

// Kleene star
static NFA nfa_star(const NFA& a, StateFactory& sf) {
    int s = sf.New(), t = sf.New();
    NFA n; n.start = s; n.accept = t; n.alphabet = a.alphabet;

    for (auto& kv1 : a.trans)
        for (auto& kv2 : kv1.second)
            for (int v : kv2.second)
```

```
                add_edge(n.trans, kv1.first, kv2.first, v);

    add_edge(n.trans, s, '\0', a.start);
    add_edge(n.trans, s, '\0', t);
    add_edge(n.trans, a.accept, '\0', a.start);
    add_edge(n.trans, a.accept, '\0', t);
    return n;
}

static bool is_sym(char c) { return isalnum(static_cast<unsigned char>(c)) ||
c == '_'; }

// Regex to RPN
static vector<char> regex_to_rpn(const string& regex) {
    vector<char> out;
    vector<char> ops;
    char prev = 0;

    auto needs_concat = [&](char curr)->bool {
        if (prev == 0) return false;
        bool prev_is_atom = is_sym(prev) || prev == ')' || prev == '*';
        bool curr_is_atom = is_sym(curr) || curr == '(';
        return prev_is_atom && curr_is_atom;
    };

    auto push_concat = [&]() {
        while (!ops.empty() && (ops.back() == '.' || ops.back() == '*') &&
ops.back() != '|') {
            out.push_back(ops.back()); ops.pop_back();
        }
        ops.push_back('.');
    };

    for (char c : regex) {
        if (isspace(static_cast<unsigned char>(c))) continue;
        if (is_sym(c)) {
```

```
            if (needs_concat(c)) push_concat();
            out.push_back(c);
        } else if (c == '(') {
            if (needs_concat(c)) push_concat();
            ops.push_back('(');
        } else if (c == ')') {
            while (!ops.empty() && ops.back() != '(') {
out.push_back(ops.back()); ops.pop_back(); }
            if (!ops.empty() && ops.back() == '(') ops.pop_back();
            else throw runtime_error("Mismatched parentheses");
        } else if (c == '*') {
            out.push_back('*');
        } else if (c == '|') {
            while (!ops.empty() && (ops.back() == '.' || ops.back() == '|')) {
                out.push_back(ops.back()); ops.pop_back();
            }
            ops.push_back('|');
        } else {
            throw runtime_error(string("Unknown regex char: ") + c);
        }
        prev = c;
    }
    while (!ops.empty()) { out.push_back(ops.back()); ops.pop_back(); }
    return out;
}

static NFA nfa_from_regex(const string& regex, unordered_set<char> alphabet) {
    StateFactory sf;
    vector<NFA> st;
    vector<char> rpn = regex_to_rpn(regex);

    for (char tok : rpn) {
        if (tok == '*') {
            NFA a = st.back(); st.pop_back();
            st.push_back(nfa_star(a, sf));
        } else if (tok == '.') {
```

```
            NFA b = st.back(); st.pop_back();
            NFA a = st.back(); st.pop_back();
            st.push_back(nfa_concat(a, b));
        } else if (tok == '|') {
            NFA b = st.back(); st.pop_back();
            NFA a = st.back(); st.pop_back();
            st.push_back(nfa_union(a, b, sf));
        } else {
            st.push_back(nfa_symbol(tok, sf, alphabet));
            alphabet.insert(tok);
        }
    }
    return st.back();
}

static set<int> epsilon_closure(const unordered_map<int, unordered_map<char,
unordered_set<int>>>& trans,
                                const set<int>& T) {
    set<int> closure = T;
    vector<int> stack(T.begin(), T.end());
    while (!stack.empty()) {
        int t = stack.back(); stack.pop_back();
        auto it1 = trans.find(t);
        if (it1 == trans.end()) continue;
        auto it2 = it1->second.find('\0');
        if (it2 == it1->second.end()) continue;
        for (int u : it2->second) {
            if (!closure.count(u)) {
                closure.insert(u);
                stack.push_back(u);
            }
        }
    }
    return closure;
}
```

```
static set<int> move_on(const unordered_map<int, unordered_map<char,
unordered_set<int>>>& trans,
                        const set<int>& S, char a) {
    set<int> out;
    for (int s : S) {
        auto it1 = trans.find(s);
        if (it1 == trans.end()) continue;
        auto it2 = it1->second.find(a);
        if (it2 == it1->second.end()) continue;
        out.insert(it2->second.begin(), it2->second.end());
    }
    return out;
}

static DFA nfa_to_dfa(const NFA& nfa) {
    DFA dfa;
    dfa.alphabet = nfa.alphabet;

    set<int> start0 = { nfa.start };
    dfa.start = epsilon_closure(nfa.trans, start0);

    deque<set<int>> work;
    set<set<int>> seen;
    work.push_back(dfa.start);
    seen.insert(dfa.start);
    if (dfa.start.count(nfa.accept)) dfa.accept.insert(dfa.start);

    while (!work.empty()) {
        set<int> S = work.front(); work.pop_front();
        for (char a : dfa.alphabet) {
            set<int> U1 = move_on(nfa.trans, S, a);
            set<int> U = epsilon_closure(nfa.trans, U1);
            dfa.trans[S][a] = U;
            if (!seen.count(U)) {
                seen.insert(U);
                work.push_back(U);
```

```
                if (U.count(nfa.accept)) dfa.accept.insert(U);
            }
        }
    }
    return dfa;
}

static bool dfa_recognize(const DFA& dfa, const string& s) {
    set<int> S = dfa.start;
    for (char c : s) {
        auto rowIt = dfa.trans.find(S);
        if (rowIt == dfa.trans.end()) return false;
        auto it = rowIt->second.find(c);
        if (it == rowIt->second.end()) return false;
        S = it->second;
    }
    return dfa.accept.count(S) > 0;
}

// Pretty printers
static void print_nfa(const NFA& nfa) {
    cout << "NFA start: " << nfa.start << "\n";
    cout << "NFA accept: " << nfa.accept << "\n";
    vector<tuple<int,char,int>> edges;
    for (auto& kv1 : nfa.trans)
        for (auto& kv2 : kv1.second)
            for (int v : kv2.second)
                edges.emplace_back(kv1.first, kv2.first, v);
    sort(edges.begin(), edges.end());
    for (auto& e : edges) {
        int u,v; char sym;
        tie(u,sym,v)=e;
        if(sym=='\0')
            cout << u << " -- ⟶ " << v << "\n";
        else
            cout << u << " --" << sym << "⟶ " << v << "\n";
```

```
    }
}

static void print_dfa(const DFA& dfa) {
    auto set_to_str = [](const set<int>& S){
        stringstream ss; ss << "{";
        bool first=true;
        for (int x : S) { if(!first) ss << ","; first=false; ss << x; }
        ss << "}"; return ss.str();
    };
    cout << "DFA start: " << set_to_str(dfa.start) << "\n";
    cout << "DFA accepts:";
    for (auto& S : dfa.accept) cout << " " << set_to_str(S);
    cout << "\n";
    vector<tuple<string,char,string>> edges;
    for (auto& kv1 : dfa.trans) {
        string Ss = set_to_str(kv1.first);
        for (auto& kv2 : kv1.second) {
            string Ts = set_to_str(kv2.second);
            edges.emplace_back(Ss, kv2.first, Ts);
        }
    }
    sort(edges.begin(), edges.end());
    for (auto& e : edges) {
        string Ss,Ts; char a;
        tie(Ss,a,Ts)=e;
        cout << Ss << " --" << a << "⟶ " << Ts << "\n";
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    unordered_set<char> alphabet = {'a','b'};
```

```
    cout << "=== Regex → NFA → DFA ===\n";
    string regex = "(a|b)*abb"; // Example regex
    NFA nfa = nfa_from_regex(regex, alphabet);
    DFA dfa = nfa_to_dfa(nfa);

    // Print constructed NFA and DFA
    cout << "\n--- NFA ---\n";
    print_nfa(nfa);
    cout << "\n--- DFA ---\n";
    print_dfa(dfa);

    // User input test
    cout << "\nEnter a string to test (over alphabet {a,b}): ";
    string input;
    cin >> input;
    bool ok = all_of(input.begin(), input.end(), [&](char c){ return
alphabet.count(c) > 0; });
    if (!ok) cout << input << " → invalid-symbol\n";
    else cout << input << " → " << (dfa_recognize(dfa, input) ? "Accepted" :
"Not Accepted") << "\n";

    return 0;
}
```

Program Output Screenshot