Compiler Design Lab Report

Santhisenan A
University Roll Number *TVE15CS050*
Department of Computer Science
College of Engineering Trivandrum

Aug - Nov 2018

Chapter 1

Automata Concepts

1.1 Finding ϵ - closures

1.1.1 Problem

Write program to find ϵ closure of all states of any given NFA with ϵ transition

1.1.2 Algorithm

The algorithm for finding ϵ Closure is as follows:

- Iterate through all the states in the NFA
- For each state, find the states reachable via epsilon transitions and add them to the epsilon closure.
- For the all the states added to epsilon closure, recursively execute the function to find the epsilon closure.

1.1.3 Code

```
void find_e_closure(nfa n, int state, unordered_set<int> &closure) {
    unordered_set<int> toStates = n.table[state][0];
    closure.insert(state);
    if (toStates.find(-1) != toStates.end())
    {
        return;
    }
    else
```

```
    {
        unordered_set<int>::iterator itr;
        for (itr = toStates.begin(); itr != toStates.end(); itr++)
        {
            if (find(closure.begin(), closure.end(), *itr) == closure.end())
            {
                closure.insert(*itr);
            }
            find_e_closure(n, *itr, closure);
        }
    }
}
```

1.2 Conversion from an NFA with ϵ transitions to an NFA without ϵ transitions

1.2.1 Problem

Write program to convert an NFA with ϵ transitions to an NFA without ϵ transitions.

1.2.2 Algorithm

- Initialize an empty object of type nf a with variable name t
- Initialize t.num states = a.num states, t.num alphabets = a.num alphabets and t.final states = a.final states
- Iterate through each of the state i in Q
  - Initialize l to the closure of state i of -NFA a Iterate through each of the input symbol j in
    - Initialize an empty list of states f Iterate through each state k in l and Add all states of a.transition table[k][j + 1] to f Remove all the duplicates from f
    - Compute the -closure c of f
    - Set t.transition table[i][j] = c
- Return t as the NFA without -transitions corresponding to the -NFA a

1.2.3 Code

```
        // To convert e-NFA to NFA
void enfa_to_nfa(nfa n, nfa &m)
{
    m.states = n.states;
    m.alphabets = n.alphabets;

    int s = n.states, a = n.alphabets;

    for (int i = 0; i < s; i++)
    {
        vector<unordered_set<int> > row;
```

```
        // Insert -1 to alphabet epsilon
        unordered_set<int> nullState;
        nullState.insert(-1);
        row.push_back(nullState);
        nullState.clear();

        // Find the e Closure of the current state and store in eClosure
        unordered_set<int> eClosure;
        unordered_set<int>::iterator itr;
        find_e_closure(n, i, eClosure);

        for (int j = 1; j < a; j++)
        {
            unordered_set<int> temp, tempClosure, toStates;
            unordered_set<int>::iterator it;
            for (itr = eClosure.begin(); itr != eClosure.end(); itr++)
            {
                temp = n.table[*itr][j];
                if (!temp.empty())
                {
                    for (it = temp.begin(); it != temp.end(); it++)
                    {
                        // cout << "Hello" << *it << endl;
                        if(*it != -1) {
                            find_e_closure(n, *it, tempClosure);
                            append_sets(toStates, tempClosure);
                        }
                    }
                }
            }
            if(toStates.empty()) {
                toStates.insert(-1);
            }
            row.push_back(toStates);
            toStates.clear();
        }
        m.table.push_back(row);
        row.clear();
    }
}
```

1.3 Conversion from an NFA to a DFA

1.3.1 Problem

Write program to convert an NFA with ϵ transitions to an NFA without ϵ transitions.

1.4 Algorithm

- Start with the start of the DFA
- Find the epsilon closure of the start state.
- For each input symbol, find the states to which the NFA goes.
- Group all those states together and name it as a new state in the nfa.
- Perform the above steps till all the states have been traversed.

1.4.1 Code

```
        // To convert e-NFA to NFA
void nfa_to_dfa(nfa n, dfa &m)
{
    m.alphabets = n.alphabets;
    for(int i = 0; i < n.alphabets; i++) {
        vector <bitset <10> > DFARow;
        for(int j = 0; j < n.states; j++) {
            unordered_set <int> eClosure;
            find_e_closure(n, j, eClosure);
            bitset <10> toStatesDFA(0);
            for(int k = 0; k < eClosure.size(); k++) {
                unordered_set <int> toStates;
                toStates = n.table[i][k];
                unordered_set <int> :: iterator itr;
                for(itr = toStates.begin(); itr != toStates.end(); itr++) {
                    if(*itr != -1) {
                        toStatesDFA.set(*itr);
                    }
                }
            }
            DFARow.push_back(toStatesDFA);
```

```
        }
        m.table.push_back(DFARow);
    }
}
```

Chapter 2

Lex and YACC

2.1 Implementation of a lexical analyser using C++

2.1.1 Aim

Design and implement a lexical analyzer for given language using C and the lexical analyzer should ignore redundant spaces, tabs and new line.

2.1.2 Lex: An introduction

Lex input language

Lex expressions are also called regular expressions or patterns. The fundamental elements of lex expressions include characters, strings and sets of characters called as character classes.

Characters include letters a to z, A to Z and _ and the digits 0 - 9. Most of the other characters are treated as special characters by lex.

A string is a sequence of characters, not including a new line, enclosed in double quotes.

A sequence of characters enclosed within square brackets([ ]) forms a character class. It matches a single instance of any character within the brackets. If a carat (^) follows the opening bracket, the class matches any character other than those inside the brackets.

Lex programs

A lex program consists of three sections:

- definitions

- translations
- functions

C style comments and newlines used in a lex program is treated as white space in lex. Lines starting with a blank or tab are copied as such to the lex output file.

Definition section: The definition section is separated form the following section by a single line containing only %%. We can define names regular expressions in this section. This section may be empty.

Translation section: This section contains regular expressions paired with actions, which define what a lexical analyser should do when a regular expression is found. The first nonesscaped tab or space marks the beginning of an action.

Function section: This section can contain anything and it will be attached to the end of lex output file as such. This section can be omitted. This section, if present should be separated from the translation section by using a %%.