Compiler Design Lab Manual

Department of Computer Science & Engineering
College of Engineering Trivandrum

Contents

1. Preface

A compiler is computer software that transforms computer code written in one programming language (the source language) into another programming language (the target language). Compilers are a type of translator that support digital devices, primarily computers. The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program.

1.1. Pre-Requisites

- CS331 System Software Lab

1.2. Course Objectives

- To implement different phases of a compiler
- To implement and test simple optimization techniques
- To give exposure to compiler writing tools

1.3. List of Exercises

1. Design and implement a lexical analyzer for a given language and the lexical analyzer should ignore redundant spaces, tabs and new lines
2. Implementation of lexical analyzer using lex tool
3. Generate YACC Specification for a few syntactic categories
    - Program to recognize a valid arithmetic expression that uses operators +, -, *, /
    - Program to recognize a valid variable which starts with a letter followed by any number of letters and digits
    - Implementation of Calculator using Lex and YACC
    - Convert the BNF Rules into YACC Form and write code to generate abstract syntax tree
4. Write program to find ϵ - closure of all states of any given NFA with ϵ transitions
5. Write program to convert NFA with ϵ transitions to an NFA without ϵ transitions
6. Write program to convert an NFA to DFA
7. Write program to minimize any given DFA
8. Develop an operator precedence parser for a given language
9. Write program to simulate First and Follow of any grammar
10. Construct a recursive descent parser for a given language
11. Construct a shift reduce parser for a given language
12. Write a program to perform loop unrolling
13. Write a program to perform constant propogation
14. Implement intermediate code generation for simple expressions
15. Implement the back end of a compiler which takes as input a three address code and produces the 8086 assembly language instructions that can be assembled and run using an 8086 assembler. The target assembly instructions can be Move, Add, Sub, Jump etc.

2. Finite Automata

In the theory of computation, a branch of theoretical computer science, a deterministic finite automaton (DFA)—also known as a deterministic finite acceptor (DFA) and a deterministic finite state machine (DFSM) or a deterministic finite state automaton (DFSA)—is a finite-state machine that accepts or rejects strings of symbols and only produces a unique computation (or run) of the automaton for each input string. Deterministic refers to the uniqueness of the computation. In search of the simplest models to capture finite-state machines, Warren McCulloch and Walter Pitts were among the first researchers to introduce a concept similar to finite automata in 1943.

Formally, A deterministic finite automata (DFA) is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$, where each of them represents:-

- a finite set of states Q
- a finite set of input symbols called the alphabet Σ
- a transition table δ consisting of a single state as entry for each state-symbol pair
- a start state q_0
- a set of final states F

Similarly, An NFA (Non-Deterministic Finite Automata) without ϵ is also represented by a 5-tuple where the transition table contains an element of the power set of Q as an entry for each state-symbol pair. An NFA with ϵ will also be represented by a 5-tuple where the transition table will contain an element of the power set of Q as an entry for each state-symbol pair as well as state-ϵ pair.

Any language that is represented by an NFA with ϵ transitions can be represented by an NFA without ϵ transitions and any language that can be represented by an NFA without ϵ transitions can be represented by a Deterministic Finite Automata (DFA). An NFA without ϵ transitions is a more restricted version of an NFA with ϵ transitions and a DFA is a more restricted version of an NFA without ϵ transitions.

2.1. Lab - Computing the ϵ closures of all states of an NFA with ϵ transitions

For convenience, the programming language used to explain concepts in the lab manual will be C++.

Input:- ϵ-NFA
Output:- List of ϵ-closures of all states of the NFA

Initially, we consider the NFA with ϵ transitions to be represented by objects of a structure of the form,

```
struct enfa
{
int num_states, num_alphabets;
vector<int> final_states;
vector<vector<vector<int> > > transititon_table;
};
```

We consider the default start state of all finite automata in the upcoming exercises of the section to be state 0. We also assume the symbols in the alphabet are numbered from 0 to $num\_alphabets$, which represents the number of symbols in the input alphabet of the NFA with ϵ transitions.

Vectors are dynamically sized arrays which has the ability to resize itself automatically as and when an element is inserted or deleted. The following link provides a basic documentation of the functions associated with vectors:- Vector Functions. Vectors performs similar functions as performed by lists in Python.

A convention used throughout is that a vector which contains the element -1 only is considered to be the null set. $transition\_table[i][0]$ represents the entry in the epsilon transition entry for state i in the transition table, while $transition\_table[i][j]$ represents the entry in the transition table for state i and input symbol $j-1$. Obtain an ϵ-NFA as input from the user using the following function,

```
enfa get_enfa()
```

The above function should obtain the variables of the enfa from the user through command prompt. The next step is to compute the closure of all states of the ϵ-NFA. **ϵ-closure of a state q is defined as the set of states along with q that can be reached by only following ϵ-transitions.**

The function which computes the closure of a state of the ϵ-NFA will have the form,

```
vector<int> compute_closure(enfa a, int k)
```

which will compute the closure of ϵ-NFA a for the state k and return the ϵ-closure of state k in a vector. The algorithm for the above function is provided as follows:-

- Initialize a list t containing only state k
- Initialize an iterator to the first element of the list t
- While iterator has not crossed the last element of the list t
  - Append all states in the i-ϵ pair in the transition table of ϵ-NFA a which is not previously present in list t to t
  - Set the iterator to the next element of the list t
- Return list t as the ϵ-closure for state k in ϵ-NFA a

2.2. Lab - Program to convert an NFA with ϵ transitions to an NFA without ϵ transitions

Input:- ϵ-NFA
Output:- NFA without ϵ-transitions

We consider an NFA without epsilon transitions to be represented by objects of a structure of the form,

```
struct nfa
{
int num_states, num_alphabets;
vector<int> final_states;
vector<vector<vector<int> > > transititon_table;
};
```

Alternatively, an object of the struct $enfa$ can be used to reprsent an NFA without ϵ transitions where all the state-ϵ pairs in the transitions will correspond to NULL sets.

As in the previous exercise, we consider the default start state of the NFA to be 0. $transition_table[i][j]$ represents the entry in the transition table for state i and input symbol j.

Obtain an NFA with ϵ transitions as input from the user. Now the aim of the exercise is to convert an NFA with ϵ-transitions to an NFA without ϵ-transitions, which will be performed by a function having the form,

```
nfa convert_enfa(enfa a)
```

The algorithm to compute the above provided function is provided as follows,

```
• Initialize an empty object of type $nfa$ 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 $\epsilon$ closure of state $i$ of $\epsilon$-NFA $a$
    – Iterate through each of the input symbol $j$ in $\Sigma$
        ∗ Initialize an empty list of states $f$
        ∗ Iterate through each state $k$ in $l$
            · Add all states of $a.transition\_table[k][j + 1]$ to $f$
        ∗ Remove all the duplicates from $f$
        ∗ Compute the $\epsilon$-closure $c$ of $f$
        ∗ Set $t.transition\_table[i][j] = c$
• Return $t$ as the NFA without $\epsilon$-transitions corresponding to the $\epsilon$-NFA $a$
```

2.3. Lab - Program to convert an NFA to DFA

Input:- NFA without ϵ-transitions
Output:- DFA

We represent DFA by objects of the structure,

```
struct dfa
{
int num_states, num_alphabets;
vector<int> final_states;
vector<vector<int> > transititon_table;
};
```

As usual, we consider state 0 to be the default start state of the DFA. Also, since in a DFA there exists exaactly one state transition for every state-symbol pair, $transition\_table[i][j]$ represents the state to which it undergoes transition on encountering input symbol j from state i.

We obtain an NFA without ϵ-transitions as input from the user, which is performed by a function of the form,

```
nfa get_nfa()
```

Now the objective of the exercise to convert it into a corresponding DFA. We will perform this conversion using the subset conversion method. Let the number of states in the NFA be num_states and the number of input symbols in the alphabet of the NFA be $num_alphabets$. Then, the number of states in the DFA initially will be $2^{(num\_states)}$ and the alphabet of the DFA will consist of $num\_alphabets$ input symbols. The function which computes the corresponding DFA will have the form,

```
dfa convert_nfa(nfa a)
```

The algorithm to compute the above function is as follows,

- Initialize an empty object of type dfa having variable name t
- Initialize $t.num\_states = 2^{(a.num\_states)}$
- Initialize $t.num\_alphabets = a.num\_alphabets$
- Let Q_t represent the states present in the DFA t and Σ_t represent the symbols in the alphabet of the DFA t. Iterate through each of the states i in Q_t
  - Let x be the subset of states of Q (set of states of the NFA a) corresponding to state i in DFA a which is obtained by the function $nfa\_state(int\ i)$ ($nfa\_state(int\ i)$ will be described later)
  - Iterate through each input symbol j in Σ_t
    - ∗ Initialize an empty list of states f
    - ∗ Iterate through each of the states k in x
      - · Append the states present in $a.transition\_table[k][j]$ to f
    - ∗ Remove all the duplicate states from f
    - ∗ If f is empty, set $t.transition\_table[i][j] = 2^{(a.num\_states)} - 1$; Else, Identify the DFA state c corresponding to the subset of states f of the NFA using the function $dfa\_state(vector < int > \ f)$ and set $t.transition\_table[i][j] = c$

- Consider all the subsets of $a.final\_states$, identify the DFA state corresponding to it and include it in the set $t.final\_states$ (set of final states of the DFA t)

The function, all the elements of the power set of the set of states of NFA, according to the binary representation of numbers between 1 and $2^{(a.num\_states)}$. The null set is assigned a DFA state number of $2^{(a.num\_states)} - 1$. The function $dfa\_state(vector < int > a)$, computes the DFA state as follows,

```
int dfa_state(vector<int> a)
{
int j = 0;
for(int i = 0; i < a.size(); i++)
{
j += pow(2, a[i]);
}
return (j-1);
}
```

The function $nfa\_state(int\ i)$ will increment i, compute it's binary representation, and then add the unit positions where 1 occurs into to a set of states. This subset of states, then corresponds to the DFA state i.

Optionally, we can identify the unreachable states from state 0 by running a Breadth First Search starting at state 0, when the DFA is considered as a graph. All the states that have not been traversed during the BFS traversal can be eliminated from the DFA. Correspondingly, the variables $t.num\_states$, $t.transition\_table$ and $t.final\_states$ has to be modified.

2.4. Lab - Program to minimize any given DFA

Input:- DFA
Output:- Minimized DFA corresponding to the input DFA

We use an object of the same structure to represent a DFA. Obtain a DFA as input from the user using a function of the form,

```
dfa get_dfa()
```

Now, the objective of the exercise is to minimize the given DFA, which will be performed using a function of the form,

```
dfa minimize_dfa(dfa a)
```

The algorithm to compute the above provided function is as follows,

- Initialize a list of lists t which contains two lists, where $t[0]$ contains the list of non-final states and $t[1]$ contains the list of final states
- Initialize a matrix m of size $a.num\_states * a.num\_states$ and set every cell in the matrix to 0
- Initialize a flag variable f to 1
- For all state pairs (x, y), Set $m[x][y] = 1$ if x is a final state and y is a non-final state or vice-versa
- While f is set to 1
  - Set f to 0
  - For all states i from 0 to $a.num\_states - 2$
    - For all states j from $i + 1$ to $a.num\_states - 1$
      - If for any symbol u in Σ, $(m[i][j] = 0$ and $m[a.transition\_table[i][u]][a.transition\_table[j][u]] = 1)$, Then Set $m[i][j] = 1$ and $f = 1$
- Initialize a DFA object and represent those pair of states (a, b) which has $m[a][b] = 0$ by a single state a in the minimized DFA

The above method of minimization of DFA is using the Myhill-Nerode Theorem.