CS-3404: MINOR PROJECT

IMPLEMENTATION OF SOME COMPLEX PROBLEMS IN AUTOMATA AND FORMAL LANGUAGES

Mentor:

Prof. A K Agrawal

Submitted By:

Prakhar Jain 10400EN002
Shubham Gupta 11400EN001

ACKNOWLEDGEMENT

We have taken efforts in this project. However, it would not have been possible without the kind support and help of many individuals. We would like to extend our sincere thanks to all of them.

We are highly indebted to **Prof. A K Agrawal** for his guidance and constant supervision as well as for providing necessary information regarding the project & also for their support in completing the project.

We would like to express our gratitude towards our parents for their kind co-operation and encouragement which help us in completion of this project.

We would like to express our special gratitude and thanks to industry persons for giving us such attention and time.
Our thanks and appreciations also go to our colleague in developing the project and people who have willingly helped us out with their abilities.

SYNOPSIS

Implementation of some complex problems in Automata and Formal Languages

Introduction

Automata theory deals with the definitions and properties of mathematical models of computation. These models play a role in several applied areas of computer science. One model, called the *finite automaton*, is used in text processing, compilers, and hardware design. Another model, called the *context-free grammar*, is used in programming languages and artificial intelligence.

Automata theory is an excellent place to begin the study of the theory of computation. The theories of computability and complexity require a precise definition of a *computer*. Automata theory allows practice with formal definitions of computation as it introduces concepts relevant to other non-theoretical areas of computer science.

Finite Automata

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where
1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta: Q \times \Sigma \longrightarrow Q$ is the ***transition function***.
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \cdots w_n$ be a string where each w_i is a member of the alphabet Σ. Then M ***accepts*** w if a sequence of states $r_0, r_1 \ldots r_n$ in Q exists with three conditions:

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0 \ldots n-1$, and
3. $r_n \in F$.

Condition 1 says that the machine starts in the start state. Condition 2 says that the machine goes from state to state according to the transition function. Condition 3 says that the machine accepts its input if it ends up in an accept state. We say that M ***recognizes language*** A if $A = \{w \mid M \text{ accepts } w\}$.

REGULAR LANGUAGE

A language is called a ***regular language*** if some finite automaton recognizes it.

REGULAR OPERATIONS

We define three operations on languages, called the ***regular operations***, and use them to study properties of the regular languages.

Let A and B be languages. We define the regular operations ***union***, ***concatenation***, and ***star*** as follows:
- **Union**: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.
- **Concatenation**: $A \circ B = \{x\,y \mid x \in A \text{ and } y \in B\}$.
- **Star**: $A^* = \{x_1 x_2 ... x_k \mid k \geq 0 \text{ and each } x_i \in A\}$.

NON DETERMINISM

When the machine is in a given state and reads the next input symbol, we know what the next state will be—it is deter- mined. We call this ***deterministic*** computation. In a ***nondeterministic*** machine, several choices may exist for the next state at any point.

The difference between a deterministic finite automaton, abbreviated DFA, and a nondeterministic finite automaton, abbreviated NFA, is immediately apparent. Every state of a DFA always has exactly one exiting transition arrow for each symbol in the alphabet. In an NFA, a state may have zero, one, or many exiting arrows for each alphabet symbol.
If a NFA has an arrow with the label ε. It is called **ε-NFA**. In general, a **ε-NFA** may have arrows labeled with members of the alphabet or ε. Zero, one, or many arrows may exit from each state with the label ε.

Nondeterministic finite automata are useful in several respects. As we will show, every NFA can be converted into an equivalent DFA, and constructing NFAs is sometimes easier than directly constructing DFAs. An NFA may be much smaller than its deterministic counterpart, or its functioning may be easier to understand. Non-determinism in finite automata is also a good introduction to non-determinism in more powerful computational models because finite automata are especially easy to understand.

REGULAR EXPRESSION

R is a **regular expression** if R is
1. a for some a in the alphabet Σ,
2. ε,
3. ∅,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

In items 1 and 2, the regular expressions a and ε represent the languages {a} and {ε}, respectively. In item 3, the regular expression ∅ represents the empty language. In items 4, 5, and 6, the expressions represent the languages obtained by taking the union or concatenation of the languages R_1 and R_2, or the star of the language R_1, respectively.

CONTEXT-FREE LANGUAGES

The collections of languages associated with context-free grammars are called the ***context-free languages***. They include all the regular languages and many additional languages. ***Pushdown automata*** are a class of machines recognizing the context-free languages. Pushdown automata are useful because they allow us to gain additional insight into the power of context-free grammars.

CONTEXT-FREE GRAMMAR

A ***context-free grammar*** is a 4-tuple (V, Σ, R, S), where
1. V is a finite set called the ***variables***,
2. Σ is a finite set, disjoint from V, called the ***terminals***,
3. R is a finite set of ***rules***, with each rule being a variable and a string of variables and terminals, and
4. S ∈ V is the start variable.

AMBIGUITY

A string w is derived ***ambiguously*** in context-free grammar G if it has two or more different leftmost derivations. Grammar G is ***ambiguous*** if it generates some string ambiguously.

CHOMSKY NORMAL FORM

A context-free grammar is in ***Chomsky normal form*** if every rule is of the form
$A \rightarrow BC$

$A \rightarrow a$
Where a is any terminal and A, B, and C are any variables—except that B and C may not be the start variable. In addition, we permit the rule $S \rightarrow \varepsilon$, where S is the start variable.

PUSHDOWN AUTOMATA

Pushdown Automata are like nondeterministic finite automata but have an extra component called a ***stack***. The stack provides additional memory beyond the finite amount available in the control. The stack allows pushdown automata to recognize some non-regular languages.
Pushdown automata are equivalent in power to context-free grammars. This equivalence is useful because it gives us two options for proving that a language is context free. We can give either a context-free grammar generating it or a push- down automaton recognizing it. Certain languages are more easily described in terms of generators, whereas recognizers more easily describe others.

A ***pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and
1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow P(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

TURING MACHINES

A very powerful model, first proposed by Alan Turing in 1936, called the ***Turing machine***. Similar to a finite automaton but with an unlimited and unrestricted memory, a Turing machine is a much more accurate model of a general-purpose computer. A Turing machine can do everything that a real computer can do. Nonetheless, even a Turing machine cannot solve certain problems. In a very real sense, these problems are beyond the theoretical limits of computation.

The Turing machine model uses an infinite tape as its unlimited memory. It has a tape head that can read and write symbols and move around on the tape. Initially the tape contains only the input string and is blank everywhere else. If the machine needs to store information, it may write this information on the tape. To read the information that it has written, the machine can move its head back over it. The machine continues computing until it decides to produce an output. The outputs accept and entering designated accepting and rejecting states obtain reject. If it doesn't enter an accepting or a rejecting state, it will go on forever, never halting.

A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** ␣,
3. Γ is the tape alphabet, where ␣∈ Γ and Σ ⊆ Γ, ▪
4. δ: Q×Γ−→Q×Γ×{L, R}is the transition function,
5. $q_0 \in Q$ is the start state, ▪
6. $q_{accept} \in Q$ is the accept state, and▪
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

Problems Covered

Non Deterministic Finite Automata (NFA) to Deterministic Finite Automata (DFA)

Proving Equivalence of DFA, NFA and ε-NFA

We will prove that every NFA or ε-NFA is DFA by converting it into its equivalent DFA. To remove its non-deterministic nature, we need to remember every subset of states of NFA or ε-NFA.
If k is the number of states of the NFA, it has 2^k subsets of states. Each subset corresponds to one of the possibilities that the DFA must remember, so the DFA simulating the NFA will have 2^k states. Now we need to figure out which will be the start state and accept states of the DFA, and what will be its transition function.

Let N = (Q, Σ, δ, q_0 , F) be the NFA recognizing some language A. We construct a DFA M =(Q', Σ, δ, q_0', F') recognizing A. Let's first consider the case when NFA has no ε arrows i.e. it is not an **ε-NFA.**

1. Q' = P (Q). Every state of M is a set of states of N. Recall that P (Q) is the set of subsets of Q.
2. For R ∈ Q' and a ∈ Σ, let δ'(R, a) = {q ∈ Q| q ∈ δ(r, a) for some r ∈ R}. If R is a state of M, it is also a set of states of N. When M reads a symbol a in state R, it shows where a takes each state in R. Because each state may go to a set of states, we take the union of all these sets.
3. q_0' = {q_0}. ▪ M starts in the state corresponding to the collection containing just the start state of N.
4. F' = {R ∈Q'| R contains an accept state of N}. ▪ The machine M accepts if one of the possible states that N could be in at this point is an accept state.

Now let us convert it into ε-NFA. To do so, we set up an extra bit of notation. For any state R of M, we define E(R) to be the collection of states that can be reached from members of R by going only along ε arrows, including the members of R themselves. Formally, for R ⊆Q let
E(R) = {q| q can be reached from R by traveling along 0 or more ε arrows}.

Then we modify the transition function of M to place additional fingers on all states that can be reached by going along ε arrows after every step. Replacing δ(r, a) by E (δ(r, a)) achieves this effect. Thus

$\delta'(R, a) = \{q \in Q \mid q \in E(\delta(r, a)) \text{ for some } r \in R\}$.

Additionally, we need to modify the start state of M to move the fingers initially to all possible states that can be reached from the start state of N along the ε arrows. Changing q_0' to be $E(\{q_0\})$ achieves this effect. We have now completed the construction of the DFA M that simulates the NFA N.

The construction of M obviously works correctly. At every step in the computation of M on an input, it clearly enters a state that corresponds to the subset of states that N could be in at that point. Thus our proof is complete.

Data Structures Used

3-D Array to represent NFA:

```
// g[s1][i][s2] = true if and only if there's an edge with symbol i from state s1 to s2.
static boolean[][][] nfa = new boolean[maxn][symbol][maxn];
```

2-D Array to represent DFA:

```
// supports for max states of maxn/20
// dfa[s1][i] = s2 if there's an edge with symbol i from state s1 to s2
static int[][] dfa = new int[1 << (maxn/20)][symbol];
```

Global Variables:

```
static int maxn = 200; // maximum number of states
static int symbol = 2; // number of symbols
static int state = 0; // current number of states
```

An ArrayList for storing the nodes, which is an active subset:

```
ArrayList<Integer> toVisit = new ArrayList<Integer>();
```

Algorithm Used

```
public static void nfa_to_dfa(ArrayList<Integer> nfa) {
        ArrayList<Integer> toVisit = new ArrayList<Integer>();
        for (int i = 0; i < state; i++)
```

```
            toVisit.add(i);
        for (int i : toVisit) {
            for (int j = 0; j < symbol; j++) {
                    int temp = 0;
            for (int k = 0; k < state; k++) {
                    if (nfa[i][j][k]) {
                            temp = temp | (1<<k);
                    }
            }
            g3[i][j] = temp;
            toVisit[temp] = true;
            }
        }
}
```

ε-Non Deterministic Finite Automata (ε-NFA) to Non Deterministic Finite Automata (NFA)

Data Structures Used

3-D Array to represent ε-NFA:

```
// epsnfa[s1][i][s2] = true if and only if there's an edge with symbol i from state s1 to s2.
static boolean[][][] epsnfa = new boolean[maxn][symbol+1][maxn];
```

2-D Array to represent closure of states. E(R) as defined above:

```
// closure[s1][s2] is true if and only if s2 is in E(s1)
static boolean[][] closure = new boolean[maxn][maxn]
```

ArrayList to represent start node and end nodes of NFA:

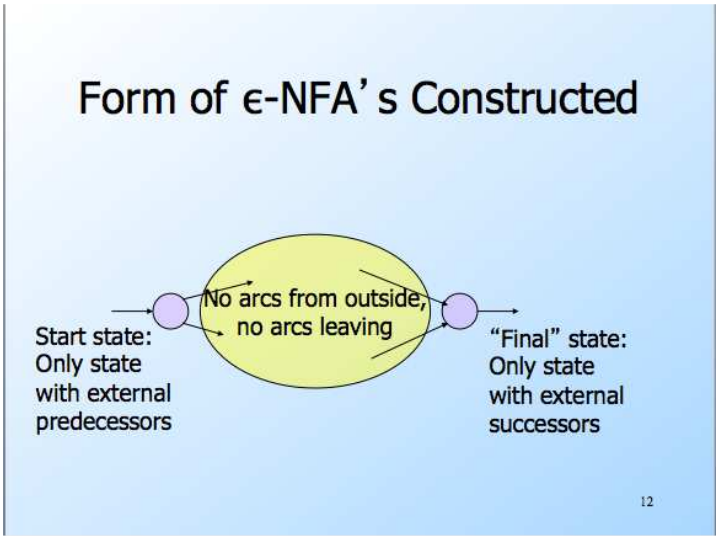
```
ArrayList<Integer> startAndEnd = new ArrayList<Integer>();
```

Algorithm Used

```
public static ArrayList epsnfa_to_nfa(int enfa[]) {
                // copy epsnfa to nfa
                for (int i = 0; i < state; i++) {
                        for (int j = 0; j < symbol; j++) {
                                for (int k = 0; k < state; k++) {
                                        nfa[i][j][k] = epsnfa[i][j][k];
                                }
```

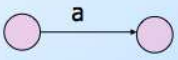
```
                }
            }
            // closure part
            for (int i = 0; i < state; i++) {
                for (int j = 0; j < state; j++) {
                    if (closure[i][j] == true) {
                        for (int k = 0; k < state; k++) {
                            for (int sym = 0; sym < symbol;
sym++) {
                                if (epsnfa[j][sym][k] == true
&& nfa[i][sym][k] == false)
                                    nfa[i][sym][k] = true;
                            }
                        }
                    }
                }
            }

            ArrayList<Integer> startAndEnd = new ArrayList<Integer>();
            nfa.add(enfa[0]);
            nfa.add(enfa[1]);
            for (int i = 0; i < state; i++) {
                if (closure[i][enfa[1]] == true)
                    if (!nfa.contains(i))
                        nfa.add(i);
            }
            return nfa;
        }
```

Regular Expression (RE) to ε-Non Deterministic Finite Automata (ε-NFA)



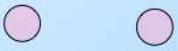
RE to ϵ-NFA: Basis

- Symbol **a**:



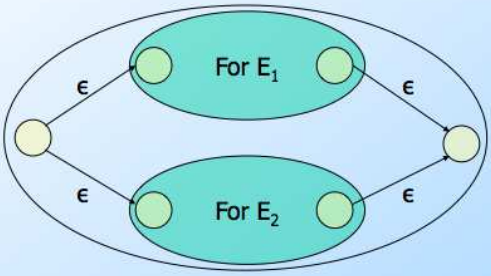
- ϵ:

- ∅:



13

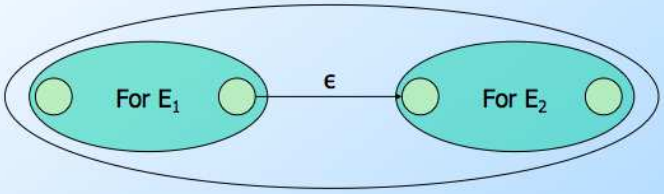
RE to ϵ-NFA: Induction 1 – Union



For $E_1 \cup E_2$

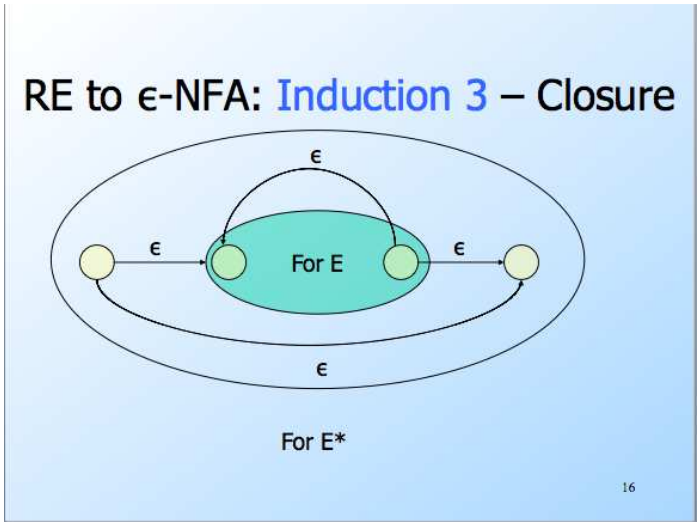
14

RE to ϵ-NFA: Induction 2 – Concatenation



For E_1E_2

15



Data Structures Used

3-D Array to represent ε-NFA:

```
// epsnfa[s1][i][s2] = true if and only if there's an edge with symbol i from state
s1 to s2.
static boolean[][][] epsnfa = new boolean[maxn][symbol+1][maxn];
```

2-D Array to represent closure of states. E(R) as defined above:

```
// closure[s1][s2] is true if and only if s2 is in E(s1)
static boolean[][] closure = new boolean[maxn][maxn]

// next[i]=i if the regular expression at position i is not '('
// next[i]=j if the regular expression at position i is '(' and jth position
// holds the corresponding ')'
static int[] next;
```

Code Snippet

```
public static void re_to_epsnfa(String re, String[] testRE) {
        for (int i = 0; i < testRE.length; i++) {
                System.out.println("Processing ...");
                calc_next(re);
                state = 0;
                int[] epsnfa = parse(re, 0, re.length() - 1);
                // calculate closure
                calc_closure();
                if (test(closure[epsnfa[0]], epsnfa[1], 0, testRE[i].length(),
testRE[i])) {
                        System.out.println("YES");
                }
                else {
```

```
                    System.out.println("NO");
                }
        }
}

static boolean test(boolean[] cur, int finalstate, int level, int len, String RE) {
        boolean[] next = new boolean[state];
        int i, j, k, c;
        if (level >= len)
                return cur[finalstate];
        if (RE.charAt(level) > '0')
                c = 1;
        else
                c = 0;
        for (i = 0; i < state; ++i)
                if (cur[i]) {
                        for (j = 0; j < state; ++j)
                                if (g[i][c][j]) {
                                        for (k = 0; k < state; ++k)
                                                next[k] = (next[k] || closure[j][k]);
                                }
                }
        boolean empty = true; // test if the state set is already empty
        for (i = 0; i < state; ++i)
                if (next[i])
                        empty = false;
        if (empty)
                return false;
        return test(next, finalstate, level + 1, len, RE);
}
```

Deterministic Finite Automata (DFA) to Regular Expression (RE)

Suppose regular expression R_{ij} represents the set of all strings that transition the automaton M from q_i to q_j . Furthermore, suppose R^k_{ij} represents the set of all strings that transition the automaton M from q_i to q_j without passing through any state higher than q_k. We can construct R_{ij} by successively constructing R^1_{ij} ; R^1_{ij} ; _ _ _ ; R^m_{ij}. R^k_{ij} is recursively defined as:

$R^k_{ij} = R^{k-1}_{ik} (R^{k-1}_{kk})^* R^{k-1}_{kj} + R^{k-1}_{ij}$

assuming we have initialized R^0_{ij} to be:

r if i != j and r transitions M from q_i to q_j
r +^ if i = j and r transitions M from q_i to q_j

o otherwise

As we can see, this successive construction builds up regular expressions until we have R_{ij} . We can then construct a regular expression representing M as the union of all $R_{\wedge F}$ where $q_{\wedge}$ is the starting state and f E M_f (the final states for M).
This technique is similar in nature to the all-pairs shortest path problem. The only difference being that we are taking the union and concatenation of regular expressions instead of summing up distances. This solution is of the same form as transitive closure and belongs to the constellation of problems associated with closed semirings.
The chief problem of the transitive closure approach is that it creates very large regular expressions. Examining the formula for an R^k_{ij} , it is clear the significant length is due to the repeated union of concatenated terms. Even by using the previous identities, we still have long expressions.

State Minimization of Deterministic Finite Automata (DFA)

For each regular language that can be accepted by a DFA, there exists a **minimal automaton**, a DFA with a minimum number of states and this DFA is unique (except that states can be given different names). The minimal DFA ensures minimal computational cost for tasks such as pattern matching.
There are two classes of states that can be removed/merged from the original DFA without affecting the language it accepts to minimize it.
Unreachable states are those states that are not reachable from the initial state of the DFA, for any input string.
Non-distinguishable states are those that cannot be distinguished from one another for any input string.
DFA minimization is usually done in three steps, corresponding to the removal/merger of the relevant states. Since the elimination of non-distinguishable states is computationally the most expensive one, it is usually done as the last step.

Algorithm Used

Let M = (Q, Σ, δ, q_0, F) be a DFA that accepts a language L. Then the following algorithm produces the DFA, denote it by M_1, that has the smallest number of states among the DFAs that accept L.

Let M = (Q, Σ, δ, q_0, F) be a DFA that accepts a language L. Then the following algorithm produces the DFA, denote it by M_1, that has the smallest number of states among the DFAs that accept L.

Construct a partition π={ A, Q - A } of the set of states Q ;

π_{new} := new_partition(π} ;
while (π_{new} != π)

$\pi := \pi_{new}$;
π_{new} := new_partition(π)
$\pi_{final} := \pi$;

function **new_partition( )**

for each set S of **do**

partition S into subsets such that two states p and q of S are in the same subset of S
if and only if for each input symbol, p and q make a transition to (states of) the same set of .

The subsets thus formed are sets of the output partition in place of S.
If S is not partitioned in this process, S remains in the output partition.
End

Minimum DFA M1 is constructed from π_{final} as follows:

Select one state in each set of the partition π_{final} as the representative for the set.
These representatives are states of minimum DFA M1.
Let p and q be representatives i.e. states of minimum DFA M1. Let us also denote by p and q the sets of states of the original DFA M
represented by p and q, respectively. Let s be a state in p and t a state in q. If a transition from s to t on symbol a exists in M, then the minimum
DFA M1 has a transition from p to q on symbol a.
The start state of M1 is the representative which contains the start state of M.
The accepting states of M1 are representatives that are in A.
Note that the sets of final are either a subset of A or disjoint from A.
Remove from M1 the dead states and the states not reachable from the start state, if there are any. Any transitions to a dead state become undefined.
A state is a **dead state** if it is not an accepting state and has no out-going transitions except to itself.

Implementation of Pushdown Automata (PDA)

Data Structures Used

```
static int state_pda = 1;
static int stack_state = 1;
static int pda_g[][][] = new int[maxn][symbol+1][maxn];
static String[] pda_t = new String[maxn];
static int start_state = 0;
static int start_symbol = 0;
```

Algorithm Used

```
public static boolean check(String test, int final_state) {
   Stack st = new Stack();
   st.push('0');
```

```
    int curr_state = 0;
    System.out.println("0");
    System.out.println(st.toString());
    for (int i = 0; i < test.length(); i++) {
        int c = (Character)st.peek() - '0';
        int temp = pda_g[curr_state][test.charAt(i)-'0'][c];
        if (temp == -1)
            return false;
        String t = pda_t[temp];
        if (t.length() == 0)
            return false;
        System.out.println(t.substring(0,1));
        curr_state = Integer.parseInt(t.substring(0,1));
        st.pop();
        for (int j = t.length()-1; j >= 1; j--) {
            st.push(t.charAt(j));
        }
        System.out.println(st.toString());
    }
    int c = (Character)st.peek() - '0';
    int temp = pda_g[curr_state][2][c];
    if (temp == -1)
        return false;
    String t = pda_t[temp];
    if (t.length() == 0)
        return false;
    System.out.println(t.substring(0,1));
    curr_state = Integer.parseInt(t.substring(0,1));
    st.pop();
    for (int j = t.length()-1; j >= 1; j--) {
        st.push(t.charAt(j));
    }
    System.out.println(st.toString());
    if (curr_state == final_state || st.isEmpty())
        return true;
    else
        return false;
}
```

Implementation of CYK Algorithm for testing Context-Free Grammar (CFG)

The Cocke–Younger–Kasami (CYK) algorithm (alternatively called CKY) is a parsing algorithm for context-free grammars, its name came from the inventors, John Cocke, Daniel Younger and Tadao Kasami. It employs bottom-up parsing and dynamic programming.

The standard version of CYK operates only on context-free grammars given in Chomsky normal form (CNF). However any context-free grammar may be transformed to a CNF grammar expressing the same language (Sipser 1997).

The importance of the CYK algorithm stems from its high efficiency in certain situations. Using Landau symbols, the worst case running time of CYK is theta (n^3 |G|), where n is the length of the parsed string and |G| is the size of the CNF grammar G. This makes it one of the most efficient parsing algorithms in terms of worst-case asymptotic complexity, although other algorithms exist with better average running time in many practical scenarios.

Data Structures Used

```
static int maxProductionNum = 100; //max number of productions
static int VarNum = 4;

int [][] production = new int[maxProductionNum+1][3];
//If this production is A->BC (two variables), then production[i][1] and
production[i][2] will contain the numbers for these two variables
//If this production is A->a (a single terminal), then production[i][1] will contain
the number for the terminal (0 or 1, 0: a, 1: b), production[i][2]=-1

boolean [][][] X;
//X[i][j][s]=true if and only if variable s (0~2, 0: S 1: A, 2: B, 3: C) is in X_ij
defined in CYK
//Suppose the length of string to be processed is L, then 0<=i<=j<L
```

Algorithm Used

```
void calcCYK(int [] w)
{
 int L = w.length;
 X=new boolean [L][L][VarNum];
 for (int i = 0; i < L; i++)
  for (int j=0; j < VarNum; j++)
   X[i][i][j] = existProd(j, w[i],-1);
 for (int s = 1; s < L; s++)
 {
  for (int i = 0; i < L-s; i++)
  {
   int j = i + s;
   for (int v = 0; v < VarNum; v++)
   {
```

```
      X[i][j][v] = false;
      for (int k = i; k < j; k++)
      {
       for (int v2 = 0; v2 < VarNum; v2++)
        for (int v3 = 0; v3 < VarNum; v3++)
         if (X[i][k][v2] && X[k+1][j][v3] && existProd(v,v2,v3))
           X[i][j][v]=true;
     }
    }
   }
  }
}
```

Implementation of Turing Machine (TM)

The Turing Machine Description Format we have used has the following conventions:

- They use a 1-way infinite tape.
- The tape alphabet has two different special symbols, □ and ⊔ that are not part of the □input alphabet.
- Initially, if the input string is w, the tape contains □w at the left end of the tape, and the rest of the tape contains only ⊔ symbols. The head of the Turing machine is initially positioned at the first character of the input string w (i.e., at the tape's second square).
- Whenever the Turing machine sees the □ symbol, it must leave it unchanged and move right (but it can change state).

We also make some naming conventions. We assume that the state set of the Turing machine is $Q = \{q_0, q_1, \ldots, q_{n-1}\}$ where $n \geq 3$ and the tape alphabet of the Turing machine is $\Gamma = \{c_0, c_1, \ldots, c_{m-1}\}$ where $m \geq 3$. We also assume that q_0 is the initial state, q_{n-2} is the accepting state and q_{n-1} is the rejecting state. We assume that the input alphabet is $\Sigma = \{c_0, c_1, \ldots, c_k\}$ where $0 \leq k \leq m-3$ and $c_{m-2} =$ ⊔ and $c_{m-1} =$ □.

Input/output Format

We use an underscore character to represent ⊔ and > to represent □. The first line of the file contains the three integers n, m, and k, separated by single spaces. (Recall that these are the sizes of the state set, tape alphabet and input alphabet, respectively.) The second line of the file contains a string $c_0c_1 \ldots c_{m-3}$ of length $m - 2$ which gives each character of the tape alphabet in order (except c_{m-2} = and c_{m-1} = >). All characters in this string should be distinct (and different from □> and). The third line contains a non-negative integer T.□
Following this, there are T lines. Each of these remaining lines of the description contains □five items i, a, i', a', d separated by single spaces, where i and i' are integers with $0 \leq i \leq n-3$ and $0 \leq i' \leq n - 1$ (inclusive), a and a' are characters in the tape alphabet and d is a single character that is either L or R. This line

indicates that $\delta(q_i, a) = (q_{i'}, a', d)$. No two lines should have the same i and a. Note that no transitions are given for situations when the machine is in state q_{n-2} or q_{n-1} since those are the accepting and rejecting states. If no transition is given to describe $\delta(q_i,a)$ for a non-halting state q_i, then it is assumed that $\delta(q_i, a) = (q_i, a, R)$.

Data Structures Used

```
final int numStates; // number of states in the state set
final int tapeAlphaSize; // number of characters in the tape alphabet
final int inputAlphaSize; // number of characters in the input alphabet
final String tapeAlpha;

int[][] nextState; // these three arrays are used to store the transition function
int[][] charToWrite;

boolean[][] moveLeft;

boolean[][] done = new boolean[numStates-2][tapeAlphaSize];
// temporary variable used to make sure that each transition is
// only defined once in the input

int state=0; // start in state 0
int pos = 1; // head position
Vector<Integer> tape = new Vector<Integer>(10000,0); // tape contents
```

Code Snippet

```
int stepCounter = 0;
if (trace) printConfig(out, stepCounter, tape, state, pos);
while (state < acceptState() && stepCounter++ < numSteps) {
   // simulate one step of the Turing machine
   if (pos == tape.size()) // extend tape if TM has run beyond right end
      tape.add(blank());
   int oldChar = tape.get(pos);
   tape.set(pos, charToWrite[state][oldChar]);
   pos = (moveLeft[state][oldChar]) ? pos - 1 : pos + 1;
   state = nextState[state][oldChar];
   if (trace) printConfig(out, stepCounter, tape, state, pos);
}
```

Problems Encountered

Modeling of different types of Machines (DFA, NFA, ε-NFA, RE, PDA, TM)

Deciding which model to use to represent DFA, NFA, ε-NFA was a major problem in the project. We need an efficient model to represent these automatons so that we are able to implement efficient time and space conversions of these automatons. Similarly, modeling PDA efficiently was required for its implementation.

Modeling Turing Machine was the most difficult work in the whole project. There are different variants of Turing Machine and deciding which variant to use in our implementation put us in big dilemma.

Solution

We decided a simple representation of DFA, NFA, ε-NFA using 3 Dimensional Arrays. Where, two variable represent two states and one a symbol between the states. We reduced DFA to a 2 Dimensional Array because of its deterministic nature. Here, the one variable represented a state and other a symbol. Its value represented the state where it went using this symbol.

We decided to make a new description format of the Turing Machine as described in the above section.

Conclusion

Studied various types of Languages and its properties.

Comparison of different types of Languages. Their PROS and CONS.

Scope and Future Work

Scope of finite automata algorithms is answered by these questions:

-is my system/language more powerful than others?
-is my system/language more efficient than others?
-expressive power or computational complexity can be studied by relating them to formal language theory: languages, grammars, automata, . . .
-tradeoff between expressive power and computational complexity
-consider restrictions of difficult problems or giving up
exact solutions.

Future work in the field of automata includes more improvement and innovation in the following areas:

- analysing problems/languages
-computability/solvability/decidability
— is there an algorithm?
-computational complexity
— is it practical?
-expressive power
— are there things that cannot be expressed?
-formal languages provide well-studied models

References

Introduction to the Theory of Computation by Michael Sipser, Third Edition.

Ullman, A. V., Hopcroft, J. E. *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

Automata Course at Coursera by Jeff Ullman, Stanford University.