Extracting Person Name, Date and Place from Text Document using LEX Tool

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Technology
in
Computer Science and Applications

Submitted By
Roohi Sharma
(601003031)

Under the supervision of:
Mrs. Sunita Garhwal
Assistant Professor, SMCA

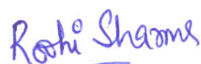
SCHOOL OF MATHEMATICS AND COMPUTER APPLICATIONS
THAPAR UNIVERSITY
PATIALA – 147004

June 2012

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, *"Extracting Person Name, Date and place from Text Document using LEX Tool"*, in partial fulfillment of the requirements for the award of degree of Master of Technology in *Computer Science and Applications* submitted in School of Mathematics and Computer Applications of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Mrs Sunita Garhwal* and refers other researcher's work which are duly listed in the reference section.

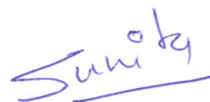
The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.



Signature:

(Roohi Sharma)

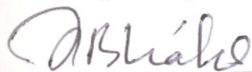
This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.



(Mrs. Sunita Garhwal)

Assistant Professor
SMCA
Thapar University
Patiala

Countersigned by



(Dr. S.S.Bhatia)
Head
School of Mathematics and Computer Applications
Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

Acknowledgement

Words are inadequate to express my gratitude to my thesis supervisor, **Mrs. Sunita Garhwal** Assistant Professor, School of Mathematics and Computer Applications, Thapar University, who has been very concerned and has aided for all the material essential for the preparation of this thesis report. She has helped me to explore this vast topic in an organized manner and provided me with all the ideas on how to work towards a research-oriented venture.

I am highly obliged to **Dr. S. S. Bhatia**, Head of Department, SMCA and **Mr. Singara Singh**, P.G. Coordinator for the motivation and inspiration that helped me for the thesis work.

I would also like to thank the staff members and my colleagues who were always there in the need and provided with all the help and facilities, which I required, for the completion of my thesis.

Most importantly, I would like to thank my parents and the almighty for showing me the right direction out of the blue, to help me stay calm in the oddest of the times and keep moving even at times when there was no hope.

Roohi Sharma

Roohi Sharma

(601003031)

ABSTRACT

Automata theory is closely related to formal language theory. Automata is used in designing and checking the behavior of digital circuits, lexical analysis, software for scanning large bodies of text and software verification, communications protocols. Finite automata have finite number of states. Finite automata provide efficient and convenient tools to represent the linguistic phenomena.

If a text document is searched manually to get some important information, the process is slow, tedious and error prone. Manual searching commonly misses important information in the document and it causes wastage of time. It has been noted that finite state automata based techniques have been widely used in natural language processing and information extraction in various natural languages.

In this thesis report various text extracting approaches are studied. Based on these approaches, a technique is given which extracts person name, place and date from text document using Lex tool. Regular expressions through which required information is extracted are also discussed.

Table of contents Page No.

Certificate i
Acknowledgement ii
Abstract iii
Table of Contents iv
List of Figures vi
List of Tables vii
Algorithms and Theorems vii
List of Abbreviations viii
Chapter 1: Introduction 1
1.1 Automata 1
1.2 Definitions and Notations 2
1.2.1 Alphabet and Language 2
1.2.2 Operations on Languages 3
1.2.3 Regular Language 4
1.2.4 Regular Expression 4
1.2.5 Deterministic Finite Automata 5
1.2.6 Non-Deterministic Finite Automata 6
1.2.7 Applications of FSA and RE 8
1.3 Lexical Analyzer 9
1.3.1 Lex Tool 10
1.4 Thesis Objective 12
1.5 Thesis Outline 12
Chapter 2: Literature Review 13
2.1 Applications of Regular Expressions 13
2.2 Finite State Machine and Information Extraction 15
2.3 Use of Lex tool for Information Extraction 22
Chapter 3: Problem Statement 28
3.1 Problem Statement 28
3.2 Thesis Motivation 29

Chapter 4: Extracting Person Name, Date and Place from Text Document using Lex Tool 30

4.1 Analysis of Existing Approaches 30

4.2 Proposed Approach 31

4.2.1. Thompson's Construction 31

4.2.2 Subset Construction 34

4.2.3 Information Extraction using FSA 34

Chapter 5: Conclusion and Future Scope 39

References 40

List of Publications 44

List of Figures Page No.

Fig 1.1 A FA for recognition of string a "start" 2
Fig 1.2 Languages and their RE 4
Fig 1.3 Transition Graph 6
Fig 1.4 Simulating a DFA 6
Fig 1.5 NDFA 7
Fig 1.6 Simulating a NFA 8
Fig 1.7 Interaction of Lexical Analyzer with Parser 9
Fig 1.8 Creating a Lexical Analyzer with Lex 11
Fig 2.1 Shortest-Method Substring Search Algorithm 13
Fig 2.2 Parsing Algorithm 15
Fig 2.3 Automaton representing the Patterns for Recognition of Company Name 16
Fig 2.4 IE system architecture 16
Fig 2.5 The Performance measure of System 17
Fig 2.6 System Overview 18
Fig 2.7 Architecture for place tagged text 19
Fig 2.8 The no. of DFA states splitted according to the size of DFA state set 22
Fig 2.9 Building Compiler with Lex and Yacc 24
Fig 2.10 Transition diagram for letters 25
Fig 2.11 Transition diagram for digits 25
Fig 2.12 Transition diagram for keywords 25
Fig 2.13 Transition diagram for whitespace 25
Fig 2.14 Lex program for token recognition 26
Fig 3.1 Regular expression to minimum DFA 28
Fig 4.1 NFA for ε 32
Fig 4.2 NFA for a 32
Fig 4.3 NFA for Union of two RE 32
Fig 4.4 NFA for Concatenation of two RE 33
Fig 4.5 NFA for Closure of RE 33

Fig 4.6 File “ppd.l” 36
Fig 4.7 Program “thesis.l” 37
Fig 4.8 Result (i) 38
Fig 4.9 Result (ii) 38

List of Tables Page No.

Table 1.1 Transition table representing transition function of DFA 5
Table 1.2 Transition table representing transition function of DFA 7
Table 1.3 Lex variables and Lex functions 10
Table 2.1 Results of Entropy on a clustering with Proper Name 18
Table 4.1 Operations on NFA states 34

Algorithms and Theorems Page No.

Algorithm 1 Simulating a DFA 6
Algorithm 2 Simulating a NFA 7

List of Abbreviations

BCAM	Binary Content Addressable Memory
CRF	Conditional Random Fields
DDC	Dewey Decimal Classification
DFA	Deterministic Finite Automata
DSE	Dynamic Symbolic Execution
FA	Finite Automata
FPGA	Field Programmable Gate Array
IDS	Intrusion Detection System
IE	Information Extraction

NE	Named Entity
NER	Named Entity Recognition
NFA	Nondeterministic Finite Automata
POS	Part Of Speech
RE	Regular Expression
SGML	Standard Generalized Markup Language
YACC	Yet Another Compiler Compiler

Chapter 1

Introduction

This chapter gives an introduction to automata, regular expression, regular languages and its applications.

1.1 Automata

Theory of computation is the branch of computer science and mathematics that deals with whether and how efficiently problems can be solved on a model of computation using an algorithm. Automata theory is the study of abstract machines and the computational problems that can be solved using these abstract machines. These abstract machines are called automata.

Automata are closely related to formal language theory as they are often classified by the class of formal languages that they are able to recognize [13, 28]. Automata used in designing and checking the behavior of digital circuits, in lexical analysis phase of compiler construction; which scans the input program character by character and groups them together to form tokens is always designed as a finite state system, in software for scanning large bodies of text and in software for verifying systems of all types that have finite number of distinct states.

Automata is a plural of Automaton i.e. a robot [7]. Finite state automata then a “robot composed of a finite number of states”. Informally, automata are a finite list of states with transitions between the states.

Finite State Automata is a mathematical model with following properties:

1. Finite state system has some inputs and outputs.
2. The system should always be in one of its finite state.
3. One state is start state and one or more states can be the final states.

It takes an input string and determines whether it’s a valid sentence of a language or not. So, Finite Automata are recognizers; they simply say “yes” or “no” about each possible input string.

There are two common ways of representing finite-state machines:

a) A *state table* is used to represent a finite-state machine by giving the values of the functions f and *g*.
b) A *state diagram* is a directed graph representation of a finite-state machine.

The circles are individual states [16]. The lines connecting the states represent the transitions, these lines are called *edges* and label on an edge represents characters that cause the transition from one state to another. An arrow leading to state "off" indicates that it is start state of finite automata. The states with double circles are called *accepting states or final states.* Entering an accepting state signifies recognition of a particular input string, and there is usually some sort of action associated with the accepting state. Following is an automata that accept the "start" string in figure 1.1.

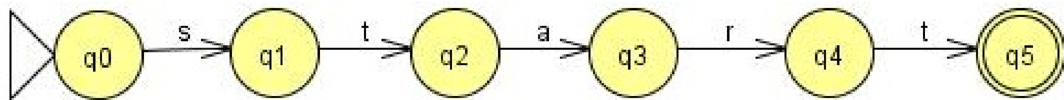


Figure 1.1: A FA for recognition of string "start" [28]

As shown in figure 1.2, a finite automata has six states and there is a transition from one state to other with particular input symbol. This finite automata recognizes the string "start" and each state of finite automata represents the different position of string that has been reached so far. Finite automata can be deterministic and non-deterministic. Every regular language that is described by non-deterministic finite automata can also be described by deterministic finite automata.

1.2 Definitions and Notations

This section contains some basic definitions and notations used in automata theory.

1.2.1 Alphabet and Language:

Definition 1: Alphabet is a finite non-empty set of symbols, on which a language is defined. Alphabet is denoted by Σ [19].

Example 1.1: Given an alphabet Σ ={a , b}. The strings corresponding to the alphabet can be a, b, ab, abab, aaabbb. The strings corresponding to an alphabet depend on the language over that alphabet.

The elements of $\sum$ are called *letters*. The notation $\sum^*$ means the set containing all the finite strings whose symbols are chosen from an alphabet. A *string* over an alphabet is a finite sequence of symbols drawn from that alphabet. The length of a string *s*, usually written |*s*|, is the number of occurrences of symbols in s. A formal language is a set of strings of symbols from some particular alphabet. The empty set [27] Ø, and the set consisting of the empty string {ε} are also languages, but they are distinct.

Formal languages can be classified as regular, context free, context sensitive and recursive language. Regular language [13, 18] can be described by regular expression, finite automata (Deterministic or Non-deterministic). For example, a language over 0 and 1 that will include all strings having length less than 2 is L= {0, 1, 00, 01, 10, 11}.

1.2.2 Operations on languages

Languages are like sets [15]. Any operation that can be performed on sets can also be performed on languages. A way of building more complex languages from simpler ones is to combine them using various operations. Union, concatenation and kleene closure are the major operation applied on languages.

1. **Union**: Given some alphabet $\sum$, for any two languages L_1, L_2 over $\sum$, the union L_1UL_2 of L_1 and L_2 is the language. Union of two languages L_1 and L_2 is set of all the strings that are in either L_1 or L_2.

   Example 1.2: If L_1={001,10,111} and L_2={01}, then L_1UL_2={01,001,10,111}.
2. **Concatenation**: The concatenation of two words a and b, written as ab or a.b, on the alphabet $\sum$ is the word obtained by writing down the letters of a followed by the letters of b. Concatenation of two languages L_1 and L_2 is set of all strings that can be formed by taking any string in L_1 and concatenating it with any string in L_2.

   Example 1.3: If L_1={001,10,111} and L_2={01} then $L_1.L_2$={00101,1001,11101}. Concatenation of languages is associative.
3. **Kleen Closure**: The unary operation L^* of a language L, called Kleene closure of L. Kleene closure of a language L represent the set of those strings that can be formed by taking any number of strings from L, possibly with repetitions(same string can be selected more than once) and concatenating all of them.

Example 1.4: If L={0,1}, then the Kleene closure of L is all strings that comprises of 0's and 1's i.e. L={0,1,01,001,0110,111010………}. L^+ is called positive closure of L.

1.2.3 Regular Language

If ∑ is an alphabet, the set R of regular languages over ∑ is defined as follows [11].

1. The language $\emptyset$ is an element of R, and for every a $\in$ ∑, the language {a} is in R.
2. For any two languages L_1 and L_2 in R, the three languages $L_1 \cup L_2$, L_1L_2, and L_1^* are elements of R.

A regular language is recognized by deterministic finite automata and non-deterministic finite automata.

The following are some regular expressions over alphabet ∑={a, b}, and corresponding regular languages:

Language	Regular Expression
{a}	a
{a,b}	a+b
{a,aa,aaa,…..}	a+
{^,a,aa,aaa,…}	a*

Figure 1.2: Languages and their Regular Expressions [17]

1.2.4 Regular Expression

A regular expression is pattern that describes some set of strings. Regular expression provides a concise and flexible means to match (specify and recognize) strings of text, such as particular characters, words, or patterns of characters. Common abbreviations for regular expression include regex and regexp. The concept of regular expressions was first popularizes by utilities provided by UNIX distributions. They were invented and first studied by Stephen Kleene [9]. Regular expression over input alphabets Σ can be defined as:

1. Every input alphabet can be represented by itself.
2. Null language and null string represent themselves.

3. If r1 and r2 are regular expressions representing the languages l1 and l2 respectively, then:

3.1 Union of r1 and r2 is represented by r1 + r2.

3.2 Kleene closure of the regular expression is represented by (r1)*.

3.3 Concatenation of r1 and r2 is represented by r1r2.

4. Rule 3 can be defined recursively.

1.2.5 Deterministic Finite Automata

Deterministic finite automata can be represented using 5-tuples (Q, ∑, δ, q_0, F), where

Q → is a finite nonempty set of states.

∑ → is a finite alphabet of input symbols.

δ → is called transition function or next state function which maps from Q x ∑ ⟶ Q.

q_0 → is called the initial or start state and q_0 □ Q.

F → is set of final states and F□Q.

Deterministic finite automata has rules for transitions from one state to another represented by transition function which depends on the present state and present input alphabet.

Example 1.5: Finite automata that accepts only those strings which start with 1 and end with 0.

Finite automata transitions can be represented by transition table or transition diagram.

Table 1.1: Transition table represents transition function of DFA [27].

State \ **Input**	**0**	**1**
q_0	-	q_1
q_1	q_2	q_1
q_2	q_2	q_1

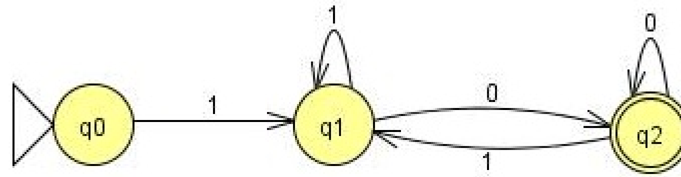


Figure 1.3: Transition graph [27]

Algorithm 1: Simulating a DFA [3].

INPUT: An input string *x* terminated by an end-of-file character eof. A DFA *D* with start state s_0, accepting states *F*, and transition function *move.*

OUTPUT: Answer "Yes" if *D* accepts x; "no" otherwise.

METHOD: Apply the algorithm in figure 1.4 to the input string *x*. The function *move(s, c)* gives the state to which there is an edge from state *s* on input *c*. The function *nextChar* returns the next character of the input string *x.*

```
s = s0;
c = nextChar ( );
while ( c != eof ) {
    s = move (s, c);
    c = nextChar (
    );
    }
if ( s is in F ) return "yes";
else return "no";
```

Figure 1.4: Simulating a DFA [3]

1.2.6 Non-Deterministic Finite Automata

A non-deterministic finite automata [13, 28] is a finite state machine where for each pair of state and input symbol there may be more than one next state.

Non-deterministic finite automata can be represented using 5-tuples (Q, ∑, δ, q_0, F),

Q → is a finite nonempty set of states.

∑ → is a finite alphabet of input symbols.

δ → is called transition function which maps from Q X {Σ U {ε}} → 2^Q.

q_0 → is called the initial or start state and q_0 ⍰ Q.

F → is set of final states and F⍰Q.

Example 1.6: Non-deterministic finite automata accepting all the strings terminating with 01.

On reading 1 on input state A, NFA remains in same state. But on reading 0 on A the choice of moves can be either on A or B.

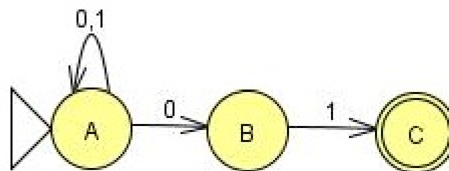


Figure 1.5: Non-deterministic finite automata [28]

The transition table for the figure 1.5 is shown below

Table 1.2: Transition table representing transition function of NFA [28]

State \ **Input**	**0**	**1**
A	{A, B}	{A}
B	_	{C}
C	_	_

Algorithm 2: Simulating a NFA [3].

INPUT: An input string *x* terminated by an end-of-file character eof. A NFA *N* with start state s_0, accepting states *F*, and transition function *move.*

OUTPUT: Answer "Yes" if *N* accepts x; "no" otherwise.

METHOD: The algorithm keeps a set of current states *S*, those that are reached from s_0 following a path labeled by the inputs read so far. If *c* is the next input character, read by the function *nextChar ( ),* then we first compute *move (S, c)* and then close that set using *ε-closure ( ).*

```
S = ε-closure (s0);
c = nextChar ( );
while ( c != eof ) {
S = ε-closure (move (S, c));
c = nextChar ( );
}
If (S □ F != Ø) return "yes";
else return "no";
```

Figure 1.6: Simulating an NFA [3]

For every NDFA, there exist a DFA which simulates the NDFA [26]. Alternatively, if L is the set accepted by some NDFA, then there exists a DFA which also accepts L. One can say that DFA and NDFA are of equal power as both can be designed from any regular language.

1.2.7 Applications of Finite Automata and Regular Expressions

Following are some of the applications in which finite state machine are used:

1. **Compiler construction phase:** Lexical Analyzer is a phase in compiler construction. A compiler is a program which converts a given source program in higher-level language into an equivalent machine language. This process is complex and therefore it is divided into lexical analysis, syntax analysis and semantic analysis, intermediate code generation, code optimization and code generation phases. The lexical analyzer uses DFA's. It reads a character from source program and based on current state and current symbol read, makes transition to some other states. When it reaches a final state of DFA, it makes the series of characters so far read and outputs the token found.
2. **Searching using Regular Expressions**: Many operating systems provide search utilities to perform frequent and complex jobs. For example, UNIX operating system provides a utility called 'grep'. Grep is a search utility and is very powerful. UNIX command grep "automata" tcs.txt searches the word automata in the file tcs.txt. The command will search for this word in file and will display the line

number(s) on which the word automata is found. The grep command is simulated using the finite automata corresponding to regular expression.

3. **Text Editors:** All popular text editors allow end user to find or replace facilities, to either find some particular word or find each occurrence of a particular word and replace each occurrence with some other word.

Basically we need to search for a particular word. Again these words can be represented in the form of regular expressions and the program then simulates the finite automata to perform the required job.

1.3 Lexical Analyzer

Lexical analysis [3] is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a lexical analyzer, lexer or scanner.

The lexical analyzer is the first phase of compiler. Its main task is to read the input characters and produces output a sequence of tokens that the parser uses for syntax analysis.

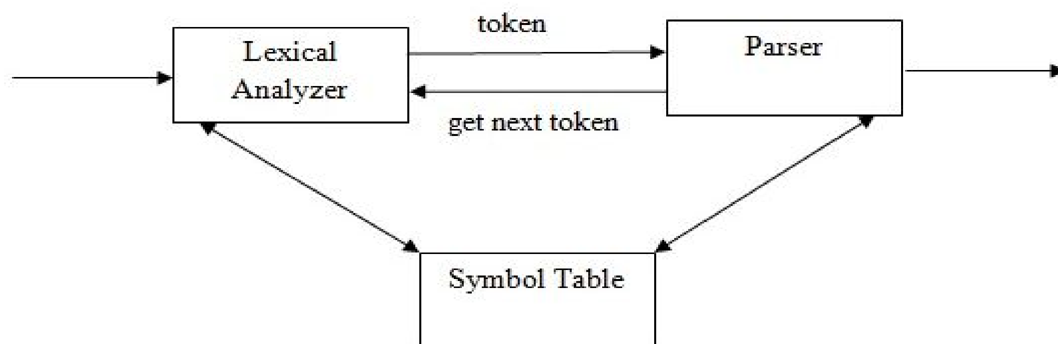


Figure 1.7: Interaction of lexical analyzer with parser [3].

Figure 1.7 represent interaction of lexical analysis with parser where, upon receiving a “get next token” command from the parser the lexical analyzer reads input characters until it can identify the next token.

Table 1.3: Lex variables and Lex Functions [8]

Lex variables and functions	Description
yyin	TYPE: FILE*. It point to the current file being parsed by the lexer.
yyout	TYPE: FILE*. It point to the location where the output of the lexer will be written. By default, both yyin and yyout point to standard input and output.
yytext	The text of the matched pattern is stored in the variable (char*).
yyleng	Specify length of the matched pattern.
yylineno	Current line number information.
yylex ( )	The function that starts the analysis. It is automatically generated by Lex.
yywrap ( )	This function is called when end of file (or input) is encountered. If this function returns 1, the parsing stops. This can be used to parse multiple files. Code can be written in the third section, which will allow multiple files to be parsed. The strategy is to make yyin file pointer (see the preceding table) point to a different file until all the files are parsed. At the end, yywrap ( ) can return 1 to indicate end of parsing.
yyless (int n)	This function can be used to push back all but first 'n' characters of the read token.
yymore ( )	This function tells the lexer to append the next token to the current token.

1.3.1 Lex Tool

Lex tool is used for automatically generating a lexer or scanner given a lex specification (.l file). A lexer or scanner is used to perform lexical analysis or the breaking up of an input stream into meaningful units, or tokens

For example, consider breaking a text file up into individual words.

The format of a Lex input file is as follow:

```
% {
definitions  % }
{ rules }
%%
{auxiliary routines}
```

A Lex program is divided into these sections:

1. Declaration section: It includes declarations of variables and name of token.
2. Rule section: It includes rule of the form Pattern {Action}.
3. Auxiliary routine section: It includes the main program and additional functions used in the actions.

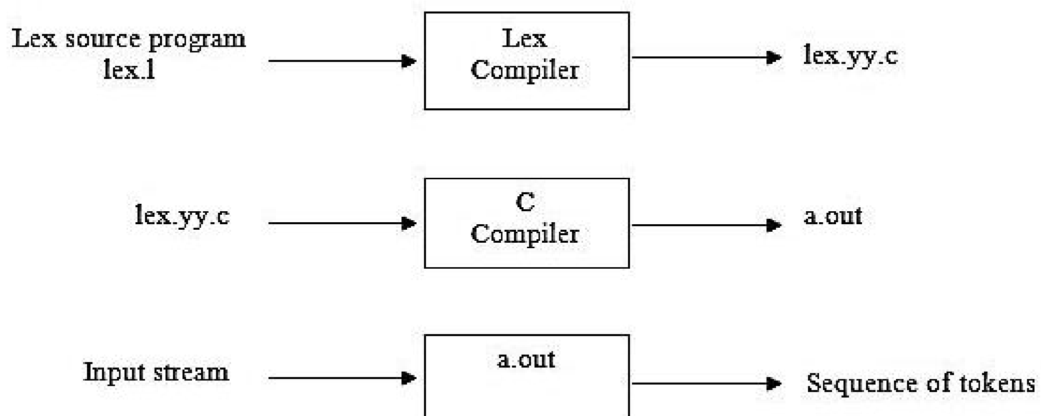


Figure 1.8: Creating a lexical analyzer with Lex [3]

Several tools have been built for constructing lexical analyzers from special purpose notations based on regular expressions. LEX is a tool that is widely used to specify lexical analyzers for a variety of languages.

Following are the steps used for generating a Lex with LEX tool:

1. A specification (file.l) is created.
2. The file.l is given as input to lex tool which will generate lex.yy.c.

3. A tabular representation is generated from Lex tool is used for identifying tokens.
4. The C program lex.yy.c is run through the C compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

1.4 Thesis Objective

The aim of this thesis is to identify and extract person name, place and date from a text file using LEX tool and finite state machine (FSA). While an FSA has powerful representations for regular languages, it is well known that the linguistic expressiveness of the FSA fails to fully recognize natural languages such as English, Chinese and others.

The specific goals of this thesis are:

1. Analysis of existing approaches used for person name identification in documents.
2. Comparison of existing approaches used for information identification in documents.
3. Identification and extraction of information using Lex tool.

1.5 Thesis Outline:

This thesis is divided into five chapters:

Chapter 1: This chapter describes background information relating to deterministic finite automata, regular expression.

Chapter 2: This chapter examines various research papers that are related to Finite State Automata. It discusses previous works that form the roots of subsequent research into Theory of Computation; the ideas and concepts of finite state automata.

Chapter 3: This chapter presents the problem definition.

Chapter 4: This chapter presents the analysis and comparison of existing approaches for information identification and describes the approach for extracting information. This chapter presents the evaluation strategy that is followed and the experimental results that are obtained.

Chapter 5: This chapter summarizes the conclusions and future scope.

Chapter 2

Literature Survey

This chapter describes the various works that has been done in field of finite state automata and regular expressions. The review work is divided into three parts:

2.1 Applications of Regular Expression

A regular expression defines simple string patterns. They were invented and first studied by Stephen Kleene [4]. The use of regular expressions for text search is widely known and well understood. The standard techniques and tools prove have limited use for searching structured text formatted with SGML or similar markup languages. Many people routinely use regular expressions to specify searches in text editors and with standalone search tools such as the Unix grep utility. In 1997, Charles L. A. et al. [16] propose a rule which is semantically cleaner.

```
1    for j ← 1 to |Q| do
2         P_j ← 0;
3    for i ← 1 to n do begin
4         P_1 ← i;
5         for j ← 1 to |Q| do
6              P'_j ← 0;
7         for j ← 1 to |Q| do
8              for q ∈ δ(j, a_i) do
9                   P'_q ← max(P'_q, P_j);
10        u ← 0;
11        for j ← 1 to |Q| do
12             if j ∈ F then u ← max(u, P'_j);
13        if u > 0 then begin
14             Output (u, i);
15             for j ← 1 to |Q| do
16                  if P'_j ≤ u then P'_j ← 0;
17        end;
18        for j ← 1 to |Q| do
19             P_j ← P'_j;
20   end;
```

Figure 2.1: Shortest-match substring search algorithm [16]

This rule is generally applicable to a variety of text search applications, including source code analysis, and has interesting properties in its own right. Researchers have written a publicly available search tool implementing the theory in the article, which has proved valuable in a variety of circumstances [16]. The algorithm gives two invariants for array P. Here t is the first element of previous output. With an output, t is effectively updated.

1. If $P_j <> 0$, then j is not final state output and string $x[P_j, i]$ becomes shortest suffix of $x[t+1, i]$.
2. If $Pj = 0$, there is no suffix of $x[t+1, i]$.

It is well known that the family of languages accepted by finite-state automata (FAs) is the same as the family of languages defined by regular expressions. It can be proved by showing that we can construct FAs from regular expressions and that we can construct regular expressions from FAs. There are several number of FA constructions from regular expressions and each construction has different properties. On the other hand, there are few methods to capture a regular expression from a given FA such as linear equations and state elimination. Researchers examine the minimization of FAs to obtain shorter expressions first [30]. They prove that bridge states are not eliminated until all non-bridge states are eliminated to obtain shorter regular expressions.

Regular Expressions are used to parse textual data to match patterns and extract variables. They have been implemented in a vast number of programming languages with a significant quantity of research devoted to improving their operational efficiency. However, regular expressions are limited to finding linear matches. Little research has been done in the field of object-oriented results which would allow textual or binary data to be converted to multi-layered objects. Studies have been done on object oriented regular expressions that presents a novel algorithmic approach to perform object-oriented parsing [9]. One of the key problems of regular expressions in modern-day programming is that they lack object-oriented extraction. This approach is relevant in both object-oriented programming languages and in object-oriented data representation. The parsing algorithm used to continue search for matching is a follows.

1. Generate a new match tree with the given text.
2. Repeat while there are new matches:

a. Iterate over each Complex expression in the ruleset.

    i. Iterate over every raw character position in the Match Tree and call the Complex expression's match algorithm, adding the resulting match to the Match Tree.

3. Call the getData algorithm on the first stack element and return the result.

Figure 2.2: Parsing Algorithm [9]

The use of regular expressions over the web sites is very common. On the web, static pages fade into the past; web sites use server and client-side scripting techniques to improve the user experience. In 2010, R. Hodovan et al. [24] take a closer look at regular expressions on the web. They investigate historical data and determine the trends of the use of regular expressions over time on various web sites. There investigations revealed that the number of regular expressions on web pages and those parsed during web browsing increased dramatically in the last few years.

2.2 Finite State Machine and Information Extraction

The finite state machine has been successfully used in various areas of computational linguistics: lexical analysis, morphology and phonology, local syntax, text-to-speech synthesis and speech recognition. The theory of automata provides efficient and convenient tools for the representation of linguistics phenomena. In 1995, a researcher proposed pseudo codes for the determinization of sting to string transducers, the deterministic union of p-subsequential string to string transducers, and all the indexation by automata [19]. This algorithm can be used in various natural language processing applications to improve the space and time efficiency. Over the years, various problems in morphology, including those in non-concatenative morphology have been solved using finite-state devices [26].

The information society will gradually provide its members with nearly unlimited access to all sorts of information. In 1999, J. Piskorski et al. [13] have presented an advanced domain-independent shallow text extraction and navigation system for processing of real-world German text. The system is implemented based on advanced weighted finite state machines and uses sophisticated linguistic knowledge sources. The system is very robust and efficient (at least from an application point of view) and has a good coverage [13].

Currently the paper focuses on the German language but has started to consider English as well, in order to support multi-lingual shallow text processing.

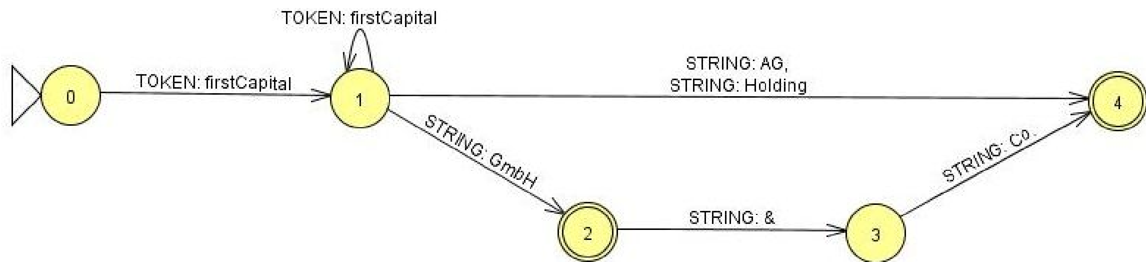


Figure 2.3: Automaton representing the pattern for recognition of company names [13]

Information extraction (IE) systems today are commonly based on pattern matching. The patterns are regular expressions stored in a customizable knowledge base. Adapting an IE system to a new subject domain constructs of a new pattern base, which is time-consuming and expensive task. R. Yangarber et al. describe a strategy for building patterns from examples [25]. To adapt the IE system to a new domain quickly, the researcher chooses a set of examples in a training text, and for each example gives their logical form. The system applies Meta rules to transform the example automatically into a general set of patterns.

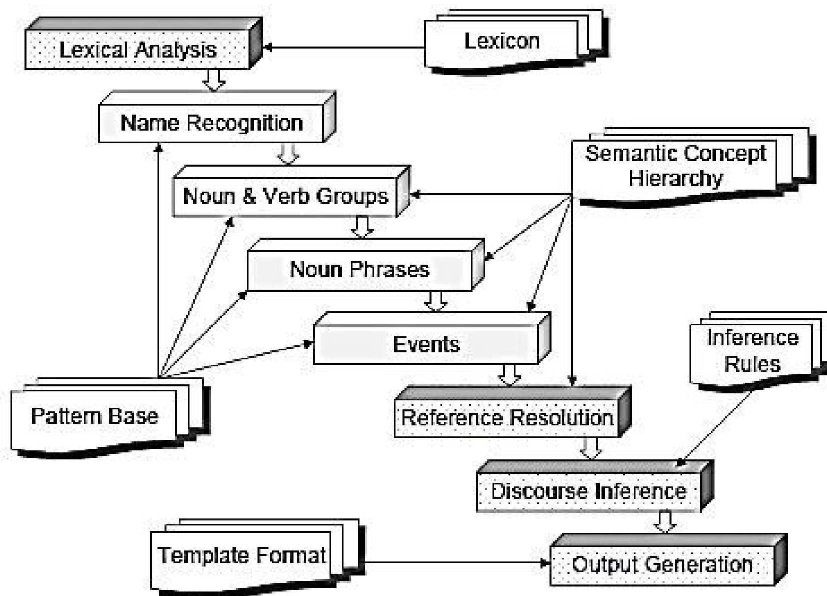


Figure 2.4: IE system architecture [25]

Unstructured data, such as free-text police narrative reports, often are stored as text objects in databases or simply text files. Valuable information in such texts is difficult to be accessed or used by crime investigators in further analyses. It would be useful to identify from such text reports meaningful entities, such as person names, narcotic drugs, or personal properties.

In 2002, M. Chau et al. [6] suggest a technique which is feasible and has some potential values for real-life applications. The neural network-based entity extractor applies named-entity extraction techniques, useful for identifying entities from police narrative reports. The system achieved encouraging precision and recall rates for person names and narcotic drugs.

$$\text{Precision} = \frac{\text{number of correct entities identified by the system}}{\text{number of all entities identified by the system}}$$

$$\text{Recall} = \frac{\text{number of correct entities identified by the system}}{\text{number of all entities identified by human}}$$

Figure 2.5: The performance measure of system [6]

Proper names have been widely studied in the field of Information Extraction. They can also play a role in systems of Information Retrieval. Names are very special words: they represent about 10% of a text in a newspaper article. The quantity of proper names and their informative quality in this type of texts make them relevant to improve the clustering. In 2002, N. Friburger et al. describe a technique in which the similarity of the vector of words and the similarity of the vector of proper names are weighed with two coefficients to increase or decrease their importance [22]. The best clustering results are obtained for a weighing scheme promoting proper names and using a similarity measure on all the words.

Internet search is an extremely important application; email is the number one online activity for most users. Despite this, there are surprisingly few advanced email technologies that take advantage of the large amount of information present in a user's inbox.

Table 2.1: Results of entropy on a clustering with proper names [22].

OFIL1-trails	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Number of classes (clusters)	40	29	38	34	39	38	28
Entropy with sim $_{allwords_TFIDF}$	0.019	0.071	0.025	0.057	0.089	0.087	0.022
Entropy with sim $_{names_TFIDF}$	0.063	0.046	0.087	0.102	0.110	0.293	0.037
Entropies sim $_{allwords_Jaccard}$	0.078	0.033	0.042	0.107	0.142	0.259	0.068

Aron Culotta et al. [1] describes a powerful, statistics and learning-based information extraction system for mining both email messages and the web to automatically extract a user's social network, and obtain expertise and contact information for each person in the network [1]. After extracting people names from email messages, system works to find each person's web presence, and then extract contact information from these pages using conditional random fields (CRFs), a probabilistic model that has performed well on similar language processing tasks.

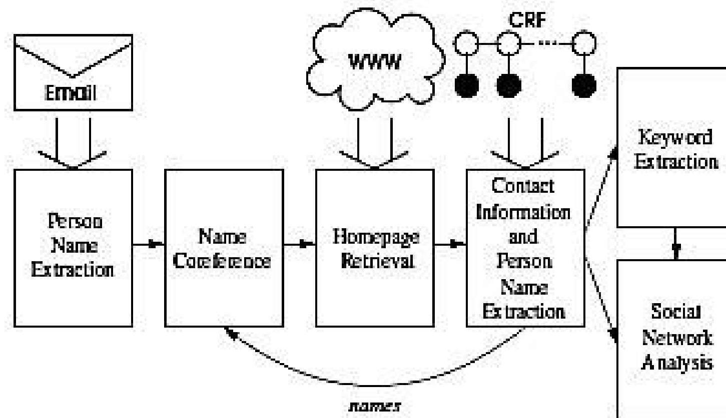


Figure 2.6: System Overview [1]

There are no exhaustive place name dictionaries for Indian place names in Indian Languages as well as in English. In 2007, there is a technique for multilingual location tagging for Indian languages. The system has a language identifier and multilingual place name tagger. The language identifier built using statistical techniques identifies the language of each sentence [29]. The location tagger is built completely based on suffix stripping techniques, which has been more explored for Part-Of-Speech tagging task. A

finite state automaton, built using the location suffixes, is used in tagger to bring a time and space efficiency. There is no need for the preprocessing of the input text. Heuristic rules are used to enhance the performance of the system. The system takes a multilingual text as an input, identifies the languages at sentence level and tags the place names.

Knowledge-based natural language processing systems have achieved good success with certain tasks but they are often criticized because they depend on a domain-specific dictionary that requires a great deal of manual knowledge engineering. This knowledge engineering bottleneck makes knowledge-based NLP systems impractical for real-world applications because they cannot be easily scaled up or ported to new domains.

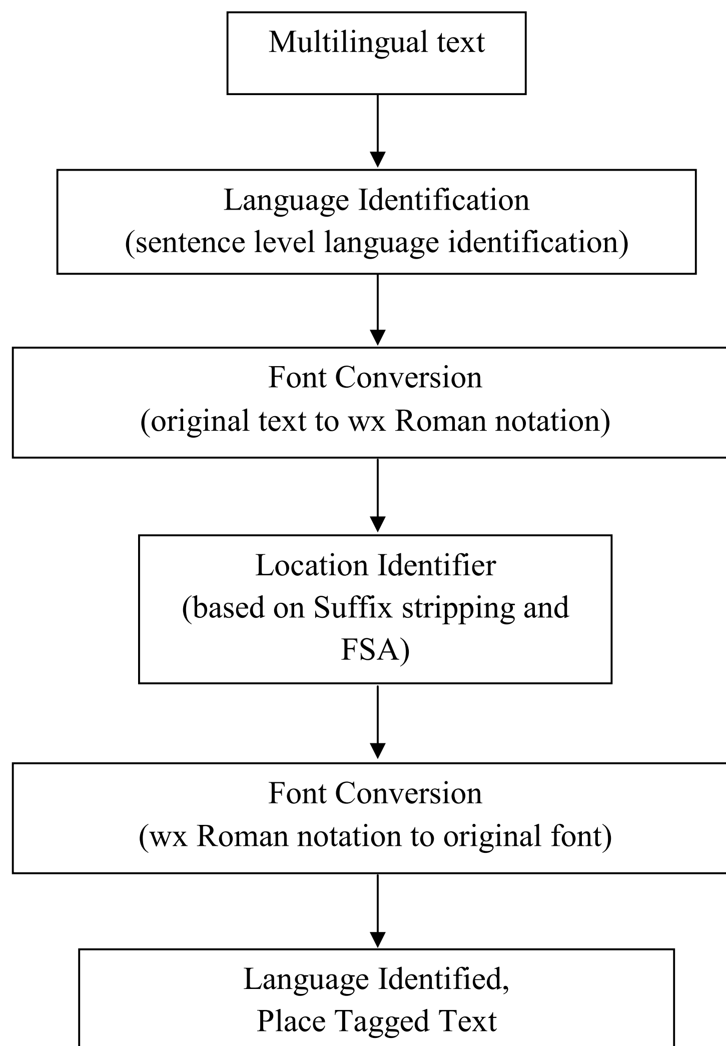


Figure 2.7: Architecture for place tagged text [26]

In response to this problem, a system called AutoSlog was develpoed that automatically builds a domain-specific dictionary of concepts for extracting information from text [19]. Using AutoSlog, a dictionary for the domain of terrorist event descriptions is constructed in only 5 person-hours.

In natural language relationships between entities can asserted within a single sentence or over many sentences in a document. Many information extraction systems are constrained to extracting binary relations that are asserted within a single sentence (single-sentence relations) and this limits the proportion of relations they can extract since those expressed across multiple sentences (inter-sentential relations) are not considered. The analysis in *Inter-sentential Relations in Information Extraction Corpora* paper focuses on finding the distribution of inter-sentential and single-sentence relations in two corpora used for the evaluation of information extraction systems: the MUC6 corpus and the ACE corpus from 2003 [15]. Result found that 28.5% of MUC6 relations and 9.4% of ACE03 relations are intersentential. This quantification of inter-sentential relations in the MUC6 corpus (28.5%) improves on previous investigation (Stevenson, 2007) that calculated an upper bound of 40%.

In 2008, Named Entity Recognition (NER) system is developed for South and South East Asian languages, particularly for Bengali, Hindi, Telugu, Oriya and Urdu as part of the IJCNLP-08 NER Shared Task1 [2]. Researchers have used the statistical Conditional Random Fields (CRFs). The system makes use of the different contextual information of the words along with the variety of features that are helpful in predicting the various named entity classes. The system uses both the language independent as well as language dependent features. The language independent features are applicable for all the languages. The language dependent features have been used for Bengali and Hindi only. The performance of the system for Bengali can further be improved by including the part of speech (POS) information of the current and/or the surrounding word(s).

Document indexation is an essential task achieved by archivists or automatic indexing tools. To retrieve relevant documents to a query, keywords describing this document have to be carefully chosen. Archivists have to find out the right topic of a document before starting to extract the keywords. For an archivist indexing specialized documents,

experience plays an important role. But indexing documents on different topics is much harder. In 2008, C. A. Chahine et al. proposes an innovative method for an indexing support system [5]. This system takes as input ontology and a plain text document and provides as output contextualized keywords of the document. The method has been evaluated by exploiting Wikipedia's category links as termino-ontological resources. In order to get more precise results, it needs to be mapped to a controlled vocabulary used by archivists like the Dewey Decimal Classification (DDC) or proprietary controlled vocabulary (like the UNIT classification).

The local grammar approach was first used to discuss recursive phrases that are commonly found in specialist literature like biochemistry and then extended to extract time, date and address expressions from letters. It has recently been applied to extract person names from English, Chinese, French, Korean, Portuguese, and Turkish news texts. In 2009, Hayssam Traboulsi shows that this approach can also be used to extract person names from Arabic counterparts [10]. This paper discusses the use of corpus linguistics methods and techniques, in conjunction with the so-called local grammar formalism, to identify patterns of person names in Arabic news texts. The method shown in the paper could save us a lot of time and effort needed if all the expectations related to the universal grammars were considered.

With the growing number of viruses and network attacks, Intrusion Detection Systems (IDS) have to match a large set of regular expressions at multi-gigabit speed to detect malicious activities on the network. Many algorithms and architectures have been designed to accelerate pattern matching, but most of them can be used only for strings or a small set of regular expressions. Jan Korenek, propose new NFA–Split architecture, which reduces the amount of consumed FPGA resources in order to match larger set of regular expressions at multi-gigabit speed [12]. The proposed reduction uses model of nondeterministic and deterministic automaton for effective mapping of regular expressions to FPGA. A new algorithm is designed to split the nondeterministic automaton transition table in order to map a part of the table into memory.

Many network security applications in today's networks are based on deep packet inspection, checking not only the header portion but also the payload portion of a packet.

For example, traffic monitoring, layer-7 filtering, and network intrusion detection all require an accurate analysis of packet content in search for predefined patterns to identify specific classes of applications, virus, attach signatures, etc. Pattern matching is a major task in deep packet inspection.

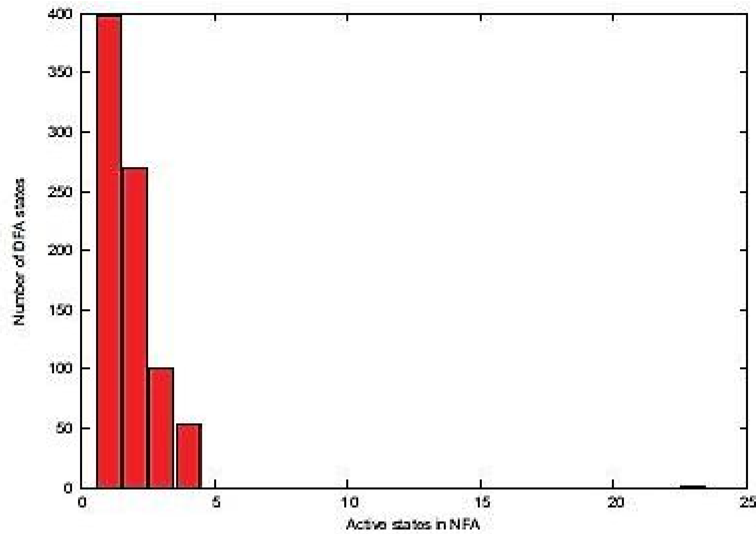


Figure 2.8: The number of DFA states splitted according to the size of DFA state set [12]

The two most common implementations of Pattern matching are based on nondeterministic finite automata and deterministic finite automata, which take the payload of a packet as an input string. Yan Sun et al. propose an efficient NFA based pattern matching in Binary Content Addressable Memory (BCAM), [31] which uses data search words consisting of 1s and 0s. The approach use can process multiple characters at a time using limited BCAM entries, which makes our approach scalable well.

2.3 Use of Lex tool for Extracting Information

To reduce tedious manual efforts in generating high-covering test inputs, various automated techniques have been proposed. Some recent effective techniques include Dynamic Symbolic Execution (DSE) based on path exploration. However, these existing DSE techniques cannot generate high-covering test inputs for programs using complex regular expressions due to large exploration space; these complex regular expressions are commonly used for input validation and information extraction. To address this issue, in 2009 researchers propose an approach, named Reggae, to reduce the exploration space of

DSE in test generation [17]. Generic functions are transformed which are invoked by a program under test to specialized functions that can help a test-generation tool to generate high-covering test inputs. Reggae is used for testing .NET programs, including C# programs.

Regular expressions are used in a large variety of applications to express validity constraints on strings. Constraints in form regular expressions over strings are ubiquitous. They occur often in programming languages like Perl and C#, in SQL in form of LIKE expressions, and in web applications. Providing support for regular expression constraints in program analysis and testing has several useful applications. Margus Veanes et al. [21] introduce a method and a tool called Rex, for symbolically expressing and analyzing regular expression constraints. Rex is implemented using the SMT solver Z3.
For many years, it was commonplace for teachers to turn to linguistic theory for grammars or syntax of what to teach in their language classes.

While grammatical rules cannot be ignored, just teaching the underlying system of a language is no guarantee that the language learners will utilize these rules in comprehending and producing language successfully in appropriate contexts. Increased opportunities for communication create a demand for a more efficient way to improve the learners' listening and speaking abilities.

The language learners communicate successfully and learn the language rule by using the lexical chunks. The studies in the corpus linguistics also found that lexical chunks of different forms form about 80 percent of natural langue [8]. Based on the research of lexical chunks, many linguists put forward the lexical approach of langue teaching, which values the importance of language chunks in natural langue and aim to improve the learner's ability of actual language use. It is a way of teaching according with the language natural acquisition and cognition, which indicate a new field of vision for teaching and learning in China.

Lex tool helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-scripts type transformations and for segmenting input in preparation for a parsing routine. Lex tool source is the table of regular expressions and corresponding program fragments. The table is translated to a

program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions.

A Calculator Compiler has been developed by M. Upadhaya [20] using Lex (Lexical Analyzer Generator) and YACC (Yet Another Compiler-Compiler). He shows that with the Lex and Yacc tool one can create its own compiler, wherever required. It is basically procedural language compiler tools and to support object oriented one need to work on structure of C language to support object oriented which makes the compiler quite complex.

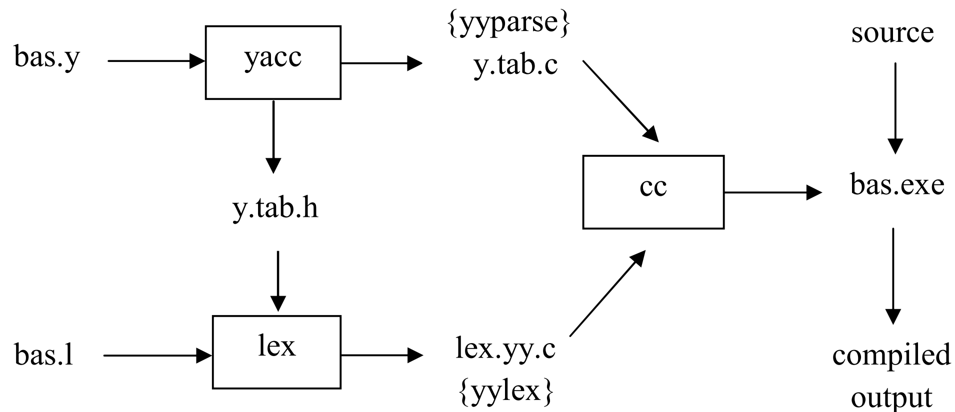


Figure 2.9: Building Compiler with Lex and Yacc [20]

There are two ways to handle the keywords and identifiers [3]:

1. Keywords are installed in the symbol table. The field of symbol-table entry gives the strings that represent the token. The function *installID( )* places the identifier in the symbol table. If identifier is present in symbol-table, a pointer is returns for the lexeme found. The function *getToken( )* examine the symbol-table for lexeme found and returns token name.
2. Transition diagram is created for each keyword. Transition diagram consists of states for each successive letter of keyword. It is necessary to check whether the identifier has ended, or return token in situations.

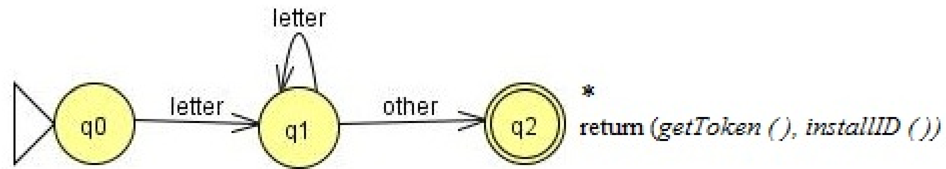


Figure 2.10: Transition diagram for *letter* [3]

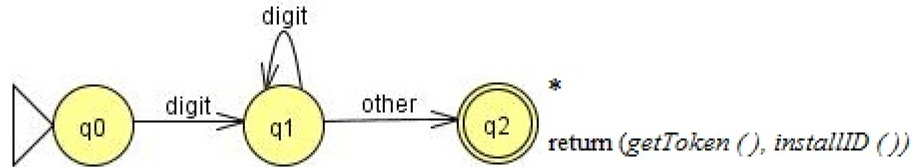


Figure 2.11: Transition diagram for *digits* [3]

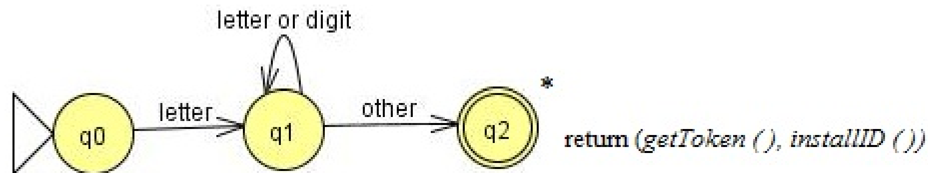


Figure 2.12: Transition diagram for *keywords* [3]

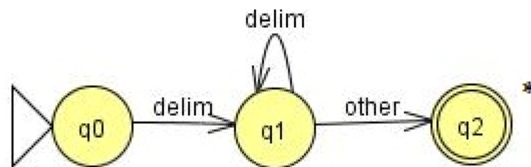


Figure 2.13: Transition diagram for *whitespace* [3]

The algorithm for Token recognition is discussed in figure 2.12.

In the declarations section of the algorithm given in the figure 4.10, there is a pair of special brackets, %{ and %}. Anything within theses brackets is copied directly to the file *lex.yy.c*. The definitions of manifest constants are placed with C# define statements in the definition section. ID, NUMBER is the manifest constants of the given algorithm.

In translation rules section regular definitions are surrounded by curly braces. The regular expression *{delim}+* is defined for *ws*, which indicates whitespace. If whitespace is found, we do not return to the parser and look for another lexeme.

In the auxiliary-function section, two functions, *installID( )* and *installNum( )* are used. Function *installID( )* is called to place the lexeme in the symbol table and returns a pointer to the symbol table, which is placed in global variable *yylval* for the use of parser. The two variables *yytext* and *yyleng* are set automatically by the lexical analyzer.

```
%{
        /* definitions of manifest constants
            ID, NUMBER */
%}
/* regular definitions */
delim          [\t \n]
ws             {delim}+
letter         [A-Za-z]
digit          [0-9]
id             {letter}({letter} | {digit})*
number         {digit}{digit}*
%%
{ws}           {/* no action and no return */}
{id}           {yylval=(int) installID( ); return(ID);}
{number}       { yylval=(int) installID( ); return(NUMBER);}
%%
int installID( )       {/*   function to install the lexeme, whose first character is pointed
                          to by yytext and whose length is yyleng, into the symbol table and
                          return a pointer there  */
}
int installNUM( )      {/*  similar to installID, but puts numerical constants into a
                          separate  table  */
}
```

Figure 2.14: Lex program for the tokens recognition [3].

The problem of finding the person name, place and date from text document by using NFA is based on construction of regular expressions. Different techniques or approaches are available for person name identification. But our purpose is to find out technique which is efficient and convert NFA to corresponding deterministic FSA with minimal number of states. So we have to carry out comparison between them, and design some new algorithm.

Chapter 3
Problem Statement

This chapter includes the problem statement followed by the motivation behind the thesis.

3.1 Problem Statement

Even though people may be able to quickly find unstructured sources, it still takes time to extract relevant information. Inspecting the document for collateral information requires reading the entire document. People may read many paragraphs containing no direct information on dates or names. It is also possible, that by reading such a lengthy document our mind may wander and may miss important information. The algorithm design for identifying and extracting person name, place and date from a document solves a number of problems that people face in genealogy. First, people generally do not want to read all the text that is given to them. If they do read it, they tend to skip over the text and look for the important parts. Though the human mind is an amazing tool, we will make mistakes and are prone to missing vital things.

FSA is a convenient tool to represent the linguistic phenomena [27]. The language of an FSA is equivalent to regular expressions. The recognition of an input symbol sequence can be solved in a linear time by an FSA. The efficient performance of an FSA is made possible by the following reason: there is a method to transform each automaton into a uniquely equivalent automaton with the least number of states.

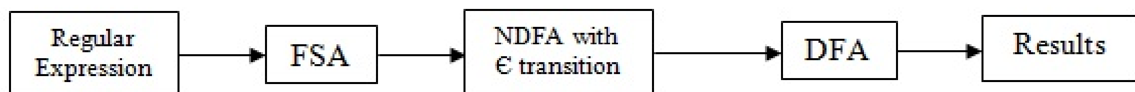


Figure 3.1: Regular expression to minimum DFA [28].

Any regular expression recognized by an NFA can be recognized by an equivalent minimal DFA. Such representations have successful applications in various language processing. While an FSA has powerful representations for regular languages, it is well

known that the linguistic expressiveness of the FSA fails to fully recognize natural languages such as English and Chinese. The problem of person name identification from documents by FSA is difficult and challenging, and hence is still an open problem.

3.2 Thesis Motivation

The finite state automata (FSA) theory is an essential tool used in many areas in computer science, including pattern matching, database, and compiler technology. The finite state system has discrete inputs and outputs and finite number of internal configurations called "states". At any given point of time the system is always in one of its finite states.

Due to the speed and the compactness of FSA representations, it provides efficient and convenient tools to represent the linguistic phenomena. It takes an input sequence of symbols such as alphabetical letters, consumes one symbol at a time, and transits one state to another state until it reaches the end of the input or halts on a state. The whole set of the recognizer input symbols accepted by an FSA is called the language of the FSA. The language of an FSA is equivalent to regular expressions.

Nevertheless, recent research shows that an FSA may be used to achieve some "simple" aspects of natural language processing such as information extraction. Motivated by this approach we propose a technique to extract information by using FSA and LEX.

Chapter 4

Extracting Person Name, Date and Place from Text Document using LEX Tool

This chapter presents the analysis and comparison of existing approaches for information identification and describes approach in this thesis work for extracting information using Lex tool. This chapter presents the evaluation strategy that is followed and the experimental results that are obtained.

4.1 Analysis of Existing Approaches

The existing approaches to person name identification in the documents in the literature fall into the following three categories as follows:

(i) Statistical methods [4]: These methods use a large corpus of tagged documents for statistical training, and the identification problem is reduced to a standard classification and labeling problem after the training is done.

(ii) Guessing-from-surnames methods [4]: These are the most commonly used methods in person name identification in documents. There is a set of commonly used surnames in a text file or document. Consequently, when encounter any of these words; one can presume that they might signify an appearance of a person name.

The advantage of these methods is that they are simple and robust as long as the encountered words are truly surnames. The disadvantages, however, are that there are false positives as it is not always true that these words are surnames in the text.

(iii) Context-based methods [4]: Methods in this category attempt to take advantage of the contextual information typically existing in any language based document to "infer" the occurrence of a person name. Examples of the contextual information include the title of a person, certain verbs, and certain adjectives.

These are the three methods used for searching a text document.

4.2 Proposed Approach

Based on the above discussion of previous work in person name identification in text documents, we propose a new method or approach using FSA for person name identification in documents.

The algorithm for date, name, and place extraction consists of four steps.

1. **Convert the content to plain text:** If source could be any programming language, it will produce a plain text document.
2. **Convert the text from a sequence of characters to a sequence of tokens:** It converts a sequence of characters into a sequence of tokens with appropriate token types. It does this in two phases. First it uses a finite state machine to separate the sequence of characters into a sequence of tokens. This is done quickly. In the second phase we categorize every token as to whether it is a name, place, date or none-of-the-above.
3. **Identify the dates, name, and places:** It uses a grammar to parse the tokens and extract complete names, dates and places. Extracting places is similar to extracting names.
4. **Format the results**.

Thompson Construction followed by Subset Construction is used to convert a regular expression into non-deterministic finite automata.

4.2.1 Thompson Construction (RE, NFA) [3]

Input: A regular expression *r* over alphabet ∑.

Output: An NFA *N*, accepting L(*r*).

Method: Begin by parsing *r* into its constituent sub expressions. The rules for constructing an NFA consist of basis rules for handling sub expressions with no operators, and inductive rules for constructing larger NFA's from the NFA's for the immediate sub expressions of a given expression.

1. **BASIC RULE:** This rule is for constructing an NFA for no operator sub expressions.

Basic regular expression: NFA's for regular expression a, ε, or φ:

For expression ε constructs the NFA

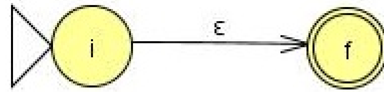


Figure 4.1: NFA for ε [3]

Here, *i* is a new state, the start state of this NFA, and *f* is another new state, the accepting state for the NFA.

For any sub expression a in ∑, construct the NFA

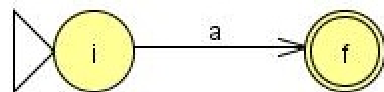


Figure 4.2: NFA for a [3]

where *i* and *f* represents the starting and final states, respectively. In both of the basic constructions, we construct a distinct NFA, with new states, for every occurrence of ε or some *a* as sub expression of *r*.

2. **INDUCTION RULE:** This rule is for constructing an NFA for one or more operator sub expressions.

Suppose *N(s)* and *N(t)* are NFA's for regular expressions *s* and *t*, respectively.

2.1 Alternative Operation: To construct an NFA equal to *s* | *t*. We add a new starting state and a new accepting state and connected them as shown in figure 4.3 using ε-transitions.

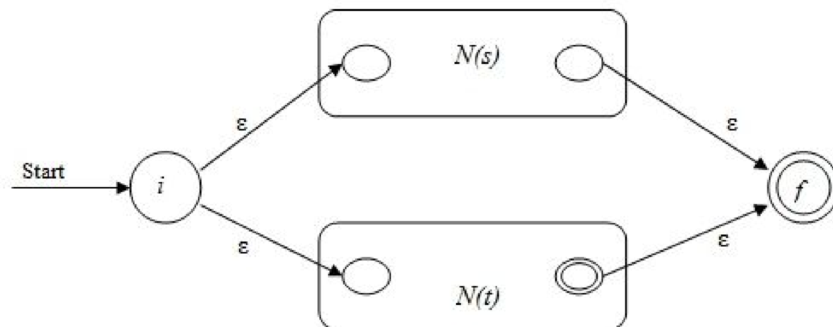


Figure 4.3: NFA for the union of two RE [3]

2.2 Concatenation: To construct an NFA equal to *st.* The accepting state of the machine *s* is connected to the start state of the machine *t* by a ε-transition. The starting state of the machine *s* becomes starting state of Concatenation and final state of the machine *t* becomes final state of NFA of Concatenation.

The figure 4.4, give the result of concatenation of two regular expressions.

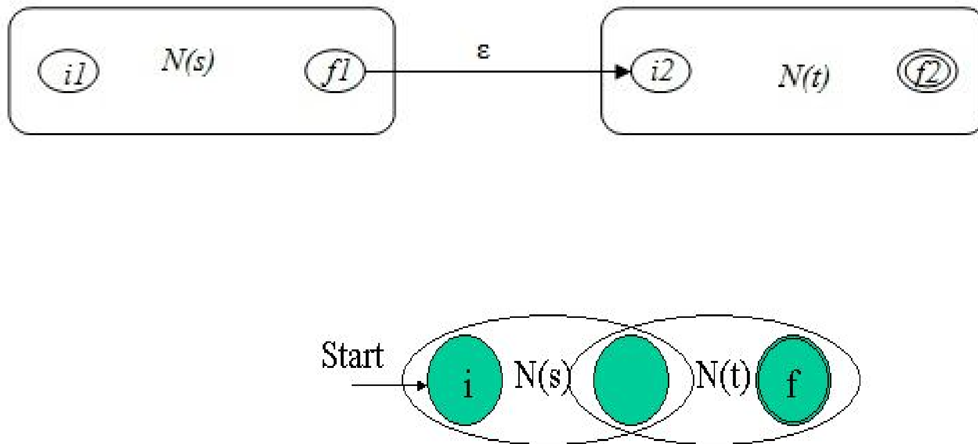


Figure 4.4: NFA for the concatenation of two RE [3]

2.3 Repetition: To construct an NFA equal to $r = s*$ following steps are followed:

1. To add two new states, a start state and an accepting state.
2. The repetition is afforded by the new ε-transition from the accepting state of the machine *r* to its start state.
3. To draw a ε-transition from the new start state to the new accepting state.

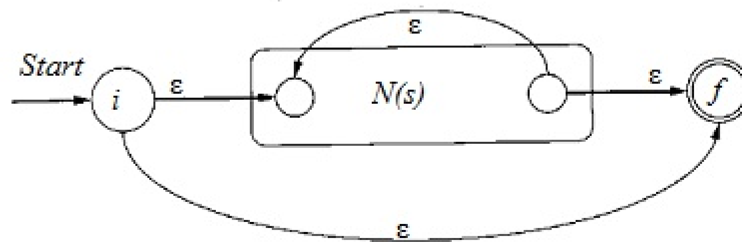


Figure 4.5: NFA for the closure of a regular expression [3]

To get from *i* to *f*, we can go to start state of *N(s),* through that NFA, then from its accepting state back to its start state zero or more times.

4.2.2 Subset Construction (NFA, DFA) [3]

DFA is constructed for a given NFA by using subset construction algorithm.

Input: An NFA *N*

Output: A DFA *D,* accepting the same languages as *N*.

Method: The rules for constructing a DFA consist of rules for eliminating ε-transitions and eliminating multiple transitions. It uses the ε-closure ( ) and move ( ) functions to build a complete DFA transition matrix from a previously constructed NFA.

Table 4.1: Operations on NFA states [3]

OPERATION	DESCRIPTION
ε-closure (*s*)	Set of NFA states reachable from NFA state *s* on ε-transitions alone.
ε-closure (*T*)	Set of NFA states reachable from any NFA *s* in set *T* on ε-transitions alone.
Move (*T, a*)	Set of NFA states to which there is a transition on input symbol *a* from some state *s* in *T*.

As shown in the table 4.1, three operations are used in subset construction algorithm. First problem is to deal with ε-transitions of *N* properly. The number of states in a DFA is equal to 2^Q states in worst case, where Q is number of states in DFA.

4.2.3 Information Extraction using FSA

Following steps are used for extracting information:

1. Segment and tag a sentence.
2. Then segregate the sentence into small regions.
3. These regions and their associated tag sequences are fed into an FSA. The FSA consumes one tag at a time, makes necessary state transitions, or stops when it reaches the final state. After all tags have been consumed, an input sequence that leads to the final state is accepted, which indicates that we have found a person name contained in the current region. If an input sequence that halts on a state is rejected, this indicates that the current region contains no person name.

4. The *filename.l* is given as input to lex tool which generates *lex.yy.c.* The *lex.yy.c* contains the specifications of finding tokens using DFA in C language.

We first make the regular expressions for the given plain text and then extract the person name and place from the text file.

Regular expression used in Lex tool for extracting name, date and place from the file as follow:

1. ([0-2] [0-9] | [3] [0-1]) / \ ((0 (1|3|5|7|8)) | (10|12)) / \ ([1-2] [0-9] [0-9] [0-9]) :

This regular expression is for the extraction of a month contains 31 days. It will check for the date first, when the look ahead '/' symbol comes it start reading the next character. Next sub-expression is for checking month. The sub-expression ((0(1|3|5|7|8)) | (10|12) is for the months having 31 days, like 01(January) or 03(March), so on. The next sub-expression is for checking the year.

2. ([0-2] [0-9] | 30) / \ ((0(4|6|9)) | 11) / \ ([1-2] [0-9] [0-9] [0-9]):

This regular expression is for the extracting the information of the months which have 30 days. The first sub-expression is for checking the dates. The day contain number only upto 30. After the look ahead '/' symbol, next character will be read. The next sub-expression ((0(4|6|8)) | 11) is for months contains 30 days i.e, 04(April) or 06(June), so on. The next character read is for checking the year. The year will be in between [1000-2999].

3. ([0-1] [0-9] | 2[0-8]) / \ 02 / \ ([1-2] [0-9] [0-9] [0-9]):

The above two regular expressions are for the month of 31 and 30 days. The next regular expression is for the month of February, having 28 or 29 days. This regular expression is for the 28 days month. The date will be anything among [01-19] and [20-28]. The last sub-expression is for the year.

4. 29 / \ 02 / \ ([1-2] [0-9] [0-9] [0-9]):

This regular expression is only for leap year. The only month having 29 days is February. It first checks for the date, then month and at last year.

5. (yr % 4 = = 0 || (yr % 100 = = 0 && yr % 400 != 0)):

This regular expression will check whether the date extracted in above regular expression is a leap year or not.

6. [keyword] {letter (letter | digit) *}

This regular is used for extracting person name or place. The name will be a letter followed by another letter or digit.

For example, the expression [keyword] {main | int | scanf | printf | if | else} will check for the keywords main, int, scanf and so on.

These regular expressions give us two things:

1. Where the person name, place and date locates in an article.
2. Whether the date written in the article is valid or not.

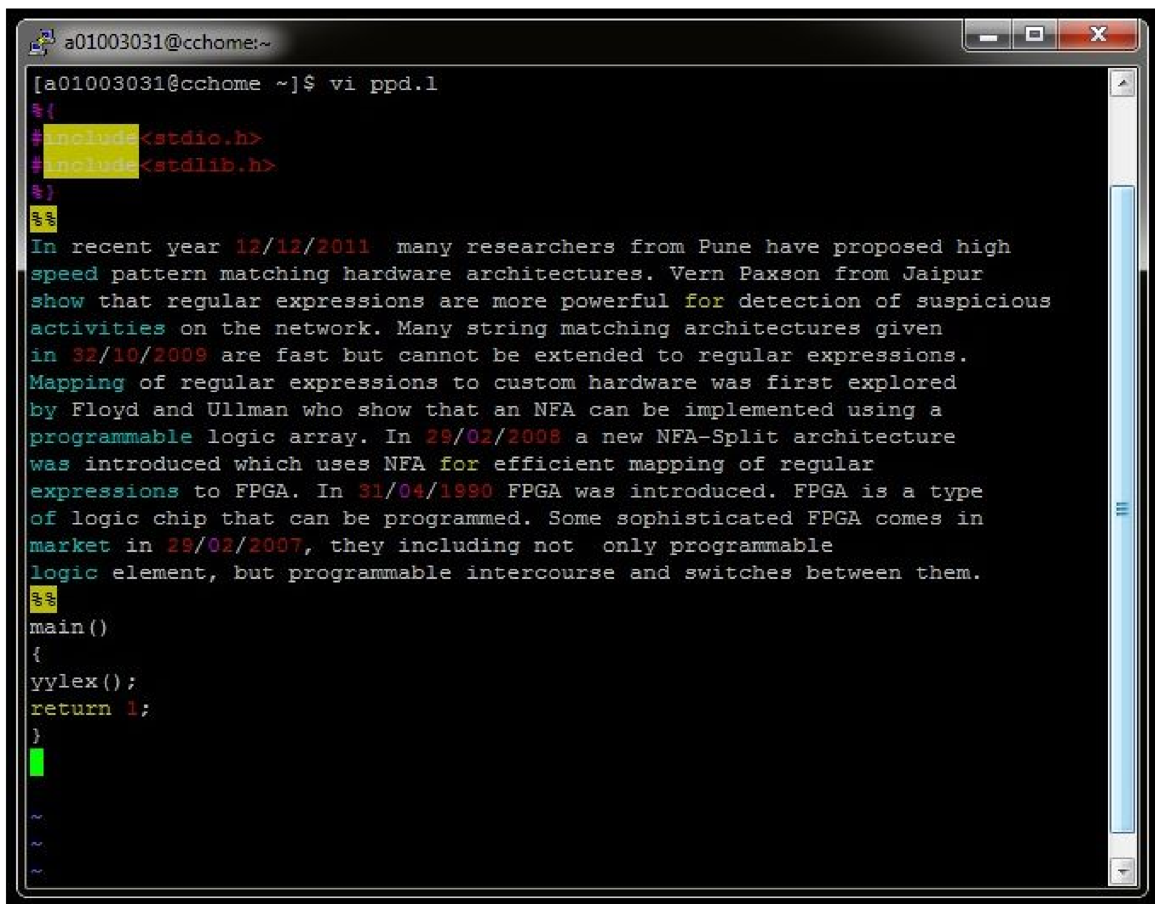


Figure 4.6: File “ppd.l”

```
a01003031@cchome:~
#include<stdio.h>
#include<stdlib.h>
int i=0,j,yr=0,valid=0;
%}
[keyword] {Vern paron|klen|pune|Floyd ullman|jaipur;}

([0-2][0-9]|[3][0-1])/((0(1|3|5|7|8))|(10|12))/([1-2][0-9][0-9][0-9])
{valid=1;}

([0-2][0-9]|30)/((0(4|6|9))11)/([1-2][0-9][0-9])
{valid=1;}

([0-1][0-9]|2[0-8])/02/([1-2][0-9][0-9][0-9])
{valod=1;}

29/02/([1-2][0-9][0-9][0-9])
{
while (yytext[i] <> '/')
i++;
while (i<yyleng)
yr=(10*yr)+(yytext[i++]-'0');
if (yr%4 == 0 ||(yr%100 == 0 &&yr%400 <> 0))
valid=1;}
%%
main()
{
yyin=fopen("ppd.l","r");
yylex();
if (valid==1)
printf("It is a valid date\n");
else
printf("It is not a valid date\n");
}
int yywrap()
{
return 1;
}
-- INSERT --                                        38,2          Bot
```

Figure 4.7: Program “thesis.l”

In file “ppd.l” we write an article contains some person name, date and place name.

The two figures given below i.e, figure 4.8 and figure 4.9 are not the exact output. These figures just show how the output window looks after the implementation of the regular expressions for extracting information given in this thesis. In figure 4.8, shows the locations along with their information about the person name, place and date. The Figure 4.9, show us whether the date given in a document is valid or not.

Names

Wiggins			Loc1, Loc2
	David		Loc1
Woodfield			
	Scott		Loc1
	Scott N.		Loc1, Loc2
Zimmerman			
	Paul		Loc1, Loc2, Loc3

Dates

1871			Loc1, Loc2
1873			
	February		
		5	Loc1, Loc2
		12	Loc1, Loc2, Loc3
	April		Loc1, Loc2
1877			Loc1, Loc2, Loc3

Places

Indiana			
		Lafayette	Loc1, Loc2
Utah			Loc1
	Weber		Loc1, Loc2
		Ogden	Loc1
	Utah		
		Provo	Loc1, Loc2, Loc3

Figure 4.8: Result (i)

OUTPUT

Content in input file (here, ppd.1)	Output
12/12/2011	It is a valid date
32/10/2009	It is not a valid date
29/02/2008	It is a valid date
31/04/1990	It is not a valid date
29/02/2007	It is not a valid date

Figure 4.9: Result (ii)

Chapter 5

Conclusion and Future Scope

Searching a text document manually for important information from a file is slow, tedious and error prone. The proposed work provides an insight into the various approaches used for extracting person name, place and date from a document. Regular expressions are being used for most of the popular web sites and only 4% of the regular expressions are unique. LEX tool is used for generating lexical analyzer.

In this thesis, regular expressions are created in LEX tool that are used for extracting information person name, place and date from a document. Lex tool generate a C program which contains the specification specified by the lex program.

Following are the future directions in which work can be carried out:

1. Use of regular expression for identification of information using lex tool in other natural languages.
2. Lex output can be integrated with YACC for checking whether the grammar used in the language is correct or not.

References

[1] A. Culotta and R. Bekkerman: "*Extracting Social Networks and Content Information from Email and the Web",* In Proceeding of the Conference on Email and Anti-Spam, july 2004.

[2] A. Ekbal, R. Haque, A. Das, V. Poka and S. Bandyopadhyay: *"Language Independent Named Entity Recognition in Indian Languages*", In Proceeding of the IJCNLP Workshop on Names Entity Recognition for South and South East Asian Languages, 2008.

[3] A.V. Aho, M.S. Lam, R. Sethi and J.D. Ullman: "Lexical Analysis in *Compilers Principles, Techniques and Tools*", Second Edition New Delhi, pp.109-155, 2008.

[4] B.S. Zhang and C. Yuan: *"Person name identification in Chinese documents using Finite State Automata*", In proceedings of the IEEE International Conference on Intelligent Agent Technology, pp. 478-482, 2003

[5] C.A. Chahine and N. Chaignand: "*Context and Keyword Extraction in the Plain Text using a Graph Representation",* In Proceedings of the IEEE International Conference on Signal Image Technology and Internet Based Systems, pp. 692-696, 2008.

[6] C. Michael and J.X. Jennifer: "*Extracting meaningful Entities from Police Narrative Reports",* In Proceedings of the annual National Conference on Digital Government Research, pp. 1-5, 2002.

[7] *"Computer Science / Information Technology",* Publishers G.K, 2009.

[8] D. Guoxiang and J. Linlin: *"The Lexical Approach for Language Teaching Based on the Corpus Language Analysis",* In Proceeding Third IEEE International Conference on Communication Software and Networks, pp. 665-668, 2011.

[9] H. Larkin: "*Object Oriented Regular Expressions*", In Proceedings of the 8th International Conference on Digital Object Identifier, pp. 491-496, 2008.

[10] H. Traboulsi: *"Arabic Named Entity Extraction: A Local Grammar-Based Approach",* In Proceeding of the IEEE International Multiconference on Computer Science and Information Technology, pp. 139-143, 2009.

[11] J.C. Martin: *"Introduction to Languages and Theory of Computation",* Fourth Edition, 2009.

[12] J. Korenek and V. Kosar: "*Efficient Mapping of Nondeterministic Automata to FPGA for Fast Regular Expression Matching*", In Proceedings of the 13th IEEE International Symposium on Digital Object Identifier, pp. 54-59, 2010.

[13] J. Piskorski and G. Neumann: "*An Intelligent Text Extraction and Navigation System",* In Proceedings of the 06th International Conference on Computer-Assisted Information Retrived, pp. 1015-1032, 1999.

[14] K.L.P Mishra and N. Chandrasekaran: *"Theory of Computer Science (Automata Languages and Computation)",* Second Edition, 1998.

[15] K. Swampillai and M. Stevenson: "*Inter-sentential Relations in Information Extraction Corpora*", In Proceedings of the 23th International Symposium, pp. 1-6, 2008.

[16] L.A. Charles and V.C. Gorden: "*On the Use of Regular Expressions for Searching Text",* In Proceedings of the ACM Transactions on Programming Languages and System, pp. 413-426, 1997.

[17] L. Nuo, T. Xie and T. Nikolai: *"Reggae: Automated Test Generation for Programs using Complex Regular Expressions",* In Proceeding 24th IEEE ACM International Conference on Automated Software Engineering, pp. 515-519, 2009.

[18] L. Peter: "*An Introduction to Formal Languages and Automata",* Jones and Bartlett Publishers, Third Edition, 2001.

[19] M. Mohri: "*On Some Applications of Finite-State-Automata Theory to Natural Language Processing",* In Journal of Natural Language Engineering, pp. 61-80, March 1996.

[20] M. Upadhyaya: "*Simple Calculator Compiler Using Lex and YACC*", In Proceeding of the International Conference on Electronics and Computer Technology, pp. 182-187, 2011.

[21] M. Veanes and P.D. Halleux: *"Rex: Symbolic Regular Expression Explorer",* In Proceeding International Conference on Software Testing, Verification and Validation, pp. 498-507, 2010.

[22] N. Friburger and D. Maurel: *"Textual Similarity Based on Proper Name",* In Proceeding 25th ACM Conference on Mathematical/Formal Methods in Information Retrieval, pp. 155-167, 2002.

[23] R. Ellen: "*On the Power of Quantum Finite State Automata",* In Proceedings of the Eleventh National Conference on Artificial Intelligence, pp. 811-816, 1993.

[24] R. Hodovan, Z. Herczeg and A. Kiss: "*Regular Expression on the Web*", In Proceedings of the 12th IEEE International Symposium on Digital Object Identifier, pp. 29-32, 2010.

[25] R. Yangarber and R. Grishman: "*Transformation Examples into Patterns for Information Extraction",* In Proceedings of a workshop on Computational Linguistic, pp. 97-103, 2000.

[26] S. Hussain: *"Finite-State Morphological Analyzer for Urdu",* at National University Computer Science of Computers and Emerging Sciences, 2004.

[27] S.S. Sane: "*Theory of Computer Science*", Second Edition, Technical Publications, Pune, 2002.

[28] Ullman, J.J.E. Hopcraft and R. Motwani: "*Introduction to Automata Theory, Languages and Computation",* Person Education Inc, ISBN 0-201-44124-1, Addison Wesley, 2001.

[29] V. S. Ram and L. Sobha: "*Multilingual Place Name Tagger for Indian Languages*", In Proceedings of the 05th International Conference on Digital Object Identifier, pp. 58-65, 2007.

[30] Y.S. Han and D. Wood: "*Obtaining Shorter Regular Expressions from Finite-State Automata",* In Proceeding of the Conference on Theoretical Computer Science, pp. 110-120, 2006.

[31] Y. Sun and V. C. Valgenti and M.S. Kim: "*NFA-based Pattern Matching for Deep Packet Inspection*", In Proceedings of the 20th International Conference on Digital Object Identifier, pp. 1-6, 2011.

List of publications

Published

[1] R. Sharma and S. Garhwal*: "Person name, Place and Date Identification and Extraction from a text document using FSA: A systematic review",* International Conference on Recent Trends in Computing, Mechatronics and Communication, Om Institute of Technology and Management, Hisar ISBN 978-81-923446-0-7, February, 2012.

Communicated

[1] R. Sharma and S. Garhwal: *"Extracting Person Name, Date and Place from Text Document using LEX Tool"*, International Journal of Computer Science.