Fuzzy Lexical Analyser: Design and Implementation

Vaishali Bhosale
Assistant Director/ Assistant Professor
YCSRD,
Shivaji University, Kolhapur

S.R. Chaudhari
Professor and Head
Dept. Of Mathematics, North Maharashtra
University, Jalgaon

ABSTRACT

The aim of this paper is to handle the errors, due to insertion, deletion, substitution, letter sequencing and typing in the lexical analysis phase of compiler. Fuzzy keywords, their fuzzy regular expressions and minimized fuzzy deterministic automaton are constructed. The issue of membership of fuzzy keyword is successfully tackled with the help of an algorithm. Full implementation of fuzzy lexical analyzer is also described.

Keywords

Fuzzy lexical analysis, Fuzzy finite automata, Fuzzy regular expression, Fuzzy tokens, Tiny compiler.

1. INTRODUCTION

Lexical analysis is a very important phase of a compiler that has the task of reading the source program character by character and separating it into tokens such as keywords, identifiers, special symbols, operators etc. [1]. Lexical analysis is also a special case of pattern matching that uses regular expressions and finite automata methods for string matching. A string is either a token or a non-token, and hence there is no middle possibility [2]. In traditional lexical analysis every token belongs to one and only one type viz. keywords, identifier, operators etc. with default membership value as 1. Whereas in fuzzy lexical analysis a token may belong to more than one token type with varying degree of membership in [0, 1].

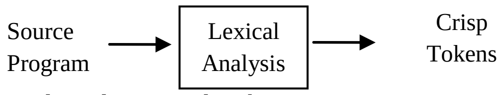
In ‘C’ programming language, if you type “integer” (due to substitution) it does not mean that it is the keyword “int” to the compiler, but it is treated as an identifier only. If you type “inttt” (an insertion error may be due to the key sticks), again it will also not treated as “int”. If you type “flot” it does not mean “float” to the compiler but as an identifier only (deletion error). Also if you type “vhar” it does not mean “char” to the compiler (a typing error). If you type “lese” it does not mean “else” to the compiler (letter sequencing error). Similarly if you type “wlse” due to tying error, it does not mean “else” to compiler.

Would it be more friendly if the compiler will simply decide for you “int” in first two cases, “float” in the third case and will it ask you whether you meant “char” in forth case and “else” in last two cases? The answer probably would be ‘no’ with existing compilers. Fuzzy lexical analysis will make it possible.

It means that the substitution, deletion, insertion, letter sequencing and typing errors are not allowed for keyword in crisp compiler. Due to the attempts by [2,3, 4] it is clear that the concept of fuzzy automata studied by many authors [5,6,7,8,9] will provide an appropriate framework to model such situations. This paper intended to address the problem of fuzzy token recognition. A fuzzy token is a sequence of characters which can have one or more of the errors due to insertion, deletion, letter sequencing and typing errors. Here, fuzzy automata model used for accepting fuzzy tokens and Tiny compiler considered for implementation.

2. PRELIMINARY

Regular languages are represented by regular expressions and they are analyzed by lexical analysis. The model of lexical analysis is shown in the following Figure 1. [10]



The lexical analysis is also known as scanning. In the following Figure 2, lexical analysis is explained with an example of variable declaration sentence.

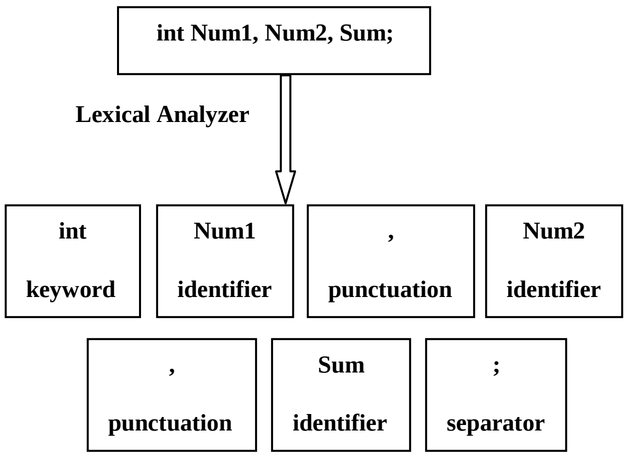


Figure 2

If the compiler received the sentence as:

inttt Num1, Num2,Sum;

then it gives error as “inttt” is not interpreted as a keyword “int”, but it may be treated as an identifier only. Fuzzy regular language can rectify this error. To widen the definition of keywords to allow errors due to insertion, deletion, letter sequencing and typing errors and the concept of regular fuzzy language defined by fuzzy regular expression found useful.

Since regular expressions (r.e.) are useful for representing certain sets of strings in an algebraic fashion. These r.e. describe languages that are accepted by finite state automata. This section of paper gives a formal recursive definition of regular expressions over a set $\sum$ as follows:

Definition2.1: [11] Let $\sum$ be any set. A regular expression over $\sum$ is recursively defined as:

1. Any terminal symbol (i.e. an element of ∑), ʌ (called null space) and ø (empty set) are regular expressions.
2. The union of two regular expressions R1 and R2, written as R1+R2, is also a regular expression.
3. The concatenation of two regular expressions R1 and R2, written as R1R2, is also a regular expression.
4. The iteration (or kleene closure) of a regular expressions R, written as R*, is also a regular expression.
5. If R is a regular expressions, then (R) is also a regular expression.
6. The regular expressions over ∑ are precisely those obtained recursively by the application of the rules 1-5 once or several times.

Definition 2.2:[2] Let ∑ be a finite alphabet and f: ∑*→[0,1]. Then the set Ã = { (w,f(w)) | w ϵ ∑* } is called a fuzzy language over ∑ and f the membership function of Ã.

Definition 2.3: [2] Let Ã be a fuzzy language over ∑ and f Ã : ∑*→[0 , 1] the membership function of Ã. Language Ã can be called a regular fuzzy language, if for each m ϵ M , S Ã (m) is regular, where S Ã (m) = { w ϵ ∑* | f Ã (w)=m }.

Example 2.1: Let Ã be a language over ∑ = { a , b} and f Ã be defined as

$$f\,\tilde{A}\,(s) = \begin{cases} 1, & \text{if } x \in ab^* \\ 0.8, & \text{if } x \in ab^*a \\ 0.5, & \text{if } x \in bab^* \\ 0 & \text{otherwise.} \end{cases}$$

Then Ã is a regular fuzzy language, since S Ã (m), for m ϵ {0, 0.5, 0.8, 1.0}, is a regular expression.

Similarly if
$$f_{\tilde{B}}\,(x) = \begin{cases} 0.8 & \text{if } x \in a^n b^n \text{ where } n > 0 \\ 0.7 & \text{if } x \in a\ \ b^* \\ 0 & \text{otherwise.} \end{cases}$$

Then $\tilde{B}$ is not a fuzzy regular language, since $S_{\tilde{B}}$ (m), for m ϵ {0.8}, is not a regular expression.

Fuzzy regular expressions over the alphabet ∑ are defines as:

Definition 2.4: [2] Let e be a regular expression over ∑ and m Є [0, 1]. Then (e) / m is a fuzzy regular expression. If e1, e2, . . . , en are fuzzy regular expressions over ∑ and m1, m2,. . . , mn are their respective degrees, then one can write it as

e1 / m1 + e2 / m2 + . . . + en / mn such form of writing of regular expression is called normalized form of the fuzzy regular expression. Regular expression verses fuzzy regular expression discussed in the following example:

Example 2.2: Regular expression for two keywords ‘int’ and ‘if’ of ‘C’ language given as:

r = (int + if), whereas fuzzy regular expression for them may be given as:

int/1 + if/1 + int(t+)/0.8 + if(f+)/0.8 + integer/0.7.

Remark: The problem of assignment of the degree of membership to fuzzy keywords is global in nature. Here in this paper researcher have developed an algorithm to resolve this problem. (see algorithm 3.2)

Definition 2.5: [11] A nondeterministic finite automata (NDFA) is a 5-tuple, (Q, Σ, δ, q0, F), where Q is finite nonempty set of states; Σ is finite nonempty set of inputs; δ is the transition function mapping from Q × Σ into 2Q , q0 ∈ Q is initial state; and F is subset of Q is the set of final states.

Instead of F as subset of Q, if F is a fuzzy subset of Q, then the nondeterministic finite automata is treated as a nondeterministic automata with fuzzy final states (NFA-FS).

The fuzzy language accepted by Ã, is denoted by L(Ã), is the set { (x, d Ã (x)) | x ϵ ∑* }, where d Ã (x) = max { F Ã (q) | (s, x, q) ∈ δ* }.

Definition 2.7: [11] A deterministic finite automata (DFA) is represented by a 5-tuple, (Q, Σ, δ, q0, F), where Q is finite nonempty set of states; Σ is finite nonempty set of inputs; δ is the transition function mapping from Q × Σ into Q and is usually called direct transition function. This is the function which describes the changes of states during the transition. The mapping δ is usually represented by a transition table or a transition diagram. q0 ∈ Q is initial state; and F is subset of Q is the set of final states.

If F is fuzzy subset of Q instead of crisp subset of Q, then the DFA is called deterministic finite automata with fuzzy final states (i.e. FS-DFA). The fuzzy language accepted by Ã denoted by L(Ã), is the set { (x, d Ã (x)) | x ϵ ∑* }, where

$$d_{\tilde{A}(x)} = \begin{cases} F(q), \text{if } \delta(s,x) = q \\ 0 \ \text{ if } \delta(s,x) \text{is not defined} \end{cases}$$

Theorem 2.1: [11] For every NDFA, there exists a DFA that simulates the behavior of NDFA. The converse of the theorem is trivial. Theorem below recite the important relation between NFA-FS and FS-DFA:

Theorem 2.2:[2] A fuzzy language is accepted by a NFA-FS iff it is accepted by a FS-DFA.

Proof : If Ã (Q, Σ, δ, s, F) is NFA-FS for fuzzy language L,

then the construction of FS-DFA Ã′ = (Q′,Σ, δ′, s′, F′) is

straightforward. One can just use the standard subset

construction method and for each P ϵ Q′ i.e.(P is subset of Q)

one can define

$$F'(P) = \max\{\, m \mid m = F(q),\ q \in P \,\}$$

The DFA is minimized by finding equivalent states and using minimization algorithm as stated in [6]. In the same way the fuzzy state DFA can be minimized by finding equivalent states and using minimization algorithm as stated in [2].

3. FUZZY LEXICAL ANALYSIS

In designing lexical analyzer for any language, first regular expressions are constructed. Lex scanner generator can be used to generate a scanner from a description of the tokens as regular expressions. On the similar lines to design fuzzy lexical analyzer, begin with fuzzy regular expressions (FREs) first. Lex tool is not used because tokens in fuzzy scanner may belong to more than one categoty with varying degree of membership between [0,1]. So from FREs develop fuzzy NFAs and fuzzy DFAs in later stages. Tiny compiler is considered as a crisp compiler. It takes as input a program written in Tiny language and converts it into assembly language. Consider scanner for Tiny language which performs lexical analysis of input program and gives tokens as output.

Table 1: Tokens of the Tiny language

Keywords	Special symbols	Other
If	+	Number(1 or more digits)
Then	-	
Else	*	
End	/	
Repeat	=	Identifier(1 or more letters)
Until	<	
Read	(	
Write	)	
	;	
	:=	

The tokens of Tiny language generally fall into three categories: keywords, special symbols and other tokens. This target compiler have eight keywords, with familiar meanings [1]. There are 10 special symbols, giving the four basic arithmetic operations on integers, two comparison operations (equal and less than), parentheses, semicolon and assignment. All special symbols are one character long except for assignment, which is of length two. For ready reference those tokens are summarized in the above **Table 1**.

An algorithm for fuzzy lexical analysis is proposed below. First consider fuzzy tokens for all eight keywords. Broadly fuzzy lexical analyzer will scan input program character by character and group them into fuzzy tokens with a degree of membership. This uses fuzzy regular expression, fuzzy NFA, fuzzy DFA and fuzzy DFA minimization. To allow fuzzy token recognition proceed as per following algorithm:

```
Algorithm 3.1 Fuzzy Lexical Analysis:
Step 1: Construction of fuzzy regular expressions for keywords, for assignment and equal operators exists in Tiny language.
Step 2: Design of FS-NFA for above fuzzy regular expressions.
Step 3: Construction of FS-DFA for FS-NFA.
Step 4: Minimization of FS-DFA if possible.
Step 5: Implementation of fuzzy lexical analyzer.
```

Few samples of fuzzy tokens for all the 8 crisp keywords are given in the form of table as follows:

Table 2: Sample Fuzzy Tokens for keywords in Tiny language

Key word	InsertionError	Deletion Error	Substitution Error	TypingError	Character Sequencing errors
If	Iifff	-	-	-	-
then	thhheenn	thn	-	rhen	Hten
else	eeelllsssee	ele, lse	Otherwise	slse,elsr	Lees
end	eendd	-	-	wnd	Edn
repeat	rrepeat	repet	-	eepeat	Erepat
until	until	untl	-	yntil	Nuitl
read	rreeaadd	rea, ead	Input	eead	Dear
write	writttttte	rite, wite	Output	qrite	ritew

The actual implementation of fuzzy lexical analysis step by step is discussed in details as follows:

Step1: Construction of Fuzzy regular expressions:

Firstly fuzzy regular expressions will be constructed for keywords that exists in Tiny language which allow insertion, deletion, substitution, letter sequencing and typing errors. Also this compiler allows synonyms for keywords wherever possible. FREs for all the keywords are constructed below:

a) The FREs for reserved word "if":

if / 1 + (ii+ff* + ii*ff+) / m

Only insertion error is considered for keyword "if". For the sake of convenience no deletion, substitution, letter sequencing and typing errors will allowed for "if" the keyword, as it has string length two.

b) The FREs for reserved word "then":

then / 1 + (tt+hh*ee*nn* + tt*hh+ee*nn* + tt*hh*ee+nn*+ tt*hh*ee*nn+) / m1 + (tt*hh*ee*n* + tt*hh*e*nn* + t*hh*ee*nn*) /m2 + (t+h+e+n)+/m3 + ((r+g+y+t)hen +t(g+y+j+n+h)en +th(e+w+d+r)n +the(n+b+h+m))/m4

The fuzzy regular expressions can be simplified and put together as :

then/1 + (t+h+e+n)+ /m1 + ((r+g+y+t)hen + t(g+y+j+n+h)en + th(e+w+d+r)n + the(n+b+h+m))/m2

Hence onwards only simplified fuzzy regular expressions for the remaining keywords are given.

c) The FREs for reserved word "else":

else / 1+ (e+l+s)+/m1+((r+w+d+e)lse +e(k+o+p+l)se +el(a+w+d+s)e+els (e+w+d+r)) /m2

d) The FREs for reserved word "end":

end/1+(e+n+d)+/m1+ ((e+w+d+r)nd +e(n+b+h+m)d +en(d+e+s+c+f))/m2

e) The FREs for reserved word “repeat”:

repeat / 1 + (r+e+p+e+a+t)+ /m1 + ((r+e+t+f)epeat + r(e+w+d+r)peat + re(p+o+l)eat + rep(e+w+d+r)at + repe(a+s+q+z)t+ repea(t+r+g+y))/m2

f) The FREs for reserved word “until”:

until/1+(u+n+t+i+l)+/m1+ ((u+y+j+i)ntil +u(n+b+j+m)til +un(t+r+g+y)il +unt(i+u+k+o) l +unti(l+k+o+p))/m2

g) The FREs for reserved word “read”:

read/1 + (r+e+a+d)+ / m1 + (input)/0.6 + ((r+e+f+t)ead +r(e+w+d+r)ad+re(a+z+s)d+rea(d+s+c+f+e))/m2

h) The FREs for reserved word “write”:

write / 1 + output/0.6 + +(w+r+i+t+e)+/m1

+ (w+q+s+e)rite + w(r+e+f+t)ite + wr(i+u+k+o)te + wri(t+r+g+y)e+writ(w+e+r+d))/m2

Note that fuzzy tokens due to substitution errors have fixed predefined membership value (in fact it is 0.6). For example output/0.6 (substitution error for keyword write) and input/0.6 (substitution error for keyword read). For other fuzzy tokens value of m is calculated runtime for each input string separately. The algorithm to compute membership value is given below

Algorithm 3.2:

```
Step 1: Find length L of each crisp keyword.
Step 2: Find occurrences of each letter l in the crisp keyword as O( l ).
Step 3: Find degree of each letter for fuzzy token as
D( l )=(1/L)*O( l )
Step 4: Initialize A ( l ) = 0.0 for all letters in crisp keyword and M (keyword) = 0.0
Step 5: For each letter ‘l’ in input string if ‘l’ is in crisp keyword then update
            A ( l ) = A ( l ) + 1
Step 6: Compute actual degree of each letter d ( l ) as below:
    If A( l ) > 0 then
    {    If (A ( l ) > O ( l )   then D ( l )= D ( l ) /A ( l )
         Else if A ( l ) = = O ( l )       then  D ( l )= D( l )
              Else  D ( l )= (A ( l ) / O ( l )) * D ( l )
    }
Step 7: M ( l ) = sum (D ( l ) ) for all l in crisp keyword
Step 8: If M ( l ) > 0.5, then it is recognized as a fuzzy keyword else as an identifier.
```

Example 3.1: Consider the fuzzy “else” keyword, having the length of “else” is L(“else”) = 4, and occurrences of e in “else” is O (e) = 2. Similarly O (l) = O (s) = 1 in “else”. Therefore by above algorithm, the degree for each character in “else” will be calculated as follows:

d (e) = 2 /4 = 0.5 d(l) = d(s) = 1 / 4 = 0.25.

The membership degree of the input string “else” will be calculated as follows:

M(“else”) = d(e) + d(l) + (s)

Few more examples of fuzzy keywords and computation of their membership is summarized in the following table 3.

Table 3 Membership value computation

input string	A (e)	A (l)	A (s)	D(e)	D (l)	d(s)	M(input string)
Lese	2	1	1	0.5	0.25	0.25	1
Els	1	1	1	0.25	0.25	0.25	0.75
Wlse	1	1	1	0.25	0.25	0.25	0.75
Else	3	1	1	0.17	0.25	0.25	0.67
Seel	2	1	1	0.5	0.25	0.25	1

2. Design of FS-NFA for fuzzy regular expressions.

In this section, nondeterministic finite automata for fuzzy regular expressions of the fuzzy tokens discussed in the above section are designed. This is possible due to the following theorem.

Theorem 2.1: For every fuzzy regular expression r there exists a FS-NFA Ã such that the language accepted by Ã is r. Conversely, the language accepted by given FS-NFA Ã is always expressed as a fuzzy regular expression.

Here, FS-NFA for fuzzy regular expressions given in the above step 1 (See (a) to (h)) are constructed with the help of above theorem and closure properties of fuzzy regular languages. In section below only representative FS-NFA diagrams for fuzzy “then” and “read” i.e. for FREs ‘b’ and ‘g’ are given.
Note that all these FS-NFAs with ε-moves converted into equivalent FS-DFA using ε-closure method. The degree of acceptance of the (fuzzy) keyword is the same as the membership degree of that keyword calculated according to algorithm 3.2.

3. Construction of FS-DFA for FS-NFA.

The construction of FS-DFA from FS-NFA according to the theorem 2.2 will be done in this section. These FS-DFAs are represented as transition tables for simplicity in Table 4 and Table 5.

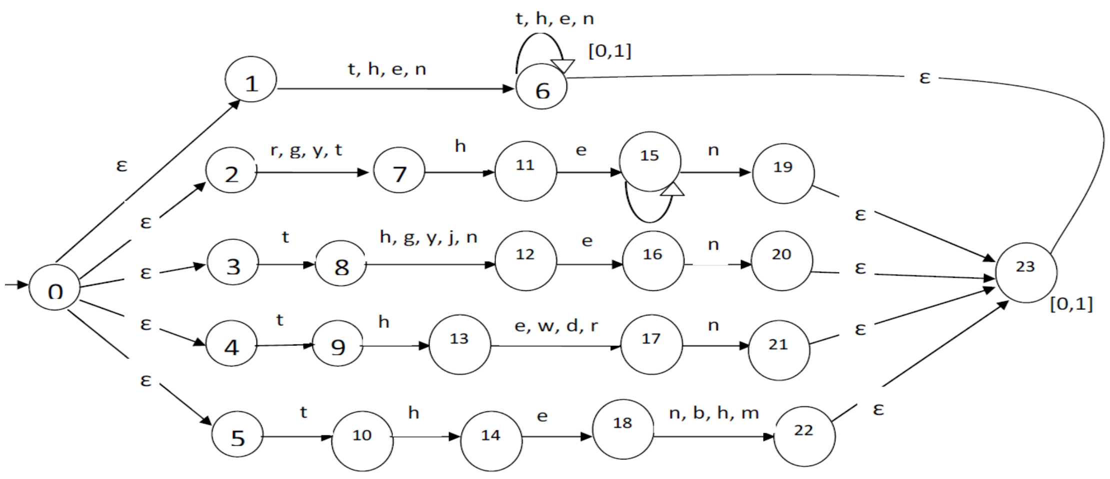


Figure 3: Fuzzy State –Non deterministic Finite Automata for "then"

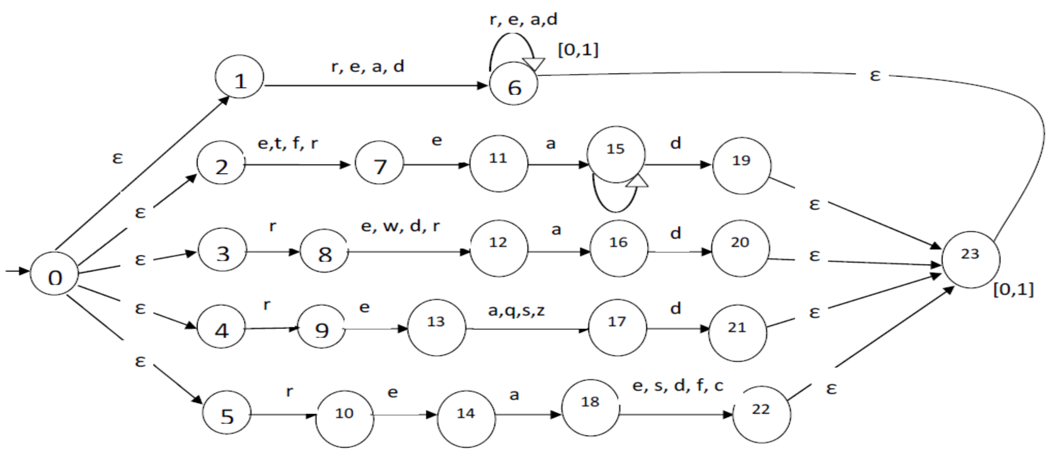


Figure 4 Fuzzy State –Non deterministic Finite Automata for "read"

Table 4 Fuzzy State -DFA for "then"

s\∑	t	h	e	n	r	g	y	j	w	d	b	m
A	B	C	C	C	D	D	D	-	-	-	-	-
B	C	E	C	F	-	G	G	G	-	-	-	-
C	C	C	C	C	-	-	-	-	-	-	-	-
D	-	H	-	-	-	-	-	-	-	-	-	-
E	C	C	I	C	J	-	-	-	J	J	-	-
F	C	C	K	C	-	-	-	-	-	-	-	-
G	-	-	L	-	-	-	-	-	-	-	-	-
H	-	-	M	-	-	-	-	-	-	-	-	-
I	C	O	C	N	-	-	-	-	-	-	P	P
J	-	-	-	Q	-	-	-	-	-	-	-	-
K	C	C	C	R	-	-	-	-	-	-	-	-
L	-	-	-	S	-	-	-	-	-	-	-	-
M	-	-	-	T	-	-	-	-	-	-	-	-
N	C	C	C	C	-	-	-	-	-	-	-	-
0	C	C	C	C	-	-	-	-	-	-	-	-
P	-	-	-	-	-	-	-	-	-	-	-	-
Q	-	-	-	-	-	-	-	-	-	-	-	-
R	C	C	C	C	-	-	-	-	-	-	-	-
S	-	-	-	-	-	-	-	-	-	-	-	-
T	-	-	-	-	-	-	-	-	-	-	-	-

Table 5: Fuzzy State -DFA for "read"

s\∑	r	e	a	d	t	f	w	q	s	z	c
1	2	4	4	4	6	6	-	-	-	-	-
2	4	8	4	4	-	-	9	-	-	-	-
4	4	4	4	4	-	-	-	-	-	-	-
6	-	12	-	-	-	-	-	-	-	-	-
8	4	4	14	4	-	-	-	15	15	15	-
9	-	-	15	-	-	-	-	-	-	-	-
12	-	-	15	-	-	-	-	-	-	-	-
14	4	4	4	4	-	18	-	-	18	-	18
15	-	-	-	18	-	-	-	-	-	-	-
18	-	-	-	-	-	-	-	-	-	-	-

4. Minimization of FS-DFA:

The minimization algorithm for DFA can also be extended for FS-DFA. In table 4 States (P, Q, S, T) and (R, O, N, C) found equivalent and hence states (L,M), (G, H) and (I, K, C, F) are also equivalent. This leads to states (J, L) are also equivalent. Hence the minimized FS-DFA for fuzzy "then" will be as shown in Figure 5 below:

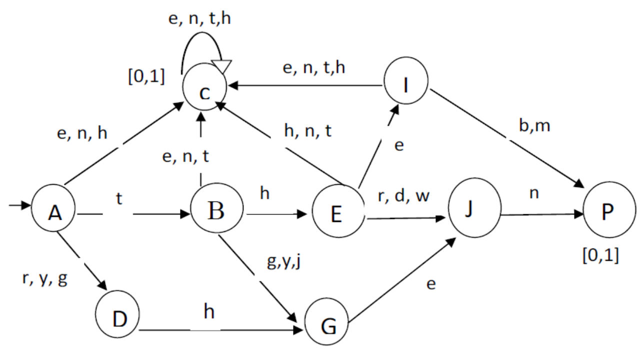


Figure 5: Minimized Fuzz State -DFA for "then"

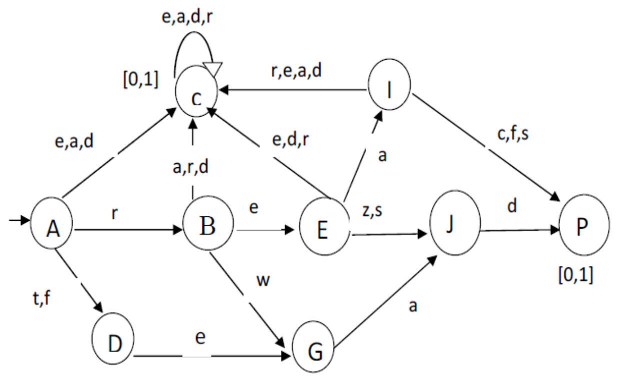


Figure 6 Minimized Fuzzy State -DFA for "read"

5. Implementation of fuzzy lexical analyzer:

This section explains full implementation of fuzzy lexical analyzer using all the steps. In the previous section, fuzzy regular expressions (FREs) for all eight keywords in Tiny Language (in step 1 of algorithm 3.1) are defined. In the implementation phase fuzzy tokens themselves are defined using enumerated types as below:

```
{ENDFILE,ERROR, IF, THEN, ELSE,END, REPEAT,
UNTIL, READ,WRITE,ID, NUM, ASSIGN ,EQ ,LT ,PLUS,
MINUS, TIMES, OVER,LPAREN,RPAREN,SEMI,
/* Fuzzy keywords */
FIF,FTHEN,FELSE,FEND,FREPEAT,FUNTIL,FREAD,
FWRITE } TokenType;
```

In the step 2, FS-NFA for each FRE have designed. FS-NFA are then converted to FS-DFA in step 3. In step 4 minimized FS-DFA for each fuzzy keyword is constructed. Fuzzy lexical analyzer implemented using switch-case constructs of 'C' programming language. The table "KeyWords" stores keyword structures as given below:

```
{{"if",IF},{"then",THEN},{"else",ELSE},{"end",END},
{"repeat",REPEAT}, {"until",UNTIL}, {"read",READ},
{"write",WRITE}, {"input",READ}, {"output",WRITE}};
```

In the implementation step sequence of alphabets is accepted as identifier first. The procedure call performs a lookup of crisp keywords, substitution keywords by string comparison.

```
strcmp(input_string, KeyWords)

    return KeyWords;
```

If no match found, then the "reserved_lookup" calls "check_fuzzy" function is used to check the token for fuzzy keyword due to insertion, deletion, letter sequencing and typing errors, if any.

```
currentToken=checkfuzzy();

        return currentToken;
```

Again if there also no match found, then current token type retained as an identifier (i.e. ID) only. The experimental results shows that the fuzzy keyword belongs to more than one category with varying degree of membership. Default membership of crisp keyword is 1, for substitution keyword it is predefined as 0.6, for character sequencing error it is 1 and computed runtime for insertion, deletion and typing errors. All fuzzy keywords are identifiers. Therefore all fuzzy keywords have membership value 1 for token type identifier.

Experimental Results: Consider input file contain following strings

"else input elsee eend repat rhen util output "

The fuzzy lexical analyzer scans input file character by character from left to right and separates the tokens. Note that only one of the token in input file above "else" is crisp and rest all are fuzzy tokens. The fuzzy keywords are not reserved words, i.e. the fuzzy keywords can also be used as an identifier name. Therefore the fuzzy keyword string and it's place in the sentence is important to finalize its token category.

The result of fuzzy lexical analysis are summarized in table below:

Table 6: Sample input strings

Input String	M(keyword)	M (identifier)	Crisp Keyword	Type of error occurred
else	1	0	else	No error
input	0.6	1	read	Substitution error
elsee	0.67	1	else	Insertion error
eend	0.8	1	end	Letter sequencing error
repat	0.85	1	repeat	Deletion error
rhen	0.75	1	then	Typing error
util	0.8	1	until	Deletion error
output	0.6	1	write	Substitution error

For example:

//program 1	//program 2	//program 3
read x;	input x;	read input;
write x	output x	write output

In the example above first program is using crisp keywords read and write. In second program read and write keywords are replaced by substitution words input and output respectively. But in third program input and output are used as variable names. Thus the final category of fuzzy token depends on the token it follows or token that follows it. So need is to change grammar rules accordingly to adapt fuzzy lexical analysis as its output is input to syntax analysis. So the need is to design fuzzy parser to allow fuzzy tokens for keywords and for operators.

4. CONCLUSION

The paper described the possibility of fuzziness in keywords due to insertion, deletion, substitution, typing and letter sequencing errors and their implementation in this paper. The implementation is restricted to those mentioned errors. The synonyms for programming language constructs from natural language can be used to allow fuzzy tokens. The work can be further extended to allow more flexibility in tokens such that the program will look like psuedocode. In this paper the implementation of fuzzy keywords is fully emphasized. The approach is to use fuzzy automata concepts to allow flexibility (or fuzziness) in token recognition process i.e. lexical analysis. It is termed as fuzzy lexical analysis. As a result of fuzzy lexical analysis a token may belong to more than one category with different degree of membership. To finalize the token category the need is to go further and discuss fuzzy parsing. In future the work can be extended for fuzzy parsing that will finalize the token category mainly based on its position in the given sentence. Fuzzy context free grammar will allow fuzziness in syntax analysis phase of compiler in order to model grammatical errors. The work can be extended using fuzzy translation rules for syntax directed semantic analysis for fuzzy relations.

5. REFERENCES

[1] Kenneth C. Louden, Compiler Construction Principles and Practice, Cengage Learning, India Edition, 2008.

[2] Mateescu A., Salomaa A., Salomaa K., Yu S., Lexical Analysis with a Simple Finite Fuzzy Automaton Model, Journal of Universal Computer Science, 1995, 292-311.

[3] Bai M. Q., Sun F., Mo Z., Closure and Commutation of Fuzzy Regular Languages, 2009 IEEE.

[4] Bai M. Q., "On the Relation between Fuzzy Regular Expression and Fuzzy Finite State Automata", Pure and Applied Mathematics, 16(2000)4, pp. 1-6.

[5] Gupta M. M., Saridis G. N. and Gaines B. R., Fuzzy Automata and Decision Processes, North-Holland, New York, 1977, pp149-168

[6] Kumbhojkar, H. V. and Chaudhari, S. R.(2002b), "Fuzzy Recognizers and recognizable sets", Int. J. of Fuzzy Sets and Systems, Vol. 131, pp. 381-92.

[7] Mordeson J. N., Malik D. S., Fuzzy Automata and Language: Theory and Applications, Chapman & Hall, 1 edition, 2002.

[8] Santos E. S. (1968a), "Maxmin automata", Information and control, Vol.21, pp. 27-47.

[9] Wee W. G. and Fu, K. S. (1969), "A formulation of fuzzy automaton and its application as a model of learning systems", IEEE Trans Syst. Sci. Cyber, Vol. 5, pp. 215-23.

[10] Hopcroft, Motwani, Ullman,Introduction to Automata Theory, Languages and Computation, Pearson Education.

[11] Mishra K.L.P., Chandrasekaran N., Theory of Computer Science, Prentice Hall of India, 1998.

[12] Asveld P.R.J., Fuzzy Context Free Languages – Part 1, Generalised fuzzy context free grammars, Theore. Comp. Sc., volume (2005).

[13] Asveld P.R.J., Fuzzy Context Free Languages – Part 2, Recognition and Parsing Algorithms, Theore. Comp. Sc., volume(2005), pp 191-213.

[14] Hsuan Shih Lee, Minimizing Fuzzy finite Automaton, 2000 IEEE.

[15] Kremer D., New Directions in Fuzzy Automaton, 2005 Science Direct.

[16] Lee E. T. and Zadeh L. A., "Note on Fuzzy Languages", Information Sciences, 1(1969) 421-434.

[17] Xuli, Jinga, Diancheng, A Fuzzy Automaton Model and It's Applications, 1996 IEEE.