International Journal of Engineering and Advanced Technology (IJEAT)
ISSN: 2249-8958 (Online), Volume-8 Issue-6S3, September 2019

An Fuzzy Lexical Research

L.Latha, Preethi M, Grahalakshmi R

Abstract---The objective of this paper is to analyse the design and implementation of the fuzzy lexical analyser and observe how it is different from the traditional lexical analyser. It is known that lexical analysis is an important phase of a compiler. It reads the source program character by character and uses regular expressions, finite automata methods for string matching. Unlike traditional lexical analysers, tokens in fuzzy analysers belong to more than one token type with varying degree of membership. The paper exchange views on the design and implementation of fuzzy lexical analysers. It observes algorithms that handle errors due to insertion, deletion etc. in the lexical analysis phase of a compiler. Several properties of fuzzy languages are also reviewed. Hence this paper gives a comprehensive view of fuzzy regular languages, models and algorithms

I. INTRODUCTION

There are 7 phases of a compiler. The lexical analyser is the first phase of the compiler. It converts the source code (human understanding) into tokens. When the source code has whitespace or comments, lexical analyser will remove them.

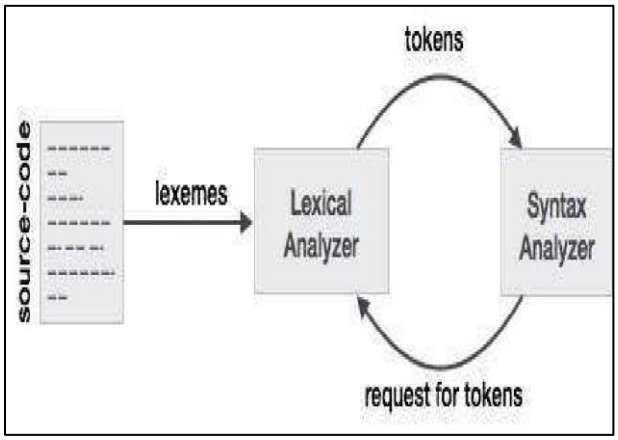

Figure 1 - Lexical Analysis

The main work of fuzzy lexical analyser is that it reads the input from source code and group them into lexemes (character) and produces tokens (eg num, id, const, if, etc) (take keywords, identifier, operators, special symbols)

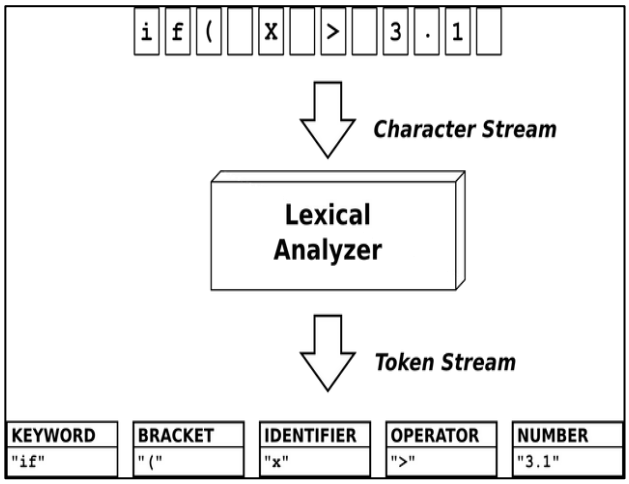

Figure 2 - Separating Characters into Tokens

Pattern matching concept is used for this lexical analyser. When we type “intt”, other compilers don’t get that it means “int”. It is treated as an

identifier. When we type “flot”, other compiler doesn’t get that it means “float”. Fuzzy lexical analyser will remove this problem. It identifies these identifiers by using pattern matching concepts. The tool used to construct a lexical analyser is “lex”.

II. PARSER GENERATION OF LEXICAL ANALYSER

Parser can be defined as top-down or bottom-up based on how the parse-tree is constructed.

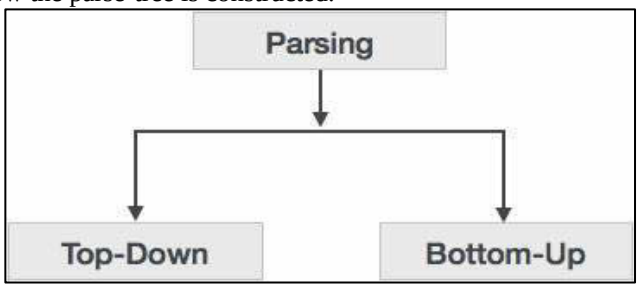

Figure 3 – Parsing types

Two Types Of Top-Down Parsing Is There:

1. Recursive descent parsing
2. Backtracking

Steps To Be Followed In Parser:

a) Identify the words construct from a given input. A parser outputs and represents convincing data in the form of a parse tree

b) For grammatically wrong data string the parser declares the recognition of syntax error. No parse tree in this case is formed.

Manuscript published on 30 September 2019.
* Correspondence Author (s)
L.Latha, Professor, CSE, Kumaraguru College of Technology, Coimbatore, Tamil Nadu, India.(Email: raviveera@gmail.com)
Preethi M, UG student, Kumaraguru College of Technology, Coimbatore, Tamil Nadu, India.
Grahalakshmi R, UG student, Kumaraguru College of Technology, Coimbatore, Tamil Nadu, India.

As an different and alternating to parsing based on the repetitive application of original expression using a shortest-match strategy, we may apply an iterative lexical technique.

III. PRELIMINARY

A regular language is analysed by lexical analyser and the regular language is represented by using the regular expression.

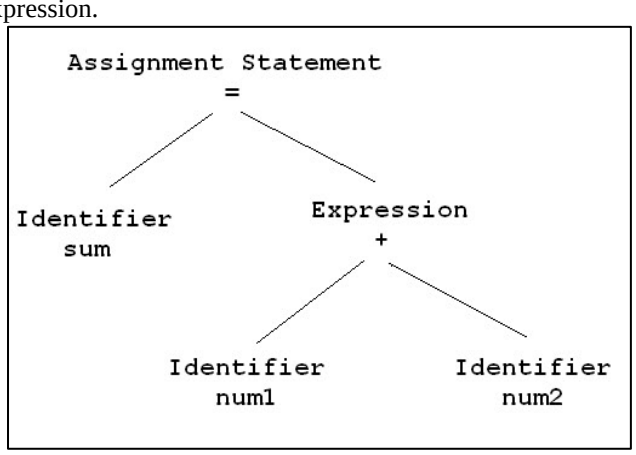

Figure 4 - Example of Lexical Analysis

In the above example, if we receive the input
"sum=num1+ num2"
Lexical analyser separates these input into identifier, operators, separators.

Method To Convert Fa To Regular Expression

Theorem:

If L=L(A) for some DFA A, then there is a regular expression R such that L=L(R).

Proof:

Let 1,2,3…n be the states present in A .Let $R_{ij}^{(R)}$ denote the regular expression from state i to state j with k intermediate states.

This can be proved by applying induction on k.

Basis:

K=0 (i.e) (number of intermediate states)

Let $R_{ij}^{(0)}$ denote the corresponding regular expression then there are two cases.

They are,

CASE 1:

I != j

Automaton:

a)

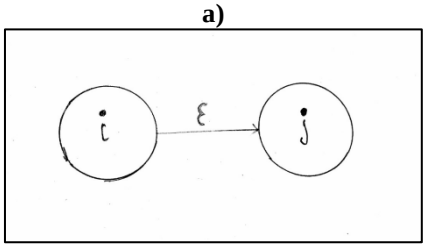

Figure 5

Regular expression: $R_{ij}^{(0)} = \varepsilon$
Loop: ε+ε

b)

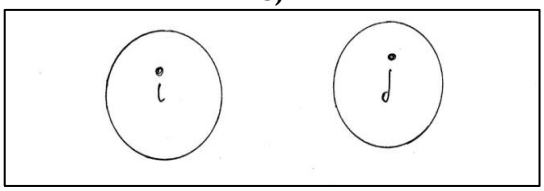

Figure 6

Regular expression: $R_{ij}^{(0)} = \Phi$
Loop: ε+ Φ= ε

c)

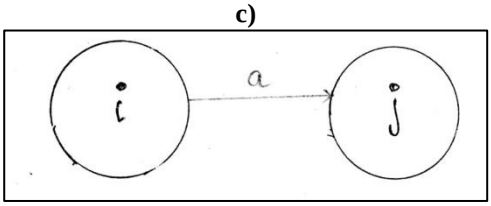

Figure 7

Regular expression: $R_{ij}^{(0)} = a$
Loop: ε+a

CASE 2:

i = j :

a)

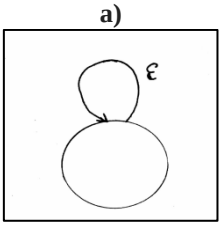

Figure 8

$R_{ij}^{(0)=}\varepsilon+\varepsilon = \varepsilon$

b)

$R_{ij}^{(0)=}\varepsilon+ \Phi = \varepsilon$

c)

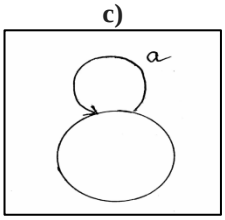

FIGURE 9

$R_{ij}^{(0)=}\varepsilon+ a$

Induction:

Assume the theorem is true for some (k-1) state . Let $\mathbf{R_{ij}}^{(k-1)}$ be the corresponding regular expression.

Now , we need to prove it for some k states. The different possibilities to reach state j from state I through 'k' intermediate states is show in the following diagram,

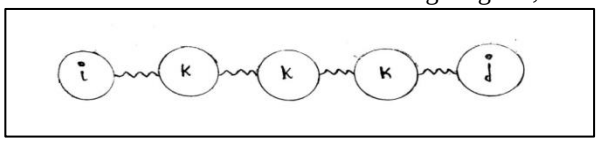

Figure 10

CASE 1:

Without moving through state k.

$R_{ij}^{(k)} = : R_{ij}^{(k-1)}$ (equation 1)

CASE 2:

Through state 'k'

$R_{ij}^{(k)} = R_{ik}^{(k-1)}. (R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$ (equation 2)

COMBINING 1 AND 2:

$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)} .(R_{kk}^{(k-1)})^* . R_{kj}^{(k-1)}$

Thus proved.

EXAMPLE :

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

K = 0

$R_{11}^{(0)} = \varepsilon + 1$

$R_{12}^{(0)} = 0$

$R_{21}^{(0)} = \Phi$

$R_{22}^{(0)} = \varepsilon + 0+1$

K = 1

$R_{11}^{(1)} = R_{11}^{(0)} + R_{11}^{(0)} (R_{11}^{(0)})^* R_{11}^{(0)}$

$= (\varepsilon + 1)+ (\varepsilon + 1) (\varepsilon + 1)^* (\varepsilon + 1)$

$=(\varepsilon + 1)+ (\varepsilon + 1) (\varepsilon + 1)^*$

$=(\varepsilon + 1)+(\varepsilon + 1)^*$

$= (\varepsilon + 1)^*$

$=1^*$

$R_{12}^{(1)} = R_{12}^{(0)} + R_{11}^{(0)} (R_{11}^{(0)})^* R_{12}^{(0)}$

$=0+(\varepsilon + 1) (\varepsilon + 1)^* . 0$

$=0+1^*0=1^*0$

$R_{21}^{(1)} = R_{21}^{(0)} + R_{21}^{(0)} (R_{11}^{(0)})^* R_{11}^{(0)}$

$= \Phi + \Phi (\varepsilon + 1)^* (\varepsilon + 1)$

$= \Phi$

$R_{22}^{(1)} = R_{21}^{(0)} + R_{21}^{(0)} (R_{11}^{(0)})^* R_{12}^{(0)}$

$=(\varepsilon + 0+1) + \Phi (\varepsilon + 1)(0)$

$= \varepsilon + 0+1$

FINAL STATE:

$R_{12}^{(2)} = R_{12}^{(1)} + R_{12}^{(1)} (R_{22}^{(1)})^* R_{22}^{(1)}$

$=1^*0 + (1^*0) (\varepsilon + 0+1)^* (\varepsilon + 0+1)$

$=1^*0 + 1^*0 (\varepsilon + 0+1)^*$

IV. FUZZY LEXICAL ANALYSIS& RESULTS

The tokens of tiny lexical analyser fall into three categories: Keywords, special symbols and tokens.

To design a fuzzy lexical analyser, we have to generate fuzzy lexical expressions(FRE's). Tokens in fuzzy scanner may belong to more than one category, so we have to construct fuzzy NFAs and fuzzy DFAs.

Example tokens, lexemes, patterns

Token	Sample Lexemes	Informal description of pattern
if	if	if
While	While	while
Relation	<, <=, = , <>, > >=	< or <= or = or <> or > or >=
Id	count, sun, i, j, pi, D2	Letter followed by letters and digits
Num	0, 12, 3.1416, 6.02E23	Any numeric constant

Table 1 keywords, special characters and tokens

Let's discuss the following algorithm for fuzzy lexical analysis:

The fuzzy lexical analyser reads the input character by character and groups them into fuzzy tokens.

ALGORITHM I

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 the fuzzy lexical analyzer.

ALGORITHM II

Where S=Input string.

Output: A set of tokens.

Step 1: Initialize S.

Step 2: Define the symbol table.

Step 3: repeat while scanning S is not completed.

1. if blank(empty space)
 a. Neglect and eliminate it.
2. if operator op //arithmetic, relational, etc.
 a. find its type.
 b. write op.
 3. if keyword key//if, while
 a. write key.
 4. if identifier id //a,b,c
 a. write id
 5. if special character sc //(.), etc
 a.write sc.

V.CONCLUSION

The implementation of the fuzzy keywords is possible. It makes the token recognition process more flexible (fuzzy). Hence, termed as the fuzzy lexical analyser. A token may now belong to one or more category. In this paper, we have surveyed two such algorithms.

REFERENCES

1 https://www.researchgate.net/publication/281719677_Fuzzy_Lexical_Analyser_Design_and_Implementation
2 https://www.semanticscholar.org/paper/Fast-join%3AAn-efficient-method-for-fuzzy-token-join-WangLi/07506336397160283caeef4a0173cfeb4ef4e45a
3 http://www.dma.fi.upm.es/recursos/aplicaciones/logica_borrosa/web/fuzzy_inferencia/funpert_en.htm
4 https://www.google.com/search?q=parsing+in+compiler+design&rlz=1C1CHBD_enIN820IN820&source=lnms&tbm=isch&sa=X&ved=0ahUKEwiFn6C-oaLfAhWCpo8KHW06CbkQ_AUIDigB&biw=999&bih=638#imgrc=lsyuo1WxOFhUhM:
5 https://www.ieee-pes.org/images/files/pdf/pg4-sample-conference-paper.