Fuzzy Syntax Analyzer for Tiny Language

[1]Vaishali P. Bhosale, [2]Shrikant R. Chaudhari
[1]Assistant Director/Assistant Professor, [2]Professor
[1]YCSRD, Shivaji University, Kolhapur.
[1]Dept. Of Mathematics, North Maharashtra University,
Jalgaon, Maharashtra (India) - 425001.

Abstract— In this paper we considered design and implementation of fuzzy version of recursive-descent parser – a top down parser- as it is the most suitable method for handwritten parser. Tiny language - as defined in [3] is used as source language. The context free grammar for tiny language is extended to accommodate flexibility while writing programs. The implementation is tested with sample programs.

Index Terms— Recursive descent parser, Tiny language, Fuzzy CFG, Fuzzy syntax analysis.

I. INTRODUCTION

Syntax analysis is also called as parsing and it follows the lexical analysis phase of the compiler. As we know lexical analyzer reads source program and produces a sequence of tokens. From these tokens the syntax analyzer (i.e. parser) determines the structure of a program and results in parse tree (or syntax tree), if the program is correct and otherwise indicates the error/s.

We know that the regular expressions are used to specify tokens, whereas context free grammar (CFG) is used to specify the syntactic structure of a programming language and it involves recursive rules. Grammar rule uses regular expressions as components and gives precise and easy way to understand syntactic specification for the programs of a particular programming language. In general, CFG involves four elements namely: terminals, non-terminals, a start symbol, and the set of production rules. Syntax analysis involves a choice from among a number of different methods, each of which has distinct properties and capabilities. There are two general categories of parsing algorithms: top down parsing and bottom up parsing. Bottom-up parsers build parse trees from the bottom (i.e. leaves) to the top (i.e. root), while top down parsers start with the root and work down to the leaves. In both cases the input to the parser being scanned from left to right, one symbol at a time.

II. NEED OF FUZZY SYNTAX ANALYSIS

In 'C' programming language if you will write the declaration statement as follows:

```
Integer num, sum;
```

Then the parser reports a syntax error, since the word "integer" do not belongs to the (crisp) class of keywords. Again if you write "do…… while" syntax as follows:

```
Repeat
        x= x-1;
Until x= =0;
```

Then the parser will again report an error, as there is no such "repeat……until" syntax in 'C' language. Similarly if you write

```
x: = 5 instead of x = 5
```

Then also the parser reports an error, as there is no such operator like ":=" in 'C' language. The traditional parser do not allow errors (fuzziness) in input statements. Thus there is need to allow slightly incorrect inputs to a parser which can be parsed. Such kinds of errors (tiny mistakes) are discussed by Asveld in [4-10]. But we will take a bold step to handle them using fuzzy parsing.

To allow tiny mistakes in input statements to the parser, we can extend existing context free grammar that can model the grammatical errors. An input to a parser may be incorrect due to one or more of the following:

1. Crisp keywords replaced by fuzzy keywords.
2. Crisp operators replaced by fuzzy operators.
3. Token sequence in a particular statement may be changed.
4. Most frequently committed error.
5. Mismatch in balanced parenthesis

In the section below the fuzzy syntax analysis for the tiny language [3] is discussed using fuzzy lexical analyzer as its one of the module. Fuzzy lexical analyzer implemented for the tiny language is discussed in [11].

III. FUZZY SYNTAX ANALYSIS

We discuss implementation of fuzzy syntax analysis for the Tiny language in detail as follows:

1. Crisp keywords in the Tiny language is replaced by fuzzy keywords:

Consider the statement

```
read x;
```

Which is a crisp read statement in Tiny language, as it follows all the syntax rules defined for it. (Rules 1, 2, 3 and 7) in [3]. If we replace (crisp) read in above statement by its fuzzy read as defined in [11], then it may be in one of the following form:

a. rread x; //insertion error
b. red x; //deletion error
c. input x; //substitution error
d. eead x; //typing error
e. dear x; //letter sequencing error

The above statements (a) to (e) when given as input to the parser, the crisp parsing technique gives an error. Because though the flexibility is allowed at lexical analysis phase, it is not reflected in grammar rules of the Tiny language. Therefore CFG need to be extended to Fuzzy CFG to include the fuzzy tokens as follows:

1. Program → stmt-sequence
2. stmt-sequence → statement { ; statement }
3. statement $\xrightarrow{\mu}$ read- stmt | fread stmt
4. read- stmt → read identifier
5. fread- stmt $\xrightarrow{\mu}$ fread identifier

The number μ represents the degree of the fuzzy read statement. For example if the statement is

rreeaad x;

Then the token rreeaadd is a fuzzy token with membership degree 0.625 (calculated using algorithm 2.x) and the tokens – x and ; (semicolon) are crisp tokens (with membership degree 1.0). Thus the membership degree of whole statement will be computed as follows: $\min(\mu\ all\ tokens)$ i.e. $\min(0.625,1,1) = 0.625$. For all kind of fuzzy read statements, we have calculated the membership degree in following table:

Table 3.1: Computation of degree of membership for fuzzy statement

Sr. No.	Input statement	Fuzziness in reserved word	Membership computation	membership
(a)	rread x;	insertion error	min(0.825,1,1)	0.825
(b)	red x;	deletion error	min(0.75,1,1)	0.75
(c)	input x;	substitution error	min(0.6,1,1)	0.6
(d)	eead x;	typing error	min(0,75,1,1)	0.75
(e)	dear x;	letter sequencing error	min(0.8,1,1)	0.8

To accept minimum errors we have considered a threshold membership value for fuzzy tokens as 0.5. Below which the token will not be accepted as fuzzy token.
For eg. if input statement is

(f) rreeaaddd x;

Then membership of "rreeaaddd" is 0.378 which is less than threshold value 0.5. Therefore there will be a syntax error.
Since we allow errors for keywords, there are infinitely many fuzzy keywords. This ended with a small problem that can be resolved depending on the situation of appearance of the keyword in the statement. All classical keywords are treated as reserved words but not as variables. But in case of fuzzy parsing fuzzy keywords can also be treated as variables. For eg.

(g) read red;
(h) write red;
(i) red := red + 1;

To make this possible we declared a fuzzy flag variable, which is true at the beginning of every statement and made false for every next token in the statement except for reserved words. Using fuzzy flag in statements (g) and (h), red is treated as a variable name, but in case of (i), we need to extend CFG rule for assignment statement also.

2. Crisp operators in the Tiny language is replaced by fuzzy operators:

In Tiny language: = is an assignment operator which can be considered as = = or as = and vice versa. To allow this we extend the CFG by adding rule as below

L $\xrightarrow{\mu}$ = | == | :=

3. Token sequence in the Tiny language statements may be changed:

In Tiny language change in sequence of tokens will be considered only for two statements. One is "read" statement and other is "write" statement where expression part is a single identifier.

read- stmt → read identifier
write- stmt → write exp

These rules can be re-written as

read- stmt → identifier read
write- stmt → identifier write

for this we extend CFG to Fuzzy CFG.

4. Most frequently committed error:

Programmers sometimes miss to put semicolon at the end of a statement. This error is also considered in Tiny language and it is allowed to input statements without separating semicolon by adding the rule:

stmt-sequence → stmt-sequence { { ; } statement }

5. Balancing of parenthesis in the Tiny language:

In Tiny language existing rule is extended to allow the balancing of parenthesis: ( ] as follows:

factor → (exp) is extended as below
factor → (exp K
K →) |]

Considering all the above tiny errors we extended the existing CFG for Tiny language to Fuzzy CFG as follows:

Program → stmt-sequence
stmt-sequence → statement { { ; } statement }
statement $\overset{\mu}{\rightarrow}$ if-stmt | repeat-stmt | read-stmt |write-stmt | fzy-stmt | assign- stmt
if- stmt $\overset{\mu}{\rightarrow}$ if A
A $\overset{\mu}{\rightarrow}$ exp B stmt-sequence [C stmt-sequence] D
B $\overset{\mu}{\rightarrow}$ then | fuzzy_then
C $\overset{\mu}{\rightarrow}$ else | fuzzy_else
D $\overset{\mu}{\rightarrow}$ end | fuzzy_end
repeat- stmt → repeat rpt
rpt $\overset{\mu}{\rightarrow}$ stmt-sequence E exp
E $\overset{\mu}{\rightarrow}$ until | funtil
read- stmt → read identifier
write- stmt → write exp
fzy-stmt $\overset{\mu}{\rightarrow}$ fuzzy_read G | fuzzy_write F | fuzzy_if T | fuzzy_repeat H
G $\overset{\mu}{\rightarrow}$ fasg | identifier
fasg → read | write | L exp
L $\overset{\mu}{\rightarrow}$ = | = = | :=
F $\overset{\mu}{\rightarrow}$ fasg | exp
T $\overset{\mu}{\rightarrow}$ fasg | A
H $\overset{\mu}{\rightarrow}$ fasg | rpt
assign- stmt $\overset{\mu}{\rightarrow}$ identifier fasg
exp $\overset{\mu}{\rightarrow}$ simple-exp [comparison-op simple-exp]
comparison-op $\overset{\mu}{\rightarrow}$ < | L
simple-exp → term {addop term }
addop → + | -
term → factor{ mulop factor }
mulop → * | /
factor → (exp K | number | rd
K →) |]

Sample programs:
Using the extended CFG for the tiny language as above few sample programs are written:

```
rread x ;
iif  0 < x  thhen
        fact := 1 ;
        rpeat
                fact = fact * x ;
                x := x - 1
        untl x = 0 ;
   writte x
eend
```

```
read x ;
if  0 < x then
        fact := 1 ;
repeat
        fact := fact * x ;
        x := x - 1
until x = 0 ;
write fact
end
```

```
read red
if  0 < red then
        rite := 1
repeat
        rite := rite * red
        red := red - 1
until red = = 0
write rite
end
```

In the example above first program is using fuzzy keywords. In second program same program with crisp keywords is given. But in third program fuzzy keywords (red and rite) are used as variable names and semicolon is missing as statement separator. There are many possible fuzzy programs for single crisp program with acceptable tiny mistakes. Same syntax tree is generated for all three programs above and will remain same for all possible fuzzy programs for a given crisp program for factorial computation. Thus it is many to one relation from input program to parse tree generated as a result of parsing.

IV. CONCLUSION

Syntax analyzer module of compiler is extended to accept fuzzy statements using concepts of fuzzy context free grammar and fuzzy parsing methods. In this paper only top-down parsing method is taken into consideration. In top-down parsing specifically recursive descent parsing is considered for implementation. The implementation can also be carried out using bottom-up parsing methods and the performance can be compared with top-down parsing methods.

REFERENCES

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

[2] Aho, Sethi, Ullman Compilers – Principles, Techniques, and Tools, Pearson Education, 2005.

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

[4] Asveld P.R.J., A fuzzy approach to erroneous inputs in context-free language recognition, In proceeding of the 4th International Workshop on Parsing Technologies, Prague/Karlovy Vary, Czech Republic. pp. 14-25, 1995.

[5] Asveld P. R. J., A bibliography on fuzzy automata, grammars and languages, Bulletin of the European Association for Theoretical Computer Science 58, pp. 187-196,1996.

[6] Asveld P.R.J., Towards robustness in parsing – Fuzzifying context-free language recognition, In: J. Dassow, G. Rozenberg and A. Salomaa (eds.), Developments in Language Theory II – At the Crossroads of Mathematics, Computer Science and Biology, World Scientific, Singapore, pp. 443-453, 1996

[7] Asveld P. R. J., Controlled fuzzy parallel rewriting, In: Gh. Păun, A. Salomaa (Eds.), New Trends in Formal Languages—Control, Cooperation, and Combinatorics, Lecture Notes in Computer Science, Vol. 1218, Springer, Berlin, pp. 49–70. 1997.

[8] Asveld P.R.J., Algebraic aspects of families of fuzzy languages, Theoret. Comput. Sci., 293(1-2), pp. 417–445, 2003.

[9] Asveld P.R.J., Fuzzy context free languages – Part 1, Generalised fuzzy context free grammars, Theoretical Computer Science, 347(1–2), pp. 167–190, 2005

[10] Asveld P.R.J., Fuzzy context free languages – Part 2, Recognition and parsing algorithms, Theoretical Computer Science, 347(1–2), pp.191–213, 2005.

[11] Vaishali Bhosale, Dr. S. R. Chaudhari, Fuzzy Lexical Analyzer: Design and Implementation, International Journal of Computer Applications, 0975-8887, vol.123-No.11, Aug.2015, p.p.1-7