iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 2 **Issue:** XI **Month of publication:** November 2014

DOI:

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)

A Review on LL (1) Parser

Faraah M. Dabhoiwala
Information Technology, Gujarat Technological University
Vadodara Institute of Engineering, Halol Road, Kotambi, Vadodara, Gujarat, India

Abstract—In any programming language, parsing is performed after scanning each and every lexical unit of the program. Lexical unit of PL domain is any keyword, identifier, constant, operator, etc. Once all the units have been identified parsing is performed. Parser checks the syntax of each and statement of PL. If the syntax is found correct, the parser generates parse trees. LL (1) parsing is a top down parsing technique. LL (1) parser constructs table and based on the entry in the table it chooses the production on right hand side of the non-terminal. Error is generated if the entry in the table is found blank. If the entry is blank the parser has no option to replace the Non terminal with the production rule which means the given string is invalid.

Keywords— top down parsing, LL (1) parser, recursive grammar, ambiguous grammar

I. INTRODUCTION

Parsing [1] is performed during syntax analysis phase. Syntax analysis is carried out by parser to check the validity of source statement. It checks whether the syntax of the statement is correct or not and generates parse tree in case of valid statement and it causes an error if the statement is found invalid.

Parsing can be performed in two ways:
Top Down Parsing
Bottom Up Parsing
LL (1) parsing is a top down parsing technique.

II. TOP DOWN PARSING

Consider a grammar G, to perform top-down parsing it must be possible to derive the string β from start symbol S through a sequence of derivation. S→….→…→α

Consider the grammar

(1) S→AC
(2 A→aA| Є
(3) C→b|bc

Grammar 1.

If we start deriving the string aaab through the sequence of derivation from start symbol S then top down parsing is said to be successful.

S→AC	using production 1
→aAC	using production 2
→aaAC	using production 2
→aaaAC	using production 2
→aaa ЄC	using production 2
→aaaC	
→aaab	using production 3

A. Drawback with the Traditional Top down Parsing Approach

In traditional top down parsing, parser derives all the possible strings from the given start symbol. This increases the overhead as parser generates all the strings and then matches each string with the one that needs to be derived. So the overall time and overhead increases.

III. LL (1) PARSING

LL (1) parsing is a predictive parsing technique. In LL(1) parsing the first L means scanning of string takes place from left to right and second L means that during parsing we choose leftmost non terminal for derivation [3]. The (1) in LL(1) means look-ahead symbol.

For a simple mathematical calculation grammar is:

E→E+T|E-T|T
T→T*F|T/F/F
F→F^P|P
P→(E)|i

Grammar 2.

A. Left Recursive grammar

LL(1) parsing cannot be performed in case of grammar which is left recursive [1]. A grammar is said to be left recursive if the non-terminal that appears on L.H.S. of production is the first non-terminal on R.H.S. too. Consider the production rule E→E+T|T. This grammar is said to be left recursive grammar because the parser might enter into infinite loop if it keeps on substituting E by E+T as E is the first non-terminal on RHS. The process will take place as shown below

E→E+T
→E+T+T
→E+T+T+T
→E+T+T+T….

1 solution of this problem is to remove left recursion and to make the grammar right recursive [2].

E→T+E|T

But this is not possible each & every time. For example, F→F^P won't give same result as F→P ^F. So there is a possibility of generating erroneous grammar rules if we try to make it right recursive. So in order to remove left recursion following rules must be applied.

General Rule:

Consider a grammar:

X→Xα1|Xα2|...|Xαn|β1|β2|…|βm

After removing left recursion

X→ β1X'| β2X'|…| βmX'

X'→ α1 X'| α2 X'|…|αn X'| Є

Applying above rules to remove left recursion on grammar (2) we get the grammar rules as follows [2]

E→TE'
E'→+TE'|-TE'| Є
T→FT'
T'→*FT'|/FT'| Є
F→PF'
F'→^PF'| Є
P→(E)|i

Grammar3

To apply LL (1) parsing FIRST and FOLLOW of each and every non terminal must be calculated and then a table must be constructed [2].

1) Rules to calculate FIRST:

1. For a production A →αB, to calculate FIRST(A) if α is a terminal then put α in FIRST(A) and if α is a non-terminal then look for production of α. If α→…→…→**β** through a sequence of derivation where is a terminal then put in FIRST(A).
2. If A→ Є then put Є in FIRST(A).

2) Rules to calculate FOLLOW:

1. Put $ in FOLLOW(A) where A is a start symbol and $ is the input right end marker.
2. If there is a production A→αBβ then put everything in FIRST(β) in FOLLOW(B) except Є and if FIRST(β) contains Є then everything in FOLLOW(A) is in FOLLOW(B).
3. If there is a production A→αB then everything in FOLLOW(A) is in FOLLOW(B).

3) Rules to construct table:

1. For each terminal in FIRST(A) except Є put A→α in the table where A is row and α represents column.

2. If FIRST(A) contains Є then check FOLLOW(A) and put A→ Є in the table where A is row and every column that is represented by terminal or $ in FOLLOW(A).

B. StringDerivation

In order to derive a string the parser starts deriving from start symbol and applies continuous check on the string. It chooses the production from the table based on which terminal needs to be derived from the CSF.

Consider the same Grammar3 for mathematical calculation metioned earlier.

Applying rules to calculate FIRST and FOLLOW

FIRST(E)={i,(}
FIRST(E')={+,-, Є }
FIRST(T)={i,(}
FIRST(T')={*,/, Є}
FIRST(F)={i,(}
FIRST(F')={^, Є }
FIRST(P)={i,(}
FOLLOW(E)={$,)}
FOLLOW(E')= {$,)}
FOLLOW(T)={+,-,$,)}
FOLLOW(T')= {+,-,$,)}
FOLLOW(F)={*,/,+,-,$,)}
FOLLOW(F')= {*,/,+,-,$,)}
FOLLOW(P)={^ ,*,/,+,-,$,)}

Construct table as per the rules mentioned above.[1]

TABLE 1

NTs	**Terminals**								
	i	**+**	**-**	*****	**/**	**^**	**(**	**)**	**$**
E	E→TE'						E→TE'		
E'		E'→+TE'	E'→-TE'					E'→ Є	E'→ Є
T	T→FT'						T→FT'		
T'		T'→ Є	T'→ Є	T'→*FT'	T'→/FT'			T'→ Є	T'→ Є
F	F→PF'						F→PF'		
F'		F'→ Є	F'→ Є	F'→ Є	F'→ Є	F'→^PF'		F'→ Є	F'→ Є
P	P→i						P→(E)		

Now suppose if it is required to derive the string $i+i*(i-i)$ then the parser will choose the productions systematically from the table. Let the Current Sentential Form(CSF) be E

CSF	**Productions Used**
E	
TE'	E→TE'
$FT'E'$	T→FT'
$PF'T'E'$	F→PF'
$iF'T'E'$	P→i
$iT'E'$	F'→ Є
iE'	T'→ Є
$i+TE'$	E'→+TE'

i+FT'E'$	T→FT'
i+PF'T'E'$	F→PF'
i+iF'T'E'$	P→i
i+iT'E'$	F'→ Є
i+i*FT'E'$	T'→*FT'
i+i*PF'T'E'$	F→PF'
i+i*(E)F'T'E'$	P→(E)
i+i*(TE')F'T'E'$	E→TE'
i+i*(FT'E')F'T'E'$	T→FT'
i+i*(PF'T'E')F'T'E'$	F→PF'
i+i*(iF'T'E')F'T'E'$	P→i
i+i*(iT'E')F'T'E'$	F'→ Є
i+i*(iE')F'T'E'$	T'→ Є
i+i*(i-TE')F'T'E'$	E'→-TE'
i+i*(i-FT'E')F'T'E'$	T→FT'
i+i*(i-PF'T'E')F'T'E'$	F→PF'
i+i*(i-iF'T'E')F'T'E'$	P→i
i+i*(i-iT'E')F'T'E'$	F'→ Є
i+i*(i-iE')F'T'E'$	T'→ Є
i+i*(i-i)F'T'E'$	E'→ Є
i+i*(i-i)T'E'$	F'→ Є
i+i*(i-i)E'$	T'→ Є
i+i*(i-i) $	E'→ Є

Hence the string i+i*(i-i) has been derived by choosing the productions systematically from the entries in the table. The above grammar leads to successful parse because the grammar is unambiguous and there is no left recursion in the grammar

C. Error Detection

Suppose if an invalid string is entered into LL (1) parser, in that case while deriving the string the terminal which is to be matched, its entry in the table is found to be empty [2]. The empty entry suggests that the string is invalid. Empty entry means that the given string can't be derived from the given grammar.

IV. CONCLUSION

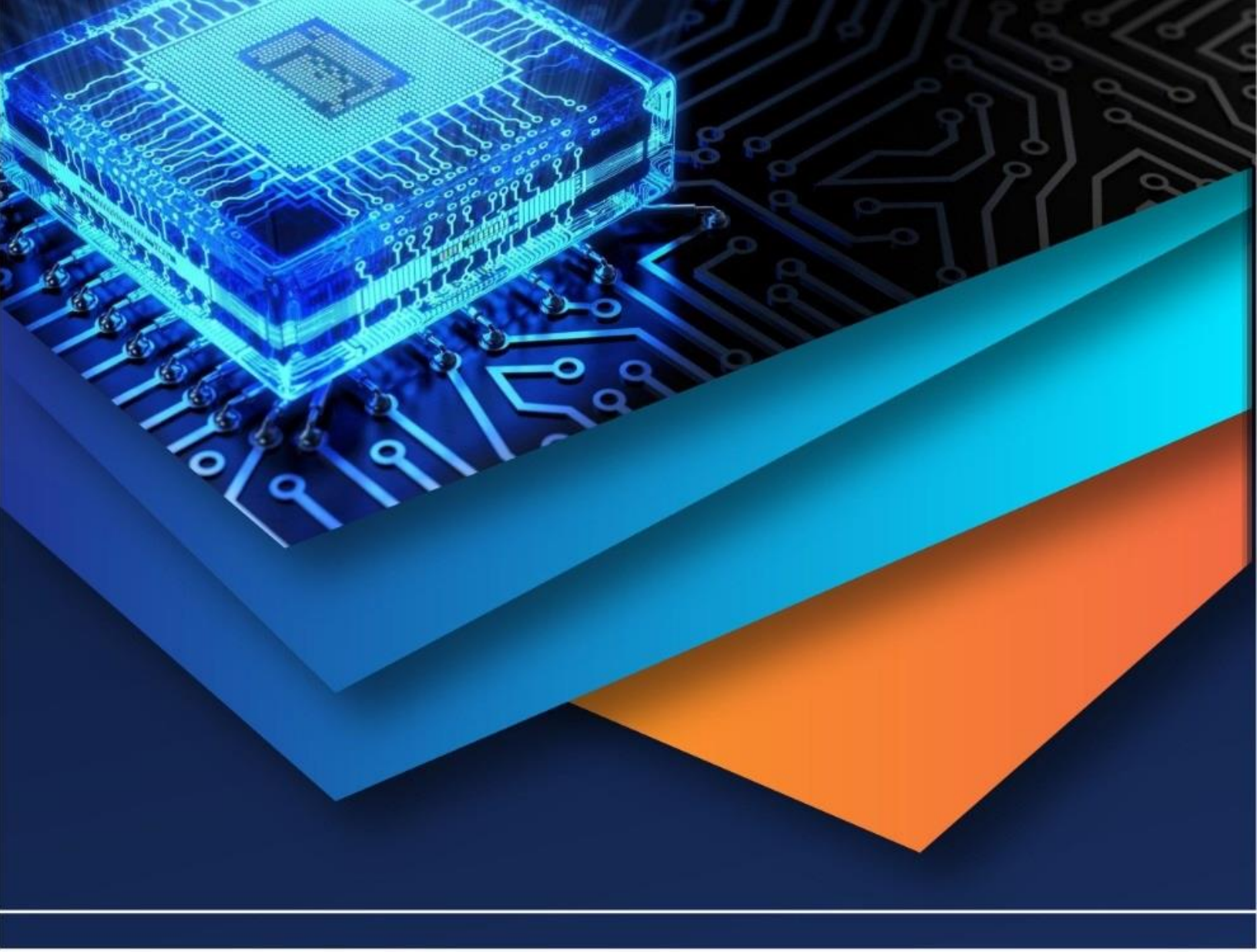
In traditional Top-Down parsing approach, the parser derives all possible string that can be derived from given grammar and at the end matches each string with the one that needs to be derived. LL (1) parser constructs the table and chooses the productions from the table hence the overall process of deriving the string takes lesser time as compared to naïve top down parsing approach. It is possible to detect the error at quite early stage once the entry in the table is found empty. Hence the precise point of error can be also be known in LL(1) parser.

V. ACKNOWLEDGMENT

I would like to thank my Parents for their endless support and Mr. Rahil Barafwala, for constantly encouraging me to work on research paper and this work is the result of the same.

REFERENCES

[1] Dhamdhere D M, Systems Programming and Operating Systems.
[2] Aho A. V., Sethi R., Ulman J.D., Compilers Principles, Techniques and Tools.
[3]Albrecht Wöß, Markus Löberbauer, Hanspeter Mössenböck,(2003),LL(1)Conflict Resolution in a Recursive Decent Parser, Modular Programming Languages,(2789), 192-201.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY