J. Appl. Environ. Biol. Sci., 7(5)64-76, 2017

ISSN: 2090-4274
Journal of Applied Environmental and Biological Sciences
www.textroad.com

The Impact of Terminal Prefixing on LL(1) Parsing

Muhammad Shumail Naveed

Department of Computer Science, University of Balochistan, Pakistan

Received: December 10, 2016
Accepted: February 22, 2017

ABSTRACT

In this paper a terminal prefixing is presented as a normal form of context-free grammar and applied on a LL(1) grammar. Its impact on the size of derivations, magnitude of parse trees and particularly on the efficiency of LL(1) parsing is analyzed. Results show that terminal prefixing has a significant impact on size of parse trees and similarly on the efficiency of LL(1) parser. The discussion included in this paper is helpful in the advancement of other parsers and also for the anatomization of topical normal forms.
KEYWORDS: parsing, context-free grammar, LL(1) parsing, Greibach normal form, ambiguity

1. INTRODUCTION

Parsing is the process of arranging a linear representation according to a particular grammar [1], and parser is a program that accomplished this process. Parsing has a great importance and extensive history of academic study yet it is not a solved problem [2], and therefore researchers have devoted decades analyzing to develop efficient parsing methods. Parsing has diverse applications especially in the area of compiler construction [3], natural language processing [4,5] and automatic speech recognition [6].

In classical compilers the parsing is a second phase and deals with the verification of syntax and generation of parse tree which defines the structure of a program. In contemporary compilers the top-down and bottom-up are the common parsing techniques.

The top-down parsing starts the parsing from the root of the parse tree and determine from the input token stream how to develop the parse tree. Top-down parser perform this duty by using the information of the lookahead tokens and of the productions in the grammar. Furthermore, it scan the incoming tokens from left to right.

Top-down parsing can be performed with a set of recursive functions or through table-driven approach which is easy to construct and modify [7]. Top-down parser can only parse a restricted class of a context-free grammar. Fundamentally, the context-free grammar (CFG) is one of a powerful concept of computational linguistics, programming languages and computer science.

In computer science the context-free grammar is generally used to describe the syntax of programming languages [8]. It is also very useful in the description of restricted natural language based programming languages [9, 10]. Formally the context-free grammar is a four tuple (N,T,P,S), where *N* is a set of nonterminals. *T* is a set of terminals. Nonterminals and terminals are always disjoint. *P* is set of productions or rules, in which each rule takes the following form [11]:

$$B \rightarrow \alpha,$$

where $B \in N$ and $\alpha \in (N \cup T)^*$. *S* is the start symbol and a subset of *N*.

The context-free grammar is usually normalized before using in any real solution and parsing is no exception. The normalization usually restrict the structure of a grammar either by restricting the size of production or the arrangement of symbols in the productions.

A restricted class of context-free grammar is used in top-down parsing. The top-down parsing has two main types: recursive-descent parsing and LL(1) parsing. The recursive-descent parsing analyze the input by using the recursive procedures where each such procedure represents one of the production of its context-free grammar. On the other hand, the LL(1) parsing generally does not require backtracking during building a parse tree and commonly used in compilers.

LL(1) parsing is one of a simple and viable choice for syntax analysis which is a second phase of a compiler. Syntax analyzer is one of a time consuming phase of a compiler and consequently the development of efficient parser remains an active area of research in computer science and computational linguistics.

This paper introduced a terminal prefixing which is a more general form of Greibach normal form and apply this terminal prefixing to simplify the LL(1) grammar and determine its impact on derivation, parse tree and LL(1) parsing.

The paper is structured as follows. In section 2 the LL(1) parsing is described and the major work conducted in the improvement of LL parsing is included in section 3. In section 4 the terminal prefixing is introduced. The initial evaluation and results are included in section 5. Finally, section 6 describes the conclusion.

***Corresponding Author**: Dr. Muhammad Shumail Naveed, Department of Computer Science, University of Balochistan, Quetta, Pakistan. Email: mshumailn@gmail.com

2. LL(1) Parsing

The LL(1) parsing is a top-down, non-recursive predictive parsing technique which requires no backtracking and constructed over the class of grammars called LL(1) grammars. The fundamental concepts of LL(1) grammars were formally presented by Lewis and Stearns [12], and the attributes of LL(1) grammar were refined by Rosenkrantz and Stearnsc [13]. LL(1) parsing used an explicit stack to parse the sentence. It is also used in the definition of other parsing techniques especially in a bottom-up parsing algorithms [8]. The first "L" in LL(1) indicates that input is processed from left to right. The second "L" indicates that it performs the left most derivation for the input, and "1" indicates that in each step only one symbol of input is used as a lookahead to predict the parsing action. In LL parsing the lookahead symbols may be the multiple symbols. LL parsing is the foundation of a contemporary parser generator ANother Tool for Language Recognition (ANTLR) [14]. LL(1) parsing is usually implemented through a table called the parsing table. The construction of a LL(1) parsing table involves the computing of two functions associated with the LL(1) grammar. These two functions, FIRST and FOLLLOW facilitate to fill up the entries of LL(1) parsing table. LL(1) grammars are the restricted class of context-free grammars and therefore LL(1) parsing is inherently a restricted class of parsing technique. There are several limitations of LL(1) parsing. First, it cannot handle the ambiguous grammar. Similarly it cannot handle the left recursive grammar and it also can't handle a grammar containing the conflict.

If G is a context-free grammar and L(G) is a language defined over G, then grammar G is said to be ambiguous if there is at least one string in L(G) for which G generates multiple parse trees [15]. Ambiguity is a common characteristic of natural language; however in programming language it is necessary to remove the ambiguity if possible. Concretely there is no well defined algorithm for the identification and removal of ambiguity, nevertheless in most of the situations it is possible to identify and remove the ambiguity. Principally any language L can be defined by different grammars. If every grammar that defines L is ambiguous grammar, then the said language is known as inherently ambiguous.

A grammar G is said to be left recursive, if it contains at least one left recursive production. A left recursive is one that takes the following form:

$$X \rightarrow X\,\alpha$$
$$X \rightarrow \beta$$

Where X is a nonterminal and α and β are the strings of nonterminals and terminals and β does not begin with X. The production having left recursion can be handled by rewriting that production into two rules: one that defines β first and another one that defines the repetition of α:

$$X \rightarrow \beta\,X'$$
$$X' \rightarrow \alpha\,X' \mid \varepsilon$$

Conflict is an undesirable condition for LL(1) parser, that occur when multiple grammar productions choices have a common prefix:

$$X \rightarrow \beta\,\alpha$$
$$X \rightarrow \beta\,\gamma$$

This problem can be solved by left factoring. Through left factoring the β is factored on the left and rule is rewritten as:

$$X \rightarrow \beta\,X'$$
$$X' \rightarrow \alpha$$
$$X' \rightarrow \gamma$$

3. Previous Work

For decades, experts and scientists have worked towards expanding the recognition strength and the performance of LL parsing. Parr et al. [16] introduced the ALL(*) parsing which integrates the plainness, effectiveness and predictability of traditional LL(k) parsers with the ability of a GLR-type method to take parsing decisions. The significant novelty is to move analysis to parse-time, which allows ALL(*) deal context-free grammar without any left recursion. Theoretically ALL(*) is $O(n^4)$ but executes linearly on grammar used in practice.

Shilling [17] introduced an efficient incremental parsing algorithm. It is devised for the language-based editors and implemented in Fred language-based editor. It can parse input at intervals of extremely tiny granularity and bound the amount of incremental parsing required when alterations are perform internal to the editing buffer.

In [18], the notion of LLLR is introduced. An LLLR parser generates the left parse of the input in a linear time and can be built for any LL(k) grammar. It works as an LL(k) and used as a foundation for a syntax-directed translation. LL(k) is the backbone of the LLLR(k) parser and generate semantic action by means of top-down approach just similar to the canonical LR(k) parser. LLLR parsing is similar to LL(*) parsing but does not perform backtracking and utilize LR(k) parser to manage the LL(k) conflicts [19].

The LL(k) parser of an LL(k) grammar is fundamentally a single state deterministic push-down automaton that performs left parse of the input string. The LL(k) parsing is evolved as duple of LR (k) parsing. LL(k) parser may be considered as a push-down automaton. In [20] the equivalence relation that yields the canonical LL (k) are considered, and a new technique for generating the canonical parser is introduced.

The parsers for LL(k) grammars are usually the deterministic pushdown automata that perform leftmost derivation for the given input string. Ukkonen [21] presented a family of LL(2) grammars and proved that all parsers developed over LL(2) grammars have exponential size. It is possible to obtain an extended parser for the LL(*k*) grammar by adding to a parser the possibility to process a constant number of pointers that point to locations in the generated part of the leftmost derivation and to vicissitude the output in such positions [22].
In [23] a threaded parse is introduced tree by introducing threaded link in parse trees, and set forth supplementary distance entry in LL parse tables, resultantly called augmented LL parse tables. These quantum leaps are afterwards used to develop a potent incremental LL parser.
In [24] dynamic context-free grammars, as an expansion of classical context-free grammars is presented and identified that deterministic top-down parsing of dynamic context-free grammars is equally robust as deterministic bottom-up parsing of context-free grammars, and are adequate to parse all deterministic context-free languages if the lookahead symbol is one. The dynamic LL(k) grammars integrate the universality of LR(k) parsing with the benefits of LL(k) parsing. However the construction of dynamic LL(k) grammars is more difficult than static LL(k) grammar but the size of dynamic LL(k) grammar is expressively smaller than an equivalent LL(k) grammar.
Bertsch and Nederhof in [25] presented a comprehensive study on the association between the size of LL(k) grammars and the size *k* of lookahead, and conferred result pertaining frugality of description of languages using LL(k) grammars if *k* varies.
Vagner and Melichar [26] introduced the parallel deterministic LL parser and defined a class of LLP grammars appropriate for deterministic parallel LL parsing. The class of LLP grammar is a proper subset of LL grammars, and encompassed significant languages, like the arithmetic expression language.
For non left-recursive grammar, a visualized LL parsing is developed which is based on relation matrix, that makes LL parsing more efficacious for large characters based grammars [27].
The GLL parsing is presented in [28], which is based on recursive-descent paradigm and capabilities to handle left recursive grammar. GLL parsers are completely general, worst-case cubic parsers, useful and easy to use for the debugging of grammar. An algorithm for constructing GLL parsers which developed an SPPF illustration of the derivations of the input is also defined. The GLL parsing algorithm is able to recognize any context free grammar, including the ambiguous grammar [29].

4. METHOD & DISCUSSION

If *L* is a language defined over the LL(1) grammar G_1 and parsed by a conventional LL(1) parser then there exist an equivalent LL(1) grammar G_2 in terminal prefixed form which can be recognized by a very similar LL(1) parser probably with better performance. The steps required by terminal prefixed LL(1) parser to parse the LL(1) grammar are defined as follows:

1. Conversion of standard LL(1) grammar into an equivalent grammar in terminal prefixed form.
2. Removal of useless productions (if exist).
3. Calculation of FIRST set for each terminal and nonterminal of a grammar.
4. Calculation of FOLLOW set for each nonterminal of a grammar
5. Construction of a parsing table.

4.1 Terminal Prefixing

Conventionally the amelioration of any parsing algorithm is performed either by increasing its grammar recognition capability or by reducing the number of steps required during parsing. For later case of enhancement the pertinent grammar of a parsing algorithm is usually simplified and normalized in different ways. Greibach normal form and Chomsky normal form are two common normal forms of context-free grammar.
Greibach normal form (GNF) was introduced for context-free grammar [30]. Greibach normal form is indispensable for several results in the domain of context-free languages. Virtual applications of Greibach normal form includes, the conversion of a context-free grammar into an ε-free pushdown automaton or the construction of efficient parser. A context-free grammar G = (N,T,P,S) is said to be in Greibach normal form, if all the productions satisfy one of the following forms:

$$X \rightarrow a\,\alpha$$
$$X \rightarrow \varepsilon$$

Where, $a \in T$ and $\alpha \in N^*$. Greibach normal form limits the shape of productions of a context-free grammar in a way that with every step of derivation the terminal is generated from left to right expending terminal prefix by one symbol in every step.
Terminal prefixed form which is introduced in this paper can be viewed as a general form of Greibach normal form. A grammar is in terminal prefixed form, if all the productions satisfy one of the following forms:

$$X \rightarrow a\,\alpha$$
$$X \rightarrow \varepsilon$$

Where, $a \in T$ and $\alpha \in (N \cup T)^*$. Unlike Greibach normal form the α in terminal prefixed form is a sequence of terminal symbols and nonterminal symbols. The LL(1) grammar never contains ambiguity, left recursion and conflict and therefore any LL(1) grammar can be converted into an equivalent terminal prefixed form.

The first and the foremost step in the definition of the proposed method is to convert the LL(1) grammar into the terminal prefixed form. The proof of this part will be by constructive algorithm. This means that an algorithm will be defined that starts out with LL(1) grammar and ends up with a terminal prefixed grammar. This algorithm satisfies two criteria. It works for every conceivable LL(1) and it must guarantee to complete its process in a finite number of steps.

Three cases are encountered during the transformation of LL(1) grammar productions in an equivalent terminal prefixed form:

Case 1:

If a production in P_1 takes the following form:

$X \rightarrow a\ \alpha$

Where *a* is a terminal and $\alpha = (N \cup T)^*$, then underlying production is already in the terminal prefixed form (1).

Consider the following fragment of productions:

$X \rightarrow a\ Y$
$Y \rightarrow b\ Y$
$Y \rightarrow c$

In above productions, the body of every production starts with the a terminal, then the said productions are already in terminal prefixed form.

Case 2:

If a production in P_1 takes the following form:

$A \rightarrow \varepsilon,$

and *A* further produces nothing, then removes that production, and eliminate *A* from the body of other productions.

Consider the following fragment of productions:

$B \rightarrow Ac$
$A \rightarrow \varepsilon$

After the transformation of the above productions, we get the following terminal prefixed production:

$B \rightarrow c$

However, If a production in P_1 takes the following form:

$A \rightarrow \varepsilon,$

And *A*, further produces something; then removes $A \rightarrow \varepsilon$, and turn by turn replace every occurrence of nonterminal *A* on the right side of productions by the symbols appearing on the right side of *A* including the ε (the body of *A*).

Consider the following fragment of productions:

$B \rightarrow Ac$
$A \rightarrow \varepsilon$
$A \rightarrow aA$
$A \rightarrow d$

After the transformation of above productions, we get the following productions:

$B \rightarrow c$
$B \rightarrow aAc$
$B \rightarrow dc$
$A \rightarrow aA$
$A \rightarrow d$

In that situation, all the productions are in terminal prefixed form, however, if the resulting productions are not in terminal prefixed form, then same procedure will be recursively applied by using the relevant methods (in Case 1 or Case 2) until the productions will be transformed into terminal prefixed form.

Case 3.

If the productions in P_1 take the following form:

$A \rightarrow x\alpha$, where $x \in N$, and $\alpha = (N \cup T)^*$, then the first step is the selection the production from the multiple productions. Although the productions can be selected in any arbitrary order, but it is recommended to select the productions in a bottom up fashion.

Suppose $A \rightarrow B\alpha_1$, $B \rightarrow C\alpha_2$, and $C \rightarrow c\alpha_3$, where A, B and $C \in N$, α_1, α_2 and $\alpha_3 \in (N \cup T)^*$, and *c* is a terminal, then virtually it is allowed to select any production for transformation. However in a bottom up manner, we strive to select a production, whose suffix of the body immediately generates the string starting with a terminal, in that case $B \rightarrow C\alpha_2$ is selected first. After selection, the suffix of a body is replaced with its generating string(s), in next step $A \rightarrow B\alpha_1$ is selected, and transformed in the same way as $B \rightarrow C\alpha_2$, finally and we get the productions in terminal prefixed form.

Consider the following fragment of productions:

A → BcA
B → Cad
C → c

Since the third production is already in terminal prefixed form, so the production B → Cad, will be selected first. After its transformation A → BcA will be selected and transformed by employing the above defined method. The equivalent terminal prefixed productions for the above productions are:

A → cadcA
B → cad
C → c

The same method will be applied, if *A*, *B* and *C* define multiple productions.

4.2 Removal of useless productions

The second step in the construction of terminal prefixed LL(1) parser is to identify and remove the useless productions from the grammar. A production in a terminal prefixed grammar is called useless, if it can never take part in any derivation. There are different forms of useless production.

Case 1: Consider the following grammar:

A → a *A* b
A → *B*
A → ε
B → c *B*

The production A → *B* never plays any role, because no terminal string can be generated by *B*.

Case 2: Consider the following grammar:

A → a *A* b
A → ε
B → b *B*
B → ε

Although *B* derives a terminal, but it is useless, since it cannot be reached from the start symbol of the grammar. Dependency graph is useful for identifying this type of useless productions.

4.3 Construction of Parsing Table

Like conventional LL(1) parsing, the parsing with terminal prefixed LL(1) grammar involves the computation of FIRST and FOLLOW set and the methods for computing these sets are quite similar to conventional methods.

Let *A* be a symbol (a terminal or nonterminal) of a grammar written in terminal prefixed form. Then the FIRST(A) contains terminals and may be ε, is defined by applying the following rules until no more entries can be inserted to some FIRST(A):

1. If *A* is a terminal symbol, then FIRST(A) = { *A* }.
2. If A is a nonterminal and produces ε, then add ε to FIRST(A).
3. If A is a nonterminal of a grammar, then for each rule of the form A → $\alpha\beta_1 \ldots \beta_n$, α must be a terminal symbol and it will be added to FIRST(A). Unlike conventional LL(1) parsing, there is no need to check the remaining symbols.

The algorithm for computing the FOLLOW set of nonterminals in the terminal prefixed grammar is similar to the standard LL(1) parsing. If *A* is a nonterminal, then the FOLLOW(A), comprises of terminals, and possibly $ is defined by applying the following rules until nothing can be added to FOLLOW(A):

1. If *A* is a distinguished symbol, then place $ in FOLLOW(*A*). Whereas $ is the input right end indicator.
2. If there is a production rule *B* → α*A*β, then add all symbol in FIRST (β) - ε in FOLLOW(A).
3. If there is a production *B* → α*A*, or a production A → α*A*β such that ε is in the FIRST(β), then everything in FOLLOW(B) is in FOLLOW(A) .

The algorithm for constructing the parsing table from the terminal prefixed LL(1) grammar is similar to the standard LL(1) parsing and described below:

1. For every production B → ∝ of the terminal prefixed grammar, perform the steps 2 and 3.
2. For all terminal 'b' exclusive of ε in FIRST(∝), add ∝ to PARSINGTABLE[B,b].
3. If ε ∈ FIRST(∝), include B → ∝ to PARSINGTABLE[B,c] for all terminal *c* in FOLLOW(B). If ε ∈ FIRST(∝), and $ ∈ FOLLOW(B), add B → ∝ to PARSINGTABLE[B,$].
4. Define every undefined column of a PARSINGTABLE be an error.

5. EVALUATION AND RESULTS

The conventional LL(1) parser does not impose any condition on the prefix of the right side of any production. So the verification of a single terminal may involves multiple steps. However, in case of terminal prefixing the LL(1) parser match at least one terminal in each step and similarly it generates a compact parse tree because it restrict the expansion of left subtrees.

In order to identify the real impact of terminal prefixing on LL(1), consider a universal example of arithmetic expression grammar which is included in almost every material of the theory of automata and compiler construction [3, 8]. For conciseness the nonterminals <expr>, <expr'>, <term>, <term'> and <factor> are abbreviated as E, E', T, T' and F respectively. So in concise form the arithmetic expression grammar would be written as:

E	→	TE'	(1)
E'	→	+E'	(2)
E'	→	-TE'	(3)
E'	→	ε	(4)
T	→	FT'	(5)
T'	→	*FT'	(6)
T'	→	/FT'	(7)
T'	→	ε	(8)
F	→	ID	(9)
F	→	(E)	(10)

Now we convert the above arithmetic expression grammar into a terminal prefixed form. As per rules, the productions 2,3,4,6,7,8, 9 and 10 are already in the desired form, and only the productions 1 and 5 are to be converted. According to the defined rules, the production 5 is initially selected:

T → FT'

Since *F* produces ID and (E), so for the transformation of production 5, the *F* will be replaced with ID and (E), and obtained the following productions which are in terminal prefixed form:

T → ID T'
T → (E) T'

Now the production 1 is selected:

E → TE'

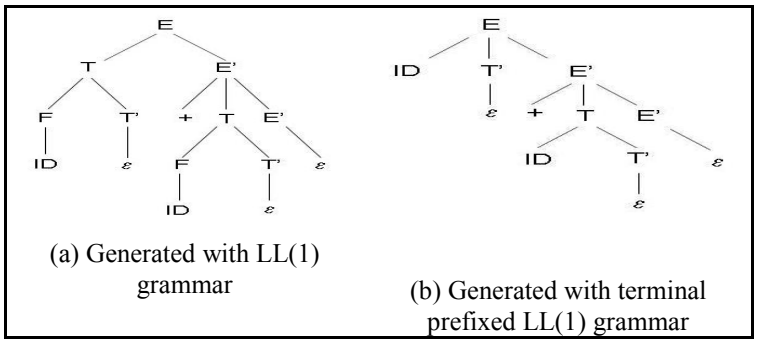
Since T produces ID T' and (E) T', so for the transformation of production 1, T will be replaced with ID T' and (E) T', and obtained the following productions which are in terminal prefixed form:

E → ID T' E'
E → (E) T'E'

Finally we obtained the terminal prefixed grammar:

E	→	ID T' E'
E	→	(E) T'E'
E'	→	+T E'
E'	→	-T E'
E'	→	ε
T	→	ID T'
T	→	(E) T'
T'	→	*FT'
T'	→	/FT'
T'	→	ε
F	→	ID
F	→	(E)

The terminal prefixed LL(1) grammar is intrinsically more coherent then the conventional LL(1) grammar as it generates at least one terminal in each step. As an illustration consider the parse trees of "ID + ID".

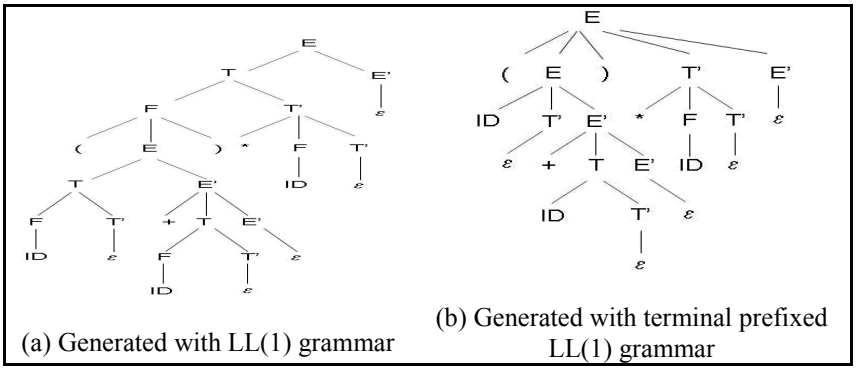


(a) Generated with LL(1) grammar

(b) Generated with terminal prefixed LL(1) grammar

Fig. 1: Parse tree of ID + ID

The parse tree generated by the conventional LL(1) parser is shown in Fig. 1(a) contains 15 nodes (including the ε-node), in which 9 are internal nodes, whereas the parse tree (shown in Fig 1(b)) generated by the terminal prefixed parser contains 11 nodes in which 6 are internal nodes. This indicates that the parse tree generated with terminal prefixed LL(1) grammar is more compact than the conventional LL(1) grammar and therefore the terminal prefixing has a significant impact on the size of parse trees. Now consider Fig. 2 for the parse trees of "(ID+ID)*ID".

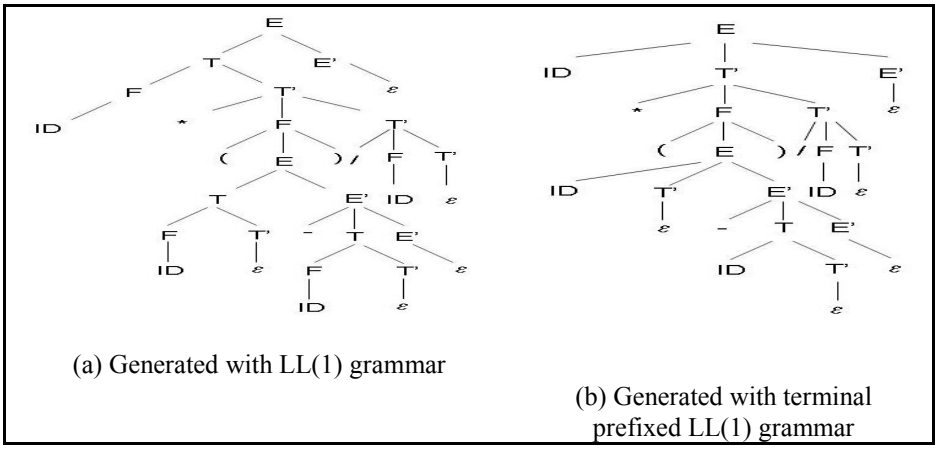


(a) Generated with LL(1) grammar

(b) Generated with terminal prefixed LL(1) grammar

Fig. 2:Parse tree of (ID + ID) * ID

Fig. 2 shows that the parse tree generated with terminal prefixed grammar contains 11 nodes whereas the parse tree generated with conventional LL(1) grammar contains 16 nodes. This signifies that by restricting the productions in a way that with every the terminal is generated from left to right expending terminal prefix by one symbol in every step generate the succinct parse tree. As a further illustration, consider Fig. 3 for the parse trees of "ID*(ID–ID)/ ID".

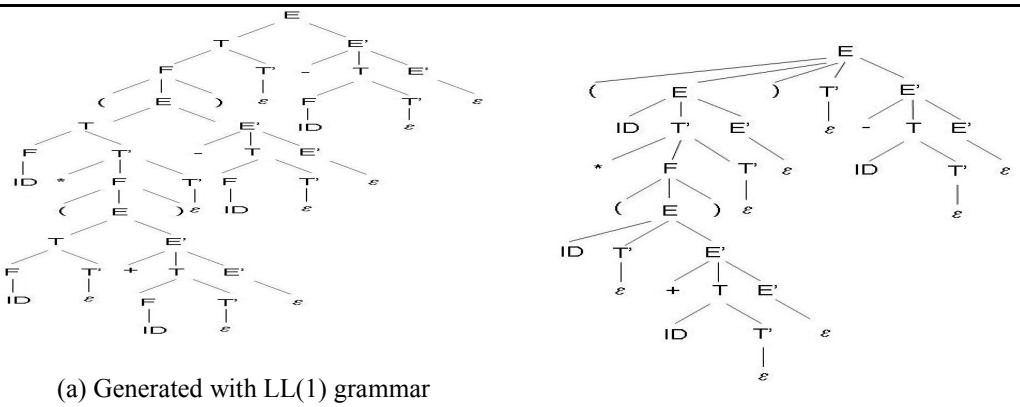


(a) Generated with LL(1) grammar

(b) Generated with terminal prefixed LL(1) grammar

Fig. 3: Parse tree of ID*(ID–ID)/ ID

The parse trees of "ID*(ID–ID)/ ID" illustrated in Fig 3 shows that terminal prefixed support the generation of compact parse tree, which indicates the positive impact of terminal prefixing on parse tree generation. As an another illustration, consider Fig. 4 for the parse tree of "(ID*(ID+ID)-ID)-ID".



(a) Generated with LL(1) grammar

(b) Generated with terminal prefixed LL(1) grammar

Fig. 4: Syntax tree of "(ID*(ID+ID)-ID)-ID"

The Fig 4. clearly manifest that the parse tree generated with terminal prefixed LL(1) grammar is more compendious then the conventional parse tree; hence the terminal prefixing has a considerable effect on the magnitude of parse trees.
The summary of above parsing trees is included in Table 1.

Table 1: Detail of Internal Nodes in Parse Trees

Input	Internal Nodes	
	LL(1) Grammar	Terminal Prefixed LL(1) Grammar
"ID+ID"	9	6
"(ID+ID)*ID"	16	11
"ID*(ID–ID)/ID"	18	13
"(ID*(ID+ID)-ID)-ID"	29	19

The information included in Table 1 indicates that parse trees which are generated with terminal prefixed grammar are more compact than those which are generated with conventional LL(1) grammar because the terminal prefixing restricts the expansion of the left subtree of every parse tree. The restriction on the expansion of left subtree intrinsically force the generation of at least one terminal at each level, so the parse trees generated with terminal prefixed grammar contain less number of internal nodes.
The significant impact of terminal prefixing on the size of derivation and parse tree suggest that it would have significant impact of parsing as well. In order to identify the actual significance of terminal prefixing on LL(1) parsing consider the parsing of an arithmetic expression. The conventional LL(1) parsing table for an arithmetic expression is shown in Table 2.

Table 2. LL(1) Parsing Table

Nonterminals	Input Symbols							
	ID	+	-	*	/	(	)	$
E	TE'					TE'		
E'		+TE'	-TE'				ε	ε
T	FT'					FT'		
T'		ε	ε	*FT'	/FT'		ε	ε
F	ID					(E)		

The LL(1) parsing developed over the terminal prefixed arithmetic expression is shown in Table 3.

Table 3. Terminal Prefixed LL(1) Parsing Table

Nonterminal	Input Symbols (T)							
	ID	+	-	*	/	(	)	$
E	ID T' E'					(E) T' E'		
E'		+TE'	-TE'				ε	ε
T	ID T'					(E) T'		
T'		ε	ε	*FT'	/FT'		ε	ε
F	ID					(E)		

The method of syntax verification is far simple and same for terminal prefixed LL(1) parser and the conventional LL(1) parser. During trace the string or a program which is to be verified by a parser is placed on the column which represents the input of a parser. A special symbol like "$" is suffixed with input to mark the end of input. Similarly the bottom of stack is marked with "$" and followed by the starting symbol. In the case of arithmetic expression grammar, the "E" which represents the expression is initially pushed on the stack of a parser. During syntax checking the parsers makes the series of moves by taking several steps. If *B* is the grammar symbol on the top of parser stack, and *b* is "$" or a terminal symbol, then following are the possibilities.

1. If B = "$" and b = "$", then parser terminates and declares successful completion of syntax checking.
2. If $B = b \neq$ "$", then parser pops B off the stack and continue the verification of next input symbol.
3. If B is a variable, the parser check entry PARSINGTABLE[B, b] of the parsing table. If PARSINGTABLE[B, b] = { $B \rightarrow \alpha\beta\gamma$}, the parser replaces B on top of the parser stack by $\gamma\beta\alpha$, with α on top. If PARSINGTABLE [B, b] = Blank Entry (error), the parser performs the error handling.

Now the syntax of randomly selected inputs will be checked by using the conventional and the terminal prefixed LL(1) parsing. As a first illustration consider the parse of "ID + ID". The entries on stack and input are delimited by double quotation marks.

Table 4. LL(1) Parse of "ID + ID"

Step #	Stack	Input
1	"$E"	"ID+ID$"
2	"$E'T"	"ID+ID$"
3	"$E'T'F"	"ID+ID$"
4	"$E'T'ID"	"ID+ID$"
5	"$E'T'"	"+ ID$"
6	"$E'"	"+ ID$"
7	"$E'T+"	"+ ID$"
8	"$E'T"	"ID$"
9	"$E'T'F"	"ID$"
10	"$E'T'ID"	"ID$"
11	"$E'T'"	"$"
12	"$E'"	"$"
13	"$"	"$"

Table 5. Terminal Prefixed LL(1) parse of "ID + ID"

Step #	Stack	Input
1	"$E"	"ID+ID$"
2	"$E'T'ID"	"ID+ID$"
3	"$E'T'"	"+ID$"
4	"$E'"	"+ID$"
5	"$E'T+"	"+ ID$"
6	"$E'T"	"ID$"
7	"$E'T'ID"	"ID$"
8	"$E'T'"	"$"
9	"$E'"	"$"
10	"$"	"$"

The trace of expression illustrated in Table 4 and 5 is based on method described in the previous discussion. It can be seen that terminal prefixed LL(1) technique required ten steps during the parse of "ID + ID", whereas standard LL(1) parsing required thirteen steps to parse "ID + ID". As a further illustration, let us consider the parse of "(ID + ID) * ID".

Table 6. LL(1) parse of (ID + ID) * ID

Step#	Stack	Input
1	"$E"	"(ID+ID)*ID$"
2	"$E'T"	"(ID+ID)*ID$"
3	"$E'T'F"	"(ID+ID)*ID$"
4	"$E'T')E("	"_ID+ID)*ID$"
5	"$E'T')E"	"ID+ID)*ID$"
6	"$E'T')E'T"	"ID+ID)*ID $"
7	"$E'T')E'T'F"	"ID+ID)*ID$"
8	"$E'T')E'T'ID"	"ID+ID)*ID$"
9	"$E'T') E' T' "	"+ ID)*ID$"
10	"$E'T') E' "	"+ ID)*ID$"
11	"$E'T') E' T +"	"+ ID)*ID$"
12	"$E'T') E' T"	"ID)* ID$"
13	"$E'T') E' T' F"	"ID)*ID$"
14	"$E'T') E' T' ID"	"ID)*ID$"
15	"$E'T') E' T' "	")*ID$"
16	"$E'T') E' "	")*ID$"
17	"$E'T') "	")*ID$"
18	"$E'T' "	"* ID$"
19	"$E'T'F* "	"* ID$"
20	"$E'T'F "	"ID$"
21	"$E'T'ID "	"$"
22	"$E'T' "	"$"
23	"$E' "	"$"
24	"$"	"$"

Table 7. Terminal Prefixed LL(1) parse of (ID + ID) * ID

Step#	Stack	INPUT
1	"$E "	"(ID+ID)*ID$"
2	"$ E'T')E("	"(ID+ID)*ID$"
3	"$ E'T')E "	"ID+ID)*ID$"
4	"$ E'T')E'T'ID "	"ID+ID)*ID$"
5	"$ E'T')E'T' "	"+ ID)*ID$"
6	"$ E'T')E' "	"+ ID)*ID$"
7	"$ E'T')E'T + "	"+ ID)*ID$"
8	"$ E'T')E'T "	"ID)*ID$"
9	"$E'T')E'T'ID "	"ID)*ID$"
10	"$E'T')E'T' "	")*ID$"
11	"$E'T')E' "	")*ID$"
12	"$E'T') "	")*ID$"
13	"$E'T' "	"*ID$"
14	"$E'T'F* "	"*ID$"
15	"$E'T'F "	"ID$"
16	"$E'T'ID "	"ID$"
17	"$E'T' "	"$"
18	"$E' "	"$"
19	"$"	"$"

From Table 6 it can be seen that standard LL(1) parsing required twenty fours steps to parse "(ID+ID)*ID", whereas parsing the same expression with terminal prefixed LL(1) parse required nineteen steps as shown in Table 7.
Now consider the parse of "ID*(ID–ID)/ ID".

Table 8. LL(1) parse of "ID*(ID–ID)/ ID"

Step#	Stack	Input
1	"$E "	"ID*(ID–ID)/ID$"
2	"$E'T "	"ID*(ID–ID)/ ID$"
3	"$E'T'F "	"ID*(ID–ID)/ID$"
4	"$E'T'ID "	"ID*(ID–ID)/ ID$"
5	"$E'T' "	"*(ID–ID)/ID$"
6	"$E'T'F* "	"*(ID–ID)/ID$"
7	"$E'T'F "	"(ID–ID)/ID$"
8	"$E'T')E("	"(ID–ID)/ID$"
9	"$E'T')E "	"ID–ID)/ID$"
10	"$E'T')E'T "	"ID–ID)/ID$"
11	"$E'T')E'T'F "	"ID–ID)/ID$"
12	"$E'T')E'T'ID "	"ID–ID)/ID$"
13	"$E'T')E'T' "	"–ID)/ID$"
14	"$E'T')E' "	"–ID)/ID$"
15	"$E'T')E'T- "	"–ID)/ID$"
16	"$E'T')E'T "	"ID)/ID$"
17	"$E'T')E'T'F "	"ID)/ID$"
18	"$E'T')E'T'ID "	"ID)/ID$"
19	"$E'T')E'T' "	")/ID$"
20	"$E'T')E' "	")/ID$"
21	"$E'T') "	")/ID$"
22	"$E'T' "	"/ID$"
23	"$E'T'F/ "	"/ID$"
24	"$E'T'F "	"ID$"
25	"$E'T'ID "	"ID$"
26	"$E'T' "	"$"
27	"$E' "	"$"
28	"$"	"$"

Table 9. Terminal Prefixed LL(1) parse of "ID*(ID–ID)/ ID"

Step#	Stack	Input
1	"$E "	"ID*(ID–ID)/ID$"
2	"$E'T'ID "	"ID*(ID–ID)/ID$"
3	"$E'T' "	"* (ID–ID)/ID$"
4	"$E'T'F* "	"* (ID–ID)/ID$"
5	"$E'T'F "	"(ID–ID)/ID$"
6	"$E'T')E("	"(ID–ID)/ID$"
7	"$E'T')E "	"ID–ID)/ ID$"
8	"$E'T')E'T'ID "	"ID–ID)/ ID$"
9	"$E'T')E'T' "	"–ID)/ ID$"
10	"$E'T')E' "	"–ID)/ ID$"
11	"$E'T')E'T- "	"–ID)/ ID$"
12	"$E'T')E'T "	"ID)/ ID$"
13	"$E'T')E'T'ID "	"ID)/ ID$"
14	"$E'T')E'T' "	")/ID$"
15	"$E'T')E' "	")/ID$"
16	"$E'T') "	")/ID$"
17	"$E'T' "	"/ID$"
18	"$E'T'F/ "	"/ID$"
19	"$E'T'F "	"ID$"
20	"$E'T'ID "	"ID$"
21	"$E'T' "	"$"
22	"$E' "	"$"
23	"$" "	"$"

Standard LL(1) parse required twenty eight steps during the parse of ID*(ID–ID)/ ID as shown in Fig 8. Whereas Terminal prefixed LL(1) technique performs twenty three steps during the parse of same input as shown in Fig 9.
Now consider the parse of (ID*(ID+ID)-ID)-ID.

Table 10. LL(1) parse of (ID*(ID+ID)-ID)-ID

Step#	Stack	Input
1	"$E "	"(ID*(ID+ID)-ID)-ID$"
2	"$E'T "	"(ID*(ID+ID)-ID)-ID$"
3	"$E'T'F "	"(ID*(ID+ID)-ID)-ID$"
4	"$E'T')E("	"(ID*(ID+ID)-ID)-ID$"
5	"$E'T')E "	"(ID*(ID+ID)-ID)-ID$"
6	"$E'T')E'T "	"ID*(ID+ID)-ID)-ID$"
7	"$E'T')E'T'F "	"ID*(ID+ID)-ID)-ID$"
8	"$E'T')E'T'ID "	"ID*(ID+ID)-ID)-ID$"
9	"$E'T')E'T' "	"*(ID+ID)-ID)-ID$"
10	"$E'T')E'T'F* "	"*(ID+ID)-ID)-ID$"
11	"$E'T')E'T'F "	"(ID+ID)-ID)-ID$"
12	"$E'T')E'T')E("	"(ID+ID)-ID)-ID$"
13	"$E'T')E'T')E "	"ID+ID)-ID)-ID$"
14	"$E'T')E'T')E'T "	"ID+ID)-ID)-ID$"
15	"$E'T')E'T')E'T'F "	"ID+ID)-ID)-ID$"
16	"$E'T')E'T')E'T'ID"	"ID+ID)-ID)-ID$"
17	"$E'T')E'T')E'T' "	"+ID)-ID)-ID$"
18	"$E'T')E'T')E' "	"+ID)-ID)-ID$"
19	"$E'T')E'T')E'T+ "	"+ID)-ID)-ID$"
20	"$E'T')E'T')E'T "	"ID)-ID)-ID$"
21	"$E'T')E'T')E'T'F "	"ID)-ID)-ID$"
22	"$E'T')E'T')E'T'ID"	"ID)-ID)-ID$"
23	"$E'T')E'T')E'T' "	")-ID)-ID$"
24	"$E'T')E'T')E' "	")-ID)-ID$"
25	"$E'T')E'T') "	")-ID)-ID$"

Table 11. Terminal Prefixed LL(1) parse of (ID*(ID+ID)-ID)-ID

Step#	Stack	Input
1	"$E "	"(ID*(ID+ID)-ID)-ID$"
2	"$E'T')E("	"(ID*(ID+ID)-ID)-ID$"
3	"$E'T')E "	"ID*(ID+ID)-ID)-ID$"
4	"$E'T')E'T'ID "	"ID*(ID+ID)-ID)-ID$"
5	"$E'T')E'T' "	"*(ID+ID)-ID)-ID$"
6	"$E'T')E'T'F* "	"*(ID+ID)-ID)-ID$"
7	"$E'T')E'T'F "	"(ID+ID)-ID)-ID$"
8	"$E'T')E'T')E("	"(ID+ID)-ID)-ID$"
9	"$E'T')E'T')E "	"ID+ID)-ID)-ID$"
10	"$E'T')E'T')E'T'ID"	"ID+ID)-ID)-ID$"
11	"$E'T')E'T')E'T' "	"+ID)-ID)-ID$"
12	"$E'T')E'T')E' "	"+ID)-ID)-ID$"
13	"$E'T')E'T')E'T+ "	"+ID)-ID)-ID$"
14	"$E'T')E'T')E'T "	"ID)-ID)-ID$"
15	"$E'T')E'T')E'T'ID"	"ID)-ID)-ID$"
16	"$E'T')E'T')E'T' "	")-ID)-ID$"
17	"$E'T')E'T')E' "	")-ID)-ID$"
18	"$E'T')E'T') "	")-ID)-ID$"
19	"$E'T')E'T' "	"-ID)-ID$"
20	"$E'T')E' "	"-ID)-ID$"
21	"$E'T')E'T- "	"-ID)-ID$"
22	"$E'T')E'T "	"ID)-ID$"
23	"$E'T')E'T'ID "	"ID)-ID$"
24	"$E'T')E'T' "	")-ID$"

26	"$E'T')E'T' "	"-ID)-ID$"
27	"$E'T')E' "	"-ID)-ID$"
28	"$E'T')E'T- "	"-ID)-ID$"
29	"$E'T')E'T "	"ID)-ID$"
30	"$E'T')E'T'F "	"ID)-ID$"
31	"$E'T')E'T'ID "	"ID)-ID$"
32	"$E'T')E'T' "	")-ID$"
33	"$E'T')E' "	")-ID$"
34	"$E'T') "	")-ID$"
35	"$E'T' "	"-ID$"
36	"$E' "	"-ID$"
37	"$E'T- "	"-ID$"
38	"$E'T "	"ID$"
39	"$E'T'F "	"ID$"
40	"$E'T'ID "	"ID$"
41	"$E'T' "	"$"
42	"$E' "	"$"
43	"$ "	"$"

25	"$E'T')E' "	")-ID$"
26	"$E'T') "	")-ID$"
27	"$E'T' "	"-ID$"
28	"$E' "	"-ID$"
29	"$E'T- "	"-ID$"
30	"$E'T "	"ID$"
31	"$E'T'ID "	"ID$"
32	"$E'T' "	"$"
33	"$E' "	"$"
34	"$"	"$"

Standard LL(1) parser shown in Table 10 required forty three steps during the parse of "(ID*(ID+ID)-ID)-ID" whereas terminal prefixed parser shown in Table 11 required thirty four steps to parse the same expression.
The summary of parsing above four expressions with standard and terminal prefixed parser is given in Table 12.

Table 12. LL(1) parsing versus terminal prefixed parsing

Input	Number of Steps	
	LL(1) Parsing	Terminal prefixed LL(1) Parsing
"ID+ID"	13	10
"(ID+ID)*ID"	24	19
"ID*(ID-ID)/ID"	28	23
"(ID*(ID+ID)-ID)-ID"	43	34

From Table 12 it can be seen that parsing with terminal prefixed parsing is much efficient then the conventional method. The information included in Table 12 is graphically represented in Fig 5.

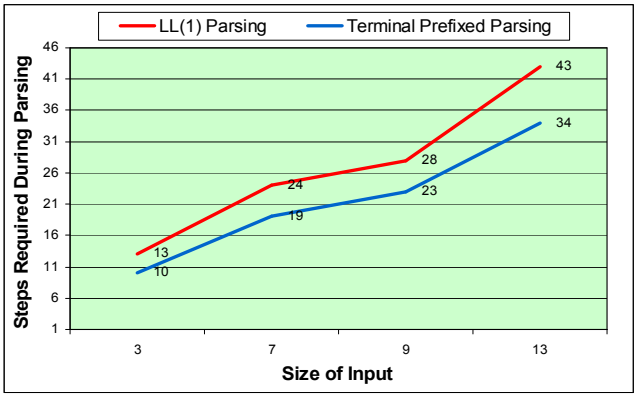


Fig. 5: LL(1) versus terminal prefixed parsing on expression grammar

The growth rate indicated in Fig 5. shows that the performance of terminal prefixed parsing is much better then the conventional method and the difference between their performance increase with the size of input.

6. DISCUSSION

The terminal prefixing presented in this article for LL(1) grammar and its impact on the parse tree and parser is illustrated in previous section. Every LL(1) grammar can be transformed in terminal prefixed grammar and it has been observed that terminal prefixed LL(1) grammar is intrinsically more coherent then the conventional LL(1) grammar as it generates at least one terminal in each step so for any string of length n, it would take around n derivations. Therefore the worst case complexity of string generation with terminal prefixed LL(1) grammar is O(n). Similarly the parse tree generated with terminal prefixed grammar is quite compact because terminal prefixing restrict the expansion of left subtree.

In terminal prefixed grammar the prefix of the body of every rule is always a terminal symbol or an epsilon production. So the LL(1) parsing based on terminal prefixing naturally have better performance because the internal processing of replacing nonterminal with other nonterminal is naturally eliminated.
The information included in Table 12 evidently shows that parsing with terminal prefixed grammar is much efficient then the conventional LL(1) grammar and similarly the growth rate illustrated in Fig. 5 suggest that performance of terminal prefixed parsing remained better with difference size of inputs and indeed the difference between the efficiency of terminal prefixed parsing and LL(1) parsing would grow with the increase of input. Although a simple arithmetic expression grammar is used to identify the influence of terminal prefixing on LL(1) grammar but conclusion identified with this single instance of a grammar is rather sufficient to discern the inceptive effect of terminal prefixing to the reduce the steps required during parsing
The overall affect of terminal prefixing on grammar, parse tree and parsing suggest its positive significance yet it has an aftermath which is exceptionally important while the application of terminal prefixing on arbitrary grammar and further exploration of terminal prefixing. The actual arithmetic expression grammar discussed in the start of this section contains 10 productions in which only 2 productions required terminal prefixing. After transformation of arithmetic expression grammar into a terminal prefix form the resultant grammar contained 12 productions which indicates that terminal prefixing would increase the size of grammar in terms of productions. Virtually the LL(1) grammar imposed no particular restriction on the right side of any production and any arbitrary sequence of symbols may be appear on the right side of a production. So in worst case the terminal prefixing of a LL(1) grammar with arbitrary size and sequence of productions would exponentially increase the size of a resultant grammar and therefore terminal prefixing is more preferable for those grammar which results in compact terminal-prefixed grammar.

7. Conclusion

In this paper terminal prefixing which is quite similar to Greibach normal form is introduced and its impact on derivation, parse tree and LL(1) parsing is analyzed. LL(1) grammar is said to be in terminal prefixed form, if the right side of the production starts with terminal symbol and followed by any sequence of terminals and terminals. The productions of LL(1) grammar can't be left recursive and always be conflict free, so every conceivable LL(1) grammar can be transformed into an equivalent terminal prefixed LL(1) grammar and construction of parsing table from terminal prefixed grammar is almost to the conventional method. The discussion and illustration included in this paper suggested that derivation with terminal prefixed grammar is more efficient than the conventional LL(1) grammar and similarly the compact parse tree is generated with terminal prefixed grammar. The terminal prefixing also increased the efficiency of LL(1) parsing but increased increase the size of a resultant grammar and therefore more suitable for grammar which engender compact terminal-prefixed grammar. Further study is underway in three dimensions: i) analyzing the appropriate class of LL(1) grammars which are more suitable for terminal prefixing ii) quantifying the growth rate of the size of a terminal prefixed grammar iii) impact of terminal prefixing on shift reduce parsing.

Acknowledgement
The author would like to express special gratitude to University of Balochistan for the support and thanks to all stakeholders who support the passage of this research. The author is also grateful to all the anonymous reviewers whose recommendations refined this article.

REFERENCES

1. Dick Grune and Ceriel Jacobs, 1991. Parsing Techniques: A Practical Guide, 2nd edition. Springer Publishing Company, Incorporated.
2. Parr, T. and K. Fisher, 2011. LL(*): The foundation of the ANTLR parser generator. In the Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp:425-436.
3. Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman, 2006. Compilers: Principles, Techniques, and Tools, 2nd Edition. Addison Wesley.
4. Kutlu, M. and I. Cicekli, 2015. Noun Phrase Chunking for Turkish Using a Dependency Parser, Information Sciences and Systems, 363:381-391.
5. Rabbi, R. Z., M. I. Shuvo and K. M. Hasan, 2016. Bangla Grammar Pattern Recognition Using Shift Reduce Parser. In the 5th International Conference on Informatics, Electronics and Vision, pp:229-234.
6. Marin, M. A., 2015. Effective Use of Cross-Domain Parsing in Automatic Speech Recognition and Error Detection, PhD dissertation, University of Washington, Seattle, Washington.

7. Thomas. W. Parsons, 1992. Introduction to Compiler construction. W. H. Freeman & Co Ltd.
8. Kenneth C. Louden, 1997. Compiler Construction Principles and Practice. Course Technology Inc.
9. Naveed, M. S., Sarim, M. and Ahsan, K., 2016. Learners Programming Language a Helping System for Introductory Programming Courses. Mehran University Research Journal of Engineering & Technology, 35(3):347-358.
10. Naveed, M. S., Sarim, M. and Ahsan, K., 2015. On the Felicitous Applications of Natural Language. Science International, 27(23):2643-2646.
11. John C. Martin, 2010. Introduction to Languages and The Theory of Computation, 4th Edition. The McGraw-Hill Companies.
12. Lewis II, P. M. and R. E. Stearns, 1968. Syntax-Directed Transduction, Journal of the *ACM*, 15(3):465-488.
13. Rosenkrantz, D. J. and R. E. Stearns, 1970. Properties of Deterministic Top-Down Grammars. Information And Control, 17:226-256.
14. V. O. Safonov, 2010. Trustworthy Compilers. A John Wiley & Sons.
15. E. Rich, 2007. Automata, Computability and Complexity Theory And Applications. Pearson Prentice Hall.
16. Parr, T., S. Harwell and K. Fisher, 2014. Adaptive LL(*) parsing: the power of dynamic analysis. ACM SIGPLAN Notices, 49(10):579-598.
17. Shilling, J. J, 1993. Incremental LL(1) parsing in language-based editors. IEEE Transactions on Software Engineering, 19(9):935-940.
18. Slivnik, B., 2013. LLLR parsing. In the Proceedings of the 28th Annual ACM Symposium on Applied Computing, pp:1698-1699.
19. Slivnik, B., 2016. LLLR Parsing: a Combination of LL and LR Parsing. 5th Symposium on Languages, Applications and Technologies, 51:1-13.
20. Sippu, S. and E. Soisalon-Soininen, 1982. On LL(k) Parsing*. Information And Control, 53(3):141-164.
21. Ukkonen, E., 1983. Lower Bounds on the Size of Deterministic Parsers. Journal of Computer and System Sciences, 26(2):153-170.
22. Blum, N., 2001. On parsing LL-languages. Theoretical Computer Science, 267(1-2):49-59.
23. Li., W. X., 1995. Building Efficient Incremental LL Parsers by augmenting LL Tables and threading parse trees. In the Proceedings of the ACM-SE 33 annual on Southeast regional conference, pp:269-270.
24. Rußmann, A., 1997. Dynamic LL(k) parsing. Acta Informatica, 34(4):267-289.
25. Bertsch, E. and Nederhof, M. –J, 2001. Size/lookahead tradeoff for LL(k)-grammars. Information Processing Letters, 80(3):125-129.
26. Vagner, L. and B. Melichar, 2007. Parallel LL parsing. Acta Informatica. 44(1):1-21.
27. Yingli, Z., 2010, Visualized LL parsing of non-left-recursive grammar. In the Proceedings of the International Conference on Optics, Photonics and Energy Engineering. pp:161-163.
28. Scott, E. and A. Johnstone, 2010. GLL Parsing. Electronic Notes in Theoretical Computer Science, 253(7):177-189.
29. Scott, E. and A. Johnstone, 2013. GLL parse tree generation. Science of Computer Programming, 78(10):1828-1844.
30. Greibach, S. A., 1965. A New Normal-Form Theorem for Context-Free Phrase Structure Grammars. Journal of the ACM, 12(1):42-52.