Spectrum of Engineering Sciences
ISSN (e) 3007-3138 (p) 3007-312X Volume 4, Issue 6, 2026

LL (1) OF TOP-DOWN PARSERS INTEGRATION WITH CHOMSKY NORMAL FORM: A CASE STUDY

Hassan Ali*[1] Dilawar Naseem[2]

[1]*Department of Computer Science, FICT, BUITMS.*
[2] *Aria Institute of Medical Sciences.*
hassanali2010@outlook.com

DOI: https://doi.org/10.5281/zenodo.20775243

Keywords
Parsing, Top-Down Parsing, Chomsky Normal Form, LL (1) Parsing, Compiler.

Article History
Received on 20 May 2026
Accepted on 16 June 2026
Published on 20 June 2026

Corresponding Author: *
Hassan Ali*

Abstract
Parsing is a phase in a compiler where the source code of a program is analyzed to determine its structure. Top-down parsing is a parsing technique used in compiler construction to analyze the structure of source code. The study intended to assimilate top-down parser (TDP), LL (1) and Chomsky Normal Form (CNF). The traditional Arithmetic Expression Grammar (AEG) was used for instigation of TDP, LL(1) and CNF, for initiation the LL(1) and CNF algorithms were assimilated with the calculation of first, follow and parsing table, however the induction of LL(1) and CNF algorithms lead to ambiguity due to epsilon productions.

INTRODUCTION

Parser is also referred to as a syntax analyzer. It can be manually programmed or created automatically or semi-automatically using a parser generator. The process of parsing complements templating, which provides prepared output. Parsers are of two types top-down and bottom-up (Aho and Ullman 1972).

Top-down parser is a parser that generates parse tree for a given input strings using grammar productions by expanding non-terminals, i.e., it begins from the start symbol and finishes at the terminal symbols. It works with the left-most derivations. In advance, a top-down parser smidgens a parse tree. A parse tree is navigated from the root to leaves and each node is visited preceding to following its branches. The branches from a specific node are tracked from left to right.

Bottom-up parser is the parser that builds the parse tree for a given input text using grammar productions by compressing non-terminals, i.e., it begins with non-terminals and ends with the start symbol. It employs the opposite of the right-most derivation. A bottom-up parser generates a parse tree by moving from the leaves to the root, beginning with the leaves. This parse order corresponds to the opposite of a derivation on the right. Bottom-up parser types include LR parser and operator precedence parser (Reddy, P, and Belwal 2024).

LL (1) parsing is a top-down parser that does not require backtracking. It starts reading the input symbols from left to right and is a precedence parser. The "L" in LL (1) distinguishes that the data is prepared from left to right. The second "L" recognizes that it

does the leftmost deduction, and the "1" indicates that only one lookahead is anticipated.

Implementations are done over a table known as the parsing table. FIRST and FOLLOW sets are amalgamated into the development of this parsing table. LL (1) language structures are a subset of context-free grammar (CFG), implying that LL (1) parsing is also a restricted subset of parsing systems. Among the limitations of LL (1) parsing are its inability to handle ambiguous grammars and left-recursive instruction sets (Naseem et al. 2021).

In a formal language theory, a context-free grammar (CFG) is said to be in CNF and was first described as Noam Chomsky form. If and only if all the production sets of the grammar (G) of this form satisfy the following (G) terms:

1. $X \rightarrow Y Z$ with X, Y, Z non-terminal symbols

OR

2. $X \rightarrow x$ with X as non-terminal and letter 'x' as terminal symbol.

OR

3. $S \rightarrow \varepsilon$

Further G has no useless symbols.

As previously specified in two forms, X, Y, and Z are nonterminal symbols, whereas the letter 'x' is a terminal symbol (terminal symbol is said to be a symbol of constant value). While S is the start symbol and (ε) epsilon represents an empty string, neither Y nor Z are the start symbol. If we consider the third production, it only occurs when (G) is part of the Language L (G), where L is the language generated by CFG. Every grammar in CNF is independent of context. To convert a simple grammar to CNF, a series of simple transformations are performed in a certain order.

The introduction builds understanding of the understudied topic and field in which topic relates to, it further brings syntactic structure of the CNF. The literature review brings clarity to the background knowledge, assimilation and the ambiguity that article concludes. Research methodology is the brief possible simplification of the phases or methods adopted to study CFG arithmetic expression grammar (G), and its conversion into equivalent version and that is G'(I)'. In the end the conclusion sums up the case study with possible future work recommendations.

Literature Review

Similar integrations are performed with different TDP and normal forms such as CNF, Greibach Normal Form (GNF), and Kuroda Normal Form (KNF). The parser is still an unresolved research area that needs improvements. Most of the TDP is done by hand or with the help semi-automated methods. Studies have shown significant limitation in case studies and evidence base methods as pointed out by manuscripts produced earlier (Rizqullah and Albassam 2026).

Parsers are a syntactic part of compiler hence called a syntax analyzer. Majority of the manuscripts produced on compilers points to optimization methods and highly diverse source code, yet compiler still struggle and require improvements in producing optimal target code.

The parsers inability to accept ambiguous grammars exaggerate further to epsilon productions, combinations, and loops. Currently compilers and parsers are written by hand made optimization methods and are limited in processing precisely (Mammadli et al. 2021).

Improved algorithms and methods of compilers as well as for parsers strengths environment and reduces negative effect on environment. The extension in computing knowledge resulted in significant energy usage. A study on similar perspective explored two fold proceedings, one is to address energy utilization and second fold is to write in addition to improve algorithms and existing methods (Reddy et al. 2024).

Similar work is done with slightly significant improvements. Traditional English grammar was utilized that can parse limited number of input stack. The CNF algorithm was studied and got induced into LL (1) parser grammar successfully with first, follow and parsing table calculated. However, the induction did not bring out a more significant outcome but newly formulated CNF induced LL (1) algorithm did pointed to some unanswered factors that this study may put some light on (Naseem et al. 2021).

In addition to the wok rehashed earlier it comes to the fact that CNF is one of the widely used and known normal forms. Yet its structure when gets induced or simulated makes grammar larger than before. That further leads to ambiguity and requires some case studies.

An equivalent kind of induction is done with LL (1) and GNF. This study reflects a successful induction with very significant results that when implemented while developing can make a faster processing compilers then before. Here the study also induced preexisting algorithms and resulted notable improvements. The article elaborated traditional LL (1) and GNF algorithms and presented a newly articulated LL (1) induced GNF algorithm (Ali et al. 2020).

Terminal prefixing is introduced on LL (1) as a normal form and its impact was elaborated and results depicted improvement and proficiency in LL (1) parsing alone (Naveed 2017).

The comparison of both the CNF and GNF induced studies points to the fact that more case studies are relevant in reaching a certain amalgamation of assumptions. Another study that puts light on the indication that studies like GNF induction with LL (1), CNF induction with LL (1) and terminal prefixing of LL (1) can be fruitful for not only computer science but for other fields like mathematics as well (Reis 2023).

Materials and Methods

A process is followed depicted below to assimilate the under adopted arithmetic expression grammar following CFG rules into equivalent LL (1) introduced CNF grammar. Let the adopted arithmetic expression grammar be G' with instruction set be I' and resultant equivalent grammar be G' with instruction set be I' as shown in figure 1 below.

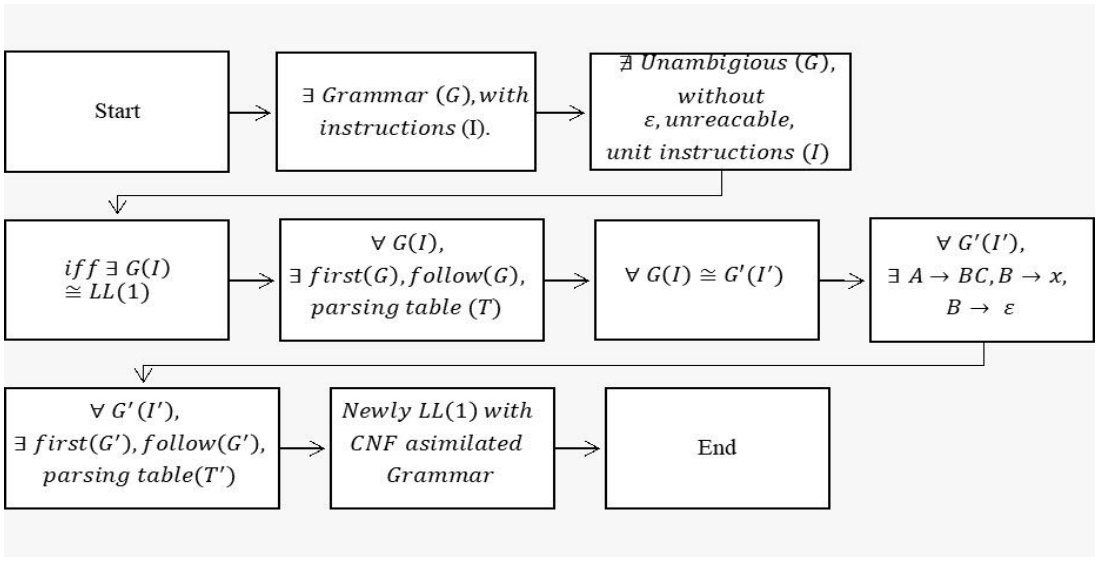

Figure 1. Research Methodology

The adopted research methodology depicted in the figure above demonstrates a method where grammar (G) is arithmetic expression grammar with unambiguous instruction sets (I) that is without epsilon, unreachable, unit and infinite loops. The grammar (G) is equivalent to LL (1), TDP only if it satisfies the TDP algorithmic rules, for smooth processing first (G), follow (G), and parsing table (T) is calculated for the respective input stack.

For TDP LL (1) to get applied on grammar (G) must satisfy LL (1) algorithmic rules then resultant grammar (G') with instructions (I') is newly generated LL (1) assimilated CNF Grammar (G'(I')).

The utilized arithmetic expression grammar (G) is below.

S -> E

E -> T E'

E' -> + T E' | - T E' | ε

T -> F T'

T' -> * F T' | / F T' | ε

F -> num | id

For the utilized arithmetic expression grammar (G (I)), first sets that are first (G) are below.

first(S)={num,id}

first(E)={num,id}

first(E')={+,-,ε }

first(T)={num,id}

first(T')={*,/,ε }

first(F)={num,id}

For the utilized arithmetic expression grammar (G (I)), follow sets that are follow (G) are below.

follow(S)={$}

follow(E)={$}

follow(E')={$}

follow(T)={+,-,$}

follow(T')={+,-, $}

follow(F)={*, /, +, -, $}

For the utilized arithmetic expression grammar (G (I)), parsing table (T) is below.

Parsing Table 1

Non-terminal	+	-	*	/	**num**	**Id**	**$**
S					S -> E	S -> E	
E					E -> T E'	E -> T E'	
E'	E' -> + T E'	E' -> - T E'					E' -> ε
T					T -> F T'	T -> F T'	
T'	T' -> ε	T' -> ε	T' -> * F T'	T' -> / F T'			T' -> ε
F					F -> num	F -> id	

Parsing tree for the above mentioned utilized arithmetic expression grammar (G), parsed instruction (I) set is below.

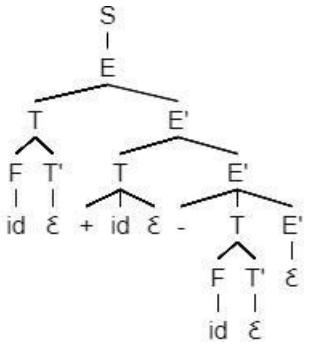

Figure 2 Parsing Tree (id + id – id)

Tree nodes for figure above are id, +, -, ε along with the level of 7, depth 6d, leaf nodes (id, +, -, ε), the root node (S).

Another parsing tree for the above mentioned utilized arithmetic expression grammar (G), parsed instruction (I) set is below.

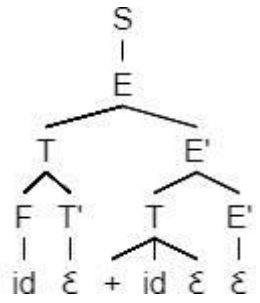

Figure 3 Parsing Tree (id + id)

Tree nodes for figure above are id, +, ε along with the level of 5, depth 4d, leaf nodes (id, +, ε), the root node (S).

A) CNF Assimilated LL (1) Arithmetic Expression Grammar

The assimilation of TDP LL (1) with CNF arithmetic expression grammar brings out syntactical ambiguity due to epsilon production allowed in CNF syntax. The assimilation points to the unknown

knowledge that further assimilations can be applied to achieve parsers efficiency.

The CNF grammar merged with LL (1) is below.

S' -> S

S -> M N | O P | M Q | O Q | ε

M -> H I

N -> J E'

O -> K I

P -> J E'

Q -> L E'

B -> *

C -> +

D -> -

G -> /

H -> F B

I -> F T

J -> C T

K -> F G

L -> D T

E' -> J E' | L E'

T -> H I | K I

F -> num | id

Evaluation and Discussion

The epsilon production in the grammar above is required for the LL (1) grammar syntax but the placement of ε for LL (1) will be different. The LL (1) grammar does not allow left recursion, and hence there is no backtracking in LL (1) parsing, so epsilon production leads the grammar syntax to accept the input stack more than once.

The CNF grammar would be the correct form if the productions are of the form like $A \rightarrow BC$ or $A \rightarrow c$ (where A, B, and C are arbitrary variables and c is an arbitrary symbol). If a language contains ε, then $S \rightarrow \varepsilon$ is allowed where S is the start symbol and forbid S, on RHS, which is the condition that LL (1) does not allow.

Most of the grammar induced with CNF is used without the epsilon productions due to the previously discussed reason that epsilon productions with CNF still require work, as mentioned by (Sochor and Ferrarotti 2022) and (Bozhko, Khatbullina, and Grigorev 2019).

The assimilation of LL (1) and CNF reveals ambiguity considering epsilon productions. The production rules for LL (1) and CNF involving epsilon negates each other resulting in ambiguous grammar structure. If no epsilon productions are considered, then the induction is effective and can parse input stack successfully.

CNF does not allow the epsilon productions but epsilon productions are used in LL (1) parser. LL (1) parser use epsilon production for the removal of conflicts and binary length (2 terminals) on right side of production.

Epsilon based production rarely be used for CNF where epsilon is used exclusively in LL (1) for reduction.

Conclusion and recommendations

The introduction of LL (1) into CNF did provide results as some studies shown mentioned above. The freshly created LL (1) merged CNF is supposed to process the same amount of input symbols with minor improvement than the non-induced LL (1) technique.

The component of the compiler that needs improvement is the parser. Parser works with grammar and languages. Grammar work is still being done because the majority of the parser's work is being done on languages.

The two algorithms LL (1) and CNF that support various grammar structures can be induced without epsilon production and in result gives parser improvement in processing but not significant enough. The typical LL(1) algorithm most likely only processes one lookahead symbol at a time, which means there are no left recursions and the algorithm only takes one symbol throughout processing. On the other hand, the CNF offers a grammatical structure with rules that only permit two non-terminal symbols on the right side and one non-terminal symbol on the left.

The LL (1) method accepts the grammar structure that any left recursion brings, but it does not support backtracking; in the absence of backtracking, left recursion leads to loops that cause the algorithm to behave incorrectly. The LL (1) algorithm only operates from the left and does not permit algorithmic miss-ruling due to conflicting base rules. In some circumstances, the algorithm to cancel out the entire production set in the middle of the processing input stack is called the -productions rule.

CNF still requires work on epsilon production, considering the traditional arithmetic expression grammar. CNF with and without epsilon can be tested by inducing other TDP algorithms.

References

Aho, Alfred V., and Jeffrey D. Ullman. 1972. *The Theory of Parsing, Translation, and Compiling*. Vol. 1. Prentice-Hall Englewood Cliffs, NJ.

Ali, Hassan, Muhammad Shumail Naveed, Dilawar Naseem, and Jawaid Shabbir. 2020. “LL (1) Parser versus GNF Inducted LL (1) Parser on Arithmetic Expressions Grammar: A Comparative Study.” *Quaid-e-Awam University Research Journal of Engineering, Science & Technology* 18(02):89–101. doi:10.52584/QRJ.1802.14.

Bozhko, Sergey, Leyla Khatbullina, and Semyon Grigorev. 2019. “Bar-Hillel Theorem Mechanization in Coq.” Pp. 264–81 in *Logic, Language, Information, and Computation*, edited by R. Iemhoff, M. Moortgat, and R. de Queiroz. Berlin,

Heidelberg: Springer.

Mammadli, Rahim, Marija Selakovic, Felix Wolf, and Michael Pradel. 2021. “Learning to Make Compiler Optimizations More Effective.” Pp. 9–20 in *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, *MAPS 2021*. New York, NY, USA: Association for Computing Machinery.

Naseem, Dilawar, Muhammad Shumail Naveed, Hassan Ali, and Samra Riaz. 2021. “Induction of Chomsky Normal Form in Con-Text-Free Grammar of LL(1) Parser: Some Initial Results.”

Naveed, Muhammad Shumail. 2017. “The Impact of Terminal Prefixing on LL (1) Parsing.” *J. Appl. Environ. Biol. Sci* 7(5):64–76.

Reddy, Nitin, Sai Teja P, and Meena Belwal. 2024. “A Survey on Top Down and Bottom Up Parsing.”

Reis, Fellipe de Souza. 2023. “Desenvolvimento de um interpretador de expressões de teoria dos conjuntos para fins didáticos.” bachelorThesis, Universidade Tecnológica Federal do Paraná.

Rizqullah, Muhammad, and Emad Albassam. 2026. “Model-Agnostic Empirical Evaluation of Test-Driven Prompt Engineering on Improving Accuracy and Efficiency in Large Language Models Python Code Generation.” *IEEE Access* 14:22801–21. doi:10.1109/ACCESS.2026.3662817.

Sochor, Hannes, and Flavio Ferrarotti. 2022. “A Refinement Based Algorithm for Learning Program Input Grammars.” Pp. 138–56 in *From Data to Models and Back*, edited by J. Bowles, G. Broccia, and R. Pellungrini. Cham: Springer International Publishing.