LL(1) Parser versus GNF induced LL(1) Parser on Arithmetic Expressions Grammar: A Comparative Study

Hassan Ali^{1,*}, Muhammad Shumail Naveed¹, Dilawar Naseem¹, Jawaid Shabbir²

¹Department of Computer Science and Information technology, University of Balochistan, Sariah Road, Quetta, Pakistan.

²Department of Computer Engineering, Sir Syed University of Engineering & Technology, Karachi, Pakistan.

*Corresponding author: hassanali2010@outlook.com

Abstract

The prime objective of the proposed study is to determine the induction of Greibach Normal Form (GNF) in Arithmetic Expression Grammars to improve the processing speed of conventional LL(1) parser. Conventional arithmetic expression grammar and its equivalent LL(1) is used in the study which is converted. A transformation method is defined which converts the selected grammar into a Greibach normal form that is further converted into a GNF based parser through a method proposed in the study. These two parsers are analyzed by considering 399 cases of arithmetic expressions. During statistical analysis, the results are initially examined with the Kolmogorov-Smirnov and Shapiro-Wilk test. The statistical significance of the proposed method is evaluated with the Mann-Whitney U test. The study described that GNF based LL(1) parser for arithmetic take fewer steps than conventional LL(1) grammar. The ranks and asymptotic significance depict that the GNF based LL(1) method is significant than the conventional LL(1) approach. The study adds to the knowledge of parsers itself, parser expression grammars (PEG's), LL(1) grammars, Greibach Normal Form (GNF) induced grammar structure, and the induction of Arithmetic PEG's LL(1) to GNF based grammar.

Keywords—Parsing, top-down parsing, LL(1) parsing, Greibach normal form, compiler

1 Introduction

ACCORDING to Grune [1], “Parsing is the process of structuring a linear representation by a given grammar”. The linear representation can be a sentence, a computer program, geological strata, a piece of music, the chemical equation, etc. There can be various reasons to do a process named parsing, but one of them is to obtain a structure of an object that can be processed and understood better. Parsers are one of the best-understood branches of computer science which specifically deals with compilers, but parsers are already being used extensively in several disciplines, such as in computer science for compiler construction, database interfaces, self-describing databases, artificial intelligence. A parser is the second important component after compiler which sees a stream of words extracted from the sentences explained to its syntactic part. The parser derives

a syntactic structure for the program, fitting the words into a grammatical model of the source programming language. If the parser determines that the input stream is a valid program, it builds a concrete model of the program for use by the later phases of compilation. If the input stream is not a valid program, the parser reports the problem and appropriate diagnostic information to the user. Parsing some string according to some grammar means to reproduce the string to know how a string is produced from some grammar. One of the first publications on parsing is Greibach’s doctoral thesis in 1963 [2].

The requirement that we have while producing a production tree in parsing is the need to know whether a string does or does not belong to some grammar. The basic connection between sentence and grammar is parsed tree and for parse tree, there are dozens of techniques, yet top-down parsing and bottom-up parsing are the popular techniques for parsing. Naveed et. al. [3] show that there are different types of parsers, top-down parsers (with and without

ISSN: 2523-0379 (Online), ISSN: 1605-8607 (Print)

DOI: 10.52584/QRJ.1802.14

This is an open access article published by Quaid-e-Awam University of Engineering Science & Technology, Nawabshah, Pakistan under CC BY 4.0 International License.

backtracking) and bottom-up parsers (operator precedence and LR parsers). Top-down parsers with full backtracking use brute force methods and the ones without backtracking are recursive descent parsers and non-recursive descent parsers (LL(1)).

Recursive descent parsing is involved when a recursive function is written. According to the dictionary of computing, recursive means to write a program that needs many executions and in the mathematical and linguistic sense means involvement of repeated applications. On the other hand, non-recursive parsing relates to LL(1) parsing and during the construction of LL(1), grammar left recursions and non-determinism must be excluded.

As per Parr et. al. [4], top-down parsing is the process that imitates the original process of production, i.e., to track down from the start symbol, and bottom-up parsing is to roll back the productions up to the start symbol. LL parser is a top-down parser for a subset of context-free language. It parses the input from left to right, performing the leftmost derivation of the sentence. An LL parser is called an LL(K) parser if it uses K tokens of look-ahead when parsing a sentence. Looking ahead at the next symbol helps in deciding which choice to make. The top-down parser does the prediction of input and this prediction has either a terminal symbol in front.

There are different types of parsers and LL(1) is one of them. LL(1) belongs to top-down parsers. LL and LL(1) grammars are of great interest and parsers of these grammars are easy to construct and the reason behind many computer languages are designed to be LL (1). LL (1) works as follows. Let the following rules be of LL (1) grammar and the input be $(\beta + \beta)$.

- $\gamma \rightarrow \alpha$
- $\gamma \rightarrow (\gamma + \alpha)$
- $\alpha \rightarrow \beta$

Formal language in mathematics, computer science, and linguistics is a set of strings of symbols together with a set of rules specific to it. In formal language theory, some of the forms that exist are Chomsky normal form, Greibach Normal Form, Kuroda Normal Form, and Normal Form. A set of strings are in normal form if it cannot be rewritten any further depending upon the rewriting an object can be in several normal forms or none at all. As per Aggarwal et. al. [5], languages can be used for multiple unsolved areas and languages in the past have also been used to

solved different complications.

In formal language theory, a context-free grammar is in Greibach normal form if the right-hand side of the production rule starts with a terminal symbol and may be followed by some other symbols. This allows epsilon (ϵ) to be a part of the production and this form is described by Sheila Greibach [2]. A context-free grammar is in Greibach normal form if the productions are in the following form,

$$\begin{aligned} A &\rightarrow b \\ A &\rightarrow bC_1, C_2, C_3, \dots, C_n \quad A \rightarrow b \\ A &\rightarrow b C_1 C_2 C_3 \dots C_n \end{aligned}$$

where $A, C_1, C_2, C_3, \dots, C_n$ are non-terminal symbols and b is a terminal symbol. If the non-terminal symbol A gives the terminal symbol, then it is Greibach normal form. And, if a production or productions like above, where A gives a terminal symbol b followed by a set of variables or non-terminal symbols, then the productions are in the form of Greibach normal form. Parsers deal by understanding languages and their grammars by dividing them into smaller components which can be analyzed further, such as parsing the sentence would involve dividing it into words and phrases and identifying each type of component. Epsilon (ϵ) free LL (1) language equals the class of deterministic languages. Deterministic top-down parsers apply a restriction on grammar that right-hand side of non-terminal symbols of production rule starts with a different terminals symbol and leaves it with at most one element.

The prime objective of the proposed study is to determine whether the induction of Geribach normal form in context-free grammar improves the processing speed of the LL(1) parser. The said objective would be addressed by considering the following sub-objectives.

- Analysis of the contemporaneous structure of the arithmetic grammar for LL (1) and GNF induced grammars.
- Definition of LL (1) and GNF induced LL (1) grammar and possible patterns for its simplification.
- Asymptotic analysis of the synthesis of LL(1) and GNF induced LL(1) parsers.

2 Literature Review

According to Fisher et. al. [6], LL (1) and strong LL (1) differ in error detection. Both have corrected prefix property, but LL (1) has an immediate error detection mechanism. When it encounters a large string, it rejects it but strong LL (1) on the other

hand can produce wrong parsing strings. Such errors can seriously interface with the correction process. Fisher et. al. [6] suggested that instead of strong LL (1), LL (1) must be used results tripled the number of non-error entries in the parsing table. Ghezzi et. al. [7] suggested the transformation of strong LL (1) into structured strong LL (1) causing the increased size of the parsing table. According to the work of Parr et. al. [8], parsers are not a tackled issue and notwithstanding the intensity of (PEGs) parser expression language structures (grammars). Adding non-determinism to conventional LL and LR, parsers can prompt unforeseen parse time conduct and presents down to earth issues with blunder taking care of, single-step investigating, and side affecting installed language activities.

Slivnik et. al. [9] and Yamaguchi et. al. [10] studied LL(K) and LR parsers per induced LR(K) parser called left LR(K) parsers. They induced the properties of LL(k) parser to LR(K) where compiler remains at present input string and does not leave any markers to the prefix of string and stop compiling while there is a suffix. The result shows same compiling time for induced and non-induced derivations. Edelman et. al. [11] made a performance comparison between simple LL(1) with derivative, Zippy parser with cache property, and Scala parser combinators library. In his analysis, Scala parsers outperforms rest, while cached zippy parsers outperforms zippy parser and simple LL(1) parser remains same. All parsers are with derivatives and numbering remains same for speed and parse time(ms).

Parr et. al. [12] argue theoretically and in practice for LL and LR parsers in making lookahead number of tokens more than one, say $K > 1$, and justify by making it possible by more heuristics computing approach. Beatty studies LL(1) and LALR (1) parsers with expansion function and states two cases for theorem by letting Grammar (G) stored in array positions and there are no lookup functions then rules of grammar (G) also stored in an array. Arithmetic arguments result in a total of $n(n + 1)$ non-error *gotos* and non-error lookaheads $3n + 1$, making LL (1) storage $O(-G-2)$ and LALR (1) storage linear $O(-G-)$. Taylor et. al. [13] provide evidence that most of the compiler designs in primers of parsing algorithms and deterministic push-down automaton associations are couched. Their evidential study shows that parsing algorithms are very much clear to extinct where they can be constructed as a deterministic automaton.

Nguyen et. al. [14] transform LL(1) into object-oriented predictive descent parsers and the resultant

is a grammar with inheritance class model properties where tokens determine whether input matches the grammar within a sequence made possible by eliminating some conditions and by adding a bit of a perturbation following the open-ended tokens. Parr and Fisher [15] studied parser expression grammars (PEG) and GLR, stating that parsers do not solve problems by pointing their focus on LL and LR. They also pointed the fact that if LL and LR get introduced to PEG altogether, GLR causes parsers to run time errors, single-step debugging, and grammar action effects. They further analyzed LL(*) parser algorithm which has properties as error handling and maintains cyclic lookaheads which on the way overcomes PEG and GLR parsers errors. Their study reveals that ANTLR parsers remove backtracking and LL (*) includes the parser performance. Slivnik et. al. [16] introduce LLLR parser which has the properties of LL(K) and LR(K) parsers with no backtracking resolving conflicts with LL(K) instead of LR(K) parser. Embedment results in a new parser strategy LLL(K) that in comparison to LL (*) comes without backtracking.

Parr et. al. [15] put on a fact that besides all advances and latest technologies introduced to parsers, parsing is still not a solved field. Further directing towards side embedment are weak performance, and counter-intuition strategies with other weaknesses of parsers such as PEG's, GLR, GLL, LL (*). They introduced a new strategy ALL (*) that includes left-recursion and grammar rewriting together with ANTLR-4 data of 5000 downloads per month. Sippu and Soininen [17] introduced canonical LL(K) parsers with $K = 1$, error detection, and error handling capabilities. Canonical LL(K) parsers are the combination of properties of LR(K) and LL(K) parsers. They also point out the fact that the LL(K) parser is a dual nature of the LR(K) parser.

Afrozeh et. al. [18] presented a generalized LL (GLL) parser and proposed a new GSS parser with properties of parsing highly ambiguous grammar and deterministic grammars. GLL and GSS both at worst complexity times are cubic. Bertsch et. al. [19] studied time complexity of parsers with spaces included in most grammar original forms. Studies of time complexity of context-free grammars (CFG's) shows that they are cubic with spaces in the input stack included. LL(1), bi-directional LL(1) and XML grammars are mostly linear, but sometimes quadratic, depending on the number of spaces included in the input stack. Scott et. al. [20] studied GLL algorithms with Tomita-style RNGLR and produced a new algorithm that performs derivation in its

algorithm formation structure and can drive SPPF style for any input string for the parser. Shilling et. al. [21] introduced a language editor for LL(1) parser generators as the same kind that is used in Fred-language editors. The resultant parser performs efficiently after the grammar is transformed and adds to the benefit of program structure.

Youshida et. al. [22] worked on algorithm H and revealed structural properties like recognizing LL(1) products by table handling limited to a set of instructions without mathematical operations. Their resultant grammar can also pinpoint the instructions that do not belong to LL(1) grammar. They further addressed the fact that most of the parsers work is related to LL(1) and is proof of the fact that LL(1) mostly has scholars' attention. Henceforth, the authors introduced the new algorithm that is semi-LL(*) with $K \geq 2$ and they also discussed algorithm proof. Lasser et. al. [23] introduced the parser generator of LL(1) grammar that recognizes sets of input strings that belong to LL(1) grammar structure. Authors used Coq proof to test and verify their parser generator. They further extracted source code of tools generator to further generate JSON parser which turns out to be cubic in time complexity in comparison to unverified ones. Singal et. al. [24] developed a tool CYCLOPS to address parse conditions related to LL(1) grammar as it is still an intensive work, they mentioned. Their tool CYCLOPS is a combination of two utilities syntax error handling and parser synthesis. Cyclops (tool) performs 80% of targets in 30 seconds. Femister et. al. [25] created a parser generator that holds the properties in the combination of skeleton parser and table generator parser that successfully generates the LL(1) based parser with implementation on turbo pascal and author also discussed new LPARSER in comparison with LR and LL parsers.

Goloveshkin et. al. [26] introduced Land parser that is used for *Any* based modification. In *tolerant* grammar syntax, water resembles irrelevant instruction that is being replaced by the symbol *Any* for Land parser. In the study, a technique is described to eliminate error and to detect error recovery under LL and LR parsers. Results describe that for complete programs *C#* and Java parsers performed more viable with a newly generated parser. Nijholt et. al. [27] suggested that for any appropriate context-free grammar G , there exists a context-free grammar G' which is Non-left recursive (NLR) and transforming grammar into normal forms or to grammars that have other parsing properties can once in a while prompt quicker or more conservative parsers for grammars.

They further suggested that the first transformations to Greibach normal form are because of the creator Greibach and her calculation changes an epsilon free context-free grammar without single generation into a context-free grammar in normal form.

An analogous method for the adaptation of context-free grammar into Greibach normal form is conferred by Hopcroft et. al. [28]. Starting with the definition of Greibach normal form, i.e., a context-free grammar is in Greibach normal form if productions are of the form as $A \rightarrow aX$, where $a \in T \wedge X \in V^*$, T is terminal symbol, V stands for variable, and $A \rightarrow aX$ represents any number of variables before terminal symbol at the very start of production.

A theorem that Hopcroft adapted for the conversion is as follows. To convert context-free grammar into Greibach normal form, context-free grammar is firstly converted into Chomsky normal form i.e., in Chomsky normal form there can only be single or only one variable after the terminal symbol as $B \rightarrow bY$, where $b \in T \wedge Y \in V$. $\forall$ context free grammar $G \ni G' \equiv$ Greibach normal form where $\epsilon \notin L(G)$. For the conversion into Chomsky normal form let there be a grammar $G = (V, T, P, S)$, conversion starts by eliminating useless variables that cannot be used as terminal symbols and by eliminating unused variables. Thirdly eliminating ϵ production (productions that leads rules to become null or void as result). Fourthly, by eliminating unit productions and hence, there will be a grammar G in Chomsky normal form. Medeiros et. al. [29] and Medeiros et. al. [30] presented a parser expression grammar (PEG). While working on the structure of PEG's, authors also demonstrated some properties of context-free grammars (CFG) and PEG's. Some of them are as follows. The empty string below provides knowledge and basic arithmetic formulation of any grammar G with any number of variables X including epsilon ϵ as part of the grammar.

Considering a grammar G with any number of variables X including epsilon ϵ , there exists a terminal symbol or terminal state with three cases. The case I was a terminal symbol (I) within grammar G exists with a combination of any other variable a and X being input string from G . Case II of a terminal symbol (II) depicts two different productions cannot lead to the same output. Case III of a terminal symbol (III) depicts ϵ can not be part of the production where it leads the whole production set to a null state. Variable I explains production A of grammar G can provide xy where it can also provide y or X separately. Variable II depicts a production must have access to all the possible symbols of grammar.

Empty String - $G[\varepsilon]X \cong (X, \varepsilon)$

Terminal Symbol(I) - $G[a]aX \cong (X, a)$

Terminal Symbol(II) - $G[b]aX \cong failure$,
where $b \neq a$

Terminal Symbol(III) - $G[c]\varepsilon \cong failure$,
where $c, b \neq a$

Variables (I)

$$G[P(A)]xy \cong (y, x)G[P(A)](y, A[x]).$$

Variables (II)

$$\frac{G[P(A)]x \cong failure}{G[(A)]x \cong failure}$$

Reghizzi et. al. [31] worked on Chomsky normal form (CNF) and Double Greibach normal form (DGNF) while inducing CNF to DGNF. Working with Dysk Ω and homomorphism, Ω leads to the properties that provides grammar G to be independent of its size and dependent on the current production parser is working on. Now, with these properties' authors work on two cases one with alphabetic type homogeneity and the second without homogeneity. They have improved the polynomial complexity of $O(|\Sigma|^{46})$ to degree two. Mascarenhas et. al. [32] formulized context free-grammar (CFG) with parsing expression grammar (PEG) and presented a newly formalized form that takes LL grammars irrespective of what type it belongs to. Authors also differentiate and formed similarities of CFG's and PEG's. They also formalize with defining parsing expression grammar syntactic structure.

$$c = \varepsilon \mid b \mid B \mid a_1 a_2 \mid a_1 \mid a_2$$

Where parsing expression grammar syntax above explains empty string, terminal, non-terminal, concatenation, and choices as a_1 and a_2 .

Medeiros et. al. [33] came up with a machine based on PEG's that can work on recursive descent parsers and their syntax. The machine focuses on the pined point of labeled errors and results in working out the bull's eye (labeled errors) by lowering the number of syntax errors. Ford et. al. [34] discussed CFG's and PEG's. Their study focused on PEG's regular expression grammar which works on a priority basis and is expressive. Their work further provides a parser that is built on PEG's and is of linear time complexity that also over achieves gaps of generalized CFG's parsers. Loff et. al. [35] explained PEG's and palindrome's language. They proposed a mathematical model that

proves the computational complexity of PEG. Further, PEG's are universally commutative, with no pumping lemma and turning machine whose input stack can do computational steps of $O(n/\log n^2)$.

Medeiros et. al. [36] transformed regex structure languages to PEG's by achieving a computational model for regex structure matches. The computational model can transform regular expressions (RE's) to parser expression grammars (PEG's). Blaudeau et. al. [37], instead of working on definitive parsers, worked on parser tree structure that represents correct parsing. By using reference parser, they have developed a different parser that is capable of semantic interpretations though the complexity of both the parsers is somewhat the same. Matsumura et. al. [38] pointed out the fact that most of the syntaxes with PEG's are hard to recognize. Their work represents Nez extension as a Nez extensive language that can overcome hard syntax of PEG with most of the scripting languages and nominating programming languages like C/C++, Python, Ruby, and C# with particle approach.

Andrain [39] investigated whether a language in Chomsky normal form (CNF) be transformed into a different language structure as Greibach normal form (GNF). Thus, their work represents the correct and complete transformation of CNF into GNF and answer the presented question by yes it can be completely and successfully be transformed. Moss et. al. [40] introduced their own new declarative parser algorithms that can parse up to polynomial worst-case scenario and adds to the computability of recursive descent parsers. In another study Moss et. al. [41] presented another algorithm that can parse the PEG's algorithm with worst-case space and rum time complexity of linear time. Lurent et. al. [42] worked with PEG's and recursive descent parser then come up with a set of properties for such type of languages. Properties like disambiguation, unification, composition of both syntactical and lexical. They also add value to PEG's study by explains the explicit phenomenon.

Yamaguchi et. al. [43] worked on tree structures of PEG's and regex language structures. They presented a new PEG structure like regex notations that represent structural constraints. They focus on RE's (regular expressions), XML schemas, and tree automaton formalization. Dawson et. al. [44] presented Automatic Validation Input Data (AVID). Through unambiguity of PEG's, they suggested that operating system vulnerability written in language RUST can be overridden. Born et. al. [45] proved the complexity practice wrong that transformation of synchronous context-free grammars (SCFG's) to synchronous tag adjoining grammars (STAG) doubles the number. Their study

reveals that the complexity of transformation is cubic instead of doubling whatever exists as the complexity number. Tratt et. al. [46] described that PEG's cannot handle left recursion under ambiguous recursive descent parsing and proves that introducing properties of packrat parsing properties by Warth et. al. overcomes the exited problem of not handling direct left recursion. Naveed et. al. [3] worked with terminal prefixing by introducing it as a normal form around CFG and creating a hybrid structure for parser grammars and especially LL(1) grammar. The induced terminal prefixing with LL(1) and achieved a polynomial difference of complexity and induced hybrid structure to achieve its goal with a visible difference in the number of steps for possible inputs.

3 Materials and Methods

This study aims to identify the impact of the induction of Greibach normal form on conventional LL(1) grammar of arithmetic expression. The significance of said transformation will be evaluated with the construction of the parsing table on GNF based LL(1) grammar and analyzed with conventional LL (1) parsing table. Following are the main steps of the proposed research methodology.

- Conversion of LL(1) grammar into GNF based LL(1) grammar.
- Construction of First & Follow Set over the GNF based LL(1) grammar.
- Construction of GNF based LL(1) parsing table.
- Execution of Test Cases on conventional LL(1) parsing table.
- Execution of Test Cases on GNF based LL(1) parsing table.
- Comparative evaluation of the results of step 4 and Step 5 through statistical analysis.

The proposed study is principally concerned with the arithmetic expression grammar; however, the above steps are not only applicable to arithmetic expression grammar, but also any conceivable LL(1) grammar. Figure 1 is an illustration of the proposed research methodology.

3.1 Conversion of LL(1) Grammar into GNF Based LL(1) Grammar

To convert the LL(1) grammar in GNF induced LL(1) grammar, following algorithm is constructed. The algorithm named “conversion algorithm for LL(1) grammar into GNF based LL(1) grammar” in the very first step satisfies the conditions that there cannot be any null or useless production left in the grammar. Step two defines that after the implication of the first rule,

Fig. 1

grammar must remain ambiguous. The algorithm below follows the steps for conventional LL(1) grammar, but is not recommended. First and follow can be found by following different steps for respective grammars.

Algorithm 1: *Conversion algorithm for LL(1) grammar into GNF based LL (1) grammar*

- 1) Start.
- 2) If for all ($\forall$) the symbols (X),
- 3) else there does not exist ($\exists$) a case, where $X = \epsilon$ and $X = a$
- 4) else for all the symbols ($\forall$), there does exist a case ($\exists$) where $X \rightarrow x_1|x_2|x_3|x_4|\dots|x_n$.
- 5) else X is the unambiguous production after satisfying rule 1.
- 6) else The First($X(a)$), the very first symbol of X is 'a'.
- 7) else Follow ($X(b)$), the very second symbol 'b' following the first symbol 'a' of X .
- 8) else for all($\forall$) non-terminals ' X ', then there exists a condition if and only if (*iff*) First(X) = ϵ .
- 9) else then there exists a juncture of First(X) with Follow(X) which equals ϵ as Follow (X) = ϵ .
- 10) else There exists resultant LL (1) grammar with parsed cases.
- 11) else LL (1) grammar induced in GNF.
- 12) else then there exists condition if and only if $\exists$ *iff* $X \rightarrow a$.
- 13) else Where $X \rightarrow a y_1, y_2 \dots y_n$.
- 14) else Where by following GNF conditions "a" is a terminal symbol, followed by any number of variables
- 15) hence $y_1, y_2 \dots y_n$.
- 16) else if Steps 1 to 4 results in parsing table and step 4 results in parsed cases converted under GNF based LL(1) grammar.
- 17) End.

Step 3 is the simple formation of first and follow for every production in the grammar, so that the parser

must know how to perform tasks following the grammar rules. Rule 6 puts forward its role of converting the said LL(1) grammar into the new form of GNF induced LL(1) grammar with steps 7 and 8 making sure GNF structure gets to be followed.

3.2 Construction of First & Follow Set Over the GNF Based LL(1) Grammar

The First set depicts the very first symbol that could appear at the very start of any production p and lead the parser towards performing said tasks of algorithm. The follow set leads the parser to keep on performing production tasks till parser reaches the very end and satisfies the appeared conditions.

Algorithm 2: *First Set*

- 1) Start.
- 2) If 'A' can be a terminal 't' for all of X, then First (A) = {A}.
- 3) else if and only if $A = \varepsilon$ then, ε ends up by adding to First (A).
- 4) else for every non-terminal 'n' of 'A' there exists, $A = B_1, B_2 \dots B_n$.
- 5) else for all of ε , ε adds up to First (B_1), First (B_2) ... First (B_n) than to First(A).
- 6) else there is a condition where A is non-terminal 'n' then, A ends up by adding to production $A \rightarrow B_1, B_2, \dots, B_n$.
- 7) else add 'A' to First (X).
- 8) else If for j^{th} A is First (B_1), then ε is in rest of First (B_1), First (B_2) ... First (B_n).
- 9) End.

The algorithm 'First Set' depicts the application of steps 1 to 3 for start. Then, further steps for First ($B_1, B_2, \dots, B_n$). Further, the ε adds to First(B_1), First(B_2) ... First(B_n). Then, there is a condition where ε which is part of First(B_1) leads to the existence of ε in (B_1), First(B_2) up to First(B_n). Algorithm 3: *Follow Set*

- 1) Start
- 2) If the end symbol is \$ with the start symbol being S then there exists $\$ \in \text{Follow (S)}$.
- 3) else there exists a production $C \rightarrow \beta D \gamma$ then First (γ) can be a subset of Follow(D).
- 4) else with production $C \rightarrow \beta D$, the production $C \rightarrow \beta D \gamma$ results in adding ε to First (γ) then to Follow (C) being a subset of Follow (D)
- 5) End.

The algorithm 'Follow set' works computation of following set with n being non-terminal concerning follow set generation. The computation of the following set depicts where 'n' non-terminal could appear on the right-hand side of any said production.

3.3 Construction of GNF Based LL(1) Parsing Table

Parsing table is a tabular format of the parser. It tells the parser about actions that leads to satisfying the appearing algorithmic conditions. The tabular formations of the parsing table may lead a grammar G in producing productions that could accomplish certain tasks in a specific and required amount of time.

The following algorithm explains the parsing procedure for the parser table. Step 1 of the algorithm named "parsing table algorithm" explains non-terminals N , lookahead t , and the number of productions carrying N , t , and X . step 2 gives knowledge of the parsing table constructed by following the simplest rules, while step 4 explains the total of tuples used within parsing table. Step 5 points to non-terminal N and lookahead t and step 6 begins for blank spaces mentioning the error.

- 1) Start
- 2) If For the parser to use non-terminals N , with lookahead t and productions.
- 3) else there exists a condition $\exists X$
- 4) else for all X such that with nonterminal 'N' and lookahead symbol 't' there is $\forall X | (N, t)$.
- 5) else $\forall X | (N, t)$ produces the number of steps for the algorithm to work.
- 6) 6. else Above-mentioned LL (1) Arithmetic Grammar for Parsing Table (X').
- 7) else there also exists X' for all nonterminal 'N' and lookahead symbol 't' then $\exists X' | \forall X' (N, t)$.
- 8) else there exists $\exists' N$ (non-terminal) with t (lookahead) $\forall X'$.
- 9) else if $\forall X'$ blank spaced entries being errors.
- 10) End.

3.4 Execution of Test Cases on Conventional LL(1) Parsing Table

Test cases of approximately 400 in several different combinations according to LL(1) parser are tested under the execution of said algorithms. The cases are to confirm the proper execution capability of the LL (1) based conventional parser form. The test case input stack also varies in size to test the LL (1) parser limits.

3.5 Execution of test Cases on GNF Based LL(1) Parsing Table

Test cases used for LL(1) parser perspective are also used for GNF based LL(1) parser to check out the parsing capability of GNF induced LL(1) parser. The test case input stack is also the same in number and size to match the exact limit parser has achieved after induction.

$S \rightarrow \text{expr}$
$\text{expr} \rightarrow \text{term expr}'$
$\text{expr}' \rightarrow + \text{expr}$
$\text{expr}' \rightarrow - \text{expr}$
$\text{expr}' \rightarrow \epsilon$
$\text{term} \rightarrow \text{factor term}'$
$\text{term}' \rightarrow * \text{term}$
$\text{term}' \rightarrow / \text{term}$
$\text{term}' \rightarrow \epsilon$
$\text{factor} \rightarrow \text{num}$
$\text{factor} \rightarrow \text{id}$

TABLE 1: LL (1) Arithmetic Grammar

3.6 Comparative Evaluation

In this phase, the results obtained from Section 3 and Section 4 will be analyzed. The first phase in the evaluation is to check the normality of data which in this case is the steps involved in the observed data, the statistical methods like T-test and U-test are used to analyze the results.

4 Evaluation and Discussion

After the text edit has been completed, the paper is ready for the template. Duplicate the template file by using the Save As command, The steps defined in Section 3 are applied to the LL (1) grammar of arithmetic expression which is shown in Table 1.

The grammar above accompanies tuples terminal symbols, non-terminal symbols, epsilon ϵ productions. The terminal symbols are $+$, $-$, $*$, $/$, id , and num where non-terminals are expr , term , expr' , term' , and factor . The epsilon ϵ in the grammar neglects the loop production scenarios. The grammar above remains unambiguous as it does not hold any non-necessary unit production, not reachable useless productions, and ϵ -free production where ϵ does not cancel out the whole production set and put LL (1) parser at halt state. The LL (1) arithmetic grammar holds the condition of non-left recursion and as it seems that there are not any such rules in the mentioned grammar that generates left-recursions.

The First and Follow of LL (1) grammar of arithmetic and the expression are shown in Table 2.

Here appendix 1 is the parsing table for above mentioned conventional LL (1) arithmetic expression grammar. The parsing table is a tabular depiction of the parser execution under-studied grammar. Parsing appendix 1 of the conventional LL(1) arithmetic expression grammar is mentioned in an appendix because of its size and readability.

First	Follow
$\{\text{num}, \text{id}\}$	$\{\$ \}$
$\{\text{num}, \text{id}\}$	$\{\$ \}$
$\{\epsilon, +, -\}$	$\{\$ \}$
$\{\text{num}, \text{id}\}$	$\{+, -, \$ \}$
$\{\epsilon, *, /\}$	$\{+, -, \$ \}$
$\{\text{num}, \text{id}\}$	$\{+, -, *, /, \$ \}$
$\{\text{id}\}$	_____
$\{\}$	_____
$\{*\}$	_____
$\{/ \}$	_____
$\{+\}$	_____
$\{-\}$	_____

TABLE 2: First and follow table for LL(1) grammar

$S \rightarrow \text{num term}' \text{expr}'$
$S \rightarrow \text{id term}' \text{expr}'$
$\text{expr} \rightarrow \text{num term}' \text{expr}'$
$\text{expr} \rightarrow \text{id term}' \text{expr}'$
$\text{expr}' \rightarrow + \text{expr}$
$\text{expr}' \rightarrow - \text{expr}$
$\text{expr}' \rightarrow \epsilon$
$\text{term} \rightarrow \text{num term}'$
$\text{term} \rightarrow \text{id term}'$
$\text{term}' \rightarrow * \text{term}$
$\text{term}' \rightarrow / \text{term}$
$\text{term}' \rightarrow \epsilon$
$\text{factor} \rightarrow \text{num}$
$\text{factor} \rightarrow \text{id}$

TABLE 3: GNF induced LL (1) grammar

4.1 GNF Based Arithmetic Expression

The grammar in Table 3 holds a newly structured grammar that is formulated by inducing GNF to the existing LL(1) grammar structure. The productions of grammar below carry syntactical formation of typical Greibach normal form (GNF) which is the terminal symbol followed by any number of said variables.

Construct 3 parsing trees for some expressions (3) with conventional grammar and with GNF based grammar. Present the trees in comparative form. Discuss each pair of parsing trees (1st of conventional grammar – 2nd of GNF based grammar) by discussing their node, levels, and depths. Below are the parsing trees of some input strings parsed under conventional LL(1) grammar and GNF based conventional LL(1) grammar as in comparison.

Figure 2(a) and Figure 2(b) show the parsing tree of input string “id + id”, but Figure 2(a) is the parsing tree of conventional LL(1) grammar with root as ‘S’ and leaf nodes are “id, ϵ , +, id, ϵ , ϵ ”, where inner nodes are “expr, term, expr’ and factor”. Figure 2(b) is the parsing tree of the same input string, but parsed with GNF based conventional LL (1) grammar through root as ‘S’ and leaf nodes are “id, ϵ , +, id, ϵ , ϵ ” along with

Fig. 2a: Parsing tree of “id + id” with conventional LL(1) grammar

Fig. 3b: Parsing tree of “id+id+num” with GNF based conventional LL (1) grammar

Fig. 2b: Parsing tree of “id +id” with GNF based conventional LL(1) grammar

Fig. 4a: Parsing tree of “id+id+num+num” with conventional LL(1) grammar

inner nodes as “term’, expr’, expr”.

The number of nodes in the parsing tree shown in Figure 2(a) in level 7, depth 6 is $2d$, where d is depth; hence, the total number of nodes in Figure 2(a) parsing tree is 64. Figure 2(b) carries a parsing tree of level 5, depth 4, and the total number of nodes as 16.

Figure 3(a) and Figure 3(b) both are the parsing

Fig. 3a: Parsing tree of “id+id+num” with conventional LL (1) grammar

tree of input string “id + id+ num” but Figure 3(a) is the parsing tree of conventional LL(1) grammar with root as ‘S’ and leaf nodes are “id, ϵ , +, id, ϵ , +, num, ϵ , ϵ ” where inner nodes are “expr’, term, expr, term’ and factor”. Figure 3 (b) is the parsing tree of the same input string but parsed with GNF based conventional LL(1) grammar thru root as ‘S’ and leaf nodes are “id, ϵ , +, id, ϵ , +, num, ϵ , ϵ ” along with inner nodes as “term’, expr”.

The number of nodes in the parsing tree shown in Figure 3(a) in level 7, depth 6 is $2d$, where d is depth; hence, the total number of nodes in Figure 3(a) parsing tree is 64. Figure 3(b) carries a parsing tree of level 5, depth 4, and the total number of nodes as 16.

Figure 4 (a) and Figure 4 (b) both are the parsing tree of input string “id + id+ num+ num” but Figure 4 (a) is the parsing tree of conventional LL (1) grammar with root as ‘S’ and leaf nodes are “id, , +, id, ,+,num,,+,num,,” where inner nodes are “expr’, term, expr, term’ and factor”. Figure 4 (b) is the parsing tree of the same input string but parsed with GNF based conventional LL (1) grammar thru root as ‘S’ and leaf

Fig. 4b: Parsing tree of “id+id+num+num” with GNF based conventional LL(1) grammar

FIRST	FOLLOW
{num,id}	{ $\$$ }
{num,id}	{ $\$$ }
{+, -, ϵ }	{ $\$$ }
{num,id}	{+, -, $\$$ }
{*, /, ϵ }	{+, -, $\$$ }
{num,id}	{undefined}

TABLE 4: 3 first & follow sets

nodes are “id, ϵ , +, id, ϵ , +, num, ϵ , +, num, ϵ , ϵ ” along with inner nodes as “term’, expr’, expr”.

The number of nodes in the parsing tree shown in Figure 4(a) in level 7, depth 6 is $2d$, where d is depth; hence, the total number of nodes in Figure 4(a) parsing tree is 64. Figure 4(b) carries a parsing tree of level 5, depth 4, and the total number of nodes as 16.

4.2 First & Follow Sets

Table 4 shows a set of first and follow of GNF induced LL(1) grammar mentioned in Table 3.

4.3 Construction of GNF Based LL(1) Parsing Table

Here follows the parsing table of GNF induced LL(1) grammar. The parsing table is of the grammar mentioned in Appendix 2.

5 Results & Discussion

During the study, 399 arithmetic expressions are evaluated with conventional LL(1) parser and parsed with GNF based LL(1) parser. The steps performed during the parse of analyzed and descriptive statistics are shown in Appendix 3. Descriptive statistical analysis in Appendix 3 provides information regarding the Mann-Whitney U test. Firstly, the reason for descriptive statistics is to compare the groups (LL(1) and

Fig. 5: Scatter diagram of steps

Fig. 6: Boxplot of step involved during parsing

GNF (GNF Induced LL(1) grammar)). The mean, median, standard deviation, skewness, and Kurtosis are mentioned in the Table 4. For further illustration, Boxplot is defined on the steps taken parsers and the result is shown in Figure 6. The above Boxplot inferred the number of steps taken by each grammar to parse input strings. Figure 6 depicts the GNF induced LL(1) (GNF) taking a smaller number of steps to parse input string than the one by conventional LL(1) parser.

For point-to-point analysis of LL(1) and GNF based LL(1) parsers, the steps taken by these parsers are shown with a scatter diagram (Figure. 5).

The scatter diagrams show that the GNF induced LL(1) takes fewer steps in execution than conventional LL(1). Normality tests are conducted on the steps taken by both parsers during checking the arithmetic expression, and the results are shown in Table 5.

The normality test of grammar (GNF) was assessed and the Shapiro-Wilk test indicated that the score was $W(399) = .000$. The normality test of grammar LL(1) was also assessed and the Shapiro-Wilk test indicated that $W(399) = 0.000$. To analyze the data arrangement, normality tests are conducted. The

Tests of Normality							
		Kolmogorov-Smirnov			Shapiro-Wilk		
Steps	Grammar	Statistic	Df	Sig.	Statistic	Df	Sig.
	GNF	.077	399	.000	.979	399	.000
	LL (1)	.070	399	.000	.979	399	.000

TABLE 5: Tests of normality

Parser	N	Mean Rank	Sum of Ranks
LL (1)	399	468.07	186758.00
GNF based LL (1)	399	330.93	132043.00
Total	798		

TABLE 6: Ranks

above tests determine the range of normal distribution compiled from random samples to original data. Mann-Whitney Test is conducted on the steps involved during parsing and ranks are shown in Table 6.

The test statistics of the Mann-Whitney Test are shown in table 7.

Table 7 shows the scores of grammars (LL(1)) Mdn =60.00 was higher than (GNF Based LL(1)) Mdn =43.00. A Mann-Whitney test indicated that this difference was statistically significant. The other values include $U(N_{LL(1)}) = 399$, $N_{GNFBasedLL(1)} = 399$ =52243.000, $Z = -8.404$, and $p < .001$.

The proposed GNF based LL(1) algorithm and the conventional LL(1) algorithm works side by side, but GNF Induced LL(1) parser takes fewer steps to execute input string as compared to conventional LL (1) parser. GNF based LL(1) parser's execution time develops linearly with the number of steps involved in executing input string. The execution time is directly proportional to the number of steps involved in completing the whole process.

The proposed GNF induced LL(1) algorithm reaches out to Big O worst-case linear complexity. Both the algorithms conventional LL (1) and GNF induced LL(1) algorithm performs under the worst-case state of $O(N)$. The number of execution steps matches the iterations needed to complete a function, but Big O notation always looks for an upper limit where the algorithm could execute at its maximum state. Hence, both the algorithms are dependent linearly.

Tests	Steps
Mann-Whitney U	52243.000
Wilcoxon W	132043.000
Z	8.404
Asymp. Sig. (2-tailed)	.000

TABLE 7: Test statistics

6 Conclusion

In this paper, we arranged grammar in two sets, i.e., onventional LL(1) and GNF based LL(1), and arranged data of 399 input stack among four cases of a different combination of values for each set of grammar. By analyzing data through statistical methods, we found that if conventional LL (1) parser is combined with GNF, the newly generated from that is GNF based LL(1) parser can work more effectively than the conventional LL(1) parser.

The induction is set to achieve parser efficiency and more often for granted grammar. This study adds to some knowledge in the respective field. The top-down parsers are factors that limit the parsers, as there is no back-tracking which involves chances of the algorithm to go in halt state and with LL(1) structure there comes limitation with single lookahead symbol and no-left recursion structure of the grammar syntax. To improve this study, a comparison can be made between two or more inductions of such kind to achieve a method that could improve the parser's working capability regarding grammars.

References

- [1] C.J. Grune, "Parsing Techniques A Practical Guide", VU University, Amsterdam, 1990.
- [2] S. A. Greibach, "A new normal-form theorem for context-free phrase structure grammars", Journal of the ACM (JACM), vol. 12(1), pp. 42-52, 1965.
- [3] M. S. Naveed, "The Impact of Terminal Prefixing on LL (1) Parsing", J. Appl. Environ. Biol. Sci, vol. 7, No. 5, pp. 64-76, 2017.
- [4] T. Parr, et al., "Adaptive LL (*) parsing the power of dynamic analysis", ACM SIGPLAN Notices, vol. 49, No. 10, pp. 579-598, 2014.
- [5] R. K. Aggarwal, "Improving Hindi Speech Recognition Using Filter Bank Optimization and Acoustic Model Refinement", 2018.
- [6] N. Fischer, et al., "An efficient insertion-only error-corrector for LL (1) parsers", In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming language, pp. 97-103, 1977.
- [7] Ghezzi, "LL (1) grammars supporting an efficient error handling". Information Processing Letters, vol. 3, No. 6, pp. 174-176, 1975.
- [8] T.J. Parr, and R. W. Quong, "LL and LR translators need $k_L \geq 1$ lookahead", ACM Sigplan Notices, vol. 31, No. 2, pp. 27-34, 1996.
- [9] Slivnik, "LL conflict resolution using the embedded left LR parser", ComSIS, vol. 9(3), 2012.
- [10] K. Yamaguchi, and Kuramitsu, "CPEG a typed tree construction from parsing expression grammars with regex-like captures", In Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, pp. 1526-1533, 2019.
- [11] R. Edelmann, et al., "LL (1) Parsing with Derivatives and Zippers", arXiv preprint arXiv:1911.12737, 2019.
- [12] T.J. Parr, and R. W. Quong, "LL and LR translators need $k_L \geq 1$ lookahead", ACM Sigplan Notices, vol. 31, No. 2, pp. 27-34, 1996.

Nonterminal	+	-	*	/	num	id	\$
S					S → expr	S → expr	
expr					expr → term expr'	expr → term expr'	
expr'	expr' → + expr	expr' → - expr					expr' → ϵ
term					term → factor term'	term → factor term'	
term'	term' → ϵ	term' → ϵ	term' → * term	term' → / term			term' → ϵ
factor					Factor → num	factor → id	

Appendix - 1

Nonterminal	Num	id	+	-	*	/	\$
S	S → num term' expr'	S → id term' expr'					
expr	expr → num term' expr'	expr → id term' expr'					
expr'			expr' → + expr	expr' → - expr			expr' → ϵ
term	term → num term'	term → id term'					
term'			term' → ϵ	term' → ϵ	term' → * term	term' → / term	term' → ϵ
factor	factor → num	factor → id					

Appendix - 2

Parser	Std. Dev.	Min	Max	Range	Skewness		Mean	Median	Variance
LL (1)	20.521	9	119	110	.296	.112	45.77	43.00	421.101
GNF induced LL (1) parser	27.532	13	161	148	.323	.122	62.14	60.00	758.022

Appendix - 3

- [13] R.G. Taylor, “LL parsing, LR parsing, complexity, and automata”, ACM SIGCSE Bulletin, vol. 34, No. 4, pp. 71-75, 2002.
- [14] Z. Nguyen, et al., “Design patterns for parsing”. ACM SIGCSE Bulletin, vol. 37, No. 1, pp. 477-481, 2005.
- [15] T. Parr, and K. Fisher, “LL (*) the foundation of the ANTLR parser generator”, ACM Sigplan Notices, vol. 46, No. 6, pp. 425-436, 2011.a
- [16] Slivnik, “LLL Parsing a Combination of LL and LR Parsing”, In 5th Symposium on Languages, Applications and Technologies (SLATE'16), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, vol. 51, pp. 1-13, 2016.
- [17] S. Sippu, and E. Soisalon-Soininen, “On LL (k) parsing”, Information and Control, vol. 53, No. 3, pp. 141-164, 1982.
- [18] Afrozeh, A and Izmaylova, “Faster, practical GLL parsing”. In International Conference on Compiler Construction. Springer, vol. 9031, pp. 89-108, 2015.
- [19] M. J. Bertsch, and Nederhof, “Gap Parsing with LL (1) Grammars”, Grammars, vol. 8, pp. 1-16, 2005.
- [20] Scott, and A. Johnstone, “GLL parsing”. Electronic Notes in Theoretical Computer Science, vol. 253, No. 7, pp. 177-189, 2010.
- [21] J. J. Shilling, “Incremental LL (1) parsing in language-based editors”, IEEE Transactions on Software Engineering, vol. 19, No. 9, pp. 935-940, 1993.
- [22] K. Yoshida, and Y. Takeuchi, “Some Properties of an Algorithm for Constructing LL (1) Parsing-Tables Using Production Indices”, Journal of Information Processing, vol. 11, No. 4, pp. 258-262, 1988.
- [23] S. Lasser, et al. “A Verified LL (1) Parser Generator”, In 10th International Conference on Interactive Theorem Proving (ITP 2019), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, vol. 141, pp. 1-13, 2019.
- [24] P. Singal, et al., “Parse Condition Symbolic Encoding of LL (1) Parsing”, In LPAR, vol. 57, pp. 637-655, 2018.
- [25] J. A. Femister, “LPARSER an LL (1) Parser Generator”, 1986.
- [26] A.V. Goloveshkin, “Tolerant parsing using modified LR (1) and LL (1) algorithms with embedded Any symbol”, , vol. 31, No. 3, pp. 7-28, 2019.
- [27] Nijholt, “Strict deterministic grammars and Greibach normal form”, Elektronische Informationsverarbeitung und Kybernetik, vol. 15(8/9), pp. 395-401, 1979.
- [28] J. Hopcroft, et al., “On the reachability problem for 5-dimensional vector addition systems”, Theoretical Computer Science, vol. 8, No. 2, pp. 135-159, 1979.
- [29] S. Medeiros, and F. Mascarenhas, “A parsing machine for parsing expression grammars with labeled failures”, In Proceedings of the 31st Annual ACM Symposium on Applied Computing, pp. 1960-1967, 2016.
- [30] S. Medeiros, and F. Mascarenhas, “Syntax error recovery in parsing expression grammars”, In Proceedings of the 33rd Annual ACM Symposium on Applied Computing, pp. 1195-1202, 2018.
- [31] S. C. Reghizzi, and P. San Pietro, “Non-erasing Chomsky-Schützenberger theorem with grammar-independent alphabet”, Information and Computation, vol. 269, pp. 104-442, 2019.
- [32] S. Mascarenhas, et al., “On the relation between context-

- free grammars and parsing expression grammars”, *Science of Computer Programming*, vol. 89, pp. 235-250, 2014.
- [33] S. Medeiros, et al., “Left recursion in parsing expression grammars”, *Science of Computer Programming*, vol. 96, pp. 177-190, 2014.
- [34] Ford, “Parsing expression grammars: a recognition-based syntactic foundation”, In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 111-122, 2004.
- [35] N. Loff, et al. “The computational power of parsing expression grammars”, *Journal of Computer and System Sciences*, vol. 111, pp. 1-21, 2020.
- [36] S. Medeiros, et al., “From regexes to parsing expression grammars”, *Science of Computer Programming*, vol. 93, pp. 3-18, 2014.
- [37] C. Blaudeau, & N. Shankar, “A verified packrat parser interpreter for parsing expression grammars”, In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 3-17, 2020.
- [38] T. Matsumura, and K. Kuramitsu, “A declarative extension of parsing expression grammars for recognizing most programming languages”, *Journal of Information Processing*, vol. 24, No. 2, pp. 256-264, 2016.
- [39] R. Andrian and A.Y.C. Dabukke, “Aplikasi Pengubah Bentuk Normal Chomsky Menjadi Bentuk Normal Greibach Dengan Metode Substitusi”, *Jurnal Komputasi*, vol. 3, No. 2, 2016.
- [40] Moss, “Derivatives of parsing expression grammars”, *arXiv preprint arXiv:1405.4841*, 2014.
- [41] Moss, “Simplified parsing expression derivatives”. In *International Conference on Language and Automata Theory and Applications*, Springer, Cham, pp. 425-436, 2020.
- [42] N. Laurent, and K. Mens, “Parsing expression grammars made practical”. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pp. 167-172, 2015.
- [43] D. Yamaguchi, & K. Kuramitsu, “CPEG: a typed tree construction from parsing expression grammars with regex-like captures”, In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pp. 1526-1533, 2019.
- [44] M. D. Dawson, “Secure input validation in Rust with parsing-expression grammars (Doctoral dissertation, Massachusetts Institute of Technology)”, 2019.
- [45] L. Born and A. Sarkar, “Prefix Lexicalization of Synchronous CFGs using Synchronous TAG”. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, vol. 1, pp. 1160-1170, 2010.
- [46] L. Tratt, “Direct left-recursive parsing expression grammars”, *School of Engineering and Information Sciences*, Middlesex University, 2010.