Pushing down Context Free Grammars

Danila Bren
d.a.bren@student.utwente.nl
University of Twente
Enschede, Netherlands

ABSTRACT

Grammars and parsing algorithms for those grammars are widely used to create and work with programming languages. While most languages use standard form of grammars like Context Free Grammars, some can be parsed faster using algorithms for Visibly Pushdown Grammars. While there has been effort in creating parser generators for VPGs, the question of conversion from CFGs to VPGs remains. We propose to create a tool that will perform operations that will make CFGs more suitable for conversion by existing tools, as well as potentially find what specific properties of CFGs may be a reason to claim that a grammar cannot be converted.

KEYWORDS

CFG, TCFG, VPG, ANTLR

1 INTRODUCTION

In 1956 Noam Chomsky introduced first classification of formal languages and gave a mathematical model of a grammar, giving start to a subarea of computer science. The concept of grammars was found to be of importance for programming languages with ALGOL being defined by a Context-Free Grammar. Since then, parser generator tools have been developed such as ANTLR [17] that works with a class of Context Free Languages. In parallel with that, a lot of research has been done to refine the first classification of languages and define sub classes in Chomsky hierarchy to achieve better performance while still preserving expressiveness necessary to define some of the languages [8] . In this paper we will zoom in on class of Visibly Pushdown Languages [1], a subclass of Context Free Languages. Specifically, we will explore possibilities to automatically transform Context Free Grammars to its Visibly Pushdown equivalent.

Chomsky hierarchy [3] has four main classes in it:

- Type-0: Recursively enumerable languages.
- Type-1: Context-sensitive languages.
- Type-2: Context-free languages.
- Type-3: Regular languages.

Languages in this hierarchy, going from regular to recursively enumerable, become more expressive and harder to work with: Context-sensitive languages have exponential parsing algorithm complexity and parsing and recognition in recursively enumerable realm is undecidable.

Context-Free Grammars are used to generate parsers by tools like ANTLR [17] and BISON [14]. However CFGs have limitations in terms of efficiency and verification: not all CFGs can be converted to deterministic pushdown automata (PDA) and worst-case running time of general CFG-based parsing algorithms is $O(n^3)$ [9]. In 2004 Rajeev Alur and P. Madhusudan suggested a class of Visibly Pushdown languages [1]. VPLs are a subclass of Context-Free Languages and a superclass of regular languages, which places them between the two classes. Intuitively, this makes VPLs more expressive then regular, while allowing for faster parsing than CFGs. The most trivial example of a VPL is Dyck language of balanced brackets [16], a language that consist of open and closed brackets for instance "(())", "()()". With brackets added to regular expressions it is possible to express real languages like JSON, XML or HTML. In their 2023 paper Jia et al [9] discuss parser generator for visibly pushdown grammars that they have developed, that is faster than CFG based parsers: complexity of their parser is $O(n)$, compared to worst case complexity of parsers for CFL of $O(n^3)$. To test their parser, Jia et al converted some grammars written for ANTLR [6], However transformation of CFGs to VPGs was not the main focus of their paper. In the discussion they write "In general, it is an open problem to determine whether a CFG can be translated to a VPG". To further explore possibilities to optimize parsing of known languages in we want to explore that problem and see if it is possible to write an algorithm that would take a Context Free Grammar and, if possible, convert it to its Visibly Pushdown equivalent. Our work will consist of two components:

- Tagging of Context Free Grammars.
- Transforming CFGs into an equivalent VPG.

To do that we chose to work with the same repository of grammars for ANTLRv4 [6] to be able to compare our results with those of Jia et al and due to availability of parsing framework. First we will give formal definition to Visibly Pushdown Grammars, introduce features of ANTLR that we will be using and introduce form of grammars we will be working with. Note that ANTLR does not have support for tagging of grammars, because of that resulting grammars will not be in the same form.

1.1 Background: EBNF and ANTLR features

Exctended Backus-Naur Form is a formal definition of syntax used for defining languages and many other formal definitions in computer science [13]. There exist many variations of EBNF using different notations to express same things. To denote repetitions ANTLR uses 3 differnt suffixes: question mark *"?"* for 0 or 1 repetitions, *Kleene star "*"* for 0 or more repetitions and plus + for 1 or more repetitions. In our resulting VPG grammars we will be

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
41st Twente Student Conference on IT, July 5, 2024, Enschede, The Netherlands
© 2024 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

using the same notation. In ANTLR it is possible to group symbols using brackets, but in our grammars we will avoid grouping by brackets and will instead create an additional intermediate rule. Further, ANTLR grammar may include semantic and syntactic predicates [19], and context free actions. Syntactic predicates are given as a grammar fragment that must match the following input. Semantic predicates are given as arbitrary Boolean-valued code in the host language of the parser. Actions are written in the host language of the parser and have access to the current state. ANTLR requires programmers to avoid left recursion [20], ANTLRv4 is able to eliminate direct left recursion, but will fail at runtime when parsing with grammars that have non immediate left recursion.

1.2 Background: Visibly Pushdown Grammar

As a class of grammars, VPGs [1] compared to CFGs have many good properties. Languages of VPGs are a subset of deterministic context-free languages, which means that it is always possible to build a deterministic PDA from a VPG. The terminals in VPGs are partitioned in three subsets: *call*, *plain*, and *return* and stack actions associated with the symbol is determined by which of the three subsets the symbol is in. An action of pushing is performed for *call* symbols, an action of popping is performed for *return* symbols and there is no action associated with *plain* symbol. As shown in Jia et al paper [9]. VPGs enable building of linear-time parsers, and suck parsers are amenable to formal verification.

Below is a formal definition of VPGs [1]. A grammar G is represented as a tuple $(V, \sum, P, L_0)$, where V is the set of nonterminals, $\sum$ is the set of terminals, P is the set of production rules, and $L_0 \in V$ is the start symbol. The alphabet $\sum$ is partitioned into three sets: $\sum_{plain}, \sum_{call}, \sum_{ret}$. Notation-wise, a terminal $\sum_{call}$ is tagged with $\langle$ on the left, and a terminal in $\sum_{ret}$ is tagged with $\rangle$ on the right.

Below is a formal definition of *well-matched VPGs*. Such VPGs generate only well-matched strings in which a call symbol is always matched with a return symbol.

Well-matched VPGs 1. *A grammar $G = (V, \sum, P, L_0)$ is a well-matched VPG with respect to the partitioning $\sum = \sum_{plain} \cup \sum_{call} \cup \sum_{ret}$ if every production rule in P is in one of the following forms:*

(1) *$L \longrightarrow \epsilon$, where ϵ stands for the empty string;*
(2) *$L \longrightarrow cL_1$, where $c \in \sum_{plain}$;*
(3) *$L \longrightarrow \langle aL_1b\rangle L_2$, where $\langle a \in \sum_{call}$ and $b\rangle \in \sum_{ret}$.*

A superclass of well-matched VPGs is general VPGs. General VPGs can specify substrings of well-matched strings. General VPGs allow pending *call* and *return* symbols by introducing additional form of rules $L \longrightarrow \langle aL'$ and $L \longrightarrow b\rangle L'$. The set of nonterminals V is partitioned into $V^0 and V^1$: nonterminals in V^0 can only generate well-matched strings, while nonterminals in V^1 can generate strings with pending symbols.

General VPGs 1. *A grammar $G = (V, \sum, P, L_0)$ is a general VPG with respect to partitioning $\sum = \sum_{plain} \cup \sum_{call} \cup \sum_{ret}$ and $V = V^0 \cup V^1$ if every rule in P is in one of the following forms:*

(1) $L \longrightarrow \epsilon$
(2) *$L \longrightarrow iL_1$, where $i \in \sum$, and if $L \in V^0$ then (1) $i \in \sum_{plain}$ and (2) $L_1 \in V^0$;*
(3) *$L \longrightarrow \langle aL_1b\rangle L_2$, where $\langle a \in \sum_{call}$, $b\rangle \in \sum_{ret}$, $L_1 \in V^0$, and if $L \in V^0$, then $L_2 \in V^0$.*

Note that terminals that are tagged as either call or return cannot occur with an EBNF suffix. Intuitively, if that was not the case, it would be impossible to claim that there will be the same amount of *call* and *return* symbols.

1.3 Grammar transformations

In general problem of equivalence of two different languages defined by two different grammars is considered undecidable. However, quite often grammars can be equivalently converted into a form feasible for deterministic analysis preserving the language defined by the old grammar form [5]. One of such equivalent transformations is removing of left recursion [20]. "In theory, the restriction to non-left-recursive CFGs puts no additional constraints on the languages that can be described, because any CFG can in principle be transformed into an equivalent non-left-recursive CFG" [15]. However, it can prove to be a problem for parsers that process text top-down, left-to-right. Suppose we are trying to parse an input string with rule $L \longrightarrow La$ at a given position of input. Using left-to-right parsing our first goal will be to parse rule L at the same position, which will put parser in an infinite loop. The standard algorithm for removing left recursion is attributed to M. C. Panll by Hopcroft and Ullman [7]. Robert C. Moore tested the algorithm suggested by Panll, compared it to other existing algorithms and suggested improvements to it [15]. The algorithm that is being compared to in their work is Left-Corner Transform, presented by Johnson [12]. The original purpose of the LR transform is to allow simulation of left-corner parsing by top-down parsing, but it eliminates left-recursion from CFGs as well. "Furthermore, in the worst case the total number of nonterminal symbols in a grammar transformed using LR cannot exceed a fixed boundary of multiple of square of number of symbols in the original grammar, in contrast to Paull's algorithm, which exponentiates the size of grammar in the worst case" [15].

2 EXPECTATIONS AND GOALS

In their paper Jia et al [9] have used Grammars-v4 repository [6] to retrieve Context Free Grammars, their tool only accepted grammars free of actions and semantic predicates. They claim to have been able to convert 136 grammars out of 239. They say that at least 34 more grammars could not be converted because of left recursion [20]. In our research we will use the same repository and we will attempt to write an algorithm that would be able to automatically separate set of terminals $\sum$ into its subsets $\sum_{call}, \sum_{ret}$ *and* $\sum_{plain}$. We will also write an algorithm that, if possible, would transform a grammar into its well-matched VPG equivalent. Our expectation is that we will be able to transform at least 170 grammars, which is the amount that Jia et al. claim to be feasible if limitations known to them are eliminated. If successful, we will attempt to push this number even further by finding and eliminating even more structural limitations.

3 METHODOLOGY

For each grammar the algorithm applies following operations in the order below:

(1) Parse a grammar using ANTLRv4 parser generator [18] with an ANTLRv4 grammar for ANTLRv4 grammars [6]. Note

that we are not working with lexers, and therefore lexers are ignored when parsing. Semantic and syntactic predicates are ignored when parsing grammars, and grammars with actions are removed from the testing set.

(2) Refactor grammars to remove groups of symbols. For instance rule $L \longrightarrow A(B|C)$ is transformed into rules $L \longrightarrow ANR$ and $NR \longrightarrow B \mid C$. The new rules created during this transformation will be named "_new_rule_N".
(3) Remove Left Recursion: Next step was to check if a grammar has left recursion and if it does, to remove it. Hypothetically, this step can be done after tagging as well, as it is an equivalent transformation.
(4) Attempt to tag grammar with one of the methods. The methods will be discussed in the next section 4.1.
(5) Evaluate the grammar by checking its structure given the tagging and conclude if this grammar is in VPG form.

3.1 Tagging

In this section we will discuss approaches that we used to tag grammars and their efficiency in detecting *call* and *return* symbols. To compare results of applying different heuristics we first tagged each grammar by iterating over ll possible pairs of elements, assuming that any two could potentially be a pair, and then check if, with that pair tagged as *call* and *return*, all rules in the grammar are well-matched. Intuitively, this approach has a very high complexity, as structure of rules will have to be checked n^2 times, where n is the number of distinct terminals. When tagging Context Free Grammars, terminals that occurred with an EBNF suffix at least once will automatically be tagged as *plain*. For the heuristics heuristics that will be discussed below, we assumed that rules in the grammar have that both *call* and *return* symbols will both be on the right side of a rule. That is, we will be treating rules as though they are written in well-matched form. We tried following methods to tag grammars:

- Tagging by precedence: when going through the grammar we counted how many times each pair of symbols occurred in the grammar as suitable candidates for *call-return* pair. At the end if there were conflicts in the resulting tagging, the pair that had less occurrences would be removed. For instance if we found pairs $(\langle a; b\rangle)$ that occurred 5 times and a pair $(\langle c; b\rangle)$ that occurred 3 times in a grammar, symbol b is a conflicting symbol and the resulting tagging would be $\sum_{call} = \{\langle a\rangle \ \sum_{ret} = \{b\rangle\}$. This approach allows to easily exclude faulty tagging.
- Tagging by distance and structure: one of the heuristics we tried was to limit the amount of symbols (both terminals and nonterminals) that is allowed to be between a *call-return* pair. The idea behind it is that potentially people writing grammars may have written them in a form close to well-matched VPG because they wanted to keep some consistent structure, for optimization reasons or by accident. That would mean that a rule with a *call-return* could not start with a nonterminal and would only have one *call-return* pair in it. If that was the case, such tagging algorithm would have lower complexity compared to tagging by precedence as we would not have to check for conflicting pairs. However, after trying it on a few grammars we realized that such assumption is not realistic: rules often have multiple pairs on the right hand side and in general grammars rarely have a common structure shared with each other.
- To explore this idea further, we attempted to tag grammars so that, when possible, a *call-return* pair would have a nested pair and compared it to approach when we avoid nested pairs. That is, rule $L \longrightarrow a\ L_1\ b\ L_2\ c\ L_3\ d$ could be tagged in two ways: (1) $\sum_{call} = \{a,\ b\}$, $\sum_{ret} = \{c,\ d\}$ or (2) $\sum_{call} = \{a,\ c\}$, $\sum_{ret} = \{b,\ d\}$. Even though the latter tagging allows for easier transformation of a rule into its well-matched VPG equivalent form, on practice tagging is usually not intuitive as for some grammars different open brackets would be grouped as a pair. In conclusion, this heuristic did not prove reliable to use for tagging of grammars, however, it helped us to make a useful conclusion on how to tag rules that do not have unique tagging.

3.2 Evaluation methodology

Our original idea on evaluation was to use the tool for grammar conversion introduced by Jia et al [9]. The converter described in their paper accepts a tagged CFG and converts it to a VPG in three steps. $AtaggedCFG \longrightarrow Simple\,form \xrightarrow{\text{If valid}} Linear\,form \longrightarrow VPG$ However, there were some limitations in setting up the tool to work with any grammars, and we had to change our approach. Our current solution is to perform the steps described by Jia et al up and including checking if *simple form* of a grammar is valid.

A grammar is in simple form if all rules of that grammar are in simple form. A rule is in simple form if it is in one of the forms (1) $L \longrightarrow \epsilon$ or (2) $L \longrightarrow q_1 ... q_k$, where $q_i \in \sum_{plain} \cup V$ or $q_i = \langle aL`b\rangle$ for some $\langle a, b\rangle and L`$ and $L`$ is a nonterminal. In other words, there should be one and only one nonterminal between any *call-return* pair. The conversion is straightforward: for each rule, replace every string $\langle asb\rangle$, where $s \in (\sum \cup V)^*$ with $\langle aL_s b\rangle$ and generate a new rule $L_s \longrightarrow s$. Let us look at an example of such transformation:

$$\begin{array}{r} L \longrightarrow \langle aLL_1 b\rangle \\ L_1 \longrightarrow c \end{array} \Rightarrow \begin{array}{r} L \longrightarrow \langle aL_{LL_1} b\rangle \\ L_{LL_i} \longrightarrow LL_1 \\ L_1 \longrightarrow c \end{array}$$

The next step is validation; It is performed with the help of dependency graph of the grammar with $V, E)$, where

$E = \{(L, L`, (s_1, s_2)) | s_1, s_2 \in (\sum \cup V)*, (L \longrightarrow s_1 L s_2)\}$.
The edges in the graph are transitions from a non-terminal to another non-terminal labeled with symbols on the right and on the left of the destination non-terminal.

The task of the validation function is to verify that for each loop in the dependency graph, either (1) in the loop there is an edge $(L, L`)$ that is labeled with $(s_1\langle a, b\rangle s_2)$, where $\langle a$ is matched with $b\rangle$ in a rule $L \longrightarrow \langle aL`b \longrightarrow)$; or (2) every edge $(L, L`)$ in the loop is labeled with (s, ϵ) for some string $s \in (\sum \cup V)^*$ and at least one such s satisfies $s \nrightarrow\!\!\!\!\rightarrow *\epsilon$.
For the example above the dependency graph will look like this:

This graph has one loop: with edges labelled $(\langle a, b\rangle)$ and (ϵ, L_1). It is easy to see that the grammar would pass validation by the first

$$\begin{tikzcd}
L \arrow[r, bend left, "(\langle a,b\rangle)"] & L_{LL_1} \arrow[l, bend left, "(\epsilon, L_1)"] \arrow[r, "(L,\epsilon)"] & L_1
\end{tikzcd}$$

criteria. The validation algorithm is described in more detail by Jia et al. [9].

4 DEVELOPMENT AND EVALUATION

To implement ideas that were discussed in Methodology, we developed a tool [2] using Java. In this section we will describe functionality and structure of this tool. The language was chosen to be able to easier integrate with ANTLR parser generator. We used *ParserBasedVisitor* to walk the parse tree of a grammar to retrieve sets of *terminals* and *nonterminals*, rules associated with each nonterminal as a map. Then the map is converted to a grammar that has the following class structure:

- Grammar : The main class of this project. It contains methods that are used to tag and transform the grammar. The fields of this class are: *start* symbol of the grammar, *set* of non-terminals, *set* of terminals, *set* of call and *set* of return symbols, *list* of pairs of terminals and a *set* of terminal symbols that can only be plain. The latter is used for keeping track of symbols that occur in a rule without any other terminal symbols, or such symbols that occur with an EBNF suffix in a grammar.
- EbnfSuffix : enumerator for possible suffixes; Question, Plus, Star or None.
- NonTerminal : extends *Node* class, has a set of rules as a set of lists of symbols and its EBNF suffixes.
- Node : a symbol in a grammar rule, has a of the symbol. Has *call* and *return* sets that are initially empty, a set of terminal and nonterminal symbols and a *start* symbol of the grammar. This is the main class with methods that implement tagging algorithms.
- DepGraph : used for validating structure of tagged grammars.
- Edge : edge in a dependency graph. Has source and destination nodes and *s1 s2* labels.

4.1 *Grammar* implementation

The *Grammar* class provides following methods:

- Grammar constructor: this method receives *start* symbol of the grammar, initializes and fills out set of *terminals*, *nonterminals*. Alternatively it is possible to pass those sets to the constructor if they are already known.
- public **makePairs**: this method iterates over the grammar and creates a list of all possible pairs of terminals in that grammar. Those pairs can be used later to tag the grammar by precedence.
  The function starts with a *start* symbol on the stack and in the set of non-terminals that store rules that should still be visited and rules that have already been visited and iterates over non-terminals by popping from the stack until it is empty. When algorithm encounters a non-terminal that is not yet in *visited* set, the node is added to both *visited* and the stack *toVisit*. For each rule, each pair of terminals (a, b) that occurs in the rule, such that a precedes b, is stored as a potential *call-return* pair in a list. This list is then used by the functions described below. For each *terminal* symbols is either an only symbol in at least one rule, or occurs with an EBNF suffix at least once in any rule, such symbol is added to a *set* of *onlyPlain* symbols.
- private **countPairs**: Creates a map $pair \rightarrow number\ of\ occurrences$. This map is used to tag grammars by precedence.
- public **tagByPrecedence**: With pairs counted, tags the grammar by precedence as described in 3 section. This function takes **removeContraditions** flag as an argument. If this flag is set, contradicting pairs with smaller number of occurrences will be removed from the collection of pairs.
  tagByPrecedence first calls **countPairs** and sorts its output by value, so that a most frequently occurring pair is first in the stream. Then it iterates over this stream of pairs and for each pair (a, b) adds its elements to corresponding *call* and *return* sets if the following holds: $a \neq b \wedge !(a \in \mathit{onlyPlain}) \wedge !(b \in \mathit{onlyPlain}) \wedge !(a \in \mathit{call}) \wedge !(a \in \mathit{return} \wedge !(b \in \mathit{call}) \wedge !(b \in \mathit{return}))$. Otherwise, if *removeContradictions* flag is set, the pair is removed from the collection of pairs.
- public **isLeftRecursive**: Checks if grammar is left recursive.
- public **removeLR**: Refactors grammar to remove left recursion. During this new
- public **convertToSimpleForm**: Converts the grammar to its simple form. This is required for evaluation step. If there is no tagging, that is if *call* and *return* sets are empty, the grammar will not be changed.
- public **bruteforceTagging**: tags the grammar by iterating over all combinations of terminals and then checking if grammar is well matched. This method has $O(n^2)$ complexity, where n is a number of distinct terminals, and may be slow for large grammars.

4.2 Implementation of validation algorithm

Although conversion of a grammar to a *simple form* is trivial, as well as process of checking the loops of the graph, finding all cycles in the graph proves to be a complex task. In this section we will describe algorithm used to build a dependency graph and limitations arising when trying to detect cycles.

To build a graph, *DepGraph* class provides method **makeGraph** that, similar to **makePairs** method iterates over *non-terminals*, that are vertices of the graph, using a stack and a set of visited nodes. For each vertex the algorithm stores a set of edges going out of that vertex. The next step is to collect all existing cycles in the graph to analyze them. There exist advanced algorithm for solving this problem, such as Johnson's algorithm [11], with complexity $O((n + e)(c + 1)$, however, this and other algorithms developed for solving the same problem work correctly only on graphs that (1) only have one edge from any vertex V to any other vertex $V‘$ and (2) does not contain loops, i.e. edges from V to V. Both of those structures can be present in a grammar in a form of recursion or by repeating same non-terminal on the right hand side of a rule.

With that in mind, algorithm implemented for finding cycles has complexity of $O(n!)$, which proved to be a limitation when testing the tool on real grammars.

5 RELATED WORK

In their other recent paper Xiadong Jia and Gang Tan discuss V-star algorithm, a tool that learns grammars of visibly pushdown languages based on program input [10]. In addition to that, they describe an algorithm that is capable of inferring tagging of visibly pushdown language based on program sample.

6 RESULTS

In this section we will first discuss the experiment setup and finally show and discuss results of tagging a large set of grammars. When running the experiment we record following statistics: total number of processed grammars, number of grammars with that had left recursion prior to transformation, number of grammars tagged grammars, i.e. with at least one *call-return* pair, number of tagged grammars that had left recursion, number of grammars that passed validation, number of tagged grammars that passed validation. For each grammar in the test set, the methods described in the implementation section are applied in the following order:

1. **removeLR** if grammar is left recursive.
2. **makePairs**.
3. **tagByPrecedence** with *removeContradictions* flag set.
4. **convertToSimpleForm**. After applying this function a grammar with tagging will be written to a file.
5. **DepGraph.findCycles**. Due to limitations discussed in previous chapter, this and the next methods are only invoked for grammars with less than 70 non-terminals.
6. **DepGraph.isValid**.

We ran our experiment on the latest version of grammar-v4 repository for ANTLR [6]. Table 1 6 contains the results of the experiment.

Table 1: Ratios of Grammar Categories

Category	Count	Ration of total grammars processed (%)
Total Processed Grammars	345	100.00%
Grammars with LR	92	26.67%
Tagged grammars	270	78.26%
Tagged grammars with LR	90	26.09%
Valid tagged grammars	98	24.35%
Valid grammars with no tagging	66	18.26%
Valid grammars that had LR	6	0.017%
Skipped validation	105	44.06%

7 DISCUSSION AND FUTURE WORK

In this section we will discuss the results of the experiments, revise the limitations solving which would potentially result in higher ratio of valid grammars, propose solutions for some of those issues and finally propose an additional tagging technique that was not explored in this article.

7.1 Discussion

According to final results, out of 240 grammars for which we were able to run validation algorithm, 164 or 68.3% of grammars turned out to be convertible to its visibly pushdown equivalent. Surprisingly, a relatively large number of grammars, namely 66 or 18.26%, did not need tagging to be convertible. Given that visibly pushdown grammars are closest to regular grammars, with exception, except for augmentation with *call* and *return* separation, this might mean that those grammars can be converted to a regular equivalent. Examining some of those grammars visually we were not able to find contradictions to this assumption. Initially as a measuring point of our progress we chose a the number of grammars that Jia et al were able to convert in their work. Even though we were not able to test for all grammars, the ratio of convertible grammars to the total number of files as a result of running our algorithm is higher: Jia et al were able to convert 136 grammars out of 239 or 56.9%. Note that in their work the total number of grammars is less too: this might be due to the fact that they used only one grammar for each language with multiple versions, when for this experiment we treated grammars for different versions of languages separately.

The results in table 1 6 show that most of grammars, namely 78.26% can be tagged. Since in this article we covered only one transformation technique, namely removal or left recursion, it is theoretically possible to achieve higher number of valid grammars through other transformations or tagging techniques. One of the possible techniques will be introduced later in this chapter 7.3.

7.2 Limitations and future work

One of the limitations that we faced during the experiment phase was that the algorithm for finding all had a high complexity. It would potentially be possible to apply Johnson's algorithm to the graph if some transformations are made to the grammar. First, to eliminate loops containing only one edge, we could introduce an intermediate rule, and therefore an intermediate edge in the graph. Second, for each repetition of the same non-terminal on the right hand side of the rule we could also introduce a unique intermediate rule. Doing this would assure that graph no two edges between any two same nodes. This would make dependency graph suitable to apply Jhonson's algorithm on it. However, we need to keep in mind that such transformations will greatly increase both number of vertices and edges in the graph, as well as the number of different, and Johnson's algorithm has complexity of $O(\mathrm{O}((\mathrm{n}+\mathrm{e})(\mathrm{c}+1))$ where n, e and c are number of nodes, edges and cycles in the graph correspondingly. Therefore, such transformation my have very little, if any, positive effect on time complexity of the algorithm.

When tagging grammars by precedence, in case there is a conflict between possible *call-return* pairs. In case there is same number of conflicting pairs, one pair is chosen arbitrary. Instead of excluding conflicts, it is possible to instead try different tagging sets. This would be valuable since there could potentially be grammars the are convertible only with specific tagging, while our algorithm faultily chooses pairs that are not useful for conversion.

7.3 Stropping

Stropping is a technique that was commonly used to make a letter syntax have a specific property, e.g. being a keyword in a language.

Essentially, stropping involves augmenting a terminal with some characters, for example with dots in FORTRAN, with intention to treat such symbols differently compared to not augmented symbols during parsing. This technique was used in ALGOL [4]. In this section we will show how stropping can be used to push limits of tagging further, potentially allowing for more extensive tagging of grammars. First we will take a look at an example.

$$\begin{aligned} L &\longrightarrow aEbS \\ L_1 &\longrightarrow aSc \\ E &\longrightarrow e \\ S &\longrightarrow s \end{aligned}$$

In the grammar above our algorithm, when collecting pairs of potential call and return symbols would find two candidates: (a, b) and (a, c). Then the algorithm would arbitrary choose one of those combination, since none of them has more occurrences and therefore they have equal priority. However, it is possible to introduce an intermediate nonterminal in place of terminal a, such that when tagging, those nonterminals would be treated differently:

$$\begin{aligned} L &\longrightarrow .aEbS \\ L_1 &\longrightarrow ..aSc \\ E &\longrightarrow e \\ S &\longrightarrow s \\ &lexer \\ .a &\longrightarrow a \\ ..a &\longrightarrow a \end{aligned}$$

Here we augmented terminals a with one or two dots and added two transitions to the lexer of our grammar. Since both $.a$ and $..a$ lead to the same terminal a, the grammar is equivalent to the original grammar. However, $.a$ and $..a$ are different symbols and therefore there will be no conflict when tagging the grammar, and both pairs $(.a, b)$ and $(..a, c)$ can be tagged. It will make possible tagging of languages that have repeating keywords shared between multiple features.

8 CONCLUSIONS

The goal of this research was to attempt converting Context Free Grammars to their Visibly Pushdown equivalent. To do that, an tagging algorithm was implemented [2] and different heuristics were tested. Then, using an approach that produced more precise tagging, we evaluated structure of tagged grammars to test if those grammars can be converted. In order to accomplish that, we implemented an algorithm that utilises dependency graph to check of loops in that graph. The next step to further increase number of convertible grammars was grammar transformation. According to our results removing left recursion can indeed be useful for converting a grammar to its VP equivalent. There are more tagging approaches, as well as grammar transformations that can be implemented to investigate their effect on performance of the algorithm. In addition to those findings, we found out that relatively large number of grammars in the repository that was used for testing can potentially be converted to regular expressions.

REFERENCES

[1] Rajeev Alur and Parthasarathy Madhusudan. “Visibly pushdown languages”. In: *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. 2004, pp. 202–211.

[2] Danila Bren. https://github.com/danilBren/CFG_VPG.git. 2024.

[3] Noam Chomsky. “Three models for the description of language”. In: *IRE Transactions on information theory* 2.3 (1956), pp. 113–124.

[4] William Clinger and Jonathan Rees. “Revised report on the algorithmic language Scheme”. In: *MIT Libraries* (1991).

[5] Ludmila Fedorchenko and Sergey Baranov. “Equivalent Transformations and Regularization in Context-Free Grammars”. In: *Cybernetics and Information Technologies* 14.4 (2015), pp. 29–44. DOI: doi:10.1515/cait-2014-0003. URL: https://doi.org/10.1515/cait-2014-0003.

[6] *Grammars-v4*. https://github.com/antlr/grammars-v4. 2014.

[7] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. “Introduction to automata theory, languages, and computation”. In: *Acm Sigact News* 32.1 (2001), pp. 60–65.

[8] Gerhard Jäger and James Rogers. “Formal language theory: refining the Chomsky hierarchy”. In: *Philosophical Transactions of the Royal Society B: Biological Sciences* 367.1598 (2012), pp. 1956–1970.

[9] Xiaodong Jia, Ashish Kumar, and Gang Tan. “A Derivative-based Parser Generator for Visibly Pushdown Grammars”. In: *ACM Trans. Program. Lang. Syst.* 45.2 (May 2023). ISSN: 0164-0925. DOI: 10.1145/3591472. URL: https://doi.org/10.1145/3591472.

[10] Xiaodong Jia and Gang Tan. “V-Star: Learning Visibly Pushdown Grammars from Program Inputs”. In: *Proceedings of the ACM on Programming Languages* 8.PLDI (2024), pp. 2003–2026.

[11] Donald B Johnson. “Finding all the elementary circuits of a directed graph”. In: *SIAM Journal on Computing* 4.1 (1975), pp. 77–84.

[12] Mark Johnson. “Finite — state approximation of constraint-based grammars using left-corner grammar transforms”. In: *36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, Volume 1*. 1998, pp. 619–623.

[13] I JTC. “Information technology — syntactic metalanguage — extended BNF”. In: *ISO/IEC Internationl Standard, vol. 14977: 1996 (E)* (1996).

[14] John Levine. *Flex & Bison: Text Processing Tools.* " O’Reilly Media, Inc.", 2009.

[15] Robert C Moore. “Removing left recursion from context-free grammars”. In: *1st Meeting of the North American Chapter of the Association for Computational Linguistics*. 2000.

[16] Maurice Nivat. “On some families of languages related to the Dyck language”. In: *Proceedings of the second annual ACM symposium on Theory of computing*. 1970, pp. 221–225.

[17] Terence Parr et al. “Antlr (another tool for language recognition)”. In: *University of San Francisco* (2006).

[18] Terence Parr, Sam Harwell, and Kathleen Fisher. “Adaptive LL (*) parsing: the power of dynamic analysis”. In: *ACM SIGPLAN Notices* 49.10 (2014), pp. 579–598.

[19] Terence J Parr and Russell W Quong. “Adding semantic and syntactic predicates to LL (k): pred-LL (k)”. In: *International Conference on Compiler Construction*. Springer. 1994, pp. 263–277.

[20] Hayo Thielecke. “Functional semantics of parsing actions, and left recursion elimination as continuation passing”. In: *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming*. PPDP ’12. Leuven, Belgium: Association for Computing Machinery, 2012, pp. 91–102. isbn: 9781450315227. doi: 10.1145/2370776.2370789. url: https://doi.org/10.1145/2370776.2370789.