C2VPG: Translating Practical Context-Free Grammars into Visibly Pushdown Grammars by Order-Based Tagging

Xiaodong Jia
Computer Science and Engineering
The Pennsylvania State University
State College, USA
xxj34@psu.edu

Gang Tan
Computer Science and Engineering
The Pennsylvania State University
State College, USA
gtan@psu.edu

***Abstract*—Context-free grammars (CFGs) are widely used to specify the syntax of programming languages. However, their inherent complexity and lack of structural nesting information make them less suitable for certain parsing and analysis tasks. Visibly pushdown grammars (VPGs) address these limitations by introducing explicit call, return, and plain symbols, enabling efficient parsing and analysis of nested structures. Translating practical CFGs into VPGs remains challenging, especially with ambiguous constructs like the dangling-else issue, where the order of call and return symbols must be carefully managed to ensure correct parsing.**

In this paper, we present C2VPG, a tool for automatically translating practical CFGs into VPGs using a novel order-based tagging method. Our approach introduces a sound algorithm that automatically determines an order on return symbols and constructs a tagger that assigns call, return, and plain tags to terminals in a CFG based on this order. This method resolves the tagging challenge posed by the dangling-else problem, where return symbols could be optional in sentences. We evaluate our approach on 396 real-world grammars from the ANTLR repository, achieving a 61% success rate in converting CFGs into VPGs. We discuss the challenges posed by practical grammar design that prevent C2VPG's translations. Our results demonstrate that C2VPG is both practical and efficient, and could assist language designers in creating more robust grammars.

***Index Terms*—context-free grammars; visibly pushdown grammars**

I. Introduction

Parsing is a fundamental component of computer systems, with applications ranging from programming language compilers to network protocol analyzers. Modern parsers must be both efficient and secure, as they often process untrusted or adversarial inputs in high-performance settings. While context-free grammars (CFGs) are widely used to specify the syntax of programming languages, they suffer from inherent limitations, such as ambiguity and lack of explicit nesting structures, which hinder their efficiency and suitability for certain parsing tasks. As a result, the worst case complexity of parsing algorithms based on CFGs is cubic time [1].

Visibly pushdown grammars (VPGs) [2] address these limitations by introducing *nesting structures* into a grammar. In particular, all terminals are tagged as call, return, and plain symbols; call and return symbols are used to enclose nesting structures (think of HTML open and close tags). For example, we can tag the string "`OPEN_TAG TEXT CLOSE_TAG`" as "<`OPEN_TAG TEXT CLOSE_TAG`>", where the angle brackets indicate that "`OPEN_TAG`" is a call symbol and "`CLOSE_TAG`" is a return symbol. This tagging of nesting structures via call/return symbols enables linear-time parsing even in the worst case [3], [4]. VPGs occupy a "sweet spot" between the expressive power of CFGs and the efficiency of regular grammars, making them particularly well-suited for applications requiring both high performance and strong security guarantees [3]–[7].

Since many languages already have their syntax specified by CFGs, one approach of taking advantage of efficient and secure VPG parsing is to convert CFGs to VPGs. However, there are many challenges for translating practical CFGs into VPGs, particularly in determining what call and return symbols are. One such case is the well-known dangling-else issue. For an example that will be discussed in detail in Section III, in the following two sentences,

```
1  if x then if y then z
2  if x then if y then z else w
```

the terminals need to be tagged differently:

```
1  <if x then> <if y then> z
2  <if x then> <if y then z else> w
```

In the first, both occurrences of `then` are tagged as return symbols that are matched with an `if`; however, in the second, the second `then` is not tagged as a return symbol. This exemplifies a challenge in CFG-to-VPG translation: the tagging of `then` as a return symbol depends on whether `else` is there, which is not allowed in VPGs. To resolve this issue, we introduce an order-based tagging approach, where the order of return symbols determines their pairing with call symbols.

In this paper, we present C2VPG, a novel approach for automatically translating practical CFGs into VPGs using order-based tagging. Our method introduces a sound algorithm

for constructing a *tagger* that assigns call, return, and plain tags to terminals in a CFG, ensuring that the resulting grammar is well-matched and can be efficiently parsed, while retaining the semantics. Specifically, the algorithm:

1) Identifies *recursion tuples* in the CFG, representing recursive structures that must be nested within call-return pairs.
2) Constructs a tagger that enforces compatibility conditions, ensuring that all recursion tuples are *well-founded* and that each rule body is well-matched.
3) Uses an order mapping to tag return symbols based on their context in a sentence, significantly extending the translation capability.

We evaluate our approach on 396 real-world grammars from the ANTLR repository, achieving a 61% success rate in converting CFGs into VPGs. Our results demonstrate that C2VPG is both practical and efficient, with most translations completing in under a second. We also analyze the reasons for translation failures and identify potential future extensions to improve the success rate.

The contributions of this paper are as follows:

1) A sound algorithm for constructing efficient taggers that translate CFGs into VPGs using order-based tagging.
2) A comprehensive evaluation of the algorithm on real-world grammars, demonstrating its practicality and efficiency.
3) Insights into the challenges of translating practical CFGs into VPGs and potential directions for future work.

The rest of the paper is organized as follows. Section II provides background on VPGs and practical CFGs. Section III presents a motivating example and introduces the problem of translating CFGs into VPGs. Section IV presents our algorithm for constructing taggers and converting CFGs into VPGs. Section V evaluates the algorithm on real-world grammars, Section VII discusses related work, and Section VIII concludes the paper.

II. Background

A. Visibly Pushdown Grammars

VPGs [2] have been used in program analysis, XML processing, and other fields. Compared with CFGs, VPGs enjoy many good properties. Since languages of VPGs are a subset of deterministic context-free languages, it is always possible to build a deterministic PDA from a VPG. The terminals in a VPG are partitioned into three kinds, and the stack action associated with an input symbol is fully determined by the kind of the symbol: an action of pushing to the stack is always performed for a *call symbol*, an action of popping from the stack is always performed for a *return symbol*, and no stack action is performed for a *plain symbol*. Furthermore, languages of VPGs enjoy appealing theoretical closure properties; e.g., the set of visibly pushdown languages is closed under intersection, concatenation, and complement [2]. VPGs also enable the building of linear-time parsers, and VPG parsers are amenable to formal verification [3], [4]. The expressive power of VPG is between regular grammars and CFGs, and is sufficient for describing the syntax of many practical languages, such as JSON, XML, and HTML, with appropriately defined call/return symbols.[1]

We represent a CFG G as a tuple (V, Σ, P, L_0), where V is the set of nonterminals, Σ is the set of terminals, P is the set of production rules, and $L_0 \in V$ is the start symbol. For a VPG, the alphabet Σ is partitioned into three sets: Σ_{plain}, Σ_{call}, Σ_{ret}, which contain plain, call, and return symbols, respectively. Notation-wise, a terminal in Σ_{call} is tagged with $\langle$ on the left, and a terminal in Σ_{ret} is tagged with $\rangle$ on the right. For example, $\langle a$ is a call symbol in Σ_{call}, and $b\rangle$ is a return symbol in Σ_{ret}.

In this paper, we only consider *well-matched VPGs*, which generate only well-matched strings, where a call symbol is always matched with a return symbol.

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

1) $L \to \epsilon$, where ϵ stands for the empty string;
2) $L \to cL_1$, where $c \in \Sigma_{\text{plain}}$;
3) $L \to \langle aL_1 b\rangle L_2$, where $\langle a \in \Sigma_{\text{call}}$ and $b\rangle \in \Sigma_{\text{ret}}$.

Note that in $L \to cL_1$ terminal c must be a plain symbol, and in $L \to \langle aL_1 b\rangle L_2$ a call symbol must be matched with a return symbol; these requirements ensure that any derived string must be well-matched.

The following is an example of a well-matched VPG, which is refactored from a grammar for XML.

element →

OpenTag content CloseTag Empty | SingleTag Empty

In this example, nonterminals start with a lowercase character, such as "element", and terminals start with an uppercase character, such as "OpenTag". The special nonterminal "Empty" has a single rule that produces the empty string. The grammar shows a typical usage of VPGs to model a *hierarchically nested matching* structure of XML texts: "OpenTag" is matched with "CloseTag", and "content" nested in between can be "element" itself (not shown in the above snippet) and forms an inner hierarchy.

The notion of a derivation in VPGs is the same as the one in CFGs. We write $w \to w'$ to mean a single derivation step according to a grammar, where w and w' are strings of terminals or nonterminals. We write $L \to^* w$ to mean that w can be derived from L via a sequence of derivation steps.

B. Context-Free Grammars in Practice

Besides grammars in the EBNF format, most practical parser generator, such as Bison, Yacc, and ANTLR, support additional features, such as regular operators, semantic actions, and annotations [8]–[11]. In this work, we ignore the semantic

[1]For instance, the XML grammar is a VPG if a whole XML tag is treated as a terminal symbol; this requires a lexer that returns XML tags as tokens.

actions and other annotations, and only consider the pure context-free grammars. In addition, a practical grammar often contains two parts: a lexer grammar that specifies how to convert raw texts (character stream) into a token stream, and, based on it, a parser grammar as a context-free grammar, where each terminal is really a token. In this work, we only consider the parser grammar.

III. A Motivating Example

Most practical formal grammars are written as CFGs, and any CFG can be converted into a VPG through appropriate tagging [2][2]. In some cases, this tagging is straightforward. For example, consider the following grammar:

$$L \to aLb \mid c.$$

Here, terminals a, b, and c can be naturally tagged as call, return, and plain symbols, respectively, converting the grammar into a VPG:

$$L \to ‹aLb›E \mid c, \quad E \to \epsilon.$$

The tagging process is not always apparent. Consider the following CFG:

$$L \to cLc \mid c,$$

where L is a nonterminal and c is a terminal. In this case, c functions as a call, return, and plain symbol simultaneously. Multiple translations to a VPG are possible. One can even rewrite the grammar as a regular grammar[3]:

$$L \to ccL \mid c.$$

Nevertheless, this transformation loses the original nesting structure of the grammar. Alternatively, we can tag each occurrence of c based on its position:

$$L \to ‹cLc› \mid c.$$

Here, ‹c, c›, and c are distinct terminals, differentiated by their tagging. To parse a sentence, such as ccc, a preprocessor called *tagger* assigns the appropriate tags to each terminal, converting ccc into ‹ccc›. This tagged sentence is then parsed according to the VPG $L \to ‹cLc› \mid c$, producing a parse tree, from which tagging information can be removed to reconstruct the original parse tree.

We define *translations* from a CFG to a VPG as follows:

Definition III.1 (Translations from a CFG to a VPG)**.** A **translation** from a CFG $G = (\Sigma, V, P, L_0)$ to a VPG $G' = (\Sigma', V', P', L_0')$ is a tuple (G, G', f), where:

- $\Sigma' \subseteq \Sigma \cup \{‹i \mid i \in \Sigma\} \cup \{i› \mid i \in \Sigma\}$, meaning that each terminal in Σ' is either a terminal from Σ, or a tagged version of a terminal as a **call** or **return** symbol.
- For each string $s \in (\Sigma \cup V)^*$ such that $L_0 \to^* s$, the **tagging function** f maps s to a string $\hat{s} \in (\Sigma' \cup V)^*$, such that s and $\hat{s}$ are of the same length, and for each position $i \in [1..|s|]$:
  1) if $s[i] \in V$, then $\hat{s}[i] = s[i]$;
  2) if $s[i] \in \Sigma$, then $\hat{s}[i] \in \{s[i], ‹s[i], s[i]›\}$.
- Ignoring the tagging, the language of G is equivalent to that of G', i.e.:

$$\{f(s) \mid L_0 \to^* s, s \in \Sigma^*\} = \{\hat{s} \mid L_0' \to^* \hat{s} \text{ and } \hat{s} \in \Sigma'^*\}$$

While always possible, finding translations with *efficient* taggers remains an open problem. However, in practice, such translations are often intuitive, and the resulting tagger tends to operate efficiently.

We introduce a general rule for tagging a CFG based on its production rules, with the objective of deriving an efficient tagger. This approach is later formalized as a sound algorithm in the following section.

In short, the general rule requires finding a tagger that ensures (1) each *dependency loop* in the CFG is *well-founded*, and (2) each CFG rule under tagging is well-matched. [3] demonstrates that such a tagger transforms the CFG into a *tagged CFG* that is always convertible to a VPG.

We formalize the related concepts as follows.

Definition III.2 (Dependency graph)**.** The dependency graph of a grammar $G = (V, \Sigma, P, L_0)$ is (V, E_G), where

$$E_G = \{(L, L', (s_1, s_2)) \mid s_1, s_2 \in (\Sigma \cup V)^*, (L \to s_1 L' s_2) \in P\}.$$

Note that an edge from L to L' is labeled with a pair of strings.

Definition III.3 (Dependency loop)**.** Let $G = (V, \Sigma, P, L_0)$ be a grammar with its associated dependency graph (V, E_G). A *dependency loop* is defined as a loop present in the dependency graph such that at least one edge in the loop is labeled with (s_1, s_2), where s_2 can derive a *nonempty* string (i.e., $s_2 \to^* w$ for some nonempty string w). In other words, there exists a set of rules

$$L_i \to \alpha_i L_{i+1} \beta_i, \quad i \in [1..n]$$

where $L_1, L_2, \ldots, L_{n+1}$ are nonterminals, $L_1 = L_{n+1}$, and α_i, β_i are strings of terminals and/or nonterminals (possibly empty). This sequence of rules rewrites L_1 into

$$\alpha_1 \alpha_2 \ldots \alpha_n L_1 \beta_n \beta_{n-1} \ldots \beta_1,$$

which has a recursion involving L_1.

Definition III.4 (Well-Founded Dependency Loop)**.** A dependency loop is defined as *well-founded* under a tagger f, if it contains an edge $(L, L', (s_1, s_2))$, where the tagged sequence $f(s_1 s_2)$ is well-matched, and there exists a call symbol in $f(s_1)$ that is paired with a return symbol in $f(s_2)$.

Now we demonstrate the general rule by a representative example of resolving the well-known "dangling-else" problem in CFGs. The dangling-else problem introduces ambiguity into a CFG. Consider the following example, which specifies the syntax of if-expressions:

[2]Note that the tagging maps a CFG to a VPG with a *new* set of terminals. CFGs are still more expressive than VPGs when they share the same terminals.

[3]More strictly, the regular grammar is $L \to cL_1 \mid c, \quad L_1 \to cL$.

```
if_expr : IF if_expr THEN if_expr
        | IF if_expr THEN if_expr ELSE
            if_expr
        | variable ;
variable: x | y | z | w ;
```

In this grammar, `IF`, `THEN`, `ELSE`, `x`, `y`, `z`, and `w` are terminals, while `if_expr` and `variable` are nonterminals.

Notice that there are three dependency loops in the CFG: from the head nonterminal `if_expr` to the first `if_expr` in the first rule, and to the first and second `if_expr` in the second rule. Below, the related nonterminals are surrounded by curly brackets:

```
{if_expr}: IF {if_expr} THEN  if_expr
         | IF {if_expr} THEN {if_expr}
             ELSE if_expr
```

Based on the general rule, we now try to tag certain terminals to make all dependency loops well-founded. Naturally, `IF` is tagged as a call symbol. The challenge is deciding whether `THEN` or `ELSE` should be tagged as the return symbol: if we tagged `THEN` as the return symbol, `if_expr` in the second rule would not be properly nested within call-return pairs; if we tagged `ELSE` as the return symbol instead, `if_expr` in the first rule would not be nested within call-return pairs.

Instead, we need to tag the terminals in a rule specific way. Below, we tag `THEN` as a return symbol in the first rule, and `ELSE` as a return symbol in the second rule, so that all loops are nested in call-return pairs.

```
if_expr : <IF if_expr THEN> if_expr
        | <IF if_expr THEN  if_expr
           ELSE> if_expr
        | variable ;
variable: x | y | z | w ;
```

Such tagging introduces a gap between a parser that takes tagged strings and an input that is untagged, since a terminal needs to be tagged based on their contexts in sentences before being sent to the parser. Consider the following two sentences:

```
1. IF x THEN IF y THEN z
2. IF x THEN IF y THEN z ELSE w
```

The tagger should tag the terminals as follows:

```
1. <IF x THEN> <IF y then> z
2. <IF x THEN> <IF y THEN z else> w
```

For practical usage, it is important to make sure the tagger is efficient. In the above example, our produced tagger operates in linear time relative to the number of tokens, making it highly efficient. The tagger works as follows: Given an input string, the tagger maintains an initially empty stack. As it reads each token sequentially, it pushes `IF` and `THEN` onto the stack. Upon encountering `ELSE`, the tagger removes the top of the stack (which must be `THEN`) and pairs the `ELSE` with the `IF` at the top of the stack, and removes the `IF`. After processing all tokens, any remaining `IF` tokens on the stack are paired with the subsequent `THEN` tokens on the stack.

Algorithm 1: `higherPri`$(b_1\rangle, b_2\rangle, M, \langle a)$: Returns `true` if $b_1\rangle$ has higher priority than $b_2\rangle$.

Input: Return symbols $b_1\rangle$, $b_2\rangle$, order mapping M, and call symbol $\langle a$.

1 Let i_1 be the index of $b_1\rangle$ in $M[\langle a]$;
2 Let i_2 be the index of $b_2\rangle$ in $M[\langle a]$;
3 **return** *`true` if $i_1 > i_2$, otherwise `false`*;

In general, we found that a significant portion of practical grammars can be successfully converted into VPGs with efficient taggers. In the next section, we propose a sound algorithm for automatically finding efficient taggers in a CFG.

IV. Approach

A. A Sound Algorithm for Finding Efficient Taggers

In this section, we propose an algorithm that, given a context-free grammar (CFG), automatically constructs a tagger that accepts a string and tags each terminal as a call, return, or plain symbol; when a symbol is tagged as a plain symbol, we also say that it is untagged since the tagging of call and return symbols is sufficient to determine that the rest are plain symbols. The algorithm is sound, meaning that if it returns a tagger, the tagger runs in linear time with respect to the length of the input string. However, the algorithm is not complete and may not return a tagger because we impose additional restrictions on the returned tagger for efficiency reasons. Future work could make our algorithm more complete.

B. Tagger Definition

Definition IV.1 (Tagger)**.** At a high level, a tagger in this section is represented as a tuple $(\Sigma_{\text{call}}, \Sigma_{\text{ret}}, M)$, where:

- Σ_{call} is a set of terminals tagged as *call symbols*,
- Σ_{ret} is a set of terminals that *can potentially be* tagged as *return symbols*, and
- M is a mapping from a call symbol $\langle a$ to an ordered list of return symbols in Σ_{ret} that can be potentially paired with $\langle a$:

$$M : \Sigma_{\text{call}} \to \Sigma_{\text{ret}}^{+}$$

Note that $M(\langle a)$ returns a list of return symbols of increasing priority.

Using a tagger, we can describe how to tag a string, as shown in Algorithm 2. The function attempts to match call symbols in the stack with return symbols in the sequence, ensuring the pairs follow the given order: if a return symbol is followed by another return symbol of higher priority, the first one is removed from the stack. If a return symbol appears without a corresponding call symbol in the stack, the function returns false. The algorithm also ensures that any remaining symbols in the stack are well-matched with return symbols. This procedure runs in linear time since each symbol is processed at most twice (once when added to the stack and once when removed).

Algorithm 2: `CheckAndTag`$(\Sigma_{\text{call}}, \Sigma_{\text{ret}}, M, s)$: Checks well-matching of a string and tags the string.

Input: Call symbols Σ_{call}, return symbols Σ_{ret}, order mapping M, and string s.
Output: A tagged version of s, if s is well-matched; otherwise None.

Let stack $T \leftarrow \bot$;
`// Store the call symbols in` $T_{\texttt{call}}$ `for quick retrieval.`
Let call symbol stack $T_{\text{call}} \leftarrow \bot$;
Initialize location mapping N as $N[i] = \texttt{call}$ for each $s[i]$ if $s[i] \in \Sigma_{\text{call}}$.
foreach *i-th symbol x in sequence s* **do**
  if $x \in \Sigma_{call}$ **then** push (i, x) onto T and T_{call};
  else if $x \in \Sigma_{ret}$ **then**
    while $T \neq \bot$ *and top of* T *is in* Σ_{ret} **do**
      Let $(j, \texttt{prev\_return}) \leftarrow$ top of T;
      if $T_{call} \neq \bot$ **then**
        Let $(\_, \texttt{top\_call}) \leftarrow$ top of T_{call};
        if $\mathit{higherPri}(x, \mathit{prev\_return}, M, \mathit{top\_call})$ **then**
          `// Remove smaller return symbols on stack.`
          Remove top of T;
        end
        else
          `// Try to pair prev_return with the stack top.`
          if $T \neq \bot$ **then**
            Remove tops of T and T_{call};
            if $\mathit{prev\_return} \in M[\mathit{top\_call}]$ **then** $N[j] \leftarrow \texttt{return}$;
            else return *None*;
            `// The new top maybe a smaller return symbol, so continue the while-loop.`
          end
          else return *None*;
        end
      end
      else return *None*;
    end
    Push (i, x) onto T;
  end
end
$N' \leftarrow \texttt{CheckStack}(\Sigma_{\text{call}}, \Sigma_{\text{ret}}, M, N, T)$;
if N' *is not None* **then return** s *tagged based on location mapping* N';
else return *None*;

C. Recursion Tuples and Compatibility of Taggers

To understand how to find a tagger compatible with a CFG, we first define the concept of *recursion tuples*, representing the recursive structures within a CFG.

Definition IV.2 (Recursion Tuples)**.** A *recursion tuple* for a given CFG is a tuple (L, s_1, s_2), such that there exists a dependency loop that rewrites L into $s_1 L s_2$.

Definition IV.3 (Compatible Taggers)**.** We call a tagger $(\Sigma_{\text{call}}, \Sigma_{\text{ret}}, M)$ *compatible* with a CFG, if the following conditions hold:

1) For each recursion tuple (L, s_1, s_2) in the CFG, $s_1 s_2$ is well-matched, i.e., $\texttt{CheckAndTag}(\Sigma_{\text{call}}, \Sigma_{\text{ret}}, M, s_1 s_2)$ returns not None, but the sequence s_2 alone is not well-matched, i.e., $\texttt{CheckAndTag}(\Sigma_{\text{call}}, \Sigma_{\text{ret}}, M, s_2)$ returns None. The second condition ensures that s_1 contains at least one call symbol that is paired with a return symbol in s_2, so that recursion is nested within a call-return pair.
2) For each rule $L \to s$, rule body s is well-matched: $\texttt{CheckAndTag}(\Sigma_{\text{call}}, \Sigma_{\text{ret}}, M, s)$.

We can use a compatible tagger $(\Sigma_{\text{call}}, \Sigma_{\text{ret}}, M)$ to tag the CFG by mapping each rule $L \to s$ of the CFG to a new rule $L \to \texttt{CheckAndTag}(\Sigma_{\text{call}}, \Sigma_{\text{ret}}, M, s)$. The resulting grammar is called a *tagged CFG*, a notion introduced in [4], which also explains how to convert a tagged CFG into a VPG. Therefore, in this paper, we only focus on the algorithm for finding a compatible tagger.

Algorithm 3: `CheckStack`$(\Sigma_{\text{call}}, \Sigma_{\text{ret}}, M, N, T)$: Checks well-matching of a stack.

Input: Call symbols Σ_{call}, return symbols Σ_{ret}, order mapping M, location mapping N, stack T.
Output: N updated with locations in T, if T is well-matched, otherwise None.

```
1  T' ← ⊥;
2  while T ≠ ⊥ do
3  |  Let (i, x) ← top of T;
4  |  if x ∈ Σ_ret then push (i, x) to T';
5  |  else if T' ≠ ⊥ then
6  |  |  Let (j, y) ← top of T';
7  |  |  if y ∈ M[x] then
8  |  |  |  N[j] ← return;
9  |  |  |  Remove top of T';
10 |  |  end
11 |  |  else return None;
12 |  end
13 |  else return None;
14 end
15 if T' ≠ ⊥ then return N;
16 else return None;
```

Algorithm 4: `FindCompatibleTagger`(V, R, Q)

Input: Nonterminal set V, recursion tuples R, and rules bodies Q.
Output: Sets of call and return symbols, and order mapping M, if a compatible tagger exists; otherwise `None`.

```
1 Let Σ_call ← {}, Σ_ret ← {};
2 Let M ← a map from each call symbol to an empty list of return symbols;
3 if Backtrack(V, R, Q, Σ_call, Σ_ret, M, 0) then return (Σ_call, Σ_ret, M);
4 else return None;
```

D. Algorithm for Finding a Compatible Tagger

We present an algorithm that searches for a compatible tagger for a given CFG. The main goal is to identify sets of call and return symbols and establish an ordering among the return symbols, ensuring that for each recursion tuple (L, s_1, s_2) of the CFG, sequence $s_1 s_2$ is well-matched under tagging. The algorithm proceeds recursively through the recursion tuples, checking for well-matched sequences and applying corrective actions if mismatches are detected.

The key steps of the algorithm are as follows. For each recursion tuple, if the sequence of symbols is not well-matched, the following three extensions of the tagger are attempted to make the sequence well-matched. If none of the approaches succeeds, the algorithm returns None, indicating that no tagger can be found.

1) **Extend Call Symbols**: The algorithm attempts to extend the set of call symbols by identifying new candidates from the left side of the recursion tuple.
2) **Pair Calls with Returns**: The algorithm attempts to add new call symbols and pair them with existing return symbols.
3) **Extend Return Order**: The algorithm attempts to add a new return symbol to enlarge the order list of an existing call symbol.

The main algorithm (Algorithm 4) employs a backtracking mechanism to explore potential call and return symbols, ensuring that all recursion tuples in the CFG meet the well-matching conditions. This backtracking process is encapsulated in the `BacktrackDFS` function (Algorithm 5), which systematically tries different combinations of call and return symbols. To streamline the exploration, the backtracking process is supported by specialized subroutines—`AddCallSymbol`, `PairCallsWithReturns`, and `ExtendOrder`—outlined in Algorithms 6, 7, and 8, respectively.

Theorem IV.1 (Correctness of Compatible Tagger Algorithm)**.** Given a CFG $G = (\Sigma, V, P, L_0)$, let Q be the set of all rule bodies in P. If `FindCompatibleTagger`(V, R, Q) returns a tagger $(\Sigma_{\text{call}}, \Sigma_{\text{ret}}, M)$, then the tagged CFG $G' = \{L \to \texttt{CheckAndTag}(\Sigma_{\text{call}}, \Sigma_{\text{ret}}, M, s) \mid (L \to s) \in P\}$ can be converted to a visibly pushdown grammar (VPG), and the language of G', when ignoring the tagging, is equivalent to the language of the original CFG G.

Proof. As guaranteed by Line 4 of Algorithm 5, all dependency loops in the tagged CFG G' are well-founded, and each rule body of G' is well-matched. As discussed in [4], such a tagged CFG can be converted into a VPG.

Regarding the equivalence of the two languages, notice that `CheckAndTag`$(-, -, -, s)$ simply provides additional

Algorithm 5: `Backtrack`$(V, R, Q, \Sigma_{\text{call}}, \Sigma_{\text{ret}}, M, i)$

Input: Nonterminal set V, recursion tuples R, rule bodies Q, call symbols Σ_{call}, return symbols Σ_{ret}, mapping M, and index of current recursion tuple i to explore.
Output: true, if a compatible tagger exists; otherwise false.

if $i == |R|$ **then return** *true* ;
Let $(L, s_1, s_2) \leftarrow R[i]$ `// Extract the current recursion tuple.`
Let sequence $s \leftarrow s_1 s_2$;
if `CheckAndTag`$(\Sigma_{call}, \Sigma_{ret}, M, s)$ *and not* `CheckAndTag`$(\Sigma_{call}, \Sigma_{ret}, M, s_2)$ *and* `CheckAndTag`$(\Sigma_{call}, \Sigma_{ret}, M, q)$ *for all* q *in* Q **then**
  return `Backtrack`$(V, R, \Sigma_{call}, \Sigma_{ret}, M, i+1)$
end
`// Step 1: Try adding a new call symbol from` s_1.
if `AddCallSymbol`$(V, s_1, \Sigma_{call}, \Sigma_{ret}, M)$ **then return** *true* ;
`// Step 2: Try pairing calls with returns in` s_1 and s_2.
if `PairCallsWithReturns`$(V, s_1, s_2, \Sigma_{call}, \Sigma_{ret}, M)$ **then return** *true* ;
`// Step 3: Try extending the order of return symbols.`
if `ExtendOrder`$(V, s_1, s_2, \Sigma_{call}, \Sigma_{ret}, M)$ **then return** *true* ;
return *false*

Algorithm 6: `AddCallSymbol`$(V, s_1, \Sigma_{\text{call}}, \Sigma_{\text{ret}}, M)$

Input: Nonterminal set V, left alternative s_1, call symbols Σ_{call}, return symbols Σ_{ret}, and mapping M.
Output: true, if a new call symbol can be added and matched successfully; otherwise false.

foreach `call_candidate` $\in s_1$ **do**
  if `call_candidate` $\notin V$ *and* `call_candidate` $\notin \Sigma_{call}$ **then**
    Add `call_candidate` to Σ_{call};
    foreach `return_candidate` $\in \Sigma_{ret}$ **do**
      M[`call_candidate`] $\leftarrow$ [`return_candidate`];
      if `Backtrack`$(V, R, \Sigma_{call}, \Sigma_{ret}, M, 0)$ **then**
        return *true*
      end
    end
    Remove `call_candidate` from Σ_{call};
  end
end
return *false*;

information to each symbol in s. Therefore, when the tagging is ignored, the language of G' is identical to that of the original CFG G. □

In conclusion, Algorithm 4 is sound, providing a method for finding a compatible tagger for a given CFG. The tagger produced by the algorithm operates efficiently, ensuring linear-time performance with respect to the length of the input sequence. This makes the approach practical for large grammars and real-world applications.

V. Evaluation

We implemented the tagging-inference algorithm in Python, and the source code is publicly available [12]. To evaluate the effectiveness of our approach, we applied the algorithm to all 396 grammars from the ANTLR repository [13]. The results are summarized in Table I. The algorithm successfully generated efficient taggers for 241 grammars, representing 61% of the total. These grammars were automatically converted into visibly pushdown grammars (VPGs). A time limit of 2 hours was imposed on the analysis, and 5 grammars exceeded this limit.

Table II presents the performance of the tagging-inference algorithm on the ANTLR grammars. The runtimes reflect the total time from reading an ANTLR grammar to constructing a tagger. The results indicate that for grammars that successfully convert, the algorithm performs efficiently, with most translations completing in under a second. However, for grammars that fail to convert, particularly larger or more complex ones, the backtracking nature of the algorithm can result in significantly longer runtimes.

Figure 1 shows the success rates of the tagging-inference algorithm across different grammar file sizes. The success rate generally decreases as file size increases. For smaller grammar files (<3KB), the success rate remains high; however, with

Algorithm 7: `PairCallsWithReturns`$(V, s_1, s_2, \Sigma_{\text{call}}, \Sigma_{\text{ret}}, M)$

Input: Nonterminal set V, left alternative s_1, right alternative s_2, call symbols Σ_{call}, return symbols Σ_{ret}, and mapping M.

Output: true, if new call-return pairs can be matched successfully; otherwise false.

```
foreach call_candidate ∈ s1 do
    if call_candidate ∉ V and call_candidate ∉ Σ_call then
        foreach return_candidate ∈ s2 do
            if return_candidate ∉ V and return_candidate ∉ Σ_ret then
                Add call_candidate to Σ_call;
                Add return_candidate to Σ_ret;
                M[call_candidate] ← [return_candidate];
                if Backtrack(V, R, Σ_call, Σ_ret, M, 0) then return true ;
                Remove return_candidate from M[call_candidate];
                Remove call_candidate from Σ_call;
                Remove return_candidate from Σ_ret;
            end
        end
    end
end
return false;
```

Algorithm 8: `ExtendOrder`$(V, s_1, s_2, \Sigma_{\text{call}}, \Sigma_{\text{ret}}, M)$

Input: Nonterminal set V, left alternative s_1, right alternative s_2, call symbols Σ_{call}, return symbols Σ_{ret}, and mapping M.

Output: true, if the order of return symbols can be extended successfully; otherwise false.

```
foreach call_candidate ∈ s1 do
    if call_candidate ∈ Σ_call then
        Let return_symbol ← last(M[call_candidate]);
        foreach return_candidate ∈ s2 do
            if return_candidate ∉ V and return_candidate ∉ Σ_ret ∪ Σ_call then
                Add return_candidate to Σ_ret;
                Append return_candidate to M[call_candidate];
                if Backtrack(V, R, Σ_call, Σ_ret, M, 0) then return true ;
                Remove return_candidate from Σ_ret;
                Remove return_candidate from M[call_candidate];
            end
        end
    end
end
return false;
```

larger files, the algorithm faces increasing difficulty in finding a suitable tagger, due to the complex grammar structures.

VI. Discussion

We discuss some representative reasons why the tagging-inference algorithm fails to identify a suitable tagger.

1) Return symbols containing multiple tokens. The algorithm is designed to tag a single ANTLR token as a call or return symbol. However, return symbols can be composed of separate tokens. For example, in the `Ada2005` grammar, `IF` is paired with `END IF`, where `IF` and `END` are distinct tokens.

2) Call symbols embedded in nonterminals. Some grammars may group call symbols within a nonterminal. For example, consider the following rule in `Modelica`:

```
class_definition
: ('encapsulated')? class_prefixes
    ... 'end' ;
```

Here, the nonterminal `class_prefixes` specifies a group of call symbols that can be paired with the return symbol `'end'`, such as `'class'`, `'model'`, and `'record'`—all of which intuitively signal the beginning of a class definition.

TABLE I
THE STATISTICS OF ALL 396 ANALYZED ANTLR GRAMMARS.

Type	Number	Ratio
Succeed	241	61%
Failed	150	38%
Out of Time (2 hours)	5	1%

TABLE II
THE RUNNING TIME OF THE TAGGING-INFERENCE FUNCTION ON ANALYZABLE ANTLR GRAMMARS.

Type	Number	Min (s)	Max (s)	Average (s)	Median (s)
Succeed	241	0.77	0.00	162.33	0.01
Failed	150	73.54	0.01	4165.08	0.10

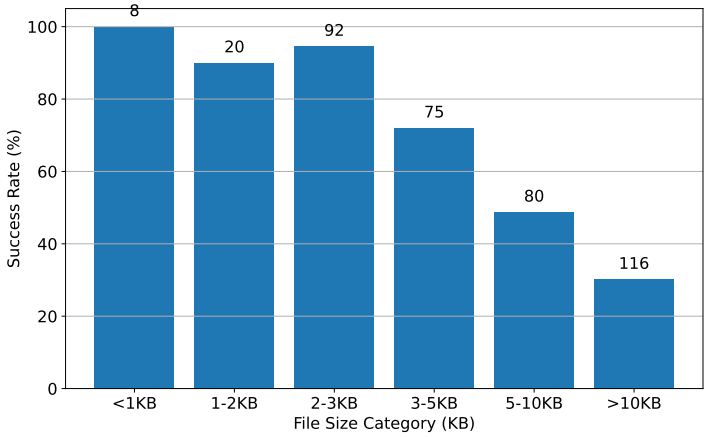

Fig. 1. Success rate of the tagging-inference algorithm by grammar file size, with the number of grammars in each category displayed above the bars.

3) Hard-to-detect regular subgrammars. Given a CFG $G = (V, \Sigma, P, L_0)$ and a nonterminal $L \in V$, we define the *subgrammar* of L as all rules in P that are reachable from L, with L as the start nonterminal. Some subgrammars can be converted to regular grammars, but detecting them is challenging. For example, the `asmMASM` grammar contains a subgrammar for the nonterminal `expression`, which includes the following rules:

```
1 expression
2 : number 'DUP' expression ('+'
      number 'DUP' expression)*
3 | number ;
```

There is no suitable tagging to make the first rule well-founded. However, the subgrammar has an equivalent regular expression:

```
1 (number 'DUP')* number ('+' (number
      'DUP')* number)*
```

Unfortunately, detecting whether a grammar can be converted to a regular grammar is an undecidable problem. Moreover, converting a CFG to a regular grammar, even when possible, can often disrupt the original structure of the CFG and undermine the intentions of the language designers.

While C2VPG achieves a high success rate for smaller-sized grammars, it could be extended to improve success rates by broadening the forms of call and return symbols, such as including token sequences, or nonterminals that can be defined using regular expressions over tokens. Such extensions would come at the cost of increased running time. We leave these enhancements for future work.

VII. RELATED WORK

Jia et al. [14] proposed V-Star, a tool that inferred program input grammars as VPGs from black-box programs. Unlike V-Star, which requires inferring a tokenizer and works on program inputs, C2VPG operates directly on existing CFGs and focuses on parser rules in ANTLR grammars. Under similar problem setting as V-Star, Michaliszyn et al. [15] studied the problem of active learning deterministic visibly pushdown automata.

Bren [16] also tried to convert CFGs to VPGs. However, their approach is based on an enumeration of all possible taggings. In contrast, C2VPG uses a more efficient algorithm that constructs a tagger based on recursion tuples, and extends the ability of the tagger with ordering.

VIII. CONCLUSION

In this paper, we present C2VPG, a novel approach for automatically translating practical CFGs into VPGs using order-based tagging. Our method introduces a sound algorithm for constructing a tagger that assigns call, return, and plain tags to terminals in a CFG, ensuring that the resulting grammar is well-matched and can be efficiently parsed. Still, different language features bring challenges to the tagging-inference algorithm, and we discuss some representative reasons for translation failures. We evaluate our approach on 396 real-world grammars from the ANTLR repository, achieving a

61% success rate in converting CFGs into VPGs. Our results demonstrate that C2VPG is both practical and efficient, paving the way for broader adoption of VPGs in parsing and analysis tasks.

REFERENCES

[1] A. Aho and M. S. Lam, *Compilers principles techniques & tools*. Pearson, 2022.

[2] R. Alur and P. Madhusudan, "Adding nesting structure to words," *J. ACM*, vol. 56, no. 3, may 2009. [Online]. Available: https://doi.org/10.1145/1516512.1516518

[3] X. Jia, A. Kumar, and G. Tan, "A derivative-based parser generator for visibly pushdown grammars," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485528

[4] ——, "A derivative-based parser generator for visibly pushdown grammars," *ACM Trans. Program. Lang. Syst.*, vol. 45, no. 2, may 2023. [Online]. Available: https://doi.org/10.1145/3591472

[5] H. Lampesberger, "An incremental learner for language-based anomaly detection in xml," in *2016 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2016, pp. 156–170.

[6] S. Cowger, Y. Lee, N. Schimanski, M. Tullsen, W. Woods, R. Jones, E. Davis, W. Harris, T. Brunson, C. Harmon *et al.*, "Icarus: Understanding de facto formats by way of feathers and wax," in *2020 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2020, pp. 327–334.

[7] R. Chrétien, V. Cortier, and S. Delaune, "From security protocols to pushdown automata," *ACM Transactions on Computational Logic (TOCL)*, vol. 17, no. 1, pp. 1–45, 2015.

[8] F. S. Foundation, "Gnu bison," https://www.gnu.org/software/bison/, 2021.

[9] J. R. Levine, T. Mason, and D. Brown, *Lex and Yacc, 2nd Edition*. O'Reilly and Associates, 1992.

[10] T. Parr, "Antlr," https://www.antlr.org/.

[11] J. Zhang, G. Morrisett, and G. Tan, "Interval parsing grammars for file format parsing," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Jun. 2023. [Online]. Available: https://doi.org/10.1145/3591264

[12] "C2vpg," https://github.com/xdjia/C2VPG, 2025.

[13] "Grammars-v4," https://github.com/antlr/grammars-v4, 2014.

[14] X. Jia and G. Tan, "V-star: Learning visibly pushdown grammars from program inputs," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, Jun. 2024. [Online]. Available: https://doi.org/10.1145/3656458

[15] J. Michaliszyn and J. Otop, "Learning Deterministic Visibly Pushdown Automata Under Accessible Stack," in *47th International Symposium on Mathematical Foundations of Computer Science (MFCS 2022)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Szeider, R. Ganian, and A. Silva, Eds., vol. 241. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 74:1–74:16. [Online]. Available: https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.MFCS.2022.74

[16] D. Bren, "Pushing down context-free grammars," B.S. thesis, University of Twente, 2024.