A Derivative-based Parser Generator for Visibly Pushdown Grammars

XIAODONG JIA, ASHISH KUMAR, and GANG TAN, The Pennsylvania State University, USA

In this article, we present a derivative-based, functional recognizer and parser generator for visibly pushdown grammars. The generated parser accepts ambiguous grammars and produces a parse forest containing all valid parse trees for an input string in linear time. Each parse tree in the forest can then be extracted also in linear time. Besides the parser generator, to allow more flexible forms of the visibly pushdown grammars, we also present a translator that converts a tagged CFG to a visibly pushdown grammar in a sound way, and the parse trees of the tagged CFG are further produced by running the semantic actions embedded in the parse trees of the translated visibly pushdown grammar. The performance of the parser is compared with popular parsing tools, including ANTLR, GNU Bison, and other popular hand-crafted parsers. The correctness and the time complexity of the core parsing algorithm are formally verified in the proof assistant Coq.

CCS Concepts: • **Software and its engineering** → **Parsers**; *Software verification*; • **Theory of computation** → **Grammars and context-free languages;**

Additional Key Words and Phrases: Parser generators, formal verification, derivative-based parsing

ACM Reference format:
Xiaodong Jia, Ashish Kumar, and Gang Tan. 2023. A Derivative-based Parser Generator for Visibly Pushdown Grammars. *ACM Trans. Program. Lang. Syst.* 45, 2, Article 9 (May 2023), 68 pages.
https://doi.org/10.1145/3591472

1 INTRODUCTION

Parsing is a fundamental component in computer systems. Modern parsers used in high-performance settings such as web browsers and network routers need to be efficient, as their performance is critical to the performance of the whole system. Furthermore, high-assurance parsers are becoming increasingly more important for security in settings such as web applications, where their parsers are directly processing potentially adversarial inputs from the network. In these settings, formally verified parsers are highly desirable.

Most parsing libraries are based on **Context-Free Grammars (CFGs)** or their variants. Although very flexible, CFGs have limitations in terms of efficiency. Not all CFGs can be converted to deterministic **pushdown automata (PDA)**; the inherent nondeterminism in some CFGs causes the worst-case running time of general CFG-based parsing algorithms to be $O(n^3)$.

To achieve efficient parsing, many parsing frameworks place restrictions on what CFGs can be accepted, at the expense of placing the burden on users to refactor their grammars to satisfy those

This work was supported by DARPA research grant HR0011-19-C-0073.
Authors' address: X. Jia, A. Kumar, and G. Tan, The Pennsylvania State University, 201 Old Main, State College, Pennsylvania, PA, 16802; emails: xxj34@psu.edu, ashishky36@gmail.com, gtan@psu.edu.

0164-0925/2023/05-ART9 $15.00
https://doi.org/10.1145/3591472

restrictions. Please see the related-work section for discussion about common kinds of restrictions, leading to parsing frameworks such as LL(k), LR(k) [11], and so on.

This article explores an alternative angle of building parsers based on **Visibly Pushdown Grammars (VPGs)** [2]. In VPGs, users explicitly partition all terminals into three kinds: plain, call, and return symbols. This partitioning makes the conversion of a VPG to a deterministic PDA always possible, which provides the foundation for efficient algorithms. Compared to requiring users to refactor their grammars to *resolve sophisticated conflicts during parser generation such as shift-reduce conflicts in LR(k)*, asking users to specify what terminals are call and return symbols is less of a burden.

VPGs have been used in program analysis, XML processing, and other applications [1], but their potential in parsing has not been fully exploited. In this article, we show that VPGs bring many benefits in parsing. First, we show an efficient, linear-time parsing algorithm for VPGs. Second, our algorithm is amenable to formal verification. Overall, this article makes the following contributions:

- We present a derivative-based algorithm for VPG recognition and parsing. The generated parser accepts ambiguous grammars and produces a parse forest for the input string, where each parse tree in the forest can be extracted in linear time.
- We mechanize the correctness proofs of the parsing algorithm in Coq. We have also formalized and proved the linear-time performance guarantee of the parsing algorithm in Coq.
- We present a surface grammar called tagged CFGs to allow a more convenient use of our parsing framework. Users can use their familiar CFGs for developing grammars and provide additional tagging information on terminals. A sound translator then converts a tagged CFG to a VPG. We note that this validator is conservative and places some restriction on acceptable tagged CFGs; in particular, left recursion is not allowed and users are required to perform standard refactoring to remove left recursion.
- During the performance evaluation of VPG-based parsers on popular formats including JSON, XML, and HTML, we discover that their performance is comparable with popular parsing generators such as ANTLR and other hand-crafted parsers. In addition, we discover that VPG-based parsers often require a special lexer that groups multiple tokens into one single call/return symbol.

The remainder of this article is organized as follows: We first introduce VPGs in Section 2 and discuss related work in Section 3. In Section 4, we provide an overview of our parsing library and explore various usage scenarios. Section 5 presents a derivative-based VPG recognizer, which sheds light on the parsing algorithm discussed in Section 6. The translator and tagged CFGs are discussed in Section 7. We then evaluate the VPG parser in Section 8.

A conference version of this article was published in the **ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)** [22]. We describe the differences between the conference version and this article as follows: First, we have redesigned the VPG parsing algorithm, including changing how the parser PDA is generated and replacing the pruner in the conference version with a set of parse-tree extraction functions. The new algorithm enables a simpler correctness proof and, more importantly, enables us to verify the performance guarantee: The parser is verified to have time complexity of $O(kn)$, where k is the number of parse trees and n is the length of the input string. Second, we provide a new method that optimizes the parse-tree extraction functions to a more efficient extraction PDA. Third, we rerun all experiments to evaluate our new parsing algorithm. Finally, we have revised our translation algorithm from tagged CFGs to VPGs and added the proofs of its soundness and termination.

2 BACKGROUND: VPG

As a class of 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]. As will be shown in this article, VPGs also enable the building of linear-time parsers, and VPG parsers are amenable to formal verification. 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 appropriated defined call/return symbols.[1]

We next give a formal account of VPGs. A grammar G is represented 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. 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}.

We first formally define *well-matched VPGs*. Intuitively, a well-matched VPG generates only well-matched strings in which a call symbol is always matched with a return symbol in a derived string.

Definition 2.1 (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:

$$\text{element} \to \text{OpenTag content CloseTag Empty} \mid \text{SingleTag Empty}.$$

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.

In the main text of this article, we consider only well-matched VPGs and use the term VPGs for well-matched VPGs. We also call rules in the form of $L \to \langle aL_1 b\rangle L_2$ *matching rules*. Our Coq library actually supports a broader class of VPGs, which we call *general VPGs*; algorithms for general

[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.

VPGs are put into the appendices. Intuitively, general VPGs can specify *substrings* of well-matched strings; e.g., $\langle/p\rangle\langle p\rangle\langle/p\rangle\langle p\rangle$ is a substring of a well-matched string. This ability enables users to parse streaming data such as network traffic. As an extension to well-matched VPGs, general VPGs allow the case of *pending calls and returns*, which means that a call/return symbol may not have its corresponding return/call symbol in the input string. To accommodate pending symbols, general VPGs in addition allow rules in the forms of $L \to \langle aL'$ and $L \to b\rangle L'$, which we call *pending rules*. Call/return symbols in pending rules are called *pending call/return symbols*. Further, the set of nonterminals V is partitioned into V^0 and V^1: nonterminals in V^0 only generate well-matched strings, while nonterminals in V^1 can generate strings with pending symbols.

Definition 2.2 (General VPGs). A grammar $G = (V, \Sigma, P, L_0)$ is a general VPG with respect to the partitioning $\Sigma = \Sigma_{\text{plain}} \cup \Sigma_{\text{call}} \cup \Sigma_{\text{ret}}$ and $V = V^0 \cup V^1$ if every rule in P is in one of the following forms:

(1) $L \to \epsilon$;
(2) $L \to iL_1$, where $i \in \Sigma$, and if $L \in V^0$ then (1) $i \in \Sigma_{\text{plain}}$ and (2) $L_1 \in V^0$;
(3) $L \to \langle aL_1 b\rangle L_2$, where $\langle a \in \Sigma_{\text{call}}$, $b\rangle \in \Sigma_{\text{ret}}$, $L_1 \in V^0$, and if $L \in V^0$, then $L_2 \in V^0$.

The above definition imposes constraints on how V^0 and V^1 nonterminals can be used in a rule. For example, in $L \to \langle aL_1 b\rangle L_2$, nonterminal L_1 must be a well-matched nonterminal; so P cannot include rules such as $L_1 \to \langle aL_3$, since L_1 is supposed to generate only well-matched strings.

The above constraints guarantee that we can identify paired call and return symbols simply by their locations. For example, consider the following string from a general VPG:

$$\langle a\langle acb\rangle.$$

Even without knowing the rules, we can determine the pairing as follows: We begin with an empty stack T and read symbols in order, ignoring the plain symbols. When we read a call symbol, we push it to T; when we read a return symbol, we match it with the top of T, and if T is empty, then the return symbol is pending. After we read all symbols, call symbols remained in T are also pending. For the above string, the stack T has the following transitions:

$$\bot \xrightarrow{\text{read } \langle a} \langle a \cdot \bot \xrightarrow{\text{read } \langle a} \langle a \cdot \langle a \cdot \bot \xrightarrow{\text{read } b\rangle} \langle a \cdot \bot.$$

Therefore, the first call symbol $\langle a$ is pending, and the second call symbol $\langle a$ is paired with the return symbol $b\rangle$. Note that, in general VPGs, paired $\langle a$ and $b\rangle$ are not necessarily generated from the same matching rule; they could also be derived from pending rules.

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.

3 RELATED WORK

Most parser libraries rely on the formalism of **Context-Free Grammars (CFGs)** and user-defined semantic actions for generating parse trees. Many CFG-based parsing algorithms have been proposed in the past, including LL(k), LR(k) [11], Earley [14], CYK [7, 24, 44], among many others. LL(k) and LR(k) algorithms are commonly used, but their input grammars must be unambiguous. Users often have to change/refactor their grammars to avoid conflicts in LL(k) and LR(k) parsing tables, a nontrivial task. Earley, CYK, and GLR parsing can handle any CFG, but their worst-case running time is $O(n^3)$. In contrast, our VPG parsing accepts ambiguous grammars and is linear time.

Our VPG parsing algorithm relies on *derivatives*. One major benefit of working with derivatives is that it is amenable to formal verification, as proofs related to derivatives involve algebraic

transformations on symbolic expressions; they are easier to develop in proof assistants than it is to reason about graphs (required when formalizing LL and LR algorithms). Brzozowski [6] first presented the concept of derivatives and used it to build a recognizer for regular expressions. The idea was revived by Owens et al. [34] and generalized to generate parsers for CFGs [30], with an exponential worst-case time complexity. More recently, a symbolic approach [21] for parsing CFGs with derivatives was presented with cubic time complexity. Derivative-based CFG parsing is further improved with generalized zippers [9]. Danielsson [8] described a monadic parser combinator library, which used derivatives to run parsers; the library had a machine-checked proof in Agda [33], a dependently typed functional programming language. Finally, Edelmann et al. [15] presented a formally verified, derivative-based, linear-time parsing algorithm for LL(1) context-free expressions.

Owl is an open-source project [20] that provides a parser generator for VPGs. It has the same goal as our work, but differs in the following critical aspects: (1) Owl supports only well-matched VPGs, while our parsing library supports general VPGs; (2) Owl adopts a different algorithm and is implemented in an imperative way, while our parsing library is derivative-based and functional; (3) Owl does not provide formal assurance, while our parsing library is formally verified in Coq; (4) Owl rejects ambiguous VPGs, while our parsing library accepts ambiguous grammars and generates parse forests; and (5) Owl does not support semantic actions embedded in grammars, while our parsing library accepts semantic actions.

Due to parsers' importance to security, many efforts have been made to build secure and correct parsers. One obvious approach is testing, through fuzz testing or differential testing (e.g., Petsios et al. [37]). However, testing generally cannot show the absence of bugs. Formal verification has also been applied to the building of high-assurance parsers. Jourdan et al. [23] applied the methodology of translation validation and implemented a verified parser validator for LR(1) grammars. RockSalt [31] included a verified parser for regular expression-based DSL. Barthwal and Norrish [4] presented a verified SLR parser. Lasser et al. [26] and Edelmann et al. [15] presented verified LL(1) parsers but we are not aware of fully verified LL(k) parsers. Koprowski and Binsztok [25] implemented a formally verified parser generator for **Parsing Expression Grammars (PEG)**. Ramananandro et al. [38] presented a verified parser generator for tag-length-value binary message format descriptions. There are also verified parsers for general CFGs [5, 17, 39]. These parsers are limited in their performance on real-world grammars, with the worst-case complexity of $O(n^3)$. The reason is that they have to handle highly ambiguous CFGs, whose nondeterminism hinders the parsers' performance. Recently, Lasser et al. [27] implemented a verified ALL(*) parser, which is the algorithm behind ANTLR4; however, it did not provide a linear-time worst-case guarantee. We formalize our derivative-based, VPG parsing algorithm and its correctness and linear-time performance proofs in Coq.

4 VPG FRONTEND OVERVIEW

The focus of this article is on VPG-based parsing. However, since users are already familiar with CFGs, our system takes as input a CFG-based surface grammar, which is called a *tagged CFG*. In this section, we will give a brief overview of tagged CFGs and some application scenarios; we also discuss tagged CFGs' expressiveness and limitations. In Section 7, we will discuss tagged CFGs in detail and how to convert a tagged CFG to a VPG if the tagged CFG passes a validation process. Once a VPG is obtained, we can use the algorithm described in Section 6 to produce a parser.

Tagged CFG overview. Roughly speaking, a tagged CFG is a CFG enhanced with tagging information about what terminals are call and return symbols; in addition, we allow regular operators for the convenience of specification. As examples, the two subfigures in Figure 1 present tagged

```
1  json   : value ;
2  obj    : <'{' pair (',' pair)* '}'>
3         | <'{' '}'> ;
4  pair   : STRING ':' value ;
5  arr    : <'[' value (',' value)* ']'>
6         | <'[' ']'> ;
7  value  : STRING | NUMBER  | obj | arr
8         | 'true' | 'false' | 'null' ;
```

```
1  document  : prolog? misc* element misc* ;
2  prolog    : XMLDeclOpen attribute* SPECIAL_CLOSE ;
3  content   : chardata?
4              ((element | reference | CDATA | PI | COMMENT)
5                  chardata?)* ;
6  element   : <OpenTag content CloseTag> | SingleTag ;
7  reference : EntityRef | CharRef ;
8  attribute : NAME '=' STRING ;
9  chardata  : TEXT | SEA_WS ;
10 misc      : COMMENT | PI | SEA_WS ;
```

Fig. 1. Tagged CFGs for JSON and XML. Nonterminals such as `json` are in lowercase and terminals such as `STRING` are in uppercase. Regular operators including `?`, `+` and the Kleene Star `*` are allowed. Two additional operators, the left bracket `<` and the right bracket `>`, specify the call and return symbols. String literals such as `'true'` are enclosed in single quotation marks `'`. The lexer rules for the terminals are omitted.

CFGs for JSON and XML documents, respectively; more examples of tagged CFGs can be found in Appendix F.

By placing a left bracket < before a terminal, the user can mark it as a call symbol; similarly, a right bracket > after a terminal marks it as a return symbol. We mentioned in Section 2 that the locations of call and return symbols in a VPG determine how they are matched. Similarly for a tagged CFG, matched call and return symbols should appear in the same rule.

Expressiveness and limitations. In practice, we can make tagged CFGs more expressive by utilizing a lexer with an extended notion of call and return symbols. Take XML as an example. If individual characters were considered as terminals, then the XML grammar could not be expressed as a tagged CFG; this is because an XML tag is complex and includes components such as the tag name and the tag attributes. However, if a whole XML tag is treated as a token in an extended lexer, then the XML grammar can be expressed as a tagged CFG and is convertible to a VPG, as our evaluation shows. For instance, the call symbol for XML is the open tag, which comprises the tag name and also possible attributes.

```
OpenTag = '<' Name attribute* '>'
```

An example open tag is as follows:

```
< p id = "a" >.
```

Since the syntax of such tags can be specified by regular expressions, we can treat it as a single token and extend the lexer to store semantic values such as the tag name `p` and the attributes `id = "a"`, which can be performed efficiently [13].

As will be discussed in Section 7, the primary limitation of a tagged CFG is that it must pass validation so we can convert it to a VPG. Approximately, it means that each *dependency loop* in the grammar must be in at least one pair of matched call and return symbols. A dependency loop is there when a nonterminal L can be used to generate itself via a set of rules; this notion will be discussed formally in Definition 7.3. For example, the following grammar includes a dependency loop, as the right-hand side of the first rule uses L itself:

$$L \to aLLb \mid c.$$

The previous grammar would not pass the validation; however, the user can use the following tags to make it a valid tagged CFG, by ensuring that the dependency stays within $\langle a$ and $b\rangle$:

$$L \to \langle aLLb\rangle \mid c.$$

In this case, we describe the dependency loop as *well founded*. For the JSON grammar in Figure 1, the call-return pairs (`'['`, `']'`) and (`'{'`, `'}'`) ensure that all dependency loops are well founded.

Because of the restriction of dependency loops, left recursion is not supported by tagged CFGs. However, in practice, we can easily remove *direct left recursion* by performing grammar refactoring [36].

5 VPG-BASED RECOGNITION

Before presenting a parsing algorithm for VPGs, we first present an algorithm for converting a well-matched VPG into a deterministic PDA for recognition using a derivative-based algorithm. The resulting PDA recognizes the same set of strings as the input VPG.

To compute a recognizer, we utilize the notion of derivatives [6, 30, 34]. The derivative of a language $\mathcal{L}$ with respect to an input symbol c is the residual set of strings of those in $\mathcal{L}$ that start with c:

$$\delta_c(\mathcal{L}) = \{w \mid cw \in \mathcal{L}\}.$$

Before giving formal derivative definitions, we first review derivatives for regular grammars [6, 34] in Example 5.1. We then extend them to VPGs in Example 5.2, where we also discuss informally the intuition of the states, the stack, and the transition function of the PDA that is converted from an input VPG. Throughout this section and Section 6, we use the VPG G shown in Figure 2(c), where the start nonterminal is L, and the right-regular grammar G', which is based on the VPG G by deleting the matching rules.

Example 5.1 (From Regular Grammars to DFAs). For regular grammars, we can build a **deterministic finite automaton (DFA)** for recognition, where a state S is a set of nonterminals. When reading a symbol c, we transfer from the current state S to the next state $\delta_c(S)$, a derivative of S with respect to the symbol c:

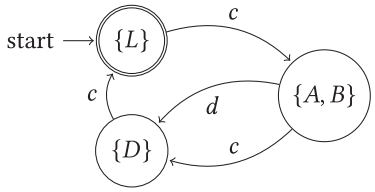
$$\delta_c\,(S) = \{L' \mid L \in S, (L \to cL') \in P\}\,.$$

We define the initial state S_0 as the singleton set containing the start nonterminal. For instance, suppose the input string is $s = ccc$. The derivatives for the grammar G' in Figure 2(a) are as follows:

$$\begin{aligned} S_0 &= \{L\},\\ S_1 &= \delta_c(S_0) = \{A, B\},\\ S_2 &= \delta_c(S_1) = \{D\},\\ S_3 &= \delta_c(S_2) = \{L\}. \end{aligned}$$

$$L \to cA \mid cB \mid \epsilon$$
$$A \to cD$$
$$B \to dD$$
$$D \to cL$$

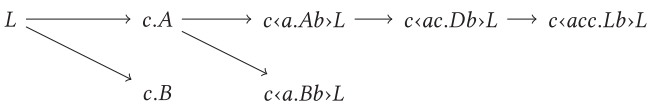
(a) A regular grammar, which is based on the VPG in Figure 2(c) by deleting the matching rules.

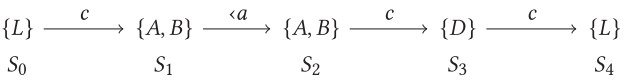


(b) The recognizer DFA computed using derivatives for the grammar in Figure 2(a).

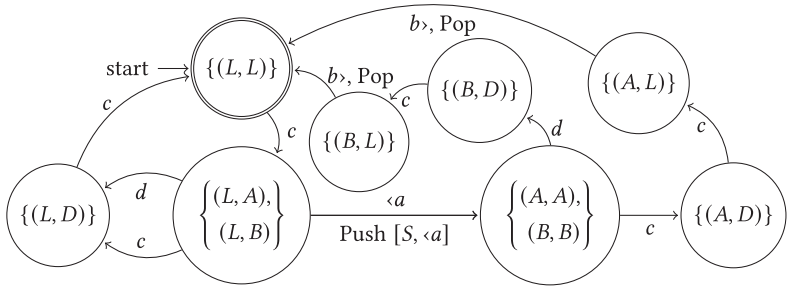
$$L \to cA \mid cB \mid \epsilon$$
$$A \to cD \mid \langle aAb\rangle L \mid \langle aBb\rangle L$$
$$B \to dD$$
$$D \to cL$$

(c) An example VPG, where the start nonterminal is L.

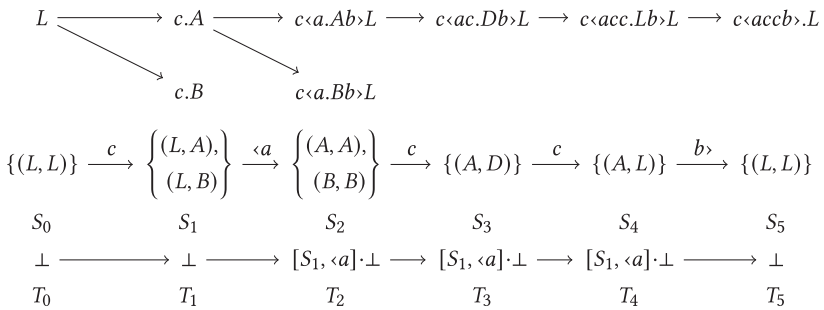




(d) An attempt to build a recognizer finite automaton for the VPG shown in Figure 2(c), given the string $s = c\langle accb\rangle$. A dot "." is used to indicate the current position for the recognition. At the state S_4, we should apply the matching rule $A \to \langle aAb\rangle L$ to derive the return symbol $b\rangle$, but we cannot retrieve the rule based on only the state S_4.



(e) The recognizer PDA for the VPG shown in Figure 2(c).



(f) With the help of the context and the stack, we can transfer from state S_4 to state S_5.

Fig. 2. Figures for Examples 5.1 and 5.2.

We accept the string if and only if the final state includes a nonterminal that can derive the empty string. For the example input string, L is in the final state and can derive the empty string; thus, the string ccc is accepted.

Using the notion of derivatives, we can compute derivatives for all possible input symbols and for all states; the process terminates. This way, we can produce a DFA. Figure 2(b) shows the DFA for the regular grammar G'.

Example 5.2 (From VPGs to Recognizer PDAs). Based on the derivatives for regular grammars, our first attempt would be to retain the same kind of states when producing PDA for VPGs. Considering the grammar G in Figure 2(c) and the string $s = c‹accb›$, the attempt is summarized in Figure 2(d) and detailed as follows: Our first two states would be as follows: $S_0 = \{L\}$; $S_1 = \delta_c(S_0) = \{A, B\}$.

Next, we would process the symbol $‹a$, and try to compute the following derivative: $\delta_{‹a}(S_1) = \delta_{‹a}(\{A, B\})$. Here, we would use the nonterminals A and B to derive $‹a$. At first glance, one might assume that the derivative of $\{A, B\}$ with respect to $‹a$ would simply be $\{Ab›L, Bb›L\}$; however, this definition can lead to an infinite number of states, e.g., the derivative of $\{Ab›L, Bb›L\}$ with respect to $‹a$ would be $\{Ab›Lb›L, Bb›Lb›L\}$. Therefore, we would instead apply the two matching rules in the grammar and produce the next state $S_2 = \{A, B\}$, which only contains the nonterminals to be parsed next. In other words, we would define the derivative function $\delta_{‹a}$ as follows:

$$\delta_{‹a}(S) = \{L_1 \mid L \in S, (L \to ‹aL_1b›L_2) \in P\}.$$

For the next two plain symbols c, the derivatives would be identical to those of regular grammars and thus before reading $b›$, we would have the following states: $S_0 = \{L\}$; $S_1 = \delta_c(S_0) = \{A, B\}$; $S_2 = \delta_{‹a}(S_0) = \{A, B\}$; $S_3 = \delta_c(S_1) = \{D\}$; $S_4 = \delta_c(S_2) = \{L\}$.

When processing the next symbol $b›$, however, we would discover that the state S_4 alone is insufficient to determine which rule to apply for deriving $b›$, as shown in Figure 2(d). First, we know that the state S_4 includes a nonterminal L that derives the empty string ϵ. Therefore, we should use the matching rules used last time that derive both the nonterminal L in the state S_4 and the symbol $b›$. However, based on only the state S_4, we would not know which matching rule should be used to produce $b›$.

To solve this issue, we extend states to be *sets of pairs of nonterminals* and also use a stack. A nonterminal pair (L_1, L_2) in a state tells that the rest of the input should match L_2 and the current context is L_1. The *context* is the nonterminal that is used to derive L_2 without consuming an unmatched call symbol before getting to L_2. Formally, it means that there exists a derivation sequence $L_1 \to^* \omega_1 L_2 \omega_2$, where ω_1 is a sequence of terminals and *does not contain an unmatched call symbol* (meaning that if ω_1 contains a call symbol, then it also contains a matching return symbol), and ω_2 is a sequence of terminals or nonterminals. We define the initial state S_0 as $\{(L, L)\}$, where L is the start nonterminal. It means that the input string should match L and the context is also L, since L can be derived from itself (in zero steps) without generating an unmatched call symbol.

For the example, we now have the following states:

$$\begin{aligned}
S_0 &= \{(L, L)\},\\
S_1 &= \delta_c\{(L, A), (L, B)\},\\
S_2 &= \delta_{‹a}(S_1) = \{(A, A), (B, B)\},\\
S_3 &= \delta_c(S_2) = \{(A, D)\},\\
S_4 &= \delta_c(S_3) = \{(A, L)\}.
\end{aligned}$$

Notice that when transitioning from S_1 to S_2, there is a context switch from L to A or B, as there is an unmatched call symbol $‹a$ that is encountered using the rules $A \to ‹aAb›L$ and $A \to ‹aBb›L$. For this transition, we will also need to use a stack to store the old context information; in particular,

our PDA will push S_1 and $\langle a$ to the stack, so when the return symbol $b\rangle$ is encountered, we can use that stack information to look up the old context and transition the PDA to state $\{(L, L)\}$.

As shown in Figure 2(f), for the grammar G, we have the following state and stack when reading the symbol $b\rangle$:

$$S_4 = \{(A, L)\},$$
$$T_4 = [S_1, \langle a] \cdot \bot.$$

To determine which rule to apply, we first find that the nonterminal L in the current state S_4 derives the empty string ϵ and has context A. Based on the context and the call symbol $\langle a$ stored at the top of T_4, we can deduce that a rule of the form $\hat{L} \to \langle aA b\rangle \hat{L}_2$ was used to derive $\langle a$, where $\hat{L}$ and $\hat{L}_2$ are unknown nonterminals. To identify them, we use the state S_1 at the top of T_4 and consider possible candidate matching rules. These rules could include:

$$(\hat{L} \to \langle aA b\rangle \hat{L}_2) \in P, \text{ for } \hat{L} \in \{L \mid (L', L) \in S_1\}.$$

The above is enough for us to identify that $A \to \langle aA b\rangle L$ is the matching rule; note that in general there may be multiple possible matching rules.

For the example input string $s = c\langle accb\rangle$, the final state is $S_5 = \{(L, L)\}$. We finish the recognition by accepting the string s, since L, the next nonterminal to parse, can derive the empty string ϵ.

Following the above process for all input symbols and all states, we can build the recognizer PDA for the VPG G shown in Figure 2(e); the detailed algorithm will be presented later in this section.

Given the above discussion, we have the following PDA states and stacks:

Definition 5.1 (PDA States and Stacks). Given a VPG $G = (\Sigma, V, P, L_0)$, we introduce a PDA whose states are subsets of $V \times V$ and whose stack contains stack symbols of the form $[S, \langle a]$, where S is a PDA state and $\langle a \in \Sigma_{\text{call}}$ is a call symbol. We write $\bot$ for the empty stack and $[S, \langle a] \cdot T$ for a stack whose top is $[S, \langle a]$ and the rest is T. Intuitively, the stack remembers a series of past contexts, which are used for matching future return symbols. We call a pair (S, T) a *configuration*, with S being the state and T being the stack.

Given a VPG $G = (V, \Sigma, P, L_0)$, we define three kinds of derivative functions: (1) δ_c is for when the next input symbol is a plain symbol c; (2) $\delta_{\langle a}$ for when the next input symbol is a call symbol $\langle a$; and (3) $\delta_{b\rangle}$ for when the next input symbol is a return symbol $b\rangle$. Each function takes the current state S and the top stack symbol and returns a new state as well as an action on the stack (expressed as a lambda function). Note that δ_c and $\delta_{\langle a}$ do not need information from the stack; therefore, we omit the top stack symbol from their parameters.

We note that previous work on derivatives [6, 30, 34] uses a direct representation of a language (for instance, a grammar) to represent the derivative result. In this work, however, what we compute over derivatives is *configurations*, which include states and stacks. A later definition (Definition 5.3) shows that a configuration of a recognizer PDA represents a language. We also note that the derivative functions defined below take only necessary information from a configuration to make clear what information is needed for a particular derivative function.

Definition 5.2 (Derivative Functions).
(1) $\delta_c(S) = (S', \lambda T.T)$, where

$$S' = \{(L_1, L_3) \mid (L_1, L_2) \in S \wedge (L_2 \to c L_3) \in P\}.$$

For a plain symbol $c \in \Sigma_{\text{plain}}$, it checks each pair (L_1, L_2) in the current state S, and if there is a rule $L_2 \to c\ L_3$, then the pair (L_1, L_3) becomes part of the new state. In addition, the stack is left unchanged.

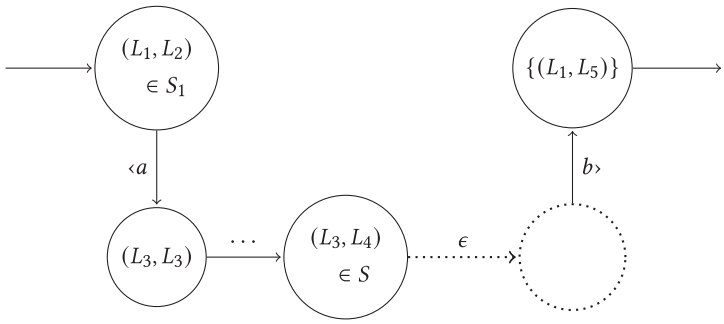


Fig. 3. An example of state transitions: Assuming there is a rule $L_2 \to \langle aL_3b\rangle L_5$, (L_1, L_2) transfers to (L_3, L_3) with symbol $\langle a$, and finally transfers to (L_1, L_5) with symbol $b\rangle$.

(2) $\delta_{\langle a}(S) = (S', \lambda T.[S, \langle a] \cdot T)$, where

$$S' = \{(L_3, L_3) \mid (L_1, L_2) \in S \wedge (L_2 \to \langle aL_3b\rangle L_4) \in P\}.$$

For a call symbol $\langle a \in \Sigma_{\text{call}}$, it checks each pair (L_1, L_2) in the current state S; if there is a rule $L_2 \to \langle aL_3b\rangle L_4$, then the pair (L_3, L_3) becomes part of the new state; note there is a context change, since a call symbol is encountered. In addition, the old state together with $\langle a$ is pushed to the stack.

(3) $\delta_{b\rangle}(S, [S_1, \langle a]) = (S', \text{tail})$, where tail is a function that returns the tail of the stack, and

$$S' = \{(L_1, L_5) \mid (L_1, L_2) \in S_1 \wedge (L_3, L_4) \in S \wedge (L_4 \to \epsilon) \in P \ \wedge (L_2 \to \langle aL_3b\rangle L_5) \in P\}.$$

For a return symbol $b\rangle \in \Sigma_{\text{ret}}$ and a stack top $[S_1, \langle a]$, it checks each pair (L_1, L_2) in the state S_1 of the stack top symbol; if there is a pair (L_3, L_4) in the current state S, L_4 can derive the empty string, and there is a rule $L_2 \to \langle aL_3b\rangle\ L_5$, then the pair (L_1, L_5) becomes part of the new state; note that it checks $L_4 \to \epsilon$ to ensure that the current level is finished before returning to the upper level. In addition, the stack top is popped from the stack. Figure 3 presents a drawing that depicts the situation when a return symbol is encountered.

We formalize the semantics of PDA configurations as sets of accepted strings:

Definition 5.3 (Semantics of PDA Configurations). We write $(S, T) \rightsquigarrow w$ to mean that a terminal string w can be accepted by the configuration (S, T). It is defined as follows:

(1) $(S, \bot) \rightsquigarrow w$ if $\exists (L_1, L_2) \in S$, s.t. $L_2 \to^* w$,
(2) $(S, [S', \langle a] \cdot T') \rightsquigarrow w_1 b\rangle w_2$ if $\exists (L_3, L_4) \in S$ s.t.
  (a) $L_4 \to^* w_1$, and
  (b) $\exists (L_1, L_2) \in S', \exists L_5, (L_2 \to \langle aL_3b\rangle L_5) \in P \wedge (\{(L_1, L_5)\},\ T') \rightsquigarrow w_2$.

The correctness of derivative functions is stated in the following theorem, whose correctness proof is detailed in Appendix A. Take the case of δ_c as an example: The theorem states that (S, T) matches cw iff the configuration after running δ_c matches w (the string after consuming c).

Theorem 5.1 (Derivative Function Correctness).

- *Assume* $\delta_c(S) = (S', \lambda T.T)$ *for a plain symbol* c. *Then* $(S, T) \rightsquigarrow cw$ *iff* $(S', T) \rightsquigarrow w$.
- *Assume* $\delta_{\langle a}(S) = (S', \lambda T.[S, \langle a] \cdot T)$ *for a call symbol* $\langle a$. *Then* $(S, T) \rightsquigarrow \langle aw$ *iff* $(S', [S, \langle a] \cdot T) \rightsquigarrow w$.
- *Assume* $\delta_{b\rangle}(S, [S_1, \langle a]) = (S', tail)$ *for a return symbol* $b\rangle$. *Then* $(S, [S_1, \langle a] \cdot T) \rightsquigarrow b\rangle w$ *iff* $(S', T) \rightsquigarrow w$.

ALGORITHM 1: Constructing the recognizer PDA.

Input : A VPG $G = (V, \Sigma, P, L_0)$ where $\Sigma = \Sigma_{\text{call}} \cup \Sigma_{\text{plain}} \cup \Sigma_{\text{ret}}$, δ;
Return : The initial state S_0, the set of all produced states A, the set of acceptance states A_{acc}, and the set of transitions $\mathcal{T}$;

1 $S_0 \leftarrow \{(L_0, L_0)\}$;
2 Initialize the set for new states $N \leftarrow \{S_0\}$;
3 Initialize the set for all produced states $A \leftarrow N$;
4 Initialize the set for transitions $\mathcal{T} \leftarrow \{\}$;
5 **repeat**
6 $\quad N' \leftarrow \{(i, f, S, S') \mid (S', f) = \delta_i(S),\ S \in N,\ \text{and } i \in \Sigma_{\text{call}} \cup \Sigma_{\text{plain}}\}$;
7 $\quad$ Add edge (S, S') marked with (i, f) to $\mathcal{T}$, where $(i, f, S, S') \in N'$;
8 $\quad$ Compute the set of stack elements: $R \leftarrow \{[S, \langle a] \mid S \in A \text{ and } \langle a \in \Sigma_{\text{call}}\}$;
9 $\quad$ Compute transitions with return symbols and stack elements in R:
10 $\quad\quad N_R \leftarrow \{(b\rangle, r, f, S, S') \mid (S', f) = \delta_{b\rangle}(S, r), S \in A, b\rangle \in \Sigma_{\text{ret}}, r \in R\}$;
11 $\quad$ Add edge (S, S') marked with $(b\rangle, r, f)$ to $\mathcal{T}$, where $(b\rangle, r, f, S, S') \in N_R$;
12 $\quad$ Collect the new states $N \leftarrow \{S' \mid (\_, \_, \_, S') \in N' \vee (\_, \_, \_, \_, S') \in N_R\} - A$;
13 $\quad$ Update the set of all produced states $A \leftarrow A \cup N$;
14 **until** $N = \emptyset$;
15 Compute the states of acceptance configurations $A_{\text{acc}} \leftarrow \{S \mid (L', L) \in S \in A, (L \to \epsilon) \in P\}$;

With those derivative functions, we can convert a VPA to a PDA, whose set of states is the least solution of the following equation; it makes sure that states are closed under derivatives:

$$\begin{aligned} A \quad = \quad & A \cup \{S' \mid c \in \Sigma_{\text{plain}},\ S \in A,\ \delta_c(S) = (S', f)\} \\ & \cup \{S' \mid \langle a \in \Sigma_{\text{call}},\ S \in A,\ \delta_{\langle a}(S) = (S', f)\} \\ & \cup \{S' \mid b\rangle \in \Sigma_{\text{ret}},\ \langle a \in \Sigma_{\text{call}},\ S \in A,\ S'' \in A,\ \delta_{b\rangle}(S, [S'', \langle a]) = (S', f)\}. \end{aligned}$$

We note that the least solution to the previous equation may include unreachable states, since the last line of the equation considers all $(S, [S', \langle a])$ without regard for whether such a configuration is possible. This may make the resulting PDA contain more states and occupy more space for the PDA representation than necessary. However, unreachable states do not affect the linear-time parsing guarantee of VPG parsing, as during parsing those unreachable states are not traversed; further, during experiments, we did not experience space issues when representing PDA states and transitions.

Algorithm 1 is an iteration-based method to solve the equation for the least solution, where the returned S_0 is the initial state, A is the set of all states, and $\mathcal{T}$ is the set of edges between states. For an iteration, N is the set of states that the algorithm should perform derivatives on. Line 7 then performs derivatives using call and plain symbols, and line 10 performs derivatives using return symbols.

At the end of each iteration, the following invariants are maintained: (1) $N \subseteq A$; (2) for state $S \in A - N$ and $i \in \Sigma_{\text{call}} \cup \Sigma_{\text{plain}}$, if $\delta_i(S) = (S', f)$, then $S' \in A$; (3) for states $S, S' \in A - N$, $\langle a \in \Sigma_{\text{call}}$, and $b\rangle \in \Sigma_{\text{ret}}$, if $\delta_{b\rangle}(S, [S', \langle a]) = (S'', f)$, then $S'' \in A$. With these invariants, when N becomes empty, A is closed under derivatives.

Once the PDA is constructed from a VPG, it can be run on an input string in a standard way. For completeness, we include the definition here so we can state the correctness theorem formally. Recall that a runtime configuration of a recognizer PDA is a pair (S, T), where S is a state and T is a stack, denoted as $T = t_1 \cdot t_2 \cdots t_k \cdot \bot$, where $t_i = [S_i, \langle a_i]$ for $i = 1..k$, $\langle a_i \in \Sigma_{\text{call}}$ is a call symbol,

t_1 the top of T, and $\perp$ the empty stack. The PDA's initial configuration is $(\{(L_0, L_0)\}, \perp)$ and its acceptance configurations are defined as follows:

Definition 5.4 (PDA Acceptance Configurations). Given a VPG $G = (V, \Sigma, P, L_0)$, a pair (S, T) is an *acceptance configuration* if $T = \perp$, and $\exists (L, L') \in S$ s.t. $(L' \to \epsilon) \in P$.

Definition 5.5 (Recognizer PDA Execution). The runtime execution $\mathcal{F}$ of a PDA $(S_0, A, \mathcal{T})$ is defined as follows, where S_0 is the start state, A is the set of states, and $\mathcal{T}$ is the set of configuration transitions:

$$\mathcal{F} : (i, S, T) \mapsto (S', T'),$$

where

(1) if $i \in \Sigma_{\text{call}} \cup \Sigma_{\text{plain}}$, then $(S, S') \in \mathcal{T}$ and is marked with (i, f), and $T' = f(T)$;
(2) if $i \in \Sigma_{\text{ret}}$ and $T = t \cdot T'$, then $(S, S') \in \mathcal{T}$ and is marked with (i, t, f), and $T' = f(T)$.

Given an input string $w = w_1 \dots w_n$, we say the PDA accepts w if there exists a sequence of configurations $(S_0, T_0), \dots, (S_n, T_n)$, so

$$\begin{aligned} &(S_0, T_0) = (\{(L_0, L_0)\}, \perp), \\ &(S_i, T_i) = \mathcal{F}(w_i, S_{i-1}, T_{i-1}), \text{for } i \in [1, n], \text{and} \\ &(S_n, T_n) \text{ is an acceptance configuration.} \end{aligned}$$

Otherwise, w is rejected.

Then PDA correctness can be stated as follows. We detail the correctness proof in Appendix A.

Theorem 5.2 (PDA Correctness). *For VPG G and its start symbol L_0, a string $w \in \Sigma^*$ can be derived from L_0 (i.e., $L_0 \to^* w$) iff w is accepted by the corresponding PDA.*

For converting general VPGs (i.e., with pending rules) to PDAs, a couple of changes need to be made to the derivative-based approach: (1) The derivative functions need to consider also pending rules; (2) the acceptance stack may be nonempty because of pending call symbols. The construction is discussed in Appendix B.

The time complexity of Algorithm 1. Algorithm 1 terminates when no new state can be produced. Since a PDA state contains pairs of nonterminals, the number of states produced at the end $|A|$ is bounded by $O(2^{|V|^2})$. For each produced state, the derivative functions are called at line 6 and line 9. The number of times they are called across all iterations is no more than $|A| \times |\Sigma_{\text{call}} \cup \Sigma_{\text{plain}}| + |A| \times |\Sigma_{\text{ret}}| \times |A| \times |\Sigma_{\text{call}}|$, which is bounded by $O(|A|^2 |\Sigma|^2)$. Among the three derivative functions, the one for return symbols has the largest time complexity. Recall that in the definition of $\delta_{b\rangle}(S, [S_1, \langle a])$, we consider each pair (L_1, L_2) in S_1, find a matching rule "$L_2 \to \langle a L_3 b \rangle L_5$," and check if (L_3, L_4) is in S for some L_4 that can derive the empty string. Therefore, the cost is bounded by $|S_1| \times |P| \times |S|$, which is further bounded by $O(|V|^4 |P|)$. At lines 7 and 11, at most $O(|A|^2)$ edges are added across all iterations. At lines 12 and 13, in one iteration collecting the states takes $O(|N'| + |N_R|) = O(|A|)$, and computing the difference and union of the sets takes $O(|N| + |A|)$; so the time complexity is $O(|A|)$. Therefore, the total time complexity is $O(|A|^2 |\Sigma|^2 |V|^4 |P|) + O(|A|^2) = O(2^{2|V|^2} |V|^4 |\Sigma|^2 |P|)$.

6 VPG-BASED PARSING

Parsing is a process to build the parse trees of a given string. It is equivalent to finding the rule sequences that generate the string. To achieve this, our VPG-based parsing framework includes two steps: We first run a *parser PDA* on the string, which leaves a trace of PDA states that can be treated as a parse forest; we then execute *extraction functions* on the forest to extract all valid parse trees. We further propose an optimization, which optimizes the second step to an *extraction PDA*: Instead of extracting all parse trees, the extraction PDA extracts a single parse tree. In the

following discussion, we first describe parse trees in the context of VPGs, then discuss the parser PDA in Section 6.1, the extraction functions in Section 6.2, and the optimization of extraction as an extraction PDA in Section 6.3. We then discuss the correctness of our parsing framework in Section 6.4 and its time complexity in Section 6.5. Both the correctness proof and the time-complexity proof have been mechanized in the proof assistant Coq[2]; we provide more details about the Coq mechanization in Section 6.6.

Before proceeding, we note that the parsing approach in this section is different from the approach described in the conference version. The conference version [22] presented a parsing method that includes a parser PDA, a pruner PDA, and an extractor. The correctness of parsing was verified, but we were not able to formalize and verify its linear-time performance guarantee when extracting a parse tree; the structure of the PDA states of that method made it hard to reason about the performance (and also correctness). Our new method improves the parsing function and the parser PDA so we can build a verified extraction function and verify the performance: For an input string w that has k valid parse trees, we verify that the time complexity of parsing with the new approach is $O(k|w|)$. Note that, when only one of the parse trees is needed, later in this article, we will describe an extractor PDA that extracts just one parse tree and prove that it takes linear time ($O(|w|)$).

Given a grammar with a rule set P, we define a notion of *dotted rules*. A dotted rule is a rule in P with an extra dot on the right-hand side. Intuitively, the dot tells the parsing position: To the right of the dot there is a nonterminal, which tells what should be parsed next.

Definition 6.1 (Dotted Rules $\dot{P}$). Given a well-matched VPG $G = (\Sigma, V, P, L_0)$, we define the set of dotted rules as

$$\begin{aligned}\dot{P} = &\{L \to c.L' \mid L \to cL' \in P\}\\ &\cup \{L \to \langle a.L_1 b\rangle L_2 \mid L \to \langle aL_1 b\rangle L_2 \in P\}\\ &\cup \{L \to \langle aL_1 b\rangle.L_2 \mid L \to \langle aL_1 b\rangle L_2 \in P\}.\end{aligned}$$

Dotted rules are similar to *items* in LR parsing [11], but dots in our dotted rules appear after only a terminal, while dots in items do not have this restriction.

Parse trees. Given a grammar $G = (\Sigma, V, P, L_0)$ and its dotted rules $\dot{P}$, we define a parse tree v as a rule sequence; i.e., $v = [r_1, \ldots, r_n]$, where $r_i \in \dot{P}, 1 \le i \le n$; when $n = 0$, it is an empty parse tree. For two parse trees $v_1 = [r_1, \ldots, r_{n_1}]$ and $v_2 = [r'_1, \ldots, r'_{n_2}]$, we use the notation $v_1 + v_2$ to represent their concatenation $[r_1, \ldots, r_{n_1}, r'_1, \ldots, r'_{n_2}]$. We use $L \Downarrow (w, v)$ for the relation that v can derive a string w starting from nonterminal $L \in V$; its rules are in Figure 4. We call v a *valid* parse tree of w if $L_0 \Downarrow (w, v)$. Note that a rule in Figure 4 produces a sequence of dotted rules. Also, the last rule produces a sequence with $L \to \langle aL_1 b\rangle L_2$ appearing twice, but with different positions for the dot: one after the call symbol $\langle a$ and one after the return symbol $b\rangle$; this is because the call and the return symbol are parsed independently, resulting in the need to differentiate how $L \to \langle aL_1 b\rangle L_2$ is used. During parsing, when we have used a dotted rule to parse a call symbol, but have not read the matched return symbol, we call the dotted rule *unclosed*.

We introduce a set of definitions to extract parts of a dotted rule. We define headNT (r) to be the head nonterminal of r and nextNT (r) to be the next nonterminal after the dot in r:

$$\begin{cases}\text{headNT}\,(L \to c.L') = L;\\ \text{headNT}\,(L \to \langle a.L_1 b\rangle L_2) = L;\\ \text{headNT}\,(L \to \langle aL_1 b\rangle.L_2) = L.\end{cases}$$

[2] https://bitbucket.org/psu_soslab/verifiedvpgparser/src/master/.

$$\frac{(L \to \epsilon) \in P}{L \Downarrow (\epsilon, [])} \qquad \frac{(L \to cL_1) \in P \quad c \in \Sigma_{\text{plain}} \quad L_1 \Downarrow (w_1, v_1)}{L \Downarrow (cw_1, (L \to c.L_1)::v_1)}$$

$$\frac{(L \to \langle aL_1 b \rangle L_2) \in P \quad L_1 \Downarrow (w_1, v_1) \quad L_2 \Downarrow (w_2, v_2)}{L \Downarrow (\langle aw_1 b \rangle w_2, [(L \to \langle a.L_1 b \rangle L_2)] + v_1 + [(L \to \langle aL_1 b \rangle .L_2)] + v_2)}$$

Fig. 4. The big-step parse-tree derivation, assuming a well-matched VPG $G = (\Sigma, V, P, L_0)$.

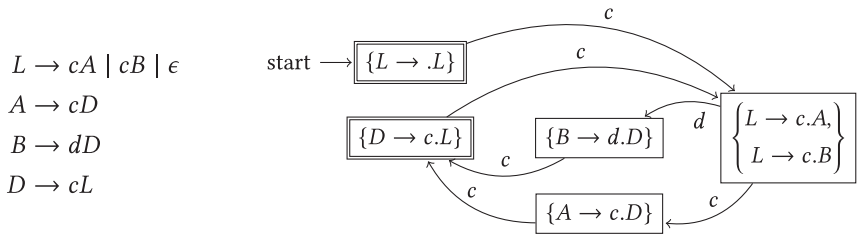


Fig. 5. The parser DFA for the regular grammar in Figure 2(b). The pseudo-rule $L \to .L$ appears only in the initial state, representing a rule in which only its next nonterminal (after the dot) matters.

$$\begin{cases} \text{nextNT}\,(L \to c.L') = L'; \\ \text{nextNT}\,(L \to \langle a.L_1 b \rangle L_2) = L_1; \\ \text{nextNT}\,(L \to \langle aL_1 b \rangle .L_2) = L_2. \end{cases}$$

We abuse the notation and for parse tree $v = [r_1, \ldots, r_n]$ write $\text{headNT}\,(v)$ to be $\text{headNT}\,(r_1)$ and $\text{nextNT}\,(v)$ to be $\text{nextNT}\,(r_n)$. We also define $\text{firstRule}(v)$ to be r_1 and $\text{lastRule}(v)$ to be r_n.

6.1 The Parser PDA

Recall that in the recognizer PDA, a transition between two states for an input symbol is computed based on possible rules that can derive the input symbol. As a result, a state-transition trace that accepts an input string essentially represents the rule sequences that can possibly derive the string. We illustrate this through the following examples, before presenting a formal algorithm for constructing a parser PDA for a VPG:

Example 6.1 (From Recognizer DFAs to Parser DFAs). We begin with regular grammars. Recall the DFA in Figure 2(b). Instead of taking the state as a set of nonterminals, let us replace each state with a set of rules that can produce the last consumed symbol; we call the new automaton the *parser DFA*, as shown in Figure 5. The parser DFA has more states than the recognizer DFA. First, we introduce a pseudo-rule $L \to .L$ to construct an initial state with that rule. In the pseudo-rule, we use a dot "." to indicate the parsing position. Second, the recognizer state $\{D\}$ is split to two states $\{B \to d.D\}$ and $\{A \to c.D\}$ to capture the parsing rule used precisely when transitioning from $\{L \to c.A,\ L \to c.B\}$.

Compared with the recognizer PDA, a trace of states in the parser DFA represents a parse tree. For example, consider the input string $s = ccc$. Its trace in the parser DFA is as the following:

$$\{L \to .L\} \longrightarrow \left\{ \begin{matrix} L \to c.A, \\ L \to c.B \end{matrix} \right\} \longrightarrow \{A \to c.D\} \longrightarrow \{D \to c.L\}.$$

This trace essentially represents the following parse tree v_s of the string ccc:

$$v_s = [L \to c.A; A \to c.D; D \to c.L].$$

In fact, we can extract a parse tree from the trace as follows: We start from the last state and take a rule r that derives the empty string ϵ for the nonterminal next to the dot and define the initial parse tree v_0 as $[r]$. In this example, we have the initial parse tree being

$$v_0 = [D \to c.L].$$

Then, we expand the parse tree by processing states in the trace backward. In particular, for each state S, we add a rule $r \in S$ that is *connected* to the head rule of the current parse tree v, i.e.,

$$\text{nextNT}\,(r) = \text{headNT}\,(v)\,, \text{where } r \in S,$$

and prepend r to the current parse tree v. For example, for the first step, we add $A \to c.D$ to v_0 to have v_1 as follows; similarly, v_2 is built by adding $L \to c.A$ to v_1:

$$v_1 = [A \to c.D; D \to c.L],$$
$$v_2 = [L \to c.A; A \to c.D; D \to c.L].$$

Note that v_2 is the final parse tree we desire. In general, there may be multiple parse trees for an input string, and we need to maintain a set of parse trees during the extraction process.

The above process is essentially the same as how a parser PDA is constructed for a VPG and how a parse tree is extracted from a trace of the parser PDA. We discuss an example next.

Example 6.2 (From Recognizer PDAs to Parser PDAs). Recall the recognizer PDA shown in Figure 2(e); we first discuss how to extend it to the parser PDA shown in Figure 6 and then illustrate it using a concrete example.

Similar to the case of regular grammars, for each pair (L_1, L_2) in a recognizer PDA state, nonterminal L_2 is replaced by the rules that can produce the last consumed symbol. The context L_1, however, is replaced with the most recent unclosed dotted rule; when there is no such a rule, we use a special context called None. The stack T could still store the parser PDA states paired with call symbols; however, since the call symbols have already been recorded in the rules in the states, the stack needs to store only the PDA states.

Now let us reexamine the VPG shown in Figure 2(c) and the input string $s = c\langle accb\rangle$. After running the parser PDA in Figure 6 on the string s, we have the following trace, where the initial state is omitted:

$$\left\{\begin{matrix}(\text{None}, L \to c.A),\\ (\text{None}, L \to c.B)\end{matrix}\right\} \longrightarrow \left\{\begin{matrix}(A \to \langle a.Ab\rangle L, A \to \langle a.Ab\rangle L),\\ (A \to \langle a.Bb\rangle L, A \to \langle a.Bb\rangle L)\end{matrix}\right\} \longrightarrow$$
$$\{(A \to \langle a.Ab\rangle L, A \to c.D)\} \longrightarrow \{(A \to \langle a.Ab\rangle L, D \to c.L)\} \longrightarrow \{(\text{None}, A \to \langle aAb\rangle.L)\}.$$

Similar to the trace in Example 6.1, the preceding trace includes the following parse tree of the string s:

$$[L \to c.A;\ A \to \langle a.Ab\rangle L;\ A \to c.D;\ D \to c.L;\ A \to \langle aAb\rangle.L].$$

Let us extract the above parse tree from the trace in a way similar to the case for regular grammars. Start from the last state, we pick a rule r that has the context None and can derive the empty string ϵ from the next nonterminal to parse. We initialize the parse tree as $v_0 = [r]$. In this example, we have

$$v_0 = [A \to \langle aAb\rangle.L].$$

Now, we trace back to the state $\{(A \to \langle a.Ab\rangle L, D \to c.L)\}$. Since the last symbol is a return symbol, we begin by identifying rules with the next nonterminal that can produce the empty string. In this case, rule $D \to c.L$ is a candidate. In addition, recall that the context rule is the most recent unclosed dotted rule; so the following return symbol should be derived from the context

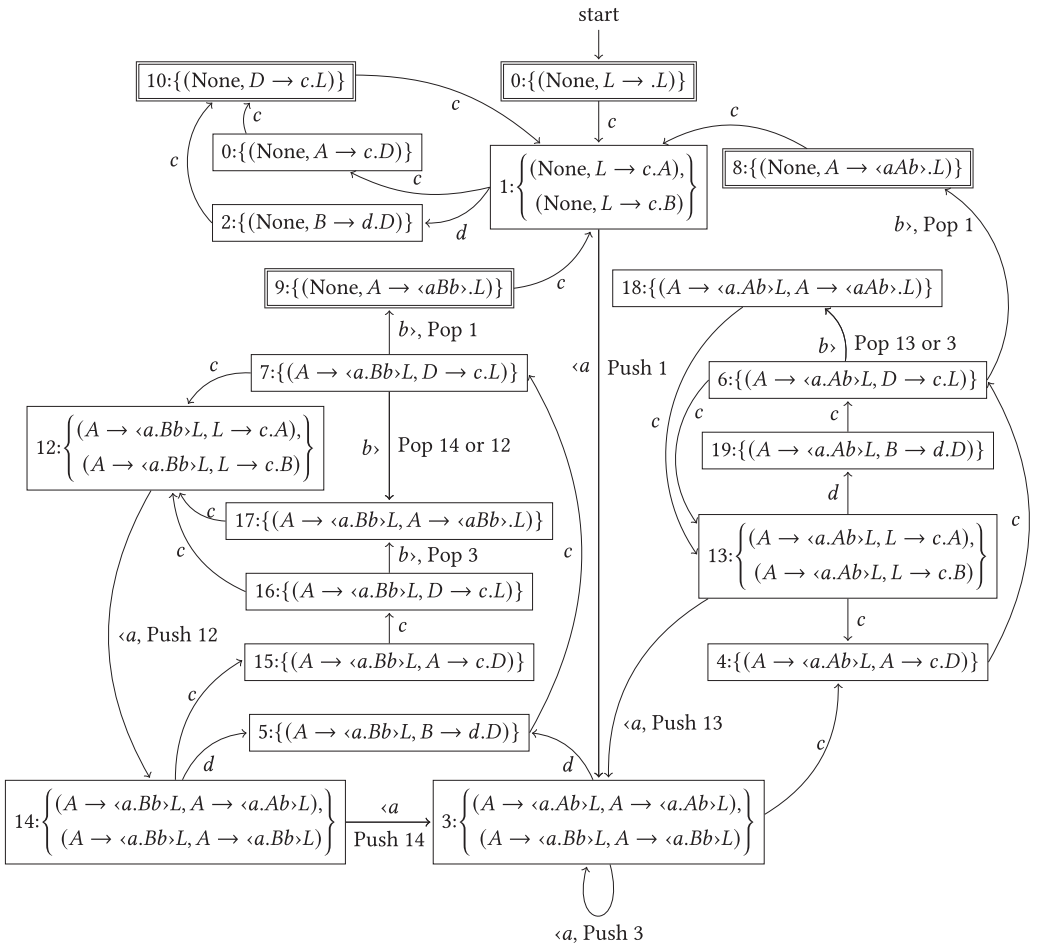


Fig. 6. The parser PDA for the VPG shown in Figure 2(c). Each state is marked with its index; e.g., the initial state is State 0. The stack operation "Push i" means pushing State i to the top of the stack, and "Pop i" means removing the top of the stack when the top is State i.

rule. The context rule is $A \to \langle a.Ab\rangle L$, which corresponds to the current parse tree's head rule $A \to \langle aAb\rangle.L$. Therefore, we can use the rule $D \to c.L$ to extend our parse tree.

$$v_1 = [D \to c.L;\ A \to \langle aAb\rangle.L].$$

Then, we continue to the next state in the trace: $\{(A \to \langle a.Ab\rangle L, A \to c.D)\}$. We first observe that the rule $A \to c.D$ is connected to the head rule $D \to c.L$. In addition, since the symbol to match next is a plain symbol, the context rule should not change and should be $A \to \langle a.Ab\rangle L$. To determine the context of the head rule efficiently, we introduce a stack T to store the current context, as shown below.

$$v_0 = [A \to \langle aAb\rangle.L],\ T_0 = (A \to \langle aAb\rangle.L) \cdot \bot$$
$$v_1 = [D \to c.L;\ A \to \langle aAb\rangle.L],\ T_1 = (A \to \langle aAb\rangle.L) \cdot \bot$$

Note that the stack T_0 could alternatively store the rule $A \to \langle a.Ab\rangle L$; the difference is just about the location of the dot. With the stack, we know that the context is the same and prepend rule

$A \to c.D$ to the parse tree.

$$v_2 = [A \to c.D;\ D \to c.L;\ A \to \langle aAb\rangle.L],\ T_2 = (A \to \langle aAb\rangle.L) \cdot \bot$$

We continue to the next state.

$$\left\{\begin{matrix}(A \to \langle a.Ab\rangle L, A \to \langle a.Ab\rangle L),\\ (A \to \langle a.Bb\rangle L, A \to \langle a.Bb\rangle L)\end{matrix}\right\}$$

To extract a matching rule for the call symbol is fairly simple at this point: We just pick the rule that corresponds to the top of the stack T; in this case it should be $A \to \langle a.Ab\rangle L$. Our final parse tree is as the following:

$$v_3 = [A \to \langle a.Ab\rangle L;\ A \to c.D;\ D \to c.L;\ A \to \langle aAb\rangle.L],\ T_3 = \bot.$$

Again, in general, there may be multiple parse trees for the input string, and we need to maintain a set of parse trees during the extraction.

Now, we formally define parser PDA states and stacks and the derivative functions for well-matched VPGs as follows (the definitions for general VPGs can be found in Appendix C):

Definition 6.2 ($\mathcal{P}_{pln}, \mathcal{P}_{call}, \mathcal{P}_{ret}$). Given a VPG $G = (\Sigma, V, P, L_0)$, based on the dotted rules in $\dot{P}$,

(1) we define a set of plain rules $\mathcal{P}_{\text{pln}}$ as $\{L \to c.L_1 \in \dot{P} \mid c \in \Sigma_{\text{plain}}\}$;
(2) we define a set of call rules $\mathcal{P}_{\text{call}}$ as $\{L \to \langle a.L_1 \in \dot{P}\} \cup \{L \to \langle a.L_1 b\rangle L_2 \in \dot{P}\}$;
(3) we define a set of return rules $\mathcal{P}_{\text{ret}}$ as $\{L \to b\rangle.L_1 \in \dot{P}\} \cup \{L \to \langle aL_1 b\rangle.L_2 \in \dot{P}\}$.

Definition 6.3 (Parser PDA States and Stacks). Given a VPG, we introduce a PDA whose states are subsets of $(\mathcal{P}_{\text{call}} \cup \{\text{None}\}) \times (\dot{P} \cup \{L_0 \to .L_0\})$ and whose stack contains PDA states. Intuitively, the stack remembers a series of past contexts, which are used for matching future return symbols. We call a pair (m, T) a *configuration*, with m being the state and T being the stack. A rule pair (r', r) in state m tells that the current rule is r and the *context rule* is r'. The context rule r' is the rule that generates the last unmatched call symbol; when all call symbols are matched with return symbols, r' is None. Initially, since the parser PDA has not read any symbol, there is no current rule; so we define the initial PDA state as $m_0 = \{(\text{None}, L_0 \to .L_0)\}$, where $L_0 \to .L_0 \notin \dot{P}$ a special rule; they only appear in m_0.

The initial stack T_0 is empty. When reading a call symbol, the parser PDA pushes the current state to its stack; when reading a return symbol, it removes the top of the stack. For a PDA stack T, the function head(T) returns the top of the stack and is defined as

$$\text{head}(T) = \begin{cases}\text{None}, & \text{if } T = \bot;\\ t, & \text{if } \exists t\ T',\ T = t \cdot T'.\end{cases}$$

For parser PDA construction, we define three kinds of derivative functions p_c, $p_{\langle a}$, and $p_{b\rangle}$, similar to the case of recognizer construction.

Definition 6.4 (The Derivative Function p for the Parser PDA). Given a VPG $G = (V, \Sigma, P, L_0)$, suppose the current state of the parser PDA is m and the current stack is T, the transition functions p_c, $p_{\langle a}$ and $p_{b\rangle}$ are defined as follows:

(1) $p_c(m) = (m', \lambda T.T)$, where

$$m' = \{(r', \text{nextNT}(r) \to c.L_1) \mid (r', r) \in m \wedge (\text{nextNT}(r) \to c.L_1) \in \dot{P}\}.$$

For each pair (r', r) in m, the new state keeps the context rule r' and updates the current rule to a rule with head nextNT (r) and that derives c.

ALGORITHM 2: Constructing the parser PDA. Differences from Algorithm 1 are highlighted.

Input : A VPG $G = (V, \Sigma, P, L_0)$ where $\Sigma = \Sigma_{\text{call}} \cup \Sigma_{\text{plain}} \cup \Sigma_{\text{ret}}$, p;
Return : The initial state S_0, the set of all produced states A, the set of acceptance states A_{acc}, and the set of transitions $\mathcal{T}$;

$m_0 \leftarrow \{(\text{None}, L_0 \to .L_0)\}$;
Initialize the set for new states $N \leftarrow \{m_0\}$;
Initialize the set for all produced states $A \leftarrow N$;
Initialize the set for transitions $\mathcal{T} \leftarrow \{\}$;
repeat
  $N' \leftarrow \{(i, f, m, m') \mid (m', f) = p_i(m),\ m \in N,\ \text{and } i \in \Sigma_{\text{call}} \cup \Sigma_{\text{plain}}\}$;
  Add edge (m, m') marked with (i, f) to $\mathcal{T}$, where $(i, f, m, m') \in N'$;
  Compute the set of stack elements: $R \leftarrow A \cup \{\emptyset\}$;
  Compute transitions with return symbols and stack elements in R:
    $N_R \leftarrow \{(b\rangle, r, f, m, m') \mid (m', f) = p_{b\rangle}(m, r), m \in A, b\rangle \in \Sigma_{\text{ret}}, r \in R\}$;
  Add edge (m, m') marked with $(b\rangle, r, f)$ to $\mathcal{T}$, where $(b\rangle, r, f, m, m') \in N_R$;
  Collect the new states $N \leftarrow \{m' \mid (\_, \_, \_, m') \in N' \vee (\_, \_, \_, \_, m') \in N_R\} - A$;
  Update the set of all produced states $A \leftarrow A \cup N$;
until $N = \emptyset$;
Compute the states of acceptance configurations
$A_{\text{acc}} \leftarrow \{m \mid (\text{None}, r) \in m \in A, (\text{nextNT}\,(r) \to \epsilon) \in P\}$;

(2) $p_{\langle a}(m) = (m', \lambda T.\ m \cdot T)$, where

$$m' = \left\{(r_1, r_1) \mid (r', r) \in m \ \wedge\ \left(r_1 = (\text{nextNT}\,(r) \to \langle a.L_1 b\rangle L_2) \in \dot{P}\right)\right\}.$$

For each pair (r', r) in m, the new state changes the context rule and the current rule to a rule with head nextNT (r) and that derives $\langle a$. We further push the current state m to the stack.

(3) $p_{b\rangle}(m, \overline{m_{\text{call}}}) = (m', \text{tail})$, where tail is a function that returns the tail of the stack, and $\overline{m_{\text{call}}} = \text{head}(T)$, and

$$\begin{aligned} m' = \{&(r', \text{nextNT}\,(r) \to \langle aL_1 b\rangle.L_2) \mid (r_1, r_2) \in m \ \wedge\ (r', r) \in \overline{m_{\text{call}}} \wedge \\ &r_1 = (\text{nextNT}\,(r) \to \langle a.L_1 b\rangle L_2) \wedge (\text{nextNT}\,(r_2) \to \epsilon) \in P\}. \end{aligned}$$

We remove the top $\overline{m_{\text{call}}}$ of the stack T and, for each pair (r_1, r_2) in m where the context rule is $r_1 = L \to \langle a.L_1 b\rangle L_2$, we match $b\rangle$ with $\langle a$ and update the current rule to $L \to \langle aL_1 b\rangle.L_2$. We also require that nextNT $(r_2) \to \epsilon$. Note that we must have $\overline{m_{\text{call}}} \neq \emptyset$, since there is an unmatched call symbol generated by r_1. For the new state pair's context rule, we select a pair (r', r) in $\overline{m_{\text{call}}}$, where nextNT $(r) = L$, and restore the context to be r'.

With derivative functions defined, Algorithm 2 constructs the parser PDA by computing the least solution of the following equation; it makes sure that states are closed under derivatives.

$$\begin{aligned} A \quad = \quad & A \cup \{(\text{None}, L_0 \to .L_0)\} \cup \{m' \mid c \in \Sigma_{\text{plain}},\ m \in A,\ p_c(m) = (m', f)\} \\ & \cup \{m' \mid \langle a \in \Sigma_{\text{call}},\ m \in A,\ p_{\langle a}(m) = (m', f)\} \\ & \cup \{m' \mid b\rangle \in \Sigma_{\text{ret}},\ m \in A,\ m'' \in A \cup \{\emptyset\},\ p_{b\rangle}(m, m'') = (m', f)\} \end{aligned}$$

Definition 6.5 (Traces of Running the Parser PDA). Given a string w, we write w_i for its ith symbol. Suppose the sequence of transitions in a parser PDA for input w is $(m_0, \bot) \xrightarrow{w_1} (m_1, T_1) \cdots \xrightarrow{w_{|w|}} (m_{|w|}, T_{|w|})$. We define the trace of running the parser PDA for w, denoted as forest(w), as the list $[m_1, m_2, \ldots, m_{|w|}]$.

Since each m_i in forest(w) is a set of possible rules at that step of parsing, forest(w) can be viewed as the representation of all parse trees of input w; that is, forest(w) is the parse forest of w.

Finally, we define the acceptance configuration for the parser PDA as follows:

Definition 6.6 (Parser PDA Acceptance Configurations). Given a VPG $G = (V, \Sigma, P, L_0)$, a pair (m, E) is an *acceptance configuration* for the parser PDA if the state m includes a pair (None, r), where nextNT (r) derives the empty string ϵ, i.e., $\exists r,\ (\text{None}, r) \in m\ \wedge\ (\text{nextNT}\,(r) \rightarrow \epsilon) \in P$.

Note that the context None indicates that the stack E is empty. For general VPGs, however, the contexts in the state of an acceptance configuration could also be pending rules, indicating that the stack E is not empty, as mentioned in Definition C.2.

6.2 Extracting Parse Trees

In this section, we discuss how to extract parse trees for well-matched VPGs and put the discussion for general VPGs in Appendix C. First, we explain our design choice to accomplish efficient extraction, and then we define the extraction functions formally.

After running the parser PDA on input w, we have forest(w) = $[m_1, m_2, \ldots, m_{|w|}]$. Intuitively, for each parse tree $v = [r_1, \ldots, r_{|w|}]$ of w, we have $r_i \in m_i$, $i = 1..|w|$. Our next step is to extract the parse trees from forest(w). In Example 6.2, we showed the backward way for extraction, where we first extract valid rules from $m_{|w|}$ then extend them backward with the rules from $m_{|w|-1}$ to m_1. There is also a forward way, where we first extract valid rules from m_1 and then extend them with the rules extracted from m_2 to $m_{|w|}$. Although both ways would give the same set of parse trees, we choose the backward way for better performance. The forward extraction may maintain invalid parse trees during extraction; it may in the worst case maintain an exponential number of invalid parse trees and remove them in later steps, which is inefficient. The backward way, however, maintains only valid parse trees. For an unambiguous grammar in particular, the backward way always maintains one parse tree and guarantees linear-time extraction. We give a concrete example as follows:

Example 6.3. Consider the unambiguous grammar

$$L \rightarrow \langle aLb\rangle E \mid \langle aLd\rangle E$$
$$E \rightarrow \epsilon$$

and the input string $w = \langle a^n b\rangle^n$. The parse forest of w is "$m_1^n + m_2^{n-1} + [m_3]$," where the notation m_1^n stands for the list that contains n copies of m_1, and

$$m_1 = \{(L \rightarrow \langle a.Lb\rangle E, L \rightarrow \langle a.Lb\rangle E),\ (L \rightarrow \langle a.Ld\rangle E, L \rightarrow \langle a.Ld\rangle E)\},$$
$$m_2 = \{(L \rightarrow \langle a.Lb\rangle E, L \rightarrow \langle aLb\rangle .E)\},$$
$$m_3 = \{(\text{None},\ L \rightarrow \langle aLb\rangle .E)\}.$$

With the forward extraction, we first extract partial parse trees from m_1. There are two possible partial parse trees $[L \rightarrow \langle a.Lb\rangle E]$ and $[L \rightarrow \langle a.Ld\rangle E]$ (their context rules can be ignored for now); we have to extract both of them, although the second one will become invalid. We then extend each partial parse tree with rules from the next m_1; e.g., a new parse tree is $[L \rightarrow \langle a.Lb\rangle E,\ L \rightarrow \langle a.Lb\rangle E]$. This process is repeated, and we get 2^n partial parse trees for $\langle a^n$, since there are 2^n ways to parse $\langle a^n$. However, only one of them is valid considering the following input $b\rangle^n$; those invalid ones will be removed when considering $m_2^{n-1} + [m_3]$. The time complexity of this process is clearly non-linear.

In contrast, with the backward extraction, we first extract the only partial parse tree $[L \rightarrow \langle a.Lb\rangle E]$ from m_3, then extend it with the rule $L \rightarrow \langle aLb\rangle .E$ from m_2. We also introduce a stack

for each partial parse tree to store the rules of unmatched return symbols, so later we know how to extend the partial parse tree with matching rules. After extracting rules from ${m_2}^{n-1} + [m_3]$, we have only one partial parse tree whose stack is $(L \to \langle aLb \rangle . E)^n \cdot \bot$. Based on this stack and m_1, only rule $L \to \langle a . Lb \rangle E$ can be used for valid backward extension. In this way, only one valid parse tree is constructed in the end. We will prove that the backward extraction always generates valid parse trees in Lemma D.16.

Now, we define a *partial parse tree* as a parse tree paired with a stack as follows:

Definition 6.7 (Partial Parse Trees). A partial parse tree (v, E) is a parse tree $v = [r_1, \ldots, r_n]$ paired with a stack $E = r_{n_1} \cdots r_{n_k} \cdot \bot$, where $n_1 < \cdots < n_k$ and $r_{n_1}, \ldots, r_{n_k}$ are the rules that generate unmatched return symbols.

Formally, we use function $\text{extract}_{\text{init}}$ to extract a set of partial parse trees V from $m_{|w|}$ of forest(w) and function $\text{extract}_{\text{oneStep}}$ to extend the current set V of partial parse trees.

$$\begin{aligned} V_{|w|} &\leftarrow \text{extract}_{\text{init}}(m_{|w|}); \\ V_i &\leftarrow \text{extract}_{\text{oneStep}}(m_i, V_{i+1}), i \text{ goes from } |w| - 1 \text{ to } 1, \end{aligned}$$

where V_1 is the final parsing result that includes all valid parse trees of w.

Next, we define different kinds of states and define $\text{extract}_{\text{init}}$ and $\text{extract}_{\text{oneStep}}$.

Definition 6.8 ($\mathcal{M}_{pln}, \mathcal{M}_{call}, \mathcal{M}_{ret}$). Given a VPG $G = (\Sigma, V, P, L_0)$, based on the dotted rules in $\dot{P}$,

(1) we define a set of plain states $\mathcal{M}_{\text{pln}}$ as $(\mathcal{P}_{\text{call}} \cup \{\text{None}\}) \times \mathcal{P}_{\text{pln}}$;
(2) we define a set of call states $\mathcal{M}_{\text{call}}$ as $(\mathcal{P}_{\text{call}} \cup \{\text{None}\}) \times \mathcal{P}_{\text{call}}$;
(3) we define a set of return states $\mathcal{M}_{\text{ret}}$ as $(\mathcal{P}_{\text{call}} \cup \{\text{None}\}) \times \mathcal{P}_{\text{ret}}$.

Definition 6.9 (The Extraction Function $\text{extract}_{\text{init}}$). We first define $\text{extract}^{\text{pre}}_{\text{init}}$, which extracts all valid rules from m, then extend $\text{extract}^{\text{pre}}_{\text{init}}$ to $\text{extract}_{\text{init}}$, which pairs each extracted rule with a stack.

$$\text{extract}^{\text{pre}}_{\text{init}}(m) = \{r \mid (r', r) \in m, (\text{nextNT}\,(r) \to \epsilon) \in P \wedge r' \text{ is None}\}.$$

A pair (r', r) in m corresponds to a parse tree that ends with r and whose last unmatched call symbol is generated by r'. This parse tree is valid if nextNT (r) can match the empty string and r' is None. The context r' cannot be a matching rule, since a return symbol would be further needed.

$$\text{extract}_{\text{init}}(m) = \begin{cases} \left\{([r], \bot) \mid r \in \text{extract}^{\text{pre}}_{\text{init}}(m)\right\}, & \text{if } m \subseteq \mathcal{M}_{\text{pln}} \text{ or } m \subseteq \mathcal{M}_{\text{call}}; \\ \left\{([r], r \cdot \bot) \mid r \in \text{extract}^{\text{pre}}_{\text{init}}(m)\right\}, & \text{if } m \subseteq \mathcal{M}_{\text{ret}}. \end{cases}$$

If $m \subseteq \mathcal{M}_{\text{ret}}$, then we push r to the stack so in a later step during backward extraction, we can match the return symbol in this step using a matching rule. Note that from the definition of derivative functions (Definition 6.4), we can easily show that any state in the parser PDA is a subset of $\mathcal{M}_{\text{pln}}$, $\mathcal{M}_{\text{call}}$, or $\mathcal{M}_{\text{ret}}$; it cannot be the case that a parser PDA state have mixed pairs (e.g., some from $\mathcal{M}_{\text{call}}$ and others from $\mathcal{M}_{\text{ret}}$).

Definition 6.10 (The One-step Extraction Function). We first define extendable, which decides whether (r', r) can be used to extend v. Based on extendable, we define $\text{extract}^{\text{pre}}_{\text{oneStep}}$, which extracts rules from m and uses them to extend the partial parse trees in V. Finally, we define

$\text{extract}_{\text{oneStep}}$, which modifies the stacks of the extended trees:

$$\text{extendable}((r', r), (v, E)) = \begin{cases} \text{true}, & \text{if } r' \text{ is None } \wedge \text{ head}(E) \text{ is None } \wedge \\ & \quad \text{nextNT}(r) = \text{headNT}(v) \ \vee \\ & r' \text{ and head}(E) \text{ are the same matching rule} \\ & \quad \text{(but with different dot positions)} \wedge \\ & \quad\quad ((\text{firstRule}(v) \in \mathcal{P}_{\text{ret}} \ \wedge \ (\text{nextNT}(r) \to \epsilon) \in P) \vee \\ & \quad\quad\ (\text{firstRule}(v) \in \mathcal{P}_{\text{pln}} \cup \mathcal{P}_{\text{call}} \wedge \\ & \quad\quad\quad \text{nextNT}(r) = \text{headNT}(v))), \\ \text{false}, & \text{otherwise.} \end{cases}$$

$$\text{extract}^{\text{pre}}_{\text{oneStep}}(m, V) = \{(r :: v, E) \mid (v, E) \in V \ \wedge \ (r', r) \in m \ \wedge \ \text{extendable}((r', r), (v, E))\}.$$

Intuitively, each valid partial parse tree v can be split into v_1 and v_2 so $v = v_1 + v_2$ and $(r', \text{lastRule}(v_1))$ is included in m for some r' and (v_2, E) is included in V for some E. Therefore, if r' is None, then E is empty; if r' is a matching rule, then the top of E should be the same matching rule. In the first case, we require that r is connected to v, i.e., $\text{nextNT}(r) = \text{headNT}(v)$. In the second case, if $\text{firstRule}(v)$ is a rule that derives a return symbol, then we require $\text{nextNT}(r)$ can match the empty string; otherwise, we still require that r is connected to v.

$$\text{extract}_{\text{oneStep}}(m, V) = \begin{cases} \{(v, E) \mid (v, E) \in \text{extract}^{\text{pre}}_{\text{oneStep}}(m, V)\}, & \text{if } m \subseteq \mathcal{M}_{\text{pln}}, \\ \{(v, \text{tail}(E)) \mid (v, E) \in \text{extract}^{\text{pre}}_{\text{oneStep}}(m, V)\}, & \text{if } m \subseteq \mathcal{M}_{\text{call}}, \\ \{(v, \text{firstRule}(v) :: E) \mid (v, E) \in \text{extract}^{\text{pre}}_{\text{oneStep}}(m, V)\}, & \text{if } m \subseteq \mathcal{M}_{\text{ret}}. \end{cases}$$

$\text{extract}_{\text{oneStep}}$ modifies E based on the type of m. If a rule in m derives a call symbol, then the call symbol matches the return symbol (if exists) specified on the top of the stack and, therefore, we remove the top of the stack. If a rule in m derives a return symbol, then we push the rule for the return symbol to the stack so it can match a future call symbol during backward extraction.

Now, we define the extract function that extracts the set of parse trees of w from a forest.

Definition 6.11 (The Extraction Function). Given a nonempty $\text{forest}(w) = [m_1, \ldots, m_{|w|}]$, define the sequence of partial parse trees as

$$V_{|w|} = \text{extract}_{\text{init}}(m_{|w|});$$
$$V_i = \text{extract}_{\text{oneStep}}(m_i, V_{i+1}), i \text{ goes from } |w| - 1 \text{ to } 1.$$

Then, we define $\text{extract}(\text{forest}(w)) = V_1$ as the set of parse trees extracted from $\text{forest}(w)$.

6.3 Optimizing the Extraction as a PDA

For an unambiguous grammar, there is only one parse tree for any accepted input string. Even for an ambiguous grammar, there may be a preferred parse tree among possible parse trees for an input string. In this section, we optimize the parse-tree extraction process to an *extraction PDA*; instead of running the execution function, we can run the extraction PDA on the parse forest and get a trace that is a valid parse tree. Running a PDA is much faster, and our evaluation in Section 8 is based on the extraction PDA.

In Lemma D.17, Appendix D, we will show that those extraction functions always extract valid parse trees. Therefore, to extract a single parse tree, we need to pick one partial parse tree (v, E) from $\text{extract}_{\text{init}}(m_{|w|})$ according to some criterion; similarly, if $\text{extract}_{\text{oneStep}}(m, V)$ returns multiple parse trees, then one needs to be picked. To see why this process can be represented as a PDA, note

that the extraction function depends on only the first rule of the current partial parse tree v and the top of E; so we can view $(\text{firstRule}(v), E)$ as a PDA configuration. We define the PDA states and stacks as follows:

Definition 6.12 (Extraction PDA States and Stacks). Given a VPG, we introduce an extraction PDA whose states are dotted rules in $\dot{P}$ and whose stack contains rules in $\mathcal{P}_{\text{ret}}$. Intuitively, the stack remembers a series of contexts in terms of return rules, which are used to match call symbols during backward extraction. We call a pair (r, E) a *configuration*, with r being the state and E being the stack.

We define extraction PDA transition functions q_{init} and q based on extraction functions $\text{extract}_{\text{init}}$ and $\text{extract}_{\text{oneStep}}$. In the following definition, we write $\text{pickOne}(V)$ for a function that returns a rule from a list of partial parse trees V; if V is empty, then we define $\text{pickOne}(V)$ to be None. In practice, pickOne can be defined, e.g., based on priority of rules; we leave it abstract in this section.

Definition 6.13 (The Single-parse-tree Extraction Function q_{init}). Let $r = \text{pickOne}(\text{extract}^{\text{pre}}_{\text{init}}(m))$. If $r = \text{None}$, then we define $q_{\text{init}}(m) = \text{None}$. Otherwise, we define

$$q_{\text{init}}(m) = \begin{cases} (r, \bot), & \text{if } m \subseteq \mathcal{M}_{\text{pln}} \text{ or } m \subseteq \mathcal{M}_{\text{call}}; \\ (r, r \cdot \bot) & \text{if } m \subseteq \mathcal{M}_{\text{ret}}. \end{cases}$$

Definition 6.14 (The Transition Function q for the Extraction PDA). Given a current configuration (r, E), let $t = \text{head}\, E$ and $r' = \text{pickOne}(\text{extract}_{\text{oneStep}}(m, \{([r], t \cdot \bot)\})$. If $r' = \text{None}$, then we define $q(m, r, t) = \text{None}$. Otherwise, we define

$$q(m, r, t) = \begin{cases} (r', \lambda E.E), & \text{if } m \subseteq \mathcal{M}_{\text{pln}}; \\ (r', \text{tail}), & \text{if } m \subseteq \mathcal{M}_{\text{call}}; \\ (r', \lambda E.r' \cdot E), & \text{if } m \subseteq \mathcal{M}_{\text{ret}}. \end{cases}$$

Algorithm 3 constructs the PDA by computing the least solution of the following equation; it makes sure that states are closed under derivatives.

$$\begin{aligned} R \quad = \quad & R \cup \{r' \mid m \in A \wedge (r', t) = q_{\text{init}}(m)\} \\ & \cup \{r' \mid m \in A, r \in R \cap (\mathcal{P}_{\text{pln}} \cup \mathcal{P}_{\text{call}}), t \in (R \cap \mathcal{P}_{\text{ret}}) \cup \{\text{None}\},\ q(m, r, t) = (r', f)\} \\ & \cup \{r' \mid m \in A, r \in R \cap \mathcal{P}_{\text{ret}},\ q(m, r, r) = (r', f)\}, \end{aligned}$$

where A is the set of all parser PDA states. Note that we do not need to compute the acceptance state for an extraction PDA: For a forest with the last state m, as long as the initial state $\text{extract}_{\text{init}}(m)$ is not empty, the extraction is guaranteed to be successful; if the initial state $\text{extract}_{\text{init}}(m)$ is empty, then the input string is invalid.

6.4 Correctness Proof

In this subsection, we prove the correctness of the parse trees constructed according to Sections 6.1 and 6.2. The correctness of the optimization in Section 6.3 using an extraction PDA follows as a corollary, since the extraction PDA extracts one parse tree out of possibly many parse trees returned by the extraction functions in Section 6.2.

Given a VPG $G = (V, \Sigma, P, L_0)$ and an input string w, let $V = \text{extract}(\text{forest}(w))$. We call V correct if it includes exactly the set of valid parse trees of w according to G, which we state formally as the following theorem:

Theorem 6.1 (Correctness of the Parser Generator).

$$\forall w\ V,\ V = \text{extract}(\text{forest}(w)) \implies (\forall v,\ L_0 \Downarrow (w, v) \iff \exists E, (v, E) \in V).$$

ALGORITHM 3: Constructing the extraction PDA.

Input : A VPG $G = (V, \Sigma, P, L_0)$ where $\Sigma = \Sigma_{\text{call}} \cup \Sigma_{\text{plain}} \cup \Sigma_{\text{ret}}$, A generated from Algorithm 2, q_{init} and q;

Return : The set of all produced states R and the set of transitions $\mathcal{T}$;

Initialize the set for new states;

$R \leftarrow \{r' \mid m \in A \wedge (r', t) = q_{\text{init}}(m)\}$;

Initialize the set for transitions $\mathcal{T} \leftarrow \{\}$;

repeat

  Compute transitions for call and plain states

  $R' \leftarrow \{(m, t, f, r, r') \mid m \in A, r \in R \cap (\mathcal{P}_{\text{pln}} \cup \mathcal{P}_{\text{call}}), t \in (R \cap \mathcal{P}_{\text{ret}}) \cup \{\text{None}\},\ q(m, r, t) = (r', f)\}$;

  Compute transitions for return states

  $R' \leftarrow R' \cup \{(m, t, f, r, r') \mid m \in A, r \in (R \cap \mathcal{P}_{\text{ret}}),\ q(m, r, r) = (r', f)\}$;

  Add edge (r, r') marked with (m, t, f) to $\mathcal{T}$, where $(m, t, f, r, r') \in R'$;

  Compute new states $N \leftarrow \{r' \mid (\_, \_, \_, \_, r') \in R'\} - R$;

  Update the set of all produced states $R \leftarrow R \cup N$;

until $N = \emptyset$;

We provide detailed lemmas and proofs in Appendix D and describe the main steps next. Remember that the parser PDA builds a set of partial parse trees in a forward way and stores the trees in a forest, from which the extraction functions extract a set of partial parse trees in a backward way. Therefore, for the correctness proof, we formalize two small-step relations: one for building *forward* parse trees and one for building *backward* parse trees. We then show that both small-step relations are equivalent to the big-step parse-tree derivation relation under certain conditions. For the backward direction of Theorem 6.1 (i.e., soundness), we prove that during backward extraction for an input string w, each extracted backward partial parse tree v_2 has a counterpart forward partial parse tree v_1 in the rest of the forest, and $v_1 + v_2$ is a valid parse tree of w. At the end of extraction, we have $v_1 = []$ and v_2 is a valid parse tree of w. For the forward direction of Theorem 6.1 (i.e., completeness), we show that each valid parse tree v can be split as $v = v_1 + v_2$, and there is an extraction step where v_1 is in the rest of the forest and v_2 is an extracted partial parse tree. Therefore, at the end of extraction, we have $v_1 = []$ and $v = v_2$ is in the extracted parse trees.

6.5 Verification of Time Complexity

Next, we discuss how we verify the performance of our VPG parser. For an input string w with a total number of k valid parse trees, we show that the VPG parser takes $O(k|w|)$ time. When $k = 1$, the time complexity becomes linear (i.e., $O(|w|)$). The verification is performed in two steps: (1) it takes $O(|w|)$ time to run a parser PDA and generates a parse forest; and (2) it takes $O(k|w|)$ time to run the extraction function on the forest when extracting all of the k parse trees.

We first introduce our method of counting time cost in Coq. Our verification utilizes a Coq monadic library [29], which provides a monad for tracking the time cost of computation. To pair a value a of type A with a time cost c, we utilize a dependent type $C(A, P)$, defined to be

$$C(A, P) = \{a : A \mid \exists c \in \mathbb{N}, P\ a\ c\}\,,$$

where C is a type constructor and $P\ a\ c$ is a property of a and c. In this article, the above dependent type is written using the following notation:

$$\{a : A \mid c \mid P\ a\ c\}.$$

For example, given $k \in \mathbb{N}$, a value a' of type $\{a : A \mid c \mid c = k\}$ means it takes k time units to compute a of type A.

To correctly count the time cost, programs must be written as a combination of the return and bind monadic operators provided by the library [29]. We briefly introduce them as follows:

(1) The return operator, denoted as "$\Leftarrow a$", returns a value a bound with 0 time cost.
(2) The insertion operator, denoted as "$+= k;\ a'$", adds k time units to the monadic value a'.
(3) The bind operator, denoted as "$a \leftarrow e_1;\ e_2$", first evaluates the computation e_1 and binds its value to a, which is used to evaluate e_2; it further pairs the result with the sum of the time cost of e_1 and e_2.

In our verification, the time cost of a function is counted by accumulating the time cost of its components, except for three kinds of primitive functions with unit cost:

(1) `eq_sym_ID`, which compares the IDs of two symbols (nonterminals or terminals).
(2) `andb` and `orb`, which compute the conjunction and disjunction of two Booleans, respectively.
(3) `cons a l`, which constructs a list from head a and tail l, and `pair a b`, which makes a pair.

The insertion function we introduced earlier is used in only two ways: (1) add the time cost `cost_eq_sym_ID`, `cost_andb`, `cost_orb`, `cost_cons`, or `cost_pair` to a corresponding function. E.g., the program

$$+= \text{cost\_eq\_sym\_ID};\ \Leftarrow \text{eq\_sym\_ID}\ s_1\ s_2;$$

compares two IDs s_1 and s_2, and returns a bool paired with `cost_eq_sym_ID`; and then (2) add `cost_branch`, which is the number of variables and functions used in a branch, similar to the counting method of McCarthy et al. [29]. Consider the following example:

$$\begin{aligned}&\lambda x.\ \text{match}\ x\ \text{with}\\ &\quad \mid [] \Rightarrow += 1;\ \Leftarrow \text{None}\\ &\quad \mid a :: l \Rightarrow += 3;\ \Leftarrow \text{Some}\ a.\end{aligned}$$

In the above example, only one unit of cost is added, because only one name x is used; three units of cost are added to the second branch, because three names (x, a, and l) are used. These branch costs are inserted manually.

With the aforementioned setup, we implemented two functions runPDA′ and extract′, where runPDA′ runs the parser PDA to generate a forest and extract′ extracts the set of parse trees from the forest. In particular, runPDA′ has the following type:

$$\text{runPDA}' : \forall \mathcal{T}\ w, \{M : \text{Forest} \mid c \mid c = \text{cost}_{\text{runPDA}}(\mathcal{T}, w)\}$$

where w is a string, Forest is the type of parse forests, and $\mathcal{T}$, representing the PDA, is a list of mappings in the form of $((i, S, t), S')$. A mapping $((i, S, t), S')$ represents the state transition from configuration (S, T) to (S', T'), where $t = \text{head}(T)$; the mapping from T to T' is further computed in runPDA′. We show the upper bound of the cost function $\text{cost}_{\text{runPDA}}$ in Theorem 6.2.

Theorem 6.2 (The Running-time Upper Bound of the Parser PDA).

$$\forall \mathcal{T},\ \exists b_1\ b_2,\ \forall w,\ \text{cost}_{\text{runPDA}}(\mathcal{T}, w) \le b_1|w| + b_2.$$

Theorem 6.2 shows that it takes $O(|w|)$ time to run the parser PDA and generates a parse forest. The proof is straightforward and we omit its discussion.

Now, we turn our attention to the extract function. We previously introduced its definition in Definition 6.11, which maintains a *set* of partial parse trees. However, it is not efficient to maintain a set-based data structure. Inserting a partial parse tree v to a set, for example, requires us to compare v with trees already in the set to avoid duplicates; comparing each of the $|w|$ rules in the

parse trees would take at least $|w|$ steps. Even if we hash parse trees before comparison, we would still have to perform hashing for each of the $|w|$ rules. Therefore, if a partial parse tree is inserted at each of the $|w|$ steps, then the total time cost is at least quadratic ($\Omega(|w|^2)$). Getting a linear-time algorithm is a challenge.

We next describe a linear-time algorithm that takes advantage of the locality of VPG parsing. We replace the set structure with a *list* structure. But we also have to avoid duplicate partial parse trees in the list. To see why this is a problem, notice that two different elements (r_1, r) and (r_2, r) in a state m can be used to extend the same parse tree v from a list V of partial parse trees, both generating the same tree $r :: v$.

To solve this problem, our observation is that if V includes no duplicates, then for two trees $v_1, v_2 \in V$, we have not only $v_1 \neq v_2$, but also $r_1 :: v_1 \neq r_2 :: v_2$ for $r_1, r_2 \in \dot{P}$. Therefore, for each $v \in V$, we can first compute m_v, the rules in m that can extend v, and remove the duplicates in m_v to get m'_v; we then extend v with m'_v to get V_v and finally concatenate V_v for all $v \in V$. To avoid duplicates in m_v, we compare elements in m_v, which takes $O(|m|^2)$ time. Since m belongs to $(P \cup \{\text{None}\}) \times P$, its size $|m|$ is at most $(|P| + 1)|P|$; therefore, $O(|m|^2)$ is independent of the input string size and can be viewed as a constant.

Definitions 6.15, 6.16, and 6.17 implement this optimized version of the extract function. It utilizes function rmDup, which removes the duplicates in a list, function list, which converts a set to a list, and standard functions map, filter, and concat. We omit their standard definitions.

Definition 6.15 (The Optimized Implementation $\text{extract}^{\text{op}}_{\text{init}}$*).*

$$\text{extract}^{\text{op}}_{\text{init}}(m) = \text{rmDup}(\text{list}(\text{extract}_{\text{init}}(m))).$$

Definition 6.16 (The Optimized Implementation $\text{extract}^{\text{op}}_{\text{oneStep}}$*).* Given a VPG $G = (\Sigma, V, P, L_0)$, a set of partial parse trees V and a state m, we first define functions $\text{extract}^{\text{filt}}$, $\text{extract}^{\text{map}}$, and $\text{extract}^{\text{concat}}(m, V)$, and then define $\text{extract}^{\text{op}}_{\text{oneStep}}$ based on them.

$$\text{extract}^{\text{filt}}(m, (v, E)) = \text{rmDup}\ (\text{map}\ (\lambda(x, y).y, (\text{list}\ \ (\text{filter}\ (\lambda x.\ \text{extendable}(x, (v, E)), m)))))$$

$$\text{extract}^{\text{map}}(m, (v, E)) = \text{map}\ (\lambda x.(x :: v, E),\ \text{extract}^{\text{filt}}(m, (v, E)))$$

$$\text{extract}^{\text{concat}}(m, V) = \text{concat}\ (\text{map}\ (\lambda(v, E).\ \text{extract}^{\text{map}}(m, (v, E)), V))$$

$$\text{extract}^{\text{op}}_{\text{oneStep}}(m, V) = \begin{cases} [(v, E) \mid (v, E) \in \text{extract}^{\text{concat}}(m, V)], & \text{if } m \subseteq \mathcal{M}_{\text{pln}}, \\ [(v, \text{tail}(E)) \mid (v, E) \in \text{extract}^{\text{concat}}(m, V)], & \text{if } m \subseteq \mathcal{M}_{\text{call}}, \\ [(v, \text{firstRule}(v) :: E) \mid (v, E) \in \text{extract}^{\text{concat}}(m, V)], & \text{if } m \subseteq \mathcal{M}_{\text{ret}}. \end{cases}$$

Definition 6.17 (The Optimized Implementation $\text{extract}^{\text{op}}$*).* Given a $\text{forest}(w) = [m_1, \ldots, m_{|w|}]$, where $w \neq \epsilon$, define the sequence of partial parse trees as

$$V_{|w|} = \text{extract}^{\text{op}}_{\text{init}}(m_{|w|});$$
$$V_i = \text{extract}^{\text{op}}_{\text{oneStep}}(m_i, V_{i+1}),\ i \text{ goes from } |w| - 1 \text{ to } 1.$$

Then, we define $\text{extract}^{\text{op}}(\text{forest}(w)) = V_1$ as the list of parse trees extracted from $\text{forest}(w)$.

Based on $\text{extract}^{\text{op}}$, We implemented a version of the extract function that performs cost counting:

$$\text{extract}' : \forall w,\ \{V : \text{ParseTreeList} \mid c \mid V = \text{extract}^{\text{op}}(\text{forest}(w)) \ \wedge\ c = \text{cost}_{\text{extract}}(\text{forest}(w))\}$$

We show the upper bound of the cost function of extract′ in Theorem 6.3, which also includes the correctness discussed in Section 6.4.

Theorem 6.3 (The Running-time Upper Bound of Extraction). *Given a VPG* $G = (\Sigma, V, P, L_0)$*, we have*

$$\begin{aligned}
&\exists k\ b_1\ b_2,\ \forall w\ V_1,\\
&\quad w \neq \epsilon\ \wedge\ V_1 = \text{extract}^{\text{op}}(\text{forest}(w))\ \Longrightarrow\\
&\qquad \text{NoDup}(V_1)\ \wedge\\
&\qquad \text{cost}_{\text{extract}}(\text{forest}(w)) \leq (k * |V_1| + b_1) * |w| + b_2\ \wedge\\
&\qquad \forall v,\ (\exists E, (v, E) \in V_1)\ \Longleftrightarrow\ L_0 \Downarrow (w, v)\,.
\end{aligned}$$

There are three properties in Theorem 6.3. The first property "NoDup(V_1)" means there is no duplicate in V_1; therefore, $|V_1|$ is the number of valid parse trees. The next property shows that the time complexity of extract′ is $O(|V_1||w|)$. The third property shows the correctness, i.e., V_1 is the set of parse trees of w.

Proof. During the extraction, we maintain lists $V_i, i = 1..|w|$ of partial parse trees:

$$\begin{aligned}
V_{|w|} &\leftarrow \text{extract}^{\text{op}}_{\text{init}}(m);\\
V_i &\leftarrow \text{extract}^{\text{op}}_{\text{oneStep}}(m_i, V_{i+1}), i \text{ goes from } |w| - 1 \text{ to } 1.
\end{aligned}$$

We will prove in Theorem 6.4 that V_i has no duplicates for $i = 1..|w|$. When $i = 1$, we get the first property. To prove the second property, we show that the size of V_i increases during backward extraction in Theorem 6.5, i.e.,

$$|V_{|w|}| \leq |V_{|w|-1}| \leq \cdots \leq |V_1|.$$

Then, we prove that during extraction, for each (m, V), it takes $O(|m|^2|V|)$ time to evaluate $\text{extract}^{\text{op}}_{\text{oneStep}}(m, V)$. To do this, we implement the monadic function

$$\text{extract}^{\text{op}}_{\text{oneStep}} : \forall m\ V,\ \{V' : \text{ParseTreeList} \mid c \mid c = \text{cost}^{\text{Extract}}_{\text{oneStep}}(m, V)\}$$

and prove in Theorem 6.6 that $\text{cost}^{\text{Extract}}_{\text{oneStep}}(m, V) \sim O(|m|^2|V|)$. We then rewrite this bound to $O(|V|)$, since $|m| \sim O(|P|^2)$. Finally, since $\text{extract}^{\text{op}}$ can be viewed as executing $\text{extract}^{\text{op}}_{\text{oneStep}}$ for $|w|$ times, we have the time complexity of $\sum_{i=1}^{|w|} O(|V_i|)$, which is further bounded by $O(|V_1||w|)$ using Theorem 6.5. □

Theorem 6.4 (No Duplicates in V_i).

$$\begin{aligned}
&\forall w\ i\ m\ M,\ \text{forest}\,(wi) = M + [m]\ \Longrightarrow\\
&\forall M_1\ M_2,\ M = M_2 + M_1\ \Longrightarrow\\
&\quad \text{NoDup}(\text{extract}^{\text{op}}(M_1 + [m])).
\end{aligned}$$

Proof. Let $V_{\text{init}} = \text{extract}^{\text{op}}_{\text{init}}(m)$. Prove by induction on M_1. There are two cases.

When $M_1 = []$, we have $\text{extract}^{\text{op}}(M_1 + [m]) = V_{\text{init}}$. Since we apply rmDup to get V_{init}, clearly, we have $\text{NoDup}(V_{\text{init}})$.

When $M_1 = [m_1] + M_1'$ for some m_1, we have $\text{extract}^{\text{op}}(M_1 + [m]) = \text{extract}^{\text{op}}_{\text{oneStep}}(m_1, V')$, where $V' = \text{extract}^{\text{op}}(M_1' + [m])$. By the induction hypothesis, we have $\text{NoDup}(V')$. Therefore, from the definition of $\text{extract}^{\text{op}}_{\text{oneStep}}$, we have $\text{NoDup}(\text{extract}^{\text{op}}(M_1 + [m]))$. □

Theorem 6.5 (Increasing $|V_i|$).

$$\forall w\ i\ m\ M,\ \text{forest}(wi) = M + [m] \implies \forall M_1\ m'\ M_2,\ M = M_2 + [m'] + M_1 \implies |\text{extract}^{\text{op}}(M_1 + [m])| \leq |\text{extract}^{\text{op}}([m'] + M_1 + [m])|.$$

Proof. From the invariant in Lemma D.17 we know that each $v \in \text{extract}^{\text{op}}(M_1 + [m])$ is a valid partial parse tree; i.e., $\exists v_1, v_1 + v$ is a parse tree of w. Let $v_1 = v_1' + [r]$ for some r; since $[r] + v$ is also a valid partial parse tree, we have $[r] + v \in \text{extract}^{\text{op}}([m'] + M_1 + [m])$. This shows an injection from $\text{extract}^{\text{op}}(M_1 + [m])$ to $\text{extract}^{\text{op}}([m'] + M_1 + [m])$. Therefore, we have $|\text{extract}^{\text{op}}(M_1 + [m])| \leq |\text{extract}^{\text{op}}([m'] + M_1 + [m])|$. □

Theorem 6.6 (The Running-Time Upper Bound of $\text{extract}^{\text{op}}_{\text{oneStep}}$). $\exists k_1\ k_2\ b_1\ b_2,\ \forall V\ m,$

$$\text{cost}^{\text{Extract}}_{\text{oneStep}}(m, V) \leq (k_1|m|^2 + k_2|m| + b_1) \cdot |V| + b_2.$$

Proof. For each v in V, it takes $O(|m|^2)$ time to remove the duplicates in a list of rules, therefore, the total time is $O(|m|^2|V|)$. □

Note that one benefit of the monadic library is that, since the running times are embedded in propositions, we can extract OCaml code from the Coq implementation and after extraction propositions with running times removed.

6.6 Coq Mechanization

All definitions and theorems presented in Sections 6.4 and 6.5 are mechanized in the proof assistant Coq. Our proof artifact consists of 19 Coq files and ~43k lines of code. The complete verification took ~ 5 minutes on an Intel® Core™ i7-9700 Processor and 16 GB of memory.

Our Coq implementation includes two hypotheses, namely, `A_VPG_Linear` and `A_VPG_Match`, corresponding to the two constraints for general VPGs we discussed in Section 2: `A_VPG_Linear` requires that for each rule $L \to iL'$ where $i \in \Sigma$, if the nonterminal L belongs to V^0, then the nonterminal L' must also belong to V^0; and `A_VPG_Match` requires that for each matching rule $L \to \langle aL_1 b \rangle L_2$, the nonterminal L_1 belongs to V^0, and if the nonterminal L belongs to V^0, then so does the nonterminal L_2. These two hypotheses guarantee that our Coq implementation accepts valid general VPGs. In Appendix E, we summarize the correspondence between theorems discussed in this section and our open-sourced Coq formalization and proofs.

We discuss some key differences between the definitions as well as theorems presented in Sections 6.4 and 6.5 and their Coq implementation as follows: First, the big-step parse-tree derivation in Figure 4 includes three rules; its Coq mechanization includes two additional rules for deriving pending call and return symbols, respectively. Second, we formalized the semantics of running the parser PDA in Coq as the relation "`Forest` $M\ T\ w$," where M is the trace for the string w, and T is the terminal stack. Recall that in Definition 6.5, we denote the trace as forest(w). In Coq, the function forest is verified to generate a trace that satisfies the relation `Forest`. Further, in the theorems for extraction PDAs, such as Theorems 6.3 and 6.4, the trace forest(w) is replaced with a trace M that satisfies the `Forest` relation.

7 DESIGNING A SURFACE GRAMMAR

The format of rules allowed in VPGs is designed for easy studying of its meta-theory but is inconvenient for expressing practical grammars. First, no user-defined semantic actions are allowed. Second, each VPG rule allows at most four terminals/nonterminals on the right-hand side. In this section, we present a surface grammar that is more user-friendly for writing grammars. We first

discuss embedding semantic actions. Then, we introduce *tagged CFGs*, which are CFGs paired with information about how to separate terminals to plain, call, and return symbols. We then describe a translator from tagged CFGs to VPGs. During the conversion, the translator also generates semantic actions that convert the parse trees of VPGs back to the ones of tagged CFGs.

7.1 Embedding Semantic Actions

Semantic actions transform parsing results to user-preferred formats. In a rule $L \to s_1 \cdots s_k$, where $s_k \in \Sigma \cup V$, we treat L as a default action that takes k arguments, which are semantic values returned by s_1 to s_k, and returns a tree with a root node and s_1 to s_k as children. The prefix notation of a parse tree gives

$$[L, v_{s_1}, \ldots, v_{s_k}],$$

where v_{s_i} is the semantic value for s_i. The above notation can be naturally viewed as a stack machine, where L is an action and v_{s_i} are the values that get pushed to the stack before the action. The VPG parse tree can be converted to the prefix notation in a straightforward way. If we then replace each nonterminal in the tree with its semantic action, then the parse tree becomes a stack machine.

The default action for a nonterminal can be replaced by a user-defined action appended to each rule in the grammar. For example, consider the grammar $L = cL \mid \langle aLb \rangle L \mid \epsilon$. Suppose we want to count the number of the symbol c in an input string; we can specify semantic actions in the grammar as follows:

$$\begin{aligned} L \to\ & cL \quad @\{\text{let } f_1\ v_1\ v_2 = 1 + v_2\} \\ \mid\ & \langle aLb \rangle L \quad @\{\text{let } f_2\ v_1\ v_2\ v_3\ v_4 = v_2 + v_4\} \\ \mid\ & \epsilon \quad @\{\text{let } f_3\ () = 0\}. \end{aligned}$$

In the above example, a semantic action is specified after each rule, e.g., "@{let $f_1\ v_1\ v_2 = 1 + v_2$}." In the actions, v_1, v_2, v_3, and v_4 represent the semantic values returned by the right-hand side symbols of the rule. For example, the first semantic action $f_1\ v_1\ v_2 = 1 + v_2$ accepts two semantic values v_1 and v_2, where v_1 is returned by c and v_2 is returned by L. Specially, the definition "@{let $f_3\ () = 0$}" defines a function that does not take any argument.

As an application, the next subsection shows how to use semantic actions to convert the parse trees of a VPG to the parse trees of its original tagged CFG.

7.2 Translating from Tagged CFGs to VPGs

Grammar writers are already familiar with CFGs, the basis of many parsing libraries. We define *tagged CFGs* to be CFGs paired with information about how to partition terminals into plain, call, and return symbols ($\Sigma = \Sigma_{\text{plain}} \cup \Sigma_{\text{call}} \cup \Sigma_{\text{ret}}$)[3]; that is, in a tagged CFG, a terminal is tagged with information about what kind of symbols it is. Compared to a regular CFG, the only additional information in a tagged CFG is the tagging information; therefore, tagged CFGs provide a convenient mechanism for reusing existing CFGs and developing new grammars in a mechanism that grammar writers are familiar with.

Not all tagged CFGs can be converted to VPGs. We use a conservative validator to determine if a tagged CFG can be converted to a VPG and, if the validator passes, translate the tagged CFG to a VPG. Without loss of generality, we assume that in each rule of the input tagged CFG a call symbol is matched with a return symbol. In the general case, we need two additional steps: (1) We first

[3]We note that our implementation of tagged CFGs additionally supports regular operators in the rules; these regular operators can be easily desugared and we omit their discussion.

Tagged CFG ⇒	Simple Form ⇒	Linear Form ⇒	VPG
$L \to A‹aAEb›E\ @L^6$	$L \to A‹aL_{AE}b›E\ @L^6$	$L \to c‹aL_{AE}b›E\ @L^6 \circ A^1$	$L \to cL_1\ @L^6 \circ A^1$
$A \to cE\ @A^2$	$L_{AE} \to AE$	$L_{AE} \to cE\ @A^1$	$L_1 \to ‹aL_{AE}b›E$
$E \to \epsilon\ @E^0$	$A \to cE\ @A^2$	$A \to cE\ @A^2$	$L_{AE} \to cE\ @A^1$
	$E \to \epsilon\ @E^0$	$E \to \epsilon\ @E^0$	$A \to cE\ @A^2$
			$E \to \epsilon\ @E^0$

Fig. 7. An example of converting a tagged CFG to a VPG.

assure that pending call and return symbols are not nested in well-matched call/return symbols; (2) we then change the tags of the pending call and return symbols to plain, making the resulting grammar ready for the validator.

The translation steps are summarized as follows:

$$\textit{A tagged CFG} \to \textit{Simple form} \xrightarrow{\textit{If valid}} \textit{Linear form} \to \textit{VPG}.$$

At a high level, a tagged CFG is first translated to a simple form, upon which validation is performed. If validation passes, then the simple-form CFG is translated to a linear-form CFG, which is finally translated to a VPG. We next detail these steps. These steps will be illustrated with a running example in Figure 7. In the example, the left-most column lists the original tagged CFG; note that in the grammar semantic actions are after the @ symbol (e.g., L^6) and they will be discussed later in the section.

Definition 7.1 (Simple Forms). A rule is in the simple form if it is of the form $L \to \epsilon$ or of the form $L \to q_1 \cdots q_k$, where either $q_i \in \Sigma_{\text{plain}} \cup V$ or $q_i = ‹aL_ib›$ for some $‹a, b›, L'$, where $i = 1..k$ and $k \geq 1$, and L' is a nonterminal. A tagged CFG $G = (V, \Sigma, P, L_0)$ is in the simple form if every rule in P is in the simple form.

Compared to a tagged CFG, a simple-form CFG requires that there must be a nonterminal between a call symbol and its matching return symbol. The conversion from a tagged CFG to a simple-form CFG is straightforward: For each rule, we replace every string $‹asb›$, where $‹a$ is matched with $b›$ and $s \in (\Sigma \cup V)^*$, with $‹aL_sb›$ and generate a new nonterminal L_s and a new rule $L_s \to s$. After this conversion, a string in the form of $‹aLb›$ can be viewed as a "plain symbol"; this is a key intuition for the following steps. We call $‹aLb›$ a *matched token* in the following discussion. For the running example, this step of translation extracts AE from the first rule and assigns it to a new nonterminal L_{AE}.

The validation can then be performed on the simple form, using its dependency graph.

Definition 7.2 (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 7.3 (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).

The validator's main task is to make sure each dependency loop is well founded. Additionally, for non-dependency loops, the validator ensures they are not solely labeled with (ϵ, ϵ) to prevent

dead loops. Specifically, the validator verifies 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 $L \to s_1 \langle aL'b \rangle s_2$; or (2) every edge (L, L') in the loop is labeled with (s, ϵ) for some string $s \in (\Sigma \cup V)^*$, and at least one such s satisfies $s \not\to^* \epsilon$. For the running example, the dependency graph of the simple form of the grammar does not have any loops and thus validation passes trivially.

Once the validation passes, the translation converts a simple-form CFG to a linear-form CFG.

Definition 7.4 (Linear Forms). A rule is in the linear form if it is in one of the following forms: (1) $L \to \epsilon$; (2) $L \to t_1 \cdots t_k$; (3) $L \to t_1 \cdots t_k L'$; where $t_i \in \Sigma_{\text{plain}}$ or $\exists \langle a, b \rangle, L_i$, s.t. $t_i = \langle aL_i b \rangle$, $i = 1..k$, $k \geq 1$. A tagged CFG $G = (V, \Sigma, P, L_0)$ is in the linear form if every rule in P is in the linear form.

Note that in a linear-form rule, t_i cannot be a nonterminal, while in a simple-form rule q_i can be a nonterminal. Further, the linear form allows rules of the form $L \to t_1 \cdots t_k L'$, where t_i is a terminal or a matched token. The main job of the translator is to convert simple-form rules to linear-form rules by strategically replacing nonterminals with the right-hand sides of their rules. Appendix G shows the translation algorithm and its termination proof. Compared to the conference version where no termination proof was provided, this version of the translation algorithm has a new reformulation, which enables the termination proof to go through. Figure 7 shows the conversion result of the running example. For this example, the conversion is simple: Nonterminal A is replaced by cE and eventually replaced by c in the first two rules.

The translation from a linear-form CFG to a VPG is simple. E.g., for a rule of the form $L \to t_1 \cdots t_k$, it is translated to $L \to t_1 L_1; L_1 \to t_2 L_2; \ldots; L_k \to t_k L_k; L_k \to \epsilon$, where L_1 to L_k are a set of new nonterminals. For the running example, the first rule in the linear form of the grammar is split into two rules, as shown in the last column in Figure 7.

All transformations are local rewriting of rules, and as a result it is easy to show that each transformation step preserves the set of strings the grammar accepts; further, for an input string there should be a one-to-one correspondence between parse trees produced by the grammar before transformation and parse trees produced by the grammar after transformation. We further note that not all tagged CFGs can be converted to VPGs. For example, grammar "$L \to cLc|\epsilon$" cannot be converted, since its terminals cannot be suitably tagged: Intuitively, c has to be both a call and a return symbol. Further, since our validation algorithm is conservative, it rejects some tagged CFGs that have VPG counterparts. For example, grammar "$L \to Lc|\epsilon$" is rejected by the validator, since it is left recursive. However, it can be first refactored to "$L \to cL|\epsilon$", which is accepted by our validator.

Generating semantic actions. During the conversion, each time the translator rewrites a rule, a corresponding semantic action is attached to the rule. Initially, every rule is attached with one default semantic action. For example, the rule $L \to AbCd$ is attached with L^4, written as $L \to AbCd\ @L^4$. As mentioned in Section 7.1, L^4 is the default semantic action for constructing a tree with a root node and children nodes that are constructed from semantic values from the right-hand side of the rule. The superscript 4 is its arity. During conversion, every time we rewrite a nonterminal L in a rule R with the right-hand side of rule $L \to s$, the semantic values for s are first combined to produce a semantic value for L, which is then used to produce the semantic value for the left-hand nonterminal of R. If a helper nonterminal L_s is introduced during conversion and a rule $L_s \to s$ is generated, then we do not generate a semantic value for L_s but leave the semantic values for s on the stack so any rule that uses L_s can use those semantic values directly. In this way, we can convert a parse tree of a VPG to the parse tree of its corresponding tagged CFG.

Back to the example in Figure 7, during the conversion to the simple form, the semantic action of the first rule does not change, since the rule for L_{AE} has no semantic actions; the semantic values of A and E are left on the stack. Thus, L^6 still expects six values on the stack. The translation step to the linear form expands A in the first rule with $A \to cE@A^2$. Then, $cE‹aL_{AE}b›E$ is simplified to $c‹aL_{AE}b›E$, and that is why A^1 is applied instead of A^2: A^1 accepts the value for c. The same transformation is applied to the second rule. The translation step to the VPG is straightforward.

As a concrete example of parsing using the VPG in Figure 7, for the input string $c‹acb›$, the parser will generate the following parse tree:

$$[L \to cL_1;\ L_1 \to ‹a.L_{AE}b›E;\ L_{AE} \to cE;\ L_1 \to ‹aL_{AE}b›.E].$$

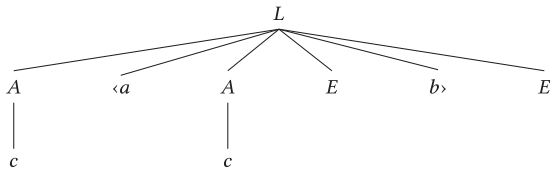
Each rule in the above parse tree is then replaced with its attached action and the semantic values of the terminals.

$$[L^6 \circ A^1, c, ‹a, A^1, c, E^0, b›, E^0]$$

And the evaluation result of the above stack machine is the following parse tree of the tagged CFG:

$$[(L, [(A, [c]), ‹a, (A, [c]), E, b›, E])].$$

The above tree can be visualized as follows:



8 EVALUATION

As discussed earlier, we implemented in Coq our VPG parsing library and mechanized its proofs; we also introduced two optimizations so parsing is performed by running two PDAs (the parser and the extraction PDAs). To evaluate performance, we extracted OCaml code from the Coq library and used it to construct the parser and extraction PDAs by Algorithms 2 and 3 and stored the PDA transitions in hashtables for constant-time look-ups. We evaluated our implementation for the following questions: (1) how applicable VPG parsing is in practice? (2) what is the performance of VPG parsing compared with other parsing approaches?

We performed a preliminary analysis for a set of ANTLR4 grammars in a grammar repository [3]. Among all 239 grammars, 136 (56.9%) grammars could be converted to VPGs by our tagged-CFG-to-VPG translation after we manually marked the call and return symbols for those grammars. Note that it does not mean the rest cannot be converted; e.g., 34 grammars cannot be converted, because they have left recursion and the conversion may become possible if the left recursion is removed. We left a further analysis for future work.

For performance evaluation, we compared our VPG parsers with two representative LL and LR parsers, namely, ANTLR 4 [35] and GNU Bison [18], using their Java backends. ANTLR 4 is a popular parser generator that implements an efficient parsing algorithm called ALL(*) [36]. The ALL(*) algorithm can perform an unlimited number of lookaheads to resolve ambiguity, and it has a worst-case complexity of $O(n^4)$; however, it exhibits linear behavior on many practical grammars. GNU Bison is a well-known parser generator that implements a class of linear-time LR parsing algorithms [11]. We utilized GNU Bison's default configuration to build LALR(1) parsers [10] in conjunction with JFlex [41], a Java lexical analyzer generator. We also compared the VPG parsers with a few hand-crafted parsers specialized for parsing JSON and XML documents, including four

mainstream JavaScript engines and four popular XML parsers. Before presenting the performance evaluation, we list some general setups:

(1) During evaluation, we adapted the grammars for JSON, XML, and HTML from ANTLR4 [3] to tagged CFGs, from which we generated VPG parsers. Appendix F shows the tagged CFGs for JSON, XML, and HTML. We also adapted these grammars to LR grammars and generated LR parsers using Bison. We then compared VPG parsers with ANTLR and Bison parsers in terms of performance.
(2) When comparing our VPG parsers with ANTLR parsers, we omitted the lexing time. This is because we used ANTLR's lexers to generate the tokens for both VPG parsers and ANTLR parsers.
(3) Since ANTLR generates CFG parse trees, for an end-to-end comparison, we used semantic actions to convert the parsing results of VPG parsers and Bison parsers to CFG parse trees and measured the conversion time.

We note that the performance of the parsing algorithms presented in this article is very similar to the performance of the algorithms in the conference version, since both utilize two PDAs and the difference in lookup time between PDA transitions is negligible. We adopted the new algorithms primarily to simplify the verification of both correctness and performance.

8.1 Parsing JSON Files

The JSON format allows objects to be nested within objects and arrays; therefore, a JSON object has a hierarchically nesting structure, which can be naturally captured by a VPG. In particular, since in JSON an object is enclosed within "{" and "}" and arrays within "[" and "]", its VPG grammar treats "{" and "[" as call symbols and "}" and "]" as return symbols.

When building a VPG parser for JSON, we reused ANTLR's lexer. Therefore, the evaluation steps are as follows:

$$\textit{Input file} \xrightarrow{\textit{ANTLR Lexer}} \textit{ANTLR tokens} \xrightarrow{\textit{ANTLR/VPG Parser}} \textit{Results}.$$

A downstream application that uses the ANTLR's JSON parser may wish to keep working on the same parsing result produced by ANTLR's parser. Therefore, we implemented a converter to convert the parse forest produced by our VPG parser to ANTLR's parse tree for the input files. When the grammar is unambiguous, which is the case for the JSON grammar and the XML grammar, the parse forest is really the encoding of a single parse tree. The algorithm of converting a VPG parse tree to a stack machine and evaluating the stack machine has been discussed in Section 7. The result of the evaluation is a structure that can be directly printed out and compared with; the same applies to the parse trees produced by ANTLR and Bison parsers. The conversion steps are summarized as follows:

$$\textit{VPG parse tree} \xrightarrow{\textit{Embed actions}} \textit{Stack machine} \xrightarrow{\textit{Evaluate}} \textit{ANTLR parse tree}.$$

Note that in practice this conversion may not be necessary. A downstream application can directly work on the VPG parse tree or associate semantic actions with the VPG parse tree to convert the tree to desired semantic values. We include the conversion time for our VPG parser so the parsing result can be compatible with legacy downstream applications.

For evaluation, we collected 23 real-world JSON files from the Awesome JSON Datasets [12], the Native JSON Benchmark [43], and the JSON.parse benchmark [19], where the sizes of the files

Table 1. Parsing and Lexing Times of JSON Files

Name	#Token	ANTLR Parse	Bison Parse	VPG Parse + Extract	VPG Conv	VPG Total	ANTLR Lex	JFlex
catalog	135,991	54 ms	20 ms	7 ms	17 ms	24 ms	40 ms	39 ms
canada	334,374	104 ms	50 ms	17 ms	52 ms	69 ms	47 ms	39 ms
educativos	426,398	92 ms	41 ms	20 ms	48 ms	69 ms	109 ms	75 ms
airlines	555,410	107 ms	58 ms	30 ms	68 ms	98 ms	88 ms	91 ms
JSON.parse	1,288,350	225 ms	128 ms	72 ms	162 ms	234 ms	142 ms	141 ms

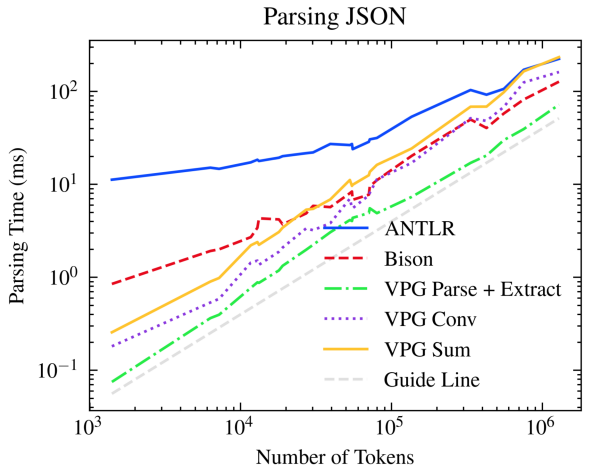


Fig. 8. Parsing times of JSON files (in log scale). "Guide Line" indicates the slope for linear time complexity.

range from 14 KB to 7 MB.[4] Figure 8 shows the parsing time of the ANTLR and Bison parsers, as well as the VPG parser's parsing plus extraction time, conversion time, and the total (parsing plus extraction plus conversion) time; note that both the x-axis and the y-axis of the figure (and other figures in this section) are in the log scale for better visualization; Table 1 shows the parsing time for the five largest files in our test set, as well as the lexing time; Appendix H shows the results for the full test set. All parsers exhibit linear-time performance when parsing JSON files. As can be seen, the VPG parsing time ("VPG Parse + Extract") is less than the time by ANTLR and Bison parsers; the total time is in general also less than the ANTLR parser and comparable to the Bison parser. For larger files, we observe that the conversion time dominates; we think the time is largely influenced by the OCaml garbage collector and defer optimizing the conversion for future work.

8.2 Parsing XML Files

XML also has a well-matched nesting structure with explicit start-tags such as and matching end-tags such as . However, compared to JSON, there is an additional complexity for the XML grammar, which makes it necessary to adapt the XML grammar provided by ANTLR. In particular, the XML lexer in ANTLR treats an XML tag as separate tokens; e.g., is converted into three

[4]Our JSON benchmarks and benchmarks for testing XML and HTML parsers are hosted at https://bitbucket.org/psu_soslab/verifiedvpgparser/.

Table 2. Parsing and Lexing Times of XML Files

Name	#Token	ANTLR Parse	Bison Parse	VPG Parse + Extract	VPG Conv	VPG Total	ANTLR Lex	JFlex
ORTCA	39,072	30 ms	8 ms	3 ms	6 ms	10 ms	54 ms	71 ms
SUAS	118,446	54 ms	31 ms	6 ms	21 ms	27 ms	88 ms	131 ms
address	1,437,142	254 ms	232 ms	74 ms	283 ms	357 ms	201 ms	250 ms
cd	4,198,694	721 ms	735 ms	204 ms	704 ms	908 ms	416 ms	509 ms
po	9,266,526	1,360 ms	1,835 ms	461 ms	1,528 ms	1,989 ms	854 ms	1,617 ms

tokens: <, p, and >. Those tokens then appear in the ANTLR XML grammar. Part of the reason for this design is that the XML format allows additional attributes within a tag; e.g., `<p id=1>` is a start-tag with an attribute with name `id` and value 1. Below is a snippet of the related XML grammar in ANTLR.

```
element : '<' Name attribute* '>' content '<' '/' Name '>'
        | '<' Name attribute* '/>' ;
```

Note that the attribute information can be specified by regular expressions and therefore be provided by lexers. To compare VPG parsers with ANTLR and Bison parsers and to expose the nesting structure within XML, we modified the grammar by grouping tag components. The following shows a snippet of our adapted XML grammar:

```
element : <OpenTag content CloseTag> | SingleTag ;
```

The above snippet introduces three new tokens, as declared below.

```
OpenTag : '<' Name attribute* '>' ;
CloseTag : '<' '/' Name '>' ;
SingleTag : '<' Name attribute* '/>' ;
```

Note that the above grammar is used for ANTLR, Bison, and our VPG parser generator, and we do not need the VPG lexer used in the conference version of this article.

For evaluation, we used the real-world XML files provided by the VTD-XML benchmarks [42], which consist of a wide selection of 23 files ranging from 1 K to 73 MB. The parsing times are presented in Figure 9; the times of the largest 5 files are presented in Table 2; the conversion times are shown in the "VPG Conv" column. Appendix H shows the results for the full test set. Similar to JSON, VPG parsing ("VPG Parse + Extract") on XML files is in general faster than ANTLR and Bison parsing; the total time is also less for smaller files.

8.3 Parsing HTML Files

A snippet of the HTML grammar in ANTLR is listed below:

```
htmlElement: '<' TAG_NAME htmlAttribute*
    ('>' (htmlContent '<' '/' TAG_NAME '>')? | '/' '>' ) ;
htmlContent: htmlChardata?
    ((htmlElement | CDATA | htmlComment) htmlChardata?)* ;
```

Similar to the XML grammar, the HTML grammar allows self-closing tags such as `<br/>`. However, the HTML grammar in addition allows optional end tags, which is not allowed in XML. For example, the HTML tag `<input type="submit" value="Ok">` cannot have a matching end tag

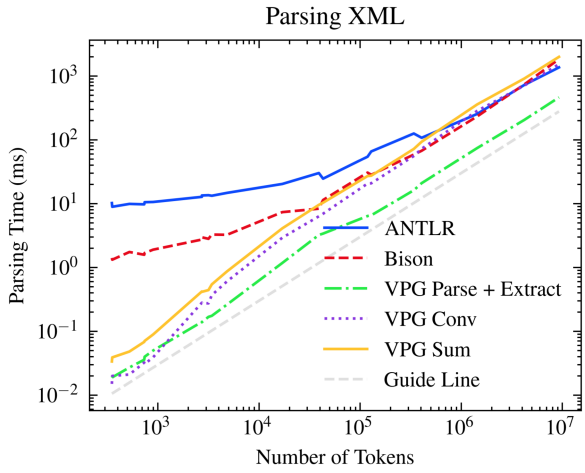


Fig. 9. Parsing times of XML files (in log scale). "Guide Line" indicates the slope for linear time complexity.

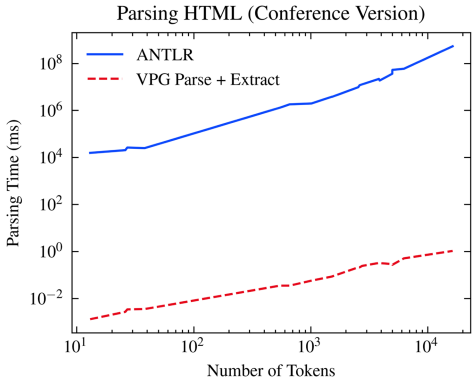


Fig. 10. Parsing times of HTML files (in log scale).

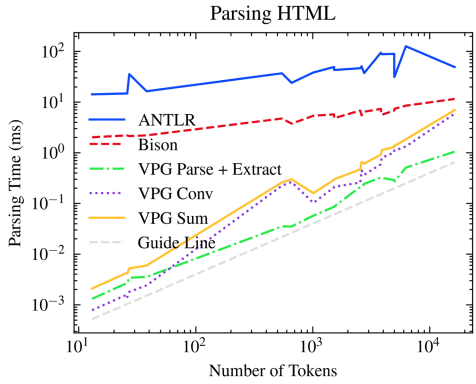


Fig. 11. Parsing times of HTML files (in log scale). "Guide Line" indicates the slope for linear time complexity.

according to the HTML standard. Although this kind of tag is also "self-closing," we will use the terminology of optional end tags, since that is how the official HTML5 standard describes it.

In the conference version of this article, we used the HTML grammar included in ANTLR. To adapt the grammar for VPGs, we combined the components of an HTML tag into a single token and tag it as a call or return symbol for our VPG parser. The performance comparison between the ANTLR parser and the VPG parser is presented in Figure 10. As can be seen, our VPG parser outperformed the ANTLR parser by more than four orders of magnitude.

We performed an investigation and identified the reason for the huge performance gap. The HTML grammar included in ANTLR does not distinguish different kinds of tags at the grammar level; all tags are modeled by a single nonterminal. The grammar by itself is not practical and causes nondeterminism at the grammar level. As a result, the ANTLR parser cannot determine how tags are matched until all tags have been read, which significantly slows down its parsing. For the same reason, this HTML grammar cannot be refactored to an LR grammar, which is required by Bison.

To address this issue, we changed the grammar to reduce nondeterminism by explicitly modeling 10 most common tags at the grammar level and requiring their correct matching. The most essential part of the updated HTML grammar is as follows:

```
htmlElement = <OpenTag_h1 htmlElement* CloseTag_h1>
            | ...
            | <OpenTag_table htmlElement* CloseTag_table>
            | OpenTag
            | CloseTag
            | SingleTag ;
```

For example, we require that table and paragraph tags be correctly matched; other tags are captured by `OpenTag` and `CloseTag`. The new tokens are declared as follows:

```
OpenTag_h1 : '<' 'h1' htmlAttribute* '>' ;
CloseTag_h1 : '<' '/' 'h1' '>' ;
...
OpenTag_table : '<' 'table' htmlAttribute* '>' ;
CloseTag_table : '<' '/' 'table' '>' ;
OpenTag : '<' TAG_NAME htmlAttribute* '>' ;
CloseTag : '<' '/' TAG_NAME '>' ;
SingleTag : '<' TAG_NAME htmlAttribute* '/' '>' ;
```

Similar to XML, the above grammar is used for all parser generators.

Another point is that the HTML grammar is ambiguous. For example, the first rule

```
htmlDocument: scriptletOrSeaWs* XML?
    scriptletOrSeaWs* DTD? scriptletOrSeaWs* htmlElements* ;
```

can parse a string of `scriptletOrSeaWs` in different ways. ANTLR's HTML parser returns the parse tree that prioritizes earlier rules in the grammar; our VPG parser returns the same parse tree as ANTLR's with a special pickOne function that always picks the first element in the list of partial parse trees.

For evaluation, we used the 19 real-world HTML files provided in ANTLR's repository [3]; the sizes of these files are smaller than the JSON and XML documents we used. The parsing time is presented in Figure 11 and Table 3. The conversion times of the parse trees are shown in the "VPG Conv" column. As can be seen, our VPG parser outperforms ANTLR and Bison, which is particularly evident for smaller file sizes. Furthermore, the performance gap between the ANTLR parser and the VPG parser is much narrower after the reduction of grammar nondeterminism, confirming our hypothesis that the ANTLR's parser struggled with the nondeterminism in the original HTML grammar.

Summary of comparison with ANTLR and Bison. Our performance evaluation shows that our VPG parsing library generates parsers that in general run faster than those generated by ANTLR, and have comparable performance as the parsers generated by Bison, on grammars that can be converted to VPGs, such as JSON, XML, and HTML.

8.4 Comparison with Hand-crafted Parsers

We also compared VPG parsers with hand-crafted parsers for JSON and XML documents. For JSON, we compared with four mainstream JavaScript engines (V8, Chakra, JavaScriptCore, and SpiderMonkey) and evaluated them on the JSON files discussed in Section 8.1. The four JavaScript

Table 3. Parsing and Lexing Times of HTML Files

Name	#Token	ANTLR Parse	Bison Parse	VPG Parse + Extract	VPG Conv	VPG Total	ANTLR Lex	JFlex
cnn1	4,974	90 ms	7 ms	0.29 ms	1.02 ms	1.31 ms	42 ms	34 ms
reddit2	4,976	32 ms	7 ms	0.28 ms	1.11 ms	1.39 ms	54 ms	42 ms
reddit	4,989	31 ms	7 ms	0.27 ms	1.10 ms	1.37 ms	53 ms	38 ms
digg	6,250	127 ms	9 ms	0.51 ms	1.34 ms	1.85 ms	52 ms	38 ms
youtube	16,316	49 ms	12 ms	1.05 ms	5.92 ms	6.97 ms	55 ms	59 ms

Table 4. Parsing Times of Five Largest JSON Files

Name	#Token	ANTLR Lex	VPG Parse + Extract	VPG Total	ANTLR Lex + VPG Total	SpiderM	JSCore	V8	Chakra
catalog	135,991	40 ms	7 ms	24 ms	64 ms	34 ms	71 ms	28 ms	25 ms
canada	334,374	47 ms	17 ms	69 ms	115 ms	57 ms	68 ms	34 ms	44 ms
educativos	426,398	109 ms	20 ms	69 ms	177 ms	71 ms	421 ms	45 ms	49 ms
airlines	555,410	88 ms	30 ms	98 ms	186 ms	74 ms	95 ms	42 ms	56 ms
JSON.parse	1,288,350	142 ms	72 ms	234 ms	376 ms	118 ms	139 ms	76 ms	88 ms

"SpiderM" stands for "SpiderMonkey," and "JSCore" for "JavaScriptCore."

engines invoke the JavaScript built-in method `JSON.parse` to convert a JSON string to a JSON object. For XML, we compared with four popular XML parsers (fast-xml-parser [32], libxmljs [28], sax-js [40], and htmlparser2 [16]) and evaluated them with the XML files discussed in Section 8.2. One caveat of this comparison is that these parsers generate different parsing results. Fast-xml-parser transforms an XML document into a JSON object; libxmljs transforms an XML document into a custom object that represents the tree structure of XML documents. Sax-js and htmlparser2 parse an XML document and execute user-supplied semantic actions; no semantic actions were used in our evaluation. Additionally, for XML documents, the time required to analyze tag components is not included in the VPG parsing time but is included in the ANTLR lexing time; for better comparison, we reported the combined lexing and parsing times.

The evaluation results for the largest files are shown in Tables 4 and 5; the full results are in Appendix H. Note that the hand-crafted parsers can process raw texts directly, while our VPG parsers process the tokens generated by ANTLR's lexers. Therefore, we show separately the lexing time of ANTLR (column "ANTLR Lex"), the parsing time of VPG parsing (column "VPG Parse + Extract"), the total time (column "VPG Total"), and the combined time (column "ANTLR Lex + VPG Total"). From the results, we can see that although the total time of VPG parsing is not the shortest among all parsers, the parsing time ("VPG Parse + Extract") alone is. Thus, VPG parsers show promising potential in performance, in addition to verified correctness over hand-crafted parsers. The total parsing time can be reduced by replacing ANTLR's lexer with a faster, custom lexer, since the parsing time of VPG parsers is shorter than the lexing time.

9 FUTURE WORK

As noted earlier, the correctness of our VPG-based parser generator is verified in Coq. Correctness means that if the generated parser constructs a parse tree, it must be a valid parse tree according to the input VPG and vice versa. However, there are gaps between our VPG parser generator's

Table 5. Parsing Times of Five Largest XML Files

Name	#Token	ANTLR Lex	VPG Parse + Extract	VPG Total	ANTLR Lex + VPG Total	Fast-XML	Libxmljs	SAX-JS	HTMLP2
ORTCA	39,072	54 ms	3 ms	10 ms	64 ms	138 ms	91 ms	665 ms	89 ms
SUAS	118,446	88 ms	6 ms	27 ms	115 ms	254 ms	182 ms	1,214 ms	169 ms
address	1,437,142	201 ms	74 ms	357 ms	558 ms	584 ms	196 ms	1,012 ms	331 ms
cd	4,198,694	416 ms	204 ms	908 ms	1,324 ms	1,298 ms	419 ms	2,103 ms	735 ms
po	9,266,526	854 ms	461 ms	1,989 ms	2,843 ms	3,278 ms	897 ms	6,618 ms	1,827 ms

"HTMLP2" stands for "HTMLParser2."

Coq formalization of and its implementation in OCaml. First, the implementation takes tagged CFGs as input and translates tagged CFGs to VPGs; the translation algorithm's correctness and termination have not been formally modeled and verified in Coq. The verification effort would be a challenge, as it depends on a global dependency graph; we leave this substantial effort to future work. Second, the implementation uses efficient data structures for performance, while their Coq models use equivalent data structures that are slower but easier for reasoning. For example, the OCaml implementation uses hash tables for storing transition tables of the two PDAs to have efficient search (with $O(1)$ search complexity), while the Coq counterpart uses a balanced tree (with $O(\log(n))$ search complexity) provided as a Coq library. Therefore, we extracted OCaml code from the Coq implementation of the parser generator and used it to generate offline parser and extraction PDAs with Algorithms 2 and 3.

Our parsing algorithm requires a VPG as the input grammar. Compared to a CFG, a VPG requires partitioning terminals into plain, call, and return symbols. Some CFGs may not admit such a partitioning; the same terminal may require different stack actions for different input strings. In particular, all languages recognized by VPGs belong to the set of deterministic context-free languages, which is a strict subset of context-free languages (the classic example that separates CFL from DCFL is $\{a^i b^j c^k \mid i \neq j \vee j \neq k\}$). We plan to extend our preliminary study on ANTLR grammars to understand how much of the syntax of practical computer languages (e.g., programming languages and file formats) can be described by VPGs.

Our framework requires two kinds of refactoring. First, as discussed in Section 8, we refactored the XML and HTML grammars by grouping the components of a tag as a single token. In general, given a grammar, we can combine tokens in the grammar as long as the combined token follows a regular expression pattern. This refactoring can be easily achieved by a lexer generator such as JFlex [41]. To reduce user burden, we intend to provide an additional operator in tagged CFGs for grouping tokens; the user can further mark a combined token as a call or return symbol. Second, we must eliminate left recursion because it is disallowed by our validator for tagged CFGs. Eliminating the commonly used direct left recursion is simple [36] and should be supported by tagged CFGs. In general, we believe that the refactoring effort required by our framework is less than the LR(k) refactoring effort required to resolve shift-reduce and reduce-reduce conflicts. We plan to compare refactoring efforts in future work.

Error information is essential for assisting users in identifying errors in grammars and input strings. When a parsing error occurs, our VPG parser outputs the current configuration, which includes the parser PDA's state and stack. The user is able to view the most recently used rules as well as their context rules, which we frequently found helpful in locating the issue. Furthermore, with the default "pickOne" function, we verified that the extraction PDA should run without error. For custom "pickOne" functions, users can provide their own error information. In future

work, we plan to evaluate the error reporting of VPG parsers and compare them to LL and LR parsers.

The translation algorithm from tagged CFGs to VPGs is sound but not complete. In general, it is an open problem to determine whether a CFG can be translated to a VPG, and to infer the call and return symbols automatically.

10 CONCLUSIONS

In this article, we present a recognizer and a parser generator for visibly pushdown grammars with formally verified correctness and time complexity, where the parsing algorithm is largely enlightened by the recognizer. We also provide a surface grammar called tagged CFGs and a translator from tagged CFGs to VPGs. We show that when a format can be modeled by a VPG and its call and return symbols can be identified, VPG parsing provides competitive performance and sometimes a significant speedup.

APPENDICES

A CORRECTNESS PROOFS OF THE RECOGNIZER

Lemma A.1. *If* $(S_1, T) \rightsquigarrow w \wedge S_1 \subseteq S_2$, *then* $(S_2, T) \rightsquigarrow w$.

Proof. By definition, if $T = \bot$, then

$$(S_1, T) \rightsquigarrow w \Rightarrow \exists (L_1, L_2) \in S_1, L_2 \rightarrow^* w.$$

Since $S_1 \subseteq S_2$, we have $(L_1, L_2) \in S_2$, so $(S_2, T) \rightsquigarrow w$.

Otherwise, $T = [S', \langle a] \cdot T'$, then $w = w_1 b \rangle w_2$, and $\exists (L_3, L_4) \in S_1$ s.t.

(1) $L_4 \rightarrow^* w_1$ and
(2) $\exists (L_1, L_2) \in S', \exists L_5, L_2 \rightarrow \langle a L_3 b \rangle L_5 \wedge (\{(L_1, L_5)\},\ T') \rightsquigarrow w_2$.

Again, since $S_1 \subseteq S_2$, we have $(L_3, L_4) \in S_2$, so $(S_2, T) \rightsquigarrow w$. □

Lemma A.2. *If* $L_2 \rightarrow^* w_1 L_3$ *and* $(\{(L_1, L_3)\}, T) \rightsquigarrow w$, *then* $(\{(L_1, L_2)\}, T) \rightsquigarrow w_1 w$.

Proof. By definition, if $T = \bot$, then

$$(\{(L_1, L_3)\}, \bot) \rightsquigarrow w \text{ implies } L_3 \rightarrow^* w.$$

Thus, $L_2 \rightarrow^* w_1 L_3$ implies $L_2 \rightarrow^* w_1 w$. By definition, $(\{(L_1, L_2)\}\bot) \rightsquigarrow w_1 w$.

Otherwise, $T = [S', \langle a] \cdot T'$, then $w = w' b \rangle w''$,

(1) $L_3 \rightarrow^* w'$ and
(2) $\exists (L', L'') \in S', \exists L_5, L'' \rightarrow \langle a L_1 b \rangle L_5 \wedge (\{(L', L_5)\},\ T') \rightsquigarrow w''$.

Thus, $L_2 \rightarrow^* w_1 L_3$ implies $L_2 \rightarrow^* w_1 w'$. By definition, $(\{(L_1, L_2)\}, T) \rightsquigarrow w_1 w$. □

Lemma A.3. *If* $(S, T) \rightsquigarrow w$, *then* $\exists (L_1, L_2) \in S$, *s.t.* $(\{(L_1, L_2)\}, T) \rightsquigarrow w$.

Proof. By definition, if $T = \bot$, then

$$(S, T) \rightsquigarrow w \Rightarrow \exists (L_1, L_2) \in S, L_2 \rightarrow^* w.$$

Then, by definition, again we have the lemma.

Otherwise, $T = [S', \langle a] \cdot T'$ and $w = w_1 b \rangle w_2$. By definition, again we have the lemma. □

Theorem A.4. *Assume* $\delta_c(S) = (S', \lambda T.T)$ *for a plain symbol* c. *Then* $(S, T) \rightsquigarrow cw$ *iff* $(S', T) \rightsquigarrow w$.

Proof. $\Rightarrow$. By case over $(S, T) \rightsquigarrow cw$.

(1) $T = \bot$ and $(S, \bot) \rightsquigarrow cw$. By definition, $\exists (L_1, L_2) \in S \wedge L_2 \rightarrow^* cw$. By the forms of VPG rules, we must have

$$\exists L_3, L_2 \rightarrow cL_3 \ \wedge \ L_3 \rightarrow^* w.$$

Since $(L_1, L_2) \in S \wedge L_2 \rightarrow cL_3$, by the definition of δ_c, and we have $(L_1, L_3) \in S'$, thus, since $L_3 \rightarrow^* w$, we have $(S', \bot) \rightsquigarrow w$.

(2) $T = [S_1, \langle a] \cdot T'$ and $(S, [S_1, \langle a] \cdot T') \rightsquigarrow cw$. By definition, $w = w_1 b \rangle w_2$, and $\exists (L_3, L_4) \in S$ s.t.
   (a) $L_4 \rightarrow^* cw_1$ and
   (b) $\exists (L_1, L_2) \in S_1, \exists L_5, L_2 \rightarrow \langle aL_3 b \rangle L_5 \wedge (\{(L_1, L_5)\}, T') \rightsquigarrow w_2$.
   Since $L_4 \rightarrow^* cw_1$, we have $\exists L_4', L_4 \rightarrow cL_4' \ \wedge \ L_4' \rightarrow^* w_1$, thus $(L_3, L_4') \in S'$, and $(S', T) \rightsquigarrow w_1 b \rangle w_2 = w$.

$\Leftarrow$.

By Lemma A.3,

$$\exists (L_1, L_3) \in S' \text{ s.t. } (\{(L_1, L_3)\}, T) \rightsquigarrow w.$$

Thus, by the definition of derivatives, $\exists L_2, (L_1, L_2) \in S \wedge L_2 \rightarrow cL_3$. By Lemma A.2, $(\{(L_1, L_2)\}, T) \rightsquigarrow cw$. By Lemma A.1, $(S, T) \rightsquigarrow cw$. □

Theorem A.5. *Assume* $\delta_{\langle a} S = (S', \lambda T.[S, \langle a] \cdot T)$ *for a call symbol* $\langle a$. *Then* $(S, T) \rightsquigarrow \langle aw$ *iff* $(S', [S, \langle a] \cdot T) \rightsquigarrow w$.

Proof. $\Rightarrow$. By case over $(S, T) \rightsquigarrow \langle aw$.

(1) $T = \bot$ and $(S, \bot) \rightsquigarrow \langle aw$. By definition, $\exists (L_1, L_2) \in S \wedge L_2 \rightarrow^* \langle aw$. Thus, from the forms of VPG rules, we have

$$\exists L_3, L_4, b\rangle \text{ s.t. } L_2 \rightarrow \langle aL_3 b\rangle L_4 \rightarrow^* \langle aw.$$

Thus, $\exists w_1, w_2$, s.t. $L_3 \rightarrow^* w_1 \wedge L_4 \rightarrow^* w_2 \wedge w = w_1 b \rangle w_2$.
By definition,

$$(\{(L_1, L_4), \bot\}) \rightsquigarrow w_2,$$

thus $(\{(L_3, L_3)\}, [S, \langle a] \cdot \bot) \rightsquigarrow w_1 b \rangle w_2$. Since $(L_3, L_3) \in \delta_{\langle a} S$, we have $(\delta_{\langle a} S, [S, \langle a] \cdot \bot) \rightsquigarrow w_1 b \rangle w_2 = w$.

(2) $T = [S_1, \langle c] \cdot T'$ and $(S, [S_1, \langle c] \cdot T') \rightsquigarrow \langle aw$. Then, by definition, we have $w = w_1 d \rangle w_2$, and $\exists (L_3, L_4) \in S$ s.t.
   (a) $L_4 \rightarrow^* \langle aw_1$ and
   (b) $\exists (L_1, L_2) \in S_1, \exists L_5, L_2 \rightarrow \langle cL_3 d \rangle L_5 \wedge (\{(L_1, L_5)\}, T') \rightsquigarrow w_2$.
   Thus, $\exists L_6, L_7$ s.t. $L_4 \rightarrow \langle aL_6 b\rangle L_7$, $L_6 \rightarrow^* w_{11}$, $L_7 \rightarrow^* w_{12}$ and $w_1 = w_{11} b \rangle w_{12}$. Thus, by definition, $(L_6, L_6) \in \delta_{\langle a} S$, so

$$(\delta_{\langle a} S, [S, \langle a] \cdot [S_1, \langle c] \cdot T') \rightsquigarrow w_{11} b \rangle w',$$

where w' satisfies $(\{(L_3, L_7)\}, [S_1, \langle c] \cdot T') \rightsquigarrow w'$. From $(\{(L_1, L_5)\}, T') \rightsquigarrow w_2$, we know w' can be $w_{12} d \rangle w_2$, so we have

$$(\delta_{\langle a} S, [S, \langle a] \cdot T) \rightsquigarrow w_{11} b \rangle w_{12} d \rangle w_2 = w.$$

$\Leftarrow$. By definition, $\exists (L_3, L_3) \in S'$, s.t. $w = w_1 b \rangle w_2$, and

(1) $\exists (L_1, L_2) \in S$ s.t. $L_3 \rightarrow^* w_1 \wedge L_2 \rightarrow \langle aL_3 b\rangle L_5$.
(2) $(\{(L_1, L_5)\}, T) \rightsquigarrow w_2$.

Since $L_2 \rightarrow \langle aL_3 b\rangle L_5 \wedge (\{(L_1, L_5)\}, T) \rightsquigarrow w_2$, by Lemma A.2, we have $(\{L_1, L_2\}, T) \rightsquigarrow \langle aw_1 b \rangle w_2 = \langle aw$, thus $(S, T) \rightsquigarrow \langle aw$. □

Theorem A.6. *Assume $\delta_{b\rangle}(S, [S_1, \langle a]) = (S', tail)$ for a return symbol $b\rangle$. Then $(S, [S_1, \langle a] \cdot T) \rightsquigarrow b\rangle w$ iff $(S', T) \rightsquigarrow w$.*

Proof. $\Rightarrow$. By definition, $\exists (L_3, L_4) \in S$ and $(L_1, L_2) \in S_1$, and we must have $L_4 \rightarrow^* \epsilon$, $L_2 \rightarrow \langle a L_3 b\rangle L_5$ and $(\{(L_1, L_5)\}, T) \rightsquigarrow w$. From $L_4 \rightarrow^* \epsilon$ and the forms of allowed VPG rules, we must have $L_4 \rightarrow \epsilon$; therefore, $(L_1, L_5) \in S'$. By Lemma A.1, we have $(S', T) \rightsquigarrow w$.

$\Leftarrow$. By Lemma A.3, we have $(L_1, L_5) \in S' \wedge (\{(L_1, L_5)\}, T) \rightsquigarrow w$, thus $\exists (L_1, L_2) \in S_1 \wedge (L_3, L_4) \in S \wedge L_4 \rightarrow \epsilon \wedge L_2 \rightarrow \langle a L_3 b\rangle L_5$, by definition, we have $(S, [S_1, \langle a] \cdot T) \rightsquigarrow b\rangle w$. □

Lemma A.7. *Given a VPG $G = (V, \Sigma, P, L_0)$, suppose a PDA is generated according to Algorithm 1. Then, for a string $w \in \Sigma^*$, $(\{(L_0, L_0)\}, \bot) \rightsquigarrow w$ iff w is accepted by the PDA.*

Proof. $\Leftarrow$. If w of length k is accepted by the PDA, then there exists a sequence of configurations $(S_i, T_i), i \in [0..k]$, s.t. (1) $S_0 = \{(L_0, L_0)\}$, $T_0 = \bot$; (2) $(S_i, T_i) = \mathcal{F}(w_i, S_{i-1}, T_{i-1})$, $i \in [1..k]$, where w_i is the ith symbol in w and $\mathcal{F}$ is the PDA transition function; and (3) (S_k, T_k) is an acceptance configuration.

For each i, perform case analysis over w_i. Suppose w_i is a plain symbol, denoted as c. By $(S_i, T_i) = \mathcal{F}(w_i, S_{i-1}, T_{i-1})$ and the PDA construction, we must have $\delta_c(S_{i-1}) = (S_i, \lambda T.T)$. By Theorem A.4, we get $(S_{i-1}, T_{i-1}) \rightsquigarrow w_i \dots w_k$ iff $(S_i, T_i) \rightsquigarrow w_{i+1} \dots w_k$. The cases for when w_i is $\langle a$ or $b\rangle$ are similar, with the help of Theorems A.5 and A.6.

Combining all steps, we have $(S_0, T_0) \rightsquigarrow w$ iff $(S_k, T_k) \rightsquigarrow \epsilon$. Since (S_k, T_k) is an acceptance configuration, we have $(S_k, T_k) \rightsquigarrow \epsilon$. Therefore, we get $(S_0, T_0) \rightsquigarrow w$.

$\Rightarrow$. We prove a more general lemma: If (S_0, T_0) is a PDA runtime configuration and $(S_0, T_0) \rightsquigarrow w$, then w is accepted by the PDA. Prove it by induction over the length of w.

When the length is zero, we must have $T_0 = \bot$ and there exists $(L_1, L_2) \in S_0$ such that $L_0 \rightarrow \epsilon$. Therefore, (S_0, T_0) is an acceptance configuration of the PDA.

For the inductive case, suppose $w = w_1 \dots w_{k+1}$. Perform case analysis over w_1, and first show that there exists (S_1, T_1) s.t. $(S_1, T_1) = \mathcal{F}(w_1, S_0, T_0)$.

(1) Suppose w_1 is a plain symbol c and $\delta_c(S_0) = (S_1, \lambda T, T)$. Since Algorithm 1 is closed under derivatives, we have that S_1 is a PDA state. Let $T_1 = T_0$. Thus, $\mathcal{F}(w_1, S_0, T_0) = (S_1, T_1)$ by the definition of $\mathcal{F}$.
(2) The case of w_1 being a call symbol is similarly to the previous case.
(3) Suppose w_1 is a return symbol $b\rangle$. By $(S_0, T_0) \rightsquigarrow b\rangle w_2 \dots w_{k+1}$, we have T_0 is not the empty stack and has a top symbol $[S, \langle a]$. Suppose $\delta_{b\rangle}(S_0, [S, \langle a]) = (S_1, \text{tail})$. Since Algorithm 1 is closed under derivatives, we have that S_1 is a PDA state. Let $T_1 = \text{tail}(T_0)$. Thus, $\mathcal{F}(w_1, S_0, T_0) = (S_1, T_1)$ by the definition of $\mathcal{F}$.

By Theorems A.4, A.5, and A.6, we get $(S_1, T_1) \rightsquigarrow w_2 \dots w_{k+1}$. By the induction hypothesis, $w_2 \dots w_{k+1}$ is accepted by the PDA. Therefore, the original string $w_1 \dots w_{k+1}$ is also accepted. □

Theorem A.8. *For VPG G and its start symbol L_0, a string $w \in \Sigma^*$ is derived from L_0, i.e., $L_0 \rightarrow^* w$, iff w is accepted by the corresponding PDA.*

Proof. By Lemma A.7, w is accepted by the PDA iff $(\{(L_0, L_0)\}, \bot) \rightsquigarrow w$, and by definition, we have $(\{(L_0, L_0)\}, \bot) \rightsquigarrow w$ iff $L_0 \rightarrow^* w$. □

B RECOGNIZING STRINGS WITH PENDING CALLS/RETURNS

In this section, we extend the work in Section 5 to build PDAs for recognizing VPG with pending call or return symbols. In general VPGs, nonterminals are classified to two categories: V^0 for matching well-matched strings and V^1 for strings with pending calls/returns. We write $V = V^0 \cup V^1$ for

the set of all nonterminals. V^0 should be disjoint from V^1. The definition also imposes constraints on how V^0 and V^1 nonterminals can be used. E.g., in $L \to \langle aL_1 b\rangle L_2$, L_1 must be in a well-matched nonterminal (i.e., in V^0). This constraint excludes a grammar like $L_1 \to \langle aL_2 b\rangle L_3; L_2 \to \langle cL_4$.

Another major difference is that in $L \to aL_1$, the symbol a can be a call/return symbol in addition to being a plain symbol. This makes matching calls and returns more complicated. For example, suppose we have rules: $L_1 \to \langle aL_2; L_2 \to b\rangle L_3 | \epsilon; L_3 \to \epsilon$. Then the string $\langle ab\rangle$ is accepted, in which case $b\rangle$ from $L_2 \to b\rangle L_3$ matches $\langle a$ from $L_1 \to \langle aL_2$. String $\langle a$ is also accepted, in which case $\langle a$ is a pending call. So, depending on the input string, $\langle a$ from $L_1 \to \langle aL_2$ may be a matching call or a pending call.

Here is an example grammar:

(1) $L_1 \to \langle aL_2 b\rangle L_3$
(2) $L_2 \to \langle aL_2 b\rangle L_4 \mid \epsilon$
(3) $L_3 \to \langle aL_1 \mid \epsilon$
(4) $L_4 \to \epsilon$.

And $L_1, L_3 \in V^1, L_2, L_4 \in V^0$. For example, $\langle ab\rangle\langle a\langle a\langle ab\rangle b\rangle$ is in the language recognized by the grammar.

General VPGs to PDAs. The PDA states and stack symbols are the same as before. We generalize the notion of the top of the stack to return the top stack symbol when the stack is non-empty and return None when the stack is empty.

A derivative-based transition function takes the current state and the top of the stack (which can be None) and returns a new state and a stack action. As before, since δ_c and $\delta_{\langle a}$ do not use the top of the stack, we omit it from their parameters.

Definition B.1 (Derivative Functions for General VPGs). Given a general VPG $G = (V, \Sigma, P, L_0)$, the transition functions δ are defined as follows: For $c \in \Sigma_{\text{plain}}$, $\langle a \in \Sigma_{\text{call}}$ and $b\rangle \in \Sigma_{\text{ret}}$,

(1) δ_c is the same as the well-matched case.
$\delta_c(S) = (S', \lambda T.T)$, where
$$S' = \{(L_1, L_3) \mid (L_1, L_2) \in S \wedge (L_2 \to cL_3) \in P\};$$
(2) For call symbols, we have $\delta_{\langle a}(S) = (S' \cup S_p, \lambda T.[S, \langle a] \cdot T)$, where
$$\begin{aligned} S' &= \{(L_3, L_3) \mid (L_1, L_2) \in S \ \wedge \exists L_4, (L_2 \to \langle aL_3 b\rangle L_4) \in P\}, \\ S_p &= \{(L_3, L_3) \mid (L_1, L_2) \in S \ \wedge (L_2 \to \langle aL_3) \in P\}. \end{aligned}$$
Compared to the well-matched case, an additional S_p is introduced for the case when $\langle a$ appears in a rule like $L_2 \to \langle aL_3$.
(3) For a return symbol $b\rangle$, if t is the top of the stack, then
$$\delta_{b\rangle}(S, t) = \begin{cases} (S' \cup S_{p1}, \text{tail}) & \text{if } t = [S_1, \langle a] \\ (S_{p2}, \lambda T.T) & \text{if } t = \text{None}, \end{cases}$$
where
$$\begin{aligned} S' &= \{(L_1, L_5) \mid (L_1, L_2) \in S_1 \wedge (L_3, L_4) \in S \wedge \\ &\qquad (L_4 \to \epsilon) \in P \ \wedge (L_2 \to \langle aL_3 b\rangle L_5) \in P\} \\ S_{p1} &= \{(L_1, L_5) \mid (L_1, L_2) \in S_1 \wedge (L_3, L_4) \in S \wedge \\ &\qquad (L_2 \to \langle aL_3) \in P \ \wedge (L_4 \to b\rangle L_5) \in P\} \\ S_{p2} &= \{(L_3, L_3) \mid (L_1, L_2) \in S \wedge (L_2 \to b\rangle L_3) \in P\}. \end{aligned}$$
S' is as before and deals with the case when there is a rule $L_2 \to \langle aL_3 b\rangle L_5$ with a proper top stack symbol. S_{p1} deals with the case when there are rules $L_2 \to \langle aL_3$ and $L_4 \to b\rangle L_5$; in this

case, we match $b\rangle$ with $\langle a$. Finally, S_{p2} deals with the case when the stack is empty; then $b\rangle$ is treated as a pending return symbol (not matched with a call symbol).

For the well-matched case, the stack should be empty after all input symbols are consumed; in the case with pending calls/returns, however, the stack is not necessarily empty at the end. For example, with the grammar $L \to \langle aL \mid \epsilon$ and the valid input string $\langle a$, the terminal stack is $[\{(L, L)\}, \langle a] \cdot \perp$.

Definition B.2 (The Acceptance Configuration for Words with Pending Calls/returns). Given a general VPG $G = (V, \Sigma, P, L_0)$, the pair (S, T) is called an *acceptance configuration* if the following are satisfied:

(1) $\exists (L_1, L_2) \in S$ s.t. $(L_2 \to \epsilon) \in P$,
(2) either (i) $T = \perp$ or (ii) $T = [S', \langle a] \cdot T'$ and $\exists (L_3, L_4) \in S' \wedge (L_4 \to \langle aL_1) \in P$ for some L_1.

In the following correctness proof, we use predicate well-matched(w) to mean that w, a string of terminals, is a well-matched string; that is, every call/return symbol is matched with a corresponding return/call symbol. We use predicate matched-rets(w) to mean that any return symbol in w is matched with a call symbol; however, a call symbol may not be matched with a return symbol. E.g., we have matched-rets($\langle a\langle ab\rangle$), but not well-matched($\langle a\langle ab\rangle$).

Definition B.3 (Semantics of PDA Configurations). We will write $(S, T) \rightsquigarrow w$ to mean that w can be accepted by the configuration (S, T). It is defined as follows:

(1) $(S, \perp) \rightsquigarrow w$ if $\exists (L_1, L_2) \in S$, s.t. $L_2 \to^* w$,
(2) $(S, [S', \langle a] \cdot T') \rightsquigarrow w_1 b\rangle w_2$ if $\exists (L_3, L_4) \in S$ s.t.
  (a) $L_4 \to^* w_1$ and well-matched(w_1) and
  (b) $\exists (L_1, L_2) \in S', \exists L_5, L_2 \to \langle aL_3 b\rangle L_5 \wedge (\{(L_1, L_5)\},\ T') \rightsquigarrow w_2$.
(3) $(S, [S', \langle a] \cdot T') \rightsquigarrow w_1 b\rangle w_2$ if $\exists (L_3, L_4) \in S$ s.t. $\exists L_5$
  (a) $L_4 \to^* w_1 b\rangle L_5$ and well-matched(w_1) and
  (b) $\exists (L_1, L_2) \in S', L_2 \to \langle aL_3 \wedge (\{(L_1, L_5)\},\ T') \rightsquigarrow w_2$.
(4) $(S, [S', \langle a] \cdot T') \rightsquigarrow w_1$ if $\exists (L_3, L_4) \in S$ s.t.
  (a) $L_4 \to^* w_1$ and matched-rets(w_1)
  (b) $\exists (L_1, L_2) \in S', L_2 \to \langle aL_3$.

In the above definition, the third case handles when the call symbol $\langle a$ in rule $L_2 \to \langle aL_3$ matches $b\rangle$ in $w_1 b\rangle L_5$ produced by L_4. The last case handles when $\langle a$ in rule $L_2 \to \langle aL_3$ does not have a matched return; that is, it is a pending call.

The following three lemmas and their proofs are the same as before (except that Lemma B.2 requires well-matched strings):

Lemma B.1. *If* $(S_1, T) \rightsquigarrow w \wedge S_1 \subseteq S_2$, *then* $(S_2, T) \rightsquigarrow w$.

Lemma B.2. *If* $L_2 \to^* w_1 L_3$, well-matched(w_1), *and* $(\{(L_1, L_3)\}, T) \rightsquigarrow w$, *then* $(\{(L_1, L_2)\}, T) \rightsquigarrow w_1 w$.

Lemma B.3. *If* $(S, T) \rightsquigarrow w$, *then* $\exists (L_1, L_2) \in S$, *s.t.* $(\{(L_1, L_2)\}, T) \rightsquigarrow w$.

In addition, we need the following lemma:

Lemma B.4. *IF* $L \to^* w\delta$, *where* δ *is a string of terminals or nonterminals, then we have either (1)* matched-rets(w), *or (2) exists* $w_1, b\rangle, w_2$, *so* $w = w_1 b\rangle w_2$ *and* well-matched(w_1) *and exists* L_1 *so* $L \to^* w_1 b\rangle L_1$ *and* $L_1 \to^* w_2\delta$.

Proof. Sketch: If $w = \epsilon$, then matched-rets(ϵ). Otherwise, prove it by induction over the length of the derivation of $L \to^* w\delta$ and then perform case analysis over the first derivation step. □

Theorem B.5. *For a plain symbol c, $(S, T) \rightsquigarrow cw$ iff $\delta_c(S) = (S', f)$, and $(S', f(T)) \rightsquigarrow w$.*

The proof is similar to the proof before, except the $\Rightarrow$ direction has more cases to consider.

Theorem B.6. *For $\langle a \in \Sigma_{call}$, $(S, T) \rightsquigarrow \langle aw$ iff $\delta_{\langle a} S = (S', f)$, and $(S', f(T)) \rightsquigarrow w$.*

The proof is similar to the proof before, except with more cases to consider. The $\Rightarrow$ direction requires the use of Lemma B.4.

Theorem B.7. *[(1)]*
(1) *If $\delta_{b\rangle}(S, [S_1, \langle a]) = (S', tail)$, then $(S, [S_1, \langle a] \cdot T) \rightsquigarrow b\rangle w$ iff $(S', T) \rightsquigarrow w$.*
(2) *If $\delta_{b\rangle}(S, None) = (S', \lambda T.T)$, then $(S, \bot) \rightsquigarrow b\rangle w$ iff $(S', \bot) \rightsquigarrow w$.*

Part (1)'s proof is similar to before, except with more cases and sometimes need to use Lemma B.4. Part(2)'s proof is straightforward.

ALGORITHM 4: Constructing the recognizer PDA. Differences from Algorithm 1 are highlighted.

Input : A VPG $G = (V, \Sigma, P, L_0)$ where $\Sigma = \Sigma_{\text{call}} \cup \Sigma_{\text{plain}} \cup \Sigma_{\text{ret}}$, δ;
Return : The initial state S_0, the set of all produced states A, the set of acceptance states A_{acc}, and the set of transitions $\mathcal{T}$;

$S_0 \leftarrow \{(L_0, L_0)\}$;
Initialize the set for new states $N \leftarrow \{S_0\}$;
Initialize the set for all produced states $A \leftarrow N$;
Initialize the set for transitions $\mathcal{T} \leftarrow \{\}$;
repeat
  $N' \leftarrow \{(i, f, S, S') \mid (S', f) = \delta_i(S),\ S \in N,\ \text{and } i \in \Sigma_{\text{call}} \cup \Sigma_{\text{plain}}\}$;
  Add edge (S, S') marked with (i, f) to $\mathcal{T}$, where $(i, f, S, S') \in N'$;
  Compute the set of stack elements: $R \leftarrow \{[S, \langle a] \mid S \in A \text{ and } \langle a \in \Sigma_{\text{call}}\}$;
  Compute transitions with return symbols and stack elements in R:
    $N_R \leftarrow \{(b\rangle, r, f, S, S') \mid (S', f) = \delta_{b\rangle}(S, r), S \in A, b\rangle \in \Sigma_{\text{ret}}, r \in R \cup \text{None}\}$;
  Add edge (S, S') marked with $(b\rangle, r, f)$ to $\mathcal{T}$, where $(b\rangle, r, f, S, S') \in N_R$;
  Collect the new states $N \leftarrow \{S' \mid (\_, \_, \_, S') \in N' \vee (\_, \_, \_, \_, S') \in N_R\} - A$;
  Update the set of all produced states $A \leftarrow A \cup N$;
until $N = \emptyset$;
Compute the states of acceptance configurations $A_{\text{acc}} \leftarrow \{S \mid (L', L) \in S \in A, (L \rightarrow \epsilon) \in P\}$

Lemma B.8. *Given a VPG $G = (V, \Sigma, P, L_0)$, suppose a PDA is generated according to Algorithm 4. Then, for a string $w \in \Sigma^*$, $(\{(L_0, L_0)\}, \bot) \rightsquigarrow w$ iff w is accepted by the PDA.*

The lemma can be proved as before, except with more cases.

Theorem B.9. *For VPG G and its start nonterminal L_0, a string $w \in \Sigma^*$ is derived from L_0, i.e., $L_0 \rightarrow^* w$, iff w is accepted by the corresponding PDA.*

The proof is as before.

C PARSER AND EXTRACTION PDAS FOR GENERAL VPGS

In this section, we discuss the parser PDA and the extraction PDA for general VPGs. For general VPGs, the definitions of headNT (), nextNT () , and dotted rules of well-matched VPGs also apply; the definition for plain symbols generalize for pending call and return symbols naturally. We define the derivative functions of the parser PDA as follows:

Definition C.1 (The Derivative Function p for the Parser PDA). Given a VPG $G = (V, \Sigma, P, L_0)$, suppose the current state of the parser PDA is m and the current stack is T, the transition functions p_c, $p_{\langle a}$, and $p_{b\rangle}$ are defined as follows:

(1) For plain symbols, the derivative function of general VPGs is the same as that for well-matched VPGs; we restate the function here: $p_c(m) = (m', \lambda T.T)$, where

$$m' = \{(r', \text{nextNT}\,(r) \to c.L_1) \mid (r', r) \in m \wedge (\text{nextNT}\,(r) \to c.L_1) \in \dot{P}\}.$$

For each pair (r', r) in m, the new state keeps the context rule r' and updates the current rule to a rule with head nextNT (r) and that derives c.

(2) For call symbols, the derivative function of general VPGs is similar to that for well-matched VPGs; we only need to introduce a new condition for pending rules: $p_{\langle a}(m) = (m', \lambda T.\, m \cdot T)$, where

$$m' = \{(r_1, r_1) \mid \exists r'\ r, (r', r) \in m \wedge \Big(\exists L_1\ b\rangle\ L_2, r_1 = (\text{nextNT}\,(r) \to \langle a.L_1 b\rangle L_2) \in \dot{P} \vee \exists L_1, r_1 = (\text{nextNT}\,(r) \to \langle a.L_1) \in \dot{P}\Big)\}.$$

(3) For return symbols, we have two more cases to consider: $p_{b\rangle}(m, \overline{m_{\text{call}}}) = (m', \text{tail})$, where tail is the function that removes the top of a stack and returns the rest of stack (for empty stacks, tail return empty stacks), and $\overline{m_{\text{call}}} = \text{head}(T)$ if $T \neq \bot$, and $\overline{m_{\text{call}}} = \emptyset$ if $T = \bot$, and

$$\begin{aligned} m' = &\{(r', \text{nextNT}\,(r) \to \langle aL_1 b\rangle.L_2) \mid \exists r_1\ r_2, (r_1, r_2) \in m\ \wedge\ (r', r) \in \overline{m_{\text{call}}} \wedge \\ &\quad \exists \langle a\ L_1\ b\rangle\ L_2, r_1 = (\text{nextNT}\,(r) \to \langle a.L_1 b\rangle L_2) \wedge (\text{nextNT}\,(r_2) \to \epsilon) \in P\} \cup \\ &\{(r', r_3) \mid \exists r_1\ r_2, (r_1, r_2) \in m\ \wedge\ (r', r) \in \overline{m_{\text{call}}} \wedge \\ &\quad \exists \langle a\ L_2, r_1 = (\text{nextNT}\,(r) \to \langle a.L_2) \wedge \exists b\rangle\ L_1, r_3 = (\text{nextNT}\,(r_2) \to b\rangle.L_1) \in \dot{P}\} \cup \\ &\{(\text{None}, r_3) \mid \exists r_2, (\text{None}, r_2) \in m \wedge \exists b\rangle\ L_1, r_3 = (\text{nextNT}\,(r_2) \to b\rangle.L_1) \in \dot{P}\}. \end{aligned}$$

We remove the top of the stack and construct the new state in three cases, depending on the context rule r_1 for a pair (r_1, r_2) in m. We have already discussed the case where r_1 is a matching rule in Definition 6.4. Below are the two new cases for pending rules.
If the context r_1 is a pending rule, then the new state updates the current rule to a rule that rewrites nextNT (r_2) and generates $b\rangle$. Similar to the first case, we have $\overline{m_{\text{call}}} \neq \emptyset$; the update of the context rule in the new state is similar.
Otherwise, if there is no context rule, then no call symbol is pending, and the $b\rangle$ symbol is an unmatched return symbol. In this case, we must have $\overline{m_{\text{call}}} = \emptyset$. We construct the new state in a way similar to the second case, except that the context rule is None.

The acceptance configurations for general VPGs also allow the contexts to be pending rules, as shown below.

Definition C.2 (Parser PDA Acceptance Configurations). Given a VPG $G = (V, \Sigma, P, L_0)$, a pair (m, E) is an *acceptance configuration* for the parser PDA if the state m includes a pair (r', r), where r' is either None or a pending rule, and nextNT (r) derives the empty string ϵ, i.e., $\exists r,\ (\text{None}, r) \in m\ \wedge\ (\text{nextNT}\,(r) \to \epsilon) \in P$.

Definition C.3 (The Extraction Function $\text{extract}_{\text{init}}$*).* We only need to add one more condition to the function $\text{extract}^{\text{pre}}_{\text{init}}$ to allow a pending rule as the context; the rest are the same as those of well-matched VPGs.

$$\text{extract}^{\text{pre}}_{\text{init}}(m) = \{r \mid (r', r) \in m, (\text{nextNT}\,(r) \to \epsilon) \in P \wedge r' \text{ is None or a pending rule}\}.$$

Note that this also implies that the stack of a acceptance configuration for general VPGs is not necessarily empty.

Definition C.4 (The One-step Extraction Function). Similar to the case of the function $\text{extract}^{\text{pre}}_{\text{init}}$, we only need to add some additional conditions to the function extendable; the rest are the same as those of well-matched VPGs.

$$\text{extendable}((r', r), (v, E)) = \begin{cases} \text{true}, & \text{if } r' \text{ is None or a pending rule} \wedge \\ & \quad \text{head}(E) \text{ is None or a pending rule} \wedge \\ & \quad \text{nextNT}(r) = \text{headNT}(v) \ \vee \\ & r' \text{ and } \text{head}(E) \text{ are the same matching rule} \\ & \quad \text{(but with different dot positions)} \wedge \\ & \quad ((\text{firstRule}(v) \in \mathcal{P}_{\text{ret}} \wedge \\ & \quad\ (\text{nextNT}(r) \to \epsilon) \in P) \vee \\ & \quad (\text{firstRule}(v) \in \mathcal{P}_{\text{pln}} \cup \mathcal{P}_{\text{call}} \wedge \\ & \quad\ \text{nextNT}(r) = \text{headNT}(v))), \\ \text{false}, & \text{otherwise.} \end{cases}$$

Compared with the case of well-matched VPGs, the only difference is that the context r' can not only be None, but also be a pending rule, and, in accordance, the stack should be either empty or have a pending rule as the stack top.

D CORRECTNESS PROOF OF VPG-BASED PARSING

Given a VPG $G = (V, \Sigma, P, L_0)$ and an input string w, let $V = \text{extract}(\text{forest}(w))$. We call V correct if it includes exactly the set of valid parse trees of w according to G; i.e.,

$$\forall v,\ L_0 \Downarrow (w, v) \iff \exists E,\ (v, E) \in V.$$

To prove the above, we first state and prove the invariants of p and $\text{extract}_{\text{oneStep}}$; the correctness follows as a corollary.

We first discuss the invariants of the parser derivative function p. Intuitively, a rule sequence $[r_1, \ldots, r_{|w|}]$ that derives w should be included in $\text{forest}(w) = [m_1, m_2, \ldots, m_{|w|}]$; i.e., there exists r'_i such that $(r'_i, r_i) \in m_i$ for $i = 1..|w|$. To formalize a rule sequence, we introduce a *forward small-step* relation in Figure 12. Judgment $(v, E) \xrightarrow{i} (v + [r], E')$ means that the rule sequence v can be extended with a rule r, which generates the input symbol i. Here, "forward" means that the relation generates the input from left to right and "small-step" means that the relation generates one symbol at a time, as opposed to the big-step relation in Figure 4. Figure 13 further defines the transitive closure of the small-step relation; consequently, $([], \bot) \xrightarrow{w}{}^{*} (v, E)$ means that v generates w in multiple steps.

The forward small-step relation is flexible in that it allows combining and splitting parse trees. We introduce a series of lemmas to formalize those operations: Lemma D.1 shows we can concatenate two forward small-step relations, assuming the first one has an empty stack; the cases of nonempty stacks are covered in Lemmas D.2 and D.3. Lemma D.4 shows that we can split a forward small-step relation into two parts and replace the first part with another small-step relation. Lemma D.5 shows we can split a forward small-step relation into two parts, and the first part is also a small-step relation. The proofs of these lemmas are directly by induction over the length of w; we omit them here.

$$\frac{(L \to cL_1) \in P \quad (v = [] \wedge E = \bot) \vee \text{nextNT}(v) = L}{(v, E) \xrightarrow{c} (v + [L \to c.L_1], E)}$$

$$\frac{(L \to \langle aL_1) \in P \quad (v = [] \wedge E = \bot) \vee \text{nextNT}(v) = L}{(v, E) \xrightarrow{\langle a} (v + [L \to \langle a.L_1], (L \to \langle a.L_1) :: E)}$$

$$\frac{(L \to b\rangle L_1) \in P \quad (v = [] \wedge E = \bot) \vee \text{nextNT}(v) = L}{(v, E) \xrightarrow{b\rangle} (v + [L \to b\rangle.L_1], \text{tail}(E))}$$

$$\frac{(L \to \langle aL_1 b\rangle L_2) \in P \quad (v = [] \wedge E = \bot) \vee \text{nextNT}(v) = L}{(v, E) \xrightarrow{\langle a} (v + [L \to \langle a.L_1 b\rangle L_2], (L \to \langle a.L_1 b\rangle L_2) \cdot E)}$$

$$\frac{(L \to \langle aL_1 b\rangle L_2) \in P \quad \text{nextNT}(v) = L_3 \quad (L_3 \to \epsilon) \in P}{(v, (L \to \langle a.L_1 b\rangle L_2) \cdot E) \xrightarrow{b\rangle} (v + [L \to \langle aL_1 b\rangle.L_2], E)}$$

Fig. 12. The forward small-step parse-tree derivation relation, given a VPG $G = (\Sigma, V, P, L_0)$.

$$\frac{}{(v, E) \xrightarrow{\epsilon}{}^{*} (v, E)} \qquad \frac{(v_1, E_1) \xrightarrow{w}{}^{*} (v_2, E_2) \quad (v_2, E_2) \xrightarrow{i} (v_3, E_3)}{(v_1, E_1) \xrightarrow{wi}{}^{*} (v_3, E_3)}$$

Fig. 13. The transitive closure of the forward small-step parse-tree derivation relation.

Lemma D.1 (Concatenating Forward Small-step Relations).

$$\forall v_1\ v_2\ w_1\ w_2\ E,\ ([], \bot) \xrightarrow{w_1}{}^{*} (v_1, \bot)\ \wedge\ ([], \bot) \xrightarrow{w_2}{}^{*} (v_2, E)\ \wedge\ \text{nextNT}(v_1) = \text{headNT}(v_2)\ \Longrightarrow\ ([], \bot) \xrightarrow{w_1 w_2}{}^{*} (v_1 + v_2, E).$$

Lemma D.2 (Concatenation with a Pending Call Rule).

$$\forall L\ \langle a\ L_1,\ ([], \bot) \xrightarrow{\langle a}{}^{*} ([L \to \langle a.L_1], (L \to \langle a.L_1) \cdot \bot)\ \Longrightarrow\ \forall v\ E\ w,\ ([], \bot) \xrightarrow{w}{}^{*} (v, E)\ \wedge\ \text{headNT}(v) = L_1\ \Longrightarrow\ ([], \bot) \xrightarrow{\langle a w}{}^{*} ([L \to \langle a.L_1] + v, E)\ \vee\ ([], \bot) \xrightarrow{\langle a w}{}^{*} ([L \to \langle a.L_1] + v, E \cdot (L \to \langle a.L_1) \cdot \bot).$$

Lemma D.3 (Concatenation with a Matching Call Rule).

$$\forall L\ \langle a\ L_1\ b\rangle\ L_2,\ ([], \bot) \xrightarrow{\langle a}{}^{*} ([L \to \langle a.L_1 b\rangle L_2], (L \to \langle a.L_1 b\rangle L_2) \cdot \bot)\ \Longrightarrow\ \forall v\ E\ w,\ ([], \bot) \xrightarrow{w}{}^{*} (v, E)\ \wedge\ \text{headNT}(v) = L_1\ \Longrightarrow\ ([], \bot) \xrightarrow{\langle a w}{}^{*} ([L \to \langle a.L_1 b\rangle L_2] + v, E \cdot (L \to \langle a.L_1 b\rangle L_2) \cdot \bot).$$

Lemma D.4 (Replacing Forward Small-steps).

$$\forall v\ v_1\ r\ E\ w_1,\ ([], \bot) \xrightarrow{w_1}{}^{*} (v + [r], E)\ \wedge\ ([], \bot) \xrightarrow{w_1}{}^{*} (v_1 + [r], E)\ \Longrightarrow\ \forall v_2\ E_2\ w_2,\ ([], \bot) \xrightarrow{w_1 w_2}{}^{*} (v + [r] + v_2, E_2)\ \Longrightarrow\ ([], \bot) \xrightarrow{w_1 w_2}{}^{*} (v_1 + [r] + v_2, E_2).$$

Lemma D.5 (Splitting Forward Small-steps).

$$\forall v_1\ v_2\ E\ w,\ ([], \bot) \xrightarrow{w}{}^{*} (v_1 + v_2, E)\ \wedge\ v_1 \neq []\ \Longrightarrow \exists E_1\ w_1, ([], \bot) \xrightarrow{w_1}{}^{*} (v_1, E_1)\ \wedge\ \exists w_2, w = w_1 w_2.$$

The forward small-step relation is equivalent to the big-step relation under certain conditions; we state the equivalence in Lemma D.7. To prove the equivalence, we first introduce the invariant of the forward small-step relation in Lemma D.6, which splits a forward small-step relation into two parts based on the last unmatched rule and considers them separately.

Lemma D.6 (Invariant of the Forward Small-step Relation).

$$\begin{aligned}
&\forall v\ E\ w,\ ([], \bot) \xrightarrow{w}{}^{*} (v, E)\ \wedge\ w \neq \epsilon\ \Longrightarrow \\
&\quad (\mathrm{head}(E)\ \textit{is None or a pending rule}\ \wedge \\
&\qquad \forall \hat{w}\ \hat{v}, \mathrm{nextNT}\,(v) \Downarrow (\hat{w}, \hat{v})\ \Longrightarrow\ \mathrm{headNT}\,(v) \Downarrow (w\hat{w}, v + \hat{v}))\ \vee \\
&\quad \exists \langle a\ r_{\langle a}\ \hat{E},\ E = r_{\langle a} \cdot \hat{E}\ \wedge r_{\langle a}\ \textit{is a matching rule}\ \wedge \\
&\qquad \exists v_1\ v_2,\ v = v_1 + [r_{\langle a}] + v_2 \wedge \exists w_1\ w_2,\ w = w_1 \langle a w_2\ \wedge \\
&\qquad\quad \left(w_1 = \epsilon \wedge v_1 = [] \wedge \hat{E} = \bot\ \vee\ w_1 \neq \epsilon \wedge ([], \bot) \xrightarrow{w_1}{}^{*} (v_1, \hat{E})\right)\ \wedge \\
&\qquad\qquad (\forall \hat{w}\ \hat{v},\ \mathrm{nextNT}\,(v) \Downarrow (\hat{w}, \hat{v})\ \Longrightarrow\ \mathrm{nextNT}\,(r_{\langle a}) \Downarrow (w_2\hat{w}, v_2 + \hat{v}))\,.
\end{aligned}$$

Proof. Prove it by induction over the length of w. There are two cases of w: If $w = i \in \Sigma$, then the lemma is straightforward to show; so assume $w = w'i$ and $w' \neq \epsilon$. By the definition of the small-step relation, we have $\exists v'\ r,\ v = v' + [r]$ and $\exists E', ([], \bot) \xrightarrow{w'}{}^{*} (v', E')$; we apply the induction hypothesis to the small-step relation of w' and have two cases of E'.

In the first case, $E' = \bot$ or $\mathrm{head}(E')$ is a pending rule. There are two cases of i.

(1) $i \in \Sigma_{\mathrm{plain}} \cup \Sigma_{\mathrm{ret}}$. We have $E = E'$ when $i \in \Sigma_{\mathrm{plain}}$, and $E = \mathrm{tail}(E')$ when $i \in \Sigma_{\mathrm{ret}}$. In both cases, either $E = \bot$ or $\mathrm{head}(E)$ is a pending rule.[5] For any big-step $\mathrm{nextNT}\,(v) \Downarrow (\hat{w}, \hat{v})$, we extend it to $\mathrm{nextNT}\,(v') \Downarrow (i\hat{w}, [r] + \hat{v})$ by definition, and by the induction hypothesis, we have $\mathrm{headNT}\,(v) \Downarrow (w'i\hat{w}, v' + [r] + \hat{v})$.

(2) $i \in \Sigma_{\mathrm{call}}$. Then, we have $E = r \cdot E'$. There are two cases of r.

(a) If r is a pending rule, then for any big-step $\mathrm{nextNT}\,(v) \Downarrow (\hat{w}, \hat{v})$, we first extend it to $\mathrm{nextNT}\,(v') \Downarrow (i\hat{w}, [r] + \hat{v})$ by definition, then we have $\mathrm{headNT}\,(v) \Downarrow (w'i\hat{w}, v' + [r] + \hat{v})$ by the induction hypothesis.

(b) If r is a matching rule, then we construct $v_1 = v'$, $r_{\langle a} = r$, $v_2 = []$ and $w_1 = w'$, $\langle a = i$, $w_2 = \epsilon$. We already have $([], \bot) \xrightarrow{w_1}{}^{*} (v_1, E')$. Since both v_2 and w_2 are empty, $\mathrm{nextNT}\,(v) \Downarrow (\hat{w}, \hat{v})\ \Longrightarrow\ \mathrm{nextNT}\,(r_{\langle a}) \Downarrow (w_2\hat{w}, v_2 + \hat{v})$ is trivial to show.

In the second case, $\mathrm{head}(E')$ is a matching rule, and we have

$$\exists \langle a'\ v_1'\ r_{\langle a'}\ v_2'\ w_1'\ w_2',\ v' = v_1' + [r_{\langle a'}] + v_2'\ \wedge\ w' = w_1' \langle a' w_2'.$$

There are three cases of i.

(1) $i \in \Sigma_{\mathrm{plain}}$. Then, we have

$$v = v_1' + [r_{\langle a'}] + (v_2' + [r])\ \wedge\ w = w_1' \langle a' (w_2' i).$$

[5] Note that in VPGs, a pending rule cannot be used inside a matching rule.

We construct $v_1 = v_1'$, $r_{\langle a} = r_{\langle a'}$, $v_2 = v_2' + [r]$, and $w_1 = w_1'$, $\langle a = \langle a'$, $w_2 = w_2' i$. We only need to prove the last clause. For any big-step $\text{nextNT}\,(v) \Downarrow (\hat{w}, \hat{v})$, we extend it to $\text{nextNT}\,(v') \Downarrow (i\hat{w}, [r] + \hat{v})$ by definition, and by the induction hypothesis, we have $\text{nextNT}\,(r_{\langle a'}) \Downarrow \left(w_2' i\hat{w}, v_2' + [r] + v'\right)$.

(2) $i \in \Sigma_{\text{call}}$. Then, we have

$$v = v' + [r] \ \wedge \ w = w' i.$$

We construct $v_1 = v'$, $r_{\langle a} = r$, $v_2 = []$, and $w_1 = w'$, $\langle a = i$, $w_2 = \epsilon$. The rest of the proof is straightforward.

(3) $i \in \Sigma_{\text{ret}}$. Then, we have $E' = r_{\langle a'} \cdot E$. For any big-step $\text{nextNT}\,(v) \Downarrow (\hat{w}, \hat{v})$, we first construct the big-step $\text{headNT}\,(r_{\langle a'}) \Downarrow \left(\langle a' w_2' i\hat{w}, [r_{\langle a'}] + v_2' + [r] + \hat{v}\right)$ by definition. Now, there are two cases of E.

(a) $E = \bot$ or $\text{head}(E)$ is a pending rule. If $w_1' = \epsilon$, then we are done. Otherwise, we apply the induction hypothesis to the small-step relation of w_1' and use it to extend the big-step relation to

$$\text{headNT}\,(v) \Downarrow (w_1' \langle a' w_2' i\hat{w}, v_1' + [r_{\langle a'}] + v_2' + [r] + \hat{v}) \,.$$

(b) $\text{head}(E)$ is a matching rule. We apply the induction hypothesis to the small-step relation of w_1', and have

$$\exists \langle a'' \ v_1'' \ r_{\langle a''} \ v_2'' \ w_1'' \ w_2,'' \ v_1' = v_1'' + [r_{\langle a''}] + v_2'' \ \wedge \ w_1' = w_1'' \langle a'' w_2''.$$

So, we have

$$v = v_1'' + [r_{\langle a''}] + (v_2'' + [r_{\langle a'}] + v_2' + [r]) \ \wedge \ w = w_1'' \langle a'' (w_2'' \langle a' w_2' i).$$

By the induction hypothesis, we extend the big-step relation to

$$\text{nextNT}\,(r_{\langle a''}) \Downarrow (w_2'' \langle a' w_2' i\hat{w}, v_2'' + [r_{\langle a'}] + v' + [r] + \hat{v}) \,. \qquad \square$$

Lemma D.7 (Relating Small-step and Big-step Relations).

$$\forall L\ w\ v,\ w \neq \epsilon \implies$$

$$L \Downarrow (w, v) \iff \exists E,\ ([], \bot) \xrightarrow{w}{}^{*} (v, E) \ \wedge \ (\text{nextNT}\,(v) \to \epsilon) \in P \wedge$$
$$(L \in V^0 \implies E = \bot) \wedge \text{head}(E) \textit{ is None or a pending rule.}$$

Proof. $\implies$. Prove it by induction over the big-step relation $L \Downarrow (w, v)$.

(1) $w = iw'$, $v = [r] + v'$, and $\text{nextNT}\,(r) \Downarrow (w', v')$. The case of $w' = \epsilon$ is trivial, so assume $w' \neq \epsilon$. Apply the induction hypothesis to w' and we have

$$\exists E',\ ([], \bot) \xrightarrow{w'}{}^{*} (v', E') \ \wedge \ (\text{nextNT}\,(v') \to \epsilon) \in P \wedge$$
$$(\text{nextNT}\,(r) \in V^0 \implies E' = \bot) \ \wedge \ \text{head}(E') \text{ is None or a pending rule.}$$

There are two cases of i.

(a) $i \in \Sigma_{\text{call}}$. We construct the small-step relation $([], \bot) \xrightarrow{i}{}^{*} ([r], r \cdot \bot)$ by definition, and by Lemma D.2, we append v' to $[r]$ and have $([], \bot) \xrightarrow{iw'}{}^{*} ([r] + v', E' \cdot [r])$ or $([], \bot) \xrightarrow{iw'}{}^{*} ([r] + v', E')$.

(b) $i \in \Sigma_{\text{plain}} \cup \Sigma_{\text{ret}}$. We construct the small-step relation $([], \bot) \xrightarrow{i}{}^{*} ([r], \bot)$ by definition, and by Lemma D.1, we append v' to $[r]$ and have $([], \bot) \xrightarrow{iw'}{}^{*} ([r] + v', E')$.

(2) $w = \langle a w_1 b \rangle w_2$, $v = [r_1] + v_1 + [r_2] + v_2$, and $\text{nextNT}(r_i) \Downarrow (w_i, v_i)$, $i = 1, 2$. We first construct the small-step relation $([], \bot) \xrightarrow{\langle a}{}^* ([r_1], r_1 \cdot \bot)$. If $w_1 \neq \epsilon$, then we apply the induction hypothesis to w_1 and have the small-step relation $([], \bot) \xrightarrow{w_1}{}^* (v_1, \bot)$ (note that $\text{headNT}(v_1) \in V^0$). Then, by Lemma D.3, we have $([], \bot) \xrightarrow{\langle a w_1}{}^* ([r_1] + v_1, r_1 \cdot \bot)$; we extend it to $([], \bot) \xrightarrow{\langle a w_1 b \rangle}{}^* ([r_1] + v_1 + [r_2], \bot)$ by definition. Finally, if $w_2 \neq \epsilon$, then we apply the induction hypothesis to w_2 and have the small-step relation of w_2. By Lemma D.4, we have $([], \bot) \xrightarrow{w}{}^* ([r_1] + v_1 + [r_2] + v_2, \bot)$.

$\Longleftarrow$. This is a corollary of Lemma D.6. □

We describe the specification of the parser derivative functions in Lemma D.8.

Lemma D.8 (Specification of Parser Derivative Functions).

$$
\begin{aligned}
&\forall c\ m\ r_1\ r,\ (r_1, r) \in p_c(m) \iff \\
&\quad \exists r'\ L', (r_1, r') \in m \wedge r = (\text{nextNT}(r') \to c.L') \in \dot{P}; \\
&\forall \langle a\ m\ r_1\ r,\ (r_1, r) \in p_{\langle a}(m) \iff \\
&\quad \exists r_2\ r',\ (r_2, r') \in m \wedge \\
&\quad ((r_2 = \mathit{None} \vee \exists L_1\ \langle a_2\ L_2,\ r_2 = (L_1 \to \langle a_2.L_2) \in \dot{P}) \wedge \\
&\qquad (\exists L', r_1 = r = (\text{nextNT}(r') \to \langle a.L') \in \dot{P} \vee \\
&\qquad\quad \exists L'\ b\rangle\ L_3,\ r_1 = r = (\text{nextNT}(r') \to \langle a.L' b\rangle L_3) \in \dot{P}) \vee \\
&\quad \exists L_1\ \langle a_2\ L_2\ b_2\rangle\ L_3,\ r_2 = (L_1 \to \langle a_2.L_2 b_2\rangle L_3) \in \dot{P} \wedge \\
&\qquad \exists L'\ b\rangle\ L_4,\ r_1 = r = (L \to \langle a.L' b\rangle L_4) \in \dot{P}); \\
&\forall b\rangle\ m\ t\ r_1\ r,\ (r_1, r) \in p_{b\rangle}(m, t) \iff \\
&\quad (t = \mathit{None} \wedge r_1 = \mathit{None} \wedge \exists (r_2, r') \in m \wedge \exists L,\ r = (\text{nextNT}(r') \to b\rangle.L') \in \dot{P}) \vee \\
&\quad (\exists r_2\ r',\ (r_2, r') \in m \wedge \exists r'', (r_1, r'') \in t \wedge \text{nextNT}(r'') = \text{headNT}(r_2)\ \wedge \\
&\qquad (\exists L\ \langle a\ L_1\ L', r_2 = L \to \langle a.L_1 \in \dot{P} \wedge r = (\text{nextNT}(r') \to b\rangle.L') \in \dot{P} \vee \\
&\qquad\quad \exists \langle a\ L\ L_1\ L', r_2 = (L \to \langle a.L_1 b\rangle L') \in \dot{P} \wedge r = (L \to \langle a L_1 b\rangle.L') \in \dot{P} \wedge \\
&\qquad\qquad (\text{nextNT}(r') \to \epsilon) \in P)).
\end{aligned}
$$

Now, we give the invariants of the parser PDA. Lemma D.9 states that the forward small-step relation for input $w_1 \ldots w_i$ is represented in m_i, for all $i \in [1..|w|]$, where m_i is the parser PDA state after processing w_i. Lemma D.10 states that each pair in m_i represents a forward small-step relation. In these lemmas, we write $\mathcal{P}$ for the transitive closure of the parser transition function (Definition D.1) and use a helper function $\text{split}(v, E, w) = (v_1, v_2, w_1, w_2)$, which splits v into $v = v_1 + [r_{\langle a}] + v_2$ and w into $w = w_1 \langle a w_2$, where $r_{\langle a}$ is the rule that generates the last unmatched call symbol $\langle a$. Specially, if w does not include unmatched call symbols, then $\text{split}(v, E, w) = ([], v, [], w)$.

Definition D.1 (Transition Closure of the Parser PDA). Given a string w, we define the relation $\mathcal{P}(m_0, \bot, w) = (m, T)$, meaning that starting from $(m_0, \bot)$, running the parser PDA on w and the last configuration is (m, T).

Lemma D.9 (Invariants of the Parser PDA, Part 1).

$$\forall w\ m\ T,\ \mathcal{P}(m_0, \bot, w) = (m, T)\ \wedge\ ([], \bot) \xrightarrow{w}{}^{*} (v, E)\ \Longrightarrow$$
$$\forall v\ E\ v_1\ v_2\ w_1\ w_2,\ (v_1, v_2, w_1, w_2) = \text{split}(v, E, w)\ \Longrightarrow$$
$$(\text{head}(E), \text{lastRule}(v)) \in m\ \wedge$$
$$(\text{head}(E) = \mathit{None} \wedge T = \bot\ \vee$$
$$\text{head}(E) \neq \mathit{None} \wedge \exists t\ T', T = t :: T' \wedge \mathcal{P}(m_0, \bot, w_1) = (t, T')).$$

Proof. Prove it by induction over the length of w.

There are two cases of w: If $w = i \in \Sigma$, then we can check the lemma straightforwardly. Next, assume $w = w'i$ and $w' \neq \epsilon$. Then, we have $v = v' + [r]$. We invert $([], \bot) \xrightarrow{w} (v, E)$ and have $([], \bot) \xrightarrow{w'} (v', E')$. There are three cases of i.

(1) $i \in \Sigma_{\text{plain}}$. Then $E = E'$. By the induction hypothesis, we have $(\text{head}(E'), \text{lastRule}(v')) \in m'$, where $(m', T) = p_{w'}(m_0, \bot)$. By Lemma D.8, we have $(\text{head}(E), \text{lastRule}(v)) \in m$. If $\text{head}(E) = \text{None}$, i.e., $E = \bot$, we are done. Otherwise, let $\text{split}(v', E', w') = (v'_1, v'_2, w'_1, w'_2)$. Clearly, we have $v_1 = v'_1$ and $w_1 = w'_1$, so we have $\exists t\ T',\ T = t :: T' \wedge \mathcal{P}(m_0, \bot, w_1) = (t, T')$.
(2) $i \in \Sigma_{\text{call}}$. Then $E = r \cdot E'$, $w_1 = w'$ and $v_1 = v'$. By Lemma D.8, $(r, r) \in m$ and $\exists T',\ T = m' :: T' \wedge \mathcal{P}(m_0, \bot, w_1) = (m', T')$.
(3) $i \in \Sigma_{\text{ret}}$. Let $\text{split}(v', E', w') = (v'_1, v'_2, w'_1, w'_2)$. There are two cases of E'.
   (a) $E' = \bot$. Then, we have $E = \bot$, and by Lemma D.8 and the induction hypothesis, we have $(\text{None}, \text{lastRule}(v)) \in m$.
   (b) $\exists r_{\lessdot a}, E' = r_{\lessdot a} \cdot E$. Then, we have $r_{\lessdot a} \in E'$, and by Lemma D.8 and the induction hypothesis, we have $(\text{head}(E), \text{lastRule}(v)) \in m$. If $\text{head}(E) = \text{None}$, then we are done. Otherwise, we have $w'_1 \neq \epsilon$. We apply the induction hypothesis to w'_1 and finish the proof. □

Lemma D.10 (Invariants of the Parser PDA, Part 2).

$$\forall m\ T\ w,\ \mathcal{P}(m_0, \bot, w) = (m, T)\ \wedge\ w \neq \epsilon\ \Longrightarrow$$
$$\forall r_1\ r,\ (r_1, r) \in m\ \Longrightarrow$$
$$(r_1 = \mathit{None} \wedge \exists v,\ ([], \bot) \xrightarrow{w}{}^{*} (v + [r], \bot)\ \vee$$
$$r_1 \neq \mathit{None} \wedge \exists v\ H,\ ([], \bot) \xrightarrow{w}{}^{*} (v + [r], r_1 :: H)).$$

Proof. Prove it by induction over the length of w.

There are two cases of w: If $w = i \in \Sigma$, then we can check the lemma straightforwardly. Next, assume $w = w'i$ and $w' \neq \epsilon$. Let $\mathcal{P}(m_0, \bot, w') = (m', T')$.

There are three cases of i.

(1) $i \in \Sigma_{\text{plain}}$. For each (r_1, r) in m, we have $\exists r', (r_1, r') \in m'$ and $\text{nextNT}(r') = \text{headNT}(r)$. The lemma is then directly from the definition and the induction hypothesis applied to w'.
(2) $i \in \Sigma_{\text{call}}$. For each (r_1, r) in m, we have $\exists r', (r_2, r') \in m'$ and $\text{nextNT}(r') = \text{headNT}(r)$. By Lemma D.8, $r_1 = r$ and $(r_1, r_1) \in m$; by the induction hypothesis applied to w', $\exists H, ([], \bot) \xrightarrow{w}{}^{*} (v' + [r_1], r_1 :: H)$.
(3) $i \in \Sigma_{\text{ret}}$. There are two cases of r_1.
   (a) $r_1 = \text{None}$. Then $T = \bot$ and by Lemma D.8 and the induction hypothesis, we have $([], \bot) \xrightarrow{w}{}^{*} (v + [r], \bot)$.

(b) $r_1 \neq \text{None}$. Then $\exists t,\ T = t \cdot T',\ \exists r_3, (r_1, r_3) \in t,\ \exists r_4\ r_5, (r_4, r_5) \in m'$, and $\text{nextNT}(r_3) = \text{headNT}(r_4)$. By Lemma D.8, we have $\exists H, ([], \bot) \xrightarrow{w}{}^{*} (v' + [r], r_1 :: H)$. □

Our next step is to introduce the invariants of extraction functions, which build parse trees in a backward way. To formalize the parse trees built by the backward extraction process, we introduce the *backward small-step relation* in Figure 14. This relation is similar to the forward one, only in the reverse direction. The corresponding transitive closure is defined in Figure 15.

$$\frac{(L \to iL_1) \in P \quad i \in \Sigma_{\text{call}} \cup \Sigma_{\text{plain}} \quad L_1 \to \epsilon}{([], \bot) \overset{i}{\rightsquigarrow} ([L \to i.L_1], \bot)}$$

$$\frac{(L \to cL_1) \in P \quad \text{firstRule}(v) = (L_1 \to i.L_2) \text{ or } L_1 \to (\langle a.L_2 b \rangle L_3) \quad i \in \Sigma}{(v, T) \overset{c}{\rightsquigarrow} ((L \to c.L_1) :: v, T)}$$

$$\frac{(L \to cL_1) \in P \quad L_1 \to \epsilon \quad \text{firstRule}(v) = (L_2 \to \langle aL_3 b \rangle .L_4)}{(v, T) \overset{c}{\rightsquigarrow} ((L \to c.L_1) :: v, T)}$$

$$\frac{(L \to \langle aL_1) \in P \quad T = \bot \vee \text{head}(T) = (L_1 \to b \rangle .L_2)}{(v, T) \overset{\langle a}{\rightsquigarrow} ((L \to \langle a.L_1) :: v, \text{tail}(T))}$$

$$\frac{\text{firstRule}(v) = (L_1 \to \langle aL_2 b \rangle .L_3) \quad (L_2 \to \epsilon) \in P}{(v, T) \overset{\langle a}{\rightsquigarrow} ((L_1 \to \langle a.L_2 b \rangle L_3) :: v, \text{tail}(T))}$$

$$\frac{\text{firstRule}(v) = (L_2 \to c.L_3) \text{ or } (L_2 \to \langle a_2 .L_4 b_2 \rangle L_5) \quad \text{head}(T) = (L_1 \to \langle aL_2 b \rangle .L_3)}{(v, T) \overset{\langle a}{\rightsquigarrow} ((L_1 \to \langle a.L_2 b \rangle L_3) :: v, \text{tail}(T))}$$

$$\frac{r \in \dot{P} \quad r = (L_1 \to b \rangle .L_2) \text{ or } (L \to \langle aL_1 b \rangle .L_2) \quad (L_2 \to \epsilon) \in P}{([], \bot) \overset{b \rangle}{\rightsquigarrow} ([r], r \cdot \bot)}$$

$$\frac{r = (L \to \langle aL_1 b \rangle .L_2) \in \dot{P} \quad (L_2 \to \epsilon) \in P \quad \text{firstRule}(v) = L_3 \to \langle a_2 L_4 b_2 \rangle .L_5}{(v, T) \overset{b \rangle}{\rightsquigarrow} (r :: v, r \cdot T)}$$

$$\frac{\begin{array}{ll}(T = \bot \vee \text{head}(T) = L_3 \to b_2 \rangle .L_4) & r \in \dot{P} \\ r = (L_1 \to b \rangle .L_2) \text{ or } (L \to \langle aL_1 b \rangle .L_2) & \text{headNT}(v) = L_2\end{array}}{(v, T) \overset{b \rangle}{\rightsquigarrow} (r :: v, r \cdot T)}$$

$$\frac{\begin{array}{ll}\text{head}(T) = L_3 \to \langle a_2 L_4 b_2 \rangle .L_5 & r = (L \to \langle aL_1 b \rangle .L_2) \in \dot{P} \\ \text{firstRule}(v) = (L_2 \to c.L_6) \text{ or } (L_2 \to \langle a_3 .L_6 b_3 \rangle L_7) & \end{array}}{(v, T) \overset{b \rangle}{\rightsquigarrow} (r :: v, r \cdot T)}$$

Fig. 14. The backward small-step parse-tree derivation, given a VPG $G = (\Sigma, V, P, L_0)$.

$$\frac{}{(v,E) \overset{\epsilon}{\rightsquigarrow}^* (v,E)} \qquad \frac{(v_1,E_1) \overset{w}{\rightsquigarrow}^* (v_2,E_2) \quad (v_2,E_2) \overset{i}{\rightsquigarrow} (v_3,E_3)}{(v_1,E_1) \overset{iw}{\rightsquigarrow}^* (v_3,E_3)}$$

Fig. 15. The transitive closure of the backward small-step relation.

Similar to the forward small-step relation, the backward small-step relation is equivalent to the big-step relation under some conditions. We state the equivalence in Lemma D.13. To prove the lemma, we first give the invariants of the backward small-step relation in Lemma D.11.

Lemma D.11 (Invariants of the Backward Small-step Relation).

$$\begin{aligned}
&\forall v\ E\ w,\ ([],\bot) \overset{w}{\rightsquigarrow}^* (v,E) \implies \\
&\quad \mathrm{head}(E)\ \textit{is None or a pending rule} \ \wedge\ \mathrm{headNT}(v) \Downarrow (w,v)\ \vee \\
&\quad \exists L_1\ \langle a\ L_2\ b\rangle\ L_3\ r\ E',\ E = r \cdot E' \wedge r = (L_1 \to \langle aL_2b\rangle.L_3) \in P \wedge \\
&\quad (v = [r] \wedge w = b\rangle\ \wedge\ (L_3 \to \epsilon) \in P \vee \\
&\qquad \exists v_2\ w_2,\ v = [r] + v_2 \wedge w = b\rangle w_2\ \wedge ([],\bot) \overset{w_2}{\rightsquigarrow}^* (v_2,E') \wedge \\
&\qquad\quad ((\exists L_4\ \langle a_2\ L_5\ b_2\rangle\ L_6\ w_3\ v_3, \\
&\qquad v_2 = [L_4 \to \langle a_2L_5b_2\rangle.L_6] + v_3 \wedge w_2 = b_2\rangle w_3\ \wedge\ (L_3 \to \epsilon \in P)) \vee \\
&\qquad\qquad (\mathrm{firstRule}(v_2)\ \textit{is not a matching return rule} \wedge L_3 = \mathrm{headNT}(v_2))) \vee \\
&\qquad \exists v_1\ w_1, v = v_1 + [r] \wedge w = w_1b\rangle \wedge\ \mathrm{headNT}(v_1) \Downarrow (w_1,v_1) \wedge (L_3 \to \epsilon) \in P \wedge E' = \bot \wedge \\
&\qquad\quad \mathrm{firstRule}(v_1)\ \textit{is not a matching return rule} \wedge \\
&\qquad\quad (\forall \hat{v}\ \hat{E}\ \hat{w},\ ([],\bot) \overset{\hat{w}}{\longrightarrow}^* (\hat{v},\hat{E}) \implies \mathrm{nextNT}(\hat{v}) = \mathrm{headNT}(v_1) \implies \\
&\qquad\qquad ([],\bot) \overset{\hat{w}w_1}{\longrightarrow}^* (\hat{v} + v_1,\hat{E}))\ \vee \\
&\qquad \exists v_1\ v_2\ w_1\ w_2,\ v = v_1 + [r] + v_2\ \wedge w = w_1b\rangle w_2 \wedge \\
&\qquad\quad \mathrm{firstRule}(v_1)\ \textit{is not a matching return rule} \wedge \\
&\qquad\quad \mathrm{headNT}(v_1) \Downarrow (w_1,v_1)\ \wedge\ ([],\bot) \overset{w_2}{\longrightarrow}^* (v_2,E') \wedge \\
&\qquad\quad (\forall \hat{v}\ \hat{E}\ \hat{w},\ ([],\bot) \overset{\hat{w}}{\longrightarrow}^* (\hat{v},\hat{E}) \implies \mathrm{nextNT}(\hat{v}) = \mathrm{headNT}(v_1) \implies \\
&\qquad\qquad ([],\bot) \overset{\hat{w}w_1}{\longrightarrow}^* (\hat{v} + v_1,\hat{E})) \wedge \\
&\qquad\quad (\exists L_4\ a_2\ L_5\ b_2\rangle\ L_6\ v_3\ w_3, v_2 = [L_4 \to \langle a_2L_5b_2\rangle.L_6] + v_3 \wedge \\
&\qquad\qquad w_2 = b_2\rangle w_3 \wedge (L_3 \to \epsilon) \in P \vee \\
&\qquad\qquad \mathrm{firstRule}(v_2)\ \textit{is not a matching return rule}\ \wedge\ L_3 = \mathrm{headNT}(v_2)))).
\end{aligned}$$

Proof. Prove it by induction over the length of w.

There are two cases of w: If $w = i \in \Sigma$, then we can prove the lemma straightforwardly; so assume $w = iw'$ and $w' \neq \epsilon$ and we have $v = [e] + v'$. From $([],\bot) \overset{w}{\rightsquigarrow}^* (v,E)$, we have $\exists E',\ ([],\bot) \overset{w'}{\rightsquigarrow}^* (v',E')$. There are three cases of i.

(1) $i \in \Sigma_{\mathrm{plain}}$. Then, we have $E = E'$. If $\mathrm{head}(E)$ is None or a pending rule, then we prove the lemma by extending $\mathrm{headNT}(v') \Downarrow (w',v')$ to $\mathrm{headNT}(v) \Downarrow (iw',[e] + v')$. So, assume

head(E) is a matching rule. Let $v' = v_1 + [r] + v_2$, we prove the lemma by extending headNT $(v_1) \Downarrow (w_1, v_1)$ to headNT $(v) \Downarrow (iw_1, [r] + v_1)$.

(2) $i \in \Sigma_{\text{call}}$. If head($E$) is None or a pending rule, then we prove the lemma by extending the big-step relation of v' to headNT $(v) \Downarrow (iw', [e] + v')$. So, assume head($E$) is a matching rule. Let $v' = v'_1 + [r'] + v'_2$ and $w' = w'_1 b' \rangle w'_2$. By induction, we have $\exists E_2,\ ([], \bot) \overset{w'_2}{\rightsquigarrow} (v'_2, E_2)$. If head($E$) is None or a pending rule, then we can directly construct headNT $(v) \Downarrow \left(w, [e] + v'_1 + [r'] + v'_2\right)$. If head($E$) is a matching rule, then we have $v'_2 = v''_1 + [r''] + v''_2$ and $w'_2 = w''_1 b'' \rangle w''_2$. We prove the lemma by combining the big-step of v'_1 and v''_1 to headNT $(v) \Downarrow \left(iw'_1 b' \rangle w''_1, [e] + v'_1 + [r'] + v''_1\right)$.

(3) $i \in \Sigma_{\text{ret}}$. This case is trivial. □

Just as the case for the forward small-step relation, we can combine two backward small-step relations. We state it in Lemma D.12.

Lemma D.12 (Combining Backward Small-step Relations).

$$\begin{aligned}
&\forall L\ w\ v,\ L \Downarrow (w, v) \\
&\quad \Longrightarrow\ (L \in V^0\ \wedge\ w \neq \epsilon) \\
&\quad \Longrightarrow\ ([], \bot) \overset{w}{\rightsquigarrow}{}^{*}\ (v, \bot)\ \wedge \\
&\qquad (\forall v_2\ E_2\ w_2\ L_1\ \langle a\ L_2\ b \rangle\ L_3, ([], \bot) \overset{w_2}{\rightsquigarrow}{}^{*}\ ([L_1 \to \langle aL_2b \rangle . L_3] + v_2, E_2) \\
&\qquad\quad \Longrightarrow\ ([], \bot) \overset{ww_2}{\rightsquigarrow}{}^{*}\ (v + [L_1 \to \langle aL_2b \rangle . L_3] + v_2, E_2)).
\end{aligned}$$

Proof. Prove it by induction over the big-step relation $L \Downarrow (w, v)$. Since $L \in V^0$, there are two cases.

(1) $w = cw'$ and $v = [r] + v'$. If $w' = \epsilon$, then we can prove the lemma straightforwardly; so, assume $w' \neq \epsilon$ and apply the induction hypothesis to headNT $(v') \Downarrow (w', v')$. We can directly extend the backward small-step relation of v' with $[r]$. For any $([], \bot) \overset{w_2}{\rightsquigarrow}{}^{*} ([L_1 \to \langle aL_2b \rangle . L_3] + v_2, E_2)$, we first extend it with the small-step relation of v' by the induction hypothesis, then further extend the small-step relation with $[r]$ by definition.

(2) $w = \langle a' w'_1 b' \rangle w'_2$ and $v = [r_{\langle a'}] + v'_1 + [r_{b' \rangle}] + v'_2$. Note that, since $L \in V^0$, we have headNT $\left(v'_2\right) \in V^0$. For any $([], \bot) \overset{w_2}{\rightsquigarrow}{}^{*} ([L_1 \to \langle aL_2b \rangle . L_3] + v_2, E_2)$, by the induction hypothesis of v'_2 and the definition of the backward small-step, we extend it with v'_2, $r_{b' \rangle}$, v'_1, and $r_{\langle a'}$ in order. □

Lemma D.13 (Correctness of the Backward Small-step Relation).

$$\forall L\ w\ v,\ L \Downarrow (w, v) \iff \exists E,\ ([], \bot) \overset{w}{\rightsquigarrow}{}^{*}\ (v, E) \wedge \text{head}(E) \textit{ is None or a pending rule.}$$

Proof. $\Longrightarrow$. Prove it by induction over the big-step relation $L \Downarrow (w, v)$.

(1) $w = iw'$, $v = [r] + v'$, and nextNT $(r) \Downarrow (w', v')$, where r is not a matching rule. The case of $w' = \epsilon$ is trivial; so, assume $w' \neq \epsilon$. Apply the induction hypothesis to w' and we have $\exists E',\ ([], \bot) \overset{w'}{\rightsquigarrow}{}^{*} (v', E')$. We can directly extend this backward small-step relation with $[r]$.

(2) $w = \langle aw_1 b \rangle w_2$, $v = [r_1] + v_1 + [r_2] + v_2$, and nextNT $(r_i) \Downarrow (w_i, v_i)$, $i = 1, 2$. If $w_2 \neq \epsilon$, then we apply the induction hypothesis to headNT $(v_2) \Downarrow (w_2, v_2)$ and have the small-step relation of

w_2. We extend it to $([], \bot) \overset{b\rangle w_2}{\rightsquigarrow}{}^* ([r_2] + v_2, r_2 \cdot \bot)$ by definition. If $w_2 = \epsilon$, then we have $v_2 = []$ and this small-step also holds. Then, if $w_1 \neq \epsilon$, then we apply the induction hypothesis to headNT $(v_1) \Downarrow (w_1, v_1)$ and have the small-step relation of w_1. By Lemma D.12, we construct $([], \bot) \overset{w_1 b\rangle w_2}{\rightsquigarrow}{}^* (v_1 + [r_2] + v_2, \bot)$. If $w_1 = \epsilon$, then this small-step also holds. Finally, we extend the small-step relation to $([], \bot) \overset{w}{\rightsquigarrow}{}^* ([r_1] + v_1 + [r_2] + v_2, \bot)$ by definition.

$\Longleftarrow$. This is a corollary of Lemma D.11. □

We give the specification of $\text{extract}_{\text{init}}$ and $\text{extract}_{\text{oneStep}}$ in Lemmas D.14, D.15, and D.16. The proofs are directly based on the definitions, and we omit them here.

Lemma D.14 (The Specification of $\text{extract}_{\text{init}}$).

$$\forall m\ M\ w\ i,\ \text{forest}(wi) = M + [m] \implies \forall v\ E, (v, E) \in \text{extract}_{\text{init}}(m) \iff ([], \bot) \overset{i}{\rightsquigarrow} (v, E) \wedge \exists r,\ v = [r] \wedge ((None, r) \in m \vee \exists L_1\ \langle a\ L_2,\ (L_1 \to \langle a.L_2, r) \in m).$$

Lemma D.15 shows that if m is a state and V includes partial parse trees in backward small-step relations, then $\text{extract}_{\text{oneStep}}$ extends the relations; Lemma D.16 further shows that the relations are extended by rules in m and trees in V.

Lemma D.15 (The Specification of $\text{extract}_{\text{oneStep}}$, Part I).

$$\forall m\ V\ i\ w,\ (\exists M\ w',\ \text{forest}(w'i) = M + [m]) \wedge \left(\forall \hat{v}\ \hat{E},\ (\hat{v}, \hat{E}) \in V \implies ([], \bot) \overset{w}{\rightsquigarrow}{}^* (\hat{v}, \hat{E})\right) \implies \forall v\ E,\ (v, E) \in (\text{extract}_{\text{oneStep}}(m, V)) \implies ([], \bot) \overset{iw}{\rightsquigarrow}{}^* (v, E).$$

Lemma D.16 (The Specification of $\text{extract}_{\text{oneStep}}$, Part II).

$$\begin{aligned}&\forall m\ V\ v\ E,\ (\exists M\ w',\ \text{forest}(w'i) = M + [m]) \wedge (v, E) \in \text{extract}_{\text{oneStep}}(m, V) \implies \\ &\quad \exists v'\ E',\ (v', E') \in V \wedge \exists r, (r, \text{firstRule}(v)) \in m \wedge \\ &\quad\quad ((E' = \bot \wedge (r = None \vee \exists L_1\ \langle a\ L_2, r = (L_1 \to \langle a.L_2))) \vee \\ &\quad\quad\quad \exists L_1'\ b\rangle\ L_2'\ E'',\ \text{head}(E') = (L_1' \to b\rangle.L_2') \wedge \\ &\quad\quad\quad\quad (r = None \vee \exists L_1\ \langle a\ L_2,\ r = (L_1 \to \langle a.L_2)) \vee \\ &\quad\quad\quad \exists L_1\ \langle a\ L_2\ b\rangle\ L_3,\ \text{head}(E) = (L_1 \to \langle aL_2b\rangle.L_3) \wedge r = (L_1 \to \langle a.L_2b\rangle L_3)).\end{aligned}$$

Now, we introduce the invariants of the extraction function.

Lemma D.17 (Invariants of the Extraction Function). *Given a VPG $G = (\Sigma, V, P, L_0)$, we have*

$$\begin{aligned}&\forall m\ M_1\ M_2\ w_1\ w_2\ i\ V, \\ &\quad \text{forest}(w_1w_2i) = M_1 + M_2 + [m] \wedge |M_2| = |w_2| \wedge V = \text{extract}(M_2 + [m]) \implies \\ &\quad\quad \forall v\ E,\ (v, E) \in V \iff ([], \bot) \overset{w_2 i}{\rightsquigarrow}{}^* (v, E) \wedge \\ &\quad\quad\quad (w_1 = \epsilon \wedge L_0 \Downarrow (w_2 i, v) \vee \exists v_1\ E_1,\ \text{headNT}(v_1) = L_0 \wedge \\ &\quad\quad\quad\quad ([], \bot) \overset{w_1}{\longrightarrow}{}^* (v_1, E_1) \wedge L_0 \Downarrow (w_1w_2i, v_1 + v)).\end{aligned}$$

Proof. Prove it by induction over w_2.

When $w_2 = \epsilon$, we have $V = \text{extract}_{\text{init}}(m)$.

$\Longrightarrow$. For each $(v, E) \in \text{extract}_{\text{init}}(m)$, by Lemma D.14, we have $([], \bot) \overset{i}{\rightsquigarrow} (v, E)$ and $\exists r, v = [r]$. By Lemma D.10, we have

$$\exists v_1\ E_1, ([], \bot) \overset{w_1 w_2 i}{\longrightarrow}{}^{*} (v_1 + [r], E_1).$$

By Lemmas D.7 and D.14, the above small-step relation can be converted to a big-step relation; i.e., $L_0 \Downarrow (w_1 w_2 i, v_1 + v)$. If $w_1 = \epsilon$, then $v_1 = []$ and $w_1 w_2 i = i$, so $L_0 \Downarrow (i, v)$. If $w_1 \neq \epsilon$, then we can invert the above small-step relation to get $\exists E_1,\ ([], \bot) \overset{w_1}{\longrightarrow}{}^{*} (v_1, E_1)$.

$\Longleftarrow$. For (v, E) that satisfy $([], \bot) \overset{w_1 w_2 i}{\rightsquigarrow}{}^{*} (v, E)$, if $w_1 = \epsilon$ and $L_0 \Downarrow (w_2 i, v)$, then we first convert the big-step relation to a small-step relation by Lemma D.7, then, by Lemma D.9, we have $(\text{head}(E), v) \in m$. Then, by Lemma D.14, we have $(v, E) \in V$. If $w_1 \neq \epsilon$ and $L_0 \Downarrow (w_1 i, v_1 + v)$, then we can similarly show that $(\text{head}(E), v) \in m$ and $(v, E) \in V$.

When $w_2 \neq \epsilon$, let $w_2 = i' w_3$ and $M_2 = [m_2'] + M_2'$, we have $V = \text{extract}_{\text{oneStep}}(m_2', \text{extract}(M_2' + [m]))$.

$\Longrightarrow$. For each $(v, E) \in V$, by Lemma D.16, we have

$$\exists r\ v'\ r_1\ E', v = [r] + v' \wedge (r_1, r) \in m_2' \wedge (v', E') \in \text{extract}(M_2' + [m]).$$

By Lemma D.10, we have

$$\exists v_r\ E_r,\ ([], \bot) \overset{w_1 i'}{\longrightarrow}{}^{*} (v_r + [r], E_r).$$

By the induction hypothesis, we have

$$([], \bot) \overset{w_3 i}{\rightsquigarrow}{}^{*} (v', E') \wedge \exists v_1\ E_1,\ ([], \bot) \overset{w_1 i'}{\longrightarrow}{}^{*} (v_1, E_1) \wedge L_0 \Downarrow (w_1 i' w_3 i, v_1 + v')\,.$$

As an overview, we will try to use v_r and v_1 to construct the counterpart of v. In the simplest case, $r_1 = \text{None}$ and v_r can be directly used. In a more complex case, r_1 is a matching rule, and we need to combine v_r and v_1 to construct the counterpart. There are two cases of r_1.

(1) $r_1 = \text{None}$ or a pending rule. We will prove $L_0 \Downarrow (w, v_r + v)$. We first convert the backward small-step $([], \bot) \overset{w_3 i}{\rightsquigarrow}{}^{*} (v', E')$ to a forward small-step. There are two cases of E'.
   (a) $E' = \bot$ or $\text{head}(E')$ is a pending rule. We convert $([], \bot) \overset{w_3 i}{\rightsquigarrow}{}^{*} (v', E')$ to a big-step by Lemma D.13 and then convert the big-step to a forward small-step by Lemma D.7.
   (b) $\text{head}(E')$ is a matching rule. This is not possible by Lemma D.16.

   Then, we combine $v_r + [r]$ and v' to $v_r + [r] + v'$ by Lemma D.1. Finally, we convert $v_r + [r] + v'$ to a big-step by Lemma D.7 and finish the proof.

(2) r_1 is a matching rule. There are two cases of E'.
   (a) $E' = \bot$ or $\text{head}(E')$ is a pending rule. This is not possible by Lemma D.16.
   (b) $\text{head}(E')$ is a matching rule. We use the split function to split v_1 into $v_{1,1} + [r_{1,\langle a}] + v_{1,2}$ and split $v_r + [r]$ into $v_{r,1} + [r_{r,\langle a}] + v_{r,2}$. Then, we construct a new forward small-step $([], \bot) \overset{w_1 w_2 i}{\rightarrow} (v_{1,1} + [r_{1,\langle a}] + v_{r,2}, E'')$ for some E''. Then, we extend this small-step with $\text{firstRule}(v')$ and use Lemma D.4 to further extend the small-step relation to $v_{1,1} + [r_{1,\langle a}] + v_{r,2} + v'$. Finally, we convert the small-step relation to a big-step by Lemma D.7.

$\Longleftarrow$. For each (v, E) that satisfies $([], \bot) \overset{w_1 w_2 i}{\rightsquigarrow}{}^{*} (v, E)$, let $v = [r] + v'$ for some v' and r, and $w_2 = i' w_3$ for some i' and w_3. We invert the backward small-step of $w_2 i$ and have $([], \bot) \overset{w_3 i}{\rightsquigarrow}{}^{*} (v', E')$

for some E'. Then, we convert $L_0 \Downarrow (w_1 w_2 i, v_1 + v)$ to a forward small-step $([], \bot) \xrightarrow{w_1 w_2 i}{}^{*} (v_1 + v, E)$ for some E, and by Lemma D.5, we have the forward small-step $([], \bot) \xrightarrow{w_1 i'}{}^{*} (v_1 + [r], E_r)$ for some E_r. With these small-steps, we apply the induction hypothesis and have $(v', E') \in \text{extract}(M_2' + [m])$, where $M_2 = [m_2'] + M_2'$ for some m_2'. Then, by the definition of $\text{extract}_{\text{oneStep}}$, we have $(v, E) \in V$. □

By Lemma D.17, when $w_1 = \epsilon$, we have the correctness of the parser generator.

Theorem D.18 (Correctness of the Parser Generator).

$$\forall w\ V,\ V = \text{extract}(\text{forest}(w)) \implies (\forall v,\ L_0 \Downarrow (w, v) \iff \exists E, (v, E) \in V).$$

E CORRESPONDENCE BETWEEN THE THEOREMS AND THE COQ PROOFS

Table 6 shows the theorems discussed in this article, the files that include the corresponding Coq theorems and their proofs, and the names of the Coq theorems.

Table 6. The Correspondence between the Theorems and the Mechanized Proofs

Theorem	File	Name
Theorem D.1	ForwardSmallStep	L4_1
Theorem D.2	ForwardSmallStep	L4_3
Theorem D.3	ForwardSmallStep	L4_4
Theorem D.4	ForwardSmallStep	L_DF_Local
Theorem D.5	ForwardSmallStep	DF_SPLIT2
Theorem D.6	ForwardSmallStep	L4_2
Theorem D.7	ForwardSmallStep	SoundV
Theorem D.8	GenForest	L_m2PlainM, L_m2CallM, L_m2RetM
Theorem D.9	GenForest	PForest1
Theorem D.10	GenForest	PForest2
Theorem D.11	BackwardSmallStep	BreakDV
Theorem D.12	BackwardSmallStep	L4_3
Theorem D.13	BackwardSmallStep	CompleteM, SoundV
Theorem D.14	Transducer	L_f_init
Theorem D.15	Transducer	L_f_b
Theorem D.16	Transducer	pg2
Theorem D.17	Transducer	L_extract
Theorems 6.1, Theorems 6.3	TimedExtraction	Property_VPG_Parser_Generator
Theorem 6.2	TimedRunPDA	bound_cost_run_PDA

F EXAMPLES OF TAGGED CFGS

The following is a tagged CFG for JSON [3], where nonterminals start with lowercase characters, such as `json`, and terminals start with uppercase characters, such as "STRING." Also, call and return symbols are tagged with "<" or ">", respectively. The declarations of terminals are omitted.

```
json = value ;
obj = <'{' pair (',' pair)* '}'> | <'{' '}'> ;
pair = STRING ':' value ;
arr = <'[' value (',' value)* ']'> | <'[' ']'> ;
value = STRING | NUMBER | obj | arr | 'true' | 'false' | 'null' ;
```

The following is a tagged CFG for HTML, which is adapted from the HTML grammar from the repository of ANTLR [3]:

```
htmlDocument = scriptletOrSeaWs* XML? scriptletOrSeaWs* DTD?
    scriptletOrSeaWs* htmlElements* ;
scriptletOrSeaWs = SCRIPTLET | SEA_WS ;
htmlElements = htmlMisc* htmlElement htmlMisc* ;
htmlElement = TagOpen | <TagOpen htmlContent TagClose>
    | TagSingle | SCRIPTLET | script | style ;
htmlContent = htmlChardata?
    ((htmlElement | CDATA | htmlComment) htmlChardata?)* ;
htmlAttribute = TAG_NAME (TAG_EQUALS ATTVALUE_VALUE)? ;
htmlChardata = HTML_TEXT | SEA_WS ;
htmlMisc = htmlComment | SEA_WS ;
htmlComment = HTML_COMMENT | HTML_CONDITIONAL_COMMENT ;
script = SCRIPT_OPEN (SCRIPT_BODY | SCRIPT_SHORT_BODY) ;
style = STYLE_OPEN (STYLE_BODY | STYLE_SHORT_BODY) ;
htmlElement =
    | <TAG_OPEN_h1 htmlElement* TAG_CLOSE_h1>
    | <TAG_OPEN_h2 htmlElement* TAG_CLOSE_h2>
    | <TAG_OPEN_h3 htmlElement* TAG_CLOSE_h3>
    | <TAG_OPEN_h4 htmlElement* TAG_CLOSE_h4>
    | <TAG_OPEN_div htmlElement* TAG_CLOSE_div>
    | <TAG_OPEN_b htmlElement* TAG_CLOSE_b>
    | <TAG_OPEN_i htmlElement* TAG_CLOSE_i>
    | <TAG_OPEN_ul htmlElement* TAG_CLOSE_ul>
    | <TAG_OPEN_ol htmlElement* TAG_CLOSE_ol>
    | <TAG_OPEN_table htmlElement* TAG_CLOSE_table>
    | TAG_OPEN
    | TAG_CLOSE
    | TAG_SCLOSE
    | CDATA
    | SCRIPTLET
    | htmlChardata
    | htmlComment
    | script
    | style
```

The following is a tagged CFG for XML adapted from ANTLR [3]:

```
document  = prolog? misc* element misc*;
prolog    = XMLDeclOpen attribute* SPECIAL_CLOSE ;
content   = chardata? ((element | reference | CDATA | PI | COMMENT)
              chardata?)* ;
element   = < OpenTag content CloseTag > |   SingleTag ;
reference = EntityRef | CharRef ;
attribute = Name '=' STRING ;
chardata  = TEXT | SEA_WS ;
misc      = COMMENT | PI | SEA_WS ;
```

G THE TRANSLATION ALGORITHM

Translating simple forms to linear forms. The translation algorithm is based on the dependency graph G. We start by removing from G any dependency edge (L, L') that is labeled with (s_1, s_2); intuitively, a matched token $\langle aL'b\rangle$ can be conceptually viewed as a "plain symbol" and its presence does not affect the following translation. We use the symbol t for either a plain symbol or a matched token. Further, a rule whose head is L is called *a rule of* L.

We find all **strongly connected components (SCCs)** in the dependency graph G and collapse each SCC into a single node; the resulting *condensation graph* is written as G^{SCC}, which is a **directed acyclic graph (DAG)**. For a nonterminal L, we write $G^{SCC}[L]$ for the nonterminals in its SCC. Because the validator passes G, we know that any edge between nonterminals L_1 and L_2 in $G^{SCC}[L]$ must be labeled with (s, ϵ).

We write allDep(G^{SCC}, L) for all nonterminals that L may (transitively) depend on according to G^{SCC}; that is,

$$\text{allDep}(G^{SCC}, L) = \{L' \mid G^{SCC}[L] \text{ transitively depends on } G^{SCC}[L']\}.$$

The translation algorithm maintains a set of processed nonterminals V_{done}, a map T, and a set P of the current set of rules. Set V_{done} is used to track the set of nonterminals whose rules have been processed, meaning those rules are in the linear form; it is initialized to be the empty set. The algorithm uses T to keep track of new nonterminals created during translation. It is initialized to the empty map. During translation, a new nonterminal L_1 may be created for a string of the form Ls, where L is a nonterminal and $s \in (\Sigma \cup V)^+$; the algorithm then adds the mapping (Ls, L_1) to T. Finally, P is initialized to be the set of rules in the input grammar; during translation, new rules may be added to P and some rules may be removed.

At each iteration, if V_{done} contains all nonterminals in G, then the algorithm terminates (which implies that all rules are already in linear forms). Otherwise, pick a nonterminal $L \notin V_{\text{done}}$ so allDep$(G^{SCC}, L) \subseteq V_{\text{done}}$; that is, nonterminals that $G^{SCC}[L]$ depend on (transitively) have only linear-form rules. Since G^{SCC} is a DAG, we can always find such an L.

If all rules of L are in the linear form, then add L to V_{done} and move to the next iteration. Otherwise, pick a non-linear form rule of $L \to t_1 \dots t_k L_1 s$, where each t_i is either a plain symbol or a matched token, L_1 is the left-most nonterminal (outside any matched token) in the rule, and $s \in (\Sigma \cup V)^+$. We must have that $L_1 \notin G^{SCC}[L]$, since otherwise it would violate the conditions imposed by the validator. So the rules of L_1 (as well as rules of those nonterminals that L_1 transitively depends on) must already be in the linear form.

With the non-linear form rule $L \to t_1 \dots t_k L_1 s$, the algorithm performs multiple rounds to transform it to a set of linear-form rules.

- The goal of the first round is to invoke linearizeFirstNT(L $\to$ $t_1 \dots t_k$L$_1$s), defined in Algorithm 5, to remove the dependency from L to L_1. When $k > 0$, what the procedure does is to introduce a new nonterminal L_1' (if $L_1 s$ is not in the translation table already), change the rule to $L \to t_1 \dots t_k L_1'$, and add the map $(L_1 s, L_1')$ to the translation table T; if further recursively invokes linearizeFirstNT(L$_1'$ $\to$ L$_1$s) to remove the dependency from L_1' to L_1. When $k = 0$, the rule becomes $L \to L_1 s$ and the procedure in Algorithm 5 uses the rules of L_1 to rewrite $L_1 s$; note that in this case L_1 can have only linear-form rules. After linearizeFirstNT(L $\to$ $t_1 \dots t_k$L$_1$s) is finished, this round removes rule $L \to t_1 \dots t_k L_1 s$ from P, updates the dependency graph G (as new nonterminals are added and rules are changed).
- It is easy to see from Algorithm 5 that all non-linear form rules added in the first round have the form of $L_u \to \vec{t} s$, where L_u can be L or a newly generated nonterminal, $\vec{t}$ is a sequence of plain symbols or matched tokens, and s is the string in the original rule $L \to t_1 \dots t_k L_1 s$.

ALGORITHM 5: linearizeFirstNT($L \to t_1 \ldots t_k L_1 s$)

```
1  Global G is the dependency graph and G^SCC is G's condensation graph;
2  Global V_done is the set of already processed terminals;
3  Global T is the translation table;
4  Global P is the current set of rules;
5  if k > 0 then
6      if L_1 s ∈ dom(T) then
7          L'_1 ← T(L_1 s);
8      else
9          L'_1 ← newNonterminal();
10         Add (L_1 s, L'_1) to T;
11         linearizeFirstNT(L'_1 → L_1 s);
12     end
13     Add L → t_1 ... t_k L'_1 to P;
14 else
15     k = 0
16 end
17 for rule r of L_1 do
18     if r is L_1 → t'_1 ... t'_j then
19         Add rule L → t'_1 ... t'_j s to P;
20     else
21         r is L_1 → t'_1 ... t'_j L_2
22     end
23     if L_2 s ∈ dom(T) then
24         L'_2 ← T(L_2 s);
25     else
26         L'_2 ← newNonterminal();
27         Add (L_2 s, L'_2) to T;
28         linearizeFirstNT(L'_2 → L_2 s);
29     end
30     Add L → t'_1 ... t'_j L'_2 to P;
31 end
```

Compared to the original rule, progress has been made, since the size of s is less than the size of $L_1 s$.

In the next round, the algorithm then takes rules of form $L_u \to \vec{t} s$ and feeds those non-linear ones to linearizeFirstNT(−) to remove the dependency from L_u to the first nonterminal in s.

We continue these rounds until the original rule $L \to t_1 \ldots t_k L_1 s$ is completely linearized.

After $L \to t_1 \ldots t_k L_1 s$ is completely linearized, the algorithm continues to the next iteration.

We have now finished the discussion of the translation algorithm. For its correctness, we can prove that the algorithm produces an equivalent grammar as the input grammar by showing that each rewriting step creates an equivalent grammar.

For termination, the following lemma shows that linearizeFirstNT(−) terminates:

Lemma G.1. linearizeFirstNT($L \to t_1 \ldots t_k L_1 s$) *terminates, if* L_1 *and terminals in* allDep(G^{SCC}, L_1) *have only linear-form rules.*

Proof. Inspecting Algorithm 5, we know that a recursive call to linearizeFirstNT($L'_u \to L_u s$) is made only when $L_u \in \{L_1\} \cup \text{allDep}(G^{SCC}, L_1)$ and $L_u s$ is not in T. Since the set $\{L_1\} \cup$

allDep(G^{SCC}, L_1) is finite and $L_u s$ is added to T when a recursive call linearizeFirstNT($L'_u \rightarrow L_u s$) is made, only a finite number of recursive calls can be made. Therefore, the procedure terminates. □

Further, as mentioned earlier, all generated rules when calling linearizeFirstNT($L \rightarrow t_1 \ldots t_k L_1 s$) have the form of $L_u \rightarrow \vec{t}s$, from which we know that the distance from the first nonterminal in $\vec{t}s$ to the end is shorter than the distance from L_1 to the end in $t_1 \ldots t_k L_1 s$. Moreover, it is easy to show only a finite number of rules of the form $L_u \rightarrow \vec{t}s$ are generated. In summary, at each round of linearizing the rule $L \rightarrow t_1 \ldots t_k L_1 s$, the number of invocations to linearizeFirstNT(−) is finite and round $j + 1$ is simpler than round j in the sense that a rule in round $j + 1$ has a shorter distance from the first nonterminal to the end of the rule. Therefore, the process of linearizing rule $L \rightarrow t_1 \ldots t_k L_1 s$ terminates.

Finally, since the number of rules in the input grammar is finite, there is only a finite number of iterations. Therefore, the translation algorithm terminates.

H THE FULL EVALUATION

The full evaluation results are shown in Tables 7–11.

Table 7. The Parsing Times of JSON

Name	#Token	ANTLR Parse	Bison Parse	VPG Parse + Extract	VPG Conv	VPG Total	ANTLR Lex	JFlex
blog_entries	1,402	11.2 ms	0.8 ms	0.08 ms	0.1 ms	0.2 ms	11.0 ms	5.5 ms
gists	6,302	15.1 ms	1.9 ms	0.36 ms	0.5 ms	0.9 ms	12.3 ms	3.9 ms
emojis	7,198	14.7 ms	2.0 ms	0.40 ms	0.5 ms	0.9 ms	12.8 ms	4.6 ms
github	11,680	17.2 ms	2.7 ms	0.77 ms	1.4 ms	2.2 ms	15.9 ms	6.8 ms
poked	12,916	18.4 ms	3.4 ms	0.88 ms	1.5 ms	2.4 ms	14.6 ms	5.6 ms
Members	13,226	17.8 ms	4.3 ms	0.87 ms	1.3 ms	2.2 ms	13.0 ms	3.9 ms
senator	17,932	19.4 ms	4.2 ms	1.19 ms	1.8 ms	3.0 ms	17.4 ms	6.4 ms
AskReddit	18,975	20.1 ms	3.7 ms	1.33 ms	2.0 ms	3.4 ms	16.8 ms	8.1 ms
parliament	27,182	21.6 ms	4.8 ms	1.98 ms	3.3 ms	5.3 ms	16.5 ms	12.0 ms
prize	30,210	22.1 ms	5.8 ms	2.24 ms	3.2 ms	5.4 ms	17.2 ms	10.6 ms
municipis	39,292	27.2 ms	5.6 ms	3.06 ms	3.8 ms	6.8 ms	23.8 ms	11.6 ms
reddit_all	52,941	26.5 ms	8.0 ms	4.14 ms	7.0 ms	11.1 ms	26.1 ms	15.1 ms
y77d_th95	54,666	27.2 ms	8.3 ms	4.04 ms	5.6 ms	9.6 ms	23.8 ms	12.5 ms
twitter	55,264	23.9 ms	6.8 ms	4.37 ms	5.5 ms	9.9 ms	104.9 ms	24.3 ms
representative	70,750	28.7 ms	7.7 ms	4.92 ms	7.6 ms	12.5 ms	29.2 ms	24.8 ms
transactions	71,744	30.5 ms	9.5 ms	5.54 ms	8.1 ms	13.6 ms	24.2 ms	19.5 ms
laureate	79,953	31.6 ms	11.0 ms	4.90 ms	11.3 ms	16.2 ms	25.2 ms	12.9 ms
catalog	135,991	53.5 ms	20.2 ms	7.33 ms	17.0 ms	24.3 ms	39.8 ms	39.1 ms
canada	334,374	103.7 ms	49.9 ms	17.02 ms	51.5 ms	68.6 ms	46.7 ms	39.5 ms
educativos	426,398	92.3 ms	40.5 ms	20.47 ms	48.2 ms	68.7 ms	108.5 ms	75.3 ms
airlines	555,410	106.5 ms	58.0 ms	30.01 ms	68.2 ms	98.2 ms	87.8 ms	91.3 ms
movies	751,917	171.2 ms	81.9 ms	39.40 ms	125.1 ms	164.5 ms	87.9 ms	97.9 ms
js	1,288,350	224.5 ms	127.5 ms	72.14 ms	161.9 ms	234.0 ms	141.6 ms	141.4 ms

Table 8. The Parsing Time (ms) of XML

Name	#Token	ANTLR Parse	Bison Parse	VPG Parse + Extract	VPG Conv	VPG Total	ANTLR Lex	JFlex
nav_63_0	351	10.18	1.40	0.02	0.01	0.03	10.56	0.76
nav_78_0	354	8.90	1.32	0.02	0.02	0.04	10.44	0.74
nav_50_0	523	9.87	1.74	0.03	0.02	0.05	11.06	0.98
cd_catalog	733	9.71	1.58	0.04	0.03	0.07	9.41	0.67
form	740	10.50	1.68	0.04	0.03	0.07	14.47	1.35
OfficeOrder	898	10.58	1.89	0.05	0.04	0.09	13.40	1.17
book-order	2,723	12.73	2.73	0.14	0.28	0.42	12.69	1.80
book	2,723	13.38	2.96	0.14	0.28	0.41	12.69	1.76
bioinfo	3,165	13.51	2.80	0.17	0.27	0.44	15.93	2.36
soap_small	3,415	13.24	3.27	0.17	0.37	0.54	14.40	2.04
cd_big	4,829	14.69	3.24	0.26	0.60	0.87	13.36	2.49
soap_mid	17,015	20.26	7.34	1.17	2.92	4.09	18.89	10.89
ORTCA	39,072	30.02	8.30	3.22	6.31	9.53	54.09	71.00
blog	43,000	24.52	11.35	3.38	6.92	10.30	55.60	35.01
SUAS	118,446	54.32	30.51	6.37	20.56	26.93	88.16	130.68
po1m	128,742	65.65	28.18	6.54	20.74	27.28	48.21	30.79
soap	340,015	125.32	58.69	16.15	55.71	71.86	67.39	52.75
bioinfo_big	406,493	107.50	67.30	20.85	72.30	93.15	89.64	74.09
address	1,437,142	253.83	231.90	74.04	283.35	357.40	200.98	250.01
cd	4,198,694	720.94	734.56	204.18	704.03	908.21	415.73	508.57
po	9,266,526	1,359.55	1,834.59	461.16	1,527.52	1,988.69	854.08	1,617.16

Table 9. The Parsing Time (ms) of HTML

Name	#Token	ANTLR Parse	Bison Parse	VPG Parse + Extract	VPG Conv	VPG Total	ANTLR Lex	JFlex
uglylink	13	14.28	2.03	0.00	0.00	0.00	9.77	0.14
script1	26	14.88	2.20	0.00	0.00	0.00	8.18	0.20
style1	26	15.42	2.20	0.00	0.00	0.00	7.89	0.22
attvalues	27	35.70	2.14	0.00	0.00	0.01	11.50	0.21
html4	38	16.42	2.21	0.00	0.00	0.01	10.85	0.31
antlr	545	37.14	4.75	0.04	0.23	0.26	22.49	2.63
google	659	24.08	3.77	0.04	0.27	0.30	31.78	33.82
gnu	1,013	38.58	5.40	0.06	0.10	0.16	19.86	2.95
freebsd	1,521	49.51	5.74	0.09	0.21	0.29	19.29	2.54
abc.com	1,524	43.20	4.83	0.09	0.21	0.30	30.64	15.06
github	2,547	46.82	6.83	0.21	0.26	0.47	26.12	4.71
metafilter	2,615	50.94	5.49	0.21	0.45	0.66	33.13	30.18
wikipedia	2,750	37.49	6.46	0.24	0.37	0.61	33.93	10.71
nbc.com	3,821	94.82	7.37	0.32	0.59	0.92	46.21	14.79
bbc	3,883	88.48	5.62	0.32	0.80	1.12	95.48	51.55
cnn1	4,974	89.63	6.84	0.29	1.02	1.31	42.38	34.00
reddit2	4,976	31.97	6.85	0.28	1.11	1.39	53.65	41.84
reddit	4,989	31.37	7.41	0.27	1.10	1.37	53.29	38.30
digg	6,250	126.95	8.53	0.51	1.34	1.85	52.07	38.00
youtube	16,316	49.35	11.56	1.05	5.92	6.97	54.83	58.53

Table 10. Comparison with Hand-crafted Parsers (ms) of Parsing JSON

Name	#Token	ANTLR Lex	VPG Parse + Extract	VPG Total	ANTLR Lex + VPG Total	SpiderM	JSCore	V8	Chakra
Members	1,402	13.03	0.87	2.24	15.27	15.06	45.33	8.80	13.05
poked	6,302	14.61	0.88	2.40	17.00	14.82	50.83	8.72	12.68
gists	7,198	12.38	0.36	0.90	13.28	16.29	85.23	9.21	13.72
senator	11,680	17.49	1.19	3.07	20.57	16.01	55.43	9.98	13.87
AskReddit	12,916	16.81	1.33	3.42	20.24	14.67	51.59	9.28	13.42
blog_entries	13,226	11.03	0.08	0.26	11.28	14.51	51.90	8.85	13.21
github	17,932	15.99	0.77	2.21	18.20	15.78	66.17	9.99	12.43
emojis	18,975	12.83	0.40	0.98	13.82	15.40	46.53	9.19	14.14
parliament	27,182	16.53	1.98	5.35	21.88	16.36	70.35	10.38	13.69
prize	30,210	17.25	2.24	5.45	22.71	16.76	134.30	9.98	13.15
y77d_th95	39,292	23.87	4.04	9.63	33.50	20.72	178.03	12.53	16.35
municipis	52,941	23.89	3.06	6.89	30.78	20.02	60.20	11.48	16.54
laureate	54,666	25.26	4.90	16.22	41.47	17.41	105.22	11.96	15.37
reddit_all	55,264	26.15	4.14	11.15	37.30	22.00	92.54	15.61	17.44
transactions	70,750	24.23	5.54	13.65	37.87	22.27	132.86	15.21	16.19
twitter	71,744	104.96	4.37	9.91	114.88	21.09	78.90	11.78	18.35
representative	79,953	29.22	4.92	12.56	41.78	20.72	178.03	12.53	16.35
catalog	135,991	39.83	7.33	24.36	64.19	33.81	70.64	27.59	24.77
canada	334,374	46.74	17.02	68.61	115.35	57.14	67.51	33.95	43.83
educativos	426,398	108.58	20.47	68.75	177.34	70.78	420.99	44.53	48.64
airlines	555,410	87.82	30.01	98.26	186.08	73.86	95.12	41.93	55.52
movies	751,917	87.92	39.40	164.58	252.51	70.93	106.41	68.00	50.85
JSON.parse	1,288,350	141.64	72.14	234.07	375.71	118.11	139.21	76.36	87.81

Table 11. Comparison with Hand-crafted Parsers (ms) of Parsing XML

Name	#Token	ANTLR Lex	VPG Parse + Extract	VPG Total	ANTLR Lex + VPG Total	Fast-XML	Libxmljs	SAX-JS	HTMLP2
nav_63_0	351	10.56	0.02	0.03	10.59	0.14	0.13	0.67	0.20
nav_78_0	354	10.44	0.02	0.04	10.47	0.12	0.12	0.68	0.20
nav_50_0	523	11.06	0.03	0.05	11.11	0.21	0.17	0.99	0.30
cd_catalog	733	9.41	0.04	0.07	9.48	0.22	0.09	0.36	0.16
form	740	14.47	0.04	0.07	14.54	0.38	0.17	0.93	0.27
OfficeOrder	898	13.40	0.05	0.09	13.49	0.37	0.14	0.88	0.35
book-order	2,723	12.69	0.14	0.42	13.11	0.69	0.26	1.41	0.54
book	2,723	12.69	0.14	0.41	13.10	0.64	0.26	1.41	0.54
bioinfo	3,165	15.93	0.17	0.44	16.37	0.99	0.38	2.16	0.90
soap_small	3,415	14.40	0.17	0.54	14.94	0.85	0.40	2.42	0.78
cd_big	4,829	13.36	0.26	0.87	14.23	1.17	0.44	2.20	0.88
soap_mid	17,015	18.89	1.17	4.09	22.98	4.11	1.95	11.74	3.81
ORTCA	39,072	54.09	3.22	9.53	63.62	138.49	91.25	664.62	88.79
blog	43,000	55.60	3.38	10.30	65.91	29.57	6.56	48.07	17.83
SUAS	118,446	88.16	6.37	26.93	115.09	254.39	182.43	1,213.67	168.74
po1m	128,742	48.21	6.54	27.28	75.49	33.51	11.56	95.22	27.11
soap	340,015	67.39	16.15	71.86	139.25	110.14	38.13	260.88	76.24
bioinfo_big	406,493	89.64	20.85	93.15	182.79	116.93	42.74	276.96	82.29
address	1,437,142	200.98	74.04	357.40	558.37	584.40	196.12	1,012.01	330.51
cd	4,198,694	415.73	204.18	908.21	1,323.94	1,298.48	419.07	2,102.58	734.97
po	9,266,526	854.08	461.16	1,988.69	2,842.77	3,278.30	897.30	6,617.60	1,826.90

ACKNOWLEDGMENTS

The authors would like to thank anonymous reviewers for their detailed, constructive comments.

REFERENCES

[1] Rajeev Alur. 2016. Nested Words. Retrieved from https://www.cis.upenn.edu/~alur/nw.html.
[2] Rajeev Alur and P. Madhusudan. 2009. Adding nesting structure to words. *J. Assoc. Comput. Machin.* 56, 3 (May 2009), 16:1–16:43.
[3] ANTLR. 2014. Grammars-v4. Retrieved from https://github.com/antlr/grammars-v4.
[4] Aditi Barthwal and Michael Norrish. 2009. Verified, executable parsing. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin, 160–174.
[5] Jean-Philippe Bernardy and Patrik Jansson. 2016. Certified context-free parsing: A formalisation of Valiant's algorithm in Agda. *Logic. Meth. Comput. Sci.* 12 (01 2016). DOI: https://doi.org/10.2168/LMCS-12(2:6)2016
[6] Janusz A. Brzozowski. 1964. Derivatives of regular expressions. *J. ACM* 11 (1964), 481–494.
[7] John Cocke. 1969. *Programming Languages and Their Compilers: Preliminary Notes.* New York University, New York, NY.
[8] Nils Anders Danielsson. 2010. Total parser combinators. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*) *(ICFP'10).* Association for Computing Machinery, New York, NY, 285–296. DOI: https://doi.org/10.1145/1863543.1863585
[9] Pierce Darragh and Michael D. Adams. 2020. Parsing with zippers (functional pearl). *Proc. ACM Program. Lang.* 4, ICFP (2020), 1–28.
[10] Franklin Lewis DeRemer. 1969. *Practical translators for LR (k) languages.* Ph. D. Dissertation. Massachusetts Institute of Technology.
[11] F. L. Deremer. 1969. *PRACTICAL TRANSLATORS FOR LR(K) LANGUAGES.* Massachusetts Institute of Technology.
[12] Justin Dorfman. 2015. awesome-json-datasets. Retrieved from https://github.com/jdorfman/awesome-json-datasets.

[13] Danny Dubé and Marc Feeley. 2000. Efficiently building a parse tree from a regular expression. *Acta Inform.* 37, 2 (2000), 121–144.
[14] Jay Earley. 1970. An efficient context-free parsing algorithm. *Commun. ACM* 13, 2 (Feb. 1970), 94–102.
[15] Romain Edelmann, Jad Hamza, and Viktor Kunčak. 2020. Zippy LL(1) parsing with derivatives. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*. Association for Computing Machinery, New York, NY, 1036–1051. DOI: https://doi.org/10.1145/3385412.3385992
[16] Felix. 2010. Htmlparser2. Retrieved from https://github.com/fb55/htmlparser2.
[17] Denis Firsov and Tarmo Uustalu. 2014. Certified CYK parsing of context-free languages. *J. Logic. Algeb. Meth. Program.* 83, 5 (2014), 459–468. DOI: https://doi.org/10.1016/j.jlamp.2014.09.002
[18] Free Software Foundation. 2021. GNU Bison. Retrieved from https://www.gnu.org/software/bison/.
[19] GoogleChromeLabs. 2019. json-parse-benchmark. Retrieved from https://github.com/GoogleChromeLabs/json-parse-benchmark.
[20] Ian Henderson. 2017. Owl. Retrieved from https://github.com/ianh/owl.
[21] Ian Henriksen, Gianfranco Bilardi, and Keshav Pingali. 2019. Derivative grammars: A symbolic approach to parsing with derivatives. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 1–28.
[22] Xiaodong Jia, Ashish Kumar, and Gang Tan. 2021. A derivative-based parser generator for visibly Pushdown grammars. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–24.
[23] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) parsers. In *Programming Languages and Systems*, Helmut Seidl (Ed.). Springer Berlin, 397–416.
[24] Tadao Kasami. 1965. *An Efficient Recognition and Syntax-Analysis Algorithm for Context-Free Languages.* Technical Report. Air Force Cambridge Research Laboratory.
[25] Adam Koprowski and Henri Binsztok. 2010. TRX: A formally verified parser interpreter. In *Programming Languages and Systems*, Andrew D. Gordon (Ed.). Springer Berlin, 345–365.
[26] Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. 2019. A verified LL(1) parser generator. In *10th International Conference on Interactive Theorem Proving (ITP 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 141)*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 24:1–24:18. DOI: https://doi.org/10.4230/LIPIcs.ITP.2019.24
[27] Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. 2021. CoStar: A verified ALL(*) parser. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21)*. Association for Computing Machinery, New York, NY, 420–434. DOI: https://doi.org/10.1145/3453483.3454053
[28] libxmljs. 2009. Libxmljs. Retrieved from https://github.com/libxmljs/libxmljs.
[29] Jay McCarthy, Burke Fetscher, Max New, Daniel Feltey, and Robert Bruce Findler. 2016. A Coq library for internal verification of running-times. In *Functional and Logic Programming*, Oleg Kiselyov and Andy King (Eds.). Springer International Publishing, Cham, 144–162.
[30] Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with derivatives: A functional pearl. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP'11)*. Association for Computing Machinery, New York, NY, 189–195. DOI: https://doi.org/10.1145/2034773.2034801
[31] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: Better, faster, stronger SFI for the X86. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*. Association for Computing Machinery, New York, NY, 395–404. DOI: https://doi.org/10.1145/2254064.2254111
[32] NaturalIntelligence. 2017. fast-xml-parser. Retrieved from https://github.com/NaturalIntelligence/fast-xml-parser.
[33] Ulf Norell. 2007. *Towards a Practical Programming Language Based on Dependent Type Theory.* Vol. 32. Chalmers University of Technology.
[34] Scott Owens, John Reppy, and Aaron Turon. 2009. Regular-expression derivatives re-examined. *J. Funct. Program.* 19, 2 (Mar. 2009), 173–190.
[35] Terence Parr. 2022. ANTLR (ANother Tool for Language Recognition). Retrieved from https://www.antlr.org/.
[36] Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) parsing: The power of dynamic analysis. *SIGPLAN Not.* 49, 10 (Oct. 2014), 579–598. DOI: https://doi.org/10.1145/2714064.2660202
[37] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. 2017. NEZHA: Efficient domain-independent differential testing. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'17)*. IEEE Computer Society, 615–632. DOI: https://doi.org/10.1109/SP.2017.27
[38] Tahina Ramananandro, Antoine Delignat-Lavaud, Cedric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. EverParse: Verified secure zero-copy parsers for authenticated message formats. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security'19)*. USENIX Association, 1465–1482. Retrieved from https://www.usenix.org/conference/usenixsecurity19/presentation/delignat-lavaud.

[39] Tom Ridge. 2011. Simple, functional, sound and complete parsing for all context-free grammars. In *Certified Programs and Proofs*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). Springer Berlin, 103–118.
[40] Isaac Z. Schlueter. 2010. sax-js. Retrieved from https://github.com/isaacs/sax-js.
[41] JFlex Team. 2023. JFlex. Retrieved from https://jflex.de/.
[42] XimpleWare. 2004. VTD-XML Benchmark Report for Version 2.3. Retrieved from https://vtd-xml.sourceforge.io/2.3/benchmark_2.3_parsing_only.html.
[43] Milo Yip. 2014. nativejson-benchmark. Retrieved from https://github.com/miloyip/nativejson-benchmark.
[44] Daniel H. Younger. 1967. Recognition and parsing of context-free languages in time n^3. *Inf. Contr.* 10, 2 (1967), 189–208.

Received 6 June 2022; revised 27 January 2023; accepted 21 March 2023