A Derivative-Based Parser Generator for Visibly Pushdown Grammars

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

In this paper, 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 a popular parsing tool ANTLR and other popular hand-crafted parsers. The correctness of the core parsing algorithm is 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. 2021. A Derivative-Based Parser Generator for Visibly Pushdown Grammars. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 151 (October 2021), 24 pages. https://doi.org/10.1145/3485528

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 and formal verification. First, 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 $O(n^3)$. Moreover, formally verifying general CFG parsing algorithms is a difficult task. To our best knowledge, there is no formally verified CFG parsing algorithm due to its complexity.

Authors' addresses: Xiaodong Jia, The Pennsylvania State University, 201 Old Main, State College, Pennsylvania, USA, 16802; Ashish Kumar, The Pennsylvania State University, 201 Old Main, State College, Pennsylvania, USA, 16802; Gang Tan, The Pennsylvania State University, 201 Old Main, State College, Pennsylvania, USA, 16802.

2475-1421/2021/10-ART151
https://doi.org/10.1145/3485528

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 restrictions. Please see the related-work section for discussion about common kinds of restrictions, leading to parsing frameworks such as LL(k), LR(k) (Deremer 1969), PEGs (Parsing Expression Grammars) (Ford 2004), etc.

This paper explores an alternative angle of building parsers based on Visibly Pushdown Grammars (VPGs) (Alur and Madhusudan 2009). 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 satisfy restrictions placed by parsing frameworks such as LL(k), asking users to specify what nonterminals are call and return symbols is less of a burden.

VPGs have been used in program analysis, XML processing, and other fields, but their potential in parsing has not been fully exploited. In this paper, 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 paper makes the following contributions.

- We present a derivative-based algorithm for VPG recognition and parsing. The algorithm is guaranteed to run in linear time. 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 present a surface grammar called tagged CFGs to allow more convenient use of our parsing framework. Users can use their familiar CFGs for developing grammars and provide additional tagging information on nonterminals. A sound translator then converts a tagged CFG to a VPG.

The remainder of this paper is organized as follows. We first introduce VPGs in Section 2 and discuss related work in Section 3. Section 4 presents a derivative-based VPG recognizer, which enlightens the parsing algorithm discussed in Section 5. The translator and tagged CFGs are discussed in Section 6. We then evaluate the VPG parser in Section 7.

2 BACKGROUND

As a class of grammars, VPGs (Alur and Madhusudan 2009) have been used in program analysis, XML processing, and other fields. Compared with CFGs, VPGs enjoy many good properties. 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, VPGs enjoy all closure properties, including intersection and complement. As will be shown in this paper, these properties enable the building of linear-time parsers for VPGs, and make VPGs 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.

We next give a formal account of VFGs. 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_l, Σ_c, Σ_r, which contain plain, call and return symbols, respectively. Notation-wise, a terminal in Σ_c is tagged with $\langle$ on the left, and a terminal in Σ_r is tagged with $\rangle$ on the right. For example, $\langle a$ is a call symbol in Σ_c, and $b\rangle$ is a return symbol in Σ_r.

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 $\Sigma = \Sigma_l \cup \Sigma_c \cup \Sigma_r$, if every production rule in P is in one of the following forms.

(1) $L \rightarrow \epsilon$, where ϵ stands for the empty string;
(2) $L \rightarrow cL_1$, where $c \in \Sigma_l$;
(3) $L \rightarrow \langle aL_1 b\rangle L_2$, where $\langle a \in \Sigma_c$ and $b\rangle \in \Sigma_r$.

Note that in $L \rightarrow cL_1$ terminal c must be a plain symbol, and in $L \rightarrow \langle aL_1 b\rangle L_2$ 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 taken from a VPG for XML:

$$\text{element} \rightarrow \text{OpenTag content CloseTag} \mid \text{SingleTag}$$

In this example, nonterminals start with a lowercase character, such as "element", and terminals start with an uppercase character, such as "OpenTag". 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 rest of the paper, we use the term VPGs for well-matched VPGs and use the term general VPGs to 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 $L \rightarrow \langle aL'$ and $L \rightarrow b\rangle L'$, which we call *pending rules*. 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 $\Sigma = \Sigma_l \cup \Sigma_c \cup \Sigma_r$ and $V = V^0 \cup V^1$, if every rule in P is in one of the following forms:

(1) $L \rightarrow \epsilon$;
(2) $L \rightarrow iL_1$, where $i \in \Sigma$, and if $L \in V^0$ then (1) $i \in \Sigma_l$ and (2) $L_1 \in V^0$;
(3) $L \rightarrow \langle aL_1 b\rangle L_2$, where $\langle a \in \Sigma_c, b\rangle \in \Sigma_r$, $L_1 \in V^0$, 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 $L \rightarrow \langle aL_1 b\rangle L_2$, nonterminal L_1 must be a well-matched nonterminal; so P cannot include rules such as $L_1 \rightarrow \langle aL_3$, since L_1 is supposed to generate only well-matched strings.

The notion of a derivation in VPGs is the same as the one in CFGs. We write $w \rightarrow w'$ to mean a single derivation step according to a grammar, where w and w' are strings of terminals or nonterminals. We write $L \rightarrow^* w$ 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) (Deremer 1969), Earley (Earley 1970), CYK (Cocke 1969; Kasami 1965; Younger 1967), 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 (1964) first presented the concept of derivatives and used it to build a recognizer for regular expressions. The idea was revived by Owens et al. (2009) and generalized to generate parsers for CFGs (Might et al. 2011), with an exponential worst-case time complexity. More recently, a symbolic approach (Henriksen et al. 2019) for parsing CFGs with derivatives was presented, with cubic time complexity. Finally, Edelmann et al. (2020) presented a formally verified, derivative-based, linear-time parsing algorithm for LL(1) context-free expressions.

Owl is an open-source project[1] 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 full 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. (2017)). However, testing cannot show the absence of bugs. Formal verification has also been applied to the building of high-assurance parsers. Jourdan et al. (2012) applied the methodology of translation validation and implemented a verified parser validator for LR(1) grammars. RockSalt (Morrisett et al. 2012) included a verified parser for regular expression based DSL. Lasser et al. (2019) and Edelmann et al. (2020) presented verified LL(1) parsers but we are not aware of fully verified LL(k) parsers. Lasser et al. (2021) implemented a verified ALL(*) parser, which is the algorithm behind ANTLR4. Koprowski and Binsztok (2010) implemented a formally verified parser generator for Parsing Expression Grammars (PEG). Ramananandro et al. (2019) presented a verified parser generator for tag-length-value binary message format descriptions. We formalize our derivative-based, VPG parsing algorithm and its correctness proofs in Coq.

4 VPG BASED RECOGNITION

We next present an algorithm for converting a VPG into a deterministic PDA using a derivative-based algorithm. The resulting PDA accepts the same set of strings as the input VPG.

Before we present the formal conversion process, we discuss informally the intuition of the states, the stack, and the transition function of the PDA that is converted from an input VPG. A state in the resulting PDA is a subset of $V \times V$, i.e., a set of nonterminal pairs. A nonterminal pair (L_1, L_2) tells that the next part 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 there exists a derivation sequence $L_1 \rightarrow^* \omega_1 L_2 \omega_2$, where ω_1 is a sequence of terminals and does not contain an unmatched call symbol, and ω_2 is a sequence of terminals or nonterminals.

[1] https://github.com/ianh/owl

To give an example, suppose we have the following VPG rules, with L_0 being the start symbol. We omit the rules for L_2 and L_3, which are irrelevant for the discussion.

$$L_0 \rightarrow cL_1;\ L_1 \rightarrow \langle aL_2b\rangle L_3.$$

The start state of the PDA should be $\{(L_0, L_0)\}$, meaning that the input string should match L_0 and the context is also L_0 since L_0 can be derived from itself (in zero steps) without generating an unmatched call symbol. Given that start state, if the next input symbol is c, then the PDA should transition to state $\{(L_0, L_1)\}$; that is, the rest of the input should match L_1 and the context is still L_0, since L_1 is derived from L_0 without generating an unmatched call symbol. Now suppose the next input char is $\langle a$; then the next state should be $\{(L_2, L_2)\}$; notice that there is a context switch as there is an unmatched call symbol that is encountered when going from L_1 to L_2 using the rule $L_1 \rightarrow \langle aL_2b\rangle L_3$. As we will show, for this transition, the PDA will also push $\{(L_0, L_1)\}$ and $\langle a$ to the stack, so that 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 $\{(L_0, L_3)\}$.

For this example, all states contain just one pair. In general, a state may contain multiple pairs because of possible ambiguity. For example, imagine there is an additional rule $L_0 \rightarrow cL_4$; then from start state $\{(L_0, L_0)\}$, after encountering c, the PDA transitions to state $\{(L_0, L_3), (L_0, L_4)\}$, reflecting that the rest of the input can match either L_3 or L_4.

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

Definition 4.1 (PDA states and stacks)**.** Given a VPG, 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 $\langle a \in \Sigma_c$ a call symbol. We write $\perp$ 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.

We next utilize the notion of derivatives (Brzozowski 1964; Might et al. 2011; Owens et al. 2009) to compute a recognizer PDA that accepts the same language (i.e., a set of strings) as an input VPG. 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:

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

As we will show in Definition 4.3, a recognizer PDA configuration (S, T) stands for a language. Transferring the general definition of derivatives to recognizer PDA configurations produces a set of derivative functions, discussed next.

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.

Definition 4.2 (Derivative functions)**.**
(1) $\delta_c(S) = (S', \lambda T.T)$, where

$$S' = \{(L_1, L_3) \mid \exists L_2, (L_1, L_2) \in S \wedge (L_2 \rightarrow cL_3) \in P\};$$

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

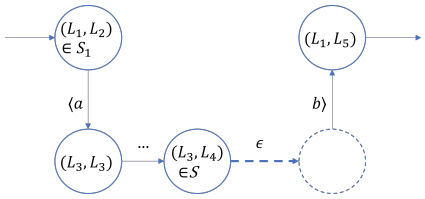


Fig. 1. An example of state transitions: (L_1, L_2) transfers to (L_3, L_3) with symbol $\langle a$ and there is a rule $L_2 \rightarrow \langle aL_3b\rangle L_5$, and finally transfers to (L_1, L_5) with symbol $b\rangle$.

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

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

For a call symbol $\langle a \in \Sigma_c$, it checks each pair (L_1, L_2) in the current state S; if there is a rule $L_2 \rightarrow \langle aL_3b\rangle L_4$, 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

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

For a return symbol $b\rangle \in \Sigma_r$ 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 $L_2 \rightarrow \langle aL_3b\rangle\ L_5$, then pair (L_1, L_5) becomes part of the new state; note that it checks $L_4 \rightarrow \epsilon$ 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 1 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 4.3 (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. $L_2 \rightarrow^* w$,
(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) $L_4 \rightarrow^* w_1$ and
  (b) $\exists (L_1, L_2) \in S', \exists L_5, (L_2 \rightarrow \langle aL_3b\rangle L_5) \in P \wedge (\{(L_1, L_5)\},\ T') \rightsquigarrow w_2$.

The correctness of derivative functions is stated in the following theorem, whose correctness proof is detailed in Appendix A of the companion technical report (Jia et al. 2021). 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 4.1 (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

1: Input: a VPG $G = (V, \Sigma, P, L_0)$, δ.
2: $S_0 \leftarrow \{(L_0, L_0)\}$.
3: Initialize the set for new states $N = \{S_0\}$.
4: Initialize the set for all produced states $A = N$.
5: Initialize the set for transitions $\mathcal{T} = \{\}$.
6: **repeat**
7: $\quad N' \leftarrow \{(i, f, S, S') \mid (S', f) = \delta_i S,\ S \in N,\ \text{and } i \in \Sigma_c \cup \Sigma_l\}$
8: $\quad$ Add edge (S, S') marked with (i, f) to $\mathcal{T}$, where $(i, f, S, S') \in N'$.
9: $\quad R \leftarrow \{[S, \langle a] \mid S \in A \text{ and } \langle a \in \Sigma_c\}$
10: $\quad N_R \leftarrow \{(b\rangle, r, f, S, S') \mid (S', f) = \delta_{b\rangle}(S, r), S \in A, b\rangle \in \Sigma_r, r \in R\}$
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: $\quad N \leftarrow \{S' \mid (\_, \_, \_, S') \in N' \vee (\_, \_, \_, \_, S') \in N_R\} - A$
13: $\quad A \leftarrow A \cup N$
14: **until** $N = \emptyset$
15: Return $(S_0, A, \mathcal{T})$.

$$L \to \langle aAb\rangle L \mid \epsilon;\ A \to cC \mid cD;\ C \to cE;\ D \to dE;\ E \to \epsilon$$

Fig. 2. An example VPG.

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.

$$\begin{aligned} A \quad = \quad & A \cup \{S' \mid c \in \Sigma_l,\ S \in A,\ \delta_c(S) = (S', f)\} \\ & \cup \{S' \mid \langle a \in \Sigma_c,\ S \in A,\ \delta_{\langle a}(S) = (S', f)\} \\ & \cup \{S' \mid b\rangle \in \Sigma_r,\ \langle a \in \Sigma_c,\ S \in A,\ S' \in A,\ \delta_{b\rangle}(S, [S', \langle a]) = (S', f)\} \end{aligned}$$

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 $i \in \Sigma_c \cup \Sigma_l$, if $\delta_i(S) = (S', f)$, then $S' \in A$; (3) for states $S, S' \in A - N$, $\langle a \in \Sigma_c$, and $b\rangle \in \Sigma_r$, 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.

As an example, consider the VPG in Figure 2. The PDA generated by Algorithm 1 for this VPG is shown in Figure 3. The input symbols and stack actions are marked on the edges.

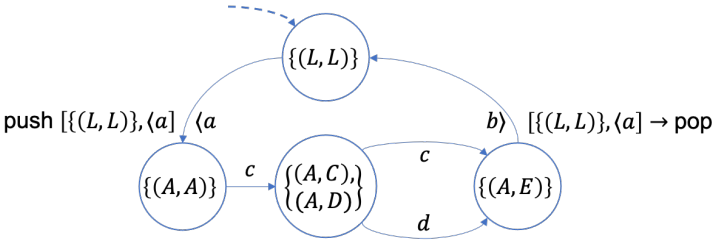


Fig. 3. The PDA generated by Algorithm 1 for grammar in Figure 2. "push $[\{(L, L), \langle a\}]$" means pushing $[\{(L, L), \langle a\}]$ to the stack, and "$[\{(L, L), \langle a\}] \rightarrow$ pop" means removing the top of the stack, when it is $[\{(L, L), \langle a\}]$.

Once the PDA is constructed from a VPG, it can be run on an input string in a standard way. Then PDA correctness can be stated as follows. We detail PDA execution and the correctness proof in Appendix A of the companion technical report (Jia et al. 2021).

Theorem 4.2 (PDA correctness). *For VPG G and its start symbol L_0, a string $w \in \Sigma^*$ can be derived from L_0 (i.e. $L_0 \rightarrow^* w$) iff w is accepted by the corresponding PDA.*

For converting general VPGs (i.e., with pending rules) to PDA, a couple of changes need to be made on 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 of the companion technical report (Jia et al. 2021).

5 VPG BASED PARSING

Parsing is a process to build the parse trees of a given string based on an input grammar. It is equivalent to finding the sequences of rules that can generate the input string. Our VPG-based parsing framework is largely enlightened by the recognizer construction in Section 4. Observe that the execution of the recognizer PDA on an input string w can be represented as a trace of runtime configurations: $(S_0, T_0) \xrightarrow{w_1} (S_1, T_1) \xrightarrow{w_2} (S_2, T_2) \xrightarrow{w_3} \cdots$, where (S_i, T_i) is the configuration after consuming w_i, the i^{th} symbol of w. Each transition in the trace is because of a set of possible rules in the input VPG; therefore, we can augment the configuration trace with information about what rules can be applied during each transition, which gives $(S_0, T_0) \xrightarrow{w_1} ((S_1, T_1), m_1) \xrightarrow{w_2} ((S_2, T_2), m_2) \xrightarrow{w_3} \cdots$, where m_i is the set of possible rules at step i. With the augmented configuration trace, we can construct parse trees for the input string.

Given this intuition, we could build a parser directly based on the recognizer. However, to reduce the burden of formally verifying the parser, we make several trade-offs in designing the parsing algorithm: (1) instead of extending the recognizer, we present a way to construct the parser PDA independently; as will be shown in Section 5.4, this allows us to formalize correctness in a natural way; (2) we replace the context nonterminal in the recognizer state with a boolean value when constructing the parser; this simplifies formal verification, but may introduce invalid edges in the parse forest; as a result, our design adds a pruning step to prune invalid edges.

5.1 Overview of VPG-based parsing

At a high level, our VPG-based parsing framework takes an input VPG and generates three components: (1) a *parser PDA*, which takes an input string and constructs a parse forest representing possibles parses of the string according to the VPG; (2) a *pruner PDA*, which takes the parse forest

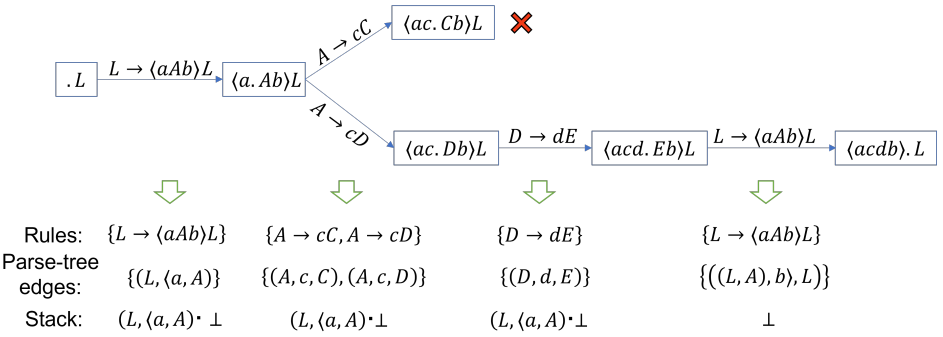


Fig. 4. The parsing process for string $\langle acdb\rangle$ for the grammar in Figure 2. There are two traces. The first one, followed by a red cross mark, becomes invalid after step 2. Possible rules applied in each step are collected in a set, which is converted to the set of parse-tree edges at that step. The stack keeps track of previous rules that generated call symbols, so that later they can be applied to generate the corresponding return symbols.

and removes invalid edges; (3) an *extractor*, which takes the pruned parse forest and extracts parse trees.

Before discussing these steps in detail, we use the well-matched VPG in Figure 2 to illustrate the steps of the parser and the pruner PDAs. Let $w = \langle acdb\rangle$ be the string to parse. Figure 4 visualizes the high-level steps of how our VPG parser parses w. Nodes in the figure are sentential forms with a dot indicating the parsing position. The prefix before the dot in a sentential form is the input seen so far; the nonterminal immediately after the dot is the one to parse next; the remainder of the sentential form is actually represented by the stack in our parser, but for ease of understanding we also add it to the nodes in Figure 4. The figure also shows for each step the set of possible rules, those rules represented as parse-tree edges, and the stack after the step.

As an example, after symbol $\langle a$, we have the sentential form $\langle a.Ab\rangle L$, where $\langle a$ is the already parsed input, A is the nonterminal to parse, and $b\rangle L$ is the remainder. For this step, only one rule is possible: $L \longrightarrow \langle aAb\rangle L$. To represent such a rule used in building a parse tree, we use a triple instead of the rule directly; the reason is to differentiate when $\langle a$ is generated by a matching rule from when $b\rangle$ is generate by the same rule. For example, the parsing of $\langle a$ using $L \longrightarrow \langle aAb\rangle L$ is converted to triple $(L, \langle a, A)$. And the remainder is represented by a stack element $(L, \langle a, A)$, which tells us that a rule such as $L \longrightarrow \langle aAb\rangle L$ was used to create this stack element; with this information, at step 4 of Figure 4, the parser knows that the next symbol should match $b\rangle$ and, after that, L. So the rule $L \longrightarrow \langle aAb\rangle L$ is used twice: at step 1 for $\langle a$ and step 4 for $b\rangle$.

As another example, for the second symbol "c" in the input string, there are two possible rules. One with C as the next nonterminal to parse, and the other with D. However, notice here in both cases the remainders are $b\rangle L$ and represented by the same stack. This example shows the crucial difference between VPG parsing and CFG parsing. In a general CFG parsing algorithm such as GLR, each possibility has its own stack, reflecting the nondeterministic nature of when stacks are changed in CFGs. In contrast, in VPG parsing all possibilities in one step share the same stack, enabled by the VPG property that the stack is changed when consuming only call/return symbols; as a result, the stack can be shared and factored out.

From this example, we can see that, given an input string w of length n, the parser PDA generates a set of possible rules for the i-th input symbol; since each rule is represented by a triple in our parser, each step generates a set of triples m_i. We call the sequence of $[m_1, \ldots, m_n]$ the *parse forest* for w; from the sequence, we can recover all parse trees for w. We call $[e_1, \ldots, e_k]$, where

$e_i \in m_i$ for $i \in [1..k]$, a *trace* of parse-tree edges; when $k = n$, it is a *complete trace*. A complete trace is a linear representation of a candidate parse tree. For example, a complete trace in Figure 4 is $[(L, \langle a, A), (A, c, D), (D, d, E), ((L, A), b\rangle, L)]$, representing a candidate parse tree. Although the number of traces may grow exponentially with the length of the input string[2], the set of distinct rules possible at each step is always finite and bounded by the size of the input grammar. As a result, the parse forest representation is linear to the size of the input string.

Notice that the first trace in Figure 4 is followed by a red cross mark, because the sentential form $\langle ac.Cb\rangle L$ cannot be followed by any rule to generate d. However, the parse forest still keeps the invalid edge (A, c, C). In our approach, we use a pruner PDA to prune invalid parse-tree edges in the parse forest. For example, the pruned parse forest for the above example is $[\{(L, \langle a, A)\}, \{(A, c, D)\}, \{(D, d, E)\}, \{((L, A), b\rangle, E)\}]$.

Among the three components of the VPG parsing framework, the first two can fail: the parser PDA may fail because it cannot make a transition with the next input symbol; the pruner PDA may prune the parse forest to an empty parse forest, meaning that the input string cannot be parsed.

5.2 The parser PDA

As discussed earlier, a parse tree for a VPG can be represented by a linear sequence of triples, each representing an edge in the parse tree. E.g., when rule $L \to \langle aL_1$ is used, it is represented as $(L, \langle a, L_1)$. However, for general VPGs with pending rules, such triples are insufficient. For example, $(L, \langle a, L_1)$ can be the result from the rule $L \to \langle aL_1$, or the rule $L \to \langle aL_1 b\rangle L_2$. We need to further differentiate pending rule edges and matching rule edges, since pending rules cannot be used within matching rules, required by general VPGs (Definition 2.2).

Our solution is to tag every nonterminal in a parse-tree edge a boolean u; a similar notion called *linear acceptance* is discussed by Alur and Madhusudan (2009). Let L be a nonterminal; a tagged L is written as L^u. Intuitively, L^{true} generates only well-matched strings and can use only well-matched rules when generating call/return symbols; in contrast, L^{false} can also use pending rules when generating call/return symbols. For a general VPG, if the parser uses a well-matched rule $L \to \langle aL_1 b\rangle L_2$ to match $\langle a$, then it has to use L_1^{true} to perform parsing next, since general VPGs require that L_1 must generate well-matched strings.

With the above discussion, parse-tree edges for general VPGs can be defined as follows:

Definition 5.1 (The edges $\mathcal{M}_{\text{pln}}, \mathcal{M}_{\text{call}}, \mathcal{M}_{\text{ret}}$)**.** Given a VPG $G = (\Sigma, V, P, L_0)$,

(1) the set of plain edges, denoted as $\mathcal{M}_{\text{pln}}$, is defined as $\{(L^u, c, L_1^u) \mid (L \to cL_1) \in P\}$;
(2) the set of call edges, denoted as $\mathcal{M}_{\text{call}}$, is defined as $\{(L^u, \langle a, L_1^{\text{true}}) \mid \exists b\rangle\, L_2,\ (L \to \langle aL_1 b\rangle L_2) \in P\} \cup \{(L^{\text{false}}, \langle a, L_1^{\text{false}}) \mid (L \to \langle aL_1) \in P\}$;
(3) the set of return edges, denoted as $\mathcal{M}_{\text{ret}}$, is defined as $\{((L^u, L_1^{\text{true}}), b\rangle, L_2^u) \mid \exists \langle a, (L \to \langle aL_1 b\rangle L_2) \in P\} \cup \{(L^{\text{false}}, b\rangle, L_1^{\text{false}}) \mid (L \to b\rangle L_1) \in P\}$.

Note that pending rules $L \to \langle aL_1$ and $L \to b\rangle L_1$ can be used only in edges starting with L^{false}; further, when using a matching rule $L \to \langle aL_1 b\rangle L_2$ to generate edges starting with L_1, its tag must be true as L_1 should match only well-matched strings. As we will formalize later, a complete trace that constitutes a parse tree must satisfy the following constraints: (1) it must start with L_0^{false}, where L_0 is the start nonterminal, (2) it must end with some L_1^{false} for some L_1 such that $(L_1 \to \epsilon) \in P$; the tag must be false so that no matching rule is waiting to be finished, and $L_1 \to \epsilon$ makes sure that no more inputs are expected to match L_1.

[2]For example, consider the grammar "$L \to \epsilon | cA | cB;\ A \to dL;\ B \to dL$" and the string $w = (cd)^n$. The number of traces is $O(2^n)$.

With the above definition of parse-tree edges, a parse tree is then a sequence of plain, call, or return edges. A parse forest is a sequence $[m_1, \ldots, m_n]$, where each m_i is a subset of $\mathcal{M}_{\text{pln}}$, $\mathcal{M}_{\text{call}}$, or $\mathcal{M}_{\text{ret}}$. In the following discussion, we use m_{pln}, m_{call}, and m_{ret} for an arbitrary subset of $\mathcal{M}_{\text{pln}}$, $\mathcal{M}_{\text{call}}$, and $\mathcal{M}_{\text{ret}}$, respectively.

Definition 5.2 (Parser PDA states and stack)**.** Given a VPG, we introduce a parser PDA, where a state, denoted as m, is a subset of $\mathcal{M}_{\text{pln}}$, $\mathcal{M}_{\text{call}}$, or $\mathcal{M}_{\text{ret}}$, and each element in the stack T is a subset of $\mathcal{M}_{\text{call}}$.

It is easy to see that a sequence of parser PDA states constitutes a parse forest.

Similar to the development of the VPG recognizer, we next define three derivative functions, denoted as p_c, $p_{\langle a}$, and $p_{b\rangle}$, to formalize how the parser PDA makes transitions. From the perspective of parse trees, each transition extends existing traces for parsing string w to new traces for parsing string wi, assuming i is the next input symbol. Notation-wise, we use the placeholder "_" to represent an entity whose value does not matter. For example, edge $(\_, \_, L) \in m$ is defined as $\exists L_1\ i,$ s.t. $(L_1, i, L) \in m$ or $\exists L_1, L_2, i, L$ s.t. $(((L_1, L_2), i, L) \in m$, where only L is of interest.

Definition 5.3 (Derivative functions)**.** 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.

(1) $p_c(m) = (m', \lambda T.T)$, where $m' = \{(L^u, c, L_1^u) \mid (\_, \_, L^u) \in m \wedge (L \to cL_1) \in P\}$.
To generate c, we consider each parse-tree edge in m. If it is of the form $(\_, \_, L^u)$, find every possible rule $L \to cL_1$ for some L_1. We then add edge (L^u, c, L_1^u) to the next state m'. Intuitively, if the current trace matches w and the nonterminal to parse is L, then with the extra c, the new trace matches wc and the new nonterminal to parse is L_1. Further, the boolean tag u is passed from L to L_1, since no matching rule is used at this step.

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

$$m' = \{(L^u, \langle a, L_1^{\text{true}}) \mid (\_, \_, L^u) \in m \wedge \exists b\rangle\ L_2,\ (L \to \langle aL_1 b\rangle L_2) \in P\} \cup \{(L^{\text{false}}, \langle a, L_1^{\text{false}}) \mid (\_, \_, L^{\text{false}}) \in m \wedge (L \to \langle aL_1) \in P\}.$$

If $(\_, \_, L^{\text{true}}) \in m$, a matching rule is waiting to be finished and we cannot use a pending rule such as $L \to \langle aL_1$. Thus, only a new matching rule can be used to generate $\langle a$. So it finds every possible rule $L \to \langle aL_1 b\rangle L_2$ and adds the parse-tree edge $(L^{\text{true}}, \langle a, L_1^{\text{true}})$ to m'. Notice that L_1 must be tagged with true. If $(\_, \_, L^{\text{false}}) \in m$, either a matching or a pending rule can be used. The matching-rule case is similar to the case when $u = \text{true}$. Further, it finds a rule like $L \to \langle aL_1$ for some L_1 and adds $(L^{\text{false}}, \langle a, L_1^{\text{false}})$ to m'. In addition, the new state m' is pushed to the stack to match a possible return symbol at a later point.

(3) $p_{b\rangle}(m, m_{\text{call}}) = (m', \text{tail})$, where $m_{\text{call}} = \text{head}\ T$ if $T \neq \bot$, and $m_{\text{call}} = \emptyset$ if $T = \bot$, and

$$m' = \{((L^u, L_1^{\text{true}}), b\rangle, L_2^u) \mid (L^u, \langle a, L_1^{\text{true}}) \in m_{\text{call}} \wedge \exists L_2, (L \to \langle aL_1 b\rangle L_2) \in P\} \cup \{(L^{\text{false}}, b\rangle, L_1^{\text{false}}) \mid (\_, \_, L^{\text{false}}) \in m \wedge (L \to b\rangle L_1) \in P\}.$$

Consider $(\_, \_, L^u) \in m$. If $u = \text{false}$, we must use a pending rule to generate $b\rangle$. Every rule $L \to b\rangle L_1$ is converted to edge $(L^{\text{false}}, b\rangle, L_1^{\text{false}})$ and added to m'. If $u = \text{true}$, intuitively we can only use a matching rule to generate $b\rangle$. The information of the last unfinished matching rule is stored in m_{call}, the top of the stack. For any $(L^u, \langle a, L_1^{\text{true}}) \in m_{\text{call}}$, it finds a rule $L \to \langle aL_1 b\rangle L_2$ for some L_2, and adds edge $((L^u, L_1^{\text{true}}), b\rangle, L_2^u)$ to the new state m'. The nonterminal L_2 inherits its tag from the tag of L. Note that in the above formulation the new state is generated based on only m_{call}, not the current state m; this design can generate invalid edges; a later pruner step will remove those invalid edges.

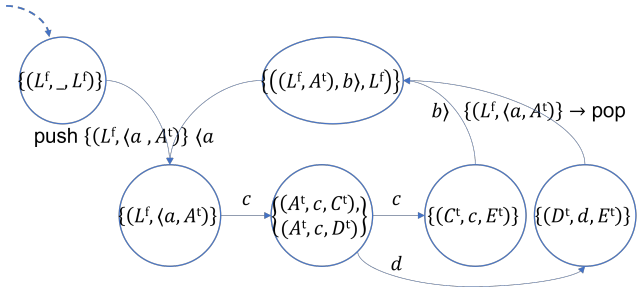


Fig. 5. The parser PDA for the grammar in Figure 2. We use f for false and t for true. Finally, the symbols and stack actions of $\langle a$ and $b\rangle$ are each shared by two transitions.

Constructing the parser PDA. Similar to constructing the recognizer PDA, the construction of the parser PDA is the least solution of the following equation.

$$\begin{aligned} A \quad = \quad & A \cup \{m' \mid c \in \Sigma_l \wedge m \in A \wedge p_c(m) = (m', f)\} \\ & \cup \{m' \mid \langle a \in \Sigma_c \wedge m \in A \wedge p_{\langle a}(m) = (m', f)\} \\ & \cup \{m' \mid b\rangle \in \Sigma_r \wedge m \in A,\ m'' \in (A \cap \mathcal{P}(\mathcal{M}_{\text{call}})) \cup \{\emptyset\} \wedge\ p_{b\rangle}(m, m'') = (m', f)\} \end{aligned}$$

Different from the case for VPG recognizers, $(A \cap \mathcal{P}(\mathcal{M}_{\text{call}})) \cup \{\emptyset\}$ is used for $p_{b\rangle}$, since a stack element for the parser PDA has to be a state generated for a call symbol and the stack can also be empty.

With the above equation, we can construct an algorithm for computing all parser PDA states and state transitions, similar to Algorithm 1. The differences are that (1) it starts with a helper state $m_0 = \{(L_0^{\text{false}}, \_, L_0^{\text{false}})\}$, where L_0 is the start nonterminal of the input VPG and _ stands for a dummy nonterminal, and (2) it uses parser derivative functions for deriving new states and transitions. The result is a parser PDA whose states are subsets of $\mathcal{M}_{\text{pln}}$, $\mathcal{M}_{\text{call}}$, or $\mathcal{M}_{\text{ret}}$, and transitions between states are labeled with (c, f), $(\langle a, f)$, or $(b\rangle, m_c, f)$, where c, $\langle a$, or $b\rangle$ is the next input symbol, m_c is the top stack element, and f is the stack action. Figure 5 shows the parser PDA generated by the algorithm for the grammar in Figure 2.

Given an input string, the parser PDA starts from $(m_0, \bot)$ and transitions to the next runtime configuration based on the following definition.

Definition 5.4 (Runtime transition for parser PDA)**.** Suppose the current configuration is (m, T) and the next input symbol is i. The *runtime transition function* $\mathcal{P}(i, m, T) = (m', T')$ of the parser PDA is defined as follows.

(1) if $i \in \Sigma_c \cup \Sigma_l$ and PDA edge (m, m') is marked with (i, f), then $\mathcal{P}(i, m, T) = (m', f(T))$;
(2) if $i \in \Sigma_r$, $T = m'' \cdot T'$, and PDA edge (m, m') is marked with (i, m'', f), then $\mathcal{P}(i, m, T) = (m', f(T))$.
(3) if $i \in \Sigma_r$, $T = \bot$, and PDA edge (m, m') is marked with $(i, \emptyset, f)$, then $\mathcal{P}(i, m, T) = (m', \bot)$.

5.3 The pruner PDA and the extractor

The parser PDA parses an input string w of length n and produces a state trace: $[m_1, \ldots, m_n]$, which can be viewed as a parse forest. As discussed earlier, it can contain invalid edges. For example, inside the last state m_n, a valid edge must be of the form $(\_, \_, L_1^{\text{false}})$ for some L_1 so that $(L_1 \to \epsilon) \in P$, signaling the end of parsing; other edges are invalid. Similarly, if the next symbol to match is a return symbol produced in a matching rule $L \to \langle a L_1 b\rangle L_2$, the parse-tree edge immediately before the one that produces the return symbol must end with L_1^{false} so that $(L_1 \to \epsilon) \in P$.

$$\frac{(L \to \epsilon) \in P}{L^u \Rightarrow (\epsilon, [])} \qquad \frac{(L \to cL_1) \in P \quad L_1^u \Rightarrow (w_1, v_1)}{L^u \Rightarrow \left(cw_1, \left(L^u, c, L_1^u\right)::v_1\right)}$$

$$\frac{(L \to \langle aL_1) \in P \quad L_1^{\text{false}} \Rightarrow (w_1, v_1)}{L^{\text{false}} \Rightarrow \left(\langle aw_1, \left(L^{\text{false}}, \langle a, L_1^{\text{false}}\right)::v_1\right)} \qquad \frac{(L \to b\rangle L_1) \in P \quad L_1^{\text{false}} \Rightarrow (w_1, v_1)}{L^{\text{false}} \Rightarrow \left(b\rangle w_1, \left(L^{\text{false}}, b\rangle, L_1^{\text{false}}\right)::v_1\right)}$$

$$\frac{(L \to \langle aL_1 b\rangle L_2) \in P \quad L_1^{\text{true}} \Rightarrow (w_1, v_1) \quad L_2^u \Rightarrow (w_2, v_2)}{L^u \Rightarrow \left(\langle aw_1 b\rangle w_2, \left[\left(L^u, \langle a, L_1^{\text{true}}\right)\right] + v_1 + \left[\left(\left(L^u, L_1^{\text{true}}\right), b\rangle, L_2^u\right)\right] + v_2\right)}.$$

Fig. 6. The big-step parse-tree derivation, assuming a VPG $G = (\Sigma, V, P, L_0)$.

After removing some invalid edges, earlier edges in the parse forest may become invalid. For instance, if pruning removes from m_n an edge $(L_2^{\text{false}}, \_, \_)$ and there are no other edges that start with L_2^{false} in the rest of m_n, then any m_{n-1} edge that ends with L_2^{false} can also be pruned, since it is not possible to connect it with an edge in the pruned m_n. This is a backward process. Therefore, our pruner PDA takes the reverse of the parse forest, $[m_n, m_{n-1}, \ldots, m_1]$, as the input and produces a pruned parse forest $[m'_n, m'_{n-1}, \ldots, m'_1]$ in reverse. Further, instead of intervening parsing and pruning steps, we choose to perform pruning after the parser PDA has finished so that every state in the parse forest gets pruned only once.

Definition 5.5 (Pruner PDA states and stack)**.** Given a VPG, we introduce a pruner PDA, where each state, denoted as m, is a subset of $\mathcal{M}_{\text{pln}}$, $\mathcal{M}_{\text{call}}$ or $\mathcal{M}_{\text{ret}}$, and each element of the stack T is a subset of $\mathcal{M}_{\text{ret}}$.

A stack is needed in the pruner to find valid call edges with respect to valid return edges in the stack. The technical details of the pruner PDA construction are introduced in Appendix C of the companion technical report (Jia et al. 2021). After pruning, we get a pruned parse forest $[m'_1, \ldots, m'_n]$ and an *extractor* is then used to extract parse trees from the forest. If the input VPG is unambiguous, at most one parse tree can be extracted. The definition of extract$([m'_1, \ldots, m'_n])$ detailed in Appendix C of the companion technical report (Jia et al. 2021) extracts a *parse-tree set* V, which is a set of parse trees together with corresponding stacks of call edges.

5.4 The correctness proof of the parsing algorithm

In this section, we discuss the correctness proof of our core VPG parsing algorithm; the proof is formalized in the proof assistant Coq. The correctness theorem is stated based on the relation $L^u \Rightarrow (w, v)$, meaning that input string w can be parsed from nonterminal L with tag u and one of the parse trees is v. We call this relation the *big-step* parse-tree derivation relation, which is presented in Figure 6. Its rules are mostly straightforward and we explain only the one for $L \to \langle aL_1 b\rangle L_2$: it first builds a parse tree for substring w_1 with L_1^{true} since w_1 must be a well-matched string; it then builds a parse tree for substring w_2 with L_2^u; then a parse tree for string $\langle aw_1 b\rangle w_2$ can be built by concatenating the parse-tree edge for $\langle a$, the parse tree for w_1, the parse-tree edge for $b\rangle$, and the parse tree for w_2.

Definition 5.6. Suppose v is a trace of parse-tree edges. We define firstNT(v) to be the starting nonterminal in the trace and lastNT(v) to be the last nonterminal in the trace. That is,

$$\text{firstNT}(v) = \{L^u \mid v = (L^u, \_, \_)::\_\}$$
$$\text{lastNT}(v) = \{L^u \mid v = \_ + [(\_, \_, L^u)]\}$$

$$\frac{(L \to cL_1) \in P \qquad v = [] \vee \text{lastNT}(v) = L^u}{(v, E) \xrightarrow{c} (v + [(L^u, c, L_1^u)], E)}$$

$$\frac{(L \to \langle aL_1) \in P \qquad v = [] \vee \text{lastNT}(v) = L^{\text{false}}}{(v, E) \xrightarrow{\langle a} (v + [(L^{\text{false}}, \langle a, L_1^{\text{false}})], (L^{\text{false}}, \langle a, L_1^{\text{false}}) \cdot E)}$$

$$\frac{(L \to \langle aL_1 b\rangle L_2) \in P \qquad v = [] \vee \text{lastNT}(v) = L^u}{(v, E) \xrightarrow{\langle a} (v + [(L^u, \langle a, L_1^{\text{true}})], (L^u, \langle a, L_1^{\text{true}}) \cdot E)}$$

$$\frac{(L \to b\rangle L_1) \in P \qquad v = [] \vee \text{lastNT}(v) = L^{\text{false}}}{(v, \bot) \xrightarrow{b\rangle} (v + [(L^{\text{false}}, b\rangle, L_1^{\text{false}})], \bot)}$$

$$\frac{(L \to \langle aL_1 b\rangle L_2) \in P \qquad \text{lastNT}(v) = L_3^{\text{true}} \qquad (L_3 \to \epsilon) \in P}{(v, (L^u, \langle a, L_1^{\text{true}}) \cdot E) \xrightarrow{b\rangle} (v + [((L^u, L_1^{\text{true}}), b\rangle, L_2^u)], E)}$$

$$\frac{(L_3 \to \langle aL_4) \in P \qquad (L_1 \to b\rangle L_2) \in P \qquad v = [] \vee \text{lastNT}(v) = L_1^{\text{false}}}{(v, (L_3^{\text{false}}, \langle a, L_4^{\text{false}}) \cdot E) \xrightarrow{b\rangle} (v + [(L_1^{\text{false}}, b\rangle, L_2^{\text{false}})], E)}$$

Fig. 7. The small-step parse-tree derivation, given a VPG $G = (\Sigma, V, P, L_0)$.

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

Fig. 8. The transitive closure of the small-step.

Theorem 5.1 (Correctness of VPG parsing). *For an input string w of length n, if the parser PDA for VPG $G = (\Sigma, V, P, L_0)$ starts with the initial configuration $m_0 = \{(L_0^{false}, \_, L_0^{false})\}$ and $T_0 = \bot$ and traverses the following configurations to parse w: $(m_0, T_0) \xrightarrow{w_1} (m_1, T_1) \cdots \xrightarrow{w_n} (m_n, T_n)$, and* $\text{extract}([m_1, \ldots, m_n]) = V$, *then*

$$\forall v,\ L_0^{false} \Rightarrow (w, v) \ \textit{iff}\ \left(\exists E, (v, E) \in V \wedge \exists L_1,\ \text{lastNT}(v) = L_1^{false} \wedge (L_1 \to \epsilon) \in P\right).$$

To prove the theorem, we need the help of a *small-step* parse-tree derivation relation so that we can formalize a set of invariants that are satisfied during each step when running the parser PDA. The relation $(v, E) \xrightarrow{i} (v', E')$, defined in Figure 7, means that starting with a parse tree v and a stack of call edges E, the parsing of symbol i results in a new parse tree v' and a new stack of call edges E'. In all rules, v' is the result of adding one new parse-tree edge to v, and therefore it formalizes the process of generating one parse-tree edge at a time, matching what the parser PDA does. The transitive closure of the small-step relation is in Figure 8. The following two theorems show the equivalence of big-step and small-step parse-tree relations.

Theorem 5.2 (From big step to small step).

(1) If $L^{true} \Rightarrow (w, v)$*, then* $([], \bot) \xrightarrow{w}{}^{*} (v, \bot)$.
(2) If $L^{false} \Rightarrow (w, v)$*, then* $\exists E$*, s.t.* $([], \bot) \xrightarrow{w}{}^{*} (v, E)$.

Theorem 5.3 (From small step to big step). *If* $([], \bot) \xrightarrow{w}{}^{*} (v, E)$*, and* $\text{firstNT}(v) = L^{false}$, $\text{lastNT}(v) = L_1^{false}$ *and* $(L_1 \to \epsilon) \in P$*, then* $L^{false} \Rightarrow (w, v)$.

With the small-step relation, we can formalize those invariants satisfied by every step when running the parser PDA. Suppose the parser PDA has consumed string w' to reach configuration (m', T') and consumes symbol i next to reach (m, T). Further, the extractor (Definition C.7 in Appendix C of the companion technical report (Jia et al. 2021)) can extract a parse-tree set V' from the parser PDA state traces for w'. Finally, extending V' with one-more parse-tree edge in m to get V. Then the following theorem can be proved: if the previous step satisfies $\text{Inv}(L, w', m', T', V')$, the latest step should also satisfy $\text{Inv}(L, w, m, T, V)$, where $w = w'i$. The invariants are defined as follows.

Property 5.4 (Invariants of VPG parsing). *Let* L *be a nonterminal,* w *an input string,* (m, T) *a parser PDA configuration, and* V *a parse-tree set. The property* $\text{Inv}(L, w, m, T, V)$ *is defined as*

(1) $\forall v\ E, (v, E) \in V$ *if and only if* $([], \bot) \xrightarrow{w}{}^{*} (v, E) \wedge \text{firstNT}(v) = L$*;*
(2) $\forall (v, E) \in V,\ (E = \bot \to T = \bot) \wedge (\exists e\ E', E = e \cdot E' \to e \in \mathit{head}\ T)$*;*
(3) $\forall v, \exists E, (v, E) \in V \to \exists e, e \in m \wedge v = \_ + [e]$.

The above correctness proof is formalized in Coq and includes around 3k lines of proofs for the correspondence between the big-step and the small-step parse-tree derivations and another 4k lines for implementing the parser and the parse-tree extractor, formalizing the invariants, and proofs for showing that the invariants are preserved during parsing.

5.5 Time and space complexity

When given an input of length n, the VPG parser runs two PDAs to construct the parse forest: a forward parser PDA and a backward pruner PDA. We assume their transition tables can be implemented via a data structure that provides constant-time lookups (e.g., via a hash table). Therefore, each transition can finish in constant time, leading to the linear-time running of VPG parsing.

The space complexity depends on the space for representing the transition tables of the two PDAs. Recall that the transition function of the parser PDA is $\mathcal{P}(i, m, T) = (m', T')$, where $i \in \Sigma$, m and m' are states, and T and T' are stacks. The transition function of the pruner PDA is $\mathcal{G}(m_1, m_2', T) = (m_1', T')$, where m_1, m_2', and m_1' are states, and T and T' are stacks of states. In fact, only the top of the stack is used by $\mathcal{P}$ and $\mathcal{G}$. Note since a state is a set of edges and an edge corresponds to a rule, the size of a state is at most $O(|P|)$. Thus, there are at most $O\left(2^{|P|}\right)$ states. So the total number of transitions in the two PDAs is bounded by $O\left(2^{3|P|} \times |\Sigma| + 2^{4|P|}\right)$, where Σ is the input alphabet. Each entry in the transition table occupies $O\left(\log |\Sigma| + |P|\right)$ bits. As a conclusion, the space complexity is $O\left(\left(\log |\Sigma| + |P|\right) \times \left(2^{3|P|} \times |\Sigma| + 2^{4|P|}\right)\right)$, which is exponential in $|P|$. However, this is the worst-case scenario as not all states can be derived; further, it is independent of the input string size.

In our evaluation, the largest space occupied by the transition tables is around 1.6 MB, for an HTML grammar (discussed in Section 7.3).

6 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.

6.1 Embedding semantic actions

Semantic actions transform parsing results to user-preferred formats. In a rule $L \rightarrow s_1 \cdots s_k$, 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

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

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, 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.

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

In the above example, a semantic action is specified after each rule, e.g., "$@\{\text{let } f_1\ v_1\ v_2 = 1 + v_2\}$". 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.

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.

6.2 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 ($\Sigma = \Sigma_l \cup \Sigma_c \cup \Sigma_r$);[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. Appendix D of the companion technical report (Jia et al. 2021) shows some example tagged CFGs.

However, 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. For simplicity, we assume every call symbol is matched with a return symbol in the input tagged CFG.

The translation steps are summarized as follows:

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

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

At a high level, a tagged CFG is first translated to a simple form, upon which validation is performed. If validation passes, the simple-form CFG is translated to a linear-form CFG, which is finally translated to a VPG. We next detail these steps.

Definition 6.1 (Simple forms)**.** A rule is in the simple form if it is of the form $L \rightarrow \epsilon$, or of the form $L \rightarrow s_1 \cdots s_k$, where $s_i \in \Sigma_l \cup V$ or $\exists \langle a, b\rangle, L_i$, s.t. $s_i = \langle aL_ib\rangle$, $i = 1..k$, $k \geq 1$. 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 $\langle asb\rangle$, where $\langle a$ is matched with $b\rangle$ and $s \in (\Sigma \cup V)^*$, with $\langle aL_sb\rangle$ and generate a new nonterminal L_s and a new rule $L_s \rightarrow s$. After this conversion, a string in the from of $\langle aLb\rangle$ can be viewed as a "plain symbol"; this is a key intuition for the following steps. We call $\langle aLb\rangle$ a *matched token* in the following discussion.

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

Definition 6.2 (Dependency graphs)**.** The dependency graph of a grammar $G = (V, \Sigma, P, L_0)$ is (V, E_G), where $E_G = \{(L, L') \mid \exists s_1, s_2 \in (\Sigma \cup V)^*, \text{s.t. } (L \rightarrow s_1L's_2) \in P\}$.

The validator checks for every loop in the dependency graph, either (1) in the loop there is an edge (L, L') that is produced from a rule of the form $L \rightarrow s_1\langle aL'b\rangle s_2$, where $s_1, s_2 \in (\Sigma \cup V)^*$; or (2) every edge (L, L') in the loop is produced from a rule of the form $L \rightarrow sL'$, $s \in (\Sigma \cup V)^*$ and at least one edge in the loop satisfies $s \not\rightarrow^* \epsilon$.

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

Definition 6.3 (Linear forms)**.** A rule is in the linear form if it is in one of the following forms: (1) $L \rightarrow \epsilon$; (2) $L \rightarrow t_1 \cdots t_k$; (3) $L \rightarrow t_1 \cdots t_kL'$; where $t_i \in \Sigma_l$ or $\exists \langle a, b\rangle, L_i$, s.t. $t_i = \langle aL_ib\rangle$, $i = 1..k$, $k \geq 1$. 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 s_i can be a nonterminal. Further, the linear form allows rules of the form $L \rightarrow t_1 \cdots t_kL'$, 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. Appendix E of the companion technical report (Jia et al. 2021) shows the translation algorithm.

The translation from a linear-form CFG to a VPG is simple. E.g., for a rule of the form $L \rightarrow t_1 \cdots t_k$, it is translated to $L \rightarrow t_1L_1; L_1 \rightarrow t_2L_2; \ldots; L_k \rightarrow t_kL_k; L_k \rightarrow \epsilon$, where L_1 to L_k are a set of new nonterminals.

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. We further note that not all tagged CFGs can be converted to VPGs. For example, grammar "$L \rightarrow cLc|\epsilon$" 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 "$L \rightarrow Lc|\epsilon$" is rejected by the validator since it is left recursive. However, it can be first refactored to "$L \rightarrow cL|\epsilon$", 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 $L \rightarrow AbCd$ is attached with L^4, written as $L \rightarrow AbCd\ @L^4$. As mentioned in Section 6.1, 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 $L \longrightarrow s$, 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 $L_s \longrightarrow s$ is generated, we do not generate a semantic value for L_s but leave the semantic values for s on the stack so that 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. Appendix F of the companion technical report (Jia et al. 2021) shows an example of the translation.

7 EVALUATION

We implemented our VPG parsing library in OCaml. The implementation used hash tables to store the transition tables of the generated parser and pruner PDAs to get constant-time lookup. 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[4]. 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 ANTLR4[5], a popular parser generator that implements an efficient parsing algorithm called ALL(*) (Parr et al. 2014). 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. 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[6] to tagged CFGs, generated VPG parsers, and compared VPG parsers with the parsers generated by ANTLR in performance. Appendix D of the companion technical report (Jia et al. 2021) shows the tagged CFGs for JSON, XML and HTML.
(2) When comparing with ANTLR, we compared only the parsing time, omitting the lexing time. This is because we used ANTLR's lexers to generate the tokens for both VPG parsers and ANTLR parsers.

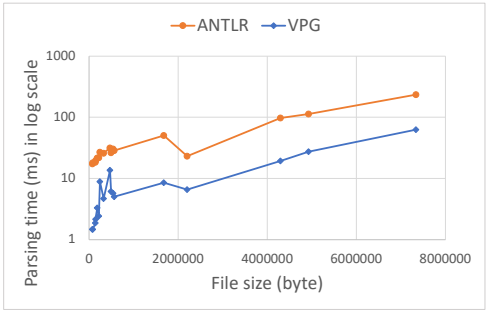
7.1 Comparison with ANTLR on parsing JSON files

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 treats "}" and "]" as return symbols.

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

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

[4] https://github.com/antlr/grammars-v4
[5] https://www.antlr.org/.
[6] https://github.com/antlr/grammars-v4/blob/master/json/, https://github.com/antlr/grammars-v4/blob/master/xml/, and https://github.com/antlr/grammars-v4/tree/master/html



Name	Size	ANTLR	VPG	Conv
JSON.parse	7.0 MB	235 ms	63 ms	61 ms
airlines	4.7 MB	113 ms	27 ms	27 ms
educativos	4.1 MB	98 ms	19 ms	19 ms
canada	2.1 MB	23 ms	15 ms	20 ms
citm_catalog	1.6 MB	50 ms	9 ms	8 ms

Fig. 9. Parsing times of JSON files (in log scale on the left).

For evaluation, we collected 23 real-world JSON files from the awesome-json repository, the nativejson benchmarks, and the `JSON.parse` benchmarks[7]. The sizes of the files range from 14 KB to 7 MB. The parsing times are shown in Figure 9; note that the y-axis of the left figure (and other figures in this section) is in the log scale for better visualization. As can be seen, the VPG parser runs much faster than ANTLR. The right of Figure 9 shows the VPG parsing times for the 5 largest files in our test set; the VPG column is the amount of time cost by running the parser and pruner PDAs. On those large files, VPG parsing is about 4 times faster than ANTLR. For smaller files, the gap is even larger; Appendix G of the companion technical report (Jia et al. 2021) shows the results for the full test set.

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 (as well as the XML and HTML grammars), the parse forest is really the encoding of a single parse tree. The algorithm of how to convert a VPG parse tree to a stack machine and how to evaluate the stack machine have been discussed in Section 6. The result of the evaluation is a structure that can be directly printed out and compared with; the same applies to the ANTLR parse tree[8]. The conversion steps are summarized as follows.

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

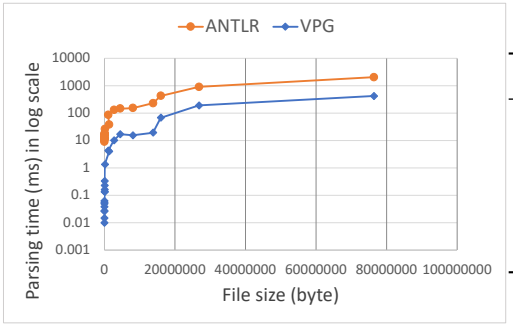
Note that in practice this conversion may not be necessary. A downstream application can directly work on the VPG parse tree. We include the time to show the conversion time for our VPG parser to work directly with legacy downstream applications. The time of conversion is shown in the "Conv" column on the right hand side of Figure 9.

7.2 Comparison with ANTLR on 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 tokens: <, p, and >. Those tokens then appear in the ANTLR XML grammar. Part of the reason for this design is because the XML format allows additional attributes within a tag; e.g.,

[7] https://github.com/jdorfman/awesome-json-datasets, https://github.com/miloyip/nativejson-benchmark, and https://github.com/GoogleChromeLabs/json-parse-benchmark

[8] By the "ANTLR parse tree", we mean the string output by the ANTLR parser with the option "-tree".



Name	Size	ANTLR	VPG	Conv
po	73 MB	2058 ms	425 ms	1070 ms
cd	26 MB	913 ms	192 ms	455 ms
address	15 MB	429 ms	68 ms	139 ms
SUAS	13 MB	232 ms	19 ms	13 ms
ORTCA	7.7 MB	154 ms	16 ms	6 ms

Fig. 10. Parsing times of XML files (in log scale on the left).

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* '/>' ;
```

To expose the nesting structure within XML, we add an additional step between ANTLR lexing and VPG parsing.

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

The step of VPG lexing coalesces tokens for a single XML tag into a single token. For example, <p> becomes a single token and is marked as a call symbol. For attributes inside tags, they are processed and attached as tags' semantic values for the following parsing step. The following shows a snippet of our adapted XML grammar.

```
element : <TagOpen content TagClose> | TagSingle ;
```

The VPG tokens are declared as follows.

```
TagOpen = '<' Name attribute* '>' ;
TagClose = '<' '/' Name '>' ;
TagSingle = '<' Name attribute* '/>' ;
```

For evaluation, we used the real-world XML files provided by the VTD-XML benchmarks[9], which consist of a wide selection of 23 files ranging from 1K to 73MB. The parsing times are presented in Figure 10; Appendix G of the companion technical report (Jia et al. 2021) shows the results for the full test set. Similar to JSON, VPG parsing on XML files is much faster than ANTLR parsing; on the 5 largest XML files, VPG parsing is about 5 times faster; on smaller files, the gap is even larger.

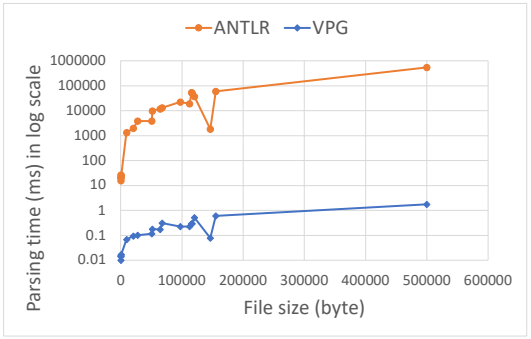
7.3 Comparison with ANTLR on parsing HTML files

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
. 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 according to the HTML standard. Although this kind of tags is also "self-closing", we will use the terminology

[9] https://vtd-xml.sourceforge.io/2.3/benchmark_2.3_parsing_only.html.



Name	Size	ANTLR	VPG	Conv
youtube	489 KB	543 s	1.8 ms	2.3 ms
digg	152 KB	60 s	0.6 ms	0.7 ms
cnn1	118 KB	37 s	0.5 ms	0.5 ms
reddit2	114 KB	52 s	0.3 ms	0.6 ms
reddit	114 KB	54 s	0.3 ms	0.6 ms

Fig. 11. Parsing times of HTML files (in log scale on the left).

of optional end tags since that is how the official HTML5 standard describes it. As will be shown in our experimental data, the complexity in this grammar makes ANTLR's parsing of HTML files extremely slow.

Similar to the XML case, we introduced a VPG lexer to coalesce tokens for a single tag into a single token. However, the optional end-tags introduce additional complexity. To explain, let us first examine the relevant part of the VPG grammar:

```
htmlElement = TagPlain | <TagOpen htmlElement TagClose> | TagSingle ;
```

The VPG tokens are declared as follows.

```
TagPlain = '<' TAG_NAME htmlAttribute* '>' ;
TagOpen = '<' TAG_NAME htmlAttribute* '>' ;
TagClose = '<' '/' TAG_NAME '>' ;
TagSingle = '<' TAG_NAME htmlAttribute* '/' '>' ;
```

`TagPlain` is for HTML tags that cannot have matching end-tags, and `TagSingle` is for self-closing tags. The VPG lexer first merges ANTLR tokens related to a single tag, and then determines which tags are call symbols, return symbols, and plain symbols. A start tag with no matching end tag is marked as a plain symbol in this process. This is implemented with a straightforward method: the first k open HTML tags are matched with the last k close HTML tags, and the rest open HTML tags are viewed as plain symbols, where k is the number of close HTML tags in the file (the number of close tags is always less than or equal to the number of open tags).

For evaluation, we used the 19 real-world HTML files provided in ANTLR's repository[10]. The parsing times are presented in Figure 11. The conversion times of the parse trees are shown in the "Conv" column. As we can see, our VPG parser significantly outperforms the ANTLR parser, with more than 4 orders of magnitude of difference. We emphasize that in our evaluation the VPG parser and the ANTLR parser accept the same HTML files, and they produce the same parse trees with the help of a converter.

From the ANTLR profiling tool, we found that around 99% time cost by ANTLR is in the prediction of

```
(htmlContent TAG_OPEN TAG_SLASH TAG_NAME TAG_CLOSE)?
```

in the rule of "htmlElement", which triggers many lookahead symbols, and also many DFA cache misses (around a miss rate of 90%). Appendix H of the companion technical report (Jia et al. 2021)

[10] https://github.com/antlr/grammars-v4/tree/master/html/examples

Table 1. Parsing times of 5 largest JSON files. “SpiderM” stands for “SpiderMonkey”, and "JSCore" for “JavaScriptCore”.

Name	Size	ANTLR Lex	VPG Parse	Lex+Parse	SpiderM	JSCore	V8	Chakra
JSON.parse	7.0 MB	130 ms	63 ms	193 ms	118 ms	139 ms	76 ms	88 ms
airlines	4.7 MB	81 ms	27 ms	108 ms	74 ms	95 ms	42 ms	56 ms
educativos	4.1 MB	108 ms	19 ms	128 ms	71 ms	421 ms	45 ms	49 ms
canada	2.1 MB	45 ms	15 ms	60 ms	57 ms	68 ms	34 ms	44 ms
citm_catalog	1.6 MB	39 ms	9 ms	47 ms	34 ms	71 ms	28 ms	25 ms

Table 2. Parsing times of 5 largest XML files. “HP2” stands for “HTMLParser2”.

Name	Size	ANTLR Lex	VPG Parse	Lex+Parse	Fast-XML	Libxmljs	SAX-JS	HP2
po	73 MB	1812 ms	425 ms	2238 ms	3278 ms	897 ms	6618 ms	1827 ms
cd	26 MB	732 ms	192 ms	924 ms	1298 ms	419 ms	2103 ms	735 ms
address	15 MB	367 ms	68 ms	435 ms	584 ms	196 ms	1012 ms	331 ms
SUAS	13 MB	237 ms	19 ms	256 ms	254 ms	182 ms	1214 ms	169 ms
ORTCA	7.7 MB	142 ms	16 ms	157 ms	138 ms	91 ms	665 ms	89 ms

shows the profiler result for “bbc.com.html”. This HTML evaluation shows the power of VPGs in designing practical language parsers, due to their ability of linear-time parsing.

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

7.4 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 in Section 7.1. For XML, we compared with four popular XML parsers (fast-xml-parser, libxmljs, sax-js, and htmlparser2) [11], and evaluated them with the XML files in Section 7.1.

The evaluation results for the largest files are shown in Table 1 and Table 2; the full results are in Appendix G of the companion technical report (Jia et al. 2021). 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”), and the combined time (column “Lex+Parse”). From the results, we can see that although the total time of VPG parsing is not the shortest among all parsers, the parsing time alone is. Thus, VPG parsers show promising potential in performance, with additionally verified correctness over hand-crafted parsers. The total parsing time can be reduced by replacing ANTLR’s lexer with a faster, customized lexer, since the parsing time of VPG

[11] https://github.com/NaturalIntelligence/fast-xml-parser#readme, https://github.com/libxmljs/libxmljs, https://github.com/isaacs/sax-js, and https://github.com/fb55/htmlparser2.

is shorter than the lexing time. Also, combining the lexing and parsing steps, as is common in hand-crafted parsers, can usually improve the overall time.

8 LIMITATIONS AND 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 Coq formalization of and its implementation in OCaml. First, the implementation takes tagged CFGs as input and translates tagged CFGs to VPGs; this translation algorithm has not been formally modeled and verified in Coq. 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. These gaps prevent us from directly extracting OCaml code from the Coq formalization.

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 kind of 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 $\{a^i b^j c^k \mid i \neq j \vee j \vee k\}$). 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.

The parsing in our VPG framework is performed in three steps: construction of a parse forest (with possible invalid edges), pruning of the parse forest, and extraction of parse trees. This design simplifies formal verification but adds a pruning step. We believe the pruning step can be possibly removed by redesigning our parser generator, where the recognizer is extended to a transducer that generates semantic actions and the execution of those semantic actions builds the parse forest directly.

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.

9 CONCLUSIONS

In this paper, we present a recognizer and a formally verified parser generator for visibly pushdown grammars. The parsing algorithm is largely enlightened by the recognizer, with several trade-offs to simplify the structure and reduce the burden of formal verification. 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 speed-up.

ACKNOWLEDGMENT

The authors would like to thank anonymous reviewers for their insightful comments. This work was supported by DARPA research grant HR0011-19-C-0073.

REFERENCES

Rajeev Alur and P. Madhusudan. 2009. Adding Nesting Structure to Words. *Journal of the Association for Computing Machinery* 56, 3 (May 2009), 16:1–16:43.

Janusz A. Brzozowski. 1964. Derivatives of regular expressions. *J. ACM* 11 (1964), 481–494.

John Cocke. 1969. *Programming Languages and Their Compilers: Preliminary Notes.* New York University, USA.

F. L. Deremer. 1969. *PRACTICAL TRANSLATORS FOR LR(K) LANGUAGES.* Technical Report. Cambridge, MA, USA.

Jay Earley. 1970. An Efficient Context-free Parsing Algorithm. *Commun. ACM* 13, 2 (Feb. 1970), 94–102.

Romain Edelmann, Jad Hamza, and Viktor Kuncak. 2020. Zippy LL(1) parsing with derivatives. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 1036–1051.

Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *ACM Symposium on Principles of Programming Languages (POPL)*. 111–122.

Ian Henriksen, Gianfranco Bilardi, and Keshav Pingali. 2019. Derivative grammars: a symbolic approach to parsing with derivatives. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–28.

Xiaodong Jia, Ashish Kuma, and Gang Tan. 2021. A Derivative-based Parser Generator for Visibly Pushdown Grammars. arXiv:2109.04258 [cs.PL]

Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) Parsers. In *European Symposium on Programming (ESOP)*. 397–416.

Tadao Kasami. 1965. *An Efficient Recognition and Syntax-Analysis Algorithm for Context-Free Languages.* Technical Report. Air Force Cambridge Research Laboratory.

Adam Koprowski and Henri Binsztok. 2010. TRX: A Formally Verified Parser Interpreter. *Logical Methods in Computer Science* 7 (2010).

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)*.

Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. 2021. CoStar: a verified ALL(*) parser. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 420–434.

Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with derivatives: a functional pearl. In *ACM International Conference on Functional programming (ICFP)*. 189–195.

Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: Better, Faster, Stronger SFI for the x86. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 395–404.

Scott Owens, John Reppy, and Aaron Turon. 2009. Regular-expression derivatives re-examined. *J. Funct. Program.* 19 (March 2009), 173–190. Issue 2.

Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) parsing: the power of dynamic analysis. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 579–598.

Theofilos Petsios, Adrian Tang, Salvatore J. Stolfo, Angelos D. Keromytis, and Suman Jana. 2017. NEZHA: Efficient Domain-Independent Differential Testing. In *IEEE Symposium on Security and Privacy (S&P)*. 615–632.

Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *Usenix Security Symposium*. 1465–1482.

Daniel H. Younger. 1967. Recognition and Parsing of Context-Free Languages in Time n^3. *Information and Control* 10, 2 (1967), 189–208.