A Derivative-based, Colored-edged Parser Generator for Nested Words

BAS MARCELIS, University of Twente, The Netherlands

The language class of *nested words* lies between regular- and context-free languages on the Chomsky hierarchy. It combines the expressiveness of hierarchy from context-free languages with the robustness of regular languages. An optimised parser generator for this language would be promising, as the nesting of words can be seen in multiple popular programming languages, such as HTML, JSON, and XML, proving the popularity of its applications. This research combines a derivative-based parser with colored nested words to provide a parser generator for well-matched VPGs that accepts ambiguity and returns all possible parse trees, while also having proper error handling by accepting pending calls. The performance of a prototype based on these findings is linear complexity for unambiguous grammars and linear complexity for every possible parse tree for ambiguous grammars, which is proven in both theory and practice.

Additional Key Words and Phrases: Nested words, Visibly Pushdown Automata, VPA, NWA, Parser generator, Derivative-based Parsing, Colored Nested Words

1 INTRODUCTION

In 2004, Alur and Madhusudan proposed the language class of *nested words* [2] which lies between regular- and context-free languages on the Chomsky hierarchy [6]. *Nested words* describe regular grammars with nesting as its only non-regular property. This intermediate class combines the expressiveness of hierarchy from context-free languages with the robustness of regular languages. An example of a use case of nested words can be seen in HTML.

Example Grammar 1.

$$S \Longrightarrow < p > S^* < /p >$$

Example Grammar 1 defines a grammar rule which nests multiple statements (S) as a paragraph. In this case, $< p >$ and $< /p >$ indicate the start and end of the nesting of the paragraph, respectively. Nesting of words does not limit itself to HTML, this dual linear-hierarchical structure is present more often, including in executions of structured programs, annotated linguistic data, XML, JSON, Markdown, and more [3]. These languages that contain balanced strings of opening and closing symbols are formally called Dyck languages [7].

Synonymous to nested word languages are *Visibly Pushdown Languages* (VPLs), described by *Visibly Pushdown Grammars* (VPGs) and modelled by *Visibly Pushdown Automata* (VPAs) [2]. VPAs have the same complexity for decision problems as pushdown automata for context-free grammars, while being more compact [3], which makes them more applicable to nested word languages. Resultingly, an optimised nested words parser generation can provide improved efficiency and robustness to languages of its class compared to the use of a CFG parser generator. Currently, there is only one publicly available parser generator for nested words grammars named OWL [9], which was found to fail to deliver its promised near-linear performance [15]. Resultingly, while the use cases of nested words parser generators seem promising, there are no *optimised* publicly available parsers that support this.

This paper considers an *optimised* parser generator to be *flexible* and *efficient*. First of all, flexibility is considered as accepting ambiguous grammars and inputs and returning a parse forest with all possible derivations. This paper aims to achieve this using a derivative-based approach, which will be discussed in Section 3. Consequently, the first research question is the following:

> **RQ1**: How can a derivative-based parser generator extract all possible derivations from well-matched VPGs?

Well-matched VPGs are in a normal form which is similar to the Greibach normal form (GNF) [8] but specifically for VPGs. In this form, opening and closing symbols of scopes have to be defined in the same rule. This form will be further defined in Section 2.

To further increase flexibility, this paper aims to provide error handling, since nested words languages are notoriously prone to forgetting closing tags. Accordingly, the second research question is the following:

> **RQ2**: How can error handling be designed to repair some common defects automatically?

Efficiency is considered as parsing with linear complexity for unambiguous grammars and linear complexity for every possible derivation for ambiguous grammars. Therefore, the third research question is the following:

> **RQ3**: What is the best possible performance of the generated parser?

The framework provided in theory in this paper is also brought into practice by a Java implementation [12], to further back its efficacy. All definitions, algorithms, etc. discussed in the paper can be found in practice in the implementation, which can be accessed by following the link in the bibliography [12]. Additionally, the implementation is used to determine practical performance, which can be compared to its theoretical performance.

This paper is structured as follows. Section 2 discusses the formal definition of nested words and their corresponding automata. Additionally, its subsection describes the grammar form for nested word grammars, as used in this paper. Then, Section 3 describes the derivative-based framework used for nested words parsing. It discusses parsing with derivates, the PDA framework (Subsection 3.1), the recognizing PDA (Subsection 3.2), parsing into a parse forest (Subsection 3.3), pruning the Parse Forest (Subsection 3.4), and, finally, extracting all Parse Trees (Subsection 3.5). Then, Section 4 describes an intuitive approach to handling pending calls. Section 5 describes the performance of the provided frameworks, both in theory and in practice. Then finally, Section 6 concludes and Section 7 discusses the limitations of the framework and potential future work.

TScIT 38, February 3, 2023, Enschede, The Netherlands

2 DEFINITION OF NESTED WORDS

VPGs define nesting with a specific call and return symbol, which can be used to control the stack of the VPA. Therefore, the alphabet of a VPL is partitioned into three disjoint finite alphabets: Σ_c is a finite set of *call* symbols which open the nesting and can push the stack, Σ_r is a finite set of *return* symbols which close the nesting and can pop the stack, and Σ_{int} is a finite set of internal symbols which do not correspond to any nesting or action on the stack [2]. The entire VPL alphabet is the union of these three disjoint sets. Additionally, the stack is limited to a finite stack alphabet. These are the alphabets used by the VPA, which is formally defined by Alur et al. as the following.

Definition 2.1. A **Visibly Pushdown Automaton** (VPA) on finite words over $\langle \Sigma_c, \Sigma_r, \Sigma_{int} \rangle$ is a tuple $M = (Q, Q_{in}, \Gamma, \delta, Q_F)$ where Q is a finite set of states, $Q_{in} \subseteq Q$ is a set of initial states, Γ is a finite stack alphabet that contains a special bottom-of-stack symbol $\perp$, $\delta \subseteq (Q \times \Sigma_c \times Q \times (\Gamma \backslash \{\perp\})) \cup (Q \times \Sigma_r \times Q \times \Gamma) \cup (Q \times \Sigma_{int} \times Q)$, and $Q_F \subseteq Q$ is a set of final states [2, p. 204].

Transitions in the VPAs are as follows.

- $(q, c, q', \gamma) \subseteq (Q \times \Sigma_c \times Q \times (\Gamma \backslash \{\perp\}))$ is a call transition, going from state q to q', consuming $c \in \Sigma_c$, pushing $\gamma \in (\Gamma \backslash \{\perp\})$ to the stack.
- $(q, c, q', \gamma) \subseteq (Q \times \Sigma_r \times Q \times \Gamma)$ is a return transition, going from state q to q', consuming $c \in \Sigma_r$, popping $\gamma \in \Gamma$ to the stack.
- $(q, c, q') \subseteq (Q \times \Sigma_{int} \times Q)$ is an internal transition, going from state q to q', consuming $c \in \Sigma_c$, without a stack action, as the symbol is an internal symbol.

Additionally, the accepting states $Q_F \subseteq Q$ in a VPA can be limited to the stack being empty, but do not have to be. Some frameworks require an empty stack to accept an input [10], while others omit this requirement to accept unmatched calls [14]. The framework provided in this paper requires an empty stack for acceptance because it handles pending calls in an alternative method, as will be discussed in Section 4.

The main difference between nested words grammars and CFGs is that in nested words grammars the stack is limited to a stack vocabulary and only specified (call and return) symbols can control the stack. Then, a created PDA can be much more succinct, as during the construction it is known which symbols can cause stack actions for every state in the PDA.

Finally, the class of VPLs, modelled by VPAs, is closed under union, intersection, complementation, renaming, concatenation, and reflexive-transitive closure [2].

2.1 Well-Matched Grammars

Grammar Specifications for VPLs can be either *well-matched*, where the call and return symbol are required to be in the same rule, or *general*, where pending rules are also accepted. The framework provided in this paper requires well-matched grammars, to keep it clear which symbols are internal, call, or return and which call and return symbols are paired. Rules in well-matched VPGs can be defined as a GNF-like form [8], where every nested words grammar can be transformed to it. Therefore, well-matched VPGs with their GNF-like form are defined as follows:

Definition 2.2. A **well-matched VPG** (V, Σ, P, L_0) is a well-matched VPG with respect to the partitioning $\Sigma = \Sigma_{int} \cup \Sigma_c \cup \Sigma_r$, if every production rule in P is in one of the following forms [10, p. 151:3].

(1) $L \Longrightarrow \epsilon$, where ϵ stands for the empty string
(2) $L \Longrightarrow cL_1$, where $c \in \Sigma_{int}$ and $L_1 \in V$
(3) $L \Longrightarrow \langle aL_1 b \rangle L_2$, where $\langle a \in \Sigma_c, b \rangle \in \Sigma_r, L_1 \in V$, and $L_2 \in V$

This form of well-matched VPGs is used in the rest of the paper to represent VPGs and is a required form for the construction of the PDA and for parsing.

3 DERIVATIVE-BASED PARSING

Derivative parsing was introduced in 1964 by Brzozowski [4]. Languages are then seen as a set of all possible words, which get repeatedly "filtered" and "chopped" by a derivative function [13]. The derivative function does this by considering a symbol and returning all remaining possible words in the language after consuming that symbol. The formal definition of the derivative functions is:

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

Where, for example:

$$\delta_a\{aaa, abc, bbb\} = \{aa, bc\}$$

To parse an input, this derivative function is repeated for every character in the input. If the final language contains the empty string, then the input word is recognized.

Brzozowski introduced derivatives for regular languages, but recently Might et al. [13] extended this to CFGs and added laziness, memoization and fixed points to improve efficiency. Due to the power of the simplicity of this derivative framework, it handles ambiguity, left-recursion, and right-recursion.

However, the framework in this paper, based on Jia et al. [10] is not exactly the same as parsing with derivatives, but is *derivative-based*. It accepts ambiguity, but, as it requires VPGs to be in the form as described previously, it does not support left recursion. This *derivative-based* framework creates a PDA where all states and transitions are based on the derivation on all *configurations*.

Definition 3.1. A **Configuration** is (S, T), where S is the current state and T is the current stack. A configuration represents a language; it represents the input words that can reach that configuration in the PDA.

The derivative function $\delta_c(S, T)$ takes the derivative of that configuration for a character c, resulting in the set of remaining configurations. Thus, this framework is *derivative-based*, in the sense that, using this derivative function on configurations, a PDA can be constructed by repeatedly considering all newly derived configurations, and creating corresponding states and transitions.

3.1 PDA Framework

The foundational PDA used in this paper is based on the PDA recognizer construction algorithm for well-matched VPGs from Jia et al. [10]. Every state in this PDA contains a set of non-terminal pairs $(L_1, L_2) \in V \times V$ where L_1 is the current context and L_2 is the next

non-terminal. The current context changes when a call symbol is consumed and is changed back to the old context when it consumes the corresponding return symbol. Therefore, only nesting rules can change the current context.

Example Grammar 2. *([10, 151:5])*

$$L_0 \Longrightarrow cL_1$$
$$L_1 \Longrightarrow \langle aL_2 b\rangle L_3$$

Example Grammar 2 displays the example grammar of Jia et al. [10, 151:5] to further illustrate the use case of non-terminal pairs. The first state should start in state $\{(L_0, L_0)\}$ as the current context and current rule are L_0. Then, if c is consumed, the first rule is applied and the PDA transitions to state $\{(L_0, L_1)\}$, as L_1 becomes the new next non-terminal. Next, upon encountering call symbol $\langle a$, the PDA transitions to state $\{(L_1, L_2)\}$ using the second rule, as there is a new context L_1 (due to the nesting) and L_2 becomes the new next non-terminal. Then, in context L_2, the PDA consumes input until return symbol $b\rangle$, where it goes to state $\{(L_0, L_3)\}$, returning to context L_0 and having next non-terminal L_3.

Also, note that states contain *sets* of non-terminal pairs, which is to handle ambiguity. Normally, nondeterministic pushdown automata are more expressive than deterministic ones, therefore, nondeterministic pushdown automata are also more expressive than a deterministic VPA [3]. However, the previously mentioned VPA handles non-determinism or ambiguity in a different manner; Every state contains a set of non-terminal pairs and if there is nondeterminism or ambiguity, then there will be only one transition with a destination state containing the set of all possible non-terminal pair destinations. Therefore, the resulting PDA will always be deterministic while still handling ambiguity and non-determinism.

While this framework handles ambiguous words, it does not support ambiguous nesting. This means that rules cannot exist if they contain a calling or returning symbol which is also a calling or returning symbol in another rule. This does not take away from the expressiveness of the context of the grammar, as different contexts inside or after a nesting can contain an *OR* on a deeper level in the grammar, to cover all the contexts. It only takes away the possibilities to use the same symbol in different nesting rules.

Example Grammar 3.

$$L \Longrightarrow aL_0 \mid aL_1$$

Example Grammar 3 displays how ambiguity is handled. The start state is $\{(L, L)\}$ and can continue to either (L, L_0) or (L, L_1) when consuming a. Therefore, the PDA will contain a transition from $\{(L, L)\}$, consuming a, to destination state $\{(L, L_0), (L, L_1)\}$, which, in this case, is the only transition.

3.2 Recognizing

The recognizer PDA is constructed by repeatedly considering all possible outward call-, return-, and internal transitions from the new states from the previous run, until the PDA converges when there are no more new states.

The stack in the PDA contains pairs $[S, \langle a]$, where S is a state and $\langle a \in \Sigma_c$. The derivative function δ is divided into δ_c for $c \in \Sigma_{int}$, $\delta_{\langle a}$ for $\langle a \in \Sigma_c$, and $\delta_{b\rangle}$ for $b\rangle \in \Sigma_r$. They take the current state and for $\delta_{b\rangle}$ also the top of the stack, and they return the set of derived states with a corresponding stack action.

The derivative functions are used to derive all possible next configurations in the PDA, given a current state, a stack, and the production rules. Therefore, they can be used in the construction of the recognizer PDA, which is done by an algorithm that repeatedly considers all newly possible states until there are no more new states, indicating a converged PDA. During every loop, all derivations of configurations for all new states are considered to create new states and transitions.

This was only a brief overview of the derivative functions and the PDA construction algorithm. Please read Jia et al. for further details [10, p. 151:5 - 151:8].

3.3 Parsing

Because every transition is formed by considering a certain rule, the recognizer PDA can be used to parse the input as well. In this case, every transition keeps a list of all the possible rule applications linked to the transitions. Then, after traversing the PDA, there is a list of sets of rules where every set of rules corresponds to one transition, where every rule is represented by a *Parse Tree Edge*.

Definition 3.2. A **Parse Tree Edge** (PTE) is a tuple representing the use of a rule. Every transition in the recognizer PDA contains a set of Parse Tree Edges, representing all possible rule application for that transition

(1) (L_0, c, L_1) where $c \in \Sigma_{int}$ for rule $L_0 \Longrightarrow cL_1$
(2) $(L_0, \langle a, L_1)$ where $\langle a \in \Sigma_c$ for rule $L_0 \Longrightarrow \langle aL_1 b\rangle L_2$
(3) $((L_0, L_1), b\rangle, L_2)$ where $b\rangle \in \Sigma_r$ for rule $L_0 \Longrightarrow \langle aL_1 b\rangle L_2$

As the PTE describes a rule but also a transition of non-terminals, the left-side of a PTE (L_0 for (1) and (2), and (L_0, L_1) for (3)) is considered the **Origin Non-Terminal** and the right-side PTE (L_1 for (1) and (2), and L_2 for (3)) is considered the **Destination Non-Terminal**

As seen in Definition 3.2, PTEs for internal- and call symbols contain the previous non-terminal, the consumed symbol, and the next non-terminal. However, PTEs for return symbols replace the previous non-terminal for a tuple (L_0, L_1) where L_0 represents the left-side non-terminal and L_1 represents the previous context, both corresponding to the rule $L_0 \Longrightarrow \langle aL_1 b\rangle L_2$. This is done to distinguish returns from internal- and call for pruning and extracting, which will be used in the following subsections on pruning and extracting.

Then, after traversing the PDA, the PDA returns the *Parse Forest*.

Definition 3.3. The **Parse Forest** is a list of sets of PTEs with length n for an input word of length n, where every index i in the list contains a set of PTEs representing all possible rules to consume the character at index i in the input word.

3.4 Pruning

Because the created Parse Forest after the recognizer PDA contains all possible rules for every symbol, there can be multiple PTEs in the list of sets that are not included in any valid parse. Therefore, the list requires pruning. Intuitively, a valid parse can be seen as a valid sequence of PTEs, denoted as a *valid trace*.

Definition 3.4. A **Trace** is a sequence of PTEs, which is a **Valid Trace** if every PTE properly transitions into the next PTE until the trace is finished and the symbols of all the PTEs in the trace form the input word.

A stack is required to check a trace, to determine if every closed scope was previously opened. Traces will be further discussed in the subsection on Extraction. This notion of a valid trace is used to prune the forest. Additionally, to check if PTE transitions are valid, a *nullability* function is needed.

Definition 3.5. A non-terminal L is said to be **Nullable** if there exists a rule $L \Longrightarrow \epsilon$. The nullability function θ returns whether a given non-terminal is nullable.

This function is used in two cases: (1) The destination non-terminal in the final PTE in a valid trace must nullable, since the trace must end there. (2) When a scope is closed, PTEs do not properly link (i.e., destination and following origin are not the same), as the trace continues in the old contex. However, the destination non-terminal of the last PTE before the return symbol must be nullable, as the trace must properly end before return to the old context.

Algorithm 1 Pruning Algorithm

```
1:  PF ← Parse Forest from the PDA
2:  S ← Stack of Non-Terminal Pairs
3:  PF[tail] remove all PTE where !θ(PTE)
4:  for i ← PF length to 1 do
5:      j ← i − 1
6:      for PTE_j in PF[j] do
7:          (_, c_j, L_j) ← PTE_j
8:          if ∄(L_i, _, _) ∈ PF[i], L_j = L_i and
              ∄((L_i0, L_i1), _, _) ∈ PF[i], θ(L_j) then
9:              remove PTE_j from PF[j]
10:         else if c_j ∈ Σ_c then
11:             (L_0, c_j, L_1) ← PTE_j
12:             if S_top = (L_0, L_1) then
13:                 pop S_top
14:             else
15:                 remove PTE_j from PF[j]
16:             end if
17:         else if c_j ∈ Σ_r then
18:             ((L_j, L_j2), c_j, _) ← PTE_j
19:             push (L_i, L_i2) to S
20:         end if
21:     end for
22: end for
```

Pruning is done by following traces in the parse forest backwards, where PTEs get removed if they are invalid. This is done backwards so that if a dead end is discovered, all PTEs in that invalid trace can be deleted from the leaf to the root. Because this process is backwards, stack actions are also reversed; A scope must before closed before it is opened, where closing pushes the stack, and closing pops the stack. Algorithm 1 displays the pseudocode of the pruning algorithm. PF is the parse forest, which is a list of sets of PTEs, created by the PDA traversal. Before discussing the algorithm, note that PTEs can be assigned multiple times, as they need to be casted to one of its types (see Definition 3.2).

First in the algorithm, in line 3, all PTEs in the tail of PF which have a non-nullable non-terminal destination, are removed, as mentioned before. Then, lines 4 and 5 represent going backwards in the parse forest, where i is the index of the previously pruned set, which goes backwards through the for-loop, and j is the index of the current set to be pruned, based on i.

The first condition in line 8 checks whether there exists a PTE in set i which has an origin non-terminal that equals the destination non-terminal of the currently checked PTE_j, because if it does, it indicates that PTE_j transitions validly to PTE_i for a call or internal symbol. The second condition in line 8 checks whether PTE_j's destination non-terminal is nullable and whether there exists a PTE in set i which continues a parent scope. If such an edge exists, it indicates that PTE_j transitions validly to PTE_i for a return symbol. Summing up line 8, if in neither of these cases such a PTE exists, then the edge is invalid and is removed in line 9.

Line 10 checks if the current PTE opens a scope, and line 11 checks if the stack corresponds to this call. If there is a call symbol which does not correspond to a previously closed scope, then the PTE is invalid and is removed in line 15. If it does correspond to the stack top, then the stack is popped in line 13. Line 17 checks if a scope is closed, and line 19 pushes the stack if so. Note that lines 10 to 19 are only relevant if PTE_j is valid, therefore they are included as *else if*'s.

Also, note that nowhere in the algorithm it is checked whether opening or closing symbols correspond to the nesting indicated by S_{top}, this is because ambiguity in nesting rules is not allowed (as mentioned in Subsection 3.1), therefore there can only exist one PTE at every index in the parse forest containing a call or return symbol. This is also the reason that the stack only contains pairs, instead of sets of pairs, as only one pair can be pushed and popped per every index in the parse forest.

3.5 Extraction

After the parse forest is pruned, only valid traces remain. The Extraction phase then extracts all traces from the pruned parse forest, which can be used to constructing the Parse Trees.

Algorithm 2 depicts a recursive algorithm to extract a valid trace, based on a given start PTE. PF is the pruned Parse Forest obtained from the pruning algorithm and S is the stack of non-terminal pairs to keep track of nesting rules. Then, in line 3, the recursive function *TRACE* starts, which returns a set of traces (which is a set of lists of PTEs). Parameter *PTE current* represents at which PTE the trace currently is and parameter i indicates the index of the next set of PTEs in PF.

Lines 4 to 6 check if the recursion is finished by checking if i has surpassed the last valid index of PF and, if so, returns a set with only the current PTE. Lines 7 to 11 check if the current PTE opens a scope, and pushes S if so. Then, Line 12 instantiates the result set as an empty set and line 13 loops over all PTE_i in $PF[i]$ to check to which next PTE PTE_i properly links to the current PTA, which is done in lines 14 to 33.

Algorithm 2 Tracing Algorithm

```
1:  PF ← Pruned Parse Forest
2:  S ← Stack of Non-Terminal Pairs
3:  function TRACE(PTE current, int i)
4:      if i = PF length then
5:          return {current}
6:      end if
7:      (_, c, L) ← current
8:      if c ∈ Σ_c then
9:          (L_0, _, L_1) ← current
10:         push (L_0, L_1) to S
11:     end if
12:     result ← {}
13:     for PTE_i in PF[i] do
14:         (_, c_i, _) ← PTE_i
15:         if c_i ∈ Σ_int or c_i ∈ Σ_c then
16:             (L_i, c_i, _) ← current
17:             if L = L_i then
18:                 for trace in TRACE(PTE_i, i + 1) do
19:                     trace ← {current} + trace
20:                     result.add(trace)
21:                 end for
22:             end if
23:         else if c_i ∈ Σ_r then
24:             ((L_i1, L_i2), c_i, _) ← PTE_i
25:             if θ(L) and S_top = (L_i1, L_i2) then
26:                 pop S
27:                 for trace in TRACE(PTE_i, i + 1) do
28:                     trace ← {current} + trace
29:                     result.add(trace)
30:                 end for
31:             end if
32:         end if
33:     end for
34:     return result
35: end function
```

Lines 14 to 22 check if the PTE_i has a call or internal symbol and casts the PTE_i properly in line 16 if it has. Line 17 checks if the current PTE is properly linked with the next PTE PTE_i. If it is, then line 18 calls *TRACE* again with the next PTE PTE_i and an incremented counter $i + 1$. The for-loop in lines 18 to 21 loops over all resulting traces (as *TRACE* returns all possible traces for the rest of the trace), appends the current PTE in front of them, and adds them to the result set. There can be multiple traces later on in the parse forest, as the ambiguity can exist later in the sequence.

Lines 23 to 32 have a similar function as lines 14 to 22, but now check if the PTE_i has a return symbol, where line 24 properly casts PTE_i if it does. Line 25 checks if the current PTE and PTE_i are properly linked, where the destination non-terminal of the current PTE must be nullable and the next PTE PTE_i must be the next expected return based on the Stack. If it is properly linked, the stack is popped and lines 27 to 30 use recursion to add the remainder of the traces in exactly the same manner as lines 18 to 21, as discussed previously.

Algorithm 3 Extraction Algorithm

```
1: AllTraces ← {}
2: for PTE_start in PF[0] do
3:     AllTraces.add(TRACE(PTE_start, 1))
4: end for
```

The tracing algorithm can extract specific traces based on where they start, but does not directly extract all traces for a pruned parse forest. Algorithm 3 does this by starting the trace for every PTE in the first set of PTEs in the pruned parse forest. Then all starting points are considered and all function calls of *TRACE* result in all possible traces.

In this case, extracted traces symbolize parse trees. While they are not directly parse trees, these traces contain all required information to construct parse trees. The tracing and extracting algorithms can easily be altered to support construction of parse trees during the extraction, based on the chosen implementation of the representing parse trees. The provided implementation [12] uses construction of abstract syntax trees (ASTs) and returns a set of such ASTs.

4 PENDING CALLS

For languages such as Python or HTML, nesting is often not properly closed. It is rare to see Python code where all indentations are properly closed by lines containing only tabs. Additionally, HTML code is very prone to forgetting closing tags. While unclosed tags are legalised in HTML5, it demonstrates the necessity of proper pending call handling for this languages of this class. Accepting pending calls can address common errors while also improving user convenience by omitting the requirement to properly close all scopes. This paper introduces an approach to manage pending calls based on the Colored Nested Words [1] by accepting pending calls if they are indirectly closed by a return symbol of a higher order.

Example Grammar 4.

$$S \Longrightarrow [A]E;$$
$$A \Longrightarrow \{B\}E;$$
$$B \Longrightarrow (E)E;$$
$$E \Longrightarrow \epsilon;$$

Example Grammar 4 displays a nested word grammar with nestings of different orders, where rules are in descending order of nesting order. The well-matched input $[\{()\}]$ is accepted by this grammar. Additionally, if pending calls are accepted, $[\{(]$, $[\{(\}]$ and $[\{()]$ are recognized as well, as a higher-order return symbols indirectly close lower-ordered scopes. This example grammar will be used throughout this section.

The following subsections will discuss details on an intuitive procedure to add transitions to the PDA corresponding to pending call acceptance. This procedure is split up into (1) Determining Colors and (2) Determining Colored Edges, which both materialize strictly after the PDA construction since the PDA is required to determine the colored edges.

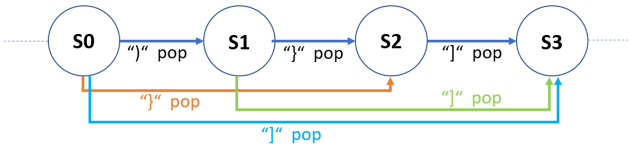

Fig. 1. Example Return Symbol Transition Sequence in a Simplified Automaton

4.1 Determining Colors

For every call and return pair, a color must be determined that corresponds to its place within the grammar-specified hierarchy. For example, in HTML $< ol >$ must be of a strictly higher order than $< li >$, as list items can only exist inside a list. To be more specific, a list item must be of a lower order when it exists on a greater depth in the grammar than the ordered list. Therefore, colors are represented by integers and every call and return pair is colored with the integer value of their corresponding depth in their grammar. Then, a call and return pair is said to be a higher order if it has a lower depth than the pair it is compared to. A recursive function with an incrementing depth counter can be used to map every call and return pair with their color/depth. In the case of Example Grammar 4, [...] has color 0, {...} has color 1, and (...) has color 2.

Definition 4.1. An opening and closing pair are labeled **color** i, if in the specified nested words grammar they are located at depth i.

4.2 Determining Colored Edges

Transitions in the PDA corresponding to these pending calls are called *colored edges*. They can be understood as shortcuts in the automata by bypassing (multiple) closing transitions. Therefore, colored edges can only exist if there are return symbol transition sequences with a minimum length of 3, where the colored edges are the shortcuts in between these transitions. Figure 1 depicts such a sequence in a simplified automaton corresponding to Example Grammar 4, where the regular (i.e. non-colored) edges are the dark blue transitions which display a sequence of the return symbol transitions for), }, and].

Definition 4.2. A **Return Symbol Transition Sequence** is a sequence of automata transitions $(O_0, c_0, D_0), (O_1, c_1, D_1)...(O_k, c_k, D_k)$ with origin state O_i, transition symbol c_i, and destination state D_i, where $k \geq 2$, $c_i \in \Sigma_r$ for $0 \leq i < k$, and $D_i = O_{i+1}$ for $0 \leq i < k-1$.

These Return Symbol Transition Sequences are traced in the PDA to determine the colored edges. Note that multiple colored edges have to be created, as some scopes can already be properly closed before a higher-order closing symbol closes lower-colored scopes. Alternatively, a higher-colored closing symbol can close the scope, but this does not have to be the highest-ordered closing symbol in the sequence. Figure 1 shows all colored edges for the Return Symbol Transition Sequence corresponding to Example Grammar 4, which shows that only the light-blue colored edge $(S0,], S3)$ does not suffice. This colored edge corresponds to the input [{(], but the inputs [{(}] and [{()] are accepting as well, which require the orange colored edge $(S0, }, S2)$ and the green colored edge $(S1,], S3)$, respectively. To conclude, for all states in the sequence, except the last two, a colored edge must be created towards all forward states, except for its direct successor (as this transition already exists), which consumes the return symbol corresponding to the last skipped transition.

Definition 4.3. The **Colored Edges** for a Return Symbol Transition Sequence $(O_0, c_0, D_0), (O_1, c_1, D_1)...(O_k, c_k, D_k)$ is the set $\{ (O_i, c_j, D_j) \mid 0 \leq i < k-1,\ i+1 < j < k \}$.

With the current definitions of colored edges, pruning and tracing would not recognize an input with pending calls as it fails to identify that lower-ordered scopes are indirectly closed. To achieve recognition by pruning and extracting, the colored edge contains a list of sets PTEs. While a normal transition contains a set of PTEs, representing all possible rules, a colored edge contains a list of sets of PTEs, representing all possible rules for every bypassed return symbol transition. This results in a parse forest just as if all scopes were properly closed. Additionally, the PTEs in the colored edges can be tagged so that they can be omitted in the resulting parse tree, as they were not in the original input.

5 PERFORMANCE

5.1 Theoretical Performance

The theoretical performance can be intuitively reasoned to be linear for unambiguous grammars and linear for every parse tree for ambiguous grammars. The PDA recognition must be linear for both cases, as the PDA traversal must entail exactly n transitions for a well-matched word with length n. All words with pending calls also have a well-matched input alternative where all calls are matched, therefore, for all pending call words their well-matched alternative is considered where there are n transitions. Thus, the production of the arse forest by the PDA is done in linear time for both unambiguous and ambiguous grammars.

The created parse forest is a list of sets of PTEs with length n for an input word with length n. For unambiguous grammars, every set only contains one PTE, as there is only one possible rule application for every character in the input word. For an ambiguous parse forest, m denotes the number of elements in its largest set. Then, the worst-case complexity of checking the final set is $O(m)$, and the worst-case complexity of comparing two sets is $O(m^2)$. This results in a worst-case complexity of $O((n-1)m^2 + m)$ for pruning, which approximates as $O(nm^2)$ for large n. Therefore, pruning for ambiguous grammars is linear to the input and quadratic to the number of parse trees. Unambiguous grammars and inputs remain linear, as $m = 1$ results in $O(n)$.

Note that m does not equal the number of possible parse trees. This is because there can exist multiple valid traces through a single PTE, therefore the number of possible trees depends on how many valid combinations of PTEs in every index of the parse forest there are. Then, k is the number of possible parse trees and the worst-case complexity in the extraction is $O(nk)$. This is because there are k valid traces with length n all needing to be extracted. Because $k = 1$ for unambiguous grammars and inputs, the complexity is, again, linear $O(n)$. Thus, in the extraction, the worst-case complexity is

$O(nm)$ for ambiguous grammars and remains $O(n)$ for unambiguous grammars.

In the worst-case scenario, all the sets in a forest for an ambiguous grammar and input have m items and there are $m^n = k$ parse trees. Because of this, extracting has worse complexity than pruning, as $m^2 < m^n$ for a large n results in $O(nm^2) < O(nk)$. Therefore, ambiguous grammars are bottle-necked by the worst-case complexity $O(nk)$ of extracting, which is linear for every parse tree. To conclude, unambiguous grammars and input remain linear at all times, while ambiguous grammars and input have linear complexity for every parse tree.

5.2 Practical Performance

This subsection illustrates the practical performance of the parsing process for the implemented parser generator based on the framework described in this paper [12] and compares it to the previously stated theoretical performance. This subsection first discusses performance for unambiguous grammars, where linearity is expected, followed by performance for ambiguous grammars, where linearity for every tree is expected.

5.2.1 Unambiguous Grammars. The test cases for unambiguous grammars are the following:

Long Input A simple grammar with a long input string. This is the null case other test cases can get compared to.

Deep Grammar [15] A grammar that chains multiple rules to each other, where the corresponding input must follow this long sequence of rules to terminate at the end. For this grammar, length/depth corresponds to the number of sequencing rules to reach termination.

Nested Grammar [15] A grammar that chains multiple nesting rules, where the corresponding input must first open many scopes and then close all of them before terminating. For this grammar, depth corresponds to the level of the deepest nesting.

Combined Grammar A randomly generated grammar which can have a chained rule or a nesting rule at any point in the grammar, where there is a 50% chance for a regular chaining rule, 25% for a new nesting, and 25% for closing the current scope and continuing the parent scope. For this grammar, length/depth corresponds to all the rules required to take for termination.

This testing framework is partly inspired by the methodology of Timmerman [15, 17] for testing the performance of OWL [9], where these entries were cited accordingly.

Figure 2 portrays the average practical performance for the previously defined test cases. Every test case is repeated 50 times, except Combined Grammar, which is repeated 10 times due to its long processing time. Every grammar case is tested up to a length/depth of 5000. However, the tests can be extended to test even longer/deeper cases, but this would require a very long processing time because numerous parsers have to be generated. As can be seen in Figure 2, the performances of all test cases stick relatively close together. At no point do any of the test cases exceed 100ms processing time. Because processing time remains significantly low even for larger

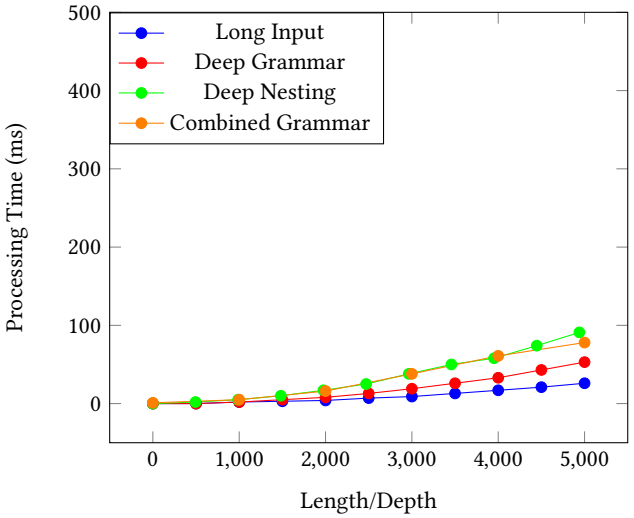

Fig. 2. Performance of Long Input and Long Grammar

lengths/depths for all cases, the performance or parsing unambiguous grammars can be considered linear, which is in line with the theoretical performance $O(n)$.

5.2.2 Ambiguous Grammars. The test cases for ambiguous grammars are the following:

Deep Ambiguity A grammar where every input symbol has two possible rule applications. Therefore, for an input with length n, the number of possible derivations becomes 2^n. This ambiguity is *deep* in the sense that is has few possible rule applications at every symbol but a long input string.

Broad Ambiguity A grammar where every input symbol has multiple possible rule applications. Therefore, for an input with length n and m possible rule applications at every step, the number of possible derivations becomes m^n. This ambiguity is *broad* in the sense that is multiple possible rule applications at every step but the input string is limited.

Figure 3 depicts the average performance for both of the defined testing criteria. Both ambiguity tests are repeated 50 times. The y-axis is defined in log-base 10, whereas the x-axis is in log-base 2. Because both deep- and broad ambiguity have exponentially more derivations for larger string input, the testing systems runs out of memory quickly. Additionally, the number of derivations gets exponentially larger before hitting this out-of-memory point. Due to this combination, there are limited testing points for both testing criteria.

For broad ambiguity $m = 10$, therefore, the number of derivations becomes 10^n, while the amount of derivations for deep ambiguity is 2^n. Consequently, they are compared based on the number of possible derivations, not on the length of the input string. Additionally, note that the lines start at 2^9 and 2^{10}, because the processing time for cases with less possible derivations is 0 ms.

As can be seen in Figure 3, the processing time grows linear to the number of possible derivations. Therefore, the practical performance

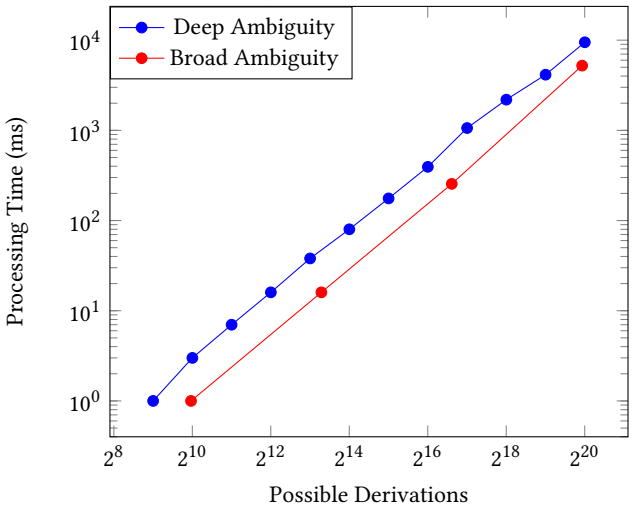

Fig. 3. Performance of Long- and Broad Ambiguity

is in line with the theoretical performance, which is linear time for every parse tree $O(nk)$.

6 CONCLUSION

This paper presented a parser generator framework for well-matched visibly pushdown grammars, supported in both theory and practice. This section concludes the paper by reflecting on the research questions established in the introduction.

RQ1: *How can a derivative-based parser generator extract all possible derivations from well-matched VPGs?* The derivate-based PDA construction algorithm [10] provides ambiguity handling and always results in a parse forest containing all possible derivations. A pruning algorithm is presented that iterates backwards through the parse forest to remove invalid traces from leaf to root. Then, the extraction algorithm extracts every valid trace, where during or after the extraction these traces can be used to construct a data structure to represent the parsing.

RQ2: *How can error handling be designed to repair some common defects automatically?* An intuitive approach for handling pending calls based on Colored Nested Words [1] increases the flexibility of the framework, as it improves error handling and user convenience by omitting the requirement to explicitly close all scopes.

RQ3: *What is the best possible performance of the generated parser?* In both theory and practice, the whole parsing process has linear complexity for unambiguous grammars and linear complexity for every possible parse tree for ambiguous grammars.

The whole framework is tested as an implementation [12] to prove its efficacy and performance. To conclude, the derivative-based, colored-edged framework is both flexible and efficient, as it accepts ambiguity and properly handles pending calls while having competitive performance.

7 LIMITATIONS AND FUTURE WORK

While the presented framework accepts very ambiguous grammars and input, it fails to process them if there are too many possible derivations. This is because every possible derivation brings a unique abstract syntax tree, where for an enormous number of possible derivations, this results in a heavy memory footprint. This could be resolved by representing all extracted traces in a Shared Packed Parse Forest (SPPF), where all possible derivations are represented in one tree structure and where equal leaves and subtrees can be shared [16]. Because equal leaves and subtrees do not exist in multiple ASTs and there is only one data structure, the memory burden might be severely less. This requires implementing SPPFs in the framework, where the SPPF construction must take place during the extraction algorithm, as otherwise, the large number of extracted traces would already flood the memory. Additionally, the framework with SPPFs must be tested to conclude if it can handle grammars and inputs with more possible derivations than the current framework.

This frameworks handles ambiguity for the (nested) words, but does not accept ambiguity in any of the call- or return symbols. The framework can be extended to attempt to support this ambiguity, where equal symbols for different nesting rules are tagged for identification and the PDA construction creates additional non terminal pairs for ambiguous calls and returns. Pruning and extracting also have to be altered, where they must push sets instead of instances, as then there can be multiple possible pushes for one PTE. Additionally, it can be experimented whether ambiguity for both regular- and nesting rules can be handled, i.e. a symbol is element of both the internal symbol alphabet, as well as the call- or return symbol alphabet.

The presented framework currently requires the VPG to be presented in a specific GNF-like form. For the framework to be practically usable requires users to be able to represent input grammars in the form of a metalanguage (such as BNF, EBNF, ABNF, or WSN), which is automatically converted to the specific required form. Then, at the end of the parsing process, this must be considered again to return parse trees with respect only to their original rules.

In 2006, Kumar et al. determined the existence of unique minimal modular VPAs and provided an active learning algorithm for automata minimization [11]. P. Chervet and I. Walukiewicz further studied minimizations of VPAs in 2007 [5]. Their insights can be used to further minimize the generated parser by the presented framework.

The derivate-based framework is already mathematically proven by Jia et al [10], but the pruning and extracting algorithms and the pending calls approach could use mathematical proof. The performance of the presented framework seemed linear for every parse tree in both theory and practice, but mathematical proof can further decide if this is true for all cases.

REFERENCES

[1] Rajeev Alur and Dana Fisman. 2021. Colored nested words. *Formal Methods in System Design* 58 (2021), 347–374. Issue 3. https://doi.org/10.1007/s10703-021-00384-2

[2] Rajeev Alur and P Madhusudan. 2004. Visibly Pushdown Languages. *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing - STOC '04* (2004). https://doi.org/10.1145/1007352

[3] Rajeev Alur and P. Madhusudan. 2009. Adding nesting structure to words. *Journal of the ACM (JACM)* 56 (5 2009). Issue 3. https://doi.org/10.1145/1516512.1516518
[4] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *Journal of the ACM (JACM)* 11 (10 1964), 481–494. Issue 4. https://doi.org/10.1145/321239.321249
[5] Patrick Chervet and Igor Walukiewicz. 2007. Minimizing variants of visibly pushdown automata. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 4708 LNCS (2007), 135–146. https://doi.org/10.1007/978-3-540-74456-6_14/COVER
[6] Noam Chomsky. 1956. Three models for the description of language. *IRE Transactions on Information Theory* 2 (1956), 113–124. Issue 3. https://doi.org/10.1109/TIT.1956.1056813
[7] Volker Diekert and Klaus-Jörn Lange. 2009. *Variationen über Walther von Dyck und Dyck-Sprachen*. Vieweg+Teubner, Wiesbaden, 147–154. https://doi.org/10.1007/978-3-8348-9982-8_13
[8] Sheila A. Greibach. 1965. A New Normal-Form Theorem for Context-Free Phrase Structure Grammars. *J. ACM* 12 (1 1965), 42–52. Issue 1. https://doi.org/10.1145/321250.321254
[9] Ian Henderson. 2017. Owl. https://github.com/ianh/owl.
[10] Xiaodong Jia and Ashish Kumar. 2021. A Derivative-Based Parser Generator for Visibly Pushdown Grammars. *Proc. ACM Program. Lang* 5 (2021), 24. https://doi.org/10.1145/3485528
[11] Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. 2006. Minimization, learning, and conformance testing of boolean programs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 4137 LNCS (2006), 203–217. https://doi.org/10.1007/11817949_14/COVER
[12] Bas Marcelis. 2023. Derivative-based-Colored-edged-Parser-Generator-for-Nested-Words. https://github.com/basmarcelis/Derivative-based-Colored-edged-Parser-Generator-for-Nested-Words/tree/master.
[13] Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with derivatives: A functional pearl. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP* (2011), 189–195. https://doi.org/10.1145/2034773.2034801
[14] Sebastian Muskalla. 2017. Visibly pushdown automata. https://www.tcs.cs.tu-bs.de/documents/AutomataTheory_SS_2017/visiblypushdown.pdf
[15] Luc Timmerman. 2022. Performance Testing Owl, Parser Generator for Visibly Pushdown Grammars. http://purl.utwente.nl/essays/91958
[16] Vadim Zaytsev. 2016. Cotransforming Grammars with Shared Packed Parse Forests. *Electronic Communications of the European Association of Software Science and Technology (EC-EASST); Graph Computation Models — Selected Revised Papers* 73 (April 2016). https://doi.org/10.14279/tuj.eceasst.73.1032
[17] Vadim Zaytsev. 2019. Event-based parsing. *REBLS 2019 - Proceedings of the 6th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, co-located with SPLASH 2019* (10 2019), 31–40. https://doi.org/10.1145/3358503.3361275