# A Parser Generator for Visibly Pushdown Languages: Translating between VPLs

MICHAEL JANSSEN, University of Twente, The Netherlands

Parsers often use the language class of Context-Free Grammars. This class has limitations, as it is not guaranteed to parse in polynomial time. In contrast, regular grammars can be parsed in linear time but are so restricted that many practical applications can not be modelled as a regular grammar. A language class has been proposed which adds some features to regular grammars that allow it to be used in a wide array of languages, such as JSON and XML. There is no publicly available language translator for this class, and there is a flawed parser generator. This paper introduces and expands upon a new parser generator and translator for the language class of Visibly Pushdown Grammars, using existing algorithms and concepts of both automata construction and language translation.

Additional Key Words and Phrases: Visibly Pushdown Languages, Visibly Pushdown Automata, Parser Generation, Language Translation

## 1 INTRODUCTION

In Computer Science, formal grammars serve as a method for describing languages. They are widely utilized, from identifying patterns in a text through regular expressions to compiling programming language code using a context-free grammar. The Chomsky hierarchy, introduced by Chomsky [3], categorizes grammars into four types, ranging from Type-0 to Type-3. Type-3 is the most restrictive of the classes yet can be modelled through a Finite State Machine. Since Chomsky's initial introduction of these types, various classes of grammars have been proposed that fall between them. For example, Type-2 grammars, also known as Context-Free Grammars (CFGs), can be non-deterministic. There is a deterministic subset of Context-Free Grammars, referred to as Deterministic Context-Free Grammars (DCFGs). The advantage of utilizing DCFGs is that they can be modelled through a Deterministic Pushdown Automaton [4] and therefore have linear parsing time. In contrast, Context-Free Grammars (CFGs) can be modelled through a Non-Deterministic Pushdown Automaton and are not guaranteed to parse linearly.

The Visibly Pushdown Grammars(VPGs) class was first introduced by Alur and Madhusudan [2]. The language class is more restrictive than Type-2 grammars, even DFCGs, but less restrictive than Type-3 grammars, the regular languages. With the addition of nesting, it can parse languages that regular grammars can not. HTML, XML, JSON and other structured languages are examples of Visibly Pushdown Languages. Due to the more restrictive nature, it has more closure properties than CFGs; it is closed under Union, Intersection and Complement [2]. This class of grammars is unrestricted enough for parsing languages such as XML and HTML, as those are too complex for regular languages. Also, automata parsing VPLs, Visibly Pushdown Automata, have a linear time complexity compared to their input string.

Current solutions for Visibly Pushdown Grammars are still being explored. The only publicly available parser generator for Visibly Pushdown Grammars is Owl, which is expanded later. Next to this, no translator from one VPG to another was found. A translator can transform a string from one language to another. Such a translator can transform to and from semantically-similar languages, such as document languages like XML and JSON, in linear time.

Looking at the current gap in currently available parser generators and translators for the language class of Visibly Pushdown Gramamrs, this research introduces a solution which can do both. This paper shows how the solution is constructed and verifies the linear parsing time claim. First related work is introduced, then the requirements of the solution are discussed, after which existing solutions are discussed, continuing to the solution proposed, verifying the results and concluding with discussing the current limitations and potential future work.

## 2 RELATED WORK

In this section, work related to this research is discussed. Specifically, the language class of Visibly Pushdown Grammars is expanded upon, the language translation is explained, and a quick introduction to regular expressions is given.

### 2.1 Visibly Pushdown Grammars

The language class of Visibly Pushdown Grammars are an extension to Regular Grammars. Compared to Regular Grammars, it allows a different rule in a grammar $L_i \rightarrow < aL_j > b$, where $< a$ is a call symbol, and $b >$ is a return symbol. This rule allows for tracking if a call symbol is closed; otherwise, the grammar would not allow it. A call symbol can be the symbol $'('$ and the return symbol $')'$, allowing matching brackets in the language. The grammar specification in [2] was defined as follows:

A context-free grammar $G = (V, S, P)$ over $\sigma$ is a Visibly Pushdown Grammar with respect to the partitioning $\tilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_l)$ if the set $V$ is partitioned into two disjoint sets $V^0$ and $V^1$, such that all the productions in $P$ are of one the following forms:

- $X \rightarrow \epsilon$;
- $X \rightarrow aY$, such that if $X \in V^0$ then $a \in \Sigma_l$ and $Y \in V^0$;
- $X \rightarrow aYbZ$ such that $a \in \Sigma_c$ and $b \in \Sigma_r$ and $Y \in V^0$ and if $X \in V^0$ then $Z \in V^0$.

The variables in $V^0$ derive only well-matched words with a one-to-one correspondence between calls and returns. The variables in $V^1$ derive words that can contain unmatched calls and unmatched returns. The symbols of the language are split into three distinct sets: $(\Sigma_c, \Sigma_r, \Sigma_l)$. $\Sigma_c$ corresponds to the call symbols, $\Sigma_r$. A Visibly Pushdown Automaton is an automaton which can recognize a VPL. Like a Pushdown Automaton(PDA), it has a stack. However, unlike a

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

PDA, actions to the stack are limited to the call and return symbols. Only a call symbol may push to the stack, and only a return symbol may pop. When a call symbol is read, it will be pushed to the stack; when a return symbol is read, it will pop from the stack. In practice, the stack contains the current nesting depth of the word being parsed. Dyck languages[5] can be modelled as a VPG, shown by Alur and Madhusudan [2].

### 2.2 Translation

Syntax-directed translation(SDT) [1] is commonly used in compiler construction to convert source code to a target language. Irons [6] was one of the first to use the technique for a compiler for ALGOL60. It is done by defining for all rules of a grammar $G$ a permutation of the nonterminals on the left-hand side. When a text has been parsed to a tree, the tree is modified at each node by:

- deleting descendants with terminal labels,
- reordering the nonterminal descendants according to the fixed rule, and
- introducing descendants labelled by output symbols.

For example, take the following grammar, where the translation is defined after the $\Rightarrow$.

$$L \rightarrow AB \Rightarrow (BA)$$

$$A \rightarrow a \Rightarrow a$$

$$B \rightarrow b \Rightarrow b$$

The permutation of the parse tree is (2, 1), as the first nonterminal, $A$, is moved to index two of the parse tree, and the second nonterminal, $B$, is moved to index one. The symbols $'('$ and $')'$ are introduced on indices zero and three. The parse tree can now be traversed depth-first to get the translated string. In this example, the input "ab" would be translated to "(ba)".

### 2.3 Regular Expressions

Regular expressions, introduced by Kleene [8], express Regular Languages. In Computer Science, it was introduced to computers by Thompson [11] for the IBM7094. They are widely used in various applications, such as websites and embedded systems. However, the most widely-used syntax and libraries for Regular Expressions are not guaranteed to run in linear time due to the inclusion of features like backreferences and look-ahead. In contrast, some libraries, such as the 'regex' crate[1] in Rust, have a parsing complexity of $O(mn)$, where $m \sim regex$ and $n \sim search\_text$, and are therefore linear for the search text because they do not have the non-regular functions. These libraries use advanced and often hand-written optimization and minimization techniques to get the most performance out of the regular expression parser.

## 3 REQUIREMENTS

This section describes the requirements that are set for the solution. Generally, it is a parser generator that takes a language specified in a specific syntax that describes the input language's recognition and translation. It then creates a parser and translator for this language to the specified output language. More specifically, the following requirements are set:

### 3.1 Syntax

A requirement is that the syntax of the input language is easy to write. Specifically, the syntax can use regular expressions, which are well-known to programmers and easy to write. This allows the solution to use existing regular expression libraries for parts of the parsing, speeding up parsing times. It can use regular expressions and nested words as easily as possible. Because ease of writing is a subjective measure, this research shows several example grammars of nested word languages in the next section, which tries to convince the reader that the syntax is easy to write.

### 3.2 Language class

For the solution, this paper restricts the original definition by Alur and Madhusudan [2] to not allow for pending calls and returns, as this is often not desired by practical applications. XML, for example, is considered malformed if an opening tag is not closed [10]. To achieve this, the production rules for the grammar are more restricted than the definition in subsection 2.1, and all transitions are in $V^0$. Because $V^1$ is now empty, the rules can be updated to the following:

- $X \rightarrow \epsilon$;
- $X \rightarrow aY$, such that $a \in \Sigma_l$ and $Y \in V$;
- $X \rightarrow aYbZ$ such that $a \in \Sigma_c$ and $b \in \Sigma_r$ and $Y \in V$ and $Z \in V$.

Because all productions $P$ of this class are also applicable to VPGs, as all rules in the matching class correspond to the rules applying to $V^0$, they are a subset of VPGs.

### 3.3 Parsing Complexity

One of the key advantages of the class of Visibly Pushdown Grammars is that their input can be parsed with linear complexity: If the input text doubles, the parsing time also does. This is in contrast to Context-Free Grammars, which have no such guarantee. Therefore, the requirement is set that the generated parser parses input texts with linear complexity. The parsing speed is tested extensively with several language features, which are expanded upon in section 6. Note that this does not mean that the complexity of the parser generation is linear, as this is done beforehand and is only needed once per parser, and thus can be used for multiple input texts.

## 4 EXISTING SOLUTIONS

### 4.1 Owl

Owl[2] is a parser generator for VPLs, written in *C*. Important to note here is that it does not have language translation, only parsing. It is, to the extent of our knowledge, the only publicly available parser generator for VPLs. Research has been done into the speed of the automata construction of Owl [12]. This research also showed that

[1] https://docs.rs/regex/latest/regex

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

it had some quirks, such as unexpectedly crashing when grammar sizes were too big and problems with ambiguity.

### 4.2 Language-specific translators

Language transformation is an important topic in Computer Science, they are found everywhere[9]. Compilers fall under them, translating input code to a specific assembly language. There are also language-specific translators for VPLs, such as a JSON to XML transformer. These are often hand-written and are, therefore, more performant than general solutions. A significant advantage that a general solution has is that once designed, it can write a translator between any two VPLs that are semantically similar. Especially for more obscure languages, this can be useful, as it might take much work to come by translators for those languages specifically.

## 5 SOLUTION

This section discusses several parts of the solution. It starts with the syntax in which the grammar and translations are specified and how it is parsed. It continues with the elaboration phase, where the restrictions of the input language are introduced to ensure that it can be compiled into a VPA. Afterwards, the process of transforming the input language to a recognizer in a VPA is described. Finally, the translation is described. The solution is built using the Rust language[3]. The full source code can be found online[7]

### 5.1 Syntax & Parsing

The grammar is described using a set of nonterminals. Each nonterminal can have one or more rules. Each rule consists of a list of terminals and nonterminals. The terminals, call and return symbols can all be written using regular expressions. If a part of the regular expression needs to be used in the translation (for example, if the inside of a string value needs to be extracted), named capture groups can be used. A complete overview of the syntax that can be used in the regular expressions can be found in the documentation of the Rust regex crate[4]. The only additional restriction is that capture groups are not allowed to start with the characters "RESTRICTED_", which the solution uses internally to decide which rule of a nonterminal is applicable.

If a nonterminal needs to be used to make the grammar recursive, that is, if that nonterminal is the same or higher in order, then a nested call and return are required. This can be indicated as follows: [*Call nonterminal Return*], where *Call* and *Return* are regular expressions.

The translation is defined by adding an arrow (->) to each rule of each nonterminal and specifying how that rule is translated. This can be any combination of strings and identifiers if each identifier is mapped once to an identifier on the left-hand side. Identifiers include the nonterminals used in the rule and the capture groups used in the terminals of that rule.

The input file is converted to an Abstract Syntax Tree using a parser combinator library[5]. An example grammar is the following:

```
Rule:
    ["\(" Rule=R "\)"] -> "[" R "]"
    String=S -> S
String:
    "\"(?P<Value>.*)\"" -> Value
```

This grammar takes a string which consists of string values, extracts the string value from the quotes and changes the brackets to square brackets. Note that in the regular expressions, the brackets needed to be escaped because regular expressions require this. The string '(("abc"))' would, for example, be translated to '[[abc]]'.

### 5.2 Elaboration

For the input grammar to be able to be compiled into a VPA, several restrictions are made on the language. These restrictions make sure that the grammar is Visibly-Pushdown. Next, there are several checks to see if the grammar is well-formed.

*Well-formed.* This is a simple check that all identifiers used in a rule are defined in the grammar. Next, it checks if all identifiers used in the rule translation are defined in the rule. It also checks if there is a one-to-one mapping between the left and right sides.

*VPA-correctness.* The compiler requires that every nonterminal used in a rule is defined after the rule it is used in. This ensures that the grammar is not recursive and, therefore, not Visibly-Pushdown, where recursion is only allowed with call and return symbols. Next to this, currently, the implementation also only allows identifiers at the end of a rule.

*Capture Groups.* As described earlier, there is a simple check whether named capture groups are not starting with *RESTRICTED_*, as this is used internally and is therefore not allowed to be used.

### 5.3 Automata Construction

To construct the VPA, several steps need to be done. First, all rules of the grammar need to be normalized to the form described in subsection 3.2. The solution does this by making every step of a rule a state in the automaton. The specific steps can be seen in Algorithm 1. The *c* function takes a regular expression as an argument and returns a compiled automaton for the provided regular expression. If multiple are supplied, it creates a regular expression that combines all supplied arguments, with all arguments being separately caught in capture group *RESTRICTED_0 ... RESTRICTED_N*. For example, if the arguments "*abc*" and "*def*" are supplied, it combines them into *(?P<RESTRICTED_0>abc)|(?P<RESTRICTED_1>def)*. This is used in the starting state of a nonterminal, where all first expressions of each rule of that nonterminal are combined into one regular expression, and the capture group that is captured decides which rule is chosen.

The algorithm also considers a special case: If the starting expression is an identifier, which means it is the only nonterminal of that rule, the starting expressions of that nonterminal are all added as an option for this nonterminal. This is done recursively. Because a rule can only reference a nonterminal lower in the ranking than itself, this expansion eventually ends. The solution also labels this transition, as otherwise, the parse tree can not be constructed.

Formally, while parsing, the input text is transformed into an annotated version of itself, which indicate if it is an internal, call or

[3]https://www.rust-lang.org
[4]https://docs.rs/regex/1.7.1/regex/#syntax
[5]https://docs.rs/nom/latest/nom/

**Algorithm 1** Automata construction of input syntax

```
state ← NTs.length + 1
next ← ∅
epsilon ← ∅
function AddReg(reg)
    if Reg.type = Nested then
        next[state] ← (c(Nested.call), Nested.id.index))
        epsilon ← (c(Nested.return), next)
    else if Ref.type = Terminal then
        next[state] ← (c(Terminal), next_state)
    end if
end function
for NT ∈ NTs.reverse() do
    starting_regs ← ∅
    for Rule in NT do
        state_after ← 0
        if Rule[−1].type == Identifier then
            state_after ← Identifier.index
        end if
        for Reg in Rule.skip(1).reverse().skip(1) do
            AddReg(reg)
        end for
        if Rule[0].type = Identifier then
            starting_regs ← get_starting_regs(Identifier)
        else if Rule[0].type = Nested then
            starting_regs ← (c(Nested.call), next)
            epsilon ← (c(Nested.return), next)
        else
            starting_regs ← (c(Terminal), next)
        end if
        next_state ← state
        state ← state + 1
    end for
    next[NT.index] ← (c(starting_regs.reg), starting_regs.next)
end for
next[0] ← (c(epsilon.return), epsilon.next)
```

return symbol. When the parser parses a new text part, the automaton's state decides if it is annotated as an internal, call, or return transition. This allows the input grammar to have an intersection between the call, return and internal symbols because the symbols are not $"a" \in \Sigma_c$, $"a" \in \Sigma_i$, but $("a", c) \in \Sigma_c$, $("a", i) \in \Sigma_i$.

As an example, take the following grammar:

```
A:
    "b" B
    ["a" A "a"]
B:
    "a"
```

This grammar has both $"a" \in \Sigma_c$, $"a" \in \Sigma_i$. But during parsing, the VPA annotates the call symbol $"a"$ as $("a", c)$ and the internal symbol $"a"$ of nonterminal B to $("a", i)$. Thus, the resulting automaton is still Visibly Pushdown.

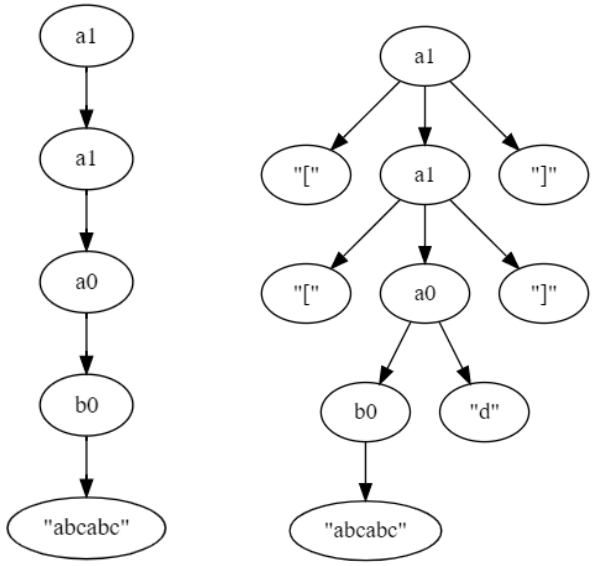
## 5.4 Translation

The solution uses the SDT approach [1] to transform the parse tree into the translation tree. As described in the syntax, the transduction is done by specifying a list of identifiers and constant strings on the right-hand side of a rule. The identifiers on the right-hand side and the left-hand side need to have a one-to-one mapping. During recognition, the VPA constructs a parse tree of the input text. Afterwards, the translator transforms the parse tree to the translation tree by changing the order of the elements of the parse tree corresponding to the mapping and adding the constant strings as leaves of the tree. Finally, the translation tree is traversed by depth-first search, and the output string is constructed by concatenating the encountered terminal strings together.

Take, for example, the following grammar:

```
A:
    "d" B              -> B "d"
    ["\(" A "\)"]        -> "[" A "]"
B:
    "(?P<value>[abc]*)" -> value
```

The input text $"((dabcabc))"$, which translates to $"[[abcabcd]]"$ would generate a parse tree as follows, where the nodes are identifiers and the index of the rule that is applied: A depth-first search of




(a) Parse tree of input string (b) Parse tree after transformation

Fig. 1. Example transformation of the parse tree for translation

this tree would result in the string $"[[abcabcd]]"$, which is indeed the expected translation.

## 5.5 Limitations & Ambiguity

Because of the choices made for the solution, there are several ways in which ambiguous grammars are parsed wrongly. Take this, for example:

```
A:  "a*" B
B:  "a"r
```

This is a problem because the solution compiles each regular expression separately and does not keep state, allowing backtracking (as this would make the parser non-linear). The first regular expression would consume all *a* characters in this case. When it arrives at nonterminal B, it does not have any *a* characters left to consume and consequently fails the parsing. This is currently a limitation of the solution and relies on the grammar writer not to allow for this.

Another way in which ambiguity would arise is in the following example:

```
A:
  "a" B
  "a" C
...
```

This is ambiguous because the automaton needs to know which nonterminal, B or C, is next. In these cases, the solution picks the first matching expression, the default behaviour of the regular expressions it uses internally. If B does not match the input, the parser fails, even if C does match. This requires attention from the writer.

## 6 VERIFICATION

### 6.1 Language class of the solution

Currently, the solution only covers part of the language class of Visibly Pushdown Languages. It is trivial to show that it does cover Regular Languages, as that only requires one rule in which the entire regular expression for that language is written down. However, the allowed rules for nested words are not expressive enough. Take, for example, a Visibly Pushdown Language, a simple bracketed language with multiple items. In a valid word in this language, items are enclosed by brackets and items inside those brackets can be separated by commas. $((a, b, c), (a, b))$ is a valid word in this language. The solution currently has no way of modelling this grammar. This is because of the following restrictions:

- Identifiers may only be used inside a nested word, or at the end of a rule
- After a call, only an identifier is allowed (e.g. *["call" A "return"]*)

If any of these restrictions were lifted, it would be possible to model the described grammar in the solution. This restriction is also why JSON and XML cannot be modelled currently.

### 6.2 Parsing speed

This section goes over the performance of the solution. It introduces several different grammars which can be easily increased in size. Each grammar is generated in increments of 100 rules, starting at 100 and ending at 10000. The methodology of each test is the same: Samples are created, and the recognizer and translator are tested with each set of rules on a valid input string. All benchmarks are run using the 'criterion' crate in Rust[6]. The method of generating grammars based on templates is based on the method used by Zaytsev [13]

[6]https://github.com/bheisler/criterion.rs

*6.2.1 Regular languages.* This test creates a language which creates *n* rules in the following way

```
regular(n): "z" regular(n-1)
...
regular1: "b" regular0
regular0: "a"
```

The results can be found in Figure 2.

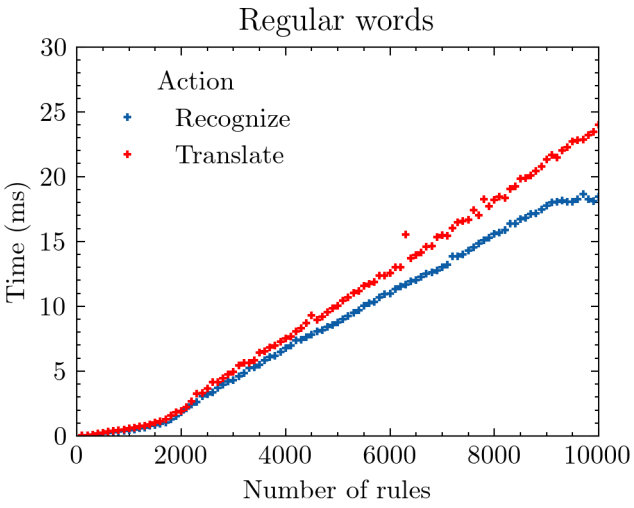



Fig. 2. Testing the regular language

Looking at this result, the solution seems to be nearing linear complexity. After 2000 rules, the parsing time increases linearly with the number of rules. It also shows that adding language translation adds extra cost to parsing time. There are also some outliers, such as near 6200 rules for the translator. This can probably be attributed to the fact that the tests were not run in a sanitized environment, but on a bare-metal machine.

*6.2.2 Deeply Nested languages.* This test creates a language which creates *n* rules in the following way

```
nested(n): ["(" nested(n-1) ")"]
...
nested1: ["(" nested0 ")"]
nested0: "a"
```

The results can be seen in Figure 3. Like the Regular words, the solution seems near linear complexity in parsing times. Next to this, several outliers can be spotted again. The extra time it takes to parse compared to the Regular words can be attributed to the fact that the input string is twice as long, indicating that whether a rule is internal or nested, this does not impact parsing speed much.

*6.2.3 Nested identifier languages.* This test creates a language which creates *n* rules in the following way

```
nested_i(n): nested_i(n-1)
...
nested_i1: nested_i0
nested_i0: "a"
```

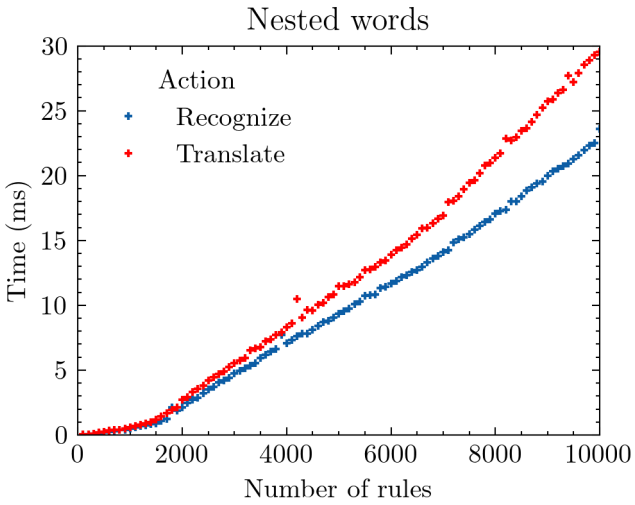



Fig. 3. Testing the nested language

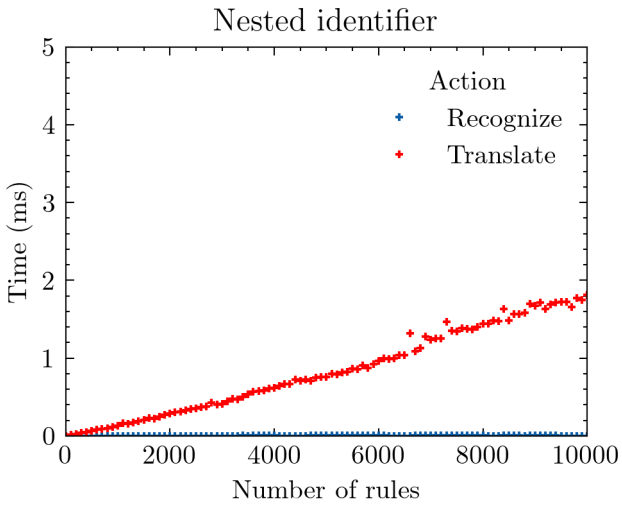



Fig. 4. Testing the nested identifier language

This makes it so that only the string "a" is valid here. The results can be seen in Figure 4. This shows that the recognition speed of the automaton is not affected by the depth of the rules. And when translation is done, the depth level adds some parsing time. This is expected, as for translation, a parse tree needs to be constructed, which gets bigger with the number of rules.

### 6.3 Generation speed

This benchmark tests the speed of the generation of the VPA and translator. This is expected to be linear with the number of rules. The test is executed on the same grammar as the 'nested' grammar in subsubsection 6.2.2. The results can be seen in Figure 5. These results show that the solution generates the automata and translator in linear complexity compared to the size of the input grammar.

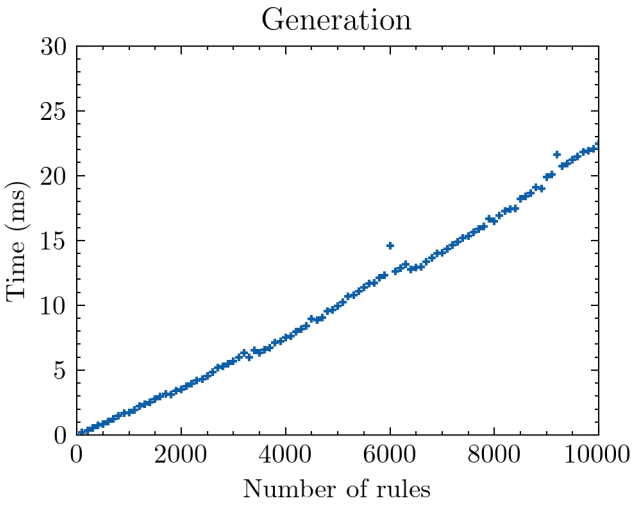



Fig. 5. Testing the generation speed

## 7 CONCLUSIONS

From the benchmarks, it can be concluded that the current implementation is a linear parser and translator for a subset of the language class of Visibly Pushdown Grammars. The translation does not have much overhead, and the generated parser can work fast. With this in mind, it is important to discuss the current limitations of the solution. The current implementation only partially covers Visibly Pushdown Languages. Specifically, it covers the entirety of Regular Languages, but some VPLs can not be modelled in the current solution, as shown in subsection 6.1. They include languages such as JSON and XML. This is a significant limitation of the solution. Future work could consider removing this limitation and allowing all VPLs to be modelled and used in this solution.

Next to this, the current implementation generates wrong parsers for some inputs, as discussed in subsection 5.5, which arise from the fact that the final automaton consists of several sub-automata, each containing a regular expression and a global call stack for the nested words. Future work could look into compiling this syntax into one combined automaton, which could increase speed as the entire automaton can be minimized and optimized and use a parsing technique that handles ambiguity.

## REFERENCES

[1] A. V. Aho and J. D. Ullman. 1971. Translations on a context free grammar. *Inform. And Control (Shenyang)* 19, 5 (Dec. 1971), 439–475. https://doi.org/10.1016/S0019-9958(71)90706-6

[2] Rajeev Alur and P. Madhusudan. 2004. Visibly pushdown languages. In *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. Association for Computing Machinery, New York, NY, USA, 202–211. https://doi.org/10.1145/1007352.1007390

[3] N. Chomsky. 1956. Three models for the description of language. *IRE Trans. Inf. Theory* 2, 3 (Sept. 1956), 113–124. https://doi.org/10.1109/TIT.1956.1056813

[4] Seymour Ginsburg and Sheila Greibach. 1965. Deterministic context free languages. In *6th Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1965)*. IEEE, 203–220. https://doi.org/10.1109/FOCS.1965.7

[5] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News* 32, 1 (2001), 60–65.

[6] Edgar T. Irons. 1961. A syntax directed compiler for ALGOL 60. *Commun. ACM* 4, 1 (Jan. 1961), 51–55. https://doi.org/10.1145/366062.366083
[7] Michael Janssen. 2023. vpl-parser-generator. https://github.com/Michael-Janssen-dev/vpl-parser-generator [Online; accessed 28. Jan. 2023].
[8] S. C. Kleene. 1956. Representation of Events in Nerve Nets and Finite Automata. In *Automata Studies*. Princeton University Press, Princeton, NJ, USA, 3–41. https://doi.org/10.1515/9781400882618-002
[9] Ralf Lämmel. 2004. Transformations everywhere. *Sci. Comput. Programming* 52, 1 (Aug. 2004), 1–8. https://doi.org/10.1016/j.scico.2004.03.001
[10] J. Rosenberg. 2007. Extensible Markup Language (XML) Formats for Representing Resource Lists. https://doi.org/10.17487/RFC4826 [Online; accessed 20. Jan. 2023].
[11] Ken Thompson. 1968. Programming Techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (June 1968), 419–422. https://doi.org/10.1145/363347.363387
[12] Luc Timmerman. 2022. *Performance Testing Owl, Parser Generator for Visibly Pushdown Grammars*. Ph. D. Dissertation. http://essay.utwente.nl/91958
[13] Vadim Zaytsev. 2019. Event-based parsing. In *REBLS 2019: Proceedings of the 6th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/3358503.3361275