
# "Learning Highly Recursive Input Grammars": A Replication Study

Sulav Malla

*A thesis presented in partial fulfilment of the requirements for the degree of*

Bachelor of Engineering Honours (Software)

**Supervisors:**

Rahul Gopinath, The University of Sydney

School of Electrical and Computer Engineering

The University of Sydney

November 04, 2025

# Abstract

# Contents

# Chapter 1

# INTRODUCTION

In the field of software testing, generating test inputs for a program (fuzzing) is a well-known and popular technique. To improve the effectiveness of fuzzers, recent research has focused on recovering input grammars from existing programs, as incorporating knowledge about the input language and grammar of the program under test drastically improves the effectiveness of fuzzers [1].

Learning Highly Recursive Input Grammars, authored by Lemieux C., Sen K., and Kulkarni N., and published in 2021 at UCB [2], presents an algorithm called ARVADA, developed for this purpose. ARVADA attempts to learn context-free grammars (CFGs) of a specified program given a set of valid inputs and a boolean-value black-box oracle $\mathcal{O}$. Along with presenting the algorithm, the paper evaluates it by comparing ARVADA to GLADE [3], a previously developed state-of-the-art algorithm for the same task in a similar setting. During the evaluation, it was observed that the F1 scores of GLADE were much lower than those reported in the official GLADE paper [3]. This led to a replication study of the original GLADE paper, conducted by Gopinath R., Bachir B., and Zeller A., and published as "Synthesizing Input Grammars: A Replication Study" [4] at CISPA, which investigated the accuracy of the results in the original paper. Similarly, as done by the researchers at CISPA, this thesis aims to replicate ARVADA to reproduce and investigate the results presented in the original paper [2].

This thesis is an attempt to reproduce ARVADA in a clean-room environment, meaning with no reference to or knowledge of the original implementation, but only the abstraction and explanations provided in the paper [2]. The implementation language of choice is C.

First, this paper will introduce general concepts of grammar and parsing, the importance of learning input grammars, a deeper explanation of ARVADA, and the motivation for conducting a replication study. This is followed by a brief explanation of ARVADA and how it works according to the original paper [2], then a more in-depth explanation of the algorithm and how it was reproduced in C with reference to the code. Afterward, an evaluation compares the results of the reproduced implementation with those from the original paper [2]. Finally, a discussion highlights and comments on the original algorithm, the reproduced results, and the overall process of conducting the replication study, followed by a brief conclusion.

• All code and implementation are open source and can be found here: `https://github.com/Stainima/ARVADA`

# Chapter 2

# GENERAL REVIEW OF LITERATURE

### 2.0.1 What are Grammars?

Grammars in computer science can be defined as a set of rules by which valid sentences in a language are constructed [5], serving as a blueprint for the language. Beginning with a start symbol, which is a single non-terminal, production rules are applied sequentially, adding symbols from the alphabet according to the grammar's production rules, to derive a string that is valid in the language [6].

A grammar can be represented by the tuple $< N, T, P, S >$, where:

- $N$ is a finite, non-empty set of non-terminal symbols/alphabet.
- $T$ is a finite set of terminal symbols/alphabet.
- $P$ is a finite set of production rules.
- $S$ is the start symbol (a non-terminal symbol/alphabet).

Here, the production rules $P$ define all symbol substitutions that can be performed recursively to generate different symbol sequences, known as *strings* [6]. These rules are written in the form $A \rightarrow w$, where $A \in N$ and $w \in (N \cup T)^*$.

### 2.0.2 What are Context-Free Grammars?

Depending on the set of production rules, each grammar can be classified according to the *Chomsky hierarchy* [7].

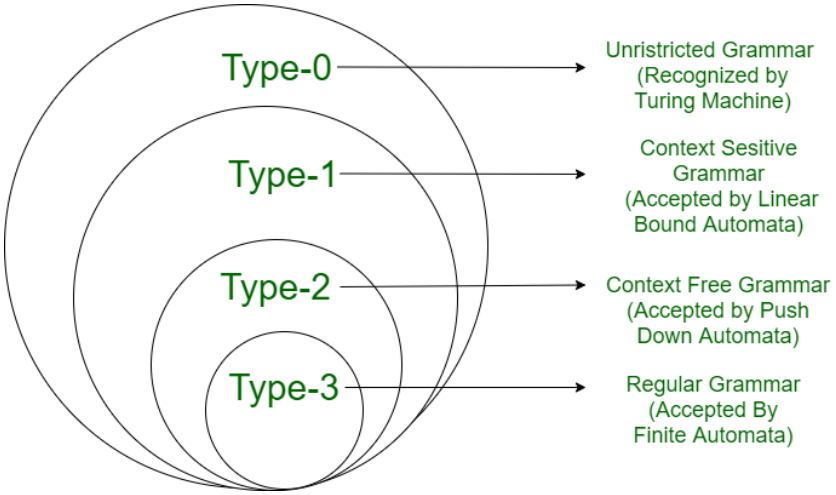


**Figure 2.1:** Chomsky hierarchy [8]

The hierarchy outlines four distinct types of grammars, ranging from **Type-3** (the most restrictive) to **Type-0** (the most general and unstructured). **Type-2** grammars in this hierarchy are known as *context-free grammars*.

### Type-3

Type-3 grammars, known as regular grammars are used to generate regular languages. A grammar is classified as a regular grammar if its production rules follow from $X \to a | X \to aY$. The left-hand side must consist of a single $T$ and the right hand side must consist of an single $T$ or single $T$ and single $N$.

Regular grammar can take 2 froms, right linear and left linear.

Right-linear take the following form, where the $N$ on the right hand side of the arrow is on the far right.

$$X \to a$$

$$X \to aB$$

$$X \to \varepsilon$$

Left-linear take the following form, where the $N$ on the right hand side of the arrow is on the far left.

$$X \to a$$

$$X \to Ba$$

$$X \to \varepsilon$$

Note that right-linear and left-linear forms cannot be mixed, as doing so may generate languages that are not regular. Additionally, production of $\varepsilon$ is allowed only if the corresponding nonterminal does not appear on the right-hand side of any production rule [9], [10]

**Type-2**

Type-2 grammars, commonly known as context-free grammars, have production rules of the form:

$$A \to \alpha$$

where $A \in N$ and $\alpha \in (T \cup N)^*$, meaning any string composed of terminal and nonterminal symbols [9], [10].

Due to the nature of the right-hand side, nonterminals are allowed to recursively expand, which can lead to repetition. For example:

$A \to sAb$

$A \to \varepsilon$

can produce derivations of the form

$$A \Rightarrow sAb \Rightarrow ssAbb \Rightarrow \cdots \Rightarrow s^n b^n,$$

until eventually $A \Rightarrow \varepsilon$.

This property is particularly useful because it allows the grammar to enforce well-formed parenthesis expressions and other recursive structures purely through production rules. As a result, context-free grammars are of great importance in the design and specification of programming languages[9], [10].

**Type-1**

Type-1 grammars, also known as context-sensitive grammars, have production rules of the form:

$$\alpha A \beta \to \alpha \gamma \beta$$

where $A \in N$ and $\alpha, \beta, \gamma \in (N \cup T)^*$. This means that $A$ can be expanded to $\gamma$ only in the specific context where it appears between $\alpha$ and $\beta$ in the given order [9], [10].

A key property of context-sensitive grammars is that production rules must be *non-contracting*: the length of the right-hand side must be greater than or equal to the length of the left-hand side. This implies that no nonterminal on the right-hand side can derive $\varepsilon$, meaning nullable productions are not allowed.

An example derivation using a context-sensitive production is:

$$A \Rightarrow aA\beta \Rightarrow aaA\beta\beta \Rightarrow \cdots$$

**Type-0**

Type-0 grammars, also known as unrestricted grammars, sit at the top of Chomsky's hierarchy. Their production have a complete lack of restrictions and take the general form:

$$\alpha \to \beta$$

where $\alpha, \beta \in (N \cup T)^+$, meaning they are strings consisting of terminal and nonterminal symbols. The only requirement is that $\alpha$ must contain at least one nonterminal symbol to allow for further derivations [9], [10].

Type-0 grammars are the most expressive class in the Chomsky hierarchy and can generate all recursively enumerable languages.

### 2.0.3 What is ARVADA?

#### Introduction

ARVADA is an algorithm published in "Learning Highly Recursive Input Grammars" [2] at the University of California, Berkeley in 2021. It is designed to learn context-free grammars from a set of positive examples and a Boolean-valued oracle $\mathcal{O}$. Starting from initially flat parse trees, ARVADA repeatedly applies two specialized operations, **bubbling** and **merging**, to incrementally add structure to these trees. From this structured representation, it extracts the smallest possible set of context-free grammar rules that accommodate all the given examples. The algorithm aims to generalize the language as much as possible without overgeneralizing beyond what is accepted by $\mathcal{O}$.

Like GLADE [3], ARVADA operates under the assumption of a black-box oracle $\mathcal{O}$. This means that ARVADA has no access to or knowledge of the internal workings of the oracle and can only observe the Boolean values returned by $\mathcal{O}$.

#### Explanation

ARVADA takes as input the oracle $\mathcal{O}$ and a set of positive, valid oracle inputs $S$. For each string $s \in S$, querying $\mathcal{O}(s)$ returns `True`. The algorithm begins by constructing a flat parse tree for each string in $S$. Each tree has a single root node $t_0$ whose children correspond to the individual characters of the input string $s$.

Next, ARVADA performs the **bubbling** operation. In this step, a sequence of sibling nodes in the tree is selected and replaced with a new non-terminal node. This new node takes the selected sibling nodes as its children, thereby introducing an additional level of structure. Essentially, ARVADA transforms sequences of terminal nodes in the flat parse tree into subtrees by introducing new non-terminal nodes and progressively adding structure to the tree.

ARVADA then decides whether to accept or reject each bubble by checking whether the newly bubbled structure enables a sound generalization of the learned grammar. Each non-leaf node in the tree can be viewed as a non-terminal in the emerging grammar. To determine whether a bubble should be accepted, ARVADA checks whether replacing any node in the tree with the new bubbled subtree results in the generation of valid input strings according to $\mathcal{O}$. If the replacement produces valid strings, the bubble is accepted, and the tree is restructured so that both the bubbled subtree and the replaced node share the same non-terminal label.

The addition of new non-terminal nodes expands the language defined by the learned grammar, since any string derivable from the same label can now be substituted interchangeably. This relabeling of the bubbled subtree and the replaced node is called a **merge**, as it merges the labels of two previously distinct nodes in the tree. If a bubble is not accepted, it is discarded, and none of the trees are affected or structurally modified.

### Walkthrough

This walkthorugh will follow very closely to the examples provided in the original paper [2], and use a concrete example to provide an indepth explaination of ARVADA.

$G_w$

$$
\begin{array}{rcl}
\textit{start} & \rightarrow & \textit{stmt} \\
\textit{stmt} & \rightarrow & \text{while}\textvisiblespace\ \textit{boolexpr}\textvisiblespace\ \text{do}\textvisiblespace\ \textit{stmt} \\
 & & \mid \text{if}\textvisiblespace\ \textit{boolexpr}\textvisiblespace\ \text{then}\textvisiblespace\ \textit{stmt}\textvisiblespace\ \text{else}\textvisiblespace\ \textit{stmt} \\
 & & \mid \text{L}\textvisiblespace = \textvisiblespace\ \textit{numexpr} \\
 & & \mid \textit{stmt}\textvisiblespace\ ;\textvisiblespace\ \textit{stmt} \\
\textit{boolexpr} & \rightarrow & \sim\textit{boolexpr} \\
 & & \mid \textit{boolexpr}\textvisiblespace\ \&\textvisiblespace\ \textit{boolexpr} \\
 & & \mid \textit{numexpr}\textvisiblespace == \textvisiblespace\ \textit{numexpr} \\
 & & \mid \text{false} \\
 & & \mid \text{true} \\
\textit{numexpr} & \rightarrow & (\textvisiblespace\ \textit{numexpr}\textvisiblespace + \textvisiblespace\ \textit{numexpr}\textvisiblespace\ ) \\
 & & \mid \text{L} \\
 & & \mid \text{n}
\end{array}
$$

$$S = \{\texttt{while true \& false do L = n},\quad \texttt{L = n ; L = (n+n)}\}$$

$$O(i) = \begin{cases} \text{True} & \text{if } i \in \mathcal{L}(G_w) \\ \text{False} & \text{otherwise} \end{cases}$$

**Figure 2.2:** Definition a simple while grammar $G_w$, sample input strings $S$, and oracle $\mathcal{O}$

**Algorithm 1** An algorithm with caption

**Require:** a set of examples $S$, a language oracle $\mathcal{O}$.
1: $bestTrees \leftarrow$ NaiveParseTrees($S$)
2: $bestTrees \leftarrow$ MergeAllValid($bestTrees, \mathcal{O}$)
3: $updated \leftarrow$ True
4: **while** $updated$ **do**
5: $updated \leftarrow$ False
6: $allBubbles \leftarrow$ GetBubbles($bestTrees$)
7: **for** $bubble$ **in** $allBubbles$ **do**
8: $bbldTrees \leftarrow$ Apply($bestTrees, bubble$)
9: $accepted, mergedTs \leftarrow$ CheckBubble($bbldTrees, \mathcal{O}$)
10: **if** $accepted$ **then**
11: $bestTrees \leftarrow mergedTs$
12: $updated \leftarrow$ True
13: **break**
14: **end if**
15: **end for**
16: **end while**
17: $G \leftarrow$ InducedGrammar($bestTrees$)
18: **return** $G$

### 2.0.4 What is GLADE?

GLADE is an algorithm proposed by Bastani et al., published in Synthesizing Input Grammars at PLDI 2017 [3]. Like ARVADA, GLADE uses a set of valid inputs and black-box access to the program, with the aim of automatically approximating the context-free input grammar of the given program.

Because GLADE and ARVADA share a similar experimental setting, GLADE was used as a benchmark for ARVADA during its evaluation. It was shown that GLADE, on average, had a faster runtime compared to ARVADA. However in terms of generalization, following the original grammar of the oracle more closely , ARVADA out-performed GLADE across the 11 benchmarks, achieving a higher F1 score on 9 of the 11 benchmarks.

### 2.0.5 Why C?

In the original study [2], the ARVADA algorithm was implemented in Python. When compared to GLADE [3], which was implemented in Java, ARVADA exhibited a slower average runtime across all benchmarks. In the study, this was attributed to the natural runtime disadvantage of Python compared to Java, which is valid, however the possibility that ARVADA itself might be inherently slow was not acknowledged.

In a comparative study, A Pragmatic Comparison of Four Different Programming Languages [11], it was found that if speed and efficiency are important, C is a better option than Python. C, being a mid-level, statically typed, structured language that runs under a compiler, is consistently faster than dynamic languages that run under an interpreter, such as Python [12]. In addition to being a structured language, C also provides only essential features. These limited

features contribute to its efficiency but also introduce a higher level of complexity compared to Python [11], [12].

Therefore, with the aim of investigating and potentially improving the runtime bottleneck, C was chosen as the language of implementation in this replication study. The increase in implementation complexity due to the nature of programming in C compared to Python was also acknowledged.

### 2.0.6 Further Quesitons

What are parsers?

Why do a replication study?

Why ARVADA / Problem Statement?

what are the drawbacks of ARVADA?

Why is learning highly input grammar important?

What are other Similar works done?

What is the work done in this field after ARVADA?

# Chapter 3

# METHODOLGIES & IMPLEMENTATION

Methodolgies: ARVADA walkthrough

How you did it, and point out any differences?

**Listing 3.1:** Struct used in code to store all *trees*

```
1 typedef struct nodes{
2     int capacity;
3     int count;
4     struct node **nodes;
5 } Nodes;
```

**Listing 3.2:** Struct used in *trees*

```
1 typedef struct node{
2     int capacity;
3     char character;
4     struct node *parent;
5     int t;
6     int num_child;
7     int pos;
8     struct node **children;
9 } Node;
```

# Chapter 4

# EVALUATION

# Chapter 5

# DISCUSSION

# Chapter 6

# CONCLUSION

# References


[1] R. Gopinath and A. Zeller. "Building Fast Fuzzers," pre-published.

[2] N. Kulkarni, C. Lemieux, and K. Sen, "Learning Highly Recursive Input Grammars," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Melbourne, Australia: IEEE, Nov. 2021, pp. 456–467, ISBN: 978-1-6654-0337-5. DOI: 10.1109/ASE51524.2021.9678879.

[3] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing Program Input Grammars,"

[4] B. Bendrissou, R. Gopinath, and A. Zeller, ""Synthesizing input grammars": A replication study," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, San Diego CA USA: ACM, Jun. 9, 2022, pp. 260–268, ISBN: 978-1-4503-9265-5. DOI: 10.1145/3519939.3523716.

[5] T. Jiang, M. Li, B. Ravikumar, and K. W. Regan, "Formal Grammars and Languages,"

[6] "Grammar in Theory of Computation," Grammar in Theory of Computation.

[7] N. Chomsky, "THREE MODELS FOR TIE DESCRIPTION OF LANGUAGE," Papaer, MIT, Massachusetta, 1956.

[8] "Chomsky Hierarchy in Theory of Computation."

[9] M. Hendriks and V. Zaytsev, "Consider it Parsed!" Thesis, University of Twente, Enschede, Netherlands.

[10] Z. Shi, *Intelligence Science*. 2021, 215-266.

[11] S. Ali and S. Qayyum. "A Pragmatic Comparison of Four Different Programming Languages," pre-published.

[12] R. Kumar, S. Chander, and M. Chahal, "Python versus C Language: A Comparison," vol. 9, 2022.

[13] M. Schröder, J. Cito, and T. U. Wien, "Static Inference of Regular Grammars for Ad Hoc Parsers,"