"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]. Additionally, also attempting to make improvements in the paper, algorithm, and its evaluation.

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 $w \in (N \cup T)^*$.

These sets of rules are used by parsers to parse a string of tokens and analyse its syntax against this set of rules(Grammars)[7], [8].

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* [9].

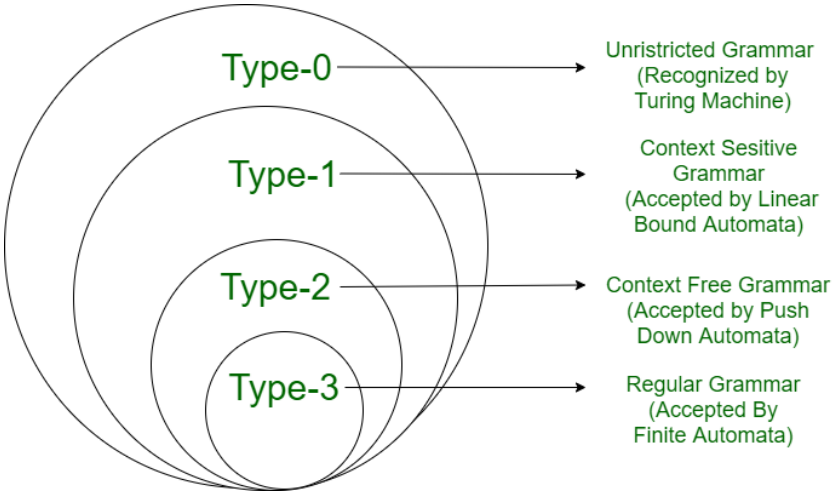


Figure 2.1: Chomsky hierarchy [10]

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 a single T or a single T and a single N.

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

Right-linear takes 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, Doing so may generate languages that are not regular. Additionally, production of ε is allowed only if the corresponding nonterminal does not appear on the right-hand side of any production rule [8], [11]

Type-2

Type-2 grammars, commonly known as context-free grammars (CFGs), 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 [8], [11].

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 parentheses 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[8], [11].

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 γ only in the specific context where it appears between α and β in the given order [8], [11].

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 ε, 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 has a complete lack of restrictions and takes 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 α must contain at least one nonterminal symbol to allow for further derivations [8], [11].

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 specialised 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 generalise the language as much as possible without overgeneralising 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 $\mathcal{O}$.

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 shares a similar experimental setting as ARVADA and is a predecessor, 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 generalisation, following the original grammar of the oracle more closely, ARVADA outperformed GLADE across the 11 benchmarks, achieving a higher F1 score on 9 of the 11 benchmarks.

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 generalisation 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 walkthrough will follow very closely to the examples provided in the original paper [2], and use a concrete example to provide an in-depth understanding of ARVADA.

G_w

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

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

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

Figure 2.2: Definition a simple while grammar G_w, sample input strings S, and oracle $\mathcal{O}$ [2]

For this walkthrough, the simple while grammar G_w and input string S provided in figure **??** will be used. Noting again that ARVADA treats $\mathcal{O}$ as a black box, meaning it has no structural knowledge of G_w, and G_w is only shown to clarify the behaviour of $\mathcal{O}$.

Algorithm 1 High-level overview of ARVADA [2]

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

ARVADA follows the high-level overview provided in algorithm 1, and begins by taking all the input strings in S and building a naive flat parse tree for each input string.

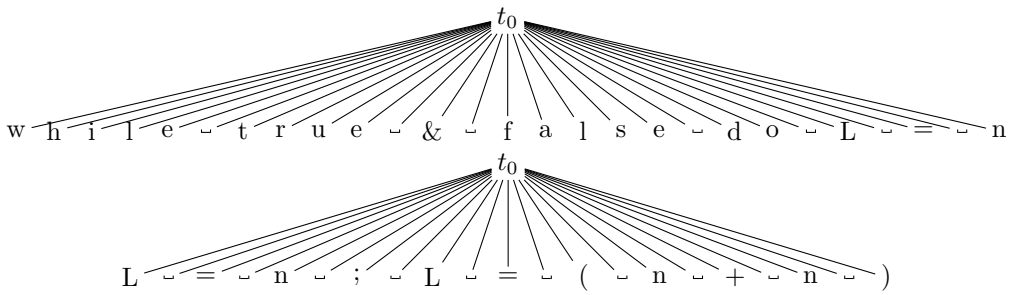


Figure 2.3: Initial set of flat naive parse trees ARVADA builds given inputs S, where each t_i is a non-terminal

In this initially naive parse tree, for each string input. There is a non-terminal t_0, and all characters in the string, including the spaces, are children nodes. From this alone the following grammar can be induced.

$$t_0 \rightarrow \text{w h i l e ␣ t r u e ␣ \& ␣ f a l s e ␣ d o ␣ L ␣ = ␣ n}$$

$$t_0 \rightarrow \text{L ␣ = ␣ n ␣ ; ␣ L ␣ = ␣ (n + n)}$$

Note that the string derivable from a node N, in this case t_0, is the concatination of all its leaf nodes.

Next, although not present in pesudocode 1, ARVADA has a functionality called pre-tokenisation which can be toggled on or off. Pre-tokenisation is a step after all the naive trees are built, where the algorithm group together sequence of contiguous characters of the same class (lowercase, uppercase, whitespaces, digits) into leaf tokens and all punctuations are kept separate.

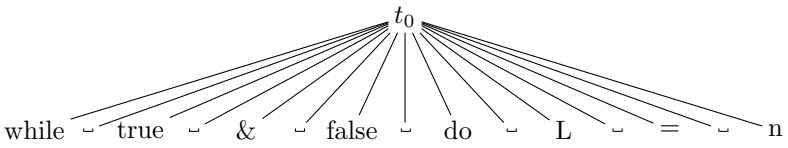


Figure 2.4: ARVADA pre-tokenisations example

Next, ARVADA attempts its first optimization performing MERGEALLVALID, which attempts to gerenalise and add stucture to the tree by seeing if any of the leaf nodes/tokens can already be merged and put under new non-terminal labels. MERGEALLVALID looks at 2 leaf nodes if it is non-tokenised or 2 non-terminals if tokenised at a time, let these be t_a and t_b. Then it goes through all the parse trees creating a replica with all instances of t_b replaced with t_a. Let the original parse trees be T and the replica with replacements be T'. From replica new candidate strings can be derived, these candidate strings are fed into $\mathcal{O}$. The same process is is done, where all instances of t_a is replaced with t_b. Now having 2 sets of candidate strings, where in 1 set t_b replaces t_a and other set has vise-versa. If all the strings in both sets are accepted

by $\mathcal{O}$, then t_a and t_b are put under a new non-terminal labelled t_c. The roots t_0 can also be t_a or t_b.

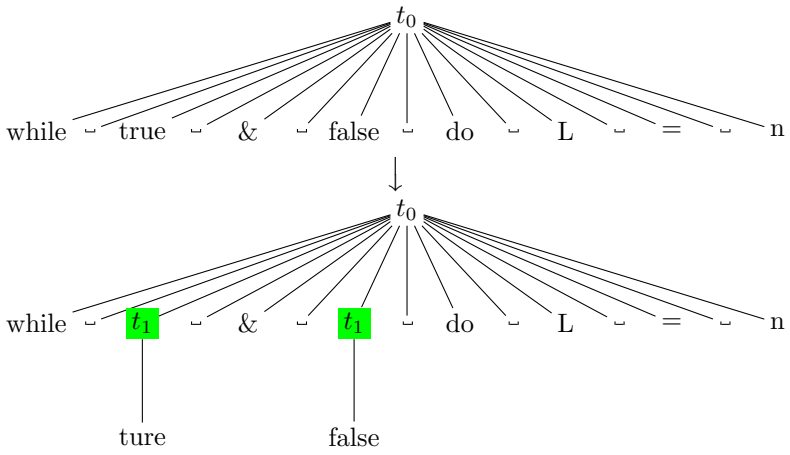


Figure 2.5: Example of a MERGEALLVALID run

As MERGEALLVALID only looked at 1 leaf-node or non-terminal on it own (no sequence of leaf-node or non-terminals), ARVADA now moves onto bubbling. Where bubbiling now looks at a sequence of leaf-nodes and/or non-terminal.

2.0.4 Why replicate ARVADA?

The replication study of GLADE [3] conducted by researchers at CISPA [4] in 2022 reported results that were inconsistent with those presented in the original paper [3]. Since ARVADA is closely related to GLADE, sharing a similar purpose and experimental setting, there is reason to suspect that ARVADA may also yield inconsistent results upon reimplementation.

Another motivation for replication arises from the incomplete nature of the ARVADA study and its evaluation. Although the paper presents a reasonable amount of testing and statistical data, compares the algorithm with GLADE, and provides a detailed explanation of ARVADA, it does not offer formal guarantees. It remains unclear whether ARVADA is applicable in all scenarios or what subset of scenarios it can reliably handle. Furthermore, when comparing ARVADA with GLADE, there is a possibility that the grammars used for testing were selected based on performance considerations. In later research, it has been stated that ARVADA only has high F1 scores when input strings are relatively small, and can vary widely on each run [12], raising suspicions that the study may have been selective.

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 [13], it was found that when speed and efficiency are prioritized, C is a better choice than Python. As a mid-level, statically typed, and structured language that runs under a compiler, C consistently outperforms dynamically typed, interpreted languages such as Python [14]. Moreover, C's provision of only essential features contributes to its efficiency but also increases programming complexity compared to Python [13], [14].

Therefore, to investigate and potentially address runtime bottlenecks, C was selected as the implementation language for this replication study. The added complexity inherent to programming in C, relative to Python, was also acknowledged.

2.0.6 Why is learning input grammar important?

Grammar inference is important for many software engineering tasks, as knowledge about a program grammar helps with code comprehension, reverse engineering, detecting and refactoring code smells, transforming source code for optimization or bug fixing, and generating test inputs[12]. However, due to the increasing restrictions in many software systems—caused by global privacy and security regulations—we often do not have access to the source code needed to learn their grammar. Even upon open-source programs and code, many only have closed source prasers, making white-box or grey-box instrumention difficult [12], [15].

This limitation highlights the importance of learning input grammars, which are formal grammars defining valid program inputs, and approaches that treat a program as a black box and recursively apply input grammars to infer or reverse-engineer the underlying grammar of the program.

2.0.7 Related Work

2.0.8 TREEVADA

Published in "Fast Deterministic Black-box Context-free Grammar inference" 2023, at the University of Texas[12], TREEAVADA is an algorithm that is based on ARVADA[2], aiming to solve the non-deterministic and speed-related limitations of ARVADA. The paper claims, TREEVADA yields better quality grammar in a single run, with faster run time.

To achieve such results, TREEVADA uses a few different techniques. First, during pretokenisation, TREEVADA pre-structures its parse according to nesting rules induced by balanced brackets common in many grammars, uses ' " quotes grouping and identifying string literals, and the same heuristics used in ARVADA, such as lowercase and numbers. Essentially, adding more structure to the initial parse trees compared to ARVADA, which does not consider brackets. Noting that TREEVADA assumes that the program uses ' " quotes to wrap a string and brackets for nesting only.

Secondly, to address the non-deterministic aspect of ARVADA, TREEAVADA switches to a deterministic data structure for specific operations and discard all bubbles with unmatched

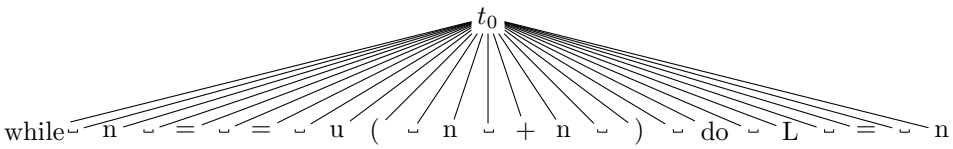


Figure 2.6: Naive Parse trees after pre-tokenisation in ARVADA

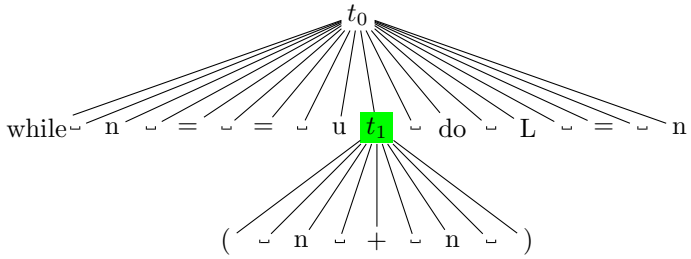


Figure 2.7: Naive Parse trees after pre-tokenisation in TREEVADA [12]

parentheses when bubbling. Additionally, two new heuristics, bubble length and bubble depth, are considered when considering bubble ranks during bubbling.

2.0.9 KEDAVRA

Proposed in "Incremental Context-free Grammar Inference in Black Box Settings" [15] in 2024, KEDAVRA is an algorithm designed to improve upon TREEVADA, in the same setting. Although TREEAVADA did improve upon ARVADA, it still had limitations of low accuracy, slow processing speeds, and limited readability due to complex grammar structures, which were inherited from ARVADA [2], [15]. The paper highlights that KEDAVRA outperforms ARVADA and TREEVADA in terms of grammar precision, recall, runtime/computational efficiency, and readability while maintaining similar memory usage.

The approach taken by KEDAVRA consists of 3 main parts:

- Tokenisation
- Data Decomposition
- Incremental grammar inference

Tokenisation

Similar to ARVADA and TREEVADA, KEDAVRA performs a tokenisation step. However, unlike ARVADA, which is tokenised based on class, and TREEAVADA, which extends it to consider brackets. KEDAVRA tokenises based on common lexical rules such as identifiers, strings, and numbers, and only considers whitespace if the given $\mathcal{O}$ is sensitive to it.

Furthermore, KEDAVRA during grammar inference performs **character-level generalisation** for token values. Given a token character type, it computes all possible characters corresponding to that character type. For instance, if a token is a lowercase letter, all lowercase letters are included in the set.

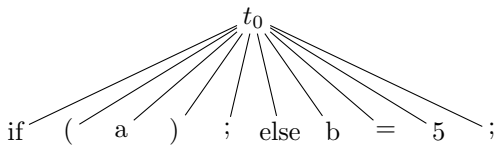
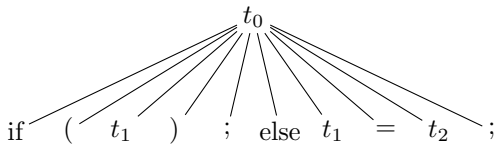


Figure 2.8: Result during KEDAVRA pre-tokenisation given input `if(a); else b = 5;`



$t_1 \rightarrow$ “a”|“b”|...|“z”

Figure 2.9: Result after KEDAVRA pre-tokenisation given input `if(a); else b = 5;`, and example generalisation of t_1

Data Decompostion

KEDAVRA then breaks down all complex example strings into simple components, while collectively preserving all grammatical structure of the original sequence and essentially, breaking the input string into smaller valid input strings. This is done to overcome the slowdown and lack of readability of generated grammars due to the complexity inherent in the entire example strings.

```
;
t2 ;
t1 ;
t1 = t2 ;
if ( t1 ) ;
while ( t2 ) ;
if ( t1 ) ; else ;
```

Figure 2.10: Decomposed Sequences of toeknised example from 2.9

Incremental Grammar inference

After Decomposition, KEDAVRA orders all the decomposed sequences by length and begins to infer grammar from the simplest token sequence, iterating over all token sequences derived from the input strings. Done similarly to ARVADA and TREEAVADA, KEDAVRA uses bubbling and merging. Also, like TREEVADA, only consider bubbles that have matched parentheses if parentheses are included.

2.0.10 NATGI

Published in "Black-Box Context-free Grammar Inference for Readable & Natural Grammars" at the University of Texas 2025[16], NATGI is a novel LLM-guided grammar inference framework that extends TREEAVADA's parse tree recovery. Aiming to improve upon the difficulties of human readability and the lack of guarantees on larger and realistic languages of previous work.

NATGI, as mentioned above, builds upon TREEAVADA's idea of bracket-induced tree structuring. It follows the same pre-tokenising step, adding two extra modifications. First, there is no separation of class between lowercase and uppercase letters, treating them as a single class. Second, it attempts to remove redundant whitespaces by iteratively removing whitespaces and checking if the resulting program is still valid. During this process, and each time a non-terminal is created via a merge, NATGI collects the subtrees that are about to be relabelled and prompts LLM models, given the sub-trees about to be merged for a more descriptive label compared to $t_1, t_2 \ldots$.

Then, before using any bubble heuristics process like ARVADA, TREEAVADA or KEDAVRA, NATGI further exploits the brackets. It automatically treats any sequence within enclosed brackets as a bubble and attempts to further structuralise the parse trees. This partially structured tree is then given to an LLM model with a specialised prompt to finish. If the LLM terminates due to LLM's limitations, such as hallucinations [16], [17] it then uses TREEAVADA heuristics and bubbling to compensate.

To allow a broader range of rules cmopared to the ones induced soley from the parse tree, NATGI decomposes each parse tree into smaller fragements. Allowing the resulting grammar to capture a broader set of language rules. This is done via effective pruning through the use of Hierarchical Delta Debugging (HDD). Where pruning is selective deleting branches of a trees. HDD is an algorithm that iteratively prunes the trees as long as a it fails a certain criteria [18].

Finally, NATGI performs lexical expansion. Since the leaf nodes are limited by the token in the input string, NATGI takes these leaf node and try to expand the lexicon accepted. For example, if leaf token is a single character, NATGI attempts to see if multiple characters are accepted.

2.0.11 Problem Statement

Due to the unknowns and gaps in the original paper [2]—including the lack of guarantees regarding the kinds of grammars ARVADA can handle and the complexity of the grammars it can learn—along with the suspicion that the original study may have been selective in nature, this thesis aims to achieve the following:

A replication of the study "Learning Highly Recursive Input Grammars" [2], through a clean-room re-implementation of ARVADA, with the intention of expanding its evaluation to

include a wider variety of grammars and identifying its key characteristics.

The clean-room re-implementation will mean development is entirely from scratch in a new environment, based solely on the explanations provided in the original paper. This will also enable an assessment of the adequacy of those explanations—specifically, whether a complete re-implementation is possible based on the information given or if the paper lacks essential details for reproducability.

Although the primary goal of this study is to re-implement, assess, and explain upon the original work, I acknowledge that this study may be incomplete or no new knowledge may be discovered. While this may seem trivial—merely confirming the findings of the previous study without adding novel insights [8], replication studies play a vital role in reinforcing the trustworthiness and confidence of empirical results, which is a central tenet of the scientific method. Replication can increase certainty when findings are reproduced and promote innovation when they are not [19]. Therefore, this study holds significance within the field of computer science.

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

Due to the incompletion of a full proper re-implementation in the given time frame, no test have been conducted and no evaluations have been made. However, reasons for incompletion and challenges faced during process are discussed in the discussion of the paper.

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] S. Mulik, S. Shinde, and S. Kapase, "Comparison of Parsing Techniques For Formal Languages,"

[8] M. Hendriks and V. Zaytsev, "Consider it Parsed!" Thesis, University of Twente, Enschede, Netherlands, 39 pp.

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

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

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

[12] M. R. Arefin, S. Shetiya, Z. Wang, and C. Csallner. "Fast Deterministic Black-box Context-free Grammar Inference," pre-published.

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

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

[15] F. Li et al. "Incremental Context-free Grammar Inference in Black Box Settings," pre-published.

[16] M. R. Arefin, S. Rahman, and C. Csallner. "Black-box Context-free Grammar Inference for Readable & Natural Grammars," pre-published.

[17] P. Orvalho and M. Kwiatkowska. "Are Large Language Models Robust in Understanding Code Against Semantics-Preserving Mutations?" Pre-published.

[18] G. Misherghi and Z. Su, "HDD: Hierarchical delta debugging," in *Proceedings of the 28th International Conference on Software Engineering*, Shanghai China: ACM, May 28, 2006, pp. 142–151, ISBN: 978-1-59593-375-1. DOI: 10.1145/1134285.1134307.

[19] M. Shepperd, "Replication studies considered harmful," in *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, May 27, 2018, pp. 73–76. DOI: 10.1145/3183399.3183423.

[20] R. Gopinath, B. Mathis, and A. Zeller, "Mining input grammars from dynamic control flow," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Virtual Event USA: ACM, Nov. 8, 2020, pp. 172–183, ISBN: 978-1-4503-7043-1. DOI: 10.1145/3368089.3409679.

[21] A. K. Lampinen. "Can language models handle recursively nested grammatical structures? A case study on comparing models and humans," pre-published.

[22] Open Science Collaboration, "Estimating the reproducibility of psychological science," *Science*, vol. 349, no. 6251, aac4716, Aug. 28, 2015. DOI: 10.1126/science.aac4716.

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