THE UNIVERSITY OF SYDNEY

# “Learning Highly Recursive Input Grammars”: A Replication Study

Sulav Malla
SID:500495980

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

Bachelor of Engineering Honours (Software)

**Supervisors:**
Dr. Rahul Gopinath, The University of Sydney
Dr. Yash Shirvastava, The University of Sydney

School of Electrical and Computer Engineering
The University of Sydney

November 07, 2025

**The University of Sydney**

**SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING**

# PROJECT CLEARANCE FORM

**Unit of Study Code and Name:** ELEC4713

**This is to certify that my student**

**Student Name:** Sulav Malla

**SID:** 500495980

**Has:**

- **Returned all books and reference material;**
- **Returned all equipment and keys; and**
- **Tidied their work place.**

**ECE Academic supervisor:**

**Signature:** yash Shrivastava **Date:** 07/11/25

**Name:** Dr Yash Shrivastava

**External supervisor (if applicable):**

**Signature:** **Date:** 07/11/25

**Name:** Dr Rahul Gopinath

# Abstract

Knowing the source code of a program greatly aids program comprehension, testing techniques such as fuzzing, optimisation, and debugging. However, due to various restrictions and other external factors, accessing source code is often not possible. To address this limitation, recent research has focused on inferring input grammars and execution behaviour directly from valid program inputs. The ARVADA algorithm, developed by Lemieux C. and Kulkarni N. at UCB in 2021, is a black-box approach designed to infer the grammar of a program using only valid inputs and a black-box oracle.

Motivated by the lack of formal guarantees and concerns about the selectiveness of the original study, this thesis attempts to re-implement the ARVADA algorithm in a clean-room environment using the C programming language. Although a complete implementation was not achieved, this paper presents a partial implementation of ARVADA algorithm in C with improvements of the function MergeAllValid, and highlights the challenges faced and insights gained during the re-implementation. These include the inherent complexity of ARVADA, the difficulty of implementing such an algorithm in C, and the limited clarity and weakeness in the explanation provided in the original paper.

# Statement of Achievements

The major achievements of this paper include the following:

- Partial re-implementation of ARVADA in a clean room environment in C, with improvements. The improvements include optimisation of MergeAllValid.
- Identification of areas in the original paper that were poorly explained, making the study difficult to reproduce.

# Acknowledgement

I would like to acknowledge my supervisors for this project. Firstly, I would like to thank **Dr. Rahul Gopinath** for providing me with the opportunity to undertake this project and for his continued support when I needed it. I would also like to thank **Dr. Yash Shrivastava** for agreeing to be my ECE supervisor.

Additionally, I wish to express my gratitude to my friends and family for their constant support, patience, and understanding throughout this. There were many high moments, low moments, and times when I felt completely stuck, but at the end am very gald to have taken this project.

**Thank you, Dr. Rahul Gopinath and Dr. Yash Shrivastava.**

# Contents

# List of Figures

# Glossary

# 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 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 "Synthesising 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, an explanation of ARVADA with a walkththrough, and the motivation for conducting this replication study. Then the paper explores related work that has been done since the publication of ARVADA, and clearly outline the problem statement. This is followed by an in-depth explanation of all the work and re-implementation that has been completed in C with reference to the code. Afterwards, a discussion highlighting and commenting on the overall process of conducting the replication study, difficulties faced during implementation, and limitation of the original paper.

And finally, a breif conclusion.

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

# Chapter 2

# GENERAL BACKGROUND & REVIEW OF LITERATURE

## 2.1 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 $A \to w$, 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].

## 2.2 What are Context-Free Grammars?

Depending on the set of production rules, each grammar can be classified according to the *Chomsky hierarchy* [9].

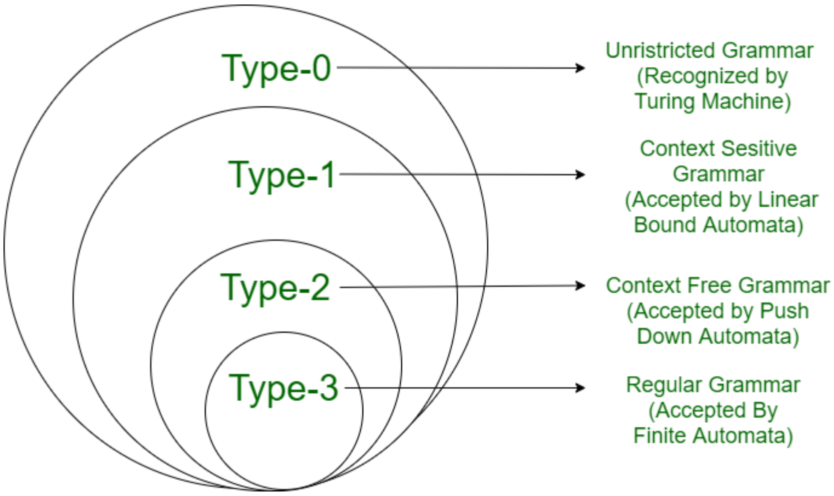

Type-0
Type-1
Type-2
Type-3
Unristricted Grammar (Recognized by Turing Machine)
Context Sesitive Grammar (Accepted by Linear Bound Automata)
Context Free Grammar (Accepted by Push Down Automata)
Regular Grammar (Accepted By Finite Automata)

**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*.

### 2.2.1 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 $X \rightarrow a|X \rightarrow aY$. 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.

$$X \rightarrow a$$

$$X \rightarrow aB$$

$$X \rightarrow \varepsilon$$

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

$$X \rightarrow a$$

$$X \rightarrow Ba$$

$$X \rightarrow \varepsilon$$

Note that right-linear and left-linear forms cannot be mixed, 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 [8], [11]

### 2.2.2 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].

### 2.2.3 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 [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 $\varepsilon$, meaning nullable productions are not allowed.

An example derivation using a context-sensitive production is:

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

### 2.2.4 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 $\alpha$ 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.

## 2.3 What is ARVADA?

### 2.3.1 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 $\mathcal{O}$.

### 2.3.2 GLADE

GLADE is an algorithm proposed by Bastani et al., published in Synthesising 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.

### 2.3.3 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.

### 2.3.4 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.

For this walkthrough, the simple while grammar $G_w$ and input strings $S$ provided in figure 2.2 will be used. Clarifying 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 $\mathcal{O}$ for understanding and comprehension.

$G_w$

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

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

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

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

**Algorithm 1** High-level overview of ARVADA [2]

```
Require: a set of examples $S$, a language oracle $\mathcal{O}$.
$bestTrees \leftarrow$ NaiveParseTrees($S$)
$bestTrees \leftarrow$ MergeAllValid($bestTrees, \mathcal{O}$)
$updated \leftarrow$ True
while $updated$ do
    $updated \leftarrow$ False
    $allBubbles \leftarrow$ GetBubbles($bestTrees$)
    for $bubble$ in $allBubbles$ do
        $bbldTrees \leftarrow$ Apply($bestTrees, bubble$)
        $accepted, mergedTs \leftarrow$ CheckBubble($bbldTrees, \mathcal{O}$)
        if $accepted$ then
            $bestTrees \leftarrow mergedTs$
            $updated \leftarrow$ True
            break
        end if
    end for
end while
$G \leftarrow$ InducedGrammar($bestTrees$)
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.

In this initially naive parse tree, for each string input. There is a non-terminal $t_0$, and all

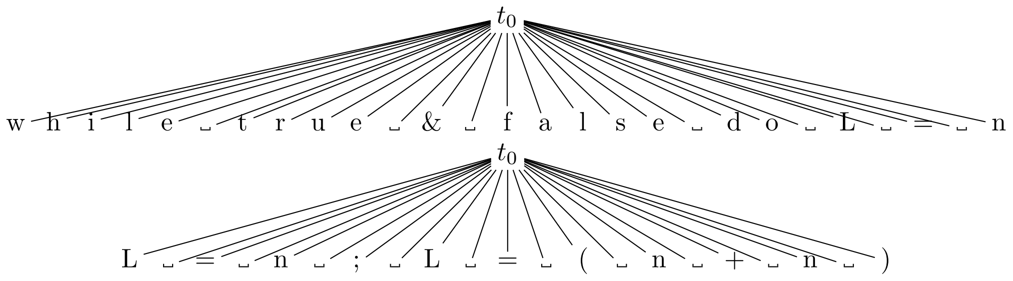

$t_0$
w h i l e ␣ t r u e ␣ & ␣ f a l s e ␣ d o ␣ L ␣ = ␣ n
$t_0$
L ␣ = ␣ n ␣ ; ␣ L ␣ = ␣ ( ␣ n ␣ + ␣ n ␣ )

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

characters in the string, including the spaces, are child nodes. From this alone, the following grammar can be induced.

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

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

The string derivable from a node $N$, in this case $t_0$, is the concatenation of all its leaf nodes or itself, if $N$ happens to be a leaf node.

Next, although not present in pseudocode 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 groups together sequences of contiguous characters of the same class (lowercase, uppercase, whitespaces, digits) into leaf tokens and all punctuations are kept separate. Essentially, grouping at the leaf node level.

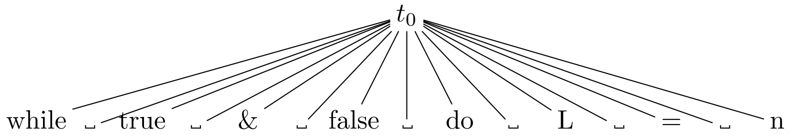


**Figure 2.4:** ARVADA pre-tokenisations example

Next, ARVADA attempts its first optimisation, performing MERGEALLVALID, which attempts to generalise and add structure 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 and/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 of each tree 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 the replica, new candidate strings can be derived; these candidate strings are fed into $\mathcal{O}$. The same process is done, where all instances of $t_a$ are replaced with $t_b$. Now having 2 sets of candidate strings, where in 1 set, $t_b$ replaces $t_a$, and other set, $t_a$ replaces $t_b$. 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 $t_c$, and a successful merge has occurred. Note roots $t_0$ are also non-terminal and can be $t_a$ or $t_b$.

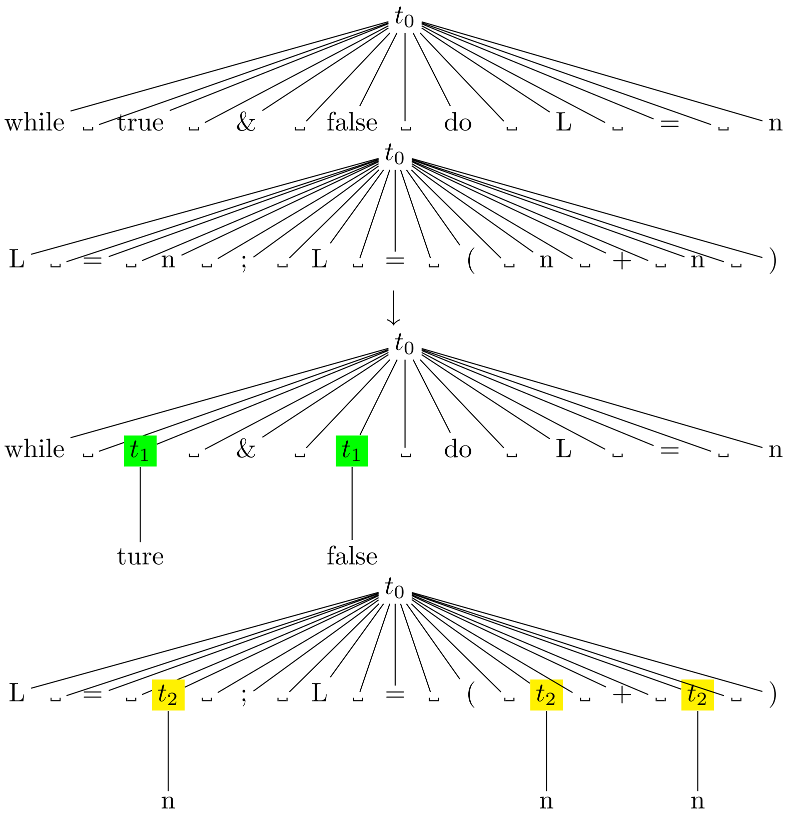

$t_0$
while ␣ true ␣ & ␣ false ␣ do ␣ L ␣ = ␣ n
$t_0$
L ␣ = ␣ n ␣ ; ␣ L ␣ = ␣ ( ␣ n ␣ + ␣ n ␣ )
↓
$t_0$
while ␣ $t_1$ ␣ & ␣ $t_1$ ␣ do ␣ L ␣ = ␣ n
ture
false
$t_0$
L ␣ = ␣ $t_2$ ␣ ; ␣ L ␣ = ␣ ( ␣ $t_2$ ␣ + ␣ $t_2$ ␣ )
n
n
n

**Figure 2.5:** Examples of a `MERGEALLVALID` run, tokenised and non tokenisation

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

The fundamental next step ARVADA perform is to bubble up a sequence of sibling nodes in the current trees $T$ into a new non-terminal. To bubble up a sequcne $s_1$ in trees $T$ a new non-terminal is created $t_{s1}$ with children $s_1$ and all occurrences of $s_1$ in each tree $t$ in $T$ with $t_{s1}$. Example shown in figure 2.6, where $t_3$ is a bubbled up. After bubbling up a sequence $s_1$, ARVADA either accepts or rejects the bubble. A bubble is only accepted if the enables a valid generalisation of the example string inputs. Meaning if the relabeling of the bubbled non-terminal, merging its label with the label of another existing node, expands the language accepted by the induced grammar while still remaining valid to $\mathcal{O}$. In practice, merging here is done in a similar fashion to the merging in MERGEALLVALID. Given 2 labels $t_a$ and $t_b$, ARVADA mutates trees from the current trees in $T$ such that subtrees rooted at $t_a$ are replaced by subtrees rooted at $t_b$ and vice versa, which, when concatenated, gives a candidate string which can be tested against the $\mathcal{O}$.

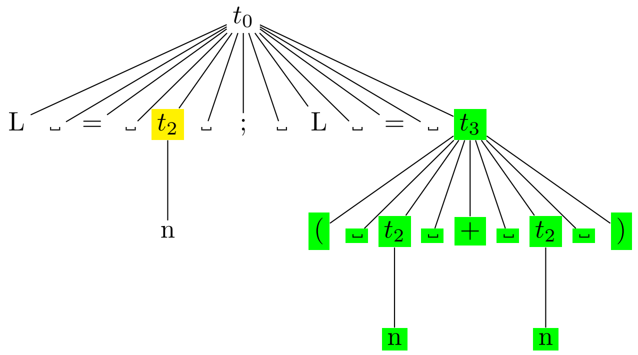


**Figure 2.6:** Possbile bubble example

Referring to the pseudocode in 1, from the current trees $T$ after MERGEALLVALID, ARVADA performs GETBUBBLES. For each tree $t \in T$, GETBUBBLES collects all proper contiguous subsequences of children in $t$. That is, if a tree contains a node $t_i$ with children $C = c_1, c_2, ..., c_n$, the potential bubble for this tree includes all sequences of length greater than 1 and less than n. Note that if the children are a subtree, the structure is maintained, like in figure 2.6. GETBUBBLES returns all these as 1-bubble, and all non-conflicting pairs of these as 2-bubbles. Two subsequences are non-conflicting if they do not strictly overlap: they can be disjoint or a proper subsequence of the other. So $((c_1, c_2, c_3), (c_4, c_5, c_6))$ and $((c_1, c_2, c_3), (c_2, c_3))$ are fine, however $((c_1, c_2, c_3), (c_2, c_3, c_4))$ is not. The purpose of 2 bubbles is to account for the limitations of 1-bubble. At some point, ARVADA will reach a stage where no 1-bubble can be merged with an existing label of a node in the tree, but more generalisations can still be made. Consider figure 2.8 (3), at this point in the run, ARVADA finds the single bubble $true \rightarrow t_5$ cannot be validly merged with any other existing node in the trees $T$. To cope with this, ARVADA bubbles up 2 distinct sequences (distinct or sub/super sets), only accepting the 2-bubble if they can merge, as seen in the example. $t_6 \rightarrow false$.

Now, ARVADA goes through 1-bubbles, taking 1 at a time and attempts to see if it can validly merge the current bubble with another non-terminal in the original trees. This is done in CHECKBUBBLE. If the merge is accepted, the trees are updated to reflect the successful merge; no further bubbles are checked, all current bubbles are discarded, and the process is repeated with the new updated trees. Meaning, new bubbles are formed and checked.

ARVADA to optimise checking bubbles that are more likely to get accepted first uses certain heuristics to order the exploration of the bubbles. Primarily sorting by similarity in context first, and frequency second and taking the top 100. Technical details are explained in the implementation.

After all the bubbles, 1-bubbles and 2-bubbles are exhausted, it looks at the final stage of the parse trees and returns the induced grammar. More technical explanations are provided in the implementation section of this paper.

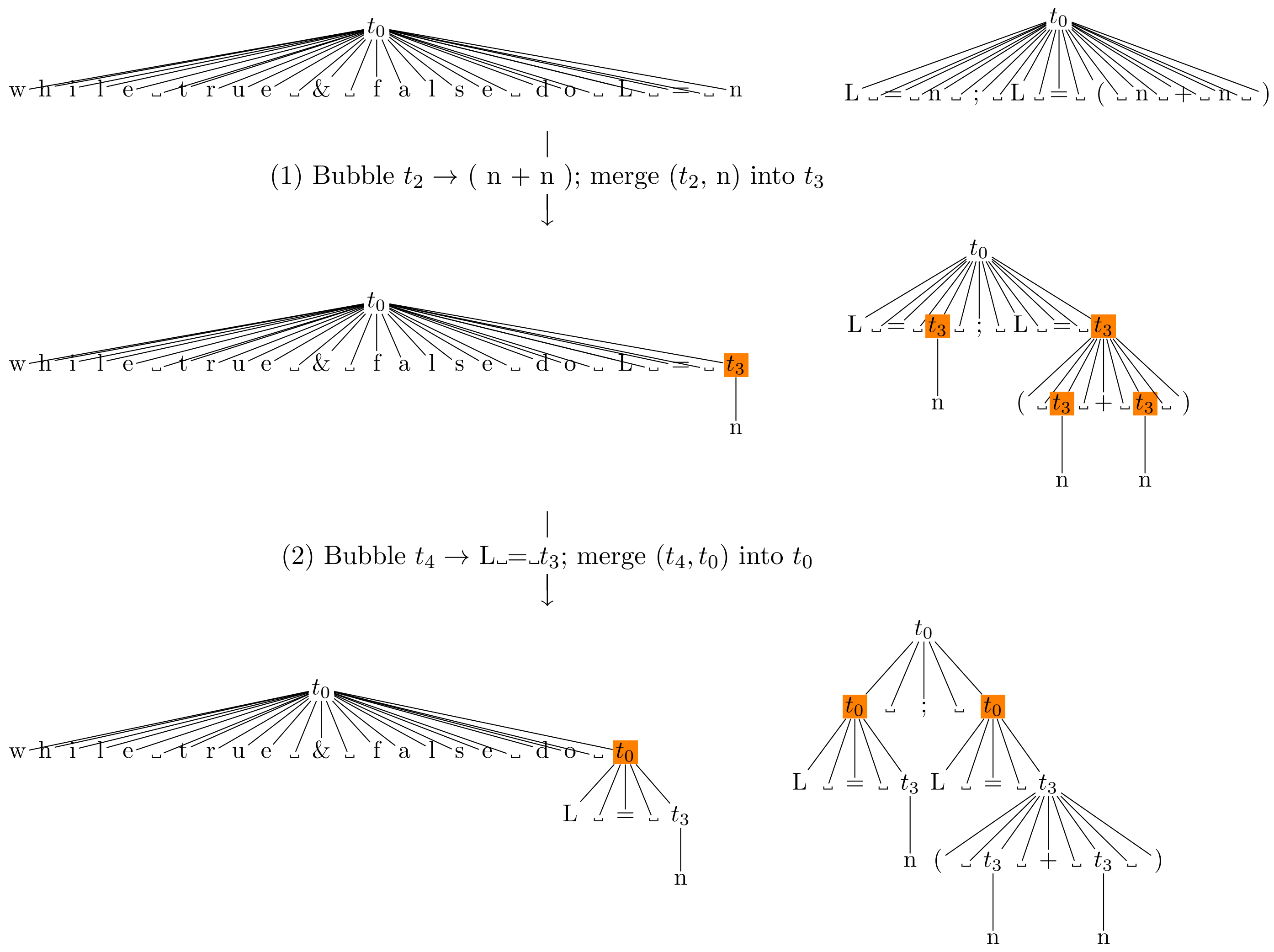

$t_0$
w h i l e ␣ t r u e ␣ & ␣ f a l s e ␣ d o ␣ L ␣ = ␣ n
$t_0$
L ␣ = ␣ n ␣ ; ␣ L ␣ = ␣ ( ␣ n ␣ + ␣ n ␣ )
(1) Bubble $t_2 \rightarrow$ ( n + n ); merge ($t_2$, n) into $t_3$
$t_0$
w h i l e ␣ t r u e ␣ & ␣ f a l s e ␣ d o ␣ L ␣ = ␣ $t_3$
n
$t_0$
L ␣ = ␣ $t_3$ ␣ ; ␣ L ␣ = ␣ $t_3$
n
( ␣ $t_3$ ␣ + ␣ $t_3$ ␣ )
n n
(2) Bubble $t_4 \rightarrow$ L␣=␣$t_3$; merge $(t_4, t_0)$ into $t_0$
$t_0$
w h i l e ␣ t r u e ␣ & ␣ f a l s e ␣ d o ␣ $t_0$
L ␣ = ␣ $t_3$
n
$t_0$
$t_0$ ␣ ; ␣ $t_0$
L ␣ = ␣ $t_3$ L ␣ = ␣ $t_3$
n ( ␣ $t_3$ ␣ + ␣ $t_3$ ␣ )
n n

**Figure 2.7:** Example full run of ARVADA bubbling using non-tokenisation nor optimised tree, and the grammar induced (part 1)[2]

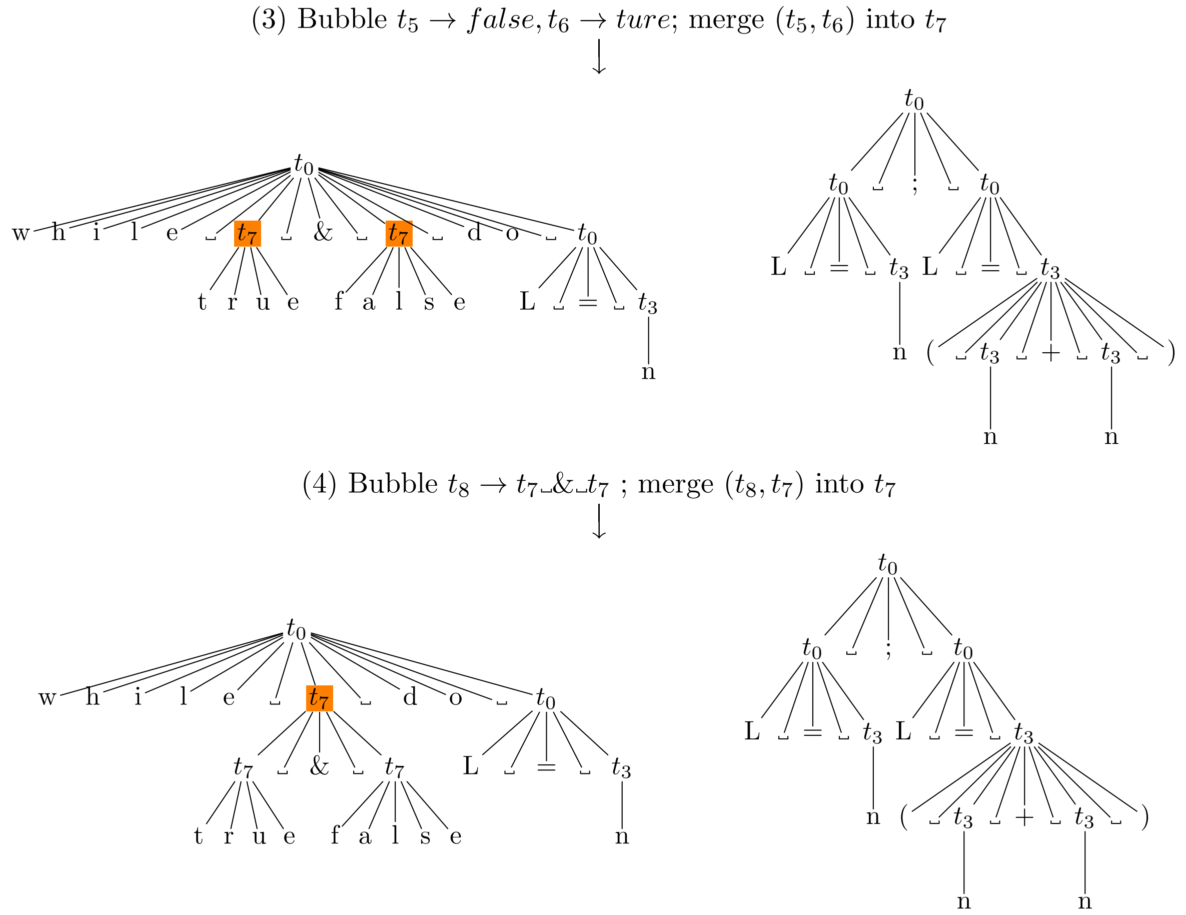

(3) Bubble $t_5 \rightarrow false, t_6 \rightarrow ture$; merge $(t_5, t_6)$ into $t_7$
(4) Bubble $t_8 \rightarrow t_7␣\&␣t_7$ ; merge $(t_8, t_7)$ into $t_7$

**Figure 2.8:** Example full run of ARVADA bubbling using non-tokenisation nor optimised tree, and the grammar induced (part 2)[2]

## 2.4 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, the possibility exists 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.

## 2.5 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 prioritised, 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.

## 2.6 Why is learning input grammar important?

Grammar inference is important for many software engineering tasks, as knowledge about a program's grammar helps with code comprehension, reverse engineering, detecting and refactoring code smells, transforming source code for optimisation or bug fixing, and generating test inputs cite arefinFastDeterministicBlackbox2024. However, due to the increasing restrictions in many software systems—caused by global privacy and security regulations—we often lack access to the source code needed to learn their grammar. Even with open-source programs and code, many only have closed-source parsers, making white-box or grey-box instrumentation 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.

## 2.7 Related Work

### 2.7.1 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 pre-tokenisation, 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.

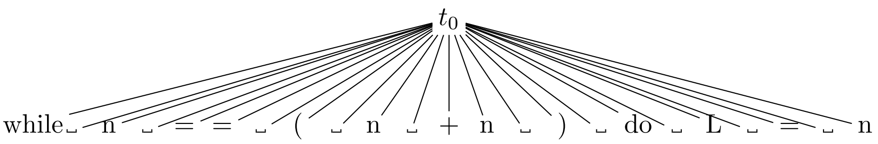


**Figure 2.9:** Naive Parse trees after pre-tokenisation in ARVADA

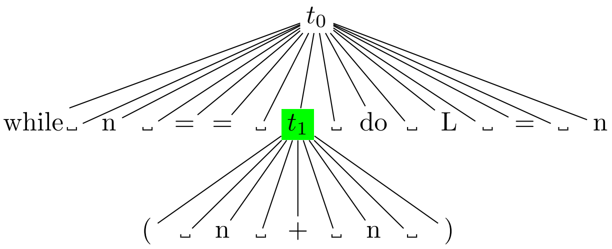


**Figure 2.10:** Naive Parse trees after pre-tokenisation in TREEVADA [12]

Secondly, to address the non-deterministic aspect of ARVADA, TREEAVADA switches to a deterministic data structure for specific operations and discards all bubbles with unmatched parentheses when bubbling. Additionally, two new heuristics, bubble length and bubble depth, are considered when considering bubble ranks during bubbling.

### 2.7.2 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.

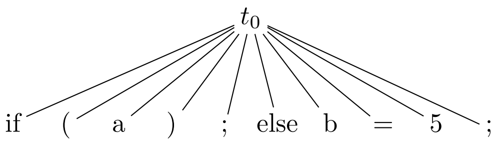
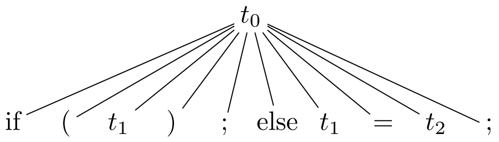


**Figure 2.11:** Result during KEDAVRA pre-tokenisation given input `if(a); else b = 5;`

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.



$$t_1 \rightarrow \text{“a”}|\text{“b”}|...|\text{“z”}$$

**Figure 2.12:** 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.

### 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.

```
;

t2 ;

t1 ;

t1 = t2 ;

if ( t1 ) ;

while ( t2 ) ;

if ( t1 ) ; else ;
```

**Figure 2.13:** Decomposed Sequences of toeknised example from 2.12

### 2.7.3 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 $t_1, t_2...$ .

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 compared to the ones induced solely from the parse tree, NATGI decomposes each parse tree into smaller fragments. 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). Pruning is the selective deletion of branches of a tree. HDD is an algorithm that iteratively prunes the trees as long as it fails a certain criterion [18].

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

## 2.8 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 expand 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

This section of the thesis aims to provide a clear explanation of how the re-implementation was conducted in C. Providing all the steps taken in this attempt, with as much technical detail as possible and referencing appropriate sections of the original paper and code.

## 3.1 Data structures, and Initial Parse Trees

### Data Structures

During this process of re-implementation, 2 structures were initialised. Structures are a way to group several related variables into one place in C [20]. One structure to represent a node in the parse trees, and another structure to hold and reference each parse tree.

Nodes Structure for storing trees

```
typedef struct nodes{
    int capacity;
    int count;
    struct node **rootNodes;
} Nodes;
```

**Figure 3.1:** Data Structure to store and track trees

Figure 3.1 shows the structure *Nodes*, used to hold all the parse trees. It has variables called `rootNodes`, which is a list containing pointers to the roots of all the parse trees. Now, as this implementation is in C and memory management is a dynamic and manual process, this structure holds 2 more variables `capacity` and `count`. Both of these variables are integers that are used to help manage the size of `rootNodes`. Variable `capacity` tracks the number of root node pointers `rootNodes` can currently hold, and variable *count* tracks the number of actual root node pointers $rootNode$ currently has.

Node Structure to build each tree

```
typedef struct node{
    int capacity;
    char character;
    struct node *parent;
    int t_label;
    int num_child;
    int pos;
    struct node **children;
} Node;
```

**Figure 3.2:** Data Structure to build each tree

Figure 3.2 shows the structure *Node*, used for each individual node in the trees. It has 7 variables inside it.

- `character`: stores a C character if the node is a leaf/terminal node, else a null character if the node is a non-terminal.
- `t_label`: stores an integer. The integer is a positive value if the node is a non-terminal, else -1 if it is a terminal.
- `parent`: hold a pointer to the current node's parent, which is a *Node* as well. If root, parent is a null pointer.
- `pos`: is an integer which represents the index of the current node when the parse trees are initially constructed for all terminal nodes. For non-terminal nodes, it will be a null value.
- `children`: hold a list of pointer, which point to *Node*.
- `capacity`: current size of the list *children*.
- `num_child`: current number of *Node*(s) pointer in list `children`.

### t_id

```
// Making tid global
extern int *tid;
tid = calloc(1, sizeof(int));
```

**Figure 3.3:** Initialising a global tid variable

An global varible, `tid` was use to keep track of node non-terminal node labelling. It is incremented each time a new non-terminal is created and a tid is assigned. The `tid` is stored in `t_label` of the node. Noting it is not used to assign t_label of root nodes, as all root nodes have label 0.

## 3.2 Building Initial parse trees

This part of the thesis is derived from section III-A of the original paper [2]. Using the 2 structures described, next, the initial parse trees were built. First, we build the *Nodes* structure, which houses all the parse tree pointers. This is done by using C's built-in dynamic memory function *malloc*, where *capacity* was set randomly to 4, *count* to 0, as it does not house any parse tree pointers yet, and list *rootNodes* was given enough memory to house 4 *Node* pointers, where 4 is the current *capacity*.

```
    // Keeping track of all root trees (each string in example file)
    Nodes *root_trees= malloc(sizeof(Nodes));
    root_trees->capacity = 4; // randomly assigned
    root_trees->count = 0;
    root_trees->rootNodes= malloc(root_trees->capacity * sizeof(Node*));
```

**Figure 3.4:** building of Nodes Structure

Next, the file containing all the valid example strings is read as standard. Refer to figure 3.5, then from the top of the file, each valid string is read and each naive flat parse tree is built one by one. This is done by, looping through all the valid string and building a root node $t_0$. Then for each strings, looping through each character, builing a node to house this character. The doing appropriate assignings to make this character node a child of the root node. After each tree is built, the pointer to the tree's root if given and housed in `root_trees` from figure 3.4.

### 3.2.1 Memory management

As the number of valid example strings and the length of each example string are not pre-determined, the number of `Nodes` (each parse tree), and the number of leaf nodes each non-terminal root node is going to is also non-determined. This means memory has to be allocated dynamically. To do this, two functions are used: one function to check and increase memory allocated to `root_nodes` in `Nodes`, and another to check and increase memory allocated to `children` in `Node`. Both these functions check the `capacity` of their respective structures, and if `count` for `Nodes` and `num_child` for `Node` equals `capacity`, they increase the memory allocated to their respective lists by 1.5 times the `capacity`, updating other variables appropriately. Function code in the appendix 6.2

Since memory is being assigned dynamically, it also has to be freed dynamically. To help with this, a function is used. This helper function is given a node that does a depth-first search (DFS), and frees nodes as it searches, working its way bottom up. This is used to free memory as needed throughout, and all memory at the end. Refer to appendix 6.3

```c
// Buffer to store the current example read.
char *current_line = NULL; //
size_t line_buffer_len = 0;
ssize_t read_line_len = 0;

//Read line by line until the end of the file has been reached.
while((read_line_len = getline(&current_line, &line_buffer_len, file_ptr)) != -1){

    current_line[read_line_len - 1] = '\0';

    //building the naive parse tree for each string
    // So the root node for each parse tree
    Node *current_tree = build_basic_node(); // appendix 6.1
    current_tree->capacity = 10; // randomly assigned
    current_tree->t_label = 0;
    current_tree->children = calloc(current_tree->capacity, sizeof(Node*));

    // Going through all the characters in the string 1 by 1.
    // Building the terminal nodes, assign a character value to the node.
    for(int i = 0; i < (read_line_len - 1); i ++ ){
        Node * node = build_basic_node();
        node->parent = current_tree;
        node->character = current_line[i];
        node->pos = i;
        current_tree->children[current_tree->num_child] = node;
        current_tree->num_child ++;
        // checking current trees' capacity
        // increase appropriately if needed
        check_node_capacity(current_tree);

    }

    // Check current capacity of the root node.
    // Increase appropriately
    root_trees->rootNodes[root_trees->count] = current_tree;
    root_trees->count = root_trees->count + 1;
    check_nodes_capacity(root_trees);

}

// Free buffer
free(current_line);
```

**Figure 3.5:** Building each initial parse tree

## 3.3 Pre-tokenisation

```
    // Step 2: Toggling pre-tokenisation
    // Section III-E of the Original paper
    int tokenise = 0;

    if(tokenise){
       for (int i = 0; i < root_trees->count; i ++){
           pre_tokenise(root_trees->rootNodes[i]);
       }
    }
```

**Figure 3.6:** Pre-tokenisation of each root node

After the initial parse trees were built, pre-tokenisation, as described in Section III-A of the original paper [2], was implemented. This process involved looping through each parse tree in the `Nodes` structure and, for each parse tree, iterating through all its leaf nodes. The goal was to group consecutive sequences of leaf nodes belonging to the same class and place them under an intermediate node positioned between the root node and the sequence of leaf nodes—also referred to as a label node. The pre-tokenisation of each parse tree was encapsulated within a dedicated function, which performed pre-tokenisation given the root node $t_0$ of a parse tree.

Given a root node $t_0$, the function first detaches all of its children $(c_0, c_1, \ldots, c_n)$ from the root. This is achieved by assigning an empty list to the root's children list pointer and storing the original list of children $(c_0, c_1, \ldots, c_n)$ in a temporary variable. It then creates a new intermediate non-terminal label node $t_i$ and began iterating through the detached children. The function takes the first child node $c_0$, assigns it as a child of $t_i$, and keep track of its class and the number of nodes assigned to $t_i$ (i.e., the length of the current sequence of the same class). It then continues through $c_1, c_2, \ldots, c_n$, and as long as the current child $c_i$ belonged to the same class, it continued assigning it as a child of $t_i$.

When a punctuation mark or a node $c_i$ of a different class was encountered, and $t_i$ contains a sequence of more than one child, the function assigns increments and assign $t_i$ a `tid`. Then assign $t_i$ as a child of the root $t_0$ and creates a new non-terminal node $t_j$ to begin a new sequence. This process is then repeated. If $t_i$ contains a sequence of length one when a class change occurrs, and a new sequence is about to begin, $t_i$ is dissolved, and the single child $c_p$ is directly linked to the root $t_0$. The process is then restarted with a new non-terminal label node $t_k$. The same approach is applied when punctuation marks are encountered.

Essentially, this process ensures that intermediate non-terminal label nodes are only created for leaf nodes of the same class that form consecutive sequences of length two or greater. Refer to Appendix 6.4 and Appendix 6.5 for the implementation details.

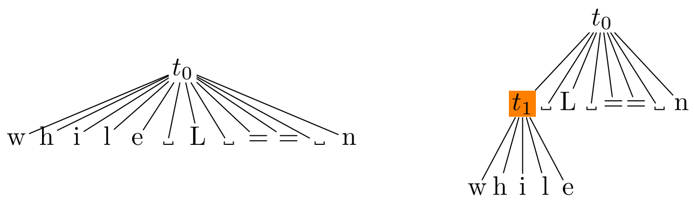


**Figure 3.7:** Visualisation of adding a intermediate node, and tokenisation

## 3.4 MERGEALLVALID

Following the explanation provided in Section III-A of the original paper [2], the function MERGEALLVALID was implemented. After pre-tokenisation, another loop is executed that iterates through each parse tree $T_i$ and calls the MERGEALLVALID function on it.

```
// Step 3: MergeAllValid
// Section III-A of Original paper
for( int i = 0; i < root_trees -> count; i++){
    merge_all_valid(root_trees->rootNodes[i], root_trees);
}
```

**Figure 3.8:** MergeAllValid call

Referencing Appendix 6.6, once the function was given a parse tree $T_i$ with root $t_0$, it iterates through all its children, which are either leaf nodes ($c_i$) or non-terminal label nodes ($t_i$). For each child node $c_i/t_i$, the function then loops from the current node $c_{i+1}/t_{i+1}$ to the final node $c_n/t_n$, effectively generating all combinations of two child nodes ($t_a/c_a$ and $t_b/c_b$). This looping process is executed twice.

While ignoring whitespace nodes as potential $t_a/c_a$ or $t_b/c_b$, the first time looping, it checks for pair of nodes $t_a/c_a$ and $t_b/c_b$ that produce the same string when concatenated. String concatenation, given a node, is a done by doing a depth-first search (DFS) traversal to the leaf nodes, retrieving each character, and storing it in a buffer (see Appendix 6.7).

If $t_a/c_a$ and $t_b/c_b$ produces identical concatenated strings, the function merged both pairs ($t_a/c_a$ and $t_b/c_b$). This merging process is carried out through another function (Appendix 6.8), which creates two new intermediate nodes with identical labels (same `t_label`) if the two child nodes being compared are leaf nodes ($c_a$ and $c_b$). Otherwise, if the two nodes were non-terminals ($t_a$ and $t_b$), the function simply updated their `t_label` values accordingly, assigning the smaller label value to the larger non-terminal.

Once all identical string-producing nodes had been processed, ARVADA iterated through all the parse trees again, generating combinations of two child nodes once more—this time dealing with every $t_a/c_a$ and $t_b/c_b$ pair that produced different concatenated strings. Before merging these nodes, several checks were performed. The function first checkes for whitespace nodes, but this time to exclude nodes that produced identical strings. It also checked whether the nodes under consideration, $t_a$ and $t_b$, had already been merged. This was determined by verifying if $t_a$ and $t_b$ shared the same non-negative `t_label`.

After this, to process all not identical nodes, a different check is done to merge. It loops through all the trees, dulicates the tree and hands it to another funciton, `string replacement`. `String replacement` checks if $t_a/c_a$ can replace $t_b/c_b$ in all parse trees and still produce valid concatenated strings—and vice versa ($t_b/c_b$ replacing $t_a/c_a$). If success full, then it is merge. In depth explanation provided in String replacement section of the paper. The merging of two different node sequences was conducted in a similar manner to the merging of identical sequence nodes, as explained in the merging section.

In this re-implementation, because the `MERGEALLVALID` function first handles identical subtrees, two different merge functions were implemented: a general merge function (Appendix 6.9) and a more specialised merge function used exclusively within `MERGEALLVALID` (Appendix 6.8).

The general merge function is used in `MERGEALLVALID` and can also be applied after the bubbling process in `CHECKBUBBLES`. This function takes two nodes as input, and if either of the given nodes is a terminal (leaf) node, it immediately introduces a non-terminal intermediate node—effectively adding an additional layer to the tree structure (similar to the operation performed during pre-tokenisation). This intermediate node is then used for re-labelling. Refer to Figure 3.7 for a visualisation of adding an intermediate node.

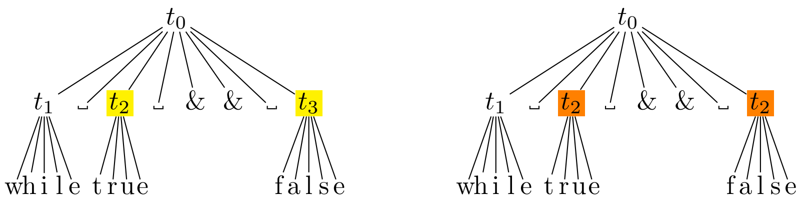
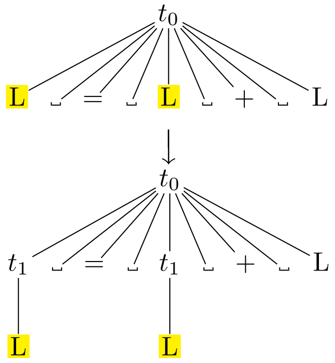
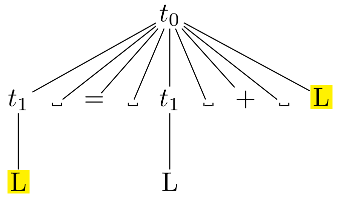

$t_0$
$t_1$ ␣ $t_2$ ␣ & & ␣ $t_3$
w h i l e t r u e f a l s e
$t_0$
$t_1$ ␣ $t_2$ ␣ & & ␣ $t_2$
w h i l e t r u e f a l s e

**Figure 3.9:** Visualisation of re-labelling

The more specialised merge function, used only in `MERGEALLVALID`, was designed to handle cases where there are more than two identical leaf nodes directly linked to the root of a parse tree during the initial traversal. Consider the following example:



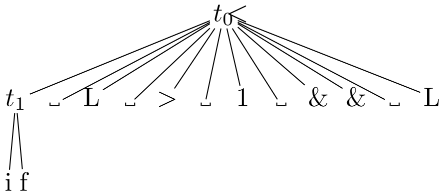
The highlighted nodes represent the nodes under consideration in the loop. The first time two identical direct leaf nodes are found, the process is straightforward: both nodes are wrapped under newly created intermediate nodes with the same label, updating the tree structure accordingly. However, since the loop continues without restarting after the modification, a situation may arise as shown below:



Here, the two identical nodes remain leaf nodes, but only one of them is still directly connected to the root. Therefore, to merge these nodes correctly, only the one directly attached to the root requires an intermediate node. The specialised merge function handles this case by performing an additional check to determine whether the nodes are directly connected to the root before merging.

## 3.5 String replacement

From sections III-D of the original paper [2], string replacement is the check done before before merging any 2 none identical nodes, appendix 6.10. String replacement, given 2 nodes $t_a/c_a$ (replacer) and $t_b/c_b$ (replacee) and a root tree $T_i$, gets all the permutation of the ways $t_a/c_a$ can replace $t_b/c_b$ in $T_i$. Then concatenates each permutation, which becomes a candidate string which is fed into the oracle to validate. For example take,



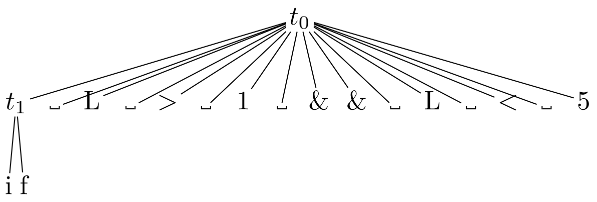
and let $t_a$ be $t_1$ and $t_b$ be L. From this the doing the permutation, the candidate string produced would be.

$$\text{if } L > 1 \;\&\&\; L < 5$$
$$\text{if if} > 1 \;\&\&\; L < 5$$
$$\text{if if} > 1 \;\&\&\; \text{if} < 5$$
$$\text{if } L > 1 \;\&\&\; \text{if} < 5$$

From Section III-D of the original paper [2], *string replacement* is the verification step performed before merging any two non-identical nodes. Given two nodes, $t_a/c_a$ (the replacer) and

$t_b/c_b$ (the replacee), and a root tree $T_i$, the string replacement process generates all permutations of the ways $t_a/c_a$ can replace $t_b/c_b$ within $T_i$. Each permutation is then concatenated to form a candidate string, which is subsequently passed to the oracle for validation.

For example, consider the following tree:



Let $t_a$ represent $t_1$ and $t_b$ represent $L$. After performing all permutations, the candidate strings generated would be:

$$\text{if } L > 1 \,\&\&\, L < 5$$

$$\text{if if} > 1 \,\&\&\, L < 5$$

$$\text{if if} > 1 \,\&\&\, \text{if} < 5$$

$$\text{if } L > 1 \,\&\&\, \text{if} < 5$$

To achieve this, the string replacement process uses recursion. The function takes as input the replacer, the replacee, the position (an integer indicating the starting point of the loop in each recursive call), a duplicate of the parse tree, and a interger pointer (initially set to 1). The interger is what is checked, if the integer pointer remains 1 at all the way to the end, it means all candidate string were valid, and merge can happen, else a candidate has failed and no merge. It

Since this is a recursive function, its first check determines whether the integer pointer still holds the value 1 (the base condition). If, at any point, this value becomes 0, it indicates that a candidate string has failed, and no further checks are required; the function exits immediately. If the value remains 1, the function then checks whether the replacee is a leaf node, setting a corresponding terminal flag if true. It then initiates a for-loop starting at position 0, iterating through each child node.

If the terminal flag is set, the function checks whether the replacee's character matches the current child's character. If it matches, the function proceeds; otherwise, it continues to the next iteration of the loop. If the replacee is a non-terminal node, the function instead compares the current child's `t_label` value, proceeding only if it matches the replacee's `t_label`. The replacee is then temporarily replaced by the replacer, and the function recursively calls itself using the same replacer and replacee, but starting at the index of the current iteration. After the recursive call, the function concatenates the root and passes it to the oracle, which evaluates the candidate string and updates the integer pointer accordingly (the oracle implementation is explained in the next section). Once the recursion returns, the replacee is restored to its original state, and another recursive call is made using the same replacer, replacee, and current index. This ensures that all possible permutations are considered.

This section is inherently abstract and best understood in conjunction with the source code provided in Appendix 6.10.

## 3.6 Oracle and other testing

### 3.6.1 Bubbling

This section of ARVADA has not been implemented yet. Hence, further future work is required to complete the re-implementation.

# Chapter 4

# EVALUATION

Due to the incompletion of a full 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 section of this paper.

# Chapter 5

# DISCUSSION

// Limitation with this re-implementation MergeALLVALID only prefrom merging in each parse tree and not across trees.

# Chapter 6

# CONCLUSION

This paper presents a re-implementation of a state-of-the-art algorithm in a clean-room environment using the C programming language. Although a complete re-implementation was not achieved, this paper provides a detailed account of all the steps undertaken of what was achieved, a partial re-implementation. This partial re-implementation providing a starting point for future research and development related to ARVADA.

Alongside the re-implementation, this paper highlights the challenges encountered and identifies areas in the original work where explanations were insufficient or ambiguous. All source code references have been documented, and the codebase has been made openly available for review and further improvement.

# 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] W3School. “C Structures (structs),” C Structures (structs).

[21] 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`.

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

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

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

# Appendix

```c
Node *build_basic_node(){

    Node * node = malloc(sizeof(Node));
    node->parent = NULL;
    node->capacity = 0;
    node->character = '\0';
    node-> t= -1;
    node->num_child = 0;
    node->pos = -1;
    node->children = NULL;
    return node;
}
```

**Figure 6.1:** Code for building a basic node

```c

// Function that checks Node capacity and count.
void check_nodes_capacity(Nodes *nodes){

    // If number for Node's is less than current capacity, return.
    if(nodes->capacity > nodes->count){
        return;
    }

    // If number of childern has reached capacity, update cap.
    int cur_cap = nodes->capacity;
    nodes->capacity = (cur_cap + (cur_cap/2) + 1);
    void *new_space = realloc(nodes->rootNodes,nodes->capacity * sizeof(Node*));
    if(errno == ENOMEM || new_space == NULL){
        fprintf(stderr, "Error increasing the capicty of nodes.");
        return;
    }
    nodes->rootNodes= new_space;

}

// Fucntion that checks Node capacity and num_child.
void check_node_capacity(Node *node){

    if(node->capacity > node->num_child){
        return;
    }

    // If number for Node's is less than current capacity, return.
    int cur_cap = node->capacity;
    node->capacity = (cur_cap + (cur_cap/2) + 1);
    void *new_space = realloc(node->children,node->capacity * sizeof(Node*));
    if(errno == ENOMEM || new_space == NULL){
        fprintf(stderr, "Error increasing the capicty of nodes.");
        return;
    }
    node->children= new_space;

}
```

**Figure 6.2:** Code for changing list capacities appropriately

```
// Function to free all the nodes given a root Node
void free_tree(Node *root){

    // dfs, freeing children nodes first
    if (root->num_child > 0){
        for( int i = 0; i < root->num_child; i++){
            free_tree(root->children[i]);
        }
        free(root->children);
    }
    free(root);
}
```

**Figure 6.3:** Code for freeing tree given a root node

```c
void pre_tokenise(Node* root){

    // Flags to count contigious seuqences.
    int sequence_begun = 0;
    int sequence_length = 0;
    // class 1 = Whitespaces
    // class 2 = Uppercase
    // class 3 = Lowercase
    // class 4 = Digits

    // Set basic node with a list up.
    // parent and children of the node assigned in progress.
    Node *tmp = build_basic_node_with_list();

    // num of child and parent save in a local var,
    // helps later for memory management
    int cur_children_num = root->num_child;
    Node ** cur_children = root->children;
    root->capacity = 1;
    root->num_child = 0;
    root->children= calloc(root->capacity, sizeof(Node*));


    for( int i = 0; i < cur_children_num; i++){

        Node * cur_node= cur_children[i];
        // Current character is a whiteSpace
        if(cur_node->character == ' '){
            check_and_tokenise(1, &sequence_length, &sequence_begun, &tmp, cur_node);

        // Current character is an uppercase character
        }else if(isupper(cur_node->character)){
            check_and_tokenise(2, &sequence_length, &sequence_begun, &tmp, cur_node);

        // Current character is a lower character.
        }else if(islower(cur_node->character)){
            check_and_tokenise(3, &sequence_length, &sequence_begun, &tmp, cur_node);

        // Current character is a digit.
        }else if(isdigit(cur_node->character)){
            check_and_tokenise(4, &sequence_length, &sequence_begun, &tmp, cur_node);

        // Current character is punctuation
        }else{

            // sequence of size 1 in progress, remove warpper
            if(sequence_length == 1){
                root->children[root->num_child] = tmp->children[0];
                root->num_child ++;
                check_node_capacity(root);
                tmp->num_child = 0;
                tmp->children[0] = NULL;
                free((tmp)->children);
                free_tree(tmp);

            }else if(sequence_length >1 ){
                // multi character sequence, close and attach
                tmp->parent = tmp->children[0]->parent;
                (*tid) ++;
                tmp->t_label = *tid;

                Node *p = cur_node->parent;
                if (p) {
                    p->children[p->num_child] = tmp;
                    (p->num_child)++;
                    check_node_capacity(p);
                }

            }

            tmp = build_basic_node_with_list();
            root->children[root->num_child] = cur_node;
            root->num_child++;
            check_node_capacity(root);
            sequence_begun = 0;
            sequence_length = 0;
        }
    }

    // End of loop is reached, deal with the left over unclosed sequence
    if(sequence_length == 1){
        root->children[root->num_child] = tmp->children[0];
        root->num_child ++;
        check_node_capacity(root);
        tmp->num_child = 0;
        tmp->children[0] = NULL;
        free((tmp)->children);
        free_tree(tmp);
    }else if(sequence_length == 0){
        free_tree(tmp);
    } else {
        root->children[root->num_child] = tmp;
        root->num_child ++;
        tmp->parent = root;
        (*tid) ++;
        tmp->t_label = *tid;
        check_node_capacity(root);
    }
    free(cur_children);
}
```

**Figure 6.4:** Pre-tokeniser code

```
    void check_and_tokenise(int cur_class, int *sequence_length, int *sequence_begun, Node **tmp,
         Node *cur_node){

    // A sequence of the same class as currnet node is in progress.
    if (*sequence_begun == cur_class && *sequence_length > 0){

        // Because the parent of the first child is never assigned,
        // this is done to deal with cases where a sequence of just
        // length 1 is found.
        if (*sequence_length == 1 && (*tmp)->num_child > 0) {
            (*tmp)->children[0]->parent = *tmp;
        }
        // Assigning the current node to tmp,
        // increase sequence number.
        cur_node->parent = *tmp;
        (*tmp)->children[(*tmp)->num_child]= cur_node;
        (*tmp)->num_child ++;
        check_node_capacity(*tmp);
        (*sequence_length)++;

    // no sequence or a different sequence has begun
    }else{

        // different sequence of length 1 is in progress.
        if(*sequence_length == 1){
            // previous sequence was just one node, discard wrapper
            Node *p = cur_node->parent;
            if (p) {
                p->children[p->num_child] = (*tmp)->children[0];
                (p->num_child)++;
                check_node_capacity(p);
                }

            (*tmp)->num_child = 0;
            (*tmp)->children[0] = NULL;
            free((*tmp)->children);
            free_tree(*tmp);
            *tmp= build_basic_node_with_list();
            (*tmp)->children[(*tmp)->num_child] = cur_node;
            (*tmp)->num_child ++;
            check_node_capacity(*tmp);

        // different sequence of length greater than 1 has begun.
        }else if(*sequence_length > 1){
            // finalise previous multi-node sequence
            (*tmp)->parent = (*tmp)->children[0]->parent;
            *tid = *tid + 1;
            (*tmp)->t_label = *tid;


            Node *p = cur_node->parent;
            if (p) {
                p->children[p->num_child] = *tmp;
                (p->num_child)++;
                check_node_capacity(p);
            }

            *tmp = build_basic_node_with_list();
            (*tmp)->children[(*tmp)->num_child] = cur_node;
            (*tmp)->num_child ++;
            check_node_capacity(*tmp);
            *sequence_length = 1;

        // no sequence in progress, assign this node as first node,
        // and begin the sequence.
        }else if(*sequence_begun == 0) {
            (*tmp)->children[(*tmp)->num_child] = cur_node;
            (*tmp)->num_child ++;
            check_node_capacity(*tmp);
            *sequence_length = 1;
        }
    }
    *sequence_begun = cur_class;
}
```

**Figure 6.5:** Pre-tokenisation helper code

```c
void merge_all_valid(Node *root, Nodes *all_trees){

    // Go through the list 1 once
    for( int i = 0; i < root->num_child; i++){

        // Skip White spaces
        if(root->children[i]->character == ' '){
            continue;
        }

        // Go through the list number 2 for perms
        for ( int j = i + 1; j < root->num_child; j++){

            // Skip White spaces
            if(root->children[j]->character == ' '){
                continue;
            }

            // concatenate ta and tb to get strings
            char *buffer_ta = calloc(1, sizeof(char));
            concatenate(root->children[i], &buffer_ta);
            char *buffer_tb = calloc(1, sizeof(char));
            concatenate(root->children[j], &buffer_tb);

            // if both t_a and t_b concatneate to the same string
            // Merge (change labels ) if they are non-terminals or non-white space.
            if(strcmp(buffer_ta,buffer_tb) == 0){
                merge_same_node(root->children[i], root->children[j], i, j, root);
                //printf("Comp: %s, %d\n", buffer_ta, root->children[i]->t_label);
                free(buffer_ta);
                free(buffer_tb);
                continue;
            }

            free(buffer_ta);
            free(buffer_tb);


        }

    }

    for( int i = 0; i < root->num_child; i++){

        // Skip White spaces
        if(root->children[i]->character == ' '){
            continue;
        }

        // Go through the list number 2 for perms
        for ( int j = i + 1; j < root->num_child; j++){

            // Skip White spaces
            if(root->children[j]->character == ' '){
                continue;
            }

            // If 2 nodes have the same non-negative label
            // means the node has already been merged. Skip.
            if((root->children[i]->t_label == root->children[j]->t_label) && (root->children[i]->t_label > -1)){
                continue;
            }
            // t_a and t_b concatenate to different strings.
            // Now perform a more extensive checl
            int *res = calloc(1, sizeof(int));
            *res = 1;
            for( int i = 0; i < all_trees->count; i++){

                Node *dup_tree = duplicate_tree(all_trees->rootNodes[i]);
                rigorous_replacement_check(root->children[i], root->children[j], dup_tree, 0, res);
            }

            if( *res ){
                for( int i = 0; i < all_trees->count; i++){

                    Node *dup_tree = duplicate_tree(all_trees->rootNodes[i]);
                    rigorous_replacement_check(root->children[i], root->children[j], dup_tree, 0, res);
                }
            }

            if(*res){
                merge(root->children[i], root->children[j], root, i, j);
            }

            free(res);


        }

    }

}
```

**Figure 6.6:** Merge All valid function code

```
    void concatenate(Node *root, char** buffer){

        if(root->t_label == -1){
            size_t len = strlen(*buffer);
            char *new_space = realloc(*buffer, (len + 2) * sizeof(char));
            new_space[len] = root->character;
            new_space[len + 1] = '\0';
            *buffer = new_space;
            return;
        }

        for( int i = 0; i < root->num_child; i++){
            concatenate(root->children[i], buffer);

        }
    }
```

**Figure 6.7:** String concatenate function

```
void merge_same_node(Node *ta, Node *tb, int i , int j, Node *root){

    // Check the labels have already been merge
    // cases where theere are 3 of the same kind
    if((ta->t_label == tb->t_label) && (ta->t_label > 0)){
        return;
    }

    // if we are trying to merge 2 leaf nodes,
    // This is assuming pre-tokenisation.
    if(ta->t_label == -1 || tb->t_label == -1){

        // Create and link the nodes.
        // Only create the Link node if a link
        // intermediate node does not exist yet.
        // This accounts for when there are multiple of the same
        // leaf node.
        if(root->children[i]->t_label == -1){

            Node *ta_label = build_basic_node_with_list();
            (*tid)++;
            ta_label ->t_label = *tid;
            ta_label->parent = ta->parent;
            ta_label->children[0] = ta;
            (ta_label->num_child) ++;
            root->children[i] = ta_label;
            ta->parent = ta_label;

        }
        Node *tb_label = build_basic_node_with_list();
        tb_label ->t_label = *tid;
        tb_label->parent = tb->parent;
        tb_label->children[0] = tb;
        (tb_label->num_child) ++;
        root->children[j] = tb_label;
        tb->parent = tb_label;
        return;
    }

    // if 2 non-terminals are given
    // converge by changing the both t-label to
    // min (ta->t_label, tb->t__label)
    if(ta->t_label < tb->t_label){
        tb->t_label = ta->t_label;
    } else {
        ta->t_label = tb->t_label;
    }
    return;
}
```

**Figure 6.8:** Merging same nodes

```
    void merge(Node *node_1, Node *node_2, Node *root, int i, int j){


    // Add an intermediate label node
    // if either of the nodes are leaf nodes.
    if(node_1->t_label == -1){
        Node *inter_node= build_basic_node_with_list();
        (*tid)++;
        inter_node->t_label = *tid;
        inter_node->parent = node_1->parent;
        inter_node->children[0] = node_1;
        (inter_node->num_child) ++;
        root->children[i] = inter_node;
        node_1->parent = inter_node;
        node_1 = node_1->parent;
    }

    if(node_2->t_label == - 1){
        Node *inter_node= build_basic_node_with_list();
        (*tid)++;
        inter_node->t_label = *tid;
        inter_node->parent = node_2->parent;
        inter_node->children[0] = node_2;
        (inter_node->num_child) ++;
        root->children[j] = inter_node;
        node_2->parent = inter_node;
        node_2 = node_2->parent;
    }

    // If already intermediate nodes, switch labels.
    if(node_1->t_label < node_2->t_label){
        node_2->t_label = node_1->t_label;
    } else{
        node_1->t_label = node_2->t_label;
    }
}
```

**Figure 6.9:** General Merge functions

```c
void rigorous_replacement_check(Node *replacer, Node *replacee, Node *dup_tree, int pos, int *res){

    //if at any point res become 0, stop execution immediately
    if (!(*res)){
        return;
    }

    // Check if you are replacing single char so you can compare char
    // as terminal do not have a tid ( implementation diff )
    int terminal_replacee = 0;

    if (replacee->t_label == -1){
        terminal_replacee = 1;
    }

    // Flags to redue calls to the oracle
    // Extra oracles occur due to recursion.
    int forward = 1;

    // Looping through all the nodes in t0
    for ( int i = pos; i < dup_tree->num_child;  i++){

        Node *cur = dup_tree->children[i];

        // if it is a terminal replacee and no the correct 1
        // right now. Continue
        if (terminal_replacee){
            if (cur->character != replacee->character){
                continue;
            }
        }

        // perform the swap.
        dup_tree->children[i] = replacer;
        advanced_replacement_check(replacer, replacee, dup_tree, i + 1, res);
        if(forward){
            // call to oracle then
            // if (call to oracle) -> pass : *res = 0;
            char *buffer = calloc(1, sizeof(char));
            concatenate(dup_tree, &buffer);
            *res = parse_string(buffer);
            //printf("Printing buffer: %s and %d.\n",buffer, *res);
            free(buffer);
            forward = 0;

        }
        dup_tree->children[i] = cur;
        // case where 1 of the candidate string is invalid, so return.
        if(!*res){
            return;
        }
        concact_and_print(dup_tree);
        advanced_replacement_check(replacer, replacee, dup_tree, i + 1, res);

    }

    *res = 1;
}
```

**Figure 6.10:** Rigorous replacement check