Learning Highly Recursive Input Grammars

Neil Kulkarni[*]
University of California, Berkeley
neil.kulkarni@berkeley.edu

Caroline Lemieux[*]
University of California, Berkeley
clemieux@cs.berkeley.edu

Koushik Sen
University of California, Berkeley
ksen@cs.berkeley.edu

***Abstract*—This paper presents ARVADA, an algorithm for learning context-free grammars from a set of positive examples and a Boolean-valued oracle. ARVADA learns a context-free grammar by building parse trees from the positive examples. Starting from initially flat trees, ARVADA builds structure to these trees with a key operation: it *bubbles* sequences of sibling nodes in the trees into a new node, adding a layer of indirection to the tree. Bubbling operations enable recursive generalization in the learned grammar. We evaluate ARVADA against GLADE and find it achieves on average increases of 4.98× in recall and 3.13× in F1 score, while incurring only a 1.27× slowdown and requiring only 0.87× as many calls to the oracle. ARVADA has a particularly marked improvement over GLADE on grammars with highly recursive structure, like those of programming languages.**

I. INTRODUCTION

Learning a high-level language description from a set of examples in that language is a long-studied and difficult problem. While early interest in this problem was motivated by the desire to automatically learn human languages from examples, more recently the problem has been of interest in the context of learning program input languages. Learning a language of program inputs has several relevant applications, including generation of randomized test inputs [1], [2], [3], as well as providing a high-level specification of inputs, which can aid both comprehension and debugging.

In this paper we focus on the problem of learning *context-free grammars* (CFGs) from a set of positive examples S and a Boolean-value oracle $\mathcal{O}$. This is a similar setting as GLADE [4]. Like GLADE, and unlike other recent related works [5], [6], [7], we assume the oracle is black-box: our technique can only see the Boolean return value of the oracle. We adopted the use of an oracle as we believe that in practice, an oracle—e.g. in the form of a parser—is easier to obtain than good, information-carrying negative examples.

In this paper, we describe a novel algorithm, ARVADA, for learning CFGs from example strings S and an oracle $\mathcal{O}$. At a high-level, ARVADA attempts to create the smallest CFG possible that accommodates all the examples. It uses two key operations—bubbling and merging—to generalize the language as much as possible, while not overgeneralizing beyond the language accepted by $\mathcal{O}$.

To create this context-free grammar, ARVADA repeatedly performs the bubbling and merging operations on tree representations of the input examples. This set of trees is initialized with one "flat" tree per input example, i.e. the tree with a single root node whose children are the characters of the input string. The *bubbling* operation takes sequences of sibling nodes in the trees and adds a layer of indirection by replacing the sequence with a new node. This new node has the bubbled sequence of sibling nodes as children.

Then ARVADA decides whether to accept or reject the proposed bubble by checking whether a relabeling of the new node enables sound generalization of the learned language. Essentially, labels of non-leaf nodes correspond to nonterminals in the learned grammar. Merging the labels of two distinct nodes in the trees adds new strings to the grammar's language: the strings derivable from subtrees with the same label can be swapped. We call this the *merge* operation since it merges the labels of two nodes in the tree. If a valid merge occurs, the structure introduced by the bubble is preserved. Thus, merges introduce recursion when a parent node is merged with one of its descendants. If the label of the new node added in the bubbling operation cannot merge with any existing node in the trees, the bubble is rejected. That is, the introduced indirection node is removed, and the bubbled sequence of sibling nodes is restored to its original parent. These operations are repeated until no remaining bubbled sequence enables a valid merge.

In this paper, we formalize this algorithm in ARVADA. We introduce heuristics in the ordering of bubble sequences minimize the number of bubbles ARVADA must check before find a successful relabeling. We implement ARVADA in 2.2k LoC in Python, and make it available as open-source. We compare ARVADA to GLADE [4], a state-of-the-art for grammar learning engine with blackbox oracles. We evaluate it on parsers for several grammars taken from the evaluation of GLADE, Reinam [5], Mimid [7], as well as a few new highly-recursive grammars. On average across these benchmarks, ARVADA achieves 4.98× higher recall and 3.13× higher F1 score over GLADE. ARVADA incurs on a slowdown of 1.27× over GLADE, while requiring 0.87× as many oracle calls. We believe this slowdown is reasonable, especially given the difference in implementation language—ARVADA is implemented in Python, while GLADE is implemented in Java. Our contributions are as follows:

- We introduce ARVADA, which learns grammars from inputs strings and oracle via bubble-and-merge operations.
- We distribute ARVADA's implementation as open source: https://github.com/neil-kulkarni/arvada.
- We evaluate ARVADA on a variety of benchmarks against the state-of-the-art method GLADE.

II. MOTIVATING EXAMPLE

ARVADA takes as input a set of example strings S and an oracle $\mathcal{O}$. The oracle returns `True` if its input string is valid and `False` otherwise. ARVADA's goal is to learn a grammar

[*] Equal contribution.

$$\boxed{\mathcal{G}_w}\quad
\begin{array}{rcl}
\textit{start} & \rightarrow & \textit{stmt} \\
\textit{stmt} & \rightarrow & \texttt{while\textvisiblespace}\ \textit{boolexpr}\ \texttt{\textvisiblespace do\textvisiblespace}\ \textit{stmt} \\
 & | & \texttt{if\textvisiblespace}\ \textit{boolexpr}\ \texttt{\textvisiblespace then\textvisiblespace}\ \textit{stmt}\ \texttt{\textvisiblespace else\textvisiblespace}\ \textit{stmt} \\
 & | & \texttt{L\textvisiblespace=\textvisiblespace}\ \textit{numexpr} \\
 & | & \textit{stmt}\ \texttt{\textvisiblespace;\textvisiblespace}\ \textit{stmt} \\
\textit{boolexpr} & \rightarrow & \texttt{\sim}\textit{boolexpr} \mid \textit{boolexpr}\ \texttt{\textvisiblespace\&\textvisiblespace}\ \textit{boolexpr} \\
 & | & \textit{numexpr}\ \texttt{\textvisiblespace==\textvisiblespace}\ \textit{numexpr} \mid \texttt{false} \mid \texttt{true} \\
\textit{numexpr} & \rightarrow & \texttt{(}\ \textit{numexpr}\,\texttt{+}\,\textit{numexpr}\ \texttt{)} \mid \texttt{L} \mid \texttt{n}
\end{array}$$

$$S = \{\text{“}\texttt{while true \& false do L = n}\text{”},\ \text{“}\texttt{L = n ; L = (n+n)}\text{”}\} \qquad \mathcal{O}(i) = \begin{cases} \texttt{True if } i \in \mathcal{L}(\mathcal{G}_w) \\ \texttt{False otherwise} \end{cases}$$

Fig. 1: Example inputs S, and oracle $\mathcal{O}$ which returns `true` if its input is in the language of the while grammar $\mathcal{G}_w$.

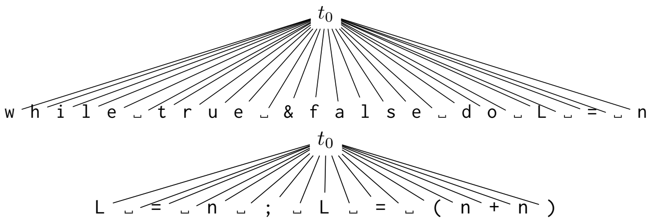

t_0
w h i l e ␣ t r u e ␣ & f a l s e ␣ d o ␣ L ␣ = ␣ n
t_0
L ␣ = ␣ n ␣ ; ␣ L ␣ = ␣ (n + n)

Fig. 2: Initial set of parse trees $\mathcal{T}$ created by ARVADA when run on S, $\mathcal{O}$ in Fig. 1. Each terminal c has a nonterminal parent t_c with rule $t_c \rightarrow c$, omitted for simplicity.

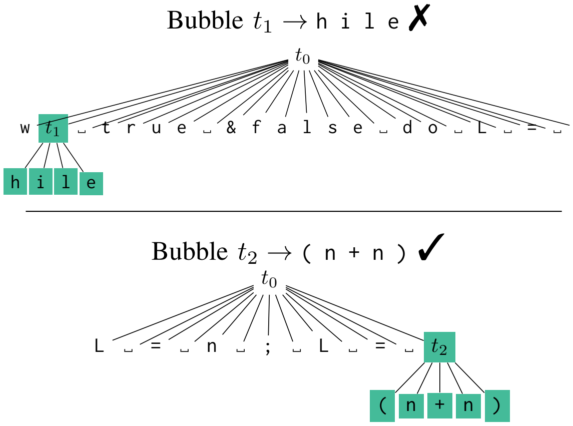

Bubble $t_1 \rightarrow$ h i l e ✗
t_0
w t_1 ␣ t r u e ␣ & f a l s e ␣ d o ␣ L ␣ = ␣ n
h i l e
Bubble $t_2 \rightarrow$ (n + n) ✓
t_0
L ␣ = ␣ n ␣ ; ␣ L ␣ = ␣ t_2
(n + n)

Fig. 3: Two possible bubbles applied to the trees in Fig. 2.

$\mathcal{G}$ which maximally generalizes the example strings S in a manner *consistent* with the oracle $\mathcal{O}$. That is, strings $i \in \mathcal{L}(\mathcal{G})$ in the language of the learned grammar should with high probability be accepted by the oracle: $\mathcal{O}(i) = \texttt{True}$. We formally describe maximal generalization in Section III.

Fundamentally, ARVADA learns a grammar by learning “parse trees” for the examples in S. These parse trees are initialized with flat trees for each example in S. Then, ARVADA adds structure, turning sequences of sibling nodes into new subtrees. The particular subtrees ARVADA keeps are those which enable generalization in the induced grammar.

From any set of trees $\mathcal{T}$ we can derive an ***induced grammar***. In particular, each non-leaf node in a tree $t \in \mathcal{T}$ with label t_{parent} and children with labels $t_{child_1}, t_{child_2}, \ldots, t_{child_n}$ ***induces the rule*** $t_{parent} \rightarrow t_{child_1} t_{child_2} \cdots t_{child_n}$. The *induced grammar* of $\mathcal{T}$ is then the set of induced rules for all nodes in the trees.

For example, the trees in Fig. 2 induce the grammar:

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

and the trees under (4) in Fig. 4 induce the grammar in Fig. 5.

Because of this mapping from trees to grammars, we will use the term “nonterminal” interchangeably with “label of a non-leaf node” when discussing relabeling trees.

A. Walkthrough

We illustrate ARVADA on a concrete example. We take the set of examples S and oracle $\mathcal{O}$ shown in Fig. 1. This oracle $\mathcal{O}$ accepts inputs as valid only if they are in the language of the while grammar $\mathcal{G}_w$, shown at the top of the figure. ARVADA treats $\mathcal{O}$ as blackbox, that is, it has no structural knowledge of $\mathcal{G}_w$: $\mathcal{G}_w$ is shown only to clarify the behavior of $\mathcal{O}$.

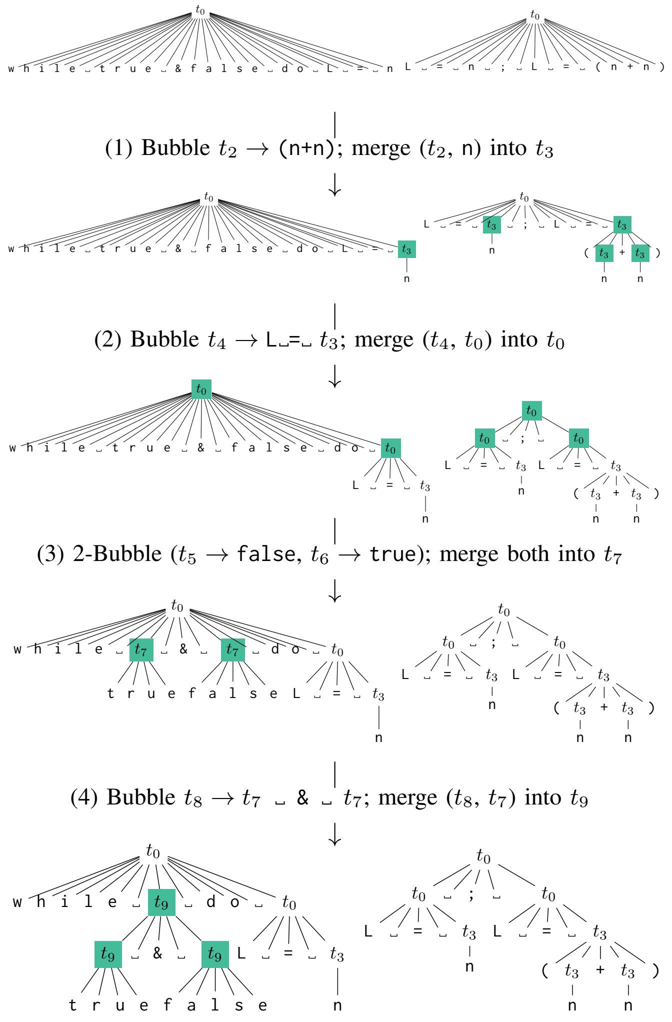
ARVADA begins by constructing naïve, flat, parse trees from the examples. These are shown in Fig. 2. Essentially, these trees simply go from the start nonterminal t_0 to the sequence of characters in each example $s \in S$. Let $\mathcal{T}$ designate the set of trees ARVADA maintains at any point in its algorithm.

1) Bubbling: The fundamental operation ARVADA performs is to *bubble up* a sequence of sibling nodes in the current trees $\mathcal{T}$ into a new nonterminal. To bubble a sequence s_1 in the trees $\mathcal{T}$, we create a new nonterminal node t_{s_1} with children s_1. Then we replace all occurrences of s_1 in each $t \in \mathcal{T}$ with t_{s_1}. Fig. 3 shows two such bubbles applied to the trees in Fig. 2. On top, we have bubbled the sequence `hile` into t_1; the second tree, unchanged, is not illustrated. On the bottom, we have bubbled `(n+n)` into t_2; the first tree is unchanged.

2) Merging: After bubbling a sequence s_1, ARVADA either *accepts* or *rejects* the bubble. ARVADA only accepts a bubble if it enables valid generalization of the examples. That is, if a relabeling of the bubbled nonterminal—merging its label with the label of another existing node—expands the language accepted by the induced grammar, while maintaining the oracle-validity of the strings produced by the induced grammar.

Consider again Fig. 3. On top, we have the bubble $t_1 \rightarrow$ `hile`. There is no terminal or nonterminal whose label can be merged with the label t_1 and retain a valid grammar: it can’t be merged with t_0, since “`hile`” on its own is not accepted by $\mathcal{O}$. Nor can it be merged with the label of any individual character: as just one example, merging with `L` would cause the $\mathcal{O}$-invalid generalization “`hile = n ; hile = (n+n)`”.

On the bottom of Fig 3, we have the bubble $t_2 \rightarrow$ `(n+n)`. We can in fact merge the label t_2 with the label t_n, the implicit nonterminal expanding to `n`. Notice that if we replace `n` with the strings derivable from t_2, we get examples like `while true & false do L = (n+n)` and `L = (n+n) ; L = ((n+n)+(n+n))`, which are all valid. Conversely, if we replace occurrences of t_2 with `n`, we get examples like `L = n ; L = n`. We accept this bubble, which expands the language accepted by



(1) Bubble $t_2 \rightarrow$ (n+n); merge (t_2, n) into t_3

(2) Bubble $t_4 \rightarrow$ L␣=␣ t_3; merge (t_4, t_0) into t_0

(3) 2-Bubble ($t_5 \rightarrow$ false, $t_6 \rightarrow$ true); merge both into t_7

(4) Bubble $t_8 \rightarrow t_7$ ␣ & ␣ t_7; merge (t_8, t_7) into t_9

Fig. 4: The state of trees $\mathcal{T}$ and the accepted bubbles of a full run of Arvada on S, $\mathcal{O}$ in Fig. 1.

the induced grammar. Thus, t_2 and n are merged and relabeled as t_3. The trees after the relabel are shown after (1) in Fig. 4. Note this merge has introduced recursive generalization; the induced grammar now includes the rules:

$t_0 \rightarrow$ L␣=␣t_3 $\quad$ $t_0 \rightarrow$ L␣=␣t_3 ␣;␣L␣=␣t_3 $\quad$ $t_3 \rightarrow$ (t_3+t_3) $\quad$ $t_3 \rightarrow$ n

In practice, Arvada checks whether labels t_a, t_b can be merged by checking *candidate strings* against the oracle. If the oracle accepts all these candidate strings, the relabeling is valid and the labels are merged. To create these candidates, Arvada creates mutated trees from the trees in $\mathcal{T}$ where (1) subtrees rooted at t_a are replaced subtrees rooted at t_b, and (2) subtrees rooted at t_b are replaced subtrees rooted at t_a. The candidate strings are then the ones derived from these trees, i.e. the ordered sequence of a tree's leaf nodes. Section III-C describes the conditions under which a bubble is accepted in more detail. Section III-D describes how to create these candidate strings, and the soundness issues this introduces.

3) Double bubbling: After accepting a bubble, Arvada continues to try and create new bubbles. It bubbles different sequences of children in the current trees $\mathcal{T}$, checking if they are accepted, and updating $\mathcal{T}$ accordingly. Fig. 4 shows a potential run of Arvada, with the state of the trees $\mathcal{T}$ as they are updated by bubbles and label merging.

$$t_0 \rightarrow \texttt{while␣}\ t_9\ \texttt{␣do␣}\ t_0 \mid \texttt{L␣=␣}\ t_3 \mid t_0\ \texttt{␣;␣}\ t_0$$
$$t_9 \rightarrow t_9\ \texttt{␣\&␣}\ t_9 \mid \texttt{true} \mid \texttt{false}$$
$$t_3 \rightarrow \texttt{(}\ t_3 \texttt{+} t_3 \texttt{)} \mid \texttt{n}$$

Fig. 5: Grammar produced by the run of Arvada in Fig. 4.

In Fig. 4, after (1) accepting the bubble $t_2 \rightarrow$ (n+n), Arvada (2) finds and accepts the bubble $t_4 \rightarrow$ L␣=␣ t_3, whose label can be merged with the start nonterminal t_0. At this point, Arvada will find no more bubbles which can be merged with any existing nodes in $\mathcal{T}$. For example, if Arvada creates the bubble $t_5 \rightarrow$ true, it will find that the label t_5 cannot be merged with the label of any existing node and reject it.

To cope with this, Arvada also considers 2-bubbles. In a 2-bubble, two distinct sequences of children—say, s_1 and s_2—in the trees are bubbled at the same time, i.e. replacing both s_1 with $t_{s_1} \rightarrow s_1$ and some other s_2 with $t_{s_2} \rightarrow s_2$. The two sequences can be totally distinct, or sub/super sets, but not overlapping: ($s_1 =$ true, $s_2 =$ false) is ok, as is ($s_1 =$ true, $s_2 =$ true & false), but ($s_1 =$rue␣&␣f, $s_2 =$e␣&␣fal) is not. Arvada accepts a 2-bubble only if the labels t_{s_1} and t_{s_2} can be merged *with each other*, not with another existing node. Otherwise, either t_{s_1} or t_{s_2} could be accepted as a 1-bubble.

4) Termination: In the run in Fig. 4, (3) Arvada applies and accepts the 2-bubble ($s_1 =$ true, $s_2 =$ false) and merges these sequences into t_7. This 2-bubble enables one final single bubble to be applied and accepted: (4) $t_8 \rightarrow t_7$ ␣&␣t_7 can be merged with t_7. After this, no more 1-bubbles or 2-bubbles can be accepted, so Arvada simply outputs the grammar induced by the final set of trees $\mathcal{T}$. Fig. 5 shows the grammar.

5) Effect of bubbling order: First, note that multiple orderings of bubbles can result in an equivalent grammar. For example, we could have applied ($s_1 =$ true, $s_2 =$ true␣&␣false) in (3), then bubbled up false alone in (4). Second, while Fig. 4 shows an ideal run, some accepted bubbles may impede further generalization of the grammar. For example, in the initial flat parse trees, $t_1 \rightarrow$ e␣&␣false can be merged with e. In the presence of the additional example "while n == n do skip", this merge prevents maximal generalization.

As such, the order in which bubbles are applied and checked has a large impact on Arvada's performance. In Section III-B, we describe heuristics that order the bubbles for exploration based on the context and frequency of the bubbled subsequence. These heuristics increase Arvada's probability of successfully finding the maximal generalization of S with respect to $\mathcal{O}$, as discussed in Section IV-B.

6) Maximality of learned grammar: The grammar in Fig. 5 is not identical to that in Fig. 1. However, it contains all the rules in $\mathcal{G}_w$ demonstrated by the examples S: t_3 has taken on the role of *numexpr*, t_9 in the role of *boolexpr*, and t_0 is effectively *stmt*. However, the rule *boolexpr* $\rightarrow$

numexpr ␣==␣ *numexpr* does not appear in Fig. 5. Fundamentally, this is because no substring derivable from this rule exists in S; as such, it is not part of S's maximal generalization.

III. Technique

We formally describe the high-level Arvada algorithm in Section III-A; Sections III-B, III-C, III-D, and III-E delve into the heuristic decisions made in Arvada's implementation.

First, we formalize our problem statement. Arvada accepts as input a set of example strings S and a Boolean-valued oracle $\mathcal{O}$ which judges the validity of the strings. Arvada's goal is to learn a context-free grammar $\mathcal{G}$ which *maximally generalizes* the set of example S in a manner *consistent* with $\mathcal{O}$.

Maximal generalization: Let S be a set of input strings and $\mathcal{O}$ be a Boolean-valued oracle accepting strings as input. Assume each $s \in S$ is accepted by the oracle, i.e., $\forall s \in S\colon \mathcal{O}(s) = \texttt{True}$. Let $\mathcal{G}_\mathcal{O}$ be a context-free grammar such that its language of strings $\mathcal{L}(\mathcal{G}_\mathcal{O})$ is equal to $\{i \in \Sigma^* \mid \mathcal{O}(i) = \texttt{True}\}$, the set of strings accepted by the oracle $\mathcal{O}$. Since $\mathcal{O}(s) = \texttt{True}$ for each $s \in S$, then each $s \in \mathcal{L}(\mathcal{G}_\mathcal{O})$. We call $\mathcal{G}_\mathcal{O}$ as the *target grammar*.

Thus, for each s, there exists a derivation $\mathcal{D}_s$ from the start symbol T_0 to s, i.e. $\mathcal{D}_s = T_0 \rightarrow \alpha_1\alpha_2 \cdots \alpha_n \rightarrow \cdots \rightarrow s$. This derivation is a sequence of nonterminal expansions according to some rules $\mathcal{G}_\mathcal{O}$. Let R_s be the set of rules in $\mathcal{G}_\mathcal{O}$ used in the derivation $\mathcal{D}_s$. Let $R_S = \cup_{s \in S} R_s$, and $\mathcal{G}^S_\mathcal{O}$ be the subset of $\mathcal{G}_\mathcal{O}$ which contains only those rules $r \in R_S$. Intuitively, $\mathcal{G}^S_\mathcal{O}$ is the sub-grammar of $\mathcal{G}_\mathcal{O}$ which is exercised by the $s \in S$.

Finally: a grammar which ***maximally generalizes*** S ***w.r.t.*** $\mathcal{O}$ is a grammar $\mathcal{G}$ such that $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}^S_\mathcal{O})$, i.e. it accepts the same language as $\mathcal{G}^S_\mathcal{O}$.

A. Main Algorithm

Algorithm 1 shows the main Arvada algorithm. It works as follows. First, Arvada builds naïve, flat, parse trees from the input strings (Line 1). Considering each $s_i \in S$ as a sequence of characters $s_i = c_i^1 c_i^2 \cdots c_i^{n_i}$, the tree constructed for s_i has a root node with the start symbol label t_0 and n_i children with labels $t_{c_i^1}, t_{c_i^2}, \ldots, t_{c_i^{n_i}}$. Each t_c has a single child whose label is the corresponding character c. Fig. 2 shows these flat parse trees for the examples strings $s \in S$ in Fig. 1, although the $t_c \rightarrow c$ are not illustrated for simplicity.

Arvada tries to generalize these parse trees by merging nodes in the tree into new nonterminal labels (Line 2). To merge two nodes t_a, t_b in a tree, we replace all occurrences of the labels t_a, t_b with a new label t_c. This creates new trees $\mathcal{T}'$; the merge is valid if the language of the induced grammar of $\mathcal{T}'$ only includes strings accepted by the oracle $\mathcal{O}$.

In practice, we check if a merge of t_a, t_b is valid by checking whether t_a can replace t_b in the example strings, and vice-versa. The strings derivable from an arbitrary nonterminal N in $\mathcal{T}$ are the concatenated leaves of the subtree rooted at N. We check whether t_a replaces t_b by checking whether the strings produced by replacing strings derivable from t_a by strings derivable from t_b, are accepted by the oracle. That is, we take the strings derivable from the trees $\mathcal{T}$, with holes in place of strings derived from t_b. Then we fill the holes with strings derivable by t_a. If all the strings are accepted by $\mathcal{O}$, Arvada judges the replacement as valid. Section III-D details this check and its soundness.

Algorithm 1 Arvada's high-level algorithm

```
Input: a set of examples $S$, an language oracle $\mathcal{O}$.
Output: a grammar $\mathcal{G}$ fitting the language.
 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: $\mathcal{G} \leftarrow$ InducedGrammar($bestTrees$)
15: return $\mathcal{G}$
```

Now the main Arvada loop starts. From the current S-derived trees $\mathcal{T}$, Arvada gets all potential "bubbles" for the trees (Algorithm 1, Line 6). For each tree $t \in \mathcal{T}$, GetBubbles collects all proper contiguous subsequences of children in t. That is, if the tree contains a node t_i with children $C = c_1, c_2, \ldots, c_n$, the potential bubbles include all subsequences of C of length greater than one and less than n. GetBubbles returns all these subsequences as 1-bubbles, and all non-conflicting pairs of these subsequences as 2-bubbles. Two subsequences are non-conflicting if they do not *strictly* overlap: they can be disjoint or one can be a proper subsequence of the other. So $((c_1, c_2, c_3), (c_2, c_3, c_4))$ conflict, but $((c_1, c_2, c_3), (c_2, c_3))$ and $((c_1, c_2, c_3), (c_4, c_5))$ do not. The order in which Arvada explores these bubbles is important for efficiency; we discuss this further in Section III-B.

Then, for each potential bubble, Arvada tries applying it to the existing set of trees $\mathcal{T}$. Suppose we have a 1-bubble consisting of the subsequence $c_i, c_{i+1}, \ldots, c_j$. To apply this bubble, we replace any sequence of siblings $t_{c_i}, t_{c_{i+1}}, \ldots, t_{c_j}$ with labels $c_i, c_{i+1}, \ldots, c_j$ in the tree with a new subtree $t_{new} \rightarrow t_{c_i}, t_{c_{i+1}}, \ldots, t_{c_j}$. Fig. 3 shows two such bubblings: `hile` is bubbled into the nonterminal t_1 at the top, and `(n+n)` is bubbled to t_2 on the bottom. If the bubbled nodes have structure under them, that structure is maintained: e.g., the bubbling of t_7␣&␣t_7 into t_9 at (4) in Fig. 4. For a 2-bubble, the same process is repeated for the two subsequences involved.

After applying the bubble, Arvada checks whether it should be accepted (Line 9). Section III-C formalizes CheckBubble, but essentially, CheckBubble accepts a bubble if the new nonterminals introduced in its application can be validly merged with some other nonterminal node in the tree.

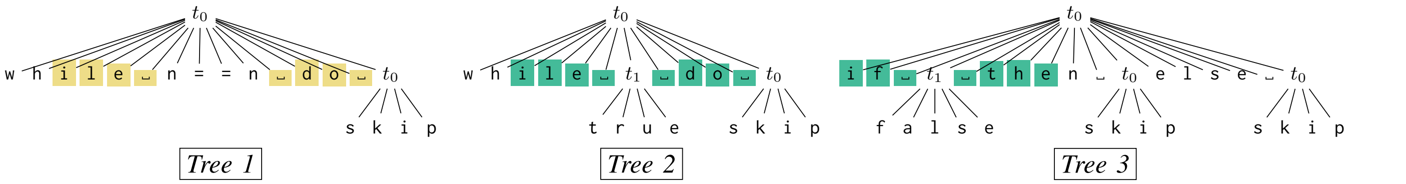

t_0
w h i l e ␣ n = = n ␣ d o ␣ t_0
s k i p
Tree 1
t_0
w h i l e ␣ t_1 ␣ d o ␣ t_0
t r u e
s k i p
Tree 2
t_0
i f ␣ t_1 ␣ t h e n ␣ t_0 e l s e ␣ t_0
f a l s e
s k i p
s k i p
Tree 3

Fig. 6: Partial parse tree $\mathcal{T}$ during run of ARVADA on while, with guide examples "while n==n do skip", "if false then skip else skip" and "while true do skip". ARVADA has applied the 1-bubble "skip", which merged with t_0, and the 2-bubble ("false", "true"). The 4-contexts for "n == n"' are highlighted in yellow, and for t_1 are highlighted in green.

If the new bubbled nonterminal allows a valid merge with some other nonterminal, CHECKBUBBLE returns True as well as the trees with the merge applied (Line 9). We update the best trees $\mathcal{T}$ to reflect the successful merge (Line 11), and GETBUBBLES is called again on the new $\mathcal{T}$. If the bubble is not accepted, ARVADA continues to check the next bubble returned by GETBUBBLES (Line 7).

The algorithm terminates when none of the bubbles are accepted, i.e. when the trees $\mathcal{T}$ cannot be further generalized, and returns the grammar $\mathcal{G}$ induced by the trees $\mathcal{T}$ (Line 15).

We can guarantee the following about ARVADA as long as merges are sound, once we consider the notion of *partially merging* two nonterminals, discussed in Section III-C2.

EXISTENCE THEOREM: There exists a sequence of k-bubbles, that, when considered by ARVADA in order, enable ARVADA to return a grammar $\mathcal{G}$ s.t. $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}_{\mathcal{O}})$, so long as the input examples S are exercise all rules of $\mathcal{G}$.

Proof Outline: The optimal bubble order always chooses the right-hand-side of some $N \rightarrow \alpha_1 \cdots \alpha_n$ in $\mathcal{G}$ as the sequence to bubble, either as 1-bubble if there exists an expansion for N in the trees already, or as a 2-bubble otherwise.

Our technical report gives a formal treatment of this and the **Generalization Theorem**, which shows that k-bubbles monotonically increase the language of the learned grammar [8].

B. Ordering Bubbles for Exploration

As described in paragraph 5) of Section II and alluded to above, the order of bubbles impacts the eventual grammar returned by ARVADA. Unfortunately, the number of orderings of bubbles is exponential. To have an efficient algorithm in practice, we must make sure the algorithm finds the correct order of bubbles early in its exploration of bubble orders. As such, GETBUBBLES returns bubbles in an order more likely to enable sound generalization of the grammar being learned.

As described in the prior section, bubble sequences consist of proper contiguous subsequences of children in the current trees $\mathcal{T}$. We increase the maximum length of subsequences considered once all bubbles of shorter length do not enable any valid merges. These subsequences (and their pairs) form the base of 1-bubbles (and 2-bubbles) returned by GETBUBBLES.

Recall that a bubble should be accepted if the bubbled nonterminal(s) can be merged with an existing nonterminal (or each other). Thus, GETBUBBLES should first return those bubbles that are likely to be mergeable. We leverage the following observation to return bubbles likely to enable merges. Expansions of a given nonterminal often occur in a similar *context*. The k-context of a sequence of sibling terminals/nonterminals s in a tree is the tuple of k siblings to the left of s and k siblings to right of s.

Fig. 6 shows an example of a run of ARVADA on the while language, after the application of the 1-bubble "skip" and the 2-bubble ("false", "true"). The set of 4-contexts for the sequence "n␣==␣n" is $\{(($i, l, e, ␣$), ($␣, d, o, ␣$))\}$. Similarly, "t_1"'s 4-contexts are $\{(($$S$, i, f, ␣$), ($␣, t, h, e$)), (($i, l, e, ␣$), ($␣, d, o, ␣$))\}$; "$S$" is a dummy element indicating the start of the example string. Note that "n==n" and "t_1" share the 4-context $\{(($i, l, e, ␣$), ($␣, d, o,␣$))\}$

With this in mind, GETBUBBLES orders the bubbles in terms of their *context similarity*. Given two contexts $c_0 = (l_0, r_0)$ and $c_1 = (l_1, r_1)$, where $l_i = (l_i^k, l_i^{k-1}, \ldots, l_i^0)$ and $r_i = (r_i^0, \ldots, r_i^{k-1}, r_i^k)$, we have $\mathit{contextSim}(c_0, c_1) = \mathit{kTupleSim}(l_0, l_1) + \mathit{kTupleSim}(r_0, r_1)$, where

$$\mathit{kTupleSim}(t_0, t_1) = \begin{cases} \frac{1}{2} \text{ if } t_0 = t_1 \\ \sum_{i=0}^{k} \frac{\mathbb{1}_{=}(t_0^i, t_1^i)}{2^{i+2}} \end{cases}$$

where $\mathbb{1}_{=}$ is the indicator function, returning 1 if its arguments are equal and 0 otherwise. This similarity function gives most weight to the context elements closest to the bubble.

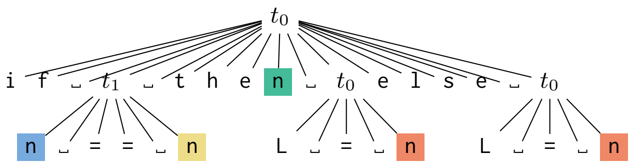
With this in mind, we define set context similarity as the maximum similarity of two contexts within the set:

$$\mathit{setContextSim}(C_0, C_1) = \max_{c_0 \in C_0, c_1 \in C_1} \mathit{contextSim}(c_0, c_1).$$

In our running example, the context similarity is 1 because n==n's 4-context set is a subset of t_1's 4-context set.

To form bubbles, GETBUBBLES first traverses all the trees $\mathcal{T}$ currently maintained by ARVADA. It considers each proper contiguous subsequence of siblings in the trees. For each subsequence s, it collects the k-contexts for s, as well as the occurrence count of the subsequence $\mathit{occ}(s)$. In Fig. 6, $\mathit{occ}($while$) = 2$, $\mathit{occ}(t_1) = 2$ and $\mathit{occ}($n␣==␣n$) = 1$. In our implementation we take $k = 4$.

ARVADA then creates a 2-bubble for each pair of sequences (s_1, s_2) where both $|s_1| > 1$ and $|s_2| > 1$. The similarity score of this 2-bubble is $\mathit{setContextSim}(\mathit{contexts}(s_1), \mathit{contexts}(s_2))$ and its frequency score is the average frequency of the two sequences in the bubble $\frac{\mathit{occ}(s_1)+\mathit{occ}(s_2)}{2}$. Additionally, for each sequence s_0 with $|s_0| > 1$, ARVADA creates a 1-bubble (s_0). Let S_1 be the set of length-one subsequences. The similarity score of (s_0) is $\max_{s_1 \in S_1} \mathit{setContextSim}(\mathit{contexts}(s_0), \mathit{contexts}(s_1))$ and its frequency score is $\mathit{occ}(s_0)$.



$$\begin{aligned} t_0 &\rightarrow \texttt{if␣}t_1\texttt{␣the}\ t_{\texttt{n}}\texttt{␣}t_0\texttt{␣else} \\ t_1 &\rightarrow t_{\texttt{n}}\texttt{␣==␣}t_{\texttt{n}} \\ t_0 &\rightarrow \texttt{L␣=␣}t_{\texttt{n}} \\ t_{\texttt{n}} &\rightarrow \texttt{n} \end{aligned} \qquad (1)$$

$$\begin{aligned} t_0 &\rightarrow \texttt{if␣}t_1\texttt{␣the}\ t_{n_1}\texttt{␣}t_0\texttt{␣else} \\ t_1 &\rightarrow t_{n_2}\texttt{␣==␣}t_{n_3} \\ t_0 &\rightarrow \texttt{L␣=␣}t_{n_4} \\ t_{n_1} \rightarrow \texttt{n} \quad t_{n_2} &\rightarrow \texttt{n} \quad t_{n_3} \rightarrow \texttt{n} \quad t_{n_4} \rightarrow \texttt{n} \end{aligned} \qquad (2)$$

Fig. 7: Example tree and rules in its induced grammar which have $t_{\texttt{n}}$ in their expansion (1), and the same grammar with $t_{\texttt{n}}$ split at different positions. For simplicity, nonterminals of the form $t_c \rightarrow c$—other than $t_{\texttt{n}}$ in (1)—are collapsed to c.

Finally, GETBUBBLES takes the top-n bubbles as sorted primarily by similarity, and secondarily by frequency. Intuitively, high-frequency sequences may correspond to tokens in the oracle's language. The order of bubbles is shuffled to prevent all runs of ARVADA from getting struck in the same manner. We find $n = 100$ to be effective in practice.

C. Accepting Bubbles

The second key component of ARVADA is deciding whether a given bubble should be accepted: this section formalizes how CHECKBUBBLE works. At the core of CHECKBUBBLE is the concept of whether two labels t_a, t_b can be merged. We say that t_a and t_b can be merged, i.e. MERGES(t_a, t_b), if and only if REPLACES(t_a, t_b)—that is, all occurrences of t_b can be replaced by t_a in the grammar—and REPLACES(t_b, t_a). We formalize how REPLACES is checked in the next section.

1) 2-Bubbles: ARVADA accepts a 2-bubble (s_1, s_2) with labels t_{s_1}, t_{s_2} only if MERGES(t_{s_1}, t_{s_2}). Intuitively, this is because both bubbles should be kept only if they together expand the grammar. For example, suppose we apply the 2-bubble ("n␣==␣n", "lse") to the trees in Fig. 6, resulting in nonterminals $t_{n==n} \rightarrow$ n␣==␣n and $t_{lse} \rightarrow$ lse. While $t_{n==n}$ can merge with t_1, t_{lse} does not contribute to this merging. So, ("n␣==␣n") should be accepted only as a 1-bubble.

2) 1-Bubbles: Recall that ARVADA scores 1-bubbles highly if they are likely to merge with an existing nonterminal. Let $NTs(\mathcal{T})$ be the nonterminal labels present in the current set of trees $\mathcal{T}$. Given a 1-bubble (s_1) with label t_{s_1}, we go through each $t_i \in NTs(\mathcal{T})$ and check whether MERGES(t_i, t_{s_1}). If MERGES(t_i, t_{s_1}) is true for some $t_i \in NTs(\mathcal{T})$, then CHECKBUBBLE accepts the bubble (s_1).

However, if t_{s_1} cannot merge with any $t_i \in NTs(\mathcal{T})$, ARVADA also looks for *partial merges*. Partial merging works as follows. Let $CNTs(\mathcal{T})$ be the *character nonterminal* labels present in the current set of trees $\mathcal{T}$. A *character nonterminal* is a nonterminal whose expansions only of a single terminal element, e.g., $t_{\texttt{n}} \rightarrow \texttt{n}$ or $t_1 \rightarrow \texttt{1} \mid \texttt{2} \mid \texttt{3}$.

For each $t_c \in CNTs(\mathcal{T})$, the partial merging algorithm identifies all the different occurrences of t_c in the right-hand-side of expansions in $\mathcal{T}$'s induced grammar. For instance, in the grammar fragment (1) of Fig. 7, we see the nonterminal $t_{\texttt{n}}$, corresponding to "n", occurs 4 distinct times in right-hand-sides of expansions. The partial merging algorithm then modifies the grammar so that the i^{th} occurrence of t_c is replaced with a fresh nonterminal t_{c_i}. Each t_{c_i} expands to the same bodies as t_c; i.e. $t_{c_i} \rightarrow c$. This replacement process is illustrated in the grammar fragment (2) of Fig. 7: the four occurrences of $t_{\texttt{n}}$ have been replaced with t_{n_1}, t_{n_2}, t_{n_3}, and t_{n_4}. Finally, we get to the *merging* in *partial merging*: for each t_{c_i}, the algorithm checks if MERGES(t_{c_i}, t_{s_1}). If MERGES(t_{c_i}, t_{s_1}) for any t_{c_i}, ARVADA accepts the bubble (s_1), and t_{s_1} is merged with all such t_{c_i}. The t_{c_j} which cannot be merged with t_{s_1} are restored to the original nonterminal t_c.

The term partial merge refers to the fact that we have effectively merged t_{s_1} with *some* of the occurrences of t_c in rule expansions. This step is useful when ARVADA's initial trees—which map each character to a single nonterminal—use the same nonterminal for characters that are conceptually separate. For instance, consider the 1-bubble ((n+n)), with label $t_{(n+n)}$. Given the tree in Fig. 7, MERGES$(t_{\texttt{n}}, t_{(n+n)})$ fails because "(n+n)" cannot replace the "n" in "then". In fact, $t_{(n+n)}$ cannot merge with any $t_i \in NTs(\mathcal{T})$ initially. But the partial merge process splits $t_{\texttt{n}}$ into t_{n_1}, t_{n_2}, t_{n_3}, t_{n_4}, and ARVADA finds that $t_{(n+n)}$ in fact merges with t_{n_2}, t_{n_3} and t_{n_4}. So, it is merged with those nonterminals and accepted.

Note: though we consider only partial merges on character nonterminals for efficiency reasons, the concept of partial merging can be applied to any pair of nonterminals.

In summary, a 1-bubble (s_1) with label t_{s_1} is accepted if either: (1) for some $t_i \in NTs(\mathcal{T})$, MERGES(t_i, t_{s_1}), or (2) for some $t_c \in CNTs(\mathcal{T})$, t_{s_1} can be partially merged with t_c.

D. Sampling Strings for Replacement Checks

The final important element affecting the performance of ARVADA is how exactly we determine whether the merge of two nonterminals labels is valid. Recall that MERGES(t_a, t_b) if and only if REPLACES(t_a, t_b) and REPLACES(t_b, t_a).

We implement REPLACES$(t_{\text{replacer}}, t_{\text{replacee}})$ as follows. From the current parse trees, we derive the *replacee strings*: the strings derivable from the parse trees in *trees*, but with holes instead of the strings derived from t_{replacee}. Then, we derive a set of *replacer strings*: the strings derivable from t_{replacer} in the trees. Finally, we create the set of *candidate strings* by replacing the holes in the replacee strings with the replacer strings. If $\mathcal{O}$ rejects any candidate string, the merge is rejected, and REPLACES returns false.

Fig. 8 shows how replacer and replacee strings are computed in the call to REPLACES(t_0, t_4), i.e. whether t_0 can replace t_4. Replacee strings for a node in the parse tree are computed by

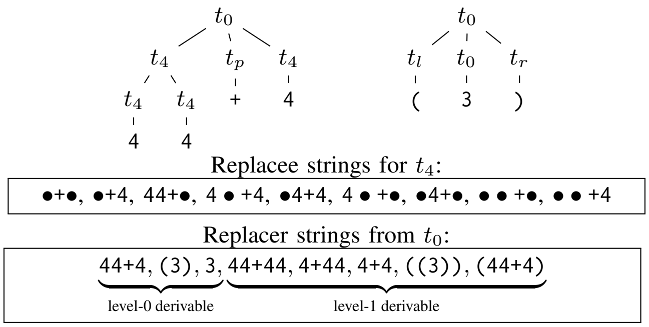


Fig. 8: Two partial parse trees and examples of replacee and replacer strings. The symbol • designates holes which will be replaced by (level-n derivable) replacer strings.

taking the product of replacee strings for all its children; the nonterminal being replaced becomes a hole.

Level-0 replacer strings for t_i are just the strings that directly derivable from t_i in the tree; in Fig. 8, the level-0 derivable strings of t_0 are `44+4`, `(3)`, `3`, and the level-0 derivable strings of t_4 are `44`, `4`. Then, the set of level-n derivable strings for a node is the set derived from taking the product of all level-($n-1$) derivable strings for each child of a node. The level-1 replacer strings for t_0 are shown in Fig. 8.

When REPLACES is run in the full MERGEALLVALID call or while evaluating a 1-bubble, we use only level-0 replacer strings. However, we found that level-1 replacer strings greatly increased soundness at a low runtime cost for 2-bubbles. Intuitively this is because nonterminals from new bubbles tend to have less structure underneath them than existing nonterminals in the trees. So it is faster to compute level-1 replacer strings for these new bubble-induced nonterminals.

Note that the both the number of replacee strings and of level-n derivable replacer strings grows exponentially. So, instead of taking the entire set of strings derivable in this manner, if there are more than p of them, we uniformly sample p of them. In our implementation we use $p = 50$, to make the number of parse calls reasonable in terms of runtime.

Unfortunately, this process allows unsound merges, where all candidate strings are accepted by the oracle, but the merge adds oracle-invalid inputs to the language of the learned grammar. First, because only p candidates are sampled. Second, because the replacee strings are effectively "level 0", and thus, not reflective of the current induced grammar from the trees. Third, because a candidate string is produced by replacing all its holes with a single replacer string, rather than filling holes with different replacer strings. Taking $p \to \infty$, $n \to \infty$ for the level-n replacer strings, and filling different holes with different replacer strings would ensure sound merges.

E. Pre-tokenization

Since ARVADA considers 2-bubbles, it is effectively n^4 in the total length of examples n. So, to improve performance as n gets large and reduce the likelihood of creating "breaking" bubbles, in our implementation we use a simple heuristic to pre-tokenize the values at leaves, rather than considering each character as a leaf. We group together sequences of contiguous characters of the same class (lower-case, upper-case, whitespace, digits) into leaf tokens. Punctuation and non-ASCII characters are still treated as individual characters. We then run the ARVADA as described previously. To ensure generalization, we add a last stage which tries to expand these tokens into the entire character class: e.g. if $t_1 \rightarrow$ `abc|cde`, we check whether t_1 can be replaced by any sequence of lower-case letters, letters, or alphanumeric characters. We construct the replacee strings as described above, and sample 10 strings from the expanded character classes as replacer strings.

IV. EVALUATION

We seek to answer the following research questions:

RQ1. Do ARVADA's mined grammars generalize better (have higher recall) than state-of-the-art?
RQ2. Do ARVADA's mined grammars produce more valid inputs (have higher precision) than state-of-the-art?
RQ3. How does the nondeterminism in ARVADA cause its behavior to vary across different invocations?
RQ4. How does ARVADA's performance compare to that of deep-learning approaches?
RQ5. What are ARVADA's major performance bottlenecks?
RQ6. What do ARVADA's mined grammars look like?

A. Benchmarks

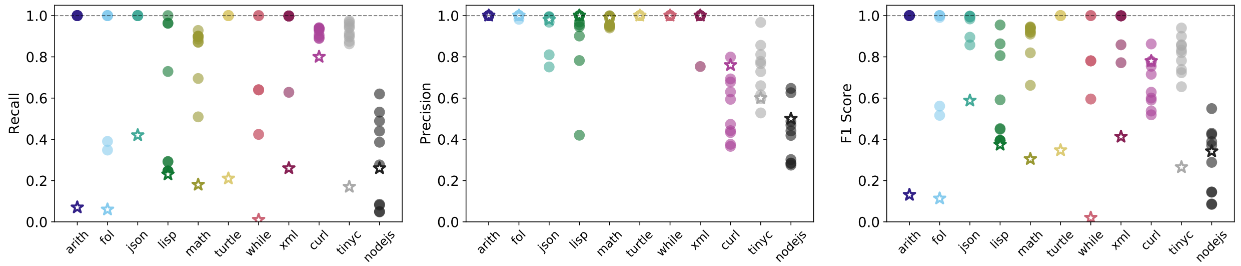
We evaluate ARVADA against state-of-the-art blackbox grammar inference tool GLADE [4] on 11 benchmarks.

The first 8 benchmarks consist of an ANTLR4 [10] parser for the ground-truth grammar as oracle and a randomly generated set of training examples S. S is sampled to cover all of the rules in the ground-truth grammar, while keeping the length of each example $s \in S$ small. The test set is randomly sampled from the ground-truth grammar. Essentially, this ensures that the maximal generalization of S covers the entire test set. Other than `turtle` and `while`, these benchmarks come from prior work [4], [5], [7]:

- **arith**: operations between integers, can be parenthesized
- **fol**: a representation of first order logic, including qualifiers, functions, and predicates
- **json**: JSON with objects, lists, strings with alpha-numeric characters, Booleans, `null`, integers, and floats
- **lisp**: generic s-expression language with "`.`" cons'ing
- **mathexpr**: binary operations and a set of function calls on integers, floats, constants, and variables
- **turtle**: LOGO-like DSL for Python's turtle
- **while**: simple while language as shown in Fig. 1
- **xml**: supporting arbitrary attributes, text, and a few labels

The next 3 benchmarks use as oracle a runnable program, and use a random input generator to create S and the test set. S consists of the first 25 oracle-valid inputs generated by the generator, and the test set of the next 1000 oracle-valid inputs generated. In this case, there is no guarantee that the maximal generalization of S covers the test set.

- **curl**: the oracle is the curl[11] url parser. We use the grammar in RFC 1738 [12] to generate S and test set.



(a) Recall. Higher is better.

(b) Precision. Higher is better.

(c) F1 Score. Higher is better.

Fig. 9: Recall, precision, and F1 score for each of the 10 runs of ARVADA (plotted with ●) and GLADE (plotted with ☆).

TABLE I: Summary of results for ARVADA and GLADE. “R” is recall, “P” is precision. Results for ARVADA are listed as the means over 10 runs with ± the standard deviation. Bolded results are 2× better.

	ARVADA					GLADE				
Bench.	Recall	Precision	F1 Score	Time(s)	# Queries	R	P	F1	Time(s)	# Queries
arith	**1.00** ± 0.00	1.00 ± 0.00	**1.00** ± 0.00	3 ± 0	828 ± 37	0.07	1.00	0.13	12	2.3K
fol	**0.87** ± 0.25	1.00 ± 0.01	**0.91** ± 0.18	372 ± 36	33K± 3.7K	0.06	1.00	0.11	107	20K
json	**1.00** ± 0.00	0.95 ± 0.08	0.97 ± 0.05	76 ± 11	16K ± 1K	0.42	0.98	0.59	61	11K
lisp	**0.52** ± 0.33	0.90 ± 0.17	0.57 ± 0.21	16 ± 4	3.6K ± 603	0.23	1.00	0.38	20	3.8K
math.	**0.84** ± 0.12	0.97 ± 0.02	**0.89** ± 0.08	65 ± 6	11K ± 1.1K	0.18	0.99	0.31	103	19K
turtle	**1.00** ± 0.00	1.00 ± 0.00	**1.00** ± 0.00	84 ± 8	10K ± 1.1K	0.21	1.00	0.34	75	14K
while	**0.70** ± 0.21	1.00 ± 0.00	**0.81** ± 0.14	54 ± 5	13K ± 1.5K	0.01	1.00	0.02	50	9.1K
xml	**0.96** ± 0.11	0.98 ± 0.07	**0.96** ± 0.08	205 ± 34	14K ± 2.4K	0.26	1.00	0.42	81	15K
curl	0.92 ± 0.02	0.55 ± 0.14	0.68 ± 0.11	111 ± 12	25K ± 3.1K	0.80	0.76	0.78	112	30K
tinyc	**0.92** ± 0.04	0.73 ± 0.13	**0.81** ± 0.08	6.4K ± 1.2K	112K ± 32K	0.17	0.60	0.26	917	252K
nodejs	0.30 ± 0.21	0.42 ± 0.13	0.29 ± 0.16	46K ± 22K	142K ± 90K	0.26	0.50	0.34	38K	113K

TABLE II: Results for CLGen's core LSTM [9]. “Model Time” is the logged model training time.

	CLGen LSTM		
Bench.	Time(s)	Model Time(s)	Precision
arith	172	9	0.002
fol	177	12	0.460
json	178	11	0.625
lisp	173	9	0.367
mathexpr	176	12	0.393
turtle	176	10	0.367
while	167	9	0.012
xml	171	12	0.228
curl	176	12	0.434
tinyc	189	21	0.062
nodejs	176	18	0.111

- **tinyc**: the oracle is the parser for tinyc [13], a compiler for a subset of C. We use the same golden grammar as in Mimid [7] to generate S and the test set.
- **nodejs**: the oracle is an invocation of `nodejs --check`, which just checks syntax [14]. To generate S and the test set, we use Zest's [15] javascript generator.

The average length of training examples in the set S is below 20 for all benchmarks except `tinyc` (77) and `nodejs` (58). We adjust the maximum bubble length hyperparameter (ref. Section III-B) accordingly: the default is to range from 3 to 10, but on `tinyc` and `nodejs` we range from 6 to 20.

B. Accuracy Evaluation

First, we evaluate the accuracy of ARVADA and GLADE's mined grammars with respect to the ground-truth grammar We ran both ARVADA and GLADE with the same oracle example strings. Three key metrics are relevant here:

Recall: the proportion of inputs from the held-out test set—generated by sampling the golden grammar/generator—that are accepted by the mined grammar. We use a test set size of 1000 for all benchmarks.

Precision: the proportion of inputs sampled from the mined grammar that are accepted by the golden grammar/oracle. We sample 1000 inputs from the mined grammar to evaluate this.

F1 Score: the harmonic mean of precision and recall. It is trivial to achieve high recall but low precision (mined grammar captures any string) or low recall but high precision (mined grammar captures only the string in S); F1 measures the tradeoff between the two.

Results. As ARVADA is nondeterministic in the order of bubbles explored, we ran it 10 times per benchmark. As GLADE is deterministic, we ran it only once per benchmark.

Table I shows the overall averaged results, Fig 9 the individual runs. We see from the table that on average, ARVADA achieves higher recall than GLADE *on all benchmarks*, and it achieves higher F1 score on all but 2 benchmarks. ARVADA achieves *over* 2× higher recall on 9 benchmarks, and *over* 2× higher F1 score on 7 benchmarks.

Even for those benchmarks where ARVADA does not have a higher F1 score on average, Fig. 9c shows that ARVADA outperforms GLADE on some runs. For `nodejs`, on 5 runs, ARVADA achieves a higher F1 score, ranging from 0.37 to 0.55. For `curl`, on 2 runs ARVADA achieves F1 scores greater than or equal to GLADE's: 0.78 and 0.86. It makes sense that GLADE performs well for `curl`: the url language is regular, and the first phase of GLADE's algorithm works by building up a regular expressions. Nonetheless Fig. 9a shows that ARVADA achieves consistently higher recall on `curl`.

Overall, on average across all runs and benchmarks, ARVADA achieves **4.98× higher recall** than GLADE, while maintaining 0.96× its precision. So, on our benchmarks, the answer to RQ1 is in the affirmative, while the answer to RQ2 is not. Given that ARVADA still achieves a **3.13× higher F1 score** on average, and that higher generalization (in the form of recall) is much more useful if the mined grammar is used for fuzzing, we find this to be a very positive result.

However, we see from the standard deviations in Table I that ARVADA's performance varies widely on some benchmarks, notable fol, lisp, while, and fol. Fig. 9, which shows the raw data, confirms this. In Fig. 9a, we see that the performance on the lisp benchmark is quite bimodal. All of the mined grammars with recall around 0.25 fail to learn to cons parenthesized s-expressions. This may be because the minimal example set did not actually have an example of this nesting. On nodejs, the two runs with recall less than 0.1 find barely any recursive structures, suggesting that on larger example sets, Arvada may get lost in bubble order. Overall, the answer to RQ3 is that Arvada's nondeterministic bubble ordering can have very large impacts on the results. We discuss possible mitigations in Section V.

C. *Comparison to Deep Learning Approaches*

Recently there has been interest in using machine learning to learn input structures. For instance, Learn&Fuzz trains a seq-2-seq model to model the structure of PDF objects [16]; it uses information about the start and end of pdf objects as well as the importance of different characters in its sampling strategy. DeepSmith [17] trains an LSTM to model OpenCL kernels for compiler fuzzing, adding additional tokenization and pre-processing stages to CLGen [9].

A natural question is how Arvada compares to these generative models. We trained the LSTM model from CLGen [9], the generative model behind DeepSmith, on our benchmarks. We removed all the OpenCL-specific preprocessing stages from the pipeline. We used the parameters given as example in the CLGen repo, creating a 2-layer LSTM with hidden dimension 128, trained for 32 epochs. We used \n!!\n as an EOF separator. Each sample consisted of 100 characters, split into different inputs where the EOF separator appeared.

Table II shows the runtime of the model on each benchmark, as well as the precision achieved on the first 1000 samples taken from the model. Generally, we see that the precision is much lower than that of GLADE or Arvada. On arith, the model over-trains on the EOF separator, adding \n and ! throughout samples. Since the model is generative—it can generate samples but not provide a judgement of sample validity—, we cannot measure Recall as in Table I. However, qualitative analysis of the samples suggests there is not much learned recursive generalization. For json, 602 of the 625 valid samples are a single string (e.g., "F"); the other 21 valid samples are numbers, false, or []. For nodejs, of the 111 valid samples, 26 are empty, 24 are a single identifier (e.g. a_0), 18 are a parenthesized integer or identifier (e.g,. (242)), and 17 are a single-identifier throw, e.g. throw (a_0).

These results are not entirely unexpected, because the LSTM underlying CLGen is learning *solely from the input examples*. Both Arvada and GLADE extensively leverage the oracle, effectively creating new input examples from which to learn. This explains why the runtimes look so different between Tables I and II. We see in Table II that the total time to setup and train the model is around 3 minutes for all benchmarks, and the core training time is around 10-20 seconds. We see the model training time is slightly higher for tinyc and nodejs, which had longer input examples.

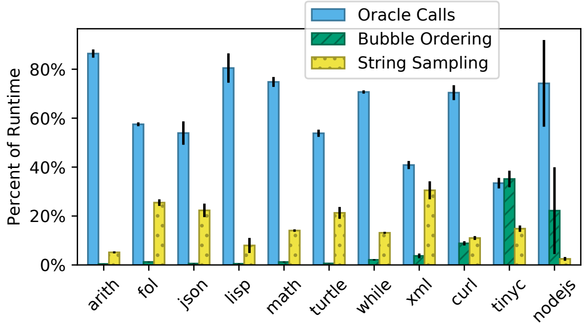


Fig. 10: Average percent of runtime spent in different components of Arvada. Error bars show std. deviation.

Overall, we expect these deep-learning approaches to be more well-suited to a case where an oracle is not available, but large amounts of sample inputs are. These models may also be more reliant input-format specific pre-processing steps, like those used on OpenCL kernels in CLGen and DeepSmith.

D. *Performance Analysis*

The next question is about Arvada's performance. Table I shows the average Arvada runtime and number of queries performed for each benchmark, and the same statistics for GLADE. On 7 of 11 benchmarks, Arvada is on average slower than GLADE; overall across benchmarks, this amounts to an average 1.27× slowdown. This is quite respectable, since Arvada has a natural runtime disadvantage due to being implemented in Python rather than Java. For the three benchmarks on which Arvada is over 2× slower than GLADE, it has huge increases in F1 score: 0.11 → 0.91 for fol, 0.42 → 0.96 for xml, and 0.26 → 0.81 for tinyc.

The story for oracle queries performed is inversed; Arvada requires more oracle queries on average on only 4 benchmarks. For all of these except nodejs, Arvada also had much higher F1 scores. However, nodejs is a benchmark with high variance. On the run with highest F1 score (0.55, higher than GLADE's 0.34), Arvada takes 86,051 s to run and makes 270k oracle calls. On the fastest run, where Arvada only gets F1 score 0.14, Arvada takes 17,775 s and makes 41k oracle calls. That is, the higher performance cost correlates with the slower runs on this benchmark: 5 of the 6 slower runs also have higher F1 scores.

Overall across all benchmarks, Arvada performs **only 0.87× as many oracle queries as GLADE**. This is encouraging as it gives more room for performance optimizations.

Fig. 10 breaks down the average percent of runtime spent in Arvada's 3 most costly components: calling the oracle; creating, scoring, and ordering bubbles; and sampling string for replacement checks. The error bars show standard deviation; note the aforementioned high variance for nodejs appears here too. On the minutes-long benchmarks on which Arvada is at least 10 seconds slower than GLADE, $> 20\%$ of the runtime is spent in sampling strings for replacement. The current implementation of this re-traverses the trees $\mathcal{T}$ after each bubble to create these examples.

```
      while →  stmt ␣ while | skip | L␣=␣
       stmt →  while␣ bool and-space do | while ␣;
                | if␣ bool and-space then␣ while ␣else
       bool →  false | ˜ bool | true | num ␣==␣ num
  and-space →  ␣ | and-space &␣ bool and
        num →  L | n | ( num + num )
```

Fig. 11: ARVADA-mined `while` grammar with 100% recall. Nonterminals renamed for readability.

```
       json →  str | dict } | false | true | [ ] | pos-int
                | float-start DIGITS | float-start pos-int | int
                | { } | [ json list-end | null | NAT
        str →  str-start ''
       dict →  dict-lst str : json           dict-lst →  dict , | {
    pos-int →  NAT                                int →  - pos-int | NAT
float-start →  int . | pos-int .
   list-end →  , json list-end | ]
  str-start →  `` chars | `` pos-int | str-start pos-int
      chars →  chars chars | pos-int chars | ALNUMS
 DIGITS : [0-9]+   NAT : 0|[1-9][0-9]*   ALNUMS : [a-Z0-9]+
```

Fig. 12: ARVADA-mined `json` grammar with maximum F1 Score. Nonterminals renamed for readability. DIGITS, NAT, and ALNUMS are tokens expanded after the Sec. III-E pass.

On the particularly slow benchmarks, `tinyc` and `nodejs`, ARVADA spends a long time ordering bubbles. This makes sense because of the larger example length of the benchmarks. It is nonetheless encouraging to see this room for improvement, as GETBUBBLES re-scores the full set of bubbles each time a bubble is accepted. It should be possible to bring down runtime by only scoring the bubbles that are modified by the application of the just-accepted bubble. On `nodejs`, ARVADA also spends a long time in oracle queries, because the time for each query is much longer (300 ms vs. 3ms for `tinyc`).

Overall, ARVADA has runtime and number of oracle queries comparable with GLADE, while achieving much higher recall and F1 score. As for RQ3, when the length of the examples in S is small, oracle calls dominate runtime. As example length grows, the ordering and scoring of bubbles—particularly computing context similarity—starts to dominate runtime.

E. Qualitative Analysis of Mined Grammars

The statistics discussed in the prior section show that ARVADA's mined grammars can closely match the ground-truth grammars in terms of inputs generated and accepted. For RQ5, we consider their human-readable complexity.

Mined grammar readability varies across benchmarks. For instance, on the 3 runs where ARVADA achieves 100% recall for `while`, the mined grammars look similar to $\mathcal{G}_w$ Fig. 1: Fig. 11 shows the grammar mined in one of these runs, randomly selected from the three. Fig. 12 shows the grammar with maximum F1 score mined by ARVADA on `json`; it splits some expansions at unusual places (e.g. the use of *float-start*) but is readable after some examination.

For `tinyc`, the mined grammars are somewhat over-bubbled: on average they have 56 nonterminals, and 217 rules of average length 1.8. On `nodejs`, the grammars have on average 40 nonterminals and 276 rules of average length 3.6. Because GLADE's grammars are not meant to be human-readable, they are significantly larger: 3505 nonterminals with 4417 rules of average length 1.3 for `tinyc`; and 2060 nonterminals with 3939 rules of average length 1.2 for `nodejs`.

V. DISCUSSION AND THREATS TO VALIDITY

Our implementation of ARVADA relies on some heuristic elements, which we developed while examining some smaller benchmarks (i.e. `arith`, `while`) on a particular set of example strings. To prevent overfitting on these benchmarks, for evaluation, we used a freshly-generated set of example strings.

The definition of maximal generalization assumes that the language accepted by the oracle is context-free. Thus, we have no formal guarantees on how the algorithm will react to context-sensitive input languages. While our results compared to GLADE are promising, there is no guarantee they will generalize to all benchmarks.

The fact that ARVADA's *maximum* results consistently beat state-of-the-art (Fig. 9) suggests a few directions for improvement. If runtime is not a constraint, ARVADA can be parallelized as-is. To choose the winner, first measure precision with respect to the oracle. Then, evaluate the grammars on inputs sampled from the other mined grammars, and choose the one which captures the most of those samples. A less-wasteful way to parallelize would be to conduct some sort of beam search, perhaps using the just-described comparative generalization metric, or to backtrack bad bubbles.

There remains much room to optimize the order in which bubbles are explored, and pre-tokenization of inputs. We chose two natural metrics for ordering (context similarity and frequency), but have not exhaustively examined how to combine them. From the difference in performance between the larger benchmarks `tinyc` (which had simple regex structure) and `nodejs` (regexes in the training set are more complex), it appears that ARVADA could benefit from running at a higher token level. Developing better heuristics for tokenization, or pairing ARVADA with a more complex regex learning algorithm than that described in Section III-E may yield benefits.

VI. RELATED WORK

Automatically synthesizing context-free grammars from examples is a long-studied problem in computer science; Lee [18], and Stevenson and Cordy [19] give a survey of some techniques. Gold's theorem [20] states that grammars cannot be learned efficiently from a set of positive examples alone. Angluin and Kharitonov [21] show that pure black-box approaches face scalability issues on arbitrary CFGs. But, real-world grammars may not be so adversarial. Our heuristics use statistical information to heavily prune the search space.

The core idea in Solomonoff's [22] algorithm is to, for each example, find substrings of the example that can be deleted. If a substring can be deleted, Solomonoff proposes to add a recursive repetition rule for the substring. Rather than trying to generalize each example string individually, ARVADA considers all example strings together when producing candidate strings. Unlike Arvada, Knobe and Knobe [23] assume a teacher that can provide new valid strings if the current proposed grammar does not match the target grammar. For each new valid string, their algorithm adds the most general valid production of the form $S \rightarrow B_1 B_2 \cdots B_n$ to the grammar, where B_i are terminals or existing nonterminal. It adds new nonterminals by merging nonterminal sequences which have the same left and right contexts in expansions. GLADE [4] learns context-free grammars in two phases. First, it learns a regular expression generalizing each input example. Then, it tries to merge subexpressions of these regular expressions in a manner similar to our label merging. REINAM [5] uses reinforcement learning to refine a learned CFG, allowing fuzzy matching through a PCFG. It is complementary to our work, as the module that learns a CFG (in their evaluation, GLADE), could be replaced by ARVADA.

L^* and RPNI are two classic algorithms for the learning of *regular* languages. L^* [24] learns regular languages with the stronger assumption of a minimally adequate teacher, which can both (1) act as an oracle for the target language , and (2) given a learned regular language, assert whether it is identical to the target language or give a counterexample. RPNI [25] learns regular languages in polynomial time, assuming a set of positive and negative examples. GLADE was found to outperform both these algorithms for program input grammars. The original L^* paper also describes L^{cf}, an algorithm for learning context-free languages in polynomial time, assuming that the set of terminals and non-terminals is known ahead of time. This assumption is not reasonable in most contexts.

Closely related is the field of distributional learning. Clark et. al [26], [27] present polynomial algorithms for learning binary context feature grammars—which capture context-free languages in addition to more complex languages—from strings. The algorithms rely on the representation of words by their *contexts*, an interesting relation to ARVADA's use of k-contexts. Unfortunately, polynomial does not mean fast in practice. We implemented these algorithms in python: even the more efficient one took nearly 5 hours to run on our `while` benchmark. Work on strong learning [28] learns grammars with good parse trees—over tokenized inputs. Again, because it uses full context information, it does not scale to large example sets and overgeneralizes on non-substitutable grammars. This highlights the practical importance of k-contexts.

Also related is the field of automata learning; learnlib [29] is a state-of-the-art Java framework implementing several of these algorithms. In particular, it provides an implementation of the TTT [30] algorithm for learning VPDA. These automata accept a subclass of deterministic context-free languages [31]. TTT is optimized for situation where the key structure of inputs used to query the oracle can be collected in a prefix-closed set, as in learning from logs of system behavior. This is less well-suited to program inputs with multiple distinct recursive structures. TTT also relies on the stronger assumption of a minimally-adequate teacher, rather than a blackbox oracle.

Another branch of works use grey- or white-box information about the oracle to learn grammars. Lin et al.'s work examines execution traces in order to reconstruct program inputs grammar [32], [33]. AUTOGRAM [6] tracks input flows into variables, and uses this dataflow information to learn a well-labeled grammar. Mimid [7] goes a step further, tracking the control-flow nodes in which input characters are accessed. It directly maps this control-flow structure to the grammar structure, and again can take advantage of function names. The use of this additional oracle information may make the final grammars more robust and speed up the inference process. On the other hand, ARVADA's blackbox assumption makes it flexible when this information is not readily accessible, or for strangely-structured programs. Our `tinyc` benchmark was taken directly from Mimid's evaluation, and ARVADA achieved an average F1 score 0.81, compared to Mimid's 0.96. This is impressive given that ARVADA uses the oracle as blackbox.

Section IV-C discussed the use of deep learning to learn input structures for fuzzing. Other techniques do something like grammar mining to increase the effectiveness of fuzzing. Parser-directed fuzzing [34] uses direct comparisons to input bytes to automatically figure out tokens of the input structure; it works best on recursive-descent parsers. GRIMOIRE [35] leverages a sort of one-level grammar by denoting "nonterminal" regions of the code as those which can be changed while maintaining a certain kind of branch coverage.

Lastly, the Sequitur compression algorithm resembles the bubbling phase of ARVADA, bubbling sequences that appear with high frequency [36]. SEQUIN [37] extends Sequitur to mine attribute grammars. Neither algorithm allows for recursive generalization by merging bubble-induced nonterminals.

VII. CONCLUSION

We presented ARVADA, a method for learning CFGs from example strings and oracles. We found that ARVADA outperformed GLADE in terms of increased generalization on 11 benchmarks, with a higher F1 score on average on 9 of these benchmarks. These two benchmarks on which ARVADA performs relatively less well are a regular language (for URLs) and a language with more complex regular expressions for tokens. This, along with qualitative analysis of the inputs generated by ARVADA and GLADE, suggests that ARVADA does best in learning recursive structures over tokens, and that a compelling avenue for improvement is a separate token learning step. ARVADA is available as open source at: https://github.com/neil-kulkarni/arvada.

ACKNOWLEDGEMENTS

Thanks to Rohan Bavishi and all our anonymous reviewers for their invaluable feedback on this paper. This research is supported in part by gifts from Fujitsu Research of America, and NSF grants CCF-1900968, CCF-1908870, CNS-1817122.

References

[1] R. Gopinath and A. Zeller, "Building Fast Fuzzers," *CoRR*, vol. abs/1911.07707, 2019.

[2] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for Deep Bugs with Grammars," in *26th Annual Network and Distributed System Security Symposium*, NDSS '19, 2019.

[3] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-Aware Greybox Fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 724–735, 2019.

[4] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing Program Input Grammars," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, (New York, NY, USA), p. 95–110, Association for Computing Machinery, 2017.

[5] Z. Wu, E. Johnson, W. Yang, O. Bastani, D. Song, J. Peng, and T. Xie, "REINAM: Reinforcement Learning for Input-Grammar Inference," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, (New York, NY, USA), p. 488–498, Association for Computing Machinery, 2019.

[6] M. Höschele and A. Zeller, "Mining Input Grammars from Dynamic Taints," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, (New York, NY, USA), p. 720–725, Association for Computing Machinery, 2016.

[7] R. Gopinath, B. Mathis, and A. Zeller, "Mining Input Grammars from Dynamic Control Flow," in *Proceedings of the 2019 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, (New York, NY, USA), pp. 1–12, Association for Computing Machinery, 2020.

[8] N. Kulkarni, C. Lemieux, and K. Sen, "Learning Highly Recursive Input Grammars," *CoRR*, vol. abs/2108.13340, 2021. Available at https://arxiv.org/abs/2108.13340.

[9] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "Synthesizing benchmarks for predictive modeling," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, p. 86–99, IEEE Press, 2017.

[10] T. J. Parr and R. W. Quong, "ANTLR: A Predicated-LL(k) Parser Generator," *Software — Practice & Experience*, vol. 25, p. 789–810, July 1995.

[11] D. Stenberg, "cURL: command line tool and library for transferring data with URLs." https://curl.se/, 2018. Accessed April 21st, 2021.

[12] T. Berners-Lee, L. Masinter, and M. McCahill, "Uniform Resource Locators (URL) ." https://tools.ietf.org/html/rfc1738, 1994.

[13] F. Bellard, "Tiny C Compiler." https://bellard.org/tcc/, 2018. Accessed April 21st, 2021.

[14] O. Foundation, "NodeJS." https://nodejs.org/en/, 2018. Accessed April 21st, 2021.

[15] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with zest," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, (New York, NY, USA), p. 329–340, Association for Computing Machinery, 2019.

[16] P. Godefroid, H. Peleg, and R. Singh, "Learn&Fuzz: Machine Learning for Input Fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, p. 50–59, IEEE Press, 2017.

[17] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler Fuzzing through Deep Learning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, (New York, NY, USA), p. 95–105, Association for Computing Machinery, 2018.

[18] L. Lee, "Learning of Context-Free Languages: A Survey of the Literature"," tech. rep., Harvard Computer Science Group, 1996.

[19] A. Stevenson and J. R. Cordy, "A Survey of Grammatical Inference in Software Engineering," *Science of Computer Programming*, vol. 96, pp. 444–459, 2014.

[20] E. M. Gold, "Language Identification in the Limit," *Information and Control*, vol. 10, no. 5, pp. 447–474, 1967.

[21] D. Angluin and M. Kharitonov, "When Won't Membership Queries Help?," *J. Comput. Syst. Sci.*, vol. 50, p. 336–355, Apr. 1995.

[22] R. J. Solomonoff, "A new method for discovering the grammars of phrase structure languages," in *Information Processing, Proceedings of the 1st International Conference on Information Processing*, pp. 285–289, UNESCO (Paris), 1959.

[23] B. Knobe and K. Knobe, "A method for inferring context-free grammars," *Information and Control*, vol. 31, no. 2, pp. 129–146, 1976.

[24] D. Angluin, "Learning Regular Sets from Queries and Counterexamples," *Inf. Comput.*, vol. 75, p. 87–106, Nov. 1987.

[25] J. Oncina and P. Garcia, "Identifying Regular Languages In Polynomial Time," in *Advances in Structural and Syntactic Pattern Recognition*, vol. 5 of *Machine Perception and Artifical Intelligence*, pp. 99–108, World Scientific, 1992.

[26] Alexander Clark and Rémi Eyraud and Amaury Habrard, "A Polynomial Algorithm for the Inference of Context Free Languages," in *Grammatical Inference: Algorithms and Applications*, (Berlin, Heidelberg), Springer, 2008.

[27] A. Clark, R. Eyraud, and A. Habrard, "Using Contextual Representations to Efficiently Learn Context-Free Languages," *Journal of Machine Learning Research*, vol. 11, no. 92, pp. 2707–2744, 2010.

[28] A. Clark, "Learning Trees from Strings: A Strong Learning Algorithm for some Context-Free Grammars," *Journal of Machine Learning Research*, vol. 14, no. 75, pp. 3537–3559, 2013.

[29] M. Isberner, F. Howar, and B. Steffen, "The open-source learnlib," in *Computer Aided Verification* (D. Kroening and C. S. Păsăreanu, eds.), (Cham), pp. 487–495, Springer International Publishing, 2015.

[30] M. Isberner, F. Howar, and B. Steffen, "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning," in *Runtime Verification* (B. Bonakdarpour and S. A. Smolka, eds.), (Cham), Springer International Publishing, 2014.

[31] R. Alur and P. Madhusudan, "Adding Nesting Structure to Words," *J. ACM*, vol. 56, May 2009.

[32] Z. Lin, X. Zhang, and D. Xu, "Reverse Engineering Input Syntactic Structure from Program Execution and Its Applications," *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 688–703, 2010.

[33] Z. Lin and X. Zhang, "Deriving Input Syntactic Structure from Execution," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, (New York, NY, USA), p. 83–93, Association for Computing Machinery, 2008.

[34] B. Mathis, R. Gopinath, M. Mera, A. Kampmann, M. Höschele, and A. Zeller, "Parser-Directed Fuzzing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, (New York, NY, USA), p. 548–560, Association for Computing Machinery, 2019.

[35] T. Blazytko, C. Aschermann, M. Schlögel, A. Abbasi, S. Schumilo, S. Wörner, and T. Holz, "GRIMOIRE: Synthesizing Structure While Fuzzing," in *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, (USA), p. 1985–2002, USENIX Association, 2019.

[36] C. G. Nevill-Manning and I. H. Witten, "Identifying Hierarchical Structure in Sequences: A Linear-Time Algorithm," *Journal of Artificial Intelligence Research*, vol. 7, p. 67–82, Sept. 1997.

[37] R. Luh, G. Schramm, M. Wagner, H. Janicke, and S. Schrittwieser, "SEQUIN: a grammar inference framework for analyzing malicious system behavior," *Journal of Computer Virology and Hacking Techniques*, vol. 14, no. 4, pp. 291–311, 2018.