



Fast Deterministic Black-box Context-free Grammar Inference

Mohammad Rifat Arefin
Computer Science and Engineering Department
University of Texas at Arlington
Arlington, Texas, USA

Suraj Shetiya
Computer Science and Engineering Department
University of Texas at Arlington
Arlington, Texas, USA

Zili Wang
Department of Computer Science
Iowa State University
Ames, Iowa, USA

Christoph Csallner
Computer Science and Engineering Department
University of Texas at Arlington
Arlington, Texas, USA

ABSTRACT

Black-box context-free grammar inference is a hard problem as in many practical settings it only has access to a limited number of example programs. The state-of-the-art approach ARVADA heuristically generalizes grammar rules starting from flat parse trees and is non-deterministic to explore different generalization sequences. We observe that many of ARVADA's generalization steps violate common language concept nesting rules. We thus propose to pre-structure input programs along these nesting rules, apply learnt rules recursively, and make black-box context-free grammar inference deterministic. The resulting TREEVADA yielded faster runtime and higher-quality grammars in an empirical comparison. The TREEVADA source code, scripts, evaluation parameters, and training data are open-source and publicly available (<https://doi.org/10.6084/m9.figshare.23907738>).

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**; *Syntax*; *Software reverse engineering*; • **Computing methodologies** → **Active learning settings**; • **Theory of computation** → *Grammars and context-free languages*.

KEYWORDS

Grammar inference, oracle, nested language concepts, bracket-implied nesting structure, deterministic synthesis

ACM Reference Format:

Mohammad Rifat Arefin, Suraj Shetiya, Zili Wang, and Christoph Csallner. 2024. Fast Deterministic Black-box Context-free Grammar Inference. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*, April 14–20, 2024, Lisbon, Portugal. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3597503.3639214>

1 INTRODUCTION

Learning a context-free grammar from sample programs with just the help of a black-box parser currently does not scale well to realistic settings. Existing approaches need a combination of a

large number of sample programs (deep learning), the ability to manipulate a grey-box or white-box parser, or are non-deterministic. The most closely related approach, the recent ARVADA work [22], is non-deterministic and thus ran ARVADA 10 times for each input to explore different sequences of grammar inference steps.

Black-box context-free grammar inference is crucially important when a language only has a black-box parser that cannot be instrumented. On the other hand, program samples are often available (e.g., as open-source code or as example programs from the language vendor). Examples of such languages typically only have closed-source parsers that are often only available remotely (or cannot be instrumented for legal reasons). This unfortunately rules out using white-box or grey-box parser instrumentation [16, 19, 27, 42].

At the same time, grammar inference is important for many software engineering tasks, including code comprehension [31], reverse engineering [28], detecting and refactoring code smells [20, 29], transforming source code [1] for optimization or bug fixing, and generating test inputs [2, 14, 30, 38].

The task of black-box inference of a context-free grammar is fundamentally hard. First, the given input programs likely do not cover all aspects of their language's "golden grammar". Second, it is often very hard to generalize from a few programs exhibiting a few combinations of language features to a grammar describing the language features with the correct nesting rules. Finally, not being able to inspect or instrument the language's parser makes black-box inference significantly harder than grey-box or white-box inference, as a black-box approach has a much narrower access to the parser's encoding of the language's golden grammar.

While there has been a lot of interest in applying deep learning techniques to learning grammars from program samples [11, 15], a principal limitation of deep-learning approaches is that (a) they need a very large amount of training samples (which may not be available) and (b) they do not take advantage of black-box parsers that are typically available even for closed-source languages. Indeed, the ARVADA paper reports on a comparison with state-of-the-art deep learning approaches, in which deep-learning tools did not match the precision of either GLADE [5] or ARVADA [22].

While ARVADA [22] has made significant improvements over the pioneering GLADE [5, 6] work, it still has several limitations. For example, ARVADA has $O(n^4)$ runtime in its n input tokens and requires its "seed" input programs to be very short. For example, on average ARVADA mostly produced [22] grammars within 5 minutes with over 80% F1 scores when running on a few dozen hand-selected minimal sample programs that on average consist of just 1.7 to



This work is licensed under a Creative Commons Attribution International 4.0 License. *ICSE '24, April 14–20, 2024, Lisbon, Portugal*
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0217-4/24/04
<https://doi.org/10.1145/3597503.3639214>

```

start → stmt
stmt → stmt ; stmt | L = numexpr | skip
      | while_ boolexp _do_ stmt
      | if_ boolexp _then_ stmt _else_ stmt
boolexp → ~boolexp | true | false
         | boolexp & boolexp | numexpr == numexpr
numexpr → ( numexpr + numexpr ) | L | n

```

Figure 1: *while*’s golden grammar \mathcal{G}_w (Arvada’s motivating example [22, Figure 1], reformatted, plus missing skip rule).

12.5 characters. However when running on 25 randomly generated nodejs programs with an average length of 50 characters, ARVADA yielded on average a 29% F1 score—after a 12 hour runtime.

TREEVADA combines several new techniques. First, to guide its grammar generalization steps to avoid breaking common nesting rules, TREEVADA first pre-structures its input programs according to nesting rules induced by balanced brackets that are common in many languages [41]. Depending on the language grammar’s nesting structure this step reduces TREEVADA’s runtime from ARVADA’s $O(n^4)$ downward up to $O(n^2)$. Second, once TREEVADA accepts a grammar generalization step, TREEVADA applies this generalization rule recursively. Finally, building on these techniques TREEVADA carefully omits non-determinism and thus yields a reproducible grammar in a single run. In an empirical comparison the resulting TREEVADA implementation achieved both faster runtime and better grammar quality than the most-closely-related ARVADA tool. To summarize, the paper makes the following major contributions.

- TREEVADA is the first fast deterministic black-box approach for context-free grammar inference that produces high-quality grammars.
- The paper compares TREEVADA empirically with its closest competitor (ARVADA) using ARVADA’s setup and achieves faster runtime and better grammar quality.
- The TREEVADA source code, scripts, evaluation parameters, and training data are open-source and publicly available (<https://github.com/rifatarefin/treevada>) and archived (<https://doi.org/10.6084/m9.figshare.23907738>).

2 BACKGROUND

While there are trade-offs and special cases, “context-free” remains an important abstraction level for programming language definition, both for human-level programming language understanding and for automated language processing tools. For example, the latest versions of the official language specifications of complex mainstream languages such as Java [17], JavaScript [45], and C++ [21] include in their language syntax descriptions context-free grammars. Similarly, many sample grammars of the widely-used ANTLR4 line of parser generators [33] are context-free.

Figure 1 is a small example context-free grammar. As usual, a rule with alternatives ($l \rightarrow r_1 \dots r_m \mid r_n \dots r_z$) is just shorthand for having both a first ($l \rightarrow r_1 \dots r_m$) and second ($l \rightarrow r_n \dots r_z$) rule. Each rule is thus essentially of the same form ($l \rightarrow r_1 \dots r_m$) with a single non-terminal (l) on the left and a sequence (r_1, \dots, r_m)

of terminals, non-terminals, or both on the right. This, of course, allows *recursive rules* (e.g.: $boolexp \rightarrow \sim boolexp$) and *balanced nesting structures* (e.g.: $numexpr \rightarrow (numexpr + numexpr)$).

Many programming languages allow balanced nesting of language concepts, where a concept has a dedicated start terminal and a dedicated end terminal and a concept can contain other concepts [41]. For these start and end terminals many languages use matching round, square, and curly brackets: $() [] \{ \}$. Common thereby nested concepts include class and function definitions, parameter lists, code blocks, array creation and access, and various other expressions—for example, a code block containing other code blocks that in turn contain arithmetic expressions.

2.1 Black-box Grammar Inference

The long-term goal of this line of work is to reverse engineer the (unknown) grammar (or specification) of a programming language from only two things, (1) a few valid sample programs and (2) a black-box parser. Having such a specification would support various software engineering tasks, including code comprehension [31], reverse engineering [28], smell detection and refactoring [20, 29], test input generation [18], and code transformation [1].

For example, some popular commercial languages (e.g., MATLAB/Simulink) neither have a formal specification nor parsers that can be analyzed. Specifically, the language’s tools are closed-source and cannot be instrumented for legal or technical reasons (e.g., they are only available as a remote service). But valid sample programs are often widely available—via GitHub or the vendor’s website (to document language features and encourage language adoption).

We follow ARVADA’s definition [22] of grammar quality. So an inferred grammar \mathcal{G}_i is better if it has a higher F1 score, i.e., if the set of input programs \mathcal{G}_i accepts is closer to the set of input programs accepted by the golden grammar.

2.2 State-of-the-art Inference: ARVADA in $O(n^4)$

Black-box inference of context-free grammars was pioneered by GLADE [5, 6] and recently advanced by ARVADA [22]. ARVADA’s evaluation showed ARVADA’s average run provided on average some 5× improvement in recall and 3× improvement in F1 score over GLADE (while being just 30% slower).

ARVADA initially treats each input program as a flat parse tree, i.e., a single rule that can only reproduce the given input program. ARVADA then iteratively generalizes grammar rules. It groups a few (leaf and/or internal) parse-tree sibling nodes t_s, \dots, t_u (aka a “bubble”) under a new bubble parent node b , reflecting a candidate grammar rule ($b \rightarrow t_s \dots t_u$). ARVADA then picks an existing interior tree node a and its child nodes t_c, \dots, t_e , which together implicitly define a grammar rule ($a \rightarrow t_c \dots t_e$). It then checks if it can rename the new bubble parent node to (and thereby merge it with) the existing interior node, yielding the generalized rule ($a \rightarrow t_c \dots t_e \mid t_s \dots t_u$).

ARVADA heuristically accepts such a rule merge when a black-box parser accepts up to 100 freshly generated sample programs that exercise the newly-merged rule. Since each such merge check (and especially failing it) is expensive, ARVADA orders its potential bubbles via heuristics. The key heuristic is to compare the k siblings immediately before (“left k -context”) and after (“right k -context”) a

candidate bubble. ARVADA thus ranks a bubble higher if the bubble’s contexts are more similar to the contexts of existing interior tree nodes. The secondary bubble-ranking metric is each bubble’s occurrence count in the input programs’ parse trees (a higher bubble occurrence frequency yields a higher rank). To increase the chance of a merge, ARVADA also tries to merge two bubbles directly with each other (“2-bubble”), for which it ranks all bubble pairs.

The evaluation of ARVADA (and GLADE) [22] points to two scalability issues. (1) First, GLADE’s and ARVADA’s training sets only contain very small input programs, i.e., the largest input programs range from just 5 (arith language) to 245 characters (tinyc). (2) Second, relatively more complex languages (tinyc and especially nodejs) have the relatively larger golden grammars and input programs. Besides the higher runtime, here GLADE and ARVADA also yield lower F1 scores. Following are the key technical challenges of ARVADA.

2.2.1 Arvada Run = 10 Non-deterministic $O(n^4)$ Runs. ARVADA’s first key challenge is its non-determinism, which makes the results hard to reproduce. For example, when we ran it 10 times on the Figure 2 input programs S_1 , ARVADA produced two different grammars. Non-determinism also creates a trade-off between using the first run’s grammar vs. re-running ARVADA in the hope of finding a better grammar. On each set of input programs, the ARVADA work ran ARVADA 10 times, effectively yielding an order-of-magnitude worse total runtime than the reported average runtime.

ARVADA uses non-determinism to explore various sequences of grammar generalization steps. Such a generalization sequence can get ARVADA stuck in the sense of cutting off subsequent generalization options, reducing the inferred grammar’s quality. For example, the ARVADA study reported for several languages a high F1 score variance among its 10 runs. For example, among 10 nodejs runs F1 scores ranged from 0.14 to 0.55. Following are ARVADA’s four main sources of non-determinism.

Shuffling Initial Candidate Node-pair Merges: ARVADA first tokenizes each input program along character classes (lower-case, uppercase, digits, whitespace), keeping only other ASCII (aka “punctuation”) and non-ASCII characters as individual tokens. (ARVADA treats each such resulting token as the only child of a token-specific “dummy” parent node connected to the root node—for brevity we omit from figures these dummy nodes.)

ARVADA then tries an initial attempt (“MERGEALLVALID”) to generalize grammar rules. Specifically, it creates all pairs of existing non-terminal (mostly dummy) nodes across all parse trees, orders the pairs arbitrarily (by storing them in a non-deterministic data structure), and tries to merge each pair. For example, as Figure 2’s two initial parse trees contain 12 unique non-terminal node types (t_0 and implicit parents of `while`, `_`, `n`, etc.), ARVADA tries merging 66 node pairs, which yields one successful merge (`skip` with t_0).

Ranking & Shuffling $O(n^4)$ Candidate Merges: After the initialization phase, for each grammar generalization step ARVADA first (re-) collects and (re-) ranks all possible parse-tree sibling-node sequences (“1-bubbles”) up to a configurable length together with their pairs (“2-bubble”). For n tokens in the initial parse trees there are essentially $O(n^2)$ 1-bubbles, which makes the ranking overall $O(n^4)$. ARVADA then takes the top-100 candidates, shuffles them, stores the existing non-terminal tree nodes in a non-deterministic structure, and iteratively tries the merges, until one succeeds. For

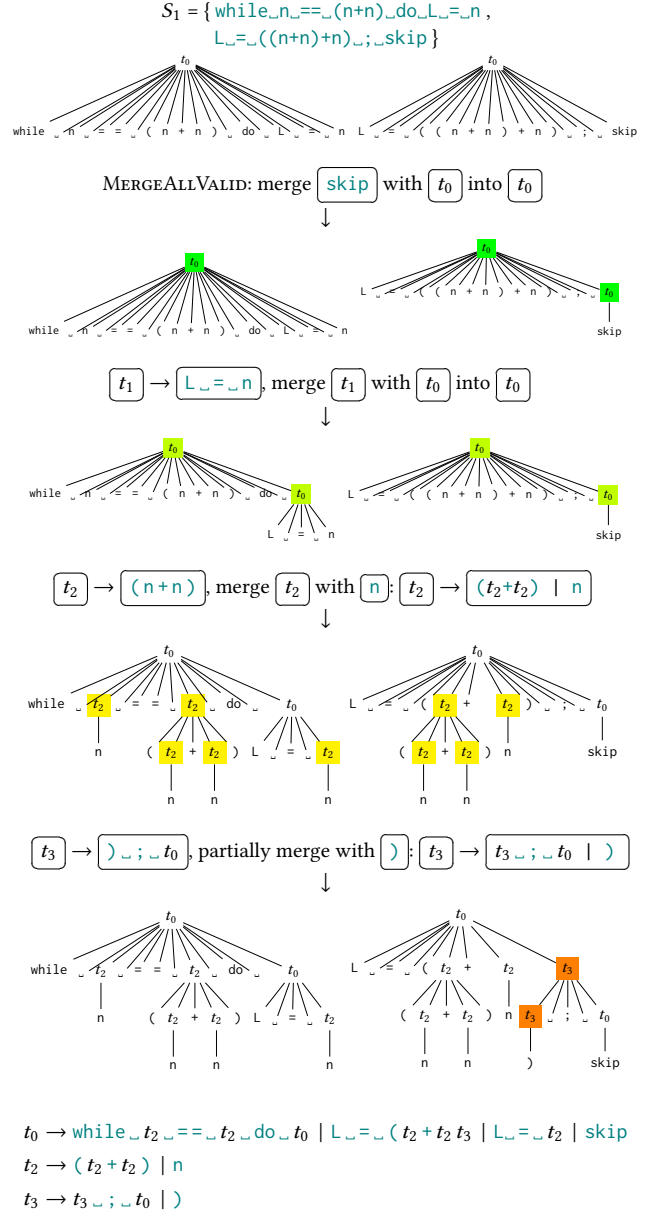


Figure 2: Top to bottom: Input `while` programs S_1 and a resulting ARVADA run: initial (pre-tokenized) flat parse trees, initial node-pair merges (green), 1st bubble merge (lime), 2nd bubble merge (yellow) without reapplying rule, and 3rd bubble merge (orange) breaking tree nesting; resulting grammar.

example, in Figure 2’s first bubbling step ARVADA ranks 1,043 candidate 1- and 2-bubbles to merge the lime bubble ($t_1 \rightarrow \text{L_} = _ n$).

Accepting Rule Generalization Via Sampled Programs: For both above cases (merging initial single nodes or a candidate bubble), ARVADA accepts the merge if the black-box parser accepts up to 100 freshly generated programs. From all programs that exercise

the proposed generalized grammar rule, ARVADA samples these 100 programs (50 per merged side) uniformly.

Final Step (Expand Terminals): At the end ARVADA expands each terminal to a larger character class, so the grammar may accept tokens that were not in the seed programs. For example, ARVADA tries to expand $t_1 \rightarrow 1 \mid 2$ to all single digits, integers, or alphanumeric letters. ARVADA then samples 10 strings, generates programs, and checks them via the parser. A grammar’s terminals may thus differ across ARVADA runs on the same seed inputs.

2.2.2 Not Generalizing Recursively. ARVADA’s second challenge is that it does not recursively reapply a rule generalization it just learned and thus on some runs needs additional expensive steps or gets stuck. For example, Figure 2 shows 5/10 runs we observed ARVADA pursue for the S_1 input programs. As the second bubble (yellow) it grouped sibling nodes $(n+n)$ under new bubble parent t_2 and merged t_2 with n ’s (not shown) dummy parent into t_2 .

While this bubble yields an appropriate generalized grammar rule ($t_2 \rightarrow n \mid (t_2+t_2)$), ARVADA does not recursively reapply this just learned rule to its parse trees—even though the sibling sequence (t_2+t_2) is now present in the right parse tree. Instead, ARVADA re-ranks all bubbles (an expensive operation), picks and merges another bubble (orange), and thus gets stuck.

2.2.3 Breaking Bracket-implied Nesting Structure. Many languages use matching round $()$, square $[\]$, and curly $\{ \}$ brackets to recursively nest concepts such as class and function definitions, code blocks, parameter lists, array creation and access, and various other expressions. ARVADA’s third key challenge is that on some runs it prioritizes a bubble that conflicts with a parse tree’s bracket-implied nesting structure and thus gets stuck.

For example, in some runs on the Figure 2 input programs S_1 , ARVADA breaks the while language’s numerical expression nesting, which is defined via matching round brackets. In these runs ARVADA partially merges the bracket-wise unbalanced orange bubble $(_; _t_0)$ with the implicit parent of the last closing bracket. ARVADA then cannot further generalize the grammar. The resulting grammar is recursive. But for statement sequences (recursive applications of the semicolon) it only allows very specific instantiations, i.e., each generated statement sequence must start with an assignment statement that contains at least one addition $(L_ = _ (t_2+t_2 \dots))$.

3 OVERVIEW AND DESIGN

We guided our design via feedback from running ARVADA and our alternatives on ARVADA’s seed programs for tinyc [22]. To prevent over-fitting we did not use feedback from any other programs or languages we used in the subsequent evaluation.

3.1 Assumptions on Strings & Brackets

TREEVADA’s current heuristics build on two “soft” assumptions, i.e., that many languages (1) use `' '` quotes to wrap strings and (2) use $() \ [\] \ \{ \}$ brackets for nesting. If a language (also) uses these characters for other purposes then TREEVADA’s F1 score may suffer.

3.2 Pre-tokenizing Input Programs

As many languages share basic tokenization (or lexing) rules (e.g., an identifier is separated by some non-identifier token from the

following token), TREEVADA and ARVADA first tokenize their input programs. Tokenizing both likely yields higher-quality grammars and is more efficient than on each run rediscovering common lexing rules via relatively expensive grammar inference.

Both approaches thus replace a sequence of elements of one of the four character classes (lower-case, uppercase, whitespace, or digits) with a new terminal. This leaves all brackets, punctuation, and other “special” characters as individual character terminals. For example, on the S_1 input programs of Figure 2, ARVADA and TREEVADA produce the same token sequence.

3.2.1 Program Structure in String Literals. While not part of the main three problems we focus on, we also notice that ARVADA treats the contents of string literals as program structure. For example, during initial tokenization ARVADA tokenizes the 7-character input fragment `"k_-: -)"` into 7 nodes. It may then bubble and merge some of these nodes and get stuck. We thus want to distinguish string literals from program elements.

As this is not the paper’s main focus we use a simple heuristic that solves several scenarios that are common in many languages. Specifically, we notice that many languages wrap a string literal in single (`'`) or double (`"`) quotes. When it encounters either quote character TREEVADA thus groups all following characters until again encountering the same quote character. While this scheme cannot handle all cases (e.g., escaped quote characters), it tokenizes common simple cases correctly, e.g., `"k_-: -)"` into three tokens, one per double-quote character plus one for the string literal’s content.

3.3 Pre-structuring Parse Trees Along Brackets

We observe that ARVADA’s first bubble-ranking generalization step is its most expensive, as it may rank in $O(n^4)$ all pairs of all possible sibling token sequences of the input programs. Many such bubbles are likely illegal as they cross a round/square/curly bracket “boundary” and thus violate a nesting rule that is common among languages. This becomes clear on the extreme example program of $n = 2b + 1$ tokens that starts with b round opening brackets followed by x and b round closing brackets. ARVADA’s first bubble generalization step ranks $O(n^4)$ bubble pairs. Most such bubbles cross a nesting boundary and thus the parser likely rejects them.

The deeper-structured the parse-trees become via generalization steps, the cheaper each subsequent generalization step is. These later steps are cheaper not so much due to earlier grammar generalization but as a more-structured tree only permits shorter (and thus fewer) sibling token sequences. Our goal is thus to quickly convert parse trees from flat to richly structured, by essentially enforcing common nesting rules. In the above extreme nesting example, the nesting-implied parse tree consists of root node plus a single stack of b layers of a single bracket-wrapped node, reducing $O(n^4)$ to a single $O(n^2)$ step, i.e., the upfront MERGEALLVALID.

Given the wide use of round/square/curly bracket-defined nesting, TREEVADA pre-structures parse trees heuristically—likely without significantly impairing the inferred grammar’s quality. Specifically, TREEVADA makes one simple stack-based pass over each input program, initializing the stack (and the parse tree) with root t_0 . TREEVADA adds each token to the parse tree as the child of the current top-of-stack node. When encountering an opening $(\ [\ {$ bracket, TREEVADA first pushes a new non-terminal onto the stack.

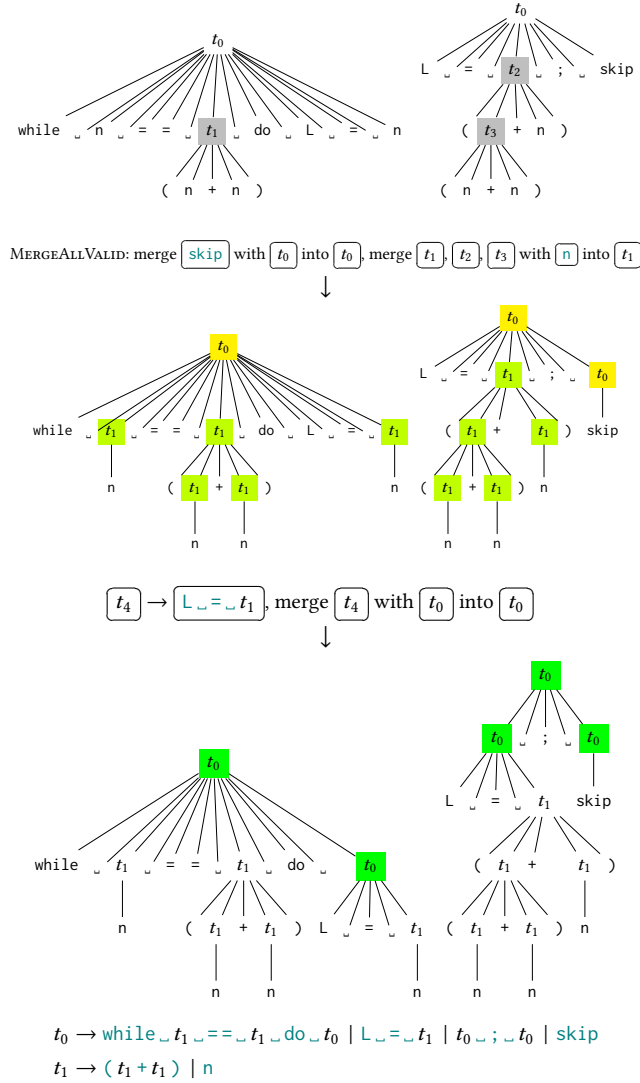


Figure 3: Top to bottom: TREEVADA’s pre-structured bracket-implied trees for Figure 2’s S_1 input programs with bracketed sequences (gray) bubbled, initial node-pair merges (yellow & lime), and 1st bubble via bubble-ranking (green); the inferred grammar captures S_1 ’s Figure 1 golden grammar rules.

When encountering a matching closing bracket `)] }`, TREEVADA then pops the top element off the stack. When brackets no longer match TREEVADA reverts to a flat tree.

The subsequent attempt to merge all tree node pairs often merges some of these new rules with each other or other nodes. For example, the Figure 3 run merges the bracket-implied t_1 , t_2 , and t_3 with the existing node n . If a pre-structured rule remains un-merged it does not generalize the grammar. As it just adds slightly to grammar verbosity, we do not remove such rules.

ARVADA’s motivating *while* example language uses brackets only lightly (i.e., only one rule in the Figure 1 golden grammar contains brackets). But even then there are several cases (e.g., for the Figure 2

input programs) where TREEVADA is faster, infers a better grammar, or does both. Figures 2 and 3 is an example of the latter.

3.4 Removing Specialized Bubbling Heuristics

As TREEVADA creates nesting structure upfront, there is less need for special cases and we therefore remove the following two rarely successful strategies ARVADA uses.

1-bracket Bubbles: Pre-structuring the parse trees ensures that each sibling node sequence contains at most two round, curly, or square brackets. While this prevents a bubble from crossing concept nesting, we observe that one bracket rarely generalizes the grammar correctly either. Not generating 1-bracket bubbles thus ensures that TREEVADA never considers a bracket-unbalanced bubble.

Partial 1-char Node Merges: When ARVADA cannot merge a given bubble with any interior node, it also tries merging the bubble with a subset of interior node instances that represent a terminal character, e.g., to merge the bubble with one `)` instance but not others. As both ARVADA and TREEVADA pre-tokenize their input programs, such special treatment of 1-char tokens rarely yields a successful merge and TREEVADA thus omits such partial merges.

3.5 Deterministic Grammar Inference

TREEVADA addresses ARVADA’s sources of non-determinism. Several of these were easy to fix without significant performance degradation, just by switching the implementation to a deterministic data-structure. Specifically, TREEVADA orders the parse trees’ unique node types by their shortest distance from any program root node (in ascending order) for two related operations. First, when trying to merge all node-pairs upfront and after exhausting bubbling. Second, when ordering merge-target nodes for a given bubble.

Similarly, when there are more than 50 candidate programs that exercise a candidate rule-merge on one of the two merged rules, TREEVADA makes ARVADA’s program sampling strategy deterministic, by switching to deterministic data structures and always using the same random number generator seed value. Finally, TREEVADA makes ARVADA’s terminal expansion deterministic, by fixing the random number generator’s seed value to sample 10 programs from the larger character class. Following is a more complex case, where TREEVADA needed a new heuristic to compensate for ARVADA’s benefits from arbitrary order and randomness.

3.5.1 Depth- and Length-aware Bubble Ranking. To avoid ARVADA’s non-deterministic shuffling of the top-100 ranked bubbles, TREEVADA builds on two observations. First, a longer bubble has a higher chance of being rejected because it tries to group together more nodes. A longer bubble, when accepted, also has a higher chance of getting grammar inference stuck. For example, in Figure 2 ARVADA’s last merge is a 5-node bubble $t_3 \rightarrow) ; t_0$. The bubble’s use of `)` prevents `)` from being used as the closing bracket in an otherwise possible subsequent bubbling of $(t_2 + t_2)$.

Second, and more importantly, a more deeply nested bubble is more promising than a bubble closer to the root. The intuition is that a more deeply nested node sibling sequence is in a more specialized area of the input program that has more of its immediate surroundings already correctly structured via other rules. The likelihood of correctly generalizing such an already specialized area

thus tends to be higher and, crucially, the impact of getting it wrong is lower—as it will only affect a relatively specialized program area.

TREEVADA thus refines the bubble ranking, by adding two new criteria, bubble depth and bubble length. We have found that with these additions the bubble ranking is more reliable and does not require shuffling to get to promising candidate bubbles early on. The resulting ranking scheme ranks by context similarity first, then resolves ties via bubble depth (i.e., the bubble occurrences' minimum root-distance), further ties via bubble occurrence counts, and additional ties via bubble length.

3.6 Applying Learned Rules Recursively

After merging a bubble, ARVADA does not recursively reapply the just learned grammar rule. For example, in Figure 2 ARVADA merged $t_2 \rightarrow (n+n)$ with n into t_2 , yielding rule $t_2 \rightarrow (t_2+t_2) \mid n$. While ARVADA proceeds by re-ranking all bubbles, TREEVADA here instead directly tries to recursively reapply the just learned rule as renamed by the merge (i.e., $t_2 \rightarrow (t_2+t_2)$).

For the Figure 2 scenario, TREEVADA would group (t_2+t_2) under a new t_2 node, yielding fewer direct children under the second parse tree's root node ($\perp \leftarrow t_2 \leftarrow t_0$). This cheaply adds to the parse trees' structure we have just accepted as correct. In this case it would also prevent ARVADA from its last bubble step that gets it stuck by breaking the parse tree's nesting structure.

4 EVALUATION

Overall we would like to get a better understanding of how TREEVADA compares with the state-of-the-art approach ARVADA, both on very small and slightly larger input programs. While the larger input programs may not yet be representative of how a user would want to apply these approaches on other languages, it at least gives us a glimpse of the scalability of the compared approaches. We thus seek to answer the following research questions.

RQ0 Baseline: How does non-determinism affect ARVADA?

RQ1 Grammar quality: At similar runtime, does TREEVADA infer better grammars than ARVADA?

RQ2 Runtime: When inferring grammars of similar quality, does TREEVADA have a lower runtime than ARVADA?

RQ3 Readability: How compact are the inferred grammars?

RQ4 Ablation study: How do TREEVADA's components influence its resource consumption and grammar quality?

To ease comparison, we run our experiments in ARVADA's Docker image¹. Specifically, from the image we reuse the ARVADA, black-box parser, grammar-sampler, random program generator, and ANTLR4 parser generator [33] binaries. From the image we also use the languages' existing 1k test programs. The following summarizes the metrics we reuse from ARVADA's work.

Precision: From each ARVADA-/TREEVADA-inferred grammar we sample 1k programs and count how many of these 1k programs the respective existing "golden" black-box parser accepts.

Recall: We compile the ARVADA-/TREEVADA-inferred grammar into a parser and count how many of the given 1k ("golden") test programs that parser accepts.

F1 score: As usual, the F1 score is the harmonic mean of precision and recall and ranges from 0 (zero precision or zero recall) to 1 (both perfect precision and perfect recall).

Runtime: The main measure is the ARVADA/TREEVADA runtime, which does not include computing precision or recall.

Averages: We follow ARVADA [22] in comparing a deterministic technique's result with the average of 10 non-deterministic runs. The latter estimates what a user may expect from running ARVADA once. We also plot each of our ARVADA and TREEVADA runs.

4.1 RQ0: Timeouts vs. Precision Results

To get a sense of non-determinism's effect on ARVADA we first reproduce ("different team, same experimental setup")² ARVADA's main results [22, Table 1]—i.e., runtime and F1 score. We use ARVADA's Docker image (same ARVADA configuration options, etc., including the same ("seed") training input programs) and reran all languages from ARVADA's experiment as in the ARVADA paper 10 times. Here we used a 24GB RAM Ryzen-9 5900HX 3.30GHz CPU laptop.

Table 1: ARVADA study [22, Table 1] (left); our rerun (middle), and TREEVADA (right) on the same input programs; ARVADA values are average over 10 runs; f1 = F1 score; t = runtime; \pm = standard deviation; bold = unreliable precision & F1 score.

	Earlier study [22]		Rerun ARVADA		TREEVADA	
	f1	t[s]	f1	t[s]	f1	t[s]
arith	1 \pm .00	3 \pm 0	1\pm.00	1 \pm 0.1		
fol	.91 \pm .18	372 \pm 36	.88 \pm .19	127 \pm 19	n/a	
math.	.89 \pm .08	65 \pm 6	.82 \pm .13	19 \pm 1.8		
json	.97 \pm .05	76 \pm 11	.97 \pm .04	28 \pm 3.1	.96	22
lisp	.57 \pm .21	16 \pm 4	.58 \pm .22	6 \pm 1.2	.71	7
turtle	1 \pm .00	84 \pm 8	1 \pm .00	31 \pm 3.0	1.0	25
while	.81 \pm .14	54 \pm 5	.83 \pm .12	20 \pm 1.6	.38	16
xml	.96 \pm .08	205 \pm 34	.88 \pm .30	78 \pm 8.2	.09	88
curl	.68 \pm .11	111 \pm 12	.68 \pm .10	60 \pm 6.9	.63	102
tinyc	.81 \pm .08	6.4k \pm 1.2k	.69 \pm .16	5k \pm 1k	.81	1.5k
nodejs	.29 \pm .16	46k \pm 22k	.39 \pm .16	23k \pm 6k	.56	4.5k

Table 1 summarizes the results. First, in our rerun the runtime was consistently lower (likely due to the different machine). For the main grammar quality measure (F1 score), the average over all 11 languages was similar (80.8% vs our rerun's 79.3%). For individual languages the impact was larger. For example, tinyc's average F1 score dropped from 81% to 69%. It may thus be misleading to compare ARVADA performance across languages. For example, while in the earlier study ARVADA produced much better grammars for tinyc than for curl (81% vs 68%), this difference all but disappeared in our rerun (69% vs 68%).

From the source code we learnt that when calling a black-box parser ARVADA's precision calculation enforces a 10s timeout. While the programs are just a few dozen tokens, arith's "golden" parser timed out for many programs sampled from the ARVADA-inferred grammars, with parse time growing quickly with program size. On one example arith run 76/1k programs timed out. ARVADA's metric

¹Accessed in January 2024: <https://github.com/neil-kulkarni/arvada>

²<https://www.acm.org/publications/policies/artifact-review-and-badging-current>

tool unfortunately treats a timeout as if the parser accepted the program, which very likely corrupts precision (and thus F1 score). As we could not easily solve the problem (e.g., by increasing the timeout by 10×), we exclude arith from tool comparisons.

Excluding a language for unreliable measurement is not meant to avoid running TREEVADA. On the Table 1 arith seeds TREEVADA “scores” 100% precision via ARVADA’s metric tool. Due to the metric tool’s parser timeout treatment, this result is equally unreliable.

4.2 Experimental Setup for RQ1 to RQ4

While the ARVADA work carefully constructed minimal input programs that cover all rules of a given golden grammar, we aim to emulate a more realistic scenario where the user does not have a golden grammar and thus cannot construct a minimal set of minimal input programs. Hence we rely on the ARVADA Docker image’s 1k test programs. These programs do not guarantee to cover all golden grammar rules. For the first 5/8 languages, we randomly pick seed inputs from this pool of 1k programs (which may slightly inflate recall but still allows comparing TREEVADA with ARVADA).

Table 2: Input programs S with their avg/max TREEVADA pre-tokenized tokens; # = nr. programs same for R1 and R2; Q = programs containing a " or ' ; c = tinyc; js = nodejs.

	$S_{H/R0[22]}$			$S_{R1/R5}$				S_{R2}		
	#	avg	m	#	avg	m	Q	avg	m	Q
json	71	3.0	13	30	8.2	72	20	5.4	25	15
lisp	26	2.3	7	30	76.7	418	0	23.8	120	0
turtle	33	7.0	13	35	30.2	91	0	19.2	63	0
while	10	7.8	12	30	110.7	534	0	135.1	1298	0
xml	40	11.4	20	20	24.4	62	14	25.4	75	13
curl	25	13.7	25	25	15.0	26	1	14.6	29	0
c	25	69.1	207	25	81.5	218	0	72.7	213	0
js	25	49.0	146	15	60.0	116	4	65.9	244	5
c-500	n/a			10	420.3	483	0	n/a		
js-500	n/a			5	307.0	385	2	n/a		

As in the ARVADA work, for the next 3/8 languages (curl, tinyc, and nodejs) we do not have a golden grammar. We thus use the same 3rd-party random program generators the ARVADA work used (with their default settings) to create new seed programs. For all 8 languages the new input sets S may thus not cover all golden grammar rules. We call this random input set $R1$.

Table 2 compares the $R1$ input programs with the ones used in the ARVADA work—the first 5 languages had handpicked seeds (“H”) and the next 3 used 3rd-party generators (“R0”). We focus on the token counts via TREEVADA’s tokenization scheme, as it only differs in how it treats " and ', which only a few input programs contain. Compared to the ARVADA study, the average token count tends to be larger—for most of the first 5 languages by an order of magnitude. Especially the largest programs are significantly larger.

To explore larger input programs we generate another set $R5$ using the same generator used for tinyc [16] (“tinyc-500”) and nodejs [32] (“nodejs-500”). Here we skip programs under 200 characters long. Table 2 shows tinyc-500 and nodejs-500 programs are

on average 5× larger than tinyc and nodejs programs by token count. $R1/R5$ seed (and test) programs either both did (json, xml, curl, js, js-500) or did not have some programs with quotes. All $R1/R5$ seed (and test) programs had brackets, except for xml. Curl had one seed program with unmatched brackets.

Here we run each experiment on an EPYC Milan 7763 64-core CPU machine in TACC’s Lonestar6 cluster³, which does not support Docker. We thus recreated the Docker image’s setup as closely as possible (same oracle binaries, etc.). After removing arith we removed the parser timeouts. First, we removed a 3s parser timeout ARVADA used for its grammar inference. This should improve the quality of ARVADA-inferred grammars as it no longer has to interpret a parser timeout as “parsed ok”.

After removing the 3s parser timeouts, sampled programs of fol and math also got stuck (for at least 30 minutes each) in their “golden” parser (due to poorly written grammars). To protect precision calculation’s integrity we also removed fol and math. For precision calculation we could then remove its 10s parser timeout, which allowed us to just use the first 1k programs sampled from an ARVADA-/TREEVADA-inferred grammar (ARVADA’s evaluation [22] silently discarded any sampled program over 300 characters).

We started all experiments with 32GB RAM. As ARVADA’s grammar inference ran out of memory on lisp, nodejs, and nodejs-500, for these three experiments we then used an otherwise identically configured 256GB RAM machine.

4.3 RQ1, RQ2: Precision, Recall, F1, Resources

Table 3 shows the main evaluation results (on $R1$ and $R5$). Across all 10 experiments, TREEVADA on average both produces better grammars and is faster than ARVADA, i.e., TREEVADA has a 9.3% higher recall, a 22.1% higher precision, a 19.5% higher F1 score, and a 2.4× speedup over ARVADA. In 9/10 experiments TREEVADA inferred a grammar of the same or higher quality than the average ARVADA run (i.e., TREEVADA’s F1 score was at least as high as ARVADA’s).

The outlier is curl, where TREEVADA’s 72% F1 score is slightly below ARVADA’s 78%. curl has brackets but does not use them for nesting. TREEVADA’s attempts to pre-structure the input programs’ parse trees thus either fail quickly during the initial pass over the input programs or get TREEVADA stuck with a sub-optimal grammar.

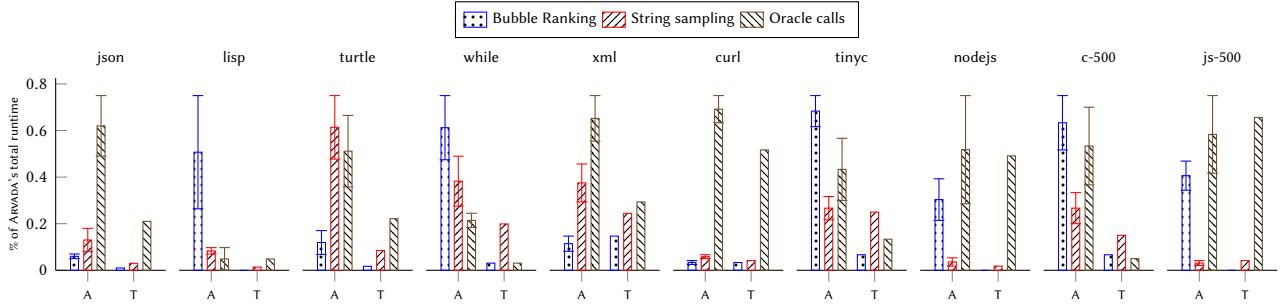
Compared to our reruns of the earlier study (Table 1, middle), ARVADA increased its F1 score on 2/8 languages—i.e., for curl from 68 to 78% and while from 83 to 100%, likely due to non-determinism and differences in input programs. On the other hand, switching from hand-picked minimal programs to randomly selected input programs here may have contributed to lowering ARVADA’s F1 score in 4/8 other languages—json from 97 to 79%, xml from 88 to 76%, tinyc from 69 to 62%, and nodejs from 39 to 12%.

At the same time, in all 10 experiments TREEVADA was faster than the average ARVADA runtime. For 6/10 experiments TREEVADA was at least twice as fast as ARVADA’s average run. On the larger input programs tinyc-500 and nodejs-500 (which have about double the total number of input tokens as tinyc and nodejs), TREEVADA remains faster than the average ARVADA run and achieves higher F1 scores, i.e., 63 vs. 67% on tinyc-500 and 12 vs. 46% on nodejs-500.

³Accessed in January 2024: <https://tacc.utexas.edu/systems/lonestar6>

Table 3: Average ARVADA & TREEVADA results over 10 runs; r = recall; p = precision; f1 = F1 score; t = runtime; t₀ = oracle time; q = queries sent to oracle; m = peak memory usage; ± = standard deviation; bold = TREEVADA ≥ 2× better in main metrics.

Name	ARVADA on randomly selected seeds: R1, R5 (R5 = c-500 & js-500)							TREEVADA on same set of random seeds, m in GB						
	r	p	f1	t[ks]	t ₀ [ks]	q[k]	m[GB]	r	p	f1	t[ks]	t ₀ [ks]	q[k]	m
json	.97 ±.02	.68 ±.13	.79 ±.09	.13 ±.0	.08 ±.0	18.0 ±4.2	.05 ±.0	.90	.96	.93	.03 ±.0	.03 ±.0	9.0	.03
lisp	.53 ±.38	.93 ±.13	.57 ±.27	6.1 ±1.5	.44 ±.4	35.8 ±30.9	25.7 ±20.2	1.0	.98	.99	.70 ±.0	.39 ±.0	58.3	.06
turtle	1.0 ±.01	.95 ±.08	.97 ±.05	.56 ±.1	.17 ±.1	29.1 ±8.3	.06 ±.0	1.0	.94	.97	.12 ±.0	.08 ±.0	16.7	.04
while	1.0 ±.00	.99 ±.03	1.0 ±.01	7.8 ±.9	1.1 ±.2	53.2 ±9.6	.76 ±.2	1.0	1.0	1.0	2.1 ±.0	.19 ±.0	34.2	.13
xml	1.0 ±.00	.70 ±.37	.76 ±.30	.30 ±.0	.12 ±.0	25.2 ±3.8	.05 ±.0	1.0	1.0	1.0	.16 ±.0	.05 ±.0	12.5	.08
curl	.90 ±.09	.71 ±.14	.78 ±.09	.15 ±.0	.12 ±.0	20.8 ±2.1	.05 ±.0	.60	.89	.72	.12 ±.0	.09 ±.0	20.3	.05
tinyc	.76 ±.20	.54 ±.19	.62 ±.18	7.4 ±1.0	1.9 ±.6	131.5 ±27.2	.77 ±.1	.97	.86	.91	3.1 ±.0	.56 ±.0	164.5	.22
nodejs	.06 ±.07	.25 ±.09	.08 ±.09	8.7 ±3.3	5.1 ±2.3	130.5 ±69.2	3.1 ±1.7	.47	.82	.59	5.1 ±.0	4.8 ±.0	169.2	.07
c-500	.55 ±.20	.81 ±.08	.63 ±.16	29.6 ±4.7	9.4 ±3.1	247.8 ±82.7	2.4 ±1.1	.53	.93	.67	6.8 ±.2	.85 ±.0	175.5	.40
js-500	.12 ±.15	.26 ±.19	.12 ±.15	23.0 ±5.0	13.0 ±3.6	218.9 ±56.7	8.9 ±2.7	.37	.62	.46	16.7 ±.3	14.4 ±.3	480.4	.35

**Figure 4: Average (and standard deviation) of time spent on ranking bubbles, sampling strings, and in black-box parser. Each value is normalized by dividing by ARVADA’s average total runtime for that language (R1, R5 seed); A = ARVADA; T = TREEVADA.**

For scalability it is further interesting to consider the group of 7/10 experiments with the highest total token counts in their input programs—i.e., lisp, turtle, while, tinyc, nodejs, tinyc-500, and nodejs-500. In these 7 experiments TREEVADA has either much better grammar quality at similar runtime, much better runtime at similar grammar quality, or both much better grammar quality and runtime. For example, on the experiment with the highest total token count (tinyc-500 with some total 4.2k tokens), TREEVADA’s F1 score is 67% vs ARVADA’s 63% while using less than a quarter of the time. To better understand the differences between ARVADA and TREEVADA, we further compare the following metrics.

Bubble Ranking Time: Both ARVADA and TREEVADA rank their potential grammar generalization steps (aka bubbles). The main difference is that TREEVADA tries to omit from this ranking bubbles that may violate common bracket-defined nesting rules.

String Sampling Time: Except for the use of non-determinism, ARVADA and TREEVADA use the same scheme for sampling programs that exercise a proposed grammar rule merge.

Oracle Calls & Time: ARVADA and TREEVADA call an external parser during grammar inference in the same way.

Memory Use: We measure the peak memory use during grammar inference via Linux’s *time* command.

Results: TREEVADA consistently uses the same or less memory than ARVADA. In 3/10 experiments this difference is one (nodejs,

nodejs-500) or even two (lisp) orders of magnitude. Figure 4 reinforces that the source of this difference is the large difference in time spent on bubble ranking. Given that in these three experiments TREEVADA also yields significantly higher F1 scores makes clear that much of ARVADA’s bubble ranking is counter-productive as it prioritizes bubbles that ultimately get the grammar inference stuck.

Bubble ranking is the dominating time expenditure for 4/10 of the ARVADA experiments, but in none of the TREEVADA experiments. Instead the external parser dominates TREEVADA’s overall runtime in 7/10 experiments and in 6/7 of these cases by a large margin. In the remaining 3/10 cases TREEVADA’s bottleneck is string sampling.

4.4 RQ3: Grammar Readability

Neither ARVADA nor TREEVADA attempt to simplify their inferred grammars. They just export the state of the grammar when they cannot find any additional grammar generalization steps. Since there are use-cases involving human consumption such as program understanding, it is still interesting to determine if higher grammar quality comes at the expense of larger grammars.

A related question is if a larger grammar for a given language is maybe structured more efficiently for parsing, i.e., in parser runtime and memory consumption. To explore these two related questions we thus measure the following two metrics.

Table 4: Grammar size and parse performance on R1 & R5: Averages for grammars inferred in 10 Table 3 runs; NT/T = unique (non-) terminals; A = rules (alternatives); l(A) = avg. rule length; S = sum of rule lengths; tp = time to parse 1k “golden” test programs; mp = peak memory while parsing; \pm = standard deviation (TREEVADA all zero); bold = TREEVADA $\geq 2\times$ better.

	ARVADA							TREEVADA, mp in GB						
	NT	A	l(A)	S	T	tp[ks]	mp[GB]	NT	A	l(A)	S	T	tp[ks]	mp
json	30 ±4	153 ±11	1.4 ±0	208 ±19.8	95 ±1	.02 ±0	.03 ±0	17	178	1.2	208	82	.01	.03
lisp	18 ±10	91 ±27	14.2 ±9.9	1040 ±639.2	40 ±0	.14 ±2	.04 ±0	18	95	1.8	171	40	.08	.04
turtle	28 ±6	123 ±17	1.6 ±2	197 ±43.9	67 ±0	.03 ±0	.04 ±0	15	95	1.4	131	67	.02	.03
while	33 ±5	85 ±13	2.3 ±1	191 ±26.6	18 ±0	.30 ±2	.22 ±1	24	55	2.3	127	18	.09	.08
xml	20 ±3	95 ±7	1.6 ±1	149 ±13.8	58 ±0	.02 ±0	.03 ±0	9	78	2.0	157	58	.02	.03
curl	22 ±2	174 ±28	1.4 ±1	237 ±29.1	91 ±14	.80 ±5	.07 ±0	16	137	1.4	189	82	.16	.04
tinyc	57 ±7	229 ±22	1.9 ±2	436 ±31.9	52 ±3	.70 ±7	.08 ±1	30	149	1.8	264	50	.34	.05
nodejs	46 ±9	273 ±43	2.6 ±6	699 ±66.4	95 ±3	.16 ±2	.04 ±0	36	196	1.8	351	95	.12	.04
c-500	63 ±7	246 ±30	2.2 ±6	514 ±75.7	51 ±2	.60 ±3	.09 ±0	41	187	1.8	345	50	.26	.05
js-500	53 ±6	319 ±38	3.0 ±4	935 ±47.8	95 ±1	10.6 ±10.4	9.9 ±10.0	45	319	1.9	596	102	.43	.10

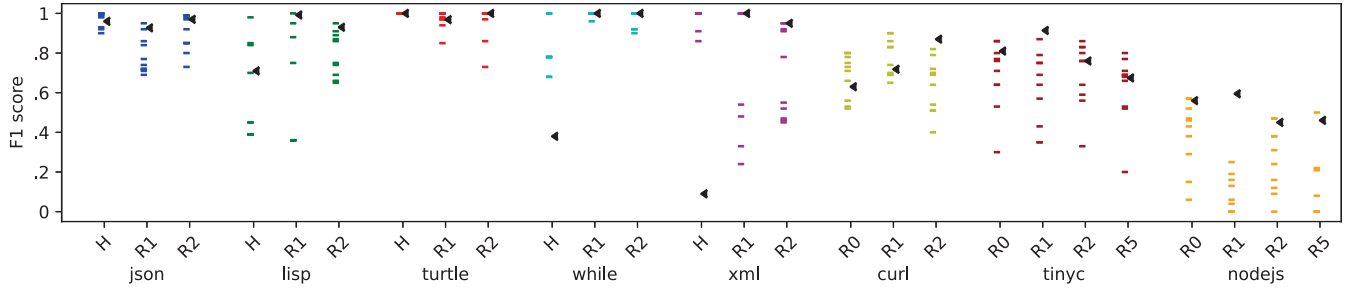


Figure 5: F1 score of 10 ARVADA (–) and TREEVADA (◄) runs on hand-picked (H [22]) and random seeds (R0 [22], R1, R2, R5).

Grammar Size: We count a grammar’s unique non-terminals, unique terminals, number of rules (i.e., rule alternatives), and each rule’s length (i.e., the length of a rule’s right-hand-side sequence of terminals and non-terminals). The grammar’s size is then the sum its rule lengths.

Parse Time & Memory: We measure the total time required and peak memory used to parse the 1k “golden” test programs using a parser generated from an ARVADA-/TREEVADA-inferred grammar.

Results: Table 4 gives an overview of grammar size and parse performance. Despite covering more of the golden grammars (higher recall) and having higher F1 scores, TREEVADA’s grammars are smaller for 8/10 languages. The biggest difference is lisp where TREEVADA’s grammar is less than one sixth the size of the average ARVADA grammar. For json, TREEVADA’s grammar size equals ARVADA’s average grammar size. The only outlier is xml (157 vs. 149)—where TREEVADA has a significantly higher F1 score.

ARVADA’s larger grammars do not improve parsing performance, as there is no experiment in which ARVADA’s average parse time or memory use is lower than TREEVADA’s. On the contrary, for 6/10 experiments, TREEVADA’s parse time is less than half of ARVADA’s.

4.5 Performance Variance Across Seed Sets: R2

To compare performance across seeds we generate a fresh round of random input program sets (“R2”) in the same style as for R1.

Table 2 shows R2’s size metrics. In R2 all seed sets contain some brackets, except for xml and curl.

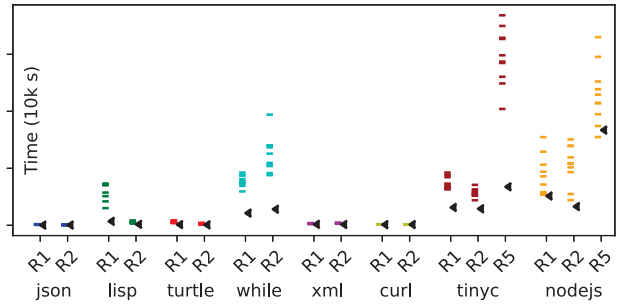


Figure 6: Runtime of Figure 5 runs; H/R0 runtimes omitted as done on different machines.

For space we only plot their individual results (with the Table 1+3 results for context), i.e., F1 score in Figure 5 and runtime in Figure 6. TREEVADA’s results were stable compared to ARVADA’s in the sense that across R1, R2, and R5 (except for curl on R1) TREEVADA’s F1 score was better than the average ARVADA F1 score and sometimes better than every ARVADA F1 score. Similarly, across all R1, R2, and R5 runs, in the single run where ARVADA was faster than TREEVADA (js-500 = R5 nodejs) ARVADA’s F1 score was zero.

Specifically, on the R1 and R2 seeds TREEVADA’s F1 score was better than all ARVADA runs for both while and xml. Across all seed sets, TREEVADA had a better F1 score than at least 9/10 ARVADA runs for nodejs. Notable outliers are TREEVADA’s while and xml runs on the ARVADA work’s seeds (H). Since the while language uses C-style bracket nesting, TREEVADA does not seem overly overfitted for C-style programming languages.

4.6 RQ4: Ablation Study

Here we explore the impact of TREEVADA’s components, using Table 3’s experimental setup. We first make Table 3’s ARVADA deterministic (Section 3.5). For each language the resulting Table 5 and following ablation tables show the average of 10 runs. For space we omit standard deviations (all very small or zero).

Making ARVADA deterministic yielded several interesting effects. For example, for lisp recall and F1 score are down, runtime is up, parser calls are down, and memory consumption is up—each by about a factor of two. Overall, though, average F1 scores improved while runtimes slowed down. This indicates that some of ARVADA’s non-deterministic runs got stuck in sub-optimal grammars and thus terminated relatively quickly.

From Table 5 to Table 6 we add recursive rule application (Section 3.6). For most languages this change had little impact on F1 scores. The notable exception are nodejs, tinyc-500, and nodejs-500, which all had lower F1 scores and a faster runtime. Since the initial bracket-implied parse tree are not imposed here and partial merge is used, this version still has bubbles breaking nesting rules. Re-applying such learned nesting-breaking rules does not improve F1 scores (but may get the grammar stuck relatively quickly).

From Table 6 to Table 7 we add bracket-based initial parse trees (Section 3.3) and ignoring likely string literal contents (Section 3.2). Adding these two features makes the F1 score of lisp jump from 0.34 to 0.99 with a decrease in memory use from 58.7 to only 0.07GB. Additionally, lisp’s runtime drops from 11.4k to 0.47k seconds. All 10/10 experiments had improvement in runtime, while languages with more nesting structure benefited the most.

From Table 7 to Table 8 we remove partial merges. The F1 scores tend to improve. Especially nodejs’s F1 score spiked from 0.09 to 0.56, which indicates that partial merges were responsible for some invalid merges, which essentially blocked learning. Finally, from Table 8 to TREEVADA in Table 3 we add the new bubble ranking scheme (Section 3.5.1), which has overall neutral results on F1 scores but overall reduces runtimes.

5 THREATS TO VALIDITY

We briefly summarize key threats to internal and external validity.

Threats to external validity: Often grammar inference tools show mixed performance with different seeds [6]. Also relying on hand-crafted small toy programs as seed does not replicate real-world situations. To overcome these challenges, we randomly sample seed sets R1, R2, R5 for our experiments.

Threats to internal validity: TREEVADA’s pre-structuring may fail if brackets serve other language-specific purposes than nesting. For example, in xml names enclosed in angle brackets `<>` are used for nesting, whereas in Java or C++ these are mainly used for relational or bit manipulation operators. We have carefully chosen

Table 5: Table 3’s ARVADA + deterministic

	r	p	f1	t	t _O	t _B	t _S	q[k]	m
json	.98	1.0	.99	.09	.06	.01	.02	18	.05
lisp	.21	1.0	.34	11.5	.13	8.6	2.6	14	58.7
turtle	1.0	1.0	1.0	.27	.07	.02	.15	16	.06
while	1.0	1.0	1.0	13.7	.38	1.5	8.8	35	.38
xml	1.0	1.0	1.0	.42	.07	.04	.19	19	.08
curl	.87	.75	.80	.14	.11	.00	.01	25	.05
tinyc	.81	.60	.69	9.5	.82	1.7	4.0	141	.66
nodejs	.05	.41	.08	17.6	11.7	2.5	1.9	420	.73
c-500	.68	.68	.68	52.3	6.4	8.3	22.1	360	1.6
js-500	.40	.29	.33	51.2	39.1	8.5	2.6	692	9.5

Table 6: Table 5 + ReApply learnt rules; rc = reApply count

	r	p	f1	t	t _O	t _B	t _S	q[k]	m	rc
json	.98	1.0	.99	.10	.06	.01	.02	20	.05	0
lisp	.21	1.0	.34	11.4	.13	8.5	2.6	15	58.7	1
turtle	1.0	1.0	1.0	.32	.09	.02	.17	17	.06	0
while	1.0	1.0	1.0	14.4	.40	1.6	9.2	38	.39	0
xml	1.0	1.0	1.0	.47	.08	.04	.21	21	.08	3
curl	.83	.77	.80	.15	.12	.00	.01	27	.05	1
tinyc	.81	.60	.69	9.77	.91	1.6	4.2	152	.67	4
nodejs	.01	.44	.03	9.71	6.3	2.2	.76	210	1.6	6
c-500	.66	.62	.64	48.2	9.4	6.8	18.7	480	1.7	1
js-500	.00	.73	.00	45.2	31.9	9.6	2.6	548	10.0	13

Table 7: Table 6 + Initial bracket tree + Ignore string literals

	r	p	f1	t	t _O	t _B	t _S	q[k]	m	rc
json	.98	.90	.94	.05	.04	.00	.00	10	.03	2
lisp	.98	1.0	.99	.47	.32	.00	.07	44	.07	2
turtle	1.0	1.0	1.0	.16	.09	.01	.04	18	.04	0
while	.98	1.0	.99	3.7	.65	.29	1.8	75	.21	1
xml	1.0	1.0	1.0	.19	.08	.03	.05	16	.07	2
curl	.58	.84	.69	.14	.12	.00	.01	24	.04	1
tinyc	.88	.43	.58	3.6	.85	.19	1.4	200	.20	6
nodejs	.05	.54	.09	7.1	6.3	.03	.38	183	.08	8
c-500	.66	.74	.69	16.3	2.6	1.2	7.5	459	.60	4
js-500	.56	.35	.43	12.8	10.9	.11	.77	298	.33	13

only three bracket types for pre-structuring parse tree as these three are the most commonly used for nesting [41]. For terminal expansion, synthesizing regular expressions for the terminals would give a more robust grammar, which is future work.

6 RELATED WORK

Grammatical inference is important as the inferred grammar can serve many tasks [39] when the language’s golden grammar is unknown. Rather than probabilistic learning [24, 40], *active learning* [3] is a good fit as a parser can serve as a *minimally adequate*

Table 8: Table 7 + Remove partial merge

	r	p	f1	t	t _O	t _B	t _S	q[k]	m	rc
json	.90	.98	.93	.04	.03	.00	.00	10	.03	0
lisp	1.0	1.0	1.0	.60	.36	.00	.11	47	.07	0
turtle	1.0	.67	.80	.27	.10	.01	.09	22	.04	0
while	1.0	1.0	1.0	2.3	.20	.17	1.2	33	.13	0
xml	1.0	1.0	1.0	.16	.06	.03	.04	13	.07	0
curl	.89	.90	.89	.13	.11	.01	.01	26	.05	1
tinyc	.87	.56	.68	2.8	.46	.18	1.2	136	.18	0
nodejs	.44	.76	.56	5.7	5.1	.02	.30	131	.07	2
c-500	.46	.97	.62	12.3	1.1	.60	6.4	235	.35	0
js-500	.48	.57	.52	17.6	14.7	.11	1.2	410	.34	0

teacher (MAT or oracle). Even as a black-box the MAT/oracle can answer membership queries. Recent work [5, 22, 42] follows this setting to infer grammars. Following is other related work.

6.1 Linguistics

The linguistics community has developed several negative [4, 12, 13] and positive results on grammar inference in a wide variety of settings, i.e., for various grammar classes (including context-free), oracles, and availability of additional kinds of input [36]. GRIDS [23] starts with flat parse trees and iteratively bubbles and merges rules. Maybe most closely related to ARVADA is applying a GRIDS-like approach iteratively [46] and thus sampling new inputs from an updated grammar. Maybe most closely related to TREEVADA are Sakakibara’s techniques for inferring a subclass of context-free grammars when given positive examples with their complete (but unlabeled) parse trees plus a parser-like oracle and a grammar equivalence oracle [34, 35].

We observe that a program’s bracket-implied nesting structure can be captured by a Dyck language [8], i.e., a context free language comprising of only balanced brackets. TREEVADA’s pre-structured parse trees can thus be seen as instances of the Dyck language D_3 on the alphabet $(,), [,], {, }$. An interesting property is that a Dyck language captures the “non-regular essence” of a context-free language. Specifically, the Chomsky–Schützenberger representation theorem [10] says any context-free language can be mapped to the intersection of a Dyck language and a regular language.

For TREEVADA, this pre-structuring is just a startup heuristic. For example, in subsequent steps TREEVADA may discover additional balanced structures defined by some other opening and closing terminal pair (which taken together may then be represented by a D_4 language). Similarly, TREEVADA may not be able to merge rules in the initial parse trees.

6.2 Deep Learning

Several reports have been negative on using deep learning for inference of a context-free grammar (CFG) [37, 43]. RNNs lack the ability to learn concrete hierarchical rules, leading to a decline in generalization with increasing input length and recursion depth [7, 44]. LSTMs learn statistical approximations, not a deterministic rule-based solution [37]. Even the state of the art attention based Seq2Seq

model struggles to understand CFGs [44]. ARVADA showed significantly better precision than an LSTM-based model (measuring recall is not possible). Deep learning’s under-performance may be due to solely relying on (many) input samples. GLADE, ARVADA, and TREEVADA use *active learning* [3], where an additional oracle guides the learning process. Another significant deep learning limitation is that it does not produce an explicit grammar, which makes it also hard to measure recall and F1 score.

6.3 Grey-box Grammar Inference

The grey-box grammar inference approaches don’t make use of the entire parser source code. GRIMOIRE [9] is a grey-box fuzzing tool that makes use of the parser’s coverage information. GRIMOIRE synthesizes a grammar like structure of the inputs while fuzzing.

6.4 White-box Grammar Inference

White-box grammar inference approaches utilize the parser source code to extract input grammars that follow the structure of the input. Lin et al. [25, 26] proposed the first white-box method that recovers parse trees from inputs using static and dynamic analysis. *Autogram* [19], introduced by Höschle et al, adopts another white-box approach that tracks the dynamic data flow between variables of the program to infer an approximate context-free grammar. *Mimid* [16] by Gopinath et al. infers a grammar by leveraging dynamic control flow and tracking input character access across parser locations.

7 CONCLUSIONS

Black-box context-free grammar inference is a hard problem as in many practical settings it only has access to a limited number of example programs. The state-of-the-art approach ARVADA heuristically generalizes grammar rules starting from flat parse trees and is non-deterministic to explore different generalization sequences. We observe that many of ARVADA’s generalization steps violate common language concept nesting rules. We thus propose to pre-structure input programs along these nesting rules, apply learnt rules recursively, and make black-box context-free grammar inference deterministic. The resulting TREEVADA yielded faster runtime and higher-quality grammars in an empirical comparison. The TREEVADA source code, scripts, evaluation parameters, and training data are open-source and publicly available.

ACKNOWLEDGMENTS

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper (<http://www.tacc.utexas.edu>). Christoph Csallner has a potential research conflict of interest due to a financial interest with Microsoft and The Trade Desk. A management plan has been created to preserve objectivity in research in accordance with UTA policy. This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 1911017 and a gift from MathWorks.

REFERENCES

- [1] Bhisma Adhikari, Eric J Rapos, and Matthew Stephan. 2021. Simulink Model Transformation for Backwards Version Compatibility. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 427–436.
- [2] Ziyad Alsaeed and Michal Young. 2023. Finding Short Slow Inputs Faster with Grammar-Based Search. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1068–1079.
- [3] Dana Angluin. 1981. A note on the number of queries needed to identify regular languages. *Information and control* 51, 1 (1981), 76–87.
- [4] Dana Angluin and Michael Kharitonov. 1991. When won't membership queries help? In *Proc. 23rd Annual ACM Symposium on Theory of Computing (STOC)*. 444–454.
- [5] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *Proc. 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 95–110.
- [6] Bachir Bendrissou, Rahul Gopinath, and Andreas Zeller. 2022. "Synthesizing input grammars": A replication study. In *Proc. 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM, 260–268.
- [7] Jean-Philippe Bernardy. 2018. Can recurrent neural networks learn nested recursion? *Linguistic Issues in Language Technology* 16 (2018).
- [8] Jean Berstel and Luc Boasson. 2002. Balanced grammars and their languages. In *Formal and Natural Computing: Essays Dedicated to Grzegorz Rozenberg*. Springer, 3–25.
- [9] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing.. In *USENIX Security Symposium*, Vol. 19.
- [10] Noam Chomsky and Marcel Paul Schützenberger. 1963. The Algebraic Theory of Context-Free Languages. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (Eds.). North-Holland Publishing, 118–161.
- [11] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *Proc. 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 95–105.
- [12] Colin De La Higuera. 1997. Characteristic sets for polynomial grammatical inference. *Machine Learning* 27 (1997), 125–138.
- [13] Rémi Eyraud, Colin de la Higuera, and Jean-Christophe Janodet. 2007. LARS: A learning algorithm for rewriting systems. *Machine Learning* 66 (2007), 7–31.
- [14] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*. 206–215.
- [15] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine learning for input fuzzing. In *Proc. 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 50–59.
- [16] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining input grammars from dynamic control flow. In *Proc. 28th Symposium on the Foundations of Software Engineering (FSE)*. ACM, 172–183.
- [17] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. 2022. *The Java Language Specification: Java SE 19 Edition*. Oracle.
- [18] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT international workshop on automating TEST case design, selection, and evaluation*. 45–48.
- [19] Matthias Höschle and Andreas Zeller. 2016. Mining input grammars from dynamic taints. In *Proc. 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 720–725.
- [20] Tae-Woong Kim, Tae-Gong Kim, and Jai-Hyun Seu. 2013. Specification and automated detection of code smells using OCL. *International Journal of Software Engineering and Its Applications* 7, 4 (2013), 35–44.
- [21] Thomas Köppe et al. 2022. *Working Draft, Standard for Programming Language C++*. Technical Report N4917. ISO/IEC.
- [22] Neil Kulkarni, Caroline Lemieux, and Koushik Sen. 2021. Learning Highly Recursive Input Grammars. In *Proc. 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 456–467.
- [23] Pat Langley and Sean Stromsten. 2000. Learning context-free grammars with a simplicity bias. In *Machine Learning: ECML 2000: 11th European Conference on Machine Learning Barcelona, Catalonia, Spain, May 31–June 2, 2000 Proceedings*. Springer, 220–228.
- [24] Ming Li and Paul MB Vitányi. 1991. Learning simple concepts under simple distributions. *SIAM J. Comput.* 20, 5 (1991), 911–935.
- [25] Zhiqiang Lin and Xiangyu Zhang. 2008. Deriving input syntactic structure from execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 83–93.
- [26] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2009. Reverse engineering input syntactic structure from program execution and its applications. *IEEE Transactions on Software Engineering* 36, 5 (2009), 688–703.
- [27] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Reverse Engineering Input Syntactic Structure from Program Execution and Its Applications. *IEEE Transactions on Software Engineering* 36, 5 (2010), 688–703.
- [28] Leon Moonen. 2001. Generating robust parsers using island grammars. In *Proceedings eighth working conference on reverse engineering*. IEEE, 13–22.
- [29] Csaba Nagy and Anthony Cleve. 2017. A static code smell detector for SQL queries embedded in Java code. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 147–152.
- [30] Hoang Lam Nguyen, Nebras Nassar, Timo Kehler, and Lars Grunse. 2020. Mo-fuzz: A fuzzer suite for testing model-driven software engineering tools. In *Proc. 35th IEEE/ACM International Conference on Automated Software Engineering*. 1103–1115.
- [31] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *Proc. 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 574–584.
- [32] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 329–340.
- [33] Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) parsing: The power of dynamic analysis. In *Proc. ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. ACM, 579–598.
- [34] Yasubumi Sakakibara. 1990. Learning context-free grammars from structural data in polynomial time. *Theoretical Computer Science* 76, 2-3 (1990), 223–242.
- [35] Yasubumi Sakakibara. 1992. Efficient learning of context-free grammars from positive structural examples. *Information and Computation* 97, 1 (1992), 23–60.
- [36] Yasubumi Sakakibara. 1997. Recent Advances of Grammatical Inference. *Theoretical Computer Science* 185, 1 (Oct. 1997), 15–45. [https://doi.org/10.1016/S0304-3975\(97\)00014-5](https://doi.org/10.1016/S0304-3975(97)00014-5)
- [37] Luzi Sennhauser and Robert C Berwick. 2018. Evaluating the ability of LSTMs to learn context-free grammars. *arXiv preprint arXiv:1811.02611* (2018).
- [38] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis*. 244–256.
- [39] Andrew Stevenson and James R Cordy. 2014. A survey of grammatical inference in software engineering. *Science of Computer Programming* 96 (2014), 444–459.
- [40] Leslie G Valiant. 1984. A theory of the learnable. *Commun. ACM* 27, 11 (1984), 1134–1142.
- [41] Rijnard van Tonder and Claire Le Goues. 2019. Lightweight multi-language syntax transformation with parser parser combinators. In *Proc. 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 363–378.
- [42] Zhengkai Wu, Evan Johnson, Wei Yang, Osbert Bastani, Dawn Song, Jian Peng, and Tao Xie. 2019. REINAM: Reinforcement learning for input-grammar inference. In *Proc. Symposium on the Foundations of Software Engineering (FSE)*. ACM, 488–498.
- [43] Daniel M Yellin and Gail Weiss. 2021. Synthesizing context-free grammars from recurrent neural networks. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 351–369.
- [44] Xiang Yu, Ngoc Thang Vu, and Jonas Kuhn. 2019. Learning the Dyck language with attention-based Seq2Seq models. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. 138–146.
- [45] Shu yu Guo, Michael Ficarra, and Kevin Gibbons. 2022. *ECMAScript 2022 Language Specification* (13th ed.). Technical Report ECMA-262. Ecma International.
- [46] Willem H. Zuidema. 2002. How the Poverty of the Stimulus Solves the Poverty of the Stimulus. In *Proc. Advances in Neural Information Processing Systems (NIPS)*. MIT, 43–50. <https://proceedings.neurips.cc/paper/2002/hash/04ad5632029cbfbed8e136e5f6f7ddfa-Abstract.html>