





A Refinement Based Algorithm for Learning Program Input Grammars

Hannes Sochor^(✉)  and Flavio Ferrarotti 

Software Competence Center Hagenberg, Hagenberg, Austria
{hannes.sochor,flavio.ferrarotti}@scch.at

Abstract. We propose and discuss a new algorithm for learning (or equivalently synthesizing) grammars. The algorithm provides good approximations to the set of valid inputs accepted by computer programs. Different to previous works, our algorithm assumes a seed grammar whose language grossly overestimates the target program input language, in the sense that it ideally constitutes a super-set of that. It works by reducing the set of productions from the seed grammar through a heuristically guided process of step-by-step refinement, until a good approximation of the target language is achieved. Evaluation results presented in this paper show that the algorithm can work well in practice, despite the fact that its theoretical complexity is high. We present the algorithm in the context of a use-case and consider possible forms of seed grammar extraction from program abstract syntax trees.

Keywords: Program input · Grammar · Learning · Synthesis · Fuzzing

1 Introduction

We propose a new algorithm for learning (or equivalently synthesizing) grammars that provide good approximations to the set of valid inputs accepted by computer programs. Our algorithm makes use of information extracted from the source code of the targeted program via standard multi-language reverse engineering tools, as provided by the eKnows platform developed in our research institute Software Competence Center Hagenberg (SCCH) [13]. More concretely, we assume that the input to the algorithm is a generic abstract syntax tree (AST). The ASTs are extracted by a fully-automated tool which works with diverse programming languages, including COBOL, Java and C among others. In this sense, we can say that our approach is generic, i.e., not tied to a specific programming language. Our main motivation is the possibility of using the resulting grammars together with grammar-based fuzzers for automated testing (see e.g. [3, 4, 8, 12]).

The research reported in this paper was supported by the Austrian Research Promotion Agency (FFG) through the COMET funding for the Software Competence Center Hagenberg.

© Springer Nature Switzerland AG 2022

J. Bowles et al. (Eds.): DataMod 2021, LNCS 13268, pp. 138–156, 2022.

https://doi.org/10.1007/978-3-031-16011-0_10

Different algorithms have been proposed in the literature for learning grammars. Some prominent examples can be found in [1, 2, 7, 10, 14]. What all of them seem to have in common is the fact that they all start from a seed grammar (obtained by diverse methods) which only covers a small subset of the targeted language. The grammar is then successively expanded by new rules by using some inductive learning method, so that it covers an increasing number of valid words. We follow a different approach. We start from a seed grammar that over-approximates the targeted language, ideally one that produces a super-set of the input language recognised by the program. This grammar is then refined in successive steps by methodically eliminating rules (of the grammar) responsible for producing invalid words.

In order to learn program input grammars, one needs to examine a finite number of executions of the program with different inputs, and then generalize these observations to a representation of valid inputs. Typical approaches are either white-box or black-box. In the former case, the learning algorithm needs full access to the source code of the program as well as to that of its associated libraries for examination during the learning process. See for instance the approach based in dynamic taint analysis implemented in [11]. By contrast, the black-box approaches do not require the source code of the program, only relying on the ability to execute the program for a given input to determine whether this input is valid or not. Common black-box approaches (see e.g. [2]) take as input a small, finite set of examples of valid inputs, and then incrementally generalize this language. Sometimes negative examples are also considered, so that gross over-generalizations can be avoided.

In this paper we are interested in what we could call a “grey-box” approach, since it sits somehow in between the white- and black-box approaches. As we have mentioned before, we do not allow direct access to the source code of the program during the learning process, only to its AST. This has the advantage that our approach is not tied to a specific programming language and particular libraries. That is, we can apply our approach without the need for time consuming re-configurations and tuning. The open question is whether we can still hope to achieve results that are comparable to state-of-the-art white-box approaches such as [11]. Different to pure black-box approaches, we can still access some generic information extracted from the source code. In a related work [16], the black-box approach have been combined with a symbolic execution engine to generate an initial set of inputs for the given target program. The question here is whether the access to the AST of the program in our approach results in an actual advantage with respect to (only) relying on symbolic execution for the initial examples in an otherwise full black-box approach.

Since it is not decidable in general to determine whether the language of a grammar G corresponds to a program input language \mathcal{L}_p , it is usual in this area to consider appropriate measures of precision and recall of G w.r.t. \mathcal{L}_p . High precision (i.e. close to 1) means in this context that most of the words in the language $\mathcal{L}(G)$ of G are also in \mathcal{L}_p . Conversely, high recall means that most of the words in \mathcal{L}_p are also in $\mathcal{L}(G)$. The new approach to learning program input

grammars proposed in this paper can be roughly described by the following two-step process:

1. *Seed grammar extraction*: Consists on extracting an initial grammar using the AST of the program source code. Here the focus is on obtaining a seed grammar that has a high recall, even at the expense of precision.
2. *Grammar refinement*: Consists on increasing the precision of the seed grammar by means of successive refinement steps of the set of productions.

Our main contribution in this paper concerns Step 2 above. This is presented in Sects. 4 and evaluated in Sect. 6. We also provide some initial ideas regarding Step 1 in Sect. 3. Although this part is still a work in progress, we think it is important to discuss that in this paper, so that the reader can have a better idea of the possibilities and limitations of the proposed approach. The necessary background and formal definitions of the key measures of precision and recall are introduced in the next section. The last section of the paper includes the conclusions and discusses future steps.

2 Preliminaries

In this section we fix the notation used through the paper. We assume basic knowledge of language theory as presented in the classic book by Hopcroft and Ullman [9].

Let Σ be an *alphabet*, i.e., a finite set of symbols. A finite sequence of symbols taken from Σ is called a *word* or *string* over Σ . The free monoid of Σ , i.e., the set of all (finite) strings over Σ plus the empty string λ , is denoted as Σ^* and known as the *Kleene star* of Σ . If $v, w \in \Sigma^*$, then $vw \in \Sigma^*$ is the *concatenation* of v and w and $|vw| = |v| + |w|$ is its length. If $u = vw$, then v is a *prefix* of u and w is a *suffix*. A *language* is any subset of Σ^* .

A grammar is formally defined as a 4-tuple $G = (N, \Sigma, P, S)$, where N and Σ are finite disjoint sets of *nonterminal* and *terminal symbols* respectively, $S \in N$ is the *start symbol* and P is a finite set of *production rules*, each of the form:

$$(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*.$$

G *derives* (or equivalently *produces*) in one step a string y from a string x , denoted $x \Rightarrow y$, iff there are $u, v, p, q \in (\Sigma \cup N)^*$ such that $x = upv$, $p \rightarrow q \in P$ and $y = uqv$. We write $x \Rightarrow^* y$ if y can be derived in zero or more steps from x , i.e., \Rightarrow^* denotes the reflexive and transitive closure of the relation \Rightarrow . The language of G , denoted as $\mathcal{L}(G)$, is the set of all strings in Σ^* that can be derived in a finite number of steps from the start symbol S . In symbols,

$$\mathcal{L}(G) = \{w \in \Sigma^* \mid S \rightarrow^* w\}$$

In this work, Σ always denotes the “input” alphabet (e.g., the set of ASCII characters) of a given executable (binary) program p . The set of *valid inputs*

of p is defined as the subset of Σ^* formed by all well formed inputs for p . In symbols:

$$\text{validInputs}(p) = \{w \in \Sigma^* \mid w \text{ is a well formed input for } p\}$$

The definition of well formed input for a given program p depends on the application at hand. Here we only need to assume that it is possible to determine whether a given input string w is well formed or not for a program p by simply running p with input w .

As usual in this setting, we assume that $\text{validInputs}(p)$ is a context-free language. Consequently, there must be a context-free grammar G_p such that $\mathcal{L}(G_p) = \text{validInputs}(p)$. Recall that a grammar is context-free if its production rules are of the form $A \rightarrow \alpha$ with A a single nonterminal symbol and α a possibly empty string of terminals and/or nonterminals.

Let s_p be the source code of a binary program p , we denote as $T(s_p)$ the standard AST model of s_p that complies with the AST metamodel specifications of the OMG¹. In this paper we present an algorithm that, given p and $T(s_p)$ as input, returns a grammar G_p . Ideally, $\mathcal{L}(G_p)$ should coincide with $\text{validInputs}(p)$. However, this is in general an undecidable problem [5]. A good alternative is to measure how well (or bad) $\mathcal{L}(G_p)$ approximates the input language of p in terms of precision and recall measures based in probability distributions over those languages, as proposed in [2].

The probability distribution of a language is calculated in [2] by using random sampling of strings from the corresponding grammar. We follow here the same approach. Let $G = (N, \Sigma, P, S)$ be a context-free grammar. As a first step, G is converted to a *probabilistic context-free grammar* by assigning a *discrete distribution* \mathcal{D}_A to each nonterminal $A \in N$. As usual, \mathcal{D}_A is of size $|P_A|$, where P_A is the subset of productions in P of the form $A \rightarrow \alpha$. Here, we assume that \mathcal{D}_A is *uniform*. We can then *randomly sample a string x from the language* $\mathcal{L}(G, A) = \{w_i \in \Sigma \mid A \rightarrow^* w_i\}$, denoted $x \sim \mathcal{P}_{\mathcal{L}(G, A)}$, as follows:

- Using \mathcal{D}_A select randomly a production $A \rightarrow A_1 \cdots A_k \in P_A$.
- For $i = 1, \dots, k$, recursively sample $x_i \sim \mathcal{P}_{\mathcal{L}(G, A_i)}$ if $A_i \in N$; otherwise let $x_i = A_i$.
- Return $x = x_1 \cdots x_k$.

The *probability distribution* $\mathcal{P}_{\mathcal{L}(G)}$ of the language $\mathcal{L}(G)$ is simply defined as the probability $\mathcal{P}_{\mathcal{L}(G, S)}$ induced by sampling strings in the probabilistic version of G defined above.

Again following [2], we now measure the quality of a learned (or induced) language \mathcal{L}' with respect to the target language \mathcal{L} in terms of precision and recall.

- The *precision* of \mathcal{L}' w.r.t. \mathcal{L} , denoted $\text{precision}(\mathcal{L}', \mathcal{L})$, is defined as the probability that a randomly sampled string $w \sim \mathcal{P}_{\mathcal{L}'}$ belongs to \mathcal{L} . In symbols, $\Pr_{w \sim \mathcal{P}_{\mathcal{L}'}}[w \in \mathcal{L}]$.

¹ <https://www.omg.org/spec/ASTM/1.0/>.

- Conversely, the *recall* of \mathcal{L}' w.r.t. \mathcal{L} , denoted $\text{recall}(\mathcal{L}', \mathcal{L})$, is defined as $\Pr_{w \sim \mathcal{P}_{\mathcal{L}}}[w \in \mathcal{L}']$.

In order to be considered a good approximation to \mathcal{L} , the learned language \mathcal{L}' needs to have both, high precision and high recall. Note that, a language $\mathcal{L}' = \{w\}$, where $w \in \mathcal{L}$, has perfect precision, but most likely has also very low recall. On the opposite end, $\mathcal{L}' = \Sigma^*$ has perfect recall, but most likely has low precision.

3 Seed Grammar Extraction

To be able to run our grammar refinement algorithm we first need to extract a seed grammar. In an optimal case, our seed grammar should have recall=1 regardless of precision. This means that every word in our target language is produced by our seed grammar. In addition, the seed grammar should contain information on the basic structure of the target grammar (e.g. Nonterminals, number of rules etc.). To extract a seed grammar we take the following steps:

1. Learn tokens of a program p .
2. Extract the basic structure of the seed grammar from the AST model $T(s_p)$.
3. Identify potential token positions.
4. Expand the seed grammar for all possible token combinations.

Token Learning: To be able to efficiently run our proposed algorithm we want to reduce the seed grammar to an absolute minimum. This can be achieved by reducing the amount of terminal symbols by summarizing them into tokens. For example a rule $S \rightarrow 1 \mid 2 \mid 3$ can be summarized to $S \rightarrow d$.

For our experiments we used a naive approach for token learning from a black-box given a set of known and valid words S of an unknown Language $L(G)$ as well as the alphabet Σ of G . We start by extracting potential token sets for each word $s \in S$. We do this by going through our alphabet and check if $a \in \Sigma$ is a substring of s . If yes we replace a in s with every other $a' \in \Sigma$ and then execute these new inputs. If the input is accepted, we add a' to the potential token set T_{as} of word s for symbol a . Note that T_{as} always contains a . If we continue to do this, we get a set of potential tokens for G . At the end we then check the set of potential tokens if one token T_{as} is a subset of another token $T_{a's'}$. If $T_{as} \subset T_{a's'}$, we remove $T_{a's'}$ from the potential tokens and add $T_{a''s''} = T_{a's'} \setminus T_{as}$. Finally we can be sure that every token left only contains symbols of A which are interchangeable in the context of the known words S .

Lets take a look at the parser source code given in listing 1.1 with alphabet $\Sigma = \{1, 2, 3, +, -, /, *, (,)\}$ and words $S = \{1 + 1/1, 1 + 1, (1)\}$. Following our simple approach for token learning we are able to extract the following tokens: $[1, 2, 3]$, $[+, -]$, $[*, /]$, $[(,)]$. If we consider these tokens when constructing our seed grammar, we are able to reduce it from 454 rules to 170 rules in Chomsky normal form.

Extract Basic Structure: Next we take a close look at the AST of a given program p . We start by defining a nonterminal n for every function in p . Then we search for function calls to get insight on the control flow of p . With this information we are able to extract a basic structure for our seed grammar following some simple heuristics:

```

1 public class ExprParser {
2     public static int parse(char[] in) {
3         int pos = expr(0, in);
4         if (pos < in.length) {
5             syntaxError("End of input", pos);
6             pos = -1;
7         }
8         return pos;
9     }
10    public static int expr(int pos, char[] in) {
11        pos = term(pos, in);
12        if (pos == -1) return pos;
13        if (pos == in.length) return pos;
14        if (in[pos] == '+') {pos++; pos = term(pos, in);}
15        else if (in[pos] == '-') {pos++; pos = term(pos, in);}
16        return pos;
17    }
18    public static int term(int pos, char[] in) {
19        pos = factor(pos, in);
20        if (pos == -1) return pos;
21        if (pos == in.length) return pos;
22        if (in[pos] == '*') {pos++; pos = factor(pos, in);}
23        else if (in[pos] == '/') {pos++; pos = factor(pos, in);}
24        return pos;
25    }
26    public static int factor(int pos, char[] in) {
27        if (pos == in.length) return -1;
28        if (in[pos] == '1') return pos + 1;
29        if (in[pos] == '2') return pos + 1;
30        if (in[pos] == '3') return pos + 1;
31        if (in[pos] == '(') {
32            pos++; pos = expr(pos, in);
33            if (pos == -1) return pos;
34            if (pos == in.length) return -1;
35            if (in[pos] == ')') {pos++; return pos;}
36            else {syntaxError("invalid factor ", pos);}
37        }
38        return -1;
39    }
40    public static void syntaxError(String msg, int pos) {
41        final boolean log = false;
42        if (log) {
43            System.out.print("Syntax error ");
44            System.out.print(msg);
45            System.out.print(" col: ");
46            System.out.println(pos);}
47    }
48 }

```

Listing 1.1. Java code.

- Add one rule to the seed grammar for every control flow path in the function where the left-hand side is the function and the right-hand side a sequence of functions which are called.
- If there exists a path without a function call, add ϵ .
- For every function in p , add potentially parsed terminals or tokens.

In the following example we assume that we know which tokens are parsed by which function of p . This gives us a more comprehensible example by reducing the amount of rules. For the example program given in Listing 1.1 this would lead to the following structure:

parse \rightarrow expr	ϵ
expr \rightarrow term term term	$\{+, -\}, \epsilon$
term \rightarrow factor factor factor	$\{*, /\}, \epsilon$
factor \rightarrow expr ϵ	$\{1, 2, 3\}, (,), \epsilon$

Identification of Potential Token Positions: As a next step we identify positions for potential tokens in our grammar. We do this by adding a position marker before and after every nonterminal in our seed grammar, except it is the first or last statement in $T(s_p)$. E.g. the first statement in the function `parse` is a call to `expr`. As the call to `expr` is the first statement we know that it is not possible that there are some checks for tokens before that. If we apply this method we get the following seed grammar, where \bigcirc denotes a potential token position:

$$\begin{aligned}
 \text{parse} &\rightarrow \text{expr} \bigcirc && \epsilon \\
 \text{expr} &\rightarrow \text{term} \bigcirc \mid \text{term} \bigcirc \text{term} && \{+, -\}, \epsilon \\
 \text{term} &\rightarrow \text{factor} \bigcirc \mid \text{factor} \bigcirc \text{factor} && \{*, /\}, \epsilon \\
 \text{factor} &\rightarrow \bigcirc \text{expr} \bigcirc \mid \bigcirc && \{1, 2, 3\}, (,), \epsilon
 \end{aligned}$$

Seed Grammar Expansion: As a final step we expand the grammar by adding a new rule for every token or terminal of p for every \bigcirc . The final seed grammar will look like this²:

$$\begin{aligned}
 \text{parse} &\rightarrow \text{expr} \\
 \text{expr} &\rightarrow \text{term} [+,-] \mid \text{term} [+,-] \text{term} \\
 \text{term} &\rightarrow \text{factor} [*,/] \mid \text{factor} [*,/] \text{factor} \\
 \text{factor} &\rightarrow [1,2,3] \mid (\mid [1,2,3] \text{expr} [1,2,3] \mid [1,2,3] \text{expr} (\mid [1,2,3] \text{expr}) \\
 \text{factor} &\rightarrow (\text{expr} [1,2,3] \mid (\text{expr} (\mid (\text{expr}) \\
 \text{factor} &\rightarrow) \text{expr} [1,2,3] \mid) \text{expr} (\mid) \text{expr})
 \end{aligned}$$

At this point we have to note that our approach to seed grammar extraction is limited on how a parser is implemented. Imagine a rule $S \rightarrow (S)$. This rule can be implemented in two ways: By means of recursive calls or by counting and remembering the amount of opening brackets and then checking if the same amount of closing brackets is present. As the latter case does not make use of a call to itself, we are not able to extract a correct seed grammar. This means that our approach to seed grammar extraction is limited to recursive descending parsers. However our approach to grammar refinement is not limited to such parsers if a correct seed grammar can be provided by other means.

4 Grammar Refinement Algorithm

The central grammar refinement task in our approach is described in Algorithm 1.

We illustrate how the algorithm works by means of a simple example. Let us assume an executable program p with valid inputs $\text{validInputs}(p) = \mathcal{L}(G_p)$, where G_p is the context-free grammar with start symbol S , set of terminal and non terminal symbols $\Sigma = \{+, *\}$ and $N = \{S\}$, respectively, and production rules:

$$S \rightarrow +* \mid +S*$$

² The expression $[\dots]$ is used to denote optional parts in our grammar.

Algorithm 1. Grammar Refinement Algorithm**Input:** Seed context-free grammar G and program p .**Output:** Refined grammar G' in Chomsky normal form.**Ensure:** $\text{precision}(\mathcal{L}(G'), \text{validInputs}(p)) \geq \text{precision}(\mathcal{L}(G), \text{validInputs}(p))$.

```

1:  $G' \leftarrow G.\text{to\_normal\_form}()$ 
2:  $\text{countEmpty} \leftarrow 0$ 
3:  $\text{length} \leftarrow 1$ 
4: while  $G'.\text{getWords}(\text{length}) = \emptyset$  do
5:    $\text{countEmpty} \leftarrow \text{countEmpty} + 1$ ;
6:    $\text{length} \leftarrow \text{length} + 1$ ;
7: end while
8:  $\text{refinedProductions} \leftarrow G'.\text{productions}()$ ;
9:  $\text{safeProductions} \leftarrow \{A \rightarrow t \in \text{refinedProductions} \mid A \notin G.\text{non\_terminals}()\}$ ;
10:  $\text{seenProductions} \leftarrow \emptyset$ ;
11:  $l \leftarrow 1$ 
12: while  $(\text{refinedProductions} \cap \text{safeProductions}) \neq \text{refinedProductions}$  do
13:   repeat
14:      $\text{words} \leftarrow G'.\text{getWords}(\text{length})$ ;
15:      $\text{deleteCandidates} \leftarrow \emptyset$ ;
16:     for  $w \in \text{words}$  do
17:        $\text{parseTree} \leftarrow G'.\text{getParseTree}(w)$ ;
18:        $\text{seenProductions} \leftarrow \text{seenProductions} \cup \text{parseTree}.\text{getProductions}()$ ;
19:        $\text{evalProductions} \leftarrow \text{parseTree}.\text{getProductionsUpToLevel}(l)$ ;
20:       if  $w \in \text{validInputs}(p)$  then
21:          $\text{safeProductions} \leftarrow \text{safeProductions} \cup \text{evalProductions}$ ;
22:       else
23:          $\text{deleteCandidates} \leftarrow \text{deleteCandidates} \cup \text{evalProductions}$ ;
24:       end if
25:     end for
26:      $\text{deleteProductions} \leftarrow \text{deleteCandidates} \setminus \text{safeProductions}$ ;
27:     if  $\text{deleteProductions} \neq \emptyset$  then
28:        $\text{refinedProductions} \leftarrow \text{refinedProductions} \setminus \text{deleteProductions}$ ;
29:        $\text{refinedProductions} \leftarrow \text{reachable}(\text{refinedProductions})$ ;
30:        $G'.\text{update}(\text{refinedProductions})$ ;
31:     end if
32:   until  $\text{deleteProductions} = \emptyset$ 
33:   if  $\text{refinedProductions} \subseteq \text{seenProductions}$  then
34:      $l \leftarrow \text{length}$ 
35:   else
36:     if  $\text{length} - \text{countEmpty} > 1 \wedge \text{isInteger}(\log_2(\text{length} - \text{countEmpty}))$  then
37:        $l \leftarrow \log_2(\text{length} - \text{countEmpty})$ 
38:     end if
39:   end if
40:    $\text{length} \leftarrow \text{length} + 1$ ;
41: end while

```

Let us further assume that the input seed grammar G in Algorithm 1 has, respectively, the same sets Σ and N of terminal and non terminal symbols than G_p , and that it has also S as its start symbol. The set of seed productions of G , i.e., the set of productions that we want to refine with our algorithm to increase the precision of G , is formed by the following rules:

$$S \rightarrow + \mid * \mid ++ \mid +* \mid ** \mid +S \mid *S \mid S+ \mid S* \mid +S+ \mid +S* \mid *S+ \mid *S*$$

The first step in the algorithm (line 1) transforms the grammar G into an equivalent Chomsky normal form grammar $G' = (N', \Sigma, P', S)$, where $N' = \{S, N_+, N_*, A_1, A_2, A_3, A_4\}$ and P' is formed by the following productions:

$$\begin{aligned} S &\rightarrow + \mid * \mid N_+N_+ \mid N_+N_* \mid N_*N_+ \mid N_*N_* \\ S &\rightarrow N_+S \mid N_*S \mid SN_+ \mid SN_* \mid N_+A_1 \mid N_+A_2 \mid N_*A_1 \mid N_*A_2 \\ A_1 &\rightarrow SN_* \\ A_2 &\rightarrow SN_+ \\ N_+ &\rightarrow + \\ N_* &\rightarrow * \end{aligned}$$

Lines 2–11 of the algorithm simply initialise the required variables. Note that $\text{countEmpty} + 1$ is the minimum word length among all words in $\mathcal{L}(G')$. This means that in our example we need to start by inspecting words of $\text{length} = \text{countEmpty} + 1 = 1$. Further, we assign to *refinedProductions* the set of production in G' , and initialize the sets of *safeProductions* and *seenProductions*. In *safeProductions* we initially include those productions in G' of the form $A \rightarrow t$, where A is a non terminal symbol added by the transformation to Chomsky normal form and $t \in \Sigma$. These rules are then safe from elimination during the whole refinement process, unless they become unreachable from S . Thus, after executing these steps, we get the following values:

$$\begin{aligned} \text{length} &= 1 \\ \text{refinedProductions} &= P' \\ \text{safeProductions} &= \{N_+ \rightarrow +, N_* \rightarrow *\} \\ \text{seenProductions} &= \emptyset \end{aligned}$$

At this point the algorithm is ready to start the refinement process, i.e., to step-by-step eliminate rules from the grammar that are responsible for producing words which do not belong to the input language of p . This has to be done as careful as possible, trying not to adversely affect the recovery rate of the candidate grammar. We have found out during the experiment reported in the paper, that a good rule of thumb is to only eliminate rules used in the corresponding parsing tree up to level denoted by the parameter l in the algorithm. Notice that to set the parameter l we use the fact that the minimum possible depth of the parse tree of a word of length n (if the grammar in Chomsky normal form) is $\lceil \log_2 n \rceil + 1$. A more aggressive approach to refinement, i.e., an approach that

increases the value of l faster, has the advantage of reducing the computation time considerably. However, as witnessed by our experiments, if the approach is too aggressive in this regard, then it considerably reduces the recall rate of the obtained grammar. The while-loop in Line 12 defines the termination condition, which is met when there is no further rule among *refinedProductions* that can be disposed of. If the termination condition has not been met, then the algorithm continues by repeating the procedure defined in Lines 13–32 until no production can be further deleted from the set of refined productions by examination of parse trees of words of the current length.

This repeat-until loop accounts for the fact that the seed grammar will most probably be ambiguous, i.e., there will be words in its language which can be derived using different parse trees. Since the number of parse trees grows too fast, we cannot look at all parse trees for each inspected word. Instead, we just take the first parse tree returned by the parsing algorithm (in our case CYK). The intuitive idea is that by repeating the process for the same length of word after each refinement of the grammar, we can still examine the relevant parse tree that were missed in the previous round.

Following with our example, at this point line 14 assigns to the variable *words*, all words of length 1 that can be produced by G' (i.e., $+$ and $*$). Clearly, after the for loop in Lines 16–25 is executed, the variable *deleteCandidates* contains the set $\{S \rightarrow +, S \rightarrow *\}$ of productions. Since none of these productions is among those in *safeProductions*, both are deleted from *refinedProductions* and the set of productions of the grammar G' is updated accordingly (Lines 28–30). Since the updated grammar G' does no longer produces any word of length 1, no refinement take place in the next iteration of the repeat-until loop. Then the value of *length* is incremented by 1 and the process repeats itself, with the following outcomes for each subsequent step:

- For *length* = 2,
 - G' produces the words: $+, *, ++$ and $**$.
 - Since $+, ++, ** \notin \text{validInputs}(p)$ and $l = 1$, *deleteCandidates* takes the value $\{S \rightarrow N_+N_+, S \rightarrow N_+N_+, S \rightarrow N_*N_*\}$.
 - Since $++ \in \text{validInputs}(p)$, $S \rightarrow N_+N_*$ is added to *safeProductions*.
 - Thus *refinedProductions* reduces to

$$\begin{aligned} S &\rightarrow N_+N_* \mid N_+S \mid N_*S \mid SN_+ \mid SN_* \mid N_+A_1 \mid N_+A_2 \mid N_*A_1 \mid N_*A_2 \\ A_1 &\rightarrow SN_* \quad A_2 \rightarrow SN_+ \quad N_+ \rightarrow + \quad N_* \rightarrow * \end{aligned}$$

- For *length* = 3,
 - G' produces the words: $++*, *+*, ++ +$ and $+++$.
 - Since none of the word of length 3 produced by G' is in *validInputs*(p) and $l = 1$, then *deleteCandidates* takes the value $\{S \rightarrow N_+S, S \rightarrow N_*S, S \rightarrow SN_+, S \rightarrow SN_*\}$ and *safeProductions* remains unchanged.
 - At the end of this step, *refinedProductions* further reduces to

$$\begin{aligned} S &\rightarrow N_+N_* \mid N_+A_1 \mid N_+A_2 \mid N_*A_1 \mid N_*A_2 \\ A_1 &\rightarrow SN_* \quad A_2 \rightarrow SN_+ \quad N_+ \rightarrow + \quad N_* \rightarrow * \end{aligned}$$

- For $length = 4$,
 - G' produces the words: $++**$, $++*+$, $*+**$ and $*+*+$ with the corresponding parse trees shown in Fig. 1.
 - Since $++*+, *+*+, *+** \notin validInputs(p)$ and $l = 1$, *deleteCandidates* takes the value $\{S \rightarrow N_+A_2, S \rightarrow N_*A_1, S \rightarrow N_*A_2\}$.
 - Since $++** \in validInputs(p)$, $S \rightarrow N_+A_1$ is added to *safeProductions*.
 - Thus, the content of *refinedProductions* further reduces to (notice that $A_2 \rightarrow SN_+$ is no longer reachable and therefore also eliminated)

$$S \rightarrow N_+N_* \mid N_+A_1 \quad A_1 \rightarrow SN_* \quad N_+ \rightarrow + \quad N_* \rightarrow *$$

- For $length = 5$, The value of l is set to 2 and nothing else changes since the grammar obtained in the previous step does not produce words of length 5.
- Finally, for $length = 5$, the grammar only produces words that belong to *validInputs(p)* and, since $l = 2$, the rule $A_1 \rightarrow SN_*$ is added to the *safeProductions*. By now all productions in *refinedProductions* belong to *safeProductions* and thus the termination condition is met.

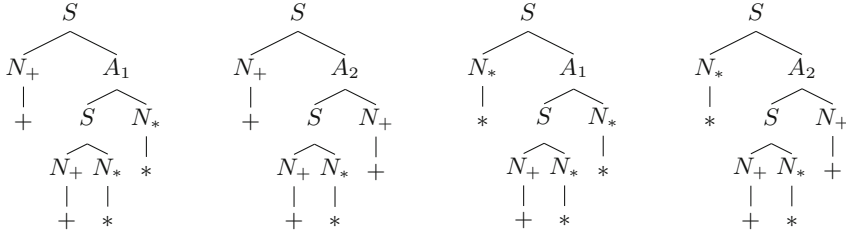


Fig. 1. Parse trees

Notice that the language of G' returned by the algorithm corresponds exactly to *validInputs(p)*. Of course, given the heuristic nature of our algorithm, this satisfactory result cannot be guaranteed. Even when we are certain that the language recognized by the seed grammar G includes the input language of p , an exact solution would need to take into account all possible derivations trees for each word generated by the grammar, which would make the approach unfeasible. A non ambiguous seed grammar would solve this problem, but this cannot be realistically ensured for the targeted use cases of our algorithm.

5 Running Time Analysis

Since G' is in Chomsky normal form (see Line 1 in Algorithm 1), we get that every parsing tree of G' is a binary tree. Therefore, the main while loop from Lines 12–41 in Algorithm 1 will be repeated at most $p = 2^m$ times, where $m = |N'|$ is the number of non-terminals in the initial G' . Notice that by the pumping lemma for context-free languages, every word z in $\mathcal{L}(G)$ such that $|z| \geq p$ can be written as $z = uvwxy$ with the following conditions:

- $|vwx| \leq p$
- $vx \neq \lambda$
- For $i \geq 0$, the word uv^iwx^iy is also in $\mathcal{L}(G)$.

Therefore, when *length* in the algorithm reaches the value p , then $l = \text{length}$ and all remaining productions are added to *seenProductions*. Thus each production in the initial G' is at this point no longer in *refinedProductions* or is in *safeProductions*, and the termination condition is met.

Our algorithm examines in the first step all words of length 1 produced by the grammar, then all words of length 2 produced by the refined grammar in the previous step, and so on. As seen before, this can go on up to words of length p . If there are $|\Sigma| = n$ terminals in the grammar G' , then the number of different words that we will have to examine is bounded by $\mathcal{O}(n^p)$, i.e., $\mathcal{O}(n^{2^m})$. This complexity bound, makes our algorithm intractable in theory. In practice however, the values for the exponent p are in our experience usually somewhere between 3 and 8, since it corresponds to the nesting depth of the production rules in the grammar. The higher this value however, the more important it is to keep the number of symbols in Σ in check, using tokens to represent sets of symbols. For instance, replacing symbols $0, 1, \dots, 9$ in Σ by a token d .

As shown in our evaluation section, another point that allows the algorithm to perform reasonably well in practice, is the fact that each time we eliminate a production rule from the grammar, the number of words that needs to be checked in the next round reduces considerably.

Regarding the production of the words ordered by length for a given grammar $G = (N, \Sigma, P, S)$ in Chomsky normal form, we notice that one can do that quite efficiently. First, we fix a total order $<$ of $N \cup \Sigma$, where the symbols in N precede those in Σ . Then using a priority queue Q defined in terms of $<$, the words belonging to $\mathcal{L}(G)$ can be generated without repetitions and ordered by length then lexicographical as follows:

1. Add S to Q .
2. Remove the first element w from Q (i.e., the one with higher priority).
3. If w contains only terminals, output w .
4. If w contains a non-terminal, for each production α for the first non-terminal in w , append the results of expanding α to Q .
5. Repeat step 2–4 until either Q is empty, or all the required words have been produced.

Finally, we note that for each word w generated by the previous procedure, we can in parallel keep an associated parsing tree. Thus, it is relatively inexpensive in terms of computational time to produce the necessary parsing tree. In our prototype implementation of the algorithm, we use instead a less efficient approach consisting on applying the well known CYK parsing algorithm for retrieving the parsing tree for each examined word. Even this less efficient approach, still works reasonably well in practice, as shown by our experiments.

Table 1. Precision improvement over time, final recall and running time

	$\mathcal{L}(G_p)$	MathExpr	MathExpr2	Mail	JSON	JSON opt.
Seed G.	0.0702	0.129	0.058	0.0025	0.3841	0.3841
Word length 1	0.1517	0.3792	0.3751	0.0025	0.3841	0.3841
Word length 2	0.5904	0.5914	0.5817	0.0025	0.3841	0.3841
Word length 3	0.6401	0.6486	0.584	0.0025	0.3841	0.3841
Word length 4	1.0	0.8974	0.5919	0.0025	0.3841	0.3841
Word length 5		1.0	1.0	0.0025	0.3841	0.3841
Word length 6				0.0025	0.3841	0.3841
Word length 7				0.0025	0.3841	0.3841
Word length 8				0.0032	0.3841	0.3841
Word length 9				0.0107	0.3841	0.3841
Word length 10				0.1808	0.3841	0.3841
Word length 11				0.1808	0.9	0.8983
Word length 12				0.1808	0.9	0.8983
Word length 13				0.1808	0.9	0.8983
Word length 14				0.59	0.9148	0.9038
Word length 15				0.59	0.964	0.9142
Word length 16				1.0	0.964	0.9142
Word length 17					1.0	0.9142
Word length 18						0.9142
Word length 19						1.0
Final Recall	1.0	1.0	1.0	1.0	0.9265	0.9604
Running time	0.4 s	2.7 s	2.6 s	34.7 s	30 m 18.8 s	260 m 2.2 s

6 Experiments and Evaluation

We conducted a series of test runs to evaluate the performance in terms of precision and recall of Algorithm 1 for grammar learning through refinement. The precision and recall was measured as described in the preliminaries, random sampling 10,000 words from each grammar. The results of those experiments are summarised in Table 1. Each column represents a different input language which we detail next. The first line with label “Seed G.” shows the initial precision of each of the seed grammars w.r.t. its corresponding target language. Each subsequent line labeled as “Word length X ” shows the evolution of the precision value after the algorithm has refined the grammar based on the examination of words of length X . The second-to-last line shows the recall of each of the refined grammars at the end of the process. At the beginning of the process, all the seed grammars have perfect recall 1. The final line is self explanatory and shows the running time for each of the experiments.

First, we should note that the running times could be improved substantially, by simply making use of the fact that the examination of words of a given length can clearly be done in parallel. Furthermore, these running times correspond to a prototype implementation of the algorithm in Python, using an educational library for formal language manipulation (Pyformlang) [15], which is simple to use but not the most efficient for the task. The experiments were run in a Linux installation in a modest virtual machine with 16 GB of RAM and an Intel(R) Core(TM) i7-8665U CPU at 1.90 GHz. In any case, even the worst case running time shown in the table is still somehow acceptable for the envisaged applications of our algorithm.

The column labeled as “ $\mathcal{L}(G_p)$ ” shows the results corresponding to the execution of the algorithm with $\text{validInputs}(p) = \mathcal{L}(G_p)$, where G_p is the simple grammar used as example in Sect. 4. The seed grammar provided as input for this experiment corresponds to the seed grammar G , also given as example in Sect. 4. With this simple example, it only takes our implementation 0.4s to return a grammar with perfect precision and recall.

The columns labeled as “MathExpr” and “MathExpr2” show the results corresponding to the execution of the algorithm with $\text{validInputs}(p) = \mathcal{L}(G)$, where G is the following grammar for mathematical expressions:

$$S \rightarrow S + S \mid S - S \mid S * S \mid S / S \mid (S) \mid d$$

In the first case, i.e. in the MathExpr experiment, the seed grammar given as input was formed by all 100 rules obtained from the expansion (as described in detail in Sect. 3) of the following skeletal rules:

$$\begin{array}{ll} S \rightarrow \bigcirc T \bigcirc T \bigcirc \mid \bigcirc T \bigcirc & +, -, \epsilon \\ T \rightarrow \bigcirc F \bigcirc F \bigcirc \mid \bigcirc F \bigcirc & *, / , \epsilon \\ F \rightarrow \bigcirc S \bigcirc \mid \bigcirc & d, (,), \epsilon \end{array}$$

In turn, in the MathExpr2 experiment we used an alternative seed grammar with 630 initial rules. This corresponds to the expansion of the following skeletal rules:

$$S \rightarrow \bigcirc S \bigcirc S \bigcirc \mid \bigcirc S \bigcirc \mid \bigcirc \quad +, -, *, / , (,), d, \epsilon$$

In both cases, the algorithm returned a grammar with perfect recall and precision in less than 3s. In the case of MathExpr2, this is rather surprising given the high number of initial rules in the grammar. This is explained by the fact that the nesting depth of the rules in the seed grammar used in MathExpr2 is 0, while the corresponding nesting depth of the rules in the seed grammar used in MathExpr is 3.

In the experiment labeled “Mail”, we used $validInputs(p) = \mathcal{L}(G)$, where G is the grammar that recognizes valid e-mail addresses defined by the following productions:

$$\begin{aligned}
 S &\rightarrow Body @ Head . Tag \\
 Body &\rightarrow Char Chars \\
 Head &\rightarrow Char Chars \\
 Tag &\rightarrow Char Char \mid Char Char Char \\
 Chars &\rightarrow Char Chars \mid \epsilon \\
 Char &\rightarrow t
 \end{aligned}$$

being t a token for $\{a, b, \dots, z\}$.

The corresponding seed grammar used in this experiment has the following skeletal rules, which generated a total of 96 actual rules:

$$\begin{aligned}
 S &\rightarrow \bigcirc Body \bigcirc Head \bigcirc Tag \bigcirc && @, ., \epsilon \\
 Body &\rightarrow Char Chars \\
 Head &\rightarrow Char Chars \\
 Tag &\rightarrow Char Char \mid Char Char Char \\
 Chars &\rightarrow \bigcirc Char \bigcirc Chars \bigcirc \mid \bigcirc && \epsilon \\
 Char &\rightarrow \bigcirc && t
 \end{aligned}$$

Since our approach does not work well when the empty symbol is part of the seed grammar, mostly due to the fact that it is based in inspecting increasing word lengths, we treat ϵ in the seed grammar as a non-empty standard symbol. Thus, the resulting seed grammar contains rules such as $Chars \rightarrow \epsilon Char Chars$, where ϵ is simply a non-terminal symbols of length 1. After the learning process is concluded, we simply eliminate all ϵ symbols from the refined grammar except for those which appear in rules of the form $A \rightarrow \epsilon$. This trick works well in our experiments. Indeed, in this case we were able to again learn a grammar with precision 1 and without degrading the perfect initial recall. The increase in running time w.r.t. the previous experiments is due mostly to the fact that the algorithm can only start to eliminate rules from the seed grammar after it has considered words of length at least 8. Notice that this seed grammar does not produce any word of length smaller than 8, and that the number of words that need to be examined grows exponentially on the length, unless some rules are eliminated from the grammar.

Our final experiment concerns learning a grammar that can recognize well formed JSON files. The set $validInputs(p)$ is defined as the language recognized by the JSON grammar with the following rules.

$$\begin{aligned}
S &\rightarrow Element \\
Element &\rightarrow Ws \ Value \ Ws \\
Value &\rightarrow Object \mid String \mid Number \mid Array \mid false \mid true \mid null \\
Object &\rightarrow \{ \ Ws \} \mid \{ \ Members \} \\
String &\rightarrow \text{“ Characters ”} \\
Number &\rightarrow Digit \mid Digit \ Number \\
Digit &\rightarrow d \quad \text{being } d \text{ a token for } \{0, \dots, 9\} \\
Array &\rightarrow [\ Elements \] \mid [\ Ws \] \\
Members &\rightarrow Member \mid Member \ , \ Members \\
Member &\rightarrow Ws \ String \ Ws : Element \\
Elements &\rightarrow Element \mid Element \ , \ Elements \\
Characters &\rightarrow Character \ Characters \mid \epsilon \\
Character &\rightarrow t \quad \text{being } t \text{ a token for } \{a, \dots, z\} \\
Ws &\rightarrow _ \mid \epsilon
\end{aligned}$$

In this case the seed grammar has the following skeletal rules and 70 actual rules:

$$\begin{aligned}
S &\rightarrow Element \\
Element &\rightarrow Ws \ Value \ Ws \\
Value &\rightarrow \bigcirc \ Object \ \bigcirc \mid \bigcirc \ String \ \bigcirc \quad false, true, null, \epsilon \\
Value &\rightarrow \bigcirc \mid \bigcirc \ Number \ \bigcirc \mid \bigcirc \ Array \ \bigcirc \mid \bigcirc \quad false, true, null, \epsilon \\
Object &\rightarrow \bigcirc \ Ws \ \bigcirc \mid \bigcirc \ Members \ \bigcirc \quad \{, \}, \epsilon \\
String &\rightarrow \bigcirc \ Characters \ \bigcirc \quad \text{“ , ”}, \epsilon \\
Number &\rightarrow Digit \mid Digit \ Number \\
Digit &\rightarrow d \\
Array &\rightarrow \bigcirc \ Elements \ \bigcirc \mid \bigcirc \ Ws \ \bigcirc \quad [,], \epsilon \\
Members &\rightarrow \bigcirc \ Member \ \bigcirc \mid \bigcirc \ Member \ \bigcirc \ Members \quad , ; \epsilon \\
Member &\rightarrow Ws \ \bigcirc \ String \ \bigcirc \ Ws \ \bigcirc \ Element \quad :, \epsilon \\
Elements &\rightarrow Element \mid Element \ , \ Elements \\
Character &\rightarrow a \dots z \\
Characters &\rightarrow Character \ \bigcirc \ Characters \mid \bigcirc \quad \epsilon \\
Ws &\rightarrow \bigcirc \quad _, \epsilon
\end{aligned}$$

As before, ϵ is considered in the last two rules not only as the empty symbol, but also as a terminal symbol of length 1.

The experiment with JSON is divided in two parts and the results shown in the last two columns of Table 1. In the column labeled “JSON” we show the result of the standard run of our algorithm. The seed grammar starts to produce words that allow the algorithm to increase the precision of the refined grammar after it has examined words of length 11. At that point the grammar is able to produce 9432 words which are used to eliminate most of the incorrect productions with *Value* in its left-hand side. Unfortunately, at this point the algorithm also eliminates a production, namely $Value \rightarrow \{\#CNF\# C\#CNF\#33$, which decreases the recall of the refined grammar from 1 to 0.9265. Note that this rule belongs to the Chomsky normal form version of the seed grammar described above. Given the high number of words examined of length 11 (after this step less than 5000 in total are produced by the grammar for lengths 12 – 17) and our inefficient prototype, the running time for learning these grammar increases considerably w.r.t. the previous experiments. Nevertheless, this is still acceptable for the use cases described in this paper.

Finally, in the column labeled “JSON opt.” we show the results of running the same JSON experiment, but including the rule $Value \rightarrow \{\#CNF\# C\#CNF\#33$ among the set of safe from the start productions (see *safeProductions* in Algorithm 1). This allowed us to increase the recall of the resulting grammar to 0.9604 without decreasing its precision. The only downside is a considerable increase in the running time, from half an hour to more than 4 hours. Again, this shows that the running time is highly influenced by the number of words of each given length that the algorithm needs to evaluate.

7 Conclusion

The main contribution of this paper is a novel algorithm for program input grammar learning. Different to previous work, our algorithm starts from a seed grammar with high recall and low precision. Then, through a process of step-by-step refinements of the seed grammar, which considers sets of words of increasing length, our heuristic algorithm tries to increase the precision of the grammar without decreasing its initial recall. While the theoretical complexity of the algorithm is high in theory, we show through a running time analysis and also through empirical experiments that the algorithm can be successfully used in practice for learning program input grammars of reasonable large size such as JSON. The initial results obtained in this paper are indeed very encouraging, achieving in all evaluated cases a perfect precision as well as a perfect or close to perfect recall.

As future work, we plan to study possible improvements to our grammar refinement algorithm, considering for instance how the measures of recall and precision can be included in the refinement process to improve the applied heuristics. Our last experiment with the JSON language offers a hint in this direction.

In this paper we only touched the problem of how to come up with appropriate seed grammars for our novel algorithm. Given the positive results obtained so far, we believe this area also warrants future research. Once these hurdles

have been cleared, we plan to perform a though and fair comparison with state of the art grammar miners such as autogram [11] and its successor mimid [6].

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. *Inf. Comput.* **75**(2), 87–106 (1987). [https://doi.org/10.1016/0890-5401\(87\)90052-6](https://doi.org/10.1016/0890-5401(87)90052-6)
2. Bastani, O., Sharma, R., Aiken, A., Liang, P.: Synthesizing program input grammars. In: Cohen, A., Vechev, M.T. (eds.) *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*, pp. 95–110. ACM (2017). <https://doi.org/10.1145/3062341.3062349>
3. Eberlein, M., Noller, Y., Vogel, T., Grunske, L.: Evolutionary grammar-based fuzzing. In: Aleti, A., Panichella, A. (eds.) *SSBSE 2020. LNCS*, vol. 12420, pp. 105–120. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59762-7_8
4. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: *SIGPLAN Not*, vol. 43, pp. 206–215, June 2008. <https://doi.org/10.1145/1379022.1375607>
5. Gold, E.M.: Language identification in the limit. *Inf. Control* **10**(5), 447–474 (1967). [https://doi.org/10.1016/S0019-9958\(67\)91165-5](https://doi.org/10.1016/S0019-9958(67)91165-5)
6. Gopinath, R., Mathis, B., Zeller, A.: Inferring input grammars from dynamic control flow. *CoRR abs/1912.05937* (2019). arxiv.org/abs/1912.05937
7. De la Higuera, C.: *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, New York (2010)
8. Hodován, R., Kiss, A., Gyimóthy, T.: Grammarinator: a grammar-based open source fuzzer. In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pp. 45–48. A-TEST 2018, Association for Computing Machinery, New York, NY, USA (2018). DOI [10.1145/3278186.3278193](https://doi.org/10.1145/3278186.3278193)
9. Hopcroft, J.E., Ullman, J.D.: *Introduction to Automata Theory. Languages and Computation*. Addison-Wesley, Boston (1979)
10. Hörschele, M., Zeller, A.: Mining input grammars from dynamic taints. In: Lo, D., Apel, S., Khurshid, S. (eds.) *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, pp. 720–725. ACM (2016). <https://doi.org/10.1145/2970276.2970321>
11. Hörschele, M., Zeller, A.: Mining input grammars with AUTOGRAM. In: Uchitel, S., Orso, A., Robillard, M.P. (eds.) *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017*, pp. 31–34. IEEE Computer Society (2017). <https://doi.org/10.1109/ICSE-C.2017.14>
12. Le, X.B.D., Pasareanu, C., Padhye, R., Lo, D., Visser, W., Sen, K.: Saffron: adaptive grammar-based fuzzing for worst-case analysis. *SIGSOFT Softw. Eng. Notes* **44**(4), 14 (2021). <https://doi.org/10.1145/3364452.3364455>
13. Moser, M., Pichler, J.: eKnows: platform for multi-language reverse engineering and documentation generation. In: *IEEE International Conference on Software Maintenance and Evolution, ICSME 2021*, pp. 559–568. IEEE (2021). <https://doi.org/10.1109/ICSME52107.2021.00057>
14. Oncina, J., García, P.: Identifying regular languages in polynomial time. In: *Advances in Structural and Syntactic Pattern Recognition*, pp. 99–108 (1992). <https://doi.org/10.1142/9789812797919.0007>

15. Romero, J.: Pyformlang: an educational library for formal language manipulation. In: Sherriff, M., Merkle, L.D., Cutter, P.A., Monge, A.E., Sheard, J. (eds.) SIGCSE 2021: The 52nd ACM Technical Symposium on Computer Science Education, pp. 576–582. ACM (2021). <https://doi.org/10.1145/3408877.3432464>
16. Wu, Z., et al.: REINAM: reinforcement learning for input-grammar inference. In: Dumas, M., Pfahl, D., Apel, S., Russo, A. (eds.) Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, pp. 488–498. ACM (2019). <https://doi.org/10.1145/3338906.3338958>