Symbolic Parsing of LL(k) Grammars

Abstract. Efficient parsing of context-free grammars is only possible for subsets of context-free languages, LL(k) being one such parsable subset. However, inferring that if an LL(k) grammar exists for a given language is undecidable in general. This makes it difficult for programming language designers to design constructs that keeps their language within the parsable subset. Students exposed to parsing technology also find it challenging to understand the underlying structure of such grammars.
In this work, we propose an end-to-end symbolic algorithm for parsing of LL(k) grammars. One can draw parallels of our contribution with the design of *verification conditions* in program analysis. We implement our ideas into our tool, $\mathcal{A}$THENA , and demonstrate its utility by building two applications over it: *automated parser synthesis* to automatically synthesize LL(k) parsers (grammar and its parse table) and *repair of token sequences* to automatically repair a buggy sequence of tokens such that the resulting string is accepted by the provided parser. Our experimental results demonstrate that our tool is indeed capable of building realistic applications. Like verification conditions have found widespread applications in program analysis, we believe that this work can serve as a foundation for the development of interesting applications in parsing.

1 Introduction

Given a string w and a grammar $\mathcal{G}$, *parsing* algorithms (*parsers*) are designed to efficiently answer the membership query: *Is* $w \in \mathcal{L}(\mathcal{G})$ (is w a member of the language described by $\mathcal{G}$)? In the process, parsers construct a *parse tree* that captures the *derivation* of the string from the grammar; parse trees expose the *structure* of a parsed string—an artifact that can be employed by a *semantic analyzer* to infer the *meaning* of the string. For example, in a compiler for a programming language, the membership query boils down to answering if some provided text is indeed a valid program (in the respective language); the parse tree of the program helps a compiler understand its *semantics* so that it can be translated to the respective target language.

Unfortunately, efficient parsers cannot be constructed for arbitrary context-free languages. But, there does exist subsets of context-free languages, like the LL(k) languages, that allow for efficient parsers. At the same time, the LL(k) languages are expressive enough for realistic programming languages [3].

LL(k) grammars, however, do not have a simple syntactic identity—that is, given a context-free grammar, it is not possible to deduce it to be LL(k) simply by examining its structure; successful construction of a *LL(k) parse table* is the test that a grammar needs to pass to qualify as LL(k). Therefore, designing an LL(k) grammar for a given programming language construct is non-trivial; one generally requires multiple trials and rounds of debugging efforts on parse table *conflicts* to design an LL(k) grammar for a given language. We illustrate the non-triviality of such an endeavor in Figure 1.

In general, inferring if an LL(k) grammar exists for a given language is **undecidable**. Hence, it is usually difficult to design programming language

$$S \rightarrow A \mid (\ S\ ) \mid id$$
$$A \rightarrow C\ +\ id \mid C\ *\ id$$
$$C \rightarrow S \mid id\ (\ S\ )$$

(a)

$$S \rightarrow A \mid (\ S\ ) \mid id$$
$$A \rightarrow C\ Z$$
$$Z \rightarrow +\ id \mid *\ id$$
$$C \rightarrow S \mid id\ (\ S\ )$$

(b)

$$S \rightarrow A \mid (\ S\ ) \mid id$$
$$A \rightarrow S\ Z \mid id\ (\ S\ )\ Z$$
$$Z \rightarrow +\ id \mid *\ id$$

(c)

$$S \rightarrow S\ Z \mid id\ (\ S\ )\ Z$$
$$\mid (\ S\ ) \mid id$$
$$Z \rightarrow +\ id \mid *\ id$$

(d)

$$S \rightarrow\ id\ (\ S\ )\ Z\ Y$$
$$\mid (\ S\ )\ Y \mid id\ Y$$
$$Y \rightarrow Z\ Y \mid \epsilon$$
$$Z \rightarrow +\ id \mid *\ id$$

(e)

$$S \rightarrow\ id\ X \mid (\ S\ )\ Y$$
$$X \rightarrow\ (\ S\ )\ Z\ Y \mid\ Y$$
$$Y \rightarrow Z\ Y \mid \epsilon$$
$$Z \rightarrow +\ id \mid *\ id$$

(f)

Fig. 1: **Transforming a grammar to LL(1)**: We demonstrate the non-triviality of converging on an LL(k) grammar by attempting to infer an LL(1) grammar for arithmetic expressions. The first grammar (Figure 1a) is clearly not LL(1) as there exists a left factor 'C' for the productions of 'A'. Left factoring is not sufficient as now there exists a (indirect) left recursion on 'S' (Figure 1b); substituting 'C' (Figure 1c) and then 'A' (Figure 1e) exposes the left recursion. Remove this left recursion (Figure 1e) now shows 'id' as a left factor for 'S'. Left factoring finally produces a grammar that is LL(1) (Figure 1f).

constructs while ensuring that the desired language remains LL(k) parsable. Not just programming language designers, but also students of courses in Compiler Theory struggle when exposed to LL parsing algorithms; some interesting queries from students on this topic can be found on StackExchange [2, 1].

We propose a symbolic algorithm for LL(k) parsing and build a tool, $\mathcal{A}$THENA , that encodes our algorithms as SMT constraints. $\mathcal{A}$THENA imbibes an end-to-end symbolic encoding of the entire parsing process—right from the grammar, down to the subject string to be parsed are encoded as a set of symbolic constraints. This allows $\mathcal{A}$THENA to be used in multiple *modes*: making the symbolic variables corresponding to the production rules as free variables allows LL(1) parser synthesis, making the symbolic variables corresponding to the input string as free variables allows us to sample strings from a given grammar; asserting both a grammar and a subject string makes $\mathcal{A}$THENA operate as a verifier to answer the query if the grammar is indeed LL(1) and if the string can be derived from this grammar (allowing building of symbolic tools to analysze conflicts). We list some of the possibilities:

- **Programming Language Design** Programming language designers could use $\mathcal{A}$THENA for design exploration of *parsable* language constructs, for creating a new language or extensions of an existing one. $\mathcal{A}$THENA can check that the language remains LL(k) parsable and provide debugging support

for conflicts; we plan to leverage research in symbolic debugging of programs using constructs like interpolants[6] and unsat-cores [38] in the future.

⋆ **Automated Parser Synthesis** $\mathcal{A}$THENA can be used to automatically generate LL(k) parsers for (small) languages when supplied with a set of example strings in the language.

⋆ **Repair of buggy token sequences** Given a set of token sequences (say from a programming language) that fails to parse on a grammar, $\mathcal{A}$THENA can synthesize the *smallest* repair to allow successful parsing. One can use this idea for automated repair of syntax errors in a program.

◦ **Computer Science Education** The symbolic encoding in $\mathcal{A}$THENA can encourage building of tools that can assist students understand the complexities of LL(k) parsers by providing relevant customized feedback. Instructors, can use such tools, to automatically generate problem statements with certain properties (eg. grammars that are LL(2) but not LL(1)). In the future, we plan to reemploy symbolic techniques in education for teaching programming[36, 17] with our symbolic model for teaching parsing technology.

To the best of our knowledge, this is the first effort in building symbolic parsing algorithms. To demonstrate the utility of our effort, we have implemented modules for the starred items above, viz. automated parser synthesis (section 4.1) and repair of token sequences (section 4.2) using $\mathcal{A}$THENA . We plan to build more modules (including the ones mentioned above) and symbolic algorithms for other parsers in the future.

We make the following contributions in this work:

- We design a symbolic parsing algorithm for LL(k) parsing; we instantiate our algorithms by building an SMT encoding for it into our tool $\mathcal{A}$THENA .
- We demonstrate the utility of our tool by building the modules for the following applications:
  - **Automated parser synthesis** Given a set of positive examples, $\mathcal{A}$THENA automatically produces an LL(k) grammar and its corresponding parse table.
  - **Repair of token sequences** Given an LL(k) grammar $\mathcal{G}$ and a string $w \notin \mathcal{G}$, $\mathcal{A}$THENA automatically constructs a set of repairs (mutations, including replacement, addition and deletion of tokens) such that the repaired string w' is accepted by the parser.
  - Our experiments on a set of benchmark grammars shown in Table 1 and 2 demonstrates that our ideas are practical.

2 Overview

A grammar $\mathcal{G}$ is described by a tuple $\langle \mathcal{T}, \mathcal{N}, S, \mathcal{P} \rangle$, where $\mathcal{T}$ is a set of terminals, $\mathcal{N}$ is a set of non-terminals, $S \in \mathcal{N}$ is the start symbol and $\mathcal{P}$ is a set of production rules. Each (production) rule $p \in \mathcal{P}$ is described by a tuple $\langle H, B \rangle$, where $H \in \mathcal{N}$ is a non-terminal (referred to as the "head" of the rule) and B is the "body" for the rule, described by a sequence of terminals and nonterminals $(\mathcal{T} \cup \mathcal{N} \cup \{\epsilon\})$. We give an overview of the classical LL(k) parsing algorithm in the Appendix.

2.1 Our LL(k) Parse Table Constraints

We describe our LL(k) parse table by defining the required first/follow set constraints. The first set constraints, $t \in First(X)$, search a path from a production

with $X \in \mathcal{N}$ as its head to a production with a body that (explicitly) derives t as a *first* terminal in any one of its strings. This path is referred to as the *witness* for $t \in First(X)$. Similar is the case for the follow set constraints.

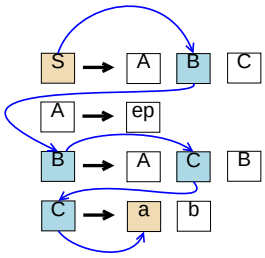


Fig. 2: First set

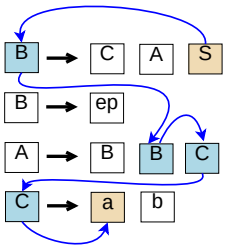


Fig. 3: Follow set

Example Figure 2 shows a snippet of a grammar where we attempt to answer if $a \in First(S)$: we label a node in the path as $\langle r, i \rangle$, where r is an identifier for a production, and i is the position of a symbol in the body of the production. In the provided example, we can provide the witness path as a sequence $[\langle 1, 2\rangle, \langle 3, 2\rangle, \langle 4, 1\rangle]$: As A is nullable, $First(B) \subseteq First(S)$ and $First(C) \subseteq First(B)$; Finally, $a \in First(C)$ as a is the first terminal in the body. Please note that $\langle 1, 2\rangle$ and $\langle 3, 2\rangle$ could be picked in the path only because A was nullable.

Figure 3 shows a snippet of a grammar where we attempt to answer if $a \in Follow(S)$: we label a node in the path as $\langle r, i, j\rangle$, where r is an identifier for a production, i is the position of a non-terminal symbol in the body of the production (for which we are trying to obtain the follow set relationship) and j is the position of the symbol in the production (in the body or the head), which is causing the follow set relationship to hold. In the provided example, we can produce the witness path as a sequence $[\langle 1, 3, 0\rangle, \langle 3, 2, 3\rangle]$: $Follow(B) \subseteq Follow(S)$ because if terminal t can follow B in a sentential form, we can replace B by CAS so that t can also follow S; if $t \in First(C)$, then, it can follow B and thus, $a \in First(C) \implies a \in Follow(S)$.

2.2 Our LL(k) Parsing Algorithm

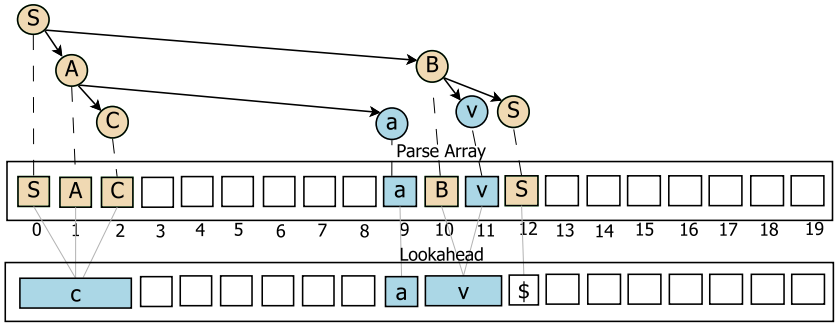
Our algorithm assumes a parse to be valid, if and only if, we are able to construct a valid parse tree corresponding to the grammar $\mathcal{G}$ and the given string w. We consider a parse tree valid iff:

- The root node contains the start symbol;
- Each non-terminal is expanded using a production of a grammar;
- Each non-terminal in the parse tree has children corresponding to the symbols appearing in one of its production rules; the production rule must be chosen according to the parse table;
- The leaves of the tree, read from left to right, correspond to the input string.

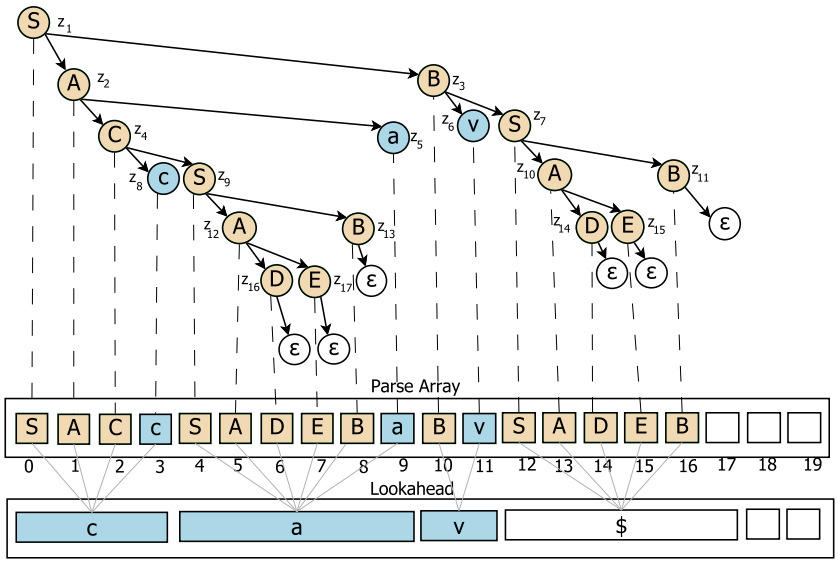
Embedding of a linear array in a parse tree We define an *embedding* of a bounded size linear array $[\mathcal{A}_0, \ldots, \mathcal{A}_b]$ into a parse tree $\mathcal{Z}$ (where the array size $b \geq |\mathcal{Z}|$) as follows:

- $\mathcal{A}_0$ maps to the root node in the parse tree;
- For any node $z \in \mathcal{Z}$ with children $c_1, c_2, \ldots, c_n, 0 \leq index(z) \leq index(c_i) \leq b$ and $0 \leq index(c_i) \leq index(c_k) \leq b$, for each $i < k$ (where $index(\mathcal{A}_i) = i$ for elements in the linear array);
- If $\mathcal{A}_i = h$, then $\Pi_{k=i}^{b} A_k = h$ (where h is a additional node introduced in the parse tree, referred to as *hole*, shown via empty boxes in Figure 4).

In other words, $\mathcal{A}_i$ maps to the parse tree node that will be the i^{th} node to be visited (not counting multiple visits to the same node) during a depth-first



(a) Intermediate stage



(b) Parsing complete

Fig. 4: Embedding of a parse tree

traversal of the parse tree; given a node, its children must be visited from left to right (in the same order in which they appear in the body of the production used for the expansion), i.e. it is a *preorder* listing of the nodes in the parse tree. The ϵ symbols are not mapped. All elements $\mathcal{A}_i$, $i > |\mathcal{Z}|$ are mapped to a special (artificial) element, *hole*.

An *embedding function* (or simply *embedding*) is a sequence of parse tree nodes as they appear in the linear array. For example, in Figure 4b , we do get a valid embedding (nodes numbered in a breadth-first order) $[z_1, z_2, z_4, z_8, z_9, z_{12}, z_{16}, z_{17}, z_{13}, z_5, z_3, z_6, z_7, z_{10}, z_{14}, z_{15}, z_{11}, z_h, z_h, z_h]$ where z_h is an artificially introduced *hole*.

Our Symbolic LL(k) Parsing Algorithm The embedding can be viewed in another way: as a sequence of symbols listed **in the order as they are popped off from the parsing stack** during the classic LL parsing algorithm (discussed

in Section A). This allows us to use this embedding to replace the parsing stack instead of exactly mimicking the LL parsing algorithm (as described in Section A) which would have required us to maintain a separate symbolic parsing stack for each step of the parser.

Given a (positive) example $w \in \mathcal{L}(\mathcal{G})$, our symbolic parsing algorithm essentially searches for a valid embedding (of a parse tree) in a bounded size array, which witnesses the derivation of w from the grammar $\mathcal{G}$. We impose additional constraints to ensure that the parse tree is constructed using a valid LL(1) parse table.

We demonstrate the process briefly using the example in Figure 4. Let us assume that when parsing using the classical LL parsing algorithm (Section A), a symbol gets popped off the parsing stack every second. We maintain a parsing array $[\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_b]$, such that the symbol that gets popped off the parsing stack at the i^{th} second occupies $\mathcal{A}_{i-1}$ in our parsing array.

As parsing commences, the start symbol S is the first symbol to be pushed, and then popped off the stack (in the first second). Hence, S gets placed at $\mathcal{A}_0$ in our parsing array. Next, (like in the classical algorithm) we consult the parse table to select a rule, say $S \rightarrow AB$, to expand. Now, we *guess the future* for the **time instants** when each of the symbols in the body of the rule will be popped off the stack (when running the classical LL parser): it is obvious that the first symbol, A will be the very next symbol to be popped off; so we place it at $\mathcal{A}_1$. Let us assume that we *guess* that B will be popped off at 11 seconds; hence, we place this symbol at $\mathcal{A}_{10}$. Similarly, the other symbols get placed in the array, each non-terminal guessing the positions of the symbols in the body of the production.

If our parsing array indeed represents an embedding of a valid parse tree (Figure 4b via the dashed lines), we accept this sequence of guesses (or non-deterministic choices) as a valid parse. In our algorithm, the constraints of the embedding are encoded as constraints and we employ an SMT solver to infer the "correct" guesses that produces a valid embedding.

3 Algorithms

We describe our symbolic algorithms via non-deterministic algorithms. $\mathcal{A}$THENA implements these algorithms as SMT constraints where the non-deterministic choices (guesses) appear as free variables to be inferred by an SMT solver. A terminating execution through the non-deterministic algorithms refers to a satisfying assignment for the constraints, the non-deterministic choices representing the solution set. We use the following two constructs to describe our algorithms:

- ⋆ **assume(ϕ)**: Used to block an execution if the condition ϕ does not hold; in other words, all executions through this construct must satisfy ϕ.
- ⋆ **choose(ψ)**: Used to non-deterministically select a value from the set ψ.

3.1 Symbolic encoding of the First set constraints

The first sets of the non-terminals can be found by computing the **minimum** fixpoint over a set of constraints (see Appendix section B).

As we need to compute the minimum fixpoint (as opposed to any fixpoint), every terminal added to the first set must have a *witness* as to why it was added.

$$FirstSetWitness_{\langle r,i\rangle}(X,t) = \begin{cases} t, & \text{if } Y_i = t \in \mathcal{T} \wedge \Pi_{k=1}^{i-1}, First(Y_k, \epsilon) \\ Y_i, & \text{if } Y_i \in \mathcal{N} \wedge \Pi_{k=1}^{i-1}, First(Y_k, \epsilon) \\ t, & \text{if } X = t \\ nil, & \text{otherwise} \end{cases}$$

$$EpInFirst_0(X) = \begin{cases} true, & \text{if there exists a rule } \langle X \rightarrow \epsilon \rangle \\ nil, & \text{otherwise} \end{cases}$$

$$EpInFirst_i(X) = \begin{cases} true, & \text{if } \Pi_{k=1}^{n}, EpInFirst_{i-1}(Y_k) \\ true, & \text{if } EpInFirst_{i-1}(X) \\ nil, & \text{otherwise} \end{cases}$$

Fig. 5: First set witness functions

$S \rightarrow A\ B$ [1]
$A \rightarrow C\ a$ [2]
$\quad \mid D\ E$ [3]
$B \rightarrow v\ S$ [4]
$\quad \mid \epsilon$ [5]
$C \rightarrow c\ S$ [6]
$D \rightarrow \epsilon$ [7]
$E \rightarrow \epsilon$ [8]

Fig. 6: $\mathcal{G}_1$

To ensure the computation of the minimum fixpoint, we compute a *witness* function (figure 5) that returns a symbol that can attest as a witness for the relation $t \in First(X)$. Algorithm 1 describes our algorithm for computing the first set. It uses two functions:

- $FirstSetWitness_{\langle r,i\rangle}(X,t)$: Given a rule `r` and a symbol Y_i in the body of the rule, this function attempts to find if $First(Y_i) \subseteq First(X)$; if it is indeed the case, then Y_i can be attributed as one of the *reasons* why the terminal `t` was introduced in the first set of `X`.
- $EpInFirst_i(X)$: The function $EpInFirst_0(X)$ would evaluate to true if we can infer that $\epsilon \in first(X)$ by a single production rule (like $X \rightarrow \epsilon$). In the same vein, $EpInFirst_i(X)$ would evaluate to true iff we infer $\epsilon \in first(X)$ via a chain of less than or equal to i production rules, each such chain terminating with an epsilon production.

Our symbolic encoding for computing the first set is descibed in Algorithm 1. Here, for a query `First(X,t)`, the algorithm has a terminating execution if and only if $t \in First(X)$; correspondingly, our SMT encoding has a satisfying assignment if and only if $t \in First(X)$. To infer if $\epsilon \in First(X)$, we invoke $EpInFirst_{|\mathcal{N}|}(X)$ in a search of a chain of production rules that could witness the derivation of ϵ from X. Note that any such chain that could derive ϵ from X cannot be longer than the number of non-terminals.

To infer if $t \in \mathcal{T}$ is in First(X), we search over all production rules $r \in \mathcal{P}$ where X appears as the head of a rule, and over all (non-epsilon) symbols Y_i in the body of such productions, to uncover a sequence of rules that could witness $t \in First(X)$. The loop at line 7 attempts to *guess* a sequence of tuples $\langle r, i\rangle$ (of length at most the number of non-terminals) that could witness the derivation of $t \in First(X)$. As this algorithm is encoded as an SMT formula, we use the SMT solver to infer the correct *guesses* for our non-deterministic choices.

Please note that we are operating on a symbolic grammar (where all the production rules appear as free variables), the first set constraints are only enforced; no real computation of the first set takes place at this point.

Algorithm 1: First Set constraints

```
1  Function First
       Input: X, t
2      if t = ϵ then
3          x ← EpInFirst_{|𝒩|}(X)
4          assume(x)
5      else
6          y ← X
7          for k ∈ [1 ... |𝒩|] do
8              ⟨r : Y_0 → Y_1 ... Y_n⟩ ← choose(𝒫)
9              assume(Y_0 = y)
10             i ← choose([1 ... n])
11             assume(Y_i ≠ ϵ)
12             y ← FirstSetWitness_{⟨r,i⟩}(y, t)
13         end
14         assume(y = t)
15     end
```

Example: Consider the LL(1) grammar $\mathcal{G}_1$ in Figure 6: We demostrate our first set constraints by assuming that the symbolic variables corresponding to the production rules are asserted with the grammar $\mathcal{G}_1$. We use the notation F to denote $FirstSetWitness$ and Ep to denote $EpInFirst$.

$F_{\langle 6,1\rangle}(C,c) = c$ and $F_{\langle 4,1\rangle}(B,v) = v$ due to the first condition in Figure 5. From rule 2, we get $F_{\langle 2,1\rangle}(A,c) = C$. From the empty productions for B, D and E (rules 5, 7 and 8 respectively), we get $Ep_0(B) = true$, $Ep_0(D) = true$ and $Ep_0(E) = true$. Applying the conditions in Figure 5 on rule 3, we get $Ep_1(A) = Ep_0(D) \wedge Ep_0(E) = true$. With rule 1, we get $Ep_2(S) = true$, $F_{\langle 1,1\rangle}(S,c) = A$ and $F_{\langle 1,2\rangle}(S,v) = B$.

Now, we simulate Algorithm 1 by analyzing $FirstSet(S,c)$. In the check on line 2, the else branch is taken. In the first iteration of the loop on line 7, we have $y = S$. Assuming that the solver non-deterministically selects $r = (1 : S \to AB)$ on line 8, the assumption is satisfied as $Y_0 = y = S$. Let us assume that the solver non-deterministically selects $i = 1$ to explore a path via the first symbol in the body of rule 1, i.e. A (line 10); hence, on line 12, $y \leftarrow F_{\langle 1,1\rangle}(S,c) = A$. In the second iteration of the loop, we attempt to extend this path from $y = A$. Again, assuming non-deterministic selections $r = 2$ and $i = 1$, we get $y \leftarrow F_{\langle 2,1\rangle}(A,c) = C$ (from figure 5). In the third iteration, the choice of $r = 6$ and $i = 1$ results in $y \leftarrow F_{\langle 6,1\rangle}(C,c) = c$.

In the subsequent iterations of the loop, the update operation is $y \leftarrow F_{\langle \_,\_\rangle}(c,c) = c$. Here, we reach a fixpoint, and hence exit the loop with $y = c$. This satisfies the assumption on line 14 and the program reaches a successful termination, thereby validating that $c \in First(S)$ with the witness path $[\langle 1,1\rangle, \langle 2,1\rangle, \langle 6,1\rangle]$.

If we try to simulate Algorithm 1 on $FirstSet(A,v)$, we would find that there are no possible assignments to r and i that can produce a terminating execution of the program, thereby disallowing this possibility in our symbolic model.

3.2 Symbolic encoding of the Follow set constraints

Follow set construction also requires the computation of the **minimum** fixpoint over a set of contstraints (see Appendix section B). We (again) define *witness* functions to ensure that the computed set is indeed the minimum fixpoint (and not some arbitrary fixpoint). Given a production rule $\langle r : Y_0 \to Y_1 \dots Y_n\rangle$ and a symbol Y_i in its body, the witness functions (Figure 7) essentially encodes if due to some symbol Y_j, $j = 0$ or $j > i$, we get $Follow(Y_j) \subseteq Follow(Y_i)$

$$FollowSetWitness_{\langle \_,\_,\_ \rangle}(S, \$) = \$$$

$$FollowSetWitness_{\langle r,i,j \rangle}(X, t) = \begin{cases} t, & \text{if } X = Y_i \in \mathcal{N} \wedge t \in First(Y_j) \wedge \Pi_{k=i+1}^{j-1} First(Y_k, \epsilon) \\ X, & \text{if } j = 0 \wedge X = Y_i \in \mathcal{N} \wedge Y_i \in \mathcal{N} \wedge \Pi_{k=i+1}^{n}, First(Y_k, \epsilon) \\ t, & \text{if } X = t \\ nil, & \text{otherwise} \end{cases}$$

Fig. 7: Follow set witness functions

or $First(Y_j) \subseteq Follow(Y_i)$ (respectively), thereby attributing Y_j as one of the *reasons* for $t \in Follow(Y_i)$ (if it eventually turns out to be the case).

Algorithm 2: Follow set constraints

```
1  Function FollowSet
       Input: X, t
2      y ← X
3      for k ∈ [1 ... |N|] do
4          ⟨r : Y_0 → Y_1 ... Y_n⟩ ← choose(P)
5          i ← choose([1, ..., n])
6          j ← choose({0} ∪ [i + 1, ..., n])
7          assume(Y_i = y)
8          y ← FollowSetWitness_⟨r,i,j⟩(y, t)
9      end
10     assume(y = t)
```

Algorithm 2 makes use of the follow witnesses to construct a terminating execution if and only if $t \in Follow(X)$. The algorithm essentially attempts to construct a chain of production rules that can attest to $t \in Follow(X)$. To do so, it guesses a sequence of tuples $\langle r, i, j \rangle$ that can serve as a witness to the follow set membership. The choice of $\langle r, i, j \rangle$ can be interpreted as follows:

- r: A rule that can potentially be used to infer that $t \in Follow(X)$;
- i: In the rule r, i is the position in the body at which X is present (note that X can be present at more than one location);
- j: With X at location i in rule r, the symbol at position j that can lead to $t \in Follow(X)$; there exist two possibilities for j:
  1. $j = 0$: $t \in Follow(Y_0)$ (head of rule r) and $\epsilon \in First(Y_k) \forall i < k <= n$;
  2. $j > i$: $t \in First(Y_j)$ (in the body) and $\epsilon \in First(Y_k) \forall i < k < j$.

We provide an example to illustrate the working of Algorithm 2 in Appendix section D. Please note that *witness* construction is essential as otherwise we may not converge on the minimum fixpoint: for the rules $\{\langle S \rightarrow AB \rangle, \langle B \rightarrow bS \rangle\}$, the follow set constraints can be satisfied for $a \in Follow(B) \wedge a \in Follow(S)$; notice that this is **not the minimum fixpoint** as there is no production rule that could introduce a in $Follow(S)$.

3.3 Symbolic encoding of the LL(k) parse table

We encode an LL(k) parse table as a function, $ParseTable(X, t_1, \ldots, t_k)$ that maps a non-terminal (X) and a sequence of lookahead terminals ($[t_1, \ldots, t_n]$) to a production rule, or **error**. We define the parse table as a function to constrain the system to valid LL(k) parse tables — parse tables where a cell is occupied

by atmost one production. In particular, an LL(1) ParseTable can be defined as:

$$ParseTable(X,t) = \begin{cases} \langle r1 : A \rightarrow \alpha \rangle \in P, & \text{if } t \in First(\alpha) \\ \langle r1 : A \rightarrow \alpha \rangle \in P, & \text{if } \epsilon \in First(\alpha) \text{ and } t \in Follow(A) \\ error, & \text{otherwise} \end{cases}$$

3.4 Symbolic encoding of the LL(k) parsing algorithm

The classical algorithm for top-down parsing[24] requires a parsing stack; a symbolic encoding of this algorithm would require encoding of a symbolic stack at each parsing step. We, rather, design a non-deterministic algorithm (Algorithm 3) that attempts to search for an embedding of a linear array (`parseArray`) in a valid parse tree (see Section 2). The SMT solver is used to "guess" the correct values for the non-deterministic choices to answer a membership query.

The recursive function `LL1Parse()` takes the following arguments: the symbol `symbol` to be expanded, the bounds of our parsing array (`parseArray`) that this instance of the call can operate on (`arrayBegin` to `arrayEnd`), and a pointer (`lookAheadIndex`) to the first symbol in the input string that this call can start consuming. The `LL1Parse()` function returns the next available position in `parseArray` (pe) and the position of the lookahead index (li) in the input string. The `Main()` procedure calls `LL1Parse()` (line 34) with the start symbol `S`, the complete bounds of `parseArray` (i.e. from `0` to `parseArrayBound-1`) and (`lookAheadIndex` at `0`). Depending on `symbol`, the algorithm proceeds as follows:

`symbol` is a non-terminal The algorithm first places `symbol` at `parseArray[arrayBegin]` at line 3; it, then, makes non-deterministic choices for the following entities:

- A rule, $r : \langle X \rightarrow Y_1 Y_2 \dots Y_n \rangle$, that must be used to expand `symbol`;
- A set of parseArray indices, $\{p_1, \dots, p_n, p_{n+1}\}$, where the expansion of `X` using the rule r (i.e. $\{Y_1, Y_2, \dots, Y_n\}$) must be placed for a successful *embedding* the parse tree; the index p_{n+1} corresponds to the index where the next symbol (after the subtree of `X`)in the preorder traversal of the parse tree must be placed;
- A set of lookahead indices, $\{a_1, \dots, a_n, a_{n+1}\}$, in the input string; the lookahead index a_i will be used when the parser processes the symbol at location p_i in `parseArray`.

Any production rule r that is selected in line 5 must meet the constraints (lines 6 – 7) that the head of the rule, `X`, matches `symbol` (that is being expanded) and should be done by consulting the `ParseTable` (i.e. the parse table entry corresponding to `X` for current lookahead character must be the rule r).

The sequence of positions in the parseArray, $[p_1, \dots, p_n]$, are chosen in a manner such that they appear in an non-decreasing order and lie within the permissible bounds in `parseArray` i.e. [`arrayBegin+1 ... arrayEnd`] (lines 9–12).An additional constraint is that the index p_1 must start from the next empty position ($arrayBegin + 1$) in the parseArray so that there are no "holes" in the middle of parseArray (holes can appear only at the end).

The sequence of lookahead indices in the input string, $[a_1, \dots, a_{n+1}]$, are chosen in a manner (lines 14–16) such that they appear in non-decreasing order, lie within the valid bounds in the input string ([`lookAheadIndex`, ..., `strlength+1`]), assuming $ is at `inputString[strlength+1]` and the index a_1 matches with the current lookahead index ($lookAheadIndex$).

Algorithm 3: Our Symbolic LL(k) Parsing algorithm

```
1  Function LL1Parse
       Input: symbol, arrayBegin, arrayEnd, lookAheadIndex
2      if symbol ∈ N then
3          assume(parseArray[arrayBegin] == symbol);
4          ;
5          r : ⟨X → Y1Y2 ... Yn⟩ ← choose(P); /* guess production rule     */
6          assume(X == symbol);
7          assume(ParseTable[X, inputString[lookAheadIndex]] == r);
8          ;
9          ⟨p1, ..., pn⟩ ← choose([arrayBegin + 1, arrayEnd]);/* guess parse pt
               */
10         pn+1 ← choose([arrayBegin + 1, arrayEnd + 1]);
11         assume(p1 ≤ p2 ≤ ··· ≤ pn ≤ pn+1);
12         assume(p1 == arrayBegin + 1);
13         ;
           /* guess lookahead indices                                        */
14         ⟨a1, ..., an, an+1⟩ ← choose([lookAheadIndex, strlength + 1]);
15         assume(a1 ≤ a2 ≤ ··· ≤ an ≤ an+1);
16         assume(a1 == lookAheadIndex);
17         ;
           /* parse subtrees concurrently                                    */
18         for parallel i ∈ {1, 2, ..., n} do
19             ⟨subpe, subli⟩ = LL1Parse(Yi, pi, pi+1 − 1, ai);
20             assume(subpe == pi+1);
21             assume(subli == ai+1);
22         end
23         return ⟨pn+1, an+1⟩
24     else
25         if symbol == ϵ then
26             return ⟨arrayBegin, lookAheadIndex⟩
27         else
28             assume(symbol == inputString[lookAheadIndex]);
29             assume(parseArray[arrayBegin] == symbol);
30             return ⟨arrayBegin + 1, lookAheadIndex + 1⟩
31         end
32     end
33 Function Main
34     ⟨pe, li⟩ = LL1Parse(S, 0, parseArrayBound-1, 0);
35     assume(pe ≤ parseArrayBound);
36     assume(inputString[li] == $);
```

The algorithm then (lines 18 – 21) recursively, and in parallel, expands each of the symbols, $\{Y_1, \ldots, Y_n\}$, filling their assigned subarrays with the embedding of the respective subtrees from the parse tree. Each recursive call to `LL1Parse` attempts to expand its respective symbol Y_i with its corresponding subarray of the parse array within $[p_i, p_{i+1} - 1]$ and its respective guess for the lookahead index at a_i; each call enforces the following constraints on its arguments:

- The last index in the parsing array by a call is returned in the variable, *subpe*; the beginning of the embedding of the next subtree, p_{i+1}, must conform with *subpe*, so that there are no "holes" in the middle of parseArray;
- The lookahead index in the input string passed by the recursive call in the variable *subli* must conform with a_{i+1}, the guess of the lookahead symbol for embedding of the next symbol Y_{i+1}.

Finally, we return the index next to the final index of the parseArray that got populated by this instance of the recursive call, p_{n+1}, and the position of the lookahead symbol after the last recursive call, a_{n+1}.

'symbol' is epsilon If the current symbol is ϵ, it simply returns with no updates to the parseArray, and relaying the same lookahead index (line 26).

'symbol' is a terminal In this case (lines 28 – 30), we *match* the current terminal with the next input symbol (at $inputString[lookAheadIndex]$), place 'symbol' at parseArray[arrayBegin], and, finally, return from the recursive call while incrementing the input pointer, `lookAheadIndex`, by one (as one terminal has now been consumed from the input and matched against).

Example: We now show the parsing of the string `cav` $\in L(\mathcal{G}_1)$ (Figure 6) using our algorithm. The first call to `LL1Parse()` (Algorithm 3) from `Main()` takes the arguments `(S, 0, parseArrayBound-1, 0)`. Since $S \in \mathcal{N}$, the control goes to line 3, where the symbol at `parseArray[0]` is assumed to be `S`. At line 5, the correct guess of $r = 1$ meets the assumptions $X == S$ and `ParseTable[S, inputString[0]] == 1`. Proceeding with correct guesses on the subsequent selections on p_1, p_2, a_1 and a_2, the constraint on line 12 and 16 leads to $p_1 == arrayBegin + 1 = 1$ and $a_1 == lookAheadIndex = 0$ respectively. Hence, in the loop on line 18, the call in the first iteration will be to `LL1Parse(A, 1,` $p_2 - 1$`, 0)`.

Within that call, `A` will be placed at `parseArray[1]` and if rule `2` is chosen at line 5, then subsequent calls will be made to `LL1Parse(`$C, 2, \_, 0$`)` and `LL1Parse(a,`$\_, \_, \_$`)` (we use '$\_$' for arguments that do not matter for our current discussion). The call to `LL1Parse(`$C, 2, \_, 0$`)` will first put c on `parseArray` (from `LL1Parse(`$c, 2, 3, 0$`)`) and, eventually make a call to `LL1Parse(`$S, 3, \_, 1$`)` to produce ϵ. The call to `LL1Parse(a,`$\_, \_, 1$`)` will place a on `parseArray` and increment `lookAheadIndex` for the subsequent `parseArray` entries (see Figure 4a). In the second iteration, a call to `LL1Parse(B,`p_2`,`$p_3 - 1$`,2)` will create other subsequent calls that will create the linear embedding of the parse tree (shown in Figure 4b). We now state a soundness theorem for Algorithm 3; the proof follows from Lemmas 3, 4, 5 and 6 (Appendix section C).

Theorem 1. *If the algorithm has a terminating execution, the parse array contains an embedding of a valid parse tree, with any* holes *only to the right of the last symbol in parse array.*

4 Applications

We implemented our tool, $\mathcal{A}$THENA , on Python using Z3 version 4.4.2 [11] as the SMT solver, used via the Z3 Python API. We evaluate our tool on a set of benchmark grammars collected from prior literature [7, 4] and online course notes [37, 19, 18, 8, 33]. We now demonstrate the two applications that we built on $\mathcal{A}$THENA .

1: $\rho_1 \leftarrow \Omega_{fst} \wedge \Omega_{flw} \wedge \Omega_{prstbl}$
2: $\rho_2 \leftarrow \rho_1 \wedge \bigwedge_{w \in \xi} \Omega_{embed}(w)$
3: $\langle result, \mathcal{G}_\rho \rangle \leftarrow$ CheckSAT(ρ_2)

(a) Synthesizing Parser

1: $\rho \leftarrow [\Omega_{prstbl} \wedge \Omega_{embed}(w)]_{hard}$
2: $\psi \leftarrow [\Omega_{repair}]_{soft}$
3: $\langle result, repair_\rho \rangle \leftarrow$ MaxSAT(ρ, ψ)

(b) Repairing token sequences

Fig. 8: Algorithms for parser synthesis and repair of token sequences

4.1 Parser Synthesis

Given a set of positive examples (ξ), $\mathcal{A}$thena asserts the symbolic encodings for the first and follow constraints ($\Omega_{fst}, \Omega_{flw}$), the parse table constraints (Ω_{prstbl}) and a (renamed) instance of the embedding/parsing algorithm (Ω_{embed}) for each positive string $w \in \xi$ (Figure 8a). We require separate instances of Ω_{embed} as each string would construct a distinct parse tree, and hence, we need to search for a different embedding in each case. We use our tool in two modes:

- Assert the parsing algorithm constraints for all the positive strings with a single call to CheckSat() (All at once).
- Add the parsing algorithm constraints for each of the positive strings incrementally, with multiple calls to CheckSat(); in this scheme, we make the initial calls to the solver with smaller queries, and hope to utilize the learning ability of the solver for the larger queries issued later.

We ran our experiments on a Linux container with a 2.4 GHz Intel processor with 198 GB main memory. Table 1 summarizes our results for our experiments. For each benchmark (identified by an identifier (Id)), we provide a *template* for the synthesized grammar in terms of the number of production rules (#P), maximum number of symbols in the body of the productions (#B), the number of terminals (#T) and non-terminals (#N). The language is described via a set of positive strings (strings accepted by the language); the number of such examples (#E) and the average length of the examples (Strlen) are also recorded. We found that the solving time in the incremental solving mode is quite sensitive to the order in which the examples are provided, which is understandable. Surprisingly though, we found that the same is also true for the All-at-once mode: the solving time was sensitive to the order in which the string constraints were listed. To understand this further, we added the set of strings in five different orders, selected uniformly at random; we report the minimum time taken (Min), the average time taken (Avg) and the standard deviation (SD) for these five runs. Though in many of them the differences in the results is small, we found cases (in blue) where one of the setting outperforms the other setting significantly. This suggests that it would be prudent to run $\mathcal{A}$thena with multiple settings in parallel, terminating others when one of them finishes.

4.2 Repair of token sequences

Given a string w as a sequence of tokens $[w_0, \ldots, w_n]$ and a parser (as a grammar $\mathcal{G}$ and an LL(1) parsing table) such that $w \notin \mathcal{L}(\mathcal{G})$, we attempt to find the **minimal** change to w such that it gets accepted by $\mathcal{G}$. We allow replacement (changing a token to another token), deletion (dropping an existing token) and insertion (addition of a new token) repairs to the existing token sequence; we implement a simplistic MaxSAT solver using Z3 to converge on the minimal changes that w requires to get accepted by the parser.

Table 1: Runtimes for Automated Parser Synthesis (in seconds)

							Incremental solver			All at once		
Id	#P	#B	#T	#N	#E	Len	Min	Avg	SD	Min	Avg	SD
1.	8	3	6	4	6	2.5	416	2254	3120	261	433	224
2.	13	2	7	7	20	3	2924	4991	2032	2800	5722	2937
3.	2	4	2	1	18	6.8	77	92	12	83	98	10
4.	5	4	3	3	20	7.8	146	974	1572	1194	2162	848
5.	4	2	2	3	2	4	3	3	0	4	6	2
6.	3	3	2	2	5	5	12	16	4	12	12	0
7.	7	3	5	4	20	5.3	473	3942	5075	1758	2532	674
8.	6	2	3	4	4	2	6	9	2	6	10	3
9.	5	3	3	3	20	6.4	95	128	50	116	268	118
10.	4	3	2	2	20	7.4	81	87	6	83	98	13
11.	3	2	2	2	20	10.5	67	78	7	66	73	7
12.	4	2	3	2	20	4.5	33	38	5	36	37	2
13.	5	2	2	3	19	3.5	30	34	4	12	30	11
14.	4	5	8	1	20	6.6	141	164	24	129	142	15

Our algorithm (Figure 8b) commences by asserting the parse table constraints and the parsing algorithm constraints; the first/follow set constraints are not required in this case as the parser is already available. We additionally require the repair constraints (Ω_{repair}) that we describe next.

Replacement We use a function `string` to return the token at a given position in the input sequence, with the *soft* constraints: $\forall_{i=0}^{n} string(i) = w_i$. The MaxSAT solver attempts to preserve the tokens, but being soft constraints, some tokens are replaced to satisfy the parsing algorithm constraints.

Deletion Deletion of a single token would break many constraints for `string` as all other token would have to be *shifted* left. Rather than deleting a token in this manner, we employ a function `next(pos)` that provides the next position the input pointer must progress from the current position `pos` to read the subsequent token. Instead of employing `string(pos+1)` to move to the subsequent token, the parsing algorithm now employs `string(next(pos))`. The following soft constraints are added on the `next` function: $\forall_{i=0}^{n} next(i) = i + 1$

Insertion Insertion uses additional *insertion* slots, $\{u_1, \ldots, u_k\}$ to accommodate additional tokens. The soft constraints corresponding to the `next` function can be broken, thereby making them point to the additional insertion slots to facilitate insertion. However, since our goal is to get the nearest possible correct string, we also use a `prev` function to ensure that the new slots added do not have a `next` pointer to a random location but instead should point close to its `prev` pointer.

Example With grammar $\mathcal{G}_2$ (Figure 9) and the string `abaacccc`, Figure 10 demonstrates all the three types of repairs: replacement by changing `string(6)` from `c` to `b`, deletion by changing `next(0)` from `1` to `2` and insertion by changing `next(7)` from `8` to `1000` (which is an insertion slot). Table 2 shows the runtimes for our attempt at evaluating the repair capabilities of $\mathcal{A}$THENA . The experiments were run on a Linux container with a 2 GHz Intel processor with 33 GB main memory. The first few column names are same as in Table 1. We randomly selected a string $w \in L(G)$ and, for each such $\langle G, w \rangle$ pair, used a random fault

$$S \rightarrow a\ A\ B\ b$$
$$A \rightarrow a\ \ A\ c \mid \epsilon$$
$$B \rightarrow b\ \ B \mid c$$

Fig. 9: Grammar $\mathcal{G}_2$

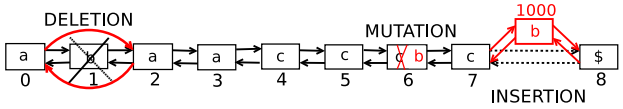


Fig. 10: Repair operations on a token sequence

Table 2: Runtimes for repairing of token sequences (in seconds)

				2I		2D		2R		1I1R		1I1D		1D1R		2R2I		2R2D		4R	
Id	#P	#B	Len	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD
1.	10	3	12	13	9	69	74	70	66	11	6	20	12	68	64	45	34	349	479	251	316
2.	7	4	10	6	1	9	1	7	1	7	1	8	1	9	1	6	1	14	3	9	1
3.	8	3	14	8	1	18	7	9	1	10	2	10	2	14	6	9	1	35	18	18	7
4.	4	5	12	54	43	182	108	36	11	36	20	51	34	57	36	94	56	809	359	125	100
5.	5	4	7	3	1	7	1	5	1	4	1	5	1	7	1	4	1	10	3	5	1
6.	6	6	10	18	5	35	21	25	17	21	7	31	16	38	21	34	14	161	103	85	64
7.	10	3	13	12	8	68	60	16	8	10	5	52	67	42	46	29	23	240	213	201	422
8.	7	4	11	13	2	26	15	15	4	12	2	14	2	15	3	18	5	31	16	25	7
9.	8	3	15	3	0	7	2	5	1	4	1	5	1	6	1	3	1	11	2	7	1
10.	4	5	13	12	3	44	23	24	11	14	4	19	5	37	16	17	5	247	247	68	32
11.	5	4	8	6	1	10	2	9	3	7	1	7	1	11	3	7	2	20	7	9	2
12.	6	6	11	32	17	58	34	33	18	40	23	36	15	87	133	95	48	338	442	212	284

injector to inject random faults (addition of extra token, replacement of a token by another, deletion of a token at randomly selected locations) of a certain class on w. This process is repeated ten times for each $\langle G, w \rangle$ pair. We report the average and standard deviation in our runtimes over such random fault injections for multiple classes for faults consisting of different numbers of replacements (R), deletions (D) and insertions (I); for instance, `2R2I` refers to the fact that two replacements and two insertions were needed for the repair.

Our results show that the average runtime are reasonable; also, the average runtimes are similar for fault classes with the same number of errors. At the same time, the high standard deviation values indicate that the some of the instances are found much easier for the solver than other, even for the same grammar—a common trait of SMT solvers.

5 Related Work

Synapse [29–32] attempts to generate grammars in the Chomsky normal form (CNF) using the Cocke-Younger-Kasami(CYK) algorithm [13]. The CYK algorithm is essentially used as an oracle to check if a grammar is consistent with all the examples; if not, the algorithm backtracks and attempts with another possible rule in CNF. Gramin [34] also adopt a similar methodology of trying out rules from all possible set of CNF rules using the CYK algorithm to check membership. Črepinšek et al. [41] drive a brute-force search for a parse tree in a bounded domain of grammar rules in an attempt to find a possible grammar that is consistent with the supplied examples. Others [15, 28, 39, 40] have used evolutionary approaches to synthesize grammars. Instead of grammar synthesis, some proposals [21, 22, 35] attempt to "recover" (or extract) grammars from

existing tools (like parsers and compilers). Our work focuses on parser synthesis rather than synthesis of context-free grammars; hence, our algorithm is not restricted to grammars in CNF. Converting a grammar to CNF can incur exponential blowup and the grammar can become difficult to comprehend, making design of semantic actions challenging. Also, instead of using brute-force search or evolutionary algorithms, we design an encoding for the parsing algorithm and leverage the power of modern SMT solvers in search for a grammar.

In the direction of synthesizing parsers from a set of examples: Jain et al. [14] use a user-defined knowledge-base of grammar rules to drive a backtracking based search through a bottom-up parser to discover a possible parse; in this scheme, success of the algorithm depends on user-interaction in terms of construction of a good knowledge base of grammar rules to be used in the search. Mernick et al. [28] use genetic programming to infer an LR(1) parser from a given set of examples; given a candidate grammar, the fitness function was designed to capture the number of examples that could be parsed by an LR(1) parser built from the grammar. Parsify [23] uses the A* search algorithm along with user feedback to generate parsers; we feel that combining symbolic technique with AI techniques (like A*) is a promising direction for parser synthesis.

In the direction of the use of symbolic methods for analyzing grammars, CFGAnalyzer [5] provides a SAT encoding for answering questions like inclusion, intersection and equivalence for context-free grammars; the SAT encoding is designed to search for a string of a bounded size that satisfies a given property (like a counterexample that two provided grammars are not equivalent). Madhavan et al. [26] propose a solution to the problem of grammar equivalence by giving an effective testing scheme which efficiently enumerates strings from a given language. They also propose a decision procedure for proving equivalence of two grammars which is complete for LL grammars. Rather than analysis of grammars, our work attempts to build an end-to-end symbolic parsing algorithm that can be employed for multiple applications from parser inference to enumeration and repair of strings in a grammar.

Our algorithms borrow heavily from symbolic techniques in program analysis and verification, especially from symbolic techniques for bounded model checking of programs using SAT/SMT solvers. Drawing parallels, one can view our work as an attempt to design *verification conditions* for parsers: as verification conditions capture the adherence of a program to a property, our encoding captures if a string is parsable by a given parser. Like verification conditions have found extensive applications in verification [9, 20], debugging [6, 16, 25], repair [6, 27] and testing [10, 12] of programs, we believe that our encoding can borrow from these ideas (for programs) to solve interesting problems in parsing.

6 Conclusion

In this work, we propose an end-to-end symbolic algorithm for LL(k) parsing, and demonstrate its utility by building two applications, viz. parser synthesis and repair of token sequences. We draw parallels of our work with the design of *verification conditions* for programs; we hope to leverage existing ideas that use verification conditions for program analysis tools for using our encoding for building interesting tools for parsing.

References

1. Is this language LL(1) parseable? http://cs.stackexchange.com/questions/3350/is-this-language-ll1-parseable, 2012. Online; accessed 25 January 2017.
2. Left-Factoring a grammar into LL(1). http://cs.stackexchange.com/questions/4862/left-factoring-a-grammar-into-ll1, 2012. Online; accessed 25 January 2017.
3. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
4. Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.
5. Roland Axelsson, Keijo Heljanko, and Martin Lange. Analyzing context-free grammars using an incremental sat solver. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part II*, ICALP '08, pages 410–422, Berlin, Heidelberg, 2008. Springer-Verlag.
6. Rohan Bavishi, Awanish Pandey, and Subhajit Roy. To be precise: Regression aware debugging. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 897–915, New York, NY, USA, 2016. ACM.
7. John C. Beatty. On the relationship between ll(1) and lr(1) grammars. *J. ACM*, 29(4):1007–1022, October 1982.
8. Charles N. Fischer. LL(1) Grammars. http://pages.cs.wisc.edu/ fischer/cs536.f13/lectures/f12/Lecture22.4up.pdf, 2012. Online; accessed 24 January 2017.
9. Edmund Clarke, Daniel Kroening, and Flavio Lerda. *A Tool for Checking ANSI-C Programs*, pages 168–176. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
10. Przemyslaw Daca, Ashutosh Gupta, and Thomas A. Henzinger. Abstraction-driven concolic testing. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, VMCAI 2016, pages 328–347, New York, NY, USA, 2016. Springer-Verlag New York, Inc.
11. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
12. M. Gao, L. He, R. Majumdar, and Z. Wang. Llsplat: Improving concolic testing by bounded model checking. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 127–136, Oct 2016.
13. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News*, 32(1):60–65, March 2001.
14. Rahul Jain, Sanjeev Kumar Aggarwal, Pankaj Jalote, and Shiladitya Biswas. An interactive method for extracting grammar from programs. *Softw. Pract. Exper.*, 34(5):433–447, April 2004.
15. F. Javed, B. R. Bryant, M. Črepinšek, M. Mernik, and A. Sprague. Context-free grammar induction using genetic programming. In *Proceedings of the 42Nd Annual Southeast Regional Conference*, ACM-SE 42, pages 404–405, New York, NY, USA, 2004. ACM.
16. Manu Jose and Rupak Majumdar. Cause clue clauses: Error localization using maximum satisfiability. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 437–446, New York, NY, USA, 2011. ACM.

17. Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. Semi-supervised verified feedback generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 739–750, New York, NY, USA, 2016. ACM.
18. Keshav Pingali. Top-down parsing. https://www.cs.utexas.edu/ pingali/CS375/2010Sp/lectures/LL1.pdf, 2010. Online; accessed 24 January 2017.
19. L van Zijl. LL(1) parsing. http://www.cs.sun.ac.za/rw711/lectures/lec4/l4.pdf. Online; accessed 24 January 2017.
20. Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. *A Solver for Reachability Modulo Theories*, pages 427–443. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
21. R. Lämmel and C. Verhoef. Semi-automatic grammar recovery. *Softw. Pract. Exper.*, 31(15):1395–1448, December 2001.
22. Ralf Lämmel and Chris Verhoef. Cracking the 500-language problem. *IEEE Softw.*, 18(6):78–88, November 2001.
23. Alan Leung, John Sarracino, and Sorin Lerner. Interactive parser synthesis by example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 565–574, New York, NY, USA, 2015. ACM.
24. P. M. Lewis, II and R. E. Stearns. Syntax-directed transduction. *J. ACM*, 15(3):465–488, July 1968.
25. Yongmei Liu and Bing Li. Automated program debugging via multiple predicate switching. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI'10, pages 327–332. AAAI Press, 2010.
26. Ravichandhran Madhavan, Mikaël Mayer, Sumit Gulwani, and Viktor Kuncak. Automating grammar comparison. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 183–200, New York, NY, USA, 2015. ACM.
27. Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 448–458, Piscataway, NJ, USA, 2015. IEEE Press.
28. Marjan Mernik, Goran Gerlič, Viljem Žumer, and Barrett R. Bryant. Can a parser be generated from examples? In *Proceedings of the 2003 ACM Symposium on Applied Computing*, SAC '03, pages 1063–1067, New York, NY, USA, 2003. ACM.
29. Katsuhiko Nakamura. Incremental learning of context free grammars by bridging rule generation and search for semi-optimum rule sets. In *Proceedings of the 8th International Conference on Grammatical Inference: Algorithms and Applications*, ICGI'06, pages 72–83, Berlin, Heidelberg, 2006. Springer-Verlag.
30. Katsuhiko Nakamura and Takashi Ishiwata. Synthesizing context free grammars from sample strings based on inductive cyk algorithm. In *Proceedings of the 5th International Colloquium on Grammatical Inference: Algorithms and Applications*, ICGI '00, pages 186–195, London, UK, UK, 2000. Springer-Verlag.
31. Katsuhiko Nakamura and Masashi Matsumoto. Incremental learning of context free grammars. In *Proceedings of the 6th International Colloquium on Grammatical Inference: Algorithms and Applications*, ICGI '02, pages 174–184, London, UK, UK, 2002. Springer-Verlag.
32. Katsuhiko Nakamura and Masashi Matsumoto. Incremental learning of context free grammars based on bottom-up parsing and search. *Pattern Recognition*, 38(9):1384 – 1392, 2005. Grammatical Inference.

33. Robin Cockett . LL(1) grammars and predictive top-down parsing. http://pages.cpsc.ucalgary.ca/ robin/class/411/LL1.2.html, 2002. Online; accessed 24 January 2017.
34. Diptikalyan Saha and Vishal Narula. Gramin: A system for incremental learning of programming language grammars. In *Proceedings of the 4th India Software Engineering Conference*, ISEC '11, pages 185–194, New York, NY, USA, 2011. ACM.
35. A. Sellink and C. Verhoef. Generation of software renovation factories from compilers. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pages 245–255, 1999.
36. Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 15–26, New York, NY, USA, 2013. ACM.
37. Stephen J. Allan. LL Parsing. http://digital.cs.usu.edu/ allan/Compilers/Notes/LLParsing.pdf. Online; accessed 24 January 2017.
38. Andre Suelflow, Goerschwin Fey, Roderick Bloem, and Rolf Drechsler. Using unsatisfiable cores to debug multiple design errors. In *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*, GLSVLSI '08, pages 77–82, New York, NY, USA, 2008. ACM.
39. Matej Črepinšek, Marjan Mernik, Barrett R. Bryant, Faizan Javed, and Alan Sprague. Inferring context-free grammars for domain-specific languages. *Electron. Notes Theor. Comput. Sci.*, 141(4):99–116, December 2005.
40. Matej Črepinšek, Marjan Mernik, Faizan Javed, Barrett R. Bryant, and Alan Sprague. Extracting grammar from programs: Evolutionary approach. *SIGPLAN Not.*, 40(4):39–46, April 2005.
41. Matej Črepinšek, Marjan Mernik, and Viljem Žumer. Extracting grammar from programs: Brute force approach. *SIGPLAN Not.*, 40(4):29–38, April 2005.

Appendix

A Background: LL(k) Parsing

The classical LL(k) parser [24] uses a *top-down* parsing algorithm, i.e., starting from the start symbol, the parser attempts to construct a parse tree such that its leaves, reading from left to right match the subject string. The algorithm implements an efficient parser by avoiding backtracking using *predictive* parsing—At each step of the parse, the parser consults a (carefully constructed) *parse table* to select a production rule to expand a (non-terminal) node in the parse tree. An LL(k) parser refers to parsing an input string **L**eft to right using **k** tokens *lookahead* (for prediction) in the subject string, constructing the **L**eftmost derivation parse tree.

The LL(k) parsing algorithm The LL(k) parser maintains a *parsing stack* of symbols, initailized to contain only the start symbol S. The parser also maintains an input pointer to maintain the next terminal to be read in the subject string, which is initialized to the first symbol in this string. The parser takes a *step* in the parsing process by popping off the topmost symbol from the stack:

– If the symbol is a non-terminal, it consults the parsing table with a lookahead of k tokens to get a *prediction* of the production rule to use; the body of this

rule is then pushed on the stack in the *reverse* order. If the parse table fails to predict a rule, the parser signals an error.
- If symbol is a terminal, it matches the terminal with the current input symbol: if they match, it increments the input pointer; otherwise, an error is signaled.

The parser considers a parse successful if the input pointer reaches the end of the string and when it does, the parsing stack is also empty.

Parse table construction, first and follow sets As one can imagine, the *magic* of LL parsing is hidden in the construction of the parse table. The construction of this table is driven using two sets, namely the *first* and *follow* sets. For a sentential form α, the first set denotes the set of those terminals which can be the first terminals to appear in a string derived from α. The follow set for a non-terminal X is the set of all terminals that can follow X in any leftmost derivation from the start symbol. Simply put, an LL(1) parse table predicts a rule $X \rightarrow \alpha$ to be used with a lookahead terminal a for expanding X if $a \in First(\alpha)$, or if $\epsilon \in First(\alpha)$ and $a \in Follow(X)$

Parse table conflicts The successful construction of an LL(k) parse table is the certificate for the grammar being LL(k). A parse table construction fails if at any point it becomes possible for the parse table to propose the use of multiple production rules for the same $\langle X, \alpha \rangle$ pair, where X is the non-terminal candidate for expansion and α is the current lookahead terminal; In such situations the parse table is referred to as having a *conflict* and the grammar is declared to be beyond LL(k).

LL(k) Grammars If a parse table has conflicts, it implies that the grammar is not LL(k); however, the language may still be LL(k) parsable as there may exist an LL(k) grammar equivalent to this language. Unfortunately, inferring if there exists an LL(k) grammar for a given language is undecidable. Therefore, it is understandable that though there exist common transformation techniques like substitution, left factoring and removal of left-recursion [4], still, many times, inferring an LL(K) grammar for an arbitrary language is a difficult proposition (illustrated in Figure 1) – this is exactly what motivated our current work!

B Constraints for membership in First and Follow sets

The first sets can be found by computing the **minimum** fixpoint over a set of constraints shown below:

1. if $t \in \mathcal{T}$, then $t \in First(t)$
2. if $\langle X \rightarrow \epsilon \rangle \in \mathcal{P}$, then $\epsilon \in First(X)$
3. if $\langle X \rightarrow Y_1 Y_2 \ldots Y_k \rangle \in \mathcal{P}$, then
   (a) if $\exists i$ such that $a \in First(Y_i) \wedge \Pi_{j<i} \epsilon \in First(Y_j)$ then, $a \in First(X)$
   (b) if $\Pi_{i \in \{1..k\}} \epsilon \in First(Y_i)$ then, $\epsilon \in First(X)$

The follow sets can be found by computing the **minimum** fixpoint over a set of constraints shown below:

1. $\$ \in Follow(S)$, where S is the start symbol
2. If $\langle A \rightarrow \alpha B \beta \rangle \in P$, $t \in First(\beta) \implies t \in Follow(B)$
3. If $\langle A \rightarrow \alpha B \rangle \in P$, $t \in Follow(A) \implies t \in Follow(B)$

4. If $\langle A \rightarrow \alpha B \beta \rangle \in P$ and $\epsilon \in \text{First}(\beta)$, then $t \in Follow(A) \implies t \in Follow(B)$

C Soundness of our symbolic encoding

Note that the statement "the parseArray is *filled* with some symbols" implies that the execution of algorithm 3 has assumed some symbols to be present on certain indices in parseArray. An embedding of a partial expansion of a valid parse tree is a *preorder* traversal on the nodes of the partial parse tree (as demonstrated in figure 4, discussed in section 2).

Lemma 1 (Lookahead). *For any position i in the $parseArray$ which is filled by a symbol x, in the corresponding function call $LL1Parse(x, i, \_, a_i)$ in the execution of algorithm 3, a_i is indeed the lookahead index for the corresponding node to x in the parse tree in which parseArray can be embedded.*

Proof. We first show that the first call to $LL1Parse$ has the correct lookahead index for the symbol passed. The first call to $LL1Parse$ is at line 34, the arguments are $\langle S, \_, \_, 0 \rangle$. This shows that the first call has the correct lookahead for the symbol passed.

We now prove that assuming a function call $LL1Parse(x, i, \_, a_i)$ with the *correct* lookahead a_i, the symbol next to x in the preorder traversal of the parse tree (say y) gets the correct lookahead symbol in its function call. The symbol x can either be or not be a nonterminal.

In case $x \in \mathcal{N}$, then $y = Y_1$ in the rule $(r : x \rightarrow Y_1 \dots Y_n)$ (chosen nondeterministically at line 5) with which x is expanded. Note that this is the same rule with which x is expanded in the parse tree, since the check on line 7 uses the correct lookahead a_i (from our earlier assumption). Also, since $x \in \mathcal{N}$, the lookahead for y will be a_i. The lines 16 and 19 ensure this.

In case $x \notin \mathcal{N}$, then the return value li of the function call $LL1Parse(x, i, \_, a_i)$ contains the *correct* lookahead index for y. This is because in case $x \in \mathcal{T}$, lookahead for y is a_i+1, which is reflected in line 30. Similarly, in case $x = \epsilon$, lookahead for y is a_i, which is reflected in line 26.

Now we just have to show that the return value li is actually the same as the lookAheadIndex argument to the function call of y. In case y is x's sibling, line 21 ensures this. However, in case y is not x's sibling, lines 21 and 23 ensure this. This concludes the proof that y has the correct lookAheadIndex in its function call.

Lemma 2 (Subtree Containment). *In the function call LL1Parse(symbol, arrayBegin, arrayEnd, lookAhead), the subtree rooted at the node corresponding to symbol is filled up in the parseArray between arrayBegin and arrayEnd.*

Proof. We prove this by induction on the function call of $LL1Parse$. In the base case, $symbol \in \mathcal{T} \cup \{\epsilon\}$. If $symbol = \epsilon$, the statement holds vacuously, since there are no nodes in the subtree rooted at *symbol* to fill in the parseArray. If $symbol \in \mathcal{T}$, the statement again holds as the only symbol to fill in the subtree is *symbol*, which is placed at $arrayBegin$ (line 29).

If $symbol \in \mathcal{N}$, we observed that *symbol* is placed at $arrayBegin$ (line 3). Also, by induction hypothesis, the subtrees of any child Y_i of *symbol* is restricted

to $[p_i, p_{i+1} - 1]$ (line 19). Hence, by the relations on lines 9 and 10, we conclude that the subtrees of the children will be restricted to $[arrayBegin{+}1, arrayEnd]$. This concludes the proof.

Lemma 3 (Partial Correctness). *Any partially filled parseArray (during execution of algorithm 3) is an embedding of a partial expansion of a valid parse tree.*

Proof. We prove this lemma via the principle of mathematical induction. The induction hypothesis is as follows:

$P(k)$ = If the parseArray is filled with exactly k symbols during execution of algorithm 3, it is an embedding of a partial expansion of a valid parse tree

The base case of induction is $P(1)$, which occurs in the first call to $LL1Parse$ at line 34. The arguments are $\langle S, 0, parseArrayBound - 1, 0\rangle$. In the check on line 2, the if branch is taken, since $S \in \mathcal{N}$. Hence, on line 3 the symbol S is filled at index 0 in the parseArray. Since the single node S is a partial expansion of any (in fact, every) valid parse tree, the base case of induction holds.

Assuming $P(l) \forall l \leq k$, we now analyze $P(k+1)$. Consider the course of execution of algorithm 3 which results in a parseArray state with exactly $k+1$ symbols filled. In this state, consider any symbol x at the parseArray index i, such that either $x \in \mathcal{T}$, or $x \in \mathcal{N}$ and one of the following criteria holds with respect to the execution of the function call of $LL1Parse$ which filled x in the parseArray:

- line 19 has not been executed at all
- line 19 has been executed at least once, and none of the function executions in the recursion tree has resulted in a symbol being filled in the parseArray

Proving such an x exists is trivial, by following the recursive function calls. Because $P(k)$ holds, the parseArray state just prior to adding $x = parseArray[i]$ at line 3 is an embedding of partial expansion of a valid parse tree. Also, the parent of the x in the recursion tree (say the symbol p) has already been filled in parseArray. Some of the siblings (and their respective descendants) of x may have already been filled (due to the concurrency in the loop on line 18).

Let's suppose that in the function execution which filled p (henceforth referred to as *parent function execution*), where j is the position (j is *not* an index in the parseArray) of x in the body of the rule which is chosen to expand p. By lemma 1, lookAheadIndex for the parent function call is the same lookahead index as that of the parse tree node corresponding to p. Hence, by the constraint posed at line 7, the rule chosen non-deterministically at line 5 to expand p is indeed the rule which will give the valid subtree for p.

Since algorithm 3 only fills $s \in \mathcal{N} \cup \mathcal{T}$ in the parseArray, $x \in \mathcal{N} \cup \mathcal{T}$. In either case, x is placed at the index equal to the first argument ($arrayBegin$) provided to function call which fills x. The parent function execution guarantees that the value passed as the argument $arrayBegin$ for filling x is equal to p_j (line 19).

During the parent function execution, line 11 imposes ordering according to the preorder traversal in the positions that can be filled by x and its siblings. Moreover, the function call for any sibling (say, at position m in the rule) of

x takes the arguments $\langle Y_m, p_m, p_{m+1} - 1, a_m \rangle$. By lemma 2, the positions corresponding to the nodes in Y_l's subtree is restricted to $[p_m, p_{m+1})$. This, along with line 9 maintains the preorder traversal on placing x at $p_j = i$. By following similar reasoning over the ancestors of x, any other node in the partial parse tree does not disturb the preorder traversal embedding of the tree in the parseArray, on filling x at location i.

Hence, after x is filled at location i, the parseArray is an embedding of a partially expanded valid parse tree. Thus, $P(k+1)$ holds, concluding the proof.

Lemma 4 (Progress). *If the (possibly partial) parse tree implied by the current parseArray can expand further (i.e. there exists one leaf node which is a non-nullable non-terminal, for the respective lookahead), then in a terminating execution, algorithm 3 will fill up at least one more symbol.*

Proof. Consider any leaf node in the partial parse tree, x such that $x \in \mathcal{N}$. There will be some rule $(r : x \rightarrow Y_1 Y_2 \dots Y_n)$ (non-deterministically chosen at line 5) which x will further expand with. By the premise of the lemma, at least one of $Y_j \neq \epsilon$. Since the recursive call to $LL1Parse(Y_j, p_j, p_{j+1} - 1, a_j)$ in line 19 has not been executed, this function call will place a symbol in the parseArray, in particular the symbol Y_j.

Lemma 5 (Non-overlap). *Whenever the algorithm 3 fills a symbol in parseArray, it does so at an index which is not filled up by a symbol.*

Proof. The algorithm 3 only fills a symbol in parseArray at lines 3 and 29. In both these cases, the return value $pe \geq arrayBegin + 1$ (by lines 11, 12 and 30). Since x is placed at the location $arrayBegin$, by lemma 2 and lines 11 and 19, x will not overlap with any of its siblings or their descendants. By following similar reasoning over the ancestors of x, we conclude that x does not overlap with any of the symbols already present in the parseArray.

Lemma 6 (Compactness). *In the (possibly partial) parse tree implied by the current parseArray, if any subtree has only terminals or ϵ in its leaf nodes, its corresponding subarray in parseArray will be contiguous, starting from the index passed as the arrayBegin argument to the function call for the root of the subtree.*

Proof. Observe that any non-ϵ symbol x is filled at the location p_x in parseArray, where p_x was the second argument to the function call of $LL1Parse$ corresponding to x (lines 3 and 29). Hence, the symbol in the root node (say y) of the *fully expanded* subtree will be filled at the position $arrayBegin$ passed as an argument to the function call for y.

We now prove that assuming a function call $LL1Parse(x, i, \_, \_)$, the symbol next to x in the preorder traversal of the parse tree (say z) gets the argument $arrayBegin = i + 1$ (in case $x \neq \epsilon$) or $arrayBegin = i$ (in case $x = \epsilon$) . The symbol x can either be or not be a nonterminal.

In case $x \in \mathcal{N}$, then $z = Y_1$ in the rule $(r : x \rightarrow Y_1 \dots Y_n)$ (chosen non-deterministically at line 5) with which x is expanded. Note that this is the same rule with which x is expanded in the parse tree, since the check on line 7 uses the correct lookahead a_i (from our earlier assumption). Also, since $x \in \mathcal{N}$, the parseArray index for z will be $i + 1$. The lines 12 and 19 ensure this.

In case $x \notin \mathcal{N}$, then the return value pe of the function call $LL1Parse(x, i, \_, \_)$ contains the *correct* lookahead index for z. This is because in case that $x \in \mathcal{T}$, parseArray index for z is $i + 1$, which is reflected in line 30. Similarly, in case $x = \epsilon$, parseArray for z is i, which is reflected in line 26.

Now we just have to show that the return value pe is actually the same as the $arrayBegin$ argument to the function call of z. In case z is x's sibling, line 20 ensures this. However, in case z is not x's sibling, lines 20 and 23 ensure this. This concludes the proof that z has the correct parseArray index in its function call, and hence, concludes the proof of the lemma.

D Example for follow set constraints

Example: Continuing the example in Figure 6, we demonstrate our follow set constraints by assuming that the symbolic variables corresponding to the production rules are asserted with the grammar $\mathcal{G}_1$ (from figure 6). We use the notation F to denote $FollowSetWitness$

$F_{\langle 1,1,2\rangle}(A, v) = v$, since $v \in First(B)$ (from the first case in 7). From rule 3, we get $F_{\langle 3,1,0\rangle}(D, v) = A$ and
$F_{\langle 3,2,0\rangle}(E, v) = A$. We further get the following witness relations for the terminal a:

- $F_{\langle 2,1,2\rangle}(C, a) = a$ (since $a \in First(A)$)
- $F_{\langle 6,2,0\rangle}(S, a) = C$
- $F_{\langle 1,2,0\rangle}(B, a) = S$
- $F_{\langle 1,1,0\rangle}(A, a) = S$
- $F_{\langle 3,1,0\rangle}(D, a) = A$
- $F_{\langle 3,2,0\rangle}(E, a) = A$

Now, we simulate Algorithm 2 by analyzing $FollowSet(D, a)$. In the first iteration of the loop on line 3, we have $y = D$. Let us assume that the solver non-deterministically selects the rule $r = (3 : A \rightarrow DE)$ on line 4. Let us also assume that the solver non-deterministically selects $i = 1$ and $j = 2$ to explore a path via the second symbol in the body i.e. E for inducing a in the follow set of the first symbol in the body i.e. D, leading to $Y_i = D = y$, thus satisfying the assumption on line 7. Hence, at the end of first iteration on line 8, $y \leftarrow FollowSet_{\langle 3,1,0\rangle}(D, a) = A$.

In the second iteration of the loop, we attempt to extend this path from $y = A$. Again, assuming non-deterministic selections $r = 1$, $i = 1$ and $j = 0$, we get $y \leftarrow FollowSet_{\langle 1,1,0\rangle}(A, a) = S$. In the third iteration, assuming the selections of $\langle r, i, j\rangle = \langle 6, 2, 0\rangle$, we get $y \leftarrow FollowSet_{\langle 6,2,0\rangle}(S, a) = C$. In the fourth iteration, assuming the selections of $\langle r, i, j\rangle = \langle 2, 1, 2\rangle$, we get $y \leftarrow FollowSet_{\langle 2,1,2\rangle}(C, a) = a$. In subsequent iterations of the loop, the update operation is $y \leftarrow FollowSet_{\langle \_,\_,\_\rangle}(a, a) = a$ from figure 7. Here, we reach a fixpoint and exit the loop with $y = a$. This satisfies the assumption on line 10 and the program reaches a successful termination, thereby validating that $a \in FollowSet(D)$ with the witness path $[\langle 3, 1, 2\rangle, \langle 1, 1, 0\rangle, \langle 6, 2, 0\rangle, \langle 2, 1, 2\rangle]$.