Formal Methods in System Design
https://doi.org/10.1007/s10703-023-00420-3

Symbolic encoding of LL(1) parsing and its applications

Pankaj Kumar Kalita[1] · Dhruv Singal[2] · Palak Agarwal[3] · Saket Jhunjhunwala[4] · Subhajit Roy[1]

Received: 22 April 2022 / Accepted: 25 January 2023

Abstract
Parsers are omnipresent in almost all software systems. However, an operational implementation of parsers cannot answer many "how", "why" and "what if" questions, why does this parser not accept this string, or, can we have to parser for it to accept a set of strings? To lift the parsing theory to answer such questions, we build a *symbolic encoding* of parsing. Our symbolic encoding, that we referred to as a *parse condition*, is a logical condition that is satisfiable if and only if a given string w can be successfully parsed using a grammar $\mathcal{G}$. We build an SMT encoding of such parse conditions for LL(1) grammars and demonstrate its utility by building three applications over it: *automated repair of syntax errors* in Tiger programs, *automated parser synthesis* to automatically synthesize LL(1) parsers from examples, and automatic *root cause analysis of parser construction* to debug errors in parse tables. We implement our ideas into a tool, Cyclops , that successfully repairs 80% of our benchmarks (675 buggy Tiger programs), clocking an average of 30 s per repair, synthesize parsers for interesting languages from examples, and ranks the ground truth as the topmost ranked reason for failure in more than 85% of our instances. On our deployment of Cyclops in a compiler design course, 91 of 128 students gave a positive response stating that assistance from Cyclops was indeed useful.

Keywords Parse condition · LL1 parser · First set · Follow set · Parse table

✉ Pankaj Kumar Kalita
pkalita@cse.iitk.ac.in

✉ Subhajit Roy
subhajit@iitk.ac.in

Dhruv Singal
singaldhruv.ds@gmail.com

Palak Agarwal
palak8669@gmail.com

Saket Jhunjhunwala
saket.sj55@gmail.com

[1] Computer Science and Engineering, Indian Institute of Technology Kanpur, Kanpur, India

[2] Columbia University, New York, USA

[3] Google, Stockholm, Sweden

[4] Tower Research Capital, Gurugram, India

Published online: 22 June 2023

1 Introduction

Parsing algorithms answer to *membership queries*: does a given string w belong to the language described by a grammar $\mathcal{G}$. Though context-free grammars allow polynomial-time parsing algorithms, parsing is still expensive in practice. Interestingly, certain subsets of context-free grammars allow efficient parsers; many of these *efficiently parseable grammars*, like LL(1), LR(1), and LALR, are expressive enough to capture the constructs of realistic programming languages [5].

However, writing and debugging parsers remains an arduous activity—even for experienced language designers [49]. The difficulty stems from the fact that efficiently parseable grammars lack a simple syntactic identity: given a context-free grammar, it is not possible to deduce it to be LL(1) simply by examining its structure—a grammar must successfully pass the test of yielding a *parse table* to be deemed as efficiently parseable.

Consider the LL(1) family of grammars: inferring whether an LL(1) grammar exists for a given language is undecidable. Hence, it is usually difficult to design programming language syntax that keeps the language LL(1) parseable. Not just programming language designers, but also students of courses in Compiler Theory struggle when exposed to LL parsing algorithms [98, 99]; some interesting queries from students on this topic can be found on *StackExchange* [98, 99].

In this work, we propose *parse conditions*—a new symbolic encoding for efficiently parseable grammars: given a context-free grammar $\mathcal{G}$ and an input string w, a *parse condition* is a logical formula that is satisfiable if and only if: (1) $\mathcal{G}$ is efficiently parseable, and (2) $\mathcal{G}$ accepts w. We build our ideas into a tool, Cyclops , that builds and dispatches parse conditions for LL(1) parsers to Satisfiability Modulo Theory (SMT) solvers. The encoding of the *parse condition* in Cyclops imbibes an end-to-end symbolic algorithm for syntax analysis—including an encoding of an LL grammar, the parse table and the actions needed to be taken by the parser to accept the subject string.

Further, to demonstrate the practical utility of *parse conditions*, we apply Cyclops to three use-cases:

- *Automated Parser Synthesis* Cyclops can be used to generate LL(1) parsers (and grammars)—automatically—from a set of example strings in the language (§ 7);
- *Repair of Syntax Errors* Given a set of token sequences that fail to parse on a grammar, Cyclops can synthesize syntactic repairs to allow successful parsing (§ 6);
- *Computer Science Education* The symbolic encoding in Cyclops can encourage building of tools that assist students in understanding the complexities of LL(1) parsers; (§ 8).

We also envisage the use of Cyclops in other applications in the future, like for *domain-specific language (DSL) design*: DSL designers can use Cyclops for design exploration of *parseable* language constructs, to create languages or language extensions; Cyclops can ensure that the grammar necessarily remains LL(1) parseable and provides debugging support for conflicts (using formal artifacts like unsat cores and interpolants). Further, there are other pedagogical applications, like instructors can use such tools to automatically generate problem statements with certain properties (e.g., grammars that are LL(2) but not LL(1)).

Readers familiar with symbolic model-checking should be able to draw parallels between *verification conditions* [20, 31] and *parse conditions*: given a program $\mathcal{P}$ and a

property Φ, a verification condition encodes the logical constraints for $\mathcal{P} \vDash \Phi$. Like verification conditions have found immense applications in the verification, synthesis, and repair of programs, parse conditions can be instrumental in designing new analysis, synthesis, and repair algorithms for parsers.

Please note that *parse conditions* are not meant to facilitate parsing of a string (i.e., answering the membership query)—context-free languages allow parsing in polynomial time. Hence, one does not require SMT solving to drive parsing. This paper aims to attack problems around the *design of efficiently parseable grammars*. We conduct case studies and evaluations to answer the following questions regarding parse conditions:

Utility Can parse conditions be used to build interesting applications? We answer this by building three interesting applications over Cyclops in § 6, § 7 and § 8;
Efficiency Is the encoding of parse conditions efficient enough for real-world applications? The applications build above do scale to practical instances. For example, our case study on repairing syntax errors for programs in the Tiger language [10] spans more than 90 productions (§ 6), and Cyclops successfully repairs 80% of our benchmarks, clocking an average of 30 s per repair.

We make the following contributions to this work:

- We propose *parse conditions* to logically encode syntax analysis;
- We develop a logical encoding of parse conditions and instantiate it into our tool Cyclops ;
- We demonstrate the practical utility of parse conditions and the efficiency of our encoding by building modules for **automated parser synthesis**, **repair of syntax errors**, and **computer science education**.

An initial version of this work appears in the conference proceedings of LPAR 2018 [96].

2 Background: LL(*k*) parsing

The classical LL(k) parser [67] uses a *top-down predictive* parsing algorithm: commencing at the start symbol, the parser attempts to construct a *parse tree* such that its leaves—read from left to right—match the subject string. The algorithm is efficient as it avoids backtracking; the parser progresses by making *predictions*: at each step of the parse, the parser consults a (carefully constructed) *parse table* to predict the "right" production rule to be used to expand a (non-terminal) node in the parse tree. An LL(k) parser consumes the input string from **L**eft to the right, using **k** tokens of the next set of symbols in the input stream (*lookahead*), to construct the **L**eftmost derivation from the start symbol.

Figure 1 shows an LL(1) grammar, and Fig. 2 shows the parse tree from a string "`l p r`". Table 2 shows the LL(1) parse table: the rows correspond to non-terminals, and the columns correspond to *lookahead* terminals. The LL(1) parser progresses by constructing the parse tree in a top-down fashion by consulting the parse table to select the production that should be used to "expand" a non-terminal with a certain lookahead terminal. For example, on encountering the non-terminal B and a lookahead symbol r, the parse table advises the parser to expand using the production $B \rightarrow r\ S$.

Fig. 1 $\mathcal{G}_1$

$$
\begin{aligned}
S &\rightarrow A\ B\ ^{[1]}\\
A &\rightarrow C\ p\ ^{[2]}\\
&\mid\ D\ E\ ^{[3]}\\
B &\rightarrow r\ S\ ^{[4]}\\
&\mid\ \epsilon\ ^{[5]}\\
C &\rightarrow l\ S\ ^{[6]}\\
D &\rightarrow \epsilon\ ^{[7]}\\
E &\rightarrow \epsilon\ ^{[8]}
\end{aligned}
$$

2.1 The LL(*k*) parsing algorithm

The class of languages recognized by LL(k) parsers is a strict subset of the class of context-free languages [5]. As LL(k) parsers do not need to backtrack, they are fast, and their expressivity allows them to be used to model the syntax of popular programming languages [5].

The LL(k) parser maintains a *parsing stack* of symbols, initialized 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 (lookahead); this pointer 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, the parser consults the parse 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. The parser signals an error if the parse table fails to predict a rule.
- If the symbol is a terminal, the parser matches this terminal with the current input symbol: if they match, it increments the input pointer; otherwise, an error is signaled.

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

At each step of the parse, the parser maintains a lookahead of k tokens to determine which production is used to expand the symbol on top of the stack. LL(k) parsers are *table* based parsers, in the sense that they use a parse table to select the production to use given a symbol and a lookahead.

2.2 Parse table construction, first and follow sets

The *magic* of LL parsing is hidden in the construction of the parse table. Construction of this table is driven by maintaining two sets, namely the *first* and *follow* sets. For a sentential form α, the first set denotes the set of all terminals that 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 single lookahead terminal a for expanding X if $a \in \mathit{First}(\alpha)$, or if $\epsilon \in \mathit{First}(\alpha)$ and $a \in \mathit{Follow}(X)$; an LL(k) parse table uses a lookahead of k terminals instead of a single terminal.

Table 1 First and follow set of grammar shown in Fig. 1

Non-term	First set	Follow set
S	$\{l, r, \epsilon\}$	$\{p, \$\}$
A	$\{l, \epsilon\}$	$\{r, p, \$\}$
B	$\{r, \epsilon\}$	$\{p, \$\}$
C	$\{l\}$	$\{p\}$
D	$\{\epsilon\}$	$\{r, p, \$\}$
E	$\{\epsilon\}$	$\{r, p, \$\}$

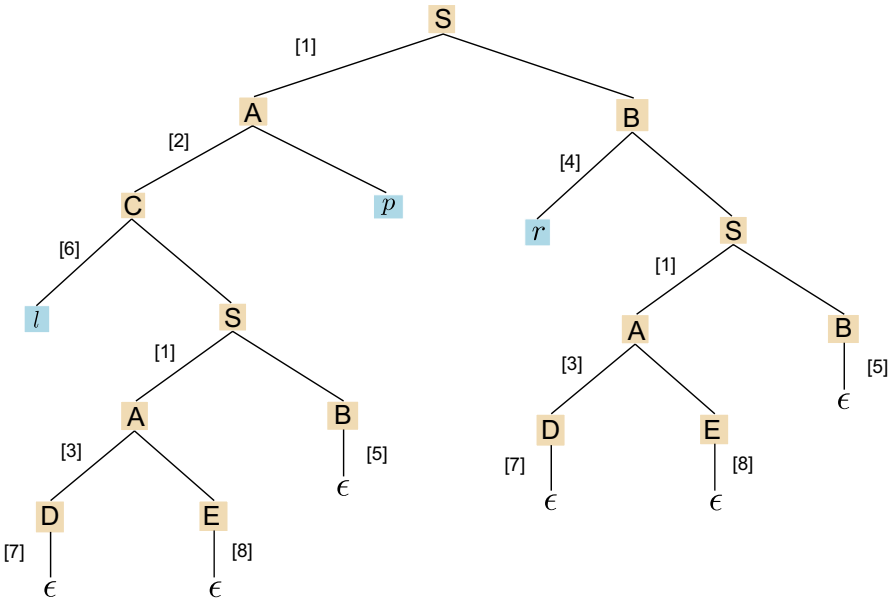


Fig. 2 Parse tree for string "`l p r`" for the grammar shown in Fig. 1

Table 2 Parse table for grammar shown in Fig. 1

Non Term	\$	p	r	l
S	$S \to A\,B$	$S \to A\,B$	$S \to A\,B$	$S \to A\,B$
A	$A \to D\,E$	$A \to D\,E$	$A \to D\,E$	$A \to C\,p$
B	$B \to \epsilon$	$B \to \epsilon$	$B \to r\,S$	
C				$C \to l\,S$
D	$D \to \epsilon$	$D \to \epsilon$	$D \to \epsilon$	
E	$E \to \epsilon$	$E \to \epsilon$	$E \to \epsilon$	

For Grammar shown in Fig. 1, the first and follow sets are shown in Table 1. If the first set of start symbol contains ϵ, then the language is *nullable* (i.e., it can derive an empty string).

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

(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. 3 Transforming a grammar to LL(1)

2.3 Parse table conflicts

The successful construction of the LL(k) parse table certifies that the grammar is LL(k). Parse table construction fails if, at any point, the parse table is unsuccessful at predicting the "right" (at most one) production that must be used for some $\langle X, \omega \rangle$ pair, where X is the non-terminal candidate for expansion, and ω is the current lookahead string (of k terminals). In such cases, the parse table is said to have a *conflict*, and the grammar is declared beyond LL(k).

An LL(1) parse table can encounter two kinds of conflicts:

- *First-first conflict* First-first conflict arises when there exists two rules $A \rightarrow \alpha_1$ and $A \rightarrow \alpha_2$ and $First(\alpha_1) \cap First(\alpha_2) \neq \emptyset$.
- *First-follow conflict* First-follow conflict arises for a non-terminal A if $First(A) \cap Follow(A) \neq \emptyset$ and A is nullable.

If a parse table has conflicts, it implies that the grammar is not LL(k); however, corresponding the *language* may still be LL(k) parseable. This implies that there may exist an LL(k) grammar that derives this language. Unfortunately, inferring whether there exists an LL(k) grammar for a given language is undecidable. Therefore, 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.

Figure 3 demonstrates the non-triviality of converging on an LL(k) grammar by attempting to infer an LL(1) grammar for arithmetic expressions. The first grammar (Fig. 3a) is 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' (Fig. 3b); substituting 'C' (Fig. 3c) and then 'A' (Fig. 3e) exposes the left recursion. Remove this left

recursion (Fig. 3e) now shows 'id' as a left factor for 'S.' Left factoring finally produces a grammar that is LL(1) (Fig. 3f).

3 An overview of parse condition

A context-free 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 $r \in \mathcal{P}$ is described by a tuple $\langle H, B \rangle$, where $H \in \mathcal{N}$ is the "head" of the rule and B is the "body" for the rule, described by a sequence of terminals and non-terminals ($\mathcal{T} \cup \mathcal{N} \cup \{\epsilon\}$). We assume that functions $\mathsf{head}(r)$ and $\mathsf{body}(r)$ return the head and body for rule r, respectively. We also assume $\mathsf{symbol}(r)$ to return the set of all the symbols, the head, as well as those that appear in the body of rule r.

Definition 1 **$\mathcal{B}$-Parse Condition**: A logical formula Ω is a $\mathcal{B}$-Parse Condition for a grammar $\mathcal{G}$ and a input string ω, Ω is satisfiable if and only if:

- The grammar $\mathcal{G}$ is a $\mathcal{B}$ grammar; correspondingly, there exists a valid $\mathcal{B}$ parser, say Λ, for $\mathcal{G}$;
- The parser Λ accepts the string ω.

In this paper, we discuss the encoding of **LL(1)-Parse Conditions**. An LL(1)-Parse Condition is satisfiable if and only if $\mathcal{G}$ is an LL(1) grammar, and the corresponding parser accepts ω. We refer to LL(1)-Parse Conditions simply as *Parse Conditions* subsequently.

The *parse condition* is composed of the following:

- $\mathcal{G}$, ω and λ: Let $\mathcal{G}$, ω, and λ be symbolic placeholders for a member from the space of all possible grammars, strings, and LL(1) parse tables, respectively. These appear as free variables in the encoding of the parse condition: hence, asserting one of them allows us to infer the others. For example, asserting a set of input strings allows us to infer an LL(1) grammar and a valid LL(1) parse table (grammar and parser synthesis); asserting a correct grammar allows one to infer/repair an input string (string sampling and repairing syntax errors).
- *ParseTable* (*ParseTable*$(\mathcal{G}, \lambda)$): This encodes the space of all consistent pairs $\langle \mathcal{G}, \lambda \rangle$, such that the grammar $\mathcal{G}$ produces a valid parse table λ. A satisfying solution to our parse table constraints ensures that an LL(1) parse table could be constructed (for the given grammar) without any parse table conflicts. We describe the *ParseTable* constraints in § 5.
- *Parser* (*Parser*$(\mathcal{G}, \lambda, \omega)$): This is a symbolic encoding of the LL(1) parsing algorithm; given a (symbolic) grammar $\mathcal{G}$, a (symbolic) parse table λ and a (symbolic) input string ω, *Parser*$(\mathcal{G}, \lambda, \omega)$ encodes the *steps* (parsing actions) taken by the parser en route to accepting ω. The core parsing algorithm is encoded as a set of these *parsing actions*. *Parser*$(\mathcal{G}, \lambda, \omega)$ is satisfiable if and only if the constraint system allows for constructing a valid parse tree for the target string (ω) under an LL(1) parse table (λ). We describe the *Parser* constraints in § 4.

Parser constraints are enforced for the first and follow sets to ensure that there are no *conflicts*.

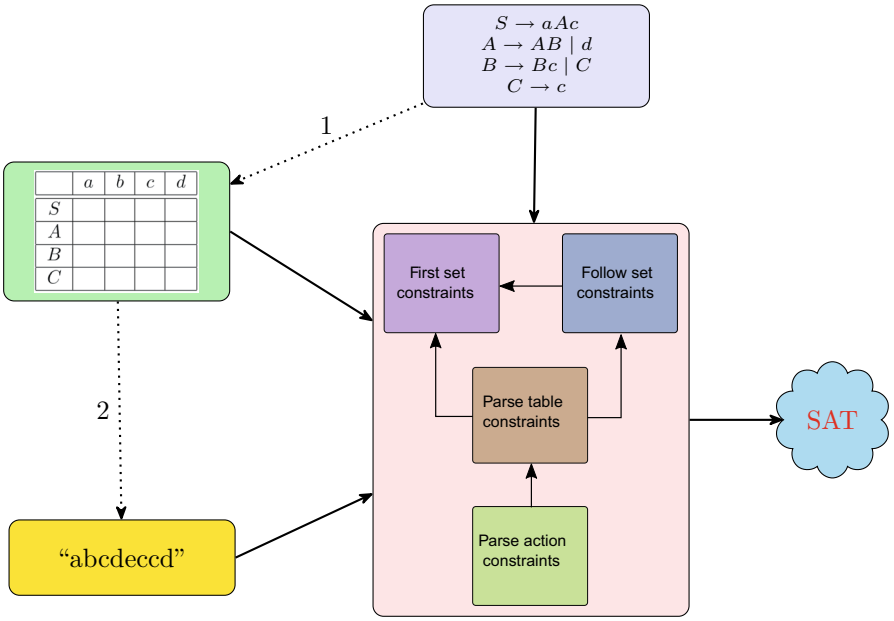


Fig. 4 Overview of CYCLOPS

- *First* and *Follow*: These encode the set of all first and follow sets allowed by $\mathcal{G}$, defined as fixpoint functions. We describe the first and follow set constraints in § 5.3 and § 5.4.
- *Conflict*: This encodes the condition that there is no first-first or first-follow conflict for the grammar $\mathcal{G}$ while creating the parse table λ. We describe the constraints for avoiding conflicts en route to building a valid parse table in § 5.5.

A high-level encoding for *parse condition* is shown below: the parse condition on a grammar ($\mathcal{G}$) and a string (ω) is satisfiable if and only if there exists a *valid* LL(1) parse table, λ, (λ satisfies $ParseTable(\mathcal{G}, \lambda)$) and there exist a derivation of λ from $\mathcal{G}$ (i.e., $Parser(\mathcal{G}, \lambda, \omega)$ holds). A parse table λ is *valid* if and only if it does not have a first-first or first-follow conflict ($ParseTable(\mathcal{G}, \lambda)$ holds).

$$ParseCondition(\mathcal{G}, \omega) \equiv \exists \lambda.\ ParseTable(\mathcal{G}, \lambda) \wedge Parser(\mathcal{G}, \lambda, \omega) \tag{1}$$

$$ParseTable(\mathcal{G}, \lambda) \equiv First(\mathcal{G}) \wedge Follow(\mathcal{G}) \wedge \neg Conflict_{LL(1)}(\mathcal{G}, \lambda) \tag{2}$$

CYCLOPS constructs a symbolic encoding for LL(1) in a fragment of first-order logic. The encoding is instantiated as a set of SMT constraints in the theory of linear integer arithmetic with uninterpreted functions.

Figure 4 shows the high-level overview of our system: the grammar ($\mathcal{G}$) and the target string (ω) appear as free variables in the encoding; asserting a target string allows one to generate an LL(1) grammar and parse table that accepts the string (see § 7); asserting a grammar allows one to generate a string (or repair a syntactically incorrect string) that is consistent with the grammar (see § 6). The arrows within the different modules in the parse conditions denote *dependence*: the follow set constraints depend on the first set

constraints, while the parse table constraints depend on both of them. The parse action constraints depend on the parse table constraints.

To allow for a bounded search, the grammar ($\mathcal{G}$) is constrained by a *grammar template* that contains placeholders (symbolic variables) for each rule in the grammar: the template is defined using hyperparameters for the size and shape of the production rules (maximum number of productions, terminals, non-terminals and the maximum size of any production). As we discuss in the following sections, we use a *parse array* to encode a parse tree as an embedding.

Though we restrict the discussion in this paper to LL(1) parsing, all the algorithms can be easily extended to the LL(k) case by using k terminals of lookahead instead of a single terminal. We describe our symbolic encoding of the LL(1) parsing algorithm in § 4 and our symbolic encoding of the LL(1) parse table (along with first and follow sets) in § 5.

4 Symbolic encoding of LL(1) parsing actions

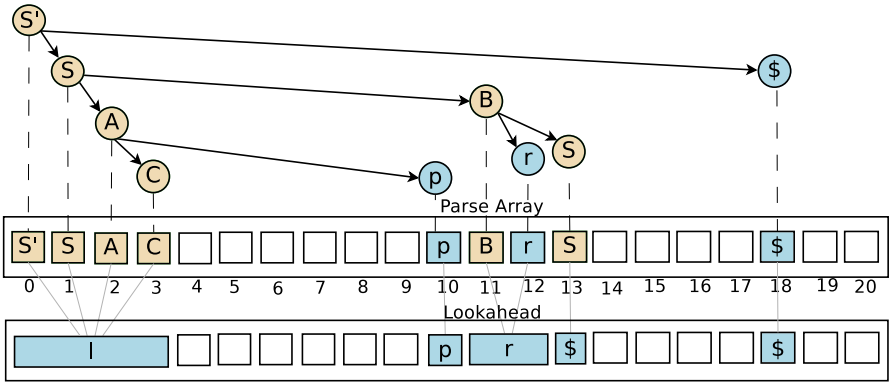
Parse condition is a symbolic encoding, serving applications other than parsing. Hence, instead of the classic LL(1) parsing algorithm, we had to design a quantifier-free first-order logic encoding that "simulates" an LL(1) parser on a *symbolic string*. As our algorithm builds a symbolic encoding, it operates differently than the classic stack-based LL(1) parser: for example, we simulate rule selection from the parse table fixpoint constraints using *witness encodings* and the parse tree as an embedding on a symbolic array.

We explain our first-order encoding—*operationally*—as a guess-and-check algorithm: each symbolic variable can be seen as making a "guess" (or a non-deterministic choice); our first-order formula (for the encoding) would be satisfied only when the "guesses" of all the symbolic variables are correct, i.e., the set of "guesses" satisfy all the constraints. Note that during the operational description, though we may refer to "reading" the string or "consulting" the parse table, the algorithm is actually operating over a set of symbolic objects—from the string being parsed to the parse table entries are all symbolic. The reader should see each *guess* by the algorithm as assignments that are (magically) all correct (as they are enforced by the constraint solver). For example, the formula $\exists x, y.\ x + y > 28$ can be seen as *guessing* values of x and y such that their sum exceeds 28; the "operational algorithm" represented by the formula will return a result only when the guesses are correct (i.e., when the formula is satisfiable); like $x = 12, y = 50$.

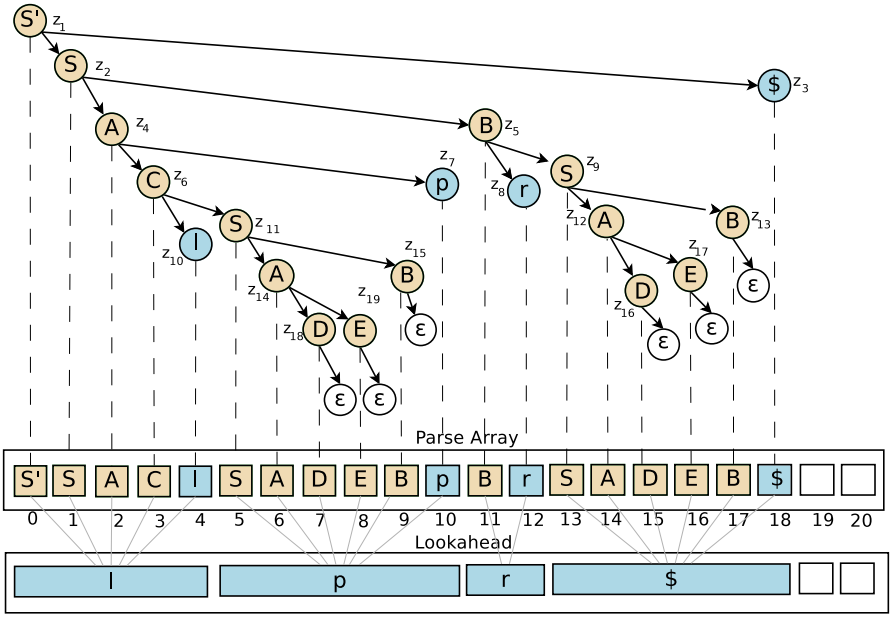
Our algorithm considers a parse *valid* if and only if we are able to construct a valid parse tree corresponding to the grammar $\mathcal{G}$ and the given string w. We consider a parse tree valid if and only if:

- The root node contains the start symbol;
- Each non-terminal is expanded using a production of $\mathcal{G}$ (as dictated by the parse table);
- Each non-terminal has children corresponding to the symbols appearing in one of its production rules;
- The leaves of the tree read from left to right, correspond to the input string.

To enable the use of an SMT solver to compute a parse tree, we define an *embedding* of a parse tree into a linear array. Our symbolic parsing actions operate on such a linear



(a) Intermediate stage



(b) Parsing complete

Fig. 5 Embedding of a parse tree

array that can be interpreted as corresponding operations on the parse tree. Figure 5 provides a visual insight into the embedding.

For the start symbol S, we add an additional production $S' \to S$ with the 'extended' start symbol S'; as this production is only applied once, the occurrence of '$' signals the end of a successful parse. We also assume that the input string w is terminated by the special symbol '$'.

4.1 Embedding of parse tree into a linear array

We define an *embedding* of a parse tree $\mathcal{Z}$ to a bounded size linear array $[\mathcal{A}_0, \ldots, \mathcal{A}_b]$ (where the array size is $b \geq \mid \mathcal{Z} \mid -1$) as follows:

- The root node of the parse tree is mapped to $\mathcal{A}_0$;
- For any node $z \in \mathcal{Z}$ with children $c_1, c_2, \ldots, c_m, 0 \leq index(z) \leq index(c_i) \leq b$ and $0 \leq index(c_i) \leq index(c_m) \leq b$, for each $i < m$ (where $index(\mathcal{A}_i) = i$);
- If $\mathcal{A}_i = h$, then $\forall_{l=i}^{b} A_l = h$ (where h is an additional node introduced in the parse tree, referred to as *hole*, shown via empty boxes in Fig. 5).

That is, $\mathcal{A}_i$ 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 *preorder* depth-first traversal of the parse tree. The ϵ symbol nodes are not mapped. All elements $\mathcal{A}_i, i > \mid \mathcal{Z} \mid$ are mapped to a special (artificial) element, *hole*.

Therefore, the *embedding* is a sequence of parse tree nodes as they appear in the linear array. For example, in Fig. 5, we get a valid embedding (nodes numbered breadth-first) $[z_1, z_2, z_4, z_6, z_{10}, z_{11}, z_{14}, z_{18}, z_{19}, z_{15}, z_7, z_5, z_8, z_9, z_{12}, z_{16}, z_{17}, z_{13}, z_3, z_h, z_h]$ where z_h is an artificially introduced *hole*. Intuitively, the holes represent "unused" array slots.

4.2 Intuition on the embedding

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(1) parsing algorithm. Hence, we can use this embedding to mimic the LL(1) parsing algorithm while eliminating the need to maintain a symbolic stack at each *step* of the parser.

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

We demonstrate the process using the example in Fig. 5 and the grammar in Fig. 1. 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} step occupies $\mathcal{A}_{i-1}$.

Parsing commences with the 'extended' start symbol S' being the first symbol to be pushed and popped off the stack (in the first step). Hence, S' gets placed at $\mathcal{A}_0$. Next, (like in the classical parser), we ask the parse table to *guess* a rule; in this case, it would guess the production rule for the extended start, $S' \rightarrow S$. We, next, *guess* for the step number when each of the symbols in the rule body will be popped off the stack (when running the classical parser): it is obvious that S 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 \$ will be popped off at the 19^{th} step; hence, we place this symbol at $\mathcal{A}_{18}$. We, similarly, *guess* the rule to expand S: let us assume that we select $S \rightarrow AB$ and *guess* the positions $\mathcal{A}_2$ for A and $\mathcal{A}_{11}$ for B. Similarly, the other symbols get placed in the array, each non-terminal placed according to the *guesses* of the positions of the symbols (in the body of the production).

If our parsing array indeed represents an embedding of a valid parse tree (Fig. 5 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

$$\forall_{0 \le i \le |\texttt{ParseArray}|}.(i < \texttt{prefixLimit}) \implies \mathsf{ValidAction}(i) \tag{3}$$

$$\begin{aligned}\mathsf{ValidAction}(i) = [\{\mathsf{ParseArray}(i) \in \mathcal{T}\} \wedge \{\mathsf{ParseArray}(i) = \mathsf{InputString}(\mathsf{ip}(i))\} \\ \wedge \{\mathsf{ip}(i+1) = \mathsf{ip}(i) + 1\}] \vee \\ [\{\mathsf{ParseArray}(i) \in \mathcal{N}\} \wedge \\ \{\exists r\{\exists i_1, i_2, \ldots, i_n . i \le i_1 \le i_2 \ldots i_n. \ \ \mathsf{ApplyProd}(r, i, i_1, \ldots, i_n)\} \\ \wedge\{\mathsf{ip}(i+1) = \mathsf{ip}(i)\} \wedge\{\mathsf{ParseTable}(\mathsf{head}(r), \mathsf{InputString}(\mathsf{ip}(i))) = r\}\}]\end{aligned} \tag{4}$$

$$\begin{aligned}\mathsf{ApplyProd}(r, i, i_1, \ldots, i_n) = \bigvee_{k \in \mathcal{P}} [r = \langle k : X \to Y_1 \ldots Y_n \rangle \wedge \mathsf{end}(i) = \mathsf{end}(i_n) \\ \wedge \ i_1 = i + 1 \ \wedge (\mathsf{end}(i_n) < \mathsf{end}(0)) \ \wedge \\ \bigwedge_{l \in \{1 \cdots |\texttt{body}(r)|-1\}} \mathsf{ApplySymb}(Y_l, i_l, i_{l+1})]\end{aligned} \tag{5}$$

$$\mathsf{ApplySymb}(Y_l, i_l, i_{l+1}) = \begin{cases} (\mathsf{ParseArray}(i_l) = Y_l) \wedge & \text{if } Y_l \in \mathcal{T} \cup \{\$\} \\ \quad (\mathsf{end}(i_l) = i_{l+1} = i_l + 1), & \\ (i_{l+1} = i_l), & \text{if } Y_l = \epsilon \\ (\mathsf{ParseArray}(i_l) = Y_l) \wedge \mathsf{end}(i_l) = i_{l+1}, & \text{if } Y_l \in \mathcal{N} \end{cases} \tag{6}$$

$$(\mathsf{ParseArray}(0) = S') \wedge (\mathsf{ip}(0) = 0) \wedge (\mathsf{end}(0) < |\ \mathsf{ParseArray}\ |) \tag{7}$$

Fig. 6 The parse action constraints

constraints, and we use an SMT solver to infer the "correct" guesses that produce a valid embedding.

4.3 The encoding of the parsing algorithm

The following are the relevant functions that are used by our constraint system (Fig. 6):

- ParseArray: the linear embedding of the parse tree;
- InputString: the string to be parsed;
- ip: position of the input pointer in InputString at the i^{th} step of parsing; the lookahead symbol corresponding to the i^{th} location in the ParseArray is InputString(ip(i));
- end(i): the last location in ParseArray containing the embedding of the subtree with root at ParseArray(i);
- ParseTable(X, a) (see § 5) – it provides the production rule to be used to expand the non-terminal X when a as the current lookahead symbol;
- prefixLimit: marks the end of parsing. For the extended grammar, prefixLimit is encoded as the first occurrence of '$' in ParseArray that leads to a successful parse.

To begin with, we apply the predicate ValidAction(i) for each cell in the parsing array to ensure that every step taken by the parser is valid (Eq. 3); in other words, it ensures

that the parse array is a valid embedding of a parse tree. Note the difference between |ParseArray| and prefixLimit: |ParseArray| denoting the size of ParseArray; is dictated by the user and denotes the maximum size of the parse tree that can be successfully "computed"; prefixLimit is a component of the solution to the constraint set, denoting the actual size of the successfully constructed parse tree (Eq. 3).

The predicate ValidAction works as follows (Eq. 4). Throughout the discussion, n indicates the maximum number of symbols in the body of any production rule (unequally sized rules are padded with ϵ); n is provided by the user as a hyperparameter.

- If the current symbol in ParseArray is a terminal: this terminal is "stored" in the current position in InputString and the input pointer is advanced;
- If the current symbol is a non-terminal: CYCLOPS "guesses" a rule $\langle r : X \rightarrow Y_1 \dots Y_n \rangle$ to expand in consultation with the *symbolic* ParseTable (we provide the parse table constraints in § 5.5), along with "start" positions in the ParseArray where the children parse trees corresponding to the symbols in the body of the selected production are to be embedded. The input pointer, ip, remains unmodified in this case.

  ApplyProd (Eq. 5) encodes the application of the selected rule $\langle r : X \rightarrow Y_1 \dots Y_n \rangle$ by setting constraints on each symbol in the body, Y_i, of the rule using ApplySymb —enforcing the following constraints (Eq. 6):

  – If Y_i is a terminal or the end-of-string marker (\$), the symbol is locked on the current location in ParseArray, and the start of the parse (sub)tree of the next symbol is constrained to start from the next slot;
  – If Y_i is a non-terminal, it is locked on the current location in ParseArray and the start of the parse (sub)tree of the next symbol is constrained to start from the next slot; the end of the current symbol is constrained to just before the start of the embedding of its sibling;
  – If Y_i is ϵ, the start of its next sibling is constrained to start from the current position (as ϵ-symbols are not recorded in the ParseArray).

Finally, we constrain the first element of ParseArray to the 'extended' start symbol S', the first element of ip to 0 (lookahead for the first parsing step), and constrain the parse tree to at most the size of the ParseArray to initiate the parsing process (Eq. 7).

Example Consider the LL(1) grammar $\mathcal{G}_1$ in Fig. 1. Assuming we have the correct ParseTable constraints for $\mathcal{G}_1$, we now walk through parsing of the string $lpr \in L(\mathcal{G}_1)$ (see Fig. 5b). We refer to the new production $r = \langle S' \rightarrow S\$ \rangle$ as rule 0. The rest of the rules are numbered in Fig. 1. The constraints are evaluated as follows:

- parseArray(0) == S' and ip(0) = 0 use $r = 0$ and set $i_S = 1$, $i_\$ = 18$ and ip(1) = 0 (due to ValidAction(0))
- ApplyProd(0, 0, 1, 18) enforces end(0) = end(18)
- ApplySymb(S, 1, 18) leads to parseArray(1) = S
- ApplySymb(\$, 18, _) causes parseArray(18) = and end(18) = 19
- parseArray(1) == S and ip(1) = 0 use $r = 1$ and set $i_A = 2$, $i_B = 11$ and ip(2) = 0 (due to ValidAction(1))
- ApplyProd(1, 1, 2, 11) enforces end(1) = end(11)
- ApplySymb(A, 2, 11) leads to parseArray(2) = A
- ApplySymb(B, 11, _) leads to parseArray(11) = B

- $\mathsf{parseArray}(11) == B$ and $\mathsf{ip}(11) = 2$ (set by some skipped steps) use $r = 4$ and set $i_v = 12, i_S = 13$ and $\mathsf{ip}(12) = 2$ (due to $\mathsf{ValidAction}(11)$)
- $\mathsf{ApplySymb}(r, 12, 13)$ leads to $\mathsf{parseArray}(12) = r$ and $\mathsf{end}(12) = 13$
- $\mathsf{parseArray}(12) == r$ and $\mathsf{ip}(12) = 2$ result in $\mathsf{ip}(13) = 3$ (due to $\mathsf{ValidAction}(12)$)

Completing the value assignments to other constraints results in the embedding of the parse tree as shown in Fig. 5b.

5 Symbolic encoding of the LL(1) parse table

We encode an LL(1) parse table as a function, $\mathsf{ParseTable}(X, t)$, that maps a non-terminal (X) and a lookahead terminal (t) to a production rule, or **error**. This encoding constrains the system to valid LL(1) parse tables such that a cell is occupied by at most one production.

5.1 Encoding least fixpoints as SMT constraints

We encode our fixpoint equations by explicitly unrolling the fixpoint iterations to a bounded depth. Consider computing a fixpoint solution for a function $f(x)$ over a lattice $(D, \sqsubseteq)$, where $\sqsubseteq$ is an ordering relation over the domain D; the fixpoint is computed by searching for an ascending chain that eventually stabilizes:

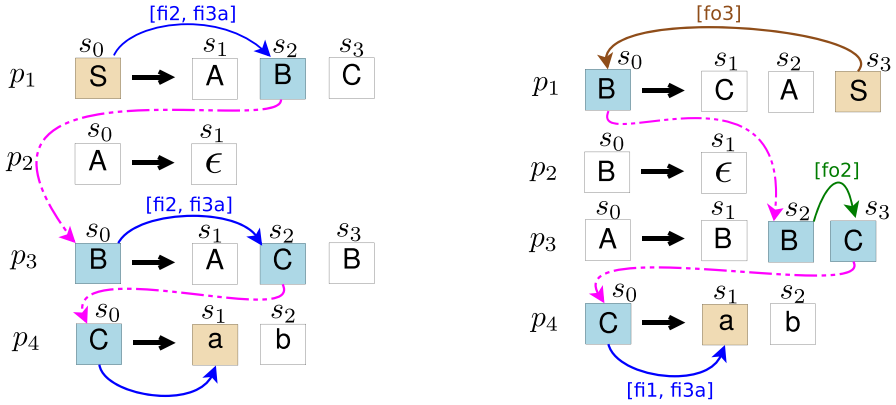
$$a_1 = f(a_0) \wedge a_2 = f(a_1) \wedge \ldots a_i = f(a_{i+1}) \wedge a_i = a_{i+1} \tag{8}$$

Any satisfying solution to this unrolling would certainly be a fixpoint solution—but not necessarily the least fixpoint! We use a fundamental result from Tarski's fixpoint theorem [102] that states that any such (ascending) chain is guaranteed to terminate at the *least* fixpoint if $a_0 = \bot$. Note that this encoding of least fixpoints for SMT solvers essentially generates a satisfying solution that corresponds to the *witness* $[a_0 = \bot, a_1, a_2, \ldots, a_i]$ for the chain that culminates to the least fixpoint. Intuitively, the witness encodes a sequence of *reasons* as to why a_i is the least fixpoint: as the element a_{i-1} was on the list (chain), and so on till we reach the bottom element $a_0 = \bot$. We use such (a sequence of) witnesses for computing the first and follow sets.

Do note that we need to encode the least fixpoint computation right in the parse condition—not simply compute a least fixpoint solution for a given set of constraints. For example, for parser synthesis, we search in the space of all grammars, and the first and follow set constraints may differ for each point in the search space. Hence, we pose it as a search for an ascending chain (from the bottom element) that eventually stabilizes.

5.2 Intuition of computing the first and follow sets as path reachability

We discussed the construction of *witnesses* above. Let us now provide some intuition as to how it enables us to provide a certificate for the first set and follow set membership relation. Moreover, we get the correct first and follow set relations by searching for a fixpoint solution over various witness predicates. In essence, the search for witnesses can simply be encoded as a search for a bounded path. We illustrate the intuition behind this process in the following examples.



(a) First set witnesses (b) Follow set witnesses

Fig. 7 First and follow set witnesses

5.2.1 Encoding construction of first sets as path reachability

The first sets are computed as the **least** fixpoints over these constraints:

(fi1) If $t \in \mathcal{T}$, then $t \in \mathit{First}(t)$
(fi2) If $\langle X \to \epsilon \rangle \in \mathcal{P}$, then $\epsilon \in \mathit{First}(X)$
(fi3) If $\langle X \to Y_1 Y_2 \ldots Y_k \rangle \in \mathcal{P}$, then

(fi3a) if $\exists i$ such that $a \in \mathit{First}(Y_i) \wedge \forall_{j<i} \epsilon \in \mathit{First}(Y_j)$, then $a \in \mathit{First}(X)$
(fi3b) if $\forall_{i \in \{1..k\}} \epsilon \in \mathit{First}(Y_i)$, then $\epsilon \in \mathit{First}(X)$

Let us show how a first set query can be formulated as a search for a *path*. Figure 7a shows a grammar where we attempt to answer whether $a \in \mathit{First}(S)$: we label a node in the path as $\langle p_r, s_i \rangle$, where p_r is an identifier for production, and s_i is the position of a symbol in the body of the production.

A (solid) edge, $X \to Y$, can be created between symbols X and Y, if and only if $\mathit{First}(Y) \subseteq \mathit{First}(X)$. Solid edges are annotated with the *reason* (with respect to the definition of the first set above) as to why the edge exists; for example, the edge $\langle p_1, s_0 \rangle \to \langle p_1, s_2 \rangle$ exists as A is nullable (case fi2) and all other non-terminals on the left of B are nullable (case fi3a). Clearly, if there exists a path $X_1 \to X_2 \to \cdots \to t$, where X_i are non-terminals and t is a terminal, $t \in \mathit{First}(X_1)$. The dotted edges are used to connect the same symbol across productions.

In the example, we can provide the *witness* path as a sequence $[\langle p_1, s_2 \rangle, \langle p_3, s_2 \rangle, \langle p_4, s_1 \rangle]$: as A is nullable, $\mathit{First}(B) \subseteq \mathit{First}(S)$ and $\mathit{First}(C) \subseteq \mathit{First}(B)$; finally, $a \in \mathit{First}(C)$ as a is the first terminal in the body. Note that $\langle p_1, s_2 \rangle$ and $\langle p_3, s_2 \rangle$ could be picked in the path only because A was nullable.

This gives an intuition as to how queries on the first sets can be reduced to reachability queries. We give details of this encoding in § 5.3.

5.2.2 Encoding construction of follow sets as path reachability

The follow sets are computed as the **least** fixpoints over these constraints:

(fo1) $\$ \in \mathit{Follow}(S)$, where S is the start symbol
(fo2) If $\langle A \to \alpha B \beta \rangle \in P, t \in \mathit{First}(\beta)$, then $t \in \mathit{Follow}(B)$
(fo3) If $\langle A \to \alpha B \rangle \in P, t \in \mathit{Follow}(A)$, then $t \in \mathit{Follow}(B)$
(fo4) If $\langle A \to \alpha B \beta \rangle \in P$ and $\epsilon \in \mathit{First}(\beta)$, then $t \in \mathit{Follow}(A) \implies t \in \mathit{Follow}(B)$

As the encoding for the first sets above, let us demonstrate how the same idea can also be applied to queries on follow sets. Figure 7b shows a grammar where we attempt to answer whether $a \in \mathit{Follow}(S)$: 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. A brown colored solid edge, $X \to Y$, can be created between symbols X and Y, if and only if $\mathit{Follow}(Y) \subseteq \mathit{Follow}(X)$. A green colored solid edge, $X \to Y$, can be created between symbols X and Y, if and only if $\mathit{First}(Y) \subseteq \mathit{Follow}(X)$. Each edge is annotated with the *reason* (with respect to the definition of first and follow sets) as to why the edge exists; for example, the edge $\langle p_3, s_2 \rangle \to \langle p_3, s_3 \rangle$ exists as the first if C is in the follow of B (case fo2). Clearly, if there exists a path $X_1 \to X_2 \to \cdots \to X_n$ and $t \in \mathit{First}(X_n)$, where X_i are non-terminals and t is a terminal, $t \in \mathit{Follow}(X_1)$. The dotted edges are simply used to connect the same symbol across productions. The blue colored edges refer to the first-set relation (these edges follow the semantics as in § 5.2.1).

In the provided example, we can produce the witness path as a sequence $[\langle p_1, s_3, s_0 \rangle, \langle p_3, s_2, s_3 \rangle]$: $\mathit{Follow}(B) \subseteq \mathit{Follow}(S)$ 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 $t \in \mathit{First}(C)$, then it can follow B and thus, $a \in \mathit{First}(C) \implies a \in \mathit{Follow}(S)$.

This gives an intuition as to how queries on follow sets can be reduced to reachability queries. We give details of this encoding in § 5.4.

5.3 Encoding of the first set constraints

The first sets of the non-terminals are computed as the **least** fixpoints over a set of constraints; we encode this problem as computing the least fixpoint on the lattice $(\mathcal{T} \cup \mathcal{N}, \sqsubseteq)$, where $\mathit{First}(Y_i) \sqsubseteq \mathit{First}(X)$ if and only if Y_i appears as the first symbol in a sentential form derived from X. We formulate it as generating a witness for an ascending chain that eventually stabilizes (as discussed above).

For the first set constraints, $t \in \mathit{First}(X)$, the witness generation is essentially a search for 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. We will refer to this path as the *witness* for $t \in \mathit{First}(X)$.

Figure 8 shows our encoding for the *witness* functions; these functions compute a symbol that acts as a link in the sequence of reasons to form the witness for the relation $t \in \mathit{First}(X)$. The core constraints use two helper functions:

$$\mathsf{FirstSetWitness}_{\langle r,i\rangle}(X,t) = \begin{cases} t, & \text{if } Y_i = t \in \mathcal{T} \wedge \forall_{k=1}^{i-1} \mathsf{FirstSet}(Y_k, \epsilon) \\ Y_i, & \text{if } Y_i \in \mathcal{N} \wedge \forall_{k=1}^{i-1} \mathsf{FirstSet}(Y_k, \epsilon) \\ t, & \text{if } X = t \\ nil, & \text{otherwise} \end{cases} \tag{9}$$

$$\mathsf{EpInFirst}_0(X) = \begin{cases} true, & \text{if there exists a rule } \langle X \rightarrow \epsilon \rangle \\ false, & \text{otherwise} \end{cases} \tag{10}$$

$$\mathsf{EpInFirst}_i(X) = \begin{cases} true, & \text{if } \exists \langle X \rightarrow Y_1 Y_2 \ldots Y_n \rangle \text{ s.t. } \forall_{k=1}^{n} \mathsf{EpInFirst}_{i-1}(Y_k) \\ true, & \text{if } \mathsf{EpInFirst}_{i-1}(X) \\ false, & \text{otherwise} \end{cases} \tag{11}$$

Fig. 8 First set witness functions

- $\mathsf{FirstSetWitness}_{\langle r,i\rangle}(X,t)$: Given a rule r and a symbol Y_i in the body of the rule, this function attempts to find whether $First(Y_i) \subseteq First(X)$; 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.
- $\mathsf{EpInFirst}_i(X)$: The function $\mathsf{EpInFirst}_0(X)$ 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, $\mathsf{EpInFirst}_i(X)$ would evaluate to true iff we infer $\epsilon \in First(X)$ via a sequence of less than or equal to i production rules, each such sequence terminating with an empty production.

We encode the predicate $\mathsf{FirstSet}(X,t)$ such that it is satisfiable if and only if $t \in First(X)$. To infer whether $\epsilon \in First(X)$, we invoke $\mathsf{EpInFirst}_{|\mathcal{N}|}(X)$ in search of a sequence of production rules that could witness the derivation of ϵ from X. Note that any such sequence that could derive ϵ from X cannot be longer than the number of non-terminals.

$$\mathsf{FirstSet}(X,\epsilon) = \mathsf{EpInFirst}_{|\mathcal{N}|}(X) \tag{12}$$

To infer whether $t \in \mathcal{T}$ is in $First(X)$, we search over all production rules $\langle r : X \rightarrow Y_1 Y_2 \ldots Y_n \rangle \in \mathcal{P}$ where X appears as the head of a rule, and over all (non-ϵ) symbols Y_i in the body of such productions, to uncover a sequence of rules that could witness $t \in First(X)$.

The existential in Eq. 13 attempts to *guesses* a sequence of tuples $\langle r, i \rangle$ (of length at most the number of non-terminals, $k = |\mathcal{N}|$) that could witness the derivation of $t \in First(X)$.

Example Consider the LL(1) grammar $\mathcal{G}_1$ in Fig. 1: we demonstrate 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 $\mathsf{FirstSetWitness}$ and Ep to denote $\mathsf{EpInFirst}$. The constraints are evaluated as follows:

- $F_{\langle 6,1\rangle}(C,l) = l$, $F_{\langle 4,1\rangle}(B,r) = r$ (first case of $F()$ in Eq. 9);
- From production 2, $F_{\langle 2,1\rangle}(A,l) = C$ (second case of $F()$ in Eq. 9);
- From the empty productions for B, D and E (productions 5, 7 and 8 respectively), we get $Ep_0(B) = true$, $Ep_0(D) = true$ and $Ep_0(E) = true$;
- $Ep_1(A) = Ep_0(D) \wedge Ep_0(E) = true$;

- With production rule 1, we get $Ep_2(S) = true$, $F_{\langle 1,1\rangle}(S,c) = A$ and $F_{\langle 1,2\rangle}(S,v) = B$.

$$
\begin{aligned}
& t \in \mathsf{First}(X) \equiv \mathsf{FirstSet}(X,t) = \exists \langle r_0, i_0\rangle, \langle r_1, i_1\rangle, \ldots, y_0, y_1, \ldots \\
& (y_0 = \mathsf{FirstSetWitness}_{\langle r_0,i_0\rangle}(X,t) \wedge X = \mathsf{head}(r_0) \wedge y_0 \in \mathsf{body}(r_0) \wedge y_0 \neq \epsilon) \wedge \\
& (y_1 = \mathsf{FirstSetWitness}_{\langle r_1,i_1\rangle}(y_0,t) \wedge y_0 = \mathsf{head}(r_1) \wedge y_1 \in \mathsf{body}(r_1) \wedge y_1 \neq \epsilon) \wedge \\
& \ldots \\
& (y_k = \mathsf{FirstSetWitness}_{\langle r_k,i_k\rangle}(y_{k-1},t) \wedge y_{k-1} = \mathsf{head}(r_k) \wedge y_k \in \mathsf{body}(r_k) \wedge y_k \neq \epsilon)
\end{aligned} \tag{13}
$$

From Eq. 11, we conclude that $\mathsf{FirstSet}(n,\epsilon) = true \ \ \forall n \in \{A,B,D,E\}$. Also, we can observe that $\mathsf{FirstSet}(S,l)$ uses $r_0 = 0, i_0 = 1, r_1 = 2, i_1 = 1, r_2 = 6, i_2 = 1$ to get $y_0 = F(S,l) = A, y_1 = F(A,l) = C, y_2 = F(C,l) = l$ (in Eq. 13). Subsequently, $y_j = F(c,l) = l$ (from third case of Eq. 9). This shows the derivation of a least fixpoint by our encoding. If we try to have a satisfying assignment for $\mathsf{FirstSet}(A,r)$, we would find that there are none possible, thereby disallowing this possibility in our symbolic model.

5.4 Encoding of the follow set constraints

The witness generation for $t \in Follow(X)$ is also posed in a similar way to that of the encoding for the first sets. For the follow set constraints, $t \in Follow(X)$, the witness generation is essentially a search for a path from a production with $X \in N$ in its body to a production with a body that (explicitly) contains t as a terminal right next to a non-terminal symbol which has an equivalent follow set as X.

Figure 9 defines the *witness* functions to compute the least fixpoint for the follow set constraints. Given a production rule $\langle r : Y_0 \rightarrow Y_1 \ldots Y_n\rangle$ and a symbol Y_i in its body, the witness functions essentially encode the condition of some symbol Y_j (either $j = 0$ or $j > i$) resulting in either $Follow(Y_j) \subseteq Follow(Y_i)$ 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).

Furthermore, the constraint in Fig. 10 provides a satisfying solution for $\mathsf{FollowSet}(X,t)$ if and only if $t \in Follow(X)$. The constraint essentially attempts to construct a chain of production rules that can attest to $t \in Follow(X)$. The constraint essentially 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:

$$
\mathsf{FollowSetWitness}_{\langle \_,\_,\_\rangle}(S,\$) = \$ \tag{14}
$$

$$
\mathsf{FollowSetWitness}_{\langle r,i,j\rangle}(X,t) = \begin{cases} t, & \text{if } j > 0 \wedge X = Y_i \in \mathcal{N} \wedge \\ & \mathsf{FirstSet}(Y_j,t) \wedge \forall_{k=i+1}^{j-1} \mathsf{FirstSet}(Y_k,\epsilon) \\ X, & \text{if } j = 0 \wedge X = Y_i \in \mathcal{N} \wedge \\ & \qquad Y_i \in \mathcal{N} \wedge \forall_{k=i+1}^{n} \mathsf{FirstSet}(Y_k,\epsilon) \\ t, & \text{if } X = t \\ nil, & \text{otherwise} \end{cases} \tag{15}
$$

Fig. 9 Follow set witness functions

$$\begin{aligned}
&\texttt{FollowSet}(X,t) = \exists \langle r_0,i_0,j_0\rangle,\langle r_1,i_1,j_1\rangle,\ldots,\langle r_k,i_k,j_k\rangle, y_0,y_1,\ldots,y_k.\\
&\quad j_0 \in \{0\}\cup[i_0+1,\ldots,n], j_1 \in \{0\}\cup[i_1+1,\ldots,n],\cdots\wedge\\
&\quad (y_0 = \mathsf{FollowSetWitness}_{\langle r_0,i_0,j_0\rangle}(X,t)\wedge X\in \mathsf{body}(r_0)\wedge y_0\in\mathsf{symbol}(r_0))\wedge\\
&\quad (y_1 = \mathsf{FollowSetWitness}_{\langle r_1,i_1,j_1\rangle}(y_0,t)\wedge y_0\in \mathsf{body}(r_1)\wedge y_1\in\mathsf{symbol}(r_1))\wedge\\
&\quad \ldots\\
&\quad (y_k = \mathsf{FollowSetWitness}_{\langle r_k,i_k,j_k\rangle}(y_{k-1},t)\wedge y_{k-1}\in \mathsf{body}(r_k)\wedge y_k\in\mathsf{symbol}(r_k))
\end{aligned} \tag{16}$$

Fig. 10 Follow set constraint

- r: A rule that can potentially be used to infer that $t \in \mathit{Follow}(X)$;
- 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 $t \in \mathit{Follow}(X)$; there exist two possibilities for j:

  1. $j = 0$: $t \in \mathit{Follow}(Y_0)$ (head of rule r) and $\epsilon \in \mathit{First}(Y_k)\forall i < k \leq n$;
  2. $j > i$: $t \in \mathit{First}(Y_j)$ (in the body) and $\epsilon \in \mathit{First}(Y_k)\forall i < k < j$.

Note that *witness* construction is essential as otherwise, we may not converge on the least fixpoint: for the rules $\{\langle S \rightarrow AB\rangle, \langle B \rightarrow bS\rangle\}$, the follow set constraints can be satisfied for $a \in \mathit{Follow}(B) \wedge a \in \mathit{Follow}(S)$; notice that this is not the least fixpoint as there is no production rule that could introduce a in $\mathit{Follow}(S)$.

Example For our grammar in Fig. 1, we demonstrate ours follow 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 FollowSetWitness. The constraints are evaluated as follows:

- $F_{\langle 2,1,2\rangle}(C,p) = p$ (first case of $F()$ in Eq. 15);
- $F_{\langle 6,2,0\rangle}(S,p) = C$ (due to production 6 and second case of $F()$ in Eq. 15);
- From production 1, we get $F_{\langle 1,1,0\rangle}(B,p) = S$ and $F_{\langle 1,2,0\rangle}(A,p) = S$
- From production 3, we get $F_{\langle 3,1,0\rangle}(D,p) = A$ and $F_{\langle 3,2,0\rangle}(E,p) = A$

From Eq. 16, we observe that $\mathsf{FollowSet}(D,p)$ uses $r_0 = 3, i_0 = 1, j_0 = 0, r_1 = 1, i_1 = 2, j_1 = 0$, $r_2 = 2, i_2 = 1, j_2 = 2$ to get $y_0 = F(D,p) = A, y_1 = F(A,p) = C, y_2 = F(C,p) = p$. Subsequently, $y_j = F(p,p) = p$ (from third case of Eq. 15). This shows the derivation of a least fixpoint by our encoding. If we try to have a satisfying assignment for $\mathsf{FollowSet}(A,l)$, we would find that there are none possible, thereby disallowing this possibility in our symbolic model.

5.5 The encoding of the parse table constraints

The constraints for an LL(1) ParseTable are then given as:

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

We encode ParseTable(X, t) as (where FiS stands for FirstSet and FoS stands for FollowSet):

$$\bigwedge_{X \in \mathcal{N}, t \in \mathcal{T}, r_i \in \mathcal{P}} [\text{FiS}(\alpha, t) \vee (\text{FiS}(\alpha, \epsilon) \wedge \text{FoS}(\alpha, t))] \iff (\text{ParseTable}(X, t) = \langle r_i : X \to \alpha \rangle) \tag{17}$$

As ParseTable is defined as a function, if the implication conditions turn *true* for multiple rules for the same $\langle X, t \rangle$ pair, the above constraint will turn unsatisfiable (as a function can take only a single value for the same arguments)—this exactly signifies a parse table conflict (inability to construct a valid LL(1) parse table). For example, if First(α_1, t) causes ParseTable(X,t)=r_1 and First(α_2, t) causes ParseTable(X,t)=r_2, it would indicate a first-first conflict.

To summarize, the constraint system for Cyclops ensures that a valid LL(1) parser can be constructed for the grammar, and a valid parse tree can be constructed for the subject string dictated by the parse table.

5.6 Implementation and experiments

We implemented Cyclops in Python using the Z3 4.4.2 [26] SMT solver. We now demonstrate the three applications that we built on Cyclops .

Our SMT encoding is built on the theory of linear integer arithmetic and uninterpreted functions. We do not use the array theory; arrays are implemented as uninterpreted functions. We use Z3 with E-matching [9, 78, 79] based quantifier instantiof each non-terminals in ation (disabled `mbqi` and `auto_config`) with carefully selected triggers via the `pattern` construct.

To the best of our knowledge, this is the first attempt at building symbolic encoding of parser constraints in SMT. We evaluate Cyclops with an objective to answer the following research questions:

Utility Can parse conditions be used to build a variety of interesting applications?
Effectiveness Can parse conditions be used to build applications that are effective at their tasks (i.e. can solve a large number of instances)?
Efficiency Are the applications built using parse conditions efficient (i.e. can solve non-trivial instances in reasonable time)?The whole appendix is wrong. You have included "appendix.tex". Actually, the main appendix is present in "Appendix.tex". Please use "Appendix.tex".

The first question answers the generality of the notion of parse conditions. We answer this by building three different applications over Cyclops : repair syntax errors in programs (§ 6), synthesize parsers from examples (§ 7), and in teaching parser technology (§ 8). For each of the applications, we answer the questions of effectiveness and efficiency:

- Our case study on repairing syntax errors for programs in the Tiger language [10] spans more than 90 productions, and Cyclops successfully repairs 80% of our benchmarks, clocking an average of 30 s per repair;
- Our parser synthesis application automatically synthesizes non-trivial grammars; most of our benchmarks are synthesized in less than 20 min;
- Our pedagogical application ranks the ground truth fault of 86% of the benchmarks at rank 1, around 13% of the benchmarks at rank 2 and around 1% at rank 3, taking less than a minute in all cases. We have also conducted a pilot study regarding the useful-

ness of our ranking algorithm, and out of 128 responses from the students, 91 responses state that CYCLOPS is useful in finding the bug in the erroneous parse table.

To the best of our knowledge, there does not exist any competing tools for the ones we build—there does not exist any tool that is capable of synthesizing LL(1) parsers from examples, any tool that implements a *zero-shot* algorithm for repairing syntax errors in programs, or any pedagogical tool that attempts to *minimize* the "distance" of the student's solution from a correct solution.

5.7 Discussion

Our encoding for parse conditions employs embedding the parse table into a vector. This allows us to build our encoding in a fragment of first-order logic employing only linear integer arithmetic, uninterpreted functions, and arrays. However, there do exist other possibilities to build encoding for parse conditions. For example, one may employ algebraic data structures supported by SMT solvers [23] to derive an encoding for parse conditions. Such encodings on SMT solvers can borrow ideas from prior work (like Zippy [27]) that model the parsing task on proof-assistants like Coq [103]. However, the performance impact of using richer theories (of algebraic data structures) over simpler theories (LIA+UF+Arrays) needs to be evaluated. The other option could be to derive a symbolic encoding from the run of an LL(1) parser. However, as symbolic runs will expand a path in the parser at a time, the encoding can tend to get large. However, both the approaches[1] look interesting, and we intend to build and evaluate them in the future.

6 CYCLOPS for program repair

Given a string w as a sequence of tokens[2] $[w_0, \ldots, w_n]$ and a parser (a grammar $\mathcal{G}$ and its LL(1) parse table) such that $w \notin \mathcal{L}(\mathcal{G})$, we attempt to find mutations to w such that it is accepted by $\mathcal{G}$. We allow replacement (changing a token to another), deletion (dropping a token) and insertion (addition of a new token) as repairs to the existing token sequence. Our repair algorithm asserts the parse table and the parsing algorithm constraints, and treats (parts of) the input string as *free* variables to synthesize possible mutations. Using an SMT solver on the Parse Condition allows us to efficiently traverse the mutation space of the program in search of a string (token sequence) that is accepted by the grammar (i.e., is syntactically correct).

We encoded mutations in SMT as follows:

- Deletions are implemented by "marking" the (index in the parse array) location deleted so that it does not map to any node in the parse tree;
- Insertions are handled by maintaining a bounded number of "floating" insertion slots (additional variables) that are *logically* accommodated in the parse array at the required array indices.
- Replacements are essentially modeled as deletion (of the incorrect symbol) followed by insertion of the "correct" symbol.

[1] we are thankful to the anonymous reviewers for suggesting these approaches.

[2] *Tokens* in the parser correspond to terminals in the grammar; we use them interchangeably.

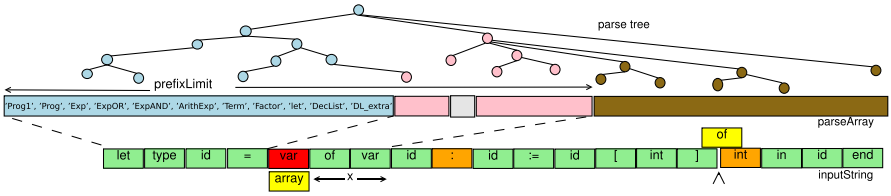


Fig. 11 Repairing syntactic errors on the Tiger grammar

The Tiger programming language [10] grammar contains 91 productions, with the size of their bodies having at most 5 symbols. We use JavaCC [51] for the lexical analysis phase to get a sequence of terminals. We use a slightly modified grammar of this language than what is provided in [89] (as the given grammar was not exactly LL(1)).

6.1 Fault localization

We use the JavaCC parser for fault localization: whenever the JavaCC parser halts with a parse error, we mark the position of the respective token in the input string as *suspicious* and then use CYCLOPS to search for a possible mutation (replacement, deletion or insertion of a new terminal) at the suspicious location that allows for a successful parse. An alternative could have been to use a statistical fault localizer [1, 16, 68, 69, 77], in which case one could also have false positives (i.e. locations marked suspicious when they are not). Our repair algorithm is oblivious to the fault localization engine used.

6.2 Repair algorithm

We draw a core insight from the properties of predictive parsing: the parsing (and the constructed parse tree) is correct till a "buggy" token (identified by the fault localization module) appears as a lookahead symbol. We use this insight to prune away any *prefix* of the parseArray that does not depend on a suspicious symbol as a lookahead from the search space of potential repairs; this prefix is constructed by running the string on a conventional LL(1) parser till a suspicious terminal, say t, appears in the lookahead. We, then, query CYCLOPS to synthesize the following sequence of symbols in the parse array, till the location where t appears.

Let us use Fig. 11 to illustrate our algorithm. Our subject string, `inputString`, has two errors: 'var' must be replaced with 'array' and 'of' must be inserted in the string. We show the partially filled parseArray in

blue till var appears as a lookahead symbol: these elements are asserted non-mutable in parseArray. CYCLOPS now attempts to synthesize a mutation to the existing parse tree (embedded in `parseArray`)—searching for possible mutations to compute a *valid* subtree (of the parse tree) that would generate the next x terminals past this suspicious location (shown in pink). In our example, it finds a consistent solution by mutating one location for var (marked in gray), along with a few mutations in the parse tree (not shown); as a side-effect, it also computes a relevant value of prefixLimit to hold this valid prefix of the parse.

CYCLOPS attempts to synthesize the parseArray upto x symbols *past* a suspicious location. These x terminals in the string are also asserted to provide enough context for a successful parse. For our experiments, we use a window with 5 terminals *before* the faulty location and 3 terminals *after* the faulty location to setup the context for repair. Of course, the above scheme is based on the hypothesis that a string can be repaired simply by ensuring that the prefixes of the string are valid (i.e. for a prefix α, there exists a suffix β such that $\alpha\beta \in L(\mathcal{G})$).

Continuing with the example, once CYCLOPS returns the repair, the synthesized terminal is asserted non-mutable and a new prefix of parseArray is generated till the next suspicious location (i.e. ':') appears as the lookahead. This is a false positive from the localization module; so CYCLOPS is able to verify that the prefix is valid *without* any mutations, thereby correctly identifying it as a false positive. Similarly the last location is handled.

Please note that the approximation of driving the repair with only a "small" window around the buggy location (prefix string, buggy location, followed by a suffix string), instead of the whole string, can temporarily lead to incorrect repairs. In such cases, the repair procedure will fail eventually, and the procedure will backtrack. Specifically, it is possible that a prefix α is valid i.e. that is there exists a following suffix β such that $\alpha\beta \in L(\mathcal{G})$), but the respective suffix γ in the ground-truth, $\alpha\gamma \notin L(\mathcal{G})$: in such cases, CYCLOPS backtracks a bounded number of times to synthesize a different repair. We reassert: when our procedure terminates, it would have synthesized a provably correct repair.

We enable the above approximation by setting the value for prefixLimit depending on the selected value of x instead of the end of the string in Fig. 6.

An interesting aspect of this algorithm is that it is *almost* independent of the length of the input string. The cost of SMT solving is dependent mostly on the parameter x which can be tuned for a given grammar.

6.3 Evaluation

We evaluated our scheme on a set of benchmarks (Tiger programs) from [45]. We ran our experiments on a Linux container with a 2.4 GHz Intel processor and 198 GB main memory. An automatic mutation tool was used to randomly insert faults: one error, either replacement of a token by another, deletion of a token or insertion of an arbitrary token. We created 5 buggy versions corresponding to each fault-type on a set of 45 Tiger programs—creating a set of 675 buggy programs. Table 3 shows the summary of our results for each fault class: TO denotes the percentage of benchmarks that timeout within a time budget of 2.5 min; all the remaining statistics provide a percentage over benchmarks that do not timeout. Time is the average time per repair, Localization Recall is the percentage of benchmarks where our JavaCC based error localizer is able to predict the correct error location. Repair Rate shows the percentage of benchmarks where we were able to generate a correct repair patch. Overall we were able to successfully repair over 80% of our benchmark programs, clocking an average of **30 s** per repair.

To test the response of our tool on programs that use almost all constructs of the Tiger grammar, we created a larger and more complex Tiger program (containing almost all possible constructs like arrays, functions, for loops, while loops, conditionals, various types of declarations and records) by borrowing statements corresponding to different constructs from our set of 49 programs. We then randomly injected various

Table 3 Repairing Tiger Programs

Fault Class	TO (%)	Time (s)	Localization Recall (%)	Repair Rate (%)
Replace	11.11	22.07	91.67	82.81
Insert	0	40.72	95.09	81.25
Delete	0.44	28.74	80.72	78.48
Total	3.76	30.93	89.05	80.75

bugs (insertion, deletion and replacements) to create 9 buggy versions of this program. CYCLOPS could repair 7 of these 9 programs within the timeout of 2.5 min, with an average repair time of 48.5 s. This also supports our hypothesis that the algorithm is *almost* independent of the length of the input string, since the repair times were almost similar.

7 CYCLOPS for parser synthesis

Given a set of positive examples ($\omega_i \in \xi$), CYCLOPS asserts the symbolic encoding for the first and follow constraints ($First(\mathcal{G}), Follow(\mathcal{G})$), the parse table constraints ($ParseTable(\mathcal{G}, \lambda)$) and a (renamed) instance of the embedding/ parsing algorithm ($ParseCondition(\mathcal{G}, \omega)$) for each positive string $\omega_i \in \xi$. In this case, we assert ξ while the grammar $\mathcal{G}$ appears free (hence, synthesized). We require separate instances of $ParseCondition(\mathcal{G}, \omega)$ 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 all the parsing constraints for all the positive strings at once, with a single call to CHKSAT();
- Add the parsing algorithm constraints for each of the positive strings incrementally, with multiple calls to CHKSAT(); we make the initial calls with smaller queries, hoping to use the learning ability of the solver for larger queries issued later.

We evaluate our tool on a set of benchmark grammars collected from prior literature [4, 14] and online course notes [7, 21, 28, 85, 118]. We ran our experiments on a Linux container with a 2.4 GHz Intel processor and 198 GB main memory. Table 4 summarizes our experimental results. For each benchmark, we provide a *template* for the synthesized grammar, specified by the tuple (P,B,T,N) in terms of the number of production rules (P), the 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; the tuple (E,L) records the number (E) and the average length (L) of such examples.

We found that the solving time in the incremental solving mode is 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 reported the minimum time taken (Min) and the average time taken (Avg) for these five runs. Though

Table 4 Parser synthesis (runtimes in seconds)

(P,B,T,N)	(E,L)	Incremental		All-at-once	
		Min	Avg	Min	Avg
(8,3,6,4)	(6,2.5)	416	2254	**261**	**433**
(13,2,7,7)	(20,3)	2924	4991	2800	5722
(2,4,2,1)	(18,6.8)	77	92	83	98
(5,4,3,3)	(20,7.8)	**146**	**974**	1194	2162
(4,2,2,3)	(2,4)	3	3	4	6
(3,3,2,2)	(5,5)	12	16	12	12
(7,3,5,4)	(20,5.3)	**473**	**3942**	1758	2532
(6,2,3,4)	(4,2)	6	9	6	10
(5,3,3,3)	(20,6.4)	**95**	**128**	116	268
(4,3,2,2)	(20,7.4)	81	87	83	98
(3,2,2,2)	(20,10.5)	67	78	66	73
(4,2,3,2)	(20,4.5)	33	38	36	37
(5,2,2,3)	(19,3.5)	30	34	12	30
(4,5,8,1)	(20,6.6)	141	164	129	142

Fig. 12 $\mathcal{G}_2$

$$
\begin{aligned}
S &\rightarrow E \ ^{[1]} \\
&\mid b\ G \ ^{[2]} \\
E &\rightarrow a \ ^{[3]} \\
G &\rightarrow F\ E \ ^{[4]} \\
F &\rightarrow \epsilon \ ^{[5]}
\end{aligned}
$$

in many of them, the differences in the results are small, we found cases (in bold letters) where one of the settings outperforms the other setting significantly.

In summary, CYCLOPS runs an inductive search to generalize from a small set of examples and hence, can synthesize grammars for *any* language for which an LL(1) grammar exists (including infinite languages). The existence of a context-free grammar for a given language (without fixing the structure of the grammar) is undecidable; we circumvent this difficulty with the user providing the *maximal* template within which the grammar would fit: for example, a bound on the size of the rule bodies, number of productions, etc. Any "smaller" grammar can be synthesized by CYCLOPS , with ϵ padding for the unused symbols (for example, $A \rightarrow B\ C$ would be synthesized as $A \rightarrow B\ C\ \epsilon\ \epsilon$ if the bound on the body is four).

8 CYCLOPS for teaching syntax analysis

Syntax Analysis (a.k.a *parsing*) is one of the challenging topics in a compiler design course. There are multiple queries posted by students where they face difficulties understanding the essence of these efficiently parseable grammars. There are interesting discussion threads on this topic on an online forum on whether certain grammars are LL(1) [67] and how to transform a given grammar to LL(1) [67]. As CYCLOPS packages strong

Table 5 Parse table for grammar shown in 12

Non Term	a	b	$
S		$S \rightarrow b\ G$	
E			
G			
F			

reasoning capabilities on parsing, it can be used to help design educational tools for students.

In this section, we show how we retargeted CYCLOPS to help students debug incorrectly constructed parse tables. Locating the root cause for incorrect parse tables is challenging—a single mistake in adhering to the condition for first, follow or parse table construction, even on a single non-terminal/terminal/production, can lead to multiple incorrectly filled cells in the parse table. For example, for the parse table provided in Table 5 for the grammar shown in Fig. 12, the parse table entries painted in red are incorrect. In this case, there is only a single error—the third condition of the first set (see § 5.3) for non-terminal `E` and terminal `a` was applied incorrectly, i.e. it was incorrectly inferred that $a \notin \mathsf{FirstSet}(E)$. This single error leads to wrong parse table entries for multiple cells—$\langle S, a\rangle$, $\langle E, a\rangle$, $\langle G, a\rangle$ and $\langle F, a\rangle$ (marked in red in parse table in Table 5). We use the notation $\langle N, T\rangle$ to represent a cell in a parse table, where N is a non-terminal and T is a terminal representing row N in the parse table, and column T in the parse table, respectively. Table 6 shows the First and Follow sets for the grammar shown in Fig. 12.

Let us now formally state the problem that CYCLOPS is required to solve:

Problem statement Given an incorrect parse table, identify the potential *root cause* for the fault. Instead of identifying a single error, we formulate it as a ranking problem and attempt to rank all the potential errors that could have led to the provided (incorrect) parse table.

8.1 Algorithm

The algorithm for this module (Algorithm 1) consumes an incorrect parse table, PT, and a grammar $\mathcal{G}$. The algorithm starts off by computing the first set and follow set on the grammar $\mathcal{G}$ (Lines 1 and 2). Then it constructs the correct parse table on $\mathcal{G}$ and matches it against PT to collect all the erroneous cells in the variable incorrectCells (Line 5). Finally, it constructs the parse condition, Ω, for the grammar $\mathcal{G}$. Next, the algorithm iterates over all the erroneous cells collected in incorrectCells, and reasons on three specializations of the parse condition:

1. The parse condition where the symbolic variable in Ω corresponding to the parse table entry *cell* is bound to the respective value v in the erroneous the parse table.
2. The constraint (1) above and, additionally, we bind the respective uninterpreted functions for the first set in Ω to the *correct* first sets computed on the parse table.
3. The constraint (2) above and, additionally, we bind the respective uninterpreted functions for the follow set in Ω to the *correct* follow sets computed on the parse table.

Table 6 First and follow sets for grammar shown in Fig. 12

Non Term	First set	Follow set
S	{a,b}	{$}
E	{a}	{$}
F	{ϵ}	{a}
G	{a}	{$}

We collect the unsatisfiable cores in all the above cases in a set, cores: while the unsatisfiable cores for case (1) attempt to reveal faults due to all possible reasons, cases (2) and (3) are more directed, that directs the SMT solver to reveal faults *if the first and/or follow sets are assumed to have been computed correctly by the student*.

The unsatisfiable cores corresponding to all the faulty entries are accumulated in allCores. We use the symbol $\Cup$ to show the union operation on multisets.

Finally, we accumulate all the terms and rank the unsatisfiable cores by their frequency in the multiset allCores to obtain a list of reasons for the faults. If an unsatisfiable core of certain constraints occurs quite frequently, it appears high on the list of suspicious reasons for the faults.

Algorithm 1 Algorithm for fault ranking for a given incorrect parse table

Function *RankedFaults()*

Input: $\mathsf{PT}, \mathcal{G}$

1 $\mathsf{FiSet} \leftarrow \mathsf{GetFirstSet}(\mathcal{G})$
2 $\mathsf{FoSet} \leftarrow \mathsf{GetFollowSet}(\mathcal{G})$
3 $\mathsf{allCores} \leftarrow \phi$
4 $\Omega \leftarrow \mathsf{GetParseCondition}(\mathcal{G})$
5 $\mathsf{incorrectCells} \leftarrow \mathsf{GetFaults}(\mathsf{PT}, \Omega)$;
6 **foreach** cell $\in$ incorrectCells **do**
7 $\quad \mathsf{v} \leftarrow \mathsf{PT}[\mathsf{cell}]$
8 $\quad \mathsf{core}_1 \leftarrow \mathsf{EnumMinCores}(\Omega \wedge \Omega[\mathsf{cell}] = \mathsf{PT}[\mathsf{cell}])$
9 $\quad \mathsf{core}_2 \leftarrow \mathsf{EnumMinCores}(\Omega \wedge \Omega[\mathsf{cell}] = \mathsf{PT}[\mathsf{cell}] \wedge \Omega[\mathsf{FiSet}] = \mathsf{FiSet})$
10 $\quad \mathsf{core}_3 \leftarrow \mathsf{EnumMinCores}(\Omega \wedge \Omega[\mathsf{cell}] = \mathsf{PT}[\mathsf{cell}] \wedge \Omega[\mathsf{FiSet}] = \mathsf{FiSet} \wedge \Omega[\mathsf{FoSet}] = \mathsf{FoSet})$
11 $\quad \mathsf{cores} \leftarrow \mathsf{core}_1 \cup \mathsf{core}_2 \cup \mathsf{core}_3$
12 $\quad \mathsf{allCores} \leftarrow \mathsf{allCores} \Cup \mathsf{cores}$
13 **end**
14 $\mathsf{faultList} \leftarrow \mathsf{AccumulateTerms}(\mathsf{allCores})$
15 **return** $\mathsf{sort}(\mathsf{faultList})$

Example Let us run Algorithm 1 to repair the faulty parse table in Table 5. At Line 5, GetFaults finds that the following parse table entries are faulty: $\langle S, a\rangle$, $\langle E, a\rangle$, $\langle G, a\rangle$ and $\langle F, a\rangle$.

Now, let us consider the parse table entry $\langle E, a\rangle$ is selected for the first iteration of the loop: taking a union over the unsatisfiable cores in Lines 8–10, cores collects the following reasons for the fault:

- it is inferred that $a \notin \mathsf{FirstSet}(E)$ due to an incorrect application of the third condition of first set construction;
- it is inferred that $a \notin \mathsf{FirstSet}(a)$ due to an incorrect application of the first condition of first set construction;
- it is inferred that the parse table entry for cell $\langle E, a \rangle$ is wrong due to the incorrect application of the first condition of parse table construction.

Now, allCores accumulates the reasons across all the mismatched entries into a multiset. Ranking them, the algorithm emits the following as the final reasons for the incorrect parse table:

1. *Rank 1* it is inferred that $a \notin \mathsf{FirstSet}(E)$ due to an incorrect application of the third condition of first set construction;
2. *Rank 2* it is inferred that $a \notin \mathsf{FirstSet}(a)$ due to an incorrect application of the first condition of first set construction;
3. *Rank 3* multiple reasons tie for this rank:
   - It is inferred that $a \notin \mathsf{FollowSet}(a)$ due to an incorrect application of the second condition of follow set construction;
   - It is inferred that the parse table entry for cell $\langle F, a \rangle$ is wrong due to the incorrect application of the first condition of parse table construction.
   - It is inferred that the parse table entry for cell $\langle G, a \rangle$ is wrong due to the incorrect application of the first condition of parse table construction.
   - It is inferred that the parse table entry for cell $\langle S, a \rangle$ is wrong due to the incorrect application of the first condition of parse table construction.

In first rank Cyclops suggested is the genuine reason for the faulty parse table.

Now let us consider the suggestion in rank-2 we can also verify that fault in first condition of first set construct for terminal a can also lead to same wrong parse table as shown in Table 5.

8.2 Experiments

We ran our experiments on a Linux machine with a 3.20GHz Intel processor and 16 GB main memory. To evaluate our framework, we created a set of 112 benchmarks by randomly introducing faults in one of the conditions while constructing for the first set, follow set or parse table for one of the non-terminal and/or lookahead.

8.2.1 Ranking effectiveness

Cyclops was quite effective at isolating the fault, ranking all faults within the top-3 ranks. Figure 15 shows the distribution: Cyclops rank the ground-truth at rank-1 for around 86%, rank-2 for around 13% and rank-3 for nearly about 1% of benchmarks.

8.2.2 Runtime performance

We found Cyclops to be quite fast. Figure 13 shows a cactus plot where a point (x, y) shows that x benchmarks were solved within y seconds. Around 80% (90) benchmarks were solved within 27 seconds by Cyclops.

8.3 Pilot study in a compilers course

Let us now discuss our endeavour to use the above module within an undergraduate course in compiler design (CS335A: Compiler Design). We posed the following problem to the students.

> You are given an LL(1) parse table for a given grammar. However, on verifying with some friends, you find that the parse table is wrong! Now, you know that this parse table was created with care, so there could have been created *exactly one mistake* while following the steps to LL(1) parser creation. To be more precise, there is a mistake in exactly one condition of the first set, follow set, and parse table construction—at that too on a single non-terminal/terminal/production. You are required to locate the bug.

We created 51 faulty parse tables and asked the students to isolate the *reason* for the fault. We provided them Cyclops to assist them in their endeavour.

Out of 128 responses from the students, 91 responses were positive, stating that Cyclops was quite useful in their bug hunt. The negative feedback of the students was mostly that Cyclops also suggests some faults that are not much related to the actual faults, alongside the suggestions related to the actual fault. For example, in the motivation example in § 15, some faults suggested by Cyclops in rank-3 are not much related to the actual fault. In the future, we aim to further refine the fault suggestions of Cyclops .

Figure 14 shows a word cloud of all the responses of the students. The figure provides a visual representation of the conducted pilot study; of course, it is provided as an informal summary of the student responses and does not qualify as empirical evidence.

Based on this novel pedagogical process design and feedback, the efficacy of Cyclops is evident at complementing tutors at imparting students a better understanding and appreciation for parser design. We aim to follow up this pilot study with more learning exercises for students involving Cyclops .

9 Related work

Program synthesis has seen a lot of success recently, thanks to significant improvements in modern SAT/SMT solvers. Some interesting proposals emerged with synthesizing loop-free bitvector programs [42, 55]. Recently, synthesizing heap manipulations [36, 86, 87, 90, 116] and data manipulations [112–114] have attracted a lot of attention. There have also been attempts at synthesizing randomized programs, like that for differential privacy [91, 115]. Many program synthesis techniques also influenced other domains, like synthesizing Skolem functions for QBF formulae [6, 39–41], and for protecting IP of hardware designs [101]. Two popular approaches for specifying synthesis problems have been through program sketching [97] and syntax-guided synthesis [8].

A related problem is that of program repair [34, 35, 64, 75, 95, 104]—instead of searching for a program from scratch, we attempt to synthesize a *patch* for a faulty program. Of course, it needs to satisfy additional constraints on the patch being small, i.e., the repaired

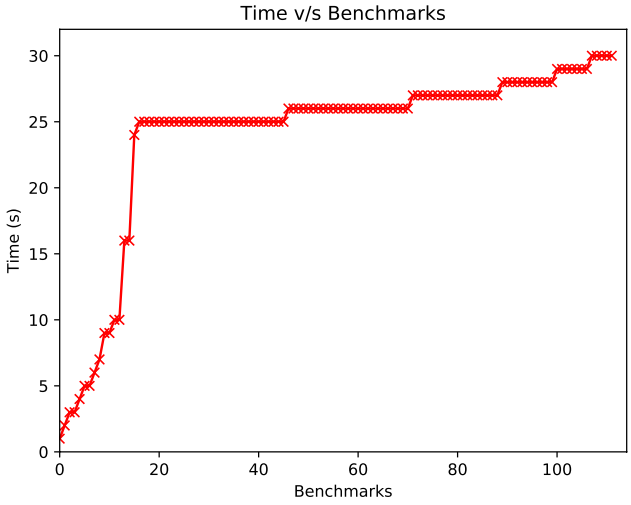


Fig. 13 Plot to show the time taken by CYCLOPS to solve 112 benchmarks

program is "close" to the original program (on some metric, either syntactic or semantic). There have been some recent approaches that attempt to fuse debugging and repair in a cohesive tool, for example, for repairing heap manipulations [109, 110] or concurrent programs in relaxed memory models [108].

To the best of our knowledge, there is no prior work that attempts an SMT-based symbolic encoding of parser constraints. We discuss and compare related ideas from other domains (for example, programs and grammars) in this section. Note that there are parser generators (like Yacc [56]) to generate a parser from a grammar specification; this is not the problem that we solve. Instead, CYCLOPS generates both the grammar as well as the parser—together—just from a set of strings in the language, hence eliminating the need for a grammar to be provided. Moreover, the grammar is guaranteed to be an LL(1) grammar (note that the existence of an LL(1) grammar for a given language is undecidable).

Grammar induction, that is, synthesis of grammar from a set of examples has been proposed in the literature. Synapse [80–83] generates grammars in the Chomsky normal form (CNF) using the Cocke-Younger-Kasami (CYK) algorithm [47]. The CYK algorithm is essentially used as an oracle to check whether a grammar is consistent with all the examples; if not, the algorithm backtracks and attempts with another possible rule in CNF. Gramin [92] also adopts a similar methodology, that of trying out all possible rules using the CYK algorithm. Črepinšek et al. [107] 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 [52, 76, 105, 106] have used evolutionary approaches to synthesize grammars. Instead of grammar synthesis, some proposals [62, 63, 93] attempt to "recover" (or extract) grammars from existing tools (like parsers and compilers). Our work on parser synthesis focuses on easily parseable context-free languages that are more constrained than context-free languages. 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.

For synthesizing parsers from a set of examples: Jain et al. [50] use a user-defined knowledge-base of grammar rules to drive a backtracking-based search through a

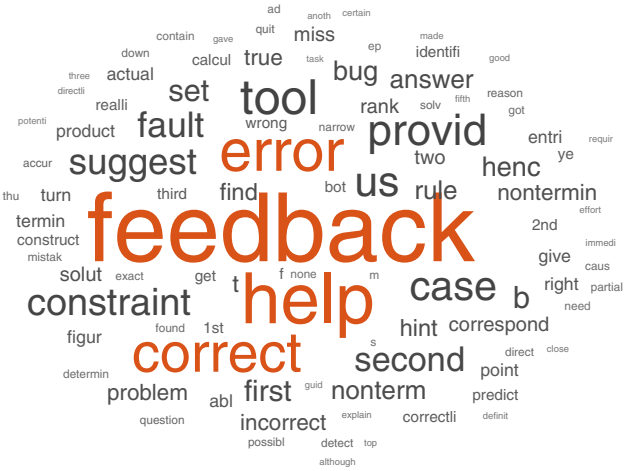


Fig. 14 Word cloud for the feedback provided by students

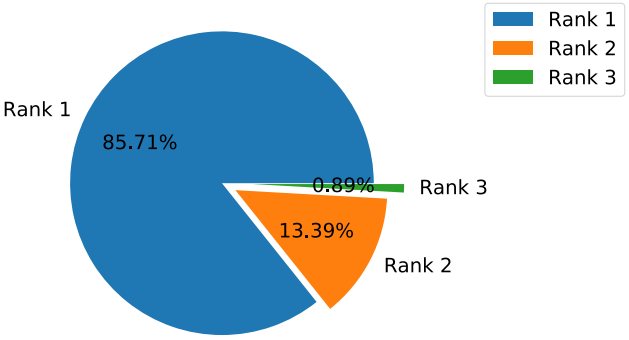


Fig. 15 Pie chart to show breakdown of benchmarks based on the position of correct fault detection by CYCLOPS

bottom-up parser to discover a possible parse; in this scheme, the 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. [76] 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 [66] uses the A* search algorithm along with visual environment for user feedback to generate parsers; we feel that combining our symbolic technique with AI techniques (like A*) is a promising direction for parser synthesis. Parsimony [65] improves upon Parsify with the ability to synthesize lexers and uses a *CYK automata* for a more robust search for generalizations. There have also been proposals that directly employ a parser to search over repair actions [22, 29, 30]. It is interesting to see how these ideas can be borrowed into the symbolic encoding. For example, our idea of verification on a k-length lookahead is similar to the idea of *validation* or *parser check*: the parser is restarted with a repair action applied; the repair is accepted if the parser runs successfully for a sizable length of the subsequent input. In our case, this check amounts to an SMT query rather than a parser invocation.

In the direction of the use of symbolic methods for analyzing context-free languages, CFGAnalyzer [12] 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). For this purpose, they translate the provided grammar to a (modified) Chomsky Normal Form (CNF) and then encode the CYK algorithm. Our work attempts to answer questions on more constrained context-free grammars (i.e., LL(1)) and, thus, requires encoding of the (more involved) LL(1) parsing algorithm. Also, we attempt to reason on the *parser* based on a given grammar; hence, Cyclops reasons on the *grammar* and not just on the *language*—this requirement forbids translation to any "normal forms" for reasoning. There are other works [58, 60, 88] that again use the CYK algorithm for building constraint solvers providing context-free grammar constraints. Madhavan et al. [71] propose a solution to the problem of grammar equivalence by giving an effective testing scheme that efficiently enumerates strings from a given language and a decision procedure for proving equivalence of two grammars which is complete for LL grammars. Zippy [27] provides an efficient and functional LL(1) parsing algorithm and formally verifies it using the Coq theorem prover [103].

Repair tools for semantic errors using symbolic techniques (like [70]) implicitly assume that the program has no syntax errors and use (symbolic) executions of the program to fix semantic errors (given a formal specification of correctness). Of course, we can use our tool in the following iterative mode to fix both syntactic and semantic errors: given a formal specification of correctness, one can feed the (potentially) repaired program from Cyclops to a program verifier to check if the fix satisfies the specification; if not, Cyclops is requested for different fix (from the space of all possible fixes). Again note that verifiers do not accept programs with syntax errors, so Cyclops , coupled with a verifier, can fix both syntax and semantic errors.

As writing parsers is a complex activity, Isradisaikul et al. [49] attempt to assist in debugging conflicts by counterexamples from a parser run. As a *parse condition* encodes the parser runs on all possible strings, they can also assist in such debugging activities by extracting the smallest counterexample and localizing a conflict to a smaller set of the grammar rules that are inconsistent (by extracting entities like unsat cores). Kalita et al. [59] provides an automated algorithm for automatically synthesizing semantic actions for parsers from a set of examples.

There have been work [11, 32, 48, 72, 94, 100, 111] in computer science pedagogy. Specifically, there has been significant work [2, 3, 18, 25, 43, 44, 108, 117] to provide helpful feedback to students on their mistakes in their programming assignments. There have been other works [37, 38, 53, 54] that provide an interactive user interface for the visualization of parsing techniques. Our work is another step in this direction.

There is ample opportunity of integrating deeper reasoning via parse conditions on many of the above tools. Further, while this work attempts to reason on the *syntax* of languages for parser design, we have other work (like Pāṇini [59]) that reasons on the *semantics* of languages—hence, there exists an exciting possibility of integrating these algorithms for end-to-end parser construction.

10 Conclusion

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 whether a string is parseable by a given parser. Like verification conditions have found extensive applications in verification [15, 17, 20, 61], debugging [13, 57, 70], repair [74] and testing [24, 33, 84] of programs, we believe that Parse Conditions can borrow from these ideas (for programs) to solve many more interesting problems in parsing.

We provide a few applications of *parse conditions* to illustrate its utility for solving interesting problems. However, similar to verification conditions, which though being conceived by Hoare's axiomatic formulation in 1969 [46], is still testing researchers on applications to large problems even after being in existence for about 50 years (while fueling innovations like abstraction-refinement [19] and proof-guided abstractions [73]), *parsing conditions* would also need further innovations to be applied to still larger problems. We believe that parse conditions will find wider adoption by the community en route to building more interesting applications.

Appendix: A Procedural description of our encoding

Notation

Instead of providing the constraints declaratively, we instead use a non-deterministic procedural representation of our constraint system. Note that Cyclops 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.

The encoding of the parsing algorithm

Algorithm 2 The parsing algorithm encoding

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

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() 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 2; it then makes non-deterministic choices for the following entities:

- A rule, $r : \langle X \rightarrow Y_1 Y_2 \ldots Y_n \rangle$, that must be used to expand `symbol`;
- A set of `parseArray` indices, $\{p_1, \ldots, p_n, p_{n+1}\}$, where the expansion of X using the rule r (i.e. $\{Y_1, Y_2, \ldots, Y_n\}$) 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, $\{a_1, \ldots, a_n, a_{n+1}\}$, 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 3 must meet the constraints (Lines 4–5) 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`, $[p_1, \ldots, p_n]$, 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 6–9). 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, $[a_1, \ldots, a_{n+1}]$, are chosen in a manner (Lines 10–12) 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`).

The algorithm then (Lines 13–16) 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 ϵ

If the current symbol is ϵ, it simply returns with no updates to the parseArray, and relaying the same lookahead index (Line 21).

Symbol is a terminal

In this case (Lines 23–25), 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).

The encoding of the first sets

Algorithm 3 The first set constraints

Function *FirstSet*

Input: X, t

1 **if** $t = \epsilon$ **then**
2 $\quad x \leftarrow \mathsf{EpInFirst}_{|\mathcal{N}|}(X)$
3 $\quad assume(x)$
4 **else**
5 $\quad y \leftarrow X$
6 $\quad$ **for** $k \in [1 \cdots \mid \mathcal{N} \mid]$ **do**
7 $\quad\quad \langle r : Y_0 \rightarrow Y_1 \ldots Y_n \rangle \leftarrow choose(\mathcal{P})$
8 $\quad\quad assume(Y_0 = y)$
9 $\quad\quad i \leftarrow choose([1 \ldots n])$
10 $\quad\quad assume(Y_i \neq \epsilon)$
11 $\quad\quad y \leftarrow \mathsf{FirstSetWitness}_{\langle r,i \rangle}(y, t)$
12 $\quad$ **end**
13 $\quad assume(y = t)$
14 **end**

For a query $\mathsf{FirstSet}(X, t)$, Algorithm 3 has a terminating execution if and only if $t \in \mathit{First}(X)$; correspondingly, our SMT encoding has a satisfying assignment if and only if $t \in \mathsf{First}(X)$. We use the functions EpInFirst and FirstSetWitness defined in Section 5.3. To infer whether $\epsilon \in \mathit{First}(X)$, we invoke $\mathsf{EpInFirst}_{|\mathcal{N}|}(X)$ 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 whether $t \in \mathcal{T}$ is in $\mathsf{First}(X)$, we search over all production rules $r \in \mathcal{P}$ where X appears as the head of a rule, and over all (non-ϵ) symbols Y_i in the body of such productions, to uncover a sequence of rules that could witness $t \in \mathsf{First}(X)$. The loop at Line 6 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 $t \in \mathsf{First}(X)$. As this algorithm is encoded as an SMT formula, we use the SMT solver to infer the correct *guesses* for our non-deterministic choices.

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.

The encoding of the follow sets

Algorithm 4 The follow set constraints

Function *FollowSet*

Input: X, t

$y \leftarrow X$

for $k \in [1 \cdots \mid \mathcal{N} \mid]$ **do**

  $\langle r : Y_0 \rightarrow Y_1 \ldots Y_n \rangle \leftarrow choose(\mathcal{P})$

  $i \leftarrow choose([1, \ldots, n])$

  $j \leftarrow choose(\{0\} \cup [i+1, \ldots, n])$

  $assume(Y_i = y)$

  $y \leftarrow \mathsf{FollowSetWitness}_{\langle r,i,j \rangle}(y, t)$

end

$assume(y = t)$

Algorithm 4 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. We use the function FollowSetWitness defined in Sect. 5.4 for this. 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 $\epsilon \in First(Y_k) \forall i < k \leq n$;
  2. $j > i$: $t \in First(Y_j)$ (in the body) and $\epsilon \in First(Y_k) \forall i < k < j$.

Data Availability The datasets generated during and/or analysed during the current study are available from the corresponding author on reasonable request.

References

1. Abreu R, Zoeteweij P, van Gemund AJ (2007) On the accuracy of spectrum-based fault localization. In: Testing: academic and industrial conference practice and research techniques - MUTATION (TAICPART-MUTATION 2007), pp 89–98
2. Ahmed UZ, Kumar P, Karkare A, et al (2018) Compilation error repair: for the student programs, from the student programs. In: 2018 IEEE/ACM 40th international conference on software engineering: software engineering education and training (ICSE-SEET), pp 78–87

3. Ahmed UZ, Sindhgatta R, Srivastava N, et al (2019) Targeted example generation for compilation errors. In: 2019 34th IEEE/ACM international conference on automated software engineering (ASE), pp 327–338
4. Aho AV, Ullman JD (1972) The theory of parsing, translation, and compiling. Prentice-Hall Inc, Upper Saddle River, NJ, USA
5. Aho AV, Sethi R, Ullman JD (1986) Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co.Inc, Boston, MA, USA
6. Akshay S, Arora J, Chakraborty S, et al (2019) Knowledge compilation for Boolean functional synthesis. In: 2019 formal methods in computer aided design (FMCAD), pp 161–169
7. Allan SJ (2017) LL Parsing. http://digital.cs.usu.edu/~allan/Compilers/Notes/LLParsing.pdf, online accessed 11 November 2017
8. Alur R, Bodik R, Juniwal G, et al (2013) Syntax-guided synthesis. In: 2013 formal methods in computer-aided design, pp. 1–8 doi: https://doi.org/10.1109/FMCAD.2013.6679385
9. Amin N, Leino KRM, Rompf T (2014) Computing with an SMT solver. In: Seidl M, Tillmann N (eds) Tests and proofs. Springer International Publishing, Cham, pp 20–35
10. Appel AW (2004) Modern compiler implementation in C. Cambridge University Press, Cambridge
11. Arawjo I, Wang CY, Myers AC, et al (2017) Teaching programming with gamified semantics. In: CHI '17. ACM, New York, NY, USA
12. Axelsson R, Heljanko K, Lange M (2008) Analyzing context-free grammars using an incremental sat solver. In: Proceedings of the 35th international colloquium on automata, languages and programming, part II. Springer-Verlag, Berlin, Heidelberg, ICALP '08, pp 410–422
13. Bavishi R, Pandey A, Roy S (2016) To be precise: regression aware debugging. In: Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications. ACM, New York, NY, USA, OOPSLA 2016, pp. 897–915
14. Beatty JC (1982) On the relationship between LL(1) and LR(1) grammars. J ACM 29(4):1007–1022. https://doi.org/10.1145/322344.322350
15. Chatterjee P, Roy S, Diep BP, et al (2020) Distributed bounded model checking. In: 2020 formal methods in computer aided design, FMCAD 2020, Haifa, Israel, September 21-24, 2020. IEEE, pp. 47–56
16. Chatterjee P, Chatterjee A, Campos J, et al (2021) Diagnosing software faults using multiverse analysis. In: Proceedings of the twenty-ninth international joint conference on artificial intelligence, IJCAI'20
17. Chatterjee P, Roy S, Diep BP et al (2022) Distributed bounded model checking. Formal Methods Syst Des. https://doi.org/10.1007/s10703-021-00385-1
18. Chhatbar D, Ahmed UZ, Kar P (2020) Macer: a modular framework for accelerated compilation error repair. In: Bittencourt II, Cukurova M, Muldner K et al (eds) Artificial intelligence in education. Springer International Publishing, Cham, pp 106–117
19. Clarke E, Grumberg O, Jha S et al (2000) Counterexample-guided abstraction refinement. In: Emerson EA, Sistla AP (eds) Computer aided verification. Springer, Berlin Heidelberg, Berlin, Heidelberg, pp 154–169
20. Clarke EM, Kroening D, Lerda F (2004) A Tool for Checking ANSI-C Programs. In: Jensen K, Podelski A (eds) TACAS, vol 2988. Lecture Notes in Computer Science. Springer, Cham, pp 168–176
21. Cockett R (2002) Compiler construction I: LL(1) grammars and predictive top-down parsing. http://pages.cpsc.ucalgary.ca/~robin/class/411/LL1.2.html, online; accessed 11 November 2017
22. Corchuelo R, Pérez JA, Ruiz A et al (2002) Repairing syntax errors in LR parsers. ACM Trans Program Lang Syst 24(6):698–710. https://doi.org/10.1145/586088.586092
23. Corporation M (2022) Datatypes (online z3 guide). https://microsoft.github.io/z3guide/docs/theories/Datatypes/, online; accessed 11 November 2022
24. Daca P, Gupta A, Henzinger TA (2016) Abstraction-driven concolic testing. In: Proceedings of the 17th international conference on verification, model checking, and abstract interpretation - Vol. 9583. Springer-Verlag New York, Inc., New York, NY, USA, VMCAI 2016, pp. 328–347
25. Das R, Ahmed UZ, Karkare A, et al (2016) Prutor: a system for tutoring cs1 and collecting student programs for analysis. https://doi.org/10.48550/ARXIV.1608.03828
26. De Moura L, Bjørner N (2008) 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. Springer-Verlag, Berlin, Heidelberg, TACAS'08/ETAPS'08, pp 337–340
27. Edelmann R, Hamza J, Kunčak V (2020) Zippy ll(1) parsing with derivatives. In: Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation. Association for computing machinery, New York, NY, USA, PLDI 2020, pp. 1036–1051

28. Fischer CN (2013) Introduction to programming languages and compilers. http://pages.cs.wisc.edu/~fischer/cs536.f13/lectures/f12/Lecture22.4up.pdf, online; accessed 11 November 2017
29. Fischer CN, Mauney J (1992) A simple, fast, and effective LL(1) error repair algorithm. Acta Inf 29(2):109–120. https://doi.org/10.1007/BF01178502
30. Fischer CN, Milton DR, Quiring SB (1980) Efficient LL(1) error correction and recovery using only insertions. Acta Inf 13(2):141–154. https://doi.org/10.1007/BF00263990
31. Flanagan C, Saxe JB (2001) Avoiding exponential explosion: generating compact verification conditions. In: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on principles of programming languages. ACM, New York, NY, USA, POPL '01, pp. 193–205
32. Framework S (2022) Scratch - imagine, program, share. https://scratch.mit.edu/
33. Gao M, He L, Majumdar R, et al (2016) LLSPLAT: improving concolic testing by bounded model checking. In: 2016 IEEE 16th international working conference on source code analysis and manipulation (SCAM), pp 127–136
34. Gao X, Mechtaev S, Roychoudhury A (2019) Crash-avoiding program repair. In: Zhang D, Møller A (eds) Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, ISSTA 2019, Beijing, China, July 15-19, 2019. ACM, pp. 8–18
35. Gao X, Wang B, Duck GJ et al (2021) Beyond tests: program vulnerability repair via crash constraint extraction. ACM Trans Softw Eng Methodol 30(2):1–27. https://doi.org/10.1145/3418461
36. Garg A, Roy S (2015) Synthesizing heap manipulations via integer linear programming. In: Blazy S, Jensen T (eds) Static analysis. Springer, Berlin Heidelberg, Berlin, Heidelberg, pp 109–127
37. Generator LP (2022a) Ll(1) parser generator. https://www.cs.princeton.edu/courses/archive/spring20/cos320/LL1/
38. Generator LP (2022b) LR(1) parser generator. http://jsmachines.sourceforge.net/machines/lr1.html
39. Golia P, Roy S, Meel KS (2020) Manthan: a data-driven approach for Boolean function synthesis. In: Computer aided verification: 32nd international conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II. Springer-Verlag, Berlin, Heidelberg, pp. 611–633
40. Golia P, Slivovsky F, Roy S, et al. (2021) Engineering an efficient Boolean functional synthesis engine. In: 2021 IEEE/ACM international conference on computer aided design (ICCAD), IEEE, pp. 1–9
41. Golia P, Roy S, Meel KS (2023) Synthesis with explicit dependencies. In: Proceedings of design, automation and test in Europe (DATE)
42. Gulwani S, Jha S, Tiwari A, et al (2011) Synthesis of loop-free programs. In: Proceedings of the 32nd ACM SIGPLAN conference on programming language design and implementation. Association for computing machinery, New York, NY, USA, PLDI '11, pp. 62–73
43. Gupta R, Pal S, Kanade A, et al (2017) Deepfix: fixing common C language errors by deep learning. In: Thirty-First AAAI conference on artificial intelligence
44. Gupta R, Kanade A, Shevade S (2019) Deep reinforcement learning for syntactic error repair in student programs. In: Proceedings of the AAAI conference on artificial intelligence, pp. 930–937
45. Hamilton G (2009) CA448 - compiler construction 1. http://www.computing.dcu.ie/~hamilton/teaching/CA448/testcases/index.html, online; accessed 11 November 2017
46. Hoare CAR (1969) An axiomatic basis for computer programming. Commun ACM 12(10):576–580. https://doi.org/10.1145/363235.363259
47. Hopcroft JE, Motwani R, Ullman JD (2001) Introduction to automata theory, languages, and computation. SIGACT News 32(1):60–65. https://doi.org/10.1145/568438.568455
48. Isohanni E, Järvinen HM (2014) Are visualization tools used in programming education? by whom, how, why, and why not? Koli calling '14. ACM, New York, NY, USA, pp 35–40
49. Isradisaikul C, Myers AC (2015) Finding counterexamples from parsing conflicts. In: Proceedings of the 36th ACM SIGPLAN conference on programming language design and implementation. ACM, New York, NY, USA, PLDI '15, pp. 555–564
50. Jain R, Aggarwal SK, Jalote P et al (2004) An interactive method for extracting grammar from programs. Softw Pract Exper 34(5):433–447. https://doi.org/10.1002/spe.v34:5
51. JavaCC (1996) JavaCC: the Java parser generator. https://javacc.org/, online; accessed 11 November 2017
52. Javed F, Bryant BR, Črepinšek M, et al (2004) Context-free grammar induction using genetic programming. In: Proceedings of the 42Nd annual southeast regional conference. ACM, New York, NY, USA, ACM-SE 42, pp. 404–405
53. JELLRAP (2008) Jellrap. https://users.cs.duke.edu/~rodger/tools/jellrap/index.html
54. JFLAP (2018) Jflap tool. https://www.jflap.org/

55. Jha S, Gulwani S, Seshia SA, et al (2010) Oracle-guided component-based program synthesis. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering - Vol 1. Association for computing machinery, New York, NY, USA, ICSE '10, pp. 215–224
56. Johnson SC (1975) Yacc: yet another compiler-compiler, vol 32
57. Jose M, Majumdar R (2011) Cause clue clauses: error localization using maximum satisfiability. In: Proceedings of the 32Nd ACM SIGPLAN conference on programming language design and implementation. ACM, New York, NY, USA, PLDI '11, pp. 437–446
58. Kadioglu S, Sellmann M (2010) Grammar constraints. Constraints 15(1):117–144. https://doi.org/10.1007/s10601-009-9073-4
59. Kalita PK, Kumar MJ, Roy S (2022) Synthesis of semantic actions in attribute grammars. In: conference on formal methods in computer-aided design–FMCAD 2022, p. 304
60. Katsirelos G, Maneth S, Narodytska N et al (2009) Restricted global grammar constraints. In: Gent IP (ed) Principles and practice of constraint programming - CP 2009. Springer, Berlin Heidelberg, Berlin, Heidelberg, pp 501–508
61. Lal A, Qadeer S, Lahiri SK (2012) A solver for reachability modulo theories. In: Proceedings of the 24th international conference on computer aided verification. Springer-Verlag, Berlin, Heidelberg, CAV'12, pp. 427–443
62. Lämmel R, Verhoef C (2001) Cracking the 500-language problem. IEEE Softw 18(6):78–88. https://doi.org/10.1109/52.965809
63. Lämmel R, Verhoef C (2001) Semi-automatic grammar recovery. Softw Pract Exp 31(15):1395–1448. https://doi.org/10.1002/spe.423
64. Lee J, Hong S, Oh H (2018) Memfix: static analysis-based repair of memory deallocation errors for c. In: Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering. Association for computing machinery, New York, NY, USA, ESEC/FSE 2018, pp. 95–106
65. Leung A, Lerner S (2017) Parsimony: an IDE for example-guided synthesis of lexers and parsers. In: Proceedings of the 32Nd IEEE/ACM international conference on automated software engineering. IEEE Press, Piscataway, NJ, USA, ASE 2017, pp. 815–825
66. Leung A, Sarracino J, Lerner S (2015) Interactive parser synthesis by example. In: Proceedings of the 36th ACM SIGPLAN conference on programming language design and implementation. ACM, New York, NY, USA, PLDI '15, pp. 565–574
67. Lewis PMII, Stearns RE (1968) Syntax-directed transduction. J ACM 15(3):465–488. https://doi.org/10.1145/321466.321477
68. Liblit B, Naik M, Zheng AX, et al (2005) Scalable statistical bug isolation. In: Sarkar V, Hall MW (eds) Proceedings of the ACM SIGPLAN 2005 conference on programming language design and implementation, Chicago, IL, USA, June 12-15, 2005. ACM, pp. 15–26
69. Liu C, Fei L, Yan X et al (2006) Statistical debugging: a hypothesis testing-based approach. IEEE Trans Softw Eng 32(10):831–848. https://doi.org/10.1109/TSE.2006.105
70. Liu Y, Li B (2010) Automated program debugging via multiple predicate switching. In: Proceedings of the twenty-fourth AAAI conference on artificial intelligence. AAAI Press, AAAI'10, pp. 327–332
71. Madhavan R, Mayer M, Gulwani S, et al (2015) Automating grammar comparison. In: Proceedings of the 2015 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications. ACM, New York, NY, USA, OOPSLA 2015, pp. 183–200
72. McGill MM, Decker A (2020) Tools, languages, and environments used in primary and secondary computing education. In: ITiCSE '20. ACM, New York, NY, USA
73. McMillan KL (2006) Lazy abstraction with interpolants. In: Proceedings of the 18th international conference on computer aided verification. Springer-Verlag, Berlin, Heidelberg, CAV'06, pp. 123–136
74. Mechtaev S, Yi J, Roychoudhury A (2015) Directfix: looking for simple program repairs. In: Proceedings of the 37th international conference on software engineering - Vol 1. IEEE Press, Piscataway, NJ, USA, ICSE '15, pp. 448–458
75. Mechtaev S, Yi J, Roychoudhury A (2016) Angelix: scalable multiline program patch synthesis via symbolic analysis. In: Dillon LK, Visser W, Williams LA (eds) Proceedings of the 38th international conference on software engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016. ACM, pp. 691–701
76. Mernik M, Gerlič G, Žumer V, et al (2003) Can a parser be generated from examples? In: Proceedings of the 2003 ACM symposium on applied computing. ACM, New York, NY, USA, SAC '03, pp. 1063–1067

77. Modi V, Roy S, Aggarwal SK (2013) Exploring program phases for statistical bug localization. In: Proceedings of the 11th ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering. Association for computing machinery, New York, NY, USA, PASTE '13, pp. 33–40
78. Moskal M (2009) Programming with triggers. In: Proceedings of the 7th international workshop on satisfiability modulo theories. ACM, New York, NY, USA, SMT '09, pp. 20–29
79. Moskal M, Lopuszanski J, Kiniry JR (2008) E-matching for fun and profit. Electron Notes Theor Comput Sci 198(2):19–35. https://doi.org/10.1016/j.entcs.2008.04.078
80. Nakamura K (2006) 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. Springer-Verlag, Berlin, Heidelberg, ICGI'06, pp. 72–83
81. Nakamura K, Ishiwata T (2000) 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. Springer-Verlag, London, UK, UK, ICGI '00, pp. 186–195
82. Nakamura K, Matsumoto M (2002) Incremental learning of context free grammars. In: Proceedings of the 6th international colloquium on grammatical inference: algorithms and applications. Springer-Verlag, London, UK, UK, ICGI '02, pp. 174–184
83. Nakamura K, Matsumoto M (2005) Incremental learning of context free grammars based on bottom-up parsing and search. Pattern Recogn 38(9):1384–1392
84. Pandey A, Kotcharlakota PRG, Roy S (2019) Deferred concretization in symbolic execution via fuzzing, association for computing machinery, New York, NY, USA, pp. 228–238
85. Pingali K (2010) Top-down parsing. https://www.cs.utexas.edu/~pingali/CS375/2010Sp/lectures/LL1.pdf, online; accessed 11 November 2017
86. Polikarpova N, Sergey I (2019) Structuring the synthesis of heap-manipulating programs. Proc ACM Program Lang. https://doi.org/10.1145/3290385
87. Qiu X, Solar-Lezama A (2017) Natural synthesis of provably-correct data-structure manipulations. Proc ACM Program Lang. https://doi.org/10.1145/3133889
88. Quimper CG, Walsh T (2008) Decompositions of grammar constraints. In: Proceedings of the 23rd national conference on artificial intelligence - Vol 3. AAAI Press, AAAI'08, pp. 1567–1570
89. Redmond C (2017) Tiger parser. http://www.credmond.net/projects/tiger-parser/, online; accessed 11 November 2017
90. Roy S (2013) From concrete examples to heap manipulating programs. In: Logozzo F, Fähndrich M (eds) Static analysis: 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings, lecture notes in computer science, vol 7935. Springer, pp. 126–149
91. Roy S, Hsu J, Albarghouthi A (2021) Learning differentially private mechanisms. In: 42nd IEEE symposium on security and privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021. IEEE, pp. 852–865
92. Saha D, Narula V (2011) Gramin: a system for incremental learning of programming language grammars. In: Proceedings of the 4th India software engineering conference. ACM, New York, NY, USA, ISEC '11, pp. 185–194
93. Sellink A, Verhoef C (1999) Generation of software renovation factories from compilers. In: 1999 Proceedings IEEE international conference on software maintenance (ICSM '99) , pp. 245–255
94. Sharfuddeen Zubair M, Brown D, Bates M, et al (2020) Are visual programming tools for children designed with accessibility in mind? In: ICETC'20. ACM, New York, NY, USA
95. Shariffdeen RS, Noller Y, Grunske L, et al (2021) Concolic program repair. In: Freund SN, Yahav E (eds) PLDI '21: 42nd ACM SIGPLAN international conference on programming language design and implementation, virtual event, Canada, June 20-25, 2021. ACM, pp. 390–405
96. Singal D, Agarwal P, Jhunjhunwala S, et al (2018) Parse condition: symbolic encoding of LL(1) parsing. In: Barthe G, Sutcliffe G, Veanes M (eds) LPAR-22. 22nd international conference on logic for programming, artificial intelligence and reasoning, EPiC series in computing, vol 57. EasyChair, pp. 637–655
97. Solar-Lezama A, Rabbah R, Bodík R, et al (2005) Programming by sketching for bit-streaming programs. In: Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation. Association for computing machinery, New York, NY, USA, PLDI '05, pp. 281–294
98. StackExchange (2012a) Is this language LL(1) parseable? http://cs.stackexchange.com/questions/3350/is-this-language-ll1-parseable, online; accessed 11 November 2017
99. StackExchange (2012b) Left-factoring a grammar into LL(1). http://cs.stackexchange.com/questions/4862/left-factoring-a-grammar-into-ll1, online; accessed 11 November 2017

100. Suh S (2022) Codetoon: a new visual programming environment using comics for teaching and learning programming. In: SIGCSE 2022. ACM, New York, NY, USA
101. Takhar G, Karri R, Pilato C et al (2022) Holl: program synthesis for higher order logic locking. In: Fisman D, Rosu G (eds) Tools and algorithms for the construction and analysis of systems. Springer, Cham
102. Tarski A (1955) A lattice-theoretical fixpoint theorem and its applications. Pacific J Math 5(2):285–309
103. The Coq Development Team (2022) Coq. https://coq.inria.fr
104. van Tonder R, Goues CL (2018) Static automated program repair for heap properties. In: Proceedings of the 40th international conference on software engineering. Association for computing machinery, New York, NY, USA, ICSE '18, pp. 151–162
105. Črepinšek M, Mernik M, Bryant BR et al (2005) Inferring context-free grammars for domain-specific languages. Electron Notes Theor Comput Sci 141(4):99–116. https://doi.org/10.1016/j.entcs.2005.02.055
106. Črepinšek M, Mernik M, Javed F et al (2005) Extracting grammar from programs: evolutionary approach. SIGPLAN Not 40(4):39–46. https://doi.org/10.1145/1064165.1064172
107. Črepinšek M, Mernik M, Žumer V (2005) Extracting grammar from programs: brute force approach. SIGPLAN Not 40(4):29–38. https://doi.org/10.1145/1064165.1064171
108. Verma A, Kalita PK, Pandey A et al (2020) Interactive debugging of concurrent programs under relaxed memory models. Association for Computing Machinery, New York, NY, USA, pp. 68–80
109. Verma S, Roy S (2017) Synergistic debug-repair of heap manipulations. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering. Association for computing machinery, New York, NY, USA, ESEC/FSE 2017, pp. 163–173
110. Verma S, Roy S (2022) Debug-localize-repair: a symbiotic construction for heap manipulations. Formal Methods Syst Des. https://doi.org/10.1007/s10703-021-00387-z
111. Vidal Duarte E (2016) Teaching the first programming course with python's turtle graphic library. In: ITiCSE '16. ACM, New York, NY, USA
112. Wang C, Cheung A, Bodik R (2017) Synthesizing highly expressive sql queries from input-output examples. In: Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation. Association for computing machinery, New York, NY, USA, PLDI 2017, pp. 452–466
113. Wang Y, Dong J, Shah R, et al (2019) Synthesizing database programs for schema refactoring. In: Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation. Association for computing machinery, New York, NY, USA, PLDI 2019, pp. 286–300
114. Wang Y, Shah R, Criswell A et al (2020) Data migration using datalog program synthesis. Proc VLDB Endow 13(7):1006–1019. https://doi.org/10.14778/3384345.3384350
115. Wang Y, Ding Z, Xiao Y, et al (2021) Dpgen: automated program synthesis for differential privacy. In: Proceedings of the 2021 ACM SIGSAC conference on computer and communications security. Association for computing machinery, New York, NY, USA, CCS '21, pp. 393–411
116. Watanabe Y, Gopinathan K, Pîrlea G et al (2021) Certifying the synthesis of heap-manipulating programs. Proc ACM Program Lang. https://doi.org/10.1145/3473589
117. Yi J, Ahmed UZ, Karkare A, et al (2017) A feasibility study of using automated program repair for introductory programming assignments. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering. Association for computing machinery, New York, NY, USA, ESEC/FSE 2017, pp. 740–751
118. van Zijl L (2017) CS711: Implementation and Application of Automata. http://www.cs.sun.ac.za/rw711/lectures/lec4/l4.pdf, online; accessed 11 November 2017