EPiC Series in Computing

Volume 57, 2018, Pages 637–655

LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning

Parse Condition: Symbolic Encoding of LL(1) Parsing

Dhruv Singal[*], Palak Agarwal[†], Saket Jhunjhunwala, and Subhajit Roy

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur, India
singaldhruv.ds@gmail.com, palak8669@gmail.com,
saketj@iitk.ac.in, subhajit@cse.iitk.ac.in

Abstract

In this work, we propose the notion of a *Parse Condition*—a logical condition that is satisfiable if and only if a given string w can be successfully parsed using a grammar $\mathcal{G}$. Further, we propose an algorithm for building an SMT encoding of such parse conditions for LL(1) grammars and demonstrate its utility by building two applications over it: *automated repair of syntax errors* in Tiger programs and *automated parser synthesis* to automatically synthesize LL(1) parsers from examples. We implement our ideas into a tool, CYCLOPS, that is able to successfully repair 80% of our benchmarks (675 buggy Tiger programs), clocking an average of 30 seconds per repair and synthesize parsers for interesting languages from examples. Like *verification conditions* (encoding a program in logic) have found widespread applications in program analysis, we believe that *Parse Conditions* can serve as a foundation for interesting applications in syntax analysis.

1 Introduction

Writing and debugging parsers is remarkably painful—debugging conflicts in grammars remains an arduous activity even for experienced language designers [21]. The difficulty stems from the fact that all efficiently parsable languages, like LL(1), 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; successful construction of an *LL(1) parse table* is the test that a grammar needs to pass to qualify as LL(1). In general, inferring whether an LL(1) grammar exists for a given language is **undecidable** [42]. Therefore, designing an LL(1) grammar for a given programming language construct is non-trivial; it takes multiple debugging cycles on parse table *conflicts* to design an LL(1) grammar. Not just programming language designers, but even students of courses in Compiler Theory also struggle when exposed to parsing algorithms [44, 43].

In this paper, we propose the notion of a *Parse Condition* and also outline an algorithm to generate the same. Given a grammar $\mathcal{G}$ and an input string w, a *parse condition* is a logical formula that is satisfiable if and only if the parser (of a certain type, LL(1) in this paper) built using $\mathcal{G}$ accepts the string w. Readers familiar with symbolic verification should be able

[*]Now with Columbia University, USA
[†]Now with WorldQuant Research, India

to draw parallels between *parse conditions* and *verification conditions* [9, 17] used in program verification: given a program $\mathcal{P}$ and a property Φ, the verification condition encodes the logical constraints for $\mathcal{P}$ to satisfy Φ. Like verification conditions have found immense applications in verification, synthesis and repair of programs, we found that *parse conditions* have applications in synthesis and repair of parsers.

To demonstrate the practical utility of *parse conditions*, we build a tool, Cyclops, that encodes the parse condition for LL(1) parsers as constraints for 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. This encoding of the *parse conditions* in Cyclops can be used in:

- ◦ **Programming Language Design** Programming language designers can use Cyclops for design exploration of *parsable* language constructs, to create languages or language extensions; Cyclops can ensure that the grammar necessarily remains LL(1) parsable and provides debugging support for conflicts (using formal artifacts like unsat cores and interpolants).
- ⋆ **Automated Parser Synthesis** Cyclops can be used to generate LL(1) parsers (and grammars)—automatically—from a set of example strings in the language (§6.2);
- ⋆ **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 (§6.1);
- ◦ **Computer Science Education** The symbolic encoding in Cyclops can encourage building of tools that assist students in understanding the complexities of LL(1) parsers; instructors can use such tools to automatically generate problem statements with certain properties (e.g. grammars that are LL(2) but not LL(1)).

Please note that the purpose of this symbolic encoding is *not* LL(1) parsing (there exist efficient polynomial time algorithms for the same) but the interesting applications enumerated above. To the best of our knowledge, this is the first attempt at building symbolic encoding of parser constraints; hence, there was no possibility for a comparison with another tool. We, instead, answer the following research questions:

RQ1 [**Usefulness**] Is the encoding useful at building interesting applications? We answer this by building two interesting applications (marked by ⋆ above) over Cyclops in §6.2;

RQ2 [**Scalability**] Is the encoding efficient enough for real-world applications? We were able to build a tool for repairing syntax errors for programs in the Tiger language [4], spanning more than 90 productions (§6.1). Cyclops is able to successfully repair 80% of our benchmarks, clocking an average of 30 seconds per repair.

We make the following contributions in this work:

- We define the notion of a *parse condition* to logically encode syntax analysis;
- We develop an algorithm for constructing parse conditions and instantiate our algorithm into our tool Cyclops;
- We demonstrate practical applications of our tool by building modules for **automated parser synthesis** and **repair of syntax errors**.

2 Background: LL(k) Parsing

The classical LL(k) parser [31] 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 via the use of *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 right, using **k** tokens of *lookahead* (for prediction), constructing the **L**eftmost derivation.

2.1 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 $X \to \alpha$ 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.

2.2 Parse table conflicts

Successful construction of LL(k) parse table certifies that the grammar as 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 $\langle X, \alpha \rangle$ pair, where X is the non-terminal candidate for expansion and α 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).

3 Parse Condition: An Overview

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 functions $\texttt{head}(r)$ and $\texttt{body}(r)$ to return the head and body for rule r respectively. We also assume $\texttt{symbol}(r)$ to return all the symbols (head as well as the body) for 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, if Ω is satisfiable if and only if:

- The grammar 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** (simply referred to as *Parse Condition* subsequently) that is satisfiable if and only if $\mathcal{G}$ is an LL(1) grammar and the corresponding parser accepts ω.

The *parse condition* is composed of the following:

- $\mathcal{G}$, ω and λ: Let $\mathcal{G}$, ω, and λ be symbolic placeholders for 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 other. 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 (repairing syntax errors).
- *ParseTable*: 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*: 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 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.

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

- *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 §5.1.
- *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 §5.3.

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).

$$ParseCondition(\mathcal{G}, \omega) \equiv \exists \lambda.\ ParseTable(\mathcal{G}, \lambda) \wedge Parser(\mathcal{G}, \lambda, \omega)$$
$$ParseTable(\mathcal{G}, \lambda) \equiv First(\mathcal{G}) \wedge Follow(\mathcal{G}) \wedge \neg Conflict_{LL(1)}(\mathcal{G}, \lambda)$$

Cyclops constructs a symbolic encoding for LL(1) as a set of SMT constraints. Figure 1 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 accept the string (see §6.2); asserting a grammar allows one to generate a string (or repair a syntactically incorrect string) that is consistent with the provided grammar (see §6.1).

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 via hyperparameters for the size and shape of the production rules (maximum number of productions, terminals, non-terminals and the maximum size of a 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

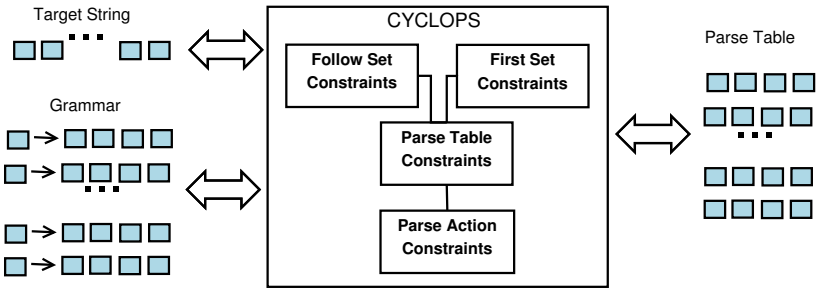


Figure 1: Overview of Cyclops

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

Cyclops is meant to be 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*. Being a symbolic encoding, our algorithm (rather encoding) operates differently than the stack-based classic LL(1) parser: for example, we simulate rule selection from the parse table by a fixpoint constraint using witness encodings and the parsing tree as an embedding on an 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 are correct, i.e. the set of "guesses" satisfy all the constraints. Please 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 linear array in a parse tree. Our symbolic parsing actions compute such a linear array, thereby generating the corresponding parse tree. Figure 2 provides a visual insight on the embedding.

For the start symbol S, we add an additional production $S' \rightarrow S\$$ with the 'extended' start symbol S'; as this production is only applied once, 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 a linear array in a parse tree

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

- $\mathcal{A}_0$ is mapped to the root node in the parse tree;
- For any node $z \in \mathcal{Z}$ with children $c_1, c_2, \ldots, c_m, 0 \leq \mathit{index}(z) \leq \mathit{index}(c_i) \leq b$ and $0 \leq \mathit{index}(c_i) \leq \mathit{index}(c_m) \leq b$, for each $i < m$ (where $\mathit{index}(\mathcal{A}_i) = i$);
- 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 Figure 2).

In other words, $\mathcal{A}_i$ maps to the parse tree node that will be the ith 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 $\mathcal{A}_i$, $i > |\mathcal{Z}|$ 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 Figure 2, 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.

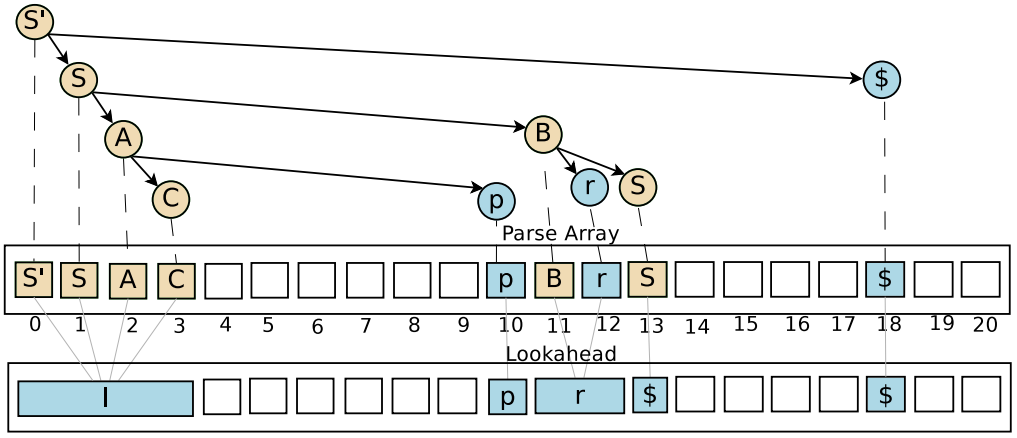
4.2 Intuition on the embedding

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 of maintaining a symbolic stack at each *step* of the parser.

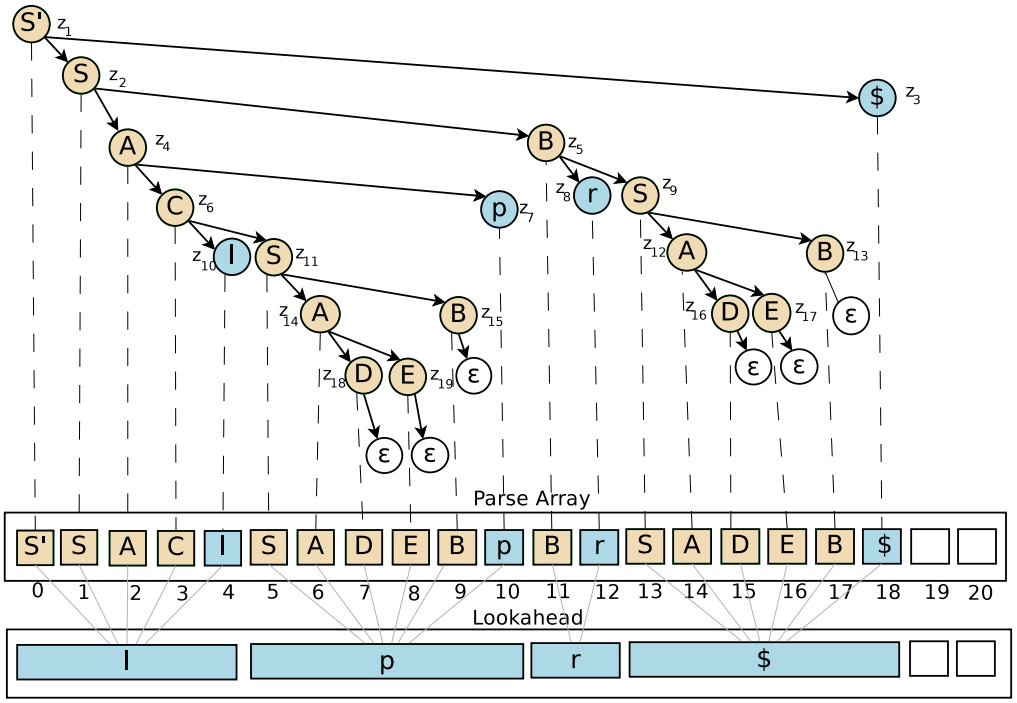
Given a string $w \in \mathcal{L}(\mathcal{G})$, our symbolic parsing algorithm essentially searches for a valid embedding (of a parse tree) in a bounded size array that witnesses the derivation of w from the grammar $\mathcal{G}$. We impose additional constraints to ensure that the tree is constructed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

We demonstrate the process using the example in Figure 2 and the grammar in Figure 4. 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 then 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



(a) Intermediate stage



(b) Parsing complete

Figure 2: Embedding of a parse tree

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 (Figure 2 via the dashed lines), we accept this sequence of guesses (or non-deterministic choices) as a valid parse. In our algorithm, the constraints of the embedding are encoded as constraints and we use an SMT solver to infer the "correct" guesses that produce a valid embedding.

$$\forall_{0 \leq i \leq |\texttt{parseArray}|}.(i < \texttt{prefixLimit}) \implies \texttt{ValidAction}(i) \tag{1}$$

$$\begin{aligned}\texttt{ValidAction}(i) = [\{\texttt{parseArray}(i) \in \mathcal{T}\} \wedge \{\texttt{parseArray}(i) = \texttt{inputString}(\texttt{ip}(i))\} \\ \wedge \{\texttt{ip}(i+1) = \texttt{ip}(i) + 1\}] \vee \\ [\{\texttt{parseArray}(i) \in \mathcal{N}\} \wedge \\ \{\exists r\{\exists i_1, i_2, \ldots, i_n . i \leq i_1 \leq i_2 \ldots i_n.\ \ \texttt{ApplyProd}(r, i, i_1, \ldots, i_n)\} \\ \wedge\{\texttt{ip}(i+1) = \texttt{ip}(i)\} \wedge\{\texttt{ParseTable}(\texttt{head}(r), \texttt{inputString}(\texttt{ip}(i))) = r\}\}]\end{aligned} \tag{2}$$

$$\begin{aligned}\texttt{ApplyProd}(r, i, i_1, \ldots, i_n) = \bigvee_{k \in \mathcal{P}} [r = \langle k : X \to Y_1 \ldots Y_n \rangle \wedge \texttt{end}(i) = \texttt{end}(i_n) \\ \wedge\ \ i_1 = i + 1\ \ \wedge (\texttt{end}(i_n) < \texttt{end}(0))\ \ \wedge \\ \bigwedge_{l \in \{1 \ldots |\texttt{body}(r)|-1\}} \texttt{ApplySymb}(Y_l, i_l, i_{l+1})]\end{aligned} \tag{3}$$

$$\texttt{ApplySymb}(Y_l, i_l, i_{l+1}) = \begin{cases} (\texttt{parseArray}(i_l) = Y_l) \wedge (\texttt{end}(i_l) = i_{l+1} = i_l + 1), & \text{if } Y_l \in \mathcal{T} \cup \{\$\} \\ (i_{l+1} = i_l), & \text{if } Y_l = \epsilon \\ (\texttt{parseArray}(i_l) = Y_l) \wedge \texttt{end}(i_l) = i_{l+1}, & \text{if } Y_l \in \mathcal{N} \end{cases} \tag{4}$$

$$(\texttt{parseArray}(0) = S') \wedge (\texttt{ip}(0) = 0) \wedge (\texttt{end}(0) < |\texttt{parseArray}|) \tag{5}$$

Figure 3: The parse action constraints

4.3 The encoding of the parsing algorithm

The following are the relevant functions that are used by our constraint system (Figure 3):

- `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 (Eqn. 1); in other words, it ensures that the parse array is a valid embedding in 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 (Eqn. 1).

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.

The predicate `ValidAction` works as follows (Eqn. 2).

- 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 $\langle r : X \to Y_1 \dots Y_n \rangle$ to expand in consultation with the `ParseTable`, 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` (Eqn. 3) encodes the application of the selected rule $\langle r : X \to Y_1 \dots Y_n \rangle$ by setting constraints on each symbol in the body, Y_i, of the rule using `ApplySymb`—enforcing the following constraints (Eqn. 4):
  - 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 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 (Eqn. 5).

5 Symbolic encoding of the LL(1) Parse Table

We encode an LL(1) parse table as a function, `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 of the first and follow sets

The first sets can be computed as the **least** fixpoint over these constraints:

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

$S \to A\ B$ [1]
$A \to C\ p$ [2]
$\quad\quad | \ D\ E$ [3]
$B \to r\ S$ [4]
$\quad\quad | \ \epsilon$ [5]
$C \to l\ S$ [6]
$D \to \epsilon$ [7]
$E \to \epsilon$ [8]

Figure 4: $\mathcal{G}_1$

$$\texttt{FirstSetWitness}_{\langle r,i\rangle}(X,t) = \begin{cases} t, & \text{if } Y_i = t \in \mathcal{T} \wedge \forall_{k=1}^{i-1} \texttt{FirstSet}(Y_k, \epsilon) \\ Y_i, & \text{if } Y_i \in \mathcal{N} \wedge \forall_{k=1}^{i-1} \texttt{FirstSet}(Y_k, \epsilon) \\ t, & \text{if } X = t \\ \mathit{nil}, & \text{otherwise} \end{cases} \tag{6}$$

$$\texttt{EpInFirst}_0(X) = \begin{cases} \mathit{true}, & \text{if there exists a rule } \langle X \to \epsilon \rangle \\ \mathit{false}, & \text{otherwise} \end{cases} \tag{7}$$

$$\texttt{EpInFirst}_i(X) = \begin{cases} \mathit{true}, & \text{if } \exists \langle X \to Y_1 Y_2 \ldots Y_n \rangle \text{ s.t. } \forall_{k=1}^{n} \texttt{EpInFirst}_{i-1}(Y_k) \\ \mathit{true}, & \text{if } \texttt{EpInFirst}_{i-1}(X) \\ \mathit{false}, & \text{otherwise} \end{cases} \tag{8}$$

Figure 5: First set witness functions

The first sets of the non-terminals can be computed as the **least** fixpoint 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 5 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:

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

We encode the predicate $\texttt{FirstSet}(X,t)$ such that it is satisfiable if and only if $t \in \mathit{First}(X)$. To infer whether $\epsilon \in \mathit{First}(X)$, we invoke $\texttt{EpInFirst}_{|\mathcal{N}|}(X)$ in a 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.

$$\texttt{FirstSet}(X,\epsilon) = \texttt{EpInFirst}_{|\mathcal{N}|}(X) \tag{9}$$

$$
\begin{aligned}
t \in \texttt{First}(X) \equiv \texttt{FirstSet}(X,t) = \exists \langle r_0, i_0\rangle, \langle r_1, i_1\rangle, \ldots, y_0, y_1, \ldots \\
(y_0 = \texttt{FirstSetWitness}_{\langle r_0, i_0\rangle}(X,t) \wedge X = \texttt{head}(r_0) \wedge y_0 \in \texttt{body}(r_0) \wedge y_0 \neq \epsilon) \wedge \\
(y_1 = \texttt{FirstSetWitness}_{\langle r_1, i_1\rangle}(y_0,t) \wedge y_0 = \texttt{head}(r_1) \wedge y_1 \in \texttt{body}(r_1) \wedge y_1 \neq \epsilon) \wedge \\
\ldots \\
(y_k = \texttt{FirstSetWitness}_{\langle r_k, i_k\rangle}(y_{k-1},t) \wedge y_{k-1} = \texttt{head}(r_k) \wedge y_k \in \texttt{body}(r_k) \wedge y_k \neq \epsilon \wedge y_k = t)
\end{aligned} \tag{10}
$$

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 Eqn. 10 attempts to *guess* 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 $t \in First(X)$.

The follow sets can be computed as the **least** fixpoint over these constraints:

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

We omit the details of the encoding for brevity.

5.2 Encoding least fixpoints as SMT constraints

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

$$a_1 = f(a_0) \wedge a_2 = f(a_1) \wedge \ldots a_i = f(a_{i+1}) \wedge a_i = a_{i+1} \tag{11}$$

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 [45] 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 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.

Please 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.

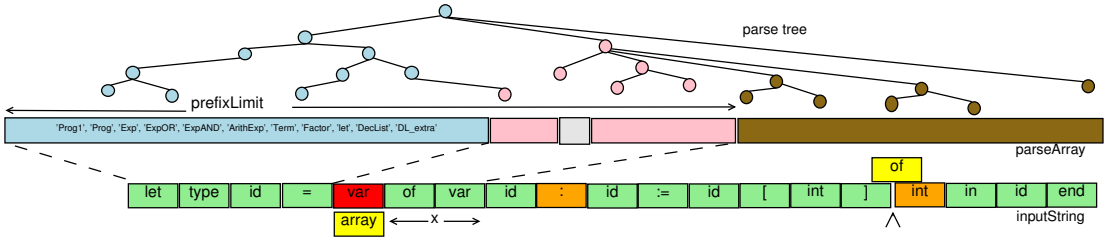


Figure 6: Repairing syntactic errors on the Tiger grammar

5.3 The encoding of the parse table

We encode $\texttt{ParseTable}(X, t)$ as (where FiS stands for FirstSet and FoS stands for FollowSet):

$$\bigwedge_{X \in \mathcal{N}, t \in \mathcal{T}, r_i \in \mathcal{P}} [\texttt{FiS}(\alpha, t) \vee (\texttt{FiS}(\alpha, \epsilon) \wedge \texttt{FoS}(\alpha, t))] \iff (\texttt{ParseTable}(X, t) = \langle r_i : X \to \alpha \rangle) \tag{12}$$

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 $\texttt{First}(\alpha_1, \text{t})$ causes $\texttt{ParseTable}(\text{X,t}) = r_1$ and $\texttt{First}(\alpha_2, \text{t})$ causes $\texttt{ParseTable}(\text{X,t}) = r_2$, 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.

6 Applications

We implemented Cyclops in Python using the Z3 4.4.2 [13] SMT solver. We ran our experiments on a Linux container with a 2.4 GHz Intel processor and 198 GB main memory. We now demonstrate the two 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 [38, 37, 3] based quantifier instantiation (disabled `mbqi` and `auto_config`) with carefully selected triggers via the `pattern` construct.

6.1 Repairing Tiger Programs

Given a string w as a sequence of tokens[1] $[w_0, \ldots, w_n]$ 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).

[1] *Tokens* in the parser correspond to terminals in the grammar.

The Tiger programming language [4] grammar contains 91 productions, with the size of their bodies having at most 5 symbols. We use JavaCC [23] 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 [41] (as the given grammar was not exactly LL(1)).

6.1.1 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, 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.

6.1.2 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 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 Figure 6 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 find 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.

We request CYCLOPS to synthesize the `parseArray` upto x symbols *past* a suspicious location (which are also asserted to the necessary terminals) to enable enough context for a successful parse; for our experiments, small values of 5 for the initial and 3 for the last few locations were enough to give us a good repair rate. 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. It is possible that a prefix α is valid (anticipating a following suffix β such that $\alpha\beta \in L(\mathcal{G})$), but the actual suffix γ, $\alpha\gamma \notin L(\mathcal{G})$: in such cases, CYCLOPS backtracks a bounded number of times to synthesize a different repair.

To enforce validity check on the prefix, we set the value for `prefixLimit` depending on the selected value of x instead of the end of the string in Eqn 3.

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.

6.1.3 Evaluation

We evaluated our scheme on a set of benchmarks (Tiger programs) from [19]. 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 1 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 minutes; 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 seconds** per repair.

Table 1: 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

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 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 minutes, with an average repair time of 48.5 seconds. 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.

6.2 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 [7, 1] and online course notes [2, 46, 39, 15, 10]. Table 2 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), 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 report the minimum time taken (Min) and the average time taken (Avg) for these five runs. Though in many of them the differences in the results are small, we found cases (in blue) where one of the setting outperforms the other setting significantly.

Table 2: Parser synthesis (runtimes in seconds)

		Incremental		All-at-once	
(P,B,T,N)	(E,L)	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

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). 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).

7 Related Work

To the best of our knowledge, there is no prior work that attempts symbolic encoding of parser constraints. We discuss and compare related ideas from other domains (for example, programs and grammars) in this section. Please note that there are parser generators (like Yacc [24]) 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).

For synthesizing parsers from a set of examples: Jain *et al.* [22] use a user-defined knowledge-base of grammar rules to drive a backtracking based search through a bottom-up parser to discover a possible parse; in this scheme, success of the algorithm depends on user-interaction in terms of construction of a good knowledge base of grammar rules to be used in the search. Mernick *et al.* [36] 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 [30] 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 [29] improves upon Parsify with 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 [14, 16, 11]. 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, CFG-Analyzer [5] provides a SAT encoding for answering questions like inclusion, intersection and equivalence for context-free grammars; the SAT encoding is designed to search for a string of a bounded size that satisfies a given property (like a counterexample that two provided grammars are not equivalent). For the 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 [26, 40, 27] that again use the CYK algorithm for building constraint solvers providing context-free grammar constraints. Madhavan *et al.* [33] propose a solution to the problem of grammar equivalence by giving an effective testing scheme which efficiently enumerates strings from a given language and a decision procedure for proving equivalence of two grammars which is complete for LL grammars.

Repair tools for semantic errors using symbolic techniques (like [32, 47]) 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. [21] attempt to assist in debugging conflicts by counterexamples from a parser run. As a *parse conditions* 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).

8 Discussion

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 parsable by a given parser. Like verification conditions have found extensive applications in verification [9, 28], debugging [6, 25, 32], repair [35] and testing [12, 18] 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 couple of preliminary applications of *parse conditions* to illustrate its utility at solving interesting problems. However, similar to verification conditions, which though being conceived by Hoare's axiomatic formulation in 1969 [20], is still testing researchers on applications to large problems even after being in existence for about 50 years (while fueling innovations like abstraction-refinement[8] and proof-guided abstractions[34]), *parsing conditions* would also need further innovations to be applied to still larger problems. This paper is just the first step!

References

[1] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

[2] Stephen J. Allan. LL Parsing. http://digital.cs.usu.edu/~allan/Compilers/Notes/LLParsing.pdf, 2017. Online; accessed 11 November 2017.

[3] Nada Amin, K. Rustan M. Leino, and Tiark Rompf. Computing with an smt solver. In Martina Seidl and Nikolai Tillmann, editors, *Tests and Proofs*, pages 20–35, Cham, 2014. Springer International Publishing.

[4] Andrew W Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 2004.

[5] Roland Axelsson, Keijo Heljanko, and Martin Lange. Analyzing context-free grammars using an incremental sat solver. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part II*, ICALP '08, pages 410–422, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] Rohan Bavishi, Awanish Pandey, and Subhajit Roy. To be precise: Regression aware debugging. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 897–915, New York, NY, USA, 2016. ACM.

[7] John C. Beatty. On the Relationship Between LL(1) and LR(1) Grammars. *J. ACM*, 29(4):1007–1022, October 1982.

[8] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, pages 154–169, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[9] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[10] Robin Cockett. Compiler Construction I: LL(1) grammars and predictive top-down parsing. http://pages.cpsc.ucalgary.ca/~robin/class/411/LL1.2.html, 2002. Online; accessed 11 November 2017.

[11] Rafael Corchuelo, José A. Pérez, Antonio Ruiz, and Miguel Toro. Repairing syntax errors in LR parsers. *ACM Trans. Program. Lang. Syst.*, 24(6):698–710, November 2002.

[12] Przemyslaw Daca, Ashutosh Gupta, and Thomas A. Henzinger. Abstraction-driven concolic testing. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, VMCAI 2016, pages 328–347, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] C. N. Fischer, D. R. Milton, and S. B. Quiring. Efficient LL(1) error correction and recovery using only insertions. *Acta Informatica*, 13(2):141–154, Feb 1980.

[15] Charles N. Fischer. Introduction to Programming Languages and Compilers. http://pages.cs.wisc.edu/~fischer/cs536.f13/lectures/f12/Lecture22.4up.pdf, 2013. Online; accessed 11 November 2017.

[16] Charles N. Fischer and Jon Mauney. A simple, fast, and effective LL(1) error repair algorithm. *Acta Informatica*, 29(2):109–120, Feb 1992.

[17] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 193–205, New York, NY, USA, 2001. ACM.

[18] Min Gao, Lei He, Rupak Majumdar, and Zilong Wang. LLSPLAT: Improving Concolic Testing by Bounded Model Checking. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 127–136, Oct 2016.

[19] Geoff Hamilton. CA448 - Compiler Construction 1. http://www.computing.dcu.ie/~hamilton/teaching/CA448/testcases/index.html, 2009. Online; accessed 11 November 2017.

[20] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[21] Chinawat Isradisaikul and Andrew C. Myers. Finding counterexamples from parsing conflicts. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 555–564, New York, NY, USA, 2015. ACM.

[22] Rahul Jain, Sanjeev Kumar Aggarwal, Pankaj Jalote, and Shiladitya Biswas. An interactive method for extracting grammar from programs. *Softw. Pract. Exper.*, 34(5):433–447, April 2004.

[23] JavaCC. JavaCC: The Java Parser Generator. https://javacc.org/, 1996. Online; accessed 11 November 2017.

[24] Stephen C Johnson. *Yacc: Yet another compiler-compiler*, volume 32. 1975.

[25] Manu Jose and Rupak Majumdar. Cause clue clauses: Error localization using maximum satisfiability. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 437–446, New York, NY, USA, 2011. ACM.

[26] Serdar Kadioglu and Meinolf Sellmann. Grammar constraints. *Constraints*, 15(1):117–144, January 2010.

[27] George Katsirelos, Sebastian Maneth, Nina Narodytska, and Toby Walsh. Restricted global grammar constraints. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, pages 501–508, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[28] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 427–443, Berlin, Heidelberg, 2012. Springer-Verlag.

[29] Alan Leung and Sorin Lerner. Parsimony: An ide for example-guided synthesis of lexers and parsers. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 815–825, Piscataway, NJ, USA, 2017. IEEE Press.

[30] Alan Leung, John Sarracino, and Sorin Lerner. Interactive parser synthesis by example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Im-*

plementation, PLDI '15, pages 565–574, New York, NY, USA, 2015. ACM.

[31] Philip M. Lewis, II and Richard E. Stearns. Syntax-directed transduction. *J. ACM*, 15(3):465–488, July 1968.

[32] Yongmei Liu and Bing Li. Automated program debugging via multiple predicate switching. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI'10, pages 327–332. AAAI Press, 2010.

[33] Ravichandhran Madhavan, Mikaël Mayer, Sumit Gulwani, and Viktor Kuncak. Automating grammar comparison. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 183–200, New York, NY, USA, 2015. ACM.

[34] Kenneth L. McMillan. Lazy abstraction with interpolants. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 123–136, Berlin, Heidelberg, 2006. Springer-Verlag.

[35] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 448–458, Piscataway, NJ, USA, 2015. IEEE Press.

[36] Marjan Mernik, Goran Gerlič, Viljem Žumer, and Barrett R. Bryant. Can a parser be generated from examples? In *Proceedings of the 2003 ACM Symposium on Applied Computing*, SAC '03, pages 1063–1067, New York, NY, USA, 2003. ACM.

[37] MichałMoskal. Programming with triggers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, SMT '09, pages 20–29, New York, NY, USA, 2009. ACM.

[38] MichałMoskal, Jakub Lopuszanski, and Joseph R. Kiniry. E-matching for fun and profit. *Electron. Notes Theor. Comput. Sci.*, 198(2):19–35, May 2008.

[39] Keshav Pingali. Top-down parsing. https://www.cs.utexas.edu/~pingali/CS375/2010Sp/lectures/LL1.pdf, 2010. Online; accessed 11 November 2017.

[40] Claude-Guy Quimper and Toby Walsh. Decompositions of grammar constraints. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, pages 1567–1570. AAAI Press, 2008.

[41] Cormac Redmond. Tiger parser. http://www.credmond.net/projects/tiger-parser/, 2017. Online; accessed 11 November 2017.

[42] D. J. Rosenkrantz and R. E. Stearns. Properties of deterministic top down grammars. In *Proceedings of the First Annual ACM Symposium on Theory of Computing*, STOC '69, pages 165–180, New York, NY, USA, 1969. ACM.

[43] StackExchange. Is this language LL(1) parseable? http://cs.stackexchange.com/questions/3350/is-this-language-ll1-parseable, 2012. Online; accessed 11 November 2017.

[44] StackExchange. Left-factoring a grammar into LL(1). http://cs.stackexchange.com/questions/4862/left-factoring-a-grammar-into-ll1, 2012. Online; accessed 11 November 2017.

[45] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[46] Lynette van Zijl. CS711: Implementation and Application of Automata. http://www.cs.sun.ac.za/rw711/lectures/lec4/l4.pdf, 2017. Online; accessed 11 November 2017.

[47] Sahil Verma and Subhajit Roy. Synergistic debug-repair of heap manipulations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 163–173, New York, NY, USA, 2017. ACM.