CoStar: A Verified ALL(*) Parser

Sam Lasser
samuel.lasser@tufts.edu
Tufts University, USA

Chris Casinghino
ccasinghino@draper.com
Draper, USA

Kathleen Fisher
kfisher@cs.tufts.edu
Tufts University, USA

Cody Roux
croux@draper.com
Draper, USA

Abstract

Parsers are security-critical components of many software systems, and verified parsing therefore has a key role to play in secure software design. However, existing verified parsers for context-free grammars are limited in their expressiveness, termination properties, or performance characteristics. They are only compatible with a restricted class of grammars, they are not guaranteed to terminate on all inputs, or they are not designed to be performant on grammars for real-world programming languages and data formats.

In this work, we present CoStar, a verified parser that addresses these limitations. The parser is implemented with the Coq Proof Assistant and is based on the ALL(*) parsing algorithm. CoStar is sound and complete for all non-left-recursive grammars; it produces a correct parse tree for its input whenever such a tree exists, and it correctly detects ambiguous inputs. CoStar also provides strong termination guarantees; it terminates without error on all inputs when applied to a non-left-recursive grammar. Finally, CoStar achieves linear-time performance on a range of unambiguous grammars for commonly used languages and data formats.

CCS Concepts: • **Software and its engineering → Parsers**; **Formal software verification**.

Keywords: parsing, interactive theorem proving

ACM Reference Format:
Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. 2021. CoStar: A Verified ALL(*) Parser. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453483.3454053

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454053

1 Introduction

Parsers are natural targets for formal verification. While its theoretical foundations are well-understood, parsing is error-prone in practice. Parsers are also important components of high-assurance software; applications often rely on them to consume input from untrusted sources. Parsers are thus relatively easy to specify, hard to implement correctly, and safety-critical: a combination of traits that makes them good candidates for the application of formal methods.

Parser vulnerabilities and their security consequences are well-documented. In recent years, faulty parsers have leaked user data from popular online services [15], allowed attackers to obtain the private data of nearly 150 million people from a major credit agency [13, 27], and enabled remote code execution on networked devices [14, 20, 26, 28]. These examples demonstrate both the safety-criticality of parsing and the challenge of implementing it in a trustworthy way.

This challenge has motivated several research efforts on verified parsing. In terms of verified parsers for context-free grammars (CFGs), the limitations of prior work fall chiefly into three categories. Top-down predictive parsers [7, 21] are limited in their *expressiveness*; they are only compatible with the restricted LL(1) class of grammars. Bottom-up parsers [2, 17] provide limited *termination guarantees*; they do not guarantee termination on all inputs, and thus are not decision procedures for language membership. Finally, parsers for general CFGs [6, 8, 9, 33] are limited in their claims about *performance* on real-world grammars. They are designed to be compatible with highly ambiguous CFGs, and to return multiple parse trees or similar values for their input. These traits are likely to hinder fast and predictable performance on the deterministic grammars that are sufficient for many practical applications.

In this work, we present a verified parser[1] that addresses these limitations. Our parser is based on the ALL(*) parsing algorithm, which forms the core of the popular ANTLR 4 parser generator [30]. ALL(*) is expressive; it is compatible with a broad class of CFGs. ALL(*) is also amenable to formal reasoning about its termination properties. In addition, ALL(*) achieves linear-time performance on grammars for

[1]The term "parser" sometimes refers to a tool that processes a single grammar. Our tool is parametric over a grammar that it interprets at parse time. Throughout, we use the term "parser" for brevity.

many programming languages and data formats [30], and the popularity of ANTLR suggests that ALL(*) is well-suited to a wide range of applications.

This paper makes the following contributions:

- We present CoStar, an ALL(*) parser implemented and verified with the Coq Proof Assistant [5].
- We give proofs of CoStar's soundness, error-free termination, and completeness for all CFGs without left recursion. The parser produces a correct parse tree for its input whenever such a tree exists, and it is a decision procedure for language membership.
- We prove that CoStar correctly detects ambiguity; i.e., it correctly labels parse trees as unique or ambiguous.
- We show that CoStar achieves linear-time performance on unambiguous grammars for real languages and data formats, including grammars that other verified top-down parsers do not handle.

Our work is also a case study on implementing a small-step interpreter with a big-step correctness specification. This design lends itself to an elegant invariant-based style of verification. We prove each big-step property by defining an invariant over the parser's state that entails the property. This approach shifts the verification burden to the task of proving that each parser step preserves the invariant. It also leads to a clean separation of concerns; we are able to isolate the thorny termination proofs from the actual code, producing an implementation that resembles what a programmer might write in a language with unguarded recursion.

The definitions and theorems in this paper have been mechanized in Coq. The development consists of roughly 5,000 lines of specification and 5,000 lines of proof. This development and an accompanying performance evaluation framework are available online as an artifact submitted with this work [22]. These materials are also available via a public GitHub repository [23].

The paper is organized as follows. In §2, we briefly introduce ALL(*) parsing. We outline the CoStar implementation in §3. We present its termination properties in §4 and correctness in §5. In §6, we give performance evaluation results. We discuss related work in §7, and offer conclusions and plans for future work in §8.

2 Overview of ALL(*) Parsing

The ALL(*) algorithm was introduced by Parr et al. [30]. It shares its high-level structure with LL(k) [24] and LL(*) [29]. These algorithms are top-down and predictive; each one conceptually builds a parse tree starting from the root node, and examines the remaining tokens to decide how to replace grammar left-hand sides (i.e., nonterminals) with right-hand sides. What distinguishes ALL(*) from its predecessors is its more powerful prediction mechanism, which has two key components: dynamic grammar analysis for expressiveness, and memoization of prediction steps for efficiency.

At each decision point, ALL(*) calls an adaptivePredict prediction routine that launches multiple subparsers: one per grammar right-hand side for the nonterminal under consideration. The subparsers advance in lockstep, consuming one token at a time, with each subparser dying off when it fails to recognize a token. This process continues until all subparsers fail (there are no viable right-hand sides), one subparser remains (there is a unique viable right-hand side), or the prediction mechanism detects ambiguity in the grammar. In this third case, ALL(*) alerts the user and chooses one of the ambiguous alternatives. By analyzing the grammar dynamically, ALL(*) is able to accept grammars for which computing lookahead information statically is infeasible.

ALL(*) prediction achieves good performance through a cache-based optimization. The adaptivePredict routine caches each analysis step in a deterministic finite automaton (DFA); a call to the routine yields both a prediction and an updated cache. Before adaptivePredict performs a grammar analysis step, it checks whether that step appears as a transition in the DFA, thereby avoiding redundant computations. This optimization makes ALL(*) efficient in practice. Parr et al. [30] prove that ALL(*) can take $O(n^4)$ time to parse n tokens, but they demonstrate that it runs in linear time on many grammars of practical interest.

3 A Verifiable ALL(*) Implementation

In this section, we describe the structure of CoStar, focusing on aspects of the parser's design that make it amenable to reasoning about its termination and correctness. Figure 1 contains key CoStar definitions that we discuss in this section and throughout the paper.

3.1 Top-Level API

The entry point to CoStar is the parse function, which takes a grammar G, a start symbol $S \in \mathcal{N}$, and an input word w. It returns one of the following values:

- A parse tree v with S at the root and w at the leaves. The tree is labeled Unique or Ambig, depending on whether v is the sole S-rooted tree for w.
- A Reject value indicating that $w \notin \mathcal{L}(G)$.
- An Error value indicating that the parser reached an inconsistent state. (We later prove that this result never occurs when G satisfies a well-formedness condition.)

At a high level, the function simply passes an appropriate set of initial arguments to a stack machine—the heart of the CoStar implementation. The underlying stack machine has three main components: (1) a machine state σ; (2) a step function that performs a single atomic update to the state; and (3) a multistep function that repeatedly calls step until the state reaches one of the final values listed above. Figure 2 depicts a trace of the machine's execution on toy input; we use this figure throughout the paper as a source of examples.

In the next two sections, we describe machine states and the step function. The notable feature of multistep is its complex termination proof, so we defer discussion of that function to Section 4.

3.2 Machine States

As the stack machine runs, it maintains several interrelated pieces of information in the machine state σ:

- A **prefix stack** holding grammar symbols that the machine has already matched against a prefix of the tokens. We call these symbols "processed." The prefix stack also stores parse trees that represent a partial derivation for the processed symbols and prefix. In the case of a successful parse, these trees will become subtrees of the parser's final return value.
  At state (σ_6) in Figure 2, the top prefix stack frame contains the trees Leaf(a) and Node(A, Leaf(b)). It also contains the processed symbols a and A, but we have omitted them from the figure for space reasons; one can simply read them off the roots of the trees.
- A **suffix stack** holding grammar symbols to match against the remaining tokens. We call these symbols "unprocessed," and we call the head symbol in the top frame the "top stack symbol."
  At state (σ_1) in Figure 2, the two suffix stack frames contain the unprocessed symbols Ad and S, respectively, and the top stack symbol is A.
- A **cache** that stores the results of previous prediction steps for later reuse (details appear in Section 3.4).
- The remaining sequence of **tokens** to parse.
- A finite set of **visited** nonterminals. This set is used to ensure that the machine terminates even on left-recursive grammars (details appear in Section 4.1).
- A **uniqueness flag** indicating whether the machine has detected that the input word is ambiguous. Figure 2 does not show this component of the state because the input is unambiguous, so the flag remains true throughout the parse.

3.3 Single-Step Parser Operations

The step function examines the current machine state σ and performs one of the following operations, producing a new state σ' (we write $\sigma \rightsquigarrow \sigma'$ to represent such an operation):

- **consume**: σ is in a consume configuration when the top stack symbol is a terminal a. The machine matches a against the terminal of the next token t; if the match succeeds, the machine pops a and t, and stores Leaf(t) on the prefix stack.
  In Figure 2, the $(\sigma_2) \rightsquigarrow (\sigma_3)$ is a consume operation. The machine's top stack terminal a matches the terminal of the next token, so it adds a leaf containing that token to the prefix stack.
- **push**: σ is in a push configuration when the top stack symbol is a nonterminal X. The machine calls ALL(*) prediction function adaptivePredict. If the call succeeds, returning a right-hand side γ for X, the machine pushes a new frame containing γ onto the suffix stack and an empty frame onto the prefix stack.
  In Figure 2, the $(\sigma_0) \rightsquigarrow (\sigma_1)$ transition is a push. The top stack nonterminal is S, and a call to adaptivePredict reveals that Ad is the only viable right-hand side for S, so the machine pushes Ad onto the suffix stack.
- **return**: σ is in a return configuration when the top suffix frame is empty and the frame below contains a head nonterminal X. (We refer to the frame below as the "caller frame" and X as an "open nonterminal.") The machine pops the top prefix frame $[\alpha, f]$ and top suffix frame, creates a new parse tree Node(X, f), and stores it on the prefix stack.
  In Figure 2, the $(\sigma_5) \rightsquigarrow (\sigma_6)$ transition is a return. The top suffix frame is empty, the top prefix frame contains Leaf(b), and the open nonterminal is A (in the figure, open nonterminals appear in bold font), so the machine stores Node(A, Leaf(b)) in the new top frame of the prefix stack.

These operations can fail in a way that indicates an invalid input word—e.g., when the top stack terminal does not match the next token in a consume operation. They can also fail in a way that indicates an inconsistent machine state—e.g., when the caller frame does not contain an open nonterminal in a return operation. We refer to these two types of failures as rejections and errors, respectively.

The step function also detects when σ is in a final configuration: i.e., when there are no more stack symbols to process or tokens to consume, and the prefix stack contains a single frame holding a single tree. This tree is the parse tree for the start symbol and input word. In this case, step simply returns the final tree.

3.4 Prediction Mechanism

The distinguishing feature of ALL(*) is its prediction algorithm, adaptivePredict. The original presentation of the algorithm is heavily imperative and has a subtle termination argument; these features make the problem of implementing it in Coq (let alone verifying it!) a challenging one.

CoStar invokes adaptivePredict when the top stack symbol is a nonterminal X (we call X a "decision nonterminal"). The resulting prediction determines which grammar right-hand side for X the machine pushes onto the suffix stack.

The adaptivePredict algorithm is really a combination of two different prediction strategies: LL and *strong* LL (SLL). Both strategies attempt to choose a right-hand side for X by simulating the parser's behavior in a nondeterministic fashion. The high-level difference is that LL is a slower and precise simulation, while SLL is faster and imprecise. LL

Basic definitions		*CoStar definitions*	
Terminals	$a, b \in \mathcal{T}$	*Prefix Stacks*	$\Phi ::= \bullet \mid [\alpha, f]\Phi$
Nonterminals	$X, Y \in \mathcal{N}$	*Suffix Stacks*	$\Psi ::= \bullet \mid [\beta]\Psi$
Symbols	$s ::= a \mid X$	*Subparsers*	$\theta ::= (\gamma, \Psi)$
Sentential Forms	$\alpha, \beta, \gamma ::= \bullet \mid s\beta$	*DFA States*	$q ::= \bullet \mid \theta, q$
Grammars	$G ::= \bullet \mid X \to \gamma,\ G$	*DFAs*	$\Delta ::= \bullet \mid (q,\ a) \mapsto q, \Delta$
Literals	$l \in string$	*Machine States*	$\sigma \in \Phi \times \Psi \times \Delta \times w \times \mathcal{S}(\mathcal{N}) \times \mathbb{B}$
Tokens	$t ::= (a, l)$	*Errors*	$e ::= \mathsf{InvalidState} \mid \mathsf{LeftRecursive}(X)$
Words	$w ::= \epsilon \mid tw$	*Predictions*	$p ::= \mathsf{Unique_P}(\gamma) \mid \mathsf{Ambig_P}(\gamma) \mid \mathsf{Reject_P} \mid \mathsf{Error_P}(e)$
Trees	$v ::= \mathsf{Leaf}(t) \mid \mathsf{Node}(X, f)$	*Step Results*	$r ::= \mathsf{Accept_S}(v) \mid \mathsf{Reject_S} \mid \mathsf{Error_S}(e) \mid \mathsf{Cont_S}(\sigma)$
Forests	$f ::= \bullet \mid v, f$	*Parse Results*	$R ::= \mathsf{Unique}(v) \mid \mathsf{Ambig}(v) \mid \mathsf{Reject} \mid \mathsf{Error}(e)$

Figure 1. Core definitions and notations used throughout this paper. We write $\mathcal{S}(A)$ to denote finite sets with elements of type A. For inductively defined types that have an empty value $\bullet$, we omit $\bullet$ when representing non-empty values. For example, we write $[\alpha, f]$ instead of $[\alpha, f]\bullet$. We often refer to tokens only by their terminal component and omit their literal component. For example, we write Leaf((Int, "42")) as Leaf(Int) when only the terminal symbol is relevant to the discussion.

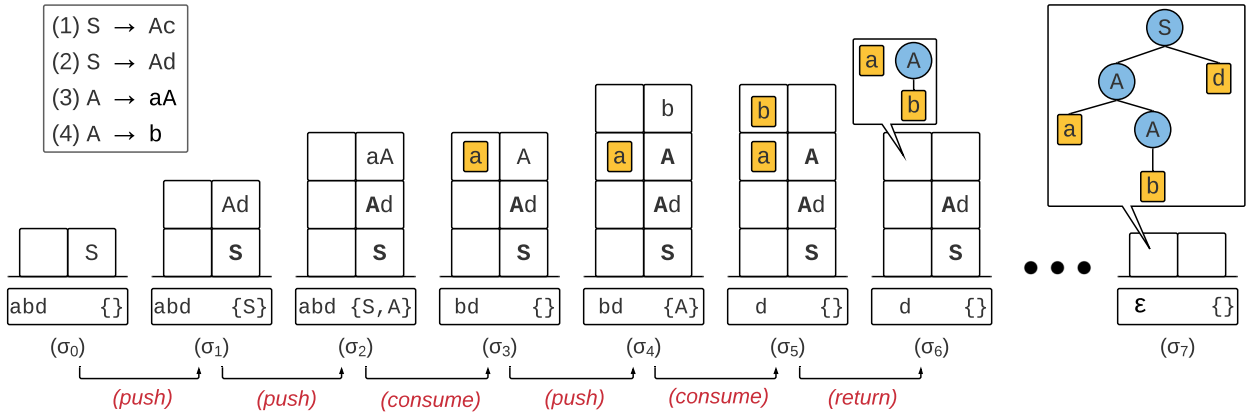

Figure 2. Trace of the CoStar stack machine's execution on a simple grammar and token sequence. Each point in the trace depicts the state of the machine's prefix stack, suffix stack, remaining tokens (represented as terminals to simplify the presentation), and visited nonterminals. For example, at state (σ_0), the prefix stack has a single empty frame, the suffix stack has a single frame containing start symbol S, the token sequence is abd, and the set of visited nonterminals is {} (the empty set). For compactness, we do not show the processed symbols or the DFA cache. We also do not show the uniqueness flag because it remains true throughout the parse (i.e., the derivation is unambiguous).

identifies all and only the viable right-hand sides, while SLL may identify additional right-hand sides that LL rules out; in this sense, SLL is an overapproximation of LL.

The tradeoff is that SLL is faster; it caches simulation steps in a DFA, and uses this cache to avoid recomputing steps. For this reason, adaptivePredict initially tries to make a prediction in SLL mode, and fails over to LL mode only when it detects that the SLL result may be unsound (a case that we describe in more detail below). This failover behavior ensures that the overall prediction algorithm is sound.

CoStar LL and SLL predictions have the same type, but a prediction's meaning can differ based on the mode that produced it. LL predictions have the following interpretations:

- $\mathsf{Unique_P}(\gamma)$: γ is the only right-hand side that may lead to a successful parse.
- $\mathsf{Ambig_P}(\gamma)$: γ leads to a successful parse, as does at least one other right-hand side $\gamma' \neq \gamma$. This result indicates that the input has at least two distinct parse trees—i.e., that the grammar is ambiguous.
- $\mathsf{Reject_P}$: No right-hand side leads to a successful parse.

- $\text{Error}_P(e)$: The prediction mechanism reached an inconsistent internal state. As with the top-level stack machine, we guarantee that this case does not occur for non-left-recursive grammars.

SLL also returns one of the four results listed above, but an $\text{Ambig}_P(\gamma)$ result has a different interpretation in SLL mode. This result indicates that SLL identified multiple right-hand sides $\{\gamma, \gamma', \dots\}$ as viable; because SLL overapproximates LL, it is possible that LL mode would have ruled out γ and marked only γ' as viable. Therefore, prediction must recommence in LL mode to prevent the parser from taking a "wrong turn" by pushing γ onto the stack.

3.5 Is It Really ALL(*)?

ALL(*) as originally presented by Parr et al. [30], and as implemented in ANTLR, has an imperative flavor. Adapting it to a total functional language like Gallina necessarily involves some creative license. ALL(*) is also a complex algorithm, and we make several simplifications to keep the verification tractable. CoStar differs from the original published description of ALL(*) in the following ways.

Original ALL(*) operates on a language representation called an augmented transition network (ATN) [36], while CoStar operates directly on a CFG. This difference is minor, because an ATN is merely a graph representation of a CFG.

Original ALL(*) uses a graph-structured stack (GSS) [34] as a compact representation of subparsers that share some of their state, whereas the current version of CoStar does not. This difference is irrelevant to the extensional behavior or correctness of CoStar, but it means that our tool may be less space-efficient than ANTLR in practice.

Original ALL(*) prediction attempts to detect ambiguity early by checking for "conflicting configurations": subparsers with different right-hand sides that are guaranteed to perform the same steps. In contrast, CoStar only reports ambiguity when subparsers for different right-hand sides advance to the end of the input. As a result, we do not expect CoStar to be performant on ambiguous grammars. In our view, this limitation is not a serious one. Parr et al. [30] assert that "for computer languages, ambiguity is almost always an error" (i.e., a grammar design error); we believe that this statement holds especially true in high-assurance settings that require verified parsing. CoStar's tolerance of ambiguity is mainly for grammar development and debugging purposes; it assists users with the process of testing unfinished grammars, detecting ambiguities, and removing them.

In original ALL(*), when an SLL subparser stack is empty, the subparser must simulate a return to all possible caller frames (this behavior is what makes SLL an overapproximation of LL). In contrast, when a CoStar SLL subparser reaches this case, it simulates a return to the frames in a stable state that are "closure-reachable" (i.e., reachable via push and return operations) from all possible caller frames. These "stable return" frames are computed statically from the grammar, which keeps the SLL termination proof tractable.

Despite these differences, we believe that CoStar is largely faithful to the published description of ALL(*). It is compatible with arbitrary non-left-recursive grammars, it makes predictions by launching subparsers that nondeterministically simulate the parser's behavior, and it caches grammar information to boost performance.

4 Termination

One of the primary challenges of implementing CoStar was proving that the parser terminates on all inputs. Gallina, the functional programming language embedded within Coq, is total—all recursive functions must terminate provably. This restriction is necessary for the soundness of Coq's underlying logic. Coq can automatically confirm termination of many functions that meet certain syntactic criteria. However, several CoStar components have termination arguments that are too subtle for Coq's termination checker to infer, so we must resort to clever means of convincing the checker that the parser always terminates.

In this section, we outline our solution to the problem of writing a provably terminating definition of multistep, the main stack machine loop. The termination argument for multistep is too complex for Coq to infer because it depends on several components of the machine state.

4.1 Handling Left Recursion

Like many top-down parsing algorithms, ALL(*) is incompatible with left-recursive grammars; left recursion has the potential to cause non-termination. In practice, ANTLR is able to avoid most instances of this problem by rewriting the grammar to eliminate common forms of left recursion [30]. We leave the task of verifying such grammar-rewriting steps for future work and adopt a simpler approach: CoStar accepts an arbitrary grammar, but it uses a provably correct scheme for detecting left recursion dynamically that was presented by Lasser et al. [21].

In this scheme, the machine state includes a set of visited nonterminals: the nonterminals that have been opened but not fully processed since the machine last consumed a token. In Figure 2, for example, the initial set of visited nonterminals is empty. The transition from (σ_0) to (σ_1) is a push operation in which S becomes an open nonterminal, so the visited set at (σ_1) is $\{S\}$. After the push operation from (σ_1) to (σ_2), the visited set is $\{S, A\}$ because S and A are both open, and neither has been fully processed. The transition from (σ_2) to (σ_3) is a consume operation, which empties the visited set.

CoStar's dynamic left recursion detection works as follows: before the step function performs a push operation, it checks whether the top stack nonterminal appears in the visited set; a "yes" answer indicates a left-recursive loop in the grammar. In this case, the machine halts and returns

an error value. In Section 5.4, we prove that such an error case never arises when the parser is applied to a non-left-recursive grammar.

4.2 Identifying a Well-Founded Measure

One way to convince Coq that a recursive function terminates is to identify a well-founded measure. Such a measure is a mapping from one or more function parameters to a value that, on each recursive call, decreases with respect to some well-founded "less than" relation.[2]

However, even with dynamic left-recursion detection, there is still no obvious candidate for a multistep decreasing measure. Figure 2 illustrates the way in which the parser tracks visited nonterminals. The number of remaining tokens decreases on some but not all steps, and the visited set grows and shrinks, as does the height of the stacks. Push operations in particular pose a problem; the number of tokens remains constant, the number of visited nonterminals increases by one, the stack heights increase by one, and the total number of stack symbols can increase as well.

The well-founded measure for multistep that we identify is a triple of natural numbers with the following components: (1) the number of remaining tokens; (2) a stackScore value (described in Section 4.3) computed from the visited set and suffix stack; and (3) the height of the suffix stack. The well-founded relation is the standard lexicographic order on triples of natural numbers; we write $<_3$ for this relation.

This measure enables us to write a provably terminating definition of multistep via the following Coq idiom:

1. Define a function meas : $\sigma \to \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ that maps a parser state to the measure described above.
2. Add a proof of the measure's accessibility[3] in $<_3$ as a parameter to multistep (its type becomes dependent). We write $\text{Acc}_{<_3}(\text{meas}(\sigma))$ for this parameter.
3. Prove that each parser step preserves the measure's accessibility (Lemma 4.1). Thanks to the Curry-Howard Isomorphism, we can view this lemma as a function from a proof term $\text{Acc}_{<_3}(\text{meas}(\sigma))$ to a smaller term $\text{Acc}_{<_3}(\text{meas}(\sigma'))$. The accessibility proof thus becomes multistep's structurally decreasing parameter.

Lemma 4.1 (Preservation of $\text{Acc}_{<_3}$)**.** If $\text{Acc}_{<_3}(\text{meas}(\sigma))$ and $\sigma \rightsquigarrow \sigma'$, then $\text{Acc}_{<_3}(\text{meas}(\sigma'))$.

Proof. The Coq Standard Library includes an inductive definition of accessibility, as well as a handy Acc_inv lemma. Given a relation $<$, a proof that element x is accessible in $<$, and a proof that $y < x$, Acc_inv produces a smaller proof term showing that y is accessible in $<$. We need only show that each machine step causes the measure to decrease with respect to $<_3$ (Lemma 4.2). □

[2] A well-founded relation is a relation with no infinite decreasing chains. For example, the $<$ relation on $\mathbb{N}$ is well-founded, because a decreasing chain from any $n \in \mathbb{N}$ must eventually end at 0.

[3] An element x is accessible in $<$ if every element $y < x$ is also accessible.

Lemma 4.2 (Steps Decrease Measure)**.** If $\sigma \rightsquigarrow \sigma'$, then $\text{meas}(\sigma') <_3 \text{meas}(\sigma)$.

Proof. By cases on the shape of σ.

- **consume** case: If σ is in a consume configuration (i.e., if the step is a consume operation), the number of tokens decreases by one.
- **push** case: In a push operation, the number of tokens remains constant, and the stack score decreases (Lemma 4.3).
- **return** case: In a return operation, the number of tokens remains constant, the stack score either (a) decreases or (b) remains constant (Lemma 4.4), and the stack height decreases. Therefore, the overall triple decreases due to a decrease in (a) its second position or (b) its third position.

□

4.3 The stackScore Function

The interesting piece of the measure defined above is the stackScore function. The intuition behind this function is that processing a symbol near the bottom of the suffix stack is "worth more" than processing a symbol near the top in terms of making progress towards termination. When the parser pushes a new frame onto the stack, it must process the pushed symbols γ and their children before it can process the newly open nonterminal X. Therefore, processing γ requires fewer steps than fully processing X.

We make this idea concrete by assigning weights to suffix stack symbols, such that symbols in lower frames receive higher weights. We then compute a numeric score for the entire stack in terms of the weights. Through the use of a carefully chosen exponent value, our formula ensures that the overall stack score decreases on push operations, and decreases or remains constant on return operations. Details of the stackScore formula are as follows:

The frameScore function assigns a score to a single suffix stack frame ψ in terms of the number of unprocessed symbols in ψ, a base b, and an exponent e (we explain how to instantiate b and e below):

$$\text{frameScore}(\psi, b, e) = b^e * (\#\text{ unprocessed symbols in } \psi)$$

The stackScore′ function sums the frameScore values for a list of frames Ψ, incrementing the exponent by one each time it traverses to a lower frame:

$$\text{stackScore}'(\Psi, b, e) = \begin{cases} 0 & \text{if } \Psi = \bullet \\ \text{frameScore}(\psi, b, e) + \\ \text{stackScore}'(\Psi', b, e+1) & \text{if } \Psi = \psi\Psi' \end{cases}$$

Finally, the stackScore function calls stackScore′ with a base of (1 + the max length of a grammar right-hand side), and an initial exponent of $|U \setminus V|$, where U is the universe of

grammar left-hand sides and V is the visited set:

$$\text{stackScore}(G, \Psi, V) = \text{stackScore}'(\ \Psi,\ 1 + \text{maxRhsLen}(G),\ |U \setminus V|\ )$$

These initial base and exponent values ensure that the stack score decreases on a push operation, and that it decreases or stays constant on a return operation:

Lemma 4.3. If the machine pushes from Ψ and V to Ψ' and V', then $\text{stackScore}(G, \Psi', V') < \text{stackScore}(G, \Psi, V)$.

Lemma 4.4. If the machine returns from Ψ and V to Ψ' and V', then $\text{stackScore}(G, \Psi', V') \leq \text{stackScore}(G, \Psi, V)$.

5 Correctness Properties

A novel feature of our correctness analysis is that it encompasses ambiguity detection. We want to ensure that the parser correctly labels trees as unique or ambiguous.[4] Therefore, for both soundness and completeness, we prove two theorems: one for unique derivations, and one for ambiguous derivations. Concretely, we prove that CoStar has the following correctness properties:

1. (*Soundness, unique derivations*): If the parser returns a tree v labeled as Unique, then v is the sole parse tree for the input.
2. (*Soundness, ambiguous derivations*): If the parser returns a tree v labeled as Ambig, then v is one of multiple distinct parse trees for the input.
3. (*Error-free termination*): The parser terminates without returning an error value, whether the input word is valid or invalid.
4. (*Completeness, unique derivations*): Given an input word with a unique parse tree v, the parser returns v and labels it as Unique.
5. (*Completeness, ambiguous derivations*): Given an input word with multiple distinct parse trees, the parser returns one of these trees and labels it as Ambig.

Each correctness property assumes that the parser is applied to a non-left-recursive grammar G. Together, these properties establish that the parser is a decision procedure for membership in $\mathcal{L}(G)$. In the remainder of this section, we discuss these properties and the notable features of their proofs in more detail.

5.1 Correctness Specification

The CoStar correctness specification is a standard grammatical derivation relation over a grammar symbol s, a word w, and a tree v (Figure 3). The relation has the judgment form $s \xrightarrow{v} w$ ("Symbol s derives word w, producing tree v"). This symbol derivation relation is mutually inductive with a second relation (also in Figure 3) for sentential forms—i.e., grammar right-hand sides. This second relation has the judgment form $\gamma \xrightarrow{f} w$ ("Symbols γ derive word w, producing forest f"). Parser soundness and completeness are defined in terms of these two relations.

$$\textsc{DerTerminal}\ \frac{}{a \xrightarrow{\text{Leaf}(a,l)} (a, l)} \qquad \textsc{DerNonterminal}\ \frac{X \to \gamma \in G \qquad \gamma \xrightarrow{f} w}{X \xrightarrow{\text{Node}(X,f)} w}$$

$$\textsc{DerNil}\ \frac{}{\bullet \xrightarrow{\bullet} \epsilon} \qquad \textsc{DerCons}\ \frac{s \xrightarrow{v} w_1 \qquad \beta \xrightarrow{f} w_2}{s\beta \xrightarrow{v,f} w_1 w_2}$$

Figure 3. Derivation relations for symbols and sentential forms with respect to a grammar G.

Some of our proofs rely on two-place variants of these relations that leave the tree or forest unspecified when it is unknown or irrelevant. For example, we write $s \to w$ to mean, "Symbol s recognizes word w."

5.2 Soundness for Unique Derivations

Theorem 5.1 (Soundness, unique derivations)**.** If parse applied to non-left-recursive grammar G, nonterminal S, and word w returns a tree v labeled as Unique, then v is the sole correct parse tree rooted at S for w.

The proof of this theorem relies on a more general lemma about the parser's underlying multistep function. This lemma exemplifies the invariant-based style of verification that we use throughout the CoStar development. Its salient feature is its reliance on two invariants of the parser state (described below). The proof is by induction on the well-founded measure for multistep (Section 4.2). In the base case, the invariants together prove the soundness of multistep directly, and in the inductive case, we need only prove that the step function preserves the invariants. In Sections 5.2.1 and 5.2.2, we describe the two invariants in more detail.

5.2.1 Stack Well-Formedness. The StacksWf_I invariant (Figure 4) captures two well-formedness properties of the parser's prefix and suffix stacks:

- The bottom frames hold only the start symbol. It appears in the bottom suffix frame of initial and intermediate machine states, and in the prefix frame of the machine's final state.
- A pair of upper frames (i.e., a prefix frame and suffix frame at the same level in their respective stacks) hold a complete grammar right-hand side for the open nonterminal in the caller suffix frame.

Lemma 5.2 (Preservation of Stack Well-Formedness)**.** If StacksWf_I(σ) and $\sigma \rightsquigarrow \sigma'$, then StacksWf_I($\sigma'$).

[4] Strictly speaking, a tree cannot be ambiguous—rather, a word is ambiguous when it has more than one parse tree. For brevity, we sometimes refer to a tree for an ambiguous word as an "ambiguous tree."

$\boxed{\Phi \approx_G \Psi}$

$$\textsc{WfInit}\ \frac{}{[\bullet, \bullet] \approx_G [S]} \qquad \textsc{WfFinal}\ \frac{}{[S, v] \approx_G [\bullet]}$$

$$\textsc{WfUpper}\ \frac{[\alpha_1, f_1]\Phi \approx_G [X\beta_1]\Psi \qquad X \to \alpha_2\beta_2 \in G}{[\alpha_2, f_2][\alpha_1, f_1]\Phi \approx_G [\beta_2][X\beta_1]\Psi}$$

$\boxed{\textsc{StacksWf\_I}(\sigma)}$ $$\frac{\Phi \approx_G \Psi}{\textsc{StacksWf\_I}(\Phi, \Psi, \_, \_, \_, \_)}$$

Figure 4. A well-formedness invariant for the prefix stack and suffix stack components of a machine state σ.

Proof. By cases on the shape of the initial machine state σ. The case where σ is in a push configuration is the most involved. Push operations involve calls to the prediction mechanism, so we need a lemma stating that if a call to adaptivePredict for nonterminal X returns some list of symbols γ, then $X \to \gamma \in G$. There are two ways for adaptivePredict to return γ—it can mark the right-hand side as a unique alternative or an ambiguous one—so we need to show that γ is a right-hand side for X in either case. □

5.2.2 Invariant for Unique Partial Derivations.

The UniqueDer_I invariant (Figure 5) says that when the machine state's unique flag is true, the processed stack symbols and trees in each prefix frame comprise a unique partial derivation for a prefix w_1 of the initial parser input w.

Its rules have the following meanings:

- The UniqueBot rule says that the processed symbols α in the bottom prefix frame derive a prefix w_1 of w. It also says that there is no other way to partition w into a different prefix and suffix $w_1'w_2'$ such that α derives w_1' and the unprocessed symbols β derive w_2'.
- The UniqueUpper rule captures the same properties as UniqueBot for upper frames. It also says that "all stack pushes are unique"—i.e., whenever X is an open nonterminal and the prefix and suffix frames above it contain α_2 and β_2 (respectively), $\alpha_2\beta_2$ is the only right-hand side for X that might lead to a successful parse. (We do not know whether $\alpha_2\beta_2$ will succeed—only that no other right-hand side will.)

UniqueDer_I(σ, w) lifts this invariant to machine states.

Lemma 5.3 (Preservation of Unique Partial Derivations)**.** If UniqueDer_I(σ, w) and $\sigma \rightsquigarrow \sigma'$, then UniqueDer_I$(\sigma', w)$.

Proof. By cases on the shape of σ, assuming StacksWf_I(σ). (The well-formedness invariant plays a supporting role in many of these proofs; we will omit it when it is irrelevant to the discussion.) There are two interesting cases:

- **return**: A return operation involves moving the forest f in the top prefix frame to the frame below. There, it becomes the subtrees of the caller nonterminal X that is reduced during the return (for an example, see the transition between states (σ_5) and (σ_6) in Figure 2). To ensure that the partial tree remains unique after this operation, the invariant needs to "remember" that f was produced by a unique viable right-hand side γ for X—in other words, when γ was pushed onto the stack earlier in the machine's execution, no other right-hand side for X would have succeeded. The "unique pushes" premise in the UniqueUpper rule records exactly this information.
- **push**: Here, we must show that a push operation preserves the "unique pushes" property that we relied on in the previous case. We accomplish this task with a key fact about the ALL(*) prediction mechanism (presented below as Lemma 5.4): if adaptivePredict returns a right-hand side γ labeled as Unique for nonterminal X, then γ is the sole right-hand side for X that may lead to a successful derivation.

□

Lemma 5.4. Assume that adaptivePredict returns the prediction Unique$_P(\gamma)$ for nonterminal X. Assume also that some right-hand side γ', together with the unprocessed stack symbols, recognizes the remaining tokens. Then $\gamma' = \gamma$.

Proof. If adaptivePredict returns Unique$_P(\gamma)$, then either (1) SLL prediction found a single viable alternative, or (2) SLL prediction failed over to LL prediction, which found a single viable alternative. In case (1), we show that SLL prediction is an overapproximation of LL prediction, and that LL prediction therefore would have reached the same decision. We can now use Lemma 5.5 (below) about the correctness of LL prediction. In case (2), we can use Lemma 5.5 directly. □

Lemma 5.5. Assume (1) LL prediction returns Unique$_P(\gamma)$ for nonterminal X and token sequence w, and (2) some right-hand side γ', together with the unprocessed stack symbols, recognizes w. Then $\gamma' = \gamma$.

Proof. LL subparsers advance in lockstep until all remaining subparsers carry the same prediction. Therefore, by assumption (1), w can be split into a prefix w_1 and suffix w_2 such that all subparsers that advance through w_1 carry the prediction γ. We call this set of subparsers Θ.

From assumption (2), there exists an initial subparser θ' carrying prediction γ' that is capable of advancing to the end of $w = w_1w_2$. This subparser must advance through prefix w_1. Therefore, $\theta' \in \Theta$, and $\gamma' = \gamma$.

The interesting aspect of this proof is that LL prediction does not advance to the end of the token sequence by default. For efficiency, it examines only as much of the sequence as is necessary to reach a decision—in this case, the minimal prefix is w_1. Therefore, to formalize the notion that θ' is "capable

$$\boxed{\langle \Phi, \Psi \rangle \rightharpoonup_{\mathrm{U}} w \mid w}$$

UniqueBot

$$\frac{\alpha \xrightarrow{f} w_1 \qquad (\forall w_1'\ w_2'\ f'.\ w_1'w_2' = w_1w_2 \wedge \alpha \xrightarrow{f'} w_1' \wedge \beta \to w_2' \implies w_1' = w_1 \wedge w_2' = w_2 \wedge f' = f)}{\langle [\alpha, f], [\beta] \rangle \rightharpoonup_{\mathrm{U}} w_1 \mid w_2}$$

UniqueUpper

$$\frac{\begin{array}{c}\alpha_2 \xrightarrow{f_2} w_2 \\ (\forall w_2'\ w_3'\ f_2'.\ w_2'w_3' = w_2w_3 \wedge \alpha_2 \xrightarrow{f_2'} w_2' \wedge \beta_2\beta_1 \neq \mathsf{unproc}(\Psi) \to w_3' \implies w_2' = w_2 \wedge w_3' = w_3 \wedge f_2' = f_2) \\ (\forall \gamma.\ X \to \gamma \in G \wedge \gamma\beta_1 \neq \mathsf{unproc}(\Psi) \to w_2w_3 \implies \gamma = \alpha_2\beta_2) \\ \langle [\alpha_1, f_1]\Phi, [X\beta_1]\Psi \rangle \rightharpoonup_{\mathrm{U}} w_1 \mid w_2w_3\end{array}}{\langle [\alpha_2, f_2][\alpha_1, f_1]\Phi, [\beta_2][X\beta_1]\Psi \rangle \rightharpoonup_{\mathrm{U}} w_1w_2 \mid w_3}$$

$$\boxed{\textsc{UniqueDer\_I}(\sigma, w)} \qquad \frac{}{\textsc{UniqueDer\_I}((\_, \_, \_, \_, \_, \mathsf{false}), w)} \qquad \frac{\langle \Phi, \Psi \rangle \rightharpoonup_{\mathrm{U}} w_1 \mid w_2}{\textsc{UniqueDer\_I}((\Phi, \Psi, \_, w_2, \_, \mathsf{true}), w_1w_2)}$$

Figure 5. The UniqueDer_I invariant states that when a machine state's unique flag is true, the prefix stack holds a unique partial parse tree for the tokens that have been consumed so far. A judgment $\langle \Phi, \Psi \rangle \rightharpoonup_{\mathrm{U}} w_1 \mid w_2$ can be read, "Φ and Ψ hold a unique partial derivation for prefix w_1 of input word $w = w_1w_2$." The unproc() function extracts the unprocessed symbols from a suffix stack and flattens them into a list.

of advancing to the end" of the sequence, we need to reason counterfactually—i.e., specify the subparser's behavior if it were allowed to advance past the point in the token sequence where prediction halts. To perform this kind of reasoning, we define a relational specification for the operations that a subparser performs, and use it to prove that θ' is capable of consuming the entire input through a sequence of such operations, even though the actual computation halts after examining w_1. □

5.3 Soundness for Ambiguous Derivations

Theorem 5.6 (Soundness, ambiguous derivations)**.** If parse applied to non-left-recursive grammar G, nonterminal S, and word w returns a tree v labeled as Ambig, then v is a correct parse tree rooted at S for w, and there exists another correct tree $v' \neq v$.

Like Theorem 5.1, this theorem relies on a more general lemma about multistep, with a proof by induction on the multistep well-founded measure. The lemma is based on a machine state invariant called AmbigDer_I. This inductive invariant says that when the unique flag is false, the prefix and suffix stacks hold an ambiguous partial derivation. We only give the intuition behind each invariant rule here; the precise definitions appear in the formal development.

A partial derivation can be ambiguous in three ways:

- (AmbigPush): The top-level pair of stack frames contains a right-hand side γ for caller nonterminal X when another right-hand side $\gamma' \neq \gamma$ also would have led to a successful parse. The alternative γ' is in effect the "road not taken."
- (AmbigForest): The processed symbols in the top prefix frame produce multiple forests for (possibly different) portions of the original input word; only one of these forests appears in the frame.
- (AmbigTail): The ambiguity appears in a lower pair of stack frames.

Figure 6 illustrates the way in which the stack machine preserves this invariant. In machine state (σ_0), the invariant holds because the state's unique flag is true. The transition from (σ_0) to (σ_1) is a push in which adaptivePredict detects ambiguity. The machine pushes right-hand side X onto the stack, but pushing Y would have worked just as well; therefore, the AmbigPush rule applies. The next push from (σ_1) to (σ_2) is unambiguous, but the overall derivation is still ambiguous; the AmbigTail rule shifts the focus to the stack tails, where the AmbigPush rule still holds. In the final state (σ_4), the bottom prefix frame holds a correct parse tree for the input word, but a different tree—one with Y as the middle node instead of X—would also be correct. The AmbigForest case of the invariant captures this property.

Lemma 5.7 (Preservation of Ambiguous Partial Derivations)**.** If AmbigDer_I(σ, w) and $\sigma \rightsquigarrow \sigma'$, then AmbigDer_I($\sigma', w$).

Proof. By cases on the shape of σ. An interesting difference between the proof of Lemma 5.3 and this one is that the

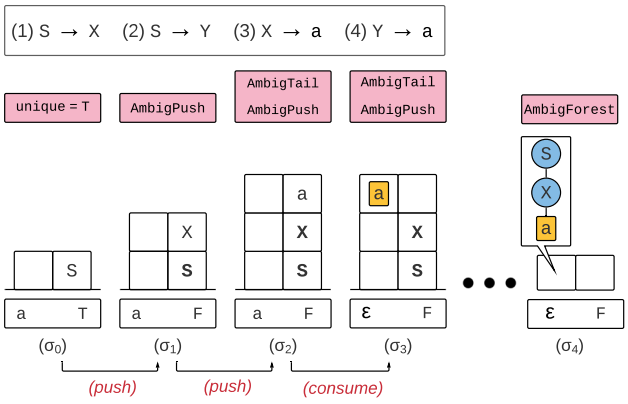

Figure 6. Machine execution trace on a simple ambiguous grammar and input word. The cases of the AmbigDer_I invariant that hold for each machine state appear above the state in pink. We replace the visited sets from Figure 2 with unique flags (T/F) because they are more relevant here.

UniqueDer_I consequent is true of the initial machine state (the empty partial derivation is unique), but AmbigDer_I is only "trivially" true at the start of parsing, because its "unique = false" antecedent is false. The unique flag is initially true and only becomes false if the prediction mechanism detects ambiguity midway through the parser's execution.

Therefore, the tricky case in the proof of Lemma 5.7 is the "ambiguous push" case, where we have to prove that the partial derivation is ambiguous at the point when the flag switches from true to false, even though no prior ambiguity has been detected. We handle this case with a lemma about adaptivePredict: if the function returns $\text{Ambig}_P(\gamma)$for nonterminal X, then both γ and a second right-hand side $\gamma' \neq \gamma$ for X lead to a successful derivation. This situation corresponds to the AmbigPush case of the invariant. □

5.4 Error-Free Termination

Theorem 5.8. Given a non-left-recursive grammar, the parser never returns an Error value.

This property is used in the completeness proofs to show that the parser never returns an error for valid words. Along with the soundness theorems, it also guarantees that the parser returns a Reject value for invalid words, rather than an error that does not clearly indicate acceptance or rejection.

Parser errors fall into the following categories:

1. InvalidState errors indicate a malformed machine state—e.g., the prefix and suffix stacks are different heights.
2. LeftRecursive(X) errors indicate that the parser has detected a left-recursive grammar nonterminal X.
3. Errors can arise within the prediction mechanism. Since prediction simulates parsing, prediction errors fall into the two categories above: a subparser can reach an invalid state (category 1) or detect left recursion dynamically (category 2). This overlap enables us to streamline the verification work by factoring out some lemmata and using them to rule out both top-level parsing errors and prediction errors.

The StacksWf_I invariant is sufficient to rule out InvalidState parser errors. The parser returns such an error when it detects that the prefix and suffix stacks are different heights, that the bottom prefix frame contains more than one tree, or that there is no caller nonterminal to return to during a return operation. The well-formedness property ensures that these cases never occur. A similar invariant rules out InvalidState errors in the prediction mechanism. This latter invariant describes only the well-formedness of a suffix stack, since subparsers do not build parse trees and therefore do not use prefix stacks.

The proof that left recursion errors do not arise is more interesting. Below, we briefly present this proof as it applies to the stack machine. The corresponding proof for the prediction mechanism involves similar reasoning, and it relies on the same key invariant (thus avoiding redundancy!).

5.4.1 Ruling Out Left Recursion Errors.

Lemma 5.9. Given a non-left-recursive grammar G, the parser never raises an error that identifies nonterminal X as left-recursive in G.

Proof. We adapt an approach that Lasser et al. [21] use to rule out similar errors in an LL(1) parser. The approach involves proving that the dynamic mechanism for detecting left recursion is sound—i.e., if the parser returns LeftRecursive(X), then nonterminal X really is left-recursive in the grammar (Lemma 5.10 below). With a little first-order reasoning, it is then easy to show that a non-left-recursive grammar and a sound detection mechanism together entail the absence of left recursion errors. □

5.4.2 Soundness of Left-Recursion Detection.

Lemma 5.10. If the parser applied to G returns LeftRecursive(X), then nonterminal X is left-recursive in G.

Proof. The proof centers on an invariant that relates the shape of the parser's suffix stack to the shape of the grammar, via the visited set V. It says, roughly, that every visited nonterminal $X \in V$ is an open nonterminal in a caller suffix frame, and that there is a nullable path from X to the top stack symbol.[5] In Figure 2 at state (σ_2), for example, $V = \{S, A\}$; both S and A are open nonterminals in lower suffix frames, and there is a nullable path S → A → a because

[5] A nullable path is a path between two positions in the grammar that does not include any terminals. It corresponds to a sequence of parser operations that does not consume any tokens. Lasser et al. [21] formally define left recursion as a special case of a nullable path.

the sequence of transitions $(\sigma_0) \to (\sigma_1) \to (\sigma_2)$ did not consume any tokens. The parser returns LeftRecursive(X) when the top stack symbol is a visited nonterminal X—per the invariant, when there is a nullable path from X to X, which is exactly the definition of left recursion! □

5.5 Completeness

Like the two soundness theorems, the two CoStar completeness theorems cover unique and ambiguous derivations:

Theorem 5.11 (Completeness, unique derivations)**.** If word w has a unique parse tree v rooted at nonterminal S, then the parser applied to S and w produces v and labels it as Unique.

Theorem 5.12 (Completeness, ambiguous derivations)**.** If word w is ambiguous (it has two correct S-rooted parse trees v and v', where $v \neq v'$), then the parser applied to S and w returns a correct tree v'' and labels it as Ambig.

Theorem 5.12 reflects the ALL(*) policy towards ambiguity, which is to treat it as a likely grammar error. In practice, designers of grammars for programming languages and data formats do not want the parser to return every possible tree for an ambiguous input, which may be expensive. Instead, they expect the parser to (1) return *a* correct tree, and (2) notify them that the input is ambiguous. These are exactly the guarantees that CoStar provides for ambiguous inputs.

To prove Theorems 5.11 and 5.12, it suffices to prove the weaker statement that when a correct parse tree v exists, the parser produces *any* tree v' (Lemma 5.13 below). We can then marshal our parser soundness theorems and a little first-order reasoning to show that v' is a correct tree, and that the parser correctly labels it as Unique (Theorem 5.11) or Ambig (Theorem 5.12). In the remainder of this section, we focus on this more general version of completeness.

Lemma 5.13 (Completeness, general)**.** If a correct S-rooted tree v exists for word w, then the parser returns a tree v'.

The proof of Lemma 5.13 follows this informal reasoning: if the parser does not (1) return errors or (2) reject valid input words, then it is complete. Theorem 5.8, our error-free termination theorem, rules out case (1), so the new challenge is to rule out case (2). As before, we proceed by introducing an invariant over the machine state. The UnprocRecognize_I invariant (Figure 7) says that the unprocessed symbols on the suffix stack recognize the remaining token sequence. This invariant holds at the start of parsing: if a correct S-rooted tree exists for the input word, then S recognizes the input word. It is also easy to show that when this invariant holds, the parser never rejects its input as invalid. The tricky part is showing that parser steps preserve the invariant:

Lemma 5.14. If UnprocRecognize_I(σ) and $\sigma \rightsquigarrow \sigma'$, then UnprocRecognize_I(σ').

$$\boxed{\textsc{UnprocRecognize\_I}(\sigma)}$$

$$\frac{\text{unproc}(\Psi) \to w}{\textsc{UnprocRecognize\_I}(\_, \Psi, \_, w, \_, \_)}$$

Figure 7. A machine state invariant for proving completeness: the unprocessed suffix stack symbols recognize the remaining suffix of the input word.

Proof. By cases on the shape of σ. The difficulty stems from the two "push" cases, which correspond to unique and ambiguous predictions, respectively. In these cases, we have to show that the prediction mechanism never causes the parser to take a "wrong turn"; i.e., adaptivePredict never returns a right-hand side for top stack symbol X that causes the parser to reject the input when some other right-hand side would have led to a successful parse.

In the "unique prediction" case, adaptivePredict returns $\text{Unique}_P(\gamma)$, and the invariant over the pre-push machine state tells us that some right-hand side γ' for X leads to a successful parse. How do we show that adaptivePredict has not led the parser astray by selecting γ? We reuse Lemma 5.4, which says that if adaptivePredict returns $\text{Unique}_P(\gamma)$, then γ is the sole right-hand side for X that may lead to a successful parse. Therefore, the adaptivePredict result γ must be equal to γ'.

In the "ambiguous prediction" case, adaptivePredict returns $\text{Ambig}_P(\gamma)$, which means that γ (as well as other right-hand sides for X) caused a subparser θ to reach the end of the remaining token sequence. This case is tricky because it involves "rewinding" θ, using a fact about its final state (that it recognized the entire sequence) to prove a fact about its initial state (that it is *capable* of recognizing the entire sequence at the start of prediction). We perform this "backward" reasoning by proving an equivalence between the prediction algorithm and a relational specification. The specification makes it easier to run prediction steps in reverse; it supports reasoning about the state of a subparser before a prediction step based on its state after the step. □

6 Performance Evaluation

To evaluate CoStar's performance, we extracted the tool to OCaml source code and measured its execution time on four benchmarks, with the goal of observing its asymptotic behavior on grammars for popular programming languages and data formats. We then measured the execution time of ANTLR parsers on the same benchmarks to assess CoStar's performance relative to ANTLR.

6.1 CoStar Benchmarks

Each CoStar benchmark involved providing the parser with a grammar for a real-world programming language or data

Benchmark	Grammar Size			Data Set Size	
	$\|\mathcal{T}\|$	$\|\mathcal{N}\|$	$\|\mathcal{P}\|$	# files	MB
JSON	11	7	17	25	21
XML	16	22	40	1260	192
DOT	20	44	73	48	19
Python 3	89	287	521	169	4

Figure 8. Measures of grammar size and data set size for the four CoStar benchmarks. Counts of terminals $\mathcal{T}$, nonterminals $\mathcal{N}$, and productions $\mathcal{P}$ are taken from the desugared BNF grammars.

format, and measuring the parser's execution time on a representative data set for that language or format.

ANTLR grammars exist for a wide variety of languages. To facilitate testing with these existing grammars, we built a tool that converts a grammar in ANTLR's input format to the OCaml data structure that CoStar takes as input. ANTLR grammars can include EBNF operators (e.g., the Kleene star), whereas CoStar is parameterized by a BNF grammar. Therefore, the grammar conversion tool desugars EBNF elements into equivalent BNF structures, generating fresh nonterminals and adding new productions to the grammar as necessary. These transformations produce a grammar that accepts the same language as the original one, but we do not prove this fact. CoStar takes pre-tokenized input, so we also built a tool that uses an ANTLR grammar to tokenize a data file.

We ran benchmarks based on four languages: JSON, XML, DOT, and Python 3. Figure 8 provides statistics on the sizes of the grammars and data sets. We reused the JSON, XML, and DOT grammars from the original ANTLR performance evaluation [30] and took the Python 3 grammar from a central repository for ANTLR grammars [18]. We also reused DOT data from the ANTLR evaluation[6] and JSON data from an LL(1) parser performance evaluation [21]. The XML data is a subset of the Open American National Corpus [32] (a collection of annotated linguistic data), and the Python 3 data is a portion of the Python 3.6.12 standard library.

To the best of our knowledge, the grammars contain no ambiguity or left recursion. CoStar does not attempt to check either grammar property statically, but the tool returns a parse tree labeled as Unique for all files in the benchmark data sets, so we can be reasonably confident that the grammars are unambiguous and left recursion-free.

At least some of the grammars take advantage of ALL(*) prediction's expressive power; consider the following rule from our XML grammar (in ANTLR's EBNF notation):

```
elt : '<' Name attribute* '>' content '<' '/' Name '>'
    | '<' Name attribute* '/>' ;
```

[6]We did not reuse the JSON or XML data sets from the ANTLR evaluation because each contains a small number of files (four and one, respectively). Testing CoStar on many files of varying size gave us a clearer picture of the tool's asymptotic behavior.

Because of this rule, the grammar is not LL(k) for any k; prediction must advance through an arbitrary number of XML attributes before determining which of the two productions matches the remaining input.

We ran the benchmarks on a laptop with 4 2.5 GHz cores, 7 GB of RAM, and the Ubuntu 16.04 OS. We used OCaml compiler version 4.11.1+flambda with optimization level -O3.

Plots of our benchmark results appear in Figure 9. Each point represents the parse time for a single input file, averaged over five trials. Note that the results strongly suggest linear performance, which is consistent with reported empirical results for ANTLR [30]. We borrowed a technique from that earlier work to bolster our claim of linear performance on the benchmarks: each plot includes a least-squares regression line and a LOWESS curve [4]. LOWESS is a method for approximating a scatter plot with a smooth curve that is not constrained to be linear. The close correspondence between LOWESS curves and regression lines in our results indicates a linear relationship between input size and parse time.

Differences in CoStar's performance across benchmarks are probably a function of grammar size. CoStar makes heavy use of finite maps and sets that contain grammar symbols (and data structures composed of grammar symbols). We used Coq Standard Library implementations of these collections that are based on AVL trees, for which insertions, deletions, and lookups require $O(\log n)$ comparisons with respect to the number of possible map keys or set elements. For larger grammars, the space of keys (or elements) is larger; as Figure 8 shows, our largest evaluation grammar is Python, so the fact that our Python benchmark is the slowest in terms of tokens processed per second does not come as a surprise.

Empirical evidence supports this explanation of performance differences across benchmarks. Profiling CoStar on the Python benchmark reveals that the function compareNT (which compares nonterminals) accounts for roughly 17% of execution time, and the five most expensive functions are all comparisons, together accounting for nearly 50% of execution time. In contrast, profiling CoStar on the JSON benchmark shows that compareNT accounts for only 5% of execution time, and garbage collection, not comparison, dominates performance.

6.2 Performance Comparison with ANTLR

To assess CoStar's performance relative to ANTLR, we created ANTLR parsers for our four benchmark languages and measured their execution time on the data sets from the CoStar evaluation.

We ran the ANTLR parser benchmarks on the test machine described in Section 6.1. Each benchmark consisted of three complete passes over the data set. To allow for JIT warm-up, we discarded the results of the first two passes and recorded the results of the third pass. Each pass involved running five parser trials per data point. In each trial, we instantiated an ANTLR parser and measured its execution time by calling

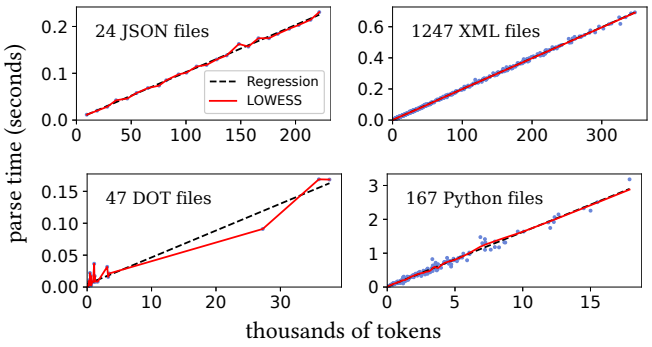

Figure 9. Input size vs. CoStar average parse time on four benchmarks. Unconstrained LOWESS curves coincide with regression lines, indicating linear-time performance. Curves were computed on the smallest 99% of files in each benchmark for scale, with a LOWESS f-hyperparameter value of 0.1. Values of f close to 0 produce a more jagged curve; values close to 1 produce a smoother curve.

System.nanoTime() immediately before and after invoking the parser on the current data file.

Before each ANTLR parser trial, we used an ANTLR lexer to pre-tokenize the input so that we could measure ANTLR parsing time separately from lexing time. This step enables a meaningful comparison between ANTLR and CoStar, since the latter tool parses pre-tokenized input.

For each ANTLR parser benchmark described above, we also ran a corresponding benchmark in which we recorded *lexing* times instead of parsing times. These measurements enable us to compare the performance of an "ANTLR lexer, ANTLR parser" pairing to that of an "ANTLR lexer, CoStar parser" pairing. This comparison represents the performance consequences of replacing an unverified parser with CoStar in a lexing/parsing pipeline. Note that CoStar is not directly interoperable with ANTLR lexers at present; we benchmarked ANTLR lexers for each language, and then combined the lexing times with ANTLR and CoStar parsing times to simulate the effect of substituting one parser for another.

Figure 10 shows CoStar's average slowdown relative to ANTLR on each benchmark. There are two bars per benchmark. The first bar shows CoStar's slowdown relative to an ANTLR parser; lexing time is excluded. The second bar shows the slowdown of an "ANTLR lexer, CoStar parser" pairing relative to an "ANTLR lexer, ANTLR parser" pairing. CoStar is roughly 6-11x slower than an ANTLR parser on the JSON, XML, and DOT benchmarks. The Python results require a bit more explanation:

First, the ANTLR Python parser's performance appears to improve slightly as file size increases (see Figure 11, left side), so CoStar's slowdown relative to ANTLR is not consistent across all data points. We believe that ANTLR exhibits this behavior because ALL(*)'s dynamic cache optimization provides a bigger performance boost on large Python files than

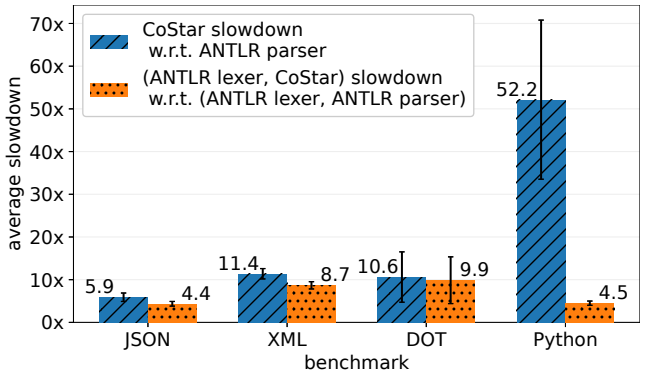

Figure 10. CoStar's average slowdown relative to ANTLR on each benchmark. Striped blue bars show CoStar's average slowdown relative to an ANTLR parser. Dotted orange bars show the average slowdown of an "ANTLR lexer, CoStar parser" pairing relative to an "ANTLR lexer, ANTLR parser" pairing. This latter measure represents the cost of replacing an unverified parser with CoStar in a lexing/parsing pipeline. Error bars show standard deviations.

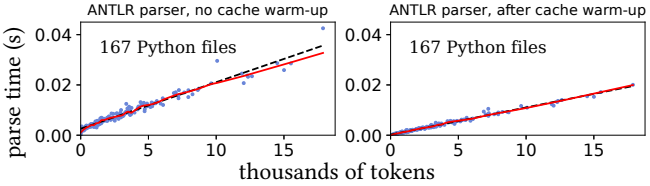

Figure 11. Benchmark results for the ANTLR Python parser. The left plot shows that when each benchmark trial involves a newly instantiated parser with an empty cache, performance improves slightly as file size increases. The right plot shows that when a parser with a pre-warmed cache is used in the benchmark, this slight nonlinear effect disappears.

on small ones.[7] To test this hypothesis, we ran a variation of the ANTLR Python parsing experiment in which we warmed up the parser's cache by parsing many files, and then ran the standard benchmark with the warmed-up parser. This approach causes the slight nonlinearity to disappear (see Figure 11, right side).

Second, the ANTLR Python lexer is slow relative to the ANTLR Python parser, possibly due to Python's complex whitespace and indentation rules. As a result, while CoStar's slowdown relative to the ANTLR Python parser is large, the overall cost of using CoStar in a lexing/parsing pipeline may be more modest.

[7] We do not see similar ANTLR behavior on the JSON, XML, and DOT benchmarks, probably because the grammars for these languages are smaller than the Python grammar (as Figure 8 shows), so cache warm-up occurs even on small files. We do not see similar CoStar behavior on the Python benchmark, either. A likely explanation is that, as Section 3.5 explains, CoStar caches some grammar information statically, so on small files, CoStar might have access to a larger cache than ANTLR.

Note that in these experiments, we ran ANTLR in a configuration that enables as direct a comparison with CoStar as possible–not a configuration that leads to optimal performance. For example, ANTLR is able to generate parser source code that is specialized to a particular grammar, but we chose to run it in "interpreter mode" because CoStar is an interpreter, not a code generator. In addition, an ANTLR parser is able to reuse the cache that it built while parsing previous input in order to improve its performance on new input. However, in each ANTLR parser trial, we instantiated a new parser with an empty cache because CoStar does not currently offer a way to reuse a cache across multiple inputs.

7 Related Work

Several recent works present verified LL(1) parsing techniques. Like ALL(*), LL(1) is a top-down predictive approach to parsing. Lasser et al. [21] use Coq to verify an LL(1) parser generator. The tool is based on the classic technique for constructing an LL(1) parsing table through static analysis of a CFG. Edelmann et al. [7] present a derivative-based LL(1) parsing algorithm, which they verify in Coq and reimplement in a Scala parser combinator framework. Parsing with derivatives [25] is an elegant formalism that is compatible with arbitrary CFGs and has cubic worst-case time complexity [1]. Edelmann et al. show that restricting derivative-based parsing to LL(1) grammars leads to linear-time performance. The main limitation of these techniques is that they are only compatible with LL(1) grammars.

Certified versions of bottom-up parsing algorithms exist as well. Barthwal and Norrish [2] verify SLR parsers with the HOL4 proof assistant, and Jourdan et al. [17] use Coq to perform *a posteriori* validation of LR(1) parsers produced by an unverified generator. Neither development includes a proof of error-free termination on all inputs; the parsers therefore cannot be viewed as verified decision procedures for language membership. One drawback of bottom-up parsing is that producing informative error messages is difficult enough to be a research area in its own right [16, 31]. Bottom-up algorithms are also not well-suited for handling non-context-free extensions to grammars, such as data dependencies [10].

There have been several successful efforts to verify parsers for general CFGs. General parsing algorithms are designed to be compatible with ambiguous grammars, and they typically return all parse trees for a given input. These properties may be undesirable in a setting where parsing is expected to be unambiguous and fast. Ridge [33] presents an elegant technique for constructing a parser for an arbitrary CFG. Using HOL4, he proves that the technique produces a terminating and correct parser, even when the grammar is left-recursive. An implementation that includes an additional memoization component has $O(n^5)$ worst-case time complexity. Firsov and Uustalu [8] describe a verified Agda implementation of the CYK algorithm, which operates on CFGs in Chomsky normal form (CNF). In subsequent work [9], they verify a CNF normalization algorithm. The combined result is a verified parser for arbitrary CFGs. Danielsson [6] presents an Agda parser combinator library that guarantees termination and correctness of parsers built with the combinators, and that accepts many left-recursive parser definitions.

Parsing expression grammars (PEGs) [12] are a language representation sometimes used in place of CFGs to specify parsers. Koprowski and Binsztok [19] present a Coq formal semantics for PEGs and prove the correctness of a PEG interpreter with respect to this semantics. PEG parsers often employ a memoization technique called packrat parsing [11] to achieve linear time and space complexity. Wisnesky et al. [35] and Blaudeau and Shankar [3] use the Ynot and PVS frameworks, respectively, to verify packrat PEG parsers. The main drawback of the PEG formalism is that its "ordered choice" operator hides ambiguities in the language definition, which can lead to counterintuitive parsing behavior.

8 Conclusions and Future Work

We have presented CoStar, a verified parser based on the ALL(*) algorithm. CoStar is sound and complete for non-left-recursive grammars; it produces a correct parse tree for its input if and only if such a tree exists, and it correctly labels the tree as unique or ambiguous. Given a non-left-recursive grammar, the parser terminates without error on all inputs. We have demonstrated empirically that CoStar achieves linear-time performance on unambiguous grammars for real-world languages. Below, we discuss several possible extensions of this work.

The "no left recursion" grammar property that appears as an assumption in our correctness theorems is decidable. We plan to implement and verify a decision procedure that checks a grammar for this property.

We plan to add support for user-defined semantic actions and predicates to CoStar, so that the tool can produce and validate semantic values with a user-defined type. One difficult aspect of this task is that it complicates our notion of ambiguity, because two distinct parse trees for an ambiguous word might map to the same semantic value. Therefore, we would have to update portions of the specification that describe the parser's ambiguity detection features.

Acknowledgments

We are extremely grateful to Terence Parr (the ANTLR guy), who generously and enthusiastically shared his knowledge of ANTLR, ALL(*), and empirical evaluation techniques with us. He also provided us with grammars and data sets from a previous ANTLR performance evaluation. We thank our PLDI shepherd, the anonymous reviewers of this paper, and our Artifact Evaluation reviewers for their careful reading of our work and thoughtful feedback. Sam Lasser's research is supported by a Draper Fellowship.

References

[1] Michael D. Adams, Celeste Hollenbeck, and Matthew Might. 2016. On the Complexity and Performance of Parsing with Derivatives. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*. ACM, 224–236. https://doi.org/10.1145/2908080.2908128

[2] Aditi Barthwal and Michael Norrish. 2009. Verified, Executable Parsing. In *Proceedings of the 18th European Symposium on Programming (ESOP 2009)*. Springer-Verlag, 160–174. https://doi.org/10.1007/978-3-642-00590-9_12

[3] Clement Blaudeau and Natarajan Shankar. 2020. A Verified Packrat Parser Interpreter for Parsing Expression Grammars. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, 3–17. https://doi.org/10.1145/3372885.3373836

[4] William S Cleveland. 1979. Robust Locally Weighted Regression and Smoothing Scatterplots. *Journal of the American Statistical Association* 74, 368 (1979), 829–836.

[5] The Coq Development Team. 2020. *The Coq Proof Assistant, version 8.11.0*. https://doi.org/10.5281/zenodo.3744225

[6] Nils Anders Danielsson. 2010. Total Parser Combinators. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010)*. ACM, 285–296. https://doi.org/10.1145/1863543.1863585

[7] Romain Edelmann, Jad Hamza, and Viktor Kunčak. 2020. Zippy LL(1) Parsing with Derivatives. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. ACM, 1036–1051. https://doi.org/10.1145/3385412.3385992

[8] Denis Firsov and Tarmo Uustalu. 2014. Certified CYK Parsing of Context-Free Languages. *Journal of Logical and Algebraic Methods in Programming* 83, 5-6 (2014), 459–468. https://doi.org/10.1016/j.jlamp.2014.09.002

[9] Denis Firsov and Tarmo Uustalu. 2015. Certified Normalization of Context-Free Grammars. In *Proceedings of the 2015 Conference on Certified Programs and Proofs (CPP 2015)*. ACM, 167–174. https://doi.org/10.1145/2676724.2693177

[10] Kathleen Fisher and Robert Gruber. 2005. PADS: A Domain-Specific Language for Processing Ad Hoc Data. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*. ACM, 295–304. https://doi.org/10.1145/1065010.1065046

[11] Bryan Ford. 2002. Packrat Parsing: Simple, Powerful, Lazy, Linear Time (Functional Pearl). In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP 2002)*. ACM, 36–47. https://doi.org/10.1145/581478.581483

[12] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004)*. ACM, 111–122. https://doi.org/10.1145/964001.964011

[13] Dan Goodin. 2017. Failure to patch two-month-old bug led to massive Equifax breach. *Ars Technica* (13 Sept. 2017). https://arstechnica.com/information-technology/2017/09/massive-equifax-breach-caused-by-failure-to-patch-two-month-old-bug

[14] Dan Goodin. 2020. Windows has a new wormable vulnerability, and there's no patch in sight. *Ars Technica* (11 March 2020). https://arstechnica.com/information-technology/2020/03/windows-has-a-new-wormable-vulnerability-and-theres-no-patch-in-sight

[15] Google Project Zero 2017. Cloudflare: Cloudflare Reverse Proxies are Dumping Uninitialized Memory. https://bugs.chromium.org/p/project-zero/issues/detail?id=1139.

[16] Clinton L. Jeffery. 2003. Generating LR Syntax Error Messages from Examples. *ACM Transactions on Programming Languages and Systems* 25, 5 (Sept. 2003), 631–640. https://doi.org/10.1145/937563.937566

[17] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) Parsers. In *Proceedings of the 21st European Symposium on Programming (ESOP 2012)*. Springer, 397–416. https://doi.org/10.1007/978-3-642-28869-2_20

[18] Bart Kiers. 2014. ANTLR4 Grammar for Python 3. Retrieved from https://github.com/antlr/grammars-v4.

[19] Adam Koprowski and Henri Binsztok. 2010. TRX: A Formally Verified Parser Interpreter. In *Proceedings of the 19th European Symposium on Programming (ESOP 2010)*. Springer, 345–365. https://doi.org/10.1007/978-3-642-11957-6_19

[20] Mohit Kumar. 2020. Critical PPP Daemon Flaw Opens Most Linux Systems to Remote Hackers. *The Hacker News* (2020). https://thehackernews.com/2020/03/ppp-daemon-vulnerability.html

[21] Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. 2019. A Verified LL(1) Parser Generator. In *Proceedings of the 10th International Conference on Interactive Theorem Proving (ITP 2019)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 24:1–24:18. https://doi.org/10.4230/LIPIcs.ITP.2019.24

[22] Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. 2021. CoStar parser implementation, correctness proofs, and performance evaluation. https://doi.org/10.5281/zenodo.4681598

[23] Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. 2021. GitHub repository for the CoStar development and evaluation framework. https://github.com/slasser/CoStar

[24] P. M. Lewis and R. E. Stearns. 1968. Syntax-Directed Transduction. *Journal of the ACM* 15, 3 (July 1968), 465–488. https://doi.org/10.1145/321466.321477

[25] Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with Derivatives: A Functional Pearl. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011)*. Association for Computing Machinery, 189–195. https://doi.org/10.1145/2034773.2034801

[26] NIST 2016. CVE-2016-0101. National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2016-0101.

[27] NIST 2017. CVE-2017-5638. National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2017-5638.

[28] NIST 2020. CVE-2020-8597. National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2020-8597.

[29] Terence Parr and Kathleen Fisher. 2011. LL(*): The Foundation of the ANTLR Parser Generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)*. ACM, 425–436. https://doi.org/10.1145/1993498.1993548

[30] Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA 2014)*. ACM, 579–598. https://doi.org/10.1145/2660193.2660202

[31] François Pottier. 2016. Reachability and Error Diagnosis in LR(1) Parsers. In *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 88–98. https://doi.org/10.1145/2892208.2892224

[32] American National Corpus Project. 2010. Open American National Corpus. http://www.anc.org/data/oanc/download/

[33] Tom Ridge. 2011. Simple, Functional, Sound and Complete Parsing for All Context-Free Grammars. In *Proceedings of the First International Conference on Certified Programs and Proofs (CPP 2011)*. Springer, 103–118. https://doi.org/10.1007/978-3-642-25379-9_10

[34] Elizabeth Scott and Adrian Johnstone. 2010. GLL Parsing. *Electronic Notes in Theoretical Computer Science* 253, 7 (2010), 177–189. https://doi.org/10.1016/j.entcs.2010.08.041

[35] Ryan Wisnesky, Gregory Malecha, and Greg Morrisett. 2009. Certified Web Services in Ynot. In *Proceedings of the 5th International Workshop on Automated Specification and Verification of Web Systems (WWV 2009)*. RISC-Linz, 5–19. https://www3.risc.jku.at/conferences/wwv09

[36] W. A. Woods. 1970. Transition Network Grammars for Natural Language Analysis. *Communications of the ACM* 13, 10 (Oct. 1970), 591–606. https://doi.org/10.1145/355598.362773