Formal Verification of Top-Down Parser Interpreters

A dissertation

submitted by

Sam Lasser

In partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Computer Science

TUFTS UNIVERSITY

August 2022

ADVISOR: Kathleen Fisher

Formal Verification of Top-Down Parser Interpreters

Sam Lasser
ADVISOR: Kathleen Fisher

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 (CFGs) are limited in terms of their termination properties, performance characteristics, or expressiveness. Some verified parsers do not guarantee termination on all inputs. Others are not designed to be performant on grammars for commonly used programming languages and data formats. Finally, real-world data formats often have non-context-free specifications that purely CFG-based parsers cannot fully capture.

This dissertation presents Vermillion and CoStar, two verified parser interpreters that address these limitations. Vermillion and CoStar are based on the LL(1) and ALL(*) parsing algorithms, respectively. We use the Coq Proof Assistant to implement both interpreters, and to prove them sound and complete with respect to high-level specifications. Vermillion provably terminates without error on all LL(1) grammars (grammars for which parsing decisions are unambiguous given a single token of lookahead), and CoStar guarantees error-free termination on all non-left-recursive grammars. Vermillion and CoStar demonstrably run in linear time on a range of grammars for popular programming languages and data formats. In addition, CoStar accepts grammars augmented with semantic predicates—functions that represent non-context-free components of the language specification. CoStar applies these functions at parse time to ensure that its input is semantically well-formed. As part of the CoStar performance evaluation, we integrate the interpreter with Verbatim, a verified lexical analysis tool that we developed in parallel with CoStar. Together, these two tools constitute a verified pipeline for lexing and parsing.

The Dissertation Committee for Sam Lasser
certifies that this is the approved version of the following dissertation:

Formal Verification of Top-Down Parser Interpreters

Committee:

Kathleen Fisher, Supervisor

Chris Casinghino

Jeffrey Foster

Norman Ramsey

Josephine Wolff

Acknowledgments

My advisor, Kathleen Fisher, showed me by example what computer science research at its best looks like. Her ability to approach ambitious problems with both purpose and a sense of fun is one that I will try to emulate. Kathleen helped me stay focused on our long-term research goals, made time to check in with me despite her many obligations, and treated me as a collaborator as well as an advisee. For all of these reasons, I'm grateful to have been her student.

My Draper supervisor, Chris Casinghino, generously shared his time and his expertise in formal verification with me. He also introduced me to the pleasures of interactive theorem proving as a collaborative social process; on many occasions, talking through the details of a knotty proof with him helped me find a path forward.

Kathleen and Chris merit additional thanks for understanding that there's more to life than research, and for accommodating my family commitments. I'm fortunate to have worked with two advisors who are reasonable human beings as well as excellent technical problem solvers.

I thank Jeff Foster, Josephine Wolff, and Norman Ramsey for serving on my dissertation committee, and for their thoughtful feedback on this document. Courses that I took with Jeff and Norman challenged me and gave me a bigger toolbox to draw from in my research.

I had many enjoyable conversations with Cody Roux, who served as an unofficial third advisor. Whiteboard sessions with him helped me get unstuck at several key points, and his enthusiasm for logic and programming language theory is inspiring.

Mentoring and collaborating with Derek Egolf was among my most gratifying experiences at Tufts. I look forward to seeing where his talents take him.

To Brian LaChance, Ferdinand Vesely, Fox Huston, Jared Chandler, Jeanne-Marie Musca, Karl Cronburg, Lucía Nuñez, Mark Aldrich, Matt Ahrens, Milod Kazerounian, Moses Huang, Nate Bragg, Sasha Fedchin, and the rest of the TuPL community: thank you for your feedback on papers and practice talks, for sharing your own work, and for being my social community at Tufts.

To Donna Cirelli, Ellen Quirk, Jenny Mooney, Megan Monaghan, Sandie Schulenberg, Sarah Richmond, and the rest of the Tufts CS department staff: thank you for keeping the de-

partment running, and for your patience and flexibility when the occasional deadline slips by unheeded.

My PhD research was funded by a Draper Scholarship. I'm grateful to Draper for its commitment to supporting academic research.

I wouldn't have been able to finish the work described in this dissertation without child care support. I thank Khadija Barre, Angela Kenneally, and the Waverley Square Daycare staff for providing my kids with a safe and stimulating environment.

My parents, Hester and Jeff, and my sister, Nina, have been constant sources of support throughout my life—thank you for encouraging me to follow my interests wherever they lead. I'm also grateful for the support of my in-laws, Karen and Mark.

Henry and Esme have provided a welcome contrast with academic research, helped me to maintain perspective, and made life more colorful with their energy and curiosity. Finally, I thank Anna for sharing the joys and occasional struggles of PhD life and parenthood with me, and for her love and partnership.

Sam Lasser

TUFTS UNIVERSITY
August 2022

Contents

List of Figures

1

Introduction

Parsers—tools for analyzing the syntax of programming languages and data formats—are natural targets for formal verification. While the theoretical foundations of parsing are well-understood, parsing is error-prone in practice and is often implemented in an ad hoc manner. Parsers are also important components of high-assurance software; applications 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 [Ormandy 2017], allowed attackers to obtain the private data of nearly 150 million people from a major credit agency [Goodin 2017; NIST 2017], and enabled remote code execution on networked devices [NIST 2016, 2020; Goodin 2020; Kumar 2020]. One survey of the Common Vulnerabilities and Exposures (CVE) system even found that 80% of CVEs are related to input validation failures [DARPA 2018]. These vulnerabilities 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 into several general categories. Tools based on bottom-up parsing algorithms [Barthwal and Norrish 2009; Jourdan et al. 2012] provide limited *termination guarantees*; they do not guarantee termination on all inputs, and thus are not decision procedures for language membership. Parsers for general CFGs [Danielsson 2010; Ridge 2011; Firsov and Uustalu 2014, 2015] 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. Finally, purely CFG-based verified parsers are necessarily unable to handle non-context-free language specifications; such parsers are therefore limited in their *expressiveness*. Many commonly used data formats have specifications that are not entirely context-free, or that sometimes include non-context-free extensions. For example, an application that consumes JSON data might only guarantee correct behavior when JSON objects (collections of key/value pairs) in its input contain no duplicate keys.

This dissertation describes Vermillion and CoStar, two formally verified parser interpreters[1] that represent attractive points in the verified parsing space because of their correctness guarantees, empirical performance results, and expressiveness. Vermillion and CoStar are based on the LL(1) and ALL(*) parsing algorithms, respectively; these algorithms are characterized as "top-down" because they search the space of possible outputs in a top-down manner. The interpreters are implemented and verified with the Coq Proof Assistant, a tool for developing programs alongside machine-checked proofs of their correctness. Each interpreter takes in a grammar and a sequence of tokens; given valid input, it produces a semantic value with a user-specified type (an initial version of CoStar simply produces a parse tree). This output is provably correct with respect to the input, where the definition of correctness is formalized in a high-level specification.

Concretely, this dissertation reports on the following contributions:

- For all LL(1) grammars, Vermillion is provably sound and complete relative to a high-level specification, and it is guaranteed to terminate without error. A Vermillion parser for the JSON data format lives up to the LL(1) algorithm's theoretical guarantee of linear-time performance, and the parser is only two to four times slower than an unverified parser produced by Menhir [Pottier and Régis-Gianas 2016], a popular OCaml parser generator.

- A purely syntactic (parse tree-producing) version of CoStar is provably sound, complete, and error-free for all non-left-recursive grammars. The interpreter tags parse trees with labels indicating whether the input is ambiguous; its ambiguity detection mechanism is provably correct. The interpreter empirically runs in linear time on a set of unambiguous grammars for widely used programming languages and data formats; its performance comes within a constant factor of ANTLR's performance on these benchmarks.

- An extended version of CoStar (called CoStar++ in this dissertation) supports two features that increase the tool's expressiveness: *semantic predicates* enforce non-context-free properties of the input specification, and *semantic actions* construct a semantic value with a user-defined type. For all non-left-recursive CFGs, CoStar++ terminates without error, and it is sound and complete with respect to a specification that takes predicates and actions into account. Preserving these correctness guarantees in the presence of semantic predicates requires nontrivial changes to the ALL(*) algorithm. In a performance evaluation, we use CoStar++ to produce parsers for several real data formats with non-context-free specifications, and we demonstrate that these parsers run in linear time. As part of the evaluation, we integrate CoStar++ with Verbatim [Egolf et al. 2021, 2022], a verified lexing tool developed in a parallel line of research. The end result is an interpreter that is suitable for use in a verified lexing and parsing pipeline, and that captures expressive language specifications and produces user-defined semantic values through its support for predicates and actions.

[1] A note on terminology: the term "parser generator" commonly refers to a tool that generates parser source code from a grammar. For example, ANTLR [Parr et al. 2014] generates mutually recursive functions that correspond to the grammar's structure, and Yacc [Johnson 1978] generates a source code representation of parsing tables along with driver code. In contrast, Vermillion and CoStar do not generate source code; each tool converts a grammar to an in-memory data structure that a generic driver interprets at parse time. Throughout this dissertation, we refer to such a tool as a "parser interpreter" to distinguish it from code-generating tools.

Our development of Vermillion and CoStar substantiates the following thesis statement:

Interactive theorem proving techniques support the design of top-down parser interpreters that are correct with respect to high-level specifications; that terminate without error for well-defined classes of grammars; that run in linear time on a range of grammars for real-world programming languages and data formats; and that give the user fine-grained control over both the language that the interpreter accepts and the interpreter's output type.

1.1 Formal Methods: Specifications and Interactive Proofs

Formal methods are techniques for specifying how a computer program should behave, and for verifying that the program meets its specification, with mathematical precision. A specification S for program P can describe any aspect of the program's behavior. For example, S might take one of the following forms:

(S_1) P takes an integer and a list of integers as input, and it returns a boolean value as output.

(S_2) The running time of P is linear in the size of the input list.

(S_3) $P(x, l)$ returns `true` if integer x appears in list l and `false` otherwise.

S_1 states the program's type, S_2 gives the program's asymptotic complexity, and S_3 describes the program's extensional (input-output) behavior. This dissertation focuses on verifying programs with respect to specifications like S_3. Its primary technical results are theorems stating that a given program's output is correct with respect to its input, for some precise definition of correctness.

Formal methods can be categorized in terms of how much input they require from a human user. Methods at the "automated" end of the spectrum require less human guidance and are generally suitable for verifying simpler program properties. For example, type systems are a widely used family of (mostly) automated formal methods; type checking algorithms can automatically determine whether program values are annotated with the correct types, and for some type systems, inference algorithms can infer the types of program values without any human-provided type annotations. In contrast, formal methods at the "interactive" end of the spectrum are more human labor-intensive, but they enable the user to state and verify richer program correctness properties. The research described in this dissertation falls at the interactive end of the spectrum; it was conducted with an interactive theorem proving tool called the Coq Proof Assistant. Coq is a unified framework for writing both functional programs and proofs about those programs. The user constructs proofs in an interactive manner by filling in explicit proof steps, or by running built-in or custom proof search procedures. Coq's trusted proof-checking kernel ensures that user-constructed proofs satisfy their associated propositions.

Machine-checked proofs can provide stronger guarantees than the techniques that are most commonly used to build trust in the correctness of software: testing and *informal* proofs. Tests can show that a program behaves correctly on a finite set of inputs; when the space of inputs is infinite, such guarantees are necessarily incomplete. A pen-and-paper proof can establish that the program is correct for all possible inputs (e.g., by induction on the input), but such proofs

have their own limitations. First, the proof itself may contain errors, in which case the property it purports to prove may not hold. Second, informal proofs about the correctness of algorithms usually describe abstract versions of those algorithms. Often, we care not only that an algorithm is correct in its idealized textbook form, but that a particular *implementation* of that algorithm is correct. Mechanized verification with a tool like Coq solves both problems: a small, trusted kernel checks that each proof step is warranted, and proofs refer to actual source code.

1.2 Parsing: Search Strategies and Semantic Actions

At its core, parsing is a decision problem. Given a grammar $\mathcal{G}$ and a string w, the job of a parsing algorithm is to determine whether $w \in \mathcal{L}(\mathcal{G})$, where $\mathcal{L}(\mathcal{G})$ is the language (set of strings) that $\mathcal{G}$ represents. To show that this membership property holds for a given string, the parsing algorithm might produce a parse tree that serves a witness to the string's membership in $\mathcal{L}(\mathcal{G})$. In this sense, one can also view parsing as a search problem, in which a parsing algorithm searches the space of parse trees to identify a tree with the following properties:

- Internal nodes are labeled with grammar nonterminal symbols, and leaves are labeled with terminal symbols.
- The root is labeled with grammar start nonterminal S.
- Concatenating the leaves yields the string w.
- The recursive structure of the tree respects the structure of the grammar in the following sense: if concatenating the children of an internal node X yields the sequence of symbols γ, then $X ::= \gamma$ is a grammar production.

For many practical applications, the input that a parser's clients require is not a parse tree per se, but a filtered or transformed version of the tree. For example, the parsing module of a compiler front end might convert source code to an abstract syntax tree (AST)—i.e., a structured value that has an expressive type and that retains only those portions of the source code that are relevant to downstream compilation steps. For this reason, parsing tools often accept grammars that are augmented with *semantic actions*: functions that map parser inputs to semantic values with user-specified types. Both Vermillion and CoStar provide this functionality, although in Chapter 3 we report on an initial version of CoStar that merely produces parse trees.

Parsing algorithms are sometimes categorized based on whether they search the space of parse trees in a top-down or bottom-up manner. (Note that even when a parser produces semantic values instead of parse trees, we can still characterize the directionality of the search process in terms of trees.) Top-down algorithms, which build parse trees starting from the root, can be classified further based on their behavior at *decision points* in the search process. A decision point occurs when a parsing algorithm must choose which grammar right-hand side it should use to construct the subtrees of a nonterminal tree node. Recursive descent algorithms simply make arbitrary choices at decision points and backtrack if that choice fails; in contrast, the LL or "top-down with lookahead" algorithms make such decisions by looking ahead at the remaining input

string to determine which right-hand side(s) may result in a successful parse. Vermillion and CoStar are based on the LL(1) and ALL(*) parsing algorithms, respectively, both of which belong to the LL family.

Top-down algorithms share several strengths relative to other parsing strategies. Top-down parsers are known to produce clear error messages that describe errors in terms of the grammar's structure; in contrast, commonly used bottom-up parsers typically report errors in terms of the parser's internal data structures [Jeffery 2003; Pottier 2016]. Top-down algorithms can also easily be extended with semantic actions that produce user-defined data structures, and relative to bottom-up approaches, they are well-suited to handling language specifications that include data dependencies [Fisher and Gruber 2005].

1.3 Recurring Themes in Top-Down Parser Verification

In this section, we briefly describe several challenges that arise repeatedly across the research that this dissertation describes.

1.3.1 Proving Termination for Inherently Partial Algorithms

Top-down parsing algorithms such as LL(1) and ALL(*) can diverge on grammars that exhibit a syntactic pattern called left recursion. A grammar nonterminal X is left-recursive when a parse tree rooted at X can contain a level in which X is the leftmost node. The divergence of these algorithms on some inputs poses a problem because Gallina, the functional programming language embedded within Coq, is total: all recursive functions must provably terminate. This restriction is necessary to ensure the soundness of the tool's underlying logic. The metatheory of Coq belongs to the "propositions as types" paradigm, in which a logical proposition and its proof are represented as a type T and a program with type T, respectively. The type of a divergent program could correspond to a proof of $\bot$, rendering the logic unsound. For this reason, one cannot even define a non-terminating function in Coq, let alone prove other properties about it!

We ensure that Vermillion and CoStar terminate with the help of a lightweight technique for dynamic left recursion monitoring. Each tool tracks the set of nonterminal symbols that it has processed or "visited" since it last consumed an input token. This piece of state enables the tool to detect left-recursive loops in the grammar; the tool halts and returns an error value when it detects such a loop. Separately, we prove that such an error never arises when the tool is supplied with a non-left-recursive grammar. This strategy enables us to define programs that terminate on all grammars, and then prove additional correctness properties of those programs when they are given non-left-recursive grammars.

Note that while Vermillion and CoStar both check for left recursion dynamically to ensure termination, the portions of their correctness theorems that relate to left recursion are different. Vermillion's correctness guarantees hold for LL(1) grammars (grammars that are suitable for unambiguous parsing with a single token of lookahead). The Vermillion development includes a verified decision procedure that checks whether a grammar is LL(1); it also includes a proof that an LL(1) grammar is non-left-recursive. These components enable us to guarantee that a grammar

that passes the LL(1) check will never cause the parsing algorithm to return an error value (see Section 2.4.3 for details). In contrast, CoStar's correctness properties hold for *all* non-left-recursive grammars. Determining whether an arbitrary CFG is left-recursive should be decidable, but we have not yet verified a decision procedure for this property; each CoStar correctness theorem simply assumes a non-left-recursive grammar.

1.3.2 Working with Non-Structural Termination Measures

Dynamic left recursion detection alone is not enough to convince Coq's termination checker that the main Vermillion and CoStar parsing algorithms terminate. The termination checker can confirm that a recursive function f terminates when f is structurally recursive on one of its parameters—for example, when all recursive calls are made on the tail of the list that f takes as input. The termination argument for each of our parsers depend on several components of the parser's state, including the number of remaining tokens, the number of "visited" nonterminals, and the height of a stack in some cases. Because there is no single structurally decreasing parameter, Coq's termination checker is unable to infer automatically that the parser terminates.

When this situation arises, we construct an explicit termination proof with the help of a technique called well-founded recursion. In the context of parsing, well-founded recursion involves defining a function f that maps a parser's state σ to a measure value $m : A$. One must then prove that every recursive parser step causes m to decrease in terms of a binary $<$ ("less than") relation on A. In other words, if the parser transitions from state σ to state σ', then $f(\sigma') < f(\sigma)$. The $<$ relation must be well-founded, meaning that every decreasing chain of steps from an element to lesser ones is finite. This technique ensures that the number of possible recursive calls is bounded, and that the parser therefore terminates.

The challenges of proving termination for parsing algorithms are not unique to Vermillion and CoStar. The most directly comparable verified parsers—i.e., those that are based on deterministic CFG-based parsing algorithms—do not guarantee termination on all inputs [Barthwal and Norrish 2009; Jourdan et al. 2012] (see Section 5.2.1 for a more detailed discussion of this issue). In contrast, Vermillion and CoStar do provide such a guarantee and thus can be viewed as decision procedures for language membership.

1.3.3 Choosing Between Implicit and Explicit Stack Representations

There is a well-known equivalence between push-down (i.e., stack-based) automata and CFGs, and the stack is a natural data structure on which to base a parser implementation. The implementer has several possible stack representations to choose from. For example, one could use an implicit representation that involves co-opting the host language's function call stack to serve as the parser stack. In this approach, function calls and returns represent stack push and pop operations, respectively. Alternatively, one could represent the stack as an explicit data structure.

In our work on Vermillion and CoStar, we experimented with both implicit and explicit stack representations. We found that different representations lead to very different verification experiences, and that one must choose between them on a per-algorithm basis. Vermillion uses

the implicit approach. This choice enables the recursive structure of the parsing algorithm implementation to mirror the inductive structure of its correctness specification. In general, this kind of structural similarity leads to simple inductive proofs because the base cases of the code and specification are "aligned," and the same is true for the inductive cases. CoStar, on the other hand, uses an explicit stack representation, and we believe that this choice better suits the ALL(*) algorithm. At decision points, ALL(*) must sometimes examine non-local context (i.e., symbols from stack frames below the topmost one) to determine which grammar right-hand side to push onto the stack. Using a reified stack data structure ensures that this non-local information is always in scope. The explicit approach does create a slight structural mismatch between the code and its correctness specification. We bridge this gap with an elegant technique that involves defining invariants over the parser stack (and other components of the parser state). These invariants entail the desired high-level correctness properties when they hold over the parser's final state.

1.4 Dissertation Structure and Publication History

The remainder of this dissertation is structured as follows:

- Chapter 2 presents the Vermillion parser interpreter. This chapter is based on the Interactive Theorem Proving (ITP) 2019 paper "A Verified LL(1) Parser Generator" [Lasser et al. 2019a].
- Chapter 3 reports on an intermediate step in the development of the CoStar parser interpreter. The version of the interpreter that this chapter describes produces parse trees rather than semantic values. This chapter is based on the Programming Language Design and Implementation (PLDI) 2021 paper "CoStar: A Verified ALL(*) Parser" [Lasser et al. 2021c].
- Chapter 4 presents our work on extending CoStar with semantic predicates and actions. We refer to this extended version of the tool as CoStar++ to distinguish it from its parse tree-producing predecessor. Chapter 4 also describes our work on integrating CoStar++ with Verbatim, a verified lexical analysis tool that we developed in a related line of research.
- Chapter 5 surveys related work on the relationship between parsing and cybersecurity, and on alternative approaches to verified parsing. (In addition, Chapters 2 and 4 discuss related work that is directly comparable to the material in those chapters.)
- Chapter 6 concludes the dissertation.

Section 4.7 is based on two papers about verified lexing: "Verbatim: A Verified Lexer Generator" [Egolf et al. 2021] and "Verbatim++: Verified, Optimized, and Semantically Rich Lexing with Derivatives" [Egolf et al. 2022]. Derek Egolf is the first author of both papers; they describe research that Derek conducted with my supervision. Section 5.1 is based on my contributions to an in-preparation review paper about language-theoretic approaches to software security.

2

Vermillion: A Verified LL(1) Parser Interpreter

An LL(1) parser uses a recursive descent algorithm with a single token of lookahead to build a parse tree or semantic value for an input sequence. In this chapter, we present Vermillion, a tool that, when applied to grammar $\mathcal{G}$, produces an LL(1) parser for $\mathcal{G}$ iff such a parser exists. We use the Coq Proof Assistant to verify that Vermillion parsers are sound and complete with respect to the grammars used to produce them, and that they terminate without error on all inputs. As a case study, we extract Vermillion's source code and use it to produce a JSON parser. The parser runs in linear time; it is two to four times slower than an unverified parser for the same grammar.

2.1 Introduction

Parsing is a widely studied topic, and it encompasses a range of techniques with different advantages and drawbacks [Grune and Jacobs 2006]. One family of parsing algorithms is the "top-down with lookahead" or LL-style algorithms. These algorithms search the space of possible parse trees in a top-down manner, and they analyze the remaining input at decision points to avoid "wrong turns" that require backtracking. The common ancestor of the LL family is LL(1), a recursive descent algorithm that looks ahead at a single input token when it reaches decision points. Its descendants, including LL(k), LL(*), and ALL(*), share an algorithmic skeleton. Each of these approaches comes with different tradeoffs with respect to expressiveness vs. efficiency. For example, LL(1) operates on a restricted class of grammars and offers linear-time execution, while ALL(*) accepts a larger class of grammars and processes n tokens in $O(n^4)$ time [Parr et al. 2014]. Different algorithms are therefore suited to different applications; it can be advantageous to choose the least expressive—and therefore most efficient—algorithm compatible with the language being parsed.

In this chapter, we present Vermillion, a formally verified LL(1) parser interpreter. We implemented and verified Vermillion using the Coq Proof Assistant [Coq Development Team 2019], a popular interactive theorem prover. Vermillion has two main components. The first is a *parse table generator* that, when applied to a context-free grammar, produces an LL(1) parse table—an encoding of the grammar's lookahead properties—if such a table exists for the grammar. The

second component is an implementation of the LL(1) parsing algorithm that is parameterized by a parse table. By converting a grammar to a table and then partially applying the parsing algorithm to the table, the user obtains a parser that is specialized to the original grammar.

The main contributions of this work are as follows:

1. **End-to-End Correctness Proofs** – We prove that both the parse table generator and the parsing algorithm are sound and complete. The generator produces a correct LL(1) parse table for any grammar if such a table exists. The parsing algorithm produces a semantic value that is correct in terms of the LL(1) parse table and token sequence to which the algorithm is applied. Although prior work has verified some of the steps involved in LL(1) parse table generation [Barthwal and Norrish 2009], to the best of our knowledge, our LL(1) parse table generator and parser are the first formally verified versions of these algorithms.

2. **Total Algorithm Implementations** – We prove that the parse table generator and parsing algorithm terminate on both valid and invalid inputs without the use of fuel-like parameters. This property distinguishes Vermillion from comparable verified parsers based on the context-free grammar formalism. Some existing verified parsers are only guaranteed to terminate on valid inputs, or they ensure termination by means of a fuel parameter, which can produce "out of fuel" return values that do not clearly indicate success or failure. A guarantee of termination on all inputs is useful for ruling out denial-of-service attacks against the parser.

3. **Efficient Extractable Code** – We used Coq's Extraction mechanism [Letouzey 2008] to convert Vermillion to OCaml source code and generated a parser for a JSON grammar. We then used Menhir [Pottier and Régis-Gianas 2016], a popular OCaml parser generator, to produce an unverified parser for the same grammar and compared the two parsers' performance on a JSON data set. The verified parser was two to four times slower than the unverified and optimized one, which is similar to the reported results for other certified parsers [Koprowski and Binsztok 2010; Jourdan et al. 2012]. Our implementation empirically lives up to the LL(1) algorithm's theoretical linear-time guarantees.

Along the way, we deal with several interesting verification challenges. The parse table generator performs dataflow analyses with non-obvious termination metrics over a context-free grammar. To implement and verify these analyses, we make ample use of Coq's tools for defining non-structurally recursive functions with well-founded measures, and we prove a large collection of domain-neutral lemmas about finite sets and maps that may be useful in other developments. The parser also uses non-structural recursion on a well-founded measure, and our initial implementation had to perform an expensive runtime computation to terminate provably; in the final version, we make judicious use of dependent types to avoid this penalty while still proving termination. Our parser completeness proof relies on a lemma stating that if a correct LL(1) parse table exists for some grammar, then the grammar contains no left recursion. The proof of this lemma is quite intricate, and we were unable to find a rigorous proof of this seemingly intuitive fact in the literature.

The Vermillion formalization consists of roughly 8,000 lines of Coq definitions and proofs. It is open-source and available online [Lasser et al. 2019b].

This chapter is organized as follows: in §2.2, we review background material on context-free grammars and LL(1) parsing. In §2.3, we describe the high-level structure of our parse table generator and its correctness proofs. In §2.4, we present the LL(1) parsing algorithm implementation and its correctness properties. In §2.5, we present the results of evaluating Vermillion's performance on a JSON benchmark. We discuss related work in §2.6 and possible extensions of our work in §2.7.

2.2 Grammars and Parse Tables

2.2.1 Grammars

A Vermillion grammar is composed of terminal symbols of type $\mathcal{T}$ and nonterminal symbols of type $\mathcal{N}$, where $\mathcal{T}$ and $\mathcal{N}$ are user-defined types. Throughout this dissertation, we use the letters $\{a, b, c\}$ as terminal names, $\{X, Y, Z\}$ as nonterminal names, $\{s, s', ...\}$ as names for arbitrary symbols (terminals or nonterminals), and $\{\alpha,\ \beta,\ \gamma\}$ as names for sentential forms (finite sequences of symbols).

A grammar consists of a start symbol $s \in \mathcal{N}$ and a finite sequence of productions $\mathcal{P}$ (described in detail below). In addition, we require the grammar writer to provide a mapping from each grammar symbol s to a semantic type $[\![s]\!]$ in the host language (i.e., a Coq type). We borrow this mapping from a certified LR(1) parser development [Jourdan et al. 2012]; it enables us to specify the behavior of a parser that maps a valid input to a *semantic value* with a user-defined type, rather than simply recognizing the input as valid or building a generic parse tree for it. The symbols-to-types mapping supports the construction of flexible semantic values as follows:

- The parser consumes a list of tokens w, where each token is a dependent pair $(a\ \&\ v)$ of a terminal symbol a and a semantic value $v : [\![a]\!]$. When the parser successfully consumes a token $(a\ \&\ v)$, it produces the value v.
- A production $X ::= \gamma\ \{f\}$ consists of a left-hand nonterminal X, a right-hand sentential form γ, and a semantic action f of type $[\![\gamma]\!] \to [\![X]\!]$. The type $[\![\gamma]\!]$ is the tuple type computed from the symbols in γ and is defined as follows:

$$[\![\bullet]\!] = \mathbb{1}$$
$$[\![s\beta]\!] = [\![s]\!] \times [\![\beta]\!]$$

A Vermillion parser produces a semantic value by recursively executing semantic actions in the following manner: after the parser uses a production's right-hand side to construct a tuple of type $[\![\gamma]\!]$, it applies f to this tuple to produce a final semantic value of type $[\![X]\!]$. The user provides semantic actions at grammar definition time; these actions are dependently typed Coq functions. Throughout this chapter, we use the notation $X ::= \gamma$ to refer to a production when its semantic action is clear from context or irrelevant to the discussion.

```
(* type definition for semantic values *)
sem_val :=
| Bool   bool
| Int    int
| Str    string
...

fun f (vs : list sem_val) : option sem_val :=
  match vs with
  | [Bool b; Int i; Str s] =>
    let i' := if b then i else length s
    in  Some (Int i')
  | _ => None
  end
```

(a) Simply typed action `f` matches on `vs` (the list of semantic values for production right-hand side `abc`) and fails when `vs` is not a three-element list consisting of a boolean, integer, and string.

```
(* map from symbols to semantic types *)
⟦a⟧ = bool
⟦b⟧ = int
⟦c⟧ = string
⟦X⟧ = int
 ...

fun f' (vs : ⟦abc⟧) : ⟦X⟧ :=
  match vs with
  | (b, (i, (s, _))) =>
    if b then i else length s
  end
```

(b) The signature of dependently typed action `f'` captures the fact that right-hand side `abc` must produce a boolean, integer, and string, and it ensures that the action produces a value of the correct semantic type for left-hand side `X`.

Figure 2.1: Pseudocode semantic actions for a production `X` ::= `abc` in a simply typed setting (a) and in a dependently typed setting (b).

The dependent representation of these actions is an important part of the tool's expressiveness. In a simply typed setting, the user would need to write action code in a tedious and potentially error-prone manner. In particular, one would need to pattern match on the list of parse trees or semantic values $\bar{v}$ that a grammar right-hand side γ produces and fail when $\bar{v}$ has an unexpected structure, even though the structure of $\bar{v}$ is wholly determined by that of γ. In Figure 2.1a, the simply typed action `f` for a hypothetical production `X` ::= `abc` exemplifies this pattern. Even if the parser's structure and the grammar's other semantic actions guarantee that the parser will produce a list containing a boolean, integer, and string for right-hand side `abc`, this fact is not captured in the type signature of `f`; the action must fail on cases where the list has some other shape. In contrast, the signature of dependently typed action `f'` (Figure 2.1b) guarantees that the semantic tuple for `abc` has the expected shape, eliminating the need for error handling.

2.2.2 LL(1) Derivations

An important component of Vermillion's correctness specification is a grammatical derivation relation called LL(1)ValueDer over a grammar symbol s, a word or token sequence w that s derives, and a semantic value $v : [\![s]\!]$ that s produces for w. Because it is useful for a parser to produce a semantic value for a prefix of its input sequence and return the remainder of the sequence along with the value, the derivation relation also includes the *remainder*, or the unparsed suffix of the input. The relation has the judgment form $s \xrightarrow{v} w \mid r$, which is read, "$s$ derives w, producing v and leaving r unparsed."

The LL(1)ValueDer relation appears in Figure 2.2. It is mutually inductive with an analogous relation called LL(1)ValuesDer (also in Figure 2.2) over a list of symbols γ, a word w, a

$$\boxed{\text{LL(1)ValueDer} : s \xrightarrow{v \,:\, [\![s]\!]} w \mid r}$$

$$\frac{}{a \xrightarrow{v} (a \,\&\, v) \mid r}\ \textsc{TerminalDer} \qquad \frac{\begin{array}{c} X ::= \gamma\ \{f\} \in \mathcal{P} \\ \mathsf{peek}(w +\!\!+ r) \in \textsc{Lookahead}(X ::= \gamma) \\ \gamma \xrightarrow{\bar{v}} w \mid r \end{array}}{X \xrightarrow{f(\bar{v})} w \mid r}\ \textsc{NonterminalDer}$$

$$\boxed{\text{LL(1)ValuesDer} : \gamma \xrightarrow{\bar{v} \,:\, [\![\gamma]\!]} w \mid r}$$

$$\frac{}{\bullet \xrightarrow{\mathtt{tt}} \epsilon \mid r}\ \textsc{NilDer} \qquad \frac{s \xrightarrow{v} w \mid w' +\!\!+ r \qquad \gamma \xrightarrow{\bar{v}} w' \mid r}{s\gamma \xrightarrow{(v,\bar{v})} w +\!\!+ w' \mid r}\ \textsc{ConsDer}$$

Figure 2.2: Derivation relations for symbols and lists of symbols.

tuple of semantic values $\bar{v} : [\![\gamma]\!]$, and a remainder r. This second relation has the judgment form $\gamma \xrightarrow{\bar{v}} w \mid r$, which is read, "$\gamma$ derives w, producing $\bar{v}$ and leaving r unparsed."

Grammatical derivation relations are a standard way of specifying the behavior of parsers. Below, we give the informal meaning of each LL(1)ValueDer and LL(1)ValuesDer rule; the parsers described in later chapters of this dissertation are specified with derivation relations that follow the same pattern.

- The TerminalDer rule says that a terminal symbol a derives the singleton token sequence $(a \,\&\, v)$ and produces v, the token's semantic value component.
- The NilDer rule says that the empty symbol sequence $\bullet$ derives the empty token sequence ϵ and produces `tt`, the only value of type `unit`.
- The ConsDer rule says that if symbol s derives token sequence w to produce value v, and sentential form γ derives token sequence w' to produce the tuple of semantic values $\bar{v}$, then sentential form $s\gamma$ derives $w +\!\!+ w'$ (the concatenation of w and w') to produce the new semantic tuple $(v, \bar{v})$.
- The NonterminalDer rule is the only LL(1)-specific rule in these relations. The `peek` function returns a lookahead value $l \in \mathcal{T} \cup \{\mathtt{EOF}\}$ that is either the first token of the input sequence $w +\!\!+ r$, or `EOF` if the entire sequence is empty. The rule itself states that if $\mathsf{peek}(w +\!\!+ r)$ is a "lookahead token" for production $X ::= \gamma\ \{f\}$, and production right-hand side γ derives token sequence w to produce the semantic tuple $\bar{v}$, then left-hand side X derives w and produces the result of applying semantic action f to $\bar{v}$.

Before giving a precise definition of a "lookahead token," we first need to introduce the definitions of several predicates that are commonly used in parsing theory to relate a grammar's structure to its semantics.

$$\textsc{NuSym}\ \frac{X ::= \gamma\ \{f\} \in \mathcal{P} \qquad \textsc{Nullable}(\gamma)}{\textsc{Nullable}(X)} \qquad \textsc{NuGamma}\ \frac{\forall i \in \{1...n\},\ \textsc{Nullable}(s_i)}{\textsc{Nullable}(s_1...s_n)}$$

Figure 2.3: NULLABLE relation.

$$\textsc{FirstTerminal}\ \frac{}{a \in \textsc{First}(a)} \qquad \textsc{FirstNonterminal}\ \frac{X ::= \gamma\ \{f\} \in \mathcal{P} \qquad a \in \textsc{First}(\gamma)}{a \in \textsc{First}(X)} \qquad \textsc{FirstGamma}\ \frac{\textsc{Nullable}(\alpha) \qquad a \in \textsc{First}(s)}{a \in \textsc{First}(\alpha s \beta)}$$

Figure 2.4: FIRST relation.

2.2.3 NULLABLE, FIRST, and FOLLOW

A nullable grammar symbol is a symbol that can derive the empty word ϵ. The NULLABLE relation (Figure 2.3) captures the syntactic pattern that makes a symbol nullable. A nonterminal is nullable if it appears on the left-hand side of a production and every symbol on the right-hand side is also nullable (note that an empty right-hand side makes the left-hand nonterminal trivially nullable). A sentential form γ is nullable if it consists entirely of nullable symbols. We overload our notation for nullable symbols, writing $\textsc{Nullable}(\gamma)$ to represent the fact that γ is a nullable symbol sequence.

The FIRST relation (Figure 2.4) for a symbol s describes the set of terminals that can begin a word derived from s. If s derives a word beginning with terminal a, then $a \in \textsc{First}(s)$. Once again, we extend this concept to sentential forms, writing $a \in \textsc{First}(\gamma)$ if γ derives a word that begins with a.

The FOLLOW relation (Figure 2.5) for a symbol s describes the set of terminals that can appear immediately after a word derived from s. There is a standard practice among parser implementers of placing the `EOF` symbol in $\textsc{Follow}(S)$, where S is the start symbol, so that the parser can consume the entire input sequence. We follow this practice by adding the FOLLOWSTART rule to the relation.

With these definitions in hand, we can give a precise definition for the judgment form $l \in \textsc{Lookahead}(X ::= \gamma)$ ("l is a lookahead token for production $X ::= \gamma$") in Figure 2.6. Intuitively, l is a token that, when it begins a token sequence $\bar{t}$, "predicts" that the production can derive a

$$\textsc{FollowStart}\ \frac{S \text{ is the start symbol}}{\texttt{EOF} \in \textsc{Follow}(S)} \qquad \textsc{FollowRight}\ \frac{X ::= \alpha Y \beta\ \{f\} \in \mathcal{P} \qquad a \in \textsc{First}(\beta)}{a \in \textsc{Follow}(Y)}$$

$$\textsc{FollowLeft}\ \frac{X ::= \alpha Y \beta\ \{f\} \in \mathcal{P} \qquad \textsc{Nullable}(\beta) \qquad l \in \textsc{Follow}(X)}{l \in \textsc{Follow}(Y)}$$

Figure 2.5: FOLLOW relation.

$$\textsc{FirstLookahead}\ \frac{l \in \textsc{First}(\gamma)}{l \in \textsc{Lookahead}(X ::= \gamma)} \qquad \textsc{FollowLookahead}\ \frac{\textsc{Nullable}(\gamma) \qquad l \in \textsc{Follow}(X)}{l \in \textsc{Lookahead}(X ::= \gamma)}$$

Figure 2.6: LOOKAHEAD relation.

(`X` is the start symbol)

1. `X ::= aY`
2. `X ::= Zc`
3. `Y ::= •`
4. `Z ::= b`
5. `Z ::= Y`

	a	**b**	**c**	**EOF**
X	`aY`	`Zc`	`Zc`	
Y			•	•
Z		`b`	`Y`	

Figure 2.7: Example grammar and its LL(1) parse table.

prefix of $\bar{t}$. As a special case, if the production derives $\bar{t} = \epsilon$, then `EOF` $\in \textsc{Lookahead}(X ::= \gamma)$. When an LL(1) parser builds a derivation from nonterminal X for a prefix of $\bar{t}$, it "looks ahead" at $\bar{t}$ and applies a production $X ::= \gamma$ such that $\mathsf{peek}(\bar{t}) \in \textsc{Lookahead}(X ::= \gamma)$.

2.2.4 Parse Tables

An LL(1) parse table is a data structure that encodes a grammar's lookahead information. An LL(1) parser uses a parse table as an oracle; it consults the table to choose which productions to apply as it builds a derivation for a token sequence.

A parse table's rows are labeled with nonterminals and its columns are labeled with lookahead symbols. Its cells contain production right-hand sides. A cell at row X and column l that contains γ, written $(X, l) \mapsto \gamma$, represents the fact $l \in \textsc{Lookahead}(X ::= \gamma)$.

Figure 2.7 contains an example grammar and its LL(1) parse table. Table cell (**X**, **b**), for instance, contains `Zc` (the right-hand side of production 2) because of the fact `b` $\in \textsc{First}($`Zc`$)$. Cell (**Z**, **c**) contains `Y` (the right-hand side of production 5) because of the facts $\textsc{Nullable}($`Y`$)$ and `c` $\in \textsc{Follow}($`Z`$)$.

A correct LL(1) parse table for grammar $\mathcal{G}$ contains all and only the lookahead facts for $\mathcal{G}$:

Definition 2.2.1 (Parse table correctness)**.** $\forall X\ l\ \gamma, (X, l) \mapsto \gamma \iff l \in \textsc{Lookahead}(X ::= \gamma)$

Not every grammar has a correct LL(1) parse table. If $l \in \textsc{Lookahead}(X ::= \gamma)$ and $l \in \textsc{Lookahead}(X ::= \gamma')$, where $\gamma \neq \gamma'$, then no correct table exists for $\mathcal{G}$—a parser would be unable to choose whether to apply γ or γ' upon encountering nonterminal X and lookahead token l. A grammar that has a correct LL(1) parse table is called an LL(1) grammar.

2.3 Parse Table Generator Correctness Properties and Verification

We now describe the process of developing and verifying an LL(1) parse table generator. Our first goal is to define the following Coq function:

```
parseTableOf : grammar → sum error_message parse_table
```

(A value of type `sum A B` is either `inl A` or `inr B`.) We then wish to prove that the function is both *sound* (every table that it produces is a correct LL(1) parse table for its input grammar) and *complete* (it produces a correct LL(1) parse table for the grammar if such a table exists).

2.3.1 Structure of Parse Table Generator

Many standard compiler references describe variations on an algorithm for constructing an LL(1) parse table from a grammar. The algorithm typically involves computing the grammar's NULLABLE, FIRST, and FOLLOW sets, and then constructing the table from these sets (or returning an error value if a table cell contains multiple entries, in which case no correct parse table exists for the grammar). Appel's *Modern Compiler Implementation in ML* [1998], for example, contains pseudocode for performing the first of these two steps. The algorithm presents several interesting challenges from a verification standpoint:

1. It uses an "iterate until convergence" strategy to perform a dataflow analysis over the grammar. Such an algorithm is difficult to implement in a total language like Gallina, the functional programming language embedded in Coq, because it has no obvious (i.e., syntactic) termination metric.

2. NULLABLE, FIRST, and FOLLOW are all computed simultaneously, so a proof of the function's correctness must simultaneously deal with the correctness of all three sets.

It is also possible to perform the NULLABLE, FIRST, and FOLLOW dataflow analyses sequentially (in that order) because each analysis depends only on the previous ones. This sequential approach is preferable from a proof engineering perspective, because we can clearly state the correctness criteria for each step and verify the implementation independently of the other steps. It is also preferable from a code reuse perspective, because some individual steps may be useful in the context of other developments (for example, several parsing algorithms need to compute the set of nullable nonterminals). Therefore, we structure our parse table generator as a pipeline of small functions that perform the following steps:

1. Compute Nullable, the set of nullable nonterminals.

2. For each nonterminal X, compute First(X) (using Nullable).

3. For each nonterminal X, compute Follow(X) (using Nullable and First).

4. Using Nullable, First, and Follow, compute the set of parse table entries.

5. Build a table from the set of entries, or return an error if the set contains a conflict.

Several steps involve similar reasoning and require the same proof techniques. In the next section, we examine step 1 and its correctness proof in detail to illustrate these techniques.

```
Lemma nullablePass_neq_candidates_lt :
  forall (ps : list production) (nu : NtSet.t),
    ~ NtSet.Equal nu (nullablePass ps nu)
    -> countNullCands ps (nullablePass ps nu) < countNullCands ps nu.

Program Fixpoint mkNullableSet' (ps : list production) (nu : NtSet.t)
        { measure (countNullCands ps nu) } : NtSet.t :=
  let nu' := nullablePass ps nu in
  if NtSet.eq_dec nu nu' then nu else mkNullableSet' ps nu'.
Next Obligation.
  apply nullablePass_neq_candidates_lt; auto.
Defined.

Definition mkNullableSet (g : grammar) : NtSet.t :=
  mkNullableSet' g.(prods) NtSet.empty.
```

Figure 2.8: Selected portions of the `mkNullableSet` implementation. The `mkNullableSet` function passes the input grammar's productions and an initially empty set of nullable nonterminals to auxiliary function `mkNullableSet'`. The latter function performs single passes of the NULLABLE dataflow analysis (represented by the `nullablePass` function) over the productions until the set converges. The `nullablePass_neq_candidates_lt` lemma states that if a `nullablePass` iteration alters the set of nullable nonterminals, then the number of remaining nullable *candidates* decreases. This lemma is used to obtain a provably terminating definition of `mkNullableSet'`.

2.3.2 Implementation of NULLABLE Dataflow Analysis

The first step in the parse table generation process is to compute the set of nullable nonterminals. Our goal is to define the function

```
mkNullableSet : grammar  →  NtSet.t
```

(where `NtSet.t` is the type of finite sets of nonterminals), and then prove that when this function is applied to grammar $\mathcal{G}$, the resulting set contains all and only the nullable nonterminals from $\mathcal{G}$. We formalize this correctness property and theorem statement in Coq as follows:

Definition 2.3.1. NULLABLE set ν is correct with respect to grammar $\mathcal{G}$ when $X \in \nu \iff$ NULLABLE(X) for all $X \in \mathcal{N}$.

Theorem 2.3.1 (`mkNullableSet` correctness)**.** NULLABLE set (`mkNullableSet` $\mathcal{G}$) is correct with respect to grammar $\mathcal{G}$.

Portions of the `mkNullableSet` implementation appear in Figure 2.8. The auxiliary function `mkNullableSet'` takes a (possibly incomplete) NULLABLE set `nu` as an argument and performs a single pass of the NULLABLE dataflow analysis over the grammar's productions, which produces a (possibly updated) set `nu'`. If `nu` has converged—i.e., if it is a fixed point of the dataflow analysis—then it is returned. Otherwise, the algorithm performs another iteration of the analysis, using `nu'` as the starting point.

Because of this algorithm's "iterate until convergence" structure, we need to do some extra work to prove that it terminates. To accomplish this task, we use Coq's Program extension [Sozeau

2007], which provides support for defining functions using well-founded recursion. The `Program Fixpoint` command enables the user to define a non-structurally recursive function by providing a measure—a mapping from one or more function arguments to a value in some well-founded relation $\mathcal{R}$—and then showing that the measure of recursive call arguments is less than that of the original arguments in $\mathcal{R}$.

In the case of `mkNullableSet'`, the measure (called `countNullCands` in Figure 2.8) is the cardinality of `nu`'s complement with respect to the universe $\mathcal{U}$ of grammar nonterminals. We then prove that if the NULLABLE set is different before and after a single iteration of the analysis, then the more recent version contains a nonterminal that was not present in the previous version, and therefore that the set's complement with respect to $\mathcal{U}$ has decreased (this fact is captured in the lemma `nullablePass_neq_candidates_lt`).

Now that we have a suitable definition of `mkNullableSet` and a proof that it terminates, we turn to the proofs of its correctness properties: soundness (Section 2.3.3) and completeness (Section 2.3.4).

2.3.3 Soundness of NULLABLE Analysis

One property of `mkNullableSet` that we wish to verify is that the function is *sound*—i.e., every nonterminal in the set that it returns really is nullable in the input grammar:

Definition 2.3.2 (Nullable soundness)**.** Nullable set ν is sound with respect to grammar $\mathcal{G}$ when $X \in \nu \implies \textsc{Nullable}(X)$ for all $X \in \mathcal{N}$.

Theorem 2.3.2 (`mkNullableSet` soundness)**.** Set (`mkNullableSet` $\mathcal{G}$) is sound with respect to $\mathcal{G}$.

The soundness proof's structure arises from the intuition that soundness holds not only of `mkNullableSet`'s final return value, but of the intermediate sets that the function computes along the way—in other words, soundness is an invariant of the function. We prove this invariant with the following two lemmas:

1. The initial set passed to `mkNullableSet'` is sound.
2. If ν is sound, then `mkNullableSet'` applied to ν is also sound.

Lemma 1 is simple to prove, because the initial ν argument passed to `mkNullableSet'` is the empty set, which is trivially sound. Our earlier reasoning about the termination properties of `mkNullableSet'` pays dividends in the proof of lemma 2, because we can proceed by well-founded induction on the function's measure. The main lemma involved in this proof states that a single iteration of the dataflow analysis (called `nullablePass` in Figure 2.8) preserves soundness of the NULLABLE set.

2.3.4 Completeness of NULLABLE Analysis

In addition to being sound, `mkNullableSet` should be complete—that is, every nullable nonterminal from the input grammar should appear in the set that the function returns:

Definition 2.3.3 (Nullable completeness)**.** Nullable set ν is complete with respect to grammar $\mathcal{G}$ when $\textsc{Nullable}(X) \implies X \in \nu$ for all $X \in \mathcal{N}$.

Theorem 2.3.3 (`mkNullableSet` completeness)**.** The set (`mkNullableSet` $\mathcal{G}$) is complete with respect to $\mathcal{G}$.

Once again, the proof is based on well-founded induction on the `mkNullableSet'` measure. In the interesting case, we must prove the set ν complete given the fact that a `nullablePass` iteration leaves ν unchanged. In other words, we need to show that any fixed point of the dataflow analysis is complete. We isolate this fact in the following lemma:

Lemma 2.3.4 (Nullable fixed point completeness)**.** If the sets ν and (`nullablePass` $\mathcal{P}$ ν) are extensionally equal (i.e., if they contain the same elements), then ν is complete with respect to $\mathcal{G}$.

After some simplification, we are left with this goal:

$$\frac{\textsc{Nullable}(X) \qquad \nu = \texttt{nullablePass}\ \mathcal{P}\ \nu}{X \in \nu}$$

The proof proceeds by induction on the $\textsc{Nullable}(X)$ judgment. Because this relation is mutually inductive with the $\textsc{Nullable}(\gamma)$ relation for lists of symbols, we use Coq's `Scheme` facility to generate a suitably powerful mutual induction principle for the two relations. Using this principle requires some extra work because the programmer must manually specify the two properties that the induction is intended to prove—one for symbols, and one for lists of symbols.

It can be difficult to come up with the right instantiations for mutual induction principles such as this one. For several of the proofs in this development, such a choice was the most difficult step. In some cases, we were able to avoid this problem by finding mutual induction-free variants of relations whose pencil-and-paper definitions seem to call for mutuality.

2.3.5 Correctness of Parse Table Generator

Computing the Nullable set is the first of several dataflow analyses involved in generating an LL(1) parse table. The correctness proofs for the remaining steps are similar in structure to the Nullable proofs. For example, the First and Follow analyses each have a soundness proof based on the fact that soundness is an invariant of the analysis, and a completeness proof based on the fact that a fixed point of the analysis must be complete.

After proving each step correct given the correctness of previous steps, we can verify `parseTableOf`—the function that implements the entire sequence—simply by chaining together the proofs for the individual steps. The `parseTableOf` soundness and completeness theorem statements appear below:

Theorem 2.3.5 (`parseTableOf` soundness)**.** If `parseTableOf` $\mathcal{G}$ = `inr` t, then t is a correct LL(1) parse table for $\mathcal{G}$.

Theorem 2.3.6 (`parseTableOf` completeness)**.** If grammar $\mathcal{G}$ has unique productions (defined below) and t is a correct LL(1) parse table for $\mathcal{G}$, then `parseTableOf` $\mathcal{G}$ = `inr` t', where table t' is extensionally equivalent to t.

In both theorems, the proposition "t is a correct LL(1) parse table for $\mathcal{G}$" means that t contains all and only the lookahead facts about $\mathcal{G}$. It is the mechanized notion of LL(1) parse table correctness from Definition 2.2.1; the only difference is that in the development, we store an entire production and its semantic action in each table cell, rather than just the right-hand side.

In Theorem 2.3.6, the "unique productions" condition says that the grammar contains no duplicate productions. Productions are considered duplicates if they are equal up to their semantic actions—i.e., the "unique productions" definition ignores actions. Duplicate productions always indicate user error; to understand why, consider a grammar with two productions, $X ::= \gamma\ \{f\}$ and $X ::= \gamma\ \{g\}$. If f and g are the same function, then the productions are redundant, and one of them can be removed without affecting the grammar's semantics. If f and g are different, then the grammar is ambiguous; the parser performs a single semantic action upon reducing a production, and it is unclear whether that action should be f or g. Coq functions cannot be compared for equality, so `parseTableOf` cannot determine whether duplicate productions are redundant or ambiguous. The "unique productions" property is decidable, however, so the function checks its input grammar for this property and alerts the user when the check fails. The user can then correct the error in the grammar.

The completeness theorem's conclusion may seem odd; why don't we use this version?

Theorem 2.3.7 (Unprovable `parseTableOf` completeness)**.** If grammar $\mathcal{G}$ has unique productions and t is a correct LL(1) parse table for $\mathcal{G}$, then `parseTableOf` $\mathcal{G}$ = `inr` t.

In the development, a parse table is simply a finite map in which keys are row/column pairs and values are cell contents. We use FMaps, a Coq finite map library, to obtain a map representation and many useful lemmas about map operations. Two maps defined with this library that contain identical entries are not definitionally equal in Coq because they might have different internal representations. Thus, if t is a correct LL(1) parse table for $\mathcal{G}$, we cannot prove that `parseTableOf` returns t itself—only that it returns a table t' containing exactly the same entries as t, which is sufficient for proving the remaining Vermillion correctness properties.

To summarize our progress so far, we have proved that the parse table generator terminates on all inputs, and that it produces a correct LL(1) parse table for its input grammar whenever such a table exists.

2.4 Parsing Algorithm Implementation and Verification

We now turn to the task of implementing and verifying the LL(1) parsing algorithm. The implementation takes the form of a `parse` function that uses an LL(1) parse table `tbl` and a symbol `s` to build a semantic value of type ⟦`s`⟧ for a prefix of the token sequence `ts`:

```
parse (tbl : parse_table) (s : symbol) (ts : list token) :
      sum parse_failure (⟦s⟧ * list token)
```

(In the case of a successful parse, the function also returns the unprocessed suffix of `ts`.) We then wish to verify that as long as the function's LL(1) parse table argument is correct for some grammar, its return value is correct with respect to the grammar's derivation relation. Below are the three main parser correctness properties that we prove:

1. (*Soundness*) - If the parser consumes a token sequence, returning a semantic value v for prefix w and an unparsed suffix r, then $s \xrightarrow{v} w \mid r$ holds.

2. (*Error-Free Termination*) - The parser never reaches an error state when applied to a correct LL(1) parse table.

3. (*Completeness*) - If $s \xrightarrow{v} w \mid r$ holds, then the parser returns v and r when applied to symbol s and token sequence $w +\!\!+ r$.

2.4.1 Parser Structure

Because our parser's correctness specification is the LL(1) derivation relation, it is natural to structure the parser in a way that mirrors the relation's structure. An intuitive way of doing so is to define two mutually recursive functions, `parseSymbol` and `parseGamma`, that respectively consume a symbol and a list of symbols and return a semantic value and a tuple of semantic values. However, a naïve attempt at defining these two functions leads to a violation of Coq's syntactic guardedness condition, which requires all recursive function calls to have a structurally decreasing argument. The termination checker is not being overly conservative—a naïvely defined LL(1) parser might actually fail to terminate on certain inputs! The reason is that our parse tables are simply finite maps, and it is possible to create a map that would cause the functions to diverge. For example, consider the singleton map containing the binding $(X,\ a) \mapsto X$. Applying the parser to this map and a token sequence beginning with *a* would cause it to loop infinitely.

The problem with this table is that it includes a *left-recursive* entry—an entry that leads the parser from nonterminal X back to X without consuming any input. Our parser detects left recursion dynamically by maintaining a set of visited nonterminals that is reset to $\emptyset$ when the parser consumes a token. If the parser reaches a nonterminal that is already present in the visited set, it halts and returns an error value. Our proof of error-free termination shows that the parser never actually returns this "left recursion detected" value as long as it is applied to a correct LL(1) parse table for some grammar, because a grammar that has such a table contains no left recursion.

Of course, left recursion is not the only failure case—the parser could also determine that no input prefix is in the language that it recognizes. In this case, it should provide some information about why it rejected the input. Therefore, our parser returns one of the following values:

- `inr (v, r)`, where `v` is a semantic value for a prefix of the input tokens and remainder `r` is the unparsed suffix, indicating a successful parse.

- `inl (Reject m r)`, where `m` is an error message and remainder `r` is the suffix that the parser was unable to consume.

- `inl (Error m x r)`, where `m` is an error message, `x` is the nonterminal found to be left-recursive, and `r` is the unparsed suffix.

After adding left recursion detection, we still have to convince Coq that `parseSymbol` and `parseGamma` terminate, because their termination metric depends on multiple function parameters. The token sequence decreases structurally in some recursive calls, while in others, the visited set grows larger (and therefore, its complement relative to the universe of grammar nonterminals grows smaller). Coq's `Function` and `Program` commands can often ease the burden of defining functions with subtle termination conditions; both commands enable the user to write a function and then provide its termination proof after the fact. Unfortunately, `Function` and `Program` do not support mutually recursive functions that are defined with a well-founded measure. Therefore, we implement well-founded recursion "by hand," mimicking the process that these commands perform automatically. The process involves the following steps:

1. Define a measure `meas` that maps arguments of `parseSymbol` and `parseGamma` to the following triple of natural numbers:
   - *(First projection)* The length of the token sequence.
   - *(Second projection)* The cardinality of the visited set's complement relative to the set of all grammar nonterminals.
   - *(Third projection)* The size of the function's "symbolic" argument, which is a symbol in the case of `parseSymbol` and a list of symbols in the case of `parseGamma`. We define the size of a symbol to be 0 and the size of a list of symbols γ to be `1 + length` γ. This choice allows `parseGamma` to call `parseSymbol` with an unchanged token sequence and visited set, and it allows `parseGamma` to call itself under the same conditions as long as `length` γ decreases.
2. Define a lexicographic ordering `triple_lt` on triples of natural numbers.
3. Add a proof of the measure value's *accessibility* in the `triple_lt` relation (i.e., a proof that there are no infinite descending chains from the value in `triple_lt`) as an extra function argument.
4. Prove lemmas showing that the size of this accessibility proof decreases on recursive calls. The proof term thus becomes the parser's structurally decreasing argument.
5. Prove that `triple_lt` is well-founded—in other words, that there is no infinite descending chain in `triple_lt` starting from *any* value. This step enables a caller to invoke the parser with any initial set of arguments; it provides an initial accessibility proof for those arguments.

This process yields functions with the signatures that appear in Figure 2.9. In each return type, `{ts' & length_lt_eq ts' ts}` is the dependent pair of a token sequence `ts'` and a proof that `ts'` is either shorter than the function's `ts` argument or definitionally equal to `ts`. In order for

```
parseSymbol (tbl : parse_table)
            (s   : symbol)
            (ts  : list token)
            (vis : NtSet.t)
            (* accessibility proof *)
            (Ha  : Acc triple_lt (meas tbl ts vis (Sym_arg s))) :
            sum parse_failure (⟦s⟧ * {ts' & length_lt_eq ts' ts})

parseGamma (tbl   : parse_table)
           (gamma : list symbol)
           (ts    : list token)
           (vis   : NtSet.t)
           (Ha    : Acc triple_lt (meas tbl ts vis (Gamma_arg gamma))) :
           sum parse_failure (⟦gamma⟧ * {ts' & length_lt_eq ts' ts})
```

Figure 2.9: Signatures of the mutually recursive `parseSymbol` and `parseGamma` functions that underlie the Vermillion LL(1) parsing algorithm implementation.

```
parse (tbl : parse_table) (s : symbol) (ts : list token) :
  sum parse_failure (⟦s⟧ * list token) :=
  match parseSymbol tbl s ts NtSet.empty (triple_lt_wf _) with
  | inl failure => inl failure
  | inr (v, existT _ ts' _) => inr (v, ts')
  end
```

Figure 2.10: Definition of the `parse` function, the top-level interface to the Vermillion LL(1) algorithm implementation. The function passes its arguments to `parseSymbol`, along with an empty set of visited nonterminals and a proof that the measure value for the initial `parseSymbol` arguments is accessible in the well-founded `triple_lt` relation.

each function f to construct an appropriate accessibility proof when it calls itself recursively, it needs to know whether a call to the other function g consumed any input. One way for f to obtain this information is by simply computing the length of the remaining input after the call to g, but this approach would be expensive. By encoding information about changes to the input length in the functions' dependent return types, we avoid computing the length of the remaining token sequence at runtime.

Finally, we define `parse` (shown in Figure 2.10), a top-level interface to the parser that invokes `parseSymbol` with an empty visited set and an appropriate accessibility proof term, and that strips out the return value's dependent component.

2.4.2 Parser Soundness

The first parser correctness property that we prove is soundness with respect to the LL(1) derivation relation. We show that whenever the parser returns a semantic value for a prefix of its input,

the LL(1)ValueDer relation (Figure 2.2) produces the same value for the same prefix:

Theorem 2.4.1 (Soundness of LL(1) algorithm)**.** If table pt is a correct LL(1) parse table for grammar $\mathcal{G}$ and `parse` $pt\ s\ (w +\!\!+ r) = \texttt{inr}(v, r)$, then $s \xrightarrow{v} w \mid r$.

We prove this theorem via a slightly different statement that implies the previous one:

Lemma 2.4.2 (Soundness of `parseSymbol`)**.** If table pt is a correct LL(1) parse table for grammar $\mathcal{G}$ and `parseSymbol` $pt\ s\ \bar{t}\ vi\ Ha = \texttt{inr}(v, (r\ \&\ Hle))$, then there exists a word w such that $w +\!\!+ r = \bar{t}$ and $s \xrightarrow{v} w \mid r$.

The main difference between these two properties is that Theorem 2.4.1 uses the append function to specify exactly how the function divides its input sequence into a parsed prefix and an unparsed suffix. It is difficult to reason directly about this statement because the input can be divided into a prefix and suffix in multiple ways.

The Lemma 2.4.2 proof relies on yet another lemma that generalizes over both `parseSymbol` and `parseGamma`. The proof of this latter lemma proceeds by nested induction on the lexicographic components of the functions' measure. The proof is straightforward by design; we were careful to define `parseSymbol` and `parseGamma` so that the "success" path through the functions' recursive calls mirrors the structure of the derivation relation.

2.4.3 Parser Error-Free Termination

Our next task is to prove that the parser never returns an error value as long as its table argument is a correct LL(1) parse table for some grammar:

Theorem 2.4.3 (Error-free termination of LL(1) algorithm)**.** If pt is a correct LL(1) parse table for $\mathcal{G}$, then `parse` $pt\ s\ w$ never returns `inl (Error` $m\ \ x\ \ w'$`)` for any m, x, w'.

However, it is certainly possible for `parseSymbol` and `parseGamma` to return an error value! For example, they will produce an error when applied to nonterminal X and a visited set that already contains X. To prove the top-level function `parse` safe, we need to specify the conditions that cause the underlying functions to produce an error, and then prove that these conditions do not apply to the top-level call.

One error condition is when the parser is applied to nonterminal X and its visited set already contains a nonterminal that is reachable from X without any input being consumed. We formalize this notion of "null-reachability" in the inductive predicate NullablePath (Figure 2.11). When this predicate holds of two nonterminals X and X', there exists a sequence of steps through the grammar from X to X' in which all symbols visited along the way are nullable.

The second error condition is when the grammar contains a left-recursive nonterminal. Left recursion is just a special case of null-reachability:

Definition 2.4.1 (LL(1) Left recursion)**.** Nonterminal X is left-recursive on lookahead token l when there exists a nullable path on l from X to X—i.e., when NullablePath $l\ X\ X$ holds.

$$\frac{X ::= \alpha Z \beta \; \{f\} \in \mathcal{G} \qquad \textsc{Nullable}(\alpha) \qquad l \in \textsc{Lookahead}(X ::= \alpha Z \beta)}{\textsc{NullablePath}\ l\ X\ Z}\ \textsc{DirectPath}$$

$$\frac{X ::= \alpha Y \beta \; \{f\} \in \mathcal{G} \qquad \textsc{Nullable}(\alpha) \qquad l \in \textsc{Lookahead}(X ::= \alpha Y \beta) \qquad \textsc{NullablePath}\ l\ Y\ Z}{\textsc{NullablePath}\ l\ X\ Z}\ \textsc{IndirectPath}$$

Figure 2.11: The NullablePath predicate holds for nonterminals X and Z when Z is reachable from X via a series of LL(1) parser steps that do not consume any input.

Note that a standard definition of left recursion would omit the lookahead component. Below, we prove that the existence of a correct LL(1) parse for grammar $\mathcal{G}$ implies that $\mathcal{G}$ contains no left recursion. However, the existence of such a table only rules out "LL(1) left recursion" as it is defined above. It does not rule out "harmless" left recursion—i.e., left recursion that can never cause the parsing algorithm to diverge because it has no lookahead information associated with it and is therefore unreachable via parse table lookups.

We prove a lemma stating that when `parseSymbol` or `parseGamma` returns an error value, one or both of the error conditions described above are present. The first condition does not apply to `parse` because the top-level function calls `parseSymbol` with an empty visited set. To prove that the second condition does not apply, we show that a grammar with a correct LL(1) parse table contains no left recursion. Although standard references mention this property in passing, we could not find a rigorous proof in the literature. Our proof involves a fair amount of machinery; it consists of the following steps:

1. We define *sized* versions of the Nullable (Figure 2.3) and First (Figure 2.4) relations. These versions include a natural number representing the proof term's size.
2. We prove that these sizes are deterministic for an LL(1) grammar—any two proofs of the same Nullable or First fact have the same size.
3. We show that if grammar $\mathcal{G}$ contains a left-recursive nonterminal, then there are two proofs of the same Nullable or First fact about $\mathcal{G}$ with *different* sizes.

These steps enable us to prove the lemma below by obtaining a contradiction from facts 2 and 3:

Lemma 2.4.4 (LL(1) parse table precludes left recursion)**.** If pt is a correct LL(1) parse table for $\mathcal{G}$, then nonterminal X is not left-recursive on lookahead token l for any X, l.

2.4.4 Parser Completeness

Finally, we prove that our parser is *complete*—if a grammar symbol derives a semantic value for a prefix of a token sequence, then the parser produces the same value for the same prefix:

Theorem 2.4.5 (Completeness of LL(1) algorithm)**.** If pt is a correct LL(1) parse table for $\mathcal{G}$ and $s \xrightarrow{v} w \mid r$, then `parse` $pt\ s\ (w +\!\!+ r) = \texttt{inr}(v, r)$.

Our error-free termination result simplifies the task of proving completeness. We begin by proving a more general lemma stating that when a grammar derivation exists, the parser either returns an error or produces the semantic value from the derivation:

Lemma 2.4.6 (`parseSymbol` does not reject valid input)**.** If pt is a correct LL(1) parse table for $\mathcal{G}$ and $s \xrightarrow{v} w \mid r$, then the following disjunction holds:

(1) `parseSymbol` $pt\ s\ (w +\!\!+ r)\ vi\ Ha = \texttt{inl (Error}\ m\ X\ r'\texttt{)}$ for some m, X, r'

or

(2) `parseSymbol` $pt\ s\ (w +\!\!+ r)\ vi\ Ha = \texttt{inr}\ (v, (r\ \&\ Hle))$
(where Hle is a proof that r is shorter than or propositionally equal to $w +\!\!+ r$)

We prove this lemma by induction on the derivation relation, use the error-free termination theorem (Theorem 2.4.3) to rule out disjunct (1), and use disjunct (2) to prove the completeness theorem itself.

This section completes our discussion of the top-level `parse` function's correctness properties. We have shown that when it is supplied with a correct LL(1) parse table for grammar $\mathcal{G}$, our implementation of the LL(1) algorithm is sound and complete with respect to the derivation relation for $\mathcal{G}$, and that the algorithm terminates without error.

2.5 Performance Evaluation

To evaluate the efficiency of a Vermillion parser, we extracted Vermillion to OCaml source code and used it to produce an LL(1) parser for the JSON data format. We also used Menhir, a popular OCaml LR(1) parser generator, to produce an unverified parser for the same grammar and compared the two parsers' performance on a JSON data set. The goals of the evaluation were to assess Vermillion's performance relative to an OCaml parsing tool that is used in practice, and to confirm that our LL(1) implementation lives up to the algorithm's theoretical guarantee of linear-time execution.

We based our Menhir lexer[1] and grammar on the ones described in the *Real World OCaml* textbook's tutorial on JSON parsing [Minsky et al. 2013]. We then replicated the grammar in Vermillion's input format. Because our tool consumes a list of tokens, we used Menhir to generate a second parser that acts as a preprocessor for Vermillion—it simply tokenizes an entire JSON string. In our evaluation, we count this tokenizer's execution time as part of the LL(1) parser's total execution time. (In Chapter 4, we describe a more principled approach to integrating a verified parser with a lexer—in that case, a lexer that is also verified.)

We ran both JSON parsers on a small data set, averaging the execution times of ten trials for each data point. The results appear in Figure 2.12. The Vermillion parser is between two and

[1]The lexer does not support Unicode escape sequences because they do not appear in the test data, but nothing prevents Vermillion or Menhir from handling Unicode tokens in principle.

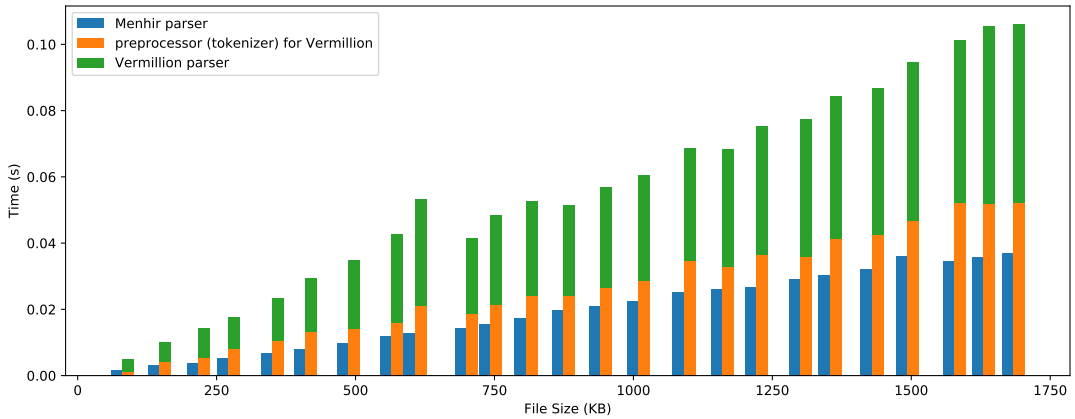


Figure 2.12: Average execution times of Menhir and Vermillion JSON parsers.

four times slower than the unverified Menhir parser on each data point. This comparison is not entirely scientific, because Menhir and Vermillion use two different parsing algorithms—LR(1) and LL(1), respectively. Nevertheless, it suggests that Vermillion's performance is reasonable, given that it was designed with ease of verification (rather than optimal performance) in mind. Other certified parsers obtain similar performance results; a validated LR(1) parser [Jourdan et al. 2012] runs about five times slower than its unvalidated counterpart, and a verified PEG interpreter [Koprowski and Binsztok 2010] is two to three times slower than an unverified version.

As an interesting side note, when we first extracted Vermillion to OCaml, we discovered that its performance was superlinear! This earlier version of the parser periodically computed the length of the remaining input to determine whether a previous recursive call had consumed any tokens, and thus whether it was safe to empty the set of visited nonterminals. With some refactoring, we were able to lift this reasoning about input length into the proof component of the parser's dependent return type, ensuring that it is erased at extraction time and therefore does not hamper performance.

2.6 Related Work

Subsequent to the development of Vermillion, Edelmann et al. [2020] present a derivative-based LL(1) parsing algorithm, which they verify in Coq and reimplement in a Scala parser combinator framework. Parsing with derivatives [Might et al. 2011] is an elegant formalism that is compatible with arbitrary CFGs and has cubic worst-case time complexity [Adams et al. 2016]. Edelmann et al. show that restricting derivative-based parsing to LL(1) grammars leads to linear-time performance.

In interesting recent work that has not been formally verified, Krishnaswami and Yallop [2019] give a language of parser combinators for building context-free expressions, and a type system that ensures that any well-typed expression built with these combinators is LL(1). They then use staging and metaprogramming techniques to transform combinator-defined parsers into a highly performant representation.

2.7 Conclusion

We have verified that Vermillion produces a sound and complete LL(1) parser for its input grammar whenever such a parser exists, and that the resulting parsers terminate on valid and invalid inputs without using fuel. Below, we discuss two possible extensions of this work: ruling out parser errors *a priori* and generating parser source code.

Our parsing algorithm implementation includes branches that represent error states. These branches survive the extraction process and slow down the resulting code, even though we prove that the algorithm never reaches them when applied to a correct LL(1) parse table. An anonymous reviewer made a useful analogy between the parser and an interpreter that checks for type errors dynamically, even when a static type system ensures that a valid input program never triggers these errors—i.e., that "well-typed programs cannot 'go wrong' " [Milner 1978]. The reviewer also noted that it might be possible to remove these branches from the parser by making it a function over correct LL(1) parse tables instead of simply-typed tables, just as one can remove dynamic type-checking from an interpreter by parameterizing it with typing derivations instead of raw terms. We chose to rule out errors after the fact because it is often simpler to separate the concerns of programming and proving, but the *a priori* approach would be more elegant to some observers and certainly more efficient.

Our parsers represent tables as finite maps and perform map lookups at decision points, which is a likely source of inefficiency. Some production-grade parser generators produce source code that is specialized to their input grammar. These parsers represent table lookups with source-level constructs (e.g., `match` expressions) instead of data structure operations. Generated parser code is likely to be more efficient than a table-based interpreter; for example, Menhir enables the user to choose between these two representations, and an informal benchmark finds that code generation produces parsers that are two to five times faster than their table-based counterparts [Pottier and Régis-Gianas 2016]. One could develop a version of our tool that generates abstract syntax for a language with mechanized semantics, such as Clight [Blazy and Leroy 2009], and verify that the abstract syntax representation of a parser is extensionally equivalent to a table-based parser for the same grammar.

3

CoStar: A Verified ALL(*) Parser Interpreter

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 chapter, we present CoStar, a verified parser interpreter that makes an attractive set of tradeoffs along these three dimensions. The tool 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.

3.1 Introduction

Vermillion, described in the previous chapter, is compatible with LL(1) grammars. Not every language has an LL(1) grammar, and even those that do might have a non-LL(1) specification that is more natural—i.e., easier for human beings to read and write. All else being equal, parsing tools should support the full range of grammars that users wish to define. One of the main arguments for using a general-purpose parsing tool is that the user gets to specify the language they wish to parse in a declarative way. When the user must contort their grammar to suit the tool's underlying parsing algorithm, the tasks of grammar writing, validating, and debugging become more difficult.

Unfortunately, all else isn't equal, and parsers that support larger classes of grammars come with costs in other areas, such as correctness guarantees and performance. In terms of verified parsers for context-free grammars (CFGs), the limitations of prior work fall chiefly into three categories. Top-down predictive parsers [Lasser et al. 2019a; Edelmann et al. 2020] are limited in their *expressiveness*; they are only compatible with the restricted LL(1) class of grammars. Bottom-

up parsers [Barthwal and Norrish 2009; Jourdan et al. 2012] 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 [Danielsson 2010; Ridge 2011; Firsov and Uustalu 2014, 2015] 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 chapter, we present CoStar, a verified parser interpreter that addresses these limitations. Our interpreter is based on the ALL(*) parsing algorithm, which forms the core of the popular ANTLR 4 parser generator [Parr et al. 2014]. 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 many programming languages and data formats [Parr et al. 2014], and the popularity of ANTLR suggests that ALL(*) is well-suited to a wide range of applications.

This chapter makes the following contributions:

- We present CoStar, an ALL(*) parser interpreter implemented and verified with the Coq Proof Assistant [Coq Development Team 2020].
- We give proofs of CoStar's soundness, error-free termination, and completeness for all CFGs without left recursion. The interpreter 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.

The chapter is also a case study on implementing a small-step interpreter with a big-step correctness specification. Because the interpreter has a complex termination proof, it is convenient to factor the implementation into a "small step" function that performs a single atomic update to the interpreter's state (Section 3.3.3) and a loop that chains small steps together (Section 3.4). This design leads to a clean separation of concerns; we are able to isolate the loop's thorny termination proof from the logic of state updates, resulting in an implementation that resembles what a programmer might write in a language with unguarded recursion. However, the standard correctness specification for parsing algorithms (Section 3.5.1) has a big-step flavor. To bridge the gap between small-step implementation and big-step specification, we employ an elegant invariant-based style of verification: for each big-step correctness property, we define an invariant over the interpreter's state that entails the property. This approach shifts the verification burden to the task of proving that each interpreter step preserves the invariant.

The definitions and theorems in this chapter have been mechanized in Coq. The development consists of roughly 5,000 lines of specification and 5,000 lines of proof. This develop-

ment and an accompanying performance evaluation framework are available online as an artifact [Lasser et al. 2021a] that we submitted to the Programming Language Design and Implementation (PLDI) 2021 conference along with the paper on which this chapter is based. The artifact underwent PLDI's artifact evaluation process, earning the "Available," "Functional," and "Reusable" badges. The development and evaluation framework are also available via a public GitHub repository [Lasser et al. 2021b].

The chapter is organized as follows. In §3.2, we briefly introduce ALL(*) parsing. We outline the CoStar implementation in §3.3. We present its termination properties in §3.4 and correctness in §3.5. In §3.6, we give performance evaluation results. We offer conclusions and possible future directions for this work in §3.7.

3.2 Overview of ALL(*) Parsing

The ALL(*) algorithm was introduced by Parr et al. [2014]. It shares its high-level structure with LL(k) [Lewis and Stearns 1968] and LL(*) [Parr and Fisher 2011]. 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. [2014] 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. The authors also report that disabling the cache hampers performance significantly, which suggests that caching is indeed responsible for the algorithm's strong empirical performance results.

Basic definitions

Terminals	$a, b \in \mathcal{T}$
Nonterminals	$X, Y \in \mathcal{N}$
Symbols	$s ::= a \mid X$
Sentential Forms	$\alpha, \beta, \gamma ::= \bullet \mid s\beta$
Grammars	$\mathcal{G} ::= \bullet \mid X ::= \gamma,\ \mathcal{G}$
Literals	$l \in$ *string*
Tokens	$t ::= (a, l)$
Words	$w ::= \epsilon \mid tw$
Trees	$v ::= \texttt{Leaf}(t) \mid \texttt{Node}(X, f)$
Forests	$f ::= \bullet \mid v, f$

CoStar definitions

Prefix Stacks	$\Phi ::= \bullet \mid [\alpha, f]\Phi$
Suffix Stacks	$\Psi ::= \bullet \mid [\beta]\Psi$
Subparsers	$\theta ::= (\gamma, \Psi)$
DFA States	$q ::= \bullet \mid \theta, q$
DFAs	$\Delta ::= \bullet \mid (q,\ a) \mapsto q, \Delta$
Machine States	$\sigma \in \Phi \times \Psi \times \Delta \times w \times \mathcal{S}(\mathcal{N}) \times \mathbb{B}$
Errors	$e ::= \texttt{InvalidState} \mid \texttt{LeftRecursive}(X)$
Predictions	$p ::= \texttt{UniqueP}(\gamma) \mid \texttt{AmbigP}(\gamma) \mid \texttt{RejectP} \mid \texttt{ErrorP}(e)$
Step Results	$r ::= \texttt{AcceptS}(v) \mid \texttt{RejectS} \mid \texttt{ErrorS}(e) \mid \texttt{ContS}(\sigma)$
Parse Results	$R ::= \texttt{Unique}(v) \mid \texttt{Ambig}(v) \mid \texttt{Reject} \mid \texttt{Error}(e)$

Figure 3.1: Core definitions and notations used throughout this chapter. 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 sometimes 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.

3.3 A Verifiable ALL(*) Implementation

In this section, we describe the structure of CoStar, focusing on aspects of the tool's design that make it amenable to reasoning about its termination and correctness. Figure 3.1 contains key CoStar definitions that we discuss in this section and throughout the chapter.

3.3.1 Top-Level API

The entry point to CoStar is the `parse` function, which takes a grammar $\mathcal{G}$, 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 $w \notin \mathcal{L}(\mathcal{G})$.

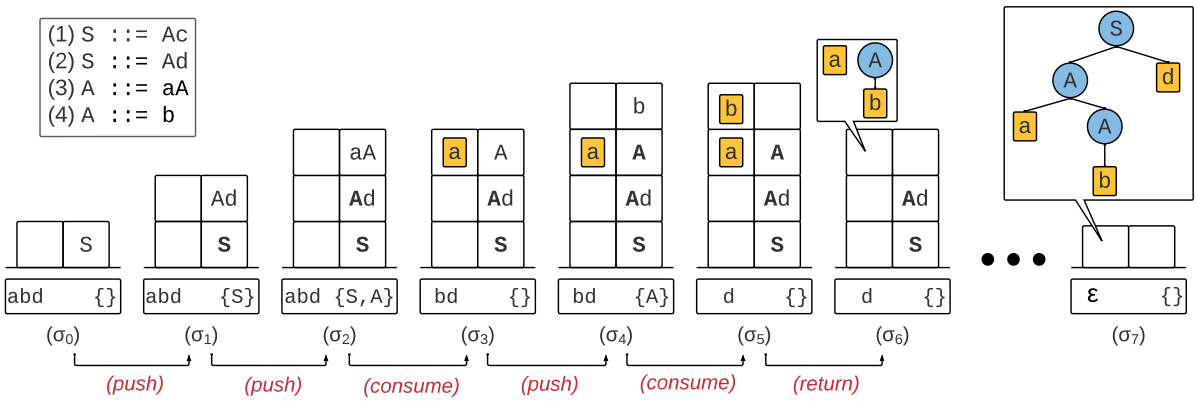


Figure 3.2: Trace of the CoStar stack machine's execution on a simple grammar and token sequence. Each point in the trace depicts the state of the machine's prefix stack, suffix stack, remaining tokens (represented as terminals to simplify the presentation), and visited nonterminals. For example, at 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).

- An `Error` value indicating that the interpreter reached an inconsistent state. (We later prove that this result never occurs when $\mathcal{G}$ is non-left-recursive.)

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 function `step` : $\sigma \to r$ that performs a single atomic update to the state; and (3) a function `multistep` : $\sigma \to R$ that repeatedly calls `step` until the state reaches one of the final values listed above. Figure 3.2 depicts a trace of the machine's execution on toy input; we use this figure throughout the chapter 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 3.4.

3.3.2 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 interpreter's final return value. At state (σ_6) in Figure 3.2, 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 3.2, the two suffix stack frames contain the unprocessed symbols `Ad` and `S`, respectively, and the top stack symbol is `A`.

  Note that the prefix stack and suffix stack together are equivalent to a single stack of pairs. We maintain two separate stacks for efficiency. The CoStar prediction subroutine does not consider the prefix stack; by maintaining two stacks, we avoid needing to project out the suffix stack each time CoStar launches the prediction subroutine at a decision point.

- A **cache** that stores the results of previous prediction steps for later reuse (details appear in Section 3.3.4).

- The remaining sequence of **tokens** to parse. At state (σ_3) in Figure 3.2, the remaining token sequence is `bd`.

- 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 3.4.1). At state (σ_2) in Figure 3.2, the visited set is `{S,A}`.

- A **uniqueness flag** indicating whether the machine has detected that the input word is ambiguous. Figure 3.2 does not show this component of the state because the input is unambiguous, so the flag remains `true` throughout the parse.

3.3.3 Single-Step Machine Operations

The `step` function examines the current machine state σ and produces a new state σ' by performing one of the following operations (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 3.2, the $(\sigma_2) \rightsquigarrow (\sigma_3)$ transition is a consume operation. The machine's top stack terminal `a` matches the terminal of the next token, so the machine adds a leaf node 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 grammar 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 3.2, 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 3.2, 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. While rejecting invalid input is expected behavior, an error indicates that one of the machine's internal invariants was broken. In Section 3.5.4, we prove that such errors never arise when CoStar is supplied with a left-recursion-free grammar.

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.

3.3.4 Prediction Mechanism

The distinguishing feature of ALL(*) is its prediction algorithm, `adaptivePredict`. 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 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 well-suited to memoization and is therefore faster. It uses a DFA to cache the results of the operations it performs, thus ensuring that no step of the nondeterministic simulation needs to be computed more than once. A DFA entry $(q, a) \mapsto q'$ represents an edge labeled with terminal a from DFA state q to state q'. The existence of such an edge indicates that during a previous round of SLL prediction, the collection of subparsers q transitioned to a new configuration q' upon consuming terminal a. For efficiency, `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 prediction results have the same type, but a result's meaning can differ based on the mode that produced it. LL predictions have the following interpretations:

- `UniqueP(`γ`)`: γ is the only right-hand side that may lead to a successful parse.
- `AmbigP(`γ`)`: 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.
- `RejectP`: No right-hand side leads to a successful parse.
- `ErrorP(`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 `AmbigP(`γ`)` 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 interpreter from taking a "wrong turn" by pushing γ onto the stack.

LL and SLL prediction share some infrastructure. LL is simpler, so we focus on the CoStar implementation of LL prediction in Section 3.3.4.1 and briefly discuss the differences between the LL and SLL implementations in Section 3.3.4.2.

3.3.4.1 LL Prediction Functions

Here, we sketch the execution of the LL prediction functions (function names resemble those in the original ALL(*) paper [Parr et al. 2014] whenever an analogue exists):

- Entry point `llPredict` calls the `llStartState` function to create an initial collection of subparsers and then passes them to the iteration function `llPredict'`.
- Given an initial suffix stack with top nonterminal X, the `llStartState` function initializes one subparser for each right-hand side for X (i.e., each possible prediction).
- The `llPredict'` function advances the subparsers in lockstep, consuming one input token at a time until it reaches one of the following outcomes:
  - All subparsers die off, indicating that no right-hand side leads to a successful parse. The function represents this outcome with a `RejectP` return value.
  - All remaining subparsers carry the same right-hand side γ, indicating that γ is the only viable choice. In this case, the function returns a `UniqueP(`γ`)` value.
  - Multiple subparsers that carry different right-hand sides $\{\gamma, \gamma', \dots\}$ reach the end of the input, indicating ambiguity. An `AmbigP(`γ`)` result represents this outcome.
  - One or more subparsers reach an error state, resulting in an `ErrorP(`e`)` value.

  In the case where `llPredict'` has not yet reached a final outcome, the `llTarget` function advances each subparser by means of a combined move/closure operation. In a move operation, a subparser consumes a single token, or dies off if its top stack symbol does not match

the token. In a closure operation, a subparser performs push and return operations until either (1) its next stack symbol is a terminal, or (2) its stack is empty. We call these states "stable" because a subparser in either state cannot perform any further operations until the next `llPredict'` iteration.

3.3.4.2 SLL Prediction Functions

SLL prediction closely resembles LL prediction in its high-level structure. SLL prediction differs in the following ways:

- The SLL algorithm maintains a DFA cache. Before performing a move/closure operation with subparsers q and terminal a, `sllPredict'` checks whether the DFA contains an edge labeled with a from q to some state q', written $(q, a) \mapsto q'$. If so, `sllPredict'` uses q' in the next iteration instead of computing q' via a call to `sllTarget`.
- When `sllPredict'` must compute a target state q' for current state q and terminal a, it adds the transition $(q, a) \mapsto q'$ to the DFA. When `sllPredict'` terminates, it returns the updated DFA along with a prediction.
- The `sllStartState` function ignores the state of the parser stack. Each initial SLL subparser stack consists of a single frame that holds a right-hand side for X.
- When a move or closure operation results in an empty subparser stack, the SLL closure algorithm must simulate a return to all possible caller frames. This simulated return is what makes SLL an overapproximation of LL. An LL subparser has access to the full parser stack, so it returns to the actual caller frame in this situation.

SLL prediction takes advantage of the fact that many decisions are not stack-sensitive: they involve the same steps even when they begin in different initial machine states. Therefore, the cache typically grows quickly, and after the SLL algorithm makes a small number of predictions, it can take most steps by performing cache lookups instead of expensive `sllTarget` operations.

3.3.5 Aside: The Benefits of a Reified Stack Machine

Our choice of an explicit stack machine representation for CoStar is somewhat unconventional. Perhaps a more common approach to implementing top-down parsers is to use mutually recursive functions. For example, the implementation of a grammar-specific parser might use one function per nonterminal [Appel 1998; Parr and Fisher 2011; Parr et al. 2014]. The Vermillion parsing algorithm implementation (Section 2.4) consists of two mutually recursive functions: one for individual symbols and one for right-hand sides. In a mutually recursive setting, the function call stack represents the parser stack. We found that such an approach is not a good fit for a functional, verifiable ALL(*) implementation. In some cases, `adaptivePredict` must examine unprocessed stack symbols beyond the decision nonterminal in order to reach a prediction (these decisions are called "stack-sensitive"). In an approach based on mutual recursion, each function must thread the remaining stack symbols through recursive calls for use in later prediction steps. This approach

quickly becomes messy; we found that reifying the parser stack as an explicit data structure leads to smoother verification. With our approach, the full parser stack remains in scope at each step of the machine's execution, which makes it easier to define invariants over this data structure.

3.3.6 Is It Really ALL(*)?

ALL(*) as originally presented by Parr et al. [2014], 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) [Woods 1970], 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) [Scott and Johnstone 2010] 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 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. [2014] 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; as we discuss in Section 3.5, CoStar provides a stronger correctness guarantee for unambiguous inputs than for ambiguous ones. 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. Each stable return frame corresponds to the top frame of a subparser stack that a sequence of closure operations would have produced.

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.

3.4 Termination

One of the primary challenges of implementing CoStar was proving that the interpreter 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 interpreter 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.

3.4.1 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 [Parr et al. 2014]. 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 is based on our scheme for ensuring that Vermillion terminates (described in Section 2.4.1).

The CoStar 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 3.2, 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 3.5.4, we prove that such an error case never arises when the interpreter is applied to a non-left-recursive grammar.

3.4.2 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.[1]

However, even with dynamic left-recursion detection, there is no obvious candidate for a `multistep` decreasing measure. Figure 3.2 illustrates the way in which the interpreter 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 3.4.3) 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 $\texttt{meas} : \sigma \to \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ that maps a machine state to the measure described above.

2. Add a proof of the measure's accessibility[2] in $<_3$ as a parameter to `multistep` (its type becomes dependent). We write $\text{Acc}_{<_3}(\texttt{meas}(\sigma))$ for this parameter.

3. Prove that each machine step preserves the measure's accessibility (Lemma 3.4.1). Thanks to the Curry-Howard Isomorphism, we can view this lemma as a function from a proof term $\text{Acc}_{<_3}(\texttt{meas}(\sigma))$ to a smaller term $\text{Acc}_{<_3}(\texttt{meas}(\sigma'))$. The accessibility proof thus becomes `multistep`'s structurally decreasing parameter.

Lemma 3.4.1 (Preservation of $\text{Acc}_{<_3}$)**.** If $\text{Acc}_{<_3}(\texttt{meas}(\sigma))$ and $\sigma \rightsquigarrow \sigma'$, then $\text{Acc}_{<_3}(\texttt{meas}(\sigma'))$.

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 3.4.2). □

Lemma 3.4.2 (Steps Decrease Measure)**.** If $\sigma \rightsquigarrow \sigma'$, then $\texttt{meas}(\sigma') <_3 \texttt{meas}(\sigma)$.

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 3.4.3).

[1] 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.

[2] An element x is accessible in $<$ when every element $y < x$ is also accessible.

- **return** case: In a return operation, the number of tokens remains constant, the stack score either (a) decreases or (b) remains constant (Lemma 3.4.4), and the stack height decreases. Therefore, the overall triple decreases due to a decrease in (a) its second position or (b) its third position.

□

3.4.3 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 interpreter 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 that it 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):

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

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:

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

Finally, the `stackScore` function calls `stackScore'` with a base of (1 + the maximum 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:

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

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 3.4.3. If the machine pushes from Ψ and V to Ψ' and V', then $\texttt{stackScore}(\mathcal{G}, \Psi', V') < \texttt{stackScore}(\mathcal{G}, \Psi, V)$.

Lemma 3.4.4. If the machine returns from Ψ and V to Ψ' and V', then $\texttt{stackScore}(\mathcal{G}, \Psi', V') \leq \texttt{stackScore}(\mathcal{G}, \Psi, V)$.

3.4.4 A Provably Terminating Prediction Mechanism

In both LL and SLL modes, ALL(*) prediction simulates parsing; each subparser mimics the behavior of the top-level stack machine for that subparser's choice of right-hand side. However, the prediction mechanism groups operations differently than the top-level machine. A subparser performs multiple closure operations (pushes and returns) as a single logical step so that each resulting subparser is stable—i.e., ready to consume a token at the start of the next prediction step. Therefore, the termination argument for the prediction mechanism is slightly different from the `multistep` argument. Both `llPredict'` and `sllPredict'` are structurally recursive on the remaining token sequence, and a move operation is not recursive—a subparser either consumes a single token successfully or dies off. Only the closure function demands an explicit termination proof. The well-founded measure for a closure operation on subparser θ corresponds to the second and third components of the `multistep` measure. In other words, it is a lexicographic pair consisting of (1) a `stackScore` value for a visited set and θ, and (2) θ's stack height. This correspondence enables us to reuse lemmas that describe how return and push operations cause these values to decrease, sparing us some repetitive proof work.

3.5 Correctness Properties

A novel feature of the CoStar correctness analysis is that it encompasses ambiguity detection. We want to ensure that the tool correctly labels trees as unique or ambiguous.[3] 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 interpreter returns a tree v labeled as `Unique`, then v is the sole parse tree for the input.

2. (*Soundness, ambiguous derivations*): If the interpreter returns a tree v labeled as `Ambig`, then v is one of multiple distinct parse trees for the input.

3. (*Error-free termination*): The interpreter 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 interpreter returns v and labels it as `Unique`.

5. (*Completeness, ambiguous derivations*): Given an input word with multiple distinct parse trees, the interpreter returns one of these trees and labels it as `Ambig`.

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

[3] 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."

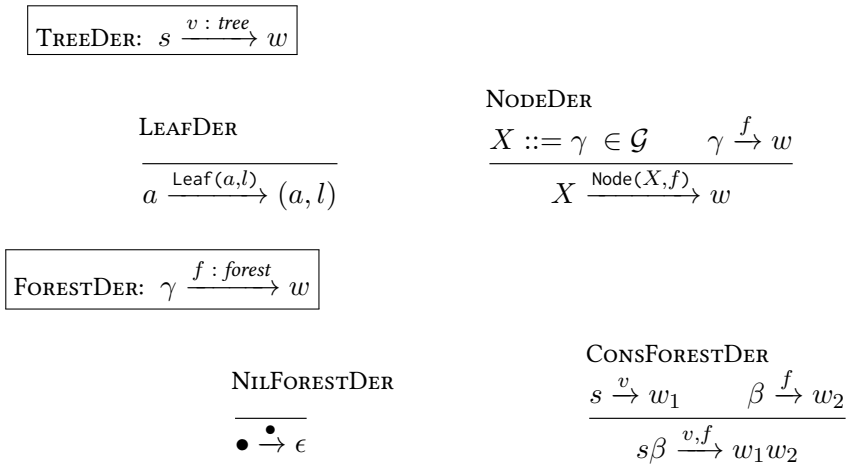


Figure 3.3: Derivation relations for symbols and sentential forms with respect to a grammar $\mathcal{G}$.

3.5.1 Correctness Specification

The CoStar correctness specification is a standard grammatical derivation relation over a grammar symbol s, a word w, and a tree v. The relation, which we call TreeDer, appears in Figure 3.3. It 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 called ForestDer (also in Figure 3.3) 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"). Interpreter soundness and completeness are defined in terms of these two relations.

Notice that TreeDer and ForestDer are big-step relations. For example, the NodeDer rule says that the left-hand side of a production fully derives a word w when the right-hand side fully derives w (in any number of derivation steps).

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."

3.5.2 Soundness for Unique Derivations

Theorem 3.5.1 (Soundness, unique derivations)**.** If `parse` applied to grammar $\mathcal{G}$, nonterminal S, and word w returns a tree v labeled as `Unique`, then v is the sole parse tree rooted at S for w.

The proof of this theorem relies on a more general lemma about the interpreter'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 machine state (described below). The proof is by induction on the well-founded measure for `multistep` (Section 3.4.2). 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 3.5.2.1 and 3.5.2.2, we describe the two invariants in more detail.

$\boxed{\Phi \approx_{\mathcal{G}} \Psi}$

$$\textsc{WfInit}\ \frac{}{[\bullet, \bullet] \approx_{\mathcal{G}} [S]} \qquad \textsc{WfFinal}\ \frac{}{[S, v] \approx_{\mathcal{G}} [\bullet]} \qquad \textsc{WfUpper}\ \frac{[\alpha_1, f_1]\Phi \approx_{\mathcal{G}} [X\beta_1]\Psi \qquad X ::= \alpha_2\beta_2 \in \mathcal{G}}{[\alpha_2, f_2][\alpha_1, f_1]\Phi \approx_{\mathcal{G}} [\beta_2][X\beta_1]\Psi}$$

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

$$\frac{\Phi \approx_{\mathcal{G}} \Psi}{\textsc{StacksWf\_I}(\Phi, \Psi, \_, \_, \_, \_)}$$

Figure 3.4: A well-formedness invariant for the prefix stack and suffix stack components of a machine state σ.

3.5.2.1 Stack Well-Formedness

The StacksWf_I invariant (Figure 3.4) captures two well-formedness properties of the machine'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 3.5.2 (Preservation of Stack Well-Formedness)**.** If StacksWf_I(σ) and $\sigma \rightsquigarrow \sigma'$, then StacksWf_I($\sigma'$).

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 $X ::= \gamma \in \mathcal{G}$. 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. □

3.5.2.2 Invariant for Unique Partial Derivations

The UniqueDer_I invariant (Figure 3.5) 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 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'.

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

UniqueBot

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

UniqueUpper

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

$\boxed{\textsc{UniqueDer\_I}(\sigma, w)}$

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

Figure 3.5: 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 $w = w_1 w_2$." The `unproc` function extracts the unprocessed symbols from a suffix stack and flattens them into a list.

- The UniqueUpper rule captures the same properties as UniqueBot for upper frames. The key third premise captures the additional fact 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.)

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

Lemma 3.5.3 (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 3.2). To ensure that the partial tree remains unique after this operation, the invariant needs to "remember" that f was produced by a uniquely 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 3.5.4): 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 3.5.4. Assume that `adaptivePredict` returns the prediction `UniqueP(`γ`)` 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 `UniqueP(`γ`)`, 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 3.5.5 (below) about the correctness of LL prediction. In case (2), we can use Lemma 3.5.5 directly. □

Lemma 3.5.5. Assume (1) LL prediction returns `UniqueP(`γ`)` for nonterminal X and token sequence w, and (2) some right-hand side γ' for X, 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 $w = w_1 w_2$. 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 actually 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 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 we 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. □

3.5.3 Soundness for Ambiguous Derivations

Theorem 3.5.6 (Soundness, ambiguous derivations)**.** If `parse` applied to grammar $\mathcal{G}$, 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 3.5.1, 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. This congruence rule enables a partial derivation to be ambiguous even when some pushes are unambiguous.

Figure 3.6 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 3.5.7 (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 3.5.3 and this one is that the 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 machine's execution.

Therefore, the tricky case in the proof of Lemma 3.5.7 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

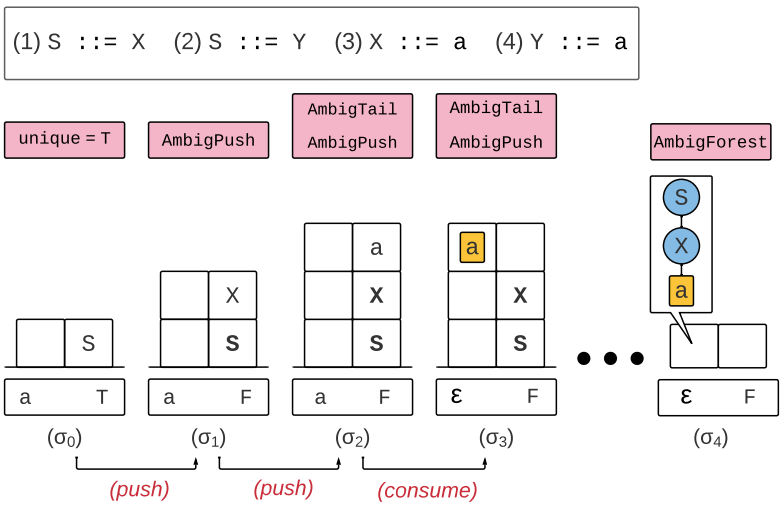


Figure 3.6: Machine execution trace on a simple ambiguous grammar and input word. The cases of the AmbigDer_I invariant that hold for each machine state appear above the state in pink. We replace the visited sets from Figure 3.2 with `unique` flags (written `T`/`F` for brevity) because they are more relevant to this example.

lemma about `adaptivePredict`: if the function returns `AmbigP(`γ`)` 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. □

3.5.4 Error-Free Termination

Theorem 3.5.8. The `parse` function never returns an `Error` value.

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

Interpreter 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 interpreter 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 lemmas and using them to rule out both top-level parsing errors and prediction errors.

The StacksWf_I invariant (Figure 3.4) is sufficient to rule out `InvalidState` interpreter errors. The interpreter 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.

3.5.4.1 Ruling Out Left Recursion Errors

Lemma 3.5.9. Given a non-left-recursive grammar $\mathcal{G}$, the interpreter never raises an error that identifies nonterminal X as left-recursive in $\mathcal{G}$.

Proof. We use an invariant-based version of the approach that we used to rule out similar errors in VERMILLION (see Section 2.4.3). The approach involves proving that the dynamic mechanism for detecting left recursion is sound—i.e., if the interpreter returns `LeftRecursive(`X`)`, then nonterminal X really is left-recursive in the grammar (Lemma 3.5.10 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. □

3.5.4.2 Soundness of Left-Recursion Detection

Lemma 3.5.10. If the interpreter applied to $\mathcal{G}$ returns `LeftRecursive(`X`)`, then nonterminal X is left-recursive in $\mathcal{G}$.

Proof. The proof centers on an invariant that relates the shape of the machine's suffix stack to the shape of the grammar, via the visited set V. The invariant 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.[4] In Figure 3.2 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` $\rightarrow$ `A` $\rightarrow$ `a` because the sequence of transitions $(\sigma_0) \rightsquigarrow (\sigma_1) \rightsquigarrow (\sigma_2)$ did not consume any tokens. The interpreter 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! □

3.5.5 Completeness

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

[4] A nullable path is a path between two positions in the grammar that does not include any terminals. It corresponds to a sequence of machine operations that does not consume any tokens. In Section 2.4.3, we give a formal definition of a nullable path, and we define left recursion as a special case of a nullable path. The CoStar nullable path definition is similar, but it omits the LL(1)-specific lookahead component.

UnprocRecognize_I(σ)

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

Figure 3.7: A machine state invariant for proving completeness: the unprocessed suffix stack symbols recognize the remaining suffix of the input word.

Theorem 3.5.11 (Completeness, unique derivations)**.** If word w has a unique parse tree v rooted at nonterminal S, then the interpreter applied to S and w produces v and labels it as `Unique`.

Theorem 3.5.12 (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 interpreter applied to S and w returns a correct tree v'' and labels it as `Ambig`.

Theorem 3.5.12 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 3.5.11 and 3.5.12, it suffices to prove the weaker statement that when a correct parse tree v exists, the interpreter produces *any* tree v' (Lemma 3.5.13 below). We can then marshal our interpreter soundness theorems and a little first-order reasoning to show that v' is a correct tree, and that the interpreter correctly labels it as `Unique` (Theorem 3.5.11) or `Ambig` (Theorem 3.5.12). In the remainder of this section, we focus on this more general version of completeness.

Lemma 3.5.13 (Completeness, general case)**.** If a correct S-rooted tree v exists for word w, then the interpreter applied to S and w returns a tree v'.

The proof of Lemma 3.5.13 follows this informal reasoning: if the interpreter does not (1) return errors or (2) reject valid input words, then it is complete. Theorem 3.5.8, the 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 3.7) 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 interpreter never rejects its input as invalid. The tricky part is showing that each interpreter step preserves the invariant:

Lemma 3.5.14. If UnprocRecognize_I(σ) and $\sigma \rightsquigarrow \sigma'$, then UnprocRecognize_I(σ').

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 interpreter to take a "wrong turn"; i.e., `adaptivePredict`

never returns a right-hand side for top stack symbol X that causes the interpreter to reject the input when some other right-hand side would have led to a successful parse.

In the "unique prediction" case, `adaptivePredict` returns `UniqueP(`γ`)`, 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 interpreter astray by selecting γ? We reuse Lemma 3.5.4, which says that if `adaptivePredict` returns `UniqueP(`γ`)`, 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 `AmbigP(`γ`)`, 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. □

3.6 Performance Evaluation

To evaluate CoStar's performance, we extracted the tool to OCaml source code and measured its execution time on four benchmarks. We then measured the execution time of ANTLR parsers on the same benchmarks. The goals of the evaluation were to demonstrate that CoStar runs in linear time on grammars for popular programming languages and data formats, and that CoStar comes within a reasonable constant factor of a comparable unverified tool in terms of its performance.

3.6.1 CoStar Benchmarks

Each CoStar benchmark involved providing the interpreter with a grammar for a real-world programming language or data format, and measuring the resulting 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; the converter is simply an unverified tool that we developed for testing purposes. CoStar takes pretokenized input, so we also built a separate tool that uses an ANTLR grammar to tokenize a data file. When an ANTLR grammar $\mathcal{G}$ is converted to a CoStar-compatible grammar $\mathcal{G}'$ and $\mathcal{G}$ is used to tokenize a data file, the resulting tokens are compatible with $\mathcal{G}'$ (i.e., they use the same terminal symbols as $\mathcal{G}'$).

Benchmark	Grammar Size			Data Set Size	
	$\lvert\mathcal{T}\rvert$	$\lvert\mathcal{N}\rvert$	$\lvert\mathcal{P}\rvert$	# 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 3.8: 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.

We ran benchmarks based on four languages: JSON, XML, DOT, and Python 3. Figure 3.8 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 [Parr et al. 2014] and took the Python 3 grammar from a central repository for ANTLR grammars [Kiers 2014]. We also reused DOT data from the ANTLR evaluation[5] and JSON data from the Vermillion performance evaluation (described in Section 2.5). The XML data is a subset of the Open American National Corpus [OANC 2010] (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 the XML grammar (in ANTLR's EBNF notation):

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

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 four 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 3.9. 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 [Parr et al. 2014]. 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 [Cleveland 1979]. 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.

[5] 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.

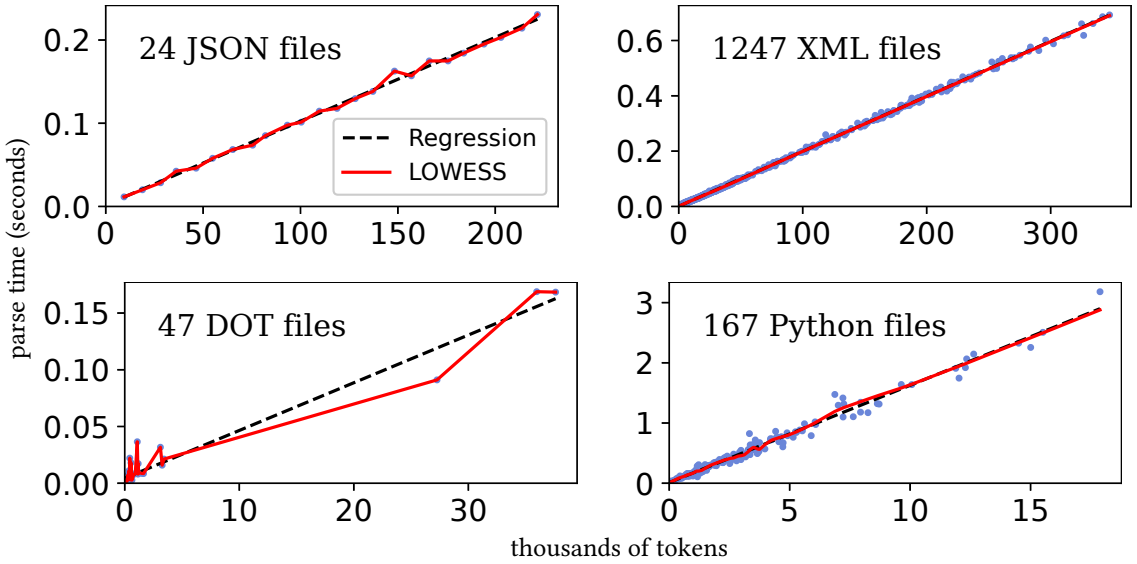


Figure 3.9: Input size vs. CoStar average parse time on four benchmarks. Unconstrained LOWESS curves coincide with regression lines, indicating linear performance. Curves were computed on the smallest 99% of files in each benchmark for scale, with a LOWESS f-hyperparameter value of 0.1. Values of f close to 0 produce a more jagged curve; values close to 1 produce a smoother curve.

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). For example, at the start of each prediction, CoStar must obtain all grammar right-hand sides for the top stack nonterminal; to make this operation efficient, CoStar compiles the grammar into a map from left-hand sides (nonterminals) to right-hand sides. We use 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 map keys or set elements. For larger grammars, the space of keys (or elements) is larger; as Figure 3.8 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 that garbage collection—not comparison—dominates performance.

3.6.2 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 3.6.1. Each benchmark consisted of three complete passes over the data set. ANTLR is written in Java; to allow

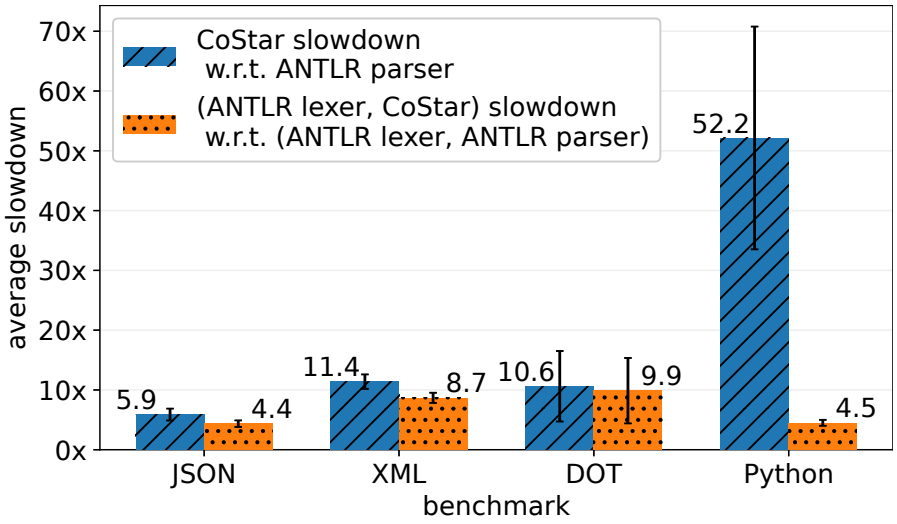


Figure 3.10: CoStar's average slowdown relative to ANTLR on each benchmark. Striped blue bars show CoStar's average slowdown relative to an ANTLR parser. Dotted orange bars show the average slowdown of an "ANTLR lexer, CoStar parser" pairing relative to an "ANTLR lexer, ANTLR parser" pairing. This latter measure represents the cost of replacing an unverified parser with CoStar in a lexing/parsing pipeline. Error bars show standard deviations.

for Java JIT compiler 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 the Java method `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 typical lexing/parsing pipeline. Note that we did not actually integrate CoStar with ANTLR lexers; we merely 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 3.10 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 3.11, left side), so CoStar's

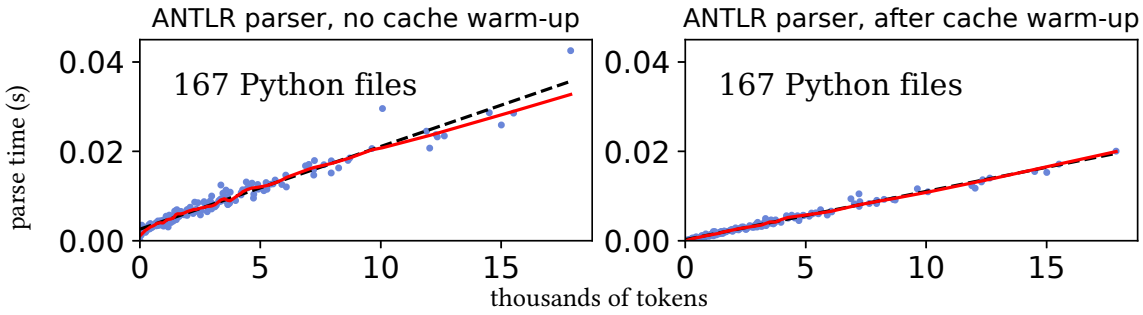


Figure 3.11: Benchmark results for the ANTLR Python parser. The left plot shows that when each benchmark trial involves a newly instantiated parser with an empty cache, performance improves slightly as file size increases. The right plot shows that when a parser with a pre-warmed cache is used in the benchmark, this slight nonlinear effect disappears.

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 on small ones.[6] 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 3.11, 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.[7] 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.

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.

There are several factors that may account for CoStar's slow performance relative to ANTLR (and for the slow performance of verified programs in general, relative to their unverified counterparts). First, the Coq extraction mechanism's default behavior is to translate Coq definitions to OCaml as directly as possible, which can result in inefficient representations (e.g., a unary representation of natural numbers). While it is possible to tweak the extraction mechanism—for

[6] 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 3.8 shows), so cache warm-up occurs even on small files. We do not see similar CoStar behavior on the Python benchmark, either. A plausible explanation is that, as Section 3.3.6 explains, CoStar caches some grammar information statically, so on small files, CoStar might have access to a larger cache than ANTLR.

[7] The ANTLR Python lexer includes embedded code that handles Python's whitespace and indentation rules; for example, it uses a stack to track the current indentation level and emit the correct number of "indent" and "dedent" tokens.

example, by specifying that Coq natural numbers should be extracted to OCaml machine integers—unprincipled tweaking can invalidate correctness guarantees that held for the original Coq program. Second, many algorithms require imperative data structures for maximal efficiency [Lammich 2016]. Algorithms that operate on purely functional data structures must sometimes pay a "$\log n$ penalty" [Appel 2021]; in other words, lookups and insertions take $\log n$ time, as opposed to constant time for an imperative data structure such as an array.

3.7 Conclusions and Future Work

In this chapter, we have presented CoStar, a verified parser interpreter 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 interpreter terminates without error on all inputs. It is therefore a decision procedure for any language denoted by a non-left-recursive grammar. We have demonstrated empirically that CoStar achieves linear-time performance on unambiguous grammars for real-world languages. We also hope that this work serves as an interesting case study on using small-step invariants to verify that an interpreter satisfies a big-step correctness specification. Below, we discuss several natural extensions of this work, one of which is the subject of the following chapter.

We could use richer dependent types in our implementation, which would enable us to rule out some error cases by construction, instead of through external lemmas. For example, we could replace the current representation of machine stacks with dependently typed "well-formed stacks," obviating the need to rule out errors that arise from malformed stacks. Similarly, the "no left recursion" grammar property that appears as an assumption in our correctness theorems is decidable. One could implement and verify a decision procedure that checks a grammar for this property, and then use this procedure to ensure that the interpreter is only invoked on non-left-recursive grammars. These changes could make the extracted code more efficient by eliminating "impossible" cases from the implementation.

The prediction mechanism that Parr et al. [2014] describe uses a graph-structured stack (GSS) to avoid exponential blowup of the search space on ambiguous input. While we have argued that the best way to deal with ambiguity in a high-assurance setting is to avoid it entirely, verifying a GSS implementation and incorporating it into CoStar would be a natural path to pursue. This task would be a research effort in its own right; graphs are challenging to work with in a verified setting because of their non-inductive structure. In a recent paper about developing a Coq graph library and using it to formalize aspects of graph theory, the authors note that "there are only a few formalizations of graph theory results in interactive theorem provers, and even fewer general purpose libraries," and that Coq in particular lacks such a library [Doczkal and Pous 2020]. The library that the paper describes is "geared towards declarative proofs and not towards the extraction of efficient code" [ibid.]—in other words, verification of efficient graph-manipulating programs remains an unsolved problem.

ANTLR rewrites its input grammar to eliminate common forms of left recursion. This

step enables users to include some left recursion in their grammars, which is convenient when the natural way of expressing a construct involves left recursion (e.g., in arithmetic expression grammars). We could implement grammar rewriting and verify that it preserves the grammar's semantics. The hitch is that rewriting changes the parse tree for some words, so we would need to investigate ways to map a tree produced with rewritten grammar $\mathcal{G}'$ to a corresponding tree for original grammar $\mathcal{G}$.

Finally, in the next chapter, we present our work on adding user-defined semantic actions and predicates to CoStar, so that the tool can produce and validate semantic values with a user-defined type instead of the generic parse trees described in this chapter. One difficult aspect of this task is that it complicates the notion of ambiguity, because two distinct parse trees for an ambiguous word can map to the same semantic value. Therefore, we have to update portions of the specification that describe the interpreter's ambiguity detection features.

4

Verified ALL(*) Parsing with Semantic Actions and Dynamic Input Validation

Formally verified parsers are powerful tools for preventing the kinds of errors associated with "shotgun parsing" [Momot et al. 2016]—i.e., interleaving parsing and validation with downstream application code in an undisciplined manner. However, verified parsers are often based on formalisms that are not expressive enough to capture the full definition of valid input for a given application. Specifications of many real-world data formats include both a syntactic component and one or more non-context-free semantic properties that a well-formed instance of the format must exhibit. A parser for context-free grammars (CFGs) cannot determine on its own whether an input is valid according to such a specification; it must be supplemented with additional validation checks.

In this chapter, we present CoStar++, a verified parser interpreter with semantic features that make it highly expressive in terms of both the language specifications it accepts and its output type. CoStar++ provides support for semantic predicates, enabling the user to write semantically rich grammars that include non-context-free properties. The interpreter also supports semantic actions that convert sequential inputs to structured outputs in a principled way and that give the user a high degree of control over the tool's output type. CoStar++ is implemented and verified with the Coq Proof Assistant, and it is based on the ALL(*) parsing algorithm. For all CFGs without left recursion, the interpreter is provably sound, complete, and terminating with respect to a semantic specification that takes predicates and actions into account. It is also provably correct with respect to an ambiguity detection specification—the interpreter tags the semantic values it produces as "unique" whenever they are uniquely correct for the input, and it correctly identifies syntactically ambiguous inputs in cases where detecting semantic ambiguity is undecidable. Verifying CoStar++ presents several interesting challenges—chiefly, ALL(*) as originally described is incomplete with respect to the tool's semantics-aware specification, so we modify the algorithm in a way that guarantees completeness without degrading performance on grammars that are used in practice. CoStar++ runs in linear time on benchmarks for a range of real-world data formats with non-context-free semantic specifications.

4.1 Introduction

The term "shotgun parsing" refers to a programming antipattern in which code for parsing and validating input is interspersed with application code for processing that input. Proponents of high-assurance software argue for the use of dedicated parsing tools as an antidote to this fundamentally insecure practice [Momot et al. 2016]. Such parsers enable the user to write a declarative specification (e.g., a grammar) that describes the structure of valid input, and they reject inputs that do not match the specification, ensuring that only valid inputs reach the downstream application code. Formally verified parsers offer even greater security to the applications that rely on them. Verification techniques can provide strong guarantees that a parser is not "leaky"—i.e., that it accepts all and only the inputs that are valid according to the user's specification.

However, dedicated parsing tools are not always expressive enough to capture the full definition of valid input. For example, a parser for context-free grammars (CFGs) provides limited value when the input specification is too complex for a CFG to capture because the parser must be supplemented with ad hoc input validation code. This situation is not hypothetical; the input specifications of many real applications consist of a context-free *syntactic* specification along with one or more non-context-free *semantic* properties. For example, a CFG can represent the syntax of valid XML, but the grammar cannot capture the requirement that names in corresponding start and end tags must match (assuming that the set of names is infinite). Similarly, the syntactic specification for JSON is context-free, but some applications impose the additional requirement that JSON objects (collections of key-value pairs) contain no duplicate keys—a non-context-free property. The JSON RFC endorses this requirement, noting that applications behave unpredictably in its absence [Bray 2017]. Data dependencies are another common type of non-context-free property; for example, many packet formats have a "tag-length-value" structure in which a length field in the packet header indicates the size of the packet's data field. In each of these cases, a CFG-based parser is an incomplete substitute for shotgun parsing because it cannot enforce the semantic component of the input specification.

In this chapter, we present CoStar++, a formally verified parser interpreter with two features—semantic predicates and semantic actions—that enable it to capture semantically rich specifications like the ones described above. Semantic predicates enable the user to write input specifications that include non-context-free semantic properties. The interpreter checks these properties at runtime, ensuring that its output is well-formed. Semantic actions give the user fine-grained control over the interpreter's output type, greatly increasing the tool's suitability for real-world usage. Actions also play an important role in supporting predicates; the interpreter must produce values with an expressive type in order to check interesting properties of those values. CoStar++ builds on the CoStar tool [Lasser et al. 2021c] described in the previous chapter. Like its predecessor, CoStar++ is based on the ALL(*) parsing algorithm, and it is implemented and verified with the Coq Proof Assistant.

Extending CoStar with semantic predicates and actions gives rise to several interesting challenges in terms of adapting the ALL(*) algorithm to a semantic setting, specifying the new version's behavior, and proving it correct with respect to the new semantic specification:

- A key feature of CoStar's specification is that the tool correctly detects syntactically ambiguous inputs (inputs with more than one parse tree). In a semantic setting, the definition of ambiguity is more complex; it can be syntactic (multiple parse trees for an input) or semantic (multiple semantic values for an input). In addition, it is not always possible to infer one kind of ambiguity from the other, because two parse trees can correspond to (a) two semantic values, (b) a single semantic value when the semantic actions for the two derivations produce the same value, or (c) no semantic value at all when predicates fail during the semantic derivations! Finally, detecting semantic ambiguity is undecidable in the general case where semantic values do not have decidable equality, and we choose not to require this property so that the parser can produce incomparable values such as functions. However, it is still possible to detect the *lack* of semantic ambiguity (i.e., semantic "uniqueness"). As part of the work described in this chapter, we modify the CoStar ambiguity detection mechanism and its associated correctness invariants so that CoStar++ provably detects uniquely correct semantic values, and it provably detects syntactic ambiguity in the cases where semantic ambiguity is undecidable.

- ALL(*) as originally described by Parr et al. [2014] and as implemented by CoStar is incomplete with respect to the CoStar++ semantic specification. ALL(*) is a predictive parsing algorithm; at decision points, it nondeterministically explores possible paths until it identifies a uniquely viable path. This prediction strategy does not speculatively execute semantic actions or evaluate semantic predicates over those actions, for both efficiency and correctness reasons (the actions could alter mutable program state in ways that cannot be undone). While this choice is reasonable in the imperative setting for which ALL(*) was developed, it renders the algorithm incomplete relative to a predicate-aware specification, because the prediction algorithm can send the parser down a path that leads to a predicate failure when a different path would have produced a successful parse. CoStar++ solves this problem by using a modified version of the ALL(*) prediction algorithm that evaluates predicates and actions only when doing so is necessary to guarantee completeness. CoStar++ semantic actions are pure functions, so speculatively executing them during prediction is safe.

This chapter makes the following contributions:

- We present CoStar++, an extension of CoStar that adds support for semantic predicates and actions. These new semantic features increase the expressivity of both the language definitions that the interpreter can accept and its output type.

- We present a modified version of the ALL(*) prediction algorithm that CoStar++ uses to ensure completeness in the presence of semantic predicates.

- We prove that for all CFGs without left recursion, CoStar++ is sound, complete, and terminating with respect to a semantics-aware specification that takes predicates and actions into account.

- We prove that CoStar++ identifies uniquely correct semantic values and that it detects syntactic ambiguity in cases where semantic ambiguity is undecidable.

```
Inductive json_value : Type :=
(* string keys in kv_pairs should be unique *)
| JObj  (kv_pairs : list (string * json_value))
| JArr  (vs : list json_value)
| JBool (b : bool)
| JNum  (i : Z)
| JStr  (s : string)
| JNull.
```

Figure 4.1: Algebraic data type representation of JSON values, shown in the concrete syntax of Gallina, the functional programming language embedded in Coq. A JSON value is either a sequence of key/value pairs, an array of JSON values, or a base value (boolean, number, string, or null).

- We demonstrate CoStar++'s expressiveness by using it to write grammars with interesting non-context-free semantic properties for a range of real-world data formats, and we show that CoStar++ achieves linear-time performance on benchmarks for these formats. As part of the CoStar++ evaluation, we integrate the tool with our Verbatim verified lexer [Egolf et al. 2021, 2022] to create a fully verified front end for lexing and parsing data formats.

CoStar++ is a Coq development consisting of roughly 6,500 lines of specification and 7,000 lines of proof. The grammars used in the performance evaluation comprise another 700 lines of specification and 100 lines of proof. CoStar++ and its accompanying performance evaluation framework are open source and available online [Lasser et al. 2021b].

The chapter is organized as follows. In §4.2, we introduce CoStar++ by example, highlighting its semantic features. We sketch CoStar++'s high-level correctness properties in §4.3. In §4.4, we discuss the challenge of specifying the interpreter's behavior on syntactically and semantically ambiguous input. In §4.5, we discuss the challenge of ensuring completeness after adding semantic predicates to the interpreter's correctness specification, and we present a semantics-aware modification of the ALL(*) prediction algorithm. In §4.6, we highlight several interesting features of the Coq mechanization. In §4.7, we give a brief overview of the Verbatim verified lexer, which is designed to be compatible with CoStar++. In §4.8, we present the results of a performance evaluation and describe the semantic features of the grammars used in the evaluation. We give an overview of related work in §4.9 and conclude in §4.10.

4.2 CoStar++ by Example

In this section, we give an example of a simple grammar that includes a non-context-free semantic property, and we sketch the execution of the CoStar++ parser that this grammar specifies, with a focus on the parser's semantic features.

```
Value  ::= Object            ⟦λ(prs,_).nodupKeys prs⟧?  ⟦λ(prs,_).JObj prs⟧!
         | Array                                         ⟦λ(vs,_).JArr vs⟧!
         | ...
Object ::= '{' Pair Pairs '}'                            ⟦λ(_,pr,prs,_,_).pr :: prs⟧!
         | '{'           '}'                             ⟦λ_.[]⟧!
...
```

Figure 4.2: JSON grammar fragment annotated with semantic predicates and actions. An annotated production has the form $X ::= \gamma\ [\![p]\!]?\ [\![f]\!]!$, where X is a nonterminal, γ is a sequence of grammar symbols, p is an optional predicate over the semantic values $\bar{v}$ that γ produces, and f is an action that is applied to $\bar{v}$. Nonterminals begin with capital letters and terminals appear in single quotes. Throughout the chapter, when it is necessary to distinguish between terminals and the literal values that they match, we write terminal names in angle brackets (e.g., `<int>` for a terminal that matches an integer).

4.2.1 A Grammar for Parsing Duplicate-Free JSON

Suppose we want to use CoStar++ to define a JSON parser, and we only want the parser to accept JSON input in which objects contain no duplicate keys. The parser's output type might look like the algebraic data type (ADT) in Figure 4.1. To obtain a parser that produces values of this type, and that enforces the "unique keys" invariant, we can provide CoStar++ with a grammar like the one excerpted in Figure 4.2. This grammar includes the following components:

- BNF **productions** describe the syntactic structure of valid JSON.
- Productions are annotated with **semantic actions** (wrapped in $\langle\rangle!$ delimiters). Actions build the semantic values that the parser produces; they determine the parser's output type. As was the case for Vermillion (Chapter 2), an action is a function with a dependent type that is determined by the grammar symbols in the accompanying production. A semantic action for production $X ::= \gamma$ has type $[\![\gamma]\!] \to [\![X]\!]$, where the semantic tuple type $[\![\gamma]\!]$ is computed as follows:

$$[\![\bullet]\!] = \mathbb{1}$$
$$[\![s\beta]\!] = [\![s]\!] \times [\![\beta]\!]$$

  and $[\![s]\!]$ is a user-defined mapping from grammar symbols to their semantic types. For the example grammar, $[\![$`Value`$]\!]$ = `json_value` (i.e., the parser produces a `json_value` each time it processes a `Value` nonterminal), and $[\![$`Object`$]\!]$ = `list (string * json_value)`.
- Productions are optionally annotated with **semantic predicates** (wrapped in $\langle\rangle?$ delimiters). A predicate for production $X ::= \gamma$ has type $[\![\gamma]\!] \to \mathbb{B}$. At parse time, CoStar++ applies predicates to the semantic values that the actions produce and rejects the input when a predicate fails.

In our grammar notation, the left-to-right structure of an annotated production hints at its procedural interpretation. A production like this one:

$$\texttt{Value} ::= \texttt{Object} \quad [\![\lambda\texttt{(prs,\_).nodupKeys prs}]\!]? \quad [\![\lambda\texttt{(prs,\_).JObj prs}]\!]!$$

can be read as follows: "To produce a result of type $[\![\texttt{Value}]\!]$, first produce a tuple of type $[\![\texttt{Object}]\!]$ and apply predicate $[\![\lambda\texttt{(prs,\_).nodupKeys prs}]\!]?$ to it (where `nodupKeys` is a function that checks whether the string keys in an association list are unique). If the predicate succeeds, apply action $[\![\lambda\texttt{(prs,\_).JObj prs}]\!]!$ to the tuple."

4.2.2 Parsing Valid Input

CoStar++ is implemented as a stack machine with a small-step semantics. At each point in its execution, the machine performs a single atomic update to its state based on its current configuration. CoStar++ has the same machine state components as CoStar (listed in Section 3.3.2) except for the fact that instead of maintaining separate prefix and suffix stacks, CoStar++ uses a single stack. Each frame $[\alpha \;\&\; \bar{v}, \beta]$ of a stack ϕ holds processed symbols α, semantic tuple $\bar{v} : [\![\alpha]\!]$ for the processed symbols, and unprocessed symbols β.

In Figure 4.3, we illustrate how CoStar++ realizes the example JSON grammar's semantics by applying CoStar++ to the grammar and tracing the resulting parser's execution on valid JSON input. Figure 4.3 shows the machine's stack at each point in the trace (other components of the machine state are omitted for ease of exposition). In the initial state σ_0, the stack consists of a single frame $[\bullet \;\&\; \texttt{tt}, \texttt{Value}]$ that holds an empty sequence of processed symbols $\bullet$, a semantic value of type $[\![\bullet]\!]$ (`tt`, the sole value of type `unit`), and a sequence of unprocessed symbols that contains only the start symbol `Value`.

Each machine state also stores the sequence of remaining tokens. Each token $(a \;\&\; v)$ is the dependent pair of a terminal symbol a and a literal value $v : [\![a]\!]$. (In our performance evaluation, we use a verified lexer that produces tokens of this type; see Section 4.8 for details.) In the Figure 4.3 example, the input string before tokenization is:

```
{"k1": "foo", "k2": 42}
```

Thus, in initial state σ_0, the machine's remaining tokens are the tokens for the full input string:

$$(\texttt{'\{'} \;\&\; \texttt{tt}),\ (\texttt{<string>} \;\&\; \texttt{"k1"}),\ (\texttt{':'} \;\&\; \texttt{tt}),\ (\texttt{<string>} \;\&\; \texttt{"foo"}) \ldots$$

In the transition from σ_0 to σ_1, the machine performs a **push** operation. A push occurs when the top stack symbol (the next unprocessed symbol in the top stack frame) is a nonterminal—`Value`, in this case. During a push, the parser examines the remaining tokens to determine which grammar right-hand side to push onto the stack. The prediction subroutine that performs this task is what distinguishes ALL(*) from other parsing algorithms. Parr et al. [2014] describe the prediction mechanism in detail; in brief, the parser launches a subparser for each candidate right-hand side and advances the subparsers only as far as necessary to identify a unique viable choice. In the example, the prediction mechanism identifies the right-hand side `Object` as the unique viable choice and pushes it onto the stack in a new frame.

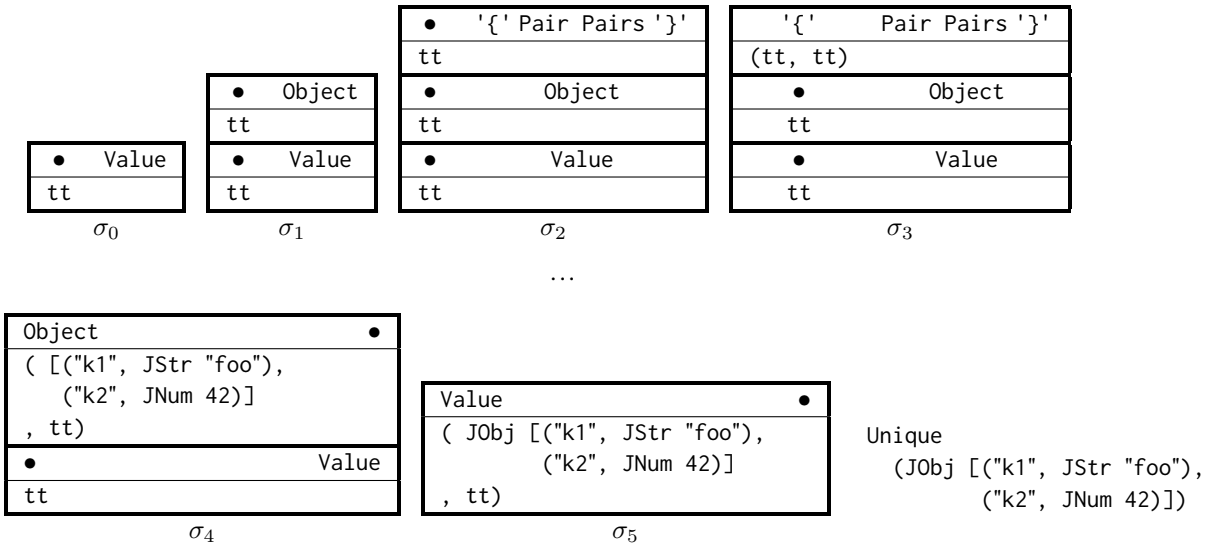


Figure 4.3: Execution trace of a CoStar++ JSON parser applied to the valid string `{"k1": "foo", "k2": 42}`. The parser's stack is shown at each point in the trace. A stack frame contains a sequence of processed grammar symbols α (shown in the upper left portion of the frame), a sequence of unprocessed grammar symbols β (upper right portion), and a semantic tuple of type $[\![\alpha]\!]$ (lower portion).

The transition from σ_1 to σ_2 is another push operation, in which the prediction mechanism identifies `'{' Pair Pairs '}'` as the unique right-hand side for nonterminal `Object` that may produce a successful parse. To transition from σ_2 to σ_3, the machine performs a **consume** operation. A consume occurs when the top stack symbol is a terminal a. The machine matches a against terminal a' from the head remaining token. In this case, the top stack terminal `'{'` matches the terminal in token (`'{'` & `tt`), so the machine pops the token and stores its semantic value `tt` in the current frame.

After performing several more operations, the machine reaches state σ_4. At this point, the machine has fully processed nonterminal `Object`, producing a semantic value of type $[\![$`Object`$]\!]$ = `list (string * json_value)`, there are no more symbols left to process in the top frame, and nonterminal `Value` in the frame below has not yet been fully processed (we call such a nonterminal "open", and the frame containing it the "caller" frame). In such a configuration, the machine performs a **return** operation. A return involves the following steps:

1. The machine retrieves the semantic predicate and action for the grammar production being reduced. In the Figure 4.3 example, the production is `Value ::= Object`, the predicate is $[\![\lambda$`(prs,_).nodupKeys prs`$]\!]?$, and the action is $[\![\lambda$`(prs,_).JObj prs`$]\!]!$.

2. The machine applies the predicate to the tuple of semantic values $\bar{v}$ in the top frame. In the example, the predicate evaluates to `true` because the list of key/value pairs contains no duplicate keys.

3. If the predicate succeeds (as it does in the example), the machine applies the action to $\bar{v}$, producing a new semantic value v'. It then pops the top frame, moves the open nonterminal

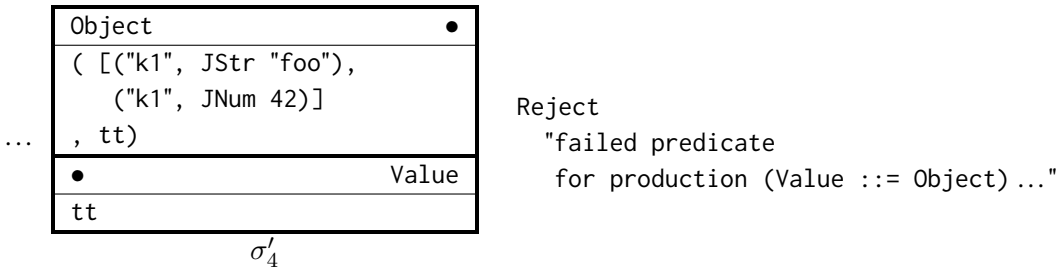


Figure 4.4: Partial execution trace of a CoStar++ JSON parser on the string `{"k1": "foo", "k1": 42}`, which is syntactically valid but semantically malformed according to the Figure 4.2 JSON grammar because of its duplicate keys. When the parser reaches state σ'_4 (which corresponds to state σ_4 in the Figure 4.3 trace), a predicate detects duplicate keys in the top frame's semantic tuple, and the parser rejects the input as invalid.

in the caller frame to the list of processed symbols, and stores v' in the caller frame. In this case, the machine makes `Value` a processed symbol (the nonterminal has now been fully reduced) and stores $v' =$ `JObj [("k1", JStr "foo"), ("k2", JNum 42)]` in the caller frame.

In state σ_5, the machine is in a final configuration; there are no unprocessed symbols in the top frame, and no caller frame to return to. In such a configuration, the machine halts and returns the semantic value it has accumulated for the start symbol. It tags the value as `Unique` or `Ambig` based on the value of another machine state component: a boolean flag indicating whether the machine detected ambiguity during the parse. In our example, the input is unambiguous, so the machine tags the final `json_value` as `Unique`.

4.2.3 Handling Semantically Malformed Input

In Figure 4.4, we illustrate how the parser's behavior differs on the JSON string `{"k1": "foo", "k1": 42}`, which is syntactically well-formed but violates the "no duplicate keys" property of our specification. During the first several steps involved in processing this string, the machine stacks match those in Figure 4.3. In state σ'_4, which corresponds to state σ_4 in Figure 4.3, the machine attempts to perform a return by applying the semantic predicate for production `Value ::= Object` to the list of key/value pairs `[("k1", JStr "foo"), ("k1", JNum 42)]`. This time, the predicate fails because of the duplicate keys, so the machine halts and returns a `Reject` value along with a message describing the failure.

Note that in Figure 4.4, a predicate fails while the machine is attempting to produce a semantic value for the start symbol, which is the very last step in the machine's execution. However, if the duplicate keys had appeared in an object within a larger JSON structure, the failing predicate would have caused the machine to halt as soon as that object was fully processed. This fail-fast behavior spares the interpreter from processing additional input when it detects a semantic property violation.

4.3 Interpreter Correctness

CoStar++ is correct with respect to a high-level specification that encompasses the behavior of semantic predicates and actions. The interpreter also correctly detects syntactically ambiguous input. CoStar++ has the following correctness properties when its input grammar is non-left-recursive:

- (*Soundness, unique derivations*): If the interpreter returns a semantic value v labeled as `Unique`, then v is the sole correct semantic value for the input.
- (*Soundness, ambiguous derivations*): If the interpreter returns a semantic value v labeled as `Ambig`, then v is a correct semantic value for the input, and there exist multiple distinct parse trees (i.e., concrete syntax trees) for the input.
- (*Error-free termination*): The interpreter terminates on all inputs without returning an error value.
- (*Completeness*): If v is a correct semantic value for the input, then either (a) v is uniquely correct for the input and the interpreter returns `Unique(`v`)`, or (b) there exist multiple distinct parse trees for the input, and the interpreter returns a correct semantic value `Ambig(`v'`)`.

4.3.1 Correctness Specification

CoStar++ is sound and complete relative to a grammatical derivation relation called SemValueDer with the judgment form $s \xrightarrow{v} w$, meaning that symbol s derives word w, producing semantic value v. Figure 4.5a shows this relation as well as a mutually inductive one, SemValuesDer, over sentential forms (grammar right-hand sides). This latter relation has the judgment form $\gamma \xrightarrow{\bar{v}} w$ (symbols γ derive word w, producing semantic tuple $\bar{v}$). In terms of semantic predicates and actions, the key rule is NonterminalSemDer, which says that if (a) $X ::= \gamma\ [\![p]\!]?\ [\![f]\!]!$ is a grammar production; (b) the right-hand side γ derives word w, producing the semantic tuple $\bar{v}$; and (c) $\bar{v}$ satisfies predicate p, then applying action f to $\bar{v}$ produces a correct semantic value for left-hand nonterminal X.

Portions of the correctness theorems refer to the existence of correct parse trees for the input. Parse tree correctness is defined in terms of a pair of mutually inductive relations, TreeDer and ForestDer (Figure 4.5b). These relations are isomorphic to SemValueDer and SemValuesDer, but they produce parse trees and parse tree lists (respectively), where a parse tree is an *n*-ary tree with terminal-labeled leaves and nonterminal-labeled internal nodes.

4.3.2 Parser Correctness Theorems

The main CoStar++ correctness theorems describe the behavior of the tool's top-level `parse` function, which has the signature shown in Figure 4.6b. The `parse` function takes a grammar $\mathcal{G}$, a proof that $\mathcal{G}$ is well-formed (Section 4.6.1 describes this grammar well-formedness property), a start nonterminal S, and a token sequence $\bar{t}$. It produces a `parse_result(`S`)`, a dependent type indexed by S. The `parse_result` definition appears in Figure 4.6a. A `parse_result(`X`)` is either a semantic

$\boxed{\textsc{SemValueDer}:\ s \xrightarrow{v\ :\ [\![s]\!]} w}$

$$\textsc{TerminalSemDer}\ \frac{}{a \xrightarrow{v} (a\ \&\ v)} \qquad \textsc{NonterminalSemDer}\ \frac{X ::= \gamma\ [\![p]\!]?\ [\![f]\!]!\ \in \mathcal{G} \qquad \gamma \xrightarrow{\bar{v}} w \qquad p(\bar{v}) = \texttt{true}}{X \xrightarrow{f(\bar{v})} w}$$

$\boxed{\textsc{SemValuesDer}:\ \gamma \xrightarrow{\bar{v}\ :\ [\![\gamma]\!]} w}$

$$\textsc{NilSemDer}\ \frac{}{\bullet \xrightarrow{\texttt{tt}} \epsilon} \qquad \textsc{ConsSemDer}\ \frac{s \xrightarrow{v} w_1 \qquad \beta \xrightarrow{\bar{v}} w_2}{s\beta \xrightarrow{(v,\bar{v})} w_1 w_2}$$

(a) Semantic derivation relations for symbols (SemValueDer) and sentential forms (SemValuesDer).

$\boxed{\textsc{TreeDer}:\ s \xrightarrow{t\ :\ \mathit{tree}} w}$

$$\textsc{TerminalLeafDer}\ \frac{}{a \xrightarrow{\texttt{Leaf}(a)} (a\ \&\ v)} \qquad \textsc{NonterminalNodeDer}\ \frac{X ::= \gamma\ \in \mathcal{G} \qquad \gamma \xrightarrow{\bar{t}} w}{X \xrightarrow{\texttt{Node}(X,\bar{t})} w}$$

$\boxed{\textsc{ForestDer}:\ \gamma \xrightarrow{\bar{t}\ :\ \mathit{list\ tree}} w}$

$$\textsc{NilForestDer}\ \frac{}{\bullet \xrightarrow{\bullet} \epsilon} \qquad \textsc{ConsForestDer}\ \frac{s \xrightarrow{t} w_1 \qquad \beta \xrightarrow{\bar{t}} w_2}{s\beta \xrightarrow{t,\bar{t}} w_1 w_2}$$

(b) Syntactic derivation relations for symbols (TreeDer) and sentential forms (ForestDer).

Figure 4.5: Mutually inductive grammar derivation relations that serve as the high-level correctness specification for CoStar++.

value of type $[\![X]\!]$ tagged as `Unique` or `Ambig` (indicating whether the input is ambiguous), a `Reject` value with a message explaining why the input was rejected, or an `Error` value indicating that the stack machine reached an inconsistent state.

We list the CoStar++ high-level correctness theorems below, and highlight several interesting aspects of their proofs in Sections 4.4 and 4.5. Each theorem assumes a non-left-recursive grammar $\mathcal{G}$.

Theorem 4.3.1 (Soundness, unique derivations)**.** If `parse` applied to $\mathcal{G}$, nonterminal S, and word w returns a semantic value `Unique`(v), then v is the sole correct semantic value for S and w.

Theorem 4.3.2 (Soundness, ambiguous derivations)**.** If `parse` applied to $\mathcal{G}$, nonterminal S, and word w returns a semantic value `Ambig`(v), then v is a correct semantic value for S and w, and there exist two correct parse trees t and t' for S and w, where $t \neq t'$.

```
parse_result (x : nonterminal) :=
| Unique (v : ⟦x⟧)
| Ambig  (v : ⟦x⟧)
| Reject (s : string)
| Error  (e : parse_error)
```

(a) parse return type

```
parse (g  : grammar)
      (Hw : grammar_wf g)
      (s  : nonterminal)
      (ts : list token) :
      parse_result s
```

(b) parse type signature

Figure 4.6: The definition of the CoStar++ interpreter's return type (a), and the type signature of parse, the interpreter's top-level entry point (b).

Theorem 4.3.3 (Error-free termination)**.** The interpreter never returns an Error value.

Theorem 4.3.4 (Completeness)**.** If semantic value v is a correct semantic value for nonterminal S and word w, then either (a) v is the sole correct semantic value for S and w and the interpreter returns Unique(v), or (b) multiple correct parse trees exist for S and w, and the interpreter returns a correct semantic value Ambig(v').

The theorems above have been mechanized in Coq. Each theorem has a proof based on (a) an invariant I over the machine state that implies the high-level theorem when it holds for the machine's final configuration; and (b) a preservation lemma showing that each machine operation (push, consume, and return) preserves I. Section 4.5.3 contains an example of such an invariant.

4.4 Semantic Actions and Correct Ambiguity Detection

There is an apparent type mismatch between the "unique" and "ambiguous" soundness theorems in Section 4.3. According to Theorem 4.3.1, a Unique(v) parse result indicates that v is a uniquely correct *semantic value* for the input, while Theorem 4.3.2 says that an Ambig(v) result implies the existence of multiple correct *parse trees* for the input. The reason for this asymmetry is that syntactically ambiguous inputs may not be ambiguous at the semantic level; semantic actions can map two distinct parse trees for an input to the same semantic value, and predicates can eliminate semantic ambiguity by rejecting semantic values as malformed. For these reasons, the problem of identifying semantic ambiguity is undecidable when semantic values lack decidable equality. When CoStar++ flags an ambiguous input, it is only able to guarantee that ambiguity exists at the syntactic level.

We illustrate this point with an example involving the somewhat contrived grammar in Figure 4.7. Start symbol X matches an <int><string><bool> sequence in two possible ways—one involving the first right-hand side for X, and one involving the second right-hand side. These two right-hand sides can be used to derive two distinct parse trees for such a token sequence (we represent leaves as terminal symbols for readability):

(1a) Node X [<int>, Node Y [<string>, <bool>]]

(1b) Node X [Node Z [<int>, <string>], <bool>]

```
X ::= <int> Y              ⟦λ(i,(s,_),_).i - String.length s⟧!
    | Z <bool>             ⟦λ((_,s),b,_).if b then String.length s else 0⟧!
Y ::= <string> <bool>      ⟦λ(s,b,_).(s,b)⟧!
Z ::= <int> <string>       ⟦λ(i,s,_).(i,s)⟧!
```

Figure 4.7: Grammar that recognizes an <int><string><bool> token sequence. The grammar is syntactically ambiguous, but for some inputs, two different semantic derivations produce the same semantic value.

However, while any <int><string><bool> sequence is ambiguous at the syntactic level, only some inputs are semantically ambiguous. For example, on input

(<int> & 10) (<string> & "apple") (<bool> & false)

the actions attached to the two right-hand sides for X produce two distinct semantic values:

(2a) `10 - String.length "apple" = 5`

(2b) `if false then String.length "apple" else 0 = 0`

However, replacing the literal value in the <bool> token with true makes the two derivations produce the same semantic value:

(3a) `10 - String.length "apple" = 5`

(3b) `if true then String.length "apple" else 0 = 5`

In theory, when CoStar++ identifies multiple semantic values for these examples, it could determine whether the input is semantically ambiguous by comparing the values, because integer equality is decidable. However, semantic types are user-defined, and we do not require them to have decidable equality; the user may want the interpreter to produce functions or other incomparable values. Therefore, in the general case, the interpreter can only certify that the input has two distinct parse trees—this guarantee is the one that Theorem 4.3.2 provides.

We can generalize this example in terms of a function $[\![\mapsto]\!]$ that maps a parse tree for a word w to a "corresponding" semantic value for w. The notation $t\ [\![\mapsto]\!]\ v$ means that parse tree t and semantic value v are constructed from the same grammar productions. For example, the JSON parse tree and semantic value in Figure 4.8 correspond; they derive the same JSON string (using the TreeDer and SemValueDer relations, respectively), and the two derivations use the same JSON grammar productions.

Figure 4.9 represents the $[\![\mapsto]\!]$ function for a hypothetical grammar. Word w_2 is syntactically ambiguous; it has two distinct parse trees that correspond to two distinct semantic values. Word w_3 is ambiguous in terms of syntax but not semantics; its two distinct parse trees correspond to the same semantic value. Word w_4 is ambiguous at the syntactic level, but one of its two parse trees corresponds to a semantic value that the grammar predicates identify as malformed, so value v_5 is uniquely correct for w_4. In general, CoStar++ cannot distinguish between the w_2 and w_3 cases, for the reasons described above; it returns an `Ambig` result in both cases. However, the interpreter identifies v_5 as uniquely correct; it returns `Unique(`v_5`)` for w_4.

```
Definition t : parse_tree :=
  Node Object [ Leaf LEFT_BRACE
              ; Node Pair [Leaf STRING; Leaf COLON; Node Value [Leaf STRING]]
              ; Node Pairs [ Leaf COMMA
                           ; Node Pair [ Leaf STRING
                                       ; Leaf COLON
                                       ; Node Value [Leaf NUM]]
                           ; Node Pairs []]
              ; Leaf RIGHT_BRACE].

Definition v : json_value := JObject [("k1", JStr "foo"); ("k2", JNum 42)].
```

Figure 4.8: Corresponding parse tree `t` and semantic value `v` for the JSON string `w = {"k1": "foo", "k2": 42}`, shown in the concrete syntax of Gallina. Tree `t` and value `v` are correct for `w` in terms of the TreeDer and SemValueDer derivation relations, respectively, and the syntactic derivation of `t` uses the same grammar productions as the semantic derivation of `v`.

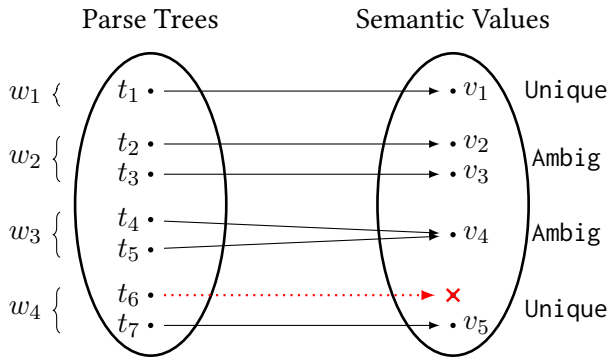


Figure 4.9: One possible mapping from correct parse trees to correct semantic values for a hypothetical grammar. The mapping is non-injective; trees t_4 and t_5 are built from grammar productions with semantic actions that produce the same semantic value, v_4. The mapping is also partial; t_6 corresponds to a semantic value that the grammar's predicates reject as malformed.

4.5 Semantic Predicates and Parser Completeness

One of the main challenges of implementing and verifying CoStar++ was ensuring completeness in the presence of semantic predicates. In this section, we describe how predicates break completeness guarantees for the standard presentation of ALL(*), and present a modification to ALL(*) that provably restores these guarantees.

4.5.1 Naive ALL(*) Prediction and Predicates

ALL(*) is a predictive parsing algorithm; at decision points, the parser speculatively explores alternative paths to identify a path that may lead to a successful derivation. ALL(*) as originally described by Parr et al. [2014] does not apply semantic actions or check CoStar++-style predicates at prediction time. (The algorithm does support predicates with a different semantics that

X ::= <int> Y			⟦λ(i,_,_).i⟧!
| Z <bool>			⟦λ(_,b,_).if b then 1 else 0⟧!
Y ::= <string> <bool>	⟦λ(s,b,_).!b || startsWith s "a"⟧?		⟦λ_.tt⟧!
Z ::= <int> <string>	⟦λ(i,s,_).i = String.length s⟧?		⟦λ_.tt⟧!

Figure 4.10: Ambiguous grammar that accepts an <int><string><bool> sequence. The predicates associated with the two possible derivation paths place different data dependencies on the input.

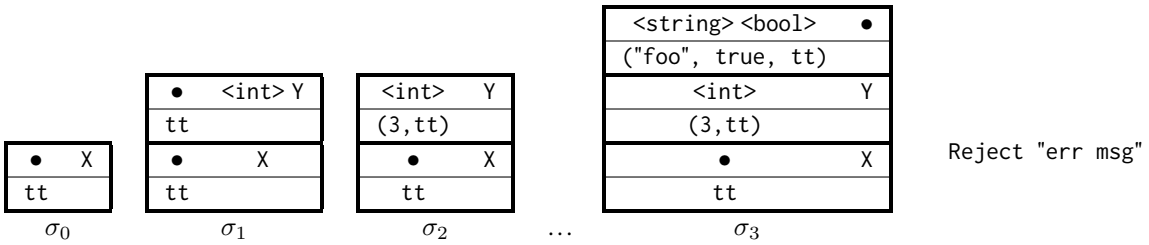
can be evaluated during predictions; see Section 4.9 for a discussion of the differences.) However, a predicate-oblivious prediction algorithm results in a parser that is incomplete relative to the SemValueDer specification; i.e., it can make a choice that eventually causes the parser to reject input as invalid, when a different choice would have led to a successful parse.

To illustrate the problem, we trace the execution of the "textbook" ALL(*) prediction mechanism on an input that matches the predicate-annotated grammar in Figure 4.10. This grammar can match an <int><string><bool> sequence in two different ways, but these two derivation paths place two different semantic restrictions on the input: the path involving nonterminal Y places a data dependency between the string and boolean values, while the path involving nonterminal Z places a dependency between the integer and string values.

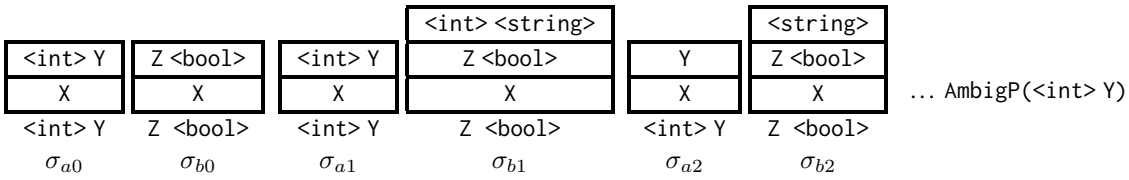
Figure 4.11 shows a possible trace of CoStar++ with predicate-oblivious prediction on the Figure 4.10 grammar and the input (<int> & 3)(<string> & "foo")(<bool> & true). In the initial parser state σ_0 (Figure 4.11a), the top stack symbol is start symbol X. To determine which right-hand side for X to push onto the stack, the prediction algorithm launches two subparsers with the stacks shown in states σ_{a0} and σ_{b0} (Figure 4.11b). Each subparser stack's top frame holds a right-hand side for the decision nonterminal X. The subparser is also labeled with this right-hand side; the label keeps track of which alternative for X the subparser is exploring.

The prediction algorithm's first step from $\{\sigma_{a0}, \sigma_{b0}\}$ to $\{\sigma_{a1}, \sigma_{b1}\}$ is a closure operation; each subparser performs push and return operations until its top stack symbol is a terminal. The next transition to $\{\sigma_{a2}, \sigma_{b2}\}$ is a move operation, in which each subparser consumes a token or dies off if its top stack terminal does not match the token. The prediction algorithm then alternates between move and closure operations until it determines that both right-hand sides for X are viable. At this point, the algorithm is free to choose either alternative; it returns AmbigP(<int> Y), indicating that right-hand side <int> Y is viable, and that the algorithm detected ambiguity.

The parser then uses the prediction result in a push operation; it transitions from σ_0 to σ_1 by pushing <int> Y onto the stack. Parsing continues until the machine reaches state σ_3. In an attempt to perform a return, the machine applies predicate ⟦λ(s,b,_).!b || startsWith s "a"⟧? to the semantic tuple ("foo", true, tt) in the top frame—the predicate fails, and the machine rejects the input as invalid. However, had the prediction mechanism returned AmbigP(Z <bool>) instead of AmbigP(<int> Y), the parser would have accepted the input as valid. It would have applied predicate ⟦λ(i,s,_).i = String.length s⟧? to the semantic tuple (3, "foo", tt)—a check that succeeds.



(a) Possible execution trace of an incomplete ALL(*) parser on the Figure 4.10 grammar and valid token sequence (<int> & 3) (<string> & "foo") (<bool> & true). The parser rejects the input, although pushing Z <bool> onto the stack instead of <int> Y in state σ_1 would have led to a successful parse.



(b) Possible execution trace of the standard ALL(*) prediction algorithm, starting from state σ_0 in the main parsing loop (Figure 4.11a). The algorithm ignores semantic predicates and thus determines that both right-hand sides for X are viable, sending the parser down a path that leads to rejection.

Figure 4.11: Execution trace of an ALL(*) parser that does not evaluate semantic predicates at prediction time, illustrating why predicate-oblivious prediction would cause CoStar++ to reject valid input.

This example demonstrates that the standard version of ALL(*) prediction is insufficient to guarantee completeness with respect to our semantic specification.

4.5.2 A Semantics-Aware Prediction Mechanism

The semantics-aware version of CoStar++ uses a modified version of the ALL(*) prediction algorithm that is guaranteed not to send the parser down a "bad path" like the one in Figure 4.11. In designing this modification, we faced a tradeoff between speed and expressiveness; checking predicates and building semantic values along all prediction paths is potentially expensive, but it is sometimes necessary to ensure completeness.

Our solution leverages the fact that the original ALL(*) prediction mechanism addresses a similar problem; it is actually a combination of two different prediction strategies that make different tradeoffs with respect to speed and expressiveness:

- **SLL prediction** is an optimized algorithm that ignores the initial parser stack at the start of prediction. As a result, subparser states are compact and recur frequently, which makes them amenable to caching. The tradeoff is that because of the missing context, SLL prediction must sometimes overapproximate the parser's behavior by simulating a return operation to *all* possible contexts. (See Section 3.3.4.2 for a more detailed description of this optimization.)

- **LL prediction** is a slower but sound algorithm in which subparsers have access to the initial parser stack; the algorithm is thus a precise nondeterministic simulation of the parser's behavior. The execution trace in Figure 4.11b is actually an example of LL prediction. When

the SLL algorithm detects an ambiguity, the prediction mechanism fails over to the LL strategy to determine whether the ambiguity is genuine or involves a spurious path introduced by the overapproximation; using the result of SLL prediction directly in such a case would render the parser incomplete.

The semantics-aware prediction mechanism works as follows:

- SLL prediction is unchanged; subparsers do not build semantic values or check semantic properties. SLL is thus still an overapproximation of the parser—not evaluating the predicates is equivalent to assuming that they succeed.
- LL prediction builds semantic values and checks semantic properties along all paths—it thus remains a precise nondeterministic simulation of the parser.

This approach is based on the assumption that most predictions are unambiguous even without considering predicates, and the more expensive LL strategy is thus rarely required.

One minor downside of the modification is that CoStar++ must use two different stack representations for SLL and LL subparsers; in contrast, CoStar uses a single stack representation. SLL prediction is a cache-based optimization, and the finite collections that CoStar++ uses to store cached SLL subparser states require their contents to have decidable equality (or decidable ordering). As discussed in Section 4.4, semantic values are not guaranteed to have these properties, so the semantic stacks used in LL prediction have an uncacheable type.

4.5.3 A Backward-Looking Completeness Invariant

Adding semantic features to LL prediction makes CoStar++ complete with respect to the SemValueDer specification. Theorem 4.3.4 (the interpreter completeness theorem) relies on the following lemma:

Lemma 4.5.1 (Completeness modulo ambiguity detection)**.** If v is a correct semantic value for nonterminal S and word w, then there exists a semantic value v' such that the interpreter returns either `Unique(`v'`)` or `Ambig(`v'`)` for S and w.

In essence, this lemma says that the interpreter does not reject valid input. Its proof is based on an invariant over the machine state guaranteeing that no machine operation can result in a rejection.

In the absence of semantic predicates, a natural definition of this invariant says that the concatenated unprocessed stack symbols recognize the remaining token sequence. CoStar uses exactly this invariant; its formal definition appears in Figure 3.7. Such an invariant is purely forward-looking; it refers only to symbols and tokens that the interpreter has not processed yet. However, this invariant is too weak to prove that CoStar++ never rejects valid input, because a predicate can fail on semantic values that were produced by earlier machine steps. To rule out such cases, we need an invariant that is both backward- and forward-looking—i.e., one that refers to both the "past" and "future" of the parse.

$\boxed{\textsc{FramesAcceptSuffix\_I} : (\bar{v} : [\![\gamma]\!]), \phi \triangleright w}$

$$\frac{}{\bar{v}, \bullet \triangleright \epsilon}\ \textsc{FramesAcceptSuffix\_Nil}$$

$$\frac{\bar{v}_\gamma : [\![\gamma]\!] \qquad \bar{v}_\alpha : [\![\alpha]\!] \qquad \bar{v}_\beta : [\![\beta]\!] \qquad p : [\![\gamma]\!] \to \mathbb{B} \qquad f : [\![\gamma]\!] \to [\![X]\!] \qquad \beta \xrightarrow{\bar{v}_\beta} w_1 \qquad X ::= \gamma\ [\![p]\!]?\ [\![f]\!]! \in \mathcal{G} \qquad p(\bar{v}_\gamma) = \texttt{true} \qquad \texttt{revTup}(\bar{v}_\alpha)\ [\![+\!+]\!]\ (f(\bar{v}_\gamma), \bar{v}_\beta), \phi \triangleright w_2}{\bar{v}_\gamma, [\alpha\ \&\ \bar{v}_\alpha, X\beta]\phi \triangleright w_1 w_2}\ \textsc{FramesAcceptSuffix\_Cons}$$

$\boxed{\textsc{StackAcceptsSuffix\_I} : \phi \blacktriangleright w}$

$$\frac{\bar{v}_\alpha : [\![\alpha]\!] \qquad \bar{v}_\beta : [\![\beta]\!] \qquad \beta \xrightarrow{\bar{v}_\beta} w_1 \qquad \texttt{revTup}(\bar{v}_\alpha)\ [\![+\!+]\!]\ \bar{v}_\beta, \phi \triangleright w_2}{[\alpha\ \&\ \bar{v}_\alpha, \beta]\phi \blacktriangleright w_1 w_2}$$

Figure 4.12: The StackAcceptsSuffix_I machine state invariant, which guarantees that the interpreter does not reject valid input. The function $[\![+\!+]\!] : [\![\alpha]\!] \to [\![\beta]\!] \to [\![\alpha +\!+ \beta]\!]$ concatenates two semantic tuples.

The CoStar++ completeness invariant, StackAcceptsSuffix_I, appears in Figure 4.12. It holds when the remaining tokens can be split into a prefix w_1 and suffix w_2 such that the unprocessed symbols β in the top stack frame produce a semantic tuple for w_1, and the auxiliary invariant FramesAcceptSuffix_I holds for the lower stack frames ϕ and w_2.

The FramesAcceptSuffix_I definition (also in Figure 4.12) is parametric over a sequence of symbols γ and a semantic tuple $\bar{v} : [\![\gamma]\!]$. The $\bar{v}$ parameter represents the "incoming" tuple during the eventual return operation from the frame above the ones in scope. (The top stack frame has no "incoming" tuple, which is why it appears as a special case in StackAcceptsSuffix_I.) The base case of FramesAcceptSuffix_I says that if the list of remaining frames is empty, then the remaining token sequence must be empty as well. In the case of a non-empty list of frames, the following properties hold:

- The remaining tokens can be split into a prefix w_1 and suffix w_2 such that the unprocessed symbols in the head frame produce a semantic tuple for w_1. This property (which appears in StackAcceptsSuffix_I as well) is the forward-looking portion of the invariant.
- There exists a grammar production $X ::= \gamma\ [\![p]\!]?\ [\![f]\!]!$, where X is the open nonterminal in the head frame and γ is the right-hand side from the frame above, such that semantic tuple $\bar{v}_\gamma$ from the frame above satisfies p. This condition is the backward-looking portion of the invariant.
- The FramesAcceptSuffix_I invariant holds for the remaining frames and w_2.

Lemma 4.5.2 (Completeness invariant prevents rejection)**.** If StackAcceptsSuffix_I holds at machine state σ, then a machine transition out of σ never produces a `Reject` result.

Lemma 4.5.3 (Preservation of completeness invariant)**.** If StackAcceptsSuffix_I holds at machine state σ and $\sigma \rightsquigarrow \sigma'$, then StackAcceptsSuffix_I holds at state σ'.

4.6 Coq Mechanization

In this section, we highlight a few of the thorny dependent type-related issues that arise in implementing CoStar++'s semantic features and mechanizing proofs about these features.

4.6.1 Representing Semantic Grammars

Morally, the grammars that CoStar++ accepts are finite sets of productions. However, we cannot use the standard Coq finite set libraries to implement them because these libraries require set elements to have decidable equality—CoStar++ grammar productions lack this property because they include function components. A reasonable alternative is to represent a grammar as a finite map from unannotated BNF productions to semantic predicates and actions; the "base" productions are suitable map keys because equality of grammar symbols is decidable. However, Coq's standard finite map libraries do not directly support this representation, either; these libraries produce maps with a non-dependent interface, and the type of a given value in our map—e.g., a (predicate, action) pair—depends on its associated key.

We solve this problem by using Coq's non-dependent FMaps library as the basis for an ad hoc dependent map interface. In our implementation, a grammar $\mathcal{G}$ is a finite map in which a production $X ::= \gamma$ maps to the dependent pair of a production $X' ::= \gamma'$ and semantic functions with types that depend on $X' ::= \gamma'$:

$$X ::= \gamma \;\mapsto_{\mathcal{G}}\; (X' ::= \gamma' \;\&\; (\llbracket \gamma' \rrbracket \to \mathbb{B} * \llbracket \gamma' \rrbracket \to \llbracket X' \rrbracket))$$

A grammar is well-formed iff each key is equal to the production in its associated value:

$$\forall p\, p'\, fs,\; p \;\mapsto_{\mathcal{G}}\; (p' \;\&\; fs) \implies p = p'$$

This scheme enables CoStar++ to look up the predicate $p : \llbracket \gamma \rrbracket \to \mathbb{B}$ and action $f : \llbracket \gamma \rrbracket \to \llbracket X \rrbracket$ associated with key $X ::= \gamma$. However, it cannot use the map's standard lookup function directly because the types of p and f are only guaranteed to correspond to the "inner" production (i.e., the one in the key's associated value)—not the key itself. Instead, we wrap the standard map lookup in a function that is parameterized by a proof that the grammar is well-formed:

```
findPredicateAndAction (x   : production)
                       (g   : grammar)
                       (Hwf : grammar_wf g) : option (⟦x⟧_p * ⟦x⟧_a)
```

where $\llbracket \mathtt{x} \rrbracket_p$ and $\llbracket \mathtt{x} \rrbracket_a$ are the types of semantic predicates and actions for production `x`, respectively.

The `findPredicateAndAction` definition relies on an auxiliary function, `castLookupResult`, which uses a proof `Hf` about the result of finding key `x` in a well-formed grammar to produce a predicate and action with the correct types for `x`:

```
castLookupResult (g    : grammar)
                 (x x' : production)
                 (fs   : ⟦x'⟧p * ⟦x'⟧a)
                 (Hwf  : grammar_wf g)
                 (Hf   : find x g = Some (x' & fs)) : ⟦x⟧p * ⟦x⟧a
```

The `castLookupResult` function uses the fact that productions `x` and `x'` are definitionally equal to safely cast the predicate and action for `x'` to a predicate and action for `x`.

A crucial fact for reasoning about `castLookupResult` (and the higher-level functions that rely on it) is that if two pairs of semantic functions are equal up to the cast operation, then they are definitionally equal:

Lemma 4.6.1 (Reflexivity of `castLookupResult`)**.** For grammar `g`, production `x`, predicate/action pairs `fs` and `fs'`, grammar well-formedness proof `Hwf`, and proof `(Hf : find x g = Some (x & fs))`, if `castLookupResult g x x fs Hwf Hf` = `fs'`, then `fs` = `fs'`.

In general, one cannot construct an axiom-free proof of this fact in Coq; it is analogous to proving that the right projections of two equal dependent pairs are equal, which requires an axiom of proof irrelevance. However, the Coq standard library includes an `Eqdep_dec` module that establishes the unicity of equality proofs for decidable types (the formalization is based on a proof by Hedberg [1998]). In other words, the module proves without axioms that there is only one proof of the form $x = x$ when x has decidable equality. Because base productions—the left projections of our dependent pairs—are composed of grammar symbols and thus have decidable equality, we are able to use lemmas from `Eqdep_dec` to prove Lemma 4.6.1 without axioms.

4.6.2 Equalities in Dependently Typed Invariants

The CoStar++ soundness theorems (4.3.1 and 4.3.2) rely on a machine state invariant about the soundness of partial derivations. In essence, the invariant says that in each stack frame, the processed symbols derive a portion of the full token sequence, producing the semantic values stored in that frame. CoStar++'s semantic features make the preservation proofs for these invariants significantly more involved than the analogous proofs for CoStar, which produces parse trees.

For CoStar, the preservation proof for partial soundness includes subgoals like this one:

$$\frac{\gamma \texttt{ : list symbol} \quad w \texttt{ : list token} \quad \bar{t} \texttt{ : list tree} \qquad \texttt{rev (rev } \gamma\texttt{)} \xrightarrow{\texttt{rev (rev } \bar{t}\texttt{)}} w}{\gamma \xrightarrow{\bar{t}} w}$$

Subgoals like this one arise because CoStar accumulates values in a stack frame in reverse order for efficiency, and then puts them in the correct order by reversing them during a return operation.

Proving these subgoals is straightforward because `rev (rev` γ`)` and `rev (rev` $\bar{t}$`)` rewrite to γ and $\bar{t}$, respectively.

The semantic version of this subgoal is as follows:

$$\frac{\gamma \texttt{ : list symbol} \qquad w \texttt{ : list token} \qquad \bar{v} \texttt{ : } [\![\gamma]\!] \qquad \texttt{rev (rev } \gamma\texttt{)} \xrightarrow{\texttt{revTup (rev } \gamma\texttt{) (revTup } \gamma\ \bar{v}\texttt{)}} w}{\gamma \xrightarrow{\bar{v}} w}$$

where the `revTup` function reverses a dependently-typed semantic tuple:

$$\texttt{revTup (}\gamma\texttt{ : list symbol) (}\bar{v}\texttt{ : } [\![\gamma]\!]\texttt{) : } [\![\texttt{rev } \gamma]\!]$$

Now we cannot simply rewrite `revTup (rev` γ`) (revTup` $\gamma\ \bar{v}$`)` to $\bar{v}$ anymore because the former value has type $[\![$`rev (rev` γ`)`$]\!]$ and the latter value has type $[\![\gamma]\!]$. In other words, the two types depend on values that are computationally equal—we can prove that `rev (rev` γ`) =` γ—but not syntactically equal.

Instead, we have to define a cast operation that lets us treat a value of one type as a value of the other type, given a proof that the values they depend on are equal:

```
castTuple (xs ys : list symbol) (Heq : xs = ys) (vs : ⟦xs⟧) : ⟦ys⟧
```

Now we can prove a lemma that lets us replace a derivation judgment with another one in which the symbols and semantic values are computationally (but not necessarily syntactically) equal to those in the first judgment:

SemDerivationEq

$$\frac{\alpha\ \beta \texttt{ : list symbol} \qquad w \texttt{ : list token} \qquad \bar{v}_\alpha \texttt{ : } [\![\alpha]\!] \qquad \bar{v}_\beta \texttt{ : } [\![\beta]\!] \qquad H \texttt{ : } \alpha = \beta \qquad \bar{v}_\beta \texttt{ = castTuple } \alpha\ \beta\ H\ \bar{v}_\alpha \qquad \alpha \xrightarrow{\bar{v}_\alpha} w}{\beta \xrightarrow{\bar{v}_\beta} w}$$

and we can prove that `revTup (rev` γ`) (revTup` $\gamma\ \bar{v}$`) =` $\bar{v}$ up to the cast operation:

revTuple_Involutive

$$\frac{\gamma \texttt{ : list symbol} \qquad \bar{v} \texttt{ : } [\![\gamma]\!] \qquad H \texttt{ : } \gamma \texttt{ = rev (rev } \gamma\texttt{)}}{\texttt{revTup (rev } \gamma\texttt{) (revTup } \gamma\ \bar{v}\texttt{) = castTuple } \gamma\ \texttt{(rev (rev } \gamma\texttt{))}\ H\ \bar{v}}$$

With these two lemmas, we can prove the problematic subgoal above.

4.7 Interlude: Verified Lexing

Like Vermillion and CoStar, CoStar++ operates on tokenized input. To evaluate the performance of the former two tools, we used ad hoc tokenization strategies: a Menhir-based prepro-

Character	$c \in \Sigma$
String	$z ::= \lambda \mid cz$
Regex	$e ::= \emptyset \mid \varepsilon \mid (\!\lvert c \rvert\!) \mid e + e \mid e \cdot e \mid e^*$
Rule	$r \in \mathcal{T} \times e$
Token	$t \in \mathcal{T} \times z$

Figure 4.13: Definition of strings, regular expressions, lexical rules, and tokens over an alphabet Σ. We write non-empty strings without a terminal λ for brevity: for example, c instead of $c\lambda$. In Section 4.7.4, we describe how Verbatim converts the simply-typed "tagged string" tokens shown here to the dependently-typed semantic tokens that CoStar++ consumes.

cessor in the case of Vermillion (see Section 2.5) and an ANTLR-based one for CoStar (Section 3.6). In the CoStar++ evaluation, we take a more principled approach to tokenization by using Verbatim, a Coq-verified lexer interpreter that we developed in a separate line of research. Verbatim is designed to be interoperable with CoStar++. Together, the two tools constitute a fully verified pipeline for lexing and parsing.

Verbatim takes a list of lexical rules and a string as input. It uses a technique for regular expression (regex) matching based on the concept of Brzozowski derivatives [Brzozowski 1964] to tokenize the string, producing semantic tokens with the type that CoStar++ accepts. The resulting tokens satisfy a standard lexer specification known as the "maximal munch" principle [Cattell 1978]: each token is the longest prefix of the remaining input string that matches a lexical rule.

Before describing the CoStar++ performance evaluation in Section 4.8, we divert briefly to outline Verbatim's tokenization strategy, correctness properties, performance characteristics, and compatibility with CoStar++. This section is based on two publications that Derek Egolf, Kathleen Fisher, and I coauthored [Egolf et al. 2021, 2022], with Derek as the first author of both works. The Verbatim development is open source and available online [Egolf 2021].

4.7.1 Background: Regex Matching with Brzozowski Derivatives

Regexes represent regular languages. If a string z is in the language that regex e represents, we say that z matches e and write $z \simeq e$. We use the inductive definitions of regexes (Figure 4.13) and regex matching (Figure 4.14) from the Software Foundations textbook on interactive theorem proving in Coq [Pierce et al. 2018].

Verbatim represents a lexical rule as a (terminal, regex) pair. A string z matches a rule (a, e) iff $z \simeq e$; we write $z \simeq (a, e)$ to represent such a match.

Verbatim's regex matching algorithm is based on the concept of Brzozowski derivatives [Brzozowski 1964]. Intuitively, the derivative of a language (set of strings) $\mathcal{L}$ with respect to character c deletes the leading c from the strings in $\mathcal{L}$ that begin with c, and it removes the other strings entirely:

$$\partial_c \mathcal{L} = \{z \mid cz \in \mathcal{L}\}$$

$$\text{(MEmpty)}\quad \lambda \simeq \varepsilon \qquad \text{(MChar)}\quad c \simeq (\!|\, c \,|\!) \qquad \text{(MApp)}\quad \frac{z_1 \simeq e_1 \qquad z_2 \simeq e_2}{z_1 +\!\!+ z_2 \simeq e_1 \cdot e_2} \qquad \text{(MUnionL)}\quad \frac{z \simeq e_1}{z \simeq e_1 + e_2}$$

$$\text{(MUnionR)}\quad \frac{z \simeq e_2}{z \simeq e_1 + e_2} \qquad \text{(MStar0)}\quad \lambda \simeq e^* \qquad \text{(MStarApp)}\quad \frac{z_1 \simeq e \qquad z_2 \simeq e^*}{z_1 +\!\!+ z_2 \simeq e^*}$$

Figure 4.14: Inductive relation for string-regex matching, where a string is a sequence of symbols from alphabet Σ and $z_1 +\!\!+ z_2$ is the concatenation of strings z_1 and z_2.

The derivative operation can be extended to strings recursively (where λ is the empty string):

$$\partial_\lambda \mathcal{L} = \mathcal{L}$$
$$\partial_{cz} \mathcal{L} = \partial_z(\partial_c \mathcal{L})$$

For string z and regular language $\mathcal{L}$, we can prove by induction on z that:

$$z \in \mathcal{L} \iff \lambda \in \partial_z \mathcal{L}$$

Because regexes represent regular languages, one can extend the concept of a derivative from a regular language to a regex: if regex e represents language $\mathcal{L}$, then $\partial_c e = e'$ represents $\partial_c \mathcal{L}$ (and by extension, $\partial_z e$ represents $\partial_z \mathcal{L}$). The algorithm in Figure 4.15a computes the derivative of a regex with respect to character c. This algorithm gives rise to an elegant technique for determining whether string z matches regex e: compute $\partial_z e$ and check whether the resulting regex is nullable.

4.7.2 Lexer Specification and Correctness Properties

A standard lexer specification is the maximal munch principle [Cattell 1978]. This principle says that the result of tokenizing an input string z is correct when each token is a "maximal munch" of the remaining string—i.e., the longest prefix of z that matches a lexical rule in the list of rules R. When two rules match the same maximal prefix, the rule that appears first in R is used to produce the token. Specifically, token (a, p) is a maximal munch of string z when the following conditions hold:

- String p matches a lexical rule $(a, e) \in R$.
- String p is the longest prefix of z that matches any lexical rule in R.
- For all $r' \neq (a, e) \in R$, if p matches r', then r' appears after (a, e) in R. In other words, if multiple rules match p, then the tie goes to the rule that appears first in R.

Verbatim is sound and complete with respect to this maximal munch specification: given a list of lexical rules and an input string, the tool produces a token sequence $\bar{t}$ iff $\bar{t}$ is a correct maximal munch tokenization of the input.

$$
\begin{aligned}
\partial_c \emptyset &:= \emptyset \\
\partial_c \varepsilon &:= \emptyset \\
\partial_c (\!| c' |\!) &:= \texttt{if } c = c' \texttt{ then } \varepsilon \texttt{ else } \emptyset \\
\partial_c (e_1 + e_2) &:= \partial_c e_1 + \partial_c e_2 \\
\partial_c (e_1 \cdot e_2) &:= (\partial_c e_1 \cdot e_2) + (\texttt{if nullable } e_1 \texttt{ then } \partial_c e_2 \texttt{ else } \emptyset) \\
\partial_c (e^*) &:= \partial_c e \cdot e^*
\end{aligned}
$$

(a) Algorithm for computing the derivative of a regex with respect to character c. The `nullable` function is defined below.

$$
\begin{aligned}
\texttt{nullable } \emptyset &:= \texttt{false} \\
\texttt{nullable } \varepsilon &:= \texttt{true} \\
\texttt{nullable } (\!| c |\!) &:= \texttt{false} \\
\texttt{nullable } (r_1 + r_2) &:= \texttt{nullable } r_1 \vee \texttt{nullable } r_2 \\
\texttt{nullable } (r_1 \cdot r_2) &:= \texttt{nullable } r_1 \wedge \texttt{nullable } r_2 \\
\texttt{nullable } r^* &:= \texttt{true}
\end{aligned}
$$

(b) Algorithm for determining whether a regex is nullable. The expression `nullable` r evaluates to `true` if $\lambda \simeq r$ and `false` otherwise.

Figure 4.15: The derivative operation for regular expressions.

4.7.3 Optimizations

The running time of the initial Verbatim prototype [Egolf et al. 2021] is quadratic in the length of the input string. In subsequent work [Egolf et al. 2022], we present a suite of optimizations that improves the tool's performance in terms of both its asymptotic complexity—$O(n \log n)$ for the final version of the tool—and its speed on a JSON lexing benchmark. Each optimization provably preserves the correctness guarantees of the earlier prototype. The optimizations are as follows:

- **Compiling regexes to deterministic finite automata (DFAs) for faster matching:** We replace Verbatim's original regex matcher, which computes Brzozowski derivatives at runtime, with a DFA-based matcher. This process involves converting the regex in each lexical rule to a DFA, which we accomplish with a modified version of Brzozowski's derivative-based conversion algorithm [Brzozowski 1964].

- **Memoizing lexer subroutine calls to avoid redundant computations:** In the process of lexing string z with respect to rule r, Verbatim repeatedly calls a function called `maxpref_one` (which finds the maximal prefix of z that matches r) with the same arguments. By memoizing (caching) the results of calls to `maxpref_one`, we avoid making redundant calls. A cache M is represented as a two-dimensional table in which rows are labeled with regexes and columns are labeled with suffixes of the input string. Before recursively calling `maxpref_one` with arguments r and s, the lexer checks whether $M[r][s]$ is populated; if it is, the lexer uses the cached result from that cell instead of making the recursive call.

- **Replacing memoized strings with string *lengths* for faster cache lookups:** Because each suffix of the lexer's original input string has a unique length, we can use suffix lengths as indices into the cache instead of suffixes themselves. This change enables us to implement the cache as a binary trie—a data structure that supports efficient lookups for binary representations of keys. Binary trie lookups are $O(\log n)$, where key n is a binary number. Note that this running time depends only on the size of the key, and not on the number of entries in the trie. In addition, a lookup operation for key n does not require comparisons between keys—the lookup algorithm simply reads n bit-by-bit. Because Verbatim's memoization strategy can produce a cache with many entries and large keys, representing the cache as a binary trie provides a significant performance benefit.

4.7.4 Semantic Actions

The version of Verbatim described thus far uses a simple "tagged string" token representation, in which each token consists of a terminal label and a portion of the input string. If CoStar++ consumed such tokens, it would need to convert the string components into values of other types to produce a semantic value with an expressive type. Morally, this functionality should be part of the lexer—all of the information required to perform these conversions is available during lexical analysis. As a final step in the development of Verbatim, we enable the user to augment a lexical specification with semantic actions that map a "tagged string" token (a, p) to a semantic token $(a \mathbin{\&} v : [\![a]\!])$ with the type that CoStar++ accepts as input.

4.8 Performance Evaluation

We evaluated CoStar++'s parsing speed and asymptotic behavior by extracting the tool to OCaml source code and recording its execution time on benchmarks for four real-world data formats. The benchmarks exercise CoStar++'s semantic features, and they demonstrate that the tool is capable of handling non-context-free semantic components of language specifications that arise in practice. CoStar++ runs in linear time on all four benchmarks.

In each experiment, we provide CoStar++ with a grammar for a data format to obtain a parser for that format, and record the parser's execution time on valid inputs of varying size. The benchmarks are as follows:

- **JSON** is a popular format for storing and exchanging structured data. The actions in our JSON grammar build an ADT representation of a JSON value with a type similar to the one in Figure 4.1. The predicates ensure that JSON objects contain no duplicate keys—our running example of a non-context-free semantic specification from Section 4.2. The JSON data set consists of biographical information for current US Members of Congress (as of March 2022); it comes from a repository developed by GovTrack.us [GovTrack contributors 2022], a non-governmental source of legislative data.

- **PPM** is a text-based image file format in which each pixel is represented by a triple of (red, green, blue) values. A PPM file includes a header with numeric values that specify the im-

age's width and height, and the maximum value of any pixel component (i.e., any color value). The actions in our PPM grammar build a record that contains the header values and a list of pixels. The predicates validate the non-context-free data dependencies between the image's header and pixels. We generated a PPM data set for the evaluation by using the ImageMagick command-line tool `convert` to convert a single PPM image to a range of different sizes.

- **Newick trees** are an ad hoc format for representing arbitrarily branching trees with labeled edges. They are used in the evolutionary biology community to represent phylogenetic relationships—nodes represent biological species and edge lengths represent genetic distances. The Newick grammar's actions convert an input to an ADT representation of an arbitrarily branching tree. Our Newick data set comes from the 10kTrees Website, Version 3 [Arnold et al. 2010], a public database of phylogenetic trees for various mammalian orders.

- **XML** is a widely used markup language and format for storing and transmitting structured data. An XML document is a tree of elements; each element begins and ends with a string-labeled tag, and the labels in corresponding start and end tags must match—a non-context-free property in the general case where the set of valid labels is infinite. The actions in our XML grammar build an ADT representation of an XML document, and the predicates check that corresponding tags contain matching labels. Our XML data set is a portion of the Open American National Corpus [OANC 2010], a collection of English texts with linguistic annotations.

CoStar++ requires tokenized input. We use the Verbatim verified lexer interpreter [Egolf et al. 2021, 2022] (described in Section 4.7) to write lexical specifications for all four formats and obtain lexers from those specifications. In our benchmarks, we use these lexers to pre-tokenize each input before supplying it to the parser.

We ran the CoStar++ benchmarks with the same hardware and compiler settings as those used in the CoStar benchmarks (Section 3.6). The test machine was a laptop with 4 2.5 GHz cores, 7 GB of RAM, and the Ubuntu 16.04 OS. We compiled the extracted CoStar++ source code with OCaml compiler version 4.11.1+flambda at optimization level -O3.

The CoStar++ benchmark results appear in Figure 4.16. Each scatter plot point represents the parse time for one input file, averaged over ten trials. Like the reported performance results for CoStar (Figure 3.9), the CoStar++ results suggest linear performance. To provide further evidence of linearity, we compute a least-squares regression line and a Locally Weighted Scatterplot Smoothing (LOWESS) curve [Cleveland 1979] for each set of benchmark results. LOWESS (also described in Section 3.6) is a non-parametric technique for fitting a smooth curve to a set of data points—i.e., it does not assume that the data fit a particular distribution, linear or otherwise. The LOWESS curve and regression line correspond closely for each set of results, indicating that the relationship between input size and execution time is linear.

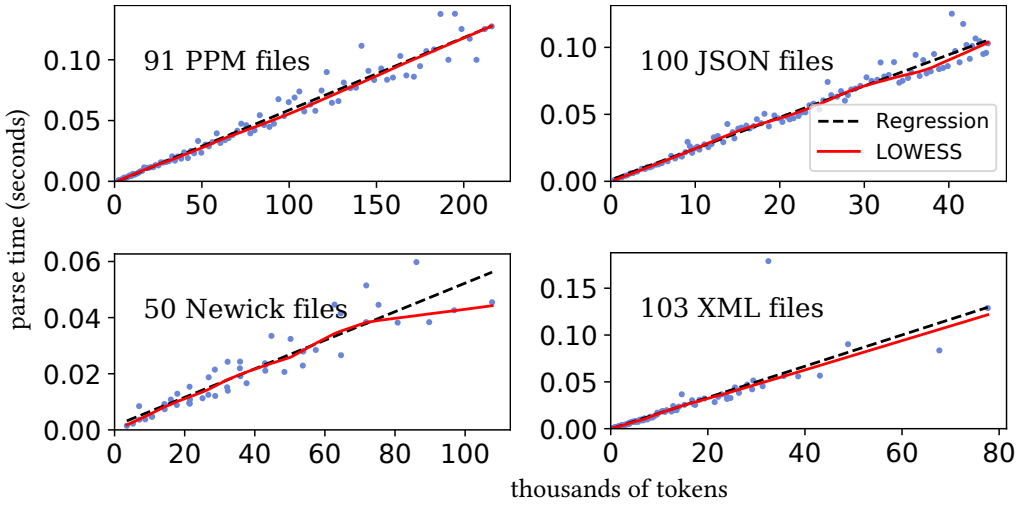


Figure 4.16: Input size vs. CoStar++ average execution time on four benchmarks. Regression lines coincide with unconstrained LOWESS curves, indicating that CoStar++ runs in linear time on the benchmarks.

4.9 Related Work

CoStar++ builds on CoStar, the version of the tool described in Chapter 3. CoStar produces concrete syntax trees that are generic across grammars modulo grammar symbol names. It is correct in terms of a specification in which concrete syntax trees are witnesses to a successful derivation. CoStar++ improves upon this work by supporting semantic actions and predicates. Actions give the user fine-grained control over the interpreter's output type, and predicates enable the user to specify language definitions with non-context-free components. CoStar++ is also correct with respect to a semantics-aware specification in which the witness to a successful derivation is a well-formed semantic value—i.e., a value that is produced by grammar actions and that satisfies grammar predicates.

ALL(*) was originally developed to serve as the algorithmic core of the ANTLR parser generator [Parr et al. 2014]. While ALL(*) as originally described and as implemented in ANTLR supports a notion of semantic predicates, its prediction mechanism does not execute semantic actions, and thus cannot evaluate predicates over the results of those actions. The original algorithm is therefore incomplete with respect to our predicate-aware correctness specification. These original design choices are reasonable in terms of efficiency, and in terms of correctness in the imperative setting for which ALL(*) was originally designed. It is potentially expensive to execute predicates and actions along a prediction path that the parser does not ultimately take. More importantly, doing so can produce counterintuitive behavior when the actions alter mutable state in ways that cannot be easily undone. These concerns do not apply to our setting, in which semantic actions are pure functions—there can be no "launch the missiles" action in a CoStar++ grammar! We also believe that having CoStar++ fail when a well-formed semantic value exists for the input could lead to difficult-to-debug behavior, just like speculatively executing actions in the imperative setting. The modified version of ALL(*) prediction described in Section 4.5.2 strikes a balance between fast performance on real grammars and completeness in the presence of semantic

predicates.

Several existing verified parsers for CFGs support some form of semantic actions. Jourdan et al. [2012] present a verified tool for validating LR(1) parsers. The tool represents a semantic action as a function with a dependent type computed from the grammar symbols in its associated production. Both Vermillion (Chapter 2) and CoStar++ use a similar representation of actions. Edelmann et al. [2020] describe a parser combinator library with an accompanying type system that ensures that any well-typed parser built from the combinators is equivalent to an LL(1) grammar. Danielsson [2010] and Ridge [2011] present similar parser combinator libraries that are compatible with arbitrary CFGs but do not provide the linear runtime guarantees of LL(1).

Note that unlike CoStar++, the tools described above do not have built-in support for semantic predicates as first-class entities. While it is possible to encode predicates in semantic actions—e.g., by making them produce an `option` value—this approach does not necessarily have the same semantics as CoStar++ predicates. For example, it may not prevent the parser from returning `None` when a different sequence of parsing decisions would have produced a `Some` result. CoStar++ helps grammar writers clearly distinguish the code that produces semantic values from the code that validates those values—we believe this distinction leads to language definitions that are easy to read and write.

In terms of unverified work, PADS [Fisher and Gruber 2005] is a tool for processing ad hoc data that consumes a user-defined data format description and generates a structured representation of the format, a parser for that format, and other utilities. PADS enables the user to describe formats with data dependencies—for example, the requirement that an integer field in a packet header is equal to the number of bytes in the packet's data payload. YAKKER [Jim et al. 2010] is a parser that uses a modified version of Earley's parsing algorithm to handle grammars that may include data dependencies. These tools approach data dependencies and other non-context-free input properties differently than CoStar++. They can record the results of previous parser actions (e.g., in a state monad) and use them in later parsing decisions—for example, parse an integer n from a packet header and then read n bytes from the payload—whereas CoStar++ must parse the header, parse the payload, and then check that the dependency holds between the two packet components. We believe that the CoStar++ approach occupies a "sweet spot" in the space of semantically rich parsing techniques: it has an intuitive correctness specification; it simplifies reasoning about grammar semantics because predicates and actions are local to a production (they can only refer to values constructed with that production); and it is expressive enough to capture the kinds of non-context-free features that arise in real data formats.

4.10 Conclusion

In this chapter, we have presented CoStar++, a verified parser interpreter that provides support for dependently typed semantic actions and predicates. These features give the interpreter a high degree of expressivity in terms of both the language specifications it can accept and its output type. They also give rise to several interesting verification challenges: among others, specifying the interpreter's behavior on ambiguous input and ensuring that the interpreter offers completeness

guarantees in the presence of semantic predicates. CoStar++ is provably correct with respect to a high-level specification that describes the behavior of actions, predicates, and the tool's ambiguity detection mechanism. Finally, CoStar++ can be integrated with a verified lexer interpreter, and it runs in linear time on a range of grammars for commonly used data formats that have non-context-free specifications. We believe that CoStar++ is an attractive point in the space of formally verified parsing because of its expressiveness, its empirical performance, and the strength of its correctness guarantees.

5

Other Related Work

5.1 Parsing and Software Security

Parser vulnerabilities and their security consequences are well-documented. In this section, we describe several notable vulnerabilities from the recent past, and we summarize several more exhaustive surveys of real-world parser bugs.

- The Heartbleed bug enabled silent exfiltration of sensitive data from network-connected systems [Durumeric et al. 2014]. The bug was an input validation error in the OpenSSL implementation of the TLS protocol's heartbeat extension. A heartbeat message includes a padding field, which must have a specific length. The correct length is determined by a formula that includes the message's payload size and the protocol version (TLS or DTLS). OpenSSL failed to correctly validate the padding length; as a result, an attacker could craft a message in which the amount of padding differs from what the formula dictates. Such a message could induce the recipient to include extra (and potentially private) data in its response.

- The Psychic Paper vulnerability [Siguza 2020] enabled attackers to escape the iOS sandbox. The sandbox used an XML-like format called property lists (plists) to represent an application's entitlements: functions that the app is allowed to perform (similar to Unix permissions). The iOS sandbox code included four different plist parsers; these parsers produced equal results when given well-formed input, but produced divergent results on erroneous input. One of these parsers ran at application startup time to determine which entitlements the app required, and a separate parser was used when the running app sought to perform actions governed by the entitlements. A malformed plist could cause the first parser to return an empty list of permissions, and then cause the second parser to read actual entitlements from the plist, thus allowing the app to perform actions that should not have passed the initial vetting step.

  Using multiple parsers on the same input has created vulnerabilities on other platforms as well. The Android Master Key bugs [Freeman 2013a,b,c] arose because two Android utilities—a cryptographic signature verifier and an installer—could parse the same appli-

cation package differently. An attacker could thus trick the system into installing a package different from the one it had verified.

- The Equifax breach [Goodin 2017; Mort 2017] exposed the personal information of nearly 150 million people, resulting in a $425M settlement for the affected consumers. The attackers achieved full remote code execution on the Equifax credit agency's servers. The breach was made possible by a flaw in the Apache Struts Jakarta Multipart parser. The parser would (correctly) throw an exception when given HTTP content with an invalid `Content-Type` value, and then pass the value to a Java utility that would build an error message. The attackers crafted input that contained Object-Graph Navigation Library (OGNL) expressions in the `Content-Type` field; OGNL is an expression language for Java. When the error message builder received this input, it evaluated the OGNL program fragment. The OGNL code was able to perform system calls and launch external processes, thus enabling the attackers to execute code remotely and exfiltrate private data.

The vulnerabilities listed above are only a small sample of the parsing bugs that have been identified in the wild. A survey of XML parser vulnerabilities [Späth et al. 2016] identified 165 bugs across 33 different parsers. In addition, a study of bugs in the OpenSSL cryptography library found that out of 47 vulnerabilities, 13 were related to shotgun parsing (i.e., interleaving parsing and validation), and 11 were related to processing invalid input or failing to reject input known to be invalid [Momot et al. 2016]. In other words, more than half of the vulnerabilities were the result of faulty parsing! Taken together, these studies suggest that parser vulnerabilities are ubiquitous, and that their security consequences are severe.

5.2 Alternative Approaches to Verified Parsing

Vermillion and CoStar are based on "top-down with lookahead" CFG parsing algorithms. Other families of CFG-based parsing algorithms exist as well, and CFGs are not the only formalism used to specify parsers. In this section, we survey alternative parsing approaches that have been formally verified.

5.2.1 Bottom-Up CFG-Based Parsers

Conceptually, a bottom-up parsing algorithm builds a parse tree from the leaves upwards (although in practice, like Vermillion and CoStar, it might produce a semantic value without materializing the parse tree). Barthwal and Norrish [2009] use the HOL4 proof assistant to prove the soundness and completeness of a parser interpreter based on the SLR parsing algorithm, a bottom-up algorithm suitable for an unambiguous subset of CFGs. The tool consists of a generator that converts a grammar to an automaton and a parse function parameterized by such an automaton—a similar structure to that of Vermillion. The work does not include performance results, but it states that part of the tool's operation involves computing DFA states on the fly (rather than statically) for ease of verification, and that this approach is likely to hinder performance.

Jourdan et al. [2012] use Coq to verify a validator that determines whether a generated LR(1) parser is sound and complete with respect to the generator's input grammar. *A posteriori* validation is a flexible and lightweight alternative to full verification; the validator is compatible with untrusted generators, and its formalization is small.

Neither development guarantees that its parsers terminate on invalid inputs. The parsers therefore cannot be viewed as decision procedures for language membership. In their discussion of this issue, Jourdan et al. suggest that identifying a termination measure for non-canonical LR(1) parsers is an open problem.

Note that in the context of a total programming language like the one that Coq supports, a program cannot literally fail to terminate. The LR(1) interpreter, for example, uses a "fuel" parameter to ensure termination—i.e., a natural number that the interpreter decrements with each step that it takes. When the fuel is exhausted, the interpreter returns an "out of fuel" result that does not indicate acceptance or rejection of the input. In this context, proving termination on invalid input would mean proving a lower bound on the amount of fuel required for the interpreter to produce a "reject" result instead of an "out of fuel" result—the LR(1) tool lacks such a guarantee.

One commonly cited drawback of bottom-up parsers is that they often report errors in terms of a compiled representation of the grammar (i.e., an automaton) instead of in terms of the original grammar, and that these reports are hard to understand as a result. In fact, getting bottom-up parsers to produce informative error messages is considered difficult enough to be a research area in its own right [Jeffery 2003; Pottier 2016]. Bottom-up algorithms are also not well-suited to handling non-context-free extensions to grammars, such as data dependencies [Fisher and Gruber 2005].

5.2.2 Parsers for General CFGs

There have been several successful efforts to verify parsers for general CFGs. Ridge [2011] 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 unverified implementation that includes an additional memoization component has $O(n^5)$ worst-case time complexity. Firsov and Uustalu [2014] describe a verified Agda implementation of the CYK algorithm, which operates on CFGs in Chomsky normal form (CNF). In subsequent work [Firsov and Uustalu 2015], they verify a CNF normalization algorithm. The combined result is a verified parser for arbitrary CFGs. Danielsson [2010] 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.

General parsing algorithms are designed to be compatible with ambiguous grammars, and they typically return all parse trees or semantic values for a given input. These properties may be undesirable in a setting where parsing is expected to be unambiguous and fast. The works listed above include few or no performance results, suggesting that performance is not a high priority.

5.2.3 Parsing Expression Grammar-Based Parsers

Parsing Expression Grammars (PEGs) [Ford 2004] are a language representation that is sometimes used in place of CFGs to specify parsers. Among other differences, a PEG instance can include an ordered choice operator (written "/") that prioritizes one grammar subexpression over another. In contrast, choice in CFGs is always unordered—any matching production may be used in a derivation. The ordered choice operator can lead to difficult-to-debug parsing behavior. Famously, PEGs exhibit the $a \ / \ ab$ quirk in which the second of these two ordered expressions never matches a string—even string ab—because the higher-priority expression immediately matches any string beginning with a.

Koprowski and Binsztok [2010] present a Coq formal semantics for PEGs and prove the correctness of a PEG interpreter with respect to this semantics. They also ensure that the interpreter terminates on both valid and invalid inputs by rejecting grammars that fail a syntactic check for left recursion. PEG parsers often employ a memoization technique called packrat parsing [Ford 2002] to achieve linear time and space complexity. Wisnesky et al. [2009] and Blaudeau and Shankar [2020] use the Ynot and PVS verification frameworks, respectively, to verify packrat PEG parsers.

5.2.4 Parsers for Binary Formats

Parsers for binary message formats target a different range of applications than CFG-based parsers. Binary formats such as those used for network communication are often simple and highly constrained; for example, instances typically have a fixed size and lack recursive structure. On the other hand, binary parsers must be able to process high volumes of incoming messages efficiently, and binary formats often include non-context-free features such as a dependency between a packet's length field and the length of its payload.

EverParse [Ramananandro et al. 2019] is a tool that generates verified parsers and serializers for binary message formats. The user provides a format description in a high-level domain-specific language (DSL), and an untrusted compiler translates the description into (1) a formal specification, and (2) F* source code for parsing and serializing messages, built with combinators from a verified parser combinator library called LowParse. The system then automatically verifies the generated code with respect to the formal specification, and uses a tool called KreMLin [Protzenko et al. 2017] to extract the code to C. Impressively, this extracted C code is competitive with hand-written code in terms of performance.

Narcissus [Delaware et al. 2019] uses a Coq library of verified combinators to generate parsers and serializers from a binary format specification. The generated parser and serializer for a given format are guaranteed to be inverses of each other. In contrast to the low-level code that EverParse produces, Narcissus-generated code is purely functional and higher-order. In some cases, Narcissus also requires the user to solve proof goals interactively. Ye and Delaware [2019] build on Narcissus to produce a fully verified compiler for the Protocol Buffer serialization format. The tool automatically generates code for converting between structured data and binary representations, as specified by the Protocol Buffer standard.

6

Conclusion

This dissertation has presented Vermillion and CoStar, two verified parser interpreters that represent attractive points in the verified parsing space because of their expressiveness, their demonstrated performance, and the strength of their termination and correctness guarantees. They serve as strong evidence for our thesis statement from Chapter 1:

Interactive theorem proving techniques support the design of top-down parser interpreters that are correct with respect to high-level specifications; that terminate without error for well-defined classes of grammars; that run in linear time on a range of grammars for real-world programming languages and data formats; and that give the user fine-grained control over both the language that the interpreter accepts and the interpreter's output type.

In terms of *correctness*, Vermillion and CoStar are sound and complete with respect to high-level relational specifications, and CoStar's ambiguity detection mechanism is also provably correct.

In terms of *termination*, the Vermillion parse table generator terminates without error on all grammars, and Vermillion's implementation of the LL(1) parsing algorithm terminates without error whenever it is supplied with a valid LL(1) parse table—i.e., on any table that the generator produces. CoStar provably terminates without error when its input grammar contains no left recursion.

In terms of *performance*, Vermillion's execution on a JSON benchmark empirically lives up to the LL(1) algorithm's linear-time guarantees. Despite the ALL(*) algorithm's higher worst-case complexity, CoStar also runs in linear time on a range of grammars for popular languages and data formats, including grammars that are not LL(1) and grammars with non-context-free extensions.

Finally, in terms of *expressiveness*, Vermillion and CoStar use dependently typed semantic actions to construct values with user-defined types. CoStar also supports non-context-free language specifications in the form of grammars annotated with semantic predicates. The tool uses these predicates at runtime to enforce the non-context-free components of the specification.

Some of the takeaways from this work extend beyond the verified parsing field. For example, parsing algorithms are canonical examples of functions with complex non-structural termination measures; our work on implementing and verifying such functions in a total setting might

serve as a useful starting point for similar future efforts. The CoStar development also exemplifies an invariant-based approach to proving correctness properties of a stack-based interpreter—in particular, one that uses a dependently typed representation of its state.

We believe that the work presented in this dissertation is a useful step towards the goal of making provably secure parsing ubiquitous.

Bibliography

Michael D. Adams, Celeste Hollenbeck, and Matthew Might. On the Complexity and Performance of Parsing with Derivatives. Programming Language Design and Implementation (PLDI), 2016. URL https://doi.org/10.1145/2908080.2908128.

Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998. ISBN 0-521-58274-1.

Andrew W. Appel. *Verified Functional Algorithms*. Software Foundations series, Volume 3. 2021. URL https://softwarefoundations.cis.upenn.edu.

Christian Arnold, Luke J Matthews, and Charles L Nunn. The 10kTrees Website: A New Online Resource for Primate Phylogeny. *Evolutionary Anthropology: Issues, News, and Reviews*, 19(3): 114–118, 2010.

Aditi Barthwal and Michael Norrish. Verified, Executable Parsing. European Symposium on Programming (ESOP), 2009. URL https://doi.org/10.1007/978-3-642-00590-9_12.

Clement Blaudeau and Natarajan Shankar. A Verified Packrat Parser Interpreter for Parsing Expression Grammars. Certified Programs and Proofs (CPP), 2020. URL https://doi.org/10.1145/3372885.3373836.

Sandrine Blazy and Xavier Leroy. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, RFC Editor, 2017. URL https://doi.org/10.17487/RFC8259.

Janusz A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494, 1964. URL https://doi.org/10.1145/321239.321249.

R. G. G. Cattell. *Formalization and Automatic Derivation of Code Generators*. PhD thesis, Carnegie Mellon University, 1978.

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

The Coq Development Team. The Coq Proof Assistant, version 8.9.0, January 2019. URL https://doi.org/10.5281/zenodo.2554024.

The Coq Development Team. The Coq Proof Assistant, version 8.11.0, January 2020. URL https://doi.org/10.5281/zenodo.3744225.

Nils Anders Danielsson. Total Parser Combinators. International Conference on Functional Programming (ICFP), 2010. URL https://doi.org/10.1145/1863543.1863585.

DARPA. Broad Agency Announcement: Safe Documents (SafeDocs), August 2018. URL https://sam.gov/opp/fa896cc409f0dba9f1f5409fda1aacc9/view.

Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. Narcissus: Correct-by-Construction Derivation of Decoders and Encoders from Binary Formats. 2019. URL https://doi.org/10.1145/3341686.

Christian Doczkal and Damien Pous. Graph Theory in Coq: Minors, Treewidth, and Isomorphisms. *Journal of Automated Reasoning*, 64(5):795–825, 2020. URL https://doi.org/10.1007/s10817-020-09543-2.

Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The Matter of Heartbleed. Internet Measurement Conference (IMC), 2014. URL https://doi.org/10.1145/2663716.2663755.

Romain Edelmann, Jad Hamza, and Viktor Kunčak. Zippy LL(1) Parsing with Derivatives. Programming Language Design and Implemenation (PLDI), 2020. URL https://doi.org/10.1145/3385412.3385992.

Derek Egolf. Verbatim source code, 2021. URL https://github.com/egolf-cs/Verbatim.

Derek Egolf, Sam Lasser, and Kathleen Fisher. Verbatim: A Verified Lexer Generator. Workshop on Language-Theoretic Security (LangSec), 2021. URL https://langsec.org/spw21/papers.html#verbatim.

Derek Egolf, Sam Lasser, and Kathleen Fisher. Verbatim++: Verified, Optimized, and Semantically Rich Lexing with Derivatives. Certified Programs and Proofs (CPP), 2022. URL https://doi.org/10.1145/3497775.3503694.

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

Denis Firsov and Tarmo Uustalu. Certified Normalization of Context-Free Grammars. Certified Programs and Proofs (CPP), 2015. URL https://doi.org/10.1145/2676724.2693177.

Kathleen Fisher and Robert Gruber. PADS: A Domain-Specific Language for Processing Ad Hoc Data. Programming Language Design and Implementation (PLDI), 2005. URL https://doi.org/10.1145/1065010.1065046.

Bryan Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time (Functional Pearl). International Conference on Functional Programming (ICFP), 2002. URL https://doi.org/10.1145/581478.581483.

Bryan Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. Principles of Programming Languages (POPL), 2004. URL https://doi.org/10.1145/964001.964011.

Jay Freeman. Android Bug Superior to Master Key, 2013a. URL http://www.saurik.com/id/18.

Jay Freeman. Exploit (& Fix) Android "Master Key", 2013b. URL http://www.saurik.com/id/17.

Jay Freeman. Yet Another Android Master Key Bug, 2013c. URL http://www.saurik.com/id/19.

Dan Goodin. Failure to patch two-month-old bug led to massive Equifax breach. *Ars Technica*, September 2017. URL https://arstechnica.com/information-technology/2017/09/massive-equifax-breach-caused-by-failure-to-patch-two-month-old-bug.

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

GovTrack contributors. congress-legislators database, 2022. URL https://github.com/unitedstates/congress-legislators.

Dick Grune and Ceriel JH Jacobs. *Parsing Techniques (Monographs in Computer Science)*. Springer-Verlag, 2006.

Michael Hedberg. A Coherence Theorem for Martin-Löf's Type Theory. *Journal of Functional Programming*, 8(4):413–436, 1998. URL http://journals.cambridge.org/action/displayAbstract?aid=44199.

Clinton L. Jeffery. Generating LR Syntax Error Messages from Examples. *ACM Transactions on Programming Languages and Systems*, 25(5):631–640, September 2003. ISSN 0164-0925. URL https://doi.org/10.1145/937563.937566.

Trevor Jim, Yitzhak Mandelbaum, and David Walker. Semantics and Algorithms for Data-Dependent Grammars. Principles of Programming Languages (POPL), 2010. URL https://doi.org/10.1145/1706299.1706347.

S.C. Johnson. *Yacc: Yet Another Compiler-compiler*. Computing science technical report. Bell Laboratories, 1978. URL https://books.google.com/books?id=FrDeHAAACAAJ.

Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating LR(1) Parsers. European Symposium on Programming (ESOP), 2012. URL https://doi.org/10.1007/978-3-642-28869-2_20.

Bart Kiers. ANTLR 4 grammar for Python 3, 2014. URL https://github.com/antlr/grammars-v4.

Adam Koprowski and Henri Binsztok. TRX: A Formally Verified Parser Interpreter. European Symposium on Programming (ESOP), 2010. URL https://doi.org/10.1007/978-3-642-11957-6_19.

Neelakantan R Krishnaswami and Jeremy Yallop. A Typed, Algebraic Approach to Parsing. Programming Language Design and Implementation (PLDI), 2019. URL https://doi.org/10.1145/3314221.3314625.

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

Peter Lammich. Refinement Based Verification of Imperative Data Structures. Certified Programs and Proofs (CPP), 2016. URL https://doi.org/10.1145/2854065.2854067.

Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. A Verified LL(1) Parser Generator. Interactive Theorem Proving (ITP), 2019a. URL https://doi.org/10.4230/LIPIcs.ITP.2019.24.

Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. GitHub repository for the Vermillion development and performance evaluation, 2019b. URL https://github.com/slasser/vermillion.

Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. CoStar artifact submitted to PLDI artifact evaluation, 2021a. URL https://doi.org/10.5281/zenodo.4681598.

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

Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. CoStar: A Verified ALL(*) Parser. Programming Language Design and Implementation (PLDI), 2021c. URL https://doi.org/10.1145/3453483.3454053.

Pierre Letouzey. Extraction in Coq: An Overview. Computability in Europe (CiE), 2008. URL https://doi.org/10.1007/978-3-540-69407-6_39.

P. M. Lewis and R. E. Stearns. Syntax-Directed Transduction. *Journal of the ACM*, 15(3):465–488, July 1968. ISSN 0004-5411. URL https://doi.org/10.1145/321466.321477.

Matthew Might, David Darais, and Daniel Spiewak. Parsing with Derivatives: A Functional Pearl. International Conference on Functional Programming (ICFP), 2011. URL https://doi.org/10.1145/2034773.2034801.

Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional Programming for the Masses*. O'Reilly, 2013.

Falcon Momot, Sergey Bratus, Sven M. Hallberg, and Meredith L. Patterson. The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them. Cybersecurity Development (SecDev), 2016. URL https://doi.org/10.1109/SecDev.2016.019.

Stephen Mort. CVE-2017-5638: The Apache Struts vulnerability explained, September 2017. URL https://www.synopsys.com/blogs/software-security/cve-2017-5638-apache-struts-vulnerability-explained/.

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

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

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

OANC. Open American National Corpus, 2010. URL http://www.anc.org/data/oanc/download/.

Tavis Ormandy. cloudflare: Cloudflare Reverse Proxies are Dumping Uninitialized Memory, 2017. URL https://bugs.chromium.org/p/project-zero/issues/detail?id=1139. Google Project Zero report.

Terence Parr and Kathleen Fisher. LL(*): The Foundation of the ANTLR Parser Generator. Programming Language Design and Implementation (PLDI), 2011. URL https://doi.org/10.1145/1993498.1993548.

Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2014. URL https://doi.org/10.1145/2660193.2660202.

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*. Software Foundations series, Volume 1. 2018. URL https://softwarefoundations.cis.upenn.edu/.

François Pottier. Reachability and Error Diagnosis in LR(1) Parsers. Compiler Construction (CC), 2016. URL https://doi.org/10.1145/2892208.2892224.

François Pottier and Yann Régis-Gianas. Menhir reference manual. 2016. URL http://gallium.inria.fr/~fpottier/menhir/manual.html.

Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified Low-Level Programming Embedded in F*. International Conference on Functional Programming (ICFP), 2017. URL https://doi.org/10.1145/3110261.

Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. USENIX Security Symposium, 2019. URL https://www.usenix.org/conference/usenixsecurity19/presentation/delignat-lavaud.

Tom Ridge. Simple, Functional, Sound and Complete Parsing for All Context-Free Grammars. Certified Programs and Proofs (CPP), 2011. URL https://doi.org/10.1007/978-3-642-25379-9_10.

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

Siguza. Psychic Paper bug report, 2020. URL https://siguza.github.io/psychicpaper/.

Matthieu Sozeau. Program-ing Finger Trees in Coq. International Conference on Functional Programming (ICFP), 2007. URL https://doi.org/10.1145/1291151.1291156.

Christopher Späth, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. SoK: XML Parser Vulnerabilities. USENIX Workshop on Offensive Technologies (WOOT), 2016. URL https://www.usenix.org/conference/woot16/workshop-program/presentation/spath.

Ryan Wisnesky, Gregory Malecha, and Greg Morrisett. Certified Web Services in Ynot. Workshop on Automated Specification and Verification of Web Systems (WWV), 2009. URL https://www3.risc.jku.at/conferences/wwv09.

W. A. Woods. Transition Network Grammars for Natural Language Analysis. *Communications of the ACM*, 13(10):591–606, October 1970. ISSN 0001-0782. URL https://doi.org/10.1145/355598.362773.

Qianchuan Ye and Benjamin Delaware. A Verified Protocol Buffer Compiler. Certified Programs and Proofs (CPP), 2019. URL https://doi.org/10.1145/3293880.3294105.