A Verified LL(1) Parser Generator

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

Chris Casinghino
Draper Laboratory, USA
ccasinghino@draper.com

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

Cody Roux
Draper Laboratory, USA
croux@draper.com

Abstract

An LL(1) parser is a recursive descent algorithm that uses a single token of lookahead to build a grammatical derivation for an input sequence. We present an LL(1) parser generator that, when applied to grammar $\mathcal{G}$, produces an LL(1) parser for $\mathcal{G}$ if such a parser exists. We use the Coq Proof Assistant to verify that the generator and the parsers that it produces are sound and complete, and that they terminate on all inputs without using fuel parameters. As a case study, we extract the tool's source code and use it to generate a JSON parser. The generated parser runs in linear time; its performance is comparable to that of other verified parsers, and it is three to five times slower than an unverified parser for the same grammar.

2012 ACM Subject Classification Theory of computation → Grammars and context-free languages; Software and its engineering → Parsers; Software and its engineering → Formal software verification

Keywords and phrases interactive theorem proving, top-down parsing

1 Introduction

Many software systems employ parsing techniques to map sequential input to structured output. Often, a parser is the system component that consumes data from an untrusted source—for example, many applications parse input in a standard format such as XML or JSON as the first step in a data-processing pipeline. Because parsers mediate between the outside world and application internals, they are good targets for formal verification; parsers that come with strong correctness guarantees are likely to increase the overall security of applications that rely on them.

Several recent high-profile software vulnerabilities demonstrate the consequences of using unsafe parsing tools. Attackers exploited a faulty parser in a web application framework, obtaining the sensitive data of as many as 143 million consumers [7, 15]. An HTML parser vulnerability led to private user data being leaked from several popular online services [8]. And a flaw in an XML parser enabled remote code execution on a network security device—a flaw that received a Common Vulnerability Score System (CVSS) score of 10/10 due to its severity [14]. These and other examples highlight the need for secure parsing technologies.

Parsing is a widely studied topic, and it encompasses a range of techniques with different advantages and drawbacks [9]. One family of parsing algorithms, the top-down or LL-style algorithms, shares several strengths relative to other strategies. LL parsers typically produce clear error messages, and they can be easily extended with semantic actions that produce user-defined data structures; in addition, generated LL parser code is often human-readable and similar to hand-written code [16].

The common ancestor of the LL family is LL(1), a recursive descent algorithm that avoids backtracking by looking 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 runs in $O(n^4)$ time [17]. Different algorithms are therefore suited to different applications; it is often advantageous to choose the most efficient algorithm compatible with the language being parsed.

In this paper, we present Vermillion, a formally verified LL(1) parser generator. This tool is part of a planned suite of verified LL-style parsing technologies that are suitable for a wide range of data formats. We implemented and verified the parser generator using the Coq Proof Assistant [20], a popular interactive theorem prover. The tool 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 LL(1) parsing algorithm implementation that is parameterized by a parse table. By converting a grammar to a table and then partially applying the parser to the table, the user obtains a parser that is specialized to the original grammar. The paper's main contributions are as follows:

1. **End-to-End Correctness Proofs** – We prove that both the parse table generator and the parser are sound and complete. The generator produces a correct LL(1) parse table for any grammar if such a table exists. The parser produces a derivation tree for its input that is correct with respect to the grammar used to generate the parser. Although prior work has verified some of the steps involved in LL(1) parse table generation, 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 parser terminate on both valid and invalid inputs without the use of fuel-like parameters. To the best of our knowledge, we are the first to prove this property about a parser generator based on the context-free grammar formalism. Some existing verified parsers are only guaranteed to terminate on valid inputs; others 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 [12] to convert the parser generator to OCaml source code and generated an LL(1) parser from a JSON grammar. We then used Menhir [18], a popular OCaml parser generator, to produce an unverified second parser for the same grammar and compared the two parsers' performance on a JSON data set. We found that the verified parser's speed was three to five times slower than the unverified and optimized tool, which is similar to the reported results for other certified parsers [10, 11]. 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 context-free grammars. To implement and verify these analyses, we make ample use of Coq's tools for defining 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 well-founded recursion on a non-syntactic measure, and building a provably terminating implementation that also runs efficiently requires judicious use of dependent types. Our parser safety proof relies on a lemma stating that if a correct LL(1) parser table exists for some grammar, then the grammar contains no left recursion. Our proof of this lemma is quite intricate, and we were unable to find a rigorous proof of this seemingly intuitive fact in the literature.

Our formalization consists of roughly 8,000 lines of Coq definitions and proofs. The development is available online [1].

This paper is organized as follows: in §2, we review background material on context-free grammars and LL(1) parsing. In §3, we describe the high-level structure of our parse table generator and its correctness proofs. In §4, we present the LL(1) parsing algorithm and its correctness properties. In §5, we present the results of evaluating our tool's performance on a JSON benchmark. We discuss related work in §6 and our plans for future work in §7.

2 Grammars and Parse Tables

2.1 Grammars

Our grammars are composed of terminals drawn from a set $\mathcal{T}$and nonterminal symbols drawn from a set $\mathcal{N}$. Throughout this work, we use the letters $\{a, b, c\}$ as terminal names, $\{S, X, Y, Z\}$ as nonterminal names, $\{s, s', ...\}$ as names for arbitrary symbols (terminals or nonterminals), and $\{\alpha, \beta, \gamma\}$ as names for finite sequences of symbols (also called sentential forms). We represent a grammar $\mathcal{G}$ as a structure with the following two components: a start symbol $S \in \mathcal{N}$, and a finite sequence of productions $\mathcal{P} \subseteq \mathcal{N} \times (\mathcal{N} \cup \mathcal{T})^*$.

2.2 LL(1) Derivations

We define a derivation relation over a grammar symbol s, a word w that it derives, and the derivation tree tr for w rooted at s. Because it is useful for a parser to produce a derivation tree for a prefix of its input sequence and return the remainder of the sequence along with the tree, the derivation relation also includes the *remainder*, or the unparsed suffix of the input. The relation has the judgment form $s \xrightarrow{tr} w \mid r$, which is read, "$s$ derives w, producing tr and leaving r unparsed."

The derivation relation appears in Figure 1. It is mutually inductive with an analogous relation (also in Figure 1) over a list of symbols γ, a word w, a forest (list of trees) f, and a remainder r. This second relation has the judgment form $\gamma \overset{f}{\Rightarrow} w \mid r$ ("γ derives w, producing f and leaving r unparsed").

$$\textsc{DerT}\quad \frac{}{\forall r, a \xrightarrow{\text{Leaf a}} a \mid r} \qquad\qquad \textsc{DerNT}\quad \frac{\begin{array}{c} peek(w +\!\!+\, r) \in LOOKAHEAD(X \to \gamma) \\ \gamma \overset{\text{f}}{\Rightarrow} w \mid r \end{array}}{X \xrightarrow{\text{Node X f}} w \mid r}$$

$$\textsc{DerNil}\quad \frac{}{\forall r, [] \overset{[]}{\Rightarrow} \epsilon \mid r} \qquad\qquad \textsc{DerCons}\quad \frac{s \xrightarrow{\text{tr}} w \mid w' +\!\!+\, r \qquad \gamma \overset{\text{f}}{\Rightarrow} w' \mid r}{s :: \gamma \xRightarrow{\text{tr :: f}} w +\!\!+\, w' \mid r}$$

Figure 1 Tree and Forest Derivation Relations

The DerNT rule is the only LL(1)-specific rule in the relation. The *peek* function returns a value $l \in \mathcal{T} \cup \{EOF\}$ 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 production $X \to \gamma$ applies when $peek(w +\!\!+\, r)$ and $X \to \gamma$ are in the LOOKAHEAD relation (Figure 5)—i.e., when the first input token "predicts" that production. To make this lookahead concept precise, we introduce the definitions of several predicates that are commonly used in parsing theory to relate a grammar's structure to its semantics.

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) 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 NULLABLE(γ) to represent the fact that γ is a nullable symbol sequence.

$$\textsc{NuSym}\quad \frac{(X \to \gamma) \in \mathcal{P} \qquad NULLABLE(\gamma)}{NULLABLE(X)} \qquad\qquad \textsc{NuGamma}\quad \frac{\forall i \in \{1 ... n\} \qquad NULLABLE(s_i)}{NULLABLE(s_1 ... s_n)}$$

Figure 2 NULLABLE Relation

The FIRST relation (Figure 3) 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 $a \in FIRST(s)$. Once again, we extend this concept to sentential forms, writing $a \in FIRST(\gamma)$ if γ derives a word that begins with a.

$$\textsc{FirstT}\quad \frac{}{a \in FIRST(a)} \qquad\qquad \textsc{FirstNT}\quad \frac{(X \to \gamma) \in \mathcal{P} \qquad a \in FIRST(\gamma)}{a \in FIRST(X)}$$

$$\textsc{FirstGamma}\quad \frac{\gamma = \alpha s \beta \qquad NULLABLE(\alpha) \qquad a \in FIRST(s)}{a \in FIRST(\gamma)}$$

Figure 3 FIRST Relation

The FOLLOW relation (Figure 4) 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 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 $l \in LOOKAHEAD(X \to \gamma)$ ("l is a lookahead token for production $X \to \gamma$") in Figure 5. Intuitively, l is a token that, when it begins a sequence ts, "predicts" that $X \to \gamma$ can derive a

$$\textsc{FollowStart}\ \frac{S \text{ is the start symbol}}{EOF \in FOLLOW(S)} \qquad \textsc{FollowRight}\ \frac{(X \to \alpha Y \beta) \in \mathcal{P} \qquad a \in FIRST(\beta)}{a \in FOLLOW(Y)}$$

$$\textsc{FollowLeft}\ \frac{(X \to \alpha Y \beta) \in \mathcal{P} \qquad NULLABLE(\beta) \qquad l \in FOLLOW(X)}{l \in FOLLOW(Y)}$$

Figure 4 FOLLOW Relation

prefix of *ts*. As a special case, if $X \to \gamma$ derives $ts = \epsilon$, then $EOF \in LOOKAHEAD(X \to \gamma)$. When an LL(1) parser builds a derivation from nonterminal X for a prefix of *ts*, it "looks ahead" at *ts* and applies a production $X \to \gamma$ such that $peek(ts) \in LOOKAHEAD(X \to \gamma)$.

$$\textsc{FirstLookahead}\ \frac{l \in FIRST(\gamma)}{l \in LOOKAHEAD(X \to \gamma)} \qquad \textsc{FollowLookahead}\ \frac{NULLABLE(\gamma) \qquad l \in FOLLOW(X)}{l \in LOOKAHEAD(X \to \gamma)}$$

Figure 5 LOOKAHEAD Relation

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 in table *tbl* at row X and column l that contains γ (written $tbl(X, l) = \gamma$) represents the fact that $l \in LOOKAHEAD(X \to \gamma)$.

Figure 6 contains a grammar and its LL(1) parse table. Cell (**X**, **b**), for instance, contains Zc (the right-hand side of production 2) because $b \in FIRST(Zc)$. Cell (**Z**, **c**) contains Y (the right-hand side of production 5) because $NULLABLE(Y)$ and $c \in FOLLOW(Z)$.

(X is the start symbol)

1. $X \to aY$ 3. $Y \to \epsilon$ 4. $Z \to b$
2. $X \to Zc$ 5. $Z \to Y$

	a	**b**	**c**	**EOF**
X	aY	Zc	Zc	
Y			ϵ	ϵ
Z		b	Y	

Figure 6 Example Grammar and LL(1) Parse Table

A correct LL(1) parse table *tbl* for grammar $\mathcal{G}$ contains all and only the lookahead facts about $\mathcal{G}$—i.e., $tbl(X, l) = \gamma \iff l \in LOOKAHEAD(X \to \gamma)$. Not every grammar has a correct LL(1) parse table. If $l \in LOOKAHEAD(X \to \gamma)$ and $l \in LOOKAHEAD(X \to \gamma')$, where $\gamma \neq \gamma'$, then no correct table exists for $\mathcal{G}$—a parser would be unable to predict whether to choose right-hand side γ or γ' upon encountering nonterminal X and token l. A grammar that has a correct LL(1) parse table is called an LL(1) grammar.

3 Parse Table Generator Correctness Properties and Verification

We now describe the process of developing and verifying an LL(1) parse table generator. Our goal is to define the Coq function `parseTableOf : grammar -> option parse_table` and prove that it is both *sound* (every table that it produces is a correct LL(1) parse table for the input grammar) and *complete* (it produces a correct LL(1) parse table for the grammar if such a table exists).

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* [2], 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 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, many species of parser generators 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 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 value if the set contains a conflict

Several of these steps involve similar reasoning and employ the same proof techniques. In the next section, we examine a single dataflow analysis implementation and its correctness proof in detail to illustrate these techniques.

3.2 `mkNullableSet` Correctness

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 `g`, the resulting set contains all and only the nullable nonterminals from `g`. We formalize this correctness property and theorem statement in Coq as follows (`nullable_sym` is the mechanized version of the NULLABLE relation in Figure 2):

```
Definition nullable_set_correct (nu : NtSet.t) (g : grammar) :=
  forall (x : nonterminal), NtSet.In x nu <-> nullable_sym g (NT x).

Theorem mkNullableSet_correct :
  forall (g : grammar), nullable_set_correct (mkNullableSet g) g.
```

Portions of the `mkNullableSet` implementation appear in Figure 7. We represent a grammar as a record with fields `start : nonterminal` and `prods : list production`; the expression `g.(prods)` projects the `prods` field from a grammar. 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.

```
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 7 `mkNullableSet` Implementation

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 [19], which provides support for defining functions using well-founded recursion. The Program Fixpoint construct 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 7) is the cardinality of `nu`'s complement with respect to the universe $\mathcal{U}$ of production left-hand sides. 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 left-hand side 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 statements and proofs of its main correctness properties.

3.3 `mkNullableSet` Soundness

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

```
Definition nullable_set_sound (nu : nullable_set) (g  : grammar) :=
  forall (x : nonterminal), NtSet.In x nu -> nullable_sym g (NT x).

Theorem mkNullableSet_sound :
  forall (g : grammar), nullable_set_sound (mkNullableSet g) 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 are computed 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 `nu` is sound, then `mkNullableSet'` applied to `nu` is also sound

(1) is simple to prove, because the initial `nu` 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 (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 7) preserves soundness of the NULLABLE set.

3.4 `mkNullableSet` Completeness

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

```
Definition nullable_set_complete (nu : NtSet.t) (g  : grammar) : Prop :=
  forall (x : nonterminal), nullable_sym g (NT x) -> NtSet.In x nu.

Theorem mkNullableSet_complete :
  forall (g : grammar), nullable_set_complete (mkNullableSet g) g.
```

Once again, the proof is based on well-founded induction on the `mkNullableSet'` measure. In the interesting case, we must prove `nu` complete given the fact that `nu` and `(nullablePass g.(prods) nu)` are equal. In other words, we need to show that any fixed point of the dataflow analysis is complete. We isolate this fact in the lemma `nullablePass_equal_complete`:

```
Lemma nullablePass_equal_complete :
  forall (g : grammar) (nu : NtSet.t),
    NtSet.Equal nu (nullablePass g.(prods) nu)
    -> nullable_set_complete nu g.
```

After some simplification, we are left with this goal:

$$\frac{\texttt{nullable\_sym g x} \qquad \texttt{nu = nullablePass g.(prods) nu}}{\texttt{NtSet.In x nu}}$$

The proof proceeds by induction on the `nullable_sym` judgment. Because this relation is mutually inductive with `nullable_gamma`, we use Coq's `Scheme` command 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 many 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.

3.5 `parseTableOf` Soundness and Completeness

Computing the NULLABLE set is the first of several dataflow analyses involved in generating an LL(1) parse table. The remaining steps follow the same pattern. For example, both the FIRST and FOLLOW analyses have termination measures that are defined in terms of a set's complement with respect to some finite universe. Each step also has a soundness proof based on the fact that soundness is an invariant for 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—by simply chaining together the correctness proofs for the individual steps. The `parseTableOf` soundness and completeness theorem statements appear below:

```
(* tbl contains all and only the LOOKAHEAD facts about g *)
Definition parse_table_correct (tbl : parse_table) (g : grammar) :=
  forall (x : nonterminal) (la : lookahead) (gamma : list symbol),
    pt_lookup x la tbl = Some gamma
    <-> In (x, gamma) g.(prods) /\ lookahead_for la x gamma g.

Theorem parseTableOf_sound :
  forall (g : grammar) (tbl : parse_table),
    parseTableOf g = Some tbl -> parse_table_correct tbl g.

Theorem parseTableOf_complete :
  forall (g : grammar) (tbl : parse_table),
    parse_table_correct tbl g
    -> exists (tbl' : parse_table),
         ParseTable.Equal tbl tbl' /\ parseTableOf g = Some tbl'.
```

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

```
Theorem unprovable_parseTableOf_complete :
  forall (g : grammar) (tbl : parse_table),
    parse_table_correct tbl g -> parseTableOf g = Some tbl.
```

We represent a parse table as a finite map with keys of type `nonterminal * lookahead` and values of type `list symbol`; a key `(x, l)` that maps to `gamma` represents a table cell at row `x` and column `l` that contains `gamma`. FMaps, the Coq finite map library that we use to implement parse tables, only defines map equality *extensionally*. If `tbl` is a correct parse table for `g`, we cannot prove that `parseTableOf` returns `tbl` itself—only that it returns a table containing exactly the same entries as `tbl`, which should be sufficient for any application.

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.

4 Parser Correctness and Verification

We now turn to the task of defining and verifying the LL(1) parsing algorithm. Our first goal is to define a function `parse` that uses an LL(1) parse table `tbl` to build a derivation tree rooted at symbol `sym` for a prefix of the token sequence `input`:

```
Definition parse (tbl : parse_table) (sym : symbol) (input : list terminal):
  sum parse_failure (tree * list terminal).
```

(A value of type `sum A B` is either `inl A` or `inr B`.) 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 an input sequence, returning a tree tr for prefix w and an unparsed suffix r, then $sym \xrightarrow{tr} w \mid r$ holds
2. (*Safety*) - The parser never returns an error value
3. (*Completeness*) - If $sym \xrightarrow{tr} w \mid r$ holds, then the parser returns tr and r when applied to input sequence $w +\!\!+\, r$

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, `parseTree` and `parseForest`, that respectively consume a symbol and a list of symbols and return a tree and a list of trees. 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—our 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 finite 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 an input 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. In our safety proof, we show that the parser never actually returns this "left recursion detected" value as long as it is applied to a well-formed table—i.e., 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 reason why the parser might fail—it 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 (t, r)`, where `t` is the tree for a prefix of the input 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 (LeftRec X v r)`, where `X` is the nonterminal found to be left-recursive, `v` is the visited set at the point when left recursion was detected, and `r` is the unparsed suffix

After adding left recursion detection to the parser, we still have to convince Coq that the function terminates, because its termination metric depends on multiple function parameters. The input sequence structurally decreases 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). To complicate matters, Coq's Function and Program facilities—standard tools for defining functions with subtle termination conditions—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 facilities perform automatically. The process involves the following steps:

- Define a measure `meas` that maps several function arguments to a triple with the following elements:
  - *(first projection)* the length of the input sequence
  - *(second projection)* the cardinality of the visited set's complement relative to the set of all grammar nonterminals
  - *(third projection)* the function's symbol argument (in the case of `parseTree`) or "list of symbols" argument (in the case of `parseForest`)
- Define a lexicographic ordering `triple_lt` on triples of this type
- 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 parameter
- Prove lemmas showing that the size of this accessibility proof decreases on recursive calls
- Prove that `triple_lt` is well-founded so that the parser can be called with any initial set of arguments

This process yields functions with the following signatures:

```
Fixpoint parseTree (tbl : parse_table) (sym : symbol)
                   (input : list terminal) (vis : NtSet.t)
                   (a : Acc triple_lt (meas tbl input vis (F_arg sym)))
  : sum parse_failure (tree * {input' & length_lt_eq _ input' input}) ...
with parseForest (tbl : parse_table) (gamma : list symbol)
                 (input : list terminal) (vis : NtSet.t)
                 (a : Acc triple_lt (meas tbl input vis (G_arg gamma)))
  : sum parse_failure (list tree * {input' & length_lt_eq _ input' input}) ...
```

In each return type, `{input' & length_lt_eq _ input' input}` is the dependent type of a list `input'` that is either shorter than the `input` parameter or definitionally equal to `input`. By including this information in the functions' dependent return types, we avoid computing the length of the remaining input, which would hamper performance.

Finally, we define `parse`, a top-level interface to the parser that simply invokes `parseTree` with an empty visited set and an appropriate accessibility proof term, and strips out the return value's dependent component:

```
Definition parse (tbl : parse_table) (sym : symbol) (input : list terminal) :
  sum parse_failure (tree * list terminal) :=
  match parseTree tbl sym input NtSet.empty (triple_lt_wf _) with
  | inl failure => inl failure
  | inr (tr, existT _ input' _) => inr (tr, input')
  end.
```

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 tree for some prefix of its input, the relation `sym_derives_prefix` (the mechanized version of the Figure 1 tree derivation relation) produces the same tree for the same prefix:

```
Theorem parse_sound :
  forall g tbl sym word rem tr,
    parse_table_correct tbl g
    -> parse tbl sym (word ++ rem) = inr (tr, rem)
    -> sym_derives_prefix g sym word tr rem.
```

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

```
Lemma parseTree_sound' :
  forall g tbl sym input rem tr vis a,
    parse_table_correct tbl g
    -> parseTree tbl sym input vis a = inr (tr, rem)
    -> exists word,
         word ++ rem = input /\ sym_derives_prefix g sym word tr rem.
```

The main difference between these two properties is that `parse_sound` 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 there are multiple ways of dividing the input into a prefix and suffix.

The proof of `parseTree_sound'` is straightforward by design; we were careful to define `parseTree` and `parseForest` so that the "success" path through the functions' recursive calls mirrors the structure of the derivation relation. The proof is by structural induction on the derivation tree, and the main challenge is coming up with a mutual induction principle that refers to both trees and lists of trees. This situation is common when trying to prove mutually inductive properties of types that are not defined mutually. One solution is to create a custom "list of trees" type that is defined mutually with `tree`. The downside of this approach is that one can no longer use the many Coq Standard Library list lemmas when proving statements about this type. Instead, we follow an example from the *Coq'Art* textbook [4] and build a custom mutual induction principle for trees and lists of trees.

4.3 Parser Safety

Our next task is to prove that the parser is *safe*—that it never returns a `LeftRec` value as long as its table argument is a correct LL(1) parse table for some grammar:

```
Theorem parse_safe :
  forall g tbl sym input,
    parse_table_correct tbl g
    -> forall x vis input',
       ~ parse tbl sym input = inl (LeftRec x vis input').
```

However, it is certainly possible for `parseTree` and `parseForest` to return `LeftRec`! For example, they will produce an error when applied to nonterminal X and a 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 symbol s and its visited set already contains a nonterminal that is reachable from s without any input being consumed. We formalize this notion of "null-reachability" in the inductive predicate `nullable_path`:

```
Inductive nullable_path g (la : lookahead) :
  symbol -> symbol -> Prop :=
| DirectPath : forall x z pre suf,
    In (x, pre ++ NT z :: suf) g.(prods)
    -> nullable_gamma g pre
    -> lookahead_for la x (pre ++ NT z :: suf) g
    -> nullable_path g la (NT x) (NT z)
| IndirectPath : forall x y z pre suf,
    In (x, pre ++ NT y :: suf) g.(prods)
    -> nullable_gamma g pre
    -> lookahead_for la x (pre ++ NT y :: suf) g
    -> nullable_path g la (NT y) (NT z)
    -> nullable_path g la (NT x) (NT z).
```

When this predicate holds of two symbols s and s', there exists a sequence of steps through the grammar from s to s' in which all symbols visited along the way are nullable.

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

```
(* symbol s is left-recursive in grammar g on lookahead token la *)
Definition left_recursive g s la :=
  nullable_path g la s s.
```

We prove a lemma stating that when `parseTree` or `parseForest` returns a `LeftRec` value, one or both of these error conditions holds. The first condition does not apply to `parse` because the top-level function calls `parseTree` 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 many 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_sym` (Figure 2) and `first_sym` (Figure 3) 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_sym` or `first_sym` fact have the same size.
(3) We show that if grammar `g` contains a left-recursive nonterminal, then there are two proofs of the same `nullable_sym` or `first_sym` fact about `g` with *different* sizes.

These steps enable us to prove the lemma `LL1_parse_table_impl_no_left_recursion` by obtaining a contradiction from (2) and (3):

```
Lemma LL1_parse_table_impl_no_left_recursion :
  forall g tbl la x,
    parse_table_correct tbl g
    -> ~ left_recursive g (NT x) la.
```

4.4 Parser Completeness

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

```
Theorem parse_complete :
  forall g tbl sym word tr rem,
    parse_table_correct tbl g
    -> sym_derives_prefix g sym word tr rem
    -> parse tbl sym (word ++ rem) = inr (tr, rem).
```

Our safety theorem simplifies the task of proving completeness. We begin by proving a more general lemma stating that when a grammar derivation exists, the parser either produces the same tree or returns an error value:

```
Lemma parse_leftrec_or_complete :
  forall g tbl sym word tr rem,
    parse_table_correct tbl g
    -> sym_derives_prefix g sym word tr rem
    -> (exists x vis' input',
            parse tbl sym (word ++ rem) = inl (LeftRec x vis' input'))
       \/ parse tbl sym (word ++ rem) = inr (tr, rem).
```

We prove this lemma by induction on the derivation relation, use the safety result to rule out the left disjunct, and use the right disjunct to prove the completeness theorem itself.

5 Evaluation

To evaluate the efficiency of our generated parsers, we extracted Vermillion to OCaml source code and generated an LL(1) parser for the JSON data format. We also used Menhir, an OCaml parser generator, to produce a second, unverified parser for the same grammar and compared the two parsers' performance on a JSON data set.

We based our Menhir lexer[1] and grammar on the ones described in the *Real World OCaml* textbook's tutorial on JSON parsing [13]. We then replicated the grammar in Vermillion's input format. Because our tool consumes a list of tokens, rather than wrapping a lexer that produces tokens "on demand" as Menhir and Yacc parsers do, 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.

[1] The lexer does not support Unicode escape sequences, but nothing prevents Vermillion or Menhir from handling Unicode tokens in principle.

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 8. The Vermillion parser is three to five times slower than the unverified Menhir parser. This comparison is not entirely scientific, because the Menhir parser builds a data structure by performing semantic actions while the Vermillion parser produces a generic tree structure. Nevertheless, it suggests that Vermillion's performance is reasonable, given that it was designed with ease of verification (rather than optimal performance) in mind, and that it is table-based (as opposed to using generated source code). Other certified parsers obtain similar performance results; a validated LR(1) parser [10] runs about five times slower than its unvalidated counterpart, and a verified PEG interpreter [11] is two to three times slower than an unverified version.

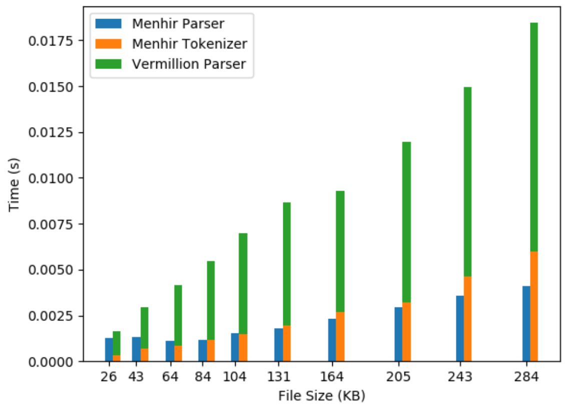

Figure 8 JSON Parser Average Execution Times

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.

6 Related Work

Barthwal and Norrish [3] use the HOL4 proof assistant to prove the soundness and completeness of generated SLR parsers. Like us, they structure their tool as a generator and a parse function parameterized by the generator's output. The parsers are not proved to terminate on invalid inputs. The work does not include performance results, but the parsers are not designed to be performant; they compute DFA states during execution rather than statically.

Jourdan et al. [10] present a validator that determines whether a generated LR(1) parser is sound and complete. *A posteriori* validation is a flexible and lightweight alternative to full verification; the validator is compatible with untrusted generators, and its formalization is small. One cost of this method is the time required to validate a generated parser. A grammar that is still in development is likely to undergo many small changes—generating and validating a parser after each change might slow the development cycle. The validator does not guarantee that a parser terminates on invalid inputs. While LR(1) parsers are compatible with a larger class of grammars than LL(1) parsers, they often produce less intuitive error messages.

Parsing Expression Grammars (PEGs) [6] are a language representation that is sometimes used in place of context-free grammars to specify parsers. Koprowski and Binsztok [11] verify the soundness and completeness of a PEG parser interpreter. They also ensure that the interpreter terminates on both valid and invalid inputs by rejecting grammars that fail a syntactic check for left recursion. Wisnesky et al. [21] verify an optimized PEG parser using the Ynot framework. Ynot is a library for proving the partial correctness of imperative programs, so the parser is not guaranteed to terminate. One drawback of using PEG parsers is that they make greedy choices at decision points—e.g., the rule $S \to a|ab$ applied to string ab parses a instead of ab—which can produce difficult-to-debug behavior.

7 Conclusions

We have verified that our parser generator produces a sound and complete LL(1) parser for its input grammar whenever such a parser exists, and that the generated parsers terminate on valid and invalid inputs without using fuel. Below, we discuss two possible extensions of this work: adding semantic actions, and generating source code for optimization purposes.

We plan to extend Vermillion with support for semantic actions. We could represent these actions as Gallina functions in the style of Menhir's Coq backend [10], or we could implement a custom language of actions so that users do not need to be familiar with Gallina.

Our parsers represent tables as finite maps and perform map lookups at decision points, which is a likely source of inefficiency. Many 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. We could develop a version of our tool that generates abstract syntax for a language with mechanized semantics, such as Clight [5], and verify that the abstract syntax representation of a parser is extensionally equivalent to a table-based parser for the same grammar.

References

1 Source files for verified LL(1) parser generator. `https://github.com/slasser/vermillion`.

2 Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

3 Aditi Barthwal and Michael Norrish. Verified, executable parsing. In *European Symposium on Programming*, pages 160–174. Springer, 2009.

4 Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

5 Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

6 Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM. URL: `http://doi.acm.org/10.1145/964001.964011`, `doi:10.1145/964001.964011`.

7 Dan Goodin. Failure to patch two-month-old bug led to massive equifax breach. `https://arstechnica.com/information-technology/2017/09/massive-equifax-breach-caused-by-failure-to-patch-two-month-old-bug/`, 2017.

8 cloudflare: Cloudflare reverse proxies are dumping uninitialized memory. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1139`, 2017.

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

10 Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating LR(1) parsers. In *European Symposium on Programming*, pages 397–416. Springer, 2012.

11 Adam Koprowski and Henri Binsztok. TRX: A formally verified parser interpreter. In *European Symposium on Programming*, pages 345–365. Springer, 2010.

12 Pierre Letouzey. Extraction in Coq: An overview. In *Conference on Computability in Europe*, pages 359–369. Springer, 2008.

13 Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional programming for the masses.* O'Reilly Media, Inc., 2013.

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

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

16 Terence Parr and Kathleen Fisher. LL(*): The foundation of the ANTLR parser generator. pages 425–436, 06 2011.

17 Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive LL(*) parsing: The power of dynamic analysis. volume 49, pages 579–598, 10 2014. `doi:10.1145/2714064.2660202`.

18 François Pottier and Yann Régis-Gianas. Menhir reference manual. *Inria, Aug*, 2016.

19 Matthieu Sozeau. Program-ing finger trees in Coq. In *In ACM SIGPLAN International Conference on Functional Programming. Association for Computing Machinery*, 2007.

20 The Coq proof assistant, version 8.9.0, January 2019. URL: `https://doi.org/10.5281/zenodo.2554024`, `doi:10.5281/zenodo.2554024`.

21 Ryan Wisnesky, Gregory Michael Malecha, and John Gregory Morrisett. Certified web services in Ynot. 2010.