Practical LR Parser Generation

Joe Zimmerman
jzim@cs.stanford.edu

Abstract

The parsing problem, taking a string of characters and producing an abstract syntax tree, is a fundamental building block in modern compilers. For industrial programming languages, parsing is a surprisingly involved task. Approaches to generate parsers automatically have been known for many years, beginning with Knuth's seminal work on canonical LR parsing in 1965, and refined by DeRemer in 1969 to the restricted LALR class of grammars (which serve as the basis for automatic parser generation tools such as `yacc` and `bison`). However, the prevailing consensus is that automatic parser generation is not practical for real programming languages: LALR parsers—and even the more general LR parsers, in many cases—are considered to be far too restrictive in the grammars they support, and moreover, LR parsers are often considered too inefficient in practice. As a result, virtually all modern languages use recursive-descent parsers written by hand, a lengthy and error-prone process that dramatically increases the barrier to new programming language development.

In this work we demonstrate that, contrary to the prevailing consensus, we can have the best of both worlds: for a very general, practical class of grammars—a strict superset of Knuth's canonical LR—we can generate parsers automatically, and such that the resulting parser code, as well as the generation procedure itself, is highly efficient. This advance relies on several new ideas, primarily: (i) an improved optimization procedure for lookaheads in LR automata; (ii) a grammar transformation, reminiscent of continuation-passing style (CPS), which defers shift/reduce decisions to the end of processing for compound rules; (iii) a refinement of the LR paradigm to include recursive-descent actions; (iv) an extension of context-free grammars to include per-symbol attributes (e.g., "expression in a type context vs. a value context"); and (v) an extension of canonical LR parsing, which we refer to as XLR, which endows shift/reduce parsers with the power of bounded nondeterministic choice.

With these ingredients, we can automatically generate efficient parsers for virtually all programming languages that are intuitively easy to parse—a claim we support experimentally, by implementing the new algorithms in a new software tool called `langcc` [Zim22] and running them on syntax specifications for Golang 1.17.8 and Python 3.9.12. The tool handles both languages automatically, and the generated code, when run on standard codebases, is 1.2x faster than the corresponding hand-written parser for Golang, and 4.3x faster than the CPython parser, respectively.

1 Introduction

1.1 General parsing

We begin with a brief exposition of the parsing problem in its full generality. Parsing takes as input a context-free grammar $\mathcal{G}$ and some input string of length n, and produces a parse tree for the string according to $\mathcal{G}$, if one exists. General parsing can be done in time $O(n^3)$, neglecting factors of $|\mathcal{G}|$, by a standard bottom-up dynamic programming approach. Valiant's algorithm [Val75] improves this result to $O(n^\omega)$, where ω denotes the matrix multiplication exponent. The algorithms of Earley [Ear70], Tomita [Tom84] and Leo [Leo91] ("ETL") also build a dynamic programming table, but do so left-to-right rather than bottom-up, and hence only generate parses for substrings that are consistent with some left-to-right partial parse of the entire input up to that point. As a result, while the ETL algorithms still require $\Omega(n^3)$ time for general grammars, they achieve much better performance on various specific subclasses of grammars.

1.2 Unambiguous grammars

Virtually all programming language grammars of practical interest are *unambiguous*, meaning that for every reachable symbol X of the grammar and every string x, there is at most one parse tree $X \stackrel{*}{\Rightarrow} x$. Of the ETL algorithms, Earley's and Leo's both run in time $O(n^2)$ on unambiguous grammars, despite requiring $O(n^3)$ time in the general case. Naively, we might hope that the class of grammars for which one could automatically generate efficient parsers would turn out to be precisely the class of unambiguous grammars.

Unfortunately, it is currently unknown whether unambiguous grammars can be parsed in sub-quadratic time. Abboud et al. [ABW15] give a reduction showing that general grammars cannot be parsed in time $o(n^\omega)$, unless there is a breakthrough in searching for cliques in graphs. However, clearly this reduction does not extend to the case of unambiguous grammars, since it is already known how to parse these in time $O(n^2)$ via ETL. In fact, as far as we know, it could be possible to parse every unambiguous grammar in $O(n)$ time. Intuitively, we conjecture a pessimistic outcome; however, we observe that we currently do not even have an example of a specific unambiguous grammar for which no linear-time parsing algorithm is known. This gap is one of the most significant open problems in the study of parsing, as it is of not only theoretical but potentially great practical importance. For the present, however, we require practical parsers to run in $O(n)$ time, and hence we must accept some limitations beyond just unambiguity.

1.3 Practical parsers

For industrial programming languages, the gold standard is the hand-written recursive-descent parser, which typically runs in time $O(n)$ with a very small constant factor, independent of $|\mathcal{G}|$.[1] This means that, in order for automatic parser generation to be competitive with this approach, it is not sufficient merely to require the input grammar to be unambiguous. In fact, even Leo's algorithm, which runs in time $O(n)$ on all grammars in the class LR(k), poses efficiency problems for competitive automatic parser generation, due to the associated constant factors. Thus, for algorithms which are competitive with hand-written parsers in practice, we look to the highly efficient online (left-to-right) family of parsing algorithms, which typically require grammars of a more restricted form.

[1] The lack of dependence on $|\mathcal{G}|$, which might seem impossible a priori, is achieved by encoding the grammar into the structure of the code itself (for hand-written parsers), or into a series of branch tables (for automatically generated parsers).

Of these online left-to-right algorithms, the primary categories are top-down ("LL(k)") and bottom-up ("LR(k)"). In top-down parsing, we require the online parser, with k symbols of lookahead, to emit the name of each production when its occurrence *begins* in the input; in bottom-up parsing, it is emitted when its occurence *ends*. It is fairly clear that top-down parsing is not sufficiently general for practical programming languages: even in simple cases such as binary operator-precedence grammars, one does not know whether one is parsing the production "$E \to E + E$" or the production "$E \to E \times E$" until after seeing the entirety of the first occurrence of E. While there are transformations of the input grammar that can make it more amenable to top-down parsing, many reasonable languages admit no LL(k) grammar for any k (e.g., the well-known "dangling-else" grammar). Moreover, even in cases where such transformations succeed, they often introduce significant overhead and drastically change the semantics of the grammar. Thus, in order to achieve automatic parser generation for a practical family of languages, we will confine our attention to bottom-up (LR-style) parsing.

1.4 Practical classes of grammars

Within the category of online bottom-up parsing, there remains the question of what restrictions we may place on the input grammar, beyond simple unambiguity. Knuth's LR(k) is a reasonable starting point, which we proceed to refine as follows. First, we address the question of which direction to refine the class of LR grammars: i.e., whether, for efficiency, we aim for a smaller class such as SLR or LALR grammars [DeR69]; or, for generality, we aim for a *larger* class. It is well-known that SLR and LALR, while being highly efficient in their implementation, fail to capture many natural constructs for which it is easy to write recursive descent parsers by hand. Thus, in order to be competitive with hand-written recursive-descent parsers, we should aim for a class of grammars *at least* as general as LR. On the other hand, conventional wisdom is that even LR(1) is often too inefficient to be practical.

In this work, we argue that both of these claims are incorrect: LR(1) parsing can be practical, with certain optimizations; and, moreover, even LR(1) parsing is not sufficiently general to handle languages that are intuitively "easy to parse" via hand-written recursive descent. To support these claims, we reason as follows.

LR parser generation can be practical. Traditionally, canonical LR is seen as impractical due to the proliferation of states in the generated automata. Specifically, the naive procedure calls for NFA states ($X \to \alpha \cdot \beta$, λ) for every dotted production $X \to \alpha \cdot \beta$ and for every possible k-follow[2] string λ. Since an industrial programming language might easily have over 1000 productions and over 50 symbols, this essentially rules out practicality of LR(k) for $k > 1$, and in some cases even $k = 1$ becomes slow enough to be questionable for a modern compiler toolchain.

In this work, we propose an optimized LR NFA construction procedure that drastically reduces the number of states required, which in turn improves both the efficiency and the debuggability of the resulting automata. At a high level, the construction proceeds by first building the LR(0) automaton (i.e., an automaton with no k-follow strings); determining what k-follow strings are possible for each state, and, of these, which are involved in simple LR conflicts; and propagating these conflicting strings through the automaton to determine a partition of each state's k-follow set. (We defer the full details of the construction procedure to Section 3.3.)

[2]Traditionally, the term "lookahead" is used to refer to both (a) the next k tokens in the input string, and (b) the k tokens that follow a given production according to a vertex in the LR automaton. To avoid confusion, in this work we use the term "lookahead" to refer only to the former, and the term "k-follow string" (or "k-follow set") to refer to the latter.

Empirically, we find that even in languages that are relatively difficult to parse, such as Golang, there are relatively few k-follow strings that are involved in simple LR conflicts. Moreover, when these strings are propagated backwards through the automaton, the partitions are often completely annihilated by other productions. (For instance, if $W \to X \cdot YZ$ and $Y \to \cdot AB$ are dotted productions, then a conflicting k-follow string on $(Y \to \cdot AB, \lambda)$ will not induce a partition of the k-follow strings on $W \to X \cdot YZ$ unless Z generates a string of length less than k—i.e., if $k = 1$, the empty string).

In addition to the improved automaton construction procedure, we also propose a refinement of the LR parsing paradigm which lends itself to more intuitive automata. Specifically, we examine the case of the *recursive-descent* (RD) parser: intuitively, it is easy to tell, at any point in time, what the parser is "doing". Namely, the stack trace indicates that, e.g., it is "parsing an expression", and further up the stack, we find that the expression is being parsed as the right-hand side of an assignment statement. These qualitative statements are much harder to read off of a DFA state in the LR construction, a fact which is largely responsible for the difficulties of debugging automatically generated parsers.

To remedy this, we introduce the concept of RD actions in an LR automaton. Specifically, in addition to "`Shift`" and "`Reduce` by production $\mathcal{P}$", our automata may call for the action "`Recur` on top-level symbol E" or "`Return` from the current `Recur` call". The action on a state, e.g., Stmt $\to E := \cdot E$, then becomes "`Recur` on E", rather than (as in standard LR) having no action itself, but generating ε prediction edges to all of the productions having E as their left-hand side. (We defer the full details to Section 3.6.) This simplification not only makes the resulting NFA states far more intuitive, but also significantly reduces the proliferation of states, by reducing the need for generated ε prediction edges.

Even canonical LR is not sufficiently general. For full industrial programming languages, we often require functionality that significantly exceeds the capabilities of canonical LR grammars. For instance:

1. Languages often require precedence specifications (e.g., $E \to E + E$ with precedence 1, and $E \to E \times E$ with precedence 2). While this requirement can, in principle, be satisfied by modifying the grammar to include new, explicit classes of nonterminals (in this case, "terms" T and "factors" F), this operation significantly complicates the resulting grammar and renders the resulting syntax trees much more distant from their intuitive meanings. We would like an approach that does not require a transformation of the input grammar to satisfy the parser generator.
2. Nonterminals may have semantic attributes—e.g., "expression in a type context" versus "expression in a value context"—that require separate treatment in order to be parsed unambiguously. While in principle this issue can be solved by creating a new nonterminal for each possible set of attributes, in practice this imposes untenable complexity on the grammar, as it requires a distinct nonterminal for every entry in the Cartesian product of attribute values (as well as again causing the resulting parse trees to be very far from their intuitive meanings).
3. Nonterminals may have *syntactic* attributes that are necessary for unambiguous parsing. Classic examples include the "`<>`" template syntax in C++ and Rust, in which the expression enclosed in angle brackets cannot be an unguarded greater-than expression (lest the parser attempt to close the angle brackets upon scanning the "`>`" operator). Golang also contains many examples of this phenomenon, particularly when the specification requires that certain constructs be parsed greedily—e.g., "`if * func () {}`" is a parse error, because "`func () {}`" must be parsed greedily as a lambda expression, leaving the if statement

without a body (despite the fact that the alternative interpretation, treating "`* func ()`" as a starred expression, would otherwise be permitted).

This type of requirement is notoriously difficult to capture in a declarative manner at the grammar level, a fact which is often invoked to support the conclusion that such languages can only be handled by hand-written recursive-descent parsers. In fact, however, we find that both cases can be represented by a form of (syntactic) attributes on nonterminals. In the case of angle-bracket templating, e.g., the inner expression must have the attribute that it is not an "`Expr.BinOp.Gt`" production; while in the case of Golang's "`if * func () {}`" example, we can require that the condition of an if statement have the attribute that it does not end with an unguarded function type "`func ()`" (a property which, in turn, is easy to specify at the level of productions of the grammar). We expand on this construction in more detail below.

4. When represented in their most natural form, grammar rules often require intermediate reductions which lead to conflicts. For instance, a simplified version of Golang's struct field declaration takes the form Field $\rightarrow *?\, E \mid E\, E$ (where "$*?$" here indicates an optional star token). Although this construction is clearly easy to parse, and poses no challenge to a hand-written recursive-descent parser, a direct implementation of an LR parser will fail: it will introduce an anonymous nonterminal for the "$*?$" expression (e.g., $X \rightarrow * \mid \varepsilon$), and, in cases in which the star is absent, will require the parser to decide whether to reduce by the production $X \rightarrow \varepsilon$ *immediately*, before seeing the token after the expression E in order to determine which of the two Field cases applies. Intuitively, this issue does not seem fundamental to the parsing problem, but it nevertheless requires a general solution if we are to use LR parsing for full industrial languages.
5. Finally, some common language constructs simply are not LR(k) at a fundamental level. One example of such a construct is the *deferred branch*: e.g., $S \rightarrow$ (Name := Expr) | (Expr = Expr), where we may have to make reduce decisions within Name and/or Expr without yet knowing which of the two we are parsing. If the parser could proceed in parallel, it could shortly discard one of the two branches upon seeing a ":=" or "=" token, but standard LR cannot predict this until seeing a potentially unbounded number of tokens of the Name or Expr instance.

In order to address these requirements, we introduce several new parser generation techniques, which we now describe.

Technique: Nonterminal attributes. To address requirements 1-3, we first give a construction which permits nonterminals to be accompanied by *attributes*, which can be used either syntactically or semantically. In our proposal, attributes can be either binary-valued (as in most syntactic attributes), or integer-valued (as in the case of operator precedence). Relationships between attributes are then represented by *constraints* on grammar rules. For example, in the case of operator precedence expressions, we might annotate rules as follows:

$$\begin{aligned} E \rightarrow E + E \quad & \text{rhs\_begin}[\text{pr}] \geq 1 \\ & \text{rhs\_end}[\text{pr}] \geq 2 \\ & \text{lhs}[\text{pr}] \leq 1 \end{aligned}$$

$$E \to E * E \quad \begin{array}{l} \text{rhs\_begin[pr]} \geq 2 \\ \text{rhs\_end[pr]} \geq 3 \\ \text{lhs[pr]} \leq 2 \end{array}$$

while, for constraints such as in Golang's "`if * func () {}`" dilemma, we might constrain the grammar rules as follows:

$$\begin{array}{rl} S \to \text{if } E_{[\tau]} \ \{\ldots\} & \text{rhs\_tag}_{[\tau]}\text{[mC]} \\ E \to \text{func ()} & \neg\text{lhs[mC]} \\ E \to E + E & \text{lhs[mC]} \leq \text{rhs\_end[mC]} \end{array}$$

(where the attribute "mC", intuitively, signifies an expression that does not end with an unguarded function-type expression).[3]

In order to implement these types of constraints, significant care is required so that compilation of the LR automata remains tractable. In particular, we restrict constraints to be *monotonic* in each attribute, and *separable* across attributes; in other words, we require every constraint to take one of the following forms:

$$\begin{array}{l} \text{lhs[attrA]} \leq C \\ \text{rhs[attrA]} \geq C \\ \text{lhs[attrA]} \leq \text{rhs[attrB]} \end{array}$$

(keeping the convention that, for boolean-valued attributes, "attrA ≥ 1" corresponds to the boolean constraint attrA, and "attrA ≤ 0" corresponds to $\neg$attrA).

These restrictions rule out compound formulas involving multiple attributes, as well as constraints that negate attributes; thus, e.g., in order to express "rhs[not attrA]", it is necessary to introduce a separate attribute, "rhs[attrNotA]". Despite the apparent severity of the restrictions, however, we have found that virtually all syntactic and semantic attributes of interest in industrial programming languages can be expressed in this fashion. Moreover, the constraints remain declarative and compositional,[4] and thus can be implemented naturally as an extension of the LR automata.

Technique: CPS transformation. To address requirement 4 (intermediate reductions), we propose a general grammar transformation, which we refer to as CPS since its structure is reminiscent of the *continuation-passing style* technique in functional programming. At a high level, the purpose of the CPS transformation is to defer the decision to reduce by a given production until the end of processing of the entire grammar rule from which the production is derived.

Concretely, we illustrate the CPS transformation on the example above, derived from Golang's grammar:

$$\text{Field.Embedded} \to *?\, E$$

[3] In fact, the implementation of operator precedence in terms of constraints is slightly more subtle than depicted here, because operators may have *left-precedence* and *right-precedence* attributes, which require separate variables (e.g., prL and prR). For instance, the C++ expression `x.y()` requires that "`.`" bind tighter than "`()`", but this should not prevent `x().y` from being a legal expression. A naive implementation, however, would stipulate that the left-hand side of a dot should have precedence at least as high as the dot expression itself, and thus would rule out the expression `x().y`. With separate left-precedence and right-precedence, we avoid this issue: a prefix operator only constrains the right-precedence of its operands, while a postfix operator only constrains the left-precedence.

[4] As opposed to constraints such as "rule $\mathcal{P}$ may not be followed by a { token", which are not compositional and may have unintended side-effects elsewhere in the grammar.

$$\text{Field.Standard} \rightarrow \ E\ E$$

(where "∗?" here indicates an optional star token). Naively, when we flatten these syntax rules into a standard context-free grammar, we would introduce an intermediate nonterminal (say, X), and productions:

$$\text{Field} \rightarrow X\ E$$

$$X \rightarrow *$$

$$X \rightarrow \varepsilon$$

$$\text{Field} \rightarrow E\ E$$

But this, of course, leads to the problem described above, namely that the parser must decide whether to reduce by the production $X \rightarrow \varepsilon$ before seeing the input tokens that follow the first expression E.

In the CPS-transformed version of such a grammar, we replace each intermediate nonterminal X with a new nonterminal X' which represents the concatenation of X with the entire remainder of the original rule in which X appears.[5] Thus, the CPS-transformed grammar would take the form:

$$\text{Field} \rightarrow X'$$

$$X' \rightarrow *E$$

$$X' \rightarrow E$$

$$\text{Field} \rightarrow E\ E$$

Formal details of the procedure are given in Section 3.2. Intuitively, we typically find that the CPS transformation does just what a human would do when we inline rules manually to resolve conflicts in traditional LR parser generation. By performing this transformation automatically, however, not only do we relieve the human programmer from performing this inlining task, but we also enable the source grammar (and hence the resulting syntax trees) to remain consistent with the programmer's intention, as expressed in the original grammar prior to CPS transformation.

Technique: Parallel processing via XLR parsing. To address requirement 5, we propose the following as a "safety net" for when LR, even in conjunction with the other techniques of this work, fails to capture a language of interest. Informally speaking, we define a t-parallel shift/reduce parser as one that may make nondeterministic shift/reduce choices, thereby spawning multiple parsing instances, but which remains bounded to t active instances throughout its execution on every input string. An XLR(k, t) grammar then is one that can be parsed by a t-parallel shift/reduce parser with k symbols of lookahead.

Our definition of XLR is similar to the established notion of Generalized LR (as in the ETL parsers), but the important difference is that in XLR, we do not memoize the intermediate parsing results for nonterminals on substrings. This means that for grammars that fail to be XLR(k, t) for bounded t, an XLR parser could require exponential time (potentially branching at every input symbol), as opposed to the cubic time of Earley's and Leo's algorithms; but, on the other hand, if t is bounded, then the XLR running time remains $O(tn)$, with a very small constant factor independent of $|\mathcal{G}|$. This allows XLR(k, t) parsers with small values of t to remain competitive

[5] We note that, for intermediate nonterminals generated from subexpressions of a given rule, there can be only one continuation of the nonterminal through the remainder of the rule, since the subexpression appears only once in the rule. For intermediate nonterminals generated as a result of iteration (e.g., a list of expressions), the translation is somewhat more complex, but follows the same principle.

with hand-written recursive-descent parsers, where the ETL algorithms would not be competitive. Intuitively, XLR corresponds to recursive-descent parsing with a bounded amount of backtracking.

Technique: Conflict tracing. Finally, we introduce a technique which does not expand the class of grammars that our parser generator recognizes, but which nevertheless greatly improves usability. Specifically, we observe that one of the most common difficulties in using LR parser generators in the wild is the opacity of conflicts. When a conflict arises, it is often unclear which rules are responsible for it, or what an example input might look like that would trigger the conflict.

To remedy this, we introduce a method of tracing conflicts in the LR automata, which enables the reconstruction of a minimal pair of "confusing inputs" which triggers the conflict. We implement conflict tracing by executing the standard subset construction to convert the LR NFA to a DFA, and noting that, if conflicts occur, then some DFA vertices will have conflicting accept actions. For each conflicting DFA vertex v, we read off the labels that lead from the start vertex to v. Then, we return to the original NFA, performing a search to identify paths whose labels concatenate to the conflicting string in question. The result is a collection of two (or more) paths in the NFA, all traversed as a result of the same string of symbols, which lead to distinct LR actions. The completion of the "confusing inputs" can then be read off of the remainders of each of the dotted productions along each path, since these are precisely what is needed to complete the given productions and pop them off of the shift/reduce parsing stack.

Empirically, we have found that in translating real industrial grammars, this conflict tracing method almost always pinpoints the precise cause of LR conflicts, and the "confusing inputs" are indeed confusing to a human observer. We also observe a surprising fact about the subset construction: for real languages, the DFA often contains *fewer* vertices than the original NFA, despite the fact that in principle it could have exponential size. We attribute this fact to the intuitive meaning of the LR DFA: for real languages, in order to be "non-confusing" to a human parser, the number of "modes" that the parser can be in should be directly related to the size of the grammar.

1.5 Related work

The study of modern bottom-up (shift/reduce) parsing was pioneered by Knuth [Knu65], who first defined the LR(k) class of grammars. Subsequent work by DeRemer [DeR69] and others led to refinements including the definition of SLR(k) and LALR(k) grammars, which are less expressive than LR(k) but which enable much more efficient parser generation, and inspired the initial development of popular parser generation tools such as Yacc [Joh79] and Bison [Cor85]. More recently, a number of software tools, including Bison [Cor85], Yacc++ [Ewi97], and gocc [Sch12] have also implemented full LR parser generation. As discussed above, our conclusion is that even LR(k)—the most general of this family—is, on its own, not sufficiently general to capture the grammars of real industrial languages in a tractable manner. However, we show that when combined with a number of features described in the present work (e.g., optimized LR automata, monotonic attributes, CPS transformation, and conflict tracing), the fundamental LR paradigm does indeed provide the basis for parser generation that is both general and practical.

Another line of work in bottom-up parsing is represented by the ETL family of algorithms [Ear70, Tom84, Leo91], sometimes referred to as Generalized LR (GLR), and implemented in software packages such as Marpa [Keg19] and Elkhound [MN04]. These parsers process their input on-line, left-to-right, but instead of maintaining a single shift/reduce stack, they maintain a dynamic programming table capturing substrings of the input generated by particular dotted productions of the grammar. The structure of this table means that unlike our parallel shift/reduce parsers,

GLR parsers maintain polynomial running time for all grammars. On the other hand, however, for grammars which are in fact XLR(k, t), our parallel shift/reduce parsers maintain running time $O(tn)$ with a small constant factor independent of $|\mathcal{G}|$.

There has also been significant development of the top-down family of parser generators, including tools such as ANTLR [PQ95] which support a superset of LL(k) grammars, as well as their parallel counterparts, referred to as Generalized LL (GLL). For the reasons described above, we believe that while top-down parsers are theoretically interesting, the grammars they support are not sufficiently general to capture full programming languages.[6] Notably, they are significantly more restrictive than LR(k), and we argue that even the latter is not sufficiently general in its standard formulation.

A completely different approach to parsing is taken by the *procedural* family of parser generators, which, instead of encoding a CFG directly in a declarative fashion, use the ordering of rules to dictate matching priority and resolve ambiguities. In this respect, they can be viewed as an optimized toolkit for constructing recursive-descent parsers more efficiently than writing code by hand—the parser definition describes which productions to try in which order, just as one does in backtracking recursive-descent. Examples of these parsers are the Parser Expression Grammars (PEGs) of [For04], the corresponding "packrat" family of parsers [For02], and a long line of work on parser combinators (e.g., the Parsec library in Haskell [LM01]). This structured approach is often superior to hand-written recursive descent in implementation cost and maintainability, though it still suffers from many of the same disadvantages: namely, there is no declarative context-free specification of the grammar, and as a result, it is often difficult to debug and verify that the grammar captures the programmer's intent. In this work, we confine our attention to fully-declarative rather than procedural parsing.

We also mention two existing algorithms for constructing optimized LR automata. First, there is a classic algorithm of Pager [Pag77]. Like our construction, it yields automata that are much smaller than standard LR, but the mechanism is different: rather than constructing LR(0) DFA states and inferring k-follow set refinements necessary to resolve conflicts, it constructs full canonical LR(1) states, then examines them to merge those that can be merged without creating new conflicts. Because of this, while the final resulting DFA may be quite small, the intermediate computation can be significantly more expensive due to the processing of the entire LR(1) automaton. Second, there is the IELR algorithm of Denny and Malloy [DM10]. IELR is somewhat similar to the backward phase of our algorithm, in that it identifies conflicts at reduction vertices and propagates them backwards to split earlier vertices. However, IELR operates on the DFA rather than the NFA, and first requires the formation of the LALR(1) automaton in order to begin splitting states.

Finally, we mention one other result that refines the implementation of LR(k) parsing in a similar direction to the present work. The tree automata of Adams and Might [AM17] are similar to our implementation of monotonic attributes; they differ in that they impose left-to-right sequencing on the attribute constraints (whereas attributes can refer to arbitrary constituents of a grammar rule), and require productions to be duplicated (whereas attributes are boolean or integer values layered independently on top of the standard context-free grammar rules).

2 Definitions

Fix a context-free grammar $\mathcal{G}$ over nonterminal alphabet Λ, terminal alphabet Σ, and extended terminal alphabet $\Gamma = \Sigma \cup \{\dashv\}$, where $\dashv$ is the standard right-end sentinel marker. We assume

[6] We note that languages such as Python, which have BNF grammars that are used in top-down parsing style, actually depend on procedural features (e.g., PEGs) rather than being purely declarative grammars.

the start symbol S of $\mathcal{G}$ does not appear in the right-hand side of any production, and appears in the left-hand side of exactly one production (this is without loss of generality, since we may always introduce a new start symbol $S' \to S$). The empty string, as usual, is denoted by ε. We denote concatenation of strings α, β as $\alpha\beta$, and we generalize this in the standard way to concatenations that include sets of strings.

Definition 2.1 (k-Prefix Set)**.** Fix an integer $k \geq 0$, and a set of strings $T \subseteq \Gamma^*$. We define the k-prefix set, $\text{First}_k(T)$, as the set of all strings $x_1 \cdots x_{\min(k,r)}$ for $x_1 \cdots x_r \in T$.

Definition 2.2 (Generated Set)**.** Fix a set of strings $T \subseteq (\Lambda \cup \Gamma)^*$. We define the generated set, $\text{Gen}(T)$, as the set of all strings y such that $x \stackrel{*}{\Rightarrow} y$ for some $x \in T$.

Definition 2.3 (Partition)**.** Sets $T_1, \ldots, T_m \subseteq S$ are a partition of S if $T_i \cap T_j = \emptyset$ for all $i \neq j$, and $\bigcup_i T_i = S$.

Definition 2.4 (Partition Refinement)**.** Suppose T and U are partitions of a set S. Then we define the refinement of T by U to be the partition $\{T_i \cap U_j : T_i \in T, U_j \in U\}$. (Note that this is a partition by construction, since if $(T_i \cap U_j) \cap (T_{i'} \cap U_{j'}) \neq \emptyset$, then $i = i'$ and $j = j'$.)

Definition 2.5 (k-Follow Set Partition)**.** Fix an integer $k \geq 0$. We define a k-follow set partition L of the grammar $\mathcal{G}$ to comprise the following, for each dotted production $\hat{\Pi}$ of $\mathcal{G}$:

- A universe $\mathcal{U}(\hat{\Pi}) \subseteq \Gamma^k$ of possible k-follow strings.
- A partition $L(\hat{\Pi})$ of $\mathcal{U}(\hat{\Pi})$.

Without loss of generality, we will always take $\mathcal{U}(\hat{\Pi})$ to be the set of strings in Γ^k that can ever follow the base production Π in derivations of the grammar $\mathcal{G}$; this can be computed efficiently by a flood-fill algorithm, as in standard SLR parsing.

In order to define the LR(k) automata, we also recount the definition of a nondeterministic finite automaton (NFA). We note that our definition differs somewhat from standard definitions, since we allow multiple actions on a given vertex, rather than at most one, and we designate particular pairs of actions as "conflicting". This aspect of the definition will be important in defining lookahead-dependent LR actions, since we will need to specify actions such as "a on lookahead λ, a' on lookahead μ" (which conflict only if $a \neq a'$ and $\lambda = \mu$).

Definition 2.6 (Nondeterministic Finite Automaton)**.** Fix sets V (the vertices), A (the alphabet), and T (the accepting actions), and assume a relation "conf" on $T \times T$, intuitively signifying when two actions conflict with one another. We extend the relation conf to sets of actions by specifying that for sets $U, V \subseteq T$, we have $\text{conf}(U, V)$ if for some $U_i \in U$ and $V_j \in V$, we have $\text{conf}(U_i, V_j)$; and, for a single set $U \subseteq T$, we have $\text{conf}(U)$ if $\text{conf}(U, U)$. Then an NFA N consists of the following.

- A designated vertex $v_s \in V$ (the starting vertex).
- For each vertex $v \in V$, a set of accepting actions, $T_v \subseteq T$.
- A set of edges $v \stackrel{\lambda}{\to} w$, with $v, w \in V$, and some label $\lambda \in A \cup \{\varepsilon\}$.

Intuitively, we will generally take $A = \Lambda \cup \Gamma$, V the set of LR vertices, and T the set of LR actions associated to each possible k-token lookahead in the input string.

Definition 2.7 (NFA Path (Sequence))**.** A path in an NFA N on input sequence $\tau = \tau_1 \cdots \tau_r$, where each $\tau_i \in A \cup \{\varepsilon\}$, consists of a sequence of vertices $v_1, \ldots, v_{r+1}$ such that $v_1 \in S$ is the starting vertex and, for each $i \in [r]$, there exists an edge from v_i to v_{i+1} on the corresponding label τ_i. The accepting action set for such a path is defined to be the accepting action set $T_{v_{r+1}} \subseteq T$ of the final vertex v_{r+1}.

Definition 2.8 (NFA Path (String))**.** We extend Definition 2.7 to strings $\zeta \in A^*$ as follows. P is a path in an NFA N on a string ζ if there exists a sequence $\tau = \tau_1 \cdots \tau_r$ such that P is a path in N on the sequence τ, and $\tau_1 \cdots \tau_r = \zeta$.

Definition 2.9 (NFA Conflict)**.** A conflict in an NFA N consists of two NFA paths on the same input string ζ whose accepting action sets conflict.

Definition 2.10 (Deterministic Finite Automaton)**.** Fix sets V (the vertices), A (the alphabet), and T (the accepting actions), and a relation "conf" on $T \times T$ as in Definition 2.6. Then a DFA D consists of the following.

- A starting vertex $S_0 \in V$.
- For each vertex $v \in V$, a set of accepting actions, $T_v \subseteq T$.
- A set of edges $v \xrightarrow{\lambda} w$, with $v, w \in V$ and $\lambda \in A$, such that for each vertex v and each label $\lambda \in A$, there exists exactly one vertex w with an edge $v \xrightarrow{\lambda} w$.

We extend Definitions 2.7 and 2.8 to DFA paths in the natural way.

Definition 2.11 (DFA Conflict)**.** A conflict in a DFA D consists of a path from the starting vertex to a vertex whose accepting action set conflicts.

We now present a definition of the LR(k) NFA, which generalizes both the SLR and canonical LR automata.

Construction 2.12 (LR(k) NFA)**.** Fix an integer $k \geq 0$, and a k-follow set partition L of the grammar $\mathcal{G}$ (Definition 2.5). We define the LR(k) NFA of $\mathcal{G}$ over L as follows (assuming the convention that accepting actions may have associated lookaheads, and two distinct actions conflict if they have the same lookahead).

- The vertices consist of all pairs $(\hat{\Pi}, \bar{\lambda})$ where $\hat{\Pi}$ is a dotted production of the grammar and $\bar{\lambda} \in L(\hat{\Pi})$.
- The starting vertex is $(S \to \cdot\, \eta, \bar{\lambda})$, where $S \to \eta$ is the production whose left-hand side is the start symbol S, and $\bar{\lambda}$ is the set in the partition such that $\dashv^k \in \bar{\lambda}$.
- The edges are determined as follows:
  - Prediction edges: for vertices $v = (X \to \alpha \cdot Y\beta, \bar{\lambda})$ and $w = (Y \to \cdot\, \gamma, \bar{\mu})$, there is an ε-edge from v to w whenever $\bar{\mu} \cap \mathrm{First}_k(\mathrm{Gen}(\beta\bar{\lambda})) \neq \emptyset$.
  - Step edges: for vertices $v = (X \to \alpha \cdot \tau\beta, \bar{\lambda})$ and $w = (X \to \alpha\,\tau \cdot \beta, \bar{\lambda})$, there is an edge from v to w with label τ (where τ is a single symbol, either terminal or nonterminal).
- The vertex accept actions are determined as follows:
  - For vertices $(X \to \alpha \cdot, \bar{\lambda})$, for every $\lambda \in \bar{\lambda}$, the associated action on lookahead λ is "Reduce $X \to \alpha$".
  - For vertices $(X \to \alpha \cdot \sigma\beta, \bar{\lambda})$, where $\sigma \in \Sigma$ is a terminal symbol, for every $\mu \in \mathrm{First}_k(\mathrm{Gen}(\sigma\beta\bar{\lambda}))$, the associated action on lookahead μ is "Shift".

Construction 2.13 (LR(k) DFA)**.** For integer $k \geq 0$, and a k-follow set partition L of $\mathcal{G}$, we define the LR(k) DFA of $\mathcal{G}$ over L to be the automaton obtained by running the standard subset construction on the LR(k) NFA (Construction 2.12), taking unions over the resulting accept action sets at each vertex.

For $k = 0$, there is only one possible k-follow set partition (namely, $L(\hat{\Pi}) = \{\{\varepsilon\}\}$ for every dotted production $\hat{\Pi}$), and thus we may refer to the LR(0) NFA/DFA without ambiguity. For $k > 0$, the LR(k) automata as we have defined them depend on the particular choice of partition L. If we take L to be the coarsest partition possible, then we recover the definition of the SLR automaton:

Definition 2.14 (SLR(k) NFA)**.** The *SLR(k) NFA* is the LR(k) NFA given by Construction 2.12 applied to the partition $L^*(\hat{\Pi}) = \{\mathcal{U}(\hat{\Pi})\}$ for every dotted production $\hat{\Pi}$.

On the other hand, if we take L to be the most refined partition possible, then we recover the standard definition of the canonical LR(k) automaton.

Definition 2.15 (Canonical LR(k) NFA)**.** The *canonical LR(k) NFA* is the LR(k) NFA given by Construction 2.12 applied to the partition $L^*(\hat{\Pi}) = \{\{\tau\} : \tau \in \mathcal{U}(\hat{\Pi})\}$ for every dotted production $\hat{\Pi}$.

We also note that if we consider the LR(k) NFA over some partition L coarser than L^*, we find that it is isomorphic to the NFA produced by taking the canonical LR(k) automaton and grouping vertices by the equivalence relation defined by the partition L. This property will be important when we consider the task of optimizing LR(k) automata, as we will ultimately aim to emulate the result of grouping the vertices of the canonical LR(k) automaton by some equivalence relation, but we will avoid explicitly constructing the canonical automaton, instead invoking Construction 2.12 on an appropriate partition.

Finally, for completeness, we also recount the standard stack-based LR parsing algorithm on an LR(k) DFA.

Construction 2.16 (Stack-Based LR Parsing Algorithm)**.** Fix a grammar $\mathcal{G}$ and an integer k and let D be a conflict-free LR(k) DFA (Construction 2.13) for $\mathcal{G}$. Let S_0 denote the starting state of D, and fix an input string $x = x_1 \dots x_n$. The stack-based LR parsing algorithm on x then operates as follows.

1. Initialize the state stack with a single element, S_0, and initialize the syntax stack to be empty. Initialize the cursor position $i = 1$.
2. Repeat:
   (a) If the state stack is empty, halt with failure.
   (b) Let v be the vertex on top of the state stack. Examine the current lookahead λ, i.e., the first k symbols of $x_i x_{i+1} \dots x_n \dashv^k$. If v has no action on λ, then halt with failure. Otherwise, act according to the action on λ:
      - If the action is "Shift", then set the current buffer symbol to x_i, push x_i onto the syntax stack, and increment i (unless $i = n + 1$, in which case halt with failure).
      - If the action is "Reduce $\Pi = X \to \alpha_1 \dots \alpha_s$" then pop s elements off of the syntax stack, construct a syntax element labeled "Π" with those s arguments as children, and push the resulting element back onto the syntax stack. Then, pop s elements off of the state stack, and set the current buffer symbol to X.
   (c) Let α denote the current buffer symbol.
      - If $\alpha = S$, then if $i = n + 1$, halt with success and return the top (i.e., only) element of the syntax stack; otherwise, halt with failure.
      - If $\alpha \neq S$, then let v' be the vertex for which there is an edge $v \xrightarrow{\alpha} v'$ (there is exactly one such v', since D is a DFA). Push v' onto the state stack.

3 Results

We now present the main theoretical contributions of this work, namely several new developments of the LR parsing paradigm. For simplicity of presentation, we introduce each development separately as an extension of canonical LR, though in our software implementation they are enabled simultaneously, along with a number of additional syntactic conveniences.

3.1 Grammar flattening

We begin by describing the basic mechanism by which we translate a full BNF-style grammar, which contains various syntactic conveniences, to a standard CFG. This transformation will take, e.g., rules such as the following for a multidimensional array index expression:[7]

$$E \to E \texttt{ [} E \ (\texttt{,}\ E)^* \ \texttt{,?} \texttt{]}$$

generate intermediate nonterminals X, Y for the nontrivial subexpressions of the rule, and produce flat CFG productions such as the following:

$$E \to E \texttt{ [} E\ X\ Y \texttt{]}$$
$$X \to \texttt{,}\ E\ X$$
$$X \to \varepsilon$$
$$Y \to \texttt{,}$$
$$Y \to \varepsilon$$

Other constructs are handled in a similar manner. We give a full description in Construction 3.2.

Definition 3.1 (BNF Grammar)**.** A BNF grammar over nonterminal symbols Λ and terminal symbols Σ consists of a sequence of *rules* of the form $X \to e$, where $X \in \Lambda$ is a nonterminal, and e is an expression defined inductively as follows:

- $e = \sigma$, where $\sigma \in \Lambda \cup \Sigma$ is a single symbol.
- $e = e_1\ e_2\ \ldots\ e_n$, a concatenation of expressions.
- $e = e_1 \mid e_2 \mid \ldots \mid e_n$, an alternation of expressions.
- $e = e_1?$, an optional expression.
- $e = e_1^*$, a repeated expression.

Construction 3.2 (Grammar Flattening)**.** Grammar flattening is an inductive definition, specified by the procedure Flatten(X, e), where X is a target symbol and e is a BNF expression as given by Definition 3.1. To flatten an entire grammar, invoke Flatten(X, e) for each BNF rule $X \to e$. The inductive cases of the procedure Flatten(X, e) as follows:

- $e = \alpha$ where α is a single symbol (either terminal or nonterminal). Emit the rule $X \to \alpha$.
- $e = e_1\ e_2\ \ldots\ e_n$, a concatenation. For each i, if e_i is a single symbol, let $\alpha_i = e_i$; otherwise, let α_i be a fresh nonterminal, and invoke Flatten(α_i, e_i). Emit the rule $X \to \alpha_1\ \alpha_2\ \ldots\ \alpha_n$.
- $e = e_1 \mid e_2 \mid \ldots \mid e_n$, an alternation. For each i, invoke Flatten(X, e_i).

[7] To avoid confusion, note that `[` and `]` are terminal symbols, while parentheses $(\ldots)$ group symbols at the BNF (meta) level.

- $e = e_1?$, an optional expression. Invoke Flatten$(X, e_1 \mid \varepsilon)$.
- $e = e_1^*$, a repeated expression. Let α be a fresh nonterminal. Invoke Flatten(α, e_1). Emit rules $X \to \alpha\, X$ and $X \to \varepsilon$.

Additional convenience constructs, such as $e = e_1^+$ (a repeated expression with at least one instance), and $e = \text{List}[e_1 :: e_2]$ (a list of expression e_1 with delimiter e_2), may be handled analogously to the repeated expression case ($e = e_1^*$) above. Our software implementation includes these and other syntactic conveniences.

3.2 CPS transformation

Once a grammar is flattened, we perform an additional transformation which we call CPS, as it is reminiscent of the *continuation-passing style* technique in functional programming. At a high level, the purpose of the CPS transformation is to eliminate conflicts by deferring reductions of intermediate productions until the latest possible moment, namely the end of processing the entire BNF rule. This is accomplished by replacing each intermediate nonterminal (i.e., those generated during flattening) with a CPS version, which corresponds to the content of that nonterminal concatenated with the entire remainder of the rule in which it appears. Following our example from Section 3.1, we would produce the following CPS-transformed rules:

$$E \to E\ \texttt{[}\ E\ X'$$
$$X' \to \texttt{,}\ E\ X'$$
$$X' \to Y'$$
$$Y' \to \texttt{,]}$$
$$Y' \to \texttt{]}$$

Note that the tail of the production, Y', is "absorbed" into the base case of the recursive symbol, X', rather than appearing in the inductive case.

Additionally, a key property of the CPS transformation is that the output remains linear in the size of the full input BNF rule. This may be somewhat counterintuitive, but can best be illustrated with an example such as the following BNF rule:

$$S \to (A \mid B)\ (C \mid D)\ (E \mid F)$$

The flattening procedure transforms such a rule into the following productions:

$$S \to X_0\ X_1\ X_2$$
$$X_0 \to A$$
$$X_0 \to B$$
$$X_1 \to C$$
$$X_1 \to D$$
$$X_2 \to E$$
$$X_2 \to F$$

The CPS transformation then yields the following grammar:

$$S \to Y_0$$
$$Y_0 \to A\ Y_1$$
$$Y_0 \to B\ Y_1$$
$$Y_1 \to C\ Y_2$$
$$Y_1 \to D\ Y_2$$
$$Y_2 \to E$$
$$Y_2 \to F$$

We note that although each original symbol X_i has been transformed to a CPS symbol Y_i which subsumes the remainder of its BNF rule, these symbols Y_i are reused in the tails of the productions, and thus avoid duplication of the tail contents (as would occur in a naive translation: $S \to A\ C\ E$, $S \to A\ C\ F$, etc.)

We now give the full definition of the transformation.

Construction 3.3 (CPS Transformation)**.** We define a nonterminal $X \in \Lambda$ to be *CPS-eligible* if it was generated as a fresh symbol during the BNF flattening procedure (Construction 3.2), as opposed to a top-level symbol of the BNF grammar. We further fix a subset of the CPS-eligible nonterminals that are specified to be *CPS-triggering*. (In a concrete implementation, these may be all CPS-eligible symbols; or, in an optimized implementation, only the subset of such symbols which might otherwise trigger an SLR reduction conflict if not CPS-expanded.) For nonterminals $X \in \Lambda$, let $\hat{X}$ be a fresh symbol (intuitively, the CPS-transformed version of X).

We now define mutually inductive procedures CPSProd($X \to \alpha_1 \dots \alpha_i \cdot \alpha_{i+1} \dots \alpha_r$, τ), where $X \to \alpha_1 \dots \alpha_i \cdot \alpha_{i+1} \dots \alpha_r$ is a dotted production of the flattened grammar, and τ is a sequence of symbols (intuitively, the tail context); and CPSSym(X, τ) where X is a symbol. To define the procedure CPSProd($X \to \alpha_1 \dots \alpha_i \cdot \alpha_{i+1} \dots \alpha_r$, τ), we consider the following cases:

- The dot is at the beginning of the production (i.e., $i = 0$). In this case, add the production $\hat{X} \to \tau$ to the CPS grammar.
- The dot is not at the beginning of the production, and the symbol α_i preceding the dot is not CPS-triggering. In this case, invoke CPSProd($X \to \alpha_1 \dots \alpha_{i-1} \cdot \alpha_i \dots \alpha_r$, $\hat{\alpha}_i \tau$).
- The dot is not at the beginning of the production, and the symbol α_i preceding the dot is CPS-triggering. We consider the following subcases:
  - The dotted production $X \to \alpha_1 \dots \alpha_i \cdot \alpha_{i+1} \dots \alpha_r$ is of the form $\alpha \to \gamma\alpha\,\cdot$, where α is CPS-triggering (i.e., the production was generated as a result of flattening a repeated expression). In this case, invoke CPSProd($\alpha \to \gamma \cdot \alpha$, $\hat{\alpha}$).
  - The dotted production is not of the indicated form. In this case, invoke CPSSym(α_i, τ), and finally invoke CPSProd($X \to \alpha_1 \dots \alpha_{i-1} \cdot \alpha_i \dots \alpha_r$, $\hat{\alpha}_i$).

To define the procedure CPSSym(X, τ), we proceed as follows:

- For each production $X \to \eta$ of the original flattened grammar, invoke CPSProd($X \to \eta\,\cdot$, τ).

Finally, to CPS-transform an entire flattened grammar, we simply invoke CPSSym(X, ε) for each non-CPS-triggering nonterminal X.

A key property of the CPS transformation is that it leaves the language generated by the grammar unchanged. We can show this by defining an appropriate relation between strings generated by symbols X and those generated by their CPS counterparts $\hat{X}$.

Definition 3.4 (CPS Tail Context)**.** For a symbol X of the original flattened grammar, we define the *CPS tail context* τ_X as follows:

- If X is a terminal symbol, define $\tau_X = \varepsilon$.
- If X is a nonterminal symbol, define τ_X to be the string such that $\mathrm{CPSSym}(X, \tau_X)$ is invoked in the CPS transformation of the grammar.

For a dotted production $\Pi = X \to \alpha_1 \dots \alpha_i \cdot \alpha_{i+1} \dots \alpha_r$, we define the tail context analogously, to be the string τ_Π such that $\mathrm{CPSProd}(\Pi, \tau_\Pi)$ is invoked in the CPS transformation.

It remains to show that this criterion is well-defined, i.e., that for each nonterminal X, the procedure $\mathrm{CPSSym}(X, \cdot)$ is invoked exactly once. This is evident for non-CPS-triggering nonterminals X; for these nonterminals, the tail context τ_X is simply ε, since CPSSym can only be invoked on X at top level. In the general case, we first note that by construction, CPSSym only makes recursive calls on the productions $X \to \eta$ for the input symbol X, and CPSProd makes a single call to CPSSym for each occurrence of a CPS-triggering symbol in the right-hand side of the production (excluding those arising from repeated expressions, $\alpha \to \gamma\alpha\cdot$). Since every CPS-triggering symbol is CPS-eligible, and every CPS-eligible symbol is reachable from a non-CPS-triggering symbol (and appears exactly once in the right-hand side of a production, excluding its occurrence in repeated expressions $\alpha \to \gamma\alpha\cdot$), we conclude that CPSSym is invoked exactly once on every nonterminal.

Lemma 3.5 (CPS Relation, Forward)**.** The following properties hold:

1. For every symbol X of the flattened grammar, for every terminal string $\lambda \in \Sigma^*$, if $X \overset{*}{\Rightarrow} \lambda$ in n steps, then $\hat{X} \overset{*}{\Rightarrow} \lambda\tau_X$ in n steps.
2. For every dotted production $\Pi = X \to \alpha_1 \dots \alpha_{i-1} \cdot \alpha_i \dots \alpha_r$, for every terminal string $\lambda \in \Sigma^*$, if $\alpha_i \dots \alpha_r \overset{*}{\Rightarrow} \lambda$ in n steps, then $\tau_\Pi \overset{*}{\Rightarrow} \lambda\tau_X$ in n steps.

Proof. By induction on the pair $(n, |\lambda|)$ in lexicographic order. To show property 1, first note that if $n = 0$, then X is a terminal symbol, and the claim trivially holds. So, suppose $n > 0$, and $X \to \alpha_1 \dots \alpha_r \overset{*}{\Rightarrow} \mu$ in n steps. Then invoking property 2 on the dotted production $X \to \cdot\, \alpha_1 \dots \alpha_r$, we find that $\tau_\Pi \overset{*}{\Rightarrow} \mu\tau_X$ in $(n-1)$ steps. Since the CPSProd procedure with the dot at the beginning outputs precisely the rule $\hat{X} \to \tau_\Pi$, we find that $\hat{X} \overset{*}{\Rightarrow} \lambda\tau_X$ in n steps, as desired.

To show property 2, first consider the case in which the dot is at the end. Then $\tau_\Pi = \tau_X$ by definition, and since $\alpha_i \dots \alpha_r$ is the empty string, λ is the empty string, and hence $\tau_\Pi = \lambda\tau_X \overset{*}{\Rightarrow} \lambda\tau_X$ in 0 steps, as desired. Now consider the case in which the dot is not at the end. Let $\Pi' = X \to \alpha_1 \dots \alpha_i \cdot \alpha_{i+1} \dots \alpha_r$. We consider the following cases:

- α_i is not CPS-triggering. Then $\tau_\Pi = \hat{\alpha}_i \tau_{\Pi'}$. Suppose $\lambda = \mu\nu$, where $\alpha_i \overset{*}{\Rightarrow} \mu$ in n_μ steps and $\alpha_{i+1} \dots \alpha_r \overset{*}{\Rightarrow} \nu$ in n_ν steps, with $n = n_\mu + n_\nu$. Now clearly $n_\mu, n_\nu \le n$, and furthermore:
  - If α_i is a terminal symbol, then $|\mu| = 1$, hence $|\nu| < |\lambda|$.
  - If α_i is a nonterminal symbol, then $n_\mu > 0$, hence $n_\nu < n$.

  We conclude inductively by property 1 that $\hat{\alpha}_i \overset{*}{\Rightarrow} \mu\tau_{\alpha_i} = \mu\varepsilon = \mu$ in n_μ steps, and by property 2 that $\tau_{\Pi'} \overset{*}{\Rightarrow} \nu\tau_X$ in n_ν steps. Hence $\tau_\Pi = \hat{\alpha}_i\tau_{\Pi'} \overset{*}{\Rightarrow} \mu\nu\tau_X = \lambda\tau_X$ in $n_\mu + n_\nu = n$ steps, as desired.

- α_i is CPS-triggering, and Π is of the form $\alpha_i \to \gamma \cdot \alpha_i$. Then $\tau_\Pi = \hat{\alpha}_i$, and we conclude inductively by property 1 that $\tau_\Pi = \hat{\alpha}_i \stackrel{*}{\Rightarrow} \lambda\tau_{\alpha_i}$ in n steps.
- α_i is CPS-triggering, and Π is not of the indicated form. Then $\tau_{\alpha_i} = \tau_{\Pi'}$, and $\tau_\Pi = \hat{\alpha}_i$. Suppose $\lambda = \mu\nu$, where $\alpha_i \stackrel{*}{\Rightarrow} \mu$ in n_μ steps and $\alpha_{i+1} \ldots \alpha_r \stackrel{*}{\Rightarrow} \nu$ in n_ν steps, with $n = n_\mu + n_\nu$. Now clearly $n_\mu, n_\nu \leq n$, and furthermore $n_\mu > 0$ (since α_i is a nonterminal), hence $n_\nu < n$. We conclude inductively by property 1 that $\hat{\alpha}_i \stackrel{*}{\Rightarrow} \mu\tau_{\alpha_i} = \mu\tau_{\Pi'}$ in n_μ steps, and by property 2 that $\tau_{\Pi'} \stackrel{*}{\Rightarrow} \nu\tau_X$ in n_ν steps. Hence $\tau_\Pi = \hat{\alpha}_i \stackrel{*}{\Rightarrow} \mu\tau_{\Pi'} \stackrel{*}{\Rightarrow} \mu\nu\tau_X = \lambda\tau_X$ in $n_\mu + n_\nu = n$ steps, as desired. □

Lemma 3.6 (CPS Relation, Reverse)**.** The following properties hold:

1. For every symbol X of the flattened grammar, if $\hat{X} \stackrel{*}{\Rightarrow} \lambda$ in n steps, then $X\tau_X \stackrel{*}{\Rightarrow} \lambda$ in n steps.
2. For every dotted production $\Pi = X \to \alpha_1 \ldots \alpha_{i-1} \cdot \alpha_i \ldots \alpha_r$, if $\tau_\Pi \stackrel{*}{\Rightarrow} \lambda$ in n steps, then $\alpha_i \ldots \alpha_r\tau_X \stackrel{*}{\Rightarrow} \lambda$ in n steps.

Proof. By induction on the pair $(n, |\lambda|)$ in lexicographic order. To show property 1, first note that if $n = 0$, then $\hat{X}$ is a terminal symbol, and the claim trivially holds. So, suppose $n > 0$, and $\hat{X} \to \eta \stackrel{*}{\Rightarrow} \lambda$ in n steps. Then the rule $\hat{X} \to \eta$ was added to the CPS grammar on an invocation CPSProd(Π, τ_Π) where $\Pi = X \to \cdot\, \alpha_1 \ldots \alpha_r$ and $\eta = \tau_\Pi$. By property 2, we have $\alpha_1 \ldots \alpha_r\tau_X \stackrel{*}{\Rightarrow} \lambda$ in $(n-1)$ steps, and hence $X\tau_X \to \alpha_1 \ldots \alpha_r\tau_X \stackrel{*}{\Rightarrow} \lambda$ in n steps.

To show property 2, first consider the case in which the dot is at the end. Then $\tau_\Pi = \tau_X$ by definition, and since $\alpha_i \ldots \alpha_r$ is the empty string, we have $\alpha_i \ldots \alpha_r\tau_X = \tau_X = \tau_\Pi \stackrel{*}{\Rightarrow} \lambda$ in n steps, as desired. Now consider the case in which the dot is not at the end. Let $\Pi' = X \to \alpha_1 \ldots \alpha_i \cdot \alpha_{i+1} \ldots \alpha_r$. We consider the following cases:

- α_i is not CPS-triggering. Then $\tau_\Pi = \hat{\alpha}_i\tau_{\Pi'}$. Suppose $\lambda = \mu\nu$, where $\hat{\alpha}_i \stackrel{*}{\Rightarrow} \mu$ in n_μ steps, and $\tau_{\Pi'} \stackrel{*}{\Rightarrow} \nu$ in n_ν steps, with $n = n_\mu + n_\nu$. Now clearly $n_\mu, n_\nu \leq n$, and furthermore:
  - If α_i is a terminal symbol, then $|\mu| = 1$, hence $|\nu| < |\lambda|$.
  - If α_i is a nonterminal symbol, then $n_\mu > 0$, hence $n_\nu < n$.

  We conclude inductively by property 1 that $\alpha_i = \alpha_i\tau_{\alpha_i} \stackrel{*}{\Rightarrow} \mu$ in n_μ steps, and by property 2 that $\alpha_{i+1} \ldots \alpha_r\tau_X \stackrel{*}{\Rightarrow} \nu$ in n_ν steps, and hence $\alpha_i \ldots \alpha_r\tau_X \stackrel{*}{\Rightarrow} \mu\nu = \lambda$ in $n_\mu + n_\nu = n$ steps, as desired.
- α_i is CPS-triggering, and Π is of the form $\alpha_i \to \gamma \cdot \alpha_i$. Then $\tau_\Pi = \hat{\alpha}_i$, and we conclude inductively by property 1 that $\alpha_i\tau_{\alpha_i} \stackrel{*}{\Rightarrow} \lambda$ in n steps.
- α_i is CPS-triggering, and Π is not of the indicated form. Then $\tau_{\alpha_i} = \tau_{\Pi'}$, and $\tau_\Pi = \hat{\alpha}_i$. We conclude inductively by property 1 that $\alpha_i\tau_{\Pi'} = \alpha_i\tau_{\alpha_i} \stackrel{*}{\Rightarrow} \lambda$ in n steps. Suppose $\lambda = \mu\nu$, where $\alpha_i \stackrel{*}{\Rightarrow} \mu$ in n_μ steps and $\tau_{\Pi'} \stackrel{*}{\Rightarrow} \nu$ in n_ν steps, with $n = n_\mu + n_\nu$. Now clearly $n_\mu, n_\nu \leq n$, and furthermore $n_\mu > 0$ (since α_i is a nonterminal), hence $n_\nu < n$. Property 2 now gives $\alpha_{i+1} \ldots \alpha_r\tau_X \stackrel{*}{\Rightarrow} \nu$ in n_ν steps, and hence $\alpha_i \ldots \alpha_r\tau_X \stackrel{*}{\Rightarrow} \mu\nu = \lambda$ in $n_\mu + n_\nu = n$ steps, as desired. □

Theorem 3.7 (CPS Equivalence)**.** Let $\mathcal{G}$ be a flattened grammar, and let CPS($\mathcal{G}$) denote the CPS-transformed version of $\mathcal{G}$. Then $\mathcal{G}$ and CPS($\mathcal{G}$) generate the same language.

Proof. Let S denote the start symbol of $\mathcal{G}$. By definition, S is not CPS-eligible and hence not CPS-triggering, so $\tau_S = \varepsilon$. It follows by Lemma 3.5 that for every $\lambda \in \Sigma^*$, if $S \stackrel{*}{\Rightarrow} \lambda$, then $\hat{S} \stackrel{*}{\Rightarrow} \lambda\tau_S = \lambda$. Similarly, it follows by Lemma 3.6 that for every $\lambda \in \Sigma^*$, if $\hat{S} \stackrel{*}{\Rightarrow} \lambda$, then $S\tau_S = S \stackrel{*}{\Rightarrow} \lambda$. We conclude that S and $\hat{S}$ generate identical sets of terminal strings, and hence $\mathcal{G}$ and CPS($\mathcal{G}$) generate the same language. □

We further note that, when a string $X \stackrel{*}{\Rightarrow} \lambda$ is generated according to the CPS-transformed grammar, then when $\hat{X} \to \eta \stackrel{*}{\Rightarrow} \lambda$ we have $X\tau_X \to \alpha_1 \dots \alpha_r \tau_X \stackrel{*}{\Rightarrow} \lambda$, where $X \to \alpha_1 \dots \alpha_r$ is the original grammar production which caused $\hat{X} \to \eta$ to be emitted in the CPSProd procedure. We exploit this observation in the LR parsers generated in our software implementation, by mapping each CPS production $\hat{X} \to \eta$ to its original $X \to \alpha_1 \dots \alpha_r$. Then, during execution, we maintain two stacks: one for the original AST elements, and one for the remaining elements from the CPS tail. When the CPS grammar's LR automaton calls for a reduction by $\hat{X} \to \eta = \eta_1 \dots \eta_m$, we look up its original production $X \to \alpha_1 \dots \alpha_r$. If $r \leq m$, we pop m elements from the first stack, reduce the earliest r into a result onto the first stack, and push the remaining unused $(m - r)$ elements onto the second stack; while if $r > m$, we pop r elements from the first stack along with $(r - m)$ additional elements from the second stack, and reduce all r into a result onto the first stack. This enables us to retain the semantics of the original grammar's AST, even though the shift/reduce instructions are generated by the CPS grammar's automaton.

3.3 k-follow set partitioning

Given a context-free grammar $\mathcal{G}$ (e.g., one generated by the CPS procedure of Construction 3.3), it remains to construct an LR automaton. In the case of lookahead $k = 0$, the standard construction (Construction 2.12) suffices, but in the more practical case of $k > 0$, in order to invoke the construction, we must specify a k-follow set partition. Taking the most refined possible partition recovers the canonical LR(k) automaton. However, this typically leads to an unnecessary proliferation of states, often making the procedure quite inefficient—a fact which has previously motivated weaker definitions such as SLR and LALR. In this work, by contrast, we give a new procedure to compute a k-follow set partition to be used in Construction 2.12, which retains the full expressive power of canonical LR, but which typically requires only a small fraction of the number of states.

As discussed in Section 2, we will reason about such k-follow set partitions as equivalence relations over sets of states in the canonical LR NFA (though we will still use Construction 2.12 to avoid explicitly constructing the canonical NFA). Specifically, we will first present algorithms for optimizing arbitrary NFAs by merging states, then describe how to map such algorithms onto the definition of k-follow set partitions.

Definition 3.8 (Forward-Admissible Initial Partition)**.** A partition L of the vertices V of an NFA is a forward-admissible initial partition if the starting vertex v_s is in a singleton set $\{v_s\} \in L$.

Definition 3.9 (Forward-Equivalent Vertices)**.** Vertices v_1, v_2 are forward-equivalent ($v_1 \sim_{\text{fwd}} v_2$) if for every sequence $\tau = \tau_1 \cdots \tau_k$ with $\tau_i \in A \cup \{\varepsilon\}$, v_1 is reachable from the starting vertex on τ if and only if v_2 is reachable from the starting vertex on τ.

Lemma 3.10 (Forward-Equivalent Reachability Preservation)**.** Let $\bar{N}$ be the NFA produced by grouping N by an equivalence relation $\sim$, with $\sim$ a refinement of $\sim_{\text{fwd}}$ (Definition 3.9). Then for every sequence $\tau = \tau_1 \cdots \tau_k$ with $\tau_i \in A \cup \{\varepsilon\}$, for every vertex $\bar{v}$ reachable on τ from the starting vertex in $\bar{N}$, for every $v \in \bar{v}$ the vertex v is reachable on τ from the starting vertex in N.

Proof. By induction on k. Clearly if $k = 0$, then $\bar{v}$ is the starting vertex in $\bar{N}$, and hence there exists $v \in \bar{v}$ the starting vertex in N. Since $\bar{v}$ is an equivalence class under $\sim$, and hence a subset of some equivalence class under $\sim_{\text{fwd}}$, it follows that every $v \in \bar{v}$ is the starting vertex in N, as desired.

For $k > 0$, we have $\tau = \tau_1 \cdots \tau_k$ for $\tau_i \in A \cup \{\varepsilon\}$, and there exists an edge $\bar{v}_i \stackrel{\tau_i}{\to} \bar{v}_{i+1}$ for each $i \leq k$. The inductive hypothesis gives that every $v_k \in \bar{v}_k$ is reachable on $\tau_1 \cdots \tau_{k-1}$ from the starting vertex in N. By definition of $\bar{N}$, there exists some $v_k \in \bar{v}_k$, $v_{k+1} \in \bar{v}_{k+1}$ such that the

edge $v_k \xrightarrow{\tau_k} v_{k+1}$ exists in N; hence, this particular v_{k+1} is reachable on τ from the starting vertex in N. Since $\bar{v}_{k+1}$ is an equivalence class under $\sim$, and hence a subset of some equivalence class under $\sim_{\text{fwd}}$, it follows that every $v_{k+1} \in \bar{v}_{k+1}$ is reachable on τ from the starting vertex in N, as desired. □

Lemma 3.11 (Forward-Equivalent Conflict Preservation)**.** Let $\bar{N}$ be the NFA produced by grouping N by an equivalence relation $\sim$, with $\sim$ a refinement of $\sim_{\text{fwd}}$ (Definition 3.9). Then there is an NFA conflict in $\bar{N}$ if and only if there is an NFA conflict in N.

Proof. Clearly if there is a conflict in N, then there is a conflict in $\bar{N}$, since $\bar{N}$ is a coarser grouping than N. To show the converse, suppose there is a conflict in $\bar{N}$, so that there is a path from $\bar{v}_0$ to $\bar{v}_f$ on some sequence τ and a path from $\bar{v}_0$ to $\bar{v}'_f$ on some sequence τ' (where τ, τ' yield identical strings when each concatenated), such that $\bar{v}_f$ and $\bar{v}'_f$ have conflicting actions in $\bar{N}$. Then by definition, there exist $v_f \in \bar{v}_f$ and $v'_f \in \bar{v}'_f$ such that v_f, v'_f have conflicting actions in N. For such v_f, v'_f, Lemma 3.10 shows that v_f and v'_f are each reachable on, resp., τ and τ', from the starting vertex in N. Hence there is a conflict in N, as desired. □

Definition 3.12 (Forward Refinement Algorithm)**.** Fix an initial partitioning L_0 of the vertices V. Repeat until convergence: for each $Z \in L_i$, and each label $\sigma \in A \cup \{\varepsilon\}$, define $Z[\sigma]$ to be the set of vertices z for which an edge $z' \xrightarrow{\sigma} z$ exists for some $z' \in Z$. Replace L_i with its refinement by each $Z[\sigma]$ in turn, obtaining L_{i+1}. Output the converged partition $L_0, L_1, \ldots \to L$.

Lemma 3.13 (Forward Refinement Correctness)**.** Suppose L_0 is a forward-admissible initial partition of the vertices of the NFA N, and let L be the result of the forward refinement algorithm (Definition 3.12) on the initial partition L_0. Fix a sequence $\tau = \tau_1 \cdots \tau_k$ and let $R(\tau)$ denote the set of vertices reachable on τ in N. Then for every set $Z \in L$, either $Z \subseteq R(\tau)$ or $Z \cap R(\tau) = \emptyset$.

Proof. We will show, by induction on i, that the partition L_i at the i^{th} step of the algorithm satisfies the corresponding property: namely, for every sequence $\tau = \tau_1 \cdots \tau_i$, for every set $Z \in L_i$, either $Z \subseteq R(\tau)$ or $Z \cap R(\tau) = \emptyset$. Clearly the claim holds for $i = 0$, since L_0 is forward-admissible. Suppose $i > 0$, and fix a sequence $\tau = \tau_1 \cdots \tau_i$. We may conclude inductively that for every $Z \in L_{i-1}$, either $Z \subseteq R(\tau_1 \cdots \tau_{i-1})$ or $Z \cap R(\tau_1 \cdots \tau_{i-1}) = \emptyset$. In other words, $R(\tau_1 \cdots \tau_{i-1})$ is exactly the union of those $Z \in L_{i-1}$ which have nonempty intersection with it. On the other hand, the union over such Z of vertices reachable on label τ_i from Z is precisely $R(\tau)$. Since L_i has been refined by the image of each such Z under τ_i, we conclude that L_i has been refined by sets whose union is precisely $R(\tau)$, and the claim follows. □

Lemma 3.14 (Forward Refinement Relation)**.** Suppose L_0 is a forward-admissible initial partition of the vertices of the NFA N, and let L be the result of the forward refinement algorithm (Definition 3.12) on the initial partition L_0. Let $\sim$ be the equivalence relation induced by L. Then $\sim$ is a refinement of $\sim_{\text{fwd}}$.

Proof. Immediate by Lemma 3.13 and the definition of $\sim_{\text{fwd}}$. □

Lemma 3.14 shows that the conditions of Lemma 3.11 apply, and hence that for the NFA $\bar{N}$ produced by grouping by the partition of the forward refinement procedure, there will be a conflict if and only if there was a conflict in the original NFA N. In practice, of course, the grouped NFA $\bar{N}$ may have significantly fewer states.

To further optimize the NFA, we also introduce definitions for backward partitioning, analogous to the forward partitioning definitions given above.

Definition 3.15 (Backward-Admissible Initial Partition)**.** A partition L of the vertices V of the NFA N is a backward-admissible initial partition if, for every lookahead $\mu \in \Gamma^k$, and every pair of vertices v_1, v_2 with conflicting actions on μ reachable on the same sequence τ from the starting vertex in N, the following holds. Let $\bar{v}_1, \bar{v}_2$ be the partition sets of, resp., v_1, v_2. Then for every $v_1' \in \bar{v}_1$ and every $v_2' \in \bar{v}_2$, the actions on v_1' and on v_2' conflict on lookahead μ.

We remark that one simple approach to establish a backward-admissible initial partition, which we will employ in the constructions below, is to determine the lookahead strings $\mu \in \Gamma^k$ that may potentially be involved in a conflict at each given dotted production $\hat{\Pi}$, and refine the partition of the corresponding NFA vertices to discriminate appropriately among these lookaheads. We defer the details to Definition 3.25, below.

Definition 3.16 (Backward-Equivalent Vertices)**.** Fix an initial partition L_0 of the vertices of the NFA N. Vertices v_1, v_2 are backward-equivalent with respect to L_0 ($v_1 \sim_{\mathrm{bwd}(L_0)} v_2$) if for every $Z \in L_0$, and for every sequence τ, some element of Z is reachable from v_1 on τ if and only if some element of Z is reachable from v_2 on τ.

Lemma 3.17 (Backward-Equivalent Reachability Preservation)**.** Fix an initial partition L_0 of the vertices of the NFA N. Let $\bar{N}$ be the NFA produced by grouping N by an equivalence relation $\sim$, with $\sim$ a refinement of $\sim_{\mathrm{bwd}(L_0)}$ (Definition 3.16). Then for every $Z \in L_0$, for every vertex $\bar{v}$ of $\bar{N}$, if $\bar{v}$ reaches some $\bar{w} \subseteq Z$ on sequence $\tau = \tau_1 \cdots \tau_k$ then for every $v \in \bar{v}$, v reaches some element of Z on τ.

Proof. By induction on k. Clearly if $k = 0$, then $\bar{v} = \bar{w} \subseteq Z$, so every $v \in \bar{v}$ reaches $v \in Z$ on τ. For $k > 0$, we have $\tau = \tau_1 \cdots \tau_k$ for $\tau_i \in A \cup \{\varepsilon\}$, and there exists an edge $\bar{v}_i \xrightarrow{\tau_i} \bar{v}_{i+1}$ for each $i \leq k$. The inductive hypothesis gives that for every $v_2 \in \bar{v}_2$, v_2 reaches some element of Z on $\tau_2 \cdots \tau_k$. By definition of $\bar{N}$, there exists some $v_1 \in \bar{v}_1$, $v_2 \in \bar{v}_2$ such that the edge $v_1 \xrightarrow{\tau_1} v_2$ exists in N; hence, this particular v_1 reaches some element of Z on τ. Since $\bar{v}_1$ is an equivalence class under $\sim$, and hence a subset of some equivalence class under $\sim_{\mathrm{bwd}(L_0)}$, it follows that every $v_1 \in \bar{v}_1$ reaches some element of Z on τ, as desired. □

Lemma 3.18 (Backward-Equivalent Conflict Preservation)**.** Let $\bar{N}$ be the NFA produced by grouping N by an equivalence relation $\sim$, with $\sim$ a refinement of $\sim_{\mathrm{bwd}(L_0)}$ (Definition 3.16). Then there is an NFA conflict in $\bar{N}$ if and only if there is an NFA conflict in N.

Proof. Clearly if there is a conflict in N, then there is a conflict in $\bar{N}$, since $\bar{N}$ is a coarser grouping than N. To show the converse, suppose there is a conflict in $\bar{N}$, so that there is a path from $\bar{v}_0$ to $\bar{v}_f$ on some sequence τ and a path from $\bar{v}_0$ to $\bar{v}_f'$ on some sequence τ' (where τ, τ' yield identical strings when each concatenated), such that $\bar{v}_f \subseteq Z$ and $\bar{v}_f' \subseteq Z'$ with $Z, Z' \in L_0$, and $\bar{v}_f, \bar{v}_f'$ have conflicting actions in $\bar{N}$. Then by Lemma 3.17, every $v_0 \in \bar{v}_0$ (including the starting vertex in N) reaches some $v_f \in Z$ on τ, and likewise every $v_0 \in \bar{v}_0$ reaches some $v_f' \in Z'$ on τ'. By Definition 3.15, the actions on v_f, v_f' conflict, i.e., there is a conflict in N. □

Definition 3.19 (Backward Refinement Algorithm)**.** Fix an initial partition L_0 of the vertices V of the NFA N. Repeat until convergence: for each $Z \in L_i$, and each label $\sigma \in A \cup \{\varepsilon\}$, define $Z[\sigma]$ to be the set of vertices z for which an edge $z \xrightarrow{\sigma} z'$ exists for some $z' \in Z$. Replace L_i with its refinement by each $Z[\sigma]$ in turn, obtaining L_{i+1}. Output the converged partition $L_0, L_1, \ldots \to L$.

Lemma 3.20 (Backward Refinement Correctness)**.** Suppose L_0 is a partition of the vertices of the NFA N, and let L be the result of the backward refinement algorithm (Definition 3.19) on the initial partition L_0. Fix $Z \in L_0$, and let $R_Z(\tau)$ denote the set of vertices that reach some element of Z on τ in N. Then for every set $Y \in L$, either $Y \subseteq R_Z(\tau)$ or $Y \cap R_Z(\tau) = \emptyset$.

Proof. We will show, by induction on i, that the partition L_i at the i^{th} step of the algorithm satisfies the corresponding property: namely, for every sequence $\tau = \tau_1 \cdots \tau_i$, for every set $Y \in L_i$, either $Y \subseteq R_Z(\tau)$ or $Y \cap R_Z(\tau) = \emptyset$. Clearly the claim holds for $i = 0$, since L_0 is a partition. Suppose $i > 0$, and fix a sequence $\tau = \tau_1 \cdots \tau_i$. We may conclude inductively that for every $Y \in L_{i-1}$, either $Y \subseteq R_Z(\tau_2 \cdots \tau_i)$ or $Y \cap R_Z(\tau_2 \cdots \tau_i) = \emptyset$. In other words, $R_Z(\tau_2 \cdots \tau_i)$ is exactly the union of those $Y \in L_{i-1}$ which have nonempty intersection with it. On the other hand, the union over such Y of vertices which reach Y on label τ_1 is precisely $R_Z(\tau)$. Since L_i has been refined by the preimage of each such Y under τ_1, we conclude that L_i has been refined by sets whose union is precisely $R_Z(\tau)$, and the claim follows. □

Lemma 3.21 (Backward Refinement Relation)**.** Suppose L_0 is a partition of the vertices of the NFA N, and let L be the result of the backward refinement algorithm (Definition 3.19) on the initial partition L_0. Let $\sim$ be the equivalence relation induced by L. Then $\sim$ is a refinement of $\sim_{\text{bwd}(L_0)}$.

Proof. Immediate by Lemma 3.20 and the definition of $\sim_{\text{bwd}(L_0)}$. □

Finally, we show how to adapt the procedures of Definitions 3.12 and 3.19 to operate on the canonical LR(k) NFA implicitly, without actually constructing it. In order to achieve this, we will determine a partition $L(\hat{\Pi})$ of the k-follow strings for each dotted production $\hat{\Pi}$, and partition the associated set of NFA vertices $\{(\hat{\Pi}, \lambda) : \lambda \in \mathcal{U}(\hat{\Pi})\}$ according to $L(\hat{\Pi})$ independently for each $\hat{\Pi}$. This means that our NFA vertex partitions may be more refined than strictly necessary, since they will by construction separate the vertices of distinct dotted productions $\hat{\Pi}, \hat{\Pi}'$. However, by making this simplification, we will avoid constructing the full canonical LR(k) NFA, resulting in a significantly more efficient NFA optimization procedure.

To begin with, we define the set of all strings $\lambda \in \Gamma^k$ which can possibly follow each production Π. Our definition follows the standard SLR flood-fill algorithm.

Definition 3.22 (Forward-Reachable k-Follow Strings)**.** For each production Π of the grammar $\mathcal{G}$, we define the set of forward-reachable k-follow strings $\lambda \in \Gamma^k$ of Π iteratively as follows.

- The string $\dashv^k$ is forward-reachable at the starting production $\Pi = S \to \eta$.
- For productions $\Pi = X \to \alpha Y \beta$ and $\Pi' = Y \to \gamma$, if λ is forward-reachable at Π, then for every $\mu \in \text{First}_k(\text{Gen}(\beta\lambda))$, μ is forward-reachable at Π'.

For ease of exposition, we also introduce a notion of lookaheads *compatible* with a given k-follow string (for a given dotted production):

Definition 3.23 (Compatible Lookaheads)**.** Let $\hat{\Pi} = X \to \alpha \cdot \beta$ be a dotted production. A lookahead $\mu \in \Gamma^k$ is $\hat{\Pi}$-*compatible* with a k-follow string $\lambda \in \Gamma^k$ if $\mu \in \text{First}_k(\text{Gen}(\beta\lambda))$.

We now define the set of lookaheads $\mu \in \Gamma^k$ that are potentially involved in a conflict, for each dotted production $\hat{\Pi}$.

Definition 3.24 (Potentially-Conflicting Lookaheads)**.** For a dotted production $\hat{\Pi}$, a lookahead $\mu \in \Gamma^k$ is *potentially-conflicting* if there exists a dotted production $\hat{\Pi}'$ and k-follow strings $\lambda \in \mathcal{U}(\hat{\Pi})$, $\lambda' \in \mathcal{U}(\hat{\Pi}')$ such that the following conditions hold:

- The vertices $(\hat{\Pi}, \{\varepsilon\})$ and $(\hat{\Pi}', \{\varepsilon\})$ are reachable on the same string $\zeta \in (\Sigma \cup \Lambda)^*$ in the LR(0) NFA.

- The vertices $(\hat{\Pi}, \{\lambda\})$ and $(\hat{\Pi}', \{\lambda'\})$ have conflicting actions on lookahead μ in the canonical LR(k) NFA.

Given these potentially-conflicting lookaheads, we can now form a partition of the k-follow sets of each dotted production that enforces the appropriate separations.

Definition 3.25 (Immediately-Conflicting Backward Partition)**.** The immediately-conflicting backward partition L_0 is defined as follows. For each dotted production $\hat{\Pi}$, let the partition $L_0(\hat{\Pi})$ be defined by an equivalence relation $\equiv_{\hat{\Pi}}$ as follows. For k-follow strings $\lambda, \lambda' \in \Gamma^k$, we have $\lambda \equiv_{\hat{\Pi}} \lambda'$ if, for every potentially-conflicting lookahead μ for $\hat{\Pi}$, the lookahead μ is $\hat{\Pi}$-compatible with λ if and only if μ is $\hat{\Pi}$-compatible with λ' (Definition 3.23).

Lemma 3.26. The partition L_0 given by Definition 3.25 is a backward-admissible initial partition (Definition 3.15).

Proof. Fix a pair of vertices $v_1 = (\hat{\Pi}_1, \lambda_1), v_2 = (\hat{\Pi}_2, \lambda_2)$ with conflicting actions on some lookahead $\mu \in \Gamma^k$, and fix $\lambda_1' \equiv_{\hat{\Pi}_1} \lambda_1$ and $\lambda_2' \equiv_{\hat{\Pi}_2} \lambda_2$. By definition of the canonical LR(k) NFA, we have that λ_1 is $\hat{\Pi}_1$-compatible with μ, and that λ_2 is $\hat{\Pi}_2$-compatible with μ. Moreover, since v_1, v_2 are reachable on the same string ζ in the canonical LR(k) NFA, we have $(\hat{\Pi}_1, \{\varepsilon\}), (\hat{\Pi}_2, \{\varepsilon\})$ each reachable on ζ in the LR(0) NFA, and thus μ is a potentially-conflicting lookahead (Definition 3.24). By Definition 3.25 we conclude that λ_1' is $\hat{\Pi}_1$-compatible with μ, and λ_2' is $\hat{\Pi}_2$-compatible with μ. Hence the actions on $(\hat{\Pi}_1, \lambda_1'), (\hat{\Pi}_2, \lambda_2')$ conflict on μ, as desired. □

Since Definition 3.25 gives a backward-admissible partition, the procedure of Definition 3.19 applies; moreover, such a procedure can be executed without constructing the full LR(k) automaton, by maintaining for each dotted production $\hat{\Pi}$ the current partitioning of the k-follow set on $\hat{\Pi}$ at each point during the procedure. Evidently, the forward procedure (Definition 3.12) can be executed in a similar manner, resulting in the following algorithm.

Construction 3.27 (Forward/Backward k-Follow Set Partition Algorithm)**.** Let $\mathcal{G}$ be a context-free grammar, let N_0 denote its LR(0) NFA, and let N denote its canonical LR(k) NFA. Perform the following steps.

1. Compute the forward-reachable k-follow set (Definition 3.22) for each dotted production $\hat{\Pi}$.
2. Form the LR(0) DFA by invoking the standard subset construction on N_0. Using the DFA to determine reachability on strings $\zeta \in (\Lambda \cup \Sigma)^*$, determine the potentially-conflicting lookaheads (Definition 3.24) for each dotted production $\hat{\Pi}$.
3. Construct the immediately-conflicting backward partition $L_{0,\text{bwd}}$ of the vertices in N (Definition 3.25), maintaining the partition implicitly via partitioning the set of forward-reachable k-follow strings for each production.
4. Execute the backward refinement algorithm (Definition 3.19) on the partition $L_{0,\text{bwd}}$ (again operating implicitly over the partitions of k-follow strings, rather than constructing the canonical NFA explicitly). We write $L_{\text{bwd}}(\hat{\Pi})$ to denote the resulting partition of k-follow strings for each production $\hat{\Pi}$.
5. Taking the constituent subsets $Z \in L_{\text{bwd}}(\hat{\Pi})$ as the new base elements, form a trivial partition $L_{0,\text{fwd}}(\hat{\Pi})$ over these elements, consisting of a single subset containing all subsets $Z \in L_{\text{bwd}}(\Pi)$ for each dotted production $\hat{\Pi}$. Since the new partition $L_{0,\text{fwd}}(\hat{\Pi})$ still already separates vertices of distinct dotted productions $\hat{\Pi} \neq \hat{\Pi}'$, it is forward-admissible (Definition 3.8).

6. Execute the forward refinement algorithm (Definition 3.12) on the partition $L_{0,\text{fwd}}$ (again operating implicitly over the partitions of sets of k-follow strings, rather than constructing the canonical NFA). We write $L(\hat{\Pi})$ to denote the resulting partition of k-follow strings for each production $\hat{\Pi}$.
7. Output the LR(k) NFA produced by invoking Construction 2.12 on the final partition $L(\hat{\Pi})$.

Theorem 3.28 (Forward/Backward Partition Correctness)**.** Let $\mathcal{G}$ be a context-free grammar, let N denote its canonical LR(k) NFA, and let N^* denote the NFA formed by running the forward/backward algorithm (Construction 3.27) on $\mathcal{G}$. Then there is a conflict in N^* if and only if there is a conflict in N.

Proof. Immediate from Lemmas 3.11 and 3.18. □

We note that other combinations of forward- and backward-partitioning are possible; however, empirically we find that a single backward pass followed by a single forward pass (i.e., the procedure of Construction 3.27) suffices to produce remarkably small automata in practice—the backward pass easily zeroes in on the "problematic" k-follow strings which must be distinguished from their complements, and the forward pass groups k-follow strings which are "similar" (e.g., binary operators with identical precedence).

3.4 Conflict tracing

Debuggability is crucial for practical automatic parser generation. In existing parser generators, the source of a conflict (shift/reduce, reduce/reduce) is often very difficult to diagnose, and new conflicts can easily arise from seemingly incidental changes to the BNF grammar. To remedy this, we introduce a method of tracing conflicts in the LR automata, which enables the reconstruction of a minimal "confusing input" which triggers the conflict.

To begin with, it will be useful to precompute, for each symbol of the grammar, a shortest string generated by that symbol. This can be done via a standard dynamic programming approach.

Construction 3.29 (Generated String Precomputation)**.** For a grammar $\mathcal{G}$ over nonterminals Λ and terminals Σ, initialize a table as follows. For terminals $\sigma \in \Sigma$, initialize $\sigma \mapsto (1, \sigma)$, and for nonterminals $\alpha \in \Lambda$, initialize $\alpha \mapsto (\infty, \bot)$. For each production $\Pi = X \to \cdots \sigma \cdots$ in which each terminal σ appears in the right-hand side (and Π is not already in the queue), enqueue Π. Then, while the queue is not empty, repeat:

1. Dequeue a production $\Pi = X \to \eta_1 \dots \eta_r$.
2. Suppose we currently have $\eta_i \mapsto (\ell_i, \lambda_i)$ in the table, for each $i \in [r]$. Compute $\ell = \sum \ell_i$, and $\lambda = \lambda_1 \cdots \lambda_r$ (where $\bot$ concatenated with any string is $\bot$).
3. If ℓ is strictly less than the current length entry for X in the table, then replace the current entry with $X \mapsto (\ell, \lambda)$, and for each production Π' in which X appears in the right-hand side (and Π' is not already in the queue), enqueue Π'.

Similarly, we can compute a shortest string generated by a given string (of terminals and/or nonterminals) which begins with a given prefix. We begin by tabulating these values for each symbol, then proceed to the case of general strings.

Definition 3.30 (Prefix Table)**.** Fix a grammar $\mathcal{G}$ over nonterminals Λ and terminals Σ and an integer $k \geq 0$. A k-prefix table for a string $\zeta \in (\Lambda \cup \Sigma)^*$ consists of a mapping $\Sigma^{\leq k} \to \Sigma^*$ such that for a given $\alpha \in \Sigma^{\leq k}$, the key α is present if and only if there exists $\beta \in \Sigma^*$ such that $\zeta \overset{*}{\Rightarrow} \beta$ and α

is the truncation to a prefix of at most k symbols of β; and, for such α that are present, we have $\alpha \mapsto \beta^*$, where β^* is a shortest such β.

Construction 3.31 (Prefix Table Precomputation)**.** Fix a grammar $\mathcal{G}$ over nonterminals Λ and terminals Σ and an integer $k > 0$. The output of the construction will be a k-prefix table (Definition 3.30) for each symbol of $(\Lambda \cup \Sigma)$, as well as a k-prefix table for the tail of each dotted production of $\mathcal{G}$ (i.e., for the string β for each dotted production $X \to \alpha \cdot \beta$). The construction proceeds as follows.

1. For each terminal $\sigma \in \Sigma$, initialize the k-prefix table to $\{\sigma \mapsto \sigma\}$, and enqueue the symbol σ.
2. For each nonterminal $\zeta \in \Lambda$, initialize the k-prefix table to $\{\}$.
3. For each dotted production tail $X \to \alpha \cdot \beta$ such that $\beta \neq \varepsilon$, initialize the k-prefix table to $\{\}$.
4. For each dotted production tail $X \to \beta \cdot$, initialize the k-prefix table to $\{\varepsilon \mapsto \varepsilon\}$, and enqueue the dotted production tail $X \to \beta \cdot$.
5. While the queue is not empty:
   - Dequeue an item (a symbol or dotted production tail). If the item is a symbol ζ, then enqueue every dotted production tail of the form $X \to \alpha\zeta \cdot \beta$ and continue from the top of the loop. If the item is a dotted production tail with the dot at the beginning, i.e., $\hat{\Pi} = X \to \cdot\,\alpha$, then for each entry $\gamma \mapsto \delta$ in the k-prefix table for $\hat{\Pi}$, add $\gamma \mapsto \delta$ to the k-prefix table for X (unless γ is already present and maps to a string at least as short as δ); enqueue X if this process resulted in any new entries in its k-prefix table; and continue from the top of the loop. Henceforth we assume the item is a dotted production tail such that the dot is not at the beginning, i.e., $\hat{\Pi} = X \to \alpha\zeta \cdot \beta$ for some symbol ζ.
   - For each entry $\gamma \mapsto \delta$ in the k-prefix table for ζ, and each entry $\gamma' \mapsto \delta'$ in the table for $\hat{\Pi}$, form the concatenation mapping $\hat{\gamma} \mapsto \hat{\delta}$, where $\hat{\gamma}$ is the truncation of $\gamma\gamma'$ to a prefix of at most k, and $\hat{\delta} = \delta\delta'$. For each such entry $\hat{\gamma} \mapsto \hat{\delta}$, add it to the k-prefix table for $\hat{\Pi}' = X \to \alpha \cdot \zeta\beta$ (unless $\hat{\gamma}$ is already present and maps to a string at least as short as $\hat{\delta}$). If this process resulted in any new entries in the k-prefix table for $\hat{\Pi}'$, then enqueue $\hat{\Pi}'$.

Construction 3.32 (Prefix-Matching Generated String Algorithm)**.** Fix a grammar $\mathcal{G}$ over nonterminals Λ and terminals Σ (where $\Gamma = \Sigma \cup \{\dashv\}$), an integer $k \geq 0$ and a target lookahead $\lambda = \lambda_1 \ldots \lambda_k \in \Gamma^k$, and an input string $\zeta = \zeta_1 \ldots \zeta_s \in (\Lambda \cup \Gamma)^*$. The output of the construction will be a shortest string generated by s that begins with the prefix λ, if one exists.

To compute such a string, perform a shortest-path search in the following graph. The vertices of the graph are pairs (i, j) for $i \in [0, k], j \in [0, s]$. The initial vertex is $(0, 0)$, and the target vertex is (k, s). There is a edge from (i, j) to $(i', j+1)$ of length ℓ if there is an entry $\alpha \mapsto \beta$ in the k-prefix table for ζ_{j+1} (Definition 3.30) such that either $\alpha = \beta = \lambda_{i+1} \ldots \lambda_{i'}$, or $i' = k$ and α begins with the prefix $\lambda_{i+1} \ldots \lambda_{i'}$; and β has length ℓ.

Equipped with the above constructions, we now present the full conflict tracing algorithm.

Construction 3.33 (LR Conflict Tracing Algorithm)**.** Let N be an LR(k) NFA for the grammar $\mathcal{G}$, and let D be its corresponding DFA. For each vertex $\bar{v}$ of D with conflicting accept actions $a, a' \in A$, perform the following steps to trace the conflict.

1. Perform a shortest-path search in D, from the starting vertex to the conflicting vertex $\bar{v}$, where the length of an edge with label $\alpha \in \Lambda \cup \Sigma$ is defined to be the length of the shortest string generated by α (Construction 3.29). Let $\zeta_1 \cdots \zeta_r$ denote the labels on the edges of the resulting path.

2. Perform a shortest-path search in $N \times \{0, \ldots, r\}$, where the starting vertex is $(v, 0)$ for v the starting vertex of N, and the edges are defined as follows.
   - If there is an edge $v \xrightarrow{\zeta_i} w$ in N, then there is an edge $(v,\, i-1) \to (w,\, i)$ in the product graph with length 0.
   - If there is an edge $v \xrightarrow{\varepsilon} w$ in N, then for all i there is an edge $(v,\, i) \to (w,\, i)$ in the product graph, each with length ℓ, where ℓ is the sum of lengths of the shortest strings generated by the tail of the production on w (i.e., if $w = (X \to \alpha \cdot \beta,\, \bar{\lambda})$, then ℓ is the sum of the lengths generated by the symbols in β).

   Let $v_0, \ldots, v_s$ denote a resulting NFA path from the starting vertex to a vertex $(v,\, r)$ whose accept action is a on lookahead λ, and $v'_0, \ldots, v'_{s'}$ denote a path to a vertex $(v',\, r)$ whose accept action is a' on lookahead λ.
3. Construct the string $\omega = \zeta_1 \ldots \zeta_r \beta_s \beta_{s-1} \ldots \beta_0$, where β_j is defined as follows:
   - If $j < s$ and the edge $v_j \to v_{j+1}$ is an ε-edge, then β_j is the proper tail of the production of v_j (i.e., β if the production of v_j is some $\Pi = \alpha \cdot Y\beta$).
   - If $j < s$ and the edge $v_j \to v_{j+1}$ is not an ε-edge, then $\beta_j = \varepsilon$.
   - If $j = s$, then β_j is the tail of the production of v_j (i.e., β if the production of v_j is some $\Pi = \alpha \cdot \beta$).

   Also construct the string $\omega' = \zeta_1 \ldots \zeta_r \beta'_s \beta'_{s-1} \ldots \beta'_0$, where β'_j is defined analogously to β_j, but for v'_i.
4. Construct the string $\hat{\omega}$ (resp., $\hat{\omega}'$) from ω (resp., ω') as follows. Substitute each nonterminal in $\zeta_1 \ldots \zeta_r$ with a minimal string that it generates (Construction 3.29), producing $\hat{\zeta}$. Then, using Construction 3.32, find a string $\hat{\beta}$ (resp., $\hat{\beta}'$) generated by $\beta_s \ldots \beta_0 \dashv^k$ (resp., $\beta'_s \ldots \beta'_0 \dashv^k$) which begins with the given conflicting lookahead λ (its existence follows inductively from the definition of the lookahead sets in Construction 2.12). Define $\hat{\omega} = \hat{\zeta}\hat{\beta}$ (resp., $\hat{\omega}' = \hat{\zeta}\hat{\beta}'$).
5. Output the pair of "confusing inputs" $\hat{\omega}$, $\hat{\omega}'$.

The "confusing inputs" $\hat{\omega}$, $\hat{\omega}'$ output by Construction 3.33 are confusing in the following sense. First, evidently they are both generated by the grammar, and have a common prefix $\hat{\zeta}$. Yet, at the point when the LR parser has processed the prefix $\hat{\zeta}$, there exist two different possible completions $(\hat{\omega}, \hat{\omega}')$, both with lookahead $\lambda \in \Gamma^k$, such that the LR automaton calls for distinct shift/reduce actions at that point in the string. In this sense, a "confusing input pair" generalizes the concept of an ambiguity in the grammar. If we had $\hat{\omega} = \hat{\omega}'$ (which can indeed arise in ambiguous grammars), then the fact that $\hat{\zeta}$ calls for conflicting actions would imply two distinct parse trees for the string $\hat{\omega} = \hat{\omega}'$. In the more general case, however, a confusing input need not indicate an ambiguity; only an LR(k) conflict. Empirically, we have found that in translating real industrial grammars, this conflict tracing method almost always pinpoints the precise cause of LR conflicts, and the "confusing inputs" are indeed confusing to a human observer.

One minor technical issue remains when we employ Construction 3.33 in practice. While the existence of a completion $\hat{\beta}$ matching the given lookahead $\beta_s \ldots \beta_0 \dashv^k$ is guaranteed in the canonical LR(k) automaton, it is not obvious that this also holds in the LR(k) automaton resulting from the forward/backward algorithm (Construction 3.27). To show this, we can generalize the results of the previous section to a modified LR NFA in which ε-edges are replaced by special symbols $\varepsilon_{\hat{\Pi},\hat{\Pi}'}$ that indicate which specific dotted production is predicted when traversing the edge. The more general analogues of Lemmas 3.10 and 3.17 then establish the existence of paths in the canonical LR(k) NFA, which match not only the symbol sequence but also the precise sequence of predicted productions, and hence the desired tail $\beta_s \ldots \beta_0 \dashv^k$.

3.5 Nonterminal attributes

In parsing industrial programming languages, we often find a need for non-context-free properties of nonterminals: either semantic attributes, such as "expression in a type context" versus "expression in a value context", or syntactic attributes, such as are needed in Golang's greedy parsing of expressions like "`if * func () {}`". To address these requirements, we give a construction which permits nonterminals to be accompanied by *attributes*, which can be either binary-valued (as in most syntactic attributes), or integer-valued (as in the case of operator precedence). Relationships between attributes are then represented by constraints on grammar rules.

We begin by defining the possible domains for nonterminal attributes, and proceed to define the syntax of attribute constraints on BNF grammar rules.

Definition 3.34 (Attribute Value Type)**.** An attribute value type is either boolean (i.e., $\{0, 1\}$) or integer-valued with range n (i.e., $\{0, 1, \ldots, n-1\}$).

Definition 3.35 (Nonterminal Attribute Domain)**.** Each nonterminal symbol has an *attribute domain* consisting of a mapping from attribute keys (fresh symbols α in some set A) to attribute value types (Definition 3.34).

Definition 3.36 (BNF Attribute Constraints)**.** Each rule of a BNF grammar is accompanied by a set of *attribute constraints*, each of one of the following forms.

- $\text{lhs}[\alpha] \leq b$
- $\mathcal{R}[\alpha] \geq b$
- $\text{lhs}[\alpha] \leq \mathcal{R}[\beta]$

where $\alpha, \beta \in A$ are attribute keys, b is an value in the attribute domain of the corresponding symbol, and $\mathcal{R}$ takes one of the following values:

- "rhs", referring to every subexpression of the right-hand side of the rule.
- "rhs_begin", referring to subexpressions which are (syntactically) the first in the right-hand side of the rule.
- "rhs_end", referring to subexpressions which are (syntactically) the last in the right-hand side of the rule.
- "rhs_mid", referring to subexpressions which are (syntactically) neither first nor last in the right-hand side of the rule.
- "$\text{rhs\_tag}_{[\tau]}$", referring to subexpressions which are explicitly tagged in the BNF syntax with the symbol τ.

Definition 3.37 (Flattened Attribute Constraints)**.** Each production $\Pi = X \to \alpha_1 \ldots \alpha_r$ of a flattened (context-free) grammar is accompanied by a set of *flattened attribute constraints*, each of one of the following forms.

- $\text{lhs}[\alpha] \leq b$
- $\text{rhs}_i[\alpha] \geq b$
- $\text{lhs}[\alpha] \leq \text{rhs}_i[\beta]$

where $i \in [r]$ refers to an index into the right-hand side of the production, and as above, α, β are attribute keys and b is an attribute in the domain of the corresponding symbol.

It is straightforward to adapt the grammar flattening procedure (Construction 3.2) and LR automaton definition (Construction 2.12) to incorporate attribute constraints of the form described in Definitions 3.36 and 3.37; we only mention a few technical details. First, when a grammar in BNF form is flattened to a context-free grammar, fresh symbols generated during the translation of the right-hand side inherit the attribute domains of their left-hand side (and propagate attributes via constraints accordingly)—e.g., in the flattening of a rule $E := (A \mid B)\, C$, a fresh symbol X is generated with $E \to X\, C$, $X \to A$, and $X \to B$; and as a result, X inherits the attribute domain of E, and the production $E \to X\, C$ carries the constraint $\mathrm{lhs}[\alpha] \leq \mathrm{rhs}_1[\alpha]$ for every attribute key α in that domain. This enables attributes to be carried through, e.g., via "rhs_begin" to A or to B. Note, however, that attributes are not carried through to right-hand side items when they are not in those items' domain; thus, for instance, a rule $E := x\, A$ where x is a terminal symbol implies no constraint on x, even if the BNF rule carries a rhs_begin constraint, since the attribute domain of a terminal symbol is empty by definition.

Second, when we form the LR(k) automata in the presence of attribute constraints, NFA vertices may carry a set of attribute bounds of the form $\mathrm{lhs}[\alpha] \geq b$, with each distinct set of attribute bounds forming a separate vertex. Because attribute constraints are separable and monotonic, a set of bounds of this form suffices to determine, in turn, the bounds on each of the symbols of the right-hand side, which themselves then become lhs constraints in the vertices predicted. Moreover, the sets of bounds that arise in practice remain quite manageable, owing to the fact that most constraints do not influence the predicted right-hand side and can hence be omitted—e.g., for a vertex $(X \to A \cdot B, -)$, a bound $\mathrm{lhs}[\alpha] \geq b$ has no influence, and can be dropped from the vertex, unless there is a constraint of the specific form $\mathrm{lhs}[\alpha] \to \mathrm{rhs}_2[\beta]$. (Even a constraint of the form $\mathrm{lhs}[\alpha] \to \mathrm{rhs}_1[\beta]$ has no effect, since the dot has already advanced past the first symbol of the right-hand side.) Once the LR(k) NFA has been formed with these attribute bounds, the DFA can then be formed via a modified subset construction: as usual, we use subsets of the NFA vertices, but the edges between them consist of concrete attribute values (rather than attribute bounds)—e.g., there may be an edge whose label is some symbol $\zeta[\alpha = 0,\, \beta = 2,\, \gamma = 1]$ where α, β, γ are attribute keys, and the values are consistent with the constraints on the production in question. This approach has the added benefit that only *inhabited* symbols $\zeta[\alpha, \beta, \gamma]$, i.e., only those attribute values which are actually attainable by the grammar, need be added as labels in the DFA.

In conjunction with the techniques described in other sections, this approach to expressing attribute constraints suffices to describe most semantic and syntactic properties of interest in real programming language grammars. For boolean-valued attributes (e.g., "type expression", "value expression"), the translation under this paradigm is clear: one may introduce attributes, e.g., T and V for type expressions and value expressions, respectively; and write, e.g., "ArrayTypeExpr := $\mathrm{Expr}_{[\tau]}\, [\, \mathrm{Expr}_{[\omega]}\,]$", with constraints such as "$\neg\mathrm{lhs}[V]$", "$\mathrm{rhs\_tag}_{[\tau]}[T]$", and "$\mathrm{rhs\_tag}_{[\omega]}[V]$". For attributes such as operator precedence, the translation is somewhat more involved and requires general integer-valued attributes. We describe one such translation here.

Definition 3.38 (BNF Precedence Stanza)**.** A precedence stanza for a BNF grammar is a sequence of items of the form $(\{\Pi_1, \ldots, \Pi_s\}, \mathcal{P})$, where $\{\Pi_1, \ldots, \Pi_s\}$ is a set of BNF grammar rules, and $\mathcal{P}$, an associativity specification, is one of {none, left, right, prefix, postfix}. We require each BNF rule Π to appear in at most one set $\{\Pi_1, \ldots, \Pi_s\}$, and we define the precedence level of Π to be the zero-based index of its containing set within the full sequence of the precedence stanza (if it appears). We also require that for every nonterminal symbol E in the BNF, if any one of the productions $E := \ldots$ has a precedence level, then all productions $E := \ldots$ have a precedence level.

Construction 3.39 (BNF Precedence Constraints)**.** Given a BNF precedence stanza (Definition 3.38), we define associated attributes and constraints as follows. First, we define two integer-

valued attributes prL, prR (intuitively, the left- and right-precedence), whose range is $\{0, \ldots, n-1\}$, where n is the number of precedence levels (i.e., number of items in the precedence stanza sequence). Then, for each BNF rule $E := \ldots$ which appears in the precedence stanza at precedence level ℓ, we add the following constraints depending on its associativity specification:

- none:
  - $\text{lhs}[\text{prL}] \leq \ell$
  - $\text{lhs}[\text{prR}] \leq \ell$
  - $\text{rhs}[\text{prL}] \geq \ell + 1$
  - $\text{rhs}[\text{prR}] \geq \ell + 1$
- left:
  - $\text{lhs}[\text{prL}] \leq \ell$
  - $\text{lhs}[\text{prR}] \leq \ell$
  - $\text{rhs\_begin}[\text{prR}] \geq \ell$
  - $\text{rhs\_mid}[\text{prL}] \geq \ell + 1$
  - $\text{rhs\_mid}[\text{prR}] \geq \ell + 1$
  - $\text{rhs\_end}[\text{prL}] \geq \ell + 1$
  - $\text{lhs}[\text{prL}] \leq \text{rhs\_begin}[\text{prL}]$
  - $\text{lhs}[\text{prR}] \leq \text{rhs\_end}[\text{prR}]$
- right:
  - $\text{lhs}[\text{prL}] \leq \ell$
  - $\text{lhs}[\text{prR}] \leq \ell$
  - $\text{rhs\_begin}[\text{prR}] \geq \ell + 1$
  - $\text{rhs\_mid}[\text{prL}] \geq \ell + 1$
  - $\text{rhs\_mid}[\text{prR}] \geq \ell + 1$
  - $\text{rhs\_end}[\text{prL}] \geq \ell$
  - $\text{lhs}[\text{prL}] \leq \text{rhs\_begin}[\text{prL}]$
  - $\text{lhs}[\text{prR}] \leq \text{rhs\_end}[\text{prR}]$
- prefix:
  - $\text{lhs}[\text{prR}] \leq \ell$
  - $\text{rhs\_mid}[\text{prL}] \geq \ell + 1$
  - $\text{rhs\_mid}[\text{prR}] \geq \ell + 1$
  - $\text{rhs\_end}[\text{prL}] \geq \ell + 1$
  - $\text{lhs}[\text{prR}] \leq \text{rhs}[\text{prR}]$
- postfix:
  - $\text{lhs}[\text{prL}] \leq \ell$
  - $\text{rhs\_begin}[\text{prR}] \geq \ell + 1$
  - $\text{rhs\_mid}[\text{prL}] \geq \ell + 1$
  - $\text{rhs\_mid}[\text{prR}] \geq \ell + 1$
  - $\text{lhs}[\text{prL}] \leq \text{rhs}[\text{prL}]$

We note that both left- and right-precedence are sometimes necessary in cases of mixed associativity. For instance, the C++ expression `x.y()` requires that "`.`" bind tighter than "`()`", but this should not prevent `x().y` from being a legal expression. A naive implementation, however, would stipulate that the left-hand side of a dot should have precedence at least as high as the dot expression itself, and thus would rule out the expression `x().y`. With separate left-precedence and right-precedence, we avoid this issue: a prefix operator only constrains the right-precedence of its operands, while a postfix operator only constrains the left-precedence.

3.6 Recursive-descent actions

One reason that hand-written recursive-descent parsers are often regarded as more intuitive than generated LR parsers is that it is usually clear, by examining the stack trace, what the parser is "doing" at any point in time (viz., "parsing an expression", "parsing an array literal", and so on). By contrast, a typical LR DFA state may refer to multiple grammar rules, and the intuitive meaning is often unclear (e.g., $X \to A \cdot B$ and $B \to \cdot CD$ are connected by an ε-edge in the NFA, and hence end up grouped into the same vertex in the DFA). In this section, we observe that we can recover much of the intuitive simplicity of recursive-descent parsing within the LR paradigm, by introducing a new class of actions for LR automata. First, we assume that some of the occurrences of nonterminals in the right-hand side of grammar rules are designated as "unfoldable", and if an occurrence is *not* unfoldable, then it must be predicted at the beginning of its occurrence in the input, in recursive-descent style. To make this distinction more precise, we describe the following modification to the LR NFA. (For simplicity, we describe only the case of $k = 0$, and omit the k-follow sets; the extension to $k > 0$ is straightforward and follows that of Construction 2.12.)

Construction 3.40 (RD NFA)**.** Fix a grammar $\mathcal{G}$. We define the RD(0) NFA of $\mathcal{G}$ as follows.

1. The vertices consist of pairs (Π, R), where Π is either a dotted production of $\mathcal{G}$ or the special production $\#R \to \cdot R$, and $R \in \Lambda \cup \{\bot\}$ is the nonterminal currently in left-recursive position (or $\bot$ if none).
2. The starting vertex is $(\#S \to \cdot S, S)$ where S is the start symbol of $\mathcal{G}$.
3. The edges are determined as follows:
   (a) Prediction edges: for every vertex $v = (X \to \alpha \cdot Y\beta, R)$, if either of the following criteria hold:
   - Y is designated unfoldable in the production $X \to \alpha \cdot Y\beta$, or:
   - $Y = R$ and $\alpha = \varepsilon$, i.e., the production is left-recursive on R (including the special production $\#R \to \cdot R$);

   then for every vertex $w = (Y \to \cdot\gamma, R)$, then there is an ε-edge from v to w.
   (b) Recur-step edges: for vertices $v = (X \to \alpha \cdot Y\beta, R)$ which do not meet the criteria of item 3(a), there is an edge with label RecurStep(Y) from v to $(\#Y \to \cdot Y, Y)$.
   (c) Step edges: for vertices $v = (X \to \alpha \cdot \tau\beta, R)$ and $w = (X \to \alpha\,\tau \cdot \beta, \bot)$, there is an edge from v to w with label τ (where τ is a single symbol, either terminal or nonterminal).
4. The vertex accept actions are determined as follows:
   (a) For vertices $(X \to \alpha\cdot, -)$, where X is not the special symbol $\#R$, the associated action is "Reduce $X \to \alpha$".
   (b) For vertices $(\#R \to R\cdot, -)$, the associated action is "Return".
   (c) For vertices $(X \to \alpha \cdot \sigma\beta, -)$, where $\sigma \in \Sigma$ is a terminal symbol, the associated action is "Shift".

(d) For vertices $v = (X \to \alpha \cdot Y\beta, -)$ which do not meet the criteria of item 3(a), the associated action is "Recur Y".

The RD DFA is defined, as usual, by applying the standard NFA/DFA subset construction to the RD NFA. We note that while it is possible for the RD automaton to have additional conflicts which the LR automaton would not (i.e., when some items conflict with the new "Recur"/"Return" actions), we can always avoid such conflicts in general by declaring the symbols in question to be unfoldable in their respective productions. In the extreme case, in which every symbol is unfoldable in every production, we recover the standard LR behavior.

We now present the modified stack-based LR parsing algorithm for such an RD DFA, an adaptation of the standard LR algorithm (Construction 2.16).

Construction 3.41 (Stack-Based RD Parsing Algorithm)**.** Fix a grammar $\mathcal{G}$ and an integer k and let D be a conflict-free RD(k) DFA for $\mathcal{G}$. Let S_0 denote the starting state of D, and fix an input string $x = x_1 \dots x_n$. The stack-based RD parsing algorithm on x then operates as follows.

1. Initialize the state stack with a single element, S_0, and initialize the syntax stack to be empty. Initialize the cursor position $i = 1$.
2. Repeat:
   (a) If the state stack is empty, halt with failure.
   (b) Let $v = (\Pi_v, R)$ be the vertex on top of the state stack. Examine the current lookahead λ, i.e., the first k symbols of $x_i x_{i+1} \dots x_n \dashv^k$. If v has no action on λ, then halt with failure. Otherwise, act according to the action on λ:
      - If the action is "Shift", then set the current buffer symbol to x_i, push x_i onto the syntax stack, and increment i (unless $i = n + 1$, in which case halt with failure).
      - If the action is "Reduce $\Pi = X \to \alpha_1 \dots \alpha_s$" then pop s elements off of the syntax stack, construct a syntax element labeled "Π" with those s arguments as children, and push the resulting element back onto the syntax stack. Then, pop s elements off of the state stack, and set the current buffer symbol to X.
      - If the action is "Recur Y", then set the current buffer symbol to RecurStep(Y).
      - If the action is "Return", then pop 2 elements off of the state stack (i.e., $(\#R \to R\cdot, R)$ and $(\#R \to \cdot R, R)$), and set the current buffer symbol to R.
   (c) Let α denote the current buffer symbol.
      - If $\alpha = S$, then if $i = n + 1$, halt with success and return the top (i.e., only) element of the syntax stack; otherwise, halt with failure.
      - If $\alpha \neq S$, then let v' be the vertex for which there is an edge $v \xrightarrow{\alpha} v'$ (there is exactly one such v', since D is a DFA). Push v' onto the state stack.

We also make one additional observation, concerning mutual left-recursion. In our definition of the RD NFA (Construction 3.40), we have specified in the criteria of item 3(a) that symbols which are immediately left-recursive are to be unfolded automatically. This is motivated by a number of common constructs, e.g., left-associative operators in operator-precedence expressions, where it would be clearly invalid to make an explicit Recur call (lest it trigger infinite recursion), but where left-recursion is nonetheless desired. However, there may be cases of mutual recursion (e.g., $A \to B \dots$ and $B \to A \dots$, where neither A nor B is marked unfoldable. This necessitates an additional check on the RD DFA: if any vertex is self-reachable via RecurStep edges, we must reject the grammar, as there exist input strings which may trigger infinite recursion.

3.7 Parallel shift/reduce parsing (XLR)

Our final modification to the LR paradigm is a very general mechanism, which we refer to as XLR (i.e., extended LR), which, when all else fails, permits LR parsers to "fork" parallel copies which can make different shift/reduce decisions in order to deal with conflicts. We emphasize that XLR should only be used as a last resort—we have found that when a parser requires XLR (as opposed to LR with our extensions), this generally indicates that parses are truly confusing for the human reader, and the grammar should be modified to avoid this. Nevertheless, we include a description of the XLR paradigm here for completeness.

To begin with, we give a definition of the XLR family of grammars, with LR as a special case.

Definition 3.42 (LR Parse)**.** Let $T \in \Sigma \cup \Lambda$ be a symbol of the grammar $\mathcal{G}$, and $\mathcal{P}$ be a parse tree for T on some input string $x = x_1 \dots x_n$. An *LR parse* of x for T, according to the parse tree $\mathcal{P}$, consists of a sequence $\mathcal{S}$ of operations "Shift(σ)" or "Reduce(Π)" (where Π is a production of $\mathcal{G}$), which describes the items encountered in a postorder traversal of $\mathcal{P}$.

Definition 3.43 (Partial LR Parse)**.** Let $T \in \Sigma \cup \Lambda$ be a symbol of the grammar $\mathcal{G}$, and let $x = x_1 \dots x_s$ be an input string. A *partial LR parse* of x for T with lookahead $\lambda \in \Gamma^k$ consists of a sequence $\mathcal{S}_{\text{part}}$ such that for some suffix $x_{s+1} \dots x_n$, with λ matching the first k symbols of $x_{s+1} \dots x_n \dashv^k$, and some sequence $\mathcal{S}$ an LR parse of x for T, $\mathcal{S}_{\text{part}}$ is the longest prefix of $\mathcal{S}$ that includes exactly s "Shift" operations.

When it is clear from context, we omit the symbol $T \in \Sigma \cup \Lambda$, and assume $T = S$, the starting symbol of the grammar $\mathcal{G}$.

Definition 3.44 (XLR)**.** For integers $k, t \geq 0$, we say that a grammar $\mathcal{G}$ is XLR(k, t) if for every string x and every lookahead $\lambda \in \Gamma^k$, x has at most t distinct partial LR parses with lookahead λ.

Evidently, a grammar $\mathcal{G}$ is XLR$(k, 1)$ if and only if it is LR(k). Moreover, we can adapt the standard stack-based LR parsing algorithm to support LR(k) automata with conflicts, via nondeterministic branching, enabling us to parse the XLR(k, t) grammars efficiently.

Construction 3.45 (Stack-Based XLR Parsing Algorithm)**.** Fix a grammar $\mathcal{G}$ and an integer k and let D be the canonical LR(k) DFA for $\mathcal{G}$ (not necessarily conflict-free). Let S_0 denote the starting state of D, and fix an input string $x = x_1 \dots x_n$. The stack-based XLR parsing algorithm on x then operates as follows.

1. Initialize the state stack with a single element, S_0, and initialize the syntax stack to be empty. Initialize the cursor position $i = 1$.
2. Repeat:
   (a) If the state stack is empty, halt with failure.
   (b) Let v be the vertex on top of the state stack. Examine the current lookahead λ, i.e., the first k symbols of $x_i x_{i+1} \dots x_n \dashv^k$. If v has no action on λ, then halt with failure. Otherwise, act according to the action or actions on λ, choosing among them nondeterministically[8] if there are multiple actions on λ:
      - If the action is "Shift", then set the current buffer symbol to x_i, push x_i onto the syntax stack, and increment i (unless $i = n + 1$, in which case halt with failure).

[8] Of course, we can simulate this on a deterministic machine by forking multiple copies of the stacks, and executing all extant (non-halted) copies in parallel, symbol-by-symbol on the input string.

- If the action is "Reduce $\Pi = X \to \alpha_1 \dots \alpha_s$" then pop s elements off of the syntax stack, construct a syntax element labeled "Π" with those s arguments as children, and push the resulting element back onto the syntax stack. Then, pop s elements off of the state stack, and set the current buffer symbol to X.

(c) Let α denote the current buffer symbol.

- If $\alpha = S$, then if $i = n + 1$, halt with success and return the top (i.e., only) element of the syntax stack; otherwise, halt with failure.
- If $\alpha \neq S$, then let v' be the vertex for which there is an edge $v \xrightarrow{\alpha} v'$ (there is exactly one such v', since D is a DFA). Push v' onto the state stack.

At the end of execution, if all runs halt with failure, then the entire process halts with failure; if multiple runs halt with success, then the entire process halts with failure (i.e., ambiguity); and finally, if exactly one run halts with success, then the entire process halts with success and returns the parse resulting from the successful run.

In order to establish the correctness of the XLR parsing algorithm, we first describe an alternate perspective on the execution of LR automata: specifically, rather than performing an LR parse in the standard manner via the LR(k) DFA, we observe that it can also be done (albeit perhaps not efficiently), nondeterministically, via the NFA.

Construction 3.46 (Nondeterministic LR Parsing Algorithm)**.** Fix a grammar $\mathcal{G}$ and an integer k and let N be the canonical LR(k) NFA (Construction 2.12) for $\mathcal{G}$. Let S_0 denote the starting state of N, and fix an input string $x = x_1 \dots x_n$. The nondeterministic LR parsing algorithm on x then operates as follows.

1. Initialize the state stack with a single element, S_0, and initialize the syntax stack to be empty. Initialize the cursor position $i = 1$.
2. Repeat:
   (a) If the state stack is empty, halt with failure.
   (b) Let v be the vertex on top of the state stack. Nondeterministically execute one of the following actions:
   - Choose a vertex w for which there exists an ε-edge $v \xrightarrow{\varepsilon} w$, and push w onto the state stack.
   - Examine the current lookahead λ, i.e., the first k symbols of $x_i x_{i+1} \dots x_n \dashv^k$. If v has no action on λ, then halt with failure. Otherwise, nondeterministically choose one of the actions on λ:
     – If the action is "Shift", then set the current buffer symbol to x_i, push x_i onto the syntax stack, and increment i (unless $i = n + 1$, in which case halt with failure).
     – If the action is "Reduce $\Pi = X \to \alpha_1 \dots \alpha_s$" then pop s elements off of the syntax stack, construct a syntax element labeled "Π" with those s arguments as children, and push the resulting element back onto the syntax stack. Then, pop $(s + 1)$ elements off of the state stack, and set the current buffer symbol to X.

     Let α denote the current buffer symbol.
     – If $\alpha = S$, then if $i = n + 1$, halt with success and return the top (i.e., only) element of the syntax stack; otherwise, halt with failure.
     – If $\alpha \neq S$, then let v' be the vertex for which there is an edge $v \xrightarrow{\alpha} v'$ (or, if none exists, halt with failure). Push v' onto the state stack.

At the end of execution, if all runs halt with failure, then the entire process halts with failure; if multiple runs halt with success, then the entire process halts with failure (i.e., ambiguity); and finally, if exactly one run halts with success, then the entire process halts with success and returns the parse resulting from the successful run.

We also introduce some notation for state stacks of the above algorithms. Specifically, we write:

$$T = T_0 \mapsto_{\tau_1} \ldots \mapsto_{\tau_m} T_m$$

to denote a stack T of the XLR algorithm with DFA vertices $T_0, \ldots, T_m$, such that $\tau_i \in \Sigma \cup \Lambda$ is the label which caused T_i to be pushed onto the stack. Similarly, we write:

$$U = U_0 \mapsto_{\sigma_1} \ldots \mapsto_{\sigma_r} U_r$$

to denote a stack U of the nondeterministic LR algorithm with NFA vertices $U_0, \ldots, U_r$, such that $\sigma_i \in \Sigma \cup \Lambda \cup \{\varepsilon\}$ is the label which caused U_i to be pushed onto the stack. We restrict our attention to *valid* stacks, i.e., those for which each edge described in the stack actually exists in the corresponding automaton.

Definition 3.47 (Related Stacks)**.** Let T be a state stack of the XLR algorithm, and let U be a state stack of the nondeterministic LR algorithm. We say T is *related* to U, denoted $T \triangleleft U$, if the sequence of symbols for T is identical to the sequence of non-ε symbols for U.

Lemma 3.48. Let $\sigma_1 \cdots \sigma_r \in (\Sigma \cup \Lambda)^*$ be a sequence of symbols, let $x_1 \cdots x_s \in \Sigma^*$ be an input string, let $\lambda \in \Gamma^k$ be a lookahead, and let $U = U_0 \mapsto_{\sigma_1} \ldots \mapsto_{\sigma_r} U_r$ be a valid stack of the nondeterministic LR algorithm such that U_r has an action on λ. If $\sigma_1 \cdots \sigma_r \overset{*}{\Rightarrow} x_1 \cdots x_s$ with shift/reduce sequence $\mathcal{S}_{\text{part}}$, then there exists $x_{s+1} \cdots x_n \in \Sigma^*$ with $\text{First}_k(x_{s+1} \cdots x_n \dashv^k) = \lambda$ such that $S \overset{*}{\Rightarrow} x_1 \cdots x_n$ with shift/reduce sequence $\mathcal{S}$, where $\mathcal{S}_{\text{part}}$ is a prefix of $\mathcal{S}$.

Proof. As in the conflict tracing algorithm, letting $U_i = (X_i \to \alpha_i \cdot \beta_i, \{\mu_i\})$, we construct the tail string $\beta_r \ldots \beta_0$, then choose $x_{s+1} \cdots x_n$ such that $\beta_r \ldots \beta_0 \overset{*}{\Rightarrow} x_{s+1} \cdots x_n$ and $\text{First}_k(x_{s+1} \cdots x_n \dashv^k) = \lambda$ (the existence of such $x_{s+1} \cdots x_n$ follows from the correctness of the LR(k) NFA construction). By a standard induction on the derivation, such a string takes the nondeterministic LR algorithm to completion. □

Lemma 3.49. Let $x = x_1 \ldots x_n$ be an input string, let $s \in \{0, \ldots, n\}$, and let λ be the lookahead for the partial input $x_1 \ldots x_s$ (i.e., the first k symbols of $x_{s+1} \ldots x_n \dashv^k$). Then the following are equivalent:

1. $\mathcal{S}_{\text{part}}$ is a partial LR parse of $x_1 \ldots x_s$ with lookahead λ (Definition 3.43).
2. For some run of the nondeterministic LR parsing algorithm (Construction 3.46) on x, with cursor at s, such that there is an action from the current state on lookahead λ, the sequence of shift/reduce operations that have been performed is $\mathcal{S}_{\text{part}}$.
3. For some run of the XLR parsing algorithm (Construction 3.45) on x, with cursor at s, such that there is an action from the current state on lookahead λ, the sequence of shift/reduce operations that have been performed is $\mathcal{S}_{\text{part}}$.

Proof. To show that (2) implies (1), we note that by correctness of the shift/reduce operations performed in the nondeterministic LR parsing algorithm, we have $\sigma_1 \cdots \sigma_r \overset{*}{\Rightarrow} x_1 \ldots x_s$ with reduction sequence $\mathcal{S}_{\text{part}}$, where $\sigma_1 \cdots \sigma_r$ are the labels on the stack. The claim then follows from Lemma 3.48.

To show that (3) implies (1), we note that by correctness of the shift/reduce operations performed in the XLR parsing algorithm, we have $\sigma_1 \cdots \sigma_r \stackrel{*}{\Rightarrow} x_1 \dots x_s$. Moreover, by correctness of the NFA/DFA subset construction, for a given valid stack T of the XLR algorithm, there exists a valid stack U of the nondeterministic LR algorithm, with $T \triangleleft U$, and with the same action on lookahead λ. The claim then follows again from Lemma 3.48.

To show that (1) implies (2), we observe that there is a run of the nondeterministic LR parsing algorithm that corresponds to any given partial parse, namely, one which nondeterministically decides to predict each production via an ε-edge precisely when its occurrence begins in the input.

Finally, to show that (2) implies (3), fix a run of the nondeterministic LR algorithm, and construct a run of the XLR algorithm as follows. We maintain the invariants, inductively, that the state stack T of the XLR algorithm is related to the state stack U of the nondeterministic LR algorithm ($T \triangleleft U$), and that the cursor of the XLR algorithm is at the same point as that of the nondeterministic LR algorithm.

- When the nondeterministic LR algorithm makes an ε-edge prediction, do nothing. This preserves relatedness of the stacks, since we do not count ε-edges in the path in the stack U.
- When the nondeterministic LR algorithm performs a "Shift" operation, perform a "Shift" operation. Such an action is permitted at the current state by the correctness of the NFA/DFA subset construction (since the stacks are related), and the cursor advances in both cases.
- When the nondeterministic LR algorithm performs a "Reduce" operation by a production Π, perform a "Reduce" operation by the same production Π. Such an action is permitted at the current state by the correctness of the NFA/DFA subset construction (since the stacks are related). Moreover, by construction, the top of the NFA stack must consist of vertices $(X \to \alpha \cdot Y\beta, -) \mapsto_\varepsilon (Y \to \cdot\sigma_1 \dots \sigma_s, -) \mapsto_{\sigma_1} \dots \mapsto_{\sigma_n} (Y \to \sigma_1 \dots \sigma_s \cdot, -)$. Since the stacks are related (and the XLR stack cannot contain ε-edges), popping $(s+1)$ items from the nondeterministic LR stack, and popping s items from the XLR stack, preserves relatedness of the stacks, as desired. □

Theorem 3.50 (Efficient XLR Parsing)**.** For fixed $k, t \geq 0$, the XLR(k,t) grammars can be parsed in linear time.

Proof. It suffices to show that Construction 3.45 runs in linear time. This, in turn, will follow if we show that at most t parallel copies of the execution are extant (non-halted) at any point in the input string. Suppose we have some t' parallel copies, with cursor s, each with an action on the current lookahead λ (this is without loss of generality, since a copy with no action on λ halts immediately with failure). Each such copy has a distinct sequence of shift/reduce operations. Moreover, by Lemma 3.49, the operation sequence of each such copy determines a distinct partial parse of the input string with the given lookahead λ. Since $\mathcal{G}$ is XLR(k,t), we conclude that $t' \leq t$, as desired. □

We remark that the XLR parsing algorithm is similar to generalized LR (GLR) parsing algorithms such as the ETL family, in that it allows for multiple partial parses to coexist simultaneously. It differs, however, in that the partial parses do not share state (e.g., reusing parses of substrings). This means that for some (non-XLR) grammars, XLR parsing may require exponential time, while the dynamic-programming approaches of ETL will always require at most cubic time. On the other hand, it also means that the constant factor on the running time of XLR parsing is very small, and does not depend on $|\mathcal{G}|$. As a result, for small t, XLR(k,t) parsing remains competitive with hand-written recursive-descent parsers, while algorithms such as ETL incur significant overhead.

Of course, the XLR algorithm would be of limited practical utility if we could not determine whether grammars of interest are XLR(k,t). Naively, Definition 3.44 seems quite intractable, as it quantifies over all input strings x. However, there are many possible heuristics that are effective in practice. We give one such approach here. (For simplicity, we confine our attention to the canonical LR(k) automaton; the extension to the optimized automata of Section 3.3 is straightforward.)

Definition 3.51 (Fork Point)**.** Let N be the canonical LR(k) NFA for the grammar $\mathcal{G}$. Suppose there is an LR conflict on some vertices $w_1, \ldots, w_d$, i.e., these vertices are each reachable from the starting vertex on some common string $\tau \in (\Lambda \cup \Sigma)^*$, and have pairwise distinct actions on some common lookahead $\lambda \in \Gamma^k$. We say that a vertex v is a *fork point* (of degree d) for the conflict if every path from the starting vertex to one of $w_1, \ldots, w_d$ passes through v exactly once, and the outgoing ε-edges $v \xrightarrow{\varepsilon} x$ can be partitioned into d sets $S_1, \ldots, S_d$ such that all paths from the starting vertex to w_i pass through an edge in S_i.

Definition 3.52 (Join-Safe Vertex)**.** Let N be the canonical LR(k) NFA for the grammar $\mathcal{G}$, and let v be a vertex of N. For an outgoing ε-edge $v \xrightarrow{\varepsilon} w$, if $w = (X \to \cdot\, \eta, -)$, define the *join tail* of the edge to be the string η. We say that v is *join-safe* if for every pair of paths $v \xrightarrow{\varepsilon} w_1 \to \ldots \to x_1$, $v \xrightarrow{\varepsilon} w_2 \to \ldots \to x_2$ on the same string τ such that x_1, x_2 have distinct actions on some common lookahead $\lambda \in \Gamma^k$, if δ_1 is the join tail of $v \xrightarrow{\varepsilon} w_1$ and δ_2 is the join tail of $v \xrightarrow{\varepsilon} w_2$, then there do not exist $z, z' \in \Sigma^*$ such that $\delta_1 \overset{*}{\Rightarrow} z$ and $\delta_2 \overset{*}{\Rightarrow} zz'$; and similarly there do not exist $z, z' \in \Sigma^*$ such that $\delta_2 \overset{*}{\Rightarrow} z$ and $\delta_1 \overset{*}{\Rightarrow} zz'$.

We remark that although the criterion of Definition 3.52 is still undecidable in general, in practical cases it tends to be a relatively straightforward theorem-proving task.

Definition 3.53 (Fork Degree)**.** Let N be the canonical LR(k) NFA for the grammar $\mathcal{G}$, and suppose that every LR conflict has a fork point that is join-safe. We define the *fork degree* of N to be the minimum value, over assignments of LR conflicts each to one of their respective join-safe fork points, of the maximum value, over all paths $v_1 \to \ldots \to v_n$ where v_1 is the starting vertex, of the product of the maximum fork degrees of each of $v_1, \ldots, v_n$, where we define the fork degree of a non-fork point to be 1. (Note that if a fork point always occurs on a reachable cycle in N, irrespective of which assignment is chosen, then the fork degree of N is ∞.)

Theorem 3.54 (XLR Fork Bound)**.** Let N be the canonical LR(k) NFA for the grammar $\mathcal{G}$, and suppose N has finite fork degree t. Then $\mathcal{G}$ is XLR(k,t).

Proof. Fix an assignment of LR conflicts to one of their respective join-safe fork points. Partition the partial runs of the nondeterministic LR parsing algorithm into equivalence classes by the sequence of reduce operations that they have performed (indexed by i as in Definition 3.43). Now, when a given partition forks (i.e., multiple runs within the partition have distinct shift/reduce actions on some lookahead), since the runs have made the same shift/reduce decisions thus far, there must be a single string $\tau \in (\Lambda \cup \Sigma)^*$ that takes the NFA starting state to a vertex with each of the distinct actions. By assumption, there is some corresponding fork point $v = (X \to \alpha \cdot Y\beta, -)$ such that for some vertices $w_1 = (Y \to \cdot\, \eta_1, -)$, $\ldots$, $w_d = (Y \to \cdot\, \eta_d, -)$, the respective stacks contain the vertices (v, w_j), sequentially, for each j.

Since no w_j can be popped from its respective stack until a string generated by η_j has subsequently been seen in the input (and, until such a point, the substring being parsed must be a prefix of one generated by η_j), and since v is join-safe (Definition 3.52), we conclude that if any partition j later forks on a vertex not reachable from v in the NFA, then no other partition $j' \neq j$ can be extant at that time. Hence, all nontrivial forks occur at successive points on paths in the

NFA. Since distinct partial parses require distinct shift/reduce decisions in the nondeterministic LR parsing algorithm (Lemma 3.49), the claim then follows by definition of XLR(k, t). □

Many other heuristics to prove the XLR criterion are possible; for instance, we may consider refining the join-safety criterion to specify that a vertex may be deemed safe if no conflicts are reachable once its parent vertices have been advanced by its predicted symbol. However, the above suffices to prove grammars XLR in a wide variety of practical cases. As an example, we consider the following simple (non-LR(1)) grammar:

$$S' \to S$$
$$S \to X\ c\ a \mid Y\ c\ b$$
$$X \to c$$
$$Y \to c$$

There is a single conflict in the LR(1) DFA, namely, at the vertex $\{(X \to c\,\cdot, \{c\}), (Y \to c\,\cdot, \{c\})\}$ on lookahead c. However, this conflict has the fork point $(S' \to \cdot\, S, \{\dashv\})$, which has ε-edges to the two vertices $(S \to \cdot\, X\ c\ a, \{\dashv\})$ and $(S \to \cdot\, Y\ c\ b, \{\dashv\})$. Since clearly no instance of $X\ c\ a$ can occur as a prefix of an instance of $Y\ c\ b$, and vice versa, the fork point is join-safe, and we conclude that the grammar is XLR$(1, 2)$, as expected.

4 Benchmarks

To test the generality of the new paradigms proposed in this work, we investigate the possibility of using them to parse full industrial programming languages. Some care needs to be taken in selecting the languages to prototype. For instance, it is known that parsing C++ is undecidable—the classic example being the simple pointer/operator ambiguity:

```
x * y;
```

Whether this is parsed as a declaration ("`y` is a pointer of type `x*`") or a multiplication expression ("`x` times `y`") depends on whether `x` names a type or a value, but this may depend on arbitrary template computation, which is known to be Turing-complete. More generally, a language construct need not be literally undecidable in order to be a poor example; it suffices that it be truly confusing for a human observer.

With this in mind, we have selected two well-specified industrial languages: Python 3.9.12 and Golang 1.17.8. We have expressed each language's syntax in a form amenable to our LR software implementation, and have endeavored the match the language specifications as closely as possible, with the following caveats:

- Both languages' specifications include some provision to limit the set of valid Unicode character encodings (e.g., to restrict characters used in identifiers to code points designated as "letters"). Enumerating these encodings is quite laborious and depends on the details of the language's compiler implementation, and so we have opted not to enforce these restrictions, instead allowing arbitrary Unicode code points (except those that are otherwise designated as distinguished by the language). This does not interfere with the parsing task, and should it be desired to enforce a particular set of restrictions, this can easily be done as a postprocessing pass on the abstract syntax tree.

- In the case of Python specifically, there are several properties which are designated as "syntax errors" by the compiler, but which often involve some semantic complexity (e.g., exception clauses must have either at least one "`except`" or at least one "`finally`", and the default "`except`" must appear in last position). While we believe it would be possible to implement such restrictions using our attribute mechanism (Section 3.5), this would greatly complicate the grammar. Moreover, we believe that LR attributes should be used only when absolutely necessary to resolve ambiguities or conflicts—if parsing is possible without encoding semantic properties as attributes, this should be preferred, as the desired semantic properties can be easily enforced via postprocessing on the abstract syntax tree. Thus, we have opted not to enforce this type of restriction at the grammar level, making our Python parser accept a language that is slightly more general than the official specification.

Subject to these caveats, we have successfully generated parsers for both of the above industrial languages, and tested it on the standard language implementation codebases (i.e., the directories "`cpython/Lib`" and "`go/src`", respectively, from the official language repositories, exempting "bad" test examples which the standard compilers also fail to parse). Tests are performed on an AMD Ryzen Threadripper PRO 3995WX CPU with 256 GiB of main memory, running Ubuntu Linux 22.04. The results are as follows:

- For Golang 1.17.8, our grammar specification requires 551 lines of code, and parser generation runs in 61 sec, producing 69 KLOC of generated code. On the standard codebase described above (parsing each file 10 times), the standard parser requires 18.28 sec, and our generated parser requires 15.24 sec (1.2x faster).
- For Python 3.9.12, our grammar specification requires 398 lines of code, and parser generation runs in 48 sec, producing 70 KLOC of generated code. On the standard codebase described above (parsing each file 10 times), the standard parser requires 33.64 sec, and our generated parser requires 7.78 sec (4.3x faster).

We remark that our generated parsing code is heavily optimized (and includes, among various other enhancements, a pool-based memory allocator); however, the parser generation code itself is not significantly optimized, and there is plenty of room should its running time need to be improved. The above benchmarks serve as a proof-of-concept that, using the techniques described in this work, automatic LR parser generation is a practical option for industrial programming language grammars.

Addendum. In the interim since this work was undertaken, both Golang and Python have introduced modifications to their respective languages that make them much more difficult to parse:

- Golang 1.18 introduced generic types, supporting declarations such as the following:

  ```
  type x [T interface{}] struct { y T }
  ```

  Generics are a valuable language feature at the semantic level. Unfortunately, the syntactic implementation creates new opportunities for confusing parses such as the following:

  ```
  type x [a *b] c
  ```

  similar to the classic C++ ambiguity, where it is unclear if we are parsing a generic declaration with type parameter `a` of type `*b`, or a non-generic declaration of an array of type `[a*b]c`, i.e., a length of `a*b` elements each of type `c`. The Golang specification indicates that this

ambiguity is to be resolved by preferring to parse as an array type whenever the tokens enclosed by the brackets could be parsed as a valid expression. Unfortunately, such a rule is not compositional: in order to write a BNF-style specification, we would need to include a criterion such as "a type parameter consists of an input sequence which is *not* parseable as an expression". Moreover, even this criterion is not known until the end of the bracketed expression, which may require the parser to make decisions with an unbounded amount of lookahead.

- Python 3.10, via PEP 634, introduced structural pattern-matching, supporting statements such as the following:

```
match x:
    case x0,x1:
        y = x0
    case x0,x1,x2:
        y = x2
```

Though seemingly innocuous from a parsing perspective, the specification of PEP 634 also requires that `match` become a "soft keyword", i.e., that it can still be used as an identifier in addition to being used as a keyword. This creates the potential for confusing partial parses such as the following:

```
match(x, y, *z, *w...
```

where it is unclear if we are matching on a tuple which has inline list expansions `z`,`w`, or alternatively if we are calling the identifier "`match`" with arguments `x`, `y`, `*z` and `*w`. If, for instance, we had written `**w` in place of `*w`, the compiler would be required to flag this as an error if we are parsing a pattern-match, but not if we are parsing a function call—but we will not know which is the case until we reach the end of the line, which could be an unbounded number of tokens away.

While we believe that some of these language modifications could be handled in practice via the XLR parsing algorithm (Section 3.7), we believe it is also the case that these new features result in partial parses which are truly confusing for a human reader, and hence fall outside the scope of the present work.

5 Conclusion

Parsing is a laborious and error-prone component of compiler construction, and despite known theoretical approaches to automatic parser generation, most industrial parsers are still written by hand. In this work, we have demonstrated that automatic parser generation can in fact be practical, by introducing a number of new developments upon the standard LR parsing paradigm of Knuth et al. With our new methodology, we can automatically generate efficient parsers for virtually all programming languages that are intuitively "easy to parse"—a claim we have supported experimentally, by implementing the new algorithms and running them on syntax specifications for Golang and Python 3. We believe that with the introduction of this and related work, parsing will no longer be a significant barrier to programming language development.

6 Acknowledgements

The author would like to thank George Kulakowski and Gideon Wald for many helpful discussions and comments on earlier drafts of this work.

References

[ABW15] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. If the current clique algorithms are optimal, so is Valiant's parser. In *FOCS*, 2015.

[AM17] Michael D. Adams and Matthew Might. Restricting grammars with tree automata. *Proc. ACM Program. Lang.*, 1(OOPSLA):82:1–82:25, 2017.

[Cor85] Robert Paul Corbett. *Static Semantics and Compiler Error Recovery*. PhD thesis, EECS Department, University of California, Berkeley, Jun 1985.

[DeR69] Franklin DeRemer. Practical translators for LR(k) languages. Technical report, 1969. `http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-065.pdf`.

[DM10] Joel E. Denny and Brian A. Malloy. The IELR(1) algorithm for generating minimal LR(1) parser tables for non-LR(1) grammars with conflict resolution. *Sci. Comput. Program.*, 75(11):943–979, 2010.

[Ear70] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970.

[Ewi97] Gregory C. Ewing. Yacc++: an object-oriented parser generator, 1997. `https://www.cosc.canterbury.ac.nz/greg.ewing/Unix_Projects.html`.

[For02] Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *ICFP*, 2002.

[For04] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *POPL*, 2004.

[Joh79] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, Bell Laboratories, 1979.

[Keg19] Jeffrey Kegler. Marpa, a practical general parser: the recognizer. *arXiv*, 2019.

[Knu65] Donald E. Knuth. On the translation of languages from left to right. *Inf. Control.*, 8(6):607–639, 1965.

[Leo91] Joop M. I. M. Leo. A general context-free parsing algorithm running in linear time on every LR(k) grammar without using lookahead. *Theor. Comput. Sci.*, 82(1):165–176, 1991.

[LM01] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, 2001. User Modeling 2007, 11th International Conference, UM 2007, Corfu, Greece, June 25-29, 2007.

[MN04] Scott McPeak and George C. Necula. Elkhound: A fast, practical GLR parser generator. In *Compiler Construction, 13th International Conference*, 2004.

[Pag77] David Pager. A practical general method for constructing LR(k) parsers. *Acta Informatica*, 7:249–268, 1977.

[PQ95] Terence John Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Softw. Pract. Exp.*, 25(7):789–810, 1995.

[Sch12] Walter Schulze. Gocc, 2012. `https://github.com/goccmack/gocc`.

[Tom84] Masaru Tomita. LR parsers for natural languages. In *COLING*, 1984.

[Val75] Leslie G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–315, 1975.

[Zim22] Joe Zimmerman. langcc: A next-generation compiler compiler. *arXiv*, 2022.