Adaptive *LL(*)* Parsing: The Power of Dynamic Analysis

Terence Parr
University of San Francisco
parrt@cs.usfca.edu

Sam Harwell
University of Texas at Austin
samharwell@utexas.edu

Kathleen Fisher
Tufts University
kfisher@eecs.tufts.edu

Abstract

Despite the advances made by modern parsing strategies such as PEG, *LL(*)*, GLR, and GLL, parsing is not a solved problem. Existing approaches suffer from a number of weaknesses, including difficulties supporting side-effecting embedded actions, slow and/or unpredictable performance, and counter-intuitive matching strategies. This paper introduces the *ALL(*)* parsing strategy that combines the simplicity, efficiency, and predictability of conventional top-down *LL(k)* parsers with the power of a GLR-like mechanism to make parsing decisions. The critical innovation is to move grammar analysis to parse-time, which lets *ALL(*)* handle any non-left-recursive context-free grammar. *ALL(*)* is $O(n^4)$ in theory but consistently performs linearly on grammars used in practice, outperforming general strategies such as GLL and GLR by orders of magnitude. ANTLR 4 generates *ALL(*)* parsers and supports direct left-recursion through grammar rewriting. Widespread ANTLR 4 use (5000 downloads/month in 2013) provides evidence that *ALL(*)* is effective for a wide variety of applications.

1. Introduction

Computer language parsing is still not a solved problem in practice, despite the sophistication of modern parsing strategies and long history of academic study. When machine resources were scarce, it made sense to force programmers to contort their grammars to fit the constraints of deterministic $LALR(k)$ or $LL(k)$ parser generators.[1] As machine resources grew, researchers developed more powerful, but more costly, nondeterministic parsing strategies following both "bottom-up" (LR-style) and "top-down" (LL-style) approaches. Strategies include GLR [26], Parser Expression Grammar (PEG) [9], *LL(*)* [20] from ANTLR 3, and recently, GLL [25], a fully general top-down strategy.

Although these newer strategies are much easier to use than $LALR(k)$ and $LL(k)$ parser generators, they suffer from a variety of weaknesses. First, nondeterministic parsers sometimes have unanticipated behavior. GLL and GLR return multiple parse trees (forests) for ambiguous grammars because they were designed to handle natural language grammars, which are often intentionally ambiguous. For computer languages, ambiguity is almost always an error. One can certainly walk the constructed parse forest to disambiguate it, but that approach costs extra time, space, and machinery for the uncommon case.

[1] We use the term *deterministic* in the way that deterministic finite automata (*DFA*) differ from nondeterministic finite automata (*NFA*): The next symbol(s) uniquely determine action.

PEGs are unambiguous by definition but have a quirk where rule $A \rightarrow a \mid ab$ (meaning "A matches either a or ab") can never match ab since PEGs choose the first alternative that matches a prefix of the remaining input. Nested backtracking makes debugging PEGs difficult.

Second, side-effecting programmer-supplied actions (mutators) like print statements should be avoided in any strategy that continuously speculates (PEG) or supports multiple interpretations of the input (GLL and GLR) because such actions may never really take place [17]. (Though DParser [24] supports "final" actions when the programmer is certain a reduction is part of an unambiguous final parse.) Without side effects, actions must buffer data for all interpretations in immutable data structures or provide undo actions. The former mechanism is limited by memory size and the latter is not always easy or possible. The typical approach to avoiding mutators is to construct a parse tree for post-parse processing, but such artifacts fundamentally limit parsing to input files whose trees fit in memory. Parsers that build parse trees cannot analyze large data files or infinite streams, such as network traffic, unless they can be processed in logical chunks.

Third, our experiments (Section 7) show that GLL and GLR can be slow and unpredictable in time and space. Their complexities are, respectively, $O(n^3)$ and $O(n^{p+1})$ where p is the length of the longest production in the grammar [14]. (GLR is typically quoted as $O(n^3)$ because Kipps [15] gave such an algorithm, albeit with a constant so high as to be impractical.) In theory, general parsers should handle deterministic grammars in near-linear time. In practice, we found GLL and GLR to be ˜135x slower than *ALL(*)* on a corpus of 12,920 Java 6 library source files (123M) and 6 orders of magnitude slower on a single 3.2M Java file, respectively.

LL(*) addresses these weaknesses by providing a mostly deterministic parsing strategy that uses regular expressions, represented as *deterministic finite automata* (DFA), to potentially examine the entire remaining input rather than the fixed k-sequences of *LL(k)*. Using DFA for lookahead limits *LL(*)* decisions to distinguishing alternatives with regular lookahead languages, even though lookahead languages (set of all possible remaining input phrases) are often context-free. But the main problem is that the *LL(*)* grammar condition is statically undecidable and grammar analysis sometimes fails to find regular expressions that distinguish between alternative productions. ANTLR 3's static analysis detects and avoids potentially-undecidable situations, failing over to backtracking parsing decisions instead. This gives *LL(*)* the same $a \mid ab$ quirk as PEGs for such decisions. Backtracking decisions that choose the first matching alternative also cannot detect obvious ambiguities such $A \rightarrow \alpha \mid \alpha$ where α is some sequence of grammar symbols that makes $\alpha \mid \alpha$ non-*LL(*)*.

1.1 Dynamic grammar analysis

In this paper, we introduce Adaptive *LL(*)* , or *ALL(*)* , parsers that combine the simplicity of deterministic top-down parsers with the power of a GLR-like mechanism to make parsing decisions. Specifically, LL parsing suspends at each prediction decision point (nonterminal) and then resumes once the prediction mechanism has chosen the appropriate production to expand. The critical innovation is to move grammar analysis to parse-time; no static grammar analysis is needed. This choice lets us avoid the undecidability of static *LL(*)* grammar analysis and lets us generate correct parsers (Theorem 6.1) for any non-left-recursive context-free grammar (CFG). While static analysis must consider all possible input sequences, dynamic analysis need only consider the finite collection of input sequences actually seen.

The idea behind the *ALL(*)* prediction mechanism is to launch subparsers at a decision point, one per alternative production. The subparsers operate in pseudo-parallel to explore all possible paths. Subparsers die off as their paths fail to match the remaining input. The subparsers advance through the input in lockstep so analysis can identify a sole survivor at the minimum lookahead depth that uniquely predicts a production. If multiple subparsers coalesce together or reach the end of file, the predictor announces an ambiguity and resolves it in favor of the lowest production number associated with a surviving subparser. (Productions are numbered to express precedence as an automatic means of resolving ambiguities like PEGs; Bison also resolves conflicts by choosing the production specified first.) Programmers can also embed *semantic predicates* [22] to choose between ambiguous interpretations.

ALL(*) parsers memoize analysis results, incrementally and dynamically building up a cache of DFA that map lookahead phrases to predicted productions. (We use the term *analysis* in the sense that *ALL(*)* analysis yields lookahead DFA like static *LL(*)* analysis.) The parser can make future predictions at the same parser decision and lookahead phrase quickly by consulting the cache. Unfamiliar input phrases trigger the grammar analysis mechanism, simultaneously predicting an alternative and updating the DFA. DFA are suitable for recording prediction results, despite the fact that the lookahead language at a given decision typically forms a context-free language. Dynamic analysis only needs to consider the finite context-free language subsets encountered during a parse and any finite set is regular.

To avoid the exponential nature of nondeterministic subparsers, prediction uses a *graph-structured stack* (GSS) [25] to avoid redundant computations. GLR uses essentially the same strategy except that *ALL(*)* only predicts productions with such subparsers whereas GLR actually parses with them. Consequently, GLR must push terminals onto the GSS but *ALL(*)* does not.

ALL(*) parsers handle the task of matching terminals and expanding nonterminals with the simplicity of LL but have $O(n^4)$ theoretical time complexity (Theorem 6.3) because in the worst-case, the parser must make a prediction at each input symbol and each prediction must examine the entire remaining input; examining an input symbol can cost $O(n^2)$. $O(n^4)$ is in line with the complexity of GLR. In Section 7, we show empirically that *ALL(*)* parsers for common languages are efficient and exhibit linear behavior in practice.

The advantages of *ALL(*)* stem from moving grammar analysis to parse time, but this choice places an additional burden on grammar functional testing. As with all dynamic approaches, programmers must cover as many grammar position and input sequence combinations as possible to find grammar ambiguities. Standard source code coverage tools can help programmers measure grammar coverage for *ALL(*)* parsers. High coverage in the generated code corresponds to high grammar coverage.

The *ALL(*)* algorithm is the foundation of the ANTLR 4 parser generator (ANTLR 3 is based upon *LL(*)*). ANTLR 4 was released in January 2013 and gets about 5000 downloads/month (source, binary, or ANTLRworks2 development environment, counting non-robot entries in web logs with unique IP addresses to get a lower bound.) Such activity provides evidence that *ALL(*)* is useful and usable.

The remainder of this paper is organized as follows. We begin by introducing the ANTLR 4 parser generator (Section 2) and discussing the *ALL(*)* parsing strategy (Section 3). Next, we define *predicated grammars*, their *augmented transition network* representation, and lookahead DFA (Section 4). Then, we describe *ALL(*)* grammar analysis and present the parsing algorithm itself (Section 5). Finally, we support our claims regarding *ALL(*)* correctness (Section 6) and efficiency (Section 7) and examine related work (Section 8). Appendix A has proofs for key *ALL(*)* theorems, Appendix B discusses algorithm pragmatics, Appendix C has left-recursion elimination details.

2. ANTLR 4

ANTLR 4 accepts as input any context-free grammar that does not contain indirect or hidden left-recursion.[2] From the grammar, ANTLR 4 generates a recursive-descent parser that uses an *ALL(*)* production prediction function (Section 3). ANTLR currently generates parsers in Java or C#. ANTLR 4 grammars use *yacc*-like syntax with extended BNF (EBNF) operators such as Kleene star (`*`) and token literals in single quotes. Grammars contain both lexical and syntactic rules in a combined specification for convenience. ANTLR 4 generates both a lexer and a parser from the combined specification. By using individual characters as input symbols, ANTLR 4 grammars can be scannerless and composable because *ALL(*)* languages are closed under union (Theorem 6.2), providing the benefits of

[2] Indirectly left-recursive rules call themselves through another rule; e.g., $A \rightarrow B$, $B \rightarrow A$. Hidden left-recursion occurs when an empty production exposes left recursion; e.g., $A \rightarrow BA$, $B \rightarrow \epsilon$.

modularity described by Grimm [10]. (We will henceforth refer to ANTLR 4 as ANTLR and explicitly mark earlier versions.)

Programmers can embed side-effecting actions (*mutators*), written in the host language of the parser, in the grammar. The actions have access to the current state of the parser. The parser ignores mutators during speculation to prevent actions from "launching missiles" speculatively. Actions typically extract information from the input stream and create data structures.

ANTLR also supports *semantic predicates*, which are side-effect free Boolean-valued expressions written in the host language that determine the semantic viability of a particular production. Semantic predicates that evaluate to false during the parse render the surrounding production nonviable, dynamically altering the language generated by the grammar at parse-time.[3] Predicates significantly increase the strength of a parsing strategy because predicates can examine the parse stack and surrounding input context to provide an informal context-sensitive parsing capability. Semantic actions and predicates typically work together to alter the parse based upon previously-discovered information. For example, a C grammar could have embedded actions to define type symbols from constructs, like `typedef int i32;`, and predicates to distinguish type names from other identifiers in subsequent definitions like `i32 x;`.

2.1 Sample grammar

Figure 1 illustrates ANTLRs yacc-like metalanguage by giving the grammar for a simple programming language with assignment and expression statements terminated by semicolons. There are two grammar features that render this grammar non-*LL(*)* and, hence, unacceptable to ANTLR 3. First, rule `expr` is left recursive. ANTLR 4 automatically rewrites the rule to be non-left-recursive and unambiguous, as described in Section 2.2. Second, the alternative productions of rule `stat` have a common recursive prefix (`expr`), which is sufficient to render `stat` undecidable from an *LL(*)* perspective. ANTLR 3 would detect recursion on production left edges and fail over to a backtracking decision at runtime.

Predicate `{!enum_is_keyword}?` in rule `id` allows or disallows `enum` as a valid identifier according to the predicate at the moment of prediction. When the predicate is false, the parser treats `id` as just `id : ID ;` disallowing `enum` as an `id` as the lexer matches `enum` as a separate token from `ID`. This example demonstrates how predicates allow a single grammar to describe subsets or variations of the same language.

2.2 Left-recursion removal

The *ALL(*)* parsing strategy itself does not support left-recursion, but ANTLR supports direct left-recursion through grammar rewriting prior to parser generation. Direct left-recursion covers the most common cases, such as arithmetic expression productions, like $E \rightarrow E\,.\,\mathbf{id}$, and C declarators. We made an engineering decision not to support indirect or hidden left-recursion

[3] Previous versions of ANTLR supported *syntactic predicates* to disambiguate cases where static grammar analysis failed; this facility is not needed in ANTLR4 because of *ALL(*)*'s dynamic analysis.

```
grammar Ex; // generates class ExParser
// action defines ExParser member: enum_is_keyword
@members {boolean enum_is_keyword = true;}
stat: expr '=' expr ';' // production 1
    | expr ';'          // production 2
    ;
expr: expr '*' expr
    | expr '+' expr
    | expr '(' expr ')' // f(x)
    | id
    ;
id  : ID | {!enum_is_keyword}? 'enum' ;
ID  : [A-Za-z]+ ; // match id with upper, lowercase
WS  : [ \t\r\n]+ -> skip ; // ignore whitespace
```

Figure 1. Sample left-recursive ANTLR 4 predicated-grammar `Ex`

because these forms are much less common and removing all left recursion can lead to exponentially-big transformed grammars. For example, the C11 language specification grammar contains lots of direct left-recursion but no indirect or hidden recursion. See Appendix 2.2 for more details.

2.3 Lexical analysis with *ALL(*)*

ANTLR uses a variation of *ALL(*)* for lexing that fully matches tokens instead of just predicting productions like *ALL(*)* parsers do. After warm-up, the lexer will have built a DFA similar to what regular-expression based tools such as lex would create statically. The key difference is that *ALL(*)* lexers are predicated context-free grammars not just regular expressions so they can recognize context-free tokens such as nested comments and can gate tokens in and out according to semantic context. This design is possible because *ALL(*)* is fast enough to handle lexing as well as parsing.

ALL(*) is also suitable for scannerless parsing because of its recognition power, which comes in handy for context-sensitive lexical problems like merging C and SQL languages. Such a union has no clear lexical sentinels demarcating lexical regions:

```
int next = select ID from users where name='Raj'+1;
int from = 1, select = 2;
int x = select * from;
```

See grammar `code/extras/CSQL` in [19] for a proof of concept.

3. Introduction to *ALL(*)* parsing

In this section, we explain the ideas and intuitions behind *ALL(*)* parsing. Section 5 will then present the algorithm more formally. The strength of a top-down parsing strategy is related to how the strategy chooses which alternative production to expand for the current nonterminal. Unlike *LL(k)* and *LL(*)* parsers, *ALL(*)* parsers always choose the first alternative that leads to a valid parse. All non-left-recursive grammars are therefore *ALL(*)*.

Instead of relying on static grammar analysis, an *ALL(*)* parser adapts to the input sentences presented to it at parse-time. The parser analyzes the current decision point (nonterminal with multiple productions) using a GLR-like mechanism to explore all possible decision paths with respect to the current "call" stack of in-process nonterminals and the remaining

```
void stat() { // parse according to rule stat
    switch (adaptivePredict("stat", call stack)) {
      case 1 : // predict production 1
        expr(); match('='); expr(); match(';');
        break;
      case 2 : // predict production 2
        expr(); match(';'); break;
    }
}
```

Figure 2. Recursive-descent code for `stat` in grammar `Ex`

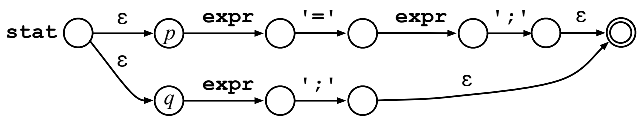


Figure 3. ATN for ANTLR rule `stat` in grammar `Ex`

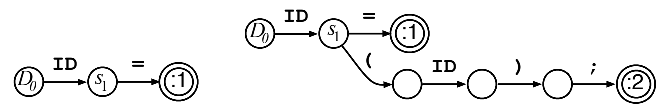
input *on-demand*. The parser incrementally and dynamically builds a lookahead DFA per decision that records a mapping from lookahead sequence to predicted production number. If the DFA constructed to date matches the current lookahead, the parser can skip analysis and immediately expand the predicted alternative. Experiments in Section 7 show that *ALL(*)* parsers usually get DFA cache hits and that DFA are critical to performance.

Because *ALL(*)* differs from deterministic top-down methods only in the prediction mechanism, we can construct conventional recursive-descent *LL* parsers but with an important twist. *ALL(*)* parsers call a special prediction function, *adaptivePredict*, that analyzes the grammar to construct lookahead DFA instead of simply comparing the lookahead to a statically-computed token set. Function *adaptivePredict* takes a nonterminal and parser call stack as parameters and returns the predicted production number or throws an exception if there is no viable production. For example, rule `stat` from the example in Section 2.1 yields a parsing procedure similar to Figure 2.

ALL(*) prediction has a structure similar to the well-known NFA-to-DFA *subset construction* algorithm. The goal is to discover the set of states the parser could reach after having seen some or all of the remaining input relative to the current decision. As in subset construction, an *ALL(*)* DFA state is the set of parser configurations possible after matching the input leading to that state. Instead of an NFA, however, *ALL(*)* simulates the actions of an *augmented recursive transition network* (ATN) [27] representation of the grammar since ATNs closely mirror grammar structure. (ATNs look just like syntax diagrams that can have actions and semantic predicates.) *LL(*)*'s static analysis also operates on an ATN for the same reason. Figure 3 shows the ATN submachine for rule `stat`.

An *ATN configuration* represents the execution state of a subparser and tracks the ATN state, predicted production number, and ATN subparser call stack: tuple (p, i, γ).[4] Configurations include production numbers so prediction can identify *which* production matches the current lookahead. Unlike static *LL(*)* analysis, *ALL(*)* incrementally builds DFA considering just the lookahead sequences it has seen instead of all possible sequences.

[4] Component i does not exist in the machine configurations of GLL, GLR, or Earley [8].



(a) After `x=y;` (b) After `x=y;` and `f(x);`

Figure 4. Prediction DFA for decision `stat`

When parsing reaches a decision for the first time, *adaptivePredict* initializes the lookahead DFA for that decision by creating a DFA start state, D_0. D_0 is the set of ATN subparser configurations reachable without consuming an input symbol starting at each production left edge. For example, construction of D_0 for nonterminal `stat` in Figure 3 would first add ATN configurations $(p, 1, [])$ and $(q, 2, [])$ where p and q are ATN states corresponding to production 1 and 2's left edges and $[]$ is the empty subparser call stack (if `stat` is the start symbol).

Analysis next computes a new DFA state indicating where ATN simulation could reach after consuming the first lookahead symbol and then connects the two DFA states with an edge labeled with that symbol. Analysis continues, adding new DFA states, until all ATN configurations in a newly-created DFA state predict the same production: $(-, i, -)$. Function *adaptivePredict* marks that state as an accept state and returns to the parser with that production number. Figure 4a shows the lookahead DFA for decision `stat` after *adaptivePredict* has analyzed input sentence `x=y;`. The DFA does not look beyond `=` because `=` is sufficient to uniquely distinguish `expr`'s productions. (Notation `:1` means "predict production 1.")

In the typical case, *adaptivePredict* finds an existing DFA for a particular decision. The goal is to find or build a path through the DFA to an accept state. If *adaptivePredict* reaches a (non-accept) DFA state without an edge for the current lookahead symbol, it reverts to ATN simulation to extend the DFA (without rewinding the input). For example, to analyze a second input phrase for `stat`, such as `f(x);`, *adaptivePredict* finds an existing `ID` edge from the D_0 and jumps to s_1 without ATN simulation. There is no existing edge from s_1 for the left parenthesis so analysis simulates the ATN to complete a path to an accept state, which predicts the second production, as shown in Figure 4b. Note that because sequence `ID(ID)` predicts both productions, analysis continues until the DFA has edges for the `=` and `;` symbols.

If ATN simulation computes a new target state that already exists in the DFA, simulation adds a new edge targeting the existing state and switches back to DFA simulation mode starting at that state. Targeting existing states is how cycles can appear in the DFA. Extending the DFA to handle unfamiliar phrases empirically decreases the likelihood of future ATN simulation, thereby increasing parsing speed (Section 7).

3.1 Predictions sensitive to the call stack

Parsers cannot always rely upon lookahead DFA to make correct decisions. To handle all non-left-recursive grammars, *ALL(*)* prediction must occasionally consider the parser call stack available at the start of prediction (denoted γ_0 in Section 5). To illustrate the need for stack-sensitive predictions, consider that predictions made while recognizing a Java method definition might depend on whether the method was defined within an interface or class definition. (Java interface methods cannot have bodies.) Here is a simplified grammar that exhibits a stack-sensitive decision in nonterminal A:

$$S \to xB \mid yC \quad B \to Aa \quad C \to Aba \quad A \to b \mid \epsilon$$

Without the parser stack, no amount of lookahead can uniquely distinguish between A's productions. Lookahead ba predicts $A \to b$ when B invokes A but predicts $A \to \epsilon$ when C invokes A. If prediction ignores the parser call stack, there is a prediction conflict upon ba.

Parsers that ignore the parser call stack for prediction are called *Strong LL* (*SLL*) parsers. The recursive-descent parsers programmers build by hand are in the *SLL* class. By convention, the literature refers to *SLL* as *LL* but we distinguish the terms since "real" *LL* is needed to handle all grammars. The above grammar is $LL(2)$ but not $SLL(k)$ for any k, though duplicating A for each call site renders the grammar $SLL(2)$.

Creating a different lookahead DFA for each possible parser call stack is not feasible since the number of stack permutations is exponential in the stack depth. Instead, we take advantage of the fact that most decisions are not stack-sensitive and build lookahead DFA ignoring the parser call stack. If *SLL* ATN simulation finds a prediction conflict (Section 5.3), it cannot be sure if the lookahead phrase is ambiguous or stack-sensitive. In this case, *adaptivePredict* must re-examine the lookahead using the parser stack γ_0. This hybrid or *optimized* LL *mode* improves performance by caching stack-insensitive prediction results in lookahead DFA when possible while retaining full stack-sensitive prediction power. Optimized *LL* mode applies on a per-decision basis, but *two-stage parsing*, described next, can often avoid *LL* simulation completely. (We henceforth use *SLL* to indicate stack-insensitive parsing and *LL* to indicate stack-sensitive.)

3.2 Two-stage *ALL(*)* parsing

SLL is weaker but faster than *LL*. Since we have found that most decisions are *SLL* in practice, it makes sense to attempt parsing entire inputs in "*SLL* only mode," which is stage one of the two-stage *ALL(*)* parsing algorithm. If, however, *SLL* mode finds a syntax error, it might have found an *SLL* weakness or a real syntax error, so we have to retry the entire input using optimized *LL* mode, which is stage two. This counter-intuitive strategy, which potentially parses entire inputs twice, can dramatically increase speed over the single-stage optimized *LL* mode stage. For example, two-stage parsing with the Java grammar (Section 7) is 8x faster than one-stage optimized *LL* mode to parse a 123M corpus. The two-stage strategy relies on the fact that *SLL* either behaves like *LL* or gets a syntax error (Theorem 6.5). For invalid sentences, there is no derivation for the input regardless of how the parser chooses productions. For valid sentences, *SLL* chooses productions as *LL* would or picks a production that ultimately leads to a syntax error (*LL* finds that choice nonviable). Even in the presence of ambiguities, *SLL* often resolves conflicts as *LL* would. For example, despite a few ambiguities in our Java grammar, *SLL* mode correctly parses all inputs we have tried without failing over to *LL*. Nonetheless, the second (*LL*) stage must remain to ensure correctness.

4. Predicated grammars, ATNs, and DFA

To formalize *ALL(*)* parsing, we first need to recall some background material, specifically, the formal definitions of predicate grammars, ATNs, and Lookahead DFA.

4.1 Predicated grammars

To formalize *ALL(*)* parsing, we first need to formally define the predicated grammars from which they are derived. A predicated grammar $G = (N, T, P, S, \Pi, \mathcal{M})$ has elements:

- N is the set of nonterminals (rule names)
- T is the set of terminals (tokens)
- P is the set of productions
- $S \in N$ is the start symbol
- Π is a set of side-effect-free semantic predicates
- $\mathcal{M}$ is a set of actions (mutators)

Predicated *ALL(*)* grammars differ from those of *LL(*)* [20] only in that *ALL(*)* grammars do not need or support syntactic predicates. Predicated grammars in the formal sections of this paper use the notation shown in Figure 5. The derivation rules in Figure 6 define the meaning of a predicated grammar. To support semantic predicates and mutators, the rules reference state $\mathbb{S}$, which abstracts user state during parsing. The judgment form $(\mathbb{S}, \alpha) \Rightarrow (\mathbb{S}', \beta)$ may be read: "In machine state $\mathbb{S}$, grammar sequence α reduces in one step to modified state $\mathbb{S}'$ and grammar sequence β." The judgment $(\mathbb{S}, \alpha) \Rightarrow^* (\mathbb{S}', \beta)$ denotes repeated applications of the one-step reduction rule. These reduction rules specify a leftmost derivation. A production with a semantic predicate π_i is viable only if π_i is true of the current state $\mathbb{S}$. Finally, an action production uses the specified mutator μ_i to update the state.

Formally, the language generated by grammar sequence α in user state $\mathbb{S}$ is $L(\mathbb{S}, \alpha) = \{w \mid (\mathbb{S}, \alpha) \Rightarrow^* (\mathbb{S}', w)\}$ and the language of grammar G is $L(\mathbb{S}_0, G) = \{w \mid (\mathbb{S}_0, S) \Rightarrow^* (\mathbb{S}, w)\}$ for initial user state $\mathbb{S}_0$ ($\mathbb{S}_0$ can be empty). If u is a prefix of w or equal to w, we write $u \preceq w$. Language L is *ALL(*)* *iff* there exists an *ALL(*)* grammar for L. Theoretically, the language class of $L(G)$ is recursively enumerable because each mutator could be a Turing machine. In reality, grammar writers do not use this generality so it is standard practice to consider the language class to be the context-sensitive languages instead. The class is context-sensitive rather than context-free as predicates can examine the call stack and terminals to the left and right.

This formalism has various syntactic restrictions not present in actual ANTLR grammars, for example, forcing mutators into their own rules and disallowing the common Extended BNF (EBNF) notation such as α^* and α^+ closures. We can make these restrictions without loss of generality because any grammar in the general form can be translated into this more restricted form.

4.2 Resolving ambiguity

An ambiguous grammar is one in which the same input sequence can be recognized in multiple ways. The rules in Fig-

$A \in N$	Nonterminal
$a, b, c, d \in T$	Terminal
$X \in (N \cup T)$	Production element
$\alpha, \beta, \delta \in X^*$	Sequence of grammar symbols
$u, v, w, x, y \in T^*$	Sequence of terminals
ϵ	Empty string
$\$$	End of file "symbol"
$\pi \in \Pi$	Predicate in host language
$\mu \in \mathcal{M}$	Action in host language
$\lambda \in (N \cup \Pi \cup \mathcal{M})$	Reduction label
$\vec{\lambda} = \lambda_1..\lambda_n$	Sequence of reduction labels
Production Rules:	
$A \to \alpha_i$	i^{th} context-free production of A
$A \to \{\pi_i\}?\ \alpha_i$	i^{th} production predicated on semantics
$A \to \{\mu_i\}$	i^{th} production with mutator

Figure 5. Predicated Grammar Notation

$$\textit{Prod}\ \frac{A \to \alpha}{(\mathbb{S}, uA\delta) \Rightarrow (\mathbb{S}, u\alpha\delta)}$$

$$\textit{Sem}\ \frac{\begin{matrix}\pi(\mathbb{S}) \\ A \to \{\pi\}?\,\alpha\end{matrix}}{(\mathbb{S}, uA\delta) \Rightarrow (\mathbb{S}, u\alpha\delta)} \qquad \textit{Action}\ \frac{A \to \{\mu\}}{(\mathbb{S}, uA\delta) \Rightarrow (\mu(\mathbb{S}), u\delta)}$$

$$\textit{Closure}\ \frac{(\mathbb{S}, \alpha) \Rightarrow (\mathbb{S}', \alpha'),\ (\mathbb{S}', \alpha') \Rightarrow^* (\mathbb{S}'', \beta)}{(\mathbb{S}, \alpha) \Rightarrow^* (\mathbb{S}'', \beta)}$$

Figure 6. Predicated Grammar Leftmost Derivation Rules

ure 6 do not preclude ambiguity. However, for a practical programing language parser, each input should correspond to a unique parse. To that end, *ALL(*)* uses the order of the productions in a rule to resolve ambiguities in favor of the production with the lowest number. This strategy is a concise way for programmers to resolve ambiguities automatically and resolves the well-known *if-then-else* ambiguity in the usual way by associating the *else* with the most recent *if*. PEGs and Bison parsers have the same resolution policy.

To resolve ambiguities that depend on the current state $\mathbb{S}$, programmers can insert semantic predicates but must make them mutually exclusive for all potentially ambiguous input sequences to render such productions unambiguous. Statically, mutual exclusion cannot be enforced because predicates are written in a Turing-complete language. At parse-time, however, *ALL(*)* evaluates predicates and dynamically reports input phrases for which multiple, predicated productions are viable. If the programmer fails to satisfy mutual exclusivity, *ALL(*)* uses production order to resolve the ambiguity.

4.3 Augmented transition networks

Given predicated grammar $G = (N, T, P, S, \Pi, \mathcal{M})$, the corresponding ATN $M_G = (Q, \Sigma, \Delta, E, F)$ has elements [20]:

- Q is the set of states
- Σ is the edge alphabet $N \cup T \cup \Pi \cup \mathcal{M}$
- Δ is the transition relation mapping $Q \times (\Sigma \cup \epsilon) \to Q$
- $E \in Q = \{p_A \mid A \in N\}$ is set of submachine entry states
- $F \in Q = \{p'_A \mid A \in N\}$ is set of submachine final states

ATNs resemble the syntax diagrams used to document programming languages, with an ATN submachine for each nonterminal. Figure 7 shows how to construct the set of states Q and edges Δ from grammar productions. The start state for A

Input Grammar Element	Resulting ATN Transitions
$A \to \alpha_i$	$p_A \xrightarrow{\epsilon} p_{A,i} \xrightarrow{\epsilon} \boxed{\alpha_i} \xrightarrow{\epsilon} p'_A$
$A \to \{\pi_i\}?\,\alpha_i$	$p_A \xrightarrow{\epsilon} p_{A,i} \xrightarrow{\pi_i} \boxed{\alpha_i} \xrightarrow{\epsilon} p'_A$
$A \to \{\mu_i\}$	$p_A \xrightarrow{\epsilon} p_{A,i} \xrightarrow{\mu_i} p'_A$
$A \to \epsilon_i$	$p_A \xrightarrow{\epsilon} p_{A,i} \xrightarrow{\epsilon} p'_A$
$\boxed{\alpha_i} = X_1X_2\ldots X_m$ for $X_j \in N \cup T,\ j = 1..m$	$p_0 \xrightarrow{X_1} p_1 \xrightarrow{X_2} \ldots \xrightarrow{X_m} p_m$

Figure 7. Predicated Grammar to ATN transformation

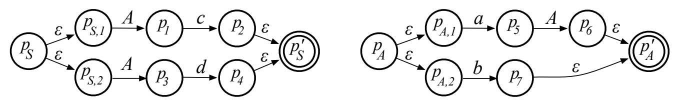


Figure 8. ATN for G with P=$\{S \to Ac \mid Ad,\ \ A \to aA \mid b\}$

is $p_A \in Q$ and targets $p_{A,i}$, created from the left edge of α_i, with an edge in Δ. The last state created from α_i targets p'_A. Nonterminal edges $p \xrightarrow{A} q$ are like function calls. They transfer control of the ATN to A's submachine, pushing return state q onto a state call stack so it can continue from q after reaching the stop state for A's submachine, p'_A. Figure 8 gives the ATN for a simple grammar. The language matched by the ATN is the same as the language of the original grammar.

4.4 Lookahead DFA

ALL(*) parsers record prediction results obtained from ATN simulation with *lookahead DFA*, which are DFA augmented with accept states that yield predicted production numbers. There is one accept state per production of a decision.

Definition 4.1. Lookahead DFA *are DFA with augmented accept states that yield predicted production numbers. For predicated grammar* $G = (N, T, P, S, \Pi, \mathcal{M})$*, DFA* $M = (Q, \Sigma, \Delta, D_0, F)$ *where:*

- Q is the set of states
- $\Sigma = T$ is the edge alphabet
- Δ is the transition function mapping $Q \times \Sigma \to Q$
- $D_0 \in Q$ is the start state
- $F \in Q = \{f_1, f_2, \ldots, f_n\}$ final states, one f_i per prod. i

A transition in Δ from state p to state q on symbol $a \in \Sigma$ has the form $p \xrightarrow{a} q$ and we require $p \xrightarrow{a} q'$ implies $q = q'$.

5. *ALL(*)* parsing algorithm

With the definitions of grammars, ATNs, and lookahead DFA formalized, we can present the key functions of the *ALL(*)* parsing algorithm. This section starts with a summary of the functions and how they fit together then discusses a critical graph data structure before presenting the functions themselves. We finish with an example of how the algorithm works.

Parsing begins with function *parse* that behaves like a conventional top-down *LL(k)* parse function except that *ALL(*)* parsers predict productions with a special function called *adaptivePredict*, instead of the usual "switch on next k token types" mechanism. Function *adaptivePredict* simulates an ATN representation of the original predicated grammar to choose an α_i production to expand for decision point $A \to \alpha_1 \mid ... \mid \alpha_n$.

Conceptually, prediction is a function of the current parser call stack γ_0, remaining input w_r, and user state $\mathbb{S}$ if A has predicates. For efficiency, prediction ignores γ_0 when possible (Section 3.1) and uses the minimum lookahead from w_r.

To avoid repeating ATN simulations for the same input and nonterminal, *adaptivePredict* assembles DFA that memoize input-to-predicted-production mappings, one DFA per nonterminal. Recall that each DFA state, D, is the set of ATN configurations possible after matching the lookahead symbols leading to that state. Function *adaptivePredict* calls *startState* to create initial DFA state, D_0, and then *SLLpredict* to begin simulation.

Function *SLLpredict* adds paths to the lookahead DFA that match some or all of w_r through repeated calls to *target*. Function *target* computes DFA target state D' from current state D using *move* and *closure* operations similar to those found in *subset construction*. Function *move* finds all ATN configurations reachable on the current input symbol and *closure* finds all configurations reachable without traversing a terminal edge. The primary difference from subset construction is that *closure* simulates the call and return of ATN submachines associated with nonterminals.

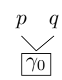
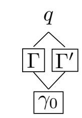
If *SLL* simulation finds a conflict (Section 5.3), *SLLpredict* rewinds the input and calls *LLpredict* to retry prediction, this time considering γ_0. Function *LLpredict* is similar to *SLLpredict* but does not update a nonterminal's DFA because DFA must be stack-insensitive to be applicable in all stack contexts. Conflicts within ATN configuration sets discovered by *LLpredict* represent ambiguities. Both prediction functions use *getConflictSetsPerLoc* to detect conflicts, which are configurations representing the same parser location but different productions. To avoid failing over to *LLpredict* unnecessarily, *SLLpredict* uses *getProdSetsPerState* to see if a potentially non-conflicting DFA path remains when *getConflictSetsPerLoc* reports a conflict. If so, it is worth continuing with *SLLpredict* on the chance that more lookahead will resolve the conflict without recourse to full *LL* parsing.

Before describing these functions in detail, we review a fundamental graph data structure that they use to efficiently manage multiple call stacks *a la* GLL and GLR.

5.1 Graph-structured call stacks

The simplest way to implement *ALL(*)* prediction would be a classic backtracking approach, launching a subparser for each α_i. The subparsers would consume all remaining input because backtracking subparsers do not know when to stop parsing—they are unaware of other subparsers' status. The independent subparsers would also lead to exponential time complexity. We address both issues by having the prediction subparsers advance in lockstep through w_r. Prediction terminates after consuming prefix $u \preceq w_r$ when all subparsers but one die off or when prediction identifies a conflict. Operating in lockstep also provides an opportunity for subparsers to share call stacks thus avoiding redundant computations.

Two subparsers at ATN state p that share the same ATN stack top, $q\gamma_1$ and $q\gamma_2$, will mirror each other's behavior until simulation pops q from their stacks. Prediction can treat those subparsers as a single subparser by merging stacks. We merge stacks for all configurations in a DFA state of the form (p, i, γ_1) and (p, i, γ_2), forming a general configuration (p, i, Γ) with *graph-structured stack* (GSS) [25] $\Gamma = \gamma_1 \uplus \gamma_2$ where $\uplus$ means graph merge. Γ can be the empty stack [], a special stack # used for *SLL* prediction (addressed shortly), an individual stack, or a graph of stack nodes. Merging individual stacks into a GSS reduces the potential size from exponential to linear complexity (Theorem 6.4). To represent a GSS, we use an immutable graph data structure with maximal sharing of nodes. Here are two examples that share the parser stack γ_0 at the bottom of the stack:

$p\gamma_0 \uplus q\gamma_0 =$  $\qquad q\Gamma\gamma_0 \uplus q\Gamma'\gamma_0 =$ 

In the functions that follow, all additions to configuration sets, such as with operator +=, implicitly merge stacks.

There is a special case related to the stack condition at the start of prediction. Γ must distinguish between an empty stack and no stack information. For *LL* prediction, the initial ATN simulation stack is the current parser call stack γ_0. The initial stack is only empty, $\gamma_0 = []$, when the decision entry rule is the start symbol. Stack-insensitive *SLL* prediction, on the other hand, ignores the parser call stack and uses an initial stack of #, indicating no stack information. This distinction is important when computing the *closure* (Function 7) of configurations representing submachine stop states. Without parser stack information, a subparser that returns from decision entry rule A must consider all possible invocation sites; i.e., *closure* sees configuration $(p'_A, -, \#)$.

The empty stack [] is treated like any other node for *LL* prediction: $\Gamma \uplus []$ yields the graph equivalent of set $\{\Gamma, []\}$, meaning that both Γ and the empty stack are possible. Pushing state p onto [] yields $p[]$ not p because popping p must leave the [] empty stack symbol. For *SLL* prediction, $\Gamma \uplus \# = \#$ for any graph Γ because # acts like a wildcard and represents the set of all stacks. The wildcard therefore contains any Γ. Pushing state p onto # yields $p\#$.

5.2 *ALL(*)* parsing functions

We can now present the key *ALL(*)* functions, which we have highlighted in boxes and interspersed within the text of this section. Our discussion follows a top-down order and assumes that the ATN corresponding to grammar G, the semantic state $\mathbb{S}$, the DFA under construction, and *input* are in scope for all functions of the algorithm and that semantic predicates and actions can directly access $\mathbb{S}$.

Function parse. The main entry point is function *parse* (shown in Function 1), which initiates parsing at the start symbol, argument S. The function begins by initializing a simulation call stack γ to an empty stack and setting ATN state "cursor" p to $p_{S,i}$, the ATN state on the left edge of S's production number i predicted by *adaptivePredict*. The function loops until the cursor reaches p'_S, the submachine stop state for S. If the cursor reaches another submachine stop state, p'_B, *parse* simulates a "return" by popping the return state q off the call stack and moving p to q.

Function 1: *parse*(S)
γ := []; i := *adaptivePredict*(S, γ); p := $p_{S,i}$;
while *true* **do**
  if $p = p'_B$ *(i.e., p is a rule stop state)* **then**
    if $B = S$ *(finished matching start rule S)* **then return**;
    else let $\gamma = q\gamma'$ **in** $\gamma := \gamma'$; $p := q$;
  else
    switch t *where* $p \xrightarrow{t} q$ **do**
      case b*: (i.e., terminal symbol transition)*
        if $b = input.curr()$ **then**
          $p := q$; $input.advance()$;
        else *parse error;*
      case B*:* $\gamma := q\gamma$; i := *adaptivePredict*(B, γ); p:=$p_{B,i}$;
      case μ*:* $\mathbb{S} := \mu(\mathbb{S})$; $p := q$;
      case π*:* **if** $\pi(\mathbb{S})$ **then** $p := q$ **else** *parse error*;
      case ϵ*:* $p := q$;
    endsw

For p not at a stop state, *parse* processes ATN transition $p \xrightarrow{t} q$. There can be only one transition from p because of the way ATNs are constructed. If t is a terminal edge and matches the current input symbol, *parse* transitions the edge and moves to the next symbol. If t is a nonterminal edge referencing some B, *parse* simulates a submachine call by pushing return state q onto the stack and choosing the appropriate production left edge in B by calling *adaptivePredict* and setting the cursor appropriately. For action edges, *parse* updates the state according to the mutator μ and transitions to q. For predicate edges, *parse* transitions only if predicate π evaluates to true. During the parse, failed predicates behave like mismatched tokens. Upon an ϵ edge, *parse* moves to q. Function *parse* does not explicitly check that parsing stops at end-of-file because applications like development environments need to parse input subphrases.

Function adaptivePredict. To predict a production, *parse* calls *adaptivePredict* (Function 2), which is a function of the decision nonterminal A and the current parser stack γ_0. Because prediction only evaluates predicates during full *LL* simulation, *adaptivePredict* delegates to *LLpredict* if at least one of the productions is predicated.[5] For decisions that do not yet have a DFA, *adaptivePredict* creates DFA dfa_A with start state D_0 in preparation for *SLLpredict* to add DFA paths. D_0 is the set of ATN configurations reachable without traversing a terminal edge. Function *adaptivePredict* also constructs the set of final states F_{DFA}, which contains one final state f_i for each production of A. The set of DFA states, Q_{DFA}, is the union of D_0, F_{DFA}, and the error state D_{error}. Vocabulary Σ_{DFA} is the set of grammar terminals T. For unpredicated decisions with existing DFA, adaptivePredict calls *SLLpredict* to obtain a prediction from the DFA, possibly extending the DFA through ATN simulation in the process. Finally, since *adaptivePredict* is looking ahead not parsing, it must undo any changes made to the input cursor, which it does by capturing the input index as $start$ upon entry and rewinding to $start$ before returning.

Function 2: *adaptivePredict*(A, γ_0) **returns** int alt
$start := input.index()$; // *checkpoint input*
if $\exists\, A \rightarrow \pi_i\alpha_i$ **then**
  $alt := LLpredict(A, start, \gamma_0)$;
  $input.seek(start)$; // *undo stream position changes*
  return *alt*;
if $\nexists$ *dfa*$_A$ **then**
  D_0 := *startState*(A, #);
  F_{DFA} := $\{f_i \mid f_i := \text{DFA\_State}(i)\,\forall\, A \rightarrow \alpha_i\}$;
  Q_{DFA} := $D_0 \cup F_{DFA} \cup D_{error}$;
  dfa$_A$:= *DFA*($Q_{DFA}, \Sigma_{DFA} = T, \Delta_{DFA} = \varnothing, D_0, F_{DFA}$);
alt := *SLLpredict*(A, D_0, $start$, γ_0);
$input.seek(start)$; // *undo stream position changes*
return alt;

Function startState. To create DFA start state D_0, *startState* (Function 3) adds configurations $(p_{A,i}, i, \gamma)$ for each $A \rightarrow \alpha_i$ and $A \rightarrow \pi_i\, \alpha_i$, if π_i evaluates to true. When called from *adaptivePredict*, call stack argument γ is special symbol # needed by *SLL* prediction, indicating "no parser stack information." When called from *LLpredict*, γ is initial parser stack γ_0. Computing *closure* of the configurations completes D_0.

Function 3: *startState*(A, γ) **returns** DFA_State D_0
$D_0 := \varnothing$;
foreach $p_A \xrightarrow{\epsilon} p_{A,i} \in \Delta_{ATN}$ **do**
  if $p_A \xrightarrow{\epsilon} p_{A,i} \xrightarrow{\pi_i} p$ **then** $\pi := \pi_i$ **else** $\pi := \epsilon$;
  if π=ϵ *or* $eval(\pi_i)$ **then** D_0+=$closure(\{\}, D_0, (p_{A,i}, i, \gamma))$;
return D_0;

Function SLLpredict. Function *SLLpredict* (Function 4) performs both DFA and *SLL* ATN simulation, incrementally adding paths to the DFA. In the best case, there is already a DFA path from D_0 to an accept state, f_i, for prefix $u \preceq w_r$ and some production number i. In the worst-case, ATN simulation is required for all a in sequence u. The main loop in *SLLpredict* finds an existing edge emanating from DFA state cursor D upon a or computes a new one via *target*. It is possible that *target* will compute a target state that already exists in the DFA, $\underline{D}'$, in which case function *target* returns $\underline{D}'$ because $\underline{D}'$ may already have outgoing edges computed; it is inefficient to discard work by replacing $\underline{D}'$. At the next iteration, *SLLpredict* will consider edges from $\underline{D}'$, effectively switching back to DFA simulation.

Function 4: *SLLpredict*(A, D_0, $start$, γ_0) **returns** int prod
$a := input.curr()$; $D = D_0$;
while *true* **do**
  let D' be DFA target $D \xrightarrow{a} D'$;
  if $\nexists\, D'$ **then** $D' := target(D, a)$;
  if $D' = D_{error}$ **then** *parse error;*
  if D' stack sensitive **then**
    $input.seek(start)$*;* **return** *LLpredict(A, start,* γ_0*);*
  if $D' = f_i \in F_{DFA}$ **then return** i*;*
  $D := D'$*;* $a := input.next()$*;*

[5] *SLL* prediction does not incorporate predicates for clarity in this exposition, but in practice, ANTLR incorporates predicates into DFA accept states (Section B.2). ANTLR 3 DFA used predicated edges not predicated accept states.

Once *SLLpredict* acquires a target state, D', it checks for errors, stack sensitivity, and completion. If *target* marked D' as stack-sensitive, prediction requires full *LL* simulation and *SLLpredict* calls *LLpredict*. If D' is accept state f_i, as determined by *target*, *SLLpredict* returns i. In this case, all the configurations in D' predicted the same production i; further analysis is unnecessary and the algorithm can stop. For any other D', the algorithm sets D to D', gets the next symbol, and repeats.

Function target. Using a combined *move-closure* operation, *target* discovers the set of ATN configurations reachable from D upon a single terminal symbol $a \in T$. Function *move* computes the configurations reachable directly upon a by traversing a terminal edge:

$$move(D, a) = \{(q, i, \Gamma) \mid p \xrightarrow{a} q,\ (p, i, \Gamma) \in D\}$$

Those configurations and their *closure* form D'. If D' is empty, no alternative is viable because none can match a from the current state so *target* returns error state D_{error}. If all configurations in D' predict the same production number i, *target* adds edge $D \xrightarrow{a} f_i$ and returns accept state f_i. If D' has conflicting configurations, *target* marks D' as stack-sensitive. The conflict could be an ambiguity or a weakness stemming from *SLL*'s lack of parser stack information. (Conflicts along with *getConflictSetsPerLoc* and *getProdSetsPerState* are described in Section 5.3.) The function finishes by adding state D', if an equivalent state, $\underline{D}'$, is not already in the DFA, and adding edge $D \xrightarrow{a} D'$.

Function 5: *target*(D, a) **returns** DFA_State D'
$mv := move(D, a)$;
$D' := \bigcup_{c \in mv} closure(\{\}, c)$;
if $D' = \varnothing$ **then** Δ_{DFA} += $D \xrightarrow{a} D_{error}$; **return** D_{error};
if $\{j \mid (-, j, -) \in D'\} = \{i\}$ **then**
  Δ_{DFA} += $D \xrightarrow{a} f_i$; **return** f_i; *// Predict rule* i
// Look for a conflict among configurations of D'
$a\_conflict$:= $\exists alts \in$ *getConflictSetsPerLoc*(D') : $|alts|>1$;
$viablealt$:= $\exists alts \in$ *getProdSetsPerState*(D') : $|alts| = 1$;
if *a_conflict and not viablealt* **then**
  mark D' as stack sensitive;
if $D' = \underline{D}' \in Q_{DFA}$ **then** D' := $\underline{D}'$; **else** Q_{DFA} += D';
Δ_{DFA} += $D \xrightarrow{a} D'$;
return D';

Function LLpredict. Upon *SLL* simulation conflict, *SLLpredict* rewinds the input and calls *LLpredict* (Function 6) to get a prediction based upon *LL* ATN simulation, which considers the full parser stack γ_0. Function *LLpredict* is similar to *SLLpredict*. It uses DFA state D as a cursor and state D' as the transition target for consistency but does not update A's DFA so *SLL* prediction can continue to use the DFA. *LL* prediction continues until either $D' = \varnothing$, D' uniquely predicts an alternative, or D' has a conflict. If D' from *LL* simulation has a conflict as *SLL* did, the algorithm reports the ambiguous phrase (input from $start$ to the current index) and resolves to the minimum production number among the conflicting configurations. (Section 5.3 explains ambiguity detection.) Otherwise, cursor D moves to D' and considers the next input symbol.

Function 6: *LLpredict*(A, $start$, γ_0) **returns** int alt
$D := D_0 :=$ *startState*(A, γ_0);
while *true* **do**
  $mv := move(D, input.curr())$;
  $D' := \bigcup_{c \in mv} closure(\{\}, c)$;
  if $D' = \varnothing$ **then** *parse error;*
  if $\{j \mid (-, j, -) \in D'\} = \{i\}$ **then return** i;
  / If all* p, Γ *pairs predict* > 1 *alt and all such production sets are same, input ambiguous. */*
  $altsets$:= *getConflictSetsPerLoc*(D');
  if $\forall\, x, y \in altsets, x = y$ *and* $|x| > 1$ **then**
    x := *any set in* $altsets$;
    report ambiguous alts x *at* $start..input.index()$;
    return *min*(x);
  $D := D'$; $input.advance()$;

Function closure. The *closure* operation (Function 7) chases through all ϵ edges reachable from p, the ATN state projected from configuration parameter c and also simulates the call and return of submachines. Function *closure* treats μ and π edges as ϵ edges because mutators should not be executed during prediction and predicates are only evaluated during start state computation. From parameter $c = (p, i, \Gamma)$ and edge $p \xrightarrow{\epsilon} q$, *closure* adds (q, i, Γ) to local working set C. For submachine call edge $p \xrightarrow{B} q$, *closure* adds the *closure* of $(p_B, i, q\Gamma)$. Returning from a submachine stop state p'_B adds the *closure* of configuration (q, i, Γ) in which case c would have been of the form $(p'_B, i, q\Gamma)$. In general, a configuration stack Γ is a graph representing multiple individual stacks. Function *closure* must simulate a return from each of the Γ stack tops. The algorithm uses notation $q\Gamma' \in \Gamma$ to represent all stack tops q of Γ. To avoid non-termination due to *SLL* right recursion and ϵ edges in subrules such as `()+`, *closure* uses a $busy$ set shared among all closure operations used to compute the same D'.

When *closure* reaches stop state p'_A for decision entry rule, A, *LL* and *SLL* predictions behave differently. *LL* prediction pops from the parser call stack γ_0 and "returns" to the state that invoked A's submachine. *SLL* prediction, on the other hand, has no access to the parser call stack and must consider all possible A invocation sites. Function *closure* finds $\Gamma = \#$ (and $p'_B = p'_A$) in this situation because *startState* will have set the initial stack as $\#$ not γ_0. The return behavior at the decision entry rule is what differentiates *SLL* from *LL* parsing.

5.3 Conflict and ambiguity detection

The notion of conflicting configurations is central to *ALL(*)* analysis. Conflicts trigger failover to full *LL* prediction during *SLL* prediction and signal an ambiguity during *LL* prediction. A sufficient condition for a conflict between configurations is when they differ only in the predicted alternative: (p, i, Γ) and (p, j, Γ). Detecting conflicts is aided by two functions. The first, *getConflictSetsPerLoc* (Function 8), collects the sets of production numbers associated with all $(p, -, \Gamma)$ configurations. If a p,Γ pair predicts more than a single production, a

Function 7: *closure*($busy$, $c = (p, i, \Gamma)$) **returns** set C
if $c \in busy$ **then return** $\varnothing$; **else** $busy$ += c;
$C := \{c\}$;
if $p = p'_B$ *(i.e., p is any stop state including* p'_A*)* **then**
  if $\Gamma = \#$ *(i.e., stack is SLL wildcard)* **then**
    C += $\bigcup_{\forall\, p_2 : p_1 \xrightarrow{B} p_2 \in \Delta_{ATN}} closure(busy, (p_2, i, \#))$; // *call site closure*
  else // nonempty *SLL* or *LL* stack
    for $q\Gamma' \in \Gamma$ (i.e., each stack top q in graph Γ) **do**
      C += $closure(busy, (q, i, \Gamma'))$; // *"return" to q*
  return C;
end
foreach $p \xrightarrow{edge} q$ **do**
  switch *edge* **do**
    case B: C += $closure(busy, (p_B, i, q\Gamma))$;
    case π, μ, ϵ: C += $closure(busy, (q, i, \Gamma))$;
return C;

conflict exists. Here is a sample configuration set and the associated set of conflict sets:

$$\{\underbrace{(p,1,\Gamma),(p,2,\Gamma),(p,3,\Gamma)}_{\{1,2,3\}},\underbrace{(p,1,\Gamma'),(p,2,\Gamma')}_{\{1,2\}},\underbrace{(r,2,\Gamma'')}_{\{2\}}\}$$

These conflicts sets indicate that location p, Γ is reachable from productions $\{1, 2, 3\}$, location p, Γ' is reachable from productions $\{1, 2\}$, and r, Γ'' is reachable from production $\{2\}$.

// For each p, Γ get set of alts $\{i\}$ from $(p, -, \Gamma) \in D$ configs
Function 8: *getConflictSetsPerLoc*(D) **returns** set of sets
$s := \varnothing$;
for $(p, -, \Gamma) \in D$ **do** $prods := \{i \mid (p, i, \Gamma)\}$; $s := s \cup prods$;
return s;

The second function, *getProdSetsPerState* (Function 9), is similar but collects the production numbers associated with just ATN state p. For the same configuration set, *getProdSetsPerState* computes these conflict sets:

$$\{\underbrace{(p,1,\Gamma),(p,2,\Gamma),(p,3,\Gamma),(p,1,\Gamma'),(p,2,\Gamma')}_{\{1,2,3\}},\underbrace{(r,2,\Gamma'')}_{\{2\}}\}$$

A sufficient condition for failing over to *LL* prediction (*LLpredict*) from *SLL* would be when there is at least one set of conflicting configurations: *getConflictSetsPerLoc* returns at least one set with more than a single production number. E.g., configurations (p, i, Γ) and (p, j, Γ) exist in parameter D. However, our goal is to continue using *SLL* prediction as long as possible because *SLL* prediction updates the lookahead DFA cache. To that end, *SLL* prediction continues if there is at least one nonconflicting configuration (when *getProdSetsPerState* returns at least one set of size 1). The hope is that more lookahead will lead to a configuration set that predicts a unique production via that nonconflicting configuration. For example, the decision for $S \to a|a|a \bullet_p b$ is ambiguous upon a between productions 1 and 2 but is unambiguous upon ab. (Location $\bullet_p$ is the ATN state between a and b.) After matching input a, the configuration set would be $\{(p'_S, 1, []), (p'_S, 2, []), (p, 3, [])\}$. Function *getConflictSetsPerLoc* returns $\{\{1, 2\}, \{3\}\}$. The next *move-closure* upon b leads to nonconflicting configuration set $\{(p'_S, 3, [])\}$ from $(p, 3, [])$, bypassing the conflict. If all sets returned from *getConflictSetsPerLoc* predict more than one alternative, no amount of lookahead will lead to a unique prediction. Analysis must try again with call stack γ_0 via *LLpredict*.

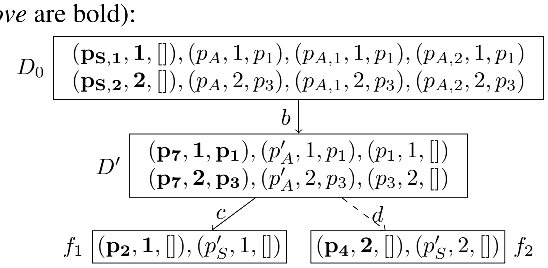
// For each p return set of alts i from $(p, -, -) \in D$ configs.
Function 9: *getProdSetsPerState*(D) **returns** set of sets
$s := \varnothing$;
for $(p, -, -) \in D$ **do** $prods := \{i \mid (p, i, -)\}$; $s := s \cup prods$;
return s;

Conflicts during *LL* simulation are ambiguities and occur when each conflict set from *getConflictSetsPerLoc* contains more than 1 production—every location in D is reachable from more than a 1 production. Once multiple subparsers reach the same $(p, -, \Gamma)$, all future simulation derived from $(p, -, \Gamma)$ will behave identically. More lookahead will not resolve the ambiguity. Prediction could terminate at this point and report lookahead prefix u as ambiguous but *LLpredict* continues until it is sure for *which* productions u is ambiguous. Consider conflict sets $\{1,2,3\}$ and $\{2,3\}$. Because both have degree greater than one, the sets represent an ambiguity, but additional input will identify whether $u \preceq w_r$ is ambiguous upon $\{1,2,3\}$ or $\{2,3\}$. Function *LLpredict* continues until all conflict sets that identify ambiguities are equal; condition $x = y$ and $|x| > 1\ \forall\, x, y \in altsets$ embodies this test.

To detect conflicts, the algorithm compares graph-structured stacks frequently. Technically, a conflict occurs when configurations (p, i, Γ) and (p, j, Γ') occur in the same configuration set with $i \neq j$ and at least one stack trace γ in common to both Γ and Γ'. Because checking for graph intersection is expensive, the algorithm uses equality, $\Gamma = \Gamma'$, as a heuristic. Equality is much faster because of the shared subgraphs. The graph equality algorithm can often check node identity to compare two entire subgraphs. In the worst case, the equality versus subset heuristic delays conflict detection until the GSS between conflicting configurations are simple linear stacks where graph intersection is the same as graph equality. The cost of this heuristic is deeper lookahead.

5.4 Sample DFA construction

To illustrate algorithm behavior, consider inputs bc and bd for the grammar and ATN in Figure 8. ATN simulation for decision S launches subparsers at left edge nodes $p_{S,1}$ and $p_{S,2}$ with initial D_0 configurations $(p_{S,1}, 1, [])$ and $(p_{S,2}, 2, [])$. Function *closure* adds three more configurations to D_0 as it "calls" A with "return" nodes p_1 and p_3. Here is the DFA resulting from ATN simulation upon bc and then bd (configurations added by *move* are bold):



After bc prediction, the DFA has states D_0, D', and f_1. From DFA state D', *closure* reaches the end of A and pops from the Γ stacks, returning to ATN states in S. State f_1 uniquely predicts production number 1. State f_2 is created and connected to the DFA (shown with dashed arrow) during prediction of the second phrase, bd. Function *adaptivePredict* first uses DFA simulation to get to D' from D_0 upon b. Before having seen bd, D' has no d edge so *adaptivePredict* must use ATN simulation to add edge $D' \xrightarrow{d} f_2$.

6. Theoretical results

This section identifies the key *ALL(*)* theorems and shows parser time complexity. See Appendix A for detailed proofs.

Theorem 6.1. *(Correctness). The* ALL(*) *parser for non-left-recursive G recognizes sentence w* iff $w \in L(G)$.

Theorem 6.2. ALL(*) *languages are closed under union.*

Theorem 6.3. ALL(*) *parsing of n symbols has $O(n^4)$ time.*

Theorem 6.4. *A GSS has $O(n)$ nodes for n input symbols.*

Theorem 6.5. *Two-stage parsing for non-left-recursive G recognizes sentence w* iff $w \in L(G)$.

7. Empirical results

We performed experiments to compare the performance of *ALL(*)* Java parsers with other strategies, to examine *ALL(*)* throughput for a variety of other languages, to highlight the effect of the lookahead DFA cache on parsing speed, and to provide evidence of linear *ALL(*)* performance in practice.

7.1 Comparing *ALL(*)*'s speed to other parsers

Our first experiment compared Java parsing speed across 10 tools and 8 parsing strategies: hand-tuned recursive-descent with precedence parsing, *LL(k)*, *LL(*)*, PEG, $LALR(1)$, *ALL(*)* GLR, and GLL. Figure 9 shows the time for each tool to parse the 12,920 source files of the Java 6 library and compiler. We chose Java because it was the most commonly available grammar among tools and sample Java source is plentiful. The Java grammars used in this experiment came directly from the associated tool except for DParser and Elkhound, which did not offer suitable Java grammars. We ported ANTLR's Java grammar to the meta-syntax of those tools using unambiguous arithmetic expressions rules. We also embedded `merge` actions in the Elkhound grammar to disambiguate during the parse to mimic ANTLR's ambiguity resolution. All input files were loaded into RAM before parsing and times reflect the average time measured over 10 complete corpus passes, skipping the first two to ensure JIT compiler warm-up. For *ALL(*)*, we used the two-stage parse from Section 3.2. The test machine was a 6 core 3.33Ghz 16G RAM Mac OS X 10.7 desktop running the Java 7 virtual machine. Elkhound and DParser parsers were implemented in C/C++, which does not have a garbage collector running concurrently. Elkhound was last updated in 2005 and no longer builds on Linux or OS X, but we were able to build it on Windows 7 (4 core 2.67Ghz 24G RAM). Elkhound also can only read from a file so Elkhound parse times are not comparable. In an effort to control for machine speed differences and RAM vs SSD, we computed the time ratio of our Java test rig on our OS X machine reading from RAM to the test rig running on Windows pulling from SSD. Our reported Elkhound times are the Windows time multiplied by that OS X to Windows ratio.

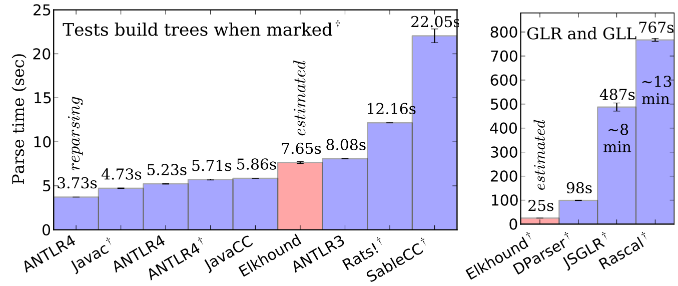


Figure 9. Comparing Java parse times for 10 tools and 8 strategies on Java 6 Library and compiler source code (smaller is faster). 12,920 files, 3.6M lines, size 123M. Tool descriptors comprise: "**tool name** *version* [*strategy*]." **ANTLR4** 4.1 [*ALL(*)*]; **Javac** 7 [handbuilt recursive-descent and precedence parser for expressions]; **JavaCC** 5.0 [*LL(k)*]; **Elkhound** 2005.08.22b [GLR] (*tested on Windows*); **ANTLR3** 3.5 [*LL(*)*]; **Rats!** 2.3.1 [PEG]; **SableCC** 3.7 [$LALR(1)$]; **DParser** 1.2 [GLR]; **JSGLR** (from Spoofax) 1.1 [GLR]; **Rascal** 0.6.1 [GLL]. Tests run 10x with generous memory, average/stddev computed on last 8 to avoid JIT cost. Error bars are negligible but show some variability due to garbage collection. To avoid a log scale, we use a separate graph for GLR, GLL parse times.

For this experiment, *ALL(*)* outperforms the other parser generators and is only about 20% slower than the handbuilt parser in the Java compiler itself. When comparing runs with tree construction (marked with † in Figure 9), ANTLR 4 is about 4.4x faster than Elkhound, the fastest GLR tool we tested, and 135x faster than GLL (Rascal). ANTLR 4's nondeterministic *ALL(*)* parser was slightly faster than JavaCC's deterministic *LL(k)* parser and about 2x faster than *Rats!*'s PEG. In a separate test, we found that *ALL(*)* outperforms *Rats!* on its own PEG grammar converted to ANTLR syntax (8.77s vs 12.16s). The $LALR(1)$ parser did not perform well against the *LL* tools but that could be SableCC's implementation rather than a deficiency of $LALR(1)$. (The Java grammar from JavaCUP, another $LALR(1)$ tool, was incomplete and unable to parse the corpus.) When reparsing the corpus, *ALL(*)* lookahead gets cache hits at each decision and parsing is 30% faster at 3.73s. When reparsing with tree construction (time not shown), *ALL(*)* outperforms handbuilt Javac (4.4s vs 4.73s). Reparsing speed matters to tools such as development environments.

The GLR parsers we tested are up to two orders of magnitude slower at Java parsing than *ALL(*)*. Of the GLR tools, Elkhound has the best performance primarily because it relies on a linear $LR(1)$ stack instead of a GSS whenever possible. Further, we allowed Elkhound to disambiguate during the parse like *ALL(*)*. Elkhound uses a separate lexer, unlike JSGLR and DParser, which are scannerless. A possible explanation for the observed performance difference with *ALL(*)* is that the Java grammar we ported to Elkhound and DParser is biased towards *ALL(*)*, but this objection is not well-founded. GLR should also benefit from highly-deterministic and unambiguous grammars. GLL has the slowest speed in this test perhaps because Rascal's team ported SDF's GLR Java grammar,

Tool	Time	RAM (M)
Javac†	89 ms	7
ANTLR4	201 ms	8
JavaCC	206 ms	7
ANTLR4†	360 ms	8
ANTLR3	1048 ms	143
SableCC†	1,174 ms	201
Rats!†	1,365 ms	22
JSGLR†	15.4 sec	1,030
Rascal†	24.9 sec	2,622
(*no DFA*) ANTLR4	42.5 sec	27
elkhound[a]	3.35 min	3
DParser†	10.5 hours	100+
elkhound†	out of mem	5390+

Figure 10. Time and space to parse and optionally build trees for 3.2M Java file. Space is median reported after GC during parse using `-XX:+PrintGC` option (process monitoring for C++). Times include lexing; all input preloaded. †Building trees. [a]Disambiguating during the parse, no trees, *estimated time*.

which is not optimized for GLL (Grammar variations can affect performance.) Rascal is also scannerless and is currently the only available GLL tool.

The biggest issue with general algorithms is that they are highly unpredictable in time and space, which can make them unsuitable for some commercial applications. Figure 10 summarizes the performance of the same tools against a single 3.2M Java file. Elkhound took 7.65s to parse the 123M Java corpus, but took 3.35 minutes to parse the 3.2M Java file. It crashed (out of memory) with parse forest construction on. DParser's time jumped from a corpus time of 98s to 10.5 hours on the 3.2M file. The speed of Rascal and JSGLR scale reasonably well to the 3.2M file, but use 2.6G and 1G RAM, respectively. In contrast, *ALL(*)* parses the 3.2M file in 360ms with tree construction using 8M. ANTLR 3 is fast but is slower and uses more memory (due to backtracking memoization) than ANTLR 4.

7.2 *ALL(*)* performance across languages

Figure 11 gives the bytes-per-second throughput of *ALL(*)* parsers for 8 languages, including Java for comparison. The number of test files and file sizes vary greatly (according to the input we could reasonably collect); smaller files yield higher parse-time variance.

- **C** Derived from C11 specification; has no indirect left-recursion, altered stack-sensitive rule to render *SLL* (see text below): 813 preprocessed files, 159.8M source from `postgres` database.
- **Verilog2001** Derived from Verilog 2001 spec, removed indirect left-recursion: 385 files, 659k from [3] and web.
- **JSON** Derived from spec. 4 files, 331k from twitter.
- **DOT:** Derived from spec. 48 files 19.5M collected from web.
- **Lua**: Derived from Lua 5.2 spec. 751 files, 123k from github.
- **XML** Derived from spec. 1 file, 117M from XML benchmark.
- **Erlang** Derived from *LALR(1)* grammar. 500 preproc'd files, 8M.

Some of these grammars yield reasonable but much slower parse times compared to Java and XML but demonstrate that programmers can convert a language specification to ANTLR's meta-syntax and get a working grammar without major modi-

Grammar	KB/sec
XML	45,993
Java	24,972
JSON	17,696
DOT	16,152
Lua	5,698
C	4,238
Verilog2001	1,994
Erlang	751

Figure 11. Throughput in KByte/sec. Lexing+parsing; all input preloaded into RAM.

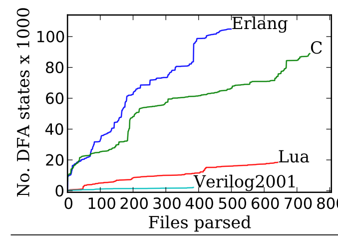


Figure 12. DFA growth rate vs number of files parsed. Files parsed in disk order.

fications. (In our experience, grammar specifications are rarely tuned to a particular tool or parsing strategy and are often ambiguous.) Later, programmers can use ANTLR's profiling and diagnostics to improve performance, as with any programming task. For example, the C11 specification grammar is *LL* not *SLL* because of rule `declarationSpecifiers`, which we altered to be *SLL* in our C grammar (getting a 7x speed boost).

7.3 Effect of lookahead DFA on performance

The lookahead DFA cache is critical to *ALL(*)* performance. To demonstrate the cache's effect on parsing speed, we disabled the DFA and repeated our Java experiments. Consider the 3.73s parse time from Figure 9 to reparse the Java corpus with pure cache hits. With the lookahead DFA cache disabled completely, the parser took 12 minutes (717.6s). Figure 10 shows that disabling the cache increases parse time from 203ms to 42.5s on the 3.2M file. This performance is in line with the high cost of GLL and GLR parsers that also do not reduce parser speculation by memoizing parsing decisions. As an intermediate value, clearing the DFA cache before parsing each corpus file yields a total time of 34s instead of 12 minutes. This isolates cache use to a single file and demonstrates that cache warm-up occurs quickly even within a single file.

DFA size increases linearly as the parser encounters new lookahead phrases. Figure 12 shows the growth in the number of DFA states as the (slowest four) parsers from Figure 11 encounter new files. Languages like C that have constructs with common left-prefixes require deep lookahead in *LL* parsers to distinguish phrases; e.g., `struct {...} x;` and `struct {...} f();` share a large left-prefix. In contrast, the Verilog2001 parser uses very few DFA states (but runs slower due to a non-*SLL* rule). Similarly, after seeing the entire 123M Java corpus, the Java parser uses just 31,626 DFA states, adding an average of ˜2.5 states per file parsed. DFA size does, however, continue to grow as the parser encounters unfamiliar input. Programmers can clear the cache and *ALL(*)* will adapt to subsequent input.

7.4 Empirical parse-time complexity

Given the wide range of throughput in Figure 11, one could suspect nonlinear behavior for the slower parsers. To investigate, we plotted parse time versus file size in Figure 13 and drew least-squares regression and LOWESS [6] data fitting curves. LOWESS curves are parametrically unconstrained (not required to be a line or any particular polynomial) and they vir-

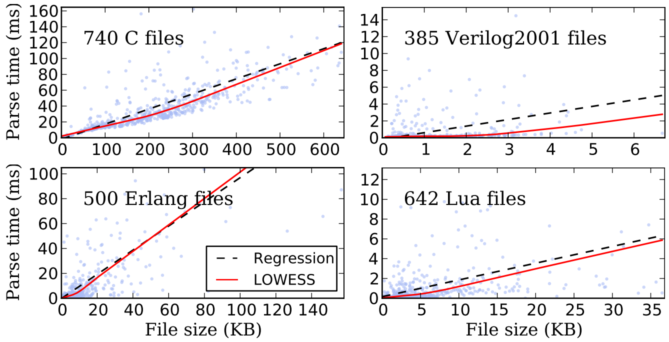


Figure 13. Linear parse time vs file size. Linear regression (dashed line) and LOWESS unconstrained curve coincide, giving strong evidence of linearity. Curves computed on (bottom) 99% of file sizes but zooming in to show detail in the bottom 40% of parse times.

tually mirrors each regression line, providing strong evidence that the relationship between parse time and input size is linear. The same methodology shows that the parser generated from the non-*SLL* grammar (not shown) taken from the C11 specification is also linear, despite being much slower than our *SLL* version.

We have yet to see nonlinear behavior in practice but the theoretical worst-case behavior of *ALL(*)* parsing is $O(n^4)$. Experimental parse-time data for the following contrived worst-case grammar exhibits quartic behavior for input a, aa, aaa, ..., a^n (with $n \leq 120$ symbols we could test in a reasonable amount of time). $S \rightarrow A\,\$,\quad A \rightarrow aAA \,|\, aA \,|\, a$. The generated parser requires a prediction upon each input symbol and each prediction must examine all remaining input. The *closure* operation performed for each input symbol must examine the entire depth of the GSS, which could be size n. Finally, merging two GSS can take $O(n)$ in our implementation, yielding $O(n^4)$ complexity.

From these experiments, we conclude that shifting grammar analysis to parse-time to get *ALL(*)* strength is not only practical but yields extremely efficient parsers, competing with the hand-tuned recursive-descent parser of the Java compiler. Memoizing analysis results with DFA is critical to such performance. Despite $O(n^4)$ theoretical complexity, *ALL(*)* appears to be linear in practice and does not exhibit the unpredictable performance or large memory footprint of the general algorithms.

8. Related work

For decades, researchers have worked towards increasing the recognition strength of efficient but non-general *LL* and *LR* parsers and increasing the efficiency of general algorithms such as Earley's $O(n^3)$ algorithm [8]. Parr [21] and Charles [4] statically generated *LL(k)* and *LR(k)* parsers for $k > 1$. Parr and Fisher's *LL(*)* [20] and Bermudez and Schimpf's $LAR(m)$ [2] statically computed *LL* and *LR* parsers augmented with cyclic DFA that could examine arbitrary amounts of lookahead. These parsers were based upon *LL*-regular [12] and *LR*-regular [7] parsers, which have the undesirable property of being undecidable in general. Introducing backtracking dramatically increases recognition strength and avoids static grammar analysis undecidability issues but is undesirable because it has embedded mutators issues, reduces performance, and complicates single-step debugging. Packrat parsers (PEGs) [9] try decision productions in order and pick the first that succeeds. PEGs are $O(n)$ because they memoize partial parsing results but suffer from the $a \,|\, ab$ quirk where ab is silently unmatchable.

To improve general parsing performance, Tomita [26] introduced GLR, a general algorithm based upon *LR(k)* that conceptually forks subparsers at each conflicting *LR(k)* state at parse-time to explore all possible paths. Tomita shows GLR to be 5x-10x faster than Earley. A key component of GLR parsing is the graph-structured stack (GSS) [26] that prevents parsing the same input twice in the same way. (GLR pushes input symbols and *LR* states on the GSS whereas *ALL(*)* pushes ATN states.) Elkhound [18] introduced hybrid GLR parsers that use a single stack for all $LR(1)$ decisions and a GSS when necessary to match ambiguous portions of the input. (We found Elkhound's parsers to be faster than those of other GLR tools.) GLL [25] is the *LL* analog of GLR and also uses subparsers and a GSS to explore all possible paths; GLL uses $k = 1$ lookahead where possible for efficiency. GLL is $O(n^3)$ and GLR is $O(n^{p+1})$ where p is the length of the longest grammar production.

Earley parsers scale gracefully from $O(n)$ for deterministic grammars to $O(n^3)$ in the worst case for ambiguous grammars but performance is not good enough for general use. *LR(k)* state machines can improve the performance of such parsers by statically computing as much as possible. LRE [16] is one such example. Despite these optimizations, general algorithms are still very slow compared to deterministic parsers augmented with deep lookahead.

The problem with arbitrary lookahead is that it is impossible to compute statically for many useful grammars (the *LL*-regular condition is undecidable.) By shifting lookahead analysis to parse-time, *ALL(*)* gains the power to handle any grammar without left recursion because it can launch subparsers to determine which path leads to a valid parse. Unlike GLR, speculation stops when all remaining subparsers are associated with a single alternative production, thus, computing the minimum lookahead sequence. To get performance, *ALL(*)* records a mapping from that lookahead sequence to the predicted production using a DFA for use by subsequent decisions. The context-free language subsets encountered during a parse are finite and, therefore, *ALL(*)* lookahead languages are regular. Ancona *et al* [1] also performed parse-time analysis, but they only computed fixed *LR(k)* lookahead and did not adapt to the actual input as *ALL(*)* does. Perlin [23] operated on an RTN like *ALL(*)* and computed $k = 1$ lookahead during the parse.

ALL(*) is similar to Earley in that both are top-down and operate on a representation of the grammar at parse-time, but Earley is parsing not computing lookahead DFA. In that sense, Earley is not performing grammar analysis. Earley also does not manage an explicit GSS during the parse. Instead, items in Earley states have "parent pointers" that refer to other states that, when threaded together, form a GSS. Earley's *SCANNER* operation correspond to *ALL(*)*'s *move* function. The *PREDIC-*

TOR and *COMPLETER* operations correspond to push and pop operations in *ALL(*)*'s *closure* function. An Earley state is the set of all parser configurations reachable at some absolute input depth whereas an *ALL(*)* DFA state is a set of configurations reachable from a lookahead depth relative to the current decision. Unlike the completely general algorithms, *ALL(*)* seeks a single parse of the input, which allows the use of an efficient *LL* stack during the parse.

Parsing strategies that continuously speculate or support ambiguity have difficulty with mutators because they are hard to undo. A lack of mutators reduces the generality of semantic predicates that alter the parse as they cannot test arbitrary state computed previously during the parse. *Rats!* [10] supports restricted semantic predicates and Yakker [13] supports semantic predicates that are functions of previously-parsed terminals. Because *ALL(*)* does not speculate during the actual parse, it supports arbitrary mutators and semantic predicates. Space considerations preclude a more detailed discussion of related work here; a more detailed analysis can be found in reference [20].

9. Conclusion

ANTLR 4 generates an *ALL(*)* parser for any CFG without indirect or hidden left-recursion. *ALL(*)* combines the simplicity, efficiency, and predictability of conventional top-down *LL(k)* parsers with the power of a GLR-like mechanism to make parsing decisions. The critical innovation is to shift grammar analysis to parse-time, caching analysis results in lookahead DFA for efficiency. Experiments show *ALL(*)* outperforms general (Java) parsers by orders of magnitude, exhibiting linear time and space behavior for 8 languages. The speed of the *ALL(*)* Java parser is within 20% of the Java compiler's hand-tuned recursive-descent parser. In theory, *ALL(*)* is $O(n^4)$, inline with the low polynomial bound of GLR. ANTLR is widely used in practice, indicating that *ALL(*)* provides practical parsing power without sacrificing the flexibility and simplicity of recursive-descent parsers desired by programmers.

10. Acknowledgments

We thank Elizabeth Scott, Adrian Johnstone, and Mark Johnson for discussions on parsing algorithm complexity. Eelco Visser and Jurgen Vinju provided code to test isolated parsers from JS-GLR and Rascal. Etienne Gagnon generated SableCC parsers.

References

[1] ANCONA, M., DODERO, G., GIANUZZI, V., AND MORGAVI, M. Efficient construction of $LR(k)$ states and tables. *ACM Trans. Program. Lang. Syst. 13*, 1 (Jan. 1991), 150–178.

[2] BERMUDEZ, M. E., AND SCHIMPF, K. M. Practical arbitrary lookahead LR parsing. *Journal of Computer and System Sciences 41*, 2 (1990).

[3] BROWN, S., AND VRANESIC, Z. *Fundamentals of Digital Logic with Verilog Design*. McGraw-Hill series in ECE. 2003.

[4] CHARLES, P. *A Practical Method for Constructing Efficient LALR(k) Parsers with Automatic Error Recovery*. PhD thesis, New York University, New York, NY, USA, 1991.

[5] CLARKE, K. The top-down parsing of expressions. Unpublished technical report, Dept. of Computer Science and Statistics, Queen Mary College, London, June 1986.

[6] CLEVELAND, W. S. Robust Locally Weighted Regression and Smoothing Scatterplots. *Journal of the American Statistical Association 74* (1979), 829–836.

[7] COHEN, R., AND CULIK, K. LR-Regular grammars—an extension of $LR(k)$ grammars. In *SWAT '71* (Washington, DC, USA, 1971), IEEE Computer Society, pp. 153–165.

[8] EARLEY, J. An efficient context-free parsing algorithm. *Communications of the ACM 13*, 2 (1970), 94–102.

[9] FORD, B. Parsing Expression Grammars: A recognition-based syntactic foundation. In *POPL* (2004), ACM Press, pp. 111–122.

[10] GRIMM, R. Better extensibility through modular syntax. In *PLDI* (2006), ACM Press, pp. 38–51.

[11] HOPCROFT, J., AND ULLMAN, J. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[12] JARZABEK, S., AND KRAWCZYK, T. LL-Regular grammars. *Information Processing Letters 4*, 2 (1975), 31 – 37.

[13] JIM, T., MANDELBAUM, Y., AND WALKER, D. Semantics and algorithms for data-dependent grammars. In *POPL* (2010).

[14] JOHNSON, M. The computational complexity of GLR parsing. In *Generalized LR Parsing*, M. Tomita, Ed. Kluwer, Boston, 1991, pp. 35–42.

[15] KIPPS, J. *Generalized LR Parsing*. Springer, 1991, pp. 43–59.

[16] MCLEAN, P., AND HORSPOOL, R. N. A faster Earley parser. In *CC* (1996), Springer, pp. 281–293.

[17] MCPEAK, S. Elkhound: A fast, practical GLR parser generator. Tech. rep., University of California, Berkeley (EECS), Dec. 2002.

[18] MCPEAK, S., AND NECULA, G. C. Elkhound: A fast, practical GLR parser generator. In *CC* (2004), pp. 73–88.

[19] PARR, T. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Programmers, 2013. ISBN 978-1-93435-699-9.

[20] PARR, T., AND FISHER, K. $LL(*)$: The Foundation of the ANTLR Parser Generator. In *PLDI* (2011), pp. 425–436.

[21] PARR, T. J. *Obtaining practical variants of $LL(k)$ and $LR(k)$ for $k > 1$ by splitting the atomic k-tuple*. PhD thesis, Purdue University, West Lafayette, IN, USA, 1993.

[22] PARR, T. J., AND QUONG, R. W. Adding Semantic and Syntactic Predicates to $LL(k)$—*pred-*$LL(k)$. In *CC* (1994).

[23] PERLIN, M. LR recursive transition networks for Earley and Tomita parsing. In *Proceedings of the 29th Annual Meeting on Association for Computational Linguistics* (1991), ACL '91, pp. 98–105.

[24] PLEVYAK, J. DParser: GLR parser generator, Visited Oct. 2013.

[25] SCOTT, E., AND JOHNSTONE, A. GLL parsing. *Electron. Notes Theor. Comput. Sci. 253*, 7 (Sept. 2010), 177–189.

[26] TOMITA, M. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, 1986.

[27] WOODS, W. A. Transition network grammars for natural language analysis. *Comm. of the ACM 13*, 10 (1970), 591–606.

A. Correctness and complexity analysis

Theorem A.1. ALL(*) *languages are closed under union.*

Proof. Let predicated grammars $G_1 = (N_1, T, P_1, S_1, \Pi_1, \mathcal{M}_1)$ and $G_2 = (N_2, T, P_2, S_2, \Pi_2, \mathcal{M}_2)$ describe $L(G_1)$ and $L(G_2)$, respectively. For applicability to both parsers and scannerless parsers, assume that the terminal space T is the set of valid characters. Assume $N_1 \cap N_2 = \varnothing$ by renaming nonterminals if necessary. Assume that the predicates and mutators of G_1 and G_2 operate in disjoint environments, $\mathbb{S}_1$ and $\mathbb{S}_2$. Construct:

$$G' = (N_1 \cup N_2, T, P_1 \cup P_2, S', \Pi_1 \cup \Pi_2, \mathcal{M}_1 \cup \mathcal{M}_2)$$

with $S' = S_1 \mid S_2$. Then, $L(G') = L(G_1) \cup L(G_2)$. □

Lemma A.1. *The* ALL(*) *parser for non-left-recursive* G *with lookahead DFA deactivated recognizes sentence* w iff $w \in L(G)$.

Proof. The ATN for G recognizes w *iff* $w \in L(G)$. Therefore, we can equivalently prove that *ALL(*)* is a faithful implementation of an ATN. Without lookahead DFA, prediction is a straightforward ATN simulator: a top-down parser that makes accurate parsing decisions using GLR-like subparsers that can examine the entire remaining input and ATN submachine call stack. □

Theorem A.2. *(Correctness). The* ALL(*) *parser for non-left-recursive* G *recognizes sentence* w iff $w \in L(G)$.

Proof. Lemma A.1 shows that an *ALL(*)* parser correctly recognizes w without the DFA cache. The essence of the proof then is to show that *ALL(*)*'s adaptive lookahead DFA do not break the parse by giving different prediction decisions than straightforward ATN simulation. We only need to consider the case of unpredicated *SLL* parsing as *ALL(*)* only caches decision results in this case.

if case: By induction on the state of the lookahead DFA for any given decision A. *Base case.* The first prediction for A begins with an empty DFA and must activate ATN simulation to choose alternative α_i using prefix $u \preceq w_r$. As ATN simulation yields proper predictions, the *ALL(*)* parser correctly predicts α_i from a cold start and then records the mapping from $u : i$ in the DFA. If there is a single viable alternative, i is the associated production number. If ATN simulation finds multiple viable alternatives, i is the minimum production number associated with alternatives from that set.

Induction step. Assume the lookahead DFA correctly predicts productions for every u prefix of w_r seen by the parser at A. We must show that starting with an existing DFA, *ALL(*)* properly adds a path through the DFA for unfamiliar u prefix of w'_r. There are several cases:

1. $u \preceq w'_r$ and $u \preceq w_r$ for a previous w_r. The lookahead DFA gives the correct answer for u by induction assumption. The DFA is not updated.
2. $w'_r = bx$ and all previous $w_r = ay$ for some $a \neq b$. This case reduces to the cold-start base case because there is no $D_0 \xrightarrow{b} D$ edge. ATN simulation predicts α_i and adds path for $u \preceq w'_r$ from D_0 to f_i.
3. $w'_r = vax$ and $w_r = vby$ for some previously seen w_r with common prefix v and $a \neq b$. DFA simulation reaches D from D_0 for input v. D has an edge for b but not a. ATN simulation predicts α_i and augments the DFA, starting with an edge on a from D leading eventually to f_i.

only if case: The *ALL(*)* parser reports a syntax error for $w \notin L(G)$. Assume the opposite, that the parser successfully parses w. That would imply that there exists an ATN configuration derivation sequence $(\mathbb{S}, p_S, [], w) \mapsto^* (\mathbb{S}', p'_S, [], \epsilon)$ for w through G's corresponding ATN. But that would require $w \in L(G)$, by Definition **??**. Therefore the *ALL(*)* parser reports a syntax error for w. The accuracy of the *ALL(*)* lookahead cache is irrelevant because there is no possible path through the ATN or parser. □

Lemma A.2. *The set of viable productions for* LL *is always a subset of* SLL*'s viable productions for a given decision* A *and remaining input string* w_r.

Proof. If the key *move-closure* analysis operation does not reach stop state p'_A for submachine A, *SLL* and *LL* behave identically and so they share the same set of viable productions.

If *closure* reaches the stop state for the decision entry rule, p'_A, there are configurations of the form $(p'_A, -, \gamma)$ where, for convenience, the usual GSS Γ is split into single stacks, γ. In *LL* prediction mode, $\gamma = \gamma_0$, which is either a single stack or empty if $A = S$. In *SLL* mode, $\gamma = \#$, signaling no stack information. Function *closure* must consider all possible γ_0 parser call stacks. Since any single stack must be contained within the set of all possible call stacks, *LL* closure operations consider at most the same number of paths through the ATN as *SLL*. □

Lemma A.3. *For* $w \notin L(G)$ *and non-left-recursive* G*,* SLL *reports a syntax error.*

Proof. As in the *only if case* of Theorem 6.1, there is no valid ATN configuration derivation for w regardless of how *adaptivePredict* chooses productions. □

Theorem A.3. *Two-stage parsing for non-left-recursive* G *recognizes sentence* w iff $w \in L(G)$.

Proof. By Lemma A.3, *SLL* and *LL* behave the same when $w \notin L(G)$. It remains to show that *SLL* prediction either behaves like *LL* for input $w \in L(G)$ or reports a syntax error, signalling a need for the *LL* second stage. Let $\mathcal{V}$ and $\mathcal{V}'$ be the set of viable production numbers for A using *SLL* and *LL*, respectively. By Lemma A.2, $\mathcal{V}' \subseteq \mathcal{V}$. There are two cases to consider:

1. If $min(\mathcal{V}) = min(\mathcal{V}')$, *SLL* and *LL* choose the same production. *SLL* succeeds for w. E.g., $\mathcal{V} = \{1, 2, 3\}$ and $\mathcal{V}' = \{1, 3\}$ or $\mathcal{V} = \{1\}$ and $\mathcal{V}' = \{1\}$.
2. If $min(\mathcal{V}) \neq min(\mathcal{V}')$ then $min(\mathcal{V}) \notin \mathcal{V}'$ because *LL* finds $min(\mathcal{V})$ nonviable. *SLL* would report a syntax error. E.g., $\mathcal{V} = \{1, 2, 3\}$ and $\mathcal{V}' = \{2, 3\}$ or $\mathcal{V} = \{1, 2\}$ and $\mathcal{V}' = \{2\}$.

In all possible combinations of $\mathcal{V}$ and $\mathcal{V}'$, *SLL* behaves like *LL* or reports a syntax error for $w \in L(G)$. □

Theorem A.4. *A GSS has $O(n)$ nodes for n input symbols.*

Proof. For nonterminals N and ATN states Q, there are $|N| \times |Q|$ $p \xrightarrow{A} q$ ATN transitions if every every grammar position invokes every nonterminal. That limits the number of new GSS nodes to $|Q|^2$ for a closure operation (which cannot transition terminal edges). *ALL(*)* performs $n + 1$ closures for n input symbols giving $|Q|^2(n + 1)$ nodes or $O(n)$ as Q is not a function of the input. □

Lemma A.4. *Examining a lookahead symbol has $O(n^2)$ time.*

Proof. Lookahead is a *move-closure* operation that computes new target DFA state D' as a function of the ATN configurations in D. There are $|Q| \times m \approx |Q|^2$ configurations of the form $(p, i, \_) \in D$ for $|Q|$ ATN states and m alternative productions in the current decision. The cost of *move* is not a function of input size n. Closure of D computes *closure*(c) $\forall\ c \in D$ and *closure*(c) can walk the entire GSS back to the root (the empty stack). That gives a cost of $|Q|^2$ configurations times $|Q|^2(n+1)$ GSS nodes (per Theorem A.4) or $O(n)$ add operations to build D'. Adding a configuration is dominated by the graph merge, which (in our implementation) is proportional to the depth of the graph. The total cost for *move-closure* is $O(n^2)$. □

Theorem A.5. ALL(*) *parsing of n input symbols has $O(n^4)$ time.*

Proof. Worst case, the parser must examine all remaining input symbols during prediction for each of n input symbols giving $O(n^2)$ lookahead operations. The cost of each lookahead operation is $O(n^2)$ by Lemma A.4 giving overall parsing cost $O(n^4)$. □

B. Pragmatics

This section describes some of the practical considerations associated with implementing the *ALL(*)* algorithm.

B.1 Reducing warm-up time

Many decisions in a grammar are $LL(1)$ and they are easy to identify statically. Instead of always generating "switch on *adaptivePredict*" decisions in the recursive-descent parsers, ANTLR generates "switch on token type" decisions whenever possible. This $LL(1)$ optimization does not affect the size of the generated parser but reduces the number of lookahead DFA that the parser must compute.

Originally, we anticipated "training" a parser on a large input corpus and then serializing the lookahead DFA to disk to avoid re-computing DFA for subsequent parser runs. As shown in the Section 7, lookahead DFA construction is fast enough that serializing and deserializing the DFA is unnecessary.

B.2 Semantic predicate evaluation

For clarity, the algorithm described in this paper uses pure ATN simulation for all decisions that have semantic predicates on production left edges. In practice, ANTLR uses lookahead DFA that track predicates in accept states to handle semantic-context-sensitive prediction. Tracking the predicates in the DFA allows prediction to avoid expensive ATN simulation if predicate evaluation during *SLL* simulation predicts a unique production. Semantic predicates are not common but are critical to solving some context-sensitive parsing problems; e.g., predicates are used internally by ANTLR to encode operator precedence when rewriting left-recursive rules. So it is worth the extra complexity to evaluate predicates during *SLL* prediction. Consider the predicated rule from Section 2.1:

```
id : ID | {!enum_is_keyword}? 'enum' ;
```

The second production is viable only when `!enum_is_keyword` evaluates to true. In the abstract, that means the parser would need two lookahead DFA, one per semantic condition. Instead, ANTLR's *ALL(*)* implementation creates a DFA (via *SLL* prediction) with edge $D_0 \xrightarrow{\texttt{enum}} f_2$ where f_2 is an augmented DFA accept state that tests `!enum_is_keyword`. Function *adaptivePredict* returns production 2 upon `enum` if `!enum_is_keyword` else throws a no-viable-alternative exception.

The algorithm described in this paper also does not support semantic predicates outside of the decision entry rule. In practice, *ALL(*)* analysis must evaluate all predicates reachable from the decision entry rule without stepping over a terminal edge in the ATN. For example, the simplified *ALL(*)* algorithm in this paper considers only predicates π_1 and π_2 for the productions of S in the following (ambiguous) grammar.

$S \rightarrow \{\pi_1\}?Ab \,|\, \{\pi_2\}?Ab$
$A \rightarrow \{\pi_3\}?a \,|\, \{\pi_4\}?a$

Input ab matches either alternative of S and, in practice, ANTLR evaluates "π_1 and (π_3 or π_4)" to test the viability of S's first production not just π_1. After simulating S and A's ATN submachines, the lookahead DFA for S would be $D_0 \xrightarrow{a} D' \xrightarrow{b} f_{1,2}$. Augmented accept state $f_{1,2}$ predicts productions 1 or 2 depending on semantic contexts $\pi_1 \wedge (\pi_3 \vee \pi_4)$ and $\pi_2 \wedge (\pi_3 \vee \pi_4)$, respectively. To keep track of semantic context during *SLL* simulation, ANTLR ATN configurations contain extra element π: (p, i, Γ, π). Element π carries along semantic context and ANTLR stores predicate-to-production pairs in the augmented DFA accept states.

B.3 Error reporting and recovery

ALL(*) prediction can scan arbitrarily far ahead so erroneous lookahead sequences can be long. By default, ANTLR-generated parsers print the entire sequence. To recover, parsers consume tokens until a token appears that could follow the current rule. ANTLR provides hooks to override reporting and recovery strategies.

ANTLR parsers issue error messages for invalid input phrases and attempt to recover. For mismatched tokens, ANTLR attempts single token insertion and deletion to resynchronize. If the remaining input is not consistent with any production of the current nonterminal, the parser consumes tokens until it finds a token that could reasonably follow the current nonterminal. Then the parser continues parsing as if the current nonterminal had succeeded. ANTLR improves error recovery over ANTLR 3 for EBNF subrules by inserting synchronization checks at the start and at the "loop" continuation test to avoid prematurely exiting the subrule. For example, consider the following class definition rule.

```
classdef : 'class' ID '{' member+ '}' ;
member : 'int' ID ';' ;
```

An extra semicolon in the member list such as `int i;; int j;` should not force surrounding rule `classdef` to abort. Instead, the parser ignores the extra semicolon and looks for another `member`. To reduce cascading error messages, the parser issues no further messages until it correctly matches a token.

B.4 Multi-threaded execution

Applications often require parallel execution of multiple parser instances, even for the same language. For example, web-based application servers parse multiple incoming XML or JSON data streams using multiple instances of the same parser. For memory efficiency, all *ALL(*)* parser instances for a given language must share lookahead DFA. The Java code that ANTLR generates uses a shared memory model and threads for concurrency, which means parsers must update shared DFA in a thread-safe manner. Multiple threads can be simulating the DFA while other threads are adding states and edges to it. Our goal is thread safety, but concurrency also provides a small speed up for lookahead DFA construction (observed empirically).

The key to thread safety in Java while maintaining high throughput lies in avoiding excessive locking (`synchronized` blocks). There are only two data structures that require locking: Q, the set of DFA states, and Δ, the set of edges. Our implementation factors state addition, $Q \mathrel{+}= D'$, into an *addDFAState* function that waits on a lock for Q before testing a DFA state for membership or adding a state. This is not a bottleneck as DFA simulation can proceed during DFA construction without locking since it traverses edges to visit existing DFA states without examining Q.

Adding DFA edges to an existing state requires fine-grained locking, but only on that specific DFA state as our implementation maintains an edge array for each DFA state. We allow multiple readers but a single writer. A lock on testing the edges is unnecessary even if another thread is racing to set that edge. If edge $D \xrightarrow{a} D'$ exists, the simulation simply transitions to D'. If simulation does not find an existing edge, it launches ATN simulation starting from D to compute D' and then sets element $edge[a]$ for D. Two threads could find a missing edge on a and both launch ATN simulation, racing to add $D \xrightarrow{a} D'$. D' would be the same in either case so there is no hazard as long as that specific edge array is updated safely using synchronization. To encounter a contested lock, two or more ATN simulation threads must try to add an edge to the same DFA state.

C. Left-recursion elimination

ANTLR supports directly left-recursive rules by rewriting them to a non-left-recursive version that also removes any ambiguities. For example, the natural grammar for describing arithmetic expression syntax is one of the most common (ambiguous) left-recursive rules. The following grammar supports simple modulo and additive expressions.

$$E \rightarrow E\%E \mid E{+}E \mid \mathbf{id}$$

E is *directly* left-recursive because at least one production begins with E ($\exists\, E \Rightarrow E\alpha$), which is a problem for top-down parsers.

Grammars meant for top-down parsers must use a cumbersome non-left-recursive equivalent instead that has a separate nonterminal for each level of operator precedence:

$E' \rightarrow M\ (+\ M)^*$	Additive, lower precedence
$M \rightarrow P\ (\%\ P)^*$	Modulo, higher precedence
$P \rightarrow \mathbf{id}$	Primary (**id** means identifier)

The deeper the rule invocation, the higher the precedence. At parse-time, matching a single identifier, `a`, requires l rule invocations for l precedence levels.

E is easier to read than E', but the left-recursive version is ambiguous as there are two interpretations of input `a+b+c`: `(a+b)+c` and `a+(b+c)`. Bottom-up parser generators such as bison use operator precedence specifiers (e.g., `%left '%'`) to resolve such ambiguities. The non-left-recursive grammar E' is unambiguous because it implicitly encodes precedence rules according to the nonterminal nesting depth.

Ideally, a parser generator would support left-recursion and provide a way to resolve ambiguities implicitly with the grammar itself without resorting to external precedence specifiers. ANTLR does this by rewriting nonterminals with direct left-recursion and inserting semantic predicates to resolve ambiguities according to the order of productions. The rewriting process leads to generated parsers that mimic Clarke's [5] technique.

We chose to eliminate just direct left-recursion because general left-recursion elimination can result in transformed grammars orders of magnitude larger than the original [11] and yields parse trees only loosely related to those of the original grammar. ANTLR automatically constructs parse trees appropriate for the original left-recursive grammar so the programmer is unaware of the internal restructuring. Direct left-recursion also covers the most common grammar cases (from long experience building grammars). This discussion focuses on grammars for arithmetic expressions, but the transformation rules work just as well for other left-recursive constructs such as C declarators: $D \rightarrow *\ D$, $D \rightarrow D\ [\,]$, $D \rightarrow D\ (\,)$, $D \rightarrow \mathbf{id}$.

Eliminating direct left-recursion without concern for ambiguity is straightforward [11]. Let $A \rightarrow \alpha_j$ for $j = 1..s$ be the non-left-recursive productions and $A \rightarrow A\beta_k$ for $k = 1..r$ be the directly left-recursive productions where $\alpha_j, \beta_k \not\Rightarrow^* \epsilon$. Replace those productions with:

$$A \rightarrow \alpha_1 A' | ... | \alpha_s A'$$
$$A' \rightarrow \beta_1 A' | ... | \beta_r A' | \epsilon$$

The transformation is easier to see using EBNF:

$$A \rightarrow A' A''^*$$
$$A' \rightarrow \alpha_1 | ... | \alpha_s$$
$$A'' \rightarrow \beta_1 | ... | \beta_r$$

or just $A \rightarrow (\alpha_1|...|\alpha_s)(\beta_1|...|\beta_r)^*$. For example, the left-recursive E rule becomes:

$$E \rightarrow \mathbf{id}\ (\%\ E \,|\, +\ E)^*$$

This non-left-recursive version is still ambiguous because there are two derivations for `a+b+c`. The default ambiguity resolution policy chooses to match input as soon as possible, resulting in interpretation `(a+b)+c`.

The difference in associativity does not matter for expressions using a single operator, but expressions with a mixture of operators must associate operands and operators according to operator precedence. For example, the parser must recognize `a%b+c` as `(a%b)+c` not `a%(b+c)`.The two interpretations are shown in Figure 14.

To choose the appropriate interpretation, the generated parser must compare the previous operator's precedence to the current operator's precedence in the $(\%\ E \,|\, +\ E)^*$ "loop." In Figure 14, $\underline{E}$ is the critical expansion of E. It must match just **id** and return immediately, allowing the invoking E to match the $+$ to form the parse tree in **(a)** as opposed to **(b)**.

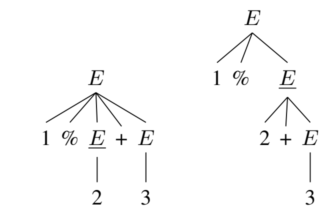
To support such comparisons, productions get precedence numbers that are the reverse of production numbers. The precedence of the i^{th} production is $n - i + 1$ for n original productions of E. That assigns precedence 3 to $E \rightarrow E \,\%\, E$, precedence 2 to $E \rightarrow E + E$, and precedence 1 to $E \rightarrow \mathbf{id}$.

Next, each nested invocation of E needs information about the operator precedence from the invoking E. The simplest mechanism is to pass a precedence parameter, pr, to E and require: *An expansion of* $E[pr]$ *can match only those subexpressions whose precedence meets or exceeds* pr.

To enforce this, the left-recursion elimination procedure inserts predicates into the $(\%\ E \,|\, +\ E)^*$ loop. Here is the transformed unambiguous and non-left-recursive rule:

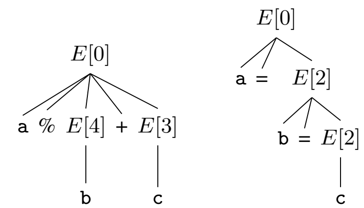
$$E[pr] \rightarrow \mathbf{id}\ (\{3 \geq pr\}?\ \%E[4] \,|\, \{2 \geq pr\}? + E[3])^*$$

References to E elsewhere in the grammar become $E[0]$; e.g., $S \rightarrow E$ becomes $S \rightarrow E[0]$. Input `a%b+c` yields the parse tree for $E[0]$ shown in **(a)** of Figure 15.



(a) `(a%b)+c` **(b)** `a%(b+c)`

Figure 14. Parse trees for `a%b+c` and $E \rightarrow \mathbf{id}\ (\%\ E | +\ E)^*$



(a) `a%b+c` assoc `(a%b)+c` **(b)** `a=b=c` assoc `a=(b=c)`

Figure 15. Nonterminal Expansion Trees for nonterminal $E[pr] \rightarrow \mathbf{id}\ (\{3 \geq pr\}?\ \%E[4] \,|\, \{2 \geq pr\}? + E[3])^*$

Production "$\{3 \geq pr\}?\ \%E[4]$" is viable when the precedence of the modulo operation, 3, meets or exceeds parameter pr. The first invocation of E has $pr = 0$ and, since $3 \geq 0$, the parser expands "$\%\ E[4]$" in $E[0]$.

When parsing invocation $E[4]$, predicate $\{2 \geq pr\}?$ fails because the precedence of the $+$ operator is too low: $2 \not\geq 4$. Consequently, $E[4]$ does not match the $+$ operator, deferring to the invoking $E[0]$.

A key element of the transformation is the choice of E parameters, $E[4]$ and $E[3]$ in this grammar. For left-associative operators like $\%$ and $+$, the right operand gets one more precedence level than the operator itself. This guarantees that the invocation of E for the right operand matches only operations of higher precedence.

For right-associative operations, the E operand gets the same precedence as the current operator. Here is a variation on the expression grammar that has a right-associative assignment operator instead of the addition operator:

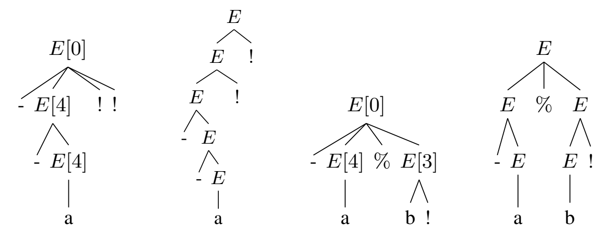
$$E \rightarrow E \,\%\, E \,|\, E\ =^{right}\ E \,|\, \mathbf{id}$$

where notation $=_{right}$ is a shorthand for the actual ANTLR syntax "`|<assoc=right>` $E\ =^{right}\ E$." The interpretation of `a=b=c` should be right associative, `a=(b=c)`. To get that associativity, the transformed rule need differ only in the right operand, $E[2]$ versus $E[3]$:

$$E[\texttt{pr}] \rightarrow \mathbf{id}\ (\{3 \geq \texttt{pr}\}?\ \%\, E[4] \,|\, \{2 \geq \texttt{pr}\}? = E[2])^*$$

The $E[2]$ expansion can match an assignment, as shown in **(b)** of Figure 15, since predicate $2 \geq 2$ is true.

Unary prefix and suffix operators are hardwired as right- and left-associative, respectively. Consider the following E



(a) `--a!!` `((-(-a))!)!` **(b)** `--a!!` **(c)** `-a%b!` `(-a)%(b!)` **(d)** `-a%b!`

Figure 16. Nonterminal call trees and parse trees for productions $E \rightarrow -E \,|\, E\,! \,|\, E\%E \,|\, \mathbf{id}$

with negation prefix and "not" suffix operators.

$$E \rightarrow -E \,|\, E\,! \,|\, E\,\%\,E \,|\, \mathbf{id}$$

Prefix operators are not left recursive and so they go into the first subrule whereas left-recursive suffix operators go into the predicated loop like binary operators:

$$\begin{aligned} E[\texttt{pr}] \rightarrow \quad & (\mathbf{id} \,|\, -E[4]) \\ & (\{3 \geq \texttt{pr}\}?\,! \,|\, \{2 \geq \texttt{pr}\}?\,\%\,E[3])^* \end{aligned}$$

Figure 16 illustrates the rule invocation tree (a record of the call stacks) and associated parse trees resulting from an ANTLR-generated parser. Unary operations in contiguous productions all have the same relative precedence and are, therefore, "evaluated" in the order specified. E.g., $E \rightarrow -E \,|\, +E \,|\, \mathbf{id}$ must interpret `-+a` as `-(+a)` not `+(-a)`.

. Nonconforming left-recursive productions $E \rightarrow E$ or $E \rightarrow \epsilon$ are rewritten without concern for ambiguity using the typical elimination technique.

Because of the need to resolve ambiguities with predicates and compute A parameters,

C.1 Left-Recursion Elimination Rules

To eliminate direct left-recursion in nonterminals and resolve ambiguities, ANTLR looks for the four patterns:

$$\begin{array}{ll} A_i \rightarrow A\alpha_i A & (\textit{binary and ternary operator}) \\ A_i \rightarrow A\alpha_i & (\textit{suffix operator}) \\ A_i \rightarrow \alpha_i A & (\textit{prefix operator}) \\ A_i \rightarrow \alpha_i & (\textit{primary or ``other''}) \end{array}$$

The subscript on productions, A_i, captures the production number in the original grammar when needed. Hidden and indirect left-recursion results in static left-recursion errors from ANTLR. The transformation procedure from G to G' is:

1. Strip away directly left-recursive nonterminal references
2. Collect prefix, primary productions into newly-created A'
3. Collect binary, ternary, and suffix productions into newly-created A''
4. Prefix productions in A'' with precedence-checking semantic predicate $\{pr(i) >= \texttt{pr}\}?$ where $pr(i) = \{n - i + 1\}$
5. Rewrite A references among binary, ternary, and prefix productions as $A[nextpr(i, assoc)]$ where $nextpr(i, assoc) = \{assoc == left\,?\,i+1 : i\}$
6. Rewrite any other A references within any production in P (including A' and A'') as $A[0]$
7. Rewrite the original A rule as $A[\texttt{pr}] \rightarrow A'A''^*$

In practice, ANTLR uses the EBNF form rather than $A'A''^*$.