LL(*): The Foundation of the ANTLR Parser Generator

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

Kathleen Fisher *
Tufts University
kfisher@eecs.tufts.edu

Abstract

Despite the power of Parser Expression Grammars (PEGs) and GLR, parsing is not a solved problem. Adding nondeterminism (parser speculation) to traditional *LL* and *LR* parsers can lead to unexpected parse-time behavior and introduces practical issues with error handling, single-step debugging, and side-effecting embedded grammar actions. This paper introduces the *LL(*)* parsing strategy and an associated grammar analysis algorithm that constructs *LL(*)* parsing decisions from ANTLR grammars. At parse-time, decisions gracefully throttle up from conventional fixed $k \geq 1$ lookahead to arbitrary lookahead and, finally, fail over to backtracking depending on the complexity of the parsing decision and the input symbols. *LL(*)* parsing strength reaches into the context-sensitive languages, in some cases beyond what GLR and PEGs can express. By statically removing as much speculation as possible, *LL(*)* provides the expressivity of PEGs while retaining *LL*'s good error handling and unrestricted grammar actions. Widespread use of ANTLR (over 70,000 downloads/year) shows that it is effective for a wide variety of applications.

Categories and Subject Descriptors F.4.2 Grammars and Other Rewriting Systems [*Parsing*]; D.3.1 Formal Languages [*syntax*]; D.3.4 Processors [*Parsing*]

General Terms Algorithms, Languages, Theory

Keywords nondeterministic parsing, backtracking, context-sensitive parsing, semantic predicates, syntactic predicates, deterministic finite automata, augmented transition networks, subset construction, memoization, PEG, GLR

1. Introduction

Parsing is not a solved problem, despite its importance and long history of academic study. Because it is tedious and error-prone to write parsers by hand, researchers have spent decades studying how to generate efficient parsers from high-level grammars. Despite this effort, parser generators still suffer from problems of expressiveness and usability.

When parsing theory was originally developed, machine resources were scarce, and so parser efficiency was the paramount concern. In that era, it made sense to force programmers to contort their grammars to fit the constraints of *LALR*(1) or *LL*(1) parser generators. In contrast, modern computers are so fast that programmer efficiency is now more important. In response to this development, researchers have developed more powerful, but more costly, nondeterministic parsing strategies following both the "bottom-up" approach (*LR*-style parsing) and the "top-down" approach (*LL*-style parsing).

In the "bottom-up" world, *Generalized LR* (GLR) [19] parsers parse in linear to cubic time, depending on how closely the grammar conforms to classic *LR*. GLR essentially "forks" new subparsers to pursue all possible actions emanating from nondeterministic *LR* states, terminating any subparsers that lead to invalid parses. The result is a parse forest with all possible interpretations of the input. *Elkhound* [12] is a very efficient GLR implementation that achieves *yacc*-like parsing speeds when grammars are *LALR*(1). Programmers unfamiliar with *LALR* parsing theory, though, can easily get nonlinear GLR parsers. Since GLR parser generators do not issue *LR* conflict warnings, programmers can unwittingly specify non-*LALR* grammars that lead to parsers with poor performance.

In the "top-down" world, Ford introduced *Packrat* parsers and the associated *Parser Expression Grammars* (PEGs) [6, 7]. PEGs preclude only the use of left-recursive grammar rules. Packrat parsers are backtracking parsers that attempt the alternative productions in the order specified. The first production that matches at an input position wins. Packrat parsers are linear rather than exponential because they memoize partial results, ensuring input states will never be parsed by the same production more than once. The *Rats!* [8] PEG-based tool vigorously optimizes away memoization events to improve speed and reduce the memory footprint.

A significant advantage of both GLR and PEG parser generators is that they accept any grammar that conforms to their meta-language (with the exception that PEGs cannot be left-recursive). Programmers no longer have to wade through reams of conflict messages. Despite this advantage, neither GLR nor PEG parsers are completely satisfactory, for a number of reasons.

First, GLR and PEG parsers do not always do what was intended. GLR silently accepts *ambiguous grammars*, those that match the same input in multiple ways, forcing programmers to detect ambiguities dynamically. PEGs have no concept of a grammar conflict because they always choose the "first" interpretation, which can lead to unexpected or inconvenient behavior. For example, the second production of PEG rule $A \rightarrow a|ab$ (meaning "A matches either a or ab") will never be used. Input ab never matches the second alternative since the first symbol, a, matches the first alternative. In a large grammar, such hazards are not always obvious and even experienced developers can miss them without exhaustive testing.

Second, debugging nondeterministic parsers can be very difficult. With bottom-up parsing, the state usually represents multiple locations within the grammar, making it difficult for programmers to predict what will happen next. Top-down parsers are easier to understand because there is

* Author worked at AT&T Labs Research when work was done.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'11, June 4–8, 2011, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0663-8/11/06... $10.00

a one-to-one mapping from *LL* grammar elements to parser operations. Further, recursive-descent *LL* implementations allow programmers to use standard source-level debuggers to step through parsers and embedded actions, facilitating understanding. In contrast, there is no easy-to-read counterpart for the state machines derived from *LR* grammars. This advantage is weakened significantly, however, for backtracking recursive-descent packrat parsers. Nested backtracking is very difficult to follow!

Third, generating high-quality error messages in nondeterministic parsers is difficult but very important to commercial developers. GLR parsers pursue all possible paths emanating from *LR* states with conflicts. If no parse attempt succeeds, the parser has to figure out or guess the parse intended by the user to emit a meaningful message. (One way is to pick the failed parse that shifted the most tokens.) Packrat parsers have the same issue since they are always speculating. In fact, they cannot recover from syntax errors because they cannot detect errors until they have seen the entire input. Reducing uncertainty during the parse is the key to recovering well from erroneous input. Deterministic *LL* and *LR* parsers handle erroneous input much better than packrat and GLR parsers.

Finally, nondeterministic parsing strategies cannot easily support arbitrary, embedded grammar actions, which are useful for manipulating symbol tables, constructing data structures, *etc.* Speculating parsers cannot execute side-effecting actions like print statements, since the speculated action may never really take place. Even side-effect free actions such as those that compute rule return values can be awkward in GLR parsers [12]. For example, since the parser can match the same rule in multiple ways, it might have to execute multiple competing actions. (Should it merge all results somehow or just pick one?) GLR and PEG tools address this issue by either disallowing actions, disallowing arbitrary actions, or relying on the programmer to avoid side-effects in actions that could be executed speculatively.

1.1 ANTLR

This paper describes version 3.3 of the ANTLR parser generator and its underlying top-down parsing strategy, called *LL(*)*, that address these deficiencies. The input to ANTLR is a context-free grammar augmented with *syntactic* [17] and *semantic* predicates and embedded actions. Syntactic predicates allow arbitrary lookahead, while semantic predicates allow the state constructed up to the point of a predicate to direct the parse. Syntactic predicates are given as a grammar fragment that must match the following input. Semantic predicates are given as arbitrary Boolean-valued code in the host language of the parser. Actions are written in the host-language of the parser and have access to the current state. As with PEGs, ANTLR requires programmers to avoid left-recursive grammar rules.

The contributions of this paper are 1) the top-down parsing strategy *LL(*)* and 2) the associated static grammar analysis algorithm that constructs *LL(*)* parsing decisions from ANTLR grammars. The key idea behind *LL(*)* parsers is to use regular-expressions rather than a fixed constant or backtracking with a full parser to do lookahead. The analysis tries to construct a deterministic finite automaton (DFA) *for each nonterminal in the grammar* to distinguish between alternative productions. If the analysis cannot find a suitable DFA for a nonterminal, it fails over to backtracking. As a result, *LL(*)* parsers gracefully throttle up from conventional fixed $k \geq 1$ lookahead to arbitrary lookahead and, finally, fail over to backtracking depending on the complexity of the parsing decision. Even within the same parsing decision, the parser decides on a strategy dynamically according to the input sequence. Just because a decision *might* have to scan arbitrarily far ahead or backtrack does not mean that it *will* at parse-time for every input sequence. In practice, *LL(*)* parsers only look one or two tokens ahead on average despite needing to backtrack occasionally (Section 6). *LL(*)* parsers are *LL* parsers with supercharged decision engines.

This design gives ANTLR the advantages of deterministic top-down parsing without the downsides of frequent speculation. In particular, ANTLR accepts all but left-recursive context-free grammars so, as with PEG parsing, programmers do not have to spend a lot of time contorting their grammars to fit the parsing strategy. The lack of left recursion is a disadvantage, though, because some very common constructs, such as arithmetic expressions, are awkward without left recursion.

The next major release of ANTLR will allow rules with immediate left-recursion (self-referential rules) while still supporting the usual tree construction operators and arbitrary actions. Following a technique similar to Hansen [9], our prototype replaces left-recursion with a predicated loop that compares the precedence of the previous and the next operator. For example, here is an expression rule that is *LR* but not *LL* because of the left recursion:

```
e : e '*' e
  | e '+' e
  | INT
  ;
```

Supporting such a grammar is a matter of identifying the operators and rewriting it to the equivalent predicated *LL(*)* grammar:

```
e : e_[0] ;
e_[int p] // pass in precedence of operator(s) to match
  : INT
    ( {p <= 2}? '*' e_[3]
    | {p <= 1}? '+' e_[2]
    )*
  ;
```

The mechanism is sufficiently general to support suffix, prefix, binary, and ternary operators. Operator precedence follows the order of the alternatives, highest to lowest.

Unlike GLR or PEGs, ANTLR can statically identify some grammar ambiguities and dead productions. ANTLR generates top-down, recursive-descent, mostly non-speculating parsers, which means it supports source-level debugging, produces high-quality error messages, and allows programmers to embed arbitrary actions. A survey of 89 ANTLR grammars [1] available from `sourceforge.net` and `code.google.com` reveals that 75% of them had embedded actions, counting conservatively, which reveals that such actions are a useful feature in the ANTLR community.

Widespread use shows that *LL(*)* fits within the programmer comfort zone and is effective for a wide variety of language applications. ANTLR 3.x has been downloaded 41,364 (binary jar file) + 62,086 (integrated into ANTLRworks) + 31,126 (source code) = 134,576 times according to Google Analytics (unique downloads January 9, 2008 - October 28, 2010). Projects using ANTLR include Google App Engine (Python), IBM Tivoli Identity Manager, BEA/Oracle WebLogic, Yahoo! Query Language, Apple XCode IDE, Apple Keynote, Oracle SQL Developer IDE, Sun/Oracle JavaFX language, and NetBeans IDE.

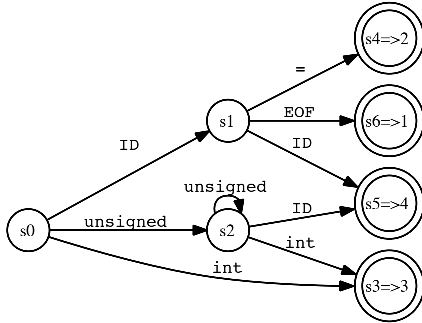


Figure 1. ***LL(*)*** **lookahead DFA for rule s. Notation sn => i means "predict the ith alternative."**

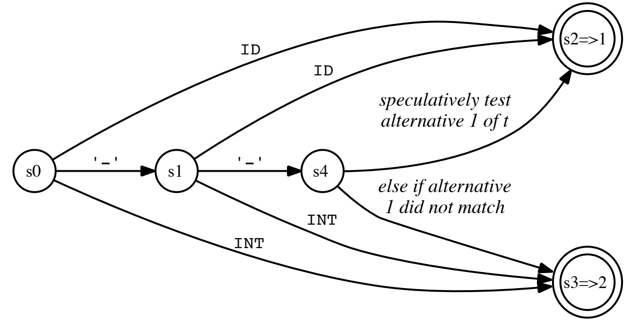


Figure 2. ***LL(*)*** **parsing decision DFA for rule t using mixed $k \leq 3$ lookahead and backtracking**

This paper is organized as follows. We first introduce ANTLR grammars by example (Section 2). Next we formally define *predicated grammars* and a special subclass called *predicated* LL-*regular* grammars (Section 3). We then describe *LL(*)* parsers (Section 4), which implement parsing decisions for predicated *LL*-regular grammars. Next, we give an algorithm that builds lookahead DFA from ANTLR grammars (Section 5). Finally, we support our claims regarding *LL(*)* efficiency and reduced speculation (Section 6).

2. Introduction to *LL(*)*

In this section, we give an intuition for *LL(*)* parsing by explaining how it works for two ANTLR grammar fragments constructed to illustrate the algorithm. Consider nonterminal `s`, which uses the (omitted) nonterminal `expr` to match arithmetic expressions.

```
s : ID
  | ID '=' expr
  | 'unsigned'* 'int' ID
  | 'unsigned'* ID ID
  ;
```

Nonterminal `s` matches an identifier (`ID`), an `ID` followed by an equal sign and then an expression, zero or more occurrences of the literal `unsigned` followed by the literal `int` followed by an `ID`, or zero or more occurrences of `unsigned` followed by two `ID`s. ANTLR grammars use *yacc*-like syntax with extended BNF (EBNF) operators such as Kleene star (`*`) and token literals in single quotes.

When applied to this grammar fragment, ANTLR's grammar analysis yields the *LL(*)* lookahead DFA in Figure 1. At the decision point for `s`, ANTLR runs this DFA on the input until it reaches an accepting state, where it selects the alternative for `s` predicted by the accepting state.

Even though we need arbitrary lookahead to distinguish between the 3rd and 4th alternatives, the lookahead DFA uses the minimum lookahead *per input sequence*. Upon `int` from input `int x`, the DFA immediately predicts the third alternative ($k = 1$). Upon `T` (an `ID`) from `T x`, the DFA needs to see the $k = 2$ token to distinguish alternatives 1, 2, and 4. It is only upon `unsigned` that the DFA needs to scan arbitrarily ahead, looking for a symbol (`int` or `ID`) that distinguishes between alternatives 3 and 4.

The lookahead language for `s` is regular, so we can match it with a DFA. With recursive rules, however, we usually find that the lookahead language is context-free rather than regular. In this case, ANTLR fails over to backtracking if the programmer has requested this feature by adding syntactic predicates. As a convenience, option `backtrack=true` automatically inserts syntactic predicates into every production, which we call *"PEG mode"* because it mimics the behavior of PEG parsers. However, before resorting to backtracking, ANTLR's analysis algorithm builds a DFA that adds a few extra states that allow it avoid backtracking for many input cases. In the following rule `t`, both alternatives can start with an arbitrary number of `-` negation symbols; the second alternative does so using recursive rule `expr`.

```
options {backtrack=true;} // auto-insert syntactic preds
t   : '-'* ID | expr ';' ;
expr: INT     | '-' expr ;
```

Figure 2 shows the lookahead DFA that ANTLR constructs for this input. For clarity, syntactic predicate edges are written in English. This DFA can immediately choose the appropriate alternative upon either input `x` or `1;` by looking at just the first symbol. Upon `-` symbols, the DFA matches a few `-` before failing over to backtracking. The number of times ANTLR unwinds the recursive rule before backtracking is controlled by an internal constant m, which we set to 1 for this example. Despite the possibility of backtracking, the decision will not backtrack in practice unless the input starts with "`--`", an unlikely expression prefix.

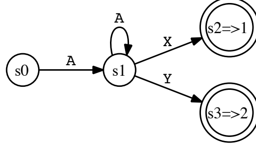
LL parsers need more lookahead in general than *LR* parsers do because *LL* parsers must predict which production will succeed whereas *LR* parsers make decisions after examining entire productions. For example, both grammars shown so far are *LR*(1) but not *LL*(k) for any fixed k, demonstrating that *LL(*)*'s arbitrary lookahead significantly increases *LL*'s recognition strength. In fact, there is no strict ordering between *LR(k)* and *LL(*)*. Nonterminal `a` in the following grammar is *LL(*)* but not *LR(k)* for any k.

```
a : b A+ X             // token vocabulary = {A, X, Y}
  | c A+ Y
  ;
b : ;
c : ;
```

The LPG (2.0.20) *LALR(k)* parser generator [3] demonstrates this by reporting a conflict even at $k = 10000$:

```
t.g:8:1:8:1:51:51: Warning: Grammar is not LALR(10000) -
it contains 1 reduce/reduce conflicts.
```

Lookahead depth $k = 100000$ exposes the exponential space requirements, forcing LPG to core dump after 1.2 minutes on the OS X box used in Section 6. In contrast, ANTLR quickly creates the following cyclic DFA that distinguishes between `a`'s productions. (0.7 seconds to analyze and generate the *LL(*)* parser and DFA.)



3. Predicated Grammars

To describe *LL(*)* parsing precisely, we need to first 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 grammars are written using the notation shown in Figure 3. Productions are numbered to express precedence as a means to resolve ambiguities. The first production form represents a standard context-free grammar rule. The second denotes a production gated by a *syntactic predicate*: symbol A expands to α_i only if the current input also matches the syntax described by A'_i. Syntactic predicates enable arbitrary, programmer-specified, context-free lookahead. The third form denotes a production gated by a *semantic predicate*: symbol A expands to α_i only if the predicate π_i holds for the state constructed so far. The final form denotes an action: applying such a rule updates the state according to mutator μ_i.

The derivation rules in Figure 4 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. To support syntactic predicates, the rules reference w_r, which denotes the input remaining to be matched. The judgment form $(\mathbb{S}, \alpha) \overset{\lambda}{\Rightarrow} (\mathbb{S}', \beta)$, may be read: "In machine state $\mathbb{S}$, grammar sequence α reduces in one step to modified state $\mathbb{S}'$ and grammar sequence β while emitting trace λ." The judgment $(\mathbb{S}, \alpha) \overset{\vec{\lambda}}{\Longrightarrow}{}^{*} (\mathbb{S}', \beta)$ denotes repeated applications of the one-step reduction rule, accumulating all actions in the process. We omit λ when it is irrelevant to the discussion. These reduction rules specify a leftmost derivation. A production with a semantic predicate π_i can fire only if π_i is true of the current state $\mathbb{S}$. A production with syntactic predicate A'_i can fire only if the string derived from A'_i in the current state is a prefix of the remaining input, written $w \preceq w_r$. Actions that occur during the attempt to parse A'_i are executed *speculatively.* They are undone whether or not A'_i matches. Finally, an action production uses the specified mutator μ_i to update the state.

Formally, the language generated by grammar sequence α is $L(\mathbb{S}, \alpha) = \{w \,|\, (\mathbb{S}, \alpha) \Rightarrow^* (\mathbb{S}', w)\}$ and the language of

$A \in N$	Nonterminal
$a \in T$	Terminal
$X \in (N \cup T)$	Grammar symbol
$\alpha, \beta, \delta \in X^*$	Sequence of grammar symbols
$u, x, y, w \in T^*$	Sequence of terminals
$w_r \in T^*$	Remaining input terminals
ϵ	Empty string
$\pi \in \Pi$	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:	
$A \rightarrow \alpha_i$	i^{th} context-free production of A
$A \rightarrow (A'_i)\texttt{=>}\ \alpha_i$	i^{th} production predicated on syntax A'_i
$A \rightarrow \{\pi_i\}\texttt{?}\ \alpha_i$	i^{th} production predicated on semantics
$A \rightarrow \{\mu_i\}$	i^{th} production with mutator

Figure 3. Predicated Grammar Notation

$$\textit{Prod}\ \frac{A \rightarrow \alpha}{(\mathbb{S}, uA\delta) \Rightarrow (\mathbb{S}, u\alpha\delta)} \qquad \textit{Action}\ \frac{A \rightarrow \{\mu\}}{(\mathbb{S}, uA\delta) \overset{\mu}{\Rightarrow} (\mu(\mathbb{S}), u\delta)}$$

$$\textit{Sem}\ \frac{\begin{array}{c}\pi_i(\mathbb{S}) \\ A \rightarrow \{\pi_i\}\texttt{?}\ \alpha_i\end{array}}{(\mathbb{S}, uA\delta) \overset{\pi_i}{\Longrightarrow} (\mathbb{S}, u\alpha_i\delta)} \qquad \textit{Syn}\ \frac{\begin{array}{c}(\mathbb{S}, A'_i) \Rightarrow^* (\mathbb{S}', w) \\ w \preceq w_r \\ A \rightarrow (A'_i)\texttt{=>}\alpha_i\end{array}}{(\mathbb{S}, uA\delta) \overset{A'_i}{\Longrightarrow} (\mathbb{S}, u\alpha_i\delta)}$$

$$\textit{Closure}\ \frac{(\mathbb{S}, \alpha) \overset{\lambda}{\Longrightarrow} (\mathbb{S}, \alpha'),\ (\mathbb{S}, \alpha') \overset{\vec{\lambda}}{\Longrightarrow}{}^{*} (\mathbb{S}, \beta)}{(\mathbb{S}, \alpha) \overset{\lambda\vec{\lambda}}{\Longrightarrow}{}^{*} (\mathbb{S}, \beta)}$$

Figure 4. Predicated Grammar Leftmost Derivation Rules

grammar G is $L(G) = \{w \,|\, (\epsilon, S) \Rightarrow^* (\mathbb{S}, w)\}$. Theoretically, the language class of $L(G)$ is recursively enumerable because each mutator could be a Turing machine. In practice, grammar writers do not use this generality, and so we consider the language class to be the context-sensitive languages instead. The class is context-sensitive rather than context-free because predicates can check both the left and right context.

This formalism has various syntactic restrictions not present in actual ANTLR input, for example, forcing predicates to the left-edge of rules and forcing mutators into their own rules. We can make these restrictions without loss of generality because any grammar in the general form can be translated into this more restricted form [1].

One of the key concepts behind parsing is the language matched by a production at a particular point in the parse.

Definition 1. $\mathcal{C}(\alpha) = \{w \mid (\epsilon, S) \Rightarrow^* (\mathbb{S}, u\alpha\delta) \Rightarrow^* (\mathbb{S}', uw)\}$ *is the* continuation language *for production* α*.*

Finally, grammar position $\alpha \cdot \beta$ means "*after* α *but before* β *during generation or parsing.*"

3.1 Resolving ambiguity

An ambiguous grammar is one in which the same string may be recognized in multiple ways. The rules in Figure 4 do not preclude ambiguity. However, for a practical parser, we want each input to correspond to a unique parse. To that end, ANTLR uses the order of the productions in the grammar to resolve ambiguities, with conflicts resolved in favor of the rule with the lowest production number. Programmers are

instructed to make semantic predicates mutually exclusive for all potentially ambiguous input sequences, making such productions unambiguous. However, that condition cannot be enforced because predicates are written in a Turing-complete language. If the programmer fails to satisfy this condition, ANTLR uses production order to resolve the ambiguity. This policy matches what is done in PEGs [6, 8] and is useful for concisely representing precedence.

3.2 Predicated *LL*-regular grammars

There is one final concept that is helpful in understanding the *LL(*)* parsing framework, namely, the notion of a *predicated* LL*-regular grammar*. In previous work, Jarzabek and Krawczyk [10] and Nijholt [15] define *LL*-regular grammars to be a particular subset of the non-left-recursive, unambiguous CFGs. In this work, we extend the notion of *LL*-regular grammars to predicated *LL*-regular grammars for which we will construct efficient *LL(*)* parsers. We require that the input grammar be non-left-recursive; we use rule ordering to ensure that the grammar is unambiguous.

LL-regular grammars differ from *LL(k)* grammars in that, for any given nonterminal, parsers can use the entire remaining input to differentiate the alternative productions rather than just k symbols. *LL*-regular grammars require the existence of a regular partition of the set of all terminal sequences for *each nonterminal A*. Each block of the partition corresponds to exactly one possible production for A. An *LL*-regular parser determines to which regular set the remaining input belongs and selects the corresponding production. Formally,

Definition 2. *Let $R = (R_1, R_2, \ldots, R_n)$ be a partition of T^* into n nonempty, disjoint sets R_i. If each block R_i is regular, R is a* regular partition. *If $x, y \in R_i$, we write $x \equiv y \,(mod\ R)$.*

Definition 3. *G is predicated* LL*-regular if, for any two alternative productions of every nonterminal A expanding to α_i and α_j, there exists regular partition R such that*

$$(\epsilon, S) \Rightarrow^* (\mathbb{S}, w_i A \delta_i) \Rightarrow (\mathbb{S}, w_i \alpha_i \delta_i) \Rightarrow^* (\mathbb{S}_i, w_i x) \quad (1)$$

$$(\epsilon, S) \Rightarrow^* (\mathbb{S}, w_j A \delta_j) \Rightarrow (\mathbb{S}, w_j \alpha_j \delta_j) \Rightarrow^* (\mathbb{S}_j, w_j y) \quad (2)$$

$$x \equiv y \,(mod\ R) \quad (3)$$

always imply that $\alpha_i = \alpha_j$ and $\mathbb{S}_i = \mathbb{S}_j$.[1]

4. *LL(*)* Parsers

Existing parsers for *LL*-regular grammars, proposed by Nijholt [15] and Poplawski [18], are linear but often impractical because they cannot parse infinite streams such as socket protocols and interactive interpreters. In the first of two passes, these parsers must read the input from right to left.

Instead, we propose a simpler left-to-right, one-pass strategy called *LL(*)* that grafts *lookahead DFA* onto *LL* parsers. A lookahead DFA matches regular partition R associated with a specific nonterminal and has an accept state for each R_i. At a decision point, *LL(*)* parsers expand production i if R_i matches the remaining input. In the worst case, an *LL(*)* parser examines the remaining input at each symbol as it consumes the input. As a result, *LL(*)* parsers are $O(n^2)$, but in practice, they typically examine one or two tokens (Section 6). As with previous parsing strategies, an *LL(*)* parser exists for every *LL*-regular grammar. Unlike previous work, *LL(*)* parsers can take as input a *predicated LL*-regular grammar; they handle predicates by inserting special edges into the *lookahead DFA* that correspond to the predicates.

[1] To be strictly correct, this definition technically corresponds to *Strong LL*-regular, rather than *LL*-regular as Nijholt [15] points out. *Strong LL* parsers ignore left context when making decisions.

$$\frac{p \xrightarrow{a} q}{(\mathbb{S}, p, aw) \mapsto (\mathbb{S}, q, w)} \qquad \frac{\pi_i(\mathbb{S}) \quad p \xrightarrow{\pi_i} f_i}{(\mathbb{S}, p, w) \overset{\pi_i}{\mapsto} (\mathbb{S}, f_i, w)}$$

$$\frac{(\mathbb{S}, f_i, w)}{\text{Accept, predict production } i}$$

Figure 5. Lookahead DFA Configuration Change Rules

Definition 4. Lookahead DFA *are DFA augmented with predicates and accept states that yield predicted production numbers. Formally, given predicated grammar $G = (N, T, P, S, \Pi, \mathcal{M})$, DFA $M = (\mathbb{S}, Q, \Sigma, \Delta, D_0, F)$ where:*

- $\mathbb{S}$ is the system state inherited from surrounding parser
- Q is the set of states
- $\Sigma = T \cup \Pi$ is the edge alphabet
- Δ is the transition function mapping $Q \times \Sigma \to Q$
- $D_0 \in Q$ is the start state
- $F = \{f_1, f_2, \ldots, f_n\}$ is the set of final states, with one $f_i \in Q$ per regular partition block R_i (production i)

A transition in Δ from state p to state q on symbol $a \in \Sigma$ has the form $p \xrightarrow{a} q$. There can be at most one such transition. Predicate transitions, written $p \xrightarrow{\pi} f_i$, must target a final state. There can be multiple predicate transitions emanating from p but each must have a unique predicate π. The instantaneous configuration c of the DFA is $(\mathbb{S}, p, w_r)$ where $\mathbb{S}$ is the system state and p is the current state; the initial configuration is $(\mathbb{S}, D_0, w_r)$. The notation $c \mapsto c'$ means the DFA changes from configuration c to c' using the rules in Figure 5. As with predicated grammars, the rules do not forbid ambiguous DFA paths arising from predicated transitions. In practice, ANTLR tests edges in order to resolve ambiguities.

For efficiency, lookahead DFA match *lookahead sets* rather than continuation languages. Given $R = (\{ac^*\}, \{bd^*\})$, for example, there is no point in looking beyond the first symbol. The lookahead sets are $(\{a\}, \{b\})$.

Definition 5. *Given partition R distinguishing n alternative productions, the* lookahead set *for production i is the minimal-prefix set of R_i that still uniquely predicts i:*

$$LA_i = \{\, w \mid ww' \in R_i, w \notin LA_j \text{ for } j \neq i \text{ and no strict prefix of } w \text{ has the same property}\}$$

4.1 Erasing syntactic predicates

To avoid a separate recognition mechanism for syntactic predicates, we reduce syntactic predicates to semantic predicates that launch speculative parses. To "erase" syntactic predicate `(`A'_i`)=>`, we replace it with semantic predicate `{synpred(`A'_i`)}?`. Function `synpred` returns true if A'_i matches the current input; otherwise it returns false. To support PEG "not predicates," we can flip the result of calling function `synpred`, as suggested by Ford [7].

4.2 Programmer-specified semantic predicates

Most grammars do not need semantic predicates, but when they do, the predicates are usually small expressions that

query a symbol table, check context (e.g., "does this token start in column 1?"), or turn on language constructs (e.g., "should the language allow GCC C extensions?"). They do not represent a significant cost to the programmer. For example, the ANTLR C grammar has just one predicate that tests whether or not an identifier is a type name:

```
type_id : {isTypeName(next input symbol)}? ID ;
```

4.3 Arbitrary actions in predicate grammars

Formal predicated grammars fork new states $\mathbb{S}$ during speculation. In practice, duplicating system state is not feasible. Consequently, ANTLR deactivates mutators during speculation by default, preventing actions from "launching missiles" speculatively. However, some semantic predicates rely on changes made by mutators, such as the symbol table manipulations required to parse C. Avoiding speculation whenever possible attenuates this issue, but still leaves a semantic hazard. To address this issue, ANTLR supports a special kind of action, enclosed in double brackets {{...}}, that executes even during speculation. ANTLR requires the programmer to verify that these actions are either side-effect free or undoable. It is also the responsibility of the programmer to insert actions that undo side-effects. Luckily, symbol table manipulation actions, the most common {{...}} actions, usually get undone automatically. For example, a rule for a `code_block` typically pushes a symbol scope but then pops it on exit. The pop effectively undoes the side-effects that occur during `code_block`.

4.4 Error reporting and arbitrary lookahead

LL(k) parsers report prediction errors by indicating the k-sequence that failed to predict a production of the current nonterminal. For example, given $A \rightarrow ab \,|\, ac$ and input ae, an *LL(k)* parser might report "no viable production upon ae." Because *LL(*)* prediction DFA can scan arbitrarily far ahead, printing the entire erroneous lookahead sequence is impractical. Instead, *LL(*)* parsers should report an error at the specific token that led to a lookahead DFA error state. Given $A \rightarrow a^+b \,|\, a^+c$ and input $aaaaad$, the parser should report that it could not predict a production due to d. Reporting an error at the current symbol (the first a) would be confusing. For prediction errors when backtracking across multiple productions, the parser should report errors at the deepest symbol reached by a failed speculative parse. ANTLR 3.3 does not always conform to this ideal; we plan to address this in a future version.

5. *LL(*)* Grammar Analysis

For *LL(*)*, analyzing a grammar means finding a lookahead DFA for each parsing decision, *i.e.*, for each nonterminal in the grammar with multiple productions. In our discussions, we use A as the nonterminal in question and α_i for $i \in 1..n$ as the corresponding collection of right-hand sides. Our goal is to find for each A a regular partition R, represented by a DFA, that distinguishes between productions. To succeed, A must be *LL*-regular: partition block R_i must contain every sentence in $\mathcal{C}(\alpha_i)$, the continuation language of α_i, and the R_i must be disjoint. The DFA tests the remaining input for membership in each R_i; matching R_i predicts alternative i. For efficiency, the DFA matches lookahead sets instead of partition blocks.

It is important to point out that we are not parsing with the DFA, only predicting which production the parser should expand. The continuation language $\mathcal{C}(\alpha_i)$ is often context-free, not regular, but experience shows there is usually an approximating regular language that distinguishes between the α_i. For example, consider rule $A \rightarrow [\,A\,] \,|\, \mathbf{id}$ that matches balanced brackets around an identifier, *i.e.*, the context-free language $\{[^n \mathbf{id}\,]^n\}$. Approximating the $\mathcal{C}(\alpha_i)$ sets with regular expressions gives a partition that satisfies the *LL*-regular condition: $R = \{\{[^* \mathbf{id}\,]^*\}, \{\mathbf{id}\}\}$. In fact, the first input symbol is sufficient to predict alternatives: $LA = \{\{[\}, \{\mathbf{id}\}\}$. The decision is $LL(1)$.

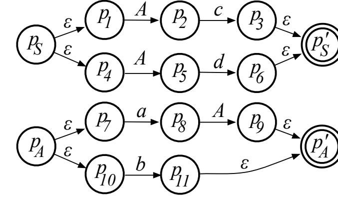


Figure 6. ATN for G with P=$\{S \rightarrow Ac \,|\, Ad, A \rightarrow aA \,|\, b\}$

Not all grammars are *LL*-regular and so our algorithm may fail to find a partition for A. Worse, Poplawski [18] showed that the *LL*-regular condition is undecidable so we must use heuristics to prevent nontermination, sometimes forcing the algorithm to give up before finding R even when A is *LL*-regular. In such cases, we fall back on other strategies, discussed in Sections 5.3 and 5.4, rather than failing to create a DFA.

The *LL(*)* analysis algorithm starts by converting the input grammar to an equivalent *augmented transition network* (ATN) [20]. It then computes lookahead DFA by simulating the actions of the ATN in a process that mimics how the well-known *subset construction* algorithm computes a DFA that simulates the actions of an NFA.

5.1 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:

- 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 $Q \times (\Sigma \cup \epsilon) \rightarrow Q$
- $E = \{p_A \mid A \in N\}$ is the set of submachine entry states
- $F = \{p'_A \mid A \in N\}$ is the set of submachine final states

We describe how to compute Q and Δ shortly.

ATNs resemble the syntax diagrams used to document programming languages, with an ATN submachine for each nonterminal. For example, Figure 6 gives the ATN for a simple grammar. Nonterminal edges $p \xrightarrow{A} p'$ are like function calls. They transfer control of the ATN to A's submachine, pushing return state p' onto a state stack so it can continue from p' after reaching the stop state for A's submachine.

To get an ATN from a grammar, we create a submachine for each nonterminal A as shown in Figure 7. Start state p_A targets $p_{A,i}$ created from the left edge of α_i. The last state created from α_i targets p'_A. The language matched by the ATN is the same as the language of the original grammar.

Grammar analysis is like an inter-procedural flow analysis that statically traces an ATN-like graph representation of a program, discovering all nodes reachable from a top-level call site. The unique configuration of a program for flow purposes is a graph node and the call stack used to reach

Input Grammar Element	Resulting ATN Transitions
$A \to \alpha_i$	$p_A \xrightarrow{\epsilon} p_{A,i} \xrightarrow{\epsilon} \boxed{\alpha_i} \xrightarrow{\epsilon} p'_A$
$A \to \{\pi_i\}?\,\alpha_i$	$p_A \xrightarrow{\epsilon} p_{A,i} \xrightarrow{\pi_i} \boxed{\alpha_i} \xrightarrow{\epsilon} p'_A$
$A \to \{\mu_i\}$	$p_A \xrightarrow{\epsilon} p_{A,i} \xrightarrow{\mu_i} p'_A$
$A \to \epsilon$	$p_A \xrightarrow{\epsilon} p_{A,i} \xrightarrow{\epsilon} p'_A$
$\boxed{\alpha_i} = X_1 X_2 \ldots X_m$ 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

that node. Depending on the type of analysis, it might also track some semantic context such as the parameters from the top-level call site.

Similarly, grammar analysis statically traces paths through the ATN reachable from the "call site" of production α_i, which is the left edge state $p_{A,i}$. The terminal edges collected along a path emanating from $p_{A,i}$ represent a lookahead sequence. Analysis continues until each lookahead sequence is unique to a particular alternative. Analysis also needs to track any semantic predicate π_i from the left edge of α_i in case it is needed to resolve ambiguities. Consequently, an *ATN configuration* is a tuple (p, i, γ, π) with ATN state p, predicted production i, ATN call stack γ, and optional predicate π. We will use the notation $c.p$, $c.i$, $c.\gamma$, and $c.\pi$ to denote projecting the state, alternative, stack, and predicate from configuration c, respectively. Analysis ignores machine storage $\mathbb{S}$ because it is unknown at analysis time.

5.2 Modified subset construction algorithm

For grammar analysis purposes, we modify subset construction to process ATN not NFA configurations. Each DFA state D represents the set of possible configurations the ATN could be in after matching a prefix of the remaining input starting from state $p_{A,i}$. Key modifications include:

- The *closure* operation simulates the push and pop of ATN nonterminal invocations.
- If all the configurations in a newly discovered state predict the same alternative, the analysis does not add the state to the work list; no more lookahead is necessary.
- To resolve ambiguities, the algorithm adds predicate transitions to final states if appropriate predicates exist.

The structure of the algorithm mirrors that of subset construction. It begins by creating DFA start state D_0 and adding it to a work list. Until no work remains, the algorithm adds new DFA states computed by the *move* and *closure* functions, which simulate the transitions of the ATN. We assume that the ATN corresponding to our input grammar G, $M_G = (Q_M, N \cup T \cup \Pi \cup \mathcal{M}, \Delta_M, E_M, F_M)$, and the nonterminal A that we are analyzing are in scope for all the operations of the algorithm.

Function *createDFA*, shown in Algorithm 8, is the entry point: calling $createDFA(p_A)$ constructs the lookahead DFA for A. To create start state D_0, the algorithm adds configuration $(p_{A,i}, i, [], \pi_i)$ for each production $A \to \pi_i\, \alpha_i$ and configuration $(p_{A,i}, i, [], -)$ for each production $A \to \alpha_i$; the symbol "$-$" denotes the absence of a predicate. The core of

Alg. 8: *createDFA*(ATN State p_A) **returns** DFA
$work := []$; $\Delta := \{\}$; $D_0 := \{\}$;
$F := \{f_i \mid f_i :=$ new DFA state, $1 \ldots numAlts(A)\}$;
$Q := F$;
foreach $p_A \xrightarrow{\epsilon} p_{A,i} \in \Delta_M$ **do**
  if $p_A \xrightarrow{\epsilon} p_{A,i} \xrightarrow{\pi_i} p$ **then** $\pi := \pi_i$ **else** $\pi := -$;
  $D_0 \mathrel{+}= closure(D_0, (p_{A,i}, i, [], \pi))$;
end
$work \mathrel{+}= D_0$; $Q \mathrel{+}= D_0$;
$DFA := DFA(-, Q, T \cup \Pi, \Delta, D_0, F)$;
foreach $D \in work$ **do**
  $T_D := \{a \mid p_1 \xrightarrow{a} p_2 \in \Delta_M$ and $(p_1, -, -, -) \in D\}$;
  foreach $a \in T_D$ **do**
    $mv := move(D, a)$;
    $D' := \bigcup_{c \in mv} closure(D, c)$;
    if $D' \notin Q$ **then**
      $resolve(D')$;
      switch $findPredictedAlts(D')$ **do**
        case *Singleton list of j:* $f_j := D'$;
        otherwise $work \mathrel{+}= D'$;
      endsw
      $Q \mathrel{+}= D'$;
    end
    $\Delta \mathrel{+}= D \xrightarrow{a} D'$;
  end
  foreach $c \in D$ *such that wasResolved(c)* **do**
    $\Delta \mathrel{+}= D \xrightarrow{c.\pi} f_{c.i}$;
  end
  $work \mathrel{-}= D$;
end
return *DFA*;

createDFA is a combined *move-closure* operation that creates new DFA states by finding the set of ATN states directly reachable upon each input terminal symbol $a \in T$:

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

and then adding the closure of those configurations. Once the algorithm identifies a new state D', it invokes *resolve* to check for and resolve ambiguities. If all of the configurations in D' predict the same alternative j, then D' is marked as f_j, the accept state for alternative j, and D' is not added to the work list: once the algorithm can uniquely identify which production to predict, there is no point in examining more of the input. This optimization is how the algorithm constructs DFA that match the minimum lookahead sets LA_j instead of the entire remaining input. Next, the algorithm adds an edge from D to D' on terminal a. Finally, for each configuration $c \in D$ with a predicate that resolves an ambiguity, *createDFA* adds a transition predicated on $c.\pi$ from D to the final state for alternative $c.i$. The test *wasResolved* checks whether the *resolve* step marked configuration c as having been resolved by a predicate.

Closure The *LL(*) closure* operation, shown in Algorithm 9, is more complex than the *closure* function from NFA subset construction because of the ATN stack. Nevertheless, the intuition is the same. When called on a configuration c, *closure* recursively finds all ATN states reachable from c's state by traversing all ATN edges that are not terminals, *i.e.*, predicates, nonterminals, and mutators.

Alg. 9: *closure*(DFA State D, $c = (p, i, \gamma, \pi)$) **returns** set *closure*
if $c \in D.busy$ **then return** {}; **else** $D.busy$ += c;
$closure := \{c\}$;
if $p = p'_A$ *(i.e., p is stop state)* **then**
  if $\gamma = p'\gamma'$ **then** $closure$ += $closure(D, (p', i, \gamma', \pi))$;
  else $closure$ += $\bigcup_{\forall p_2 : p_1 \xrightarrow{A} p_2 \in \Delta_M} closure(D, (p_2, i, [], \pi))$;
end
foreach *transition t emanating from ATN state p* **do**
  switch t **do**
    case $p \xrightarrow{A} p'$:
      $depth$:= number of occurrences of p' in γ;
      if $depth = 1$ **then**
        $D.recursiveAlts$ += i;
        if $|D.recursiveAlts| > 1$ **then**
          throw *LikelyNonLLRegularException*;
      end
      if $depth \geq m$, (i.e., *the max recursion depth)* **then**
        mark D to have recursion overflow;
        return *closure*;
      end
      $closure$ += $closure(D, (p_A, i, p'\gamma, \pi))$;
    case $p \xrightarrow{\pi'} q$, $p \xrightarrow{\mu} q$, *or* $p \xrightarrow{\epsilon} q$ *transition:*
      $closure$ += $closure(D, (q, i, \gamma, \pi))$;
  endsw
end
return *closure*;

The call $closure(D, c)$ takes the DFA state D to which c belongs as an additional argument. The function starts by adding the argument configuration to a *busy* list to avoid redundant computation and infinite loops. To simulate ATN nonterminal transition $p \xrightarrow{A} p'$, *closure* duplicates configuration c, pushing return state p' onto its stack. At submachine stop state p'_A, *closure* duplicates c, popping p' from its stack. If *closure* reaches p_A with an empty stack, we have no information statically about which rule invoked A. (This situation only happens when there is a path from $p_{A,i}$ to p'_A with no terminal edges.) In this case we have to assume any production $p_1 \xrightarrow{A} p_2$ in the input grammar might have invoked A, and so *closure* must chase all such states p_2.

If *closure* detects recursive nonterminal invocations (submachines directly or indirectly invoking themselves) in more than one alternative, it terminates DFA construction for A by throwing an exception; Section 5.4 describes our fall back strategy. If *closure* detects recursion deeper than internal constant m, *closure* marks the state parameter D as having *overflowed*. In this case, DFA construction for A continues but *closure* no longer pursues paths derived from c's state and stack. We discuss this situation more in Section 5.3.

DFA State Equivalence The analysis algorithm relies on the notion of equivalence for DFA states:

Definition 6. *Two DFA states are equivalent, $D \equiv D'$, if their configuration sets are equivalent. Two ATN configurations are equivalent, $c \equiv c'$, if the p, i, and π components are equal and their stacks are equivalent. Two stacks are equivalent, $\gamma_1 \equiv \gamma_2$, if they are equal, if at least one is empty, or if one is a suffix of the other.*

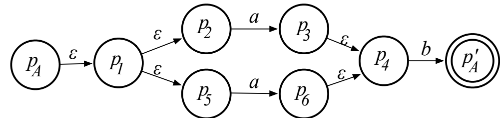
This definition of stack equivalence reflects ATN context information when *closure* encounters a submachine stop state. Because analysis searches the ATN for all possible lookahead sequences, an empty stack is like a wildcard. Any transition $p_1 \xrightarrow{A} p_2$ could have invoked A, so analysis must include the *closure* of every such p_2. Consequently, a *closure* set can have two configurations c and c' with the same ATN state but with $c.\gamma = \epsilon$ and $c'.\gamma \neq \epsilon$. This can only happen when *closure* reaches a nonterminal's submachine stop state with an empty stack and by chasing states following references to that nonterminal, *closure* reenters that submachine. For example, consider the DFA for S in grammar $S \rightarrow a|\epsilon, A \rightarrow SS$. Start state construction computes *closure* at positions $S \rightarrow \cdot a$ and $S \rightarrow \cdot \epsilon$ then $S \rightarrow \epsilon \cdot$, S's stop state. The state stack is empty so *closure* chases states following references to S, such as position $A \rightarrow S \cdot S$. Finally, *closure* reenters S, this time with a nonempty stack.

Equivalence of $\gamma_1 \equiv \gamma_1\gamma_2$ where $\gamma_1, \gamma_2 \neq \epsilon$ degenerates to the previous case of $\gamma_1 = \epsilon$. Given configurations $(p, _, \gamma_1, _)$ and $(p, _, \gamma_1\gamma_2, _)$ in D, *closure* reaches p following the same sequence of most recent submachine invocations, γ_1. Once γ_1 pops off, *closure* has configurations $(p, _, [], _)$ and $(p, _, \gamma_2, _)$.

Resolve One of benefits of static analysis is that it can sometimes detect and warn users about ambiguous nonterminals. After *closure* finishes, the *resolve* function (Algorithm 10) looks for *conflicting configurations* in its argument state D. Such configurations indicate that the ATN can match the same input with more than one production.

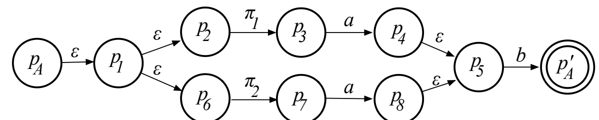
Definition 7. *If DFA state D contains configurations $c = (p, i, \gamma_i, \pi_i)$ and $c' = (p, j, \gamma_j, \pi_j)$ such that $i \neq j$ and $\gamma_i \equiv \gamma_j$, then D is an* ambiguous DFA state *and c and c′ are* conflicting configurations. *The set of all alternative numbers that belong to a conflicting configuration of D is the* conflict set *of D.*

For example, the ATN for subrule $(a|a)$ in $A \rightarrow (a|a)\, b$ merges back together and so analysis reaches the same state from both alternatives with the same (empty) stack context:



$D_0 = \{(p_2, 1), (p_5, 2)\}$, where we abbreviate $(p_2, 1, [], -)$ as $(p_2, 1)$ for clarity. $D_1 = \{(p_3, 1), (p_4, 1), (p_6, 2), (p_4, 2)\}$, reachable with symbol a, has conflicting configurations $(p_4, 1)$ and $(p_4, 2)$. No further lookahead will resolve the ambiguity because ab is in the continuation language of both alternatives.

If *resolve* detects an ambiguity, it calls *resolveWithPredicate* (Algorithm 11) to see if the conflicting configurations have predicates that can resolve the ambiguity. For example, a predicated version of the previous grammar, $A \rightarrow (\{\pi_1\}?\, a \,|\, \{\pi_2\}?\, a)\, b$, yields ATN:



D_0 for decision state p_1 starts with $\{(p_2, 1, [], \pi_1), (p_6, 2, [], \pi_2)\}$ to which we add $\{(p_3, 1, [], \pi_1), (p_7, 2, [], \pi_2)\}$ for *closure*.

Alg. 10: *resolve*(DFA State D)
$conflicts$:= the conflict set of D;
if $|conflicts| = 0$ *and not overflowed(D)* **then return**;
if $resolveWithPreds(D, conflicts)$ **then return**;
resolve by removing all c from D such that $c.i \in conflicts$ and $c.i \neq min(conflicts)$;
if *overflowed(D)* **then** report recursion overflow;
else report grammar ambiguity;

Alg. 11: *resolveWithPreds*(DFA State D, set $conflicts$) **returns** boolean
$pconfigs := []$; // config with predicate for alt i
foreach $i \in conflicts$ **do**
  $pconfigs[i]$:= pick any representative $(_, i, _, \pi) \in D$;
end
if $|pconfigs| < |conflicts|$ **then return false**;
foreach $c \in pconfigs$ **do** mark c as *wasResolved*;
return true;

D_1 has conflicting configurations as before, but now predicates can resolve the issue at runtime with DFA $D_0 \xrightarrow{a} D_1$, $D_1 \xrightarrow{\pi_1} f_1$, $D_1 \xrightarrow{\pi_2} f_2$.

If *resolve* found predicates, it returns without emitting a warning, leaving *createDFA* to incorporate the predicates into the DFA. Without predicates, there is no way to resolve the issue at runtime, so *resolve* statically removes the ambiguity by giving precedence to A's lowest conflicting alternative by removing configurations associated with higher-numbered conflicting alternatives. For example, in the unpredicated grammar for A above, the resulting DFA is $D_0 \xrightarrow{a} f_1$ because the analysis resolves conflicts by removing configurations not associated with highest precedence production 1, leaving $\{(p_3, 1), (p_4, 1)\}$.

If *closure* tripped the recursion overflow alarm, *resolve* may not see conflicting configurations in D, but D might still predict more than one alternative because the analysis terminated early to avoid nontermination. The algorithm can use predicates to resolve the potential ambiguity at runtime, if they exist. If not, the algorithm again resolves in favor of the lowest alternative number and issues a warning to the user.

5.3 Avoiding analysis intractability

Because the *LL*-regular condition is undecidable, we expect a potential infinite loop somewhere in any lookahead DFA construction algorithm. Recursive rules are the source of nontermination. Given configuration $c = (p, i, \gamma)$ (we omit π from configurations for brevity in the following) the *closure* of c at transition $p \xrightarrow{A} p'$ includes $(p_A, i, p'\gamma)$. If *closure* reaches p again, it will include $(p_A, i, p'p'\gamma)$. Ultimately, *closure* will "pump" the recursive rule forever, leading to stack explosion. For example, for the ATN with recursive nonterminal A given in Figure 6, the DFA start state D_0 for $S \to Ac \,|\, Ad$ is:

$$D_0 = \{(p_1, 1, []), (p_A, 1, p_2), (p_7, 1, p_2), (p_{10}, 1, p_2), (p_4, 2, []), (p_A, 2, p_5), (p_7, 2, p_5), (p_{10}, 2, p_5)\}$$

Function $move(D_0, a)$ reaches $(p_8, 1, p_2)$ to create D_1 via $(p_7, 1, p_2)$ in D_0. The *closure* of $(p_8, 1, p_2)$ traverses the implied ϵ edge to p_A, adding three new configurations: $(p_A, 1, p_9p_2)$, $(p_7, 1, p_9p_2)$, $(p_{10}, 1, p_9p_2)$. D_1 has the same configuration as D_0 for p_7 but with a larger stack. The configuration stacks grow forever as $p_9^m p_2$ for recursion depth m, yielding an ever larger DFA path: $D_0 \xrightarrow{a} D_1 \xrightarrow{a} \ldots \xrightarrow{a} D_m$.

There are two solutions to this problem. Either we forget all but the top m states on the stack (as Bermudez and Schimpf [2] do with $LAR(m)$) or simply avoid computing *closure* on configurations with m recursive invocations to any particular submachine start state. We choose the latter because it guarantees a strict superset of *LL(k)* (when $m \geq k$). We do not have to worry about an approximation introducing invalid sequences in common between the alternatives. In contrast, Bermudez and Schimpf give a family of $LALR(1)$ grammars for which there is no fixed m that gives a valid $LAR(m)$ parser for every grammar in the family. Hard-limiting recursion depth is not a serious restriction in practice. Programmers are likely to use (regular) grammar $S \to a^*bc \,|\, a^*bd$ instead of the version in Figure 6.

5.4 Aborting DFA construction

As a heuristic, we terminate DFA construction for nonterminal A upon discovering recursion in more than one alternative. (Analysis for S in Figure 6 would actually terminate before triggering recursion overflow.) Such decisions are extremely unlikely to have exact regular partitions and, since our algorithm does not approximate lookahead, there is no point in pursuing the DFA in vain. ANTLR's implementation falls back on $LL(1)$ lookahead for A, with backtracking or other predicates if *resolve* detects recursion in more than one alternative.

5.5 Hoisting Predicates

This algorithm and the formal predicated grammar semantics in Section 3 require that predicates appear at production left edges. This restriction is cumbersome in practice and can force users to duplicate predicates. The full algorithm in ANTLR automatically discovers and *hoists* all predicates visible to a decision even from productions further down the derivation chain. (See [1] for full algorithm and discussion.) ANTLR's analysis also handles EBNF operators in the right-hand side of productions, *e.g.* $A \to a^*b$ by adding cycles to the ATN.

6. Empirical Results

This paper makes a number of claims about the suitability and efficiency of our analysis algorithm and the *LL(*)* parsing strategy. In this section, we support these claims with measurements obtained by running ANTLR 3.3 on six large, real-world grammars, described in Figure 12, and profiling the resulting parsers on large sample input sentences, described in Figure 13. We include two PEG-derived grammars to show ANTLR can generate valid parsers from PEGs. (Readers can examine the non-commercial grammars, sample input, test methodology, and raw analysis results [1].)

6.1 Static grammar analysis

We claim that our grammar analysis algorithm statically optimizes away backtracking from the vast majority of parsing decisions and does so in a reasonable amount of time. Table 1 summarizes ANTLR's analysis of the six grammars. ANTLR processes each of them in a few seconds except for the 8,231 line-SQL grammar, which takes 13.1 seconds. Analysis times include input grammar parsing, analysis (with EBNF construct processing and predicate hoisting [1]), and parser generation. (As with the exponentially-complex classic subset construction algorithm, ANTLR's analysis can hit

Java1.5: Native ANTLR grammar, uses PEG mode.

RatsC, RatsJava: *Rats!* grammars, manually converted to ANTLR syntax using PEG mode while preserving the essential structure. We removed left-recursion, which *Rats!* supports but ANTLR does not.

VB.NET, TSQL, C#: Commercial grammars for Microsoft languages provided by *Temporal Wave, LLC.*

Figure 12. Benchmark ANTLR grammars

RatsJavaParser.java: Parser generated by ANTLR for RatsJava grammar; tests both Java parsers.

pre_javaParser.c: Preprocessed ANTLR-generated parser for Java1.5 grammar; tests RatsC parser.

LinqToSqlSamples.cs: Microsoft sample code; tests C#;

big.sql: Collected Microsoft SQL samples; tests TSQL.

Northwind.vb: Microsoft sample code; tests VB.NET.

Figure 13. Benchmark input sentence

Grammar	Lines	n	Fixed	Cyclic	Backtrack	Runtime
Java1.5	1,022	170	150	1	20 (11.8%)	3.1s
RatsC	1,174	143	111	0	32 (22.4%)	2.8s
RatsJava	763	87	73	6	8 (9.2%)	3s
VB.NET	3,505	348	332	0	16 (4.6%)	6.75s
TSQL	8,241	1,120	1,053	10	57 (5.1%)	13.1s
C#	3,476	217	189	2	26 (12%)	6.3s

Table 1. Grammar decision characteristics. "Lines" is the size of the grammar, n is the number of parsing decisions in the grammar, "Fixed" is the number of pure *LL(k)* decisions, "Cyclic" is the number of cyclic DFA falling between LL(k)'s acyclic DFA and DFA with syntactic predicate edges, "Backtrack" is the number of decisions that potentially backtrack, and the "runtime" is the time to process a grammar and generate an *LL(*)* parser. Tests performed under OS X with 2x3.2Ghz Quad-Core Intel Xeon with 14G RAM.

a "land-mine" in rare cases; ANTLR provides a means to isolate the offending decisions and manually set their lookahead parameters. None of these grammars hits land-mines.)

All the grammars use backtracking to some extent, providing evidence that it is not worth contorting a large grammar to make it *LL(k)*. The first three grammars use PEG mode, in which ANTLR automatically puts a syntactic predicate on the left edge of every production. Unlike a PEG parser, however, ANTLR can statically avoid backtracking in many cases. For example, ANTLR strips away syntactic predicates from all but 11.8% of the decisions in Java1.5. As expected, the RatsC grammar has the highest ratio of backtracking decisions at 22.4% because C variable and function declarations and definitions look the same from the left edge. The author of the three commercial grammars manually-specified syntactic predicates, which reduced lookahead requirements. In 4 out of 6 grammars, ANTLR was able to construct cyclic DFA to avoid backtracking.

Table 1 shows that the vast majority of decisions in the sample grammars are fixed *LL(k)* for some k. Table 2 reports the number of decisions per lookahead depth, showing that most decisions are in fact $LL(1)$. The table also shows that

Grammar	*LL(k)*	$LL(1)$	Lookahead depth k					
			1	2	3	4	5	6
Java1.5	88.24%	74.71%	127	20	2	1		
RatsC	77.62%	72.03%	103	7	1			
RatsJava	83.91%	73.56%	64	8	1			
VB.NET	95.40%	88.79%	309	18	4	1		
TSQL	94.02%	83.48%	935	78	11	14	9	6
C#	87.10%	78.34%	170	19				

Table 2. Fixed lookahead decision characteristics

Grammar	Input lines	parse-time	n	avg k	back. k	max k
Java1.5	12,416	78ms	111	1.09	3.95	114
RatsC	37,019	771ms	131	1.88	5.87	7,968
RatsJava	12,416	412ms	78	1.85	5.95	1,313
VB.NET	4,649	351ms	166	1.07	3.25	12
TSQL	794	13ms	309	1.08	2.63	20
C#	3,807	524ms	146	1.04	1.60	9

Table 3. Parser decision lookahead depth. n is the number of decision points covered while parsing. "avg k" is the sum of all decision event lookahead depths divided by the number of decision events. "back. k" is the average speculation depth for backtracking decision events only. "max k" is the deepest lookahead depth encountered during the parse. *LL(*)* parsers were run with Java 1.6.0 under OS X with 2x3.2Ghz Quad-Core Intel Xeon with 14G RAM. VB.NET time includes preprocessing. All times include lexing. Times reflect prior on-the-fly compiler warm-up. Parse times for last 3 include tree construction.

ANTLR is able to statically determine k almost all the time, even though this problem is undecidable in general.

6.2 Parser runtime profiling

At runtime, we claim that *LL(*)* parsing decisions use only a few tokens of lookahead on average and parse with reasonable speed. Table 3 shows that for each sample input file, the average lookahead depth per decision event is roughly one token, with PEG-mode parsers requiring almost two. The average parsing speed is 71,035 lines / second for the first three grammars and 27,196 for the other grammars, which have tree-building overhead.

The average lookahead depth for just the backtracking decisions is less than six tokens, highlighting the fact that although backtracking can scan far ahead, usually it does not. For example, a parser might need to backtrack to match a C type specifier, but most of the time such specifiers look like `int` or `int *` and so they can be identified quickly. The grammars not derived from PEGs have much smaller maximum lookahead depths, indicating the authors did some restructuring to take advantage of $LL(k)$ efficiency. The RatsC grammar, in contrast, backtracks across an entire function (looking ahead 7,968 tokens in one decision event) instead of looking ahead just enough to distinguish declarations from definitions, *i.e.*, between `int f();` and `int f() {...}`.

Although we derived two sample grammars from *Rats!*, we did not compare parser execution times nor memory utilization as one might expect. Both ANTLR and *Rats!* are practical parser generators with parsing speed and memory footprints suitable to most applications. *Rats!* is also scannerless, unlike ANTLR, which makes memoization caches hard to compare.

Grammar	Can back.	Did back.	decision events	Back-track	Back. rate
Java1.5	19	16	462,975	2.36%	45.22%
RatsC	30	24	1,343,176	16.85%	65.27%
RatsJava	8	7	628,340	14.07%	74.68%
VB.NET	6	3	109,257	0.46%	20.84%
TSQL	29	19	17,394	3.38%	27.01%
C#	24	19	141,055	3.68%	40.22%

Table 4. Parser decision backtracking behavior. "Can back." is the number of decisions that potentially backtrack. "Did back." is the number of those that did for the sample input. "Backtrack" is the percentage of decision events that backtracked. "Back. rate" is the likelihood that a potentially backtracking decision actually backtracks when triggered.

ANTLR statically removes backtracking from most parser decisions as shown in Table 1. At runtime, ANTLR backtracks even less than predicted by static analysis. For example, statically we find an average of 10.9% of the decisions backtrack for our sample grammars but Table 4 shows that the generated parsers backtrack in only 6.8% of the decision events on average. The non-PEG-derived grammars backtrack only about 2.5% of the time. This is partly because some of the backtracking decisions manage uncommon grammar constructs. For example, there are 1,120 decisions of any kind in the TSQL grammar but the sample input exercises only 309 of them.

Most importantly, just because a decision *can* backtrack, does not mean it will. The last column in Table 4 shows that the potentially backtracking decisions (PBDs) only backtrack about half the time on average across the sample grammars. The commercial VB.NET and TSQL grammars yield PBDs that backtrack in only about 30% of the decision events. Some PBDs never trigger backtracking events. Subtracting the first two columns in Table 4 gives the number of PBDs that avoid backtracking altogether.

We should point out that without memoization, backtracking parsers are exponentially complex in the worst case. This matters for grammars that do a lot of nested backtracking. For example, the RatsC grammar appears not to terminate if we turn off ANTLR memoization support. In constrast, the VB.NET and C# parsers are fine without it. Packrat parsing [6] achieve linear parsing results at the cost of the increased memory for the memoization cache. In the worst case, we need to squirrel away $O(|N| * n)$ decision event results (one for each nonterminal decision at each input position). The less we backtrack, the smaller the cache since ANTLR only memoizes while speculating.

We did not specifically measure the performance of the lookahead DFA portion of ANTLR parsers because the overall performance of such parsers is more than adequate, a claim supported by the performance numbers provided in this section and the experience of developers in the real world. For example, Twitter parses 1 billion queries/day with an ANTLR-generated parser. The speed of the lookahead DFA likely does not contribute significantly to the overall parse time because the average lookahead depth for any decision, backtracking or not, is only one or two tokens (column "avg k" in Table 3).

ANTLR v3's *LL(*)* parsers are about 2.5x faster than ANTLR v2 parsers for the same grammar. For example, parsing the 565k lines of the Java 1.3 library takes 4.9 seconds with ANTLR v2's Java grammar. The same grammar converted to v3 syntax takes only 1.9 seconds. (Neither test warmed up the on-the-fly compiler.) The v2 version needed to backtrack but v3's more powerful *LL(*)* made it unnecessary to backtrack. The Java1.5 grammar from this section, written specifically for v3 and a larger language than Java 1.3, backtracks occasionally and took 3 seconds. Unfortunately, it is difficult to draw conclusions from this specifically about *LL(*)* parsing. The runtime and lexer capabilities of v3 and v2 are so different that we cannot be sure where the speed increase lies.

7. Related work

Many parsing techniques exist, but currently the two dominant strategies are Tomita's bottom-up GLR [19] and Ford's top-down packrat parsing [6], commonly referred to by its associated meta-language PEG [7]. Both are nondeterministic in that parsers can use some form of speculation to make decisions. *LL(*)* is an optimization of packrat parsing just as GLR is an optimization of Earley's algorithm [5]. The closer a grammar conforms to the underlying *LL* or *LR* strategy, the more efficient the parser in time and space. *LL(*)* ranges from $O(n)$ to $O(n^2)$ whereas GLR ranges from $O(n)$ to $O(n^3)$. Surprisingly, the $O(n^2)$ potential comes from cyclic lookahead DFA not backtracking (assuming we memoize). ANTLR generates *LL(*)* parsers that are linear in practice and that greatly reduce speculation, reducing memoization overhead over pure packrat parsers.

GLR and PEG tend be scannerless, which is necessary if a tool needs to support *grammar composition*. Composition means that programmers can easily integrate one language into another or create new grammars by modifying and composing pieces from existing grammars. For example, the *Rats!* Jeannie grammar elegantly composes all of C and Java.

The ideas behind *LL(*)* are rooted in the 1970s. *LL(*)* parsers without predicated lookahead DFA edges implement *LL*-regular grammars, which were introduced by Jarzabek and Krawczyk [10] and Nijholt [15]. Nijholt [15] and Poplawski [18] gave linear two-pass *LL*-regular parsing strategies that had to parse from right-to-left in their first pass, requiring finite input streams (*i.e.*, not sockets or interactive streams). They did not consider semantic predicates.

Milton and Fischer [13] introduced semantic predicates to $LL(1)$ grammars but only allowed one semantic predicate per production to direct the parse and required the user to specify the lookahead set under which the predicates should be evaluated. Parr and Quong [17] introduced syntactic predicates and semantic predicate hoisting, the notion of incorporating semantic predicates from other nonterminals into parsing decisions. They did not provide a formal predicated grammar specification or an algorithm to discover visible predicates. In this paper, we give a formal specification and demonstrate limited predicate discovery during DFA construction. Grimm supports restricted semantic predicates in his PEG-based *Rats!* [8] and arbitrary actions but relies on programmers to avoid side-effects that cannot be undone. Recently, Jim *et al* added semantic predicates to transducers capable of handling all CFGs with Yakker [11].

ANTLR 1.x introduced syntactic predicates as a manually-controlled backtracking mechanism. The ability to specify production precedence came as a welcome, but unanticipated side-effect of the implementation. Ford formalized the notion of ordered alternatives with PEGs. Backtracking in versions of ANTLR prior to 3.x suffered from exponential time complexity without memoization. Ford also solved this

problem by introducing packrat parsers. ANTLR 3.x users can turn on memoization with an option.

Previous attempts to extend parsing strength added fixed $k > 1$ lookahead to *LL* and *LALR* parsers. Since k-lookahead sets have space $O(|T|^k)$, the challenge was to compress sets to a practical size. Charles [3] represented *LALR(k)* lookahead sets as minimal acyclic DFA with practical space costs on average. Parr [16] used a lossy compression called *linear approximate lookahead* that reduced space to $O(|T| \times k)$ and, in practice, worked for almost all decisions. (ANTLR 2.x used this compression.) The key difference between these techniques and *LL(*)* is that *LL(*)* lookahead DFA can be cyclic, thus, significantly extending parsing strength with arbitrary lookahead.

LL-regular grammars are the analog of *LR*-regular grammars [4]. Bermudez and Schimpf [2] provided a parsing strategy for *LR*-regular grammars called $LAR(m)$. Parameter m is a stack governor, similar to ours, that prevents analysis algorithm nontermination. $LAR(m)$ builds an $LR(0)$ machine and grafts on lookahead DFA to handle nondeterministic states. Finding a regular partition for every *LL*-regular or *LR*-regular parsing decision is undecidable. Analysis algorithms for *LL(*)* and $LAR(m)$ that terminate construct a valid DFA for a subset of the *LL*-regular or *LR*-regular grammar decisions, respectively. In the natural language community, Nederhof [14] uses DFA to approximate entire CFGs, presumably to get quicker but less accurate language membership checks. Nederhof inlines rule invocations to a specific depth, m, effectively mimicking the constant from $LAR(m)$.

8. Conclusion

LL(*) parsers are as expressive as PEGs and beyond because of semantic predicates. While GLR accepts left-recursive grammars, it cannot recognize context-sensitive languages as *LL(*)* and other predicated parsers can. Unlike PEGs or GLR, *LL(*)* parsers enable arbitrary action execution and provide good support for debugging and error handling. The *LL(*)* analysis algorithm constructs cyclic lookahead DFA to handle non-*LL(k)* constructs and then fails over to backtracking via syntactic predicates when it fails to find a suitable DFA. Experiments reveal that ANTLR generates efficient parsers, eliminating almost all backtracking. ANTLR is widely used in practice, indicating *LL(*)* hits a sweet spot in the parsing spectrum.

9. Acknowledgements

We would like to thank the anonymous reviewers for their detailed comments and to thank Sriram Srinivasan for discussing *LL(*)* parsing and grammar analysis at length.

References

[1] Appendix. `http://antlr.org/papers/LL-star/index.html`.

[2] Bermudez, M. E., and Schimpf, K. M. Practical arbitrary lookahead LR parsing. *Journal of Computer and System Sciences 41*, 2 (1990), 230–250.

[3] 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.

[4] Cohen, R., and Culik, K. LR-Regular grammars an extension of LR(k) grammars. In *SWAT '71: Proceedings of the 12th Annual Symposium on Switching and Automata Theory (swat 1971)* (Washington, DC, USA, 1971), IEEE Computer Society, pp. 153–165.

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

[6] Ford, B. Packrat Parsing: Simple, powerful, lazy, linear time. In *Proceedings of annual ACM SIGPLAN International Conference on Functional Programming* (2002), ACM Press, pp. 36–47.

[7] Ford, B. Parsing Expression Grammars: A recognition-based syntactic foundation. In *POPL '04: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages* (2004), ACM Press, pp. 111–122.

[8] Grimm, R. Better extensibility through modular syntax. In *PLDI'06: Proceedings of annual ACM SIGPLAN Conference on Programming Language Design and Implementation* (2006), ACM Press, pp. 38–51.

[9] Hanson, D. R. Compact recursive-descent parsing of expressions. *Software Practice and Experience 15* (December 1985), 1205–1212.

[10] Jarzabek, S., and Krawczyk, T. LL-Regular grammars. *Information Processing Letters 4*, 2 (1975), 31 – 37.

[11] Jim, T., Mandelbaum, Y., and Walker, D. Semantics and algorithms for data-dependent grammars. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages* (New York, NY, USA, 2010), ACM, pp. 417–430.

[12] McPeak, S., and Necula, G. C. Elkhound: A fast, practical GLR parser generator. In *Compiler Construction* (2004), pp. 73–88.

[13] Milton, D. R., and Fischer, C. N. LL(k) parsing for attributed grammars. In *International Conference on Automata, Languages, and Programming* (1979), pp. 422–430.

[14] Nederhof, M.-J. Practical experiments with regular approximation of context-free languages. *Computational Linguistics 26*, 1 (2000), 17–44.

[15] Nijholt, A. On the parsing of LL-Regular grammars. In *Mathematical Foundations of Computer Science 1976* (Heidelberg, 1976), A. Mazurkiewicz, Ed., vol. 45 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 446–452.

[16] 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.

[17] Parr, T. J., and Quong, R. W. Adding Semantic and Syntactic Predicates to $LL(k)$—*pred-LL(k)*. In *Proceedings of the International Conference on Compiler Construction; Edinburgh, Scotland* (April 1994).

[18] Poplawski, D. A. On LL-Regular grammars. *Journal of Computer and System Sciences 18*, 3 (1979), 218 – 227.

[19] Tomita, M. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, 1986.

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