ANTLRWorks: An ANTLR Grammar Development Environment
UNPUBLISHED DRAFT

Jean Bovet, jbovet at bea.com, BEA Systems, Inc.

Terence Parr, parrt at cs.usfca.edu, University of San Francisco

July 11, 2007

Abstract

Programmers tend to avoid using language tools, resorting to ad-hoc methods, because tools can be hard to use, their parsing strategies can be difficult to understand and debug, and their generated parsers can be opaque black-boxes. In particular, there are two very common difficulties encountered by grammar developers: Understanding why a grammar fragment results in a parser nondeterminism and determining why a generated parser incorrectly interprets an input sentence.

This paper describes ANTLRWorks, a complete development environment for ANTLR grammars that attempts to resolve these difficulties and, in general, make grammar development more accessible to the average programmer. The main components are a grammar editor with refactoring and navigation features, a grammar interpreter, and a domain-specific grammar debugger. ANTLRWorks' primary contributions are a parser nondeterminism visualizer based upon syntax diagrams and a time-traveling debugger that pays special attention to parser decision-making by visualizing lookahead usage and speculative parsing during backtracking.

1 INTRODUCTION

Generative programming and other forms of software automation are becoming more prevalent as applications become larger and more complicated. From scientific computing to web development, programmers are building domain-specific languages, configuration file formats, network protocols and numerous data file formats as well as the traditional programming language compilers and interpreters. Unfortunately, as Klint, Lämmel, and Verhoef point out [1], many of these language-related applications are written entirely by hand without the use of automated language tools such as parser generators, tree walker generators, and other code generators.

Programmers tend to avoid using language tools, resorting to ad-hoc methods, partly because of the raw and low-level interface to these tools. The threat of having to contort grammars to resolve parser nondeterminisms is enough to induce many programmers to build recursive-descent parsers by hand; some readers are familiar with *LALR reduce-reduce* warnings from YACC [2] or *LL* warnings from other parser generators. Programmers commonly resort to hand-built parsers despite the fact that grammars offer a more natural, high-fidelity, robust and maintainable means of encoding a language-related problem.

The ANTLR parser generator [3] attempts to make grammars more accessible to the average programmer by accepting a larger class of grammars than $LL(k)$ and generating recursive-descent parsers that are very similar to what a programmer would build by hand. Still, developing grammars is a nontrivial task. Just as developers use IDEs (integrated development environments) to dramatically improve their productivity, programmers need a sophisticated development environment for building, understanding, and debugging grammars. Unfortunately, most grammar development is done today with a simple text editor.

This paper introduces *ANTLRWorks*, a domain-specific development environment for ANTLR version 3 grammars that we built in order to:

- Address the most common questions and problems encountered by developers, derived from years of personal experience observing the ANTLR mailing list. E.g., "*Why does my grammar yield a nondeterministic parser?*" and "*Why doesn't my parser recognize input correctly?*".
- Make grammars more accessible to the average programmer.
- Speed up development for both experts and beginners.
- Improve maintainability and readability of grammars by providing grammar navigation and refactoring tools.

To achieve these goals, ANTLRWorks provides three main components: a grammar-aware editor, a grammar interpreter, and a grammar debugger.

The main window has two ever-present panes: the rule navigation list on the left and the actual editing pane on the right. Below these panes is usually a syntax diagram view of the current rule. The editor pane shows the grammar with syntax highlighting and supports identifier auto-completion, rule and semantic action folding, rule dependence diagrams, simple text search, "find rule usage..." search, and grammar refactoring operations.

To help resolve parser nondeterminisms, ANTLRWorks highlights nondeterministic paths in the syntax diagram and provides sample input for which the parser cannot uniquely predict a path. ANTLRWorks can also visualize the parser decision-making process (implemented with lookahead state machines) to help figure out which input sequences predict which alternative productions. ANTLRWorks is guaranteed to provide the same parsing information as ANTLR because it uses ANTLR as a library to obtain lookahead and nondeterminism information.

The ANTLR tool library has a built-in interpreter that ANTLRWorks invokes directly to compute and display the parse tree associated with a particular input sequence and start rule—all without requiring a complete grammar and without having to generate code, compile, and run the application incorporating the parser. The experience is similar to programming in an interpreted language such as Python where individual methods can be interactively executed to get immediate feedback about their correctness. Developers tend to test methods, rather than waste time mentally checking the functionality of a method, because it is so quick and easy. Similarly, being able to dynamically test rules as they are written can dramatically reduce development time.

Once a grammar is more-or-less complete and the generated parser has been integrated into a larger application, the grammar interpreter is less useful primarily because it cannot execute embedded semantic actions. ANTLRWorks has a domain-specific debugger that attaches to language applications running natively via a network socket using a custom text-based protocol. Parsers generated by ANTLR with the `-debug` command-line option trigger debugging events that are passed over the socket to ANTLRWorks, which then visually represents the data structures and state of the

parser. Because ANTLRWorks merely sees a stream of events, it can rewind and replay the parse multiple times by re-executing the events without having to restart the actual parser. This domain-specific time-travel debugging mechanism is similar to the more general framework of Bhansali *et al* [4]. The primary advantage of the socket connection, however, is that the debugger can debug parsers generated in any programming language that has a socket library. Because ANTLR can generate parsers in many different target languages, we needed a mechanism capable of supporting more than just Java (ANTLR's implementation language).

The debugger dynamically displays a parser's input stream, parse tree, generated abstract syntax tree (AST), rule invocation stack, and event stream as the user traces through the parser execution. The grammar, input, and tree display panes are always kept in sync so that clicking on, for example, an AST node shows the grammar element that created it and the token within the input stream from which it was created. ANTLRWorks has breakpoints and single-step facilities that allow programmers to stop the parser when it reaches a grammar location of interest or even an input phrase of interest. Sometimes it is useful to jump to a particular event (such as a syntax error) within the parse and then back up to examine the state of the parser before that event. To accommodate this, ANTLRWorks has a "step backwards" facility.

Complex language problems are often broken down into multiple phases with the first phase parsing the input and building an intermediate-form AST. This AST is then passed between multiple tree walkers to glean information or modify the AST. ANTLR accepts tree grammars and can automatically generate tree walkers, again in the form of a recursive-descent parser. The ANTLRworks debugger graphically illustrates the node-by-node construction of ASTs as the parser being debugged constructs these nodes. ASTs grow and shrink as the developer steps forward and backwards in the parse. ANTLR treats tree grammars just like parser grammars except that the input is a tree instead of a flat token sequence. In the ANTLRWorks debugger, programmers can set breakpoints in the input tree and single step through tree grammars to detect errors just like when debugging a token stream parser.

The next (second) section provides an overview of ANTLR syntax and $LL(*)$ parsing concepts required to understand the operation and appreciate the utility of ANTLRWorks. The third section describes the grammar interpreter feature, which is useful for rapid prototyping. The fourth section describes ANTLRWorks' debugger including information on its socket protocol, single stepping and breakpoints, dynamic AST display, and tree parser debugging. The fifth section describes some of the miscellaneous features found in ANTLRWorks. Finally, we discuss related work and then a few of the planned features. This paper is illustrated throughout with screen snapshots from ANTLRWorks.

2 SYNTAX DIAGRAM AND LOOKAHEAD DFA VISUALIZATION

ANTLR is a recursive-descent parser generator that accepts a large class of grammars called $LL(*)$ that can be augmented with semantic and syntactic predicates [5] to resolve parsing nondeterminisms and grammar ambiguities for such complex languages as C++. ANTLRWorks supports the development of grammars by visualizing data structures and processes associated with ANTLR and ANTLR-generated recognizers. This section describes ANTLR's predicated-$LL(*)$ parsing algorithm in sufficient detail to follow the ANTLRWorks discussion and associated visualizations in the remainder of the paper.

2.1 $LL(*)$ and Lookahead DFA

ANTLR's core parsing strategy is called $LL(*)$ and is a natural extension to $LL(k)$. $LL(k)$ parsers make parsing decisions (i.e., distinguishing between alternative productions) using at most k symbols of lookahead. In contrast, $LL(*)$ parsers can scan arbitrarily far ahead. Because $LL(k)$ lookahead is bounded, the lookahead language is regular and can be encoded within an acyclic deterministic finite automaton (DFA) [6]. $LL(*)$ simply allows cycles in the lookahead DFA. Lookahead decisions for $LL(*)$ are no different than $LL(k)$ decisions in that, once an alternative is predicted, LL parsing of that alternative proceeds normally.

$LL(*)$ represents a much larger class of grammars than $LL(k)$ because parsing decisions can see past arbitrarily-long common left-prefixes. For example, consider the following non-$LL(k)$ grammar expressed in ANTLR EBNF notation.

```
grammar T;
def : modifier+ 'int' ID '=' INT ';' // E.g., "public int x=3;"
    | modifier+ 'int' ID ';'         // E.g., "public static int x;"
    ;
modifier
    : 'public'
    | 'static'
    ;
```

Rule names (nonterminals) begin with a lowercase letter and token names (terminals) begin with an uppercase letter. The two alternatives of rule `def` both begin with the same arbitrarily-long prefix: `modifier+` (one-or-more modifiers). Unfortunately, no amount of fixed lookahead can see past this unbounded left-prefix to the distinguishing symbol beyond, "`=`" or "`;`", in order to predict which alternative will succeed. So, rule `def` is not $LL(k)$ and thus would have to be left-factored to produce an $LL(k)$-conformant grammar (in this case $LL(1)$):

```
def : modifier+ 'int' ID ('=' INT)? ';' ;
```

But, this makes the grammar harder to read and left-factoring is not always possible in the presence of programmer-supplied grammar actions. Either grammar is no problem for $LR(k)$ parsers such as those generated by YACC, but there are similar examples that trip up $LR(k)$ parsers (see the final example in this section).

A better solution is to recognize that, while this lookahead language is infinite, it is still regular and so there must be a (cyclic) DFA that recognizes sentences in that lookahead language. ANTLR automatically creates these lookahead DFA and makes them available to ANTLRWorks. Figure 1 is an ANTLRWorks screen shot showing a portion of the grammar as well as the lookahead DFA for the parsing decision in rule `def`. Being able to examine the lookahead DFA for a particular decision is very helpful when debugging grammars. Often it is not clear why a rule parses a certain input sequence improperly. Tracing that input sequence through the lookahead DFA reveals where it splits off down the wrong path, ultimately predicting the wrong alternative (currently ANTLRWorks does not visually step through the lookahead DFA).

The lookahead DFA in Figure 1 predicts alternative one (state s5) or two (state s4) depending on which symbol follows the `ID` (identifier). Notice that the DFA does not encode the entire first production of rule `def` because the $LL(*)$ DFA construction algorithm terminates for a particular decision when the DFA uniquely predicts all alternatives (or when the decision is found to be nondeterministic). For prediction, the DFA can stop examining input at the "`=`" or "`;`" symbol.

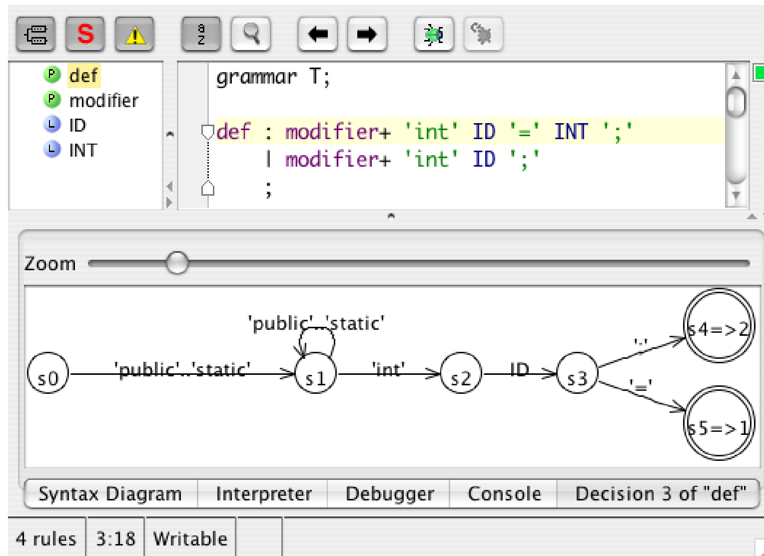


Figure 1: ANTLRWorks Window Showing Lookahead DFA Predictor for Rule `def`

ANTLR's use of a lookahead DFA at a decision point avoids the need for the full parser to backtrack. Running this DFA is much more efficient than running the general recursive-descent parsing process for each alternative for several reasons. First, the DFA lookahead process stops as soon as it reaches a point at which it can decide between the alternatives, and therefore it usually only looks at the first few tokens, even in a large recursive grammar rule. Second, the DFA lookahead process does not execute grammar actions, so there is no danger of executing them more than once or having to undo them.

$LL(*)$ does not approximate the distinguishing lookahead language nor the entire context-free grammar. A grammar is only $LL(*)$ when the lookahead needed to distinguish alternative productions is regular for each decision. This does not mean that the language generated by the grammar itself must be regular, however. The language generated by the following $LL(1)$ rule is not regular, but the lookahead language used to distinguish productions is regular (`'('` or `INT`):

```
e : '(' e ')' // '(' predicts this alternative
  | INT       // INT predicts this alternative
  ;
```

Not all lookahead languages are regular. When the lookahead computation encounters recursion reachable by more than one alternative, for example, the analysis algorithm fails rather than approximating the lookahead. In this case, ANTLR can still generate a parser that uses backtracking to distinguish the alternatives at run-time. In practice, this means that ANTLR can accept any non-left-recursive grammar without complaint (see the following two sections on predicated-$LL(*)$).

ANTLR's DFA construction algorithm proceeds in stages beginning with nondeterministic finite automata (NFA) [6] construction. The meta-language parser converts each grammar into a set of interconnected NFA, one per rule. Token references in the grammar become labeled NFA transitions. Rule references become ϵ-transitions to the start state of the NFA representing the target rule. Emanating from the final state of every rule's NFA are ("FOLLOW") transitions pointing back to states that have ϵ-transitions to that rule. The start state of each rule has ϵ-transitions to the start state of every alternative production in that rule. ANTLRWorks can visualize the NFA as an option,

but the syntax diagram view of the NFA is easier to read as shown in figure 2. In the early ANTLR $LL(k)$ lookahead analysis algorithms [7], these NFA were called *grammar lookahead automata*.

ANTLR's next task is to construct the set of DFA needed by the generated parser, one DFA for each fork in the NFA. A fork represents a parsing decision where each branch corresponds to a production in the grammar. ANTLR must create a DFA that uniquely predicts productions by differentiating accept states with the predicted production number (e.g., states s4 and s5 Figure 1). In addition, the DFA must represent the exact lookahead language rather than an approximation in order to maximize parser strength. The lookahead language is generally a subset of the language generated by the grammar rule and is the minimal set of input sequences that distinguishes between alternative productions.

ANTLR expresses the DFA construction problem as a variant of the classical NFA to DFA subset construction algorithm that uses *reach* and *closure* operations [6]. The classical algorithm uses a simple set of NFA states to represent a DFA state. This set of states encodes the set of possible NFA configurations that the NFA could be in after accepting a particular input sequence. An NFA configuration is just a state number in the classical algorithm, but ANTLR's NFA configurations are tuples: *(NFA state, production, context)*. ANTLR uses the predicted production to split DFA accept states and uses grammatical context to differentiate NFA configurations within a DFA state. The grammatical context is purely a rule invocation stack in its most basic form. NFA closure operations push NFA states as they transition to rule start states. Closure operations that reach a rule's stop NFA state "return" to the state that invoked that rule rather than following all emanating transitions. With this context information, ANTLR pursues only those NFA paths corresponding to valid rule invocation sequences in a grammar; i.e., ANTLR examines the exact lookahead language rather than an approximation. ANTLR needs a stack per NFA state not because the NFA has no stack, but because the DFA conversion algorithm pursues all possible lookahead sequences in parallel.

The closure operation and the entire algorithm terminate due to a finite amount of work like the original subset construction algorithm. Because recursion could force closure operations to build infinitely large context stacks, the closure operation caps context stack size. Decisions where recursion would force infinitely large stacks during closure are not $LL(*)$ anyway, so ANTLR can immediately report an $LL(*)$ nondeterminism in such situations (recursion in more than one alternative production). With finite stacks, there are a finite number of NFA configurations per DFA state and, hence, a finite number of unique DFA states (sets of NFA configurations). A finite number of DFA states leads to a finite amount of work.

ANTLR's $LL(*)$ algorithm is the dual of Bermudez's $LAR(m)$ [8], which uses cyclic lookahead DFA to resolve nondeterministic $LR(0)$ states. The only substantive difference is that Bermudez bounds the $LAR(m)$ algorithm's context stack depth at analysis time, m, whereas ANTLR limits only the recursion depth rather than absolute stack size.

In the natural language realm, Nederhof [9] first converts grammars to recursive transition networks [10] (RTNs) and then to NFA identical to ANTLR's. Nederhof seeks to reduce the complexity of matching natural languages by collapsing a complete context-free grammar to a single DFA. ANTLR, on the other hand, generates a recursive descent parser to match the entire context-free language but needs a set of DFA predictors, one for each grammar decision point.

There is no strict ordering between $LL(*)$ and $LR(k)$ because there is at least one grammar that is $LL(*)$ but not $LR(k)$. For example, the following $LL(*)$ grammar segregates the set of declaration modifiers into two different rules in an effort to be more strict syntactically.

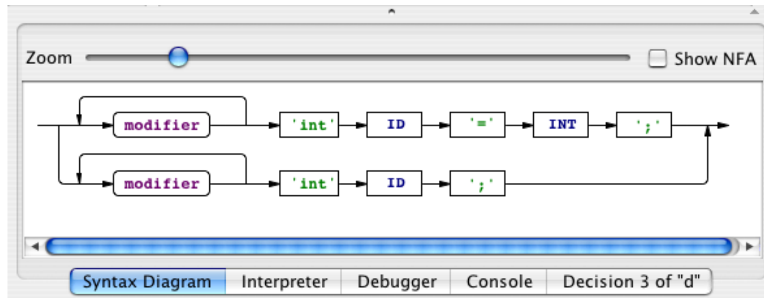


Figure 2: Partial ANTLRWorks Window Showing the Syntax Diagram for Rule `def`

```
// simplified Java declaration rule
decl : variable_modifier+ variable // E.g., "public int i;"
     | function_modifier+ function // E.g., "public int f() {...}"
     ;
```

Because the two modifier rules have input symbols in common, such as `public`, the grammar is not $LR(k)$. The parser has a reduce-reduce conflict between rule `variable_modifier` and `function_modifier` upon seeing any modifier in common. No matter how large k is, the parser will not be able to resolve the conflict by seeing past the arbitrarily-long modifier sequence to the distinguishing symbol beyond (";" or "("). This grammar is, however, $LL(*)$ because the cyclic DFA can see past the modifiers. On the other hand, many left-recursive grammars are $LR(1)$ but not $LL(*)$.

As with all parsing strategies, pure CFGs cannot describe context-sensitive language phrases. Typically these lead to grammar ambiguities, which leads to parser nondeterminisms. Generalized LR (GLR) systems, such as ASF+SDF's Meta-Envionrment [11], deal with this by pursuing all ambiguous paths and then using a semantic action later to pick the correct parse tree from the forest. ANTLR moves that semantic check earlier, into the production prediction phase, by predicating alternatives with boolean expressions.

2.2 Semantic Predicates

Most programming languages are context-sensitive even though we describe them with context-free grammars for parsing efficiency reasons. Context-sensitive constructs are resolved with semantic actions during or after parsing to verify semantic validity. For example, "`x=y;`" only makes sense in the context of a visible variable declaration for `y` (and `x` in languages that require variables to be declared before use). In this case, context affects the validity, but not the meaning of the phrase. Syntax alone is sufficient to determine that the phrase is an assignment statement.

Inescapable context-sensitivity occurs when the proper interpretation of a phrase relies on information about the surrounding context. Because there is no way to specify context in a pure context-free grammar, context-sensitivity results in ambiguous grammars. Ambiguous grammars can generate the same phrase in more than one way because the alternative productions cannot be predicated upon context. An ambiguous context-free grammar then has no deterministic parser (a parser that chooses exactly one phrase interpretation), rendering deterministic parsing tools based on pure context-free grammars ineffective in this case.

ANTLR augments context-free grammars with semantic predicates that can specify the semantic validity of applying a production, thus, providing a context-sensitive parsing mechanism. ANTLR

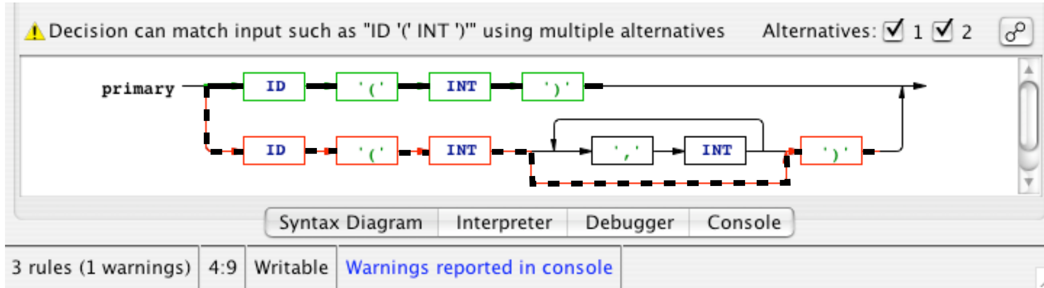


Figure 3: ANTLRWorks Visualizing Rule `primary`'s Nondeterminism

is applicable to languages such as C++ that present a number of challenging parsing problems. One of the most difficult issues involves a syntactic ambiguity that must be resolved with symbol table information. Consider phrase C++ expression "`T(34)`". This phrase can be either a constructor style typecast or a method call depending on whether `T` is a type name or a method name. A C++ grammar rule would have two alternative productions that could match "`T(34)`" as illustrated by the following extremely simplified grammar fragment.

```
primary
    :   ID '(' INT ')'       // ctor-style typecast; E.g., float(3)
    |   ID '(' INT (',' INT)* ')'    // method call; E.g., f(3, 4)
    ;
```

Communicating the cause of parser nondeterminisms to the programmer in a simple and helpful manner is one of the most important problems in parser generator design. Using the command-line interface, ANTLR identifies and reports the issue during grammar analysis:

```
Decision can match input such as "ID '(' INT ')'" using alternatives: 1, 2
As a result, alternative(s) 2 were disabled for that input
```

This message provides a sample input sequence for which the generated parser couldn't choose an alternative and also indicates how ANTLR resolves the nondeterminism. ANTLR resolves all nondeterminism issues statically by generating a DFA predictor that will choose the alternative production specified first in the grammar (unless it can find a predicate; see below).

ANTLRWorks obtains the same nondeterminism information from ANTLR but takes a visual approach, highlighting the various nondeterministic paths in the syntax diagram. If the nondeterminism traverses multiple rules, ANTLRWorks computes and presents the syntax diagrams for only those rules. Figure 3 illustrates the nondeterminism for rule `primary`. ANTLR statically resolves the nondeterminism by predicting the first alternative, identified by the thick line (the path taken by the parser is shown in green when running the tool). The other path viable upon "`ID '(' INT ')'`" is unreachable for that input and is identified here by the dashed thick line (ANTLRWorks shows this in red). ANTLRWorks can also single step through the syntax diagram along any nondeterminism path. This ability clearly identifies which areas of the grammar contribute to the nondeterminism.

Given only context-free grammar rules, the generated parser must pick a single path at any nondeterministic decision point. In this case, the parser will always interpret "`T(34)`" as a constructor style typecast, which is not always correct. The parser needs semantic information to distinguish between the alternatives. If `T` is a type name, the parser should choose the first alternative else

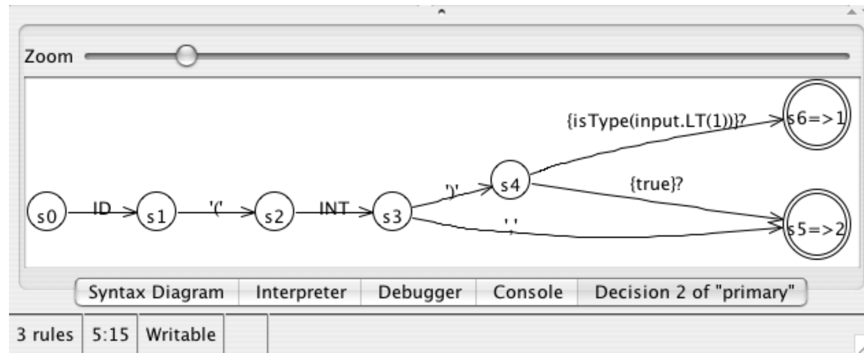


Figure 4: Lookahead DFA for Rule `primary` Illustrating Semantic Predicate Evaluation

it should choose the second. That is straightforward to express using a semantic predicate that examines the first symbol of lookahead:

```
primary
    :   {isType(input.LT(1))}? ID '(' INT ')'
    |                          ID '(' INT (',' INT)* ')'
    ;
```

Semantic predicates are unrestricted boolean expressions enclosed in curly braces followed by a question mark that, in a sense, gate productions in and out dynamically. At run-time, the parser will try the nondeterministic alternatives in the order specified, choosing the first alternative production whose semantic predicate evaluates to true. Productions without semantic predicates implicitly have `{true}?` predicates. Here, expression "`isType(input.LT(1))`" returns true if the first lookahead token is a type as determined by the symbol table. Presumably other actions within the grammar update the symbol table and `isType()` is some user-defined method.

ANTLR incorporates semantic predicates into the prediction DFA as if they were input symbols as shown in Figure 4. Upon input sequence "`T(34)`", the lookahead DFA relies on run-time information (the predicate) to resolve the syntactic ambiguity. At s4, the DFA runs finishes testing the lookahead language input symbols and begins testing the semantic predicate transitions, but only when necessary. Upon input sequences such as "`f(3, 4)`", the DFA predicts the second alternative (method call) without having to evaluate the semantic predicate; the state sequence is s0, s1, s2, s3, s5. The transition from s4 to s5 with predicate label `true` is evaluated if the predicate on transition s4 to s6 fails. The notation `s6=>1` implies that state s6 predicts alternative one.

Because semantic predicates are unrestricted expressions in the target language, they can look at symbols far ahead of the current position (i.e., `input.LT(`i`)`) and can even speculatively parse alternatives in full using a backtracking mechanism. Such semantic predicates manually specify a lookahead language recognizer, but the syntactic predicate mechanism described in the next section offers a more formal approach.

2.3 Syntactic Predicates and Backtracking

One weakness of the $LL(*)$ approach is that a DFA does not have a stack and consequently cannot see past nested language structures; that is, $LL(*)$ cannot see past recursive rule invocations to what

lies beyond. To overcome this limitation, ANTLR supports syntactic predicates that predicate alternative productions with a context-free lookahead language rather than the weaker regular lookahead language used by $LL(*)$. Syntactic predicates not only increase the strength of $LL(*)$ by providing a controlled backtracking mechanism, but they also provide a means of resolving grammar ambiguities.

To illustrate the convenience of syntactic predicates, consider building a grammar for C++. Ellis and Stroustrup [12] point out that some statements cannot be distinguished from declarations without backtracking or some other form of arbitrary lookahead. For example, given type `T`, the following statement looks like a declaration until the `++` symbol is encountered:

```
T(*a)++; // expression statement; cast *a to T then increment
```

Worse, some C++ phrases can be both expressions and declarations. For example, "`T(x)`" is syntactically both a declaration (`x` is an integer as in "`T x;`") and an expression (cast `x` to type `T` as in "`(T)x;`"). The most obvious grammar for the C++ statement rule begins as follows:

```
stat: declaration
    | expression ';'
   ...
    ;
```

There are two problems with this specification. First, distinguishing between declarations and expression statements requires backtracking; rule `stat` is not $LL(*)$ nor is it $LR(k)$. ANTLR would report a nondeterminism between those two alternatives during grammar analysis. Second, the rule is ambiguous because at least one input phrase can be matched by both alternatives. Even if backtracking were not required to distinguish between the alternatives, the grammar ambiguity would result in a parser nondeterminism. The C++ reference guide resolves the ambiguity by giving precedence to `declaration` over `expression` when a phrase is consistent with both. But, how can such precedence be encoded in a grammar?

Backtracking conveniently solves both problems by adding more recognition power and implicitly ordering the alternatives within a decision. The following version of rule `stat` overcomes the weakness of $LL(*)$ by simply trying out the alternatives and correctly expresses the precedence by choosing the first alternative that matches.

```
stat: (declaration)=> declaration // if looks like declaration, it is
    | expression ';'              // else its expression
   ...
    ;
```

The `(declaration)=>` notation indicates that the lookahead language for the first alternative is the language generated by the entire `declaration` rule. If the lookahead is not consistent with a declaration, the parser attempts `expression`, the next viable alternative.

Readers familiar with GLR will notice that the original rule without the syntactic predicates is GLR because it is a CFG. GLR would deal with the nondeterminism associated with arbitrary lookahead by pursuing all viable paths with multiple parse stacks in a manner similar to backtracking. GLR resolves the second problem related to ambiguity by simply returning both possible parse trees. A later semantic phase, or parser callbacks on the fly as Elkhound [13] does, needs to choose declaration trees over expression trees when both are available at the same input position. GLR is similar to $LAR(m)$ in that both engage a form of arbitrary lookahead when confronted with

nondeterminisms. $LAR(m)$ engages a DFA to examine the regular lookahead language whereas GLR engages multiple, full parsers to explore viable paths.

Ford [14] formalized the notion of ordered productions and syntactic predicates in LL-based grammars by defining *parser expression grammars* (PEGs). Whereas a generative CFG formally expresses a language in terms of rules to *generate* phrases of the language, a PEG in contrast expresses a language in terms of the way a backtracking recursive-descent parser would *recognize* phrases of the language. Like CFGs, there are multiple alternative methods to parse PEGs, one being simply to transform them directly into backtracking recursive-descent parsers. Ford also introduced *packrat parsing* [15] that reduces the worst-case exponential time complexity of backtracking to linear complexity by memoizing partial parsing results at the cost of a potentially large heap; space is $O(nm)$ for n input symbols and m rules. *Rats!* [16] is an example parser generator implementing linear time PEG parsing in the Java arena that does heavy optimization to reduce heap usage. A TXL [17] specification is similar to a PEG in that TXL backtracks across the alternatives in order (but does not support predicates).

Because PEG parsers generally decide among alternatives through backtracking, programmer-supplied actions must usually be stateless, or else the programmer must somehow manually "undo" any state changes an action makes when the parser backtracks. Without the ability to change state such as updating a symbol table, altering the parse based upon run-time information via semantic predicates is not feasible. The constant in front of the parse-time space complexity can also be much larger than for more traditional top-down parsers that use lookahead to predict alternatives.

ANTLR provides the power of a PEG with the semantic action flexibility and low overhead of a traditional LL-based parser. ANTLR not only supports manual backtracking via syntactic predicates, but supports an automatic backtracking option that tells ANTLR to backtrack at run-time if $LL(*)$ static analysis fails. Consider the following modified version of `stat`.

```
grammar Cpp;
options {backtrack=true;}
...
stat
    : declaration     {unrestricted action in target language}
    | expression ';'  {another unrestricted action in target language}
    | 'while' '(' expression ')' stat
   ...
    ;
```

Because of the `backtrack=true` option, rule `stat` automatically detects the $LL(*)$ nondeterminism between the first two alternatives and backtracks when the lookahead is consistent with a declaration or expression. If, on the other hand, the first symbol of lookahead is `while`, then the parser immediately jumps to the third alternative without backtracking just like an $LL(1)$ parser. ANTLR implements syntactic predicates via semantic predicates whose expressions reference backtracking routines. Since semantic predicates are only evaluated when $LL(*)$ lookahead is insufficient, ANTLR parsers automatically only backtrack when $LL(*)$ is insufficient.

Action execution is also straightforward in ANTLR grammars. If the parser is not backtracking, it simply executes the actions. During backtracking, however, ANTLR skips actions because they cannot be undone in general; each action is surrounded by an "if not backtracking" conditional. Once a production succeeds, the parser rewinds the input and matches that alternative a second time, this time "with feeling" to execute any actions within that production.

ANTLR's predicated-$LL(*)$ strategy is equivalent in power to PEG[1] and can be viewed as an optimization to PEG. $LL(*)$ is strong enough to prevent backtracking in most decisions for common languages. If ANTLR accepts an unpredicated grammar without reporting a nondeterminism, the generated parser runs in a purely predictive fashion. The benefit of using manually-specified syntactic predicates is that, if performance issues arise, the developer knows where to begin optimizing. The auto-backtracking mode is easier to use but masks which decisions are expensive. The efficiency of ANTLR's mechanism over pure packrat parsing is analogous to GLR's efficiency over Earley's [18] algorithm. GLR relies on traditional LR parsing for all but the nondeterministic states. In principle, GLR could be made even more efficient by using Bermudez's $LAR(m)$ rather than LR in order to reduce the number of nondeterministic states.

For completeness, it is worth pointing out the relative syntactic expressiveness of the various parsing strategies discussed in this paper. PEGs are far stronger than $LL(k)$, because of backtracking, and can express all $LR(k)$ languages [14] (though not every $LR(k)$ grammar; e.g., a left-recursive grammar). Pure $LL(*)$ without predicates sits somewhere between $LL(k)$ and PEG; $LL(*)$ uses arbitrary regular lookahead versus the arbitrary context-free lookahead of PEG. Similarly, GLR is strictly stronger than both $LAR(m)$ and unpredicated $LL(*)$ because GLR can handle any context-free language. As mentioned previously, there is no strict ordering between $LR(k)$ and $LL(*)$. While GLR handles all context-free languages, it is not yet known whether a PEG exists for every context-free language. Ford [14] does, however, provide a PEG for the context-sensitive language $\{a^n b^n c^n\}$, suggesting that either there is no strict ordering between the two grammar classes or possibly that PEG is larger than GLR. In practice, PEG, ANTLR's predicated-$LL(*)$, and GLR are sufficiently expressive.

The previous sections described ANTLR's lookahead mechanism and how ANTLRWorks visualizes lookahead DFA, grammar rules (using syntax diagrams), and nondeterminisms. The next sections focus on how ANTLRWorks helps developers to debug grammars by visualizing the $LL(*)$ parsing process.

3 RAPID PROTOTYPING WITH GRAMMAR INTERPRETER

In order to develop a correct grammar quickly, the developer needs the ability to test rules as they are written. As the grammar grows, the developer adds rules to an existing base of tested rules, making it easier to track down parse errors (the errors are most likely in any newly added rules). The developer needs more than just a yes or no answer as to whether or not the input sequences match—they need parse trees, which describe exactly how the grammar matched the input sequences. Ideally, this testing would be done without code generation and target-language compilation in order to provide instantaneous feedback.

ANTLRWorks supports rapid grammar development by using ANTLR's built-in interpreter, thus, providing immediate feedback during development (Meta-Environment [11] [19], TextTransformer [20], and LAUNCHPADS [21] also have interpreters). Figure 5 shows the parse tree associated with matching input "`public static int x=3;`" starting at rule `def` in the grammar from the $LL(*)$ section. Lexical rules were added to define the tokens used by the parser rules; ANTLR supports combined lexer and parser specifications. The developer can manually specify which tokens should

[1]For the most part, ANTLR grammars are only syntactically different from PEGs with the exception of the "not predicate" that ANTLR must simulate with a semantic predicate. The not predicates are primarily used in lexers.

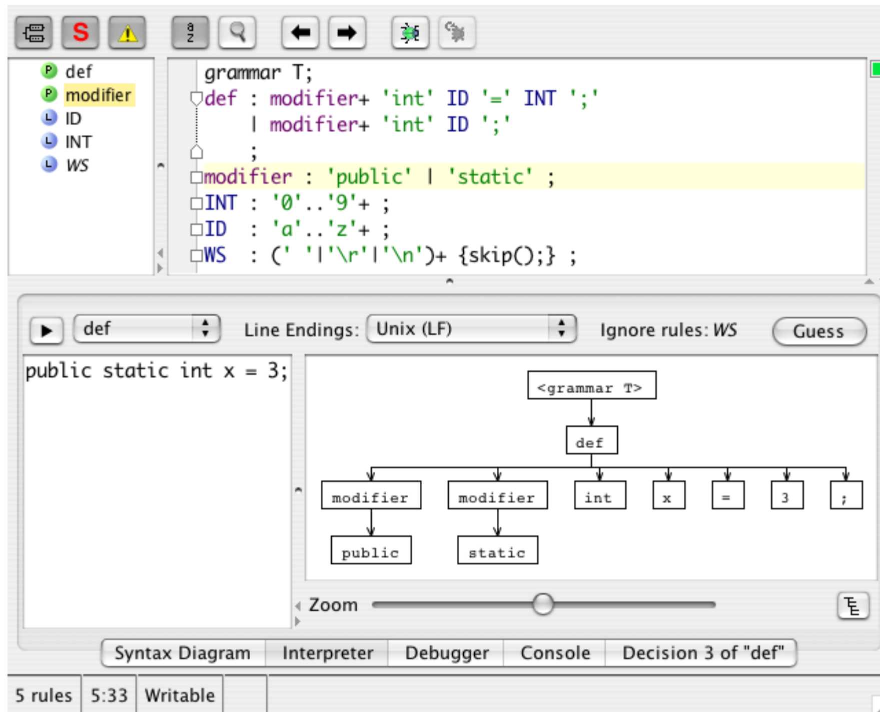


Figure 5: ANTLRWorks Window Showing Lexer/Parser, Input, and Resulting Parse Tree

be ignored by the parser, such as whitespace tokens, but ANTLRWorks can usually guess by looking for actions such as `skip()`.

The interpreter interprets both the lexer and the parser rules by walking the internal NFA grammar representation as if it were a recursive transition network, matching input symbols against the NFA transition labels and using a stack to simulate rule invocations and returns. To ensure that the behavior of the interpreter and generated parsers is identical, the interpreter uses the lookahead DFA as computed by ANTLR in order to choose alternative NFA transitions (predict alternative productions). Here is what the command-line equivalent emits:

```
$ java org.antlr.tool.Interp T.g WS def T-input
(<grammar T> (def (modifier public) (modifier static) int x = 3 ;))
```

where file `T-input` contains input "`public static int x = 3;`". The arguments are the grammar file, whitespace rule, start rule, and input file. The output is in LISP notation and is a serialized version of ANTLRWorks' visual display in Figure 5. For very large trees, ANTLRWorks also provides a hierarchical view, accessed by clicking on the lower right icon shown in Figure 5.

Because any parser rule can be the start symbol, the interpreter can begin in any rule. For example, by selecting rule `modifier` instead of `def` from the drop-down menu shown in Figure 5, that rule can be tested with input "`public`" or "`static`". New input sequences can be tested instantly simply by entering them into the leftmost input pane and hitting the go button (the right-facing triangular button).

If the input sequence is not in the language recognized by the specified start rule, ANTLRWorks inserts an error node into the parse tree to indicate where the problem occurred and what it was. Figure 6 demonstrates the result of interpreting an invalid input sequence for rule `def` (the `INT` is

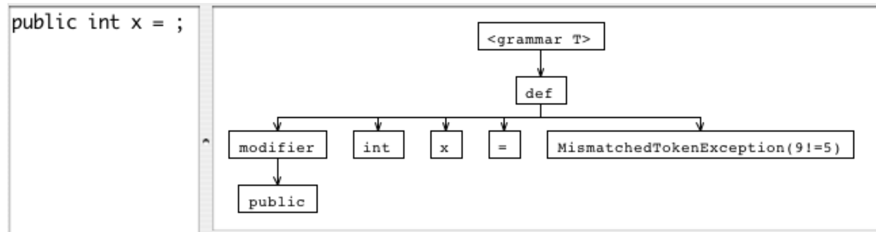


Figure 6: ANTLRWorks Window Showing Parse-Tree Resulting From Invalid Input

missing); the `MismatchedTokenException` node indicates that the interpreter found token type 9 not 5.

The main disadvantage of the interpreter is that it cannot execute actions or semantic predicates (and backtracking is not yet implemented in the interpreter). To execute actions, ANTLRWorks provides a debugger that connects to natively running parsers.

4 GRAMMAR DEBUGGER

Completing a grammar project involves verifying that the resulting parser matches input correctly, detects erroneous input, and builds a proper data structure or emits proper output. By single stepping and using breakpoints, a debugger helps clear up grammar issues but more importantly highlights which user actions are executed and in what order. Tracing through a grammar also helps track down errors in tree construction.

While all of this can be accomplished clumsily using a generic programming language debugger, generic debuggers offer little more than stepping through methods and evaluating raw expressions. ANTLRWorks' debugger, on the other hand, focuses on the higher level, domain-specific data structures and processes associated with language recognition. ANTLRWorks displays input streams, parser lookahead, parse trees, parse stacks, and ASTs as they change during recognition and ensures that all visualizations stay in sync.

This section describes the ANTLRWorks debugger, one of its primary contributions. The debugger is written in Java, but works with parsers written in any target language. It provides a number of interesting capabilities including the ability to back up, which rewinds the input stream, parser state, and partially deconstructs parse trees and ASTs.

4.1 Parse Tree Visualization

To illustrate the core features of the debugger, reconsider the grammar used in the interpreter section above. ANTLRWorks can automatically generate a simple test rig and launch the debugger on a grammar via the debug icon. ANTLRWorks uses ANTLR to generate code for the lexer and parser, compiles them along with the test rig, and then asks for some input text and a start rule via a dialog box. By clicking on the “fast forward” button, the debugger will initiate and race through the entire parsing process as shown in Figure 7. The “fast forward” continues until end of file if there are no breakpoints (the solid cursor is at the end of the start rule because it has finished parsing).

As with the interpreter, ANTLRWorks displays the input stream and the parse tree. The key difference is that the display panes are active: clicking on a node in the parse tree, such as `int`, highlights the corresponding token in the input pane and the token reference in the grammar that

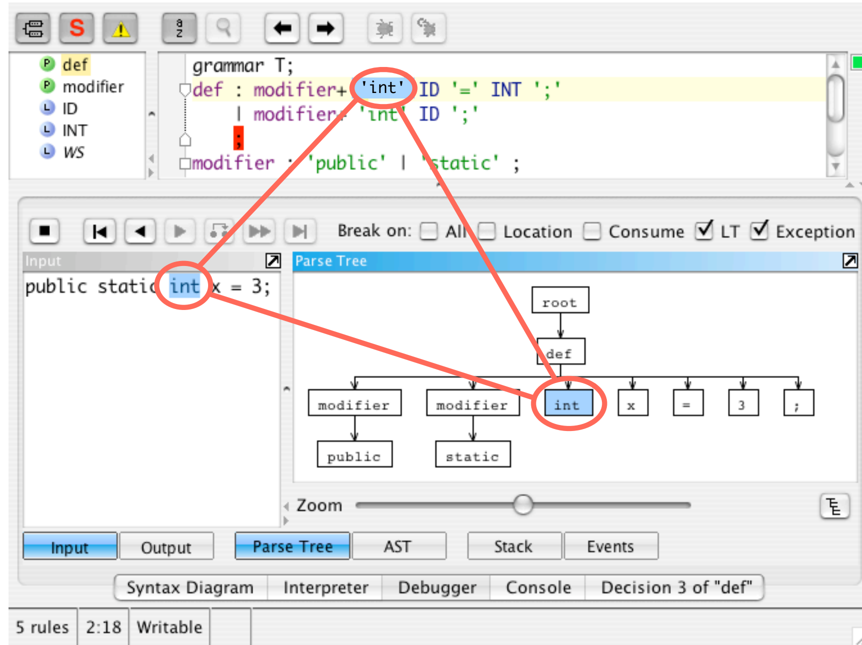


Figure 7: ANTLRWorks Debugger Showing Relationship Between Grammar, Input Stream, and Parse Tree

matched it (Figure 7 identifies the relationship with manually drawn thick lines). Very large parse tree and input stream panes can be detached using the icon in the pane's upper right corner.

4.2 Debugging syntactic predicates

When ANTLR must backtrack to distinguish between alternative productions, it is usually difficult to debug the parser because developers must track when the parser is speculating and when it is not. ANTLRWorks clearly distinguishes between the two modes by showing all speculative parsing branches in the parse tree in red. Consider rule `s` in the following simple grammar that uses a syntactic predicates to distinguish between the two alternatives.

```
grammar B;
s : (e ':')=> e ':' // E.g., "x:", "(x):", "((x)):", ...
  |           e ';' // E.g., "x;", "(x);", "((x));", ...
  ;
e : '(' e ')'
  | ID
  ;
```

Figure 8 shows the parse tree obtained from matching input "`((x));`" starting in rule `s`. The first `e` subtree under the `s` node is highlighted (manually thickened for printing here) to indicate that it was matched only speculatively. The second subtree is the parse tree for the second alternative in rule `s` that matches successfully. In situations where ANTLR must nest the backtrack, ANTLRWorks changes the color through a series of gradations, one for each backtracking nesting level.

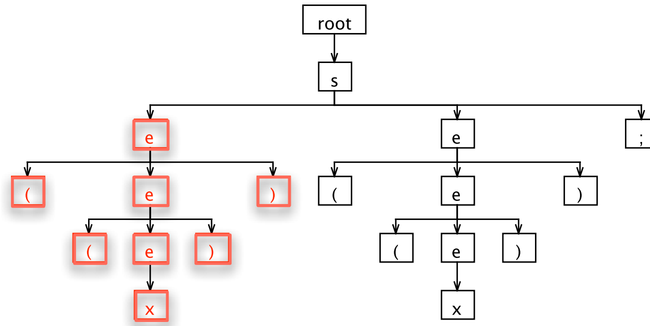


Figure 8: Backtracking Across Two Alternatives of Rule `s` for Input "`((x));`"

4.3 Socket Protocol, Debug Events, and Language-Independent Debugging

When debugging, parsers run natively and in a separate process from ANTLRWorks. They communicate over a socket connection using a simple text-based protocol. The parser acts as the server while ANTLRWorks acts as the client. The socket protocol is sufficiently rich to allow ANTLRWorks to reconstruct all of the relevant data structures and state of the parser. This loosely-coupled mechanism is similar to Meta-Environment's debugger connection except that Meta-Environment is based upon a general remote procedure call mechanism rather than a high-level debug event protocol.

Grammars processed with the ANTLR `-debug` option result in parsers instrumented with debugging event triggers, which amount to method calls to an event listener (in the Java target). Upon the first parser rule invocation, the parser blocks waiting for a client connection from ANTLRWorks. The debugging event method calls are translated directly to sockets events, which ANTLRWorks decodes at the receiving end and retriggers as method calls to a parallel debug event listener. Figure 9 defines the socket protocol for the core debug events. Figure 10 shows the event trace generated by the parser to recognize input "`public`" starting in rule `modifier`. Text strings associated with tokens are terminated by newline, the end of event terminator, not an end quote for simplicity; e.g., see the `LT` and `consumeToken` events in Figure 10.

Using the Java target, the `DebugEventListener` interface identifies the set of events emitted by a running parser and the `DebugEventSocketProxy` forwards events, one per line, over a socket to ANTLRWorks or any other listener (such as a profiling listener). Any client listener can use a `RemoteDebugEventSocketListener` object to convert socket events back into `DebugEventListener` method call events. These events are similar in concept to the parser events generated by Eli's Noosa debugger [22]. Any top-down parsing strategy that can emit these socket events can potentially use ANTLRWorks' debugger.

By using a text protocol, ANTLRWorks avoids byte-ordering issues which can occur on binary channels between different architectures. Unicode characters and other binary data are transmitted using a UTF-8 encoding. Objects are serialized with an obvious but custom mechanism rather than a Java-specific serialization to avoid limiting ANTLRWorks to debugging Java parsers. ANTLRWorks should be able to connect to parsers written in any language with a socket library. Because local and remote sockets behave identically, ANTLRWorks can also tap into running parsing applications even

	Event and arguments
Handshaking	commence (implicit upon connection) `ANTLR` *protocol-version* `grammar` *filename* `terminate`
Grammar navigation	`enterRule` *rulename* `exitRule` *rulename* `enterAlt` *alt-number* `exitAlt` *alt-number* `enterDecision` *decision-number* `exitDecision` *decision-number* `location` *line col*

Prediction, stream control	`LT` *index type channel line col text*
	`consumeToken` *index type channel line col text*
	`consumeHidden` *index type channel line col text*
	`mark` *marker*
	`rewind` *marker*
	`rewind`
	`beginBacktrack`
	`endBacktrack`
	`beginResync`
	`endResync`
	`semanticPredicate` *boolean-result expression-text*

Syntax errors	`exception` *classname input-stream-index line col*

Figure 9: ANTLRWorks Socket Protocol for Core Debug Events

```
ANTLR 1
grammar T.g
enterRule modifier
enterAlt 1
location 5 12
LT 1 0 10 0 1 0 "public
LT 1 0 10 0 1 0 "public
consumeToken 1 0 10 0 1 0 "public
location 5 32
exitRule modifier
terminate
```

Figure 10: Literal Socket Protocol Generated by the Parser for Input "`public`" Starting in Rule `modifier`

if they are on another machine. Traditional target language debuggers can be used in conjunction with ANTLRWorks' debugger to examine program state while viewing the parser state visually.

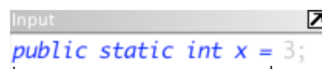
The event stream sent by the parser to the debugger can be saved as a text file at any time so that it may be analyzed later. For example, the programmer may want to compare the event dumps for different inputs or different versions of the grammar to isolate problems in the grammar. The event dumps could also be useful for analyzing grammar coverage and for unit testing.

The socket approach solves one final serious problem: how can ANTLRWorks pause, single step, and stop at breakpoints in a running parser generated in an arbitrary target language running on another machine? And, how can it do this without relying on native debuggers? It turns out that a simple acknowledgment mechanism in the protocol provides sufficient control over the running parser. Debugging events fired across a socket with the `DebugEventSocketProxy` wait for an acknowledgment signal from the remote listener before proceeding. The parser continues when it receives this signal after, for example, the ANTLRWorks user hits the start button again. Until the parser receives an acknowledgment, it is effectively paused or stopped at a breakpoint. In this manner, ANTLRWorks can step with any granularity, such as when it sees a particular event or input token, all without using any native debugging facilities for the various targets.

4.4 Breakpoints and Single Stepping

ANTLRWorks single steps by grammar location by default, which mirrors the code execution and rule method call trace of the running program; the user has the option of stepping over rule invocations. There are also checkboxes to step by input symbol consumption and lookahead symbol examination. For really fine-grained stepping, ANTLRWorks can single step on every event emitted by the parser.

Aside from the path taken through a grammar to match input sequences, developers often want to know how the parser makes the actual decisions to choose from among alternative productions. Because of its fine event stepping granularity, ANTLRWorks can step through the lookahead decision-making process itself. Tokens examined during a single lookahead decision are italicized in the input buffer pane (and highlighted in blue) so that the programmer knows how much of the input is used to make a particular decision. For example, at the start of rule `def`, the lookahead DFA scans ahead until it sees either "`=`" or "`;`" in order to choose between the alternatives. ANTLRWorks highlights more and more the input stream as the user steps forward in the DFA prediction until the DFA reaches the distinguishing symbol:



The input symbols used during prediction have been manually underlined here to make the figure more clear. At this point, the parser rewinds the input stream to the `public` symbol and begins parsing the first alternative.

As with a conventional code debugger, the user can set breakpoints in the grammar to stop the parser when, for example, it reaches a particular rule. For very large input, however, stepping ahead or jumping from breakpoint to breakpoint until an input phrase of interest is infeasible. Often the event of interest is actually a recognition error and the programmer wants to quickly jump to whatever state the parser is in when the input fails to match. There is no corresponding grammar location where a breakpoint should be set—the debugger must be able to break upon recognition exception. Once stopped at this event, the programmer can back up and replay the last few events

of the parser to understand the chain of events that led to this exception (as described in the next section).

Most debuggers are code-centric or in this case grammar-centric. In some circumstances, this is very useful, but most of the time there is a very particular input sequence within a large file that the parser is interpreting improperly. The developer needs to stop the parser at a particular input location rather than at a particular grammar location. Without the ability to stop at a particular input token, the debugger would have to stop many times at a breakpoint before reaching the input location of interest.

It is worth noting that LR-based parsing strategies are at a distinct disadvantage in terms of debugging, ironically because of LR's recognition strength. LR parsers provisionally match multiple productions at once before deciding which production to reduce and, therefore, have trouble moving a traditional debugging cursor through a grammar. LL, on the other hand, always has an exact grammar location, just like code is always executing in an exact program location. An LR debugger could, however, use input token breakpoints just as easily as LL.

4.5 Rewind and Stepping Backwards

When tracking down parse errors or problems related to improperly recognized input sequences, often the most crucial piece of information is what happens right before an error occurs or the parser recognizes a particular input sequence. To aid in this situation, ANTLRWorks' debugger can "time travel" by stepping backwards in the execution trace once stopped at a breakpoint or when single stepping forwards. The complete state of the parser including tree construction is unwound upon each step backwards. ANTLRWorks can also rewind the entire event stream to restart the debugging session.

User-defined actions in the target language are, of course, executed only once during the first run of the debugger. They are executed by the natively running parser in a different process. If the parser process terminates for any reason, the debugger can still navigate forwards or backwards through the event stream. This ability is particularly useful if the parser crashes or terminates early.

4.6 AST Visualization

Translators are often broken into multiple phases out of necessity because of symbol resolution issues or purely for simplicity reasons. The first phase is a parser that builds ASTs (and possibly other data structures such as a symbol table). When the grammar produces an incorrect AST, it can be difficult to track down exactly where in the grammar the erroneous subtree is created. ANTLRWorks visually displays ASTs as they are built so that the developer can single step through the grammar at the appropriate input position to discover why the improper tree is being created. The developer can view trees as two-dimensional graphs or as hierarchal lists, which are sometimes easier to visualize for very large inputs. Finally, as with the parse trees, ANTLRWorks shows the relationship between AST nodes, the input stream, and the grammar.

ANTLR builds ASTs when the `output` option is set to `AST`; each rule implicitly returns an AST. By default, the parser yields a flat list of nodes whose "payload" pointers point at the associated tokens created by the lexer. ANTLR's primary AST construction facility is based upon grammar rewrite rules that map parser grammar productions to output tree grammar productions using the `->` operator. For example, the following rewrites added to rule `def` return trees with `int` at the root and the identifier as the first child; the first alternative's rewrite includes the initialization value as the second child.

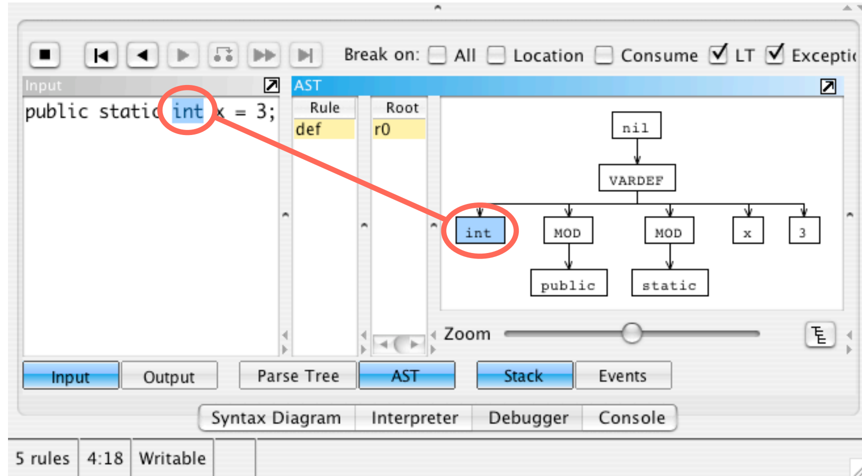


Figure 11: ANTLRWorks Debugger Showing AST for Rule `def` and Input-AST Relationship

```
grammar T;
options {output=AST;}
def : modifier+ 'int' ID '=' INT ';' -> ^('int' ID INT)
    | modifier+ 'int' ID ';'          -> ^('int' ID)
    ;
modifier : 'public' | 'static' ;
```

Rule `modifier` has no rewrite specifications, but it builds the obvious single node from the single input token automatically. The `^(...)`tree specifiers in the rewrites follow a LISP-like syntax where the first element is the root and the following elements are children. The rewrite rules can be any valid production and, hence, may also contain repeated elements. For example, the following rewrites include the modifier symbols in the AST:

```
def : modifier+ 'int' ID '=' INT ';' -> ^('int' modifier+ ID INT)
    | modifier+ 'int' ID ';'          -> ^('int' modifier+ ID)
    ;
modifier : 'public' | 'static' ;
```

In order to structure ASTs so that all subtree roots are operators and children are operands, imaginary nodes (nodes for which there is no corresponding input) are often used as abstract operators. The following grammar version adds two imaginary token definitions and places the modifiers in their own subtrees as shown in Figure 11.

```
grammar T;
options {output=AST;}
tokens {VARDEF; MOD;} // imaginary token types
def : modifier+ 'int' ID '=' INT ';'
      -> ^(VARDEF 'int' ( ^(MOD modifier) )+ ID INT)
    | modifier+ 'int' ID ';'
      -> ^(VARDEF 'int' ( ^(MOD modifier) )+ ID)
    ;
```

	Event and arguments
AST parsing	`consumeNode` *unique-ID type text* `LN` *stream-index type channel line col text*
AST construction	`nilNode` *unique-ID* `createNodeFromToken` *unique-ID token text* `createNode` *unique-ID token-stream-index* `becomeRoot` *new-root-ID old-root-ID* `addChild` *root-ID child-ID* `setTokenBoundaries` *unique-ID start-index stop-index*

Figure 12: ANTLRWorks Socket Protocol for AST-Related Debug Events

By suffixing the subrule around the nested tree specifier with the `+` closure operator rather than the `modifier` rule, a series of trees is produced with `MOD` at the root and a single modifier symbol as child. If "`^(MOD modifier+)`" were used instead, a single `MOD`-rooted tree with multiple modifier symbols as children would result.

In order to accommodate AST construction, the debugging protocol supports six extra events as summarized in the bottom half of Figure 12. One of the challenges faced by ANTLRWorks when trying to display trees as they are built is that ANTLRWorks does not have pointers into the memory space of the natively running parser process. Building trees incrementally means that ANTLRWorks must be able to uniquely identify nodes added previously to the tree in order to hang new nodes. We settled on having the parser assign a unique identifier (like a "foreign" pointer) to each node that it could pass to the debugger over the socket. In this way, the parser can send instructions about the relationship between nodes. For example, the following event sequence creates a nil node with identifier 5395534 as an initial root, creates a node with identifier 1892095 from the token at index 0 in the token stream, and then makes that node the child of the nil root node.

```
nilNode 5395534
createNode 1892095 0
addChild 1892095 5395534
```

The Java target uses `System.identityHashCode(`*node*`)` to generate unique identifiers.

4.7 Debugging Tree Parsers

After a language application's parser creates an AST intermediate form, the subsequent phases walk or manipulate the AST, passing it along to the next phase. A final phase emits output (usually structured text) as a function of the AST and any supporting data structures, such as a symbol table, constructed previously. The core activity of any phase is tree walking using one of three fundamental approaches: the visitor pattern, a hand-built tree walker, or a formal tree grammar.

The visitor pattern performs a depth-first walk, executing an action method at each node. Although easy to grasp, visitors work for only the simplest of translators. Visitors do not validate tree structure, and actions are isolated "event triggers" that know only about the current node and nothing about the surrounding tree structure. For example, visitor actions do not know whether an identifier node is the left-hand side of a variable assignment or in an expression.

Perhaps the most common implementation strategy used today is the hand-built tree walker, most likely with heterogeneous tree nodes (a target language object type for each node type). The structure of the tree is verified as the walker recursively descends just like a top-down parser. Arbitrary tree structure context is available in the form of parameters passed downwards or globally visible variables such as instance variables. As with hand-built parsers, this method is extremely flexible but lacks the rigor of a grammatical approach.

The final tree walking approach is to use a tree grammar [3] from which ANTLR can generate a tree walker. Tree grammars are easier to read, write, and debug than unrestricted handwritten code just as parser grammars are easier to deal with than hand-built parsers. Further, ANTLR can analyze the grammar looking for problems that would go unnoticed in a hand-built tree walker, such as unreachable alternative productions. In the compiler arena, tree grammars are commonplace in the form of instruction selectors such as BURG [23]. The main difference between an ANTLR tree grammar used for translation and a tree grammar used for code generation is that ANTLR tree grammars must be unambiguous.[2] As with parser grammars, tree grammars usually result in more maintainable solutions and provide accurate and concise documentation about the structure of a translator's intermediate form.

Parsing a tree is a matter of walking it and verifying that it has not only the proper nodes but also the proper two-dimensional structure. ANTLR serializes trees into one-dimensional streams of tree nodes computed by iterating over the nodes in a tree via a depth-first walk. To encode the two-dimensional structure, ANTLR inserts imaginary UP and DOWN nodes to indicate the start or end of a child list. In this manner, ANTLR reduces tree parsing to conventional one-dimensional token stream parsing. In fact ANTLR lexers, parsers, and tree parsers all derive from the same `BaseRecognizer` class in the Java target. This normalization makes ANTLRWorks' job much easier. There are only two differences from token string parsing: (1) the input pane must display a tree instead of input text and (2) token lookahead and consume events become `LN` (lookahead node) and `consumeNode`, respectively; see top half of Figure 12.

Because of their two-dimensional nature, debugging tree parsers with a generic programming debugger is notoriously difficult. Also, the relationship of the AST nodes to original input symbols is not always obvious. As observed on the ANTLR mailing list, this is one of the reasons why developers resort to hand-built tree walkers.

ANTLRWorks can debug tree parsers just as easily as token stream parsers so that developers can trace the parse. Further, clicking on an input AST node or parse tree node highlights the associated element in the input tree pane and shows the associated grammar position. Figure 13 shows the AST built by the `def` rule in the previous section as input to the following tree grammar:

```
tree grammar TP;
options {tokenVocab=T;} // share token types with T grammar
def : ^(VARDEF 'int' ( ^(MOD modifier) )+ ID INT)
    | ^(VARDEF 'int' ( ^(MOD modifier) )+ ID)
    ;
modifier : 'public' | 'static' ;
```

[2]Code generator grammars specify multiple productions for the same tree construct because the same operation can often be implemented in multiple ways in machine language. E.g., `x+1` can be implemented with an add instruction and an increment register instruction. BURG performs an optimal bottom-up walk of the tree, choosing productions that provide the lowest overall cost.

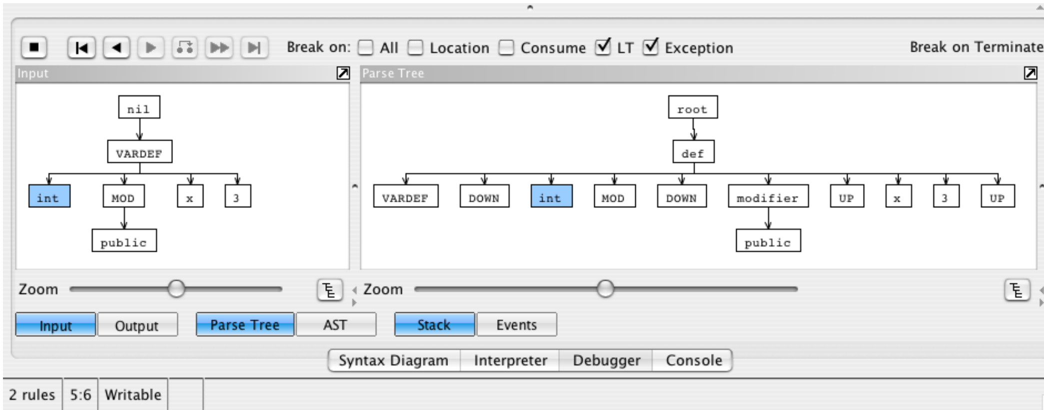


Figure 13: AST Input and Parse Tree from Tree Grammar `TP` and Input "`public int x = 3;`"

The tree grammar was built by cutting out the parse grammar productions, leaving only the AST rewrite specifications in `def`. ANTLRWorks shows the parse tree computed while applying grammatical structure to the input AST using the rules in tree grammar `TP`. The parse tree illustrates that the AST created from input "`public int x = 3;`" (shown in the input pane) is serialized to node stream:

```
VARDEF DOWN int MOD DOWN public UP x 3 UP
```

The original input is obtained from a parse tree by collecting the leaves. The relationship between the AST and the serialized node stream is made more clear by comparing the node stream and the AST in tree grammar notation:

```
^(VARDEF int ^(MOD public) x 3)
```

DOWN immediately follows root nodes and UP ends child lists.

5 MISCELLANEOUS FEATURES

ANTLRWorks provides a grammar-aware editor that has the expected features like syntax coloring, auto-indentation, and identifier auto-completion but also has a number of other features that make grammar development easier as described in the next two sections.

5.1 Grammar navigation and structure

ANTLRWorks provides several navigation features: the developer can jump to the declaration of any token or rule from the reference with a keystroke and jump to any rule by typing its first few letters. In addition, a "find usages" function can be used to find all usages of a particular token or rule. These features are useful when trying to understand the structure of a grammar.

Getting an overall view of the rule relationships can be challenging with large grammars so ANTLRWorks also provides a rule dependency graph showing the rules invoked by each rule. This helps when modifying a rule because it gives some idea of the effect on the overall grammar. ANTLRWorks can produce a dependency graph starting at any lexer or parser rule.

As grammars get larger, organizing the grammar becomes more important. For example, developers usually define all of the expression-related rules contiguously in a grammar file. ANTLRWorks can logically group rules in the rule list pane as well, allowing the developer to see the grammar as a hierarchy of rules instead of a flat text file. The lexical analyzer rules are automatically grouped under a lexer group. In addition, as actions become more numerous in a grammar, the surrounding grammar elements become obscured; actions within rules can be collapsed to hide them.

5.2 Refactoring

Refactoring code [24] is an important programmer activity when developing large, long-lived applications because it improves code readability, stability, and maintainability. Eclipse [25] and IntelliJ [26] offer a huge variety of convenient refactoring operations and programmers have come to expect these features in their development environments. ANTLRWorks provides a small but useful set of refactoring operations that help developers clean up grammars and migrate legacy LR-based grammars for tools such as YACC to an LL-based grammar for ANTLR:

- *Remove left recursion.* Grammars found on the Internet often use left recursion to encode repeated elements. ANTLRWorks can refactor left-recursive rules into equivalent versions using EBNF closure subrules.

- *Rename tokens, rename rules, and replace literals with token name.* As grammar development proceeds, the functionality of rules can drift over time. Renaming grammar elements can make grammars easier to maintain.

- *Extract and inline rules.* As rules grow and shrink during grammar development, these refactoring patterns are very helpful because they encourage programmers to build well-structured grammars.

6 RELATED WORK

There are numerous tools related to grammar development environments with graphical interfaces. Most of them are academic, but there are some commercial tools. Some of the academic tools are focused on generating language development editors or other applications using generative programming from syntax and semantic specifications. ANTLRWorks is tightly focused on grammar development itself rather than grammar development as a means to creating another application. In that sense, ANTLRWorks has more in common with commercial tools, which are typically parser generators that come with grammar development environments.

Commercial parser generator tools are, unfortunately, not open source, can be expensive, and some of them seem not to be maintained anymore: we were unable to get answers to e-mails about ProGrammar [27] and Visual Parse++ [28]. While each commercial tool has some interesting features, ANTLRWorks has a superset of these features. Naturally all commercial tools have a grammar editor but most of them appear to use a raw text editor. ANTLR Studio [29] has probably the best editor experience because of it fast syntax coloring, intelligent auto-completion, and ability to edit Java grammar actions. ANTLR Studio, however, only works with ANTLR version 2 grammars not version 3. ANTLRWorks' editor also provides features such as "find usages", rule dependency graph visualization, and grammar refactoring. The rapid prototyping interpreter feature of ANTLRWorks

appears to be unavailable in commercial tools; TextTransformer [20] has a similar feature, but we were unable to get it to work.

Every commercial grammar development tool has a debugger, but only ProGrammar and TextTransformer have the data-centric breakpoint feature. ANTLRWorks has three main advantages: (1) its rewind and replay mode based on the event stream, (2) its ability to attach to a remote parser running on another machine, (3) its ability to debug parsers written in any language.

There are a number of academic tools that allow you to develop grammars such as ASF+SDF's Meta-Environment [11], LAUNCHPADS [21], Synthesizer Generator [30] (has now become commercial), SmartTools [31], LISA [32], and GTB [33] (Grammar ToolBox). Meta-Environment has grammar-aware editing, on-the-fly parser generation (supporting immediate grammar testing), GLR parse forest visualization, and some nice features that identify grammar issues such as useless symbols, typos, and inconsistent priorities and associativity. Meta-Environment can also generate syntax highlighters and debuggers for languages described by grammars, which includes debugging its own grammars [19]. ANTLRWorks' debugger is less general because it works only on ANTLR grammars but is commensurately simpler and more task specific.

LAUNCHPADS is an interesting grammar development environment for data description languages that allows users to iteratively develop data grammars by selecting and identifying the various fields. Users are also able to quickly test data against the grammar and can see resulting parse trees. A statistical approach to automatically discovering underlying grammatical structure of data (such as log files) is under development.

SmartTools [31] has a grammar aware editor and can display parse trees.

LISA has a number of features that are similar to ANTLRWorks including a grammar-aware editor, BNF viewer, syntax tree viewer, and automata visualizer. Like Meta-Environment and Synthesizer Generator, LISA can automatically generate syntax directed editors and other visualization tools.

While GTB has a variety of visualization views to show grammars, finite automata, and grammar dependency graphs, GTB is not strictly speaking an interactive grammar development environment.

In the related world of natural language processing, Allman and Beale [34] provide a system for building natural language grammars using a visual interface and also contains a grammar debugger.

DDF [35] (DSL Debugger Framework) is a set of Eclipse plugins for debugging domain specific languages described with ANTLR grammars. Aspects are used to weave in support code that maps the generated general purpose code back into the domain specific language source code. DDF reuses the existing Java Eclipse general debugger to debug domain specific languages. DDF is similar to ANTLR Studio in that both map generated code back to a domain specific language, (an ANTLR grammar in ANTLR Studio's case and the domain specific language source code in DDF's case). In contrast, ANTLRWorks uses a custom parser debugger that works with any target language.

Finally, we understand that a grammar development environment for SableCC [36] is under construction.

7 FUTURE WORK

There are several potential improvements to ANTLRWorks we are considering. For example, ANTLRWorks is unable to debug lexers because ANTLR currently does not trigger debugging events for lexers. Also, ANTLR integrates the StringTemplate template engine [37] by allowing rules to specify template rewrites analogous to AST rewrites. At the moment, ANTLRWorks does not have syntax highlighting for templates.

In the testing realm, a grammar testing tool called *gUnit* is under development (inspired by parse-unit, a part of Stratego/XT that can test SDF syntax definitions [38]). ANTLRWorks will need to associate unit tests with rules, letting developers specify input/output pairs, input/parse-tree pairs, etc...

ANTLRWorks could also introduce a test button that automatically sought out input sequences that crashed the language application. ANTLR knows how to generate random sentences from combined lexer/parser grammars. ANTLRWorks could automatically test these language applications with these random sentences, looking for exceptions and other program errors.

8 CONCLUSION

ANTLRWorks is a complete development environment for ANTLR grammars. Its editor has useful grammar editing, refactoring, and navigation features. The interpreter supports rapid grammar prototyping by allowing developers to test grammar rules separately as they are developed, all without having to go through the build process each time. By highlighting nondeterministic paths in the syntax diagram, ANTLRWorks provides a helpful tool for identifying and understanding parser nondeterminisms and other grammar issues. The debugger supports any ANTLR target language and has some features unavailable in other systems including its ability to "time travel", stepping backwards in the parse to discover the chain of events leading up to a parser state of interest. ANTLRWorks tries to make debugging backtracking parsers simpler. Speculatively-matched alternatives are included in the parse tree and differentiated from successful alternatives by color gradations. For parsers that build ASTs, ANTLRWorks makes it easy to identify which nodes are created and where during the parse to isolate erroneously constructed ASTs. Subsequent phases of a translator that use tree grammars may also be debugged like parser grammars. ANTLRWorks is written entirely in Java for portability reasons and is available open-source under the BSD license from `http://www.antlr.org/works`. A plug-in for IntelliJ is available.

9 ACKNOWLEDGMENTS

We would like to thank Bryan Ford, Sylvain Schmitz, and the anonymous reviewers for their detailed reviews and excellent suggestions.

References

[1] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005.

[2] Stephen C. Johnson. YACC — yet another compiler compiler. Computing science technical report 32, AT&T Bell Laboratories, Murray Hill, New Jersey, July 1975.

[3] Terence J. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Programmers, 2007. ISBN 0-9787392-5-6.

[4] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program

executions. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 154–163, New York, NY, USA, 2006. ACM Press.

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

[6] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers*. Addison Wesley, January 1986.

[7] Terence John Parr. *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.

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

[9] Mark-Jan Nederhof. Practical experiments with regular approximation of context-free languages. *Comput. Linguist.*, 26(1):17–44, 2000.

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

[11] Mark van den Brand, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter Olivier, Jeroen Scheerder, Jurgen Vinju, Eelco Visser, and Joost Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proceedings of Compiler Construction 2001 (CC 2001)*, LNCS. Springer, 2001.

[12] Margaret A. Ellis and Bjarne Stroustrup. *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[13] Scott McPeak and George C. Necula. Elkhound: A fast, practical GLR parser generator. In *Compiler Construction*, pages 73–88, 2004.

[14] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL'04*, pages 111–122. ACM Press, 2004.

[15] Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time. In *ICFP'02*, pages 36–47. ACM Press, 2002.

[16] Robert Grimm. Better extensibility through modular syntax. In *PLDI'06*, pages 38–51. ACM Press, 2006.

[17] J. Cordy. The TXL source transformation language, 2006.

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

[19] P. Olivier. A framework for debugging heterogeneous applications, 2000.

[20] Detlef Meyer-Eltz. TextTransformer. http://www.texttransformer.com.

[21] Mark Daly, Yitzhak Mandelbaum, David Walker, Mary F. Fernández, Kathleen Fisher, Robert Gruber, and Xuan Zheng. PADS: an end-to-end system for processing ad hoc data. In *SIGMOD Conference*, pages 727–729, 2006.

[22] Anthony M. Sloane. Debugging Eli-generated compilers with Noosa. In *Compiler Construction 1999*, Amsterdam, The Netherlands, 1999.

[23] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *LOPLAS*, 1(3):213–226, 1992.

[24] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[25] Eclipse homepage. http://www.eclipse.org.

[26] Intellij homepage. http://www.jetbrains.com/idea/.

[27] NorKen Technologies Inc. The ProGrammar IDE. http://www.programmar.com.

[28] SandStone. Visualparse++. http://www.sand-stone.com.

[29] Prashant Deva. Antlrstudio. http://www.placidsystems.com.

[30] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator: a system for constructing language-based editors*. Springer-Verlag, New York, USA, 1989. ISBN 0-387-96857-1.

[31] Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Didier Parigot, and Claude Pasquier. SmartTools: a generator of interactive environment tools. *Electr. Notes Theor. Comput. Sci.*, 44(2), 2001.

[32] Pedro Henriques, Maria Joo Varanda, Marjan Mernik, Mitja Lenic, Jeff Gray, and Hui Wu. Automatic generation of language-based tools using LISA system. *IEE Software Journal*, 152(2):54–70, April 2005.

[33] Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. Evaluating GLR parsing algorithms. *Sci. Comput. Program.*, 61(3):228–244, 2006.

[34] T. Allman and S. Beale. An environment for quick ramp-up multi-lingual authoring. *International Journal of Translation*, 16(1), 2004.

[35] Hui Wu, Jeff Gray, Suman Roychoudhury, and Marjan Mernik. Weaving a debugging aspect into domain-specific language grammars. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1370–1374, New York, NY, USA, 2005. ACM Press.

[36] Etienne M. Gagnon and Laurie J. Hendren. SableCC, an object-oriented compiler framework. In *TOOLS (26)*, pages 140–154, 1998.

[37] Terence Parr. StringTemplate template engine. http://www.stringtemplate.org.

[38] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.16. a language and toolset for program transformation. *Science of Computer Programming*, 2007. (To appear).