# LL and LR in Context: Why Parsing Tools Are Hard

September 6, 2013

In my last blog entry LL and LR Parsing Demystified, we explored LL and LR parsers from a black-box perspective. We arrived at a model for these parsers where both their input and output were streams of tokens, with the parser inserting rules as appropriate according to Polish and Reverse Polish notation.

In future articles I want to focus in even closer on the details of LL and LR algorithms, but I realized that I should first zoom out and give some motivation for why anyone should care about LL or LR to begin with.

As I wrote this article, it turned into an answer to the question "why is parsing hard?" Or alternatively "why doesn't everybody use parser generators?" LL and LR parsing theory is taught in in books like Compilers: Principles, Techniques, and Tools (known as "The Dragon Book" and used in many university compilers courses), but then people graduate to find that most parsers in the real world don't work like this. What gives? This article is my answer to that question.

## Theory vs. Practice

The theory of LL and LR parsing is almost 50 years old: Knuth's paper On the Translation of Languages from Left to Right that first defined LR(k) was published in 1965. This is only one of an incredible number of mathematically-oriented papers about parsing and language theory. Over the last 50 years academics have explored the mathematical dimensions of parsing with great vigor, but the field is nowhere near exhausted; even in the last five years we've seen some entirely new and important results published. One of the best surveys of the field is the book Parsing Techniques: A Practical Guide, whose bibliography contains over 1700 cited papers!

Despite this vast body of theoretical knowledge, few of the parsers that are in production systems today are textbook cases of the theory. Many opt for hand-written parsers that are not based on any formalism at all. Language specifications are often defined in terms of a formalism like BNF, but it's almost never the case that real parsers can be generated directly from this formalism. GCC moved *away* from their Bison-based parser to a handwritten recursive descent parser. So did Go. While some notable language implementations do use Bison (like Ruby and PHP), many choose not to.

Why this divergence between theory and practice? While it is tempting to blame ignorance of the literature, that could hardly explain why GCC moved *away* from an LR parser.

I think it is safe to say that *pure* LL and LR parsers have proven to be largely inadequate for real-world use cases. Many grammars that you'd naturally write for real-world use cases are not LL or LR, as we will see. The two most popular LL and LR-based parsing tools (ANTLR and Bison, respectively) both extend the pure LL and LR algorithms in various ways, adding features such as operator precedence, syntactic/semantic predicates, optional backtracking, and generalized parsing.

But even the evolved tools that are currently available sometimes come up short, and are still evolving to address the traditional pain points of parser generators. ANTLR v4 completely reworked its parsing algorithm to improve ease-of-use vs ANTLR v3 with a new algorithm it calls ALL(*). Bison is experimenting with IELR, an alternative to LALR that was published in 2008 and intended to expand the number of grammars it can accept and parse efficiently. Some people have explored alternatives to LL/LR such as Parsing Expression Grammars (PEGs) that attempt to solve these pain points in a different way entirely.

Does this mean that LL and LR are obsolete? Far from it. While *pure* LL and LR do indeed come up short in several ways, these algorithms can be extended in ways that preserves their strengths, in much the same way that a multi-paradigm programming language can offer features of imperative, functional, and object-oriented programming styles. I firmly believe that as parser tools continue to improve with better tooling, better error reporting, better visualization, better language integration, etc. they will become something you'd reach for as readily as you reach for a regex today. There is a lot of room for improvement in this space, and I want to help make that happen (my tabled project Gazelle is where I have invested effort so far and I

intend to do more). But I digress.

LL and LR parsers have some indisputable strengths. They are the most efficient parsing algorithms around. The grammar analysis they perform ahead-of-time can tell you important things about your grammar, and properly visualized can help you catch bugs, in much the same way that regex visualizing tools like regexper can. They offer some of the earliest and best error reporting of syntax errors at parse time (this is separate from the shift/reduce and reduce/reduce errors you might get at *grammar analysis* time).

Even if you are not sold on the usefulness of LL and LR, learning about them will help you better understand the tradeoffs that your favorite parsing method makes compared to LL/LR. Alternatives to LL/LR are generally forced to give up some at least one of the advantages of LL/LR.

## Clarifying “LL parser” and “LR parser”

“LL parser” and “LR parser” are not actually specific algorithms at all, but rather generic terms referring to *families* of algorithms. You may have seen names such as LR(k), “full LL”, LALR(1), SLR, LL(*), etc; these are specific algorithms (or variants of the same algorithm, depending on how you look at it) that fall under the category of “LL parser” or “LR parser.” These variants have different tradeoffs in terms of what grammars they can handle and how big the resulting parsing automata are, but they share a common set of characteristics.

LL and LR parsers *usually* (but not always) involve two separate steps: a grammar analysis step that is performed ahead-of-time and the actual parser that runs at parse time. The grammar analysis step builds an automaton if it can, otherwise the grammar is rejected as not LALR/SLL/SLR/whatever. Once the automaton is built, the parsing step is much simpler because the automaton encodes the structure of the grammar such that what to do with each input token is a simple decision.

What then is the distinguishing characteristic that makes a parser an LL parser or an LR parser? We will answer the question with a pair of definitions. Don’t worry if these definitions make no sense to you; the entire rest of the article is dedicated to explaining them. These definitions are not given in the literature, since they are informal terms, but they correspond to the general usage of what is meant if you look at (for example) the Wikipedia pages for “LL Parser” or “LR Parser.”

*An **LL parser** is a deterministic, canonical top-down parser for context-free grammars.*

*An **LR parser** is a deterministic, canonical bottom-up parser for context-free grammars.*

Any parser that meets these definitions is an LL or LR parser. Both the strengths and weaknesses of LL and LR are encapsulated in these definitions.

Beware that not every parser with “LR” or “LL” in its name is actually an LR or LL parser. For example, GLR, LR(k,∞), and Partitioned LL(k) are all examples of parsing algorithms that are not actually LL or LR; they are variations on an LL or LR algorithm, but give up one or more of the essential LL/LR properties.

We will now more deeply explore the key parts of these definitions.

## Context-Free Grammars: powerful, but not all-powerful

LL and LR parsers use *context-free grammars* as their way of specifying formal languages. Most programmers have seen context-free grammars in one form or another, possibly in the form of BNF or EBNF. A close variant called ABNF is used in documentation of protocols in RFCs.

On one hand CFGs are really nice, because they match the way that programmers think about languages. The fact that RFCs use a CFG-like abstraction to write documentation speaks to how readable context-free grammars can be.

Here is the JSON context-free grammar from my previous article:

```
object → '{' pairs '}'

pairs → pair pairs_tail | ε
pair → STRING ':' value
pairs_tail → ',' pairs | ε

value → STRING | NUMBER | 'true' | 'false' | 'null' | object | array
array → '[' elements ']'

elements → value elements_tail | ε
elements_tail → ',' elements | ε
```

This is so intuitive to read that I didn't even bother to explain it. An object is a bunch of pairs surrounded by curly brackets. A "pairs" is either a pair followed by a pairs_tail or empty. It reads really nicely.

Context-free grammars not only tell us whether a given string is valid according to the language, they also define a tree structure for any valid string. It's the tree structure that helps us figure out what the string actually *means*, which is arguably the most important part of parsing (a compiler that did nothing but say "yep, this is a valid program" wouldn't be very useful). So writing a CFG really goes a long way in helping us parse and analyze a language.

On the other hand context-free grammars can be frustrating, for two related reasons:

1. When writing a CFG intuitively, you often end up with something ambiguous.
2. When writing a CFG intuitively, you often end up with something that is unambiguous but can't be parsed by LL or LR algorithms.

While the second problem is LL/LR's "fault" the first is just an inherent challenge of designing formal languages. Let's talk about ambiguity first.

## Ambiguity in CFGs

If a grammar is ambiguous, it means that there is at least one string that can have multiple valid parse trees. This is a real problem in the design of a language, because the two valid parse trees almost certainly have different *semantic* meaning. If both are valid according to your grammar, your users cannot know which meaning to expect.

The simplest and most common example is arithmetic expressions. The intuitive way to write a grammar is something like:

```
expr → expr '+' expr |
       expr '-' expr |
       expr '*' expr |
       expr '/' expr |
       expr '^' expr |
       - expr |
       NUMBER
```

But this grammar is highly ambiguous because it doesn't capture the standard rules of

precedence and associativity. Without these rules to disambiguate, a string like `1+2*3-4^5` has exponentially many valid parse trees, all with different meanings.

It's possible to rewrite this to capture the rules of precedence and associativity:

```
expr → expr '+' term |
       expr '-' term |
       term

term → term '*' factor |
       term '/' factor |
       factor

factor → '-' factor |
         prim

prim → NUMBER |
       NUMBER '^' prim
```

Now we have an unambiguous grammar that encodes the precedence and associativity rules, but it is not at all easy or self-evident what these rules are from reading the grammar. For example, it's certainly not obvious at first glance that all of the operators in this grammar are left-associative *except* exponentiation (`^`) which is right-associative. And it's not easy to write grammars in this style; I have a fair amount of experience writing grammars but I still have to slow down and be careful (without testing, I'm not even 100% confident I have got it right).

It's really unfortunate that one of the first and most common use cases of text parsing is one that pure context-free grammars are so bad at. No wonder people can get turned off of CFG-based tools when something that seems like it should be so simple ends up being so complicated. It's especially unfortunate because other non-CFG parsing techniques like the Shunting Yard Algorithm are so good at this kind of operator precedence parsing. This is clearly one of the most glaring examples where CFGs and *pure* LL/LR let us down.

Another famous example of actual grammar ambiguity is the dangling else problem. For languages that don't have an "endif" statement, what does this mean?

```
if a then if b then s else s2

// Ambiguity: which of these is meant?
if a then (if b then s) else s2
if a then (if b then s else s2)
```

Unlike with arithmetic expressions, there are no standard precedence/associativity rules to tell us which of these interpretations is "correct." The choice here is pretty much arbitrary. Any language that has this construct must tell users which meaning is correct. The grammar ambiguity here is a *symptom* of the fact that our language has a confusing case.

One final example of ambiguity, this one from C and C++. This is known as the type/variable ambiguity.

```
  // What does this mean?
  x * y;
```

The correct answer is that it depends on whether `x` was previously declared as a type with `typedef`. If `x` is a type, then this line declares a pointer-to-`x` named `y`. If `x` is *not* a type, then this line multiplies `x` and `y` and throws away the result. The traditional solution to this problem is to give the lexer access to the symbol table so it can lex a type name differently than a regular variable; this is known as the lexer hack. (While this may seem out of place in an article about parsers, the same ambiguity manifests in C++ in a way that cannot so easily be confined to the lexer).

In other words, this ambiguity is resolved according to the semantic context of the statement. People sometimes refer to this as a "context-sensitive," (like the article The context sensitivity of C's grammar), but context-sensitive grammar is a very specific term that has a mathematical meaning in the Chomsky hierarchy of languages. The Chomsky definition refers to *syntactic* context sensitivity, which almost never occurs in computer languages. Because of this, it's good to clarify that we are talking about *semantic* context-sensitivity, which is an entirely different thing.

A key point about semantic context-sensitivity is that you need a Turing-complete language to properly disambiguate between the ambiguous alternatives. What this means for parser generator tools is that it's effectively impossible to parse languages

like this correctly unless you allow the user to write arbitrary disambiguation code snippets in a Turing-complete language. No mathematical formalism alone (not CFGs, not PEGs, not operator grammars) can be sufficient to express these languages. This is one very notable case where theory and practice diverge.

In tools such as ANTLR, these disambiguating code snippets are known as “semantic predicates.” For example, to disambiguate the type/variable ambiguity, you’d need to write code to build/maintain a symbol table whenever a “typedef” is seen, and then a predicate to see if a symbol is in the table or not.

## Dealing with ambiguity in CFGs

No matter what parsing strategy is being used, a language designer must be aware of and directly confront any ambiguities in the language. If possible, the best idea is often to change the language to avoid having the ambiguity at all. For example, most languages these days do not have the dangling else problem because “if” statements have an explicit end (either an “endif” keyword or curly brackets that surround the statements in the “else” clause).

If the designer can’t or doesn’t want to remove the ambiguity, they must decide which meaning is intended, implement the ambiguity resolution appropriately, and communicate this decision to users.

But to confront ambiguities you must first know about them. Unfortunately this is easier said than done. One of the huge bummers about grammars (both CFGs and other formalisms like PEGs) is that many of the useful questions you might want to ask about them are *undecidable* (if you haven’t studied Theory of Computation, a rough approximation for “undecidable” is “impossible to compute”). Determining whether a context-free grammar is ambiguous is unfortunately one of these undecidable problems.

If it’s impossible to compute whether a grammar is ambiguous *a priori*, how can we be aware of ambiguities and address them?

One approach is to use a parsing algorithm that can handle ambiguous grammars (for example GLR). These algorithms can handle any grammar and any input string, and can detect at parse time if the input string is ambiguous. If ambiguity is detected, they can yield all valid parse trees and the user can disambiguate between them however

they see fit.

But with this strategy you don't learn about ambiguity until an ambiguous string is seen in the wild. You can never be *sure* that your grammar is unambiguous, because it could always be the case that you just haven't seen the right ambiguous string yet. You could ship your compiler only to learn, years later, that your grammar has had an unknown ambiguity in it all along. This can actually happen in the real world; it was not discovered that ALGOL 60 had a "dangling else" problem until the language had already been published in a technical report.

Another strategy is to abandon context-free grammars completely and use a formalism like parsing expression grammars that is unambiguous by definition. Parsing expression grammars avoid ambiguity by forcing all grammar rules to be defined in terms of *prioritized* choice, so in cases where multiple grammar rules match the input the first one is correct *by definition*.

```
// PEG solution of the if/else ambiguity:
stmt <- "if" cond "then" stmt "else" stmt /
        "if" cond "then" stmt /
        ...
```

Prioritized choice is a great tool for resolving some ambiguities; it works perfectly for the solving the dangling else problem. But while this has given us a tool for *resolving* ambiguity, it hasn't solved the problem of *finding* ambiguities. Every rule in PEGs is required to be defined in terms of prioritized choice, which means that every PEG rule could be hiding a "conceptual" ambiguity:

```
// Is this PEG rule equivalent to a <- c / b ?
a <- b / c

// We can't know (it's undecidable in general),
// so every rule could be hiding an ambiguity we don't know about.
```

I call this a "conceptual" ambiguity because even though a PEG-based tool does not consider this ambiguous, it still looks ambiguous to a user. Another way of thinking about this is that you have resolved the ambiguity without ever being aware the ambiguity existed, thus denying you the opportunity to think about it and make a conscious choice about how it should be resolved. Prioritized choice doesn't make the

dangling else problem go away, it just hides it. Users still see a language construct that could plausibly be interpreted in two different ways, and users still need to be informed which option the parser will choose.

Prioritized choice also requires that an ambiguity is resolved in the same way each time; it can't accommodate cases like the C/C++ variable/type ambiguity which are resolved by semantic information.

And unlike GLR, Packrat Parsing (the linear-time algorithm for parsing PEGs) doesn't tell you even at parse time if the input string is ambiguous. So with a Packrat-Parsing-based strategy, you are really flying blind about whether there are "conceptual" ambiguities in your grammar. It's also possible for entire alternatives or rules of a PEG to be unreachable (see here for a bit more discussion of this). The net result is that with PEGs you know very little about the properties of your grammar.

So far none of the options we've discussed can actually help us find ambiguities up-front. Surely there must be a way of analyzing our grammar ahead of time and proving that it's not ambiguous, as long as the grammar isn't too crazy? Indeed there is, but the answer brings us back to where we started: LL and LR parsers.

It turns out that simply trying to construct an LR parser for a grammar is very nearly the most powerful ambiguity test we know of for CFGs. We know it cannot be a perfect test, since we already stated that testing ambiguity is undecidable. If a grammar is not LR we don't know whether it is ambiguous or not. But every grammar that we *can* construct an LR parser for is guaranteed to be unambiguous, and as a bonus, you also get an efficient linear-time parser for it.

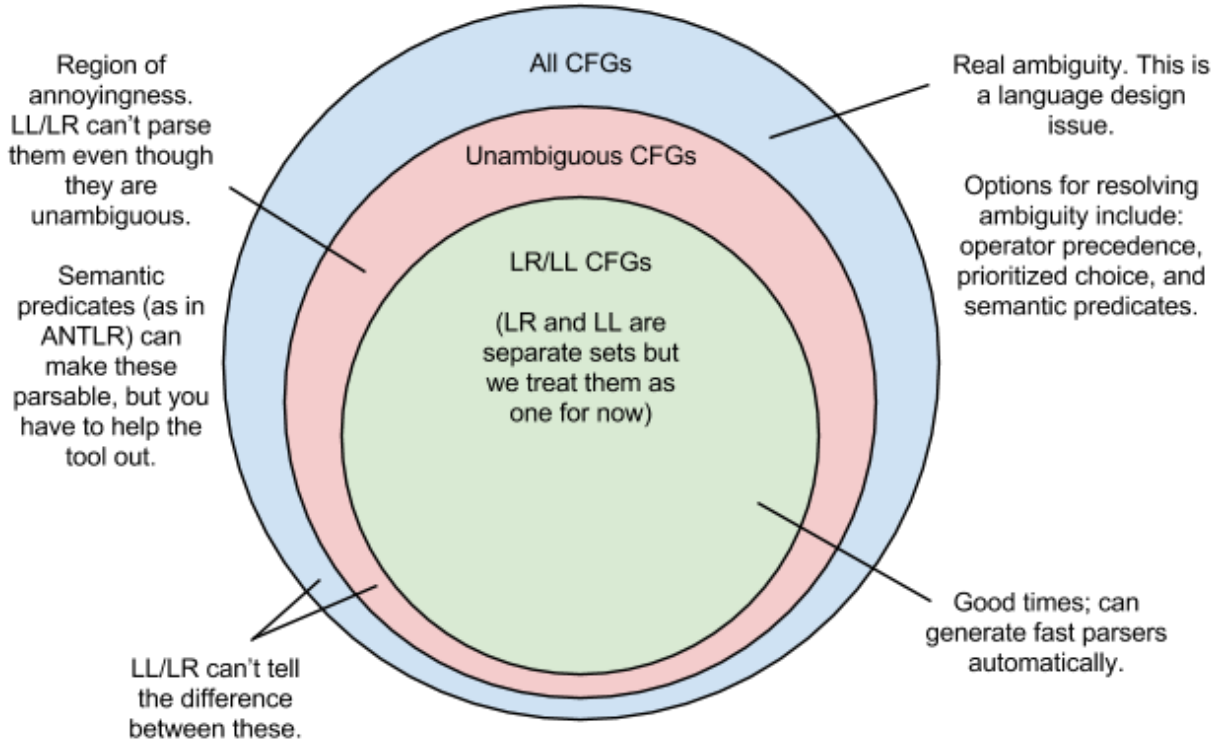
But wait, there's more. Let's review the three types of ambiguity we've encountered and the most natural solution for solving each:

1. *Arithmetic expressions*: the ideal solution is to be able to **declare precedence/associativity directly**, and *not* have to solve it at a grammar level.
2. *Dangling else*: because this can be resolved by always preferring one alternative over another, the ideal solution is **prioritized choice**. Another example of this case is C++'s "most vexing parse", which is likewise resolved by simply preferring one interpretation over the other when both are valid.
3. *Type/variable ambiguity*: the only real solution is to allow Turing-complete **semantic predicates** to resolve the ambiguity. C++ has an even more extreme version of this problem since type/variable disambiguation could require arbitrary

amounts of template instantiation, and therefore just *parsing* C++ is technically undecidable (!!), unless you limit template instantiation depth – more gory detail and an example here.

The good news is that all three of these ambiguity-resolution strategies can be incorporated into a LL or LR-based parser generator, to some extent. While *pure* LL or LR only support context-free grammars, it is entirely possible to add operator precedence, prioritized choice, *and* semantic predicates to such tools, subject to some limitations. I liken this to multi-paradigm programming languages. Just as a language that supports procedural, OO, and functional styles is more powerful and expressive than a language offering just one, so is a CFG+precedence+predicates tool more powerful than one that only supports CFGs.

We can sum up all of this information with this venn diagram:




To more fully understand how LL/LR grammar analysis can prove grammars unambiguous (and how we can add non-CFG features like prioritized choice to them), we will explore the concept of a deterministic parser.

## Deterministic Parsers

Without going into too much detail, a *deterministic* parser is one that works by building a deterministic automaton (this is very much related to automata theory for regular expressions, except that parsing automata also have a stack). That means that as the parser reads tokens left-to-right, it is always in one particular state, and each token transitions it into exactly one other state.

More informally, we can say that a deterministic parser is one that doesn't have to do any guessing or searching. Terence Parr, author of ANTLR, often uses the metaphor of parsing as a maze; at every fork in the maze, a deterministic parser always knows which fork to take the first time. It might "look ahead" to make the decision, but it never makes a decision and then backs up (a parser that "backs up" is known as a "backtracking parser" and has exponential worst-case running time).

This determinism is really the defining characteristic that gives LL/LR both their advantages and disadvantages. They are the fastest algorithms because they are simply transitioning a state machine. They are not only fast, they are *predictably* fast: they have have *worst-case* O(n) performance. Some approaches like NFAs (nondeterministic finite automata) or backtracking parsers have good performance in common cases but can degrade severely (even exponentially) in degenerate cases. Many popular regex engines have this issue (see the article Regular Expression Matching Can Be Simple And Fast for an example). You might think the bad cases are uncommon, but anyone who wants to DoS-attack your service can find and exploit them.

Besides being fast, you can know that an LL/LR grammar is unambiguous because an ambiguous grammar won't allow you to build a deterministic automaton. To build the automaton you have to be able to prove, for every grammar state and every input token, that there is only one valid path through the grammar that you can take for this token. A Bison "shift/reduce" or "reduce/reduce" conflict is a case where Bison was not able to make the parser deterministic, because two state transitions for the same token were both valid. But Bison can't prove whether this is because of grammar ambiguity (in which case both transitions could ultimately lead to a successful parse) or whether this is an unambiguous grammar that just isn't LR (in which case one of the paths would eventually hit a dead-end.

"Generalized parsing" algorithms like GLR and GLL can handle any grammar, because they just take both paths simultaneously. If one of the paths hits a dead-end, then we're back to an unambiguous string. But if multiple paths turn out to all be valid,

we have an ambiguous string and the parser can give you *all* of the valid parse trees.

This also gives us a hint about how pure LL/LR algorithms can be extended with extra features. Any method we can think of for deciding which path is the "right" one is fair game! It turns out that operator precedence declarations can provide a way of resolving shift/reduce and reduce/reduce conflicts in LR parsers, and Bison supports this with its precedence features. Prioritized choice can also give us enough information in some cases to resolve a nondeterminism by deciding that one of the paths is correct because it has a higher priority. And when all else fails, if we can write our own predicates that run at parse time, we can write arbitrary logic that uses whatever other criteria we could possibly want to decide which path is the correct one.

## Conclusion: so why are parsing tools hard?

Given all of this, what is the answer to our original question? Why are parsing tools hard? I think there are two sets of reasons: some *inherent* reasons and some reasons that are just a weakness of current parsing tools and could be improved.

The *inherent* reasons parsing tools are hard have to do with ambiguity. More specifically:

1. The input grammar may be ambiguous but we can't robustly check for this because it's undecidable.
2. We can use a deterministic (LL/LR) parsing algorithm: this gives us a fast parser and a proof that the grammar is unambiguous. But unfortunately no deterministic parsing algorithm can handle all unambiguous grammars. So in some cases we're forced to adapt our grammar and debug any nondeterminism.
3. We can use a generalized parsing algorithm like GLL or GLR that can handle all grammars (even ambiguous ones), but because of (1) we can't know for sure whether the grammar is ambiguous or not. With this strategy we have to always be prepared to get multiple valid parse trees at parse time. If we didn't know about the ambiguity, we probably won't know how we should disambiguate.
4. We can use a formalism like Parsing Expression Grammars that defines ambiguity away – this will always give us one unique parse tree, but can still hide "conceptual" ambiguities.
5. Some real-world ambiguities can't be resolved at a grammar-level because they have semantic context-sensitivity. To parse these languages, there *must* be a way of embedding arbitrary logic into the parser to disambiguate.

While this may all seem annoying, ambiguity is a real language design issue, and anyone designing a language or implementing a parser benefits from getting early warning about ambiguity. In other words, while LL and LR tools are not perfect, some of their pain simply comes from the fact that *parsing* and *language design* are complicated. While rolling your own parser will free you from ever getting error messages from your parser generator, it will also keep you from learning about ambiguities you may be inadvertently designing into your language.

The other reasons parsing tools are hard are things that could realistically be improved. The tools could benefit from greater flexibility, more reusable grammars, better ability to compose languages, and more. This is where the opportunity lies.

---

Josh Haberman
jhaberman@gmail.com

haberman

Parsing, performance, and low-level programming.