

## How did you get started with antlr?

The critical moment was in 1988 while working in Paris with my friend Tom Burns, who later became my business partner at jGuru.com. At the time, Tom and I were working for a robotics company--I was building a compiler, interpreter, VM, and so on for a robot control language called Karel. I built the Karel parser with a recursive descent mechanism because I didn't grok yacc and I knew how to build recursive descent parsers really well. Tom showed me a section I had forgotten in Niklaus Wirth's book: "Algorithms + Data Structures = Programs" that discussed how to represent grammars as syntax diagrams and then how to translate them to Pascal programs that would recognize the syntax. I looked at it and said, "I could do that automatically".

Fast-forward six months to my second year of graduate school at Purdue University. I started taking a course from Hank Dietz on how to build parser generators and lexer generators. As part of the course, I built "yucc" a recursive descent parser generator that was barely LL(1). ANTLR grew out of the project in that class. I started doing all of the grammar analysis necessary and really got into that stuff. Too bad I hadn't paid attention in my automata theory class as an undergraduate.

I was supposed to be studying computer architecture for my masters, but that was not very interesting to me. Since I was already working on ANTLR for fun, my friend Tom suggested I turn that into my masters thesis and get the hell out of there. ;) Ultimately, as you can see, this is all Tom's fault. That explains the mysterious dedication in the upcoming ANTLR book:

<http://www.pragmaticprogrammer.com/titles/tpantlr>

## How does antlr3 compares to antlr2 and antlr1?

ANTLR v1 was written in C and could generate C, C++ output. It introduced a number of important features such as semantic predicates and syntactic predicates. It also had a really nice AST construction facility. v1 also was the first practical parser generator that used  $k > 1$  lookahead. v2 was written (in Java) in a terrible hurry during my start of days at jGuru as sort of a side project. As a result, I had to cut corners and make a number of quick decisions. v2 had semantic predicates, but lacked the sophisticated "hoisting" mechanism from v1. v2 was pretty quirky because I did not have enough time to think about the design. v1 required the use of a separate lexer generator (called DLG, written by Will Cohen). v2 used the same recursive descent mechanism for lexers as it did for parsers, which meant that you could have recursive lexer rules. v2 had C++, C#, Python, and Java targets.

v3 is a completely rewritten version (again in Java) that I have built very carefully and slowly over the past four years. The new code base is very clean and has many many unit tests. After about 20 years of doing research in parser generation, I think I finally understand the problem. v3 represents my thoughts on the ultimate parser generator design. There are number of things have changed for the better in v3 including: BSD license, a brand-new very powerful extension to LL(k) called LL(\*), an auto backtracking mode, partial parsing result memoization to increase the speed of backtracking, a really nice AST rewrite rule mechanism, integration of the StringTemplate template engine for generating structured text, improved error reporting and recovery, and a truly rechargeable code generator. Each target is just a set of templates--you don't have to write code to

make a new target for ANTLR. (Oh, and full semantic predicate hoisting is back in).

ANTLR v3 also has the awesome ANTLRWorks grammar development environment, written by Jean Bovet: <http://www.antlr.org/works>

## **Why swing for antlrworks and how is that working out?**

We were going for maximum portability, which I believe was the right decision. At least, our experience so far is that it works pretty well across platforms. Jean Bovet is an expert GUI builder and worked extremely hard to beat Swing into submission. It is because of his skill that we were able to get Swing to work so well and across platforms. People always express shock that ANTLRWorks is written in Swing.

## **Why LL(k) and how does that compare to other algorithms like SLR LALR, GLR, CYK, Earley etc?**

First, let's get some of the easy relationships out-of-the-way. SLR and LALR are more efficient but weaker variants of LR, which is weaker than GLR (generalized LR). GLR and Earley and CYK can deal with the same class of grammars (all context-free grammars), but GLR is more efficient. GLR is just an LR parser that forks a new LR parser to pursue ambiguous paths. When static grammar analysis reveals a non-LR decision, a GLR parser generator simply generates a special state to simply try out the various alternatives at run time. In a crude sense, a GLR parser uses LR to backtrack across the paths in non-LR parsing decisions. GLR has best case runtime  $O(n)$ , but with the worst case of  $O(n^3)$  just like CYK and Earley. GLR should be nearly linear for most programming language grammars.

The most exciting parsing strategy to appear recently is called packrat parsing (by wizard Bryan Ford). A packrat parser is a top-down recursive descent backtracking parser that records partial parsing results to ensure linear time complexity, at the cost of some memory. A packrat parser chooses from among alternatives purely with backtracking. Bryan Ford also defined PEGs (parser expression grammars) that have no strict ordering with GLR, because there is at least one PEG that is not GLR. Bryan extended and formalized my original syntactic predicates into these really cool PEGs. I have added his PEG technology to ANTLR via an auto backtracking feature. So, ANTLR can deal with just about any grammar you give it now and parse the associated language with linear time complexity.

There is an interesting relationship to point out: GLR is to Earley as ANTLR is to packrat. GLR is more efficient than Earley because GLR relies on traditional LR parsing for all but the non-LR decisions. Similarly, ANTLR is more efficient than a packrat parser because it uses an LL-based parser for all but the non-LL decisions. Furthermore, ANTLR's LL(\*) algorithm is much stronger than the traditional LL(k), thereby reducing even further the amount of backtracking it has to do (due to fewer non-LL(\*) decisions).

ANTLR's new LL(\*) algorithm allows parser lookahead to roam arbitrarily far ahead looking for a token or sequence that disambiguates a decision. LL(k) can only look a fixed k symbols ahead. It is the difference between using a cyclic DFA and an acyclic DFA to make lookahead decisions.

When you put it all together, ANTLR v3 is as powerful as any other current parsing

technique. However, I think you will find that it is more efficient than the others. Perhaps more importantly, ANTLR is more natural to use because it builds human-readable/debuggable recursive descent parsers and allows arbitrary actions anywhere in a grammar.

## **Why use a custom DSL with antlr than xml?**

XML is a data format not a human interface (it's a parse tree rather than the sentence). I cannot stand typing/reading XML. Remember, being an expert in XML is like being an expert in comma separated values. Cracks me up that there are conferences on this data format. I carefully crafted my argument that "Humans should not have to grok XML" here:

<http://www-128.ibm.com/developerworks/xml/library/x-sbxxml.html>

## **What style of "DSLs" or "little languages" would you recommend or NOT recommend antlr?**

Well, ANTLR is great for "heavy lifting". If you know Ruby, you can probably implement a teeny little DSL very quickly because it is so flexible. I mean this not because it is a nice language to write in, but because it has optional parentheses on method calls. Consequently, you can pretty much make Ruby code look like anything you want. Cue ruby on rails, rake, etc...

As you might expect, I am pretty fast at building little languages with ANTLR, but I use sed and awk to do quick little translations. Heck, I can even do some pretty fancy footwork in Emacs for one-off translations.

## **What types of languages is antlr best at? Are there any that antlr is not suited for? (what are alternatives)?**

I don't think there is a classification I can give you here. If you are building a language, ANTLR is a good approach unless it is so small you can do it faster in the tools I mention above. When your implementation starts to look like a handbuilt parser, though, you should step up to ANTLR for that task.

## **How does antlr make it easier to make "user friendly" parsers with good error reporting (versus what you would do yourself)?**

ANTLR automatically generates parsers that provide excellent error messages. ANTLR gives you a "user friendly" parser for free. For example, you automatically get messages such as

```
line 102:34 mismatched input ';' expecting ')'
```

The parser will suppress any further syntax errors until the parser properly resynchronizes. By "resynchronize", I mean that ANTLR-generated parsers also automatically recover very

well from errors. If you forget a ')', the parser will pretend it saw it anyway and keep going etc...

Good error messages can even help during development of a grammar because you will have a lot of bugs among the rules. Good error messages help you track down these problems. You can easily override a method in your parser to generate detailed errors such as:

```
line 802:71 [program, method, stat, expr, atom] no viable alternative,  
token=[@2921,802:71=';',<7>,6092:6093] (decision=5 state 0)  
decision=<<35:1: atom : ( INT | '(' expr ')' );>>
```

Some people claim that they can do very good error reporting and recovery in handbuilt parsers. This is true, but the reality is that parser generators don't get tired whereas programmers do. Consequently, ANTLR will generate much better error messages than you could possibly do by hand for a large grammar. ANTLR gives you all flexibility you have in handbuilt recognizers. For example, you can trap recognition exceptions wherever you want just like in regular code.

## How does antlr compares to javacc and other known parsers?

Well, ANTLR v3 is very similar still to javacc, though v3 now has the auto backtracking and LL(\*) features that give it a good boost. ANTLR v3's new AST rewrite mechanism is also unavailable in other parser generators; it effectively uses a parser grammar to tree grammar mapping to define what trees should look like.

Yacc is still used by a frightening number of people, probably just because that is what they grew up using. Well, a lot has changed in 30 years since yacc came out. There are a number problems with yacc. Two of the biggest are: lack of good error reporting and sensitivity to action placement. It is notoriously difficult to get good error messages and error recovery using yacc (much of the problem stems from the bottom up approach of LR-based parsers).

When it comes to actions, yacc can be really frustrating. Just when you get your grammar to recognize all of your test cases, inserting a single action can cause a grammar ambiguity. This ambiguity can break your grammar and make your input tests fail. LR-based parsers can only execute actions on the right edge of an alternative, because that is the only place the parser knows precisely where it is. To allow actions at non-right edges, yacc splits the rule to get the action on the right edge of a new rule. Unfortunately, splitting the rule can introduce ambiguities. The worst-case scenario is an action on the left edge of every alternative in an LR grammar. Due to rule splitting, the LR parser is reduced in strength to that of an LL parser.

The list goes on and on. For example, yacc has no support for building trees and has very primitive attribute handling. Yacc was great in its time, but there are many better parser generators available.

JavaCC is very similar to the previous incarnation of ANTLR, v2; choosing between them was really a matter of style as they were about as powerful and generated about the same thing with some minor differences (oh, I think JavaCC was a little faster than ANTLR v2).

With the introduction of ANTLR v3, I believe that the choice is now very clearly in ANTLR's

favor as it represents a whole new generation of software (Sriram Sankar, supersmart author of JavaCC, has been too busy in industry to make a new version of JavaCC). To summarize the key advantages (which ANTLR v3 has over most of the parser generators as well):

- LL(\*) parsing technology that is much more forgiving in that it accepts many more grammars
- ANTLRWorks grammar development environment
- AST construction facilities via rule rewrites
- Tree grammars that resulting tree walkers so you don't have to build them by hand
- StringTemplate integration for emitting structure text (i.e., code generation)
- Sophisticated semantic and syntactic predicates mechanisms
- Auto-backtracking so you can type in any old grammar you want and have ANTLR just figured out at runtime (memoization keeps us down to linear parsing time)
- v3 supports many different languages at this point with many others on their way; even Ada!
- A carefully written book that makes it much easier to learn

## **Any chance of getting rid of antlr namespaced runtime dependency? (and inlining by default) - also, why is there a runtime?**

Do you mean generating only the necessary runtime support into the output so you don't have a runtime library? If so, I think ANTLR will always have a runtime library dependency. There's just too much code to generate. It would also make it hard to fix bugs because people have to regenerate their parsers to get the fixed support code in their parsers.

## **How can "downstream" users like us contribute back to antlr? is everything controlled tightly through ter or will he open it up to contributors one day?**

For now, I have been extremely strict about who can touch the code base. Unlike most projects in academia, none of my graduate students have touched it. ;) I do welcome bug fixes and reports, which I require people to submit through a click-wrap license. This allows me to guarantee an untainted code base, which makes big companies much more comfortable using my software. I have added a page to describe some of the projects I would like people to work on however:

<http://www.antlr.org/summer-of-code/2007/index.html>

## **What's next for antlr?**

Incremental parsing is the first thing, which allows you to take any grammar and use it in an IDE. Then, AST construction from tree parsers (which is currently missing). I'm contemplating the idea of starting another book: "The ANTLR Cookbook: The taste of venison". 'course no publisher will let me get away with that subtitle, but hey it's funny. I am also going to work on a language textbook that focuses on language translation, as opposed to all of the other compiler textbooks.