The Pragmatic Programmers

The Definitive ANTLR Reference

Building Domain-Specific Languages

Terence Parr

What readers are saying about
The Definitive ANTLR Reference

Over the past few years ANTLR has proven itself as a solid parser generator. This book is a fine guide to making the best use of it.

► **Martin Fowler**
Chief Scientist, ThoughtWorks

The Definitive ANTLR Reference deserves a place in the bookshelf of anyone who ever has to parse or translate text. ANTLR is not just for language designers anymore.

► **Bob McWhirter**
Founder of the JBoss Rules Project (a.k.a. Drools), JBoss.org

Over the course of a career, developers move through a few stages of sophistication: becoming effective with a single programming language, learning which of several programming languages to use, and finally learning to tailor the language to the task at hand. This approach was previously reserved for those with an education in compiler development. Now, *The Definitive ANTLR Reference* reveals that it doesn't take a PhD to develop your own domain-specific languages, and you would be surprised how often it is worth doing. Take the next step in your career, and buy this book.

► **Neal Gafter**
Java Evangelist and Compiler Guru, Google (formerly at Sun Microsystems)

This book, especially the first section, really gave me a much better understanding of the principles of language recognition as a whole. I recommend this book to anyone without a background in language recognition looking to start using ANTLR or trying to understand the concept of language recognition.

► **Steve Ebersole**
Hibernate Lead Developer, Hibernate.org

Eclipse IDE users have become accustomed to cool features such as single-click navigation between symbol references and declarations, not to mention intelligent content assist. ANTLR v3 with its *LL(*)* parsing algorithm will help you immensely in building highly complex parsers to support these features. This book is a critical resource for Eclipse developers and others who want to take full advantage of the power of the new features in ANTLR.

► **Doug Schaefer**
Eclipse CDT Project Lead, Tools PMC Member, QNX Software Systems

Terence's new book is an excellent guide to ANTLR v3. It is very well written, with both the student and the developer in mind. The book does not assume compiler design experience. It provides the necessary background, from a pragmatic rather than a theoretical perspective, and it then eases the new user, whether someone with previous compiler design experience or not, into the use of the ANTLR tools. I recommend this book highly for anyone who needs to incorporate language capabilities into their software design.

► **Jesse Grodnik**
Software Development Manager, Sun Microsystems, Inc.

ANTLR v3 and *The Definitive ANTLR Reference* present a compelling package: an intuitive tool that handles complex recognition and translation tasks with ease and a clear book detailing how to get the most from it. The book provides an in-depth account of language translation utilizing the new powerful *LL(*)* parsing strategy. If you're developing translators, you can't afford to ignore this book!

► **Dermot O'Neill**
Senior Developer, Oracle Corporation

Whether you are a compiler newbie itching to write your own language or a jaded YACC veteran tired of shift-reduce conflicts, keep this book by your side. It is at once a tutorial, a reference, and an insider's viewpoint.

► **Sriram Srinivasan**
Formerly Principal Engineer, BEA/WebLogic

The Definitive ANTLR Reference

Building Domain-Specific Languages

Terence Parr

The Pragmatic Bookshelf
Raleigh, North Carolina Dallas, Texas

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragmaticprogrammer.com

Copyright © 2007 Terence Parr.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN-10: 0-9787392-5-6

ISBN-13: 978-09787392-4-9

Printed on acid-free paper with 85% recycled, 30% post-consumer content.

First printing, May 2007

Version: 2007-5-17

This is Tom's fault.

Contents

Acknowledgments

A researcher once told me after a talk I had given that "It was clear there was a single mind behind these tools." In reality, there are many minds behind the ideas in my language tools and research, though I'm a benevolent dictator with specific opinions about how ANTLR should work. At the least, dozens of people let me bounce ideas off them, and I get a lot of great ideas from the people on the ANTLR interest list.[1]

Concerning the ANTLR v3 tool, I want to acknowledge the following contributors for helping with the design and functional requirements: Sriram Srinivasan (Sriram had a knack for finding holes in my *LL(*)* algorithm), Loring Craymer, Monty Zukowski, John Mitchell, Ric Klaren, Jean Bovet, and Kay Roepke. Matt Benson converted all my unit tests to use JUnit and is a big help with Ant files and other goodies. Juergen Pfundt contributed the ANTLR v3 task for Ant. I sing Jean Bovet's praises every day for his wonderful ANTLRWorks grammar development environment. Next comes the troop of hardworking ANTLR language target authors, most of whom contribute ideas regularly to ANTLR:[2] Jim Idle, Michael Jordan (no not that one), Ric Klaren, Benjamin Niemann, Kunle Odutola, Kay Roepke, and Martin Traverso.

I also want to thank (then Purdue) professors Hank Dietz and Russell Quong for their support early in my career. Russell also played a key role in designing the semantic and syntactic predicates mechanism.

The following humans provided technical reviews: Mark Bednarczyk, John Mitchell, Dermot O'Neill, Karl Pfalzer, Kay Roepke, Sriram Srinivasan, Bill Venners, and Oliver Ziegermann. John Snyders, Jeff Wilcox, and Kevin Ruland deserve special attention for their amazingly detailed feedback. Finally, I want to mention my excellent development editor Susannah Davidson Pfalzer. She made this a much better book.

1. See http://www.antlr.org:8080/pipermail/antlr-interest/.
2. See http://www.antlr.org/wiki/display/ANTLR3/Code+Generation+Targets.

Preface

In August 1993, I finished school and drove my overloaded moving van to Minnesota to start working. My office mate was a curmudgeonly astrophysicist named Kevin, who has since become a good friend. Kevin has told me on multiple occasions that only physicists do real work and that programmers merely support physicists. Because all I do is build language tools to support programmers, I am at least two levels of indirection away from doing anything useful.[3] Now, Kevin also claims that Fortran 77 is a good enough language for anybody and, for that matter, that Fortran 66 is probably sufficient, so one might question his judgment. But, concerning my usefulness, he was right—I am fundamentally lazy and would much rather work on something that made other people productive than actually do anything useful myself. This attitude has led to my guiding principle:[4]

Why program by hand in five days what you can spend five years of your life automating?

Here's the point: The first time you encounter a problem, writing a formal, general, and automatic mechanism is expensive and is usually overkill. From then on, though, you are much faster and better at solving similar problems because of your automated tool. Building tools can also be much more fun than your real job. Now that I'm a professor, I have the luxury of avoiding real work for a living.

3. The irony is that, as Kevin will proudly tell you, he actually played solitaire for at least a decade instead of doing research for his boss—well, when he wasn't scowling at the other researchers, at least. He claimed to have a winning streak stretching into the many thousands, but one day Kevin was caught overwriting the game log file to erase a loss (apparently per his usual habit). A holiday was called, and much revelry ensued.

4. Even as a young boy, I was fascinated with automation. I can remember endlessly building model ships and then trying to motorize them so that they would move around automatically. Naturally, I proceeded to blow them out of the water with firecrackers and rockets, but that's a separate issue.

My passion for the last two decades has been ANTLR, ANother Tool for Language Recognition. ANTLR is a parser generator that automates the construction of language recognizers. It is a program that writes other programs.

From a formal language description, ANTLR generates a program that determines whether sentences conform to that language. By adding code snippets to the grammar, the recognizer becomes a translator. The code snippets compute output phrases based upon computations on input phrases. ANTLR is suitable for the simplest and the most complicated language recognition and translation problems. With each new release, ANTLR becomes more sophisticated and easier to use. ANTLR is extremely popular with 5,000 downloads a month and is included on all Linux and OS X distributions. It is widely used because it:

- Generates human-readable code that is easy to fold into other applications
- Generates powerful recursive-descent recognizers using *LL(*)*, an extension to *LL(k)* that uses arbitrary lookahead to make decisions
- Tightly integrates StringTemplate,[5] a template engine specifically designed to generate structured text such as source code
- Has a graphical grammar development environment called ANTLRWorks[6] that can debug parsers generated in any ANTLR target language
- Is actively supported with a good project website and a high-traffic mailing list[7]
- Comes with complete source under the BSD license
- Is extremely flexible and automates or formalizes many common tasks
- Supports multiple target languages such as Java, C#, Python, Ruby, Objective-C, C, and C++

Perhaps most importantly, ANTLR is much easier to understand and use than many other parser generators. It generates essentially what you would write by hand when building a recognizer and uses technology that mimics how your brain generates and recognizes language (see Chapter 2, *The Nature of Computer Languages*, on page 34).

5. See http://www.stringtemplate.org.
6. See http://www.antlr.org/works.
7. See http://www.antlr.org:8080/pipermail/antlr-interest/.

You generate and recognize sentences by walking their implicit tree structure, from the most abstract concept at the root to the vocabulary symbols at the leaves. Each subtree represents a phrase of a sentence and maps directly to a rule in your grammar. ANTLR's grammars and resulting top-down recursive-descent recognizers thus feel very natural. ANTLR's fundamental approach dovetails your innate language process.

Why a Completely New Version of ANTLR?

For the past four years, I have been working feverishly to design and build ANTLR v3, the subject of this book. ANTLR v3 is a completely rewritten version and represents the culmination of twenty years of language research. Most ANTLR users will instantly find it familiar, but many of the details are different. ANTLR retains its strong mojo in this new version while correcting a number of deficiencies, quirks, and weaknesses of ANTLR v2 (I felt free to break backward compatibility in order to achieve this). Specifically, I didn't like the following about v2:[8]

- The v2 lexers were very slow albeit powerful.
- There were no unit tests for v2.
- The v2 code base was impenetrable. The code was never refactored to clean it up, partially for fear of breaking it without unit tests.
- The linear approximate *LL(k)* parsing strategy was a bit weak.
- Building a new language target duplicated vast swaths of logic and print statements.
- The AST construction mechanism was too informal.
- A number of common tasks were not easy (such as obtaining the text matched by a parser rule).
- It lacked the semantic predicates hoisting of ANTLR v1 (PCCTS).
- The v2 license/contributor trail was loose and made big companies afraid to use it.

ANTLR v3 is my answer to the issues in v2. ANTLR v3 has a very clean and well-organized code base with lots of unit tests. ANTLR generates extremely powerful *LL(*)* recognizers that are fast and easy to read.

8. See http://www.antlr.org/blog/antlr3/antlr2.bashing.tml for notes on what people did not like about v2. ANTLR v2 also suffered because it was designed and built while I was under the workload and stress of a new start-up (jGuru.com).

Many common tasks are now easy by default. For example, reading in some input, tweaking it, and writing it back out while preserving whitespace is easy. ANTLR v3 also reintroduces semantic predicates hoisting. ANTLR's license is now BSD, and all contributors must sign a "certificate of origin."[9] ANTLR v3 provides significant functionality beyond v2 as well:

- Powerful *LL(*)* parsing strategy that supports more natural grammars and makes it easier to build them
- Auto-backtracking mode that shuts off all grammar analysis warnings, forcing the generated parser to simply figure things out at runtime
- Partial parsing result memoization to guarantee linear time complexity during backtracking at the cost of some memory
- Jean Bovet's ANTLRWorks GUI grammar development environment
- StringTemplate template engine integration that makes generating structured text such as source code easy
- Formal AST construction rules that map input grammar alternatives to tree grammar fragments, making actions that manually construct ASTs no longer necessary
- Dynamically scoped attributes that allow distant rules to communicate
- Improved error reporting and recovery for generated recognizers
- Truly retargetable code generator; building a new target is a matter of defining StringTemplate templates that tell ANTLR how to generate grammar elements such as rule and token references

This book also provides a serious advantage to v3 over v2. Professionally edited and complete documentation is a big deal to developers. You can find more information about the history of ANTLR and its contributions to parsing theory on the ANTLR website.[10,11]

Look for *Improved in v3* and *New in v3* notes in the margin that highlight improvements or additions to v2.

9. See http://www.antlr.org/license.html.
10. See http://www.antlr.org/history.html.
11. See http://www.antlr.org/contributions.html.

Who Is This Book For?

The primary audience for this book is the practicing software developer, though it is suitable for junior and senior computer science undergraduates. This book is specifically targeted at any programmer interested in learning to use ANTLR to build interpreters and translators for domain-specific languages. Beginners and experts alike will need this book to use ANTLR v3 effectively. For the most part, the level of discussion is accessible to the average programmer. Portions of Part III, however, require some language experience to fully appreciate. Although the examples in this book are written in Java, their substance applies equally well to the other language targets such as C, C++, Objective-C, Python, C#, and so on. Readers should know Java to get the most out of the book.

What's in This Book?

This book is the best, most complete source of information on ANTLR v3 that you'll find anywhere. The free, online documentation provides enough to learn the basic grammar syntax and semantics but doesn't explain ANTLR concepts in detail. This book helps you get the most out of ANTLR and is required reading to become an advanced user. In particular, Part III provides the only thorough explanation available anywhere of ANTLR's *LL(*)* parsing strategy.

This book is organized as follows. Part I introduces ANTLR, describes how the nature of computer languages dictates the nature of language recognizers, and provides a complete calculator example. Part II is the main reference section and provides all the details you'll need to build large and complex grammars and translators. Part III treks through ANTLR's predicated-*LL(*)* parsing strategy and explains the grammar analysis errors you might encounter. Predicated-*LL(*)* is a totally new parsing strategy, and Part III is essentially the only written documentation you'll find for it. You'll need to be familiar with the contents in order to build complicated translators.

Readers who are totally new to grammars and language tools should follow the chapter sequence in Part I as is. Chapter 1, *Getting Started with ANTLR*, on page 21 will familiarize you with ANTLR's basic idea; Chapter 2, *The Nature of Computer Languages*, on page 34 gets you ready to study grammars more formally in Part II; and Chapter 3, *A Quick Tour for the Impatient*, on page 59 gives your brain something

concrete to consider. Familiarize yourself with the ANTLR details in Part II, but I suggest trying to modify an existing grammar as soon as you can. After you become comfortable with ANTLR's functionality, you can attempt your own translator from scratch. When you get grammar analysis errors from ANTLR that you don't understand, then you need to dive into Part III to learn more about *LL(*)*.

Those readers familiar with ANTLR v2 should probably skip directly to Chapter 3, *A Quick Tour for the Impatient*, on page 59 to figure out how v3 differs. Chapter 4, *ANTLR Grammars*, on page 86 is also a good place to look for features that v3 changes or improves on.

If you are familiar with an older tool, such as YACC [Joh79], I recommend starting from the beginning of the book as if you were totally new to grammars and language tools. If you're used to JavaCC[12] or another top-down parser generator, you can probably skip Chapter 2, *The Nature of Computer Languages*, on page 34, though it is one of my favorite chapters.

I hope you enjoy this book and ANTLR v3 as much as I have enjoyed writing them!

Terence Parr
March 2007
University of San Francisco

12. See https://javacc.dev.java.net.

Part I

Introducing ANTLR and Computer Language Translation

Chapter 1

Getting Started with ANTLR

This is a reference guide for ANTLR: a sophisticated parser generator you can use to implement language interpreters, compilers, and other translators. This is not a compiler book, and it is not a language theory textbook. Although you can find many good books about compilers and their theoretical foundations, the vast majority of language applications are not compilers. This book is more directly useful and practical for building common, everyday language applications. It is densely packed with examples, explanations, and reference material focused on a single language tool and methodology.

Programmers most often use ANTLR to build translators and interpreters for *domain-specific languages* (DSLs). DSLs are generally very high-level languages tailored to specific tasks. They are designed to make their users particularly effective in a specific domain. DSLs include a wide range of applications, many of which you might not consider languages. DSLs include data formats, configuration file formats, network protocols, text-processing languages, protein patterns, gene sequences, space probe control languages, and domain-specific programming languages.

DSLs are particularly important to software development because they represent a more natural, high-fidelity, robust, and maintainable means of encoding a problem than simply writing software in a general-purpose language. For example, NASA uses domain-specific command languages for space missions to improve reliability, reduce risk, reduce cost, and increase the speed of development. Even the first Apollo guidance control computer from the 1960s used a DSL that supported vector computations.[1]

1. See http://www.ibiblio.org/apollo/assembly_language_manual.html.

This chapter introduces the main ANTLR components and explains how they all fit together. You'll see how the overall DSL translation problem easily factors into multiple, smaller problems. These smaller problems map to well-defined translation phases (lexing, parsing, and tree parsing) that communicate using well-defined data types and structures (characters, tokens, trees, and ancillary structures such as symbol tables). After this chapter, you'll be broadly familiar with all translator components and will be ready to tackle the detailed discussions in subsequent chapters. Let's start with the big picture.

1.1 The Big Picture

A translator maps each input sentence of a language to an output sentence. To perform the mapping, the translator executes some code you provide that operates on the input symbols and emits some output. A translator must perform different actions for different sentences, which means it must be able to recognize the various sentences.

Recognition is much easier if you break it into two similar but distinct tasks or phases. The separate phases mirror how your brain reads English text. You don't read a sentence character by character. Instead, you perceive a sentence as a stream of words. The human brain subconsciously groups character sequences into words and looks them up in a dictionary before recognizing grammatical structure. The first translation phase is called *lexical analysis* and operates on the incoming character stream. The second phase is called *parsing* and operates on a stream of vocabulary symbols, called *tokens*, emanating from the lexical analyzer. ANTLR automatically generates the lexical analyzer and parser for you by analyzing the grammar you provide.

Performing a translation often means just embedding *actions* (code) within the grammar. ANTLR executes an action according to its position within the grammar. In this way, you can execute different code for different phrases (sentence fragments). For example, an action within, say, an expression rule is executed only when the parser is recognizing an expression.

Some translations should be broken down into even more phases. Often the translation requires multiple passes, and in other cases, the translation is just a heck of a lot easier to code in multiple phases. Rather than reparse the input characters for each phase, it is more convenient to construct an intermediate form to pass between phases.

Language Translation Can Help You Avoid Work

In 1988, I worked in Paris for a robotics company. At the time, the company had a fairly demanding coding standard that required very formal and structured comments on each C function and file.

After finishing my compiler project, I was ready to head back to the United States and continue with my graduate studies. Unfortunately, the company was withholding my bonus until I followed its coding standard. The standard required all sorts of tedious information such as which functions were called in each function, the list of parameters, list of local variables, which functions existed in this file, and so on. As the company dangled the bonus check in front me, I blurted out, "All of that can be automatically generated!" Something clicked in my mind. Of course. Build a quick C parser that is capable of reading all my source code and generating the appropriate comments. I would have to go back and enter the written descriptions, but my translator would do the rest.

I built a parser by hand (this was right before I started working on ANTLR) and created template files for the various documentation standards. There were holes that my parser could fill in with parameters, variable lists, and so on. It took me two days to build the translator. I started it up, went to lunch, and came back to commented source code. I quickly entered the necessary descriptions, collected my bonus, and flew back to Purdue University with a smirk on my face.

The point is that knowing about computer languages and language technology such as ANTLR will make your coding life much easier. Don't be afraid to build a human-readable configuration file (I implore everyone to please stop using XML as a human interface!) or to build domain-specific languages to make yourself more efficient. Designing new languages and building translators for existing languages, when appropriate, is the hallmark of a sophisticated developer.

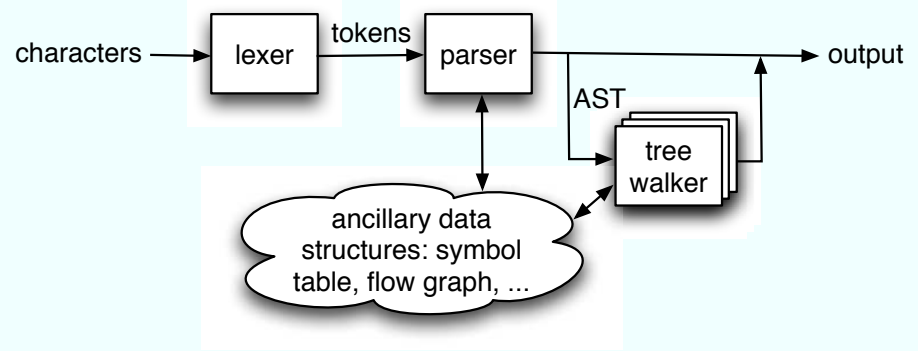

Figure 1.1: Overall translation data flow; edges represent data structure flow, and squares represent translation phases

This intermediate form is usually a tree data structure, called an *abstract syntax tree* (AST), and is a highly processed, condensed version of the input. Each phase collects more information or performs more computations. A final phase, called the *emitter*, ultimately emits output using all the data structures and computations from previous phases.

Figure 1.1 illustrates the basic data flow of a translator that accepts characters and emits output. The lexical analyzer, or *lexer*, breaks up the input stream into tokens. The parser feeds off this token stream and tries to recognize the sentence structure. The simplest translators execute actions that immediately emit output, bypassing any further phases.

Another kind of simple translator just constructs an internal data structure—it doesn't actually emit output. A configuration file reader is the best example of this kind of translator. More complicated translators use the parser only to construct ASTs. Multiple *tree parsers* (depth-first tree walkers) then scramble over the ASTs, computing other data structures and information needed by future phases. Although it is not shown in this figure, the final emitter phase can use templates to generate structured text output.

A template is just a text document with holes in it that an emitter can fill with values. These holes can also be expressions that operate on the incoming data values. ANTLR formally integrates the StringTemplate engine to make it easier for you to build emitters (see Chapter 9, *Generating Structured Text with Templates and Grammars*, on page 206).

sample content of The Definitive ANTLR Reference: Building Domain-Specific Languages (Pragmatic Programmers)

- *Memoirs of a Dance Hall Romeo pdf, azw (kindle), epub, doc, mobi*
- download online Masochism: Coldness and Cruelty & Venus in Furs
- The Korean War: An International History online
- read online Chicken: The New Classics
- click The Tartar Steppe

- http://cavalldecartro.highlandagency.es/library/Vampire-World-I--Blood-Brothers--Necroscope--Book-6-.pdf
- http://pittiger.com/lib/La-cueva-de-Salamanca--La-prueba-de-las-promesas.pdf
- http://conexdxb.com/library/Managing-Multiple-Sclerosis-Naturally--A-Self-Help-Guide-to-Living-with-MS.pdf
- http://musor.ruspb.info/?library/The-Liberty-Amendments--Restoring-the-American-Republic.pdf
- http://aseasonedman.com/ebooks/The-Tartar-Steppe.pdf