Extracted from:

The Definitive ANTLR Reference

Building Domain-Specific Languages

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.

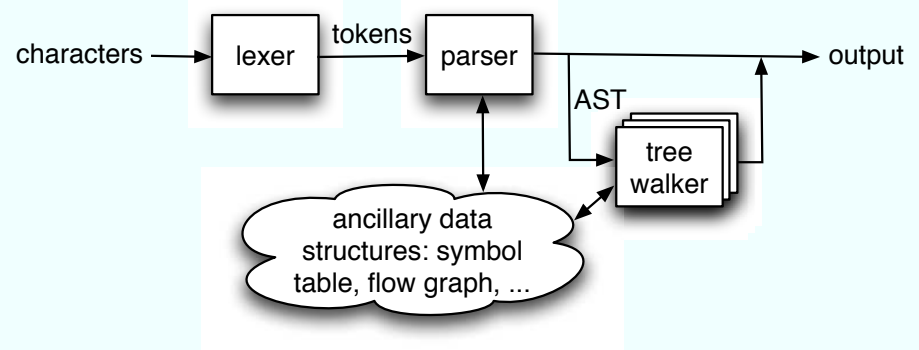


Figure 1.1: Overall translation data flow; edges represent data structure flow, and squares represent translation phases

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 208).

StringTemplate is a domain-specific language for generating structured text from internal data structures that has the flavor of an output grammar. Features include template group inheritance, template polymorphism, lazy evaluation, recursion, output autoindentation, and the new notions of group interfaces and template regions.[2] StringTemplate's feature set is driven by solving real problems encountered in complicated systems. Indeed, ANTLR makes heavy use of StringTemplate to translate grammars to executable recognizers. Each ANTLR language target is purely a set of templates and fed by ANTLR's internal retargetable code generator.

Now, let's take a closer look at the data objects passed between the various phases in Figure 1.1, on the previous page. Figure 1.2, on the following page, illustrates the relationship between characters, tokens, and ASTs. Lexers feed off characters provided by a CharStream such as ANTLRStringStream or ANTLRFileStream. These predefined streams assume that the entire input will fit into memory and, consequently, buffer up all characters. Rather than creating a separate string object per token, tokens can more efficiently track indexes into the character buffer.

Similarly, rather than copying data from tokens into tree nodes, ANTLR AST nodes can simply point at the token from which they were created. CommonTree, for example, is a predefined node containing a Token *payload*. The type of an ANTLR AST node is treated as an Object so that there are no restrictions whatsoever on your tree data types. In fact, you can even make your Token objects double as AST nodes to avoid extra object instantiations. The relationship between the data types described in Figure 1.2, on the next page, is very efficient and flexible.

The tokens in the figure with checkboxes reside on a hidden *channel* that the parser does not see. The parser *tunes* to a single channel and, hence, ignores tokens on any other channel. With a simple action in the lexer, you can send different tokens to the parser on different channels. For example, you might want whitespace and regular comments on one channel and Javadoc comments on another when parsing Java. The token buffer preserves the relative token order regardless of the token channel numbers. The token channel mechanism is an elegant solution to the problem of ignoring but not throwing away whitespace and comments (some translators need to preserve formatting and comments).

2. Please see http://www.stringtemplate.org for more details. I mention these terms to entice readers to learn more about StringTemplate.

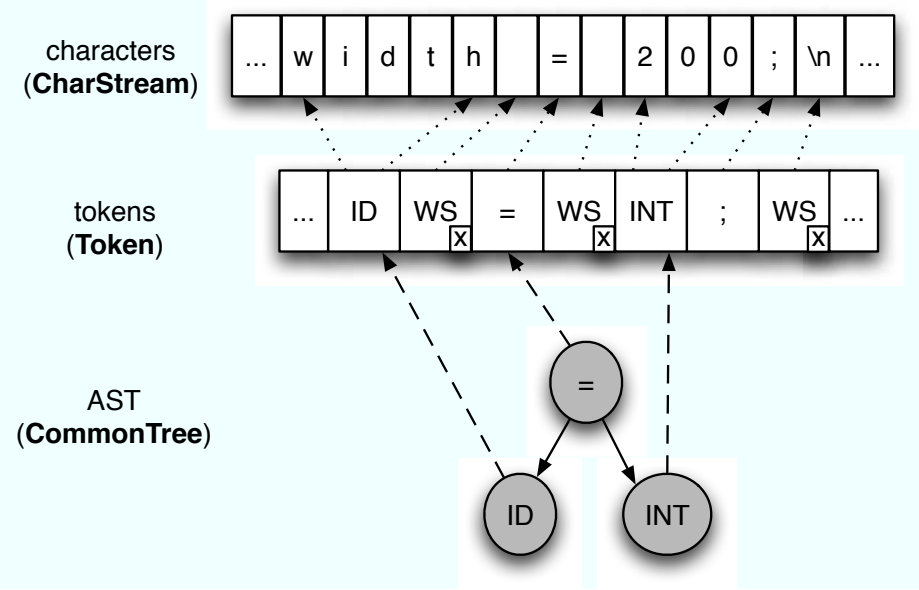


Figure 1.2: Relationship between characters, tokens, and ASTs; CharStream, Token, and CommonTree are ANTLR runtime types

As you work through the examples and discussions later in this book, it may help to keep in mind the analogy described in the next section.

1.2 An A-mazing Analogy

This book focuses primarily on two topics: the discovery of the implicit tree structure behind input sentences and the generation of structured text. At first glance, some of the language terminology and technology in this book will be unfamiliar. Don't worry. I'll define and explain everything, but it helps to keep in mind a simple analogy as you read.

Imagine a maze with a single entrance and single exit that has words written on the floor. Every path from entrance to exit generates a sentence by "saying" the words in sequence. In a sense, the maze is analogous to a grammar that defines a language.

You can also think of a maze as a sentence recognizer. Given a sentence, you can match its words in sequence with the words along the floor. Any sentence that successfully guides you to the exit is a valid sentence (a *passphrase*) in the language defined by the maze.

Language recognizers must discover a sentence's implicit tree structure piecemeal, one word at a time. At almost every word, the recognizer must make a decision about the interpretation of a phrase or subphrase. Sometimes these decisions are very complicated. For example, some decisions require information about previous decision choices or even future choices. Most of the time, however, decisions need just a little bit of *lookahead* information. Lookahead information is analogous to the first word or words down each path that you can see from a given fork in the maze. At a fork, the next words in your input sentence will tell you which path to take because the words along each path are different. Chapter 2, *The Nature of Computer Languages*, on page 36 describes the nature of computer languages in more detail using this analogy. You can either read that chapter first or move immediately to the quick ANTLR tour in Chapter 3, *A Quick Tour for the Impatient*, on page 61.

In the next two sections, you'll see how to map the big picture diagram in Figure 1.1, on page 26, into Java code and also learn how to execute ANTLR.

1.3 Installing ANTLR

ANTLR is written in Java, so you must have Java installed on your machine even if you are going to use ANTLR with, say, Python. ANTLR requires a Java version of 1.4 or higher. Before you can run ANTLR on your grammar, you must install ANTLR by downloading it[3] and extracting it into an appropriate directory. You do not need to run a configuration script or alter an ANTLR configuration file to properly install ANTLR. If you want to install ANTLR in /usr/local/antlr-3.0, do the following:

```
$ cd /usr/local
$ tar xvfz antlr-3.0.tar.gz
antlr-3.0/
antlr-3.0/build/
antlr-3.0/build.properties
antlr-3.0/build.xml
antlr-3.0/lib/
antlr-3.0/lib/antlr-3.0.jar
...
$
```

3. See http://www.antlr.org/download.html.

Pragmatic Projects

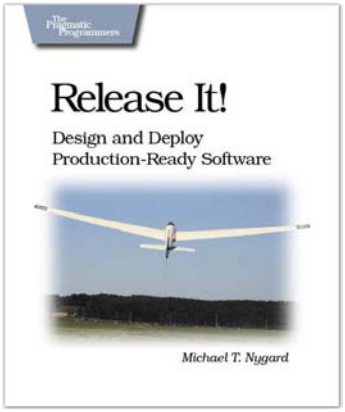
Your application is feature complete, but is it ready for the real world? See how to design and deploy production-ready software and *Release It!*.

Have you ever noticed that project retrospectives feel too little, too late? What you need to do is start having *Agile Retrospectives*.

Release It!

Whether it's in Java, .NET, or Ruby on Rails, getting your application ready to ship is only half the battle. Did you design your system to survive a sudden rush of visitors from Digg or Slashdot? Or an influx of real world customers from 100 different countries? Are you ready for a world filled with flakey networks, tangled databases, and impatient users?

If you're a developer and don't want to be on call at 3AM for the rest of your life, this book will help.

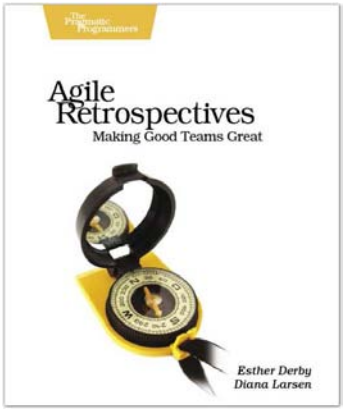


Design and Deploy Production-Ready Software
Michael T. Nygard
(368 pages) ISBN: 0-9787392-1-3. $34.95
http://pragmaticprogrammer.com/titles/mnee

Agile Retrospectives

Mine the experience of your software development team continually throughout the life of the project. Rather than waiting until the end of the project—as with a traditional retrospective, when it's too late to help—agile retrospectives help you adjust to change *today*.

The tools and recipes in this book will help you uncover and solve hidden (and not-so-hidden) problems with your technology, your methodology, and those difficult "people issues" on your team.



Agile Retrospectives: Making Good Teams Great
Esther Derby and Diana Larsen
(170 pages) ISBN: 0-9776166-4-9. $29.95
http://pragmaticprogrammer.com/titles/dlret

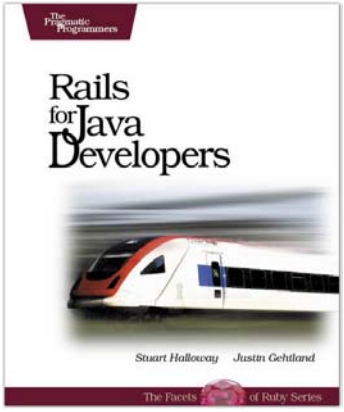
Rails and More

If you know Java, and are curious about Ruby on Rails, you don't have to start from scratch. Read *Rails for Java Developers* and get a head start on this exciting new technology.

And whatever language you use, you'll need a good text editor, too. On the Mac, we recommend TextMate.

Rails for Java Developers

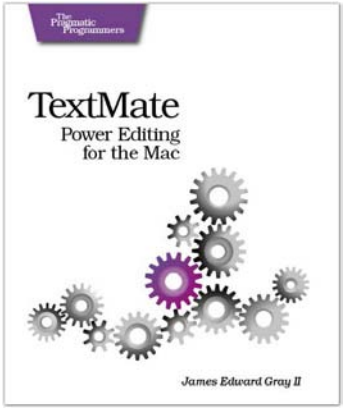
Enterprise Java developers already have most of the skills needed to create Rails applications. They just need a guide which shows how their Java knowledge maps to the Rails world. That's what this book does. It covers: • The Ruby language • Building MVC Applications • Unit and Functional Testing • Security • Project Automation • Configuration • Web Services This book is the fast track for Java programmers who are learning or evaluating Ruby on Rails.



Rails for Java Developers
Stuart Halloway and Justin Gehtland
(300 pages) ISBN: 0-9776166-9-X. $34.95
http://pragmaticprogrammer.com/titles/fr_r4j

TextMate

If you're coding Ruby or Rails on a Mac, then you owe it to yourself to get the TextMate editor. And, once you're using TextMate, you owe it to yourself to pick up this book. It's packed with information which will help you automate all your editing tasks, saving you time to concentrate on the important stuff. Use snippets to insert boilerplate code and refactorings to move stuff around. Learn how to write your own extensions to customize it to the way you work.



TextMate: Power Editing for the Mac
James Edward Gray II
(200 pages) ISBN: 0-9787392-3-X. $29.95
http://pragmaticprogrammer.com/titles/textmate

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

The Definitive ANTLR Reference
http://pragmaticprogrammer.com/titles/tpantlr
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragmaticprogrammer.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragmaticprogrammer.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragmaticprogrammer.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragmaticprogrammer.com/titles/tpantlr.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragmaticprogrammer.com/catalog
Customer Service:	orders@pragmaticprogrammer.com
Non-English Versions:	translations@pragmaticprogrammer.com
Pragmatic Teaching:	academic@pragmaticprogrammer.com
Author Proposals:	proposals@pragmaticprogrammer.com