Extracted from:

Language Implementation Patterns

Create Your Own Domain-Specific and General Programming Languages

The Pragmatic Programmers

Language Implementation Patterns

Create Your Own Domain-Specific and General Programming Languages

Terence Parr

Edited by Susannah Davidson Pfalzer

With permission of the creator we hereby publish the chess images in Chapter 11 under the following licenses:

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.pragprog.com

Printed in the United States of America.

ISBN-10: 1-934356-45-X

ISBN-13: 978-1-934356-45-6

Printed on acid-free paper.

P1.0 printing, December 2009

Version: 2010-1-13

Preface

The more language applications you build, the more patterns you'll see. The truth is that the architecture of most language applications is freakishly similar. A broken record plays in my head every time I start a new language application: "First build a syntax recognizer that creates a data structure in memory. Then sniff the data structure, collecting information or altering the structure. Finally, build a report or code generator that feeds off the data structure." You even start seeing patterns within the tasks themselves. Tasks share lots of common algorithms and data structures.

Once you get these language implementation design patterns and the general architecture into your head, you can build pretty much whatever you want. If you need to learn how to build languages pronto, this book is for you. It's a pragmatic book that identifies and distills the common design patterns to their essence. You'll learn why you need the patterns, how to implement them, and how they fit together. You'll be a competent language developer in no time!

Building a new language doesn't require a great deal of theoretical computer science. You might be skeptical because every book you've picked up on language development has focused on compilers. Yes, building a compiler for a general-purpose programming language requires a strong computer science background. But, most of us don't build compilers. So, this book focuses on the things that we build all the time: configuration file readers, data readers, model-driven code generators, source-to-source translators, source analyzers, and interpreters. We'll also code in Java rather than a primarily academic language like Scheme so that you can directly apply what you learn in this book to real-world projects.

What to Expect from This Book

This book gives you just the tools you'll need to develop day-to-day language applications. You'll be able to handle all but the really advanced or esoteric situations. For example, we won't have space to cover topics such as machine code generation, register allocation, automatic garbage collection, thread models, and extremely efficient interpreters. You'll get good all-around expertise implementing modest languages, and you'll get respectable expertise in processing or translating complex languages.

This book explains how existing language applications work so you can build your own. To do so, we're going to break them down into a series of well-understood and commonly used patterns. But, keep in mind that this book is a learning tool, not a library of language implementations. You'll see many sample implementations throughout the book, though. Samples make the discussions more concrete and provide excellent foundations from which to build new applications.

It's also important to point out that we're going to focus on building applications for languages that already exist (or languages you design that are very close to existing languages). Language design, on the other hand, focuses on coming up with a syntax (a set of valid sentences) and describing the complete semantics (what every possible input means). Although we won't specifically study how to design languages, you'll actually absorb a lot as we go through the book. A good way to learn about language design is to look at lots of different languages. It'll help if you research the history of programming languages to see how languages change over time.

When we talk about language applications, we're not just talking about *implementing* languages with a compiler or interpreter. We're talking about any program that processes, analyzes, or translates an input file. Implementing a language means building an application that executes or performs tasks according to sentences in that language. That's just one of the things we can do for a given language definition. For example, from the definition of C, we can build a C compiler, a translator from C to Java, or a tool that instruments C code to isolate memory leaks. Similarly, think about all the tools built into the Eclipse development environment for Java. Beyond the compiler, Eclipse can refactor, reformat, search, syntax highlight, and so on.

You can use the patterns in this book to build language applications for any computer language, which of course includes domain-specific languages (DSLs). A domain-specific language is just that: a computer language designed to make users particularly productive in a specific domain. Examples include Mathematica, shell scripts, wikis, UML, XSLT, makefiles, PostScript, formal grammars, and even data file formats like comma-separated values and XML. The opposite of a DSL is a general-purpose programming language like C, Java, or Python. In the common usage, DSLs also typically have the connotation of being smaller because of their focus. This isn't always the case, though. SQL, for example, is a lot bigger than most general-purpose programming languages.

How This Book Is Organized

This book is divided into four parts:

- *Getting Started with Parsing*: We'll start out by looking at the overall architecture of language applications and then jump into the key language recognition (parsing) patterns.
- *Analyzing Languages*: To analyze DSLs and programming languages, we'll use parsers to build trees that represent language constructs in memory. By walking those trees, we can track and identify the various symbols (such as variables and functions) in the input. We can also compute expression result-type information (such as **int** and **float**). The patterns in this part of the book explain how to check whether an input stream makes sense.
- *Building Interpreters*: This part has four different interpreter patterns. The interpreters vary in terms of implementation difficulty and run-time efficiency.
- *Translating and Generating Languages*: In the final part, we will learn how to translate one language to another and how to generate text using the StringTemplate template engine. In the final chapter, we'll lay out the architecture of some interesting language applications to get you started building languages on your own.

The chapters within the different parts proceed in the order you'd follow to implement a language. Section 1.2, *A Tour of the Patterns*, on page 24 describes how all the patterns fit together.

What You'll Find in the Patterns

There are 31 patterns in this book. Each one describes a common data structure, algorithm, or strategy you're likely to find in language applications. Each pattern has four parts:

- *Purpose*: This section briefly describes what the pattern is for. For example, the purpose of Pattern 21, *Automatic Type Promotion*, on page 210 says "...how to automatically and safely promote arithmetic operand types." It's a good idea to scan the Purpose section before jumping into a pattern to discover exactly what it's trying to solve.
- *Discussion*: This section describes the problem in more detail, explains when to use the pattern, and describes how the pattern works.
- *Implementation*: Each pattern has a sample implementation in Java (possibly using language tools such as ANTLR). The sample implementations are not intended to be libraries that you can immediately apply to your problem. They demonstrate, in code, what we talk about in the Discussion sections.
- *Related Patterns*. This section lists alternative patterns that solve the same problem or patterns we depend on to implement this pattern.

The chapter introductory materials and the patterns themselves often provide comparisons between patterns to keep everything in proper perspective.

Who Should Read This Book

If you're a practicing software developer or computer science student and you want to learn how to implement computer languages, this book is for you. By computer language, I mean everything from data formats, network protocols, configuration files, specialized math languages, and hardware description languages to general-purpose programming
languages.

You don't need a background in formal language theory, but the code and discussions in this book assume a solid programming background.

To get the most out of this book, you should be fairly comfortable with recursion. Many algorithms and processes are inherently recursive. We'll use recursion to do everything from recognizing input, walking trees, and building interpreters to generating output.

How to Read This Book

If you're new to language implementation, start with Chapter 1, *Language Applications Cracked Open*, on page 22 because it provides an architectural overview of how we build languages. You can then move on to Chapter 2, *Basic Parsing Patterns*, on page 39 and Chapter 3, *Enhanced Parsing Patterns*, on page 67 to get some background on grammars (formal language descriptions) and language recognition.

If you've taken a fair number of computer science courses, you can skip ahead to either Chapter 4, *Building Intermediate Form Trees*, on page 90 or Chapter 5, *Walking and Rewriting Trees*, on page 118. Even if you've built a lot of trees and tree walkers in your career, it's still worth looking at Pattern 14, *Tree Grammar*, on page 136 and Pattern 15, *Tree Pattern Matcher*, on page 140.

If you've done some basic language application work before, you already know how to read input into a handy tree data structure and walk it. You can skip ahead to Chapter 6, *Tracking and Identifying Program Symbols*, on page 148 and Chapter 7, *Managing Symbol Tables for Data Aggregates*, on page 172, which describe how to build symbol tables. Symbol tables answer the question "What is *x*?" for some input symbol *x*. They are necessary data structures for the patterns in Chapter 8, *Enforcing Static Typing Rules*, on page 198, for example.

More advanced readers might want to jump directly to Chapter 9, *Building High-Level Interpreters*, on page 234 and Chapter 12, *Generating DSLs with Templates*, on page 325. If you really know what you're doing, you can skip around the book looking for patterns of interest. The truly impatient can grab a sample implementation from a pattern and use it as a kernel for a new language (relying on the book for explanations).

If you bought the e-book version of this book, you can click the gray boxes above the code samples to download code snippets directly. If you'd like to participate in conversations with me and other readers, you can do so at the web page for this book[1] or on the ANTLR user's

1. http://www.pragprog.com/titles/tpdsl

list.[2] You can also post book errata and download all the source code on the book's web page.

Languages and Tools Used in This Book

The code snippets and implementations in this book are written inJava, but their substance applies equally well to any other general programming language. I had to pick a single programming language for consistency. Java is a good choice because it's widely used in industry.[3,4] Remember, this book is about design patterns, not "language recipes." You can't just download a pattern's sample implementation and apply it to your problem without modification.

We'll use state-of-the-art language tools wherever possible in this book. For example, to recognize (parse) input phrases, we'll use aparser generator (well, that is, after we learn how to build parsers manually in Chapter 2, *Basic Parsing Patterns*, on page 39). It's no fair using a parser generator until you know how parsers work. That'd be like using a calculator before learning to do arithmetic. Similarly, once we know how to build tree walkers by hand, we can let a tool build them for us.

In this book, we'll use ANTLR extensively. ANTLR is a parser generator and tree walker generator that I've honed over the past two decades while building language applications. I could have used any similar language tool, but I might as well use my own. My point is that this book is not about ANTLR itself—it's about the design patterns common to most language applications. The code samples merely help you to understand the patterns.

We'll also use a template engine called StringTemplate a lot in Chapter 12, *Generating DSLs with Templates*, on page 325 to generate output. StringTemplate is like an "unparser generator," and templates are like output grammar rules. The alternative to a template engine would be to use an unstructured blob of generation logic interspersed with print statements.

You'll be able to follow the patterns in this book even if you're not familiar with ANTLR and StringTemplate. Only the sample implementations use them. To get the most out of the patterns, though, you should walk

2. http://www.antlr.org/support.html
3. http://langpop.com
4. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

through the sample implementations. To really understand them, it's a good idea to learn more about the ANTLR project tools. You'll get a taste in Section 4.3, *Quick Introduction to ANTLR*, on page 101. You can also visit the website to get documentation and examples or purchase *The Definitive ANTLR Reference* [Par07] (shameless plug).

One way or another, you're going to need language tools to implement languages. You'll have no problem transferring your knowledge to other tools after you finish this book. It's like learning to fly—you have no choice but to pick a first airplane. Later, you can move easily to another airplane. Gaining piloting skills is the key, not learning the details of a particular aircraft cockpit.

I hope this book inspires you to learn about languages and motivates you to build domain-specific languages (DSLs) and other language tools to help fellow programmers.

Terence Parr
December 2009
`parrt@cs.usfca.edu`

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

Language Implementation Patterns' Home Page
http://pragprog.com/titles/tpdsl
Source code from this book, errata, and other resources. Come give us feedback, too!

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

Join the Community
http://pragprog.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://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

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

Contact Us

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