Book reviews

Language Implementation Patterns: Create your own Domain-Specific and General Programming Languages, by Terence Parr, Pragmatic Bookshelf, http://www.pragprog.com, ISBN 9781934356456
doi:10.1017/S0956796810000298

The Pragmatic Bookshelf certainly lives up to its name by publishing Terence Parr's *Language Implementation Patterns*. The Gang Of Four rolled into one, his mission in this book is to provide people at or even below the level of undergraduates with just enough compiler construction savvy to be able to solve low to medium complexity translation problems. And as becomes quickly evident in this book: there are plenty of such problems around, waiting for a solution.

Clearly, relatively few people construct a compiler for a general programming language, but as the subtitle makes clear, the material in this book can be used by domain experts to learn how to construct their own parsers, type checkers and interpreters. The alternative would be to embed such a language in a (general) host language, but that solution is not pursued in this book.

The book clearly follows the compiler pipeline, but Parr's approach is refreshingly different. Where most compiler construction books focus on compiling general programming languages to a suitable backend, he recognizes that translation problems occur everywhere. Since sophistication often comes at the price of increased computational demands and implementation complexity, it makes sense to discuss not one, but a sequence of solutions to, say, programming tree traversals. Each solution comes with a description when it applies, and how it may be implemented. It is up to the reader of the book to make a correct estimation of the generality of his problem, and to choose the easiest solution that will solve the problem.

The phases that are considered are the usual suspects: two chapters on parsing, one on abstract syntax trees, four chapters on analysis, two chapters on interpretation and finally a few chapters that illustrate how to build a small variety of translators. Compared to most compiler books, Parr spends quite some time on code generation and interpretation. Again, there is a strong tendency to reuse what is there: why spend time on register allocation when you can map to LLVM (Lattner 2010).[1]

Parr's tone and style fit in well with his pragmatic approach: direct and hands-on. Background is introduced when necessary. Pointers to issues on the side are given, but only when the problem at hand makes it necessary. More than for ordinary compiler construction books, it is very important to actually try the examples given in the book, and introduce your own variations on them. You really have to get your hands dirty. Fortunately, all the examples in the book come with an implementation. All the ones I tried compiled without a hitch, and were easy to get to run.

Does that leave anything to complain about? Yes, it does. The first bad news, and particularly hard on people who regularly visit the pages of *JFP*, is that the focus is fully on imperative, object-oriented programming. First of all, Java is used exclusively in the implementations while the more complex patterns are implemented in ANTLR (Parr 2007);

[1] Although the implementation is not discussed in the book, you can find it at http://www.antlr.org/wiki/display/ANTLR3/LLVM.

this is not surprising since the ANTLR website proclaims that "Terence Parr is the maniac behind ANTLR". Moreover, the translation problems considered never touch upon issues that are close to the functional programmer's heart, including higher-order functions, closures and let-polymorphism. If you happen to want to build DSLs that borrow concepts specific to functional languages, you will not find that kind of information here. In fact, the only hint of 'functional programming' that I managed to find is the mention of Daan Leijen's Parsec combinators, see, e.g., Leijen & Meijer (2001).

Parr's take on static analysis deals for the most part with manipulating symbol tables, and a bit of *ad hoc* type checking. I understand that to keep things simple, he does not want to tackle issues such as dataflow analysis. But in view of the intended audience it might be useful to let the reader know that such a field does exist and where to start looking for more information (Aho *et al.* 1986; Grune *et al.* 2000; Nielson *et al.* 2005; Khedkar *et al.* 2009); even DSLs sometimes can do with a bit of optimisation. I also think that by providing a hands-on introduction to dataflow analysis in the style of this book, Parr could have provided a useful service to the field of program analysis. Although most treatments of dataflow analysis are mathematically more challenging than the material that Parr considers in this book, I do believe a good intuition of the basic notions, e.g., fixed-point iteration and monotonicity, can be provided without going very much into technical details, and an implementation in terms of a simple work-flow algorithm is not harder than some of the code already in the book.

What I find more problematic however, is that attribute grammars (Knuth 1968; Aho *et al.* 1986; Grune *et al.* 2000; Swierstra *et al.* 2010), as a particularly simple way of implementing the forms of static analysis he does in fact consider, are never mentioned at all, not even Java implementations such as JastAdd (Hedin & Magnusson 2003). In my opinion, the declarativity of attribute grammars can greatly simplify the bookkeeping that is involved in these straightforward and tedious forms of static analysis.

The book is not without mistakes; hardly any book is. What is important though is that the Pragmatic Bookshelf take mistakes seriously. When you visit their website you can easily find a list of known mistakes for the book, categorised by type of mistake (Parr 2009). This allows you to quickly see whether a technical mistake you may have spotted has been found, and where you can contribute any issue that you have with the book.

Summarizing, *Language Implementation Patterns* is a well-written book that even undergraduates can use to teach themselves how to build all kinds of translators, *if* they are willing to take the Java/ANTLR focus for granted. Indeed, unfamiliarity with Java and even ANTLR, is a serious handicap in appreciating this book. When it comes to translating functional programming languages, and anonymous functions, let-polymorphism, and higher-order functions kick in, this book offers the reader little help. Still, the patterns that are described will be useful in the context of most translation problems including those that involve, say, implementing DSLs founded on the functional programming paradigm.

References

Aho, A., Sethi, R. & Ullman, J. D. (1986) *Compilers: Principles, Techniques, and Tools.* Addison Wesley.

Grune, D., Bal, H. E., Jacobs, C. J. H. & Langendoen, K. G. (2000). *Modern Compiler Design.* Worldwide Series in Computer Science. Wiley.

Hedin, G. & Magnusson, E. (2003) The JastAdd system – an aspect-oriented compiler construction system, *Sci. comput. program.,* **47**(1): 37–58. http://www.cs.lth.se/~gorel/publications/2003-JastAdd-SCP-Preprint.pdf.

Khedkar, U., Sanyal, A. & Karkare, B. (2009) *Data Flow Analysis: Theory and Practice.* 1st ed. CRC Press.

Knuth, D. E. (1968) Semantics of context-free languages, *Theory comput. syst.*, **2**(2): 127–145.

Lattner, C. (2010) *The LLVM Compiler Infrastructure*. http://llvm.org.

Leijen, D. & Meijer, E. (2001). *Parsec: Direct style Monadic Parser Combinators for the Real World*. Technical Report UU-CS-2001-35. Department of Computer Science, Utrecht University. http://www.cs.uu.nl/~daan/parsec.html.

Nielson, F., Nielson, H. R. & Hankin, C. (2005) *Principles of Program Analysis*. 2nd printing edn. Springer Verlag.

Parr, T. (2007) *The Definitive Antlr Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf. http://www.pragprog.com/titles/tpantlr/the-definitive-antlr-reference.

Parr, T. (2009) *Language Implementation Patterns*. The Pragmatic Bookshelf. http://www.pragprog.com/titles/tpdsl/language-implementation-patterns.

Swierstra, S. D., Rodriguez, A., Middelkoop, A., Baars, A. I. & et al., A. Loeh. (2010) *The Haskell Utrecht Tools (hut)*. http://www.cs.uu.nl/wiki/HUT/WebHome.

JURRIAAN HAGE
s.j.thompson@ukc.ac.uk

Foundations of F# Robert Pickering, Apress, 2007 ISBN 10: 1-59059-757-5

doi:10.1017/S0956796810000110

The fact that Microsoft has created a functional programming language on top of the .NET platform enables a very large community of programmers to experiment with functional programming in an environment that they are familiar with. If they decide that functional programming is of value for them, it will be a big advantage that it can be deployed on a platform that is known and trusted. Especially in bigger commercial companies it is considered quite an investment to start operating a new platform, and the fact that F# runs on the .NET platform is really going to make a difference there. As an extra bonus there is the huge set of supporting tools and libraries that programmers will have access to.

Since F# can be downloaded from Microsoft's website for free, all that is needed for a .NET programmer to give it a try is a good description of the language and the way it can be put to use. For most people it will be difficult to get going without something like a book they can hold in their hand, read on the train, put little slips into as bookmarks etc – just the online manuals simply aren't enough. "Foundations of F#" by Robert Pickering seems to be aimed at this category of users.

After a short introduction, the book dedicates three chapters on what could indeed be called the "foundations" of F#. The chapters describe the Functional, Imperative and Object Oriented aspects of the language in a fairly formal way. It even starts with a list of the keywords, which maybe a bit too fundamental for practical purposes. Fortunately the rest of these chapters find a good balance between concise description of the language, some examples, and some general comments on functional programming.

In general the book doesn't waste much time explaining concepts like variable scope, or why you might want to use a relational database. The information in its 385 pages is relevant for a professional programmer who understands the principal concepts of computer programming. At the same time it doesn't assume that the reader understands typical functional programming constructs like currying. As a consequence, the level is suited for experienced programmers who would like to start with a functional programming language.

The examples provide good bits of functional programming "jargon", but they can sometimes be a bit difficult to follow. The use of variable names that are sometimes rather short and cryptic doesn't help here. In a similar vein, I am not sure that it is such a good idea