Book review of
Andrew W. Appel – Modern Compiler Implementation in Java
Cambridge University Press, 1998, ISBN 0 521 58388 8, x+548 pages, hardback.

Andrew W. Appel's new series of books *Modern Compiler Implementation* contains a lucent explanation of the *full employment theorem for compiler writers*: No matter what optimizing compiler we consider, there must always exist another (usually bigger) compiler that does a better job. In real life, Appel adds to this, we improve a compiler by finding some reasonably general program transformation that improves the performance of many programs. We add this transformation to the optimizer's "bag of tricks" and we get a more competent compiler. When our compiler knows enough tricks, we deem it *mature*.

A corollary of the theorem for compiler writers is full employment for compiler *book* writers. Appel took up the challenge, and added a highly practical and readable book to the compiler literature. In fact, he did not restrict himself to just one, but he added three at once: *Modern Compiler Implementation in Java* is also available in C and ML. The basic text of the three books is the same, but the programming exercises and examples are given in different languages.

Obviously, Java, C, and ML are not just different languages: they are representatives of the object-oriented, imperative and functional programming paradigms. This has left some interesting traces throughout the text. An example is the discussion of symbol tables: Appel explains how in an imperative style the insert and delete operators are destructive, whereas in a functional style these operators would return a new table, making it unnecessary to have explicit operators to mark the beginning and end of symbol scopes.

An interesting question about the Java version of the book is whether it is "object-oriented". At page 99 of the book, Appel addresses exactly this issue. It may well be, however, that Appel is not entirely objective: the motto given at the beginning of the chapter about the compilation of object-oriented languages is a definition of the word "object": *to feel distaste for something*. Appel explains why he uses a non-object-oriented style for representing abstract syntax trees: the crucial question is where to put the methods implementing typical compiler operations, such as typechecking, translation to various target machines, optimizations, etc. In a compiler setting, Appel argues, one wants these aspects to be isolated and separated from each other, rather than having them spread across all the classes representing different kinds of abstract syntax tree nodes.

An object-oriented theme not mentioned is design patterns. Nevertheless, the *visitor* pattern would have been of help to Appel, for example for type checking expressions. In order to determine the syntactic construct used in an expression, Appel makes use of explicit class membership testing, which is generally considered a violation of the object-oriented paradigm. With the visitor pattern, this is not necessary: it implements a traversal of a tree, passing an object through every node of the tree. At each type of node, a specific method is invoked, making class membership tests unnecessary. The typechecker will be a refinement of this visitor class, implementing specific type checking operations for each sort of node. Moreover, the same pattern can be used to implement other tasks, such as pretty printing, or translation to intermediate code.

The book itself covers all important phases of a compiler, as well as a selection of advanced compiler implementation topics. The running example of the book is a compiler for a simple language called *Tiger*. In the first part of the book, each chapter deals with one of the phases of the *Tiger* compiler, going from lexical analysis via activation records up to liveness analysis. Compared to other books on compiler construction, the chapters on syntactic analysis are relatively short, emphasizing the *use* of parser generators in favor of the underlying theory. In return for that, Appel devoted extra space to the compiler's backend, resulting in elaborate coverage of, e.g., register allocation. Sometimes the *Tiger* language is too simple to explain certain concepts: in such cases Appel explains how an industrial-strength compiler would handle a particular problem in a more realistic manner.

The aim of the book is to give the reader hands-on experience with compiler writing. With each chapter, a series of Java classes is available from the author's web site, implementing the key concepts discussed in the chapter. The reader can participate in a compiler implementation project, using the available classes to carry out a number of proposed programming assignments. Some assignments go into considerable detail: an example is the handling of *spilling*, which is needed if the target machine does not have enough registers to hold all live variables at a given program point, making it necessary to store intermediate values in memory.

The second part of the book deals with a selection of advanced compiler construction topics, such as garbage collection, compilation of object-oriented and functional languages, loop optimizations, etc. In several chapters, an appropriate extension of the *Tiger* language is made, such as *Object-Tiger*, *PureFun-Tiger*, *Lazy-Tiger*, etc. These are then used to explain issues such as efficient method lookup under multiple inheritance, tail recursion removal, deforestation, strictness analysis, type inference, and so on. Other chapters deal with the internals of the compiler, covering such topics as data flow analysis, static single assignment forms, and pipelining and scheduling. In these chapters, the Tiger language is less prominently present, and no programming exercises are given.

The book can be used at universities in various ways: for a one-semester course introducing the essentials of a compiler, the material of the first part could be used, with students implementing the Tiger compiler. For an advanced or graduate course, all or a selection of the chapters from the second part could be used, depending on the interests and taste of the lecturer. The book is not entirely free of errors: the author has a list of errata available at his home page.

Last but not least, this book is a pleasure to read and study. If you want to refresh or update your knowledge of compiler implementation topics, this book is warmly recommended.

Arie van Deursen
CWI, Amsterdam, The Netherlands