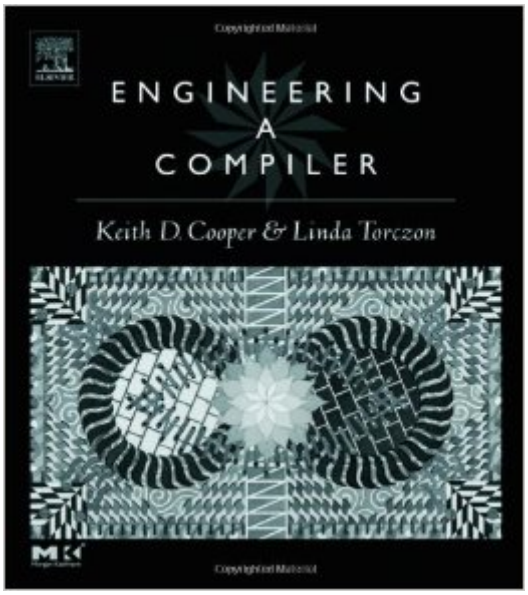
Engineering A Compiler



Synopsis

The proliferation of processors, environments, and constraints on systems has cast compiler technology into a wider variety of settings, changing the compiler and compiler writer's role. No longer is execution speed the sole criterion for judging compiled code. Today, code might be judged on how small it is, how much power it consumes, how well it compresses, or how many page faults it generates. In this evolving environment, the task of building a successful compiler relies upon the compiler writer's ability to balance and blend algorithms, engineering insights, and careful planning. Today's compiler writer must choose a path through a design space that is filled with diverse alternatives, each with distinct costs, advantages, and complexities. Engineering a Compiler explores this design space by presenting some of the ways these problems have been solved, and the constraints that made each of those solutions attractive. By understanding the parameters of the problem and their impact on compiler design, the authors hope to convey both the depth of the problems and the breadth of possible solutions. Their goal is to cover a broad enough selection of material to show readers that real tradeoffs exist, and that the impact of those choices can be both subtle and far-reaching. Authors Keith Cooper and Linda Torczon convey both the art and the science of compiler construction and show best practice algorithms for the major passes of a compiler. Their text re-balances the curriculum for an introductory course in compiler construction to reflect the issues that arise in current practice. Â Focuses on the back end of the compilerâ •reflecting the focus of research and development over the last decade. Â Uses the well-developed theory from scanning and parsing to introduce concepts that play a critical role in optimization and code generation. Â Introduces the student to optimization through data-flow analysis, SSA form, and a selection of scalar optimizations. Â Builds on this background to teach modern methods in code generation: instruction selection, instruction scheduling, and register allocation. Â Presents examples in several different programming languages in order to best illustrate the concept. Â Provides end-of-chapter exercises, with on-line solutions available to instructors.

Book Information

Hardcover: 801 pages
Publisher: Morgan Kaufmann; 1 edition (November 10, 2003)
Language: English
ISBN-10: 155860698X
ISBN-13: 978-1558606982

Product Dimensions: 9.5 x 8 x 1.4 inches

Shipping Weight: 4.2 pounds

Average Customer Review: 3.5 out of 5 starsÂ Â See all reviewsÂ (13 customer reviews)

Best Sellers Rank: #784,708 in Books (See Top 100 in Books) #51 inÂ Books > Computers & Technology > Programming > Languages & Tools > Compiler Design #102 inÂ Books > Computers & Technology > Hardware & DIY > Microprocessors & System Design > Computer Design #149 inÂ Books > Computers & Technology > Programming > Languages & Tools > Compilers

Customer Reviews

This is really a super compiler text. It is also one of the most recent compiler books you can buy.First of all this is a theoretical book. If you read the title 'Engineering a compiler' as 'Coding/Building a compiler' you would be disappointed! So, if you're looking for a learing-by-coding book, this is not for you (but I have some recommendations at the end of this review in the latest paragraph). The difference with most of the other theoretical books is that this book is not a dry text. It has also a nice layout. It gives plenty of examples, and all topics are well connected to each other. It's a pleasure to read for not native English people, so native English people can read it pretty fast.This book read like a novel.. It does contain enough diagrams, tables, etc. but not too much (crowded), and everything is well explained.You can read this book as a compiler introduction book. But I can only recommend this to B.Sc/M.Sc Computer Science students (like me). You don't need to have a M.Sc in Mathematics to understand this text, (all the math, eg. liveness graphs are well explained), but you will understand everything better if you have some background in algorithms (design), pseudocode, etc. like you gained during your B.Sc program. People without formal computer science education I would recommend to read a practical book first (see at the end of this review), because you may find else this text too theoretical.This book focus on code optimizations. According to the authors (and me) compiler front ends (scanning/parsing/etc) are commodities today, and the backend (codegeneration) is where the difference is made nowadays. So if you're looking for a introduction text into compiler optimization this book is for you!

What it is: A great introduction to basic concepts in contemporary compilers.What it's not: A handbook for someone thrown in at the deep end of commercial compiler development.I can imagine a very good one-term course in compiler construction built around this text. After a brief introduction, it gets immediately into the classic topics of lexical scanning, parsing, and syntax

analysis. These three chapters help any beginner understand the multiple levels of processing, from the character level, up through reorganizing grammars for practical parsing and table-driven techniques, to the lower levels of sematic analysis. This includes a very brief discussion of type systems and type inference - less than 20 pages, on a topic that whole books devote themselves to. These 200 pages typify what you'll see in the rest of the book: a lot of attention paid to lexical analysis, a problem largely eliminated by automated tools (lex and yacc being the best known), and thin mention of the harder problems that differ significantly across languages and applications of languages.Chapter 5 addresses the critical issue of intermediate representation, the data structures that represent the program during analysis, optimization, and code generation. Chapter 6 is titled "The Procedure Abstraction." It deals with much more than its name suggests, including procedure activation records (generalizations of stack frames), parameter passing, stack management, symbol visibility and scoping, and scraps of symbol table organization - important stuff, but hard to understand as "procedure abstaction." The next chapter deals with "Code Shape," a grab-bag including value representations, arrays and strings, control constructs, and procedures (again).

(Environmental Engineering)