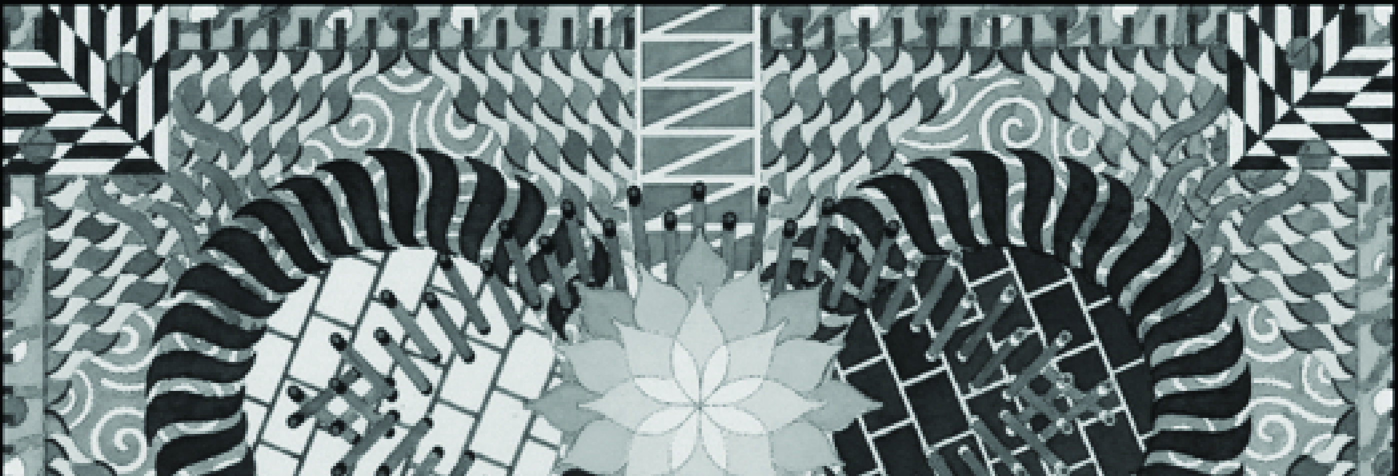
ENGINEERING A COMPILER

Keith D. Cooper & Linda Torczon



Praise for Engineering a Compiler

"Keith Cooper and Linda Torczon are leading compilers researchers who have also built several state-of-the-art compilers. This book adeptly spans both worlds, by explaining both time-tested techniques and new algorithms and by providing practical advice on engineering and constructing a compiler. *Engineering a Compiler* is a rich survey and exposition of the important techniques necessary to build a modern compiler."

—Jim Larus, Microsoft Research

"A wonderful introduction to the theory, practice, and lore of modern compilers. Cooper and Torczon convey the simple joys of this subject that follow from the elegant interplay between compilation and the rest of computer science. If you're looking for an end-to-end tour of compiler construction annotated with a broad range of practical experiences, this is the book."

—Michael D. Smith, Harvard University

"I am delighted to see this comprehensive new book on modern compiler design. The authors have covered the classical material, as well as the important techniques developed in the last 15 years, including compilation of object-oriented languages, static single assignment, region-based register allocation, and code scheduling. Their approach nicely balances the formal structure that modern compilers build on with the pragmatic insights that are necessary for good engineering of a compiler."

—John Hennessy, Stanford University

"Cooper and Torczon have done a superb job of integrating the principles of compiler construction with the pragmatic aspects of compiler implementation. This, along with the excellent coverage of recent advances in the field, make their book ideal for teaching a modern undergraduate course on compilers."

—Ken Kennedy, Rice University

Engineering a Compiler

About the Authors

Dr. Cooper, Professor, Dept. of Computer Science at Rice University, is the leader of the Massively Scalar Compiler Project at Rice, which investigates issues relating to optimization and code generation for modern machines. He is also a member of the Center for High Performance Software Research, the Computer and Information Technology Institute, and the Center for Multimedia Communication—all at Rice. He teaches courses in compiler construction at the undergraduate and graduate level.

Dr. Torczon, Research Scientist, Dept. of Computer Science at Rice University, is a principal investigator on the Massively Scalar Compiler Project at Rice and on the Grid Application Development Software Project sponsored by the Next Generation Software program of the National Science Foundation. She also serves as the executive director of the Center for High Performance Software Research and of the Los Alamos Computer Science Institute.

Engineering a Compiler

Keith D. Cooper and Linda Torczon

Rice University

MORGAN KAUFMANN PUBLISHERS

AN IMPRINT OF ELSEVIER SCIENCE

SAN FRANCISCO SAN DIEGO NEW YORK BOSTON
LONDON SYDNEY TOKYO

Senior Editor: Denise E. M. Penrose
Publishing Services Manager: Simon Crump
Project Management: Overton Editorial and Production
Editorial Coordinator: Emilia Thiuri
Cover Design: Ross Carron Design
Cover Image: John Outram
Text Design: Rebecca Evans and Associates
Composition: Integra Software Services Pvt Ltd
Technical Illustration: Dartmouth Publishing, Inc.
Copyeditor: John Hammett
Proofreader: Carol Leyba and Associates
Indexer: Steve Rath
Printer: The Maple-Vail Book Manufacturing Group

Cover Image: "The Landing of the Ark," a vaulted ceiling-design whose iconography was narrated, designed, and drawn by John Outram of John Outram Associates, Architects and City Planners, London, England. To read more visit
www.johnoutram.com/rice.html

Morgan Kaufmann Publishers
An Imprint of Elsevier Science
500 Sansome Street, Suite 400
San Francisco, CA 94111
www.mkp.com

2008 2007 2006 2005 2004 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data Application submitted

ISBN: 1–55860–698-X (hardback)–ISBN 1–55860–699–8 (paperback)

This book is printed on acid-free paper.

We dedicate this volume to

- our parents, who instilled in us the thirst for knowledge and supported us as we developed the skills to follow our quest for knowledge;
- our children, who have shown us again how wonderful the process of learning and growing can be; and
- our spouses, without whom this book would never have been written.

About the Cover

The cover of this book features a portion of the drawing, “The Landing of the Ark,” which decorates the ceiling of Duncan Hall at Rice University (see the picture below). Both Duncan Hall and its ceiling were designed by British architect John Outram. Duncan Hall is an outward expression of architectural, decorative, and philosophical themes developed over Outram’s career as an architect. The decorated ceiling of the ceremonial hall plays a central role in the building’s decorative scheme. Outram inscribed the ceiling with a set of significant ideas—a creation myth. By expressing those ideas in an allegorical drawing of vast size and intense color, Outram created a signpost that tells visitors who wander into the hall that, indeed, this building is not like other buildings.

By using the same signpost on the cover of *Engineering a Compiler*, the authors intend to signal that this work contains significant ideas that are at the core of their discipline. Like Outram’s building, this volume is the culmination of intellectual themes developed over the authors’ professional careers. Like Outram’s decorative scheme, this book is a device for communicating ideas. Like Outram’s ceiling, it presents significant ideas in new ways.

By connecting the design and construction of compilers with the design and construction of buildings, we intend to convey the many similarities in these two distinct activities. Our many long discussions with Outram introduced us to the Vitruvian ideals for architecture: commodity, firmness, and delight. These ideals apply to many kinds of construction. Their analogs for compiler construction are consistent themes of this text: function, structure, and elegance. Function matters; a compiler that generates incorrect code is useless. Structure matters; engineering detail determines a compiler’s efficiency and robustness. Elegance matters; a well-designed compiler, in which the algorithms and data structures flow smoothly from one pass to another, can be a thing of beauty.

We are delighted to have John Outram’s work grace the cover of this book.

Duncan Hall’s ceiling is an interesting technological artifact. Outram drew the original design on one sheet of paper. It was photographed and scanned at 1200 dpi yielding roughly 750 MB of data. The image was enlarged to form 234 distinct 2 x 8 foot panels, creating a 52 x 72 foot image. The panels were printed onto oversize sheets of perforated vinyl using a 12 dpi acrylic-ink printer. These sheets were precision mounted onto 2 x 8 foot acoustic tiles and hung on the vault’s aluminum frame.

CONTENTS

Preface

Over the last twenty years, the practice of compiler construction has changed dramatically. Front ends have become commodity components; they can be purchased from a reliable vendor or adapted from one of the many public-domain systems. At the same time, processors have become more performance sensitive; the actual performance of compiled code depends heavily on the compiler's ability to optimize for specific processor and system features. These changes affect the way that we build compilers; they should also affect the way that we teach compiler construction.

Compiler development today focuses on optimization and on code generation. A new hire in a compiler group is far more likely to port a code generator to a new processor or modify an optimization pass than to work on a scanner or parser. Preparing students to enter this environment is a real challenge. Successful compiler writers must be familiar with current best-practice techniques in optimization and code generation. They must also have the background and intuition to understand new techniques as they appear during the coming years. Our goal in writing *Engineering a Compiler* (EAC) has been to create a text and a course that exposes students to the critical issues in modern compilers and provides them with the background to tackle those problems.

Motivation for Studying Compiler Construction

Compiler construction brings together techniques from disparate parts of computer science. At its simplest, a compiler is just a large computer program. A compiler takes a source-language program and translates it for execution on some target architecture. As part of this translation, the compiler must perform syntax analysis to determine if the input program is valid. To map that input program onto the finite resources of a target computer, the compiler must manipulate several distinct name spaces, allocate several different kinds of resources, and orchestrate the behavior of multiple run-time data structures. For the output program to have reasonable performance, it must manage hardware latencies in functional units, predict the flow of execution and the demand for memory, and reason about the independence and dependence of different machine-level operations in the program.

Open up a modern optimizing compiler and you will find greedy heuristic searches that explore large solution spaces, deterministic finite automata that recognize words in the input, fixed-point algorithms that help reason about program behavior, simple theorem provers and algebraic simplifiers that try to predict the values of expressions, pattern-matchers that map abstract computations to machine-level operations, solvers for diophantine equations and Pressburger arithmetic used to analyze array subscripts, and such classic algorithms and data-structures as hash tables, graph algorithms, and sparse set implementations.

Balance

Our primary goal in writing *Engineering a Compiler* (EAC) has been to create a text for use in an introductory course on the design and implementation of compilers. EAC lays out many of the problems that face compiler writers and explores some of the techniques used by compiler writers to solve them. EAC presents a pragmatic selection of practical techniques that you might use to build a modern compiler.

In selecting material for EAC, we have deliberately rebalanced the curriculum for a first course in compiler construction to cover the material that a student will need in the job market. This shift reduces the coverage of front-end issues in favor of increased coverage of optimization and code generation. In these latter areas, EAC focuses on best-practice techniques such as static single-assignment form, list scheduling, and graph-coloring register allocation. These topics prepare students for the algorithms that they will encounter in a modern commercial or research compiler.

The book also includes material for the advanced student or the practicing professional. Most chapters include an *Advanced Topics* section that discusses issues and techniques that are beyond a typical undergraduate course. In addition, Chapters 9 and 10 introduce data-flow analysis and scalar optimization in greater depth than a typical undergraduate course will cover. Including this material in EAC makes it available to the more advanced or curious student; professionals may also find these chapters useful as they try to implement some of the techniques.

Approach

Compiler construction is an exercise in engineering design. The compiler writer must choose a path through a design space that is filled with diverse alternatives, each with distinct costs, advantages, and complexity. Each decision has an impact on the resulting compiler. The quality of the end product depends on informed decisions at each step along the way.

Thus, there is no single right answer for many of the design decisions in a compiler. Even within "well understood" and "solved" problems, nuances in design and implementation have an impact on both the behavior of the compiler and the quality of the code that it produces. Many considerations play into each decision. As an example, the choice of an intermediate representation for the compiler has a profound impact on the rest of the compiler, from time and space requirements through the ease with which different algorithms can be applied. The decision, however, is often given short shrift. Chapter 5 examines the space of intermediate representations and some of the issues that should be considered in selecting one. We raise the issue again at several points in the book—both directly in the text and indirectly in the exercises.

EAC tries to explore the design space and convey both the depth of the problems and the breadth of the possible solutions. It presents some of the ways that problems have been solved, along with the constraints that made those solutions attractive. A student needs to understand both the parameters of the problems and their solutions, as well as the impact of those decisions on other facets of the compiler's design. Only then can the compiler writer make informed and intelligent choices.

Philosophy

This text exposes our philosophy for how compilers should be built, developed in more than twenty years each of research, teaching, and practice. For example, intermediate representations should expose those details that matter in the final code; this belief leads to a bias toward low-level representations. Values should reside in registers until the allocator discovers that it cannot keep them there; this practice produces examples that use virtual registers and store values to memory only when it cannot be avoided. It also increases the importance of effective algorithms in the compiler's back end. Every compiler should include optimization; it simplifies the rest of the compiler.

EAC departs from some of the accepted conventions for compiler construction textbooks. For example, we use several different programming languages in the examples. It makes little sense to describe call-by-name parameter passing in c, so we use Algol-60. It makes little sense to describe tail-recursion in FORTRAN, so we use Scheme. This multilingual approach is realistic; over the course of the reader's career, the "language of the future" will change several times.[1] Algorithms in EAC are presented at a reasonably high level of abstraction. We assume that the reader can fill in the details and that those details might be tailored to the specific environment in which the code will run.

Organizing the Text

In writing EAC, our overriding goal has been to create a textbook that prepares a student to work on real compilers. We have taught the material in this text for a decade or more, experimenting with the selection, depth, and order. The course materials available on the website show how we adapt and teach the contents of EAC in the undergraduate course at Rice University.

The desire to teach modern code generation techniques complicates the problem of ordering the material. Modern code generators rely heavily on ideas from optimization, such as data-flow analysis and static single-assignment form. This dependence suggests teaching optimization before covering back-end algorithms. Covering optimization before any discussion of code generation means that a student may not see the code generated for a case statement, a loop, or an array reference before trying to improve that code.

Since no linear ordering of the material is perfect, EAC presents the material, to the extent possible, in the order that the algorithms execute at compile time. Thus, optimization follows the front end and precedes the back end, even though the discussion of code shape is part of the back end material. The chapter opening graphic serves as a reminder of this order. In practice, an undergraduate course will take some of the material out of order.

1. Over the past thirty years, Algol-68, APL, PL/I, Smalltalk, C, Modula-3, C++, Java, and even ADA have been hailed as the language of the future.

Content

After the introduction (Chapter 1), the text divides into four sections.

Front End. Successive chapters in the first section present scanning, parsing, and context-sensitive analysis. Chapter 2 introduces recognizers, finite automata, regular expressions, and the algorithms for automating the construction of a scanner from a regular expression. Chapter 3 describes parsing, with context-free grammars, top-down recursive-descent parsers, and bottom-up, table-driven, LR(1) parsers. Chapter 4 introduces type systems as an example of a practical problem that is too complex to express in a context-free grammar. It then shows both formal and ad hoc techniques for solving such context-sensitive problems.

These chapters show a progression. In scanning, automation has replaced hand coding. In parsing, automation has dramatically reduced the programmer's effort. In context-sensitive analysis, automation has not replaced ad hoc, hand-coded methods. However, those ad hoc techniques mimic some of the intuitions behind one of the formal techniques, the use of attribute grammars.

Infrastructure. The second section brings together material that is often scattered throughout the course. It provides background knowledge needed to generate intermediate code in the front end, to optimize that code, and to transform it into code for a target machine.

Chapter 5 describes a variety of intermediate representations that compilers use, including trees, graphs, linear codes, and symbol tables. Chapter 6 introduces the run-time abstractions that a compiler must implement with the code that it generates, including procedures, name spaces, linkage conventions, and memory management. Chapter 7 provides a prelude to code generation, focusing on what kind of code the compiler should generate for various language constructs rather than on the algorithms to generate that code.

Optimization. The third section covers issues that arise in building an optimizer, a compiler's middle section. Chapter 8 provides an overview of the problems and techniques of optimization by working one problem at several different scopes. Chapter 9 introduces iterative data-flow analysis and presents the construction of static single-assignment form. Chapter 10 shows an effects-based taxonomy for scalar optimization and then populates the taxonomy with selected examples.

This division reflects the fact that a full treatment of analysis and optimization may not fit into a single-semester course, while making the material available in the book for the more advanced or curious student. In teaching this material, we cover Chapter 8 and then move on to code generation. During code generation, we refer back to specific sections in Chapters 9 and 10 as the need or interest arises. We also use this section of the book, augmented with a selection of papers, to teach a second course on scalar optimization.

Code Generation. The final section looks at the three primary problems in code generation. Chapter 11 covers instruction selection; it begins with tree-pattern matching and then delves into peephole-style matchers. Chapter 12 examines instruction scheduling; it focuses on list scheduling and its variants. Chapter 13 presents register allocation; it gives an in-depth treatment of algorithms for both local and global allocation. The algorithms that EAC presents are techniques that a student might find used inside a modern compiler.

For some students, these chapters are the first time that they must approximate the solution to an NP-complete problem rather than prove it equivalent to three-satisfiability. The chapters emphasize best-practice approximation algorithms. The exercises give students the opportunity to work tractable examples.

Crosscutting Ideas. Compiler construction is a complex, multifaceted discipline. Due to the sequential flow of information in a compiler, solutions chosen for one problem determine the input that later phases see and the opportunities that those phases have to improve the code. Small changes made in the front end can hide opportunities for optimization; the results of optimization have a direct impact on the code generator (changing, for example, the demand for registers). The complex, interrelated nature of design decisions in a compiler are one reason that this material is often used in a capstone course for undergraduates.

These complex relationships also arise in a compiler construction course. Solution techniques appear again and again in the course. Fixed-point algorithms play a critical role in the construction of scanners and parsers. They are a primary tool for the analyses that support optimization and code generation. Finite automata arise in scanning. They play a key role in the LR(1) table construction and, again, in pattern matchers for instruction selection. By identifying and emphasizing these common techniques, EAC makes them familiar. Thus, when a student encounters the iterative data-flow algorithm in

Chapter 8, it is just another fixed-point algorithm and, thus, familiar. Similarly, the discussion on the scope of optimization in Chapter 8 is reinforced by the transition from local algorithms to regional or global algorithms in Chapters 12 and 13.

ORGANIZING THE COURSE

A class in compiler construction offers both student and teacher the opportunity to explore all these issues in the context of a concrete application—one whose basic functions are well understood by any student with the background for a compiler construction course. In some curricula, the course serves as a capstone course for seniors, tying together concepts from many other courses in a practice-oriented project course. Students in such a class might write a complete compiler for a simple language or add support for a new language feature to an existing compiler such as GCC or the ORC compiler for the IA-64. This class might present the material in a linear order that closely follows the text's organization.

If other courses in the curriculum give students the experience of large projects, the teacher can focus the compiler construction course more narrowly on algorithms and their implementation. In such a class, the labs can focus on abstracted instances of truly hard problems, such as register allocation and scheduling. This class might skip around in the text, adjusting the order of presentation to meet the needs of the labs. For example, any student who has done assembly-language programming can write a register allocator for straightline code. We have often used a simple register allocator as the first lab.

In either scenario, the course should draw material from other classes. Obvious connections exist to computer organization and assembly-language programming, operating systems, computer architecture, algorithms, and formal languages. Although the connections from compiler construction to other courses may be less obvious, they are no less important. Character copying, as discussed in Chapter 7, plays a critical role in the performance of applications that include network protocols, file servers, and web servers. The techniques developed in Chapter 2 for scanning have applications that range from text editing through URL-filtering. The bottom-up local register allocator in Chapter 13 is recognizable as a cousin of the optimal offline page replacement algorithm.

Supporting Materials

Morgan Kaufmann's website for the book contains a variety of resources that should help you adapt the material presented in EAC to your course. The web site includes

1. a complete set of lectures for the course as taught at Rice University;
2. example lab assignments from a capstone-project version of the course;
3. example lab assignments from the course as taught at Rice;
4. an instructor's manual that contains solutions for the exercises;
5. a glossary of abbreviations, acronyms, and terms defined in EAC;
6. single-page copies of the line art from the book; and
7. the syllabus and lectures for a course on scalar optimization taught from the optimization section of EAC and a selection of recent papers.

The Art and Science of Compiler Construction

The lore of compiler construction includes both amazing success stories about the application of theory to practice and humbling stories about the limits of what we can do. On the success side, modern scanners are built by applying the theory of regular languages to automatic construction of recognizers. LR parsers use the same techniques to perform the handle-recognition that drives a shift-reduce parser. Data-flow analysis (and its cousins) apply lattice theory to the analysis of programs in ways that are both useful and clever. The approximation algorithms used in code generation produce good solutions to many instances of truly hard problems.

On the other side, compiler construction exposes some complex problems that defy good solutions. The back end of a compiler for a modern superscalar machine must approximate the solution to two or more interacting NP-complete problems (instruction scheduling, register allocation, and, perhaps, instruction and data placement). These NP-complete problems, however, look easy next to problems such as algebraic reassociation of expressions (see, for example, Figure 7.1). This problem admits a huge number of solutions; to make matters worse, the desired solution depends on the other transformations that the compiler applies. While the compiler attempts to solve these problems (or approximate their solutions), it must run in a reasonable amount of time and consume a modest amount of space. Thus, a good compiler for a modern superscalar machine must artfully blend theory, practical knowledge, engineering, and experience.

In this book, we have tried to convey both the art and the science of compiler construction. EAC includes a sufficiently broad selection of material to show the reader that real tradeoffs exist and that the impact of those choices can be both subtle and far-reaching. EAC omits techniques that have been rendered less important by changes in the marketplace, in the technology of languages and compilers, or in the availability of tools. Instead, EAC provides a deeper treatment of optimization and code generation.

ACKNOWLEDGMENTS

Many people have provided us with useful feedback on the form, content, and exposition of EAC. Among these are L. Almagor, Saman Amarasinghe, Thomas Ball, Preston Briggs, Corky Cartwright, Carolyn Cooper, Christine Cooper, Anshuman Das Gupta, Jason Eckhardt, Stephan Ellner, Mike Fagan, Matthias Felleisen, Alex Grosul, John Greiner, Dan Grossman, Timothy Harvey, James Larus, Ken Kennedy, Shriram Krishnamurthy, Ursula Kuterbach, Robert Morgan, Guilherme Ottoni, Vishal Patel, Norm Ramsey, Steve Reeves, Martin Rinard, L. Taylor Simpson, Reid Tatge, Dan Wallach, Todd Waterman, and Christian Westbrook.

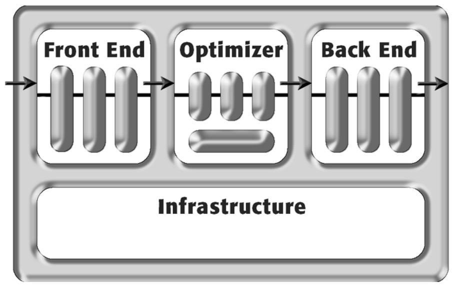
Steve Carr served as exercise editor; he coordinated a team that produced the exercises. His team included Chen Ding, Rodolfo Jardim de Azevedo, Zhiyuan Li, Guilherme Ottoni, and Sandra Rigo. Aaron Smith, Ben Hardekopf, and Paul A. Navratil checked the exercise solutions. The manuscript went through many rounds of review and revision; Saman Amarasinghe, Guido Araujo, Preston Briggs, Steve Carr, James Larus, Gloria Melara, Kathryn McKinley, Robert Morgan, Thomas Murtagh, Gordon Novak, Santosh Pande, Allan Porterfield, Martin Rinard, Mark Roberts, Michael Smith, and Hongwei Xi all served as reviewers. Wilson Hsieh, Jurek Jaromczyk, Tevfik Bultan, Chau-Wen Tseng, Mahmut Kandemir, and Zhiyuan Li all tested the book in their classrooms, as did several members of the exercise team and the reviewing team. Their work improved this volume, changing its style and its contents and improving its accuracy.

Michael Scott and Steve Muchnick each reviewed the complete manuscript several times. Their attention to detail, their patience, and their suggestions substantially improved this book.

The editorial and production team that Morgan Kaufmann put together for this book has been wonderful. The team, led by Denise Penrose and Yonie Overton, includes John Hammet, Carol Leyba, Rebecca Evans, Steve Rath, Emilia Thiuri, and Lauren Wheelock. The illustrations were redrawn by a team at Dartmouth Publishing; the composition work was done by a team at Integra. As authors, we could not ask for a more flexible, skilled, or patient group of people. Their suggestions and insights improved the book; their professionalism made the process of producing it substantially easier.

Finally, many people have provided us with intellectual and emotional support over the last five years. First and foremost, our families and our colleagues at Rice have encouraged us at every step of the way. In addition, Regina Brooks and Janice Bordeaux encouraged us to start this effort. Denise Penrose and Yonie Overton made it possible for us to finish it. We are deeply indebted to Kathryn O'Brien for undertaking the many administrative tasks that arise in a project of this size and for keeping a sense of humor through it all. Finally, Ken Kennedy and Scott Warren have both shaped the way that we think about programming, about language issues, and about compilation.

CHAPTER 1

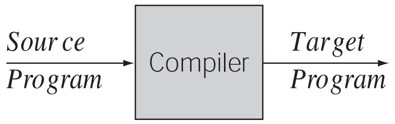


Overview of Compilation

1.1 INTRODUCTION

The role of computers in daily life is growing each year. Modern microprocessors are found in cars, microwave ovens, dishwashers, mobile telephones, GPS navigation systems, video games, and personal computers. These computers perform their jobs by executing programs—sequences of operations written in a "programming language." The programming language is a formal language with mathematical properties and well-defined meanings, as opposed to a natural language with evolved properties and ambiguities. Programming languages are designed for expressiveness, conciseness, and clarity. Programming languages are designed to specify computations—to record a sequence of actions that perform a particular computational task or produce a specific computational result.

Before a program can execute, it must be translated into a set of operations that are defined on the target computer. This translation is done by a specialized program called a *compiler*. The compiler takes as input the specification for an executable program and produces as output the specification for another, equivalent executable program. Of course, if it finds errors in the input program, the compiler should produce an appropriate set of error messages. Viewed as a black box, a compiler might look like this:

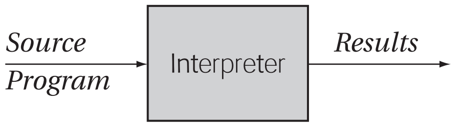


Typically, the “source” language that the compiler accepts is a programming language, such as FORTRAN, C, C++, Ada, Java, or ML. The “target” language is usually the instruction set of some computer system.

Some compilers produce a target program written in a full-fledged programming language rather than the assembly language of some computer. The programs that these compilers produce require further translation before they can execute directly on a computer. Many research compilers produce C programs as their output. Because compilers for C are available on most computers, this makes the target program executable on all those systems, at the cost of an extra compilation for the final target. Compilers that target programming languages rather than the instruction set of a computer are often called *source-to-source translators*.

Many other systems qualify as compilers. For example, a typesetting program that produces PostScript can be considered a compiler. It takes as input a specification for how the document should look on the printed page and it produces as output a PostScript file. PostScript is simply a language for describing images. Since the typesetting program takes an executable specification and produces another executable specification, it is a compiler.

The code that turns PostScript into pixels is typically an *interpreter,* not a compiler. An interpreter takes as input an executable specification and produces as output the result of executing the specification.



Interpreters can be used to implement programming languages as well. For some languages, such as Perl, Scheme, and APL, interpreters are more common than compilers.

Interpreters and compilers have much in common. They perform many of the same tasks. Both examine the input program and determine whether or not it is a valid program. Both build an internal model of the structure and meaning of the program. Both determine where to store values during execution. However, interpreting the code to produce a result is quite different from emitting a translated program that can be executed to produce the result. This book focuses on the problems that arise in building compilers. However, an implementor of interpreters may find much of the material relevant.

1.2 Why Study Compiler Construction?

A compiler is a large, complex program. Compilers often include hundreds of thousands, if not millions, of lines of code. Their many parts have complex interactions. Design decisions made for one part of the compiler have important ramifications for other parts. Thus, the design and implementation of a compiler is a substantial exercise in software engineering.

A good compiler contains a microcosm of computer science. It makes practical application of greedy algorithms (register allocation), heuristic search techniques (list scheduling), graph algorithms (dead-code elimination), dynamic programming (instruction selection), finite automata and push-down automata (scanning and parsing), and fixed-point algorithms (data-flow analysis). It deals with problems such as dynamic allocation, synchronization, naming, locality, memory hierarchy management, and pipeline scheduling. Few software systems make as many complex and diverse components work together to achieve a single purpose. Working inside a compiler provides practical experience in software engineering that is hard to obtain with smaller, less intricate systems.

Compilers play a fundamental role in the central activity of computer science: preparing problems for solution by computer. Most software is compiled; the correctness of that process and the efficiency of the resulting code have a direct impact on our ability to build large systems. Most students are not satisfied with reading about these ideas; many of the ideas must be implemented to be appreciated. Thus, the study of compiler construction is an important component of a computer science education.

The lore of compiler construction includes many success stories. Formal language theory has led to tools that automate the production of scanners and parsers. These same tools and techniques find application in text searching, website filtering, word processing, and command-language interpreters. Type checking and static analysis apply results from lattice theory, number theory, and other branches of mathematics to understand and improve programs. Code generators uses algorithms for tree-pattern matching, parsing, dynamic programming, and text matching to automate the selection of instructions.

At the same time, the history of compiler construction includes its share of humbling experiences. Compilation includes problems that are truly hard. Attempts to design a high-level, universal, intermediate representation have foundered on complexity. Several parts of the process have resisted automation—at least, automatic techniques have not yet replaced hand-coded solutions. In many cases, we have had to resort to ad hoc methods. The dominant method for instruction scheduling is a greedy algorithm with several layers of tie-breaking heuristics. While it is obvious that the compiler can use commutativity and associativity to improve the code, most compilers that try to do so simply rearrange the expression into some canonical order.

Building a successful compiler requires a blend of algorithms, engineering insights, and careful planning. Good compilers approximate the solutions to hard problems. They emphasize efficiency—in their own implementations and in the code they generate. They have internal data structures and knowledge representations that expose the right level of detail—enough to allow strong optimization, but not enough to force the compiler to wallow in detail.

1.3 The Fundamental Principles of Compilation

Compilers are engineered objects—large software systems built with distinct goals. Building a compiler requires myriad design decisions, each of which has an impact on the resulting compiler. While many issues in compiler design are amenable to several different solutions, there are two principles that should not be compromised. The first principle that a compiler must observe is inviolable.

The compiler must preserve the meaning of the program being compiled.

Correctness is a fundamental issue in programming. The compiler must preserve correctness by faithfully implementing the "meaning" of its input program. This principle lies at the heart of the social contract between the compiler writer and compiler user. If the compiler can take liberties with meaning, then why not simply generate a `nop` or a `return`? If an incorrect translation is acceptable, why expend the effort to get it right?

The second principle that a compiler must observe is practical.

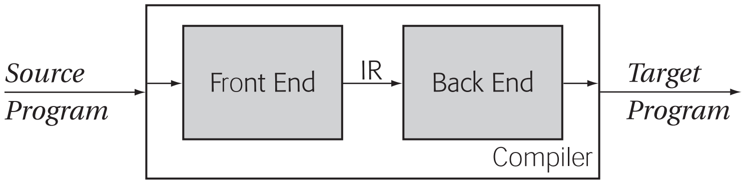
The compiler must improve the input program in some discernible way.

A traditional compiler improves upon the input program by making it directly executable on some target machine. Other "compilers" improve their input in different ways. For example, `tpic` is a program that takes the specification for a drawing written in the graphics language `pic` and converts it into LaTeX; the "improvement" lies in LaTeX's greater availability and generality. Some compilers produce output programs in the same language as their input; we call these source-to-source translators. In general, these systems rewrite a program in a way that will lead to an improvement when the program is finally translated into code for some target machine. If the compiler does not improve the code in some way, why should anyone invoke it?

1.4 COMPILER STRUCTURE

A compiler is a large and complex software system. The compiler community has been building compilers since 1955; over those years, we have learned many lessons about how to structure a compiler. Earlier, we depicted a compiler as a single box that translates a source program into a target program. Reality, of course, is more complex than that simple picture.

As this single-box model suggests, a compiler must both understand the source program presented for compilation and map its functionality to the target machine. The distinct nature of these two tasks suggests a division of labor and leads to a design that decomposes compilation into two major pieces: a *front end* and a *back end*.



The front end focuses on understanding the source-language program. The back end focuses on mapping programs to the target machine. This separation of concerns has several important implications for the design and implementation of compilers.

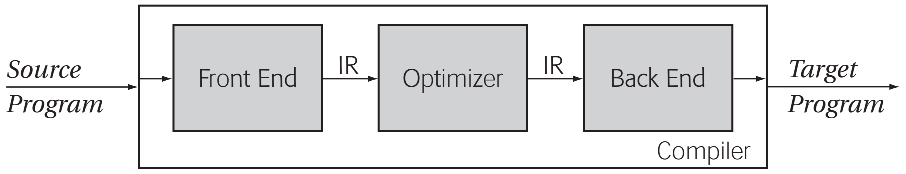
The front end must encode its knowledge of the source program in some structure for later use by the back end. This *intermediate representation* (IR) becomes the compiler's definitive representation for the code it is translating. At each point in compilation, the compiler will have a definitive representation. It may, in fact, use several different IRs as compilation progresses, but at each point, one representation will be the definitive IR. We think of the definitive IR as the version of the program passed between independent phases of the compiler, like the IR passed from the front end to the back end in the preceding drawing.

In a two-phase compiler, the front end must ensure that the source program is well formed, and it must map that code into the IR. The back end must map the IR program into the instruction set and the finite resources of the target machine. Since the back end only processes IR created by the front end, it can assume that the IR contains no syntactic or semantic errors.

The compiler can make multiple passes over the IR form of the code before emitting the target program. This should lead to better code; the compiler can, in effect, study the code in its first phase and record relevant details. Then, in the second phase, it can use these recorded facts to improve the quality of translation. (This idea is not new. The original FORTRAN compiler made several passes over the code.) This strategy requires that knowledge derived in the first pass be recorded in the IR where the second pass can find and use it.

Finally, the two-phase structure may simplify the process of retargeting the compiler. We can easily envision constructing multiple back ends for a single front end to produce compilers that accept the same language but target different machines. Similarly, we can envision front ends for different languages producing the same IR and using a common back end. Both scenarios assume that one IR can serve for several combinations of source and target; in practice, both language-specific and machine-specific details usually find their way into the IR.

Introducing an IR makes it possible to add more phases to compilation. The compiler writer can insert a third phase between the front end and the back end. This middle section, or *optimizer*, takes an IR program as its input and produces an equivalent IR program as its output. By using the IR as an interface, the compiler writer can insert this third phase with minimal disruption to the front end and the back end. This leads to the following compiler structure, termed a *three-phase compiler*.



The optimizer is an IR-to-IR transformer that tries to improve the IR program in some way. (Notice that these transformers are, themselves, compilers according to our definition in Section 1.1.) The optimizer can make one or more passes over the IR, analyze the IR, and rewrite the IR. The optimizer may rewrite the IR in a way that is likely to produce a faster target program from the back end or a smaller target program from the back end. It may have other objectives, such as a program that produces fewer page faults or uses less power.

Conceptually, this three-phase structure represents the classic optimizing compiler. In practice, the phases are divided internally into a series of passes. The front end consists of two or three passes that handle the details of recognizing valid source-language programs and producing the initial IR form of the program. The middle section contains several passes that perform different optimizations. The number and purpose of these passes vary from compiler to compiler. The back end consists of a series of passes, each of which takes the IR program one step closer to the target machine's instruction set. The three phases and their individual passes share a common infrastructure. This structure is shown in Figure 1.1.

In practice, the conceptual division of a compiler into three phases, a front end, a middle section, and a back end, is useful. The problems addressed by these phases are different. The front end is concerned with understanding the source program and recording the results of its analysis into IR form. The middle section focuses on improving the IR form. The back end must map the transformed IR program onto the bounded resources of the target machine in a way that leads to efficient use of those resources.

Of these three phases, the middle section has the murkiest description. The term *optimization* implies that the compiler discovers an optimal solution to some problem. The issues and problems that arise in optimization are so complex and so interrelated that they cannot, in practice, be solved

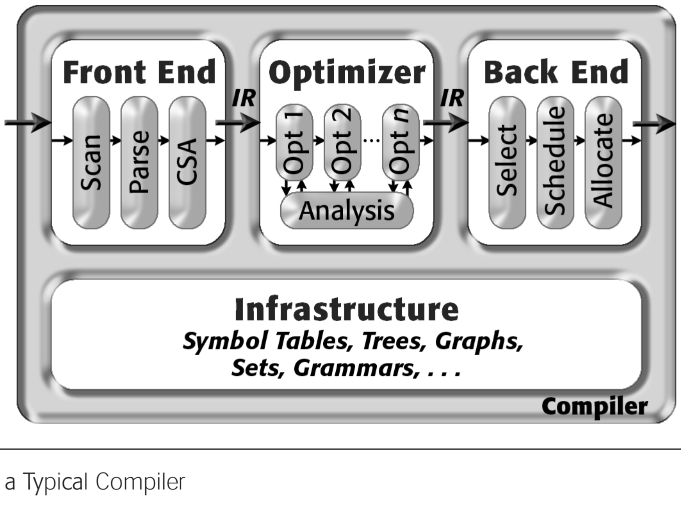


FIGURE 1.1 Structure of a Typical Compiler

optimally. Furthermore, the actual behavior of the compiled code depends on interactions among all of the techniques applied in the optimizer and the back end. Thus, even if a single technique can be proved optimal, its interactions with other techniques may produce less than optimal results. As a result, a good optimizing compiler can improve the quality of the code, relative to an unoptimized version. It will almost always fail to produce optimal code.

The middle section can be a single monolithic pass that applies one or more optimizations to improve the code, or it can be structured as a series of smaller passes with each pass reading and writing IR. The monolithic structure may be more efficient. The multipass structure may lend itself to a less complex implementation and a simpler approach to debugging the compiler. It also creates the flexibility to employ different sets of optimization in different situations. The choice between these two approaches depends on the constraints under which the compiler is built and operates.

1.5 High-Level View of Translation

To gain a better understanding of the tasks that arise in compilation, consider what must be done to generate executable code for the following expression:

$$w \leftarrow w \times 2 \times x \times y \times z$$

where w, x, y, and z are variables, ← indicates an assignment, and × is the operator for multiplication. To learn what facts the compiler must discover and what questions it must answer, we will trace the path that a compiler takes to turn such a simple program into executable code.

1.5.1 Understanding the Input

The first step in compiling $w \leftarrow w \times 2 \times x \times y \times z$ is to determine whether or not these characters form a legal sentence in the programming language. This job falls to the compiler's front end. It involves both form, or *syntax*, and meaning, or *semantics*. If the program is well formed in both these respects, the compiler can continue with translation, optimization, and code generation. If it is not well formed, the compiler should report back to the user with a clear error message that isolates the problems with the sentence, to the extent possible.

Notation

Compiler books are, in essence, about notation. After all, a compiler translates a program written in one notation into an equivalent program written in another notation. A number of notational issues will arise in your reading of this book. In some cases, these issues will directly affect your understanding of the material.

Expressing Algorithms We have tried to keep the algorithms concise. Algorithms are written at a relatively high level, assuming that the reader can supply implementation details. They are written in a *slanted, sans-serif font*. Indentation is both deliberate and significant; this matters most in an *if-then-else* construct. Indented code after a *then* or an *else* forms a block. In the following code fragment

```
if Action [s,word] = "shift $s_i$" then
    push word
    push $s_i$
    word ← NextWord()
else if ···
```

all the statements between the *then* and the *else* are part of the *then* clause of the *if-then-else* construct. When a clause in an *if-then-else* construct contains just one statement, we write the keyword *then* or *else* on the same line as the statement.

Writing Code In some examples, we show actual program text written in some language chosen to demonstrate a particular point. Actual program text is written in a `typewriter` font.

Arithmetic Operators Finally, we have forsaken the traditional use of $*$ for × and of / for ÷ , except in actual program text. The meaning should be clear to the reader.

Checking Syntax

To check the syntax of the input program, the compiler must compare the program's structure against a definition for the language. This requires an appropriate formal definition, an efficient mechanism for testing whether or not the input meets that definition, and a plan for how to proceed on an illegal input.

Mathematically, the source language is a set, usually infinite, of strings defined by some finite set of rules, called a *grammar*. In a compiler's front end, the scanner and the parser determine whether the input program is, in fact, an element of that set of valid strings. The engineering challenge is to make this membership test efficient.

Grammars for programming languages usually refer to words by their parts of speech, or syntactic categories. Basing the grammar rules on parts of speech lets a single rule describe many sentences. For example, in English, many sentences have the form

$$\textit{Sentence} \rightarrow \textit{Subject}\ \texttt{verb}\ \textit{Object}\ \texttt{endmark}$$

where `verb` and `endmark` are parts of speech, and *Sentence*, *Subject*, and *Object* are syntactic variables. *Sentence* represents any string with the form described by this rule. The symbol "$\rightarrow$" reads "derives" and means that an instance of the right-hand side can be abstracted to the syntactic variable on the left-hand side.

To apply this rule, the user must map words to their parts of speech. For example, `verb` represents the set of all English-language verbs, and `endmark` represents all sentence-ending punctuation marks, such as a period, a question mark, or an exclamation point. For English, the reader generally recognizes several thousand words and knows the possible parts of speech that each can fulfill. For an unfamiliar word, the reader consults a dictionary. Thus, the syntax of this example is described with a set of rules, or grammar, and a system for finding words and classifying them into syntactic categories.

This specification-based approach to defining syntax is critical to compilation. We cannot build a front end that contains an infinite set of rules or an infinite set of sentences. Instead, we need a finite set of rules that generate or specify the sentences in our language. As we shall see in Chapters 2 and 3, the finite nature of a grammar does not limit the expressiveness of the language.

To understand whether the sentence "Compilers are engineered objects." is, in fact, a valid English sentence, we first establish that each word exists in English with a dictionary lookup. Next, each word is replaced by its syntactic category to create a somewhat more abstract representation of the sentence:

```
noun verb adjective noun endmark
```

Finally, we try to fit this sequence of abstracted words into the rules for an English sentence. A working knowledge of English grammar might include the following rules:

1	*Sentence*	→	*Subject* `verb` *Object* `endmark`
2	*Subject*	→	`noun`
3	*Subject*	→	*Modifier* `noun`
4	*Object*	→	`noun`
5	*Object*	→	*Modifier* `noun`
6	*Modifier*	→	`adjective`
	...		

By inspection, we can discover the following *derivation* for our example sentence:

Rule	Prototype Sentence
—	*Sentence*
1	*Subject* `verb` *Object* `endmark`
2	`noun verb` *Object* `endmark`
5	`noun verb` *Modifier* `noun endmark`
6	`noun verb adjective noun endmark`

The derivation starts with the syntactic variable *Sentence*. At each step, it rewrites one term in the prototype sentence, replacing the term with a right-hand side that can be derived from that rule. The first step uses Rule 1 to replace *Sentence*. The second uses Rule 2 to replace *Subject*. The third replaces *Object* using Rule 5, while the final step rewrites *Modifier* with `adjective` according to Rule 6. At this point, the prototype sentence generated by the derivation matches the abstract representation of our input sentence.

This derivation proves that "Compilers are engineered objects." belongs to the language described by Rules 1 through 6. The process of discovering words in a string of characters and classifying them according to their parts of speech is called *scanning*. Discovering whether a stream of classified words has a derivation in some set of grammatical rules is called *parsing*. Scanning and parsing are the first two steps in compiling a program.

Of course, the scanner and parser might discover that the input is not a valid sentence. In this case, the compiler must report the error back to the user. It should provide concise and useful feedback that lets the user isolate and correct the syntactic error.