PEARSON Prentice Hall

大学计算机教育国外著名教材系列

影印版

# Crafting a Compiler

# 编译器构造

Charles N. Fischer
Ronald K. Cytron
Richard J. LeBlanc, Jr. 著

PEARSON Education

清华大学出版社

大学计算机教育国外著名教材系列（影印版）

# Crafting a Compiler

# 编译器构造

Charles N. Fischer

Ronald K. Cytron 著

Richard J. LeBlanc, Jr.

清华大学出版社

北京

北京市版权局著作权合同登记号　图字：01-2010-1905 号



**图书在版编目(CIP)数据**

编译器构造：英文/（美）费希尔（Fischer, C. N.），赛特朗（Cytron, R. K.），勒布兰（Jr. R. J. L.）著. --影印本. --北京：清华大学出版社，2010.6
（大学计算机教育国外著名教材系列）
ISBN 978-7-302-22720-5

Ⅰ. ①编…　Ⅱ. ①费…　②赛…　③勒…　Ⅲ. ①编译码器－构造－教材－英文　Ⅳ. ①TN762

中国版本图书馆 CIP 数据核字（2010）第 088371 号

责任印制：王秀菊

出版发行：清华大学出版社　　地　　址：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　邮　　编：100084
社　　总　　机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62795954，jsjjc@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015, zhiliang@tup.tsinghua.edu.cn
印 刷 者：清华大学印刷厂
装 订 者：三河市金元印装有限公司
发 行 者：全国新华书店
开　　本：185×230　　印张：44.75
版　　次：2010 年 6 月第 1 版　　印　　次：2010 年 6 月第 1 次印刷
印　　数：1~3000
定　　价：79.00 元

产品编号：026160-01

# *Preface*

Much has changed since *Crafting a Compiler*, by Fischer and LeBlanc, was published in 1988. While instructors may remember the $5\frac{1}{4}$-inch floppy disk of software that accompanied that text, most students today have neither seen nor held such a disk. Many changes have occurred in the programming languages that students experience in class and in the marketplace. In 1991 the book was available in two forms, with algorithms presented in either C or Ada. While C remains a popular language, Ada has become relatively obscure and did not achieve its predicted popularity. The C++ language evolved from C with the addition of object-oriented features. Java™ was developed as a simpler object-oriented language, gaining popularity because of its security and ability to be run within a Web browser. The College Board Advanced Placement curriculum moved from Pascal to C++ to Java.

While much has changed, students and faculty alike continue to study and teach the subject of compiler construction. Research in the area of compilers and programing language translation continues at a brisk pace, as compilers are tasked with accommodating an increasing diversity of architectures and programming languages. Software development environments depend on compilers interacting successfully with a variety of software toolchain components such as syntax-informed editors, performance profilers, and debuggers. All modern software efforts rely on their compilers to check vigorously for errors and to translate programs faithfully.

Some texts experience relatively minor changes over time, acquiring perhaps some new exercises or examples. This book reflects a substantive revision of the material from 1988 and 1991. While the focus of this text remains on teaching the fundamentals of compiler construction, the algorithms and approaches have been brought into modern practice:

- Coverage of topics that have faded from practical use (e.g., **attribute grammars**) has been minimized or removed altogether.
- Algorithms are presented in a **pseudocode** style that should be familiar to students who have studied the fundamental algorithms of our discipline.

Pseudocode enables a concise formulation of an algorithm and a rational discussion of the algorithm's purpose and construction.

The details of implementation in a particular language have been relegated to the *Crafting a Compiler Supplement* which is available online:

```
http://www.pearsonhighered.com/fischer/
```

- Parsing theory and practice are organized to facilitate a variety of pedagogical approaches.

  Some may study the material at a high level to gain a broad view of top-down and bottom-up parsing. Others may study a particular approach in greater detail.

- The front- and back-end phases of a compiler are connected by the **abstract syntax tree** (AST), which is created as the primary artifact of parsing. Most compilers build an AST, but relatively few texts articulate its construction and use.

  The **visitor pattern** is introduced for traversing the AST during semantic analysis and code generation.

- Laboratory and studio exercises are available to instructors.

  Instructors can assign some components as exercises for the students while other components are supplied from our course-support Web site.

Some texts undergo revision by the addition of more graduate-level material. While such information may be useful in an advanced course, the focus of *Crafting a Compiler* remains on the undergraduate-level study of compiler construction. A graduate course could be offered using Chapters 13 and 14, with the earlier portions of the text serving as reference material.

## Text and Reference

As a classroom text, this book is oriented toward a curriculum that we have developed over the past 25 years. The book is very flexible and has been adopted for courses ranging from a three-credit upper-level course taught in a ten-week quarter to a six-credit semester-long graduate course. The text is accessible to any student who has a basic background in programming, algorithms, and data structures. The text is well suited to a single semester or quarter offering because its flexibility allows an instructor to craft a syllabus according to his or her interests. Author-sponsored solutions are available for those components that are not studied in detail. It is feasible to write portions of a compiler from parsing to code generation in a single semester.

This book is also a valuable professional reference because of its complete coverage of techniques that are of practical importance to compiler construction. Many of our students have reported, even some years after their graduation, of their successful application of these techniques to problems they encounter in their work.

## Instructor Resources

The Web site for this book can be found at `http://www.pearsonhighered.com/fischer/`. The material posted for qualified instructors includes sample laboratory and project assignments, studio (active-learning) sessions, libraries of code that can be used as class-furnished solutions, and solutions to selected exercises.

For access to these materials, qualified instructors should contact their local Pearson Representative by visiting `http://www.pearsonhighered.com`, by sending email to `computing@aw.com`, or by visiting the Pearson Instructor Resource Center at `http://www.pearsonhighered.com/irc/`.

## Student Resources

The book's Web site at `http://www.pearsonhighered.com/fischer/` contains working code for examples used throughout the book, including code for the toy language ac that is introduced in Chapter 2. The site also contains tutorial notes and a page with links to various compiler-construction tools.

Access to these materials may be guarded by a password that is distributed with the book or obtained from an instructor.

## Project Approach

This book offers a comprehensive coverage of relevant theoretical topics in compiler construction. However, a cohesive implementation project is typically an important aspect of planning a curriculum in compiler construction. Thus, the book and the online materials are biased in favor of a sequence of exploratory exercises, culminating in a project, to support learning this material.

Lab exercises, studio sessions, and course projects appear in the *Crafting a Compiler Supplement*, and readers are invited to send us other materials or links for posting at our Web site. The exercises parallel the chapters and progression of material presented in the text. For example, Chapter 2 introduces the toy

language ac to give an overview of the compilation process. The Web site contains full, working versions of the scanner, parser, semantics analyzer, and code generator for that language. These components will be available in a variety of source programming languages.

The Web site also offers material in support of developing a working compiler for a simple language modeled after Java. This allows instructors to assign some components as exercises while other components are provided to fill in any gaps. Some instructors may provide the entire compiler and ask students to implement extensions. Polishing and refining existing components can also be the basis of class projects.

## Pseudocode and Guides

A significant change from the Fischer and LeBlanc text is that algorithms are no longer presented in any specific programming language such as C or Ada. Instead, algorithms are presented in **pseudocode** using a style that should be familiar to those who have studied even the most fundamental algorithms [CLRS01]. Pseudocode simplifies the exposition of an algorithm by omitting unnecessary detail. However, the pseudocode is suggestive of constructs used in real programming languages, so implementation should be straightforward. An index of all pseudocode methods is provided as a **guide** at the end of this book.

The text makes extensive use of abbreviations (including acronyms) to simplify exposition and to help readers acquire the terminology used in compiler construction. Each abbreviation is fully defined automatically at its first reference in each chapter. For example, AST has already been used in this preface, as an abbreviation of abstract syntax tree, but **context-free grammar** (CFG) has not. For further help, an index of all abbreviations appears as a **guide** at the end of the book. The full index contains abbreviations and indicates where they are referenced throughout the book. Terms such as **guide** are shown in boldface. Each reference to such terms is included in the full index.

## Using this Book

An introductory course on compiler construction could begin with Chapters 1, 2, and 3. For parsing technique, either top-down (Chapter 5) or bottom-up (Chapter 6) could be chosen, but some instructors will choose to cover both. Material from Chapter 4 can be covered as necessary to support the parsing techniques that will be studied. Chapter 7 articulates the AST and presents the **visitor pattern** for its traversal. Some instructors may assign AST-management utilities as a lab exercise, while others may use the utilities provided by the

Web site. Various aspects of semantic analysis can then be covered at the instructor's discretion in Chapters 8 and 9. A quarter-based course could end here, with another quarter continuing with the study of code generation, as described next.

Chapter 10 provides an overview of the **Java Virtual Machine** (JVM), which should be covered if students will generate JVM code in their project. Code generation for such virtual machines is covered in Chapter 11. Instructors who prefer students to generate machine code could skip Chapters 10 and 11 and cover Chapters 12 and 13 instead. An introductory course could include material from the beginning of Chapter 14 on automatic program optimization.

Further study could include more detail of the parsing techniques covered in Chapters 4, 5, and 6. Semantic analysis and type checking could be studied in greater breadth and depth in Chapters 8 and 9. Advanced concepts such as **static single assignment** (SSA) Form could be introduced from Chapters 10 and 14. Advanced topics in program analysis and transformation, including data flow frameworks, could be drawn from Chapter 14. Chapters 13 and 14 could be the basis for a gradute compiler course, with earlier chapters providing useful reference material.

## Chapter Descriptions

### *Chapter 1 Introduction*

The text begins with an overview of the compilation process. The concepts of constructing a compiler from a collection of components are emphasized. An overview of the history of compilers is presented and the use of tools for generating compiler components is introduced.

### *Chapter 2 A Simple Compiler*

The simple language ac is presented, and each of the compiler's components is discussed with respect to translating ac to another language, dc. These components are presented in pseudocode, and complete code can be found in the *Crafting a Compiler Supplement*.

### *Chapter 3 Scanning—Theory and Practice*

The basic concepts and techniques for building the lexical analysis components of a compiler are presented. This discussion includes the development of hand-coded scanners as well as the use of scanner-generation tools for implementing table-driven lexical analyzers.

### *Chapter 4 Grammars and Parsing*

This chapter covers the fundamentals of formal language concepts, including context-free grammars, grammar notation, derivations, and parse trees. Grammar-analysis algorithms are introduced that are used in Chapters 5 and 6.

### *Chapter 5 Top-Down Parsing*

Top-down parsing is a popular technique for constructing relatively simple parsers. This chapter shows how such parsers can be written using explicit code or by constructing a table for use by a generic top-down parsing engine. Syntactic error diagnosis, recovery, and repair are discussed.

### *Chapter 6 Bottom-Up Parsing*

Most compilers for modern programming languages use one of the bottom-up parsing techniques presented in this chapter. Tools for generating such parsers automatically from a context-free grammar are widely available. The chapter describes the theory on which such tools are built, including a sequence of increasingly sophisticated approaches to resolving **conflicts** that hamper parser construction for some grammars. Grammar and language **ambiguity** are thoroughly discussed, and heuristics are presented for understanding and resolving ambiguous grammars.

### *Chapter 7 Syntax-Directed Translation*

This marks the mid-point of the book in terms of a compiler's components. Prior chapters have considered the lexical and syntactic analysis of programs. A goal of those chapters is the construction of an AST. In this chapter, the AST is introduced and an interface is articulated for constructing, managing, and traversing the AST. This chapter is pivotal in the sense that subsequent chapters depend on understanding both the AST and the **visitor pattern** that facilitates traversal and processing of the AST. The *Crafting a Compiler Supplement* contains a tutorial on the visitor pattern, including examples drawn from common experiences.

### *Chapter 8 Symbol Tables and Declaration Processing*

This chapter emphasizes the use of a symbol table as an abstract component that can be utilized throughout the compilation process. A precise interface is defined for the symbol table, and various implementation issues and ideas are presented. This discussion includes a study of the implementation of nested scopes.

The semantic analysis necessary for processing symbol declarations is introduced, including types, variables, arrays, structures, and enumerations. An introduction to type checking is presented, including object-oriented classes, subclasses, and superclasses.

### *Chapter 9 Semantic Analysis*

Additional semantic analysis is required for language specifications that are not easily checked while parsing. Various control structures are examined, including conditional branches and loops. The chapter includes a discussion of exceptions and the semantic analysis they require at compile-time.

### *Chapter 10 Intermediate Representations*

This chapter considers two intermediate representations are are widely used by compilers. The first is the JVM instruction set and bytecode format, which has become the standard format for representing compiled Java programs. For readers who are interested in targeting the JVM in a compiler project, Chapters 10 and 11 provide the necessary background and techniques. The other representation is SSA Form, which is used by many optimizing compilers. This chapter defines SSA Form, but its construction is delayed until Chapter 14, where some requisite definitions and algorithms are presented.

### *Chapter 11 Code Generation for a Virtual Machine*

This chapter considers code generation for a **virtual machine** (VM). The advantages of considering such a target is that many of the details of runtime support are subsumed by the VM. For example, most VMs offer an unlimited number of registers, so that the issue of **register allocation**, albeit interesting, can be postponed until the fundamentals of code generation are mastered. The VM's instruction set is typically at a higher level than machine code. For example, a method call is often supported by a single VM instruction, while the same call would require many more instructions in machine code.

While an eager reader interested in generating machine code may be tempted to skip Chapter 11, we recommend studying this chapter before attempting code generation at the machine-code level. The ideas from this chapter are easily applied to Chapters 12 and 13, but they are easier to understand from the perspective of a VM.

### *Chapter 12 Runtime Support*

Much of the functionality embedded in a VM is its runtime support (e.g., its support for managing storage). This chapter discusses various concepts and implementation strategies for providing the runtime support needed for modern programming languages. Study of this material can provide an understanding of the construction of a VM. For those who write code generators for a target architecture (Chapter 13), runtime support must be provided, so the study of this material is essential to creating a working compiler.

The chapter includes discussion of storage that is statically allocated, stack allocated, and heap allocated. References to **nonlocal storage** are considered, along with implementation structures such as frames and displays to support such references.

### *Chapter 13 Target Code Generation*

This chapter is similar to Chapter 11, except that the target of code generation is a relatively low-level instruction set when compared with a VM. The chapter includes a thorough discussion of topics that arise in such code generation, including register allocation, management of temporaries, code scheduling, instruction selection, and some basic peephole optimization.

*Chapter 14 Program Optimization*

Most compilers include some capability for improving the code they generate. This chapter considers some of the practical techniques commonly used by compilers for program optimization. Advanced control flow analysis structures and algorithms are presented. An introduction to **data flow analysis** is presented by considering some fundamental optimizations that are relatively easy to implement. The theoretical foundation of such optimizations is studied, and the chapter includes construction and use of SSA Form.

## Acknowledgements

We collectively thank the following people who have supported us in preparing this text. We thank Matt Goldstein of Pearson Publishing for his patience and support throughout the revision process. We apologize to Matt's predecessors for our delay in preparing this text. Jeff Holcomb provided technical guidance in Pearson's publication process, for which we are very grateful. Our text was greatly improved at the hands of our copy editors. Stephanie Moscola expeditiously and expertly proofread and corrected every chapter of this text. She was extraordinarily thorough, and any remaining errors are the authors' fault. We are grateful for her keen eye and insightful suggestions. We thank Will Benton for his editing of Chapters 12 and 13 and his authoring of Section 12.5. We thank Aimee Beal who was retained by Pearson to copyedit this book for style and consistency.

We are very grateful to the following colleagues for their time spent reviewing our work and providing valuable feedback: Ras Bodik (University of California–Berkeley), Scott Cannon (Utah State University), Stephen Edwards (Columbia University), Stephen Freund (Williams College), Jerzy Jaromczyk (University of Kentucky), Hikyoo Koh (Lamar University), Sam Midkiff (Purdue University), Tim O'Neil (University of Akron), Kurt Stirewalt (Michigan State University), Michelle Strout (Colorado State University), Douglas Thain (University of Notre Dame), V. N. Venkatakrishnan (University of Illinois–Chicago), Elizabeth White (George Mason University), Sherry Yang (Oregon Institute of Technology), and Qing Yi (University of Texas–San Antonio).

**Charles Fischer** My fascination with compilers began in 1965 in Mr. Robert Eddy's computer lab. Our computer had all of 20 kilobytes of main memory, and our compiler used punched cards as its intermediate form, but the seed was planted.

My education really began at Cornell University, where I learned the depth and rigor of computing. David Gries' seminal compiler text taught me much and set me on my career path.

The faculty at Wisconsin, especially Larry Landweber and Tad Pinkerton, gave me free rein in developing a compiler curriculum and research program. Tad, Larry Travis and Manley Draper, at the Academic Computing Center, gave me the time and resources to learn the practice of compiling. The UW-Pascal compiler project introduced me to some outstanding students, including my co-author Richard LeBlanc. We learned by doing, and that became my teaching philosophy.

Over the years my colleagues, especially Tom Reps, Susan Horwitz, and Jim Larus, freely shared their wisdom and experience; I learned much. On the architectural side, Jim Goodman, Guri Sohi, Mark Hill, and David Wood taught me the subtleties of modern microprocessors. A compiler writer must thoroughly understand a processor to harness its full power.

My greatest debt is to my students who brought enormous energy and enthusiasm to my courses. They eagerly accepted the challenges I presented. A full compiler, from scanner to code generator, must have seemed impossible in one semester, but they did it, and did it well. Much of that experience has filtered its way into this text. I trust it will be helpful in teaching a new generation how to craft a compiler.

**Ron K. Cytron** My initial interest and subsequent research into programming languages and their compilers are due in large part to the outstanding mentors who have played pivotal roles in my career. Ken Kennedy, of blessed memory, taught my compilers classes at Rice University. The courses I now teach are patterned after his approach, especially the role that lab assignments play in helping students understand the material. Ken Kennedy was an outstanding educator, and I can only hope to connect with students as well as he could. He hosted me one summer at IBM T.J. Watson Research Labs, in Yorktown Heights, New York, where I worked on software for automatic parallelization. During that summer my investigations naturally led me to the research of Dave Kuck and his students at the University of Illinois.

I still consider myself so very fortunate that Dave took me on as his graduate student. Dave Kuck is a pioneer in parallel computer architecture and in the role compilers can play to make to make such advanced systems easier to program. I strive to follow his example of hard work, integrity, and perseverance and to pass those lessons on to my students. I also experienced the vibrancy and fun that stems from investigating ideas in a group, and I have tried to create similar communities among my students.

My experiences as an undergraduate and graduate student then led me to Fran Allen of IBM Research, to whom I shall always be grateful for allowing me to join her newly formed PTRAN group. Fran has inspired generations of research in data flow analysis, program optimization, and automatic parallelization. She has amazing intuition into the important problems and their

likely solution. In talking with colleagues, some of our best ideas are due to Fran and the suggestions, advice, or critiques she has offered us.

Some of the best years of my professional life were spent learning from and working with Fran and my PTRAN colleagues: Michael Burke, Philippe Charles, Jong-Deok Choi, Jeanne Ferrante, Vivek Sarkar, and David Shields. At IBM I also had the privilege of learning from and working with Barry Rosen, Mark Wegman, and Kenny Zadeck. While the imprint of my friends and colleagues can be found throughout this text, any mistakes are mine.

If the reader notices that the number 431 appears frequently in this book, it is an homage to the students who have studied compilers with me at Washington University. I have learned as much from my students as I have taught them, and my contribution to this book stems largely from my experiences in the classroom and lab.

Finally, I thank my wife and children for putting up with the time I wanted to spend working on this book. They have shown patience and understanding throughout this effort. And thank you, Aunt Carole, for always asking how this book was coming along.

**Richard LeBlanc** After becoming more excited about computers than physics problem sets while getting my B.S. in physics, I moved to Madison and enrolled at the University of Wisconsin as a computer science Ph.D. student in 1972. Two years later, a young assistant professor, Charles Fischer, who had just received his Ph.D. from Cornell, joined the faculty of the Computer Science Department. The first course he taught was a graduate compiler course, CS 701. I was enrolled in that course and still remember it as a really remarkable learning experience, all the more impressive since it was his first time teaching the course. We obviously hit it off well, since this introduction has led to a rather lengthy series of collaborations.

Through the sponsorship of Larry Travis, I began working at the Academic Computing Center in the summer of 1974. I was thus already part of that organization when the UW-Pascal project began a year later. That project not only gave me the opportunity to apply what I had learned in the two courses I had just taken, but also some great lessons about the impact of good design and design reviews. I also benefited from working with two fellow graduate students, Steve Zeigler and Marty Honda, from whom I learned how much fun it can be to be part of an effective software development team. We all discovered the value of working in Pascal, a well-designed language that requires disciplined thought while programming, and of using a tool that you are developing, since we bootstrapped from the Pascal P-Compiler to our own compiler that generated native code for the Univac 1108 early in the project.

Upon completion of my graduate work, I took a faculty position at Georgia Tech, drawn by the warmer weather and an opportunity to be part of

a distributed computing research project led by Phil Enslow, who provided invaluable guidance in the early years of my career. I immediately had the opportunity to teach a compiler course and attempted to emulate the CS 701 course at Wisconsin, since I strongly believed in the value of the project-based approach Charles used. I quickly realized that that having the students write a complete compiler in a 10-week quarter was too much of a challenge. I thus began using the approach of giving them a working compiler for a very tiny language and building the project around extending all of the components of that compiler to compile a more complex language. The base compiler that I used in my 10-week course became one of the support items distributed with the Fischer–LeBlanc text.

My career path has taken me to greater involvement with software engineering and educational activities than with compiler research. As I look back on my early compiler experiences at Wisconsin, I clearly see the seeds of my interests in both of these areas. The decision that Charles and I made to write the original *Crafting a Compiler* was based in our belief that we could help other instructors offer their students an outstanding educational experience through a project-based compiler course. With the invaluable help of our editor, Alan Apt, and a great set of reviewers, I believe we succeeded. Many colleagues have expressed to me their enthusiasm for our original book and *Crafting a Compiler with C*. Their support has been a great reward and it also served as encouragement toward finally completing this text. Particular thanks go to Joe Bergin, who went well beyond verbal support, translating some of our early software tools into new programming languages and allowing us to make his versions available to other instructors.

My years at Georgia Tech provided me with wonderful opportunities to develop my interests in computing education. I was fortunate to have been part of an organization led by Ray Miller and then Pete Jensen during the first part of my career. Beginning in 1990, I had the great pleasure of working with Peter Freeman as we created and developed the College of Computing. Beyond the many ways he mentored me during our work at Georgia Tech, Peter encouraged my broad involvement with educational issues through my work with the ACM Education Board, which has greatly enriched my professional life over the last 12 years.

Finally, I thank my family, including my new granddaughter, for sharing me with this book writing project, which at times must have seemed like it would never end.

# 出 版 说 明

进入21世纪，世界各国的经济、科技以及综合国力的竞争将更加激烈。竞争的中心无疑是对人才的竞争。谁拥有大量高素质的人才，谁就能在竞争中取得优势。高等教育，作为培养高素质人才的事业，必然受到高度重视。目前我国高等教育的教材更新较慢，为了加快教材的更新频率，教育部正在大力促进我国高校采用国外原版教材。

清华大学出版社从1996年开始，与国外著名出版公司合作，影印出版了“大学计算机教育丛书（影印版）”等一系列引进图书，受到国内读者的欢迎和支持。跨入21世纪，我们本着为我国高等教育教材建设服务的初衷，在已有的基础上，进一步扩大选题内容，改变图书开本尺寸，一如既往地请有关专家挑选适用于我国高校本科及研究生计算机教育的国外经典教材或著名教材，组成本套“大学计算机教育国外著名教材系列（影印版）”，以飨读者。深切期盼读者及时将使用本系列教材的效果和意见反馈给我们。更希望国内专家、教授积极向我们推荐国外计算机教育的优秀教材，以利我们把“大学计算机教育国外著名教材系列（影印版）”做得更好，更适合高校师生的需要。

清华大学出版社

# *Brief Contents*


| | | |
|---|---|---|
| 1 | *Introduction* | 1 |
| 2 | *A Simple Compiler* | 31 |
| 3 | *Scanning—Theory and Practice* | 57 |
| 4 | *Grammars and Parsing* | 113 |
| 5 | *Top-Down Parsing* | 143 |
| 6 | *Bottom-Up Parsing* | 179 |
| 7 | *Syntax-Directed Translation* | 235 |
| 8 | *Symbol Tables and Declaration Processing* | 279 |
| 9 | *Semantic Analysis* | 343 |
| 10 | *Intermediate Representations* | 391 |
| 11 | *Code Generation for a Virtual Machine* | 417 |
| 12 | *Runtime Support* | 445 |
| 13 | *Target Code Generation* | 489 |
| 14 | *Program Optimization* | 547 |

# Contents


## 1 Introduction 1

1.1 History of Compilation . . . . . 2
1.2 What Compilers Do . . . . . 4
1.2.1 Machine Code Generated by Compilers . . . . . 4
1.2.2 Target Code Formats . . . . . 7
1.3 Interpreters . . . . . 9
1.4 Syntax and Semantics . . . . . 10
1.4.1 Static Semantics . . . . . 11
1.4.2 Runtime Semantics . . . . . 12
1.5 Organization of a Compiler . . . . . 14
1.5.1 The Scanner . . . . . 16
1.5.2 The Parser . . . . . 16
1.5.3 The Type Checker (Semantic Analysis) . . . . . 17
1.5.4 Translator (Program Synthesis) . . . . . 17
1.5.5 Symbol Tables . . . . . 18
1.5.6 The Optimizer . . . . . 18
1.5.7 The Code Generator . . . . . 19
1.5.8 Compiler Writing Tools . . . . . 19
1.6 Programming Language and Compiler Design . . . . . 20
1.7 Computer Architecture and Compiler Design . . . . . 21
1.8 Compiler Design Considerations . . . . . 22
1.8.1 Debugging (Development) Compilers . . . . . 22
1.8.2 Optimizing Compilers . . . . . 23
1.8.3 Retargetable Compilers . . . . . 23
1.9 Integrated Development Environments . . . . . 24
Exercises . . . . . 26