

<<编译原理>>

图书基本信息

书名：<<编译原理>>

13位ISBN编号：9787111326748

10位ISBN编号：7111326741

出版时间：2011-1

出版时间：机械工业出版社

作者：Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

页数：1009

版权说明：本站所提供下载的PDF图书仅提供预览和简介，请支持正版图书。

更多资源请访问：<http://www.tushu007.com>

前言

In the time since the 1986 edition of this book, the world of compiler design has changed significantly. Programming languages have evolved to present new compilation problems. Computer architectures offer a variety of resources of which the compiler designer must take advantage. Perhaps most interestingly, the venerable technology of code optimization has found use outside compilers. It is now used in tools that find bugs in software, and most importantly, find security holes in existing code. And much of the "front-end" technology — grammars, regular expressions, parsers, and syntax-directed translators — are still in wide use. Thus, our philosophy from previous versions of the book has not changed. We recognize that few readers will build, or even maintain, a compiler for a major programming language. Yet the models, theory, and algorithms associated with a compiler can be applied to a wide range of problems in software design and software development. We therefore emphasize problems that are most commonly encountered in designing a language processor, regardless of the source language or target machine. Use of the Book It takes at least two quarters or even two semesters to cover all or most of the material in this book. It is common to cover the first half in an undergraduate course and the second half of the book — stressing code optimization — in a second course at the graduate or mezzanine level. Here is an outline of the chapters: Chapter 1 contains motivational material and also presents some background issues in computer architecture and programming-language principles. Chapter 2 develops a miniature compiler and introduces many of the important concepts, which are then developed in later chapters. The compiler itself appears in the appendix. Chapter 3 covers lexical analysis, regular expressions, finite-state machines, and scanner-generator tools. This material is fundamental to text-processing of all sorts.

<<编译原理>>

内容概要

本书是编译领域无可替代的经典著作，被广大计算机专业人士誉为“龙书”。本书上一版自1986年出版以来，被世界各地的著名高等院校和研究机构（包括美国哥伦比亚大学、斯坦福大学、哈佛大学、普林斯顿大学、贝尔实验室）作为本科生和研究生的编译原理课程的教材。该书对我国高等计算机教育领域也产生了重大影响。

第2版对每一章都进行了全面的修订，以反映自上一版出版20多年来软件工程专业设计语言和计算机体系结构方面的发展对编译技术的影响。本书全面介绍了编译器的设计，并强调编译技术在软件设计和开发中的广泛应用。每章中都包含大量的习题和丰富的参考文献。

本书适合作为高等院校计算机专业本科生和研究生的编译原理与技术课程的教材，也可供广大计算机技术人员参考。

cd中包含如下内容：

- comptia project+考试模拟题。
- 作者的项目管理培训视频。
- 项目管理工作表和模板。

<<编译原理>>

作者简介

Alfred

V. Aho, 美国哥伦比亚大学教授, 美国国家工程院院士, ACM和IEEE会士, 曾获得IEEE的冯·诺伊曼奖。

著有多部算法、数据结构、编译器、数据库系统及计算机科学基础方面的著作。

<<编译原理>>

书籍目录

- 1 introduction
 - 1.1 language processors
 - 1.2 the structure of a compiler
 - 1.3 the evolution of programming languages
 - 1.4 the science of building a compiler
 - 1.5 applications of compiler technology
 - 1.6 programming language basics
 - 1.7 summary of chapter 1
 - 1.8 references for chapter 1
- 2 a simple syntax-directed translator
 - 2.1 introduction
 - 2.2 syntax definition
 - 2.3 syntax-directed translation
 - 2.4 parsing
 - 2.5 a translator for simple expressions
 - 2.6 lexical analysis
 - 2.7 symbol tables
 - 2.8 intermediate code generation
 - 2.9 summary of chapter 2
- 3 lexical analysis
 - 3.1 the role of the lexical analyzer
 - 3.2 input buffering
 - 3.3 specification of tokens
 - 3.4 recognition of tokens
 - 3.5 the lexical-analyzer generator lex
 - 3.6 finite automata
 - 3.7 from regular expressions to automata
 - 3.8 design of a lexical-analyzer generator
 - 3.9 optimization of dfa-based pattern matchers
 - 3.10 summary of chapter 3
 - 3.11 references for chapter 3
- 4 syntax analysis
 - 4.1 introduction
 - 4.2 context-free grammars
 - 4.3 writing a grammar
 - 4.4 top-down parsing
 - 4.5 bottom-up parsing
 - 4.6 introduction to lr parsing: simple lr
 - 4.7 more powerful lr parsers
 - 4.8 using ambiguous grammars
 - 4.9 parser generators
 - 4.10 summary of chapter 4
 - 4.11 references for chapter 4
- 5 syntax-directed translation
 - 5.1 syntax-directed definitions

<<编译原理>>

- 5.2 evaluation orders for sdd's
- 5.3 applications of syntax-directed translation
- 5.4 syntax-directed translation schemes
- 5.5 implementing l-attributed sdd's
- 5.6 summary of chapter 5
- 5.7 references for chapter 5
- 6 intermediate-code generation
 - 6.1 variants of syntax trees
 - 6.2 three-address code
 - 6.3 types and declarations
 - 6.4 translation of expressions
 - 6.5 type checking
 - 6.6 control flow
 - 6.7 backpatching
 - 6.8 switch-statements
 - 6.9 intermediate code for procedures
 - 6.10 summary of chapter 6
 - 6.11 references for chapter 6
- 7 run-time environments
 - 7.1 storage organization
 - 7.2 stack allocation of space
 - 7.3 access to nonlocal data on the stack
 - 7.4 heap management
 - 7.5 introduction to garbage collection
 - 7.6 introduction to trace-based collection
 - 7.7 short-pause garbage collection
 - 7.8 advanced topics in garbage collection
 - 7.9 summary of chapter 7
 - 7.10 references for chapter 7
- 8 code generation
 - 8.1 issues in the design of a code generator
 - 8.2 the target language
 - 8.3 addresses in the target code
 - 8.4 basic blocks and flow graphs
 - 8.5 optimization of basic blocks
 - 8.6 a simple code generator
 - 8.7 peephole optimization
 - 8.8 register allocation and assignment
 - 8.9 instruction selection by tree rewriting
 - 8.10 optimal code generation for expressions
 - 8.11 dynamic programming code-generation
 - 8.12 summary of chapter 8
 - 8.13 references for chapter 8
- 9 machine-independent optimizations
 - 9.1 the principal sources of optimization
 - 9.2 introduction to data-flow analysis
 - 9.3 foundations of data-flow analysis

<<编译原理>>

- 9.4 constant propagation
- 9.5 partial-redundancy elimination
- 9.6 loops in flow graphs
- 9.7 region-based analysis
- 9.8 symbolic analysis
- 9.9 summary of chapter 9
- 9.10 references for chapter 9
- 10 instruction-level parallelism
 - 10.1 processor architectures
 - 10.2 code-scheduling constraints
 - 10.3 basic-block scheduling
 - 10.4 global code scheduling
 - 10.5 software pipelining
 - 10.6 summary of chapter 10
 - 10.7 references for chapter 10
- 11 optimizing for parallelism and locality
 - 11.1 basic concepts
 - 11.2 matrix multiply: an in-depth example
 - 11.3 iteration spaces
 - 11.4 affine array indexes
 - 11.5 data reuse
 - 11.6 array data-dependence analysis
 - 11.7 finding synchronization-free parallelism
 - 11.8 synchronization between parallel loops
 - 11.9 pipelining
 - 11.10 locality optimizations
 - 11.11 other uses of affine transforms
 - 11.12 summary of chapter 11
 - 11.13 references for chapter 11
- 12 interprocedural analysis
 - 12.1 basic concepts
 - 12.2 why interprocedural analysis?
 - 12.3 a logical representation of data flow
 - 12.4 a simple pointer-analysis algorithm
 - 12.5 context-insensitive interprocedural analysis
 - 12.6 context-sensitive pointer analysis
 - 12.7 datalog implementation by bdd's
 - 12.8 summary of chapter 12
 - 12.9 references for chapter 12
- a a complete front end
 - a.1 the source language
 - a.2 main
 - a.3 lexical analyzer
 - a.4 symbol tables and types
 - a.5 intermediate code for expressions
 - a.6 jumping code for boolean expressions
 - a.7 intermediate code for statements

<<编译原理>>

a.8 parser

a.9 creating the front end

b finding linearly independent solutions

index

章节摘录

插图：Language, are used to search databases. Database queries consist of predicates containing relational and boolean operators. They can be interpreted or compiled into commands to search a database for records satisfying that predicate. Compiled Simulation Simulation is a general technique used in many scientific and engineering disciplines to understand a phenomenon or to validate a design. Inputs to a simulator usually include the description of the design and specific input parameters for that particular simulation run. Simulations can be very expensive. We typically need to simulate many possible design alternatives on many different input sets, and each experiment may take days to complete on a high-performance machine. Instead of writing a simulator that interprets the design, it is faster to compile the design to produce machine code that simulates that particular design natively. Compiled simulation can run orders of magnitude faster than an interpreter-based approach. Compiled simulation is used in many state-of-the-art tools that simulate designs written in Verilog or VHDL.

1.5.5 Software Productivity Tools

Programs are arguably the most complicated engineering artifacts ever produced; they consist of many many details, every one of which must be correct before the program will work completely. As a result, errors are rampant in programs; errors may crash a system, produce wrong results, render a system vulnerable to security attacks, or even lead to catastrophic failures in critical systems. Testing is the primary technique for locating errors in programs. An interesting and promising complementary approach is to use data-flow analysis to locate errors statically (that is, before the program is run). Data-flow analysis can find errors along all the possible execution paths, and not just those exercised by the input data sets, as in the case of program testing. Many of the data-flow-analysis techniques, originally developed for compiler optimizations, can be used to create tools that assist programmers in their software engineering tasks. The problem of finding all program errors is undecidable. A data-flow analysis may be designed to warn the programmers of all possible statements with a particular category of errors. But if most of these warnings are false alarms, users will not use the tool. Thus, practical error detectors are often neither sound nor complete. That is, they may not find all the errors in the program, and not all errors reported are guaranteed to be real errors. Nonetheless, various static analyses have been developed and shown to be effective in finding errors, such as dereferencing null or freed pointers, in real programs. The fact that error detectors may be unsound makes them significantly different from compiler optimizations. Optimizers must be conservative and cannot alter the semantics of the program under any circumstances.

<<编译原理>>

编辑推荐

《编译原理(英文版·第2版)》：经典原版书库

<<编译原理>>

版权说明

本站所提供下载的PDF图书仅提供预览和简介，请支持正版图书。

更多资源请访问:<http://www.tushu007.com>