

TURING

图灵原版计算机科学系列

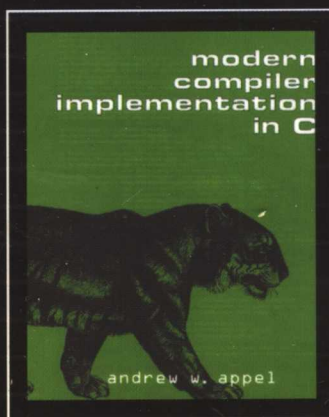
Modern Compiler Implementation in C

现代编译原理

——C语言描述

(英文版)

[美] Andrew W. Appel 著



人民邮电出版社
POSTS & TELECOM PRESS

TURING

图灵原版计算机科学系列

Modern Compiler Implementation in C

现代编译原理

——C 语言描述

(英文版)

[美] Andrew W. Appel 著
Maia Ginsburg

人民邮电出版社

图书在版编目 (CIP) 数据

现代编译原理: C 语言描述 / (美) 阿佩尔 (Appel, A. W.), (美) 金斯伯格 (Ginsburg, M.) 著.
—北京: 人民邮电出版社, 2005.9

(图灵原版计算机科学系列)

ISBN 7-115-13771-4

I. 现... II. ①阿... ②金... III. ①编译程序—程序设计—英文 ②C 语言—程序设计—英文 IV. TP314

中国版本图书馆 CIP 数据核字 (2005) 第 092228 号

版 权 声 明

Andrew W. Appel and Maia Ginsburg: Modern Compiler Implementation in C (ISBN: 0-521-60765-5).
Originally published by Cambridge University Press in 1998.

This reprint edition is published with the permission of the Syndicate of the Press of the University of Cambridge, Cambridge, England.

Copyright ©1998 by Andrew W. Appel and Maia Ginsburg.

This edition is licensed for distribution and sale in the People's Republic of China only, excluding Hong Kong, Taiwan and Macao and may not be distributed and sold elsewhere.

本书原书由剑桥大学出版社出版。

本书英文影印版由剑桥大学出版社授权人民邮电出版社出版。

此版仅限在中华人民共和国境内 (香港、台湾、澳门地区除外) 销售发行。

版权所有, 侵权必究。

图灵原版计算机科学系列

现代编译原理—C 语言描述 (英文版)

-
- ◆ 著 [美] Andrew W. Appel Maia Ginsburg
责任编辑 陈贤舜
 - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销
 - ◆ 开本: 800×1000 1/16
印张: 34.75
字数: 764 千字 2005 年 9 月第 1 版
印数: 1—3 000 册 2005 年 9 月北京第 1 次印刷

著作权合同登记号 图字: 01-2005-4374 号

ISBN 7-115-13771-4/TP · 4891

定价: 59.00 元

读者服务热线: (010) 67132705 印装质量热线: (010) 67129223

内 容 提 要

本书全面讲述了现代编译器的各个组成部分，包括：词法分析、语法分析、抽象语法、语义检查、中间代码表示、指令选择、数据流分析、寄存器分配以及运行时系统等。全书分成两部分，第一部分是编译的基础知识，适用于第一门编译原理课程（一个学期）；第二部分是高级主题，包括面向对象语言和函数语言、垃圾收集、循环优化、SSA（静态单赋值）形式、循环调度、存储结构优化等，适合于后续课程或研究生教学。书中专门为学生提供了一个用 C 语言编写的实习项目，包括前端和后端设计，学生可以在一学期内创建一个功能完整的编译器。

本书适用于高等院校计算机及相关专业的本科生或研究生，也可供科研人员或工程技术人员参考。

Preface

Over the past decade, there have been several shifts in the way compilers are built. New kinds of programming languages are being used: object-oriented languages with dynamic methods, functional languages with nested scope and first-class function closures; and many of these languages require garbage collection. New machines have large register sets and a high penalty for memory access, and can often run much faster with compiler assistance in scheduling instructions and managing instructions and data for cache locality.

This book is intended as a textbook for a one- or two-semester course in compilers. Students will see the theory behind different components of a compiler, the programming techniques used to put the theory into practice, and the interfaces used to modularize the compiler. To make the interfaces and programming examples clear and concrete, I have written them in the C programming language. Other editions of this book are available that use the Java and ML languages.

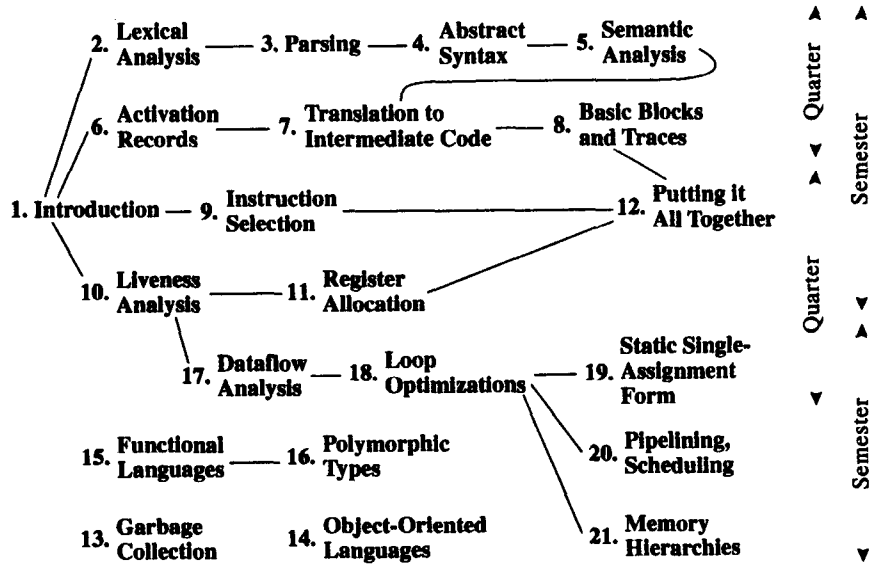
Implementation project. The “student project compiler” that I have outlined is reasonably simple, but is organized to demonstrate some important techniques that are now in common use: abstract syntax trees to avoid tangling syntax and semantics, separation of instruction selection from register allocation, copy propagation to give flexibility to earlier phases of the compiler, and containment of target-machine dependencies. Unlike many “student compilers” found in textbooks, this one has a simple but sophisticated back end, allowing good register allocation to be done after instruction selection.

Each chapter in Part I has a programming exercise corresponding to one module of a compiler. Software useful for the exercises can be found at

<http://www.cs.princeton.edu/~appel/modern/c>

Exercises. Each chapter has pencil-and-paper exercises; those marked with a star are more challenging, two-star problems are difficult but solvable, and the occasional three-star exercises are not known to have a solution.

Course sequence. The figure shows how the chapters depend on each other.



- A one-semester course could cover all of Part I (Chapters 1–12), with students implementing the project compiler (perhaps working in groups); in addition, lectures could cover selected topics from Part II.
- An advanced or graduate course could cover Part II, as well as additional topics from the current literature. Many of the Part II chapters can stand independently from Part I, so that an advanced course could be taught to students who have used a different book for their first course.
- In a two-quarter sequence, the first quarter could cover Chapters 1–8, and the second quarter could cover Chapters 9–12 and some chapters from Part II.

Acknowledgments. Many people have provided constructive criticism or helped me in other ways on this book. I would like to thank Leonor Abraido-Fandino, Scott Ananian, Stephen Bailey, Max Hailperin, David Hanson, Jeffrey Hsu, David MacQueen, Torben Mogensen, Doug Morgan, Robert Netzer, Elma Lee Noah, Mikael Petterson, Todd Proebsting, Anne Rogers, Barbara Ryder, Amr Sabry, Mooly Sagiv, Zhong Shao, Mary Lou Soffa, Andrew Tolmach, Kwangkeun Yi, and Kenneth Zadeck.

Contents

Part I Fundamentals of Compilation

1	Introduction	3
1.1	Modules and interfaces	4
1.2	Tools and software	5
1.3	Data structures for tree languages	7
2	Lexical Analysis	16
2.1	Lexical tokens	17
2.2	Regular expressions	18
2.3	Finite automata	21
2.4	Nondeterministic finite automata	24
2.5	Lex: a lexical analyzer generator	30
3	Parsing	39
3.1	Context-free grammars	41
3.2	Predictive parsing	46
3.3	LR parsing	56
3.4	Using parser generators	69
3.5	Error recovery	76
4	Abstract Syntax	88
4.1	Semantic actions	88
4.2	Abstract parse trees	92
5	Semantic Analysis	103
5.1	Symbol tables	103
5.2	Bindings for the Tiger compiler	112

CONTENTS

5.3	Type-checking expressions	115
5.4	Type-checking declarations	118
6	Activation Records	125
6.1	Stack frames	127
6.2	Frames in the Tiger compiler	135
7	Translation to Intermediate Code	150
7.1	Intermediate representation trees	151
7.2	Translation into trees	154
7.3	Declarations	170
8	Basic Blocks and Traces	176
8.1	Canonical trees	177
8.2	Taming conditional branches	185
9	Instruction Selection	191
9.1	Algorithms for instruction selection	194
9.2	CISC machines	202
9.3	Instruction selection for the Tiger compiler	205
10	Liveness Analysis	218
10.1	Solution of dataflow equations	220
10.2	Liveness in the Tiger compiler	229
11	Register Allocation	235
11.1	Coloring by simplification	236
11.2	Coalescing	239
11.3	Precolored nodes	243
11.4	Graph coloring implementation	248
11.5	Register allocation for trees	257
12	Putting It All Together	265
 Part II Advanced Topics		
13	Garbage Collection	273
13.1	Mark-and-sweep collection	273
13.2	Reference counts	278

CONTENTS

13.3	Copying collection	280
13.4	Generational collection	285
13.5	Incremental collection	287
13.6	Baker's algorithm	290
13.7	Interface to the compiler	291
14	Object-Oriented Languages	299
14.1	Classes	299
14.2	Single inheritance of data fields	302
14.3	Multiple inheritance	304
14.4	Testing class membership	306
14.5	Private fields and methods	310
14.6	Classless languages	310
14.7	Optimizing object-oriented programs	311
15	Functional Programming Languages	315
15.1	A simple functional language	316
15.2	Closures	318
15.3	Immutable variables	319
15.4	Inline expansion	326
15.5	Closure conversion	332
15.6	Efficient tail recursion	335
15.7	Lazy evaluation	337
16	Polymorphic Types	350
16.1	Parametric polymorphism	351
16.2	Type inference	359
16.3	Representation of polymorphic variables	369
16.4	Resolution of static overloading	378
17	Dataflow Analysis	383
17.1	Intermediate representation for flow analysis	384
17.2	Various dataflow analyses	387
17.3	Transformations using dataflow analysis	392
17.4	Speeding up dataflow analysis	393
17.5	Alias analysis	402
18	Loop Optimizations	410
18.1	Dominators	413

CONTENTS

18.2	Loop-invariant computations	418
18.3	Induction variables	419
18.4	Array-bounds checks	425
18.5	Loop unrolling	429
19	Static Single-Assignment Form	433
19.1	Converting to SSA form	436
19.2	Efficient computation of the dominator tree	444
19.3	Optimization algorithms using SSA	451
19.4	Arrays, pointers, and memory	457
19.5	The control-dependence graph	459
19.6	Converting back from SSA form	462
19.7	A functional intermediate form	464
20	Pipelining and Scheduling	474
20.1	Loop scheduling without resource bounds	478
20.2	Resource-bounded loop pipelining	482
20.3	Branch prediction	490
21	The Memory Hierarchy	498
21.1	Cache organization	499
21.2	Cache-block alignment	502
21.3	Prefetching	504
21.4	Loop interchange	510
21.5	Blocking	511
21.6	Garbage collection and the memory hierarchy	514
	Appendix: Tiger Language Reference Manual	518
A.1	Lexical issues	518
A.2	Declarations	518
A.3	Variables and expressions	521
A.4	Standard library	525
A.5	Sample Tiger programs	526
	<i>Bibliography</i>	528
	<i>Index</i>	537

PART ONE

Fundamentals of Compilation

1

Introduction

A **compiler** was originally a program that “compiled” subroutines [a link-loader]. When in 1954 the combination “algebraic compiler” came into use, or rather into misuse, the meaning of the term had already shifted into the present one.

Bauer and Eickel [1975]

This book describes techniques, data structures, and algorithms for translating programming languages into executable code. A modern compiler is often organized into many phases, each operating on a different abstract “language.” The chapters of this book follow the organization of a compiler, each covering a successive phase.

To illustrate the issues in compiling real programming languages, I show how to compile Tiger, a simple but nontrivial language of the Algol family, with nested scope and heap-allocated records. Programming exercises in each chapter call for the implementation of the corresponding phase; a student who implements all the phases described in Part I of the book will have a working compiler. Tiger is easily modified to be *functional* or *object-oriented* (or both), and exercises in Part II show how to do this. Other chapters in Part II cover advanced techniques in program optimization. Appendix A describes the Tiger language.

The interfaces between modules of the compiler are almost as important as the algorithms inside the modules. To describe the interfaces concretely, it is useful to write them down in a real programming language. This book uses the C programming language.

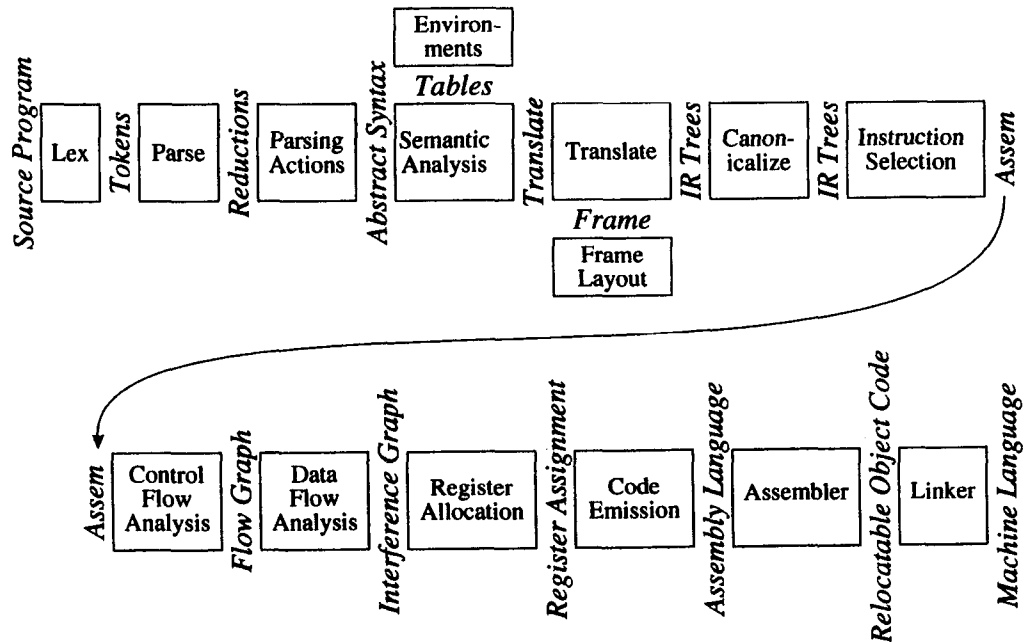


FIGURE 1.1. Phases of a compiler, and interfaces between them.

1.1

MODULES AND INTERFACES

Any large software system is much easier to understand and implement if the designer takes care with the fundamental abstractions and interfaces. Figure 1.1 shows the phases in a typical compiler. Each phase is implemented as one or more software modules.

Breaking the compiler into this many pieces allows for reuse of the components. For example, to change the target-machine for which the compiler produces machine language, it suffices to replace just the Frame Layout and Instruction Selection modules. To change the source language being compiled, only the modules up through Translate need to be changed. The compiler can be attached to a language-oriented syntax editor at the *Abstract Syntax* interface.

The learning experience of coming to the right abstraction by several iterations of *think–implement–redesign* is one that should not be missed. However, the student trying to finish a compiler project in one semester does not have

this luxury. Therefore, I present in this book the outline of a project where the abstractions and interfaces are carefully thought out, and are as elegant and general as I am able to make them.

Some of the interfaces, such as *Abstract Syntax*, *IR Trees*, and *Assem*, take the form of data structures: for example, the Parsing Actions phase builds an *Abstract Syntax* data structure and passes it to the Semantic Analysis phase. Other interfaces are abstract data types; the *Translate* interface is a set of functions that the Semantic Analysis phase can call, and the *Tokens* interface takes the form of a function that the Parser calls to get the next token of the input program.

DESCRIPTION OF THE PHASES

Each chapter of Part I of this book describes one compiler phase, as shown in Table 1.2

This modularization is typical of many real compilers. But some compilers combine Parse, Semantic Analysis, Translate, and Canonicalize into one phase; others put Instruction Selection much later than I have done, and combine it with Code Emission. Simple compilers omit the Control Flow Analysis, Data Flow Analysis, and Register Allocation phases.

I have designed the compiler in this book to be as simple as possible, but no simpler. In particular, in those places where corners are cut to simplify the implementation, the structure of the compiler allows for the addition of more optimization or fancier semantics without violence to the existing interfaces.

1.2

TOOLS AND SOFTWARE

Two of the most useful abstractions used in modern compilers are *context-free grammars*, for parsing, and *regular expressions*, for lexical analysis. To make best use of these abstractions it is helpful to have special tools, such as *Yacc* (which converts a grammar into a parsing program) and *Lex* (which converts a declarative specification into a lexical analysis program).

The programming projects in this book can be compiled using any ANSI-standard C compiler, along with *Lex* (or the more modern *Flex*) and *Yacc* (or the more modern *Bison*). Some of these tools are freely available on the Internet; for information see the World Wide Web page

<http://www.cs.princeton.edu/~appel/modern/c>

CHAPTER ONE. INTRODUCTION

Chapter	Phase	Description
2	Lex	Break the source file into individual words, or <i>tokens</i> .
3	Parse	Analyze the phrase structure of the program.
4	Semantic Actions	Build a piece of <i>abstract syntax tree</i> corresponding to each phrase.
5	Semantic Analysis	Determine what each phrase means, relate uses of variables to their definitions, check types of expressions, request translation of each phrase.
6	Frame Layout	Place variables, function-parameters, etc. into activation records (stack frames) in a machine-dependent way.
7	Translate	Produce <i>intermediate representation trees</i> (IR trees), a notation that is not tied to any particular source language or target-machine architecture.
8	Canonicalize	Hoist side effects out of expressions, and clean up conditional branches, for the convenience of the next phases.
9	Instruction Selection	Group the IR-tree nodes into clumps that correspond to the actions of target-machine instructions.
10	Control Flow Analysis	Analyze the sequence of instructions into a <i>control flow graph</i> that shows all the possible flows of control the program might follow when it executes.
10	Dataflow Analysis	Gather information about the flow of information through variables of the program; for example, <i>liveness analysis</i> calculates the places where each program variable holds a still-needed value (is <i>live</i>).
11	Register Allocation	Choose a register to hold each of the variables and temporary values used by the program; variables not live at the same time can share the same register.
12	Code Emission	Replace the temporary names in each machine instruction with machine registers.

TABLE 1.2. Description of compiler phases.

Source code for some modules of the Tiger compiler, skeleton source code and support code for some of the programming exercises, example Tiger programs, and other useful files are also available from the same Web address. The programming exercises in this book refer to this directory as \$TIGER/ when referring to specific subdirectories and files contained therein.

1.3. DATA STRUCTURES FOR TREE LANGUAGES

$Stm \rightarrow Stm ; Stm$	(CompoundStm)	$ExpList \rightarrow Exp , ExpList$	(PairExpList)
$Stm \rightarrow id := Exp$	(AssignStm)	$ExpList \rightarrow Exp$	(LastExpList)
$Stm \rightarrow print (ExpList)$	(PrintStm)	$Binop \rightarrow +$	(Plus)
$Exp \rightarrow id$	(IdExp)	$Binop \rightarrow -$	(Minus)
$Exp \rightarrow num$	(NumExp)	$Binop \rightarrow \times$	(Times)
$Exp \rightarrow Exp Binop Exp$	(OpExp)	$Binop \rightarrow /$	(Div)
$Exp \rightarrow (Stm , Exp)$	(EseqExp)		

GRAMMAR 1.3. A straight-line programming language.

1.3

DATA STRUCTURES FOR TREE LANGUAGES

Many of the important data structures used in a compiler are *intermediate representations* of the program being compiled. Often these representations take the form of trees, with several node types, each of which has different attributes. Such trees can occur at many of the phase-interfaces shown in Figure 1.1.

Tree representations can be described with grammars, just like programming languages. To introduce the concepts, I will show a simple programming language with statements and expressions, but no loops or if-statements (this is called a language of *straight-line programs*).

The syntax for this language is given in Grammar 1.3.

The informal semantics of the language is as follows. Each *Stm* is a statement, each *Exp* is an expression. $s_1 ; s_2$ executes statement s_1 , then statement s_2 . $i := e$ evaluates the expression e , then “stores” the result in variable i . $print(e_1, e_2, \dots, e_n)$ displays the values of all the expressions, evaluated left to right, separated by spaces, terminated by a newline.

An *identifier expression*, such as i , yields the current contents of the variable i . A *number* evaluates to the named integer. An *operator expression* $e_1 \text{ op } e_2$ evaluates e_1 , then e_2 , then applies the given binary operator. And an *expression sequence* (s, e) behaves like the C-language “comma” operator, evaluating the statement s for side effects before evaluating (and returning the result of) the expression e .