

Modern Compiler Implementation in C

现代编译原理

C语言描述（修订版）

[美] Andrew W. Appel Maia Ginsburg◎著
赵克佳 黄春 沈志宇◎译

- 经典编译原理教材，MIT、普林斯顿大学、剑桥大学等诸多名校采纳
- 与“龙书”齐名的“虎书”，中文版全新修订
- 巩固基础，注重实践，教你自己动手构造编译器



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

Modern Compiler Implementation in C

现代编译原理

C语言描述（修订版）

[美] Andrew W. Appel Maia Ginsburg◎著

赵克佳 黄春 沈志宇 ◎译

人民邮电出版社
北 京

图书在版编目(CIP)数据

现代编译原理：C语言描述：修订版 / (美) 安德鲁·W. 安佩尔 (Andrew W. Appel), (美) 马亚·金斯伯格 (Maia Ginsburg) 著；赵克佳, 黄春, 沈志宇译. — 2版. — 北京：人民邮电出版社, 2018. 4

(图灵计算机科学丛书)

ISBN 978-7-115-47688-3

I. ①现… II. ①安… ②马… ③赵… ④黄… ⑤沈… III. ①编译程序—程序设计②C语言—程序设计 IV. ①TP314

中国版本图书馆CIP数据核字(2017)第330597号

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。

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

-
- ◆ 著 [美] Andrew W. Appel Maia Ginsburg
译 赵克佳 黄 春 沈志宇
责任编辑 杨 琳
责任印制 周昇亮
- ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
三河市潮河印业有限公司印刷
- ◆ 开本: 787×1092 1/16
印张: 25
字数: 665千字 2018年4月第2版
印数: 7 501—11 000册 2018年4月河北第1次印刷
著作权合同登记号 图字: 01-2017-6266号
-

定价: 89.00元

读者服务热线: (010)51095186 转 600 印装质量热线: (010)81055316

反盗版热线: (010)81055315

广告经营许可证: 京东工商广登字 20170147号

版 权 声 明

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

All rights reserved.

This Simplified Chinese edition for the People's Republic of China is published by arrangement with the Press Syndicate of the University of Cambridge, Cambridge, United Kingdom.

© Cambridge University Press & Posts & Telecom Press 2017.

This book is in copyright. No reproduction of any part may take place without the written permission of Cambridge University Press and Posts & Telecom Press.

This edition is for sale in the People's Republic of China (excluding Hong Kong SAR, Macao SAR and Taiwan Province) only.

本书原版由剑桥大学出版社出版。

本书中文翻译版由剑桥大学出版社授权人民邮电出版社出版。此版本仅限在中华人民共和国境内（不包含香港、澳门特别行政区及台湾地区）销售。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

译者序

本书全面讲述了现代编译器的结构、编译算法和实现方法，是 Andrew W. Apple 的“虎书”——*Modern Compiler Implementation*——“红、蓝、绿”三序列之一。这三本书的内容基本相同，但是使用不同的语言来实现书中给出的一个编译器。本书使用的是更适合广大读者的 C 语言，而另外两本书分别采用 ML 和 Java 语言。

国外关于编译技术有三本比较著名的书，分别被誉为“龙书”“鲸书”和“虎书”。“虎书”即本书，已经被国外许多著名大学选作编译原理课程的教材。编译器的设计与实现是一种实践性很强的工程。作为讲述编译器实现方法的编译原理课程，既需要讲述理论和原理，也离不开具体的实践。本书的章节按照编译器处理过程的各个阶段依次组织，并精心设计了一个“学生项目编译器”的框架和模块接口。每一章结尾给出了与该编译器一个模块对应的编译器项目实现的习题，使得学生在掌握了编译原理和方法的同时，能够理论联系实际地亲自动手体验具体的实现过程，并逐步实现一个编译器。它弥补了目前一些编译原理教科书在实践方面的不足。这是本书的特点之一。

本书的另一个特点是增加了一些其他编译原理教科书没有涉及的内容。前端增加了面向对象的程序设计语言、函数式程序设计语言等现代语言的编译实现方法，后端增加了针对现代计算机体系结构特征的一些比较成熟的优化方法。这部分内容展现了现代商业编译器需解决的一些关键问题，开拓了学生的视野，为学生未来进行更深入的研究奠定了基础。

在翻译过程中，我们力图忠实于原文，使译文通顺流畅，并保持专用术语的译法与惯用的一致。对于那些没有明确译法的术语，则根据原义拟定，并给出了英文以利读者对照。

本书第 1~4 章由沈志宇翻译，第 5~14 章以及前言和附录由赵克佳翻译，第 15~21 章由黄春翻译。全书经过三人的反复校对，最后由赵克佳定稿。在本书的翻译中，我们深感阅读英文专业书籍和较好地翻译出来是两回事。翻译好一本专业类图书除了要有相应的英文和专业功底外，还要有比较好的中文功底。我们自感在这三方面都有所欠缺，难免会留有遗憾。衷心欢迎读者批评指正，因为读者的批评指正对我们是极好的帮助。

前言

近十余年来，编译器的构建方法出现了一些新的变化。一些新的程序设计语言得到应用，例如，具有动态方法的面向对象语言、具有嵌套作用域和一等函数闭包（first-class function closure）的函数式语言等。这些语言中有许多都需要垃圾收集技术的支持。另一方面，新的计算机都有较大的寄存器集合，且存储器访问成为了影响性能的主要因素。这类机器在具有指令调度能力并能对指令和数据高速缓存（cache）进行局部性优化的编译器辅助下，常常能运行得更快。

本书可作为一到两个学期编译课程的教材。学生将看到编译器不同部分中隐含的理论，学习到将这些理论付诸实现时使用的程序设计技术和以模块化方式实现该编译器时使用的接口。为了清晰具体地给出这些接口和程序设计的例子，我使用 C 语言来编写它们。本（序列）书还有使用 ML 和 Java 语言的另外两个版本。

实现项目。我在书中概述了一个“学生项目编译器”，它相当简单，而且其安排方式也便于说明现在常用的一些重要技术。这些技术包括避免语法和语义相互纠缠的抽象语法树，独立于寄存器分配的指令选择，能使编译器前期阶段有更多灵活性的复写传播，以及防止从属于特定目标机的方法。与其他许多教材中的“学生编译器”不同，本书中采用的编译器有一个简单而完整的后端，它允许在指令选择之后进行寄存器分配。

本书第一部分中，每一章都有一个与编译器的某个模块对应的程序设计习题。在 <http://www.cs.princeton.edu/~appel/modern/c> 中可找到对这些习题有帮助的一些软件。

习题。每一章都有一些书面习题；标有一个星号的习题有点挑战性，标有两个星号的习题较难但仍可解决，偶尔出现的标有三个星号的习题是一些尚未找到解决方法的问题。

授课顺序。图 0-1 展示了各章相互之间的依赖关系。

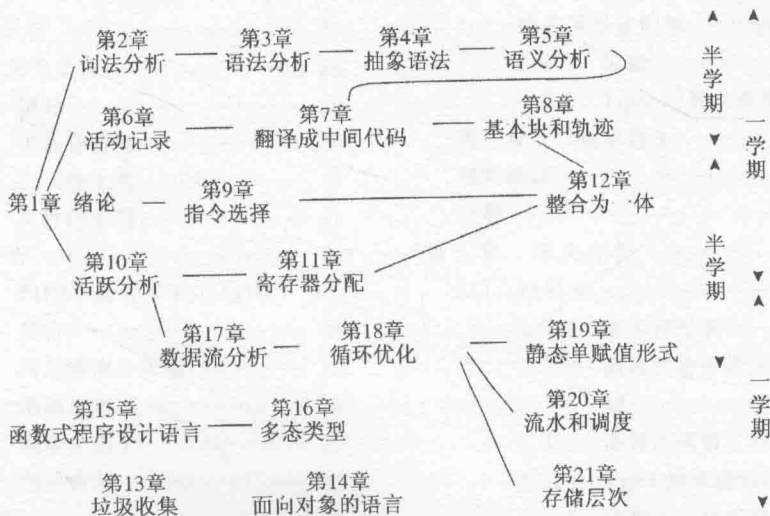


图 0-1 内容结构图

- 一学期的课程可包含第一部分的所有章节（第 1~12 章），同时让学生实现项目编译器（多半按项目组的方式进行）。另外，授课内容中还可以包含从第二部分中选择的一些主题。
- 高级课程或研究生课程可包含第二部分的内容，以及另外一些来自其他文献的主题。第二部分中有许多章节与第一部分无关，因此，对于那些在初始课程中使用不同教材的学生而言，仍然可以给他们讲授高级课程。
- 若按两个半个学期来安排教学，则前半学期可包含第 1~8 章，后半学期包括第 9~12 章和第二部分的某些章。

致谢。对于本书，许多人给我提出了富有建设性的意见，或在其他方面给我提供了帮助。我要感谢这些人，他们是 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 和 Kenneth Zadeck。

目 录

第一部分 编译基本原理

第 1 章 绪论	1
1.1 模块与接口	1
1.2 工具和软件	3
1.3 树语言的数据结构	3
程序设计: 直线式程序解释器	7
推荐阅读	8
习题	9
第 2 章 词法分析	10
2.1 词法单词	10
2.2 正则表达式	11
2.3 有限自动机	13
2.4 非确定有限自动机	15
2.4.1 将正则表达式转换为 NFA	16
2.4.2 将 NFA 转换为 DFA	18
2.5 Lex: 词法分析器的生成器	20
程序设计: 词法分析	22
推荐阅读	23
习题	23
第 3 章 语法分析	27
3.1 上下文无关文法	28
3.1.1 推导	29
3.1.2 语法分析树	29
3.1.3 二义性文法	30
3.1.4 文件结束符	31
3.2 预测分析	32
3.2.1 FIRST 集合和 FOLLOW 集合	33
3.2.2 构造预测分析器	35
3.2.3 消除左递归	36
3.2.4 提取左因子	37
3.2.5 错误恢复	37
3.3 LR 分析	39
3.3.1 LR 分析引擎	40
3.3.2 LR(0) 分析器生成器	41

3.3.3 SLR 分析器的生成	43
3.3.4 LR(1) 项和 LR(1) 分析 表	45
3.3.5 LALR(1) 分析表	46
3.3.6 各类文法的层次	47
3.3.7 二义性文法的 LR 分析	47
3.4 使用分析器的生成器	48
3.4.1 冲突	49
3.4.2 优先级指导	50
3.4.3 语法和语义	53
3.5 错误恢复	54
3.5.1 用 error 符号恢复	54
3.5.2 全局错误修复	55
程序设计: 语法分析	57
推荐阅读	58
习题	58
第 4 章 抽象语法	62
4.1 语义动作	62
4.1.1 递归下降	62
4.1.2 Yacc 生成的分析器	62
4.1.3 语义动作的解释器	64
4.2 抽象语法分析树	65
4.2.1 位置	67
4.2.2 Tiger 的抽象语法	68
程序设计: 抽象语法	71
推荐阅读	71
习题	72
第 5 章 语义分析	73
5.1 符号表	73
5.1.1 多个符号表	74
5.1.2 高效的命令式风格符号 表	75
5.1.3 高效的函数式符号表	76
5.1.4 Tiger 编译器的符号	77
5.1.5 函数式风格的符号表	79
5.2 Tiger 编译器的绑定	79
5.3 表达式的类型检查	82

5.4 声明的类型检查	84	7.3 声明	120
5.4.1 变量声明	84	7.3.1 变量定义	120
5.4.2 类型声明	85	7.3.2 函数定义	120
5.4.3 函数声明	85	7.3.3 片段	121
5.4.4 递归声明	86	程序设计：翻译成树	122
程序设计：类型检查	86	习题	123
习题	87	第8章 基本块和轨迹	125
第6章 活动记录	89	8.1 规范树	126
6.1 栈帧	90	8.1.1 ESEQ 的转换	126
6.1.1 帧指针	91	8.1.2 一般重写规则	126
6.1.2 寄存器	92	8.1.3 将 CALL 移到顶层	130
6.1.3 参数传递	92	8.1.4 线性语句表	131
6.1.4 返回地址	94	8.2 处理条件分支	131
6.1.5 栈帧内的变量	94	8.2.1 基本块	131
6.1.6 静态链	95	8.2.2 轨迹	132
6.2 Tiger 编译器的栈帧	96	8.2.3 完善	133
6.2.1 栈帧描述的表示	98	8.2.4 最优轨迹	133
6.2.2 局部变量	98	推荐阅读	134
6.2.3 计算逃逸变量	99	习题	134
6.2.4 临时变量和标号	100	第9章 指令选择	136
6.2.5 两层抽象	100	9.1 指令选择算法	138
6.2.6 管理静态链	102	9.1.1 Maximal Munch 算法	138
6.2.7 追踪层次信息	102	9.1.2 动态规划	140
程序设计：栈帧	102	9.1.3 树文法	141
推荐阅读	103	9.1.4 快速匹配	143
习题	103	9.1.5 覆盖算法的效率	143
第7章 翻译成中间代码	106	9.2 CISC 机器	144
7.1 中间表示树	106	9.3 Tiger 编译器的指令选择	146
7.2 翻译为树中间语言	108	9.3.1 抽象的汇编语言指令	146
7.2.1 表达式的种类	108	9.3.2 生成汇编指令	148
7.2.2 简单变量	111	9.3.3 过程调用	151
7.2.3 追随静态链	112	9.3.4 无帧指针的情形	151
7.2.4 数组变量	113	程序设计：指令选择	152
7.2.5 结构化的左值	113	推荐阅读	153
7.2.6 下标和域选择	114	习题	154
7.2.7 关于安全性的劝告	115	第10章 活跃分析	155
7.2.8 算术操作	116	10.1 数据流方程的解	156
7.2.9 条件表达式	116	10.1.1 活跃性计算	156
7.2.10 字符串	117	10.1.2 集合的表示	158
7.2.11 记录和数组的创建	118	10.1.3 时间复杂度	158
7.2.12 while 循环	119	10.1.4 最小不动点	159
7.2.13 for 循环	119	10.1.5 静态活跃性与动态活 跃性	160
7.2.14 函数调用	120		

10.1.6 冲突图	161
10.2 Tiger 编译器的活跃分析	162
10.2.1 图	162
10.2.2 控制流图	163
10.2.3 活跃分析	164
程序设计: 构造流图	164
程序设计: 活跃分析模块	165
习题	165
第 11 章 寄存器分配	166
11.1 通过简化进行着色	166
11.2 合并	168
11.3 预着色的结点	171
11.3.1 机器寄存器的临时副 本	171
11.3.2 调用者保护的寄存器 和被调用者保护的寄 存器	172
11.3.3 含预着色结点的例子 ..	172
11.4 图着色的实现	175
11.4.1 传送指令工作表的 管理	176
11.4.2 数据结构	176
11.4.3 程序代码	177
11.5 针对树的寄存器分配	181
程序设计: 图着色	184
推荐阅读	185
习题	185
第 12 章 整合为一体	188
程序设计: 过程入口/出口	189
程序设计: 创建一个可运行的编译器 ..	191

第二部分 高级主题

第 13 章 垃圾收集	193
13.1 标记-清扫式收集	194
13.2 引用计数	197
13.3 复制式收集	198
13.4 分代收集	201
13.5 增量式收集	203
13.6 Baker 算法	205
13.7 编译器接口	205
13.7.1 快速分配	205
13.7.2 数据布局的描述	206

13.7.3 导出指针	207
程序设计: 描述字	208
程序设计: 垃圾收集	208
推荐阅读	208
习题	210
第 14 章 面向对象的语言	211
14.1 类	211
14.2 数据域的单继承性	213
14.3 多继承	214
14.4 测试类成员关系	216
14.5 私有域和私有方法	218
14.6 无类语言	219
14.7 面向对象程序的优化	219
程序设计: Object-Tiger	220
推荐阅读	220
习题	221
第 15 章 函数式程序设计语言	222
15.1 一个简单的函数式语言	222
15.2 闭包	224
15.3 不变的变量	225
15.3.1 基于延续的 I/O	226
15.3.2 语言上的变化	227
15.3.3 纯函数式语言的优 化	228
15.4 内联扩展	229
15.5 闭包变换	233
15.6 高效的尾递归	235
15.7 懒惰计算	236
15.7.1 传名调用计算	237
15.7.2 按需调用	238
15.7.3 懒惰程序的计算	239
15.7.4 懒惰函数式程序的 优化	239
15.7.5 严格性分析	241
推荐阅读	243
程序设计: 编译函数式语言	244
习题	244
第 16 章 多态类型	246
16.1 参数多态性	246
16.1.1 显式带类型的多态 语言	247
16.1.2 多态类型的检查	248
16.2 类型推论	253

16.2.1 一个隐式类型的多态语言	254	习题	285
16.2.2 类型推论算法	255	第 18 章 循环优化	287
16.2.3 递归的数据类型	257	18.1 必经结点	289
16.2.4 Hindley-Milner 类型的能力	259	18.1.1 寻找必经结点的算法	289
16.3 多态变量的表示	259	18.1.2 直接必经结点	289
16.3.1 多态函数的扩展	260	18.1.3 循环	290
16.3.2 完全的装箱转换	261	18.1.4 循环前置结点	291
16.3.3 基于强制的表示分析	262	18.2 循环不变量计算	292
16.3.4 将类型作为运行时参数传递	264	18.3 归纳变量	293
16.4 静态重载的解决方法	265	18.3.1 发现归纳变量	294
推荐阅读	266	18.3.2 强度削弱	295
习题	266	18.3.3 删除	296
第 17 章 数据流分析	269	18.3.4 重写比较	296
17.1 流分析使用的中间表示	270	18.4 数组边界检查	297
17.2 各种数据流分析	271	18.5 循环展开	300
17.2.1 到达定值	271	推荐阅读	301
17.2.2 可用表达式	273	习题	301
17.2.3 到达表达式	274	第 19 章 静态单赋值形式	303
17.2.4 活跃分析	274	19.1 转化为 SSA 形式	305
17.3 使用数据流分析结果的几种转换	274	19.1.1 插入 ϕ 函数的标准	306
17.3.1 公共子表达式删除	274	19.1.2 必经结点边界	306
17.3.2 常数传播	275	19.1.3 插入 ϕ 函数	308
17.3.3 复写传播	275	19.1.4 变量重命名	309
17.3.4 死代码删除	275	19.1.5 边分割	310
17.4 加快数据流分析	276	19.2 必经结点树的高效计算	310
17.4.1 位向量	276	19.2.1 深度优先生成树	310
17.4.2 基本块	276	19.2.2 半必经结点	311
17.4.3 结点排序	277	19.2.3 Lengauer-Tarjan 算法	312
17.4.4 使用一定值链和定值-使用链	277	19.3 使用 SSA 的优化算法	315
17.4.5 工作表算法	278	19.3.1 死代码删除	315
17.4.6 增量式数据流分析	278	19.3.2 简单的常数传播	316
17.5 别名分析	281	19.3.3 条件常数传播	317
17.5.1 基于类型的别名分析	282	19.3.4 保持必经结点性质	319
17.5.2 基于流的别名分析	283	19.4 数组、指针和存储器	320
17.5.3 使用可能别名信息	284	19.5 控制依赖图	321
17.5.4 严格的纯函数式语言中的别名分析	285	19.6 从 SSA 形式转变回来	323
推荐阅读	285	19.7 函数式中间形式	325
		推荐阅读	327
		习题	328
		第 20 章 流水和调度	331
		20.1 没有资源约束时的循环调度	332
		20.2 有资源约束的循环流水	336
		20.2.1 模调度	337

20.2.2 寻找最小的启动间距	338	21.2 cache 块对齐	349
20.2.3 其他控制流	340	21.3 预取	350
20.2.4 编译器应该调度指令吗 ..	340	21.4 循环交换	354
20.3 分支预测	341	21.5 分块	355
20.3.1 静态分支预测	342	21.6 垃圾收集和存储层次	357
20.3.2 编译器应该预测分支吗 ..	342	推荐阅读	358
推荐阅读	343	习题	358
习题	343	附录 Tiger 语言参考手册	360
第 21 章 存储层次	346	参考文献	368
21.1 cache 的组织结构	346	索引	376

第一部分 编译基本原理

第 1 章 绪 论

编译器 (compiler)：原指一种将各个子程序装配组合到一起的程序 [连接-装配器]。当 1954 年出现了 (确切地说是误用了) 复合术语“代数编译器” (algebraic compiler) 之后，这个术语的意思变成了现在的含义。

Bauer 和 Eickel [1975]

本书讲述将程序设计语言转换成可执行代码时使用的技术、数据结构和算法。现代编译器常常由多个阶段组成，每一阶段处理不同的抽象“语言”。本书的章节按照编译器的结构来组织，每一章循序渐进地论及编译器的一个阶段。

为了阐明编译真实的程序设计语言时遇到的问题，本书以 Tiger 语言为例来说明如何编译一种语言。Tiger 语言是一种类 Algol 的语言，它有嵌套的作用域和在堆中分配存储空间的记录，虽简单却并不平凡。每一章的程序设计练习都要求实现相应的编译阶段；如果学生实现了本书第一部分讲述的所有阶段，便能够得到一个可以运行的编译器。将 Tiger 修改成函数式的或面向对象的 (或同时满足两者的) 语言并不难，第二部分中的习题说明了如何进行这种修改。第二部分的其他章节讨论了有关程序优化的高级技术。附录描述了 Tiger 语言。

编译器各模块之间的接口几乎和模块内部的算法同等重要。为了具体描述这些接口，较好的做法是用真正的程序设计语言来编写它们，本书使用的是 C 语言。

3

1.1 模块与接口

对于任何大型软件系统，如果设计者注意到了该系统的基本抽象和接口，那么对这个系统的理解和实现就要容易得多。图 1-1 展示了一个典型的编译器的各个阶段，每个阶段由一至多个软件模块来实现。

将编译器分解成这样的多个阶段是为了能够重用它的各种构件。例如，当要改变此编译器所生成的机器语言的目标机时，只要改变栈帧布局 (Frame Layout) 模块和指令选择 (Instruction Selection) 模块就够了。当要改变被编译的源语言时，则至多只需改变翻译 (Translate) 模块之前的模块就可以了，该编译器也可以在抽象语法 (Abstract Syntax) 接口处与面向语言的语法编辑器相连。

每个学生都不应缺少反复多次“思考-实现-重新设计”，从而获得正确的抽象这样一种学习经历。但是，想要学生在一个学期内实现一个编译器是不现实的。因此，我在书中给出了一个项目框架，其中的模块和接口都经过深思熟虑，而且尽可能地使之既精巧又通用。

4

抽象语法 (Abstract Syntax)、IR 树 (IR Tree) 和汇编 (Assem) 之类的接口是数据结构的形式，例如语法分析动作阶段建立抽象语法数据结构，并将它传递给语义分析阶段。另一些接

口是抽象数据类型：翻译接口是一组可由语义分析阶段调用的函数；单词符号（Token）接口是函数形式，分析器通过调用它而得到输入程序中的下一个单词符号。

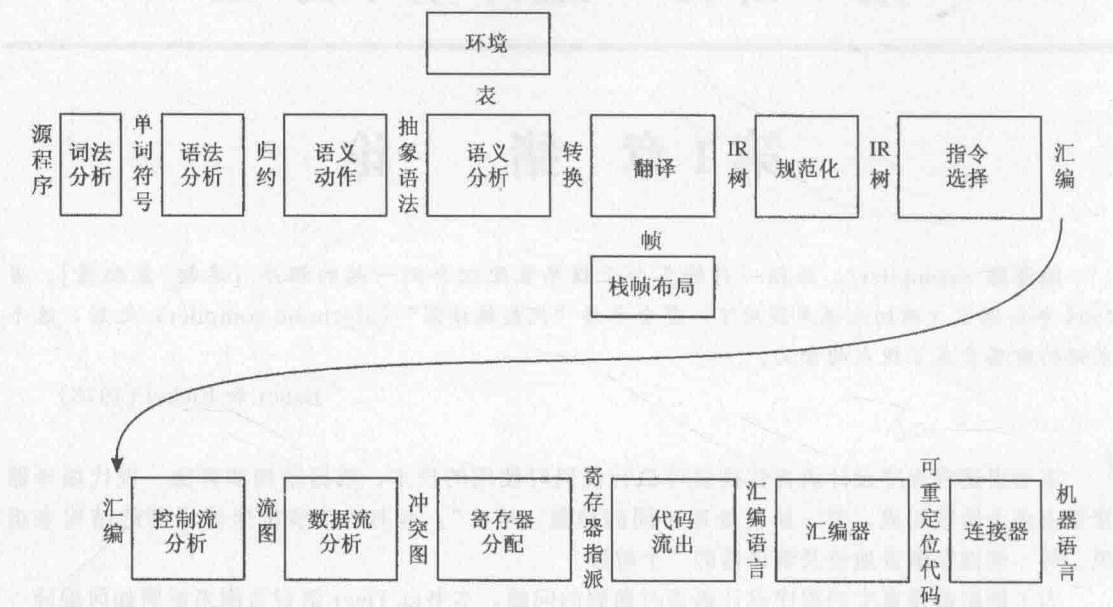


图 1-1 编译器的各个阶段及其之间的接口

各个阶段的描述

第一部分的每一章各描述编译器的一个阶段，具体如表 1-1 所示。

表 1-1 编译器的各阶段

章号	阶段	描 述
2	词法分析	将源文件分解为一个个独立的单词符号
3	语法分析	分析程序的短语结构
4	语义动作	建立每个短语对应的抽象语法树
5	语义分析	确定每个短语的含义，建立变量和其声明的关联，检查表达式的类型，翻译每个短语
6	栈帧布局	按机器要求的方式将变量、函数参数等分配于活跃记录（即栈帧）内
7	翻译	生成中间表示树（IR 树），这是一种与任何特定程序设计语言和目标机体系结构无关的表示
8	规范化	提取表达式中的副作用，整理条件分支，以方便下一阶段的处理
9	指令选择	将 IR 树结点组合成与目标机指令的动作相对应的块
10	控制流分析	分析指令的顺序并建立控制流图，此图表示程序执行时可能流经的所有控制流
10	数据流分析	收集程序变量的数据流信息。例如，活跃分析（liveness analysis）计算每一个变量仍需使用其值的地点（即它的活跃点）
11	寄存器分配	为程序中的每一个变量和临时数据选择一个寄存器，不在同一时间活跃的两个变量可以共享同一个寄存器
12	代码流出	用机器寄存器替代每一条机器指令中出现的临时变量名

这种模块化设计是很多真实编译器的典型设计。但是,也有一些编译器把语法分析、语义分析、翻译和规范化合并成一个阶段,还有一些编译器将指令选择安排在更后一些的位置,并且将它与代码流出合并在一起。简单的编译器通常没有专门的控制流分析、数据流分析和寄存器分配阶段。

我在设计本书的编译器时尽可能地进行了简化,但并不意味着它是一个简单的编译器。具体而言,虽然为简化设计而去掉了一些细枝末节,但该编译器的结构仍然可以允许增加更多的优化或语义而不会违背现存的接口。

1.2 工具和软件

现代编译器中使用的两种最有用的抽象是上下文无关文法 (context-free grammar) 和正则表达式 (regular expression)。上下文无关文法用于语法分析,正则表达式用于词法分析。为了更好地利用这两种抽象,较好的做法是借助一些专门的工具,例如 Yacc (它将文法转换成语法分析器) 和 Lex (它将一个说明性的规范转换成一个词法分析器)。

本书的程序设计项目可借助 Lex (或更为现代的 Flex) 和 Yacc (或更为现代的 Bison), 用任何 ANSI 标准 C 编译器来编译。这些工具中有些可免费从因特网上得到,更多的信息可参阅网页: <http://www.cs.princeton.edu/~appel/modern/c/>。

Tiger 编译器中某些模块的源代码、某些程序设计习题的框架源代码和支持代码、Tiger 程序的例子以及其他一些有用的文件都可以从该网址中找到。本书的程序设计习题中,当提及特定子目录或文件所在的某个目录时,指的是目录 \$TIGER/。

1.3 树语言的数据结构

编译器中使用的许多重要数据结构都是被编译程序的中间表示。这些表示常常采用树的形式,树的结点有若干种类型,每一种类型都有一些不同的属性。这种树可以作为图 1-1 所示的许多阶段的接口。

树表示可以用文法来描述,就像程序设计语言一样。为了介绍有关概念,我将给出一种简单的程序设计语言,该语言有语句和表达式,但是没有循环或 if 语句 [这种语言称为直线式程序 (straight-line program) 语言]。

该语言的语法在文法 1-1 中给出。

文法 1-1 直线式程序设计语言

$Stm \rightarrow Stm ; Stm$	(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)		

这个语言的非形式语义如下。每一个 Stm 是一个语句,每一个 Exp 是一个表达式。 $s_1 ; s_2$ 表示先执行语句 s_1 ,再执行语句 s_2 。 $i := e$ 表示先计算表达式 e 的值,然后把计算结果赋给变量 i 。

$\text{print}(e_1, e_2, \dots, e_n)$ 表示从左到右输出所有表达式的值，这些值之间用空格分开并以换行符结束。

标识符表达式，例如 i ，表示变量 i 的当前内容。数按命名它的整数计值。操作符表达式 $e_1 \text{ op } e_2$ 表示先计算 e_1 再计算 e_2 ，然后按给定的二元操作符计算表达式结果。表达式序列 (s, e) 的行为类似于 C 语言中的逗号操作符，在计算表达式 e （并返回其结果）之前先计算语句 s 的副作用。

例如，执行下面这段程序：

```
a := 5+3; b := (print(a, a-1), 10*a); print(b)
```

将打印出：

```
8 7
80
```

那么，这段程序在编译器内部是如何表示的呢？一种表示是源代码形式，即程序员所编写的字符，但这种表示不易处理。较为方便的表示是树数据结构，每一条语句（ Stm ）和每一个表达式（ Exp ）都有一个树结点。图 1-2 给出了这个程序的树表示，其中结点都用文法 1-1 中产生式的标识加以标记，并且每个结点的子结点数量与相应文法产生式右边的符号个数相同。

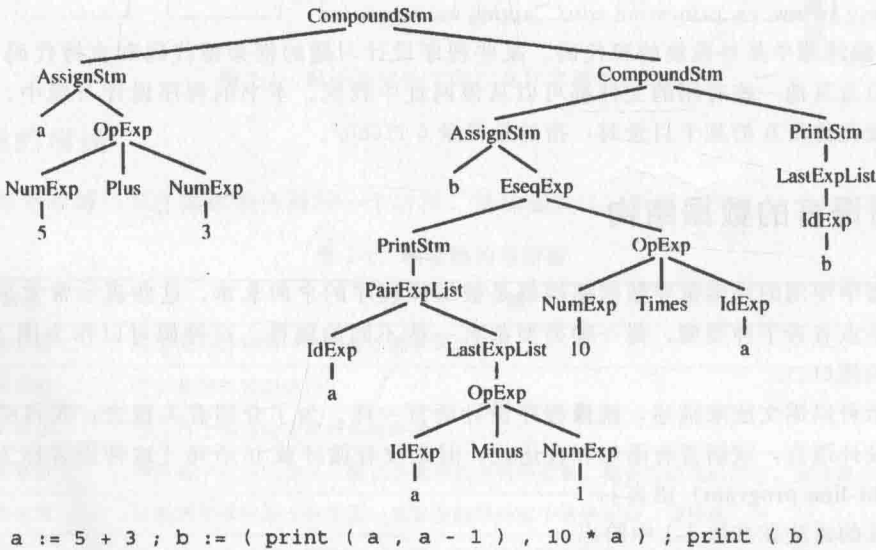


图 1-2 直线式程序的树表示

我们可以将这个文法直接翻译成数据结构定义，如程序 1-1 所示。每个文法符号对应于这些数据结构中的一个 typedef：

文法	typedef
Stm	A_stm
Exp	A_exp
$ExpList$	$A_expList$
id	string
num	int

程序 1-1 直线式程序的表示

```

typedef char *string;
typedef struct A_stm_ *A_stm;
typedef struct A_exp_ *A_exp;
typedef struct A_expList_ *A_expList;
typedef enum {A_plus,A_minus,A_times,A_div} A_binop;

struct A_stm_ {enum {A_compoundStm, A_assignStm, A_printStm} kind;
               union {struct {A_stm stm1, stm2;} compound;
                     struct {string id; A_exp exp;} assign;
                     struct {A_expList exps;} print;
               } u;
};

A_stm A_CompoundStm(A_stm stm1, A_stm stm2);
A_stm A_AssignStm(string id, A_exp exp);
A_stm A_PrintStm(A_expList exps);

struct A_exp_ {enum {A_idExp, A_numExp, A_opExp, A_eseqExp} kind;
               union {string id;
                     int num;
                     struct {A_exp left; A_binop oper; A_exp right;} op;
                     struct {A_stm stm; A_exp exp;} eseq;
               } u;
};

A_exp A_IdExp(string id);
A_exp A_NumExp(int num);
A_exp A_OpExp(A_exp left, A_binop oper, A_exp right);
A_exp A_EseqExp(A_stm stm, A_exp exp);

struct A_expList_ {enum {A_pairExpList, A_lastExpList} kind;
                  union {struct {A_exp head; A_expList tail;} pair;
                        A_exp last;
                  } u;
};

```

每一项文法规则都有一个构造器 (constructor), 隶属于规则左部符号的联合 (union)。这些构造器的名字在文法 1-1 各项右部的括号内。

每一项文法规则有若干右部成分, 这些成分都必须用数据结构来表示。例如, CompoundStm 的右部有两个 Stm; AssignStm 有一个标识符和一个表达式, 等等。表示每一个文法符号的结构 (struct) 都含有一个联合 (union) 和一个 kind 域, 前者用于存放可选的成分值, 后者用于指明选用这个联合中的哪一个成员。

对于每一种选择 (CompoundStm、AssignStm 等), 我们创建一个构造函数 (constructor function), 它用 malloc 为此数据结构分配空间并对其进行初始化。程序 1-1 只给出了这些函数的原型; A_CompoundStm 可这样定义:

```

A_stm A_CompoundStm(A_stm stm1, A_stm stm2) {
    A_stm s = checked_malloc(sizeof(*s));
    s->kind = A_compoundStm;
    s->u.compound.stm1=stm1; s->u.compound.stm2=stm2;
    return s;
}

```

二元操作符 (Binop) 的情形要简单些。尽管我们也可以为 Binop 创建一个结构 (此结构中的联合的成员分别表示 Plus、Minus、Times、Div), 但这样做是多余的, 因为这些成员并不存放数