内容简介

本书介绍编译器构造的一般原理、基本设计方法和主要实现技术，内容包括词法分析、语法分析、语义分析、中间代码生成、目标代码生成、独立于机器的代码优化和依赖于机器的代码优化等。本书除了介绍命令式编程语言的编译技术，还介绍面向对象语言和函数式语言的实现技术。另外，本书还强调一些相关的理论知识，如形式语言和自动机理论、语法制导的定义和属性文法、类型论和类型系统等。

本书内容丰富、讲解深入，注意理论联系实际，可作为高等学校计算机科学及相关专业的教材，也可作为计算机软件工程技术人员的参考用书。

策划编辑 张 曦 责任编辑 张 曦 封面设计 李卫青 版式设计 童 丹
责任绘图 邓 超 责任校对 责任印制

出版发行 高等教育出版社 网 址 http://www.hep.edu.cn
社 址 北京市西城区德外大街4号 http://www.hep.com.cn
邮政编码 100120 网上订购 http://www.hepmall.com.cn
印 刷 http://www.hepmall.com
开 本 mm× mm 1/16 http://www.hepmall.cn
印 张
字 数 千字 版 次 年 月第1版
购书热线 010-58581118 印 次 年 月第 次印刷
咨询电话 400-810-0598 定 价 元

本书如有缺页、倒页、脱页等质量问题，请到所购图书销售部门联系调换
版权所有 侵权必究
物 料 号 -

WBS要素：18-00807-003（正式出版前务必删除此项）

第 4 版前言

本教材自首次出版以来，承蒙广大读者、教师和同行专家的厚爱，曾连续三次（“十五”“十一五”和“十二五”）入选普通高等教育本科国家级规划教材，并于 2009 年获评普通高等教育精品教材。这份殊荣既是肯定，更是对持续前行的鞭策。当前，随着计算技术的快速演进，特别是国产芯片架构的崛起、新兴计算范式的发展以及现代编译基础设施的普及，编译技术作为连接软硬件的核心纽带，其教学内容与方法亟待更新。本次修订旨在传承经典编译理论的同时，紧密衔接当前技术生态与实践需求，致力于培养适应产业发展的高层次创新人才。

修订之旅，承载着厚重的使命与深切的缅怀。本书作为 2020 年底立项的安徽省省级一流教材建设项目，本应早日面世，却因本人日常教学与科研工作的繁重而一再延迟，直至 2025 年 1 月方得以完稿。尤为令人痛心与遗憾的是，我的恩师、挚友，也是本教材前三版的第一作者——陈意云教授，于 2022 年 3 月 5 日溘然长逝，未能见证此版的诞生。本次改版由我独立完成。陈意云教授作为本教材前几版的主要作者，其学术思想与教学体系仍在本版中得以延续与体现。谨以此书纪念他的贡献。

为切实体现“厚基强践、面向生态、引领前沿”的修订宗旨，本次改版在传承经典核心理论的前提下，进行了体系化重构与实质性扩充：

第一，拓宽学科视野，融入技术生态。 新增“编程语言及其演变”专题，通过对比不同范式语言解决同一问题的视角，剖析语言设计与编译的关联；系统梳理以 GCC、LLVM 为代表的**通用编译基础设施**，并首次将**国内编译技术生态**纳入教材视野；同时概览面向**智能计算**、**图计算**、**量子计算等新兴领域**的编程系统，帮助学生建立从理论到产业的立体认知。

第二，强化工程实践，剖析现代架构。 编译前端重点引入工业级工具 ANTLR，详解其解析算法及应用方法；编译中后端采用“基于树访问”和访问者模式为核心的现代设计范式，替代传统边解析边翻译；同时，新增对 LLVM 编译系统及其 LLVM IR 的深入剖析，涵盖其模块化设计、中间表示、遍管理器与优化框架，并阐述线性扫描、图着色等寄存器分配算法在 LLVM 中的实现机制，实现经典理论、算法设计与当代主流编译基础设施的有机贯通。

第三，优化教学适配，完善实践体系。 为适应不同教学安排，建议 40 学时课程讲授第 1 至 9 章（不含加星节），60 学时课程可增加第 11 章 11.1 与 11.2 节，其余章节供学有余力的

编译原理（第4版） 正文 一校一次 文前+编译原理(第4版) 一校.pdf 第3页 文件编号: 422623 页面编号: 134538412 打印人: 8685 2026-03-09 10:22:34

读者深入研习。配套课件、试题等资源可通过作者教学网页获取。我们已构建起覆盖编译全流程的实验体系，形成“编译原理应用与实践”智慧课程，并部署于头歌平台和中国科学技术大学计算机实践教学综合服务平台。读者亦可选用我们早期编写的《编译原理实验教程》中的大型综合性项目开展教学与实践。

本次修订得到中国科学技术大学及多位专家的支持，高等教育出版社张曦编辑为书稿提供了细致专业的审校与匡正，在此一并致以诚挚感谢。书中虽经反复推敲，但仍难免存在疏漏与不足，恳请广大师生和读者通过邮件（yuzhang@ustc.edu.cn）批评指正，以便后续完善。

谨以此书，献给所有致力于中国编译技术教学、研究与产业实践的同行者。

张昱

于中国科学技术大学

2025 年 12 月

第 3 版前言

本书连续三次（“十五”“十一五”和“十二五”）入选普通高等教育本科国家级规划教材，并于 2009 年被评为普通高等教育精品教材，感谢广大读者、教师和评审专家对本教材的肯定和期待。本次改版受 2012 年底第三次入选规划教材的推动。

编译器技术近年来的主要进展在面向新型计算机体系结构的优化方面，如并行化和低功耗。本教材第 2 版已增加一章专门介绍依赖于机器的代码优化，再增加这方面内容会使得教材各部分的比例失调。另外，对高可信软件的需求也在推动编译器技术的研究，如出具证明的编译器（certifying compiler）和经过验证的编译器（verified compiler），但相关成果尚未成熟到值得写入教材。

基于上述考虑，本次改版没有增删内容，但添加了 30 多道习题。这些习题大部分都从学习编程语言和编译技术时碰到的实际问题中抽象出来，对把握编程语言和理解编译技术很有帮助。此外，本次改版还对各章内容进行了局部修订。配合本次改版，作者也改版了与本教材配套的《编译原理习题精选与解析》（见文献 [8]）。

对于 40 学时的课程，作者建议讲授第 1 ～ 9 章（不包括加星号的节），也可以直接采用作者专为普通高等学校学生编写的《编译原理与技术》（见文献 [9]），它就是按这个讲授建议形成的精简教材。对于 60 学时的课程，作者建议讲授第 1 ～ 9 章、第 11 章（不含 11.3 节）和第 12 章，第 10 章和第 13 章留给感兴趣的读者。作者讲授编译原理课程的教学课件和历年试题在作者教学网页上。

对于课程的综合实践，若不打算涉及代码生成和代码优化，作者建议选用《编译原理与技术》附录给出的基于 PL/0 语言的课程实践项目，该项目的参考资料、项目要求、测试程序，以及用 C 语言编写的 PL/0 语言编译器和解释器的源代码都可以从作者教学网页下载。若打算实施覆盖编译所有阶段的大型综合课程实践，可以采用作者编写的《编译原理实验教程》（见文献 [10]）上的项目，相关资料也可以从作者教学网页下载。

本次改版得到中国科学技术大学的支持和一些专家的建议，得到了高等教育出版社刘艳编辑的匡正与审订。作者谨向所有为本次改版提供支持与帮助的人员致以最诚挚的谢意。

虽经多次改版，书中仍难免存在不妥和错误之处。若发现本书有错误或对本书提出建议，

编译原理（第4版） 正文 一校一次 文前+编译原理(第4版) 一校.pdf 第5页 文件编号: 422623 页面编号: 154538417 打印人: 8685 2026-03-09 10:22:34

欢迎发送电子邮件（yiyin@ustc.edu.cn,yuzhang@ustc.edu.cn）批评指正，作者将与出版社联系，在再次印刷或再版时改正错误和采纳合理的建议。

作　者

于中国科学技术大学

2013 年 8 月

第 2 版前言

本书改版的动力主要来自以下几个方面：

（1）编译器技术的发展，特别是计算机体系结构的发展对编译器技术的推动；

（2）第 1 版教材在使用中得到的积极反馈和建议；

（3）作者在教学实践中的心得和体会，在相关科研活动中的进展与收获。

第 2 版增加了有关新理论和新技术的介绍，重写了因理论和技术发展而需要调整的部分，改写了原先有疏漏而需要改善的地方，并且继承和发扬了第 1 版的特色。

本次改动主要体现在以下几方面：

（1）增加了第 10 章“依赖于机器的优化”。简要介绍现代计算机体系结构、指令调度、基本块调度、全局调度、软件流水、并行性和数据局部性优化。

（2）重写了第 9 章“独立于机器的优化”。突出数据流分析的理论基础，强调在数据流分析的一般框架下解决各个具体数据流问题，使本章立足于更高层面讨论代码优化。

（3）改写了第 5 章“类型检查”的前两节。减少一些不必要的概念，把保留的概念区分得更清楚一些。

（4）删掉一些过时的或较少使用的内容。此外，考虑到 Pascal 语言目前已较少使用，因此，把第 1 版中采用的大部分 Pascal 示例代码改为 C 语言示例代码。

本次改版主要参考了文献 [9]，在此向作者表示衷心的感谢。

特别感谢中国科学院计算技术研究所张兆庆研究员和冯晓兵研究员，他们在百忙之中仔细审阅了全稿，并对书稿提出了宝贵的意见。中国科学技术大学为本书的编写提供了充分的资金支持。付雄博士为新增的第 10 章准备了初稿。在此谨向所有为本次改版提供支持与帮助的人员致以最诚挚的谢意。

未加星号的各章已构成编译原理知识的一个基本架构，本科阶段的教学可在此基础上适当地增删，加星号的各章可供更深入学习时使用。与本书配套的习题解答是由作者编写、高等教育出版社出版的《编译原理习题精选与解析》。由作者编写的课程实践教材《编译原理实验教程》将在 2008 年年底由高等教育出版社出版。作者讲授编译原理课程的教学课件和历年试题可在作者的教学网页上下载。

编译原理（第4版） 正文 一校一次 文前+编译原理(第4版) 一校.pdf 第7页 文件编号: 422623 页面编号: 174538413 打印人: 8685 2026-03-09 10:22:34

限于时间和学识水平，书中难免存在不妥和错误之处。如果发现本书有任何错误或有任何建议，欢迎给作者发送电子邮件（yiyun@ustc.edu.cn,yunzhang@ustc.edu.cn）批评指正，作者将及时改正错误。

作　者

于中国科学技术大学

2008 年 3 月

第 1 版前言

本书介绍编译器构造的一般原理、基本设计方法和主要实现技术，可作为高等院校计算机科学及相关专业的教材。

虽然只有少数人从事构造或维护程序设计语言编译器的工作，但是编译原理和技术对高校学生和计算机软件工程技术人员来说仍是重要的基础知识之一。本书能使读者对程序设计语言的设计和实现有深刻的理解，对与程序设计语言有关的理论有所了解，对宏观上把握程序设计语言来说，能起一个奠基的作用。本书的学习还有助于读者快速理解、定位和解决在程序调试与运行中出现的问题。

对软件工程来说，编译器是一个很好的实例（基本设计、模块划分、基于事件驱动的编程等），本书所介绍的概念和技术能应用到一般的软件设计之中。

大多数程序员同时也是语言的设计者，虽然是一些简单语言（如输入输出、脚本语言）的设计者，但学习本书有助于提高他们设计这些语言的水平。

编译技术在软件安全、程序理解和软件逆向工程等方面有着广泛的应用。

作为一本教材，本书具有如下一些特点：

（1）在介绍语言实现技术的同时，强调一些相关的理论知识，如形式语言和自动机理论、语法制导的定义和属性文法、类型论和类型系统等。它们是计算机专业理论知识的一个重要部分，本书结合应用来介绍这些知识，有助于学生较快领会和掌握。

（2）在介绍编译器各逻辑阶段的实现时，强调形式化描述技术，并以语法制导定义作为翻译的主要描述工具。

（3）强调对编译原理和技术的宏观理解及全局把握，而不是把读者的注意力分散到一些枝节的算法上，如计算开始符号集合和后继符号集合的算法、回填技术等。出于同样的目的，本书较详细地介绍了编译系统和运行系统。

（4）本书还介绍面向对象语言和函数式语言的实现技术，有助于加深读者对语言实现技术的理解。书中带星号的章节，作为教学的可选部分。

（5）作为原理性教材，本书主要介绍基本的理论和方法，而不偏向于某种源语言或目标机器。

编译原理（第4版） 正文 一校一次 文前+编译原理(第4版) 一校.pdf 第9页 文件编号: 422623 页面编号: 194538419 打印人: 8685 2026-03-09 10:22:34

（6）我们鼓励读者用所学的知识去分析和解决实际问题，因此本书中有很多习题是从实际碰到的问题中抽象出来的。这些习题也能激发读者学习编译原理和技术的积极性。

（7）为了便于读者学习，本书配有习题解答（见文献 [8]）。

本书的多数章节是参考了文献 [1] 和文献 [7] 编写的，部分习题取自文献 [8]。在此向有关作者表示感谢。

本书第 1 章到第 6 章以及第 12 章主要由陈意云编写，第 7 章到第 11 章主要由张昱编写。作者的学生陈晖准备了 10.2 节的初稿，李筱青准备了 10.3 节的初稿，吴萍和项森也为第 10 章的编写做了很多技术工作。

中国科学院软件研究所研究员程虎先生审阅了全书，并提出了许多宝贵的意见，在此表示衷心的感谢。

由于作者水平有限，书中难免存在一些缺点和错误，恳请广大读者批评指正。

作　者

于中国科学技术大学

2003 年 5 月

编译原理（第4版） 正文 一校一次 文前+编译原理(第4版) 一校.pdf 第10页 文件编号: 422623 页面编号: 2104538414 打印人: 8685 2026-03-09 10:22:34

目　录

第一章　引论 ····· 1
1.1　编程语言 ····· 1
1.1.1　编程语言比较：同一问题的不同编程语言表示 ····· 2
1.1.2　编程语言的主要特点 ····· 4
1.1.3　编程语言的演变历史 ····· 5
1.2　编译过程 ····· 8
1.2.1　词法分析 ····· 9
1.2.2　语法分析 ····· 9
1.2.3　语义分析 ····· 10
1.2.4　中间代码生成 ····· 10
1.2.5　代码优化 ····· 11
1.2.6　代码生成 ····· 12
1.2.7　符号表管理 ····· 13
1.2.8　阶段的分组 ····· 13
1.3　编译器的作用及形式 ····· 14
1.3.1　编译器的作用 ····· 14
1.3.2　编译器的形式 ····· 16
1.4　主流的编译基础设施及发展 ····· 18
1.4.1　通用编译基础设施：GCC 和 LLVM ····· 18
1.4.2　具有代表性的现代编程语言的编译运行时系统 ····· 20
1.4.3　国内编译技术生态 ····· 21
1.4.4　新兴计算领域的编程系统 ····· 23
1.5　编译技术的发展与应用 ····· 25
1.5.1　高级语言演变与实现 ····· 26
1.5.2　针对计算机体系结构的优化 ····· 26
1.5.3　新计算机体系结构的设计 ····· 27
1.5.4　程序翻译 ····· 28
1.5.5　提高软件开发效率的工具 ····· 28
习题 ····· 29
第二章　词法分析 ····· 30
2.1　词法记号、属性及词法错误 ····· 30
2.1.1　词法单元、词法记号、模式 ····· 31
2.1.2　词法记号的属性 ····· 32
2.1.3　词法错误 ····· 33
2.2　词法记号的描述与识别 ····· 33
2.2.1　字母表、串和语言 ····· 33
2.2.2　正规式 ····· 35
2.2.3　正规定义 ····· 36
2.2.4　状态转换图 ····· 37
2.2.5　基于转换图的词法分析器的构造 ····· 40
2.2.6　词法分析中的冲突及解决 ····· 41
2.3　有限自动机 ····· 42
2.3.1　不确定的有限自动机 ····· 42
2.3.2　确定的有限自动机 ····· 44
2.4　从正规式到有限自动机 ····· 45
2.4.1　从正规式构造 NFA ····· 45
2.4.2　NFA 到 DFA 的变换 ····· 48
2.4.3　DFA 的化简 ····· 51
2.5　词法分析器的生成器 ····· 53
2.5.1　词法分析器的生成器 Lex ····· 53
2.5.2　ANTLR 中的 Lex 部分 ····· 57
习题 ····· 59

编译原理（第4版） 正文 一校一次 文前+编译原理(第4版) 一校.pdf 第11页 文件编号: 422623 页面编号: 2114538412 打印人: 8685 2026-03-09 10:22:34

第三章 语法分析 ····· 63
3.1 上下文无关文法 ····· 64
3.1.1 正规式的表达能力不足 ····· 64
3.1.2 上下文无关文法的定义 ····· 64
3.1.3 正规式与上下文无关文法的比较 ····· 66
3.1.4 分离词法分析器的理由 ····· 67
3.1.5 推导 ····· 67
3.1.6 解析树 ····· 69
3.1.7 二义性 ····· 69
3.2 语言和文法 ····· 70
3.2.1 验证文法产生的语言 ····· 70
3.2.2 无二义性的表达式文法 ····· 71
3.2.3 消除二义性 ····· 72
*3.2.4 文法的种类——形式语言简介 ····· 73
*3.2.5 非上下文无关的语言构造 ····· 74
3.3 自上而下解析 ····· 76
3.3.1 自上而下解析的一般方法 ····· 76
3.3.2 消除左递归 ····· 77
3.3.3 提左因子 ····· 78
3.3.4 LL(1) 文法 ····· 79
3.3.5 递归下降的预测解析 ····· 81
3.3.6 非递归的预测解析 ····· 82
3.3.7 构造预测分析表 ····· 84
3.3.8 预测解析的错误恢复 ····· 87
3.4 LL 解析器的生成器：ANTLR ····· 89
3.4.1 LL(*) 解析技术概述 ····· 90
3.4.2 谓词文法及其含义 ····· 90
3.4.3 ANTLR 3 的 LL(*) 解析算法 ····· 92
3.4.4 ANTLR 4 的 ALL(*) 解析算法 ····· 95
3.4.5 ANTLR 的使用 ····· 97
3.5 自下而上解析 ····· 103
3.5.1 归约 ····· 103
3.5.2 句柄 ····· 104
3.5.3 用栈实现移进–归约解析 ····· 105
3.5.4 移进–归约解析的冲突 ····· 106
3.6 LR 解析器 ····· 108
3.6.1 LR 解析算法 ····· 108
3.6.2 LR 文法和 LR 解析方法的特点 ····· 111
3.6.3 从文法构造识别活前缀的 DFA ····· 113
3.6.4 从识别活前缀的 DFA 构造 SLR 分析表 ····· 119
3.6.5 构造规范的 LR 分析表 ····· 123
3.6.6 构造 LALR 分析表 ····· 124
3.6.7 非二义且非 LR 的上下文无关文法 ····· 127
3.7 二义文法的应用 ····· 129
3.7.1 使用运算符的优先级和结合性来解决冲突 ····· 129
3.7.2 使用其他约定来解决冲突 ····· 131
3.7.3 LR 解析的错误恢复 ····· 132
3.8 LR 解析器的生成器 Yacc ····· 134
3.8.1 使用 Yacc 生成 LR 解析器 ····· 134
3.8.2 用 Yacc 处理二义文法 ····· 137
3.8.3 Yacc 的错误恢复 ····· 140
习题 ····· 141
第四章 语法制导的翻译 ····· 148
4.1 语法制导的定义 ····· 148
4.1.1 语法制导定义的形式 ····· 148
4.1.2 综合属性 ····· 150
4.1.3 继承属性 ····· 150
4.1.4 属性依赖图 ····· 151
4.1.5 *L* 属性定义 ····· 152
4.1.6 属性计算次序 ····· 153
4.2 *S* 属性定义的自下而上计算 ····· 154
4.2.1 语法树 ····· 154
4.2.2 构造语法树的语法制导定义 ····· 155

编译原理（第4版） 正文 一校一次 文前+编译原理(第4版) 一校.pdf 第12页 文件编号: 422623 页面编号: 2124538410 打印人: 8685 2026-03-09 10:22:34

4.2.3 S 属性定义的翻译方案 ····· 156
4.2.4 S 属性的自下而上计算 ····· 157
4.3 L 属性定义的自上而下计算 ···· 159
4.3.1 L 属性定义的翻译方案 ····· 159
4.3.2 预测翻译器的设计 ····· 163
4.3.3 用综合属性代替继承属性 ····· 164
4.4 L 属性定义的自下而上计算 ···· 165
4.4.1 删除翻译方案中的内嵌动作 ··· 166
4.4.2 分析栈上的继承属性 ····· 166
4.4.3 模拟继承属性的计算 ····· 169
4.5 基于树访问的翻译 ····· 171
4.5.1 访问者模式 ····· 171
4.5.2 ANTLR 解析对访问者模式的支持 ····· 172
习题 ····· 177
第五章 类型检查 ····· 180
5.1 类型在编程语言中的作用 ····· 181
5.1.1 类型化语言和类型系统 ····· 181
5.1.2 执行错误和安全语言 ····· 182
5.1.3 不安全性及其检查 ····· 183
5.1.4 类型化语言的优点 ····· 184
5.2 类型系统的描述语言 ····· 185
5.2.1 定型断言 ····· 186
5.2.2 定型规则 ····· 187
5.2.3 类型检查和类型推断 ····· 188
5.3 一个简单类型检查器的规范 ···· 189
5.3.1 一个简单的语言 ····· 189
5.3.2 类型系统 ····· 190
5.3.3 类型检查 ····· 192
5.3.4 类型转换 ····· 195
5.4 类型表达式的等价 ····· 196
5.4.1 类型表达式的结构等价 ····· 197
5.4.2 类型表达式的名字等价 ····· 197
5.4.3 记录类型 ····· 198
5.4.4 类型表示中的环 ····· 199
*5.5 多态函数 ····· 200
5.5.1 为什么要使用多态函数 ····· 200
5.5.2 类型变量 ····· 201
5.5.3 一个含多态函数的语言 ····· 202
5.5.4 代换、实例与合一 ····· 204
5.5.5 多态函数的类型检查 ····· 206
5.6 函数和运算符的重载 ····· 210
5.6.1 子表达式的可能类型集合 ····· 210
5.6.2 缩小可能类型的集合 ····· 211
习题 ····· 212
第六章 运行时存储空间的组织和管理 ··· 219
6.1 名字、绑定与作用域 ····· 220
6.1.1 过程 ····· 220
6.1.2 名字的作用域和绑定 ····· 220
6.1.3 活动记录 ····· 222
6.1.4 局部数据的布局 ····· 223
6.1.5 程序块 ····· 224
6.2 全局栈式存储分配 ····· 225
6.2.1 运行时内存的划分 ····· 225
6.2.2 活动树和运行栈 ····· 226
6.2.3 调用序列和返回序列 ····· 228
6.2.4 不同计算机架构下寄存器在过程调用中的使用 ····· 231
6.2.5 参数个数可变 ····· 237
6.2.6 栈上可变长度数据 ····· 238
6.2.7 悬空引用 ····· 238
6.3 非局部名字的访问 ····· 239
6.3.1 无过程嵌套的静态作用域 ····· 240
*6.3.2 有过程嵌套的静态作用域 ···· 240
*6.3.3 动态作用域 ····· 244
6.4 参数传递 ····· 245
6.4.1 值调用 ····· 245
6.4.2 引用调用 ····· 246
6.4.3 换名调用 ····· 246
*6.5 堆管理 ····· 247

编译原理（第4版） 正文 一校一次 文前+编译原理(第4版) 一校.pdf 第13页 文件编号: 422623 页面编号: 2134538418 打印人: 8685 2026-03-09 10:22:34

6.5.1 内存管理器 ····· 248
6.5.2 计算机内存分层 ····· 249
6.5.3 程序局部性 ····· 250
6.5.4 手工回收请求 ····· 251
习题 ····· 251
第七章 中间语言与中间代码生成 ····· 266
7.1 中间语言 ····· 267
7.1.1 后缀表示和前缀表示 ····· 267
7.1.2 图形表示 ····· 267
7.1.3 三地址代码 ····· 268
7.1.4 静态单赋值形式 ····· 270
7.2 基本块和流图 ····· 271
7.2.1 基本块 ····· 271
7.2.2 流图 ····· 272
7.3 LLVM 编译系统与 LLVM IR ··· 273
7.3.1 LLVM 编译系统 ····· 274
7.3.2 LLVM IR ····· 275
7.3.3 LLVM 的遍管理器 ····· 280
7.3.4 LLVM 的应用场景 ····· 283
7.4 中间代码生成概述 ····· 283
7.5 声明语句 ····· 285
7.5.1 仅包含主过程时声明语句的翻译 ····· 285
7.5.2 作用域信息的保存 ····· 286
7.5.3 记录的域名 ····· 289
7.6 赋值语句 ····· 289
7.6.1 符号表中的名字 ····· 290
7.6.2 数组元素的地址计算 ····· 290
7.6.3 数组元素地址计算的翻译方案 ··· 292
7.6.4 类型转换 ····· 294
7.7 布尔表达式和控制流语句 ····· 295
7.7.1 布尔表达式 ····· 295
7.7.2 控制流语句的翻译 ····· 296
7.7.3 布尔表达式的控制流翻译 ····· 298
7.7.4 开关语句的翻译 ····· 300
7.7.5 过程调用的翻译 ····· 303
习题 ····· 304
第八章 代码生成 ····· 311
8.1 代码生成器设计中的问题 ····· 311
8.1.1 目标程序 ····· 311
8.1.2 指令选择 ····· 312
8.1.3 编译器生成的不同架构汇编码示例 ····· 313
8.1.4 寄存器分配 ····· 315
8.1.5 计算次序选择 ····· 316
8.2 目标语言 ····· 316
8.2.1 目标机器的指令集 ····· 316
8.2.2 指令代价 ····· 317
8.2.3 指令选择的实现：中间代码树和 DAG ····· 318
8.2.4 LLVM 后端描述方法：TableGen ····· 320
8.3 基本块和流图 ····· 323
8.3.1 基本块的优化 ····· 324
8.3.2 下次引用信息 ····· 325
8.4 一个简单的代码生成器 ····· 326
8.4.1 寄存器描述和地址描述 ····· 327
8.4.2 代码生成算法 ····· 327
8.4.3 寄存器选择函数 ····· 328
8.4.4 为变址和指针语句产生代码 ··· 329
8.4.5 条件语句 ····· 330
8.5 寄存器分配算法 ····· 331
8.5.1 线性扫描算法 ····· 332
8.5.2 图着色算法 ····· 333
8.5.3 LLVM 中的寄存器分配算法 ··· 335
习题 ····· 335
第九章 独立于机器的代码优化 ····· 343
9.1 优化的主要种类 ····· 343
9.1.1 优化的主要源头 ····· 344
9.1.2 一个实例 ····· 344

编译原理（第4版） 正文 一校一次 文前+编译原理(第4版) 一校.pdf 第14页 文件编号: 422623 页面编号: 2144538416 打印人: 8685 2026-03-09 10:22:34

9.1.3 公共子表达式删除 ····· 346
9.1.4 复写传播 ····· 348
9.1.5 死代码删除 ····· 349
9.1.6 代码外提 ····· 350
9.1.7 强度削弱和归纳变量删除 ····· 350
9.2 数据流分析介绍 ····· 352
9.2.1 数据流抽象 ····· 353
9.2.2 数据流分析模式 ····· 354
9.2.3 到达–定值 ····· 355
9.2.4 活跃变量 ····· 360
9.2.5 可用表达式 ····· 361
9.2.6 小结 ····· 364
9.3 数据流分析的基础 ····· 364
9.3.1 半格 ····· 365
9.3.2 迁移函数 ····· 368
9.3.3 一般框架的迭代算法 ····· 370
9.3.4 数据流解的含义 ····· 372
9.4 常量传播 ····· 374
9.4.1 常量传播框架的数据流值 ····· 374
9.4.2 常量传播框架的迁移函数 ····· 375
9.4.3 常量传播框架的单调性 ····· 376
9.4.4 常量传播框架的非分配性 ····· 377
9.4.5 结果的解释 ····· 378
9.5 部分冗余删除 ····· 379
9.5.1 冗余的根源 ····· 379
9.5.2 能否删除所有的冗余 ····· 381
9.5.3 惰性代码移动问题 ····· 382
9.5.4 预期表达式 ····· 383
9.5.5 惰性代码移动算法 ····· 383
9.6 流图中的循环 ····· 389
9.6.1 支配结点 ····· 389
9.6.2 回边和可归约性 ····· 391
9.6.3 流图的深度 ····· 393
9.6.4 自然循环 ····· 393
9.6.5 迭代流图算法的收敛速度 ····· 394
习题 ····· 396
* 第十章 依赖于机器的代码优化 ····· 406
10.1 处理器体系结构 ····· 407
10.1.1 指令流水线和分支延迟 ····· 407
10.1.2 流水化的执行 ····· 408
10.1.3 多指令发射 ····· 408
10.2 代码调度的约束 ····· 409
10.2.1 数据相关 ····· 409
10.2.2 发现内存访问中的相关性 ····· 410
10.2.3 寄存器使用和并行执行之间的折中 ····· 411
10.2.4 寄存器分配和代码调度的次序安排 ····· 412
10.2.5 控制相关 ····· 413
10.2.6 投机执行的支持 ····· 414
10.2.7 一个基本的机器模型 ····· 415
10.3 基本块调度 ····· 415
10.3.1 数据依赖图 ····· 415
10.3.2 基本块的表调度 ····· 417
10.3.3 区分优先级的拓扑次序 ····· 418
10.4 全局代码调度 ····· 418
10.4.1 简单的代码移动 ····· 419
10.4.2 向上的代码移动 ····· 420
10.4.3 向上的代码移动 ····· 421
10.4.4 更新数据相关 ····· 422
10.4.5 全局调度的其他问题 ····· 422
10.4.6 静态调度器和动态调度器的相互影响 ····· 423
10.5 软件流水 ····· 424
10.5.1 引言 ····· 424
10.5.2 循环的软件流水 ····· 426
10.5.3 寄存器分配和代码生成 ····· 428
10.5.4 do-across 循环 ····· 428
10.5.5 软件流水的目标和约束 ····· 430
10.5.6 软件流水算法 ····· 432

编译原理（第4版） 正文 一校一次 文前+编译原理(第4版) 一校.pdf 第15页 文件编号: 422623 页面编号: 2154538413 打印人: 8685 2026-03-09 10:22:34

10.5.7 无环数据依赖图的调度 …… 432
10.6 并行性和数据局部性优化概述 … 434
10.6.1 多处理器 …… 434
10.6.2 应用中的并行性 …… 435
10.6.3 循环级并行 …… 437
10.6.4 数据局部性 …… 438
10.6.5 矩阵乘法算法 …… 439
10.6.6 矩阵乘法算法的优化 …… 441
习题 …… 443
第十一章 编译系统和运行时系统 …… 446
11.1 C 语言的编译系统 …… 446
11.1.1 预处理器 …… 447
11.1.2 汇编器 …… 448
11.1.3 连接器 …… 450
11.1.4 目标文件的格式 …… 451
11.1.5 符号解析 …… 453
11.1.6 静态库 …… 454
11.1.7 可执行的目标文件及装入 …… 456
11.1.8 动态连接 …… 457
11.1.9 处理目标文件的一些工具 …… 459
11.2 基于虚拟机的语言运行时：以 Java 和 JVM 为例 …… 459
11.2.1 Java 虚拟机语言简介 …… 460
11.2.2 Java 虚拟机 …… 461
11.2.3 即时编译器 …… 463
*11.3 无用单元收集 …… 464
11.3.1 标记和清扫 …… 465
11.3.2 引用计数 …… 466
11.3.3 复制收集 …… 467
11.3.4 分代收集 …… 468
11.3.5 渐增式收集 …… 469
11.3.6 编译器与收集器之间的相互影响 …… 470
习题 …… 473
第十二章 面向对象语言的编译 …… 478
12.1 面向对象语言的概念 …… 478
12.1.1 对象和对象类 …… 478
12.1.2 继承 …… 479
12.1.3 信息封装 …… 481
12.2 方法的编译 …… 481
12.3 继承的编译方案 …… 484
12.3.1 单一继承的编译方案 …… 485
12.3.2 多重继承的编译方案 …… 487
习题 …… 492
第十三章 函数式语言的编译 …… 495
13.1 函数式编程语言简介 …… 495
13.1.1 语言构造 …… 495
13.1.2 参数传递机制 …… 497
13.1.3 变量的自由出现和约束出现 … 498
13.2 函数式语言的编译简介 …… 499
13.2.1 几个受启发的例子 …… 500
13.2.2 编译函数 …… 502
13.2.3 环境与约束 …… 502
13.3 抽象机的体系结构 …… 503
13.3.1 抽象机的栈 …… 504
13.3.2 抽象机的堆 …… 505
13.3.3 名字的寻址 …… 506
13.3.4 约束的建立 …… 507
13.4 指令集和编译 …… 508
13.4.1 表达式 …… 508
13.4.2 变量的引用出现 …… 510
13.4.3 函数定义 …… 511
13.4.4 函数应用 …… 512
13.4.5 构造和计算闭包 …… 515
13.4.6 **letrec** 表达式和局部变量 … 517
习题 …… 519
参考文献 …… 521

第 1 章

引 论

编程语言（programming language），又称程序设计语言，是用于描述计算的符号系统。用高级语言编写的程序怎样才能在计算机上执行呢？理论上，构造专用计算机直接执行用某种高级语言编写的程序是可能的，如 1974 年美国麻省理工学院计算机科学与人工智能实验室研制的 CONS 机器可以直接执行 LISP 语言程序。但是这类做法并不多见，因为存在开发成本高、灵活性差和市场接受度低等问题。实际上，计算机能直接执行的都是低级的**机器语言**程序，是由 0 和 1 组成的指令序列，称为二进制代码。其中每条指令对应计算机硬件中的一种特定操作。每种计算机架构都有其特定的机器语言指令集，如 x86 架构的机器语言、ARM 架构的机器语言、龙架构的机器语言。

高级语言程序需要翻译成机器语言程序才能在计算机上执行。能够完成从一种语言到另一种语言的保语义变换的软件称为**翻译器**，这两种语言分别称为该翻译器的**源语言**和**目标语言**。**编译器**是一种翻译器，它的特点是目标语言比源语言低级。

本章简要介绍编程语言及其编译器的发展阶段、作用和形式，以及主流编译基础设施及其发展。编译技术涉及编程语言、计算机体系结构、形式理论、类型论、算法和软件工程等方面的知识。

1.1 编 程 语 言

编程语言作为描述计算的符号系统，既向人类描述计算逻辑，又向机器提供可执行指令。通过一组语法规则和语义规范，编程语言使开发者能够编写、阅读、维护和优化程序代码，从而实现与计算机的有效沟通和各种任务的执行。编程语言的设计需要在易于人类理解和高效机器执行之间找到平衡。这种权衡为编程语言的设计和实现带来了持续的挑战。

自 1966 年设立 ACM 图灵奖以来，已有约三分之一的获奖者因其在程序设计语言和编译领域的杰出贡献而获此殊荣。例如，阿尔弗雷德·艾侯（Alfred Aho）和杰弗里·乌尔曼（Jeffrey U llman）因其在编程语言实现的基础算法和理论方面的卓越贡献（他们所著的《编译程序设计原理》被称为“龙书”）而共同荣获 2020 年 ACM 图灵奖。这些获奖者的工作为

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第2页 文件编号: 422624 页面编号: 124538448 打印人: 8685 2026-03-09 10:22:57

现代编程语言的翻译技术做出重要贡献，为计算机科学的发展奠定了坚实的基础。

本节从同一问题的不同编程语言表示入手，让读者了解不同编程语言的特点及其演变历史。

1.1.1 编程语言比较：同一问题的不同编程语言表示

编程语言数量众多。“在线编程语言历史百科全书”记录了从 18 世纪至今的 8945 种编程语言；而 GitHub 软件仓库在 2022 年使用了近 500 种主要语言构建。

不同编程语言解决同一问题的方法和思路可能有很大差异。这不仅反映了语言的语法和特性的差异，还体现了编程范式、工具库和社区支持的不同。本小节以求解最大公约数（greatest common divisor, GCD）问题为例，展示用 Python、C、Java、JavaScript、OCaml、Prolog 这 6 种编程语言实现相同功能的计算。为简便起见，编程者可以用 Compiler Explorer 或 OneCompiler 在线编译网站窗口中编译和运行程序。需注意前者不支持 Prolog 编程语言。

图 1.1 给出了 Compiler Explorer 在线编译网站的窗口，其中从左到右依次为源代码、编译器、运行输出、LLVM IR 输出 4 个子窗口。你可以在前两个子窗口中分别设置编程语言和编译器。而 OneCompiler 在线编译网站的功能相对简单，只提供源代码和运行输出两个子窗口。

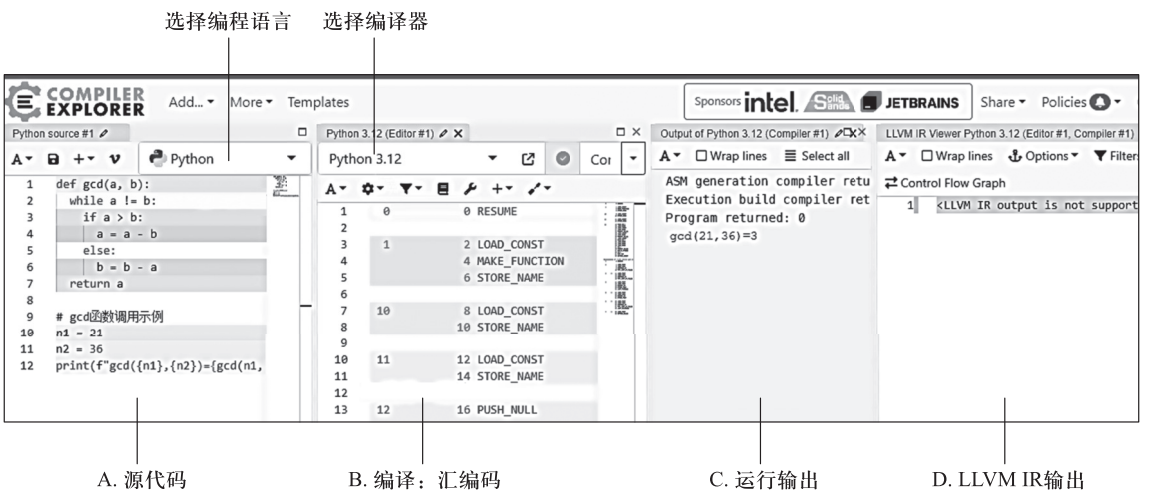


图 1.1 Compiler Explorer 在线编译网站窗口

图 1.2 列出了使用辗转相减法（尼考曼彻斯法）求解最大公约数的 6 种编程语言实现。每种语言程序在语法和函数调用方式上有所不同。辗转相减的循环处理在 C、Java 语言中的表示形式完全一样，分别见图 1.2(b) 和图 1.2(c) 的第 3~8 行；而这部分在 JavaScript 中为图 1.2(d) 的第 2~7 行，它们几乎一样，只是 JavaScript 语言中的不等号用“!==”表示。三

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第3页 文件编号: 422624 页面编号: 134538446 打印人: 8685 2026-03-09 10:22:57

种语言的 gcd 函数头和函数调用及结果输出的语法表示都有所不同，体现出不同编程语言在具体语法形式上的差异性。

```python
def gcd(a, b):
  while a != b:
    if a > b:
      a = a - b
    else:
      b = b - a
  return a

# gcd函数调用示例
n1 = 21
n2 = 36
print(f"gcd({n1},{n2})={gcd(n1,␣n2)}")
```

(a) Python 程序 gcd.py

```c
#include <stdio.h>
int gcd(int a, int b) {
  while (a != b) {
    if (a > b)
      a = a - b;
    else
      b = b - a;
  }
  return a;
}
int main() {
  // gcd函数调用示例
  int n1 = 21, n2 = 36;
  printf("gcd(%d,%d)=%d\n",n1,n2, gcd(n1, n2));
  return 0;
}
```

(b) C 程序 gcd.c

```java
public class GCD {
  public static int gcd(int a, int b) {
    while (a != b) {
      if (a > b)
        a = a - b;
      else
        b = b - a;
    }
    return a;
  }
  public static void main(String[] args){
    // gcd函数调用示例
    int n1 = 21, n2 = 36;
    System.out.println("gcd("+n1+","+n2+")="+gcd(n1, n2
        ));
  }
}
```

(c) Java 程序 GCD.java†

```javascript
function gcd(a, b) {
  while (a !== b) {
    if (a > b)
      a = a - b;
    else
      b = b - a;
  }
  return a;
}
// gcd函数调用示例
let n1 = 21;
let n2 = 36;
console.log('gcd(${n1},${n2})=${gcd(n1, n2)}');
```

(d) JavaScript 程序 gcd.js

```prolog
gcd(A, B, G) :-
  A = B, G = A.
gcd(A, B, G) :-
  A > B, A1 is A - B, gcd(A1, B, G).
gcd(A, B, G) :-
  A < B, B1 is B - A, gcd(A, B1, G).

% gcd示例查询
:- gcd(21, 36, Result), write('gcd(21,36)='), write(
      Result), nl.
```

(e) Prolog 程序 gcd.pl

```ocaml
let rec gcd a b =
  if a = b then a
  else if a > b then gcd (a - b) b
  else gcd a (b - a)
(* gcd函数调用示例 *)
let n1 = 21
let n2 = 36
let result = gcd n1 n2
let () = Printf.printf "gcd(%d,%d)=%d\n" n1 n2 result
```

(f) OCaml 程序 gcd.ml

注: 在 Compiler Explorer 在线编译网站上编译图 (c) 所示的 Java 程序时会报错，这是因为该平台在默认情况下无法修改文件名，解决办法是去掉图 (c) 中第 1 行的 public。

图 1.2 求解最大公约数 (GCD) 的不同编程语言实现对比

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第4页 文件编号: 422624 页面编号: 144538444 打印人: 8685 2026-03-09 10:22:58

Python、JavaScript、Prolog 和 OCaml 语言都无须声明参数（a、b）和变量（n1 和 n2）的类型，这是因为前三种语言是动态定型语言（变量的类型在运行时决定），而 OCaml 则是有强大类型推断能力的静态定型语言。Prolog 和 OCaml 语言对 gcd 都采用分情况的递归定义，递归调用的结束条件是 gcd 的前两个参数的值相等，图 1.2 (e) 和 (f) 中的第 2 行。

1.1.2 编程语言的主要特点

可读性：对人类来说，编程语言设计得越清晰，代码就越容易理解和维护。常见的编程语言如 Python，以其简洁和易读著称。而对机器来说，编程语言则需要通过编译器或解释器将人类编写的代码转换为机器代码，使计算机能够理解和执行。

抽象能力：编程语言提供不同级别的抽象，帮助开发者专注于解决问题，而不是陷入底层实现细节。例如，高级语言如 Python 和 Java 隐藏了许多底层操作细节，而中级语言如 C 及低级语言如汇编语言则提供对硬件的直接控制。编程语言中的抽象能力包括过程抽象、数据抽象、控制抽象、模块化抽象、对象抽象、泛型和多态、元编程（如反射和注解）、并发抽象、领域特定语言等。

表达能力：编程语言需要具备表达各种计算的算法、逻辑、数据结构等的能力。这种能力不仅取决于语言本身的设计特点和提供的库支持，还包括开发者编写代码的技巧和习惯。表达能力可以通过表达的清晰度和简洁度、数据结构和算法的支持、领域特定语言的支持、可扩展性和自定义能力、代码的可读性和可维护性等方面来评估和比较。例如，函数式语言（如 OCaml）以及现代高级语言（如 Python）通常能以较少的代码行实现复杂功能，提升了代码的可读性和可维护性；C++ 和 Java 等语言提供了丰富的标准库，可以直接支持复杂的数据结构和算法；某些语言支持特定领域的语言扩展，如 SQL（structured query language，结构化查询语言）用于数据库查询、MATLAB 用于数学计算和工程领域；还有一些语言如 LISP 和 Scala 通过宏或语言特性的灵活性允许开发者定制语言的行为和语法，以适应不同的编程需求。此外，良好的表达能力还包括代码结构的清晰度、良好的命名规范以及注释的合理使用，这些都有助于提高代码的可读性和可维护性，使开发更为高效和可靠。

类型系统：编程语言的类型系统决定了如何定义、使用和检查变量和函数的数据类型，直接影响代码的安全性、可靠性和表达能力。类型系统可以是静态的（如 C/C++、Java）或动态的（如 Python、JavaScript）。静态定型语言在编译时检查类型，要求显式声明和严格的类型转换，有助于提前捕获错误；动态定型语言在运行时进行类型检查，变量类型根据赋值动态确定，具备灵活性和快速开发的优势，但可能增加运行时类型错误的风险。类型推断和泛型是现代编程语言的重要特性。类型推断减少了冗余的类型声明，而泛型则提高了代码的重用性和类型安全性。

编程范式：编程语言的编程范式（programming paradigm）是指编程语言所支持的编程风格和方法。每种范式都提供了不同的抽象和工具来处理计算任务。不同的编程范式反映了

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第5页 文件编号: 422624 页面编号: 154538441 打印人: 8685 2026-03-09 10:22:58

不同的思维方式和设计理念，适用于不同类型的问题和应用场景。常见的编程范式有：

(1) 命令式（imperative）编程：侧重于描述计算的步骤和指令，如 C、C++、Java、Go、Rust、Python（也支持其他范式）。

(2) 函数式（functional）编程：强调函数的应用和组合，函数是“一等公民”（函数能像数据一样被操作和传递），如 Haskell、ML、OCaml、Scala 和 Clojure。

(3) 逻辑式（logic）编程：使用逻辑表达式描述问题，通过推导来解决问题，如 Prolog、Datalog。

(4) 事件驱动（event-driven）编程：程序的执行由事件触发，如用户输入、消息到达或定时事件，常用于图形用户界面（graphical user interface，GUI）和网络编程，如 JavaScript（网页开发），C#（GUI 开发）。

(5) 并行（parallel）和分布式（distributed）编程：并行编程涉及多个计算单元同时执行任务，可以通过共享内存或消息传递进行通信，如 MPI（message passing interface，消息传递接口）、OpenMP 和 CUDA（用于 GPU 并行计算）。分布式编程涉及在多个计算机上协调任务的执行，如 Hadoop、Spark。

(6) 声明式（declarative）编程：关注描述问题的本质，而非解决方案的具体步骤。通常使用规则或约束来定义问题，如 SQL 和 CSS（用于网页样式定义）。

(7) 面向对象（object-oriented）编程：通过对象、类和继承等概念组织和管理代码。对象是数据和操作的封装，通过消息传递进行交互，如 Java、C++ 和 Python。

(8) 脚本（scripting）编程：常用于编写自动化任务的小程序，具有解释执行和动态类型的特点，适合快速开发和原型设计，如 Python、Ruby、Perl、Bash。

1.1.3 编程语言的演变历史

1. 编程语言的发展

编程语言的发展受计算技术发展和应用需求的双重驱动。这两个因素相互作用，推动了编程语言的不断创新和优化。1942 年诞生的第一台电子计算机 ABC（Atanasoff–Berry Computer）是不可编程的，仅用于求解线性方程组。而 1946 年诞生的第二台电子计算机 ENIAC(electronic numerical integrator and computer) 最初使用插拔电缆和开关设置来支持编程以解决各种计算问题，后来改用打孔卡片以简化编程。自此之后，编程语言的发展经历了从低级到高级、从简单到复杂的过程。

20 世纪 40 至 50 年代：早期的计算机使用机器语言和汇编语言进行编程。

20 世纪 50 至 60 年代：IBM 公司的约翰·巴克斯（John Backus）领导团队研制，并于 1957 年发布世界上第一个高级编程语言——FORTRAN（Formula Translation，公式翻译）语言，它专为科学和工程计算设计。高级编程语言（如 FORTRAN、COBOL 和 LISP）的出现，使编程更高效、更易于理解。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第6页 文件编号: 422624 页面编号: 164538449 打印人: 8685 2026-03-09 10:22:58

20 世纪 70 年代：结构化编程语言（如 C 和 Pascal）的兴起，推动了编程范式的进步。

20 世纪 80 年代：面向对象编程语言（如 Smalltalk 和 C++）的出现，深刻改变了传统的软件开发方法。

20 世纪 90 年代至 21 世纪初：互联网的快速发展催生了 Java、JavaScript 等编程语言，有力推动了 Web 开发技术的进步。

21 世纪初至今：现代编程语言如 Python（1991 年发布但于 2000 年后流行）、Go（2009 年发布）、Rust（2010 年发布）、TypeScript（2012 年发布，扩展 JavaScript 引入静态类型系统和现代编程特性）、Swift（2014 年发布）等进一步提高了编程效率和代码可读性。

为适应多样化应用需求和异构硬件的发展，越来越多的领域特定语言被开发出来。例如，2012 年发布的 Halide 解耦计算定义和调度过程，用于优化图像处理算法；2018 年发布的 TVM 承袭解耦思想，可扩展调度原语以加速深度学习模型的优化。为处理日益增长的复杂关系数据，图数据库（如 Neo4j）及其查询语言 Cypher 分别于 2007 年、2011 年推出。为统一图数据管理和查询的语法与功能，国际标准化组织（International Organization for Standardization，ISO）和国际电工委员会（International Electrotechnical Commission，IEC）于 2024 年 4 月合作推出图查询语言（graph query language，GQL）标准。2024 年 6 月，华为首次发布自主研发的仓颉编程语言，以期支持全场景应用开发。

2. 编程语言自身的演变

除了产生新的编程语言，编程语言自身也在不断演化。例如，C 语言诞生于 1972 年，由丹尼斯·里奇（Dennis Ritchie）在美国电话电报公司（AT&T）的贝尔实验室开发，用于编写 UNIX 操作系统。它引入类型、结构等高级编程构造以便有效管理和操作计算机硬件。为确保一致性和兼容性，美国国家标准学会（American National Standards Institute，ANSI）于 1989 年发布 ANSI C（C89），ISO 于 1990 年采纳为 ISO C（C90）。1999 年的 C99 引入如内联函数、变量声明混合语句、更多的库函数等新特性。2011 年的 C11 增加了多线程和原子操作支持。C++ 语言由比雅尼·斯特劳斯特鲁普（Bjarne Stroustrup）于 1980 年代初开发，以扩展 C 语言支持面向对象编程。1998 年（ISO）发布的 C++98 标准化了核心特性和标准库，2003 年的 C++03 修正了一些错误。2011 年的 C++11 做了重大升级，增加了自动类型推导（auto）、范围 for 循环、lambda 表达式和智能指针等新特性，显著提高了 C++ 编程的效率和安全性。2014 年的 C++14 对其进行了补充和改进。2017 年的 C++17 增加了结构化绑定、折叠表达式等。2020 年的 C++20 则引入协程、模块和概念等，使 C++ 更加强大和现代化。

3. 编程语言流行度的变化

编程语言的流行度随着时间的推移会不断变化，受多种因素的影响，包括技术趋势、社区支持、企业需求和开发工具的创新。GitHub 和 TIOBE 是两个常用的编程语言流行度指标来源，各自有不同的评估方法和关注点。GitHub 作为全球最大的代码托管平台之一，拥有数百万个开源项目，其流行度分析通常反映开发者实际使用的情况和社区活跃度。TIOBE 编程

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第7页 文件编号: 422624 页面编号: 174538447 打印人: 8685 2026-03-09 10:22:58

语言社区指数则通过分析全球范围内的搜索引擎查询来评估编程语言流行度。

GitHub 年度报告从 2012 年开始发布，汇总 GitHub 上一年中开源社区的发展趋势、热门项目、开发者活动等。图 1.3 为 2014—2024 年 GitHub 年度报告列出的前十名流行编程语言。其中，JavaScript 和 Python 近年来一直稳居前两位，反映了它们在前端开发、数据科学和自动化等领域的广泛应用。2024 年，AI 的蓬勃发展更是助推 Python 登顶，成为最受欢迎的编程语言。新兴的 Go 语言在 2022 年跃居第十位，其简洁高效的设计、强大的生态系统（特别是在云计算、容器化和微服务架构领域）和企业支持（由 Google 开发和支持），使其在现代高性能和高并发应用场景中备受青睐。从图 1.3 中还可以看到，TypeScript 的流行度近年持续上升，2023 年升至第 3 位，反映了开发者对类型安全、开发体验和现代框架兼容性的需求增加。

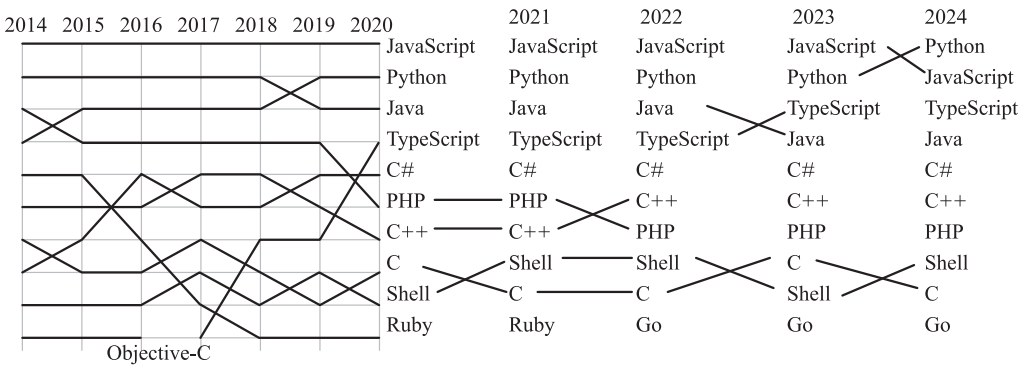


图 1.3 GitHub 上编程语言流行度的变化（2014—2024 年）

GitHub 上的开源项目近年来也呈现出向更安全、更高效、更智能的方向发展。2020 年 GitHub 年度报告的中心是使软件更安全；2021 年强调如何通过工具和框架来更快地编写代码；2022 年，AI 技术（如 Copilot）的引入提升了开发者编写和优化代码的速度；到 2023 年，全球生成式 AI 项目（如 LangChain 和 Stable Diffusion 等）激增，许多项目利用 ChatGPT API 等推动智能化开发。

4. 编程语言历史 HOPL

要想进一步了解编程语言历史、设计理念和技术演变，可以查阅美国计算机协会（ACM）主办的 HOPL（history of programming languages）会议的论文集。HOPL 会议不定期举行，迄今为止已举办四界。

（1）HOPL I（1978 年）：主要讨论早期编程语言的历史和设计理念，包括 FORTRAN、LISP、ALGOL、COBOL 在内的 13 种编程语言。

（2）HOPL Ⅱ（1993 年）：讨论 1960 年代到 1980 年代的编程语言，包括 Ada、C、C++、Smalltalk 等 14 种编程语言，探讨在语言设计、编译技术和编程范式方面的创新和影响。

（3）HOPL Ⅲ（2007 年）：讨论 20 世纪 80 年代至 90 年代编程语言的发展，包括 Python、

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第8页 文件编号: 422624 页面编号: 184538445 打印人: 8685 2026-03-09 10:22:58

Java、Perl、MATLAB 和 Haskell 等 15 种编程语言，还探讨面向对象、函数式和脚本语言的发展。

（4）HOPL Ⅳ（2020 年）：涵盖 1990 年代末到 2010 年代的编程语言发展，包括 JavaScript、Rust、Swift 和 Go 等。HOPL 会议重点讨论了现代编程语言在安全性、并发性和多核处理器上的创新。

编程语言的多样性和演化是技术发展和应用需求的自然反映。没有一种语言能满足所有应用场景。开发者对语言的优劣看法各异，没有普遍标准。随着数字经济的深入，编程语言将继续演化，以满足万物智联下对软件产能、安全性、可靠性和可维护性等更高的需求。

1.2 编 译 过 程

高级语言编写的源代码通过编译器翻译得到目标机器代码。编译过程可以分成若干阶段，每个阶段将源程序从一种表示形式转换为另一种表示形式。编译过程的如图 1.4 所示，左侧每个矩形框表示它的一个阶段。

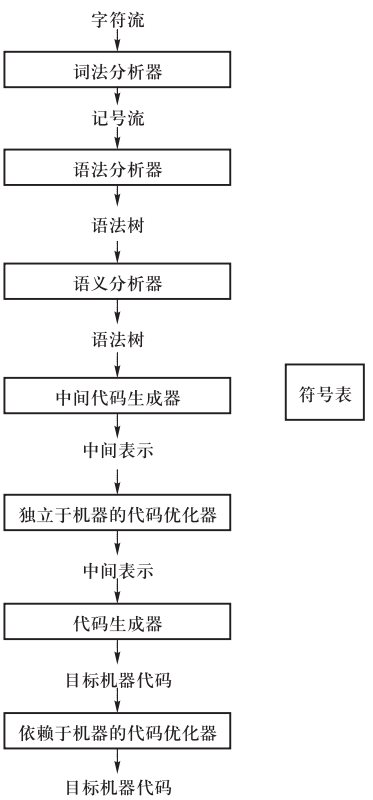


图 1.4 编译过程的各个阶段

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第9页 文件编号: 422624 页面编号: 194538443 打印人: 8685 2026-03-09 10:22:58

本节以赋值语句

$$\text{position=initial+rate*60} \tag{1.1}$$

的翻译（假定变量都是实型）为例，简要介绍编译的各个阶段。

1.2.1 词法分析

词法分析（lexical analysis）读取构成源程序的字符流，按编程语言的词法规则识别出一个个**词法单元**（lexeme，又称词素），再把它们组成**词法记号**（token）流。对于一个词法单元，词法分析产生的记号是二元组：

〈记号名，属性值〉

记号名是同类词法单元共用的名称，而属性值是一个词法单元有别于同类中其他词法单元的特征值。

赋值语句(1.1) 的字符流在词法分析时被依次组成下面这些记号。

（1）标识符 position 形成的记号是〈**id**, 1〉，其中 **id** 是标识符的总称，1 代表 position 在符号表中的条目。符号表的条目用来存放标识符的各种属性，如它的名称和类型。

（2）赋值号 = 形成的记号是〈**assign**〉，因为该记号只有一个实例，所以不需要以属性值来区分实例。为直观起见，下面直接用赋值号作为记号名，写成〈=〉。

（3）标识符 initial 形成的记号是〈**id**, 2〉。

（4）加号 + 形成的记号是〈+〉。

（5）标识符 rate 形成的记号是〈**id**, 3〉。

（6）乘号 * 形成的记号是〈*〉。

（7）数 60 形成的记号是〈60〉。

从技术上讲，程序中出现的常数也要放到符号表或单独的常数表中，形成记号〈**number**, 60 在表中的条目〉。这个问题在第 2 章词法分析中具体介绍。为直观起见，这里直接使用 60 在源程序中的字符序列作为记号名。

分隔单词的空格通常在词法分析时被省略。

于是，赋值语句(1.1)在词法分析后形成的记号流是

$$\langle \mathbf{id}, 1\rangle\langle =\rangle\langle \mathbf{id}, 2\rangle\langle +\rangle\langle \mathbf{id}, 3\rangle\langle *\rangle\langle 60\rangle \tag{1.2}$$

第 2 章将讨论词法分析，词法分析也称为**线性分析**或**扫描**。

1.2.2 语法分析

语法分析（syntax analysis）简称**分析**或**解析**（parsing）。它检查词法分析输出的记号流是否符合编程语言的语法规则，并依据这些规则所体现出的语言**构造**（construct，如函数、语

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第10页 文件编号: 422624 页面编号: 2104538448 打印人: 8685 2026-03-09 10:22:58

句、表达式等）的层次性，用各记号的第一元建成一种树形的中间表示，这个中间表示用抽象语法的方式描绘了该记号流的语法情况。一种典型的中间表示是**语法树**，其中内部结点表示运算，它们的子结点代表该运算的运算对象。为记号流（1.2）建立的语法树如图 1.5(a) 所示。

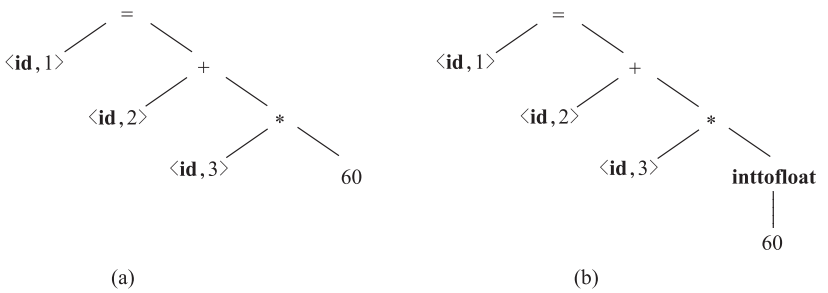


图 1.5 语义分析插入了类型转换

编译器后面各个阶段使用这种语法树进一步分析源程序和生成目标代码。第 3 章专门讨论语法分析算法。图 1.5 这种形式的语法树及其产生方法将在第 4 章讨论。

1.2.3 语义分析

语义分析（semantic analysis）阶段使用语法树和符号表中的信息，依据语言定义来检查源程序各部分之间的语义一致性，以保证程序各部分能有意义地结合在一起。它还收集类型信息，并把它们保存在符号表或语法树中。

语义分析的一个重要部分是**类型检查**（type checking）。编译器检查每个运算符的运算对象，检查它们的类型是否适当。例如，当实数作为数组的下标时，许多语言的定义都要求编译器报告错误。语言定义也可能允许运算对象的类型作隐式转换，如当二元运算符作用于一个整数和一个实数时，编译器会把其中的整数转换为实数。类型检查和语义分析将在第 5 章讨论。

【例 1.1】(算术运算上的自动类型转换) 在机器内部，整数的二进制表示和实数的二进制表示是有区别的，不论它们是否有相同的值。在图 1.5 中，所有的变量都是实型，另外，由 60 本身可知它是整数。对图 1.5 (a) 进行类型检查会发现 * 作用于实型变量 rate 和整数 60，可以建立一个额外的运算符结点 **inttofloat**[见图 1.5 (b)]，它显式地把整数转换为实数。 □

1.2.4 中间代码生成

经过语法分析和语义分析后，许多编译器为源程序产生更低级的显式**中间表示**（intermediate representation），可以把这种中间表示想象成一种抽象机的程序。这种中间表示必须具有两个性质：它易于产生，且易于翻译成目标程序。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第11页 文件编号: 422624 页面编号: 2114538446 打印人: 8685 2026-03-09 10:22:58

第 7 章介绍的中间语言主要采用一种称为**三地址代码**的中间表示形式，它像一种抽象机器的汇编语言，这种机器中存储单元的作用类似于寄存器。三地址代码由三地址指令序列组成，每条三地址指令最多有 3 个操作数，赋值语句（1.1）的三地址代码如下面的指令序列（1.3）。这种中间形式如下特点：首先，除了赋值运算符，每条指令至多还有一个运算符。因此，在生成这些指令时，编译器必须决定运算完成的次序，赋值语句（1.1）中的乘优先于加。其次，编译器必须产生临时变量名，用于保留每条指令的计算结果。第三，某些三地址指令涉及的运算对象不足 3 个，如指令序列（1.3）的第一条和最后一条指令。

$$\begin{aligned} &\mathrm{t1} = \mathrm{inttofloat}(60) \\ &\mathrm{t2} = \mathrm{id3} * \mathrm{t1} \\ &\mathrm{t3} = \mathrm{id2} + \mathrm{t2} \\ &\mathrm{id1} = \mathrm{t3} \end{aligned} \tag{1.3}$$

本书在第 7 章详细介绍编译器采用的主要中间表示形式。通常，除了计算表达式，这些中间表示还要做其他事情，如有条件控制转移、无条件控制转移和过程调用。第 4 章和第 7 章提供了一些为编程语言的典型构造产生中间代码的算法。

1.2.5 代码优化

独立于机器的**代码优化**（code optimization）阶段试图改进中间代码，以便产生较好的目标代码。通常，“较好” 是指执行速度更快，但也可能指目标代码较短或执行时的能耗较低。

如果中间代码生成算法比较简单，就会为代码优化留下很多机会。例如，若中间代码生成算法直接为语法树上的每个运算符产生一条指令，就可以得到如指令序列（1.3）的中间代码，然后进行代码优化，可以产生更优的代码。例如，代码优化器可以推断出，把 60 从整数转换为浮点数可以在编译时完成，从而用 60.0 代替 inttofloat(60)。还有，t3 只被引用一次，即取其值传给 id1，因此用 id1 代替 t3 可以把指令序列（1.3）中的最后一条指令删除。这样，优化器可以得到如下结果：

$$\begin{aligned} &\mathrm{t1} = \mathrm{id3} * 60.0 \\ &\mathrm{id1} = \mathrm{id2} + \mathrm{t1} \end{aligned} \tag{1.4}$$

不同的编译器所实现的优化程度是不同的，能完成大部分优化的编译器称为 “优化编译器”，但这时编译时间中相当可观的部分都消耗在优化上。某些简单的优化也能大大缩短目标程序的运行时间，且并不降低太多编译速度。第 9 章和第 10 章分别讨论独立于机器和依赖于机器的代码优化，第 8 章代码生成也会涉及代码优化。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第12页 文件编号: 422624 页面编号: 2124538444 打印人: 8685 2026-03-09 10:22:58

1.2.6 代码生成

代码生成（code generation）是指取源程序的一种中间表示作为输入并把它映射到一种目标语言。如果目标语言是机器代码，则需要为源程序所用的变量选择寄存器或内存单元，然后把中间指令序列翻译为完成同样任务的机器指令序列。此阶段的一个关键问题是寄存器分配。

例如，使用寄存器 R1 和 R2，中间代码（1.4）可以翻译成：

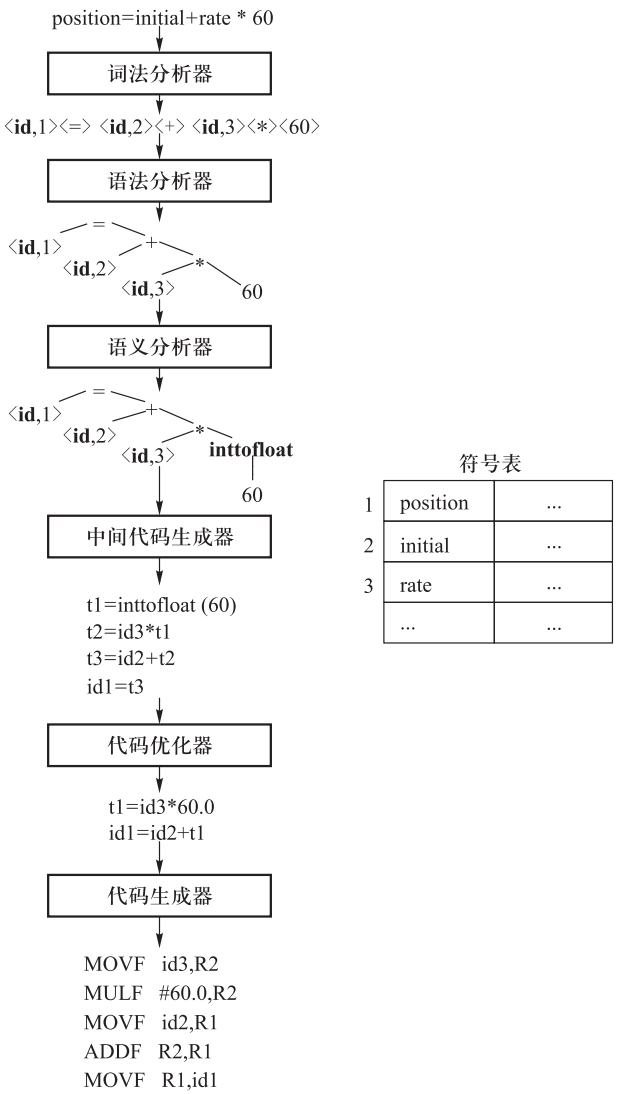


图 1.6 赋值语句（1.1）的编译过程

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第13页 文件编号: 422624 页面编号: 2134538442 打印人: 8685 2026-03-09 10:22:58


```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1                    (1.5)
ADDF R2, R1
MOVF R1, id1
```

每条指令的第一个操作数和第二个操作数分别代表源操作数和目的操作数，指令名中的“F”表示处理浮点数。代码（1.5）中第一条指令把地址 id3 的内容取到寄存器 R2（在此认为指令中 id3 代表对象 id3 的地址，变量的存储分配将在第 6 章讨论）。第二条指令将其乘上浮点数 60.0，# 号代表 60.0 作为立即数处理。第三条指令把 id2 取到寄存器 R1。第四条指令再把寄存器 R2 的值加到 R1。最后一条指令把寄存器 R1 的值存入地址 id1。这样，该段代码实现了语句（1.1）的赋值。代码生成将在第 8 章讨论。

1.2.7 符号表管理

编译器的一项重要工作是记录源程序中使用的变量名字，并收集每个名字的各种属性。这些属性提供与该名字有关的存储分配、类型和作用域等信息。如果是过程名字，还包含参数的个数、每个参数的类型、参数传递方式和返回值类型等属性。

符号表（symbol table）是为每个变量名字保存一个记录的数据结构，记录的域是该名字的属性。该数据结构应该设计成允许编译器迅速地找到一个名字的记录，并在此记录中迅速地存储和读取数据。符号表在第 7 章讨论。赋值语句（1.1）的编译过程如图 1.6 所示。

1.2.8 阶段的分组

上面的介绍把编译过程从逻辑上分成了 7 个阶段。最粗略的划分方式是把编译过程分成**分析**（analysis）和**综合**（synthesis）两大部分。分析部分揭示源程序的基本元素和它们所形成的层次结构，决定它们的含义，建立起源程序的中间表示，分析部分通常被称为**前端**（front end）。综合部分从源程序的中间表示建立起与源程序等价的目标程序，它通常被称为**后端**（back end）。

在实际的编译中，源于几个阶段的活动可以组合在一起，各阶段之间的中间表示也无须显式构造。源于几个阶段的活动常用**一遍**（pass）扫描来实现，即对一个输入（文件）在一遍扫描期间完成一项或多项活动，再输出。例如：前端的词法分析、语法分析、语义分析和中间代码生成可以组成一遍；独立于机器的代码优化可以单独作为可选择的一遍；而为特定目标机器的代码生成可以作为后端的一遍。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第14页 文件编号: 422624 页面编号: 2144538440 打印人: 8685 2026-03-09 10:22:58

取一个编译器前端，重写它的后端可以产生同一源语言在另一种机器上的编译器，已经是件普通的事情。如果原先的后端是经过仔细设计的，那么往往不需要对它做很多的重新设计。

把几种不同的语言编译成同一种中间表示，让不同的前端使用同一个后端，从而得到一种机器上不同源语言的编译器，已经有不少成功的例子。

1.3 编译器的作用及形式

在 1.1.1 节中，已经介绍了可以使用在线编译网站来编译和运行一些用高级语言编写的程序。读者也可以在本机安装编译工具链来编译和运行程序。主流的能编译 C、C++ 等语言程序的开源编译工具有 GNU 编译器套件 GCC 和 Clang/LLVM。本节结合实例说明编译器的作用及其形式。

1.3.1 编译器的作用

编译器的主要作用包括翻译、分析和优化代码。

翻译代码：是指将人类可读的高级编程语言代码翻译成低级机器语言代码，使程序能够在目标机器上执行。编译器不仅需要支持高层的编程抽象，还要能支持底层的硬件体系结构。

分析代码：是指利用编译技术分析程序，理解代码结构和逻辑（程序理解），帮助检测和修复错误以确保代码的正确性（safety），检测预防一些常见安全漏洞以提升程序的安全性（security）。

优化代码：是指通过各种优化技术，如代码内联、循环展开、常量折叠等，提高生成代码的执行效率和资源利用率，提升程序性能。优化目标可以是执行的时间或空间、能耗、代码行数等。

【例 1.2】(命令行编译示例) 在 Linux 下，用户可以用 GCC 编译图 1.2 (b) 中的 C 语言程序 gcd.c，然后执行所生成的 gcd。过程如下：

```
1 [root@host ~]# gcc gcd.c -o gcd
2 [root@host ~]# ./gcd
3 gcd(21,36)=3
```

注意，GCC 是一个驱动程序，而不是编译器。GNU 编译套件及编译流程如图 1.7 所示，它根据命令行参数选择调用编译套件中的预处理器、编译器、汇编器、链接器等工具中的部分或全部来完成编译任务。表 1.1 列出了 GCC 或 clang 主要的编译选项及含义。上面第 1 行的 gcc 命令将编译 gcd.c 文件，并生成一个名为 gcd 的可执行程序。第 2 行命令执行 gcd，输出结果见第 3 行。如果你要编译 C++ 程序（如 gcd.cpp），则需将 gcc 改成 g++。如果使用 Clang/LLVM 编译，则需要将 gcc 或 g++ 改为 clang 或 clang++。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第15页 文件编号: 422624 页面编号: 2154538447 打印人: 8685 2026-03-09 10:22:58

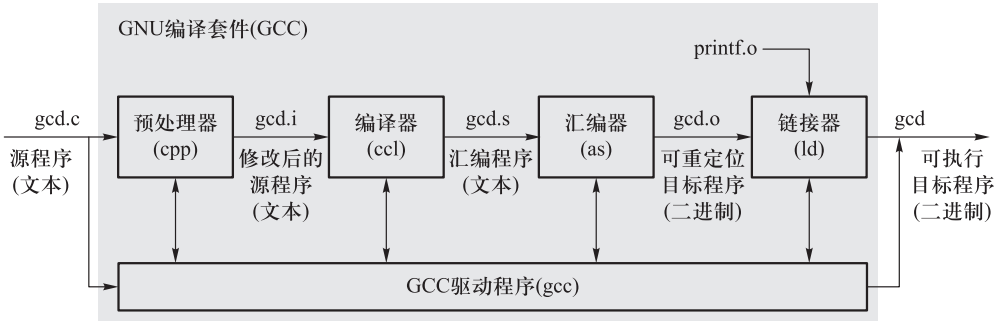


图 1.7 GCC 编译套件及编译流程

表 1.1 GCC 或 Clang 主要的编译选项、含义及示例

编译选项	含义	示例
-c	只编译或汇编源程序文件，不执行链接	gcc -c gcd.c
-o <file>	指定输出文件的名称	gcc gcd.c -o gcd
-E	只执行预处理	gcc -E gcd.c -o gcd.i
-S	完成编译但不执行汇编，产生汇编文件	gcc -S gcd.c
-m32, -m64, -m16	为 32 位、64 位或 16 位环境产生代码	gcc -m32 gcd.c -o
-I<path>	指定头文件搜索路径	gcc -I/path/to/include gcd.c
-L<path>	指定库文件搜索路径	gcc -L/path/to/lib gcd.c -lm
-l<lib>	链接指定的库	如-lm，会链接库 libm.so 或 libm.a
-O<level>	进行代码优化，等级从 0 到 3，编译时间变长，但生成的代码运行速度会更快 等级 s 表示优化生成较小的代码 等级 g 用于调试的优化	gcc -O0 gcd.c 无优化 (默认) gcc -O2 gcd.c 较高级优化 gcc -Os gcd.c gcc -Og gcd.c

【例 1.3】(程序安全) 对于下面的 foo 函数定义，调用 foo() 会产生怎样的结果？

```
1 foo (char * s) {
2     char buf[32];
3     strcpy (buf, s);
4 }
```

若传入的参数 s 指向的字符串的长度超出 31，则复制时会超出 buf 数组的有效区域，导致缓冲区溢出，进而可能引发难以预料的结果。在学习有关程序运行时存储空间的布局（第 6 章）之后，你将对为什么会产生这种难以预料的结果有更深入的认识。

【例 1.4】(程序性能) 下面两段代码中，哪段代码执行得更快，为什么？

```
1
2 for (i=0; i<n; i++)  a[i] = 1;
```

(a)

```
1 pend = a+n;
2 for (p=a; p<pend; p++) *p = 1;
```

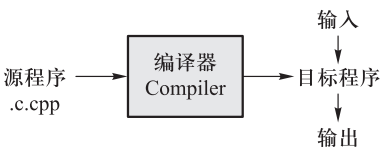
(b)

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第16页 文件编号: 422624 页面编号: 2164538445 打印人: 8685 2026-03-09 10:22:58

在编译器默认无优化时，代码 (b) 执行得更快。因为对于代码 (a)，在每次循环时都需要对数组元素 a[i] 进行地址计算；而对于代码 (b)，则只需要执行 p++ 来使 p 指向下一个元素。你可以检查用不同编译优化等级得到的结果。有关数组元素地址计算的翻译方法在第 7 章介绍，在学习和理解编译机理后，你将能解释产生这种差异的原因，由此指导你编写出更高性能的代码。

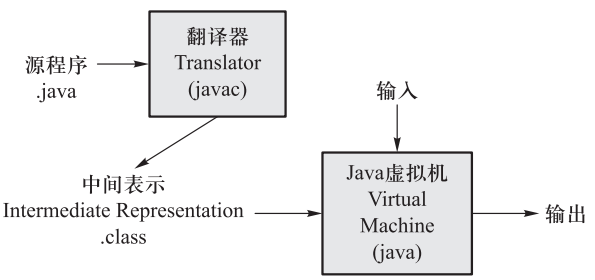
1.3.2 编译器的形式

图 1.8 展示了现代编程语言三种主要的编译和执行方式。



(a) C/C++ 编译器

(b) 解释器



(c) Java 编译器

图 1.8 编译器及解释器

如图 1.8 (a) 所示，对于以 C/C++ 为代表的编译型语言，编译器对输入的源程序（.c 或.cpp 文件）按类似图 1.4 的编译阶段翻译为目标程序（机器代码）。目标程序在执行时接受用户输入，输出程序执行结果。

还有一些编程语言采用如图 1.8 (b) 所示的**解释器**（interpreter）。解释器不同于编译器，它不会将整个源代码翻译为目标程序后再执行，而是逐行读取源程序（脚本文件）并立即执行相应的运算。解释器能够接受用户输入并输出程序执行结果。与编译器类似，解释器也需要对源程序进行词法分析、语法分析和语义分析等，这样它才有可能知道源程序指定了哪些

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第17页 文件编号: 422624 页面编号: 2174538443 打印人: 8685 2026-03-09 10:22:58

运算。

早期的高级语言如 Lisp（1958 年）和 BASIC（1964 年）等，它们的第一个实现都是解释器。这种实现方式允许开发者在命令行或 REPL（read-eval-print loop）环境中直接输入代码并立即看到结果，即支持交互式编程，极大地方便教学和快速原型开发。后来崛起的动态脚本语言如 Perl（1987 年）、Python（1991 年）、JavaScript（1995 年）最早实现的也是解释器。

解释执行的效率通常低于编译器生成的机器代码的执行效率。这是因为编译器在生成机器代码时，仅需对源程序进行一次词法分析、语法分析和语义分析；而解释执行则需要在每次执行到源程序的某个语句时，都进行这些分析，然后执行该语句指定的运算。这种反复分析源程序的过程显著降低了解释执行的效率。为了提高效率，解释器通常采用一种适合于解释的中间语言。从而解释器可分成编译和解释两个阶段：编译阶段将高级语言源程序翻译成中间表示形式，而解释阶段对中间表示进行解释执行。这两个阶段合在一个程序中。因此这个程序被统称为解释器。

进入 Java 语言（1995 年）时代，解释器的上述两个阶段被分成两个程序，如图 1.8 (c) 所示。前一个程序称为翻译器（javac），它将 Java 语言的源程序 (.java 文件) 翻译成字节码中间表示（.class 文件）。后一个程序称为 Java 虚拟机（Java Virtual Machine，JVM），通常用 java 命令启动），它负责加载和执行字节码中间表示，在执行期间接受用户输入，输出程序执行结果。

早期的 JVM 主要通过解释执行来运行 Java 字节码。随着技术的发展，为了提高执行效率，现代 JVM 引入**即时编译**（just-in-time compilation, JIT）技术，如 HotSpot、Graal、DRLVM。在 JIT 编译中，JVM （图 1.9）在运行时动态地将热点字节码编译为本地机器代码，并直接执行这些机器代码，从而显著提升了程序的执行性能。JIT 编译技术使得 Java 程序不仅能够在不同平台上保持良好的跨平台性，还能达到接近本地编译语言的执行速度。

JIT 编译虽然能动态优化代码并提高程序性能，但也存在启动时间较长、运行时性能波动、内存资源消耗大的不足。为适应嵌入式系统和移动应用对快速启动和低内存占用的需求，**预先编译器**或**提前编译器**（ahead-of-time compiler, AOT）被提出。这是一种在程序运行之前将源代码或中间代码编译成机器代码的工具。例如，Android 的 ART（2013 年）和 iOS 的 Swift 语言（2014 年）在应用安装时使用 AOT 编译字节码或 Swift 代码为本地机器码，从而加快应用启动速度。AOT 编译提高了程序的执行效率和稳定性，但可能会牺牲一些灵活性和动态优化能力。

增量编译器（incremental compiler）是一种提高编译效率的工具。它只编译程序中发生变化的部分，而不是每次都重新编译整个源代码。这种方法通过检测自上次编译以来的代码变化，专注于编译这些变化的部分，显著减少了编译时间，特别是在大型项目中更加凸显。增量编译器常用于在开发过程中加快迭代速度。常见的增量编译工具包括 Java 的 Eclipse JDT 编译器和 Android 构建工具 Gradle 中的增量编译功能、阿里巴巴开发的 Android 应用增量

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第18页 文件编号: 422624 页面编号: 2184538441 打印人: 8685 2026-03-09 10:22:58

编译工具 Freeline 等。

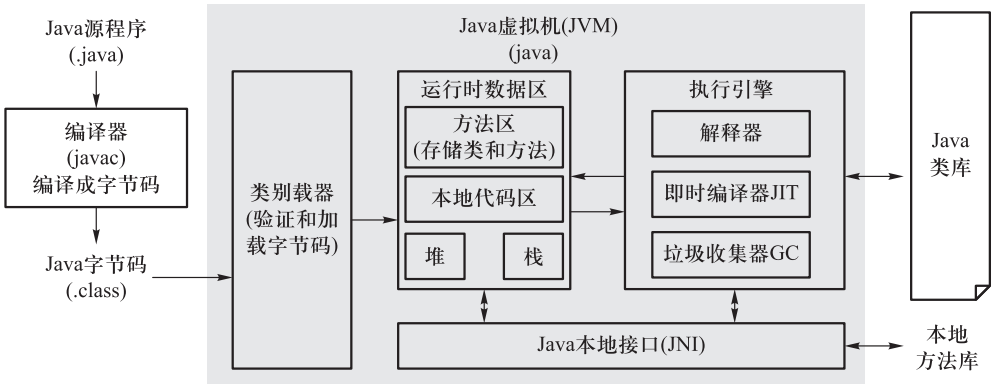


图 1.9 JVM 概览

自适应编译器（adaptive compiler）能根据运行时的性能数据动态调整编译策略，以进一步提升效率。它通过监控程序的实际执行情况，识别出热点代码，并根据这些数据优化编译过程，提高程序整体性能。HotSpot 和 GraalVM 等 JVM、JavaScript 引擎 V8 等都支持自适应编译。

交叉编译器（cross compiler）是一种将源代码编译成与当前开发平台不同的目标平台机器码的工具。它允许开发者在一种平台上编写代码，并将其编译为另一种平台上可执行的代码。这种编译器特别适用于开发嵌入式系统、移动设备或其他与开发环境不同的计算平台。例如，arm-linux-gcc -o gcd-arm gcd.c 将 gcd.c 编译为适用于 ARM 处理器的 gcd-arm。

1.4 主流的编译基础设施及发展

编译基础设施（compiler infrastructure）是指用于支持编译过程的工具、框架和技术集合，通常包括编译器前端、优化中间层、后端、工具链等组件。下面首先介绍支撑编译型语言的两大主流通用编译基础设施 GCC 和 LLVM，然后介绍 Java 和 Go 这两种富含运行时支持的现代编程语言的编译运行时系统，再简要介绍国内编译技术生态，最后概述深度学习、图查询、异构计算与量子计算 4 个新兴领域的编译技术发展。

1.4.1 通用编译基础设施：GCC 和 LLVM

1. GCC（GNU 编译套件）

GCC 由 Richard Stallman，于 1987 年首次发布，最初为 GNU 操作系统而开发，起初只能编译 C 语言。GCC 目前由自由软件基金会（Free Software Foundation，FSF）维护，能支持

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第19页 文件编号: 422624 页面编号: 2194538449 打印人: 8685 2026-03-09 10:22:58

多种编程语言（如 C、C++、Objective-C、FORTRAN、Ada 和 Go）、多种硬件架构（如 x86、ARM、PowerPC、MIPS、RISC-V）和多种操作系统（能在 Windows、Linux、macOS 和其他类 UNIX 系统上运行）。GCC 以其强大的功能、广泛的支持和高效的优化能力而闻名，被广泛用于操作系统、嵌入式系统、应用软件和开源项目的开发。2014 年 GCC 获 ACM SIGPLAN 编程语言软件奖。2015 年 Richard Stallman 因领导开发 GCC 获 ACM 软件系统奖。

GCC 遵循 GNU General Public License（GPL）许可，其当前版本的代码大多遵循 GPL v3 许可，但一些组件可能使用 GPL v2 许可。GPL v2 强调软件自由和源代码的可用性，但对专利和 Tivoization（例如制造商提供了源代码但却通过硬件限制不允许自由修改这个软件）问题关注较少；而 GPL v3 增加了对专利侵权的保护，防止 Tivoization，确保用户对软件的自由控制权。

2. LLVM

LLVM 由 Chris Lattner 于 2000 年在美国伊利诺伊大学香槟分校（UIUC）创建，最初作为一个静态编译器中间表示和优化框架。随着社区的发展，它已演变为一个全面的开源编译基础设施。如图 1.10 所示，它通过不同的前端支持多种编程语言（如 C、C++、Objective-C、Go、Swift、Rust、Haskell、Julia），在语言无关、平台无关的中间表示 LLVM IR 上开展丰富的中端优化，再由后端生成多种硬件平台（如 x86、ARM、PowerPC、MIPS、RISC-V）的目标代码。自 2007 年起，LLVM 每年举办 1~2 次开发者大会。2012 年，LLVM 研发团队获 ACM 软件系统奖。LLVM 的模块化设计、高级优化技术和广泛的应用领域（包括编译器开发、语言研究、高性能计算和嵌入式系统等）使其成为现代编译技术的重要基石。

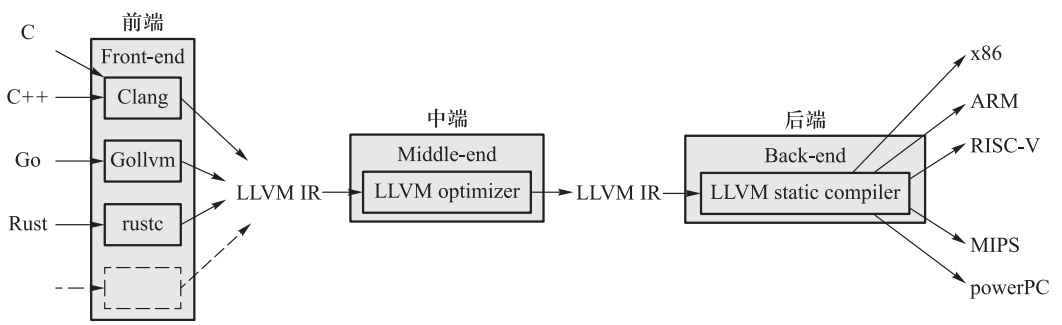


图 1.10 LLVM 编译基础设施

LLVM 遵循 Apache License 2.0 许可证，允许用户自由使用、修改和分发软件，无须支付费用。用户可以发布修改后的代码，无论是开源还是闭源，但必须保留原始许可声明、版权声明和免责声明。该许可证还提供广泛的专利授权，防止专利侵权诉讼。其宽松性和明确的专利条款使得 LLVM 成为一个非常开放和灵活的编译基础设施，广泛应用于开源和商业项目中。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第20页 文件编号: 422624 页面编号: 2204538447 打印人: 8685 2026-03-09 10:22:58

1.4.2 具有代表性的现代编程语言的编译运行时系统

这里选择 Java 和 Go 这两种富含运行时和并发能力的语言进行介绍。Java 通过 JVM 实现跨平台和自动垃圾回收，支持即时编译优化，适合企业级应用。Go 通过轻量级 goroutines 实现高效并发，内置垃圾回收和快速编译，适合云计算和分布式系统。两者在各自领域表现出色。

1. Java 虚拟机 (JVM)

JVM 由 Sun Microsystems 公司（现为 Oracle 公司的一部分）于 1995 年推出，其核心理念是“一次编写，各处运行”（write once, run anywhere）。JVM 为编译成 Java 字节码的编程语言（还包括 Kotlin、Scala、Groovy、Clojure 等）提供一个抽象的执行环境，使得 Java 字节码可以在任何支持 JVM 的平台上运行而无需修改代码。JVM 的核心组件参见图 1.9，包括类加载器、运行时数据区和执行引擎，通过垃圾收集器（GC）自动管理内存，提升内存安全性；通过 Java 本地接口（JNI）复用本地代码资源，避免重复开发，并保证跨平台性和兼容性。JVM 技术对现代编程语言和平台具有重要影响，推动了跨平台计算和动态编译技术的发展。JVM 的主要开源实现包括 OpenJDK（2007 年发布 OpenJDK 6，由 OpenJDK 项目委员会管理）、Eclipse OpenJ9（2017 年开源，源自 IBM 的 J9 JVM）和 GraalVM（2018 年发布，由 Oracle 公司开发，支持多种编程语言）。

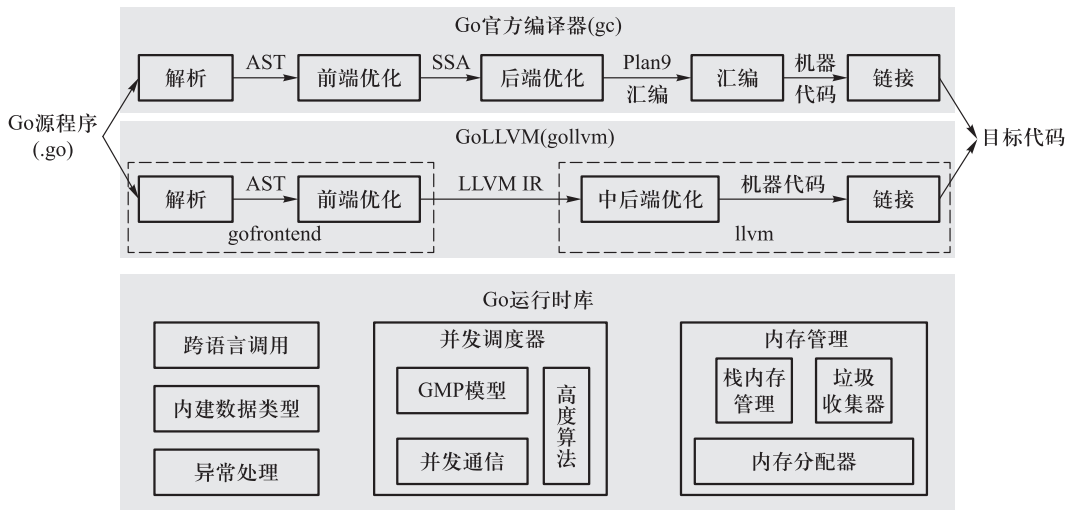


图 1.11 Go 编译运行系统概览

OpenJDK 遵循 GPL v2 许可证，并附加 Classpath Exception，允许将 GPL 代码与非 GPL 代码链接，减少对其他开源许可的限制。Eclipse OpenJ9 遵循 Eclipse Public License (EPL) v2.0 许可证，支持自由使用、修改和分发，并允许与其他开源或专有软件集成。GraalVM 的

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第21页 文件编号: 422624 页面编号: 2214538445 打印人: 8685 2026-03-09 10:22:58

商业版本遵循 Oracle Binary Code License Agreement，而开源组件遵循 Apache License 2.0 或 GPL v2。

2. Go 编译器

Go（又称 Golang）语言由 Google 公司于 2009 年发布，以其简洁语法、自动内存管理、快速编译、高并发支持和强大标准库而闻名，在容器、云原生数据库、微服务等新兴领域得到广泛使用。Go 语言的设计结合了静态语言的安全性和高效性与动态语言的开发便利性，能够快速生成高效的二进制文件，并内置垃圾收集器进行内存管理。如图 1.11 所示，Go 语言提供丰富的运行时库，支持高效轻量的并发机制（如 goroutines 和 channels）以及丰富的复合数据类型（如 slice、map）。其核心组件包括 goroutine 调度器、内存管理、垃圾收集、异常处理（如 defer 语句）和调试 API，为 Go 程序提供了高效、可靠、安全和易用的运行环境。图 1.11 还展示了当前 Go 语言的编译路线：Go 官方编译器（gc）在 AST 上优化，并转换为静态单赋值形式（SSA），随后进行后端优化和目标代码生成。但为了实现快速编译，gc 编译器的优化有限；为了复用 LLVM 的优化能力和多平台代码生成能力，GoLLVM（自 2018 年起开发）将 Go 代码编译为 LLVM IR，增强了 Go 在不同硬件和操作系统上的性能。

Go 语言的开源许可为 Go 语言许可，基于 BSD 许可，但含有自定义条款。它允许自由使用、修改和分发 Go 的源代码，适用于商业和非商业项目。GoLLVM 使用 MIT License 许可，允许对代码进行自由使用、修改和分发，同时对源代码的使用和再分发有一些简单的要求。

1.4.3 国内编译技术生态

当前，国内外在编译技术生态方面存在显著差距。我国在应用软件创新上发展迅速，但在编程语言和编译器等基础软件技术上仍有短板。缺乏自主可控的编程语言和编译器技术，将使我国在全球软件产业链中处于被动地位。

在我国，算法和应用因其直接影响实际效果而长期受到关注，而底层芯片设计因美国制裁近年备受重视。然而，系统软件是确保芯片高效利用和应用软件开发运行的关键，是信息技术生态的基础。推动系统软件发展需要更多研究投入和应用支持，以全面提升技术生态的竞争力。

国产 CPU 处理器的发展始于 21 世纪初，主要包括龙芯（2000 年启动）、飞腾（1999 年启动）、申威（2003 年启动）和海思（成立于 2004 年）等。它们在技术性能上取得了显著提升，并逐渐在国内市场中获得应用。相比之下，国产 GPU 和 FPGA 技术仍处在发展初期。近年来，国产 AI 处理器迅速崛起，主要公司和产品包括寒武纪（2016 年成立）、华为昇腾（2018 年发布）、比特大陆（2013 年成立）和地平线（2015 年成立）。这些芯片的应用依赖于编程语言和编译器的支持。针对国产 CPU 处理器的编译系统，早期主要基于 GCC 进行研发，近年来由于 LLVM 的灵活性和扩展性，越来越多的国产芯片开始转向基于 LLVM 的编译系统。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第22页 文件编号: 422624 页面编号: 2224538443 打印人: 8685 2026-03-09 10:22:58

龙芯团队从 2000 年开始研制通用 CPU，2002 年研制成功我国第一款通用 CPU 龙芯 1 号，并在后续发展中形成了面向特定领域的龙芯 1 号 MCU 系列、面向工控领域的龙芯 2 号 SoC 系列和面向桌面服务器的龙芯 3 号系列。早期的龙芯处理器采用 MIPS 指令集架构。2021 年龙芯中科技术股份有限公司发布自主指令集架构“龙架构”（LoongArch），目前龙芯三大系列处理器均已全面切换为龙架构指令集。龙架构兼容主流计算平台，提供更高的多核性能和自主控制能力。龙芯致力于打造基于龙架构的独立产业生态，向 GCC、LLVM、.NET、Golang、Rust 和 OpenJDK 等主流编译器开源社区贡献了数十万行代码，并实现了绝大部分开源软件对龙架构的支持。GCC 和 LLVM 开源社区分别于 2022 年 5 月和 9 月在 GCC 12.1 版本和 LLVM 15 版本初步支持龙架构，可以通过标准选项“-march=”和“mtune=”为龙架构提供结构级和微结构级编译支持。2024 年发布的 GCC14 和 LLVM18 进一步支持龙架构的 LSX、LSAX 向量指令和自动向量化。.NET、Golang 和 Rust 社区分别于 2022 年 4 月、2022 年 8 月和 2023 年 7 月正式加入对龙架构的支持，进一步丰富了龙架构的软件生态。此外，龙芯还基于 OpenJDK 开发并发布了龙芯平台 Java 环境，针对市场需要推出了数学库、人工智能算子库等一系列基础库的龙架构优化版本。基于龙架构的诸多产品已经实现大规模商业化应用，龙架构生态日益繁荣。

申威团队自 2003 年起研发国产高性能处理器。2006 年在 Alpha 指令集基础上自主设计出“申威 1”单核 CPU，2008 年实现从单核到多核的技术革新，2010 年发布了“申威 1600”16 核 CPU，形成可自主发展的申威 64 指令集系统。申威系列 CPU 涵盖服务器（如 2020 年 32 核“威鑫 3231”）、终端（如 2021 年 8 核“威焱 831”）以及嵌入式系统，能满足服务器、桌面终端、嵌入式等应用场景。在基础软件生态方面，申威团队自“申威 1”起研发配套的高性能编译器。2010 年先后依托 GCC 和 OpenJDK，推出了申威 GCC 编译器和申威 Java 虚拟机。此后，申威团队持续深耕，构建基础编译器、语言虚拟机、基础库三大核心支柱体系。在基础编译器领域，持续优化更新 GCC 编译器，并于 2018 年支持 Golang 和 LLVM 编译器，2021 年支持 Rust 语言编译器。在语言虚拟机领域，在保持对 Java 虚拟机深度优化与更新的同时，于 2014 年支持 V8、SpiderMonkey 虚拟机，2022 年支持.NET 运行框架，极大丰富了申威平台的运行时支撑能力。在基础库领域，团队近年先后推出高性能基础数学库、高性能扩展数学库以及安全加解密库等一系列关键基础库，显著提升了应用程序的执行效率和安全性。如今，申威已构建起涵盖数十款编译器产品、支持主流及新兴编程语言的基础软件生态体系，形成了丰富多元且高效协同的软件生态格局。

华为编译器团队于 2009 年成立，起先主要面向无线的数字信号处理器（digital signal processing，DSP）研制编译器并在 2013 年规模商用；之后自研 ARM C/C++ 编译器，于 2016 年实现 ARM64 领域竞争力，并在华为公司多个场景商用。2019 年起基于 LLVM 研制面向通用处理器架构的高性能、高可信、易扩展的编译器工具链——毕昇编译器（图 1.12）。它支持多种编程语言，包括 C/C++ 和 FORTRAN，并兼容 OpenMP 4.0/4.5/5.0 API 标准；引入多种编译优化技术，致力于提升编译后的代码性能、可信性和可扩展性。特别地，毕昇

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第23页 文件编号: 422624 页面编号: 2234538441 打印人: 8685 2026-03-09 10:22:58

编译器对鲲鹏架构进行了优化，通过软硬件协同，显著提升了性能。例如，与开源 LLVM 相比，毕昇编译器在 CPU 基准测试中的性能提高了 30%以上，有效释放了鲲鹏芯片的强大算力。毕昇编译器首次发布于 2020 年 9 月的华为全联接大会，目前已经推出多个版本，如 1.0、1.3.1、2.5.0、3.2.0 和 4.0.0 等。其首个发行版本即支持自动向量化、循环优化、浮点优化等特性，集成了针对 ARM 处理器优化的数学库、减少内存碎片和提高并发性能的 jemalloc 动态库，还提供了自动调优工具 Autotuner。后续版本陆续增加了对 Aarch64 版本 bolt 组件的支持、鲲鹏 KML neon 版本数学库的集成等。此外，毕昇编译器强化了对 FORTRAN 的支持，包括对 4 倍精度的计算、最大 15 维数组、type-bound、value attribute 等 FORTRAN 2008 语言特性的支持，并改进了 FORTRAN 代码的别名分析能力。

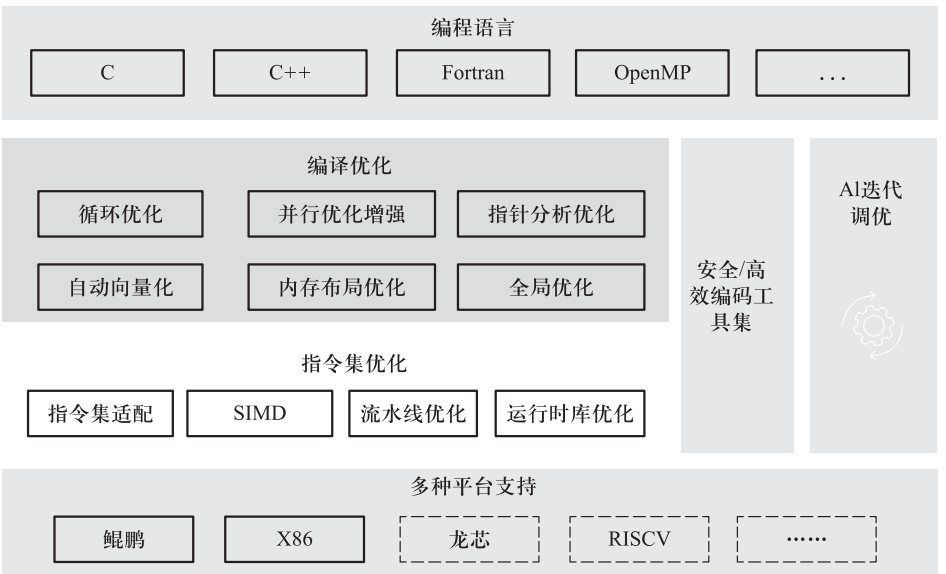


图 1.12 基于 LLVM 扩展中端和后端优化的毕昇编译器

1.4.4 新兴计算领域的编程系统

本小节简要介绍智能计算、图计算、异构计算、量子计算四个新兴领域的编程系统发展。

1. 人工智能编译基础设施

随着 2012 年 AlexNet 在 ImageNet 竞赛中的突破、2020 年 1 750 亿参数的大语言模型（large language model，LLM）GPT-3 的发布，以及 2022 年 ChatGPT 的推出，人工智能（artificial intelligence，AI）模型在精度和应用范围上取得了显著进展，同时对计算和优化提出了更高需求。AI 编译旨在将 AI 模型和算法转化为高效计算代码以优化在特定硬件上的执

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第24页 文件编号: 422624 页面编号: 2244538449 打印人: 8685 2026-03-09 10:22:58

行性能。早期，性能优化主要依靠手动调整计算图。后来，Google 的 TensorFlow XLA（2017 年）、Facebook 的 TorchScript 和 PyTorch Glow（2018 年）、TVM（2018 年）等专用编译器和优化工具自动优化计算图并生成高效的本地代码，提升了模型在多种硬件平台上的执行效率。为促进兼容性，2017 年提出了 ONNX 模型交换格式。ONNX Runtime 用于优化和加速 ONNX 模型的推理。2019 年发布的 MLIR（Multi-Level Intermediate Representation）提供灵活的表示层次结构，支持多种计算模型和硬件优化。NVIDIA 的 TensorRT（2017 年）也提供高性能的模型推理优化库。

国内的 AI 编译基础设施也在快速发展。华为于 2020 年推出面向全场景 AI 计算的 MindSpore；百度于 2016 年开源 PaddlePaddle；阿里巴巴 2019 年发布专注于移动设备上高效推理的 MNN；旷视科技于 2020 年发布具备高性能和灵活性的 MegEngine。

XLA、TVM、MLIR、Glow、MindSpore、PaddlePaddle、MegEngine 都遵循 Apache License 2.0 许可。ONNX 和 ONNX Runtime、MNN 遵循 MIT License 许可，允许用户自由使用、复制、修改、合并、出版、分发、再许可和销售软件的副本。TensorRT 是 NVIDIA 提供的商用软件，使用时需要遵循 NVIDIA 的商用许可协议。

2. 图查询语言和编译

图查询语言（graphquery langnage，GQL）和编译技术在处理复杂图数据中扮演了关键角色。自 2000 年代初期，图查询语言如 Neo4j 的 Cypher（2011 年）、Apache TinkerPop 的 Gremlin（2009 年）和 SPARQL（2008 年成为 W3C 推荐标准）提供了强大的查询和操作功能。这些语言支持复杂的图遍历和模式匹配，使得图数据库能够高效处理和分析数据。图编译技术将这些查询语言转换为优化的执行计划，提升查询性能。现代图数据库系统如 Neo4j、JanusGraph（2012 年）和 Amazon Neptune（2018 年）采用查询优化、执行计划生成和并行处理等编译优化策略，以应对大规模图数据的挑战。为统一图查询语言，GQL 于 2019 年成为 ISO/IEC 的国际标准项目，该标准于 2024 年发布。GQL 结合 Cypher、Gremlin 和 SPARQL 的特性，旨在提供更强大和灵活的查询能力，促进跨平台和跨系统的互操作性。GQL 的编译和优化有待发展。

国内蚂蚁集团于 2017 年开始研发支持 OpenCypher 的图查询处理器 TuGraph，并于 2022 年正式开源。Neo4j 的 Cypher 遵循 GPLv3 许可证，Gremlin 和 JanusGraph 遵循 Apache 2.0 许可证。SPARQL 是 W3C 推荐标准，无具体软件许可证；Amazon Neptune 是商用软件许可证。

3. 异构计算编程模型及编译

随着数据量的增加和应用需求的多样化，单一处理器（如 CPU）难以满足高性能和高效能的要求。异构计算通过结合不同处理器（如 CPU、GPU、FPGA）的特点，实现优势互补，从而提高计算效率和性能。这一趋势在图形处理、科学计算、机器学习等领域尤为显著，推动了新的编程模型和编译技术的快速发展。CUDA（compute unified device architecture，统一计算设备架构）由 NVIDIA 公司于 2007 年发布，是一个面向 NVIDIA GPU 的闭源并行计算

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第25页 文件编号: 422624 页面编号: 2254538446 打印人: 8685 2026-03-09 10:22:58

平台和编程模型，广为使用。OpenCL（open computing language）协会和 SYCL 由 Khronos Group 协会于 2008 年和 2012 年推出。OpenCL 提供低层次的跨平台编程模型，SYCL 则是一种允许在单一 C++ 源文件中编写主机和设备代码的高级异构编程模型。Intel 于 2019 年推出的 oneAPI 通过统一编程模型简化多种硬件架构的开发，其核心 DPC++（Data Parallel C++）基于 SYCL 扩展，支持异构计算，并兼容包括 Intel、AMD 和 NVIDIA 在内的多种硬件平台。

CUDA 软件开发包（software development kit，SDK）由 NVIDIA 公司提供，免费使用，但受 NVIDIA 自有许可约束，限制了某些商业和分发用途。OpenCL 和 SYCL 均是由 Khronos Group 管理的开放标准，任何厂商都可以实现和使用，具体实现可能有不同的开源许可。oneAPI 的工具和库有不同的许可协议。DPC++ 遵循 Apache 2.0 许可证，而其他组件如 Intel oneAPI 工具包可能有 Intel 自有许可证。

4. 量子计算及其编程语言与编译

量子计算机利用量子力学原理，通过量子比特（qubit）的叠加和纠缠特性进行计算，能在某些计算问题上显著提高效率。量子计算的理论由 Richard Feynman 和 David Deutsch 等人在 20 世纪 80 年代提出，并在 20 世纪 90 年代逐渐成型，21 世纪初的实验验证奠定了基础。2010 年代 IBM 和 Google 等公司推出了超导量子比特处理器，逐步提高了规模和稳定性。目前，探索中的量子计算机体系包括离子阱（如 IonQ 和 HoneyWell）、中性原子（如 QuTech 和 Atom Computing）、拓扑量子比特（Microsoft）、光量子比特（如 Xanandu、PsiQuantum）等，各具特色。量子编程语言的开发始于 21 世纪初期，目前代表性的量子编程语言或框架有 IBM 的 Qiskit（2017 年）、Google 的 Cirq（2018 年）、剑桥量子的 tket（2018 年）、Microsoft 的 Q#（2019 年）、国内本源量子的 QPanda（2019 年）等，为量子算法开发提供了工具。2019 年，Google 的 Sycamore 处理器实现了量子优越性，是重要的里程碑。尽管面临稳定性和纠错等挑战，量子计算在密码学、化学、优化问题和机器学习等领域展现了显著潜力。与传统计算相比，量子计算的编程语言及编译技术发展尚显初级，有更为广阔的发展空间。

Qiskit、Cirq、tket、QPanda 遵循 Apache 2.0 许可证。Q# 使用 Microsoft 的自有许可，这通常允许自由使用和分发，具体使用条款需要查看 Microsoft 的官方网站或许可证文档。

1.5 编译技术的发展与应用

编程语言及编译器设计的从业人员较少，因此对于大多数生来说，学习编译技术对学生的广泛意义在加深对程序从源代码到机器码转化过程的理解，提升编程、优化和调试能力，同时强化逻辑思维和问题解决能力。此外，编译器的设计还影响着计算机科学的其他一些领域。本节回顾编译器设计与计算机科学其他主要领域的相互影响和编译技术的重要应用。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第26页 文件编号: 422624 页面编号: 2264538444 打印人: 8685 2026-03-09 10:22:58

1.5.1 高级语言演变与实现

高级编程语言定义了一种编程抽象：允许开发者用这种语言表达算法，再由编译器翻译为目标代码。通常，高级语言易于编程，但直接翻译得到的程序运行较慢。低级语言则允许更细粒度的控制，可以得到更高效的代码，但难编写、易出错，且难以维护和移植。编译优化通过各种技术改进所生成代码的性能，弥补高级编程抽象带来的低效。

历史上，流行编程语言的演变多集中在提升抽象级别上。从 20 世纪 80 年代 C 语言主导系统编程，到 90 年代 C++ 语言的兴起，再到 1995 年 Java 语言的问世并迅速普及，每一轮编程语言的新特性都推动了编译器优化技术的发展。下面简要回顾促进编译技术显著发展的主要语言特征。

所有通用编程语言（如 C、FORTRAN）都支持用户自定义聚合数据类型（如数组和记录）及高级控制流（如循环和过程调用）。为提升所生成代码的性能，使其接近有经验开发者用低级语言编写的代码，编译器必须进行数据流分析和优化。这些程序分析和优化技术至今仍被广泛研究。

面向对象的概念首次出现在 1967 年的 Simula67 语言中，随后被 Smalltalk、C++、C# 和 Java 等语言采用。其核心理念是数据抽象和继承，使程序更加模块化、易于复用。面向对象程序的类定义中可能包含许多小过程（方法），编译器需要优化跨方法操作的效率。基于此，用过程体替代过程调用的内联（inlining）和虚方法调度（dispatch）的优化技术尤为重要。

Java 语言的类型安全、自动数组边界检查和垃圾回收等机制简化了编程，但增加了运行开销。为减少这些开销，编译优化技术，（如删除不必要的边界检查、通过逃逸分析识别短生命期对象并将其分配到栈上等）应运而生。极小化垃圾收集开销的算法也在不断改进。此外，Java 通过字节码支持代码移植和移动，字节码需被解释或动态编译为本地代码。动态编译在运行时生成优化代码，常用技术是仅编译和优化经常执行的部分，以降低编译和执行开销。

Go 语言兼具编译型语言的高效性和自动垃圾回收的安全性。其并发构件（goroutine 和 channel）、高级数据结构（如 slice 和 map）以及异常机制（defer）等特性依靠强大的运行时库支持。Go 编译优化必须权衡优化代价和系统轻量化，以确保高效、可靠的编译和运行。

Rust 语言引入所有权和借用特性，回归显式内存管理并确保内存安全和高性能。Rust 编译器通过前端的所有权和借用检查分析避免内存错误，并在中端和后端使用 LLVM 进行优化。

1.5.2 针对计算机体系结构的优化

计算机体系结构的迅速演化带来对新编译技术的需求。高性能系统主要利用并行化和内存分层技术。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第27页 文件编号: 422624 页面编号: 2274538442 打印人: 8685 2026-03-09 10:22:58

并行性可以在指令级和处理器级挖掘。现代微处理器已经普遍实现了指令级并行，这种并行对程序员是透明的，意味着程序员编写的串行代码可以被硬件动态检查指令间的相关性，并尽可能并行发送这些指令。在某些体系结构中，如超长指令字（very long instruction word，VLIW）机器（如 Intel 的 IA64），指令集显式支持并行操作，使得一个时钟周期内可执行多条指令。所有高性能通用微处理器也通常包括支持数据并行处理的向量化指令。为了提高并行效率，硬件调度器或编译器可以重新排序指令、识别适合向量化的代码段并将其转换为向量化指令等，进一步发掘指令级的并行性。

多处理器系统的普及进一步推动了并行计算的发展。程序员可以为多处理器编写多线程代码，以充分利用多个处理器的计算能力。同时，编译器也需要能够自动从传统的串行程序生成并行代码。这样的编译器需要能够自动分析程序的并行性，在不同处理器之间合理分布计算任务，并有效管理处理器间的同步与通信，从而使程序的并行执行更为高效。这对于科学计算和工程应用等计算密集型领域尤为重要，能够显著提升运算效率。

内存分层是指内存体系由速度和容量不同的多层存储器组成，并且最靠近处理器的那一层速度最快、容量最小。如果一个程序的大部分访问都落在较快的内存层次上，那么它的平均内存访问时间就会缩短。

内存分层普遍存在于所有计算机中。它通常包括几百字节的少数几个寄存器、几千字节到几百万字节的多级高速缓存、几百万字节到几十亿字节的物理存储器，以及更大的二级存储。相邻层间的访问速度差异可达 2~3 个数量级。系统的性能往往受限于内存子系统，而非处理器速度。编译器不仅应优化处理器执行，还应有效利用内存分层，以减少内存访问延迟，从而提升程序的运行效率。

在优化程序时，寄存器的有效使用至关重要。编译器可以直接控制数据何时存储在寄存器中。相比之下，缓存和物理内存通常由硬件自动管理，对指令集透明。虽然硬件缓存策略通常有效，但是在处理大数据结构（如科学计算中的大型数组）时并不总是理想的。通过调整数据布局、指令访问数据的次序或代码布局，可以提升内存体系的运行效率。

1.5.3 新计算机体系结构的设计

在早期计算机体系结构设计中，编译器的开发在计算机造出后进行。如今，计算机系统的性能不仅取决于其原始速度（raw speed），还依赖编译器生成的代码效率。因此，现代计算机体系结构研究在处理器设计阶段就开始开发编译器，并通过模拟器运行代码来评估体系结构性能。

编译技术影响计算机体系结构设计的一个著名例子是精简指令集计算机（reduced instruction set computer，RISC）。在 David Patterson 和 David Ditzel 于 1981 年提出 RISC 设计方法之前，复杂指令集计算机（complex instruction set computing，CISC）倾向于将复杂性转移到硬件中，开发许多专用指令来简化汇编编程。例如，CISC 指令集包含支持不同内存

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第28页 文件编号: 422624 页面编号: 2284538440 打印人: 8685 2026-03-09 10:22:58

寻址模式的多种存储指令。编译器通过删除复杂指令序列中的冗余，将其精简为较少的简单操作，使得 RISC 成为可能。这样，编译器能有效利用简化指令，硬件也更易优化。

许多通用处理器体系结构，包括 PowerPC、SPARC、MIPS、Alpha 和 PA-RISC，都是基于 RISC 概念的。尽管流行的 x86 体系结构采用 CISC 指令集，但许多 RISC 思想也应用于这类处理器的实现上。高效使用 x86 机器的最佳方式是尽量使用其简单指令。

过去 50 多年提出了很多新的体系结构概念，包括 1970 年代的数据流机和向量机，1980 年代的超长指令字（VLIW）、单指令多数据（single instruction multiple data，SIMD）和脉动阵列，1990 年代的多处理器，2000 年代至今的多核处理器和 GPU，2010 年代至今的加速器架构（如 TPU、FPGA）、异构计算架构、RISC-V 开源指令集架构和量子计算机等。相应的编译器研究和开发都伴随着这些新的体系结构概念的发展而发展。

通用处理器和专用处理器在体系结构上的发展表现出不同的趋势。编译器技术不仅需要支持在这些多样化体系结构上的编程，还需要能对拟采用体系结构的设计进行评估。

1.5.4 程序翻译

编译器是从高级语言到低级语言的翻译器，同样的技术可用于不同种类语言之间的翻译。下面是程序翻译技术的两个重要应用。

1. 二进制翻译

编译技术可用于把一种机器指令集代码翻译成另一种机器指令集代码，以在不同硬件平台上运行原先为其他指令集编译的代码。许多计算机公司通过这种方式增强其机器上的软件可用性。由于 x86 在个人计算机市场上的主导地位，大部分软件都是为 x86 架构开发的。为了在其他架构上运行这些软件，早期开发了将 x86 代码转换为 Alpha 和 SPARC 代码的二进制翻译器。2020 年，Apple 公司发布了 Rosetta 2，它能在基于 Apple Silicon（ARM 架构）的 Mac 上后台自动将 x86 代码翻译为 ARM 代码，使用户能无缝运行 x86 应用程序。二进制翻译也可用于反向兼容性。2021 年，Windows 10 on ARM 引入了 x64 仿真功能，使 ARM64 设备能够运行为 x64 架构编写的软件。

2. 数据库查询解释器

数据库查询语言（如结构化查询语言 SQL）由一些谓词组成，这些谓词由包括关系运算在内的布尔表达式组成。查询解释器可以直接解释执行这些表达式，也可以将它们编译成查询计划。查询计划由查询优化器生成，描述了如何从数据库中检索所需数据，并通过选择最佳执行路径来最小化查询的执行时间和资源消耗。

1.5.5 提高软件开发效率的工具

程序是最复杂的人工制品，要求每个细节都准确无误。测试是定位程序错误的基本技术。除测试之外，数据流分析作为静态检测技术，能够发现所有可能执行路径上的错误，而非仅

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第29页 文件编号: 422624 页面编号: 2294538448 打印人: 8685 2026-03-09 10:22:58

依赖测试数据集所触发的错误。尽管是否能找出程序中所有错误的问题是不可判定的，数据流分析工具可以帮助发现潜在的安全漏洞或程序错误。然而，这些工具通常不可靠也不完备，即它们不能保证报告给程序员的都是真正的错误，并且也发现不了程序中所有的错误。尽管如此，各种静态分析工具仍然层出不穷并能在实际查找错误时发挥作用，如检测空指针或已释放指针的脱引用（dereferencing）问题。与编译器优化不同，错误探测工具可能产生虚假警告。

源于编译器中代码优化技术的程序分析方法有助于提高软件开发效率。下面从 3 个方面介绍静态发现程序潜在安全漏洞的技术。

1. 类型检查

类型检查是一种捕捉程序中类型不一致性的成熟且有效的技术。例如，它能检测运算对象的类型是否符合运算要求，以及实在参数和形式参数是否类型匹配。此外，通过分析程序的数据流，程序分析还能发现除类型错误以外的其他错误，如对 null 指针的脱引用。

类型检查技术可用于捕捉多种安全漏洞。例如，攻击者可能利用程序中某些字符串发起攻击。为应对这种风险，程序分析可以将用户提供的字符串加类型标记“危险”。如果程序未对这些字符串进行格式检查，它们将保持“危险”标记，从而防止在控制流中引发潜在的安全缺陷。

2. 边界检查

用较低级语言（如 C 语言）编程时容易出错。例如，C 语言不检查数组越界，可能导致缓冲区溢出，从而破坏或误用缓冲区以外的用户数据。攻击者可以利用这些漏洞，通过恶意数据使程序行为异常，威胁系统安全。尽管已有许多检测技术，但检测的成功率仍然较低。

删除冗余边界检查的数据流分析技术可以帮助定位缓冲区溢出问题。虽然简单的边界检查技术可能导致无法删除冗余边界检查而增加运行开销，但是其难以有效定位潜在的缓冲区溢出问题而影响系统安全。为实现高质量的错误探测，需采用更复杂的分析技术，如跨过程的指针值跟踪。

3. 内存管理

垃圾收集等自动内存管理技术可以消除内存泄漏等问题，这些问题在 C 和 C++ 程序中较为常见。Purify 等工具能够动态捕获程序运行时出现的内存管理错误，还有一些静态分析工具可用于预先识别这些错误，从而提高程序的可靠性和效率。

习　题

1.1　解释下列名词：源语言、目标语言、翻译器、编译器、解释器。

1.2　典型的编译器可以划分成几个主要的逻辑阶段？各阶段的主要功能是什么？

1.3　使用 GCC 或 Clang 编译 C 程序，理解并运用各种编译选项，学习编写 Makefile 并利用它编译 C 程序。请查阅相关“在线实验资源”网址获取任务分解及参考说明。

第 2 章

词 法 分 析

词法分析是编译的第一阶段，负责扫描源程序对应的字符流，产生用于语法分析的词法记号序列。图 2.1 展示了一种常见的实现方式，即将词法分析器作为语法分析器的子程序。当语法分析器请求“取下一个记号”时，词法分析器扫描剩余输入字符流，直到识别出下一个词法记号并输出给语法分析器。词法分析器会将识别出的标识符等信息记录到符号表中。

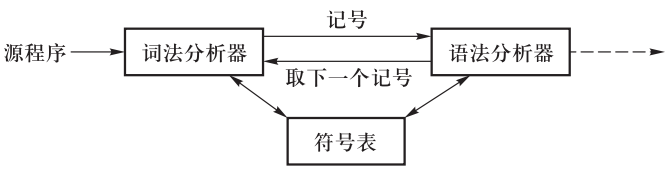


图 2.1 词法分析器和语法分析器的相互作用示意图

构造词法分析器的一种简单办法是用状态转换图来描述源语言词法记号的结构，然后手动将其翻译成识别词法记号的程序，从而生成词法分析器。本章重点围绕词法分析器的自动生成展开，首先介绍词法记号及用于描述词法规则的正规式，然后讨论如何通过正规式自动构造有限自动机，最后介绍词法分析器的自动生成工具。

2.1 词法记号、属性及词法错误

词法分析器是编译器中负责扫描源程序、生成词法记号的组件，它还可以执行与用户交互的其他一些任务。它可以过滤掉源程序中的注解和空白字符（如空格、制表符和换行符），还可以把编译器各个阶段生成的错误信息与源程序对应起来，（如记录字符的行列号以便将错误信息与源程序中的位置关联起来）。在某些编译器中，词法分析器还负责复制源程序并把错误信息嵌入其中。此外，如果源语言支持宏定义，那么对宏定义的预处理也可以在词法分析时实现。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第31页 文件编号: 422624 页面编号: 2314538444 打印人: 8685 2026-03-09 10:22:58

2.1.1 词法单元、词法记号、模式

在讨论词法分析时，会使用以下 3 个相关但有区别的术语。

（1）词法单元（lexeme）。词法单元又称单词或词素，是源程序中具有某种词法含义的字符序列，如标识符“sum”、某个特定的关键字“if”、某个关系运算符“>”等。

（2）词法记号（token）。词法记号简称记号，是由记号名和属性值构成的二元组，属性值不是必需项。记号名是代表一类词法单元的抽象名字，在下面都用黑体表示，如标识符 **id**、某个特定的关键字 **if**。词法记号是语法分析的输入符号，一般用记号名来引用记号。

（3）模式（pattern）。一个记号的模式描述属于该记号的词法单元的形式。在一个关键字作为一个记号的情况下，它的模式就是构成该关键字的字符序列。对于标识符和其他一些记号，它们的模式有更复杂的结构且有很多字符串可以**匹配**它们。

词法分析器的任务是扫描输入的源程序，尝试将读入的字符序列匹配到一个记号模式，所匹配的字符序列是一个词法单元，它被词法分析器识别为该记号的一个实例。

表 2.1 给出了一些典型的记号名、词法单元及其非形式描述的模式。例如，在 C 语言的语句

```
printf(''Total = %d\n'', score);
```

中，printf 和 score 是匹配 **id** 模式的词法单元，“Total = %d n”是匹配 **literal** 模式的词法单元。

表 2.1 一些典型的记号名、词法单元及其非形式描述的模式

记号名	词法单元列举	模式的非形式描述
if	if	字符 i，f
for	for	字符 f，o，r
relop	<, >, <=, >=, =, <>	<或>或<=或>=或=或<>
id	sum，count，D5	由字母开头的字母数字串
number	3.1416，10，2.08E12	任何数值常数
literal	“segmentation error”	引号和之间任意不含引号本身的字符串

在大多数编程语言中，各种关键字、各类运算符、标识符、常数、文字串（字符串）和标点符号都处理为记号。

某些语言的一些规定给词法分析带来了困难。例如，在对待空格上，不同语言有不同规定。在早先的一些语言（如 FORTRAN 和 ALGOL68）中，空格无意义（文字串中的除外），并不作为词法单元之间的分隔符，空格可以随便加入，以改变程序的可读性。空格的这种约定增加了处理标识符记号的复杂性。典型例子是 FORTRAN 的 DO 语句，在语句

```
DO 8 I = 3. 75
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第32页 文件编号: 422624 页面编号: 2324538442 打印人: 8685 2026-03-09 10:22:58

中，词法分析器只有看到小数点后，才能确定此处的 DO 在这里不是关键字，而 DO8I 是一个标识符。但是若把小数点换成逗号，得到语句

DO 8 I = 3, 75

这时 DO 是关键字，该语句共有 7 个记号：关键字 DO、语句标号 8、标识符 I、运算符 =、常数 3、逗号和常数 75。在没有看到逗号之前，词法分析器不能确定 DO 是关键字。空格的这种规定给词法分析带来的困难是，需要向前额外多扫描若干字符，才能回过头来确定一个记号。

另一个例子是，关键字是否保留。**保留字**是语言预先确定了含义的词法单元，程序员不可以对这样的词法单元重新声明它的含义，如 Pascal 语言的 var 和 begin 等是保留字。很多语言使用关键字概念，并且关键字是保留的，因此它和保留字概念没有区别，如 C 语言和 Java 语言。但是 FORTRAN 语言的关键字不保留，如 IF，当它作为语句的第一个词法单元时，很可能是关键字, 但也不排除它是一个程序中显式声明的标识符。若 IF 出现在语句的其他地方，它一定是程序中显式或隐式声明的标识符。这就给词法分析带来很大困难，因为识别一个记号和该记号所处的上下文有关了。

有些语言还使用的标准标识符概念。**标准标识符**也是预先确定了含义的标识符，但程序可以重新声明它的含义。在该声明的作用域内，程序声明的含义起作用，而预先确定的含义失效，在其他地方则是预先确定的含义起作用，如 Pascal 语言的 integer 和 true 等。词法分析器对标准标识符没有特别的处理，交由符号表管理来处理。

2.1.2 词法记号的属性

由 2.1.1 节可知，C 语言中的 6 个关系运算符都归属记号 **relation**。因为从程序的语法是否正确的角度看，使用哪个关系运算符都一样。但是从翻译成目标代码来考虑，不同的关系运算符，其翻译结果不一样。因此，词法分析器需要给记号以属性值，用属性值来记住记号的附加信息，以便需要时使用。概括地说，记号名影响语法分析的决策，属性值影响记号的翻译。

【例 2.1】 语句

```
position = initial + rate * 60
```

的记号用二元组序列表示如下：

<**id**，指向符号表中 position 条目的指针 >
<**assign_op**>
<**id**，指向符号表中 initial 条目的指针 >
<**add_op**>
<**id**，指向符号表中 rate 条目的指针 >

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第33页 文件编号: 422624 页面编号: 2334538440 打印人: 8685 2026-03-09 10:22:58

<mul_op>

<number，整数值 60>

上面有些二元组没有属性值，因为它的第一个成分足以辨别词法单元，如运算符、标点符号和关键字等。在这个例子中，记号 **number** 被赋予了一个整数值属性。编译器也可以把形成数的字符串存入数表，让记号 **number** 的属性值是指向这个条目的指针。

2.1.3 词法错误

词法分析难以发现源程序中的错误，因为词法分析器对源程序的分析限于局部。在 C 语言的语句

fi (a == f (x)) ⋯

中，词法分析器把 fi 当作一个普通的标识符交给编译的后续阶段，而不会把它看成是关键字 if 的拼写错误。

有些语言要求实型常量的小数点后面必须有数字。如果程序中出现小数点后面没有数字的情况，词法分析器会报错。

最简单的错误恢复策略是“紧急方式”恢复。它删掉输入指针当前指向的若干个字符（剩余输入的前缀），直到词法分析器能发现一个正确的记号为止。

另一种策略是进行错误修补尝试。最简单的办法是看剩余输入的前缀能否用下面的一个变换变成一个合法的词法单元：

（1）删除一个多余的字符；

（2）插入一个遗漏的字符；

（3）用一个正确的字符代替一个不正确的字符；

（4）交换两个相邻的字符。

这种策略基于的假设是，大多数词法错误是多、漏或错了一个字符，或相邻的两个字符错位。这种假设通常是（但不总是）正确的。

2.2 词法记号的描述与识别

2.1 节提到，字符串集合由模式的规则描述。正规式是表示这些规则的一种重要方法，本节围绕正规式来介绍记号的描述与识别。在介绍正规式前，先给“语言”一个形式定义。

2.2.1 字母表、串和语言

术语“**字母表**”表示符号的有限集合，用 $\sum$ 表示。符号的典型例子有英文字母、数字和标点符号等。集合 {0, 1} 是二进制字母表；ASCII 是字母表的一个重要例子，它用于很多

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第34页 文件编号: 422624 页面编号: 2344538448 打印人: 8685 2026-03-09 10:22:58

软件系统中；Unicode 是另一个重要例子，它包含了取自全世界各个字母表的大约 10 万个字符。

字母表上的**串**是指该字母表符号的有穷序列。串 s 的长度是指出现在 s 中符号的个数，往往写作 $|s|$。例如，banana 是长度为 6 的串。空串是指长度为 0 的特殊串，用 ε 表示。

术语“**语言**”表示字母表上的一个串集。属于该语言的串称为该语言的**句子**或**字**。这个定义相当宽，像 $\varnothing$（空集）和 $\{\varepsilon\}$（仅含空串的集合）这样的抽象语言也符合这个定义，所有语法正确的 C 程序的集合和所有语法正确的英语句子集合也都分别符合此定义。当然，后两个集合更难描述。注意，这个定义并没有对语言中的串赋予任何意义，这个问题将在第 4 章讨论。

如果 x 和 y 都是串，那么 x 和 y 的**连接**（写成 xy）是指把 y 加到 x 后面形成的串。对连接运算而言，空串是一个恒等元素，也就是 $s\varepsilon = \varepsilon s = s$。

如果把连接看成“积”，那么可以定义串的“幂”。定义 s^0 为 ε，s^i 为 $s^{i-1}s$（$i > 0$）。因为 εs 是 s 本身，所以 $s^2 = ss$、$s^3 = sss$ 等。

有一些重要的运算可以作用于语言。与词法分析相关的运算有并、连接、闭包和正闭包，它们定义在表 2.2 中，其中 L、M 表示语言。幂运算也可用于语言 L，定义 L^0 是 $\{\varepsilon\}$，L^i 是 $L^{i-1}L$，即 L^i 是 L 连接它自己 $i-1$ 次。

表 2.2 语言运算的定义

运算	定义
L 和 M 的并（写做 $L \cup M$）	$L \cup M = \{s \mid s \in L \text{ 或 } s \in M\}$
L 和 M 的连接（写做 LM）	$LM = \{st \mid s \in L \text{ 且 } t \in M\}$
L 的闭包（写做 L^*）	$L^* = \bigcup_{i=0}^{\infty} L^i$，$L^*$ 表示零个或多个 L 连接的并集
L 的正闭包（写做 L^+）	$L^+ = \bigcup_{i=1}^{\infty} L^i$，$L^+$ 表示一个或多个 L 连接的并集

【例 2.2】 令 L 表示集合 $\{A, B, \cdots, Z, a, b, \cdots, z\}$，令 D 表示集合 $\{0, 1, \cdots, 9\}$。下面是用表 2.2 定义的运算作用于 L 和 D 所得到的新语言的例子。

（1）$L \cup D$ 是字母和数字的集合。

（2）LD 是所有由一个字母后跟一个数字组成的串的集合。

（3）L^6 是所有由 6 个字母组成的串的集合。

（4）L^* 是所有字母串（包括 ε）的集合。

（5）$L(L \cup D)^*$ 是以字母开头的所有字母数字串的集合。

（6）D^+ 是不含空串的数字串的集合。

从这个例子可以看出，从基本集合开始，利用运算可以定义新的语言。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第35页 文件编号: 422624 页面编号: 2354538445 打印人: 8685 2026-03-09 10:22:58

2.2.2 正规式

正规式（regular expression），又称正规表达式、正则表达式，是通过应用如下一组定义规则构成的表达式。其中每个正规式 r 表示一个语言 $L(r)$，该语言称为**正规语言**或**正规集**。这些定义规则同时也说明 $L(r)$ 是怎样从 r 的子正规式所表示的语言中构造出来的。

（1）ε 是正规式，它表示 $\{\varepsilon\}$。

（2）如果 a 是 Σ 上的符号，那么 a 可以作为正规式，它表示语言 $\{a\}$。虽然都用 a 表示，但正规式 a 不同于语言 $\{a\}$ 中的句子 a。根据上下文可以清楚地区分 a 是正规式还是串。

（3）假定 r 和 s 都是正规式，它们分别表示语言 $L(r)$ 和 $L(s)$，那么 $(r)|(s)$、$(r)(s)$、$(r)^*$ 和 (r) 都是正规式，分别表示语言 $L(r)\cup L(s)$、$L(r)L(s)$、$(L(r))^*$ 和 $L(r)$。

约定：

①闭包运算（运算符是 *）优先级最高且是左结合的；

②连接运算（两个正规式并列）的优先级次之且也是左结合的；

③选择运算（运算符是 $|$）的优先级最低且仍然是左结合的；

这样可以避免正规式中一些不必要的括号，如 $((a)(b)^*)|(c)$ 等价于 $ab^*|c$。

【例 2.3】 令字母表 $\Sigma=\{a,b\}$，那么：

（1）正规式 $a|b$ 表示集合 $\{a,b\}$；

（2）正规式 $(a|b)(a|b)$ 表示 $\{aa,ab,ba,bb\}$，即由 a 和 b 构成的所有长度为 2 的串集，这个集合也可以用正规式 $aa|ab|ba|bb$ 表示；

（3）正规式 a^* 表示仅由字母 a 构成的所有串的集合，包括空串；

（4）正规式 $(a|b)^*$ 表示由 a 和 b 构成的所有串的集合，包括空串。

如果正规式 r 和 s 表示同样的语言，则 r 和 s 等价，写作 $r=s$。例如，$(a|b)=(b|a)$。正规式遵守一些代数定律，它们可用于正规式的等价变换，表 2.3 列出了正规式 r、s 和 t 遵守的部分代数定律。

表 2.3 正规式的代数定律

定律	描述
$r\|s=s\|r$	$\|$ 是可交换的
$r\|(s\mid t)=(r\mid s)$ t	$\|$ 是可结合的
$(rs)t=r(st)$	连接是可结合的
$r(s\mid t)=rs\|rt;(s\mid t)r=sr\|tr$	连接对 $\|$ 是可分配的
$\varepsilon r=r;r\varepsilon=r$	ε 是连接的恒等元素
$r^*=(r\mid\varepsilon)^*$	ε 肯定出现在一个闭包中
$r^{**}=r^*$	$*$ 是幂等的

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第36页 文件编号: 422624 页面编号: 2364538443 打印人: 8685 2026-03-09 10:22:58

2.2.3 正规定义

可以对正规式命名，并用这些名字来引用相应的正规式，以求得表示上的简洁。这些名字也可以像符号一样，出现在正规式中。

如果 Σ 是基本符号的字母表，那么**正规定义**是形式为

$$
\begin{aligned}
d_1 &\rightarrow r_1 \\
d_2 &\rightarrow r_2 \\
&\cdots\cdots\cdots\cdots \\
d_n &\rightarrow r_n
\end{aligned}
$$

的定义序列，各个 d_i 的名字都不相同，每个 r_i 都是 $\Sigma \cup \{d_1, d_2, \cdots, d_{i-1}\}$ 上的正规式。由于每个 r_i 只能含 Σ 上的符号和前面定义的名字，因此不会出现递归定义的情况。把这些名字用它们所表示的正规式来代替，就可以为任何 r_i 构造 Σ 上的正规式。

为了区别名字和符号，本书在正规定义中用黑体字表示名字。

【例 2.4】 C 语言的标识符是以字母或下划线开头并由字母、数字和下划线组成的串，下面是 C 语言标识符的正规定义。

$$
\begin{aligned}
&\textbf{letter_} \rightarrow \mathrm{A} \mid \mathrm{B} \mid \cdots \mid \mathrm{Z} \mid \mathrm{a} \mid \mathrm{b} \mid \cdots \mid \mathrm{z} \mid _ \\
&\textbf{digit} \rightarrow 0 \mid 1 \mid \cdots \mid 9 \\
&\textbf{id} \rightarrow \textbf{letter_}(\textbf{letter_} \mid \textbf{digit})^*
\end{aligned}
$$

【例 2.5】 无符号数（整数和浮点数）是如同 1946、11.28、63.6E8 和 1.999E-6 这样的串，下面是这个串集合的正规定义。

$$
\begin{aligned}
&\textbf{digit} \rightarrow 0 \mid 1 \mid \cdots \mid 9 \\
&\textbf{digits} \rightarrow \textbf{digit digit}^* \\
&\textbf{optional_fraction} \rightarrow .\textbf{digits} \mid \varepsilon \\
&\textbf{optional_exponent} \rightarrow (E(+ \mid - \mid \varepsilon)\ \textbf{digits}) \mid \varepsilon \\
&\textbf{number} \rightarrow \textbf{digits optional_fraction optional_exponent}
\end{aligned}
$$

从这个定义可知，无符号数由整数部分、小数部分和指数部分组成，其中小数部分和指数部分可能出现也可能不出现。如果出现指数部分，则是 E 及可能有的 + 或 − 号，再跟上一个或多个数字。注意小数点后面至少有一个数字，所以 **number** 能匹配 “2.0”，但不能匹配 “2.”。

在正规式中，某些结构频繁出现，为方便起见，可以用缩写表示它们：

（1）一个或多个实例。一元后缀运算符 “+” 的意思是 “一个或多个实例”，即正规式 a^+ 表示所有由一个或多个 a 组成的串的集合。运算符 $^+$ 和运算符 * 有同样的优先级和结合性。代数恒等式 $r^* = r^+|\varepsilon$ 和 $r^+ = rr^*$ 表达了这两个运算符之间的关系。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第37页 文件编号: 422624 页面编号: 2374538441 打印人: 8685 2026-03-09 10:22:58

（2）零个或一个实例。一元后缀运算符“？”的意思是“零个或一个实例”，$r?$ 是 $r|\varepsilon$ 的缩写。如果 r 是正规式，那么 $(r)?$ 是表示语言 $L(r)\cup\{\varepsilon\}$ 的正规式。使用这两种缩写，可以用

$$\textbf{number}\rightarrow\textbf{digit}^{+}(.\textbf{digit}^{+})?(E(+|-)?\textbf{digit}^{+})?$$

来描述无符号数。

（3）字符组。$[abc]$（其中 a、b 和 c 是字母表的符号）表示正规式 $a\mid b\mid c$。缩写字符组 $[a-z]$ 表示正规式 $a\mid b\mid\cdots\mid z$。使用字符组，可以用正规式

$$\begin{aligned}&\textbf{letter_}\rightarrow[\mathrm{A}-\mathrm{Za}-\mathrm{z_}]\\&\textbf{digit}\rightarrow 0\mid 1\mid\cdots\mid 9\\&\textbf{id}\rightarrow\textbf{letter_}(\textbf{letter_}\mid\textbf{digit})^{*}\end{aligned}$$

描述标识符。

利用上述表示，可以表达一些实际的例子。例如，表示所有以.o 结尾的文件名的正规式是 (. | o | c)*.o，其中 c 代表除. 和 o 以外的任何字符。又如，C 语言的注释是以/* 开始和以 */结束的任意字符串，但注释的任何前缀（本身除外）不以 */结尾。作为练习，读者可以用正规式表示这种注释。

2.2.4 状态转换图

本节以下面正规定义描述的语言为例，讨论怎样识别记号。

【例 2.6】 考虑下面的正规定义：

$$\begin{aligned}&\textbf{while}\rightarrow\mathrm{while}\\&\textbf{do}\rightarrow\mathrm{do}\\&\textbf{relop}\rightarrow<\mid<=\mid=\mid<>\mid>\mid>=\\&\textbf{letter}\rightarrow[\mathrm{A}-\mathrm{Za}-\mathrm{z}]\\&\textbf{id}\rightarrow\textbf{letter}(\textbf{letter}\mid\textbf{digit})^{*}\\&\textbf{number}\rightarrow\textbf{digit}^{+}(.\textbf{digit}^{+})?(E(+\mid-)?\textbf{digit}^{+})?\end{aligned}$$

其中，**digit** 与前面所定义的一样。以上正规定义是某语言 while 语句中可能出现的部分记号的描述。词法分析器将识别关键字 while 和 do，还有关系运算符、标识符和数值。假定词法单元之间在必要时用空格 (或制表符、换行符) 分开，词法分析器通过把剩余输入的前缀和下面的正规定义 **ws** 相比较来完成忽略词法单元之间的空白：

$$\begin{aligned}&\textbf{delim}\rightarrow\textbf{blank}\mid\textbf{tab}\mid\textbf{newline}\\&\textbf{ws}\rightarrow\textbf{delim}^{+}\end{aligned}$$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第38页 文件编号: 422624 页面编号: 2384538449 打印人: 8685 2026-03-09 10:22:58

如果剩余输入的前缀能由 **ws** 匹配，词法分析器不返回记号给分析器，它继续去寻找空白后面的记号，然后返回给分析器。

词法分析器输出的是 < 记号名，属性值 > 二元组序列，表 2.4 给出了各正规式描述的这种记号的二元组。注意，正规式 **ws** 没有对应的记号，另外，关系运算符的属性值由符号常量 LT、LE、EQ、NE、GT 或 GE 给出，它们分别表示 <，⩽，=，<>，>，⩾。

表 2.4 正规式、记号名和属性值

正规式	记号名	属性值
ws	—	—
while	**while**	—
do	**do**	—
id	**id**	符号表条目的指针
number	**number**	数表条目的指针
relop	**relop**	LT、LE、EQ、NE、GT 或 GE

绘制**状态转换图**（简称转换图）是构造词法分析器的第一步。状态转换图描绘词法分析器被语法分析器调用时，词法分析器为返回下一个记号所做的动作。

图 2.2 是识别记号 **relop** 的转换图，可以用这张图来解释有关转换图的概念。转换图上的圆圈称为**状态**。状态由**有向边**连接，边上有指示输入字符的标记，标记通常是一个字符。若离开状态 s 的某条边上有标记 **other**，则它表示离开 s 的其他边所指示的字符以外的任意字符。本小节讨论的转换图是确定的，即不可能出现某字符与离开一个状态的两条边上的标记都匹配的情况。

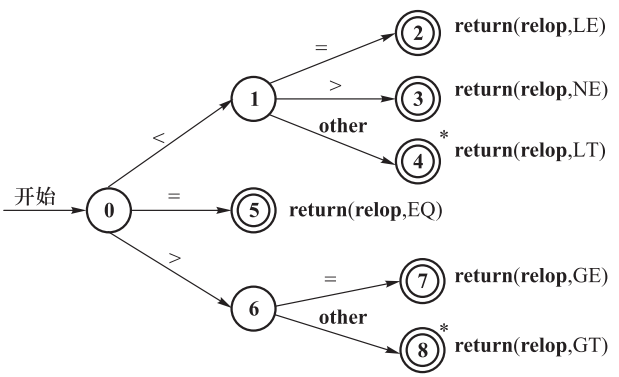


图 2.2 关系运算符的转换图

在状态转换图中，有一个状态标记为**开始状态**，这是转换图的初始状态（简称**初态**）。当开始识别记号时，控制进入开始状态。当控制进入某个状态时，读输入串的下一个字符。如

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第39页 文件编号: 422624 页面编号: 2394538447 打印人: 8685 2026-03-09 10:22:58

果离开这个状态的一条边上的标记和该输入字符匹配，控制就进入由这条边指向的状态，否则识别过程失败。某些状态有双圈，表示它们是**接受状态**（简称**终态**）。控制进入这样的状态表明识别了一个记号。接受状态可以有动作，控制到达接受状态时执行该状态的动作。

对于图 2.2 的状态转换图，如果输入串是“<=···”，那么控制从开始状态 0 到达接受状态 2，读出词法单元 <=，执行动作 **return**(**relop**, LE)。

注意，如果到达接受状态 4，则意味着 < 和另一个字符已被读过。由于第二个字符不是关系运算符 < 的一部分，必须把输入串上指示下一个字符的指针回退一个字符。用 * 表示输入指针必须回退的状态。

【例 2.7】 图 2.3 所示为标识符和关键字的转换图，其中边上的标记 **letter** 和 **digit** 分别是字母集和数字集。当边上的标记是一个字母集时，若输入字符是该字母集的成员，则称该标记和这个字母匹配。该转换图也可以用来识别关键字，因为关键字是特殊的标识符。当然，到达接受状态时，需要执行某段代码，以判定到达接受状态的词法单元是关键字还是标识符。

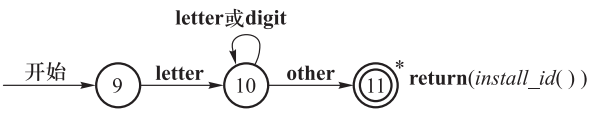


图 2.3 标识符和关键字的转换图

把关键字从标识符中分离出来的简单办法是建立一张关键字表。在扫描字符之前，把构成关键字的串，如 while 和 do 以及与它们对应的记号名都加入该表，以便识别这些串时返回。图 2.3 中接受状态旁边的返回语句使用 installId( ) 来获得要返回的记号名和属性值。过程 installId( ) 首先查看关键字表，如果当前词法单元构成关键字，则返回相应的记号；否则该词法单元是标识符。该过程再查标识符表，如果在表中发现该词法单元，则返回相应的条目指针；如果没有找到，则把该词法单元填入标识符表，并返回新建条目的指针（很多编译器在语法分析阶段才将标识符填入标识符表，这时 **id** 的属性值是它的拼写形式）。

如果要识别的关键字有所变化，则无须修改转换图，只需重置关键字表的初值即可。也可以为关键字单独构造转换图，但对典型的编程语言来说，这么做会使词法分析器的状态数多达几百个，而用上面的方法，可能用不到 100 个状态就够了。

【例 2.8】 根据正规定义，

$$\textbf{number} \rightarrow \textbf{digit}^{+}(.\textbf{digit}^{+})?(\mathrm{E}(+|-)?\textbf{digit}^{+})?$$

为无符号数构造识别器。注意，在这个定义中，小数部分 (.**digit**$^{+}$) 和指数部分 (E (+ | −)? **digit**$^{+}$) 是可选的。图 2.4 是无符号数转换图。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第40页 文件编号: 422624 页面编号: 2404538445 打印人: 8685 2026-03-09 10:22:58

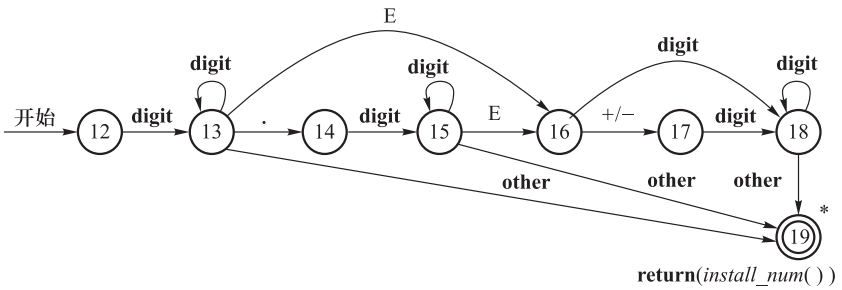


图 2.4 无符号数转换图

到达接受状态的动作是调用过程 installNum()，它把词法单元置入数表，并返回建立的条目指针。词法分析器返回记号名 **number**，以及作为属性值的该指针。

把图 2.2、图 2.3 和图 2.4 的 3 个开始状态 0、9 和 12 合并成一个开始状态，就可以把这 3 个转换图合并成一个转换图。

剩下的问题就是词法单元间的空白。代表空白的 **ws** 的处理与上面讨论的代表各种记号的正规式的处理有所不同，因为在输入串中发现空白时，并不返回任何东西给语法分析器。识别 **ws** 的转换图如图 2.5 所示。把这个转换图和其他几个转换图合并成一个转换图后，到达接受状态 22 的动作就是回到开始状态，即识别下一个记号。

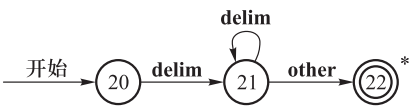


图 2.5 **ws** 的转换图

2.2.5 基于转换图的词法分析器的构造

根据正规式构造出转换图后，基于这种直观的转换图，可以相对容易地编写一个程序来识别正规式所描述的记号。

【例 2.9】 对于图 2.2 关系运算符的转换图，可以编写图 2.6 所示的 getRelop() 函数来识别关系运算符。

getRelop() 函数首先创建一个记号名为 **RELOP** 的新的记号对象 retToken。变量 state 保存识别器当前所处的转换图状态的编号，初始为转换图的开始状态（如 0）。图 2.6 中第 4~16 行是根据当前状态决定转换到的下一状态。例如，在状态 0 下，先调用 nextChar( ) 读入下一字符到局部变量 c；再检查 c 是不是所期望的 3 种字符之一：若是，则按照转换图的示意进行状态转换；否则说明不是关系运算符，调用 fail( ) 进行出错处理。当转换图处在状态 8 时，由于状态 8 标有 *，因此调用 retract( ) 将输入指针回退一个字符（把当前读入的字符 c 回退到输入流中）；另外，由于状态 8 为识别出 > 的接受状态，因此把要返回的记号

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第41页 文件编号: 422624 页面编号: 2414538443 打印人: 8685 2026-03-09 10:22:58

的属性值设置为 GT。第 3 行的无限循环是为了循环处理字符，直至返回关系运算符 (第 15 行) 或者出错退出 (第 9 行)。 □

```
1  TOKEN getRelop() {
2     TOKEN retToken = new(RELOP);
3     while (1) {
4       switch (state) {
5         case 0: c = nextChar();
6            if (c == '<') state = 1;
7            else if (c == '=') state = 5;
8            else if (c == '>') state = 6;
9            else fail();
10           break;
11        case 1: ...
12        ...
13        case 8: retract();
14           retToken.attribute = GT;
15           return(retToken);
16      }
17    }
18 }
```

图 2.6 基于转换图的关系运算符识别函数的实现概要

当然，编写词法分析器并非一定要基于转换图，本节介绍状态转换图是为了便于大家理解后续的有限自动机的概念。

2.2.6 词法分析中的冲突及解决

词法分析中的冲突主要是指当一个输入序列能够匹配多个不同的词法规则时，词法分析器无法确定应该选择哪一个规则来生成词法记号。这类冲突主要有以下两种常见情况。

1. 最长匹配规则

当多个规则都可以匹配同一输入序列时，词法分析器通常会选择匹配最长字符串的规则。这被称为**最长匹配（maximal munch）优先**。例如，假设有规则：

$$R = \text{Whitespace} \mid \text{Integer} \mid \text{Identifier} \mid \text{'+'}$$

在分析字符串''foo+3''时，''f''、''fo''、''foo''都匹配 R，更精确地说是标识符 Identifier，但是''foo+''不匹配，这时词法分析器会选择匹配最长的''foo''为标识符。

一般地，如果 $x_1 \cdots x_i \in L(R)$ 且 $x_1 \cdots x_k \in L(R)$, 而 $i < k$，则可按最长匹配规则来选择，即选择匹配 R 的最长前缀 $x_1 \cdots x_k$。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第42页 文件编号: 422624 页面编号: 2424538441 打印人: 8685 2026-03-09 10:22:58

冲突解决方式：使用最长匹配规则，即词法分析器向前看字符，尽可能匹配最长的词法单元，不匹配时则回退并生成词法记号。

2. 优先级

当两个规则都可以匹配相同长度的输入时，通常词法分析器会根据词法规则的顺序来决定优先级。例如，假设有规则

$$R = \text{Whitespace} \mid \text{'new'} \mid \text{Integer} \mid \text{Identifier}$$

在分析字符串''new foo''时，''new''匹配关键字'new'，也匹配标识符 Identifier。对于'new'这个输入，如果关键字规则在标识符规则之前定义，那么词法分析器会将其匹配为关键字，而不是标识符。

一般地，如果有 $x_1 \cdots x_i \in L(R_j)$ 和 $x_1 \cdots x_i \in L(R_k)$，则选择先列出的模式 R_j，如果 $j < k$。

冲突解决方式：通过调整规则的顺序，将更具体的规则放在前面，更通用的规则放在后面。

2.3 有限自动机

语言的**识别器**是一个程序，它取串 x 作为输入。当 x 是语言的句子时，它回答“是”，否则回答“不是”。可以通过构造有限自动机的更一般的转换图，把正规式翻译成识别器。

有限自动机分成确定的和不确定的两种情况。“不确定”的含义是，存在这样的状态，对于某个输入符号，它存在不止一种转换。本节先介绍这两种自动机的定义，2.4 节介绍如何从正规式自动地构造有限自动机。本节和 2.4 节都以正规式 $(a|b)^*ab$ 表示的语言为例进行介绍。

2.3.1 不确定的有限自动机

不确定的有限自动机（nondeterministic finite automata，NFA）是一个数学模型，它包括：

（1）一个有限的状态集合 S；

（2）一个输入符号的集合 Σ（也称输入符号字母表，代表空串的 ε 不出现在 Σ 中）；

（3）一个转换函数 move：$S \times (\Sigma \cup \varepsilon) \to P(S)$[$P(S)$ 指 S 的幂集]，它把状态和符号（可以是 ε）二元组映射到一个状态集合；

（4）状态 s_0 是唯一的开始状态；

（5）状态集合 F 是接受（或终止）状态集合，并且 $F \subseteq S$。

NFA 可以用带标记的有向图表示，叫做状态转换图，结点表示状态，有标记的边代表转换函数。这种转换图和 2.2 节所讲的略有区别，这里可以把同样的符号标记在从同一个状态引出的多条边上。另外，在边上可以标记输入符号，也可以标记特殊符号 ε。

可以识别语言 $(a|b)^*ab$ 的 NFA 转换图如图 2.7 所示。这个 NFA 的状态集合是 {0, 1, 2}，输入符号表是 $\{a, b\}$，状态 0 是开始状态，接受状态 2 用双圈表示。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第43页 文件编号: 422624 页面编号: 2434538449 打印人: 8685 2026-03-09 10:22:58

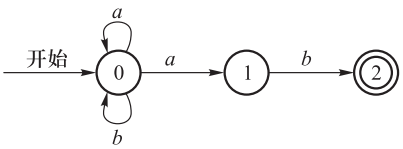


图 2.7　识别 $(a|b)^*ab$ 的 NFA

NFA 可以用这种转换图来描述，在计算机上可以用不同方法实现。最简单的方法是**转换表**，每个状态一行，每个输入符号和 ε(如果需要的话) 各占一列，表的第 i 行中符号 a 对应的单元格条目是一个状态集合（实现时可以是状态集合的指针），表示 NFA 在输入符号 a 时，状态 i 所能到达的状态集合。表 2.5 是对应图 2.7 的 NFA 的转换表。

表 2.5　图 2.7 的 NFA 的转换表

状态	输入符号	
	a	b
0	$\{0,1\}$	$\{0\}$
1	$\varnothing$	$\{2\}$
2	$\varnothing$	$\varnothing$

这种转换表的优点是可以快速获得给定状态在给定输入符号时转换到的状态集；缺点是当字母表 Σ 较大，并且大多数转换是空集时，转换表占用了大量空间。

NFA **接受**输入串 x 的条件是，当且仅当转换图中存在从开始状态到某个接受状态的路径，该路径各边上的标记可拼成 x。图 2.7 的 NFA 可以接受输入串 ab，aab，bab，$aaab$，$\cdots$。例如，从状态 0 开始，沿着标记为 a 的边转换到状态 0，然后沿着标记为 a 和 b 的边先后转换到达状态 1 和 2，按此路径转换可以接受 aab。对一个输入串，可能有多条路径可以到达接受状态。

由 NFA 定义的语言是它所接受的输入串集合。

【例 2.10】　图 2.8 是识别 $aa^*|bb^*$ 的 NFA，串 aaa 由通过 0，1，2，2 和 2 的路径来接受，相应边上的标记分别是 ε，a，a 和 a，它们拼成 aaa，ε 在拼接中“消失”。

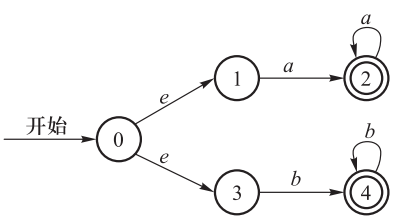


图 2.8　识别 $aa^*|bb^*$ 的 NFA

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第44页 文件编号: 422624 页面编号: 2444538447 打印人: 8685 2026-03-09 10:22:59

2.3.2 确定的有限自动机

确定的有限自动机 (deterministic finite automata，DFA) 是 NFA 的特殊情况。其中:

（1）任何状态下都没有 ε 转换，即任何状态都必须进行输入符号的匹配才能进入下一个状态；

（2）对任何状态 s 和任何输入符号 a，最多只有一条标记为 a 的边离开 s，即转换函数 move: $S \times \Sigma \to S$ 可以是一个部分函数。

确定的有限自动机从任何状态出发，对于任何输入符号，最多只有一个转换。如果用转换表来表示 DFA 的转换函数，那么表中的每个条目最多只有一个状态。结果是，很容易判断 DFA 能否接受一个输入串，因为从开始状态起，最多只有一条到达某个终态的路径可以由这个串标记。下面的算法说明怎样模拟 DFA 的行为。

算法 2.1 模拟 DFA。

输入：输入串 x，由文件结束符 **eof** 结尾。一个 DFA D，其开始状态是 s_0，其接受状态集合是 F。

输出：如果 D 接受 x，则返回“yes”，否则返回“no”。

方法：把图 2.9 的算法施加于输入串 x。函数 move(s,c) 给出一个状态，它是对于输入符号 c，状态 s 的转换。函数 nextChar( ) 返回输入串 x 中的下一个字符。

```
s = s0;
c = nextChar( );
while (c != eof) {
    if (move(s, c)未定义) return "no"; else s = move (s, c);
    c = nextChar( );
}
if (s 属于 F) return "yes";
else return "no" ;
```

图 2.9 模拟 DFA

【例 2.11】 如图 2.10 所示的转换图表示一个 DFA，它和图 2.7 的 NFA 识别同样的语言 $(a|b)^*ab$。用这个 DFA 和输入串 $abab$ 作为算法2.1 的输入，根据算法可知状态沿着 0，1，2，1 和 2 移动，并返回 yes。

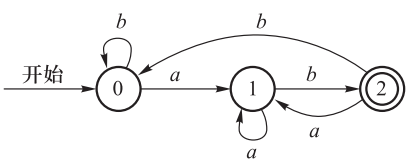


图 2.10 识别 $(a|b)^*ab$ 的 DFA

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第45页 文件编号: 422624 页面编号: 2454538444 打印人: 8685 2026-03-09 10:22:59

NFA 和 DFA 都能识别正规集，它们能识别的语言需要是正规语言。它们之间存在**时空权衡**问题：从 DFA 得到的识别器，比从等价的 NFA 得到的识别器要快得多；但是，DFA 可能比等价的 NFA 占用更多的空间。

2.4 从正规式到有限自动机

有很多方法可以从正规式建立它所定义语言的识别器，每种方法各有优劣。本节介绍的方法是先从正规式构造 NFA，再用子集构造法把 NFA 变成 DFA，最后对 DFA 化简。

2.4.1 从正规式构造 NFA

这里介绍的构造算法是**语法制导的**（syntax-directed），即它根据正规式的语法结构来指导 NFA 的构造过程。首先构造识别 ε 和字母表中一个符号的自动机，然后构造识别主运算符为选择、连接或闭包的正规式的自动机。例如，正规式 $r|s$ 的 NFA 是从 r 和 s 的 NFA 中归纳构造出的。

在构造过程中，每步最多引入两个新的状态，所以为正规式构造的最终 NFA 状态数最多是正规式中符号和运算符总数的两倍。

算法 2.2 从正规式构造 NFA。

输入：字母表 Σ 上的正规式 r。

输出：接受 $L(r)$ 的 NFA N。

方法：首先分析 r，把它分解成子表达式，然后使用下面的规则（1）和（2），为 r 中的每个基本符号（ε 或字母表符号）构造 NFA。注意，如果符号 a 在 r 中出现多次，那么要为它的每次出现构造一个 NFA。

然后，根据正规式 r 的语法结构，用下面的规则（3）归纳地组合这些 NFA，直到获得整个正规式的 NFA 为止。在构造过程中所产生的每个中间 NFA 均满足一些重要的性质：只有一个接受状态，没有边进入开始状态，也没有边离开接受状态。

（1）对于 ε，构造如图 2.11 所示的 NFA，其中 i 是开始状态，f 是接受状态。很明显，这个 NFA 识别 $\{\varepsilon\}$。

（2）对 Σ 中的每个符号 a，构造如图 2.12 所示的 NFA。同样，i 是开始状态，f 是接受状态。这个 NFA 识别 $\{a\}$。

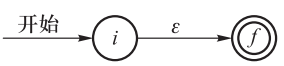


图 2.11 识别正规式 ε 的 NFA

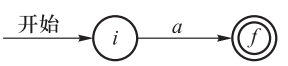


图 2.12 识别正规式 a 的 NFA

（3）如果 $N(s)$ 和 $N(t)$ 分别是正规式 s 和 t 的 NFA，则

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第46页 文件编号: 422624 页面编号: 2464538442 打印人: 8685 2026-03-09 10:22:59

① 对于正规式 $s|t$，构造合成的 NFA $N(s|t)$，结果如图 2.13 所示。这里 i 是新的开始状态，f 是新的接受状态。从 i 到 $N(s)$ 和 $N(t)$ 的开始状态有 ε 转换，从 $N(s)$ 和 $N(t)$ 的接受状态到 f 也有 ε 转换。$N(s)$ 和 $N(t)$ 的开始和接受状态不是 $N(s|t)$ 的开始和接受状态。这样，从 i 到 f 的任何路径必须通过 $N(s)$ 或 $N(t)$。这个合成的 NFA 识别 $L(s) \cup L(t)$。

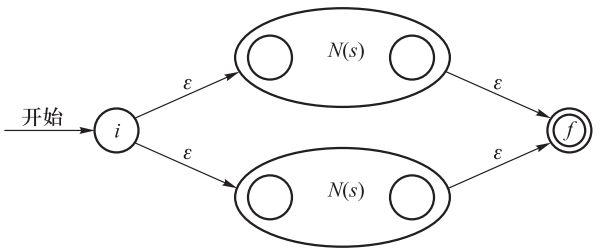


图 2.13 识别正规式 $s|t$ 的 NFA

② 对于正规式 st。构造合成的 NFA $N(st)$，结果如图 2.14 所示。$N(s)$ 的开始状态成为合成后的 NFA 的开始状态，$N(t)$ 的接受状态成为合成后的 NFA 的接受状态。$N(s)$ 的接受状态和 $N(t)$ 的开始状态合并，也就是 $N(t)$ 开始状态的所有转换成为 $N(s)$ 的接受状态的转换。合并后的这个状态不作为合成后的 NFA 的接受状态或开始状态。从 i 到 f 的路径必须首先经过 $N(s)$，然后经过 $N(t)$，所以这种路径上的标记拼成 $L(s)L(t)$ 的串。因为没有边进入 $N(t)$ 的开始状态或离开 $N(s)$ 的接受状态，所以在 i 到 f 的路径中不存在 $N(t)$ 回到 $N(s)$ 的现象，故合成的 NFA 识别 $L(s)L(t)$。

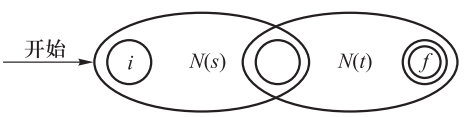


图 2.14 识别正规式 st 的 NFA

③ 对于正规式 s^*，构造合成的 NFA $N(s^*)$，结果如图 2.15 所示。同样，i 和 f 分别是新的开始状态和接受状态。在这个合成的 NFA 中，可以沿着 ε 边直接从 i 到 f，这代表 ε 属于 $(L(s))^*$，也可以从 i 经过 $N(s)$ 一次或多次。显然，这个 NFA 识别 $(L(s))^*$。

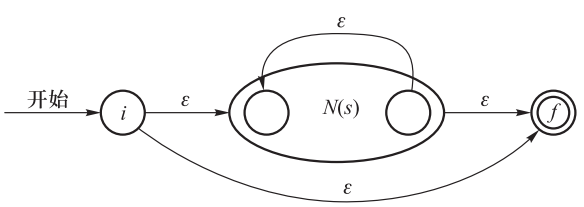


图 2.15 识别正规式 s^* 的 NFA

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第47页 文件编号: 422624 页面编号: 2474538440 打印人: 8685 2026-03-09 10:22:59

④ 对于外加括号的正规式 (s)，与 s 相对应的 NFA $N(s)$ 就作为 (s) 的 NFA。

对于每次构造的新状态都赋予不同的名字。这样，所有的状态都有不同的名字。

可以检验，算法2.2 的每一步都构造产生识别对应语言的 NFA。此外，产生的 NFA 有下列性质:

（1）$N(r)$ 的状态数最多是 r 中符号和运算符总数的两倍。因为构造的每一步最多引入两个新的状态。

（2）$N(r)$ 只有一个接受状态，接受状态没有向外的转换。

（3）$N(r)$ 的每个状态有一个用 Σ 中的符号标记的指向其他状态的转换，或者最多两个指向其他状态的 ε 转换。

【例 2.12】 用算法2.2 构造正规式 $r=(a|b)^*ab$ 的 NFA $N(r)$。

图 2.16 所示为 $(a|b)^*ab$ 的的分析树。对于其中的 $r_1=a$ 和 $r_2=b$，构造 NFA，再用选择规则组合 $N(r_1)$ 和 $N(r_2)$，得到 $r_3=r_1|r_2$ 的 NFA。(r_3) 的 NFA 和 r_3 的一样，再构造 $(r_3)^*$ 的 NFA。这样依次下去，最后得到 $r=(a|b)^*ab$ 的 NFA，如图 2.17 所示。 □

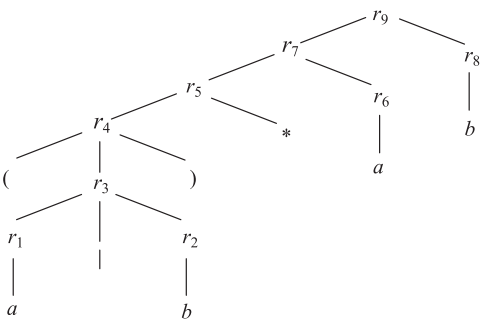


图 2.16 $(a|b)^*ab$ 的分析树

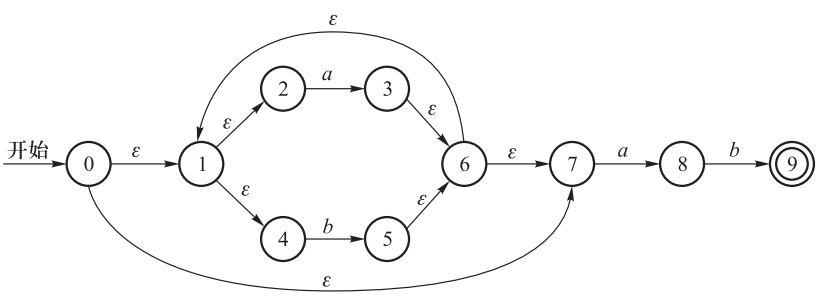


图 2.17 识别正规式 $(a|b)^*ab$ 的 NFA

从手工构造 NFA 的角度看，算法2.2 的缺点是引入大量的 ε 转换，使得后续手工将 NFA 确定化时，容易因疏忽而出错。因此在手工构造 NFA 时，应避免引入 ε 转换，例如，图 2.7

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第48页 文件编号: 422624 页面编号: 2484538448 打印人: 8685 2026-03-09 10:22:59

识别 $(a|b)^*ab$ 的 NFA 就比图 2.17 的简单得多。

2.4.2 NFA 到 DFA 的变换

对于图 2.7 的 NFA，它在状态 0 下，面临 a 时有两个转换，分别可以进入状态 0 或 1。类似地，图 2.8 的 NFA 在状态 0 下，不接受任何输入（即面临 ε）时也有两个转换。有时还会出现既可以根据 ε 又可以根据一个实际输入符号进行转换的情况，这同样会引起二义。这种转换函数多值的情况导致很难用计算机程序模拟 NFA。“接受”的定义是说必须存在从开始状态到某个接受状态的一条路径，该路径的标记正好拼成输入串。这样，在找到一条接受路径或确定没有这样的路径前，不得不同时考虑所有路径。为此，就希望将 NFA 转换为等价的 DFA。

下面介绍称为**子集构造法**的算法，它可以从 NFA 构造出识别同样语言的 DFA。首先概述算法思想。在 NFA 的转换表中，每个单元格条目是一个状态集；在 DFA 的转换表中，每个条目最多只有一个状态。从 NFA 构造等价 DFA 的一般思想是让新构造的 DFA 的每个状态对应该 NFA 的一个状态集，即这个 DFA 在读取输入 $a_1a_2\cdots a_n$ 后到达的状态，对应该 NFA 从开始状态沿着 $a_1a_2\cdots a_n$ 的路径能到达的所有状态的集合。显然，DFA 的状态数与对应的 NFA 的状态数之间的差距是是呈指数级的，但实际上，这种最坏情况很少发生。

算法 2.3 从 NFA 到 DFA 的子集构造法。

输入：一个 NFA N。

输出：一个接受同样语言的 DFA D。

方法：为 D 构造转换表 Dtran，表中的每个状态是 N 的状态集合，D “并行”地模拟 N 对于输入串的所有可能的移动。

表 2.6 定义的 3 个函数描述了在 N 状态上的基本运算，它们用在图 2.18

的算法中。注意 s 代表 N 的单个状态，T 代表 N 的一个状态集合。如图 2.18 所示的子集构造法，N 在读取第一个符号前可以处于集合 ε-closure(s_0) 的任何状态，其中 s_0 是 N 的开始状态。假定集合 T 是从 s_0 出发，面临某个输入串所能到达的所有状态的集合，令 a 是下一个输入符号，那么看见 a 时，N 可以移动到集合 move(T,a) 中的任意一个状态。由于允许 ε 转换，看见 a 后，N 可以处于 ε-closure(move(T,a)) 中的任意一个状态。

表 2.6 在 NFA 状态上的运算

运算	描述
ε-closure(s)	从状态 s 出发，只用 ε 转换能到达的状态集合
ε-closure(T)	状态集合 $\cup_{t\in T}\varepsilon$-closure(t)
move(T,a)	状态集合 $\cup_{t\in T}$move(t,a)(move 被拓展成多态函数)

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第49页 文件编号: 422624 页面编号: 2494538446 打印人: 8685 2026-03-09 10:22:59


```
初始时,ε-closure(s0)是 Dstates 仅有的状态,并且尚未标记;
while (Dstates 有尚未标记的状态 T) {
    标记 T;
    for (每个输入符号 a) {
        U = ε-closure(move(T, a));
        if (U 不在 Dstates 中)
            把 U 作为尚未标记的状态加入 Dstates;
        Dtran[T, a] = U;
    }
}
```

图 2.18 子集构造法

```
把 T 的所有状态压入栈;
ε-closure(T)的初值置为 T;
while (栈非空) {
    把栈顶元素 t 弹出栈;
    for (满足从 t 到 u 存在标记为 ε 的每个状态 u)
        if (u 不在 ε-closure(T)中) {
            把 u 加入 ε-closure(T);
            把 u 压入栈;
        }
}
```

图 2.19 ε-closure(T) 的计算

按上述子集构造法可以构造出 D 的状态集合 Dstates 和转换表 Dtran。D 的开始状态是 ε-closure(s_0)。如果 D 的某个状态 T 所对应的 N 状态集至少含 N 的一个接受状态，那么 T 是 D 的一个接受状态。

ε-closure(T) 的计算是从给定结点集合出发，搜索可达结点的过程。该计算只包含 NFA 中含 ε 标记的边，T 是给定的结点集合。如图 2.19 所示，计算 ε-closure(T) 时，可以用栈来保存那些还没有完成 ε 转换边检查的状态。

【例 2.13】 图 2.17 是接受语言 $(a|b)^*ab$ 的另一个 NFA N，它是按算法2.2 机械性构造出来的 NFA。对 N 执行算法2.3，其等价的 DFA 的开始状态是 ε-closure(0)，记为 $A = \{0,1,2,4,7\}$，因为它们正好是从状态 0 出发的，经过标记都是 ε 的路径所能到达的所有状态。由于路径可以没有边，所以 0 也是经这样的路径从 0 能到达的状态。

这里的字母表 $\Sigma = \{a,b\}$。根据图 2.18 的算法，首先标记 A，然后计算 ε-closure(move(A,a))。由于在 A={0, 1, 2, 4, 7} 中，只有状态 2 和 7 能发生 a 转换，分别转换为状态 3 和 8，因此

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第50页 文件编号: 422624 页面编号: 2504538444 打印人: 8685 2026-03-09 10:22:59

move$(A, a) = \{3, 8\}$。所以

$$\varepsilon\text{-closure}(\text{move}(A, a)) = \varepsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$$

后一步的结果是因为 ε-closure$(3) = \{1, 2, 3, 4, 6, 7\}$ 并且 ε-closure$(8) = \{8\}$。称这个集合为 B，于是，Dtran$[A, a] = B$。

在集合 A 中，只有状态 4 能发生 b 转换并到达状态 5，所以状态 A 的 b 转换并到达

$$\varepsilon\text{-closure}(\text{move}(A, b)) = \varepsilon\text{-closure}(\{5\}) = \{1, 2, 4, 5, 6, 7\}$$

令该集合为 C，于是，Dtran$[A, b] = C$。

用尚未标记的新集合 B 和 C 继续这个过程，最终会达到：所有的集合（DFA 的所有状态）都已标记。因为 10 个状态的集合的不同子集只有 2^{10} 个，一个集合一旦标记就永远是标记的，所以必定可以结束计算。本例实际构造出的 4 个不同状态集合为

$$A = \{0, 1, 2, 4, 7\}$$
$$B = \{1, 2, 3, 4, 6, 7, 8\}$$
$$C = \{1, 2, 4, 5, 6, 7\}$$
$$D = \{1, 2, 4, 5, 6, 7, 9\}$$

状态 A 是开始状态，状态 D 是仅有的接受状态，完整的转换表 Dtran 如表 2.7 所示。

表 2.7 DFA 的转换表 Dtran

状态	输入符号	
	a	b
A	B	C
B	B	D
C	B	C
D	B	C

这个 DFA 的转换图如图 2.20 所示。必须注意，图 2.10 的 DFA 也接受 $(a|b)^*ab$，并且少一个状态。2.4.3 节将讨论 DFA 的化简问题。

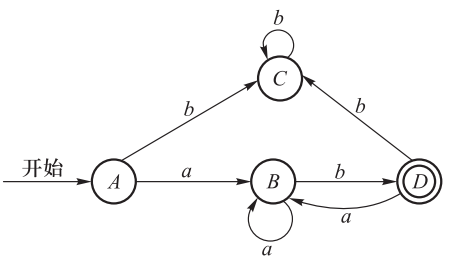


图 2.20 子集构造法用于图 2.17 的结果

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第51页 文件编号: 422624 页面编号: 2514538442 打印人: 8685 2026-03-09 10:22:59

2.4.3 DFA 的化简

理论上，每个正规集都可由一个状态数最少的 DFA 识别，这个 DFA 是唯一的（因状态名不同的同构情况除外）。本小节介绍如何把一个 DFA 化简到状态数最少，并且识别同样的语言。

这里介绍的方法要求**转换函数必须是全函数**。如果一个 DFA 的转换函数不是全函数，可以引入一个“死状态”s_d，s_d 对所有输入符号都转换到 s_d 本身。如果状态 s 对符号 a 没有转换，那么加上从 s 到 s_d 的 a 转换。显然，加入状态 s_d 后的 DFA 和原来的 DFA 等价。由此可知，DFA 也可以基于 move 函数是全函数来定义。

对于 DFA M，如果从状态 s 出发，在面临输入串 w 时，最终停在一个接受状态，而从状态 t 出发，面临 w 时，它停在一个非接受状态，或者反过来，串 w 可用来区分状态 s 和 t。如果找不到任何串来区分 s 和 t，那么 s 和 t 是不可区分的。接受状态和非接受状态可用空串 ε 来区分。在图 2.20 中，A 和 B 可由输入符号 b 来区分，因为面对输入 b，A 转换到非接受状态 C，而 B 转换到接受状态 D。

极小化 DFA 状态数的算法是把它的状态分成一些不相交的子集，每个子集的状态都是不可区分的，不同子集的状态都是可区分的。每个子集合并成一个状态。

首先，将 DFA 的状态划分成两个子集，即接受状态子集和非接受状态子集，因为它们可用空串来区分。然后，检查每个子集，看其中的状态是否还可区分。例如，对状态子集 $A = \{s_1, s_2, \cdots, s_k\}$ 和输入符号 a，检查 $s_1, s_2, \cdots, s_k$ 面临 a 的转换，如果这些转换所到的状态落入当前划分的两个或更多的状态子集中，那么 A 必须进一步划分，使得 A 的每个子集面临 a 的转换能落入当前划分的某一个状态子集中。例如，若 s_1 和 s_2 的 a 转换分别到达 t_1 和 t_2，并且 t_1 和 t_2 在当前划分的不同子集中，那么 A 至少要分成两个子集，一个含 s_1，另一个含 s_2。注意，如果 t_1 和 t_2 是可由某个串 w 区分的，那么 s_1 和 s_2 一定是可由串 aw 区分的。

重复上面的过程，对当前划分进一步细分，直到没有任何一个子集再需细分为止。以上在说明为什么分在不同子集中的状态是可区分的时候，并没有说明最终留在同一个子集中的状态是不能由任何输入串来区分的，这个证明留给有兴趣的读者。还有另一个问题也留给有兴趣的读者，这个问题是，从最终划分的每个子集中取一个状态，滤掉死状态和从开始状态不可达的状态，所构造的 DFA 就是接受同样语言的状态数最少的 DFA。

算法 2.4 极小化 DFA 的状态数。

输入：一个 DFA M，它的状态集合是 S，输入符号集合是 Σ，转换函数是 move: $S \times \Sigma \to S$，开始状态是 s_0，接受状态集合是 F。

输出：一个 DFA M'，它和 M 接受同样的语言，且状态数最少。

方法：(1) 构造状态集合的初始划分 Π：划分两个子集——接受状态子集 F 和非接受状态子集 $S - F$。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第52页 文件编号: 422624 页面编号: 2524538440 打印人: 8685 2026-03-09 10:22:59

(2) 应用下面的过程对 Π 构造新的划分 Π_{new}：

for (Π 中的每个子集 G) {

把 G 划分成若干子集，G 的两个状态 s 和 t 在同一子集中，当且仅当对任意输入符号 a，s 和 t 的 a 转换是到 Π 的同一子集中

在 Π_{new} 中，用 G 的划分代替 G

}

(3) 若 Π_{new}=Π，则让 $\Pi_{\text{final}} = \Pi$，再执行步骤 (4)；否则，令 $\Pi = \Pi_{\text{new}}$，转到步骤 (2)。

(4) 在 Π_{final} 的每个状态子集中选一个状态代表它，这些代表就是最简 DFA M' 的状态。如果状态 s 是这样的一个代表，在 DFA M 中，若状态 s 的 a 转换到状态 t，并且 t 所在子集的代表是 r（r 可能就是 t），那么，在 M' 中，s 的 a 转换到 r。包含 s_0 的状态子集的代表是 M' 的开始状态，M' 的接受状态是那些原先属于 F 集合的代表。注意，Π_{final} 的每个子集或者仅含 F 中的状态，或者不含 F 中的状态。

(5) 去掉 M' 中的“死状态”和从开始状态不可达的状态（如果存在的话）。把从任何其他状态到“死状态”的转换都改成无定义。

注意，使用这个算法时，其输入 DFA 的状态转换函数必须是全函数，否则有可能导致得到的新 DFA 和原来的 DFA 接受的语言不同。前面介绍的增加“死状态”，其目的就是把转换函数变成全函数。

【例 2.14】 对图 2.20代表的 DFA，进行化简。

初始划分 Π 包括两个子集：接受状态子集 $\{D\}$ 和非接受状态子集 $\{A, B, C\}$。为了构造 Π_{new}，首先考虑 $\{D\}$，因为这个子集只包含一个状态，它不能再划分，所以在 Π_{new} 中仍是 $\{D\}$。然后考虑 $\{A, B, C\}$，对于输入 a，这些状态都转换到 B，但对于输入 b，A 和 C 都转换到状态子集 $\{A, B, C\}$ 的一个成员，而 B 转换到 D，是另一个子集的成员。于是，在 Π_{new} 中，状态子集 $\{A, B, C\}$ 必须分成两个新子集 $\{A, C\}$ 和 $\{B\}$，Π_{new} 成了 $\{A, C\}$、$\{B\}$ 和 $\{D\}$。

对于这些子集，只有 $\{A, C\}$ 有划分的可能。但是对于输入 a 和 b。它们都分别转换到 B 和 C，因此不必再划分，即 Π_{new}=Π。所以，Π_{final} 是 $\{A, C\}$、$\{B\}$ 和 $\{D\}$。

如要选择 A 作为 $\{A, C\}$ 的代表，选择 B 和 D 作为其他单状态子集的代表，可以得到最简自动机。它的转换表如表 2.8 所示，状态 A 是开始状态，状态 D 是唯一的接受状态。

表 2.8 最简 DFA 的转换表

状态	输入符号	
	a	b
A	B	A
B	B	D
D	B	A

例如，在这个最简自动机中，D 面临 b 转换到 A，因为在原来的自动机中，D 面临 b 转

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第53页 文件编号: 422624 页面编号: 2534538448 打印人: 8685 2026-03-09 10:22:59

换到 C，并且 A 是 C 所在子集的代表。类似的变化也发生在状态 A 面临输入 b 时。其余的转换与图 2.20 所示相同。该图没有“死状态”，并且所有的状态都可以从开始状态 A 到达。

2.5 词法分析器的生成器

本节描述广泛使用的词法分析器的生成工具 LexLex（a lexical analyzer generator）。最早由美国贝尔实验室的迈克尔·E·莱斯克（Michael E. Lesk）和埃星克·施密特（Eric Schmidt）共同开发，于 1975 年发布 [6]。它从一组正规式描述的模式规则生成高效的词法分析器。Lex 有许多变种和替代品以适应不同的编程语言和需求。代表性的有 1987 年由弗恩·帕克森（Vern Paxson）开发推出的面向 C 语言的 Flex（fast lexical analyzer generator），1998 年又推出的为 Java 语言设计的 JFlex，以及 1992 年由特伦斯·帕尔（Terence Parr）等研制推出的 ANTLR [7] 包含的词法功能（http://jflex.de/）。本节主要介绍 Flex 和 ANTLR。

通过本节的介绍读者可以了解要求词法分析器完成的动作（例如，在符号表中增加新条目），如何与基于正规式的模式说明组织在一起形成词法分析器的规范。即使不用 Lex 的情况下，这样的规范仍然有用，因为可以根据 2.2 节的转换图技术手工构造出词法分析器。

2.5.1 词法分析器的生成器 Lex

Lex 通常按图 2.21(a) 描绘的方式使用。首先，用 Lex 语言将词法分析器的规范表达在文件（如 lex.l，扩展名也可以改为 lex) 中。在 Linux 下用 Flex 命令

```
flex lex.l
```

把词法规范 lex.l 翻译成 C 语言文件 lex.yy.c。可以用“`flex --help`”查看 flex 支持的命令行选项。程序 lex.yy.c 包括从 lex.l 的正规式构造出的转换图（用表格形式表示）和使用这张转换图识别词法单元的标准子程序。在 lex.l 中，和正规式相关联的动作是用 C 语言的代码表示的，它们被直接复制到 lex.yy.c 中。然后，用 C 编译器（这里的 cc 可以是 gcc 或 clang）命令

```
cc lex.yy.c -ll
```

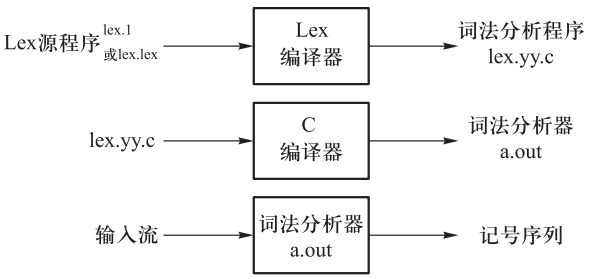
编译 lex.yy.c，其中选项-ll（现代 flex 为-lfl）用于链接 flex 生成的词法分析器所依赖的库 libl.a（或 libfl.a）。编译结果默认为目标程序 a.out，它就是把输入串识别成记号序列的词法分析器。

图 2.21(b) 示意了用 JFlex 或 ANTLR 从词法规范生成分析程序的简图。ANTLR 可以生成多种语言的分析程序，应用范围更广，2.5.2 节将简要介绍 ANTLR 支持的 Lex 能力。

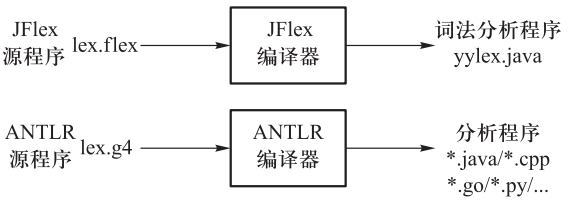
用 Lex 语言描述的词法规范（也称为 Lex 程序）包含以下 3 个部分：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第54页 文件编号: 422624 页面编号: 2544538446 打印人: 8685 2026-03-09 10:22:59


```
声明
%%
翻译规则
%%
辅助函数
```



(a) Lex 建立词法分析器的一般过程



(b) 用 JFlex 或 ANTLR 生成分析程序

图 2.21 用 Lex 建立词法分析器

第一部分是声明，包括变量声明、常量定义和正规定义。这里的正规定义和 2.2 节介绍的方法类似，可以作为正规式出现在翻译规则中。

第二部分为翻译规则，形式如下：

```
模式  {动作}
```

其中，每个模式是一个正规式，每个动作描述该模式匹配词法单元时，词法分析器应执行的程序段。在 Lex 中，动作是用 C 语言编写的代码段。现在也有很多使用其他语言的 Lex 变种。

第三部分包括动作所用到的辅助函数。这些函数也可以写在其他文件中，分别编译，然后再和这里的词法分析器进行连接。

由 Lex 建立的词法分析器通常作为语法分析器的一个子程序。每次被语法分析器调用时，词法分析器逐个字符地读取剩余输入，直到在剩余输入中发现能和某个模式 P 匹配的最长前缀为止，即遵循**最长匹配**原则。然后执行与模式 P 对应的动作 A。通常，动作 A 会把控制返回语法分析器。如果不是这样，如 P 描述空白或注释时，则词法分析器会继续寻找后继的词法单元，直到有一个动作引起控制回到语法分析器为止。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第55页 文件编号: 422624 页面编号: 2554538443 打印人: 8685 2026-03-09 10:22:59

词法分析器仅返回记号名给语法分析器，记号的属性值则通过由 Yacc（见 3.8 节）生成的语法分析器共享的全局变量 yylval 传递。在下面的例子中，yylval 的类型简单地按 int 类型处理。实际应用时，yylval 的类型是 Yacc 生成的语法分析器中定义的 YYSTYPE 类型。

```
%{
/*  tokens.h包含常量LT, LE, EQ, NE, GT, GE, WHILE, DO, ID, NUMBER, RELOP的定义，它们用C
    的#define方式来写。*/
#include "tokens.h"
int addSymbol();
int addNumber();
%}
/* 正规定义 */
delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?

%%
{ws}        {/* 没有动作，也不返回 */}
while       {return (WHILE);}
do          {return (DO);}
{id}        {yylval = installId (); return (ID);}
{number}    {yylval = installNum(); return(NUMBER);}
"<"         {yylval = LT; return (RELOP);}
"<="        {yylval = LE; return (RELOP);}
"="         {yylval = EQ; return (RELOP);}
"<>"        {yylval = NE; return (RELOP);}
">"         {yylval = GT; return (RELOP);}
">="        {yylval = GE; return (RELOP);}
%%

int installId() {
    /*  把词法单元装入符号表并返回指向它的指针。yytext指向该词法单元的第一个字符，yyleng给出
        它的长度 */
    return addSymbol(yytext, yyleng);
}
int installNum () {
    /* 类似上面的过程，但词法单元不是标识符而是数，并且把它放到数表中 */
    return addNumber(atof(yytext));
}
```

图 2.22　识别表 2.4记号的 Lex 程序

【例 2.15】　图 2.22是识别表 2.4 中记号的 Lex 程序。可以从此程序观察到 Lex 的一些重要特征。

声明部分定义的头文件包含函数声明等，由%{和%} 括起，出现在括号中的所有内容都被直接复制到词法分析器 lex.yy.c 中，不作为正规定义和翻译规则的一部分。tokens.h 中包含通过宏定义`#define`声明的符号常量，在动作代码中会使用到这些常量。第三部分中的辅助函数定义，如图 2.22 中的 installId() 和 installNum()，也被直接复制到 lex.yy.c 中。Lex 并不检查被复制内容的合法性。

声明部分还包括一些正规定义，每个定义由一个名字和该名字指示的正规式组成。例如，第一个定义的名字是 delim，它代表字符类 [\ t\ n]，也就是空格、制表 (\ t) 和换行 (\ n) 这 3 个字符中的任意一个。第二个是空白定义，由名字 ws 表示，空白是一个或多个分界符

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第56页 文件编号: 422624 页面编号: 2564538441 打印人: 8685 2026-03-09 10:22:59

的序列。

在第五个定义 id 中使用了圆括号，它们是 Lex 的元符号，其含义是把一些正规式组成一个整体。同样地，竖线“|”在 Lex 中表示选择。反斜线作为转义字符，让作为 Lex 元符号的字符取其自身原来的含义。

在 number 的正规定义中，十进制小数点由\.表示，因为在 Lex 及许多处理正规式的 UNIX 或 Linux 程序中，点可以代表除换行符以外的任一字符。在字符类[+\-]中，减号的前面放了反斜线，因为减号也是元符号。减号可以用于表示范围，如 [A-Z]。

还可以把元符号放在引号中，用于表示原来的含义。在翻译规则部分有这样的例子，6 个关系运算符都由引号括注。

在翻译规则中，第一条规则表明，如果看见 ws，也就是空格、制表和换行符组成的串，则忽略之，也不将控制返回分析器。词法分析器将继续识别记号，直至所识别记号的动作引起返回为止。注意，在 Lex 中，ws 必须由花括号括注，以区别由两个字母“ws”组成的模式，下面的 id 和 number 同理。

第二条规则表明，如果看见字母 while，则返回记号 WHILE，它是代表某个整数的常量（在 tokens.h 中声明），语法分析器把这个整数理解为 while。下面一条规则以同样的方式处理 do。

在 id 的规则中，其动作含两个语句。第一个语句调用函数 installId()，该函数定义在最后，该函数的返回值赋给变量 yylval。变量 yylval 定义在 Lex 的输出 lex.yy.c 中，它的作用是保存返回记号的属性值。动作的第二个语句 return(ID) 只返回记号名。

图 2.22 中 installId() 函数调用 addSymbol(yytext, yyleng)，后者在符号表中查找当前识别出的词法单元（其首字符由指针变量 yytext 指向，长度保存在整型变量 yyleng 中）。注意，这里的 installId() 和 2.2 节介绍的 installId() 在功能上有区别，因为在这里关键字是用另外的正规式定义的。通过变量 yytext 和 yyleng，Lex 使得词法单元的字面量对函数是可用的。

再下一条规则以类似方式处理数。在 installNum() 函数里，首先调用 atof(yytext) 将识别出的词法单元转换成浮点数，然后调用 addNumber() 将这个浮点数登记到数表中。

最后 6 条规则都返回记号 RELOP，用 yylval 的值来区分不同的关系运算符。

如果要匹配的词法单元是 while，那么模式 while 和 {id} 都匹配这个词法单元，而且它们都不能匹配更长的串。这时应该选哪一个呢？Lex 对这个冲突的解决方法是**选择排在前面的模式**。由于图 2.22 中关键字 while 的模式先于标识符的模式，因此 while 会被识别为关键字。

如果要读的前两个字符是 <= ，这时模式 < 匹配第一个字符，而模式 <= 可以匹配更长的前缀。根据**最长匹配原则**，将选择 <= 作为下一个记号。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第57页 文件编号: 422624 页面编号: 2574538449 打印人: 8685 2026-03-09 10:22:59

2.5.2 ANTLR 中的 Lex 部分

ANTLR（another tool for language recognition）是由特伦斯·帕尔（Terence Parr）从1989年在普渡大学读研期间开始研制并持续维护演进的一款强大的分析器生成工具。其早期版本是作为 Purdue compiler construction tool set(PCCTS) [7] 的一部分发布。当前最新版本是 ANTLR 4[9]，支持生成包括 Java、C++、Go、Python、JavaScript、Rust 等多种语言的分析器，并具有良好的跨平台支持能力。本小节简要介绍 ANTLR 4 对词法分析的支持，3.4节将介绍对语法分析的支持。

1. ANTLR 文法文件的结构

在 ANTLR 中，描述语言词法或语法规范的文法文件的扩展名为 g4。.g4 文件的基本结构及含义如下：

```
grammar MyG;          指明所定义的文法名称为MyG
options { … }         设置分析器的选项
import XXX;           导入其他文法文件或模块
tokens { … }          定义自定义的词法记号
@actionName { … }     定义与特定动作相关的代码块
ruleName : <stuff> ;  定义词法或语法规则
……
```

如果在 grammar 命令左边加上 lexer，那么表示纯词法分析。ANTLR 对规则名 ruleName 有约定：词法规则名以大写字母开头，语法规则名以小写字母开头。词法规则的格式如下：

```
TokenName : 选项1|…|选项N [-> 命令名 [(参数)]];
```

如果在规则左边加上 fragment，则表示正规定义。例如：

```
fragment DIGIT : [0-9] ;  // 正规定义十进制数DIGIT，但不是记号
INT : DIGIT+ ;            // 定义整数记号INT，它引用正规名DIGIT
ESC : '\\' . ;            // 匹配任一转义字符序列，.匹配除换行符以外的任一字符
```

词法规则可以使用一些特定的命令来控制特定规则的行为。常见的命令有：

（1）skip：不为匹配的文本生成词法记号，并忽略它（不返回给语法分析器）。例如，识别出空白符：

```
WS : [ \t\r\n]+ -> skip ;
```

（2）channel(C)：将词法记号发送到指定的通道 C。如果 C 为 HIDDEN，则该通道的记号在解析时默认会被忽略。例如，下面识别出注释后发送到 HIDDEN 通道：

```
Comment : '/*' .*? '*/' -> channel(HIDDEN) ;
```

（3）type(T)：将匹配的记号的类型强制设置为指定的类型 T。例如，下面将识别出的由双引号括起的串设置为 STRING 类型：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第58页 文件编号: 422624 页面编号: 2584538447 打印人: 8685 2026-03-09 10:22:59


```
DOUBLE : '"' .*? '"'   -> type(STRING) ;
```

（4）more：指示词法分析器继续扫描更多字符，将当前匹配的字符追加到下一个记号。

（5）mode 模式定义及模式调用命令 mode(M)、pushMode(M)、popMode：这些模式命令用来改变词法分析器所处的词法模式。记号类型将是最终匹配的规则的类型（没有 more 或 skip 命令的那个规则）。例如：

```
lexer grammar Strings;
LQUOTE : '"' -> more, mode(STR) ;
WS : [ \r\t\n]+ -> skip ;
mode STR;
STRING : '"' -> mode(DEFAULT_MODE) ; // 语法分析器希望看到的 token
TEXT : . -> more ;                   // 收集更多的字符串文本
```

在该模式下遇到左双引号时，进入词法模式 STR，遇到右双引号时，识别出字符串记号 STRING，回到默认的词法模式。模式 STR 下有两条规则，STRING 和 TEXT。

2. 用 ANTLR 构造词法分析器

ANTLR 的核心是用 Java 语言实现的，默认产生分析器的 Java 源代码。2024 年 8 月发布的 4.13.2 版还支持产生分析器的 C#、Python 3、JavaScript、TypeScript、Go、C++、Swift、PHP、DART 语言代码。这里介绍如何用 ANTLR 产生分析器的 Java 源码和 C++ 源码。

首先从 ANTLR 官网下载 antlr4，安装 Java、cmake、uuid-dev、pkg-config 和最新的 ANTLR jar 包 antlr4-complete.jar 及 C++ 运行时库 antlr4-cpp-runtime-source.zip（这里忽略版本号）。

antlr4 和 grun 是如下命令的别名：

```
antlr4='java org.antlr.v4.Tool'
grun='java org.antlr.v4.gui.TestRig'
```

antlr4 根据提供的.g4 文法文件生成用于处理语言语法的分析器；grun 用于测试和运行 ANTLR 生成的解析器。下面结合例子说明使用方法。

【例 2.16】 图 2.23 给出了识别表达式语言词法记号的 ANTLR 文法文件 exprLexer.g4。执行命令

```
antlr4 exprLexer.g4 -o build
```

将生成如下 3 个文件到目录 build 中：

（1）exprLexer.java：包含词法分析器的类定义。

（2）exprLexer.tokens：包含词法记号名及其数值的定义。

（3）exprLexer.interp：包含允许运行内建解释器的数据，用于支撑 IDE 对文法的调试。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第59页 文件编号: 422624 页面编号: 2594538445 打印人: 8685 2026-03-09 10:22:59

然后在 build 目录下，对生成的 exprLexer 分析器的 Java 源程序进行编译，再调用 grun 执行 exprLexer 分析器对 <expr 程序 > 进行词法分析：

```
javac *.java
grun exprLexer tokens -tokens <expr程序>
```

如果希望生成 C++ 源码的词法分析程序，那么执行如下命令：

```
antlr4 exprLexer.g4 -Dlanguage=Cpp -o build
```

将生成的 exprLexer.cpp、exprLexer.h、exprLexer.tokens 和 exprLexer.interp 四个文件放入到 build 目录。再进入 build 目录，对生成的分析器进行编译：

```
g++ -std=c++17 exprLexer.cpp -I../include/antlr4-runtime -I. -lantlr4-runtime
```

注意，-I 和-l 选项指定了所依赖的 antlr4-runtime 头文件和库，这些文件和库需要事先从 antlr4-cpp-runtime-source.zip 文件中解压出来，然后进行编译和安装。

```
1  lexer grammar exprLexer;
2  tokens {
3      LeftParen, RightParen, Plus, Minus, Multiply, Divide, IntConst, ID
4  }
5  LeftParen: '(';
6  RightParen: ')';
7  Plus: '+' ;
8  Minus: '-' ;
9  Multiply: '*' ;
10 Divide: '/' ;
11 IntConst : [0-9]+ ;
12 ID: [a-zA-Z_][a-zA-Z_0-9]* ;
13 WS: [ \t\n\r\f]+ -> skip ;
```

图 2.23 一种表达式语言的 ANTLR 词法描述文件 exprLexer.g4

习 题

2.1 从下列每种语言的参考手册确定它们形成输入字母表的字符集（不包括那些只可以出现在字符串或注释中的字符）。

(a) C (b) C++
(c) C# (d) Java
(e) SQL (f) Go
(g) Python (h) Rust

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第60页 文件编号: 422624 页面编号: 2604538443 打印人: 8685 2026-03-09 10:22:59

2.2 在下面的 C 函数中，按序列出所有的记号，并给每个记号以合理的属性值。

```
long gcd(long p, long q) {
    if (p%q == 0)
        /* then part */
        return q;
    else
        /* else part */
        return gcd(q, p%q);
}
```

2.3 叙述由下列正规式描述的语言：

(a) 0 (0 | 1)* 0

(b) ((ε | 0) 1*)*

(c) (0 | 1)* 0 (0 | 1) (0 | 1)

(d) 0*10*10*10*

(e) (00 | 11)*((01 | 10) (00 | 11)*(01 | 10) (00 | 11)*)*

***2.4** 为下列语言编写正规定义：

(a) 包含 5 个元音的所有字母串，其中每个元音只出现一次且按顺序排列。

(b) 按词典序排列的所有字母串。

(c) 某语言的注释，它是以/* 开始并以 */结束的任意字符串，但它的任何前缀（本身除外）不以 */结尾。

(d) 相邻数字都不相同的所有数字串。

(e) 最多只有一处相邻数字相同的所有数字串。

(f) 由偶数个 0 和偶数个 1 构成的所有 0 和 1 的串。

(g) 由偶数个 0 和奇数个 1 构成的所有 0 和 1 的串。

(h) 所有不含子串 011 的 0 和 1 的串。

(i) 字母表 $\{a, b\}$ 上，a 不会相邻出现的所有串。

2.5 说明习题 2.1 中各种语言的数值常数的词法形式。

2.6 说明习题 2.1 中各种语言的标识符的词法形式。

2.7 给定正规式 $(a \mid b)^*ab(a \mid b)^*$，请：

(a) 叙述该正规式描述的语言；

(b) 手工构造 NFA，给出它处理输入串 bab 的状态转换序列；

(c) 手工构造 DFA，给出它处理输入串 bab 的状态转换序列；

(d) 用算法2.3 将 NFA 变换成 DFA；

(e) 用算法2.4 构造最简的 DFA。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第61页 文件编号: 422624 页面编号: 2614538441 打印人: 8685 2026-03-09 10:22:59

2.8 用算法2.2 为下列正规式构造不确定有限自动机，给出它们处理输入串 $ababbab$ 的状态转换序列。

(a) $(a \mid b)^*$

(b) $(a^* \mid b^*)^*$

(c) $((\varepsilon \mid a)b^*)^*$

(d) $(a \mid b)^*abb(a \mid b)^*$

2.9 用算法2.3 把习题 2.8 的 NFA 变换成 DFA，给出它们处理输入串 $ababbab$ 的状态转换序列。

2.10 根据表 2.4 中记号的转换图构造 DFA。

2.11 某语言的注释是以/* 开始和以 */结束的任意字符串，但它的任何前缀（本身除外）不以 */结尾。画出接受这种注释的 DFA 状态转换图。

2.12 可以从正规式的最简 DFA 同构来证明两个正规式等价。使用这种技术，证明正规式 $(a \mid b)^*$、$(a^*|b^*)^*$ 和 $((\varepsilon \mid a)b^*)^*$ 等价。

2.13 为下列正规式构造最简的 DFA。

(a) $(a \mid b)^*a(a \mid b)$

(b) $(a \mid b)^*a(a \mid b)(a \mid b)$

(c) $(a \mid b)^*a(a \mid b)(a \mid b)(a \mid b)$

2.14 构造一个 DFA，它接受 $\Sigma = \{0, 1\}$ 上 0 和 1 的个数都是偶数的字符串。

2.15 构造一个 DFA，它接受 $\Sigma = \{0, 1\}$ 上能被 5 整除的二进制数。

2.16 构造一个最简的 DFA，它接受所有大于 101 的二进制整数。

2.17 修改算法2.2, 使之尽可能少用 ε 转换，并保持所产生的 NFA 只有一个接受状态。

2.18 若 L 是正规语言，证明下面的 L' 语言也是正规语言。L' 语言的定义是

$$L' = \{x \mid x^R \in L\}$$

其中，x^R 表示 x 的逆。

2.19 一个 C 语言编译器编译下面的 gcd() 函数时，报告`expected ';' before 'else'`。这是因为else的前面少了一个分号。但是如果第一个注释

```
/* then part */
```

误写成

```
/* then part
```

那么该编译器发现不了遗漏分号的错误。这是为什么？

```
long gcd(long p, long q) {
    if (p%q == 0)
        /* then part */
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第62页 文件编号: 422624 页面编号: 2624538449 打印人: 8685 2026-03-09 10:22:59


```
        return q
    else
        /* else part */
        return gcd(q, p%q);
}
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第63页 文件编号: 422624 页面编号: 2634538447 打印人: 8685 2026-03-09 10:22:59

第 3 章

语 法 分 析

每种编程语言都有一些规则来描述**良形**（well-formed）程序的语法结构。例如，C 语言的程序由若干函数组成，函数由若干声明和语句组成，语句包括若干表达式等。这些规则可以用上下文无关文法或 BNF（Backus-Naur form，巴克斯–诺尔范式）表示法来描述。

编译器的语法分析方法主要有自上而下和自下而上两种。自上而下分析方法是从根结点构建分析树，而自下而上分析方法是从叶结点开始构建分析树。两者都是从左向右扫描输入，每次处理一个符号。有效的自上而下分析法（如 LL 分析法）和自下而上分析法（如 LR 分析法）都只能处理上下文无关文法的子类。这些子类足以描述编程语言的大多数语言构造（constructs）和它们的语法特征。

图 3.1展示了**语法分析器**（syntax analyzer）在编译器中的位置，语法分析器简称**分析器**或**解析器**（parser）。解析器从词法分析器接收记号流，检查这些记号是否符合源语言的文法规则，并输出分析树的某种表示。此外，解析器应能以易于理解的方式报告语法错误，并在发现语法错误时，从错误状态中恢复过来，继续分析剩余的程序。在语法分析过程中，还可以同时完成其他任务，如把各种记号的信息存入符号表中，进行类型检查和其他语义检查，并产生中间代码。这些任务都属于图 3.1 中“前端的其余部分”，将在后续章节中详细讨论。图 3.1 中的虚线表示分析树仅为概念性的，实际中不一定要真正生成。

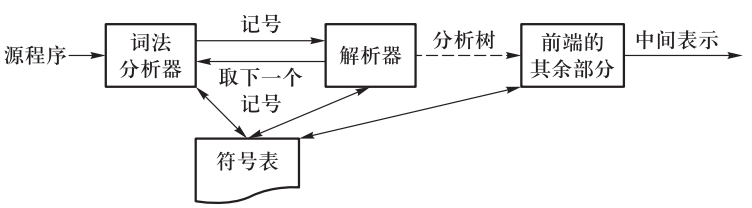


图 3.1　解析器在编译器中的位置

本章首先介绍语言语法的描述方法——上下文无关文法，然后介绍适合手工实现的预测分析技术，最后给出自动工具用的 LR 分析算法。由于程序员编写代码时常出现语法错误，本章还扩展了所介绍的分析方法，使其能从常见的错误中恢复过来。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第64页 文件编号: 422624 页面编号: 2644538445 打印人: 8685 2026-03-09 10:22:59

3.1 上下文无关文法

本节首先从正规式的表达能力不足引出上下文无关文法及其定义；然后比较正规式与上下文无关文法，说明分离词法分析器的原因；最后介绍推导、解析树和文法二义性等概念。

3.1.1 正规式的表达能力不足

第 2 章介绍了使用正规式定义一些简单的语言，这些语言称为正规语言。然而，正规式在描述复杂语言时存在局限性，例如无法描述配对或嵌套结构。具体而言，正规式无法描述如配对括号构成的串的集合或语句的嵌套结构。此外，正规式对某些重复串的表达能力也有限，例如集合

$$\{wcw \mid w\text{是由}a\text{和}b\text{组成的串，字母表}\Sigma=\{a,b,c\}\}$$

不能用正规式描述。正规式只能表示给定结构的固定次数的重复或者不指定次数的重复。

【例 3.1】 语言 $L=\{a^nb^n \mid n\geqslant 1\}$ 不能用正规式描述。

证明 (用反证法)：假定 L 可以由某个正规式描述，那么就可以构造一个接受 L 的 DFA D，它包含 k 个有限的状态。假设 D 读完 $\varepsilon, a, aa, \cdots, a^k$ 分别到达状态 $s_0, s_1, \cdots, s_k$，即 D 读 i 个 a 后到达状态 $s_i(0\leqslant i\leqslant k)$。

由于 D 只有 k 个不同的状态，那么在状态序列 $s_0s_1\cdots s_k$ 中至少有两个状态相同，假设是 s_i 和 $s_j(0\leqslant i<j\leqslant k)$。从状态 s_i 出发，D 可以接受 i 个 b 到达接受状态 f，因为 a^ib^i 属于 L。同时，D 还存在一条从状态 s_0 到状态 s_i 再到 f 的路径，该路径上的标记为 a^jb^i，如图 3.2 所示。于是，D 也接受 a^jb^i，但它不在 L 中，这与 D 接受的语言是 L 的假设相矛盾。

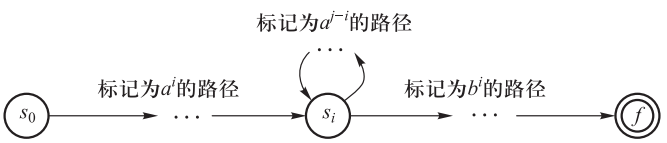


图 3.2 接受 a^ib^i 和 a^jb^i 的 DFA D

通俗地说，有限自动机不能计数，即有限自动机不能接受像 L 这样的语言，因为它要求在接受 b 之前能数出 a 的个数。

3.1.2 上下文无关文法的定义

本节定义描述功能比正规式更强的上下文无关文法，并介绍一些与分析有关的术语。

形式上说，一个**上下文无关文法** G 是一个四元组 (V_T, V_N, S, P)，其中：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第65页 文件编号: 422624 页面编号: 2654538442 打印人: 8685 2026-03-09 10:22:59

（1）V_T 是一个非空有限集合，其元素称为**终结符**。在谈论编程语言的文法时，记号名是终结符的同义词。

（2）V_N 是一个非空有限集合，其元素称为**非终结符**，并有 $V_T \cap V_N = \varnothing$。在下面的例 3.2 中，*expr* 和 *op* 是非终结符。非终结符用于定义由文法规定的语言。非终结符之间的关系构成了语言中的层次结构，这种层次结构对语法分析和翻译都是有用的。

（3）S 是一个非终结符，称为**开始符号**，它定义的终结符串集就是文法定义的语言。

（4）P 是**产生式**的有限集合，每个产生式的形式是 $A \to \alpha$（有些书上用 ::= 代替箭头），其中 $A \in V_N$, $\alpha \in (V_T \cup V_N)^*$。开始符号至少出现在某个产生式的左部。产生式指明了终结符和非终结符组成串的方式。

【例 3.2】 文法 $(\{\mathbf{id}, +, *, -, (,)\}, \{\text{expr}, \text{op}\}, \text{expr}, P)$ 定义了包含加、乘和一元减的算术表达式。P 由下列产生式组成：

$$\begin{array}{ll} \text{expr} \to \text{expr op expr} & \text{expr} \to \mathbf{id} \\ \text{expr} \to (\text{expr}) & \text{op} \to + \\ \text{expr} \to -\text{expr} & \text{op} \to * \end{array}$$

为了表示上的简洁，本书将采用下列约定来表示文法。

（1）下列符号是终结符：

① 字母表前面的小写字母，如 a、b、c；

② 黑体串，如 **id** 或 **while**；

③ 数字 0, 1, ⋯, 9；

④ 标点符号，如分号、逗号等；

⑤ 运算符号，如 +、− 等。

（2）下列符号是非终结符：

① 字母表前面的大写字母，如 A、B、C；

② 字母 S，并且它通常代表开始符号；

③ 小写字母组成的名字，如 expr 和 stmt。

（3）字母表后面的大写字母，如 X、Y 和 Z，代表文法符号，即非终结符或终结符。

（4）字母表后面的小写字母，如 u、v、⋯、z，代表终结符号串。

（5）小写希腊字母，如 α、β 和 γ，代表文法的符号串。

（6）如果 $A \to \alpha_1, A \to \alpha_2, \cdots, A \to \alpha_k$ 是所有以 A 为左部的产生式（称它们为 A 产生式），则可以把它们按 BNF 表示法写成 $A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$ 的形式，称该产生式的右部 $\alpha_1, \alpha_2, \cdots, \alpha_k$ 是 A 的**选择**（或**选项**或**产生式分支**）。

有了上面的这些约定后就可以直接用产生式集合代替四元组来描述文法。此时，第一个产生式左部的符号就是文法开始符号。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第66页 文件编号: 422624 页面编号: 2664538440 打印人: 8685 2026-03-09 10:22:59

【例 3.3】 使用上述简写，例 3.2 的文法可以重新表示如下：

$$\begin{aligned} E &\to EAE \mid (E) \mid -E \mid \mathbf{id} \\ A &\to + \mid * \end{aligned}$$

其中，E 和 A 都是非终结符，E 是开始符号，其余符号都是终结符。

巴科斯–诺尔范式 (Backus-Naur form，BNF) 由约翰·巴科斯 (John Backus) 和彼得·诺尔 (Peter Naur) 发明，是描述编程语言语法的基础，在编译器设计等领域具有重要影响。

约翰·巴科斯（1924—2007 年）是美国著名计算机科学家，因领导开发 FORTRAN 编程语言和提出 BNF 而获得 1977 年图灵奖。他早年就读于弗吉尼亚大学学习化学，但因出勤率低而被开除，随后入伍服役。战后，他进入哥伦比亚大学，分别于 1949 年和 1950 年获得数学学士和硕士学位。之后，他加入 IBM 公司，担任 FORTRAN 编程语言开发团队的负责人。在 1977 年的图灵奖获奖演讲中，他发表了题为 *Can Programming Be Liberated from the von Neumann Style?* 的演说，探讨如何摆脱冯·诺依曼架构的编程模式，对编程范式的发展产生了深远影响。

彼得·诺尔（1928—2016 年）是丹麦计算机科学家，因参与开发 ALGOL 60 语言而闻名。虽然 BNF 最早由约翰·巴科斯提出，但他对其进行了简化和完善，并在 ALGOL 60 的报告中使用了这种表示法。Naur 因这两项重要贡献获得了 2005 年图灵奖。他早年在哥本哈根大学获得天文学学士（1949 年）和博士学位（1957 年），曾于 1950—1951 年在剑桥大学学习，尽管因天气恶劣影响了天文观测计划，但他却利用这段时间编程来解决天文学中的扰动问题，这也促使他毕业后转向计算机科学领域。在 2005 年图灵奖的获奖演讲 *Computing vs. Human Thinking* 中，他探讨了计算与人类思维的不同。

3.1.3 正规式与上下文无关文法的比较

正规式与上下文无关文法都能描述语言，但是后者描述能力更强。凡是能用正规式描述的语言（称为正规语言），都能用上下文无关文法来描述。例如，正规式

$$(a \mid b)^* ab$$

和上下文无关文法

$$\begin{aligned} A_0 &\to aA_0 \mid bA_0 \mid aA_1 \\ A_1 &\to bA_2 \\ A_2 &\to \varepsilon \end{aligned}$$

描述同样的语言。

可以机械地把一个非确定的有限自动机变换成一个上下文无关文法，它产生的语言和这个自动机识别的语言相同。上述文法是从图 2.7 的 NFA 用下列规则构造的。

首先确定终结符集合，它是 NFA 的字母表。其次为 NFA 的每个状态 i 引入非终结符 A_i，其中 A_0 是开始符号（因为 0 是开始状态）。如果状态 i 有一个 a 转换到状态 j，则引入产生式 $A_i \to aA_j$；如果是 ε 转换，则引入 $A_i \to A_j$。如果 i 是接受状态，就再引入 $A_i \to \varepsilon$。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第67页 文件编号: 422624 页面编号: 2674538448 打印人: 8685 2026-03-09 10:22:59

3.1.4 分离词法分析器的理由

既然正规语言都是上下文无关语言，那么为什么要用正规式定义语言的词法呢？其理由如下：

（1）语言的词法规则简单，无需用功能更强的上下文无关文法。

（2）对于词法记号，正规式给出的描述比上下文无关文法给出的描述更简洁易懂。

（3）从正规式自动构造出的词法分析器比从上下文无关文法构造出的更高效。

词法分析与语法分析的分离还有如下好处：

（1）将语法结构分成词法和非词法部分，可以简化词法分析器设计，便于编译器前端模块化。

（2）提高编译效率，专门的词法分析器能加快读字符和处理记号的速度。

（3）增强编译器的可移植性，输入字符集的特殊性等问题可限制在词法分析器中处理。

通常，正规式用于描述标识符、常数和关键字等词法结构，而上下文无关文法适合处理复杂的语法结构，如括号配对、begin 和 end 配对、语句嵌套、表达式嵌套等。

将词法分析直接并入语法分析会增加文法和分析器的复杂性，因为文法需包含处理注释和空白的规则，而解析器需处理这些内容，这比在词法分析器中过滤后的解析要复杂得多。因此，编译器通常将词法分析和语法分析分离，但在同一遍（pass）中进行，以提高效率。

3.1.5 推导

为描述文法定义的语言，需要使用推导的概念。推导是指把产生式看成重写规则，把符号串中的非终结符用其产生式右部的串来代替。例如，对于下面的算术表达式文法：

$$E \to E+E \mid E*E \mid (E) \mid -E \mid \mathbf{id} \tag{3.1}$$

产生式 $E \to E+E$ 表示两个表达式相加仍然是表达式。该产生式允许用 $E+E$ 代替 E 的任何出现，从而由简单的表达式产生更复杂的表达式。如果用 $E+E$ 代替单个 E，这个动作可以用式子

$$E \Rightarrow E+E$$

来描述，读做“E 推导出 $E+E$”。产生式 $E \to (E)$ 表示 E 的任何出现都可以用文法符号串 (E) 来代替，如 $E*E \Rightarrow (E)*E$ 或 $E*E \Rightarrow E*(E)$。

从开始符号 E 出发，不断使用产生式，可以得到一个代换序列，如：

$$E \Rightarrow E+E \Rightarrow \mathbf{id}+E \Rightarrow \mathbf{id}+\mathbf{id}$$

这个代换序列称为从 E 到 $\mathbf{id}+\mathbf{id}$ 的**推导**，这个推导表明了串 $\mathbf{id}+\mathbf{id}$ 是表达式的实例。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第68页 文件编号: 422624 页面编号: 2684538446 打印人: 8685 2026-03-09 10:22:59

如果 $A \to \gamma$ 是产生式，α 和 β 是文法的任意符号串，那么有 $\alpha A\beta \Rightarrow \alpha\gamma\beta$。如果 $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$，就称 α_1 **推导出** α_n。符号 $\Rightarrow$ 表示“一步推导”，符号 $\Rightarrow^*$ 表示“零步或多步推导”。于是

（1）对任何串 α，有 $\alpha \Rightarrow^* \alpha$，并且

（2）如果 $\alpha \Rightarrow^* \beta$，$\beta \Rightarrow \gamma$，那么 $\alpha \Rightarrow^* \gamma$。

类似地，用 $\Rightarrow^+$ 表示“一步或多步推导”。

对于开始符号为 S 的文法 G，可以用 $\Rightarrow^+$ 来定义 G 产生的语言 $L(G)$, $L(G)$ 的串仅包含 G 的终结符。终结符串 w 在 $L(G)$ 中，当且仅当 $S \Rightarrow^+ w$。若串 w 是 $L(G)$ 的句子，则 w 也是 G **的句子**。如果 $S \Rightarrow^* \alpha$，α 可能含非终结符，则称 α 为 G 的**句型**。句子是只含终结符的句型。

由上下文无关文法产生的语言称为**上下文无关语言**。如果两个文法产生同样的语言，则称这两个文法**等价**。

【例 3.4】 串 $-(\mathbf{id}+\mathbf{id})$ 是文法（3.1）的句子，因为存在推导：

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id}) \tag{3.2}$$

出现在这个推导中的 $E, -E, -(E), \cdots, -(\mathbf{id}+\mathbf{id})$ 都称为这个文法的句型。

按推导长度进行归纳可以证明，对于文法（3.1），其句子是由二元运算符“+”和“*”、一元运算符“−”、括号和运算对象 **id** 组成的算术表达式。反过来，按算术表达式长度进行归纳，可以证明，这样的算术表达式都可以由此文法产生。于是文法（3.1）恰好产生这样的算术表达式集合。

如果在推导过程中出现的句型有两个或多个非终结符，那么就需要决定下一步推导代换哪个非终结符。例如，例 3.4 的推导在得到 $-(E+E)$ 后，可以继续进行：

$$-(E+E) \Rightarrow -(E+\mathbf{id}) \Rightarrow -(\mathbf{id}+\mathbf{id}) \tag{3.3}$$

推导（3.3）两步非终结符代换所用的 E 的选择和推导（3.2）一样，但有不同的代换次序。

为了理解解析器的工作原理，需要考虑每一步都是代换句型中最左边非终结符的推导，这样的推导称为**最左推导**。若 $\alpha \Rightarrow \beta$ 是最左推导，则可写成 $\alpha \Rightarrow_{\mathrm{lm}} \beta$。推导（3.2）是最左推导，可以写成

$$E \Rightarrow_{\mathrm{lm}} -E \Rightarrow_{\mathrm{lm}} -(E) \Rightarrow_{\mathrm{lm}} -(E+E) \Rightarrow_{\mathrm{lm}} -(\mathbf{id}+E) \Rightarrow_{\mathrm{lm}} -(\mathbf{id}+\mathbf{id})$$

使用前面的约定，每步最左推导可写成 $wA\gamma \Rightarrow_{\mathrm{lm}} w\delta\gamma$ 的形式，其中 w 只含终结符，$A \to \delta$ 是所用的产生式，γ 是文法的符号串。为了强调 α 最左推导出 β，可写成 $\alpha \Rightarrow^*_{\mathrm{lm}} \beta$。

类似地，可以定义**最右推导**，即每步都代换句型中最右边非终结符的推导，用 $\Rightarrow_{\mathrm{rm}}$ 表示。最右推导又称为**规范推导**。

若 $S \Rightarrow^*_{\mathrm{lm}} \alpha$，则称 α 为 G 的**左句型**；若 $S \Rightarrow^*_{\mathrm{rm}} \alpha$，则称 α 为 G 的**右句型**。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第69页 文件编号: 422624 页面编号: 2694538444 打印人: 8685 2026-03-09 10:22:59

3.1.6 解析树

解析树（parse tree）是推导的图形表示。解析树上的每个分支结点都由非终结符标记，它的子结点由该非终结符本次推导所用产生式的右部中各符号从左到右依次来标记。解析树的叶结点由非终结符或终结符标记，所有这些标记从左到右构成一个句型。例如，表达式 $-(\mathbf{id}+\mathbf{id})$ 最左推导的解析树（包括推导过程中的解析树）如图 3.3 所示。

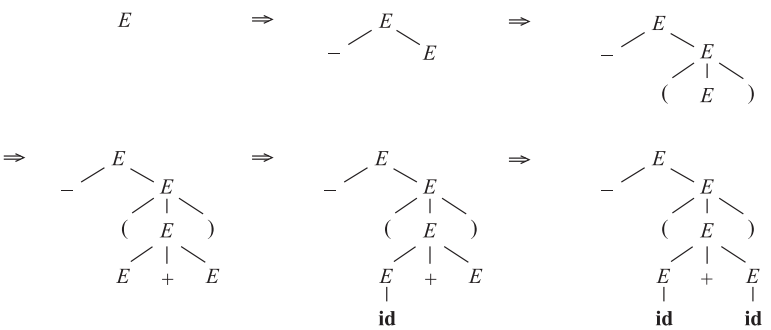


图 3.3 推导（3.2）的解析树

显然，表达式 $-(\mathbf{id}+\mathbf{id})$ 最左推导和最右推导的最终解析树是相同的，也就是该解析树未受推导次序的影响。不难看出，每棵解析树都有和它对应的最左推导和最右推导。

3.1.7 二义性

有些文法的一些句子存在不止一棵解析树，或者说这些句子存在不止一种最左（最右）推导。

【例 3.5】 考虑算术表达式文法（3.1），句子 $\mathbf{id}*\mathbf{id}+\mathbf{id}$ 的两种不同的最左推导如下：

$$
\begin{aligned}
E &\Rightarrow E*E \\
&\Rightarrow \mathbf{id}*E \\
&\Rightarrow \mathbf{id}*E+E \\
&\Rightarrow \mathbf{id}*\mathbf{id}+E \\
&\Rightarrow \mathbf{id}*\mathbf{id}+\mathbf{id}
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow E+E \\
&\Rightarrow E*E+E \\
&\Rightarrow \mathbf{id}*E+E \\
&\Rightarrow \mathbf{id}*\mathbf{id}+E \\
&\Rightarrow \mathbf{id}*\mathbf{id}+\mathbf{id}
\end{aligned}
$$

因而有两棵不同的解析树，如图 3.4 所示。

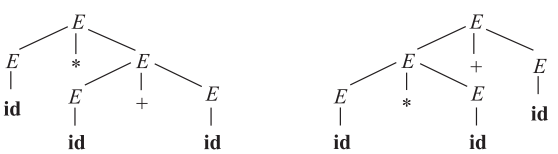


图 3.4 $\mathbf{id}*\mathbf{id}+\mathbf{id}$ 的两棵解析树

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第70页 文件编号: 422624 页面编号: 2704538442 打印人: 8685 2026-03-09 10:22:59

注意，图 3.4 中右边的解析树反映了 + 和 * 通常的运算符优先关系，而左边的解析树则不是。也就是说，习惯上 * 的优先级高于 +，因而表达式 $a*b+c$ 看成 $(a*b)+c$，而不是 $a*(b+c)$。

一个文法，如果存在某个句子有不止一棵解析树与之对应，那么称这个文法是**二义**的。也就是说，二义文法是至少存在一个句子有不止一个最左（最右）推导的文法。有些类型的解析期望处理的文法是无二义的，否则它不能唯一确定对某个句子应选择哪棵解析树。出于某些需要，也可以构造允许二义文法的解析器，不过这样的文法要附带**消除二义性的规则**，以便解析器扔掉不希望的解析树，为每个句子只留一棵解析树。

注意，二义文法并不代表语言一定是二义的。只有当产生一个语言的所有文法都是二义时，这个语言才称为二义的。

3.2 语言和文法

在编程语言和编译器的设计方面，文法都提供了诸多优势：

（1）文法为语言提供了精确且易于理解的语法说明。

（2）对于某些类型的文法，可以为其自动生成高效的解析器。在自动构造解析器过程中，可以揭示文法的二义性和难以解析的语言构造，这些问题很可能在最初设计阶段未被发现。

（3）在合理设计的文法中，非终结符往往对应语言构造（如过程、语句和表达式），这种对应有助于生成目标代码和进行错误诊断。此外，还有工具能将文法翻译描述转换为编译器或翻译器。

（4）语言是逐步完善的，基于文法的语言实现便于添加新语言构造。

需要注意的是，上下文无关文法只能描述编程语言的大部分语法规定，而无法处理像标识符声明先于使用等上下文相关的约束。这类限制需在语法分析后，通过静态语义检查处理。

本节首先介绍文法产生的语言及其验证，然后举例说明如何通过重写文法来消除二义性。本节还介绍文法的种类，并讨论某些不能用上下文无关文法描述的语言构造。

3.2.1 验证文法产生的语言

【例 3.6】 考虑文法

$$S \to \text{'('}\ S\ \text{')'}\ S \mid \varepsilon \tag{3.4}$$

这个简单的文法产生所有配对的括号串，也只产生这样的串。为了明白这一点，首先证明 S 产生的每个句子都是配对的括号串，然后证明任何配对括号串都可以从 S 推导产生。前一个问题可以按推导步数进行归纳。对于归纳基，可以看到，从 S 经一步推导能得到的终结符串只有空串，它是配对的。这里，归纳基是指空串可以从 S 经一步推导得到。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第71页 文件编号: 422624 页面编号: 2714538440 打印人: 8685 2026-03-09 10:22:59

假定所有少于 n 步的推导都能产生配对的括号串，然后考虑 n 步的最左推导。这个推导必定是下面这种形式：

$$S \Rightarrow {}'('S')'S \Rightarrow^* {}'('x')'S \Rightarrow^* {}'('x')'y$$

由于从 S 到 x 和 y 的推导分别都少于 n 步，由归纳假设可知，x 和 y 都是配对括号串，所以串 $(x)y$ 是配对括号串。

下一步证明任何配对括号串都可由 S 产生，按串长进行归纳。

假定长度小于 $2n$ 的配对括号串都可以从 S 推导出来，下面考虑长度 $2n(n \geqslant 1)$ 的配对括号串 w。可以肯定，w 由左括号开始，令 (x) 是 w 的有相同个数的左括号和右括号的最短前缀，那么 w 可以写成 $(x)y$，其中 x 和 y 都是配对括号串，长度都小于 $2n$，由归纳假设，它们都可以从 S 推导出来。这样，w 可以有如下的推导序列：

$$S \Rightarrow {}'('S')'S \Rightarrow {}^*{}'('x')'S \Rightarrow^* {}'('x')'y$$

从而证明了 $w = (x)y$ 可以由 S 推导出来。

读者并不需要掌握这样的证明技术，但是，当为一些例子语言设计文法时，如果能用这样的方式去思考问题，设计出正确文法的可能性就大得多。

3.2.2 无二义性的表达式文法

3.1 节的表达式文法有二义性，一个句子的不同解析树体现不同的运算符优先关系和结合性。下面构造无二义的有 $+$ 和 $*$ 运算的表达式文法，该文法和通常的运算符优先关系和结合性相对应。

设置两个非终结符 expr 和 term（expr 是开始符号），用以表示不同层次的表达式和子表达式，再用非终结符 factor 来产生表达式的基本单位。基本单位有 **id** 和外加括号的表达式，即

$$\text{factor} \rightarrow \mathbf{id} \mid (\text{expr})$$

然后考虑二元运算符 $*$，它既有较高的优先级，又是左结合的运算符，因而产生式如下：

$$\text{term} \rightarrow \text{term} * \text{factor} \mid \text{factor}$$

同样地，expr 产生由加法运算符隔开的、左结合的 term 表，其产生式如下：

$$\text{expr} \rightarrow \text{expr} + \text{term} \mid \text{term}$$

这个表达式文法是无二义的。句子 $\mathbf{id} * \mathbf{id} * \mathbf{id}$ 和 $\mathbf{id} + \mathbf{id} * \mathbf{id}$ 的解析树如图 3.5 所示。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第72页 文件编号: 422624 页面编号: 2724538448 打印人: 8685 2026-03-09 10:22:59

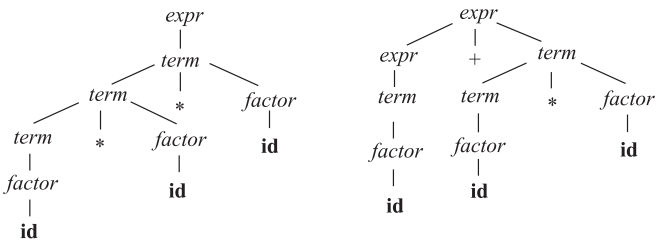


图 3.5 $\mathbf{id} * \mathbf{id} * \mathbf{id}$ 和 $\mathbf{id} + \mathbf{id} * \mathbf{id}$ 的解析树

图 3.5 中两棵解析树展现的运算符优先关系和结合性与通常的规定是一致的。如果语言的语义要求不同的运算符优先关系和结合性，则文法可能需要重新设计；否则，生成的解析树在语义分析和中间代码生成阶段可能不便于使用。例如，如果规定 $*$ 和 $+$ 是右结合的运算，那么文法应调整为

$$\begin{aligned} &\text{expr} \rightarrow \text{term} + \text{expr} \mid \text{term} \\ &\text{term} \rightarrow \text{factor} * \text{term} \mid \text{factor} \\ &\text{factor} \rightarrow \mathbf{id} \mid (\text{expr}) \end{aligned}$$

图 3.6是句子在文法修改后对应的解析树，将其与图 3.5 的解析树比较，应该不难看出区别。

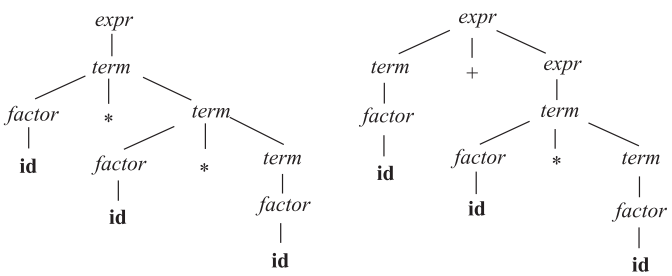


图 3.6 文法修改后的解析树

3.2.3 消除二义性

从上一小节的例子可以看到，有些二义文法可以通过重写文法来消除二义性。再举一个例子，消除下面“悬空 else”文法的二义性：

$$\begin{aligned} \text{stmt} \rightarrow\ & \mathbf{if}\ \text{expr}\ \mathbf{then}\ \text{stmt} \\ & \mid \mathbf{if}\ \text{expr}\ \mathbf{then}\ \text{stmt}\ \mathbf{else}\ \text{stmt} \\ & \mid \mathbf{other} \end{aligned} \tag{3.5}$$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第73页 文件编号: 422624 页面编号: 2734538446 打印人: 8685 2026-03-09 10:22:59

这里的 **other** 代表任何其他语句。按照这个文法，形式为

$$\textbf{if}\ \text{expr}\ \textbf{then}\ \ \textbf{if}\ \text{expr}\ \textbf{then}\ \text{stmt}\ \textbf{else}\ \text{stmt} \tag{3.6}$$

的嵌套条件语句有两个最左推导：

$$\text{stmt} \Rightarrow \textbf{if}\ \text{expr}\ \textbf{then}\ \text{stmt} \Rightarrow \textbf{if}\ \text{expr}\ \textbf{then}\ \textbf{if}\ \text{expr}\ \textbf{then}\ \text{stmt}\ \textbf{else}\ \text{stmt}$$
$$\text{stmt} \Rightarrow \textbf{if}\ \text{expr}\ \textbf{then}\ \text{stmt}\ \textbf{else}\ \text{stmt} \Rightarrow \textbf{if}\ \text{expr}\ \textbf{then}\ \textbf{if}\ \text{expr}\ \textbf{then}\ \text{stmt}\ \textbf{else}\ \text{stmt}$$

因此文法（3.5）是有二义的。

所有含这种条件语句的语言都使用前一种最左推导，因为它和这些语言所采用的规则“每个 **else** 和左边最接近的还没有配对的 **then** 相配对”（简称“**else** 的就近匹配”规则）是一致的。这条规则可以直接体现在文法中，例如，可以把文法（3.5）改写成下面无二义的文法。主要思路是，出现在 **then** 和 **else** 之间的语句必须是“配对”的，配对语句是指那些不是条件语句的语句，还有那些只含配对语句的 if-then-else 语句。

$$\begin{aligned}
\text{stmt} &\rightarrow \text{matched_stmt} \\
&\quad \mid \text{unmatched_stmt} \\
\text{matched_stmt} &\rightarrow \textbf{if}\ \text{expr}\ \textbf{then}\ \text{matched_stmt}\ \textbf{else}\ \text{matched_stmt} \\
&\quad \mid \textbf{other} \\
\text{unmatched_stmt} &\rightarrow \textbf{if}\ \text{expr}\ \textbf{then}\ \text{stmt} \\
&\quad \mid \textbf{if}\ \text{expr}\ \textbf{then}\ \text{matched_stmt}\ \textbf{else}\ \text{unmatched_stmt}
\end{aligned} \tag{3.7}$$

文法 (3.7) 和文法（3.5）产生同样的串集，但是对于句型（3.6）只存在一种最左推导。注意，unmatched_stmt 第二个产生式的右部 **if** expr **then** matched_stmt **else** unmatched_stmt 的 matched_stmt 和 unmatched_stmt 是不能对调的，否则仍然是二义的。

你可能会问，为什么大多数编程语言不使用无二义的文法（3.7），而采用二义文法（3.5）？这是因为无二义的文法（3.7）缺乏简洁性。在定义语言语法时，二义性并非不可解决，只要有相应的规则即可消除。

*3.2.4 文法的种类——形式语言简介

乔姆斯基（Chomsky）把文法分成 4 种类型，即 0 型、1 型、2 型和 3 型，0 型的描述能力强于 1 型，1 型的强于 2 型，2 型的强于 3 型。这四类文法的差别在于对产生式施加不同的限制。

文法 $G = (V_T, V_N, S, P)$ 是 0 型文法，如果它的每个产生式 $\alpha \rightarrow \beta$ 满足约束：$\alpha \in (V_N \cup V_T)^*$，且至少含一个非终结符，而 $\beta \in (V_N \cup V_T)^*$。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第74页 文件编号: 422624 页面编号: 2744538444 打印人: 8685 2026-03-09 10:22:59

0 型文法也称**短语文法**。一个非常重要的理论结果是，0 型文法的能力相当于图灵机。或者说，任何 0 型语言都是递归可枚举的；反之，递归可枚举集也必定是一个 0 型语言。

如果对 0 型文法加上以下第 i 条限制，就可以得到 i 型文法：

（1）G 的任何产生式 $\alpha \to \beta$ 都满足 $|\alpha| \leqslant |\beta|$ （$|x|$ 用来表示 x 中符号的个数）。只有 $S \to \varepsilon$ 例外，但此时 S 不得出现在任何产生式的右部。

（2）G 的任何产生式为 $A \to \beta$ 的形式，$A \in V_N$，$\beta \in (V_N \cup V_T)^*$。

（3）G 的任何产生式为 $A \to aB$ 或 $A \to a$ 或 $A \to \varepsilon$ 的形式，$A, B \in V_N$，$a \in V_T$。

1 型文法也称**上下文有关文法**。这种文法意味着对非终结符的替换需要考虑上下文，并且一般不允许换成 ε 串。例如，若 $\alpha A \beta \to \alpha \gamma \beta$ 是 1 型文法的产生式，且 α 和 β 不都为空，则非终结符 A 只有在 α 和 β 这样的上下文环境下才可以替换成 γ。

2 型文法也就是**上下文无关文法**，非终结符的替换不必考虑上下文。

3 型文法等价于正规式，因而也称**正规文法**。

自乔姆斯基于 1956 年建立形式语言的描述以来，形式语言的理论发展得很快。这种理论对计算机科学有着深刻的影响，特别是对编程语言的设计、编译方法和计算复杂性分析等方面有重大影响。有兴趣的读者可以阅读有关形式语言和自动机理论方面的书籍。

由于本书后面主要涉及上下文无关文法，因而在不引起歧义的前提下把它简称为文法。

*3.2.5 非上下文无关的语言构造

在很多编程语言中，仅用上下文无关文法难以表达其中一些语言构造的规范。本小节将给出一些这样的构造，并用简单的抽象语言来说明其中的难点。

【例 3.7】 考虑抽象语言 $L_1 = \{wcw \mid w\text{属于}(a|b)^*\}$。$L_1$ 句子的特点是，其前后是由 a 和 b 组成的相同的串，中间由 c 把它们隔开，如 $aabcaab$。这个抽象语言是对程序中标识符的声明应先于其引用的抽象，wcw 中的第一个 w 代表标识符 w 的声明，第二个 w 代表它的引用。可以证明该语言不是上下文无关语言，但是这个证明超出了本书的范围。这个例子意味着 C 和 Java 都不是上下文无关语言，因为它们都要求标识符的声明先于引用，并且允许标识符任意长。由于这一点，描述这些语言语法的文法只是用 **id** 这样的记号来代表所有的标识符，而在这些语言的编译器中，由语义分析阶段检查标识符的声明必须先于引用。

【例 3.8】 语言 $L_2 = \{a^n b^m c^n d^m \mid n \geqslant 0, m \geqslant 0\}$ 不是上下文无关语言。L_2 是正规式 $a^* b^* c^* d^*$ 所表示语言的子集，其要求是 a 和 c 的个数相等，b 和 d 的个数相等。它是对过程声明中形参个数和该过程引用的实参个数应该相同这一问题的抽象，a^n 和 b^m 代表两个过程声明的形参表中分别有 n 和 m 个参数，c^n 和 d^m 分别代表这两个过程调用的实参表。

语言中过程声明和引用的语法并不涉及参数的个数。例如 FORTRAN 语言的 call 语句

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第75页 文件编号: 422624 页面编号: 2754538441 打印人: 8685 2026-03-09 10:22:59

可以描述为：

$$\begin{aligned} \text{stmt} &\rightarrow \quad \textbf{call id}\ (\text{expr_list}) \\ \text{expr_list} &\rightarrow \quad \text{expr_list},\ \text{expr} \\ &\quad\ \mid \text{expr} \end{aligned}$$

实参和形参个数的一致性检查也是放在语义分析阶段完成的。

【例 3.9】 语言 $L_3 = \{a^n b^n c^n \mid n \geqslant 0\}$ 也不是上下文无关语言，它是 $L(a^*b^*c^*)$ 中含 a、b 和 c 三个字符个数相等的串。它是对早先排版描述的一个现象的抽象。

L_3 可以用上下文有关文法来定义，其产生式如下：

$$\begin{array}{llllllll} S \rightarrow & aSBC & bB \rightarrow & bb & CB \rightarrow & BC & aB \rightarrow & ab \\ S \rightarrow & aBC & bC \rightarrow & bc & cC \rightarrow & cc & & \end{array}$$

$a^n b^n c^n$ 的推导过程如下：

（1）$S \rightarrow aSBC$ 用 $n-1$ 次得到 $S \Rightarrow^* a^{n-1}S(BC)^{n-1}$；

（2）$S \rightarrow aBC$ 用 1 次得到 $S \Rightarrow^+ a^n(BC)^n$；

（3）$CB \rightarrow BC$ 用 $n(n-1)/2$ 次，交换相邻的 CB，得到 $S \Rightarrow^+ a^n B^n C^n$；

（4）$aB \rightarrow ab$ 用 1 次得到 $S \Rightarrow^+ a^n bB^{n-1}C^n$；

（5）$bB \rightarrow bb$ 用 $n-1$ 次得到 $S \Rightarrow^+ a^n b^n C^n$；

（6）$bC \rightarrow bc$ 用 1 次得到 $S \Rightarrow^+ a^n b^n cC^{n-1}$；

（7）$cC \rightarrow cc$ 用 $n-1$ 次得到 $S \Rightarrow^+ a^n b^n c^n$。

由此可见，上下文有关文法的能力强于上下文无关文法。

有趣的是，有些类似于 L_1、L_2 或 L_3 的语言却是上下文无关的。例如，$L_1' = \{wcw^R | w \in (a|b)^*\}$ 是上下文无关的，其中 w^R 代表串 w 的逆序，它可由下面的文法产生：

$$S \rightarrow aSa \mid bSb \mid c$$

语言 $L_2' = \{a^n b^m c^m d^n \mid n \geqslant 1, m \geqslant 1\}$ 是上下文无关的，它可由下面的文法产生：

$$S \rightarrow aSd \mid aAd$$
$$A \rightarrow bAc \mid bc$$

$L_2'' = \{a^n b^n c^m d^m \mid n \geqslant 1\ m \geqslant 1\}$ 也是上下文无关的，它可由下面的文法产生：

$$S \rightarrow AB$$
$$A \rightarrow aAb \mid ab$$
$$B \rightarrow cBd \mid cd$$

最后，$L_3' = \{a^n b^n \mid n \geqslant 1\}$ 也是上下文无关的，它可由下面的文法产生：

$$S \rightarrow aSb \mid ab$$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第76页 文件编号: 422624 页面编号: 2764538449 打印人: 8685 2026-03-09 10:22:59

3.3 自上而下解析

本节首先介绍自上而下解析的基本概念和一般方法、定义适用于自上而下解析的 LL(1) 文法，然后介绍一些实用的自上而下解析方法及分析表的自动生成，最后讨论自上而下的错误恢复。

3.3.1 自上而下解析的一般方法

自上而下解析的核心是，从文法的开始符号（根结点）出发，尝试通过所有可能方法，自上而下、从左到右为输入串建立解析树，即寻找最左推导。这种分析本质上是一种试探过程，通过反复使用不同的产生式匹配输入串。然而，这种方法遇到失败时会回溯，导致效率低且代价高。

【例 3.10】 若有文法

$$S \to aCb$$
$$C \to cd \mid c$$

为了自上而下地为输入串 $w = acb$ 建立解析树，首先建立只有标记为开始符号 S 的单个结点树，输入指针指向 w 的第一个符号 a。然后用 S 产生式来扩展该树，得到图 3.7(a) 所示的树。

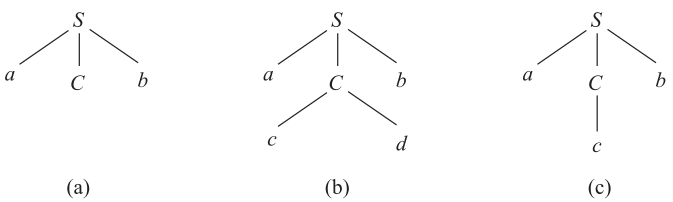


图 3.7 自上而下解析的试探过程

树中最左边的叶子标记为 a，匹配 w 的第一个符号。于是，推进输入指针到 w 的第二个符号 c，并考虑解析树中下一个叶子 C，它是非终结符。用 C 的第一个选择来扩展 C，得到图 3.7(b) 所示的树。现在第二个输入符号 c 能匹配，继续将输入指针向前推进到 b，将其与解析树中下一个叶子 d 比较。因为 b 和 d 不匹配，回溯到 C，看它是否还有别的选择尚未尝试。

在回到 C 时，必须让输入指针重新指向第二个符号，与第一次进入 C 时的位置一致。现在尝试 C 的第二个选择，得到图 3.7(c) 的解析树。叶子 c 匹配 w 的第二个符号，叶子 b 匹配 w 的第三个符号。这就得到 w 的解析树，表明解析完全成功。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第77页 文件编号: 422624 页面编号: 2774538447 打印人: 8685 2026-03-09 10:22:59

这种自上而下解析法存在着困难和缺点。首先，如果存在非终结符 A，并且有

$$A \Rightarrow^{+} Aa$$

这样的**左递归**（非终结符 A 经过至少一步推导能产生以自身开头的符号串），那么在使用 A 去匹配输入串时，分析过程可能会陷入无限循环 (若输入串不是该文法的句子，则一定陷入无限循环)。这是因为左递归会导致解析器在没有消耗任何输入符号的情况下，不断重复对 A 进行匹配，从而无法推进分析过程。因此，在使用自上而下解析法时，文法必须没有左递归。

当非终结符用某个选择匹配成功时，这种成功可能是暂时的。后续分析失败时需要使用复杂的回溯技术。试探与回溯是一种穷尽一切可能的办法，效率低且代价高，实际应用中价值有限。

3.3.2 消除左递归

一个文法是**左递归的**，如果它存在某个非终结符 A 和某个文法符号串 α，有推导 $A \Rightarrow^{+} A\alpha$。自上而下的分析方法不能用于左递归文法，因此需要消除左递归。由形式为 $A \to A\alpha$ 的产生式引起的左递归称为**直接左递归**。

直接左递归产生式 $A \to A\alpha \mid \beta$ 可以用非左递归

$$\begin{aligned} A &\to \beta A' \\ A' &\to \alpha A' \mid \varepsilon \end{aligned}$$

来代替，它们没有改变从 A 推导出的串集。

【例 3.11】 考虑下面的算术表达式文法：

$$\begin{aligned} E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid \mathbf{id} \end{aligned}$$

消除 E 和 T 的直接左递归，可以得到

$$\begin{array}{lll} E \to TE' & T \to FT' & F \to (E) \mid \mathbf{id} \\ E' \to +TE' \mid \varepsilon & T' \to *FT' \mid \varepsilon & \end{array} \tag{3.8}$$

不管有多少 A 产生式，都可以用下面的技术消除直接左递归。首先把 A 产生式组合在一起：

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第78页 文件编号: 422624 页面编号: 2784538445 打印人: 8685 2026-03-09 10:22:59

其中，β_i 都不以 A 开始，α_i 都非空，然后用

$$A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \varepsilon$$

代替前面的 A 产生式。这些产生式和前面的产生式产生一样的串集，但是不再有左递归。这个过程可消除直接左递归，但不能消除两步或多步推导形成的间接左递归。例如，考虑文法：

$$S \to Aa \mid b$$
$$A \to Sd \mid \varepsilon$$

其中，非终结符 S 是左递归的，因为 $S \Rightarrow Aa \Rightarrow Sda$，但它不是直接左递归的。用 S 产生式代换 $A \to Sd$ 中的 S，可以得到下面的文法：

$$S \to Aa \mid b$$
$$A \to Aad \mid bd \mid \varepsilon$$

消除其中的直接左递归，得到如下的文法：

$$S \to Aa \mid b$$
$$A \to bdA' \mid A'$$
$$A' \to adA' \mid \varepsilon$$

由此可见，编写一个消除文法左递归的算法并不困难。

如果对某非终结符推导期间，会在调用未消耗任何输入 (即 ε) 的其他产生式规则后再次调用自身尝试匹配，就称发生了**隐藏左递归**。例如文法

$$S \to ABc$$
$$B \to Cd \mid ABe$$
$$C \to Sf$$
$$A \to \varepsilon$$

中，$B \Rightarrow ABe \Rightarrow Be$ 是隐藏直接左递归，因为对 B 尝试按 ABe 匹配期间，A 未消耗任何输入，就又再次调用对 B 的匹配。$S \Rightarrow ABc \Rightarrow Bc \Rightarrow Cdc \Rightarrow Sfdc$ 则是隐藏间接左递归。隐藏左递归相对难检测，处理起来也更棘手。

3.3.3 提左因子

提左因子也是一种文法变换，它用于产生适合于自上而下解析的文法。在自上而下解析中，当不清楚应该用非终结符 A 的哪个选择来替换它时，可以通过重写 A 产生式来推迟这种决定，直到看到足够多的输入信息，以便做出正确的选择。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第79页 文件编号: 422624 页面编号: 2794538443 打印人: 8685 2026-03-09 10:22:59

例如，条件语句有两个产生式：

$$\begin{aligned}\text{stmt} \rightarrow\ & \textbf{if}\ \text{expr}\ \textbf{then}\ \text{stmt}\ \textbf{else}\ \text{stmt}\\ \mid\ & \textbf{if}\ \text{expr}\ \textbf{then}\ \text{stmt}\end{aligned}$$

当看见输入记号 **if** 时，不能马上确定用哪个产生式来扩展 stmt。

一般来说，如果 $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ 是 A 的两个产生式，当输入串的前缀是从 α 推导出的非空串时，解析器无法确定是用 $\alpha\beta_1$ 还是 $\alpha\beta_2$ 来扩展 A。此时，可以通过先将 A 扩展为 $\alpha A'$ 来推迟这个决定。随后，在看完了从 α 推出的输入后，再扩展 A' 为 β_1 或 β_2。这种方法称为**提左因子**，原来的产生式变为：

$$\begin{aligned}A &\rightarrow \alpha A'\\ A' &\rightarrow \beta_1 \mid \beta_2\end{aligned}$$

【例 3.12】 对于悬空 **else** 的文法

$$\begin{aligned}\text{stmt} \rightarrow\ & \textbf{if}\ \text{expr}\ \textbf{then}\ \text{stmt}\ \textbf{else}\ \text{stmt}\\ \mid\ & \textbf{if}\ \text{expr}\ \textbf{then}\ \text{stmt}\\ \mid\ & \textbf{other}\end{aligned}$$

提左因子后的文法成为

$$\begin{aligned}\text{stmt} \rightarrow\ & \textbf{if}\ \text{expr}\ \textbf{then}\ \text{stmt}\ \text{optional_else_part}\\ \mid\ & \textbf{other}\\ \text{optional_else_part} \rightarrow\ & \textbf{else}\ \text{stmt}\\ \mid\ & \varepsilon\end{aligned}$$

这样，如果输入的第一个记号是 **if**，那么扩展 stmt 到 **if** expr **then** stmt optional_else_part，等到 **if** expr **then** stmt 都看见后，再决定扩展 optional_else_part 到 **else** stmt 还是到 ε。

3.3.4 LL(1) 文法

为构造不带回溯的自上而下解析算法，首先要消除文法的左递归，然后要找出避免回溯的充分必要条件。下面讨论如何避免回溯。

对文法的任何非终结符，当要用它去匹配输入串时，如果能够根据所面临的输入符号准确指派它的一个选择去执行任务，那么就肯定能消除回溯。这个“准确”是指：若此选择匹配成功，则这种匹配确实真实有效；若此选择无法完成匹配任务，则任何其他的选择也肯定无法完成。

在讨论不得回溯的前提对文法有哪些限制之前，先定义两个与文法有关的函数。一个文法的符号串 α 的**开始符号集合** $\text{FIRST}(\alpha)$ 是由 α 推出的非空串中首终结符组成的集合；特别地，在 α 推出空串时，规定 $\varepsilon \in \text{FIRST}(\alpha)$。也就是

$$\text{FIRST}(\alpha) = \{a \mid \alpha \Rightarrow^* a\cdots, a \in V_T\} \cup \{\varepsilon \mid \alpha \Rightarrow^* \varepsilon\}$$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第80页 文件编号: 422624 页面编号: 2804538441 打印人: 8685 2026-03-09 10:22:59

如果对 A 的任何两个不同的选择 α_i 和 α_j，有

$$\mathrm{FIRST}(\alpha_i) \cap \mathrm{FIRST}(\alpha_j) = \varnothing$$

那么，当要求 A 匹配输入串时，A 就能根据解析器当前面临的下一个输入符号 a，准确地指派某一个选择前去执行任务。这个选择就是满足 $a \in \mathrm{FIRST}(\alpha)$ 的那个 α。

利用提左因子的方法，可以把文法改造成对任何非终结符 A，A 的所有选择的开始符号集合两两不相交。

如果 ε 属于 A 的某个选择的开始符号集合，那么问题就比较复杂，需要定义文法非终结符的后继符号集合后才能解释。非终结符 A 的**后继符号集合** $\mathrm{FOLLOW}(A)$ 是所有在句型中可以直接出现在 A 后面的终结符的集合；如果 A 是某个句型的最右符号，那么 $\$ \in \mathrm{FOLLOW}(A)$，即

$$\mathrm{FOLLOW}(A) = \{a \mid S \Rightarrow^* \cdots Aa \cdots, a \in V_T\} \cup \{\$ \mid S \Rightarrow^* \cdots A\}$$

【例 3.13】 考虑文法（3.8），重复如下：

$$\begin{array}{lll} E \to TE' & T \to FT' & F \to (E) \mid \mathbf{id} \\ E' \to +TE' \mid \varepsilon & T' \to *FT' \mid \varepsilon & \end{array}$$

那么

$$\begin{aligned} &\mathrm{FIRST}(E) = \mathrm{FIRST}(T) = \mathrm{FIRST}(F) = \{(, \mathbf{id}\} \\ &\mathrm{FIRST}(E') = \{+, \varepsilon\} \\ &\mathrm{FIRST}(T') = \{*, \varepsilon\} \end{aligned}$$

有了上面这些 FIRST 集合后，就不难计算各产生式右部的 FIRST 集合了。例如，对于 $T \to FT'$，$\mathrm{FIRST}(FT') = \mathrm{FIRST}(F) = \{(, \mathbf{id}\}$。

再看 FOLLOW 集合。这里要注意的是，如果有产生式 $A \to \alpha B$ 或 $A \to \alpha B\beta$ 且 $\beta \Rightarrow^* \varepsilon$，那么 $\mathrm{FOLLOW}(A)$ 的一切元素都要加入 $\mathrm{FOLLOW}(B)$ 中：

$$\begin{aligned} &\mathrm{FOLLOW}(E) = \mathrm{FOLLOW}(E') = \{), \$\} \\ &\mathrm{FOLLOW}(T) = \mathrm{FOLLOW}(T') = \{+,), \$\} \\ &\mathrm{FOLLOW}(F) = \{+, *,), \$\} \end{aligned}$$

设计一个算法来计算开始符号集合和后继符号集合是件简单的事情。

下面回到 ε 属于 A 的某个选择 β 的开始符号集合这个问题。如果 a 属于 A 的另一个选择 α 的开始符号集合，并且 a 属于 $\mathrm{FOLLOW}(A)$，那么当面临输入符号 a 时，为 a 选择 α 或 β 都是有理由的，其中选择后者的理由是让 β 推出空串，把这个 a 看成是 A 的后继符号。

这样，要想不出现回溯，需要文法的任何含两个选择的产生式 $A \to \alpha \mid \beta$ 都满足：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第81页 文件编号: 422624 页面编号: 2814538449 打印人: 8685 2026-03-09 10:22:59

（1）$\mathrm{FIRST}(\alpha) \cap \mathrm{FIRST}(\beta) = \varnothing$；

（2）若 $\beta \to^* \varepsilon$，那么 $\mathrm{FIRST}(\alpha) \cap \mathrm{FOLLOW}(A) = \varnothing$。

把满足这两个条件的文法称为 **LL(1) 文法**，其中的第一个 “L” 代表从左向右扫描输入，第二个 “L” 表示产生最左推导，“1” 代表在决定解析器的每步动作时需要向前查看下一个输入符号（输入指针所指向的符号）。除了没有公共左因子外，LL(1) 文法还有一些明显的性质：它不是二义的，也不含左递归。还有一些性质将在 3.3.7 小节构造预测分析表时再介绍。很明显，表达式文法（3.8）是 LL(1) 的。

如果将 FIRST 函数改造成计算给定文法符号串 α 的前 k 个终结符串形成的集合，即

$$\mathrm{FIRST}(k, \alpha)$$

再将两个限制条件改为

（1）$\mathrm{FIRST}(k, \alpha) \cap \mathrm{FIRST}(k, \beta) = \varnothing$；

（2）若 $\beta \Rightarrow^* \varepsilon$，那么 $\mathrm{FIRST}(k, \alpha) \cap \mathrm{FOLLOW}(A) = \varnothing$。

那么满足这两个条件的文法就称为 LL(k) 文法，其中 k 表示需要向前查看 k 个输入符号来决定解析器的每步动作。

3.3.5 递归下降的预测解析

预测解析是指能根据当前面临的输入符号，为非终结符预测采用哪一个选择来展开而不引起自上而下解析回溯。LL(1) 文法是满足这个要求的。**递归下降**的预测解析是指为每一个非终结符编写一个分析函数，因为文法的定义可能是递归的，所以这些函数也可能是递归的。在处理输入串时，首先执行的是开始符号所对应的分析函数，然后根据产生式右部出现的非终结符，依次调用相应的分析函数，这种逐步下降的函数调用序列隐含地为输入串建立了对应的解析树。

下面通过一个例子来说明如何构造递归下降的预测解析程序。如下的文法产生 Pascal 语言的类型子集，用记号 **dotdot** 表示 “..” 以强调这个字符序列是一个词法单元：

$$\begin{aligned}
\mathrm{type} \to\ & \mathrm{simple} \\
& \mid \uparrow \mathbf{id} \\
& \mid \mathbf{array}[\mathrm{simple}]\ \mathbf{of}\ \mathrm{type} \\
\mathrm{simple} \to\ & \mathbf{integer} \\
& \mid \mathbf{char} \\
& \mid \mathbf{num}\ \mathbf{dotdot}\ \mathbf{num}
\end{aligned}$$

显然，该文法是 LL(1) 的。

图 3.8 所示为上述文法的递归下降预测解析器代码。这个解析器包括非终结符 type 和

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第82页 文件编号: 422624 页面编号: 2824538447 打印人: 8685 2026-03-09 10:22:59

simple 的分析函数以及辅助函数 match()。引入 match() 函数是为了简化 type() 函数和 simple() 函数的代码。match() 试图将传入的参数 t 与面临的输入符号进行匹配，若匹配则调用函数 nextToken() 获取下一个记号，并更新下一输入符号到变量 lookahead，否则就调用 error() 进行错误处理。对于 type 产生式，会引入相应的 type 分析函数，其函数体依次处理产生式的 3 个选择，它根据每个选择的 FIRST 集来形成分支条件。以第 1 个选择 simple 为例，由于 FIRST(simple) = {**integer, char, num**}，故当 lookahead 为 **integer** 或 **char** 或 **num** 时，调用 simple(); 而对于第 2 个选择，当 lookahead 为 ↑ 时，依次调用 match(↑) 和 match(**id**) 来匹配该选择的终结符串。

```
void match (terminal t) {
    if (lookahead == t) lookahead = nextToken();
    else error();
}

void type() {
    if ((lookahead == integer) || (lookahead == char) || (lookahead == num)) simple();
    else if(lookahead == '↑') {match ('↑'); match (id);}
    else if (lookahead == array) {
        match(array); match ('['); simple(); match (']'); match (of); type();
    }
    else error();
}

void simple() {
    if(lookahead == integer) match (integer);
    else if (lookahead == char) match (char);
    else if (lookahead == num) {match(num); match(dotdot); match(num); }
    else error();
}
```

图 3.8 递归下降预测解析器的代码

从这个例子可以看到，递归下降预测解析程序的构造与文法结构直接对应，这种分析程序的编写是简单的。

3.3.6 非递归的预测解析

如果显式地维护一个栈，而不是隐式地通过递归调用，那么可以构造非递归的预测解析器。预测解析的关键问题是，在扩展一个非终结符时怎样为它选择合适的产生式。图 3.9 中

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第83页 文件编号: 422624 页面编号: 2834538445 打印人: 8685 2026-03-09 10:22:59

的非递归预测解析器是通过查询分析表来决定产生式的。

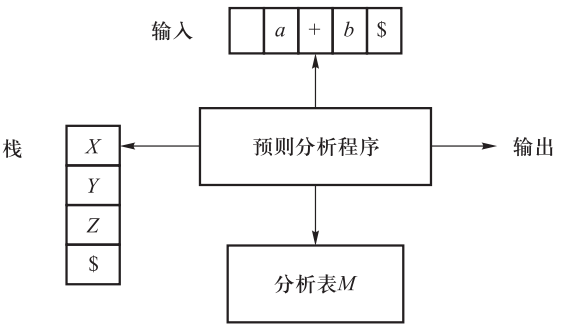


图 3.9 非递归预测解析器的模型

表驱动的预测解析器有一个输入缓冲区、一个栈、一张分析表和一个输出流。输入缓冲区包含待分析的串，后面跟一个表示输入串结束标记的符号 $。栈中存放文法的符号串，栈底符号是 $。初始时，栈中含文法的开始符，它在 $ 的上面。分析表是一个二维数组 M，它包含 $|V_N|$ 行、$|V_T|+1$ 列，$|V_N|$ 和 $|V_T|$ 分别代表非终结符和终结符，单元格 $M[A,a]$ 指示在面临输入符号 a（为终结符或 $）时对非终结符 A 可用哪个产生式展开。

现在说明这个解析器的工作过程。预测解析程序根据当前的栈顶符号 X 和面临的输入符号 a 决定解析器的动作，它有 4 种可能：

（1）如果 $X=a=\$$，解析器宣布分析完全成功而停机。

（2）如果 $X=a\neq\$$，解析器弹出栈顶符号 X，并使输入指针指向下一个符号。

（3）如果 X 是终结符但不是 a，则解析器报告发现语法错误（出错），调用错误恢复例程。

（4）如果 X 是非终结符，程序会访问分析表 M；若 $M[X,a]$ 是 X 的产生式，如 $M[X,a]=\{X\to UVW\}$，那么解析器用 WVU 代替栈顶的 X，并让 U 在栈顶。作为输出，在此假定解析器打印出所用的产生式，当然也可以执行其他代码。如果 $M[X,A]$ 指示出错，则解析器调用错误恢复例程。

算法 3.1 非递归的预测解析。

输入：串 w 和文法 G 的分析表 M。

输出：如果 w 属于 $L(G)$，则输出 w 的最左推导，否则报告错误。

方法：初始时解析器的**格局** (configuration) 是：S 在栈里，其中 S 是开始符号并且在栈顶；w $ 在输入缓冲区中。图 3.10是用预测分析表 M 对输入串进行分析的程序。不论分析处在什么时候，若 w 是句子，栈和输入缓冲区始终满足一种不变式，即栈中从栈顶到栈底形成的文法符号串经过若干步可推导出输入缓冲区从左到右的剩余输入记号流。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第84页 文件编号: 422624 页面编号: 2844538443 打印人: 8685 2026-03-09 10:22:59


```
让输入指针ip指向剩余输入记号流ω$的第一个符号，记为a；
令 X 等于栈顶符号；
while（X != $）{ /* 栈非空 */
    if（X 是 a）把 X 从栈顶弹出并把 ip 指向下一个符号；
    else if（X 是终结符）error（）；
    else if（M[X, a] 是出错入口）error（）；
    else if（M[X, a] = X → Y1Y2 ⋯ Yk）{
        输出产生式 X → Y1Y2 ⋯ Yk；
        从栈中弹出 X；
        把 Yk, Yk−1, ⋯, Y1依次压入栈，Y1在栈顶；
    }
    令 X 等于栈顶符号；
}
```

图 3.10 预测解析程序

【例 3.14】 考虑文法（3.8），该文法的预测分析表见表 3.1，表中单元格为空白表示出错，非空白指示一个用来展开栈顶非终结符的产生式，即用其右部替换出现在栈顶的左部非终结符。

表 3.1 文法（3.8）的分析表

非终结符	输入符号					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to$ **id**			$F \to (E)$		

如果输入是 **id** $*$ **id** $+$ **id**，分析过程中各部分的变化见表 3.2。输入指针指向剩余输入中最左边的符号。解析器的输出动作反映了最左推导的过程，也就是输出最左推导所用的那些产生式。已匹配的输入符号与栈中的文法符号（从栈顶到栈底）一起，构成输入串的最左推导句型。

3.3.7 构造预测分析表

在非递归的预测解析中，构造预测分析表是关键步骤。下面的算法为文法 G 构造预测分析表 M，并填写单元格 $M[A, a]$，其中 A 是非终结符，a 是终结符或 $。

算法思想：对于每个产生式 $A \to \alpha$，若 a 在 FIRST(α) 中，则将产生式 $A \to \alpha$ 填入

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第85页 文件编号: 422624 页面编号: 2854538440 打印人: 8685 2026-03-09 10:22:59

$M[A,a]$；若 $\alpha \Rightarrow^* \varepsilon$，则还需检查当前输入符号（包括 \$ ）是否在 FOLLOW(A) 中，若在，则仍应用 α 展开 A。

表 3.2 预测解析器接受输入 id * id + id 的动作

栈	输入	动作
$\$E$	**id** * **id** + **id** \$	
$\$E'T$	**id** * **id** + **id** \$	输出 $E \to TE'$
$\$E'T'F$	**id** * **id** + **id** \$	输出 $T \to FT'$
$\$E'T'$ **id**	**id** * **id** + **id** \$	输出 $F \to$ **id**
$\$E'T'$	* **id** + **id** \$	匹配 **id**
$\$E'T'F*$	* **id** + **id** \$	输出 $T' \to *FT'$
$\$E'T'F$	**id** + **id** \$	匹配 *
$\$E'T'$ **id**	**id** + **id** \$	输出 $F \to$ **id**
$\$E'T'$	+ **id** \$	匹配 **id**
$\$E'$	+ **id** \$	输出 $T' \to \varepsilon$
$\$E'T+$	+ **id** \$	输出 $E' \to +TE'$
$\$E'T$	**id** \$	匹配+
$\$E'TF'$	**id** \$	输出 $T \to FT'$
$\$E'T'$ **id**	**id** \$	输出 $F \to$ **id**
$\$E'T'$	\$	匹配 **id**
$\$E'$	\$	输出 $T' \to \varepsilon$
\$	\$	输出 $E' \to \varepsilon$

算法 3.2 构造预测分析表。

输入：文法 G。

输出：分析表 M。

方法：对文法的每个产生式 $A \to \alpha$，执行步骤（1）和步骤（2）。

（1）对 FIRST(α) 中的每个终结符 a，把 $A \to \alpha$ 加入 $M[A,a]$。

（2）如果 ε 在 FIRST(α) 中，对 FOLLOW(A) 中的每个符号 b（包括 \$ ），把 $A \to \alpha$ 加入 $M[A,b]$（包括 $M[A,\$]$）。

M 中剩余未定义的条目都是出错条目，通常用空白表示。

【例 3.15】 把算法3.2 用于文法（3.8），产生的分析表见表 3.1。因为 $\text{FIRST}(TE') = \text{FIRST}(T) = \{(, \textbf{id}\}$，因此将产生式 $E \to TE'$ 加入 $M[E,(]$ 和 $M[E,\textbf{id}]$。

产生式 $E' \to +TE'$ 加入 $M[E',+]$ 是显然的。因为 $\varepsilon \in \text{FIRST}(E')$，并且 $\text{FOLLOW}(E') =$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第86页 文件编号: 422624 页面编号: 2864538448 打印人: 8685 2026-03-09 10:22:59

{),$}，所以将产生式 $E' \to \varepsilon$ 加入 $M[E',)]$ 和 $M[E', \$]$。

算法3.2 可用于为任何文法 G 产生分析表 M。然而对某些文法，M 可能含有一些多重定义的条目。例如，G 是左递归或二义的，则 M 至少含一个单元格，其中有多个产生式规则。

【例 3.16】 以例 3.12 的条件语句为例，为方便起见，将文法重写如下：

$$\begin{aligned} &\text{stmt} \to \textbf{if}\ \text{expr}\ \textbf{then}\ \text{stmt}\ \text{e_part} \mid \textbf{other} \quad /*\ \text{e_part指optional_else_part}\ */ \\ &\text{e_part} \to \textbf{else}\ \text{stmt} \mid \varepsilon \\ &\text{expr} \to \text{b} \end{aligned} \tag{3.9}$$

文法 (3.9) 的预测分析表见表 3.3。

表 3.3 文法（3.9）的预测分析表

非终结符	输入符号					
	other	b	**else**	**if**	**then**	$
stmt	stmt → **other**			stmt → **if** ···		
e_part			e_part → **else** stmt e_part → ε			e_part → ε
expr		expr → b				

M[e_part, **else**] 条目包括 e_part→ **else** stmt 和 e_part → ε，后者是因为 $\varepsilon \in$ FIRST (e_part) 并且 FOLLOW(e_part) = {**else**, $}。可以通过删去该条目中的 e_part → ε 来解决这种二义性，这个选择刚好满足 “**else** 和最接近的 **then** 配对” 约定，即 **else 的就近匹配**原则。

可以证明，一个文法的预测分析表没有多重定义的条目，当且仅当该文法是 LL(1) 的。还可以证明，算法3.2 为 LL(1) 文法 G 产生的分析表能且仅能分析 $L(G)$ 的所有句子。

剩下的问题是，当分析表有多重定义的条目时应该怎么解决？一种办法是通过文法变换来解决，如消除左递归和提取左因子，使得变换后的新文法对应的分析表中不再有多重定义的条目。然而，有些文法不论怎么变换也无法得到 LL(1) 文法。例如，文法（3.9）代表的语言无法用 LL(1) 文法表示。但是，如果让 M[e_part, **else**] = {e_part → **else** stmt}, 基于修改后的分析表就可以用预测解析器对文法（3.9）进行分析。不过一般来说，没有一个通用的规则能够指导如何删除多重定义的条目，使其成为单一值而不影响解析器识别的语言。

使用预测解析的主要困难在于为源语言编写适合构造预测解析器的文法。虽然消除左递归和提左因子相对简单，但是它们会导致文法难以理解和翻译。3.6 节的 LR 解析器可以克服这些问题。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第87页 文件编号: 422624 页面编号: 2874538446 打印人: 8685 2026-03-09 10:23:00

3.3.8 预测解析的错误恢复

1. 编译器的错误处理概述

如果编译器只处理正确的程序，它的设计和实现会大大简化，但是程序员通常需要多次修改程序才能使其正确。好的编译器应能帮助程序员识别和定位错误。错误比较容易发生，但是几乎所有编程语言的规范都没有描述编译器应该怎样处理错误。

在设计编译器时，如果从一开始就规划错误处理，就可以简化编译器的结构，并且改进它对错误的响应。普通的编程错误会出现在不同的层次上：

（1）词法错误，如标识符、关键字或运算符的拼写错误，遗漏字符串两端的引号等。在第 2 章已经提到，这些基于程序员观点的词法错误并非一定能在词法分析阶段发现。

（2）语法错误，如算术表达式的括号不配对等。这些错误在语法分析阶段可以被发现。另外一些情况，如在 C 和 Java 中，case 语句如果没有外围的 switch 语句，则是一个语法错误。由于这样的约束没有体现在语言的文法中，因此解析器通常发现不了这种错误，但是它会在编译器试图产生代码时或在其他场合被捕获。

（3）语义错误，如运算符和运算对象之间类型不匹配。具体例子有，在 Java 方法中，不带表达式的 return 语句要求相应方法的返回类型是 void，否则就是错误。

（4）逻辑错误，如在 C 程序中用赋值号 “=” 代替比较运算符 “==”。这时，从编译器看，整个程序可能仍然合法，但是它未能正确反映程序员的意图。普通的编译器难以发现这样的错误。

之所以强调分析期间的错误恢复，是因为大多数错误都是语法错误。虽然类型不匹配等少数语义错误也能被有效诊断出来，但是在编译时精确诊断语义错误和逻辑错误通常是非常困难的。

解析器对错误处理的基本目标是：

（1）清楚而准确地报告所出现的错误；

（2）迅速地从每个错误中恢复过来，以便诊断剩余程序的错误；

（3）它不应该使处理正确程序的速度降低太多。

有效实现这些目标是非常困难的。幸好，常见的错误往往比较简单，一般可以通过直接的错误处理机制解决。但是，有时错误的实际位置可能早于被发现的位置，并且具体性质难以推断。还有一些情况，错误处理程序甚至需要推测程序员的意图。

错误处理程序应该怎样报告错误？它应该至少报告源程序中检测到错误的位置，尽管这可能与实际错误的位置有所偏差。编译器通常通过打印出错的程序行，指出检测到错误的位置。如果编译器能够推测出具体的错误类型，它还会提供诊断信息，如 “此处漏了分号”。

2. 语法分析阶段的错误恢复

不同的语法分析方法对错误诊断的有效性影响不同。最简单的错误处理方法是，解析器在发现错误时暂停分析并提供相关信息。如果解析器能够进入一个可以继续处理剩余输入的

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第88页 文件编号: 422624 页面编号: 2884538444 打印人: 8685 2026-03-09 10:23:00

状态，它可以从该状态恢复，以继续揭示后续程序中的错误。然而，如果错误信息太多，那么编译器可能需要限制错误信息的输出，以免让程序员被大量无关错误信息所困扰。

在预测解析中，非递归预测解析器利用栈来清晰识别希望与剩余输入匹配的符号。当栈顶的终结符和下一个输入符号不匹配，或者栈顶是非终结符 A 而输入符号是 a 且 $M[A,a]$ 指示出错时，预测解析器会发现错误。这种技术同样适用于递归下降分析，帮助实现更有效的错误恢复。

对于非递归的预测解析，**紧急错误恢复**是一种简单的策略。当发现错误时，解析器会逐一丢弃输入记号，直到遇到指定的同步记号集合中的符号为止。这种方法适用于大多数分析方法，其效果依赖于同步记号集合的选择。编译器设计者需要选择适当的同步记号，以便解析器能迅速从错误中恢复。然而，这种方法的缺点是，解析器可能会跳过一段输入记号，不检查其中是否有其他错误。有关同步记号选择的一些提示如下：

（1）可以把 FOLLOW(A) 集合中的所有终结符放入非终结符 A 的同步记号集合。出错时，若栈顶是 A，则解析器会丢弃一些记号，直到遇到 FOLLOW(A) 中的元素为止，然后把 A 弹出栈，恢复分析。这样做的本质是放弃对 A 推出部分的分析，在其后恢复分析。

（2）仅使用 FOLLOW(A) 作为 A 的同步记号集合可能不足以有效恢复错误。例如，在 C 语言中，分号作为语句的结束符，但语句开始的关键字通常不会出现在表达式非终结符的 FOLLOW 集合中。因此，如果仅按上面的（1）来设定同步记号集合，当赋值语句中的分号被遗漏时，下一语句的开始关键字可能会被错误地丢弃，甚至导致整个语句被忽略。

为此，可以利用语言结构的层次性，将高层构造的开始符号加到低层构造的同步记号集合中。例如，将语句开始的关键字加入到表达式非终结符的同步记号集合中。

（3）将 FIRST(A) 中的终结符加入 A 的同步记号集合中，可以在输入中出现 FIRST(A) 中的终结符时恢复对 A 的分析。

（4）如果一个非终结符可以推导出空串，就把能推导出空串的产生式作为默认产生式，在出错时若栈顶是这样的非终结符，就使用该默认产生式。虽然这样做可能会延迟错误的发现，但不会遗漏错误，而且还可以减少错误恢复要考虑的非终结符数。

（5）如果终结符在栈顶而不能匹配，简单的办法是，除了报告错误外，弹出此终结符，继续分析。在效果上，这种方式等于把所有其他的记号作为该终结符的同步记号集合。

【例 3.17】 按照文法（3.8）分析表达式时，使用 FOLLOW 符号和 FIRST 符号作为同步记号是合情合理的。在文法（3.8）的分析表表 3.1 中加入同步记号 synch，如表 3.4 所示，可用 synch 来指示从非终结符的 FOLLOW 集合中得到的同步记号。非终结符的 FOLLOW 集合从例 3.13可得。

表 3.4 的使用如下：如果解析器查找条目 $M[A,a]$，发现它是空的，则跳过输入符号 a；如果条目是 synch，则调用同步过程并把栈顶的非终结符弹出，恢复分析；如果栈顶的记号与输入符号不匹配，则从栈顶弹出该记号。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第89页 文件编号: 422624 页面编号: 2894538442 打印人: 8685 2026-03-09 10:23:00

表 3.4 同步记号加到表 3.1 的分析表中

非终结符	输入符号					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to$ **id**	synch	synch	$F \to (E)$	synch	synch

表 3.4 的解析器和错误恢复机制在面临有语法错误的输入 + **id** * + **id** 时，其行为见表 3.5。

表 3.5 由预测解析器产生的分析和错误恢复动作

栈	输入	输出
$E	+ **id** * + **id** $	出错，跳过“+”
$E	**id** * + **id** $	**id** 属于 FIRST(E)
$E'T	**id** * + **id** $	
$E'T'F	**id** * + **id** $	
$E'T' **id**	**id** * + **id** $	
$E'T'	* + **id** $	
$E'T'F*	* + **id** $	
$E'T'F	+ **id** $	出错；“+” 正好在 F 的同步记号集合中，无须跳过任何记号；F 被弹出
$E'T'	+ **id** $	
$E'	+ **id** $	
$E'T+	+ **id** $	
$E'T	**id** $	
$E'T'F	**id** $	
$E'T' **id**	**id** $	
$E'T'	$	
$E'	$	
$	$	

上述紧急恢复方法没有涉及错误信息的问题。通常，错误信息必须由编译器设计者提供。

3.4 LL 解析器的生成器：ANTLR

ANTLR（Another Tool for Language Recognition）是一种广泛使用的解析器生成工具，它提供灵活高效的自上而下解析技术。2010 年发布的 ANTLR 3.3 引入 LL(*) 解析方法[8]，允许动态确定向前看符号的数量（LL(k) 中 k 不再固定），能够处理更复杂的文法。同时，

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第90页 文件编号: 422624 页面编号: 2904538440 打印人: 8685 2026-03-09 10:23:00

ANTLR 3 还引入了语法树（abstract syntax tree，AST）和树遍历器，支持后续的语义分析和代码生成。

2014 年发布的 ANTLR 4 进一步简化了语法定义，并引入自适应 LL(*)（adaptive LL(*)）解析技术[9]，简称 ALL(*)。ALL(*) 通过自适应向前看和动态状态机构建，提高了解析效率。ANTLR 4 还支持直接生成解析树，方便开发者进行语义分析和其他编译步骤。

本节首先介绍 ANTLR 基于的 LL(*) 解析技术，然后介绍用 ANTLR 4 构造解析器的要点。

3.4.1 LL(*) 解析技术概述

LL(*) 解析器的核心思想是通过使用正规式进行向前看（lookahead），而不是依靠固定的向前看深度或完全依靠解析器的回溯。其分析目标是为每个非终结符构建一个向前看的确定性有限自动机 (DFA)，以区分产生式选择，从而做出预测决策。

使用 DFA 进行向前看时，会将 LL(*) 的解析决策限制为：向前看语言（剩余输入序列的集合）必须是正规的，但这些语言通常是上下文无关的。

ANTLR 3 采用静态文法分析算法从 ANTLR 文法构建 LL(*) 解析决策。当 LL(*) 文法的决策条件静态不可判定时，即静态文法分析无法找到区分不同产生式选择的正规式时，ANTLR 3 的静态分析能够检测这种不可判定的情况，失败时转而采用回溯解析决策。回溯解析决策会选择第一个匹配的产生式继续解析，这种决策方法无法检测到像 $A \to \alpha \mid \alpha$ 这种显而易见的冲突，其中 α 是任意使 $\alpha \mid \alpha$ 不是 LL(*) 的文法符号序列。

ANTLR 4 为解决 LL(*) 文法条件静态不可判定的问题，提出了自适应的 LL(*) 解析技术，通过将文法分析从编译时移到解析时进行，利用记忆化分析结果和**图结构栈**（graph-structured stack，GSS）等手段避免重复计算，支持对任意 LL(*) 文法的增量式动态解析决策。ALL(*) 预测机制的核心思想是在每个非终结符的决策点启动多个子解析器，每个子解析器对应一个产生式选择。这些子解析器以伪并行方式运行，探索所有可能的解析路径。当某条路径无法匹配剩余输入时，相应的子解析器会终止。子解析器同步推进输入，解析器能够基于最小向前看深度，唯一预测出产生式选择。如果多个子解析器汇合或到达输入文件末尾，预测器将报告歧义，并优先选择编号最低的子解析器关联的产生式继续解析。

ALL(*) 在解析时构造向前看 DFA 的做法给文法功能测试带来额外的负担。与所有动态方法一样，程序员必须尽可能覆盖尽量多的文法位置（$\alpha \cdot \beta$，表示已分析完 α 准备分析 β）和输入序列组合，以发现文法中的歧义。标准的源代码覆盖工具可以帮助程序员测量 ALL(*) 解析器的文法覆盖率。在生成的代码中的高覆盖率就对应于文法的高覆盖率。

3.4.2 谓词文法及其含义

为了精确描述 LL(*) 解析器，需要先形式定义它们基于的谓词文法，然后介绍文法的含义。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第91页 文件编号: 422624 页面编号: 2914538448 打印人: 8685 2026-03-09 10:23:00

1. 谓词文法

ANTLR 3 的输入是带**语法谓词**（syntactic predicate）和**语义谓词**（semantic predicate）及内嵌动作的上下文无关文法。这种谓词文法 $G=(N,T,P,S,\Pi,\mathcal{M})$ 包含如下元素：N 是非终结符（规则名）的集合，T 是终结符（记号）的集合，P 是产生式的集合，$S\in N$ 是开始符号，Π 是一组无副作用的语义谓词，$\mathcal{M}$ 是一组动作（状态修改器 mutators）。产生式通过编号来表示优先级，用于解决歧义问题。产生式有如下 4 种形式：

（1）标准的产生式$A\to\alpha_i$：A 的第 i 个上下文无关产生式。

（2）带语法谓词的产生式 $A\to(A_i')\Rightarrow\alpha_i$：仅当由 A_i' 推导出的串是当前剩余输入串的前缀时，符号 A 展开为 α_i。这里 (A_i') 为语法谓词，允许向前看任意个记号，看是否匹配 A_i'。

（3）带语义谓词的产生式$A\to\{\pi_i\}?\alpha_i$：非终结符 A 仅在语义谓词 π_i 对当前已构造的状态（或上下文）成立时，才能展开为 α_i。

（4）动作产生式$A\to\{\mu_i\}$：执行特定动作 μ_i 来修改当前状态。这里的状态修改器 μ_i 用解析器的宿主语言编写，在当前状态被执行。

为了避免语法谓词单独设计识别机制，ANTLR 3 将语法谓词简化为启动预测解析的语义谓词，即语法谓词 $(A_i')\Rightarrow$ 可以替换成语义谓词 $\{\text{synpred}(A_i')\}?$。函数 synpred 返回真，当且仅当 A_i' 匹配当前输入；否则，返回假。

ANTLR 4 引入的 ALL(*) 谓词文法 $G=(N,T,P,S,\Pi,\mathcal{M})$ 也是六元组。与上述 LL(*) 谓词文法不同的是，ALL(*) 文法不再需要或支持语法谓词，即 ALL(*) 的产生式只有标准产生式、带语义谓词的产生式和动作这三种形式。ALL(*) 支持所有的非左递归文法，并能够接受直接左递归文法（ANTLR 会在内部自动消除直接左递归），但不支持任何形式的间接或隐藏的左递归。

2. 谓词文法的含义

为了准确描述 ALL(*) 解析，图 3.11 中的推导规则定义了 ALL(*) 谓词文法的含义。每条推导规则用分式表示，分子是前提，分母是结论。

标准产生式	$\dfrac{A\to\alpha_i}{(\mathbb{S},uA\delta)\Rightarrow(\mathbb{S},u\alpha_i\delta)}$	动作产生式	$\dfrac{A\to\{\mu_i\}}{(\mathbb{S},uA\delta)\Rightarrow(\mu_i(\mathbb{S}),u\delta)}$
语义谓词产生式	$\dfrac{\pi_i(\mathbb{S})\qquad A\to\{\pi_i\}?\alpha_i}{(\mathbb{S},uA\delta)\Rightarrow(\mathbb{S},u\alpha\delta)}$	闭包	$\dfrac{(\mathbb{S},\alpha)\Rightarrow(\mathbb{S}',\alpha'),(\mathbb{S}',\alpha')\Rightarrow^*(\mathbb{S}'',\beta)}{(\mathbb{S},\alpha)\Rightarrow^*(\mathbb{S}'',\beta)}$

图 3.11　谓词文法的最左推导规则

为了支持语义谓词和动作，这些规则用状态 $\mathbb{S}$ 抽象在解析过程中的用户状态。判断 $(\mathbb{S},\alpha)\Rightarrow(\mathbb{S}',\beta)$ 可以理解为：“在机器状态 $\mathbb{S}$ 下，文法符号串 α 一步最左推导到修改后的状态 $\mathbb{S}'$ 和文法符号串 β。”判断 $(\mathbb{S},\alpha)\Rightarrow^*(\mathbb{S}',\beta)$ 表示一步或多步最左推导。带语义谓词 π_i 的产生式只有在 π_i 对当前状态 $\mathbb{S}$ 为真时才会触发，记为前提 $\pi_i(\mathbb{S})$。动作产生式使用指定的变换器 μ_i 更新状态，即修改后的状态 $\mu_i(\mathbb{S})$。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第92页 文件编号: 422624 页面编号: 2924538446 打印人: 8685 2026-03-09 10:23:00

对于 ANTLR 3 支持的语法谓词，若用 w_r 表示待匹配的剩余输入。带语法谓词 A' 的产生式只有在当前状态下从 A' 推导出的字符串是剩余输入的前缀时（记作 $w \preceq w_r$）才会触发。在尝试解析 A' 期间发生的动作是试探性的，无论 A' 是否匹配，这些动作都会被撤销。

$$\frac{(\mathbb{S}, A') \Rightarrow^* (\mathbb{S}', w), w \preceq w_r \qquad A \to (A') \Rightarrow \alpha}{(\mathbb{S}, uA\delta) \Rightarrow (\mathbb{S}, u\alpha\delta)}$$

形式地，文法符号串 α 产生的语言是 $L(\mathbb{S}, \alpha) = \{w \mid (\mathbb{S}, \alpha) \Rightarrow^* (\mathbb{S}', w)\}$，文法 G 产生的语言是 $L(\mathbb{S}_0, G) = \{w \mid (\mathbb{S}_0, S) \Rightarrow^* (\mathbb{S}, w)\}$，其中 $\mathbb{S}_0$ 是初始用户状态，可以为 ε。由于谓词实际可以检查其左右的上下文，因此 $L(G)$ 的语言类是上下文有关的，而不是上下文无关的。

实际的 ANTLR 输入文法需要遵循一些语法限制，例如，强制谓词只能位于规则的左边，强制状态修改器只能用在自己的规则内。

图 3.11 中的推导规则并不能排除歧义。对于实际编程语言的解析器，每个输入都应该对应唯一的解析结果。为此，ALL(*) 按产生式选择在文法中的先后顺序来解决歧义，优先选取编号最小的产生式。

3.4.3 ANTLR 3 的 LL(*) 解析算法

在 LL(*) 解析算法中，对于文法中每个具有多个产生式选择的非终结符 A，会通过静态分析构建一个向前看 DFA。这个 DFA 用来匹配与 A 的各个产生式选择关联的正规划分 R，其中第 i 个正规分区 R_i 对应第 i 个产生式，每个 R_i 都有一个接受状态。在语法分析的决策点上，LL(*) 解析器会根据匹配剩余输入前缀的 R_i，选择相应的第 i 个产生式展开 A。因此，LL(*) 解析器的时间复杂度为 $O(n^2)$。

1. 向前看 DFA

向前看 DFA 是在普通 DFA 上增强有谓词和能输出预测产生式编号的接受状态。这种 DFA 能依据当前输入符号及文法中的谓词来确定应选择的产生式，并在接受状态输出预测的产生式编号。给定谓词文法 $G = (N, T, P, S, \Pi, \mathcal{M})$，向前看 DFA 是 $M = (\mathbb{S}, Q, \Sigma, \Delta, D_0, F)$，其中：

（1）$\mathbb{S}$ 是从外围解析器继承的系统状态；

（2）Q 是状态集合；

（3）$\Sigma = T \cup \Pi$ 是边上的字母表；

（4）Δ 是转移函数：$Q \times \Sigma \to Q$；

（5）$D_0 \in Q$ 是开始状态；

（6）$F = \{f_1, f_2, \cdots, f_n\}$ 是一组接受状态，每个正规分区 R_i(产生式选择 i) 有一个 $f_i \in Q$。

Δ 包含 DFA 原有的状态转移，即若状态 p 面临符号 $a \in \Sigma$ 转移到状态 q，记为 $p \xrightarrow{a} q$；新增了谓词转移 $p \xrightarrow{\pi} f_i$，表示状态 p 满足谓词 π 到达接受状态 f_i，但是从 p 可以有多个这

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第93页 文件编号: 422624 页面编号: 2934538444 打印人: 8685 2026-03-09 10:23:00

样的转换。

DFA 的**瞬时格局**（instantaneous configuration）c 表示为 $(\mathbb{S}, p, w_r)$，其中 $\mathbb{S}$ 是系统状态、p 是 DFA 的当前状态、w_r 是剩余输入；初始格局是 $(\mathbb{S}, D_0, w_r)$。$c \mapsto c'$ 表示 DFA 使用图 3.12 中的规则从格局 c 变为 c'。三条规则的含义依次是：

（1）如果 $p \xrightarrow{a} q$ 成立，那么格局由 $(\mathbb{S}, p, aw)$ 变为 $(\mathbb{S}, q, w)$，变换后状态为 q，剩余输入为 w。

（2）如果谓词 π_i 在系统状态 $\mathbb{S}$ 下满足，谓词转移 $p \xrightarrow{\pi_i} f_i$ 成立，那么格局 $(\mathbb{S}, p, w)$ 在满足谓词 π_i 时变为 $(\mathbb{S}, f_i, w)$。

（3）若进入格局 $(\mathbb{S}, f_i, w)$，则接受并预测用编号为 i 的产生式展开。与谓词文法类似，上述规则不能避免因谓词转移产生的 DFA 路径存在歧义。实际应用时，ANTLR 会按次序试探边来解决歧义。

$$\frac{p \xrightarrow{a} q}{(\mathbb{S}, p, aw) \mapsto (\mathbb{S}, q, w)} \qquad \frac{\pi_i(\mathbb{S}) \quad p \xrightarrow{\pi_i} f_i}{(\mathbb{S}, p, w) \xrightarrow{\pi_i} (\mathbb{S}, f_i, w)} \qquad \frac{(\mathbb{S}, f_i, w)}{\text{Accept, 预测产生式}i}$$

图 3.12　向前看 DFA 的格局改变规则

为了提高效率，向前看 DFA 只需匹配向前看集合，而不是整个后继语言的内容。例如，若 $R = (\{ac^*\}, \{bd^*\})$，向前看集合是 $(\{a\}, \{b\})$，那么没必要向前看第一个符号之后的内容。给定区分 n 个产生式的正规划分 R，第 i 个产生式的**向前看集合**仍然是唯一预测第 i 个产生式 R_i 的最小前缀集合：

$$LA_i = \{w \mid ww' \in R_i, w \notin LA_j, j \neq i, \text{且没有 } w \text{ 的前缀有相同性质}\}$$

2. 静态文法分析算法

对于谓词文法 G，为了给有多个产生式选择的非终结符 A 构造向前看 DFA，LL(*) 解析算法会先按图 3.13 中的构造规则将 G 转换为等价的**增强型转换网络**（augmented transition network, ATN）$M_G = (Q, \Sigma, \Delta, E, F)$，其中：

（1）Q 是状态集合；

（2）$\Sigma = N \cup T \cup \Pi \cup \mathcal{M}$ 是边上的字母表；

（3）Δ 是转移函数：$Q \times (\Sigma \cup \varepsilon) \to Q$；

（4）$E = \{p_A \mid A \in N\}$ 是子自动机的开始状态；

（5）$F = \{p'_A \mid A \in N\}$ 是子自动机的接受状态。

输入文法的元素	对应的 ATN 状态转换
$A \to \alpha_i$	$p_A \xrightarrow{\varepsilon} p_{A,i} \xrightarrow{\varepsilon} \boxed{\alpha_i} \xrightarrow{\varepsilon} p'_A$
$A \to \{\pi_i\}?\alpha_i$	$p_A \xrightarrow{\varepsilon} p_{A,i} \xrightarrow{\pi_i} \boxed{\alpha_i} \xrightarrow{\varepsilon} p'_A$
$A \to \{\mu_i\}$	$p_A \xrightarrow{\varepsilon} p_{A,i} \xrightarrow{\mu_i} p'_A$
$A \to \varepsilon$	$p_A \xrightarrow{\varepsilon} p_{A,i} \xrightarrow{\varepsilon} p'_A$
$\boxed{\alpha_i} = X_1 X_2 \cdots X_m$	$p_0 \xrightarrow{X_1} p_1 \xrightarrow{X_2} \cdots \xrightarrow{X_m} p_m$

图 3.13　谓词文法到 ATN 状态转换的构造

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第94页 文件编号: 422624 页面编号: 2944538442 打印人: 8685 2026-03-09 10:23:00

文法中的每个非终结符 A 都有一个对应的 ATN 子自动机。非终结符边 $p \xrightarrow{A} p'$ 类似于函数调用，它将 ATN 的控制权转移到 A 的子自动机，同时将返回状态 p' 压入状态栈，以便在 A 的子自动机到达接受状态后可以从 p' 继续执行。对于 A 的每个产生式选择 α_i，A 对应的 ATN 子自动机的起始状态 p_A 指向从产生式 α_i 的左边生成的状态 $p_{A,i}$（称为 α_i 的“调用点”）；从产生式 α_i 生成的最后一个状态指向 p'_A。

【例 3.18】 如果谓词文法 G 的产生式集合 $P = \{S \to Ac \mid Ad, A \to aA \mid b\}$，则为 G 中非终结符 S 和 A 构造的 ATN 如图 3.14 所示。

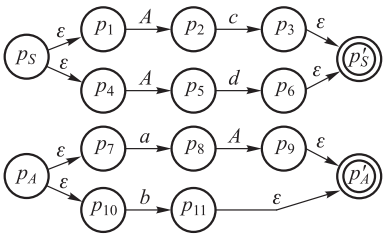


图 3.14 例 3.18 的文法对应的 ATN $P = \{S \to Ac \mid Ad, A \to aA \mid b\}$

对谓词文法的分析类似于跨过程流分析，该分析通过静态跟踪文法的 ATN 图，发现从顶层调用点（文法的开始符号）可以到达的所有节点。在分析过程中，流分析的唯一格局状态由图结点和到达该结点的调用栈组成，这与程序分析中跟踪过程调用的状态类似。根据分析的类型，还可能记录一些语义上下文信息，如从顶层调用点传递的参数。

当分析非终结符 A 的产生式选择 α_i 时，会静态跟踪从其“调用点” $p_{A,i}$ 出发、通过 ATN 可达的路径。沿着这些路径收集终结符边，形成一个向前看序列。分析持续进行，直到每个向前看序列能唯一对应一个特定的候选产生式。与此同时，还需要跟踪 α_i 左边的任何语义谓词 π_i，以便需要时解决歧义。因此，ATN 的格局表示为一个四元组 (p, i, γ, π)，其中 p 是 ATN 状态，i 是预测的产生式编号，γ 是 ATN 调用栈，π 是可选的谓词，这种格局类比为 NFA 的一个状态。

通过模拟 ATN 的动作可以计算向前看 DFA。每个向前看 DFA 的状态 D 代表 ATN 在从状态 $p_{A,i}$ 开始匹配剩余输入的一个前缀后可能处于的一组格局，它们由修改过的子集构造法得到。DFA 的状态由如下转移闭包操作创建，它通过找到在每个输入终结符 $a \in T$ 上直接可达的 ATN 状态，创建新的 DFA 状态：

$$\text{move}(D, a) = \{(q, i, \gamma, \pi) \mid p \xrightarrow{a} q, (p, i, \gamma, \pi) \in D\}$$

有关修改后的子集构造算法、DFA 状态的冲突解决算法详情可参阅文献 [8]。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第95页 文件编号: 422624 页面编号: 2954538449 打印人: 8685 2026-03-09 10:23:00

3.4.4 ANTLR 4 的 ALL(*) 解析算法

与 LL(*) 静态构造向前看 DFA 不同，ALL(*) 解析器会在解析时根据输入的句子动态适应。对于每个决策点 $A \to \alpha_1 \mid \cdots \mid \alpha_n$，解析器会调用特殊函数 adaptivePredict 来动态增量地构建向前看 DFA，记录从向前看序列到预测产生式编号的映射。该函数以非终结符和解析器调用栈作为参数，返回预测的产生式编号。如果当前构建的 DFA 与向前看序列匹配，解析器可以跳过详细分析，直接展开预测的候选产生式。如果没有可行的产生式，则抛出异常。ALL(*) 解析器的时间复杂度为 $O(n^4)$。最坏情况下，解析器必须对每个输入符号进行预测且每次都需要检查剩余的所有输入；检查单个输入符号的代价可能是 $O(n^2)$。下面重点介绍与 LL(*) 解析器的不同之处。

1. 向前看 DFA

在普通 DFA 基础上仅增添了生成预测产生式编号的接受状态，而不含谓词。这是因为向前看 DFA 是运行时动态构建的，谓词判断在构建的过程中就能决策。形式地，对于谓词文法 $G = (N, T, P, S, \Pi, \mathcal{M})$，ALL(*) 解析器所使用的向前看 DFA 是 $M = (Q, \Sigma, \Delta, D_0, F)$，其中：

（1）Q 是状态集合；

（2）$\Sigma = T$ 是边上的字母表；

（3）Δ 是转移函数：$Q \times \Sigma \to Q$；

（4）$D_0 \in Q$ 是开始状态；

（5）$F = \{f_1, f_2, \cdots, f_n\}$ 是一组接受状态，每个产生式 i 有一个 f_i。

2. 调用栈敏感的预测

为了能处理所有的非左递归文法，ALL(*) 预测算法有时必须考虑在预测开始时解析器调用栈的可用信息。考虑文法

$$S \to xB \mid yC \qquad B \to Aa \qquad C \to Aba \qquad A \to b \mid \varepsilon \tag{3.10}$$

若没有解析器栈，即便有再多的向前看 DFA，也无法唯一区分非终结符 A 的产生式。例如，向前看符号 ba 在 B 调用 A 时预测使用 $A \to b$ 展开，而在 C 调用 A 时预测使用 $A \to \varepsilon$ 展开。如果预测忽略了解析器调用栈，就会在 ba 上产生预测冲突。

忽略解析器调用栈进行预测的解析器称为**强 LL（SLL）解析器**。手动编写的递归下降解析器属于 SLL 类。按照惯例，文献中常将 SLL 称为 LL，但这里用 SLL 表示栈无关的解析、用 LL 表示栈敏感的解析。文法（3.10）是 LL(2) 的，但该文法对任何 k 来说都不是 SLL(k)，尽管将 A 产生式复制到每个 A 的调用点可以使文法变为 SLL(2)。

为每种可能的解析器调用栈创建不同的向前看 DFA 并不可行，因为随着栈深度的增加，栈的排列数呈指数级增长。相反，实际观察到大多数决策都不是栈敏感的，因此可以构建忽略解析器调用栈的向前看 DFA。如果 SLL ATN 模拟发现预测冲突，它无法确定向前看短语

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第96页 文件编号: 422624 页面编号: 2964538447 打印人: 8685 2026-03-09 10:23:00

是歧义的还是栈敏感的。在这种情况下，adaptivePredict 必须使用解析器栈重新检查向前看。这种混合或优化的 LL 模式会在可能的情况下将栈无关的预测结果缓存到向前看 DFA 中，从而提高了性能，同时保留了完整的栈敏感预测能力。

3. 两阶段 ALL(*) 解析

SLL 比 LL 更快，但功能较弱。由于实际中大多数决策是 SLL 的，因此 ALL(*) 解析算法的第一阶段用 SLL 模式解析。如果 SLL 模式发现有语法错误，它可能发现了 SLL 的局限性或真正的语法错误，这时就需要使用第二阶段的优化的 LL 模式重新尝试整个输入。这种看似反直觉的策略，尽管可能需要对整个输入进行两次解析，但可以显著提高速度，相比单纯用优化的 LL 模式更为高效。

4. ALL(*) 解析算法

为了避免对相同输入和非终结符重复进行 ATN 模拟，adaptivePredict 构建了 DFA，将输入与预测的产生式映射存储起来，每个非终结符对应一个 DFA。每个 DFA 状态 D 是在匹配向前看符号后可能的 ATN 格局集合。adaptivePredict 调用 startState 来创建初始 DFA 状态 D_0，然后调用 SLLpredict 开始模拟。SLLpredict 函数通过多次调用 target 向前看 DFA 中添加与输入符号 w_r 部分或全部匹配的路径。target 函数通过 move 和 closure 操作从当前状态 D 计算出目标 DFA 状态 D'，这些操作类似于子集构造中的操作。move 函数找到在当前输入符号上可到达的所有 ATN 格局，而 closure 找到不经过终结符边即可到达的所有格局。与子集构造的主要区别在于，closure 模拟与非终结符关联的 ATN 子自动机的调用和返回。

如果 SLLpredict 模拟发现了冲突，则会回退输入并调用 LLpredict 重新尝试预测，这次会结合调用栈来预测。LLpredict 函数类似于 SLLpredict，但不会更新非终结符的 DFA，因为 DFA 必须与调用栈无关，以便在所有栈上下文中都适用。由 LLpredict 发现的 ATN 格局集合中的冲突代表了歧义。两个预测函数都使用 getConflictSetsPerLoc 来检测冲突，该函数识别出代表相同解析器位置但不同产生式的格局集合。

为了避免不必要地调用 LLpredict，SLLpredict 使用 getProdSetsPerState 检查在 getConflictSetsPerLoc 报告冲突时是否仍然存在潜在的非冲突 DFA 路径。如果存在，这表明继续使用 SLLpredict 是值得的，因为更多的向前看可能会解决冲突，而无须完全依赖 LL 解析。

5. 图结构的调用栈

ALL(*) 解析器不采用经典的回溯策略，经典策略会为每个候选产生式 α_i 启动一个独立的子解析器来尝试解析剩余的输入。由于这些子解析器在回溯时互不知晓对方的状态，因此无法协作，也无法及时停止，导致解析器可能会出现指数级时间复杂度的问题。

为了提升效率并避免回溯带来的高复杂度，ALL(*) 解析器采取同步推进解析的方式，子解析器每一步都消费部分输入符号 w_r。预测过程在以下两种情况下终止：当只有一个子解析器存活时，或者检测到产生式之间存在冲突。

在同步推进过程中，子解析器之间还可以共享调用栈。对于 ATN 状态 p 上运行的两个子解析器，它们的 ATN 调用栈顶部是相同的（分别为 $q\gamma_1$ 和 $q\gamma_2$），那么在栈顶 q 弹出之前，

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第97页 文件编号: 422624 页面编号: 2974538445 打印人: 8685 2026-03-09 10:23:00

它们的行为是相同的。在这种情况下，预测过程可以将这些子解析器视为一个解析器，通过合并它们的栈来处理。这种合并行为有效减少了冗余的计算。

具体而言，ALL(*) 解析器通过**图结构栈**来表示多个子解析器的栈，从而合并所有具有相同 ATN 状态和产生式编号的子解析器格局。例如，两个子解析器格局 (p, i, γ_1) 和 (p, i, γ_2) 可以合并成一个公共格局 (p, i, Γ)，其中 $\Gamma = \gamma_1 \uplus \gamma_2$，这里 $\uplus$ 表示图的合并。通过这种栈合并机制，栈的大小潜在可以从指数级降低至线性复杂度。Γ 可以为空栈 [ ]、特殊栈符号 #（表示 SLL 预测）、单个栈，或者是由多个栈结点形成的图结构。图 3.15是在栈底共享解析器栈 γ_0 的两个示例。

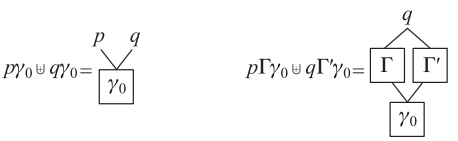


图 3.15 图结构栈：通过合并来共享解析器栈

有关 ALL(*) 原理的更深入讨论详见文献 [9]。

3.4.5 ANTLR 的使用

在了解 ANTLR 的解析原理后，接下来介绍用 ANTLR 4 构造解析器的方法及注意要点。

在学习使用 ANTLR 时，你可以访问 ANTLR Lab 在线平台。这个平台允许用户无须安装任何软件就可以在线编写和测试 ANTLR 语法。用户可以直接在浏览器中定义语法规则，模拟解析过程，并通过直观的方式可视化解析树，帮助更好地理解语法的工作原理。图 3.16展示了 ANTLR Lab 在线平台的交互界面。

1. 构造解析器的基本步骤

(1) 编写文法文件。ANTLR 4 的解析器是基于.g4 文法文件生成的，文法文件包含词法规则和语法规则。图 3.16 为 ANTLR Lab 在线实验平台的交互界面。已经介绍过 ANTLR 的词法规则，本节主要介绍语法规则。

图 3.16 的 ANTLR 文法编写区展示了一个简单表达式语言的文法示例，program 是开始符号，包含语句 (stat) 和函数定义 (def) 两种形式；stat 描述赋值和表达式语句；def 描述函数定义；expr 支持变量 (ID)、整数常量 (INT)、函数调用 (func) 和逻辑表达式。文法描述的第 1 行定义了解析器的名字为 ExprParser；第 2 行通过 options 命令指定了使用词法分析器 ExprLexer 产生的词法单元；第 4~23 行定义了文法中各个非终结符的语法规则。

(2) 使用 ANTLR 生成解析器：编写好文法文件后，可以运行以下命令：

```
$ antlr4 ExprParser.g4
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第98页 文件编号: 422624 页面编号: 2984538443 打印人: 8685 2026-03-09 10:23:00


```
$ javac ExprParser*.java
$ grun Expr program -gui sample.expr
```

先通过 antlr4（`java org.antlr.v4.Tool`）将.g4 文件编译为 Java 语言编写的解析器代码；然后用 Java 编译器 (javac) 对生成的解析器源文件进行编译，产生相应的 Java 字节码文件；再通过 grun（`java org.antlr.v4.gui.TestRig`）运行解析器来解析输入的文本。grun 工具提供了不同的选项来观察解析过程：

① -gui：以图形界面（GUI）形式展示解析树。它会生成一个窗口，直观显示解析树结构。

② -tree：在终端以文本的层次结构表示解析树的树形结构，在没有图形界面环境时很有用。

③ -trace：详细输出解析步骤，包括进入和退出每个规则时的状态，有助于调试和分析。

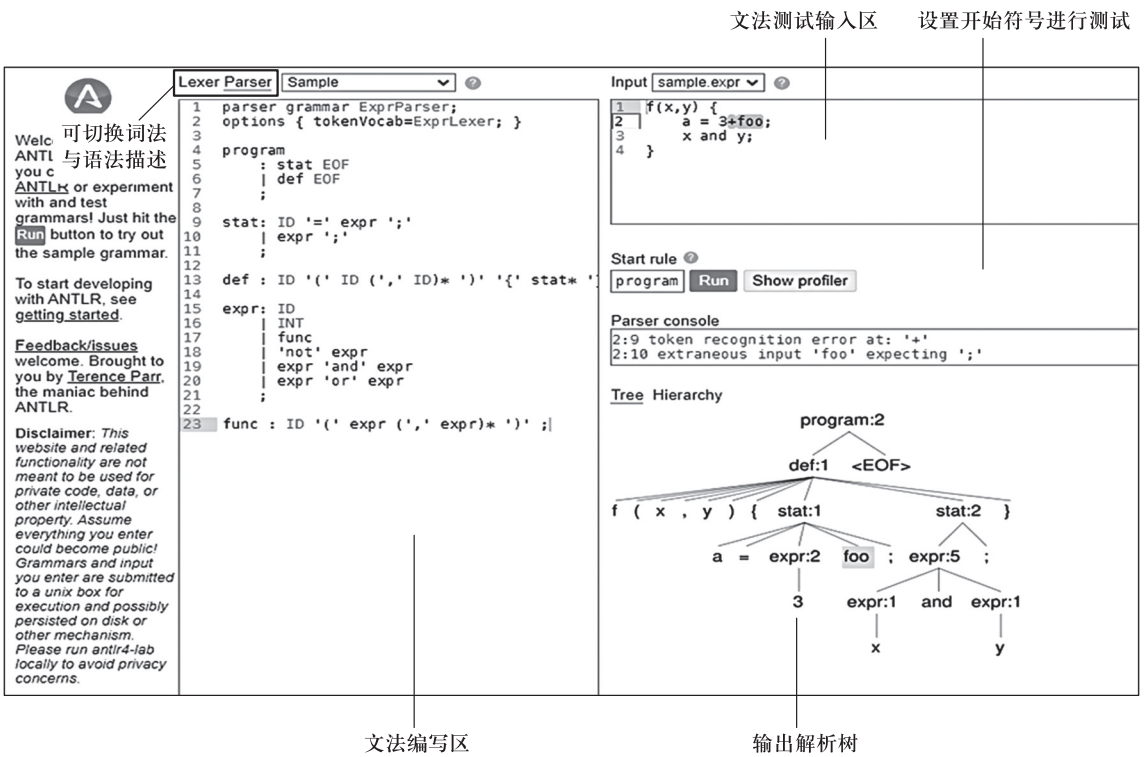


图 3.16 ANTLR Lab 在线平台的交互界面

以图 3.16 右上方的文法文件 sample.expr 为例，用户可以在图的右中部指定入口非终结符（可以是 program 或其他子结构，如 def）并运行，这时在下方会显示解析结果，一方面在控制台输出解析错误，另一方面以树形结构显示对应的解析树。用户还可以根据需要查看解析的统计信息（表 3.6），其中包含各非终结符的调用次数（Invocations）、分析时长（Time）、

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第99页 文件编号: 422624 页面编号: 2994538441 打印人: 8685 2026-03-09 10:23:00

最长向前看的记号数（Max k）、文法冲突数（Ambiguities）、DFA 缓存缺失数（DFA cache miss）等。

表 3.6 ANTLR 4 文法解析的统计信息表

Rule	Invocations	Time (ms)	Total k	Max k	Ambiguities	DFA cache miss
program:0	0	0.0	0	0	0	0
stat:1	1	0.032964	2	2	0	2
def:2	0	0.0	0	0	0	0
def:3	0	0.0	0	0	0	0
expr:4	3	0.013663	6	2	0	4
expr:5	0	0.0	0	0	0	0
expr:6	3	0.035941	3	1	0	3
func:7	2	0.001484	2	1	0	2

(3) 集成解析器代码。

首先用户可以查阅 org.antlr.v4.gui.TestRig 类的实现代码来了解怎么将生成的解析器代码集成到项目中，创建输入流，将其传递给词法分析器，然后将词法分析结果传递给语法分析器，最后语法分析器会生成解析树。图 3.17提供了一个调用生成的解析器的 Java 主程序 Main.java 的示例。执行 java Main sample.expr 命令时，程序将加载并解析 sample.expr 文件的内容，生成对应的解析树并打印。

```
1  import org.antlr.v4.runtime.*;
2  import java.nio.charset.Charset;
3  import java.nio.file.Paths;
4  import java.io.IOException;
5  public class Main {
6      public static void main(String[] args) throws Exception {
7          Charset charset = Charset.defaultCharset ();
8          CharStream input = CharStreams.fromPath(Paths.get(args[0]), charset);
9          ExprLexer lexer = new ExprLexer(input);
10         CommonTokenStream tokens = new CommonTokenStream(lexer);
11         ExprParser parser = new ExprParser(tokens);
12         //parser.setBuildParseTree(false);  // 设置不产生解析树
13         //parser.program();
14         ParserRuleContext tree = parser.program();  // 开始解析
15         System.out.println(tree.toStringTree(parser));  // 打印语法树
16     }
17 }
```

图 3.17 使用表达式语言解析器的一个简单主程序 Main.java

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第100页 文件编号: 422624 页面编号: 31004538448 打印人: 8685 2026-03-09 10:23:00

2. ANTLR 4 的语法规则

在 ANTLR 中，语法规则（parser rules）是定义语言语法的核心部分，它们描述了如何从词法分析器生成的记号序列构建更复杂的结构，如解析树或第 4 章介绍的抽象语法树（abstract syntax tree，AST）。语法规则通常具有以下结构：

规则名：选择1 | ··· | 选择n;

其中，每个选择可以是规则元素序列，也可以为空。规则元素用来指定解析器在特定时刻应该执行的操作。规则元素的主要种类及含义如表 3.7 所示。

表 3.7 语法规则中的元素

元素	含义
T、'literal'	大写字母开头的词法规则名或文本串，均代表词法单元
r	在当前输入位置匹配规则 r，相当于函数调用；r 是小写字母开头的语法规则名，代表非终结符
r [«args»]	在当前输入位置匹配规则 r，传入逗号分隔的参数列表 «args»，相当于函数调用
{«action»}	在之前的元素之后、后继元素之前立即执行动作
{«p»}?	计算语义谓词 «p»。如果谓词为假，则解析器不会继续分析
.	通配符，匹配除文件结束标志之外的任意单个记号

(1) 给规则选择加标签。在定义语法时，可以为特定的规则部分添加注释或标签。这些标签可以帮助解释规则的不同部分，尤其在调试或理解复杂的语法结构时非常有用。标签通常以#开头，后面跟随标签名，并且可以用空格或换行来分隔标签和规则。通过标签，开发者可以更直观地识别每个选择在语法规则中的功能。此外，ANTLR 还会生成带有这些标签信息的解析树，帮助开发者更好地理解和调试语法结构。例如，以下语法规则定义了一个算术表达式的结构：

```
e : e '*' e   # Mult   | e '+' e # Add   | INT     # Int ;
```

其中，e 是规则名，e'∗'e 表示一个用 '∗'连接两个 e 的表达式形式，并且这个选择被标签为 # Mult，即表示乘法表达式。ANTLR 会为每个带标签的选择生成一个对应的规则上下文类，例如 XXXParser.MultContext，并生成与该标签对应的在遍历解析树时要监听触发的事件方法，如 enter 和 exit 方法，方便监听和处理特定的语法结构。

```
1  public interface XXXListener extends ParseTreeListener {
2      void enterMult(XXXParser.MultContext ctx);
3      void exitMult(XXXParser.MultContext ctx);
4      ……
5  }
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第101页 文件编号: 422624 页面编号: 31014538446 打印人: 8685 2026-03-09 10:23:00

用户可以为不同的选择添加相同的标签，以指示在遍历解析树时，监听器会为这些选择触发相同的事件。例如，上面的算术表达式规则可以改写为

```
e : e '*' e   # BinaryOp  | e '+' e # BinaryOp  | INT     # Int ;
```

其中，e '∗'e 和 e '+'e 都被标记为 # BinaryOp，表示这两种操作符都属于二元运算操作。这样，ANTLR 会为这两种选择生成相同的上下文类 XXXParser.BinaryOpContext，以及相同的监听事件方法 enterBinaryOp 和 exitBinaryOp。

(2) 规则属性定义。与编程语言中的函数类似，规则可以有参数、返回值和局部变量。规则中还可以嵌入动作，这些动作可以使用规则定义的属性。下面是包含所有可能属性定义位置的规则的通用语法：

```
规则名[参数args] returns [返回值rvals] locals [局部变量lvars] : ... ;
```

方括号中定义的属性可以像其他变量一样使用。例如：

```
add[int x] returns [int ret] : '+' INT {$ret = $x + $INT.int; } ;
```

图 3.18展示了一个包含内嵌动作的简单算术表达式语言的文法示例。通过这个文法，可以用 ANTLR 生成词法、语法分析器。随后，通过图 3.17 中的主程序，将生成的解析器集成起来，再执行 java Main test.expr 可得图 3.19(b) 中的解析结果。

3. ANTLR 的文法复用

ANTLR 支持将多个文法文件进行复用，以减少代码重复、提升文法设计的可维护性。文法复用主要通过文法导入（import）来复用公共的语法规则或词法规则。

图 3.20(a)、(b) 分别描述了 Common 和 Common2 两个词法规范，两者的区别仅在 Common.g4 中的 ID 规则允许首字母为下划线。图 3.20(c) 描述的 Base.g4 导入了 Common 词法规范，复用其中的 ID、INT 和 WS 规则。图 3.20(d) 描述的 Extend.g4 先后导入了 Common2 词法规范和 Base 语法规范。图 3.20(e) 刻画了四个文法之间的导入继承关系，每个方框列出对应文法包含的规则名和模式标识（这里略去 WS 规则）。以 Extend 文法为例，它的 ID 和 INT 规则采用 Common2 中的定义，而 prog、stmt 规则来自 Base 文法，expr 规则在 Extend 中被重新定义。

使用图 3.20(f) 中的程序测试时，Base 解析器将 _abc 识别为 ID（见图 3.20(g)），而 Extend 解析器则检测到下划线的错误（见图 3.20(h)）。此外，Base 解析器的 expr 不含 INT，故输出的解析树中仅有一条 stmt，而 Extend 解析器输出的解析树中包含两条 stmt。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第102页 文件编号: 422624 页面编号: 31024538444 打印人: 8685 2026-03-09 10:23:00


```
grammar Expr;

@header {
//package tools;
import java.util.*;
}
@parser::members {
    /** "memory" for our calculator; variable/value pairs go here */
    Map<String, Integer> memory = new HashMap<String, Integer>();
    int eval(int left, int op, int right) {
        switch ( op ) {
            case MUL : return left * right;
            case DIV : return left / right;
            case ADD : return left + right;
            case SUB : return left - right;
        }
        return 0;
    }
}
program: stat+ ;
stat:   e ';' NEWLINE           {System.out.println($e.v);}
    |   ID '=' e ';' NEWLINE   {memory.put($ID.text, $e.v);
    System.out.println($ID.text+"="+$e.v);}
    |   NEWLINE
    ;
e returns [int v]
    : a=e op=('*'|'/') b=e  {$v = eval($a.v, $op.type, $b.v);}
    | a=e op=('+'|'-') b=e  {$v = eval($a.v, $op.type, $b.v);}
    | INT                   {$v = $INT.int;}
    | ID
      {
      String id = $ID.text;
      $v = memory.containsKey(id) ? memory.get(id) : 0;
      }
    | '(' e ')'             {$v = $e.v;}
    ;
MUL : '*' ;
DIV : '/' ;
ADD : '+' ;
SUB : '-' ;
ID  :   [a-zA-Z]+ ;      // match identifiers
INT :   [0-9]+ ;         // match integers
NEWLINE:'\r'? '\n' ;     // return newlines to parser (is end-statement signal)
WS  :   [ \t]+ -> skip ; // toss out whitespace
```

图 3.18 一个包含内嵌动作的简单算术表达式语言文法示例

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第103页 文件编号: 422624 页面编号: 31034538442 打印人: 8685 2026-03-09 10:23:00


```
a=3 +4 ;
b=a+4 ;
```

(a) sample.expr

```
a=7
b=11
(program (stat a = (e (e 3) + (e 4)) ; \n) (stat b = (e (e a) + (e 4)) ; \n))
```

(b) 解析器对 test.expr 解析的输出结果

图 3.19 输入程序 test.expr 及其解析结果

```
lexer grammar Common;
ID  : [a-zA-Z_][a-zA-Z_0-9]* ;
INT : [0-9]+ ;
WS  : [ \t\r\n]+ -> skip ;
```

(a) Common.g4

```
lexer grammar Common2;
ID  : [a-zA-Z][a-zA-Z_0-9]* ;
INT : [0-9]+ ;
WS  : [ \t\r\n]+ -> skip ;
```

(b) Common2.g4

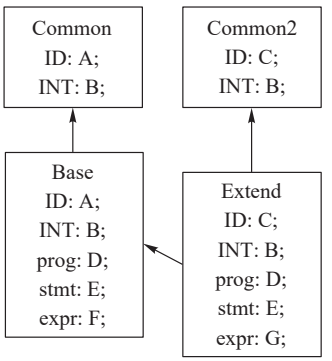
```
grammar Base;
import Common;
prog : stmt* ;
stmt : expr ';' ;
expr : ID ;
```

(c) Base.g4

```
grammar Extend;
import Common2,Base;

// 扩展表达式规则
expr : ID | INT ;
```

(d) Extend.g4



(c) 文法复用关系

```
_abc;
20;
```

(f) abc.exp

```
$ grun Base prog -tree abc.exp
(prog (stmt (expr _abc) ;))
```

(g) Base解析器的运行与结果

```
$ grun Extend  prog -tree abc.exp
line 1:0 token recognition error at: '_'
(prog (stmt (expr abc) ;) (stmt (expr 20) ;))
```

(h) Extend解析器的运行与结果

图 3.20 ANTLR 的文法复用示例

3.5 自下而上解析

本节介绍自下而上解析的基本模式，称做**移进-归约解析**。编译器常用的移进-归约解析方法称为 **LR 解析**，将在 3.6 节讨论。L 是指从左向右扫描输入，R 是指构造最右推导的逆。

3.5.1 归约

移进–归约解析为输入串构造解析树是从叶结点开始的，朝着根结点方向逆序前进。这个过程可以看作是把输入串逐步**归约**成文法的开始符号。在每一步归约中，一个子串和某个产生式的右部匹配，然后用该产生式的左部符号代替这个子串。如果每步归约都准确地选择合适的子串，那么整个归约过程实际上是最右推导过程的逆过程。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第104页 文件编号: 422624 页面编号: 31044538440 打印人: 8685 2026-03-09 10:23:00

【例 3.19】 考虑文法

$$S \to aABe$$
$$A \to Abc \mid b$$
$$B \to d$$

句子 $abbcde$ 可以按如下步骤归约成 S。首先扫描 $abbcde$，寻找能够匹配某产生式右部的子串，子串 b 和 d 都可以。选择最左边的 b, 用 A 代替它（因为有 $A \to b$），得到 $aAbcde$。现在子串 Abc、b 和 d 分别都匹配一个产生式的右部，其中 Abc 和 b 都是 $aAbcde$ 中匹配 A 产生式某右部的最左子串，用 A 代替子串 b 会使归约进行不下去，而用 A 代替子串 Abc（有 $A \to Abc$）则可以，因此得到 $aAde$。然后，因有 $B \to d$，那么用 B 代替 d，得 $aABe$，再用 S 代替此串。这样，归约序列：

$$abbcde, aAbcde, aAde, aABe, S$$

表示从 $abbcde$ 到 S 的归约。事实上，这些归约刻画出 $abbcde$ 的最右推导过程

$$S \Rightarrow_{\mathrm{rm}} aABe \Rightarrow_{\mathrm{rm}} aAde \Rightarrow_{\mathrm{rm}} aAbcde \Rightarrow_{\mathrm{rm}} abbcde$$

的逆过程，而每步归约得到的串都是右句型。

3.5.2 句柄

如前所述，归约是最右推导的逆过程，最右推导得到的句型称为右句型。右句型 γ 的**句柄**（handle）是 γ 中能和产生式（如 $A \to \beta$）右部匹配的最左子串 β，并且，把这个子串归约成产生式左部非终结符 A 得到最右推导的前一个右句型。即，如果 $S \Rightarrow^*_{\mathrm{rm}} \alpha Aw \Rightarrow_{\mathrm{rm}} \alpha\beta w$，$w$ 不含非终结符，那么 β 是右句型 $\alpha\beta w$ 的句柄。注意，如果文法二义，那么句柄可能不唯一，因为一个句子可能不止一个最右推导。只有文法无二义时，它的每个右句型才有唯一的句柄。

在例 3.19 中，$abbcde$ 是右句型，它的句柄是位于位置 2 的 b（有产生式 $A \to b$）。同样，$aAbcde$ 也是右句型，它的句柄是 Abc（有产生式 $A \to Abc$），并且也在位置 2。虽然有产生式 $A \to b$，$aAbcde$ 中位置 3 的 b 不是句柄，因为若用 A 代替 b, 得到的 $aAAcde$ 不是右句型，即不能继续归约成 S。

【例 3.20】 考虑文法

$$\begin{array}{ll} E \to E + E & E \to (E) \\ E \to E * E & E \to \mathbf{id} \end{array} \tag{3.11}$$

和最右推导：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第105页 文件编号: 422624 页面编号: 31054538447 打印人: 8685 2026-03-09 10:23:00

$$
\begin{aligned}
E &\Rightarrow_{\mathrm{rm}} \underline{E * E} \\
&\Rightarrow_{\mathrm{rm}} E * \underline{E + E} \\
&\Rightarrow_{\mathrm{rm}} E * E + \underline{\mathbf{id}_3} \\
&\Rightarrow_{\mathrm{rm}} E * \underline{\mathbf{id}_2} + \mathbf{id}_3 \\
&\Rightarrow_{\mathrm{rm}} \underline{\mathbf{id}_1} * \mathbf{id}_2 + \mathbf{id}_3
\end{aligned}
$$

为方便起见，给 **id** 以下标，并给每个右句型的句柄加下划线。例如，$\mathbf{id}_1$ 是右句型 $\mathbf{id}_1 * \mathbf{id}_2 + \mathbf{id}_3$ 的句柄。注意，句柄右边的串仅含终结符。

文法（3.11）是二义的，该句子还存在另一个最右推导：

$$
\begin{aligned}
E &\Rightarrow_{\mathrm{rm}} \underline{E + E} \\
&\Rightarrow_{\mathrm{rm}} E + \underline{\mathbf{id}_3} \\
&\Rightarrow_{\mathrm{rm}} \underline{E * E} + \mathbf{id}_3 \\
&\Rightarrow_{\mathrm{rm}} E * \underline{\mathbf{id}_2} + \mathbf{id}_3 \\
&\Rightarrow_{\mathrm{rm}} \underline{\mathbf{id}_1} * \mathbf{id}_2 + \mathbf{id}_3
\end{aligned}
$$

考虑右句型 $E * E + \mathbf{id}_3$，在这个推导中 $E * E$ 是句柄，而在上一个推导中 $\mathbf{id}_3$ 是句柄。

此例的两个最右推导类似于例 3.5 中的两个最左推导。第一推导体现 $+$ 的优先级高于 $*$，而第二个反过来。

3.5.3 用栈实现移进–归约解析

如果使用移进-归约的方式解析句子，有两个问题必须解决。一是怎样确定右句型中将要归约的子串；二是若被归约子串碰巧是多个产生式的右部时，如何确定选择哪一个产生式。在讨论这些问题之前，首先看一下移进–归约解析器所使用数据结构的类型。

实现移进–归约解析的一种简便方法是用栈保存文法符号，用输入缓冲区保存要解析的串 w，用 \$ 标记栈底，也用它标记输入串的右端。如下所示，起初栈是空的，串 w 在输入中：

栈	输入
\$	w\$

解析器移动若干个（包括零个）输入符号入栈，直到句柄 β 在栈顶为止，再把 β 归约成恰当的产生式左部。解析器重复这个过程，直到它发现错误或者栈中只含开始符号并且输入串为空为止：

栈	输入
\$$S$	\$

进入这个格局后，解析器停机并宣告解析完全成功。

【例 3.21】 逐步观察移进–归约解析器解析输入串 $\mathbf{id}_1 * \mathbf{id}_2 + \mathbf{id}_3$ 时的动作，其中文法按式（3.11）规则，并采用例 3.20 的第一种最右推导过程的逆过程，动作序列见表 3.8。

注意，由于该输入有两种最右推导，所以还存在解析器可取的另一个动作序列。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第106页 文件编号: 422624 页面编号: 31064538445 打印人: 8685 2026-03-09 10:23:00

解析器的基本动作有移进和归约，实际可能还有两种：接受和报错。

（1）**移进**：把下一个输入符号移进栈。

（2）**归约**：解析器知道句柄的右端已在栈顶，然后它确定句柄的左端在栈中的位置，再决定用什么样的非终结符代替句柄。

（3）**接受**：解析器宣告解析成功。

（4）**报错**：解析器发现语法错误，调用错误恢复例程。

在移进–归约解析中使用栈是合理的：句柄总是出现在栈顶而不是在栈的里面。

表 3.8 移进-归约解析器对于输入 $\mathbf{id}_1 * \mathbf{id}_2 + \mathbf{id}_3$ 的格局

栈	输入	动作
\$	$\mathbf{id}_1 * \mathbf{id}_2 + \mathbf{id}_3$	移进
\$ $\mathbf{id}_1$	$*\mathbf{id}_2 + \mathbf{id}_3$	按 $E \to \mathbf{id}$ 归约
\$ E	$*\mathbf{id}_2 + \mathbf{id}_3$	移进
\$ $E*$	$\mathbf{id}_2 + \mathbf{id}_3$	移进
\$ $E * \mathbf{id}_2$	$+\mathbf{id}_3$	按 $E \to \mathbf{id}$ 归约
\$ $E * E$	$+\mathbf{id}_3$	移进
\$ $E * E+$	$\mathbf{id}_3$	移进
\$ $E * E + \mathbf{id}_3$	\$	按 $E \to \mathbf{id}$ 归约
\$ $E * E + E$	\$	按 $E \to E + E$ 归约
\$ $E * E$	\$	按 $E \to E * E$ 归约
\$ E	\$	接受

从表 3.8 可以看出，在第一步归约前和每一步归约后，解析器必须移进一些（包括零个）符号，以使下一个句柄进栈，但它不需要深入栈中查找句柄。由此可知，使用栈对实现移进–归约是特别方便的。当然，为了确保移进–归约解析器正常工作，还需要合理选择动作，详见 3.6.3 小节的 LR 解析技术。

3.5.4 移进–归约解析的冲突

有些上下文无关文法不能使用移进–归约解析。这种文法的移进–归约解析器会到达这样的格局：它根据栈中所有的内容和下一个输入符号难以决定是移进还是归约（**移进–归约冲突**），或者难以决定按哪一个产生式进行归约（**归约–归约冲突**）。现在给出属于这类文法的一些语法构造的例子。从技术上讲，这些文法不属于 3.6 节定义的 LR(k) 类，称它们为**非 LR 文法**。

【例 3.22】 考察如下二义的悬空 else 文法（3.5）

$$
\begin{aligned}
\text{stmt} \to\ & \textbf{if}\ \text{expr}\ \textbf{then}\ \text{stmt} \\
\mid\ & \textbf{if}\ \text{expr}\ \textbf{then}\ \text{stmt}\ \textbf{else}\ \text{stmt} \\
\mid\ & \textbf{other}
\end{aligned}
$$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第107页 文件编号: 422624 页面编号: 31074538443 打印人: 8685 2026-03-09 10:23:00

如果移进–归约解析器当前处于某一格局

栈	输入
⋯**if** *expr* **then** *stmt*	**else** ⋯$

则不知道 **if** expr **then** stmt 是否为句柄，是继续移进，还是当成句柄进行归约，即产生移进–归约冲突。 □

不过，像这种引起冲突的二义文法仍然可以使用移进–归约解析的。例如，在这个例子中，可以采取优先移进的策略来解决这个冲突，这种优先移进也满足“else 就近匹配”的原则。3.7 节将会讨论这种二义文法的解析器。

移进–归约解析的另一种难题是，知道栈的顶部存在句柄，但根据栈里的内容和下一个输入符号不足以决定按哪个产生式归约，即产生归约-归约冲突。下面的例子说明这种情况。

【例 3.23】 假定词法分析器对任何标识符都回送记号 **id** 而不管它是如何使用的，假如语言的过程调用是给出它们的名字和参数表，并且数组元素的引用也用同样的语法。因为过程调用的参数和数组引用的下标的翻译是不一样的，所以需要用不同的产生式来产生实参表和下标表。这样，文法可以有下面一些产生式：

(1) stmt → **id**(parameter_list)
(2) stmt → expr = expr
(3) parameter_list → parameter_list, parameter
(4) parameter_list → parameter
(5) parameter → **id**
(6) expr → **id**(expr_list)
(7) expr → **id**
(8) expr_list → expr_list, expr
(9) expr_list → expr

由 $A(I, J)$ 开始的语句经词法分析后，变为记号流 **id** (**id**, **id**) 进入解析器。前三个记号移进栈后，解析器的格局是：

栈	输入
⋯**id** (**id**	, **id**)⋯

很明显，栈顶的 **id** 必须归约，但是按哪个产生式归约？如果 A 是过程，应按产生式 (5) 归约；如果 A 是数组，应按产生式 (7) 归约。但是栈中的信息不能告诉应按哪个产生式归约，必须使用符号表中有关 A 的信息。

解决这个问题的一种办法是把产生式 (1) 的记号 **id** 改为 **procid**，并且使用更智能的词法分析器，它识别出作为过程名的标识符时，返回特殊记号 **procid**。当然，这样做要求词法分析器在返回记号前访问符号表，以确定该标识符的具体类别。

这样修改后，处理 $A(I, J)$ 时解析器处于如下格局：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第108页 文件编号: 422624 页面编号: 31084538441 打印人: 8685 2026-03-09 10:23:00

栈　　输入

…procid (id , id) …

或处于先前的那个格局。前者用产生式 (5) 归约, 后者用产生式 (7) 归约。注意，栈中的第三个符号用来决定归约用的产生式，虽然它本身不包含在这个归约中。移进-归约解析可以深入栈中取信息来指导解析。

3.6 LR 解析器

本节提出一种高效的、自下而上的语法分析技术，它能适用于一大类上下文无关文法的解析。这种技术称为 **LR(k) 解析技术**，L 是指从左向右扫描输入，R 是指构造最右推导的逆，k 是指在决定分析动作时向前查看的符号个数。当 “(k)” 省略时，表示 k 是 1。

本节首先介绍 LR 解析算法，然后介绍构造 LR 分析表的 3 种技术。第一种是简单的 LR 方法（SLR），实现最为简便但功能最弱；第二种是规范的 LR 方法，功能最强但代价最大；第三种是向前搜索的 LR 方法（lookahead LR，LALR），在功能和代价上介于前两者之间。LALR 方法适用于大多数编程语言的文法，且可以高效实现。最后，例举一些非 LR 的上下文无关文法。

3.6.1 LR 解析算法

LR 解析器的模型如图 3.21 所示，它包括输入、输出、栈、驱动程序和含**动作**函数和**转移**函数两部分的分析表。驱动程序对所有的 LR 解析方法都一样，用不同的解析方法构造的分析表不同。为了快速识别栈的顶部是否形成句柄，LR 解析算法引入能概括栈中所含信息的抽象状态 s_i。驱动程序每次从输入缓冲区读一个符号，它使用栈存储形式为 $s_0X_1s_1X_2s_2\cdots X_ms_m$ 的串，s_m 在栈顶，X_i 是文法符号。这样根据栈顶的状态和当前的输入符号检索分析表，就可以决定移进–归约解析的动作。在实际实现时，文法符号不必保存在栈中，这里包含它们是为了方便解释 LR 解析的行为。

LR 解析器的格局是二元组，它的第一个成分是栈的内容，第二个成分是尚未扫描的输入：

$$(s_0X_1s_1X_2s_2\cdots X_ms_m,\ a_ia_{i+1}\cdots a_n\$)$$

在这个格局中，从栈底到栈顶的文法符号串连接尚未扫描的输入记号流，始终构成右句型：

$$X_1X_2\cdots X_ma_ia_{i+1}\cdots a_n$$

文法 G 的**活前缀**是它的右句型的前缀，该前缀不超过该右句型的最右句柄的右端。由这个定义可知，在活前缀的右端加上某个终结符串 w 后可以使它成为右句型，即

$$S \Rightarrow^*_{\mathrm{rm}} \delta Aw \Rightarrow_{\mathrm{rm}} \delta\beta w$$

因此，只要输入串的已扫描部分可以归约成一个活前缀如 $\delta\beta$，就意味着已经扫描的部分没有错误。LR 解析栈中从栈底到栈顶连接形成的文法符号串总是构成活前缀。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第109页 文件编号: 422624 页面编号: 31094538449 打印人: 8685 2026-03-09 10:23:00

分析表由两部分组成，动作函数 action 和转移函数 goto。LR 解析驱动程序的行为是：它根据栈顶当前的状态 s_m 和当前的输入符号 a_i，访问 action$[s_m, a_i]$ 来决定解析器的下一个动作。转移函数 goto 取状态和文法符号作为变元，返回一个转移到达的状态。分析表中有如下 4 种引起格局变化的动作。

（1）如果 action$[s_m, a_i]$ = 移进 s，则解析器执行移进动作，进入格局

$$(s_0X_1s_1X_2s_2\cdots X_ms_ma_is,\ a_{i+1}\cdots a_n\$)$$

即解析器把当前输入符号 a_i 和下一个状态 s 移进栈，a_{i+1} 成为当前输入符号。

（2）如果 action$[s_m, a_i]$ = 按 $A\to\beta$ 归约，则解析器执行归约动作，进入格局

$$(s_0X_1s_1X_2s_2\cdots X_{m-r}s_{m-r}As,\ a_ia_{i+1}\cdots a_n\$)$$

其中 r 是 β 的长度，s = goto$[s_{m-r}, A]$。这里解析器首先从栈中弹出 $2r$ 个符号，即 r 个状态符号和 r 个文法符号，这些文法符号刚好匹配产生式右部 β，这时栈顶暴露出状态 s_{m-r}。然后，把产生式左部符号 A 和 goto$[s_{m-r}, A]$ 的状态 s 压入栈。在执行归约时，当前输入符号不变。

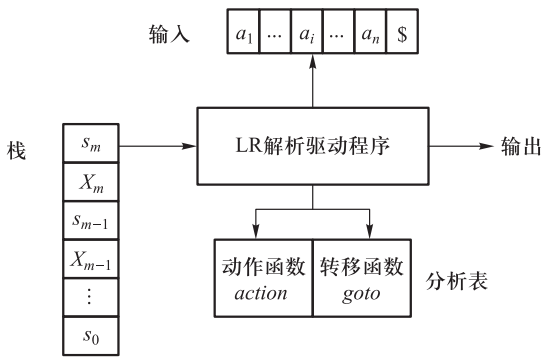


图 3.21 LR 解析器的模型

LR 解析器的输出由归约时执行的与归约产生式有关的语义动作有关，现在暂且认为输出就是打印归约产生式。

（3）如果 action$[s_m, a_i]$ = 接受，则解析完成。

（4）如果 action$[s_m, a_i]$ = 出错，则解析器发现错误，调用错误恢复例程。

由此可见，LR 解析器本质上和 3.5 节介绍的移进–归约解析器一样，只是这里栈中出现了为高效判断栈中内容的抽象状态。LR 解析算法总结在算法 3.3 中，所有 LR 解析器的行为都遵从该算法，唯一的区别在于所使用的分析表内容不一样，分析表及其关联的文法将在后续小节中讨论。

算法 3.3 LR 解析算法。

输入：输入串 w 和文法 G 的 LR 分析表。

输出：若 $w\in L(G)$，则输出 w 自下而上解析的归约步骤，否则报错。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第110页 文件编号: 422624 页面编号: 31104538447 打印人: 8685 2026-03-09 10:23:00

方法：解析器的初始格局是，初始状态 s_0 在解析器的栈顶，$w\$$ 在输入缓冲区，然后解析器执行图 3.22 的程序。

```
令 a 是 w$ 的第一个符号；
while（1）  /* 始终重复 */
    令 s 是栈顶的状态；
    if（action [s, a] == 移进 t）{
        把 a 和 t 依次压入栈；
        令 a 是下一个输入符号；
    } else if（action [s, a] == 按 A → β 归约）{
        栈顶退掉 2 * |β| 个符号；
        令 t 是现在的栈顶状态；
        把 A 和 goto [t, A] 压入栈；
        输出产生式 A → β；
    } else if（action [s, a] == 接受）break；  /* 分析完成 */
    else 调用错误恢复例程；
}
```

图 3.22　LR 解析程序

【例 3.24】　表 3.9 给出一个算术表达式文法的 LR 分析表，该算术表达式的文法如下：

(1) $E \to E + T$　　(3) $T \to T * F$　　(5) $F \to (E)$

(2) $E \to T$　　(4) $T \to F$　　(6) $F \to \mathbf{id}$

表 3.9　表达式文法的分析表

状态	动作						转移		
	id	+	*	(	)	$	E	T	F
0	$s5$			$s4$			1	2	3
1		$s6$				acc			
2		$r2$	$s7$		$r2$	$r2$			
3		$r4$	$r4$		$r4$	$r4$			
4	$s5$			$s4$			8	2	3
5		$r6$	$r6$		$r6$	$r6$			
6	$s5$			$s4$				9	3
7	$s5$			$s4$					10
8		$s6$			$s11$				
9		$r1$	$s7$		$r1$	$r1$			
10		$r3$	$r3$		$r3$	$r3$			
11		$r5$	$r5$		$r5$	$r5$			

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第111页 文件编号: 422624 页面编号: 31114538445 打印人: 8685 2026-03-09 10:23:00

动作表中各类动作的含义如下：

（1）si 表示移进，把当前输入符号和状态 i 压进栈；

（2）rj 表示按第 j 个产生式进行归约；

（3）acc 表示接受；

（4）空白表示出错。

注意，对于终结符 a，状态转移动作可以在动作表的条目 action$[s,a]$ 中找到，所以转移表中仅给出非终结符 A 的 goto$[s,A]$。

面对输入 $\mathbf{id}*\mathbf{id}+\mathbf{id}$, 栈内容和输入内容的变化序列在表 3.10 中给出。在第一行，LR 解析器处于状态 0，当前输入符号是 **id**。表 3.9 的第 0 行和 **id** 列的动作是 $s5$，即移进 **id**, 再把状态 5 压入栈，如表 3.10 第二行所示。然后，$*$ 成为当前输入符号，状态 5 面对输入 $*$ 的动作是按 $F\to\mathbf{id}$ 归约。这时两个符号（状态符号 5 和文法符号 **id**）弹出栈，状态 0 显露出来。因为 goto$[0,F]$ 是 3，所以把 F 和 3 压入栈，到达第三行所示的格局。剩余的动作也类似地决定。

表 3.10 LR 解析器对于输入 $\mathbf{id}_1*\mathbf{id}_2+\mathbf{id}_3$ 的格局变化和相应动作

栈	输入	动作
0	**id** * **id** + **id** $	移进
0 **id** 5	* **id** + **id** $	按 $F\to\mathbf{id}$ 归约
0F3	* **id** + **id** $	按 $T\to F$ 归约
0T2	* **id** + **id** $	移进
0 T 2 * 7	**id** + **id** $	移进
0 T 2 * 7 **id** 5	+ **id** $	按 $F\to\mathbf{id}$ 归约
0T2 * 7F10	+ **id** $	按 $T\to T*F$ 期约
0T2	+ **id** $	按 $E\to T$ 归约
0E1	+ **id** $	移进
0E1 + 6	**id** $	移进
0E1 + 6 **id** 5	$	按 $F\to\mathbf{id}$ 归约
0E1 + 6F3	$	按 $T\to F$ 归约
0E1 + 6 T9	$	按 $E\to E+T$ 归约
0E1	$	接受

3.6.2 LR 文法和 LR 解析方法的特点

一个文法，如果能为它构造出所有条目（若存在）都唯一的 LR 分析表，就说它是 **LR 文法**。直观上，当句柄出现在分析栈的栈顶时，如果自左向右扫描的移进-归约解析器能及时识别它，那么文法一定是 LR 的。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第112页 文件编号: 422624 页面编号: 31124538443 打印人: 8685 2026-03-09 10:23:00

LR 解析器不需要扫描整个栈就可以知道句柄是否出现在栈顶，因为栈顶的状态包含了确定句柄所需要的一切信息。学习了下面的分析表构造方法后就不难理解这句话。能够用来帮助 LR 解析器作出移进-归约决策的另一个信息源是剩余输入的前 k 个符号，最感兴趣的是 $k=0$ 或 $k=1$ 的情况，并且本书仅讨论 $k \leqslant 1$ 的情况。例如，表 3.9 仅向前搜索一个符号。如果最多向前搜索 k 个符号就可以唯一决定动作，那么这种 LR 解析器所解析的文法被称为 **LR(k) 文法**。

为了刻画 LR(k) 解析器的状态，需要为文法 G 引入一组 **LR(k) 项目**（$k \geqslant 0$）。LR(0) 项目是在文法产生式右部添加一个点，表示当前的分析位置。例如 $A \to XY \cdot Z$ 表示刚解析完由 XY 推出的输入串，期望看见由 Z 推出的串。而 LR(k) 项目则进一步增加一个长度不超过 k 的向前搜索符（lookhead），作为项目的第二个成分。例如 $[A \to XY \cdot Z, a_1a_2\cdots a_k]$ 中的第二元指出期望 XYZ 推出的输入串之后紧跟的是 $a_1a_2\cdots a_k$。所有的项目被组合成集合，每个集合对应 LR 解析器的一个状态，即识别文法 G 的活前缀的 DFA 状态。这样的项目集集合被称为**项目集规范族**，其中 LR(0) 项目集规范族用于构造 SLR 分析表，而 LR(1) 项目集规范族用于构造规范的 LR 分析表和 LALR 分析表。3.6.3 节将介绍如何从文法构造识别活前缀的 DFA。

分析表中的移进动作和 goto 函数，都是识别文法 G 的活前缀的 DFA 的转换函数。这个 DFA 的开始状态是初始时置于分析栈中的状态，LR 分析表中的所有状态都是 DFA 的接受状态，而所有的终结符、非终结符和 \$ 组成了 DFA 的字母表。从 3.6.4 节、3.6.5 节和 3.6.6 节介绍的 SLR、规范的 LR 和 LALR 分析表的构造可以进一步理解 3 种分析表在描述能力和规模上的差异。

相比 LL 解析器，LR 解析器富有吸引力的原因是：

（1）LR 解析器可以被构造来识别所有使用上下文无关文法写出的编程语言构造。3.6.7 节讨论非 LR 的上下文无关文法，典型的编程语言构造都能避免出现这种情况。

（2）LR 解析方法是已知的最一般的无回溯的移进-归约方法，能有效地实现。

（3）LR 方法能解析的文法类是预测解析法或者 LL 方法能解析的文法类的真超集。

（4）在自左向右扫描输入的前提下，LR 解析器能尽可能快地发现语法错误。

LL 方法和 LR 方法有明显的区别。LR(k) 解析器需要在看到产生式右部推出的所有内容及其后的 k 个向前搜索符号后，才确定是否用该右部归约。相比之下，LL(k) 文法要求在仅看到产生式右部的前 k 个符号时就要预测使用该产生式。因此，LR 文法能比 LL 文法描述更多的语言，因为 LR 方法的决策基于更多的已知信息。

【例 3.25】 现有句型 $\delta l\beta bw$（l 和 b 是终结符、w 是终结符串），其规范推导是

$$S \Rightarrow_{\mathrm{rm}} \cdots \Rightarrow_{\mathrm{rm}} \delta Abw \Rightarrow_{\mathrm{rm}} \delta l\beta bw$$

最后一步用的产生式是 $A \Rightarrow l\beta$。如图 3.23 所示，LL(1) 方法在看见右部 $l\beta$ 的第一个符号 l 时就必须决定用这个产生式，而 LR(1) 方法在看见右部 $l\beta$ 的后继符号 b 时决定用这个产生式。显然，与 LL(1) 方法相比，LR(1) 方法是在掌握更多信息后才决定用哪个产生式，因此

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第113页 文件编号: 422624 页面编号: 31134538441 打印人: 8685 2026-03-09 10:23:00

LR(1) 方法的表达能力较强。

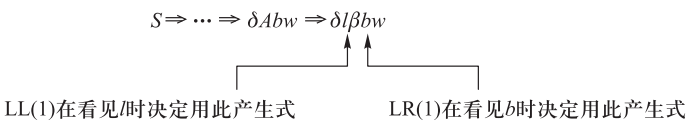


图 3.23 LL(1) 和 LR(1) 的比较

LR 方法的主要缺点是，对真正的编程语言文法，手工构造 LR 解析器的工作量非常大，因而需要专门的 LR 解析器的生成器。幸好有许多这样的工具可用，3.8 节将介绍代表性工具 Yacc。这些生成器能为上下文无关文法自动产生解析器。如果文法含二义的或者很难通过从左到右扫描进行解析的构造，那么这些生成器能定位这些构造，并提供详细的诊断信息。

3.6.3 从文法构造识别活前缀的 DFA

LR 方法的主要思想是首先从文法构造识别文法活前缀的 DFA。下面先介绍 LR(0) 项目和 LR(1) 项目，然后介绍形成 DFA 的 LR(0) 项目集规范族和 LR(1) 项目集规范族。

1. LR(0) 项目和 LR(1) 项目

文法 G 的 LR(0) 项目是在产生式右部的某个位置加点而得到的。这个点表示解析器已经看到点左侧的部分，而右侧部分尚未处理。以产生式 $A \to XYZ$ 为例，它有如下四个 LR(0) 项目：

$$A \to \cdot XYZ \qquad A \to XY \cdot Z$$
$$A \to X \cdot YZ \qquad A \to XYZ \cdot$$

其中：$A \to \cdot XYZ$ 表示期望看见由 XYZ 推出的输入串；$A \to X \cdot YZ$ 表示刚解析完由 X 推出的输入串，期望看见由 YZ 推出的串，即点的左侧代表历史信息，点的右侧代表展望信息。

对于产生式 $A \to \varepsilon$，只有一个 LR(0) 项目 $A \to \cdot$。

LR(1) 项目是在 LR(0) 项目上增加一个向前搜索符号。这个符号指示在某个特定上下文中，能否应用某个产生式，从而把更多信息并入状态。LR(1) 项目通常表示为 $[A \to \alpha \cdot \beta, a]$，其中 $A \to \alpha\beta$ 是产生式，向前搜索符 a 表示 $\alpha\beta$ 之后面临的下一个输入符号是 a。

2. 拓广文法

为了构造文法的项目集规范族，首先定义拓广文法。拓广文法旨在使解析能从唯一的项目开始（称为初始项目），并且用唯一的项目指示成功完成解析并停止（称为结束项目）。如果文法 G 的开始符号是 S，那么 G 的拓广文法 G' 是在 G 的基础上增加了新的开始符号 S' 和产生式 $S' \to S$。新增的产生式对应有表 3.11 所示的两个加点的 LR(0) 项目 $S' \to \cdot S$ 和 $S' \to S \cdot$，分别为初始项目和结束项目；而相应的 LR(1) 项目会加上向前搜索符 \$，表示解析器解析完输入串之后将面临结束标志 \$，指示解析成功，即接受输入串。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第114页 文件编号: 422624 页面编号: 31144538449 打印人: 8685 2026-03-09 10:23:00

表 3.11 拓广文法下的初始项目和结束项目

名称	SLR 解析 LR(0) 项目	规范的 LR 或 LALR 解析 LR(1) 项目	含义
初始项目	$[S' \to \cdot S]$	$[S' \to \cdot S, \$]$	指示解析器开始解析
结束项目	$[S' \to S \cdot]$	$[S' \to S \cdot, \$]$	指示解析器成功完成解析

3. 项目集的闭包

对于文法 G，项目集 I 的闭包 closure(I) 是指从初始项目集 I 开始，不读入任何输入就可以转换到达的所有项目形成的集合。这可以类比到 2.4 节中 NFA 状态的 ε – closure。

如果 I 是 LR(0) 项目集，其闭包 closure(I) 可用如下两条规则构造得到，算法如图 3.24 所示：

（1）初始时，I 的每个项目都加入 closure(I)。

（2）如果 $A \to \alpha \cdot B\beta$ 在 closure(I) 中，则对于产生式 $B \to \gamma$，若 $B \to \cdot\gamma$ 还不在 closure(I) 中，就将其加入 closure(I)。运用这条规则，直到没有更多项目可加入 closure(I) 为止。

```
setOfItems closure(I) {
    repeat
        for (I 的每个项目 A → α · Bβ )
            for (G 的每个产生式 B →γ )
                if (B → · γ 不在 I 中)
                    把 B → · γ 加入 I;
    until  在一次重复中没有项目加入 I;
    return I;
}
```

图 3.24 closure 的计算

直观上，closure(I) 中的 $A \to \alpha \cdot B\beta$ 表示在解析过程的某个时刻，期望从输入中看见的可能是由 $B\beta$ 推出的串。若有产生式 $B \to \gamma$，则从输入中也可能看见由 γ 推出的子串，为此把项目 $B \to \cdot\gamma$ 加入 closure(I)。

【例 3.26】 考虑拓广的表达式文法

$$\begin{aligned} &E' \to E && T \to T * F \mid F \\ &E \to E + T \mid T && F \to (E) \mid \mathbf{id} \end{aligned} \tag{3.12}$$

如果 I 是包含文法初始项目的 LR(0) 项目集 $\{[E' \to \cdot E]\}$，那么 closure(I) 包含下列项目：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第115页 文件编号: 422624 页面编号: 31154538446 打印人: 8685 2026-03-09 10:23:00

$$E' \to \cdot E \qquad T \to \cdot T * F \qquad F \to \cdot (E)$$
$$E \to \cdot E + T \qquad T \to \cdot F \qquad F \to \cdot \mathbf{id}$$
$$E \to \cdot T$$

由规则（1），把 $E' \to \cdot E$ 加入 closure(I)。因为 E 紧挨在点的右侧，由规则（2），把 E 产生式的、点在最左侧的项目 $E \to \cdot E + T$ 和 $E \to \cdot T$ 加入。同理，把 $T \to \cdot T * F$ 和 $T \to \cdot F$ 都加入。最后把 $F \to \cdot (E)$ 和 $F \to \cdot \mathbf{id}$ 也加入。此时，没有其他项目可由规则（2）加入。

由图 3.24 的算法可知，如果 B 在点右邻的一个项目加入 closure(I)，那么与 B 产生式对应且点在最左端的项目都加入该闭包。在空间受限的场合，并不需要在项目集中实际列出像 $B \to \cdot\gamma$ 这样的项目，而只需标出这样的非终结符 B 就足够了。从而，项目集的项目可以分成两类：

（1）核心项目：它包括表 3.11 中的初始项目和所有点不在最左端的项目。

（2）非核心项目：除去初始项目，所有点在最左端的项目，它们可通过闭包计算重新生成。

LR(1) 项目集的闭包：LR(1) 项目集的闭包函数 closure 的算法如图 3.25 所示。相比图 3.24 的算法，这里需要计算向前搜索符。考虑项目 $[A \to \alpha \cdot B\beta, a]$，对每个形式为 $B \to \gamma$ 的产生式，需要为 $B \to \cdot\gamma$ 确定向前搜索符 b。由于 $B\beta a \Rightarrow \gamma\beta a$，$\gamma\beta a \Rightarrow^* \gamma b\cdots$，于是 $b \in FIRST(\beta a)$。

4. 对活前缀有效的项目

如果 $S' \Rightarrow^*_{\text{rm}} \delta Aw \Rightarrow_{\text{rm}} \delta\alpha\beta w$，则说 LR(0) 项目 $\alpha \cdot \beta$ 对活前缀 $\delta\alpha$ 是**有效的**。因为该项目的期望部分是 β，在活前缀 $\delta\alpha$ 右边添上 β 后仍然构成活前缀。对任何活前缀 $\delta\alpha$，从其有效项目 $A \to \alpha \cdot \beta$ 可以知道，当 $\delta\alpha$ 在分析栈的栈顶时，解析器应该执行移进还是归约：如果 $\beta \neq \varepsilon$，意味着句柄尚未完全进栈，应该移进；如果 $\beta = \varepsilon$，那么 α 是句柄，应该按产生式 $A \to \alpha$ 归约。

LR(1) 项目 $[A \to \alpha \cdot \beta, a]$ 对活前缀 $\delta\alpha$ **有效**，如果存在着推导 $S' \Rightarrow^*_{\text{rm}} \delta Aw \Rightarrow_{\text{rm}} \delta\alpha\beta w$，其中搜索符 a 是 w 的第一个符号，或者 w 是 ε 且 a 是输入结束标志 \$。搜索符 a 对 β 非空的项目 $[A \to \alpha \cdot \beta, a]$ 是不起作用的；但对形式为 $[A \to \alpha \cdot, a]$ 的项目会要求只有在下一个输入符号是 a 时，才按 $A \to \alpha$ 归约。这样的 a 的集合是 FOLLOW(A) 的子集，完全可能是真子集。

【例 3.27】 考虑文法

$$S \to BB \qquad B \to bB \mid a$$

它有一个最右推导 $S \Rightarrow^+_{\text{rm}} bbBba \Rightarrow_{\text{rm}} bbbBba$。可以看出，LR(1) 项目 $[B \to b \cdot B, b]$ 对活前缀 $\gamma = bbb$ 是有效的。根据上面定义，只需令 $\delta = bb, A = B, w = ba, \alpha = b$ 且 $\beta = B$ 即可。

再看该文法的另一个最右推导 $S \Rightarrow^+_{\text{rm}} BbB \Rightarrow_{\text{rm}} BbbB$，可以看出项目 $[B \to b \cdot B, \$]$ 对活前缀 Bbb 是有效的。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第116页 文件编号: 422624 页面编号: 31164538444 打印人: 8685 2026-03-09 10:23:00

5. 转移函数

对于项目集 I、文法符号 X，转移函数 $\text{goto}(I,X)$ 表示项目集 I 代表的状态经 X 转移所到达的项目集闭包，记为 $\text{closure}(J)$。如果 I 是 LR(0) 项目集，对于 I 中形如 $[A\to\alpha\cdot X\beta]$ 的项目，经 X 转移到达的项目集 J 包含形如 $[A\to\alpha X\cdot\beta]$ 的项目。如果 I 是 LR(1) 项目集，对于 I 中形如 $[A\to\alpha\cdot X\beta,a]$ 的项目，将形如 $[A\to\alpha X\cdot\beta,a]$ 的项目加入集合 J（算法见图 3.25）。

```
setOfItems closure（I）｛
    repeat
        for(I 的每个项目[A→α · Bβ, a])
            for(G′中的每个产生式 B→γ)
                for(FIRST(βa)的每个终结符 b)
                    if([B→ · γ, b]不在 I 中)
                        把[B→ · γ, b]加到 I;
    until 在一次重复中没有项目加入 I;
    return I;
｝

setOfItems goto(I, X)｛
    置 J 为空集;
    for(I 中的每个项目[A→α · Xβ, a])
        把项目[A→α X · β, a]加到集合 J;
    return closure(J);
｝

void items(G′)｛
    置 C 的初值为｛closure(｛[S′ → · S, $]｝)｝;
    repeat
        for(C 的每个项目集 I)
            for(每个文法符号 X)
                if(goto(I, X)非空且不在 C 中)
                    把 goto(I, X)加入 C 中;
    until 在一次重复中没有项目集加入 C 中;
｝
```

图 3.25 构造文法 G' 的 LR(1) 项目集

如果 I 是对某个活前缀 γ 有效的项目集，那么 $\text{goto}(I,X)$ 就是对活前缀 γX 有效的项目集。

【例 3.28】 如果 I 是 LR(0) 项目集 $\{[E'\to E\cdot],[E\to E\cdot+T]\}$，那么 $\text{goto}(I,+)$ 包括项目：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第117页 文件编号: 422624 页面编号: 31174538442 打印人: 8685 2026-03-09 10:23:00

$$E \to E + \cdot T \qquad T \to \cdot T * F \qquad F \to \cdot (E)$$
$$T \to \cdot F \qquad F \to \cdot \mathbf{id}$$

这是通过查看 I 中的项目，看 + 是否紧挨在点的右侧来计算 goto$(I, +)$ 的。$E' \to E \cdot$ 不是这样的项目，但是 $E \to E \cdot + T$ 是。把点移过 + 得到 $J = \{[E \to E + \cdot T]\}$，然后求 closure$(J)$。

6. 项目集规范族的构造

现在可以构造拓广文法 G' 的项目集规范族了。图 3.26 给出了 LR(0) 项目集规范族的构造算法。算法3.4 给出了 LR(1) 项目集规范族的构造算法。

```
void items(G′) {
    C = {closure({[S′→ · S]})};
    repeat
        for(C 的每个项目集 I)
            for(每个文法符号 X)
                if(goto(I, X)非空并且不在 C 中)
                    把 goto(I, X)加入 C 中;
    until  在一次重复中没有新的项目加入 C;
}
```

图 3.26 LR(0) 项目集规范族的计算

算法 3.4 构造 LR(1) 项目集。

输入：拓广文法 G'。

输出：LR(1) 项目集，其中每个项目集都是对 G' 的若干个活前缀有效的项目集。

方法：构造 LR(1) 项目集的函数 closure 和 goto 及主例程 items 列在图 3.25 中。

【例 3.29】 例 3.26 文法（3.12）的 LR(0) 项目集规范族见图 3.27，这些项目集的 goto() 函数以及接受活前缀的确定有限自动机 D 的转换图如图 3.28 所示。

如果图 3.28 中 D 的每个状态都是接受状态且 I_0 是初态，则 D 识别的是文法（3.12）的活前缀。可以证明，对每个文法 G，项目集规范族的 goto() 函数定义了一个识别 G 的活前缀的 DFA。

一个活前缀可能有多个有效项目。一个活前缀 δ 的有效项目集就是从上述 DFA 的初态出发，沿着标记为 δ 的路径到达的那个项目集（状态），这是 LR 解析理论的一条基本定理。

【例 3.30】 再次考察文法（3.12）和图 3.27 和图 3.28 中。显然，串 $E + T *$ 是（3.12）的活前缀。在读完 $E + T *$ 后，图 3.28 的自动机处于状态 I_7，它包含项目

$T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第118页 文件编号: 422624 页面编号: 31184538440 打印人: 8685 2026-03-09 10:23:00

它们都对活前缀 $E+T*$ 有效，下面 3 个最右推导说明了这一点：

$$
\begin{aligned}
E' &\Rightarrow E \\
&\Rightarrow E+T \\
&\Rightarrow E+T*F \\
&\Rightarrow E+T*\mathbf{id} \\
&\Rightarrow E+T*F*\mathbf{id}
\end{aligned}
\qquad
\begin{aligned}
E' &\Rightarrow E \\
&\Rightarrow E+T \\
&\Rightarrow E+T*F \\
&\Rightarrow E+T*(E)
\end{aligned}
\qquad
\begin{aligned}
E' &\Rightarrow E \\
&\Rightarrow E+T \\
&\Rightarrow E+T*F \\
&\Rightarrow E+T*\mathbf{id}
\end{aligned}
$$

这 3 个推导分别展示了 LR(0) 项目 $T \to T* \cdot F$、$F \to \cdot(E)$ 和 $F \to \cdot\,\mathbf{id}$ 的有效性。可以证明，不存在对活前缀 $E+T*$ 有效的其他项目了。

I_0：$E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot\,\mathbf{id}$

I_1：$E' \to E \cdot$
$E \to E \cdot + T$

I_2：$E \to T \cdot$
$T \to T \cdot * F$

I_3：$T \to F \cdot$

I_4：$F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot\,\mathbf{id}$

I_5：$F \to \mathbf{id} \cdot$

I_6：$E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot\,\mathbf{id}$

I_7：$T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot\,\mathbf{id}$

I_8：$F \to (E \cdot)$
$E \to E \cdot + T$

I_9：$E \to E + T \cdot$
$T \to T \cdot * F$

I_{10}：$T \to T * F \cdot$

I_{11}：$F \to (E) \cdot$

图 3.27　文法 (3.12) 的 LR(0) 项目集规范族

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第119页 文件编号: 422624 页面编号: 31194538448 打印人: 8685 2026-03-09 10:23:00

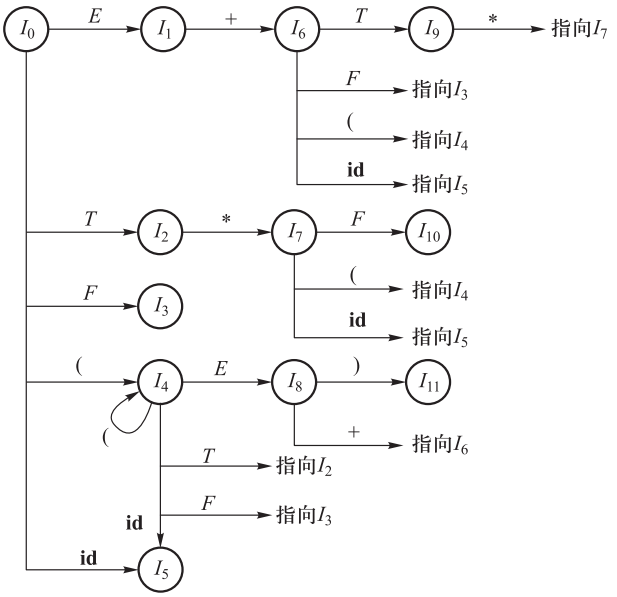


图 3.28 接受活前缀的 DFA D 的转换图

识别活前缀的 NFA。事实上，也可以构造一个识别活前缀的 NFA N，它的状态就是项目本身，从 $A \to \alpha \cdot X\beta$ 到 $A \to \alpha X \cdot \beta$ 有转换标记 X，从 $A \to \alpha \cdot B\beta$ 到 $B \to \cdot\gamma$ 有转换标记 ε。项目集（也就是 N 的状态集）I 的 closure(I) 正好是 2.4.2 小节定义的 NFA 状态集的 ε 闭包。若从 N 用子集构造法构造 DFA，那么 goto(I, X) 正是这个 DFA 的 move(I, X)。按照这种观点，图 3.26 的 items(G') 正是子集构造法本身运用于从 G' 构造的这个 NFA N。

【例 3.31】 例 3.27 文法的拓广文法为

$$\begin{aligned} S' &\to S \\ S &\to BB \\ B &\to bB \mid a \end{aligned} \tag{3.13}$$

它的 LR(1) 项目集规范族见图 3.29 所示，goto() 函数已在图中给出。其中，$[B \to \cdot bB, b/a]$ 是两个项目 $[B \to \cdot bB, b]$ 和 $[B \to \cdot bB, a]$ 的缩写。

识别该文法的活前缀的 NFA 见图 3.30，其中每个 NFA 状态是一个 LR(0) 项目。

3.6.4 从识别活前缀的 DFA 构造 SLR 分析表

以上完成了构造分析表的第一步，即从文法 G 的拓广文法 G' 构造识别活前缀的 DFA。现在进行第二步，从识别活前缀的 DFA 构造 LR 分析表的动作函数和转移函数。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第120页 文件编号: 422624 页面编号: 31204538446 打印人: 8685 2026-03-09 10:23:00

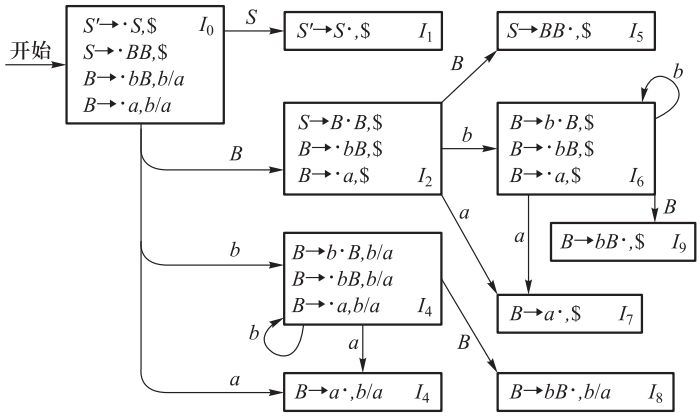


图 3.29 文法（3.13）的 DFA 状态转换图

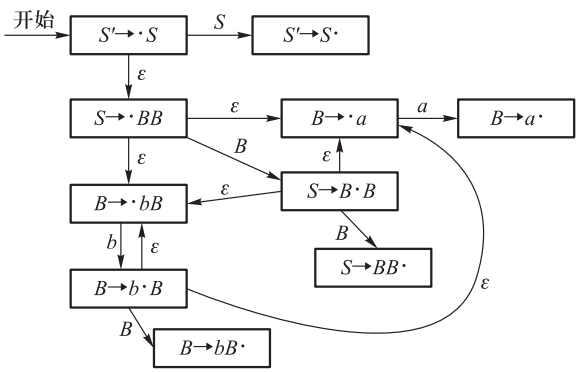


图 3.30 文法 (3.13) 的 NFA 转换图

如果一个 DFA 状态有至少两个有效的 LR(0) 项目，这可能要求解析器采取不同的动作。一些动作冲突可以由向前搜索下一个输入符号来解决，即下面的 SLR(1) 分析表的构造；还有一些则用 LR(1) 项目集表示识别活前缀的 DFA 状态，参见 3.6.5 小节中规范的 LR 分析表构造。不过，当 LR 方法用于构造任意文法的分析表时，不要期望所有分析动作的冲突都可以解决。

1. SLR 分析表的构造

从拓广文法 G' 构造 LR(0) 项目集规范族 C 后，使用算法 3.5 可以构造 SLR 分析表的动作函数 action() 和转移函数 goto()。该算法不能为任何文法都产生 SLR 分析表，但是它对许多编程语言的文法是有效的。这个算法需要知道每个非终结符后继符号集合（见 3.3.4 小节）。

算法 3.5 构造 SLR 分析表。

输入：拓广文法 G'。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第121页 文件编号: 422624 页面编号: 31214538444 打印人: 8685 2026-03-09 10:23:00

输出：G' 的 SLR 分析表的 action() 函数（动作表）和 goto() 函数（转移表）。

方法：（1）构造 G' 的 LR(0) 项目集规范族 $C = \{I_0, I_1, \cdots, I_n\}$。

（2）状态 i 从 I_i 构造。action() 函数在状态 i 的值按如下方法确定：

①如果 $[A \to \alpha \cdot a\beta]$ 在 I_i 中，并且 $\text{goto}(I_i, a) = I_j$，那么置 $\text{action}[i, a]$ 为 sj，其含义是把 a 和状态 j 移进栈。此时 a 必须是终结符。

②如果 $[A \to \alpha \cdot]$ 在 I_i 中，那么对 FOLLOW(A) 中的所有 a，置 $\text{action}[i, a]$ 为 rj, j 是产生式 $A \to \alpha$ 的编号。这个动作的意思是按产生式 $A \to \alpha$ 归约，此时 A 不可以是 S'。

③如果 $[S' \to S \cdot]$ 在 I_i 中，那么置 $\text{action}[i, \$]$ 为接受 acc。

如果由上述规则产生的动作出现冲突，那么该文法就不是 SLR(1) 的。在这种情况下，该算法会失败，无法生成相应的解析器。

（3）使用下面规则构造 goto() 函数在状态 i 的值：

对所有的非终结符 A，如果 $\text{goto}(I_i, A) = I_j$, 那么 $\text{goto}[i, A] = j$。

（4）不能由规则（2）和规则（3）定义的条目都是空白，表示出错。

（5）解析器的初始状态是包含 $[S' \to \cdot S]$ 的项目集对应的状态。

由算法 3.5 决定的动作函数和转移函数组成的分析表称为文法 G 的 **SLR(1) 分析表**，使用 G 的 SLR(1) 分析表的 LR 解析器称为 G 的 **SLR(1) 解析器**，有 SLR(1) 分析表的文法称为 **SLR(1) 文法**。通常省略 SLR 后面的 “(1)”，因为本书不讨论向前搜索两个或更多符号的解析器。

【例 3.32】 为文法式（3.12）构造 SLR 分析表，它的 LR(0) 项目集规范族已在图 3.27 中给出。首先考虑项目集 I_0：

$$
\begin{array}{lll}
E' \to \cdot E & T \to \cdot T * F & F \to \cdot (E) \\
E \to \cdot E + T & T \to \cdot F & F \to \cdot \mathbf{id} \\
E \to \cdot T &&
\end{array}
$$

项目 $F \to \cdot (E)$ 使得条目 $\text{action}[0, (] = s4$, 项目 $F \to \cdot \mathbf{id}$ 使得条目 $\text{action}[0, \mathbf{id}] = s5$。$I_0$ 的其他项目不产生动作。现在考虑项目集 I_1：

$$
\begin{array}{l}
E' \to E \cdot \\
E \to E \cdot + T
\end{array}
$$

第一个项目使得 $\text{action}[1, \$] = acc$, 第二个项目使得 $\text{action}[1, +] = s6$。再考虑项目集 I_2：

$$
\begin{array}{l}
E \to T \cdot \\
T \to T \cdot * F
\end{array}
$$

因为 $\text{FOLLOW}(E) = \{\$, +,)\}$, 所以第一个项目使得 $\text{action}[2, \$] = \text{action}[2, +] = \text{action}[2,)] = r2$（因为 $E \to T$ 的序号为 2），第二个项目使得 $\text{action}[2, *] = s7$。以这种方法继续下去可得到表 3.9 的动作表和转移表。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第122页 文件编号: 422624 页面编号: 31224538442 打印人: 8685 2026-03-09 10:23:00

2. 描述能力有限的 SLR

每个 SLR(1) 文法都不是二义的，但是有很多非二义的文法也不是 SLR(1) 的，这说明 SLR(1) 文法的描述能力有限。

【例 3.33】 对于如下文法：

$$\begin{array}{lll} S \to V = E & V \to *E & E \to V \\ S \to E & V \to \mathbf{id} & \end{array} \tag{3.14}$$

可以把 V 和 E 分别想象成代表左值和右值，左值表示一个存储单元，右值是一个可存储的值。$*$ 表示“取单元内容”的运算符。文法（3.14）的 LR(0) 项目集规范族如图 3.31 所示。

I_0：$S' \to \cdot S$
$S \to \cdot V = E$
$S \to \cdot E$
$V \to \cdot *E$
$V \to \cdot \mathbf{id}$
$E \to \cdot V$

I_1：$S' \to S \cdot$

I_2：$S \to V \cdot = E$
$E \to V \cdot$

I_3：$S \to E \cdot$

I_4：$V \to * \cdot E$
$E \to \cdot V$
$V \to \cdot *E$
$V \to \cdot \mathbf{id}$

I_5：$V \to \mathbf{id} \cdot$

I_6：$S \to V = \cdot E$
$E \to \cdot V$
$V \to \cdot *E$
$V \to \cdot \mathbf{id}$

I_7：$V \to *E \cdot$

I_8：$E \to V \cdot$

I_9：$S \to V = E \cdot$

图 3.31 文法（3.14）的 LR(0) 项目集规范族

考察项目集 I_2，这个集合的第一项目使得 action$[2,=]$ $=$ $s6$。由于 FOLLOW(E) 包含 $=$（因为 $S \Rightarrow V = E \Rightarrow *E = E$），第二项目使得 action$[2,=]$ 为按 $E \to V$ 归约。这样，action$[2,=]$ 有多重定义的条目，既有移进又含有归约，因此状态 2 在输入符号是 $=$ 时有移进-归约冲突。

实际上，文法（3.14）并不是二义的，项目集 I_2 的冲突应该是可以避免的。“=”虽然是 E 的一个后继符号，但通过观察 $S \Rightarrow V = E \Rightarrow *E = E$ 等推导，文法并不存在以“$E = \cdots$”开始的右句型，因此解析器此时不应该把 V 归约为 E。在状态 2，只有下一个输入是 \$ 时，才能用产生式 $E \to V$ 归约。这说明 SLR 方法在把项目集 I_2 的归约项目填入分析表时，掌握的信息不够精确。改进的方法是增加向前搜索符信息，即引入 LR(1) 项目 $[E \to V \cdot, \$]$，从而使得每个状态比 SLR 的状态包含更准确的信息。这就是下面要讨论的规范 LR 方法。一般来说，规范 LR 方法会导致状态数大大增加，而 LALR 方法是通过损失一点精度来达到减少状态数的目的。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第123页 文件编号: 422624 页面编号: 31234538440 打印人: 8685 2026-03-09 10:23:00

3.6.5 构造规范的 LR 分析表

算法 3.6 给出了从 LR(1) 的项目集构造 LR(1) 解析的动作函数和转移函数的规则。与构造 SLR 分析表的算法 3.5 相比，重要的区别是在状态 i 下面临什么样的输入符号时用产生式 $A \to \alpha$ 归约。对于 SLR 解析，是在遇到 FOLLOW(A) 中的任一符号时归约；而对于规范的 LR 解析，则是在遇到 I_i 中包含的 LR(1) 项目 $[A \to \alpha \cdot, a]$ 中指定的搜索符 a 时进行归约。显然，后者能更准确地确定归约动作。

算法 3.6 构造规范的 LR 分析表。

输入：拓广文法 G'。

输出：文法 G' 的规范 LR 解析的 action() 函数和 goto() 函数。

方法：（1）构造 G' 的 LR(1) 项目集规范族 $C = \{I_0, I_1, \cdots, I_n\}$。

（2）从 I_i 构造解析器的状态 i, action() 函数在状态 i 的值按如下方法确定：

① 如果 $[A \to \alpha \cdot a\beta, b]$ 在 I_i 中，且 goto(I_i, a) $= I_j$，那么置 action$[i, a]$ 为 sj, 这里，a 一定是终结符。

②如果 $[A \to \alpha \cdot, a]$ 在 I_i 中，且 $A \neq S'$，那么置 action$[i, a]$ 为 rj，j 是产生式 $A \to \alpha$ 的序号。

③如果 $[S' \to S \cdot, \$]$ 在 I_i 中，那么置 action$[i, \$] = acc$。

如果用上面规则构造出现了冲突，那么该文法就不是 LR(1) 的。该算法对此文法不适用。

（3）goto 函数在状态 i 的值按如下方法确定：

如果 goto(I_i, A) $= I_j$, 那么 goto$[i, A] = j$。

（4）用规则（2）和规则（3）未能定义的所有条目都是空白，表示出错。

（5）解析器的初始状态是包含 $[S' \to \cdot S, \$]$ 的项目集对应的状态。

用算法3.6 产生的动作函数和转移函数构成的表称为**规范的 LR(1) 分析表**，使用这种表的 LR 解析器称为**规范的 LR(1) 解析器**。如果动作函数没有多重定义的条目，那么该文法称为 **LR(1) 文法**。与前面一样，如果不会引起误解，那么通常省略 LR 后面的“(1)”。

【例 3.34】 文法（3.13）的规范 LR 分析表见表 3.12，产生式 1、2 和 3 分别是 $S \to BB$，$B \to bB$ 和 $B \to a$。

所有的 SLR(1) 文法都是 LR(1) 文法，但对于 SLR(1) 文法，规范的 LR 解析器可能比同一文法的 SLR 解析器有更多的状态。文法（3.13）是 SLR 文法，它的 SLR 解析器只有 7 个状态，而规范的 LR 解析器有 10 个状态（DFA 见图 3.29，分析表见表 3.12）。

对于例 3.33，当构造它的规范 LR(1) 分析表时，可以看到不存在任何冲突。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第124页 文件编号: 422624 页面编号: 31244538448 打印人: 8685 2026-03-09 10:23:00

表 3.12 文法（3.13）的规范 LR 分析表

状态	动作			转移	
	b		\$	S	B
0	$s3$	$s4$		1	2
1			acc		
2	$s6$	$s7$			5
3	$s3$	$s4$			8
4	$r3$	$r3$			
5			r^1		
6	$s6$	$s7$			9
7			$r3$		
8	$r2$	$r2$			
9			r^2		

3.6.6 构造 LALR 分析表

LALR 方法的描述能力介于 SLR 和规范 LR 之间，是实际编译器常用的方法。对于同一文法，LALR 分析表状态数量与 SLR 分析表的相同，但比规范 LR 的小得多。大多数编程语言的语法构造都能通过 LALR 文法方便地表示，尽管 SLR 在多数情况下也能处理，但某些构造需要用 LALR 才能解决（见例 3.33）。例如，对 C 语言，SLR 和 LALR 的分析表通常有几百个状态，而规范 LR 的分析表则有几千个状态。因此，使用 SLR 和 LALR 分析表比使用规范 LR 分析表更加经济。

再次考虑文法式（3.13），它的 LR(1) 项目集见图 3.29。其中 I_3 和 I_6、I_4 和 I_7、I_8 和 I_9 这三对 LR(1) 项目集有一个共同特征：每对项目集的第一成分都包含相同的 LR(0) 项目集，即略去搜索符后它们是相同的集合，称这样的一对 LR(1) 项目集是**同心的**。

以 I_4 和 I_7 为例，它们的第一成分都是 $B \to a\cdot$，I_4 的搜索符是 b 或 a，I_7 的搜索符是 \$。观察一下它们在解析器中的不同作用可以发现，文法（3.13）产生的是正规语言 b^*ab^*a。

当读入串 $bb\cdots babb\cdots ba$ 时，解析器会把第一组 b 和它后面的 a 移进栈，进入状态 4。如果接下来的输入符号是 b 或 a，解析器会按产生式 $B \to a$ 归约，因为 b 或 a 是第 2 个 b^*a 的开始符号。如果接下来的输入符号是 \$，那么整个输入实际是 $bb\cdots ba$ 的形式，它并不属于这个语言，此时状态 4 能正确地指出错误。

在读过第二个 a 后，解析器进入状态 7。这时解析器必须看到输入结束标记 \$，否则输入串就不符合 b^*ab^*a 的形式。因此，合理的做法是：当面临 \$ 时，状态 7 应按 $B \to a$ 归约，而在面临 b 或 a 时则报告错误。

现在将同心项目集 I_4 和 I_7 合并为 I_{47}，并将相同 LR(0) 项目的搜索符合并，形成 $[B \to a\cdot, b/a/\$]$。从 I_0、I_2、I_3 和 I_6 到达 I_4 或 I_7 的 a 转移现在都会进入 I_{47}，状态 I_{47} 的动作是

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第125页 文件编号: 422624 页面编号: 31254538445 打印人: 8685 2026-03-09 10:23:00

不管面临任何符号都归约。修改后的解析器的行为本质上和原来的一样，但它会把输入 bba 的 a 归约成 B，把输入 $babab$ 的第二个 a 归约成 B，而原来的解析器在这两种情况下都会报错。幸好，这些错误最终还是会被修改后的解析器捕获，而且是在移进下一个输入符号前被捕获。

对 LR(1) 文法，如果把所有的同心集合并，需要修改 action() 函数和 goto() 函数。对于 goto() 函数的修改不会引起问题，因为 $\text{goto}(I, X)$ 的结果项目集的心仅依赖于 I 的心，所以合并后的集合的 goto() 函数的结果也可以合并。

然而，对 action() 函数的修改需要能够反映合并前所有项目集的非出错动作。需要注意的是，同心集合并可能会导致冲突，但这种冲突不会是移进–归约冲突。因为如果存在这种冲突，说明当面临输入符号 a，对某项目 $[A \to \alpha \cdot, a]$ 需要进行归约操作，而另一项目 $[B \to \beta \cdot a\gamma, b]$ 要求将 a 移进。既然这两个项目都处于合并之后的项目集中，则在合并前必有某个 c 使得 $[A \to \alpha \cdot, a]$ 和 $[B \to \beta \cdot a\gamma, c]$ 同处于合并前的某一集合中。这意味着原来的 LR(1) 项目集已经存在移进–归约冲突，从而文法不是 LR(1) 的。因此，同心集的合并不会引发新的移进–归约冲突。

不过，同心集的合并有可能产生新的归约–归约冲突，例 3.35 就说明了这一问题的可能性。

【例 3.35】 考虑文法

$$\begin{array}{ll} S' \to S & A \to c \\ S \to aAd \mid bBd \mid aBe \mid bAe & B \to c \end{array}$$

它只产生 4 个串：acd、bcd、ace 和 bce。构造该文法的 LR(1) 项目集不出现冲突，它是 LR(1) 文法。在它的项目集中，对活前缀 ac 有效的项目集为 $\{[A \to c \cdot, d], [B \to c \cdot, e]\}$，对活前缀 bc 有效的项目集为 $\{[A \to c \cdot, e], [B \to c \cdot, d]\}$，这两集合都不含冲突，它们是同心的。然而，它们合并后变成 $\{[A \to c \cdot, d/e], [B \to c \cdot, d/e]\}$，出现归约–归约冲突，因为面临 e 或 d 时，不知道应该用哪个产生式进行归约。 □

算法 3.7 给出 LALR 分析表的构造算法，其基本思想是，首先构造 LR(1) 项目集规范族，如果不存在冲突，则合并同心集，再按合并后的项目集构造分析表。这个方法可以作为描述 LALR(1) 文法的基本定义。由于该方法先构造完整的 LR(1) 项目集规范族，需要较多的时间和空间，因此该方法只具有概念上的意义。在实际应用中，通常使用避免构造完整 LR(1) 项目集规范族的算法，有兴趣的读者可以查阅相关书籍。

算法 3.7 一个简易但时空开销大的 LALR 分析表构造法。

输入：拓广文法 G'。

输出：G' 的 LALR 分析表的 action() 函数和 goto() 函数。

方法：（1）构造 LR(1) 项目集规范族 $C = \{I_0, I_1, \cdots, I_n\}$。

（2）寻找 LR(1) 项目集规范族中同心的项目集，用它们的并集代替它们。

（3）令 $C' = \{J_0, J_1, \cdots, J_m\}$ 是合并后的 LR(1) 项目集族。action() 函数在状态 i 的值按

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第126页 文件编号: 422624 页面编号: 31264538443 打印人: 8685 2026-03-09 10:23:00

算法 3.6 从 J_i 构造。如果出现分析动作的冲突，则算法不能产生分析表，此文法不是 LALR(1) 的。

（4）goto() 函数按如下方式构造：如果 J 是若干个 LR(1) 项目集的并集，即 $J = I_1 \cup I_2 \cup \cdots \cup I_m$，那么 $\text{goto}(I_1, X), \text{goto}(I_2, X), \cdots, \text{goto}(I_m, X)$ 也同心，因为 $I_1, I_2, \cdots, I_m$ 都同心。记 K 为所有和 $\text{goto}(I_1, X)$ 同心的项目集的并集，那么 $\text{goto}(J, X) = K$。

由算法 3.7 产生的表称为 G 的 **LALR 分析表**。如果没有分析动作的冲突，那么该文法称为 **LALR(1) 文法**。在第（3）步构造的项目集族称为 **LALR(1) 项目集族**。

【例 3.36】 再次考虑文法（3.13），它的转换图见图 3.29。如已提到的那样，有三对项目集可以合并，I_3 和 I_6 合并成

$$\begin{aligned} I_{36} \quad : \quad & B \to b \cdot B, b/a/\$ \\ & B \to \ \cdot bB, b/a/\$ \\ & B \to \ \cdot a, b/a/\$ \end{aligned}$$

I_4 和 I_7 合并成

$$I_{47} : B \to a \cdot, b/a/\$$$

I_8 和 I_9 合并成

$$I_{89} : B \to bB \cdot, b/a/\$$$

压缩项目集后的 LALR 动作函数和转移函数如表 3.13 所示。

表 3.13 文法（3.13）的 LALR 分析表

状态	动作			转移	
	b	a	\$	S	B
0	$s36$	$s47$		1	2
1			acc		
2	$s36$	$s47$			5
36	$s36$	$s47$			89
47	$r3$	$r3$	r^3		
5			$r1$		
89	r^2	r^2	$r2$		

为了理解转移函数的计算，考察 $\text{goto}(I_{36}, B)$，在原来的 LR(1) 项目集中，$\text{goto}(I_3, B) = I_8$，而 I_8 现在是 I_{89} 的一部分，因此置 $\text{goto}(I_{36}, B) = I_{89}$。如果考察 I_{36} 的另一部分 I_6，可以得到同样的结论，即 $\text{goto}(I_6, B) = I_9$，而 I_9 现在是 I_{89} 的一部分。再看 $\text{goto}(I_2, b)$，它指出了在面临 b 执行了 I_2 的移进动作之后的转移方向。原来的 LR(1) 项目集中，$\text{goto}(I_2, b) = I_6$，因为 I_6 现在是 I_{36} 的一部分，所以 $\text{goto}(I_2, b) = I_{36}$。于是，分析表中状态 2 面临 b 的条目是 $s36$，其含义是移进 b，再把状态 36 压在栈顶。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第127页 文件编号: 422624 页面编号: 31274538441 打印人: 8685 2026-03-09 10:23:00

当输入串为 b^*ab^*a 时，不论是表 3.12 的 LR 解析器还是表 3.13 的 LALR 解析器，都会产生相同的移进–归约序列，仅状态名不同。这表明，在输入串正确的情况下，LR 和 LALR 解析器的行为是同步的，始终如影随形。然而，当输入串有错时，LALR 解析可能会进行一些不必要的归约，但不会比 LR 解析多进行移进操作。也就是说，在准确指出输入串的出错位置方面，LALR 解析与 LR 解析是等效的。例如，若输入串是 bba 时，LALR 解析的动作序列见表 3.14。

表 3.14 LALR 解析器对于输入 bba 的格局

	栈	符号	输入	动作
（1）	0		bba \$	移进
（2）	0 36	b	ba\$	移进
（3）	0 36 36	bb	a \$	移进
（4）	0 36 36 47	bba	\$	按 $B \to a$ 归约
（5）	0 36 36 89	bbB	\$	按 $B \to bB$ 归约
（6）	0 36 89	bB	\$	按 $B \to bB$ 归约
（7）	02	B	\$	报告错误

而如果用 LR 解析，解析器将把状态 0 3 3 4 压进栈，并在状态 4 发现错误，因为状态 4 面临 \$ 的条目内容是空白。两者的区别是 LR 解析能及时发现错误。

从 SLR、规范 LR 和 LALR 分析表的构造和使用可以看出，实际操作中只需将状态压入栈，而无须将文法符号进栈，因为解析过程中并不会根据栈中的文法符号来决定动作。之前介绍中把文法符号压栈是为了便于直观理解。表 3.14 的动作序列就是按文法符号不进栈的方式给出。第 3 列的符号也可以省略。从现在开始，只讨论仅将状态进栈。

例 3.35 的文法是规范的 LR(1) 的，但不是 LALR(1) 的。例 3.33 的文法是 LALR(1) 的，但不是 SLR(1) 的。这两个例子说明这三类文法的集合是不同的。

3.6.7 非二义且非 LR 的上下文无关文法

在 3.6.2 小节曾经提到，若自左向右扫描的移进–归约解析器能及时识别出现在栈顶的句柄，那么相应的文法就是 LR 的。而二义文法一定不是 LR 的，那么是否存在非二义并且非 LR 的文法呢？回答是肯定的，可以用例子来说明。

【例 3.37】 语言 $L = \{ww^R \mid w \in (a|b)^*\}$ 的文法

$$S \to aSa \mid bSb \mid \varepsilon$$

不是 LR 的。直观上说，对于该语言的任何句子，如 $abaaba$，扫描前一半字符时应该压栈，扫描后一半字符时先做一次空归约（ε 归约），然后将剩余字符和栈中的字符按 $S \to aSa$ 或

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第128页 文件编号: 422624 页面编号: 31284538449 打印人: 8685 2026-03-09 10:23:00

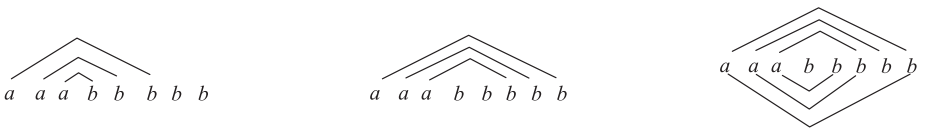
$S \to bSb$ 归约进行比较，以保证后一半是前一半的逆。问题是，向前搜索一个字符无法判断是否已到达串的中点。因此，该文法不是 LR(1) 的，构造分析表时肯定会出现移进–归约冲突。事实上，对于任意大的 k，总能找到一个句子，即使是向前搜索 k 个字符也无法判断是否应该做空归约了。因此，该文法不是 LR(k) 的。

【例 3.38】 为语言

$$L = \{a^m b^n \mid n > m \geqslant 0\}$$

写 3 个文法。它们分别是 LR(1) 的、二义的和非二义且非 LR(1) 的。

该语言的句子是 $aa\cdots abb\cdots b$ 的形式，但后面 b 的个数比前面 a 的个数多。为了保证 b 出现在 a 的后面，并且 b 的个数不少于 a 的个数，那么应该有形如 $S \to aSb$ 这样的产生式。为了能推导出更多的 b，应该有形如 $S \to Sb$ 的产生式。前一个产生式用来把 b 和前面的 a 进行配对，由于 b 的个数比 a 的多，配对方式可以有多种，不同配对方式形成不同的文法。图 3.32 为三种不同配对方式。



(a) (b) (c)

图 3.32 三种不同配对方式

如果将 $a^m b^n$ 的前 m 个 b 和 a^m 配对，如图 3.32 (a) 所示，那么按此特点写出的文法

$$\begin{aligned} S &\to AB \\ A &\to aAb \mid \varepsilon \\ B &\to Bb \mid b \end{aligned}$$

是 LR(1) 文法。

如果将 $a^m b^n$ 的后 m 个 b 和 a^m 配对，如图 3.32 (b) 所示，那么按此特点写出的文法

$$\begin{aligned} S &\to aSb \mid B \\ B &\to Bb \mid b \end{aligned}$$

是非二义且非 LR(1) 文法。因为在解析时，当把所有的 a 压进栈后，要先将中间若干个 b 归约成 B，使得剩下的 b 和前面的 a 一样多。向前搜索一个符号是不可能确定将栈顶的 B 归约成 S，还是移进下一个 b 并把 Bb 归约成 B。

如果让 $a^m b^n$ 中的 a 和 b 有不止一种配对方式，如图 3.32 (c) 所示，那么它的文法就是二义的，如

$$S \to aSb \mid Sb \mid b$$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第129页 文件编号: 422624 页面编号: 31294538447 打印人: 8685 2026-03-09 10:23:00

3.7 二义文法的应用

任何二义文法绝不是 LR 文法，因而不属于 3.6 节讨论的任何一类文法。但是，正如在本节要介绍的，某些二义文法对规范和实现语言是有用的。像表达式这样的语言构造，二义文法提供的规范比任何其他非二义文法提供的都要简短且更自然。另外，为了便于对某个语法构造的特殊情况安排特别的语义处理，需要在文法中增加针对这种特殊情况的产生式，以便把它从该语法构造的一般情况中分离出来，这种产生式的加入会使文法二义，这是二义文法的另一应用。

必须强调，虽然使用的文法是二义的，但若对某种二义情况都有消除二义的规则，保证每个句子正好只有唯一的解析树，那么对应的语言是无二义的。在这种情况下，可以为该语言设计一个 LR 解析器，使其消除二义的选择与语言规范中消除二义的规则一致。

3.7.1 使用运算符的优先级和结合性来解决冲突

考虑编程语言的表达式。下面含有二元运算符 $+$ 和 $*$ 的算术表达式文法是有二义性的，因为它没有指出运算符 $+$ 和 $*$ 的结合性和优先级：

$$E \to E + E \mid E * E \mid (E) \mid \mathbf{id} \tag{3.15}$$

无二义的文法

$$\begin{aligned} E &\to E + E \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid \mathbf{id} \end{aligned} \tag{3.16}$$

产生同样的语言，但它给 $+$ 以较低的优先级，并让两个运算符都具有左结合性。使用文法（3.15）有优于使用文法（3.16）的地方。首先，运算符 $+$ 和 $*$ 的结合性和优先级可以方便地改变而无须修改文法（3.15），也不会改变解析器的状态数。其次，文法（3.16）的解析器要花一部分时间来完成产生式 $E \to T$ 和 $T \to F$ 的归约，而文法（3.15）的解析器不会消耗时间在归约这样的**单非产生式**（右部只有一个非终结符产生式）上。

文法（3.15）用 $E' \to E$ 拓广后的 LR(0) 项目集如图 3.33 所示。因为文法（3.15）二义，所以从这些项目集产生 LR 分析表时，肯定会出现动作冲突，冲突出现在项目集 I_7 和 I_8 对应的状态。假如用 SLR 方法构造动作表，I_7 引起的冲突是在面临 $+$ 和 $*$ 时，移进该符号还是按 $E \to E + E$ 归约，因为 $+$ 和 $*$ 都在 FOLLOW(E) 中。I_8 引起的冲突是在面临 $+$ 和 $*$ 时，移进该符号还是按 $E \to E * E$ 归约。事实上，用任何一种 LR 分析表的构造方法都会产生这些冲突。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第130页 文件编号: 422624 页面编号: 31304538445 打印人: 8685 2026-03-09 10:23:00

这些冲突可以用有关 + 和 * 的优先级和结合性的信息来解决。考虑输入 **id**+**id***id，基于图 3.33 的解析器在扫描过 **id**+**id** 后进入状态 7，形成如下格局：

栈	符号	输入
0 1 4 7	**id**+**id**(归约成了$E+E$)	*id$

I_0: $E' \to \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot$ **id**

I_1: $E' \to E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_2: $E' \to (\cdot E)$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot$ **id**

I_3: $E \to$ **id** $\cdot$

I_4: $E \to E + \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot$ **id**

I_5: $E \to E * \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot$ **id**

I_6: $E \to (E \cdot)$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_7: $E \to E + E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_8: $E \to E * E \cdot$
$E \to E \cdot * E$
$E \to E \cdot * E$

I_9: $E \to (E) \cdot$

图 3.33 文法（3.15）拓广后的 LR(0) 项目集

如果 * 的优先级高于 +，解析器应把 * 移进栈，准备归约 * 和它两侧的 **id** 到一个表达式。这正是基于该语言无二义文法的 SLR 解析器（表 3.9）要做的。另外，如果 + 的优先级高于 *, 那么解析器应该归约。这样，根据 + 和 * 的优先关系，可以解决在状态 7 面临 * 时，移进 * 和按 $E \to E + E$ 归约之间的冲突。

如果输入是 **id**+**id**+**id**，解析器扫描完 **id**+**id** 后到达的格局与上面的唯一区别是，下一个输入符号是 + 而不是 *。在状态 7 面临 + 时仍有移进–归约冲突，现在由运算符 + 的结合性来解决冲突。如果 + 是左结合，正确的动作是按 $E \to E + E$ 归约，即第一个 + 及其前后的 **id** 应看成一组。这个选择和例 3.26 文法的 SLR 解析器的动作是一致的。

总之，假如 + 是左结合的，那么在状态 7 面临 + 时应该按 $E \to E + E$ 归约；如果 * 的优先级高于 +，那么在状态 7 面临 * 时应该移进。类似地，可以讨论状态 8，最后得出：如果 * 是左结合且优先级高于 +，那么不论面临 + 还是 *，解析器在状态 8 的动作都按 $E \to E * E$ 归约。

按这种方式处理，可以得到表 3.15 的 LR 分析表，产生式为

(1) $E \to E + E$
(2) $E \to E * E$
(3) $E \to (E)$
(4) $E \to$ **id**

有趣的是，类似的动作表可以从表 3.9 的 SLR 分析表中删去文法（3.16）的单非产生式

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第131页 文件编号: 422624 页面编号: 31314538443 打印人: 8685 2026-03-09 10:23:00

$E \to T$ 和 $T \to F$ 的归约来得到。在构造 LALR 分析表和规范的 LR 分析表时，也可用类似的方法来处理（3.15）这样的二义文法。

表 3.15 文法（3.15）的分析表

状态	动作						转移
	id	+	*	(	)	$	E
0	$s3$			$s2$			1
1		$s4$	$s5$			acc	
2	$s3$			$s2$			6
3		$r4$	$r4$		$r4$	$r4$	
4	$s3$			$s2$			7
5	$s3$			$s2$			8
6		$s4$	$s5$		$s9$		
7		$r1$	$s5$		$r1$	$r1$	
8		$r2$	$r2$		r^2	r^2	
9		$r3$	$r3$		$r3$	$r3$	

3.7.2 使用其他约定来解决冲突

考虑条件语句文法

$$\begin{aligned} \text{stmt} \quad &\to \quad \textbf{if}\ \text{expr}\ \textbf{then}\ \text{stmt}\ \textbf{else}\ \text{stmt} \\ &\quad\ \mid \quad \textbf{if}\ \text{expr}\ \textbf{then}\ \text{stmt} \\ &\quad\ \mid \quad \textbf{other} \end{aligned}$$

该文法是二义的，因为它没有解决悬空 **else** 的二义性。可以肯定，构造 LR 分析表时，会碰到移进–归约冲突，即当 **if** expr **then** stmt 在栈顶，并且 **else** 是下一个输入符号时，究竟是将 **if** expr **then** stmt 归约还是将 **else** 移进。根据语言关于 **else** 配对的约定可以知道，对于这种移进–归约冲突，应该忽略归约，采用移进，即优先移进。

如果需要引入额外的产生式来表示其他产生式所生成的语法构造的一种特殊情况时，这种新增的产生式可能会导致文法产生二义性，从而引起分析动作的冲突。下面先举一个这种文法的例子。

历史上，数学公式编排预处理器 EQN 中使用了特殊情况产生式，这是一个有趣的应用。在 EQN 中，描述数学表达式的文法用运算符 **sub** 表示下标并用运算符 **sup** 表示上标，这个文法的片段如下：

$$\begin{aligned} &(1)\ E \to E\ \textbf{sub}\ E\ \textbf{sup}\ E \qquad &&(4)\ E \to \{E\} \\ &(2)\ E \to E\ \textbf{sub}\ E \qquad &&(5)\ E \to c \\ &(3)\ E \to E\ \textbf{sup}\ E \end{aligned} \tag{3.17}$$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第132页 文件编号: 422624 页面编号: 31324538441 打印人: 8685 2026-03-09 10:23:00

其中的花括号在 EQN 中表示复合表达式，c 表示任意正文串的记号。

文法（3.17）是二义的。该文法没有说明运算符 **sub** 和 **sup** 的结合性和优先级。即使由它们引起的二义性解决了，如规定这两个运算符的优先级相同且都是右结合的，该文法仍然存在二义的。这是因为产生式（1）分离出了由产生式（2）和产生式（3）产生的表达式的一种特例，即形式为 E **sub** E **sup** E 的表达式。没有产生式（1），该文法产生的语言是一样的。把这种形式的表达式处理为一种特殊情况的理由是，像 a **sub** i **sup** 2 这样的表达式应该排版成 a_i^2，而不是 $a_i\ ^2$ 或 $a^2\ _i$ 的形式。只有加上特殊情况产生式后，EQN 才能够产生这样特殊的输出。

构造该文法的 LR 分析表时会发现，存在移进-归约冲突和归约-归约冲突。其中，移进–归约冲突可以根据 **sub** 和 **sup** 的优先级和结合性来解决。归约–归约冲突在产生式

$$E \to E \textbf{ sub } E \textbf{ sup } E$$
$$E \to E \textbf{ sub } E$$

之间，即当 E **sub** E **sup** E 出现在栈顶时，是按前一个产生式将 E **sub** E **sup** E 归约，还是按后一个产生式将 E **sub** E 归约。显然，应该优先特殊情况产生式，即按前一个产生式归约。这样，与该特殊情况产生式相关联的语义动作才可能用更专门的措施来输出特定的格式。

编写一个无二义文法，提取出语法构造的特殊情况，这是非常困难的。为了体会其中的难点，请读者为文法（3.17）构造等价的无二义文法，它的分离形式为 E **sub** E **sup** E 的表达式。

3.7.3 LR 解析的错误恢复

LR 解析器在访问动作表时，若遇到空白条目，表明发现了错误。但是在访问转移表时，它绝不会遇到空白条目。只要已扫描的输入出现一个不正确的后继符号，LR 解析器就会立即报告错误，而不会把不正确的后继符号移进栈。规范的 LR 解析器在报告错误之前绝不会执行任何无效的归约，但 SLR 和 LALR 解析器在报告错误之前有可能执行几步无效的归约。

在 LR 解析中，可按如下方法实现紧急方式的错误恢复：从栈顶开始退栈，直至出现状态 s, 它有预先确定的非终结符 A 的转移；然后抛弃若干个（可以是零个）输入符号，直至找到符号 a, 它能合法地跟随 A。解析器再把 A 和状态 goto$[s, A]$ 压进栈，恢复正常解析。A 的选择可能不唯一，一般来说 A 应是代表主要语法构造的非终结符，如表达式、语句或程序块。例如，如果 A 是非终结符 stmt，那么 a 可以是分号或}，后者标记一个语句序列的结束。

这种错误恢复方法的实质是试图忽略含有语法错误的短语。解析器认为由 A 推出的串含有错误，这个串的一部分已经处理，而其余部分仍在剩余输入中。解析器将试图跳过这个串的其余部分，寻找一个能够合法跟随 A 的后继符号。通过从栈中弹出一些状态、跳过若干输入符号，并把 goto$[s, A]$ 推进栈，解析器当成已发现了 A 的一个实例，从而恢复正常的解析。

错误恢复的另一种方式称为**短语级恢复**。当发现错误时，解析器对剩余输入进行局部纠

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第133页 文件编号: 422624 页面编号: 31334538449 打印人: 8685 2026-03-09 10:23:00

正，它用可以使解析器继续解析的串来代替剩余输入的前缀。典型的局部纠正有：用分号代替逗号，删除多余的分号，或者插入遗漏的分号等。编译器的设计者必须仔细选择替换的串，以免引起死循环。引起死循环是可能的，例如总是在当前输入符号的前面插入一些其他符号。

这种替换可以纠正任何输入串，但主要缺点是很难应对实际错误出现在诊断点之前的情况。

对 LR 解析来说，短语级恢复的实现过程是，考察 LR 分析表的每一个空白条目并根据语言的使用情况，决定最可能进入该条目的输入错误，然后为该条目编一个适当的错误恢复过程。

可以在分析表的动作区域的每个空白条目中填入一个指示器，指向编译器设计者为之设计的错误处理例程。该例程可以执行插入、删除或置换栈顶状态或输入符号的操作。要注意确保这些操作不会导致 LR 解析器陷入无限循环。安全的策略是确保至少删除一个输入符号或最终将其移进，或者在到达输入末尾时确保栈会缩短。应该避免从栈中弹出对应非终结符的状态，因为这会破坏已经成功解析的部分。

【例 3.39】 考虑表达式文法

$$E \to E + E \mid E * E \mid (E) \mid \mathbf{id}$$

表 3.16 给出了该文法的 LR 分析表，它是在表 3.15 的基础上加了错误诊断和恢复。在有的状态（如状态 3、7、8、9）下对某些输入符号的动作由空白改成按某个产生式进行归约，这种修改会推迟错误的发现，多执行了一步或几步归约，但错误仍能在移进下一个符号前被捕获。表 3.15 的其余空白条目已经改成了对错误处理例程（$e1 \sim e4$ 之一）的调用。

表 3.16 有错误处理例程的 LR 分析表

状态	动作						转移
	id	+	*	(	)	$	E
0	$s3$	$e1$	$e1$	$s2$	$e2$	$e1$	1
1	$e3$	$s4$	$s5$	$e3$	$e2$	acc	
2	$s3$	el	el	$s2$	$e2$	el	6
3	$r4$	$r4$	$r4$	$r4$	$r4$	$r4$	
4	$s3$	$e1$	el	$s2$	$e2$	$e1$	7
5	$s3$	el	el	$s2$	$e2$	el	8
6	$e3$	$s4$	$s5$	$e3$	$s9$	$e4$	
7	$r1$	rl	$s5$	$r1$	$r1$	$r1$	
8	r^2	$r2$	r^2	$r2$	r^2	$r2$	
9	$r3$	$r3$	$v3$	$r3$	$r3$	$r3$	

错误处理例程如下：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第134页 文件编号: 422624 页面编号: 31344538447 打印人: 8685 2026-03-09 10:23:00

*e*1：/* 解析器处于状态 0、2、4 或 5 时，期望输入符号为运算对象的首终结符，即 **id** 或左括号，现在遇到的是 +，* 或 $，调用此例程。*/

把状态 3 压进栈（状态 0、2、4 和 5 面临 **id** 时的转移）；给出诊断信息“缺少运算对象”。

*e*2：/* 解析器处于状态 0、1、2、4 或 5，遇到右括号时调用此例程。*/

从输入中删除右括号；给出诊断信息“不配对的右括号”。

*e*3：/* 解析器处于状态 1 或 6，期望运算符，但遇到的却是 **id** 或左括号，调用此例程。*/

把状态 4 压进栈（对应符号 +）；给出诊断信息“缺少运算符”。

*e*4：/* 解析器处于状态 6，遇到 $ 时，调用此例程。*/

把状态 9 压入栈（对应右括号）；给出诊断信息“缺少右括号”。

读者可以用有语法错误的简短表达式，如 **id**+)，体会该错误恢复方法的实际效果。

3.8 LR 解析器的生成器 Yacc

本节介绍 LALR 解析器的生成器 Yacc（yet another compiler compiler）。Yacc 最早由斯蒂芬·C·约翰逊（Stephen C. Johnson）于 1970 年代在贝尔实验室开发，称为 AT&T Yacc。随后出现了许多变种，如 Berkeley 的 Yacc(简称 BYacc) 和 GNU 的 Bison 等，这些工具能够从文法规则自动生成语法分析器的 C 代码。CUP（constractor of useful parsers）是一个类似工具，用于生成语法分析器的 Java 代码。本节以 Linux 操作系统为平台介绍 Yacc 及其变种 Bison。Bison 能将带注释的上下文无关文法转换为使用 LALR(1) 分析表的确定性 LR 或广义 LR（GLR）解析器。Bison 向前兼容 Yacc：所有正确编写的 Yacc 规范（文法描述）都可以用 Bison 处理，无须任何更改。大多数 Linux 和 BSD 系统都自带 Bison 作为系统的基础部分。

3.8.1 使用 Yacc 生成 LR 解析器

图 3.34 展示了用 Yacc 建立翻译器的过程。

首先，用 Yacc 语言编写翻译器的规范并保存在一个文件（如 translate.y）中。Linux 操作系统下执行命令（yacc 实际可以是 bison -d -y）：

```
yacc translate.y
```

把文件 translate.y 翻译为 C 语言文件 y.tab.c。bison 的选项-y 指示模拟 Yacc 的行为，-d 选项生成对应的头文件 y.tab.h。生成的 y.tab.c 包含用 C 语言编写的 LALR 解析器和用户自定义的 C 语言函数。为了减少 LALR 分析表的空间占用，使用紧凑技术来压缩分析表的大小。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第135页 文件编号: 422624 页面编号: 31354538444 打印人: 8685 2026-03-09 10:23:00

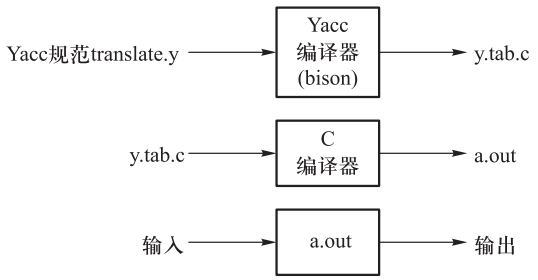


图 3.34 用 Yacc 建立翻译器的过程

然后，用 C 编译器命令 (cc 实际可以是 gcc 或 clang)

cc y.tab.c -ly

编译 y.tab.c，其中的选项-ly 表示链接名为 liby.so 或 liby.a 的 LR 解析器库，具体名称依赖于系统配置。Bison 生成的解析器不需要额外的链接库选项，因为它自身包含 LR 解析驱动程序。编译结果默认是目标程序 a.out，该程序能完成 Yacc 规范（如 translate.y）所指定的翻译。如果还需要调用其他过程，那么可以让它们和 y.tab.c 一起编译和链接。

用 Yacc 编写的规范由三部分组成：

```
声明
%%
翻译规则
%%
用C语言编写的支持例程
```

【例 3.40】 以构造一个简单的计算器为例，说明怎样准备 Yacc 规范。该计算器读取一个算术表达式，计算并打印它的值。构造该计算器从下面算术表达式的文法开始：

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid \textbf{digit}$$

记号 **digit** 是 0~9 之间的单个数字。基于这个文法的计算器的 Yacc 规范见图 3.35。

Yacc 规范的声明部分（第一部分）有两节，它们都是可选的。第一节处于分界符%{和%}之间，它是一些普通的 C 语言的声明，这里声明的常量和变量等提供给第二部分和第三部分的翻译规则和过程使用。在图 3.35 中，声明部分的第一节只有一个包含语句

include <ctype.h>

因为这个文件含有对判断函数 isdigit 的声明。

声明部分的第二节是文法终结符（词法记号）的声明，在图 3.35 中的语句

%token DIGIT

其中，DIGIT 是记号。声明的记号可用于 Yacc 规范的第二部分和第三部分。如果词法分析

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第136页 文件编号: 422624 页面编号: 31364538442 打印人: 8685 2026-03-09 10:23:00

器是由 Lex 建立的，这些记号声明对该词法分析器也是可用的。

```
%{
#include <ctype.h>
%}
%token DIGIT
%%
line    : expr '\n'       {printf ("%d\n", $1 );}
            ;
expr    : expr '+' term     {$$ = $1 + $3;}
            | term
            ;
term    : term '*' factor   {$$ = $1 * $3;}
            | factor
            ;
factor  : '(' expr ')'      {$$ = $2;}
            | DIGIT
            ;
%%
yylex ( ) {
    int c;
    c = getchar ( );
    if (isdigit (c) ) {
        yylval = c - '0';
        return DIGIT;
    }
    return c;
}
```

图 3.35　简单计算器的 Yacc 规范

Yacc 规范的第二部分位于第一个%%后面，该部分放置翻译规则，每条规则由一个文法产生式及与它相联系的语义动作组成。产生式集合

规则名: 选择1 | ··· | 选择n

在 Yacc 中写成

```
规则名      : 选择1      {语义动作1}
            | 选择2      {语义动作2}
            ...
            | 选择n      语义动作n
            ;
```

在 Yacc 产生式中，没有加引号的字母数字串，若没有被声明为记号，则被看成非终结符；加引号的单个字符，如'+'，被看成是由这个 + 字符代表的记号。右部的各个选择及其语义动作之间用竖线隔开，最后一个选择的后面用分号，表示该产生式集合的结束。第一个规则的左部非终结符是开始符号。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第137页 文件编号: 422624 页面编号: 31374538440 打印人: 8685 2026-03-09 10:23:00

Yacc 的语义动作是 C 语句序列。在语义动作中，符号 $$ 表示引用左部非终结符的属性值，而 $$i$$ 表示引用右部第 i 个文法符号的属性值。每当归约一个产生式时，执行与之关联的语义动作，所以语义动作一般是从各 $$i$$ 的值决定 $$ 的值。在图 3.35 的 Yacc 规范中，两个 E 产生式

$$E \to E + T \mid T$$

及与它们相关的语义动作写成

```
expr    : expr '+' term   {$$ = $1 + $3;}
        | term
        ;
```

注意，在第一个产生式中，非终结符 term 是右部的第三个文法符号，'+'是第二个文法符号。第一个产生式的语义动作是把右部 expr 的值和 term 的值相加，把结果赋给左部非终结符 expr，作为它的值。第二个产生式的语义动作的描述被省略，因为当右部只有一个文法符号时，默认的语义动作就是 { $$ = $1; }，这正是所希望的动作。

注意，一个新的开始产生式

```
line    : expr '\n'    {printf ("%d\n", $1);}
```

被加到这个 Yacc 规范中。该产生式的含义是，这个计算器的输入是一个表达式后面跟一个换行字符。它的语义动作是打印表达式的十进制值且换行。

Yacc 规范的第三部分是一些 C 语言编写的支持例程。必须提供名字为 yylex( ) 的词法分析器（当然也可以用 Lex 来产生 yylex( )），其他的过程也可以添加上，如错误恢复例程。

词法分析器 yylex( ) 返回记号，它由记号名和属性值两部分组成。如果返回的是 DIGIT 这样的记号名，该名字还必须声明在 Yacc 规范第一部分的第一节，给它一个区分于其他记号的值。属性值是通过 Yacc 定义的变量 yylval 传给解析器的。

图 3.35 的词法分析器是非常粗糙的。它用 C 语言的函数 getchar ( ) 每次读取一个输入字符，如果是数字字符，取它的值存入变量 yylval，返回记号 DIGIT，否则把字符本身作为记号返回。

3.8.2 用 Yacc 处理二义文法

修改 3.8.1 节的 Yacc 程序，使计算器功能更强。首先，让计算器计算一系列表达式，每行一个，也允许表达式之间有空白行。为此，将第一个翻译规则改为

```
lines: lines expr 'n'   { printf( "%g\n", $2 ); }
    |  lines  '\n'
    |  /* empty */
    ;
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第138页 文件编号: 422624 页面编号: 31384538448 打印人: 8685 2026-03-09 10:23:00

按照 Yacc 的规定，第三行的空选择表示 ε。

其次，允许表达式使用由多个数字组成的数，并增加运算符包括 +、− (一元和二元)、* 和/。使用二义文法

$$E \to E + E \mid E - E \mid E * E \mid E/E \mid (E) \mid -E \mid \textbf{number}$$

来描述表达式。最终的 Yacc 程序如图 3.36 所示。

```
%{
# include <ctype.h>
# include <stdio.h>
# define YYSTYPE double /* 将 Yacc 栈定义为 double 类型 */
%}

%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
lines : lines expr '\n' {printf ( "%g \n", $2 );}
        | lines '\n'
        | /* empty */
        ;
expr : expr '+' expr {$$ = $1 + $3;}
        | expr '-' expr {$$ = $1 - $3;}
        | expr '*' expr {$$ = $1 * $3;}
        | expr '/' expr {$$ = $1 / $3;}
        | '(' expr ')' {$$ = $2;}
        | '-' expr %prec UMINUS {$$ = -$2;}
        | NUMBER
        ;
%%
yylex ( ) {
    int c;
    while ( ( c = getchar ( ) ) = = ' ' );
    if ( ( c = = ' . ' ) || (isdigit (c) ) ) {
        ungetc (c, stdin);
        scanf ("%lf", &yylval);
        return NUMBER;
    }
    return c;
}
```

图 3.36　更高级的计算器的 Yacc 规范

因为图 3.36 的 Yacc 程序的文法是二义的，LALR 算法将产生分析动作的冲突。Yacc 会报告产生了多少个分析动作的冲突。可以用编译选项-v 来获得产生的项目集（仅含核心项目）和分析动作冲突的描述，该选项将这些信息放到另一个文件 y.output 中，该文件还包含了 LR

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第139页 文件编号: 422624 页面编号: 31394538446 打印人: 8685 2026-03-09 10:23:00

分析表的可读表示，它表明了 Yacc 是怎样解决这些分析动作冲突的。

当 Yacc 报告发现分析动作冲突时，明智的做法是建立和查阅文件 y.output，以明白为什么会出现分析动作的冲突以及它们是否已经得到恰当的解决。

除非另有说明，否则 Yacc 按下面两条规则来解决分析动作的冲突：

（1）对于归约–归约冲突，选择在 Yacc 规范中最先出现的那个冲突产生式。按此规则，为了正确解决排版文法（3.17）的冲突，只要把产生式（1）放在产生式（3）的前面就足够了。

（2）对于移进–归约冲突，优先移进。这条规则正确解决了悬空 else 的移进–归约冲突。

因为这些默认的规则并不总是编译器设计者所想要的，所以 Yacc 允许编译器设计者提供一些解决移进–归约冲突的说明。在声明部分，可以为终结符指定优先级和结合性。声明

```
%left  '+' '-'
```

表示 + 和 − 有同样的优先级且它们是左结合的。声明

```
%right  '^'
```

表示运算符^是右结合的。还可以用声明来限制二元运算符为不可结合的（不允许该运算符相邻出现），如

```
%nonassoc  '<'
```

终结符的优先级按它们在声明部分出现的次序而定，先出现的优先级低，同一声明中的终结符有相同的优先级。这样，图 3.36 的声明

```
%right UMINUS
```

使得 UMINUS 的优先级高于前面 5 个终结符。

Yacc 解决移进-归约冲突时，首先参考这个冲突涉及的产生式和终结符的优先级和结合性。通常，产生式的优先级和结合性同它最右边的终结符一致，在大多数情况下这是合理的决策。

如果 Yacc 必须在移进输入符号 a 和按产生式 $A \to \alpha$ 归约这两个动作之间进行选择的话，那么当这个产生式的优先级高于 a，或者优先级相同但产生式左结合时，选择归约动作，否则选择移进。例如，给定产生式

$$E \to E + E \mid E * E$$

若下一个输入符号是 +，归约项目的产生式是 $E \to E + E$，那么优先做归约，因为该产生式右部的 + 和下一个输入符号有同样的优先级，而 + 是左结合的。如果下一个输入符号为 *，那么选择移进，因为 * 的优先级高于该产生式右部 + 的优先级。

在那些最右边的终结符不能给产生式以适当优先级和结合性的情况下，可以增加一个标签

```
%prec 终结符
```

来强制产生式，使得它的优先级和结合性与该标签终结符的一样。这个终结符仅仅是一个占位符，就像图 3.36 的 UMINUS 那样，它不由词法分析器返回，仅通过声明来决定一个产生式的优先级和结合性。在图 3.36 中，声明

```
%right UMINUS
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第140页 文件编号: 422624 页面编号: 31404538444 打印人: 8685 2026-03-09 10:23:00

给记号 UMINUS 指定高于 * 和/的优先级。在翻译规则部分，标签

```
%prec UMINUS
```

出现在产生式

```
expr  :  '-' expr
```

的后面，它使得该产生式的一元运算符–的优先级高于任何其他运算符。

Yacc 不报告用上述优先级和结合性能解决的移进–归约冲突。

3.8.3 Yacc 的错误恢复

Yacc 使用一种称为**错误产生式**的形式来进行错误恢复。首先，它要求编译器设计者决定为哪些“主要的”非终结符设置与它们相关联的错误恢复处理，这些非终结符的典型选择是那些用于产生表达式、语句、程序块和过程的非终结符。然后编译器设计者把形式为 $A \to$ **error** α 的错误产生式加到文法上，其中 A 是主要非终结符，α 是文法符号串，可以是空串，**error** 是 Yacc 的保留字。Yacc 将从这样的规范产生解析器，把错误产生式当作普通产生式来处理。

当 Yacc 产生的解析器遇到错误时，它采用特殊的方式来处理其项目集含错误产生式的状态。当遇到错误时，解析器从状态栈中逐个弹出状态，直到发现栈顶状态的项目集包含形式为 $A \to \cdot$ **error** α 的项目为止。然后解析器把虚构的记号 **error** 移进栈，好像它在输入中看见了这个记号。

当 α 为 ε 时，解析器立即进行对 A 的归约并执行产生式 $A \to$ **error** 的语义动作（它可能是编译器设计者指定的错误恢复例程），然后它抛弃若干输入符号，直到发现一个能回到正常处理的输入符号为止。

如果 α 非空，解析器就在输入上向前寻找能够归约为 α 的子串。如果 α 中所含的全是终结符，那么它在输入上直接寻找这样的串，把其中的符号移进栈，使得 **error** α 在解析器的栈顶。随后解析器把 **error** α 归约为 A 并执行产生式 $A \to$ **error** 的语义动作，恢复正常分析。

3.8.2 小节介绍的短语级错误恢复方法显然较难用于 Yacc 自动生成解析器的情况。使用 Yacc 的编译器设计者通常用紧急方式的错误恢复思想。例如，错误产生式

stmt → **error** ;

要求解析器看见错误时跳过下一个分号，好像这个语句已经被看见一样。

【例 3.41】 图 3.37 所示的 Yacc 程序在图 3.36 的基础上增加了错误产生式。错误产生式为

lines : **error** '\n'

当输入行有语法错误时，解析器从栈中弹出状态，直至碰到一个有移进 **error** 动作的状态。状态 0 是唯一符合条件的这种状态，因为它的项目集包含

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第141页 文件编号: 422624 页面编号: 31414538442 打印人: 8685 2026-03-09 10:23:00

lines → · **error** '\n'

状态 0 总是在栈底。解析器把终结符 **error** 移进栈，抛弃剩余输入符号，直至发现换行符。然后解析器把换行符移进栈，**error** '\n'归约成 lines, 输出诊断信息“重新输入上一行”。专门的 Yacc 例程 yyerrok 用来重置解析器，回到正常操作方式。

```
%{
# include <ctype.h>
# include <stdio.h>
# define YYSTYPE double /* 将 Yacc 栈定义为 double 类型 */
%}

%token NUMBER
%left '+' '-'
%left '*' '/ '
%right UMINUS
%%
lines : lines expr '\n' {printf ( "%g \n", $2 );}
        | lines '\n'
        | /* empty */
        | error '\n' {yyerror( "reenter previous line:"); yyerrok;}
        ;
expr : expr '+' expr {$$ = $1 + $3;}
        | expr '-' expr {$$ = $1 - $3;}
        | expr '*' expr {$$ = $1 * $3;}
        | expr '/' expr {$$ = $1 / $3;}
        | '(' expr ')' {$$ = $2;}
        | '-' expr %prec UMINUS {$$ = -$2;}
        | NUMBER
        ;
%%
yylex ( ) {
    int c;
    while ( ( c = getchar ( ) ) = = ' ' );
    if ( ( c = = ' . ' ) || (isdigit (c) ) ) {
        ungetc (c, stdin);
        scanf ("%lf", &yylval);
        return NUMBER;
    }
    return c;
}
```

图 3.37 有错误恢复的计算器

习 题

3.1 考虑文法 (其中括号、逗号和 a 均为终结符)

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第142页 文件编号: 422624 页面编号: 31424538440 打印人: 8685 2026-03-09 10:23:00

$S \to (L) \mid a$

$L \to L, S \mid S$

(a) 建立句子 $(a,(a,a))$ 和 $(a,((a,a),(a,a)))$ 的解析树。

(b) 为 (a) 的两个句子构造最左推导。

(c) 为 (a) 的两个句子构造最右推导。

(d) 这个文法产生的语言是什么？

3.2 考虑文法（$V_T = \{a, b\}$）

$S \to aSbS \mid bSaS \mid \varepsilon$

(a) 为句子 $abab$ 构造两个不同的最左推导，以此说明该文法是二义的。

(b) 为 $abab$ 构造对应的最右推导。

(c) 为 $abab$ 构造对应的解析树。

(d) 这个文法产生的语言是什么？

3.3 下面的二义文法描述命题演算公式，$V_T = \{\textbf{and}, \textbf{or}, \textbf{not}, \textbf{true}, \textbf{false}, (,)\}$，为它编写一个等价的非二义文法。

$S \to S$ **and** $S \mid S$ **or** $S \mid$ **not** $S \mid$ **true** $\mid$ **false** $\mid (S)$

3.4 文法

$R \to R\,\texttt{'|'}\,R \mid RR \mid R\,\texttt{'*'} \mid (R) \mid a \mid b$

产生字母表 $\{a, b\}$ 上所有不含 ε 的正规式。注意，第一条竖线加了引号，它是正规式的或运算符号，而不是文法产生式右部各选择之间的分隔符，另外 '*' 是一个普通的终结符。该文法是二义的。

(a) 证明该文法产生字母表 $\{a, b\}$ 上的所有正规式。

(b) 为该文法编写一个等价的非二义文法。它给予运算符 *、连接和 | 的优先级和结合性与2.2节中定义的一致。

(c) 按上面两个文法构造句子 $ab \mid b^*a$ 的解析树。

3.5 下面的条件语句文法

stmt → **if** expr **then** stmt | matched_stmt

matched_stmt → **if** expr **then** matched_stmt **else** stmt | **other**

试图消除悬空 else 的二义性，请证明该文法仍然存在二义。

3.6 为字母表 $\Sigma = \{a, b\}$ 中的下列每个语言设计一个文法，其中哪些语言是正规的？

(a) 每个 a 后面至少有一个 b 跟随的所有串。

(b) a 和 b 的个数相等的所有串。

(c) a 和 b 的个数不相等的所有串。

(d) 不含 abb 作为子串的所有串。

*(e) 形式为 xy 且 $x \neq y$ 的所有串。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第143页 文件编号: 422624 页面编号: 31434538448 打印人: 8685 2026-03-09 10:23:00

3.7 可以在文法产生式的右部使用类似正规式的运算符。方括号可以用来表示产生式的可选部分，例如可以用

stmt → **if** expr **then** stmt[**else** stmt]

表示 else 子句是可选的。通常，$A \to \alpha[\beta]\gamma$ 等价于两个产生 $A \to \alpha\beta\gamma$ 和 $A \to \alpha\gamma$。

花括号用来表示短语可重复出现若干次（包括零次），例如

stmt → **begin** stmt {; stmt} **end**

表示处于 **begin** 和 **end** 之间的、由分号分隔的语句表。通常，$A \to \alpha\{\beta\}\gamma$ 等价于 $A \to \alpha B\gamma$ 和 $B \to \beta B \mid \varepsilon$。

概念上，$[\beta]$ 代表正规式 $B \mid \varepsilon$，$\{\beta\}$ 代表 β^*，现在把它们推广为允许文法符号的任何正规式出现在产生式的右部。

(a) 修改上面的 *stmt* 产生式，使得每个语句都以分号终止的语句表出现在产生式右部。

(b) 给出上下文无关的产生式，它和 $A \to B^*\ a\ (C \mid D)$ 产生同样的串集。

(c) 说明如何用一组有限的上下文无关产生式来代替产生式 $A \to \gamma$，其中 γ 是正规式。

3.8 (a) 消除习题 3.1 文法的左递归。

(b) 为 (a) 的文法构造预测解析器。

3.9 为习题 3.3 的文法构造预测解析器。

3.10 构造下面文法的 LL(1) 分析表，其中 $V_T = \{\mathbf{int}, \mathbf{real}, \mathbf{id}, \texttt{','}\}$。

$D \to TL$

$T \to \mathbf{int} \mid \mathbf{real}$

$L \to \mathbf{id}\ R$

$R \to \texttt{','}\ \mathbf{id}\ R \mid \varepsilon$

3.11 构造下面文法的 LL(1) 分析表。

$S \to aBS \mid bAS \mid \varepsilon$

$A \to bAA \mid a$

$B \to aBB \mid b$

3.12 下面的文法是否为 LL(1) 文法？说明理由。

$S \to AB \mid PQx$　　$A \to xy$　　$B \to bc$

$P \to dP \mid \varepsilon$　　$Q \to aQ \mid \varepsilon$

***3.13** 证明左递归的文法不是 LL(1) 文法。

***3.14** 证明 LL(1) 文法不是二义的。

3.15 证明没有 ε 产生式的文法，只要每个非终结符的各个选择以不同的终结符开始，那么它就是 LL(1) 的。

3.16 (a) 用习题 3.1 的文法构造 $(a,(a,a))$ 的最右推导，说出每个右句型的句柄。

(b) 给出对应 (a) 的最右推导的移进–归约解析器的步骤。

(c) 对照 (b) 的移进–归约，给出自下而上构造解析树的步骤。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第144页 文件编号: 422624 页面编号: 31444538446 打印人: 8685 2026-03-09 10:23:01

3.17 给出接受文法 (其中括号、逗号和 a 均为终结符)

$S \to (L) \mid a \qquad L \to L, S \mid S$

的活前缀的一个 DFA。

3.18 为习题 3.3 的文法构造 SLR 分析表。

3.19 考虑下面的文法 (其中 $V_T = \{+, ^*, a, b\}$)

$E \to E + T \mid T$

$T \to TF \mid F$

$F \to F^* \mid a \mid b$

(a) 为此文法构造 SLR 分析表。

(b) 构造 LALR 分析表。

3.20 证明下面的文法 (其中 $V_T = \{a\}$)

$S \to SA \mid A$

$A \to a$

是 SLR(1) 文法，但不是 LL(1) 文法。

3.21 (a) 证明下面文法 (其中 $V_T = \{a, b\}$)

$S \to AaAb \mid BbBa$

$A \to \varepsilon$

$B \to \varepsilon$

是 LL(1) 文法，但不是 SLR(1) 文法。

*(b) 证明所有 LL(1) 文法都是 LR(1) 文法。

3.22 证明下面文法 (其中 $V_T = \{a, b, c, d\}$)

$S \to Aa \mid bAc \mid dc \mid bda$

$A \to d$

是 LALR(1) 文法，但不是 SLR(1) 文法。

3.23 说明每个 SLR(1) 文法都是 LALR(1) 文法。

3.24 证明下面文法 (其中 $V_T = \{a, b, c, d\}$)

$S \to Aa \mid bAc \mid Bc \mid bBa$

$A \to d$

$B \to d$

是 LR(1) 文法，但不是 LALR(1) 文法。

3.25 一个非 LR(1) 的文法 (其中 $V_T = \{a, b\}$) 如下:

$L \to MLb \mid a$

$M \to \varepsilon$

请给出所有含移进–归约冲突的规范 LR(1) 项目集，以说明该文法确实不是 LR(1) 的。

3.26 假设 $V_T = \{+, \mathbf{id}\}$,

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第145页 文件编号: 422624 页面编号: 31454538443 打印人: 8685 2026-03-09 10:23:01

(a) 通过构造识别活前缀的 DFA 和构造分析表，来证明文法 $E \to E+$ **id** | **id** 是 SLR(1) 文法。

(b) 下面左右两个文法都和 (a) 的文法等价：

$E \to E + M$ **id** | **id**　　$E \to ME +$ **id** | **id**

$M \to \varepsilon$　　$M \to \varepsilon$

请指出其中是否有不是 LR(1) 文法的，并给出它们不是 LR(1) 文法的理由。

3.27 对于文法 G，$V_T = \{d, p\}$，产生式如下：

$S \to I \mid R$　　$I \to d \mid Id$　　$R \to WpF$

$W \to Wd \mid \varepsilon$　　$F \to Fd \mid d$

(a) 令 d 表示任意数字，p 表示十进制小数点，那么非终结符 S、I、R、W 和 F 在编程语言中分别表示什么？

(b) 该文法是 LR(1) 文法吗？为什么？

3.28 假设 $V_T = \{$**begin**, semicolon, **end**$\}$, d 和 s 分别代表单个声明和单个语句。下面文法不是 LR(1) 的，对它略做修改，使之成为一个等价的 SLR(1) 文法。

PROGRAM → **begin** DECLIST semicolon STATELIST **end**

DECLIST → d semicolon DECLIST | d

STATELIST → s semicolon STATELIST | s

3.29 （a）为下面文法构造规范 LR(1) 分析表，画出像图 3.29 这样的状态转换图。

$S \to V = E \mid E$

$V \to *E \mid$ **id**

$E \to V$

（b）上述状态转换图有同心项目集吗？若有，合并同心项目集后是否会出现动作冲突？

3.30 描述文法

$S \to aSbS \mid aS \mid \varepsilon$

产生的语言，并为此语言编写一个 LR(1) 文法。

3.31 下面两个文法中哪个不是 LR(1) 文法，并对非 LR(1) 的那个文法给出含有移进–归约冲突的规范的 LR(1) 项目集。

$S \to aAc$　　$S \to aAc$

$A \to Abb \mid b$　　$A \to bAb \mid b$

3.32 现有字母表 $\Sigma = \{a\}$，编写一个与正规式 a^* 等价的上下文无关文法，要求所编写的文法既不是 LR 文法，也不是二义文法。

3.33 为语言

$L = \{a^m b^n \mid 0 \leqslant m \leqslant 2n\}$（$a$ 的个数不超过 b 的个数的两倍）

编写 3 个文法，它们分别是 LR(1) 的、二义的和非二义且非 LR(1) 的。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第146页 文件编号: 422624 页面编号: 31464538441 打印人: 8685 2026-03-09 10:23:01

3.34 为语言

$$L = \{w \mid w \in (a \mid b)^* \text{ 并且在 } w \text{ 的任何前缀中，} a \text{ 的个数不少于 } b \text{ 的个数}\}$$

编写 3 个文法，它们分别是 LR(1) 的、二义的和非二义且非 LR(1) 的。

3.35 为习题 3.4 的文法构造 SLR(1) 分析表，分析动作冲突的解决要保证正规式能以正常的方式解析。

3.36 由于文法二义性引发的 LR(1) 分析动作冲突，可以依据消除二义性的规则而得到相应文法的 LR(1) 分析表，根据此表可以正确识别输入串是否为相应语言的句子。对于非二义非 LR(1) 文法引发的 LR(1) 分析动作的冲突，是否也可以依据一定的规则来消除这种分析动作的冲突而得到 LR(1) 分析表，并且根据此表识别相应语言的句子？若可以，你是否能给出这样的规则？

3.37 下面是一个二义文法

$$S \to AS \mid b$$
$$A \to SA \mid a$$

如果为该文法构造 LR 分析表，则一定存在某些有分析动作冲突的条目，它们是哪些？假定分析表这样来使用：出现冲突时，不确定地选择一个可能的动作。给出对于输入 $abab$ 所有可能的动作序列。

3.38 假设 $V_T = \{\textbf{integer}, \textbf{boolean}, \textbf{array}, \textbf{of}, \textbf{record}, \textbf{id}, \textbf{num}, \uparrow, :, ;\}$，下面是类型表达式的语法：

type $\to$ **integer** | **boolean** | **array** [**num**] **of** type | **record** field_list **end** | $\uparrow$ type

field_list $\to$ **id** : type | **id** : type; field_list

若规定：在记录类型中不能出现数组类型（包括不能出现数组的指针类型）。请重新设计一个文法，把该约束体现在文法中，即它与上述文法的区别就是所定义的语言满足该约束。

***3.39** 为排版文法 (3.17) 构造一个等价的 LR 文法，它能把形式为 E **sub** E **sup** E 的表达式处理为一种特殊情况。

3.40 编写一个 Yacc 程序，它把输入的算术表达式翻译成对应的后缀表达式输出。

3.41 编写一个 ANTLR 程序，它把输入的算术表达式翻译成对应的后缀表达式输出。

3.42 编写一个 Yacc“计算器”程序，它计算布尔表达式。

3.43 编写一个 ANTLR“计算器”程序，它计算布尔表达式。

3.44 编写一个 Yacc 程序，它取正规式作为输入，产生它的解析树作为输出。

3.45 编写一个 ANTLR 程序，它取正规式作为输入，产生它的解析树作为输出。

3.46 对于例 3.17 和例 3.39 的预测解析器和 LR 解析器，追踪它们面临下面有语法错误的输入时的动作。

(a) (**id** + (∗**id**)

(b) ∗ + **id**) + (**id**∗

***3.47** 为下面的文法构造有短语级错误恢复的 LR 解析器。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第147页 文件编号: 422624 页面编号: 31474538449 打印人: 8685 2026-03-09 10:23:01

stmt → **if** e **then** stmt
| **if** e **then** stmt **else** stmt
| **while** e **do** stmt
| **begin** list **end**
| s
list → list; stmt
| stmt

3.48 对于一段 C 语言代码：

```
long gcd(long p, long q) {
    if (p%q == 0)
        return q;
    else
        return gcd(q, p%q);
}
```

基于 LALR(1) 方法的一个编译器的报错情况如下：

如果缺少第 1 行的逗号，编译器扫描第 1 行时报告 “expected '; ', ', 'or ')'before 'long'”。如果缺少第 2 行的右括号，编译器扫描第 3 行时报告 “(expected ') 'before 'return'”。这两个例子表明 LALR(1) 方法能及时发现错误，且不会把出错点后面的符号移进分析栈（活前缀性质）。

如果第 2 行的 if 误写成 fi，编译器扫描第 3 行时报告 “expected'; 'before 'return'”，扫描第 4 行时报告 “'else'without a previous 'if'”。此时是否违反了活前缀性质？

第 4 章

语法制导的翻译

在 3.4 节和 3.8 节给出的 ANTLR 或 Yacc 文法例子中，大家看到了一种有用的描述形式：将语法构造的属性附加在代表语法构造的文法符号上，这些属性值由伴随着文法产生式的语义动作来计算；语义动作会在解析器分析到相应位置时触发，从而产生相应的结果。这种伴随语法分析进行的属性计算称为“语法制导的”（syntax-directed）语言翻译，可以用来描述编译器的语义分析和中间代码生成等。

本章系统地介绍“语法制导的语言翻译”的描述方法及其实现。它的语义动作（有时称为语义规则）可以构建抽象语法树、产生代码、把信息存入符号表、显示出错信息或完成其他工作。本章讨论语义规则和产生式相联系的两种方式：语法制导的定义和语法制导的翻译方案（后者简称翻译方案）。语法制导定义是比较抽象的翻译规范，它描述了文法符号的属性及属性之间的依赖关系，而不涉及具体实现。翻译方案则更注重实现细节，它通过在文法规则中嵌入语义动作，指明了各语义规则的计算时机。在第 5 章规范语义检查时，将使用这两种方法。

本章还讨论语法制导定义和翻译方案的实现方法，包括针对特定输入记号串的**解析树方法**（4.1 节）、针对特定文法的**基于规则的方法**，以及结合解析方法的**忽略规则的方法**（4.2 节—4.4 节）。解析树方法直接构造解析树并显式管理属性依赖，而基于规则和忽略规则的方法则通过预定义的计算策略简化了依赖管理，提升了编译效率。

4.1 语法制导的定义

本节介绍语法制导定义的形式及其概念上的实现模型，即解析树方法。

4.1.1 语法制导定义的形式

语法制导的定义是带属性和语义规则的上下文无关文法，其中的文法称为**基础文法**；每个文法符号有一组属性，每个产生式有一组语义规则。如果 X 是文法符号，a 是它的一个属

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第149页 文件编号: 422624 页面编号: 31494538445 打印人: 8685 2026-03-09 10:23:01

性，则 $X.a$ 指称 X 的属性 a 的值，属性可以表示字符串、数值、类型、表引用等。在产生式 $A \to XX$ 中，为区分两次出现的 X，可用下标表示为 $A \to X_1X_2$，分别代表不同实例的 X。这样，属性 $X_1.a$ 和 $X_2.a$ 可以独立定义和计算，确保语义规则中准确处理每个实例的属性值。

如果解析树的结点通过记录或对象实现，那么 X 的属性 a 就可以用代表 X 的记录或对象中的数据域 a 来实现。解析树中不同的 X 结点有各自的 a 属性。这些属性由结点所应用的产生式的语义规则来定义和计算。

在语法制导定义中，文法符号的属性是通过与产生式关联的语义规则进行定义和计算的。每个文法产生式 $A \to \alpha$ 有一组形式为 $b = f(c_1, c_2, \cdots, c_k)$ 的语义规则，其中 f 是函数，b 和 $c_1, c_2, \cdots, c_k$ 是该产生式的文法符号的属性，该规则定义属性 b，并且：

（1）如果 b 是 A 的属性，$c_1, c_2, \cdots, c_k$ 是产生式右部文法符号的属性或 A 的其他属性，那么 b 称为 A 的**综合属性**（synthesized attribute）。

（2）如果 b 是产生式右部某个文法符号 X 的属性，$c_1, c_2, \cdots, c_k$ 是 A 的属性或右部文法符号的属性，那么 b 称为 X 的**继承属性**（inherited attribute）。

在这两种情况下，都说属性 b 依赖于属性 $c_1, c_2, \cdots, c_k$。每个文法符号的综合属性集和继承属性集的交集应为空。一般来说，一个结点的综合属性的值通过解析树中其子结点的属性值来计算；继承属性的值由结点的兄弟结点、父结点和自己的属性值来计算。终结符只有综合属性，它由词法分析器提供，即记号的属性。因此，在语法制导定义中没有计算终结符的语义规则。

语义规则的函数 f 通常是表达式。在实际使用的语法制导定义中，经常还会出现一些规则的目的是打印值、输出中间代码或修改全局变量等，它们是一些产生副作用的操作。这样的语义规则写成过程调用或程序段。可以把它们想象成定义了产生式左部非终结符的一个匿名综合属性，这个匿名属性和赋值符号 $=$ 在该规则中没有显式表示出来。

属性文法是指语义规则函数无副作用的语法制导定义。本书大部分语法制导定义都使用有副作用的语义规则。

【例 4.1】 表 4.1 所示的语法制导定义表示一个计算器程序。这个定义分别给非终结符 E、T 和 F 一个存放整数值的综合属性 val。对每个 E、T 和 F 产生式，语义规则从产生式右部非终结符的属性值 val 或 **digit** 的 $lexval$ 来计算产生式左部非终结符的属性值 val。

记号 **digit** 有综合属性 $lexval$，它的值由词法分析器提供。在产生式 $L \to E\mathbf{n}$ 中，**n** 是表达式的结束标记（如换行符）。该产生式的语义动作比较特殊，它是打印由 E 产生的算术表达式的值，这被看成定义了 L 的一个匿名属性。这个计算器的 Yacc 规范在图 3.35 中已给出，目的是描述 LR 解析时的翻译。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第150页 文件编号: 422624 页面编号: 31504538443 打印人: 8685 2026-03-09 10:23:01

表 4.1 简单计算器的语法制导定义

产生式	语义规则
$L \to En$	$print(E.val)$
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val \times F.val$
$T \to F$	$T.val = F.val$
$F \to (E)$	$F.val = E.val$
$F \to$ **digit**	$F.val =$ **digit**$.lexval$

4.1.2 综合属性

综合属性在实践中有广泛应用。各文法符号的属性均是综合属性的语法制导定义称为 S **属性定义**。对于 S 属性定义，解析树中各结点属性的计算可以自下而上地完成：从叶结点到根结点，通过计算语义规则而得到结点的属性。

每个结点的属性值都标注出来的解析树称为**注释解析树**（annotated parse tree），计算各结点属性值的过程称为解析树的**注释**或**修饰**。

【例 4.2】 例 4.1 的 S 属性定义规范一种计算器。例如，若输入是表达式 $8+5*2$，并跟随一个换行符 **n** 时，则该计算器打印值为 18。图 4.1是 $8+5*2$**n** 的注释解析树，在树的根结点打印 18。

为了明白属性值是怎么计算的，考虑最左下方的分支结点，它使用的产生式是 $F \to$ **digit**，对应的语义规则是 F.val = **digit**.lexval。从该规则得到此 F 结点的 val 属性值为 8，因为它的子结点 **digit** 的 lexval 是 8。同样地，该结点的父结点的属性 T.val 的值也为 8。

再以产生式 $E \to E+T$ 的结点为例，左部 E 结点的属性 val 由产生式 $E \to E_1 + T$ 的语义规则 E.val = E_1.val + T.val 来定义。当在这个结点运用该语义规则时，子结点 E_1 和 T 的 val 分别为 8 和 10，故在此结点求得 E.val 的值为 18。

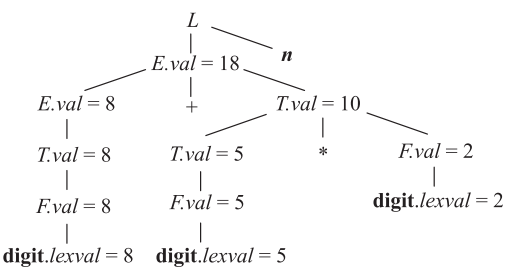


图 4.1 $8+5*2$**n** 的注释解析树

4.1.3 继承属性

在解析树中，一个结点的继承属性是由它的兄弟结点、父结点和自己的属性来定义的。编程语言的一些构造的属性依赖于它们所在的上下文，此时使用继承属性是方便的。在下面的

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第151页 文件编号: 422624 页面编号: 31514538441 打印人: 8685 2026-03-09 10:23:01

例子中，继承属性将类型信息传递给一张声明表中的各个标识符。

【例 4.3】 在表 4.2 所示的语法制导定义中，非终结符 D 产生的声明由关键字 **int** 或 **real** 及一张标识符列表组成。非终结符 T 有综合属性 type，它的值由声明中的关键字决定。产生式 $D \to TL$ 的语义规则置 L 的继承属性为声明中的类型。产生式 $L \to L_1, \mathbf{id}$ 的语义规则 $L_1.\text{in} = L.\text{in}$ 把继承属性 in 沿解析树向下传递类型。L 产生式的规则调用过程 addType，把类型信息加到符号表中各个标识符的条目（属性 entry 给出条目的入口）中。

表 4.2 有继承属性的语法制导定义

产生式	语义规则
$D \to TL$	$L.\text{in} = T.\text{type}$
$T \to \mathbf{int}$	$T.\text{type} = \text{integer}$
$T \to \mathbf{real}$	$T.\text{type} = \text{real}$
$L \to L_1, \mathbf{id}$	$L_1.\text{in} = L.\text{in}$ addType($\mathbf{id}$.entry, L.in)
$L \to \mathbf{id}$	addType($\mathbf{id}$.entry, L.in)

图 4.2 给出句子 **int** $\mathbf{id}_1, \mathbf{id}_2, \mathbf{id}_3$ 的注释解析树（仅注释了部分属性）。这些属性值的计算次序是，首先计算根的左子结点的属性 T.type，然后在根的右子树中自上而下地计算 3 个 L 结点的 L.in。在每个 L 结点还调用过程 addType，在符号表中将 **id** 子结点上标识符的类型记为整型。

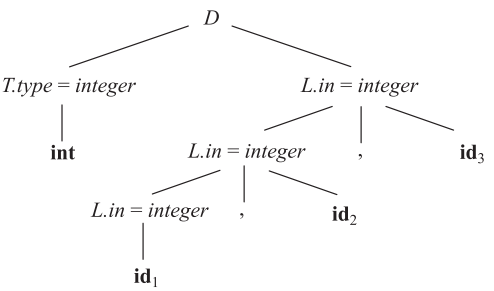


图 4.2 int id1, id2, id3 的注释解析树

使用继承属性的地方很多。例如，可以用继承属性来记住标识符是出现在赋值号的左边还是右边，以便决定是需要它的地址还是需要它的值。

重写语法制导定义使之仅使用综合属性总是可能的，但有时会使得重写后的文法失去简洁和直观，反而不如使用带继承属性的语法制导定义自然。

4.1.4 属性依赖图

如果解析树上一个结点的属性 b 依赖某个结点的属性 c，那么定义属性 b 的语义规则的计算必须在定义属性 c 的语义规则的计算之后。解析树结点的属性之间的相互依赖可以用一种称为**依赖图**的有向图来描绘。

在构造解析树的依赖图之前，首先为由过程调用组成的语义规则引入匿名综合属性 b，这样每条语义规则都能写成 $b = f(c_1, c_2, \cdots, c_k)$ 的形式。依赖图的结构是这样的：解析树上每个结点的每个属性在依赖图上都有一个对应的结点；如果属性 b 依赖于属性 c，那么在依赖

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第152页 文件编号: 422624 页面编号: 31524538449 打印人: 8685 2026-03-09 10:23:01

图中从 c 的结点到 b 的结点有一条有向边。

例如，若 $S.s = f(A.a, B.b, C.c)$ 是产生式 $S \to ABC$ 的语义规则，它定义了依赖于属性 $A.a$、$B.b$ 和 $C.c$ 的综合属性 $S.s$。如果该产生式出现在解析树上，则它在解析树上的每个实例在依赖图中都有对应的 4 个结点 $S.s$、$A.a$、$B.b$ 和 $C.c$，并且结点 $A.a$、$B.b$ 和 $C.c$ 分别有到结点 $S.s$ 的弧。

如果产生式 $S \to ABC$ 的语义规则是 $A.a = f(S.s, B.b, C.c)$，那么在依赖图上，结点 $S.s$、$B.b$ 和 $C.c$ 分别有边到结点 $A.a$，因为 $A.a$ 依赖于 $S.s$、$B.b$ 和 $C.c$。

【例 4.4】 图 4.3 给出了图 4.2 解析树的依赖图。在图 4.3 中，虚线表示的是解析树；依赖图的结点用数字表示，边用实线表示。从结点 4 的 T.type 到结点 5 的 L.in 有一条边，因为根据产生式 $D \to TL$ 的语义规则 $L.in = T$.type，L.in 依赖于 T.type。产生式 $L \to L_1$, **id** 的语义规则 L_1.in = L.in 导致 L_1.in 依赖于 L.in，因此依赖图上有分别到达结点 7 和 9 的两条向下的边。L 产生式的语义规则 addType(**id**.entry, L.in) 导致对应匿名属性的结点，结点 6、8、10 是这样的匿名属性结点。

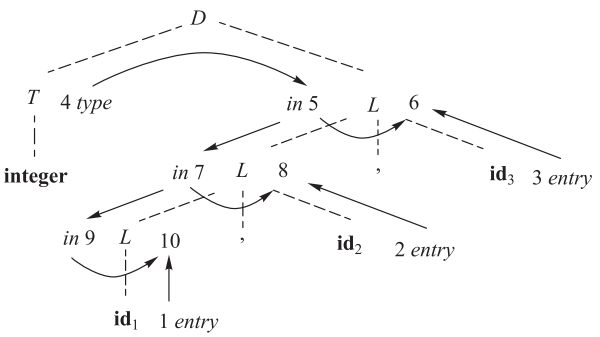


图 4.3 图 4.2 解析树的依赖图

4.1.5 L 属性定义

从图 4.3 中可以看到，类型信息从左流向右，也就是说，属性信息从左向右传递。这种语法制导定义称为 L **属性定义**。语法制导定义是 L 属性的，如果每个产生式 $A \to X_1X_2 \cdots X_n$ 的每条语义规则计算的属性是 A 的综合属性；或者计算的是 X_j 的继承属性 $(1 \leqslant j \leqslant n)$，它仅依赖：① 该产生式中 X_j 左边符号 $X_1, X_2, \cdots, X_{j-1}$ 的属性；② A 的继承属性。

显然，S 属性定义属于 L 属性定义，因为限制①和②仅对继承属性进行约束。

对于像例 4.3 这样含有继承属性的 L 属性定义，语义规则不能都等到归约时才执行。例如，产生式 $D \to TL$ 的语义规则 L.in = T.type 需要在分析由 L 推出的标识符列表之前执行，以便获得 L.in 的值。因此，需要仔细考虑语义规则的执行时机，以及反映执行时机的描述方法。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第153页 文件编号: 422624 页面编号: 31534538447 打印人: 8685 2026-03-09 10:23:01

4.1.6 属性计算次序

拓扑排序是一种对有向无环图中结点的排序方式，记为 $m_1, m_2, \cdots, m_k$。它使得图中每条有向边都只会从排序中靠前的结点指向排序中靠后的结点，也就是若图中存在从 m_i 到 m_j 的边，那么在拓扑排序中 m_i 一定排在 m_j 之前。

显然，依赖图的任何拓扑排序都是解析树中结点属性计算的一个正确次序，即按拓扑排序进行计算，用语义规则 $b = f(c_1, c_2, \cdots, c_k)$ 计算 b 时，属性 $c_1, c_2, \cdots, c_k$ 已经计算出来了。

这样，由语法制导定义规范的翻译可以准确地按下述步骤完成：

（1）首先根据基础文法构造输入的解析树。

（2）按上面讨论的方法构造属性依赖图。

（3）对依赖图的结点进行拓扑排序，得到语义规则的计算次序。

（4）按这个次序计算属性，得到输入串的翻译。

【例 4.5】 在图 4.3 所示的依赖图中，每条边从序号较低的结点到序号较高的结点，因此结点 1, 2, ⋯, 10 构成该图的一个拓扑排序。把依赖图中序号为 n 的结点的属性写成 a_n，那么从这个拓扑排序可以得到下面的程序：

$$
\begin{aligned}
&a_4 = \text{integer};\\
&a_5 = a_4;\\
&\text{addType}(\mathbf{id}_3.\text{entry}, a_5);\\
&a_7 = a_5;\\
&\text{addType}(\mathbf{id}_2.\text{entry}, a_7);\\
&a_9 = a_7;\\
&\text{addType}(\mathbf{id}_1.\text{entry}, a_9);
\end{aligned}
$$

这些语义规则的计算把类型 integer 存于符号表中每个标识符的条目中。

语义规则的这种计算方法称为**解析树方法**。若依赖图有环，则这种方法失败。这种方法的缺点是编译速度很慢，因为属性计算次序是在对特定输入编译过程中确定的。为提高编译速度，需在构造编译器时提前确定属性计算次序，避免在编译时生成依赖图和进行拓扑排序。下面概述的两种方法分别是从手工构造编译器和自动生成编译器的角度考虑的改进。

在构造编译器时，可通过专门的工具或手工分析产生式的语义规则，确定每个产生式所关联的语义规则的计算次序。据此，将计算次序在编译前预先确定下来。编译时，解析树上结点属性的计算就按事先确定的次序进行。这种方法称为**基于规则的方法**，它适用于手工构造编译器。这种方法的缺点是，一些属性依赖关系复杂的语法制导定义很难事先确定属性计算次序，因此这种方法对语法制导定义的种类有限制。

ANTLR、Yacc 等编译器生成工具采用固定的属性计算策略，编译器设计者必须根据所选工具规定的计算策略编写语义规则。如果语义规则不符合该计算策略对语义规则的约束，则属性计算不能正确完成。这种方法称为**忽略规则的方法**（oblivious method），即它不是根

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第154页 文件编号: 422624 页面编号: 31544538445 打印人: 8685 2026-03-09 10:23:01

据语义规则的特点来选择计算策略的，而是要求编译器设计者按照预定的计算策略编写语义规则。这种方法虽然限制了能够实现的语法制导定义的种类，但是能够得到高效的编译器。

基于规则的方法和忽略规则的方法都不需要在编译时显式构造依赖图。与解析树方法相比，它们显著提升了编译器的时空效率。从 4.2 节开始介绍的各种翻译方法都是这两种方法之一。其中 4.2 节介绍的综合属性自下而上计算方法最容易理解，也是编译器生成工具提供的基本方法。

4.2 S 属性定义的自下而上计算

4.1 节初步介绍了如何用语法制导定义来规范翻译。从本节开始，将一边熟悉语法制导定义，一边研究其实现。虽然对任意语法制导定义都适用的翻译器很难建立，但对某些常见的语法制导定义形式，构造其翻译器相对容易。本节以构造语法树为例，来熟悉 S 属性定义，并研究如何在自下而上解析中完成属性计算。

4.2.1 语法树

语法树（syntax tree）是解析树的浓缩表示。在语法树中，运算符和关键字不是作为叶结点，而是作为分支结点。例如，对于产生式 $S \to \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2$，可以把它的右部看成是有一个运算符 **if-then-else** 和 3 个运算对象 B、S_1 和 S_2 的语言构造，它的语法树如图 4.4(a) 所示。语法树中另一个简单的地方是单非产生式链可能消失，图 4.4(b) 是表达式 $8+5*2$ 的语法树。这两个例子的特点是，语法树上的内部结点都代表运算，其子结点都是它的运算对象。

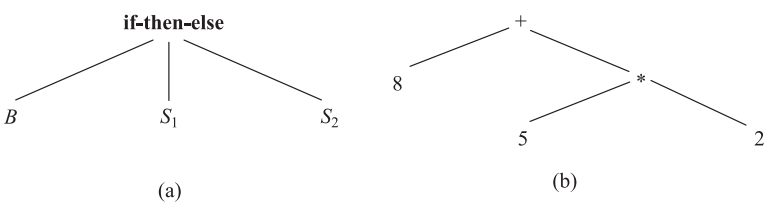


图 4.4 语法树的例子

语法树作为一种中间表示，允许把翻译从解析中分离出来，形成先解析后翻译的方式，即先解析生成语法树，再基于语法树进行翻译。即使是边分析边翻译，语法树作为一种概念上的中间表示，也是有用的。C 和 Java 的编译器通常显式构造语法树。

语法制导翻译可以基于解析树，也可以基于语法树，方法是一样的。如同在解析树中那样，在语法树中也可以把属性附加到结点。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第155页 文件编号: 422624 页面编号: 31554538442 打印人: 8685 2026-03-09 10:23:01

4.2.2 构造语法树的语法制导定义

本节给出构造表达式的语法树的语法制导定义。首先介绍一下数据结构，语法树的结点可以用有若干域的记录来实现。对于运算符结点，一个域存放运算符，该域作为该结点的标记，其余两个域存放指向运算对象的指针。对于基本运算对象结点，一个域存放运算对象类别，另一个域存放其值。当真正用于翻译时，语法树的结点可能还有其他域来保存加在该结点的其他属性值（或属性值的指针）。

下面解释语义规则中用到的函数，这些函数用来建立语法树的叶结点和分支结点，每个函数都返回新建结点的指针：

（1）mkLeaf(**id**, entry)。它建立标记为 **id** 的标识符结点；结点另有一个域，其值等于 entry，它是符号表中该标识符条目的指针。

（2）mkLeaf(**num**, val)。它建立标记为 **num** 的整数结点；结点另有一个域，其值等于 **val**，它是该整数的值。

（3）mkNode(op, left, right)。它用来建立标记为 op 的运算符结点；结点另有两个域，其值分别等于 left 和 right，它们是该结点左右子树的指针。

表 4.3 为含 + 和 * 的表达式构造语法树的 S 属性定义。它给文法的产生式添加语义规则来安排对函数 mkNode 和 mkLeaf 的调用，以便构造语法树。E、T 和 F 的综合属性 nptr 用来记住函数调用返回的指针。

表 4.3 构造表达式语法树的语法制导定义

产生式	语义规则
$E \to E_1 + T$	E.nptr = mkNode('+', E_1.nptr, T.nptr)
$E \to T$	E.nptr = T.nptr
$T \to T_1 * F$	T.nptr = mkNode('*', T_1.nptr, F.nptr)
$T \to F$	T.nptr = F.nptr
$F \to (E)$	F.nptr = E.nptr
$F \to$ **id**	F.nptr = mkLeaf(**id**, **id**.entry)
$F \to$ **num**	F.nptr = mkLeaf(**num**, **num**.val)

【例 4.6】 图 4.5 给出了表达式 $a + 5 * b$ 的注释解析树和执行语义规则后所构造出的语法树，解析树由带点的线表示。标有非终结符 E、T 和 F 的解析树结点，其综合属性 nptr 指向由该非终结符推出的表达式对应的语法树的根结点。

产生式 $F \to$ **id** 和 $F \to$ **num** 的语义规则定义的属性 F.nptr 也是指针，它们分别指向代表标识符和整数的叶结点。**id**.entry 和 **num**.val 是对应记号的属性值。

在图 4.5 中，当表达式 E 只有一项，也就是使用产生式 $E \to T$ 时，属性 E.nptr 和 T.nptr 的值一样。当使用产生式 $E \to E_1 + T$ 的语义规则 E.nptr = mkNode('+', E_1.nptr, T.nptr) 时，

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第156页 文件编号: 422624 页面编号: 31564538440 打印人: 8685 2026-03-09 10:23:01

先前的规则已把 E_1.nptr 和 T.nptr 分别置为指向代表 a 的叶结点和代表 $5*b$ 的分支结点。

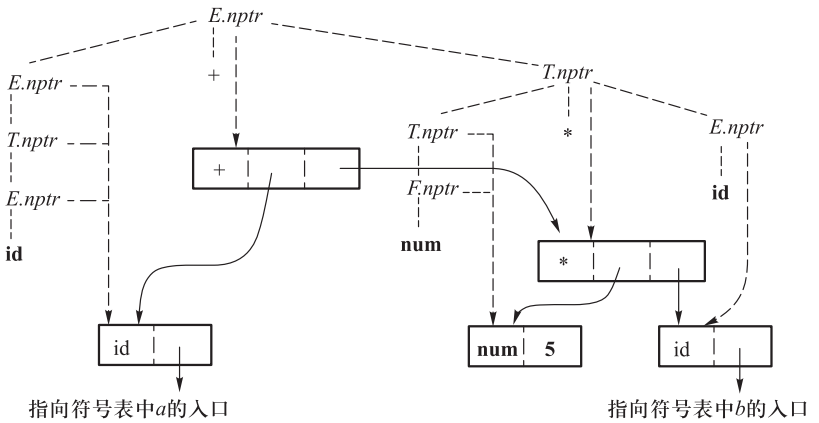


图 4.5 $a+5*b$ 的语法树的构造

处于图 4.5 的下部，由记录形成的树是构成输出的真正语法树；而虚线表示的树是解析树，它可能仅有象征意义。

根据表 4.3 所示的语法制导定义，对于输入 $a+5*b$，实际执行的一系列函数调用如下：

（1）$p_1 = \text{mkLeaf}(\textbf{id}, \text{entrya})$;

（2）$p_2 = \text{mkLeaf}(\textbf{num}, 5)$;

（3）$p_3 = \text{mkLeaf}(\textbf{id}, \text{entryb})$;

（4）$p_4 = \text{mkNode}(\texttt{'*'}, p_2, p_3)$;

（5）$p_5 = \text{mkNode}(\texttt{'+'}, p_1, p_4)$。

可以看出，写语法制导定义比直接用编程语言编写程序更困难。对程序而言，整个程序的执行流程是显式地用语句描述的。而语法制导定义中语义规则的执行受输入记号串的语法结构指导，当输入串中的一个语法构造被识别时（可以看成发生一个事件），其相应的语义规则被执行，因此本质上这是一种事件驱动的程序设计。

4.2.3 S 属性定义的翻译方案

语法制导的翻译方案和语法制导定义的不同之处是它的**语义动作**（在此不称为语义规则）放在 { } 括号内，并且可以插入产生式右部的任何地方。这是一种动作和解析交错的表示法，以表达动作的执行时机。若 $A \to \alpha\{\cdots\}\beta$，则 $\{\cdots\}$ 中语义动作的执行在解析完 α 以后，开始解析 β 之前。可以把 { } 之间的语义动作想象成一个文法符号，在解析过程中对该文法符号进行推导（或归约）的时候，就是该语义动作执行的时候。

当语法制导定义只有综合属性时，即为 S 属性定义，此时翻译方案相对简单：只需为每条语义规则建立一个对应的语义动作，把该动作放在对应产生式右部的末端。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第157页 文件编号: 422624 页面编号: 31574538448 打印人: 8685 2026-03-09 10:23:01

【例 4.7】 如果把表 4.3 构造语法树的语法制导定义变成翻译方案，那么得到如下翻译方案：

$$
\begin{array}{ll}
E \to E_1 + T & \{E.\text{nptr} = \text{mkNode}(\texttt{'+'}, E_1.\text{nptr}, T.\text{nptr});\} \\
E \to T & \{E.\text{nptr} = T.\text{nptr};\} \\
T \to T_1 * F & \{T.\text{nptr} = \text{mkNode}(\texttt{'*'}, T_1.\text{nptr}, F.\text{nptr});\} \\
T \to F & \{T.\text{nptr} = F.\text{nptr};\} \\
F \to (E) & \{F.\text{nptr} = E.\text{nptr};\} \\
F \to \textbf{id} & \{F.\text{nptr} = \text{mkLeaf}(\textbf{id}, \textbf{id}.\text{entry});\} \\
F \to \textbf{num} & \{F.\text{nptr} = \text{mkLeaf}(\textbf{num}, \textbf{num}.\text{val});\}
\end{array}
$$

4.2.4 S 属性的自下而上计算

综合属性可以由自下而上解析器在分析输入的同时完成计算。解析器可以把文法符号的综合属性放在栈中，每当归约时，根据出现在栈顶的产生式右部符号的属性来计算左部符号的综合属性。下面说明如何扩展解析器的栈，使之能够保存综合属性。在 4.3 节将看到这种实现也支持一些有继承属性的情况。

S 属性定义的翻译器可以借助 LR 解析器的生成器来实现，如 3.8 节讨论的 Yacc。解析器的生成器根据 S 属性定义，可以构造出翻译器，它在分析输入的同时计算属性。

自下而上解析器用栈来保存已分析子树的信息。可以在分析栈中增加一个域来保存综合属性，图 4.6 给出了一个例子。拓展后分析栈的每个栈元素由状态域 state 和属性域 val 组成，或者说该栈由 state 和 val 两部分组成。为直观起见，图 4.6 的 state 部分用文法符号来代替状态，这个符号就是图 3.2 所描述的分析栈中被状态直接压住的那个符号。val 部分为文法符号存放综合属性。如果 state 部分某个位置的符号是 A，那么 val 部分相应位置保存解析树上对应 A 的结点的属性值。

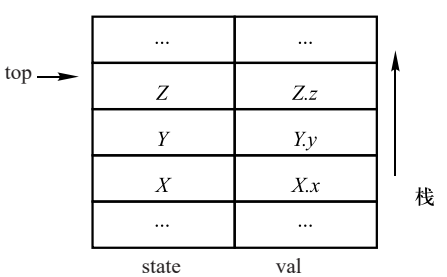


图 4.6 有综合属性域的分析栈

栈顶由指针 top 指示，并假定综合属性刚好在每步归约前计算。如果产生式 $A \to XYZ$ 的语义规则是 $A.a = f(X.x, Y.y, Z.z)$，那么在 XYZ 归约成 A 之前，属性 $Z.z$ 的值在 stack[top].val（指栈顶元素的 val 域），$Y.y$ 的值在 stack[top − 1].val（指次栈顶元素的 val 域，以下类推），$X.x$ 的值在 stack[top − 2].val。如果某个符号没有属性，那么其对应的 val 域没有定义。归约时，覆盖 A 的状态放进 stack[top − 2].state（X 原来的位置），综合属性的值 $A.a$ 放进 stack[top − 2].val，然后 top 的值减 2。

【例 4.8】 再次考虑表 4.1 计算器的语法制导定义。用 3.6 节的技术构造基础文法的 LR 解析器。为了计算属性，需要修改解析器，让它在归约前执行表 4.1 的语义规则，这些语

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第158页 文件编号: 422624 页面编号: 31584538446 打印人: 8685 2026-03-09 10:23:01

义规则可以翻译成表 4.4 的栈操作代码段。这些代码段实际上就是通过用属性在栈中的位置代替表 4.1 语义规则中的属性而得到的。

表 4.4 用 LR 解析器实现计算器

产生式	表 4.1 的语义动作翻译成的代码段
$L \to En$	print(stack[top − 1].val);
$E \to E_1 + T$	stack[top − 2].val = stack[top − 2].val + stack[top].val;
$E \to T$	
$T \to T_1 * F$	stack[top − 2].val = stack[top − 2].val × stack[top].val;
$T \to F$	
$F \to (E)$	stack[top − 2].val = stack[top − 1].val;
$F \to$ **digit**	

表 4.4 中的代码段中，属性计算的结果都置入 top − 2 的位置，因为刚好对应产生式的右部都是 3 个符号。这里的 top 指归约前的栈顶，归约后解析器会根据右部符号的多少调整它的值。注意，对应产生式 $E \to T$、$T \to F$ 和 $F \to$ **digit** 没有代码段，因为相应语义规则的翻译结果是 stack[top].val = stack[top].val，显然可以忽略。

表 4.5 展示了面临输入 $8 + 5 * 2$**n**（注释解析树见图 4.1）时解析器的动作序列。每步动

表 4.5 翻译器面临 $8 + 5 * 2$**n** 时的动作

state	val	输入	所用产生式
–	–	8 +5 * 2**n**	
8	8	+5 * 2**n**	
F	8	+5 * 2**n**	$F \to$ **digit**
T	8	+5 * 2**n**	$T \to F$
E	8	+5 * 2**n**	$E \to T$
$E+$	8+	5 * 2**n**	
$E + 5$	8+5	* 2**n**	
$E + F$	8+5	* 2**n**	$F \to$ **digit**
$E + T$	8+5	* 2**n**	$T \to F$
$E + T*$	8+5*	2**n**	
$E + T * 2$	8+5*2	**n**	
$E + T * F$	8+5*2	**n**	$F \to$ **digit**
$E + T$	8+10	**n**	$T \to T * F$
E	18	**n**	$E \to E + T$
En	18		
L	18		$L \to E$n

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第159页 文件编号: 422624 页面编号: 31594538444 打印人: 8685 2026-03-09 10:23:01

作后分析栈的 state 域和 val 域的内容都在表中给出，仍然用对应的文法符号代替状态。为直观起见，还用实际的输入数字代替记号 **digit**。

考虑看见第一个输入符号 8 时的动作序列。第一步，解析器把对应记号 **digit** 的状态和属性值移进栈（状态由 8 表示，属性值 8 在 val 域）。第二步，解析器用产生式 $F \to \textbf{digit}$ 归约并执行表 4.4 中对应的代码段。第三步，解析器按 $T \to F$ 归约，执行对应的代码段。第四步，解析器按 $E \to T$ 归约，执行对应的代码段。注意，没有代码段和这 3 个产生式相关联，所以 val 域一直不变，但每步归约后栈顶 val 域代表归约后产生式左部符号的属性。

在上面的实现轮廓中，代码段恰好在归约之前执行。归约提供了一种“挂钩”机制，允许任何由代码段组成的动作挂接在归约操作上。这意味着用户可以把代码与产生式关联起来，当解析器按此产生式归约时执行这些代码。

本节实际上揭示了 3.8 节介绍的生成器 Yacc 的语义动作的实现方法，对今后使用 Yacc 和类似的生成器有很大帮助。

4.3 L 属性定义的自上而下计算

当翻译在解析过程中进行时，属性计算的次序受解析方法所限定的解析树结点建立次序的限制。不管是自上而下解析还是自下而上解析，一个共同的特点是，解析树的结点总是自左向右地生成。由于 L 属性定义的属性信息是自左向右流动的，因此可以在解析的同时完成属性计算。本节介绍在自上而下解析中实现 L 属性定义的方法，包括文法转换和预测翻译器的设计。本节介绍的一些技术已经应用于 ANTLR 或类似的 LL 解析器的生成器中。

4.3.1 L 属性定义的翻译方案

从 4.2.3 节可见，S 属性定义的翻译方案相对简单，语义动作放在对应产生式右部的末端。但是，实际应用时可能需要将语义动作内嵌在产生式右部文法符号之间或开头。

【例 4.9】 下面的翻译方案把含 + 和 − 运算符的中缀表达式翻译成后缀表达式。如果输入是 8 + 5 − 2，该翻译方案的输出是 8 5 + 2 −。

$$
\begin{aligned}
&E \to TR \\
&R \to \textbf{addop}\ T\ \{\ \text{print}(\textbf{addop}.\text{lexeme});\}R_1 \mid \varepsilon \\
&T \to \textbf{num}\ \{\text{print}(\textbf{num}.\text{val});\}
\end{aligned}
\tag{4.1}
$$

第 2 行的语义动作 print(**addop**.lexeme) 必须放在 T 和 R_1 之间，移到别处会导致结果错误。

在例 4.9 中，内嵌的语义动作不涉及例 4.4 那样的类型信息传递，因此翻译方案的设计相对简单。在设计翻译方案时，必须保证语义动作在引用属性时其值已经可用，避免引用未

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第160页 文件编号: 422624 页面编号: 31604538442 打印人: 8685 2026-03-09 10:23:01

计算的属性。对于包含继承属性的语法制导定义，其翻译方案在设计时要遵循以下 3 个限制：

（1）产生式右部文法符号的继承属性必须在解析该文法符号之前的语义动作中计算。

（2）一个语义动作不能引用其右边文法符号的综合属性。

（3）产生式左部非终结符的综合属性只能在其依赖的所有属性都计算完后才能计算。通常将计算该属性的动作放在产生式右部的末端。

对 L 属性定义，总能构造满足上述 3 个限制的翻译方案。下述例子说明这一点，它基于早期的数学排版语言 EQN。对于输入

```
E  sub  1  .val
```

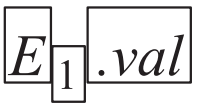


图 4.7 编排单元的排版结果

EQN 编排的 E、1 和.val 的相对位置和相对大小如图 4.7 所示，下标 1 以较小的字体印刷，它的位置也低于 E 和.val。

【例 4.10】 根据表 4.6 的 L 属性定义，构造图 4.8 的翻译方案。表 4.6 中非终结符 B 代表公式编排单元，产生式 $B \to BB$ 代表两个单元的并置，$B \to B$ **sub** B 表示第二个单元作为第一个单元的下标，它的编排尺寸比第一个单元的小，并且位置较低。

表 4.6 定义编排单元大小和高度的语法制导定义

产生式	语义规则
$S \to B$	$B.\text{ps} = 10 \quad S.\text{ht} = B.\text{ht}$
$B \to B_1 B_2$	$B_1.\text{ps} = B.\text{ps} \quad B_2.\text{ps} = B.\text{ps} \quad B.\text{ht} = \max(B_1.\text{ht}, B_2.\text{ht})$
$B \to B_1 \textbf{sub} B_2$	$B_1.\text{ps} = B.\text{ps} \quad B_2.\text{ps} = \text{shrink}(B.\text{ps}) \quad B.\text{ht} = \text{disp}(B_1.\text{ht}, B_2.\text{ht})$
$B \to \textbf{text}$	$B.\text{ht} = \textbf{text}.\text{h} \times B.\text{ps}$

继承属性 ps 表示点的大小，它会影响公式的高度。产生式 $B \to$ **text** 的规则用正文的正常高度乘以点的大小以得到正文的实际高度。**text** 的属性 h 可以根据 **text** 的性质查表得到。在产生式 $B \to B_1 B_2$ 的语义规则中，通过复写规则，B_1 和 B_2 从 B 继承了点的大小。B 的高度由综合属性 ht 表示，它取 B_1 和 B_2 高度的较大值。

在产生式 $B \to B_1$ **sub** B_2 的语义规则中，函数 shrink 将 B_2 的点缩小 30%。函数 disp 在计算 B 的高度时，把单元 B_2 向下偏置。输出编排结果的语义动作在图 4.8 中没有给出。

在表 4.6 中，唯一的继承属性是非终结符 B 的 ps，定义 ps 的语义规则仅依赖产生式左部非终结符的继承属性，所以该定义是 L 属性定义的。

图 4.8 的翻译方案是根据上面三点要求，通过把对应于表 4.6 语义规则的语义动作插入产生式而得到的。为了可读性，把

$$S \to \{B.\text{ps} = 10;\}\ B\ \{S.\text{ht} = B.\text{ht};\}$$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第161页 文件编号: 422624 页面编号: 31614538440 打印人: 8685 2026-03-09 10:23:01

写成了

$$\begin{aligned} S \to \quad & \{B.\mathrm{ps} = 10;\} \\ B \quad & \{S.\mathrm{ht} = B.\mathrm{ht};\} \end{aligned}$$

注意，给继承属性 $B_1.\mathrm{ps}$ 和 $B_2.\mathrm{ps}$ 赋值的动作刚好在产生式右部 B_1 和 B_2 的前面。

$$\begin{aligned} S \to \quad & \{B.\mathrm{ps} = 10;\} \\ \quad B \quad & \{S.\mathrm{ht} = B.\mathrm{ht};\} \\ \\ B \to \quad & \{B_1.\mathrm{ps} = B.\mathrm{ps};\} \\ \quad B_1 \quad & \{B_2.\mathrm{ps} = B.\mathrm{ps};\} \\ \quad B_2 \quad & \{B.\mathrm{ht} = \max(B_1.\mathrm{ht}, B_2.\mathrm{ht});\} \\ \\ B \to \quad & \{B_1.\mathrm{ps} = B.\mathrm{ps};\} \\ \quad B_1 \quad & \\ \quad \mathbf{sub} \quad & \{B_2.\mathrm{ps} = \mathrm{shrink}(B.\mathrm{ps});\} \\ \quad B_2 \quad & \{B.\mathrm{ht} = \mathrm{disp}(B_1.\mathrm{ht}, B_2.\mathrm{ht});\} \\ \\ B \to \mathbf{text} \quad & \{B.\mathrm{ht} = \mathbf{text}.h \times B.\mathrm{ps};\} \end{aligned}$$

图 4.8 从表 4.6 构造的翻译方案

回顾表 4.3 构造表达式语法树的 S 属性定义，其基础文法包含左递归。若想自上而下翻译，就必须消除文法中的左递归，而左递归的消除可能会引起继承属性的出现。下面举例说明。

【例 4.11】 对表 4.3 消除左递归，变换后的翻译方案如图 4.9 所示，其中 F 的产生式和语义动作类似于表 4.3 最初定义中的那些产生式和语义动作。

图 4.10 给出了图 4.9 的动作是怎样为 $a * 5 * b$ 构造语法树的。为使图简单起见，在图 4.10 中略去了 $E \Rightarrow TR \Rightarrow T$ 的部分。综合属性在文法符号的右边，继承属性在文法符号的左边。语法树的叶结点由与产生式 $F \to \mathbf{id}$ 和 $F \to \mathbf{num}$ 关联的动作来构造，与例 4.6 的一样。最左边的 F，其属性 $F.\mathrm{nptr}$ 指向叶结点 a，指向 a 的指针由 TFW 右部 W 的属性 $W.i$ 继承。

当产生式 $W \to *FW_1$ 用于根的右子结点时，$W.i$ 指向 a 结点，$F.\mathrm{nptr}$ 指向结点 5。现在 mkNode 作用于乘运算符和这两个指针，构造出对应 $a * 5$ 的结点。

最后使用的产生式是 $W \to \varepsilon$ 时，$W.i$ 指向整棵树的根。该指针通过 W 结点的 s 属性返回（图 4.10 中没有画出），直至把它赋给 $T.\mathrm{nptr}$（最终赋给 $E.\mathrm{nptr}$ 不在图 4.10 的范围中）。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第162页 文件编号: 422624 页面编号: 31624538448 打印人: 8685 2026-03-09 10:23:01

$E \rightarrow T$ $\{R.i = T.\text{nptr};\}$

R $\{E.\text{nptr} = R.s;\}$

$R \rightarrow +$

T $\{R_1.i = \text{mkNode}('+', R.i, T.\text{nptr});\}$

R_1 $\{R.s = R_1.s;\}$

$R \rightarrow \varepsilon$ $\{R.s = R.i;\}$

$T \rightarrow F$ $\{W.i = F.\text{nptr};\}$

W $\{T.\text{nptr} = W.s;\}$

$W \rightarrow *$

F $\{W_1.i = \text{mkNode}('*', W.i, F.\text{nptr});\}$

W_1 $\{W.s = W_1.s;\}$

$W \rightarrow \varepsilon$ $\{W.s = W.i;\}$

$F \rightarrow (E)$ $\{F.\text{nptr} = E.\text{nptr};\}$

$F \rightarrow \textbf{id}$ $\{F.\text{nptr} = \text{mkLeaf}(\textbf{id}, \textbf{id}.\text{entry});\}$

$F \rightarrow \textbf{num}$ $\{F.\text{nptr} = \text{mkLeaf}(\textbf{num}, \textbf{num}.\text{val});\}$

图 4.9 构造语法树的翻译方案

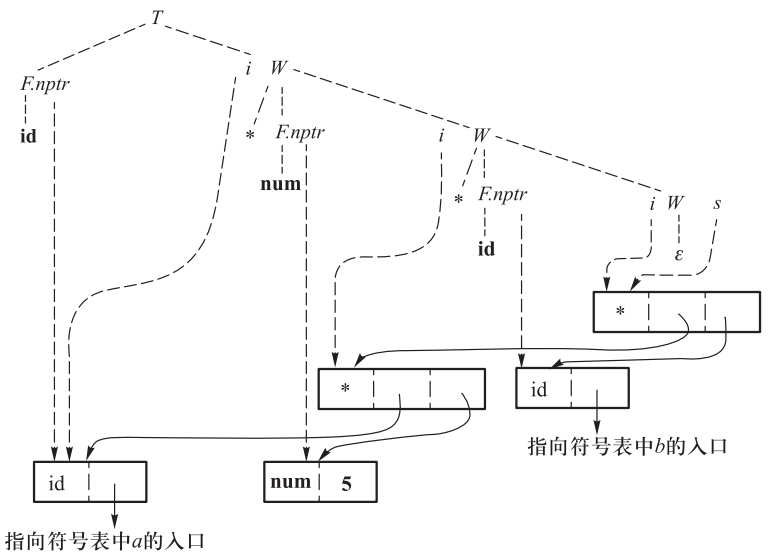


图 4.10 $a * 5 * b$ 的语法树的构造

原来的左递归文法没有继承属性，为什么把文法从左递归改成右递归后会出现继承属性？可以这样直观地解释：对于表 4.3 的文法，属性信息的流动方向和归约方向是一致的，因此，一般来说只用综合属性就可以了；而图 4.9 翻译方案的属性信息（语法树结点指针）是从左

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第163页 文件编号: 422624 页面编号: 31634538446 打印人: 8685 2026-03-09 10:23:01

向右流动的，归约却是从右向左的，这种不一致性导致了继承属性的出现。

4.3.2 预测翻译器的设计

本小节讨论在预测解析的同时完成 L 属性定义。为了能明显看出语义动作执行的时机，本小节基于翻译方案而不是语法制导定义来讨论。

下面的算法把预测解析器的构造方法推广到翻译方案的实现，该翻译方案的文法必须适合自上而下解析。

算法 4.1 构造语法制导的预测翻译器。

输入：语法制导的翻译方案，其基础文法适合预测解析。

输出：语法制导翻译器的代码。

方法：修改预测解析器的构造技术。

（1）为每个非终结符 A 构造一个函数，A 的每个继承属性声明为该函数的一个形式参数，A 的综合属性作为它的返回值（可以是记录，或者是每个属性占一个域的记录的指针）。为简单起见，假定每个非终结符正好只有一个综合属性。A 的函数还为 A 产生式中的其他每个文法符号的每个属性声明一个局部变量。

（2）A 函数的代码骨架与预测解析过程的一致，主要是根据当前输入决定使用什么产生式。

（3）和每个产生式相关联的代码按下面的方法生成，即产生式右部的记号、非终结符和语义动作被从左向右地依次考虑。

① 对于有属性 x 的记号 X，把 x 的值保存在为 $X.x$ 声明的变量中。然后产生匹配记号 X 的调用，并推进输入指针。

② 对于非终结符 B，产生赋值 $c = B(b_1, b_2, \cdots, b_k)$，它的右部是函数调用，其中 b_1，b_2，$\cdots$，b_k 是对应 B 继承属性的变量，c 是代表 B 综合属性的变量。

③ 对于每个语义动作，把代码复制到函数体中，把对属性的引用改成对相应的局部变量的引用。

【例 4.12】 图 4.9 的文法是 LL(1) 的，因而适合自上而下解析。从该文法的非终结符的属性，可以得到函数 E、R、T、W 和 F 的首部如下。因为非终结符 E、T 和 F 没有继承属性，所以对应的函数没有变元。

```
syntaxTreeNode* E()
syntaxTreeNode* R(syntaxTreeNode* i)
syntaxTreeNode* T()
syntaxTreeNode* W(syntaxTreeNode* i)
syntaxTreeNode* F()
```

R 的代码基于图 4.11的解析过程。如果向前看符号是 +，那么使用产生式 $R \to +TR$，先

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第164页 文件编号: 422624 页面编号: 31644538444 打印人: 8685 2026-03-09 10:23:01

用 match 过程去读 + 后面的下一个输入记号，然后调用过程 T 和 R；否则，该产生式什么也不做，使用的是产生式 $R \to \varepsilon$。

```
void R( ) {
    if ( lookahead == '+' ) {
        match ('+'); T( ); R( );
    }
    else /* 什么也不做 */
}
```

图 4.11 产生式 $R \to +TR \mid \varepsilon$ 的解析过程

图 4.12 所示的 R 函数含计算属性的代码，有这样几步：记号 '+' 的值 lexval 存于 addoplexeme，匹配 +，调用 T，用 nptr 保存它的结果。变量 i1 对应继承属性 $R_1.i$, s1 对应综合属性 $R_1.s$。**return** 语句返回 s。其他函数可以效仿这个函数去构造。

```
syntaxTreeNode * R ( syntaxTreeNode * i ) {
    syntaxTreeNode * nptr, * i1, * s1, * s;
    char addoplexeme;

    if ( lookahead == '+' ) { /* 产生式 R→+TR */
        addoplexeme = lexval;
        match('+');
        nptr = T( );
        i1 = mkNode( addoplexeme, i, nptr);
        s1 = R ( i1);
        s = s1;
    }
    else  s = i; /* 产生式 R→ε */
    return s;
}
```

图 4.12 语法树的递归下降构造

4.3.3 用综合属性代替继承属性

从先前的例子已经看出，S 属性定义比 L 属性定义易于理解。从例 4.11 还知道，改变文法可能会引入继承属性。同样，改变文法有时可以避免使用继承属性，后者对于构造只允许综合属性的翻译方案极其有用。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第165页 文件编号: 422624 页面编号: 31654538441 打印人: 8685 2026-03-09 10:23:01

例如，Pascal 语言的变量声明由标识符列表加类型组成，类型出现在变量列表的后面，如 m, n : integer。这种声明可以用下面的文法描述：

$$D \to L : T$$
$$L \to L, \mathbf{id} \mid \mathbf{id}$$
$$T \to \mathbf{integer} \mid \mathbf{char}$$

在这个文法中，标识符列表由 L 产生，但类型 T 不是 L 的子树，因此不能靠仅使用综合属性把类型和标识符联系起来。事实上，在第一个产生式中，由于非终结符 L 从它右边的 T 获得类型信息，所写的语法制导定义不是 L 属性定义的，所以基于它的翻译也不可能在解析期间完成。

这个问题可以通过重新构造文法来解决，把类型作为标识符列表的最后一个成分：

$$D \to \mathbf{id}\ L$$
$$L \to , \mathbf{id}\ L \mid : T$$
$$T \to \mathbf{integer} \mid \mathbf{char}$$

现在，类型作为 L 的综合属性 L.type，当标识符由 L 产生时，它的类型可以填入符号表，具体的翻译方案如下：

$$D \to \mathbf{id}\ L \qquad \{\text{addType}(\mathbf{id}.\text{entry}, L.\text{type});\}$$
$$L \to , \mathbf{id}\ L_1 \qquad \{L.\text{type} = L1.\text{type}; \text{addType}(\mathbf{id}.\text{entry}, L_1.\text{type});\}$$
$$L \to : T \qquad \{L.\text{type} = T.\text{type};\}$$
$$T \to \mathbf{integer} \qquad \{T.\text{type} = \text{integer};\}$$
$$T \to \mathbf{char} \qquad \{T.\text{type} = \text{char};\}$$

可以说，在 Pascal 声明中，类型信息是从右向左流动的。本小节开始给出的文法是按从左向右归约设计的，因此无法在解析期间完成把类型信息填入符号表的计算。而修改后的文法是按从右向左归约设计的，类型信息流向和归约方向的一致性使得属性计算可以在解析期间完成。

4.4 L 属性定义的自下而上计算

本节提出在自下而上解析的框架中实现 L 属性定义的方法。它能实现任何基于 LL(1) 文法的 L 属性定义，也能实现许多（但不是所有的）基于 LR(1) 的 L 属性定义。该方法是对 4.2.4 节自下而上翻译技术的推广。本节所讲的一些技术已经应用于 Yacc 或类似的生成器中。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第166页 文件编号: 422624 页面编号: 31664538449 打印人: 8685 2026-03-09 10:23:01

4.4.1 删除翻译方案中的内嵌动作

4.2.4 节的自下而上翻译方法适用于 S 属性定义，S 属性定义的语义动作都可以放在产生式的右端，在归约时执行。虽然 L 属性定义的继承属性计算需要嵌在产生式右部的不同地方，但是可以通过修改文法来把翻译方案中所有的嵌入动作都变换成只出现在产生式的右端，当嵌入动作只涉及匿名属性时，这种变换尤其容易。

这种变换在文法中加入推出空串的**标记非终结符**，使每个内嵌动作由不同标记非终结符 M 代表，并将该动作放在产生式 $M \to \varepsilon$ 的右端。例如，使用标记非终结符 M 和 N，改写翻译方案（4.1）为

$$
\begin{aligned}
&E \to TR \\
&R \to +T\{\text{print('+');}\}\ R_1 \mid -T\{\text{print('-');}\}\ R_1 \mid \varepsilon \\
&T \to \textbf{num}\ \{\text{print}(\textbf{num}.\text{val});\}
\end{aligned}
$$

变换成

$$
\begin{aligned}
&E \to TR \\
&R \to +TMR_1 \mid TNR_1 \mid \varepsilon \\
&T \to \textbf{num}\ \{\text{print}(\textbf{num}.\text{val});\} \\
&M \to \{\text{print('+');}\} \\
&N \to \{\text{print('-');}\}
\end{aligned}
$$

这两个翻译方案中的文法接受同样的语言，并且对任何输入，它们执行的语义动作是一样的。变换后的翻译方案中，动作都在产生式的右边，所以它们可以在自下而上解析过程中归约产生式右部的时候完成。

4.4.2 分析栈上的继承属性

对产生式 $A \to XY$，自下而上解析器从它的栈顶移出 Y 和 X，并用 A 代替它们，完成产生式右部的归约。如果 X 有综合属性 $X.s$，4.2.4 节的实现把该属性放在栈中与 X 对应的地方。

因为在 Y 以下的任何子树归约前，$X.s$ 的值已经在栈中，所以它的值可以被 Y 继承。如果 Y 的继承属性由复写规则 $Y.i = X.s$ 定义，那么在需要使用 $Y.i$ 的地方，可以取 $X.s$ 的值来代替。如果能静态地确定 $X.s$ 在栈中的位置，那么这个想法很容易实现。

【例 4.13】 考虑由表 4.2 的语法制导定义改写的翻译方案：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第167页 文件编号: 422624 页面编号: 31674538447 打印人: 8685 2026-03-09 10:23:01

$$
\begin{array}{lll}
D \to & T & \{L.\text{in} = T.\text{type};\} \\
 & L & \\
T \to & \textbf{int} & \{T.\text{type} = \text{integer};\} \\
T \to & \textbf{real} & \{T.\text{type} = \text{real};\} \\
L \to & & \{L_1.\text{in} = L.\text{in};\} \\
 & L_1, \textbf{id} & \{\text{addType}(\textbf{id}.\text{entry}, L.\text{in});\} \\
L \to & \textbf{id} & \{\text{addType}(\textbf{id}.\text{entry}, L.\text{in});\}
\end{array}
$$

该翻译方案使用继承属性，并且继承属性由复写规则传递。下面以输入 int p, q, r 为例，考察在需要 L.in 值的地方，能否静态地确定 T.type 在栈中的位置。

先用图 4.13 给出该输入的解析树及 type 和 in 属性之间的依赖关系，以增强直观性。表 4.7 给出了解析器面临该输入时的动作序列。为清楚起见，仍然用对应的文法符号代替状态，用实际的标识符代替记号 **id**。在表 4.7 中，每次归约 L 产生式右部时，T 在栈中刚好处于该右部的下面，因此 T.type 在栈中的位置可以静态地确定。这个事实可用来访问属性 T.type。

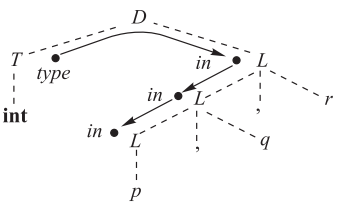


图 4.13 在每个 L 结点上，L.in = T.type

表 4.7 解析器面临 **int** p, q, r 时的动作序列

状态	输入	所用产生式
–	**int** p, q, r	
int	p, q, r	
T	p, q, r	$T \to$ **int**
Tp	$, q, r$	
TL	$, q, r$	$L \to$ **id**
TL	q, r	
TL, q	$, r$	
TL	$, r$	$L \to L,$ **id**
TL	r	
TL, r		
TL		$L \to L,$ **id**
D		$D \to TL$

按照 4.2.4 节的假设，分析栈由 state 和 val 两部分来实现，并且如果 stack[i].state 对应文法符号 X，那么 stack[i].val 保存 X 的综合属性 s。表 4.8 的实现利用了属性 T.type 在栈中的位置已知这个事实。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第168页 文件编号: 422624 页面编号: 31684538445 打印人: 8685 2026-03-09 10:23:01

表 4.8 T.type 的值可用来代替 L.in

产生式	代码段
$D \to TL$	
$T \to$ **int**	stack[top].val = integer;
$T \to$ **real**	stack[top].val = real;
$L \to L_1$, **id**	addType(stack[top].val, stack[top − 3].val);
$L \to$ **id**	addType(stack[top].val, stack[top − 1].val);

当使用产生式 $L \to$ **id** 时，**id**.entry 在 val 栈顶，T.type 刚好在它的下面。所以 addType (stack[top].val, stack[top−1].val) 等价于 addType(**id**.entry, L.in)。同样，因为产生式 $L \to L$, **id** 右部有 3 个符号，所以归约时 T.type 出现在 stack[top − 3].val。其他与 L.in 有关的动作是复写规则，它们完全可以省略掉。

如果属性的位置不能预测，可以尝试增加标记非终结符，使属性的位置变得可以静态确定。下面通过一个例子来说明。

【例 4.14】 考虑下面的语法制导定义：

$$\begin{array}{ll} \text{产生式} & \text{语义规则} \\ S \to aAC & C.i = A.s \\ S \to bABC & C.i = A.s \\ C \to c & C.s = g(C.i) \end{array} \tag{4.2}$$

通过复写规则，$C.i$ 继承综合属性 $A.s$。由于在栈中，B 可能在也可能不在 A 和 C 之间，在用产生式 $C \to c$ 归约时，$C.i$ 的值（也就是 $A.s$ 的值）在 stack[top−2].val 或者在 stack[top−1].val。

增加一个标记非终结符 M，插在语法制导定义式（4.2）的第二个产生式右部的 C 之前，并增加了属性复写规则。修改后的语法制导定义如下：

产生式	语义规则
$S \to aAC$	$C.i = A.s$
$S \to bABMC$	$M.i = A.s \quad C.i = M.s$
$C \to c$	$C.s = g(C.i)$
$M \to \varepsilon$	$M.s = M.i$

产生式 $S \to bABMC$ 和 $M \to \varepsilon$ 的语义规则保证 $C.i = M.s = M.i = A.s$，也就是对于第二个产生式 $S \to bABMC$，$C.i$ 间接地通过 $M.i$ 和 $M.s$ 继承 $A.s$ 的值。这样，不管在 aAC 还是 $bABMC$ 的情况下，在使用 $C \to c$ 时，$C.i$ 的值都可以在 stack[top − 1].val 中找到。在使用 $M \to \varepsilon$ 时，$M.i$ 的值可以在 stack[top − 1].val 中找到。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第169页 文件编号: 422624 页面编号: 31694538443 打印人: 8685 2026-03-09 10:23:01

4.4.3 模拟继承属性的计算

如果继承属性并不直接等于某个综合属性，而是它的一个函数，能否利用上一小节的办法？回答是可以的。在这种情况下，可以用标记非终结符来模拟继承属性的计算。例如，考虑

产生式	语义规则
$S \to aAC$	$C.i = f(A.s)$
$C \to c$	$C.s = g(C.i)$

这里，定义 $C.i$ 的规则不是简单的复写规则。如果不完成这个计算，则在栈中取不到 $C.i$ 的值。解决办法是增加一个标记非终结符，把 $f(A.s)$ 的计算移到对标记非终结符归约时进行，如下所示：

产生式	语义规则
$S \to aANC$	$N.i = A.s \quad C.i = N.s$
$N \to \varepsilon$	$N.s = f(N.i)$
$C \to c$	$C.s = g(C.i)$

当按 $N \to \varepsilon$ 归约时，$N.i$ 的值可以在放 $A.s$ 的地方（stack[top].val）找到。当按 $C \to c$ 归约时，$C.i$ 的值可以在对应于 N 的地方（stack[top − 1].val）找到。

【例 4.15】 对于表 4.6 的排版语言例子，可以增加 3 个标记非终结符 L、M 和 N，以保证当 B 的子树被归约时，所需的继承属性 B.ps 的值出现在分析栈中已知的位置。修改后的语法制导定义见表 4.9。

表 4.9 由复写规则设置所有继承属性

产生式	语义规则
$S \to LB$	$B.\text{ps} = L.s \quad S.\text{ht} = B.\text{ht}$
$L \to \varepsilon$	$L.s = 10$
$B \to B_1 M B_2$	$B_1.\text{ps} = B.\text{ps} \quad M.i = B.\text{ps} \quad B_2.\text{ps} = M.s \quad B.ht = \max(B_1.\text{ht}, B_2.\text{ht})$
$M \to \varepsilon$	$M.s = M.i$
$B \to B_1 \textbf{sub} N B_2$	$B_1.\text{ps} = B.\text{ps} \quad N.i = B.\text{ps} \quad B_2.\text{ps} = N.s \quad B.\text{ht} = \text{disp}(B_1.\text{ht}, B_2.\text{ht})$
$N \to \varepsilon$	$N.s = \text{shrink}(N.i)$
$B \to \textbf{text}$	$B.\text{ht} = \textbf{text}.h \times B.\text{ps}$

标记非终结符 L 的作用是完成 $S \to LB$ 中的属性 B.ps 的初始化。$L \to \varepsilon$ 的语义规则 $L.s = 10$ 使得 B.ps 的初值 10 在栈中有存放的地方，以便后面引用。

标记非终结符 M 和 N（还有 L）的一个共同作用是，保证了在任何情况下向 B 归约时，B.ps 的值总是正好在右部的下面。这个结论的证明留做一个练习。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第170页 文件编号: 422624 页面编号: 31704538441 打印人: 8685 2026-03-09 10:23:01

N 的另一个作用是模拟继承属性的计算。在表 4.6 中，产生式 $B \to B_1$ **sub** B_2 的语义规则 $B_2.\text{ps} = \text{shrink}(B.\text{ps})$ 表明 $B_2.\text{ps}$ 不是简单地由复写规则定义的。产生式 $N \to \varepsilon$ 的动作完成这个计算，并将计算结果保留在栈上。

实现表 4.9 语法制导定义的代码段列在表 4.10 中。在表 4.9 中，所有的继承属性都由复写规则来赋值，因而只要知道它们在栈中的位置，就可以获得它们的值。

表 4.10 表 4.9 的语法制导定义的实现

产生式	语义规则
$S \to LB$	stack[top − 1].val = stack[top].val;
$L \to \varepsilon$	stack[top + 1].val = 10;
$B \to B_1 M B_2$	stack[top − 2].val = max(stack[top − 2].val, stack[top].val);
$M \to \varepsilon$	stack[top + 1].val = stack[top − 1].val;
$B \to B_1$**sub**$N B_2$	stack[top − 3].val = disp(stack[top − 3].val, stack[top].val);
$N \to \varepsilon$	stack[top + 1].val = shrink(stack[top − 2].val);
$B \to$ **text**	stack[top].val = stack[top].val × stack[top − 1].val;

可以证明，对于基于 LL(1) 文法的 L 属性定义，系统地引入标记非终结符后，可以在 LR 解析期间完成属性计算。因为每个标记非终结符至多只有一个产生式，因此标记非终结符加入后文法仍然是 LL(1) 的。任何 LL(1) 文法一定是 LR(1) 文法，因此加标记非终结符后不会引起 LR 分析动作的冲突。

但是对于基于 LR(1) 文法的 L 属性定义，情况就不这么简单，标记非终结符的加入有可能使得文法不再是 LR(1) 的了。

【例 4.16】 文法 $L \to Lb \mid a$ 是 LR(1) 的，加了标记非终结符 M 后的文法

$$L \to MLb \mid a$$
$$M \to \varepsilon$$

不是 LR（1）的。

该语言的句子是 $abb\cdots b$ 的形式。根据这个文法，空归约的个数和 b 的个数一样多。在面临 a 时，后面有多少个 b 是不知道的，因此一定会有移进-归约冲突。所以，这个加标记非终结符的文法不是 LR(1) 的。

除 4.1 节所讨论的解析树方法以外，本章具体讨论的都是边解析边计算属性的方法。它们的优点是效率较高，缺点是访问结点的次序受到解析方法的限制，因而只能完成 L 属性计算。

对于非 L 属性定义，可以将解析和属性计算分开，先建立解析树（也可以用语法树），然后遍历解析树来计算属性。可以将 4.3 节的预测解析技术推广，仍然为每个非终结符构造一个函数，但是这些函数不含语法分析部分，它们是通过遍历解析树来完成属性计算的递归函

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第171页 文件编号: 422624 页面编号: 31714538449 打印人: 8685 2026-03-09 10:23:01

数。每个函数对解析树中相应结点的子结点的访问次序由产生式的语义规则决定，而无须要求从左到右。

这种方法不受解析方法的限制，属性计算的次序可在构造编译器时通过分析语法制导定义来得出，而且可以推广到对解析树需要进行多遍扫描的翻译。

4.5 基于树访问的翻译

在编译器设计中，抽象语法树（abstract syntax tree, AST）是表示源代码结构的核心数据结构。遍历 AST 是执行程序分析和转换操作的关键步骤。为了在 AST 遍历过程中对不同类型的结点进行特定处理，访问者模式（visitor pattern）成为一种常用设计模式。它通过将对树结点的操作封装到访问者类中，实现了操作与结点结构的解耦，而无须修改结点结构即可灵活添加新操作。这种设计不仅便于扩展结点类，还允许通过新增不同访问者类来执行多样化的分析或转换操作。通过这种方式，编译器能够在保持结点类稳定的前提下，逐步增加更多的分析和处理能力，从而提升系统的可维护性和可扩展性。

4.5.1 访问者模式

访问者模式的核心是定义一个访问者接口 Visitor，通过接口中的不同方法访问对象结构中的各结点类型。例如，visit××(node) 表示访问 ×× 类型结点实例 node，并执行对该结点的特定操作，如类型检查、计算或代码生成等。每个结点类型都实现了一个 accept（Visitor）方法，通过将自身（this）传递给访问者，实现对结点的具体操作。

在访问者模式中，访问者通常包含以下几类方法，以便在遍历 AST 时执行不同类型的操作：

（1）**visit××(node) 方法**（或 visit 方法）：最基本的访问方法，通常用于处理 ×× 类结点的通用操作。每当访问者遍历到某个特定类型 ×× 的结点时，会调用相应的 visit×× 方法。在这个方法中可以包含对该结点进行操作的代码。它是对某一类结点（如所有表达式结点）的处理入口，并可以调用进入和退出方法完成对结点的详细操作。

（2）**enter××(node) 方法**：在进入特定结点之前执行的操作。每当访问者遍历进入某个特定类型 ×× 的结点时（在处理子结点之前），访问者会调用相应的 enter×× 方法，如 enterExpr 或 enterStmt。这种方法适用于需要在处理结点内部之前执行初始化操作的情况，如建立作用域、分配内存、记录日志等。

（3）**exit××(node) 方法**：在处理完特定结点之后执行的操作。一旦访问者处理完该结点及其所有子结点，exit×× 方法（如 exitExpr 或 enterStmt）会被调用。这个方法通常用于清理、恢复上下文、汇总结果或进行某些后置操作。它确保在离开结点时，所有子结点处理都已经完成。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第172页 文件编号: 422624 页面编号: 31724538447 打印人: 8685 2026-03-09 10:23:01

（4）**defaultVisit(node) 方法：**提供默认行为以处理未定义的结点类型。如果访问者遇到没有特定方法处理的结点，它会调用 defaultVisit 方法。这可以是一个通用的访问方法，用于捕获所有未显式定义的结点类型。在复杂的树结构中，defaultVisit 方法可以避免访问者模式需要定义过多的特定方法，同时为未定义的结点类型提供统一的行为。

（5）**visitChildren(node) 方法：**递归遍历当前结点的所有子结点。这个方法会遍历 node 的所有子结点，并对每个子结点调用 enter××、visit×× 和 exit××。它在 visit×× 方法中调用，确保当前结点的所有子结点都得到访问和处理，如遍历语句块中的每个语句或函数体中的每条指令等。

通过 accept 方法和 visit×× 方法，可以实现双分发。**双分发**的原理在于，结点通过 accept 方法的第一次分发选择合适的访问者方法，随后访问者根据结点类型进行第二次分发，调用 visit×× 方法实施具体的处理逻辑。这种模式将结点类型与访问操作解耦，增强了系统的灵活性。

4.5.2 ANTLR 解析对访问者模式的支持

ANTLR 支持访问者模式，并在生成的代码中提供解析树的访问者接口。解析树和解析树访问者是构建和操作 AST 的关键概念。本小节以 ANTLR 的 C++ 运行时库（以下简称 antlr4-runtime）中的核心接口（图 4.14和图 4.15）和为示例语言 C1 生成的 C++ 源码解析器 C1Parser（图 4.16）为例进行介绍。

1. ANTLR 运行时库提供的解析树及其访问和操作接口

ANTLR 的运行时库定义了解析树类 ParseTree 及其访问者接口类 ParseTreeVisitor 和监听器接口类 ParseTreeListener，分别位于 antlr4-runtime/tree 目录下的 ParseTree.h、ParseTreeVisitor.h、ParseTreeListener.h 文件中，如图 4.14 所示。

```
1  // ANTLR运行时库提供的解析树结点类，位于antlr4-runtime/tree/ParseTree.h
2  class ANTLR4CPP_PUBLIC ParseTree {
3    public:
4      ParseTree();
5      ......
6      ParseTree *parent;
7      std::vector<ParseTree *> children;
8      virtual antlrcpp::Any accept(ParseTreeVisitor *visitor) = 0;
9  }
10
11 // ANTLR运行时库提供的解析树访问者接口类，位于antlr4-runtime/tree/ParseTreeVisitor.h
12 class ANTLR4CPP_PUBLIC ParseTreeVisitor {
13   public:
14     virtual ~ParseTreeVisitor();
15     virtual antlrcpp::Any visit(ParseTree *tree) = 0;
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第173页 文件编号: 422624 页面编号: 31734538445 打印人: 8685 2026-03-09 10:23:01


```
16      virtual antlrcpp::Any visitChildren(ParseTree *node) = 0;
17      virtual antlrcpp::Any visitTerminal(TerminalNode *node) = 0;
18      virtual antlrcpp::Any visitErrorNode(ErrorNode *node) = 0;
19  };
20
21  // ANTLR运行时库提供的解析树监听器接口类，位于antlr4-runtime/tree/ParseTreeListener.h
22  class ANTLR4CPP_PUBLIC ParseTreeListener {
23    public:
24      virtual ~ParseTreeListener();
25      virtual void visitTerminal(TerminalNode *node) = 0;
26      virtual void visitErrorNode(ErrorNode *node) = 0;
27      virtual void enterEveryRule(ParserRuleContext *ctx) = 0;
28      virtual void exitEveryRule(ParserRuleContext *ctx) = 0;
29      bool operator == (const ParseTreeListener &other) {
30        return this == &other;
31      }
32  };
```

图 4.14 ANTLR 的 ParseTree、ParseTreeVisitor 和 ParseTreeListener 的定义

ParseTree 包含一个重要方法 accept(ParseTreeVisitor *visitor)，用于接收访问者对象并调用其中的特定访问操作。

ParseTreeVisitor 接口类包含 visit(ParseTree *tree)、visitChildren(ParseTree *node)、visitTerminal(TerminalNode *node)、visitErrorNode(ErrorNode *node)4 个接口方法，分别用于访问解析树结点、所有子结点、终结符结点和错误结点。ParseTreeVisitor 以访问者模式提供对解析树的访问和操作，使用深度优先遍历方式，以递归的形式访问解析树的各个结点，在访问过程中会依次调用 visit 和 visitChildren 方法；结点的 accept 方法会调用与之对应的 visit 方法。用户通过重写 visit×× 方法（如后文 C1ParserVisitor 中的方法），可以在遇到特定类型结点时执行操作，并返回结果值。

ParseTreeListener 接口类包含 enterEveryRule(ParserRuleContext *ctx)、exitEveryRule(ParserRuleContext *ctx)、visitTerminal(TerminalNode *node)、visitErrorNode(ErrorNode *node)4 个接口方法，分别用于处理进入每个规则结点、退出每个规则结点、遇到终结符结点和遇到错误结点时要执行的操作。ParseTreeListener 以监听器模式提供对解析树的访问和操作，使用事件驱动的方式，通过回调机制自动触发，每进入或退出一个结点时会调用相应结点的 enter 和 exit 方法，从而实现解析树遍历。用户通过重写 enter×× 和 exit×× 方法（如后文 C1ParserListener 中的方法），可以分别在进入和退出特定结点时执行操作；这些方法不会返回值，因为它主要用于执行操作，而不是获取结果。

在 ANTLR C++ 实现中，antlrcpp::Any 是一个通用容器，能够灵活地处理不同类型的数据。特别是在访问者模式下，antlrcpp::Any 可以简化返回值的管理，并提高类型安全性。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第174页 文件编号: 422624 页面编号: 31744538443 打印人: 8685 2026-03-09 10:23:01

ANTLR 提供了 AbstractParseTreeVisitor 类（位于 AbstractParseTreeVisitor.h），作为 ParseTreeVisitor 的一个默认实现。其中，提供了 visit 和 visitChildren 方法的默认实现，如图 4.15 所示。visit 方法的默认实现会调用解析树结点的 accept 方法，实现访问树结点的操作；而 visitChildren 方法的默认实现则会依次调用每个子结点的 accept 方法，从而实现对树的递归访问。这种设计通过双分发机制，允许用户统一使用 visit 或 visit×× 等方法对解析树进行遍历访问，而不必直接处理每个结点的细节。这一默认实现为用户简化了解析树的遍历工作，同时提供了灵活性，使用户可以根据需要重写特定的访问方法。

```
// ANTLR运行时库提供的解析树访问者接口的默认实现，位于antlr4-runtime/tree/
       AbstractParseTreeVisitor.h
class ANTLR4CPP_PUBLIC AbstractParseTreeVisitor : public ParseTreeVisitor {
  public:
    virtual antlrcpp::Any visit(ParseTree *tree) override {
      return tree->accept(this);
    }
    virtual antlrcpp::Any visitChildren(ParseTree *node) override {
      antlrcpp::Any result = defaultResult();
      size_t n = node->children.size();
      for (size_t i = 0; i < n; i++) {
        if (!shouldVisitNextChild(node, result)) {
          break;
        }
        antlrcpp::Any childResult = node->children[i]->accept(this);
        result = aggregateResult(result, childResult);
    }
    return result;
  }
  ......
};
```

图 4.15 ANTLR 的 AbstractParseTreeVisitor 的定义

2. ANTLR 生成的解析器中的主要类

假设用 ANTLR 为 C1 语言构造解析器，C1 语言的文法描述中，开始符号假定为 compilationUnit，表示整个 C1 程序的语法结构。ANTLR 将根据 C1 文法生成解析器类 C1Parser、解析器访问者接口类 C1ParserVisitor 和监听器接口类 C1ParserListener，它们分别是 antlr4::ParseTree（图 4.14）、antlr4::tree::AbstractParseTreeVisitor（图 4.15）和 antlr4::tree::ParseTreeListener（图 4.14）的派生类。解析树的结点通常由 ParserRuleContext 类及其子类构成，代表文法中的每条规则。每条语法规则在解析树中都对应一个上下文类，这些类用于存储与特定语法构造相关的信息。例如，为 C1 语言的开始符号 CompilationUnit 定义了图 4.16

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第175页 文件编号: 422624 页面编号: 31754538440 打印人: 8685 2026-03-09 10:23:01

所示的上下文类 CompilationUnitContext。该上下文类中包含 accept 方法，支持用传入的访问者访问该类型结点；还包含 enterRule 和 exitRule 方法，支持用传入的监听器触发 enterRuleName 和 exitRuleName 方法（其中 RuleName 是规则的名称）。ANTLR 会为每个上下文类自动生成 accept、enterRule、exitRule 方法的实现，这些方法定义了访问该上下文类结点的流程。有关上下文类 CompilationUnitContext 的这 3 个方法的定义如图 4.16 中的第 15～33 行。

```
1  class  CompilationUnitContext : public antlr4::ParserRuleContext {
2    public:
3      CompilationUnitContext(antlr4::ParserRuleContext *parent, size_t invokingState);
4      virtual size_t getRuleIndex() const override;
5      antlr4::tree::TerminalNode *EOF();
6      std::vector<DeclContext *> decl();
7      DeclContext* decl(size_t i);
8      std::vector<FuncdefContext *> funcdef();
9      FuncdefContext* funcdef(size_t i);
10
11     virtual void enterRule(antlr4::tree::ParseTreeListener *listener) override;
12     virtual void exitRule(antlr4::tree::ParseTreeListener *listener) override;
13     virtual antlrcpp::Any accept(antlr4::tree::ParseTreeVisitor *visitor) override;
14 };
15 // ANTLR v4 生成的enterRule方法
16 void C1Parser::CompilationUnitContext::enterRule(tree::ParseTreeListener *listener) {
17   auto parserListener = dynamic_cast<C1ParserListener *>(listener);
18   if (parserListener != nullptr)
19     parserListener->enterCompilationUnit(this);
20 }
21 // ANTLR v4 生成的exitRule方法
22 void C1Parser::CompilationUnitContext::exitRule(tree::ParseTreeListener *listener) {
23   auto parserListener = dynamic_cast<C1ParserListener *>(listener);
24   if (parserListener != nullptr)
25     parserListener->exitCompilationUnit(this);
26 }
27 // ANTLR v4 生成的accept方法
28 antlrcpp::Any C1Parser::CompilationUnitContext::accept(tree::ParseTreeVisitor *
         visitor) {
29     if (auto parserVisitor = dynamic_cast<C1ParserVisitor*>(visitor))
30         return parserVisitor->visitCompilationUnit(this);
31     else
32         return visitor->visitChildren(this);
33 }
34 // 根据文法生成的解析器访问者类，假设为C1语言文法
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第176页 文件编号: 422624 页面编号: 31764538448 打印人: 8685 2026-03-09 10:23:01


```
35 class  C1ParserVisitor : public antlr4::tree::AbstractParseTreeVisitor {
36   public:
37     virtual antlrcpp::Any visitCompilationUnit(C1Parser::CompilationUnitContext *
         context) = 0;
38     virtual antlrcpp::Any visitDecl(C1Parser::DeclContext *context) = 0;
39     ......
40 }
41 // 根据文法生成的监听器类，假设为C1语言文法
42 class  C1ParserListener : public antlr4::tree::ParseTreeListener {
43 public:
44   virtual void enterCompilationUnit(C1Parser::CompilationUnitContext *ctx) = 0;
45   virtual void exitCompilationUnit(C1Parser::CompilationUnitContext *ctx) = 0;
46     ......
47 }
```

图 4.16 ANTLR 生成的 C1Parser 相关的代码片段

在 C1ParserVisitor 访问者类中，文法的每个语法规则（假设规则名为 RuleName）都有对应的 visitRuleName 方法，如 visitCompilationUnit、visitDecl。这样就可以通过实现访问者接口，将对各个结点的操作定义在具体的访问者类中。

在 C1ParserListener 监听器类中，文法的每个语法规则都有对应的 enterRuleName 和 exitRuleName 方法，如 enterCompilationUnit、enterDecl、exitCompilationUnit、exitDecl。这样就可以通过实现监视器接口，将进入和退出特定规则结点需要执行的操作定义在具体的监视器类中。

回顾 3.4 节，用户可以为特定语法规则添加标签，例如：

```
e :  e '+' e # Add  | INT    # Int ;
```

ANTLR 会根据这些标签为对应的规则选择生成更具描述性的规则上下文子类：AddContext 和 IntContext，以及关联的字段名或方法，便于访问这些特定结点。用 P 可以根据标签很容易在生成的解析器源码中搜索到与该规则相关的上下文类及方法等定义。

在理解 ANTLR 解析产生的解析树及其访问者的相关定义之后，就可以着手利用它们构造树访问的翻译方案了。例如，定义如下名称的 AST 构造类：

```
class syntax_tree_builder : public C1ParserBaseVisitor
```

该类中可以包含要实现的特定语法构造类型 ×× 的 visit××、enter××、exit×× 等方法。更多的内容可以参见提供的“使用 ANTLR 为 C1 语言构造生成 AST 的解析器”实验的实验软件包。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第177页 文件编号: 422624 页面编号: 31774538446 打印人: 8685 2026-03-09 10:23:01

习 题

4.1 根据表 4.1 的语法制导定义，为输入表达式 $5*(4*3+2)$ 构造注释解析树。

4.2 构造表达式 $((a*b)+(c))$ 的解析树和语法树：

(a) 根据表 4.3 的语法制导定义。

(b) 根据图 4.9 的翻译方案。

4.3 为文法

$S \to (L) \mid a$

$L \to L, S \mid S$

(a) 写一个语法制导定义，它输出括号的对数。

(b) 写一个语法制导定义，它输出括号嵌套的最大深度。

4.4 下面的文法定义语言 $L = \{a^n b^n c^m \mid m, n \geqslant 1\}$。写一个语法制导定义，其语义规则的作用是对不属于语言 L 的子集 $L_1 = \{a^n b^n c^n \mid n \geqslant 1\}$ 的句子，打印出错信息。

$S \to DC$

$D \to aDb \mid ab$

$C \to Cc \mid c$

4.5 为下面的文法写一个语法制导的定义，它完成一个句子的 **while-do** 最大嵌套层次的计算并输出这个计算结果。

$S \to E$

$E \to \textbf{while}\ E\ \textbf{do}\ E \mid \textbf{id} := E \mid E + E \mid \textbf{id} \mid (E)$

4.6 下列文法产生由 + 作用于整常数或实常数的表达式。两个整数相加时，结果是整型，否则是实型。

$E \to E + T \mid T$

$T \to \textbf{num.num} \mid \textbf{num}$

(a) 给出决定每个子表达式类型的语法制导定义。

(b) 扩充 (a) 的语法制导定义，既决定类型，又把表达式翻译成后缀表示。使用一元运算符 **inttoreal** 把整数变成等价的实数，使得后缀式中 + 的两个对象有同样的类型。

4.7 给出对表达式求导数的语法制导定义，表达式由 + 和 * 作用于变量 x 和常数组成，如 $x*(3*x+x*x)$，并假定没有任何化简，如将 $3*x$ 翻译成 $3*1+0*x$。

***4.8** 给出把中缀表达式翻译成没有冗余括号的中缀表达式的语法制导定义。例如，因为 + 和 * 是左结合，$((a*(b+c))*(d))$ 可以重写成 $a*(b+c)*d$。

4.9 S 的综合属性 val 给出下面文法中 S 产生的二进制数的值。例如，输入 101.101 时，$S.\text{val} = 5.625$。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第178页 文件编号: 422624 页面编号: 31784538444 打印人: 8685 2026-03-09 10:23:01

$$S \to L.L \mid L$$
$$L \to LB \mid B$$
$$B \to 0 \mid 1$$

(a) 仅用综合属性决定 S.val。

(b) 用 L 属性定义决定 S.val。在该定义中，B 的唯一综合属性是 c（还需要继承属性），它给出由 B 产生的位对最终值的贡献。例如，101.101 的最前一位和最后一位对值 5.625 的贡献分别是 4 和 0.125。

4.10 重写例 4.3 语法制导定义的基础文法，然后重新设计语法制导定义，使得仅用综合属性就能把类型信息填入符号表。

4.11 由下列文法产生的表达式包括赋值表达式。

$$S \to E$$
$$E \to E = E \mid E + E \mid (E) \mid \mathbf{id}$$

表达式的语义和 C 语言的一样，即 $b = c$ 是把 c 的值赋给 b 的赋值表达式，而且 $a = (b = c)$ 把 c 的值赋给 b，然后再赋给 a。构造一个语法制导定义，它检查赋值表达式的左部是否为左值。

4.12 文法如下：

$$S \to (L) \mid a$$
$$L \to L, S \mid S$$

(a) 写一个翻译方案，它输出每个 a 的嵌套深度。例如，对于句子 $(a,(a,a))$，输出的结果是 1 2 2。

(b) 写一个翻译方案，它打印出每个 a 在句子中是第几个字符。例如，当句子是 $(a,(a,(a,a),(a)))$ 时，打印的结果是 2 5 8 10 14。

4.13 语句的文法如下：

$$S \to \mathbf{id} := E \mid \mathbf{if}\ E\ \mathbf{then}\ S \mid \mathbf{while} E\ \mathbf{do}\ S \mid \mathbf{begin}\ S; S\ \mathbf{end} \mid \mathbf{break}$$

写一个翻译方案，其语义动作的作用是若发现 **break** 不是出现在循环语句中，则及时报告错误。

4.14 程序的文法如下：

$$P \to D$$
$$D \to D; D \mid \mathbf{id} : T \mid \mathbf{proc}\ \mathbf{id}; D; S$$

(a) 写一个语法制导定义，打印该程序一共声明了多少个 **id**。

(b) 写一个翻译方案，打印该程序每个变量 **id** 的嵌套深度。

4.15 下面是构造语法树的一个 S 属性定义。将这里的语义规则翻译成 LR 翻译器的栈操作代码段。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第179页 文件编号: 422624 页面编号: 31794538442 打印人: 8685 2026-03-09 10:23:01

$E \to E_1 + T$	$E.\text{nptr} = \text{mkNode}(\text{'+'}, E_1.\text{nptr}, T.\text{nptr})$
$E \to E_1 - T$	$E.\text{nptr} = \text{mkNode}(\text{'-'}, E_1.\text{nptr}, T.\text{nptr})$
$E \to T$	$E.\text{nptr} = T.\text{nptr}$
$T \to (E)$	$T.\text{nptr} = E.\text{nptr}$
$T \to \textbf{id}$	$T.\text{nptr} = \text{mkLeaf}(\textbf{id}, \textbf{id}.\text{entry})$
$T \to \textbf{num}$	$T.\text{nptr} = rmmkLeaf(\textbf{num}, \textbf{num}.\text{val})$

4.16 参照4.3.2 节，给出例 4.11 中对应非终结符 T 的翻译函数。

***4.17** 假定有 L 属性定义，它的基础文法是 LL(1) 的，或者是一个能解决其二义性并且为之构造预测解析器的文法，证明可以把继承属性和综合属性放在由预测分析表驱动的自上而下解析器的分析栈中。

***4.18** 证明在 LL(1) 文法的任何地方加上可区别的标记非终结符后，结果文法仍然是 LR(1) 的。

第 5 章

类 型 检 查

编译器必须检查源程序是否满足源语言在语法和语义两个方面的约定。这种检查称为**静态检查**（以区别在目标程序运行时的**动态检查**），它诊断和报告程序错误。静态检查的例子有：

（1）类型检查。如果使用未定义的变量、调用未定义的方法、运算符作用于不相容的运算对象等，编译器将报告错误。例如，数组变量和函数变量相加。

（2）控制流检查。对于任何使控制流离开某种程序构造的语句，程序中必须有该控制流可以转到的合法位置。例如，C 语言的 break 语句会将控制流移出包含其的最内层 while、for 或 switch 语句，如果没有这样的包围语句，程序会产生错误。

（3）唯一性检查。有些场合，对象必须只被定义一次。例如，在 C 语言中，重叠作用域中的局部变量必须唯一声明，switch 语句的分情形常量必须互不相同，枚举类型的元素不能重复。

（4）关联名字检查。有时，同一个名字必须出现两次或更多次。例如，Ada 语言中的过程 (procedure) 名需要出现在过程的开头和结尾。编译器必须检查这两个位置是否使用相同的名字。

本章集中于类型检查。上面的列举表明，大多数其他静态检查都是一些简单的工作，可以用第 4 章所讲的技术实现，其中有些工作可以并入编译器的其他部分。例如，在把名字的信息填入符号表时，可以检查该名字声明的唯一性。许多编译器把静态检查和中间代码生成组织在分析的同时完成。对于更复杂的构造，如 Ada 中的一些构造，在分析和中间代码生成之间加一遍类型检查可能会方便一些，如图 5.1 所示。

图 5.1 类型检查器的位置

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第181页 文件编号: 422624 页面编号: 31814538448 打印人: 8685 2026-03-09 10:23:01

5.1 类型在编程语言中的作用

本节非形式地介绍执行错误、安全语言、类型系统及其他相关概念，讨论所期望的类型系统的性质和优点，并概述怎样将类型系统形式化。

5.1.1 类型化语言和类型系统

1. 类型化语言

一个程序变量在程序执行期间可取的值有一定的限制，这个限制的依据就是该变量的类型。例如，布尔类型 (Boolean) 的变量 x 在程序运行时只能取布尔值。若 x 是布尔类型变量，那么表达式 not(x) 在程序每次运行时都有意义，因为 not 函数接受布尔类型的参数，返回布尔类型的值。若语言中变量都被给定了类型，运算都被定义了运算对象和运算结果所允许的类型，则该语言称为**类型化语言**（typed language）。对于类型化语言，如果对变量、表达式的类型指派是在编译时进行，则称为**静态类型语言**（statically typed language）；如果在运行时决定表达式的类型，则称为**动态类型语言**（dynamically typed language）。静态类型语言有 C\C++、Ada、ML、Java、Go、Rust 等，动态类型语言有 Lisp、Perl、PHP、Python、JavaScript、Ruby 等。

不限制变量的取值范围的语言被称为**未类型化语言**或**无类型语言**（untyped language），它们没有显式的类型系统（缺少静态类型），或者说仅有一个包含所有值的泛类型。在这类语言中，一个运算可以作用到任意运算对象，其结果可能是一个有意义的值、一个错误、一个异常或一个语言未定义的结果。纯 λ 演算是无类型语言的极端情况，它不产生任何错误：其唯一的操作是函数应用，并且由于所有值都是函数，该操作永远不会失败。而汇编语言也是典型的无类型语言，不过它会产生除数为零等执行错误。

2. 类型系统

类型系统（type system）是类型化语言的组成部分，它由一组**定型规则**（typing rule）构成，这些规则用于为程序中的各种语言构造（如变量、表达式、函数和模块等）指派类型。定型规则的一个非形式描述的例子：如果 M 和 N 都是整型表达式，那么 $M+N$ 也是整型表达式。

基于类型系统可以对程序进行**类型检查**（type checking），即根据定型规则来确定程序中各语法构造的类型，以拒绝那些有类型错误的程序，如含表达式 true+3 的程序。不能通过类型检查的程序不是合法的程序，严格地说，它们不能称为程序。能够通过类型检查的程序称为**良类型的**（well-typed）程序。完成这种检查的算法称为**类型检查器**（type checker）。

语言的类型化特性取决于其是否具备类型系统，与类型是否显式出现在程序语法中无光。如果语言中类型是语法的一部分，函数和变量的类型必须被显式声明，则这种语言是**显式类**

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第182页 文件编号: 422624 页面编号: 31824538446 打印人: 8685 2026-03-09 10:23:01

型化的（explicit typing），否则是**隐式类型化的**（implicit typing）。C、Java、C++、Go、Rust 等都是显式类型化语言。在主流编程语言中没有纯隐式类型化的，但一些静态类型语言（如函数式语言 ML 和 Haskell）允许在大型代码中省略类型标注，而动态类型语言（如 JavaScript、Python 等）则通过运行时类型检查确定变量的数据类型。这些语言的类型系统会自动地根据程序中表达式的使用和上下文来推断变量或表达式的类型，这种过程称为**类型推断**（type inference）。例如，ML 程序片段

```
fun add x y = x + y;
```

定义了一个名为 add 的函数，接受两个参数 x 和 y，并返回两个参数之和。ML 语言的编译器会根据对变量 x 和 y 执行加法运算，而推断 x 和 y 是数值类型。

在一个语言定义中，除了语法，若上下文有关的所有限制都体现在类型系统中，则良类型的程序就是合法程序。从本章开头所举的一些上下文有关约束可知，控制流和唯一性等上下文有关的约束通常都不在类型系统中表达。

5.1.2 执行错误和安全语言

1. 执行错误

程序运行时出现的错误称为**执行错误**（execution error）。有些执行错误，如非法指令、非法内存访问和除数为零错误，机器都会报告错误，因为它们会导致计算立即停止，操作系统也会提供错误位置和性质。然而，还有一些更难察觉的执行错误，如对合法地址的不适当访问（如数组越界），可能不会立即引发错误报告。另外，跳转到错误地址，可能暂时不会引起异常，但最终会导致不可预测的行为。因此，执行错误可以分成两类：一类会立即导致计算停止，即**可捕获的错误**（trapped error）；另一类在当时不会触发可捕获事件，但可能在后续引发问题，称为**不可捕获的错误**（untrapped error）。区别这两类错误有助于更好地理解错误的性质。

2. 安全语言

如果一个程序的运行不会引起不可捕获错误的出现，那么称该程序是**良行为的**（well-behaved）。对于编程语言来说，若其所有合法程序都是良行为的，则该语言称为**安全语言**（safe language）。保证语言安全性的通常做法是为语言设计一个类型系统，通过静态（编译时）检查、动态（运行时）检查或静态检查与动态检查混合的方式来拒绝一切有潜在不安全性的程序。

虽然安全性对程序而言是至关重要的性质，但是对类型化语言来说，设计一个正好排除不可捕获错误的类型系统是非常困难的。因此，安全语言的类型系统往往设计成在排除所有不可捕获错误的同时，也排除部分可捕获的错误。这样，对任何一种语言，都可以将所有可能的执行错误划分出一个子集，称为**禁止错误**（forbidden error）。在安全语言中，禁止错误应包含所有不可捕获的错误，以及一部分可捕获的错误。良行为的程序和安全语言也可以基

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第183页 文件编号: 422624 页面编号: 31834538444 打印人: 8685 2026-03-09 10:23:01

于禁止错误来定义。

设计类型系统的根本目的是防止程序运行时出现禁止错误，从而确保其运行是良行为的。如果良类型程序一定是良行为的，则称该语言是**类型可靠的**（type sound）。类型可靠的语言一定是安全语言。对于类型可靠的语言来说，良类型的程序有以下几个性质：

（1）不会出现不可捕获的错误（安全的）。

（2）不会出现已列入的禁止错误在运行时会被捕获。

（3）有可能出现其他可捕获的错误，避免这些错误是程序员的责任。

执行错误、良行为程序和安全语言是基于程序运行时特征来定义的，属于动态概念；而类型错误、良类型程序和类型化语言则是静态概念。类型可靠是联系两者的一个重要概念，类型检查的目的就是确保程序运行时的良行为。

5.1.3 不安全性及其检查

1. 语言的不安全性

现实中有些语言即便是类型化语言，也不能保证安全性，因为它们的禁止错误集合并没有包含所有不可捕获的错误。从历史上看，安全和不安全语言之间的选择可能与开发维护时间和执行时间之间的权衡有关。在语言设计的早期，安全性考虑不足是性能上的原因，最典型的是 C 语言。C 语言以处理低级构造的灵活性著称，这使得 C 语言编程灵活、程序运行高效，但牺牲了安全性。C 语言有很多不安全但广泛使用的特征，如指针算术运算、强制类型转换、union 类型。C 程序员十诫中的前五条是针对 C 语言的不安全所做的补充。

C 语言的一些不安全问题已在 C++ 中缓解，如通过类和对象封装数据和方法、通过访问修饰符控制成员的访问权限、引入异常处理机制等。更多的不安全问题在 Java 中已得到解决，如引入自动内存管理、数组越界检查、强类型系统、安全沙盒等。Modula-3、Rust 语言支持一些不安全特征，但是仅限于显式标记为不安全的模块中，并且不允许在安全模块中导入不安全的接口。例如，Rust 语言使用所有权及其借用和转移规则在编译时强制内存安全性，但也提供 unsafe 关键字允许切换到不安全的代码块，在 unsafe 代码块中允许使用解引用裸指针（raw pointer）、调用不安全的函数、访问或修改可变静态变量、实现不安全的 trait、访问 union 的字段等。

2. 安全性检查

为排除程序中的安全漏洞，需要进行大量调试。对于安全性不足的语言，可通过运行时检查来提高安全性。而对于类型化安全语言，即使对程序做了细致的静态分析，要想获得彻底的安全性也需要一点运行时检查。例如，数组越界不可能通过编译时检查完全排除。又如，Modula-3 的 TYPECASE、Java 的 instanceof 构造需要检测对象的运行时类型。这些仍然（虽不太恰当）被认为是静态检查，因为动态类型测试是在静态类型系统的基础上定义的。大多数无类型高级语言是安全的（如 Lisp），它们通过彻底的运行时详细检查来排除所有禁止错

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第184页 文件编号: 422624 页面编号: 31844538442 打印人: 8685 2026-03-09 10:23:01

误，如检查所有数组的访问是否越界、检查所有的除法操作，当出现禁止错误时，抛出可恢复的异常。

随着编程语言设计的发展，**渐进类型**（gradual typing）的概念应运而生（首次由 Jeremy Siek 和 Walid Taha 在 2006 年提出）。渐进类型系统允许开发者根据需要结合静态和动态类型，在提升安全性的同时保持一定的开发灵活性。通过这种方式，程序员能够在关键部分施加静态类型约束，而在不确定性较高的部分灵活使用动态类型，努力在安全性、灵活性和性能之间实现最佳平衡。目前，Python（自 3.5 版本引入类型提示）和 TypeScript 等都支持渐进类型。图 5.2 给出了使用渐进类型的 Python 程序示例，其中第 1 行的 ":int" 为类型提示。

```
1 def f(x, y : int):
2   return x+y
3
4 result = f(2, 4)
5 print(result)
```

图 5.2 渐进类型示例

运行时检查的代价有时是非常高的，然而为获得安全性所付出的代价是值得的。安全性使得程序出现执行错误就会中止，这能减少调试时间。安全性也能保证运行时存储空间结构的完整性，因而使无用存储单元收集（garbage collection，俗称垃圾收集）可以完成。而垃圾收集可以大大降低代码开发时间和代码的规模，尽管它需要额外的运行时代价。

3. 安全性日趋重要

随着信息技术的迅猛发展、互联网的广泛应用以及开源软件的盛行，关键基础设施如通信、能源、交通、电力、金融、卫生保健等系统，都越来越依赖于不是单一管理机构管辖的计算基础和资源（包括开源软件）。这些关键基础设施的稳定运行直接关系到社会、经济和国家的正常运转。因此，提高相关信息系统对恶意攻击和有漏洞软件的破坏的抵抗能力越来越重要。为增强关键基础设施及软件供应链的安全性，各国都采取了一系列措施，包括建立安全标准和法规（例如，2021 年我国颁布和实施的《关键信息基础设施安全保护条例》）、定期进行安全审查、实施监控和防护措施等，而采用更安全的编程语言编写软件也是提升安全性的一个重要考虑因素。

2008 年运营的代码托管平台 GitHub 在其 2018 年 Octoverse 年度报告中称，专注于线程安全和互操作性的静态类型语言越来越受欢迎，Kotlin（在 Java 虚拟机上运行的一种静态类型语言）、TypeScript（添加类型系统的 JavaScript）及 Rust（专注于安全的编译型语言）当年贡献者数快速增长，增速分别为 2.6 、1.9、1.7 倍；以灵活性著称的 JavaScript 受欢迎程度最高，而 TypeScript 跃升到第 7 位。在 2024 年 Octoverse 年度报告中，Python、JavaScript 和 TypeScript 是前三个受欢迎的语言。

5.1.4 类型化语言的优点

编程语言是否应该有类型仍然是一个颇具争议的话题，不同的编程语言设计哲学和使用场景可能导致不同的观点。从可维护性的角度看，类型可以提高代码的正确性和可维护性，而

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第185页 文件编号: 422624 页面编号: 31854538449 打印人: 8685 2026-03-09 10:23:01

无类型语言则允许更大的灵活性，减少定型规则的限制，但通常会降低代码质量和可维护性。从工程的角度看，类型化语言有下面一些优点：

（1）开发的实惠。类型系统可以较早地发现程序错误，如整数和串相加。若类型系统设计得很好，类型检查可以发现大部分常见的程序设计错误，剩下的错误也就相对容易调试。

针对大型软件开发，采用接口和模块有方法学上的优点，类型信息可以被组织到模块的接口中。程序员可以共同讨论接口，然后各自独立实现，从而最小化代码之间的相互依赖。这种方式使得局部范围内的代码重组不会影响全局，增强了系统的灵活性和可维护性。

程序中的类型信息还具有文档作用。程序员声明标识符和表达式的类型，这就提供了有关所期望的值的部分信息，有助于阅读理解程序。

（2）编译的实惠。程序模块可以分别独立编译，如 C/C++ 的程序文件、Modula-2 和 Ada 的模块。每个模块仅依赖于其他模块的接口。这样在大型系统的编译中只要接口保持稳定，改变一个模块不会导致其他模块的重新编译，从而提高了编译效率。

（3）运行的实惠。在编译时收集类型信息，使得编译器能够知道数据的大小，从而实现更有效的空间布局和访问方式，提升目标代码的运行效率。例如，像 C 语言中的结构体和 C++ 语言中的对象，其成员的偏移量可以基于类型信息静态确定。此外，准确的类型信息在编译时可以保证运行时的运算都作用于合适的对象且无需昂贵的运行时检查，从而提高了程序运行的效率。例如，在 ML 语言中，利用精确的类型信息可以删除在指针解引用（dereference）中的 nil 检查。

上面提到，类型信息具有文档作用，但是它与其他形式的程序标注不同。一般来说，程序行为的标注形式可以从非形式的注释一直到精确表述的形式规范。类型信息处在该范围的中间：它比程序注释更精确，又比形式规范更容易理解。

5.2 类型系统的描述语言

类型系统是一种逻辑系统，用来描述编程语言的类型与定型规则，即推理程序中各语言构造所具有的类型的推理规则。对类型系统的研究可以用形式化方法（formal method）或非形式方法。非形式描述通常难以准确而无歧义地表达语言的类型结构，导致同一语言的不同编译器实现间可能存在类型系统的差异（详见 5.4 节）。形式的类型系统会先给出符号化的定型断言，再用这样的定型断言描述定型规则和定型公理。形式的类型系统是类型化编程语言定义的一部分，它提供概念上的工具去评判语言定义中某些重要方面是否适当。依靠形式的类型系统，才有可能证明语言是否类型可靠。

类型系统独立于类型检查算法，这类似于形式文法描述编程语言的语法，而与具体的解析算法无关。将类型系统与类型检查算法分离是有益的：类型系统定义了语言的定型规则，而类型检查算法则由编译器实现。不同编译器可能使用不同的类型检查算法来实现相同的类型系统。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第186页 文件编号: 422624 页面编号: 31864538447 打印人: 8685 2026-03-09 10:23:01

设计类型系统时，常会考虑能否实现有效的类型检查算法。但某些类型系统可能并没有易于实现的类型检查算法，甚至完全没有类型检查算法。本章介绍的简单类型系统不涉及这种情况。

类型系统的基本概念实质上可用于所有的计算范型（paradigm），包括函数式语言、命令式语言和并行语言等。一些定型规则还可以不作修改地用于不同的范型。例如，对函数来说，不管是函数式语言还是命令式语言，不管是值调用还是换名调用，其基本定型规则都一样。

形式的类型系统是编程语言手册中非形式类型系统的一种数学描述。鉴于它的精确性，本章尽量采用形式方法来描述类型系统。

5.2.1 定型断言

与逻辑系统类似，类型系统的形式化从描述**断言**（assertion）开始。典型的断言具有形式：

$$\Gamma \vdash S \qquad S\text{的所有自由变量都声明在}\Gamma\text{中}$$

这里的 Γ 是**静态定型环境**（static typing environment），其形式是 n 个不同名字的变量 $x_1,\cdots,x_n$ 和它们所属类型 $T_1,\cdots,T_n$ 的有序表 $x_1:T_1,\cdots,x_n:T_n$。这相当于程序中的变量声明语句和编译器的符号表。在 Γ 中声明的这组变量 $x_1,\cdots,x_n$ 用 $\mathrm{dom}(\Gamma)$ 表示。变量个数为零的空环境用 $\varnothing$ 表示。S 的形式随断言形式的不同而不同，但是 S 的所有自由变量必须在 Γ 中声明。

断言有 3 种形式：环境断言、语法断言和定型断言。例如，下面这个环境断言直接声称定型环境 Γ 是**良形的**（well-formed），即它是结构良好的：

$$\Gamma \vdash \Diamond$$

后面将用环境推理规则来定义定型环境的语法。

语法断言用来表示类型表达式的良形性。语言构造的类型可以用“类型表达式”来表示，基本类型是类型表达式，把类型构造符应用到类型表达式而形成的表达式也是类型表达式。例如：

$$\Gamma \vdash \mathbf{nat}$$

表示在静态定型环境 Γ 下，**nat** 是一个类型表达式。后面将用语法推理规则来定义类型系统中的类型表达式的语法。在编程语言的语法中，类型表达式是用文法描述的（图 5.3）。但是在设计类型系统时，通常都用本节的方式来描述，其优点是能够用 Γ 来表示上下文有关的约束。

最重要的断言是**定型断言**，其形式是

$$\Gamma \vdash M : T$$

它声称在静态定型环境 Γ 下，M 具有类型 T（M 中的自由变量都出现在 Γ 中），其中 M 是表达式和语句这样的程序构造。例如：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第187页 文件编号: 422624 页面编号: 31874538445 打印人: 8685 2026-03-09 10:23:01

$\varnothing \vdash \textbf{true} : \textbf{boolean}$ 和 $x : \textbf{nat} \vdash x + 1 : \textbf{nat}$

第一个定型断言告知 **true** 是 **boolean** 类型的。因为 **true** 是常量，所以在空定型环境下也能判断它的类型。第二个定型断言告知，当 x 具有类型 **nat** 时，$x+1$ 的类型是 **nat**。后面将用形式的定型规则来定义程序构造的类型。

断言可能是**有效的**（valid）或**无效的**（invalid）。例如，$\Gamma \vdash \textbf{true} : \textbf{boolean}$ 是有效断言，因为 **true** 在布尔值集合中；显然，$\Gamma \vdash \textbf{true} : \textbf{nat}$ 是无效断言，因为 **true** 不在自然数集合中。

5.2.2 定型规则

作为一个逻辑系统，类型系统也有它的推理规则。它们是在一组已知有效的断言的基础上，声称某个断言的有效性。它们的一般形式是

$$\frac{\Gamma_1 \vdash S_1 \quad \cdots \quad \Gamma_n \vdash S_n}{\Gamma \vdash S}$$

在每条推理规则中，横线上面是一组称为**前提**的断言，横线下面是一个称为**结论**的断言。推理规则的含义是，当所有的前提都有效时，则结论也有效。前提的个数可以为零，这样的规则称为**公理**，即横线下的断言是有效的。

为便于引用，每条规则有一个规则名。其名称的第一个词由结论断言来确定。例如，形式为（Exp$\cdots$）的规则名用于这样的规则：其结论是一个表达式的定型断言。需要时，可以规定规则使用的限制条件，还可以在规则中使用一些缩写，它们都可以列在规则名或规则的右边，称为注释。因此，下面按照

（规则名）（注释） 推理规则 （注释）

的形式来介绍推理规则，其中注释可以为空。

针对 3 种不同的断言形式，有 3 类推理规则。例如，环境规则

$$\text{(Env } \varnothing\text{)} \qquad \frac{}{\varnothing \vdash \Diamond}$$

表示空环境是良形的，它是一个公理。

下面的语法规则本质上是说，在任何良形环境 Γ 下，**boolean** 是一个类型表达式：

$$\text{(Type Bool)} \qquad \frac{\Gamma \vdash \Diamond}{\Gamma \vdash \textbf{boolean}}$$

若推理规则的结论是定型断言，则称规则为**定型规则**。例如，下面的第一条规则意为，在任何良形环境下，任何一个自然数都是类型为 **nat** 的表达式。第二条规则意为，两个 **nat** 类型的表达式 M 和 N 相加所得表达式 $M+N$ 仍是一个 **nat** 类型表达式，而且 M 和 N 的环境 Γ 继续作为 $M+N$ 的环境。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第188页 文件编号: 422624 页面编号: 31884538443 打印人: 8685 2026-03-09 10:23:01

$$(\text{Val } n)(n \text{ 可以是} 0, 1, \cdots) \qquad \frac{\Gamma \vdash \Diamond}{\Gamma \vdash n : \mathbf{nat}}$$

$$(\text{Exp } +) \qquad \frac{\Gamma \vdash M : \mathbf{nat} \quad \Gamma \vdash N : \mathbf{nat}}{\Gamma \vdash M + N : \mathbf{nat}}$$

一个语言的定型规则（还有它的环境规则和语法规则）的集合构成该语言的（**形式的**）**类型系统**。从技术上讲，类型系统符合**形式证明系统**的一般框架，一组定型规则用于执行一步步的演绎推理，类型系统中的这种演绎推理用于对程序或程序构造定型。

5.2.3 类型检查和类型推断

在一个可靠的类型系统中，通过演绎推理得到的断言一定是该类型系统的**有效断言**。也就是说，一个有效断言是通过正确地应用类型规则可以得到的断言。形式的演绎推理在其他课程中学习过，在此不作介绍，而是在 5.5 节举例说明。

在一个类型系统中，如果存在一个类型 T 使得 $\Gamma \vdash E : T$ 是一个有效断言，那么 E 对环境 Γ 来说是良类型的，并且它的类型是 T。用语法制导的方式，根据上下文有关的定型规则来判定程序构造是否良类型的过程，通常被称作类型检查。例如，$\Gamma \vdash E : T$ 的任何推导必须根据 E 的语法结构，因此类型检查通常是语法制导的。

若函数形式为 $f(x : t) = E$，则函数首部规定了 x 的类型是 t。此时，对 E 的类型检查是基于已知 x 的类型这一前提进行的，这在命令式语言中很常见。但是，如果允许程序员忽略函数首部的参数类型，把该函数写成 $f(x) = E$ 的形式，就不清楚 x 是什么类型了。此时需要对 E 进行更复杂的分析，才能判断出 x 是什么类型。这种在类型信息不全情况下的定型判定称为**类型推断**（type inference）。区别类型推断和类型检查是有用的，但是精确区分它们是困难的。

若不存在 E 的任何定型推导，则称 E 不可定型，即 E 有类型错误或类型缺陷。

程序构造的类型检查或类型推断问题，与所使用的类型系统密切相关。具体来说，类型系统决定了算法的难易程度及效率。类型系统的实际效用依赖于是否能找到高效的类型检查或类型推断算法。

对于 C 和 Java 这样的显式类型化命令式语言，类型检查问题还算容易解决。C 和 C++ 的类型检查相对较弱，编译器会检查基本的类型一致性，但对诸如空指针引用等高级的类型错误不够严格；而 Java 的编译器则会进行较为严格的类型检查。C++11 及之后的版本引入了一些类型推断的特性，如 auto 关键字和模板的类型推断，这些特性使得编程更灵活，但它们并非完备的类型推断；Java 8 引入了 Lambda 表达式和用于流式处理集合数据的 Stream API，其中包含了一些类型推断的特性，如目标类型推断。

对于支持隐式类型化的语言，如 ML 系列函数式语言和 Rust 命令式语言，类型推断问题相当困难，其基本算法主要基于 Hindley-Milner 类型系统扩展并已被很好地理解和广泛地

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第189页 文件编号: 422624 页面编号: 31894538441 打印人: 8685 2026-03-09 10:23:01

使用，但是实际使用的算法相当复杂，一些问题仍然在研究中。ML 系列语言以其强大的类型推断而著称，可以推断出许多表达式的类型，减少了显式类型注解的需求。Rust 的类型检查非常强大，它通过借用系统和生命周期系统来保证内存安全性，捕获潜在的数据竞争和空指针引用等问题；Rust 的类型推断在很多情况下相当智能，但有时需要显式注解以帮助编译器更好地理解代码。

5.3 一个简单类型检查器的规范

本节描述一个简单语言的类型系统和语法制导的类型检查器。在此语言中，每个标识符的类型必须在该标识符使用前先声明。这个类型检查器可以是翻译方案，或基于树访问，它根据子表达式的类型来确定表达式的类型。该类型检查器能够处理数组、指针、语句和函数。

5.3.1 一个简单的语言

图 5.3 是该语言的文法，其中 P 表示程序，它由声明 D 和随后的语句 S 组成。为集中于类型检查器的介绍，这里采用了一个比较简洁的二义文法，并且允许直接声明函数标识符的类型而无须给出具体的函数定义。

$$
\begin{aligned}
&P \to D\ ;S\\
&D \to D\ ;D \mid \mathbf{id} : T\\
&T \to \mathbf{boolean} \mid \mathbf{integer} \mid \mathbf{array[num]\ of}\ T \mid\ *T \mid\ T' \to\ 'T\\
&S \to \mathbf{id} = E \mid \mathbf{if}\ E\ \mathbf{then}\ S \mid \mathbf{while}\ E\ \mathbf{do}\ S \mid S\ ;S\\
&E \to\ \ \mathbf{truth} \mid \mathbf{num} \mid \mathbf{id} \mid E\ \mathbf{mod}\ E \mid E[E] \mid\ *E\ \mid E(E)
\end{aligned}
$$

图 5.3 源语言文法

图 5.3 中的第 3 行是描述类型的产生式。经从 T 开始的语法推导得到的终结符号串都是类型表达式，它们可以像算术表达式那样用树来描述。每个类型表达式都是该语言的一个类型，因此经从 T 开始的推导，可以得到该语言所有的类型。在 5.3.2 小节的类型系统中，类型表达式用下面更接近抽象的语法：

$$T \to \text{boolean} \mid \text{integer} \mid \text{array}(N, T) \mid \text{pointer}(T) \mid T' \to\ 'T \mid \text{void}$$

其中，N 表示自然数，void 在类型系统中作为语句的类型。在 5.3.3 小节的类型检查器中，还需要增加 type_error，作为出现类型错误的程序构造的类型。

有关图 5.3 文法的其他一些解释，穿插在下面各小节。由该文法产生的程序实例有

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第190页 文件编号: 422624 页面编号: 31904538449 打印人: 8685 2026-03-09 10:23:01


```
i : integer;
j : integer;
j = i mod 2000
```

在该程序段中，选择 mod 作为二元运算符的代表。

5.3.2 类型系统

事实上，语言的设计和其类型系统的设计是同时进行并相互影响的。类型系统的设计包括三部分：环境规则、语法规则和定型规则。

1. 环境规则

表达式类型的确定与环境有关，因为表达式中可能有变量。两条环境规则与图 5.3 所示文法中的类型声明有关，第一条规则意为空环境 $\varnothing$ 是良形的，第二条规则意为程序中一个新的变量声明引起在环境中增加一个变量到类型的映射。

(Env $\varnothing$) $$\frac{}{\varnothing \vdash \Diamond}$$

(Decl Var) $$\frac{\Gamma \vdash T \quad \mathbf{id} \notin \mathrm{dom}(\Gamma)}{\Gamma, \mathbf{id} : T \vdash \Diamond}$$

2. 语法规则

类型系统设计的第二部分是设计在定型规则中使用的类型表达式的语法，即良形类型表达式的定义。5.2.2 小节已经提到，这里采用类型表达式的抽象语法，它与 5.2.3 小节类型检查算法中所用类型表达式的语法一致。

下面 3 条语法规则表示，在任何良形环境下，boolean、integer 和 void 都是类型表达式。其中，void 类型本质上表示语句是没有类型的语法构造。

(Type Bool) $$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \mathrm{boolean}}$$

(Type Int) $$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \mathrm{integer}}$$

(Type Void) $$\frac{\Gamma \vdash \Diamond}{\Gamma \vdash \mathrm{void}}$$

下面 3 条语法规则说明构造类型的语法。pointer、array 和 $\rightarrow$ 都是类型构造符。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第191页 文件编号: 422624 页面编号: 31914538447 打印人: 8685 2026-03-09 10:23:01

$$\text{(Type Ref)}(T \neq \text{void}) \qquad \frac{\Gamma \vdash T}{\Gamma \vdash \text{pointer}(T)}$$

$$\text{(Type Array)}(T \neq \text{void}) \qquad \frac{\Gamma \vdash T \quad \Gamma \vdash N : \text{integer}}{\Gamma \vdash \text{array}(N, T)} \qquad (N > 0)$$

$$\text{(Type Fun)}(T_1, T_2 \neq \text{void}) \qquad \frac{\Gamma \vdash T_1 \quad \Gamma \vdash T_2}{\Gamma \vdash T_1 \rightarrow T_2}$$

规则（Type Ref）说明，如果 T 是类型表达式，那么类型表达式 pointer(T) 表示指向类型为 T 的对象的指针类型。在图 5.3 的具体语法中，类型声明 $*T$ 中的前缀运算符 $*$ 用于构造指针类型，表达式 $*E$ 中的前缀运算符 $*$ 用于解引用。

规则（Type Array）说明，如果 T 是类型表达式，那么 array(N, T) 表示元素类型为 T 的数组类型。所有数组的下标都假定从 0 开始，数组的下标集合是 $0..N-1$。

规则（Type Fun）说明，类型表达式 $T_1 \rightarrow T_2$ 表示定义域类型为 T_1 和值域类型为 T_2 的函数类型。为简单起见，图 5.3 的语言只考虑了一元函数。

3. 定型规则

类型系统设计的第三部分是设计定型规则，也就是给各种形式的表达式和语句指派类型。这是类型系统最重要的部分，因此下面详细给出针对各种形式的表达式和语句的定型规则，首先针对表达式的 7 种语法形式，有 7 条定型规则用来确定表达式的类型。

$$\text{(Exp Truth)} \qquad \frac{\Gamma \vdash \Diamond}{\Gamma \vdash \text{truth} : \text{boolean}}$$

$$\text{(Exp Num)} \qquad \frac{\Gamma \vdash \Diamond}{\Gamma \vdash \text{num} : \text{integer}}$$

$$\text{(Exp Id)} \qquad \frac{\Gamma_1, \mathbf{id} : T, \Gamma_2 \vdash \Diamond}{\Gamma_1, \mathbf{id} : T, \Gamma_2 \vdash \mathbf{id} : T}$$

$$\text{(Exp Mod)} \qquad \frac{\Gamma \vdash E_1 : \text{integer} \quad \Gamma \vdash E_2 : \text{integer}}{\Gamma \vdash E_1 \bmod E_2 : \text{integer}}$$

$$\text{(Exp Index)} \qquad \frac{\Gamma \vdash E_1 : \text{array}(N, T) \quad \Gamma \vdash E_2 : \text{integer}}{\Gamma \vdash E_1[E_2] : T} \quad (0 \leqslant E_2.\text{val} \leqslant N-1)$$

$$\text{(Exp Deref)} \qquad \frac{\Gamma \vdash E : \text{pointer}(T)}{\Gamma \vdash *E : T}$$

$$\text{(Exp FunCall)} \qquad \frac{\Gamma \vdash E_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash E_2 : T_1}{\Gamma \vdash E_1(E_2) : T_2}$$

上述规则（Exp Id）表示，如果 **id** 出现在环境 Γ 中，则 **id** 的类型就是它在 Γ 中的类型。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第192页 文件编号: 422624 页面编号: 31924538445 打印人: 8685 2026-03-09 10:23:01

通常认为语句是没有类型的。在类型系统的设计中，可以给它们以特殊的基本类型 *void* 并使用下面这些定型规则，这样便于判断整个程序可执行的语句部分是否有类型错误。为简单起见，这里没有设计把声明部分和语句部分结合起来（整个程序）的定型规则。

$$\text{(State Assign)}(T = \text{boolean or } T = \text{integer}) \qquad \frac{\Gamma \vdash \mathbf{id} : T \quad \Gamma \vdash E : T}{\Gamma \vdash \mathbf{id} = E : \text{void}}$$

$$\text{(State If)} \qquad \frac{\Gamma \vdash E : \text{boolean} \quad \Gamma \vdash S : \text{void}}{\Gamma \vdash \text{if } E \text{ then } S : \text{void}}$$

$$\text{(State While)} \qquad \frac{\Gamma \vdash E : \text{boolean} \quad \Gamma \vdash S : \text{void}}{\Gamma \vdash \text{while } E \text{ do } S : \text{void}}$$

$$\text{(State Seq)} \qquad \frac{\Gamma \vdash S_1 : \text{void} \quad \Gamma \vdash S_2 : \text{void}}{\Gamma \vdash S_1; S_2 : \text{void}}$$

5.3.3 类型检查

根据 5.3.2 小节的类型系统，设计类型检查器是一个相对直接的过程。类型系统中的定型环境 Γ 的信息存入编译器的符号表，每条定型规则在类型检查器中都有对应的类型检查处理。同时，类型检查器需要具备处理程序中类型错误的能力：不仅要能报告错误的性质和位置，还要能从错误中恢复，以便检查剩余输入中的各种错误。因此，在类型检查器中增加一个特殊的类型标识 type_error，表示存在类型错误的语法构造的类型，但是当前未讨论如何报告类型错误。

在下面的类型检查方案中，引入两个与符号表有关的辅助函数：addtype 函数将 **id** 的类型添加到符号表中，若出现重复定义则报错；lookup 函数则从符号表中查找 **id** 的类型，若未找到则报错，返回 type_error。这里的类型表达式和类型系统中的形式一致，当然，也可以另选一种表示形式。在实现上，类型表达式可以用记录类型来表示，第一元是类型所属的类别，第二元是该类别的类型的其他信息。例如，array(N, T) 表示为元组 (Array, (N, T))，pointer(T) 表示为元组 (Pointer, T)，Array 和 Pointer 为符号常量。

类型检查器可以通过边解析边进行类型检查，也可以通过遍历语法树进行类型检查。基于解析器的翻译方案要求类型检查器的语义动作只能依赖已解析程序部分的信息，而不能依赖尚未解析部分所拥有的信息。这种方式是早期编译器中的做法，那时硬件的计算和存储能力还比较弱。现代编译器通常选择更易于开发和维护的基于树访问的类型检查方法。这种方式在生成语法树后再进行检查，即便程序中某位置的类型检查依赖该位置之后的信息，也能从语法树中获取。下面先介绍类型检查的翻译方案，再简要介绍基于树访问的类型检查方案。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第193页 文件编号: 422624 页面编号: 31934538443 打印人: 8685 2026-03-09 10:23:01

1. 类型检查的边解析边翻译

假设为文法符号引入 type 属性保存其类型。首先看声明语句的语义动作。在类型构造 T 的语义动作中，主要构造类型表达式并将其保存在 T 的综合属性 type 中。而在声明语句 D 的第二种语法结构 **id** : T 的语义动作中，会调用 addtype 函数将 **id** 的类型加入符号表中。

$D \to D ; D$	
$D \to \textbf{id} : T$	$\{\text{addtype}(\textbf{id}.\text{entry}, T.\text{type}); \}$
$T \to \textbf{boolean}$	$\{T.\text{type} = \text{boolean}; \}$
$T \to \textbf{integer}$	$\{T.\text{type} = \text{integer}; \}$
$T \to *T_1$	$\{T.\text{type} = \text{pointer}(T_1.\text{type}); \}$
$T \to \textbf{array}[\textbf{num}]\ \textbf{of}\ T_1$	$\{T.\text{type} = \text{array}(\textbf{num}.\text{val}, T_1.\text{type}); \}$
$T \to T_1\ ' \to\ ' T_2$	$\{T.\text{type} = T_1.\text{type} \to T_2.\text{type}; \}$

在表达式的语义动作中，E 的综合属性 type 给出了 E 产生的表达式的类型表达式。如果碰到类型错误时，该表达式的类型表达式是 type_error。如果表达式 E 的某个子表达式的类型是 type_error，那么 type_error 从这个子构造一直传到 E。对于变量引用 **id**，会调用 lookup 函数查符号表，获取 **id** 的类型。这条语义规则实现类型系统中的定型规则 (Exp Id)。

$E \to \textbf{truth}$	$\{E.\text{type} = \text{boolean}; \}$
$E \to \textbf{num}$	$\{E.\text{type} = \text{integer}; \}$
$E \to \textbf{id}$	$\{E.\text{type} = \text{lookup}(\textbf{id}.\text{entry}); \}$
$E \to E_1\ \textbf{mod}\ E_2$	$\{\textbf{if}\ (E_1.\text{type} == \text{integer}\ \&\&\ E_2.\text{type} == \text{integer})$ $\quad E.\text{type} = \text{integer};$ $\textbf{else}\ E.\text{type} = \text{type_error}; \}$
$E \to E_1\,[E_2]$	$\{\textbf{if}\ (E_2.\text{type} == \text{integer}\ \&\&\ E_1.\text{type} == \text{array}(s,t))$ $\quad E.\text{type} = t;$ $\textbf{else}\ E.\text{type} = \text{type_error}; \}$
$E \to *E_1$	$\{\textbf{if}\ (E_1.\text{type} == \text{pointer}(t))$ $\quad E.\text{type} = t;$ $\textbf{else}\ E.\text{type} = \text{type_error}; \}$
$E \to E_1(E_2)$	$\{\textbf{if}\ (E_2.\text{type} == s\ \&\&\ E_1.\text{type} == s \to t)$ $\quad E.\text{type} = t;$ $\textbf{else}\ E.\text{type} = \text{type_error}; \}$

对于数组引用 $E_1[E_2]$，下标表达式必须为 integer 类型。结果类型是从 E_1 的类型 array(s, t) 中得到的元素类型 t。这里 s 未采用数组的下标集合，因为数组引用的越界检查在运行时

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第194页 文件编号: 422624 页面编号: 31944538441 打印人: 8685 2026-03-09 10:23:01

完成。

对于函数调用 $E_1(E_2)$，E_1 的类型必须是 E_2 的类型 s 到某个值域类型 t 的函数类型 $s \to t$，$E_1(E_2)$ 的类型是 t。这里只考虑一元函数，后面介绍积类型后，可把类型检查推广到多元函数。

从下面各种语句的语义规则可以看出，在没有类型错误时，整个程序可执行的语句部分的类型是 void，否则是 type_error。

$$
\begin{array}{ll}
S \to \textbf{id} \ = E & \{\textbf{if}\ (\textbf{id}.\text{type} == E.\text{type}\ \&\&\ E.\text{type} \in \{\text{boolean}, \text{integer}\}) \\
 & \quad S.\text{type} = \text{void}; \\
 & \ \textbf{else}\ S.\text{type} = \text{type_error};\} \\
S \to \textbf{if}\ E\ \textbf{then}\ S_1 & \{\textbf{if}\ (E.\text{type} == \text{boolean}) \quad S.\text{type} = S_1.\text{type}; \\
 & \ \textbf{else}\ S.\text{type} = \text{type_error};\} \\
S \to \textbf{while}\ E\ \textbf{do}\ S_1 & \{\textbf{if}\ (E.\text{type} == \text{boolean}) \quad S.\text{type} = S_1.\text{type}; \\
 & \ \textbf{else}\ S.\text{type} = \text{type_error};\} \\
E \to S_1; S_2 & \{\textbf{if}\ (S_1.\text{type} == \text{void}\ \&\&\ S_2.\text{type} == \text{void}) \\
 & \quad S.\text{type} = \text{void}; \\
 & \ \textbf{else}\ S.\text{type} = \text{type_error};\}
\end{array}
$$

2. 基于树访问的类型检查

在类型检查的树访问方案中，可以为表达式等树结点引入 type 属性保存它的类型。下面以声明语句 D 为例，来说明树访问方案需要注意的地方。

$$
\begin{array}{lll}
D \to D; D & & //\text{D1} \\
D \to \textbf{id} : T & \{\text{addtype}(\textbf{id}.\text{entry}, T.\text{type});\} & //\text{D2}
\end{array}
$$

声明语句 D 有两种语法结构，分别是表示多个声明的 D1 和单个 **id** 声明的 D2。针对 D1，需要考虑如何在语法树中表示多个声明，以及如何访问这些声明。一种方法是引入声明列表结点类，其中用线性表（如 vector 容器类）保存关联的多个声明，并提供迭代器 iterator 对表中元素（每个声明实例）迭代访问。类型检查器在处理这类结点时，只需调用迭代器依次获取每个声明的信息处理即可。针对 D2，可以在对应的语法树结点访问结束时增加对 addtype 的调用。

图 5.4(a) 和图 5.4(b) 分别示意了基于树访问的类型检查器的 C++ 代码片段和本文采用的简记方法伪代码。简记方法不会详细描述每个语句，而是以简化的方式概括重要的步骤和逻辑。在图 5.4(b) 的简记表示中，**visitD1** 是对 D1 结构的操作示例，它负责遍历 D1 列表中的所有声明项，对每个声明子结点 child，迭代调用 visitD2 方法来处理每个变量声明；**exitD2** 在遍历 D2 结构完成时被调用，此时会调用 addtype 方法将变量类型添加到符号表中。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第195页 文件编号: 422624 页面编号: 31954538448 打印人: 8685 2026-03-09 10:23:01


```
class ASTChecker : public ASTBaseVisitor {
public:
    SymTab &symTab;
    // 构造函数：接受符号表的引用
   ASTChecker(SymTab &symTab) : symTab(symTab) {}
    // 访问 D1 结点：遍历其子结点列表
    void visitD1(D1Context &ctx) override {
        // 获取 D1 子结点列表，并使用迭代器遍历
        for (auto childD : ctx.D1()) {
            visit(childD);
        }
    }
    // 访问 D2 结点：添加类型到符号表
    void visitD2(D2Context &ctx) override {
        std::string idName = ctx.id()->getText();
        std::string typeName = ctx.T()->getText();
        // 将 id 的类型 T 更新到符号表
        symTab.addType(idName, typeName);
    }
};
```

(a) C++ 代码片段

visitD1(D1) : $//D \to D;D$
    foreach(child : $D1$.list)
        visitD2(child);

exitD2(D2) : $//D \to \mathbf{id} : T$
    addtype(**id**.entry, T.type);

(b) 本文采用的简记方法伪代码

图 5.4 树访问类型检查代码示例

对于其他语言构造，类型检查的处理与 D2 类似，将翻译方案中对应的语义动作转变成对应语法树结点访问结束时安插的类型检查处理代码。

5.3.4 类型转换

考虑表达式 $x+i$, 其中 x 是实型，i 是整型。因为在计算机中实数和整数有不同的表示，并且对实数和整数有不同的机器指令，所以编译器可能首先对其中一个运算对象进行类型转换，以保证相加的两个对象有同样的类型。

语言的类型系统会指出哪些不同类型的对象可以在一起运算，语言的实现会考虑是否要进行类型转换。例如，当将整数表达式赋给实型变量时，应该把赋值号右边对象的类型转换成左边对象的类型。在表达式中，通常是把整数转换成实数，然后在一对实型对象上进行实数运算。编译器的类型检查器会在源程序的中间表示中插入这些转换操作。例如，$x+i$ 的后缀式可以是

$$x\ i\ \mathbf{inttoreal}\ \text{real+}$$

这里，首先由 **inttoreal** 运算符把 i 从整数转换成实数，然后由 real+ 将该转换结果和 x 进行实数加。

如果从一个类型转换到另一类型可以由编译器自动完成，这样的转换称为**隐式转换**。例

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第196页 文件编号: 422624 页面编号: 31964538446 打印人: 8685 2026-03-09 10:23:01

如，在 C 语言的算术表达式中，编译器把 ASCII 字符强制（隐式转换）为 0 ～ 127 之间的整数。在许多语言中，要求隐式转换原则上不丢失信息。例如，整数可以转换为实数，但反过来则不行。不过，当实数和整数用同样多比特表示时，还是有可能丢失信息的。

如果类型转换必须由程序员写出，那么这种转换称为**显式转换**。Ada 的所有转换都是显式转换。显式转换实际是一种函数调用，因此它们不会给类型检查器提出新问题。

$E \to \textbf{num}$ {E.type = integer;}

$E \to \textbf{num.num}$ {E.type = real;}

$E \to \textbf{id}$ {E.type = lookup(**id**.entry);}

$E \to E_1$ **op** E_2 {**if** (E_1.type == integer && E_2.type == integer)
E.type = integer;
else if (E_1.type == integer && E_2.type == real)
E.type = real;
else if (E_1.type == real && E_2.type == integer)
E.type = real;
else if (E_1.type == real && E_2.type == real)
E.type = real;
else E.type = type_error;}

图 5.5 从整型到实型的类型转换

【例 5.1】 考虑把算术运算符 **op** 作用于常数和标识符形成的表达式，如图 5.5 所示的文法那样。假定有实数和整数两个类型，必要时整数转换成实数。非终结符 E 的属性 type 可以是 integer 或 real，类型检查在图 5.5 中给出。

常量的隐式转换可以在编译时完成，并且常常可以使目标程序的运行时间明显缩短。

5.4 类型表达式的等价

5.3 节的类型检查有许多 “如果两个类型表达式相同，则返回某个类型，否则返回 type_error” 的情况。此处的 “相同” 指的是 5.4.1 小节定义的结构等价。

在有些编程语言中，类型表达式可以被命名并用于后续的类型表达式中，这会引发二义性：这些名字是否代表独立的类型（不同名字代表不同类型），还是其他类型表达式的缩写（不同名字有可能代表相同的类型）。这引出了与结构等价不同的名字等价概念。

本节基于类型表达式的图表示讨论类型表达式的等价性。若将名字看成类型表达式的缩写，递归定义的类型会导致图中出现环。需要注意的是，类型名字的引入只是语法层面的约定，并不扩展所表达的类型范围。另外，本节的类型表达式未包含类型变量，即不涉及多态类型。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第197页 文件编号: 422624 页面编号: 31974538444 打印人: 8685 2026-03-09 10:23:01

5.4.1 类型表达式的结构等价

如果类型表达式仅由类型构造符作用于基本类型组成，则可用**结构等价**来定义两个类型表达式是否相等。具体来说，两个表达式等价的条件是它们要么是同样的基本类型，要么是同样的类型构造符作用于结构等价的类型。也就是说，两个类型表达式结构等价，当且仅当它们完全相同。例如，类型表达式 integer 等价于 integer，因为它们是同一基本类型。类似地，pointer(integer) 等价于 pointer(integer)，因为它们把同一构造符 pointer 作用于等价的类型。

在实际使用中，结构等价的概念常会根据源语言的类型检查规则进行调整。例如，当数组作为参数传递时，可能不希望数组的界作为类型的一部分。

图 5.6 是测试结构等价的算法，假定仅有的类型构造符是数组、积、指针和函数。为简单起见，类型表达式（而不是它的树形表示）直接出现在算法中。这个算法递归地比较类型表达式的结构而不检查环，所以它能用于类型表达式的树表示。

```
1   boolean sequiv (typeExpression s, typeExpression t){
2       if (s 和 t 是相同的基本类型)
3           return true;
4       else if (s == array(s1, s2) && t == array(t1, t2))
5           return sequiv(s1, t1) && sequiv(s2, t2);
6       else if (s == s1 × s2 && t == t1 × t2)
7           return sequiv(s1, t1) && sequiv(s2, t2);
8       else if (s == pointer(s1) && t == pointer(t1))
9           return sequiv(s1, t1);
10      else if (s == s1 → s2 && t == t1 → t2)
11          return sequiv(s1, t1) && sequiv(s2, t2);
12      else return false;
13  }
```

图 5.6 两个类型表达式 s 和 t 的结构等价测试

如果图 5.6 第 4 行和第 5 行的数组等价测试重写为

```
else if (s == array(s1, s2) && t == array(t1, t2))
    return sequiv(s2, t2);
```

那么表示忽略对数组的界的检查。

5.4.2 类型表达式的名字等价

在某些语言中，类型可以命名。例如，在 C 程序段

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第198页 文件编号: 422624 页面编号: 31984538442 打印人: 8685 2026-03-09 10:23:01


```
typedef cell *link;
link next;
link last;                                    (5.1)
cell *p;
cell *q, *r;
```

中，标识符 link 声明为类型 cell * 的一个名字。由此引出的问题是，变量 next、last、p、q 和 r 都有相同的类型吗?

为了讨论这个问题，允许给类型表达式命名，并且允许这些名字出现在类型表达式中基本类型可以出现的地方。例如，如果 cell 是类型表达式的名字，那么 pointer(cell) 是类型表达式。暂且假定没有带环的类型表达式定义。

当名字允许出现在类型表达式中时，就引起了类型等价的两种不同概念，它们取决于如何看待名字。按结构等价的观点，先把两个类型表达式中所有的类型名字用它们定义的类型表达式代换，如果完成代换后的两个类型表达式结构等价，则认为原来的两个类型表达式结构等价。按**名字等价**的观点，不同的类型名看成不同的类型，因此两个类型表达式的名字等价，当且仅当这两个类型表达式不做名字代换就结构等价。

【例 5.2】 下面列出了和声明 (5.1) 的各变量联系的类型表达式。

变量	类型表达式
next	link
last	link
p	pointer (cell)
q	pointer (cell)
r	pointer (cell)

在名字等价下，变量 next 和 last 有相同的类型，因为它们有同样的类型表达式。变量 p、q 和 r 也有相同的类型，但是 p 和 next 的类型不相同，因为它们的类型表达式不同。在结构等价下，这 5 个变量都有相同的类型，因为 link 是类型表达式 pointer(cell) 的名字。

在不同语言中，标识符和类型通过声明联系的规则是不同的，在解释这些规则时，结构等价和名字等价是两个有用的概念。

5.4.3 记录类型

记录类型从某种意义上来说是它各域类型的积，记录和积之间的主要区别是记录的域被命名。把记录构造符 record 作用于域名及其类型的二元组序列，就形成记录类型表达式，由此可以写出与记录有关的定型规则如下：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第199页 文件编号: 422624 页面编号: 31994538440 打印人: 8685 2026-03-09 10:23:01

$$\text{(Type Record)}(l_i\text{是有区别的}) \qquad \frac{\Gamma \vdash T_1 \quad \cdots \quad \Gamma \vdash T_n}{\Gamma \vdash \text{record}(l_1 : T_1, \cdots, l_n : T_n)}$$

$$\text{(Val Record)}(l_i\text{是有区别的}) \qquad \frac{\Gamma \vdash M_1 : T_1 \quad \cdots \quad \Gamma \vdash M_n : T_n}{\Gamma \vdash \text{record}(l_1 = M_1, \cdots, l_n = M_n) : \text{record}(l_1 : T_1, \cdots, l_n : T_n)}$$

$$\text{(Val Record Select)}(1 \leqslant i \leqslant n) \qquad \frac{\Gamma \vdash M : \text{record}(l_1 : T_1, \cdots, l_n : T_n)}{\Gamma \vdash M.l_i : T_i}$$

根据这些定型规则写一个翻译方案，以完成记录类型和记录域访问的类型检查是很容易的。

例如，C 程序段

```
typedef struct {
    int address;
    char lexeme [15 ];
} row;
```

定义类型名 row 代表类型表达式

```
record(address : int, lexeme : array(15, char))
```

5.4.4 类型表示中的环

链表和树这样的基本数据结构经常是递归定义的。例如，一个链表可以是空表或者是由一个表元和一个链表指针组成的。这样的数据结构通常用记录实现，这种记录包含指向相同类型的记录的指针。在定义这样的记录类型时，类型名起着重要的作用。

为更好地分析递归类型和等价性，可以使用**类型图**来表示类型表达式。在类型图中，结点表示类型，边表示类型之间的引用或构造关系，从而直观展示出类型的组成及递归结构。

以链表为例，假设链表的每个结点由整型信息和指向下一结点的指针组成。在 C 语言中，这种链表结点可以用图 5.7(a) 所示的 struct cell 声明。由于 struct cell 引用了自己，类型图在处理这种递归类型时会形成如图 5.7(b) 所示的环。通过名字等价，可以简化为图 5.7(c)，即不必继续递归展开该结构，只需识别 struct cell 这一类型名，将其视为独立类型，从而避免重复解析。

【例 5.3】 C 语言对除记录（结构体）、共用体以外的所有类型使用结构等价，而对记录类型和共同体类型使用的是名字等价，以避免类型图中的环，因为环的出现会使结构等价的判断很复杂。当碰到记录构造符时，结构等价的测试停止，被比较的类型或者由于它们是同样的命名记录类型而等价，或者它们不等价。因此在效果上，C 语言使用的是图 5.7(c) 所示的无环表示形式。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第200页 文件编号: 422624 页面编号: 32004538446 打印人: 8685 2026-03-09 10:23:01

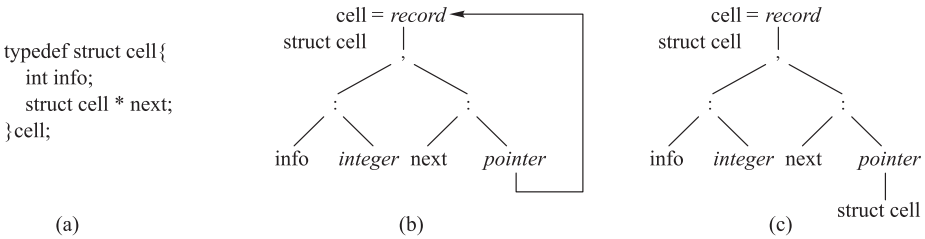


图 5.7 递归定义的类型名

*5.5 多态函数

普通的函数要求变元有唯一的类型，即函数体中的语句只能在变元类型固定的情况下执行。多态函数允许参数变元有不同的类型，使得函数能够适应多种数据类型，提高了代码的灵活性与可重用性。术语“多态”包括参数化多态和 Ad-hoc 多态。**参数化多态**能够以相同的方式处理多种类型的数据；而 **Ad-hoc 多态**允许同一函数或运算符在不同类型上表现出不同的行为。

内部定义的运算符，如数组索引、函数应用和通过指针的间接访问等通常都是多态的，因为它们并没有限于特定类型的数组、函数和指针。例如，C 语言的参考手册关于指针 & 的论述是：“一元 & 运算符的运算对象应是函数指示符、[ ] 或一元 * 运算符的结果、或代表非位字段且未使用 register 存储类修饰符的对象的左值”，“如果运算对象的类型是‘···’，那么结果类型是指向‘···’的指针”。因为有多种类型都可以代替“···”，所以 C 的运算符 & 是多态的。

本节讨论为支持多态函数的语言设计类型检查器时面临的问题。为处理多态性，需要拓广类型表达式集合，引入类型变量。类型变量的引入产生了一些和类型表达式等价有关的算法问题。

5.5.1 为什么要使用多态函数

多态函数属于参数化多态，其吸引力在于能够实现那些操作于某种数据结构，而无须考虑其成员类型的算法。例如，用多态函数很容易写出求链表长度的程序而不必管表元的类型。

像 C 这样的语言，函数参数的类型必须显式指定，所以确定整数链表长度的函数不能用于实数链表。求整数链表长度的 C 程序如图 5.8 所示。函数 length() 顺着链表的 next 链直至 NULL 为止。虽然函数实现不依赖表元信息的类型，但是 C 语言要求必须显式指明表元结构中 info 域的类型。

在支持多态函数的语言（如 ML）中，函数 length() 可以编写成适用于任何类型的表，如图 5.9 所示。关键字 fun 表示 length 是函数。函数 null() 和 tl() 是预定义的，null() 测试

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第201页 文件编号: 422624 页面编号: 32014538444 打印人: 8685 2026-03-09 10:23:01

表是否为空，tl() 返回除去第一个表元后剩下的部分。用图 5.9 的定义，函数 length() 的下列应用都产生值 3：

```
length (['sun', 'mon', 'tue']);
length ([10, 9, 8 ]);
```

在第一个应用中，length 作用于串表；在第二个应用中，它作用于整数表。

```
typedef cell * link;
typedef struct {
        int info;
        link next;
} cell;
```

图 5.8 求表长的 C 程序

```
int length(link lptr) {
    int len;
    len = 0;
    while (lptr != NULL){
        len = len + 1;
        lptr = lptr->next;
    }
    return len;
}
```

```
fun length (lptr) =
    if null (lptr) then 0
    else length (tl (lptr)) + 1;
```

图 5.9 求表长的 ML 程序

要想完成多态函数的类型检查，首先要能为这些函数提供适当的类型描述。以 length 函数为例，其参数可以是任意类型的表，因此需要用类型变量来表示这种灵活性。通过引入类型变量及全称量词 $\forall$，可以更准确地描述 length 函数的类型，它可以写成 $\forall\alpha.\text{list}(\alpha) \rightarrow \text{integer}$。

5.5.2 类型变量

下面允许类型表达式包含变量，变量的值是类型表达式。本小节的其余部分用希腊字母 α 和 β 等作为类型变量，用于类型表达式中。

类型变量除了可出现在多态函数的类型表达式中，还可以用在其他方面，如使用类型变量便于讨论未知类型。在不要求标识符的声明先于使用的语言中，类型变量的一个重要运用是检查标识符使用的一致性。类型变量代表没有声明的标识符的类型，查看程序可以知道没有声明的程序变量的使用情况，如果它在某个语句作为整型使用，而在另一语句作为数组使用，那么可以报告使用不一致的错误；相反，如果变量总是作为整型使用，那么不仅能保证它使用的一致性，而且能推断出它的类型必定是整型。

【例 5.4】 类型推断技术可用于 C 和 Pascal 这样的语言，在编译时补上程序中欠缺的类型信息。以图 5.10 中的 C 代码段为例，函数 mlist() 接受一个参数 p，该参数 p 是指向函数的指针。虽然从函数 mlist() 的第一行能知道 p 指向一个函数，但不能确定其所指向函数的变元个数和类型。在早期的 C 和 Pascal 参考手册中，这种类型不完整的 p 声明是被允许的。

函数 mlist() 把参数 p 指向的函数用于链表的每个表元。例如，p 可用于给表元中整型变量置初值或打印其值。虽然 p 指向的函数的变元类型没有指明，但从表达式 p(lptr) 中 p 的

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第202页 文件编号: 422624 页面编号: 32024538442 打印人: 8685 2026-03-09 10:23:01


```
typedef cell * link;
void mlist(link lptr, void (*p)()){
    while (lptr != NULL){
        p(lptr);
        lptr = lptr -> next;
    }
}
```

图 5.10 带函数指针参数 p 的函数 mlist()

使用可以推断出 p 的类型必须是

$$\text{pointer}(\text{link} \to \text{void})$$

对于函数 mlist 的任何调用，如果参数不是这个类型，则是一个错误。

类型推断技术和类型检查技术有很多共同的地方。在这两种情况下都要处理含变量的类型表达式。类似于例 5.5 的推导，在本节将用一个类型检查器来推断变量代表的类型。

【例 5.5】 在下面假想的程序中，可以推断出多态函数 deref() 的类型。函数 deref() 和 C 语言指针的解引用运算符 * 有同样的作用：

```
function deref (p){
    return *p;
}
```

看见 **function** deref (p) 时，对 p 的类型一无所知，因而用类型变量 β 代表它的类型。根据定义，运算符 * 作用于一个指针，返回该指针指向的对象。因为运算符 * 在表达式 *p 中作用于 p，因此 p 必定是指向某个对象的指针，让该对象的类型用含另一个类型变量 α 的类型表达式来表示，则可以断定

$$\beta = \text{pointer}(\alpha)$$

而且，表达式 *p 有类型 α，因此函数 deref() 的类型表达式可以写成

$$\forall\alpha.\text{pointer}(\alpha) \to \alpha$$

5.5.3 一个含多态函数的语言

到目前为止，多态函数是指可以用“不同类型”的变元来执行的函数。多态函数的参数所允许的类型集合的精确定义可以用全称量词符号 $\forall$ 表示，其含义是“对任何类型”。非形式地，含有符号 $\forall$ 的类型表达式称为“多态的类型”。

用于检查多态函数的抽象语言由图 5.11的文法定义。之所以称为抽象语言，是因为它像图 5.3 所示的文法那样，忽略了函数定义的函数体，而且让非终结符 T 直接产生抽象类型表达式，以简化语言和突出要讨论的问题。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第203页 文件编号: 422624 页面编号: 32034538440 打印人: 8685 2026-03-09 10:23:02

$$
\begin{aligned}
&P \to D;\ E \\
&D \to D;\ D \mid \textbf{id}\ :\ Q \\
&Q \to \forall\ \textbf{type_variable}.\ Q \mid T \\
&T \to T\ '\to '\ T \\
&\quad \mid T\ \times\ T \\
&\quad \mid \textbf{unary_constructor}\ (T) \\
&\quad \mid \textbf{basic_type} \\
&\quad \mid \textbf{type_variable} \\
&\quad \mid (T) \\
&E \to E\ (E) \mid E,\ E \mid \textbf{id}
\end{aligned}
$$

图 5.11　有多态函数的语言的文法

该文法对图 5.3 的文法还做了一些简化。首先，程序是由类型声明和表达式（而不是语句）组成的，这样可以免去考虑语句，因为语句部分的类型检查不会因加入多态类型而有什么变化。表达式也做了简化，略去了与问题无关的一些选择。用 **basic_type** 类型来表示像 **boolean** 和 **integer** 这样的基本类型，用 **unary_constructor** 表示一元构造符，它允许写出像 pointer(integer) 和 list(integer) 形式的类型。

该文法增加了多态类型函数的声明及笛卡儿积类型的声明。笛卡儿积类型可以用来表示多元函数类型，产生式 $E \to E,\ E$ 用以推导多个实参表达式的情况。$T \to (T)$ 仅用来组合类型。

由该文法产生的一个程序实例如图 5.12 所示。

$$
\begin{aligned}
&\text{deref} : \forall\alpha.\text{pointer}(\alpha) \to \alpha; \\
&q : \text{pointer}(\text{pointer}(\text{integer})); \\
&\text{deref}(\text{deref}(q))
\end{aligned}
$$

图 5.12　由图 5.11 所示文法产生的一个程序实例

再来看引入类型变量、笛卡儿积类型和多态函数后增加的推理规则。

下面的环境规则（Env Var）用来把一个类型变量加到静态定型环境中，这样，环境就不再仅仅是程序变量到类型的映射了。

$$
\text{(Env Var)} \qquad \frac{\Gamma \vdash \Diamond \quad \alpha \notin \text{dom}(\Gamma)}{\Gamma, \alpha \vdash \Diamond}
$$

语法规则增加了 4 条：规则（Type Var）用来从环境中得到一个类型变量；规则（Type Product）说明积类型的语法；规则（Type Parenthesis）表示类型表达式外加括号后得到的仍然是类型表达式；规则（Type Forall）是为多态类型准备的，它表示加全称量词以形成多态类型。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第204页 文件编号: 422624 页面编号: 32044538448 打印人: 8685 2026-03-09 10:23:02

$$\text{(Type Var)} \qquad \frac{\Gamma_1, \alpha, \Gamma_2 \vdash \Diamond}{\Gamma_1, \alpha, \Gamma_2 \vdash \alpha}$$

$$\text{(Type Product)} \qquad \frac{\Gamma \vdash T_1 \quad \Gamma \vdash T_2}{\Gamma \vdash T_1 \times T_2}$$

$$\text{(Type Parenthesis)} \qquad \frac{\Gamma \vdash T}{\Gamma \vdash (T)}$$

$$\text{(Type Forall)} \qquad \frac{\Gamma, \alpha \vdash T}{\Gamma \vdash \forall \alpha . T}$$

下面是关于积的定型规则

$$\text{(Exp Pair)} \qquad \frac{\Gamma \vdash E_1 : T_1 \quad \Gamma \vdash E_2 : T_2}{\Gamma \vdash E_1, E_2 : T_1 \times T_2}$$

比较直观，如果两个表达式 E_1 和 E_2 的类型分别是 T_1 和 T_2，那么 E_1 和 E_2 的类型是 $T_1 \times T_2$。它用于多元函数的情况。

多态类型的函数标识符在使用时要去掉全称量词，用一个新的类型变量代替约束变量。记号 $[\alpha_i/\alpha]T$ 表示 T 中自由出现的 α 用 α_i 代换后的结果。

$$\text{(Exp Id Fresh)} \qquad \frac{\Gamma_1, \mathbf{id} : \forall \alpha . T, \Gamma_2, \alpha_i \vdash \Diamond}{\Gamma_1, \mathbf{id} : \forall \alpha . T, \Gamma_2, \alpha_i \vdash \mathbf{id} : [\alpha_i/\alpha]\, T}$$

对于非多态的函数标识符，仍然用 5.3 节的规则 (Exp Id)。

多态函数调用的定型规则非常复杂，为避免复杂，让一个代换（类型变量到类型表达式的映射）S 直接出现在类型表达式中，$S(T)$ 表示代换结果的类型表达式。

(Exp FunCall)(S是T_1和T_3最一般的合一代换)

$$\frac{\Gamma \vdash E_1 : T_1 \to T_2 \quad \Gamma \vdash E_2 : T_3}{\Gamma \vdash E_1(E_2) : S(T_2)}$$

在上面这条定型规则中，对 S 的限制是，它是 T_1 和 T_3 最一般的合一代换。非多态的函数调用可以统一使用本定型规则。下面先解释代换与合一等概念，然后给出多态函数检查的翻译方案。

5.5.4 代换、实例与合一

将类型表达式中的类型变量用指定的类型表达式替换，称为**代换**。假设代换 Sv 是类型变量到类型表达式的映射，下面的递归函数 subst(t, Sv) 阐明了应用代换 Sv 去替换不含量

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第205页 文件编号: 422624 页面编号: 32054538445 打印人: 8685 2026-03-09 10:23:02

词的类型表达式 t 中所有类型变量的概念，函数类型构造符在 subst 中被作为“典型”的类型构造符。

typeExpression subst (typeExpression t, typeVar $\rightarrow$ typeExpression Sv) {
    if (t是基本类型) **return** t;
    else if (t是类型变量) **return** $Sv(t)$;
    else if (t是 $t_1 \rightarrow t_2$) **return** subst$(t_1, Sv) \rightarrow$ subst(t_2, Sv);
}

为方便起见，用 $S(t)$ 表示 subst 用于 t 后所得的类型表达式，即 S 可以从 Sv 扩展为类型表达式到类型表达式的映射，这个结果 $S(t)$ 称为 t 的实例。如果代换 S 对变量 α 没有指定类型表达式，则假定 $S(\alpha)$ 仍是 α，即 S 在这样的类型变量上是恒等映射。

【例 5.6】 下面给出一些类型表达式实例的例子，其中 $s < t$ 用来表示 s 是 t 的实例。

pointer(integer) < pointer(α)
pointer(real) < pointer(α)
integer $\rightarrow$ integer < $\alpha \rightarrow \alpha$
pointer(α) < β
$\alpha < \beta$

但是，下面左边的类型表达式不是右边的实例，原因列在括号中：

integer	real	（代换不能用于基本类型）
integer $\rightarrow$ real	$\alpha \rightarrow \alpha$	(α的代换不一致)
integer $\rightarrow \alpha$	$\alpha \rightarrow \alpha$	(α的所有出现都应该代换)

如果存在某个代换 Sv 使得 $S(t_1) = S(t_2)$，那么称类型表达式 t_1 和 t_2 能够**合一**（unify），S 是合一代换。实际感兴趣的是**最一般的合一代换**（the most general unifier），它是对类型表达式中的变量限制最少的代换。更精确地说，类型表达式 t_1 和 t_2 最一般的合一代换是具有下列性质的代换 S：

（1）$S(t_1) = S(t_2)$；

（2）对任何其他满足 $S'(t_1) = S'(t_2)$ 的代换 Sv'，代换 $S'(t_1)$ 是 $S(t_1)$ 的实例。

【例 5.7】 假设有类型表达式 $t_1 =$ pointer(list(α))、$t_2 =$ pointer(β)。若存在：

代换 $Sv : \alpha \rightarrow \alpha, \beta \rightarrow$ list(α)，使得 $S(t_1) = S(t_2) =$ pointer(list(α))；

代换 $Sv' : \alpha \rightarrow$ integer, $\beta \rightarrow$ list(integer)，使得 $S'(t_1) = S'(t_2) =$ pointer(list(integer))，则代换 Sv 是最一般的合一代换。

下面所说的合一都是指最一般的合一代换。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第206页 文件编号: 422624 页面编号: 32064538443 打印人: 8685 2026-03-09 10:23:02

5.5.5 多态函数的类型检查

多态函数的类型检查在 3 个方面和 5.3 节的普通函数的类型检查有区别。在说明多态函数的类型检查前，用图 5.12 所示程序的表达式 deref (deref (*q*)) 来说明这些不同。这个表达式的语法树在图 5.13 中给出。每个结点有两个标记：第一个标记指出结点代表的子表达式，第二个标记是指派给该表达式的类型表达式。下标 i 和 o 分别用来区分 deref 的里外两次出现。

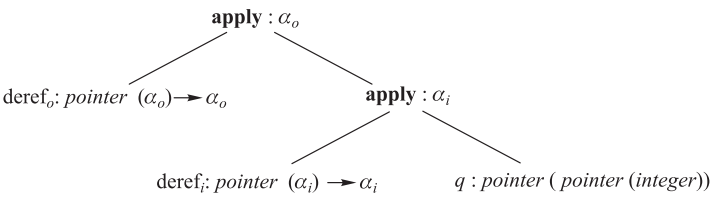


图 5.13 deref(deref (*q*)) 的带标记的语法树

多态函数的类型检查和普通函数的类型检查的区别如下：

（1）表达式中同一多态函数的不同出现变元无须有相同类型。在表达式 $\text{deref}_o(\text{deref}_i\ (q))$ 中，deref_i 会去掉一层指针间接访问，所以 deref_o 和 deref_i 作用于不同类型的变元。这个性质来源于对 deref 函数的类型表达式中的 $\forall\alpha$ 的解释：对任何类型 α，deref 的每次出现对约束变量 α 代表什么类型有它自己的观点，所以对 deref 的每次出现指派一个类型表达式，这个类型表达式是用新的类型变量代替 deref 的类型表达式中的 α 并且去除全称量词 $\forall$ 后形成的，也就是使用规则（Exp Id Fresh）。在图 5.13 中，新的变量 α_o 和 α_i 分别用于里外两个 deref 的类型表达式中。

（2）由于变量可以出现在类型表达式中，在某些场合必须把类型相同的概念推广到类型合一。假如类型为 $s \rightarrow s'$ 的 E_1 作用于类型为 t 的 E_2, 不是简单地确定 s 和 t 是否相同，而是要看它们能否合一。例如，图 5.13 中的根结点右子树上那个标有 **apply** 的结点，如果 α_i 用 pointer(integer) 代替，就能使得 pointer(α_i) 和 pointer(pointer(integer)) 相同。

（3）要有办法记录两个子表达式的合一代换。通常，类型变量可以出现在几个类型表达式中。如果 s 和 s' 的合一使得变量 α 代表类型 t，那么在类型检查的过程中 α 必须继续代表 t。例如，图 5.13 中，α_i 是 deref_i 的值域类型，所以它代表 $\text{deref}_i\ (q)$（也就是右子树上那个 **apply** 结点）的类型。deref_i 定义域类型和 q 的类型的合一使得 α_i 代表 pointer(integer)。图 5.13 中的另一个类型变量 α_o 是 $\text{deref}_o(\text{deref}_i\ (q))$ 的类型，合一运算使得 α_o 代表 integer。

现在可以考虑多态函数的类型检查算法了。由图 5.11 所示文法产生的表达式的类型检查的翻译方案如图 5.14 所示，图 5.14 中略去了声明语句和其他无关语句的翻译方案。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第207页 文件编号: 422624 页面编号: 32074538441 打印人: 8685 2026-03-09 10:23:02

$E \to E_1(E_2)$	$\{p = \text{mkLeaf}(\text{newTypeVar}());$
	$\text{unify}(E_1.\text{type}, \text{mkNode}(' \to ', E_2.\text{type}, p));$
	$E.\text{type} = p;\}$
$E \to E_1, E_2$	$\{E.\text{type} = \text{mkNode}(' \times ', E_1.\text{type}, E_2.\text{type});\}$
$E \to \textbf{id}$	$\{E.\text{type} = \text{fresh}(\text{lookup}(\textbf{id}.\text{entry}));\}$

图 5.14 检查多态函数的翻译方案

与 5.3 节的翻译方案不同的是，当由非终结符 T 和 Q 产生的类型表达式被扫描时，它产生语法树形式的类型表达式，也就是这里的类型表达式都用语法树来表示。类型图中的叶结点和内部结点可用类似于 4.2 节的 mkLeaf 和 mkNode 的操作来构造。图 5.14 的翻译方案还根据下面的操作定义。

（1）fresh(t)：它把 t 所指类型表达式中的约束变量用新的变量代替，返回指向替换后的类型表达式的指针。它对应规则（Exp Id Fresh）。

（2）unify(m, n)：它合一由 m 和 n 所指向的类型表达式。它有副作用，记住使类型表达式相同的代换 S，也就是规则（Exp FunCall）中提到的代换 S。如果 m 和 n 所指类型表达式不能合一，整个类型检查过程失败。本书不介绍合一算法。

这里需要解释函数应用 $E \to E_1(E_2)$ 的语义动作，它受到 E_1.type 和 E_2.type 都是类型变量情况的启发。例如，若 $E_1.\text{type} = \alpha$ 且 $E_2.\text{type} = \beta$，那么 α 必须是函数，使得存在某个未知类型 γ 满足 $\alpha = \beta \to \gamma$。在图 5.14 中，对应 γ 的新类型变量由 newTypeVar 调用获得，并且 E_1.type 和 $E_2.\text{type} \to \gamma$ 需要合一而不是相等。合一后，γ 代表结果类型。该语义动作和规则（Exp FunCall）有些区别，但本质上一样。

用一个简单的例子来说明图 5.14 的语义动作。表 5.1 给出了指派到每个子表达式的类型表达式，总结了这个算法的执行结果。在每次函数应用时，unify 操作都有副作用，它记录类型变量代表的类型表达式。这样的副作用由表 5.1 中的“代换”栏表示。

表 5.1 自下而上确定类型的小结

表达式：类型	代换
q:pointer(pointer(integer))	
deref_i:pointer(α_i) $\to \alpha_i$	
$\text{deref}_i(q)$: pointer(integer)	α_i = pointer(integer)
deref_o:pointer(α_o) $\to \alpha_o$	
$\text{deref}_o(\text{deref}_i(q))$: integer	α_o = integer

下面一个例子把 ML 多态函数的类型推断同 5.3.3 小节和图 5.14 的类型检查联系起来。ML 的函数定义的语法由

$$\textbf{fun}\ \textbf{id}_0(\textbf{id}_1, \cdots, \textbf{id}_k) = E;$$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第208页 文件编号: 422624 页面编号: 32084538449 打印人: 8685 2026-03-09 10:23:02

给出。其中，$\mathbf{id}_0$ 代表函数名，$\mathbf{id}_1, \cdots, \mathbf{id}_k$ 代表它的参数。为简单起见，假定表达式 E 的语法如图 5.11 所示，并且 E 中的标识符都是函数（包括内部函数）名或参数名。这里是对5.5方法的形式化。在5.5 中，类型推断用来决定 deref 的多态类型。现在，先为函数和它的变元构造新的类型变量。内部函数通常是多态类型的，出现在这些类型中的任何类型变量都由全称量词 $\forall$ 约束。然后检查表达式 $\mathbf{id}_0(\mathbf{id}_1, \cdots, \mathbf{id}_k)$ 的类型和 E 的类型是否匹配。若匹配成功，则得出该函数的类型。最后，将所得函数类型的任何变量都用 $\forall$ 量词约束，得到该函数的多态类型。

【例 5.8】 写出图 5.9 的确定表长度的 ML 函数：

```
fun length (lptr) =
    if null (lptr) then 0
    else length (tl (lptr)) + 1;
```

为 length 和 lptr 分别引入类型变量 β 和 γ 来表示它们的类型。在形式证明中，可以发现 length(lptr) 的类型和函数体的类型能够匹配，并且得到 length 的类型是

$$\forall\alpha.\text{list}(\alpha) \to \text{integer}$$

更详细地，可以建立图 5.15 所示的程序来解释如何运用 5.3.3 小节和图 5.14 的翻译方案来进行类型检查。该程序的声明把新类型变量 β 和 γ 与 length 和 lptr 联系起来，并把内部函数的类型显式化。按图 5.11 的风格，把多态函数 if 作用于 3 个运算对象，它们分别代表被测试的条件，then 部分和 else 部分的类型。这个声明指出 then 和 else 部分可以和任何类型 α 匹配，它也是结果的类型。

$$
\begin{array}{l}
\text{length} : \beta; \\
\text{lptr} : \gamma; \\
\text{if} : \forall\alpha.\text{boolean} \times \alpha \times \alpha \to \alpha; \\
\text{null} : \forall\alpha.\text{list}(\alpha) \to \text{boolean}; \\
\text{tl} : \forall\alpha.\text{list}(\alpha) \to \text{list}(\alpha); \\
0 : \text{integer}; \\
1 : \text{integer}; \\
+ : \text{integer} \times \text{integer} \to \text{integer}; \\
\text{match} : \forall\alpha.\alpha \times \alpha \to \alpha; \\
\text{match}(\\
\quad \text{length(lptr)}, \\
\quad \text{if(null(lptr)}, 0, \text{length(tl(lptr))} + 1) \\
)
\end{array}
$$

图 5.15 类型声明及要检查的表达式

显然，length(lptr) 必须和 length() 的函数体有同样的类型，这个检查用多态函数 match 表示。使用 match 是为了技术上的方便，它使得所有的检查都可以用图 5.11 所示风格的程

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第209页 文件编号: 422624 页面编号: 32094538447 打印人: 8685 2026-03-09 10:23:02

序完成。

把图 5.14 所示的类型检查用于图 5.15 的结果见表 5.2。这里只给出了自下而上分析表达式 match (···) 的情况，略去了类型检查用于声明语句的情况。该表的第 3 列是所用的合一代换，第 4 列是得出该行定型断言所使用的规则。规则（Exp Id Fresh）（或者说操作 fresh）引入的用于多态的内部函数的新类型变量由 α 的下标区别。

从第 3 行可以知道 length() 必须是从 γ 到某个未知类型 δ 的函数。然后，检查子表达式 null(lptr) 时，在第 6 行确定 γ 和 list(α_n) 合一，其中 α_n 是未知类型。在该点，length 的类型必须是

$$\forall\alpha_n.\text{list}(\alpha_n) \to \delta$$

最后，在第 15 行检查加运算时，确定 δ 和 integer 合一。为清楚起见，把 “+” 写在两个变元之间。当类型检查完毕时，类型变量 α_n 仍留在函数 length() 的类型中。因为对类型 α_n 没有任何限制，所以使用该函数时可用任何类型代换它。让它成为约束变量，并写成

$$\forall\alpha_n.\text{list}(\alpha_n) \to \text{integer}$$

这就是 length 的类型。

表 5.2 length 的类型推导过程

行	定型断言	代换	规则
1	`lptr` : γ		(Exp Id)
2	`length` : β		(Exp Id)
3	`length(lptr)` : δ	$\beta = \gamma \to \delta$	(Exp FunCall)
4	`lptr` : γ	从第 1 行可得	
5	`null` : list(α_n) → boolean		(Exp Id Fresh)
6	`null(lptr)` : boolean	$\gamma = \text{list}(\alpha_n)$	(Exp FunCall)
7	0 : integer		(Exp Num)
8	`lptr` : list(α_n)	从第 1 行可得	
9	`tl` : list(α_t) → list(α_t)		(Exp Id Fresh)
10	`tl(lptr)` : list(α_n)	$\alpha_t = \alpha_n$	(Exp FunCall)
11	`length` : list(α_n) → δ	从第 2 行可得	
12	`length(tl(lptr))` : δ		(Exp FunCall)
13	l : integer		(Exp Num)
14	+ : integer × integer → integer		(Exp Id)
15	`length(tl(lptr))` + 1 : integer	δ = integer	(Exp FunCall)
16	`if` : boolean × α_i × α_i → α_i		(Exp Id Fresh)
17	`if(···)` : integer	α_i = integer	(Exp FunCall)
18	`match` : α_m × α_m → α_m		(Exp Id Fresh)
19	`match(···)` : integer	α_m = integer	(Exp FunCall)

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第210页 文件编号: 422624 页面编号: 32104538445 打印人: 8685 2026-03-09 10:23:02

5.6 函数和运算符的重载

一个容易和多态性混淆的概念是符号（如函数名）的重载。尤其是在某些不支持多态性的语言中将类似于本书重载的概念定义为多态，这使得二者更加难以区分。**重载**（overload）**符号**是指该符号有多个含义，但在该符号的每个引用点，其含义可以依赖上下文来唯一地确定。在数学中，加法运算符“+”是重载的，因为当 A 和 B 是整数、实数、复数或矩阵时，$A+B$ 中的“+”有不同的含义。在 Ada 语言中，括号 () 是重载的，表达式 $A(I)$ 可能是数组 A 的第 I 个元素的引用，也可能是用变元 I 调用函数 A。多态的符号只有一个含义，但是它允许参数类型在一定的范围内变化。

在重载符号的引用点，如果其含义能唯一地确定，就称为重载的消除。例如，如果“+”可以表示整数加或实数加，那么在 $x+(i+j)$ 中，“+”的两个出现可以表示不同形式的加，它取决于 x、i 和 j 的类型。重载的消除有时也称为**运算符的辨别**，因为它确定运算符号指称哪个运算。

在大多数语言中，算术运算符是重载的。不过，它们的重载可以通过仅看它们的变元的类型而消除。它们唯一含义的确定类似于图 5.5 中 $E \to E_1$ **op** E_2 的语义规则，即 E 的类型可以由 E_1 和 E_2 的类型来确定。

注意：重载的函数和符号的引入使得程序员可以用一个名字或符号表示多个不同类型的函数或运算，它并不像引入类型构造符或类型变量那样能丰富所能表达的类型。

5.6.1 子表达式的可能类型集合

并不总是只看函数的变元就可以消除重载。如例 5.9 所示，一个子表达式本身有一个可能类型的集合，而不只是一个类型。在 Ada 语言中，上下文必须提供足够的信息来缩小到单元素集合，也就是唯一类型。

【例 5.9】 在 Ada 语言中，运算符 $*$ 的一个标准（内部定义）解释是整数集合上的二元函数。加入下面这样的声明：

function ''*'' (i, j : integer) return complex;
function ''*'' (x, y : complex) return complex;

会使得 $*$ 重载。在上述声明后，$*$ 可能的类型包括：

integer × integer → integer
integer × integer → complex
complex × complex → complex

如果从 2、3 和 5 的字面知道它们是整型常量，那么在上述声明的环境下，子表达式 $3*5$ 是整型或复型，到底是哪一个则取决于它的上下文。如果完整的表达式是 $2*(3*5)$，那

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第211页 文件编号: 422624 页面编号: 32114538443 打印人: 8685 2026-03-09 10:23:02

么 $3 * 5$ 必须是整型，因为 $*$ 的两个变元要么都是整型，要么都是复型；相反，如果表达式是 $(3 * 5) * z$ 且 z 是复型，那么 $3 * 5$ 必须是复型。

5.3 节假定每个表达式有唯一的类型，所以函数应用的类型检查是

$$E \to E_1(E_2) \qquad \{\textbf{if } (E_2.\text{type} == s \;\&\&\; E_1.\text{type} == s \to t)$$
$$E.\text{type} = t;$$
$$\textbf{else } E.\text{type} = \text{type_error};\}$$

表 5.3 把这条规则自然地推广到有类型集合的情况。表 5.3 仅有的运算是函数应用，表达式中其他运算符的类型检查仍类似于前面的检查。重载的标识符可能有几个声明，所以假定符号表的条目包含可能类型的集合，这个集合由 lookup() 函数返回。开始非终结符 E' 产生完整的表达式，它的作用将在下面阐述。

表 5.3 确定表达式可能类型的集合

产生式	语义规则
$E' \to E$	$E'.\text{types} = E.\text{types}$
$E \to \textbf{id}$	$E.\text{types} = \text{lookup}(\textbf{id}.\text{entry})$
$E \to E_1(E_2)$	$E.\text{types} = \{t \mid \exists s \in E_2.\text{types}, \text{使得} s \to t \in E_1.\text{types}\}$

如果用自然语言叙述，表 5.3 第 3 行的语义规则可以这么说：如果 s 是 E_2 的一个类型，并且 E_1 的一个类型能把 s 映射到 t，那么 t 是 $E_1(E_2)$ 的一个类型。函数应用的类型不匹配会使集合 E.types 为空，可以用它作为通知类型错误的条件。

【例 5.10】 考虑表达式 $3 * 5$，设运算符 $*$ 的声明如5.9 所示的那样，即根据上下文，$*$ 可以把一对整数或复数映射到整数或复数。子表达式 $3 * 5$ 的可能类型集合在图 5.16 中，其中 i 和 c 分别是 integer 和 complex 的缩写。除了解释表 5.3 所示的语义规则，例 5.10 还提示了如何将这种方法应用于其他程序构造。

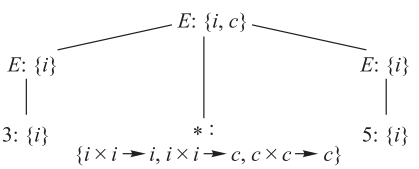


图 5.16 表达式 $3 * 5$ 可能的类型集合

5.6.2 缩小可能类型的集合

Ada 要求完整的表达式有唯一的类型。根据上下文确定了表达式的唯一类型，然后据此可以缩小每个子表达式的类型选择。如果这个过程不能使每个子表达式的类型唯一确定，那么宣布该表达式有类型错误。表 5.4 为缩小表达式的类型集合。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第212页 文件编号: 422624 页面编号: 32124538441 打印人: 8685 2026-03-09 10:23:02

表 5.4 缩小表达式的类型集合

产生式	语义规则	两遍计算时机
$E' \to E$	E'.types = E.types;	第 1 遍：exit 时
	if (E'.types == $\{t\}$) E.unique = t; else E.unique = type_error;	第 2 遍：enter 时
	E'.code = E.code;	第 2 遍：exit 时
$E \to$ **id**	E.types = lookup(**id**);	第 1 遍：exit 时
	E.code = gen(**id**.lexeme, ':', E.unique);	第 2 遍：exit 时
$E \to E_1(E_2)$	E.types = $\{s' \mid E_2$.types中存在一个s, 使得$s \to s'$属于E_1.types$\}$;	第 1 遍：exit 时
	t = E.unique;	第 2 遍：enter 时
	if (S == $\{s\}$) E_2.unique = s; else E_2.unique = type_error;	第 2 遍：enter 时
	if (S == $\{s\}$) E_1.unique = $s \to t$; else E_1.unique = type_error;	第 2 遍：enter 时
	E.code = E_1.code || E_2.code || gen('**apply**', ':', E.unique);	第 2 遍：exit 时

表 5.4 的语法制导定义由表 5.3 的定义加上定义 E 的继承属性 unique 的语义规则得到。由于整个表达式由 E' 产生，所以希望 E'.types 是单个类型的集合。这个类型由 E.unique 继承，再传入 E。E 的综合属性 code 表示生成的中间代码，这里用的是后缀表示。在中间表示中，每个标识符和 **apply** 运算符的实例都有一个类型。函数 gen 表示产生字符串形式的代码，“||”表示串的连接运算。基本类型 type_error 仍是通知出现了错误。

如果函数应用 $E_1(E_2)$ 返回类型 t，那么可以找到类型 s, 它对变元 E_2 是可行的，同时，$s \to t$ 对该函数是可行的。表 5.4 中语义规则的集合 S 用来检查具有这个性质的唯一类型 s。

注意：表 5.4 的语法制导定义不是 L 属性定义，不能边解析边计算，因为 E 的继承属性 unique 依赖于产生式左部符号的综合属性 types 及其自身或右边文法符号的综合属性 types。这时，可以通过对表达式语法树的两次遍历来实现。在第一次遍历时，属性 types 被自下而上地综合，即在树结点返回 (exit) 时计算。第二次扫描时，属性 unique 被自上而下传播，即在进入 (enter) 结点时计算；而当从结点返回 (exit) 时，code 属性被综合。各语义规则的计算时机列在表 5.4 的第 3 列。

习 题

5.1 (a) 写一个简单的有控制流错误的 C 程序并提交编译，确认编译器是否报告了这个控制流错误。

(b) 写一个简单的不满足唯一性要求的 C 程序并提交编译，确认编译器是否进行了唯一性检查并报告错误。

5.2 语句的文法如下：

$S \to$ **id** := E | **if** E **then** S | **while** E **do** S | **begin** S; S **end** | **break**

写一个翻译方案，其语义动作的作用是若发现 **break** 不是出现在循环语句中，则及时报告

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第213页 文件编号: 422624 页面编号: 32134538449 打印人: 8685 2026-03-09 10:23:02

错误。

5.3 下面是一个 C 语言程序，其中的形参采用传统的声明方式。虽然 main() 函数中两次调用函数 gcd() 的参数个数都不对，但是该程序能够通过编译并且连接而形成一个目标程序。试问：为什么 C 编译器和连接器没能发现这样的错误？

```
long gcd(p,q) long p,q; {
    if (p%q == 0)
        return q;
    else
        return gcd(q, p%q);
}
void main() {
    printf( "%ld,%ld\n" ,gcd(5),gcd(5,10,20));
}
```

5.4 (a) 指向实数的指针数组，数组的下标从 0 到 99。

(b) 二维数组（元素类型为一维数组的数组），它的行下标从 0 到 9，列下标从 0 到 19。

(c) 函数，它的定义域是从整数到整数指针的函数，它的值域是由一个整数和一个字符组成的记录。

5.5 假如有下列 C 程序声明

```
typedef struct {
    int a, b;
} CELL, * PCELL;
CELL foo[100];
PCELL bar(int x, CELL y){ ... }
```

为变量 foo 和函数 bar 的类型写出类型表达式。

5.6 写出下面的变量 a 的类型表达式。

```
(1) int a[][3];
(2) int *a[3];
(3) int (*a)[3];
(4) int *(*a)[3];
(5) int **a[3];
(6) int *(*a[3])[2];
```

5.7 下列文法定义字面常量表的表。符号的解释与图 5.3 所示文法中的解释相同，增加了类型 **list**，它表示类型 T 的元素表。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第214页 文件编号: 422624 页面编号: 32144538447 打印人: 8685 2026-03-09 10:23:02

$P \to D\,;E$
$D \to D\,;D \mid \textbf{id} : T$
$T \to \textbf{list of}\ T \mid \textbf{char} \mid \textbf{integer}$
$E \to (L) \mid \textbf{literal} \mid \textbf{num} \mid \textbf{id}$
$L \to E, L \mid E$

写一个类似 5.3 节中的翻译方案，以确定表达式 (E) 和表 (L) 的类型。

5.8 把产生式

$E \to \textbf{nil}$

加入习题 5.7 的文法。含义是表达式可以是空表。修改习题 5.7 的答案，把 **nil** 看成空表，其元素可以是任何类型。

5.9 修改 5.3.3 小节的翻译方案，发现错误时打印描述信息，并继续检查，好像所期望的类型已经看见。

5.10 修改 5.3.3 小节的翻译方案，使之能处理：

(a) 各种有值语句。赋值语句的值是赋值号右边的表达式的值，条件语句或循环语句的值是语句体的值，语句表的值是表中最后一个语句的值。

(b) 布尔表达式。加上逻辑运算符 **and**、**or** 及 **not** 和关系运算符的产生式。然后给出适当的翻译规则，它们检查这些表达式的类型。

5.11 推广在 5.3 节给出的一元函数类型检查，使之能处理 n 元函数。

5.12 C 语言是一种类型化语言，但它不是类型可靠的语言，因为运行前的类型检查不能保证所接受的程序没有不会被捕获的错误。例如，编译时的类型检查一般不能保证运行时共用体中数据的使用不出现类型问题。请你再举一个例子说明 C 语言不是类型可靠的语言。

5.13 拓展 5.3.3 小节的类型检查，使之能包含记录。有关记录部分的类型和记录域引用表达式的语法如下：

$T \to \textbf{record}\ \text{fields}\ \textbf{end}$
$\text{fields} \to \text{fields};\text{field} \mid \text{field}$
$\text{field} \to \textbf{id} : T$
$E \to E.\textbf{id}$

5.14 在文件 stdlib.h 中，关于 qsort 的外部声明如下：

```
extern void qsort(void *, size_t, size_t, int (*)(const void *, const void *));
```

下面的 C 语言程序所在的文件名是 type.c

```
1  # include <stdlib.h>
2  typedef struct{
3    int Ave;
4    double Prob;
5  }HYPO;
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第215页 文件编号: 422624 页面编号: 32154538444 打印人: 8685 2026-03-09 10:23:02


```
6  HYPO *astHypo;
7  int n;
8  int HypoCompare(HYPO *stHypo1, HYPO *stHypo2) {
9    if (stHypo1->Prob > stHypo2->Prob){
10     return(-1);
11   }else if (stHypo1->Prob < stHypo2->Prob) {
12     return(1);
13   }else{
14     return(0);
15   }
16 }/* end of function HypoCompare */
17 void main() {
18   qsort(astHypo, n, sizeof(HYPO), HypoCompare);
19 }
```

用某个 C 编译器编译时，错误信息如下：

type.c:18: warning: passing argument 4 of 'qsort' from incompatible pointer type

请对该程序略作修改，使得该警告错误能消失，并且不改变程序的结果。

注：程序中关于变量 astHypo 和 n 的赋值及其他部分被略去。

5.15 使用类型变量表示下列函数的类型。

(a) 函数 ref()，它取任意类型的对象作为变元，返回这个对象的指针。

(b) 函数 arrayderef()，它以一个数组为变元，数组的下标是整型，数组的元素是某类型的指针，返回一个数组，它的元素是变元数组的元素所指向的对象。

5.16 找出下列表达式的最一般的合一代换。

(a) (pointer $(\alpha)) \times (\beta \to \gamma)$

(b) $\beta \times (\gamma \to \delta)$

如果 (b) 的 δ 是 α 呢？

5.17 对下面的每对表达式，找出最一般的合一代换。

(a) $\alpha_1 \to (\alpha_2 \to \alpha_1)$

(b) $\text{array}(\beta_1) \to (\text{pointer}(\beta_1) \to \beta_2)$

(c) $\gamma_1 \to \gamma_2$

(d) $\delta_1 \to (\delta_2 \to \delta_1)$

5.18 效仿例 5.8，推导下面 map 的多态类型。

map: $\forall\alpha.\forall\beta.((\alpha \to \beta) \times \text{list}(\alpha)) \to \text{list}(\beta)$

map 的 ML 定义是

```
fun map ( f, l ) =
    if null ( l ) then nil
    else cons ( f ( hd ( l )), map ( f, tl ( l )));
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第216页 文件编号: 422624 页面编号: 32164538442 打印人: 8685 2026-03-09 10:23:02

在这个函数体中，内部定义的标识符的类型是

null: $\forall\alpha.\text{list}(\alpha) \to \text{boolean}$;

nil: $\forall\alpha.\text{list}(\alpha)$;

cons: $\forall\alpha.(\alpha \times \text{list}(\alpha)) \to \text{list}(\alpha)$;

hd: $\forall\alpha.\text{list}(\alpha) \to \alpha$;

tl: $\forall\alpha.\text{list}(\alpha) \to \text{list}(\alpha)$;

5.19 对于下面的 C 语言程序，在 x86/Linux 系统上，某版本的 GCC 编译器报告第 11 行有错误：

incompatible types in return

在 C 语言中，数组和结构体都是构造类型，为什么下面第二个函数有类型错误，而第一个函数没有？

```
1  typedef int A1[10];
2  typedef int A2[10];
3  A1 a;
4  typedef struct {int i;} S1;
5  typedef struct {int i;} S2;
6  S1 s;
7  A2 *fun1() {
8    return(&a);
9  }
10 S2 fun2() {
11   return(s);
12 }
```

5.20 一个 C 语言程序如下：

```
1  void fun(struct {int x; double r;} val) { }
2  void main() {
3    struct {int x; double r;} val;
4    fun(val);
5  }
```

该程序在 x86/Linux 系统上用某版本的 GCC 编译器编译时，报告的错误信息如下：

4: error: incompatible type for argument 1 of 'fun'

1: note: expected 'struct <anonymous>' but argument is not of type 'struct <anonymous>'

请问，报告错误的原因是什么？如何修改程序，使得编译时不再出现这个错误信息。

5.21 推广表 5.4 的算法，使表达式有类型构造符 array 和 pointer。

5.22 使用5.10 的规则，确定下列哪些表达式有唯一类型（假定 z 是复数）。

(a) $1 * 2 * 3$

(b) $1 * (z * 2)$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第217页 文件编号: 422624 页面编号: 32174538440 打印人: 8685 2026-03-09 10:23:02

(c) (1 * z) * z

5.23 在 C 语言的教材中，称 & 为地址运算符，&a 为变量 a 的地址。但是教材中没有说明表达式 &a 的类型是什么。另外，教材中说，数组名代表数组的首地址，但是也没有说明这个值的类型。它们所带来的一个问题是，如果 a 是一个数组名，那么表达式 a 和 &a 的值都是数组 a 的首地址，但是它们的使用是有区别的，初学时很难掌握。

下面给出 4 个 C 文件，请根据某版本的 GCC 编译器报错信息和程序运行结果，判断出表达式 a 和 &a 的类型。如果能明白它们的类型，就能知道它们的区别，从而可以正确使用它们。

（1）文件 1：

```
1  typedef int A[10][20];
2  A a;
3  A *fun() {
4    return(a);
5  }
```

该函数在 Linux 上用 GCC 编译器编译时，报告的类型错误如下：

第 4 行：warning: return from incompatible pointer type

（2）文件 2：

```
1  typedef int A[10][20];
2  A a;
3  A *fun() {
4    return(&a);
5  }
```

该函数在 Linux 上用 GCC 编译器编译时，没有类型方面的错误。

（3）文件 3：

```
1  typedef int A[10][20];
2  typedef int B[20];
3  A a;
4  B *fun() {
5    return(a);
6  }
```

该函数在 Linux 上用 GCC 编译时，没有类型方面的错误。

（4）文件 4：

```
1  typedef int A[10][20];
2  A a;
3  void fun() {
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第218页 文件编号: 422624 页面编号: 32184538448 打印人: 8685 2026-03-09 10:23:02


```
4    printf("%d,%d,%d\n", a, a+1, &a+1);
5  }
6  void main() {
7    fun();
8  }
```

该程序的运行结果是：

134518112, 134518192, 134518912

5.24 下面是一个 C 语言文件的内容。

```
1  long a1[10][10][10];
2  long a2[ ][10][10];
3  long a3[ ][ ][10];
4  void f (a4) long a4[ ][10][10]; {
5    a1[2][2][2] = a4[2][2][2];
6  }
7  void main() {
8    f (a1);
9  }
```

把它提交给 GCC 编译器 [版本 (GNU) 4.2.3 (Debian 4.2.3-5)] 时，编译器对第 2 行给出警告：array 'a2' assumed to have one element。对第 3 行报告错误：array type has incomplete element type。

(a) 为什么第 2 行给出警告，而第 3 行报告错误？

(b) 第 4 行的 a4 和第 2 行的 a2 的描述一样，为什么编译器对 a4 没有警告？

(c) 若在 f 函数中增加输出表达式 sizeof(a2) 和 sizeof(a4) 的值，你认为编译时会报错吗？

第 6 章

运行时存储空间的组织和管理

本章把过程和函数这样的程序单元统称为过程，程序运行时过程的一次执行称为过程的**活动**（activation）。过程的每次活动都需要可执行代码和存放所需数据的存储空间，所需的局部数据通常用一块连续的存储区来存放，称为**活动记录**。过程 p 的一次活动的**生存期**（lifetime）是从过程体开始执行到执行结束的时间，包括消耗在执行被 p 调用的过程所需的时间，以及再由这样的过程调用过程所花的时间等。在考虑代码生成之前，需要把静态的程序正文和运行时的活动联系起来，考察静态的名字和运行时数据对象之间的绑定关系。

由于过程可以递归，在程序运行中的某一时刻，可能有多个存活的活动与一个过程对应。因此递归过程中的一个变量可以联系到目标机器上不同的数据对象，虽然任何时刻只有一个这样的对象是可访问的。因此，本章不仅要讨论一个活动记录中的数据布局，还要讨论程序执行过程中，所有存活过程活动的活动记录的组织方式。

当过程执行结束时，一般而言，其局部变量不再可访问。然而，许多语言允许创建对象或其他数据，它们的生存期没有约束在创建它们的过程活动的生存期内。本章还需要讨论这类数据的存储分配。

过程能否递归和程序能否动态创建数据对象都是影响存储分配策略的重要语言特征，影响存储分配策略的主要语言特征是：

（1）过程能否递归；

（2）当控制从过程活动返回时，局部变量的值是否要保留；

（3）过程能否访问非局部变量；

（4）过程调用的参数传递方式；

（5）过程能否作为参数被传递；

（6）过程能否作为结果值被传递；

（7）存储块能否在程序控制下动态地分配；

（8）存储块是否必须显式地回收。

编译器组织运行时的存储空间和把名字绑定到数据单元的方式，在很大程度上取决于对上述问题的回答，这也是本章重点要解决的问题。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第220页 文件编号: 422624 页面编号: 32204538444 打印人: 8685 2026-03-09 10:23:02

面向对象语言和函数式语言特有的一些特征也会影响存储分配策略，其中面向对象语言的特征将在第 12 章介绍。

从编译器设计者的观点，目标程序运行在它自己的逻辑地址空间，该逻辑地址空间的组织和管理由编译器、操作系统和目标机器共同参与。操作系统把逻辑地址映射到物理地址，后者通常遍及整个内存。因此，本书的存储分配在逻辑地址空间中讨论。

6.1 名字、绑定与作用域

本节讨论一个过程活动所需局部信息的存储分配，先回顾与这个存储分配有关的语言概念，然后介绍活动记录中的数据布局，最后介绍对过程中并列的程序块实行重叠分配的方法。

6.1.1 过程

过程定义是一种声明，其最简单的形式是将一个名字和一个语句关联起来。该名字称为**过程名**，而这个语句则是**过程体**。在大多数编程语言中，有返回值的过程称为**函数**，而完整的程序也可以看作一个过程。在面向对象编程语言中，这种结构通常被称为**方法**。图 6.1 是用 C 语言写的快速排序程序示例，除 main 外，它还有 3 个函数。

在调用语句中出现的过程名表示该过程在该程序点**被调用**。过程调用是指执行被调用过程的过程体。如果过程调用出现在表达式中，这时称为**函数调用**。出现在过程定义中的参数称为该过程的**形式参数**（简称形参）。当调用过程时，过程调用语句中的**实在参数**（简称实参）会传递给被调用过程，并替换掉过程体中的形参。建立实参与形参之间对应关系的方法在 6.4 节讨论。

过程的一次调用称为**过程的活动**，而**活动的生存期**是指从该过程开始执行到执行结束的时间段。在此期间，过程的局部变量、参数及控制流都处于活动状态。当活动的生存期结束时，过程执行完成，控制权返回给调用者，过程的局部资源也随之被释放或回收。

6.1.2 名字的作用域和绑定

在编程语言中，**名字**是符号的文本表示，用来引用变量、常量、函数、类型等实体。编程语言对名字的硬性规定由构词规则描述，如第 2 章介绍的标识符的构词规则；而命名约定则是为了提高代码可读性和一致性而制定的软性约束，如约定变量名使用小写字母加下划线。

绑定（binding）是指在编程中将某个程序实体（如变量、常量、函数、类型等）与特定属性（如类型、值、存储位置、作用域等）关联的过程。通过绑定，程序中的标识符（名字）被赋予了实际含义，使它们在程序执行中可以正确引用或使用。语言中的声明是把信息与名

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第221页 文件编号: 422624 页面编号: 32214538442 打印人: 8685 2026-03-09 10:23:02

字关联起来的一种语法构造。例如，声明 int x = 5; 将名字 x 绑定到整型变量并初始化为 5。绑定可以在不同的程序点发生，如编译时（静态绑定）或运行时（动态绑定）。

```
int a[11];
void readArray() {/* Reads 9 integers into a[1],…,a[9].*/
    int i;
    ...
}
int partition(int k, int n) {
    /* Picks a separator value v, and partitions a[k..n] so that a[k..p-1] are less
       than v, a[p]=v, and a[p+1..n] are equal to or great than v. Returns p.*/
    ...
}
void quickSort(int k, int n) {
    int i;
    if(n>m){
        i=partition(k,n);
        quickSort(k,i-1);
        quickSort(i+1,n);
    }
}
main() {
    readArray();
    a[0]=-9999;
    a[10]=9999;
    quickSort(1,9);
}
```

图 6.1 快速排序程序的概略

一个声明起作用的程序部分称为该声明的**作用域**。当一个名字出现在其声明的作用域内时，称为局部出现；否则，称为非局部出现。局部和非局部的区分也适用于其他任何可包含声明的语法构造。作用域规定了名字在程序中的可见范围，即在哪些部分可以引用或访问该名字绑定的实体。在图 6.1 的程序中，i 在两个过程中都有声明，但它们的使用互相独立。

作用域分为静态作用域和动态作用域。**静态作用域**是编译时确定的，也称为词法作用域（lexical scope）。变量的作用域由代码的静态结构决定，即通过程序源代码中的层次结构（如函数、块）来确定。查找变量时，编译器会根据函数或语句块的嵌套关系，从内向外逐层查找变量的定义。**动态作用域**是在程序运行时确定的。变量的作用域取决于函数调用的顺序，而不是代码的结构。查找变量时，程序会根据调用栈的顺序从调用者往回逐层查找变量的定义。大多数现代编程语言，如 C、C++、Java、Python、JavaScript 等主要采用静态作用域，而某些早期的语言如 Lisp 的早期版本及一些脚本语言主要采用动态作用域。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第222页 文件编号: 422624 页面编号: 32224538440 打印人: 8685 2026-03-09 10:23:02

即使一个名字在程序中只声明一次，该名字在程序运行时也可能代表不同的数据对象。非正式的术语“数据对象”是指保存值的存储单元。

在编程语言的语义中，通常用环境和状态来表示变量名字到值的映射。术语**环境**表示将名字映射到存储单元的函数，术语**状态**表示将存储单元映射到它所保存值的函数，如图 6.2 所示。也可以说，环境把名字映射到其左值，而状态把名字的左值映射到名字的右值。左值和右值的概念源于 C 语言，分别指变量的地址和值，因为对于赋值号左右两边的变量，被关注的分别是它们的地址和值。区别状态和环境是有意义的，赋值改变状态，但不改变环境；过程调用改变环境。

图 6.2 从名字到值的两步映射

如果环境将名字 x 映射到存储单元 s，就说 x 被**绑定**到 s。术语“存储单元”是象征性的，因为如果 x 不是基本类型，那么 x 的存储单元 s 可能是内存中的若干个字。

在学习本章时，应注意区分静态概念与其动态对应物，见表 6.1。特别是，递归过程在某一时刻可能有多个活动同时存在，每个活动中的局部变量绑定到不同的存储单元。

表 6.1 静态概念和动态概念的对应

静态概念	动态概念
过程的定义	过程的活动
名字的声明	名字的绑定
声明的作用域	绑定的生存期

6.1.3 活动记录

过程的一次执行所需的局部信息通常用一块连续的存储区来管理，这块存储区称为**活动记录**或**帧**（frame），其具体结构示例如图 6.3 所示。不同编程语言或同一语言的不同编译器可能会使用不同的域，并且这些域在活动记录中的布局也可能存在差异。此外，某些域（如参数、返回值）往往可以通过寄存器传递。

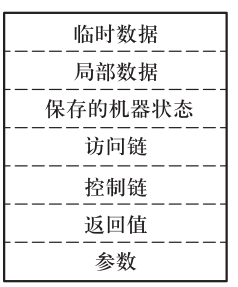


图 6.3 一般的活动记录

活动记录中各个域的用途如下。

（1）临时数据。保存临时值，如当寄存器不足以存放表达式计算的中间结果时，可以把中间结果存放在这儿。

（2）局部数据。保存过程的局部数据。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第223页 文件编号: 422624 页面编号: 32234538448 打印人: 8685 2026-03-09 10:23:02

（3）保存的机器状态。在过程调用时，保存调用前的机器状态信息，典型信息包括返回地址 [程序计数器的值，它是被调用者（callee）必须返回的地址]，以及调用者（caller）使用且在返回时必须恢复的寄存器的内容。

（4）访问链。有些语言（如支持过程嵌套定义）往往需要通过访问链来访问非局部数据，6.3 节将详细介绍。

（5）控制链。用来指向调用者的活动记录。

（6）返回值。用于存放被调用者返回给调用者的值。为提高效率，该值也常常用寄存器返回。

（7）参数。存放调用者提供的实在参数，由被调用者使用。为提高效率，实际上常常用寄存器传递参数。6.4 节将介绍各种参数传递方式的实现。

每个域的长度都可以在过程调用时确定。事实上，大多数域的长度都可以在编译时确定。一个例外是，如果过程中存在只能在执行时才能确定大小的局部数组，那么只有在执行到过程调用时才能确定局部数据域的大小。6.2.6 小节将讨论活动记录中可变长度的数据的分配方法。

活动记录没有包含过程一次执行所需的全部信息，如非局部数据就不在这个活动记录中，6.2 节和 6.3 节将介绍非局部数据的布局和访问方法。还有，在程序控制下动态分配的数据对象通常放在堆上，6.5 节将讨论堆分配。

6.1.4 局部数据的布局

假定运行时存储空间是一个连续的字节区域，其中字节是可编址内存的最小单位，每个字节由 8 位组成，多个字节形成一个机器字。机器字的字节数取决于系统的架构，如 32 位系统的机器字为 4 字节（32 位）、64 位系统的机器字为 8 字节（64 位）。机器字的长度决定了 CPU 处理数据的基本单元大小、内存寻址范围和指令集架构的设计。

多字节的数据对象存储在连续的字节中，以该区域内最小的字节地址作为对象的地址。不同的架构对多字节数据的存放顺序存在差异，通常采用大端（big endian）或小端（little endian）两种**字节序**（俗称大尾端、小尾端）。在大端模式中，高位字节存储在起始地址；在小端模式中，低位字节存储在起始地址。这种字节序的选择会影响数据在不同架构上的解释方式，因此在跨平台开发和网络通信中，需要特别考虑字节序的兼容性。

变量所需的存储空间可以由它的类型确定。基本类型的数据，如字符、整数或实数，可以用几个连续字节保存。对于数组，数组元素在分配的存储区内依次存放，以便根据下标计算元素的地址。对于记录，其域通常按声明时出现的顺序存放在分配的存储区内。

在编译时，一个过程所声明的局部变量通常按声明顺序依次分配到活动记录的局部数据域中。然而，现代编译器趋向于根据类型进行布局优化，局部变量往往先按类型分类，再按声明顺序分配。这些局部数据的地址可以用相对于活动记录中某个位置的相对地址来表示，如相对于活动记录的开始点或中部的某个特定单元。相对地址（或称为**偏移**）是指数据对象与

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第224页 文件编号: 422624 页面编号: 32244538446 打印人: 8685 2026-03-09 10:23:02

该位置的地址差，活动记录中的其他域也可以通过相对于该位置的偏移来访问。

数据的存储布局还受目标机器寻址限制的影响。例如，整数加的指令可能要求参与相加的整数必须**对齐**（alignment），即它们被放在内存中满足一定条件的位置，如能被 4 整除的地址。例如，10 个单字符的数组只需要 10 个字节，假如下面紧接着安排 1 个整数，编译器就会跳过 2 个字节后再进行分配，以保证该整数的地址是 4 的倍数。由于考虑对齐而引起的无用空间称为**衬垫区**（padding）。如果空间很宝贵，编译器可能紧凑布局数据，使得没有任何衬垫区出现，但是运行时可能要执行一些额外的指令来取出这些紧凑布局的数据，然后才能对它们进行操作。

6.1.5 程序块

程序块（block，又翻译为分程序）是本身含有局部变量声明的语句。程序块的一个特点是它的嵌套结构，即不可能出现程序块 B_1 先于 B_2 开始执行，又先于 B_2 结束执行的情况，这种嵌套性又称作程序块结构。程序块结构中声明的作用域按最接近的嵌套规则给出。图 6.4 是有嵌套程序块的 C 程序，其中变量声明的作用域见表 6.2。4 个打印语句按照 B_2、B_3、B_1 和 B_0 的次序执行，即控制离开这些程序块的次序。在这些程序块中，a 和 b 的值分别为：

```
2    1
0    3
0    1
0    0
```

```
1   main() {                                /* begin of B0 */
2   int a=0;
3   int b=0;
4   {                                   /* begin of B1 */
5       int b=l;
6       {                           /* begin of B2 */
7           int a=2;
8           printf( " %d %d\n" ,a,b);
9       }                           /* end of B2 */
10      {                           /* begin of B3 */
11          int b=3;
12          printf("%d %d\" ,a,b);
13      }                           /* end of B3 */
14      printf(" %d % d\n" ,a,b);
15  }                                   /* end of B1 */
16  printf("%d %dn" ,a,b);
17 }                                        /* end of B0 */
```

图 6.4 C 程序的程序块

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第225页 文件编号: 422624 页面编号: 32254538443 打印人: 8685 2026-03-09 10:23:02

表 6.2 图 6.4 中各声明的作用域

声明	作用域
int a=0;	$B_0 - B_2$
int b=0;	$B_0 - B_1$
int b=1;	$B_1 - B_3$
int a=2;	B_2
int b=3;	B_3

如果过程中有嵌套的程序块，编译器在分配存储空间时需要为嵌套块中声明的变量预留空间。例如，对于图 6.4 中的程序，可以按图 6.5 分配存储空间，局部变量 a 和 b 的下标用来标识其声明所在的程序块。注意，a_2 和 b_3 可以重叠分配，因为它们所在的程序块不会同时活跃。

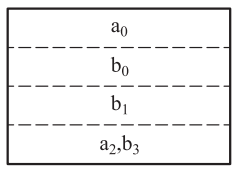


图 6.5 图 6.4 程序所声明变量的存储单元

需要注意的是，在确定过程所需的局部存储空间时，编译器会稳妥地假设过程执行时会走遍所有可能的控制路径，即假定条件语句的 then 和 else 部分都会执行，循环体的各语句也都会执行。因此，即便过程内有嵌套的程序块，活动记录中局部数据域的大小依然可以在编译阶段静态确定，不受不同控制路径的影响。

6.2 全局栈式存储分配

6.1 节介绍了单个活动记录内数据的布局，本节介绍程序运行时所有活动记录在存储空间的组织，并描述过程的目标代码怎样访问绑定到局部名字的存储单元。本节讨论的栈式存储分配可用于 C、FORTRAN 和 Pascal 等面向过程的语言，也适用于 Java 和 C++ 等面向对象的语言。

6.2.1 运行时内存的划分

在逻辑地址空间（进程地址空间），程序运行时的内存布局如图 6.6 所示，由代码区和若干个数据区组成。例如，在 Linux 操作系统上，C 编译器就是按这种方式将内存细分的。

代码区存储程序的目标代码，大小在编译时确定，并且在运行时不会改变，通常置于内存的低地址区。为了提高安全性，代码区设置为只读，以防止程序意外修改代码。

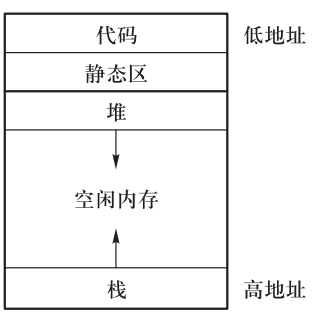


图 6.6 运行时内存空间的划分

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第226页 文件编号: 422624 页面编号: 32264538441 打印人: 8685 2026-03-09 10:23:02

此外，代码区是可共享的，多个进程可以共享诸如系统库的相同代码段。

静态数据区用于存储全局常量、全局变量和静态变量，还有编译器产生的辅助信息，如垃圾收集机制所需的信息。这些数据的存储大小在编译时都可以确定，因此把它们安排在静态确定的数据区中。将数据放在静态区的一个优势是，编译时可以将这些数据的地址直接嵌入目标代码中，从而提高运行时访问数据的速度。在 FORTRAN 语言的早期版本中，由于不支持递归，所有数据都可以静态分配。静态数据区分为初始化数据区（如.data）和未初始化数据区 (如.bss)。

堆用于动态内存分配，栈用于存储活动记录。为了在运行时充分利用存储空间，栈和堆一般安排在地址空间剩余部分的两端，相向增长：栈从高地址向低地址增长，堆从低地址向高地址增长。共享库在程序运行时根据需要动态加载、随机放置到进程地址空间中。

对于 C、FORTRAN 和 Pascal 等语言，因为过程递归调用次数不可预知，所以无法静态分配活动记录，尽管每个活动记录的大小可以静态确定。当过程调用结束后，其活动记录被回收，因此可以将所有尚未结束的过程调用的活动记录按栈的方式管理。这种栈式存储分配策略将在 6.2.2 小节开始讨论。

许多编程语言允许程序员在程序控制下分配和释放存储块，如 C 语言的函数 malloc() 和 free() 用于此目的。这些存储块中的数据对象的生存期可能比要求分配这些存储块的过程更长，因此它们的生存期不遵守栈式规则。堆用来管理这类生存期比过程长的数据。数据放在栈上比放在堆上的开销要小一些，这是由它们对数据的分配和回收方式决定的，6.5 节专门讨论堆管理。

此外，Linux 进程通常启用了地址空间布局随机化（address space layout randomization，ASLR），在每次运行时随机化堆、栈、共享库等内存区域的起始地址，以提升程序安全性，防止攻击者通过已知的内存地址进行缓冲区溢出等漏洞的利用。

第 10 章还会对运行时内存空间的组织做更详细的介绍。

6.2.2 活动树和运行栈

对于常见的面向过程的语言，如果 a 和 b 都是过程活动，那么它们的生存期或者嵌套，或者无重叠。也就是说，如果控制在离开 a 之前进入 b，那么控制在离开 b 之后才能离开 a。由此可知，可以用树来描绘控制进入和离开活动的次序，这样的树称为**活动树**。在活动树中：

（1）每个结点代表一个过程活动；

（2）根结点代表主过程的活动；

（3）结点 a 是结点 b 的父结点，当且仅当控制流从活动 a 进入活动 b；

（4）结点 a 处于结点 b 的左边，当且仅当活动 a 的生存期先于活动 b 的生存期。

因为结点和活动一一对应，所以当控制处于某结点代表的活动时，直接说控制处于该结点。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第227页 文件编号: 422624 页面编号: 32274538449 打印人: 8685 2026-03-09 10:23:02

【例 6.1】 图 6.1 所示程序的活动树画在图 6.7 中，其中过程名都用其第一个字母表示，括号中的数据是活动的实参。程序的控制流对应于活动树的后根遍历。

【例 6.2】 图 6.8 给出了控制进入 q (2,3) 代表的活动时，已经执行完毕和正在执行的活动。其中已经执行完毕的活动在虚线的下端，正在执行、尚未结束的活动在根结点到结点 q (2,3) 的这条实线路径上。尚未开始的活动图中没有画出。

从图 6.8 可以看出，当前活跃的过程活动可以保存在一个栈中。当活动开始时，把这个活动的结点压入栈中，当活动结束时，把这个活动的结点从栈中弹出。这样的栈被称为**控制栈**。对于图 6.8 的情况，控制栈的内容从栈底到栈顶依次为

$$\text{m, q (1, 9), q (1, 3), q (2, 3)}$$

如果控制栈中的信息包含过程活动的活动记录，控制栈就成为活动记录栈，通常称为**运行栈**。当一个过程被调用时，它的一个新活动记录被压入栈，局部变量被绑定到其中的存储单元; 当对应这次调用的活动终止时，该活动记录被退栈，局部变量不再绑定到其中的存储单元，恢复原来的绑定（如果有）。因为一个过程的每次调用都会引起一个新的活动记录进栈，所以过程的每次活动都会把局部变量绑定到新的存储单元。

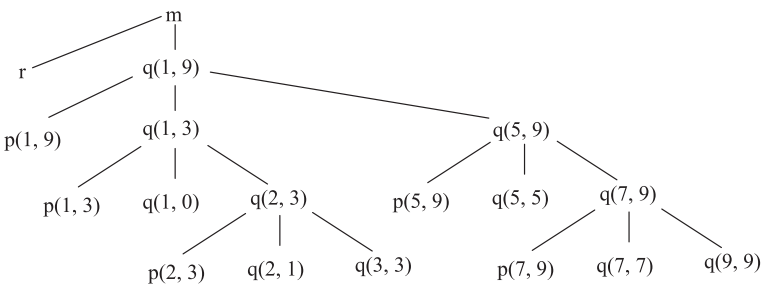


图 6.7 图 6.1 程序的活动树

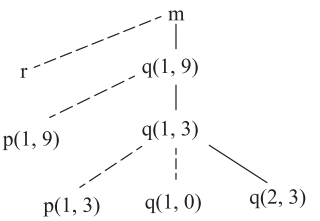


图 6.8 控制栈包含实线上的结点

【例 6.3】 图 6.9 表示当控制流通过如图 6.7 所示的活动树时，活动记录压入运行栈和从运行栈退出的几个快照（snapshot），树上的虚线仍然引向已经结束的活动。注意，因为数组 a 是全局的，它分配在静态区，所以没有出现在该图上。程序执行开始时有过程 m 的活动，当控制到达 m 体中第一个调用时，过程 r 被激活，它的活动记录被压入栈。当控制从该活动返回时，栈顶活动记录被退栈，栈中仅剩下 m 的活动记录。在 m 的继续执行过程中，当控制到达以 1 和 9 为实参的对过程 q 的调用时，q 的一个活动记录被压入栈。

图 6.9 的最后两个快照之间有多个活动发生。在最后一个快照中，活动 p (1, 3) 和 q (1, 0) 已经开始并终止在 q (1, 3) 的生存期中，它们的活动记录也都已经压栈并退栈，因此只留下 q (1, 3) 的活动记录在栈顶。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第228页 文件编号: 422624 页面编号: 32284538447 打印人: 8685 2026-03-09 10:23:02

6.2.3 调用序列和返回序列

从先前的介绍可以知道，过程调用和过程返回都需要执行一些代码来管理运行栈：分配和回收活动记录、保存或恢复机器状态等。在过程调用时执行的“分配活动记录，以及把信息填入其域中”的代码称为**过程调用序列**；而在过程返回时执行的“恢复机器状态，回收活动记录，以及让调用过程继续执行”的代码称为**过程返回序列**。

即使是同一种语言，过程调用序列、过程返回序列和活动记录的布局也会因实现而异。过程调用序列的代码常常分成两部分，分别位于调用过程和被调用过程中。过程调用序列在这两者之间的划分也不是唯一的。源语言、目标机器和操作系统强加的约束可能使得某种方法比另一些方法更合适。过程返回序列也是如此。

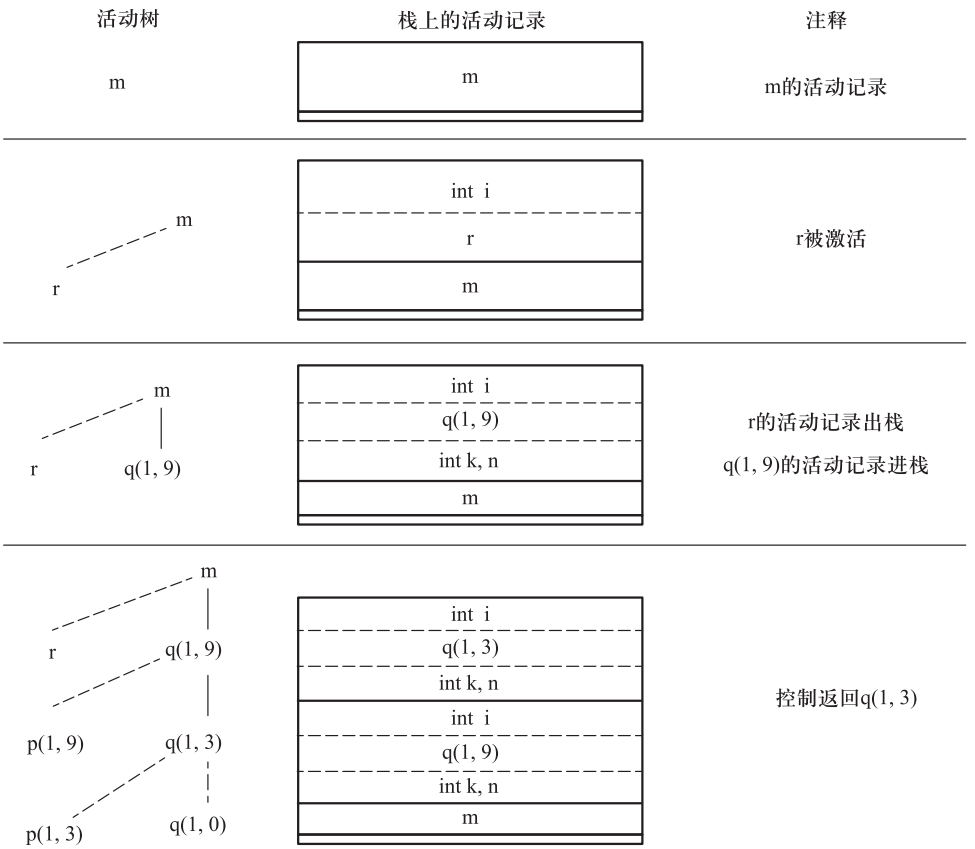


图 6.9 活动记录在栈中的分配（栈向上长）

下面一些原则对设计调用序列、返回序列和活动记录布局是有益的。

（1）调用者和被调用者之间交流的数据一般放在被调用者活动记录的开始处，尽可能靠

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第229页 文件编号: 422624 页面编号: 32294538445 打印人: 8685 2026-03-09 10:23:02

近调用者的活动记录。这样做便于调用者把实参的值放到自身活动记录的顶端，而不需要此刻就建立被调用者完整的活动记录，甚至不用知道它的布局。这样做还允许使用参数个数可变的过程，如 C 语言的标准库函数 printf()（这个问题后面再讨论）。被调用者也知道返回值应该放在靠近调用者活动记录的地方，不管有多少个实参，它们都正好在返回值的下面。

（2）固定长度的项通常放在活动记录的中间。它们一般包括控制链、访问链和机器状态域。如果每次调用时正好保存机器状态中同样的成分，那么可以执行同样的代码来完成机器状态的保存和恢复。而且，如果机器状态信息是标准化的，那么出现错误时，调试器等程序可以很轻松地解读运行栈的内容。

（3）对编译时不能及时知道大小的一些项，放在活动记录的末端。大多数局部变量有固定的长度，这些长度能够由编译器在检查变量类型时确定。但是，有些局部变量的大小在程序运行前一直不能确定，最常见的例子是动态大小的数组（大小依赖于过程参数的数组）。还有，所需的临时数据空间的大小通常依赖于代码生成阶段怎样尽可能地把临时数据存放在寄存器中。因此，虽然临时数据空间的大小在编译时最终可以知道，但它可能不是在中间代码生成时知道的。

图 6.10 给出了一个调用者和被调用者怎样协作管理栈的建议示例（该语言无须访问链）。**栈指针寄存器** top_sp 指向栈顶活动记录的末端，用来跟踪调用栈的增长和缩减；**帧指针寄存器** base_sp 指向栈顶活动记录中控制链所在的位置，它作为访问栈顶活动记录中内容的基地址。表 6.3 列出了 x86-32/64、ARM64、LoongArch 和 RISC-V 架构中主要寄存器的名称及其功能。

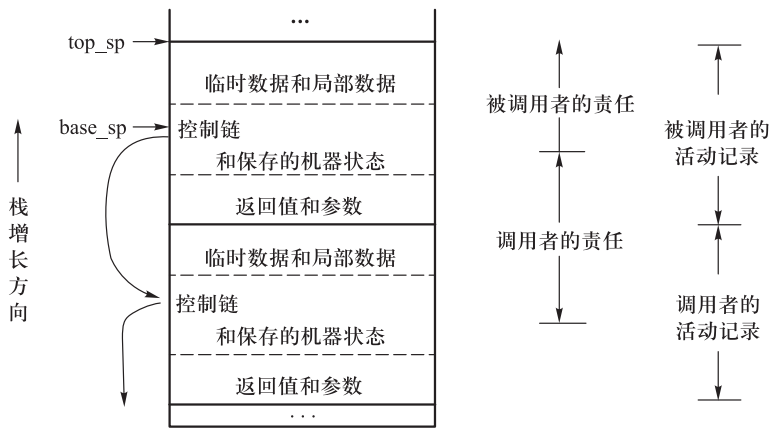


图 6.10 调用者和被调用者之间的任务划分

假定过程 p 调用过程 q，调用序列如下：

（1）p 计算实参，依次将它们压入栈顶，也就是放到 q 的活动记录中，并在栈顶上留出

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第230页 文件编号: 422624 页面编号: 32304538443 打印人: 8685 2026-03-09 10:23:02

放返回值的空间。在这些操作过程中，top_sp 的值在不断减小。

表 6.3 不同架构中主要寄存器资源对照表

主要寄存器	x86-32/64 架构	ARM64 架构	LoongArch 架构	RISC-V 架构
栈指针寄存器（top __ sp）	esp/rsp	SP(x31)	r3(sp)	sp
帧指针寄存器（base __ sp）	ebp/rbp	FP(x29)	r22(fp/s9)	fp(s0)
返回地址寄存器	eip/rip	LR (x30)	r1(ra)	ra

（2）p 把返回地址压入 q 的活动记录中，把控制转到 q。

（3）q 将要保存的寄存器（base_sp 除外）的值和其他机器状态信息压入栈，然后把 base_sp 的当前值（p 的 base_sp）压入栈，并把当前 top_sp 的值作为自己 base_sp 的值。

（4）q 根据局部数据域和临时数据域的大小来减小 top_sp 的值，也就是进行局部数据和临时数据的空间分配，并初始化它的局部数据，开始执行过程体，如图 6.10 所示。

在上述步骤（3）中，把 p 的 base_sp 压入栈就是建立控制链，由于控制链是按运行时过程调用关系从被调用者的活动记录指向调用者的活动记录，因此控制链也称为**动态链**。

返回序列如下：

（1）q 把返回值置入活动记录中存放返回值的地方。

（2）与调用序列的步骤（4）相对应，q 增加 top_sp 的值。

（3）q 恢复寄存器（包括 base_sp）和机器状态，把控制转到 p。

（4）与调用序列的步骤（1）相对应，p 增加 top_sp 的值，并取出返回值。

【例 6.4】 一个 C 语言函数如下：

```
1  void func(long i) {
2      long j;
3      j=i-1;
4      func(j);
5  }
```

为使生成的汇编代码尽可能简洁，这里使用了一个不会终止的递归函数。在 x86/Linux 32 位系统上用版本标识为 GCC: (GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release) 的 GCC 编译器，为该函数生成的汇编代码（略去问题无关部分）及注释如图 6.11 所示。

从该汇编代码可以分析出该函数一个活动记录的内容，如图 6.12 所示。其中，esp 是栈指针寄存器（执行“movl 8(%ebp),%edx”指令时 esp 所指的位置就是图中所标明的位置），ebp 是帧指针（基地址）寄存器。

在这个例子中，调用序列的代码在调用者的部分是 pushl %eax 和 call func，在被调用者的部分是 pushl %ebp、movl %esp,%ebp 和 subl $4,%esp。返回序列的代码在被调用者的部分是 leave 和 ret，在调用者的部分是 addl $4,%esp。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第231页 文件编号: 422624 页面编号: 32314538441 打印人: 8685 2026-03-09 10:23:02


```
1  func:
2      pushl %ebp                  // 将旧的帧指针（基地址）压栈，即保存控制链
3      movl %esp,%ebp               // 将当前栈（顶）指针作为帧指针
4      subl $4,%esp                // 为局部变量j分配空间
5      movl 8(%ebp),%edx            // 取形参i的值到寄存器edx，i的地址是8(%ebp)
6      decl %edx                   // i - 1
7      movl %edx, -4(%ebp)          // i - 1的值赋给 j，分配给j的地址是-4(%ebp)
8      movl -4(%ebp),%eax           // 和下一条指令一起，完成将实参j的值压栈
9      pushl %eax
10     call func                   // 调用func函数，将返回地址压栈，修改程序计数器
11     addl $4,%esp                 // 将栈指针寄存器的值恢复到参数压栈前的位置
12 .L1:
13     leave                       // 相当于movl %ebp, %esp和popl %ebp
14     ret                         // 相当于popl %eip，将esp恢复到调用前参数压栈后的位置，返
           回调用者。
15                                 // eip存放下一步执行指令的地址。
```

图 6.11 例 6.4 中 func 函数的 x86-32 位汇编代码（GCC 2.x）

注意，在较高版本的 GCC 编译器中，为局部数据所分配的空间往往会超过实际局部变量所需的大小。这种设计一方面是为了满足硬件架构对栈对齐的要求（通常是 16 字节对齐），避免未对齐造成的性能损失；另一方面是预留少量空间以支持调试、异常处理或后续指令生成中的优化需求。此外，为提高参数传递的效率，GCC 编译器通常不会直接用 push 指令将参数逐个压栈，而是先修改 esp 的值，一次性空出所需的参数区，然后通过 movl 指令将实参逐个进栈。这种做法避免了多次调用 push，减少内存操作开销；同时，顺序存储参数也有助于提高指令流水线的效率。随着硬件架构和编译技术的发展，寄存器传递参数正逐步成为主流方式。有兴趣的读者可以参考相关资料，在 6.2.4 小节也将举例讨论。

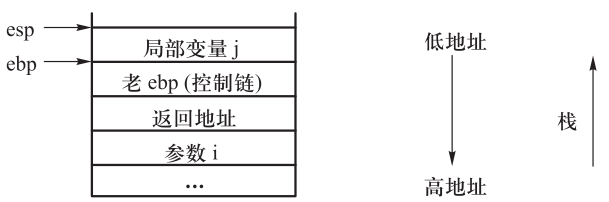


图 6.12 活动记录的内容及相关信息

6.2.4 不同计算机架构下寄存器在过程调用中的使用

在不同的计算机架构中，由于寄存器资源和命名的差异，过程调用对寄存器的使用也各不相同。前面表 6.3 列出了几种主要架构中栈指针、帧指针和返回地址寄存器的对照信息。在

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第232页 文件编号: 422624 页面编号: 32324538449 打印人: 8685 2026-03-09 10:23:02

此基础上，本小节将关注几种 64 位架构在参数传递和返回值处理方面的寄存器使用约定，并通过 GCC 生成的汇编码实例，说明这些架构在汇编码实现上的差异，为理解过程调用性能优化提供基础。

1. 寄存器在传递参数和返回值中的使用约定

除了硬件架构，寄存器的使用约定还受操作系统和应用程序二进制接口（application binary interface，ABI）的影响。

Windows 和 Linux 操作系统在 x86-64 架构上的过程调用约定差异较大，而在 ARM64、LoongArch 和 RISC-V 等现代架构上，二者的调用约定趋于一致。

（1）x86-64 架构。Windows 系统遵循微软的 x64 调用约定，它规定前四个整数或指针类型的参数通过 rcx、rdx、r8、r9 寄存器传递，浮点数参数则通过 xmm0 到 xmm3 寄存器传递。而 Linux 系统遵循 System V AMD64 ABI 调用约定，它规定前六个整数或指针类型的参数通过 rdi、rsi、rdx、rcx、r8 和 r9 寄存器传递，浮点数参数通过 xmm0 到 xmm3 寄存器传递。

如果参数的数量超过上述约定的寄存器的数量，多余的参数将通过栈传递。对于返回值，Windows 和 Linux 都通过 rax（整数或指针类型）或 xmm0（浮点数）寄存器返回。

（2）ARM64 架构 (AArch64)。AArch64 的 32 个通用寄存器 x0~x31 分为 4 组，x0~x7 用于传递整数或指针类型的参数，x9~x15 是调用者保存（caller-saved）的寄存器，x19~x29 是被调用者保存（callee-saved）的寄存器，剩下的是特殊目的寄存器。其中，x8 是间接结果寄存器，用于传递间接结果的地址位置（如大型结构的位置）；x16 和 x17（分别是 IP0 和 IP1）是过程内部调用的临时寄存器；x18 是平台寄存器，保留供 ABI 使用；x29~x31 在表 6.3 中已介绍，依次是帧指针寄存器、链接寄存器（link register，LR，保存返回地址）和栈指针寄存器。

浮点数参数通过 v0~v7 寄存器传递，被调用者必须保存 v8~v15 的内容，以确保调用者在调用返回后可以继续使用这些寄存器的值。而对于 v16~v31 寄存器，则由调用者在调用前将这些寄存器中需要保留的值保存到栈上，并在调用结束后从栈中恢复。

对于返回值，整数或指针类型的值通过 x0 寄存器返回，而浮点值则以适当方式在 s0（单精度）、d0（双精度）或 v0（128 位矢量）中返回。

（3）LoongArch 架构。龙芯架构 ELF psABI 规范约定，整数或指针类型的参数通过寄存器 r4~r11（别名为 a0~a7）传递；浮点数参数通过寄存器 f0~f7（别名为 fa0~fa7）传递；剩余的参数通过栈传递。对于返回值，LoongArch 没有专门的返回值寄存器，一般通过 a0~a1 返回单个或两个整数，fa0~fa1 返回单个或两个单精度或双精度浮点数。

（4）RISC-V 架构。RISC-V 架构中，整数或指针类型的参数通过寄存器 x10~x17（别名为 a0~a7）传递；浮点数参数通过寄存器 f10~f17（别名为 fa0~fa7）传递；剩余的参数通过栈传递。对于返回值，一般通过 a0~a1 返回单个或两个整数，fa0~fa1 返回单个或两个单精度或双精度浮点数。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第233页 文件编号: 422624 页面编号: 32334538447 打印人: 8685 2026-03-09 10:23:02

2. 不同架构下汇编程序示例及说明

针对例 6.4 中的函数，我们使用较新的 GCC 编译器为之生成不同目标架构下的汇编代码。通过对比这些汇编代码，深入了解各架构在过程调用实现上的具体差异。

（1）**x86-64 架构**。在 Linux 系统上，用 GCC 11.4.0 编译生成的汇编代码（略去无关部分）及添加的注释如图 6.13 所示（用-fno-pie 选项可以关闭地址随机化，否则会生成诸如 ___x86.get_pc_thunk.ax 的调用指令用于将 eax 寄存器中的值加载到 rip）。

```
1  func:
2      pushq %rbp                  // 将旧的帧指针（基地址）压栈，即保存控制链
3      movq %rsp,%rbp               // 将当前栈指针作为帧指针
4      subq $32,%rsp               // 对齐及为局部变量j分配空间
5      movq %rdi, -24(%rbp)         // 将通过寄存器rdi传递的参数i的值存入栈中-24(%rbp)处
6      movq -24(%rbp),%rax          // 将形参i的值加载到寄存器rax
7      subq $1, %rax               // 计算i - 1并存入rax
8      movq %rax, -8(%rbp)          // i - 1的值赋给j，分配给j的地址是-8(%rbp)
9      movq -8(%rbp),%rax           // 将实参j的值加载到寄存器rax
10     movq %rax, %rdi              // 通过寄存器rdi传递实参j
11     call func                   // 调用func函数，将返回地址压栈，修改程序计数器
12     nop
13     leave                       // 相当于movq %rbp, %rsp和popq %rbp
14     ret                         // 相当于popq %rip，将rsp恢复到调用前，返回调用者
```

图 6.13 例 6.4 中 func() 函数的 x86-64 汇编代码（GCC 11.4.0）

这个版本的汇编代码有以下几处值得注意。

① 局部数据空间分配得更多了。由第 4 行可知，分配的局部数据空间为 32 字节，多于局部变量的实际需求。这主要是为了满足栈对齐并预留一定的额外空间。从 GCC 4.5 版本开始，栈上的数据对齐要求从 4 字节提升到 16 字节。这是出于程序优化的需要，例如在 x86 平台上，SSE（streaming SIMD extensions）指令要求操作的数据在内存中必须是 16 字节对齐的。通过栈对齐，可以减少在将数据加载到寄存器时所需的对齐修正开销，从而充分发挥 SIMD 指令的性能，尤其是在处理向量化数据时更为重要。

② 通过寄存器传参来提升效率。在第 9~10 行，实参 j 的值先被加载到寄存器 rax，再通过寄存器 rdi 传递给被调用函数 func()。而在第 5 行，通过 rdi 传入的参数 i 会先被取出并暂存到栈上，以释放 rdi 寄存器，使其在函数内部可用于其他计算或传递额外的参数。

（2）**ARM64 架构**。若 GCC 版本仍为上面的 11.4.0，在 ARM64 平台或在 x86-64 机器上使用交叉编译器[①]进行编译，可以生成图 6.14 所示的 ARM64 汇编代码。

① 可以通过 sudo apt install gcc-aarch64-linux-gnu 安装交叉编译器。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第234页 文件编号: 422624 页面编号: 32344538445 打印人: 8685 2026-03-09 10:23:02


```
1       .align  2
2       .global func        .type   func, %function
3   func:
4       stp     x29, x30, [sp, -48]! // 将x29和x30的值压栈，sp减去 48 并更新sp
5       mov     x29, sp              // 设置x29为新的sp
6       str     x0, [sp, 24]         // 将寄存器x0(参数i)的值存入sp+24单元
7       ldr     x0, [sp, 24]         // 将sp+24单元的值加载到x0
8       sub     x0, x0, #1           // x0=x0-1，即x0=i-1
9       str     x0, [sp, 40]         // 将x0的值存入sp+40（局部变量j）
10      ldr     x0, [sp, 40]         // 加载sp+40单元的值（实参j）到x0
11      bl      func                 // 跳转到func函数，返回地址保存到x30
12      nop
13      ldp     x29, x30, [sp], 48   // 从栈中恢复FP（x29）和LR（x30）
14      ret
15      .size func, .-func
```

图 6.14 例 6.4 中 func() 函数的 ARM64 汇编代码（GCC 11.4.0）

尽管指令的具体形式与图 6.14 不一样，但是在代码结构的语义上是类似的：第 4 行的 stp 指令与图 6.13 的第 2（保存控制链）行和第 4（分配局部数据区域）行对应，第 8 行与图 6.13 的第 3 行对应（设置新的帧指针），第 8~13 行与图 6.13 的第 5~11 行对应。函数的形参 i 通过寄存器 x0 传入，然后暂存到 sp+24 单元；局部变量 j 保存在 sp+40 单元。

不同的是，ARM64 架构下返回地址（x30）的保存由被调用者通过 stp 指令完成（callee-saved），而 x86 架构下则由调用者通过 call 指令完成（caller-saved）；局部数据区的分配在 ARM64 架构下由 stp 完成，而 x86 架构下则由独立的 subl 指令完成；局部数据的寻址在 ARM64 架构下是以栈指针（寄存器 sp）为基址，而 x86 架构下是以帧指针（寄存器 rbp）为基址。

（3）**龙架构（LoongArch）**。在龙架构机器上或者获取交叉编译工具链[①]，使用 GCC [如 LoongArch GNU toolchain rc1.5 (20240802)] 编译器对例 6.4 的 C 程序文件编译，可以生成图 6.15 所示的龙架构汇编代码（删去无关的部分）。龙架构具有 RISC 指令架构的典型特征，指令长度固定且编码格式规整，绝大部分指令只有两个源操作数和一个目的操作数，仅有 load/store 访存指令可以访问内存，其余指令的操作对象均是寄存器和立即数。

从图 6.15 可见，与 ARM64 一样，返回地址寄存器（r1）是 callee-saved 的，分配的栈帧大小都是 48 字节。但是，与 ARM64 架构不同的是，局部数据的寻址是相对于帧指针的偏移，而非栈指针的偏移，这一点与 x86 架构相似。函数的参数通过 r4 寄存器传递。函数的调用与返回都是通过跳转指令来实现的。

（4）**RISC-V 架构**。若 GCC 版本仍为上面的 11.4.0，在 RISC-V 平台或者在 x86-64

① 可以访问 https://www.loongnix.cn/zh/toolchain/GNU/获取交叉编译工具链。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第235页 文件编号: 422624 页面编号: 32354538442 打印人: 8685 2026-03-09 10:23:02

机器上使用交叉编译器[①]进行编译，可以生成图 6.16 所示的 RISC-V 汇编代码。该代码结构与图 6.15 的龙架构汇编代码基本类似，但在代码段对齐、寄存器名、汇编表示上略有不同，这里不再赘述。

```
1       .align    2
2       .globl    func       .type    func, @function
3   func:
4       addi.d  $r3,$r3,-48      # 将栈指针寄存器$r3的值减去48，用于分配栈帧
5       st.d    $r1,$r3,40       # 将返回地址寄存器$r1的值存入当前栈偏移40处
6       st.d    $r22,$r3,32      # 将旧的帧指针存入当前栈偏移32处，即保存控制链
7       addi.d  $r22,$r3,48      # 将当前栈帧基址存入帧指针寄存器$r22
8       st.d    $r4,$r22,-40     # 将形参从$r4存入栈帧偏移-40处(i的存储位置)
9       ld.d    $r12,$r22,-40    # 加载i
10      addi.d  $r12,$r12,-1     # i-1
11      st.d    $r12,$r22,-24    # i-1的值赋给局部变量j（相对栈帧偏移-24）
12      ld.d    $r4,$r22,-24     # 加载j
13      bl      func             # 递归调用，并传递参数$r4
14      nop                      # 延迟槽
15      ld.d    $r1,$r3,40       # 恢复返回地址
16      ld.d    $r22,$r3,32      # 恢复$r22
17      addi.d  $r3,$r3,48       # 增加栈指针$r3 的值（释放栈帧）
18      jr      $r1              # 使用寄存器 $r1 的值作为返回地址，跳转回调用者
```

图 6.15　例 6.4 中 func 函数的龙架构汇编代码（GCC 8.3.0）

3. 过程定义的不同特征对汇编代码的影响

对例 6.4 中的 func() 函数，可以通过小的修改来观察编译器生成的汇编代码是如何变化的。

例如，将形参 i 和局部变量 j 声明为 int（如下面左边的代码）。用 GCC 11.4.0 在 x86-64 Linux 系统上编译，得到的汇编代码与图 6.13 大致相同，只是在第 5~10 行有所不同（如下面右边的代码）。具体而言，int 类型的参数通过 32 位寄存器 edi 传递，参数 i 和局部变量 j 在栈帧中均按 4 字节存储：

```
1  void func(int i) {
2      int j;
3      j=i-1;
4      func(j);
5  }
```

```
5       movq    %edi, -20(%rbp) // 参数i 暂存到栈
6       movq   -20(%rbp), %eax  // i加载到寄存器eax
7       subq    $1, %eax        // i-1=>eax
8       movq   %eax, -4(%rbp)   // i-1存入变量j
9       movq    -4(%rbp), %eax  // 加载j到寄存器eax
10      movq    %eax, %edi      // 通过edi传参
```

① 使用命令 sudo apt-get install gcc-riscv64-linux-gnu 安装交叉编译工具链。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第236页 文件编号: 422624 页面编号: 32364538440 打印人: 8685 2026-03-09 10:23:02


```
1       .align   1
2       .globl   func        .type    func, @function
3   func:
4       addi  sp,sp,-48      # 将栈指针 sp 向下移动 48 字节，分配栈帧
5       sd    ra,40(sp)      # 将返回地址寄存器 ra 的值保存到栈的偏移量40处
6       sd    s0,32(sp)      # 将旧的帧指针（s0）存入栈的偏移量32处，即保存控制链
7       addi  s0,sp,48       # 将当前栈帧基址存入帧指针寄存器s0
8       sd    a0,-40(s0)     # 从寄存器a0取形参i的值暂存到栈，i的地址是-40(s0)
9       ld    a5,-40(s0)     # 将形参i的值加载到寄存器a5
10      addi  a5,a5,-1       # 计算i-1并存入a5
11      sd    a5,-24(s0)     # 将a5的值存入局部变量j（地址是-24(s0)）
12      ld    a0,-24(s0)     # 将j的值存入寄存器a0
13      call  func           # 调用func
14      nop
15      ld    ra,40(sp)      # 从栈中恢复返回地址ra
16      ld    s0,32(sp)      # 从栈中恢复栈基址寄存器s0
17      addi  sp,sp,48       # 将栈指针sp向上移动48字节，释放栈帧
18      jr    ra             # 跳转到返回地址ra，返回到调用者
```

图 6.16 例 6.4 中 func() 函数的 RISC-V 架构汇编代码（GCC 11.4.0）

如果将函数的返回类型改为 int 并增加 return 语句（如下面左边的代码），则生成的汇编代码只是将图 6.13 中第 12 行的 nop 替换为下面右边的 3 行，其中 int 型返回值通过 32 位寄存器 eax 传递：

```
1  int func(long i) {
2      long j;  j=i-1;
3      return i+ func(j);
4  }
```

```
12      movl  %eax, %edx          // 返回值保存到edx
13      movq -24(%rbp), %rax      // 加载i的值到rax
14      addl  %edx, %eax          // eax存放i+func(j)
```

如果按早期 C 语言的参数声明方式进行声明，并将形参声明为 int（如下面左边的代码），则生成的汇编码将图 6.13 中第 5～10 行替换为下面右边的代码：

```
1  void func(i)
2  int i;
3  {
4      long j;
5      j=i-1;
6      func(j);
7  }
```

```
5       movq    %edi, -20(%rbp) // 参数i暂存到栈
6       movq    -20(%rbp), %eax // i加载到寄存器eax
7       subq    $1, %eax        // i-1=>eax
8       cltq                    // 类型提升 int=>long
9       movq    %rax, -8(%rbp)  // long类型的i-1存入变量j
10      movq    -8(%rbp), %rax  // 加载j到寄存器rax
11      movq    %rax, %rdi      // 通过rdi传参
12      movl    $0, %eax
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第237页 文件编号: 422624 页面编号: 32374538448 打印人: 8685 2026-03-09 10:23:02

有兴趣的读者可以继续做些小修改并在不同的平台编译，查阅编译器、汇编指令等相关的文档，以深入理解这些变化背后的原因和编译器优化的原理。

6.2.5 参数个数可变

在过程参数个数可变的情况下，6.2.3 小节介绍的调用序列和返回序列依旧适用。以 C 语言的标准库函数 printf() 为例，每次调用 printf() 时，调用的实参个数是清楚的，编译器产生将这些实参逆序压栈或通过寄存器传参的代码。尽管被调用函数 printf() 并不预先知道参数的个数，但是它要能准确知道第一个参数的位置。因此，printf() 的实现首先取第一个参数，即格式控制字符串，然后分析它的格式控制要求，根据格式控制中的格式说明，到栈中取第二个和第三个参数等。

【例 6.5】 一个 C 语言程序文件 print.c，由如下函数组成：

```
1  void print()
2  {
3      printf("%d,%d,%d");
4  }
```

用 gcc -S 编译该程序文件 print.c，会产生警告信息，但是仍然会生成汇编代码：

```
1  $ gcc -S print.c
2  print.c: In function 'print':
3  print.c:3:5: warning: implicit declaration of function 'printf' [-Wimplicit-function-
           declaration]
4       printf("%d,%d,%d");
5       ^~~~~~
6  print.c:3:5: warning: incompatible implicit declaration of built-in function 'printf'
7  print.c:3:5: note: include '<stdio.h>' or provide a declaration of 'printf'
8  print.c:3:14: warning: format '%d' expects a matching 'int' argument [-Wformat=]
9       printf("%d,%d,%d");
10              ~^
11 ......
```

为了方便对比，下面给出生成的 x86-64 和 ARM 的汇编代码：

编译输出信息的第 3 行警告是因为没有声明 printf，而第 8 行指出缺少与格式控制字符串中的格式说明%d 相对应的 int 实参表达式。从给出的 x86-64 和 ARM 汇编代码来看，尽管具体的指令形式不一样，但是汇编代码的组织结构和语义是类似的。具体而言，第 5~6 行是保存控制链并设置基址寄存器为当前活动记录的基址，第 7~9 行是通过寄存器传参并调用 printf，第 10 行是没有返回值的占位指令 nop，第 11 行是恢复基址寄存器为调用者活动记录的基址。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第238页 文件编号: 422624 页面编号: 32384538446 打印人: 8685 2026-03-09 10:23:02


```
// print的x86-64 汇编
.LC0:
    .string "%d,%d,%d"
print:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $.LC0, %edi
    movl    $0, %eax
    call    printf
    nop
    popq    %rbp
    ret
```

```
// print的ARMv8-a 汇编
.LC0:
    .string "%d,%d,%d"
print:
    stp     x29, x30, [sp, -16]!
    mov     x29, sp
    adrp    x0, .LC0
    add     x0, x0, :lo12:.LC0
    bl      printf
    nop
    ldp     x29, x30, [sp], 16
    ret
```

6.2.6 栈上可变长度数据

在现代编程语言中，通常将编译时无法确定大小的数据分配在堆上（如 Java 语言）。然而，在某些情况下，编译器可以把数组、对象或其他未知大小的结构分配在栈上（如 C 语言）。这样做的优势是避免了显式调用回收函数或由垃圾收集器回收的开销，从而提升性能。需要注意的是，只有那些在过程返回后不再访问的局部数据适合分配在栈上。这里以动态数组为例，说明编译时大小未知的数据在栈上的分配，该方案同样适用于那些局部于过程且大小依赖于过程参数的其他数据。

在图 6.17 中，局部数组的大小要等到过程激活时才能确定。过程 p 有两个不能静态确定大小的局部数组，在编译时，可以在活动记录中为这两个数组分别分配一个存放数组指针的单元。运行时，这两个数组的大小确定后，在栈顶为这些数组分配空间，并把起始地址置入存放数组指针的单元。这样，对这些数组的访问是通过活动记录中的数组指针间接进行的。

6.2.7 悬空引用

只要存储空间可以回收，就有可能出现悬空引用问题。引用某个已被回收的存储单元称为**悬空引用**（dangling reference）。使用悬空引用是一种逻辑错误，因为按大多数语言的语义，已被回收的存储单元的值是没有定义的。更糟糕的是，已被回收的存储单元可能随后被分配用来存放其他数据，因此有悬空引用错误的程序会出现难以理解的不会被捕获的错误。悬空引用在栈上和堆上都有可能出现，下面是一个在栈上出现悬空引用的示例。

【例 6.6】 在图 6.18 的 C 程序中，过程 dangle 返回指向绑定到局部名 j 的存储单元的指针。当控制从 dangle 返回到 main 时，dangle 的活动记录已回收，并可能已另有安排。因为 main 中 q 的值是这个存储单元的地址，q 被称为**悬空指针**，对 q 指向的对象的访问就是一个**悬空引用**。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第239页 文件编号: 422624 页面编号: 32394538444 打印人: 8685 2026-03-09 10:23:02

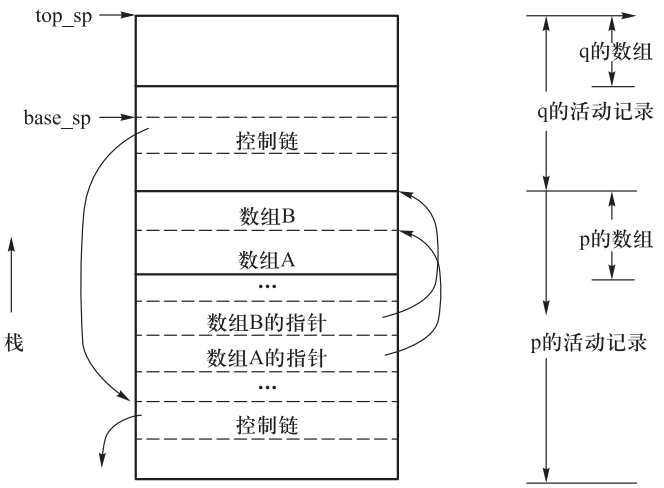


图 6.17 访问动态分配的数组

```
1  int* dangle() {
2      int j = 20;
3      return &j; // 返回局部变量j的地址
4  }
5  int main() {
6      int* q;
7      q = dangle();  // q指向无效地址
8  }
```

图 6.18 q 指向已经回收了的存储单元

6.3 非局部名字的访问

语言的作用域规则规定了如何处理非局部名字的访问。一种常用的规则是**词法作用域**或**静态作用域**规则，它仅根据程序正文静态地确定适用于一个名字的某个引用性出现的该名字的声明（简单说就是用于名字的声明）。许多语言，如 C、Java 和 Pascal 等，都使用静态作用域规则。本节首先考虑 C 语言那样的非局部名字。由于 C 语言不允许嵌套的过程声明，所以所有的非局部名字都可以静态地绑定到所分配的存储单元。

Pascal 和 ML 等语言允许过程嵌套，并使用静态作用域，确定用于名字的声明需要根据过程的嵌套层次来决定。与 C 语言不同的是，Pascal 语言的非局部名字不一定就是全局名字。当运行时访问非局部名字，首先要确定该非局部名字被绑定到的活动记录，本节讨论利用访问链寻找该活动记录的一种方法。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第240页 文件编号: 422624 页面编号: 32404538442 打印人: 8685 2026-03-09 10:23:02

另一种作用域规则是动态作用域规则，它在运行时根据当前存活的过程活动来确定用于名字的声明。使用动态作用域的语言很少，如 Lisp。作为比较，6.3.3 小节将讨论动态作用域的实现。

6.3.1 无过程嵌套的静态作用域

由于 C 语言不允许过程嵌套，如果在某个函数中对名字 a 有非局部引用，那么 a 必须作为外部变量，声明在所有函数的外面，如图 6.1 中的数组 a。函数外声明的作用域是该声明后的所有函数体，但是某个函数体中有该名字的重新声明的情况要除外。在图 6.1 中，readArray、partition 和 main 中对 a 的非局部引用都是引用第 1 行声明的数组 a。

由于没有过程嵌套，6.2 节中局部名的栈式分配策略可以直接用于像 C 这样的静态作用域语言。声明在过程外面的所有变量都可以分配在静态区，它们的存储位置在编译时都可以确定。所以，过程体中的非局部引用可以直接使用静态确定的地址。任何其他变量必定局部于栈顶的活动记录，可以通过 base_sp 指针来访问。过程嵌套会使这种方法失败，因为对非局部名字的访问需要深入栈中访问数据，这个问题在 6.3.2 小节讨论。

对非局部名字进行静态分配的一个重要好处是，程序中声明的过程可以作为参数来传递，也可以作为结果来返回（C 语言传递和返回的都是过程指针）。这是因为在静态作用域且无嵌套过程的情况下，一个过程的任何非局部名字也是所有过程的非局部名字，该名字的静态地址可以被所有过程使用，而不用管这些过程是怎样被激活的。同样，如果过程作为结果返回，该被返回的过程中对非局部名字的引用，仍然是引用静态分配给这些名字的地址。

*6.3.2 有过程嵌套的静态作用域

在 Pascal 语言的过程 p 中若没有名字 a 的声明，则适用于 p 中 a 出现的 a 的声明在 p 的某个外围过程 q 中，并且从静态程序正文看，q 中的这个 a 声明比任何其他 a 声明更靠近 p 中的这个 a 出现。

为说明问题，把图 6.1 所示快速排序程序用 Pascal 语言重写在图 6.19 中。在图 6.19 中 partition 函数声明在 quickSort 过程的里面，并增加了 exchange 过程。

修改后的快速排序程序中，过程声明的嵌套关系如下面的阶梯表示：

```
sort
    readArray
    exchange
    quickSort
```

在图 6.19 中，第 15 行的 a 出现在函数 partition 中，最接近这个 a 的外嵌 a 声明在第 2 行，它属于构成整个程序的过程。最接近的嵌套规则同样用于过程名。第 17 行中由函数

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第241页 文件编号: 422624 页面编号: 32414538440 打印人: 8685 2026-03-09 10:23:02

partition() 调用的过程 exchange 对 partition() 来说是非局部的，应用这个规则，首先检查 exchange 是否定义在 quickSort 中。因为 exchange 没有定义在 quick Sort 中，所以在主程序 sort 中寻找它。

在实现静态作用域时，需要了解过程**嵌套深度**概念。设主程序的嵌套深度为 1，从一个过程进入一个嵌套定义的过程时，嵌套深度加 1。因此，第 11 行的 quickSort 过程的嵌套深度是 2，而第 13 行的 partition 过程的嵌套深度是 3。对名字的每次出现，把它的声明所在过程的嵌套深度作为该名字的嵌套深度。在 partition 的第 15~17 行的 a、v 和 i 的嵌套深度分别为 1、2 和 3。

```
(1)  program sort(input,output);
(2)      var a:array[0..10]of integer;
(3)      x::integer;
(4)      procedure readArray;
(5)          var i:integer;
(6)          begin…a…end{readArray};
(7)      procedure exchange(i,j:integer);
(8)          begin
(9)              x:=a[i];a[i]:=a[j];a[j]:=x
(10)         end{exchange};
(11)     procedure quickSort(m,n:integer);
(12)         var k,v:integer;
(13)         function partition(y,z:integer):integer;
(14)             var i,j:integer;
(15)             begin    …a…
(16)                      …v…
(17)                      …exchange(i,j);…
(18)             end{partition};
(19)         begin…end{quickSort};
(20)     begin…end{sort}.
```

图 6.19 有过程嵌套的 Pascal 程序

过程嵌套的静态作用域的直接实现是在每个活动记录中增加一个称为访问链的指针。如果过程 p 直接嵌在过程 q 中，那么过程 p 活动记录的访问链直接指向最靠近的那个属于过程 q 的活动记录的访问链。

图 6.19 所示的程序运行时，运行栈的快照在图 6.20 中给出。为了节约空间，图 6.20 中仍然只给出过程名的第一个字母。sort 活动的访问链为空（图 6.20 中用 nil 表示），因为它不再有外围的过程。quickSort 的每个活动的访问链都指向 sort 的活动记录。在图 6.20(c) 中，parti-

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第242页 文件编号: 422624 页面编号: 32424538448 打印人: 8685 2026-03-09 10:23:02

tion(1,3) 活动记录的访问链指向最靠近的那个 quickSort 活动记录的访问链，即 quickSort(1, 3) 活动记录的访问链。

假定过程 p 的嵌套深度为 n_p，它引用一个嵌套深度为 n_a 的变量 a，$n_a \leqslant n_p$，则 a 的存储单元可以按如下方式找到：

（1）当控制在 p 中时，p 的一个活动记录肯定在栈顶。首先从栈顶的活动记录开始，追踪访问链 $n_p - n_a$ 次（$n_p - n_a$ 的值可以在编译时计算）。如果一个活动记录的访问链正好指向另一个活动记录的访问链，那么访问链的追踪用间接操作就可以完成。

（2）追踪访问链 $n_p - n_a$ 次后，到达 a 的声明所在过程的活动记录。根据 6.1 节和 6.2 节的讨论，它的存储单元在相对该活动记录中某个位置的固定偏移处。

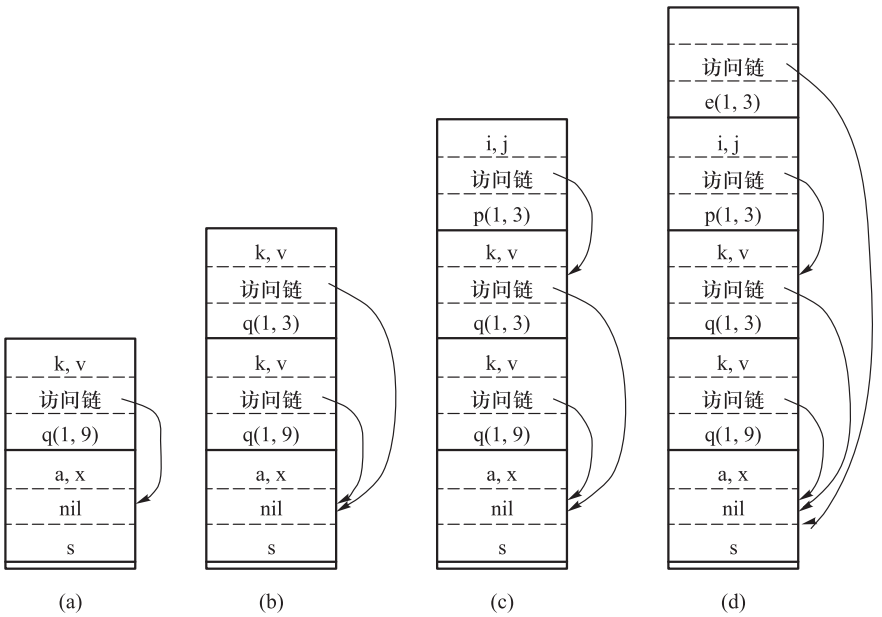


图 6.20　寻找非局部名字存储单元的访问链

因此，过程 p 对变量 a 访问时，a 的地址由下面的二元组表示：

（$n_p - n_a$，a 在活动记录中的偏移）

其中第一个分量给出追踪访问链的次数。

例如，在图 6.19 的第 15 行和第 16 行中，过程 partition 的嵌套深度为 3，它所引用的非局部变量 a 和 v 的嵌套深度分别为 1 和 2。包含这些非局部变量存储单元的活动记录可以从 partition 的活动记录分别追踪访问链 $3-1=2$ 次和 $3-2=1$ 次而找到。访问链是按照过程的静态嵌套关系建立的，因此它也称为**静态链**。

建立访问链的代码是过程调用序列的一部分。假定嵌套深度为 n_p 的过程 p 调用嵌套深度为 n_x 的过程 x，建立被调用过程访问链的代码取决于被调用过程是否嵌在调用过程中：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第243页 文件编号: 422624 页面编号: 32434538446 打印人: 8685 2026-03-09 10:23:02

（1）$n_p < n_x$ 的情况。这表明被调用过程 x 比 p 嵌得更深，而且 x 肯定就声明在 p 中，否则 p 不能访问 x。图 6.20(a) 中 sort 调用 quickSort 和图 6.20(c) 中 quickSort 调用 partition 都属于这种情况。此时，被调用过程的访问链必须指向栈中刚好在它下面的调用过程的活动记录的访问链。

（2）$n_p \geqslant n_x$ 的情况。根据作用域规则，p 和 x 的嵌套深度分别为 1，2，…，$n_x - 1$ 的外围过程肯定相同。图 6.20(b) 中 quickSort 调用本身和图 6.20(d) 中 partition 调用 exchange 都属于这种情况。从调用过程追踪访问链 $n_p - n_x + 1$ 次，即到达了静态包围 x 和 p 的且离它们最近的那个过程的最新活动记录。所到达的这个访问链就是被调用过程 x 的活动记录中的访问链应该指向的那个访问链。同样，$n_p - n_x + 1$ 的值可以在编译时计算。

当过程作为参数传递，尤其是被嵌套过程作为参数传递的情况，应该怎样在该过程被激活时建立它的访问链呢？下面以图 6.21 所示的 Pascal 程序为例说明。在该程序的第 8 行，c 的过程体把 f 作为参数传递给 b。在 b 的过程体中，语句 writeln(h(2)) 激活 f，因为形参 h 代表的是 f，即 writeln 打印调用 f(2) 的结果。可以看出，f 在 b 的过程体中被激活，但是从 b 的访问链难以建立 f 的访问链。

怎样解决这个问题呢？在过程作为参数传递时，必须将其访问链和它一起传递。如图 6.22 所示，当过程 c 传递 f 时，会确定 f 的访问链，并将该链与 f 一起传递给 b。随后，当 f 在 b 中被激活时，该链用来建立 f 活动记录的访问链，而不是按上面的常规方法来建立。

```
(1)  program param(input,output);
(2)      procedure b(function h(n:integer):integer);
(3)          begin writeln(h(2))end{b};
(4)      procedure c;
(5)          var m:integer;
(6)          function f(n:integer):integer;
(7)              begin f:=m+n end{f};
(8)          begin m:=0;b(f)end{c};
(9)      begin
(10)         c
(11)     end.
```

图 6.21 过程作为参数

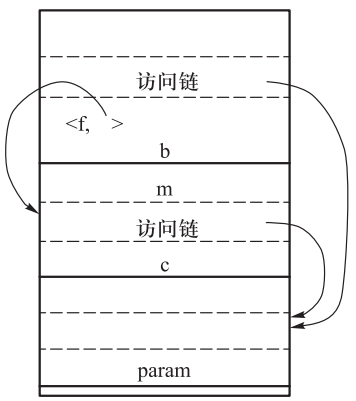


图 6.22 过程 f 带着它的访问链一起传递

对于 Pascal 语言来说，如果允许一个函数的返回值是函数，那么有可能出现作为返回值的函数执行时，需访问的非局部数据已经不存在，习题***6.22** 可以说明这一点。所以 Pascal 语言不允许函数类型作为函数的返回值类型。

前面已经提到，C 语言的函数声明不能嵌套，因此 C 语言的函数不论在什么情况下激活，在执行它的代码时要访问的数据分成两种情况：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第244页 文件编号: 422624 页面编号: 32444538444 打印人: 8685 2026-03-09 10:23:02

（1）非静态局部变量（包括参数），它们分配在运行栈顶的那个活动记录中。

（2）外部变量（包括定义在其他源文件中的外部变量）和静态的局部变量，它们都分配在静态数据区。

因此，C 语言不会出现 Pascal 语言碰到的那种困难。

*6.3.3 动态作用域

在动态作用域下，被调用过程 q 的非局部名字 a 到存储单元的绑定，与 q 的调用过程 p 中对 a 的绑定是一致的，即 q 的非局部名字 a 和 p 中的 a 引用的是同样的存储单元。在 q 被调用时，新的绑定仅为 q 的局部名字建立，这些名字在 q 的活动记录中占用存储单元。也就是说，一个名字的声明在运行时实施它的影响，直至过程调用时在被调用过程中遇到该名字的一个新的声明为止；当被调用过程停止时，该影响恢复。

图 6.23 所示的程序可用来说明动态作用域的概念。第 3 和 4 行的过程 show 输出非局部量 r 的值。按照 Pascal 的静态作用域规则，非局部量 r 在第（2）行声明的作用域中，所以程序的输出是

```
0.250    0.250
0.250    0.250
```

如果是动态作用域，则它的输出是

```
0.250    0.125
0.250    0.125
```

```
(1)  program dynamic(input,output);
(2)      var r:real;
(3)      procedure show;
(4)          begin write(r:5:3)end;
(5)      procedure small;
(6)          var r:real;
(7)          begin r:=0.125;show end;
(8)      begin
(9)          r:=0.25;
(10)         show;small;writeln;
(11)         show;small;writeln
(12)     end.
```

图 6.23 用于说明作用域的一个程序

当主程序在第 10 和 11 行调用 show 时，写出 0.250，因为使用的是局部于主程序的 r。但是在第 7 行从 small 调用 show 时，输出 0.125，因为使用的是局部于 small 的变量。在第 11 行重复第 10 行的调用，目的是想说明第 10 行的 small 调用结束后，第 6 行 r 声明的影响结束，第 2 行 r 声明的影响恢复。因此，当第 11 行再次调用 show 时，会再次输出 0.250。

下面介绍两种实现动态作用域的方法：

（1）深访问。概念上，如果访问链指向的活动记录与控制链指向的活动记录一样，那么实现了动态作用域。因此，一种简单的实现是省略访问链，并用控制链搜索运行栈，寻找包含所需非局部名字的第一个活动记录。深访问的意思是，搜索可能要深入运行栈中，搜索的深度可能取决于程序的输入，编译时决定不了。

（2）浅访问。为程序中每个名字在静态数据区分配空间，用于保存它的当前值。当过程 p 的新活动出现时，p 的局部名字 n 使用在静态数据区分配给 n 的存储单元。n 先前的值保

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第245页 文件编号: 422624 页面编号: 32454538441 打印人: 8685 2026-03-09 10:23:02

存在 p 的活动记录中，当 p 的活动结束时再恢复。

深访问对非局部名字需要较长的访问时间，但是它在活动的开始和结束处没有额外开销；浅访问则相反，它可以直接访问非局部名字，但在活动的开始和结束处需要时间来维护这些值。当函数作为参数传递和作为结果返回时，深访问可以较直截了当地实现。

6.4 参数传递

过程调用时，调用者和被调用者之间交换信息的方法通常是通过非局部名字和通过被调用过程的参数。本节讨论形参和实参联系的几种一般方法，它们是值调用、引用调用和换名调用。了解语言（或编译器）的参数传递方法是很重要的，因为程序的结果依赖于所使用的方法。参数传递方法的区别在于传递实参的右值、左值还是实参本身的正文。

6.4.1 值调用

值调用是最简单的传递参数的方法。调用者计算实参，并把它的值（右值）传递给被调用者。C 语言和 Java 语言使用值调用，值调用也是 C++ 和 Pascal 等很多语言参数传递方式的一种选择。本章到目前为止的所有程序都是按这种方式传递参数的。值调用可以如下实现：

（1）把形参当做所在过程的局部名看待，形参的存储单元在该过程的活动记录中。

（2）调用者计算实参，并把它的值放入形参的存储单元中。

值调用的显著特征是，对形参的任何操作不会影响调用者实参的值。

需要注意的是，在 C 语言中，指向一个变量的指针可以传递给被调用过程，导致该变量在被调用过程中可能被修改。同样地，在 C、C++ 和 Java 语言中，当数组名或对象名作为参数传递时，本质上是将它们的指针或引用传递给被调用过程。例如，如果 a 是调用者的一个数组名，它的值传递给被调用者的形参 x，那么 x[i] = 2 这样的赋值改变数组元素 a[i]。

在 Go 语言中，参数传递是按值传递的，即在函数调用时会将参数的副本传递给函数。对于整数、浮点数等基本类型，以及 string、数组、结构体等直接存储值的类型，按值传递会创建这些类型的副本，因此在函数中修改参数不会影响原始变量。然而，对于切片（slice）、映射（map）、通道（channel）、接口（interface）、函数（func）等类型，虽然仍是按值传递，但实际上只传递了底层数据结构的引用。因此，对这些类型的内容进行修改会影响原始数据。要实现引用传递，Go 语言提供了指针类型（如 *int），可以通过指针参数来传递外部对象的地址。在需要避免数据复制或希望直接修改外部对象的场景下，使用指针传递是一个有效的方法。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第246页 文件编号: 422624 页面编号: 32464538449 打印人: 8685 2026-03-09 10:23:02

6.4.2 引用调用

引用调用（也称为**地址调用**）的参数传递方式是，调用者把实参存储单元的地址（实参的左值）传递给被调用者，使得被调用者可以直接操作实参的存储单元。引用调用可以实现如下：

（1）如果实参是有左值的名字或表达式，则把该左值放入形参的存储单元。如果实参是 $a+b$ 或 2 这样没有左值的表达式，则把它的值计算到新的存储单元，然后传递这个单元的地址。

（2）在被调用过程的目标代码中，任何对形参的访问都是通过传递给该过程的地址来间接访问实参的。

引用调用的显著特征是，对形参的任何赋值都会影响调用者的实参。

在 Pascal 语言中，通过 var 参数实现按引用调用传递，允许函数直接访问调用时传递的变量，从而在函数内对该变量进行更改。类似地，在 C++ 语言中，引用类型参数（如 int& 等）也是通过引用调用的，这意味着传递的是变量的地址，而不是值。这样，引用调用在修改大数据对象时尤其重要，因为避免了值调用所需的完整复制，降低了开销。

对于较大数据对象，如数组或复杂对象，引用调用显著减少了开销。按值传递会将整个对象复制到新空间，而引用调用仅传递一个地址。6.4.1 小节已经提到，Java 和 C++ 等语言采取了一种混合策略：基本类型（如整型和实型）按值传递，其他类型按引用传递。

在 Python 语言中，参数传递采用“按对象引用”的方式，即将对象的引用传递给函数。对于可变对象（如列表、字典等），在函数中对它们的修改会直接影响原始对象，相当于引用调用。而对于不可变对象（如整数、字符串、元组等），虽然引用被传递，但在函数中修改这些对象会生成新对象，因此不会影响原始值，效果类似值调用。如果要避免原始对象被修改，可对可变对象进行复制（如使用 copy 模块中的浅复制 copy() 或深复制 deepcopy() 函数）。整体而言，Python 语言始终按引用传递参数，不涉及数据复制，除非显式地创建副本。

6.4.3 换名调用

换名调用出自一种计算模型——λ 演算，它用于早期的编程语言 ALGOL 60。换名调用可以用 ALGOL 的复制规则来定义，具体如下：

（1）把过程当作宏来对待，也就是在调用点，用被调用过程的体来替换调用者的调用，但是形参用对应的实参正文来代换。这种正文替换方式称为**宏展开**或**内联展开**。

（2）被调用过程的局部名与调用过程的名字保持区别。可以认为在宏展开前，被调用过程的局部名字都系统地被重新命名成可区别的名字。

（3）为保持实参的完整性，实参可以用括号包围。

【例 6.7】 对于过程

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第247页 文件编号: 422624 页面编号: 32474538447 打印人: 8685 2026-03-09 10:23:02


```
procedure swap(var x, y: integer);
var temp: integer;
begin
    temp := x;
    x := y;
    y := temp
end
```

调用 swap(i, a[i]) 在换名调用下的实现好像它是

```
temp := i;
i := a[i];
a[i] := temp
```

从这里可以看出换名调用和其他方式的重要区别。第 2 行引用的 a[i] 和第 3 行被赋值的 a[i] 可能是不同的数据单元，因为 i 的值在第 2 行可能被改变了。

换名调用虽然最初源于理论研究的兴趣，但由于其实现的复杂性，逐渐不再受欢迎。尽管如此，它的内联展开的概念表明，可以有效缩短程序的运行时间。建立过程的活动，包括活动记录的空间分配、机器状态的保存、链的建立和控制的转移等，都需要一定的代价。如果过程体较小，过程调用序列的代码可能会超过过程体本身的代码。此时，把过程体的代码内联展开到调用者的代码中，可以提升运行程序运行效率，尽管可能会使代码稍长。基于此考虑，C/C++ 语言可以使用 inline 关键字定义内联函数，Rust 语言则使用 #[inline] 属性来标识，编译器在合适的情况下会将其展开为过程代码，从而提高执行效率。Java 语言虽然没有直接的 inline 关键字，但其编译器和虚拟机能够进行内联优化。Go 语言没有显式的内联函数机制，但其编译器在编译时也可能自动进行内联优化。

*6.5 堆 管 理

当过程终止时，一般而言，其局部变量不再可访问。然而，许多语言允许创建对象或其他数据，它们的生存期没有被约束在创建它们的过程活动的生存期之内。例如，C++ 和 Java 语言允许程序员用 new 创建对象，这些对象（或者说是指向它们的指针）可以从一个过程传递到另一个过程。因此，创建它们的过程终止后，它们可能继续长期存在。这样的对象存放在堆上，堆用来存放这种生存期不确定或一直到程序显式释放的数据。

本节讨论在堆上分配和回收空间的内存管理器，它是应用程序和操作系统之间的一个接口。内存管理在不同编程语言中具有各自的实现机制。C 语言使用 malloc 和 free 进行动态内存分配和释放，常见的实现包括 glibc 的 ptmalloc 和 Doug Lea 的 dlmalloc。此外，高效的并发内存分配器如 jemalloc、TBBmalloc 和 TCMalloc 也得到了广泛应用。C++ 语言同样使用 new 和 delete 创建对象，并在其并发内存模型中有专门的研究。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第248页 文件编号: 422624 页面编号: 32484538445 打印人: 8685 2026-03-09 10:23:02

Java 语言使用 new 创建对象，并依赖垃圾回收（garbage collection）机制来自动管理内存，因此垃圾回收器是 Java 运行系统中完成内存管理的重要子系统，它将在 11.3 节介绍。Go 语言也使用 new 和垃圾回收机制，具体的内存模型可参考 Go 的官方文档。Rust 语言引入了所有权（ownership）的概念，使用 new 分配内存，并通过 drop 或 dealloc 进行释放。综上所述，每种语言的内存管理策略旨在提高效率并减少内存泄漏的风险，体现了其对程序性能和安全性的关注。

6.5.1 内存管理器

内存管理器所掌握的基本信息是堆中的空闲空间，它执行的基本函数有以下两个：

（1）分配函数。当程序请求空间以存放数据时，内存管理器按所需大小分配一块连续的堆块。只要有可能，它总是用堆中的空闲空间来满足分配请求；如果堆中没有满足大小要求的空闲块，它就从操作系统申请虚拟内存的连续字节以增加堆空间。如果空间被耗尽，内存管理器将此信息反馈给用户程序。

（2）回收函数。内存管理器将回收的空间返回到空闲空间池，以便在后续的内存分配请求中重新利用。内存管理器通常选择保持回收的内存空间在内部使用，而不是立即返还给操作系统，以实现更高效的内存管理。

在下面两种情况下，内存管理器非常简单：

（1）所有分配请求需要同样大小的存储块。Lisp 语言属于这种情况。纯 Lisp 仅使用一种数据元素，它是含两个指针的存储单元，所有数据结构都是在这样的数据元素上构建的。

（2）存储的回收是可以预见的，如先分配的先回收。在某些场合下这种情况会出现，最常见的就是可以分配在运行栈上的数据。

然而，对于大多数编程语言来说，上面两种情况都不满足。也就是说，需要分配不同大小的数据对象，而且没有好的方法可以预测它们的生存期。因此，内存管理器必须应对任何次序、任何大小的分配和释放请求，小到 1 字节，大到整个程序地址空间。

内存管理器被期望具有下列性质：

（1）空间有效性。内存管理器应该极小化程序需要的堆空间总量。这样做允许较大的程序运行在固定虚拟地址空间上。空间有效性可以通过极小化碎片来获得。

（2）程序有效性。内存管理器应该很好地利用内存子系统，使得程序能运行得快一些。在 6.5.2 小节将看到，一条指令的执行时间将在相当大的范围内变化，它取决于所涉及对象在内存的什么地方。幸好，程序通常展示出较好的局部性，即程序访问内存是以一种非随机的聚集方式（6.5.3 小节讨论这种现象）。通过关注数据对象在内存中的布局，内存管理器可以较好地利用空间，从而使程序运行得较快。

（3）低开销。因为内存分配和释放操作在许多程序中频繁出现，所以这些操作要尽可能有效，使得分配和回收操作所花时间在整个程序执行时间中的比例尽量小。注意，分配的代

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第249页 文件编号: 422624 页面编号: 32494538443 打印人: 8685 2026-03-09 10:23:02

价被小数据对象的分配请求所支配，而管理较大数据对象的开销不是很重要，因为它通常可以从这个数据对象上的大量计算中得到分摊。

由于数据结构的教材上通常都介绍堆空间的分配和回收算法，包括怎样减少碎片等，因此本书略去这些算法的介绍。

6.5.2 计算机内存分层

了解内存的行为特点是优化内存管理设计的前提。虽然现代计算机都设计成程序员无须关心内存子系统的细节就能写出正确的程序，但是程序的效率不仅取决于被执行的指令数，还受到每条指令执行时间的影响。由于访问不同部分内存的耗时可能从纳秒到微秒不等，因此不同指令的执行时间会有显著差异。数据密集型程序若能有效利用内存子系统，会显著提升性能。在 6.5.3 小节中将讨论一种称为“局部性”的现象，这是一种典型程序展现出的非随机行为。

内存访问时间差别大源于硬件技术的基本局限。可以构造小容量高速存储器，也可以构造大容量低速存储器，但是构造不了大容量的高速存储器。例如，现在不可能构造和高性能处理器的运行一样快的纳秒访问时间的千兆字节存储器。因此，现代计算机都把存储器进行分层安排，图 6.24 是一种内存分层（memory hierarchy），它由一系列存储元件组成，其中最小最快的最“靠近”处理器，较大较慢的则离处理器较远一些。

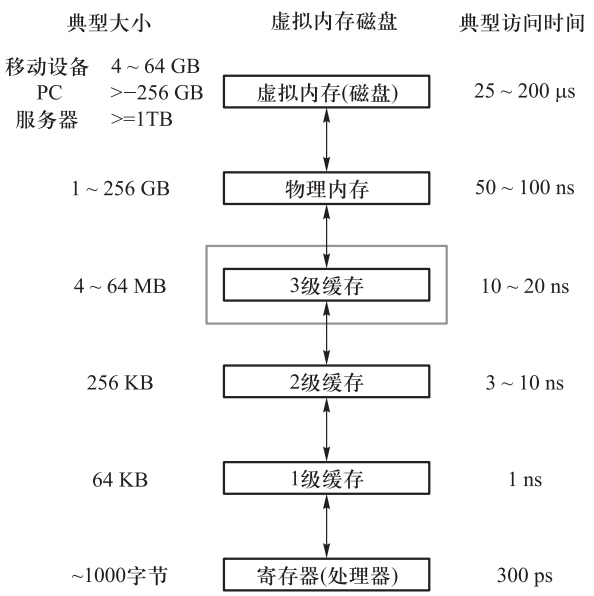


图 6.24 典型的内存分层配置

典型情况是，一个处理器有少数寄存器，它们的内容由软件控制。往上一层是一级或多级缓存，通常由静态 RAM 构成，大小在几十千字节（KB）到几十兆字节（MB）。再往上一

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第250页 文件编号: 422624 页面编号: 32504538441 打印人: 8685 2026-03-09 10:23:02

层是物理内存（主存），由一百到几百吉字节（GB）的动态 RAM 构成。物理内存由虚拟内存支持，虚拟内存由几吉字节（GB）到几十太字节（TB）的磁盘实现。

对于一个内存访问，机器从最底层开始，逐层查找，直到定位数据为止。

寄存器数量很少，为了充分发挥和利用它们，通常由编译器生成的代码来管理，而内存分层中其他层次则被自动管理。这不仅简化了编程，而且同一程序在不同内存配置的机器上能同样有效地运行。缓存由硬件管理，以跟上相对快的 RAM 访问时间。磁盘相对较慢，因此虚拟内存会借助一种称为翻译后备缓冲器（translation lookaside buffer）的硬件结构，由操作系统来管理。

数据以块为单位在相邻层次之间进行传送。为分摊访问代价，较大的块用在内存分层的较慢层次之间。在主存和缓存之间，数据传送的块称为缓存行（cache line），长度为 32~256B。在虚拟内存和主存之间，数据传送的块称为页，大小为 4~64KB。

6.5.3 程序局部性

大多数程序展现出显著的局部性特征，即它们通常集中在一小部分代码段运行，并且主要操作一小部分数据。如果程序在短时间内多次访问同一内存单元，则称该程序具有**时间局部性**。如果程序在短时间内访问的内存单元彼此相邻，则称该程序具有**空间局部性**。

传统的说法是，程序中 90%的时间消耗在执行 10%的代码上，其原因如下：

（1）程序中经常含有永远不会被执行的指令。由组件和库构建的程序通常仅使用所提供功能的一小部分。此外，随着需求变化和程序演化，遗留系统（legacy system）中常包含许多不再使用的指令。

（2）在程序的典型执行中，通常只有一小部分代码可能被真正执行。例如，处理非法输入和例外情况的指令虽然对程序的正确性来说至关重要，但它们很少被执行。

（3）典型程序的大部分时间消耗在程序中最内层循环和深度递归的执行上。

现代计算机的内存分层设计充分利用了程序的局部性特性。通过将频繁使用的指令和数据存放在速度快但空间小的内存层（如缓存）中，而把其余的放在大而慢的内存层（如主存和磁盘）中，可以显著降低程序的平均内存访问时间，从而提升系统的整体性能。

一般来说，从代码本身很难看出哪部分代码会被频繁使用，尤其是对某个特定的输入。即使知道了哪些指令会被频繁执行，最快的缓存也可能没有大到足以把它们同时放在其中。于是，必须动态调整最快缓存的内容，让它保存在不久的将来会频繁使用的指令。

怎样最优化利用内存分层？把最近使用的指令保存在缓存往往是一种较好的策略，换句话说，过去使用情况一般是内存未来使用的一个有益预测。当一条新的指令被执行，很可能下一条指令也将被执行，这个现象是空间局部性的一个例子。改进指令空间局部性的一种有效技术是，编译器尽可能把可能毗邻执行的基本块（基本块是一个顺序执行的指令序列，见第 8 章）放在同一页中，甚至放在同一缓存行上。属于同一个循环或同一个函数的代码也很

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第251页 文件编号: 422624 页面编号: 32514538449 打印人: 8685 2026-03-09 10:23:02

可能一起执行。

改变数据布局或计算次序也可以改进程序数据访问的时间和空间局部性。例如，对于需重复访问大批数据的计算来说，每次遍历数据完成一小部分计算的方式则效率不高。较好的做法是每次从较慢层取部分数据到较快层（如从磁盘到主存），在这些数据驻留较快层期间完成对它们的所有计算。同样的思想可用在主存、缓存和寄存器中的数据。

6.5.4 手工回收请求

C 和 C++ 语言要求程序员显式管理堆内存的分配与释放，这种方式称为**手工回收**。这意味着程序员需要直接介入内存管理。理想情况下，任何不再访问的内存都应该被释放，同时任何有可能被访问的内存都需要保留。然而，在实际编程中，要严格满足这两个条件是非常困难的。

手工内存管理很容易出错。常见的错误有两种形式：一种是没有释放已经引用不到的堆块，称为**内存泄漏**错误；另一种是引用已经被释放的堆块，称为悬空引用错误。后者也可能发生在栈上，6.2.7 小节已经介绍过。注意，虽然内存泄漏会使内存使用不断增加，进而导致程序运行速度降低，但是只要内存没有用尽，它就不会影响程序的正确性。许多程序容忍内存泄漏，尤其是当泄漏速度很慢时。然而，对于长时间运行的程序，特别是不间断运行的操作系统或服务器代码来说，不出现内存泄漏是至关重要的。

自动无用单元收集通过回收所有无用单元来摆脱内存泄漏。但即使是这样，程序仍然可能占用了一些不再使用的内存。例如，一个对象在程序中不再使用，但目前还存在指向它的引用，这导致该对象无须保留却又不能自动回收。在这种情况下，程序员应该移开对该对象的引用，以便它能被自动回收。

过度积极地释放数据对象可能会导致比内存泄漏更严重的悬空引用问题。悬空引用容易引发难以检测的错误。由于很难判断某个数据对象是否已经不再被使用，程序员通常会选择不主动释放这个对象，以规避悬空引用可能引发的不稳定性和程序崩溃的风险。

一个相关的编程错误是访问非法地址。常见的例子有对空指针（null pointer）进行解引用（dereferencing）和越界访问数组元素。这样的安全违规行为需要尽早被发现，它们可能导致黑客控制程序和机器。为防止此类错误，编译器可以插入检查代码，确保每次访问都在有效范围内。编译器的优化器可以发现其中没有必要的检查并把它们删除，因为优化器可以推导出一些访问一定落在有效范围内。

习 题

6.1 使用 Pascal 语言的作用域规则，确定下面程序中用于名字 a 和 b 的每个出现的声明。该程序的输出是整数 1，2，3，4。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第252页 文件编号: 422624 页面编号: 32524538447 打印人: 8685 2026-03-09 10:23:02


```
program a(input, output);
    procedure b(u, v, x, y: integer);
        var a: record  a, b: integer end;
            b: record  b, a: integer end;
        begin
            with a do begin a := u; b := v end;
            with b do begin a := x; b := y end;
            writeln(a.a, a.b, b.a, b.b)
        end;
    begin
        b(1, 2, 3, 4)
    end.
```

6.2 一个 C 程序的 3 个文件 head.h、file1.c 和 file2.c 的内容分别如下：

```
head.h:                 file1.c:                    file2.c:
short int a = 10;       #include "head.h"          #include "head.h"
                        main() {
                        }
```

在 x86/Linux 系统上，使用某版本的 GCC 编译器，编译命令如下：

```
cc file1.c file2.c
```

编译结果报错的主要信息如下：

```
multiple definition of 'a'
```

试分析为什么会报这样的错误。

***6.3** 考虑下面的 C 语言程序：

```
#include <string.h>
#include <stdio.h>
void main() {
    char *cp1, *cp2; cp1 = "12345";
    cp2 = "abcdefghij";
    strcpy(cp1,cp2);
    printf("cp1 = %s\ncp2 = %s\n", cp1, cp2);
}
```

(a) 该程序经早先某些 C 编译器的编译，其目标程序的运行结果如下：

```
cp1 = abcdefghij
cp2 = ghij
```

试分析：为什么 cp2 所指向的串被修改了？

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第253页 文件编号: 422624 页面编号: 32534538445 打印人: 8685 2026-03-09 10:23:02

(b) 在 x86/Linux 系统上经任何近期版本的 GCC 编译器编译后，该程序运行时，操作系统报告段错误（segmentation fault）并终止运行，请分析原因。

6.4 一个 C 语言程序如下：

```
1  typedef struct _a {
2      char c1;
3      long i;
4      char c2;
5      double f;
6  }a;
7  typedef struct _b {
8      char c1;
9      char c2;
10     long i;
11     double f;
12 }b;
13 void main() {
14     printf("Size of double, long, char = %d, %d, %d\n",
15         sizeof(double), sizeof(long), sizeof(char));
16     printf("Size of a, b = %d,%d\n", sizeof(a), sizeof(b));
17 }
```

该程序在早先的 SPARC/Solaris 系统上的运行结果如下：

```
Size of double, long, char = 8, 4, 1
Size of a, b = 24,16
```

在 x86-32/Linux 系统上经编译器 GCC 11.4.0 编译后[①]，运行时第 2 行输出不一样，是

```
Size of a, b = 20,16
```

在 x86-64/Linux 系统上经编译器 GCC 11.4.0 编译后，运行结果如下：

```
Size of double, long, char = 8, 8, 1
Size of a, b = 32,24
```

结构体类型 a 和 b 的域都一样，仅次序不同，为什么它们需要的存储空间不一样？为什么在不同的机器上情况又不一样？

***6.5** 一个 C 语言程序如下：

```
1  typedef struct _a {
2      short i;
3      short j;
```

① 在 64 位系统用 gcc -m32 进行编译，需事先通过 sudo apt-get install gcc-multilib 安装 32 位开发库。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第254页 文件编号: 422624 页面编号: 32544538443 打印人: 8685 2026-03-09 10:23:02


```
    short k;
} a;
typedef struct _b {
    long i;
    short k;
} b;
void main() {
    printf("Size of short, long, a and b = %d,%d,%d,%d\n",
    sizeof(short), sizeof(long), sizeof(a), sizeof(b));
}
```

该程序在 x86-64/Linux 系统上，经编译器 GCC 11.4.0 编译后，运行结果如下：

```
Size of short, long, a and b = 2, 8, 6, 16
```

已知 short 和 long 类型分别按 2 和 8 的倍数对齐。试问：为什么类型 b 的 size 会等于 16？

6.6 下面是 C 语言两个函数 f() 和 g() 的概略（它们不再有其他的局部变量）：

```
int f (int x) { int i; …return i +1; … }
int g (int y) {int j; … f (j +1); … }
```

请按照图 6.12 所示的形式，画出函数 g() 调用 f()，f() 的函数体正在执行时，活动记录栈的内容及相关信息，并按图 6.10 所示左侧箭头方式画出控制链。假定函数返回值是通过寄存器传递的。

***6.7** 下面给出 C 语言程序 static.c 及其在 x86-32/Linux 系统上的编译结果（编译器版本见汇编代码最后一行）。根据所生成的汇编程序来解释程序中 4 个变量在存储分配、作用域、生存期和置初值方式等方面的区别。

```
static long aa = 10;
short bb = 20;
void func() {
    static long cc = 30;
    short dd = 40;
}
```

该 C 语言程序生成的汇编代码如下：

```
    .file "static.c"
    .version "01.01"
gcc2_compiled.:
.data
    .align 4
    .type aa,@object
    .size aa,4
aa:
    .long 10
.globl bb
    .align 2
    .type bb,@object
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第255页 文件编号: 422624 页面编号: 32554538440 打印人: 8685 2026-03-09 10:23:02


```
    .size bb,2
bb:
    .value 20
    .align 4
    .type cc.2,@object
    .size cc.2,4
cc.2:
    .long 30
.text
    .align 4
.globl func
    .type func,@function
func:
    pushl %ebp
    movl %esp,%ebp
    subl $4,%esp
    movw $40,-2(%ebp)
.L1:
    leave
    ret
.Lfe1:
    .size func,.Lfe1-func
    .ident "GCC: (GNU) egcs
        -2.91.66 19990314/Linux (
        egcs-1.1.2 release)"
```

***6.8** 下面给出 C 语言程序 ex6-8.c 及其在 x86-64/Linux 系统上的编译结果。根据所生成的汇编程序来解释程序中 5 个变量在存储分配、作用域、生存期和置初值方式等方面的区别。

```
static long aa = 10;
int cc;
short bb;
void func() {
    static long cc;
    short dd = 40;
}
```

该 C 语言程序生成的汇编代码如下：

```
  .file "ex6-8.c"
  .text
  .data
  .align 8
  .type aa, @object
  .size aa, 8
aa:
  .quad 10
  .globl  cc
  .bss
  .align 4
  .type cc, @object
  .size cc, 4
cc:
  .zero 4
  .globl  bb
  .align 2
  .type bb, @object
  .size bb, 2
bb:
  .zero 2
  .text
  .globl  func
  .type func, @function
func:
.LFB0:
  endbr64  ; 确保间接分支跳转的合法性
  pushq %rbp
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第256页 文件编号: 422624 页面编号: 32564538448 打印人: 8685 2026-03-09 10:23:02


```
29    movq  %rsp, %rbp
30    movw  $40, -2(%rbp)
31    nop
32    popq  %rbp
33    ret
34  .LFE0:
35    .size func, .-func
36    .local  cc.0
37    .comm cc.0,8,8
38    .ident  "GCC: (Ubuntu 11.4.0-1
            ubuntu1~22.04) 11.4.0"
```

6.9 假定使用值调用、引用调用、换名调用，下面的程序输出结果是什么？

```
1  program main(input, output);
2  var a, b: integer;
3  procedure p(x, y, z: integer);
4      begin
5          y := y+1;
6          z := z+x;
7      end;
8  begin
9      a := 2;
10     b := 3;
11     p(a+b, a, a);
12     print a;
13 end.
```

6.10 C 语言函数 f 的定义如下：

```
int f (int x, int *py, int **ppz) {
    **ppz +=1; *py +=2; x +=3; return x + *py +
        **ppz;
}
```

变量 a 是指向 b 的指针，变量 b 是指向 c 的指针，c 是整型变量且当前值是 4。那么执行 f (c, b, a) 的返回值是多少？

***6.11** 下面的 C 语言程序中，函数 printf() 的调用仅含格式控制字符串 1 个参数，该程序在 x86/Linux 系统上，经某编译器编译后，运行时输出 3 个整数。试从运行环境和 printf() 的实现来分析，为什么此程序会有 3 个整数输出。

```
#include <stdio.h>
void main() {
    printf("%d, %d, %d\n");
}
```

6.12 下面是一个 C 语言程序：

```
long f1(i) long i; {
    return(i*10);
}
long f2(long i) {
    return(i*10);
}
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第257页 文件编号: 422624 页面编号: 32574538446 打印人: 8685 2026-03-09 10:23:02


```
7  void main() {
8      printf("f1 = %ld, f2 = %ld\n", f1(10.0), f2(10.0) );
9  }
```

其中，函数 f1 和 f2 仅形式参数的描述方式不一样。该程序在 x86-64/Linux 系统上，用编译器 GCC 11.4.0 编译后，运行结果如下：

```
f1 = 100, f2 = 100
```

但若用 gcc -m32 编译，运行结果为

```
f1 = 0, f2 = 100
```

请解释：为什么用同样的实参调用这两个函数，在 32 位和 64 位系统下表现出不一样的运行结果。

6.13 一个 C 语言程序如下：

```
1  void func(i1,i2,i3) long i1,i2,i3; {
2      long j1,j2,j3;
3      printf("Addresses of i1,i2,i3 = %o,%o,%o\n",&i1,&i2,&i3);
4      printf("Addresses of j1,j2,j3 = %o,%o,%o\n",&j1,&j2,&j3);
5  }
6  void main() {
7      long i1,i2,i3;
8      func(i1,i2,i3);
9  }
```

该程序在 x86/Linux 系统上，经某编译器编译后的运行结果如下：

```
Addresses of i1,i2,i3 = 27777775460,27777775464,27777775470
Addresses of j1,j2,j3 = 27777775444,27777775440,27777775434
```

从上面的结果可以看出，func 函数的 3 个形式参数的地址依次升高，而 3 个局部变量的地址依次降低。试说明为什么会有这个区别。注意，输出的数据是八进制的。

6.14 一个 C 语言程序如下：

```
1  int fact(i) int i; {
2      if(i==0)
3          return 1;
4      else
5          return i*fact(i-1);
6  }
7  void main() {
8      printf("%d\n", fact(5));
9      printf("%d\n", fact(5,10,15));
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第258页 文件编号: 422624 页面编号: 32584538444 打印人: 8685 2026-03-09 10:23:03


```
10      printf("%d\n", fact(5.0));
11      printf("%d\n", fact());
12  }
```

该程序在 x86/Linux 系统上，用 GCC 编译器按 gcc-m32 编译后，运行结果如下：

```
120
120
1
Segmentation fault (core dumped)
```

请解释下面的问题：

(a) 第二个 fact 调用：结果为什么没有受参数过多的影响？

(b) 第三个 fact 调用：为什么用浮点数 5.0 作为参数时结果变成 1？

(c) 第四个 fact 调用：为什么没有提供参数时会出现“Segmentation fault”？

(d) 若按 64 位系统编译，运行结果为

```
120
120
Segmentation fault (core dumped)
```

为什么第三个 fact 调用会出现“Segmentation fault”？

6.15 一个 C 语言程序文件 array.c 仅有下面两行代码：

```
1  char a[ ][4] = { "123", "456"};
2  char *p[ ]  = { "123", "456"};
```

在 x86-64/Linux 系统上编译生成的汇编代码列在下面（编译器版本见汇编代码最后一行），从中可以看出，对数组 a 和指针 p 的存储分配是不同的。试依据这里的存储分配，为置了初值后的数组 a 和指针 p 写出类型表达式。

```
        .file "array.c"
        .globl a
        .data
        .align 8
        .type a, @object
        .size a, 8
a:
        .string "123"
        .string "456"
        .globl  p
        .section .rodata
.LC0:
        .string "123"
.LC1:
        .string "456"
        .align 16
        .type p, @object
        .size p, 16
p:
        .quad .LC0
        .quad .LC1
        .ident "GCC: (Ubuntu 11.4.0-1
            ubuntu1~22.04) 11.4.0"
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第259页 文件编号: 422624 页面编号: 32594538442 打印人: 8685 2026-03-09 10:23:03

***6.16** 下面给出一个 C 语言程序及其在早先的 SPARC/SunOS 系统，以及现代的 32 位 x86/Linux 系统上经某编译器编译后的运行结果。

```
1  void func(i,j,f,e) short i,j; float f,e; {
2      short i1,j1; float f1,e1;
3      printf("Address of i,j,f,e = %o,%o,%o,%o\n",&i,&j,&f,&e);
4      printf("Address of i1,j1,f1,e1 = %o,%o,%o,%o\n",&i1,&j1,&f1,&e1);
5      printf("Sizes of short, int, long, float, double = %d,%d,%d,%d,%d\n",
6       sizeof(short),sizeof(int),sizeof(long),sizeof(float),sizeof(double));
7  }
8  void main() {
9      short i,j; float f,e;
10     func(i,j,f,e);
11 }
```

(1) 在 SPARC/SunOS 系统上的运行结果如下：

```
Address of i,j,f,e = 35777772536,35777772542,35777772544,35777772554
Address of i1,j1,f1,e1 =
    35777772426,35777772424,35777772420,25777772414
Sizes of short, int, long, float, double = 2,4,4,4,8
```

从运行结果看，函数 func() 中 4 个局部变量 i1,j1,f1,e1 的地址间隔和它们类型的大小是一致的，而 4 个形式参数 i,j,f,e 的地址间隔和它们类型的大小不一致，试分析不一致的原因。注意，输出的数据是八进制的。

(2) 在 32 位 x86/Linux 系统上的运行结果如下：

```
Address of i,j,f,e = 37777542274,37777542270,37777542260,37777542250
Address of i1,j1,f1,e1 =
    37777542300,37777542302,37777542304,37777542310

Sizes of short, int, long, float, double = 2,4,4,4,8
```

试分析输出这种结果的原因。

***6.17** 一个 C 语言的函数

```
1  void func(c,l) char c;long l; {
2      func(c,l);
3  }
```

在 x86/Linux 系统上编译生成的汇编代码如下（编译器版本见汇编代码最后一行）：

```
1      .file "parameter.c"
2      .version "01.01"
3  gcc2_compiled.:
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第260页 文件编号: 422624 页面编号: 32604538440 打印人: 8685 2026-03-09 10:23:03


```
.text
    .align 4
.globl func
    .type func,@function
func:
    pushl %ebp          // 将旧的基地址指针压栈
    movl %esp,%ebp      // 将当前栈顶指针作为基地址指针
    subl $4,%esp        // 分配空间
    movl 8(%ebp),%eax
    movb %al,-1(%ebp)
    movl 12(%ebp),%eax
    pushl %eax
    movsbl -1(%ebp),%eax
    pushl %eax
    call func
    addl $8,%esp
.L1:
    leave               // 和下一条指令一起完成恢复旧的基地址指针，
    ret                 // 将栈顶指针恢复到调用前参数压栈后的位置，并返回调用者
.Lfe1:
    .size func,.Lfe1-func
    .ident "GCC: (GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

请说明字符型参数和长整型参数在参数传递和存储分配方面的区别（小于长整型 size 的整型参数的处理方式和字符型参数的处理方式是一样的）。

6.18 C 程序设计的教材上说，可以用两种形式表示字符串：一是用字符数组存放一个字符串，二是用字符指针指向一个字符串。教材上同时介绍了这两种形式的共同点和不同点，但有一种可能的区别没有介绍。下面是一个包含这两种形式的 C 语言程序：

```
char c1[]="good!";
char *c2="good!";
void main() {
    c1[0]= 'G';    printf("c1=%s\n", c1);
    c2[0]= 'G';    printf("c2=%s\n", c2);
}
```

该程序在 x86-64/Linux 上经过 GCC 编译器编译后，运行时的信息如下：

```
c1=Good!
Segmentation fault (core dumped)
```

请问：出现“Segmentation fault”的原因是什么？

***6.19** 下面是一个 C 语言程序：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第261页 文件编号: 422624 页面编号: 32614538448 打印人: 8685 2026-03-09 10:23:03


```
1  void main() {
2      long i;    long a[0][4];    long j;
3      i = 4; j = 8;  printf("%ld, %ld\n", sizeof(a), a[0][0]);
4  }
```

虽然出现 long a[0][4] 这样的声明，但在 x86/Linux 系统上，用编译器 GCC 11.4.0 (Ubuntu 11.4.0-1ubuntu1 22.04) 编译时，该程序能够通过编译并生成目标代码。请回答下面两个问题：

(a) sizeof(a) 的值是多少，请说明理由。

(b) a[0][0] 的值是多少，请说明理由。

6.20 从例 6.4 可以看到，C 语言程序执行时只用到了控制链，不需要使用访问链。为什么 Pascal 语言程序执行时需要使用访问链，而 C 语言程序不需要。

6.21 下面是求阶乘的 Pascal 语言程序。画出程序第 3 次进入函数 factor() 时的活动记录栈和静态链。

```
1  program fact(input, output);
2    var f, n : integer;
3    function factor(n : integer) : integer;
4      begin
5        if n=0 then factor :=1
6        else factor :=n*factor(n-1)
7      end;
8    begin n :=5; f :=factor(n); write(f)
9    end.
```

***6.22** 在下面假想的程序中，第 11 行语句 f := a 调用函数 a，a 传递函数 addm 作为返回值。

```
1  program ret (input, output);
2      var f: function (integer): integer;
3      function a: function (integer): integer;
4          var m: integer;
5          function addm (n: integer): integer;
6      begin return m+n end;
7          begin m:= 0; return addm end;
8      procedure b (g: function (integer): integer);
9    begin writeln ( g(2)) end;
10     begin
11         f := a; b(f)
12     end.
```

(a) 画出该程序执行的活动树。

(b) 假定非局部名字使用静态作用域，为什么该程序在栈式分配情况下不能正确工作？

(c) 在堆分配策略下，该程序的输出是什么？

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第262页 文件编号: 422624 页面编号: 32624538446 打印人: 8685 2026-03-09 10:23:03

***6.23** 为什么 C 语言允许函数类型（的指针）作为函数的返回值类型，而 Pascal 语言却不允许？

6.24 一个 C 语言程序如下：

```
#include <stdio.h>
int n;
int f(g) int g(); {
    int m;
    m = n;
    if(m == 0) return 1;
    else {
        n = n-1; return m * g(g);
    }
}
void main() {
    n = 5;
    printf("%d factorial is %d\n", n, f(f) );
}
```

该程序在 Ubuntu 12.04.2 LTS (GNU/Linux 3.2.0-42-generic x86_64) 系统上，经过编译器 GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3 编译后，运行结果不是所期望的

```
5 factorial is 120
```

而是

```
0 factorial is 120
```

试说明原因。

***6.25** 下面的程序在早先的 SPARC/SunOS 系统上运行时陷入死循环，试说明原因。如果将第 5 行的 long *p 改成 short *p，并且将第 15 行 long k 改成 short k 后，loop 中的循环体执行一次便停止了。试说明原因。

```
1  void main() {
2      addr();
3      loop();
4  }
5  long *p;
6  void loop() {
7      long i,j;
8      j=0;
9      for(i=0;i<10;i++){
10         (*p)--;
11         j++;
12     }
13 }
14 void addr() {
15     long k;
16     k=0;
17     p=&k;
18 }
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第263页 文件编号: 422624 页面编号: 32634538444 打印人: 8685 2026-03-09 10:23:03

***6.26** 一个 C 语言程序如下：

```
#include <stdio.h>
void main() {
    func();
    printf("Return from func\n");
}
void func() {
    char s[8];
    strcpy(s,"12345678");
    printf("%s\n",s);
}
```

该程序经过编译器 GCC: (Ubuntu 11.4.0-1ubuntu1 22.04) 11.4.0 编译后，运行结果如下：

```
12345678
Return from func
```

若将语句 strcpy(s,''12345678'') 改成 strcpy(s,''123456789'')，目标程序 a.out 的运行结果及运行失败的报告如下：

```
123456789
*** stack smashing detected ***: a.out terminated
Aborted (core dumped)
```

试分析：为什么会出现这样的运行情况？

***6.27** 通常，函数调用的返回值是简单类型时，用寄存器传递函数值。当返回值是结构体类型时需要采用其他方式。下面是一个 C 语言文件和它在 x86/Linux 系统上编译（编译器版本见汇编代码最后一行）生成的汇编代码。(备注，该汇编码略经修改，以便于阅读。该修改没有影响结果)

(a) 请分析这些代码，总结出函数返回值是结构体类型时，返回值的传递方式。

(b) 若 m() 函数的语句 s = f(10) 改成 s.i = f(10).i + f(20).i + f(30).i，你认为 m() 函数的局部存储分配应该怎样修改，以适用该语句的计算。

源文件 return.c 的内容如下：

```
typedef struct{long i;}S;
S f(k) long k; {
    S s;
    s.i =k;
    return s;
}
void m() {
    S s;
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第264页 文件编号: 422624 页面编号: 32644538442 打印人: 8685 2026-03-09 10:23:03


```
9       s.i = 20;
10      s = f(10);
11  }
```

汇编文件 return.s 的内容如下：

```
1       .file "return.c"
2     .text
3   .globl f
4     .type f, @function
5   f:
6     pushl %ebp
7     movl %esp, %ebp
8     subl $4, %esp
9     movl 8(%ebp), %eax
10    movl 12(%ebp), %edx
11    movl %edx, -4(%ebp)
12    movl -4(%ebp), %edx
13    movl %edx, (%eax)
14    leave
15    ret
16    .size f, .-f
17  .globl m
18    .type m, @function
19  m:
20    pushl %ebp
21    movl %esp, %ebp
22    subl $4, %esp
23    movl $20, -4(%ebp)
24    leal -4(%ebp), %eax
25    pushl $10
26    pushl %eax
27    call f
28    addl $8, %esp
29    leave
30    ret
31    .size m, .-m
32    .section .note.GNU-stack,"",
33          @progbits
34    .ident "GCC: (GNU) 3.3.5 (
35          Debian 1:3.3.5-13)"
```

***6.28** 一个 C 语言的文件如下：

```
1   void func(long i, long j, long k) {
2       k = (i + j) - (i - j - f(k));
3   }
```

经 x86/Linux 系统上编译（编译器版本见汇编代码最后一样）得到的汇编代码分两列在下面给出：

```
1       .file "call.c"
2     .text
3   .globl func
4     .type func, @function
5   func:
6     pushl %ebp
7     movl $esp, %ebp
8     pushl %esi
9     pushl %ebx
10    subl $4, %esp
11    movl 12(%ebp), %eax
12    movl 8(%ebp), %esi
13    addl %eax, %esi
14    movl 12(%ebp), %edx
15    movl 8(%ebp), %eax
16    movl %eax, %ebx
17    subl %edx, %ebx
18      movl 16(%ebp), %eax
19      movl %eax, (%esp)
20      call f
21      subl %eax, %ebx
22      movl %ebx, %eax
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第265页 文件编号: 422624 页面编号: 32654538449 打印人: 8685 2026-03-09 10:23:03


```
23      subl %eax, %esi
24      movl %esi, %eax
25      movl %eax, 16(%ebp)
26      addl $4, %esp
27      popl %ebx
28      popl %esi
29      popl %ebp
30      ret
31      .size func, .-func
32      .section .note.GNU-stack,"",
            @progbits
33      .ident "GCC: (GNU)  3.4.6"
```

请根据上述汇编代码进行总结：为适应函数调用引起的跨函数执行，该编译器在寄存器的值的保护方面有哪些约定？

6.29 下面是一个以函数作为参数的 C 语言程序：

```
1  int f(int g()) {return g(g);}
2  void main(){f(f);}
```

（a）对于函数类型的形式参数，调用时的参数传递传什么？

（b）该程序执行时，系统报告“Segmentation fault”，分析其原因。

6.30 有人认为，下面的 C 语言程序中结构体类型 record 的定义方式可用来动态生成其中 a 数组的大小不一样的结构体，以适应某些编程场合的需要。你认为这样的程序能够通过 C 编译器的类型检查吗？请说明理由。

```
1  #include <malloc.h>
2  typedef struct {double r; int n; float a[];}
3          record;
4  void main() {
5      record * p;
6      p = malloc(sizeof(record) + sizeof(float) *
7          5);
8      p->n = 5; p->a[4] = 100.0; ...
9  }
```

6.31 （a）Java 语言的编译器通常把数组分配在堆上。Java 语言中的数组一般不能静态确定大小应该不是将它分配在堆上的原因，因为图 6.17 给出了一种将不能静态确定大小的数组动态地分配在活动记录栈上的方法。Java 语言中的数组一般不能分配在活动记录栈上的原因是什么？

（b）将数组分配在活动记录栈上和分配在堆上，对程序运行有什么区别？

***6.32** Java 语言的实现通常把对象和数组都分配在堆上，把指向它们的指针分配在栈上，依靠运行时的垃圾收集器来回收堆上那些从栈不可达的空间（垃圾）。这种方式提高了语言的安全性，但是增加了运行开销。编译时能否采用一些技术，以降低垃圾收集所占用的运行开销？简述你的方案。

第 7 章

中间语言与中间代码生成

第 1 章已经介绍，编译器的前端把源程序翻译成中间表示（intermediate representation，IR），后端从中间表示产生目标代码，与目标语言有关的细节尽可能限制在后端。使用独立于机器的中间表示，主要有以下两点优势：

（1）再目标（retargeting，指生成另一种机器的目标代码）比较容易。把针对新机器的后端与现成的前端组合起来，就可以得到另一种机器的编译器。

（2）独立于机器的代码优化器可以作用于这种中间表示。第 9 章将介绍这种代码优化。因此，虽然可以直接翻译源程序并生成目标代码，但编译器一般都采用中间表示形式。

本章将用 4.5 节和 5.3.3 小节介绍的树访问方法来描述编程语言的构造怎样被翻译成独立于机器的中间表示。为简单起见，假定对源程序的分析和静态检查已经完成，如图 7.1 所示。

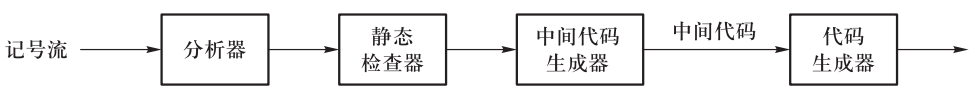


图 7.1　中间代码生成器的位置

实际的编译器可能构造一系列的中间表示。高级一点的中间表示接近源语言，语法树就是其中一种，它们适合于完成静态类型检查等任务。低级的中间表示接近目标机器，它们适合于完成依赖于机器的任务，如寄存器分配和指令选择。本章介绍的三地址代码（包括 LLVM IR）对高级和低级中间表示都适用，区别主要在于所选择的运算符。对于表达式，语法树和三地址代码中的表示方式差异较小。但对于控制流语句，如循环语句，语法树侧重表示该语句的各组成成分，而三地址代码则通过标号和跳转指令来体现控制流，类似于机器语言的表达方式。

中间表示设计的选择随编译器的不同而不同。中间表示可以是一种实际的语言，也可以是编译各阶段共享的内部数据结构。C 是一种编程语言，但它经常被作为一种中间形式，这是因为它灵活，能生成高效的机器代码，并且它的编译器到处可用。例如，最初的 C++ 编译器就是由一个生成 C 的前端和作为后端的 C 编译器组成的。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第267页 文件编号: 422624 页面编号: 32674538445 打印人: 8685 2026-03-09 10:23:03

7.1 中间语言

4.2.1 小节介绍的语法树是一种图形化的中间表示。本节再介绍几种常用的中间表示：后缀表示、其他图形表示和三地址代码。本章主要使用三地址代码。从编程语言的各种构造产生三地址代码的语义规则类似于产生语法树或后缀表示的那些规则。

7.1.1 后缀表示和前缀表示

表达式 E 的**后缀表示**（逆波兰式）可以归纳定义如下：

（1）如果 E 是变量或常数，那么 E 的后缀表示就是 E 本身。

（2）如果 E 是形式为 $E_1 op E_2$ 的表达式（其中 op 是任意的二元运算符），那么 E 的后缀表示是 $E_1'\ E_2' op$，其中 E_1' 和 E_2' 分别是 E_1 和 E_2 的后缀表示。

（3）如果 E 是形式为 (E_1) 的表达式，那么 E_1 的后缀表示也是 E 的后缀表示。

后缀表示不需要括号，因为运算符的位置及其运算对象的个数使得后缀表示仅有一种解释。例如，$(8-5)+2$ 的后缀表示是 8 5 − 2 +，而 $8-(5+2)$ 的后缀表示是 8 5 2 + −。

上面的定义很容易拓广到含一元运算符的表达式。

后缀表示的最大优点是便于计算机处理表达式。利用一个栈，自左向右扫描表达式的后缀表示：每碰到运算对象，就把它压进栈；每碰到运算符，就从栈顶取出相应个数的运算对象进行计算，再将结果压进栈；最终的结果留在栈顶。

后缀表示也可以拓广到表示赋值语句和控制语句，但很难用栈来描述控制语句的计算。

前缀表示（波兰式）是一种用于逻辑、算术和代数的表达方式，其格式为操作符在前、操作数在后的形式，如 $op(a,b,c)$。这种表示方法的优势在于简化命题逻辑的表达，尤其适合用于表示上层的逻辑规范。

7.1.2 图形表示

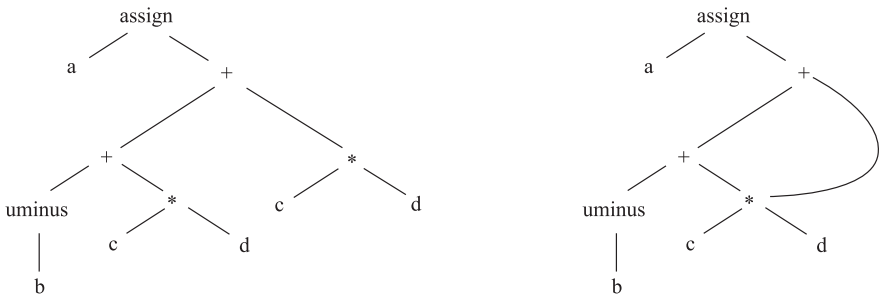
语法树是一种图形化的中间表示，它是解析树的浓缩表示，描绘了源程序在语义上的层次结构。后缀表示是语法树的一种线性表示。例如，赋值语句 a = (-b + c * d) + c * d 的语法树如图 7.2(a) 所示，对应的后缀表示为 a b uminus c d * + c d * + assign。

有向无环图（directed acyclic graph，DAG）也是一种中间表示。与语法树相比，它以更紧凑的方式给出同样的信息，因为公共子表达式标识出来了。在图 7.2(b) 所示的 DAG 中，公共子表达式 c*d 不止一个父结点。在考虑代码优化时，有向无环图比语法树更适用。

表 7.1 给出的构造赋值语句语法树的语法制导定义是 4.2.2 小节构造表达式语法树的语法制导定义的一个拓展，其中 mkUNode(op, child) 是构造一元运算结点的函数。这里用的是二义文法，假定运算符的结合性和优先关系与通常的一样，虽然没有把它们加入文法。这个

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第268页 文件编号: 422624 页面编号: 32684538443 打印人: 8685 2026-03-09 10:23:03

定义从输入 a = (b + cd) + cd 构造出如图 7.2(a) 所示的语法树（按照该定义，叶结点的形式应该像图 4-5 所示的叶结点那样）。



(a)语法树　　(b)有向无环图

图 7.2　a = (-b + c * d) + c * d 的图形表示

表 7.1　构造赋值语句语法树的语法制导定义

产生式	语义规则
$S \to \mathbf{id} = E$	S.nptr=mk Node (′ assign′, mkLeaf(**id ， id**. entry), E.nptr)
$E \to E_1 + E_2$	E.nptr=mk Node(′+′, E_1. nptr, E_2. nptr)
$E \to E_1 * E_2$	E.nptr=mk Node (′ ∗ ′, E_1. nptr, E_2. nptr)
$E \to -E_1$	E.nptr=mk UNode（′ uminus ′, E_1, nptr）
$E \to (E_1)$	E.nptr=E_1 nptr
$E \to \mathbf{id}$	E.nptr=mkLeaf（**id，id**. entry）

修改构造结点的函数，仍然用这个语法制导定义，可以构造出有向无环图。如果构造结点的函数首先检查是否已经有相同的结点存在，有则返回先前构造的这种结点的指针而不是构造新结点，这样就可以得到 DAG。这个语法制导定义从输入 **a = (b + c*d) + c*d** 构造出的 DAG 如图 7.2(b) 所示。

7.1.3　三地址代码

三地址代码是一种一般形式为

$$x = y\ op\ z$$

的指令（也称语句）序列。其中，x、y 和 z 是名字、常数或编译器产生的临时变量；op 代表运算符，如定点或浮点算术运算符，或是对布尔类型数据操作的逻辑运算符等。之所以称为**三地址代码**，是因为每条指令通常包含 3 个地址，即两个运算对象的地址和一个结果的地址。因为每条指令的右边只有一个运算符，所以源语言的表达式 x + y * z 翻译成的三地址指令序列是

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第269页 文件编号: 422624 页面编号: 32694538441 打印人: 8685 2026-03-09 10:23:03

$t_1 = y * z$

$t_2 = x + t_1$

其中，t_1 和 t_2 是编译器产生的临时名字。采用临时名字保存中间结果，可以比较容易地为三地址代码重新安排计算次序；而在后缀表示上要改变计算次序是很困难的。

三地址代码是语法树或 DAG 的一种线性表示，其中新增加的临时名字对应图的内部结点。图 7.2 所示的语法树和 DAG 分别由图 7.3(a) 和图 7.3(b) 中的三地址指令序列表示。

$t_1 = -b$
$t_2 = c * d$
$t_3 = t_1 + t_2$
$t_4 = c * d$
$t_5 = t_3 + t_4$
$a = t_5$

(a) 对应语法树的代码

$t_1 = -b$
$t_2 = c * d$
$t_3 = t_1 + t_2$
$t_4 = t_3 + t_2$
$a = t_4$

(b) 对应 DAG 的代码

图 7.3 对应图 7.2 的语法树和 DAG 的三地址代码

三地址指令类似于汇编代码。指令可以有符号标号，而且可以有控制流指令。下面是本书常用的三地址指令，其中符号标号 L 是三地址指令序列中某条三地址指令的索引：

（1）形式为 $x = y\ op\ z$ 的赋值指令，其中 op 是二元算术或逻辑运算符。该指令被称为**引用**了 y 和 z 的值，并对 x 进行了**定值**。

（2）形式为 $x = op\ y$ 的赋值指令，其中 op 是一元运算符。一元运算符主要包括一元减、逻辑否定、移位和类型转换运算符。

（3）形式为 $x = y$ 的复写指令。

（4）无条件转移 goto L 。标号为 L 的指令是下一步将要执行的三地址指令。

（5）形如 if x relop y goto L 的条件转移。这种指令用于 x 和 y 的关系运算（如 <、== 和 >= 等）。如果关系成立，执行标号为 L 的指令；否则，按照通常的顺序，执行 if x relop y goto L 的下一条三地址指令。

（6）param x 和 call p, n 用于过程调用，其中 n 表示实参个数。return y 用于过程返回，y 代表返回值，它是可选的。过程调用的典型使用是

param x_1

param x_2

$\cdots$

param x_n

call p, n

这些指令是过程调用 $p(x_1, x_2, \cdots, x_n)$ 的中间代码。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第270页 文件编号: 422624 页面编号: 32704538449 打印人: 8685 2026-03-09 10:23:03

（7）形式为 $x = y[i]$ 和 $x[i] = y$ 的索引赋值。第一条指令将 y 的存储单元之下的第 i 个存储单元的值赋给 x。第二条指令将 y 的值赋给 x 的存储单元之下的第 i 个存储单元。在这些指令中，x、y 和 i 都代表数据对象。

（8）形式为 $x = \&y$、$x = *y$ 和 $*x = y$ 的地址和指针赋值。第一条指令将 y 的存储单元地址赋给 x。可以猜想 y 是名字（或临时变量），表示像 a 和 A[i][j] 这样有左值的表达式，x 是指针名字或临时变量，即 x 的右值是某个对象的左值。在第二条指令中，y 是一个指针或临时变量，其右值是一个存储单元，x 的右值被设置为该存储单元的内容。第三条指令 $*x = y$ 将 y 的右值放入 x 指向的存储单元。

选择适当的运算符集合是中间代码设计的重要问题。显然，它必须足够大以实现源语言的操作。较小的运算符集合易于在新的目标机器上实现，但较小的算符集合可能迫使前端为源语言的某些操作生成较长的三地址指令序列，此时如果想产生好代码，优化器和代码生成器的工作会比较艰巨。

三地址指令是中间代码的抽象形式。在编译器中，三地址指令可以用记录来实现，这种记录有运算符域和运算对象域。不同的编译器可能对这种记录的设计有所不同。

7.1.4 静态单赋值形式

静态单赋值形式（static single-assignment form，SSA）是一种便于某些代码优化的中间表示。SSA 有两个显著特点可区别它和三地址代码。第一个特点是，SSA 中所有的赋值指令都是对不同变量的赋值，所以才有“静态单赋值”这个术语。图 7.4 是用三地址代码和 SSA 写的同一个中间语言程序。注意，在 SSA 表示中，使用不同下标的 p 和 q 代表不同的变量。

p = a+b	$p_1 = a+b$
q = p−c	$q_1 = p_1 - c$
p = q * d	$p_2 = q_1 * d$
p = e−p	$p_3 = e - p_2$
q = p+q	$q_2 = p_3 + q_1$
(a) 三地址代码	(b) 静态单赋值形式

图 7.4 用三地址代码和 SSA 写的中间语言程序

在一个程序中，同一个变量可能在不同的控制流路径上都有定值。例如，源程序

```
if (flag) x = 1; else x = 1;
y = x + a;
```

有定值 x 的两条不同控制流路径。如果在该条件语句的两个分支中为 x 使用不同的名字，那么在赋值 y = x + a 中应该为 x 使用哪个名字？这就是 SSA 的第二个特点。它使用一种记号上的约定，称为 ϕ 函数，以组合 x 的两个定值：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第271页 文件编号: 422624 页面编号: 32714538447 打印人: 8685 2026-03-09 10:23:03

if (flag) $x_1 = -1$; else $x_2 = 1$;

$x_3 = \phi(x_1, x_2)$;

在这里，如果控制流通过条件为真的部分，则 $\phi(x_1, x_2)$ 的值是 x_1；如果控制流通过条件为假的部分，则 $\phi(x_1, x_2)$ 的值是 x_2。也就是说，$\phi(x_1, x_2)$ 返回其哪个变元的值，取决于到达 $\phi(x_1, x_2)$ 时所经过的控制流路径。

7.2 基本块和流图

三地址语句序列的一种图形表示称为**流图**。流图的结点代表一个顺序计算的三地址语句序列，边代表控制流。即使代码生成算法不明显构造流图，流图对于理解代码生成算法也是有用的。第 8 章将充分利用流图作为从中间代码收集信息的媒介。

7.2.1 基本块

基本块是一个连续的三地址语句序列，控制流从它的开始进入，并从它的末尾离开，没有停止或分支的可能性（末尾除外）。下面的三地址语句序列形成一个基本块：

$$\begin{aligned} t_1 &= a * a & t_4 &= t_1 + t_3 \\ t_2 &= a * b & t_5 &= b * b \\ t_3 &= 2 * t_2 & t_6 &= t_4 + t_5 \end{aligned} \tag{7.1}$$

下面的算法可用于把三地址语句序列分成基本块。

算法 7.1 划分基本块。

输入：三地址语句序列。

输出：基本块列表，每条三地址语句仅在一个基本块中。

方法：

（1）确定所有的**入口语句**（基本块的第一条语句）。规则如下：

（a）该中间代码的第一条语句是入口语句。

（b）任何作为条件转移或无条件转移目标的语句是入口语句。

（c）紧跟在条件转移或无条件转移之后的语句是入口语句。

（2）对于每条入口语句，它所在的基本块由从它开始一直到程序结束为止或下一条入口语句为止（但不含该入口语句）的所有语句组成。

【例 7.1】 考虑图 7.5 所示的源代码段，它计算两个长度为 20、元素类型为 int 的向量 a 和 b 的点积。在目标机器上完成这个计算的三地址语句序列如图 7.6 所示。需要注意的是，高级语言程序中数组元素（如图 7.5 中的 a[i] 和 b[i]）的地址偏移是基于元素类型的大小进行的，而在三地址码中，元素地址需要转换为**按字节寻址**的偏移方式。例如，图 7.6 中的

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第272页 文件编号: 422624 页面编号: 32724538445 打印人: 8685 2026-03-09 10:23:03

第 (3) 和 (4) 条指令用于计算 a[i]，首先根据索引 i 和元素类型的大小 4 来计算出偏移的字节数 t_1，然后通过数组基址 a 加上偏移量 t_1，将 a[i] 的值加载到 t_2。

```
prod = 0;
i = 1;
do {
    prod = prod+a[i] * b[i];
    i = i+1;
} while ( i <= 20 ) ;
```

图 7.5　计算点积的程序

把算法7.1 用到图 7.6 所示的三地址代码可知，语句 (1) 和 (3) 都是入口语句。语句 (3) 为入口语句的原因是最后一条语句可以转移到它。这样，语句 (1) 和 (2) 构成一个基本块，其余的语句形成一个基本块。

```
( 1 ) prod = 0                                  ( 7 ) t5 = t2 * t4
( 2 ) i = 1                                     ( 8 ) t6 = prod+t5
( 3 ) t1 = 4 * i                                ( 9 ) prod = t6
( 4 ) t2 = a[t1]        /* 计算 a[i] */         ( 10 ) t7 = i+1
( 5 ) t3 = 4 * i                                ( 11 ) i = t7
( 6 ) t4 = b[t3]        /* 计算 b[i] */         ( 12 ) if i <= 20 goto ( 3 )
```

图 7.6　计算点积的三地址代码

7.2.2　流图

可以把控制流信息加到基本块集合，形成一个有向图，用以表示程序，这样的有向图称为**流图**。流图的结点是基本块（简称块），有一个特殊的结点称为**初始结点**，它的入口语句是程序的第一条语句。对两个基本块 B_1 和 B_2，如果在程序的某个执行序列中 B_2 紧跟 B_1，那么 B_1 到 B_2 有一条有向边。也就是说，如果 B_1 的最后一条语句是条件转移或无条件转移到 B_2 的第一条语句，或者按程序正文的顺序 B_2 紧跟 B_1，并且 B_1 不是结束于无条件转移，那么 B_1 到 B_2 有一条有向边。称 B_1 是 B_2 的**前驱**，B_2 是 B_1 的**后继**。

【例 7.2】　图 7.6 所示程序的流图如图 7.7 所示。图 7.7 (a) 和图 7.7 (b) 分别是变换为 SSA 格式前后的流图。B_1 是初始结点。注意，最后一条语句原来是条件转移到语句 (3)，现已转移到由 B_2 开始的等价语句代替。在图 7.7 (a) 中，同一变量如 prod 和 i 可能被多次赋值，每次赋值都会覆盖之前的值，增加了数据依赖分析的复杂性。而在图 7.7 (b) 中，经过 SSA 转换，每个变量在每条路径上都只被赋值一次，通过引入带下标的变量来保持唯一性，使每个使用的变量关联的定值更加明确。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第273页 文件编号: 422624 页面编号: 32734538443 打印人: 8685 2026-03-09 10:23:03

在构造 SSA 格式时，通过流图图 7.7 (a) 可以快速找到 B_2 的前驱基本块 B_1 和 B_2；接着，从每个前驱基本块出发，按控制流逆向查找最近的对 i 和 prod 的定值。在 B_2 的入口处插入 $i_3 = \phi(i_1,i_2)$ 和 $prod_3 = \phi(prod_1,prod_2)$ 两条指令，从而确保每个变量赋值的唯一性。SSA 格式为编译器提供了优化和数据流分析的便利，使依赖关系更加清晰。

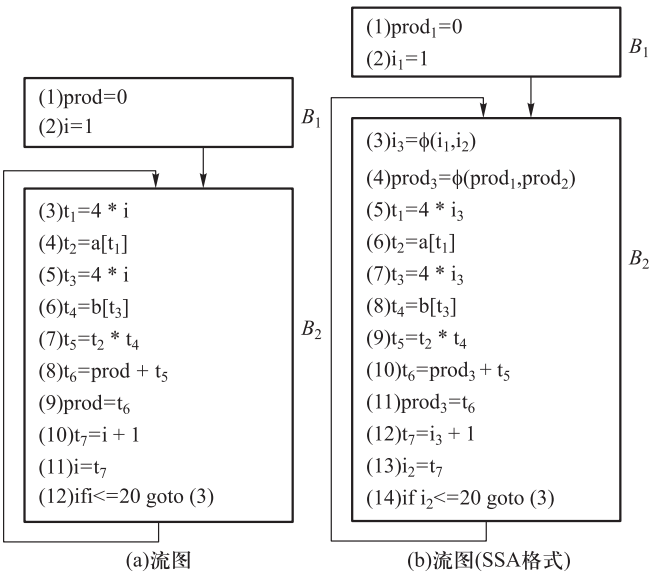


图 7.7 图 7.6 所示程序的流图

在流图中，什么是循环？怎样找出所有的循环？大多数情况下，这些问题是容易回答的。例如，图 7.7 存在一个循环，它由块 B_2 组成。对更一般情况的回答需要等第 9 章详细讨论之后才能作答。目前，只要知道循环是流图中满足下面条件的一组结点即可：

（1）该组中所有结点是**强连通**的，即从循环中任一个结点到另一个结点都有一条路径，路径上的所有结点都在这组结点中。

（2）该组结点有唯一的**入口**结点，从循环外的结点到达循环中任意一个结点的唯一方式是首先通过入口。

不包含其他循环的循环称为**内循环**。

7.3 LLVM 编译系统与 LLVM IR

LLVM（low-level virtual machine）[5] 是一个开源的编译器基础设施项目，旨在提供一种灵活的、模块化的编译架构，适用于开发编译器和代码优化工具。LLVM 项目由许多工具和库组成，支持从源代码编译到机器码生成的整个流程，包括前端解析、优化和后端代码生

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第274页 文件编号: 422624 页面编号: 32744538441 打印人: 8685 2026-03-09 10:23:03

成。其主要特性是支持跨平台和跨语言的代码优化和编译流程，并具有良好的扩展性。

7.3.1 LLVM 编译系统

如图 7.8 所示，LLVM 编译系统的核心由以下几个主要部分组成。

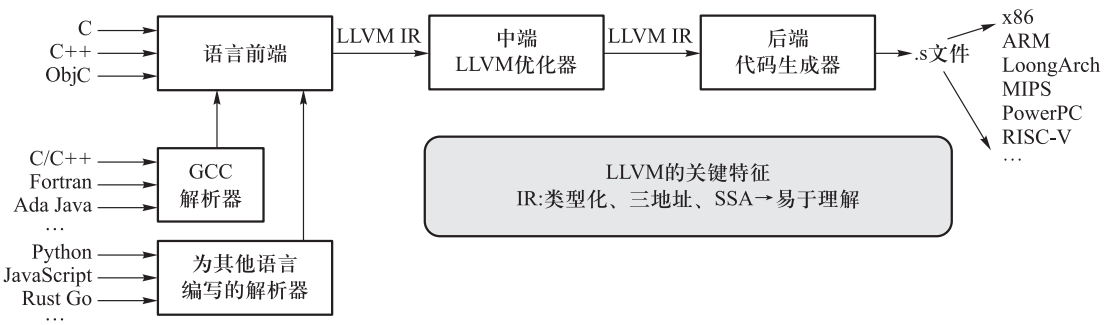


图 7.8 LLVM 编译系统概览

（1）**前端**（front-end）。前端将源代码转换为与平台无关的 LLVM IR，主要由 Clang 编译器处理，支持 C/C++/Objective-C。越来越多的编程语言也通过 GCC 解析器（如 FORTRAN、Ada、Java 等）或专门编写的解析器（如 Numba 用于 Python、js2llvm 用于 JavaScript、GoLLVM 用于 Go、rustc 用于 Rust）集成到 LLVM 中。这种多样化的前端支持使更多语言能够利用 LLVM 的优化和代码生成能力，从而提升性能和跨平台兼容性。

（2）**中端**（middle-end）。中端负责对中间表示进行各种优化和转换，主要集中在 LLVM IR 上，以提高程序性能。它执行各种优化操作，如常量传播、死代码删除、循环优化，同时保持程序语义不变。中端使用与平台无关的 IR，支持强类型系统，使得优化过程更高效。其设计允许开发者添加自定义优化遍（pass），并包含多种分析工具，帮助识别优化机会。通过这些功能，LLVM 中端显著提升了最终生成代码的性能和效率。该阶段的优化对不同平台通用。

（3）**后端**（back-end）。后端将优化后的 LLVM IR 转换为特定平台的机器码，并执行优化以提高生成代码的执行效率。它针对不同硬件架构（如 x86、ARM、LoongArch、RISC-V 等）实现特定代码的生成策略，并进行寄存器分配、指令选择和调度等，确保生成的代码能够高效运行。后端还可处理连接和汇编过程，生成最终的可执行文件。这种设计使 LLVM 实现了对多种平台的支持，提升了编译器的灵活性和可扩展性。

（4）**LLVM 工具链**。LLVM 提供系列工具以支持代码转换、连接、调试和编译等。例如，llvm-as（将 LLVM IR 的文本格式.ll 转换为二进制格式.bc）、llvm-config（打印 LLVM 编译选项）、llvm-dis（反汇编）、llc（编译成汇编）、lli（从 LLVM IR 执行程序）、opt（优化器）、llvm-link（连接器，连接多个.bc 文件）、lldb（调试工具），这些工具有助于用户更好地分析和优化代码。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第275页 文件编号: 422624 页面编号: 32754538448 打印人: 8685 2026-03-09 10:23:03

7.3.2 LLVM IR

LLVM IR 是一种平台无关、语言无关的中间表示，其设计既适用于高级语言的表示，也适用于低级机器代码的优化。它采用 RISC 风格的三地址代码结构，每一条指令有一个目标值和最多两个源操作数，并默认使用 SSA 格式，确保每个变量在定义后不再改变。SSA 格式使得变量的依赖关系更为清晰，有利于数据流分析和优化。此外，LLVM IR 提供无限的虚拟寄存器，以及简单、低级的控制流结构，具有模块化特性，便于组合不同模块的 IR 代码。它是强类型的，每个值都有明确的类型，有助于编译器进行检查和优化。

LLVM IR 提供人类可读的文本格式（.ll 文件）和机器可读的二进制格式（bitcode，.bc 文件）。文本格式便于开发者阅读和调试，而二进制格式则可以更高效地存储和加载。

可以用如下命令将 C 程序文件编译生成 LLVM IR：

```
gcc -S f.c -o f-gcc.s            # 编译生成汇编文件
clang -S f.c -o f-clang.s        # 编译生成汇编文件

clang -emit-llvm -S f.c -o f.ll # 编译生成.ll文件
clang -emit-llvm -c f.c -o f.bc # 编译生成.bc文件

lli f.ll                         # 执行f.ll
lli f.bc                         # 执行f.bc

llvm-as f.ll -o f.bc             # 汇编
llvm-dis f.bc -o f.ll            # 反汇编
llc f.bc  -o f.s                 # 编译生成汇编文件
```

表 7.2 给出了 C 语言程序与 LLVM IR 在相同功能上的不同表示。图 7.9 给出了一个简单的 C 程序及其 LLVM IR 程序的示例。LLVM IR 文件的前两行定义目标的数据布局 target datalayout 和目标类型 target triple，其中前者用来描述目标的字节序和类型长度信息。datalayout 中的第一个字母表示字节序，小写的 e 表示小端字节序，大写的 E 表示大端字节序；m:e 表示目标支持小端的内存模型。数据布局的类型信息以 type:<size>:<abi>:<preferred> 的格式提供。例如：

- p270:32:32、p271:32:32、p272:64:64 中的 p270、p271、p272 表示特定的指针地址空间编号（如 270、271 和 272），32:32 或 64:64 中的第一个数字表示指针的位宽，第二个数字表示指针对齐的位数；
- i64:64 表示按 64 位对齐的 64 位整数；
- f80:128 表示按 128 位对齐的 80 位浮点数（扩展精度格式）；
- n8:16:32:64 指定了该目标架构支持的原生整数宽度，即 8 位、16 位、32 位和 64 位，这些整数宽度在内存访问中是高效的；
- S128 表示栈的对齐要求为 128 位。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第276页 文件编号: 422624 页面编号: 32764538446 打印人: 8685 2026-03-09 10:23:03

表 7.2 C 语言程序与 LLVM IR 的对比

特性	C 语言程序	LLVM IR
作用域	文件、函数	module、function
类型	bool, char, int, ⋯, float, double struct{int, char}, int *, int [2][10]	i1, i8, i32, ⋯, float, double (iN 表示位宽为 N 的整型) {i32, i8}, i32*, [2 × [10 × i32]]
语句	一个含多个表达式的语句	形如 `x = y op z` 的指令序列
数据流	作用于变量的读/写序列	1. 加载值：将内存地址（变量）中的值加载到寄存器 2. 计算值：在寄存器上执行所需计算或操作 3. 存储值：将寄存器中的结果存回指定的内存地址
控制流	if, for, while, do while, switch 等	一组基本块，每个块以条件跳转（或返回）结束
变量定义	`int a = 5;`	`%a = alloca i32`

赋值	`a = a + 1;`	`%tmp = load i32, i32* %a`
		`%sum = add i32 %tmp, 1`
		`store i32 %sum, i32* %a`

条件语句	`if (a > 0)`
	`{`
	`  printf("Hello");`
	`}`
		`br i1 %cond, label %then, label %else`
		`%then:`
		`call void @printf(i8* getelementptr`
		`  inbounds (%str, i32 0, i32 0)`
		`br label %end`

返回值	`return 0;`	`ret i32 0`

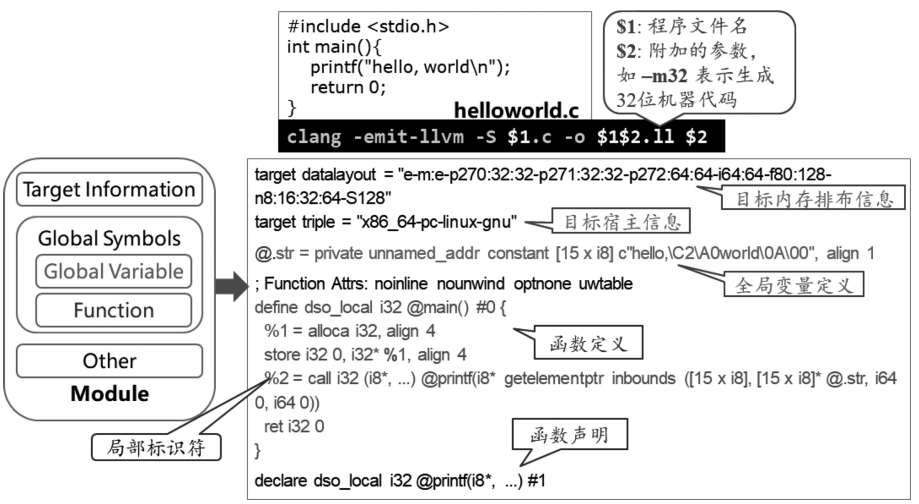


图 7.9 C 语言程序及其 LLVM IR 程序示例

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第277页 文件编号: 422624 页面编号: 32774538444 打印人: 8685 2026-03-09 10:23:03

LLVM IR 的程序结构为模块化且层次分明，包含模块（Module）、值（Value）、使用（User）、类型（Type）、全局变量（GlobalVariable）、函数（Function）、基本块（BasicBlock）、指令（Instruction）、控制流（Control Flow）和元数据（Metadata）等组成部分。其类体系如图 7.10 所示，通过层级划分模块、函数、基本块和指令，形成高效、可扩展的结构，支持编译器的优化和代码生成任务。这些类均定义在名为 llvm 的空间下。

```
Module
 |-- Function
      |-- BasicBlock
           |-- Instruction (User)
                |-- Value
                     |-- Type
Metadata
```

图 7.10 LLVM IR 类层次关系简图

值（Value）和使用（User）：非常基础的两个类，构成了 LLVM IR 的核心结构。Value 表示任何有类型的值，包括常量、变量、函数参数、指令结果等。几乎所有 LLVM IR 中的对象（如常量、指令、函数、全局变量等）都是 Value 类的派生类。每个 Value 都关联一个类型，确保 LLVM IR 的类型安全。User 继承自 Value，表示使用了其他 Value 对象的实体，即 User 包含指向其他 Value 的引用。User 类的实例包括指令和常量表达式，使用这些引用来形成指令间的依赖关系。通过 User 类，LLVM 可以追踪每个值的使用位置，这对优化和分析流程非常重要。

模块（Module 类）：LLVM IR 程序以模块为顶层容器，每个模块包含全局变量、函数和类型定义等实体。模块提供了如 getFunction()、getGlobalVariable() 等方法，用于访问特定的函数或全局变量；同时通过迭代器范围适配器 functions() 和 globals()，方便地遍历模块中的所有函数和全局变量。一个模块可由一个 C/C++ 文件编译生成，共享同一地址空间。

类型（Type 类）：LLVM IR 支持多种数据类型，其基类为 Type。该类及其子类用于描述各种类型，包括整型（IntegerType）、指针类型（PointerType）、数组（ArrayType）、结构体（StructType）、函数类型（FunctionType）等。此外，还有一些基本类型没有显式的类名，如单精度浮点型（FloatType）、双精度浮点型（DoubleType）、扩展精度浮点型（FP128Type）等，但可以通过 llvm::Type::getFloatTy、llvm::Type::getDoubleTy、llvm::Type::getFP128Ty 等获取。

全局变量（GlobalVariable 类）：指在模块级别定义的变量，它们的生命期贯穿整个程序的执行，并且可以被所有函数访问。它们通常在模块的开头定义，用 @ 符号来标识。

函数（Function 类）：每个函数定义在 LLVM IR 中由 Function 类描述，文本格式中以 define 关键字开头。函数定义包括函数的名称、返回类型、参数类型和函数体。函数体由一系列指令组成，指令实现了函数的具体逻辑。在函数体中，局部变量以 % 为前缀来标识。Function 类提供了多个关键方法，如 getEntryBlock() 用于获取入口基本块，getArg() 用于获取函数的某个参数，getReturnType() 返回函数的返回类型，args() 用于遍历所有函数参数。

基本块（BasicBlock 类）：函数体由一个或多个基本块组成，每个基本块由一系列连续执行的指令组成，没有控制流分支。每个基本块以一个标签（或称标号）标识，后接一组指

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第278页 文件编号: 422624 页面编号: 32784538442 打印人: 8685 2026-03-09 10:23:03

令。基本块之间可以通过条件跳转或无条件跳转指令连接。BasicBlock 类提供了关键方法，如 getTerminator() 用于获取基本块的终止指令，getFirstInstruction() 用于获取基本块中的第一条指令。

指令（Instruction 类）：指令是 LLVM IR 的基本操作单元，继承自 User 类。指令通常以三地址代码形式表示，每条指令包含一个目标值和最多两个源操作数。LLVM 提供了多种指令类型，如算术运算（BinaryOperator）、加载（LoadInst）、存储（StoreInst）、控制流（如 BranchInst、SwitchInst）、调用函数（CallInst）等。这些指令通过 getOpcode() 方法返回操作码，通过 getOperands() 方法返回操作数。

控制流：LLVM IR 支持条件跳转和无条件跳转，允许程序在基本块之间流转。常见的控制流指令包括 br 指令（条件跳转和无条件跳转）、ret 指令（函数返回）和 call 指令（函数调用）。通过这些指令，LLVM 能够实现各种控制结构，如条件、分支和循环。

元数据（Metadata 类）：LLVM IR 还可以包含元数据，用于提供关于程序的附加信息，如调试信息、优化提示等。元数据不会影响程序的执行，但对程序分析和优化非常重要。Metadata 类有多个子类，其中包括 MDNode（表示元数据结点）和 MDString（表示元数据字符串）。

【例 7.3】(函数结构示例) 对于如下 C 程序文件 f.c

```
1  double foo();
2  double bar(float a) {
3    return foo(a, 4.0) + bar(31337);
4  }
```

用 Clang v14 编译器执行 clang -emit-llvm -S f.c -o f.ll，可得到 LLVM IR 文本文件 f.ll，其中注释以 “;” 开头，bar 和 foo 的定义如图 7.11 所示。

```
1  ; Function Attrs: noinline nounwind optnone uwtable
2  define dso_local double @bar(float noundef %0) #0 {
3    %2 = alloca float, align 4
4    store float %0, float* %2, align 4
5    %3 = load float, float* %2, align 4
6    %4 = fpext float %3 to double
7    %5 = call double (double, double, ...) bitcast (double (...)* @foo to double (
         double, double, ...)*)(double noundef %4, double noundef 4.000000e+00)
8    %6 = call double @bar(float noundef 3.133700e+04)
9    %7 = fadd double %5, %6
10   ret double %7
11 }
12
13 declare double @foo(...) #1
```

图 7.11 例 7.3 所示程序对应的 LLVM IR

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第279页 文件编号: 422624 页面编号: 32794538440 打印人: 8685 2026-03-09 10:23:03

第 1 行注释用来指示 bar 函数的一些属性，noinline 表示不允许编译器内联此函数，nounwind 表示函数不会引发异常，optnone 禁用优化，uwtable 表示该函数使用异常处理表。

第 2 行开始定义名为 @bar 的函数，它接收一个 float 类型参数%0，dso_local 表示此函数在同一链接单元（如同一可执行文件或共享库）内被解析，dso 是 dynamic shared object 的缩写。这种标记允许编译器针对这些符号进行优化，如避免动态链接时的符号解析检查，允许编译器进行直接寻址或内联优化。

第 3 行用 alloca 指令在栈上分配一个 float 浮点数的空间，并将这个分配的内存地址（指针）存储在虚拟寄存器 %2 中，同时确保内存按 4 字节对齐。

第 4 行将 float 类型参数%0 的值存入虚拟寄存器%2（类型为 float*）指向的单元。

第 5 行加载参数值到虚拟寄存器%3。

第 6 行使用 fpext 指令将单精度浮点数 %3 扩展为双精度浮点数 %4，即由编译器进行自动类型提升：float 提升为 double。

第 7 行调用名为 foo 的函数，bitcast 指令用于将 foo 强制转换为至少有两个 double 型参数的函数指针类型，使其匹配该调用点的实参类型。

第 8 行递归调用 bar 函数，将参数 31337 处理成 float 类型来传递，函数返回值保存到%6。

第 9 行执行 double 类型的浮点数加法运算 fadd。

第 10 行通过 ret 指令将 float 类型计算结果%7 返回给调用者。

第 13 行声明一个外部函数 foo，其返回类型为双精度浮点数，可以接收可变数量的参数。

【例 7.4】(基本块和流图示例)　对于如下 C 程序文件

```
1  #define N 10
2  void f(int A[])
3  {
4      int i, P;
5      for (i = 0; i < N; ++i)
6          Sum(&A[i], &P);
7  }
```

用 Clang v14 编译可得到 LLVM IR，其中函数 f() 的定义如图 7.12 所示。

第 2 行是分配存放形参值的空间，第 4、5 行分配局部变量 i 和 P 的空间，分别保存到虚拟寄存器%2 和%4。

第 8 行是无条件跳转到标号为 5 的语句“br label 标号”。

第 10 行是编号为 5 的标号语句，其注释 preds = %14, %1 指示基本块 5 的前驱基本块是编号为 14 和 1 的两个基本块。

第 18、19 行通过 getelementptr(gep) 获取元素 A[i] 的地址，其中的 inbounds 表示 i 超出 10 (%11)，则 gep 返回 poison value（“毒值”）。Poison value 是 LLVM IR 中的一个特殊概念，用来表示未定义、无效或不安全的值。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第280页 文件编号: 422624 页面编号: 32804538448 打印人: 8685 2026-03-09 10:23:03


```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local void @f(i32* noundef %0) #0 {
  %2 = alloca i32*, align 8
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32* %0, i32** %2, align 8
  store i32 0, i32* %3, align 4
  br label %5

5:                                                ; preds = %14, %1
  %6 = load i32, i32* %3, align 4
  %7 = icmp slt i32 %6, 10
  br i1 %7, label %8, label %17

8:                                                ; preds = %5
  %9 = load i32*, i32** %2, align 8
  %10 = load i32, i32* %3, align 4
  %11 = sext i32 %10 to i64
  %12 = getelementptr inbounds i32, i32* %9, i64 %11
  %13 = call i32 (i32*, i32*, ...) bitcast (i32 (...)* @Sum to i32 (
        i32*, i32*, ...)*)(i32* noundef %12, i32* noundef %4)
  br label %14

14:                                               ; preds = %8
  %15 = load i32, i32* %3, align 4
  %16 = add nsw i32 %15, 1
  store i32 %16, i32* %3, align 4
  br label %5, !llvm.loop !6

17:                                               ; preds = %5
  ret void
}
```

图 7.12 例 7.4 所示程序对应的 LLVM IR

从例 7.4 的 LLVM IR 可见，每个基本块都有唯一的标号，基本块之间的控制流依赖通过基本块的标号和控制指令（如 br 和 cond br）来建立，形成控制流图。在生成的 LLVM IR 文本格式中，注释中的 preds 指示基本块在流图中的直接前驱，从而清晰地表达流图中的控制依赖。

7.3.3 LLVM 的遍管理器

在 LLVM 中，**遍**（pass）是对代码进行分析或优化的操作单元，旨在逐步改进生成的代码。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第281页 文件编号: 422624 页面编号: 32814538446 打印人: 8685 2026-03-09 10:23:03

1. 遍的类型

LLVM 中的遍（pass）根据功能不同一般分为分析、转换、使用（utility passes）3 种类型：

（1）分析遍（analysis pass）：不改变代码，仅收集信息，如数据流分析、控制流分析、依赖分析等。分析结果可供其他优化遍使用。

（2）转换遍（transform pass）：改变代码结构以优化性能，如删除冗余代码、循环展开、循环分裂、常量折叠等。

（3）使用遍（utility pass）：提供辅助功能，帮助其他遍完成任务，如特定的资源管理、报告生成、日志记录等。

根据优化的作用范围或优化阶段还可以进一步分成：

（1）过程间优化遍（interprocedural optimization pass, IPO）：进行跨函数或模块的优化，如内联（inline）优化、全局变量优化、跨模块分析等。

（2）机器码级遍（machine code level pass）：在生成的机器码上应用的遍，用于指令选择、寄存器分配、指令调度等低级优化。

所有 LLVM 中的遍都是 Pass 类的子类，它们通过重写从遍继承的虚方法来实现功能。根据遍访问代码的范围，LLVM 中的遍可以进一步分为模块遍（ModulePass）、调用图遍（CallGraphSCCPass）、函数遍（FunctionPass）、循环遍（LoopPass）或区域遍（RegionPass）。

2. 遍管理器

在 LLVM 中，遍管理器（pass manager）是用于管理和调度不同优化遍的工具。它负责组织和调度遍的执行、管理依赖关系以及遍执行过程中的资源。遍管理器允许在编译过程中动态地应用不同的优化遍，并确保各个遍按正确的顺序和条件执行，从而提高代码优化的效率。LLVM 提供了以下几个主要的遍管理器类型：

（1）较旧的遍管理器接口（LegacyPassManager）：它支持对单个模块、函数或循环应用遍。虽然它被新的遍管理器所取代，但在某些老版本的 LLVM 中仍然存在。可以使用遍管理器来调度和运行不同的遍：

```
    llvm::legacy::PassManager passManager;

    // 添加优化遍到遍管理器
    passManager.add(llvm::createDeadCodeEliminationPass());  // 删除死代码
    passManager.add(llvm::createConstantPropagationPass());  // 常量传播

    // 运行所有添加的遍
    passManager.run(module);
```

（2）流水线遍管理器（PipelinePassManager）：LLVM 10 及以后的版本引入了新的遍管理器 PipelinePassManager。这是一个更加灵活和高效的遍调度管理器，提供了更好的依赖管理和执行性能。PipelinePassManager 通过流水线形式将多个优化遍按顺序执行，并且每个遍的结果可以传递给下一个遍，从而实现更精细的优化控制。例如：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第282页 文件编号: 422624 页面编号: 32824538444 打印人: 8685 2026-03-09 10:23:03


```
// 创建 PassBuilder
llvm::PassBuilder PassBuilder;
llvm::ModulePassManager MPM;
llvm::FunctionPassManager FPM;

// 添加多个遍到流水线
PassBuilder.registerModuleAnalyses(MPM);
// 将 ModulePassManager 和 FunctionPassManager 适配在一起
// 这样，MPM 可以调用并执行 FPM 中的遍
PassBuilder.registerModuleToFunctionPassAdaptor(MPM);

// InstSimplifyPass 是函数遍
FPM.addPass(InstSimplifyPass());
MPM.addPass(createModuleToFunctionPassAdaptor(std::move(FPM)));
MPM.run();
```

（3）模块遍管理器（ModulePassManager）：作用于整个模块，适用于全局优化或跨函数的优化，如常量合并、内联函数等。

（4）调用图遍管理器（CGSCCPassManager）：用于分析和优化基于调用图的信息，通常用于跨函数优化。

（5）函数遍管理器（FunctionPassManager、FPPassManager）：用于管理和调度针对单个函数的遍（如 FunctionPass），负责函数级别的优化。

（6）循环遍管理器（LoopPassManager、LPPassManager）：用于管理和调度针对循环的遍（如 LoopPass），负责对循环进行更细粒度的优化，如循环展开、循环分裂等。

（7）区域遍管理器（RGPassManager）：用于管理和调度区域级遍（如 RegionPass）。它针对程序的特定代码区域进行优化，如嵌套循环、条件分支等。

（8）基本块遍管理器（BBPassManager）：用于对单个基本块进行优化和分析，不涉及跨基本块的依赖，因此能更高效地执行优化任务、提升编译效率。

LLVM 中的遍可能依赖于其他遍。例如，某些转换遍需要先完成特定的分析遍，才能基于分析结果进行优化。LLVM 的遍管理器会自动处理遍的依赖关系，以确保在合适的顺序执行遍。每类遍管理器有各自的约束，如 FunctionPass 只能查看当前函数，不维护跨函数之间的状态。

3. 常见的 LLVM 遍

一些典型的 LLVM 遍包括：

- mem2reg：将内存上的局部变量提升到寄存器中，利用 SSA 优化。
- dead code elimination（DCE）：移除没有使用的代码。
- loop unrolling：展开循环体，提高指令并行性。
- inline：在调用处展开内联函数，减少函数调用开销。

开发者可以使用 opt 工具来运行特定的遍。例如：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第283页 文件编号: 422624 页面编号: 32834538442 打印人: 8685 2026-03-09 10:23:03


```
opt -mem2reg -loop-unroll < input.ll > output.ll
```

在代码中，开发者也可以直接通过 API 向遍管理器添加遍，并按照需求应用到代码上。

7.3.4 LLVM 的应用场景

LLVM 编译系统广泛应用于多个领域，包括：

（1）编译器开发：LLVM 为编译器提供了一个强大的后端支持，可以快速开发多语言编译器。

（2）程序优化：LLVM 提供了丰富的优化工具，适用于各种场景下的代码优化，如编译时优化和运行时优化。

（3）跨平台支持：由于 LLVM 的中间表示是平台无关的，因此很容易生成不同平台的代码。

（4）动态编译和即时编译：LLVM 支持 JIT（Just-in-Time，即时编译）技术，适用于动态语言和虚拟机。

7.4 中间代码生成概述

与 5.3.3 小节的类型检查类似，中间代码生成同样可以采用边解析边生成中间代码，也可以基于树访问来生成中间代码，后者是现代编译器的主流做法。本章将以基于树访问的中间代码生成方法为主来介绍。

1. 中间代码生成中的关键问题

在中间代码生成中，若采用类似三地址码的中间语言，则中间代码生成的关键问题包括类型和符号表的变化与管理、语句的翻译以及控制流的处理。

（1）类型与符号表的变化：在中间代码生成过程中，源语言中的多样化类型会被转换为基本类型（整型、浮点型等），以及基于已有类型构造的派生类型（如数组、结构体等）。对于用户自定义的类型，编译器需要通过符号表来管理这些类型的详细信息，除了记录类型属性，还需要包括字宽（width，即在内存中占用的字节数）、起始地址（address，即变量在数据区中的相对地址）。特别需要注意的是，在三地址码中，数据是以字节为单位寻址的，复合类型（如结构体、共用体等）的内存布局和对齐方式也需要在中间代码生成时进行精确计算。

（2）语句的翻译：声明语句本身不直接生成指令，但会更新符号表，不同作用域的变量可能会用不同的符号表管理。在中间代码生成过程中，会根据变量声明的类型计算变量的字宽，为其分配存储单元，并将字宽和所存放的相对地址信息登记到符号表中。赋值语句需要计算右值并将其存储到左值，可能需要引入临时变量来存储中间计算结果，计算数组或结构体元素的地址，或进行类型转换（例如，将整型赋值给浮点型）。控制流语句则涉及跳转目标

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第284页 文件编号: 422624 页面编号: 32844538440 打印人: 8685 2026-03-09 10:23:03

的确定，通常通过标号或回填技术实现，同时还会对逻辑运算中的短路计算进行优化，以避免不必要的计算。

2. 符号表的设计

在第 5 章类型检查后的符号表中，每个符号条目会包含标识符、存储类别（如 extern、static、register）、类型信息等。类型信息一般由 < 类型标识，该类别关联的信息 > 组成。例如，数组类型可以表示为 <Array, (len, elemType)>（len 和 elemType 分别表示一维数组的长度和元素类型），指针类型则表示为 <Pointer, T>（T 为指针指向的类型）。

在本章，符号表会有一些变化。首先，需要管理符号的作用域，可以使用不同的符号表来处理不同作用域的符号。其次，对于变量，符号条目要增加字宽、存储的相对地址（以字节为单位）；对于复合类型，如记录类型或共用体类型等，则需要用符号表进一步管理各个成员的字宽和相对地址。相对地址是对静态数据区基址的偏移，或是对活动记录中某个基址的偏移。

前端分配地址时必须考虑目标机器的一些特点，如目标机器的指令系统可能偏爱数据对象和它们地址的某种布局。这里的讨论忽略数据对象的对齐等问题。

3. 一个简单的语言 IMP

表 7.3 所示的第 2 列给出了一种简单命令式示例语言 IMP 的文法，该语言的程序 P 由声明 D 和语句 S 组成。声明 D 可以是声明列表或单个变量声明；类型 T 包括布尔类型、整型、实型、数组类型、指针类型。语句 S 涵盖赋值、条件、循环和顺序语句。表达式 E 支持常数 **num**、变量 **id**、二元算术表达式、一元算术表达式、带括号的表达式；条件表达式 B 包括逻辑或、逻辑与、逻辑非表达式，带括号的逻辑表达式、关系表达式以及真值和假值。IMP 语言的扩展功能还包括过程声明、记录类型和调用语句。

本章后续内容将以 IMP 语言及其扩展为例，介绍中间代码生成方法。为便于讨论，假设示例语言中每种语法结构对应的语法树结点类名及子结构参数如表 7.3 第 3 列所示。中间代码生成通过为这些结点编写合适的 visit××、enter××、exit×× 方法来实现，逐步构建中间表示，其中 ×× 是结点的类名。

表 7.3 IMP 语言的文法及语法树结点

编号	产生式	语法树结点 (子结构参数)	描述
(P)	$P \to D; S$	P(Decls, Stmts)	程序
(D1)	$D \to D; D$	Decls	声明列表
(D2)	$D \to \textbf{id} : T$	DId(id, T)	变量声明
(T1)	$T \to \textbf{boolean}$	TBool	布尔类型
(T2)	$T \to \textbf{integer}$	TInt	整型
(T3)	$T \to \textbf{real}$	TReal	实型
(T4)	$T \to \textbf{array[num] of } T$	TArr(num, T)	数组类型
(T5)	$T \to *T$	TPtr(T)	指针类型
(E1)	$E \to E_1 + E_2$	EBop(op, E_1, E_2)	二元算术表达式

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第285页 文件编号: 422624 页面编号: 32854538447 打印人: 8685 2026-03-09 10:23:03

续表

编号	产生式	语法树结点 (子结构参数)	描述
(E2)	$E \to -E_1$	EUop(op, E_1)	一元算术表达式
(E3)	$E \to (E_1)$	EPara(E_1)	带括号的表达式
(E4)	$E \to$ **id**	EId	变量
(E5)	$E \to$ **num**	ENum	常数
(S1)	$S \to$ **id** $:= E$	SAss(id, E)	赋值语句
(S2)	$S \to$ **if** B **then** S_1	SIft(B, S_1)	if-then 语句
(S3)	$S \to$ **if** B **then** S_1 **else** S_2	SIfte(B, S_1,S_2)	if-then-else 语句
(S4)	$S \to$ **while** B **do** S_1	SWhile(B, S_1)	while 语句
(S5)	$S \to S; S$	Stmts	顺序语句
(B1)	$B \to B_1$ **or** B_2	LOr(B_1,B_2)	逻辑或表达式
(B2)	$B \to B_1$ **and** B_2	LAnd(B_1,B_2)	逻辑与表达式
(B3)	$B \to$ **not** B_1	LNot(B_1)	逻辑非表达式
(B4)	$B \to (B_1)$	LPara(B_1)	带括号的逻辑表达式
(B5)	$B \to E_1$ **relop** E_2	LRel(relop, E_1,E_2)	关系表达式
(B6)	$B \to$ **true**	LTrue	真值
(B7)	$B \to$ **false**	LFalse	假值
	语言的扩展		
(D3)	$D \to$ **proc id**; D_1; S	DProc(id,D_1,S)	过程定义
(T6)	$T \to$ **record** D **end**	TRec(D)	记录类型
(S1′)	$S \to L := E$	SAssL(L, E)	赋值语句
(E4′)	$E \to L$	ELval(L)	左值表达式
(E6)	$L \to$ **id**[Elist]	ELArr(id,$E_1, \cdots, E_n$)	数组引用
(E7)	$L \to$ **id**	ELId(id)	简单变量
(S6)	$S \to$ **call id** (Elist)	SCall(id, Elist)	调用语句

7.5 声明语句

C、Java、Pascal 和 FORTRAN 这些语言的语法允许一个过程中的所有声明集中在一起处理。在这种情况下，可以使用全局变量（如 offset）来跟踪当前可用的相对地址。每处理一个新声明，offset 就根据变量的类型和字宽进行更新，从而为下一个声明分配正确的内存位置。下面将讨论声明语句的翻译，分别针对源程序中仅包含主过程和允许自定义过程的情况进行分析。

7.5.1 仅包含主过程时声明语句的翻译

首先考虑 IMP 程序中仅包含主过程的情况。在图 7.13 所示的基于树访问的翻译方案中，对于程序结点 P，在检查第一个声明之前，即在 enterP 方法中，将全局整型变量 offset 初始

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第286页 文件编号: 422624 页面编号: 32864538445 打印人: 8685 2026-03-09 10:23:03

化为 0。每次为一个名字分配存储单元时，其偏移量等于 offset 的当前值，同时将 offset 增加该名字对应数据对象的宽度。需要注意的是，offset 初值可能不是 0，并且在某些情况下，如第 6 章中介绍的栈分配机制中，offset 可能逐步减小。

过程 enter(name, type, offset) 为名字 name 建立符号表条目，该名字的类型是 type，它在数据区的相对地址是 offset。属性 type 和 width 用来表示结点的类型和字宽。属性 type 代表从基本类型 integer 和 real 等，应用类型构造器 pointer 和 array 等构造出的类型表达式。如果类型表达式用图表示，那么属性 type 可以是一个指针，指向代表类型表达式的结点。

编号	事件处理方法	语义动作
（P）	enterP:	offset = 0;
（D1）	visitDecls:	foreach D in Dcels do visit(D);
（D2）	exitDId(**id**, T):	enter(**id**.lexeme, T.type, offset); offset = offset + T.width;
（T1）	exitTBool:	x.type = bool; x.width = 1;
（T2）	exitTInt:	x.type = integer; x.width = 4;
（T3）	exitTReal:	x.type = real; x.width = 8;
（T4）	exitTArr(**num**, T):	x.type = array (**num**.val, T.type); x.width = **num**.val $\times T$.width;
（T5）	exitTPtr:	x.type = pointer (T.type); x.width = 4;

图 7.13 计算被声明名字的类型和相对地址

在图 7.13 中，假设每个事件处理方法正在处理的语法树结点实例为 x，不同类型的数据对象具有各自的字宽：整数的字宽是 4 字节，实数的字宽是 8 字节，数组的总宽度是其每个元素的字宽乘以数组元素的个数，指针的字宽假定为 4 字节（在 64 位系统中则为 8 字节）。根据结点 x 的类型，翻译器将按图 7.13 中的语义动作计算字宽并更新偏移量 offset。

7.5.2 作用域信息的保存

现在考虑像 Pascal 这样允许过程嵌套定义的语言。为简单起见，仅讨论无参过程，并且认为过程不会递归。所讨论语言的文法如下：

$$\begin{aligned} P &\rightarrow D\ ;\ S \\ D &\rightarrow D\ ;\ D \mid \mathbf{id}\ :\ T \mid \mathbf{proc}\ \mathbf{id}; D; S \end{aligned} \tag{7.2}$$

在这样的语言中，局部于每个过程的名字仍然可以用图 7.13 所示的方式来分配相对地址，每个过程建立单独的符号表。这样，每个过程要有自己的符号表指针和 offset。

在一边扫描一边建立符号表和完成存储分配的情况下，当碰到内嵌的过程定义时，对外围过程声明的处理需要暂时停止，直到内嵌的过程定义处理完后再继续。可以用两个栈 tblStack 和 offsetStack 分别保存尚未处理完的过程的符号表指针和它们的 offset，这两个栈的栈顶元素分别是正在处理的过程的符号表指针和 offset。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第287页 文件编号: 422624 页面编号: 32874538443 打印人: 8685 2026-03-09 10:23:03

按照这种方式，文法式 (7.2) 的翻译方案如 图 7.14 所示，其中类型 T 的语义动作和图 7.13 的一致，在此略去；语句 S 的语义动作在 7.6 节介绍。

编号	事件处理方法	语义动作
（P）	enterP(Decls,Stmts) exitP(Decls,Stmts)	t = mkTable (null); push(t, tblStack); push (0, offsetStack); addWidth (top (tblStack), top (offsetStack)); pop(tblStack); pop (offsetStack);
（D2）	exitDId(**id**, T)	enter(top(tblStack), **id**.lexeme, T.type, top (offsetStack)); top(offsetStack) = top(offsetStack) + T.width;

（D3）	visitDProc(**id**, D_1, S)
	
	exitDProc(**id**,D_1, S)
		visit D_1 之前:
		t = mkTable(top(tblStack)); push(t,tblStack);push(0, offsetStack);
		// enterProc(top(tblStack), id.lexeme, t); // 支持过程递归调用
		t = top(tblStack); addWidth(t, top(offsetStack));
		pop(tblStack); pop(offsetStack);
		enterProc(top(tblStack), id.lexeme, t); // 不支持过程递归调用

图 7.14　处理嵌套过程中的声明

图 7.14 中的语义动作根据下面的辅助函数定义：

（1）mkTable(previous)。建立新的符号表，并返回新符号表的指针。参数 previous 指向先前建立的符号表，通常是直接外围过程的符号表。新符号表的首部不仅存储 previous 用于访问直接外围过程的符号表，还记录该符号表中所有局部变量所需的存储单元总量等信息。

（2）enter(table, name, type, offset)。在 table 指向的符号表中为变量名 name 建立新条目。和前面一样，enter 把类型 type 和相对地址 offset 置于该条目的域中。如果要说得再详细一些，那么本过程还需要完成检查名字是否重复定义等。

（3）addWidth(table, width)。把 table 指向的符号表中所有局部变量条目的字宽的累加值记录在该符号表的首部。

（4）enterProc(table, name, newTable)。在 table 指向的符号表中为过程名 name 建立新条目。参数 newTable 指向过程 name 本身的符号表。

再对图 7.14 中的语义动作做一些解释。在进入程序 P，即 enterP 时，需要完成一些初始化工作：调用操作 mkTable(null) 建立最外层作用域的符号表，并用两个 push 操作来完成栈 tblStack 和 offsetStack 的初始化。

对于过程声明 $D \rightarrow$ **proc id**; ND_1; S 对应的语法树结点 DProc(id, D_1,S)，在访问 D_1 之前，需要建立新的符号表，让它指向直接外围过程的符号表，并将新符号表指针压入栈 tblStack，将 0 压入栈 offsetStack。然后为 D_1 中声明的名字在该新符号表中建立条目，同时根据新的 offset 进行存储分配。在结束内嵌过程定义的扫描，即 exitDProc(id,D_1,S) 时，由 D_1 产生的所有声明的总字宽在 offsetStack 的栈顶上，用 addWidth 把它记录在符号表中，然后栈 tblStack 和 offsetStack 的顶元退栈，再把过程名 **id** 的条目建立在直接外围过程的符号

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第288页 文件编号: 422624 页面编号: 32884538441 打印人: 8685 2026-03-09 10:23:03

表中。这些都结束后，继续分析外围过程的声明。

按图 7.14 所示的翻译方案，为图 6.19 所示的 Pascal 程序生成的符号表在图 7.15 中给出（没有包括形式参数）。过程的符号表之间用双向链连接，从这个双向链可以知道过程的嵌套关系。例如，过程 readArray、exchange 和 quicksort 的符号表逆向指向直接外围过程 sort 的符号表，而 sort 的符号表中有 3 个指针指向这 3 个过程的符号表。

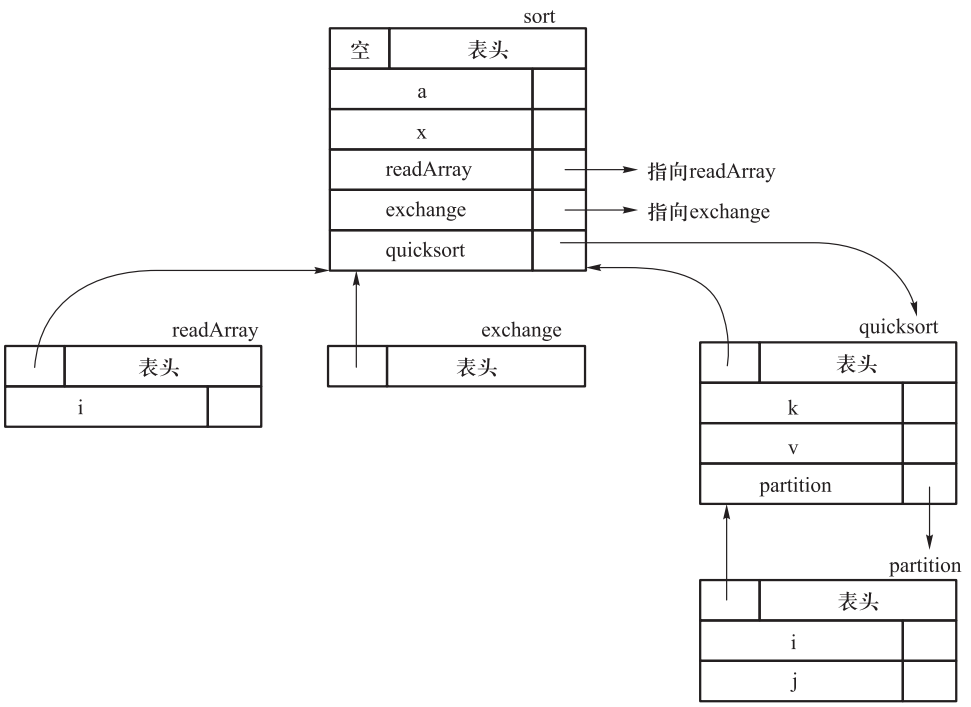


图 7.15 嵌套过程的符号表

当遇到对名字的引用时，如果在本过程的符号表中找不到该名字，就需要到它的外围过程的符号表中去找，符号表中的逆向指针可用于这个目的。如果语言有预定义的标准标识符（指程序员可以重新定义其含义的标识符），如 Pascal 语言的 integer、true 等，那么主过程符号表的逆向指针应该指向标准标识符的符号表，以便确定程序员没有定义的标识符是不是一个标准标识符。在此实际上已经解释了 7.6 节用到的符号表查找函数 lookup 的处理过程。

如果允许过程递归，图 7.14 所示的翻译方案需要在访问 D_1 之前就将该过程 **id** 加入到符号表，否则在 S 中有直接递归调用时在符号表查找会报告该 **id** 没有定义。也就是说，将 exitDProc(**id**,D_1, S) 中的enterProc(top(tblStack), id.lexeme, t); 移到 visit D_1 之前执行。

若是含有参数的过程，对构造符号表来说，形式参数的处理和其他局部名字的处理没有

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第289页 文件编号: 422624 页面编号: 32894538449 打印人: 8685 2026-03-09 10:23:03

很大的区别，但是填在条目中的属性，如存储分配信息等，会有些区别。

如果是一遍扫描的编译器，每个过程被扫描后，它的目标代码已经生成，若无其他需要，该过程的符号表可以释放。这时编译器可以将符号表组织成一个栈，碰到一个过程声明时将在该过程中声明的名字进栈，该过程被扫描结束时将它的符号全部弹出栈。

7.5.3 记录的域名

若在图 7.13 所示的 IMP 语言的文法上扩展增加记录类型

（T6）$T \rightarrow$ **record** D **end**　　TRec(D)

的翻译方案在图 7.16 中。语义动作强调了记录类型中域的存储布局与过程中局部名字在活动记录中布局的相似性。因为过程定义不影响图 7.14 中的宽度计算，因此把上面的产生式放宽到允许过程定义出现在记录类型中，这只是为了使翻译方案简洁易懂。

编号	事件处理方法	语义动作
（T6）	enterTRec(D):	t = mkTable(null); push(t, tblStack); push(0, offsetStack);
	exitTRec(D):	x.type = record (top(tblStack));
		x.width = top(offsetStack); pop(tblStack); pop(offsetStack);

图 7.16　为记录中的域名建立符号表

看见关键字 **record** 后，即 enterTRec(D) 时会为域名建立新的符号表。该符号表的指针入栈 tblStack，相对地址 0 入栈 offsetStack。访问语法树结点 DId(id, T) 时执行的语义动作会把域名 **id** 的信息加入到记录类型的符号表中。在记录类型的域分析完后，offsetStack 栈顶保存记录类型中所有对象的总字宽。在图 7.16 中，exitTRec(D) 的动作会把所有域的字宽总和设置到记录类型的 width。把类型构造器 record 作用于该记录的符号表指针就相当于得到 T.type。

记录类型定义和过程声明的处理还是有区别的。处理记录类型时并未真正为哪个变量分配存储单元，只是决定了该类型的字宽和每个域在记录存储空间中的相对位置。在处理记录类型的变量声明时，才真正在活动记录中为记录类型的变量分配存储单元。

7.6 赋值语句

本节中表达式的类型可以是整型、实型和数组。为使翻译方案简洁，采用二义的表达式文法，并选择加运算作为二元运算的代表。作为赋值语句翻译成三地址指令的一部分工作，下面说明怎样从符号表中查找名字，怎样访问数组的元素和记录的域。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第290页 文件编号: 422624 页面编号: 32904538447 打印人: 8685 2026-03-09 10:23:03

7.6.1 符号表中的名字

在 7.1 节介绍中间语言时，为直观起见，让名字本身直接出现在三地址代码中，实际上应该把名字理解为它们在符号表中位置的指针。编译器在处理表达式、赋值语句等构造中的名字时，需要在符号表中查找它的定义，获得它的属性，然后在生成的三地址代码中使用它在符号表中位置的指针。

在图 7.17 所示的翻译方案中，lookup(**id**.lexeme) 用于根据名字 **id** 的字面量检查符号表中是否存在该名字的条目。若有，返回该条目的指针，否则返回 null，以表示没有找到。是否有过程嵌套，会影响 lookup 函数的设计，但不影响图 7.17 所示的翻译方案使用该函数来查符号表。

编号	事件处理方法	语义动作
（S1）	enterSAss(id, E)	p = lookup(**id**.lexeme); if (p! = null) emit (p, '=', E.place) else error;
（E1）	exitEBop(op, E_1, E_2)	x.place = newTemp(); emit (x.place, '=', E_1.place, op, E_2.place);
（E2）	exitEUop(op, E_1)	x.place = newTemp(); emit (x.place, '=', op, E_1.place);
（E3）	exitEPara(E_1)	x.place = E_1.place;
（E4）	exitEId(id)	p = lookup(**id**.lexeme); if (p! = null) x.place=p else error;

图 7.17 为赋值语句产生三地址代码的翻译方案

表达式 E 的属性 place 用来记住符号表条目的地址。函数 newTemp 用来产生一个新的临时变量的名字，把该名字也存入符号表，并返回该条目的地址。过程 emit 将其参数写到输出文件上，emit 的参数构成一条三地址指令。

如果碰到像 p.info 这样对记录域的访问，如何查表找到所需的属性呢？首先，从过程的名字表中可以找到 p，它应该是记录类型的变量。根据 7.4 节可以知道，p 的类型是 record(tblptr)，其中 tblptr 是该记录类型的符号表，从该表中可以找到 info 的条目，从而可以得到所需的属性。

7.6.2 数组元素的地址计算

一个数组的所有元素通常按一定的次序存于连续的存储块中，这样可以迅速访问这些元素。一维数组的元素一般是顺序存放的，如果每个数组元素的宽度是 w，那么一维数组 A 的第 i 个元素从地址

$$\text{base} + (i - \text{low}) \times w \tag{7.3}$$

开始，其中 low 是下标的下界，base 是分配给该数组的地址（可能是活动记录中的相对地址），即 base 是 A[low] 的地址。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第291页 文件编号: 422624 页面编号: 32914538445 打印人: 8685 2026-03-09 10:23:03

如果把表达式 (7.3) 重写成

$$i \times w + (\text{base} - \text{low} \times w)$$

那么可以在编译时完成该表达式后一部分的计算，从而减少了运行时的计算。

二维数组通常用两种形式之一存储：行为主（一行接一行）或列为主（一列接一列）。Fortran 系列语言使用列为主形式，C、Java 和 Pascal 都使用行为主形式。对于一个 2×3 的数组 A，若是行为主，其元素的存放次序是

$$A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]$$

若是列为主，其元素的存放次序是

$$A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3]$$

在行为主的情况下，所以 $A[i,j]$ 等价于 $A[i][j]$，所以可以说是各个一维数组 $A[i]$ 依次连续存放。

在行为主的二维数组情况下，$A[i_1, i_2]$ 的地址可以由公式

$$\text{base} + ((i_1 - \text{low}_1) \times n_2 + (i_2 - \text{low}_2)) \times w$$

计算，其中 low_1 和 low_2 分别是这两维的下界，n_2 是第二维的大小。也就是说，如果 high_2 是 i_2 的上界，那么 $n_2 = \text{high}_2 - \text{low}_2 + 1$。假定 i_1 和 i_2 都是编译时不能确定的值，可以把上面的表达式重写成

$$((i_1 \times n_2) + i_2) \times w + (\text{base} - ((\text{low}_1 \times n_2) + \text{low}_2) \times w) \tag{7.4}$$

同样，该表达式的后一项可以在编译时计算。

可以推广行为主或列为主的形式到多维数组。行为主形式的存储可以这样理解，当朝高地址扫描存储块时，最右边的下标变化最快，就像里程计上的数字。表达式 (7.4) 的推广使得 $A[i_1, i_2, \cdots, i_k]$ 的地址表达式为

$$\begin{aligned} &((\cdots((i_1 \times n_2 + i_2) \times n_3 + i_3)\cdots) \times n_k + i_k) \times w \\ +&\text{base} - ((\cdots((\text{low}_1 \times n_2 + \text{low}_2) \times n_3 + \text{low}_3)\cdots) \times n_k + \text{low}_k) \times w \end{aligned} \tag{7.5}$$

因为对所有的 j，$n_j = \text{high}_j - \text{low}_j + 1$ 是固定的，表达式 (7.5) 第二行的项可以在编译器分析数组声明时计算，存于符号表的 A 条目中。列为主的形式的布局相反，最左边的下标变化最快。

某些语言允许数组的大小在过程调用时动态确定，这种数组在运行栈上的分配在 6.2 节已经讨论了，其元素的访问公式和固定大小的数组相同，但由于上下界在编译时不知道，因此地址计算全部在运行时完成。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第292页 文件编号: 422624 页面编号: 32924538443 打印人: 8685 2026-03-09 10:23:03

7.6.3 数组元素地址计算的翻译方案

根据式 (7.5) 可知，k 维数组引用 $A[i_1, i_2, \cdots, i_k]$ 的三地址代码主要是完成

$$(\cdots((i_1 \times n_2 + i_2) \times n_3 + i_3)\cdots) \times n_k + i_k \tag{7.6}$$

的计算，然后乘以 w，再加上式 (7.5) 第二行的值。式 (7.6) 的值可以通过递推计算

$$\begin{aligned} e_1 &= i_1 \\ e_m &= e_{m-1} \times n_m + i_m \end{aligned} \tag{7.7}$$

来完成，一直到 $m = k$ 为止。

根据式 (7.7) 的递推计算，除了第一个下标表达式 i_1，对其他每个下标表达式，需要产生乘和加两条三地址指令（没有包括下标表达式本身值计算的三地址指令），因此在处理下标表达式时需要访问符号表中 A 的条目，以得到 A 各维的大小。

如果使用下列产生式

$$L \to \textbf{id}\,[\text{Elist}] \mid \text{id}$$

$$\text{Elist} \to \text{Elist},\, E \mid E$$

的非终结符 L 取代图 7.17 中的 **id**，那么数组引用可以出现在赋值语句中。如果边解析边翻译，在仅用综合属性的情况下处理 Elist $\to E$ 和 Elist $\to$ Elist, E 时，访问不到符号表中 A 的条目，因为这是数组 **id** 的属性，产生式需要重写为

$$L \to [\text{Elist}] \mid \textbf{id}$$

$$\text{Elist} \to \text{Elist},\, E \mid \textbf{id}\,[E]$$

即数组名和最左边一个下标表达式连在一起。这个文法很不直观。但是，如果基于语法树来翻译就简单多了，无需修改文法，翻译方案如图 7.18 所示，其中的编号对应的产生式见表 7.3。

假设 ndim 记录已分析过的下标表达式的个数，函数 limit(A, j) 返回数组 A 中第 j 维的大小 n_j，函数 invariant(A) 从 A 所指的符号表条目中取静态可计算的值，即式 (7.5) 第二行的值，函数 width(A) 从 A 所指的符号表条目中取数组元素的宽度。

左值表达式 L 有两个属性，place 和 offset。当 L.offset 等于 null 时，表示该左值是一个简单名字，此时 L.place 是该名字的符号表条目指针（直接用 **id**.place 表示）。当 L 是数组引用时，这两个属性的用途见下面的解释。与图 7.17 一样，算术表达式 E 有 place 属性，并且含义一样。

与 7.6.1 节的表达式类似，三地址代码通过调用过程 *emit* 产生。下面给出为支持数组而需要对中间代码生成的翻译方案调整或扩展的部分。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第293页 文件编号: 422624 页面编号: 32934538441 打印人: 8685 2026-03-09 10:23:03

（1）对于赋值语句 $S \to L := E$ 对应的语法树结点 SAssL(L, E)，如果 L 是简单名字，则产生正常的赋值；否则产生对由 L 的两个属性确定的存储单元的索引赋值。

（2）对于表达式 $E \to L$ 对应的语法树结点 ELval(L)，若 L 的 offset 为空，表示是简单名字，则右值从 L.place 获取。否则 L 是数组元素，右值可用索引得到存储单元 L.place[L.offset] 的内容，见图 7.18 中的 exitELval(L)。L.place 和 L.offset 都是新的临时变量，前者对应式 (7.5) 的第二项，在 exitELval(L) 中创建；后者保存 w 乘以 n 维下标的线性索引，对应式 (7.5) 的第一项，在 visitELArr(id, $E_1, \cdots, E_n$) 中计算。

编号	事件处理方法	语义动作
（S1'）	exitSAssL(L, E):	if (L.offset == null) emit (L.place, '=', E.place); else emit (L.place, '[', L.offset, ']', '=', E.place);

（E4'）	exitELval(L):	if (L.offset == null) x.place =L.place;
		else {x.place = newTemp();
		emit (x.place,'=', L.place, '[', L.offset, ']');}

（E6）	visitELArr(id, $E_1, \cdots, E_n$):	访问 E_1 之后：
		ndim = 1; place = E_1.place;
		每次访问 E_i 之后计算:
		t = newTemp(); ndim ++;
		emit (t, '=', place, '*', limit(**id**.place, ndim));
		emit (t, '=', t, '+', E_i.place); place = t;
		结尾：x.place = newTemp();
		emit (x.place, '=', base(**id**.place), '−', invariant (**id**.place));
		x.offset = newTemp();
		emit (x.offset, '=', place, '*', width(**id**.place)));

（E7）	exitELId(id):	x.place = **id**.place; x.offset=null; 注：x 表示正在处理的语法树结点实例

图 7.18　为含数组引用的表达式产生三地址代码

（3）对于表达式 $L \to$ **id** 对应的语法树结点 ELId(id)，会获取简单名字的存储位置 **id**.place，并将 **offset** 属性设置为 null。

【例 7.5】　设 A 是 10×20 的数组，$n_1 = 10$ 且 $n_2 = 20$，取 $w = 4$。赋值语句 $x := A[y, z]$ 的注释分析树如图 7.19 所示。该赋值语句翻译成下列三地址指令序列：

$$
\begin{aligned}
t_1 &= y \times 20 \\
t_1 &= t_1 + z \\
t_2 &= c \qquad\qquad /^* \text{ 常量 } c = \text{base}_A - 84) \ ^*/ \\
t_3 &= t_1 \times 4
\end{aligned}
$$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第294页 文件编号: 422624 页面编号: 32944538449 打印人: 8685 2026-03-09 10:23:03

$$t_4 = t_2[t_3]$$

$$x = t_4$$

对每个变量，**id**.place 已经由它的名字代替了。

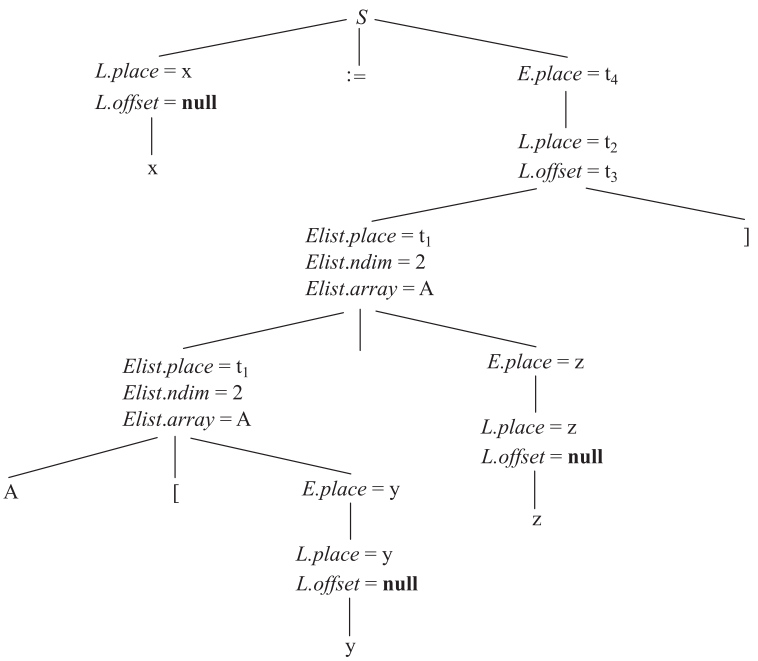


图 7.19 $x := A[y, z]$ 的注释分析树

7.6.4 类型转换

到目前为止的中间代码生成，都忽略了可能有的数据对象的类型转换操作。考虑上述赋值语句的文法，假定有整数和实数两种类型，必要时整数可转换为实数。这里直接使用在 5.3 节已经熟悉的属性 E.type，它的值是 real 或 integer，并且还忽略类型错误的检查。

$E \to E_1 + E_2$ 对应的语法树结点 EBop(op,E_1,E_2) 和大多数其他产生式的语义动作都可以修改成必要时产生 x = inttoreal y 的三地址指令，它的作用是把整数 y 转换成值相等的实数，再赋给 x，还必须给运算符一个标记，用以标明这是定点还是浮点算术运算。产生式 $E \to E_1 + E_2$ 的完整语义动作列在图 7.20 中。

例如，假定 x 和 y 的类型是 real，i 和 j 的类型是 integer，对于输入

$$x = y + i * j$$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第295页 文件编号: 422624 页面编号: 32954538446 打印人: 8685 2026-03-09 10:23:03

根据图 7.20，输出的三地址指令序列是

$$
\begin{aligned}
t_1 &= i \ \text{int*}\ j \\
t_2 &= \text{inttoreal}\ t_1 \\
t_3 &= y \ \text{real+}\ t_2 \\
x &= t_3
\end{aligned}
$$

图 7.20 所示的语义动作为非终结符 E 使用了两个属性 E.place 和 E.type。

编号	事件处理方法	语义动作

（E1）	exitEBop('+ ', E_1, E_2):	x.place = newTemp();
		if (E_1.type == integer && E_2.type == integer) {
		emit (x.place, '=', E_1.place, 'int+', E_2.place); x.type = integer;
		} else if (E_1.type == integer && E_2.type == real) {
		u = newTemp( ); emit (u, '=', 'inttoreal', E_1.place);
		emit(x.place, '=', u, 'real+', E_2.place); x.type = real;
		}

图 7.20 $E \to E_1 + E_2$ 的语义动作

7.7 布尔表达式和控制流语句

在编程语言中，布尔表达式有两个基本目的：第一个是用于计算逻辑值；第二个更经常的是作为条件表达式，用于控制流语句，如 if-then、if-then-else 和 while-do 语句。因此本节将布尔表达式和控制流语句放在一起进行讨论。

7.7.1 布尔表达式

布尔表达式也可以像算术表达式那样归纳地定义。下面是本节所用的布尔表达式的文法和对应的语法树结点，即表 7.3 中的 (B1)∼(B7)，其中 **relop** 是关系运算符。

(B1)	$B \to B_1$ **or** B_2	LOr(B_1, B_2)
(B2)	$\mid B_1$ **and** B_2	LAnd(B_1, B_2)
(B3)	$\mid$ **not** B_1	LNot(B_1)
(B4)	$\mid (B_1)$	LPara(B_1)
(B5)	$\mid E_1$ **relop** E_2	LRel(relop, E_1, E_2)
(B6)	$\mid$ **true**	LTrue
(B7)	$\mid$ **false**	LFalse

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第296页 文件编号: 422624 页面编号: 32964538444 打印人: 8685 2026-03-09 10:23:03

按惯例，**or** 和 **and** 都是左结合的，**or** 的优先级最低，然后是 **and**，最后是 **not**。

在有些情况下，只要完成了布尔表达式的部分计算就可以知道它的真假，那么它剩余部分是否还要计算？这由编程语言的语义决定。C 和 Java 采用部分计算，称为短路计算。短路计算有严格的语义规定，即把 B_1 **or** B_2 定义为

if B_1 **then true else** B_2

把 B_1 **and** B_2 定义为

if B_1 **then** B_2 **else false**

如果 B_1 或 B_2 是有副作用的表达式（例如，含有修改某个全局变量的函数调用），则完全计算和短路计算的结果可能是有区别的。

布尔表达式的值可以通过两种主要方式来表示。第一种方式是把真和假数值化，使布尔表达式的计算类似于算术表达式的计算，常常用 1 表示真，用 0 表示假。当然，还有其他编码方式，例如，用非 0 表示真，用 0 表示假；或者用非负数表示真，用负数表示假。显然，在这种情况下，布尔表达式和算术表达式的中间代码生成没有多少区别。

实现布尔表达式的第二种方式是借助控制流，即用程序中的位置来表示布尔表达式的值，这适用于短路计算的情况，用这种方式实现控制流语句中的布尔表达式尤其方便。因为对于控制流语句来说，只要能根据布尔表达式的结果跳转到正确的位置继续执行就可以了，对它的具体值无须再关心。7.7.3 小节介绍布尔表达式在这种方式下的中间代码生成。

7.7.2 控制流语句的翻译

现在考虑 if-then、if-then-else、while-do 和顺序语句到三地址指令的翻译，这些语句由下面的文法产生：

(S2)	S	$\rightarrow$ **if** B **then** S_1	$\mathrm{SIft}(B, S_1)$
(S3)		| **if** B **then** S_1 **else** S_2	$\mathrm{SIfte}(B, S_1, S_2)$
(S4)		| **while** B **do** S_1	$\mathrm{SWhile}(B, S_1)$
(S5)		| $S_1; S_2$	Stmts

在这些产生式中，B 是布尔表达式。图 7.21 用图形表示出这 4 条语句的三地址代码的结构。图中 B.code 和 S.code 分别表示 B 和 S 的三地址代码，它们在图中的次序就表示了它们在整个代码中的次序。例如，在 if-then 的结构图中，B.code 在 S.code 的上面，它表示 B 的三地址代码在 S 的三地址代码前面。

图 7.21 中的 B.true、B.false、S.begin 和 S.next 都是三地址指令的标号，它们都是继承属性。图 7.21(a) 为 if-then 的结构图，由于 B.code 究竟有多少条指令需等 B 翻译结束才知道，因此在翻译 B 的过程中难以知道 B.true 在三地址指令序列中的准确位置，为便于翻译，采用给三地址指令加标号的方式，函数 newLabel 每次调用时返回一个新的标号。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第297页 文件编号: 422624 页面编号: 32974538442 打印人: 8685 2026-03-09 10:23:03

对布尔表达式 B，用两个标号 B.true 和 B.false 分别表示 B 为真和为假时控制流应该转向的标号，这两个属性由 B 的上下文决定。S.begin 是 S 的第一条三地址指令的标号，S.next 表示执行完 S 后应该执行的第一条三地址指令的标号，它们都由 S 的上下文决定。对于 $S_1;S_2$ 的情况，S_1.next 显然应该定位在 S_2 的第一条三地址指令处，如图 7.21(d) 所示。对于其他语句，情况就不这么简单了，见图 7.21(b) 所示的 if-then-else 语句。在 S_1.code 的后面有 goto S.next，这条指令是需要的，因为 S_1 执行结束意味着 S 执行结束，并且当 S_1 是赋值语句时肯定会执行这条指令。问题是 S_1.next 应该定位在什么地方，它可以定位在指令 goto S.next 处，但这意味着如果 S_1 中有三地址指令跳转到 S_1.next，那么紧接着再执行 goto S.next。这种连续跳转显然应该避免并且可以避免，具体做法是暂不定位 S_1.next，让它等于 S.next。出于同样的原因，S.next 不一定定位在 S_2 的后面并且紧挨着 S_2 的地方。

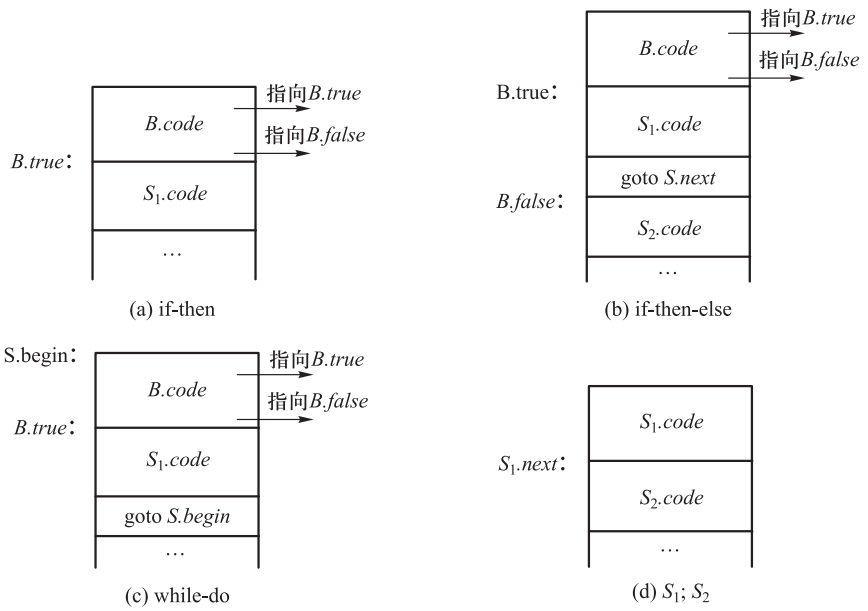


图 7.21 if-then、if-then-else、while-do 和顺序语句的代码

采用标号技术的控制流语句的翻译方案，在边解析边自下而上的翻译过程中，由于需要处理继承属性，往往需要对文法进行改写，因此实现起来较为复杂；然而，如果采用基于树访问的方式进行翻译，方案实现会相对简单。

图 7.22 是基于树访问的翻译方案。和赋值语句的翻译方案不一样的是，用函数 gen 代替了过程 emit，gen 把形成的代码串作为函数值，而不是输出到文件上。还有，|| 是作为串的连接算符。布尔表达式的翻译方案在下一小节另行给出。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第298页 文件编号: 422624 页面编号: 32984538440 打印人: 8685 2026-03-09 10:23:03

编号	事件处理方法	语义动作
（S2）	visitIft(B, S_1):	访问 B 前： B.true = newLabel(); B.false = x.next; 进入 S_1 前: S_1.next = x.next; x.code = B.code || gen(B.true, ':'); 访问 S_1 后： x.code = x.code || S_1.code;
（S3）	visitIfte(B, S_1, S_2):	访问 B 前： B.true = newLabel(); B.false = x.next; 进入 S_1 前: S_1.next = x.next; x.code = B.code || gen(B.true, ':'); 进入 S_2 前: S_2.next = x.next; x.code = x.code || S_1.code || gen('goto', x.next); 访问 S_2 后： x.code = x.code || gen(B.false, ':') || S_2.code;
（S4）	visitWhile(B, S_1):	访问 while 前： x.begin = newLabel(); 访问 B 前： B.true = newLabel(); B.false = x.next; x.code = gen(S.begin, ':'); 进入 S_1 前: S_1.next = x.begin; x.code = x.code || B.code || gen(B.true, ':'); 访问 S_1 后： x.code = x.code || S_1.code || gen('goto', x.begin);

图 7.22 基于树访问的控制流语句的翻译

7.7.3 布尔表达式的控制流翻译

现在讨论上一小节的 B.code，即为布尔表达式 B 产生中间代码，把 B 翻译成一串条件转移和无条件转移的三地址指令序列。

设计这个翻译的基本想法是，假定 B 是 $a < b$ 的形式，那么生成的代码形式为

if $a < b$ **goto** B.true
goto B.false

设 B 是 B_1 **or** B_2 的形式。如果 B_1 为真，那么可以知道 B 本身为真，即 B_1.true 和 B.true 一样。如果 B_1 为假，那么 B_2 必须计算，可以把 B_1.false 定位在 B_2 代码第一条指令处。B_2 的 true 和 false 分别与 B 的 true 和 false 一样。

类似的考虑可用于 B_1 **and** B_2 的翻译。形式为 **not** B_1 的表达式无须产生新的代码，只要交换 B 的 true 和 false 就得到 B_1 的 true 和 false。按这种方法为布尔表达式生成三地址

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第299页 文件编号: 422624 页面编号: 32994538448 打印人: 8685 2026-03-09 10:23:03

代码的树访问翻译方案如图 7.23 所示。

编号	事件处理方法	语义动作
（B1）	visitLOr(B_1, B_2):	访问 B_1 前： $B1$.true = x.true; B_1.false = newLabel(); 进入 B_2 前: B_2.true = x.true; B_2.false = x.false; x.code = B_1.code || gen(B_1.false, ':'); 访问 B_2 后： x.code = x.code || B_2.code;
（B2）	visitLAnd(B_1, B_2):	访问 B_1 前： $B1$.true = newLabel(); B_1.false = x.false; 进入 B_2 前: B_2.true = x.true; B_2.false = x.false; x.code = B_1.code || gen(B_1.true, ':'); 访问 B_2 后： x.code = x.code || B_2.code;
（B3）	visitLNot(B_1):	访问 not 前： B_1.true = x.false; B_1.false = x.true; 访问 B_1 后： x.code = B_1.code;
（B4）	exitLRel(relop, E_1, E_2):	x.code = E_1.code || E_2.code || gen('if ', E_1.place, relop.op, E_2.place, 'goto', x.true) || gen('goto', x.false);
（B5）	visitLPara(B_1):	访问（前： B_1.true = x.true; B_1.false = x.false; 访问 B_1 后： x.code = B_1.code;
（B6）	exitLTrue:	x.code = gen('goto', x.true);
（B7）	exitLFalse:	x.code = gen('goto', x.false);

图 7.23 基于树访问的布尔表达式的翻译

【例 7.6】 考虑布尔表达式

$$a < b \text{ or } c < d \text{ and } e < f$$

假定整个表达式为真和为假的出口分别置为 L_{true} 和 L_{false}。那么用图 7.23 所示的定义可以得到下列代码：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第300页 文件编号: 422624 页面编号: 33004538444 打印人: 8685 2026-03-09 10:23:03

$$
\begin{aligned}
&\textbf{if } a < b \text{ goto } L_{\text{true}}\\
&\text{goto } L_1\\
L_1:\quad &\textbf{if } c < d \text{ goto } L_2\\
&\text{goto } L_{\text{false}}\\
L_2:\quad &\textbf{if } e < f \text{ goto } L_{\text{true}}\\
&\text{goto } L_{\text{false}}
\end{aligned}
$$

生成的代码还不是最优的，因为删掉第二条指令不会改变代码的值。这种形式的冗余指令不难在以后的处理中删除。避免产生这些冗余转移的另一个办法是把形式为 $E_1 < E_2$ 的关系表达式翻译成 **if** $E_1 \geqslant E_2$ goto B.false，即条件为真时执行紧跟代码，而条件为假时跳转，简称为假转方式。

【例 7.7】 考虑语句

```
while a < b do
    if c < d then
        x := y + z
    else
        x := y - z
```

将图 7.23 所示的翻译方案、赋值语句的翻译方案（见图 7.17、图 7.18）和控制流语句的翻译方案（见图 7.22）合在一起，为该语句产生下列代码：

$$
\begin{aligned}
L_1:\quad &\textbf{if } a < b \text{ goto } L_2\\
&\text{goto } L_{\text{next}}\\
L_2:\quad &\textbf{if } c < d \text{ goto } L_3\\
&\text{goto } L_4\\
L_3:\quad &t_1 = y + z\\
&x = t_1\\
&\text{goto } L_1\\
L_4:\quad &t_2 = y - z\\
&x = t_2\\
&\text{goto } L_1
\end{aligned}
$$

改变条件测试的方向可以删除前两个 `goto` 语句。

7.7.4 开关语句的翻译

在许多语言中都有“开关”语句或“分情况”语句，这里讨论的开关语句如下：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第301页 文件编号: 422624 页面编号: 33014538442 打印人: 8685 2026-03-09 10:23:03

switch E
  begin
    case $V_1 : S_1$
    case $V_2 : S_2$
    . . .
    case $V_{n-1} : S_{n-1}$
    default: S_n
  end

它的执行流程如下：

（1）计算开关表达式 E 的值。

（2）分支测试，即在 $n-1$ 个常量 $V_1, V_2, \cdots, V_{n-1}$ 中寻找和 E 值相同的常量。若存在，则认为该常量和 E 匹配，否则认为默认值 `default` 和 E 匹配。

（3）执行相匹配常量后的分支的语句。

（4）执行了这三步后就代表开关语句执行结束。

步骤（2）的分支测试有多种实现方法。如果分支数不是很多，如少于 10 时，那么把开关语句翻译成下面的条件转移序列是合理的：

```
        t = E的代码
        if t != V1 goto L1
        S1的代码
        goto next
L1:     if t != V2 goto L2
        S2的代码
        goto next
L2:     ...
        ...
Ln-2:   if t != Vn-1 goto Ln-1
        Sn-1 的代码
        goto next
Ln-1:   Sn 的代码
next:
```

该方式的缺点是，由于分支测试的代码散在各处，很难对它们进行专门的优化处理。为了提高效率，程序员应该把经常出现的分支放在前面。

另一种翻译方式是将分支测试的代码集中在一起，放在该语句代码的后部。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第302页 文件编号: 422624 页面编号: 33024538440 打印人: 8685 2026-03-09 10:23:03

$t = E$的代码
goto test
L_1 : S_1的代码
goto next
L_2 : S_2的代码
goto next
...
L_{n-1} : S_{n-1}的代码
goto next
L_n : S_n的代码
goto next
test : **if** $t == V_1$ **goto** L_1
if $t == V_2$ **goto** L_2
...
if $t == V_{n-1}$ **goto** L_{n-1}
goto L_n
next :

把分支测试序列的代码放在该语句代码的前部对一遍扫描的编译器来说是不方便的，因为它不可能在扫描每个 S_i 前就产生这样的测试代码，而扫描每个 S_i 时又必须产生 S_i 的代码。

为了便于识别从标号 test 开始的测试是开关语句的分支测试序列，以便代码生成器对它进行特别处理，可以给中间代码增加一种 **case** 指令，将分支测试序列的中间代码改成下面的形式：

test : **case** V_1 L_1
case V_2 L_2
...
case V_{n-1} L_{n-1}
case t L_n
next :

其中， **case** V L 和 **if** $t == V$ **goto** L 的含义一样。

代码生成器实现这个测试序列的一种紧凑办法是建立一张常量值和标号的二元组表。通过一个循环，将开关表达式的值和表中的各个值相比较，如果没有其他的值可匹配，最后的默认条目肯定匹配。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第303页 文件编号: 422624 页面编号: 33034538448 打印人: 8685 2026-03-09 10:23:03

如果常量值的数目较多，那么用散列表效率会更高一些。

对一种经常出现的特殊情况，可以用更有效的办法实现 n 个分支。如果所有的常量值落在一个较小的区间，例如在 $i_{\min}$ 和 $i_{\max}$ 之间，并且 $n/(i_{\max}-i_{\min})$ 的值较大，那么可以构造标号表，让与常量 j 对应的标号放在偏移是 $j-i_{\min}$ 的条目中，空白的条目全填上默认语句的标号。执行该语句时，先计算开关表达式，获得它的值 m，检查 m 是否在 $i_{\min}$ 和 $i_{\max}$ 之间，若不在其中，则直接执行默认语句；若在其中，则跳转到偏移为 $m-i_{\min}$ 的条目所指地址去执行。

为把开关语句翻译成上面形式的代码，编译器看见保留字 **switch** 时，产生两个新的标号 test 和 next 以及新的临时变量 t，在分析表达式 E 时，产生计算 E 到 t 的代码，并产生跳转语句 **goto** test。然后为每个 **case** $V_i:S_i$ 产生新建的标号 L_i，后面跟着 S_i 的代码，再后面是跳转 **goto** next 指令。与此同时，将标号 L_i 加入符号表，并将这个条目的指针和常量 V_i 放入专用于存储 V_i 和 L_i 对应关系的队列中。当看见终止开关体的保留字 **end** 时，用这个队列中的信息产生分支测试的指令序列。

7.7.5 过程调用的翻译

第 6 章已详细介绍了过程调用的实现，包括存储空间的组织、过程调用序列和返回序列等，本小节使用下面的文法和语法树节点：

$$\text{(S6)} \quad S \rightarrow \textbf{call id}(\text{Elist}) \quad \text{SCall}(\textbf{id}, \text{Elist})$$

其中，Elist 表示由若干实在参数值形成的参数列表。

这里简单介绍过程调用语句的中间代码生成。过程调用的不同实现方式有不同的过程调用序列和返回序列，为了让中间表示能方便地用于不同的实现方式，特别为它增设一种 **param** 指令，专用于指示实在参数，就像开关语句的中间代码 **case** 指示分支测试一样。过程调用 $\textbf{id}(E_1, E_2, \cdots, E_n)$ 的中间代码结构如下：

E_1.place = E_1的代码
E_2.place = E_2的代码
...
E_n.place = E_n的代码
param E_1.place
param E_2.place
...
param E_n.place
call id.place, n

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第304页 文件编号: 422624 页面编号: 33044538446 打印人: 8685 2026-03-09 10:23:03

由上面的代码结构可知，在生成 $E_i.\text{place} = E_i$ 的代码后，需要保存 $E_i.\text{place}$ 的值，这些最终可以以队列的方式存储在 Elist 中。下面是采用这种数据结构的翻译方案：

编号	事件处理方法	语义动作
（S6）	exitSCall(id, Elist):	foreach E in Elist emit('**param**', E.place); emit('**call**', **id**.place, n);

对于返回语句，中间代码的生成相对简单直接。

习 题

7.1 将算术表达式 $(a+b)\times(c+d)+(a+b+c)$ 翻译成以下形式：

（a）语法树。

（b）有向无环图。

（c）后缀表示。

（d）三地址代码。

7.2 将 C 语言程序

```
1  void main () {
2      int i;
3      int a[10];
4      while (i <= 10 )
5          a[i] = 0;
6  }
```

中的可执行语句翻译成以下形式：

（a）语法树。

（b）后缀表示。

（c）三地址代码。

***7.3** 证明：如果所有运算符都是二元的，那么由运算符和运算对象组成的串是后缀表达式，当且仅当

（1）运算符个数正好比运算对象个数少一个；

（2）在这个表达式的每个非空前缀中，运算符数少于运算对象数。

7.4 修改图 7.13 中计算声明名字的类型和相对地址的翻译方案，以允许名字表而不是单个名字出现在形式为 $D \to \textbf{id} : T$ 的声明中。

7.5 修改图 7.13 中计算声明名字的类型和相对地址的翻译方案，offset 不是全局变量，而是文法符号的继承属性。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第305页 文件编号: 422624 页面编号: 33054538443 打印人: 8685 2026-03-09 10:23:03

7.6 算符 θ 作用于表达式 $e_1, e_2, \cdots, e_k$ 的前缀形式是 $\theta\ p_1\ p_2\ \cdots\ p_k$，其中 p_i 是 e_i 的前缀形式。

（a）写出 a * - (b + c) 的前缀形式。

*（b）证明：所有的语义动作都是打印，并且所有的动作都出现在产生式右部末端的翻译方案不可能把中缀表达式翻译成前缀表达式。

（c）给出把中缀表达式翻译成前缀形式的语法制导定义。

（d）给出把中缀表达式翻译成前缀形式的翻译方案。

7.7 编写一个程序，实现图 7.23 所示的翻译布尔表达式到三地址代码的语法制导定义。

7.8 图 7.23 所示的翻译方案把 $B \leftarrow E_1 < E_2$ 的控制转移部分翻译成两条指令

```
if E1.place < E2.place goto ...
goto ...
```

这些指令可以用一条指令

```
if E1.place >= E2.place goto ...
```

来代替，当 B 为真时执行后继代码。修改图 7.23 所示的翻译方案，使之产生这种性质的代码。

***7.9** 下面是一个 C 语言程序：

```
1  int main() {
2      int i, j;
3      while ( (i || j) && (j > 5) ) {
4          i = j;
5      }
6  }
```

如下是在不同平台上编译生成的汇编代码（编译器版本见汇编代码最后一行），请在其中有关指令后加注释，将源程序中的操作和生成的汇编代码对应起来，以判断确实是用短路计算方式来完成布尔表达式计算的。

（1）在 x86/Linux 32 位系统上编译生成的汇编代码如下：

```
1   .file   "ex7-9.c"
2   .version    "01.01"
3   gcc2_compiled.:
4   .text
5   .align 4
6   .globl main
7   .type   main,@function
8   main:
9       pushl   %ebp
10      movl    %esp,%ebp
11      subl    $8,%esp
12      nop
13      .p2align 4,,7
14  .L2:
15      cmpl    $0,-4(%ebp)
16      jne .L6
17      cmpl    $0,-8(%ebp)
18      jne .L6
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第306页 文件编号: 422624 页面编号: 33064538441 打印人: 8685 2026-03-09 10:23:03


```
    jmp .L5
    .p2align 4,,7
.L6:
    cmpl    $5,-8(%ebp)
    jg .L4
    jmp .L5
    .p2align 4,,7
.L5:
    jmp .L3
    .p2align 4,,7
.L4:
    movl    -8(%ebp),%eax
    movl    %eax,-4(%ebp)
    jmp .L2
    .p2align 4,,7
.L3:
.L1:
    leave
    ret
.Lfe1:
    .size   main,.Lfe1-main
    .ident  "GCC: (GNU) egcs
        -2.91.66 19990314/Linux (
        egcs-1.1.2 release)"
```

（2）在 x86/Linux 64 位系统上编译生成的汇编代码如下：

```
    .file   "ex7-9.c"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    pushq   %rbp
    movq    %rsp, %rbp
    jmp     .L2
.L5:
    movl    -4(%rbp), %eax
    movl    %eax, -8(%rbp)
.L2:
    cmpl    $0, -8(%rbp)
    jne     .L3
    cmpl    $0, -4(%rbp)
    je      .L4
.L3:
    cmpl    $5, -4(%rbp)
    jg      .L5
.L4:
    movl    $0, %eax
    popq    %rbp
    ret
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 7.5.0-3
        ubuntu1~16.04) 7.5.0"
```

（3）在 Linux ARM64 系统上编译生成的汇编代码如下：

```
  .arch armv8-a
  .file "ex7-9.c"
  .text
  .align  2
  .global main
  .type main, %function
main:
.LFB0:
  sub sp, sp, #16
  b .L2
.L5:
  ldr w0, [sp, 12]
  str w0, [sp, 8]
.L2:
  ldr w0, [sp, 8]
  cmp w0, 0
  bne .L3
  ldr w0, [sp, 12]
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第307页 文件编号: 422624 页面编号: 33074538449 打印人: 8685 2026-03-09 10:23:03


```
  cmp w0, 0
  beq .L4
.L3:
  ldr w0, [sp, 12]
  cmp w0, 5
  bgt .L5
.L4:
  mov w0, 0
  add sp, sp, 16
  ret
.LFE0:
  .size main, .-main
  .ident  "GCC: (Ubuntu 11.4.0-1
        ubuntu1~22.04) 11.4.0"
  .section  .note.GNU-stack,"",
        @progbits
```

（4）在 LoongArch 系统上编译生成的汇编代码如下：

```
  .file "ex7-9.c"
  .text
  .align  2
  .globl  main
  .type main, @function
main:
.LFB0 = .
  addi.d  $r3,$r3,-32
  st.d  $r22,$r3,24
  addi.d  $r22,$r3,32
  b .L2
.L5:
  ld.w  $r12,$r22,-24
  st.w  $r12,$r22,-20
.L2:
  ld.w  $r12,$r22,-20
  bnez  $r12,.L3
  ld.w  $r12,$r22,-24
  beqz  $r12,.L4
.L3:
  ld.w  $r12,$r22,-24
  or  $r13,$r12,$r0
  addi.w  $r12,$r0,5      # 0x5
  bgt $r13,$r12,.L5
.L4:
  or  $r12,$r0,$r0
  or  $r4,$r12,$r0
  ld.d  $r22,$r3,24
  addi.d  $r3,$r3,32
  jr  $r1
.LFE0:
  .size main, .-main
  .ident  "GCC: (LoongArch GNU
        toolchain rc1.5
        (20240802)) 8.3.0"
  .section  .note.GNU-stack,"",
        @progbits
```

7.10 下面是一个 C 语言程序：

```
main() {
    long i, j;
    if ( j )
        i++;
    else
        while ( i ) j++;
}
```

在 x86/Linux 系统上编译（版本较低的 GCC 编译器，并且未使用优化）该程序得到的汇编代码（为便于理解，略去了和讨论本问题无关的部分，并改动了一个地方）如下：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第308页 文件编号: 422624 页面编号: 33084538447 打印人: 8685 2026-03-09 10:23:03


```
1  main:
2      pushl   %ebp
3      movl    %esp,%ebp
4      subl    $8,%esp
5      cmpl    $0,-8(%ebp)
6      je .L2
7      incl    -4(%ebp)
8      jmp .L3
9  .L2:
10 .L4:
11     cmpl    $0,-4(%ebp)
12     jne .L6
13     jmp .L5
14 .L6:
15     incl    -8(%ebp)
16     jmp .L4
17 .L5:
18 .L3:
19 .L1:
20     leave
21     ret
```

(a) 为什么会出现一条指令前有多个标号的情况，如.L2 和.L4，还有.L5、.L3 和.L1？从控制流语句的中间代码结构加以解释。

(b) 每个函数都有.L1 这样的标号，它可能的作用是什么，为什么本函数没有引用该标号的地方？（习题*7.9 的汇编代码也有该现象。）

7.11 编写一个程序，实现图 7.22 给出的控制流语句的翻译方案。

7.12 用 7.6 节的翻译方案，把赋值语句 A[x, y]:= z 翻译成三地址代码（其中 A 是 10×5 的数组）。

***7.13** C 语言的 for 语句有下列形式：

```
for (e1; e2; e3) stmt
```

它和

```
e1;
while (e2) do begin
    stmt;
    e3
end
```

有同样的含义。构造一个翻译方案，把 C 语言风格的 for 语句翻译成三地址代码。

7.14 Pascal 标准定义语句

```
for v := initial to final do stmt
```

和下面的代码序列

```
begin
    t1 := initial;
    t2 := final;
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第309页 文件编号: 422624 页面编号: 33094538445 打印人: 8685 2026-03-09 10:23:03


```
        if t1 <= t2 then begin
            v := t1;
            stmt;
            while v != t2 do begin
                v := succ(v);
                stmt
            end
        end
    end
```

有同样的含义。

(a) 考虑下面的 Pascal 语言程序：

```
program forloop (input, output);
    var i, initial, final: integer;
    begin
        read (initial, final);
        for i := initial to final do
            writeln (i)
    end.
```

当 initial = MAXINT - 5 并且 final = MAXINT 时，该程序的行为是什么？其中 MAXINT 是目标机器的最大整数。

*(b) 构造翻译方案，为 Pascal 的 for 语句产生正确的三地址代码。

7.15 语句

```
for i := 1 step 10-j until 10*j do j := j + 1
```

可以有 3 种不同的语义定义。第一种可能的含义是，步长表达式 10 - j 和终值表达式 10*j 都仅在循环前计算一次。这样，如果在循环前 j = 5，那么该循环体执行 10 次。第二种语义完全不同，要求每次通过循环体时，都要执行步长表达式和终值表达式。例如，若在循环前 j = 5，那么该循环将不会终止，C 语言的 for 语句属于这种情况，其表达式 e2 和 e3 是要重复计算的（见习题***7.13**）。第三种含义由 ALGOL 这样的语言给出。当步长是负数时，该循环终止的测试是 i < 10*j，而不是 i > 10*j。分别写出这 3 种定义下的中间代码结构。

7.16 赋值语句 A[x, y] := z（其中 A 是 10×5 的数组）的注释分析树如图 7.24 所示。请根据 7.6.3 小节的翻译方案，把图 7.24 中的属性值都补上（像图 7.19 那样），并且把每步归约产生的中间代码写在相应产生式的旁边。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第310页 文件编号: 422624 页面编号: 33104538443 打印人: 8685 2026-03-09 10:23:03

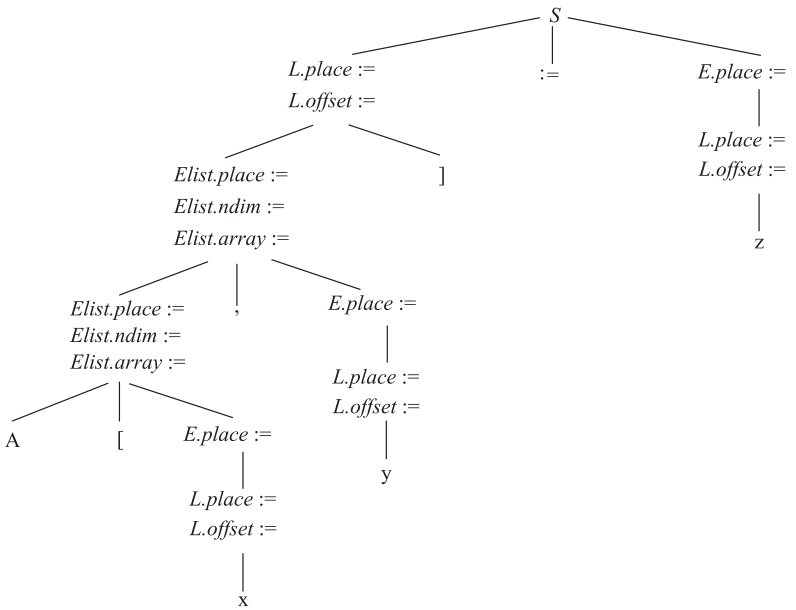


图 7.24 A[x, y]:= z 的注释分析树

7.17 C 语言和 Java 语言的数组声明和数组元素引用的语法形式与 7.6 节讨论的不一样，如 float A[10][20] 和 A[i+1][j-1]，并且每一维的下界都是 0。若适应这种情况的赋值语句的文法如下：

$$S \to L := E$$
$$E \to E + E \mid (E) \mid L$$
$$L \to L[E] \mid \mathbf{id}$$

（a）重新设计 7.6.2 小节数组元素的地址计算公式，以方便编译器产生数组元素地址计算的中间代码。不要忘记每一维的下界都是 0。

（b）重新设计数组元素地址计算的翻译方案。只需写出产生式 $L \to L[E] \mid \mathbf{id}$ 的翻译方案，但要能和 7.6.3 小节中产生式 $S \to L := E$ 和 $E \to E + E \mid (E) \mid L$ 的翻译方案衔接。若翻译方案中引入新的函数调用，要解释这些函数的含义。

7.18 （a）图 7.14 所示的翻译方案不能用于允许递归过程的情况（见 7.5.2 小节中的解释）。对图 7.14 所示的文法及其翻译方案做最少修改（不改变该文法定义的语言），使得修改后的翻译方案能够用于有直接递归调用的情况（可以修改或增加语义动作）。

（b）对于允许过程嵌套的语言，结合图 7.14 和图 7.17 的翻译方案，像图 2.18 和图 2.19 那样给出函数 lookup(name) 的描述。

第 8 章

代码生成

编译器的最后一个阶段的工作是代码生成，它取源程序的中间表示作为输入，产生等价的目标程序作为输出，如图 8.1 所示。不管代码生成阶段的前面是否有代码优化阶段，本章提出的代码生成技术都是适用的。

图 8.1 代码生成器的位置

对代码生成器的要求是严格的。首先，所生成的目标代码必须保持源程序的语义，并且是高质量的。所谓高质量的目标程序，是指能够高效利用目标机器资源的程序。其次，代码生成器本身应该高效地运行。此外，易于实现、测试和维护也是重要的设计目标。

从理论上讲，产生最优代码问题是不可判定的。在实践中，有很多技术能够产生令人满意的代码（虽不是最优的）。本章仅介绍一个简单的代码生成算法，以便读者能对代码生成技术有一个大致的了解。

8.1 代码生成器设计中的问题

虽然代码生成器的具体细节因目标机器和操作系统不同而不同，但是很多问题（如存储管理、指令选择、寄存器分配和指派、计算次序选择等）是几乎所有的代码生成器都会碰到的问题。本节考察代码生成器设计中的一些公共问题，其中存储管理在第 6 章已经详细介绍。

代码生成器的输入包括由前端产生的中间表示和符号表信息。其中，符号表信息用来决定中间表示中名字所代表的数据对象的运行地址。本章采用的中间表示是三地址代码。

8.1.1 目标程序

代码生成器的输出称为目标程序。像中间代码那样，目标程序的形式也有多种，如绝对机器语言程序、可重定位机器语言程序或汇编语言程序。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第312页 文件编号: 422624 页面编号: 33124538449 打印人: 8685 2026-03-09 10:23:03

可重定位机器语言程序是指代码装入内存的起始地址可以任意，代码中有一些重定位信息，以适应重定位的要求。绝对机器语言程序装入内存的起始地址是固定的。

产生绝对机器语言程序作为输出的好处是，目标程序可以放在内存的固定地方并且立即执行。

产生可重定位机器语言程序（经常被称为目标模块）作为输出，则允许分别编译子程序。连接装入程序可连接一组可重定位目标模块并将其装入机器等待运行。虽然产生可重定位目标模块必定会增加额外的开销以进行连接，但带来的好处是提升了程序的灵活性。因为这种方式允许分别编译程序模块或子程序，可以从目标模块中调用其他先前编译好的程序模块。如果目标机器不能自动管理重定位，编译器必须提供显式的重定位信息，供连接装入程序连接分别编译的程序模块。第 11 章将详细介绍目标模块中的重定位信息和连接信息。

产生汇编语言程序作为输出，能有效简化代码生成的过程，因为可以产生符号指令并利用汇编器的宏机制来帮助生成代码，所付出的代价是代码生成后的汇编工序。

6.2.3 小节和 6.2.4 小节介绍了 x86、ARM、龙架构、RISC-V 架构的主要寄存器资源，并结合实例说明了这些架构汇编代码的特点和异同。本章主要使用 CISC（complex instruction set computer，复杂指令集计算机）风格的目标机器，并使用简要的汇编代码作为目标语言。同时，对于部分示例，还将展示 x86、ARM、龙架构的汇编代码。需要指出的是，只要可以通过符号表中的偏移和相关信息计算地址，那么产生名字的重定位地址或绝对地址就与产生符号地址一样简单。

8.1.2 指令选择

代码生成器必须把中间表示程序映射到能够在目标机器上执行的指令序列。映射的复杂性取决于中间表示的层次、指令集架构的性质和所生成代码的质量期望等因素。

如果中间表示的层次较高，则代码生成器可以依据代码模板把中间代码的每个语句翻译成一个机器指令序列。然而这样逐个语句产生目标代码的方式总是产生的是需要进一步优化的低劣代码。如果中间表示可以反映下面目标机器的低层细节，那么代码生成器可以利用这些信息生成较有效的代码序列。

目标机器指令集的性质决定了指令选择的难易程度。指令集的统一性和完备性是其中两个重要的因素。如果目标机器不能以统一的方式支持各种数据类型，那么每种例外的数据类型都需要专门的处理。例如，在某些机器上，浮点运算需要使用分离寄存器。

指令速度和机器方言（machine idiom）是另外两个重要的因素。如果不考虑目标程序的效率，指令选择是直截了当的。对每类三地址语句，可以设计所生成的目标代码的框架。例如，形式为 x = y + z 的三地址语句，若 x、y 和 z 都是静态分配，那么它可以翻译成代码序列：

```
MOV  y, R0   /* 把y装入寄存器R0 */
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第313页 文件编号: 422624 页面编号: 33134538447 打印人: 8685 2026-03-09 10:23:03


```
ADD  z, R0   /* z加到R0上 */
MOV  R0, x   /* 把R0存入x中 */
```

注意，这里使用了简化形式的汇编指令，其中 MOV 表示传送指令，ADD 表示加法指令。这种简化形式有助于更清晰地展示指令的基本功能，而不涉及目标架构的具体细节。

然而遗憾的是，这种逐条语句产生代码的方式常常会得到低质量的代码。例如

```
1  int a,b,c,d;
2  void f() {
3      a = b+c;
4      d = a+c;
5  }
```

中的第 3、4 行语句将翻译成

```
MOV  b, R0
ADD  c, R0
MOV  R0,a
MOV  a, R0
ADD  e, R0
MOV  R0,d
```

显然，第 4 条指令是多余的。如果能够知道 a 以后不再使用，那么第 3 条也是多余的。实际编译器针对此例生成的不同架构汇编代码的示例见 8.1.3 小节。

所生成的代码的质量通常取决于它的长度和执行速度。在大多数机器上，一个中间表示程序可以由许多不同的指令序列来实现，并且不同的实现之间的代价差别相当大。因此对中间代码进行简单的翻译会产生正确的但效率可能难以接受的目标代码。

例如，如果目标机器有加 1 指令（INC），那么三地址语句 a = a + 1 的高效实现是一条指令 INC a，而不是下面的指令序列：

```
MOV  a, R0
ADD  #1,R0
MOV  R0,a
```

为设计好的代码序列，需要知道指令的代价。不幸的是精确的代价信息通常很难得到。决定哪个指令序列对给定的三地址构造是最优的，可能还要用到该构造出现的上下文知识。

8.1.3 编译器生成的不同架构汇编码示例

为了让读者对逐语句代码生成与实际架构汇编码有感性认识，我们对 8.1.2 节的示例程序用 GCC 编译，产生的 x86-64、鲲鹏 AArch64、龙架构 LoongArch64 汇编码对比如表 8.1，其中略去了过程调用和返回等部分的汇编代码，这部分可参见 6.2.4 小节。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第314页 文件编号: 422624 页面编号: 33144538445 打印人: 8685 2026-03-09 10:23:03

表 8.1 几种架构的汇编代码对比

源程序	x86-64	ARM64（AArch64）	龙架构（LoongArch64）
int a,b,c,d;	.globl a .bss .align 4 .type a, @object .size a, 4 a: .zero 4 ...	.comm a,4,4 .comm b,4,4 .comm c,4,4 .comm d,4,4	.comm a,4,4 .comm b,4,4 .comm c,4,4 .comm d,4,4

a = b+c;	movl b(%rip), %edx
	movl c(%rip), %eax
	addl %edx, %eax
	movl %eax, a(%rip)
		adrp x0, b
		add x0, x0, :lo12:b
		ldr w1, [x0]
		adrp x0, c
		add x0, x0, :lo12:c
		ldr w0, [x0]
		add w1, w1, w0
		adrp x0, a
		add x0, x0, :lo12:a
		str w1, [x0]
			la.global $r12,b
			ld.w $r13,$r12,0
			la.global $r12,c
			ld.w $r12,$r12,0
			add.w $r12,$r13,$r12
			or $r13,$r12,$r0
			la.global $r12,a
			st.w $r13,$r12,0

d = a+c;	movl a(%rip), %edx
	movl c(%rip), %eax
	addl %edx, %eax
	movl %eax, d(%rip)
		adrp x0, a
		add x0, x0, :lo12:a
		ldr w1, [x0]
		adrp x0, c
		add x0, x0, :lo12:c
		ldr w0, [x0]
		add w1, w1, w0
		adrp x0, a
		add x0, x0, :lo12:d
		str w1, [x0]
			la.global $r12,a
			ld.w $r13,$r12,0
			la.global $r12,c
			ld.w $r12,$r12,0
			add.w $r12,$r13,$r12
			or $r13,$r12,$r0
			la.global $r12,d
			st.w $r13,$r12,0

ARM64 (AArch64) 汇编指令解释如下：

- adrp x0, b：将 b 所在页的地址加载到寄存器 x0 中，adrp 表示 “address of page”（页地址）。
- add x0, x0, :lo12:b：将 x0 中存储的页地址和 b 的低 12 位偏移量相加，最终得到 b 的实际地址，并存回到 x0。
- ldr w1, [x0]：从 x0 寄存器指向的内存地址（即 b 的完整地址）加载数据，并将其存入 w1 寄存器中。
- str w1, [x0]：将寄存器 w1 中的值写入 x0 寄存器指向的内存地址。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第315页 文件编号: 422624 页面编号: 33154538442 打印人: 8685 2026-03-09 10:23:03

龙架构（LoongArch64）汇编指令解释如下：

- `la.global $r12, a`：用于加载全局符号（变量或标签）a 的地址到寄存器 $r12 中。
- `ld.w $r13, $r12, 0`：从寄存器 $r12 指向的内存中加载一个字（word）到 $r13。
- `add.w $r12, $r13, $r12`：将两个寄存器 $r13 和 $r12 中的值相加，并将结果存入寄存器 $r12。
- `or $r13, $r12, $r0`：将寄存器 $r12 和零寄存器 $r0 中的值进行按位或操作，结果存放到 $r13，该操作相当于将 $r12 的值复制到 $r13 中。
- `st.w $r13, $r12, 0`：将寄存器 $r13 中的一个字存入 $r12 指向的内存地址。

从表 8.1 中可以看出，虽然 x86-64、ARM64 和龙架构的汇编代码在具体形式上有所差异，但它们采用了类似的生成策略：编译器通过逐条翻译每条源代码语句来生成目标代码。从表 8.1 中还可以看到指令集引起的差异。主要有以下三方面：

（1）指令集复杂性。CISC 如 x86-64 提供复杂指令，每条指令可执行多个操作（如加载、运算和存储的组合），从而用更少的指令完成任务。相比之下，RISC（reduced instruction set computer 精简指令集计算机）如 ARM64 和龙架构则强调简化指令集，每条指令只执行单一操作，指令长度固定，简化了指令解码过程，提升了流水线效率，但对于同样的任务可能需要更多的指令来完成。

（2）执行效率与硬件设计。CISC 复杂指令需要更高性能的硬件和微代码支持。而 RISC 通过简化指令集并直接由硬件执行，减少了解码复杂度，能够通过更高效的流水线和指令并行机制提升执行效率。

（3）内存访问与寄存器使用。CISC 允许指令直接操作内存，减少了对寄存器的依赖，但同时增加了内存访问次数。RISC 则着重于寄存器间的操作，内存访问仅限于加载和存储指令，缓解了内存瓶颈，优化了数据处理速度。

8.1.4 寄存器分配

运算对象处于寄存器中的指令通常比运算对象处于内存的指令要短一些，执行也快一些。因此，充分利用寄存器对生成高质量的代码尤其重要。寄存器的使用可以分成两个子问题：

（1）寄存器分配：为程序各点选择驻留在寄存器中的一组变量。

（2）寄存器指派：为驻留寄存器的变量选择保存其值的具体寄存器。

很难找出最优的寄存器指派方案，即便对于单寄存器机器也是一样。这个问题属于 NP 完全问题。因为目标机器的硬件和/或操作系统可能对寄存器的使用有一些要求，问题的复杂度还会进一步增加。

本章介绍的简单代码生成算法将寄存器分配和指派合在一起，统称为寄存器分配。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第316页 文件编号: 422624 页面编号: 33164538440 打印人: 8685 2026-03-09 10:23:03

8.1.5 计算次序选择

计算的执行次序会影响目标代码的效率。例如，对某个表达式，一种计算次序可能会比其他次序占用较少的寄存器来保存中间结果。选择最佳计算次序是一个 NP 完全问题。本章只讨论按中间代码生成器所生成的三地址语句的次序来生成目标代码，但在习题中将给出一些体现计算次序选择的例子。第 10 章讨论在一个时钟周期内能执行多条指令的流水线机器上的代码调度。

8.2 目标语言

熟悉目标机器和它的指令集是设计一个良好代码生成器的先决条件。遗憾的是，在代码生成的一般性讨论中，不能对目标机器描述到足够详细的程度，因而难以为该机器完整的机器语言产生高效的代码。本章选择一种简化的多寄存器机器作为目标计算机，所提出的一些技术可用于许多其他类机器。

8.2.1 目标机器的指令集

该目标机器是字节寻址机器，4 个字节组成 1 个字。有 n 个通用寄存器 R0，R1，…，Rn-1。它有形式为

op 源，目的

的二地址指令，其中 *op* 是操作码，源和目的都是数据域。该机器有如下操作码：

```
MOV     /* 源传到目的 */
ADD     /* 源加到目的 */
SUB     /* 目的减去源 */
```

其他的指令等用到时再介绍。

由于源和目的这两个域没有长到足以保存内存地址，所以它们的某些位用来指明指令的下一个字或两个字包含运算对象或地址。于是，一条指令的源和目的由带地址模式的寄存器和内存单元的组合来指明。在下面的描述中，contents(a) 表示由 a 代表的寄存器或内存单元的内容。

地址模式和它们的汇编语言形式及附加代价见表 8.2。

8.2.2 小节将解释附加代价。

当用作源或目的时，内存单元 M 和寄存器 R 都代表本身。例如，指令

```
MOV R0,  M
```

把寄存器 R0 的内容存入内存单元 M。

寄存器 R 的值加上偏移 c 表示的地址写成 c(R)。例如，指令

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第317页 文件编号: 422624 页面编号: 33174538448 打印人: 8685 2026-03-09 10:23:04


```
MOV 4(R0), M
```

把值

contents(4 + contents(R0))

存入内存单元 M。

表 8.2 地址模式和汇编语言形式及附加代价

地址模式	汇编语言形式	地址	附加代价
绝对地址	M	M	1
寄存器	R	R	0
变址	c(R)	c + contents(R)	1
间接寄存器	*R	contents(R)	0
间接变址	*c(R)	contents(c + contents(R))	1
直接量	#c	c	1

上述两个模式的间接形式由加前缀 * 来指明。例如，指令

```
MOV *4(R0), M
```

把值

contents(contents(4 + contents(R0)))

存入内存单元 M。

最后一个地址模式允许源是常数：

模式 形式 常数 附加代价

立即数 #c c 1

例如，指令

```
MOV #1, R0
```

把 1 保存到寄存器 R0 中。

8.2.2 指令代价

在目标机器上，指令的代价是

1 +源地址模式的附加代价+目的地址模式的附加代价

其中，附加代价见表 8.2 的最后一列。这个代价对应指令以字计算的长度。寄存器地址模式的代价是 0，而那些含内存单元或常数的地址模式的代价是 1，因为这样的运算对象必须与指令存放在一起。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第318页 文件编号: 422624 页面编号: 33184538446 打印人: 8685 2026-03-09 10:23:04

如果空间是至关重要的，应该使指令的长度尽可能短。这样做的好处是：对大多数机器和大多数指令来说，从内存取指令的时间超过执行指令的时间，因而极小化指令的长度也使得指令的执行时间趋于最小。下面是几个例子：

（1）指令 MOV R0, R1 把寄存器 R0 的内容复写到寄存器 R1，这条指令的代价是 1，因为它仅占 1 个字的内存。

（2）指令 MOV R5, M 把寄存器 R5 的内容复写到内存单元 M。这条指令的代价是 2，因为内存单元的地址 M 存于指令的第二个字中。

（3）指令 ADD #1, R3 把常数 1 加到寄存器 R3 的内容上。这条指令的代价是 2，因为常数 1 出现在指令的第二个字中。

（4）指令 SUB 4(R0), *12(R1) 把值

$$\text{contents}(\text{contents}(12 + \text{contents}(\text{R1}))) * \text{contents}(4 + \text{contents}(\text{R0}))$$

存入目的地址 *12(R1)。这条指令的代价是 3，因为常量 4 和 12 存于指令的第二个字和第三个字中。

为这种机器产生代码的困难可以从下面的例子略知一二。考虑为形式是 a = b + c 的三地址语句产生代码，其中 a、b 和 c 都是简单变量，它们分配在静态数据区，直接用它们的名字表示相应的单元。这个语句可以由许多不同的指令序列来实现，例如：

```
(1)  MOV b, R0
     ADD c, R0          代价= 6
(2)  MOV R0, a
     MOV b, a
     ADD c, a           代价= 6
```

（3）若 R0、R1 和 R2 分别含 a、b 和 c 的地址，则可以使用

```
MOV *R1, *R0
ADD *R2, *R0     代价= 2
```

（4）若 R1 和 R2 分别含 b 和 c 的值，并且 b 的值在这个赋值后不再需要，则可以使用

```
ADD R2, R1
MOV R1, a        代价= 3
```

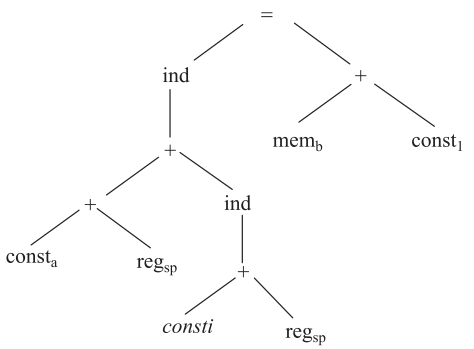
可以看出，要想为这种机器产生好的代码，必须有效地使用它的寻址能力。如果可能，可尽量把名字的左值或右值保存在寄存器中，以便在不久的将来使用。

8.2.3 指令选择的实现：中间代码树和 DAG

【例 8.1】 图 8.2 是 C 语言赋值表达式 a[i] = b + 1 的中间代码树，其中 a[i]、i 和 b 都是 long 类型，并且 a 和 i 都是动态栈式分配的，b 是静态分配的。为简单起见，假定机器

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第319页 文件编号: 422624 页面编号: 33194538444 打印人: 8685 2026-03-09 10:23:04

是按字编址，并且 long 类型的值占一个字。与第 7 章的中间代码不同的是，图 8.2 形式的中间代码是一种层次较低的中间代码，它体现了很多和机器有关的特征，如动态栈式分配变量的地址描述和左右值的区别，因此它向代码生成提供了丰富的信息。



(a) 中间代码树

```
MOV #a, R0
ADD SP, R0
ADD i(SP), R0
MOV b, R1
INC R1
MOV R1, *R0
```

(b) 目标代码序列

图 8.2 赋值表达式 a[i] = b +1 的中间代码树和目标代码序列

图 8.2(a) 中结点标记的解释如下：

（1）reg_{sp} 表示栈的基址寄存器 SP；

（2）const 表示常数，其中 $const_1$ 表示常数 1，$const_a$ 和 $const_i$ 分别表示 a 和 i 在活动记录中的相对地址，它们是常数；

（3）mem 表示内存地址，mem_b 表示分配给 b 的内存地址；

（4）ind 表示间接访问，它的子树是一个左值表达式，当需要左值时，直接用 ind 下面的子树；当需要右值时，把 ind 下面的子树作为地址，取该地址的内容。

有了这个解释，就不难理解表达式树的含义了。mem_b+ $const_1$ 表示 b+1，$const_i$+ reg_{sp} 表示 i 的地址，$const_a$+ reg_{sp} 表示 a 的起始地址。($const_a$+ reg_{sp}) + ind($const_i$+ reg_{sp}) 表示 a[i] 的地址，注意这是一个左值（a 的起始地址）和一个右值（i 的值）相加，得到的是一个左值。ind((const$_a$ + reg$_{sp}$) + ind(const$_i$+ reg$_{sp}$)) 在赋值号的左边，因此生成代码时直接用最外层括号中的左值表达式。

用本节介绍的各种指令和一种叫作树重写的指令选择算法（本书不介绍），可为该中间代码树生成如图 8.2(b) 的代码序列。

有向无环图（directed acyclic graph，DAG）：一种特殊的有向图结构。现代编译器常使用 DAG 来表示基本块或函数的一部分。在 DAG 中，结点表示计算操作或数据值，而有向边则表示计算之间的依赖关系。与控制流图（control flow graph，CFG）相比，DAG 更适用于表示计算流程，因为它能有效展现各个操作的顺序关系以及数据流的依赖性。

例如，LLVM 中的 SelectionDAG 就是一种用于中间代码优化的 DAG 结构。在 LLVM 中，SelectionDAG 用来表示目标机器指令的选择过程。它通过构建 DAG 来帮助编译器进行

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第320页 文件编号: 422624 页面编号: 33204538442 打印人: 8685 2026-03-09 10:23:04

各种优化，如指令合并、常量折叠、死代码消除等。使用 DAG 的好处是能够简化对基本块或函数中各个操作的管理，使得优化算法可以高效地进行，同时避免了复杂的控制流问题。

通过这种方式，编译器不仅能够优化指令选择过程，还能更好地处理依赖关系，从而提升生成代码的质量和效率。

8.2.4 LLVM 后端描述方法：TableGen

为了简化后端的开发工作，LLVM 提供一种 TableGen 的工具，用于描述目标架构的关键信息。

1. TableGen 简介及使用

TableGen 是一种领域特定语言，用于以声明式的方式定义目标机器信息，包括指令集、寄存器、指令模式匹配规则和其他机器相关特性。开发者通过编写扩展名为.td 的文件来描述这些信息，由 TableGen 工具 llvm-tblgen 处理并生成相关的 C++ 代码（文件扩展名为.inc）供 LLVM 后端使用。常用命令如下：

```
// 打印所有的记录（包括抽象记录和具体记录）
llvm-tblgen -print-records <input-file> -o <output-file>
// 生成寄存器信息所对应的 C++ 代码
llvm-tblgen -gen-register-info <input-file> -o <output-file>
// 打印指定类别对应的 C++ 代码中的枚举值,<class-name>可以为Register、Instruction等
llvm-tblgen -print-enums -class=<class-name> <input-file> -o <output-file>
```

每个 TableGen 源文件由 0 条或多条语句（Statement）组成，语句有 10 种：

Statement ::= Assert | Class | Def | Defm | Defset | Defvar | Foreach | If | Let | MultiClass

其中，Class、Def 和 Let 语句比较常用。语句 Class 用于定义一个抽象的记录类，它可以继承自一个或多个其他抽象记录类；语句 Def 用于定义一个具象记录（抽象记录类的实例）；语句 Let 用于设置抽象记录类或具象记录中的字段。

2. 寄存器和寄存器类描述

TableGen 允许开发者定义目标架构的寄存器及其分组信息。例如，在 llvm/lib/Target/x86/x86RegisterInfo.td 文件中定义了如下 x86Reg 抽象类，它继承自 llvm/Target/Target.td 中定义的 Register 抽象记录：

```
class x86Reg<string n, bits<16> Enc,list<Register> subregs = []> : Register<n> {
    let Namespace = “x86";
    let HWEncoding = Enc;
    let SubRegs = subregs;
}
```

将 x86Reg 作为父类定义具体的寄存器：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第321页 文件编号: 422624 页面编号: 33214538440 打印人: 8685 2026-03-09 10:23:04


```
let SubRegIndices = [sub_16bit, sub_16bit_hi], CoveredBySubRegs = 1 in {
def EAX : x86Reg<"eax", 0, [AX, HAX]>, DwarfRegNum<[-2, 0, 0]>;
def EDX : x86Reg<"edx", 2, [DX, HDX]>, DwarfRegNum<[-2, 2, 2]>;
def ECX : x86Reg<"ecx", 1, [CX, HCX]>, DwarfRegNum<[-2, 1, 1]>;
def EBX : x86Reg<"ebx", 3, [BX, HBX]>, DwarfRegNum<[-2, 3, 3]>;
def ESI : x86Reg<"esi", 6, [SI, HSI]>, DwarfRegNum<[-2, 6, 6]>;
def EDI : x86Reg<"edi", 7, [DI, HDI]>, DwarfRegNum<[-2, 7, 7]>;
def EBP : x86Reg<"ebp", 5, [BP, HBP]>, DwarfRegNum<[-2, 4, 5]>;
def ESP : x86Reg<"esp", 4, [SP, HSP]>, DwarfRegNum<[-2, 5, 4]>;
def EIP : x86Reg<"eip", 0, [IP, HIP]>, DwarfRegNum<[-2, 8, 8]>;
}
```

在 x86RegisterInfo.td 文件中，除了定义寄存器，还定义了许多寄存器类 (register class)，如下面定义了 32 位通用寄存器类 GR32，它是 Target.td 中定义的寄存器类抽象记录的实例：

```
def GR32 : RegisterClass<"x86", [i32], 32,
                         (add EAX, ECX, EDX, ESI, EDI, EBX, EBP, ESP, R8D, R9D,
                              R10D, R11D, R16D, R17D, R18D, R19D, R22D, R23D,
                              R24D, R25D, R26D, R27D, R30D, R31D, R14D, R15D,
                              R12D, R13D, R20D, R21D, R28D, R29D)>;
```

3. 指令描述

TableGen 允许开发者定义指令的操作码、操作数、模式匹配规则和编码格式。例如，在 llvm/lib/Target/x86/x86InstrFormat.td 中定义了所有指令的超类：

```
class x86Inst<bits<8> opcod, Format f, ImmType i, dag outs, dag ins,
              string AsmStr, Domain d = GenericDomain>
  : Instruction { ... }
```

对于不同类型的指令，可以再定义不同的抽象类。例如，在 llvm/lib/Target/x86/x86InstrUtils.td 中定义名为 I 的通用指令模板类，它直接继承自 x86Inst：

```
class I<bits<8> o, Format f, dag outs, dag ins, string asm,
        list<dag> pattern, Domain d = GenericDomain>
  : x86Inst<o, f, NoImm, outs, ins, asm, d> {...}
```

其中，模板参数 o 表示指令的 8 位操作码，f 为指令的编码格式；outs 和 ins 分别是指令的输出和输入参数，均用 DAG 表示，方便进行模式匹配；asm 是指令的汇编表示；pattern 是一个 DAG 列表，表示 LLVM IR 中可能与该指令匹配的模式，用于从 LLVM IR 映射到目标机器指令的指令选择阶段；d 表示指令所属的域，默认为 GenericDomain。

进一步地，可以定义具体的指令。例如，在 llvm/lib/Target/x86/x86InstrArithmetic.td 文件中描述算术指令，下面的 LEA16r 描述了 x86 的 LEA（load effective address）指令的 16 位版本，它继承自上面定义的 I 类，并为模板参数赋值以描述具体的指令特性：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第322页 文件编号: 422624 页面编号: 33224538448 打印人: 8685 2026-03-09 10:23:04


```
def LEA16r    : I<0x8D, MRMSrcMem,
                      (outs GR16:$dst), (ins anymem:$src),
                      "lea{w}\t{$src|$dst}, {$dst|$src}", []>, OpSize16;
```

其中，指令操作码为 0x8D，编码格式 MRMSrcMem 表示使用 ModR/M 字节[①]对源操作数进行寻址；指令的输出操作数是通用的 16 位寄存器（GR16），输入操作数为任意内存地址（anymem）。

此外，TableGen 提供模式匹配机制将 LLVM 的中间表示（intermediate representation，IR）转换为目标指令，定义寄存器分配规则以支持调用约定下的参数传递和返回值，并描述目标指令的调度模型以优化性能。详细信息可查阅相关资料。

4. 生成.inc 文件

在编写完描述目标机器特征的.td 文件之后，可以使用 TableGen 工具生成对应的 C++ 代码。例如，执行

```
llvm-tblgen x86.td -gen-register-info
```

可以由 x86.td 描述生成对应的 x86 目标机器相关的 C++ 代码。例如，下面是 GR32 寄存器类变量在.inc 文件中的表示：

```
// GR32 Register class...const MCPhysReg GR32[]={
    x86::EAX, x86::ECX, x86::EDX, x86::ESI, x86::EDI, x86::EBX, x86::EBP,
    x86::ESP, x86::R8D, x86::R9D, x86::R10D, x86::R11D, ...
};
```

各个寄存器在.inc 中用枚举类型表示：

```
namespace llvm{
class MCRegisterClass;
extern const McRegisterClass x86MCRegisterclasses[];
namespace x86{
enum {
    NoRegister,
    AH = 1, AL = 2,AX = 3, BH = 4BL =5,BP = 6, ...
}
```

5. 指令代价

在 llvm/Target/×××/×××TransformationInfo.cpp 中规定了 ××× 架构中指令代价，以 x86 为例，下面的 SLMCostTable 存储了指令的代价信息，SLM 指代某个微架构（如 Intel 的 Silvermont 微架构）：

```
using CostKindTblEntry = CostTblEntryT<CostKindCosts>;
...
```

① ModR/M 字节是 x86 指令集架构中的一个 8 位字段：高 2 位 (Mod) 用来指定地址模式，中间 3 位 (Reg/Opcode) 在寄存器寻址模式下用来指定目标寄存器，低 3 位 (R/M) 指定寄存器或内存的操作数。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第323页 文件编号: 422624 页面编号: 33234538446 打印人: 8685 2026-03-09 10:23:04


```
static const CostKindTblEntry SLMCostTable[] = {
    { ISD::MUL,   MVT::v4i32, { 11, 11, 1, 7 } }, // pmulld
    { ISD::MUL,   MVT::v8i16, {  2,  5, 1, 1 } }, // pmullw
    { ISD::FMUL,  MVT::f64,   {  2,  5, 1, 1 } }, // mulsd
    { ISD::FMUL,  MVT::f32,   {  1,  4, 1, 1 } }, // mulss
    { ISD::FMUL,  MVT::v2f64, {  4,  7, 1, 1 } }, // mulpd
    { ISD::FMUL,  MVT::v4f32, {  2,  5, 1, 1 } }, // mulps
    { ISD::FDIV,  MVT::f32,   { 17, 19, 1, 1 } }, // divss
    { ISD::FDIV,  MVT::v4f32, { 39, 39, 1, 6 } }, // divps
    { ISD::FDIV,  MVT::f64,   { 32, 34, 1, 1 } }, // divsd
    { ISD::FDIV,  MVT::v2f64, { 69, 69, 1, 6 } }, // divpd
    { ISD::FADD,  MVT::v2f64, {  2,  4, 1, 1 } }, // addpd
    { ISD::FSUB,  MVT::v2f64, {  2,  4, 1, 1 } }, // subpd
    // v2i64/v4i64 mul is custom lowered as a series of long:
    // multiplies(3), shifts(3) and adds(2)
    // slm muldq version throughput is 2 and addq throughput 4
    // thus: 3X2 (muldq throughput) + 3X1 (shift throughput) +
    //       3X4 (addq throughput) = 17
    { ISD::MUL,   MVT::v2i64, { 17, 22, 9, 9 } },
    // slm addq\subq throughput is 4
    { ISD::ADD,   MVT::v2i64, {  4,  2, 1, 2 } },
    { ISD::SUB,   MVT::v2i64, {  4,  2, 1, 2 } },
  };
```

LLVM 自 16.x 版本开始使用如下模板类 CostTblEntryT 定义代价表条目的类型，以支持更多的代价类型；之前版本使用的 CostTblEntry 结构类型在这里被实例化为 CostTblEntryT<unsigned>。

```
/// Cost Table Entry
template <typename CostType>
struct CostTblEntryT {
  int ISD;
  MVT::SimpleValueType Type;
  CostType Cost;
};
using CostTblEntry = CostTblEntryT<unsigned>;
```

在上述代价表条目的结构中，ISD 是 SelectionDAG 结点的编号，Type 是目标机器值类型，Cost 是 SelectionDAG 结点对应的代价。

8.3 基本块和流图

在 7.2 节已介绍语句序列形成的基本块和流图的定义。本节介绍基本块的优化、基本块中名称的下次引用信息。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第324页 文件编号: 422624 页面编号: 33244538444 打印人: 8685 2026-03-09 10:23:04

8.3.1 基本块的优化

三地址语句 x = y + z 被称为**引用** y 和 z 的值并对 x **定值**。如果一个名字的值在基本块的某点之后还要引用（包括在后继基本块的引用），则称这个名字在该点是**活跃**的。

若两个基本块是**等价**的，则在它们的出口点有同样的活跃变量集合，并且在这两个出口点，对于该集合中的任何一个活跃变量，代表该变量值的两个表达式可证明是相等的。

有很多等价变换可用于基本块，这些变换改进基本块代码的质量，称为**局部优化**。这里介绍两类可用于基本块的局部优化，它们是保结构变换和代数变换。第 9 章介绍的一些全局优化技术，如删除公共子表达式，也可用在局部优化中。作为简单介绍，假定基本块没有数组、指针和过程调用，并且不同名字代表不同变量，即没有别名问题。

先介绍 3 种保结构变换。

1. 删除局部公共子表达式

考虑基本块：

$$a = b + c$$
$$b = a - d$$
$$c = b + c$$
$$d = a - d$$

第二条语句和第四条语句计算同样的表达式，即 b + c − d，因此该基本块可以变换成等价的基本块：

$$a = b + c$$
$$b = a - d$$
$$c = b + c$$
$$d = b$$

虽然第一条语句和第三条语句有同样的 b + c 出现在右部，但由于第二条语句给 b 定值，使得第一条和第三条语句引用的 b 可能有不同的值，所以两个 b + c 的值可能不同。

实际编译器实现时往往用 DAG 表示基本块或函数的一部分，如 LLVM 中的 SelectionDAG，用于简化代码，方便实现优化。

2. 删除死代码

语句 x = y + z 给 x 定值，若 x 此后不再被引用，则称 x 为死变量。删除这样的语句是基本块的一种等价变换。

3. 交换相邻的独立语句

如果基本块有两个相邻的语句：

$$t_1 = b + c$$
$$t_2 = x + y$$

交换这两条语句而不影响基本块所计算的表达式。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第325页 文件编号: 422624 页面编号: 33254538441 打印人: 8685 2026-03-09 10:23:04

有许多代数变换可用来对基本块实施等价变换，其中有价值的是那些可以简化表达式或用较快运算代替较慢运算的变换。例如：

```
x = x + 0
```

或

```
x = x * 1
```

这样的语句可以从基本块删除，这是基本块的等价变换。语句

```
x = y ** 2
```

的指数运算通常需要用函数调用实现。使用代数变换，该语句可以由快速、等价的语句

```
x = y * y
```

代替。

8.3.2 下次引用信息

下面介绍如何收集基本块中名字的下次引用（next-use）信息，8.4 节的简单代码生成器需要该信息来完成寄存器分配。这是因为，如果知道了其值存于一个寄存器的某个名字没有下次引用，那么该寄存器可用来存放其他名字的值。

在一条三地址语句中，名字的**引用**定义如下：假定三地址语句 i 对 x 定值，如果语句 j 用 x 作为运算对象，控制可以从 i 流到 j，并且在这条路径上没有对 x 的其他定值，那么称 j 引用 x 在 i 的定值。

目前需要为基本块中的每条三地址语句 x = y op z 决定 x、y 和 z 在该块中的下次引用信息。暂时不考虑它们在基本块外的被引用情况，若需要这种信息，可用第 9 章的活跃变量分析技术获得。

对每个基本块，收集下次引用信息的算法从最后一条语句反向扫描到第一条语句。在反向扫描一个基本块前，根据第 9 章的活跃变量分析，在符号表中把在出口点活跃的变量都置上活跃标记，并置上没有下次引用（指在本块中没有下次引用）。而对于在出口点不活跃的变量，都置上不活跃也没有下次引用。

假如反向扫描到达三地址语句 i：x = y op z，执行下面几步：

（1）从符号表中找出 x、y 和 z 的下次引用信息和活跃标记，并把它们加到语句 i 上。

（2）在符号表中把 x 置成不活跃和没有下次引用。

（3）在符号表中，置 y 和 z 活跃，并且置它们的下次引用信息为 i。

注意，（2）和（3）的次序不能颠倒，因为 x 可能就是 y 或 z。

如果三地址语句 i 是 x = y 或 x = op y 的形式，其步骤同上，但忽略 z。

利用下次引用信息，也可以压缩临时变量需要的空间。一般来说，如果两个临时变量不同时活跃（生存期不重叠），可以把它们压缩在同一单元中。因为大多数临时变量都在同一

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第326页 文件编号: 422624 页面编号: 33264538449 打印人: 8685 2026-03-09 10:23:04

块中定义和引用，因而下次引用信息就可紧缩临时变量的占用空间。对于穿越多个基本块的引用问题，放在第 9 章数据流分析时讨论。

临时变量存储单元的分配可以这样进行：依次检查临时变量区域的单元，找到第一个不含活跃临时变量的单元，把它分配给待分配的临时变量。如果没有这样的单元，则在活动记录的临时变量区域加一个单元。在许多情况下，临时变量可以压缩到只使用寄存器而不需要内存单元，见 8.4 节的讨论。

例如，下面的基本块有 6 个临时变量，可压缩为只需要两个临时变量，它们是 t_1 和 t_2。

$$\begin{aligned} t_1 &= a * a & t_4 &= t_1 + t_3 \\ t_2 &= a * b & t_5 &= b * b \\ t_3 &= 2 * t_2 & t_6 &= t_4 + t_5 \end{aligned} \tag{8.1}$$

8.4 一个简单的代码生成器

本节的代码生成算法为一个基本块产生目标代码。它依次考虑该块中的每条语句，根据寄存器当前的使用情况（什么值在寄存器中，在哪个寄存器中），为其产生代码，并根据产生的代码修改寄存器的使用情况。这样可以避免产生不必要的装入寄存器和存回内存单元的指令。

代码生成期间的一个基本问题是怎样最佳地使用寄存器。寄存器的 4 种主要用途如下。

（1）在大多数机器架构（machine architecture）上，为完成某种运算，它的部分或所有的运算对象都必须在寄存器中。

（2）寄存器优选作为临时单元，即在计算一个大表达式时，作为保存子表达式值的地方，或者更广一点，作为保存一个基本块中还要使用的值的地方。

（3）寄存器用来保存在一个基本块中计算而在另一个基本块中使用的全局值。例如，在循环体的一次执行中，循环控制变量在某处被修改，可能在多处被引用。

（4）寄存器经常用于运行时存储空间的管理。例如，作为栈顶指针和活动记录的基地址指针。

本节的算法假定有部分寄存器可用于保存基本块中使用的值，假定基本块已完成 8.3 节所提到的一些优化变换，还假定三地址语句中出现的每种运算都有对应的目标机器指令。

另外，本算法将计算结果尽量长时间地保留在寄存器中，只有在下面两种情况下才把它存入内存：

（1）寄存器要用于其他计算；

（2）到了基本块的出口点。

在基本块的出口点，以后还要引用的值都必须保存起来。原因是，本算法的寄存器分配以基本块为单位，不考虑用寄存器保留值来穿越基本块。因为一个基本块可能有若干后继，也

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第327页 文件编号: 422624 页面编号: 33274538447 打印人: 8685 2026-03-09 10:23:04

可能有若干前驱，因而不经过额外的分析，就认为一个基本块引用的某个数据在入口点一定处于某个寄存器中是不稳妥的。

8.4.1 寄存器描述和地址描述

对三地址语句 a = b + c，如果寄存器 Ri 含 b，Rj 含 c，且 b 在此语句后不再活跃，那么可以为它产生代价为 1 的代码 ADD Rj, Ri，结果 a 在 Ri 中。

如果 Ri 含 b，但 c 在内存单元，b 仍然不再活跃，那么可以产生代价为 2 的代码

```
ADD c, Ri
```

或代价为 3 的代码序列

```
MOV c, Rj
ADD Rj, Ri
```

如果 c 的值以后还要用，则该代码序列比较好，因为可以从寄存器 Rj 中取 c 的值。

还有很多的情况可以考虑，这取决于 b 和 c 的值当前在什么地方以及 b 的值以后是否还要用。还必须考虑 b 和 c 的一个或两个都是常数的情况。如果 + 运算可交换，需考虑的情况还会增加。所以，可以看出，代码生成包含了对大量情况的考察，哪种情况占优势依赖于三地址语句出现的上下文。

从上面的例子可以看出，在代码生成过程中，需要跟踪寄存器的内容和名字的地址。本节的代码生成算法使用寄存器和名字的描述来跟踪寄存器的内容和名字的地址。

（1）寄存器描述。它记住每个寄存器当前存储的是什么。初始时寄存器描述表明供基本块使用的寄存器都空闲。随着对基本块的代码生成逐步前进，在任何一点，每个寄存器保存若干个（包括零个）名字的值。寄存器的这些信息可以单独用一张寄存器表来描述。

（2）名字的地址描述。它记住每个名字的当前值可以在哪个场所找到。这个场所可以是寄存器、栈单元、内存地址，甚至是它们的某个集合，因为复写时值仍然留在原来的地方。这些信息可以存于符号表中。

8.4.2 代码生成算法

该代码生成算法以构成一个基本块的三地址语句序列作为输入，对每个三地址语句 $x = y\ op\ z$ 完成下列动作：

（1）调用函数 getReg ()，决定存放 $y\ op\ z$ 计算结果的场所 L。L 通常是寄存器，但也可能是内存单元。8.4.3 小节将简要描述 getReg () 的算法。

（2）查看 y 的地址描述，确定存放 y 当前值的一个场所 y'。如果 y 的值既在内存单元又在寄存器中，优先选择寄存器作为 y'，特别是 y 的值所在的寄存器正好是 L 的时候。如果 y 的值不在 L 中，则产生指令 MOV y′, L，把 y 的值复写到 L 中。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第328页 文件编号: 422624 页面编号: 33284538445 打印人: 8685 2026-03-09 10:23:04

（3）产生指令 $op\ z', L$，其中 z' 是存放 z 当前值的场所之一。同上面一样，如果 z 的值既在寄存器又在内存单元，就优先选择前者。修改 x 的地址描述，以表示 x 在场所 L。如果 L 是寄存器，修改它的描述，以表示它含 x 的值。

（4）如果 y 或 z 的值在块内不再引用，在块的出口点也不活跃，并且还在寄存器中，那么修改寄存器描述，以表示在执行了 $x = y\ op\ z$ 以后，这些寄存器分别不再含 y 或 z 的值。

如果三地址语句有一元运算符，步骤同上面的类似，在此略去。一个重要的特例是复写语句 $x = y$。如果 y 在寄存器中，只要改变寄存器和地址描述，记住 x 的值当前只能在存 y 值的寄存器中找到；进一步，如果 y 在块内不再引用，并且在块出口点也不活跃，那么这个寄存器不再保存 y 的值。如果 y 的值仅在内存，原则上，可以记住 x 的值在 y 的内存单元，但是这样会使算法复杂，因为以后若要改变 y 的值时必须先保存 x 的值。所以，如果 y 在内存，可用 getReg () 来找到一个存放 y 的寄存器，并记住此寄存器也是保存 x 的场所。另一种方法是产生指令 MOV y, x。尤其是 x 在块中不再引用时，这样做比较好。值得注意的是，如果应用第 9 章的各种优化，尤其是复写传播算法，大多数复写指令可以删去。

一旦处理完一个基本块的所有三地址语句，在基本块出口点，用 MOV 指令把那些值尚不在它们内存单元的活跃名字的值都存入它们的内存单元。为完成这件事，用寄存器描述来决定什么名字的值仍在寄存器中，再用地址描述来决定其中哪些名称的值不在它们的内存单元，最后用活跃变量信息来决定其中需要存储的名称。如果基本块之间的数据流分析没有计算活跃变量信息，只好认为所有用户定义的名称在基本块末尾都是活跃的。

8.4.3 寄存器选择函数

函数 getReg 以三地址语句 $x = y\ op\ z$ 为参数，返回所选择的保存 x 值的场所 L。本节算法为实现该函数投入大量精力，旨在更好地选择场所 L。本小节将讨论一种基于下次引用信息的简单方法。

（1）如果名字 y 在寄存器中，此寄存器不含其他名字的值（注意，$x = y$ 这样的复写语句会使得寄存器可能同时保存多个变量的值），并且在执行 $x = y\ op\ z$ 后 y 在块内不再引用，那么返回当前保存 y 值的这个寄存器作为 L。

（2）当（1）失败时，返回一个空闲寄存器（如果有）。

（3）当（2）不能成功时，如果 x 在块中还会被引用，或者 op 是必须用寄存器的运算，如变址操作，那么找一个已被占用的寄存器 R。如果 R 保存的是变量 w 的值，并且还没有保存到 w 的内存单元，则生成 MOV R, w 指令，把 R 的值存入 w，并修改 w 的地址描述，返回 R。如果 R 保存着几个变量的值，那么对于每个需要存储的变量都产生 MOV 指令。怎样恰当地选择这个寄存器呢？可优先选择其数据在将来最晚使用的寄存器，或者其数据同时在内存的寄存器。

（4）如果 x 在块内不再引用或找不到适当的被占用寄存器，可选择 x 的内存单元作为

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第329页 文件编号: 422624 页面编号: 33294538443 打印人: 8685 2026-03-09 10:23:04

L。

更复杂的 getReg () 函数在决定存放 x 值的寄存器时要考虑 x 随后的使用情况和运算符 *op* 的交换性等。

【例 8.2】 赋值语句 d = (a − b) + (a − c) + (a − c) 可以翻译成下面的三地址语句序列：

$$t_1 = a - b$$
$$t_1 = a - c$$
$$t_1 = t_1 + t_1$$
$$d = t_1 + t_1$$

假定只有 d 在基本块出口活跃。本节的代码生成算法为该三地址语句序列产生如表 8.3 所示的代码序列。

表 8.3 中给出代码生成过程中相关的寄存器描述和地址描述，但是忽略了 a、b 和 c 的值总是在内存中这样一个事实。

表 8.3 目标代码序列

语句	生成的代码	寄存器描述	名称地址描述
		寄存器空闲	
$t_1 = a - b$	MOV a，R0 SUB b，R0	R0 含 t_1	t_1 在 R0 中
$t_2 = a - c$	MOV a，R1 SUB c，R1	R0 含 t_1 R1 含 t_2	t_1 在 R0 中 t_2 在 R1 中
$t_3 = t_1 + t_2$	ADD R1，R0	R0 含 t_3 R1 含 t_2	t_3 在 R0 中 t_2 在 R1 中
$d = t_3 + t_2$	ADD R1，R0	R0 含 d	d 在 R0 中
	MOV R0，d		d 在 R0 和内存中

getReg( ) 的第一次调用返回 R0 作为保存 t_1 的场所。因为 a 不在 R0，所以产生 MOV a，R0 和 SUB b，R0 的指令。修改寄存器描述表示 R0 含 t_1。

代码生成以这种方式前进，直到最后一个三地址语句处理完。注意，这时 R1 为空，因为 t_2 不再引用，然后在基本块的出口点产生 MOV R0, d，存储活跃变量 d。

表 8.3 生成的代码的代价是 12。可以把它缩减到 11，在第一条指令后立即产生指令 MOV R0，R1，删去指令 MOV a，R1，但是这需要更复杂的代码生成算法。代价能降低的原因是从 R1 取到 R0 比从内存取到 R0 要快。

8.4.4 为变址和指针语句产生代码

变址与指针运算的三地址语句的处理和二元运算符的处理相同。表 8.4 给出了为变址语句 a = b [i] 和 b[i] = a 产生的代码序列，假定 b 是静态分配的。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第330页 文件编号: 422624 页面编号: 33304538441 打印人: 8685 2026-03-09 10:23:04

i 当前所在场所决定了代码序列。表 8.4 给出 3 种情况，分别是 i 在寄存器 Ri 中、i 在内存单元 Mi 中、i 在栈中。对于后者，偏移为 Si，且 i 所在活动记录的基地址寄存器是 Rs。寄存器 R 是调用函数 getReg () 时返回的寄存器，对于赋值 a = b[i]，如果 a 在块中有下次引用，并且寄存器 R 是可用的，那么宁愿把 a 留在寄存器 R 中。

表 8.4 变址语句的代码序列

语句	i 在寄存器 Ri 中		i 在内存 Mi 中		i 在栈中	
	代码	代价	代码	代价	代码	代价
a = b[i]	MOV b(Ri)，R	2	MOV Mi，R MOV b(R)，R	4	MOV Si(Rs)，R MOV b(R)，R	4
b[i] = a	MOV a，b(Ri)	3	MOV Mi，R MOV a，b(R)	5	MOV Si(Rs)，R MOV a，b(R)	5

表 8.5 给出了为指针赋值语句 a = *p 和 *p = a 产生的代码序列，它由 p 当前所在场所决定。同上面一样，这里也给出了 3 种情况。寄存器 R 也是调用函数 getReg () 返回的寄存器。

表 8.5 指针赋值语句的代码序列

语句	p 在寄存器 Rp 中		p 在内存 Mp 中		p 在栈中	
	代码	代价	代码	代价	代码	代价
a = *p	MOV*Rp，a	2	MOV Mp，R MOV*R，R	3	MOV Sp(Rs)，R MOV*R, R	3
*p = a	MOV a，*Rp	2	MOV Mp，R MOV a，*R	4	MOV a，*Sp(Rs)	3

8.4.5 条件语句

机器实现条件转移至少有以下 3 种方式。

1. 基于寄存器值的分支转移

这种方式是根据寄存器中的值是否为下面 6 种情况之一进行分支：负、零、正、非负、非零和非正。在这样的机器上，if x < y goto L 的实现通常是把 x−y 的值存入寄存器 R，如果 R 的值为负，则跳转到 L。

2. 基于条件码的条件转移

这种方式使用条件码来表示计算的结果或寄存器值的状态（如负、零或正）。这种方法适用于很多机器。通常，比较指令（本章机器模型上是 CMP）会设置条件码标志，而非实际计算数值。例如，若 x > y，那么 CMP x, y 置条件码标志为正。条件转移指令根据指定的条

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第331页 文件编号: 422624 页面编号: 33314538449 打印人: 8685 2026-03-09 10:23:04

件 <、==、>、<=、!= 或 >= 是否满足来决定转移。指令 CJ<= L 用来表示如果条件码标志是负或零则跳转到 L。例如，if x < y goto L 可以通过下面来实现：

```
CMP  x, y
CJ<  L
```

当涉及复合表达式时，条件码的使用尤为方便。例如：

```
x = y + z
if x < 0 goto L
```

可以转换为以下伪汇编：

```
MOV  y,  R0
ADD  z,  R0
MOV  R0, x
CJ<  L
```

条件码标志能够准确描述最近一次操作的结果，因此无须额外操作来重新计算判断条件。

3. 基于条件移动的无分支实现

现代硬件还支持通过条件移动指令实现条件选择，而不需要显式的跳转。这种方法依赖特殊的硬件指令来完成，如 x86 的 CMOV 指令和 ARM 的条件执行指令，根据条件直接将值移动到目标寄存器中，避免了分支预测和可能的性能开销。

假设条件为 “x = (y < z) ? a : b;”，可以实现为如下的伪汇编代码：

```
CMP     y,  z   # 比较y和z
CMOVL   R0, a   # 如果 y<z, R0 = a
CMOVGE  R0, b   # 如果 y>=z, R0 = b
```

这种方法无需跳转，能够减少流水线中断，尤其适合条件选择逻辑较为简单的场景。

另外，if-conversion 优化是一种重要的代码优化技术，用于将显式的条件分支转化为无条件操作，从而减少分支跳转带来的性能开销。if-conversion 通常结合条件转移的硬件实现方式（上述第 3 种方式），在特定场景下显著优化指令流水线效率。

8.5 寄存器分配算法

寄存器分配是代码生成的重要阶段，它负责把程序中使用的变量分配到机器的物理寄存器中，对于无法放到物理寄存器中的变量，则需要放到内存中。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第332页 文件编号: 422624 页面编号: 33324538447 打印人: 8685 2026-03-09 10:23:04

8.5.1 线性扫描算法

线性扫描寄存器分配算法于 1999 年被提出[10]。它通过分析变量的活跃性构建活跃区间，然后线性遍历这些活跃区间，决定哪些变量可以保留在寄存器中，哪些需要溢出到内存中（**寄存器溢出**）。在此过程中，它维护一个激活表（active list），用于跟踪当前已分配寄存器的变量及其活跃区间，并根据一定规则分配和释放寄存器。

一个变量的**活跃区间**表示该变量在程序中的使用范围，用 [i, j] 表示，其中 i 是变量第一次被使用的指令编号，j 是最后一次使用的指令编号。

激活表用于存储当前活跃的变量的活跃区间，并按照区间的结束位置递增排序。激活表中的变量表示这些变量已分配寄存器，且其活跃区间尚未结束。

图 8.3 为线性扫描寄存器分配算法示例。

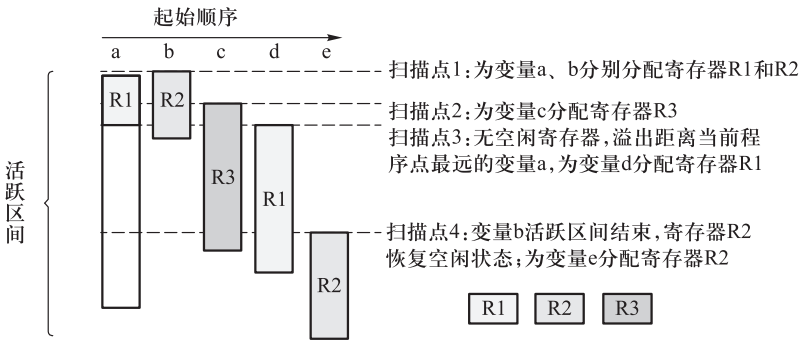


图 8.3 线性扫描寄存器分配算法示例

下面结合图 8.3 的示例，简述线性扫描寄存器分配算法的主要思路和步骤：

（1）排序活跃区间。将所有变量的活跃区间按照起始位置的指令编号 i 递增排序。例如，图中包含 5 个变量 a ～ e，它们的活跃区间按起始位置由上到下排序。

（2）线性扫描。遍历排序后的活跃区间列表，为每个变量分配寄存器。检查激活表中所有活跃区间，移除结束位置早于当前区间起始位置的活跃区间，并释放这些区间占用的寄存器（如图中扫描点 4 处，释放寄存器 R2）。如果没有空闲寄存器可分配，溢出结束位置距当前程序点最远的活跃区间对应的变量（如图中扫描点 3 处，溢出 a 到内存）。如果有空闲寄存器，则将其分配给当前活跃区间，并将该区间添加到激活表中，同时保持激活表按照结束位置递增排序（如图中扫描点 1、2 处，为变量 a、b、c 分配寄存器）。

（3）更新激活表。每次分配或释放寄存器后，更新激活表中的活跃区间信息，确保激活表始终有序。

原始的线性扫描寄存器分配算法如图 8.4 所示。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第333页 文件编号: 422624 页面编号: 33334538445 打印人: 8685 2026-03-09 10:23:04


```
LinearScanRegisterAllocation
    active ← {}
    foreach live interval i, in order of increasing start point
        ExpireOldintervals(i)
        if length (active)=R then
            SpillAtInterval(i)
        else
            register [i] ← a register removed from pool of free registers
            add i to active, sorted by increasing end point

ExpireOldintervals(i)
    foreach interval j in active, in order of increasing end point
        if endpoint [j] ⩾ startpoint [i] then
            return
        remove j from active
        add register [j] to pool of free registers

SpillAtInterval(i)
    spill ← last interval in active
    if endpoint[spill] > endpoint[i] then
        register [i] ← register[spill]
        location[spill] ← new stack location
        remove spill from active
        add i to active, sorted by increasing end point
    else
        location [i] ← new stack location
```

图 8.4 原始线性扫描寄存器分配算法

线性扫描寄存器分配算法属于一类相对轻量级的寄存器分配方法，主要应用于交互式环境和即时编译器等对编译速度敏感的场景中。

其局限性在于活跃区间是粗粒度的，即变量的活跃区间可能覆盖整个使用范围，即便中间大部分时间未被访问，仍会占用寄存器，导致寄存器利用率低。此外，该算法对复杂控制流缺乏优化支持，容易在寄存器压力大的场景下出现溢出问题。

改进线性扫描寄存器分配算法可以通过细化活跃区间划分（如活跃区间分裂）来提高寄存器利用率，避免长期占用无用寄存器。同时，引入更智能的溢出策略，结合程序的控制流和动态运行时信息，优化变量溢出的选择与处理。此外，整合启发式优化技术，适应复杂程序结构和寄存器压力，提高整体分配效率和执行性能。

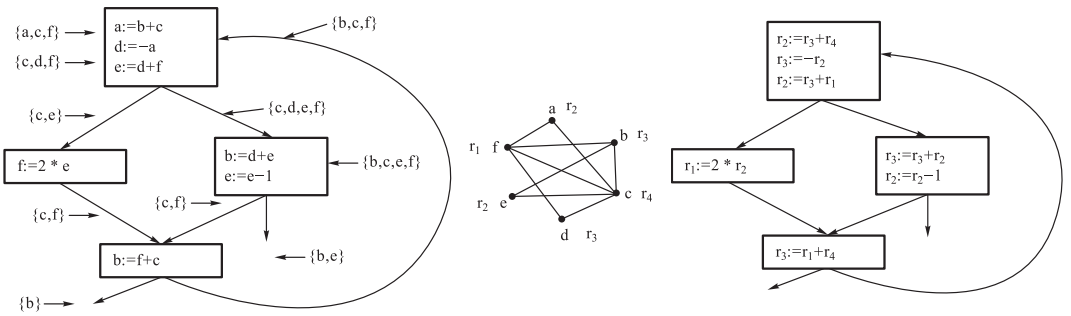
8.5.2 图着色算法

图着色算法是编译器中一种经典的寄存器分配方法[1]。它将寄存器分配问题建模为图着色问题，利用干涉图（interference graph，也称冲突图）表示变量之间的冲突关系，是静态编译环境中常见的高级优化技术。

干涉图是一种无向图，其中结点表示程序中的变量，边表示两个变量在某一程序点上同时活跃（活跃区间重叠）。干涉图是寄存器分配问题建模为图着色问题的核心数据结构。图

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第334页 文件编号: 422624 页面编号: 33344538443 打印人: 8685 2026-03-09 10:23:04

8.5 为图着色寄存器分配算法示例。



(a) 各程序点的活跃变量集合　(b) 干涉图　(c) 着色后的代码（分配寄存器后）

图 8.5　图着色寄存器分配算法示例

下面结合图 8.5 的示例，简述图着色寄存器分配算法的主要思路和步骤。

（1）计算活跃变量集合。在程序的控制流图上执行活跃变量分析（如 9.2.4 节所述），计算每个程序点的活跃变量集合。例如，图 8.5（a）展示了在程序的各个程序点标记的活跃变量集合，包括基本块内各语句之间的活跃变量集合，以及有向边上（由该边进入的基本块入口处）的活跃变量集合。这里变量在某个程序点被认为是活跃的，意味着该变量在该程序点之后的可能路径上至少会在某个地方被引用。

（2）构造干涉图。如图 8.5（b）所示，为每个变量在干涉图中建立一个结点。对于每个程序点的活跃变量集合，如果两个变量同时存在于该活跃变量集合中，那么在干涉图中对应的两个结点之间连接一条边，表示它们在执行过程中会冲突（它们的活跃区间重叠）。例如，活跃变量集合 b、c、f 会在结点 b、c 和 f 之间加三条无向边。

（3）图着色。用图着色算法给干涉图中的每个结点分配颜色，要求相邻结点的颜色不同。每种颜色对应一个寄存器。若干涉图能用不超过可用寄存器数量的颜色着色，则可以完成寄存器分配。如果寄存器数量不足以完成着色，必须选择某些结点溢出到内存（通过存储和加载指令访问）。例如，图 8.5（b）中将 a、e 着色为一种颜色，b、d 着色为一种颜色，共 4 种颜色。

（4）对代码着色（分配寄存器后）。对于每个程序语句，将其中的变量替换为已分配的寄存器编号。在开始时，需要考虑将分配寄存器的变量加载到寄存器中。如果某些变量未能分配到寄存器，则该变量通过存储和加载指令访问。

图着色寄存器分配算法适用于处理复杂控制流和大规模程序，能提供全局优化，尤其在寄存器资源充足时表现优秀。但是，图着色算法计算复杂度高，尤其在即时编译中可能影响编译速度。当寄存器不足时，频繁的溢出处理会增加内存访问，降低性能。此外，它在寄存器资源紧张时可能无法提供最佳的分配方案。

图着色算法的改进主要包括启发式优化、溢出策略的改进、动态寄存器分配和分阶段分

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第335页 文件编号: 422624 页面编号: 33354538440 打印人: 8685 2026-03-09 10:23:04

配。启发式方法通过优先处理度数小的结点来提高效率，溢出策略则选择性能影响最小的变量进行溢出，减少内存访问开销。动态分配利用运行时信息精确调整寄存器分配，而分阶段分配则将问题分解成较小的局部任务，降低计算复杂度并提高寄存器利用率。通过这些改进，图着色算法能够在复杂环境中更高效地工作。

8.5.3 LLVM 中的寄存器分配算法

在 LLVM 源码的 llvm/lib/CodeGen 目录下，实现了以下几种寄存器分配算法。

（1）**Basic 寄存器分配器 RABasic**（RegAllocBasic.cpp）。RABasic 是线性扫描算法的改进，实现了一个最小化的基础寄存器分配算法。它根据溢出权重对寄存器的生存期进行优先级排序，当寄存器不可用时直接执行溢出操作。尽管 RABasic 并不适合在生产环境中实际使用，但作为一种基线实现，它为评估其他寄存器分配器提供了参考标准，并用于对比基本算法与其他分配器的速度和效率。

（2）**Fast 寄存器分配器 RAFast**（RegAllocFast.cpp）。RAFast 是一种专为调试代码设计的寄存器分配算法。它对每个基本块进行寄存器分配，尝试将值保存在寄存器中，并在合适的时候重用寄存器以提高效率。如果没有空闲寄存器可用，RAFast 会选择溢出代价最小的寄存器进行溢出操作。

（3）**Greedy 寄存器分配器 RAGreedy**（RegAllocGreedy.cpp）。RAGreedy 是在 LLVM 3.0 中引入，是 RABasic 的高度优化版本，结合了全局生存期分割和启发式优化技术。它通过贪心策略对寄存器的分配进行优化，努力最小化溢出代码的成本，从而提升了寄存器的利用率和生成代码的性能。与传统的线性扫描分配器相比，RAGreedy 能够更好地处理复杂的寄存器分配问题，在寄存器压力较大的场景中表现尤为优异，同时仍然保持了较高的编译速度。

（4）**PBQP 寄存器分配器**（RegAllocPBQP.cpp）。PBQP 寄存器分配算法基于分区布尔二次编程（partitioned boolean quadratic programming）[3]，它利用 PBQP 对寄存器分配问题进行数学建模，通过 PBQP 求解器生成优化的解决方案，并将其映射回具体的寄存器分配。该方法能够有效处理复杂架构和高寄存器压力场景，在资源紧张时通过优化溢出决策最小化性能损失。如果变量需要溢出，会插入溢出代码并重复分配过程。PBQP 分配器以其高效的优化能力，适用于性能要求高但允许一定编译时间的场景，为生成高质量目标代码提供了重要支持。

习 题

8.1 为下列 C 语句产生 8.2 节目标机器的代码，假定所有的变量都是静态的，并假定有 3 个寄存器可用于保存计算结果。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第336页 文件编号: 422624 页面编号: 33364538448 打印人: 8685 2026-03-09 10:23:04


```
(a) x = 1
(b) x = y
(c) x = x + 1
(d) x = a + b * c
(e) x = a / ( b + c ) - d * ( e + f )
```

8.2 重复 8.1，假定所有的变量都是自动变量（分配在栈上）。

8.3 为下列 C 语句产生8.2节目标机器的代码，假定所有的变量都是静态的，并假定有 3 个寄存器可用于保存计算结果。

```
(a) x = a[i] + 1
(b) a[i] = b[c[i]]
(c) a[i] = a[i] + b[j]
```

8.4 使用 8.4 节的算法重做 8.1。

8.5 一个 C 语言程序如下：

```
1  #include <stdio.h>
2  int fact() {
3      static int i=5;
4      if(i==0) {
5          return(1);
6      }
7      else {
8          i=i-1;
9          return((i+abs(1))*fact());
10     }
11 }
12 int main() {
13     printf("factor of 5 = %d\n", fact
           ());
14 }
```

该程序在早先的 SPARC/SUNOS 系统上，经某编译器编译后运行结果是 120；但是如果把第 9 行的 abs(1) 改成 1，则程序结果是 1。

（1）试分析为什么会有这样不同的结果。

（2）在现代 x86/Linux 系统上用 GCC 编译器编译 [GCC: gcc version 10.5.0 (Ubuntu 10.5.0-1ubuntu1 22.04)]，两种情况下的运行结果都是 120，试说明编译器的演进特点。

8.6 一个 C 语言程序如下：

```
1  #include "stdio.h"
2  int main() {
3      long i;
4      i=0;
5      printf("%ld\n",(++i)+(++i)+(++i));
6  }
```

该程序在 x86/Linux 系统上，编译后的运行结果是 7[编译器版本是 GCC: gcc version 10.5.0 (Ubuntu 10.5.0-1ubuntu1 22.04)]，而在早先的 SPARC/SUNOS 系统上编译后的运行结果是 6。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第337页 文件编号: 422624 页面编号: 33374538446 打印人: 8685 2026-03-09 10:23:04

（1）试分析运行结果不同的原因。

（2）请自行编译和运行，说明编译器的版本和运行结果，并结合汇编码对运行结果进行分析。

8.7 一个 C 语言程序如下：

```
#include "stdio.h"
int main() {
    long i;
    i=10;
    i=(i+5) + (i=i*5);
    printf("%ld\n",i);
}
```

（1）该程序在 x86/Linux 系统上用早期的 GCC 编译器（版本见汇编代码最后一行）编译，运行时输出 105。编译生成的汇编代码如下，从生成的汇编代码可知，表达式 (i+5) + (i = i*5) 的右子树先计算，你能猜测出有关的代码生成策略吗？

```
    .file "ex8-7.c"
    .version  "01.01"
gcc2_compiled.:
.section  .rodata
.LC0:
    .string "%d\n"
.text
    .align 4
.globl main
    .type  main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    subl $4,%esp
    movl $10,-4(%ebp)
    movl -4(%ebp),%edx
    movl %edx,%eax
    sall $2,%eax
    addl %edx,%eax
    movl %eax,%edx
    movl %edx,-4(%ebp)
    leal 5(%edx),%eax
    addl %eax,-4(%ebp)
    movl -4(%ebp),%eax
    pushl %eax
    pushl $.LC0
    call printf
    addl $8,%esp
.L1:
    leave
    ret
.Lfe1:
    .size  main,.Lfe1-main
    .ident  "GCC: (GNU) egcs-2.91.66
        19990314/Linux (egcs-1.1.2
        release)"
```

（2）该程序在 x86/Linux 系统上用较新的 GCC 编译器 [版本是 GCC: gcc version 10.5.0 (Ubuntu 10.5.0-1ubuntu1 22.04)] 编译后，运行时输出 65，汇编代码如下：

```
  .file "ex8-7.c"
  .text
  .section  .rodata
.LC0:
  .string "%ld\n"
  .text
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第338页 文件编号: 422624 页面编号: 33384538444 打印人: 8685 2026-03-09 10:23:04


```
7     .globl  main
8     .type main, @function
9   main:
10  .LFB0:
11    endbr64
12    pushq %rbp
13    movq  %rsp, %rbp
14    subq  $16, %rsp
15    movq  $10, -8(%rbp)
16    movq  -8(%rbp), %rax
17    leaq  5(%rax), %rcx
18    movq  -8(%rbp), %rdx
19    movq  %rdx, %rax
20    salq  $2, %rax
21    addq  %rdx, %rax
22    movq  %rax, -8(%rbp)
23    addq  %rcx, -8(%rbp)
24    movq  -8(%rbp), %rax
25    movq  %rax, %rsi
26    leaq  .LC0(%rip), %rdi
27    movl  $0, %eax
28    call  printf@PLT
29    movl  $0, %eax
30    leave
31    ret
32  .LFE0:
33    .size main, .-main
34    .ident  "GCC: (Ubuntu 10.5.0-1
          ubuntu1~22.04) 10.5.0"
35    .section  .note.GNU-stack,"",
          @progbits
36    .section  .note.gnu.property,"a"
37    .align 8
38    .long  1f - 0f
39    .long  4f - 1f
40    .long  5
41  0:
42    .string  "GNU"
43  1:
44    .align 8
45    .long  0xc0000002
46    .long  3f - 2f
47  2:
48    .long  0x3
49  3:
50    .align 8
51  4:
```

请结合汇编码说明现代编译器的代码生成策略。

注：第 11 行的 endbr64 指令和第 35~51 行分别是 x86-64 架构中与控制流保护相关的指令和 ELF 元信息，首次引入自 Intel 的控制流执行技术（control flow enforcement technology，CET）[①]，用于确保代码执行安全，防止控制流劫持攻击（如返回导向编程 ROP、跳转导向编程）。endbr64 指令用于标记间接跳转目标。第 35 行设置栈属性为不可执行 (对应""），第 36 行开始描述支持的 CET 特性。第 38 行定义的 4 字节常量为标号 1 和 0 的偏移差（表示第 42 行串的长度为 4 字节）；第 39 行存放标号 4 和 1 的偏移差，表示描述性数据的长度；第 40 行存放常量 5，表示 GNU 的特定属性类型；第 45 行存放的值 0xc0000002 是 GNU_PROPERTY_X86_FEATURE_1_IBT，用于表明目标文件支持 CET 的间接跳转跟踪（Indirect Branch Tracking, IBT）；第 48 行存放的常量 0x3 是 GNU_PROPERTY_X86_FEATURE_1_IBT | GNU_PROPERTY_x86_FEATURE_1_SHSTK，表示同时支持 IBT 和

① 2016 年 Intel 首次发布 CET 规范，2020 年首次在消费者处理器（Intel 第 11 代酷睿，Tiger Lake）中实现，它通过影子栈（shadow stack）、间接分支跟踪（IBT）这两种新机制来保护控制流。CET 的软件支持也始于 2020 年，分别在 Windows 10 2004 版和 Linux 内核 v5.7 中引入。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第339页 文件编号: 422624 页面编号: 33394538442 打印人: 8685 2026-03-09 10:23:04

Shadow Stack，即防止间接跳转被劫持和防止调用栈被篡改。这段信息会被连接器整合到最终生成的可执行文件或共享库中，用于描述文件的安全属性和优化支持。

8.8 一个 C 语言程序如下：

```
int main() {
    int i;
    i=50;
    switch(i*i){
        case 10: i=10; break;
        case 80: i=80; break;
        case 50: i=50; break;
        case 70: i=70; break;
        case 20: i=20; break;
        default: i=40;
    }
    switch(i*i){
        case 7: i=7; break;
        case 1: i=1; break;
        case 6: i=6; break;
        case 9: i=9; break;
        case 5: i=5; break;
        case 10: i=10; break;
        case 2: i=2; break;
        default: i=40;
    }
}
```

它在 x86/Linux 系统上编译后生成的汇编代码如下（编译器版本见汇编代码最后一行），请根据所生成的汇编代码写出程序中两条 switch 语句的目标代码结构的特点。

```
    .file   "ex8-8.c"
    .text
    .globl   main
    .type    main, @function
main:
    pushq    %rbp
    movq    %rsp, %rbp
    movl    $50, -4(%rbp)
    movl    -4(%rbp), %eax
    imull    %eax, %eax
    cmpl    $80, %eax
    je    .L2
    cmpl    $80, %eax
    jg    .L3
    cmpl    $70, %eax
    je    .L4
    cmpl    $70, %eax
    jg    .L3
    cmpl    $50, %eax
    je    .L5
    cmpl    $50, %eax
    jg    .L3
    cmpl    $10, %eax
    je    .L6
    cmpl    $20, %eax
    je    .L7
    jmp    .L3
.L6:
    movl    $10, -4(%rbp)
    jmp    .L8
.L2:
    movl    $80, -4(%rbp)
    jmp    .L8
.L5:
    movl    $50, -4(%rbp)
    jmp    .L8
.L4:
    movl    $70, -4(%rbp)

    jmp    .L8
.L7:
    movl    $20, -4(%rbp)
    jmp    .L8
.L3:
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第340页 文件编号: 422624 页面编号: 33404538440 打印人: 8685 2026-03-09 10:23:04


```
45      movl    $40, -4(%rbp)
46  .L8:
47      movl    -4(%rbp), %eax
48      imull    %eax, %eax
49      cmpl    $10, %eax
50      ja    .L9
51      movl    %eax, %eax
52      leaq    0(,%rax,4), %rdx
53      leaq    .L11(%rip), %rax
54      movl    (%rdx,%rax), %eax
55      cltq
56      leaq    .L11(%rip), %rdx
57      addq    %rdx, %rax
58      notrack jmp    *%rax
59      .section    .rodata
60      .align 4
61      .align 4
62  .L11:
63      .long    .L9-.L11
64      .long    .L17-.L11
65      .long    .L16-.L11
66      .long    .L9-.L11
67      .long    .L9-.L11
68      .long    .L15-.L11
69      .long    .L14-.L11
70      .long    .L13-.L11
71      .long    .L9-.L11
72      .long    .L12-.L11
73      .long    .L10-.L11
74      .text
75  .L13:
76      movl    $7, -4(%rbp)
77      jmp    .L18
78  .L17:
79      movl    $1, -4(%rbp)
80      jmp    .L18
81  .L14:
82      movl    $6, -4(%rbp)
83      jmp    .L18
84  .L12:
85      movl    $9, -4(%rbp)
86      jmp    .L18
87  .L15:
88      movl    $5, -4(%rbp)
89      jmp    .L18
90  .L10:
91      movl    $10, -4(%rbp)
92      jmp    .L18
93  .L16:
94      movl    $2, -4(%rbp)
95      jmp    .L18
96  .L9:
97      movl    $40, -4(%rbp)
98  .L18:
99      movl    $0, %eax
100     popq    %rbp
101     ret
102 .LFE0:
103     .size    main, .-main
104     .ident    "GCC: (Ubuntu 10.5.0-1
          ubuntu1~22.04) 10.5.0"
```

8.9 一个 C 语言程序如下：

```
1  extern int a;
2  static int b;
3  int c;
4  int main() {
5      b = a;
6  }
```

它在 x86/Linux 系统上编译后生成的汇编代码如下（编译器版本见汇编代码最后一行，略去了与题目不相关的内容），请说明编译时对 extern 变量的处理和外部变量的处理有什么区别。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第341页 文件编号: 422624 页面编号: 33414538448 打印人: 8685 2026-03-09 10:23:04


```
    .file   "ex8-9.c"
    .text
    .local   b
    .comm   b,4,4
    .globl   c
    .bss
    .align 4
    .type   c, @object
    .size   c, 4
c:
    .zero   4
    .text
    .globl   main
    .type   main, @function
main:
    pushq   %rbp
    movq   %rsp, %rbp
    movl   a(%rip), %eax
    movl   %eax, b(%rip)
    movl   $0, %eax
    popq   %rbp
    ret
.LFE0:
    .size   main, .-main
    .ident   "GCC: (Ubuntu 10.5.0-1
        ubuntu1~22.04) 10.5.0"
```

8.10 在 C 语言中，若 a 和 b 是相同的结构体类型，那么赋值 a=b 是可以的。但是编译器对这种赋值的实现方式可能与它们的字节数（size）有关。下面是两个 C 语言程序及其在 x86/Linux 系统上，经某版本的 GCC 编译器编译生成的汇编代码（略去与本题无关的部分）。请简要叙述这些汇编代码中体现出的实现方式上的区别。

第一个 C 语言程序及对应汇编代码：

```
struct {long i,j; double m,n;}a,b;
int main() {
    a=b;
}
```

```
main:
    pushq   %rbp
    movq   %rsp, %rbp
    movq   b(%rip), %rax
    movq   8+b(%rip), %rdx
    movq   %rax, a(%rip)
    movq   %rdx, 8+a(%rip)
    movq   16+b(%rip), %rax
    movq   %rax, 16+a(%rip)
    movl   $0, %eax
    popq   %rbp
    ret
```

第二个 C 语言程序及对应汇编代码：

```
struct {long i,j; double m,n;}a,b;
int main() {
    a=b;
}
```

```
main:
    pushq   %rbp
    movq   %rsp, %rbp
    movq   b(%rip), %rax
    movq   8+b(%rip), %rdx
    movq   %rax, a(%rip)
    movq   %rdx, 8+a(%rip)
    movq   16+b(%rip), %rax
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第342页 文件编号: 422624 页面编号: 33424538446 打印人: 8685 2026-03-09 10:23:04


```
9       movq    24+b(%rip), %rdx
10      movq    %rax, 16+a(%rip)
11      movq    %rdx, 24+a(%rip)
12      movl    $0, %eax
13      popq    %rbp
14      ret
```

8.11 在 C 语言中，若 a 和 b 是相同的结构体类型，那么赋值 a=b 是可以的。这句话对下面程序中出现的赋值 p = q 是否有效？编译器会报错吗？若编译器不报错，程序的输出是什么？

```
typedef struct{int n; float a[ ];} record;
record p = {2, {1.0, 2.0} }, q = {3, {1.0, 2.0, 3.0} };
main() {
    p = q;
    printf( “%d, %f, %f, %f\n” , p.n, p.a[0], p.a[1], p.a[2]);
    printf( “%d, %f, %f, %f\n” , q.n, q.a[0], q.a[1], q.a[2]);
}
```

第 9 章

独立于机器的代码优化

如果简单地把高级语言的每个构造独立地翻译成机器代码，这种方式会造成运行开销较大。在编译过程中，通过删除目标代码中不必要的指令，用速度较快并完成同样事情的代码序列代替较慢的代码序列等变换来降低运行开销，称为**代码改进**或**代码优化**。

第 8 章已经介绍了一些在基本块内的局部优化，本章介绍全局代码优化。它的信息收集和代码改进变换都不局限于一个基本块。大多数全局优化基于**数据流分析**。数据流分析由一些收集程序信息的算法构成，其结果形式相同：对程序中的每条指令，它们描述该指令每次执行时程序必须保持的性质。不同的分析计算不同的性质。例如，常量传播分析用于确定在每个程序点上，每个变量的值是否保持不变。活跃变量分析则用于判断在每个程序点上，各变量的当前值在下次引用之前是否会被覆盖。若是，则在该程序点，无需在寄存器或内存中保留这些值。

9.1 节通过一个实例来介绍代码改进的一些主要机会。9.2 节介绍的数据流分析包括几个重要的实例，它们从全局收集一些重要信息，随后将其用于代码改进变换。9.3 节介绍数据流分析的理论基础和一般框架，9.2 节所介绍的几种数据流分析都是该框架的实例，这些数据流分析用的算法本质上相同，因而可以用同样的方式来评估分析的性能和证明其正确性。9.4 节介绍一种在此框架上能力较强的数据流分析。9.5 节介绍部分冗余删除（partial redundancy elimination），它是能力更强的优化技术，用于确定程序中计算每个表达式的恰当位置。这个问题的解决需要得出多种不同数据流问题的解。9.6 节讨论程序中循环的识别和分析。循环的识别引出了求解数据流问题的另一类算法，它基于可归约程序的循环层次结构。

9.1 优化的主要种类

编译器的优化必须保证程序的语义不变。除了非常特殊的情况，对于程序员选择并实现的算法，编译器不可能用另一个本质上不同的更有效算法来代替。编译器只知道如何实施一些相对低级的语义变换，如利用像代数恒等 $i+0=i$ 这样的一般事实，或者利用相同的运算作用于相同的值得到相同的结果这样的程序语义。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第344页 文件编号: 422624 页面编号: 33444538442 打印人: 8685 2026-03-09 10:23:04

9.1.1 优化的主要源头

程序中常存在许多冗余计算。有时，这些冗余在源代码级就很明显。程序员为了方便，可能会重复某些计算，期待编译器优化它们。但更常见的是高级语言的特性导致的冗余。例如，在大多数编程语言中（不包括允许指针算术的 C 和 C++ 语言），访问数组元素或结构体字段时，如 A[i][j] 和 X.f1，程序员只能通过高层次方式来操作。但在编译时，这些访问会展开为多步低级算术运算，如数组 A 第 (i, j) 个元素的地址计算。在一个程序中，对同一数据结构的多次访问往往会导致许多相同的低级运算，程序员无法手动消除这些冗余。

事实上，从软件工程的视角上看，限制程序员只通过变量名访问数据结构，更有利于代码的编写、理解和维护。编译器负责优化这些冗余运算，从而兼顾代码的效率与可维护性。

9.1.2 一个实例

下面用快速排序程序 quickSort 来介绍公共子表达式删除、复写传播、死代码删除、强度削弱和归纳变量删除等几种重要的代码改进变换。图 9.1 是快速排序的 C 语言代码。a[0] 和 a[max] 分别是被排序的最小元素和最大元素。

```
1  void quickSort(int m, int n) {
2      int i, j;
3      int v, x;
4      if (n <= m) return;  // 递归结束条件
5
6      // 快速排序核心部分
7      i = m - 1; j = n; v = a[n];
8      while (1) {
9          do   i = i + 1; while (a[i] < v);
10         do   j = j - 1; while (a[j] > v);
11         if (i >= j) break;
12         x = a[i]; a[i] = a[j]; a[j] = x; /* 交换a[i]和a[j] */
13     }
14     x = a[i];  a[i] = a[n];  a[n] = x;   /* 交换a[i]和a[n] */
15
16     quickSort(m, j);   quickSort(i + 1, n);
17 }
```

图 9.1 快速排序的 C 语言代码

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第345页 文件编号: 422624 页面编号: 33454538449 打印人: 8685 2026-03-09 10:23:04

在通过优化删除图 9.1 程序中冗余的地址计算前，首先将这些地址计算分解成低级的算术运算以揭示其中的冗余。本章剩余部分假定采用三地址语句方式的中间表示，其中临时变量用来保存所有中间表达式的结果。图 9.2 所示为图 9.1 中被标注程序段产生的中间代码。

(1) i=m-1
(2) j=n
(3) t_1=4*n
(4) v=a[t_1]
(5) i=i+1
(6) t_2=4*i
(7) t_3=a[t_2]
(8) if t_3<v goto(5)
(9) j=j-1
(10) t_4=4*j
(11) t_5=a[t_4]
(12) if t_5>v goto(9)
(13) if i>=j goto(23)
(14) t_6=4*i
(15) x=a[t_6]
(16) t_7=4*i
(17) t_8=4*j
(18) t_9=a[t_8]
(19) a[t_7]=t_9
(20) t_{10}=4*j
(21) a[t_{10}]=x
(22) goto(5)
(23) t_{11}=4*i
(24) x=a[t_{11}]
(25) t_{12}=4*i
(26) t_{13}=4*n
(27) t_{14}=a[t_{13}]
(28) a[t_{12}]=t_{14}
(29) t_{15}=4*n
(30) a[t_{15}]=x

图 9.2 图 9.1 部分程序的三地址代码

假定每个整数占 4 字节。赋值 x = a[i] 按 7.6.3 小节的方式被翻译成两条三地址语句，见图 9.2 中的第 14 和 15 行：

t_6 = 4 * i
x = a[t_6]

类似地，a[j] = x 也被翻译成两条三地址语句，见图 9.2 中的第 20 和 21 行：

t_{10} = 4 * j
a[t_{10}] = x

正因为源程序中每个数组访问都被翻译成两步（下标乘以 4 和数组索引操作），这段较短的源代码被翻译成相当长的三地址语句序列。

图 9.3 是图 9.2 程序的流图，块 B_1 是初始结点。程序所有的条件转移和无条件转移在图 9.3 中都改成转移到目标语句所在的基本块。图 9.3 的流图有 3 个循环：块 B_2 和 B_3 分别单独构成一个循环；块 $B_2 \sim B_5$ 一起形成一个循环，入口结点是块 B_2。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第346页 文件编号: 422624 页面编号: 33464538447 打印人: 8685 2026-03-09 10:23:04

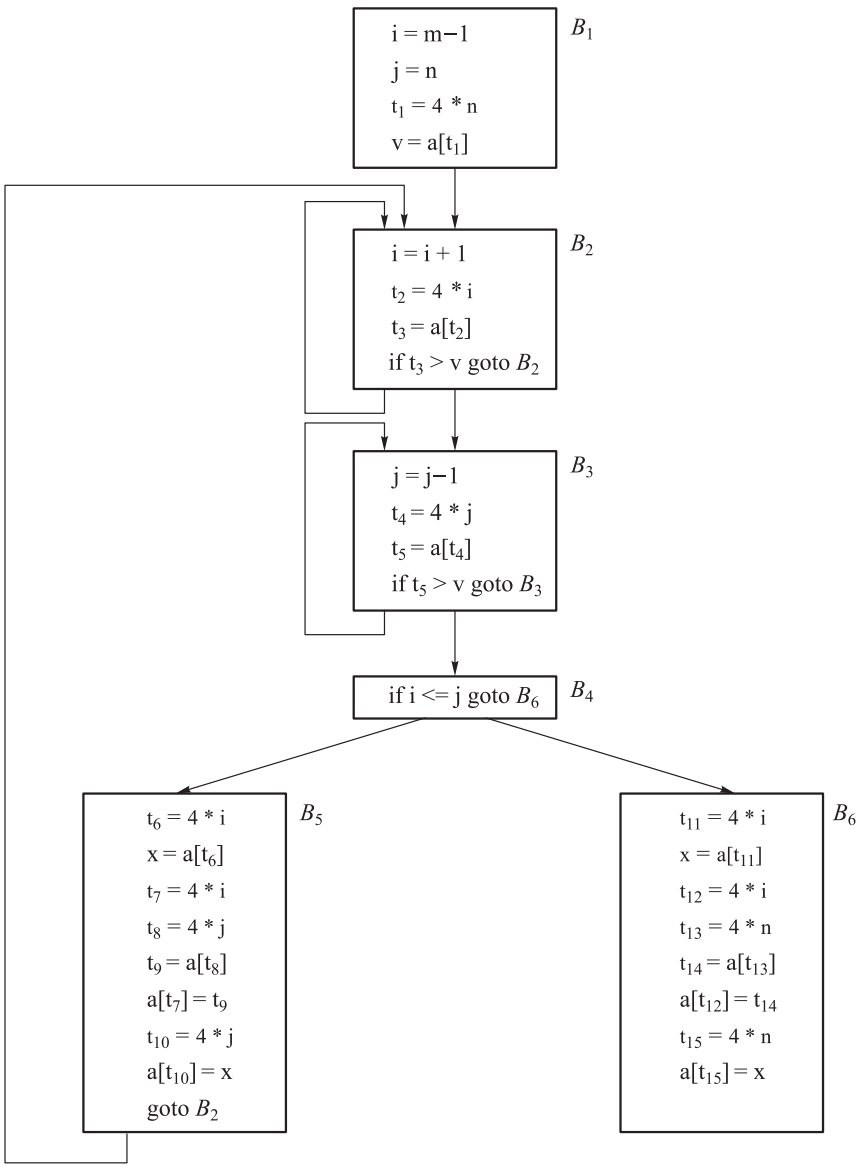


图 9.3 图 9.2 程序的流图

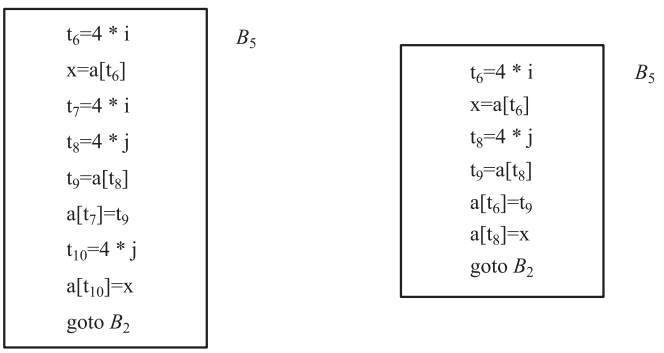
9.1.3 公共子表达式删除

如果表达式 E 先前已计算，并且从先前的计算到 E 的再次出现，E 中变量的值没有改变，那么 E 的再次出现称为**公共子表达式**。利用先前的计算结果可以避免重复计算表达式。

【例 9.1】 在图 9.3 的基本块 B_5 中（见图 9.4(a)），对 t_7 和 t_{10} 赋值的语句分别有公共

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第347页 文件编号: 422624 页面编号: 33474538445 打印人: 8685 2026-03-09 10:23:04

子表达式 4 * i 和 4 * j 出现在它们的右部。用 t_6 代替 t_7，用 t_8 代替 t_{10}，这些公共子表达式得以删除，删除后该基本块的代码如图 9.4(b) 所示。这是仅考察基本块内部可完成的公共子表达式的删除。



(a) 删除前　　(b) 删除后

图 9.4　局部公共子表达式删除

【例 9.2】 在图 9.3 中，块 B_5 和块 B_6 中全局公共子表达式和局部公共子表达式删除后的结果在图 9.5 中给出。这里重点讨论块 B_5 的变换。

删除了局部公共子表达式后，B_5 仍然计算 4 * i 和 4 * j，从全局看，它们仍然是公共子表达式。B_5 中的三条语句：

$t_8 = 4 * j$　　$t_9 = a[t_8]$　　$a[t_8] = x$

可以由

$t_9 = a[t_4]$　　$a[t_4] = x$

代替。因为从图 9.5 中可以看到，t_4 在 B_3 中计算，当控制从 B_3 中 4 * j 的计算流到 B_5 时，中间没有改变 j 的值，所以在 B_5 中需要 4 * j 的值时可以引用 t_4。

t_4 代替 t_8 后，B_5 的另一个公共子表达式变得清楚了，即 $a[t_4]$，对应于源代码中的 a[j]。不仅控制离开 B_3 和进入 B_5 时 j 的值没有变，而且 a[j] 的值不变（a[j] 的值计算在临时变量 t_5 中），因为在这段区间中没有对 a 的元素赋值。这样，B_5 的语句

$t_9 = a[t_4]$　　$a[t_6] = t_9$

可以由 $a[t_6] = t_5$ 代替。

类似地，图 9.4(b) 的 B_5 中对 x 赋的值和 B_2 中对 t_3 赋的值一样。删掉图 9.4(b) 中对应到源代码表达式 a[i] 和 a[j] 的公共子表达式后，其结果就是图 9.5 中的 B_5。

图 9.5 中的 B_6 也是完成了一系列类似变换后的结果。图 9.5 中的 B_1 和 B_6 的表达式 $a[t_1]$ 不能看作公共子表达式，虽然 t_1 在两个地方都使用，但是因为控制离开 B_1 进入 B_6 之前，它可以通过 B_5，B_5 有对 a 的赋值，因此在到达 B_6 时，$a[t_1]$ 的值可能和离开 B_1 时的值不一样，把 $a[t_1]$ 作为公共子表达式是不稳妥的。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第348页 文件编号: 422624 页面编号: 33484538443 打印人: 8685 2026-03-09 10:23:04

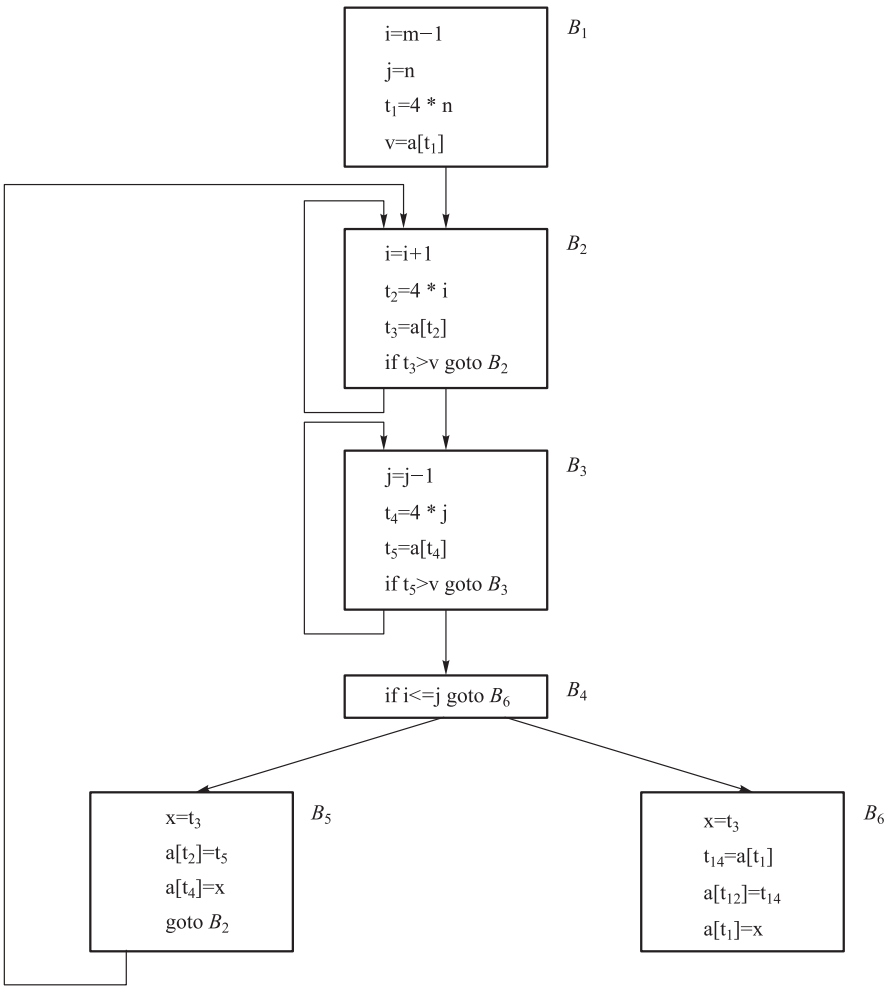


图 9.5 删除公共子表达式后的块 B_5、B_6

9.1.4 复写传播

图 9.5 中的块 B_5 可以通过两种新的变换来删除 x，实现进一步化简。一种变换是 9.1.5 节介绍的死代码删除；另一种变换与形式为 f = g 的赋值有关，这种赋值称为**复写语句**，简称**复写**。复写传播变换是指在复写语句 f = g 的后面，尽可能用 g 代表 f。例 9.2 其实已经用到了复写，不仅删除公共子表达式会引入复写，而且利用复写传播会得到新的公共子表达式。某些场合下的删除公共子表达式可能会引入更多的复写。例如，当删除图 9.6 中的公共子表达式 c = d + e 时，需要使用新的变量 t 来保存 d + e 的值。因为控制到达 c = d + e 可能会在对 a 的赋值之后，也可能会在对 b 的赋值之后，因此用 c = a 或 c = b 来代替 c = d

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第349页 文件编号: 422624 页面编号: 33494538441 打印人: 8685 2026-03-09 10:23:04

+ e 都是不稳妥的。

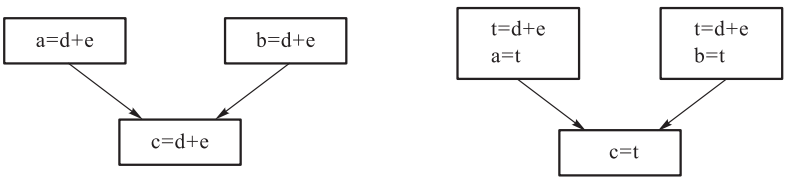


图 9.6 删除局部公共子表达式时引入复写

图 9.5 中块 B_5 的赋值 $x = t_3$ 是一个复写。把复写传播运用于块 B_5 后，它的 4 条语句为

$$\begin{aligned} &x = t_3 \\ &a[t_2] = t_5 \\ &a[t_4] = t_3 \\ &\text{goto } B_2 \end{aligned} \tag{9.1}$$

这看起来似乎没有改进，但在 9.1.5 小节会看到，它增加了删除对 x 赋值的机会。

复写传播还可以引入一些其他的优化机会。若在编译时能推断出一个表达式的所有运算对象都是常量，则可以在编译时完成该表达式的计算，并用计算结果代替该表达式，这种变换称为**常量合并**。复写传播可能会引入一些常量合并的机会。9.1.2 小节的例子没有可以实施常量合并的地方。

9.1.5 死代码删除

如果一个变量的值以后还要引用，则称它在该程序点是活跃的，否则它在该点已死亡。**死代码**或**无用代码**是指计算的结果不被引用的语句，删除它们不会影响程序的语义。虽然程序员一般不会引入死代码，但是上面介绍的变换可能会引入死代码。事实上，例 9.2 已经使用了死代码删除。

复写传播的优点之一是经常导致复写语句成为死代码。语句式（9.1）中的代码可以进一步优化，删掉语句式（9.1）中的死代码 $x = t_3$，即变成

$$\begin{aligned} &a[t_2] = t_5 \\ &a[t_4] = t_3 \\ &\text{goto } B_2 \end{aligned}$$

这段代码是对图 9.5 中块 B_5 的进一步改进。

有时程序员也会利用优化编译器删除死代码的特点来方便自己调试程序。例如，在程序中增加像

```
if (debug) print ...
```
(9.2)

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第350页 文件编号: 422624 页面编号: 33504538449 打印人: 8685 2026-03-09 10:23:04

这样的条件打印语句，当调试结束时并不把它们从程序中删除，而是将程序开头的

```
debug = TRUE
```

改成

```
debug = FALSE
```

当复写传播用 FALSE 代替 debug 时，条件打印语句的条件为假，打印语句就成了死代码，条件判断和打印语句都可以从目标代码中删除。

9.1.6 代码外提

本节和 9.1.7 小节简单介绍一个非常重要且值得优化的地方——循环，尤其是消耗程序运行大部分时间的内循环。如果内循环的指令数量得以减少，即使外循环的指令数增加，程序的运行时间仍可能缩短。减少循环中代码数量的一种重要办法是**代码外提**。它把**循环不变计算**，即运算结果独立于循环体执行次数的表达式，放到循环之前计算。例如，语句

```
while (i <= limit - 2 ) { /* 不改变 limit 的语句 */ }
```

中 limit - 2 是循环不变计算。代码外提的结果是

```
t = limit - 2;
while (i <= t ) { /* 不改变 limit 的语句 */ }
```

在 9.1.2 小节的实例中没有可以实施代码外提的地方。

9.1.7 强度削弱和归纳变量删除

循环的另一种优化是寻找循环中的归纳变量，并优化对它们的计算。变量 x 称为归纳变量，如果存在一个正或负的常量 c，使得每次对 x 赋值时，它的值实质上增加 c。例如，i 和 t_2 是图 9.5 中包含块 B_2 的循环中的归纳变量。对 t_2 来说，虽然语句是 $t_2 = 4 * i$，但实际上 t_2 的值是在原来基础上加 4。在每次循环迭代时，归纳变量可以用加或减一个常量来计算。用一个开销较小的运算代替一个开销较大的运算称为**强度削弱**，如用加代替乘。

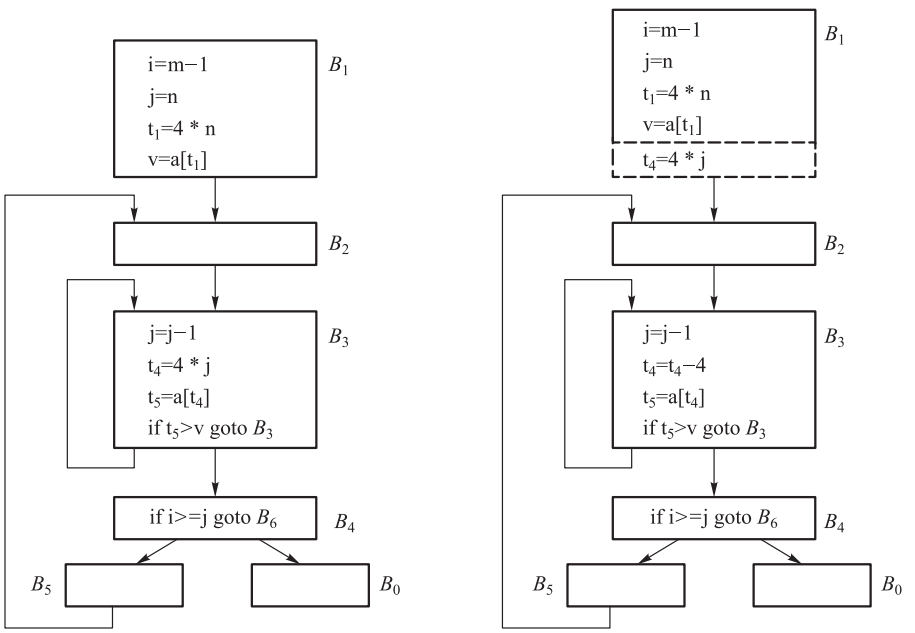
对一组在循环中的归纳变量如果它们的值变化一致，可以仅保留其中一个归纳变量，删除其余的归纳变量。

在处理循环时，可以“从里向外”，即从最内层循环开始，沿直接外围循环逐层向外。把这种思想用于快速排序程序，从块 B_3 构成的最内层循环开始来进行优化。从图 9.5 中可以看出，j 和 t_4 形成一组归纳变量，因为它们的值变化一致，每次 j 的值减 1 后，4 * j 赋给 t_4，t_4 的值实质上减 4。但是仅看由块 B_3 构成的内循环，j 和 t_4 都不能删去，因为 t_4 在块 B_5 中引用，j 在块 B_4 中引用。然而，可以用它们来举例说明强度削弱，而这个强度削弱又为删除归纳变量创造了机会。最终，当考察由块 $B_2 \sim B_5$ 构成的外循环时，j 可以被删除。

【例 9.3】 在图 9.7(a) 中，对由块 B_3 构成的内循环，若不考虑第一次进入块 B_3，关系 $t_4 = 4 * j$ 在块 B_3 的入口一定保持，在 $j = j - 1$ 后，关系 $t_4 = 4 * j + 4$（$4 * j = t_4 -$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第351页 文件编号: 422624 页面编号: 33514538447 打印人: 8685 2026-03-09 10:23:04

4）也保持，那么 $t_4 = 4 * j$ 可以用 $t_4 = t_4 - 4$ 代替。要进行这个变换的唯一问题是第一次进块 B_3 时 t_4 没有初值，所以在给 j 置初值的那个基本块末尾添加对 t_4 置初值 $4 * j$ 的语句。在图 9.7(b) 中，这个语句放在块 B_1 的最后。在许多机器上，乘运算比加或减运算需要更多时间，这种变换会加快目标代码的运行速度，虽然在块 B_1 的最后增加了一个语句。



(a) 变换前　　(b) 变换后

图 9.7　强度削弱用于块 B_3 的 $4 * j$

下面再举一个删除归纳变量的例子来作为本节的结束，该例在外循环块 $B_2 \sim B_5$ 的上下文中处理 i 和 j。

【例 9.4】　把强度削弱用于块 B_2、B_3 的内循环后，i 和 j 的作用仅在于决定块 B_4 的测试结果。由于 j 和 t_4 满足关系 $t_4 = 4 * j$、i 和 t_2 满足关系 $t_2 = 4 * i$，测试 $t_2 >= t_4$ 等价于 i >= j。一旦作出这种替换，块 B_2 的 i 和块 B_3 的 j 就成了死变量，在这些块中对它们的赋值也就成了死代码，可以删除，最终结果在图 9.8 中给出。

本节快速排序程序的代码改进变换效果明显。在图 9.8 中，块 B_2、B_3 的指令数都从图 9.3 所示的最初流图的 4 条减到 3 条，块 B_5 从 9 条减到 3 条，块 B_6 从 8 条减到 3 条。虽然块 B_1 从 4 条增加到 6 条，但它在这段程序中仅执行一次，所以总的运行时间几乎不受块 B_1 大小的影响。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第352页 文件编号: 422624 页面编号: 33524538445 打印人: 8685 2026-03-09 10:23:04

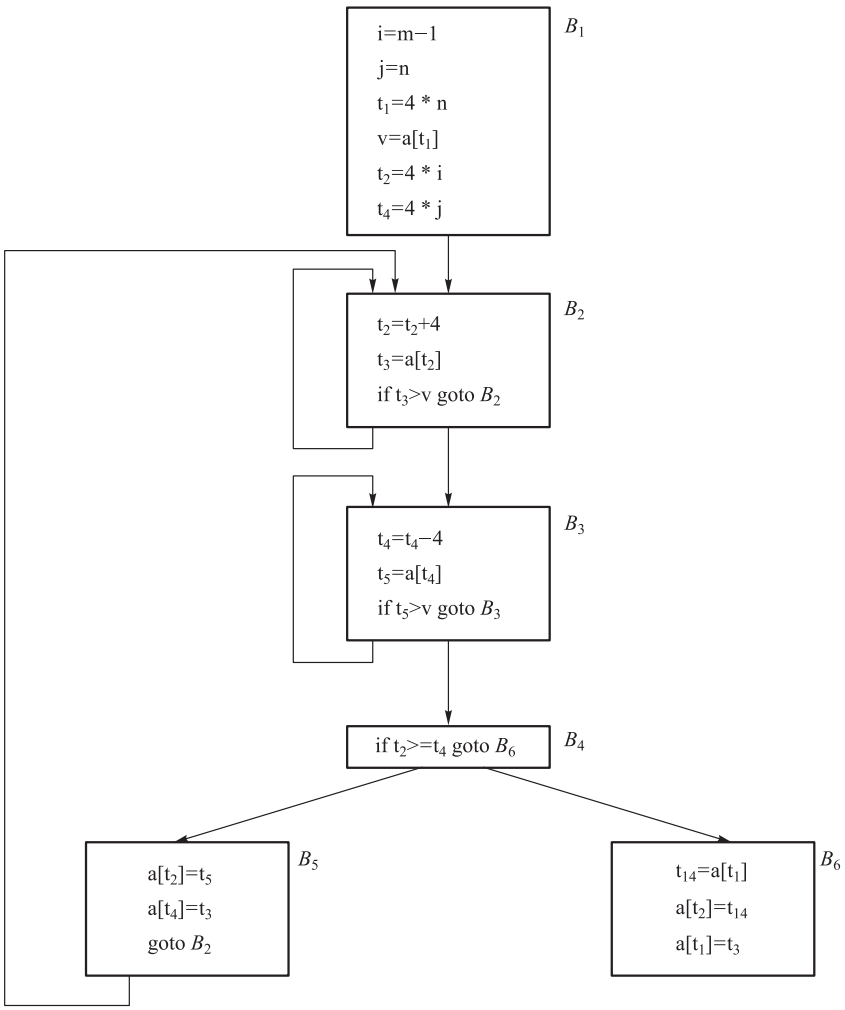


图 9.8 删除归纳变量后的流图

9.2 数据流分析介绍

9.1 节介绍的所有优化都依赖于**数据流分析**。数据流分析推导的是数据沿着程序执行路径流动的信息。例如，实现全局公共子表达式删除的方式需要判断，在程序的任何执行路径上，两个正文上完全一样的表达式计算得到相同的值。此外，如果一个赋值的结果在随后的任何执行路径上都不使用，则可以把该赋值作为死代码删除。这类问题都可以通过数据流分析来解决。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第353页 文件编号: 422624 页面编号: 33534538443 打印人: 8685 2026-03-09 10:23:04

9.2.1 数据流抽象

根据 6.1.2 小节的介绍可知，程序状态由程序所有单元的值组成，包括运行栈的栈顶活动记录之下的那些活动记录中的值。程序的执行可以看成是在程序状态上的一系列变换。每个三地址语句的执行将一个输入状态转换为一个输出状态。输入状态和输出状态分别联系到该语句前后的那两个**程序点**。

在分析程序的行为时，必须在其流图上考虑该程序所有可能的执行路径（程序点序列），然后从每个程序点的可能状态中抽取解决特定数据流分析所需要的信息。在更复杂的分析中，如调用语句或返回语句被执行时，还需要考虑路径在多个流图之间的跳转。本章作为对优化的初步介绍，只集中在单个流图（对应一个过程）的路径上。

下面介绍从一个流图可以得到的执行路径。先看点之间的一些特征：

（1）在一个基本块内，一条语句之后的程序点就是下一条语句之前的程序点。

（2）如果有一条边从块 B_1 到块 B_2，那么块 B_1 最后一条语句之后的那个程序点（块 B_1 的出口点）可以由块 B_2 的第一条语句之前的那个程序点（块 B_2 的入口点）直接跟随。

这样，从点 p_1 到点 p_n 的**执行路径**（简称**路径**）定义为点序列 $p_1, p_2, \cdots, p_n$，并且每个 $i = 1, 2, \cdots, n-1$ 满足：p_i 是一条语句前的那个点，p_{i+1} 是该条语句后的那个点；或者 p_i 是某块的出口点，p_{i+1} 是该块的一个后继块的入口点。

通常来说，从一个流图得到的程序执行路径的数目无限，且不存在执行路径长度的有限上界。程序分析从一个程序点的所有可能状态中为该点总结出一组有限的事实，不同的程序分析可能提炼出不同的信息。程序分析收集的信息不一定是状态的完整描绘。

【例 9.5】 即使是图 9.9 这样简单的程序，它的执行路径数也是无穷多的。不进入循环且最短的完整执行路径由程序点（1, 2, 3, 4, 9）组成。下一条最短路径执行循环的一次迭代，它由程序点（1, 2, 3, 4, 5, 6, 7, 8, 3, 4, 9）组成。例如，第一次到达点 (5) 时，a 的值是 1，它是由 d_1 定值的。因此第一次迭代时，d_1 到达点 (5)。在随后的迭代中，d_3 到达点 (5)，a 的

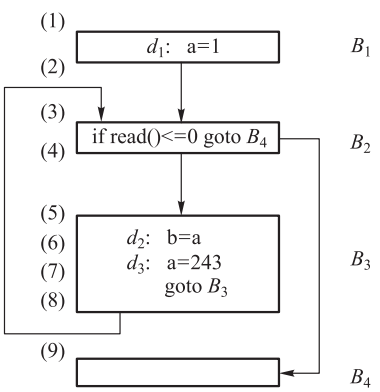


图 9.9 阐明数据流抽象的例子

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第354页 文件编号: 422624 页面编号: 33544538441 打印人: 8685 2026-03-09 10:23:04

值是 243。

通常，不可能追踪所有路径上的所有程序状态。数据流分析虽然无法区分不同路径上的程序状态或掌握完整的状态信息，但是能从中提取出特定的细节，用于进一步地优化和分析。下面通过两个例子说明怎样从一个点的状态中提炼出不同的信息。

（1）为调试程序，需要找出一个变量在某个程序点的所有可能取值，以及它们是在何处定值的。例如，对图 9.9 中点 (5) 的所有程序状态可以这样总结：a 的值是 {1, 243} 中的一个，并且由 $\{d_1, d_3\}$ 中的一条语句定值。沿着某条路径可以到达一个程序点的定值称为该点的**到达–定值**。

（2）考虑常量折叠（constant folding）的实现。如果某个变量 x 的一个引用仅能由它的一个定值到达，并且该定值赋 x 给一个常量，那么可以直接用这个常量来代替 x 的这个引用。但是，如果 x 的多个定值都可能到达一个程序点，就不可能对 x 执行常量折叠。因此，对于常量折叠问题，希望的是找到这样的定值，它们是相应变量到达一个程序点的唯一定值，而不管该定值经过什么样的路径。例如，对于图 9.9 中的点 (5)，不存在 a 的某个定值，它是到达该点的 a 的唯一定值。即使某变量在某点有唯一定值，该定值还必须给这个变量赋常量才能用于常量折叠。因此，对于图 9.9 中的点 (5)，可以简单地描述 a “不是常量”，而不是去收集所有可能的值或所有可能的定值。

这两个例子表明，从相同的信息可以提炼出不同的总结，具体取决于程序分析的目的。

9.2.2 数据流分析模式

在数据流分析的每种应用中，总把每个程序点和一个**数据流值**联系起来，数据流值代表在该点能观测到的所有可能程序状态集合的一个抽象。可能数据流值的集合称为该应用的**论域**。例如，对于到达–定值，数据流值的论域就是由该程序中所有定值组成集合的幂集，其中一种具体的应用是，联系到每个程序点的数据流值正好是能够到达该点的定值集合。如前文所叙，信息提炼的选择依赖于数据流分析的目的；为了使分析高效，数据流分析只跟踪记录相关的信息。

每条语句 s 前后两个点的数据流值分别用 IN[s] 和 OUT[s] 来表示。**数据流问题**就是对所有语句 s 的 IN[s] 和 OUT[s] 上的约束寻找一个解。这些约束分成两组，分别是基于语句语义的约束（迁移函数）和基于控制流的约束。

（1）迁移函数。一条语句前后两点的数据流值受该条语句的语义约束。例如，假定数据流分析是用来确定变量在程序点的值。如果变量 a 在语句 b = a 执行前有值 v，那么该语句执行后 a 和 b 的值都是 v。该赋值语句前后两点的数据流值之间的联系称为**迁移函数**。

迁移函数有两种风格：信息沿执行路径正向传播或逆向传播。在一个正向流问题中，一条语句 s 的迁移函数（通常用 $f_s()$ 来表示）以该语句前的那个点的数据流值为变元，产生该语句后那个点的数据流值，即

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第355页 文件编号: 422624 页面编号: 33554538448 打印人: 8685 2026-03-09 10:23:04

$$\mathrm{OUT}[s] = f_s(\mathrm{IN}[s])$$

反过来，在一个逆向流问题中，迁移函数以该语句后的那个点的数据流值为变元，产生该语句前的那个点的数据流值，即

$$\mathrm{IN}[s] = f_s(\mathrm{OUT}[s])$$

（2）控制流约束。数据流值上的第二组约束从控制流推导出来。以正向流问题为例，在基本块内部，控制流是简单的。如果基本块 B 由语句 $s_1, s_2, \cdots, s_n$ 依次组成，那么一定有

$$\mathrm{IN}[s_i + 1] = \mathrm{OUT}[s_i], \quad i = 1, 2, \cdots, n-1$$

然而，基本块之间的控制流边引出在基本块出口点和后继基本块入口点之间更加复杂的约束。例如，如果收集到达程序点的所有定值，那么到达基本块入口点的定值集合，是到达该块各前驱块出口点的定值集合的并集。下面更仔细地讨论数据在基本块之间的流动。

由于基本块内的控制流较简单，没有中断、分支等，因此可以将基本块内各语句的迁移函数复合起来，形成基本块的迁移函数。仍以正向流问题为例，若基本块 B 由语句 $s_1, s_2, \cdots, s_n$ 依次组成，则基本块 B 的迁移函数 f_B 为（逆向流问题的 f_B 可类似地得到）

$$f_B = f_{s_n} \circ \cdots \circ f_{s_2} \circ f_{s_1}$$

在建立控制流约束时不用关心基本块内部的情况，称 $\mathrm{IN}[B]$（$=\mathrm{IN}[s_1]$）和 $\mathrm{OUT}[B]$（$=\mathrm{OUT}[s_n]$）分别为基本块 B 入口和出口的数据流值（因而也不区分块入口和块入口点概念，对块出口也是这样）。这样，基本块的出口和入口之间数据流值的联系可以用下面的等式表达

$$\mathrm{OUT}[B] = f_B\ (\mathrm{IN}[B])$$

现在可以来写控制流约束了。例如，如果数据流值是赋给某个变量的常量集合，那么该正向流问题的控制流约束可以用等式表示，即

$$\mathrm{IN}[B] = \cup_{P\text{是}B\text{的前驱}}\mathrm{OUT}[P]$$

对于逆向流问题，如活跃变量分析，其等式类似，但是 IN 和 OUT 的作用颠倒，即

$$\mathrm{IN}[B] = f_B\ (\mathrm{OUT}[B])$$

$$\mathrm{OUT}[B] = \cup_{S\text{是}B\text{的后继}}\mathrm{IN}[S]$$

和线性算术方程不同，由上述数据流等式形成的方程组通常解不唯一。求解的目标是要找到满足这两组约束（控制流约束和迁移约束）的最“精确”解。也就是需要一种能够鼓励合法代码改进的解，而非可能引入不安全变换（改变程序计算结果）的解。以下几小节将讨论一些可以通过数据流分析解决的重要问题。

9.2.3 到达–定值

到达–定值是一种最常用的数据流模式。如果控制到达每个程序点 p 时，能够知道每个变量 x 是在程序的哪些地方被定值的，就可以确定关于 x 的许多事情。在两个例子中，编译器可以知道 x 在点 p 是否为常量，调试器可以告知 x 在点 p 是否为没有定值的变量。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第356页 文件编号: 422624 页面编号: 33564538446 打印人: 8685 2026-03-09 10:23:04

如果存在从对 x 的定值 d 之后那个点到点 p 的一条路径，并且在这条路径上没有对 x 的定值，那么称定值 d 到达点 p。如果在这条路径上其他某个地方有对 x 的定值，那么称变量 x 在 d 的定值被**注销**。从直观上说，如果某个变量 x 在 d 的定值到达点 p，并且运行时在点 p 引用 x，那么 d 可能是 x 最近一次定值的位置。

在有别名的情况下，变量 x 的定值是一条语句，它给 x 赋值或可能给 x 赋值。因为过程参数、数组访问、间接引用等都可能引起别名，所以很难确定一条语句是否联系到某个特定变量 x。程序分析必须是稳妥的，如果不能确定语句 s 是否给变量 x 赋值，那么必须认为它可能给 x 赋值。也就是说在语句 s 之后，x 的值可能是原来的，也可能是语句 s 赋给它的新值。为简单起见，本章其余部分仅考虑变量无别名的情况，即只考虑大多数语言都有的局部标量，并且排除 C 和 C++ 语言能够获取局部变量地址的情况。

【例 9.6】 图 9.10 是有 7 个定值的流图。重点考察到达块 B_2 的定值。块 B_1 的所有定值都可以到达块 B_2 的入口。块 B_2 的 d_5: j = j − 1 也可以到达块 B_2 的入口，因为在回到块 B_2 的循环中不存在其他对 j 的定值。然而，这个定值注销了定值 d_2: j = n，阻止它到达块 B_3 或块 B_4。语句 d_4: i = i + 1 不能到达块 B_2 的入口，因为变量 i 总是在 d_7: i = u3 被重新定值。最后，定值 d_6: a = u2 总能到达块 B_2 的入口。

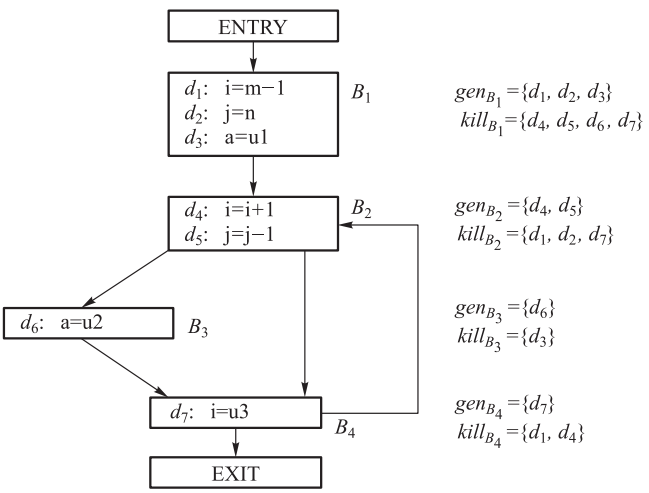


图 9.10 说明到达–定值的流图

从到达–定值的定义可以看出，数据流分析有时是不精确的。例如，到达–定值的定义基于流图中的每条边都会经过，这个假设与实际情况有出入。例如，无论 a 和 b 取何值，都不能使控制流真正到达下面程序段的 statement2，即流图中的一条执行路径并不一定是运行时的执行路径。

```
if (a == b) statement1; else if (a == b) statement2;
```

一般而言，流图中每条路径是否为实际运行路径是不可判定的。因此，数据流分析简单

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第357页 文件编号: 422624 页面编号: 33574538444 打印人: 8685 2026-03-09 10:23:04

地假设流图中的每条路径在程序的某次执行中总会被经过。这个假设是稳妥的，因为不会导致改变程序计算结果的变换，但是有可能错过能够保持程序语义的优化。例如，在到达–定值分析中，即使某个定值假设到达了某个点而实际上未到达，这种假设依然是稳妥的。当到达–定值信息用于常量折叠时，这种假设可能导致某些常量未被折叠，但不会引入改变程序含义的错误变换。

现在开始建立到达–定值问题的约束等式，首先是迁移等式。首先考察语句，考虑定值

$$d\colon \mathrm{u} = \mathrm{v} + \mathrm{w}$$

其中，“+” 经常作为二元运算符的代表用于下面的讨论。

该语句产生变量 u 的一个定值 d，注销程序中 u 的所有其他定值，并且维持其他变量的定值没有变化。因此定值 d 的迁移函数可以表达为

$$f_d(x) = \mathrm{gen}_d \cup (x - \mathrm{kill}_d) \tag{9.3}$$

其中，$\mathrm{gen}_d = \{d\}$，即由该语句产生的定值集合；kill_d 是该程序中 u 所有其他定值的集合。

如 9.2.2 节所讨论的那样，一个基本块的迁移函数由该块中各语句的迁移函数复合而成。下面可以看到，表达式 (9.3) 形式的函数的复合结果仍然是这种形式，称这种形式为 “产生–注销形式”。假定有两个函数 $f_1(x) = \mathrm{gen}_1 \cup (x - \mathrm{kill}_1)$ 和 $f_2(x) = \mathrm{gen}_2 \cup (x - \mathrm{kill}_2)$，那么

$$\begin{aligned} f_2(f_1(x)) &= \mathrm{gen}_2 \cup (\mathrm{gen}_1 \cup (x - \mathrm{kill}_1) - \mathrm{kill}_2) \\ &= (\mathrm{gen}_2 \cup (\mathrm{gen}_1 - \mathrm{kill}_2)) \cup (x - (\mathrm{kill}_1 \cup \mathrm{kill}_2)) \end{aligned}$$

这个规则可以延伸到一个块包含任意数目的语句上。假定块 B 有 n 条语句，它们的迁移函数分别是 $f_i(x) = \mathrm{gen}_i \cup (x - \mathrm{kill}_i)(i = 1, 2, \cdots, n)$，那么块 B 的迁移函数可以写成

$$fB(x) = \mathrm{gen}_B \cup (x - \mathrm{kill}_B)$$

其中

$$\mathrm{kill}_B = \mathrm{kill}_1 \cup \mathrm{kill}_2 \cup \cdots \cup \mathrm{kill}_n$$

并且

$$\begin{aligned} \mathrm{gen}_B = {} & \mathrm{gen}_n \cup (\mathrm{gen}_{n-1} - \mathrm{kill}_n) \cup (\mathrm{gen}_{n-2} - \mathrm{kill}_{n-1} - \mathrm{kill}_n) \cup \cdots \\ & \cup (\mathrm{gen}_1 - \mathrm{kill}_2 - \mathrm{kill}_3 - \cdots - \mathrm{kill}_n) \end{aligned}$$

于是，和一条语句类似，一个块也是产生一组定值并且注销一组定值。gen 集合包含该块中所有在该块出口 “可见” 的定值集合，称它们为**向下暴露**的定值。一个定值在块中向下暴露，是指它没有被该块中随后的定值注销。一个基本块的 kill 集合直接就是该块中各语句注销集合的并集。注意，一个定值 d 可能同时出现在一个块 B 的 gen_B 和 kill_B 中。如果是这样，定值 d 在 gen_B 中的事实优先，因为在产生–注销形式中，减去 kill_B 的运算先于和 gen_B 求并集的运算，因此定值 d 一定出现在 f_B 的结果中。

【例 9.7】 基本块

$$d_1\colon \mathrm{a} = 3$$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第358页 文件编号: 422624 页面编号: 33584538442 打印人: 8685 2026-03-09 10:23:04

d_2: a = 4

的 gen 集合是 $\{d_2\}$，因为 d_1 没有向下暴露。kill 集合包含 d_1 和 d_2，因为 d_1 注销 d_2，d_2 也注销 d_1，故该块迁移函数的结果总会包含定值 d_2。

讨论完迁移等式后，下面考虑从基本块之间的控制流推导出来的一组约束等式。只要存在一条路径使一个定值能沿着它到达某个程序点，就称该定值能到达该点，所以只要有控制流边从 P 到 B，则有 $\mathrm{OUT}[P] \subseteq \mathrm{IN}[B]$。另外，若不存在任何路径使一个定值能沿着它到达某个程序点，就称该定值不能到达该点，因此 $\mathrm{IN}[B]$ 也没有必要大于它所有前驱块出口的到达–定值集合的并集。于是，取

$$\mathrm{IN}[B] = \cup_{P\text{是}B\text{的前驱}} \mathrm{OUT}[P]$$

是安全的。

这里的集合并算符称为到达–定值的**交算符**（**meet operator**）。在任何数据流模式中，交算符是用来总结汇聚到同一点的各条路径的贡献，因此本章称它为**汇合算符**。

建立起到达–定值的数据流等式后，下面讨论相应方程组的迭代求解。为方便起见，假定每个流图都有两个空的基本块：一个称为 ENTRY，代表流图的起点；另一个称为 EXIT，代表流图的终点，见图 9.10。对于到达–定值问题，显然 OUT[ENTRY] 是空集。

这样，到达–定值问题由下面的方程组定义：

$$\mathrm{OUT}[\mathrm{ENTRY}] = \varnothing$$

和对所有块 B（ENTRY 除外）有

$$\mathrm{OUT}[B] = \mathrm{gen}_B \cup (\mathrm{IN}[B] - \mathrm{kill}_B)$$
$$\mathrm{IN}[B] = \cup_{P\text{是}B\text{的前驱}} \mathrm{OUT}[P]$$

该方程组可以用下面的算法 9.1 求解。该算法所得解是该方程组的最小不动点，也就是该解给各基本块赋予的 IN 和 OUT 值包含于该方程组其他任何解对应的值中。该算法的解是可以接受的，因为对这些 IN 和 OUT 集合中的每个集合来说，它的任何一个定值，肯定到达相应的点，且不包含任何不能保证到达的定值。

算法 9.1 到达–定值的迭代求解。

输入：一个流图及各块 B 的 kill_B 和 gen_B。

输出：该流图中每个块 B（ENTRY 除外）的 $\mathrm{IN}[B]$ 和 $\mathrm{OUT}[B]$。

方法：迭代从所有的 $\mathrm{OUT}[B]$ 都等于空集开始，逐步收敛到所期望的 IN 和 OUT 值为止。因为必须迭代到 IN 收敛（从而 OUT 也收敛），可以使用一个布尔变量 exchange 来记录对这些块处理一遍时，是否有 OUT 发生变化。为简洁起见，本算法以及后续一些类似算法都省略了准确判断是否发生变化的机制。

计算到达一定值的迭代算法如图 9.11 所示。前两行给各个 OUT 置初值。第 3 行启动迭代到收敛的循环，第 4~6 行的内循环把数据流等式应用到除 ENTRY 以外的所有块。

从直观上看，只要定值没有被注销，算法 9.1 传播定值到尽可能远的地方，因此算法可

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第359页 文件编号: 422624 页面编号: 33594538440 打印人: 8685 2026-03-09 10:23:04

以模拟程序所有可能的执行。算法 9.1 是终止的，因为在迭代过程中，任何块 B 的 OUT[B] 集合不会缩小；一旦某个定值被加入，它就一直在里面。由于一个程序的定值是有限的，最终总有一遍 while 循环没有向任何 OUT 添加任何定值，于是算法终止。这个终止是安全的，因为 OUT 没有改变时，下一遍的 IN 也不会改变。如果 IN 不变，那么 OUT 也不变，所有以后各遍都不会有任何改变。

```
(1) OUT[ENTRY] = ∅;
(2) for(除 ENTRY 以外的每个块 B)   OUT[B] = ∅;
(3) while(任何一个 OUT 出现变化)
(4)     for(除 ENTRY 以外的每个块 B){
(5)         IN[B] = ∪_{P是B的前驱} OUT[P];
(6)         OUT[B] = gen_B ∪ (IN[B] − kill_B);
(7)     }
```

图 9.11 计算到达–定值的迭代算法

流图中的结点数是 while 循环体执行次数的一个上界。因为如果一个定值到达某点，那么它能够沿着无环路径继续到达其他点，流图的结点数目是该流图中任何一个无环路径的结点数目的上界。while 循环的循环体每执行一次，一个定值就会沿着正在考察的路径至少前进一个结点，并且经常是前进多个结点，取决于这些结点被访问的次序。

经验表明，对第 4 行 for 循环中的结点适当地排序，可以使 while 循环的平均迭代次数小于 5。定值集合可以用位向量表示，其上的运算可以由位向量的逻辑运算完成。因此，算法 9.1 在实际使用中效率很高。

【例 9.8】 图 9.10 中的 7 个定值 $d_1, d_2, \cdots, d_7$ 用位向量来表示，从左边开始的第 i 位表示定值 d_i。集合的并集用对应位向量上的逻辑或运算来计算，两个集合 S 和 T 的差 $S-T$ 由先对 T 的位向量求补，然后和 S 的位向量进行逻辑与运算来完成。

表 9.1 列出了执行算法 9.1 时各个块的 IN 和 OUT 值的变化。初值由加上角标 0 的 $\text{OUT}[B]^0$ 表示，它们由算法 9.1 中的第 2 行赋值，都是空集。第一遍执行循环体后的值由 $\text{IN}[B]^1$ 和 $\text{OUT}[B]^1$ 表示，以此类推。

假定第 4~6 行 for 循环的执行按块 $B_1 \sim B_4$ 和 EXIT 的次序处理各基本块。当 $B = B_1$ 时，因为 OUT[ENTRY] 和 $\text{IN}[B_1]^1$ 都是空集，所以 $\text{OUT}[B_1]^1$ 等于 gen_{B_1}。该值和 $\text{OUT}[B_1]^0$ 有区别，因此知道将还有下一遍。

当 $B = B_2$ 时，

$$\begin{aligned}\text{IN}[B_2]^1 &= \text{OUT}[B_1]^1 \cup \text{OUT}[B_4]^0\\ &= 111\ 0000 + 000\ 0000 = 111\ 0000\\ \text{OUT}[B_2]^1 &= \text{gen}_{B_2}(\text{IN}[B_2]^1 - \text{kill}_{B_2})\\ &= 000\ 1100 + (111\ 0000 - 110\ 0001) = 001\ 1100\end{aligned}$$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第360页 文件编号: 422624 页面编号: 33604538448 打印人: 8685 2026-03-09 10:23:04

第二遍扫描后，OUT 集合不再有什么变化，因此在第三遍扫描后算法终止，各 IN 和 OUT 的值就是表 9.1 最后两列的值。

表 9.1 IN 和 OUT 的计算

块	$\mathrm{OUT}[B]^0$	$\mathrm{IN}[B]^1$	$\mathrm{OUT}[B]^1$	$\mathrm{IN}[B]^2$	$\mathrm{OUT}[B]^2$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111

9.2.4 活跃变量

一些代码改进变换依赖于从程序流图中逆向计算得出的信息，如活跃变量信息。**活跃变量分析**旨在确定从某程序点 p 出发的某条路径上变量 x 的值是否可能被引用。如果 x 可能被引用，就称 x 在程序点 p 是**活跃**的；否则称 x 在点 p 是**非活跃**的或**死亡**的。

活跃变量信息的一种重要应用就是基本块的寄存器分配，8.3 节和 8.4 节对此已有简单的介绍。一个保存在寄存器中的值，如果在该块出口是死亡的，那么离开该块时不必存储这个值。此外，如果所有寄存器都被占用，需要释放一个寄存器，那么首先选择保存死亡值的寄存器。

数据流等式的定义仍然直接根据 $\mathrm{IN}[B]$ 和 $\mathrm{OUT}[B]$，它们分别代表基本块 B 入口和出口的活跃变量集合。这些等式也是由先定义单条语句的迁移函数并把它们复合成基本块的迁移函数。对于单条语句 d: u = v + w 来说，它产生 v 和 w 的活跃性，注销 u 的活跃性，因此迁移函数也是产生–注销形式。逆向复合单条语句的迁移函数，得到基本块的迁移函数如下：

（1）use_B 是在块 B 中有引用并且在该引用前块 B 中无定值的变量集合。

（2）def_B 是在块 B 中有定值的变量集合。

它们分别对应到达–定值中的 gen_B 和 kill_B。

【例 9.9】 图 9.10 中的块 B_2 对 i 重新定值前引用 i，对 j 也是这样。因此 $\mathrm{use}_{B_2} = \{i, j\}$。另外，块 B_2 给 i 和 j 定值，因此 $\mathrm{def}_{B_2} = \{i, j\}$。

根据这两个定义，use_B 中的任何变量在块 B 的入口应该是活跃的，而 def_B 中的变量在块 B 的入口应该是死亡的，除非它也出现在 use_B 中。

将 def 和 use 联系到未知的 IN 和 OUT 的方程组定义：

$$\mathrm{IN}[\mathrm{EXIT}] = \varnothing$$

和对所有块 B（EXIT 除外）有

$$\mathrm{IN}[B] = \mathrm{use}_B \cup (\mathrm{OUT}[B] - \mathrm{def}_B)$$

$$\mathrm{OUT}[B] = \cup_{S\text{是}B\text{的后继}}\mathrm{IN}[s]$$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第361页 文件编号: 422624 页面编号: 33614538446 打印人: 8685 2026-03-09 10:23:04

第一个等式描述了边界条件：没有任何变量在程序终点活跃。第二个等式指出，如果一个变量在块中定值前有引用，或者在该块出口活跃且没有被该块定值，那么它在该块入口活跃。第三个等式指出，一个变量在块的出口活跃，当且仅当它在该块的某个后继块入口活跃。

必须注意的是，活跃性等式和到达–定值等式之间的联系。

（1）它们都以集合并运算符作为汇合运算符。因为在这两种数据流模式中，信息都是沿着路径传播，并且仅关心是否有一条具备相关性质的路径存在，而不是关心某个性质是否在所有路径上都保持。

（2）活跃性的信息流是逆控制流方向遍历，因为在这个问题中，要保证的是，变量 x 在点 p 之后可能被引用的信息要沿路径逆向传递到点 p 的前驱点，除非 x 正好在这两点之间被定值。

为求解该逆向问题，注意它和到达–定值问题的联系与区别。这里初始化 IN[EXIT] 而不是 OUT[ENTRY]，IN 和 OUT 集合的作用相互交换，use 和 def 分别代替了 gen 和 kill。同样，方程组的解不一定唯一，需要的也是最小解。用于求这个最小解的算法本质是算法 9.1 的逆向版本。

算法 9.2 活跃变量的迭代求解。

输入：一个流图及各块 B 的 def_B 和 use_B。

输出：该流图中每个块 B（EXIT 除外）的 IN[B] 和 OUT[B]。

方法：执行图 9.12 中的程序。

```
IN[EXIT] = ∅;
for(除 EXIT 以外的每个块 B)   IN[B] = ∅;
while(任何一个 IN 出现变化)
    for(除 EXIT 以外的每个块 B) {
        OUT[B] = ∪_{S是B的后继} IN[S];
        IN[B] = use_B ∪ (OUT[B] − def_B);
    }
```

图 9.12 计算活跃变量的迭代算法

9.2.5 可用表达式

如果从流图起点到点 p 的每条路径上都计算 $x+y$，并且在到达点 p 之前的最后一次计算 $x+y$ 后，一直到点 p 为止都没有对 x 或 y 赋值，那么称表达式 $x+y$ 在点 p **可用**。

在可用表达式数据流分析模式中，对于单条语句 d: $z=x+y$ 来说，它产生可用表达式 $x+y$，注销含 z 的可用表达式，因此迁移函数仍然是产生–注销形式。将基本块中每条语句的迁移函数复合，可以得到基本块 B 的迁移函数：

（1）e_gen_B 是在块 B 中产生的可用表达式集合。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第362页 文件编号: 422624 页面编号: 33624538444 打印人: 8685 2026-03-09 10:23:04

一个基本块产生可用表达式 $x+y$ 是指它计算 $x+y$，并且随后没有对 x 或 y 的定值。

（2）e_kill_B 是在块 B 中注销的可用表达式集合。

一个基本块注销含 z 的可用表达式，若它有对 z 定值的语句。

注意：可用表达式的产生和注销概念与到达–定值的这些概念不完全一样，然而，它们在这里的行为与它们在到达–定值中的行为本质上是一致的。

可用表达式的基本应用是寻找全局公共子表达式。例如，在图 9.13 中，若表达式 4 * i 在块 B_3 的入口可用，则块 B_3 的 4 * i 是公共子表达式。这有两种可能：一种是块 B_2 没有对 i 定值, 如图 9.13(a) 所示；另一种如图 9.13(b) 所示，块 B_2 对 i 定值后又重新计算 4 * i。

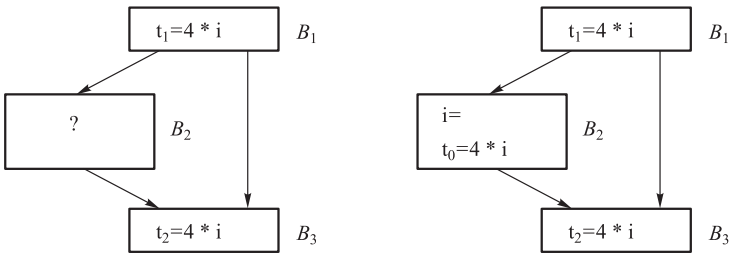


图 9.13 穿越块的公共子表达式

可用类似于计算到达–定值集合的方式寻找可用表达式。假定 U 代表程序中出现在语句右部的所有表达式的集合。对于每个块 B，令 IN[B] 和 OUT[B] 分别是块 B 入口和出口的可用表达式的集合，下列方程组把未知的 IN 和 OUT 与已知的 e_gen 和 e_kill 联系起来：

$$\mathrm{OUT}[\mathrm{ENTRY}] = \varnothing$$

和对所有块 B（ENTRY 除外）有

$$\mathrm{OUT}[B] = e_gen_B \cup (\mathrm{IN}[B] - e_kill_B)$$

$$\mathrm{IN}[B] = \cap_{P\text{是}B\text{的前驱}}\mathrm{OUT}[P]$$

该方程组和到达–定值分析的方程组几乎一样，其边界条件也与到达–定值分析一致：

$$\mathrm{OUT}[\mathrm{ENTRY}] = \varnothing$$

因为在块 ENTRY 的出口不可能有可用表达式。最重要的区别是，汇合算符不是集合并算符，而是集合交算符。用集合交运算是合适的，因为一个表达式在某块所有前驱的出口都可用时，才能说它在该块入口可用。而到达–定值的情况相反，一个定值只要能到达某块一个前驱的出口，它就能到达该块的入口。

使用 $\cap$ 而不是 $\cup$ 使得可用表达式方程的行为和到达–定值方程的行为有区别。虽然这两

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第363页 文件编号: 422624 页面编号: 33634538442 打印人: 8685 2026-03-09 10:23:04

种方程组的解都不是唯一解，但对到达-定值方程组而言，需要的是最小集合的解。为了得到这个解，由假设没有任何东西可到达任何地方开始，然后逐步增大到这个解。按这种方式，除非真能找到把 d 传播到 p 的一条路径，否则决不会获得定值 d 能到达点 p 的信息。相反，对可用表达式方程，想要的是最大集合的解，因此从足够大的近似开始，然后逐步缩小到所要的解。迭代计算从假定任何东西（集合 U）在任何地方都可用开始，然后逐步删掉那些不可用的表达式。删除一个表达式的依据是，能够找到一条路径，沿着这条路径该表达式是不可用的。最终获得真正的可用表达式集合。

在可用表达式情况下，产生可用表达式精确集合的一个子集是稳妥的，因为利用这些信息是为了用先前计算的值来代替可用表达式的计算，把一个其实可用的表达式当成不可用仅仅是阻止了进行这种优化。

【例 9.10】 把注意力集中在单个基本块，以图 9.14 中的块 B_2 来说明 $\mathrm{OUT}[B_2]$ 的初始近似对 $\mathrm{IN}[B_2]$ 的影响。令 G 和 K 分别是 $e_\mathrm{gen}_{B_2}$ 和 $e_\mathrm{kill}_{B_2}$ 的缩写，块 B_2 的数据流方程为

$$\mathrm{IN}[B_2] = \mathrm{OUT}[B_2] \cap \mathrm{OUT}[B_2]$$
$$\mathrm{OUT}[B_2] = G \cup (\mathrm{IN}[B_2 - K)$$

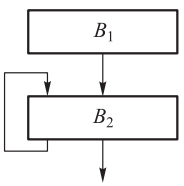


图 9.14 OUT 初值为空集则限制太紧

用 I^j 和 O^j 分别表示 $\mathrm{IN}[B_2]$ 和 $\mathrm{OUT}[B_2]$ 的第 j 次近似，这些方程可以重写成递推的形式：

$$I^{j+1} = \mathrm{OUT}[B_1] \cap O^j$$
$$O^{j+1} = G \cup (I^{j+1} - K)$$

从 $O^0 = \varnothing$ 开始，得到 $I^1 = \mathrm{OUT}[B_1] \cap O^0 = \varnothing$。但是，如果从 $O^0 = U$ 开始，得到 $I^1 = \mathrm{OUT}[B_1] \cap O^0 = \mathrm{OUT}[B^1]$，这才是应该得到的。直观上说，从 $O^0 = U$ 开始，所得到的解更有希望，因为它正确反映了一个事实：$\mathrm{OUT}[B_1]$ 中没有被块 B_2 注销的表达式在块 B_2 的出口可用。

算法 9.3 可用表达式的迭代求解。

输入：一个流图及各块 B 的 e_gen_B 和 e_kill_B。

输出：该流图中每个块 B（ENTRY 除外）的 $\mathrm{IN}[B]$ 和 $\mathrm{OUT}[B]$。

方法：执行图 9.15 的算法，每步的解释和前面对图 9.11 的解释类似。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第364页 文件编号: 422624 页面编号: 33644538440 打印人: 8685 2026-03-09 10:23:04


```
OUT[ENTRY] = ∅;
for(除 ENTRY 以外的每个块 B)  OUT[B] = U;
while(任何一个 OUT 出现变化)
    for(除 ENTRY 以外的每个块 B){
        IN[B] = ∩_{P是B的前驱} OUT[P];
        OUT[B] = e_gen_B ∪ (IN[B] − e_kill_B);
    }
```

图 9.15 计算可用表达式的迭代算法

9.2.6 小结

本节讨论了 3 个重要的数据流问题：到达–定值、活跃变量和可用表达式。如表 9.2 总结的那样，每个问题由数据流值的论域、数据流的方向、迁移函数、边界条件、汇合算符和数据流方程组成。汇合算符一般用 $\wedge$ 表示。

表 9.2 的最后一行是迭代算法所用的初值。初值这样选择可以保证迭代算法找到所求解方程组的最精确解。严格地说，这种选择不是数据流问题定义的一部分，因为它是用迭代算法求解时所需要的。此外，还有其他办法可用来求解这些问题。例如，从本节已经知道怎样从单条语句的迁移函数复合得到基本块的迁移函数，类似方式的复合可用来计算整个过程的迁移函数，或者从过程入口到任何程序点的迁移函数。

表 9.2 3 个数据流问题总结

数据流模式	到达–定值	活跃变量	可用表达式
论域	定值集合	变量集合	表达式集合
方向	正向	逆向	正向
迁移函数	$\text{gen}_B \cup (x - \text{kill}_B)$	$\text{use}_B \cup (x - \text{def}_B)$	$e_\text{gen}_B \cup (x - e_\text{kill}_B)$
边界	OUT [ENTRY] $= \varnothing$	IN[EXIT] $= \varnothing$	OUT [ENTRY] $= \varnothing$
汇合运算符 ($\wedge$)	$\cup$	$\cup$	$\cap$
方程	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \wedge_{P\text{是 }B\text{ 的前驱}}\ \text{OUT}[P]$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \wedge_{S\text{ 是 }B\text{ 的后继}}\ \text{IN}[S]$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \wedge_{P\text{是 }B\text{ 的前驱}}\text{OUT}[P]$
初始化	$\text{OUT}[B] = \varnothing$	$\text{IN}[B] = \varnothing$	$\text{OUT}[B] = U$

9.3 数据流分析的基础

在展示了数据流抽象的一些实例后，现在可以把各种数据流模式作为一个整体来抽象地研究，然后可以形式地回答数据流算法的下列几个基本问题：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第365页 文件编号: 422624 页面编号: 33654538447 打印人: 8685 2026-03-09 10:23:04

（1）在什么情况下数据流分析中使用的迭代算法是正确的？

（2）该迭代算法所得解的精度如何？

（3）该迭代算法是否收敛？

（4）数据流方程的解的含义是什么？

9.2 节首先针对到达-定值问题，非形式地讨论了上面这些基本问题，然后基于已经讨论过问题的答案来解释新问题，而不是针对每个新问题简略地回答这些基本问题。本节给出一种一般方式，它可以对一大类数据流问题回答这些基本问题。首先确定数据流模式被期望的性质，然后证明这些性质蕴涵数据流算法的正确性、精确性和收敛性，最后还有解的含义。这样，为了理解老算法或阐明新算法，只需要证明所定义的数据流问题具有某些性质即可，然后立即可以回答上面这些基本问题。

对一类数据流模式有一个共同理论框架的概念也具有实现上的意义。在软件设计过程中，该框架能帮助确认可复用的算法组件。这样不仅降低编码的工作量，而且减少编程错误，因为不需要为雷同的细节重新编码。

一个**数据流分析框架** $(D, V, \wedge, F)$ 包括：

（1）数据流分析的方向 D，它可以是正向或逆向；

（2）数据流值的论域 V 和其上汇合算子 $\wedge$ 的一个半格；

（3）一族从 V 到 V 的迁移函数 F。这族函数必须包括适用于边界条件的函数，这些边界条件是用于流程图中像 ENTRY 和 EXIT 这样特殊结点的常函数。

9.3.1 半格

由离散数学课程可知，**半格**由一个集合 V 和一个二元交运算（仍称为汇合运算）$\wedge$ 组成，并且该汇合运算满足下面 3 点性质：

（1）幂等性：对 V 中所有的 x，$x \wedge x = x$。

（2）交换性：对 V 中所有的 x 和 y，$x \wedge y = y \wedge x$。

（3）结合性：对 V 中所有的 x、y 和 z，$x \wedge (y \wedge z) = (x \wedge y) \wedge z$。

通常用二元组 $(V, \wedge)$ 表示半格。半格有顶元，用 $\top$ 表示，使得下式成立：

对 V 中的所有 x，$\top \wedge x = x$

半格可能还有底元，用 $\bot$ 表示，使得下式成立：

对 V 中所有的 x，$\bot \wedge x = \bot$

集合 V 上的关系 $\preceq$ 若满足下面 3 点性质，则它是一种偏序关系：

（1）自反性：对 V 中所有的 x，$x \preceq x$。

（2）反对称性：对 V 中所有的 x 和 y，如果 $x \preceq y$ 并且 $y \preceq x$，那么 $x = y$。

（3）传递性：对 V 中所有的 x、y 和 z，如果 $x \preceq y$ 并且 $y \preceq z$，那么 $x \preceq z$。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第366页 文件编号: 422624 页面编号: 33664538445 打印人: 8685 2026-03-09 10:23:04

二元组 $(V, \preceq)$ 称为**偏序集合**。在偏序集合上建立一个关系 $\prec$ 对今后的使用是方便的，其定义如下：

$x \prec y$ 当且仅当 $(x \preceq y)$ 并且 $(x \neq y)$。

半格 $(V, \wedge)$ 中的汇合运算 $\wedge$ 确定了半格值集 V 上的一种偏序 $\preceq$，它的定义如下：

对 V 中所有的 x、y 和 z，$x \preceq y$ 当且仅当 $x \wedge y = x$。

因为汇合运算具有幂等性、交换性和结合性，因此所定义的关系 $\preceq$ 具有自反性、反对称性和传递性，是一种偏序关系。其证明如下：

（1）自反性：由 $\preceq$ 的定义和 $\wedge$ 的幂等性立即可得 $x \preceq x$。

（2）反对称性：由 $\preceq$ 的定义，$x \preceq y$ 意味着 $x \wedge y = x$，$y \preceq x$ 意味着 $y \wedge x = y$，由 $\wedge$ 的交换性，$x = (x \wedge y) = (y \wedge x) = y$。所以，如果 $x \preceq y$ 并且 $y \preceq x$，那么 $x = y$。

（3）传递性：由 $\preceq$ 的定义，$x \preceq y$ 并且 $y \preceq z$ 意味着 $x \wedge y = x$ 并且 $y \wedge z = y$，由 $\wedge$ 的结合性，$(x \wedge z) = ((x \wedge y) \wedge z) = (x \wedge (y \wedge z)) = (x \wedge y) = x$。所以，如果 $x \preceq y$ 并且 $y \preceq z$，那么 $x \preceq z$。

【例 9.11】 在 9.2 节 3 种数据流分析模式中使用的汇合运算是集合并和集合交，它们都具有幂等性、交换性和结合性。对于集合并，空集 $\varnothing$ 是顶元，所有元素的集合 U 是底元，因为对 U 的任何子集 x，$\varnothing \cup x = x$ 并且 $U \cup x = U$。对于集合交，情况反过来，U 是顶元，$\varnothing$ 是底元。无论集合并还是集合交作为汇合运算，相应半格的值论域 V 是 U 的所有子集的集合，即 U 的幂集，用 2^U 表示。

对 V 中所有的 x 和 y，$x \cup y = x$ 蕴涵 $x \supseteq y$，因此由集合并强加的偏序是集合包含 $\supseteq$。对应地，由集合交强加的偏序是集合包含于 $\subseteq$，这时有较少元素的集合在该偏序中被视为较小的。而对于集合包含作为偏序，有较多元素的集合在该偏序中被视为较小的，这虽然违反直觉，但在上面这些定义下是不可避免的。

如 9.2 节所讨论的那样，数据流方程组通常有很多解，但是按偏序 $\preceq$ 意义上的最大解是最精确的。例如，在到达-定值分析中，其数据流方程最精确的解是含最少定值的那个解，它对应到这时的汇合运算 $\cup$ 所定义偏序中的最大元素。在可用表达式分析中，最精确的解是含最多表达式的那个解。同样，它对应到这时的汇合运算 $\cap$ 所定义偏序中的最大元素。

在半格的汇合运算和由它所确定的偏序关系之间存在另一种有用的联系。假定 $(V, \wedge)$ 是一个半格，V 中元素 x 和 y 的**最大下界**是 V 中的元素 g，满足：

（1）$g \preceq x$；

（2）$g \preceq y$；

（3）如果存在元素 z 也满足 $z \preceq x$ 和 $z \preceq y$，则 $z \preceq g$。

显然，x 和 y 汇合运算的结果就是它们唯一的最大下界。为了弄清这一点，令 $g = x \wedge y$，则可以看出：

（1）$g \preceq x$，因为利用汇合运算的结合性、交换性和幂等性，可以得到 $g \wedge x = (x \wedge y) \wedge x = x \wedge (y \wedge x) = x \wedge (x \wedge y) = (x \wedge x) \wedge y = x \wedge y = g$；

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第367页 文件编号: 422624 页面编号: 33674538443 打印人: 8685 2026-03-09 10:23:04

（2）同理可得 $g \preceq y$；

（3）假定元素 z 也满足 $z \preceq x$ 和 $z \preceq y$，则有 $z \preceq g$，因为 $z \wedge g = z \wedge (x \wedge y) = (z \wedge x) \wedge y = z \wedge y = z$，所以 $z \preceq g$。

因此，g 是 x 和 y 的唯一最大下界。

用格图画出论域 V 对讨论问题是很有帮助的。格图的结点是 V 的元素；如果 $y \preceq x$，则有从 x 朝下到 y 的有向边。例如，图 9.16 给出一个只有 3 个定值 d_1、d_2 和 d_3 的到达–定值数据流模式的集合 V。因为 $\preceq$ 是 $\supseteq$，所以从这 3 个定值的一个子集到它的每个超集都有一条边。因为 $\preceq$ 是传递的，所以当图中存在从 x 到 y 的其他路径时，省略从 x 到 y 的边是方便的。例如，虽然 $\{d_1, d_2, d_3\} \preceq \{d_1\}$，图中没有画相应的边，因为它可以由通过 $\{d_1, d_2\}$ 的路径来代表。

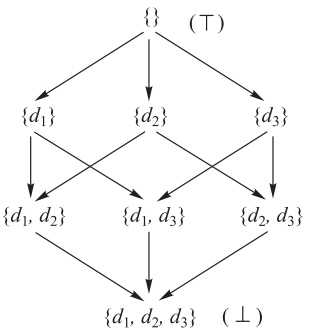


图 9.16　定值子集形成的格

从图 9.16 很容易看出，汇合运算的结果。因为 $x \wedge y$ 是 x 和 y 最大下界，所以它总是那个能从 x 和 y 向下的路径到达位于格图中最高的那个 z。例如，在图 9.16 中，$\{d_1\}$ 和 $\{d_2\}$ 的汇合结果是 $\{d_1, d_2\}$，因为这时的汇合运算是集合并。顶元 $\top$ 出现在格图的顶部，从它到任何元素都有路径。同样地，底元 $\perp$ 在格图的底部，任何元素到它都有路径。

本课程用半格概念就够了，因此没有进一步定义格。在图 9.16 中，如果把并（join）运算或最小上界运算（对应集合的 $\cap$ 运算）考虑进去，它实际上形成一个格。

图 9.16 仅涉及 3 个定值，而一个典型程序的格图可能非常大。数据流值的集合是定值集合的幂集，如果一个程序中有 n 个定值，那么数据流值集包含 2^n 个元素。然而，一个变量的定值是否能到达一个程序点，与其他变量的定值是否可达无关。因此，可以用从每个变量的一个简单半格构造出的“积半格”（product semilattice）来表示整个定值半格。例如，若图 9.16 中的定值 d_1、d_2 和 d_3 正好是程序中某个变量的仅有定值，则图 9.16 是该变量的定值半格。若对另一个变量仅有 d 一个定值，则相应的半格只有两个元素。空集 $\{\}$ 是顶元，$\{d\}$ 是底元。

积半格可以形式地定义如下。假定 $(A, \wedge_A)$ 和 $(B, \wedge_B)$ 是半格，它们的**积半格**定义如下：

（1）积半格的论域是 $A \times B$。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第368页 文件编号: 422624 页面编号: 33684538441 打印人: 8685 2026-03-09 10:23:04

（2）积半格的汇合运算 $\wedge$ 这样定义：如果 (a,b) 和 (a',b') 是积半格论域的元素，那么

$$(a,b)\wedge(a',b')=(a\wedge_A a',b\wedge_B b') \tag{9.4}$$

积半格 $(A\times B,\wedge)$ 上的偏序可由其分量 A 和 B 的偏序 $\preceq_A$ 和 $\preceq_B$ 按逐分量比较来定义：

$$(a,b)\preceq(a',b')\text{当且仅当}a\preceq_A a'\text{且}b\preceq_B b' \tag{9.5}$$

很容易说明规则(9.5)是由规则(9.4)得到的，因为若询问在什么情况下有

$$(a,b)\wedge(a',b')=(a,b)\qquad[(a,b)\preceq(a',b')]$$

则可知只有在 $a\wedge_A a'=a$ 且 $b\wedge_B b'=b$ 时才会出现，而这两个条件和 $a\preceq_A a'$ 且 $b\preceq_B b'$ 一样。

半格的积是可结合的运算，因此可以证明，规则(9.4)和规则(9.5)可以推广到任意数目的半格。给定半格 $(A_i,\wedge_i)$，$i=1,2,\cdots,k$，那么按此次序，这 k 个半格的积论域是 $A_1\times A_2\times\cdots\times A_k$，其汇合运算定义为

$$(a_1,a_2,\cdots,a_k)\wedge(b_1,b_2,\cdots,b_k)=(a_1\wedge_1 b_1,a_2\wedge_2 b_2,\cdots,a_k\wedge_k b_k)$$

其偏序定义为

$$(a_1,a_2,\cdots,a_k)\preceq(b_1,b_2,\cdots,b_k)\text{ 当且仅当对所有的 }i\text{ 有 }a_i\preceq_i b_i$$

从半格的高度可以推测数据流分析算法的收敛速度。在偏序集合 $(V,\preceq)$ 中，一个上升链是一个序列 $x_1\prec x_2\prec\cdots\prec x_n$。一个半格的高度就是其中最长上升链中 $\prec$ 的个数，即该链中元素个数减 1。例如，对一个有 n 个定值的程序，到达-定值的半格高度是 n。

如果半格的高度有限，则证明一个数据流分析迭代算法的收敛是非常容易的。显然，如果半格的值论域有限，则它的高度一定有限；对于值论域无限的半格，其高度有限也是可能的。9.4 节详细讨论的常量传播算法中使用的半格就是一个这样的例子。

9.3.2 迁移函数

在一个数据流分析框架中，迁移函数族 $F:V\to V$ 有下列性质：

（1）F 有一个恒等函数 I，即对 V 中所有的 x，有 $I(x)=x$。

（2）F 封闭于复合，即对 F 中任意两个函数 f 和 g，由 $h(x)=g(f(x))$ 定义的函数 h 在 F 中。

【例 9.12】 在到达-定值分析中，迁移函数族 F 包含一个恒等函数，其中 gen 和 kill 都是空集。9.2.3 小节已展示其复合的封闭性，这里简要重复如下。假如 F 中有两个函数

$$f_1(x)=G_1\cup(x-K_1)\text{ 和 }f_2(x)=G_2\cup(x-K_2)$$

那么

$$f_2(f_1(x))=G_2\cup((G_1\cup(x-K_1))-K_2)$$

其中，等号的右边代数地等价于

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第369页 文件编号: 422624 页面编号: 33694538449 打印人: 8685 2026-03-09 10:23:04

$$(G_2 \cup (G_1 - K_2)) \cup (x - (K_1 \cup K_2))$$

如果令 $K = K_1 \cup K_2$ 并且 $G = G_2 \cup (G_1 - K_2)$，那么 f_1 和 f_2 的复合 f 为

$$f = G \cup (x - K)$$

f 与 f_1 和 f_2 的形式一样，因此它也是 F 的一个成员。

如果考虑可用表达式，则根据类似于到达–定值的理由同样表明，F 有一个恒等函数且封闭于复合。

为了使数据流分析的迭代算法正常工作，还需要数据流分析框架满足单调性。形式地说，一个数据流分析框架 $(D, V, \wedge, F)$ 是单调的，如果对 V 中所有的 x 和 y 以及 F 中所有的 f，

$$x \preceq y \text{ 蕴涵} f(x) \preceq f(y) \tag{9.6}$$

那么单调性也可以等价地定义为对 V 中所有的 x 和 y 以及 F 中所有的 f，

$$f(x \wedge y) \preceq f(x) \wedge f(y) \tag{9.7}$$

式(9.6)和式(9.7)这两个定义的差异较大，因此它们各自都有优于对方的使用场合。

下面给出这两个定义等价的概略证明。首先证明式(9.6)蕴涵式(9.7)。因为 $x \wedge y$ 是 x 和 y 的最大下界，所以

$$x \wedge y \preceq x \text{并且} x \wedge y \preceq y$$

由式(9.6)有

$$f(x \wedge y) \preceq f(x) \text{并且} f(x \wedge y) \preceq f(y)$$

因为 $f(x) \preceq f(y)$ 是 $f(x)$ 和 $f(y)$ 的最大下界，所以得到式(9.7)。

再证明式(9.7)蕴涵式(9.6)。假定 $x \preceq y$，由定义可知 $x \wedge y = x$。由式(9.7)可知

$$f(x \wedge y) \preceq f(x) \wedge f(y)$$

由假定可得

$$f(x) \preceq f(x) \wedge f(y)$$

因为 $f(x) \wedge f(y)$ 是 $f(x)$ 和 $f(y)$ 的最大下界，所以 $f(x) \wedge f(y) \preceq f(y)$。于是

$$f(x) \preceq f(x) \wedge f(y) \preceq f(y)$$

因而得到式(9.6)。

一个框架经常遵循比式(9.7)更强的条件，称为分配性条件。

对 V 中所有的 x 和 y 以及 F 中所有的 f，都有

$$f(x \wedge y) = f(x) \wedge f(y)$$

如果 $a = b$，那么由幂等性，$a \wedge b = a$，所以 $a \preceq b$。因此分配性蕴涵单调性，但反过来不成立。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第370页 文件编号: 422624 页面编号: 33704538447 打印人: 8685 2026-03-09 10:23:04

【例 9.13】 令 x 和 y 是到达–定值框架中的定值集合，令 f 是对某两个定值集合 G 和 K 的迁移函数 $f = G \cup (x - K)$。可以由检查下面的等式：

$$G \cup ((y \cup z) - K) = (G \cup (y - K)) \cup (G \cup (z - K))$$

来验证到达–定值框架满足分配性条件。首先考虑 G 中的定值，它们肯定既在等号左边的集合又在等号右边的集合中。然后考虑不属于 G 的定值，此时可以把 G 从等号的两边都删掉，也就是检验下面的等式是否成立。

$$(y \cup z) - K = (y - K) \cup (z - K)$$

该等式很容易从集合论的文氏图（Venn diagram）得到。

9.3.3 一般框架的迭代算法

现在把算法 9.1 推广到适用于很多种数据流分析问题。

算法 9.4 一般数据流分析框架的迭代求解。

输入：由下面几部分组成的数据流分析框架。

（1）带特殊结点 ENTRY 和 EXIT 的数据流图。

（2）数据流方向 D。

（3）数据流值集 V。

（4）汇合运算符 $\wedge$。

（5）迁移函数集合 F，其中 f_B 是块 B 的迁移函数。

（6）V 中的常量 v_{ENTRY} 或 v_{EXIT} 分别代表正向或逆向框架的边界条件。

输出：该流图中每个块 B（ENTRY 或 EXIT 除外）的数据流值集 IN[B] 和 OUT[B]。

方法：正向和逆向数据流问题的求解算法分别在图 9.17(a) 和图 9.17(b) 中。和 9.2 节的那些数据流迭代算法一样，用逐次逼近法计算每个块的 IN 和 OUT。

重写算法 9.4 的正向和逆向版本，使得实现汇合运算的函数作为参数是可能的，流图本身和边界条件也可以作为参数。按这种方式，编译器的实现者可以避免为编译器优化阶段使用的每种数据流分析框架存档基本迭代算法。

借助前面讨论的抽象框架，可以证明迭代算法的一些有用性质：

（1）如果算法 9.4 收敛，则结果是数据流方程组的一个解。

（2）如果框架单调，则所求得的解是数据流方程组的最大不动点（maximal fix point，MFP）。最大不动点解具有的性质是：对任何其他解，其 IN[B] 和 OUT[B] 的值 $\preceq$ 最大不动点解中相应的值。

（3）如果框架单调并且半格的高度有限，那么可以保证算法收敛。

现在基于正向框架来讨论这几点。逆向框架的情况本质上是与正向框架的情况一样的。第一个性质很容易证明。如果 while 循环体的一轮执行使得所有的 OUT 都不发生变化，则

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第371页 文件编号: 422624 页面编号: 33714538445 打印人: 8685 2026-03-09 10:23:04

所有的 IN 也不再发生变化，这就意味着方程组得到了满足。

为了证明第二个性质，首先证明对任何块 B 在算法迭代时，IN[B] 和 OUT[B] 的值仅能减小（在 $\preceq$ 关系的意义上）。这可以由归纳法来证明。

```
(1) OUT[ENTRY] = v_ENTRY;
(2) for(除 ENTRY 以外的每个块 B)   OUT[B] = ⊤;
(3) while(任何一个 OUT 出现变化)
(4)     for(除 ENTRY 以外的每个块 B){
(5)         IN[B] = ∧_{P是B的前驱} OUT[P];
(6)         OUT[B] = f_B(IN[B]);
(7)     }
```

（a）正向数据流问题的迭代算法

```
(1) IN[EXIT] = v_EXIT;
(2) for(除 EXIT 以外的每个块 B)   IN[B] = ⊤;
(3) while(任何一个 IN 出现变化)
(4)     for(除 EXIT 以外的每个块 B){
(5)         OUT[B] = ∧_{S是B的后继} IN[S];
(6)         IN[B] = f_B(OUT[B]);
(7)     }
```

（b）逆向数据流问题的迭代算法

图 9.17　迭代算法的正向和逆向版本

归纳基：基本情况是证明在 while 循环第一轮迭代后 IN[B] 和 OUT[B] 的值不大于初始化值。这是简单的，因为对不是 ENTRY 的所有块 B，IN[B] 和 OUT[B] 都被初始化为 $\top$（在图 9.17(a) 中，在 while 循环前增加将所有 IN[B] 都初始化为 $\top$ 的语句不会影响结果）。

归纳假设：假设经过 k 次迭代后，IN[B] 和 OUT[B] 的值都不大于 $k-1$ 次迭代后对应的值。

归纳证明：要证明同样情况也出现在 $k+1$ 次迭代和 k 次迭代之间。图 9.17(a) 的第 (5) 行有

$$\mathrm{IN}[B] = \wedge_{P\text{是}B\text{的前驱}}\mathrm{OUT}[P]$$

用记号 $\mathrm{IN}[B]^i$ 和 $\mathrm{OUT}[B]^i$ 分别表示经过 i 次迭代后 IN[B] 和 OUT[B] 的值。由归纳假设，$\mathrm{OUT}[B]^k \preceq \mathrm{OUT}[B]^{k-1}$，根据汇合运算的性质，可以知道 $\mathrm{IN}[B]^{k+1} \preceq \mathrm{IN}[B]^k$。下一步，第 (6) 行是

$$\mathrm{OUT}[B] = f_B(\mathrm{IN}[B])$$

因为 $\mathrm{IN}[B]^{k+1} \preceq \mathrm{IN}[B]^k$，根据单调性，有 $\mathrm{OUT}[B]^{k+1} \preceq \mathrm{OUT}[B]^k$。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第372页 文件编号: 422624 页面编号: 33724538443 打印人: 8685 2026-03-09 10:23:04

注意，IN[B] 和 OUT[B] 值的任何变化都必须满足方程组，而汇合运算总是返回它们输入的最大下界，迁移函数总是返回和本身块以及输入一致的唯一解。因此，如果迭代算法终止，则所求得解中的值都至少和其他解对应的值一样大，即算法 9.4 的解是方程组的最大不动点。

最后考虑第三个性质，数据流分析框架单调并且高度有限时迭代算法收敛。因为每个 IN[B] 和 OUT[B] 值有变化时一定减小，并且如果某轮迭代中没有变化，则算法终止，所以该算法保证在迭代次数不大于

$$\text{框架高度} \times \text{流图中结点数}$$

的情况下收敛。

9.3.4 数据流解的含义

由 9.3.3 小节可知，使用迭代算法得到的解是最大不动点。从程序语义的观点考察，这样的解代表什么？为了理解数据流分析框架（$D, F, V, \wedge$）的解，首先描述一个框架的理想解是什么样的，然后说明理想解在一般情况下是得不到的。算法 9.4 的结果是理想解的一种稳妥近似。

不失一般性，仍然假定所讨论的是正向数据流分析框架。考虑一个基本块 B 的入口，理想解的讨论从找出由 ENTRY 到基本块 B 入口所有**可能的**执行路径开始。只有存在程序的某次执行准确跟随某条路径时，这条路径被称为“可能的”。前面已说明，流图上的路径并不一定都能成为程序运行时所走的路径，即可能的执行路径集合是执行路径集合的子集。然后理想解计算每条可能路径末端的数据流值，并把汇合运算用于这些数据流值，以得到它们的最大下界。这个最大下界就是理想解。因此，程序的任何执行不可能为该程序点产生小于理想解的数据流值。此外，这个界是紧的，由最大下界唯一性可知，不存在较大的数据流值，它可以作为沿流图中每条到达块 B 的可能路径计算出的数据流值的最大下界。

现在可以形式地定义理想解了。对流图中任何块 B，令 f_B 是块 B 的迁移函数。考虑从起点 ENTRY 到某个块 B_k 的任意路径

$$P = \text{ENTRY} \to B_1 \to B_2 \to \cdots \to B_{k-1} \to B_k$$

由于程序路径可能有环，因此一个基本块可能在路径 P 中出现多次。定义 P 的迁移函数 f_P 为 $f_{B_1}, f_{B_2}, \cdots, f_{B_{k-1}}$ 的复合。注意，f_{B_k} 不是该复合的一部分，因为这条路径到达块 B_k 的入口为止。这样，执行这条路径建立的数据流值是 $f_P(v_{\text{ENTRY}})$，其中 v_{ENTRY} 是起点 ENTRY 的迁移常函数的结果。于是，块 B 的理想解是

$$\text{IDEAL}[B] = \wedge_{P\text{是从 ENTRY 到}B\text{的一条可能路径}} f_P(v_{\text{ENTRY}})$$

根据所讨论框架的格论偏序 $\preceq$，可以认定：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第373页 文件编号: 422624 页面编号: 33734538441 打印人: 8685 2026-03-09 10:23:04

（1）任何大于理想解 IDEAL 的回答一定是不对的；

（2）任何小于或等于 IDEAL 的值是稳妥的，也就是安全的。

从直观上讲，在稳妥的值中，越接近 IDEAL 的值越精确。为什么解必须小于或等于 IDEAL？注意，对于任何块，任何大于 IDEAL 的解都可以通过忽略某些可能路径而得到，但是可能路径是不能忽略的，因为一般来说，无法保证在这些路径上不存在这样的影响：它们导致基于该解所进行的程序变换无效。反过来说，任何小于 IDEAL 的解都可被看成是包括某些在流图中不存在的路径，或者虽然存在于流图但在程序执行时决不会走到的路径。该较小解允许对程序的所有可能执行都正确的变换，但是禁止了某些基于 IDEAL 允许的变换。

然而，先前已介绍过，寻找所有可能的执行路径是不可判定的，从而必须采取近似的办法。在数据流抽象中，假定流图中的每条路径都会走到，因而为块 B 定义的是所有路径上的汇合（meet over paths）解

$$\mathrm{MOP}[B] = \wedge_{P\text{是从 ENTRY 到 } B \text{ 的一条路径}} f_P(v_{\mathrm{ENTRY}})$$

MOP 解考虑的所有路径是 IDEAL 解考虑的所有可能路径的超集。因此，MOP 解不仅汇集了所有可能路径的数据流值，还包括那些不可能被执行路径的数据流值。显然，对所有的块 B，$\mathrm{MOP}[B] \preceq \mathrm{IDEAL}[B]$，简写成 $\mathrm{MOP} \preceq \mathrm{IDEAL}$。

注意，如果一个流图包含环，那么 MOP 解所考虑的路径数是无界的。因此，MOP 的定义并没有通向一个直接算法。图 9.17(a) 所示的迭代算法不是首先找出到达一个块的所有路径，然后使用汇合运算，而是首先访问每个基本块，并且不一定按照程序执行时的次序，然后在每个汇合点，把汇合运算作用在到目前为止所得到的数据流值上。其中所用的一些值是在初始化时人工引入的，它们并不代表从程序起点开始的任何执行的结果。

那么，MOP 解和由图 9.17(a) 得到的最大不动点解 MFP 之间有什么联系？

首先讨论结点的访问次序。在一次迭代中，可能先访问某个基本块而后访问它的某些前驱。如果前驱是 ENTRY 结点，OUT[ENTRY] 已经被适当地初始化为一个常量值。如果前驱是其他结点，它们都被初始化为 $\top$，这是一个不小于最终解的值。根据单调性，通过使用 $\top$ 作为输入得到的解不小于所期望的解。从某种意义上说，可以认为 $\top$ 代表没有任何信息。

较早地使用汇合运算会带来什么影响？考虑图 9.18 的简单例子并假定所关心的是 $\mathrm{IN}[B_4]$ 的值。根据 MOP 的定义，有

$$\mathrm{MOP}[B_4] = (f_{B_3} \circ f_{B_1})(v_{\mathrm{ENTRY}}) \wedge (f_{B_3} \circ f_{B_2})(v_{\mathrm{ENTRY}})$$

在迭代算法中，如果按块 B_1、B_2、B_3 和 B_4 的次序访问结点，那么

$$\mathrm{IN}[B_4] = f_{B_3}(f_{B_1}(v_{\mathrm{ENTRY}}) \wedge f_{B_2}(v_{\mathrm{ENTRY}}))$$

汇合运算在 MOP 定义的最后使用，但它在迭代算法中使用得较早一些。当数据流分析框架具有分配性时，它们的结果是一样的。如果框架具有单调性而没有分配性时，仍然有 $\mathrm{IN}[B_4] \preceq \mathrm{MOP}[B_4]$。如果能对所有的块 B 证明 $\mathrm{IN}[B] \preceq \mathrm{MOP}[B]$，就很容易得到对所有的块 B，$\mathrm{IN}[B]$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第374页 文件编号: 422624 页面编号: 33744538449 打印人: 8685 2026-03-09 10:23:04

是稳妥的了。

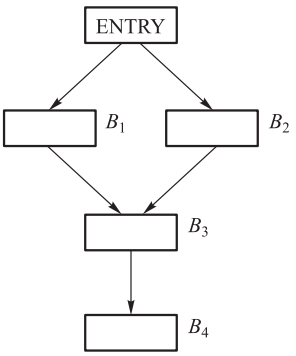


图 9.18 说明路径上较早汇合的影响的流图

现在可以简短概述为什么由迭代算法得到的 MFP 解总是安全的。对迭代次数 i 进行归纳可以证明，i 次迭代后得到的值总小于或等于长度不大于 i 的所有路径的汇合运算结果。但是，只有在到达所得结果肯定与无界次迭代的结果一样时，迭代算法才会终止。因此，迭代结果不会大于 MOP 的结果，即 MFP $\preceq$ MOP。因为前面已经说明，MOP $\preceq$ IDEAL，因此 MFP $\preceq$ IDEAL。所以，由迭代算法得到的 MFP 解是安全（稳妥）的。

9.4 常量传播

9.2 节讨论的数据流模式实际都是高度有限且具有分配性的框架的简单例子，对它们使用算法 9.4 的正向或逆向版本能够得到 MOP 解。本节将重点分析常量传播框架，它是一种具有一些特殊性质的数据流分析框架。

与之前讨论过的数据流分析问题相比，常量传播框架的不同之处如下。

（1）即使对于一个确定的流图，其数据流值的集合仍然是无界的。

（2）它不具备分配性。

常量传播是一个正向数据流问题。

9.4.1 常量传播框架的数据流值

常量传播框架的数据流值集合是一个积半格，其中每个成员对应程序中的一个变量。对于单个变量的半格，其论域包括以下内容。

（1）变量的类型允许的所有常量，即变量可能取的具体常量值。

（2）特殊值 NAC，表示非常量。如果一个变量的值被确定为非常量，则该变量映射到 NAC。被确定为非常量的具体情况有：变量被赋予一个输入值；或者依赖于不等于常量的其

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第375页 文件编号: 422624 页面编号: 33754538446 打印人: 8685 2026-03-09 10:23:04

他变量；或者变量在不同路径被赋予不同的常量。

（3）特殊值 UNDEF，表示没有定义。如果无法确定一个变量的值，则该变量映射到 UNDEF。具体情况有：该变量没有任何定值到达所关心的程序点。

NAC 和 UNDEF 是不同的，本质上是相反的。一个变量在某程序点的值为 UNDEF，表示没有该变量的定值到达该点；而值为 NAC 表示有定值到达该点，但不能确定它是常量。

整型变量的半格如图 9.19 所示。顶元是 UNDEF，底元是 NAC，它们分别是该偏序的最大值和最小值。整数之间无偏序关系，但是它们都小于 UNDEF 并且都大于 NAC。两个值之间的汇合是它们的最大下界。因此对任何 v，

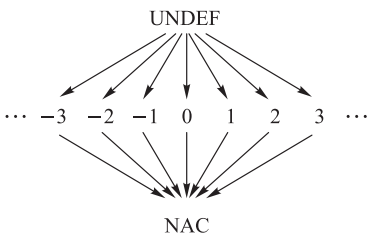


图 9.19 表示单个整型变量可能“值”的半格

$$\text{UNDEF} \wedge v = v \text{ 并且 } \text{NAC} \wedge v = \text{NAC}$$

对任何常量 c，

$$c \wedge c = c$$

对任何两个不同的常量 c_1、c_2，

$$c_1 \wedge c_2 = \text{NAC}$$

上面讨论的是单个变量的半格。常量传播框架的数据流值是程序中各变量的半格的积，即该框架的数据流值是一个映射，它把程序中每个变量映射到整型变量值半格中的一个值。若变量 v 在映射 m 中的值用 $m(v)$ 表示，那么 $m \preceq m'$ 当且仅当对所有的变量 v 有 $m(v) \preceq m'(v)$。用另一种方式表示若对所有的变量 v 有 $m''(v) = m(v) \wedge m'(v)$，则定义映射上的汇合运算为 $m \wedge m' = m''$。

9.4.2 常量传播框架的迁移函数

假定每个基本块只有一条语句。因为基本块有多条语句时，其迁移函数可以通过复合其中每条语句的迁移函数来构造。

集合 F 由迁移函数组成，每个迁移函数以 9.4.1 节最后介绍的变量到值的映射为变元，并且返回一个这样的映射。F 包含恒等函数，还包含用于 ENTRY 结点的迁移常函数。对任

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第376页 文件编号: 422624 页面编号: 33764538444 打印人: 8685 2026-03-09 10:23:04

何映射，该常函数返回 m_0。m_0 的定义是：对所有变量 v，$m_0(v) = \text{UNDEF}$。该边界条件是有意义的，因为在程序的所有语句开始执行前，任何变量都还没有定值。

令 f_s 是语句 s 的迁移函数，令 m 和 m' 代表满足 $m' = f(m)$ 的数据流值。f_s 的描述可以根据 m 和 m' 之间的联系确定：

（1）如果 s 不是赋值语句，则 f_s 是恒等函数。

（2）如果 s 是对变量 x 赋值，那么对所有的 $v \neq x$，$m'(v) = m(v)$。$m'(x)$ 的值由下面几种情况决定：

① 如果语句 s 的右部是一个常量 c，则 $m'(x) = c$；

② 如果语句 s 的右部是 $y + z$，那么

$$m'(x) = \begin{cases} m(y) + m(z), & \text{如果 } m(y) \text{ 和 } m(z) \text{ 都是常量值} \\ NAC, & \text{如果 } m(y) \text{ 或 } m(z) \text{ 是 NAC} \\ UNDEF, & \text{否则} \end{cases}$$

9.4.3 常量传播框架的单调性

下面证明常量传播框架是单调的。首先考虑 f_s 在一个变量 x 上的影响。如果 s 的右部不是 $y + z$ 这种情况，函数 f_s 要么不改变 $m(x)$ 的值，要么将其改变为一个具体的常量。在这些情况下，f_s 肯定是单调的。

然后看 s 的右部是 $y + z$ 这种情况，f_s 的影响列在表 9.3 中。前两列是 y 和 z 可能的值，最后一列是 x 的值，它们的值按从最大到最小的次序出现在各列或子列中。不难看出，f_s 函数是单调的。

表 9.3 语句 $x = y + z$ 的常量传播迁移函数

$m(y)$	$m(z)$	$m'(x)$
UNDEF	UNDEF	UNDEF
	c_2	UNDEF
	NAC	NAC
c_1	UNDEF	UNDEF
	c_2	$c_1 + c_2$
	NAC	NAC
NAC	UNDEF	NAC
	c_2	NAC
	NAC	NAC

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第377页 文件编号: 422624 页面编号: 33774538442 打印人: 8685 2026-03-09 10:23:05

9.4.4 常量传播框架的非分配性

常量传播框架不具备分配性。也就是说，迭代算法的解 MFP 是安全的，但可能小于 MOP 解。下面举例来证明该框架无分配性。

【例 9.14】 在图 9.20 中，块 B_1 分别给 x 和 y 赋常量 2 和 3，块 B_2 给它们置的值相反。显然，不管取什么路径，在块 B_3 的出口，z 的值是 5。然而，算法 9.4 未能发现这个事实。因为该迭代算法在块 B_3 的入口而不是出口使用汇合运算，使得 x 和 y 在块 B_3 入口的值都是 NAC，从而在块 B_3 的出口 z 等于 NAC。这个结果虽不精确，但是安全的。

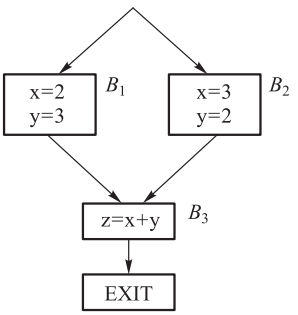


图 9.20 说明常量传播框架无分配性的示例

算法 9.4 不精确是因为它无法明确区分 x=2 且 y=3 和 x=3 且 y=2 之间的关联性。使用更加复杂的框架有可能明确所有涉及的程序变量的表达式之间的相等性，但是代价将大大增加。

从理论上讲，可以把这种精确性的丢失归因于常量传播框架没有分配性。令 f_1、f_2 和 f_3 分别代表块 $B_1 \sim B_3$ 的迁移函数，表 9.4 体现了

$$f_3(f_1(m_0) \wedge f_2(m_0)) \prec f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$$

成立，即该框架不具备分配性。

表 9.4 无分配性的迁移函数的示例

m	$m(x)$	$m(y)$	$m(z)$
m_0	UNDEF	UNDEF	UNDEF
$f_1(m_0)$	2	3	UNDEF
$f_2(m_0)$	3	2	UNDEF
$f_1(m_0) \wedge f_2(m_0)$	NAC	NAC	UNDEF
$f_3(f_1(m_0) \wedge f_2(m_0))$	NAC	NAC	NAC
$f_3(f_1(m_0))$	2	3	5
$f_3(f_2(m_0))$	3	2	5
$f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$	NAC	NAC	5

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第378页 文件编号: 422624 页面编号: 33784538440 打印人: 8685 2026-03-09 10:23:05

9.4.5 结果的解释

UNDEF 在算法 9.4 中有两个用途：给块 ENTRY 置初值和在迭代前给程序其他块置初值。这两种情况的含义略有区别。第一种情况表示，在程序开始执行前，所有的变量都没有定值；第二种情况表示，在迭代开始前且缺乏信息的情况下，把解先近似为顶元 UNDEF。

在迭代过程结束时，变量在块 ENTRY 出口的值仍然都是 UNDEF，因为 OUT[ENTRY] 没有改变。UNDEF 也可能出现在其他程序点的迭代结果中，它表示沿任何到达该点的路径上没有对某个变量的定值。注意，按照汇合运算的定义方式，只要存在给某个变量定值的一条路径到达一个程序点，变量在该点的值就肯定不是 UNDEF。这样，如果某变量到达一个程序点的所有定值都有同样的常量值，即使沿某条到达该点的路径上没有该变量的定值，该变量在该点也被看成常量。

【例 9.15】 在图 9.21 中，在块 B_2、B_3 的出口，x 的值分别为 10 和 UNDEF。因为 UNDEF $\wedge$ 10 = 10，x 在块 B_4 入口的值是 10。于是，块 B_5 的 x 引用可以用 10 来代替。但是，如果执行路径是 $B_1 \to B_3 \to B_4 \to B_5$，那么 x 到达块 B_5 的值是 UNDEF。因此，用 10 代替 x 似乎不正确。

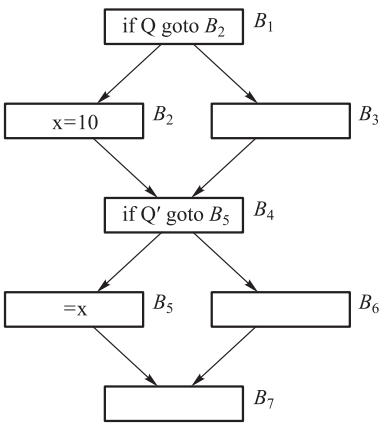


图 9.21 UNDEF 和一个常量的汇合

但是，如果程序正确，所有变量一定先定值后引用，则这条路径在运行时根本不会被执行，因为该路径引用了没有定值的 x。因此在程序正确的情况下，认为 x 在块 B_5 入口的值只能为 10 是合适的。如果程序员能够知道断言 Q 为假而 Q′ 为真是不可能出现的，就知道这条路径不可能执行；但是，一般来说，它已经超出了任何数据流分析的判断能力。不过，即使程序不正确，该路径可能执行，选择 10 作为 x 的值并不比 x 是某个随机值更坏。因此，块 B_5 的 x 引用可以用 10 来代替。

从例 9.15 中可以看出，该算法能找出更多的常量。其做法是，为了使程序更有效，它方

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第379页 文件编号: 422624 页面编号: 33794538448 打印人: 8685 2026-03-09 10:23:05

便地为那些可能没有定值的变量选择一个初值。在多数编程语言中，这样做是合法的，因为允许给没有定值的变量任取一个值。

如果语言的语义要求给所有没有定值的变量一个特定的值，则常量传播问题的形式化需要做相应的修改。如果兴趣在于找出程序中可能没有定值的变量，那么可以形式化另一个数据流分析问题来求解。

9.5 部分冗余删除

本节讨论如何最小化表达式计算的次数。具体来说，针对流图中的所有路径，分析表达式（如 x + y）的计算次数。通过检查 x + y 的计算点及其周围区域，在必要时将 x + y 的计算结果保存在一个临时变量中，可以减少 x + y 在多个路径上的重复计算，而不会增加任何路径上 x + y 的计算次数。即使这种优化有可能在流图中增加少量 x + y 的计算点，只要 x + y 的总体计算次数减少，这种优化依然是值得的。

应用本节介绍的代码变换可以改进代码的性能，因为这里的策略是：一个计算尽量不做，除非它必须得做。每个优化编译器实现了类似这里所描述的变换，虽然它使用的算法可能不像本节算法那样“积极进取”。

本节选择删除冗余以达到最小化表达式计算次数问题来讨论，还有另一个动机，那就是找出流图中计算每个表达式的恰当位置需要 4 种不同的数据流分析，这有助于增强对数据流分析在编译器中作用的理解。

程序中的冗余以两种形式存在：一种是以公共子表达式的形式出现；另一种是以循环不变表达式的形式出现。冗余也可能是部分的，它可能在一部分路径上出现，但并非在所有路径上都存在。公共子表达式和循环不变表达式可被看成是部分冗余的特殊情况，于是一个部分冗余删除算法可以衍生到删除各种形式的冗余。

下面首先讨论冗余的不同形式，以建立对问题的直观认识，然后描述广义的冗余删除问题，最后提出算法。该算法特别有意义，因为它涉及求解多个正向或逆向的数据流问题。

9.5.1 冗余的根源

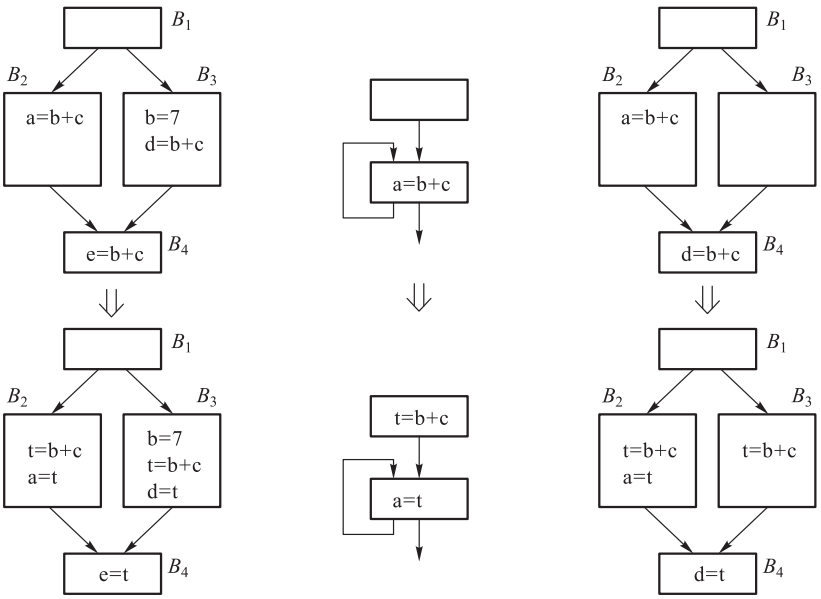
图 9.22 列举了 3 种形式的冗余：公共子表达式、循环不变表达式和部分冗余表达式。该图同时给出了优化前后的代码。

在图 9.22(a) 中，块 B_4 中表达式 b + c 为公共子表达式，它的计算冗余。可以在块 B_2 和块 B_3 中把 b + c 的结果保存在同样的临时变量 t 中，删除块 B_4 中的 b + c 计算，把 t 赋给 e。

形式地说，表达式 b + c 在点 p（完全）**冗余**，如果它是 9.2.5 节概念上的可用表达式，即在到达点 p 的所有路径上都计算 b + c，并且在最后一次计算 b + c 后没有对 b 或 c 重新

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第380页 文件编号: 422624 页面编号: 33804538446 打印人: 8685 2026-03-09 10:23:05

定值。



(a) 全局公共子表达式　(b) 循环不变代码外提　(c) 部分冗余删除

图 9.22　优化的例子

注意，使用可用表达式分析只能识别出正文上完全一样的冗余表达式。例如，公共子表达式删除可以识别代码段

```
t1 = b + c
a = t1 + d
t2 = b + c
e = t2 + d
```

的 t_1 和 t_2 有相同的值，只要 b 和 c 在它们之间没有重新定值。但是它不能识别出 a 和 e 的值也一样。当然，通过反复使用公共子表达式删除，直到没有新的公共子表达式被发现为止，是可以删除这种“深层”的公共子表达式的。

图 9.22(b) 是循环不变表达式的例子。表达式 b + c 是循环不变计算，只要 b 和 c 在循环中都没有重新定值。可以这样来优化，将 b + c 外提到循环外面计算一次，赋给临时变量 t，循环中的 b + c 用 t 来代替。

循环不变表达式的删除需要将表达式从循环的里面移到循环的外面，而不像公共子表达式删除那样简单地忽略冗余计算，因此这种优化通常被称为**循环不变代码外提**。循环不变代码外提也可能需要反复使用，因为一旦某个变量被确定为具有循环不变的值，则引用该变量的表达式也有可能成为循环不变的。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第381页 文件编号: 422624 页面编号: 33814538444 打印人: 8685 2026-03-09 10:23:05

图 9.22(c) 是一个部分冗余表达式的例子。块 B_4 的表达式 b + c 在路径 $B_1 \to B_2 \to B_4$ 上是冗余的，但在路径 $B_1 \to B_3 \to B_4$ 上不冗余，可以在块 B_3 中添加 b + c 的计算来删除前一条路径上的冗余。这样，与循环不变代码外提一样，部分冗余删除相当于移动了一个 b + c 的计算。

9.5.2 能否删除所有的冗余

每条路径上的所有冗余计算是否都能删除？回答是否定的，除非允许在流图中增加新的块。

【例 9.16】 在图 9.23(a) 中，如果程序的执行路径是 $B_1 \to B_3 \to B_4$，则块 B_4 的表达式 b + c 是冗余的。然而，不能简单地把 b + c 的计算移到块 B_3，因为这么做会导致路径 $B_1 \to B_3 \to B_5$ 上增加了一个额外的 b + c 计算。

若要删除这个部分冗余，则 b + c 的计算只能沿块 B_3 到块 B_4 的边放置，具体做法可以像图 9.23(b) 那样增加块 B_6，并把 b + c 计算放置在块 B_6 中。

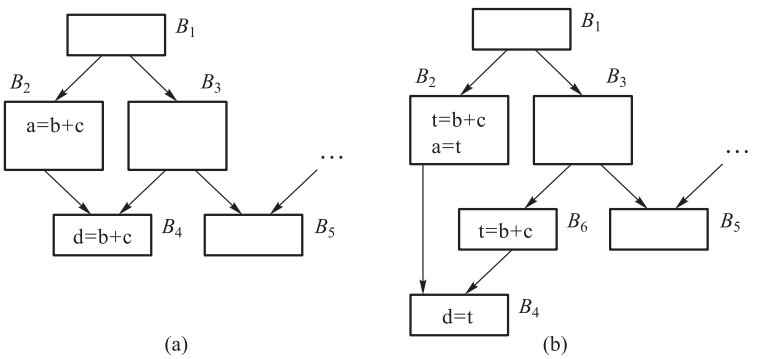


图 9.23 B3 → B4 是一条关键边

流图中的一条边，若其源结点有多个后继结点并且其目的结点有多个前驱结点，则称它为**关键边**，如图 9.23(a) 中块 B_3 到块 B_4 的边。通过在关键边的地方增加新块，总能适应放置表达式的需求。

但是增加新块并不是删除所有冗余计算的充分条件。例如，下面的例 9.17，有时需要复制代码以分离发现冗余的路径。

【例 9.17】 在图 9.24(a) 中，表达式 b + c 在路径 $B_1 \to B_2 \to B_4 \to B_6$ 上被冗余地计算。期望的优化是将这条路径上块 B_6 的冗余计算 b + c 删除，而仅在路径 $B_1 \to B_3 \to B_4 \to B_6$ 上放置一个 b + c 计算。然而，在路径 $B_1 \to B_3 \to B_4 \to B_6$ 的任何地方放置 b + c 计算，都会改变其他某条路径上出现的计算。为了做到不影响其他路径，需要复制块 B_4、B_6，使得块 B_4 和块 B_6 通过块 B_2 到达，而它们的复制块 B_4' 和块 B_6' 通过块 B_3 到达，

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第382页 文件编号: 422624 页面编号: 33824538442 打印人: 8685 2026-03-09 10:23:05

如图 9.24(b) 所示。

因为路径数目与同程序中的分支数目呈指数关系，删除所有的冗余可能导致优化后的代码规模大大增加，所以，以下仅考虑增加基本块的冗余代码技术，而不考虑复制基本块的情况。

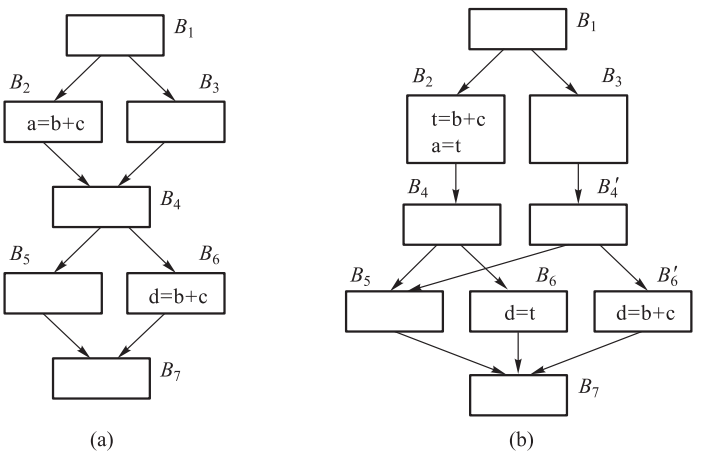


图 9.24 复制代码以删除冗余

9.5.3 惰性代码移动问题

实施部分冗余删除算法，使优化后的程序具有下列性质是可能的。

（1）不需要通过复制代码就可以删除的表达式冗余计算都被删除。

（2）优化后的程序不会执行不出现在原来程序中的任何计算。

（3）表达式的计算尽可能推迟。

最后一个性质是重要的，因为被用来避免冗余计算的表达式值通常需要保存在寄存器中，直到它最后一次被使用。推迟计算不仅可以最小化表达式的生存期（从该值的生成到它最后一次被引用的期间），还减少了寄存器的占用时间。这种通过延迟计算来删除部分冗余的优化技术被称为 **惰性代码移动**。

为建立对这个问题的直观认识，首先讨论怎样推理有关单个表达式沿单条路径的部分冗余。为方便起见，在该讨论的其余部分假定每条语句构成一个只包含它本身的基本块。在讨论之前，再次认识一下完全冗余和部分冗余。

块 B 的表达式 e 是完全冗余的，如果沿所有到达块 B 的路径，e 都被求值，并且 e 的运算对象随后都没有被重新定值。令 S 是这样的基本块集合，其中每个块都含表达式 e，它们致使块 B 中的 e 冗余。S 中各块出边的集合一定形成一个 **割集**，如果将该割集删掉，那么从程序起点到块 B 的路径将全部被割断。而且，从 S 中的任何块到块 B 的路径都没有重新给 e 的运算对象定值。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第383页 文件编号: 422624 页面编号: 33834538440 打印人: 8685 2026-03-09 10:23:05

如果块 B 的表达式 e 仅是部分冗余，那么惰性代码移动算法试图通过在流图中放置该表达式的副本来致使块 B 中的 e 完全冗余。如果这种尝试成功，那么修改后的流图也存在基本块集合 S，每个块都包含表达式 e，并且它们的出边构成程序起点和块 B 之间的一个割集。像完全冗余情况一样，从 S 中任何块到块 B 的路径都没有重新给 e 的运算对象定值。

9.5.4 预期表达式

为了保证没有额外的运算被执行，需要对被添加的表达式强加一个约束。一个表达式的副本只能放置在该表达式被预期的程序点。如果所有从点 p 开始的路径上最终都从该点可用的 b 和 c 的值计算表达式 b + c 的值，表达式 b + c 在点 p 被认为是**预期的**。

现在讨论怎样删除一条无环路径 $B_1 \to B_2 \to \cdots \to B_n$ 上的部分冗余。假定表达式 e 仅在块 B_1、B_n 计算，并且 e 的运算对象在这条路径上没有重新定值。这条路径上有入边进入，也有出边离开。e 在块 B_i 的入口不是所期望的，当且仅当存在离开 $B_j (i \leqslant j \leqslant n)$ 的出边，它通向一条不使用 e 值的路径。于是，预期约束限制了表达式能放置的最早位置。

如果 e 在块 B_i 的入口是可用的或预期的，那么可以组成包含边 $B_{i-1} \to B_i$ 的一个割集，并且该割集致使块 B_n 的 e 是冗余的。如果 e 在块 B_i 的入口是预期的但不是可用的，则必须沿块 B_i 的入边放置 e 的副本。

由于流图中通常存在满足所有要求的若干割集，放置表达式 e 副本的位置也有多种选择。在上述方式中，e 沿正在关注路径的入边放置，它保证没有引入冗余，并且 e 副本的计算已经尽量靠近引用。注意，放置的这些 e 副本本身可能成为程序中其他路径上 e 的部分冗余。这样的部分冗余可以由进一步移动这些计算来删除。

总体来说，表达式的预期用来约束表达式的最早放置，表达式的放置不能再提早到它不具备预期性质的位置。表达式的放置越提早，可以删除的冗余可能越多。在删除同样冗余的所有放置中，最迟计算该表达式的位置是保存该表达式值的寄存器的生存期最短的位置。

9.5.5 惰性代码移动算法

由上面的讨论可构建一个 4 步算法。第一步使用预期来决定表达式可以放置在什么地方。第二步在那些能删除尽可能多的冗余计算并且不必复制代码也不会引入任何多余计算的割集中，找出最早的割集。该步把计算放置在它们的值最早被预期的程序点。第三步将割集尽量朝下推，一直到再推就会改变程序语义或引入冗余为止。第四步是一遍简单的扫描，通过删除对仅引用一次的临时变量的赋值来整理代码。每步的完成都需要数据流分析，第一步和第四步是逆向数据流问题，第二步和第三步是正向数据流问题。

1. 预处理

下面给出完整的惰性代码移动算法。为了使算法简洁，假定最初每条语句构成一个只含本身的基本块，并且只在块的入口放置表达式计算的副本。为了保证这种简化不会降低这种

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第384页 文件编号: 422624 页面编号: 33844538448 打印人: 8685 2026-03-09 10:23:05

技术的效力，将新基本块添加在一条边的源结点和目的结点之间，如果该目的结点的前驱多于一个。这样做明显关照到了程序中所有的关键边。

每个块的语义用两个集合来抽象，e_use_B 是块 B 中计算的表达式集合，e_kill_B 是块 B 注销的表达式集合，即程序中所有的至少有一个运算对象在块 B 中定值的表达式。下面的例 9.18 用来贯串该算法所涉及的 4 种数据流分析的讨论，这 4 种数据流分析的定义列在表 9.5 中。

【例 9.18】 在图 9.25(a) 中，表达式 b + c 出现 3 次。因为块 B_9 是循环的一部分，该表达式可能计算许多次。块 B_9 的这个计算不仅是循环不变计算，也是一个冗余表达式，因为它的值在块 B_7 中已经使用。在本例中，b + c 仅需要计算两次：一次在块 B_5；另一次在块 $B_2 \sim B_7$ 的路径上。惰性代码移动算法把 b + c 的计算放置在块 B_4、B_5 的入口。

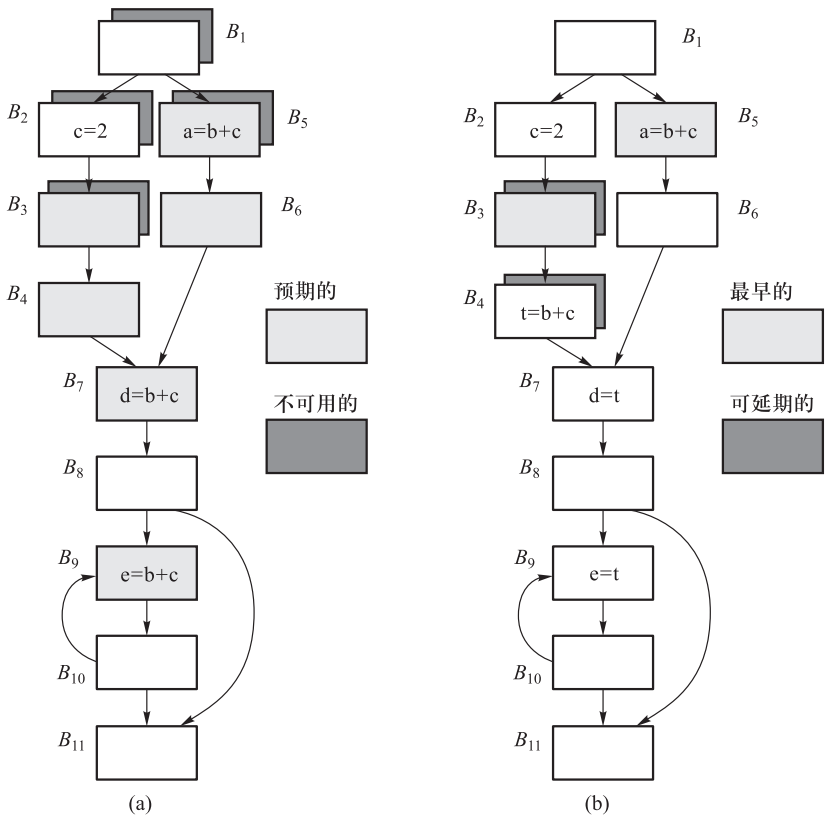


图 9.25 例 9.18 的流图

2. 预期表达式

在图 9.25(a) 中，所有在入口预期表达式 b + c 的基本块都用灰色表示，即 b + c 被块 $B_3 \sim B_7$、B_9 预期。它在块 B_2 的入口不是预期的，因为 c 的值在该块中重新计算，从而在

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第385页 文件编号: 422624 页面编号: 33854538445 打印人: 8685 2026-03-09 10:23:05

块 B_2 的入口如果计算 b + c，那么其值是不能在任何路径上使用的。b + c 在块 B_1 的入口也不是预期的，因为 c 在此处的值是不能沿从块 B_1 经块 B_2 的路径继续被引用的（虽然 b + c 沿路径 $B_1 \to B_5 \to B_6$ 可以被引用）。类似地，该表达式在块 B_8 的入口不是预期的，因为有从块 B_8 到 B_{11} 的分支。一个表达式的预期性可以沿路径摆动，这可以用路径 $B_7 \to B_8 \to B_9$ 来说明。

预期表达式问题的数据流方程展示在表 9.5 中的“预期表达式”列。它是逆向分析的。块 B 出口的预期表达式，如果它不在 e_kill_B 中，则也是 B 的入口的预期表达式。此外，块 B 产生 e_use_B 中表达式的新的预期。在程序的终点，没有任何表达式是预期的。因为兴趣在于找到沿每条随后的路径都被预期的表达式，所以汇合运算是集合交。这样，除了 EXIT，其他块必须被初始化为全集 U，就像 9.2.5 小节讨论的可用表达式那样。

表 9.5 部分冗余删除中的 4 个数据流问题

参数	预期表达式	可用表达式
论域	表达式集合	表达式集合
方向	逆向	正向
迁移函数	$f_B(x) = \text{e_use}_B \cup (x - \text{e_kill}_B)$	$f_B(x) = (\text{anticipated } [B].in \cup x) - \text{e_kill}_B$
边界	$\text{IN}[\text{EXIT}] = \varnothing$	$\text{OUT}[\text{ENTRY}] = \varnothing$
汇合运算符	$\cap$	$\cap$
方程	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \wedge_{S\text{是}B\text{的后继}}\text{IN}[S]$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \wedge_{P\text{是}B\text{的前驱}}\text{OUT}[P]$
初始化	$\text{IN}[B] = U$	$\text{OUT}[B] = U$
参数	可延迟表达式	引用表达式
论域	表达式集合	表达式集合
方向	正向	逆向
迁移函数	$f_B(x) = (\text{earliest}[B] \cup x) - \text{e_use}_B$	$f_B(x) = \text{e_use}_B \cup (x - \text{latest}[B])$
边界	$\text{OUT}[\text{ENTRY}] = \varnothing$	$\text{IN}[\text{EXIT}] = \varnothing$
汇合运算符	$\cap$	$\cup$
方程	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \wedge_{P\text{是}B\text{的前驱}}\text{OUT}[P]$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \wedge_{S\text{是}B\text{的后继}}\text{IN}[S]$
初始化	$\text{OUT}[B] = U$	$\text{IN}[\text{B}] = \varnothing$

3. 可用表达式

在第二步的结尾，表达式的副本将被放置在该表达式最早被期望的程序点。要做到这一点，需要定义可用表达式数据流问题。一个表达式在点 p **可用**，如果在到达点 p 的所有路径上它都被期望。这里定义的问题和 9.2.5 节描述的可用表达式类似，区别在于迁移函数。一个表达式在一个基本块的出口可用，如果该表达式在入口可用或在入口的预期表达式集合中（如果选择在这里计算它，它就成为可用的了），并且没有被该基本块注销。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第386页 文件编号: 422624 页面编号: 33864538443 打印人: 8685 2026-03-09 10:23:05

可用表达式的数据流方程展示在表 9.5 中的“可用表达式”列。为了避免和 IN 混淆，引用先前分析结果时，在先前分析的名字后面加上“[B].in”。

按照最早放置策略，在块 B 放置的表达式集合由下式定义：

$$\text{earliest}[B] = \text{anticipated}[B].\text{in} - \text{available}[B].\text{in}$$

即预期表达式中那些尚未可用的表达式。

【例 9.19】 在图 9.26 的流图中，表达式 b + c 在块 B_3 的入口不是预期的，但是它在块 B_4 的入口是预期的。然而，它无须在块 B_4 中计算，因为由于块 B_2，该表达式已经可用。

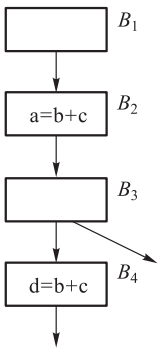


图 9.26 例 9.19 解释可用性的流图

【例 9.20】 在图 9.25(a) 中，带深灰色阴影的块表示表达式 b + c 在入口不可用，它们是块 $B_1 \sim B_3$ 和 B_5。最早放置由带深灰色阴影的灰色块表示，它们是块 B_3 和 B_5。例如，b + c 在块 B_4 的入口是可用的，因为存在路径 $B_1 \to B_2 \to B_3 \to B_4$，并且 b + c 在块 B_3 是被预期的，还有从块 B_3 的入口开始，b 和 c 都没有被重新计算。

4. 可延迟表达式

第三步是在保持源程序语义并且最小化冗余的情况下，尽可能延迟表达式的计算。下面以例 9.21 说明这一步的重要性。

【例 9.21】 在图 9.27 中，表达式 b + c 沿路径 $B_1 \to B_5 \to B_6 \to B_7$ 计算两次。b + c 在块 B_1 的入口甚至也是预期的。如果按照最早放置策略，就会在块 B_1 计算 b + c。若这样，b + c 的结果从执行的一开始就要保存，并通过由块 B_2、B_3 构成的循环，一直到使用它的块 B_7。显然可以缩短该结果的生存期，只要把 b + c 的计算延迟到块 B_5 的入口和控制流将要从块 B_4 转换到块 B_7 的时候。

形式地说，表达式 x + y 可以延迟到点 p，如果 x + y 的一个较早放置会遭遇这样的情况：从起点到 p 的每条路径都遇到这个放置，并且最后一次遇到该放置之后到 p 没有 x + y 的引用。

【例 9.22】 再次考虑图 9.25 的表达式 b + c。b + c 的两个最早位置是块 B_3 和 B_5，

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第387页 文件编号: 422624 页面编号: 33874538441 打印人: 8685 2026-03-09 10:23:05

它们是图 9.25(a) 中仅有的带深灰阴影的灰色块，表示 b + c 对这些块的入口来说是预期的和不可用的。不能把 b + c 从块 B_5 延迟到块 B_6，因为 b + c 在块 B_5 中有引用，但是可以把它从块 B_3 延迟块 B_4。

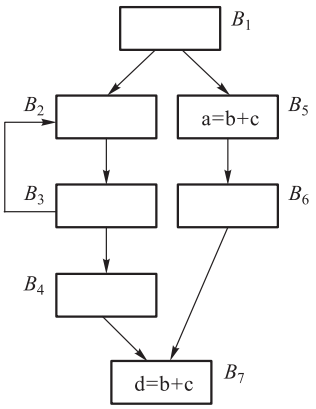


图 9.27 例 9.21 说明延迟表达式的流图

不能把 b + c 从块 B_4 再延迟到块 B_7，因为虽然 b + c 在块 B_4 中无引用，但是把 b + c 的计算放置在块 B_7 将导致在路径 $B_5 \to B_6 \to B_7$ 上出现 b + c 的冗余计算。下面将看到，块 B_4 是计算 b + c 的最晚位置。

可延迟表达式问题的数据流方程展示在表 9.5 中的“可延迟表达式”列。它是正向分析的，由于不可能把一个表达式“延迟”到程序的起点，因此 OUT[ENTRY] = ∅。一个表达式可以延迟到块 B 的出口，如果它在块 B 的入口是可延迟的或在 earliest[B] 中，并且在块 B 中没有引用。一个表达式不可能延迟到一个块的入口，除非这个表达式在该块所有前驱的出口都是可延迟的。因此，汇合运算是集合交，并且除了 ENTRY，其他块初始化为该半格的顶元——全集。

简略地说，为尽量延迟，一个表达式放置在从可延迟到不可延迟的边界上。具体而言，一个表达式 e 可以放置在块 B 的入口，只要它在块 B 的 earliest 或块 B 入口的 postponable 集合中。除此以外，要使块 B 是 e 的可延迟边界，当且仅当下面两个条件成立。

（1）e 不在 postponable[B].out 中。换句话说，e 在 e_use_B 中。

（2）e 不能延迟到块 B 的一个后继。换句话说，存在块 B 的一个后继，使得 e 不在该后继的 earliest 集合中，也不在该后继入口的 postponable 集合中。

该算法预处理步添加了适当的新基本块，因此，如果上面两个条件之一成立，那么表达式 e 就放置在块 B 的入口。

对于上面可延迟和不可延迟边界的非形式描述，可以给出 latest 集合的精确定义：

$$\text{latest}[B] = (\text{earliest}[B] \cup \text{postponable}[B].\text{in}) \cap$$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第388页 文件编号: 422624 页面编号: 33884538449 打印人: 8685 2026-03-09 10:23:05

$$(\text{e_use}_B \cup \neg(\cap_{S\text{是}B\text{的后继}}(\text{earliest}[S] \cup \text{postponable}[S].\text{in})))$$

其中，¬ 表示关于该程序计算的所有表达式集合的补集。

【例 9.23】 图 9.25(b) 表明了分析结果。浅灰色块表示它的 earliest 集合包含 b + c。深灰阴影表示相应块入口的 postponable 集合包含 b + c。因此 b + c 最迟放置在块 B_4 和 B_5 的入口，因为：

（1）b + c 在块 B_4 入口但不在块 B_7 入口的 postponable 集合中；

（2）块 B_5 的 earliest 集合包含 b + c，并且它引用 b + c。

b + c 的值在块 B_4 和 B_5 存入临时变量 t，然后 t 在其他地方用来代替 b + c，如图 9.25(b) 中显示的那样。

5. 引用表达式

最后一种逆向分析用来确定临时变量的引用是否超出给它们定值的块。一个表达式在点 p 被引用，如果存在从点 p 开始的一条路径，该路径在重新计算该表达式前引用它。这本质上是活跃性分析，但是面向表达式，而不是先前介绍的面向变量。

引用表达式问题的数据流方程展示在表 9.5 中的“引用表达式”列。它是逆向的。块 B 出口的引用表达式在块 B 的入口仍然是引用表达式，只要它不在 latest 集合中。一个基本块产生的新引用集合就是 e_use_B 集合。在程序的终点，没有表达式可引用。因为兴趣在于找出在随后任意一条路径上都有引用的表达式，所以汇合运算是集合并。因此，除了 EXIT，其他块初始化为该半格的顶元——空集。

6. 小结

该算法的所有步骤总结在算法 9.5 中。

算法 9.5 惰性代码移动。

输入：一个流图及各块 B 的 e_use_B 和 e_kill_B。

输出：一个被修改后的流图，满足 9.5.3 小节介绍的惰性代码移动的 3 个性质。

方法：按照下面的 8 个步骤。

（1）在所有不止一个前驱的基本块的所有入边上插入一个空块。

（2）按表 9.5 中的“预期表达式”列的定义，找出所有块 B 的 $\text{anticipated}[B].\text{in}$。

（3）按表 9.5 中的“可用表达式”列的定义，找出所有块 B 的 $\text{vailable}[B].\text{in}$。

（4）为所有的块 B 计算最早放置：

$$\text{earliest}[B] = \text{anticipated}[B].\text{in} - \text{available}[B].\text{in}$$

（5）按表 9.5 中的“可延迟表达式”列的定义，找出所有块 B 的 $postponed[B].in$。

（6）为所有的块 B 计算最迟放置：

$$\text{latest}[B] = (\text{earliest}[B] \cup \text{postponable}[B].\text{in}) \cap (\text{e_use}_B \cup \neg(\cap_{S\text{ 是 }B\text{ 的后继}}(\text{earliest}[S] \cup \text{postponable}[S].\text{in})))$$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第389页 文件编号: 422624 页面编号: 33894538447 打印人: 8685 2026-03-09 10:23:05

（7）按表 9.5 中的“引用表达式”列的定义，找出所有块 B 的 used[B].out。

（8）对程序计算的每个表达式，如 x+y，执行下面的步骤。

① 为 x+y 确定一个新临时变量，如 t。

② 对满足 x+y 属于 latest[B]∩used[B].out 的所有基本块 B，在块 B 的入口加上 t=x+y。

③ 对满足 x+y 属于 $e_use_B \cap (\neg latest[B] \cup used[B].out)$ 的所有基本块 B，把原来的 x+y 都用 t 代替。

该算法通过一种统一的方法识别多种不同形式的冗余计算，并结合多个数据流分析问题确定表达式的最佳放置位置。

9.6 流图中的循环

在前面的学习中，我们尚未把循环作为一类特殊的控制流来处理。然而，循环非常重要：程序执行的大部分时间消耗在循环上，改进循环性能的优化会对程序执行产生显著影响。因此标识循环并对它们进行专门处理至关重要。

循环也会影响程序分析的运行时间。如果程序不包含任何循环，只要对程序进行一遍扫描就可以回答数据流问题，如一个正向数据流问题可以通过按照拓扑顺序遍历所有结点一次来解决。

本节介绍支配结点、深度优先排序、回边、图的深度和可归约性等概念。这些概念用在随后的寻找循环和迭代数据流分析收敛速度的讨论中。

9.6.1 支配结点

流图中结点 d **支配**结点 n, 如果从流图起点开始，每条到达 n 的路径都要经过 d。写成 d dom n, 称 d 为 n 的支配结点，也称 d 为 n 的**必经结点**。根据这个定义，每个结点支配自身。

【例 9.24】 考虑图 9.28 所示的流图，它的起点是 1。起点支配所有结点（这句话对所有流图都成立）。结点 2 仅支配它自身，因为控制可以沿 $1 \to 3$ 开始的路径到达任何其他结点。结点 3 支配除结点 1、2 以外的所有结点。结点 4 支配除结点 1~3 以外的所有结点，因为从 1 出发的所有路径必须由 $1 \to 2 \to 3 \to 4$ 或 $1 \to 3 \to 4$ 开始。结点 5、6 分别仅支配自身，因为控制流可以经过这两者之一而跳过另一个。最后，结点 7 支配结点 7~10；结点 8 支配结点 8~10；结点 9、10 分别仅支配自身。

下面介绍一个算法，它计算流图中每个结点 n 的支配集，即 n 的所有支配结点。它基于的原理是，如果 $p_1, p_2, \cdots, p_k$ 是 n 的所有前驱，并且 $d \neq n$, 那么 d dom n, 当且仅当对所有的 $i(0 \leqslant i \leqslant k)$ 有 d dom p_i。该问题可以形式化为一个正向数据流分析。数据流值是基本块集合。一个结点的支配集是它各前驱支配集的交集再加上它自身，于是汇合运算是集合交。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第390页 文件编号: 422624 页面编号: 33904538445 打印人: 8685 2026-03-09 10:23:05

基本块 B 的迁移函数直接把块 B 自身加到输入结点集合上。边界条件是：ENTRY 结点支配它自身。最后，除了 ENTRY，其他结点初始化为全集，也就是所有结点的集合。

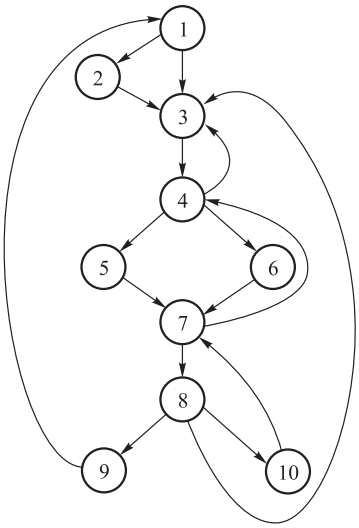


图 9.28 流图

算法 9.6 计算支配集。

输入：一个流图，其结点集为 N，起点为结点 ENTRY。

输出：N 中所有结点 n 的支配集 $D(n)$。

方法：以表 9.6 所列参数求解数据流问题。对 N 中所有结点，$D(n) = \mathrm{OUT}[n]$。

表 9.6 计算支配集的数据流问题

参数	支配集
论域	结点集合 N 的幂集
方向	正向
迁移函数	$f_B(x) = x \cup \{B\}$
边界	$\mathrm{OUT}[\mathrm{ENTRY}] = \{\mathrm{ENTRY}\}$
汇合运算符	$\cap$
方程	$\mathrm{OUT}[B] = f_B(\mathrm{IN}[B])$ $\mathrm{IN}[B] = \wedge_{P\text{是}B\text{的前驱}}\mathrm{OUT}[P]$
初始化	$\mathrm{OUT}[B] = N$

【例 9.25】 考虑图 9.28 所示的流图，并且假定图 9.17(a) 中程序的第 4~6 行的 for 循环是按数值序访问结点。令 $D(n)$ 等于 $\mathrm{OUT}[n]$。结点 1 是起点，在第 1 行赋值语句使

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第391页 文件编号: 422624 页面编号: 33914538443 打印人: 8685 2026-03-09 10:23:05

$D(1)=\{1\}$。结点 2 的唯一前驱是结点 1，因此 $D(2)=\{2\}\cup D(1)=\{1,2\}$。结点 3 有前驱结点 1、2、4、8。由于除了结点 1，其他结点都初始化为 $\{1,2,\cdots,10\}$，因此

$$D(3)=\{3\}\cup(\{1\}\cap\{1,2\}\cap\{1,2,\cdots,10\}\cap\{1,2,\cdots,10\})=\{1,3\}$$

其余结点的计算为

$$\begin{aligned}
D(4)&=\{4\}\cup(D(3)\cap D(7))=\{4\}\cup(\{1,3\}\cap\{1,2,\cdots,10\})=\{1,3,4\}\\
D(5)&=\{5\}\cup D(4)=\{5\}\cup\{1,3,4\}=\{1,3,4,5\}\\
D(6)&=\{6\}\cup D(4)=\{6\}\cup\{1,3,4\}=\{1,3,4,6\}\\
D(7)&=\{7\}\cup(D(5)\cap D(6)\cap D(10))\\
&=\{7\}\cup(\{1,3,4,5\}\cap\{1,3,4,6\}\cap\{1,2,\cdots,10\})=\{1,3,4,7\}\\
D(8)&=\{8\}\cup D(7)=\{8\}\cup\{1,3,4,7\}=\{1,3,4,7,8\}\\
D(9)&=\{9\}\cup D(8)=\{9\}\cup\{1,3,4,7,8\}=\{1,3,4,7,8,9\}\\
D(10)&=\{10\}\cup D(8)=\{10\}\cup\{1,3,4,7,8\}=\{1,3,4,7,8,10\}
\end{aligned}$$

因为第二次迭代时 [图 9.17(a) 中程序的第 3~6 行的外循环]，这些值都不变，所以上述这些值就是该支配集问题的结果。

9.6.2 回边和可归约性

深度优先搜索是一种遍历图中所有结点的算法之一。它从图的起点开始，并优先深入访问离起点最远的结点。深度优先搜索的搜索路径形成一棵深度优先生成树。由数据结构课程可知，先序遍历是指在访问某结点时，优先访问该结点本身，然后按从左到右的顺序递归访问它的子结点；后序遍历则是先递归地按从左到右的顺序访问结点的子结点，然后访问结点本身。

对于程序流图来说，深度优先排序是一种重要的排序方式，它是后序遍历的逆序。在深度优先排序中，首先访问某个结点，然后依次从它的最右子结点向左遍历，直到最左子结点。在构建流图的深度优先生成树之前，需要对流图中的每个结点的后继关系进行定义，确定哪些后继结点依次被视为该树的最右子结点、次右子结点，直至最左子结点。

【例 9.26】 图 9.28 所示流图的一种深度优先表示如图 9.29 所示。图中实线边形成相应的树，虚线边是该流图的其他边。该树的深度优先遍历由 $1\to3\to4\to6\to7\to8\to10$ 开始，然后回到结点 8，进到结点 9；再次回到结点 8，继续回到结点 7、6、4，再进到结点 5；从结点 5 回到结点 4，继续回到结点 3 和 1，进到结点 2；最后回到结点 1，完成了整个树的遍历。

先序遍历的结点序列是 1, 3, 4, 6, 7, 8, 10, 9, 5, 2，后序遍历的结点序列是 10, 9, 8, 7, 6, 5, 4, 3, 2, 1。

深度优先排序是后序遍历序列的逆序，即 1, 2, 3, 4, 5, 6, 7, 8, 9, 10。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第392页 文件编号: 422624 页面编号: 33924538441 打印人: 8685 2026-03-09 10:23:05

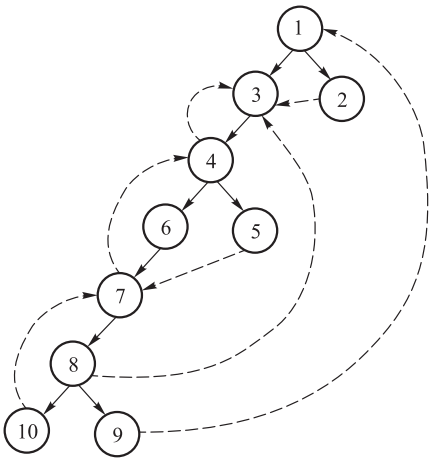


图 9.29 图 9.28 流图的深度优先表示

在构造一个流图的深度优先生成树时，该流图的边分成 3 类。

有关寻找图的深度优先生成树和深度优先排序的算法，可以参考数据结构教材。图 9.29 所示的深度优先生成树就是使用这样的算法得到图 9.28 的结果。

（1）如果 n 在深度优先生成树上是 m 的真后代，那么从结点 m 到 n 的边称为**前进边**（advancing edge）。在深度优先生成树上的所有边都是前进边。图 9.29 没有其他的前进边。但是，如果有边 $4 \to 8$，它就是前进边。

（2）如果 n 在深度优先生成树上是 m 的祖先（包括 m 本身），那么从结点 m 到 n 的边称为**后撤边**（retreating edge）。例如，在图 9.29 中，$4 \to 3$、$7 \to 4$、$10 \to 7$、$8 \to 3$ 和 $9 \to 1$ 都是后撤边。

（3）如果 m 和 n 在深度优先生成树上互不为对方的祖先，那么从结点 m 到 n 的边称为**交叉边**（crossing edge）。在图 9.29 中，$2 \to 3$ 和 $5 \to 7$ 是仅有的交叉边。交叉边的一个重要性质是，如果在画深度优先生成树时，一个结点的子结点从左到右画出的次序与它们加到该树上的次序一样，则所有的交叉边都从右到左。

如果 b 支配 a，那么流图的边 $a \to b$ 称为**回边**（back edge）。对任何流图，回边一定是后撤边，但后撤边不一定是回边。如果在一个流图任何深度优先生成树上，所有的后撤边都是回边，就称为**可归约**的。换句话说，如果一个流图可归约，那么它所有的深度优先生成树都有同样的后撤边，并且它们正好就是流图的回边。如果流图不可归约，在它的任何深度优先生成树上，回边都是后撤边，但每个深度优先生成树上都可能有不是回边的后撤边。这些后撤边可能因深度优先生成树的不同而不同。因此，如果把一个流图中所有回边都删掉后，剩余的图有环，则原来的流图是不可归约的，反之也成立。

实际出现的流图几乎都是可归约的。程序若仅使用结构化控制流语句，如 if-then-else、while-do、continue 和 break 语句等，其中间代码对应的流图都是可归约的。甚至使用 goto

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第393页 文件编号: 422624 页面编号: 33934538449 打印人: 8685 2026-03-09 10:23:05

语句的程序，只要程序员是合乎逻辑地基于循环和分支来思考的，它们就是可归约的。

图 9.28 所示的流图是可归约的，该图中的后撤边都是回边。

【例 9.27】 考虑图 9.30 所示的流图，它的起点是 1。结点 1 支配结点 2、3，但结点 2、3 相互不支配，因此该流图没有回边。该图有两种可能的深度优先生成树：一种情况下边 $3 \to 2$ 是后撤边但不是回边；另一种情况下边 $2 \to 3$ 是后撤边但不是回边。直观上说，它不可归约的原因是由结点 2、3 构成的循环可以从两个不同的地方（结点 2、3）进入。

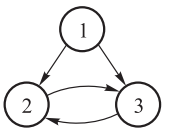


图 9.30 非归约流图

9.6.3 流图的深度

给定一个流图的一棵深度优先生成树，其深度就是在任何可能无环路径上的最大后撤边数。可以证明，深度决不会大于直观上认为的该流图中循环嵌套的层数。如果流图是可归约的，则可以用回边取代后撤边来定义深度，因为在任何深度优先生成树中，后撤边正好就是回边。这时，深度的概念就独立于所选择的深度优先生成树，即指流图的真正深度，而不是流图联系到某棵深度优先生成树的深度。

例如，在图 9.29 中，流图的深度是 3，因为存在一条路径

$$10 \to 7 \to 4 \to 3$$

它有 3 条后撤边。该图不存在含更多后撤边的无环路径。在这里最深路径碰巧仅含后撤边；一般情况下，最深路径可能是后撤边、前进边和交叉边的混合。

9.6.4 自然循环

在源语言一级，循环可以用多种方式来表达，如 for 循环、while 循环和 repeat 循环，甚至可以用标号和 goto 语句方式。从程序分析的观点上看，循环的具体编写方式并不是关键，重要的是它们是否具备易于分析的特性。特别需要关注的是，循环是否仅有一个入口结点。如果满足这一条件，编译器的优化在分析时可以假设，每次循环迭代开始时都会满足某些特定的初始条件。这种特点引出了“自然循环”的概念。

自然循环通过以下两个关键性质来定义：

（1）循环必须有唯一的入口结点，称为**首结点**。首结点支配该循环中的所有结点；否则，它就不是该循环的唯一入口。

（2）至少存在一条回边进入该循环的首结点；否则，控制流不可能从该“循环”回到首结点，即这种结构根本就不构成一个循环。

给定一条回边 $n \to d$，该回边确定的自然循环是 d 加上不经过 d 能到达 n 的所有结点。结点 d 是该循环的首结点。

算法 9.7 构造回边的自然循环。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第394页 文件编号: 422624 页面编号: 33944538447 打印人: 8685 2026-03-09 10:23:05

输入：流图 G 和回边 $n \to d$。

输出：由回边 $n \to d$ 确定的自然循环中所有结点的集合 loop。

方法：令 loop 初值是 $\{n, d\}$。标记 d 为“已访问”，以保证搜索不会超出 d。从结点 n 开始，完成对流图 G 的逆向流图的深度优先搜索，把搜索过程中访问的所有结点都加入 loop。该过程找出不经过 d 能到达 n 的所有结点。

【例 9.28】 图 9.28 所示的流图有 5 条回边：$10 \to 7$、$7 \to 4$、$4 \to 3$、$8 \to 3$ 和 $9 \to 1$。这些正好是直观上认为该流图中形成循环的边。

回边 $10 \to 7$ 确定自然循环 {7, 8, 10}，因为结点 8、10 是不经过结点 7 能到达结点 10 的仅有结点。回边 $7 \to 4$ 确定自然循环 {4, 5, 6, 7, 8, 10}，它包含了自然循环 {7, 8, 10}，因此相对来说，{7, 8, 10} 被认为是内循环。

回边 $4 \to 3$ 和 $8 \to 3$ 确定的循环有同样的首结点，并且碰巧有同样的结点集合 {3, 4, 5, 6, 7, 8, 10}，显然应该把它们看成一个循环。上面提到的两个循环是该循环的内循环。

最后，回边 $9 \to 1$ 确定的自然循环是整个流图，并且它是最外循环。在该例中，4 个循环正好依次一个嵌套在下一个的里面。然而，两个循环相互不嵌套才是更经常的情况。

在归约流图中，因为所有的后撤边都是回边，可以把自然循环联系到后撤边来定义。但这句话对非归约流图不成立。例如，在图 9.30 所示的非归约流图中，结点 2 和结点 3 构成一个环，但它们都不是回边，因此该环不符合自然循环的定义。不把该环标识为自然循环，当然也就不会对它进行循环优化。这样做是可接受的，因为假定所有循环都只有一个入口可以简化循环分析，而非归约流图在实际中又很少出现。

由于只把自然循环作为“循环”，因此流图还有这样的性质：除非两个循环有同样的首结点，否则它们要么不相交，要么一个嵌套在另一个中。

如果两个循环有同样的首结点，如图 9.31 所示，这时很难说哪个是内循环。因此当两个循环有同样的首结点，而且并非一个包含另一个时，把它们合在一起看成一个循环。

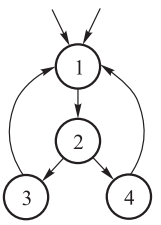


图 9.31 有相同首结点的两个循环

【例 9.29】 在图 9.31 中，回边 $3 \to 1$ 和 $4 \to 1$ 确定的自然循环分别是 {1,2,3} 和 {1,2,4}。它们被合成一个循环 {1,2,3,4}。

如果该图还有另一条回边 $2 \to 1$，那么它确定的自然循环是 {1,2}，结点 1 成为第三个循环的首结点。{1,2} 真包含于 {1,2,3,4}，因此 {1,2} 不合并到自然循环 {1,2,3,4} 中，而是作为一个内循环来处理。 □

9.6.5 迭代流图算法的收敛速度

本节讨论迭代算法的收敛速度。如 9.3.3 小节所讨论的那样，该算法迭代的最大次数可以取格的高度和流图中结点数的积。对许多数据流分析，通过给结点的计算排序，使得该算法以一个小得多的迭代次数收敛是可能的。所关心的性质是，在一个结点处所有有意义的结

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第395页 文件编号: 422624 页面编号: 33954538444 打印人: 8685 2026-03-09 10:23:05

果是否都可以通过无环路径传播到另一个结点。在讨论到目前为止的数据流分析中，到达–定值、可用表达式和活跃变量都有这个性质，但是常量传播没有这个性质。具体来说，分为以下 3 种情况：

（1）如果定值 d 在 IN[B] 中，那么存在某条从包含 d 的块到块 B 的无环路径，且 d 在这条路径上所有的 IN 和 OUT 中。

（2）如果表达式 x + y 在块 B 的入口不是可用的，那么存在一条从起点开始的无环路径，该路径上没有产生 x + y，或者存在一条从注销 x + y 的某块开始的无环路径，该路径随后没有产生 x + y。

（3）如果 x 在某块 B 的出口是活跃的，那么存在一条从块 B 开始到 x 的一个引用的无环路径，在这条路径上没有 x 的定值。

对上面这 3 种情况都必须检查，带环的路径没有加入任何东西。例如，如果 x 的一个引用是它的某个定值 d 沿带环的路径可以到达，那么 d 沿着无这些环的相应路径也可以到达。

相反，常量传播没有这个性质。考虑只有一个基本块且该块单独构成一个循环的简单程序：

```
L: a = b
   b = c
   c = 1
   goto L
```

第一次访问该基本块时，发现 c 有常量值 1，但是 a 和 b 都没有定值；第二次访问该块时，发现 b 和 c 都有常量值 1；通过对该块的 3 次访问，给 c 赋的常量值 1 到达 a。

如果所有有用信息都沿无环路径传播，就有机会在迭代数据流算法中选择合适的结点访问次序，使得可以用相对少的遍数来保证信息已经通过所有无环路径。

从 9.6.2 节可以知道，只有边 $a \to b$ 是一条后撤边时，b 的深度优先序号小于 a 的深度优先序号。对于正向数据流问题，按照深度优先排序来访问结点是所希望的。具体来说，将图 9.17(a) 中算法第 (4) 行（访问流图中各基本块）替换为

for (除 ENTRY 以外，按深度优先次序的每个块 B) {

【例 9.30】 假如定值 d 沿一条路径传播，如

$$3 \to 5 \to 19 \to 35 \to 16 \to 23 \to 45 \to 4 \to 10 \to 17$$

其中整数代表这条路径上基本块的深度优先序号。按照图 9.17(a) 的算法，第一次通过第 (4)∼(6) 行的循环时，d 从 OUT[3] 到 IN[5]，再到 OUT[5]，等等，如此最后到 OUT[35]。在这一遍 d 到达不了 IN[16]，因为 16 先于 35，在 d 被加入 OUT[35] 时，IN[16] 已经计算过了。第二次通过第 (4)∼(6) 行的循环时，d 将被包含在 IN[16] 中，因为它在 OUT[35] 中，定值 d 传播到 OUT[16]，继续到 IN[23]，等等，最后一直到 OUT[45]，与第一遍一样，在这里它必须等待，因为 IN[4] 在这一遍已经计算过了。第三次通过第 (4)∼(6) 行的循环时，d 从 IN[4] 到 OUT[4]、IN[10]、OUT[10]，最后到 IN[17]。这样，经过 3 遍扫描，d 到达块 17。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第396页 文件编号: 422624 页面编号: 33964538442 打印人: 8685 2026-03-09 10:23:05

从这个例子不难总结出一般原则。如果图 9.17(a) 的算法使用深度优先次序，那么沿任何无环路径传播任何定值需要的遍数不会超过后撤边数加 1，即流图的深度加 1。当然，在发现新一遍扫描没有引起任何变化前，算法 9.4 并没有去判断所有的定值是否到达了它们可以到达的地方。因此，使用深度优先排序的算法 9.4，其迭代遍数的上界是深度加 2。研究表明，典型流图的平均深度约为 2.75。因此该算法会很快收敛。

对于逆向问题，如活跃变量分析问题，则以深度优先次序的逆序来访问结点。因此，块 17 中一个变量引用沿路径

$$3 \to 5 \to 19 \to 35 \to 16 \to 23 \to 45 \to 4 \to 10 \to 17$$

逆向第一遍到 IN[4]，第二遍到 IN[16]，第三遍到 IN[3]。同样，采用了深度优先次序的逆序后，沿任何无环路径传播任何变量引用需要的遍数不会超过流图的深度加 1。

到目前为止，描述的上界是一类问题的上界，这类问题的特点是，有环路径没有为分析加入任何信息。对于一些特别问题，如支配集计算，算法收敛得更快。在这个问题中，如果采用一个按深度优先排序访问结点的数据流算法，那么当输入流图可归约时，每个结点的支配集可以在第一次迭代时得到。如果不知输入流图是否可归约，那么需要一次额外的迭代来确定是否已经收敛。

习 题

9.1 对于图 9.32 所示的流图：

(a) 识别该流图的循环。

(b) 块 B_1 中的语句 (1)、(2) 都是复写语句，并且它们给 a 和 b 赋的都是常量。可以对 a 和 b 的哪些引用实施复写传播并将这些引用替换成对常量的引用？

(c) 识别每个循环的全局公共子表达式。

(d) 识别每个循环的归纳变量，不要忘记把 (b) 的复写传播引入的常量考虑进去。

(e) 识别每个循环的循环不变计算。

9.2 为图 9.33 计算向量 A 和 B 点积的中间代码完成下列优化：删除公共子表达式、归纳变量上的强度削弱和尽量删除归纳变量。

9.3 对图 9.32 所示的流图，计算：

(a) 为到达-定值分析，计算每个块的 gen、kill、IN 和 OUT 集合；

(b) 为可用表达式分析，计算每个块的 $e_$gen、$e_$kill、IN 和 OUT 集合；

(c) 为活跃变量分析，计算每个块的 def、use、IN 和 OUT 集合。

***9.4** 通过对算法 9.1 第 4~6 行 for 循环的迭代次数的归纳来证明，IN 和 OUT 集合不会缩小，即一旦某个定值在某轮迭代中被加入，则在随后的继续迭代中决不会消失。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第397页 文件编号: 422624 页面编号: 33974538440 打印人: 8685 2026-03-09 10:23:05

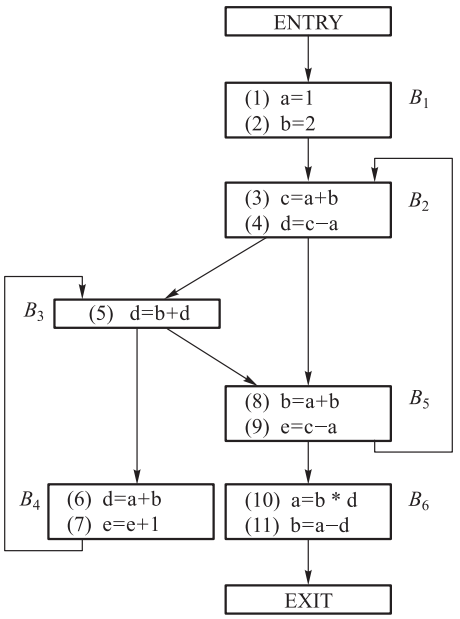


图 9.32 流图

```
        dp=0
        i=0
L:      t1=i*8
        t2=A[t1]
        t3=i*8
        t4=B[t3]
        t5=t2*t4
        dp=dp+t5
        i=i+1
        if i<n goto L
```

图 9.33 计算点积的中间代码

***9.5** 证明算法 9.1 的正确性，即证明：

(a) 如果定值 d 加入 IN[B] 或 OUT[B]，那么存在一条从 d 开始分别到块 B 入口或出口的路径，d 定值的变量在该路径上没有重新定值；

(b) 如果定值 d 没有加入 IN[B] 或 OUT[B]，那么不存在任何从 d 开始分别到 B 入口或出口的路径，d 定值的变量在该路径上没有重新定值。

***9.6** 对算法 9.2 证明：

(a) IN 和 OUT 集合决不会缩小；

(b) 如果变量 x 加入 IN[B] 或 OUT[B]，那么存在一条分别从 B 入口或出口开始的路径，x 在该路径上被引用；

(c) 如果变量 x 没有加入 IN[B] 或 OUT[B]，那么不存在分别从 B 入口或出口开始的任何路径，x 在该路径上被引用。

***9.7** 对算法 9.2 证明：

(a) IN 和 OUT 集合决不会扩大，即在迭代过程中，这些集合后续的值是先前值的子集（并非一定是真子集）；

(b) 如果表达式 e 从 IN[B] 或 OUT[B] 中删除，那么存在一条从流图起点分别到块 B 入口或出口的路径，e 在该路径上没有计算，或者在最后一次计算后它的某个运算对象重新定值；

(c) 如果表达式 e 一直留在 IN[B] 或 OUT[B] 中，那么对于从流图起点分别到块 B 入口

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第398页 文件编号: 422624 页面编号: 33984538448 打印人: 8685 2026-03-09 10:23:05

或出口的任何一条路径，e 在该路径上被计算，并且在最后一次计算后，它的任何运算对象都没有重新定值。

9.8 假定 V 是复数集合。下面哪个运算可以作为 V 上一个半格的汇合运算？

(a) 加：$(a+\mathrm{i}b)\wedge(c+\mathrm{i}d)=(a+b)+\mathrm{i}(c+d)$。

(b) 乘：$(a+\mathrm{i}b)\wedge(c+\mathrm{i}d)=(ac-bd)+\mathrm{i}(ad+bc)$。

(c) 取较小成员：$(a+\mathrm{i}b)\wedge(c+\mathrm{i}d)=\min(a,c)+\mathrm{i}\ \min(b,d)$。

(d) 取较大成员：$(a+\mathrm{i}b)\wedge(c+\mathrm{i}d)=\max(a,c)+\mathrm{i}\ \max(b,d)$。

9.9 在 9.3 节中提到，如果块 B 由 n 条语句组成，并且每条语句的产生和注销集合分别是 gen_i 和 $kill_i$，那么块 B 迁移函数的产生和注销集合 gen_B 和 kill_B 由下面的等式给出

$$\mathrm{kill}_B=\mathrm{kill}_1\cup\mathrm{kill}_2\cup\cdots\cup\mathrm{kill}_n$$

和

$$\begin{aligned}\mathrm{gen}_B=\mathrm{gen}_n\cup(\mathrm{gen}_{n-1}-\mathrm{kill}_n)\cup(\mathrm{gen}_{n-1}-\mathrm{kill}_{n-1}-\mathrm{kill}_n)\cup\cdots\\ \cup(\mathrm{gen}_1-\mathrm{kill}_2-\mathrm{kill}_3-\cdots-\mathrm{kill}_n)\end{aligned}$$

基于对 n 的归纳证明上面的等式。

9.10 每个定值 $d_i(i=1,2,3)$ 构成一个格，为这 3 个格的积构造格图。所构造的格图和图 9.16 所示的格图有什么联系？

9.11 在 9.3.3 小节提到，如果框架是有限高度，则迭代算法收敛。本题是高度无限且迭代算法不收敛的例子。令集合 V 是非负实数集合，并且汇合运算是取极小值。3 个迁移函数为恒等函数 $f_I(x)=x$、取半函数 $f_H(x)=x/2$、恒等于 1 的常函数 $f_O(x)=1$。

迁移函数集合 F 由上述 3 个函数加上它们的任意复合构成。

(a) 描述集合 F。

(b) 在该框架中，$\preceq$ 关系是什么？

(c) 给出一个带指定迁移函数的流图，使得算法 9.4 不收敛。

(d) 该框架单调吗？它是否可分配？

9.12 假定想检查到达某变量一个引用点的所有路径中是否存在未初始化该变量的路径，试问应该怎样修改 9.4 节的框架。

9.13 对图 9.34 所示的流图：

(a) 为每个基本块的入口和出口计算 anticipated。

(b) 为每个基本块的入口和出口计算 available。

(c) 计算每个块的 earliest。

(d) 为每个基本块的入口和出口计算 postponable。

(e) 为每个基本块的入口和出口计算 used。

(f) 计算每个块的 latest。

(g) 引入临时变量 t，并说明它在哪里定值和在哪里引用。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第399页 文件编号: 422624 页面编号: 33994538446 打印人: 8685 2026-03-09 10:23:05

9.14 为图 9.32 重复 9.13。可以把你的分析限制到表达式 a + b、c − a 和 b * d。

9.15 对图 9.32 所示的流图：

(a) 计算支配关系；

(b) 找出一种深度优先排序；

(c) 对 (b) 的结果，标明前进边、后撤边和交叉边；

(d) 该流图是否可归约；

(e) 计算该流图的深度；

(f) 找出该流图的自然循环。

9.16 对图 9.3 所示的流图重复 9.15。

9.17 对图 9.35 所示的流图重复 9.15。

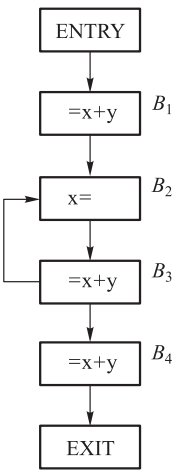


图 9.34 9.13 的流图

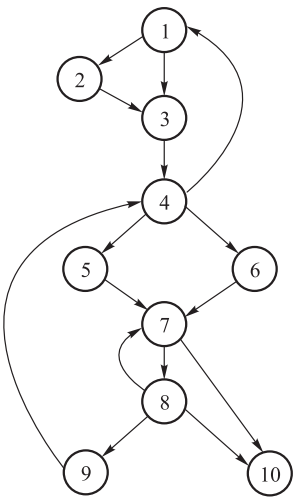


图 9.35 9.17 的流图

9.18 一个 C 语言程序如下：

```
1  void main() {
2      int i, j, k;
3      i=5;
4      j=1;
5      while(j<100){
6          k=i+1;
7          j=j+k;
8      }
9  }
```

在 x86/Linux 系统上经优化编译后（编译器版本见汇编代码的最后一行），生成的代码如下：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第400页 文件编号: 422624 页面编号: 34004538442 打印人: 8685 2026-03-09 10:23:05


```
    .file "optimize.c"
    .version  "01.01"
gcc2_compiled.:
.text
    .align 4
.globl main
    .type  main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    movl $1,%eax
    movl $6,%edx
    .p2align 4,,7
.L4:
    addl %edx,%eax
    cmpl $99,%eax
    jle .L4
    leave
    ret
.Lfe1:
    .size  main,.Lfe1-main
    .ident  "GCC: (GNU) egcs-2.91.66
        19990314/Linux (egcs-1.1.2
        release)"
```

试说明编译器对这个程序进行了哪些种类的优化（只需要说明复写传播、删除公共子表达式等，不需要具体说明怎样完成这些优化的）。

***9.19** 下面是用 C 语言编写的求最大公约数的函数：

```
long gcd(p, q) long p, q; {
    if (p%q == 0)
        return q;
    else
        return gcd(q, p%q);
}
```

其中，递归调用称为尾递归（return 后的表达式是一个递归调用，而其他地方没有递归调用）。对于尾递归，编译器可以产生和一般的函数调用不同的代码，使得目标程序运行时，递归调用所需的存储空间大大减少，也缩短了运行时间。对于尾递归，编译器应怎样产生代码，简述你的想法。（若用源语言一级的优化来回答此问题，则不符合题目要求。）

9.20 C 语言程序

```
void main() {
    long i, j;
    while ( i ) {
        if ( j ) { i = j; }
    }
}
```

的编译结果如下：

```
    .file "control.c"
    .version  "01.01"
gcc2_compiled.:
.text
    .align 4
.globl main
    .type  main,@function
main:
    pushl %ebp
    movl %esp,%ebp
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第401页 文件编号: 422624 页面编号: 34014538440 打印人: 8685 2026-03-09 10:23:05


```
    subl $8,%esp
    nop
    .p2align 4,,7
.L2:
    cmpl $0,-4(%ebp)
    jne .L4
    jmp .L3
    .p2align 4,,7
.L4:
    cmpl $0,-8(%ebp)
    je .L5
    movl -8(%ebp),%eax
    movl %eax,-4(%ebp)
.L5:
    jmp .L2
    .p2align 4,,7
.L3:
.L1:
    leave
    ret
.Lfe1:
    .size  main,.Lfe1-main
    .ident  "GCC: (GNU) egcs-2.91.66
        19990314/Linux (egcs-1.1.2
        release)"
```

它的优化编译结果如下：

```
    .file "control.c"
    .version  "01.01"
gcc2_compiled.:
.text
    .align 4
.globl main
    .type  main,@function
main:
    pushl %ebp
    movl %esp,%ebp
.L7:
    testl %eax,%eax
    je .L3
    testl %edx,%edx
    je .L7
    movl %edx,%eax
    jmp .L7
    .p2align 4,,7
.L3:
    leave
    ret
.Lfe1:
    .size  main,.Lfe1-main
    .ident  "GCC: (GNU) egcs-2.91.66
        19990314/Linux (egcs-1.1.2
        release)"
```

请分析优化编译器所做的控制流优化。

9.21 UNIX 下的 C 编译命令 cc 的选项 g、O 的解释如下，其中 dbx 的解释是“dbx is an utility for source-level debugging and execution of programs written in C”。试说明为什么用了选项 g 后，选项 O 便被忽略。

-g Produce additional symbol table information for dbx(1) and dbxtool(1) and pass -lg option to ld(1) (so as to include the g library, that is: /usr/lib/libg.a). When this option is given, the -O and -R options are suppressed.

-O[level] Optimize the object code. Ignored when either -g, -go, or -a is used. ...

9.22 请利用代码优化的思想（代码外提和强度削弱等），改写下面 C 语言程序中的循环，得到优化后的 C 语言程序。

```
void main() {
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第402页 文件编号: 422624 页面编号: 34024538448 打印人: 8685 2026-03-09 10:23:05


```
    int i, j;
    int r[20][10];

    for(i=0; i<20; i++){
        for(j=0; j<10; j++){
            r[i][j] = 10 i j;
        }
    }
}
```

9.23 C 语言程序引用 sizeof（求字节数运算符）时，该运算是在编译该程序时完成的，还是在运行该程序时完成的？说明理由。

9.24 某优化编译器对下面程序的局部变量 i 和 j 不分配空间，为什么？

```
void main() {
    long i, j;
    i = 5;
    j = i * 2;
    printf("%d\n", i+j);
}
```

9.25 某 C 语言的函数如下：

```
func(i) long i; {
    long j;
    j = i - 1;
    func(j);
}
```

下面左右两边的汇编代码是两个不同版本 GCC 编译器为该函数产生的代码。左边的代码在调用 func() 之前将参数压栈，调用结束后将参数退栈（见例 6.4）。右边的代码对参数传递的处理方式没有实质区别。请叙述右边代码对参数传递的处理方式并推测它所带来的优点。

```
func:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $4, %esp
    movl    8(%ebp), %edx
    decl    %edx
    movl    %edx, -4(%ebp)
    movl    -4(%ebp), %eax
    pushl   %eax
    call    func
    addl    $4, %esp
    leave
    ret
```

```
func:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    movl    8(%ebp), %eax
    decl    %eax
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第403页 文件编号: 422624 页面编号: 34034538446 打印人: 8685 2026-03-09 10:23:05


```
7       movl    %eax, -4(%ebp)
8       movl    -4(%ebp), %eax
9       movl    %eax, (%esp)
10      call    func
11      leave
12      ret
```

9.26 考虑一个类 Pascal 的语言，其中所有的变量都是整型（不需要显式声明），并且仅包含赋值语句、读语句、写语句、条件语句和循环语句。下面的产生式定义了该语言的语法（其中 lit 表示整型常量；OP 的产生式没有给出，因为它和所讨论的问题无关）。

Program → Stmt
Stmt → **id** := Exp
Stmt → read (**id**)
Stmt → write (Exp)
Stmt → Stmt ; Stmt
Stmt → **if** (Exp) **then begin** Stmt **end else begin** Stmt **end**
Stmt → **while** (Exp) **do begin** Stmt **end**
Exp → **id**
Exp → **lit**
Exp → Exp **OP** Exp

定义 Stmt 的两个属性：Def 表示在 Stmt 中一定会定值且在该定值前没有引用的变量集合；MayUse 表示在 Stmt 中有引用且在该引用前可能没有定值的变量集合。

(a) 写一个语法制导定义或翻译方案，它计算 Stmt 的 Def 和 MayUse 属性。

(b) 基于上面的计算，程序可能未赋初值的变量集合从哪儿可以得到？

可能未赋初值的变量是这样定义的：若存在从程序开始点到达变量 a 的某引用点的一条路径，在这条路径上没有对变量 a 赋值，则变量 a 属于程序可能未赋初值的变量集合。

9.27 考虑一种简单语言，其中所有的变量都是整型（不需要显式声明），并且仅包含赋值语句、读语句和写语句。下面的产生式定义了该语言的语法（其中 **lit** 表示整型常量；**OP** 的产生式没有给出，因为它和所讨论的问题无关）。

Program → StmtList
StmtList → Stmt StmtList | Stmt
Stmt → **id** := Exp; | read (**id**); | write (Exp);
Exp → **id** | **lit** | Exp **OP** Exp

把不影响 write 语句输出值的赋值（包括通过 read 语句来赋值）称为无用赋值，写一个语法指导定义，它确定一个程序中出现过赋予无用值的变量集合（不需要知道无用赋值的位置）和没有置初值的变量集合（不影响 write 语句输出值的未置初值变量不在考虑之中）。

非终结符 StmtList 和 Stmt 用下面 3 个属性（用户根据需要来定义其他文法符号的属性）：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第404页 文件编号: 422624 页面编号: 34044538444 打印人: 8685 2026-03-09 10:23:05

(1) uses_in：在本语句表或语句入口点的引用变量集合，其值影响在该程序点后的输出。

(2) uses_out：在本语句表或语句出口点的引用变量集合，其值影响在该程序点后的输出。

(3) useless：本语句表或语句中出现的无用赋值变量集合。

9.28 下面左边的函数被 GCC: (GNU) 3.3.5 (Debian 1:3.3.5-13) 优化成右边的代码。若认为该优化结果不对或该优化不合理，则阐述理由。若认为该优化结果是对的，则说明实施了哪些优化，并解释循环优化结果的合理性。

```
1
2  void f(int a, int b, int c, int d, int
           x, int y, int z) {
3      while (a < b) {
4          if (c < d) {
5              x = x + z;
6          } else {
7              x = y - z;
8          }
9      }
10 }
```

```
1  f:
2      pushl   %ebp
3      movl    %esp, %ebp
4      movl    8(%ebp), %edx
5      movl    12(%ebp), %eax
6      cmpl    %eax, %edx
7      jge     .L9
8  .L7:
9      jl      .L7
10 .L9:
11     popl    %ebp
12     ret
```

9.29 下面左栏的 C 语言程序计算 5!，结果是 120。经优化编译器（编译器版本见汇编代码的最后一行）生成的汇编程序跟在源程序的后面（先左栏后右栏），其中在省略号处略去了 main 函数的汇编程序。请问：优化编译器对 factorial 函数进行了怎样的优化？

C 语言程序：

```
1  #include <stdio.h>
2  long factorial(long n, long res) {
3      if (n == 0)
4          return res;
5      else
6          return factorial(n-1, n * res)
           ;
7  }
8  void main() {
9      printf("%ld\n", factorial(5,1));
10 }
```

汇编程序：

```
1      .file "recursion.c"
2      .text
3      .p2align 4,,15
4      .globl factorial
5      .type factorial, @function
6  factorial:
7      pushl   %ebp
8      movl    %esp, %ebp
9      movl    8(%ebp), %edx
10     movl    12(%ebp), %eax
11     testl   %edx, %edx
12     je      .L3
13     .p2align 4,,7
14 .L6:
15     imull   %edx, %eax
16     subl    $1, %edx
17     jne     .L6
18 .L3:
19     popl    %ebp
20     ret
21     .size factorial, .-factorial
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第405页 文件编号: 422624 页面编号: 34054538441 打印人: 8685 2026-03-09 10:23:05


```
22      ...
23      .ident "GCC: (GNU) 4.2.3 (Debian
            4.2.3-5)"
24      .section .note.GNU-stack, "",
            @progbits
```

9.30 对于例 6.4 的程序（见下面的 C 代码），用编译器 GCC(GNU) 4.2.3 (Debian 4.2.3-5) 编译后所生成的优化代码（见下面右栏的汇编代码）与例 6.4 给出的代码（见下面左栏的汇编代码）略有不同。请指出该较新版本编译器完成的优化（或采用的不同代码生成策略）。

```
1  func(i) long i; {
2      long j; j=i-1; func(j);
3  }
```

```
1  func:
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $4, %esp
5      movl    8(%ebp), %edx
6      decl    %edx
7      movl    %edx, -4(%ebp)
8      movl    -4(%ebp), %eax
9      pushl   %eax
10     call    func
11     addl    $4, %esp
12 .L1:
13     leave
14     ret
```

```
1  func:
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $4, %esp
5      movl    8(%ebp), %eax
6      subl    $1, %eax
7      movl    %eax, (%esp)
8      call    func
9      leave
10     ret
```

提示：若把调用语句改成 func(j, j)，则所生成的优化代码和右栏的汇编代码的区别说明如下。

（1）右栏第 4 行改为 subl $8, %esp。

（2）右栏第 7 行和第 8 行指令之间增加 movl %eax, 4(%esp)。

***9.31** 下面的 C 程序分别经非优化编译和二级以上（含二级）的优化编译后，生成的两个目标程序运行时的表现不同（编译器是 GCC: (GNU) 4.2.3 (Debian 4.2.3-5)）。请回答它们运行时的表现有何不同，并说明原因。

```
int f(int g( ) ) {
    return g(g);
}
void main() {
    f(f);
}
```

*第 10 章

依赖于机器的代码优化

现代高性能处理器能够在一个指令周期内执行几个操作。在这种指令级并行的机器上，程序的运行能快到什么程度？答案是依赖于下面几个因素：

（1）程序中潜在的并行；

（2）处理器上可用的并行；

（3）从串行程序提取并行的能力；

（4）在给定的调度约束下发现最佳并行调度的能力。

如果程序中的所有运算高度地相互依赖，那么不管是硬件还是并行化技术都不能使程序并行地快速运行。典型的非数值应用有许多固有的相关性。例如，这样的程序有许多依赖于数据的分支，它们使预测将要被执行的指令都变得非常困难，更不要说确定可以并行执行的操作。因此，在该领域的研究集中在放宽调度约束方面，包括引入新的体系结构特征，而不是调度技术本身。

数值应用，如科学计算和信号处理，一般有更多的并行。这些应用处理大批量数据，这些数据不仅具有一定的数据结构，而且该结构中每个元素上的运算相互独立，因而在不同元素上的运算可以并行执行。高性能通用计算机和数字信号处理器上都有附加的硬件资源，可以利用这种并行。该领域的程序一般控制结构比较简单和数据处理模式规整，并且许多从这类程序中提取并行的静态技术已经研发出来。这类应用的代码调度非常重要且有意义，因为这些应用提供了可以映射到大量计算资源上的大量独立运算。

并行的提取和并行执行的调度都可以静态地在软件中或动态地在硬件中完成。事实上，即使机器有硬件调度，它也可以从软件调度中得到帮助。本章从解释使用指令级并行的基础问题开始（忽略该并行是由软件还是硬件管理的），然后讨论提取并行的数据相关性分析，接下来给出代码调度的基本概念。随后，本章描述基本块调度的技术、应对通用程序中高度数据相关的控制流的方法和用于调度数值程序的软件流水线技术。

10.6 节简要介绍使用数组的计算密集型程序在多处理器系统上的优化问题。许多科学、工程和商业应用对计算周期的渴求有无法满足的需求，如天气预报和设计药物的蛋白质折叠。加速计算的一种方式是采用并行。但是，研发利用并行机特点的软件并不容易。把一个大型

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第407页 文件编号: 422624 页面编号: 34074538447 打印人: 8685 2026-03-09 10:23:05

计算任务分成若干在不同处理器上并行执行的计算单元是一件相当有难度的事情，而且它不能保证一定得到加速的效果。另外，还必须考虑让处理器之间的通信最少，因为通信开销很容易使并行代码运行得比串行执行还要慢。

最小化通信问题可被看成改进程序**数据局部性**的一种特殊情况。简单地说，处理器如果经常访问它最近使用过的数据，则认为相应程序有良好的数据局部性。可以肯定地说，并行机上的一个处理器如果有良好的数据局部性，则它和其它处理器之间不需要频繁通信。因此，并行和数据局部性需要联合考虑。另外，数据局部性对于一个内存分层的现代处理器的性能来说也是重要的，就像 6.5.3 小节讨论的程序局部性那样。10.6 节概述并行化和数据局部性优化的概念和方法，不介绍具体实现算法。

本章使用 CISC 风格的目标机器，其常用指令的形式如下：

（1）OP 目的，源 1，源 2。

（2）LD 目的，地址。

（3）ST 地址，源。

10.1 处理器体系结构

在考虑指令级并行时，通常将其想象成一个处理器在单个时钟周期内发射几个操作。事实上，在每个时钟周期内发射一个操作是可能的，而指令级并行的获得是通过使用流水线的概念。下面首先解释流水线，然后讨论多指令问题。

10.1.1 指令流水线和分支延迟

实际上，每种处理器，不管是高性能的超级计算机还是普通的计算机，都使用**指令流水线**。在采用指令流水线技术时，当先前的指令仍在继续通过流水线时，每个时钟周期可以取一条新指令。图 10.1 给出一个简单的 5 级指令流水线：取指令（IF）、译码（ID）、执行操作（EX）、访问内存（MEM）和回写结果（WB），显示了指令 i、$i+1$、$i+2$、$i+3$ 和 $i+4$ 同时执行的情况。每行对应一个时钟周期，每列说明每条指令在每个时钟周期内所占用的流水线级。

如果指令的结果数据在随后的指令需要该数据时已经可用，那么处理器可以在每个时钟周期发射一条指令。但是分支指令会引起问题，因为一直到这条指令被取出、译码和执行后，处理器才知道下面执行哪条指令。许多处理器投机地取出并译码分支指令的直接后继指令，但是如果发现应该执行一个分支而不是直接后继时，则投机失败，指令流水线被清空，转向该分支目的地址去取指令。因此，转向一个分支时会引起取分支目标地址指令的延迟并引起指令流水线“打嗝”。高级的处理器通过使用硬件，根据各分支的执行历史来预测选取分支的结果并从预测的目的地址预取指令。即使这样，分支延迟也不可避免，因为分支预测会发生偏差。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第408页 文件编号: 422624 页面编号: 34084538445 打印人: 8685 2026-03-09 10:23:05

	i	i+1	i+2	i+3	i+4
1.	IF				
2.	ID	IF			
3.	EX	ID	IF		
4.	MEM	EX	ID	IF	
5.	WB	MEM	EX	ID	IF
6.		WB	MEM	EX	ID
7.			WB	MEM	EX
8.				WB	MEM
9.					WB

图 10.1 5 级指令流水线中的 5 条连续指令

10.1.2 流水化的执行

有些指令的执行需要几个时钟周期。一个简单的例子是读取内存指令。即使内存访问命中缓存，但通常缓存也需要几个时钟周期才能返回数据。若不依赖某条指令结果的随后指令在其结果产生前就被允许执行，则该条指令的执行被称为流水化的。因此，即使处理器每个时钟周期只能发射一个操作，也可能出现几个操作同时出现在它们的执行级上。如果最长的执行流水线是 n 级，n 个操作同时进行的可能性是存在的。然而，并非所有的指令都被完全流水化。例如，虽然浮点加和乘经常可以完全流水化，但是更加复杂而很少执行的浮点除经常是不行的。

大多数通用处理器动态地察觉相继指令之间的依赖性，并且若它们的操作数不可用，则它仍将自动地延迟指令的执行。某些处理器，特别是嵌入在手持设备中的那些处理器，把数据相关性的检查交给软件，以保持硬件简单且功耗低。在这种情况下，如果要求保证结果在需要时一定可用，那么编译器插入什么也不做的 NOP 指令。

10.1.3 多指令发射

通过每个时钟周期发射几个操作，处理器可以让更多的操作同时进行。同时被执行操作的最大数目可以是指令发射的宽度乘以执行流水线中的平均级数。

像流水线一样，多指令发射机上的并行可以由软件或硬件来管理，依靠软件管理这种并行的机器称为超长指令字机器，而依靠硬件的称为超标量机器。超长指令字机器，正如该名称所暗示的，有比正常指令字更宽的指令字，它能将若干操作编码在单个时钟周期中发射出去，而编译器需要确定哪些操作可以并行发射，并用机器代码显式地编码这些信息。超标量机器有按普通顺序执行语义的正规指令集。超标量机器自动察觉指令之间的相关性，并且在它们的操作数可用时就发射它们。有些处理器同时包括超长指令字和超标量功能。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第409页 文件编号: 422624 页面编号: 34094538443 打印人: 8685 2026-03-09 10:23:05

简单的硬件调度器按指令被取出的顺序执行指令，如果它碰巧遇到相关指令，那么该指令和所有后续指令都必须等待，直到该相关性被解决（也就是所需要的数据可用）为止。这样的机器显然会从安排相互不相关的指令相继执行的静态调度器中获益。

更复杂的代码调度器能够“乱序”执行指令。在这样的机器上，操作会自动地延迟，直到所依赖的所有值都产生时再继续执行。就是这样的调度器也可以从静态调度中获益，因为硬件调度器只有有限的空间来缓冲必须延迟的操作。静态调度可以把相互独立的操作靠近以获得较好的硬件利用率。更重要的是，不管动态调度器怎么复杂，它总不能执行尚未取出的指令。当处理器遇到未预料到的分支跳转时，它只能在新取的指令流中挖掘并行性。编译器可以通过保证这些新取指令能够并行执行来增强动态调度器的性能。

10.2 代码调度的约束

代码调度是一种用于代码生成器产生的机器代码上的程序优化。代码调度受到下面 3 类约束：

（1）控制相关约束：在原程序中执行的所有操作都必须在优化代码中执行。

（2）数据相关约束：优化程序中的操作产生的结果必须与原程序对应操作的结果相同。

（3）资源约束：调度不得过分占用机器的资源。

这些调度约束保证优化程序和原程序产生同样的结果。但是，因为代码调度会改变操作执行的次序，因此某个程序点的内存状态可能与顺序执行的任何内存状态都不匹配。如果程序的执行出现异常或碰到用户设定的断点等情况而被中断，那么这种不相匹配就是一个问题——优化程序很难调试。当然，这个问题不是代码调度专有的，其他的优化，如部分冗余删除和寄存器分配等也会引起同样的问题。

10.2.1 数据相关

很容易明白，如果两个操作不涉及任何相同的变量，那么它们执行次序的改变不会影响结果；甚至它们读同一个变量时，交换它们的次序也无妨。只有当一个操作写由另一个操作读或写的变量时，改变它们的执行次序才会改变它们的结果。这样的一对操作被称为共享一个**数据相关**，它们的相对执行次序必须保持。下面介绍 3 类数据相关：

（1）真相关：如果对同一个单元先写后读，那么这个读依赖于所写的值，这种相关称为真相关。

（2）反相关：如果对同一个单元先读后写，那么存在一个从这个读到这个写的一个反相关。这个写本质上不依赖于这个读，但是如果这个写碰巧先于这个读被执行，那么这个读操作将取到一个错误的值。反相关是命令式语言的副产品，因为在命令式语言中，一个内存单

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第410页 文件编号: 422624 页面编号: 34104538441 打印人: 8685 2026-03-09 10:23:05

元通常用来存储不同的值。反相关不是一种“真”相关，因为它可以通过把值存在不同的单元来删除。

（3）输出相关：如果对同一个单元先后写两次，则称为输出相关。如果该相关性被违反，则在这两个操作都完成后，该内存单元将有一个错误的值。

反相关和输出相关统称为和内存单元联系的相关。它们不是真正的相关，因为可以通过使用不同的单元保存不同值来删除它们。注意，数据相关的概念可同时用于内存访问和寄存器访问。

10.2.2 发现内存访问中的相关性

为了检查两个内存访问是否共享一个数据相关，需要知道它们是否引用同一个单元，但不用知道真正访问哪个单元。例如，可以知道 * p 和 (* p)+4 这两个访问肯定不引用同一个单元，虽然不知道 p 究竟指向哪个单元。一般来说，数据依赖在编译时是不可判定的。因此编译器必须假定不同操作可能引用同一个单元，除非能证明它们引用不同的单元。

【例 10.1】 对于下面的代码序列

(1) a = 1

(2) * p = 2

(3) x = a

除非编译器知道 p 不可能指向 a，否则它只能总结出以上 3 个操作必须串行执行。因为语句 (1) 和 (2) 可能构成一个输出相关，语句 (1) 和 (3) 以及语句 (2) 和 (3) 可能构成两个真相关。

数据依赖分析是和写程序所用的编程语言密切相关的。例如，像 C 和 C++ 这样的非类型可靠语言，它们的指针可以被强制为指向任何种类的对象，因此需要复杂的分析来证明任何一对基于指针的内存访问之间的独立性。这件事甚至还牵扯到局部和全局的标量，因为它们可以被间接地访问，除非能够证明它们的地址没有被程序中的任何指令存放在任何地方。在像 Java 这样的可靠语言中，不同类型的对象必定相互区别。类似地，在栈上的局部简单变量不可能是任何通过其他名字访问的变量的别名。因此，对于 Java 语言，相应的分析要简单得多。

正确发现数据相关需要几种不同形式的分析。下面先简要说明编译器若要找出程序中存在的所有相关则必须解决的主要问题，然后主要介绍怎样把这些信息用到代码调度中。本书不介绍怎样完成这些分析。

1. 数组数据相关分析

数组数据相关分析是指在数组元素的访问中，消除下标表达式值之间的二义。例如，循环

```
for (i = 0; i < n; i++)
    A[2 * i] = A[2 * i + 1];
```

将数组 A 中每个奇元素复写到先于并紧靠它的偶元素。因为在该循环中，读写的单元相互区别，所以在这些访问之间不存在相关，因而该循环中的所有迭代可以并行执行。数组数据相

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第411页 文件编号: 422624 页面编号: 34114538449 打印人: 8685 2026-03-09 10:23:05

关分析（通常简称数据相关分析），在数值应用的优化中是非常重要的。

2. 指针别名分析

在编程语言中，两个不同的名字（包括有左值的表达式）如果代表运行时同一个内存单元，则它们互为**别名**（alias）。别名通常由指针、引用调用、下标变量和 C 语言的联合体等引起的。上面所说的消除下标表达式值之间的二义问题也就是分析两个下标变量是否互为别名问题。如果两个指针的值相等，则它们会引起很多的别名。例如，若 p 和 q 相等，则 * p 和 * q、p->next 和 q->next、p->data 和 q->data 等都分别互为别名（如果它们都是程序中合法的表达式）。由此，两个指针的值相等，即它们指向同一个单元时，称它们为**别名化指针**（aliased pointer）。指针别名分析就是通过分析指针运行时可能的值（也称为指针分析）来确定可能别名化指针或者指针可能指向的单元集合（也称为 point-to 分析）。指针别名分析是非常困难的，因为程序中存在许多潜在的别名化指针，并且其中每个指针在程序运行过程中都可以随着时间的推移而指向不计其数的动态对象。为了提高指针分析的精度，指针分析还必须穿越一个程序的所有过程。因为没有过程间的分析，所以只能假定被调用过程会改变所有可访问指针变量的内容，从而使过程内的指针分析也变得低效。

3. 过程间分析

对于采用引用调用的语言，过程间分析用来确定同一个变量是否作为两个或更多的不同变元被传递（习题 6.8 是一个例子）。这样的别名会导致在表面上不同的形参访问之间建立了相关。类似地，当全局变量作为实参时，也会建立形参访问和全局变量访问之间的相关。过程间分析对确定这样的别名是必须的。

10.2.3 寄存器使用和并行执行之间的折中

本章假定在翻译源程序得到的独立于机器的中间表示中，使用个数不限的伪寄存器来表示可以分配到寄存器中的变量。这些变量包括源程序中不能通过其他名字引用的标量，还有编译器为了保存表达式中间结果的临时变量。不像内存单元那样，寄存器是惟一命名的，因此很容易为寄存器访问产生精确的数据相关约束。

中间表示采用的个数不限的伪寄存器，最终都必须映射到目标机器上可用的少数物理寄存器上。把几个伪寄存器映射到同一个物理寄存器上会人为导致存储相关，限制了指令级并行。反过来，并行执行指令会要求更多的存储单元来保存同时计算的值。因此，极小化寄存器使用个数和极大化指令级并行这两个目标是相冲突的。下面的例 10.2 和例 10.3 说明寄存器使用和并行执行之间的典型折中。

【例 10.2】 下面的代码使用伪寄存器 t_1、t_2，把内存变量 a、c 的值分别复写到内存变量 b、d。

```
LD t1, a
ST b, t1
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第412页 文件编号: 422624 页面编号: 34124538447 打印人: 8685 2026-03-09 10:23:05

LD t_2, c

ST d, t_2

如果所有被访问的内存单元相互是有区别的，那么这两个复写可以并行执行。但是如果把 t_1 和 t_2 指派到同一个寄存器，那么寄存器的使用是极小化了，但是这两个复写就只能串行执行了。

【例 10.3】 传统的寄存器分配技术瞄准极小化寄存器的使用个数。考虑下面的表达式：

$$(a+b)+c+(d+e)$$

其语法树在图 10.2 中。完成这个计算可以只使用 3 个寄存器，其代码如下。

```
LD R1,a          LD R2,d
LD R2,b          LD R3,e
ADD R1,R1,R2     ADD R2,R2,R3
LD R2,c          ADD R1,R1,R2
ADD R1,R1,R2
```

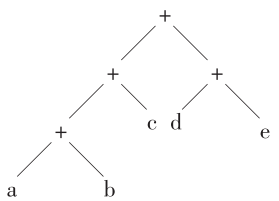


图 10.2 例 10.3 的表达式语法树

重复利用寄存器导致计算的串行，仅允许的并行操作是同时把 a 和 b 分别取到寄存器 R1 和 R2，还有同时取 d 和 e 的操作。因此并行完成整个计算需要 7 步。

如果对每个中间结果使用不同寄存器，则完成该表达式的计算只需要 4 步，它和图 10.2 中表达式的高度是一致的。具体的并行方式如图 10.3 所示。

R1 = a	R2 = b	R3 = c	R4 = d	R5 = e
R6 = R1 + R2	R7 = R4 + R5			
R8 = R6 + R3				
R9 = R8 + R7				

图 10.3 图 10.2 表达式的并行计算

10.2.4 寄存器分配和代码调度的次序安排

如果在代码调度之前进行寄存器分配，则结果代码中会有很多存储相关，它们限制了代码调度。如果代码调度在寄存器分配之前，那么代码调度会导致需要许多寄存器的代码，寄存器溢出会抵消指令级并行的优点。寄存器溢出是指把寄存器的内容存入内存单元，使得寄

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第413页 文件编号: 422624 页面编号: 34134538445 打印人: 8685 2026-03-09 10:23:05

存器可以用于其它目的。由此，对于寄存器分配和代码调度，编译器究竟应该先做哪个？还是编译器应该同时考虑这两个问题？

为了回答上面的问题，必须考虑被编译程序的特点。许多非数值应用没有很多可用的并行，考虑用较少的寄存器来保存表达式的中间结果即可。这时可以先分配寄存器，然后调度代码。但是这种方式不适用于有许多大表达式的数值应用。对它们可以从最内层循环开始，由里向外逐层地优化代码。在假定每个伪寄存器就是物理寄存器情况下，首先调度指令，然后进行寄存器分配，把处理寄存器溢出的代码附加在必要的地方，并再次进行代码调度。对较外层循环的代码重复这个过程。当同时考虑处于一个较外层循环的多个内循环时，同样的变量可能被指派不同的寄存器。通过调整寄存器的指派来避免在寄存器之间进行值的复写。10.5节将讨论在特定调度算法中，寄存器分配和代码调度之间的相互影响。

10.2.5 控制相关

基本块内的代码调度相对容易，因为当控制流到达该块入口时，该块所有指令肯定要被执行一次。所以，基本块中的指令可以任意重新排序，只要所有的数据相关得到满足即可。但是，基本块非常小，尤其是非数值程序中的基本块，平均一个基本块只有 5 条指令。此外，一个基本块中的操作通常高度相关，几乎不能并行。因此，调查跨基本块的并行就是至关重要的。

优化程序必须执行源程序中的所有操作。它可以比源程序执行更多的指令，只要额外的指令不改变程序的语义就可以。为什么执行额外的指令会加快程序的执行？如果知道一条指令很可能被执行且有空闲的资源可“免费”用于完成该指令的操作，那么可以投机地执行该指令。如果投机成功，则程序可以运行得快一些。

如果 i_2 的结果决定 i_1 是否被执行，则指令 i_1 被认为控制相关于指令 i_2。例如，在条件语句

```
if (c) s1;else s2;
```

中，s1 和 s2 都控制相关于 c。类似地，在循环语句

```
while (c) s;
```

中，循环体 s 控制相关于 c。

【例 10.4】 在代码片段

```
if (a > t)
    b = a * a;
d = a + c;
```

中，语句 b = a * a 和 d = a + c 与该片段的其它部分没有数据相关。语句 b = a * a 依赖于比较 a > t 的结果，但是语句 d = a + c 不依赖于该比较且可以在任何时候执行。假定乘操作 a * a 不会产生任何副作用，则它可以投机地执行，只要在发现 a > t 之后才把 a * a 的结果写入 b 就可以了。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第414页 文件编号: 422624 页面编号: 34144538443 打印人: 8685 2026-03-09 10:23:05

10.2.6 投机执行的支持

内存读取是一类能从投机执行大大获益，并且使用相当频繁的指令。它们有相对长的输入延迟，读取指令中用的地址通常是事先可用的，并且读取的结果可以存放于新的临时变量而不会破坏任何其它变量的值。但是，如果地址非法，则内存读取会引发异常，因此投机访问非法地址可能会引起正确程序意料不到的停止。此外，由于预测不正确发生的内存读取会引起额外的缓存未命中和缺页，这个代价很高。

【例 10.5】 在代码片段

```
if (p != null)
    q = *p;
```

中，投机地对 p 脱引用，将引起该正确代码片段因试图取 null 指向对象的错误而停止。

许多高性能处理器提供专门的特性来支持投机地访问内存，下面介绍其中的一些最重要特性。

（1）预取。预取指令被设计在数据使用前，将它从内存取到缓存。预取指令告诉处理器，程序在不久的将来很可能使用某个特定的内存单元。如果该单元无效或者访问它会引起缺页，则处理器简单地忽略该操作；否则，如果它不在缓存，那么处理器将数据从内存取到缓存。

（2）抑制位。抑制位（poison bit）的体系结构特征被设计来允许投机地从内存将数据读取到寄存器堆。该处理器上的每个寄存器都增加了抑制位。如果出现非法内存访问或者所访问的页不在内存，该处理器并不马上引发异常，而只是设置目标寄存器的抑制位。只有当已经设置了抑制位的寄存器的内容被使用时，才会引发异常。

（3）判定执行。由于分支的花费很大，尤其是在分支预测出现错误时，因此采用判定指令（decision instruction）来减少程序中的分支数。判定指令与普通指令类似，但其由一个额外的判定操作数来看守它的执行。只有在判定为真的情况下，判定指令才执行。

例如，条件传送指令 CMOVZ R2, R3, R1 只有在 R1 为 0 的情况下，才将寄存器 R3 的内容移到寄存器 R2 中。假定 a、b、c 和 d 分别被分配了寄存器 R1、R2、R4 和 R5，代码片段

```
if (a == 0)
    b = c + d;
```

可以用两条机器指令实现：

```
ADD R3, R4, R5
CMOVZ R2, R3, R1
```

这个转换将一个控制相关的指令序列变换为只有数据相关的指令序列。于是这些指令可以用来将相邻基本块组合成一个更大的基本块。更重要的是，使用这样的代码可以保证指令流水的平滑运行，因为处理器不会出现预测错误。

判定执行伴随着一些代价的发生。判定指令被预取和译码，即使它们有可能最后不被执行。静态调度必须预留所有需要用于它们执行的资源，并保证所有潜在的数据相关都满足。应谨慎使用判定执行，除非机器有足够多的资源。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第415页 文件编号: 422624 页面编号: 34154538440 打印人: 8685 2026-03-09 10:23:05

10.2.7 一个基本的机器模型

许多机器可以用下面的简单模型表示。机器 $M=(R,T)$ 由 R 和 T 两部分组成：

（1）操作类型的集合 T，如读取、存储和算术运算等等。

（2）代表硬件资源的向量集合 $R=[r_1,r_2,\cdots]$，其中 r_i 代表第 i 类资源中可用的部件数目。典型的资源类型包括内存访问部件、算术运算部件和浮点功能部件等。

每个操作有一组输入操作数、一组输出操作数和一个资源需求。和每个输入操作数相关的是一个输入延迟，用于指示什么时候输入值必须可用（相对于该操作的启动）。典型的输入操作数是零延迟的。也就是说，该值在该操作被发射的那个时钟周期就需要。类似地，和每个输出操作数相关的是一个输出延迟，它指示什么时候结果可用（相对于该操作的启动）。

对于每种机器操作类型 t，资源使用由一张二维资源预留表 RT_t 来建模。该表的宽度是该机器上资源种类数，该表的长度是资源被该类型操作使用的持续时间。条目 $RT_t[i,j]$ 是 t 类型的一个操作在它被发射 i 个时钟周期后，使用第 j 种资源的部件数。为了表示上的简单，如果 i 引用该表中一个不存在的条目（例如 i 大于该操作执行所需的时钟周期数），则假定 $RT_t[i,j]=0$。当然，对任何 t、i 和 j，$RT_t[i,j]$ 必须小于或等于 $R[j]$，其中 $R[j]$ 是该机器拥有的第 j 类型资源数目。

典型的机器操作在它被发射时仅占用一个资源部件。有些操作可能使用的功能部件多于一个。例如，一个乘加操作可以在第一个时钟周期使用一个乘法器，在第二个时钟周期使用一个加法器。全流水的操作是那些能够在每个时钟周期都发射的操作，即使它们的结果在几个时钟周期后才可用。不需要为一条流水线各级的资源显式地建模，只需要为代表第一级的资源部件建模，占据一条流水线第一级的任何操作被保证在随后各时钟周期正确处理随后的各级。

10.3 基本块调度

现在开始讨论代码调度算法。先从最简单的调度基本块中的操作开始。该问题的最优解是一个 NP 完全问题。但是在实际中，一个典型的基本块只有几个高度受约束的操作，只需要简单的调度技术就足够了。本节介绍一个称为**表调度**的简单高效算法。

10.3.1 数据依赖图

机器指令的每个基本块由数据依赖图 $G=(N,E)$ 来表示，其中结点集合 N 表示该块的机器指令中的操作集合，有向边集合 E 表示这些操作之间的数据相关约束。G 的结点集合 N 和边集合 E 按如下两步构造。

（1）结点集合 N 中的每个操作 n 有一张资源预留表 RT_n，它的值直接就是 n 的操作类型的资源预留表。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第416页 文件编号: 422624 页面编号: 34164538448 打印人: 8685 2026-03-09 10:23:05

（2）边集合 E 中的每条边 e 都标示有延迟 d_e，它表示 e 的目的结点必须在它的源结点发射 d_e 个时钟周期后才可以发射。假定操作 n_1 由操作 n_2 跟随，并且它们访问同一个单元，前者的输出延迟和后者的输入延迟分别是 l_1 和 l_2。也就是说，该单元的值在第一条指令发射 l_1 个时钟周期后产生，并且在第二条指令发射 l_2 个时钟周期后被需要（通常 $l_1 = 1$ 且 $l_2 = 0$）。那么边 $n_1 \to n_2$ 标有延迟 $l_1 - l_2$。

【例 10.6】 考虑一种简单的机器，它的每个时钟周期可以执行两个操作：一个操作必须是分支操作或算术运算部件 ALU 的操作；另一个操作必须是读取（LD）或存储（ST）操作。读取操作是全流水的且需要两个时钟周期，读取操作可以直接由写到该读取单元的存储操作跟随。其他所有操作都在一个时钟周期内完成。

图 10.4 是一个基本块的依赖图和它的资源需求。可以把 R1 想象成一个栈指针，通过像 0 或 12 这样的偏移来访问栈上的数据。由于第 1 条指令读取数据到 R2 需要两个时钟周期，因此从它到第 2 条指令和第 5 条指令的边上的标记都是 2。类似地，从第 3 条指令到第 4 条指令的边上的标记也是 2。

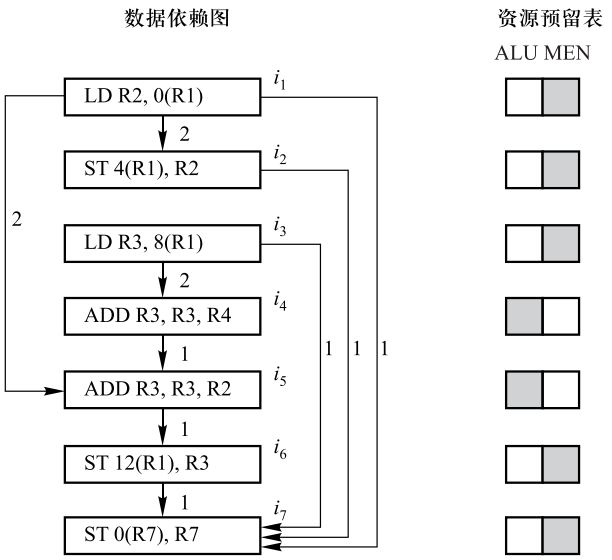


图 10.4 例 10.6 的数据依赖图

由于不知道 R1 和 R7 的值是怎样的关系，则不得不认为 8(R1) 和 0(R7) 有可能是同一地址。这样，第 7 条指令存储的地址可能与第 3 条指令读取的地址相同。根据所用的机器模型，存储一个单元的指令可以在读取同一个单元的指令发射一个时钟周期后就发射，虽然这时值还需一个时钟周期才能取到寄存器。所以，从第 3 条指令到第 7 条指令的边的标记为 1，第 1 条指令和第 2 条指令到第 7 条指令的边的标记也为 1 是同样的道理。

通常用灰白两色的方格来形象化资源预留表。每列对应到机器的一种资源，每行对应到

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第417页 文件编号: 422624 页面编号: 34174538446 打印人: 8685 2026-03-09 10:23:05

相应操作执行过程中的一个时钟周期。假定各操作对任何一种资源最多只需要一个部件，则可以用灰色方块表示 1，用白色方块表示 0。此外，如果操作是全流水的，那么只需要显示在第 1 行使用的资源，这样资源预留表就可以简化成单行的表。

例如，例 10.6 使用的就是这种表示。在图 10.4 中，资源预留表用方块行表示，两个加运算需要 ALU 资源，读取和存储指令需要内存资源。

10.3.2 基本块的表调度

调度基本块的最简单方式，涉及用区分优先级的拓扑次序（prioritized topological order）来访问数据依赖图中的各个结点。因为数据依赖图无环，所以至少存在这些结点的一种拓扑次序。然而，在这些可能的拓扑次序中，某些次序可能比其他次序优越。10.3.3 节将讨论选择拓扑次序的某些策略，但是现在假定存在算法选择一个较优的次序。

下面描述的表调度算法按所选定的区分优先级的拓扑次序访问结点。结点调度的次序和它们被访问的次序可能相同，也可能不相同。但是，指令应尽可能早地放进调度表，因此一种趋势是指令以近似被访问的次序被调度。

更具体一点，该算法根据每个结点与先前已经被调度的各结点之间的数据相关约束，来计算该结点可以执行的最早时间槽。然后，该结点所需资源根据一张资源预留表来进行检查。该资源预留表收集了所有到目前为止被占用资源。该结点的调度按有足够资源的最早时间槽来安排。

算法 10.1 基本块的表调度。

输入：机器资源向量 $\boldsymbol{R}=[r_1,r_2,\cdots]$，其中 r_i 是机器第 i 种资源的可用部件数目；另外，还有数据依赖图 $G=(N,E)$。N 中的每个操作 n 都带有它的资源预留表 RT_n（根据上面的讨论，简化为只有一行的表），E 中每条边 $e=n_1\rightarrow n_2$ 标有 d_e，表示 n_2 必须在 n_1 开始 d_e 个时钟周期后才能开始。

输出：一张调度表 S 把 N 中的每个操作映射到一个时间槽，该操作在该时间槽可以启动，满足所有数据和资源的约束。

方法：执行图 10.5 所示的程序。有关其中“区分优先级的拓扑次序”在 10.3.3 节中讨论。

```
for(N 中按区分优先级的拓扑次序的每个 n){
    s=max_{e=p→n in E}(S(p)+d_e);      /* 查找该指令在所有前驱指令都启动后,它能够启动的最早时间 */
    while (存在一个 i 使得 RT[s,i]+RT_n[i]>R[i])
        s=s+1;                          /* 延迟指令直到所需资源可用 */
    S(n)=s;
    for (所有的 i)
        RT[s,i]=RT[s,i]+RT_n[i];
    }
```

图 10.5 一个表调度算法

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第418页 文件编号: 422624 页面编号: 34184538444 打印人: 8685 2026-03-09 10:23:05

注意，其中 $RT[s,i]$ 表示第 i 种资源在时间槽 s 的预留数，$RT_n[i]$ 表示依赖图上第 n 条指令对第 i 种资源的需求数，$R[i]$ 表示第 i 种资源的可用部件数。p 是已经进入调度表 S 的第 n 条指令的前驱指令。

10.3.3 区分优先级的拓扑次序

表调度算法不需要回溯，它调度每个结点都正好一次。在调度中，它使用启发式优先级函数，从可被调度的结点中选择一个结点。该启发式调度的一些原则如下。

（1）在没有资源约束时，最短调度由**关键路径**给出。关键路径是数据依赖图上的最长路径。作为优先级函数的一个有用尺度是结点的高度，也就是该图中源自该结点的最长路径的长度。

（2）如果所有操作都是相互独立的，那么调度的长度只受资源约束。关键资源是所有可用资源中使用频率最高的资源，那些使用较多关键资源的操作可以给予较高优先级。

（3）可以使用原先的代码排序来打破操作之间调度的平局，在原先的代码中较早出现的操作将优先被调度。

【例 10.7】 对于图 10.4 所示的数据依赖图，关键路径包括最后 5 个结点，该路径边上延迟的总和是 5，最后 1 个操作需要的时间是一个时钟周期，因此该路径是 6 个时钟周期。

使用高度作为优先级函数，算法 10.1 找出一种优化调度如图 10.6 所示。注意，第 3 条指令由于具有最高高度而首先被调度，下一步原本可以调度第 4 条指令，但是由于第 3 条指令读取操作的延迟是 2，因此必须等到第 3 个时钟周期才能调度第 4 条指令。该基本块经过表调度后，原来 7 条指令能够在 7 个时钟周期内完成，包括执行最后一条指令需要的一个时钟周期。

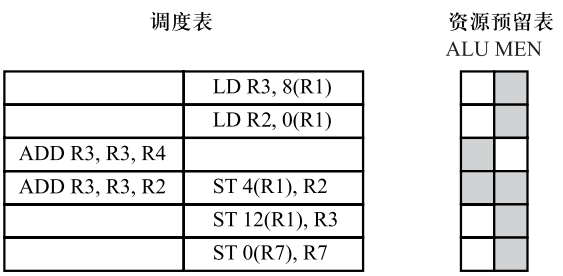


图 10.6 表调度用到图 10.4 中例子的结果

10.4 全局代码调度

对于有适度指令级并行的机器，仅考虑紧凑单个基本块的调度会引起许多资源空闲。为了更好地利用机器资源，需要考虑把指令从一个基本块移到另一个基本块的代码生成策略。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第419页 文件编号: 422624 页面编号: 34194538442 打印人: 8685 2026-03-09 10:23:05

每次考虑多于一个基本块的调度策略被称为**全局调度**算法。为了正确地进行全局调度，不仅需要考虑数据相关，还需要考虑控制相关。因此，必须保证：

（1）原来程序中所有的指令在优化程序中都被执行；

（2）当该优化程序可以投机地执行额外指令时，这些指令肯定不能有任何多余的作用。

10.4.1 简单的代码移动

以例 10.8 为例来研究将操作在基本块之间移动所涉及的问题。

【例 10.8】 假定机器可以在一个时钟周期执行任意的两个操作。除读取操作外，每个操作的执行有一个时钟周期的延迟，读取操作有两个时钟周期的延迟。为简单起见，假定该例中所有内存访问操作都是有效的，并且能命中缓存。图 10.7(a) 给出只有 3 个基本块的一个流图；图 10.7(b) 是翻译成机器操作后的代码。其中，BEQZ R6, L 是条件转移指令，表示如果 R6 的值为 0，则跳转到 L 去执行；否则执行后继指令。由于数据相关，每个基本块中的指令必须串行执行，并且还必须插入代表空操作的 NOP 指令。

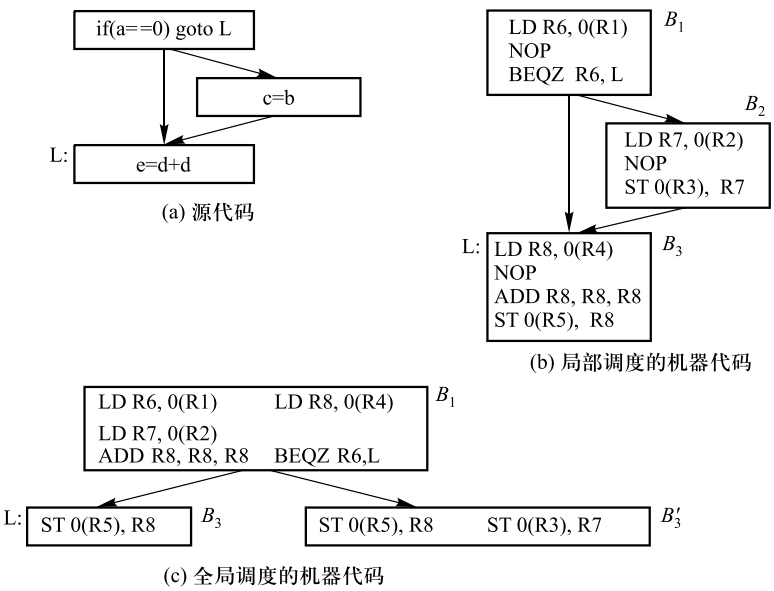


图 10.7 例 10.8 中全局调度前后的流图

假定变量 a、b、c、d 和 e 的地址是有区别的，它们的地址分别保存在寄存器 R1 ～ R5。这样，不同基本块之间的操作没有数据相关。可以看出，不管分支测试最终选择什么路径，块 B_3 的所有操作肯定都要执行，因而它们可以和块 B_1 的操作并行执行。但是不能把块 B_1 的操作移动到块 B_3，因为它们是用来确定分支测试结果的。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第420页 文件编号: 422624 页面编号: 34204538440 打印人: 8685 2026-03-09 10:23:05

块 B_2 的操作是控制依赖于块 B_1 测试结果的。在块 B_1 中可以投机地完成块 B_2 的读取操作，若测试结果导致块 B_2 被执行，则此举可以节约两个时钟周期的执行时间。

存储操作不能投机地完成，因为它覆盖了一个存储单元的旧值，但是延迟一个存储操作是可以的。由于块 B_2 并不一定执行，因此不能把块 B_2 的存储操作直接放到块 B_3 中，但是可以把该存储操作放到块 B_3 的一份拷贝中。图 10.7(c) 展示了一个优化的调度，其中用逗号隔开的并排两条指令同时执行。该优化代码执行只需要 4 个时钟周期，它和单独执行原先块 B_3 的时间是一样的。

例 10.8 表明，把操作沿执行路径上下移动是可能的。该例中的每对基本块有不同的“支配关系”，因而指令在每对基本块之间何时和怎样移动的考虑是不同的。9.6 节已经介绍过，如果每条从控制流图入口到达块 B' 的路径都要通过块 B，那么称 B 支配 B'。类似地，如果每条从 B' 到达该图出口的路径都要通过 B，那么称 B 后支配（postdominate）B'。如果 B 支配 B' 并且 B' 后支配 B，那么称 B 和 B' 是**控制等价**的，即它们中一块被执行当且仅当另一块也被执行。例如，假定图 10.7 中 B_1 和 B_3 分别是入口和出口，则

（1）B_1 和 B_3 控制等价，B_1 支配 B_3，B_3 后支配 B_1；

（2）B_1 支配 B_2，但是 B_2 并非后支配 B_1；

（3）B_2 不支配 B_3，但是 B_3 后支配 B_2。

显然，一条路径上一对基本块相互之间既无支配关系也无后支配关系也是可能的。

10.4.2 向上的代码移动

现在仔细考察沿一条路径向上移动一个操作意味着什么。假如想把一个操作沿控制流从块 src 向上移动到块 dst。假定这个移动没有违反任何数据相关，并且它使通过 dst 到 src 的路径运行得较快。

如果 dst 和 src 等价，那么当被移动操作应该被执行时，它正好仅被执行一次。

如果 src 没有后支配 dst，那么存在通过 dst 但没有到达 src 的路径。在这种情况下，代码移动会导致在这样的路径上额外执行了被移动操作，因此这时代码移动是非法的，除非该被移动操作没有多余的副作用。如果该被移动操作可利用空闲资源“免费”执行，那么这个移动不增加代价，但是它仅在控制流到达 src 时获益。

如果 dst 不支配 src，那么存在没有首先经过 dst 而到达 src 的路径。在这种情况下，需要沿这样的路径插入被移动操作的副本。在 9.5 节讨论部分冗余删除时，已经知道怎样插入代码：把该被移动操作的副本放置到形成一个割集（分离入口和 src）的各基本块中。在每步插入被移动操作的位置，下面的约束必须得到满足：

（1）该被移动操作的操作数必须和在原来位置时的值一样；

（2）其结果不会覆盖还将使用的值；

（3）在到达 src 以前，该结果不会被随后的操作覆盖。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第421页 文件编号: 422624 页面编号: 34214538448 打印人: 8685 2026-03-09 10:23:05

这些拷贝使得 src 中原来的那个操作完全冗余，因而可以把它删除。

被移动操作的额外拷贝被称为**补偿代码**。如 9.5 节讨论的那样，新基本块可以沿关键边插入以构成保存这些拷贝的地方。这些补偿代码可能会使某些路径的执行变得慢一些。因此，只有在被优化路径比未优化路径执行得更频繁时，这种代码移动才能改进程序的执行。

10.4.3 向上的代码移动

现在考虑把一个操作沿控制流从块 src 向下移动到块 dst。它可用类似于上一小节介绍的方式来推理代码移动。

如果 src 和 dst 等价，那么情况和上一小节介绍的一样。

如果 src 不支配 dst，那么存在没有首先经过 src 而到达 dst 的路径。同样，在这样的路径上额外操作将被执行。但是，向下移动的代码经常是存储操作，它们有覆盖旧值的副作用。这个问题的解决可以通过复制从 src 到 dst 路径上的各基本块，并且把被移动操作仅放置在 dst 的新副本中。若有判定指令可用，另一种方式则是采用判定指令，通过用看守块 src 的判定条件来看守被移动操作。注意，判定指令只能被调度到由该判定条件的计算支配的基本块中，因为该判定条件在其它地方不可用。

如果 dst 没有后支配 src，那么如上面讨论的那样，必须插入补偿代码以保证被移动操作在所有不访问 dst 的路径上都执行。除了被移动代码的拷贝被放置在 src 下的一个割集（分离 src 和出口）中，这个变换仍然类似于局部冗余删除。

由上面的讨论小结一下上、下两个方向的代码移动。全局代码移动涉及的因素包括利益、代价和实现的复杂性等。表 10.1 给出了各种代码移动的小结，表中的 4 行分别对应下面的 4 种情况。

表 10.1 代码移动的小结

	向上：src 后支配 det 向下：src 支配 det	向上：det 支配 src 向下：det 后支配 src	投机 代码复制	补偿代码
1	是	是	否	否
2	否	是	是	否
3	是	否	否	是
4	否	否	是	是

（1）在控制等价的块之间移动指令是最简单的，并且是最经济的。这时无额外的操作被执行，也不需要补偿代码。

（2）在向上（向下）的代码移动中，如果块 src 没有后支配（支配）块 dst，就可能会执行额外的操作。如果该额外操作可以免费执行并且通过块 src 的路径被执行，那么代码移动是有益处的。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第422页 文件编号: 422624 页面编号: 34224538446 打印人: 8685 2026-03-09 10:23:05

（3）在向上（向下）的代码移动中，如果块 dst 没有支配（后支配）块 src，那么需要补偿代码。因为增加了补偿代码的路径的执行可能会变慢，所以重要的是被优化路径的执行频率应该较高。

（4）这种情况集中了第（2）种和第（3）种情况的缺点，也就是既可执行额外操作，又需要补偿代码。

10.4.4 更新数据相关

下面的例 10.9 表明，代码移动会改变操作之间的数据相关关系，因此在每次代码移动后数据相关必须被更新。

【例 10.9】 在图 10.8 所示的流图中，两个对 x 的赋值之一可以移动到最上面的那个基本块，因为该变换能维持原来程序中的所有相关性。但是，如果其中一个对 x 的赋值被上移，则另一个就不能移动了。具体来说，移动前变量 x 在最上面块的出口不是活跃的，移动后成为活跃的。如果一个变量在某个程序点活跃，则不能把对该变量的投机定值移到该程序点的上面。

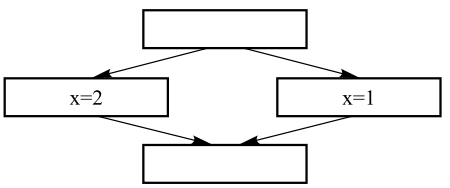


图 10.8 解释由于代码移动引起数据相关性改变的示例

10.4.5 全局调度的其他问题

由前文可知，代码移动使某些路径受益，但也可能损害了其它路径的性能。由于程序 90%的执行时间消耗在低于 10%的代码上，因此程序调度应该使经常执行的路径运行得快一些，而不经常执行的路径可能会因这种调度变得慢一些。

编译器可用来估计执行频率的技术有若干种。下面的一些假设是合理的：最内循环中的指令比较外循环中的指令执行得更频繁，分支指令往回跳转比不跳转要更经常些，看守程序出口或异常处理例程的分支语句很少被执行。最好的频率估计来自动态剖析。在这种技术中，程序被静态插桩（instrument）以用来运行时记录条件分支每次的走向。然后程序在有代表性的输入上执行，以推断它在一般情况下的行为。从这种技术得到的结果相当精确，这些信息反馈给编译器，用在它的代码优化中。

即使是最简单的全局调度算法也相当复杂，并且涉及本书没有介绍的一些概念，因此本书不介绍全局调度算法。下面再说明一个和全局调度有关的循环展开问题。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第423页 文件编号: 422624 页面编号: 34234538444 打印人: 8685 2026-03-09 10:23:05

在一些全局调度算法中，循环迭代的边界是代码移动的一种屏障。一次迭代中的操作不可以和另一次迭代中的操作有任何重叠。打破这种屏障的一种简单而非常有效的技术是在代码调度前把循环展开几次迭代，使循环体中有更多的指令，从而全局调度算法有机会发现更多的并行性。图 10.9 给出了 for 循环展开前后的示例。

```
for(i=0;i<N;i ++) {
    S(i);
}
```

(a) 展开前的 for 循环

```
for(i=0;i+4<N;i += 4) {
    S(i);
    S(i+1);
    S(i+2);
    S(i+3);
}
for( ;i<N;i++) {
    S(i);
}
```

(b) 展开后的 for 循环

图 10.9 for 循环展开前后的示例

10.4.6 静态调度器和动态调度器的相互影响

动态调度器的优点是可以根据运行时的情况建立新的调度表，而不需要事先编码所有可能的调度表。如果目标机器有动态调度器，那么静态调度器的基本功能是保证尽早地取高延迟的指令，使动态调度器能够尽早地发射它们。

缓存未命中是一类不可预测的事件，它们会给程序性能带来很大影响。如果数据预取指令可用，静态调度器通过尽早地安排预取指令，使数据到要用时已经在缓存中，给动态调度器以很大帮助。如果没有预取指令可用，编译器可以预测哪些操作很可能不命中并尽早地安排它们，这对动态调度器来说也是有用的。

如果目标机器上无动态调度可用，则静态调度器必须是稳妥的，它通过极小化延迟来分离每对数据相关的操作。但是，如果动态调度可用，则编译器只需给数据相关的操作安排正确的次序以保证程序的正确性。为了获得最好的性能，编译器应该分配长延迟到很可能出现的数据相关，而分配短延迟到那些不大可能出现的数据相关。

分支预测错误是性能损失的一个重要原因。由于长预测错误的处罚（long misprediction penalty），在较少执行的路径上的指令仍然会对总执行时间有重要影响。因此应该给这样的指令较高优先级，以减少预测错误的代价。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第424页 文件编号: 422624 页面编号: 34244538442 打印人: 8685 2026-03-09 10:23:05

10.5 软 件 流 水

如本章开头所述，数值应用中有很多并行性，尤其是经常使用各次迭代之间相互独立的循环。这些循环称为 do-all 循环，从并行化的观点来看，它们特别有吸引力，因为这种循环的各次迭代可以并行执行，以获得按该循环次数的线性加速。迭代次数很多的 do-all 循环有足够的并行性使一个处理器上的资源饱和，调度器可以充分利用这种并行性。本节描述一种称为**软件流水**的调度算法，它每次调度一个完整的循环，以充分利用穿越迭代的并行性。

10.5.1 引言

本节使用例 10.10 的 do-all 循环来解释软件流水。首先说明穿越迭代的调度非常重要，因为在单次迭代的操作中几乎没有什么并行性。然后通过重叠被展开循环的计算来说明循环展开可以改进性能。但是，被展开循环的边界仍然是代码移动的屏障，因而循环展开仍然有许多性能问题有待讨论。最后软件流水技术不断地重叠一些相继迭代，直到所有迭代都填入流水线为止。这种技术使得软件流水产生高效和紧凑的代码。

【例 10.10】 下面的代码是一个典型的 do-all 循环：

```
for (i = 0; i < n; i ++)
    D[i] = A[i] * B[i] + c;
```

在该循环中，各次迭代写不同的内存单元，并且它们与任何读单元都有区别。因此这些迭代之间没有任何内存相关，所有的迭代可以并行处理。

本节使用下面的模型作为目标机器。在该模型中：

（1）该机器在一个时钟周期内可以同时发射一个读取、一个存储、一个算术运算和一个分支操作。

（2）该机器有形式为

```
BL R, L
```

的循环回跳操作，它使得寄存器 R 的值自动减 1，除非 R 的结果为 0，否则跳转到单元 L。

（3）内存操作有一种自动增量的寻址模式，由寄存器后的 ++ 表示。该寄存器的值自动地增加，指向当前访问地址的后继地址。

（4）算术操作是全流水的，它们可以在任意周期启动，但是它们的结果在两个时钟周期后才能用。所有其它指令只有一个时钟周期的延迟。

如果循环迭代每次一个地被调度，那么能得到该机器模型上的最好调度如图 10.10 所示；一些关于数据布局的假定也在该图中给出：寄存器 R1、R2 和 R3 分别保持数组 A、B 和 D 的开始地址，寄存器 R4 保存常量 c，并且寄存器 R10 保存值 n，它是在迭代外计算的。该计算大部分是串行的，它需要 7 个时钟周期；只有循环回跳指令和迭代中最后一条指令重叠。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第425页 文件编号: 422624 页面编号: 34254538449 打印人: 8685 2026-03-09 10:23:05


```
    // R1,R2,R3=&A,&B,&D
    // R4=c
    // R10=n
L:  LD  R5,0(R1++)
    LD  R6,0(R2++)
    MUL R7,R5,R6
    NOP
    ADD R8,R7,R4
    NOP
    ST  0(R3++),R8,     BL R10,L
```

图 10.10 例 10.10 局部调度后的代码

一般来说，展开一个循环的几次迭代可以改进硬件利用。但是，这样做也增加了代码的规模，即它对全面性能也产生负面影响。因此必须取一个折中，展开一个循环若干次以获得最好的性能改进，而又不让代码膨胀得过大。下面是一个说明这种折衷的例子。

【例 10.11】 在例 10.10 中，虽然很难发现在一次迭代中的并行性，但是有很多穿越迭代的并行性。循环展开把循环中几次迭代的代码放在一个大基本块中，然后一个简单的表调度算法就可让这些操作并行执行。如果把该例的循环展开 4 次迭代，然后把算法 10.1 用到展开后的代码，得到的调度表如图 10.11（为简单起见，忽略了寄存器分配的细节）所示。展开后每次迭代的执行用 13 个时钟周期，即原来的每次迭代仅需要 3.25 个时钟周期。

```
L:LD
  LD
       LD
MUL    LD
       MUL     LD
ADD            LD
       ADD             LD
ST             MUL     LD
       ST              MUL
                ADD
                        ADD
                ST
                        ST      BL(L)
```

图 10.11 例 10.10 的展开代码

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第426页 文件编号: 422624 页面编号: 34264538447 打印人: 8685 2026-03-09 10:23:05

对例 10.10 来说，展开 k 次迭代的循环至少需要 $2k+5$ 个时钟周期，那么获得原先每次迭代仅需 $2+5/k$ 个时钟周期的吞吐能力。展开的迭代次数越多，该循环运行得越快。当 k 趋向于 ∞ 时，一个完全展开循环的执行可以使原先每次迭代平均只需两个时钟周期。然而，展开的迭代次数越多，代码的规模就越大，显然，机器可能供不起展开所有的迭代。

10.5.2 循环的软件流水

软件流水提供一种便利的方式来获得优化的资源利用并同时紧凑代码。仍用例子说明这一点。

【例 10.12】 在图 10.12 中，例 10.10 的代码展开 5 次（同样，在此没有考虑寄存器分配）。第 i 行是在第 i 个时钟周期发射的所有操作；第 j 列是第 j 次迭代的所有操作。注意，每次迭代相对它们的开始点有同样的调度，并且每次迭代的启动比前一次迭代滞后两个时钟周期。很容易看出，该调度满足所有的资源和数据相关约束。

周期	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
1	LD				
2	LD				
3	MUL	LD			
4		LD			
5		MUL	LD		
6	ADD		LD		
7			MUL	LD	
8	ST	ADD		LD	
9				MUL	LD
10		ST	ADD		LD
11					MUL
12			ST	ADD	
13					
14				ST	ADD
15					
16					ST

图 10.12 例 10.10 代码经 5 次展开的迭代

还可以看出，在第 7 个和第 8 个时钟周期执行的操作和在第 9 个和第 10 个时钟周期执行的操作是一样的。第 7 个和第 8 个时钟周期执行的操作源于原来程序的前 4 次迭代，而第 9 个和第 10 个时钟周期执行的操作源于第 2 ～ 5 次迭代。事实上，可以持续执行同样的多操作指令对，以有效地撤出老的迭代并加入新的迭代，直到所有的迭代都填入为止。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第427页 文件编号: 422624 页面编号: 34274538445 打印人: 8685 2026-03-09 10:23:05

如果该循环至少有 4 次迭代，那么这样的动态行为可以简洁地编码成图 10.13 所示的代码。该图中每行对应到一条机器指令。第 7、8 两行构成一个 2 时钟的循环，重复执行 $n-3$ 次，其中 n 是原来循环的迭代次数。

```
1          LD
2          LD
3          MUL    LD
4                 LD
5                 MUL    LD
6          ADD           LD
7    L:                  MUL    LD
8          ST     ADD           LD     BL(L)
9                                MUL
10                ST     ADD
11
12                       ST     ADD
13
14                                ST
```

图 10.13　例 10.10 经软件流水化的代码

上面描述的技术称为软件流水，因为它是类似于调度硬件流水线技术的软件。在该例中，由每次迭代执行的调度可被看成 8 级流水线，每经过两个时钟周期，新一次迭代可以在该流水线上开始。最初只有第 1 次迭代在该流水线上，随着该迭代前进到第 3 级，第 2 次迭代开始执行它的第 1 级。

在第 7 个时钟周期，该流水线由前 4 次迭代填满。进入稳定状态后，相继的 4 次迭代同时执行；当一个老的迭代在该流水线上撤出，会填入一个新的迭代。当所有迭代都填完，该流水线排空，那么所有迭代在该流水线上都完成运行。在该例中，填充该流水线的第 1 ～ 6 行的指令序列称为序曲，第 7、8 行称为稳定状态，用来排空该流水线的第 9 ～ 14 行指令序列称为尾声。

在该例中，循环的运行速度不可能快于每两个时钟周期一次迭代，因为机器在每个时钟周期只能发射一个读操作，而每次迭代有两个读操作。该例被软件流水化的循环需要执行 $2n+6$ 个时钟周期，其中 n 是原来循环的迭代次数。当 n 趋于 ∞ 时，该循环的吞吐能力逼近每两个时钟周期完成一次迭代的速度。因此，和循环展开不一样的是，软件调度能以非常紧凑的代码序列来编码最优调度。

注意，为单次迭代采用的调度可能不是最短的。从图 10.10 的局部最优调度可以看出，在 ADD 操作前引入一个延迟，该延迟的安排是战略性的，以保证该调度能在每两个时钟周期内启动而没有资源冲突。如果在这里采用局部最优调度，启动间隔就不得不加长到 4 个时钟周

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第428页 文件编号: 422624 页面编号: 34284538443 打印人: 8685 2026-03-09 10:23:05

期，以避免资源冲突，那么吞吐能力就降到一半。该例说明流水线调度的一个重要原则，即必须仔细安排调度以最优化吞吐能力。局部紧凑的调度虽然极小化了单次迭代的时间，但把它流水化时可能得不到最理想的吞吐能力。

10.5.3 寄存器分配和代码生成

下面对例 10.12 中软件流水化的循环讨论寄存器分配。

【例 10.13】 在例 10.12 中，第 1 次迭代乘运算的结果在第 3 个时钟周期产生，在第 6 个时钟周期使用。在此期间，第二次迭代乘运算的结果在第 5 个时钟周期产生；该结果在第 8 个时钟周期使用。显然，这两次迭代乘运算的结果必须保存在不同的寄存器中，以防止它们相互干扰。由于相互干扰仅出现在相邻的一对迭代之间，因此这种干扰可以通过使用两个寄存器分别保存相邻迭代的乘运算结果来解决。这样，奇数次迭代和偶数次迭代的目标代码不完全相同，所以进入稳定状态后的循环由两个时钟周期加倍成 4 个时钟周期。

对迭代次数大于或等于 5 的任何奇数次迭代，很容易写出相应的代码。为了也能够处理少于 5 次的迭代和偶数次迭代的循环，应该生成对应于图 10.14 源代码的目标代码，其中第一个循环处理迭代次数大于或等于 5 的任何奇数次迭代，该循环被流水化，它对应的目标代码如图 10.15 所示。第 2 个循环不需要优化，因为它最多 4 次迭代。

10.5.4 do-across 循环

软件流水也可以用到迭代之间存在数据相关的循环，这样的循环称为 do-across 循环。

【例 10.14】 代码

```
for (i = 0; i < n; i ++) {
    sum = sum + A[i];
    B[i] = A[i] * b;
}
```

在连续的迭代之间存在数据相关，因为 A[i] 加到 sum 先前的值，以建立 sum 新值。如果机器能提供足够的并行，那么执行求和可以在 $O(\log n)$ 的时间复杂度内完成，但是为便于讨论起见，假定所有的串行相关必须遵守，并且加法必须按原来的串行顺序完成。因为先前假设了完成 ADD 需要两个时钟周期，因此该循环的执行不可能快于每两个时钟周期一次迭代。该机器即使有更多的加法器或乘法器，也不可能使该循环更快一点。这个例子的 do-across 循环的吞吐能力受到穿越迭代的数据相关链的限制。

对每次迭代的最好紧凑调度如图 10.16(a) 所示，软件流水化的代码如图 10.16(b) 所示。软件流水化的代码每两个时钟周期开始一次迭代，因而它是以最优的速度运算。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第429页 文件编号: 422624 页面编号: 34294538441 打印人: 8685 2026-03-09 10:23:05


```
if(N >= 5)
    N2=1+2 *((N-1) / 2);
else
    N2=0;
for(i=0;i<N2;i ++)
    D[i]=A[i] * B[i]+c;
for(i=N2;i<N;i ++)
    D[i]=A[i] * B[i]+c;
```

图 10.14 例 10.10 循环的源级展开

```
(1)        LD R5,0(R1++)
(2)        LD R6,0(R2++)
(3)        LD R5,0(R1++)   MUL R7,R5,R6
(4)        LD R6,0(R2++)
(5)        LD R5,0(R1++)   MUL R9,R5,R6
(6)        LD R6,0(R2++)   ADD R8,R7,R4
(7)   L:   LD R5,0(R1++)   MUL R7,R5,R6
(8)        LD R6,0(R2++)   ADD R8,R9,R4   ST 0(R3++),R8
(9)        LD R5,0(R1++)   MUL R9,R5,R6
(10)       LD R6,0(R2++)   ADD R8,R7,R4   ST 0(R3++),R8   BL R10,L
(11)                       MUL R7,R5,R6
(12)                       ADD R8,R9,R4   ST 0(R3++),R8
(13)
(14)                       ADD R8,R7,R4   ST 0(R3++),R8
(15)
(16)                                      ST 0(R3++),R8
```

图 10.15 例 10.13 经软件流水和寄存器分配后的代码

```
// R1=&A;R2=&B
// R3=sum
// R4=b
// R10=n
L:LD R5,0(R1++)
  MUL R6,R5,R4
  ADD R3,R3,R4
ST R6,0(R2++)   BL R10,L
```

(a) 最好的局部紧凑代码

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第430页 文件编号: 422624 页面编号: 34304538449 打印人: 8685 2026-03-09 10:23:05


```
// R1=&A;R2=&B
// R3=sum
// R4=b
// R10=n
LD R5,0(R1++)
MUL R6,R5,R4
L:ADD R3,R3,R4      LD R5,0(R1++)
ST R6,0(R2++)       MUL R6,R5,R4      BL R10,L
                    ADD R3,R3,R4
                    ST R6,0(R2++)
```

(b) 软件流水化的版本

图 10.16 do-across 循环的软件流水

10.5.5 软件流水的目标和约束

软件流水的基本目标是极大化耗时较长的循环的吞吐能力，次要目标是保持所产生代码的规模较小。换句话说，软件流水化的循环应该有较小的流水线稳定状态。通过要求每次迭代的相对调度都相同且这些迭代以同样的时间间隔逐步启动，那么可以获得较小的稳定状态。因为一个循环的吞吐能力直接就是启动间隔的倒数，因此软件流水的目标是极小化这个间隔。

一个软件流水调度可以由下面两点来规范。

（1）启动间隔 T。

（2）相对调度表 S。S 描述一个迭代的每个操作相对于本迭代的执行时刻。

因此，第 i 次（从 0 开始计数）迭代的一个操作 n，在 $i \times T + S(n)$ 个时钟周期执行。像所有其它调度问题一样，软件流水有两类约束：资源约束和数据相关约束，下面分别讨论。

（1）资源约束。令机器资源由 $R = [r_1, r_2, \cdots]$ 表示，其中 r_i 是第 i 类资源可用的部件数目。如果某循环的一次迭代需要第 i 类资源的 n_i 个部件，那么流水化循环的平均启动间隔至少是 $\max_i(n_i/r_i)$ 个时钟周期。软件流水要求任何一对迭代之间的启动间隔是常数，因此启动间隔至少是 $\max_i\lceil n_i/r_i \rceil$ 个时钟周期。如果 $\max_i(n_i/r_i)$ 小于 1，将源代码展开少数几次是有用的。

【例 10.15】 现在回到图 10.13 软件流水化循环。前面已经说过，该目标机器在每个时钟周期内可以同时发射一个读取、一个存储、一个算术运算和一个循环回跳操作。因为该循环有两个读取、两个算术运算和一个存储操作，因此根据资源的约束，最小启动间隔是两个时钟周期。

从图 10.13 可以看出，4 次相继迭代越过迭代边界的资源需求。随着各迭代逐步启动，越来越多的资源被占用，资源占用的高峰在稳定状态。令 RT 代表单次迭代的资源预约表，并

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第431页 文件编号: 422624 页面编号: 34314538447 打印人: 8685 2026-03-09 10:23:05

且 RT_s 代表稳定状态的资源预约表。RT_s 组合了启动间隔为 T 的 4 次相继迭代的资源预约，表 $RT_s[0]$ 对应 $RT[0]$、$RT[2]$、$RT[4]$ 和 $RT[6]$ 的资源之和，$RT_s[1]$ 对应 $RT[1]$、$RT[3]$、$RT[5]$ 和 $RT[7]$ 的资源之和。因此稳定状态第 i 行的资源预约由下式给出：

$$RT_s[i] = \sum_{|t|(t \mod 2)=i} RT[t]$$

代表稳定状态的资源预约表称为该流水化循环的模资源预约表（modular resource- reservation table）。

为了检查软件流水调度是否会出现资源冲突，只需要检查模资源预约表即可。肯定地说，如果该表的需求能够满足，那么流水线的序曲和尾声部分肯定也能满足。

一般而言，给定启动间隔 T 和单次迭代的资源预约表 RT，在资源向量为 $\boldsymbol{R}$ 的机器上，如果对所有的 $i = 0, 1, \cdots, T-1$，都有 $RT_s[i] \leqslant R$，那么流水化的调度不会出现资源冲突。

（2）数据相关约束。软件流水中的数据相关不同于到目前为止碰到的数据相关。一个操作可能依赖于前一次迭代中同样操作的结果，这时仅用延迟来标记边已经不够用了，需要区别不同迭代中同一操作的实例。对于边 $n_1 \to n_2$，如果在第 i 次迭代中的操作 n_2，至少等第 $i-\delta$ 次迭代中的操作 n_1 开始执行 d 个时钟周期后才能执行，则该边上的标记是 (δ, d)。令 S 是软件流水调度函数，它是从数据依赖图的结点到整数的函数，并且令 T 是启动间隔。那么：

$$(\delta \times T) + S(n_2) - S(n_1) \geqslant d$$

必须满足，其中迭代次数差 δ 必须非负。对于数据依赖图 G 上的一个环，至少有一条边上标记的迭代次数差是正的。

【例 10.16】 考虑下面的循环

```
for (i = 0; i < n; i ++)
    *(p ++) = *(q ++) + c;
```

并且假定不知道 p 和 q 的值。为稳妥起见，必须认为任何一对 * (p++) 和 * (q++) 访问可能引用同样的内存单元。因此，所有的读写必须按它们原来的次序执行。假定目标机器的特征与例 10.10 所说的一样，则这段代码的数据依赖图 G 如图 10.17 所示。注意，循环控制指令被忽略。

两个相关操作的迭代次数差可以大于 1，例如：

```
for (i = 2; i < n; i ++):
    A[i] = B[i] + A[i -2]
```

显然，存储 A[i] 和读取 A[i-2] 的依赖边上标记的迭代次数差是 2。

循环中数据依赖环的出现对程序执行的吞吐能力强加了另一个限制。例如，图 10.17 中的数据依赖环在相继迭代

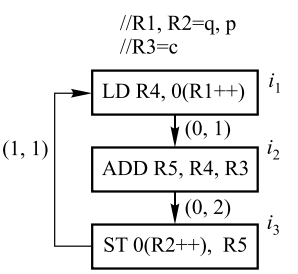


图 10.17 例 10.16 的数据依赖图

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第432页 文件编号: 422624 页面编号: 34324538445 打印人: 8685 2026-03-09 10:23:05

的两个读取操作之间强加了 4 个时钟周期的延迟。因此，该循环的执行速度不可能快于每 4 个时钟周期一次迭代。

流水化循环的启动间隔 T 不能小于

$$\max_{c \text{ is a cycle in } G} \left\lceil \frac{\sum_{e \text{ in } c} d_e}{\sum_{e \text{ in } c} \delta_e} \right\rceil$$

个时钟周期。

总之，软件流水化循环的启动间隔受每次迭代中资源使用情况的限制，即对于任何一种资源，启动间隔不能小于所需部件数和该机器所能提供部件数的比值。此外，如果循环的数据依赖图中存在环，则启动间隔会进一步受到约束，即启动间隔不能小于环中延迟的和除以迭代次数差的和。这些量中最大的一个定义了启动间隔的一个下界。

10.5.6 软件流水算法

软件流水的目标就是找到一种最小启动间隔的调度。该问题是 NP 完全的，并且能够被形式化为一个整数线性规划问题。由前面的讨论看出，如果知道了最小启动间隔，则调度算法可以利用模资源预约表来安排每个操作，以避免资源冲突。但是，只有找到了一种调度，才能知道最小启动间隔。如何解决这两者之间的相互依赖？

由前面的讨论可知，启动间隔必须大于或等于从一个循环的资源需求和数据依赖环算出的下界。如果能够找到一种正好等于这个下界的调度，就是找到了最优调度。如果找不出这样的调度，就可以用较大一点的启动间隔再尝试，直至找到一种调度。注意，如果采用启发式，而不是穷尽搜索，则有可能找不到最优调度。

能否找到接近下界的调度取决于数据依赖图的性质和目标机器的体系结构。如果数据依赖图无环并且每条机器指令仅需要一种资源的一个部件，那么很容易找到最优调度。如果有足够的硬件资源供带环的数据依赖图使用，那么很容易找到接近下界的调度。对于这种情况，很明显可以从把下界作为初始启动间隔开始，然后每次加一个时钟周期再尝试，直至找到合适的启动间隔。另一种可能方式是用二分查找来寻找启动间隔。启动间隔的上界是表调度为一次迭代产生的调度的长度。

10.5.7 无环数据依赖图的调度

为简单起见，假定被软件流水化的循环只有一个基本块。

算法 10.2 无环依赖图的软件流水。

输入：机器资源向量 $\boldsymbol{R} = [r_1, r_2, \cdots]$，其中 r_i 是机器第 i 种资源的可用部件数目；另外，还有数据依赖图 $G = (N, E)$。N 中的每个操作 n 都带有它的资源预留表 RT_n，E 中每条边 $e = n_1 \to n_2$ 标有 (δ_e, d_e)，表示 n_2 必须在比本次早 δ_e 次的迭代的 n_1 开始 d_e 个时钟周期后才能开始。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第433页 文件编号: 422624 页面编号: 34334538443 打印人: 8685 2026-03-09 10:23:05

输出：一张软件流水化调度表 S 和一个启动间隔 T。

方法：执行图 10.18 所示的程序。

```
main( ) {
    T0 = max_j ⌈ ( Σ_{n,i} RTn(i,j) ) / rj ⌉ ;
    for( T=T0, T0+1, …, 一直到 N 中的所有结点都被调度 ) {
        RT=一张 T 行的空资源预约表;
        for( N 中按照区分优先级的拓扑次序的每个结点 n ) {
            s0 = max_{e=p→n in E}( S(p)+de ) ;
            for( s=s0, s0+1, …, s0 + T −1 )
                if( NodeScheduled ( RT, T, n, s ) ) break;
            if( n 在 RT 中不能被调度 ) break;
        }
    }
}

NodeScheduled ( RT, T, n, s ) {
    RT' = RT;
    for( RTn 中每行 i )
        RT'[ (s+i) mod T ] = RT'[ (s+i) mod T ] + RTn[ i ];
    if( 对所有的 i, RT'[ i ] ≤ R ) {
        RT = RT';
        S(n) = s;
        return true;
    }
    else return false;
}
```

图 10.18 无环依赖图的软件流水算法

算法 10.2 首先依据图 G 中各操作的资源需求，计算出启动间隔的下界 T_0；然后尝试把 T_0 作为启动间隔来寻找软件流水化调度。如果它还没有找出一种调度，该算法通过逐步增加启动间隔来重复。

在每次尝试时，该算法使用一种表调度方式。它利用模资源预约表 RT 来明了稳定状态中的资源需求。操作按照区分优先级的拓扑次序来调度，因此数据相关总可以通过延迟操作而得到满足。当调度每个操作时，它首先根据数据相关的约束找出一个下界 s_0，然后调用 NodeScheduled 函数来检查在此稳定状态中是否存在资源冲突。如果存在冲突，它尝试把该

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第434页 文件编号: 422624 页面编号: 34344538441 打印人: 8685 2026-03-09 10:23:05

操作调度在下一个周期。如果该操作在 T 个连续周期的调度尝试都失败，那么由资源冲突检测的模本性知道，继续尝试是不会有结果的。因此，在这一点，该算法认为当前尝试的启动间隔不可采用，转而尝试较大一点的启动间隔。

启发式的调度尽可能极小化一次迭代的调度表长度，然而，尽早地调度一条指令会导致某些变量的生存期变长。例如，变量的读取操作倾向于尽早调度，有时会出现远早于它们的使用情况。一种简单的启发式方法是逆向调度数据依赖图，因为通常读取操作多于存储操作。

本书不再继续介绍有环数据依赖图的调度。

10.6 并行性和数据局部性优化概述

本节介绍与并行性和数据局部性优化有关的基本概念。如果操作可以被并行地执行，那么它们也可以因数据局部性或其它目的而被重排执行次序。反过来，如果一个程序中的数据相关限定指令必须串行执行，那么显然不存在并行性，也不可能有机会重排执行次序以改善数据局部性。因此，并行化分析同时是发现改进数据局部性的机会。

为了极小化并行代码中的通信，需要把所有相关的操作聚合在一起，并把它们指派到同一台处理器，这样的结果代码肯定有较好的数据局部性。在单处理器上获得良好数据局部性的一种天然方式是让该处理器相继执行指派给每个处理器的代码。

本节首先介绍并行计算机系统结构的概况，然后给出并行化的基本概念、程序循环的变换和对并行化有用的概念，再讨论类似的考虑怎样用于优化数据局部性，最后以矩阵乘算法的优化为例。

10.6.1 多处理器

最流行的并行机体系结构是对称多处理器。高性能个人计算机常常有两个处理器，许多服务器有 4 个、8 个甚至 10 个处理器。由于把多个高性能处理器集成在一块芯片上是可行的，因此多处理器已经被广泛使用。

对称多处理器上的各处理器共享同样的地址空间。为了通信，一个处理器直接写一个内存单元，然后该单元被其他处理器读取。对称多处理器只所以这样命名。是因为所有处理器以统一的存取时间访问该系统中所有内存。图 10.19 显示了对称多处理器的高层体系结构。各处理器可以有它们自己的一级缓存和二级缓存，在某些情况下甚至有三级缓存。级数最高的缓存通过共享总线连到内存。

对称多处理器使用一种**一致性缓存协议**来对程序员隐藏缓存的存在。在这样的协议下，允许几个处理器同时保持同样缓存行的副本，只要它们都只读这些数据。当一个处理器打算写一个缓存行时，它在其他缓存上的副本被删除。当一个处理器请求的数据不在它的缓存上时，该请求到达共享总线，该数据从内存或从另一个处理器的缓存取出。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第435页 文件编号: 422624 页面编号: 34354538448 打印人: 8685 2026-03-09 10:23:06

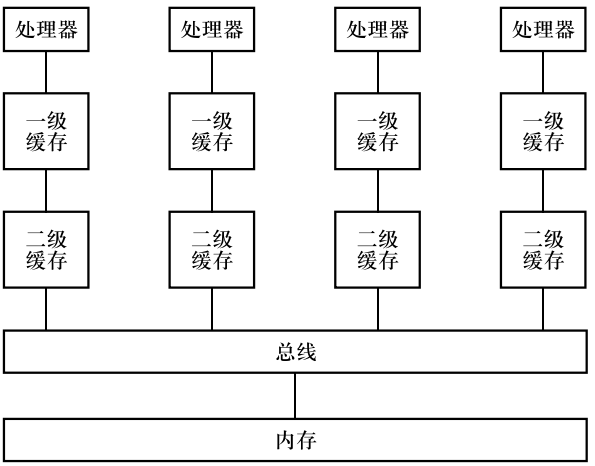


图 10.19 对称多处理器的高层体系结构

一个处理器和另一个处理器通信所需的时间大约是一次内存访问代价的 2 倍。缓存行中的数据必须首先从第一个处理器的缓存写到内存，然后从内存取到第二个处理器的缓存。读者可能认为处理器之间的通信是相对廉价的，因为它只不过是一次内存访问时间的 2 倍。但是，要记住，和缓存命中相比，内存访问的代价是非常大的，其耗时可能是前者的 100 倍。该分析使人认识到有效并行化和局部性分析之间的相似性。为了使一个处理器（在自身单处理器或多处理器环境上）的性能更好，它必须在它自己的缓存中找到它操作的大部分数据。

21 世纪初，对称多处理器的设计已经不再超过 10 个处理器的规模，因为共享总线或为此目的的任何其它方式互联，其操作不能像增加处理器个数那样增速。为了使处理器设计可扩展，体系结构师在内存分层中又引入一层。他们将内存分布到各处理器，使得每个处理器能迅速访问它自己的局部内存，如图 10.20 所示。较远的内存则构成内存分层中的下一个层次，它们容量较大，但需要较长的访问时间。类似于存取速度越快则容量越小的内存分层设计原则，支持处理器之间快速通信的机器只有少量处理器。

带分布内存的并行机有两种衍生：非均匀内存访问的机器和消息传递的机器。非均匀内存访问的机器给软件提供共享内存空间，允许处理器通过读写共享内存来进行通信。在消息传递的机器上，各处理器有不相交的地址空间，处理器之间通过发送消息进行通信。相比之下，为共享内存机器写代码是比较容易的；但是无论哪种机器，为获得良好的性能，软件都必须有很好的局部性。

10.6.2 应用中的并行性

有两种衡量标准可用来估计并行应用的性能：一种是**并行覆盖**，是指整个计算中并行运行部分的百分比；另一种是**并行粒度**，是指每台处理器上无需和其它处理器同步或通信的计

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第436页 文件编号: 422624 页面编号: 34364538446 打印人: 8685 2026-03-09 10:23:06

算量。对并行化来说，循环是一个特别有吸引力的地方，一个循环可以有许多次迭代计算，如果这些计算相互独立，则它们是并行计算的主要来源。

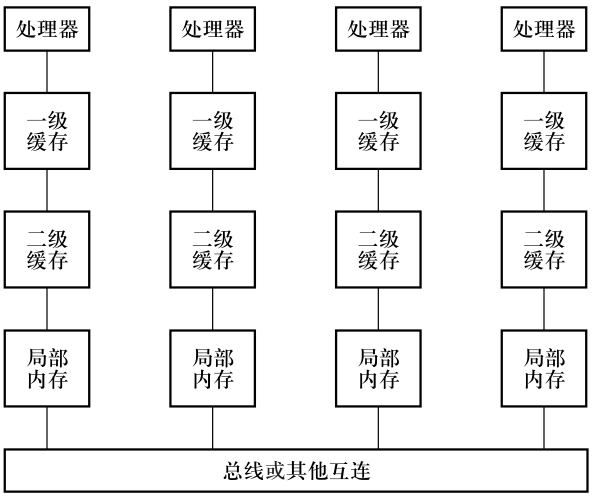


图 10.20 分布式内存储器

并行覆盖的意义由 **Amdahl 定律**简洁地表达。

Amdahl 定律：如果 f 是被并行化的代码占的比例，并行化后的版本在一台有 p 个处理器的机器上运行且无通信或并行开销，那么加速比是

$$\frac{1}{(1-f)+(f/p)}$$

例如，若有一半计算仍然是串行的，则不管使用多少个处理器，计算速度最多只能是加倍。如果是 4 个处理器，那么加速比是 1.6。即使并行覆盖部分占到 90%，在 4 个处理器的情况下，加速比最多也只能达到 3。当处理器个数无限时，加速比也只能达到 10。

对于并行粒度，理想的情况是一个应用的整个计算能够被划分成许多独立的粗粒度任务，因为这时只需直接把不同的任务指派给不同的处理器即可。一个典型例子是地外文明探索（search for extra-terrestrial intelligence）项目，它使用联到因特网上各家庭计算机的闲置计算能力来分析世界上最大射电望远镜所获得数据的不同部分，以帮助科学家探索外星生命。参与该项目的每台计算机只需要少量输入，产生少量输出，并且以独立于其它所有参与该项目计算机的方式工作。虽然因特网相对来说有较高的通信延迟和较低的带宽，但这样的计算能很好地在联到因特网的这些计算机上运行。

大多数应用虽然需要处理器之间的更多通信和交互，但仍然允许粗粒度的并行。例如，考虑一台 web 服务器，它为某个公共数据库的大量访问请求提供服务，这些请求大都相互独立。这个应用可以运行在一个多处理器上，用一个线程实现数据库，用一组其它线程来服务用户

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第437页 文件编号: 422624 页面编号: 34374538444 打印人: 8685 2026-03-09 10:23:06

请求。又如，包括药物设计和机翼模拟，其中许多不同参数的结果可以独立地评估。有时，就是对一次模拟中一组参数的评估也需要很长时间，因而希望通过并行来加速。当一种应用中可用并行的粒度降低时，则需要较好的处理器间通信的支持和更多的编程努力。

许多控制结构简单、数据量大并且计算时间长的科学和工程应用，很容易以较细粒度（和上面提到的应用相比）被并行化。下面给出这类程序的一个例子。

10.6.3 循环级并行

循环是并行化的主要对象，特别是那些使用数组的应用。长时间运行的应用一般都使用大数组，这些数组导致程序中出现有许多次迭代的循环，每次迭代用于计算这些数组的一个元素，而且这些迭代经常是相互独立的。这类循环的大量迭代可以分派到各处理器上。如果在每个处理器上完成的工作大体上是相同的，那么直接把这些迭代平分到各处理器上即可获得最大的并行性。例 10.17 是一个展示怎样利用循环级并行的简单例子。

【例 10.17】 循环

```
for (i = 0; i < n; i++) {
    Z[i] = X[i] - Y[i];
    Z[i] = Z[i] * Z[i];
}
```

计算向量 $\boldsymbol{X}$ 和 $\boldsymbol{Y}$ 对应分量差的二次方，并把结果作为向量 $\boldsymbol{C}$ 的对应分量。该循环可以并行，因为每次迭代访问一组不同的数据。在一台有 M 个处理器（它们的唯一标识分别为 $p = 0, 1, \cdots, M-1$）的计算机上，该循环可以由每个处理器执行下面相同的代码来完成：

```
b = ceil (n/M);
for (i = b p; i < min(n, b (p+1)); i++) {
    Z[i] = X[i] - Y[i];
    Z[i] = Z[i] * Z[i];
}
```

注意，由于迭代的次数并不一定能被 M 整除，因此通过使用求最小值函数来保证最后一个处理器的执行不会超出原循环的上界。

例 10.17 的并行代码是一个单程序多数据的程序。所有的处理器执行同样的代码，但是它由每个处理器的唯一标识来参数化，所以不同的处理器完成的是不同的任务。典型的情况是，一个处理器作为主处理器，执行该计算的所有串行部分。当主处理器到达该代码的一个并行化部分时，它唤醒所有的从处理器，这些从处理器执行该并行化的代码。当到达该并行化代码的末尾时，所有这些处理器都参与一个**障碍同步**。在所有其它处理器被允许离开该同步障碍并执行障碍之后的操作之前，一个处理器进入该同步障碍之前的任何操作由系统确保完成。

如果仅并行化像例 10.17 这样的小循环，则结果代码很可能有较低的并行覆盖和相对细

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第438页 文件编号: 422624 页面编号: 34384538442 打印人: 8685 2026-03-09 10:23:06

粒度的并行。因此，更喜欢做的是去并行化一个程序的最外层循环，因为它产生最粗粒度的并行。一个典型的例子是 $n \times n$ 数据集上的二维快速傅里叶变换。很多应用没有可以并行且上规模的最外层循环。这些应用的执行时间经常由耗时的**内核**（kernel）支配，这样的内核可能是嵌套层次不同的循环组成的几百行代码。在有些情况下，重组内核的计算，通过聚焦在它的局部性上，把它划分成最大部分的、独立的若干计算单元是可能的。

寻找循环迭代以外的并行性也是可能的。例如，可以把两个不同的函数调用或两个不同的循环指派给两个不同的处理器。这种形式的并行被认为是**任务级并行**。对并行化来说，任务级不像循环级那样有吸引力。因为对一个程序而言，独立的任务数是一个常数，它不像典型的循环那样，独立的计算单元随迭代次数的增加而增加。任务通常不是等规模的，因此很难保证所有的处理器在所有时间都处于忙碌。

10.6.4 数据局部性

在并行化程序时，有两个不同的数据局部性概念需要考虑，它们在 6.5.3 节讨论程序局部性时已经被提到。如果某个单元的数据在很短的时间被多次使用，就会出现**时间局部性**；如果毗邻单元的数据在很短的时间内都被访问，就会出现**空间局部性**。同一个缓存行上的元素一起被使用的情况是空间局部性的一种重要形式。因为某缓存行上的一个元素被需要时，该行上的所有元素就都被取到缓存，并且如果它们很快被使用，则很可能还在缓存中。由于这种空间局部性将缓存未命中降到最低，因此使程度获得明显的加速。

程序的内核经常可以写成语义等价而数据局部性和性能大不相同的多种形式。例 10.18 是表达例 10.17 计算的另一种方法。

【例 10.18】 下面的程序和例 10.17 一样，计算向量 $\boldsymbol{X}$ 和 $\boldsymbol{Y}$ 对应元素差的二次方。

```
for (i = 0; i < n; i++) {
    Z[i] = X[i] - Y[i];
}
for (i = 0; i < n; i++) {
    Z[i] = Z[i] * Z[i];
}
```

第一个循环计算对应元素的差，第二个循环计算它们的二次方。例 10.17 是把这里的两个循环融合成一个循环。在实际程序中经常出现本例这样的代码，因为它对向量机来说是一种优化形式，向量机是一种一次能对整个向量完成简单算术运算的超级计算机。

例 10.17 被融合循环的性能较好，因为它有较好的数据局部性。每个差计算出来后就立即计算它的二次方。事实上，差通常保存在寄存器中，然后计算它的二次方，再把结果写到内存单元 Z[i]，而且就只写这一次。而本例需要取 Z[i] 一次，写 Z[i] 两次。更糟的是，如果数组 Z 大于缓存，那么在第二次使用 Z[i] 时需要从内存重新取。因此，本例的代码慢的多。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第439页 文件编号: 422624 页面编号: 34394538440 打印人: 8685 2026-03-09 10:23:06

【例 10.19】 假定给以行为主（见 7.3.2 节）的数组 Z 的所有元素置零。图 10.21(a)和图 10.21(b) 的程序分别逐列地和逐行地扫过该数组。这两个程序可以互相变换到对方。根据空间局部性，显然更愿意逐行地给该数组元素置零，因为在一个缓存行中的所有字相继置零。而对于逐列的方式，如果列的大小超过缓存大小，虽然每个缓存行在外循环的下次迭代时再次被使用，但是缓存行在再次使用前已经被抛弃。

为了获得最好的性能，应该像例 10.17 那样并行化图 10.21(b) 的外循环，如图 10.21(c)所示。

```
for(j=0;j<n;j++)
    for(i=0;i<n;i++)
        Z[i][j]=0;
```
(a) 逐列置零

```
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        Z[i][j]=0;
```
(b) 逐行置零

```
b=ceil(n/M);
for(i=b*p;i<min(n,b*(p+1));i++)
    for(j=0;j<n;j++)
        Z[i][j]=0;
```
(c) 逐行并行置零

图 10.21 数组元素置零的顺序和并行代码

上面两个例子说明了操作在数组上的数值应用的几个重要特征。

（1）数组代码经常有许多可以并行化的循环。

（2）当循环有并行性时，它们的迭代可按任意次序执行，因而可以重新安排它们的计算次序以彻底改进数据局部性。

（3）在创建相互独立的并行计算大单元时，串行执行这些单元往往会产生较好的数据局部性。

10.6.5 矩阵乘法算法

本节和 10.6.6 节以矩阵乘法算法为例，来说明即使是简单和易于并行化的程序，优化也是不容易的。通过这两节的讨论将会明白，和直截了当的程序相比，重写代码能怎样有效地改进数据局部性，让缓存行既能保存连续数据元素，也能够有效地缩短像矩阵乘法这种程序的运行时间。图 10.22是一个典型的矩阵乘法程序。

该程序取两个 $n \times n$ 的矩阵 $\boldsymbol{X}$ 和 $\boldsymbol{Y}$，产生它们的积在第三个 $n \times n$ 的矩阵 $\boldsymbol{Z}$ 中。该图的代码产生 n^2 个结果，每个结果是矩阵 $\boldsymbol{X}$ 的一行和矩阵 $\boldsymbol{Y}$ 的一列的内积。显然，$\boldsymbol{Z}$ 中每个元素的计算是独立的，它们可以并行地执行。

n 的值越大，该算法计算 $\boldsymbol{Z}$ 中每个元素需要的时间就越长。因为这 3 个矩阵虽然只有

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第440页 文件编号: 422624 页面编号: 34404538448 打印人: 8685 2026-03-09 10:23:06

$3n^2$ 个存储单元，但是该算法完成 n^3 次操作，每次操作完成 $\boldsymbol{X}$ 中的一个元素和 $\boldsymbol{Y}$ 中的一个元素相乘，再把积加到 $\boldsymbol{Z}$ 中的一个元素上（后面简称乘加计算）。因此，该算法是计算密集型的，原则上内存访问不应该构成瓶颈。

```
for(i=0;i<n;i++)
    for(j=0;j<n;j++) {
        Z[i][j]=0.0;
        for(k=0;k<n;k++)
            Z[i][j]=Z[i][j]+X[i][k]*Y[k][j];
    }
```

图 10.22　基本的矩阵乘法算法

先考虑该程序在单处理器上顺序执行时的行为。最内循环读写 $\boldsymbol{Z}$ 中的同一个元素，并且使用 $\boldsymbol{X}$ 的一行和 $\boldsymbol{Y}$ 的一列。$Z[i][j]$ 可以很容易地保存在寄存器中，因而除了最后一次存储，无须访问内存。不失一般性，假定矩阵的布局是以行为主，并且为简单起见，假定正好 c 个数组元素能够放满一个缓存行。

图 10.23 表达执行图 10.22 最外循环一次迭代时的访问模式。图 10.23 具体表示的是 $i=0$ 时的第一次迭代。当从 $\boldsymbol{X}$ 中第一行的一个元素移到下一个元素时，$\boldsymbol{Y}$ 中某列的所有元素被访问。从图 10.23 中可以看出，所假设的矩阵到缓存行的组织，每个小长方形代表保存 4 个数组元素的一个缓存行（该图中取 $c=4$，$n=12$）。

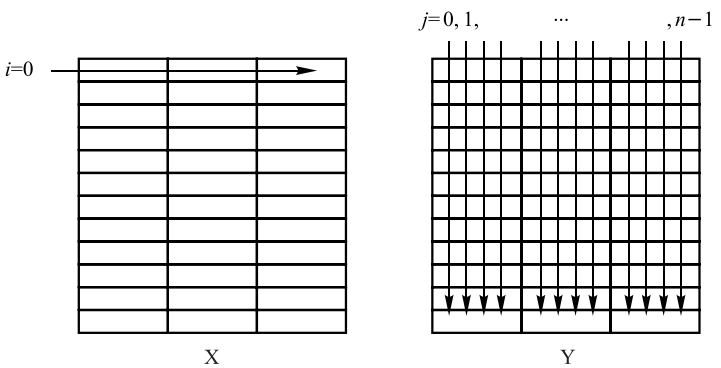


图 10.23　在矩阵乘中数据访问模式

访问 $\boldsymbol{X}$ 几乎不给缓存带来什么负担。$\boldsymbol{X}$ 中的一行仅散布在 n/c 个缓存行上，假定缓存足以放下 $\boldsymbol{X}$ 中所有的缓存行，那么对 $\boldsymbol{X}$ 中的每行来说，缓存未命中仅出现 n/c 次。对整个 $\boldsymbol{X}$ 来说，一共出现 n^2/c 次，不可能比它更小了。

但是，当使用 $\boldsymbol{X}$ 中的完整一行时，该矩阵乘法算法需要逐列访问 $\boldsymbol{Y}$ 中的所有元素。也就是说，当 $j=0$ 时，内循环需要把 $\boldsymbol{Y}$ 中第一列的所有元素都取到缓存，而这些元素分散在

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第441页 文件编号: 422624 页面编号: 34414538446 打印人: 8685 2026-03-09 10:23:06

n 个不同的缓存行中。如果缓存足够大（或 n 足够小）到能保存 n 个缓存行，并且缓存的其它应用也不会把它们从缓存中驱逐，那么当需要 $\boldsymbol{Y}$ 中的第二列（$j=1$）时，这些数据已经在缓存，此时不会出现缓存未命中。一直到 $j=c$ 时才需要把 $\boldsymbol{Y}$ 中完全不同的另一组缓存行取到缓存。因此，对外循环的第一次（$i=0$）迭代，缓存未命中次数在 n^2/c 和 n^2 之间，取决于在缓存中 $\boldsymbol{Y}$ 的缓存行，能否从第二层循环的一次迭代驻留到下一次迭代需要它的时候。

在对 $i=1,2,\cdots,n-1$ 逐步完成最外循环的过程中，当需要 $\boldsymbol{Y}$ 时，可能还会出现或者根本不出现缓存未命中。如果缓存足够大，存放 $\boldsymbol{Y}$ 中的所有 n^2/c 个缓存行都可以驻留在缓存上，就不会再出现缓存未命中。这时该算法的所有缓存不命中共 $2n^2/c$ 次，一半为 $\boldsymbol{X}$，另一半为 $\boldsymbol{Y}$。然而，如果缓存只能驻留 $\boldsymbol{Y}$ 中的一列数据而不是 $\boldsymbol{Y}$ 中所有数据，那么对最外循环的一次迭代，需要重新将 $\boldsymbol{Y}$ 中的所有数据取到缓存。这时缓存未命中的次数是 $n^2/c+n^3/c$，前一部分为 $\boldsymbol{X}$，后一部分为 $\boldsymbol{Y}$。最坏情况是，缓存连 $\boldsymbol{Y}$ 中的一列数据也不能驻留，则对最外循环的一次迭代，出现 n^2 次缓存未命中，该算法总共有 $n^2/c+n^3$ 次缓存未命中。

现在考虑怎样使用 p 个处理器来加速图 10.22 所示矩阵乘法程序的执行。并行化该程序的一种明显方法是把 $\boldsymbol{Z}$ 不同行的计算指派到不同处理器，每个处理器计算 $\boldsymbol{Z}$ 的连续 n/p 行（为简单起见，假定 n 可以由 p 整除）。这样，每个处理器需要访问矩阵 $\boldsymbol{X}$ 和 $\boldsymbol{Z}$ 的 n/p 行以及整个 $\boldsymbol{Y}$；每个处理器将完成 n^3/p 次乘加运算来实现对 $\boldsymbol{Z}$ 中的 n^2/p 个元素的计算。

虽然计算时间与 p 成比例减少，但通信代价与 p 成比例增加。因为每个处理器需要读 $\boldsymbol{X}$ 中的 n^2/p 个元素和 $\boldsymbol{Y}$ 中的所有元素，那么交付给 p 个处理器之缓存的总缓存行是 $n^2/c+pn^2/c$，这两项分别代表交付 $\boldsymbol{X}$ 和交付 $\boldsymbol{Y}$ 的 p 个副本。当 p 逼近 n 时，计算时间复杂度为 $O(n^2)$，而通信代价为 $O(n^3)$，也就是在内存和处理器之间传送数据的总线成为瓶颈。因此，按现在的数据布局，使用大量处理器来分享这些计算，不仅不能加速，还会使计算速度降低。

10.6.6 矩阵乘法算法的优化

图 10.22 所示的矩阵乘法算法表明，一个算法复用同样的数据并不代表它的数据局部性就好，关键是看复用数据时能否做到缓存命中。若要做到缓存命中，则复用应该很快发生，而且在数据从缓存转移出去以前。在该算法中，n^2 个乘加操作隔开了矩阵 $\boldsymbol{Y}$ 中同一个数据的复用，n 个乘加操作隔开了 $\boldsymbol{Y}$ 中同一个缓存行的复用。另外，在一个处理器上，数据只有由本处理器复用时才可能出现缓存命中。在 10.6.6 节考虑并行实现时已经看到，每个处理器都需要使用 $\boldsymbol{Y}$ 中所有的元素。这样的 $\boldsymbol{Y}$ 复用不会带来所期望的数据局部性。

改进程序数据局部性的一种方法是改变其数据结构的布局。例如，若把 $\boldsymbol{Y}$ 改成以列为主次序存储，则为矩阵 $\boldsymbol{Y}$ 改进了缓存行的复用。但这种方法的可用性有限，因为同一个矩阵通常会用于不同的操作中。如果在另一个矩阵乘法中，$\boldsymbol{Y}$ 扮演 $\boldsymbol{X}$ 的角色，那么以列为主的存放就给它带来困难，因为在矩阵乘法中，第一个矩阵以行为主的次序为好。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第442页 文件编号: 422624 页面编号: 34424538444 打印人: 8685 2026-03-09 10:23:06

有时，改变指令的执行次序可以改进数据局部性。10.6.4 节已经见过这样的例子。但是，互换内外循环的技术并不能改进矩阵乘法程序。如果互换图 10.22 所示矩阵乘法程序的最外层和第二层的循环，也就是每次产生矩阵 $\boldsymbol{Z}$ 的一列而不是一行，若矩阵仍然是以行为主的次序，则矩阵 $\boldsymbol{Y}$ 可从这种互换中获得较好的空间和时间局部性，但是矩阵 $\boldsymbol{X}$ 的局部性下降了。

分块是重排循环中迭代次序的另一种方法，它能够极大地改进程序的局部性。分块就是把一个矩阵分成若干子矩阵，或称为块，如图 10.24 所示。分块是为了以块为单位来完成计算，即整个块在很短的时间内使用，而不是原来那样每次计算矩阵的一行或一列。在典型情况下，块为正方形，如长度为 b。如果 b 能够整除 n，那么所有的块都是正方形；否则靠近底边和右边的块有一个边或相邻两边都小于 b。

图 10.25 给出了基本矩阵乘法算法的另一个版本，其中 3 个矩阵都正好能分成边长为 b 的若干正方形。如果 b 不能整除 n，则第 (4) 行的循环上界需要修改成 “min(ii+b, n)”，并且第 (5)、(6) 行也要做相应修改。假定矩阵 $\boldsymbol{Z}$ 中的所有元素已经初始化为 0。

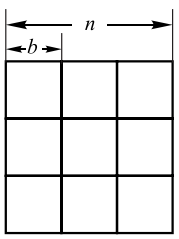


图 10.24 矩阵分块

```
(1)   for(ii=0;ii<n;ii=ii+b)
(2)     for(jj=0;jj<n;jj=jj+b)
(3)       for(kk=0;kk<n;kk=kk+b)
(4)         for(i=ii;i<ii+b;i++)
(5)           for(j=jj;j<jj+b;j++)
(6)             for(k=kk;k<kk+b;k++)
(7)               Z[i][j]=Z[i][j]+X[i][k]*Y[k][j];
```

图 10.25 分块的矩阵乘法

从第 1 ～ 3 行的 3 个较外循环分别使用 ii、jj 和 kk 作为循环控制变量，并且它们的值每次都增加 b，因此它们总是用来定位块的左上角。对一组确定的 ii、jj 和 kk，从第 4 ～ 7 行的程序用来计算左上角为 X[ii][kk] 和 Y[kk][jj] 的两块对左上角为 Z[ii][jj] 的块的贡献。

如果 b 取得适当，该算法和图 10.22 的基本算法相比，并且在缓存不足以装下整个 $\boldsymbol{X}$、$\boldsymbol{Y}$ 和 $\boldsymbol{Z}$ 的情况下，可以大大减少缓存未命中的数量。现在适当选择 b，使 3 个矩阵都各有一个块可以装到缓存。由于这些循环的次序，$\boldsymbol{Z}$ 的每块实际只需要取到缓存中一次，因此和 10.6.5 节对基本算法的分析一样，在这里不考虑由 $\boldsymbol{Z}$ 引起的缓存未命中。

为了把 $\boldsymbol{X}$ 或 $\boldsymbol{Y}$ 的一个块取到缓存，会出现 b^2/c 次缓存未命中，c 仍然是一个缓存行的元素个数。对于 $\boldsymbol{X}$ 和 $\boldsymbol{Y}$ 的一对块，图 10.25 从第 4 ～ 7 的程序需要完成 b^3 次乘加计算。因为整个矩阵乘法需要 n^3 次乘加计算，所以取一对块到缓存的操作的总次数是 n^3/b^3。由于对于 $\boldsymbol{X}$ 和 $\boldsymbol{Y}$ 的一对块会有 $2b^2/c$ 次缓存未命中，因此缓存未命中的总次数是 $2n^3/bc$。

把 $2n^3/bc$ 和 10.6.5 节的估计进行比较是有意义的。在那里的估计是，如果 3 个完整的矩阵都能装到缓存，那么缓存未命中的数量是 $O(n^2/c)$。在这种情况下，可以取 $b=n$，即让

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第443页 文件编号: 422624 页面编号: 34434538442 打印人: 8685 2026-03-09 10:23:06

整个矩阵是一个块，则得到的缓存未命中估计再次为 $O(n^2/c)$。另外，并非 3 个完整的矩阵都能装到缓存时，10.6.5 节的估计是出现 $O(n^3/c)$ 次缓存未命中，甚至 $O(n^3)$ 次缓存未命中。在这种情况下，假定 b 能够取得比较大（如 200，那么 8B 一个元素的 3 个块可以装到一个 1 MB 的缓存中的），则从 $2n^3/bc$ 可以看出使用分块方法是有很大好处的。

分块技术可以用到内存分层的每层。例如，可以把一个 2×2 矩阵乘法的运算对象保存在寄存器中，以优化寄存器的使用。为不同层次的缓存和物理内存按序选择越来越大的块。

类似地，可以把块分布在各处理器上，以最小化数据通信量。经验已经表明，这样的优化可以使单处理器的性能提升到原来的 3 倍，在多处理器上的加速接近和所使用的处理器个数成线性关系。

然而，缓存的实际情况并非本节所介绍的这么简单，通常一个缓存行不能放置在缓存中任意的地方。因此，上面所介绍的方法还要依据缓存的约束进行调整。

习　题

10.1　严格按照括号指定的次序，计算表达式 $((\mathrm{u}+\mathrm{v})+(\mathrm{w}+\mathrm{x}))+(\mathrm{y}+\mathrm{z})$，也就是不利用交换律或结合律来改变加法的计算次序。像例 10.3 那样给出提供最大可能并行的寄存器级的机器代码。

10.2　为下面的表达式完成习题 10.1 的要求：

（a）$(\mathrm{u}+(\mathrm{v}+(\mathrm{w}+\mathrm{x})))+(\mathrm{y}+\mathrm{z})$；

（b）$(\mathrm{u}+(\mathrm{v}+\mathrm{w}))+(\mathrm{x}+(\mathrm{y}+\mathrm{z}))$。

若用最小化寄存器数来取代最大化并行，则需要多少步计算？用最大化并行节省了几步？

10.3　习题 10.1 的表达式可以由下面的指令序列来执行。如果所需要的并行都能得到，那么这些指令的执行需要几步？

```
LD R1, u          // R1 = u
LD R2, v          // R2 = v
ADD R1, R1, R2    // R1 = R1 + R2
LD R2, w          // R2 = w
LD R3, x          // R3 = x
ADD R2, R2, R3    // R2 = R2 + R3
ADD R1, R1, R2    // R1 = R1 + R2
LD R2, y          // R2 = y
LD R3, z          // R3 = z
ADD R2, R2, R3    // R2 = R2 + R3
ADD R1, R1, R2    // R1 = R1 + R2
```

10.4　把例 10.4 讨论的代码片段翻译成采用 CMOVZ 条件转移指令（10.2.6 节）的形式。在你的代码中数据相关出现在什么地方？

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第444页 文件编号: 422624 页面编号: 34444538440 打印人: 8685 2026-03-09 10:23:06

10.5 画出图 10.26 各代码片段的数据依赖图。

```
LD R1,a          LD R1,a          LD R1,a
LD R2,b          LD R2,b          LD R2,b
SUB R3,R1,R2     SUB R1,R1,R2     SUB R3,R1,R2
ADD R2,R1,R2     ADD R2,R1,R2     ADD R4,R1,R2
ST a,R3          ST a,R1          ST a,R3
ST b,R2          ST b,R2          ST b,R4
   (a)              (b)              (c)
```

图 10.26 习题 10.5 的机器代码

10.6 假定机器有一个算术运算部件 ALU（用于 ADD 和 SUB 操作）和一个内存资源 MEM（用于 LD 和 ST 操作）。再假定，除了 LD 操作需要两个时钟周期外，所有操作都只需要一个时钟周期。像例 10.6 那样，在 LD 操作一个内存单元启动一个时钟周期后，操作同一个内存单元的 ST 可以启动。为图 10.26 中的各代码片段寻找最短调度。

10.7 根据下面的资源情况，重做习题 10.6。

(a) 机器有一个 ALU 资源和两个 MEM 资源。

(b) 机器有两个 ALU 资源和一个 MEM 资源。

(c) 机器有两个 ALU 资源和两个 MEM 资源。

10.8 假定机器模型和习题 10.6 的一样。

(a) 为下面的代码片段画出数据依赖图。

```
LD R1, a
ST b, R1
LD R2, c
ST c, R1
LD R1, d
ST d, R2
ST a, R1
```

(b) 在 (a) 答案的图中，关键路径是哪几条？

10.9 考虑下面的代码片段

```
if (x == 0) a = b;
else a = c;
d = a;
```

假定机器使用例 10.6 的延迟模型，还假设机器一次可以同时执行任意两条指令。找出该程序片段可能的最短执行。

10.10 下面是一个循环体的代码

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第445页 文件编号: 422624 页面编号: 34454538447 打印人: 8685 2026-03-09 10:23:06


```
L: LD R1, a(R9)
   ST b(R9), R1
   LD R2, c(R9)
   ADD R3, R1, R2
   ST c(R9), R3
   SUB R4, R1, R2
   ST b(R9), R4
   BL R9, L
```

a(R9) 这样的地址表示内存单元，其中 a 是常量，R9 是记录循环迭代次数的寄存器。假定每次迭代访问不同的内存单元，因为 R9 有不同的值。使用例 10.10 的机器模型，按下面的方式调度该循环。

（a）让每次迭代尽可能紧凑（在每个算术操作后只引入一个 NOP 操作），循环展开两次。在不违反机器限制的前提下，尽早启动第二次迭代。

（b）重复（a）的工作，但循环展开 3 次。同样，在不违反机器限制的前提下，尽早启动每次迭代。

***10.11** 使用例 10.10 的机器模型，为循环

```
for (i = 1; i < n; i++) {
   A[i] = B[i -1] + 1;
   B[i] = A[i -1] + 2;
  }
```

找出最小启动间隔和对此间隔的一种调度。注意，迭代的计数通过寄存器的自动减 1 来完成，无需单独完成此计数的操作。

***10.12** 证明，在每个操作对每种资源仅需要一个部件的特殊情况下，算法 10.2 总能够找到满足下界的软件流水调度。

10.13 图 10.25 基于块的矩阵乘法算法没有把矩阵 $\boldsymbol{Z}$ 初始化为零，请加上把 $\boldsymbol{Z}$ 初始化为零的步骤。

第 11 章

编译系统和运行时系统

通常，除了编译器，我们还需要一些其他工具的帮助，才能得到可执行的目标程序，这些工具包括预处理器、汇编器和连接器等。对于 FORTRAN 和 C 等语言，这些工具都较简单。了解这些工具有助于掌握从源程序到可执行目标程序的实际处理过程，这些知识对于参与大型软件系统的开发很有用。本章介绍 C 语言编译系统。

此外，目标代码运行时，还需要一些工具的支撑，如动态连接程序、无用单元收集程序等，这些工具的集合称为运行时系统。本章还介绍 Java 语言的运行时系统及其无用单元收集程序。

11.1 C 语言的编译系统

除了编译器，我们还需要其他工具来建立一个可执行的目标程序。本节以 GNU 编译器套件（简称 GCC 系统）为例，说明编程语言编译系统的一般工作过程。

一个 C 源程序可以分成若干个模块，且存储在不同的文件中。C 编译系统先对这些源文件分别进行预处理、编译和汇编，形成可重定位的目标文件；再利用连接器将这些目标文件和必要的库文件连接成一个可执行的目标文件，即具有绝对地址的机器代码。这一过程可用图 11.1 描绘。

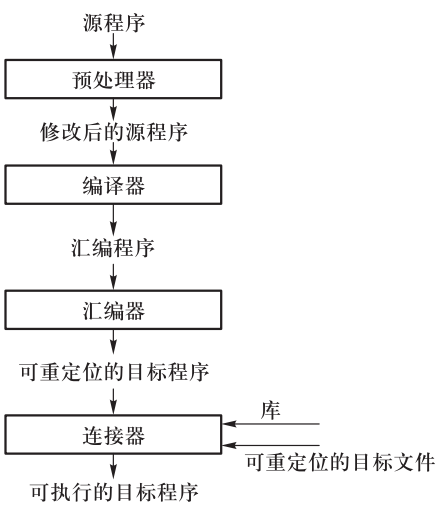


图 11.1 一个语言编译系统

大多数编译系统提供一个**驱动程序**来调用语言的预处理器、编译器、汇编器、连接器，以支持用户完成从源程序到可执行程序的翻译。在 GCC 系统中，编译 C 语言程序的驱动程序的名字是 gcc（或 cc）。

下面结合一个 C 语言的程序实例来讨论 GCC 系统的工作步骤。图 11.2 中的程序由 main.c 和 swap.c 两

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第447页 文件编号: 422624 页面编号: 34474538443 打印人: 8685 2026-03-09 10:23:06

个文件组成，为便于引用中间的语句，增加了行号。在 UNIX（还有 Linux）环境下，输入如下命令可以得到该程序的可执行文件 swap：

```
gcc -v -o swap main.c swap.c
```

这里，使用选项-v 可以输出该编译系统各步骤执行的命令和执行结果，选项-o 后紧跟的字符串指示生成的可执行文件的名字。

main.c

```
1  //#include <stdio.h>
2  # if 1
3  int buf[2];
4  # else
5  int buf[2] = {10, 20};
6  # endif
7  void swap();
8  # define A buf[0]
9  int main()
10 {
11     scanf("%d, %d", buf, buf+1);
12     swap();
13     printf("%d, %d", A, buf[1]);
14     return 0;
15 }
```

swap.c

```
1  extern int buf[2];
2  int * bufp0 = buf;
3  int * bufp1;
4  void swap()
5  {
6      int temp;
7      bufp1 = buf + 1;
8      temp = *bufp0;
9      *bufp0 = *bufp1;
10     *bufp1 = temp;
11 }
```

图 11.2 main.c 和 swap.c 两个文件组成的程序

11.1.1 预处理器

1. 预处理器概述

gcc 首先调用**预处理器** cpp，将源程序文件翻译成一个 ASCII 码中间文件，它是经修改后的源程序。图 11.3 是执行 “cpp main.c main.i” 或 “gcc -E main.c -o main.i” 命令后，main.c 被翻译为中间文件 main.i（开头做了删减）。注意，生成的中间文件也随 GCC 版本的不同而不同。

预处理器产生编译器的输入，它实现以下功能。

（1）文件包含。预处理器可以把源程序文件中的包含声明（`#include`）展开为程序正文。例如，当源程序文件含有语句

```
#include <stdio.h>
```

时，预处理器会在系统标准路径下搜索 stdio.h，再用 stdio.h 文件中的内容来代替这条语句。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第448页 文件编号: 422624 页面编号: 34484538441 打印人: 8685 2026-03-09 10:23:06

（2）宏展开。C 程序可以使用#define 来定义**宏**。一个宏定义给出一段 C 程序代码的缩写。预处理器将源程序文件中出现的对宏的引用展开成相应的宏定义，这一过程称为**宏展开**。例如，main.c 的第 8 行为宏 A 的定义，第 13 行中的 A 是对该宏的引用。在预处理后产生的 main.i 中，宏 A 的定义转换成一个空行，对宏 A 的引用则展开成 buf[0]。

（3）条件编译。预处理器根据#if 和#ifdef等编译命令及其后的条件，将源程序中的某部分包含进来或排除在外。通常把排除在外的语句转换成空行。

显然，实现一个这样的预处理器并不困难。有些语言的预处理器用于增强老的语言，使之包含现代的控制结构和数据类型。从增强语言到老语言的翻译由这样的预处理器完成。

```
1  # 1 "main.c"
2
3
4  int buf[2];
5
6
7
8  void swap();
9
10 int main()
11 {
12     scanf("%d, %d", buf, buf+1);
13     swap();
14     printf("%d, %d", buf[0], buf[1]);
15     return 0;
16 }
```

图 11.3 main.i 的内容

2. 宏的种类与应用

宏是一种用于代码生成和替代的编程工具，其种类和应用方式因语言而异。根据工作层次，宏可分为*词法级*和*语法级*两种。词法级宏（如 C 语言中的宏）主要在编译前对代码进行简单的文本替换，而语法级宏（如 Lisp 家族语言、Rust 和 Scala）则能直接操作语法树，实现更高级的代码生成和模式匹配。

在宏展开过程中，不同宏机制的实现展现出独有的特性。例如，Hygienic 宏[2,4] 确保命名安全，避免命名冲突。如果宏调用和宏定义中使用同样的变量名，Hygienic 宏会自动对宏内部变量改名以避免覆盖上下文变量。普通宏则缺乏这种机制，可能导致隐藏错误。

Macrofication[11] 是一种逆向宏展开的技术，通过动态模式匹配识别代码片段并替换为宏调用，它用于 JavaScript 的 Hygienic 宏系统，提升了代码表达能力和灵活性。

宏在程序优化、代码复用和领域特定语言的设计中都有广泛应用。通过减少重复代码和扩展语言的表达能力，宏可以帮助开发者构建高效、简洁且可维护的代码库。

11.1.2 汇编器

GCC 系统的内部组件 cc1 负责把 C 语言程序文件编译成汇编代码，执行“cc1 main.i -o main.s”（通常不会直接使用）或“gcc -S main.i -o main.s”命令后，main.i 被编译成 ASCII

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第449页 文件编号: 422624 页面编号: 34494538449 打印人: 8685 2026-03-09 10:23:06

汇编文件 main.s，如图 11.4 所示。这些汇编代码由**汇编器**进一步处理。

```
        .file"main.c"
        .version"01.01"
gcc2_compiled.:
.section.rodata
.LC0:
        .string"%d,%d"              // scanf 和 printf 中使用的格式串
.text
        .align 4                    // 按 4 字节对齐
.globl main
        .type main,@function        // 本模块定义的函数 main
main:
        pushl %ebp                  // 将老的基地址指针压栈
        movl %esp,%ebp              // 将当前栈顶指针作为基地址指针
        pushl $buf+4                // 实参 buf+1 入栈
        pushl $buf                  // 实参 buf 入栈
        pushl $.LC0                 // 格式串指针入栈
        call scanf                  // 调用函数 scanf
        addl $12,%esp               // 栈顶指针恢复到参数压栈前的位置
        call swap                   // 调用函数 swap
        movl buf+4,%eax             // 取实参 buf[1]到寄存器
        pushl %eax                  // 实参 buf[1]入栈
        movl buf,%eax               // 取实参 buf[0]到寄存器
        pushl %eax                  // 实参 buf[0]入栈
        pushl $.LC0                 // 格式串指针入栈
        call printf                 // 调用函数 printf
        addl $12,%esp               // 栈顶指针恢复到参数压栈前的位置
        xorl %eax,%eax
        jmp .L1
        .p2align 4,,7
.L1:
        leave
        ret
.Lfe1:
        .size main,.Lfe1-main
        .comm buf,8,4               // 本模块定义的未初始化全局变量 buf
                                    // 占 8 字节，按 4 字节对齐
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第450页 文件编号: 422624 页面编号: 34504538447 打印人: 8685 2026-03-09 10:23:06


```
.ident "GCC:（GNU）egcs-2.91.66 19990314/Linux（egcs-1.1.2 release）"
```

图 11.4 汇编文件 main.s

最简单的汇编器对输入进行两遍扫描。在第一遍，汇编器扫描输入，将表示存储单元的所有标识符都存入符号表，并分配地址；在第二遍，汇编器再次扫描输入，把每个操作码翻译成机器语言中代表那个操作的位串，并把代表存储单元的每个标识符翻译成符号表中为这个标识符分配的地址。

倘若一个汇编代码文件中有对外部符号的引用，如果汇编器生成的是可重定位的目标文件，那么汇编器的工作将变得略微复杂。在 11.1.4 小节，当知道了目标文件的格式后，要实现一个汇编器并不困难。另外，一遍扫描完成汇编代码到可重定位目标代码的翻译也是完全可能的。

在 GCC 系统中，要想得到源程序被编译成的汇编代码，只要加编译选项-S 即可。例如，用

```
gcc -S main.c
```

可以得到汇编文件 main.s。使用汇编器 as

```
as -o main.o main.s
```

可以将 main.s 汇编成可重定位的目标文件 main.o。

11.1.3 连接器

由第 8 章可知，汇编器或编译器输出的机器代码称为目标模块或目标文件，它有以下两种形式：

（1）可重定位的目标文件。它包含二进制代码和数据，可以和其他可重定位目标文件组成一个可执行的目标文件。

（2）可执行的目标文件。它包含二进制代码和数据，可以直接被复制到内存并被执行。

实际上，还有另一种形式，即共享目标文件。它是一种特殊的可重定位目标文件，可以在装入程序或运行程序时，动态地把共享目标文件装入内存并将它和程序连接。

从技术角度看，一个**目标模块**是一个字节序列，而一个**目标文件**则是一个以文件形式存储在外部存储器上的目标模块。但是，本书将不加区分地使用这两个术语。

连接是一个收集、组织程序所需的不同代码和数据（它们可能在不同的目标模块中）的过程，以便程序能被装入内存并执行。连接可以在将源代码翻译成机器代码的编译阶段完成，也可以在程序装入内存并执行的装入时完成，甚至可以在程序运行时完成。

静态连接器负责将多个可重定位的目标文件组成一个可执行的目标文件（也可以组成一个可重定位的目标文件）；**动态连接器**则支持在内存中的可执行程序在执行时与共享目标文件进行动态的连接。有些系统将装入可执行程序时与共享目标文件进行的连接称为动态连接。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第451页 文件编号: 422624 页面编号: 34514538445 打印人: 8685 2026-03-09 10:23:06

如果这些目标文件是以有用的方式组织在一起的，那么它们之间就会出现一些外部引用，即一个文件中的代码引用另一文件中的存储单元。这种引用可以分为两种情况：一种是在一个文件中定义数据单元，而在另一个文件中使用它；另一种是函数入口点出现在一个文件中，而调用点出现在另一个文件中。

在连接器 ld 的上下文中，一个重定位模块 M 定义和引用的符号通常分成以下 3 类：

（1）*全局符号*：指那些在模块 M 中定义，可以被其他模块引用的符号。它包括模块 M 中定义的非 static 属性的函数和全局变量。

（2）*局部符号*：指那些在模块 M 中定义，且只能在本模块中引用的符号。它包括模块 M 中定义的有 static 属性的函数和全局变量。

（3）*外部符号*：指那些由模块 M 引用并由其他模块定义的符号。

这样，连接器主要完成以下两个任务。

（1）*符号解析*（symbol resolution）。连接器识别各个目标模块中定义和引用的符号，为每个符号引用确定它所关联的一个同名符号的定义。

（2）*重定位*。编译器和汇编器产生的代码节和数据节分别都是从零地址开始的。连接器按如下方式来重定位这两节：将每个符号定义关联到一个内存位置，然后修改所有对这些符号的引用，以使它们指向相关联的内存位置。

在理解了 11.1.4 中的目标文件的格式后，实现连接器并不困难。

11.1.4 目标文件的格式

目标文件的格式随系统的不同而不同。来自 Bell 实验室的第一个 UNIX 系统使用 a.out 格式，至今 UNIX 环境下的可执行文件仍被默认命名为 a.out。System V UNIX 的早期版本使用 COFF（common object file format）格式。Windows 平台使用 COFF 的一个变体，称为 PE（portable executable）格式。现代 UNIX 系统，如 Linux、System V UNIX 的后期版本、BSD UNIX 变体和 Sun Solaris，都使用 UNIX 的 ELF（executable and linkable format）格式。这里仅讨论 UNIX 使用的 ELF 文件格式。

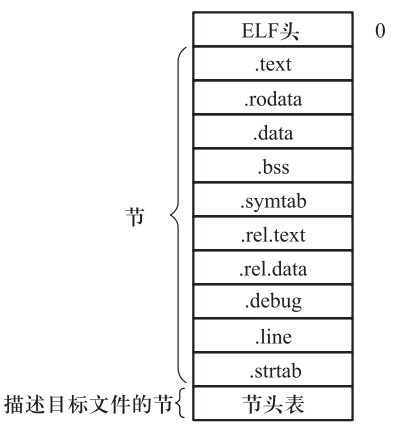


图 11.5 ELF 可重定位目标文件格式

图 11.5 所示为典型的 ELF 可重定位目标文件格式。ELF **头**（header）开始于一个 16B 的序列，它描述了字的大小和产生此文件的系统的字节次序。ELF 头的其余部分包含的信息用于连接器分析和解释目标文件，其中包括 ELF 头的大小、目标文件的类型（可重定位、可执行或共享等）、机器类型（如 IA32）、节头表（section header table）在本目标文件中的偏移、节头表

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第452页 文件编号: 422624 页面编号: 34524538443 打印人: 8685 2026-03-09 10:23:06

中条目的大小和数量。

节头表描述目标文件中各节的位置和大小。在 ELF 头和节头表之间是节本身。典型的 ELF 可重定位目标文件包含以下各节：

（1）.text：被编译程序的机器代码。

（2）.rodata：诸如 printf 语句中的格式串和 switch 语句的跳转表（见 7.4.4 小节）等只读数据。

（3）.data：已初始化的全局变量，如 swap.c 中的 bufp0。

（4）.bss：未初始化的全局变量，如 main.c 中的 buf 及 swap.c 中的 bufp1。该节在目标文件中不占实际的空间，只是一个占位符。目标文件格式区分已初始化和未初始化变量是为了提高空间的利用率：在目标文件中，未初始化变量不必占用实际外部存储器的任何空间（历史上，bss 是 better save space 的缩写）。在有些汇编语言中，已经用.comm 代替了.bss。

（5）.symtab：记录在该模块中定义和引用的函数和全局变量信息的符号表。和编译器内部的符号表不同的是，.symtab 中不包含局部变量。

【例 11.1】 表 11.1 是用 readelf 工具显示的 main.o 的符号表的后 5 个条目。没有显示的前 9 个条目是供连接器内部使用的局部符号。注意，符号表的内容也随 GCC 版本的不同而略有不同。

表 11.1 main.o 的符号表的后 5 个条目

Num	Value	Size	Type	Bind	Ot	Ndx	Name
9	0	66	FUNC	GLOBAL	0	1	main
10	4	8	OBJECT	GLOBAL	0	COM	buf
11	0	0	NOTYPE	GLOBAL	0	UND	scanf
12	0	0	NOTYPE	GLOBAL	0	UND	swap
13	0	0	NOTYPE	GLOBAL	0	UND	printf

在符号表中，Value 域表示符号的地址：对于可重定位的目标文件，它是相对于定义该对象的节的起始处的偏移；对于可执行的目标文件，它是绝对的运行地址。Size 域表示该对象的字节数。Type 域表示符号的类型，一般是数据对象（OBJECT）或函数（FUNC）。Bind 域表明符号是局部的（LOCAL）、全局的（GLOBAL）还是外部的（EXTERN）。Ndx 域指示与该符号关联的节，它的值是节头中的索引号。除了节头表中记录的节，还有 ABS、UND 和 COM 3 个特殊的**伪节**。其中，ABS 指不需要重定位的符号；UND 指被该模块引用但没有在该模块中定义的符号；COM 是未初始化的数据对象。对于 COM 符号，Value 域表示值的对齐方式，Size 域表示其最小的字节数。

在该例中，第 9 个符号 main 是位于.text 节 (Ndx=1)、偏移为 0 的 66 字节的函数；第 10 个符号 buf 是位于.bss 节、按 4 字节对齐的 8 字节对象；随后是引用的 3 个外部符号 scanf、swap 和 printf。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第453页 文件编号: 422624 页面编号: 34534538441 打印人: 8685 2026-03-09 10:23:06

（6）.rel.text: .text 节中需要修改的单元位置列表。当连接器将该目标文件和其他目标文件连接时需要这些信息，它包括任何调用外部函数或引用全局变量的指令。

（7）.rel.data: 用于被本模块引用或定义的全局变量的重定位信息。通常，任何要初始化的全局变量，若它的初值为某全局变量或外部函数的地址，则它的值需要修改。

（8）.debug: 用于调试程序的调试符号表，它包含在源程序文件中定义的局部变量和类型定义、在源程序文件中定义和引用的全局变量，以及最初的源程序文件等条目。只有用选项-g 调用 gcc 才会出现此节。

（9）.line: 源程序文件和.text 节中的机器指令之间的行号映射。该节仅在用选项-g 调用 gcc 时才会出现。

（10）.strtab: 一组有空结束符的串构成的串表。它用于保存.symtab 节和.debug 节的符号表中的名字和节头表中节的名字。

11.1.5 符号解析

连接器需要将每个符号引用正确地与来自输入的可重定位模块的符号表中的一个符号定义相关联，从而确定各个符号引用的位置。在编译时，当遇到当前源文件没有定义的符号时，它假定该符号已在其他某个模块中定义，并为该符号产生一条符号表条目（如 main.o 符号表中的 swap 符号），把它留给连接器处理。如果连接器在所有输入模块中都找不到被引用符号的定义，则报告错误消息并结束连接。

解析全局符号的引用还是有点棘手，因为同一个符号可能被多个目标模块定义。此时，连接器必须报告一个错误，或者按某种方式选择其中一个定义而放弃其余的定义。UNIX 系统中采用的方法涉及编译器、汇编器和连接器之间的合作，它会给粗心的程序员引入一些莫名其妙的错误。

在编译时，编译器向汇编器输出的全局符号区分为强和弱两种，汇编器隐式地将这些信息编码在可重定位的目标文件的符号表中。函数和已初始化的全局变量为强符号；未初始化的全局变量为弱符号。对于图 11.2 所示的程序，main、bufp0 和 swap 为强符号，buf 和 bufp1 为弱符号。UNIX 连接器使用如下规则处理多重定义的符号：

规则 1 不允许有多重的强符号定义。

规则 2 出现一个强符号定义和多个弱符号定义时，选择强符号的定义。

规则 3 出现多个弱符号定义时，选择任意一个弱符号的定义。

例如，若将图 11.2 中的 swap.c 的第 1 行“extern int buf[2];”改为“int buf[2] = 11, 21;”并存入 swap1.c，再将 main.c 的第 2 行“#if 1”改为“#if 0”并存入 main1.c，则试图编译和连接 main1.c 和 swap1.c 时，连接器将报告错误信息，因为强符号 buf 被多次定义（规则 1）。

若试图将 swap1.c 和原先的 main.c 进行编译和连接生成可执行文件时，由于 main.c 中

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第454页 文件编号: 422624 页面编号: 34544538449 打印人: 8685 2026-03-09 10:23:06

定义的 buf 为弱符号，这时连接器将隐式地选择在 swap1.c 中定义的强符号 buf，从而顺利地生成可执行文件（规则 2）。

再将 swap.c 的第 1 行中的“extern”去掉并存入 swap2.c 中，然后编译和连接 main.c 和 swap2.c，这时也能生成可执行文件（规则 3）。

但是，若将 swap2.c 中 buf 的类型由 int 改为 double 并存入 swap3.c，再编译和连接 main.c 和 swap3.c 时，虽能生成可执行文件（规则 3），但会因为多重定义的两个弱符号的类型不相容而给出警告信息。并且由于这个类型不相容，生成的目标程序执行时，buf 数组元素的值和上一步得到的目标程序执行时的值不一样。

规则 2 和规则 3 的应用可能会产生一些难以理解的不会被捕获的错误，尤其是在多重定义的符号具有不同类型时。为避免这样的问题，程序员可以用适当的编译选项（如–warn-common 选项）调用连接器，以获得解析多重全局符号定义的警告信息。

11.1.6 静态库

到目前为止，一直假设连接器读入一组可重定位的目标文件，将它们连接并输出一个可执行文件。事实上，所有的编译系统都提供一种机制，将相关的可重定位目标模块打包成一个文件，称为**静态库**。静态库可以作为连接器的输入被多次使用。当建立可执行文件时，连接器仅复制库中被应用程序引用的模块。

考虑 ANSI C，它定义了一个范围极广的标准 I/O、串操作和整型数学函数的集合，如 printf、strcpy、atoi 等。它们被编译打包在 libc.a 库中，可以供所有的 C 程序使用。ANSI C 还定义了浮点型的数学函数集，如 libm.a 库中的 sin 和 cos。现在思考编译器开发者不用静态库将这些函数提供给用户的几种不同方法。

（1）编译器认可对标准函数的调用并直接产生相关的代码。Pascal 语言使用这种方法，为程序员提供了少量的标准函数。对于 C 语言，这种方法并不可行，因为 C 语言的标准定义了大量的标准函数，这样做会显著增加编译器的复杂性。当每次增加、删除或修改标准函数时，都需要发行新的编译器版本。但是，对应用程序员来说，这种方法相当方便，因为这些标准函数总是可用的。

（2）将 C 语言的所有标准函数放在一个可重定位的目标文件中，如 libc.o，应用程序员可以将它连接并生成自己的可执行文件。这种方法的好处是标准函数的实现不再是编译器实现的一部分，而对程序员来说仍然相当方便。但是，该方法有两个严重的缺点：①系统中每个可执行文件都可能包含全部标准函数集的副本，这将极大地浪费外部存储器空间，更糟的是每个正在运行的程序在内存中都可能包含这些函数的副本，这将极大地消耗内存资源；②任何对标准函数的修改，不论是多么小的修改，都需要库开发者重新编译整个库的源文件，这种耗时的操作将使标准函数的开发和维护变得复杂。

（3）通过为每个标准函数创建一个可重定位的目标文件，并按照大家熟知的目录存储它

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第455页 文件编号: 422624 页面编号: 34554538446 打印人: 8685 2026-03-09 10:23:06

们来部分地解决这些问题。但是，这种方法要求应用程序员显式地将有关的目标模块连接到他们的可执行文件中，这一过程很容易出错并且耗时。

静态库可以用来解决上述各种方法存在的问题。可以把相关的函数分成若干源文件，分别编译，然后把生成的目标模块打包成一个静态库文件。应用程序可以通过在命令行中指定该库文件名来使用库中定义的任何函数。

在连接时，连接器将只复制被程序引用的目标模块，这就减少了可执行程序在外部存储器和内存中的大小。另外，应用程序员的编译命令只需要把用到的库文件的名称包含进来即可。事实上，gcc 会自动地将 libc.a 等库传递给连接器，不需要应用程序员显式地指明。但是，在直接用 ld 连接目标模块时，必须显式地给出被引用的所有库文件，否则连接器不能正确地完成连接。

在 UNIX 系统中，静态库以一种特殊的文件格式，即档案文件（archive），存储在外部存储器中。一个档案文件是一个带档案头的被连接的目标文件的集合，档案头描述了每个成员的目标文件的大小和位置。档案文件以.a 为扩展名表示。例如，可以用两个命令将 swap.c 编译并打包成一个自己的库 mylib.a：

```
gcc -c swap.c
ar rcs mylib.a swap.o
```

其中，编译选项-c 表示生成目标文件，不进行连接。

这时可以编译 main.c，并将它和 mylib.a 连接，建立可执行程序 swap1：

```
gcc -static -o swap1 main.c /usr/lib/libc.a mylib.a
```

其中，-static 参数要求连接器建立一个完全连接的可执行的目标文件，它可以直接被装入内存运行，不需要任何进一步的连接。

图 11.6 总结了带有-static 参数的连接器的活动。当连接器运行时，它确定由 swap.o 定义的 swap 符号被 main.o 引用，因此它将 swap.o 复制到可执行文件 swap1 中。而 mylib.a 中的其他成员模块（如果有），由于其定义的符号没有被引用，则不被连接器复制到可执行文件中。连接器同样复制来自 libc.a 的 printf.o 模块、scanf.o 模块和一些来自 C 程序运行时系统的其他模块。

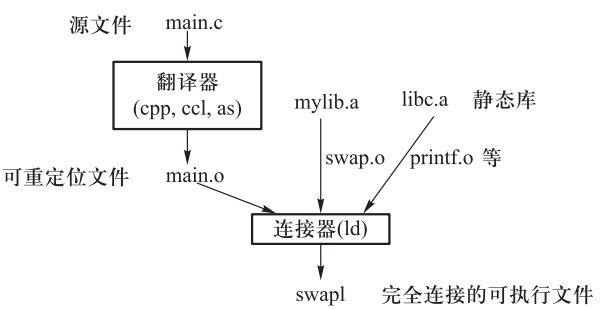


图 11.6 和静态库连接

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第456页 文件编号: 422624 页面编号: 34564538444 打印人: 8685 2026-03-09 10:23:06

静态库一方面是一种有用的基本工具；另一方面它们也让程序员感到混乱。因为 UNIX 连接器使用静态库来解析外部引用，在符号解析阶段，连接器按照可重定位的目标文件和档案文件在 gcc 或 ld 命令行中出现的次序从左到右扫描它们。命令行中档案文件和目标文件的次序相当重要。如果定义一个符号的档案文件在命令行中出现在引用该符号的目标文件之前，则这种引用可能不被解析，从而导致连接失败。

使用档案文件的一般规则是将它们放在命令行的最后。如果不同档案文件的成员是相互独立的，则由于没有一个档案文件的成员引用其他档案文件的成员定义的符号，这些档案文件可以按任何次序放在命令行的最后。如果这些档案文件之间相互不独立，则它们必须按一定的次序出现在命令行中，使得对每个被一个档案文件中的成员引用的外部符号 s，至少有定义 s 的一个档案文件在命令行中出现在含有对 s 的引用的档案文件之后。

11.1.7 可执行的目标文件及装入

图 11.7 概括了典型 ELF 可执行的目标文件中的信息种类。它与可重定位的目标文件的格式类似。ELF 头描述文件的整体格式，它包含程序的入口点，即当程序运行时要执行的第一条指令的地址。对于.text、.rodata 和.data 节，除了已经被重定位成最后运行时的内存地址，它们还与可重定位的目标文件中对应的节类似。.init 节定义一个称为 _init 的小函数，它由程序的初始化代码调用。由于可执行的目标文件已被完全连接，因此它不需要与可重定位目标文件的.rel.text、.rel.data 类似的可重定位节。

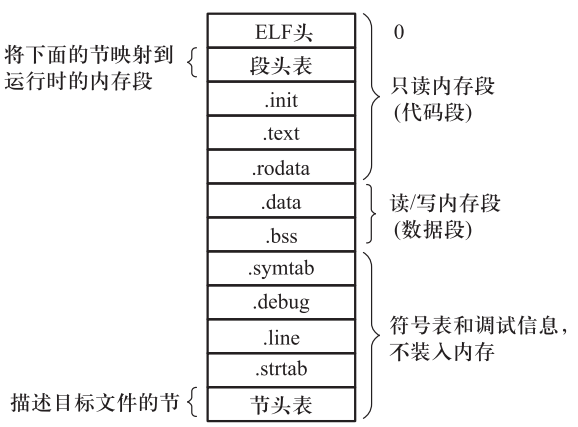


图 11.7 典型的 ELF 可执行目标文件

ELF 可执行的目标文件被设计成易于装入内存，可执行目标文件中相邻的块被映射到相邻的内存段（segment）中。这种映射关系在段头表中描述。

为了运行一个可执行文件 swap，可以在 UNIX shell 的命令行中输入它的名字：

```
./swap
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第457页 文件编号: 422624 页面编号: 34574538442 打印人: 8685 2026-03-09 10:23:06

由于 swap 不是内部的 shell 命令，shell 假定 swap 是一个可执行的目标文件，它通过调用称为**装载器**的驻留在内存中的操作系统代码来运行 swap。任何 UNIX 程序都可以通过调用 execve 函数来调用装载器。装载器将可执行的目标文件从外部存储器复制到内存，然后通过跳转到程序的入口点来运行程序。复制程序到内存中并运行程序的过程被称为**装入**。

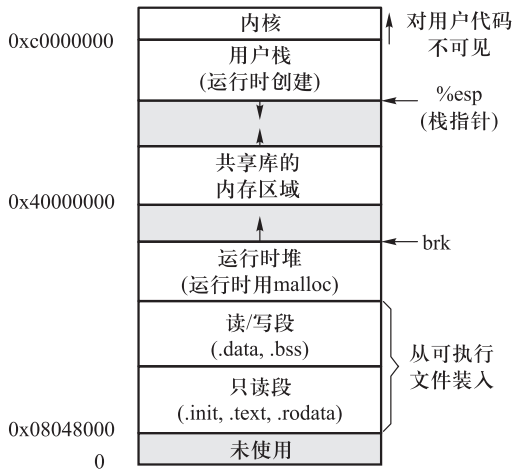


图 11.8 32 位 Linux 运行时的内存映像

每个 UNIX 程序都有一个和图 11.8 类似的运行时内存映像。在 Linux 系统中，代码段一般总是从 0x08048000 地址开始。数据段从下一个按 4KB 对齐的地址开始。运行时的堆跟在可读写段后，从第一个按 4KB 对齐的地址开始，并通过调用 malloc() 库函数向上增长。从 0x40000000 地址处开始的段被保留用于共享库。用户栈总是从 0xbfffffff 地址开始并向较低的内存地址增长。位于栈上，从 0xc0000000 地址开始的段被保留用于操作系统驻留内存部分的代码和数据，即操作系统内核。

当装载器运行时，它创建如图 11.8 所示的内存映像。在可执行目标文件的段头表的指导下，装载器将它的块复制到内存中的代码段和数据段。接着，装载器跳转到程序的入口点，它就是 _start 符号的地址。_start 地址处的启动代码定义在目标文件 crt1.o 中，它对于所有的 C 语言程序都一样。_start 完成用户程序运行前的准备，它调用.text 和.init 节中的初始化例程等后，调用应用程序的 main 函数，开始执行用户的 C 代码。

注意，这里描述的装入过程从概念上说是正确的，只有在理解了进程、虚拟内存和内存分页等概念后，才能真正了解装入过程是如何工作的。

11.1.8 动态连接

与所有软件一样，静态库需要周期性地维护和更新。如果应用程序员想要使用库的最新版本，那么他们必须设法知道库已经改变，然后显式地将他们的程序和被更新的库重新连接。另一个问题是，几乎每个 C 语言程序都使用像 printf 和 scanf 这样的标准 I/O 函数。在运行时，这些函数的代码被复制在每个正在运行的进程的 text 段中。在一个运行有 50~100 个进程的典型系统上，这会显著地浪费内存系统资源。

共享库是弥补静态库的缺点所进行的一场变革。共享库，也称共享目标文件，它在运行时可以装到任意的内存位置并和内存中的程序连接。这一过程称为动态连接，它由动态连接器执行。在 UNIX 系统上，共享库一般以.so 为扩展名。微软操作系统称共享库为动态连接

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第458页 文件编号: 422624 页面编号: 34584538440 打印人: 8685 2026-03-09 10:23:06

库，一般以.dll 为扩展名。

共享库以两种不同的方式被共享：一种是在任何给定的文件系统中，对每个库，正好只存在一个.so 文件，该.so 文件中的代码和数据被所有引用该库的可执行目标文件所共享，这与静态库的内容被复制和嵌入到引用它的可执行目标文件中的方式相反；另一种是在运行时，共享库的.text 节在内存中的一个副本可以被正在运行的不同进程共享。

图 11.9 总结了图 11.2 所示程序的动态连接过程，这里首先将源文件 swap.c 编译、连接成自己的共享库 mylib.so：

```
gcc -shared -fPIC -o mylib.so swap.c
```

其中，-fPIC 选项指示编译器产生位置无关的代码（指无须连接器的修改而能装入到内存任意地址执行的代码）；-shared 选项指示连接器创建一个共享目标文件。

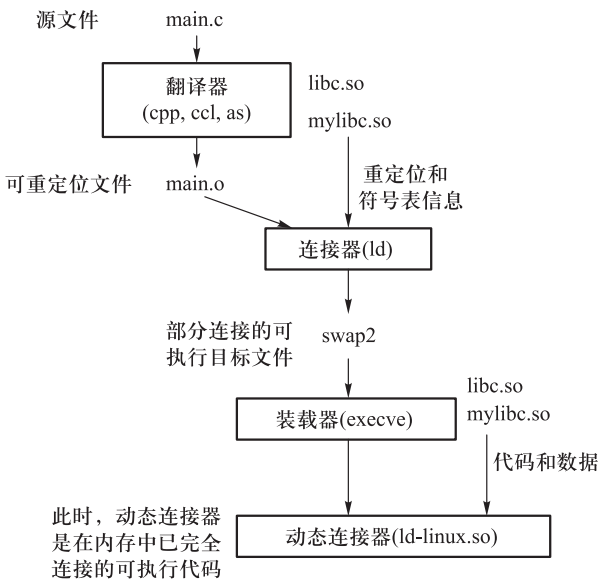


图 11.9 和动态库连接

一旦创建了 mylib.so 共享库，就可以将它和 main.c 连接成可执行程序 swap2：

```
gcc -o swap2 main.c ./mylib.so
```

动态连接的基本思想是在可执行目标文件被创建时静态地做一部分连接准备工作，然后在程序被装入时动态地完成连接过程。需要注意的是，mylib.so 中的代码节和数据节并没有被复制到可执行目标文件 swap2 中。连接器只是复制一些重定位信息和符号表信息，它们用于在运行时解析对 mylib.so 中代码和数据的引用。

装载器装入和运行可执行目标文件 swap2 时使用的是 11.1.7 讨论的技术，装入已被部分地连接的可执行目标文件 swap2。swap2 包含有.interp 节，该节包含动态连接器的路径名，如

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第459页 文件编号: 422624 页面编号: 34594538448 打印人: 8685 2026-03-09 10:23:06

Linux 系统下的 ld-linux.so，它本身是个共享对象。装载器通常装入和运行动态连接器，而不是将控制直接传给应用程序。

动态连接器接着完成以下连接任务。

（1）把 libc.so 的文本和数据装入内存并进行重定位，在 IA32/Linux 系统上，共享库被装载在起始地址为 0x40000000 的地方（见图 11.8）。

（2）把 mylib.so 的文本和数据装入内存并进行重定位。

（3）重定位目标文件 swap2 中任何对 libc.so 或 mylib.so 定义的符号的引用。

动态连接器将控制传递给应用程序。此后，共享库的位置被确定，并在该程序执行期间不再改变。

上面介绍了共享库的一种装入方式，即在应用程序被装入时，动态连接器装入和连接共享库。此外，还有另一种方式，即应用程序在运行过程中通过显式的函数调用来请求动态连接器装入和连接共享库。两种方式的装入和连接过程本质上相同，这里不再介绍。

11.1.9 处理目标文件的一些工具

在 UNIX 系统中，有很多工具可以用来帮助用户理解和处理目标文件。尤其是 GNU 的 binutils 包特别有用，可以运行在每种 UNIX 平台上。下面是一些工具的名称和功能简介：

- ar：创建静态库，插入、删除、罗列和提取成员。
- strings：列出包含在目标文件中的所有可打印串。
- strip：从一个目标文件中删除符号表信息。
- nm：列出一个目标文件的符号表中定义的符号。
- size：列出目标文件中各段的名字和大小。
- readelf：显示目标文件的完整结构，包括编码在 ELF 头中的所有信息。它包括 size 和 nm 的功能。
- objdump：所有的二进制工具之母，可以显示目标文件中的所有信息。其最有用的功能是反汇编.text 节中的二进制指令。
- ldd：列出可执行目标文件在运行时需要的共享库。ldd 程序在 UNIX 系统中用来处理共享库。

11.2 基于虚拟机的语言运行时：以 Java 和 JVM 为例

随着计算机技术的发展，编程语言不断向更高层次的抽象演进，以提升开发效率和代码的可移植性。然而，高级语言的多样性和跨平台运行的需求带来了新的挑战：如何在不同硬

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第460页 文件编号: 422624 页面编号: 34604538446 打印人: 8685 2026-03-09 10:23:06

件和操作系统上高效、可靠地执行程序，同时支持现代化的运行时特性？为了解决这些问题，基于虚拟机的语言运行时系统被发明出来，成为现代编程语言设计的重要突破之一。

基于虚拟机的运行时系统通过将源代码编译成与底层平台无关的中间代码，使同一份代码能够在不同平台上运行。代表性语言包括 Java（1995 年发布）和 C#（2000 年发布），它们分别依赖 Java 虚拟机（JVM）和公共语言运行时（CLR）来实现跨平台执行。这些语言将源代码编译为字节码，由虚拟机进行解释执行或即时编译（JIT）成机器代码。

但是，虚拟机技术并非 Java 首创。在 Java 出现之前，UCSD Pascal 系统就已采用虚拟机思想，并通过 P-Code 机器在商业产品中实现了类似的技术。

此外，诸如 Python（1991 年发布，通过 PyPy 或其他实现）、Ruby（1995 年发布，如 JRuby）、Kotlin（2011 年发布）和 Scala（2004 年发布）等语言也依赖虚拟机技术来支持其跨平台执行。

虚拟机是一种通过软件模拟的抽象机，按执行模型分为基于栈的虚拟机和基于寄存器的虚拟机两种。基于栈的虚拟机以操作数栈为核心，指令集设计简单，跨平台性强，但由于频繁的栈操作，性能较低，适用于嵌入式和解释执行场景；而基于寄存器的虚拟机模拟物理 CPU 寄存器，指令效率更高，但设计复杂、实现难度较大、跨平台支持较为困难，适合 JIT 等高性能需求。Java 和 Python 采用基于栈的模型，Android 早期的 Dalvik 虚拟机则基于寄存器模型。

JavaScript（1995 年发布）并不像 Java 那样依赖传统虚拟机，它通过现代浏览器中的 JavaScript 引擎（如 2008 年由 Google 发布的 V8）实现类似功能。V8 使用 JIT 技术，在运行时将 JavaScript 代码转换为机器码，提升性能并支持跨平台执行，使其在浏览器和 Node.js 中具备强大的执行与优化能力。

本节以 Java 为例介绍基于虚拟机的语言运行时系统，了解虚拟机技术的主要构成和原理。

11.2.1 Java 虚拟机语言简介

JVM 是 Java 跨平台特性的关键。Java 源程序被编译成与平台无关的中间代码——Java 虚拟机语言（JVML）程序，存储在扩展名为“.class”的字节码文件中。每个.class 文件是一个 8 位字节流，包含 Java 类或接口的信息，包括符号表、方法的字节码序列以及其他辅助信息。JVM 通过自身的指令系统和内存管理机制，屏蔽底层硬件与操作系统差异，使 JVML 程序只需与虚拟机交互即可运行。理解 JVM 的工作机制需要熟悉.class 文件的关键组成部分和相关概念：

（1）常量池（constant pool）。常量池包含了在该.class 文件结构及其子结构中引用的各种类型（字符串、浮点等）的常量。常量池的功能类似于传统编程语言中的符号表，但是它比通常的符号表包含更多的信息。

（2）类成员信息。一个 Java 类的成员信息放在两个长度可变的表中：域信息表和方法信息表。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第461页 文件编号: 422624 页面编号: 34614538444 打印人: 8685 2026-03-09 10:23:06

（3）JVML 指令序列。一条 JVML 指令由 1 字节的操作码以及若干供该操作使用的操作数构成。Java 虚拟机上同样有运行数据区，每个线程都有一个运行栈，该栈保存局部变量和计算的中间结果，并参与方法的调用和返回。所有线程共享的堆用来动态分配对象。JVML 指令描述的就是在这样的抽象机上进行的操作。图 11.10 展示了一个 JVML 程序的例子。

```
Java 源程序中的方法：
int calculate (int i){
    int j=2;
    return ((i+j) * (j-1));
}
对应的字节码程序：
int calculate (int i)
iconst_2                // 在 Java 栈中压入常数 2
istore_2                // 将栈顶常数存放到局部变量 2(局部变量 j)中
iload_1                 // 将局部变量 1(参数 i)压入栈顶
iload_2                 // 将局部变量 2(局部变量 j)压入栈顶
iadd                    // 从栈中弹出两个整数并相加，结果压栈
iload_2                 // 将局部变量 2(局部变量 j)压入栈顶
iconst_1                // 在 Java 栈中压入常数 1
isub                    // 从栈中弹出两个整数并相减，结果压栈
imul                    // 从栈中弹出两个整数并相乘，结果压栈
ireturn                 // 将栈顶元素返回
```

图 11.10 JVML 程序示例

11.2.2 Java 虚拟机

Java 虚拟机一般由以下几部分构成：类装载器（字节码验证器）、解释器或/和编译器，无用单元收集器（garbage collector）和线程控制模块在内的运行时支持系统，以及一些标准类和应用接口的.class 文件库。其组成情况如图 11.11所示。

首先运行 Java 编译器把 Java 应用程序编译成字节码程序。字节码程序可以在网络上传送，在另一台机器的 Java 虚拟机上运行。Java 虚拟机执行字节码的过程可以分为 3 步：代码的装入、代码的验证和代码的执行。

代码的装入由类装载器完成，类装载器负责装入程序运行时需要的所有代码，其中包括程序代码中用到的所有类。随后，被装入的代码由字节码验证器进行安全性检查，以确保代码不违反 Java 的安全性规则，字节码验证器还可以发现操作数栈溢出、非法数据类型转换等多种错误，通过验证之后，代码就可以提交运行了。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第462页 文件编号: 422624 页面编号: 34624538442 打印人: 8685 2026-03-09 10:23:06

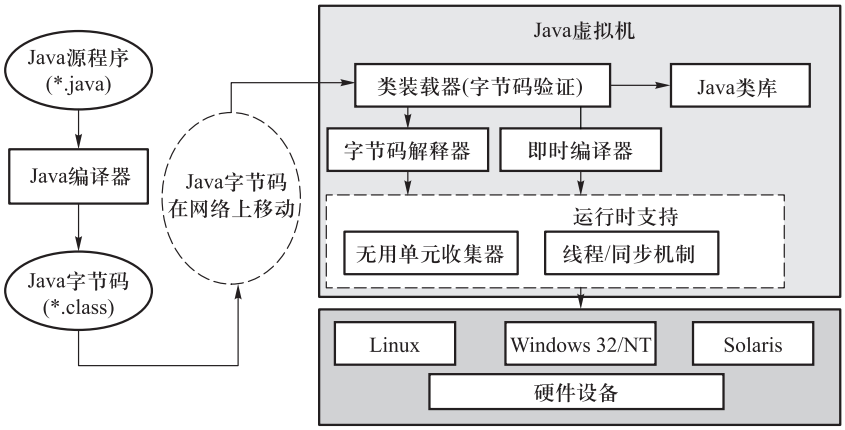


图 11.11 Java 虚拟机组成及运行环境

Java 字节码的运行有两种方式：解释执行方式和即时编译（just-in-time compilation）方式。解释执行的缺点是太过简单而且速度很慢，早期的 SUN JDK 解释器速度只有类似的 C++ 代码的 1/30 ～ $\frac{1}{5}$。即时编译方式是由即时编译器先将字节码编译成本地机器代码之后再执行。即时编译的方法能够生成质量较高、执行速度较快的代码，但需要花费额外的编译时间。需求驱动是即时编译的另一个特点，一个方法直到被调用时才将字节码翻译成机器代码，当一个方法被调用两次以上时，机器代码的执行效率便足以补偿编译耗费的时间。

Java 虚拟机还需要给字节码的执行提供其他运行时的支持。无用单元收集器用来管理所有线程共享的运行时的数据空间。在 Java 虚拟机的逻辑组件中，运行时数据空间包括 Java 栈、堆、方法区（method area）、常量池。其中，Java 栈是每个 Java 虚拟机线程私有的，与线程同时创建并同时结束；而堆、方法区、常量池则是所有线程共享的。堆是从中分配所有类实例和数组的运行时的数据区。Java 对象从不被显式地回收，无用单元收集器自动地将程序中不再用到的单元回收。方法区类似于传统语言的代码存储区，如 UNIX 进程中的.text 段，它存储每个类的公用数据和方法代码。方法区逻辑上是堆的一部分，也由无用单元收集器来管理，但是一些简单的实现可以选择不回收它。常量池是各个.class 文件中常量池的运行时的表示，其内容包括从编译时已知的数值和文字到必须在运行时解析的方法和域引用。常量池是方法区的一部分。11.3 节将介绍无用单元收集技术。

Java 语言支持多线程，对于多线程的应用而言，线程调度和同步支持是多线程协同工作正确性的重要保证。在典型的商务应用中，同步操作占了相当大的部分，因此高效地实现同步也是高性能虚拟机的重要因素。对于线程的管理，虚拟机可以选择由自己来对程序中的多个线程进行调度，也可以采用本地绑定（native-binding）的方法，将每个 Java 线程都映射为实际运行的操作系统上的线程，使用操作系统的调度器来实现对它们的高效支持。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第463页 文件编号: 422624 页面编号: 34634538440 打印人: 8685 2026-03-09 10:23:06

11.2.3 即时编译器

由于 JVML 平台无关的特点，JVML 程序的装载、连接、编译过程与传统编程语言的编译过程相比更具有动态性，但是没有本质的区别，因此仅介绍即时编译器。

当一个类的某个方法第一次被调用时，虚拟机才激活即时编译器并将它编译成机器代码，编译器被称为“即时”也源于此。即时编译器以一个方法为单位进行编译，能够生成较高质量的代码，它生成的代码的执行速度可以达到解释执行的 10 倍。

即时编译器的出现使 Java 程序的执行效率得到很大提高，但是执行过程不得不等待编译结束，因此使得执行时间变长。在传统的静态编译中，编译时间可以忽略不计，因为经过一次编译得到的可执行文件，可以被多次执行。对于即时编译器来说却并非如此，即时编译器在运行时编译字节码，编译时间是运行时间的一部分。为了缓解执行效率和编译开销之间的矛盾，很多虚拟机都会使用快速解释器和优化编译器的组合或者是简单编译器和复杂编译器的组合。下面用一个具体的实现来说明即时编译器是如何动态地、按需地编译一个方法的。

Intel 于 2000 年发布的开放式运行时平台（open runtime platform, ORP）是一个开源 JVM 项目，旨在构建模块化、灵活且高性能的运行时平台，以支持 Java 应用程序的执行。2005 年，Intel 将 ORP 捐赠给 Apache 基金会，并命名为 Apache Harmony。它是一个开源的 Java SE 实现项目，致力于提供完整、兼容的 Java 运行时环境和开发工具链。尽管 Apache Harmony 在推动虚拟机技术（包括垃圾回收、即时编译器优化和线程调度等）方面起到了积极作用，但由于与 Oracle 的 Java 标准兼容性问题及其他高性能 JVM（如 HotSpot）的主导地位，该项目于 2011 年停止开发。然而，ORP 及其衍生项目在模块化设计、优化技术以及开源虚拟机研究中留下了深影响。它的即时编译器具有动态特性，不仅在方法首次调用时进行编译，还能够根据代码的特点动态选择最合适的编译策略。

图 11.12 展示了 ORP 即时编译方法的过程。当一个类的方法表刚创建时，方法的代码指针指向一段 compile-me 代码，负责编译方法的字节码。当方法首次被调用时，执行即时编译器来编译该方法的字节码。编译完成后，compile-me 代码段将把方法表中的代码指针更新为指向编译得到的本地代码，并且执行第一次调用。当方法再次被调用时，执行的就是本地代码了。

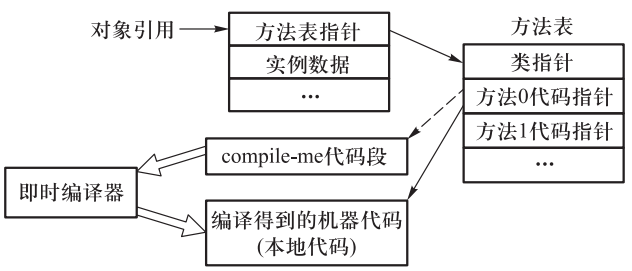


图 11.12 ORP 即时编译方法的过程

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第464页 文件编号: 422624 页面编号: 34644538448 打印人: 8685 2026-03-09 10:23:06

ORP 的编译器还实现了动态重编译机制，其关键在于对不同代码自适应使用不同编译策略：对于执行频率低的“冷”代码，采用快速粗糙的编译；而对执行频率高的“热”代码，则进行更细致的编译。该机制能够根据运行时信息，判断代码是否为热点而动态调整编译策略。

图 11.13 展示了 ORP 的编译结构。这个编译结构有 3 个重要的组成部分：快速生成代码的编译器、优化编译器、统计信息。

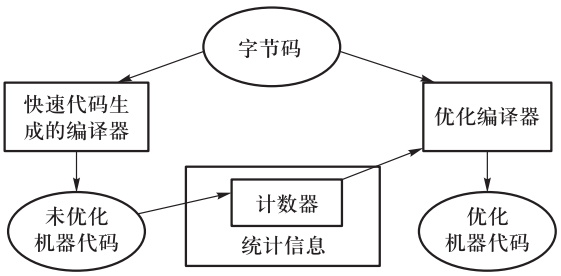


图 11.13 ORP 编译结构

快速生成代码的编译器在 ORP 中通常也被称为 O1 编译器，所有的方法在第一次被调用时都用这个编译器编译成机器代码。ORP 快速编译器不但能达到比较理想的编译速度，而且能够对机器代码进行轻量级的优化，生成的机器代码的执行速度要比解释执行快得多。O1 编译器的目标是快速地生成机器代码，并且维持一个合理的代码质量，它能够在对字节码进行两次遍历后生成机器代码。

O1 编译器在机器代码中插入一些统计代码，用来收集统计信息，这些信息记录了方法或一段循环代码被调用的次数。这些统计信息包含进行重编译的触发条件。有了统计信息，就能够判断某段代码是否为“热”代码，当代码的调用次数达到某个阈值时，它将被优化编译器重新编译。优化编译器对方法重新编译时还将利用 O1 编译器收集的这些统计信息。

优化编译器也称为 O3 编译器，它对代码进行更为细致的优化，生成质量较好的目标代码。之后，当方法再次被调用时，就会执行优化版本的机器代码。O3 编译器为了生成高质量的目标代码，需要耗费更多时间在优化上。它采用传统的编译方法，为字节码建立一种中间表示，基于中间表示进行全局优化。

ORP 就是这样采用简单编译器和复杂编译器的组合，构造了一种平衡编译时间和代码质量的动态编译机制。

*11.3 无用单元收集

从理论上讲，无用单元（garbage）就是那些在程序的继续运行过程中不会再被使用的数据单元。被无用单元占据的内存空间应该被回收，以便把它们分配给需要空间的变量使用，这

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第465页 文件编号: 422624 页面编号: 34654538445 打印人: 8685 2026-03-09 10:23:06

样一个处理过程就称为**无用单元收集**（garbage collection），俗称**垃圾收集**。该过程不需要程序员的干预，它可以自动地执行对内存的这种管理。但它可能需要来自编译器、操作系统和硬件方面的支持，并且由运行时系统来决定何时和怎样执行无用单元收集。

无用单元收集器（以下简称收集器）需要根据数据的活跃性来判断哪些是无用单元。由于并非总能判断出一个数据记录的值以后是否还需要，收集器所使用的活跃性分析采用稳妥的策略。在该策略下，活跃性通过**根集**（roots set）以及从根集开始的**可达性**来定义。按照这个策略所得到的可达记录并非都是真正活跃的，但是它们都被保留；另外，编译器被要求努力最小化可达记录中实际上并不活跃的记录的数目。这样，通常所指的无用单元是那些不可能从程序变量经指针链到达的堆分配记录。

函数式语言一般都用这种技术来回收内存空间。与 C、C++ 语言不同，命令式语言 Java 的对象和数组这样的数据都分配在堆上，在栈中有相应的指针指向它们，并且语言不向使用者提供释放空间的函数。这些指针在栈中被释放后，堆中相应的记录很可能不可达，因此需要依靠收集器来回收它们。

本节简要介绍一些主要的无用单元收集方法，并且描述编译器和收集器之间的一些相互影响，包括编译器提供给收集器的支持和收集器提供给编译器的接口。

11.3.1 标记和清扫

标记和清扫收集方法首先标记堆上所有可达记录，然后回收未被标记的记录。

在进行无用单元收集时，全局可见的变量都被看作是活跃的，任何活跃着的过程的任何一个活动记录中的局部变量也都被看作是活跃的。这样，根集就包含了全局变量、活动记录栈中的局部变量和被活跃着的过程使用的寄存器。堆上活跃记录的集合也就是从根集开始的任何一条指针路径上的记录的集合，它们可被看成一个有向图，其中记录是结点，指针是边，程序变量是根，因此任何图遍历算法（如深度优先算法）都可用于在这个图上标记所有的可达记录，这就是**标记**阶段。

任何未被标记的记录都是无用单元，应该被回收。回收过程称为**清扫**。从堆的首地址开始，逐个记录地考察整个堆，寻找未被标记的记录，把它们链成一个空闲链表。这一阶段同时清除被标记的记录的标记，以备下一轮收集。当空闲链表中的空间不足以支持程序的存储分配请求时，就启动下一轮收集。

如果用户程序需要很多不同大小的记录，一个简单的空闲链表对于分配函数来说可能显得效率不高。因为当分配一个大小为 n 字节的记录时，它可能需要沿着这个空闲链表寻找，直至找到一个大小合适的空闲块。通过建立有若干空闲链表的一个数组来解决这个问题，如 freelist$[i]$ 就是所有大小为 2^i 的空闲块的链表。这样，如果程序需要分配一个大小为 $n(2^{k-1} < n \leqslant 2^k - 1)$ 的记录，就从 freelist$[k]$ 中取一块。如果该空闲链表为空，就从 freelist$[k+1]$ 中取一块，将其中一半分配给用户程序，剩余的一半链回到 freelist$[k]$ 链表中。如果空闲块大小小于

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第466页 文件编号: 422624 页面编号: 34664538443 打印人: 8685 2026-03-09 10:23:06

2^{k+i} 的所有链表均为空，而 freelist$[k+i]$ 链表不为空，就从 freelist$[k+i]$ 链表中取一块，将其中的 2^k 大小分配给用户程序，而将剩余部分分割成若干块，分别插入到 freelist$[k]$~freelist$[k+i-1]$ 链表中。如果这样的操作失败，就调用收集器来补充空闲链表。

传统的标记和清扫方法通常有以下两大问题：

（1）**碎片**（fragmentation）问题。它可以分为外部碎片和内部碎片两方面。外部碎片是指当要分配一个 n 字节大小的记录时，发现有很多小于 n 字节的空闲块存在，但就是没有合适的空闲块可分配给这个记录。内部碎片是指程序使用一个过大的记录而没有拆分它，导致没有被使用的内存处于该记录中。其后果是空闲块和活跃记录交织在一起，使得对大记录的分配非常困难。通过维护上面提到的一组空闲链表并合并相邻空闲块可以缓解这个问题，但不能完全解决问题。

（2）**引用局部性**（locality of reference）问题。既然记录都不会被移动，那么活跃记录在一次收集以后仍然在原位置，和空闲块相交织。然后新记录使用这些空闲块，其结果是不同年代的记录交织在一起。这给引用局部性带来了消极的影响，因为这种交织有可能使当前要使用的各个活跃记录被分散到很多的虚拟内存页中，这些页在内存中可能被频繁地换进换出。在有虚存或缓存的计算机系统中，良好的引用局部性非常重要。

引用局部性也是一类数据局部性，但是它的改进主要依赖于收集算法的改进。

11.3.2 引用计数

标记和清扫收集方法通过找出堆上所有可达记录来确定无用单元。也可以通过记住有多少指针指向每个记录来直接完成，这称为记录的**引用计数**，记录的引用计数存在该记录中。

在使用这项技术的系统中，编译器需要在每个出现指针存储的地方生成额外的指令，以调整一些引用计数器的值。例如，记录 p 的引用存进 $x.f$ 中，那么 p 的引用计数值会加 1，而 $x.f$ 原来指向的记录的引用计数值会减 1。当一个记录的引用计数值为 0 时，可以把该记录加入空闲链表，并且一一检查被回收记录本身的指针域，它们所指向的记录的引用计数值也都要减 1。

引用计数算法看起来非常简单，具有吸引力，但是除了碎片问题和引用局部性问题，它还有两大问题：第一，它并不总是有效；第二，很难提高效率。

（1）引用计数的技术对于循环的数据结构会失效。如果一组记录中的指针形成了一个循环，那么即使从根集也已经不可能到达这些记录了，这些记录的引用计数也永远不可能减到零。事实上，这种循环在一般的程序中经常会产生。

（2）引用计数的效率问题是它的代价。因为每当执行指针存储时，都需要执行额外的指令来调整一些引用计数器的值。

上述问题使得引用计数技术近年来已失去了吸引力。对于大部分高性能的通用系统来说，

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第467页 文件编号: 422624 页面编号: 34674538441 打印人: 8685 2026-03-09 10:23:06

引用计数收集已经被跟踪型收集代替，这种收集器是在遍历可达记录图时，将活跃记录和无用单元区分开来。11.3.1 小节的标记和清扫收集方法就是一种跟踪型收集方法。

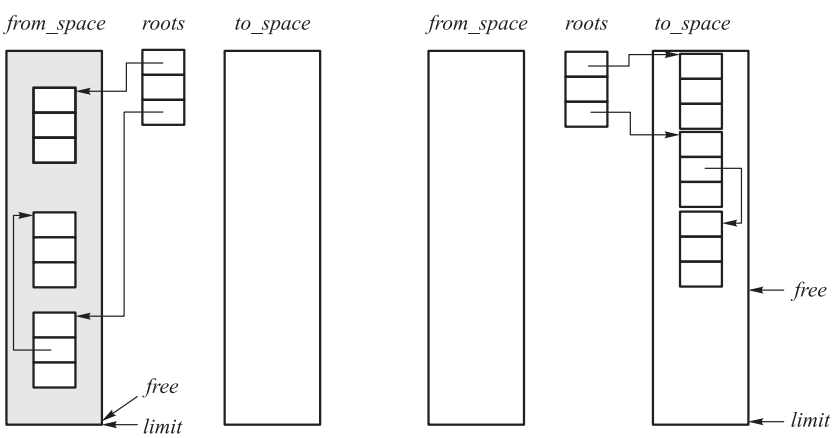
但是引用计数方法本身还是有很多优点。第一，它回收迅速。第二，即使大部分堆空间都被占据的时候，它的性能仍然不受影响，很多其他的收集器这时会需要更多的空间用于交换以提高效率。第三，它可以被直接做成完全渐增的和实时的。将来也许还会发现这种方法的其他用途，比如在混合型的收集器中使用，或者通过特殊的硬件可以提高它的性能。但是，一般来说，不会将引用计数方法作为传统的单处理器上一种主要的无用单元收集技术。

11.3.3 复制收集

复制收集方法与 11.3.1 小节中的标记和清扫收集方法一样，它也遍历可达记录所组成的有向图，只不过它在遍历的同时进行清扫，并且这种清扫主要是**复制**活跃记录。

通常，这种方法将整个堆空间分成大小相等的两块，每块都是连续的区域，分别称为 from_space 和 to_space。在程序运行时，只有 from_space 是可用的。当运行程序要求内存时，就在 from_space 中向上线性地分配内存。当运行程序要求的内存分配超过了 from_space 中空闲区域的大小时，运行程序被暂停，复制收集器被激活，用来回收空间。

复制收集器根据某种遍历算法将所有的活跃记录遍历一次，同时把它们从 from_space 复制到 to_space，并且在 to_space 中，这些记录被紧缩到一边。一旦复制完成，to_space 和 from_space 的角色相互交换，然后程序继续运行。图 11.14 给出了复制收集的一个示例。



(a) 收集前　　(b) 收集后

图 11.14　复制收集示例

对这种方法做稍微深入一点的讨论。该方法最基本的操作是指针的**转移**（forwarding）操作，就是把一个指向 from_space 的指针 p，修改成指向 to_space 中合适的位置。转移指针

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第468页 文件编号: 422624 页面编号: 34684538449 打印人: 8685 2026-03-09 10:23:06

的算法如图 11.15 所示。

```
Forward(p);
    if (p 指向 from_space ) {
        if (p->f1指向 to_space )return p->f1;
        else {
            for(p 指向记录的每个域 fi) free->fi=p->fi;
            p->f1=free;
            free=free+p 指向记录的大小;
            return p->f1;
        }
    } else return p;
```

图 11.15 转移指针的算法

以下 3 种情况需要分别对待：

（1）如果指针 p 指向一个已经被复制到 to_space 的 from_space 记录，那么该 from_space 记录的第一个域，也就是 $p->f_1$ 是一个特殊的转移指针，它指示该 from_space 记录复制在什么地方。因此，现在只需要将它返回即可。

（2）如果指针 p 指向的 from_space 记录还没有被复制，那么将它复制到 to_space 中 free 所指的位置，并把这个转移指针赋给 $p->f_1$，也就是让 $p->f_1$ 指向该记录在 to_space 中的位置。

（3）如果 p 不是指针，或者如果 p 指向 from_space 之外的区域（如指向收集区域以外的地方，或者指向 to_space），那么对 p 的转移操作将什么也不做。

从理论上说，只要有足够的内存，复制收集器可以得到很高的效率。另外，它还可以将活跃数据紧缩在一起，碎片情况消失，引用局部性得到改善。当然，在实际使用中，这些好处会受到很多限制。首先，这个算法需要的空间是实际需要空间的 2 倍；其次，复制大记录的代价太大。此外，若遍历采用宽度优先搜索，则引用局部性的改善不充分。但总体来说，复制收集方法前景广阔，它被广泛用作其他无用单元收集算法的基础，如分代算法、渐增式算法。

11.3.4 分代收集

分代收集的原理基于对这样一个现象的观察：在很多程序运行的过程中，新建记录很可能很快失效，不会出现对它的复制；反过来，如果一个记录在几次收集后还可到达，那么很可能还会活跃到许多次收集后。因此，收集器应该将精力集中到“年轻”的数据上，因为这里有相对高的无用单元比率。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第469页 文件编号: 422624 页面编号: 34694538447 打印人: 8685 2026-03-09 10:23:06

堆将被分成**代**（generation），最“年轻”的记录在 G_0 代，G_i（$i>0$）代中的记录比 G_{i-1} 代中的任何记录都要老。越“年轻”的代越被频繁地收集。

在对 G_0 代进行收集时，就是从根集开始，使用复制方法或者使用标记和清扫方法。但是，这时的根集不仅是程序变量，还包括 $G_1, G_2, \cdots$ 中指向 G_0 的指针。如果这样的指针太多，那么处理根的过程将比遍历 G_0 的可达记录的时间还要长。幸好，老记录指向年轻得多的记录的情况极少出现。通常是较新的记录指向老记录。

为了避免确定 G_0 的根集时在 $G_1, G_2, \cdots$ 中查找，需要编译器提供一些支持，让编译过的程序运行时能**记住**哪些地方有老记录到新记录的指针。这样的方法有以下几种：

（1）**记忆集合**。编译器产生指令，在每个执行形式为 b−>f=a 的修改存储单元后，将 b 放到被修改记录的一个向量（记忆集合）中。然后在收集时，收集器扫描记忆集合，寻找指向 G_0 的老指针 b。

（2）**卡标记**。该方法将存储区域划分为大小为 2^k 的逻辑“卡”，一个记录可以占据一块卡的一部分，或者从一块卡的中间开始，继续到下一块。每当地址 c 的内容被更新时，包含这个地址的卡就被标记。这里用 1 字节数组作标记，字节索引可以通过将地址 c 右移 k 位获得。

（3）**页标记**。该方法类似于卡标记方法，如果 2^k 正好是页的大小，那么可以用操作系统的虚拟内存管理机制来标记页，而不需要编译器生成额外的指令。

记忆集合的优点在于精确，因为它包含的是被更新记录的地址，而卡标记方法的优点在于简单。对于每条更新指令的维护，记忆集合方法可能需要 10 条额外的指令。在卡标记最少的情况下只需要两条额外指令就够了，它所需要的开销远远小于记忆集合方法。如果将两种方法结合在一起使用，会带来更高的效率。

当收集开始时，由记忆集合可知，老一代的哪些记录（或者卡、页）可能包含指向 G_0 的指针。这些指针是 G_0 根集的一部分。

通常，G_i（$i>0$）是呈指数地大于 G_{i-1}，如 G_0 是 0.5MB 大小，那么 G_1 应该是 2MB 大小，G_2 就应该是 8MB 大小。假设 G_0 是当前要收集的代，它里面的活跃记录将复制到 G_1 中。经过几次对 G_0 的收集之后，G_1 可能聚集了相当数量的无用单元需要被收集。因为 G_0 可能包含了很多指向 G_1 的指针，所以最好将 G_0 和 G_1 一起收集。和前面一样，这时需要扫描记忆集合，以获得包含在 $G_2, G_3, \cdots$ 中的那部分根。这样，各代都可能被收集，当然越老的代收集频率越低。

11.3.5 渐增式收集

虽然收集时间的总和只占整个程序运行时间很小的百分比，但是收集器仍有可能将运行程序中断相对长的时间。例如，分代算法提到 $G_0, G_1, \cdots$ 的大小呈指数变化，当对较老或最老的代进行收集时，可能就需要较多的时间。对于交互式程序和实时程序来说，这一点是难以接受的。**渐增式收集器**或者**并发收集器**将程序运行和无用单元收集交错进行，避免了出现

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第470页 文件编号: 422624 页面编号: 34704538445 打印人: 8685 2026-03-09 10:23:06

这种长时间的中断。

在渐增式算法中，只有当运行程序请求时，收集器才开始工作；在并行算法中，收集器可以在运行程序执行任何指令期间或之间进行工作。渐增式算法和并发算法解决的问题虽然类似，但是显然后者更加复杂和困难。

这些收集算法的困难在于，当收集器在做遍历以得到一个可达记录图时，运行程序并没有停止修改可达记录图。因此，收集算法必须有某种方法来跟踪可达记录图的变化。对于这些变化，可能需要重新计算可达记录图中的某些部分。已经有很多这方面的技术，在此不作介绍。这些技术都需要编译器生成额外的指令来提供支持，有些也可以利用虚拟内存的硬件来实现。

11.3.6 编译器与收集器之间的相互影响

在收集器的设计中，对收集器的底层有以下基本要求。

（1）收集器必须能确定堆上分配的记录大小，这样它们才能被复制。

（2）收集器必须能定位包含在堆记录中的指针，这样它们才能被跟踪和更新。

（3）收集器必须能定位所有在全局变量中的指针。

（4）收集器必须能在程序中任何一个可以进行收集的地方找到所有在活动记录栈中和寄存器中的指针。

（5）收集器必须能找到所有由指针运算所产生的值指向的记录。

（6）收集器必须能在记录被移动时，更新所有涉及的指针值。涉及的指针包括指向记录的指针，以及由指向记录的指针经计算而得到的指针。

这些要求都需要编译器的支持才能得到满足，因为很多所需信息只有在编译时能够获得。这是编译器必须对收集器提供支持的一些基本方面。在上述 6 点中，对最后两点的理解需要阅读下面的导出指针部分。

概括地说，对于使用无用单元收集技术的语言，其编译器与收集器之间的互响是：编译器生成用来分配记录的代码（可能需要用收集器提供给它的接口函数）；编译器为每个收集周期提供根集元素的存储位置描述；编译器向收集器提供堆上数据记录的布局描述等。

另外，对于渐增式收集和分代收集，编译器还必须生成一些额外的代码，如分代收集中所讲的建立记忆集合的指令。

1. 快速分配

某些编程语言和程序可以快速分配堆记录，同时快速产生无用单元。对于函数式语言来说尤其如此，因为它不鼓励更新旧数据。

可以想象，最快的分配记录和产生无用单元的速度是每遇到一条存储指令就分配一个字，这是因为一个堆分配记录的每个字通常都需要被初始化。经验数据表明，无论采用哪种编程语言或者编写什么样的程序，每 7 条指令中就有一条是存储指令。因此，可以认为每条运行指令要进行 1/7 个字的分配。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第471页 文件编号: 422624 页面编号: 34714538443 打印人: 8685 2026-03-09 10:23:06

即使收集的代价可以通过调整分代收集的代的数目变得很小，那么仍然需要较大代价用于创建堆记录。若想最小化这样的代价，应使用复制收集方法，因为它供分配的空间是连续空间，而不是空闲链表。若 free 指针指示下一个空闲位置，limit 指针指示空闲区域的末端（见图 11.14），要分配一个 N 字节大小的记录，一般是调用分配函数，下面列出其执行步骤。

（1）分配函数的调用序列。

（2）测试 free + N < limit ？（如果失败，则调用收集器）。

（3）result = free。

（4）将 free 开始的 N 字节都清空。

（5）free = free + N。

（6）分配函数的返回序列：

① 将 result 移到对后面的计算有用的某个地方；

② 用一些有用的值将该记录初始化。

通过将分配函数在每个调用它的地方进行内联展开，步骤（1）、步骤（6）可以删除。步骤（3）经常可以删除，将它合并到步骤 ① 即可，步骤（4）也可以删除，用步骤 ② 即可。步骤 ① 和步骤 ② 之所以编号不同，是因为它们应该被看成有用计算中的一部分，而不属于分配开销。

步骤（2）、步骤（5）不能删除，但是如果在同一基本块中不止一个分配，则它们可以被多个分配操作共享。将 free 和 limit 放在寄存器中，步骤（2）、步骤（5）总共用 3 条指令即可完成。

通过这种合并技术，分配一个记录并且最终回收它的开销大约只需要 4 条指令即可。

2. 堆上数据布局的描述

收集器必须能够处理各种类型的记录，这些类型包括链表、树、程序声明的类型等。它必须能够知道每种类型的域的数目，以及其中哪些是指针域。

对于静态类型化的语言，如 Pascal 语言和 C 语言，或者面向对象的语言，如 Java 语言，确定堆记录最简单的方法就是让每个记录的第一个字指向一个特殊的类型或者类的描述符记录。该描述符记录会告知该记录的大小以及每个指针域的位置。

这样，对于静态类型化语言，每个记录有一个字（描述符指针）的开销用于收集。但是对于面向对象语言，每个对象本来就需要这样一个描述符指针用来实现对方法的动态查找，因此每个对象不存在用于收集的额外开销。

类型或类的描述符必须由编译器的语义分析阶段所得的静态类型信息来生成。描述符指针将是运行时系统的 alloc 存储分配函数的参数。

除了描述堆上每个记录，编译器还必须为收集器确定每个包含指针的临时变量和局部变量，不管它们是在一个寄存器中还是在一个活动记录中。因为每条指令都可能改变活跃临时变量的集合，所以**指针映像**（指活跃指针的集合）在程序的每个点都可能不同。于是，为简单起见，编译器一般只在收集工作可以启动的地方描述指针映像，这些地方就是 alloc 函数的调用点。此外，因为任何被函数调用可能会调用 alloc 的函数，所以在每个函数调用点必须描述指针映像。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第472页 文件编号: 422624 页面编号: 34724538441 打印人: 8685 2026-03-09 10:23:06

在函数调用点的指针映像，检索它的最好键值是该调用点的返回地址，即一个处于地址 a 处的函数调用最好用紧跟在它后面的返回地址来索引，因为返回地址保存在活动记录中，所以它是收集器从活动记录栈中能看到的值。

在收集开始时，为了获得所有的根，收集器从栈顶开始，对所有活动记录逐个地往下遍历栈。其中每个活动记录中保存的返回地址都是描述下一个活动记录的指针映像入口的关键字。在每个活动记录中，收集器从源于这个活动记录的指针去进行标记（如果是标记和清扫方法）或者转移（如果是复制收集方法）。

3. 导出指针

程序的指针有时可能指向一个堆记录的中间，或者指向这个记录的前面或后面。例如，表达式 a[i−2000] 经编译后被当作 M[a − 2000 + i] 来计算（其中 M[a − 2000 + i] 理解为取地址 a − 2000 + i 的内容）：

$t_1 = a - 2000$

$t_2 = t_1 + i$

$t_3 = M[t_2]$

如果表达式 a[i − 2000] 出现在一个循环内部，那么代码优化可能会将 $t_1 = a - 2000$ 作为循环不变计算提升到该循环外面。如果该循环包括一个 alloc 调用，并且一个收集发生时 t_1 还是活跃的，那么收集器是否会感到困惑？因为指针 t_1 没有指向一个记录的开头，甚至出现更坏的情况，它指向一个不相关的记录。

t_1 被称为从基指针 a 导出的指针。指针映像必须能识别导出指针，并且知道每个导出指针的基指针是哪个指针。于是，当收集器将 a 重新分配到地址 a' 时，它必须把 t_1 调整到 $t_1 = t_1 + a' - a$。

当然，这意味着只要 t_1 活跃，a 就必须保持活跃。考虑图 11.16 左边的一个函数体，其执行语句就是一个循环，该函数体的实现在图 11.16 的右边，假定数组分配在堆上。如果不存在 a 的其他使用，那么变量 a 在对 t_1 的赋值后不再活跃。但是和返回地址 L2 相关的指针映像将不能适当地解释 t_1。于是，对于编译器的活跃变量分析来说，一个导出指针的活跃隐性要求保持它的基指针活跃。

```
int a[100] = {0};                    r1 = 100
int i;                               r2 = 0
for (i = 2001; i <= 2100; i++)       call alloc
    f(a[i-2000]);                    a = r1
                                     t1 = a-2000
                                     i = 2001
                                L1 : r1 = M[t1+i]
                                     call f
                                L2 : if i <= 2100 goto L1
```

图 11.16 一个函数体及它的实现

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第473页 文件编号: 422624 页面编号: 34734538449 打印人: 8685 2026-03-09 10:23:06

习 题

11.1 如果 cfile 是一个 C 语言源程序（注意，该文件名没有扩展名），在 x86/Linux 系统上，某版本的编译命令

cc cfile

的结果是错误信息

/usr/bin/ld: cfile: file format not recognized: treating as linker script
/usr/bin/ld: cfile: 1: parse error
collect2: ld returned 1 exit status

请解释为什么会是这样的错误信息。

11.2 C 语言程序（存储为一个文件）

```
long gcd(p,q) long p,q; {
      if (p%q == 0)
    return q;
      else
    return gcd(q, p%q);
}
main() {
      printf("\n%ld\n",gcdx(4,12));
}
```

在 x86/Linux 系统中用某版本的 gcc 命令得到的编译结果为

In function 'main':
undefined reference to 'gcdx'
ld returned 1 exit status.

请问：这个 gcdx 没有定义，是在编译时发现的，还是在连接时发现的？试说明理由。

11.3 一个 C 语言程序的 3 个文件的内容如下：

```
head.h:
short int a = 10;

file1.c:
#include "head.h"
main() {
}

file2.c:
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第474页 文件编号: 422624 页面编号: 34744538447 打印人: 8685 2026-03-09 10:23:06


```
#include "head.h"
```

在 x86/Linux 系统上用某版本的命令

cc file1.c file2.c

编译，其结果报错的主要信息为

multiple definition of 'a'

试分析为什么会报这样的错误。

11.4 一些 C 语言程序设计的教材上指出："在需要使用标准 I/O 库中的函数时，应在程序前使用 #include <stdio.h> 预编译命令，但在用 printf（函数和 scanf）函数时，则可以不要。"（备注：有些编译器给出警告错误）事实上，并非仅限于这两个函数。例如，下面的 C 语言程序编译后运行时输出字符 A 并换行，它并没有预编译命令 #include <stdio.h>。试解释为什么。

```
main() {
      putchar('A');
      putchar('\n');
}
```

11.5 把下面左边的 file1.c 文件提交给编译器，编译器没有报告任何错误。而把 file2.c 文件提交给编译器，错误报告如下：

file2.c: 2: error: conflicting types for 'func'

file2.c: 1: error: previous declaration of 'func'

试分析原因。(在这两个文件中，第一行都是函数 func() 的原型，第二行都是函数 func() 的定义，函数体为空。)

file1.c

```
int func(double);
int func(f) float f; {}
```

file2.c

```
int func(double);
int func(float f) {}
```

11.6 C 语言的一个源文件可以包含若干函数，该源文件经编译可以生成一个目标文件；若干目标文件可以构成一个函数库。如果一个用户程序引用库中的某个函数，那么在连接时的做法是下面 3 种情况中的哪种，说明你的理由。

(a) 将该库函数的目标代码连接到用户程序。

(b) 将该库函数的目标代码所在的目标文件连到用户程序。

(c) 将该函数库全部连到用户程序。

11.7 cc 是 UNIX 系统上 C 语言编译命令，-l 是连接库函数的选项。某程序员自己编写了两个函数库：libuser1.a 和 libuser2.a，当用命令

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第475页 文件编号: 422624 页面编号: 34754538444 打印人: 8685 2026-03-09 10:23:06


```
cc test.c -luser1.a -luser2.a
```

编译时，报告有未定义的符号，而改用命令

```
cc test.c -luser2.a -luser1.a
```

时，能得到可执行程序。试分析原因。（备注：库名中的 lib 在命令中省略。上面第一个命令和命令 `cc test.c libuser1.a libuser2.a` 的效果是一致的。）

11.8 cc 是 UNIX 系统上 C 语言编译命令，-l 是连接库函数的选项。两个程序员分别编写了两个函数库：libuser1.a 和 libuser2.a。当用命令

```
cc test.c -luser1.a -luser2.a
```

编译时，报告有重复定义的符号，而改用命令

```
cc test.c -luser2.a -luser1.a
```

时，能得到可执行程序。试分析原因。

11.9 现将图 11.2 中的 swap.c 编译成可重定位目标文件 swap.o。对于在 swap.o 文件中定义或引用的符号，请说明它是否在 swap.o 的.symtab 节中有相应的符号表条目？如果有，指出该符号被定义的模块（swap.o 或 main.o）、符号类型（local、global 或 extern），以及所在节（.text、.data 或.bss）。

符号	在 swap.o 的.symtab 节中是否有条目	符号类型	由哪个模块定义	所在节
buf				
bufp0				
bufp1				
swap				
temp				

11.10 修改习题 11.2 中的程序，将 main 函数中的 gcdx 改为 gcd，并将修改后的程序保存在 gcd.c 中。对该程序采用以下两种方式进行编译、连接

```
gcc -o gcd1 gcd.c
gcc -static -o gcd2 gcd.c
```

所产生的可执行目标文件 gcd1 和 gcd2 的大小并不相同，前者约为 11KB，后者约为 1MB。请分析产生这种不同的原因，它们在执行时存在什么样的差异？

11.11 a 和 b 表示当前目录中的目标模块或静态库，a→b 表示 a 依赖于 b，即 b 定义了被 a 引用的符号。对于以下情况，给出最小的命令行（包含最少数目的目标文件和库参数），从而使静态连接器能解析所有的符号引用。

（a）p.o→libx.a

（b）p.o→libx.a→liby.a

（c）p.o→libx.a→liby.a 并且 liby.a→libx.a→p.o

11.12 两个 C 语言程序文件 link1.c 和 link2.c 的内容分别为

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第476页 文件编号: 422624 页面编号: 34764538442 打印人: 8685 2026-03-09 10:23:06


```
    int buf[1] ={100};
```

和

```
    extern int *buf;
    main() {
        printf("%d\n", *buf);
    }
```

在 x86/Linux 系统上，经 cc link1.c link2.c 命令编译后，运行时产生如下的出错信息：

```
    Segmentation fault
```

请说明原因。

11.13 两个 C 语言程序文件 long.c 和 short.c 的内容分别为

```
    long i = 32768 * 2;
```

和

```
    extern short i;
    main() {
        printf("%d\n", i);
    }
```

在 x86/Linux 系统上，用 cc long.c short.c 命令编译这两个文件，能否得到可执行目标程序？若能得到目标程序，运行时是否报错？若不报错，则运行结果输出的值是否为 65536？若不等于 65536，原因是什么？

11.14 下面左右两边分别是两个 C 语言程序文件 file1.c 和 file2.c 的内容，用 cc file1.c file2.c 命令对这两个文件进行编译和连接。请回答：

(a) 编译器是否会报错？若你认为会，则说明理由。

(b) 若编译器不报错，连接器是否会报错？若你认为会，则说明理由。

(c) 若上面两步都不报错，则运行时是否会报错？若你认为会，则说明理由。

(d) 若上面三步都不报错，则运行输出的结果是什么？说明理由。

```
char k = 2;
char j = 1;
```

```
#include <stdio.h>
extern short k;
main(){
    printf("%d\n", k);
}
```

11.15 下面左右两边是一个 C 语言程序的两个源文件的内容：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第477页 文件编号: 422624 页面编号: 34774538440 打印人: 8685 2026-03-09 10:23:06


```
#include <stdio.h>
char *p = "0123456789";
f() {
    printf("%s\n", p);
}
```

```
#include <stdio.h>
char p[10];
main() {
    f(); p[1] = '1'; f();
}
```

运行目标程序所得到的结果（编译器是 GCC: (GNU) 4.2.3 (Debian 4.2.3-5) ）为

0123456789

Segmentation fault

请描述该目标程序各数据区的内容，并回答为什么运行时会出现 Segmentation fault。

11.16 下面 3 列是 3 个 C 语言程序（其中第 3 个程序由两个文件构成）。它们在编译、连接或运行时是否报错？若是，则指出编译器、连接器或运行时系统报告错误的原因。对于不报错的程序，除了告知输出结果，还要对比有错的程序，说明为什么没有错误。

```
#include <stdio.h>
void f(m) long m[2]; {
    printf("%d, %d\n",
    m[0], m[1]);
}
typedef struct {
long n[2];
} stype;
main() {
    stype k;
    k.n[0]=0; k.n[1]=1
    f(k);
}
```

```
#include <stdio.h>
void f(m) long m[2]; {
    printf("%d, %d\n",
  m[0], m[1]);
 }
main() {
  long m[2];
    m[0]=0; m[1]=1;
    f(m);
 }
```

```
long k[2]={0,1};
#include <stdio.h>
typedef struct {
long n[2];
} stype;
extern stype k;
main() {
 printf("%d, %d\n",
 k.n[0], k.n[1]);
}
```

第 12 章

面向对象语言的编译

随着软件系统规模的扩大和复杂性的不断增加，人们对高效、透明的开发方法的需求也日益增加。通过标准化构件来构建系统以达成类似于硬件的模块化设计，成为追求的目标。模块化、可重用性、可扩展性和抽象性是实现这一目标的重要方向，而面向对象编程和现代开发框架为此提供了新的可能性。如今，面向对象编程已成为管理复杂软件系统的核心范式，与其他先进技术共同推动了软件开发的高效与灵活性。

本章回顾面向对象语言的一些重要概念，概述它们的实现技术。为说明一些重要概念的实现技术，本章以 C++ 语言为例，介绍如何将 C++ 程序翻译成 C 程序；实际的编译器大都把 C++ 程序直接翻译成低级语言程序，而不是把 C 语言作为中间语言并利用 C 语言编译器。

12.1 面向对象语言的概念

面向对象语言可被看成命令式语言，除了变量、数组、结构体和函数等被熟知的概念，它还引入一些新概念。本节只介绍面向对象语言的主要概念。

12.1.1 对象和对象类

命令式语言的主要模块化单元是过程（包括函数）。当数据的复杂性与处理的复杂性相比显得微不足道时，适宜采用抽象和模块化的方法。但是，当任务的描述和有效的解决方案需要使用复杂的数据结构时，单用函数进行模块化是不够的。有效处理这些任务的合适抽象级别应该是允许把数据结构和操作这些数据结构的相关函数封装在一个单元中。

面向对象语言最基本的概念是**对象**。一个对象由一组**属性**和操作于这组属性的过程组成，属性到值的映射称为对象的**状态**，过程也称为**方法**。属性和方法共同形成了对象的**特征**。因此，对象封装了数据及其上的操作。面向对象语言最重要的基本操作是激活对象 o 的方法 m，写成 o.m。在这种范式中，对象处于核心地位，而方法则是对象的成分并且从属于对象。

面向对象语言拓宽了 Pascal 和 C 等命令式语言的类型概念，增加了**对象类**（简称类）的

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第479页 文件编号: 422624 页面编号: 34794538446 打印人: 8685 2026-03-09 10:23:06

概念。一个对象类规范了该类中对象的属性和方法，包括它们的类型和原型（参数和返回值类型）。一个对象要想属于该类，它必须含有这些特征，当然还可以含有其他一些特征。某些面向对象的语言（如 Eiffel）允许对方法进一步规范，如规定方法的前置条件和后置条件，用于约束方法的语义。这种机制通常被称为“设计契约”（design by contract）。

对象类形成了面向对象语言的模块单元。对于圆、椭圆、矩形、三角形、多边形、点、直线和折线等图形对象，可以为不同类型的图形对象定义不同的对象类。同一类中的不同对象，其属性值可能不同，因此对象必须有自己存放属性的存储单元，但是它们的方法是一样的，因此它们可以共享方法的代码。这一原则指导 12-2 节开始介绍的 C++ 程序到 C 程序的翻译。

下面将把术语“类”和“类型”看成是同义的。

12.1.2 继承

继承定义为类 A 的所有特征并入到新的类 B，B 可以进一步定义自己的一些其他特征，在一定条件下还可以重写或覆盖（overwrite）从 A 继承来的方法。某些语言允许重新命名继承来的特征，以避免名字冲突，或者允许在新的上下文中使用更有意义的名字。

如果类 B 继承类 A，那么类 B 称为类 A 的**派生类**，而类 A 称为类 B 的**基类**。

继承是面向对象语言最重要的概念之一。继承的层次性允许把类库结构化和引入不同级别的抽象。这里仍以图形对象的例子来说明这一点。图 12.1 给出了图形对象的继承层次结构。

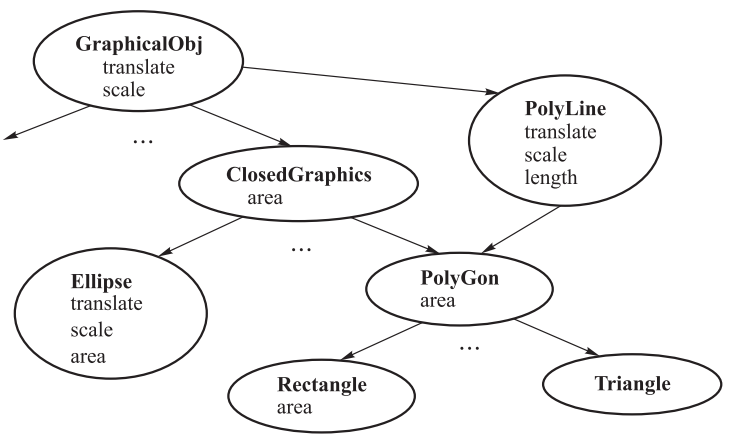


图 12.1 图形对象的继承层次结构

在图 12.1 中，用椭圆形表示对象类，椭圆形中包括类名和该类的部分方法；继承由箭头表示。例如，图形对象类 GraphicalObj 有方法 translate 和 scale，那么所有的图形对象都可以被平移（translate）和缩放（scale）。封闭图形类 ClosedGraphics 和折线类 PolyLine 从类 GraphicalObj 继承了这些方法。在类 PolyLine 中，这些被继承的方法被覆盖，并且引入计算

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第480页 文件编号: 422624 页面编号: 34804538444 打印人: 8685 2026-03-09 10:23:06

周长的方法 length。类 ClosedGraphics 引入新方法 area，它计算封闭图形对象的面积。多边形类 PolyGon 同时继承类 ClosedGraphics 和类 PolyLine，其中 area 被覆盖。矩形类 Rectangle 继承类 PolyGon 并且覆盖 area。

虽然方法 translate 和 scale 在类 GraphicalObj 中引入，但是它们却不能在这个类中定义。因为这些方法只能结合具体的图形对象来定义，如椭圆类 Ellipse 和折线类 PolyLine。不过，这些方法在类 GraphicalObj 中引入，意味着每个图形对象必须有这些方法，虽然这个类并不提供它们的实现。包含未定义方法的类称为**抽象类**，它没有自己的任何对象实例。

类似地，类 ClosedGraphics 引入方法 area，但该方法也不能在这个类中定义。area 首先在类 Ellipse 和类 PolyGon 中定义。但是，对一般的多边形而言，面积计算是复杂的，它取决于该多边形能划分成多少个三角形以及这些三角形的面积之和。另外，矩形的面积计算是简单的。如果矩形经常使用，最好在类 Rectangle 中以更有效的方法实现 area，如引入矩形的边长属性，利用它们直接计算面积。对于类 PolyLine 的所有方法，类 PolyGon 最好接管所有这些方法，不做任何覆盖。

综上所述，继承概念提供了这样的可能性：用简单的方法可以复用或扩充部分现有实现，还可以重写个别方法以适应局部的特殊需求和环境。

此外，任何事物，如果它能用较高级别的抽象表述，那么它就有较广的应用可能性，因而有较高程度的可复用性。所以应尽可能使用高的抽象级别。另外，有时必须转移到更具体的级别，使得在解决某些特定问题时有更多的结构可用。假定图形对象的一种变换可以由一串平移、缩放和图形对象的一些其他操作描述，那么该变换可以用一个函数实现，该函数对任何类型的图形对象都可用。如果无抽象类并且编译器要进行类型检查，那么需要为每个类编写一个这样的函数，即使这些函数的实现具有相同的内容也不能省略。

于是，类型化的面向对象语言考虑其类型系统中的继承层次结构。如果类 B 继承类 A，那么指派给 B 的类型是指派给 A 的类型的子类型。子类型的每个对象自动地是它超类型的一个元素。一个继承类是被它继承的类的**子类**。这样就有下列三条规则。

（1）**子类型规则**：当在某个输入位置（函数的输入参数、赋值的右部）需要某个类型的一个对象时，该类型的任何子类型的对象可以出现在这些位置。若函数的返回值要求是某个类型的一个对象，该类型的任何子类型的对象也可以作为返回值。

类 B 的一个对象，若它不同时是 B 的某个真子类的对象，那么称该对象是 B 的一个**真对象**，称 B 是该对象的**运行时类型**。这样，每个对象有唯一且确定的运行时类型，它是该对象所属的最小类型。此外，一个对象也是它运行时类型的每个超类型的元素。

凭借子类型规则，面向对象语言的方法可以接受运行时类型不同（因而结构也不同）的对象，如在参数位置就可以这样。这是多态性的一种形式。

继承的子类型规则及继承类对被继承方法的覆盖，引发了一个值得关注的问题。它对编译器来说是重要的。例如，有一个函数 f()，它允许类 ClosedGraphics 的对象作为参数，并假定 f() 调用该参数的 area 方法。因为类 ClosedGraphics 不能定义 area 方法，所以调用的实

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第481页 文件编号: 422624 页面编号: 34814538442 打印人: 8685 2026-03-09 10:23:06

参肯定是类 ClosedGraphics 某子类的对象，而不是它本身的真对象。

（2）**方法选择规则**：如果类 B 继承类 A 并且重写了方法 m，那么对类 B 的对象 b 来说，即使它作为类 A 的对象使用，也必须使用在类 B 中定义的方法 m。

该规则给编译器提出一个问题：编译器必须产生调用一个方法的代码，但在编译时它很可能确定不了究竟要调用哪个方法。对上面的例子来说，在 f 的代码生成中，编译器不知道应该把名字 area 绑定到哪个类的 area 方法。一般来说，这个绑定只有在运行时当实参已经明确的情况才能完成，这样的绑定称为**动态绑定**。因此，为体现动态特征，也可以把方法选择规则按下面的方式表述。

（3）**动态绑定规则**：当对象 o 的一个方法可能被子类重新定义时，如果编译器不能确定对象 o 的运行时类型，那么必须对该方法进行动态绑定。

12.1.3 信息封装

大多数面向对象语言提供了一种机制，它可用来将类的特征分为私有的（private）和公共的（public）两类。私有特征完全不可见，或者至少在某个上下文中不可见。某些面向对象语言用不同的上下文区分作用域，如“在一个类中”“在派生类中”“在友元类中”等。语言构造或一般性规则可以指明上下文的可见、可读、可写或可调用特征。

由编译器来实现这些作用域规则是简单而明显的。因此，虽然信息封装是非常重要的，但是下面不讨论它的实现。

现在，把到目前为止所讨论的概念小结如下：

（1）面向对象语言引入了新的模块化单元：对象类。对象类可以封装数据及在这些数据上的操作。

（2）继承概念对构造现有模块（对象类）的派生模块是极其有用的。

（3）面向对象语言的类型系统使用继承概念：派生类是其基类的子类型，派生类的对象可以用在其基类的对象所允许出现的任何地方。

（4）继承层次结构把不同级别的抽象引入程序，即在程序或系统的不同点使用不同级别的抽象。

（5）抽象类型可用作规范（规格说明），通过逐步继承而求精，直至最终的实现。也就是说，它提供从规范经过逐步设计到各种实现的无缝转变。

本章将以面向对象语言 C++ 和 Eiffel 为例进行介绍。

12.2 方法的编译

本节用具体的例子来说明方法的编译。这里用 C++ 语言描述前一节图形对象的部分类层次结构。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第482页 文件编号: 422624 页面编号: 34824538440 打印人: 8685 2026-03-09 10:23:06

先定义一般的图形对象类 GraphicalObj 如下：

```
class GraphicalObj {
    virtual void translate (double x_offset, double y_offset);
    virtual void scale (double factor);
    . . .   // 可能还有一些其他方法
};
```

其中，方法 translate 和 scale 声明为虚方法。在 C++ 语言中，这是要求它的派生类重写此方法。

类 GraphicalObj 的一个特别重要的子类是点类 Point。几乎每种具体的图形对象类的实现都以某种方式使用点类。类 Point 的定义如下：

```
class Point : public GraphicalObj {
    double xc, yc;
    public :
    void translate (double x_offset, double y_offset) {
        xc += x_offset;
        yc += y_offset;
    }
    void scale (double factor) {
        xc *= factor;
        yc *= factor;
    }
    Point(double x0 = 0, double y0 = 0) {xc = x0; yc = y0; }
    void set(double x0, double y0) {xc = x0; yc = y0;}
    double x(void) {return xc;}
    double y(void) {return yc;}
    double dist (Point &);
};
```

一个点有 x 轴和 y 轴坐标 xc 和 yc，它们表示点在二维空间中的位置。xc 和 yc 是点的私有数据，只能在该类的方法中访问（或者由友元函数访问）。在类 Point 中定义的方法是公共的，它可以在知道一个点的任何场合使用。例如，如果 p 是一个点，那么 p.x( ) 激活 p 的方法 x 并且返回 p 的 x 轴坐标；p.translate(1, 2) 使点 p 在 x 轴方向上移动一个单位，在 y 轴方向上移动两个单位。

一般而言，对一个对象 o，它带有参数 $\mathrm{arg}_1, \mathrm{arg}_2, \cdots, \mathrm{arg}_n$ 的方法 m 由 o.m($\mathrm{arg}_1, \mathrm{arg}_2, \cdots, \mathrm{arg}_n$) 激活。

将一个 C++ 语言的类翻译成 C 语言的程序段，主要步骤有如下几点（由继承引出的问题放在 12.3 考虑）：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第483页 文件编号: 422624 页面编号: 34834538448 打印人: 8685 2026-03-09 10:23:06

（1）将 C++ 语言中一个类的所有非静态属性构成一个 C 语言的结构体类型，取类的名字作为结构体类型的名字。

（2）类的静态属性是该类的所有对象共有的，应当翻译成 C 语言中的全局变量，但是需要改一个名字。根据下面（4）中的 ①，读者应该明白如何改名。

（3）C++ 语言中类的对象声明不加翻译就成了 C 语言中相应结构体类型的变量声明，不管对象声明出现在程序中的什么位置。

（4）在解释类的非静态方法的翻译之前，先做一个约定。在 C++ 语言中，函数的参数传递有值调用和引用调用两种方式，当形参名前加字符 & 时，表示该参数是引用调用。但是 C 语言的参数传递只有值调用。为简便起见，下面假定 C 语言也有引用调用方式，也用字符 & 表示这种方式，以避免在解释如何翻译引用调用上花笔墨。

将 C++ 语言中类的非静态方法翻译成 C 语言的函数，对应的方法和函数的区别如下：

① 由于类声明在 C++ 语言中形成一层作用域，类中方法声明的作用域就是该类；而在 C 语言中，函数声明的作用域至少是所在的文件。为了避免不同类的同名方法在 C 语言程序中变成同名函数，函数的名字必须在原来方法名的基础上修改，比较容易的做法是把类名字加上去。考虑方法的重载，参数类型也编码到函数名中，这样才能保证不会有名字冲突。

② 与方法声明相比，函数声明增加一个形参，作为它的第一个形参，该形参的类型就是对应该类的结构体类型，该形参的名称通常取 this，传递方式是引用调用。

③ 与方法调用相比，在函数体中出现的函数调用也要增加一个实参，作为它的第一个实参。若原来是调用本对象的方法，则新增的实参就是 this；若是调用其他对象的方法，则新增实参是该对象对应的结构体变量。

④ 在方法中对本对象的非静态属性的访问，改成对 this 相应域的访问。在方法中对其他对象的非静态属性的访问不必修改，直接就成了对对应结构体变量的相应域的访问。若是对静态属性的访问，则翻译成对 C 语言的全局变量的访问。

（5）类的静态方法在定义和调用时，与该类的特定对象无关，因此在翻译时，无须增加表示当前操作对象的参数，只需要按（4）中的 ①，将方法名改成函数名即可。

必须注意，对 C++ 程序的语法和语义分析在这个翻译之前进行，因此生成 C 程序时，C++ 语言的可见性规则已经检查过；C 语言的可见性规则虽然不一样，这时也没有什么影响了。

下面通过一个例子来看方法是怎样用等价的函数实现的。

【例 12.1】 如果 m 是类 C 的一个非静态方法，它的原型是“返回值类型 m（形参表）”，那么等价于 m 的函数 fm 的原型是（下面给出的语法是非标准的）：

返回值类型 fm（C &this, 形参表）

其中，C &this 表示第一个形参的类型和它的名字，传递方式是引用调用。如果 m 中有对当前对象的非静态属性 k 的访问，有对 m 本身的递归调用，有对某个对象 o 的非静态方法 n 的调用 o.n，有对 o 的非静态属性 k 的访问 o.k，那么它们的翻译结果如表 12.1 所示。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第484页 文件编号: 422624 页面编号: 34844538446 打印人: 8685 2026-03-09 10:23:06

表 12.1 类 C 的方法 m 被翻译成函数 fm

	方法	函数
原型	返回类型 m（形参表）	返回类型 fm（C &.this，形参表）
调用	m（实参表） o.n（实参表）	fm（this，实参表） fn（o，实参表）
属性访问	k o.k	this.k o.k

类 Point 的方法 x 翻译成等价的函数`x__5Point`，其定义为

```
double x__5Point (Point &this) { return this.xc;}
```

方法调用 p.x( ) 翻译成`x__5Point(p)`。

类 Point 的方法 translate 翻译成函数`translate__5Pointdd`：

```
void translate_ _5Pointdd(Point &this, double x_offset , double y_offset)
{
    this.xc += x_offset;  this.yc += y _offset;
}
```

方法的名字本身没有作为实现函数的名字，而是扩展成还包含所属类的类名和参数类型的编码。把类名编码加到实现函数的名字中是必要的，类名的编码保证了生成的函数名是唯一的。考虑到方法的重载，在 C++ 语言的实现中，参数类型也编码到函数名中。例如，名字`translate__5Pointdd`有下列五部分：

（1）方法名 translate；

（2）分隔符`__`；

（3）类名的编码 5Point，领头的数表示后面有多少个字符属于类名；

（4）double 类型的编码 d（第一个参数的类型）；

（5）double 类型的编码 d（第二个参数的类型）。

从本节可以看出，把 C++ 这样的面向对象语言的程序翻译成 C 语言程序的可能性。

12.3 继承的编译方案

继承是面向对象语言引入的最重要概念。本节讨论它的实现。

如果类 B 直接或间接继承类 A，那么类 A 是类 B 的超类，类 B 的对象可以用在几乎所有类 A 的对象可用的地方。出于效率的考虑，编译器要求类的对象具有某种灵活的结构，为了使类 B 的对象可以作为类 A 的对象使用，编译器必须以一种有效的方式产生类 B 对象的 A 视图。在这种视图中，编译器关于类 A 的对象结构的假设必须满足。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第485页 文件编号: 422624 页面编号: 34854538443 打印人: 8685 2026-03-09 10:23:06

大家知道，类 A 的虚方法可以在类 B 中被重写。12.1.2 小节给出的动态绑定规则要求，如果编译器不能直接确定类 A 的对象 o 的运行时类型，那么该方法应该动态绑定。例如，如果对象 o 的运行时类型是 B，那么应该使用 B 的方法，而不是 A 的方法。在这种情况下，编译器必须做一些准备，使得在程序运行时，被激活方法所期望的视图（B 视图）能够有效地从 A 视图产生。

许多面向对象语言，尤其是一些老的面向对象语言，仅支持单一继承。12.3.1 小节先讨论编译单一继承的编译方案。12.3.2 小节接着讨论编译多重继承的编译方案。

12.3.1 单一继承的编译方案

对于只有单一继承的语言来说，每个类最多从一个类继承。这种语言的继承层次结构是树或森林。

图 12.2 的程序描述类 PolyLine 和它的派生类 Rectangle。

```
#include "graphicalobj.h"              /* imported GraphicalObj */
#include "list.h"                      /* imported lists */
#include "point.h"                     /* imported Point */
class PolyLine : public GraphicalObj {
      list <Point> points;
   public:
      void translate (double x_offset,double y_offset);
      virtual void scale (double factor);
      virtual double length (void);
};

#include "polyline.h"
class Rectangle : public PolyLine {
      double side1_length,double side2_length;
   public :
      Rectangle (double s1_len,double s2_len,double x_angle=0);
      void scale (double factor);
      double length (void);
};
```

图 12.2 类 PolyLine 和它的派生类 Rectangle

出于效率的原因，在矩形类 Rectangle 的定义中引入属性 side1_length 和 side2_length 存储矩形两条邻边的长度，这允许得到一个效率较高的 length 的重新定义。scale 也必须重

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第486页 文件编号: 422624 页面编号: 34864538441 打印人: 8685 2026-03-09 10:23:06

新定义，因为缩放操作会改变边的长度。此外，translate 可以直接由 PolyLine 的相应定义接管，因为平移操作不改变边的长度。

仍须解释编译器是怎样有效地实现动态绑定的。在图 12.2 中，PolyLine 的 scale 方法不能用于缩放矩形，因为它不知道新加的矩形边长属性也由缩放操作改变，所以必须使用 Rectangle 的 scale 方法。我们知道，类 Rectangle 的对象可以作为方法的参数传递，只要超类 PolyLine 的对象在这些地方被允许，例如它可以用于函数 zoom：

```
void zoom (GraphicalObj &obj, double zoom_factor, Point &center) {
    obj.translate (-center.x, -center.y);   // 将“中心点”移至“点(0, 0)”
    obj.scale (zoom_factor);                // 缩放
}
```

该函数首先平移一个图形对象，使得中心点 center 落到原点，然后根据值 zoom_factor 缩放该对象。

如果函数 zoom 作用于矩形，那么 zoom 的体必须调用 Rectangle 的缩放函数，而不是 PolyLine 或 GraphicalObj 的缩放函数。然而，zoom 可能在类 Rectangle 被定义前已经被编译并存于库中。也就是说，当编译 zoom 时，编译器不知道运行时在 zoom 体应该激活哪个方法。而且，在 zoom 的不同调用点，应该激活的方法可能是不一样的。因此，编译器不可能把 scale 绑定到一个具体的方法，而不得不由 zoom 在执行时将 scale 动态地绑定到一个方法。

编译器可以用下面的方案来有效地处理动态绑定。对于每个类，编译器会建立一张方法表，该表包含定义在该类中且必须动态绑定的方法。在 C++ 语言中，这样的方法表称为**虚函数表**，它们包含一个类或它的超类中所有定义为 virtual 的方法的入口。12.2 节提到，每个对象在 C 程序中有对应的结构体，现在为这样的结构体增加一个域，作为第一个域，其内容是这样的方法表的指针。编译器把方法名绑定到方法表的索引。当运行时调用某方法时，存储在方法表中相应索引下的函数被激活。继承类的方法表按如下方式产生：首先复制其基类的方法表，其中被重新定义的方法由新的定义覆盖；然后新引入的方法被追加到这张表上。这就保证了基类中定义的方法名在新类中具有相同的方法表索引。

如果 B 是一个类，A 是 B 的超类，那么类 B 的对象 b 的 A 视图包含两部分：b 的前一部分域和 b 引用的方法表的前一部分。其中，属于 A 视图的 b 的前一部分域，是由方法表的指针和从 A 继承的属性组成的；属于 A 视图的方法表部分包括在 A 及其超类中引入的方法的索引。b 的其他各种视图由一个指向 b 的指针以同样的方式表示。

图 12.3 和图 12.4 用来解释该编译方案。

图 12.3 给出了 GraphicalObj、PolyLine 和 Rectangle 的方法表。PolyLine 的方法表是从 GraphicalObj 的方法表派生出来的。首先，由 PolyLine 重新定义的方法 translate_PL 和 scale_PL 取代 GraphicalObj 中的相应方法。然后新定义的虚方法 length_PL 被加入。依次地，Rectangle 的方法表从 PolyLine 中的方法表派生。在 Rectangle 中重新定义的方法取

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第487页 文件编号: 422624 页面编号: 34874538449 打印人: 8685 2026-03-09 10:23:06

代 PolyLine 的相应方法。没有重新定义的方法 translate_PL 仍然保留。编译器把 translate、scale 和 length 分别绑定到方法表索引 0、1 和 2。

图 12.4 给出了 Rectangle 的对象表示。除了自身的状态外，每个这样的对象包含指向类 Rectangle 方法表的指针。

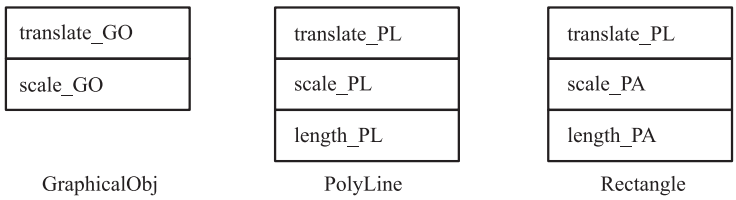


图 12.3 图形对象的不同子类的方法表

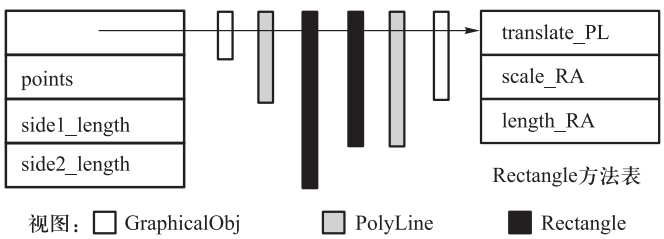


图 12.4 Rectangle 的对象表示

用动态绑定来实现单一继承，每个对象需要一个指针的额外存储空间开销。另外，与每个类关联的方法表都需要占用存储空间。动态绑定引起了运行时方法调用的时间增加，因为通过指针找到方法表及获得要被激活方法的入口地址需要消耗时间。

12.3.2 多重继承的编译方案

单一继承的编译方案比较容易，而多重继承对语言定义和编译器设计来说，都具有很大的挑战性。

在有多重继承的语言中，一个类可以从多个类继承，即一个类可以有多个直接的超类。因此，继承的层次结构不再是树，而是有向无环图。

多重继承的使用支持这样的编程风格：基本功能用一些很小的基类实现，通过继承用这些基类去构造更加复杂的类。多重继承也可用于抽象类，如可以把一个抽象类给出的规范和作为它的实现的一个具体类集成起来。对于一个预先定义了大小的栈，这样的类可以通过抽象类 Stack 和作为它实现的具体类 Array 的继承来实现。

所有会碰到的问题和可能的解决方法都可以通过双重继承来阐明。本节假定类 C 同时从类 B1 和 B2 派生，如图 12.5 所示。下面两点引出了语言定义中的问题，在某种程度上也引

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第488页 文件编号: 422624 页面编号: 34884538447 打印人: 8685 2026-03-09 10:23:06

出了编译器设计的两个问题：

（1）B1 和 B2 间的冲突与矛盾。例如，如果在两个基类中，方法或属性使用相同的名称，那么继承将会引起冲突。

（2）多重继承。例如，若 B1 和 B2 都直接从 A 继承，则 C 将从 A 多重继承。

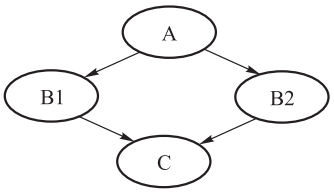


图 12.5 多重继承

问题（1）基本上是语言定义的问题。下面一些解决方法可以独立或组合地应用于语言设计中：

① 将 B1 定义为主要后代，冲突解决优先于 B1。这种方法主要用于解释执行的 Lisp 的面向对象的扩充上。它通过预先定义的次序查找继承层次结构来动态地绑定名称。在图 12.5 中，首先 C，然后 B1，最后 B2。以首先找到的为准。

这种办法并非没有缺点，因为开发者不一定能发现可能的冲突。

② 语言允许重新命名被继承的特征，允许程序员通过显式的干预来解决可能的冲突。使用这种方式的有 Eiffel 语言。

③ 语言提供别的显式手段来解决冲突。例如，如果 B1 和 B2 中名字 n 的定义有矛盾，那么 B1::n 或 B2::n 将无二义地指明应该使用 B1 还是 B2 的 n 定义。使用这种方式的有 C++ 语言。

实现这些解决方法并不困难，只涉及编译器符号表的组织和管理问题。这里不再讨论。

但对于问题（2），存在以下两种截然相反的解决方法：

① 被多重继承的类可以有多个实例，如图 12.6 所示。

② 被多重继承的类只能有一个实例，如图 12.7 所示。

图 12.6 多重继承的多个实例

图 12.7 多重继承的单个实例

这两种方法都有各自的优缺点。在有些场合下需要被多重继承的类有多个实例，而在另一些场合只需要它们的单个实例。甚至会有这样的需求：对被多重继承的某些特征需要单个实例，而对另一些特征则需要多个实例。Eiffel 语言提供了这种灵活性。

C++ 语言包含对上述两种方法的支持。以图 12.5 所示的多重继承为例，在 C++ 语言中，当把这些类定义为

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第489页 文件编号: 422624 页面编号: 34894538445 打印人: 8685 2026-03-09 10:23:06


```
class A { ··· };
class B1 : public A { ··· };
class B2 : public A { ··· };
class C : public B1, public B2 { ··· };
```

时，不论是类 B1 或类 B2 都内含一个类 A 的复制，这样在类 C 的对象布局中将包含两个独立的类 A 子对象。在 C++ 语言中，一般直接称这种继承为多重继承。

若将类 B1 和类 B2 设定为从类 A 虚拟继承，再让类 C 从类 B1 和类 B2 继承，即

```
class A { ··· };
class B1 : public virtual A { ··· };
class B2 : public virtual A { ··· };
class C : public B1, public B2 { ··· };
```

这时，在类 C 的对象布局中将只包含一个类 A 子对象。

下面考虑仅允许被多重继承的类有多个实例的编译方案。这些方案比同时允许单个实例的编译方案要简单，产生的代码效率也更高。在此讨论独立的多重继承的编译方案。

在独立的多重继承情况下，来自基类的继承是相互独立的。相应地，继承类 C 的对象包含基类 B1 和 B2 的完整复制，如图 12.8 所示。

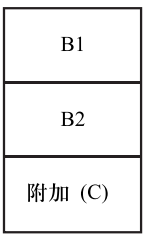


图 12.8 独立的多重继承的对象结构（程序视图）

如图12.6 所示，多重继承在下述情况会导致冲突和二义：

（1）当多实例的特征被用于访问、调用和覆盖的时候。

（2）当类 C 的对象的 A 视图被建立时。因为类 C 的对象包含多个类 A 的子对象。

可见性规则可在某些情况下帮助规避这些问题，如 C++ 语言允许一个类对它的继承者隐藏它自己的继承性。例如，B1 可以私有地从 A 继承而 C 并不知道这一点，此时 B1 不是 A 的子类，并且 C 中不会由于 A 的多个实例而出现二义。在可见性规则不足以消除二义的地方，需要引入额外的语言构造：在 C++ 语言中，受限运算符“::”可以用于 B1::f 的形式，以表示特性 f 属于 B1。此外，还可以使用类型转换，如显式地把 C 的对象转换成 B1 的对象。

现在把单一继承的编译方案扩充到这种独立的双重继承场合。在单一继承的情况下（见图 12.4），为了有效地实现方法的动态绑定，在每个对象中加入了一个指针作为该类的第一个成分，该指针指向方法表。这里仍然保持这种方式。

注意，对于类 C 的每个超类 B，编译器必须能够产生类 C 对象的 B 视图。因为 B1 子对象处在 C 对象的开头，所以对 B1 仍可以使用单一继承的办法，即类 C 对象的 B1 视图是 C 视图的开头部分（对象成分和方法表都这样）。但是不能用 C 视图的开头部分作为 B2 视图，因为一个对象的 B2 视图必须有一个方法表的指针作为它的第一个成分，该方法表的内

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第490页 文件编号: 422624 页面编号: 34904538443 打印人: 8685 2026-03-09 10:23:06

容是依据 B2 和 C 确定的；跟随该指针的是 B2 的属性值。这就导致下面的方法：在 C 对象中，在 B2 属性值的前面加上指向另一方法表的指针；这个方法表由 B2 方法表的复制经 C 中重新定义的方法覆盖而产生。于是，B2 视图由一个 B2 引用表示，它指向 B2 子对象的开始。B2 子对象的第一个成分是指向 B2 子对象方法表的指针。对于超类 A 的每个实例，编译器知道 C 对象中对应子对象的偏移。编译器通过把这个偏移加到 C 引用而产生相应的 A 引用。得到的结构显示在图 12.9 中。

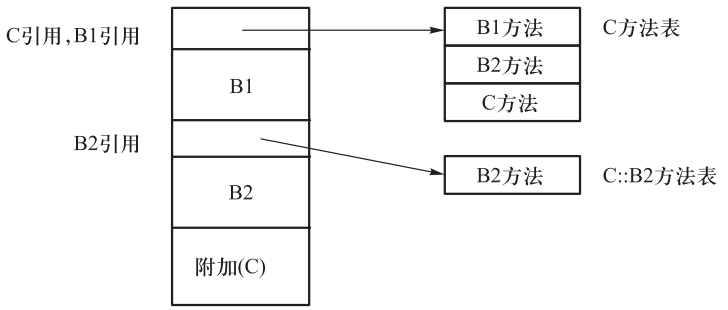


图 12.9 独立的多重继承的对象结构（实现视图）

下面通过例子来讲解这种偏移在对象视图切换中的应用。例如，对于语句

```
B2 *pb2 = new C;
```

则新创建的 C 对象的地址在赋给 pb2 时必须调整，以指向其 B2 子对象，即提供 C 对象的 B2 视图。编译时将会产生以下代码：

```
C *temp = new C;
B2 *pb2 = temp ? temp + sizeof(B1) : 0;
```

使得 pb2 是 B2 引用，以便通过它使用 B2 子对象的特征。

当程序员要删除 pb2 所指的对象时，例如：

```
delete pb2;
```

如果指针 pb2 所指对象的运行时类型是 C，那么 pb2 的值必须从 B2 引用调整到 C 引用，即调整到 C 视图。然而，由于 pb2 所指对象的运行时类型一般不是静态可确定的，因此上述调整的偏移量一般不是静态可确定的。

更复杂的情况是，在 C 中定义的方法总是期望得到它为之激活的对象的 C 视图。如果一个这样的方法覆盖超类 A 的一个方法（更精确地说，超类 A 某实例的方法，因为 C 可以含 A 的几个实例），那么在该方法被激活时，可能只有相应对象的 A 视图可用。运行时，必须能够从它计算出所需的 C 视图。如果 A 和 C 都是已知的，那么这很容易做：因为视图可

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第491页 文件编号: 422624 页面编号: 34914538441 打印人: 8685 2026-03-09 10:23:06

由相应的引用表示，并且对于每个 C 对象，A 引用和 C 引用之间的差 d 是一个常数，即在 C 对象中 A 子对象的偏移，所以 C 引用可以用 A 引用减去 d 计算出来。但可惜的是，在该方法调用（它可以出现在 B1 或 B2 某方法的代码中）被编译和运行时，C 可能是未知的。

基于这些原因，编译器把用于确定所需视图的偏移存放在方法表中下邻该方法指针的地方。方法表中的每个入口有方法指针和偏移两个成分，使得该方法所期望的视图可以从该方法激活时的可用视图中生成。

于是第 i 个虚函数的调用操作，由

```
(*pb2->vptr[i])(pb2);
```

改为

```
(*pb2->vptr[i].faddr)(pb2 + pb2->vptr[i].offset);
```

其中，faddr 是虚函数的地址；offset 是调整视图的偏移。

这种做法的缺点是，不管是否需要用 offset 来调整，所有的虚函数调用都必须这样操作。

图 12.9 描述的方法有一个虽小但很重要的问题：类 C 的两个方法表包含 B2 方法表的两个（修改过的）副本。结果是，存放类 C 的方法表所需的存储空间可能会随 C 定义的复杂度增加而呈指数增长 [类定义的复杂度指的是类的展开定义（unfolded definition）的大小，而类的展开定义是通过把类定义中的基类定义用该基类的展开定义代替而得到的]。

为了避免存储空间呈指数增长，必须只使用 B2 方法表的一个副本。为做到这一点，把 C 方法表以分布方式存储，如图 12.10 所示。该图隐蔽了一个有意义的细节：B1 和 B2 方法表的副本也可以按分布的方式存储。

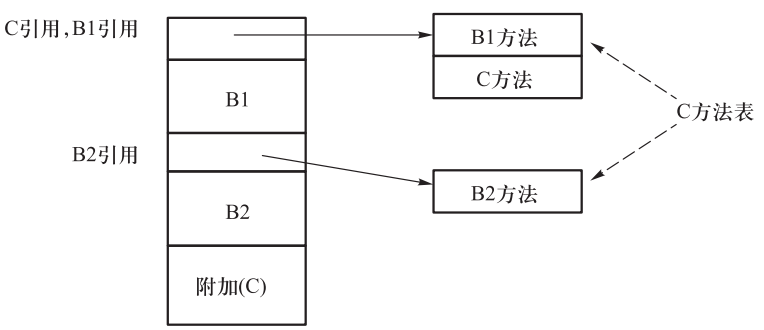


图 12.10 独立的多重继承的对象结构（分布式存储）

单个实例的多重继承，以及其他一些形式的多重继承，它们的实现都比较复杂，在此不再介绍。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第492页 文件编号: 422624 页面编号: 34924538449 打印人: 8685 2026-03-09 10:23:06

习 题

12.1 图 12.11 给出了图形对象类库的部分程序。根据 12.3.1 小节的编译方案，确定该图中定义的类的方法表和方法索引。

```
class ClosedGraphics : public GraphicalObj {
  public:
    // Surface area enclosed by object
    virtual double area (void);
};

class Ellipse : public ClosedGraphics {

  Point _center;                    // Centre of the ellipse
  double _x_radius,_y_radius;       // Radii of the ellipse
  double _angle;                    // Rotation from x-axis

public:
  // Constructor
  Ellipse (Point &center,double x_radius,double y_radius,double angle=0) {
      _center=center;
      _x_radius=x_radius;
      _y_radius=y_radius;
      _angle=angle;
  }

  // Ellipse area--overwrites 'ClosedGraphics :: area'
  double area (void) {return PI * _x_radius * _y_radius;}

  // Distance to a point--expensive!
  virtual double dist (Point &);

  // Center
  const Point & center (void) {return _center;}

  // Translate--overwrites 'GraphicalObj :: translate'
```

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第493页 文件编号: 422624 页面编号: 34934538447 打印人: 8685 2026-03-09 10:23:06


```
    void translate (double x_offset,double y_offset) {
        _center. translate(x_offset,y_offset);
    }

    // Scale--overwrites 'GraphicalObj :: scale'
    void scale (double scale_factor) {
        _x_radius * =scale_factor;
        _y_radius * =scale_factor;
    }

    //…
};

class Circle : public Ellipse {
  public:
        // Constructor
        Circle (Point &center,double radius) {
            Ellipse (center,radius,radius);
        }

        // Distance to a point--overwrites 'Ellipse :: dist'
        virtual double dist (Point &p) {
            double center_dist=_center. dist (p);
            if (center_dist <=radius) return 0;
            else return center_dist-radius;
        }
        // …
    };
```

图 12.11 类 ClosedGraphics、Ellipse 和 Circle

12.2 根据独立的多重继承的编译方案，确定图 12.11 中 Circle 的方法表和方法索引，并且以编译形式为 c.dist (p) 的方法调用，其中 c 和 p 分别是类 Circle 和类 Point 的对象。

12.3 C++ 语言中的对象声明语句应如何翻译成 C 语句，如图 12.11 程序中的

```
Point  _center;
```

应翻译成什么？

12.4 表 12.1 给出了函数调用的翻译，但是这种翻译方式不能用于虚函数的情况。试说明 12.3.1 小节中的虚函数调用 obj.scale zoom_factor 应该翻译成什么形式的 C 语句。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第494页 文件编号: 422624 页面编号: 34944538445 打印人: 8685 2026-03-09 10:23:06

12.5 12.2 节在介绍类的翻译时有这么一句话：“将 C++ 语言中一个类的所有非静态属性构成一个 C 语言的结构体类型，取类的名字作为结构体类型的名字”。在学完本章后，你认为这句话需要修改吗？

12.6 在面向对象语言中，编译器给每个对象分配空间的第 1 个域存放虚方法表的指针，是否可以把虚方法表的指针作为最后一个域而不是第一个域？请简要说明理由。

12.7 在面向对象语言中，编译器给每个类建立虚方法表，如图 12.3 和图 12.4 那样。请简要说明，为什么编译器给每个类仅建立虚方法表，而不是建立所有方法的方法表。

12.8 12.2 节的结尾介绍，类 Point 的方法 translate 翻译成函数`translate__5Pointdd`:

```
void translate_ _5Pointdd(Point &this, double x_offset , double y_offset) {
    this.xc += x_offset;  this.yc += y_offset;
}
```

其中假设 C 语言也有引用调用方式（形参前面加字符 &）。请按 C 语言只有值调用方式来考虑，写出方法 translate 翻译成的 C 语言函数，并说明调用点需要做的修改。

12.9 请按图 12.4 的方式给出 12.2 节中类 Point 的对象表示。

第 13 章

函数式语言的编译

函数式程序设计语言起源于 LISP，可以追溯到 20 世纪 60 年代初期。到 20 世纪 70 年代末，当新的概念和实现方法出现时，这类语言才摆脱了 LISP 语言的统治，成为一类比较成熟的语言，其代表有 Miranda 和 ML 等。

在函数式语言中，函数是构造程序的基本成分，并且语言还提供一些机制用于构造更为复杂的函数。纯函数式语言禁止使用赋值语句，从而不会产生副作用，其优点是具有引用透明性，有助于程序的等式变换和推理。

函数式程序设计的主要任务是定义（或构造）函数，以求解所提出的问题。所定义的作为主程序的函数可以包含一些辅助函数（可看作子程序）。计算机按照所定义函数的相应表达式，根据计算规则逐步计算，最后得到所需的结果。表达式中可能包含函数名，计算时可将其相应的定义作为归约规则。

13.1 函数式编程语言简介

本节介绍一种简单的函数式编程语言 SFP（Simple functional Programming），并用它来解释编译函数式编程语言的一些原理。本章的重点在于定义一个适当的抽象机和为该机器产生代码，而非关注诸如类型检查等编译细节。例如，多态性概念由编译器的其他部分支持，因而不放入 SFP 语言中；另外，SFP 语言也不考虑函数定义中的模式匹配或模板等复杂特性。

13.1.1 语言构造

现在描述 SFP 语言的构造。为此，假定 SFP 有一些基值类型和这些类型上的运算。这些基值类型是什么，对下面的讨论并不重要，但它们必须包括布尔类型。另外，假定这些类型的每个值都只需占用抽象机的一个存储单元，以简化讨论。SFP 的表达式通过使用内部定义的运算符和函数抽象及函数应用，可以直接从基值和变量归纳地构造。SFP 的语法论域总结在表 13.1 中，SFP 的语法总结如图 13.1 所示。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第496页 文件编号: 422624 页面编号: 34964538440 打印人: 8685 2026-03-09 10:23:06

$$
\begin{aligned}
e \longrightarrow\ & b \mid v \mid (op_{\text{un}}\ e) \mid (e_1\, op_{\text{bin}}\ e_2) \\
& \mid (\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) \\
& \mid (e_1\ e_2) && \text{// 函数应用} \\
& \mid (\lambda v.\, e) && \text{// 函数抽象} \\
& \mid (\textbf{letrec}\ v_1 == e_1; && \text{// 联立递归定义} \\
& \qquad\quad v_2 == e_2; \\
& \qquad\quad \cdots \\
& \qquad\quad v_n == e_n \\
& \quad\ \textbf{in}\ e_0)
\end{aligned}
$$

图 13.1 SFP 的语法

表 13.1 SFP 的语法论域

元素名字	语法论域	
b	B	基值集合，如布尔值、整数、字符 ……
op_{bin}	Op_{bin}	基值上的二元运算符集合，如 +、 −、 =、 ≠、**and**、**or** ……
op_{un}	Op_{un}	基值上的一元运算符集合，如 −、**not** ……
v	V	变量集合
e	E	表达式集合

对表 13.1 中的每个语法论域，给它一个“类型化”的名字，这样便于在图 13.1 中引用这些集合上的元素。图 13.1 中的 SFP 大部分构造是清楚的，但是函数应用、函数抽象和联立递归定义需要给出解释。

构造 $\lambda v.e$ 定义一个一元函数，其中 v 是形式参数，e 是定义该函数的表达式，即函数体。函数 $\lambda v.e$ 应用到表达式 e' 写成 $(\lambda v.e)e'$。根据该语言的语义和实现，它的效果是用表达式 e'（或它的值）代替形式参数 v 的所有自由出现，或者说把 v 约束到 e'（或 e' 的值）。

为了少用括号，规定函数应用具有最高优先级且遵循左结合；算术和逻辑运算符遵循常规的优先级顺序。在一个表达式中，λ 抽象选择最大可能的语法表达式作为 $\lambda v.e$ 的体 e，即 e 延伸到表达式的结尾或碰到第一个不能配对的右括号为止。

SFP 的语法允许函数定义和函数应用的嵌套。n 元函数可以用 $f == \lambda v_1.\lambda v_2.\cdots.\lambda v_n.e$ 来定义，一个函数可以用 $fe_1e_2\cdots e_m$ 的方式应用到若干变元。为了书写的简便和执行的效率，通常把 n 元函数写成 $\lambda v_1\lambda v_2\cdots\lambda v_n.e$ 的形式，并把 $fe_1e_2\cdots e_m$ 实现为一次函数应用，而不是 m 次函数应用。

为了保证函数应用的语义十分清楚，有两点必须说明：一是参数传递机制，即对于 e_1e_2，传给 e_1 的是什么；二是在解释自由变量时，是用静态约束还是用动态约束，即第 6 章所讲的静态作用域和动态作用域。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第497页 文件编号: 422624 页面编号: 34974538448 打印人: 8685 2026-03-09 10:23:07

13.1.2 参数传递机制

在第 6 章介绍过值调用、换名调用和引用调用。在函数式语言中，后者没有意义，因为对变量只使用名字和值，不使用地址。在函数式语言中，对于表达式 e_1e_2，参数传递机制确定表达式 e_2 本身还是它的值被传递。有以下 3 种可能性需要区分：

（1）**值调用**。先计算 e_2，然后把它的值传给 e_1。其优点是 e_2 只计算一次。其缺点是即使 e_2 的值不用，它也得计算；这时如果 e_2 的计算不终止，那么它将是灾难性的。

（2）**换名调用**。从 e_1 的计算开始，每当需要 e_2 时，就计算它的值。于是 e_2 以没有被计算的形式传到 e_1 中相应形式参数出现的地方。其优点是 e_2 只在真正被需要时才计算，因而它比值调用有较好的终止性。其缺点是 e_2 可能计算多次，而每次计算的值都一样。

（3）**按需调用**（又称**惰性计算**）。从 e_1 的计算开始，当第一次需要 e_2 时，计算它的值，也就计算这一次。因此，第一次访问时引起 e_2 的计算，其他的访问用第一次访问时计算的值。这种方式结合了前两种方式的优点。

在 SFP 语言中，对用户定义的函数用按需调用的方式传递参数。

虽然参数传递机制和约束规则并非完全独立，但是在讨论参数传递机制时，暂不考虑是静态约束还是动态约束。

下面通过几个例子来熟悉 SFP 语言。

【例 13.1】 表达式

$$\begin{aligned}&\textbf{letrec } x == 2;\\ &\quad f == \lambda y.x + y;\\ &\quad F == \lambda g\ x.g\ 2\\ &\textbf{in } F\ f\ 1\end{aligned}$$

如果是静态约束，f 体中的自由变量 x 引用定义 $x == 2$，此时，$F\ f\ 1$ 的值是 4。若是动态约束，在 f 体中访问 x 的值之前把 x 约束到 1，因此结果是 3。

考虑静态约束的实现问题。在例 13.1 中，存在 x、f 和 F 的约束，特别是 x 约束到 2。这样的一列约束通常称为**环境**。为了使一个变元（相当于命令式语言中的实参表达式概念）中的自由变量的正确环境在这些自由变量的每一个出现处都可用，该环境 u 和变元 e 需要一起传递。这样形成的二元组 (e,u) 称为**闭包**。在闭包 (e,u) 中，环境 u 用来保证 e 中的自由变量会被正确地解释。当然，在值调用场合，有时也需要形成闭包，主要是在有自由变量的函数作为参数传递的时候。

表达式 **letrec** $v_1 == e_1;\cdots;v_n == e_n$ **in** e_0 把 n 个新名字 $v_1,\cdots,v_n$ 填入环境，它们的作用域是 $G = e_0e_1\cdots e_n$。可以这样直观地解释，每当这些名字在这个作用域范围内碰到且它们的值被需要时，定义这些名字的等式的右部就被使用。

SFP 语言虽然很简单，但仍可以被用来编程。

【例 13.2】 表达式

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第498页 文件编号: 422624 页面编号: 34984538446 打印人: 8685 2026-03-09 10:23:07

letrec fac $==\lambda n$ **if** $n = 0$ **then** 1 **else** $n * \text{fac}\ (n-1)$;
fib $==\lambda n$ **if** $n = 0$ **or** $n = 1$ **then** 1 **else** $\text{fib}(n-1) + fib(n-2)$;
one $==\lambda n.1$
in $\text{fib}((\text{fac}\ 2) + \text{one}((\text{fac}\ 2) - 2))$

SFP 语言展示了函数式语言的一个重要特征，即高阶函数。函数可以作为函数的变元，也可以作为函数的结果。

【例 13.3】 表达式

letrec $F == \lambda xy.xy$; // 函数作为变元
inc $==\lambda x.x + 1$
in F inc 5 // 值是 6

letrec comp $==\lambda f.\lambda g.\lambda x.f(gx)$; // 函数作为变元和结果
$F == \lambda x.\cdots$;
$G == \lambda z.\cdots$;
$h ==$ comp F G
in $h(\cdots) + F(\cdots) + G(\cdots)$

每个 n 元函数可以作为一个高阶函数使用。当它作用于 m $(m < n)$ 个变元时，由于变元个数不足，其结果是一个 $n - m$ 元的函数。

13.1.3 变量的自由出现和约束出现

在命令式语言中，变量和类型等的声明及形式参数的声明引入新的名字。在 SFP 语言中，名字定义在 letrec 等式的左部和函数定义 $\lambda v_1\ \cdots\ v_n.e$ 的 $\lambda v_1 \cdots v_n$ 中。把前者称为**等式定义的**名字，后者称为 λ **定义的**名字。就函数式语言的语义和它的编译来考虑，在一个表达式中，一个（全局）变量的自由出现与它的定义性出现之间的联系是重要的。下面先归纳定义一个表达式 e 中自由变量的集合 $\text{freevar}(e)$，然后定义约束变量的集合。

定义 13.1 表达式中自由出现的变量集合。

$\text{freevar}(\text{b}) = \varnothing$ // 只由基值构成的表达式无自由变量
$\text{freevar}(v) = \{v\}$ // 只由一个变量构成的表达式含一个自由变量，即它本身
$\text{freevar}(op_{\text{un}}\ e) = \text{freevar}(e)$
$\text{freevar}(e_1\ \text{op}_{\text{bin}}\ e_2) = \text{freevar}(e_1) \cup \text{freevar}(e_2)$
$\text{freevar}(\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) = \text{freevar}(e_1) \cup \text{freevar}(e_2) \cup \text{freevar}(e_3)$
$\text{freevar}(e_1\ e_2) = \text{freevar}(e_1) \cup \text{freevar}(e_2)$
$\text{freevar}(\lambda v_1 \cdots v_n.e) = \text{freevar}(e) - \{v_1, \cdots, v_n\}$
// e 中的自由变量$v_1 \cdots v_n$在整个表达式中是受约束的

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第499页 文件编号: 422624 页面编号: 34994538444 打印人: 8685 2026-03-09 10:23:07

$$\text{freevar}(\textbf{letrec } v_1 == e_1; \cdots; v_n == e_n \textbf{ in } e_0) = \bigcup_{i=0}^{n} \text{freevar}(e_i) - \{v_1, \cdots, v_n\} \quad // \text{ 同上}$$

如果 $x \in \text{freevar}(e)$，就说 x 在 e 中有**自由出现**。

类似地，可以定义约束出现的变量集合。

定义 13.2 表达式中约束出现的变量集合。

$$\text{bdvar}(b) = \varnothing$$
$$\text{bdvar}(v) = \varnothing$$
$$\text{bdvar}(\text{op}_{un}\ e) = \text{bdvar}(e)$$
$$\text{bdvar}(e_1\ \text{op}_{\text{bin}}\ e_2) = \text{bdvar}(e_1) \cup \text{bdvar}(e_2)$$
$$\text{bdvar}(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) = \text{bdvar}(e_1) \cup \text{bdvar}(e_2) \cup \text{bdvar}(e_3)$$
$$\text{bdvar}(e_1\ e_2) = \text{bdvar}(e_1) \cup \text{bdvar}(e_2)$$
$$\text{bdvar}(\lambda v_1 \cdots v_n.e) = \text{bdvar}(e) \cup (\{v_1, \cdots, v_n\} \cap \text{freevar}(e))$$
$$\text{bdvar}(\textbf{letrec } v_1 == e_1; \cdots; v_n == e_n \textbf{ in } e_0) = \left(\bigcup_{i=0}^{n} \text{bdvar}(e_i)\right) \cup \left(\{v_1, \cdots, v_n\} \cap \bigcup_{i=0}^{n} \text{freevar}(e_i)\right)$$

如果 $x \in \text{bdvar}(e)$，就说 x 在 e 中有**约束出现**。 □

【例 13.4】 表达式

$$e = (\lambda x\, y.(\lambda z.x + z)\,(y + z))\,x$$

中的自由变量和约束变量分别为

$$\text{freevar}(e) = \{x, z\}$$
$$\text{bdvar}(e) = \{x, y, z\}$$

可以看出，在一个表达式中，一个变量可以既有自由出现又有约束出现。但是，一个变量的某个具体出现只能是自由的或约束的。

定义 13.3 自由出现和约束出现。

变量 x 的一个出现是**自由的**，如果这个出现既不是 $\lambda \cdots x \cdots .e$ 的项 e 的子项，也不是 **letrec** $v_1 == e_1; \cdots; v_n == e_n$ **in** e_0，并且 $x = v_j$，$e = e_i$（$0 \leqslant i \leqslant n$，$1 \leqslant j \leqslant n$）的项 e 的子项。否则，$x$ 的这个出现是**约束的**。 □

在例 13.4 中，$x + z$ 中的 z 是约束的，而 $y + z$ 中的 z 是自由的。

在 SFP 语言中采用**静态约束**，换句话说，在一个表达式或子表达式中，变量的自由出现联系到该变量的第一个外围 **letrec** 定义或 λ 抽象（命令式语言的静态作用域）。

13.2 函数式语言的编译简介

13.3 将介绍一种函数式抽象机（functional abstract machine，FAM），它的机器语言是 SFP 语言的目标语言。更一般地说，SFP 程序用按需调用语义和静态约束的函数式语言的程

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第500页 文件编号: 422624 页面编号: 35004538449 打印人: 8685 2026-03-09 10:23:07

序可以编译成 FAM 的机器语言。该机器有一个栈，生存期符合栈式管理的所有变量都存在栈中；它还有一个堆，所有其他的变量都存在堆中。

本节首先用一连串的例子来启发后面从 SFP 程序到 FAM 程序的编译描述，然后讨论环境和约束。

13.2.1 几个受启发的例子

首先从简单的例子开始，然后逐步引入较复杂的例子。一个 SFP 程序最外的表达式称为程序表达式，换句话说，该表达式的计算给出程序的结果。

【例 13.5】 程序表达式是

$$e = 1 + 2$$

希望 e 的编译结果被执行时，它把 e 的值留在 FAM 上可访问的存储空间（栈或堆）中。基值类型的结果是可以保留在栈中的，但是如果结果是一个函数，它就没有办法存放在栈中。当希望所有的程序表达式被执行时都以同样的方式表达结果，且独立于它们的结果类型时，可以把程序表达式的结果统一存放在堆中，在栈顶用一个指针指向堆中的结果。

【例 13.6】 有表达式

$$\textbf{letrec}\ x == 1/y; y == 0; z == x\ \textbf{in}\ 1 + 2$$

该程序表达式必须这样编译，其结果 3 存放在堆中，并且栈顶的指针指向堆中这个位置。该表达式包含变量。在函数式语言中，变量代表值。针对实现的效率而言，快速访问这些值是重要的。在编译（而不是解释）实现中，尽可能将可直接访问的存储单元分配给这些变量。在 SFP 实现中，由 **letrec** 或函数抽象引入的变量将在 FAM 的栈中分配单元。

x、y 和 z 的等式应该这样编译：产生的指令序列并不直接计算这些等式的右部（将来真正需要这些值的时候再计算）。另外，x、y 和 z 分别约束到这些右部的信息必须在将来需要 x、y 或 z 的时候是可用的。于是，生成的指令序列必须构造 x、y 和 z 的闭包，并分别将指向这 3 个闭包的 3 个指针存放在栈中。因为表达式 1 + 2 的计算不需要访问 x、y 或 z 的值，所以这 3 个闭包根本不计算。

再引入两点改进。一是 y 的等式无须构造闭包，因为它的右部不含自由变量，只要让指针指向一个对象，该对象由值 0 与标明其为基值的标记组成。二是让 z 和 x 约束到同一个闭包。

可以看出，上下文清楚地决定了一个表达式应该怎样编译。例 13.6 中的表达式 $1 + 2$ 必须这样编译：生成的代码产生它的值；而表达式 $1/y$ 的代码必须为它产生一个闭包。

【例 13.7】 表达式

$$\textbf{if}\ \ (\textbf{if}\ \ 1 \neq 2\ \ \textbf{then true else false})\ \ \textbf{then}\ \ 3\ \ \textbf{else}\ \ 4$$

如果仍希望结果在堆中，那么表达式 3 和 4 都必须按这种方式编译，即它们的值都应在堆中。但是对于条件表达式 (**if** $1 \neq 2$ **then true else false**)，FAM 的假转指令 **jfalse** 希望

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第501页 文件编号: 422624 页面编号: 35014538447 打印人: 8685 2026-03-09 10:23:07

在栈顶测试它的值，因此，该条件表达式是这样编译，即生成的指令序列把它的值留在栈中。同样，表达式 **true** 和 **false** 也必须这样编译，即它们的值在栈顶。

【例 13.8】 表达式

$$\begin{aligned}&\textbf{letrec } f = \lambda y\, z.\, \textbf{if } z = 0 \textbf{ then } 1 \textbf{ else } 1/y;\\ &\qquad x == 5\\ &\textbf{in } f\ 1\ (x+1)\end{aligned}$$

这个程序可用来研究两个问题：关于函数变元的闭包的构造和以函数为值的表达式的构造。先从后者开始。从前面的例子知道，为联立等式产生的代码必须为等式的右部形成闭包，并为左部存储一个指向该闭包的指针。像 $x == 5$ 这类等式的优化已经碰到过了，现在来考虑右部是函数定义的情况。当为 $\lambda y\, z.\, \textbf{if } z = 0 \textbf{ then } 1 \textbf{ else } 1/y$ 构造闭包时，把它进一步做成 FUNVAL 对象，因为 f 的函数应用代码的第一步必须保证 f 是可应用的函数。FUNVAL 对象和该闭包的区别仅在于前者还包含存放变元指针的存储空间。构造闭包并随后转变成 FUNVAL 对象是由该等式的代码直接完成的。

怎样编译 f 的变元 1 和 $x+1$？FAM 的按需调用语义使得变元必须以闭包的形式传递，这样才可能保证它们的值仅在需要时计算。因此，必须为 f 的变元 $x+1$ 产生闭包。它包含一个指令序列和一个约束向量，该指令序列的执行可得出该变元的值，该约束向量包含一些指针，它们指向出现在该变元中的自由变量的值（或值的表示）。现在的这个向量仅含指向 x 的闭包的指针。f 的等式和 x 的等式的指令序列是在构造 $x+1$ 闭包之前执行的，得到关于 f 的 FUNVAL 对象和对应 x 的基对象，这两个堆对象的指针在栈中，它们被赋给 f 和 x。当构造 $x+1$ 的闭包时，x 的闭包的指针可以从栈中复制到该闭包的约束向量中。对于 f 的变元 1，由前文可知，不必创建一个闭包，因为它不含自由变量。

【例 13.9】 表达式

$$\begin{aligned}&\textbf{letrec } \ \mathrm{x}== 2 + 1;\\ &\qquad f == \lambda\ a\ b.\ g\ (a+h)\ b;\\ &\qquad g == \lambda x.\cdots\\ &\qquad h = \lambda y.\cdots\\ &\textbf{in } fx\ x\end{aligned}$$

该程序表明，SFP 程序的（以闭包或值形式的）表达式的指针可以复制任意多份，作为函数变元和全局变量的值等。因此在 SFP 的实现中，传递的始终是值的指针和闭包的指针，而非它们本身，并且将它们存放在约束向量和栈帧（相当于第 6 章的活动记录）中。因为复制的是表达式的指针而不是表达式（更确切说是它的闭包）本身，所以每个表达式只有一个实例存在，在 letrec 定义中该表达式对应的变量的所有出现都有一个指针指向该实例。表达式中对应变量的首次使用，会触发该表达式闭包的计算，该计算是面向该变量的所有出现的。以后的出现直接访问这个值而无须重新计算。

【例 13.10】 表达式

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第502页 文件编号: 422624 页面编号: 35024538445 打印人: 8685 2026-03-09 10:23:07


```
letrec f == letrec x == 2
            in λy. x + y
in f 5
```

该程序可用来说明命令式语言和函数式语言在局部变量生存期上的区别。在 Pascal 语言中，除了那些由 new 过程在堆上创建的对象外，所有在过程激活时建立的对象都有相同的生存期，它们在过程终止时消失。

对于 SFP 函数则不同。在上面的程序中，为了把 f 作用于 5，需要计算由最内的 letrec 构造出的函数。x 在这里局部于这个 letrec。若最内的这个 letrec 已经计算，栈式管理会清除属于这个 letrec 的一切内容，包括局部变量。这个例子说明，高阶函数需要延长局部变量的生存期，这意味着在该例中，局部变量必须封装在一个 FUNVAL 对象中，并在堆中留存。

13.2.2 编译函数

13.2.1 节的示例已清楚地表明，同一个表达式在不同的上下文中会编译成不同的指令序列。在前面的示例中遇到了 4 种上下文，分别用不同的字母表示它们：P（program）、B（basic）、V（value）和 C（closure）：

（1）P：编译完整的程序表达式。结果在堆中，栈顶有一指针指向它。它总是最外上下文，编译是在这种上下文中开始的。

（2）B：结果必须是基值且存放在栈中。例如，处理条件表达式中的条件时，这种上下文会出现。

（3）V：结果在堆中，栈顶有一指针指向它。这是计算的正常情况。但是对于基值类型的表达式，作为结果的基值先存放在栈中，然后将它做成一个堆对象。

（4）C：结果必须是被编译的表达式的闭包。函数的变元和递归等式的右部总是这种情况。

以上 4 种上下文对应 4 个编译函数 P_code、B_code、V_code 和 C_code，这些函数为相应上下文中的表达式产生指令序列。

13.2.3 环境与约束

SFP 有两类名字定义：由 λ 定义的名字，它出现在一个 λ 抽象的名字序列中；由等式定义的名字，它出现在 letrec 表达式的一个等式的左部。从例 13.5 ～ 例 13.10 可知，一个名字的定义出现总是关联到一个闭包。当一个等式被编译时，其左部的名字总是关联到其右部的闭包；而 λ 抽象中的约束名字是在函数应用时关联到该应用的变元的闭包。

名字的引用出现应该这样编译：它们获得相关联的定义出现的值。SFP 语言按需调用语义是这样实现的：第一次碰到一个名字的引用出现时，其对应的定义出现的闭包被计算且该闭包被计算结果覆盖，以后再碰到该名字的引用出现时就直接取这个值。FAM 的指令 **eval** 可处理这两种情况。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第503页 文件编号: 422624 页面编号: 35034538443 打印人: 8685 2026-03-09 10:23:07

SFP 的编译器遵循通理，它使用编译时静态可用的信息去有效地管理运行时的动态对象。下面来看一下 SFP 程序的哪些信息是静态的，因此方便用户在编译时使用。表 13.2 列出从程序中可读到的关于函数的静态信息（称为原始静态信息）和从它们可以导出的信息。其中，前两种情况的信息用于确定在函数应用时创建栈帧中的寻址，最后一种情况的信息可用于在约束向量中的相对寻址。

表 13.2 原始信息和导出信息

原始信息	导出信息
变元个数	形参地址（相对于第一个形参的地址）
在函数体中任何一点都可访问的由等式定义的局部名字的集合	这组等式定义的局部名字的地址（相对于第一个等式定义的局部名字的地址）
函数体中全局名字的集合	全局名字在约束向量中的下标

于是，对于 SFP 程序中变量的引用出现，编译器知道它对于直接围绕它的 letrec 或函数定义是局部（约束）的还是全局（自由）的，并且知道哪个相对地址或约束向量中的哪个下标指派给它。这种静态信息包含在对应的**编译环境**中（表 13.3）。

表 13.3 位置和编译环境

名字	论域	备注
p	$P = \{\mathrm{LOC}, \mathrm{GLOB}\} \times \mathrm{integer}$	位置（相对地址或下标）
β	$B = (V \to P)$	编译环境

在 Pascal 编译器中，地址环境中包含变量的相对地址和它的嵌套深度。在 SFP 编译器中，对应每个变量，编译环境包含其位置和有关它是自由变量还是约束变量的信息。

当 SFP 程序被编译时，什么时候编译环境会改变？很显然，当函数抽象的体或 letrec 中的表达式开始编译时，新引入的局部变量必须被加入编译环境。

当生成构造闭包或 FUNVAL 对象的代码时，必须把这时的全局变量构造成它们的**运行环境**。

表达式中的局部变量在 FAM 栈帧上分配存储单元，使用栈帧的相对地址。全局变量存储单元的指针分配在约束向量中，依据下标可以找到这些指针。

13.3 抽象机的体系结构

从本节开始，逐步描述函数式抽象机（FAM）的体系结构和指令集合，以及把 SFP 编译到 FAM 的编译方案。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第504页 文件编号: 422624 页面编号: 35044538441 打印人: 8685 2026-03-09 10:23:07

FAM 的存储器包含一个程序存储区 PS，每个单元含一条 FAM 指令。某些指令含一个运算对象。程序计数器 PC 总是保留当前指令的地址。在正常情况下，FAM 重复执行下列三步：

（1）取当前指令；

（2）PC 增 1；

（3）解释当前指令。

当碰到 **stop** 指令或发生错误时，FAM 停机。

另外，存储器中还包括一个栈 ST 和一个堆 HP，它们的空间不受限制。假定 FAM 有一个堆管理器，它能在执行 new 过程时分配空间，并能自动回收不再使用的空间。

13.3.1 抽象机的栈

FAM 的栈与第 6 章介绍的活动记录栈类似，但 FAM 栈向下增长。SP 是栈顶指针寄存器，它指示已被占用的最高地址。下列 4 种对象都只占用栈中的一个单元：

（1）基值，如整数、布尔值等；

（2）栈地址；

（3）堆地址；

（4）程序指令的地址。

栈被分成若干帧，栈帧在函数应用和闭包计算时建立。函数应用的栈帧与 Pascal 的过程激活时建立活动记录是一致的。闭包计算的栈帧在 Pascal 中无对应的概念，它是按需调用语义要求延迟变元计算引起的，稍后会讨论。图 13.2 所示为这两种情况的栈帧结构。

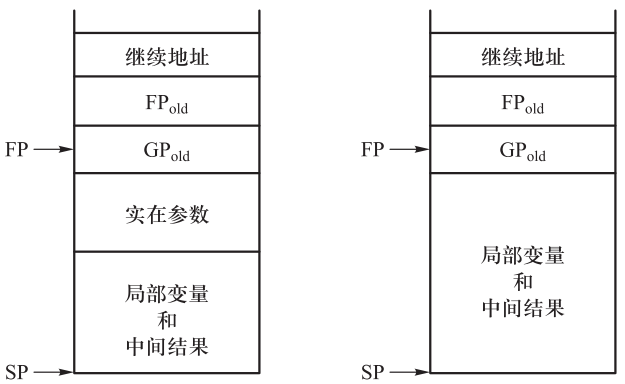


图 13.2 栈帧结构

闭包计算时没有实在参数单元（指向实在参数的指针单元），除此以外，两种情况下的栈帧有同样的结构。栈帧指针寄存器 FP 指示 3 个有序单元中地址最高的那个。第一个单元

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第505页 文件编号: 422624 页面编号: 35054538448 打印人: 8685 2026-03-09 10:23:07

是当前活动（函数应用或闭包计算）结束后继续执行的指令地址，第二个单元是老 FP 的值，第三个单元是所有全局变量构成的向量的指针 GP。FP 和 GP 分别相当于第 6 章所介绍的动态链和静态链，但二者的区别还是有的。在第 6 章中 GP 是链的开始，通过它可以找到所有的非局部变量，在这里 GP 指向一个向量，该向量包含指向一个函数或表达式所有自由变量值的指针。

这种栈帧结构暗示了建立栈帧和释放栈帧可用一组固定的指令，如图 13.3 所示，它们可以集成在 FAM 的 **mark**、**eval** 或 **return** 指令中。

```
ST[SP+1]=继续地址;
ST[SP+2]=FP;
ST[SP+3]=GP;
SP=SP+3;
FP=SP;
```

(a) 建立栈帧

```
PC=ST[FP-2];
GP=ST[FP];
SP=FP-2;
FP=ST[FP-1];
```

(b) 释放栈帧

图 13.3 栈帧的建立与释放指令

13.3.2 抽象机的堆

FAM 的堆所存储的是其生存期和栈管理方式不相容的对象。堆对象有一个标记，用于指示对象的性质。4 种标记 BASIC、FUNVAL、CLOSURE 和 VECTOR 分别表示下列对象。

（1）BASIC：存放基值的单元 b。

（2）FUNVAL：该对象表示一个函数值。它是一个三元组（cf，fap，fgp），其中指针 cf 指向程序区中函数体开始的地方，指针 fap 指向函数变元向量，指针 fgp 是函数各全局变量值的指针所组成的向量的指针。这两个向量也存在堆中。FUNVAL 对象是在函数定义的翻译结果被处理时构造的。这里变元向量当然是空的，因为现在还没有变元。当 n 元函数应用于 m（$m<n$）个变元时，FUNVAL 对象的变元向量就变成非空。如前面所提到的，变元不足的应用结果仍然是函数，它仍由 FUNVAL 对象表示，该对象是在把这些变元装进先前的 FUNVAL 对象后得到的。

（3）CLOSURE：该对象是一个闭包。它表示一个延迟的计算，因为按需调用的参数传递是把变元的代码及其全局变量的值组成一个闭包，并且仅在需要时才计算闭包。该对象有两个成分，即代码的指针 cp 和指向全局变量值指针向量的指针 gp，后面将会看到，变量的所有定义出现都被赋了一个闭包。

（4）VECTOR：该对象是堆对象指针的向量。该向量存放函数变元的指针，或者存放 FUNVAL 对象或 CLOSURE 对象的全局变量的指针。

标记选择子 tag 用来确定堆对象的标记。函数 size 用来确定 VECTOR 对象的向量长度。

某些 FAM 的指令根据标记来解释堆对象的内容，当它不能作用于堆对象（由于对象类

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第506页 文件编号: 422624 页面编号: 35064538446 打印人: 8685 2026-03-09 10:23:07

型不对）时会报告错误。有些指令从最高栈单元的内容构造这样的对象，并且把新创建的这个堆对象的指针放在栈顶。这些指令列在表 13.4 中。

表 13.4 建立堆对象的指令

指令	含义	备注
mkbasic	ST[SP]=new（BASIC：ST[SP]）	建立基本堆对象
mkfunval	ST[SP−2]=new（FUNVAL：ST[SP]，ST[SP−1].ST[SP−2]）； SP=SP−2	建立函数堆对象
mkclos	ST[SP−1]=new（CLOSURE：ST[SP]，ST[SP−1]）； SP=SP−1	建立闭包
mkve n	ST[SP−n+1]=new（VECTOR：ST[SP−n+1]， ST[SP−n+2]，⋯，ST[SP]）； SP=SP−n+1	建立有 n 个分量的向量
alloc	SP=SP+1； ST[SP]=new（CLOSURE：NI，NIL）	建立空闭包

13.3.3 名字的寻址

对于 SFP 来说，栈管理和名字的寻址不同于 Pascal 等语言。函数定义的编译必须考虑函数应用允许变元不足和变元过剩的情况。当编译函数应用 $e\ e_1\ \cdots\ e_m$ 时，编译器并非总是能知道被应用函数的变元个数。例如，e 可以是外围高阶函数的一个形参。

考虑编译函数应用 $e\ e_1\ \cdots\ e_m$ 的可能方法。显然，由函数应用所产生的指令序列必须产生 m 个闭包和一个 FUNVAL 对象。它们的指针必须放在这个函数应用的栈帧中，这些指针在栈帧中有两种安排方式，如图 13.4 所示。

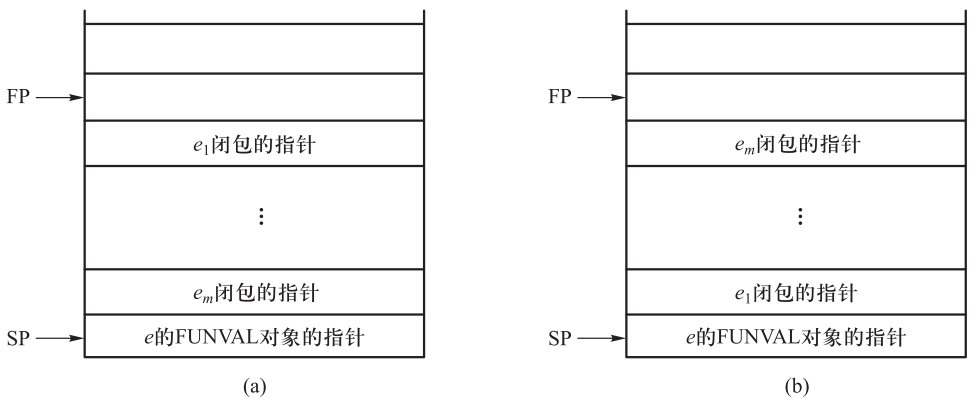


图 13.4 变元在栈帧中的可能安排

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第507页 文件编号: 422624 页面编号: 35074538444 打印人: 8685 2026-03-09 10:23:07

现在讨论这两种情况。图 13.4(a) 的存储安排允许变元和形式参数的寻址相对于 FP 的内容。但是，因为编译函数定义时 m 的值是未知的，而局部变量在栈帧中是邻近这些变元的，所以局部变量的寻址就有困难。此外，如果函数应用提供过多的变元时，也会遇到类似的问题。

图 13.4(b) 的存储安排不允许相对于 FP 的内容来作为形式参数的寻址。但是形式参数和局部变量可以相对于 SP 的内容寻址。SP 在建立新的局部变量和中间结果时是变化的，因此选择某个动态地址作为形式参数和局部变量相对寻址的基地址。这个地址可选择为正好在 e_1 闭包的指针单元的下面，把它称为 sp_0。这样，函数 $\lambda v_1 \cdots v_n.e$ 的形式参数 $v_1, \cdots, v_n$ 的相对地址必定是 $-1, -2, \cdots, -n$，而局部变量的地址必定是 $0, 1, 2, \cdots, m$。

下面的情况出现在一个函数体的处理的开始（由后面描述的 **apply** 指令产生）。PC 寄存器置到函数体指令区的开始位置，指针 GP 指向恰当的全局约束。栈包含 3 个有序的单元，分别是继续地址、$\mathrm{FP}_{\mathrm{old}}$ 和 $\mathrm{GP}_{\mathrm{old}}$，还包含 $e_m, \cdots, e_1$ 闭包的指针单元。SP 指向 e_1 闭包的指针单元。e_1 闭包的指针之上的地址是 sp_0，如图 13.5 所示。

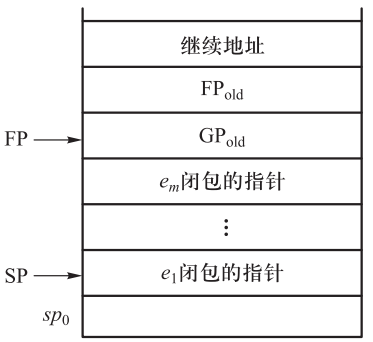


图 13.5 处理函数体之前的栈布局

如果处理函数体，由 letrec 表达式引入的局部变量必须在栈上分配空间，SP 的值相应地增加。而 SP 的当前值和 sp_0 的值之间的差，对于函数体每点来说是静态可确定的，因为已出现的新局部变量和中间结果的数目是已知的。这个差值在编译时保存在参数 sl 中，传给 P_code、B_code、V_code 和 C_code 4 个编译函数。也就是说，如果编译到函数体中的一个点 a，sl 的当前值为 sl_a，运行时执行到这一点的 SP 值为 sp_a，那么下列地址关系式成立：

$$\mathrm{sp}_a = \mathrm{sp}_0 + \mathrm{sl}_a - 1 \tag{13.1}$$

因此，生成的指令可以使用编译时确定的值 sl_a 和运行时 SP 的值 sp_a 来计算运行时的值 sp_0。形式参数可以用负的相对地址寻址，而局部变量可以用非负的相对地址寻址。

13.3.4 约束的建立

前面已讨论过，对于每个函数定义及每个表达式，自由变量集合都是静态可知的。这个集合的元素可按任意次序排列，其位置便可作为自由变量的相对地址。这些相对地址在运行时用于访问自由（全局）变量的值。这些值的地址存放在一个向量中，该向量的指针存放在堆中作为 FUNVAL 或 CLOSURE 对象的一部分；在计算闭包或函数应用时，该指针复写到 GP，即运算时可以通过 GP 去寻找该向量的元素。到目前为止，还不清楚这些全局变量的值的指针是怎样进入这样的向量的。但有一点是清楚的，当整个程序表达式被编译或被编译的程序开始执行时，自由变量集合和对应的值向量都为空。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第508页 文件编号: 422624 页面编号: 35084538442 打印人: 8685 2026-03-09 10:23:07

当函数对象或闭包被构造时，在这两种情况下，都必须将全局变量的值的指针向量和它们一同组装。因为 SFP 语言已规定为静态约束，所以刚处理的外围函数的形式参数、局部变量和全局变量都应该看成该函数定义或表达式的全局变量。在 13.3.3 小节已知道形式参数和局部变量在运行时可正确寻址。如果归纳地假定能通过 GP 访问当前的全局变量，那么就可复制所有新的全局变量的值的指针到栈中，形成一个向量，并把它作为一个堆对象的一部分。

13.4 指令集和编译

通过对问题的分析和直观介绍，现在可以一步步地描述编译和所需要的 FAM 指令。前面已提到过，共有 4 个编译函数 P_code、B_code、V_code 和 C_code，它们与所期望的生成代码执行结果的性质相对应。这些编译函数有 3 个参数：被编译的表达式 e，变量环境 β 和栈标高 sl。前面已经提到过，sl 定义了被生成的代码执行前 SP 寄存器的值和地址 sp_0 的差，形式参数和局部变量可相对于 sp_0 寻址。

13.4.1 表达式

SFP 程序表达式 e 的编译总是从 P_code 函数的一个应用开始：

P_code e = V_code e [ ] 0;
stop

因为 e 不能含任何自由变量，所以环境为空。因为栈未做任何填充，所以栈标高 sl 是 0。**stop** 指令停止机器的执行。

现在编译简单表达式，即仅由基值、运算符和 **if** 构成的表达式。考虑需要一个基值作为执行结果的编译，编译函数 B_code 适用于这种情况。

由 B_code 生成的指令列在表 13.5。假定对 SFP 语言的每个一元运算符 op_{un} 和二元运算符 op_{bin}，在 FAM 中都有对应的机器指令 $\mathbf{op}_{\mathrm{un}}$ 和 $\mathbf{op}_{\mathrm{bin}}$，于是

B_code b β sl = **ldb** b

B_code (e_1 op_{bin} e_2) β sl = B_code e_1 β sl;
B_code e_2 β (sl + 1);
$\mathbf{op}_{\mathrm{bin}}$

B_code (op_{un} e) β sl = B_code e β sl;
$\mathbf{op}_{\mathrm{un}}$

B_code (**if** e_1 **then** e_2 **else** e_3) β sl = B_code e_1 β sl;
false l_1;

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第509页 文件编号: 422624 页面编号: 35094538440 打印人: 8685 2026-03-09 10:23:07

B_code e_2 β sl;
ujmp l_2;
l_1 : B_code e_3 β sl;
l_2 :

B_code e β sl = V_code e β sl;
getbasic

表 13.5 基值、运算和分支指令

指令	含义	备注
Iab b	SP=SP+1; ST[SP]=b	装入基值
getbasic	**if** HP [ST[SP]].tag≠BASIC **then** error **fi**; ST[SP]=HP[ST[SP]].b	从堆中往栈上 装入基值
op$_{\text{bin}}$	ST[SP−1]=ST[SP−1]op_{tin} ST[SP]; SP=SP−1	二元运算
op$_{\text{un}}$	ST[SP]=op_{un} ST[SP]	一元运算
false l	**if** ST[SP]=false **then** PC=l fi; SP=SP−1	条件分支
ujmp l	PC=l	无条件分支
Idl l	SP=SP+1; ST[SP]=l	把标号放入栈中

函数 V_code 的对应情况与上面类似，增加了把作为结果的基值放入堆中并用栈顶指针指向它的指令：

V_code b β sl = B_code e β sl;
mkbasic

V_code (e_1 op_{bin} e_2) β sl = B_code (e_1 op_{bin} e_2) β sl;
mkbasic

V_code (op_{un} e) β sl = B_code (op_{un} e) β sl;
mkbasic

V_code (**if** e_1 **then** e_2 **else** e_3) β sl =
B_code e_1 β sl;
false l_1;
V_code e_2 β sl;
ujmp l_2;
l_1 : V_code e_3 β sl;
l_2 :

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第510页 文件编号: 422624 页面编号: 35104538448 打印人: 8685 2026-03-09 10:23:07

注意上面条件表达式的编译，e_1 是用 B_code 函数编译的，而 e_2 和 e_3 是用 V_code 函数编译的，因为 e_1 的值需要放在栈中，而 e_2 和 e_3 的值需要放到堆中。

13.4.2 变量的引用出现

当上下文为 V 时，变量的引用出现必须编译成访问其值；而当上下文为 C 时，变量的引用出现必须编译成访问其闭包。在前一种情况下，如果值还没有计算，必须首先从一个闭包计算这个值。

$$\begin{aligned}\text{V_code } v\ \beta\ \text{sl} = \ & \text{getvar } v\ \beta\ \text{sl};\\ & \textbf{eval}\\ \text{C_code } v\ \beta\ \text{sl} = \ & \text{getvar } v\ \beta\ \text{sl}\end{aligned}$$

必须深入考虑代码生成函数 getvar：

$$\begin{aligned}\text{getvar } v\ \beta\ sl = \ & \textbf{let } (p,i) = \beta(v)\\ & \textbf{in if } p = \text{LOC } \textbf{then pushloc } sl - i\\ & \qquad\qquad \textbf{else pushglob } i\\ & \textbf{fi}\end{aligned}$$

它为局部变量和形式参数产生 **pushloc** 指令，为全局变量产生 **pushglob** 指令。变量值的入栈指令如表 13.6 所示。

表 13.6 变量值的入栈指令

指令	含义	备注
pushloc j	SP=SP+1; ST[SP]=ST[SP−j]	把形式参数或局部变量的值的指针压入栈
pushglob j	SP=SP+1; ST[SP]=HP[GP].V[j]	把全局变量的值的指针压入栈

现在考虑对于变量 v 的调用 getvar $v\ \beta\ \text{sl}_a$，其中 v 是形式参数或局部变量。环境 β 把它约束到二元组 (LOC, i)，其中 i 是非负数（局部变量）或负数（形式参数）。于是 getvar 生成一条指令 **pushloc** $\text{sl}_a - i$。

假定在指令 **pushloc** 执行前，地址关系式（13.1）对于参数 sl_a 和 SP 的状态 sp_a 成立，即 $\text{sp}_a = \text{sp}_0 + \text{sl}_a - 1$。执行 **pushloc** $\text{sl}_a - i$ 的效果是

$$\text{ST[sp]} = \text{ST[sp} - (\text{sl}_a - i)]$$

其中 $\text{sp} = \text{sp}_a + 1$，因为 sp 在存储访问前增 1。但是仍然有

$$\text{sp} - (\text{sl}_a - i) = \text{sp}_a + 1 - \text{sl}_a + i = (\text{sp}_0 + \text{sl}_a - 1) + 1 - \text{sl}_a + i = \text{sp}_0 + i$$

于是形式参数和局部变量都能正确地装入。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第511页 文件编号: 422624 页面编号: 35114538446 打印人: 8685 2026-03-09 10:23:07

就访问全局变量而言，假定编译时的环境 β 为一个向量中的所有全局变量定义一个下标，并假定在运行时寄存器 GP 指向一个向量，该向量填充了根据这种下标定义安排的这些全局变量的指针。于是 **pushglob** 指令的效果就是从该向量中把一个全局变量的指针压入栈。

13.4.3 函数定义

函数定义可以在两种上下文（值上下文 V 和闭包上下文 C）中编译。前面已介绍过，生成的代码应该构造闭包，但是为了提高函数应用的效率，生成的代码应立即建立 FUNVAL 对象。前面已经讲过，这样的对象有 3 个成分：函数代码的起始地址、变元指针向量（初始时为空）的指针和约束向量的指针。这些指针必须在这个定义点赋值。给全局变量所置的相对地址，加上形式参数的相对地址，就构成了开始编译函数体时的环境：

V_code $(\lambda v_1 \cdots v_n.e)\ \beta$ sl=C_code $(\lambda v_1 \cdots v_n.e)\ \beta$ sl

C_code $(\lambda v_1 \cdots v_n.e)\ \beta$ sl=

```
        pushfree fr β sl;                    // 复制全局变量的值的指针
        mkvec g;
        mkvec 0;                             // 空的变元向量
        ldl l1;                              // 函数代码的地址
        mkfunval;
        ujmp l2;
l1:     targ n;                              // 测试变元个数
        V_code e([v_i ↦(LOC,−i)]_{i=1}^n [v'_j ↦(GLOB, j)]_{j=1}^n) 0;
        return n;
l2 :
```

其中，

fr=$[v'_1, \cdots, v'_g]$=list(freevar $(\lambda v_1 \cdots v_n.e)$)

pushfree $[v_1, \cdots, v_g]\ \beta$ sl=getvar $v_1\beta$ sl;

getvar $v_2\beta$ (sl+1);

$\cdots$

getvar $v_g\beta$ (sl+$g-1$)

总体来说，由 C_code 函数生成的指令序列用于构造一个 FUNVAL 对象。但与此同时，C_code 函数也必须编译函数定义，产生函数的代码。在上面的编译方案中，从标号 l_1 到 **return** n 部分是函数的代码。其余部分用于生成 FUNVAL 对象，并把该函数代码的起始地址 l_1 放到这个 FUNVAL 对象中。**ujmp** 指令用于跳过这段函数代码。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第512页 文件编号: 422624 页面编号: 35124538444 打印人: 8685 2026-03-09 10:23:07

必须注意函数全局变量的处理。它们是静态可知的，这些变量的集合是用函数 freevar 构造的。函数 list 从这个集合构造它的（无重复的）成员表。全局变量的次序也决定了它们值的指针的次序。pushfree $[v_1, \cdots, v_g]$ β sl 生成建立该向量的指令序列。它用 getvar 把变量 v_1 , $\cdots$, v_g 的值的指针压入栈，getvar 用环境中的信息进行寻址。在函数体的编译开始时，参数 sl 的值为 0，在函数体的代码开始执行前，栈的布局如图 13.5 所示。于是地址关系式（13.1）在函数体执行前保持不变，有

$$\mathrm{sp}_a = \mathrm{sp}_0 + 0 - 1$$

因为每个 getvar 产生一条指令，它的执行使 SP 增 1，所以 getvar 的参数 sl 也增 1，模拟运行时的 SP 增 1，这就保证了地址关系式（13.1）在执行时成立，因此局部变量和形式参数可以正确地寻址。

13.4.4 函数应用

为函数应用所生成的指令序列必须保证：当进入函数执行时，栈的布局应保持为编译函数定义时所假定的布局。在进入前的栈布局由图 13.5 说明。

$$\begin{array}{lll} \text{V_code } (e\ e_1 \ldots e_m)\ \beta\ \text{sl} & = & \textbf{mark}\ l; \\ e \neq e'e'' & & \text{C_code } e_m\ \beta\ (sl+3); \\ & & \ldots \\ & & \text{C_code } e_1\ \beta\ (sl+m+2); \\ & & \text{V_code } e\ \beta\ (sl+m+3); \\ & & \textbf{apply}; \\ & & l: \end{array}$$

首先 **mark** l 为这个应用建立一个新的栈帧，继续地址 l、FP 与 GP 的当前值都被保留。然后生成的指令序列为变元在堆上建立闭包，并把闭包的指针压入栈中。

注意，**mark** 的执行使 SP 加 3。如果地址关系式（13.1）在编译和执行开始时均保持，那么它在 e_m 编译前和相应的代码执行前也都保持。因为每个闭包的指针都需要一个存储单元，所以编译每个 e_i $(m \geqslant i \geqslant 1)$ 时都让参数 sl 加 1，保证了地址关系式（13.1）在该编译方案自始至终都保持。表 13.7 列出了 **mark** 指令的定义。**mark** 指令的效果由图 13.6 说明。

表 13.7 建立一个栈帧

指令	含义	备注
mark l	ST[SP+1]=l; ST[SP+2]=FP; ST[SP+3]=GP; SP=SP+3; FP=SP;	建立栈帧，保存有关地址

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第513页 文件编号: 422624 页面编号: 35134538442 打印人: 8685 2026-03-09 10:23:07

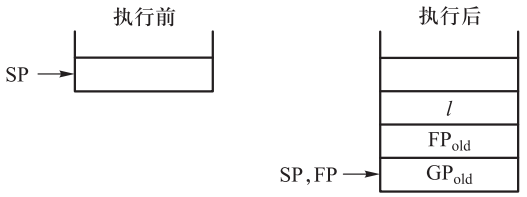


图 13.6 **mark** 指令执行前后情况

表 13.8 列出了 **apply** 指令的定义。在变元都从 FUNVAL 对象装入栈中且约束指针置好后，它跳到函数体指令的开始点。它执行后栈的布局如图 13.7 所示。

表 13.8 函数应用

指令	含义	备注
apply	**if** HP[ST[SP]].*tag* ≠ FUNVAL **then** error **fi**; **let**（FUNVAL：cf，fap，fgp）=HP[ST[SP]] **in** PC=cf; GP=fgp; SP=SP−1; **for** *i*=1 **to** size（HP[fap]. *v*）**do** SP=SP+1; ST[SP]=HP[fap].*v*[*i*] **od** **tel**	函数应用 指向约束 从 FUNVAL 对象中把变元装入褓

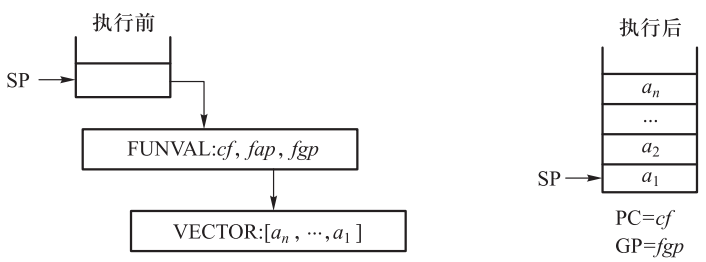


图 13.7 **apply** 指令执行前后情况

就实现函数的整个任务来说，已经讨论了下列子任务：

（1）从函数定义生成 FUNVAL 对象，并且在正确的环境下编译函数体。

（2）用 **mark** 指令为函数应用建立一个栈帧，随后的存储单元用于存放变元指针。

（3）用 **apply** 指令，在建立了 FUNVAL 对象的正确约束后，函数代码开始执行。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第514页 文件编号: 422624 页面编号: 35144538440 打印人: 8685 2026-03-09 10:23:07

下面介绍围绕函数体的代码的两条指令，即 **targ** 和 **return** 指令。**targ** 指令的定义列在表 13.9 中，解释在图 13.8 中。**targ** 测试提供给函数的变量是否不足。如果不足，它把现存的变元组装到 FUNVAL 对象中，并且释放栈帧。**return** 指令的责任是对函数应用和闭包计算结尾的处理。它的参数定义了由函数应用或闭包计算消费的变元个数。它对应两种情况：一是栈帧包含的变元指针个数和被应用函数所需要的一样多，此时将函数值复制到适当的地方，并释放当前栈帧；二是栈帧包含的变元指针个数多于被应用函数所需要的个数，此时的函数应用消费适当个数的变元，其结果是一个函数，再应用到剩余的变元，这些变元的指针仍在栈中。表达式 $(\lambda x.(\lambda yz.x+y+z)\ 3)\ 4\ 5$ 的执行会出现第二种情况，读者可以自行分析一下。

表 13.9 如果变元个数不足，形成 FUNVAL 对象

指令	含义	备注
targ n	**if** SP−FP $< n$ **then** 　　h=ST[FP−2]; 　　ST[FP−2]=new（FUNVAL：PC−1， 　　new（VECTOR：[ST[FP+1]，ST[FP+2]，⋯，ST[SP]]），GP）； 　　GP=ST[FP]; 　　SP=FP−2; 　　FP=ST[FP−1]; 　　PC=h **fi**	变元个数不足

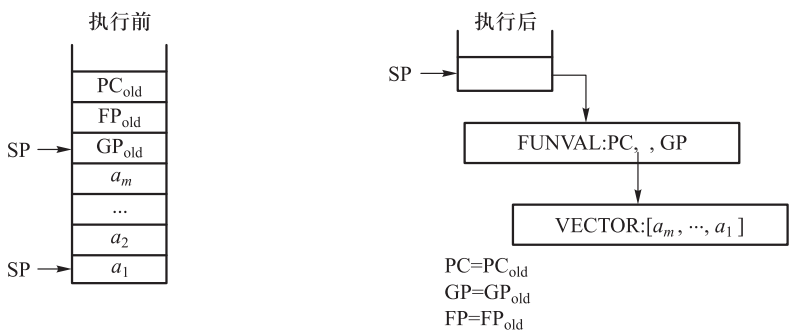


图 13.8 **targ** 指令发现变元不足（$n > m$），建立 FUNVAL 对象

return 指令的定义在表 13.10 中，它的效果说明在图 13.9 中。

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第515页 文件编号: 422624 页面编号: 35154538447 打印人: 8685 2026-03-09 10:23:07

表 13.10 函数应用和闭包计算结尾的处理

指令	含义	备注
return$n$	**if** SP=FP+1+n	结束
	then	继续地址
	PC=ST[FP−2];	
	GP=ST[FP];	
	ST[FP−2]=ST[SP];	结果
	SP=FP−2;	
	FP=ST[FP−1]	
	else	变元过多，函数的结果应用
	if HP[ST[SP]].tag≠FUNVAL **then** error **fi**;	到剩余变元
	let（FUNVAL：cf，fap，fgp）=HP[ST[SP]]	
	in PC=cf;	
	GP=fgp;	
	SP=SP−n − 1;	消费了 n 个变元
	for$i$=1 **to** size（HP[fap].v）**do**	
	SP=SP+1;	
	ST[SP]=HP[fap].v[i]	
	od	
	tel	
	fi	

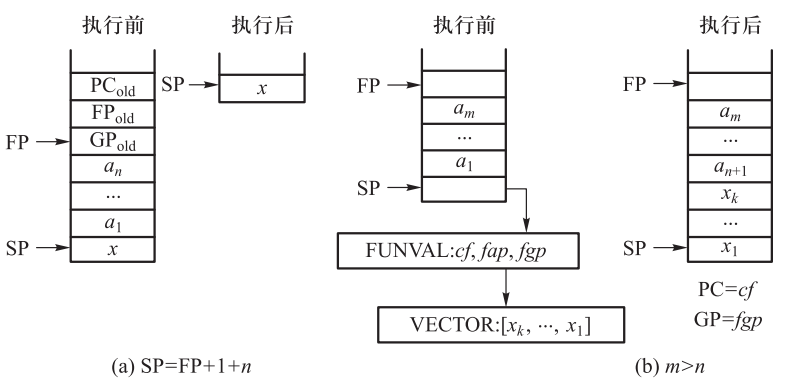


图 13.9 **return** 指令的两种情况

13.4.5 构造和计算闭包

用 C_code 函数编译表达式，其生成的代码执行时会为该表达式建立一个闭包。堆上的这个 CLOSURE 对象由两个指针组成，分别指向表达式代码和一个向量，该向量的每个元素

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第516页 文件编号: 422624 页面编号: 35164538445 打印人: 8685 2026-03-09 10:23:07

指向一个全局变量的值。因为这些值都已给出，所以一个闭包是对它的外围没有其他需求的对象。与函数的处理类似，该表达式的真正编译是在上下文 V 和一个新环境下，该环境只知道全局变量和值为零的参数 sl。与函数应用一样，闭包的计算需要建立一个栈帧，在该栈帧中，局部变量可以按前面讨论过的方案寻址。

与编译函数定义一样，编译表达式得到的代码结构包含一个较外的块（建立闭包）和一个较内的块（表达式计算的代码）。

$$
\begin{array}{rcll}
\text{C_code } e\ \beta\ \text{sl} & = & \text{pushfree fr } \beta\ \text{sl}; & \text{// 将全局变量的值压栈} \\
 & & \textbf{mkver}\ g; & \text{// 把它们做成一个向量} \\
 & & \textbf{ldl}\ l_1; & \\
 & & \textbf{mkclos}; & \\
 & & \textbf{ujmp}\ l_2; & \\
 & l_1: & \text{V_code } e\ [v_i \mapsto (\text{GLOB},\ i)]_{\text{i=1}}^{g}\ 0; & \\
 & & \textbf{update}; & \\
 & l_2: & &
\end{array}
$$

其中，

$$\text{fr} = [v_1, \cdots, v_g] = \text{list}(\text{freevar}(e))$$

利用简单的优化，基本表达式可以处理得更有效。在这种情况下，不必显式地构造闭包。

$$\text{C_code } b\ \beta\ \text{sl} = \text{V_code } b\ \beta\ \text{sl}$$

当生成的闭包的值被需要时，执行为 e 生成的代码。这由 **eval** 指令完成，如果它在栈顶得到的是一个闭包的指针，那么它就建立一个栈帧来计算该闭包。正如上下文 V 所刻画的，这个计算把结果留在堆中，并且在栈中该闭包的指针上面用一个指针指向堆中的这个结果。**update** 指令（表 13.11 和图 13.10）用这个结果去覆盖该闭包对象。这代表了 FAM 按需调用语义，因为以后再访问时无需重新计算，直接用第一次的计算结果即可。

图 13.11 说明 **eval** 指令在栈上的效果，它的定义列在表 13.12 中。

表 13.11 update 指令

指令	含义	备注
update	HP[ST[SP−4]]=HP[ST[SP]]; PC=ST[FP−2]; GP=ST[FP]; SP=FP−3; FP=ST[FP−1]	覆盖闭包

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第517页 文件编号: 422624 页面编号: 35174538443 打印人: 8685 2026-03-09 10:23:07

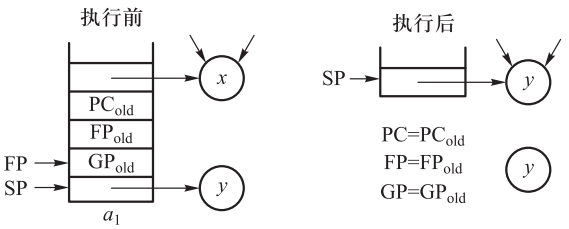


图 13.10 **update** 指令的效果

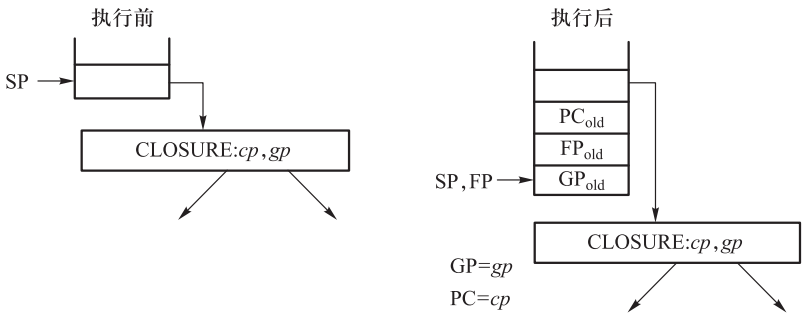


图 13.11 **eval** 指令在栈上的效果

表 13.12 在计算闭包时由 eval 指令完成栈帧的建立

指令	含义	备注
eval	**if** HP[ST[SP]].tag=CLOSURE **then** ST[SP+1]=PC; ST[SP+2]=FP; ST=[SP+3]=GP; GP=HP[ST[SP]].gp; PC=HP[ST[SP]].cp; SP=SP+3; FP=SP **fi**	计算闭包 约束指针 表达式代码的起始地址

13.4.6 letrec 表达式和局部变量

对于 **letrec** 表达式

$$\textbf{letrec}\ v_1 == e_1; \cdots ; v_n == e_n\ \textbf{in}\ e_0$$

当在上下文 V 中编译它时，必须产生下列指令和环境：

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第518页 文件编号: 422624 页面编号: 35184538441 打印人: 8685 2026-03-09 10:23:07

（1）为这 n 个表达式 $e_1,\cdots,e_n$ 建立闭包的指令序列。

（2）计算 e_0 的指令序列。

（3）根据全局变量及 $v_1,\cdots,v_n$，为 $e_0,e_1,\cdots,e_n$ 构建相同的环境。

它们由下列编译方案完成：

V_code（**letrec** v_1==e_1；$\cdots$；v_n==e_n **in** e_0）β sl=repeat n **alloc**;
C_code e_1 β' sl′;
rewrite n;
C_code e_2 β' sl′;
rewrite $n-1$;
$\cdots$
C_code e_n β' sl′;
rewrite 1;
V_code e_0 β' sl′;
slide n

其中，

$\beta'=\beta[v_i\mapsto(\text{LOC},\ \text{sl+i}-1)]_{i=1}^{n}$;
sl′=sl+n;
repeat n c=**if** n=0 **then** nocode
else c;
repeat（$n-1$）c
fi

逐步来考察这个方案。首先生成 n 个 **alloc** 指令，它在堆上建立 n 个空对象，并把它们的指针压入栈中；然后为每个表达式 e_j 生成指令序列

C_code e_j（$\beta[v_i\mapsto(\text{LOC},\ \text{sl+}i-1)]_{i=1}^{n}$）(sl+$n$);
rewrite（$n-j+1$）

该序列为 e_j 建立闭包，然后覆盖对应的空闭包对象。覆盖指令 **rewrite** m 的含义如表 13.13 所示。因此，空闭包对象必须已经存在，建立闭包的代码才可以使用这些空对象的指针，但是它不访问这些空对象的成分。

表 13.13 rewrite 指令和 slide 指令

指令	含义	备注
rewrite n	HP[ST[SP−m]]=HP[ST[SP]]; SP=SP−1	覆盖堆对象
slide n	ST[SP−m]=ST[SP]; ST=SP−m	复制结果

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第519页 文件编号: 422624 页面编号: 35194538449 打印人: 8685 2026-03-09 10:23:07

在这里，对于定义的次序必须有个假定。变量的 C_code 方案前面已经做了改进，用变量的约束作为返回而不是建立一个闭包。若不对定义排序，对于例13.11，这将是灾难性的，因为这里空闭包仍将用 **pushloc** 访问。为了避免这种情况，需对定义进行排序，确保像 $a == b$ 这样的变量命名总是在 b 的定义之后。因为循环重新命名没有意义，所以会被编译器拒绝。

【例 13.11】 表达式

letrec a = = b;
b = = 0
in ⋯;

还需确定地址关系式（13.1）在这种情况下是否仍然保持，前提是它在 **letrec** 编译前和生成的代码执行前分别保持。假设 sl 对应的开始值为 sl_0，n 次 **alloc** 会使 SP 的值增加 n，此时 e_1 编译时的参数是 $\mathrm{sl}_0 + n$。由于 **rewrite** 会释放栈的最高存储单元，因此其他表达式 $e_2, \cdots, e_n, e_0$ 的编译仍然有正确的 sl 参数。

局部变量 v_i 分配的地址是 $\mathrm{sl}_0 + i - 1$，于是 v_1 的地址是 sl_0，v_2 的地址是 $\mathrm{sl}_0 + 1$，如果该 **letrec** 是函数体中第一个 **letrec**，那么 v_1 的相对地址是 0，v_2 的相对地址是 1。因此，在函数 getvar 中用 **pushloc** 指令进行局部变量寻址也是正确的。

习　题

13.1 为下列函数写出 SFP 表达式：

（a）求两个数 a 和 b 的最大公约数的函数 gcd()。

（b）检查一个自然数是否为完全数的函数 isperfect()。如果一个数等于它的真因子的和，那么这个数就是完全数。例如，6 是完全数，因为 $6 = 1 + 2 + 3$。

13.2 确定下面 SFP 表达式中自由变量的集合和约束变量的集合：

$(\lambda x.xy)(\lambda y.y)$
$\lambda xy.z(\lambda z.z(\lambda x.y))$
$(\lambda xy.xz(yz))(\lambda x.y(\lambda y.y))$
$\lambda x.x$ + **letrec** $a == x$;
$x == f\ y$;
$y == z$
in $x + y + z$

13.3 把下列 SFP 表达式编译成 FAM 抽象机指令序列：

（a）$(\lambda x.x + 1)3$

（b）**letrec** $F == \lambda xy.xy$;
inc $== \lambda x.x + 1$

编译原理（第4版） 正文 一校一次 正文+编译原理(第4版) 一校.pdf 第520页 文件编号: 422624 页面编号: 35204538447 打印人: 8685 2026-03-09 10:23:07

in F inc 5

并说明在生成的 FAM 指令执行过程中，栈和堆是怎样变化的。

13.4 把下列 SFP 表达式编译成 FAM 代码：

letrec $fac == \lambda n.$**if** $n = 0$ **then** 1 **else** $n * \mathrm{fac}(\mathrm{dec}\ n)$;
$\mathrm{dec} == \lambda n.n - 1$
in fac 4

13.5 考虑如下形式的 SFP 程序（最外表达式是 **letrec** 表达式）：

letrec $v_1 == e_1$;
$\cdots$
$v_n == e_n$
in e

变量 v_i 可以按它们在栈开始点的绝对地址来寻址。引入一类称为 ABS 的变量，并且相应地修改编译方案，此时可以进行什么样的优化？

13.6 本习题优化下列形式的递归函数：

$f = \lambda v_1 \cdots v_n.\cdots(f\ e_1 \cdots e_n)\cdots$
in $\cdots$

如果该递归调用 $(fe_1 \cdots e_n)$ 既不是一个运算符的运算对象，又不是函数变元，那么称它为尾递归。在这种情况下，函数值的计算是通过对该函数的这个递归调用来完成的。通常，运行时碰到函数调用，一个新的栈帧必须建立。然而，对于尾递归，当前栈帧是空或者从递归调用返回后不再使用。于是，不必建立一个新的栈帧，而是在当前栈帧中计算递归调用。此时，老变元的指针直接由那些新变元的指针代替。

定义相应的指令，并且修改编译方案，使它能识别尾递归调用且用有效的方式处理它们。（提示：扩充那些代码函数，增加一个参数，该参数包含有关被编译表达式上下文的信息。）

13.7 修改 FUNVAL 对象的结构以及函数定义和函数应用的代码方案，使得在函数变元不足的情况下，变元的解包和重新打包可以避免。

参考文献

[1] CHAITIN G J. Register allocation & spilling via graph coloring[C]. In: Association for Computing Machinery. SIGPLAN Symposium on Compiler Construction, New York, NY, USA: ACM, 1982: 98–105.

[2] CLINGER W D, WAND M. Hygienic macro technology[C]. Proceedings of the ACM Program. Languages, Volume4, Lssue HOPL, 2020.

[3] HAMES L, SCHOLZ B. Nearly optimal register allocation with pbqp. In: 7th Joint Conference on Modular Programming Languages (JMLC 2006), Berlin, Heidelberg: Springer-Verlag, 2006: 346-361.

[4] HERMAN D, WAND M. A theory of hygienic macros[C]. In: Sophia Drossopoulou, editor, Programming Languages and Systems, Berlin, Heidelberg: Springer, 2008: 48-62.

[5] LATTNER C, ADVE V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation[C]. In: International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization. USA: IEEE Computer Society 2004: 75.

[6] LESK M E. Lex: a Lexical Analyzer Generator[R]. Computing science technical report. Bell Laboratories, 1975.

[7] T. J. Parr, H. G. Dietz, and W. E. Cohen. PCCTS reference manual: version 1.00. SIGPLAN Not., 27(2):88–165, feb 1992.

[8] Terence Parr and Kathleen Fisher. LL(*): the foundation of the ANTLR parser generator. In 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), page 425–436, 2011.

[9] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive LL(*) parsing: the power of dynamic analysis. In 29th ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA), page 579–598, 2014.

[10] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. ACM Trans. Program. Lang. Syst., 21(5):895–913, September 1999.

[11] Christopher Schuster, Tim Disney, and Cormac Flanagan. Macrofication: Refactoring by reverse macro expansion. In 25th European Symposium on Programming Languages and Systems (ESOP), volume 9632, page 644–671, Berlin, Heidelberg, 2016. Springer-Verlag.

郑重声明

高等教育出版社依法对本书享有专有出版权。任何未经许可的复制、销售行为均违反《中华人民共和国著作权法》，其行为人将承担相应的民事责任和行政责任；构成犯罪的，将被依法追究刑事责任。为了维护市场秩序，保护读者的合法权益，避免读者误用盗版书造成不良后果，我社将配合行政执法部门和司法机关对违法犯罪的单位和个人进行严厉打击。社会各界人士如发现上述侵权行为，希望及时举报，我社将奖励举报有功人员。

反盗版举报电话 （010）58581999 58582371

反盗版举报邮箱 dd@hep.com.cn

通信地址 北京市西城区德外大街 4 号 高等教育出版社知识产权与法律事务部

邮政编码 100120

防伪查询说明

用户购书后刮开封底防伪涂层，使用手机微信等软件扫描二维码，会跳转至防伪查询网页，获得所购图书详细信息。

防伪客服电话 （010）58582300