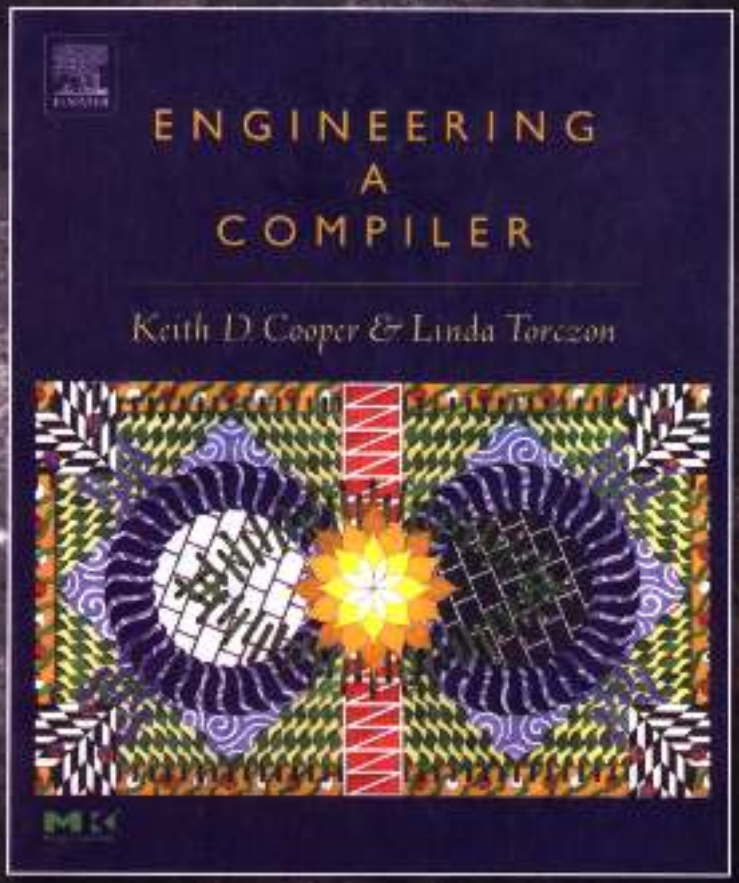
计 算 机 科 学 丛 书

编译器工程

（美） Keith D. Cooper Linda Torczon 著 冯速 译



Engineering a Compiler

机械工业出版社
China Machine Press

"Keith Cooper和Linda Torczon是顶级编译器研究人员，他们还创建了若干艺术品般的编译器。本书通过解释久经考验的技术和新算法，提供编译器设计与实现的实用方法，成功地融合了编译器的技术和艺术，是构建现代编译器的必读参考。"

——Jim Larus，微软研究院

本书深入探索编译器设计领域，涉及这个领域中的各种问题及解决方案，通过展示问题的参数和这些参数对编译器设计的影响，阐述问题的深度和可能解决方案的广度，本书介绍了实际设计中该如何权衡，以及那些微妙而高深莫测的选择对编译器的影响。

本书特点

- 集中研究编译器的后端——反映了近十几年来研究和发展的成果。
- 使用扫描和分析的成熟理论引入在优化和代码生成中起关键作用的概念。
- 介绍数据流分析、SSA形式和标量优化等优化方法。
- 传授代码生成中的现代方法：指令筛选、指令调度和寄存器分配。
- 给出程序设计语言中最能解释这些概念的实例。

作者简介

Keith D. Cooper Cooper博士，Rice大学计算机科学系教授，是Rice巨型标量编译器项目的负责人，这一项目主要研究与现代计算机的优化和代码生成相关的问题。他还是Rice大学高性能软件研究中心、计算机与信息技术学院和多媒体通信中心的成员。他开设本科生和研究生的编译器设计课程。

Linda Torczon Torczon博士，Rice大学计算机科学系研究员，曾参与Rice巨型标量编译器项目和美国国家科学基金支持的下一代软件程序赞助的网格应用开发软件项目，并是主要研究员。她还是高性能软件研究中心和洛斯阿莫斯计算机科学院的执行主管。

ISBN 7-111-17962-5

9 787111 179627

HZ BOOKS

华章图书

上架指导：计算机科学/编译器

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

投稿热线：(010) 88379604

购书热线：(010) 68995259, 68995264

读者信箱：hzjsj@hzbook.com

ISBN 7-111-17962-5/TP・4571

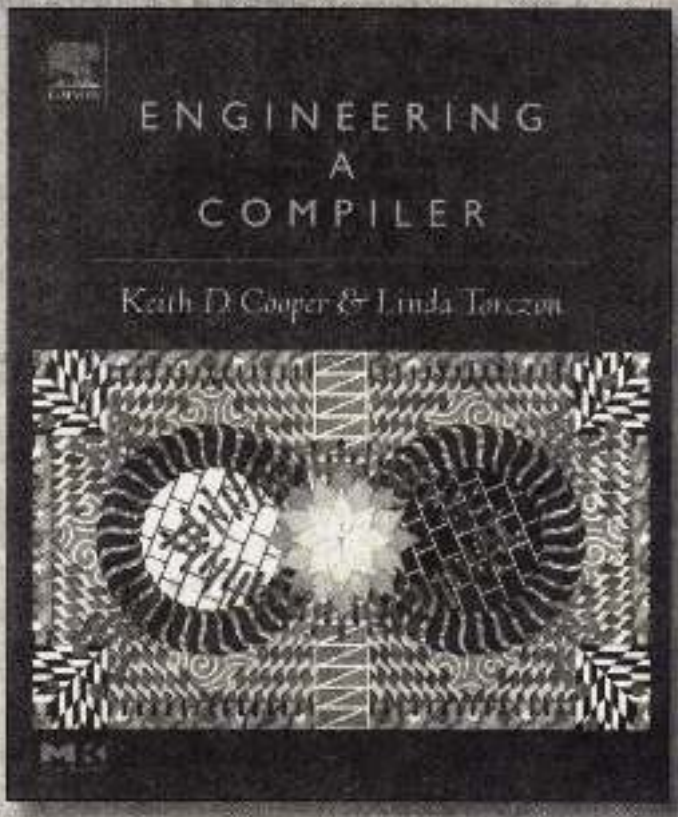
定价：68.00 元

TP314
56

计 算 机 科 学 丛

编译器工程

(美) Keith D. Cooper Linda Torczon 著 冯速 译



Engineering a Compiler

机械工业出版社
China Machine Press

本书旨在介绍编译器构造法中的艺术和科学。用大量素材向读者展示现实权衡的存在，展示这些选择的影响可能是微妙且深远的。省略由于商业、语言和编译器技术以及可用工具的变迁而变得不太重要的技术、C语言对优化和代码生成提供更深层次的处理。本书内容分为四部分。前端部分介绍扫描、语法分析、上下文相关分析的内容；基础结构部分阐述中间表示、过程抽象、代码形态为主线的知识；优化部分阐述构建编译器的中间部分——优化器所出现的问题；代码生成部分着眼于代码生成中的三个主要问题。

本书内容翔实，文笔流畅，适合作为高等院校计算机专业本科生和研究生编译课程的教材和参考书。

Keith D. Cooper, Linda Torczon: Engineering a Compiler (ISBN 1-55860-698-X).
Copyright © 2004 by Elsevier Science (USA).
Translation Copyright © 2006 by China Machine Press.
All rights reserved.

本书中文简体字版由美国Elsevier Science公司授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。
本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2004-5175

图书在版编目（CIP）数据

编译器工程 /（美）酷伯（Cooper, K. D.），（美）琳达·特克森（Torczon, L.）著；冯速译. －北京：机械工业出版社，2006.2
（计算机科学丛书）
书名原文：Engineering a Compiler
ISBN 7-111-17962-5

Ⅰ. 编…　Ⅱ. ① 酷…　② 琳…　③ 冯…　Ⅲ. 编译码器　Ⅳ. TN762

中国版本图书馆CIP数据核字（2005）第142110号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）
责任编辑：隋　曦
北京京北制版厂印刷·新华书店北京发行所发行
2006年2月第1版第1次印刷
787mm × 1092mm 1/16 · 32印张
印数：0 001 - 4 000册
定价：68.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线：（010）68326294

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall，Addison-Wesley，McGraw-Hill，Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum，Stroustrup，Kernighan，Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及庋藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业的教学度身订造的。其中许多教材均已为M. I. T.，Stanford，U.C. Berkeley，C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设

计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

电子邮件：hzjsj@hzbook.com
联系电话：（010）68995264
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

专家指导委员会

（按姓氏笔画顺序）

尤晋元	王　珊	冯博琴	史忠植	史美林
石教英	吕　建	孙玉芳	吴世忠	吴时霖
张立昂	李伟琴	李师贤	李建中	杨冬青
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
周克定	周傲英	孟小峰	岳丽华	范　明
郑国梁	施伯乐	钟玉琢	唐世渭	袁崇义
高传善	梅　宏	程　旭	程时端	谢希仁
裘宗燕	戴　葵			

对本书的赞誉

“Keith Cooper和Linda Torczon是顶级编译器研究人员，他们还创建了若干艺术品般的编译器。这本书贯穿编译器理论和实践两个领域，解释久经考验的技术和算法，并为编译器的工程化和编译器的构建提出很多切实可行的建议。《编译器工程》是对构建现代编译器必不可少的重要技术的概括和展示。”

——Jim Larus，微软研究院

“本书是对现代编译器的理论、实践和知识的完美介绍。Cooper和Torczon通过编译和计算机科学其他领域之间的优美的相互关联展示这一课题的朴素乐趣。如果你正在寻找由具有广博实践经验诠释而成的编译器构造法的完整之旅，那么这本书就是你的首选。”

——Michael D. Smith，哈佛大学

“我很高兴看到这本关于现代编译器设计的内容全面的书。作者涵盖了经典的素材，也涵盖了近15年间发展起来的重要技术，包括面向对象语言的编译、静态单一赋值、基于区域的寄存器分配以及代码调度。他们的方法把现代编译器依据的形式结构与编译器的优秀工程化所必需的实际观察完美地协调起来。”

——John Hennessy，斯坦福大学

“Cooper和Torczon做了一项完美的工作，他们集成了编译器构建理论和编译器实现的实际问题。本书所涵盖的这一领域的最新进展使得它成为向当今的本科生讲授编译器课程的理想教材或教学参考书。”

——Ken Kennedy，Rice大学

译 者 序

经过数十年的科学研究和经验积累，计算机科学已经取得了长足的发展，计算机已经普及并在各个方面得到广泛的应用。计算机技术已不再是极少数人的专利，它被众多的科学家、开发人员以及相关专业的学生和教师掌握，成为人类的有力工具。

在这样的环境下，计算机相关专业的教学也应该做相应的变革，以适应计算机科学和技术的发展。一方面，一些曾经是关键性的课程已不再是必须掌握的，一些曾经高深的技术已经唾手可得。另一方面，学生需要掌握许多新知识、新技术，而如何取舍这些新知识和新技术成为一个重要的课题。在这一问题上，本书的作者给我们提供了一个非常重要的启示。

经过20多年的研究和实践（从第一个编译器的诞生到今天已近半个世纪），编译器的前端技术已经基本成熟。特别是词法分析和语法分析的技术已经形成固定的模式，有一系列自动生成相关模块的软件，不再需要手工编写词法分析器和语法分析器。相反，代码生成和代码优化日益成为我们关心的技术，许多老技术需要我们去运用、去完善，更多的新技术需要我们去掌握、去开拓。因此，为了顺应时代的要求，同时为了学生和专业人士的各种需要，当代的编译技术的教学应该把重点放在优化和代码生成上，应该简化对编译器前端的论述。这也正是作者完成《编译器工程》这本书的首要动机。

本书一改把编译工作分解成前端和后端两个主要部分的惯例做法，把优化器作为介于前端和后端之间的第三个阶段，这样构成了典型优化编译器的工作流程。虽然计算机的性能越来越高，但人们对它的期待也越来越高，期待的提高远远超出了计算机性能的提高。因此，优化也越来越重要，并且已经成为一项必不可少的技术。

除对讲授内容做了各方面的权衡外，作者对本书的组织也非常讲究。现代编译器技术错综复杂，不存在完美的线性排列。本书通过四个部分把编译技术尽量排列成线性的顺序，以便读者轻松学习。在对编译技术进行总览之后，本书第一部分介绍编译器的前端，讲解扫描、语法分析和上下文相关分析。这里强调自动化技术和形式方法。第二部分讲解编译器中的基础结构，包括各种中间表示、过程抽象和代码形态。把贯穿这一课程的一些分散的内容组织在一起，使得本书其他部分得以依照编译器在编译时的执行顺序进行阐述。第三部分包括优化阶段，展示各种优化技术。最后的部分阐述代码生成技术，包括指令筛选、指令调度和寄存器分配的各种技术。

书中类似算法的比对是一大特色。通过比对，读者可以对类似算法有更加直观且充分的认识。另一个特点是作者把许多算法归类为不动点算法。这一方面使这些算法更加清晰，同时也使读者了解不动点算法的重要性，对算法的学习、研究有很大的帮助。

总之，顶级编译器研究人员Keith Cooper和Linda Torczon给我们带来了一本新颖、独到的编译技术参考教材。本书不仅可作为大学本科和研究生的编译技术或编译原理课程的教材，也应是计算机软件相关人员必备的一本参考书。

最后，作为讲授编译原理的教师，译者经常被学生问道：为什么要学编译原理和编译技术？我希望读者通过本书能够理解编译技术中蕴涵的思维方式，理解程序设计与开发的思维方式，理解本书的风格主题：功能、结构和精美。

译者水平有限，恳请读者不吝指正。

前　言

过去的20年间，编译器构造的实践活动发生了翻天覆地的变化。前端已变成商品组件，它们可以从可靠的供应商那里买到，或从众多自由软件的系统改写而来。与此同时，处理器变得对性能更加敏感；编译代码的实际性能很大程度上依赖于编译器针对特定处理器和系统特征进行优化的能力。这些变化影响着我们构建编译器的方式；它们也应该影响我们教授编译器构造法的方式。

今天，编译器的发展主要致力于优化和代码生成。编译器团体的新任务很可能是把代码生成器移植到新的处理器上，或修改优化遍，而不是工作于扫描器或语法分析器。为学生进入这一环境做好准备是一个真正的挑战。成功的编译器设计者必须熟知优化和代码生成中当前最实用的技术。他们还必须具有理解将出现的新技术的背景知识和洞察力。我们撰写本书（EAC, Engineering a Compiler）的目标是创作一本教科书并创建一门课程，向学生展示现代编译中的重要问题，并为他们提供解决这些问题的背景知识。

学习编译器构造法的动机

编译器构造法结合了来自计算机科学各个领域的技术。最简单地说，编译器就是一个大型计算机程序。编译器读取一个源语言的程序，并为在某个目标体系结构上执行而翻译它。作为翻译的一部分，编译器必须执行语法分析以确定输入程序是否合法。为了把输入程序映射到目标计算机的有限资源上，编译器必须操纵若干不同的名字空间，分配若干不同种类的资源，并协调多个运行时数据结构的行为。为了使输出程序有合理的性能，它必须管理功能单元中的硬件等待时间，预测执行流及内存需求，并推断出程序中的不同机器级操作的独立性和相关性。

打开一个现代优化编译器，你将发现揭示巨大求解空间的贪婪启发式搜索，识别输入中的字的确定性有穷自动机，帮助推断程序行为的不动点算法，试图推测表达式值的简单定理证明器和代数化简器，把抽象计算映射到机器级操作上的模式匹配器，用于分析数组下标的丢番图（diophantine）方程和Pressburger算术的解算器，以及诸如散列表、图算法和稀疏集合实现等经典算法和数据结构。

权衡

我们撰写EAC这本书的主要目标是创建一本用于编译器设计与实现的入门课程的教科书。EAC展示了编译器设计者需要面对的诸多问题，揭示出编译器设计者解决这些问题所用到的一些技术。EAC还提供了一系列构建现代编译器所用的实用技术的有效选择。

在EAC的选材中，为了满足学生在就业市场的需要，我们有意重新权衡了编译器构造法的入门课程的教材内容。我们减少了前端的内容而增加了优化和代码生成的内容。在后者的领域，EAC集中讨论诸如静态单一赋值形式、列表调度以及图着色寄存器分配等最实用的技术。这些课题为学生们准备了他们将来在现代商业编译器和科研编译器中所能遇到的算法。

本书还包含高年级学生或专业人士所需的内容。大多数章节都有*高级话题*一节，这一节讨论超出一般低年级课程内容所涉及的问题和技术。另外，第9章和第10章介绍比一般低年级课程所涉及的内容更深一些的数据流分析和标量优化。把这些内容包含在EAC中，使得它能够满足高年级学生或充满好奇心的学生的需要；而且专业人士也可能在他们实现某些技术时发现这些章节很有用。

途径

编译器构造法是工程设计中的一项实践。编译器设计者必须在充满不同选择，而且每一种选择都有其各自的代价、优势和复杂度的设计空间中选择一条路径。每一个决策都会对结果编译器产生影响。最终产品的质量依赖于在这一路径上的每一步明智的决策。

因此，编译器中的诸多设计决策没有惟一正确的答案。即使对于那些“著名的”和“已解决的”问题，设计和实现的细微差异都将对编译器的行为和编译器生成的代码质量产生影响。对每个决策都有很多需要考虑的问题。作为一个例子，编译器的中间表示的选择对编译器的其余部分，无论是对时间和空间的需求，还是可以运用的算法的难易程度都将产生影响。然而，这些决策通常都有不尽人意之处。第5章分析了中间表示的空间和在做出选择时应该考虑的其他问题。我们将在本书的若干地方，或是直接或在习题中间接地提及这一问题。

EAC尝试探讨设计空间问题，并讨论问题的深度以及可能的解决方案的广度。它给出解决这些问题的一些方法，以及使这些解决方案有吸引力的某些限制。学生需要理解问题和解决方案的参数，需要理解他们的决策对编译器设计其他方方面面的影响。只有这样，编译器设计者才能做出明智的决策。

哲学思想

这本教科书揭示20多年来我们通过研究、教学和实践而发展起来的关于如何构建编译器的哲学思想。例如，中间表示应该展示对最终代码至关重要的细节；这些想法导致低级中间表示的采用。值应该在寄存器中，直到分配器发现它不能把这些值保存在那里为止；这种实践引发了使用虚拟寄存器并只在无法回避时才把值存储于内存中的例子。它还增加了编译器后端中高效算法的重要性。每一个编译器都应该包含优化；它简化编译器的其他部分。

EAC不同于编译器构造法的教科书普遍接受的约定。例如，在例子中我们使用若干种不同的程序设计语言。用C语言描述名字调用参数传递没有意义，所以我们使用Algol-60。用FORTRAN语言描述尾递归没有意义，所以我们使用Scheme。这种使用多语言的方法是现实的；贯穿于读者的整个职业生涯，“未来的语言”将发生若干变化。[⊖]我们用抽象级较高的形式表示EAC中的算法。我们假设读者可以填充这些细节，并处理这些细节以适应代码运行的特定环境。

本书内容的组织

在撰写EAC时，我们的首要目标是为学生工作于真实编译器而编写一本教科书。我们已教授了10年以上本书中的内容，试验了材料的选择、深度和顺序的安排。从网站上可获得的课程内容展示了我们如何改编并在Rice大学教授学生EAC的内容。

教授现代代码生成技术的愿望使编排内容顺序的问题变得复杂。现代代码生成强烈依赖于诸如数据流分析和静态单一赋值形式等优化中的思想。这种依赖关系暗示应在覆盖后端算法之前教授优化。在讨论代码生成的内容之前覆盖优化的内容，意味着学生在尝试着改进选择语句、循环以及数组引用的代码之前是看不到这些代码的。

因为没有这些内容的完美的线性排列，所以EAC尽可能按编译器在编译时的执行顺序给出这些内容。因此，优化的内容处于前端之后却在后端之前，即使代码形态的讨论是后端的内容。章节开篇的图示起到提示这一顺序的作用。教学实践时也可以不完全按照这一顺序。

⊖ 在过去的30年间，Algol-68、APL、PL/I、Smalltalk、C、Modula-3、C++、Java，甚至是ADA都作为未来的语言一个接一个地涌现出来。

本书内容

在概述（第1章）之后，本书内容分为四个部分：

前端

第一部分的各章给出扫描、语法分析和上下文相关分析的内容。第2章介绍识别器、有穷自动机、正则表达式以及从正则表达式出发自动构建扫描器的算法。第3章使用上下文无关文法、自顶向下递归下降分析器和自底向上表驱动LR(1)分析器描述分析。第4章介绍类型系统，这是现实问题中因太复杂而无法使用上下文无关文法表示的一个例子。接着，我们给出解决这样的上下文相关问题的形式技术和专门的技术。

这些章节展示一个进展。在扫描中，自动化已经取代了手工编码。在分析中，自动化大幅度减少了程序员的工作量。在上下文相关分析中，自动化没有取代专门的手工编码方法。然而，这些专门的技术模拟一种形式技术背后的某些思想，这就是属性文法的运用。

基础结构

第二部分把贯穿这一课程的一些分散的内容组织起来。它给出在前端生成中间代码、优化这一代码以及把这一代码转换成目标机器代码所需的背景知识。

第5章描述编译器使用的各种中间表示，包括树、图、线性代码和符号表。第6章介绍编译器必须在它所生成的代码中实现的运行时抽象，包括过程、名字空间、链接约定和内存管理。第7章给出代码生成的前奏，集中讨论编译器为不同的语言结构所生成的代码形态，而不讨论生成这一代码的算法。

优化

第三部分包括构建编译器的中间部分——优化器所出现的问题。第8章通过讨论在不同作用域中出现的一个问题，引出优化的问题和技术。第9章介绍迭代数据流分析，并展示静态单一赋值形式的结构。第10章为标量优化给出一种基于效应的分类法，然后使用精选的例子探讨这一分类法。

这一内容上的划分说明，分析和优化的全面讨论可能超出了一个学期的课程，但是这样处理使本书的内容能够满足高年级和好奇心强的学生的需要。在教授本书内容时，我们在讲解第8章之后讲解代码生成。在讲解代码生成时，有必要的话我们返回第9章和第10章的特定部分。我们还使用本书的这一部分，附加若干论文节选来教授关于标量优化的进阶课程。

代码生成

最后一部分着眼于代码生成中的三个主要问题。第11章涵盖指令筛选的内容；它开始于树模式匹配，然后集中讨论窥孔风格的匹配器。第12章研究指令调度；它集中讨论列表调度及其变形。第13章描述寄存器分配；它给出局部分配和全局分配算法的较深入的论述。EAC给出的算法是学生可能发现的用于现代编译器的技术。

对于某些学生，这些章节是他们第一次要对NP完全问题给出近似解，而不是证明它等价于三可满足性问题。这些章节强调最实用的近似算法。练习为学生们提供在容易处理的例子中实践的机会。

裁剪的思想

编译器构造法是一个复杂、多层面的学科。由于编译器中信息的持续流动，对于一个问题的解决方案的选择决定后期各阶段的输入以及这些阶段改进代码的机会。对前端所做的微小变动可能隐藏优化的

机会；优化的结果对代码生成器有直接的影响（例如，改变对寄存器的需求）。编译器设计决策的复杂、相互作用的属性是这些内容通常被作为本科生顶级课程的理由之一。

这些复杂关系也出现在编译器构造法的课程中。求解技术反复出现在这一课程中。不动点算法在扫描器和语法分析器的构造中起着至关重要的作用。它们是支持优化和代码生成分析的主要工具。在扫描中出现了有穷自动机。它们在LR(1) 表驱动构造法以及指令筛选的模式匹配器中起着关键性的作用。通过强调这些常用的技术，EAC使学生熟悉它们。因此，当学生遇到第8章中的迭代数据流算法时，它只是另外一个不动点算法，因为它已是学生熟悉的内容。同样地，我们把对第12章和第13章中的局部算法到区域算法或到全局算法转换的讨论作为第8章关于优化作用域的补充。

组织课程

编译器构造法课程在具体应用的环境中为教师和学生提供揭示所有这些问题的机会，所有有编译器构造法课程知识背景的学生都能很好地理解它的基本功能。在某些课程计划中，编译器构造法课程把面向实践项目的其他课程的概念结合到一起，成为高年级学生的顶级课程。这一课程的学生可以为简单语言编写完整的编译器，或者把对一个语言的新特性的支持加到现存的编译器中，例如加到GCC或IA-64的ORC编译器中。这一课程可以严格按本书的组织以线性顺序展开教学。

如果课程计划中其他课程为学生提供大项目的实践，那么教师可以把编译器构造法课程局限在算法及其实现上。在这样的课程中，实验可以集中于真正难题的抽象实例，诸如寄存器分配和调度。这一课程可以为了满足实验的需要在内容上进行筛选，并调整授课的顺序。例如，所有学过汇编语言程序设计的学生都能够写出直线代码的寄存器分配器。我们经常把简单的寄存器分配器作为第一个实验。

无论是哪种情况，这一课程都应该从其他课程中汲取素材。和它明显相关的课程包括计算机组成原理和汇编语言程序设计、操作系统、计算机体系结构、算法以及形式语言。尽管编译器构造法与其他课程的联系并不明显，但是它们并非不重要。例如，第7章所讨论的字符拷贝在包括网络协议、文件服务器以及网络服务器在内的应用性能中起着重要的作用。第2章所开发的扫描技术在从文本编辑到URL过滤等方面均有应用。第13章中的自底向上局部寄存器分配器作为最优离线页替换算法的兄弟而得到公认。

编译器构造法中的艺术和科学

编译器构造法这一门学问中既有理论应用于实践的令人惊叹的成功故事，又有让我们束手无策的无奈。就其成功的一面，我们可以通过把正则语言的理论运用到识别器的自动构建中来构建现代扫描器。LR分析器使用相同的技术执行操纵移入归约分析器的句柄识别。数据流分析（及其兄弟）使用既实用又聪明的方式把格论运用于程序的分析。用于代码生成的近似算法对许多真正难题给出优秀的解决方案。

另一方面，编译器构造法也揭示了不存在优秀解决方案的某些复杂问题。现代超标量计算机的编译器后端必须对两个或多个相互作用的NP完全问题（指令调度、寄存器分配以及指令和数据放置）近似求解。然而，这些NP完全问题接近于诸如表达式的代数重组的问题（例如，参见图7-1）。表达式的代数重组问题有上百种解决方案；使情况变得更糟的是，理想的解决方案依赖于编译器运用的其他转换。尽管编译器尝试去解决这些问题（或近似的解决方案），但是它必须在合理的时间内运行并消耗不太多的空间。因此，现代超标量计算机的优秀编译器必须把理论、实践知识、工程学和经验艺术性地结合起来。

在本书中，我们尝试着传播编译器构造法中的艺术和科学。EAC包含大量的素材向读者展示现实权衡的存在，以及展示这些选择的影响可能是微妙且深远的。EAC省略那些由于商业、语言和编译器技术，或者可用工具的变迁而变得不太重要的技术。另外，EAC对优化和代码生成提供更深层次的处理。

关于原书封面

本书的封面是名画“方舟的登陆（The Landing of the Ark）”的一部分，它装饰着Rice大学的邓肯礼堂（Duncan Hall）的天花板（参看下面这幅画）。邓肯礼堂和它的天花板都是由英国设计师John Outram设计的。邓肯礼堂是Outram作为一名设计者在其职业生涯中所开拓的建筑学、装潢学以及哲学主题的外在表现。这一礼堂的天花板装潢对这一建筑物的设计方案起着重要的作用。Outram把一组寓意深刻的造物神话铭刻在天花板上。通过用巨大、色彩鲜明的寓言画面表达他的思想，Outram创造了一个标志，它告知所有进入这一礼堂的参观者这一建筑物的与众不同。

把这一相同的标志用于我们的《编译器工程》一书的封面，旨在表示这一著作所包含的重要思想是作者的学科的核心。如同Outram的建筑物一样，本书是作者职业生涯中所发展起来的知识主题的巅峰；如同Outram的装潢设计方案一样，本书是交流思想的手段；如同Outram的天花板一样，本书以新方式呈现重要的思想。

通过把编译器的设计与结构与这一建筑物的设计和结构结合在一起，我们意图展示这两个不同活动中的很多相似之处。与Outram的长期讨论把我们引入建筑的维特鲁威（Vitruvian）风格上来：实用、稳固和喜庆。这些思想被运用于很多建筑物。编译器结构也是一样，而这正是本书的一贯主题：功能、结构和精美。功能方面：生成不正确代码的编译器是无用的。结构方面：工程的细节决定了编译器的效率和稳健性。精美方面：设计良好的编译器可以是一件优美的艺术品，其中的算法和数据结构流畅地从一个遍流向另一个遍。



我们很荣幸能用John Outram的作品装饰本书的封面。

邓肯礼堂的天花板是一个有趣的技术艺术品。Outram把原始设计画在一张纸上。这幅画被拍摄下来，并用1200dpi扫描成大约750MB的数据。照片被放大成234个2×8英尺[⊖]的板块，制作成一张52×72英尺的图片。使用12dpi的丙烯酸喷墨打印机把这些板块印刷到一片片穿孔乙烯基上。这些穿孔乙烯基被装到2×8英尺的吸声瓦上，而吸声瓦被它悬挂到拱顶的框架上。

致谢

很多人为EAC的形式、内容及说明提出了很有价值的反馈。他们包括L. Almagor、Saman Amarasinghe、Thomas Ball、Preston Briggs、Corky Cartwright、Carolyn Cooper、Christine Cooper、Anshuman Das Gupta、Jason Eckhardt、Stephan Ellner、Mike Fagan、Matthias Felleisen、Alex Grosul、John Greiner、Dan Grossman、Timothy Harvey、James Larus、Ken Kennedy、Shriram Krishnamurthy、Ursula Kuterbach、Robert Morgan、Guilherme Ottoni、Vishal Patel、Norm Ramsey、Steve Reeves、Martin Rinard、L.Taylor Simpson、Reid Tatge、Dan Wallach、Todd Waterman和Christian Westbrook。

Steve Carr担当练习的编辑；他与一个小组共同编写了本书的练习。他的小组成员包括Chen Ding、Rodolfo Jardim de Azevedo、Zhiyuan Li、Guilherme Ottoni和Sandra Rigo。Aaron Smith、Ben Hardekopf和Paul A. Navratil检查了练习的答案。本书的手稿经过了很多遍的检查和修改：Saman Amarasinghe、Guido Araujo、Preston Briggs、Steve Carr、James Larus、Gloria Melara、Kathryn McKinley、Robert Morgan、Thomas Murtagh、Gordon Novak、Santosh Pande、Allan Porterfield、

⊖ 1英尺=0.025米。

Martin Rinard、Mark Roberts、Michael Smith和Hongwei Xi担当了审稿人。Wilson Hsieh、Jurek Jaromczyk、Tevfik Bultan、ChauWen Tseng、Mahmut Kandemir和Zhiyuan Li作为练习小组和检查小组的成员在课堂上试用了这本书。他们的工作改进了这本书，使它的风格和内容得到改善，从而更精确。

Michael Scott和Steve Muchnick反复检查了整个手稿。他们付出的细心和耐心以及他们提出的建议都使本书得到极大的改进。

Morgan Kaufmann为本书所组建的编辑、出版小组的工作也极其完美。由Denise Penrose和Yonie Overton所领导的这个小组成员包括John Hammet、Carol Leyba、Rebecca Evans、Steve Rath、Emilia Thiuri和Lauren Wheelock。Dartmouth出版公司的小组负责重新绘制本书的插图；这些图的布局是由Integra的小组完成的。作为作者，我们无法找到比他们更灵活、更富于技术及更耐心的人。他们的建议和观点大大改进了本书，他们的敬业精神使本书的出版得以顺利进行。

最后，在历时五年的时间里，很多人向我们提供了很多充满智慧和热情的支持。首先，我们的家庭和Rice大学的同事一路上给予我们鼓励。另外，Regina Brooks和Janice Bordeaux鼓励我们开始这一艰难工作。Denise Penrose和Yonie Overton使得我们能够完成这一艰巨的工作。我们由衷感谢Kathryn O'Brien以他一贯的幽默风格承担这一项目实施中的所有行政工作，并保证这一项目顺利进行。最后，Ken Kennedy和Scott Warren促使我们形成了关于程序设计、语言问题和编译的思想。

目　录

出版者的话
专家指导委员会
对本书的赞誉
译者序
前言

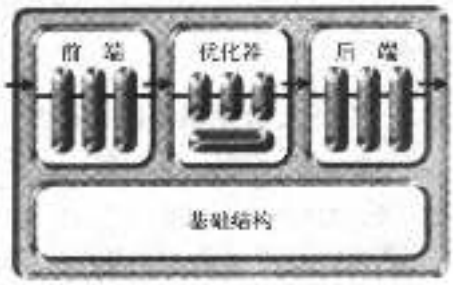
第1章　编译总览 ……1
1.1　概述 ……1
1.2　为什么研究编译器构造法 ……2
1.3　编译的基本原则 ……2
1.4　编译器的结构 ……3
1.5　翻译综述 ……5
1.5.1　理解输入 ……5
1.5.2　创建和维护运行时环境 ……7
1.5.3　改进代码 ……8
1.5.4　生成输出程序 ……9
1.6　编译器应有的性质 ……13
1.7　概括和展望 ……14
本章注释 ……15
第2章　扫描 ……16
2.1　概述 ……16
2.2　识别字 ……17
2.2.1　识别器的形式 ……18
2.2.2　识别更复杂的字 ……19
2.2.3　扫描器的自动构建 ……20
2.3　正则表达式 ……21
2.3.1　正则表达式的定义 ……22
2.3.2　例子 ……23
2.3.3　RE的性质 ……25
2.4　从正则表达式到扫描器以及从扫描器到正则表达式 ……26
2.4.1　非确定性有穷自动机 ……26
2.4.2　正则表达式到NFA：Thompson构造法 ……28
2.4.3　NFA到DFA：子集构造法 ……29
2.4.4　DFA到最小DFA：Hopcroft算法 ……32
2.4.5　DFA到正则表达式 ……34
2.4.6　将DFA作为识别器 ……35
2.5　实现扫描器 ……35
2.5.1　表驱动扫描器 ……35
2.5.2　直接编码扫描器 ……37
2.5.3　处理关键字 ……37
2.5.4　描述动作 ……38
2.6　高级话题 ……38
2.7　概括和展望 ……41
本章注释 ……41
第3章　语法分析 ……42
3.1　概述 ……42
3.2　表示语法 ……42
3.2.1　上下文无关文法 ……43
3.2.2　构造句子 ……45
3.2.3　使用结构描述优先权 ……48
3.2.4　发现特定派生 ……49
3.2.5　上下文无关文法与正则表达式的对比 ……50
3.3　自顶向下分析 ……51
3.3.1　例子 ……52
3.3.2　自顶向下分析的复杂因素 ……54
3.3.3　消除左递归 ……54
3.3.4　消除回溯 ……55
3.3.5　自顶向下递归下降分析器 ……59
3.4　自底向上分析 ……62
3.4.1　移入归约分析 ……63
3.4.2　发现句柄 ……65
3.4.3　LR(1)分析器 ……67
3.5　构建LR(1)表格 ……70
3.5.1　LR(1)项目 ……71

3.5.2 构造规范集合 ……72
3.5.3 填充表格 ……74
3.5.4 表构造法的出错 ……76
3.6 实践中的问题 ……78
3.6.1 错误恢复 ……79
3.6.2 一元操作符 ……79
3.6.3 处理上下文相关歧义性 ……80
3.6.4 左递归与右递归 ……81
3.7 高级话题 ……82
3.7.1 优化文法 ……83
3.7.2 减小LR(1)表格的大小 ……84
3.8 概括和展望 ……86
本章注释 ……87
第4章 上下文相关分析 ……88
4.1 概述 ……88
4.2 类型系统概述 ……89
4.2.1 类型系统的目的 ……89
4.2.2 类型系统的组成部分 ……93
4.3 属性文法框架 ……99
4.3.1 评估方法 ……101
4.3.2 循环性 ……102
4.3.3 扩展例子 ……102
4.3.4 属性文法方法的问题 ……107
4.4 特定语法制导翻译 ……109
4.4.1 实现特定语法制导翻译 ……110
4.4.2 例子 ……111
4.5 高级话题 ……117
4.5.1 类型推断中的较难问题 ……117
4.5.2 更改结合性 ……118
4.6 概括和展望 ……119
本章注释 ……120
第5章 中间表示 ……121
5.1 概述 ……121
5.2 分类法 ……121
5.3 图示IR ……123
5.3.1 与语法相关的树 ……123
5.3.2 图 ……126
5.4 线性IR ……128
5.4.1 栈机器代码 ……129
5.4.2 三地址代码 ……129
5.4.3 表示线性代码 ……130
5.5 静态单赋值形式 ……131
5.6 把值映射到名字 ……133
5.6.1 命名临时值 ……134
5.6.2 内存模型 ……135
5.7 符号表 ……137
5.7.1 散列表 ……138
5.7.2 构建符号表 ……139
5.7.3 处理嵌套作用域 ……139
5.7.4 符号表的多种运用 ……142
5.8 概括和展望 ……143
本章注释 ……143
第6章 过程抽象 ……144
6.1 概述 ……144
6.2 控制抽象 ……145
6.3 名字空间 ……147
6.3.1 类Algol语言的名字空间 ……147
6.3.2 活动记录 ……150
6.3.3 面向对象语言的名字空间 ……153
6.4 过程间的值传递 ……158
6.4.1 参数传递 ……158
6.4.2 返回值 ……160
6.5 建立可寻址性 ……161
6.5.1 平凡基地址 ……161
6.5.2 其他过程的局部变量 ……162
6.6 标准链接 ……164
6.7 管理内存 ……167
6.7.1 内存布局 ……167
6.7.2 堆管理算法 ……170
6.7.3 隐式释放 ……172
6.8 概括和展望 ……175
本章注释 ……176
第7章 代码形态 ……177
7.1 概述 ……177
7.2 指定存储位置 ……178
7.2.1 布局数据区 ……178
7.2.2 把值保存在寄存器中 ……179
7.2.3 机器特性 ……180

7.3 算术操作符 ……180
7.3.1 降低对寄存器的要求 ……181
7.3.2 存取参数值 ……183
7.3.3 表达式中的函数调用 ……184
7.3.4 其他算术操作符 ……184
7.3.5 混合型表达式 ……184
7.3.6 作为操作符的赋值 ……184
7.3.7 交换性、结合性以及数系 ……185
7.4 布尔操作符和关系操作符 ……185
7.4.1 表示 ……186
7.4.2 关系操作的硬件支持 ……189
7.4.3 选择表示 ……191
7.5 存储和存取数组 ……191
7.5.1 引用向量元素 ……191
7.5.2 数组存储布局 ……193
7.5.3 引用数组元素 ……194
7.5.4 范围检测 ……197
7.6 字符串 ……197
7.6.1 串的表示 ……198
7.6.2 串赋值 ……198
7.6.3 串连接 ……200
7.6.4 串长 ……201
7.7 结构引用 ……201
7.7.1 装入和存储匿名值 ……201
7.7.2 理解结构布局 ……202
7.7.3 结构数组 ……203
7.7.4 联合和运行时标签 ……204
7.8 控制流结构 ……204
7.8.1 条件执行 ……205
7.8.2 循环和迭代 ……207
7.8.3 选择语句 ……210
7.8.4 中断语句 ……211
7.9 过程调用 ……212
7.9.1 评估实参 ……212
7.9.2 过程值参数 ……213
7.9.3 保存和恢复寄存器 ……213
7.9.4 叶过程的优化 ……214
7.10 实现面向对象语言 ……214
7.10.1 单一类，无继承 ……214
7.10.2 单一继承 ……216
7.11 概括和展望 ……219
本章注释 ……219
第8章 代码优化概述 ……221
8.1 概述 ……221
8.2 背景知识 ……222
8.2.1 LINPACK的一个例子 ……223
8.2.2 优化的各种考虑 ……224
8.2.3 优化的机会 ……226
8.3 冗余表达式 ……226
8.3.1 构建有向无环图 ……227
8.3.2 值编号 ……229
8.3.3 冗余消除的经验 ……232
8.4 优化作用域 ……233
8.4.1 局部方法 ……233
8.4.2 超局部方法 ……233
8.4.3 区域方法 ……234
8.4.4 全局方法 ……235
8.4.5 完整程序方法 ……235
8.5 比基本块大的区域上的值编号 ……235
8.5.1 超局部值编号 ……235
8.5.2 基于支配者的值编号 ……238
8.6 全局冗余消除 ……241
8.6.1 计算可用表达式 ……242
8.6.2 取代冗余计算 ……243
8.6.3 结合上述两个阶段 ……244
8.7 高级话题 ……245
8.7.1 通过复制增加上下文 ……246
8.7.2 内联替换 ……247
8.8 概括和展望 ……248
本章注释 ……249
第9章 数据流分析 ……250
9.1 概述 ……250
9.2 迭代数据流分析 ……251
9.2.1 活变量 ……251
9.2.2 迭代LIVEOUT解算器的性质 ……256
9.2.3 数据流分析的局限性 ……258
9.2.4 数据流分析的其他问题 ……260
9.3 静态单一赋值形式 ……262

9.3.1 构建SSA形式的简单方法……263
9.3.2 支配……263
9.3.3 放置φ函数……267
9.3.4 重命名……269
9.3.5 从SSA形式重新构造可执行代码……273
9.4 高级话题……277
9.4.1 结构数据流算法和可约性……277
9.4.2 过程间分析……279
9.5 概括和展望……281
本章注释……282
第10章 标量优化……283
10.1 概述……283
10.2 转换分类……284
10.2.1 机器无关转换……284
10.2.2 机器相关转换……286
10.3 优化示例……286
10.3.1 消除无用和不可达代码……286
10.3.2 代码移动……291
10.3.3 特化……297
10.3.4 激活其他转换……299
10.3.5 冗余消除……301
10.4 高级话题……302
10.4.1 优化组合……302
10.4.2 强度减弱……304
10.4.3 优化的其他目标……311
10.4.4 优化序列的选择……312
10.5 概括和展望……312
本章注释……313
第11章 指令筛选……315
11.1 概述……315
11.2 简单的树遍历方案……318
11.3 通过树模式匹配实现指令筛选……322
11.3.1 重写规则……323
11.3.2 寻找铺盖……326
11.3.3 工具……328
11.4 指令筛选与窥孔优化……329
11.4.1 窥孔优化……329
11.4.2 窥孔转换器……333
11.5 高级话题……334
11.5.1 学习窥孔模式……334
11.5.2 生成指令序列……335
11.6 概括和展望……336
本章注释……336
第12章 指令调度……338
12.1 概述……338
12.2 指令调度问题……339
12.2.1 调度质量的其他度量……342
12.2.2 使调度困难的因素……342
12.3 列表调度……343
12.3.1 效率……345
12.3.2 其他的优先度方案……346
12.3.3 前向与向后列表调度……346
12.3.4 使用列表调度的原因……348
12.4 高级话题……349
12.4.1 区域调度……349
12.4.2 上下文的复制……354
12.5 概括和展望……356
本章注释……356
第13章 寄存器分配……357
13.1 概述……357
13.2 背景问题……357
13.2.1 内存与寄存器……358
13.2.2 分配与赋值……358
13.2.3 寄存器分类……359
13.3 局部寄存器分配和赋值……359
13.3.1 自顶向下的局部寄存器分配……360
13.3.2 自底向上的局部寄存器分配……360
13.4 移到超出单一块的范围……363
13.4.1 活性和存活范围……363
13.4.2 块边界处的复杂性……364
13.5 全局寄存器分配和赋值……365
13.5.1 发现全局存活范围……366
13.5.2 评估全局溢出代价……367
13.5.3 冲突和冲突图……368
13.5.4 自顶向下着色……370
13.5.5 自底向上着色……371
13.5.6 接合存活范围以降低度……372

13.5.7 回顾与比较 ……373
13.5.8 在冲突图中编码机器限制 ……374
13.6 高级话题 ……375
13.6.1 图着色分配的若干变形 ……375
13.6.2 寄存器分配中较困难的问题 ……377
13.7 概括和展望 ……379
本章注释 ……379
附录A ILOC ……380
A.1 概述 ……380
A.2 命名约定 ……381
A.3 各种操作 ……382
A.3.1 算术 ……382
A.3.2 移位 ……382
A.3.3 内存操作 ……383
A.3.4 寄存器到寄存器的拷贝操作 ……383
A.4 例子 ……384
A.5 控制流操作 ……384
A.5.1 其他比较和分支语法 ……385
A.5.2 跳转 ……386
A.6 SSA形式的表示 ……386
附录B 数据结构 ……389
B.1 概述 ……389
B.2 表示集合 ……389
B.2.1 把集合表示成有序表 ……390
B.2.2 把集合表示成位向量 ……391
B.2.3 表示稀疏集合 ……391
B.3 实现中间表示 ……392
B.3.1 图式中间表示 ……392
B.3.2 线性中间形式 ……395
B.4 实现散列表 ……396
B.4.1 选择散列函数 ……397
B.4.2 开放散列 ……398
B.4.3 开放寻址 ……399
B.4.4 存储符号记录 ……400
B.4.5 增加嵌套词法作用域 ……401
B.5 一个灵活的符号表设计 ……403
附录注释 ……404
参考文献 ……405
练习 ……424
索引 ……452



第1章 编译总览

1.1 概述

计算机在日常生活中的作用在逐年提高。现代微处理器应用于汽车、微波炉、洗衣机、移动电话、GPS导航系统、计算机游戏和个人电脑等各种设备。这些计算机运行用某种“程序设计语言”编写的一系列操作，也就是程序来完成它们的工作。与逐步进化而来且带有歧义性的自然语言相比，程序设计语言是一种具有数学特征的、有明确意义的形式语言。程序设计语言是以高表达能力、简洁、清晰为目的而设计的，其目的是描述特定的计算，记录执行特定的计算任务或生成特定的计算结果的一系列动作。

程序在执行之前，必须被翻译成目标计算机上定义的一系列操作。这个翻译工作是由一个被称为编译器（compiler）的特殊程序来完成的。编译器以一个可执行程序的描述作为输入，以另外一个等价的可执行程序的描述作为输出。当然，如果它在输入程序中发现了错误，那么编译器将生成一组适当的错误消息。把编译器看成一个黑盒子，如右图所示。



通常，编译器接受的“源”语言是一种程序设计语言，例如FORTRAN、C、C++、Ada、Java或ML等。而“目标”语言通常是某个计算机系统的指令集合。

某些编译器生成用成熟的程序设计语言编写的目标程序，而不是用某些计算机的汇编语言编写的目标程序。这些编译器生成的程序在计算机上直接运行之前还需要进一步翻译。很多研究用的编译器生成C语言的程序作为输出，因为大多数计算机都有C语言的编译器，这使目标程序在所有这些系统中都可运行。其代价就是一次额外的编译以生成最终的目标程序。以程序设计语言为目标语言而不是以计算机的指令集合为目标语言的编译器通常称为源程序到源程序编译器（source-to-source translator）。

可以把很多其他系统看成编译器。例如，生成PostScript文档的排版程序就可以看成是一种编译器，它以对文档外观的描述为输入，而输出一个PostScript文档。PostScript是描述图像的一种语言。因为排版程序以一个可执行描述为输入而且生成另外一个可执行描述，因此它是一个编译器。

把PostScript转换成像素的代码通常是解释器（interpreter）而不是编译器。解释器以一个可执行描述为输入，以执行这一可执行描述的结果为输出。



解释器也可用于实现程序设计语言。对某些语言，例如Perl、Scheme和APL来说，解释器比编译器更常用。

解释器和编译器有很多共同之处。它们执行很多相同的任务。二者都检查输入程序并确定这个程序是否是一个有效程序；二者都建立一个内部模型来刻画输入程序的结构和含义；二者都决定在执行期间值的存放位置。然而，二者的行为完全不同：解释器解释代码来生成结果，而编译器是将放通过运行来

2 生成结果的翻译程序。本书集中讨论在构建编译器中可能遇到的各种问题。然而，解释器设计者也可以找到许多相关内容。

1.2 为什么研究编译器构造法

编译器是一个大型、复杂的程序。它通常包含成百上千乃至上百万行代码。编译器的很多部分具有很复杂的相互作用。为编译器的一个部分所做的设计方案对它的其他部分会产生重要的影响。因此，编译器的设计与实现是软件工程学的一类重要的实践。

一个好的编译器就是计算机科学的一个缩影。它实际运用大量的技术，包括贪婪算法（寄存器分配）、启发式搜索技术（列表调度）、图形算法（死码消除）、动态规划（指令筛选）、有穷自动机和下推自动机（扫描和语法分析）以及不动点算法（数据流分析）。它处理诸如动态分配、同步、命名、局部化、存储器分层管理和管道调度等问题。很少有软件系统为了实现一个目标把如此繁多而复杂的任务结合起来。科研编译器的内部结构为软件工程提供实践经验，而这些实践经验是很难从那些小而不太复杂的系统中得到的。

编译器是计算机科学的核心活动的基础：使用计算机解决问题时我们需要编译器。大多数软件都需要被编译；编译过程的正确性和结果代码的高效性直接影响着我们构建大型系统的能力。大多数学生不满足于研读这些思想；这些思想必须加以实现才能体现出它们的价值。因此学习编译器构造法是计算机科学教育的一个重要组成部分。

编译器构造的学问中包含很多成功的故事。形式语言理论已引发出自动生成扫描器和语法分析器的工具。在文本检索、网站过滤、文字处理和命令语言解释器中，同样的工具和技术也找到了用武之地。类型检验和静态分析运用了格论和数论以及其他数学分支来理解和改进这些程序。代码生成器使用树模式匹配算法、分析、动态规划和文本匹配来实现指令筛选的自动化。

与此同时，编译器构造的历史也经历了一段艰难的历程。编译含有相当棘手的问题。设计一种高级、
3 通用的中间表示的尝试在复杂性的面前步履维艰。这一过程的若干部分现在还不能自动化：至少自动化技术还没有取代手工编码。在很多情况下，我们不得不求助特殊的方法。指令调度的主导方法就是带有若干层次的决胜探索的贪婪算法。虽然编译器显然可以利用交换律和结合律来改进代码，但是，尝试这样做的大多数编译器只是单纯地按某个标准的顺序重组表达式。

为了构建一个成功的编译器，我们需要把算法、工程学的洞察力以及周密的计划有机地结合在一起。好的编译器给出这些棘手问题的综合性的解答方案。它们强调效率，包括实现编译器自身的效率以及编译器所生成的代码的效率。它们拥有揭示恰当细节的内部数据结构和知识表示：这些细节足以实现高度优化，但又不会使编译纠缠于细节。

1.3 编译的基本原则

编译器是工程对象，是具有独特目标的大型软件系统。构建一个编译器需要大量的设计决策，每个设计决策都对最终的编译器产生影响。虽然编译器的很多设计决策可以使用不同的解决方案，但是有两个基本原则应坚持。编译器必须遵守的第一个原则是不违背原义（inviolable）。

编译器必须保持被编译程序的含义不变。

正确性是程序设计中的一个基础问题。编译器必须忠实地实现它的输入程序的“含义”来保证其正确性。这一原则是编译器设计者与编译器用户之间的契约的核心。如果编译器能够自由处理输入程序的含义的话，那么又为什么不能仅仅生成一个`nop`或`return`呢？如果允许错误翻译的话，那么我们为什么

还要花力气去把它改正过来呢?

编译器必须遵守的第二个原则是实用性。

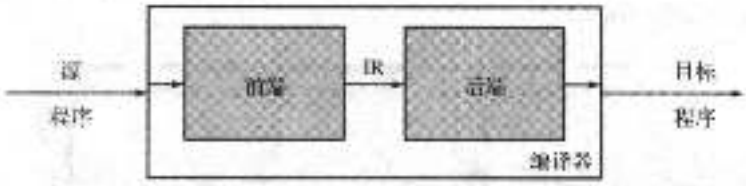
> 编译器必须用某种明确的方式改进输入程序。

传统编译器使它的输入程序可以在某个目标计算机上直接运行来完成对它的改进。其他"编译器"却以不同的方式改进它们的输入程序。例如，tpic是一个读入用图形语言pic编写的图画描述，并把它 4
转化成TEX的一个程序；它的"改进"在于TEX具有更广泛的可用性和一般性。某些编译器生成的输出程序与输入程序属于同一种语言；我们称这些编译器为源程序到源程序翻译器。一般地，这样的系统重写输入程序，使得当这一程序最终被翻译成某个目标计算机的代码时可以有所改进。如果一个编译器不能以某种方式改进代码的话，那么谁又会调用它呢?

1.4 编译器的结构

编译器是一个大型而复杂的软件系统。编译器社区从1955年开始构筑编译器；经过多年的研究，我们已经获得了很多关于如何构筑编译器的经验。初期，我们把编译器描述成一个能够把源程序翻译成目标程序的单个盒子。当然，实际的编译器要比这样的简单图示复杂得多。

正如这样的单盒子模型所示，编译器必须既能够理解需要编译的源程序，又能够把它的功能映射到目标机器上。这两个任务的不同性质表明我们可以进行分工，从而把编译工作分解成两个主要部分：前端（front end）和后端（back end）。其图示如下：



前端致力于理解源语言程序，而后端致力于把程序映射到目标机器上。这种分解对于编译器的设计与实现有几个重要的意义。

前端必须以某种结构对源程序的信息进行编码以供后端使用。这一中间表示（intermediate representation，IR）成为编译器所翻译的代码的确定表示。在编译过程中的每一点，编译器都有一个确定的表示。事实上，在编译的进程中，它也许使用若干不同的IR，但是在每一点，表示将是确定的IR。我们把这个确定的IR作为编译器各自独立阶段间传递的程序版本，就如同在前面给出的图示中从前端传 5
递到后端的IR那样。

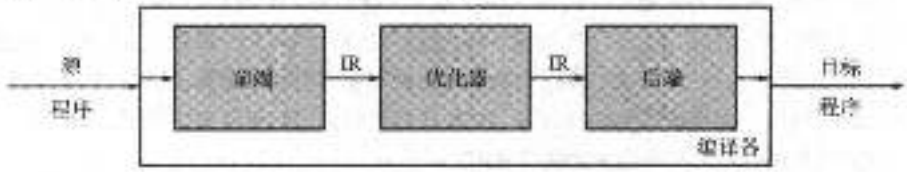
在一个两阶段编译器中，前端必须确保源程序的正确性，而且必须把源代码映射到IR。而后端必须把这个IR程序映射到目标机器的指令集和有限资源上。因为后端只加工前端产生的IR，它可以假设IR不包含语法和语义错误。

编译器在生成目标程序之前可能要对代码的IR形式进行多遍处理。这将产生更好的代码；实际上，编译器可以在它的第一个阶段研究这个代码并记录相关细节，然后，在第二个阶段，它就能够使用这些记录来改进翻译的质量。（这一观点并不新鲜，最初的FORTRAN编译器就对代码进行了多遍处理。）这一策略要求把第一遍得到的信息记录在IR中，使其在第二遍中可以找到并使用。

最后，这种两阶段结构可以简化对编译器重新制定目标的过程。我们可以设想对应于单一的前端构造若干个后端，从而构筑接受相同语言但是以不同的机器为目标的编译器。类似地，我们能够设想对不同的语言构筑相同的IR并使用同一个后端。以上两种情形都假设一个IR可以服务于源语言和目标语言的

若干组合；在实践中，特定语言和特定目标机器的细节通常都能找到适当的IR。

引入IR使我们可以在编译过程中增加更多的阶段。编译器设计者可以在前端和后端之间插入第三个阶段。这一中间部分，或称优化器（optimizer）以IR程序作为它的输入，并生成等价的IR程序作为输出。通过把IR作为接口，编译器设计者能够以最少的变更将第三个阶段插入到前端和后端之间，这导致下面的编译器结构，称为三阶段编译器（three-phase compiler）。



优化器是一个IR到IR转换器，它以某种方式改进IR程序。（注意，根据1.1节的定义，这些转换器本身也是编译器。）优化器可以对IR进行多遍处理，分析IR和重写IR。优化器可能是为了使后端生成较快的目标程序而重写IR，也可能是为了使后端生成较小的目标程序而重写IR，也许还有其他目的，例如像
6 生成产生较少页失效或省电的程序等。

从概念上看，这种三阶段结构代表了典型的优化编译器。在实践中，这些阶段可以进一步分解成一系列的遍。前端由两遍或三遍组成，负责处理识别合法的源程序并生成这一程序的初始IR形式。中间部分包含完成各种优化的若干遍。这些遍的数量和目的随着编译器的不同而不同。后端由若干遍组成，其中的每遍都生成一个更加接近目标机器指令集的IR程序。三个阶段和它们各自的遍分享一个共同的底层基础结构。这一结构如图1-1所示。

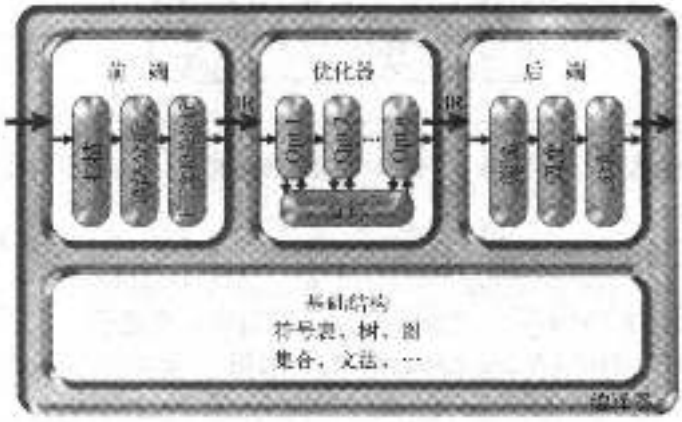


图1-1 典型编译器的结构

在实践中，将编译器从概念上分解为前端、中间部分和后端三个阶段是有利的。每个阶段要处理的问题是不同的。前端主要关心对源程序的理解并将分析的结果记录到IR形式中。中间部分致力于改进IR形式。后端把转换后的IR程序映射到目标机器的有限资源上，映射的结果必须能够有效地利用目标机器的有限资源。

在这三个阶段中，中间部分的描述最不清晰。优化（optimization）这一术语意味着编译器发现对某
7 个问题的优化解。优化中所出现的问题是如此错综复杂，以至于在实践中我们无法得到最优解。另外，编译后的代码的实际行为依赖于应用于优化器中的所有技术以及它与后端之间的相互作用。因此，即使一种技术已经被证明是优化的，它与其他技术间的相互作用也可能产生非优化结果。所以，相对于非优

化编译器，好的优化编译器能够提高代码的质量。但它将几乎总是无法生成最优代码。

中间部分可以是通过一项或多项优化来改进代码的单遍结构，也可以是每个遍都读写IR的多遍结构。单遍结构也许更有效。多遍结构的实现也许更简单，调试起来也更容易。它更灵活，可以在不同的情况下使用不同的优化组合。这两种途径的选择依赖于对编译器的创建和操作的限制。

1.5 翻译综述

为了对编译过程中的任务有更好的理解，考虑对于如下表达式生成可执行代码我们必须做些什么？

w ← w × 2 × x × y × z

其中，w、x、y和z都是变量，←表示赋值，×是乘法操作符。为了了解编译器必须发现的事实和它必须回答的问题，我们将跟踪编译器把这样一个简单的程序转化成可执行代码的过程。

1.5.1 理解输入

在编译w←w × 2 × x × y × z时，第一步是决定这些字符是否形成程序设计语言中的合法语句。这一工作由编译器的前端来完成。它关系到格式，或称语法（syntax）和含义，或称语义（semantics）。如果在这两个方面都合法的话，编译器就能够继续进行翻译、优化和代码生成。如果程序不合法的话，那么编译器将向用户报告明确的错误信息，这一错误信息将尽量表明语句中出现的问题。 8

表记法

本质上，关于编译器的书籍是关于表记法的书籍。毕竟，编译器把一个用一种表记法写成的程序翻译成用另外一种表记法写成的等价程序。在你阅读这本书时，会遇到若干关于表记法的问题。在某些情况下，这些问题将直接影响你对本书内容的理解。

1. 表示算法

我们尽量保持算法的简洁性。我们假设读者能够提供实现细节，所以使用相对高级的形式给出算法。我们使用代码体斜体书写算法并采用缩进格式，这很有意义，特别是对*if-then-else*结构。一个*then*或*else*之后的缩进使相关代码形成一个块。对于下面的代码片段而言

```
if Action [s,word] = "shift s_i" then
    push word
    push s_i
    word←NextWord()
else if ···
```

*then*和*else*之间的所有语句都是*if-then-else*结构中的*then*子语句的部分。当在*if-then-else*结构的一个子语句中只包含一个语句时，我们把关键字*then*或*else*与子语句写在同一行上。

2. 编写代码

在某些例子中，为了展示特定的观点，我们给出某个特定语言的实际程序。我们用代码体书写实际程序。

3. 算术操作符

最后，除了在实际程序之外，我们摒弃传统的用*表示×和用/表示÷的用法。对读者来说含义应该是清晰的。

1. 检测语法

为了检测输入程序的语法，编译器必须把这一程序的结构和语言的定义对照比较。这需要有一个适
9 当的形式定义，一个测试输入是否符合定义的有效机制以及一个如何处理不合法输入的方案。

从数学的角度看，源语言是一个集合，通常是由有限个规则定义的符号串的无穷集合。这样的规则的有限集合称为文法（grammar）。在编译器的前端，实际上是扫描器和语法分析器确定输入程序是否是合法串的集合中的元素。在这里，我们面临的工程学挑战是如何高效地进行成员测试。

程序设计语言的文法通常把词性称为字或语法范畴。基于构筑在词性上的文法规则使每一个规则可以描述很多句子。例如，在英语中，很多句子有如下形式

Sentence → *Subject* verb *Object* endmark

其中，verb和endmark是词性，而*Sentence*、*Subject*和*Object*是语法变量。*Sentence*代表由这一规则所描述的形式的任意串。符号“→”读做“派生”，表示右部的实例可以抽象成为左部的语法变量。

为了运用这一规则，用户必须把字映射到词性。例如，verb代表英语的所有动词的集合，endmark代表所有表示句子结束的标点符号，例如句号、问号或惊叹号。对于英语，读者一般都能认识几千个单词并且知道它们所代表的词性。对于不熟悉的单词，读者可以查字典。因此，本例的语法是由一个规则集合或文法以及一个查找单词并把它们归类成相应语法范畴的系统组成的。

这一基于描述定义语法的方法是编译的关键。我们不能构筑一个含有规则的无穷集合或句子的无穷集合的前端。取而代之，我们需要用规则的有限集合来生成或描述语言中的句子。正如我们将在第2章和第3章看到的那样，文法的有限性对语言的表达能力没有约束。

为了弄清楚“Compilers are engineered objects.”是否是英语中的一个合法句子，我们首先要查字典来判定每个单词都是英文单词。其次，用每个单词的文法范畴取代这个单词来生成这个句子的更抽象的表示。

```
noun verb adjective noun endmark
```

10 最后，我们设法把这个抽象了的单词序列填入英语句子的规则中。已知英语文法可能包括下面规则：

1	*Sentence*	→	*Subject* verb *Object* endmark
2	*Subject*	→	noun
3	*Subject*	→	*Modifier* noun
4	*Object*	→	noun
5	*Object*	→	*Modifier* noun
6	*Modifier*	→	adjective
	...		

通过观察，我们可以发现这个例句的如下派生（derivation）：

规　则	句　型
—	*Sentence*
1	*Subject* verb *Object* endmark
2	noun verb *Object* endmark
5	noun verb *Modifier* noun endmark
6	noun verb adjective noun endmark

这一派生始于语法变量*Sentence*。在每一步，它重写句型中的一项，使用左部为这个项的适当规则的右

部替换这个项。第一步使用规则1的右部来替换*Sentence*。第二步使用规则2的右部替换*Subject*。第三步使用规则5的右部替换*Object*，最后一步根据规则6使用adjective重写*Modifier*。此时，派生生成的句型与输入句子的抽象表示相匹配。

这一派生证明“Compilers are engineered objects.”属于由规则1到规则6所描述的语言。发现字符串的字并根据它们的词性分类的过程称为*扫描*（scanning）。发现一个已分类的字流是否在某个文法规则的集合中存在派生的过程称为语法分析（parsing）。扫描和语法分析是编译一个程序的前两个步骤。

当然，扫描器和语法分析器也许会发现输入不是一个合法的语句。在这种情况下，编译器必须向用户报告错误信息。它将提供简明而有用的反馈，从而使用户能够找到并改正语法错误。 11

2. 检测语义

语法正确性完全依赖于词性和字到它的词性的映射。语法检测忽视字的意义。文法规则不区分两个名词之间的差异，例如“compilers”和“rocks”之间的差异。因此，在文法上，句子“Compilers are engineered objects.”与“Rocks are engineered objects.”之间没有区别，即使它们有完全不同的含义。为了弄清楚这两个句子的差异需要软件系统和地质对象的相关知识。

在前端能够把输入程序翻译成编译器的IR之前，它必须确认程序有明确的意义。语法分析可以在把词性和文法规则相比较的级别上确认句子是合法的。然而，正确性和意义不仅局限于此。在英语句子中，读者必须理解单词的定义，即它们的外延和内涵。在程序设计语言中，编译器必须保证变量名的使用与它们的声明一致。在上述两种情况下，分析必须做更多工作，而不只是局限于检查拼写和它的语法范畴。

合式的计算机程序描述某种计算。下面的表达式可以在很多情况下不合法，而不仅仅会出现典型的语法错误。

```
w←w × 2 × x × y × z
```

例如，一个或多个名字可能没有定义。变量x也许从没有声明。变量y和z也许是不同类型的，它们不能做乘法。

在编译器中，可以通过*语义分析*（semantic analysis）或*上下文相关分析*（context-sensitive analysis）来发现这样的错误。后者强调这样的观点：输入的某些部分的含义可能依赖于它前面的内容、后面的内容或同时依赖于两者。编译器的前端也许包含独立的执行语义分析的遍，或者把这一分析叠加到语法分析器中。无论是哪种情况，检查输入的语义错误的过程推导出程序含义的重要信息，这一信息塑造编译器前端所生成的程序的IR的形式。

3. 学习向导

第2章到第4章描述编译器前端中用于分析输入程序、确定输入程序是否合法以及使用某种内部形式构造代码表示的算法和技术。第5章和附录B阐述在设计和实现用于整个编译器的内部结构时出现的问题。前端构筑许多这样的结构。 12

1.5.2 创建和维护运行时环境

编译器实现由源语言定义的抽象。编译中所关心的就是寻找创建这些抽象表示的有效方法。考虑我们的例子，w←w × 2 × x × y × z。它展示了一个特殊的抽象：符号名。这一表达式涉及w、x、y和z。这些名字不只是值；w既出现在赋值操作符的右边又出现它的左边。显然，当x × y × z不为1/2时，在执行这一表达式之前它有一个值，而在执行之后它又有另外一个值。因此，w涉及存储在一个被命名的位置中的这个值，而不是一个如15这样的特定值。

现代计算机是通过数值地址而不是通过文本名来组织内存的。在一个正在运行的程序的地址空间内，

这些地址惟一确定存储位置。然而在源程序中，程序员可以创建多个具有相同文本名的不同变量。例如，很多程序在若干不同的过程中定义变量i、j和k；它们是循环索引变量的通用名字。编译器负责把名字j的每次使用映射到j的适当实例上，由此再映射到为j的那个实例保留的存储位置上。计算机没有这种存储位置的名字空间；它是由语言设计者创建并由编译器生成的代码以及它的运行时环境所维护的一个抽象。

为了翻译w←w×2×x×y×z，编译器必须给每个名字指定一个存储位置。（此时，我们假设常量2不需要内存位置，因为它是一个小整数而且可以通过立即装入指令得到它。）编译器通过给每个名字指定一个地址在内存中保存这些名字的值，例如<w，0>、<x，4>、<y，8>和<z，12>，这里假设每个值取四个字节。另外，编译器也许会选择使用一系列赋值，如<w，r_1>、<x，r_2>、<y，r_3>和<z，r_4>等在机器的寄存器中保存这些变量。

存储位置的选择既依赖于语境的上下文，同时又对上下文产生影响。将w存放于寄存器中很有可能导致更快的执行。遗憾的是，目标机器只提供有限组寄存器单元；因此，也许没有足够的寄存器来存放
13 w。另外，程序也可能不允许编译器在寄存器中存放w。

名字只是编译器维护的抽象的一种。每个程序设计语言都定义很多抽象，程序员使用这些抽象来编写代码。（Scheme、Java、FORTRAN中的快速排序（QuickSort）的实现看起来就相当不同。）这些抽象使程序员不必了解他们所使用的计算机系统的底层细节。某些抽象很容易实现；在汇编语言中的助记符直接映射到目标机器的操作码上。其他一些抽象则很难实现；SETL中的基于集合的抽象需要编译器从高级别的集合论操作符中推断出高效的算法。

为了处理一个完整的程序设计语言，编译器必须创建并支持各种抽象。过程、参数、名字、词法作用域以及控制流操作都是源语言的抽象。在与其他系统软件的协同中，编译器创建并维护这些抽象在目标机器上的实现。编译器必须在编译时发行适当的指令；而其他部分还涉及编译代码与这些代码所支持的运行时环境之间的相互作用。

设计和实现编译器涉及一系列机制的构建，这些机制将创建并维护程序运行时的必要的抽象。这些机制必须处理内存的布局和分配、过程间的控制转换、值的传播、在过程的边界处的名字空间的映射，以及处理与编译器控制之外的外部世界的接口，这些接口包括输入和输出设备、操作系统以及其他正在运行的程序。

学习向导

第6章探讨编译器必须维护的抽象，抽象的作用是在源语言中的程序设计模型与操作系统及实际的硬件所提供的设备之间搭建桥梁。我们将描述编译器用来实现包含在语言定义中的各种抽象所需要的算法和技术，阐述编译器领域和操作系统领域的一些边缘问题。

第7章阐述本书其他部分的基础知识。优化和代码生成只是挖掘IR程序展示的细节。因此，IR表示什么以及如何表示等的特定决策对优化器的有效性和代码生成器所生成的代码的质量产生重要的影响。
14 这一章关注“代码形态”和编译器设计者就如何实现特定源语言结构所做的一系列选择。

1.5.3 改进代码

通常，编译器可以使用上下文关系的知识来改进它为语句生成的代码的质量。如图1-2左边所示，如果我们给出的例子中的语句被嵌入一个循环，那么上下文的信息可能使得编译器给代码带来显著的改进。编译器能够识别出子表达式2×x×y是循环中的不变量，也就是说，这个子表达式的值在循环间不发生变化。那么编译器就可以如图1-2右侧所示重写代码。转换后的代码在循环体中执行更少的操作。如果循环执行多次，那么转换后的代码应该比原来的代码运行得更快。

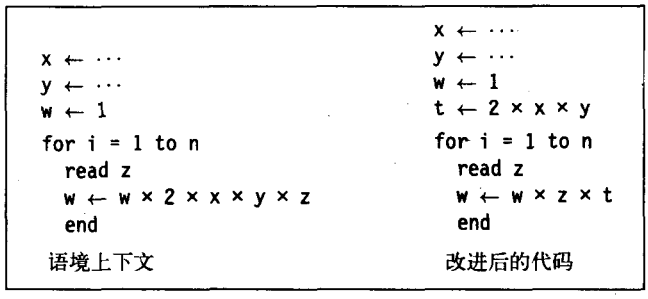


图1-2 不同的上下文导致不同的结果

根据上下文的关系发现事实，分析代码并用这些知识来改进代码的过程通常称为*代码优化*（code optimization）。大体说来，优化包括两种不同的活动：分析代码从而理解代码的运行时行为，以及转换代码来利用在分析中得到的知识。这些技术在编译代码的性能上起着至关重要的作用；好的优化器的存在对编译器其余部分的设计和实现有着重要的影响。

1. 分析

编译器使用多种分析来支持转换。*数据流分析*（data-flow analysis）涉及在编译时推演出运行时的值的流向。数据流分析器通常要对一组集合等式同时求解，这些集合等式由翻译代码的结构推出。*相关性分析*（dependence analysis）使用数论测试来对下标表达式的值进行推理。它通常用来消除数组元素引用中的歧义性。 15

2. 转换

已经开发出很多不同的转化方法，它们都是为了改进可执行代码所需要的时间和空间而设计的。其中有一些转换，例如像发现循环不变量的计算并把这些循环不变量移到不频繁执行的位置这样的转换，可以改进程序的运行时间。另外一些转换使代码更紧凑。对于不同的转换，它们的效果、它们操作的作用域以及进行转换所需的分析等都各不相同。通常，我们将转换与确保这个转换安全且有益所需要的分析捆绑在一起。我们称这样的组合为优化（optimization）。

3. 学习向导

第8章到第10章介绍优化器。第8章通过详细的例子介绍优化的一些术语，其中不做过多的计算。第8章还给出跨越不同作用域并以不同方式工作的算法。第9章介绍数据流分析领域的知识，同时给出构建程序的静态单赋值形式的算法。第10章给出标量转换的分类，以及基于这一分类的大部分种类的优化实例。

1.5.4 生成输出程序

编译的最后阶段是代码生成。在代码生成的过程中，编译器遍历代码的IR形式，并为目标机器生成等价的代码。它选择目标机器的操作来实现代码中的每个IR操作。它要决定使操作能够高效运行的顺序，决定哪些值需要存放在寄存器中，哪些值应该存放在内存中，并插入实施这些决定的代码。

1. 指令筛选

指令筛选是代码生成的第一个阶段。代码生成器选择一系列机器指令来实现被编译的代码。为使语句w←w × 2 × x × y × z在ILOC虚拟计算机上执行，编译器可能选择图1-3所示的操作。代码假设w、x、y和z分别被存放在距寄存器r_{arp}中的地址偏移量为@w、@x、@y和@z的位置。

```
loadAI  r_arp, @w ⇒ r_w      // 装入 'w'
loadI   2         ⇒ r_2      // 把常量2装入r_2
loadAI  r_arp, @x ⇒ r_x      // 装入 'x'
loadAI  r_arp, @y ⇒ r_y      // 装入 'y'
loadAI  r_arp, @z ⇒ r_z      // 装入 'z'
mult    r_w, r_2  ⇒ r_w      // r_w ← w × 2
mult    r_w, r_x  ⇒ r_w      // r_w ← (w × 2) × x
mult    r_w, r_y  ⇒ r_w      // r_w ← (w × 2 × x) × y
mult    r_w, r_z  ⇒ r_w      // r_w ← (w × 2 × x × y) × z
storeAI r_w       ⇒ r_arp, @w // 把r_w写回'w'
```

图1-3 w←x × 2 × y × z的ILOC代码

关于ILOC

本书中，我们使用称为ILOC的表记法来书写低级的例子。ILOG是“intermediate language for an optimizing compiler”的首字符缩写。多年来，这一表记法已经发生了很多变化。我们将在附录A中详细说明本书所使用的版本。

把ILOC想像成简单的RISC计算机的汇编语言。它有一个标准的操作集合。大部分操作的参数是寄存器。内存操作load和store在内存和寄存器之间传输值。为了简化文字说明，大多数例子都假设所有数据都是由整数组成的。

每个操作都有一组操作数和一个目标。操作由五部分组成：操作名、操作数列表、分隔符号、目标列表和可选的注释。因此，将寄存器1和寄存器2的内容相加并把结果保存在寄存器3中的操作为：

```
add r1, r2 ⇒ r3  // 指令例子
```

分隔符号⇒置于目标列表的前面。它形象地表示信息是从左边流向右边的。特别地，它消除了混淆操作数和目标的可能，而阅读汇编文本的人很容易产生这样的混淆。（参看下表中的loadAI和storeAI。）

图1-3中的例子只使用四个ILOC操作：

ILOC操作	意　义
loadAI $r_1, c_2 \Rightarrow r_3$	Memory $(r_1 + c_2) \rightarrow r_3$
loadI $c_1 \Rightarrow r_2$	$c_1 \rightarrow r_2$
mult $r_1, r_2 \Rightarrow r_3$	$r_1 \times r_2 \rightarrow r_3$
storeAI $r_1 \Rightarrow r_2, c_2$	$r_1 \rightarrow$ Memory $(r_2 + c_2)$

附录A给出了ILOC的更详细描述。这些例子始终使用名字r_{arp}为特定寄存器，它或者存放当前过程的数据存储的起始地址，或者存放活动记录指针（activation record pointer）。

16
~
17

这一代码序列是显而易见的。它把所有相关数据装入到寄存器中，依次执行乘运算，并将结果存储于w的内存位置中。它假设有足够多的寄存器，并以诸如r_w来命名这些寄存器来保存w，同时用r_{arp}来保存数据存储的起始地址。无疑，指令筛选器需要依靠寄存器分配器来把这些符号寄存器名或虚拟寄存器映射到目标机器的真实寄存器上。

依赖于目标机器的性质，指令筛选的遍可以作出其他选择。例如，如果立即乘操作（multI）是可用的，那么可以将mult r_w，$r_2 \Rightarrow r_w$换成multI r_w，$2 \Rightarrow r_w$，消除对操作loadI $2 \Rightarrow r_2$的需求，而且减少所需寄存器的数量。如果加法操作比乘法操作更快的话，那么这个操作将会被操作add r_w，$r_w \Rightarrow r_w$取代，从而回避loadI和它对r_2的使用，同时用较快的add取代mult。

2. 寄存器分配

在指令筛选中，编译器有意忽略目标计算机只有有限多个寄存器的事实。它使用任意多个*虚拟寄存器*并假设有“足够多”的寄存器。在实践中，编译的早期阶段也许会创建比硬件所能支持的寄存器更多的虚拟寄存器。把这些虚拟寄存器映射到目标计算机的真实寄存器的任务就落到了寄存器分配器的身上。

寄存器分配器决定在代码的各点哪些值应该存放在目标计算机的寄存器中。然后它要修改代码来反 18
映它的决定。例如，试图最小化所使用的寄存器数目的寄存器分配器也许要把图1-3的代码重写成如下形式：

```
loadAI  r_arp, @w ⇒ r1      // 装入 'w'
add     r1, r1    ⇒ r1      // r1 ← w × 2
loadAI  r_arp, @x ⇒ r2      // 装入 'x'
mult    r1, r2    ⇒ r1      // r1 ← (w × 2) × x
loadAI  r_arp, @y ⇒ r2      // 装入 'y'
mult    r1, r2    ⇒ r1      // r1 ← (w × 2 × x) × y
loadAI  r_arp, @z ⇒ r2      // 装入 'z'
mult    r1,r2     ⇒ r1      // r1 ← (w × 2 × x × y) × z
storeAI r1        ⇒ r_arp, @w  // 把r_w写回'w'
```

这一代码序列使用三个寄存器而不是六个。

最小化寄存器的使用也许不能带来理想的结果。例如，如果所有被命名的值w、x、y和z都已经在寄存器中，那么代码将直接引用这些寄存器。如果所有的值都在寄存器中，我们无需额外的寄存器就可以实现这一代码序列。反之，如果某个邻近的表达式也计算w × 2，那么把那个值保存在寄存器中要比以后重新计算这个值更好。这将增加对寄存器的需求，但是消除了后面的一个指令。

把任意多个值分配到有限的机器寄存器集合的同时保证装入和存储的次数最少的问题是NP完全的。因此，我们不能期望编译器能给出这一问题的最优解，除非我们对某些编译允许指数时间（的复杂性）。在实践中，编译器使用逼近技术来发现这一问题的较好解；这些解可能不是最优解，但是逼近技术确保可以在合理的时间内找到某个解。

3. 指令调度

为了生成快速运行的代码，代码生成器也许需要重新排列操作来反映目标计算机的特定性能约束。不同操作的执行时间可能完全不同。内存存取操作可能会花几十到几百个周期，而某些算术操作，特别是乘法，可能花费几个周期。这些等待时间较长的操作对编译代码的性能的影响可能是巨大的。 19

术语

细心的读者会注意到我们在很多使用*程序*（program）或*过程*（procedure）可能更自然的地方使用了单词*代码*（code）。我们是有意这样做的；编译器可能用于翻译小到单一的引用大到一个完整程序系统的代码片段的翻译。我们不去特殊指定编译的范围，而是继续使用这一含义模糊但更具一般性的词汇：*代码*。

在此，我们假设loadAI和storeAI操作需要三个周期，mult需要两个周期，所有其他操作都需要一个周期。下面给出的代码列表给出了前面代码片段在这些假设下的运行状况。“开始”栏给出了每个操作开始时的周期数，而“结束”栏给出每个操作完成时的周期数。

开 始	结 束				
1	3	loadAI	r_{arp}, @w	$\Rightarrow r_1$	// 装入'w'
4	4	add	r_1, r_1	$\Rightarrow r_1$	// $r_1 \leftarrow w \times 2$
5	7	loadAI	r_{arp}, @x	$\Rightarrow r_2$	// 装入'x'
8	9	mult	r_1, r_2	$\Rightarrow r_1$	// $r_1 \leftarrow (w \times 2) \times x$
10	12	loadAI	r_{arp}, @y	$\Rightarrow r_2$	// 装入'y'
13	14	mult	r_1, r_2	$\Rightarrow r_1$	// $r_1 \leftarrow (w \times 2 \times x) \times y$
15	17	loadAI	r_{arp}, @z	$\Rightarrow r_2$	// 装入'z'
18	19	mult	r_1, r_2	$\Rightarrow r_1$	// $r_1 \leftarrow (w \times 2 \times x \times y) \times z$
20	22	storeAI	r_1	$\Rightarrow r_{arp}$, @w	// 把r_w写回'w'

上表中的九个操作的运行需要22个周期。最小化寄存器数目不能导致快速运行。

很多现代处理器具有这样的性质：它们可以在运行长等待时间的操作的同时启动新的操作。只要直到长等待时间操作的结果在它完成运行之前不被引用，那么运行就可正常进行。然而，如果某个插进来的操作要在这一长等待时间操作完成之前读取它的结果，那么处理器就会“停机”，或者等待这个长等待时间操作完成。一个操作在它的操作数准备就绪之前不能开始运行，并且在这一操作结束之前不能读取它的结果。

20

指令调度器要重排代码中的操作。指令调度器设法使浪费在停机的周期数目最少。当然，调度器必须确保新的操作序列产生与原来操作序列相同的结果。在很多情况下，调度器能够显著改进“朴素”代码的性能。对于我们的例子，优秀的调度器可能会产生下面的操作序列：

开 始	结 束				
1	3	loadAI	r_{arp}, @w	$\Rightarrow r_1$	// 装入'w'
2	4	loadAI	r_{arp}, @x	$\Rightarrow r_2$	// 装入'x'
3	5	loadAI	r_{arp}, @y	$\Rightarrow r_3$	// 装入'y'
4	4	add	r_1, r_1	$\Rightarrow r_1$	// $r_1 \leftarrow w \times 2$
5	6	mult	r_1, r_2	$\Rightarrow r_1$	// $r_1 \leftarrow (w \times 2) \times x$
6	8	loadAI	r_{arp}, @z	$\Rightarrow r_2$	// 装入'z'
7	8	mult	r_1, r_3	$\Rightarrow r_1$	// $r_1 \leftarrow (w \times 2 \times x) \times y$
9	10	mult	r_1, r_2	$\Rightarrow r_1$	// $r_1 \leftarrow (w \times 2 \times x \times y) \times z$
11	13	storeAI	r_1	$\Rightarrow r_{arp}$, @w	// 把r_w写回'w'

代码的这一版本的运行只需要13周期。这一代码比最小寄存器数目版本多需要一个寄存器。除了第8、第10和第12个周期外，它在每个周期启动一个操作。这一调度不是惟一的；存在若干等价的调度，例如使用更多寄存器的等长调度。

如同寄存器分配一样，指令调度是一个困难的问题。它的一般形式是NP完全的。因为这一问题以各种形式出现在很多领域，所以它备受关注。

4. 交互作用

编译过程中的大多数棘手问题都出现在代码生成的过程中。这些问题之间的交互作用使问题更加复杂。例如，指令调度将load操作从依赖于它们的算术操作移开。这一做法可能增加需要这些值的时间，

相应地增加在这段时间内所需要的寄存器的数目。同样地，特殊值到特定寄存器的赋值可能通过引发两个操作之间的“假”依赖而限制指令调度。（第一个操作完成之前不能调度第二个操作，即便公用寄存器中的值相互独立时也是如此。以使用更多寄存器为代价，重命名值可以消除这种假依赖。） 21

令人振奋的时代

这是编译器设计和实现的令人振奋的时代。在20世纪80年代，几乎所有的编译器都是大型单块系统。它们以少数几种语言之一作为输入，并为某个特定的计算机产生汇编代码。这一汇编代码与包括系统函数库和应用函数库在内的其他编译所产生的代码一起联合起来，形成可执行代码。这一可执行代码被存储在磁盘上；在适当的时候，这一最终的代码被从磁盘移到主存中并执行。

今天，编译器技术已开始应用于很多不同的框架。随着计算机在各个不同领域找到其应用，编译器必须应付新的不同的限制。速度不再是衡量编译代码的惟一标准。今天，人们可能要依据编译代码的大小、所消耗的能源量、压缩的程度或者运行时所生成的页错误的多少等因素来衡量代码。

与此同时，编译技术已经脱离了20世纪80年代的单块系统。它们呈现出很多新局面。Java编译器采用（以Java的“字节代码”形式出现的）特殊的编译程序，并把它们翻译成目标机器的本地代码。在这样的环境下，成功要求编译时间与运行时间的总和必须小于解释代码所需的成本。分析整个程序的技术正在从编译时转向链接时，因为连接器可以为整个应用程序分析汇编代码，并使用分析得到的知识改进这一程序。最后，人们正在尝试在运行时调用编译器来利用任意早期无法知道的事实生成优化代码。如果编译时间可以保持到很小，而且收益很大，那么这一策略能够产生显著的改进。

5. 学习向导

第11章到第13章描述代码生成过程中出现的问题，并给出处理这些问题的各种技术。第11章讨论指令筛选的算法，也就是如何把一个特殊的代码形态映射到目标计算机的指令集上。因为执行顺序可以很大程度地影响编译代码的性能，所以第12章集中研究指令调度的算法。最后，第13章将考虑决定将哪些值保存在寄存器中的问题，并探讨编译器用来做出这些决定的算法。 22

1.6 编译器应有的性质

编译器的基本原则告诉我们编译器必须做什么。然而，这些原则却不能对一个编译器应有的所有性质和行为给出描述。虽然特定的编译器拥有它们自己的优势和局限，我们往往要根据五个不同领域的性能来评价编译器。

1. 速度（Speed）

无论何时，我们都会要求某些应用软件具有比它们能够容易得到的性能更高的性能。例如，模拟像微处理器这样的数字电路的能力总是远远落后于对这样的模拟的要求。类似地，诸如气候建模问题等大型物理问题总是需要大量的计算。对于这些应用软件，编译代码的运行时性能是至关重要的问题。获得可预期的好性能需要编译时的额外分析和转换。这样的额外工作需要更长的编译时间。

2. 空间（Space）

很多应用软件对编译代码的尺寸有很严格的限制。这些限制通常来自于物理因素或经济因素。例如，手持计算机的能量消耗部分地依赖于它所含有的内存量。代码经常存放在*只读存储器（ROM）*中；代码尺寸决定设备所需要的ROM。从网格计算到嵌入了applet的网页的各种环境在代码运行之前都要在计

算机之间传递可执行代码；这对编译代码的尺寸增加了额外开销。通过设计编译器可以致力于生成紧凑代码。然而，紧凑代码与快速代码的期望可能相互抵触。

3. 反馈（Feedback）

当编译器遇到不正确的程序时，它必须向用户报告这一事实。提供给用户的信息量可以千差万别。
23 例如，早期的Unix编译器通常生成一个简单而统一的信息："语法错误。(syntax error.)"而作为另一个极端，Cornell PL/C系统和UW-Pascal系统被设计为"学生"编译器，它们努力改正程序中的每一个语法错误再去编译它。

4. 调试（Debugging）

遗憾的是，大多数程序不能在编译后马上正确地运行。因此，程序员对调试器的能力寄予了很高的期望，这样的调试器能够对编译代码进行源程序级别的调试。如果调试器设法把出错的可执行代码的状态与源代码联系起来的话，那么大动干戈的程序转换所带来的复杂性可能会使调试器误导程序员。因此，编译器设计者和用户都可能被迫在编译器代码的效率和调试器的透明度之间做出选择。这就是为什么很多编译器都有一个"调试"标志，这一调试标志阻止编译器使用使源代码和执行程序之间的关系模糊的转换。

5. 编译时效率（Compile-Time Efficiency）

编译器的使用频率非常高。在很多情况下，编译器的用户要等待编译的结果，所以编译速度可能是一个重要的问题。在实践中，没有人喜欢等待编译器完成工作。也许有些用户能够容忍缓慢的编译，特别是当代码质量出现严重问题时。然而，如果在产生相同结果的缓慢的编译器和快速的编译器之间可以作出选择的话，用户无疑会选择快速编译器。

在阅读本书其余部分之前，你应该列出你希望编译器应有的特性顺序表。你可以运用软件工程的传统标准评估它的特性，就好像你需要为它们花费你自己的金钱那样！仔细审查你的列表，你会学会在构建你自己的编译器时应该如何作各种权衡。

1.7 概括和展望

构造编译器是一个复杂的任务。优秀的编译器把来自于形式语言理论、算法研究、人工智能、系统设计、计算机体系结构以及程序设计语言理论的思想结合起来，并把它们运用于翻译程序的任务。编译
24 器把贪婪算法、试探技术、图形算法、动态规划、DFA和NFA、不动点算法、分配和命名、同步和局部化、管道管理等技术结合在一起。编译器所面临的很多问题都很难得到最优的解决方案；因此，编译器使用逼近、试探和经验。这导致令人咋舌的复杂的相互作用，无论是好的还是坏的。

为了使这一活动有序进行，大多数编译器都组织成三个主要阶段：前端、优化器和后端。每个阶段都有各自需要处理的问题；解决这些问题的方法也大相径庭。前端主要致力于把源程序代码翻译成某个IR。前端依赖于形式语言理论和类型论的结果，依赖于大量的算法和数据结构。而中间阶段或优化器，把IR程序翻译成另外的IR程序，其目的是获得高效的IR程序。优化器分析程序得到程序的运行时行为的信息，然后利用这些信息转换代码并改进它的行为。后端把IR程序映射到特定处理器的指令集上。后端对分配和调度中的难题给出近似解；近似解的质量对编译代码的速度和尺寸有直接的影响。

本书分别讨论这三个部分。第2章到第4章处理编译器前端所使用的算法。第5章到第7章给出讨论优化和代码生成所需的背景资料的描述。第8章介绍代码优化；第9章和第10章为感兴趣的读者提供更详细的分析和优化处理。最后，第11章到第13章涵盖后端所需的指令筛选、调度和寄存器分配技术。

本章注释

第一批编译器出现于20世纪50年代。这些早期的系统显示出令人惊讶的复杂度。原始的FORTRAN编译器是包括独立的扫描器、语法分析器和寄存器分配器以及某些优化的多遍系统[26，25]。由Ershov和他的同事在Novosibirsk设计的Alpha系统实现了局部优化[13]并使用图形着色技术来降低数据项所需要的内存数量[132，133]。

20世纪60年代早期，Knuth重新概括了一些有趣的编译器构造法[221]。Randell和Russell描述了对Algol 60的早期实现所做的努力[282]。Allen描述了IBM内部的编译器发展历史，着重强调理论和实践的相互作用[14]。 25

20世纪60年代和20世纪70年代，人们构建了很多有影响力的编译器。其中包括经典的优化编译器FORTRAN H[243，297]，Bliss-11和Bliss-32编译器[339，67]，以及可移植的BCPL编译器[289]。这些编译器为各种复杂指令集计算机（CISC）生成了高质量的代码。另一方面，学生编译器则致力于快速编译、优秀的诊断信息以及错误更正[92，139]。

20世纪80年代，精简指令集计算机（RISC）构架的出现导致了新一代的编译器；这些编译器致力于强大的优化和高质量代码生成[84，23，76，194]。这些编译器配备了如图1-1所示的成熟的优化器结构。现代的RISC编译器仍然沿用这一模型。

20世纪90年代期间，编译器构造法的研究集中于对发生在微型处理器结构的快速变化做出响应。这10年开始于Intel的*i*860处理器向编译器设计者发起的管道管理和内存等待的直接挑战。此后，编译器又面临着来自于多功能单元、长内存等待时间直到并行代码生成等领域的挑战。实践证明，20世纪80年代的RISC编译器的结构和组织足以灵活地应付这些新挑战，所以研究人员在编译器的优化和代码生成阶段插入新遍。 26



第2章
扫　描

2.1 概述

扫描是编译器用于理解输入程序的三个部分过程的第一个。扫描器，或词法分析器以一串字符为输入，生成一串字以及与这些字相关的语法范畴作为输出。扫描器收集字符形成字并使用一系列规则来决定每个字在源语言中是否合法。如果字是合法的，那么扫描器就给出它的语法范畴或词性。为了使这一过程得以高效进行，编译器使用专门的识别器。

本章将阐述用于实现词法分析的数学工具和程序设计技术。扫描器构架的大部分工作可以自动完成；实际上，这是运用理论结果解决重要实际问题的一个典型例子：描述并识别串的模式。用于描述模式的自然的数学公式，称为*正则表达式*（regular expression）。这一数学工具直接导致称为*有穷自动机*
27 （finite automata）的识别器，识别器扫描符号串寻找已描述的模式。已经存在这样的工具，它们利用正则表达式和有穷自动机之间的理论关系，从描述出发构建高效、专用的识别器。这一技术已应用于很多方面，从诸如Unix grep程序这样的工具到网站过滤软件，到在文字编辑器、字处理工具以及命令行下的正则表达式"查找"命令。

扫描器查看字符串并识别字，控制程序设计语言的词法结构的规则，有时称为*微语法*（microsyntax），是简单且正则的。这导致扫描的高效、特化识别器。通常编译器的前端使用扫描器来识别并分类字。扫描器的输出是一串字，每个字都注明了它的语法范畴或词性。语法分析器依次使用这些字。语法分析器确定这些字是否形成程序设计语言中的一个语法上正确的句子，即一个程序。一旦编译器知道这个输入在语法上是正确的，它的下一个工作就是进行更深一层的分析来查看程序是否有一致的意义。这一分析有时称为*上下文相关分析*（context-sensitive analysis）。确定程序是否有意义的许多细节难以用语法表示；因此编译器必须使用更复杂的技术来检测这些细节。

概念上，以上三种分析是不同的任务。在实践中，它们通常以交叉的方式运行。语法分析器通过调用要求扫描器生成分类的字，而当它识别出代码的各语法子成份时，调用上下文相关分析。这些不同的分析器组合在一起形成编译器的前端，正如在图1-1或每一章开始的图解所示的那样。

把微语法从语法中分离出来能够在以下三个方面简化编译器：

1）用于语法分析器中的语法描述是用字和语法范畴写成的，而不是用字母、数字和空格。这就使得语法分析器可以忽视诸如吸收多余的空格、换行和注释这样的无关问题。这些问题被隐藏在扫描器的内部，在那里它们可以得到清楚而有效的处理。

2）扫描器构建几乎可以完全自动化。我们用形式表记法对词法规则编码，并将其供给扫描器生成器，其结果是为语法分析器生成输入的可执行程序。基于高级描述所生成的扫描器是高效的。

3）被移入扫描器的规则使语法分析器变小。语法分析比扫描更困难；语法分析器的大小随着文法的增长而增长。因为构建语法分析器需要程序员更直接的努力，缩小语法分析器将减少编译设计者的
28 工作。

最后，优秀的扫描器比优秀的语法分析器的额外开销更少（以对于每个输入符号所需执行的指令为

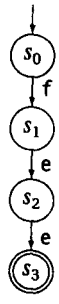
标准)。扫描器把字符收集起来形成字，使得语法分析器能够把每个字当作一个符号处理。这样的做法减少了语法分析器必须处理的字的数量。

本章考察识别字符串中的字并识别每个字的语法范畴的技术。我们展示如何简明地描述程序设计语言中的字，给出从这些描述直接获得扫描器的方法。最后，我们还将给出一些例子来展示使识别字和分类字的任务复杂化的语言设计。

2.2 识别字

当描述识别字的算法时，按字符给出的公式有时可以使事情变得清晰。对于简单的算法，代码的结构能够展现出潜在的一些问题。考虑识别字fee的问题。假设有例行程序*NextChar*，它返回下一个字符。这个识别fee的代码可能如下面的片段:

```
c ← NextChar()
if (c ≠ 'f')
  then do something else
  else
    c ← NextChar()
    if (c ≠ 'e')
      then do something else
      else
        c ← NextChar()
        if (c ≠ 'e')
          then do something else
          else report success
```

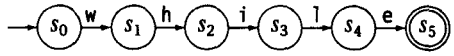


这一代码测试f后面跟着一个e再跟着一个e。在每一步，与相应字符匹配的失败导致这一代码拒绝这个字符串并做一些其他的事情。(如果程序的惟一目标是识别字fee，那么适宜的行为也许是打印一个错误信息或返回失败。正如我们将看到的那样，扫描器很少只识别一个字，所以在此我们不明确给出“出错”处理。)

29

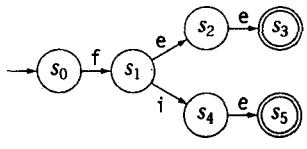
这一简单的代码片段对每一个字符使用*if-then-else*结构执行一个测试。我们可以用代码右边所示的简单图形来表示这一代码片段。圆圈或结点，表示计算的抽象状态，以0到3编号。初始状态或开始状态，被标记为s_0。在本章中，除非明确指出，s_0都是开始状态。终结状态，如本例中的s_3，以双圈表示。箭头表示基于输入字符从一个状态到另一个状态的转换。如果我们从最高点的状态开始，并看到字符f、e和e，这一转换带着我们进入最下面的状态。对于其他的输入，例如f、i、e，将会发生什么呢？f带着我们来到第二个状态。而i与边不匹配，于是我们停留在第二个状态，所以我们知道这个输入字不是fee。与fee不匹配的所有情况都按代码中的“做其他事情”处理。我们可以把这看成是一个到错误状态的转换。

使用同样的实现策略，一个识别字while的代码片段可以编码成五个嵌套的*if-then-else*结构。由于这一代码阅读起来很烦琐，我们只给出转换图:



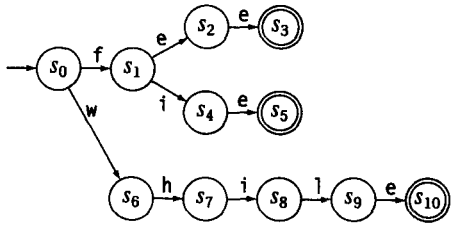
如果我们在状态s_0启动代码，而且达到了s_5，我们就知道这个输入串的前五个字符是while。

为了识别多个字，我们可以填充代码中的“做其他事情”这部分。同时识别fee和fie的代码片段可以如下表示:



这一片段对f使用一个公用的测试，这一测试把状态从s_0转换到s_1。如果第二个字符是e，那么就进行转换$s_1 \rightarrow s_2$。反之，如果第二个字符是i，那么就进行转换$s_1 \rightarrow s_4$。最后，如果它在状态s_2遇到一个e，那么
30 就进行转换：$s_2 \xrightarrow{e} s_3$。在状态s_4看到e导致转换$s_4 \xrightarrow{e} s_5$。在这一片段中，状态s_3和s_5是终结状态。

如果需要，我们可以通过合并初始状态并且对其他状态重新编号把识别fee和fie的片段与识别while的片段结合起来。这一结合产生如下转换图：



现在，s_0有相对于f和w的两个转换，而且有三个终结状态。当在任意状态遇到了一个与它的转换中的字符不匹配的输入字符时，就显示一个错误。

2.2.1 识别器的形式

这些转换图是需要实现的代码的抽象表示。也可以把它们看成是称为*有穷自动机*（finite automaton）的，用于描述识别器的形式数学对象。一个有穷自动机（FA）是由以下元素组成的：状态的有穷集合、这些状态之间的转换的集合、字母表、初始状态（s_0）以及由一个或多个终结状态组成的集合。

形式上，一个FA是一个五元组（S，Σ，δ，s_0，S_F），其中，

- S是一个状态集合。这个集合包含转换图中的每个状态，还有一个特殊的错误状态s_e。S必须是有穷的。
- Σ是一个字母表或识别器所用的字符集合。通常，Σ是转换图中的边的标签的全体所组成的集合。Σ必须是有穷的。
- $\delta(s, c)$是一个两参数函数，一个参数是状态$s \in S$，一个参数是字符$c \in \Sigma$。这个函数表示FA的转换。当FA处于状态s，并看到一个c时，下一个状态转换成状态$\delta(s, c)$。
- $s_0 \in S$是特定的初始状态。
31 - S_F是终结状态的集合。S_F是S的子集。S_F中的每一个状态在转换图中以双圈表示。

为了使上述表述更加具体，让我们回顾前一节最后的FA，这一FA识别fee或fie或while。把这个FA变成形式定义，我们得到如下形式：

$$S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_e\}$$

$$\Sigma = \{\mathtt{e}, \mathtt{f}, \mathtt{i}, \mathtt{h}, \mathtt{l}, \mathtt{w}\}$$

$$\delta = \left\{ \begin{array}{lllll} s_0 \xrightarrow{f} s_1, & s_0 \xrightarrow{w} s_6, & s_1 \xrightarrow{e} s_2, & s_1 \xrightarrow{i} s_4, & s_2 \xrightarrow{e} s_3, \\ s_4 \xrightarrow{e} s_5, & s_6 \xrightarrow{h} s_7, & s_7 \xrightarrow{i} s_8, & s_8 \xrightarrow{l} s_9, & s_9 \xrightarrow{e} s_{10} \end{array} \right\}$$

$$s_0 = s_0$$

$$S_F = \{s_3, s_5, s_{10}\}$$

这个五元组等价于这个FA的转换图；给定其中一个，我们很容易生成另外一个。在某种意义上，转换图是相应FA的刻画图。

如果FA处于状态s，读入字符c，且$\delta(s, c)$无定义，则表明输入字符串有错误。在这种情况下，$\delta(s, c)$将返回一个出错的信号。我们也可以把这个错误信号看成是返回一个明确的错误状态s_e，而且假设，对于任意的$c \in \Sigma$有$s_e \xrightarrow{c} s_e$。

一个FA接受一个字符串x当且仅当，从状态s_0开始，字符串带着FA通过一系列转换，并当整个字符串被用尽后停在一个终结状态。这与我们对转换图的直观认识相符。对于字符串fee，我们例子中的识别器经过$s_0 \xrightarrow{f} s_1$，$s_1 \xrightarrow{e} s_2$，$s_2 \xrightarrow{e} s_3$。因为$s_3 \in S_F$，且没有剩余的输入，所以这个FA接受这个字符串fee。对于字符串foe，FA的行为就不同了。初始转换$s_0 \xrightarrow{f} s_1$是相同的。然而在状态s_1处没有作用于o的正常转换，所以δ返回s_e。

为了更加形式化，如果字符串x是由字符$x_1x_2x_3\cdots x_n$组成的，那么这个FA$(S, \Sigma, \delta, s_0, S_F)$接受$x$当且仅当

$$\delta(\delta(\ldots\delta(\delta(\delta(s_0, x_1), x_2), x_3)\ldots, x_{n-1}), x_n) \in S_F$$

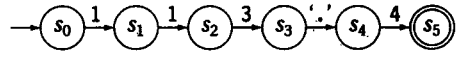
直观上，这一表达式对应于将δ反复作用于由属于S的状态s和输入符号x_i组成的序对。基本情况是$\delta(s_0, x_1)$，这对应于FA的初始状态。接着，通过运用δ所产生的这一状态，连同x_2，被用作δ用于产生下一个状态的输入，依此类推，直到所有的输入都被用尽为止。最后一次运用δ的结果还是一个状态。如果这个状态在S_F中，那么FA就接受$x_1x_2x_3\cdots x_n$。

32

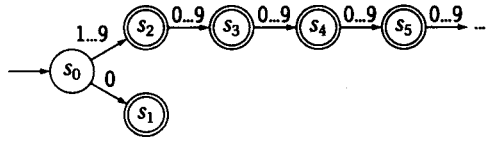
还有另外两种情况。FA在处理字符串时，也许会遇到一个错误。也就是说，它也许在状态s_i遇到一个字符x_j，并发现$\delta(s_i, x_j)$无定义。这显示一个词法错误。字符串$x_1x_2x_3\cdots x_j$不是这个FA所接受的语言中的任意合法字的有效前缀。另外，这个FA也许到达x_n，处理它，但停在一个非终结状态。在这种情况下，这个输入字符串是FA所接受的某个字的真前缀。同时，这也显示一个错误，并应报告给终端用户。

2.2.2 识别更复杂的字

很容易把前面的fee的识别器中所展示的逐字处理模型扩展到处理任意多个完整刻画的字。那么，我们怎样使用这样的识别器识别一个数呢？对于特殊的数，例如113.4，这很容易办到。



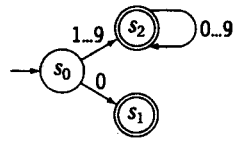
然而，为使这样的识别实用，我们需要能够识别任意一个数的转换图（以及相应的代码片段）。为了简便起见，我们只局限于对无符号整数的讨论。一般来说，一个整数或者是0或者是一个由一个或多个数字组成的序列，序列的第一个元素是从1到9的数字，而其他元素是从0到9的数字。（这排除了前导零。）我们如何画出这一定义的转换图呢？



转换$s_0 \xrightarrow{0} s_1$，处理0的情况。其他路径，从s_0到s_2，到s_3，等等，处理大于零的整数。然而，这一路径给我们提出几个问题。第一个问题是，它不结束。这违背了FA有有穷状态集合的要求。第二个问题是，开始于s_3的路径上的所有状态都是等价的：它们的输入和输出转换上有相同的标签，而且它们都是终结状态。

33

如果我们允许转换图包含循环，那么我们可以显著简化这个FA。我们可以把从s_2开始的整个状态链替换成从s_2开始返回到其本身的单一转换，如下所示：



上面的转换图作为FA是有意义的。然而，从实现的观点看，它比前面给出的无环图更复杂。我们不能把它直接转换成一系列嵌套的*if-then-else*结构。在转换图中引入循环导致对循环控制流的需要。我们可以用图2-1所示的*while*循环来实现上述的转换图。我们可以用一个表来有效地刻画δ:

δ	0	1	2	3	4	5	6	7	8	9	其他
s_0	s_1	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_1	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e
s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e

```
char ← NextChar()
state ← s0
while (char ≠ eof and state ≠ se)
   state ← δ(state,char)
   char ← NextChar()
if (state ∈ SF)
   then report acceptance
   else report failure
```

$$S = \{s_0, s_1, s_2\}$$

$$\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$$

$$\delta = \left\{ \begin{array}{c} s_0 \xrightarrow{0} s_1,\ s_0 \xrightarrow{1\text{-}9} s_2 \\ s_2 \xrightarrow{0\text{-}9} s_2 \end{array} \right\}$$

$$S_F = \{s_1, s_2\}$$

图2-1 无符号整数识别器

34 通过修改这个表，我们可以使用相同的基本代码框架来实现其他的识别器。注意，上面的表有充分的压缩潜力。数字1到9的各列是相同的，所以它们可以只表示一次。这样就使这个表只有三列：0、1...9和“其他”。仔细考察这一代码的框架可知，这一代码只要进入状态s_e就会报告失败，所以它从不引用表中对应于s_e的那一行。实现可以消除整个这一行，于是原来的4行11列的表可以换成一个3行3列的表。

我们也可以开发接受带符号整数、实数和复数的FA。对于上述每一种情况，FA识别字的一个无穷集合。虽然这样的FA可以看成是识别器的描述，但它们并不是特别简明的描述。为了简化扫描器的实现，我们希望有一个描述字的词法结构的简明表记法，和一整套把这些描述转变成FA并转变实现这个FA的代码的自动化技术。本章其余小节讨论这样的表记法和技术。

2.2.3 扫描器的自动构建

从描述出发自动构建扫描器的工具随手可得。这些工具的基本结构如图2-2所示。编译器设计者为工具提供一组用来描述特定源语言中各种字的词法模式。扫描器生成器分析这些模式并产生一个成为扫
35 描器核心的识别器。这个识别器可能被编码成一组表格，或者直接编码成可执行代码。无论是哪一种情况，它都可以从这些模式得到正确、高效的扫描器。因为这些工具生成优秀的扫描器，所以编译器设计者很少手工构建扫描器。

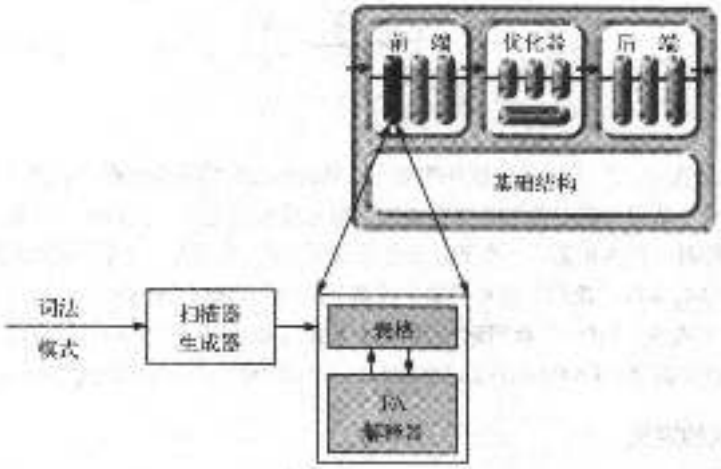


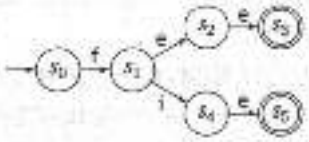
图2-2 自动扫描器生成

2.3 正则表达式

一个有穷自动机F接受的字的集合形成一个语言，记作$L(F)$。这一FA的转换图精确地描述这个语言。然而，它对我们来说并不直观。对于任意的FA，我们也可以使用称为正则表达式（regular expression）的表记法来描述它的语言。

正则表达式（RE）等价于上节所述的有穷自动机（FA）。（我们将运用2.4节所给的构造法证明这一点。）简单的识别器有简单的RE描述。

- 由单个字fee组成的语言可以用写成*fee*的RE来描述。这里，彼此相邻的两个字符表示我们希望它们按书写的顺序依次出现。表达式*fee*是这一语言的一个RE。
- 由两个字fee和while组成的语言可以写作*fee or while*。为了避免误解*or*，我们用符号|表示or（或者）。因此，我们把这个RE写作*fee*|*while*。
- 由fee和if1e组成的语言可以写作*fee*|*fie*，也可以写作*f*(*e*|*i*)*e*。这个RE *f*(*e*|*i*)*e*比*fee*|*fie*更接近FA的结构。



为使上面的描述更具体，考虑程序设计语言中的一些例子，标点符号，例如冒号、分号、逗号和各种括号，可以用它们的字符形式来表示。因此，我们可以得到典型程序设计语言的如下词法描述RE：

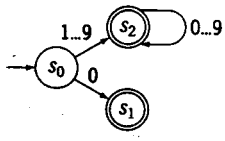
: ; ? => ( ) { } [ ]

同样地，关键字有如下简单的RE：

if while this integer instanceof

为了模型化更复杂的结构，例如，整数或标识符，我们需要能够刻画FA中循环边的本质的表记法。

无符号整数的FA有三个状态，一个初始状态s_0，对应于单一整数0的终结状态s_1和对应于所有其他整数的终结状态s_2。



这一FA的优点所在是，对于每一个额外的数字，状态s_2出发的转换都返回其本身。状态s_2把描述折叠回其本身，生成一个从现存的无符号整数得到新无符号整数的规则：把另外一个数字添加到现存整数的右侧。这一规则的另一种刻画是，一个无符号整数或者是0，或者是一个非0数后面跟着零个或多个数字。为了刻画这一FA的本质，我们需要对“零个或多个出现”的概念给出一个表记法。对于RE x，我们用x^*来表示“x的零个或多个出现。”我们称*运算符为*克林闭包*（Kleene closure）或简称*闭包*（closure）。使用闭包运算符，我们把这个FA对应的RE记作0|(1|2|3|4|5|6|7|8|9) (0|1|2|3|4|5|6|7|8|9)*。

2.3.1 正则表达式的定义

为了用更严格的方式使用正则表达式，我们必须更加形式化地定义它们。一个正则表达式描述某个字母表Σ上的字符串，加上表示空字符的ε，组成的一个集合。我们称这个字符串集合为*语言*（language）。给定正则表达式r，我们把r描述的语言记作$L(r)$。正则表达式由下面三个基本操作构筑而成：

（1）*选择*（alternation）

37 两个集合R和S的选择，或并，记作$R|S$，定义为$\{s|s\in R$或$s\in S\}$。

（2）*连接*（concatenation）

两个集合R和S的连接，记作RS，定义为$\{st|s\in R$且$t\in S\}$。有时我们把RR写作R^2，这是集合R与其本身的连接，同样，把RRR（或RR^2）写作R^3。

（3）*闭包*（closure）

集合R的克林闭包，记为R^*，定义为$\cup_{0}^{\infty}R^i$。也就是说，先让集合R与其本身经过零次或多次连接，然后对所有这些结果取并集。

虚拟生活中的正则表达式

很多描述字符串模式的应用使用正则表达式。把正则表达式翻译成代码的一些早期工作产生于在文本编辑器的“寻找”指令中提供的刻画字符串的灵活方法。从这些早期的工作开始，这一表记法已经在不知不觉中运用于不同的应用领域。

在Unix和很多其他操作系统中，星号被用作匹配文件名的任意子串的通配符。这里，*是正则表达式Σ^*的简写形式，表示零个或多个来自于合法字符组成的完整字母表的字符串。（因为很少有键盘有Σ键，所以这一速记形式一直沿用了下来。）很多系统都要使用“?”作为与单一字符匹配的通配符。

grep类的工具以及非Unix系统中的类似工具都实现了正则表达式的模式匹配。（事实上，grep是global regular-expression pattern match and print的字头缩写。）

由于正则表达式很容易书写而且很容易理解，它得到日益广泛的应用。当一个程序必须识别一个固定的词汇时，它是一个可选的技术。它对那些满足它的规则限制的语言很有效。正则表达式很容易翻译成可执行形式；其结果的识别器也很快。

有时，我们使用R的*正闭包*（positive closure），记作R^+，定义为$\cup_{1}^{\infty}R^i$。因为R^+总能写成RR^*，在随后的讨论中我们忽视正闭包。

使用这三个运算，我们能够如下定义字母表Σ上的正则表达式的集合[⊖]：

1）如果$a \in \Sigma$，那么a也是一个RE，表示只包含a的集合。

2）ε是一个RE，表示只含空字符串的集合。 38

3）如果r和s是RE，分别表示集合$L(r)$和$L(s)$，那么

(r)是一个RE，表示集合$L(r)$，

$r|s$是一个RE，表示$L(r)$和$L(s)$的并或选择，

rs是一个RE，表示$L(r)$和$L(s)$中的字符串的连接的集合，

r^*是一个RE，表示$L(r)$的克林闭包。

为了消除歧义性，闭包具有最高优先权，其次是连接，然后是选择。

作为一个便利的简记法，我们用省略号连接第一个元素和最后一个元素来描述字符的范围。为了使这一简写更突出，我们总是使用一对方括号把它括起来。因此，[0...9]表示十进制数字的集合。它与正则表达式（0|1|2|3|4|5|6|7|8|9）描述同一个集合。

2.3.2 例子

本章的目标是展示如何使用形式技术来自动构造高质量扫描器，以及如何用这种形式对程序设计语言的微语法结构编码。在展开我们的讨论之前，在此我们列出来自于实际程序设计语言的一些例子。

1）Algol和它的后代定义标识符为一个字母字符后面跟随着零个或多个字母或数字字符。如果我们假设所用的字母都是小写字母[⊖]，就可以用RE $[a...z]\ ([a...z]|[0...9])^*$来描述标识符的定义。很多语言在标识符中还允许出现若干特殊的字符，例如下划线（_）、百分号（%）或“与”符号（&）。

2）无符号整数可以描述成0或非0数字后面跟随零个或多个数字。RE 0|[1...9][0...9]*更加简明。在实践中，很多实现都允许把一个更大的字符串类作为整数，承认语言[0...9]+。

3）实数比整数更复杂。一个实数可能被描述成 (0|[1...9][0...9]*)(ε|.[0...9]*)。这一表达式的第一部分是整数的RE。余下的部分生成空字符串或一个十进制小数点后面跟随零个或多个数字。 39

程序设计语言通常把这一形式扩展成科学表记法，可以用RE (0|[1...9][0...9]*)(.[0...9]*|ε)E(+|-|ε)(0|[1...9][0...9]*）来描述。这一RE刻画一个实数，后面跟随一个E，后面跟随一个描述指数的整数。

4）被引用字符串有其自身的复杂性。在多数语言中，任何一个字符都可以出现在字符串的内部。这些字符包括空格、制表符、换行符，甚至包括被用于定界字符串的字符。为了表示集合$\{\Sigma - c\}$，我们使用表记法[^c]，它取自于扫描器生成器lex。C中的字符串可以描述成"[^"]"。

诸如C语言中表示换行的\n和表示"的\"这样的转义字符简化了字符串的处理。描述转义字符的另外一个方法是写两个所涉及的字符。因此，为了把字符"放入字符串中，程序员将键入""。这使描述字符常量的RE复杂化。

5）注释以若干形式出现。有些语言允许诸如#或//这样的定界符，表示从定界符一直到当前行的结束为注释。这导致形如//[^\n]*的RE，其中\n表示换行符。其他语言以不同的定界符作为注释的起点和终点。Pascal语言使用{和}作为注释的定界符，它导致形如{[^}]*}的RE。Java语言和C语言允许二字符定界符/*和*/；这导致一个更复杂的RE。

以上各例子都表示字的无穷集合。对字集合的限制通常导致更长、更复杂的RE。例如，某些语言

⊖ 原书把2)和3)调换了位置。作者的定义不容许写诸如$a|\varepsilon$这样的正则表达式，而这样的正则表达式有时是很方便的。见下一个译者注和后文。——译者注

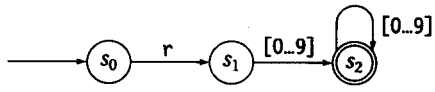
⊖ 如果程序设计语言忽视大小写，那么扫描器可以把所有字母都改成小写字母。如果程序设计语言是大小写敏感的，那么相应的RE也就变得更复杂了。

限制标识符的长度。这导致比以上正则表达式更长的正则表达式。例如，Algol这样的语言把标识符的最大长度限制为6个字符，这导致如下的RE[⊖]：

$$[a...z]([a...z]|[0...9])([a...z]|[0...9])([a...z]|[0...9])([a...z]|[0...9])([a...z]|[0...9])$$

它所对应的FA的状态比无长度限制标识符的FA的状态要多。我们可以引入一个有限闭包c^k的表记法来描述c的0个到k个拷贝。这样，我们就可以将上面六字符标识符的RE写成 $[a...z]([a...z]|[0...9])^5$。它简
40 化并缩短这一正则表达式，但是对相应的FA没有起到简化的作用。

试图使RE更具体可能会导致复杂的表达式。例如，考虑这样的例子：典型汇编语言中的寄存器描述器是由字母r后面跟随着一个小整数组成的。尽管ILOC的寄存器名字的描述是RE $r[0...9]^+$，它容许寄存器名字的无穷集合。相应的识别器如下所示：

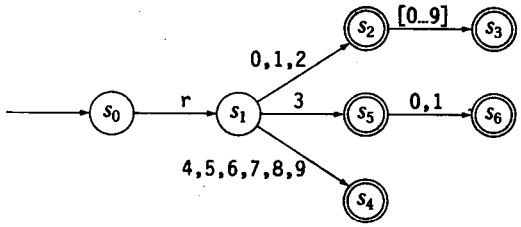


这个识别器接受r29而拒绝s29。它也接受r99999，即使目前不存在100 000个寄存器的计算机。

然而，在一台真实的计算机上，寄存器名集合受到严格的限制，如32、64或128个寄存器。我们能够构建这个寄存器描述器的一个更精确的RE，而不是编写将每一个寄存器名字转换成一个整数，然后看它是否在0到31的范围的代码。下面是一个这样的RE:

$$r((0|1|2)([0...9]|\varepsilon)|(4|5|6|7|8|9)|(3(0|1|\varepsilon)))$$

这一表达式的构造很巧妙，它接受r0、r29和r31，而不接受r32或r99999。它可以用于定义寄存器名字的语法范畴。它刻画一个小语言，将寄存器限于0到31，而且对于一位数的寄存器名可以加入一个前导零。相应的FA如下所示：



哪个FA更好呢？两个FA都对每一个输入字符做一次转换。因此，二者都做同样次数的转换，即使复杂的FA检查更复杂的微语法描述。复杂的FA有更多的状态和更多的转换，所以，它的表示需要更大的空间。然而本质上，它们的操作代价是相同的。

41 这是非常关键的：FA的操作代价与输入字符串的长度成正比，而不与生成识别器的正则表达式的长度或复杂性成正比。增加正则表达式的复杂性能够增加相应FA的状态数目。从而增加表示这个FA所需要的空间和自动构建它的代价，但是操作的代价仍然是每个输入字符一个转换。

我们能够改进寄存器描述器的表述吗？这个RE既复杂又不直观。另外一个RE如下所示：

$$r0|r00|r1|r01|r2|r02|r3|r03|r4|r04|r5|r05|r6|r06|r7|r07|r8|r08|r9|r09|r10|r11|r12|$$
$$r13|r14|r15|r16|r17|r18|r19|r20|r21|r22|r23|r24|r25|r26|r27|r28|r29|r30|r31$$

⊖ 实际上，这一正则表达式表示的是长度刚好为6的标识符的集合，长度最大为6的标识符的集合应该用诸如$[a...z]([a...z]|[0...9]|\varepsilon)([a...z]|[0...9]|\varepsilon)([a...z]|[0...9]|\varepsilon)([a...z]|[0...9]|\varepsilon)([a...z]|[0...9]|\varepsilon)$ 这样的正则表达式来表示。——译者注

这个表达式在概念上更简单，但是比原来的表达式更长些。导致FA仍然对每个输入符号做一个转换。因此，如果我们能够阻止空间需求增加过快，那么我们也许更倾向于这个寄存器描述器，因为它更清晰、明了。

2.3.3 RE的性质

可以用RE表示的语言的集合称为*正则语言*（regular languages）集合。正则语言已得到深入研究；它们有很多有趣而重要的性质。其中一些性质在生成扫描器中起着重要的作用。

正则表达式在很多运算下封闭，也就是说，如果我们对一个RE或一组RE运用一个运算，那么其结果仍旧是一个RE。显然的例子是连接、选择和闭包运算。两个RE，x和y的连接是xy。它们的选择是$x|y$。x的克林闭包是x^*。所有这些结果都符合RE的定义。

RE是封闭的事实在使用RE构建扫描器的工作中起着至关重要的作用。假设对于源语言中的每一个语法范畴我们都有RE，这些RE为a_1，a_2，$a_3\cdots$，a_n。我们能够使用选择把这些RE连结起来形成表示所有有效字构成的语言的RE $a_1|a_2|a_3|\cdots|a_n$。因为RE在选择运算下是封闭的，因此我们的结果一定是一个RE。我们对单一语法范畴的RE所能做的任何事情都同样可以运用到这个语言的所有字的集合的RE上。

连接的封闭性使得我们可以将简单的RE连接起来构建复杂的RE。这一性质似乎很显然而不重要。然而，它使我们可以系统地将多个RE组合起来。因此，封闭性确保：只要a和b都是RE，那么ab也是RE。因此，任意可以运用于a或b上的技术也可以运用于ab上；这包括从RE自动生成识别器的构建。 42

程序设计语言与自然语言的比较

词法分析凸现了程序设计语言不同于诸如英语或中文等自然语言的一个微妙之处。在自然语言中，单词的表示，即它的拼写或它的象形图与它的含义之间的关系不是显然的。在英语中，*are*是一个动词而*art*则是一个名词，尽管它们仅在最后一个字符不同。另外，并非所有字符的组合都是合法的。例如，*arz*与*are*及*art*仅有微小的不同，但是它不是正常英语用法中的单词。

英语扫描器可以运用基于FA的技术来识别潜在的单词，因为所有英语单词来自于一个受限字母表。然而，必须通过查阅某个字典来看这个潜在的单词是否是真实的单词。如果这个单词能归类为一个词性，那么这次查找就可以解决归类问题。然而，很多英语单词可以归类若干个词性。这样的例子包括*buoy*和*stress*；二者都可以是动词或名词。对于这样的单词，其词性依赖于上下文。在某些情况下，理解语法内容足以将这个单词分类。而在另外一些情况下，为了归类，我们需要理解我们所讨论的单词的含义以及它的上下文的含义。

相反，程序设计语言中的字几乎总是通过词法来描述的。因此，(0...9)(0...9)*中的任意字符串是正整数。[*a*...*z*]([*a*...*z*]|[0...9])*中的任意字符串是Algol标识符。字符串*arz*与*are*同样合法，不需要通过查找来证实它的合法性。固然，某些标识符也许被作为关键字而保留起来。然而，这些例外情况同样可以用词法来描述。不需要上下文。

在程序设计语言的设计中，这是一个有意的决定。使一个拼写表示惟一词性的做法简化扫描、简化分析，而且显然并没有丧失语言的表达能力。某些语言允许字带有双重的词性，例如，PL/I没有保留关键字。后来设计的语言放弃了这一观点，因为这一复杂化超出了它所能带来的语言灵活性。 43

克林闭包的封闭性使得我们能够描述特殊种类的带有有限模式的无穷集合。这是非常重要的；无限的模式对于一个实现器是毫无用处的。因为C语言的标识符[⊖]规则不限定名字的长度，这一规则允许字

⊖ 原书说的是Algol标识符，然而事实上，Algol限制标识符的长度不大于6，不符合这里的说明。——译者注

的无穷集合。这使程序员可以写出任意的有限长度的标识符。闭包允许我们不用指定最大长度就可以写出这样的集合的简明规则。

下一节将给出如何构建识别用RE来描述的语言的FA，同时给出构建FA接受的语言的RE的算法。这些构造法结合在一起建立RE和FA之间的等价性。RE在选择、连接和闭包下封闭的事实对这些构建至关重要。

RE和FA之间的等价性也表明另一种封闭性质。例如，给定一个FA，我们可以构建识别所有不在L(FA) 中的字w的FA，并称其为L(FA) 的补。为了构建这个补的FA，我们可以令所有非终结状态为终结状态，而所有终结状态为非终结状态。这表明，我们可以在我们的正则表达式的表记法中加入求补运算符而不改变正则表达式的性质。很多使用正则表达式的系统都有这样的运算符。

2.4 从正则表达式到扫描器以及从扫描器到正则表达式

我们研究有穷自动机的目标是从一组正则表达式出发自动地构建可执行扫描器。本节阐述把RE转化成FA的构造法，这个构造法适合于直接实现。我们还将阐述构建FA接受的语言所对应的RE的算法。图2-3展示出所有这些构造法之间的关系。

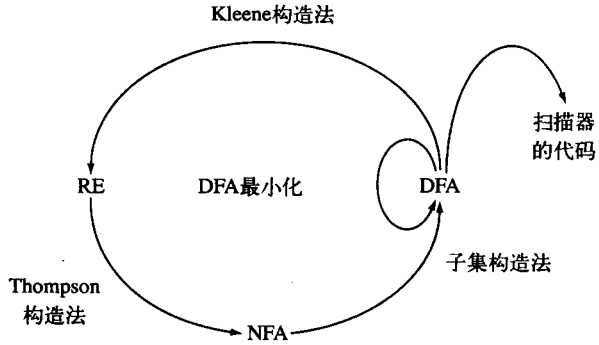
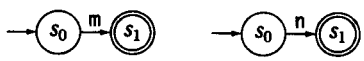


图2-3 构造法的循环关系

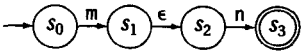
为了给出这些构造法，我们首先要介绍两个FA子类，*非确定性*（nondeterministic）FA（NFA）和*确定性*（deterministic）FA(DFA)。2.4.1节介绍NFA并论述其与DFA的不同。然后，我们通过三个步骤给出这一构造法。Thompson构造法从RE构筑FA。这一子集构造法构筑模拟NFA的DFA。2.4.4节给出的Hopcroft算法最小化DFA。2.4.5给出描述DFA接受的语言相对应的RE的Kleene算法。这一算法在通过完
44 成图2-3中构造法的循环关系建立FA与RE间的等价性中起着重要作用。2.5节给出若干实现FA的方案。

2.4.1 非确定性有穷自动机

根据正则表达式的定义，空字符串ε为一个RE。我们前面构建的FA中都不包含ε，但是有些RE却包含ε。在FA中ε扮演着什么样的角色呢？我们可以对ε使用转换来把FA结合起来构造对应于更复杂的RE的FA。例如，假设有对应于RE m和n的FA，分别称作FA_m和FA_n。

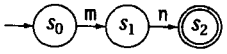


我们可以如下构造mn对应的FA：加入从FA_m的终结状态到FA_n的初始状态的对ε的转换，对状态重新标号，并以FA_n的终结状态为新FA的终结状态。



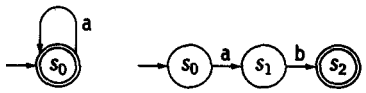
由于对ε的转换，我们必须稍加修改接受的定义，以允许在输入字符串中任意两个字符之间存在一个或多个ε转换。例如，在状态s_1处，FA不读取输入就可进行转换$s_1 \xrightarrow{\varepsilon} s_2$。这是一个较小的变化，它似乎很直观。

通过观察，我们可以看到，状态s_1和s_2可以组合起来，ε上的转换可以消去。

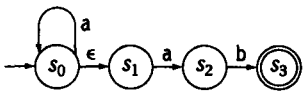


45

用ε转换把两个FA合并起来会使FA的运行模型复杂化。考虑语言a^*和ab的FA。



我们可以用ε转换把它们合并起来形成对应于a^*ab的FA。



实际上，对于这个FA，由于ε转换从s_0出发对字母a有两个不同的转换。它可以取转换$s_0 \xrightarrow{a} s_0$或两个转换$s_0 \xrightarrow{\varepsilon} s_1$和$s_1 \xrightarrow{a} s_2$。哪个转换是正确的呢？考虑字符串aab和ab。这个FA接受这两个字符串。对于aab，FA将取转换$s_0 \xrightarrow{a} s_0$，$s_0 \xrightarrow{\varepsilon} s_1$，$s_1 \xrightarrow{a} s_2$，以及$s_2 \xrightarrow{b} s_3$，对于ab，它取转换$s_0 \xrightarrow{\varepsilon} s_1$，$s_1 \xrightarrow{a} s_2$，$s_2 \xrightarrow{b} s_3$。

正如这两个字符串所示，从s_0出发对a的正确转换取决于跟在a后面的字符。在每一步，FA检查当前的字符。它的状态编码当前字符的左上下文，也就是它已经处理过的那些字符。因为FA必须在检查下一个字符之前完成转换，因此诸如s_0这样的状态违背我们的按序算法的行为的概念。含有诸如s_0这样的对一个字符有多个转换的状态的FA称作*非确定性有穷自动机*（nondeterministic finite automaton，NFA）。与此对应，在每一个状态都有惟一字符转换的FA称为*确定性有穷自动机*（deterministic finite automaton，DFA）。

为了使NFA有意义，我们需要描述它的行为的一组新的规则。基于历史原因，对NFA的行为有两个不同的模型。

1）每当NFA遇到非确定的选择时，如果存在导致输入字符串进入接受状态的转换，那么它就选择这个转换。这一使NFA带有智能的模型很有吸引力，因为（在表面上）它保留了DFA优秀的接受机制。

2）每当NFA遇到非确定的选择时，NFA复制自身以跟踪每一个可能的转换。因此，对于给定的输入字符，通过复制，NFA处于特定的状态集合。我们称这样的状态集合为NFA的*格局*（configuration）。 46
当NFA到达一个格局刚好耗尽了输入，而且一个或多个复制达到终结状态时，它接受这个字符串。尽管这一模型相对比较复杂，但是这个模型跟踪通过NFA的转换图的各个路径，直到成功或耗尽所有路径。

在以上两个模型中，有必要形式化NAF的接受标准。给定NFA(S，Σ，δ，s_0，S_F）和输入字符串$x_1x_2x_3\cdots x_k$，当且仅当在转换图中至少存在一条开始于s_0，结束于某个$s_k \in S_F$的路径，使得沿着这一路径的边的标签构成输入字符串（忽略标签为ε的边）时，该NFA接受该字符串。换句话说，第i条路径的标签必须是x_i。这一定义同时符合NFA的两个行为模型。

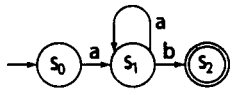
本节所给构造法的循环关系的一个结论是NFA和DFA等价。DFA是NFA的特殊情况。因此，NFA至少与DFA一样功能强大。任意的NFA可以用DFA模拟：这是2.4.3节中的子集构造法所确立的事实。它的思想很简单；但构造法要稍稍复杂一些。

考虑当一个NFA到达输入字符串中的某个点时的状态。在NFA行为的第二个模型下，这时NFA有有限个复制。相应的格局的数目是有界的；对于每一个状态，格局或者包含处于这个状态的复制，或者不

包含这样的复制。如果NFA有n个状态，它最多产生2^n个格局。

为了模拟NFA的行为，我们需要一个对NFA的每一个格局都有一个对应状态的DFA。因此，DFA的状态比原来的NFA的状态是指数增加。我们把N的所有子集记作2^N，称为N的幂集。因此，S_{DFA}与$2^{S_{DFA}}$一样大。注意，$2^{S_{DFA}}$是有限的。另外，这个DFA仍然对每个输入符号做一个转换。因此，模拟NFA的这个DFA的运行时间仍然按输入字符长度线性增加。使用DFA模拟NFA存在空间问题，但不存在时间问题。

因为NFA和DFA是等价的，我们应该可以构造a^*ab对应的DFA。下面是一个这样的DFA。



47 注意，a^*ab描述的字的集合与aa^*b描述的集合相同。这一FA的结构与RE aa^*b的结构类似。

2.4.2 正则表达式到NFA：Thompson构造法

从RE出发实现扫描器的第一步是从这个RE得到相应的NFA。Thompson构造法依据一个直截了当的想法。它有一个构建对应于单一字母RE的NFA模板，以及刻画RE运算符连接、选择和闭包的效应的NFA的转换。图2-4给出了对应于RE a和b的NFA，以及从对应于a和b的NFA构造对应于RE ab、$a|b$和a^*的NFA的转换。在对应于a和b的NFA处可以是任意的NFA[⊖]。

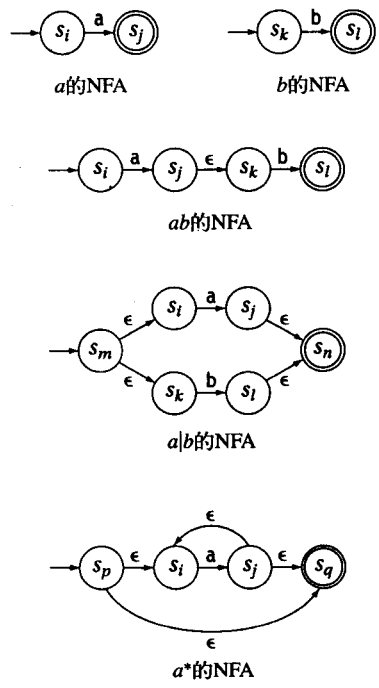


图2-4 对应于正则表达式运算符的小NFA

首先，Thompson构造法开始于对输入RE中的每个字符构建一个小的NFA。其次，它按优先性和括号所指定的顺序依次对这些小NFA做连接、选择和闭包对应的转换。

48 对于RE $a(b|c)^*$的例子，Thompson构造法首先构建对应于a、b和c的NFA。因为括号的优先权最高，它下一步构建对应于$b|c$的NFA。闭包的优先权高于连接的优先权，所以它构建对应于$(b|c)^*$的NFA。最后，它运用连接运算符来构建对应于$a(b|c)^*$的NFA。图2-5给出了这一系列转换。

这一构造法取决于RE的几个性质。它依赖于RE运算符与NFA上的转换之间的直接的对应关系。它把这一对应关系与RE的封闭性结合起来以确保转换生成正确的NFA。最后，它使用ε转换来连结对应于子表达式的NFA；这使得转换得以按简单的模式进行。

这一构造法得到的NFA有几个有用的性质。

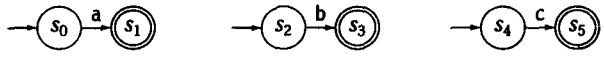
1）每个NFA有一个初始状态和一个终结状态。进入初始状态的惟一转换是初始转换。没有从终结状态出发的转换。

2）ε转换总是连结在这一过程早期得到的对应于部分RE的NFA的初始状态或终结状态。

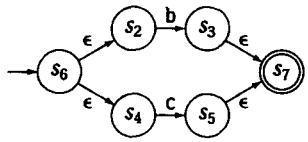
3）每个状态最多有两个进入的ε转换和两个出去的ε转换，而且最多有一个标签为字母表中符号的入边和一个标签为字母表中符号的出边。

这些性质简化这一构造法的实现。例如，这一构造法只需处理一个终结状态，而不需要反复处理子表达式对应的NFA的所有终结状态。

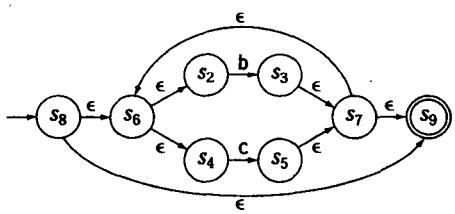
⊖ 实际上，这里的NFA有一定的限制。参见下文中关于由此构造法构造出的NFA的性质。——译者注



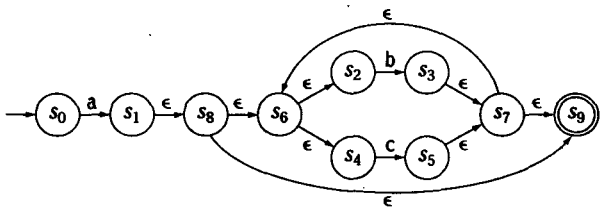
a、*b*和*c*的NFA



b|*c*的NFA



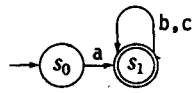
$(b|c)^*$的NFA



$a(b|c)^*$的NFA

图2-5　运用Thompson构造法构建对应于$a(b|c)^*$的NFA

注意，Thompson构造法所构建的对应于$a(b|c)^*$的NFA中的状态数目是如此之多。我们也许喜欢生成一个简单的NFA，例如如下所示的NFA。



我们可以直接消除由Thompson构造法所构建的NFA中的许多ε转换。这将大大缩小NFA的尺寸。然而，我们是在这一构造法的后期阶段将它们消除，所以我们将消去ε转换的一般算法作为练习留给读者。

49
~
50

2.4.3　NFA到DFA：子集构造法

Thompson构造法生成识别由RE描述的语言的NFA。因为NFA的两个运行模型都不适于显然、高效的实现，所以，下一步的工作是把NFA转换成具有简单而高效实现的DFA。从NFA(N，Σ，δ_n，n_0，n_F)出发，这一转换必须生成一个DFA(D，Σ，δ_D，d_0，D_F)。这一转换过程的关键步骤是从NFA得到D和δ_D。(NFA和DFA的字母表Σ相同，而d_0和D_F会在D和δ_D的构造过程中显现出来。) 构造这一DFA的算法称为子集构造法。图2-6给出这一算法的高级描述。

这一子集构造法构建一个集合Q，它的元素q_i是由N的子集所构成的状态集合，也就是说，$q_i \in 2^N$。

当这个集合被构造出后，每一个元素$q_i \in Q$将对应于DFA中的一个状态。这一构造法通过NFA对给定的输入能做出的所有转换的集合构造各个q_i。因此，集合q_i恰好是对应于特定的输入集合，NFA所能达到的状态的集合。每一个q_i代表NFA的一个有效格局。

```
q0 ← ε-closure(n0)
Q ← {q0}
WorkList ← {q0}
while (WorkList ≠ ∅)
  remove q from WorkList
  for each character c ∈ Σ
    t ← ε-closure(Delta(q,c))
    T[q, c] ← t
    if t ∉ Q then
      add t to Q and to WorkList
```

图2-6 子集构造法

为了构造Q，这一算法构造一个初始集合q_0，它是由n_0以及NFA通过只包含ε转换的路径从n_0所能达到的状态组成的。这些状态都是等价的，因为它们都能通过空输入达到。

这一代码使用函数ε-*closure*：$2^N \rightarrow 2^N$来从n_0构造q_0。它始于集合$n=\{n_0\}$，然后系统地在n中加入那些可以从n中的某个状态通过ε转换达到的状态。如果m是从n_0出发通过标签为abc的路径能够达到的状态的集合，那么ε-*closure* (m) 则包含从n_0出发通过标签为abcε^*的路径能够达到的所有状态。

51

这一算法使用第二个函数*Delta*：$2^N \times \Sigma \rightarrow 2^N$来计算NFA的状态集合的转换。*Delta*($q_i$, c) 把NFA的状态转换函数作用于q_i的每个元素。它返回NFA的状态的集合，计算对应于每个$n \in q_i$的$\delta_n(n, c)$的并。

为了构造Q的其余部分，这一构造法选出一个集合q_i并使用Delta去寻找从q_i开始沿着标签为c的转换能够达到的状态的集合。它计算这一集合的ε-closure，并为这个闭包指定一个临时名字t。它把从q_i到t的转换记录在一个表T中。如果t不属于Q，那么它就把t加入到Q中。对每一个$q_i \in Q$和每个符号$c \in \Sigma$重复这一过程。

*while*循环反复从工作列表中消去集合q_i并处理q_i。q_i代表NFA的一个格局。这一算法通过对Σ的元素进行迭代并运用*Delta*和ε-*closure*来寻找从q_i出发能够达到的格局。它把找到的新状态加入到Q和工作列表中。因为NFA至多有$|2^N|$个格局，因此，这一过程一定会终止。

遗憾的是，Q可以变得很大，最多可以有$|2^N|$个不同的状态。输入的NFA中的非确定性的程度决定状态扩张的程度。然而，结果的DFA仍然对每个输入字符做一个转换，与D的大小无关。因此使用非确定性来描述和构建NFA增加表示相应的DFA所需要的空间，但是不增加识别输入字符串所需要的时间。

1. 从Q到D

当这一算法结束时，它已构造了有效NFA格局的集合Q和记录Q的元素之间的可能转换的表格T。同时，Q和T形成模拟原来NFA的DFA的模型。从Q和T构建DFA是直观的。每个集合$q_i \in Q$生成一个代表状态$d_i \in D$。如果q_i包含NFA的终结状态，那么d_i是DFA的终结状态。转换函数δ_D可以直接从T通过从q_i到d_i的映射构造出来。最后，从q_0构造出的状态成为DFA的初始状态d_0。

2. 例子

考虑2.4.2节中给出对应于$a(b|c)^*$的NFA。这一NFA如图2-7的上部分所示。其中的状态已经重新编号。

52

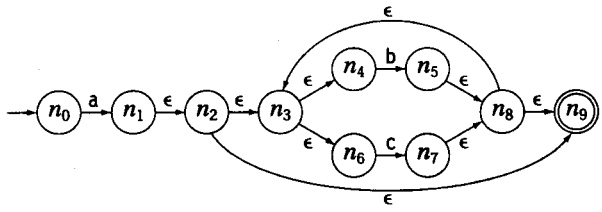
这一图中部的表格粗略给出了子集构造法的各步骤。在图2-7中，表格的第一列给出在一个给定的迭代中被处理的Q中的集合的名字。第二列给出新DFA中相应的状态名。第三列给出Q的当前集合中的NFA状态的名字的集合。后三列给出在相应状态下对于Σ中的每个字符的*Delta*值的ε-*closure*。

这一算法的步骤如下所示：

1）初始化集合q_0为ε-*closure*($\{n_0\}$)。第一次迭代计算ε-*closure*(*Delta*(q_0, a))、ε-*closure*(*Delta*(q_0, b)) 和ε-*closure*(*Delta*(q_0, c))。

各列给出对那些*Delta*计算所返回的集合运用ε-*closure*的结果。

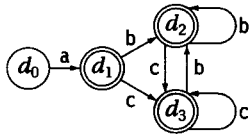
2）其次，算法在第一次迭代对q_1施行同样的处理。这生成两个集合，q_2和q_3。



a) $a(b|c)^*$的NFA（所有状态重新编号）

集合名	DFA状态名	NFA状态名	ϵ-closure(Delta(q, *))		
			a	b	c
q_0	d_0	n_0	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	– none –	– none –
q_1	d_1	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	– none –	$\{n_5, n_8, n_9, n_3, n_4, n_6\}$	$\{n_7, n_8, n_9, n_3, n_4, n_6\}$
q_2	d_2	$\{n_5, n_8, n_9, n_3, n_4, n_6\}$	– none –	q_2	q_3
q_3	d_3	$\{n_7, n_8, n_9, n_3, n_4, n_6\}$	– none –	q_2	q_3

b) 子集构造法的迭代



c) 结果DFA

图2-7　运用子集构造法从图2-5构建DFA

53

3）在第三次和第四次迭代中，算法尝试从q_2和q_3出发构造新状态。这生成与q_2和q_3相同的集合，如上表所示。

图2-7的最底部分给出结果DFA；它的状态由表中“DFA状态名”列给出，它的状态转换由生成这些状态的*Delta*运算给出。因为集合q_1，q_2和q_3全都包含n_9（NFA的终结状态），所以这三个集合都对应于DFA的终结状态。

3. 不动点计算

子集构造法是*不动点计算*（fixed-point computation）的一个实例，不动点计算是计算机科学中经常出现的一种特定计算类型。这些计算的特征是，将单调函数迭代应用于这一其结构已知的单调函数定义域中的一组集合。（一个函数$f: D \to D$是*单调函数*仅当对于任意的$x \in D$，$f(x) \geqslant x$，其中D是定义域。）当这些计算到达一个状态，它们在这个状态处的进一步的迭代产生相同的答案，也就是由这个算法产生的连续的迭代空间中的“不动点”时，这些计算终止。不动点计算在编译器构造法中扮演着重要的角色，我们将反复运用这样的计算。

不动点算法终止时的参数通常依赖于定义域的已知性质。对于子集构造法，因为$Q=\{q_0, q_1, q_2, \cdots, q_k\}$，其中每一个$q_i \in 2^N$，定义域$D$是$2^{2^N}$。因为$N$是有穷的，所以$2^N$和$2^{2^N}$也是有穷的。*while*循环把元素添

加到Q；它不能从Q中消除元素。（*while*循环是单调递增函数；比较运算符是$\supset$和$\supseteq$，定义域是D。）因为
54 Q至多有$|2^N|$个不同元素，所以*while*循环至多可以迭代$|2^N|$次。当然，它也可能更快地到达不动点并结束。

4. 脱机计算ε-closure

子集构造法的一个直观的实现可以通过跟踪NFA转换图中的路径计算ε-*closure*()。然而，我们也可考虑另外一种方法：对于NFA的转换图中的每个状态n计算ε-*closure*($\{n\}$)，它的脱机算法如下所示：

```
for each state n ∈ N
  E(n) ← {n}
  WorkList ← N
while (WorkList ≠ ∅)
  remove n from WorkList
  t ← { n } ∪ ⋃_{n -ε-> m ∈ δ_n} E(m)
  if t ≠ E(n) then
    E(n) ← t
    WorkList ← WorkList ∪ { k | k -ε-> n ∈ δ_N }
```

这也是一个不动点计算。

这个算法终止时的参数比图2-6的算法终止时的参数更复杂。当工作列表为空时这一算法终止。这个工作列表的初始值是N。每当集合发生变化时，算法把它在转换图中的前趋加到工作列表中。

$E(n)$集合是单调增加的。因此，每个$E(n)$集合至多可以变化$|N|$次。每当它发生变化，它的前趋就被加入到工作列表中。每个集合有固定数目的后继，这些后继可以将其添加到工作列表$|N|$次。因此，最终工作列表变空且计算终止。

为了得到一个时间上界，我们观察到每一个ε转换？$n \xrightarrow{\varepsilon} m$可以把$n$加入工作列表至多$|E(m)|$次。因为$|E(m)| < |N|$，且ε转换的数目不超过转换图中边的数目$|\delta_n|$，所以工作列表中结点的数目在整个算法中不超过$|\delta_n| \cdot |N|$。因为工作列表中的每一个结点需要*while*循环的一次迭代，上式也给出了迭代数目的上界。

2.4.4 DFA到最小DFA：Hopcroft算法

作为对RE→DFA变换的最后改进，我们可以加入最小化由子集构造法所构建的DFA的步骤。由子集
55 构造法构建的DFA可能有庞大的状态集合。尽管这不增加扫描字符串所需的时间，但是它的确增加内存中识别器的大小。对于现代计算机，内存的存取速度通常决定着计算的速度。较小的识别器占有较少的磁盘空间、随机存取存储器（RAM）空间和处理器的高速缓冲存储器空间。所有这些都可能是优势。

为了最小化DFA(D，Σ，δ，d_0，D_F)，我们需要一个识别两个状态是否等价的技术，在这里两个状态等价是指对于任意输入字符串这两个状态的行为相同。图2-8的算法把DFA中的这些状态基于它们的行为划分成等价类的集合。

这一算法构造出集合p_1，p_2，p_3，…，p_m的集合p，其中每一个p_i是含有一个或多个DFA状态的集合。之所以说p划分D，是因为：

1）每个$p_i \in P$包含一个或多个DFA的状态。

2）每个$d_i \in D$刚好是一个$p_j \in P$的成员。

综上所述，p中的集合覆盖D：$\cup_{1<i<m} p_i = D$。这些性质定义了D的一个划分。

为了最小化DFA，这一算法构造出原来DFA的状态的一个特定划分。它根据状态的行为把它们组合起来形成一个划分。对于任意的$p_i \in P$，p_i中任意两个状态d_i和d_j对于任意的输入字符串必须有相同

```
P ← { D_F, {D − D_F} }
while (P is still changing)
  T ← ∅
  for each set p ∈ P
    T ← T ∪ Split(p)
  P ← T

Split(S)
  for each c ∈ Σ
    if c splits S into s1 and s2
      then return { s1, s2 }
  return S
```

图2-8 DFA的最小化算法

的行为。为了最小化DFA，每个集合$p_i \in P$应该在行为等价的条件下尽可能大。 56

为了构造成代表最小DFA的划分，这一算法开始于一个初始的粗略划分，这一划分满足除行为等价性之外的所有性质，并且通过迭代细化这一划分来逐渐使其满足行为等价性。初始划分包含两个集合$p_0 = D_F$和$p_1 = \{D - D_F\}$。把终结状态隔离到p_0确保在最终划分中没有既包含终结状态又包含非终结状态的集合，因为这一算法从不把两个划分的集合组合在一起。

算法通过反复检查每一个$p_i \in P$来寻找不应在同一个集合p_i内的状态来细化初始划分。显然，这一算法不能跟踪DFA在每一个字符串上的行为。然而，它能够模拟给定状态对于单一输入字符的行为。这导致细化划分的一个简单条件：符号$c \in \Sigma$对每个状态$d_j \in p_i$必须产生相同的行为。

这一分离动作是理解这一算法的关键。为使d_i和d_j保留在同一个集合p_i中，它们必须对每个字符$c \in \Sigma$做等价的转换。也就是说，对任意$c \in \Sigma$，有$d_i \xrightarrow{c} d_x$和$d_j \xrightarrow{c} d_y$，其中d_x和d_y在同一个集合$p_t \in P$中。所有使$d_k \xrightarrow{c} d_z$，d_z不属于p_i的状态$d_k \in p_i$都不能与d_i和d_j在相同的划分集合内。同样地，如果d_k没有对于c的转换，那么它也不能与d_i和d_j在相同的划分集合内。

图2-9给出了这一动作的图示。这里，$p_1 = \{d_i, d_j, d_k\}$中的状态是等价的当且仅当对于所有$c \in \Sigma$，这些状态的转换把它们带到同一等价类中的状态。如图所示，对于c，每个状态有一个转换：$d_i \xrightarrow{c} d_x$、$d_j \xrightarrow{c} d_y$和$d_k \xrightarrow{c} d_y$。如果如左图所示，d_x、d_y和d_y在当前划分中的同一个集合内，那么d_i、d_j和d_k也应保留在同一个集合内，因此c不分离p_1。另一方面，如果如右图所示，d_x、d_y和d_z不全在同一个集合内，那么c分离p_1。此时，这一算法必须构造两个新集合$\{d_i\}$和$\{d_j, d_k\}$来反应对于以c开始的字符串会产生不同的结果这一可能性。如果状态d_i没有对于c的转换，那么应该产生相同的分离结果。 57

为了细化划分P，这一算法对每个$p \in P$和每个$c \in \Sigma$做检查。如果c分离p，这一算法从p构造出两个新集合并把它们加到T中。（算法也可以把p分离成两个以上的集合，每个集合对于c的行为都一致。然而，分离出一个行为一致的状态集合，而用p所有其余的状态组成另一个集合就可以了。如果后一个集合中的状态对于c的行为不一致，那么这一算法将在以后的迭代分离它。）算法重复这一过程，直到发现不能继续分离划分中的集合。

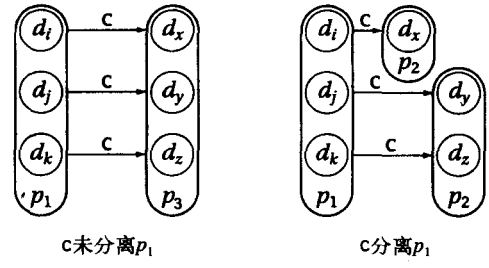
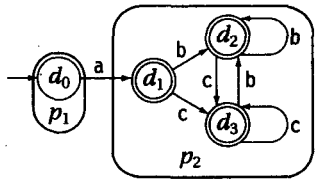


图 2-9

为了从最终的划分p构造新的DFA，我们可以对每个集合$p \in P$生成一个代表它的状态并把适当的转换加入到这些新代表状态之间。对于代表p_i的状态和字符c，如果某个$d_j \in p_i$有到某个$d_k \in p_m$的转换，那么对于c，我们加入一个从代表p_i的状态到代表p_m的状态的转换。根据这一构造法，如果d_j有这样的转换，那么p_i中的所有其他状态都有这样的转换；如果情况不是这样，那么在这一算法中，c将分离p_i。结果DFA是最小的；它的证明超出了我们的讨论范围。

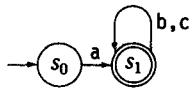
再次考虑由Thompson结构法和子集构造法所生成的对应于$a(b|c)^*$的DFA：



这一算法构造初始划分$p_1 = \{d_0\}$和$p_2 = \{d_1, d_2, d_3\}$。因为p_1只有一个状态，它不能再被分离。当这一算法检查p_2时，它发现对于a没在从p_2的任意状态出发的转换。对于b，每个状态有一个到p_2的转换。同样地，

c总是引发一个到p_2的转换。因此Σ中没有符号分离p_2而且划分$P = \{\{d_0\}\{d_1, d_2, d_3\}\}$最小化该DFA。

选择p_1和p_2的代表状态并加入转换生成如下的DFA:



58 这正是我们手工构造的DFA。经过最小化，自动技术产生了同样的结果。

这一算法是不动点计算的另外一个例子。P是有穷的；它至多包含$|D|$个元素。*while*循环分离P中的集合，但不把它们组合起来。因此$|P|$单调增大。当某次迭代不分离P中的集合时，这一循环终止。当DFA中的每个状态都有不同的行为时，是最坏的情况；在这最坏的情况中，当对每个$d_i \in D$，P都有不同的集合时，*while*循环终止。把这一算法运用于最小的DFA时发生这种情况。

2.4.5 DFA到正则表达式

如图2-3所示的循环结构中的最后一步是从DFA构造相应的正则表达式。Thompson构造法与子集构造法相结合表明DFA至少有与RE同样强大的构造化证明。本节给出Kleen构造法，对于任给的DFA，这一构造法构建表示该DFA接受的字符串集合的RE。这一算法证明RE与DFA同样强大。

把DFA的转换图考虑成一个带有标签边的图。得到描述这个DFA所接受的语言的RE的问题对应于DFA转换图上的路径问题。L(DFA) 中的字符串集合是由所有从d_0到d_i的路径的边标签组成的集合，其中$d_i \in D_F$。对任意带有循环转换图的DFA，这样的路径集合是无穷的。幸运的是，我们有克林闭包运算符来处理这种情况并能够合成由循环所生成的整个子路径集合。

59 图2-10给出计算这一路径表达式的一个算法。这一算法假设这个DFA的状态的编号由0到$|D|-1$，且d_0为初始状态。它生成表示沿着转换图中每对结点间的所有路径的标签的表达式。在最后一步，它把表示从d_0到某个属于D_F的终结状态d_i的路径的表达式组合起来。这样，它便系统地构造出所有路径的路径表达式。

```
for i = 0 to |D| − 1
  for j = 0 to |D| − 1
    R^0_ij = {a | δ(d_i, a) = d_j}
    if (i = j) then
      R^0_ij = R^0_ij | {ε}
for k = 1 to |D| − 1
  for i = 0 to |D| − 1
    for j = 0 to |D| − 1
      R^k_ij = R^{k−1}_ik (R^{k−1}_kk)* R^{k−1}_kj | R^{k−1}_ij
L = |_{s_j ∈ S_F} R^{|D|−1}_0j
```

图2-10 从DFA构造正则表达式

这一算法计算对应于对所有相关的i、j和k的记作R_{ij}^k的一系列表达式。而R_{ij}^k是描述转换图中所有从状态i到状态j且不通过标记比k大的状态的所有路径的表达式。在这里，通过（through）意味着既进入又离开某个状态，因此如果存在直接从1到16的边，那么$R_{1,16}^2$就是非空的⊖。

最初，这一算法设置R_{ij}^0为包含直接从i到j的所有边的标签，因为直接路径不通过结点。通过一系列迭代，算法通过在R_{ij}^{k-1}中添加从i到j通过k的路径来构建更长的路径。给定R_{ij}^{k-1}，从$k-1$构造k时要加入的路径正好是一条从i跑到k但不通过超过$k-1$的状态的路径，后面跟着若干条从k到其身的不通过超过$k-1$的状态的路径，再跟着一条从k到j的不通过超过$k-1$的状态的路径。也就是说，k上的每次迭代在R_{ij}^{k-1}中加入通过k的路径。

当k循环结束时，各个R_{ij}^k表达式统计转换图中的所有路径。最后一步计算开始于d_0结束于某个最终状态$d_j \in D_F$的路径的集合作为所求路径表达式。

⊖ 如果有一条从1到2且有一条从2到16的边，那么该表达式也非空。——译者注

2.4.6 将DFA作为识别器

至此，我们已开始开发了从一个RE构造DFA实现的机制。然而，编译器的扫描器必须识别出现在源语言文法中的所有语法范畴。因此，我们所需要的是能够处理对应于语言微语法的所有RE的识别器。给定对应于各语法范畴的RE r_1，r_2，r_3，…，r_k，我们就可以构造对应于整个语言范畴的RE $(r_1|r_2|r_3|\cdots|r_k)$。

如果我对这个RE进行处理，建立对应于子表达式的NFA，把它们用ε转换连结起来，合并状态，构建模拟这个NFA的DFA，然后把这个DFA转化成可执行代码，我们就可以得到一个识别与某个r_i相匹配的下一个字的扫描器。也就是说，当我们对某个输入调用扫描器时，它将一个一个地读入字符，如果它耗尽输入时处于终结状态，它就接受这个字符串。因为大多数实际程序都包含多个字，所以我们需要或者改造语言或者改造识别器。 60

在语言层面，我们可以认为每个字都结束于某个容易识别的分隔符，如空格或制表符。这并不吸引人。从字面上理解的话，这将需要我们用分隔符包围逗号及诸如＋和－这样的运算符，以及括号。

在识别器层面，我们可以改变DFA的实现和接受的概念。为了找到与RE相匹配的最长的字，DFA应该一直运行下去，直到它到达一个状态s，s对于下一个字符没有出去的转换。这时，实现必须决定它与哪一个RE匹配。此时有两种情况；第一种情况很简单。如果s是终结状态，那么这个DFA已在语言中找到一个字，并将报告这个字和它的语法范畴。

如果s不是终结状态，情况就更复杂。如果DFA在它通向s的路径上通过了一个或多个终结状态，那么识别器将报告它最后遇到的终结状态，这个终结状态对应于DFA所匹配的最长的关键字。为了实现这一点，实现可以跟踪最近的终结状态，以及与它对应的输入字符串中的位置。那么，当它达到一个没有合法转换的状态s时，它可以报告最近的终结状态，这一状态或者是s或者是前面的某个状态。如果识别器在通向s的路径上没有遇到终结状态，那么输入不始于一个有效字，并且这时识别器将向用户报告一个错误。

另外一个需要考虑的复杂问题是，DFA中的终结状态可能代表原来NFA中的几个终结状态。例如，如果词法描述包括关键字的RE和标识符的RE，那么如new这样的关键字可能与两个RE匹配。识别器必须决定返回哪个语法范畴：是标识符还是关键字new的单一类别。

大多数扫描器生成器工具允许编译器设计者指定模式间的优先级。当识别器与多个模式匹配时，它返回最高优先级模式的语法范畴。这一机制用简单的方法解决了这一问题。随着许多Unix系统发布的扫描器生成器lex基于正则表达式在列表中的位置指定优先级。第一个RE的优先级最高；最后的RE的优先级最低。

2.5 实现扫描器

直截了当的扫描器生成器以一系列RE为输入，为每个RE构造相应的NFA，用ε转换连结这些NFA（按Thompson构造法中对应于$a|b$的模式。）执行子集构造法生成相应的DFA，然后再最小化这一DFA。此后，扫描器生成器必须给出DFA的可执行代码。 61

本节覆盖从DFA到扫描器实现中出现的若干问题。前两小节给出从DFA构造可执行代码的若干策略。其余几小节描述在扫描器设计和实现中出现的若干低层次问题。

2.5.1 表驱动扫描器

为了把DFA转化成可执行程序，扫描器生成器使用一个框架扫描器，它以实现转换函数δ的表为参数。改变扫描器识别的语言需要修改这个表。回想一下我们最初的ILOC寄存器描述（$r[0...9][0...9]^*$）所对应的FA。

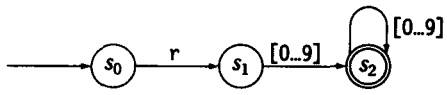


图2-11给出了这个DFA的表驱动实现。这一表代表转换函数δ。

```
char ← NextChar()
state ← s0
while (char ≠ eof)
    state ← δ(state,char)
    char ← NextChar()
if (state ∈ SF)
    then report acceptance
    else report failure
```

δ	r	0, 1, 2, 3, 4 5, 6, 7, 8, 9	其 他
S_0	S_1	S_e	S_e
S_1	S_e	S_2	S_e
S_2	S_e	S_2	S_e
S_e	S_e	S_e	S_e

图2-11 寄存器名的扫描器

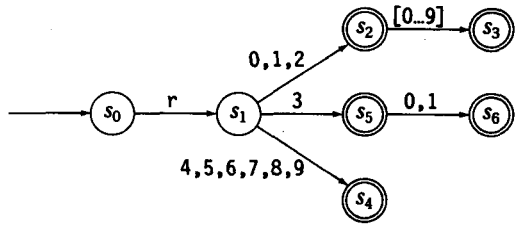
作表时，我们把数字0到9的各列压缩到一列中。这表明使用简单的方法，如合并相同列，就可以进行压缩。然而，使用压缩表需要对应于输入字符的进一步转换。通常这对每个字符需要一个额外的存储器引用；存储器的额外引用与大表格的代价权衡取决于目标机器。

62

这一表格可压缩程度取决于DFA的结构。例如，改进后的寄存器描述

$$r((0|1|2)([0...9]|\varepsilon)|(4|5|6|7|8|9)|(3(0|1|\varepsilon)))$$

生成一个七状态DFA，图2-12给出了它对应的表格。这张表比图2-11包含更多的信息；因此，它不能压缩那么多。



DFA

δ	r	0, 1	2	3	4...9	其 他
S_0	S_1	S_e	S_e	S_e	S_e	S_e
S_1	S_e	S_2	S_2	S_5	S_4	S_e
S_2	S_e	S_3	S_3	S_3	S_3	S_e
S_3	S_e	S_e	S_e	S_e	S_e	S_e
S_4	S_e	S_e	S_e	S_e	S_e	S_e
S_5	S_e	S_6	S_e	S_e	S_e	S_e
S_6	S_e	S_e	S_e	S_e	S_e	S_e
S_e	S_e	S_e	S_e	S_e	S_e	S_e

转换表

图2-12 实现改进后的寄存器描述

从DFA的描述生成相应表格是直截了当的。依次对于每个状态，代码检查每个出去的转换并释放表中的适当行。 63

2.5.2 直接编码扫描器

表驱动扫描器使用一个明确的变量*state*存放DFA的当前状态。*while*循环测试*char*是否是*eof*，计算新状态，读入下一个字符，并分支到循环的顶部。实现花费大量的时间来处理或测试这一状态。

表存取也导致一些额外的开销；特别地，地址计算需要一个乘法和一个加法（参见7.5节）。我们可以通过将状态的信息间接地编码到程序计数器中来避免大部分额外开销。在这一模型中，每个状态依据它的转换检查下一个字符并直接分支到下一个状态。这样就可生成一个带有复杂控制流的程序。图2-13给出按这种方法写成的框架识别器；它实现图2-11的FA。这一代码比表驱动代码更长、更复杂。因为对每个字符它导致的额外开销较少，因此它也应该比表驱动代码更快。

```
    goto s0
s0: char ← NextChar()
    if (char = 'r')
      then goto s1
      else goto se
s1: char ← NextChar()
    if ('0'≤char≤'9')
      then goto s2
      else goto se
s2: char ← NextChar()
    if ('0'≤char≤'9')
      then goto s2
      else if (char = eof)
        then report acceptance
        else goto se
se: report failure
```

图2-13 “r digit digit*”的直接编码识别器

当然，这一实现范例违反许多结构化程序设计的规则。像这样的小例子，代码也许容易理解。当RE描述变得更复杂且生成更多的状态和更多的转换时，增加的复杂性可能导致代码令人难以理解。如果我们使用自动工具直接从一系列RE生成这一代码，那么我们没有多少理由去阅读或调试这一扫描器代码。因此，更低的开销和更好的内存局部化所带来额外的速度使得直接编码成为吸引人的选择。 64

2.5.3 处理关键字

到此为止，对于输入语言的关键字，我们始终假设应该通过在生成DFA和识别器的描述中明确包含它们的RE来进行识别。很多作者提出了另外一个策略：让DFA把关键字分类成标识符，并测试每个标识符来决定它是否是关键字。

这一策略对于手工扫描器是有意义的。由于明确地关键字检测所增加的额外复杂性导致DFA状态数目的急剧增加。这会增加手工编码程序的实现负担。用一个适当的散列表（参见B.4节），每次查找的期望代价应该是一个常量。事实上，这一方案已被用作完全散列（perfect hashing）的一个典型应用。在完全散列中，对一个固定的键集合，实现器确保散列函数生成一个不含冲突的整数的紧致集。这可以降低每个关键字的查找代价。如果表实现考虑完全散列函数，仅一次探测就可以区分关键字和标识符。然而，如果探测反复失败，那么这一行为对非关键字比对关键字要更加不利。

如果编译器设计者使用扫描器生成器构建识别器，那么就是由这一工具来处理识别DFA中的关键字所增加的复杂性。这些额外增加的状态消耗内存但不消耗编译时间。使用DFA机制识别关键字避免对每个标识符的表查找。它还避免实现关键字表和支持函数的开销。在大多数情况下，把关键字识别掺入DFA比使用分离查找表更有意义。

2.5.4 描述动作

在构建扫描器生成器中，设计者可以允许对DFA的每个转换都做相应的动作，也可以只容许对DFA的终结状态做相应动作。它的选择对结果DFA的效率有很大的影响。例如，考虑我们早些时候给出的表示无符号整数的RE 0|[0...9][0...9]*。只容许对终结状态做动作的扫描器生成器将迫使用户重新扫描字符串来计算它的实际值。因此，扫描器需要重写读取已识别的字的各字符，实施适当的动作来将文本转换
65 成十进制的值。更糟糕的是，如果系统为这样的转换提供内置的机制，那么程序员将很可能使用它，即便在调用这一进行简单且常用的操作是需要额外的开销。(在Unix系统中，很多lex生成的扫描器包含调用sscanf()来精确执行这一功能的动作。)

然而，如果扫描器生成器允许对每个转换做动作，那么编译器设计者可以运用古老的汇编语言技巧来实现这一转换。在识别出第一个数字时，累加器被设置为这个数字的值。对每一个后继数字，累加器将其自身乘10并把新数字加进来。这一算法避免两次读取字符；识别器更快地产生结果，而且可以内联地使用已知的转换算法；它消除第一个解决方案中隐含的字符串处理的开销。(对于第一个解决方案，对于每一个转换，扫描器也可能把输入缓冲器中的字符拷贝到结果字符串中。)

一般地，扫描器应该避免对字符进行多次处理。编译器设计者在安排动作上自由度越大，就越容易实现避免拷贝字符并多次检查它们的高效且功能强大的算法。在典型编译器中，人们也许会忽视这种程度的低效性。然而，在基于DFA的应用中，如果模式匹配是决定时间效率的重要因素，那么加倍数据处理量是至关重要的。

表示字符串

扫描器把输入程序中的字归类成若干范畴。从功能的角度看，输入流中的每个字变成一个序对〈*word*, *type*〉，其中*word*是形成这个字的真实文本，而*type*表示这个字的语法范畴。

对于很多范畴，同时拥有*word*和*type*是多余的。字+、×和for只有一种拼写。然而，对于标识符、数和字符串，编译器将反复使用*word*。遗憾的是，很多编译器是用缺少对序对中的*word*部分的适当表示的语言写成的。我们需要一种紧凑且能进行快速对比的表示。

处理这一问题的常用的做法是，让扫描器生成一个的散列表（参见B.4节）来存放输入程序中的所有不同的字符串。于是，编译器或者使用这个“字符串表”中的字符串索引，或者使用指向这个字符串表中的它的储存映像的指针，作为这个字符串的代理。诸如字符常量的长度或数字常量的值和类型等字符串的信息可以通过这个表一次计算出来并快速引用。因为大多数计算机对整数和指针有高效存储表示，从而从根本上减少了编译器内部的内存使用量。对整数和指针代理使用硬件比较机制，这也简化了比较代码。

2.6 高级话题

本章在很大程度上讨论了扫描器描述和自动生成理论。这一理论的发展对程序设计语言的设计产生了重大的影响。因此大多数现代程序设计语言具有简单的词法结构，这样就可以使用基于DFA的识别器从而容易扫描。为了找出难以扫描的特征，我们必须考虑老式的语言。

FORTRAN 77有一些不易扫描的特征。空格的制表符没有太大意义；程序员可以加入它们或省略它们而不改变语句的含义。(Algo68有相似的规则。)程序员可以用anint、an int或an int来描述同一个变量。标识符最多含六个字符，而且语言依赖于这一性质来使某些结构得以识别。关键字不是保留字；

程序员可以把它们用作标识符。(PL/I有类似的规则。)因此，扫描器必须使用上下文关系对某些字分类。最后，FORTRAN支持若干种书写文字常数字符串的方式。

如图2-14所示的FORTRAN代码片段展示出扫描FORTRAN中的一些困难。标有标签10的语句包含三个字，integer、function和a。为了发现这一事实，扫描器必须运用标识符的六字符限制。语句20声明a和b为“参数”：可以代替文字常量的命名明显常数。以语句20为上下文，语句30中的character*(a-b)被展开成character*4。

Fortran代码	解　释
10 integerfunctiona	字符规则
20 parameter(a=6,b=2)	改变下一个语句
30 implicit character*(a-b)(c-d)	*(a-b)变成*4
40 integer format(10),if(10),d09e1	
50 format(4h)=(3)	format语句
60 format(4)=(3)	对format的赋值
70 do9e1=12	对do9e1的赋值
80 do9e1=1,2	do循环头部
	9e1扫描为9和e1
90 if(x)=1	对if的赋值
100 if(x)h=1	
110 if(x)120,130	
120 end	
c This is a comment	必须查看前面内容的注释
$file(1)	$在继续域
130 end	

图 2-14

语句50是一个format语句，因为序列4H是表明)=(3是一个文字字符串的前缀，称为Hollerith常量。它包含四个字：关键字format、(、字符串常量)=(3和)。语句60是一个给整数数组format的第四个元素赋值的语句。这与format在语法范畴的差异取决于4后面是否有字符h。⊖ 66～67

上下文还使语句70和80不同。语句70是对整形变量do9e1的赋值语句，而语句80中的逗号使它成为do循环的头。语句90、100和110在形式上是相似的。然而，在语句90中，if是一个标识符，而在语句100和110中它是关键字。

语句120似乎是一个表明过程结束的end语句。然而它后面跟着一个注释；第一列中的c表明一个注释行。注释后面的一行在第六列包含一个字符($)，这使它成为前一条语句的继续。这使file成为标签为120的语句的一部分。因为空格、换行和插入的注释行没有意义，120中的第一个字是endfile，它被一个内部注释分为两部分。(在一个语句的中间[或在一个关键字的中间]能够出现的注释行的数目没有限制。)语句130使得整个过程真正结束。 68

1. 两遍扫描器

很多FORTRAN编译器使用两遍扫描器。其中有一些，例如像Rice的PFC系统的扫描器有两个明确的遍。而另外一些，例如原始的Unix f77编译器和f2c编译器的扫描器，把某些扫描工作和简化工作作

⊖ 虽然FORTRAN的1977标准已经取消了Hollerith常量，但是几乎所有的FORTRAN 77编译器都支持这些常量，以向后兼容1966 FORTRAN标准。

为输入缓冲的一部分来做。这两种方式都多次扫描某些文本。

PFC扫描器的第一遍执行特殊的分析，把每一个语句分类，并重写文本以简化第二遍。正确把文本分解成字的关键是把赋值语句和注释与其他语句区分开来。赋值语句始于一个标识符。所有其他FORTRAN语句始于一个关键字。

为了发现赋值语句，PFC扫描器使用若干规则。一个操作符被认为是“自由的（free）”，如果它不在一个或多个括号的内部。赋值句必须包含一个自由的=。如果一个自由的斜线出现于第一个自由的=之前，那么它就不是赋值语句。语句中的自由的逗号表明这个语句不是一个赋值语句。

最后的规则是复杂的。如果扫描器在找到自由的=之前遇到了括号，而且括号表达式后面的第一个字符既不是=也不是(，那么这个语句不是赋值语句。如果括号表达式后面的字符是=，那么这个语句是一个赋值语句（正如例子中的语句90）。如果字符是(，那么它也许是子串表达式；在这种情况下，这个=必须迅速跟随第二个括号表达式。最后，如果这个括号表达式后面的字符既不是=也不是(，那么这个语句不是赋值语句，如例子中的语句100和110。

PFC扫描器的第一遍使用指明语句类型的标签标识每个语句。它删除所有注释。它发现字符串常量并使用统一且可识别的语法重写这些字符串常量。它展开参数（parameter）的每个使用。伴随这些改变，代码变得更容易扫描。

第二遍逐语句检查代码。它检查语句的类型并对语句的其余部分调用适当的识别器。Format语句的语法与其他语句有很大不同。

69 在扫描期间仍然需要有些上下文关系。例如，字符串do9e1是语句70中的一个字，而在语句80中是三个字。在处理语句80中的do之后，扫描器遇到9e1，并识别标签90和标识符e1，而不是识别一个浮点数9e1。

2. 使用右上下文

某些扫描器生成器包含描述右上下文的表记法。FORTRAN的一些困难可以通过在识别器中使用右上下文而得到解决。考虑do关键字的右上下文的RE。它必须识别一个标签，其后是一个标识符，一个=、一个数和一个逗号。

$$[0...9]^* \ ([A...Z] \mid [0...9])^* = [0...9]^*$$

如果把关键字do加到上面表达式的前面，它将与语句80匹配，但与语句70不匹配。[⊖]通过使这个RE的优先级高于标识符的RE的优先级，并且使识别器在最初的do的后面标示出匹配字符串的结尾，编译器设计者就可以使用基于RE的识别器寻找关键字do。类似的右上下文使识别器能够在语句80中把9e1作为一个标签和一个标识符识别，而不是当作一个浮点数来识别。

使用右上下文使得编译器设计者处理很多FORTRANR的复杂问题。然而，有一些问题使用这一机制也是不容易解决的。如语句120中的内部注释需要一个能够包含任意字符的无界右上下文，只要这个注释出现在以c为第一列字符的那一行。为了处理内部注释，我们可以在RE的任意两个符号之间加入表示注释的正则表达式。这样就使得RE变得很大且更不容易设计。我们可以通过缓冲输入的代码中的一个巧妙的机制识别注释和字符串；f77和f2c的扫描器就使用了这一机制。

与此相对应，参数展开引起的复杂问题可以推迟到分析器来处理。使用这一方法，扫描器把参数常量的使用作为标识符来识别。分析器对语言使用较宽松的文法。例如，文法将需要允许字符串长度的表达式包含含有标识符的表达式。（这一标准指出这样的表达式必须是一个无符号整数常量、一个值为正

⊖ 我们已通过假设循环索引变量的初始值是一个正整数简化了这一表达式。

整数的整数常量表达式或者是一个星号。）分析器将为这一表达式构建一棵小的表达式树，计算它，并把它替换成一个整数值。 70

2.7 概括和展望

搜索和扫描中的正则表达式的广泛运用是现代计算机科学的一个成功的故事。这些想法是作为形式语言和自动机理论的早期部分而发展起来的。作为一种精确描述恰好是正则语言的字符串集合的一种手段，它们被广泛运用从文本编辑，到网络过滤引擎，再到编译器的工具中。只要我们要识别字的有限集合，那么基于DFA的识别器就值得我们认真考虑。

大多数现代编译器使用已生成的扫描器。确定性有穷自动机的性质与编译器的需求非常匹配。识别一个字的代价与它的长度成正比。在精心的实现下，每个字符的开销很小。可以通过广泛运用的最小化算法减少状态数。状态的直接编码比表驱动解释器的速度更快。因为广泛可用的扫描器生成器很优秀，所以很少有理由去手工实现。

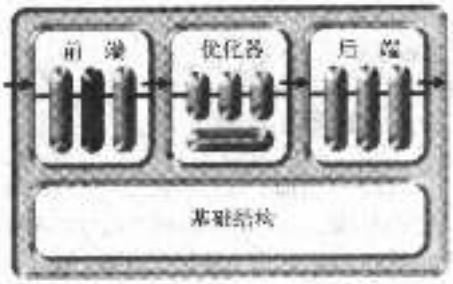
本章注释

将词法分析（即扫描）与语法分析分离开来最初是出于效率的问题。因为扫描的代价按字符数量成线性增长，其系数不大，把语法分析器的语法分析交给独立的扫描器来完成降低了编译的代价。高效分析技术的出现削弱了这一观点，但是由于在词法结构和语法结构之间存在着清晰的界线，我们仍然坚持构建扫描器。

因为扫描器构造法在建立实际的编译器中起的作用很小，我们尽量使本章简洁。因此，本章省略了很多关于正则语言和有穷自动机的定理。有强烈求知欲的读者也许对这些感兴趣。很多这方面的优秀教科书对有穷自动机、正则表达式以及它们很多有用的性质给出了更深的论述[184，222，307]。

Kleene（克林）[213]确立了RE与FA之间的等价关系。克林闭包和RE→DFA算法都以他的名字命名。McNaughton和Yamada给出了从RE到NFA的构造法[256]。本章的构造法是以Thompson的工作[321]为蓝图的。Thompson构造法是从为了实现一个早期的文本编辑器中的文本搜索指令得来的。Johnson首先把 71
这一技术应用于自动构建扫描器[197]。DFA最小化算法归功于Hopcroft[183]。这一算法在很多不同的问题中得到了应用，其中包括检测两个程序变量总是取相同值的时间[21]。

Preston Briggs指出了保留字的完全散列[248]对扫描器性能的负面影响。图2-14所示的FORTRAN代码来自于与E. K. Zadeck的一席谈话，当时他还是一个研究生。 72



第3章
语法分析

3.1 概述

编译器的语法分析器主要负责识别语法，也就是说，负责确定被编译程序是否是这一程序设计语言中的语法模型的合法句子。语法分析器工作于程序的抽象译本，即由扫描器产生的字和词性的流。如果字流形成一个合法的程序，那么语法分析器就建立这个程序的一个具体模型以供编译器的后期阶段使用。编译器的后期阶段分析这个具体模型，进行语义加工和翻译。这一语法分析的结果作为程序的编译器内部模型而记录下来。如果输入不形成合法程序，语法分析器将向用户报告存在的问题，并给出有用的诊断信息。

语法分析与扫描有很多共同之处。特别是，语法分析是另外一个这样的领域，在这一领域作为基础的数学的深入研究直接导致了很多种语言高效识别器或语法分析器的构建。有许多语法分析器结构的自动构建工具。本章学习如何使用上下文无关文法表示程序设计语言的语法，以及学习如何根据这样的文
73 法构建健壮且高效的语法分析器。本章的后半部分集中讨论三种特殊的语法分析技术。讨论的焦点是自顶向下的递归下降分析和自底向上的LR(1) 分析。对于典型的程序设计语言，这两种方法可带来高质量的语法分析器。这里介绍的自顶向下的递归下降分析导致手工编码分析器的系统构造法。递归下降分析器通常是紧凑且高效的。因为同样的观点直接导致表驱动LL(1) 分析器，所以本章简要地对其加以阐述。这里介绍的自底向上的LR(1) 分析导致高效、精确、表驱动分析器的自动结构法。所有这三种技术都用于商业编译器。

在实践中，科研文献中以及其他教科书中还有很多其他语法分析方法。我们聚焦于递归下降分析来解释自顶向下分析，这是因为代码与相应的文法之间具有直观关系，还因为使用这一关系可以容易地构建高效的递归下降分析器。为了解释自底向上分析，我们选择规范的LR(1) 分析器，因为它更具一般性。其他LR(1) 分析技术，特别是SLR(1) 和LALR(1)，可以作为规范的LR(1) 结构的简化或扩展来理解。

本章的3.2节介绍上下文无关文法，并探讨把程序设计语言转换成上下文无关文法形式时出现的若干问题。3.3节揭示自顶向下分析背后的思想，并描述通过在文法结构的框架下使用手写代码构建递归下降分析器的方法。3.4节介绍称之为移入归约分析的自底向上分析的风格，并发展构造这些语法分析器的精致的数学，并给出LR(1) 分析器的例子。3.5节给出如何得到构成LR(1) 分析器核心的分析表；在实践中，分析器生成器系统实现这些算法，使得编译器设计者无需手工执行这些算法。3.6节提出在设计和构建顶级语法分析器中出现的一系列实际问题。

3.2 表示语法

本质上，语法分析器是确定输入程序是否是源语言中的语法上合法句子的工具。为了回答这一问题，我们需要描述输入语言的形式机制和确定在这一形式描述的语言中的成员的系统方法。本节描述一种表
74 示语法的机制：书写形式文法的Backus-Naur形式的一种简单形式。本章的其余部分讨论确定由形式文

法描述的语言中的成员的技术。

3.2.1 上下文无关文法

为了描述程序设计语言的语法，我们需要能够刻画这样的语言的语法结构并导致高效识别器的表记法。第2章的正则表达式不足以刻画诸如Java或C等语言的语法。我们需要更强大的表记法。表示语法的一种传统的表记法是*文法*（grammar），它是从数学上定义哪些符号串是合法句子的一组规则。称为*上下文无关文法*（context-free grammar）的一类文法为我们提供这种能力。幸运的是，上下文无关文法的许多大子类具有导致高效识别器的性质。

一个上下文无关文法G是一组描述如何形成句子的规则；能够从G得到的句子集合称为*G定义的语言*（language defined by G），记作$L(G)$。举一个例子来帮助我们理解。考虑下面称为*SN*的文法：

$$\begin{aligned} \textit{SheepNoise} &\rightarrow \texttt{baa}\ \textit{SheepNoise} \\ &\mid \texttt{baa} \end{aligned}$$

第一个规则，或*产生式*（production），读作“*SheepNoise*可以派生字baa后面跟着另一个*SheepNoise*”。这里，*SheepNoise*是表示可以从这个文法派生出来的字符串集合的语法变量。我们称这样的语法变量为*非终结符*（nonterminal symbol）。我们称由这一文法所定义的语言中的字为*终结符*（terminal symbol）。第二个规则读作“*SheepNoise*也可以派生出字符串baa。”

为了理解*SN*文法与$L(SN)$之间的关系，我们需要指定如何运用这一文法中的规则来派生出$L(SN)$的句子。首先，我们必须确定SN的*目标符号*（goal symbol）或*开始符号*（start symbol）。目标符号表示$L(SN)$中的所有串的集合。因此，它不会是这一语言中的某个字。相反，它必须是以在这一语言中添加结构和抽象为目的而引入的语法变量中的一个。因为*SN*只有一个语法变量，所以*SheepNoise*必须是目标符号。

为了派生句子，我们从目标符号*SheepNoise*出发。在我们的原型字符串中选出一个语法变量α，选择一个文法规则$\alpha \rightarrow \beta$，然后用β替换所选出的α。重复这一过程，直到字符串不再包含语法变量；此时，这个字符串完全由字构成，它是这个语言中的一个句子。 75

Backus-Naur形式

计算机科学中表示上下文无关文法的传统表记法称为Backus-Naur形式即BNF。BNF使用尖括号将非终结符括起来，如<SheepNoise>，来表示非终结符。它在终结符下画下划线。符号::=表示“派生”，而符号|表示“也派生”。在BNF中，Sheep Noise（羊发出的声音）的文法变成

```
<SheepNoise>   ::=  baa <SheepNoise>
                |   baa
```

这与我们的文法*SN*完全等价。

BNF起源于20世纪50年代末到20世纪60年代初。尖括号、下划线、::=和|是考虑到书写语言描述的人在排版上的限制而使用的。（作为一个极端的例子，参看David Gries的《Compiler Construction for Digital Computers》，这本书是用标准行打印机印刷的[166]。）本书在排版上使用BNF的更新形式。我们使用斜体书写非终结符，用代码体书写终结符，用符号→表示“派生”。

在这样的派生过程的每一点，字符串是文法中的一组符号。这些符号可以是终结符，也可以是非终结符。当这样的字符串出现于一个合法派生中的某一步中时，也就是说，它可以从开始符号派生出来并

且从它可以派生出合法句子时，我们称其为句型（sentential form）。如果我们开始于*SheepNoise*，并使用这两个规则持续重写它，那么这一过程中的各字符串都是句型[⊖]。当我们到达只包含字（而不包含语法变量）的字符串时，那么这个字符串就是$L(SN)$中的一个句子。

对于*SN*，我们必须开始于字符串*SheepNoise*。使用规则2，我们可以将*SheepNoise*重写成baa。因为
76 这一句型只包含终结符，没有进一步重写的可能。因此，句型baa是这一文法定义的语言中的合法句子。我们可以用表格形式表示这一派生。

规　则	句　型
	SheepNoise
2	baa

我们也可以开始于*SheepNoise*并运用规则1派生出句型baa *SheepNoise*。接着，我们使用规则2派生出句子baa baa。

规　则	句　型
	SheepNoise
1	baa *SheepNoise*
2	baa baa

为方便表记，我们扩展符号→的解释；方便时，我们使用→*表示“用一步或多步派生出。”因此，我们可以写*SheepNoise*→*baa baa。

当然，我们用规则1取代规则2，生成更多个baa组成的字符串。反复运用这一规则模式，即使用规则序列 (*rule* 1)**rule* 2，将派生出由字baa的一次或多次出现组成的语言。这对应于在通常环境下羊发出的声音组成的集合。所有这些派生都有同样的形式。

规　则	句　型
	SheepNoise
1	baa *SheepNoise*
1	baa baa *SheepNoise*
	...and so on...
1	baa...baa *SheepNoise*
2	baa baa...baa

77

更加形式地，上下文无关文法*G*是一个四元组（*T*，*NT*，*S*，*P*），其中

*T*是语言中终结符或字的集合。在编译器中，终结符对应于词法分析中识别的字。终结符是文法句子的基本单位。

*NT*是出现于文法规则中的非终结符的集合。*NT*由规则中涉及的所有不在*T*中的符号组成。非终结符是文法设计者用于为规则提供抽象和结构的语法变量。

*S*是称为目标符号或开始符号的*NT*的特定成员。*G*描述的语言，记为$L(G)$，刚好包含那些从*S*派生出来的合法句子。换句话说，*S*表示$L(G)$的句子的集合。

*P*是产生式或重写规则的集合。形式地，*P*: $NT \rightarrow (T \cup NT)^*$，或*P*将*NT*中的元素映射到 $(T \cup NT)^*$中的

⊖ 在每一步，我们可以使用左部出现于句型的任意规则。这些规则没有通过上下文规定我们使用规则的优先级。由于这一原因，我们称这样的文法为上下文无关文法。当我们更加形式地定义上下无关文法时，将通过文法中对于每个规则的形式的限制来具体体现这一概念。

元素。注意，这一定义规定产生式的左部必须是单一的非终结符。这一限制确保文法是上下文无关的。

P中的规则刻画这一文法的语法结构。

我们可以直接从文法规则得到T、NT和P。对于SN，这些集合的取值如下所示：

$$T = \{\text{baa}\} \qquad NT = \{SheepNoise\}$$

$$S = SheepNoise$$

$$P = \left\{ \begin{array}{l} SheepNoise \rightarrow \text{baa } SheepNoise \\ SheepNoise \rightarrow \text{baa} \end{array} \right\}$$

在SN中，$SheepNoise$必须是开始符号S，因为NT只包含一个符号。一般地，发现开始符号是不可能的。例如，考虑文法：

$$\begin{array}{lcl} Paren & \rightarrow & (\quad Bracket \quad) \\ & | & (\qquad\quad) \end{array} \qquad \begin{array}{lcl} Bracket & \rightarrow & [\quad Paren \quad] \\ & | & [\qquad\quad] \end{array}$$

78

这一文法描述由交替出现的圆括号和方括号组成的平衡对构成的句子的集合。然而，我们不清楚最外面的一对括号应该是圆括号还是方括号。指定$Paren$为开始符号S时强制最外面的是圆括号。指定$Bracket$为开始符号S时强制最外面的是方括号。如果希望最外面的括号既可以是圆括号也可以是方括号，那么我们需要两个额外的产生式：

$$\begin{array}{lcl} Start & \rightarrow & Paren \\ & | & Bracket \end{array}$$

现在，这个文法有一个明确的目标符号$Start$，它不出现于任意产生式的右部。对文法进行操作的某些工具需要文法拥有不出现于任何产生式的右部的开始符号$Start$，它们使用这一性质简化发现S的过程。其他工具则简单地假设S是第一个产生式的左部。正如上述例子所提示的那样，我们总可以通过增加一个非终结符及若干简单的产生式来生成惟一的开始符号。

3.2.2 构造句子

为了探索上下无关文法的功能和复杂性，我们需要一个比SN更复杂的例子。考虑下面的文法：

1	*Expr*	→	*Expr Op* num
2		|	num
3	*Op*	→	+
4		|	-
5		|	×
6		|	÷

这一文法定义num和四个操作符+、-、×和÷上的表达式集合。使用这一文法作为重写系统，我们可以获生出很多表达式。例如，运用规则2生成一个只含有num的平凡表达式。使用规则序列1、3、2生成表达式num + num。

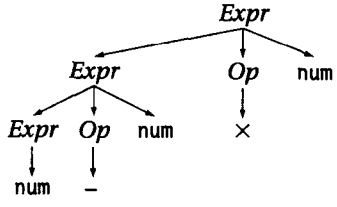
79

规　则	句　型
	Expr
1	*Expr Op* num
3	*Expr* + num
2	num + num

重写的序列越长就生成越复杂的表达式。例如，序列1，5，1，4，2派生句子num - num × num。

规　则	句　型
	Expr
1	*Expr Op* num
5	*Expr* × num
1	*Expr Op* num × num
4	*Expr* − num × num
2	num − num × num

我们可以用图形的形式描述这一派生过程。



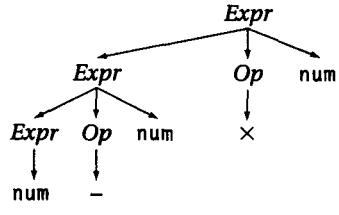
这一派生树或分析树，表示按某种顺序的派生的每一步。

至此，我们的派生总是展开字符串中的最右边的非终结符。也可以做其他选择；一个显然的选择是在每一点选择最左边的非终结符展开。使用最左的选择可以为同一句子生成不同的派生序列。对于num - num × num，最左派生是：

80

规　则	句　型
	Expr
1	*Expr Op* num
1	*Expr Op* num *Op* num
2	num *Op* num *Op* num
4	*num* − *num Op num*
5	num − num × num

这一“最左”派生使用与“最右”派生相同的一组规则，但是运用的顺序不同。对应的分析树如下所示：



这与最右派生的分析树相同！这一分析树表示运用于派生的所有规则，但是不描绘运用的顺序。

我们希望给定句子的最右派生（或最左派生）惟一。如果某个句子存在多个最右（或最左）派生，那么在派生的某点，我们可以对最右（或最左）的非终结符做不同的展开而得到同样的句子。这将生成多种派生，也可能是多棵分析树。因为后期的翻译阶段将把意义与分析树的详细形状联系起来，拥有生成惟一最右（或最左）派生的文法对翻译来说是至关重要的。

当且仅当在$L(G)$中存在拥有多个最右（最左）派生的句子时文法G是歧义的（ambiguous）。因为文

法结构是句子背后的含义的重要线索。歧义性通常是不受欢迎的。如果编译器不能确定一个句子的含义，那么它就不能把它翻译成有确定意义的代码序列。

程序设计语言的文法中出现的歧义结构的经典例子是很多类Algol语言中的if-then-else结构。if-then-else的直观文法可能是：

81

```
1  Statement  →  if Expr then Statement else Statement
2             |  if Expr then Statement
3             |  Assignment
4             |  ... other statements ...
```

这一片段表明else部分是可选的。遗憾的是，使用这一文法，代码片段：

```
if Expr1 then if Expr2 then Assignment1 else Assignment2
```

有两个不同的最右派生。这两个最右派生之间的区别很简单。使用缩进显示语句各部分之间的关系，我们有：

```
if Expr1                          if Expr1
   then if Expr2                     then if Expr2
           then Assignment1                  then Assignment1
           else Assignment2          else Assignment2
```

左边的版本中，$Assignment_2$受控于里面的if，所以当$Expr_1$为真且$Expr_2$为假时，执行$Assignment_2$。右边的版本把else部分与第一个if联系起来，因此当$Expr_1$为假时执行$Assignment_2$，与$Expr_2$的值无关。显然，派生之间的不同将导致被编译代码的不同行为。

为了消除这一歧义性，必须修改上面的文法来描绘决定哪个if控制else的规则。为了修改if-then-else文法，我们把它重写成：

```
1  Statement  →  if Expr then Statement
2             |  if Expr then WithElse else Statement
3             |  Assignment
4  WithElse   →  if Expr then WithElse else WithElse
5             |  Assignment
```

这一解决方案规定在if-then-else结构中的then部分可以出现的语句集合。这和原来的文法接受相同的句子集合，但是确保每个else非歧义地与一个特定的if匹配。它在文法中描绘了一个简单的规则：每个else绑定到最内部的未闭合if。对于上例，它只有一个最右派生。

82

规 则	句 型
	Statement
1	if *Expr* then *Statement*
2	if *Expr* then if *Expr* then *WithElse* else *Statement*
3	if *Expr* then if *Expr* then *WithElse* else *Assignment*
5	if *Expr* then if *Expr* then *Assignment* else *Assignment*

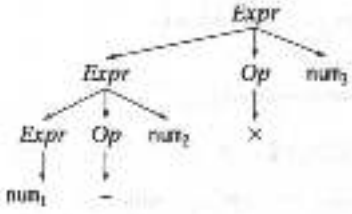
这一重写的文法消除了歧义性。

if-then-else的歧义性源于原来文法中的一个缺点。这个解决方案通过强制程序员容易记住的一条规则解决了这一歧义性。（为了完全避免歧义性，某些语言设计者通过引入elseif和endif重新构建if-

then-else结构。）在3.6.3节，我们将看到其他类型的歧义性以及处理这些歧义性的系统方法。

3.2.3 使用结构描述优先权

if-then-else文法的歧义性指出了文法结构与语句意义间的关系。然而，歧义性并非文法结构与语句意义相互关联的惟一之处。再次考虑简单表达式num - num × num的分析树。



为使讨论清晰，我们在number的实例中加入下标。计算这一表达式的一个自然的方法是使用简单的
83 后序树遍历。这将首先计算num_1-num_2，然后再将其结果乘以num_3，生成（num_2-num_2）× num_3。这一计算顺序与代数课程所教授的代数优先顺序相矛盾。标准的优先顺序将按如下顺序计算表达式：

$$num_1 - (num_2 \times num_3)$$

因为分析这一表达式的最终目标是生成实现这一表达式的代码，表达式的文法应该具有这样的性质：它构建的树的“自然”遍历计算生成正确的结果。

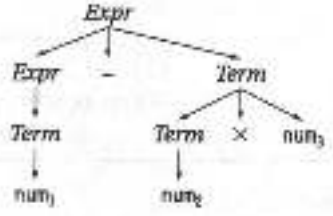
这一问题取决于这一文法的结构。这一文法按同一方式派生所有的算术操作符。为使分析树编码代数优先权，我们必须重新构建这一文法使其适应正确的优先权。

为了引入优先权，我们必须确定语言中的适当优先权级别。对于我们的简单表达式文法，我们有两个优先权级别：+和-的优先权低，而×和÷的优先权高。我们对每一个优先权级别指定不同的非终结符并分离出相应的文法部分。

1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* × num
5		|	*Term* ÷ num
6		|	num

在这里，Expr表示由+和-构成的低优先权级别表达式，而*Term*则表示由×和÷构成的高优先权级
84 别表达式。这一文法按与-和×相关的优先权级别一致的顺序派生num_1-num_2 × num_3：

规 则	句 型
	Expr
2	*Expr* - *Term*
4	*Expr* - *Term* × num_3
6	*Expr* - num_2 × num_3
3	*Term* - num_2 × num_3
6	num_1 - num_2 × num_3



这一分析树上的后序树遍历首先计算num$_1$ × num$_2$，然后再从num$_3$中减去这一结果。这实现了算术优先权的标准规则。注意，为强制优先权而增加的非终结符会增加分析树的内部结点。同样地，用具体的操作符替换Op的出现可以减少分析树的内部结点。

为把括号加入到这一文法中，我们需要另外一个优先级别。括号的优先权高于×和÷。这迫使我们在评估括号表达式前面和后面的操作符之前先评估括号内的表达式。因此，对于a × (b-c)，我们先评估b-c，再把评估值与a的值相乘。为了把这一功能插入到文法中，我们加入另外一个非终结符*Factor*。如图3-1所示的结果文法正确地表示+、-、×、÷和括号表达式的相对优先权。我们把这一文法称为*经典表达式文法*（classic expression grammar）。

1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* × *Factor*
5		|	*Term* ÷ *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	num
9		|	ident

图3-1 经典表达式文法

其他操作需要高优先权。例如，应该在标准算术运算之前运用数组下标。这样可以确保对x+y[1]这样的表达式，在把y[1]的值加到x上之前评估下标的值，而不是把1看成位置由x+y计算得来的某个数组的下标。同样地，诸如C或Java中的强制类型转换（type cast）这样的改变值的类型的操作的优先权高于算术的优先权，但低于括号和下标运算的优先权。

如果语言允许赋值出现于表达式的内部，那么这个赋值操作将有低优先权。低优先权确保代码在执行赋值之前完全评估赋值的左边和右边。例如，如果赋值（←）与加法有同样的优先权，那么假设是从左到右评估的话，表达式x←y+z在执行加法之前应把y值赋给x。 85

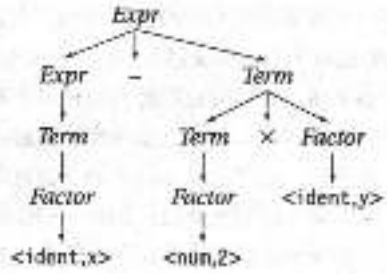
3.2.4 发现特定派生

给定文法G，我们已经看到如何发现$L(G)$中的句子。相反，编译器必须对给定的输入串推断出一个派生，或者确定这样的派生不存在。从特定输入句子出发构造派生的过程称为*语法分析*（parsing）[⊖]。

语法分析器以某种源语言写成的假定的程序作为输入。这一程序对语法分析器来说表面上是字及其语法范畴的流。因此，由经典表达式文法生成的语言的表达式x-2 × y可以如下表示：

⟨ident, x⟩ - ⟨num, 2⟩ × ⟨ident, y⟩

作为输出，语法分析器产生输入程序的一个派生或给出输入不合法程序的提示。因为对于非歧义语言，分析树等价于派生，所以我们可以认为语法分析是从输入串构建分析树的过程。因此，经典表达式文法的语法分析器可能为表达式x-2 × y构建如下分析树：



⊖ parsing通常译作分析，但本书为了区分作者常用的parsing和analysis而把parsing译作语法分析，但作为修饰词时译作分析，同时，单独的parser译作语法分析器，而带修饰词时译作分析器。——译者注

构建派生等同于构建分析树。分析树的根和叶子是已知的。分析树的根必须是表示开始符号S的结点。按照从左到右的顺序，分析树的叶子必须是与扫描器返回的字流相匹配的结点。这一问题的最难的部分是使用嵌入到文法中的语言结构连结根与叶子。有两种刚好相反的构建分析树的方法。

86 1）*自顶向下分析器*（top-down parser）从根开始构建并使其向着叶子的方向伸展分析树。在每一步，自顶向下分析器在树的较低边缘上选择某个非终符结点，并从这个结点开始向下方扩展这棵树。

2）*自底向上分析器*（bottom-up parser）从叶子开始构建并向根的方向伸展分析树。在每一步，自底向上分析器加入可以使部分树向上方扩展的结点。

无论哪一种方法，语法分析器都将对产生式的运用做出一系列选择。在语法分析中大多数智能上的复杂性依赖于做出这些选择的机制。

3.2.5 上下文无关文法与正则表达式的对比

为了理解正则表达式与上下文无关文法之间的差异，我们把经典表达式文法与下面的正则表达式进行对照：

$$((ident \mid num)\ (+ \mid - \mid \times \mid \div))^*\ (ident \mid num)$$

这一正则表达式和前述的上下文无关文法都描述相同的表达式集合。

为了弄清正则表达式与上下文无关文法之间的差异，考虑*正则文法*（regular grammar）的概念。正则文法与正则表达式有同样的表达能力，也就是说，它们都刚好能够描述所有正则语言。

一个正则文法是一个四元组$R=(T, NT, S, P)$，其成员的含义与上下文无关文法相同。然而在正则文法中，P中的产生式局限于两种形式，或者是$A \rightarrow a$或者是$A \rightarrow aB$，其中A，$B \in NT$且$a \in T$。对产生式形式的限制给正则文法带来另外一个名字，有时它们被称为*左线性文法*（left-linear grammar）。

与此相反，上下文无关文法允许产生式的右部包含（$T \cup NT$）的任意多个符号。因此，正则文法是上下文无关文法的真子集。同样的关系也存在于由正则文法描述的正则语言，和由上下文无关文法描述的上下文无关语言之间。正则语言是上下文无关语言的真子集。本章早前所用的文法*SheepNoise*是一个正则文法。

87 当然我们会问：是否存在可以用上下文无关文法表示但不能用正则文法表示的有趣的程序设计语言结构呢？现代程序设计语言的很多重要特征都陷入上下文无关文法与正则文法之间这一沟壑之中。这样的例子包括匹配括号，例如大括号和圆括号以及关键字配对（例如`begin`和`end`的配对）。同样重要的是，编译器设计者可以用上下文无关文法编码如操作符优先权和`if-then-else`结构这样的概念。分析经典表达式文法的表达式`x-2×y`导致描绘带有期望的评估顺序（×运算在-运算之前）的分析树。使用正则表达式识别同样的字符串不给出评估优先权的信息。

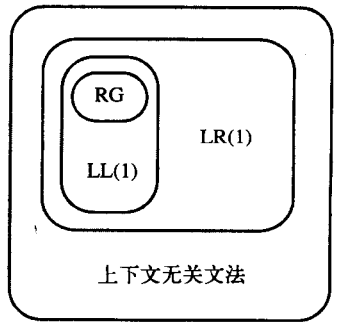
既然上下文无关文法可以识别由正则表达式指定的任意结构，究竟为什么要使用正则表达式呢？编译器设计者可以把语言的词法结构直接编码成它的文法。有两个因素使我们倾向于分离扫描器和分析器。第一个因素是，基于DFA的识别器效率高。它们所花的时间与输入字符串的长度成正比。使用合理的实现技术，甚至渐近复杂度的单位字符处理常量，即渐近复杂度的系数也很小。能够处理上下文无关文法的适当子集的分析器比扫描器要复杂得多，对每个输入符号的开销也高。第二个因素是，扫描器使用简化的输入文本，这大大减小了上下文无关文法的复杂度。例如，扫描器一般会发现注释，并把它们从输入流中清除出去。想像一下，扩展经典表达式文法使其允许在任意两个终结符号之间存在注释；在扫描器中识别并清除它们就简单得多了。对于文本字符串也存在相同的问题。

因此，为了高效和方便，编译器设计者使用基于DFA的扫描器。把微语法移入上下文无关文法将扩大文法，增加派生的长度并使前端变得更慢。一般地，正则表达式被用于字的分类，匹配正则模式，清

除具有内部词法结构的对象。当需要高层次结构，如匹配括号匹配透露结构，或在上下文间进行匹配时，上下文无关文法是绝好的工具。

上下文无关文法及其语法分析器分类

我们根据文法分析的难易程度对上下文无关文法的集合进行层次划分。这一层次划分有很多等级。本章涉及其中的四个等级，分别是任意的上下文无关文法、LR(1) 文法、LL(1) 文法和正则文法。这些集合存在嵌套的包含关系：上下文无关文法包含LR(1) 包含LL(1) 包含RG，如下图所示。



对任意的上下文无关文法进行语法分析需要比其受限集合LR(1) 或LL(1) 更多的时间。例如，分析上下文无关文法的Earley算法的最坏情况下的时间复杂度是$O(n^3)$，其中n是输入流中字的数量。当然，实际运行时间也许会更好些。编译器设计者历来都回避“普遍的”技术，因为我们已经认识到它们并不是很有效。

LR(1) 文法包含非歧义上下文无关文法的一个大子集。LR(1) 文法可以使用自底向上的分析，从左到右的线性扫描当前输入符号，最多向前看一个字的方式进行语法分析。有若干从文法得到LR(1) 分析器的算法。自动化这一过程的广泛可用的工具已使LR(1) 分析器成为“大家喜爱的分析器。”

LL(1) 文法是LR(1) 文法的一个重要子集。LL(1) 文法可以使用自顶向下的语法分析，从左到右的线性扫描，向前看一个字的方式进行分析。LL(1) 文法的两个不同类型的分析器很受欢迎，手编代码递归下降分析器和称为LL(1) 分析器的表驱动分析器。很多有趣的程序设计语言都容易使用LL(1) 文法来表示。

正则文法刚好精确描绘那些可以用DFA识别的语言。编译器构造中正则文法的主要用途是描述扫描器。

几乎所有程序设计语言的结构都可以表示成LR(1) 形式（通常是LL(1) 形式）。因此，大多数编译器使用基于上下文无关文法的这两个受限类中的快速分析算法。

3.3 自顶向下分析

自顶向下分析器从分析树的根开始，并系统地向下扩展分析树直到分析树的叶子与扫描器返回的已分类字匹配。在每一点，上述过程考虑已部分建立起来的分析树。它在这棵树较低的边缘处选择一个非终结符，并通过加入对应于那个非终结符的某个产生式的右部的子结点来扩展这个非终结符。它不扩展终结符的边缘。这一过程一直进行下去，直到下面两种情况之一发生：

88 ~ 89

a）分析树的边缘只包含终结符号，并且输入流已被耗尽；

b）在已部分建立起来的分析树的边缘与输入流之间出现明确的不匹配。

在第一种情况下，分析成功。在第二种情况下，存在两种可能性。在这一过程中的早前的某一步上，分析器也许选择了一个错误的产生式，在这种情况下回溯可以带来正确的选择。另一种可能是，输入串不是分析语言中的合法语句，在这种情况下回溯会失败，而且分析器应该向用户报告这个语法错误。

一个关键性的观察结果使自顶向下分析高效：*存在上下文无关文法的一个大子集，对此从不需要回溯*。本节的后半部分将介绍有助于把任意的文法转化成无回溯文法或预测文法的技术。然而，首先我们

介绍一个例子来展示自顶向下分析背后的思想，并说明非限制上下文无关文法引发的一些问题。

3.3.1 例子

图3-2给出一个自顶向下分析的简单算法。这一过程完全作用于分析树的较低边缘：这样的边缘总对应于句型。在每一步，我们选择扩展最左边的非终结符。这对应于最左派生，因为这一分析器按照扫描器产生字的从左到右的顺序考虑这些字。

```
word ← NextWord()
root ← start_symbol
node ← root
loop forever
  if node ∈ T & node matches word then
    advance node to the next node on the fringe
    word ← NextWord()
  else if node ∈ T & node does not match word then
    backtrack
  else if node ∈ NT then
    pick a rule "node→β"
    extend tree from node by building β
    node ← leftmost symbol in β
  if node is empty & word is eof then
    accept and exit the loop
  else if node is empty & word is not eof then
    backtrack
```

图3-2 最左自顶向下分析算法

为了理解自顶向下分析算法，考虑经典表达式文法中这一分析器识别表达式x-2 × y的步骤。这一文法的目标符号是*Expr*；因此，分析器开始于一个以*Expr*为根的树。为了展示分析器的行为，我们对派生
90 的表格表示进行扩展。最左边的一列给出达到每个状态所使用的文法规则；中间一列给出部分构建起来的分析树的较低边缘，这是最新得到的句型。在右边，我们加入了输入流的一个表示。↑表明扫描器的位置；它处于当前输入符号之前。我们在规则列中加入两个动作→和←，分别用来表示输入指针的前进和产生式集中的回溯。语法分析器可能采取的前几个移动是：

规 则	句 型	输 入
	Expr	↑x - 2 × y
1	*Expr* + *Term*	↑x - 2 × y
1	*Expr* + *Term*+ *Term*	↑x - 2 × y
1	*Expr* + *Term*+ ···	↑x - 2 × y
1	···	↑x - 2 × y

语法分析器开始于开始符号*Expr*，使用文法中的第一个产生式的右部展开它。这产生边缘*Expr* +
91 *Term*。为了生产最左派生，语法分析器必须展开这一边缘的最左非终结符，此时它仍是*Expr*。如果语法分析器仍旧选择规则1，这将导致一个无限的展开序列，其中的每个展开都不再带来任何进展。这对表达式文法的形式提出了问题。

如果在某个产生式的右部的第一个符号与它的左部的符号相同，那么这个产生式是*左递归的*。左递归也可以间接地发生，正如我们将在3.3.3节中看到的那样。带左递归的文法可以使自顶向下分析器永远

展开下去而不带来任何进展。

为了绕开左递归问题并展示自顶向下分析器的另一个潜在问题，我们让分析器以任意一种方式选择产生式。实际的实现无疑会用确定且一致的方式做出这些选择。例如，分析器也许尝试着把*Expr*重写成ident。这生成展开序列：规则3（*Expr*→*Term*）、规则6（*Term*→*Factor*）和规则9（*Factor*→*ident*）。

规　则	句　型	输　入
–	*Expr*	↑x - 2 × y
3	*Term*	↑x - 2 × y
6	*Factor*	↑x - 2 × y
9	ident	↑x - 2 × y
→	ident	x ↑- 2 × y

这时，边缘的最左符号是终结符，所以分析器检查这个符号是否与存放在*word*中的当前的输入符号相匹配。因为它们相匹配，它在边缘向右前进一个符号，并通过调用*NextWord*推进输入流。遗憾的是，分析树的边缘结束于ident，而输入流中的当前字是-。这一不匹配表明到此所采取的步骤无法带来一个有效的语法分析。或者是语法分析器在某个早前的展开中做了不正确的选择，或者输入字符串不是表达式文法的合法句子。

语法分析器通过回溯处理这种情况。如果语法分析器正在做系统的选择，那么语法分析器将取消最新近的动作，在此是通过规则9所做的展开，并对*Factor*尝试其他可能性。当这些都失败时，语法分析器取消规则6所做的展开，并对*Term*尝试其他可能性。最后，语法分析器将对规则3所做的展开重新尝试其他选择，并发现第一步应该是使用规则2的展开*Expr*→*Erpr*-*Term*。

从这一点开始，语法分析器可以向前工作，运用规则序列3、6和9从边缘的第一个位置的*Expr*派生出ident。将边缘和输入流向前移动，语法分析器发现输入流中的记号-与边缘的-相匹配。 92

规　则	句　型	输　入
–	*Expr*	↑x - 2 × y
2	*Expr* - *Term*	↑x - 2 × y
3	*Term* - *Term*	↑x - 2 × y
6	*Factor* - *Term*	↑x - 2 × y
9	ident - *Term*	↑x - 2 × y
→	ident - *Term*	x ↑- 2 × y
→	ident - *Term*	x - ↑2 × y

在此，语法分析器将通过规则序列4、6和8展开去与num匹配，并在边缘处留下适当的右上下文。语法分析器能够使边缘与num匹配，然后前进并与 × 匹配。最后的展开使用规则9把ident放置在边缘处，与最后的输入符号匹配。

规　则	句　型	输　入
4	ident - *Term* × *Factor*	x - ↑2 × y
6	ident - *Factor* × *Factor*	x - ↑2 × y
8	ident - num × *Factor*	x - ↑2 × y
→	ident - num × *Factor*	x - 2↑ × y
→	ident - num × *Factor*	x - 2 × ↑y
9	ident - num × ident	x - 2 × ↑y
→	ident - num × ident	x - 2 × y↑

在此，较低的边缘只包含终结符，而且输入已经用尽。在语法分析器中，*node*是空结点且*word*是*eof*，所以分析器报告成功并终止。

（注意，在语法分析过程中的很多点也许会做出错误的展开，引发更多的回溯。为简明起见，我们没有这样做。然而，读者应该考虑如果我们按确定的方式选择规则，例如按规则序号递增的顺序选取规
93 则的话，将会出现多少次回溯。）

3.3.2 自顶向下分析的复杂因素

正如前面的例子所展示的那样，若干问题可能使自顶向下分析器的使用复杂化。左递归文法引发终止问题。选择错误的展开必然导致回溯。一般地，关键问题是在每步都要确保语法分析器能够选择出正确的产生式来展开分析树边缘的生长。

书写无回溯文法需要小心。文法必须避免左递归。文法还必须确保单一向前看符号在每步足以确定正确的展开。以下两小节详细讨论这些问题。

3.3.3 消除左递归

左递归文法可能在不生成终结符号的情况下展开边缘，从而使确定的自顶向下分析器产生无穷循环。因为只有当边缘上的终结符号与当前输入符号之间不匹配时引发回溯，所以分析器不能从由左递归导致的展开中恢复。

幸运的是，我们可以机械地修改一个文法以消除左递归。对于如下图左边所示的直接左递归，我们如下图右边所示的那样使用右递归重写它。

$$\begin{array}{lcl} Fee & \rightarrow & Fee\,\alpha \\ & | & \beta \end{array} \qquad\qquad \begin{array}{lcl} Fee & \rightarrow & \beta\,Fie \\ Fie & \rightarrow & \alpha\,Fie \\ & | & \epsilon \end{array}$$

这一转换引入一个新的非终结符*Fie*，并把递归转移到*Fie*上。它还加入规则$Fie \rightarrow \varepsilon$，其中$\varepsilon$表示空字符串。在分析算法中$\varepsilon$产生式需要仔细的解释。如果语法分析器使用规则$Fie \rightarrow \varepsilon$进行展开，那么效果是沿着分析树的边缘使当前节点*node*到达下一个位置。

94 在表达式文法中，直接左递归出现于*Expr*和*Term*的产生式中。

原规则			变换后的规则		
Expr	→	*Expr* + *Term*	*Expr*	→	*Term Expr′*
	|	*Expr* - *Term*	*Expr′*	→	+ *Term Expr′*
	|	*Term*		|	- *Term Expr′*
				|	ϵ
Term	→	*Term* × *Factor*	*Term*	→	*Factor Term′*
	|	*Term* ÷ *Factor*	*Term′*	→	× *Factor Term′*
	|	*Factor*		|	÷ *Factor Term′*
				|	ϵ

把这些替换放回到经典表达式文法导致图3-3所示的这个文法的右递归变形。新文法与经典表达式文法描述相同的表达式集合，但是它使用的是右递归。它对自顶向下分析器很有用。

这一转换消除直接左递归。诸如$\alpha \rightarrow \beta$、$\beta \rightarrow \gamma$和$\gamma \rightarrow \alpha\delta$等规则链结合会生成$\alpha \rightarrow^{+} \alpha\delta$的情况，所以也可

能出现间接的左递归。这样的间接左递归不总是很明显：它可能产生于一个长的产生式链，从而被隐蔽起来。

为了把这些间接左递归转换成右递归，我们需要一个更系统的检查方法。图3-4给出可以达到这一目标的算法。这一算法假设原来的文法没有循环（$A \rightarrow^+ A$）和ε产生式（$A \rightarrow \varepsilon$）。

这一算法为非终结符强加一个任意的顺序。外部循环就以这一顺序处理所有非终结符。而内部循环对于每个$A_i \rightarrow A_j\gamma$（$j<i$）型的产生式，展开其右部的$A_j$。这样的展开可能导致间接的左递归。为了避免这样的左递归，这一算法用左部是A_j的所有可能产生式的右部取代A_j的出现。也就是说，如果内部循环发现一个产生式$A_i \rightarrow A_j\gamma$，且$A_j \rightarrow \delta_1 | \delta_2 | \cdots | \delta_k$，那么它用产生式集合$A_i \rightarrow \delta_1\gamma | \delta_2\gamma \cdots | \delta_k\gamma$取代前面的产生式。这最终把每个间接左递归转换成直接左递归。在外循环的最后一步使用前述的简单转换方法把任意开始于A_i的直接左递归转换成右递归。因为只有在最后才加入新的非终结符，并且它只包含右递归，所以循环可以忽视它们：它们无需检查和转换。

```
1   Expr    →   Term Expr'
2   Expr'   →   + Term Expr'
3           |   - Term Expr'
4           |   ε
5   Term    →   Factor Term'
6   Term'   →   × Factor Term'
7           |   ÷ Factor Term'
8           |   ε
9   Factor  →   ( Expr )
10          |   num
11          |   ident
```

图3-3 经典表达式文法的右递归变形

```
assume the nonterminals are ordered arbitrarily
  A1, A2, ..., An
for i ← 1 to n
  for j ← 1 to i - 1
    if ∃ a production Ai → Ajγ then
      replace it with one or more productions
      that expand Aj
  use the direct transformation to eliminate
    any immediate left recursion on Ai
```

图3-4 一般左递归的消除

考虑外部循环的循环不变量可以使这一过程更清晰。在第i次外部循环迭代

$\forall k < i$，对$l < k$不存在右部含A_l的展开A_k的规则

在这一过程的末尾（$i=n$时），通过反复使用内部循环，所有间接左递归都已被消除，在每次迭代的最后一步，所有直接左递归都被消除。

3.3.4 消除回溯

使用右递归表达式文法，自顶向下分析器能够不回溯而处理字符串x−2 × y。事实上，右递归表达式文法避开所有对回溯的需要。为了看到这一点，考虑语法分析器是如何通过回溯撤回动作的。

当分析器选择展开一个部分构建起来的分析树的较低边缘所用的产生式时，就会发生至关重要的选择问题。当分析器试图展开某个非终结符A时，它必须选择一个型为$A \rightarrow \beta$的规则，如图3-2所示的算法随机选取这样的规则。然而，如果语法分析器总能够选出适当的规则，那么它就可以避免回溯。

在右递归表达式文法中，语法分析器总是可以通过将输入流中下一字同对应于边缘处最左非终结符的右部进行比较而做出正确的选择。根据这种规律，对于右递归表达式文法，我们可以如下构建x−2 × y的分析。

规 则	句 型	输 入
	Expr	↑ x - 2 × y
1	*Term Expr'*	↑ x - 2 × y
5	*Factor Term' Expr'*	↑ x - 2 × y
11	ident *Term' Expr'*	↑ x - 2 × y
→	ident *Term' Expr'*	x ↑ - 2 × y
8	ident *Expr'*	x ↑ - 2 × y
3	ident - *Term Expr'*	x ↑ - 2 × y
→	ident - *Term Expr'*	x - ↑ 2 × y
5	ident - *Factor Term' Expr'*	x - ↑ 2 × y
10	ident - num *Term' Expr'*	x - ↑ 2 × y
→	ident - num *Term' Expr'*	x - 2 ↑ × y
6	ident - num × *Factor Term' Expr'*	x - 2 ↑ × y
→	ident - num × *Factor Term' Expr'*	x - 2 × ↑ y
11	ident - num × ident *Term' Expr'*	x - 2 × ↑ y
→	ident - num × ident *Term' Expr'*	x - 2 × y ↑
8	ident - num × ident *Expr'*	x - 2 × y ↑
4	ident - num × ident	x - 2 × y ↑

97

前两个展开是显然的：在这两种情况下，文法只有一个展开最左非终结符的规则。扩展*Factor*到ident需要一个选择；检查输入流中当前字使选择明确。在此，语法分析器面临着*Term'*的展开，且-为当前输入字。规则6和规则7不匹配，所以语法分析器选择规则8。现在，语法分析器面临着*Expr'*的展开以及输入字-。这与规则3匹配，所以语法分析器使用规则3的右部展开它，使用输入符号引导选择。这一过程继续下去，直到语法分析器达到它的接受状态。

我们可以形式化右递归表达式文法无回溯的性质。在分析过程的每一点，展开的选择是显然的，因为最左非终结符的每种选择会导致一个不同的终结符，对照输入流中的下一个字和那些可选的右部就可显露出正确的展开。

这一直观想法很容易刻画。如下定义$TUNTU\{\varepsilon\}$上的FIRST集合。对于符号α，定义FIRST(α)为从α派生出的串中的作为第一个符号的字所组成的集合。考虑*Expr*的规则：

```
2   Expr'  →  + Term Expr'
3          |  - Term Expr'
4          |  ε
```

这些产生式都有惟一的FIRST集合。对于终结符号+和-，它们的FIRST集合只包含一个元素，那就是这个符号本身。ε产生式引发一个很困难的问题。这一产生式不能派生出任何终结符，所以FIRST(ε)只包含ε。但是ε从不在输入流中出现。

对于ε产生式，语法分析器必须将下一个字与可能出现于边缘中ε的直接右部（或等价地到达*Expr'*的直接右部）的符号集合比较。在这种情况下，这个集合将是某个产生式的右部中跟在*Expr'*后面的符号派生出的符号的集合。如果这个*Expr'*的右部的符号可以派生出ε，那么分析器还必须考虑它右部的符号，以此类推，直到语法分析器发现不能派生出ε的符号。设FOLLOW(*A*)为合法句子中可能出现于非终结符*A*的直接后面的符号组成的集合。在这一文法中，FOLLOW(*Expr'*)是{eof,)}。

无回溯的条件依赖于FIRST和FOLLOW集合。图3-5给出计算这些集合的算法。FIRST集合必须在FOLLOW集合之前计算。因为FOLLOW集合的计算依赖于FIRST集合的计算。二者的计算可以公式化为不动点计算。

```
for each α ∈ (T ∪ ε)
    FIRST(α) ← α
for each A ∈ NT
    FIRST(A) ← ∅
while (FIRST sets are still changing)
    for each p ∈ P, where p has the form A→β
        if β is β1β2...βk, where βi ∈ T ∪ NT, then
            FIRST(A) ← FIRST(A) ∪ (FIRST(β1) − {ε})
            i ← 1
            while (ε ∈ FIRST(βi) and i < k-1)
                FIRST(A) ← FIRST(A) ∪ (FIRST(βi+1) − {ε})
                i ← i + 1
        if i = k and ε ∈ FIRST(βk)
            then FIRST(A) ← FIRST(A) ∪ {ε}

for each A ∈ NT
    FOLLOW(A) ← ∅
FOLLOW(S) ← {eof}
while (FOLLOW sets are still changing )
    for each p ∈ P of the form A → β1β2 ··· βk
        FOLLOW(βk) ← FOLLOW(βk) ∪ FOLLOW(A)
        TRAILER ← FOLLOW(A)
        for i ← k down to 2
            if ε ∈ FIRST (βi) then
                FOLLOW(βi−1) ← FOLLOW(βi−1) ∪
                                {FIRST(βi) − ε} ∪ TRAILER
            else
                FOLLOW(βi−1) ← FOLLOW(βi−1) ∪ FIRST(βi)
                TRAILER ← ∅
```

图3-5 计算FIRST和FOLLOW集合

我们可以使用FIRST和FOLLOW集合精确地公式化无回溯文法必须满足的条件。如下定义集合 98
FIRST$^+(\alpha)$：若FIRST(α) 不包含ε，则为FIRST$^+(\alpha)$=FIRST(α)，否则为FIRST$^+(\alpha)$=FIRST$(\alpha)\cup$FOLLOW(α)。于是，一个文法有无回溯性质的条件是，对于任意带有多个右部$A\to\beta_1|\beta_2|\cdots|\beta_n$的非终结符$A$，它满足下式：

$$\text{FIRST}^+(\beta_i)\cap\text{FIRST}^+(\beta_j)=\emptyset,\ \forall 1\le i<j\le n$$

因为FIRST集合是定义在文法符号上而不是定义文法符号的串上的，所以我们把FIRST(β_i) 解释成β_i的第 99
一个符号的FIRST集合。上面的规则刻画了无回溯性质。

例如，考虑扩展右递归表达式文法，使其包含标量变量引用、数组元素引用和函数调用的语法。为了扩展这一文法，我们用描述标量变量引用、数组元素引用和函数调用语法的产生式集合取代产生式*Factor*→ident。

```
11  Factor     →  ident
12             |  ident [ ExprList ]
13             |  ident ( ExprList )
14  ExprList   →  Expr , ExprList
15             |  Expr
```

这些产生式使用方括号来标记数组引用，用圆括号来标记函数调用。

上述文法片段不满足无回溯条件，因为产生式11、12和13都开始于ident。它们的右部的FIRST集合是相同的。当分析器尝试展开分析树较低边缘的*Factor*时，通过只向前看一个字分析器无法区分11、12和13。（当然，向前看两个字可以使它能够预测正确地展开。）

正由于很多文法不满足无回溯条件，我们可以重写文法，使其即不改变语言又无文法回溯。下面的重写版本就是如此：

```
11  Factor     →  ident Arguments
12  Arguments  →  [ ExprList ]
13             |  ( ExprList )
14             |  ε
15  ExprList   →  Expr , ExprList
16             |  Expr
```

这一版本的文法把派生分成两步，一步识别三个原始右部的公共前缀ident，另外一步包含三个不同选项。为了实现这一点，我们引入了一个新的非终结符*Arguments*，并把[、(和ε叠加到*Arguments*的右部。我们称这一转换为ident的*左因子分解*（left-factoring）。

100

我们可以把这一转换运用到任意文法上。图3-6给出完成这一工作的一个简单的算法。这一算法系统地识别有两个或多个产生式公共前缀的非终结符并分解展开这个非终结符的产生式。因此，如果这一算法找到有如下产生式的非终结符A，

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_j$$

其中α是公共前缀，γ_j的表示不开始于α的右部，那么算法使用如下产生式组取代这些产生式

$$\begin{aligned} A &\rightarrow \alpha B \mid \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_j \\ B &\rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{aligned}$$

其中B是一个新的非终结符。repeat循环的下一次迭代还将考虑B的左因子分解。当这一算法再也找不到公共前缀时就终止。

```
for each A ∈ NT
    if a common prefix exists for two or more
        right-hand sides for A then
      find the longest common prefix α
      left-factor α out of the right-hand sides for A
repeat until no common prefixes remain
```

图 3-6

左因子分解可以把那些需要回溯的文法转换成无回溯文法。然而，并非所有上下文无关语言都能够

用无回溯文法表示。使用左递归消除和左因子分解，我们也许能够把一个文法转换成可以预测分析的形式。然而一般地，上下文无关语言是否有无回溯文法是不可解的。

3.3.5 自顶向下递归下降分析器

给定一个预测文法G，我们可以为G手工构造递归下降分析器。一个递归下降分析器是一组交互递归的过程，每个过程对应文法中的一个非终结符。 101

预测分析器和DFA的比较

预测分析是DFA类型的推断到语法分析器的一种自然扩展。DFA基于下一个输入字符进行转换。预测分析器要求展开由输入流中的下一个字惟一确定。因此，对于文法中的每个非终结符，必须存在从任意可接受输入字符串中的第一个字到导致这个字符串的一个派生的特定产生式的惟一映射。DFA与可预测分析文法即LL(1) 文法间能力的实际差异来自于这样的事实：一个预测可以导致一个带有很多符号的右部，而在正则文法中，它只预测一个符号。这使得预测文法包含诸如$p\rightarrow(p)$这样的产生式，这一产生式超出了正则表达式的描述范畴。（回想一下正则表达式能够识别$(^*\Sigma^*)^+$，但是这并不表示开括号和闭括号的数目必须匹配）。

当然，手工构造的递归下降分析器可以使用任意技巧来惟一确定产生式的选择。例如，如果特定的左部不能通过一个向前看符号来预测，那么语法分析器可以使用两个向前看符号。只要明智地做，这不会引发问题。

非终结符A的过程识别输入流中A的实例。为了完成这一识别，A的过程调用识别A的右部各非终结符的过程。

考虑G中的一组产生式$A\rightarrow\beta_1|\beta_2|\varepsilon$。因为$G$是可预测的，只使用一个输入字和FIRST$^+$集合，分析器就能够选择适当的右部（$\beta_1$、$\beta_2$或$\varepsilon$中的一个）。因此，识别$A$的过程应该包含检查每一个可选择的右部的代码。这可以是如下形式： 102

```
/* find an A */
if (word ∈ FIRST+(β1)) then
  look for a β1
else if (word ∈ FIRST+(β2)) then
  look for a β2
else if (word ∈ FOLLOW(A)) then
  return true
else
  report an error
  return false
```

分析器必须用不同于处理其他右部的方式处理ε产生式。

每个右部β_i的代码必须依次识别β_i中的每个文法符号。如果这一符号是终结符，那么分析器就可直接用输入字与这一终结符比较。如果二者匹配则继续分析下去；如果二者不匹配则表明有一个错误。如果这一符号是非终止的，那么分析器调用识别这一非终结符的过程。它对应于成功或失败分别返回真或假。另外，如果成功则继续分析下去。如果识别过程返回假，那么当前的过程也应该返回假。发现不匹配的过程可以立即报告一个错误。

对于右部$\beta_1=cDE$，其中$c\in T$，且D，$E\in NT$，识别过程需要去识别一个c，然后是一个D，最后是一个E。在这一代码片段中，我们用“*look for a* β_1”抽象这些动作。在处理β_1的过程*Parse_A*的代码如下所示：

```
if (word ∈ FIRST+(β1)) then
  if (word ≠ c) then
    report an error finding c in cDE
    return false
  else /* word is c */
    word ← NextWord()
    if (Parse_D() = true)
      then return Parse_E()
      else return false
```

*Parse_A*对*A*的每个可选择右部包含一个类似的代码片段。

构造完整的递归下降分析器的策略是显然的。对于每个非终结符，我们构造一个识别这个非终结符的过程。每个过程依赖于识别其他非终结符的其他过程，而且直接测试这一非终结符右部所有的终结符。图3-7给出3.3.3节中得到的预测文法的一个自顶向下递归下降分析器。类似右部的代码已经结合在一起。
103 过程*NextWord()*是扫描器的递增接口。

```
Main()
  /* Goal → Expr */
  word ← NextWord();
  if (Expr() and word = eof)
    then proceed to the next step
    else return false

Expr()
  /* Expr → Term Expr' */
  if (Term() = false)
    then return false
    else return EPrime()

EPrime()
  /* Expr' → + Term Expr'*/
  /* Expr' → - Term Expr'*/
  if (word = + or word = -) then
    word ← NextWord()
    if (Term() = false)
      then return false
      else return EPrime()
  /* Expr' → ε */
  return true

Term()
  /* Term → Factor Term' */
  if (Factor() = false)
    then return false
    else return TPrime()

TPrime()
  /* Term' → × Factor Term' */
  /* Term' → ÷ Factor Term' */
  if (word = × or word = ÷ ) then
    word ← NextWord()
    if (Factor() = false)
      then return false
      else return TPrime()
  /* Term' → ε */
  return true

Factor()
  /* Factor → ( Expr ) */
  if (word = ( ) then
    word ← NextWord()
    if (Expr() = false)
      then return false
      else if (word ≠ ) ) then
        report syntax error
        return false
  /* Factor → num */
  /* Factor → ident */
  else if (word ≠ num and
        word ≠ ident) then
    report syntax error
    return false
  word ← NextWord()
  return true
```

图3-7 表达式的递归下降分析器

1. 自动化递归下降分析器

自顶向下递归下降分析通常被认为是手工构造分析器的技术。当然，我们能够构建自动生成适当文法的自顶向下递归下降分析器的分析器生成器。分析器生成器会为文法构造FIRST和FOLLOW集合，检查每一个非终结符以确保它的可选右部有互不相交的初始终结符集合，并对文法中的每一个非终结符生成适当的分析过程。这一结果的语法分析器具有自顶向下递归下降分析器的优点，诸如高速、代码空间

局部化和良好的错误检测等等。它还应该具有文法生成系统的优点，诸如简明、高级描述以及降低实现的工作量等等。

预测分析的代价与分析树的大小及在每次展开时选择正确右部所需要的时间量成正比。为了减小后者的代价，编译器设计者可以把嵌套的*if*语句修改成关于当前字的选择语句。如果能用适当的方法翻译选择语句的话，那么这可以产生更有效的代码。

另外，编译器设计者也可以构造刻画这些动作的表格。例如，对于右递归表达式文法，分析器有三种*Factor*展开：(*Expr*)，ident和num。使用表格形式，这可以如下表示：

	+	−	×	÷	(	)	ident	num	*eof*
Factor	−	−	−	−	9	−	11	10	−

其中，第一行是终结符号列表，而所有的条目或者是规则编号或者是表示一个语法错误的破折号（−）。这张表格作为当前输入符号函数预测*Factor*的可能展开。

如果我们扩张这张表格使其包含所有非终结符，那么图3-8中的算法可以使用这张表格来完成自顶向下的表驱动分析。结果语法分析器是一个表驱动LL(1) 分析器。名字LL(1) 得自于这样的语法分析器从左到右（L）扫描它们的输入，并构造一个最左派生（L），并使用一个向前看符号（1）的事实。可以用LL(1) 分析的文法通常被称为LL(1) 文法。根据定义，LL(1) 文法是无回溯的。 104

```
word ← NextChar()
push eof onto stack
push Start Symbol onto stack
TOS ← top of stack
loop forever
  if TOS = eof and word = eof
    then report success and exit the loop
  else if TOS is a terminal or eof then
    if TOS matches word then
      pop the stack
      word ← NextWord()
    else
      report an error looking for TOS
  else /* TOS is a nonterminal */
    if Table[TOS,word] is A → B1B2 ··· Bk then
      pop the stack
      for i ← k to 1 by -1
        push Bi onto the stack
    else
      report an error expanding TOS
  TOS ← top of stack
```

图3-8 LL(1) 框架分析器

为了构建LL(1) 分析器的表格，编译器设计者必须计算FIRST和FOLLOW集合，然后填写表格的每一个条目。对于*Table*[*X*，*y*]，其中*X*是一个非终结符而*y*是一个终结符，如果*y*∈FIRST$^+$(*β*)，那么条目应该是产生式*X*→*β*的规则编号。如果*X*→*ε*也在文法中，那么FOLLOW(*X*) 中的每个终结符的条目应该是这个*ε*产生式的规则编号。没有被这些规则定义的所有条目都是错误条目。如果有多次定义的条目，那么构造失败。

所有构造成功的文法都是LL(1) 文法。结果表格可以直接用于得到基于图3-8的框架分析器的分析器。这一文法也适合于自顶向下递归下降分析器。

2. 总结

预测分析是重要的，因为大多数程序设计语言结构可以用无回溯文法表示。在实践中，对非终结符的各右部有不同FIRST集合的限制不对LL(1) 文法的有效性产生太多的制约。我们给这些文法构建的语法分析器都很简洁且有效。它们可以对错误的输入产生高质量的诊断信息。对于小型语言以及良好的错误信息至关重要的情况，自顶向下递归下降分析器是一个可行的选择。

105 ~ 106

3.4 自底向上分析

自底向上分析器从分析树的叶子出发并向它的根部来构建分析树。当它遇到输入流中的每个字时，它构造一个叶结点。这些叶结点形成分析树的基础。为了构造派生，自底向上分析器以由文法和输入流共同指定的结构，在叶子的上方层层加入非终结符。这一部分构建起来的分析树的上部边缘称为它的上边界（upper frontier）。这一过程向着树的根部向上延伸上边界。

在延伸的每一步，语法分析器寻找与文法中的某个产生式的右部匹配的上边界的一个截取。当它找到一个匹配时，语法分析器就构建一个表示该产生式左部非终结符的结点，并加入从该结点到表示右部符号的结点的边。因为产生式的左部只有一个非终结符，这些向上的扩展用单一符号取代上边界上的一个或多个符号。语法分析器重复这一过程，直到出现下面的两个条件之一：

1）上边界减少到表示这一文法开始符号的单一结点。如果语法分析器已匹配输入中的所有字，那么输入是语言中的合法句子。如果某些字剩余下来，那么输入不是合法句子。相反，它是一个合法句子后面跟着额外的字，语法分析器应该向用户报告这一错误。

2）找不到匹配。因为语法分析器还不能为输入流构建一个派生，输入不是合法句子。语法分析器应该向用户报告这一失败。分析树的上边界包含可以用于构建诊断消息的信息。

成功的语法分析贯串派生的每一步。失败的语法分析当它没有找到下一步时停止，在这一点它可以使用从树中收集起来的上下文来生成有意义的错误信息。在很多情况下，语法分析可以从错误中恢复并继续分析，这样它可以在一次分析过程中发现尽可能多的语法错误（见3.6.1节）。

自底向上分析器中的派生开始于目标符号并逐渐得到一个句子。因为语法分析器自底向上地建立分析树，所以它以相反的顺序发现派生步骤。如果派生是由产生如下句型的一系列步骤组成的

107

$$S_0 = \gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \cdots \rightarrow \gamma_{n-1} \rightarrow \gamma_n = \text{句子},$$

那么自底向上分析器将在发现$\gamma_{n-2} \rightarrow \gamma_{n-1}$之前发现$\gamma_{n-1} \rightarrow \gamma_n$。树的自底向上构造法导致这一顺序。语法分析器必须将这些由$\gamma_{n-1} \rightarrow \gamma_n$指明的结点加入到上边界上，而且必须是在它发现任何包含这些结点的匹配之前。因此，它只能按逆向派生的顺序发现结点。

因为扫描器是按从左到右的顺序寻找输入流中的字，语法分析器就应该以左到右的顺序检查叶子。这就给出从右到左产生终结符的派生顺序，使得它的逆顺序与扫描器的行为吻合。这相当自然地导致按逆顺序构造最右派生的自底向上分析器。在构建的每一步，语法分析器将在部分构建起来的分析树的上边界进行操作；当前上边界是派生中相应的句型的前缀。因为每个句型出现在最右派生中，所以还没有检查到的后缀由终结符组成。

当文法非歧义时，自底向上分析比较容易。对于非歧义文法，最右派生是惟一的。对于一大类非歧义文法，可以从γ_i和有限的上下文来决定γ_{i-1}。这导致一类高效的自底向上分析器。

本节考虑称为LR(1) 分析器的自底向上分析器的特定类。这些分析器从左到右扫描输入，这是扫描器返

回归类字的顺序。这些语法分析器按相反顺序构建最右派生。在分析过程的每一步，LR(1) 分析器根据到此为止的分析历史和最多一个向前看字符做出决定。LR(1) 的名字得自于以下三个性质：从左到右扫描（L），逆向最右派生（R）和向前看一个符号（1）。非形式地，如果一个语言可以通过单一的从左到右扫描、构建逆向最右派生，并且只使用一个向前看符号来分析的话，那么我们称这一语言具有LR(1) 性质。[⊖]

3.4.1 移入归约分析

构造有效的自顶向下分析器的关键是发现在每步使用的产生式的正确右部。在自底向上的语法分析中，至关重要的步骤是开发有效的寻找沿着树的当前上边界的匹配的机制。形式地，语法分析器必须寻 108
找上边界的某个子串β，其中

1）β是某个产生式$A\rightarrow\beta$的右部，而且

2）$A\rightarrow\beta$是输入流的最右派生中的一步。

出于效率的缘故，我们希望在最多看到β右侧的下一个字时，语法分析器完成这一工作。

我们可以把每个可能的匹配表示成序对$\langle A\rightarrow\beta, k\rangle$，其中$A\rightarrow\beta$是$G$中的产生式，而$k$是树的当前上边界上$\beta$的右部的位置。如果使用$A$取代$\beta$的这一出现是输入字符串的逆向最右派生中的下一步，那么$\langle A\rightarrow\beta, k\rangle$称为自底向上分析器的一个*句柄*（handle）。一个句柄精确地表示出在构建逆向最右派生的下一步。

自底向上分析器是通过反复锁定当前部分分析树上边界上的句柄，且执行这些句柄所描述的归约来工作的。当上边界不包含句柄时，分析器调用扫描器得到下一个字，构建相应的叶结点，并使其成为部分构建起来的树中的最右叶子。这样，向前扩展一个叶结点。

为了看清这一工作过程，考虑使用图3-1的经典表达式文法分析字符串x-2 × y。分析器的每一步状态被概括在图3-9中。图3-10给出在这一过程中每一步相应的部分分析树；我们使用点线框把每个图中沿其顶端的上边界圈起来。在每一步，分析器或者寻找上边界上的句柄，或者把下一个字加到上边界上。

	下一个字（Next Word）	边界（Frontier）	句柄（Handle）	动作（Action）
1	ident		— none —	*extend*
2	-	ident	⟨*Factor*→ident,1⟩	*reduce*
3	-	*Factor*	⟨*Term*→*Factor*,1⟩	*reduce*
4	-	*Term*	⟨*Expr*→*Term*,1⟩	*reduce*
5	-	*Expr*	— none —	*extend*
6	num	*Expr* -	— none —	*extend*
7	×	*Expr* - num	⟨*Factor*→num,3⟩	*reduce*
8	×	*Expr* - *Factor*	⟨*Term*→*Factor*,3⟩	*reduce*
9	×	*Expr* - *Term*	— none —	*extend*
10	ident	*Expr* - *Term* ×	— none —	*extend*
11	*eof*	*Expr* - *Term* × ident	⟨*Factor*→ident,5⟩	*reduce*
12	*eof*	*Expr* - *Term* × *Factor*	⟨*Term*→*Term* × *Factor*,5⟩	*reduce*
13	*eof*	*Expr* - *Term*	⟨*Expr*→*Expr* - *Term*,3⟩	*reduce*
14	*eof*	*Expr*	⟨*Goal*→*Expr*,1⟩	*reduce*
15	*eof*	*Goal*	— none —	*accept*

图3-9 自底向上分析器分析x-2 × y时的状态

⊖ LR分析理论定义一系列分析技术，即LR(k) 分析器，其中k是任意非负整数。这里，k表示语法分析器所需的向前看符号的数量。对于任意的$k>1$，LR(1) 分析器与LR(k) 分析器接受相同的语言集合；然而，语言的LR(1) 文法可能比需要更多向前看符号的文法更加复杂。

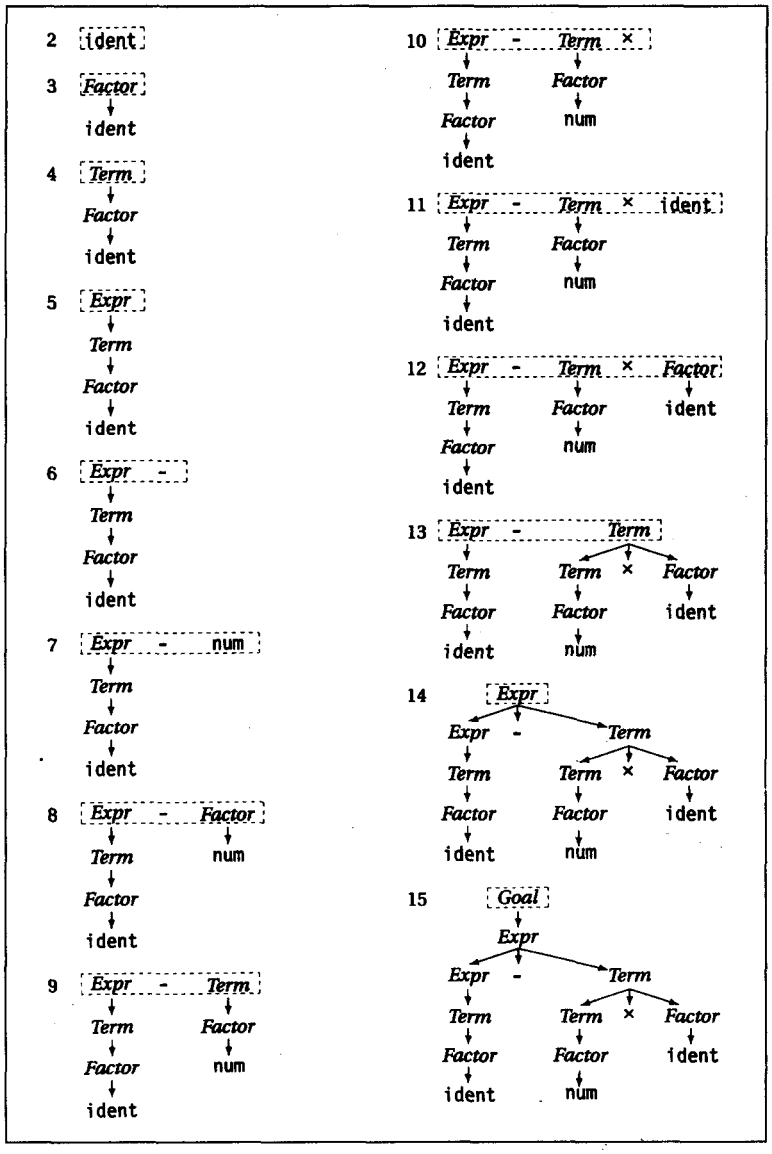


图3-10 x-2×y在自底向上分析中的部分分析树

正如例子所示的那样，语法分析器只需检查已部分构建起来的分析树的上边界。利用这一事实，我们可以构建称为*移入归约*（shift-reduce）分析器的自底向上分析器的一种非常清晰的形式。这样的语法分析器使用栈来保存上边界；从而以两种方法简化算法。首先，栈简化保存上边界的空间组织问题。为了扩展边界，语法分析器只需把当前输入符号压入栈的顶端。其次，这确保所有句柄的右端都位于栈顶。这样，我们就不必再明确地表示句柄的位置，简化了表示并使得句柄集合有限。

图3-11给出一个移入归约分析器的例子。一开始，它将一个特定的符号*invalid*移入到栈上，并从扫描器中获取第一个输入符号。接着，它遵守一个简单的规律：把输入符号移入到栈上，直到发现一个句柄，句柄一旦被发现，语法分析器就归约这个句柄。当栈顶包含*Goal*且分析器消耗了所有的输入时，语法分析器终止这一过程。

109

```
push invalid
word ← NextWord()
repeat until (word = eof & the stack contains
      exactly Goal on top of invalid)
   if a handle for A→β is on top of the stack then
     /* reduce by A→β */
     pop | β | symbols off the stack
     push A onto the stack
   else if (word ≠ eof) then
     /* shift word onto the stack */
     push word
     word ← NextWord()
   else /* no handle, no input */
     report syntax error & halt
```

图3-11 移入归约分析算法

当句柄发现机制失败时，这一算法发现语法错误。因为图3-11只是用一种抽象的方法描述句柄查询，"*if a handle for A→β is on top of the stack*，"错误检查的细节不很清晰。事实上，图3-11表明，只有当所有输入已被移入到栈中时，错误才可能发生。当我们揭示句柄发现机制时，我们将会看到语法分析器在这一过程的更早阶段发现语法错语。例如，输入字符串x+ ÷ y不在经典表达式文法所描述的语言中。句柄发现器应该在看到 ÷ 跟在一个+后面时马上就识别出这一错误。

使用图3-11中的算法，我们可以重新解释图3-9来展示我们的移入归约分析器作用于输入流x-2 × y的动作。标有"Next Word"的列给出这一算法中的变量*word*的内容。标有"Frontier"的列描述每一步上栈的内容；栈顶向右。最后，动作*extend*标明一个移入；*reduce*表示一个归约。 110 ≀ 111

对于长度为s的输入流，移入归约分析器执行s次移入。它对于派生的每步执行一个归约，共有r步。*repeat until*循环的每次迭代寻找一个句柄，所以它必须执行$s + r$次句柄发现操作。这等价于分析树上结点的数目；每次移入和每次归约都在分析树上创建一个新结点。

对于固定的文法，r必须是$O(s)$，所以句柄发现操作的数目与输入字符串的长度成正比。如果我们能够维持句柄发现的代价为一个小常量，那么语法分析器的操作时间与$s + r$成正比。当然，这样的句柄发现代价的约束使我们无法在每次句柄发现操作中使用遍历整个栈的任何技术。而这对高效的句柄发现非常重要。

3.4.2 发现句柄

句柄发现机制是高效自底向上分析的关键。当它处理输入字符串时，语法分析器必须发现并跟踪所有可能的句柄。例如，每个合法输入最终把整个上边界归约成文法的目标符号。在经典表达式文法中，*Goal*→*Expr*是归约到*Goal*的惟一产生式。它必须是任意成功的语言分析中的最后归约。如果整个上边界被归约到*Goal*，那么句柄的位置必须是1。因此，〈*Goal*→*Expr*，1〉是每次语法分析开始处的可能句柄。 112

当语法分析器建立一个派生时，它发现其他句柄。在每一步，可能的句柄集合应该表示导致归约的不同后缀，前提是能够看到这样的句柄。给定已构建的派生部分，每个可能的句柄表示某个文法符号的串，这一文法符号串是文法中某个产生式的完整右部。图3-9给出移入归约分析器在处理x-2 × y时发现的9个完整句柄。

在上述分析中的第三步和第八步的句柄都是由*Term*→*Factor*描述的归约。句柄有不同位置域，表明上边界中符号*Factor*出现的地方。然而，在移入归约分析器中，表中所给出的上边界的位置总是处于栈中，而且最新标识符在栈顶。在第三步和第八步，句柄的位置域描述栈顶的符号。事实上，对于语法分

析器识别的每个句柄，位置域都描述栈的顶端。如果我们将这一位置域看成栈顶的相关位置，那么我们就能够显著减少不同句柄的数量：直至文法中每一个产生式有一个句柄。

继续推进这一表记法，我们可以表示移入归约分析器应该跟踪的可能句柄。如果我们使用占位符•来表示栈顶，那么图3-9中的九个句柄变成：

⟨*Factor*→ident •⟩	⟨*Term*→*Factor* •⟩	⟨*Expr*→*Term* •⟩
⟨*Factor*→num •⟩	⟨*Term*→*Factor* •⟩	⟨*Factor*→ident •⟩
⟨*Term*→*Term* × *Factor* •⟩	⟨*Expr*→*Expr* - *Term* •⟩	⟨*Goal*→*Expr* •⟩

上面的表记法表明第二个和第五个句柄是相同的，第一和第六个也是相同的。这一表记法还生成将来发现句柄的可能性的一种方法。

考虑步骤六处的分析器状态，此时，输入符号为num和上边界为*Expr*-。根据表格，我们知道下一个归约将是⟨*Factor*→num•⟩，后面跟着下一个移入和归约以标识符 × 和ident。然而，当前的上边界是什么呢？分析器已识别出*Expr*-。分析器在它能够归约上边界的这一部分之前，它需要识别一个*Term*。使用这一栈相关的表记法，我们可以把语法分析器的状态表示为*Expr*→*Expr*-•*Term*。语法分析器已经识别出一个*Expr*和一个-，并且-在栈顶。如果语法分析器达到一个将一个*Term*移入到*Expr*-的上端的状态，那么它将完成句柄*Expr*→*Expr*-*Term*•。

使用同时表示句柄和可能句柄的一个具体的表记法，我们可以问这样的问题：语法分析器必须识别多少个可能句柄？每个产生式的右部在它的开始处、在它的结束处以及在任意两个相连的符号之间都可
113 以有一个占位符。如果右部有k个符号，它就有k + 1个占位符位置。文法的可能句柄的数目是所有产生式的右部的长度和。完整句柄的数目是产生式的数目。这两个事实导致LR(1) 分析器背后的一个重要结果。

一个文法生成语法分析器必须识别的句柄（和可能句柄）的有穷集合。

根据第2章，我们有识别字的有穷集合的合适工具DFA。LR(1) 分析器使用句柄识别DFA来有效地发现处于分析栈栈顶的句柄。表格构造算法构建这样一个DFA模型，并用表格序对来实现它。

仔细检查图3-9中的分析揭示出这一推理中的一个缺点。考虑第九步处的语法分析器动作。上边界是*Expr*-*Term*，表明句柄是⟨*Expr*→*Expr*-*Term*•⟩。然而，语法分析器决定通过把 × 移入到栈中来扩展这一边界，而不是把这一边界归约到*Expr*。显然，对语法分析器来说这是正确的动作。没有包含后面跟有 × 的*Expr*的可能句柄。

为了决定正确的动作，语法分析器能够识别不同的右上下文，也就是输入流中在后面出现的符号所需要的不同动作。在第九步，可能的句柄集合是：

⟨*Expr*→*Expr* - *Term* •⟩ ⟨*Term*→*Term* • × *Factor*⟩ ⟨*Term*→*Term* • ÷ *Factor*⟩

下一个输入符号 × 显然与第二个选择匹配，而且剔除第三个选择。语法分析器需要一个在第一个和第二个选择之间做出选择的标准。这需要比上边界所拥有的上下文更多的上下文。

为了在将*Expr*-*Term*归约到*Expr*还是移入 × 以识别*Term* × *Factor*之间做出选择，语法分析器必须知道在有效分析中哪个符号可以出现在*Expr*和*Term*的右边。考虑我们的文法，在产生式1和2中，+和-直接跟在*Expr*的后面。根据产生式4和5，× 和 ÷ 可以跟在*Term*的后面。因此，在步骤九，语法分析器可以通过查看下一个符号来选择正确的动作：对于+和-，语法分析器应该进行归约；对于 × 和 ÷，它应该移入。因为下一个符号是 ×，所以语法分析器应该移入。

LR(1) 分析器可以精确识别那些一个向前看符号就足以决定是移入还是归约的语言。LR(1) 构造算

法构建一个句柄识别DFA；分析算法使用这一DFA来识别分析栈中的句柄和可能句柄。它使用移入归约分析框架来引导DFA的运用。这一框架可以递归地调用DFA，为了完成这一调用，在栈中储存表示分析 114
树上边界的文法符号的同时，还储存有关DFA内部状态的信息。

3.4.3 LR(1) 分析器

LR(1) 分析器，包括它的称为SLR(1) 和LALR(1) 的限定形式，是得到最广泛应用的语法分析器系列。表驱动LR(1) 分析器具有很高的效率，存在很多自动构建表格的工具。这些工具接受的文法使我们能够自然地表示大多数程序设计语言的结构。本节简述LR(1) 分析器的工作方式，并展示如何构造一种称为规范LR(1) 分析器的LR(1) 分析器的分析表格。

图3-12给出典型的LR(1) 分析器生成器系统的结构。编译器设计者创建描述源程序语言的文法。分析器生成器读取这一文法并生成一对驱动LR(1) 分析器的表格。这些表格描绘分析所需要的所有文法信息；在某种意义上，分析器生成器把识别句柄、决定移入时机、决定归约时机以及在归约中使用哪个规则所需的所有信息预编译到这些表格中。在大多数这样的系统中，编译器设计者也可以为将执行归约的产生式提供代码片段。这些特定“动作”提供一种在编译时执行语法制导计算的机制。第4章给出这些 115
特定动作的某些运用。

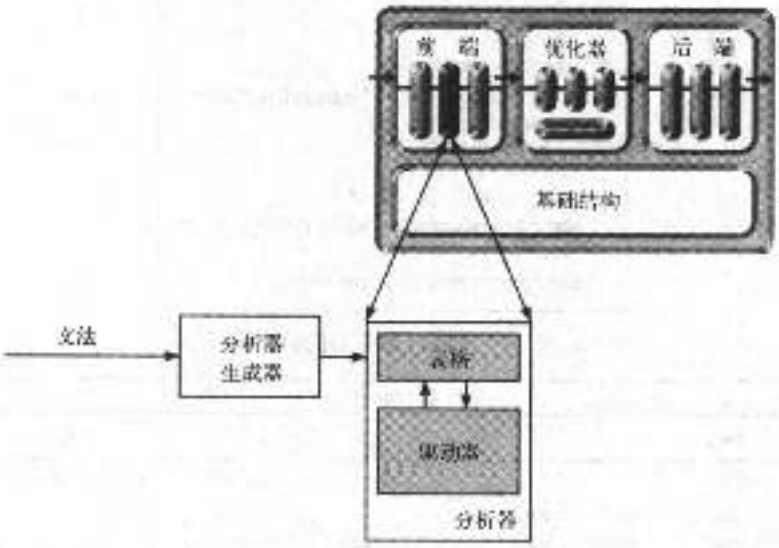


图3-12 LR(1) 分析器生成器系统的结构

LR(1) 表格的构造法是理论应用于实践的一个出色的实例。这一构造法系统地构建句柄识别DFA的模型，然后把这一模型翻译成表驱动框架分析器的一对表格。然而，这一构造法是一个复杂的任务，需要格外注意细节。这正是应该自动化的任务：分析器生成器在跟踪这些繁杂计算上要比人类更优秀。尽管如此，技术娴熟的编译器设计者应该理解这一表格构造算法，因为他们可以借助语法分析器的工作方式，分析器生成器所能遇到的错误，这些错误是如何引发的，以及如何修正它们。

如图3-12所示，LR(1) 分析器由一个表驱动器和驱动分析器的一对表格组成。图3-13给出表驱动器。称为*Action*和*Goto*的一对表格描绘驱动器所需的文法信息。图3-14和图3-15给出经典表达式文法的相应表格。为了简化表格构造算法，我们为文法添加了一个目标产生式并对其余产生式适当编号。结果文法 116
如下：

```
1   Goal   → Expr
2   Expr   → Expr + Term
3          | Expr - Term
4          | Term
5   Term   → Term × Factor
6          | Term ÷ Factor
7          | Factor
8   Factor → ( Expr )
9          | num
10         | ident
```

```
push invalid
push start state, s0
word ← NextWord()
while (true)
  s ← top of stack
  if Action[s,word] = "shift si" then
    push word
    push si
    word ← NextWord()
  else if Action[s,word] = "reduce A → β" then
    pop 2 × | β | symbols
    s ← top of stack
    push A
    push Goto[s, A]
  else if Action[s,word] = "accept" then
    return
  else report syntax error and halt
```

图3-13 LR(1)分析器的驱动器

Action表

状态	eof	+	-	*	÷	(	)	num	ident
0						s4		s5	s6
1	acc	s7	s8						
2	r4	r4	r4	s9	s10				
3	r7	r7	r7	r7	r7				
4						s14		s15	s16
5	r9	r9	r9	r9	r9				
6	r10	r10	r10	r10	r10				
7						s4		s5	s6
8						s4		s5	s6
9						s4		s5	s6
10						s4		s5	s6
11		s21	s22				s23		
12		r4	r4	s24	s25		r4		
13		r7	r7	r7	r7		r7		
14						s14		s15	s16

图3-14 经典表达式文法的Action表

Action表									
状态	eof	+	−	×	÷	(	)	num	ident
15		r9	r9	r9	r9		r9		
16		r10	r10	r10	r10		r10		
17	r2	r2	r2	s9	s10				
18	r3	r3	r3	s9	s10				
19	r5	r5	r5	r5	r5				
20	r6	r6	r6	r6	r6				
21						s14		s15	s16
22						s14		s15	s16
23	r8	r8	r8	r8	r8				
24						s14		s15	s16
25						s14		s15	s16
26		s21	s22				s31		
27		r2	r2	s24	s25		r2		
28		r3	r3	s24	s25		r3		
29		r5	r5	r5	r5		r5		
30		r6	r6	r6	r6		r6		
31		r8	r8	r8	r8		r8		

图3-14 （续）

Goto表				Goto表			
状态	Expr	Term	Factor	状态	Expr	Term	Factor
0	1	2	3	16			
1				17			
2				18			
3				19			
4	11	12	13	20			
5				21		27	13
6				22		28	13
7		17	3	23			
8		18	3	24			29
9			19	25			30
10			20	26			
11				27			
12				28			
13				29			
14	26	12	13	30			
15				31			

图3-15 经典表达式文法的*Goto*表

框架分析器类似于图3-11所给的移入归约分析器。在每一步，框架分析器把两个对象压入栈：来自于上边界的一个文法符号和来自于句柄识别器的一个状态。它有下面4种动作：

（1）移入（shift）

通过将向前看符号移入栈中来扩展上边界，同时将句柄识别器的新状态压入栈中。这可能导致识别器的递归调用。

（2）归约（reduce）

通过使用当前句柄的左部取代这一句柄缩短上边界。通过对每个句柄右部符号弹出两个栈中元素，丢弃所有用于识别这一句柄的中间状态。下一步，它使用句柄下面的状态和句柄的左部寻找新的识别器状态，并把句柄的左部和新状态压入栈中。

（3）接受（accept）

报告成功。只有当分析器将上边界归约到目标符号且向前看符号是*eof*时，才能达到这一状态。

（4）错误（error）

报告语法错误。在*Action*表格包含移入（*shift*）、归约（*reduce*）和接受（*accept*）之外的条目时达到这一状态。在图中，这些条目是空的。

在*Action*表中，字母*s*表示移入，*r*表示归约。因此，条目*s*3表示“移入并进入状态s_3，”的动作，而
117 ~ 118
*r*5表示“用产生式5进行归约。”*Goto*表描绘在归约动作后必须采取的转换。这一表格的稀疏性反应只有较少的状态表示归约的事实。

为了理解这一分析器，再次考虑我们的例子。图3-16给出LR(1) 分析器分析表达式x-2 × y时的状态序列。分析器以移入*invalid*并将以0表示的状态压入栈中开始。下一步，分析器把ident移入到栈中，接着将其归约到*Factor*、*Term*和*Expr*。接着，它移入-，再移入num。此时，它将num归约到*Factor*，接着将*Factor*归约到*Term*。下一步，它移入×，然后移入ident。最后，它将ident归约到*Factor*，将*Term* × *Factor*归约到*Term*。然后，将*Expr*-*Term*归约到*Expr*。此时，因为栈顶为*Expr*而下一个字为EOF，它接受这一输入字符串。这与图3-9所描述的序列类似。

	当前符号（Current Symbol）	栈（Stack）	动作（Action）
1	ident	*invalid* 0	*shift*
2	-	*invalid* 0 ident 6	*reduce 10*
3	-	*invalid* 0 *Factor* 3	*reduce 7*
4	-	*invalid* 0 *Term* 2	*reduce 4*
5	-	*invalid* 0 *Expr* 1	*shift*
6	num	*invalid* 0 *Expr* 1 - 8	*shift*
7	×	*invalid* 0 *Expr* 1 - 8 num 5	*reduce 9*
8	×	*invalid* 0 *Expr* 1 - 8 *Factor* 3	*reduce 7*
9	×	*invalid* 0 *Expr* 1 - 8 *Term* 18	*shift*
10	ident	*invalid* 0 *Expr* 1 - 8 *Term* 18 × 9	*shift*
11	*eof*	*invalid* 0 *Expr* 1 - 8 *Term* 18 × 9 ident 6	*reduce 10*
12	*eof*	*invalid* 0 *Expr* 1 - 8 *Term* 18 × 9 *Factor* 19	*reduce 5*
13	*eof*	*invalid* 0 *Expr* 1 - 8 *Term* 18	*reduce 3*
14	*eof*	*invalid* 0 *Expr* 1	*accept*

图3-16 LR(1) 分析器分析x-2 × y时的状态

构建LR(1) 分析器的关键是构建*Action*表和*Goto*表。这些表格描绘句柄识别器DFA的动作，同时还描绘使用限定的右上下文决定是移入、归约还是接受等所需的信息。虽然可以通过手工构建这些表格，
119
但是算法需要处理大量的细节和细心的记录。构建LR(1) 表格是应该交给计算机进行自动化的一个最典型的例子。

3.5 构建LR(1) 表格

为了构建*Action*表和*Goto*表，LR(1) 分析器生成器构建句柄识别DFA的模型，并使用这一模型填充

这些表格。这一模型使用一组将在下一小节描述的LR(1) 项目来表示语法分析器的状态；使用系统技术构造这些项目集合。这一模型称为LR(1) 项目集合的规范集合（canonical collection of sets of LR(1) items），记作$CC=\{CC_0, CC_1, CC_2, \cdots, CC_n\}$，$CC$中的每个集合代表最终分析器的一个状态。

我们使用两个例子来解释这一构造法。第一个例子是我们的*SheepNoise*文法*SN*，并添加了一个目标符号，它足够小以至于可用作详细说明这一过程各个步骤的例子。 120

```
1  Goal       →  SheepNoise
2  SheepNoise →  baa SheepNoise
3             |  baa
```

在构建这些表格中，我们将显式地把符号*eof*作为一个终结符，它出现于输入流的末端，在文法中，它隐式地出现于目标产生式的末端。

第二个例子就是3.5.4节中的例子，这一例子是典型if-then-else歧义文法的抽象版本，这个例子比*SN*更复杂，因为这一文法是歧义的，表格构造法会失败。然而，失败出现于这一过程的后期，所以这一例子展示这一构造法的更加复杂的部分，它强调了导致错误的情况。

3.5.1 LR(1) 项目

LR(1) 表格构造法需要句柄、可能句柄以及与其相关的向前看符号的具体表示。我们称这样的表示为LR(1) 项目。一个LR(1) 项目是一个序对$[A\rightarrow\beta\bullet\gamma, a]$，其中$A\rightarrow\beta\bullet\gamma$代表句柄或可能句柄，占位符•表示栈顶的位置，$a\in T$是源语言的字。各个LR(1) 项目描述自底向上语法分析器的格局，它们表示与语法分析器已检查的左上下文一致的可能句柄。

对于一个产生式$A\rightarrow\beta\ \gamma$和向前看符号$a\in T$，占位符可以生成三种不同的项，每一种都有其自身的含义。对于每一种情况，项目出现在与某个语法分析器状态相关的集合的事实表明，语法分析器已检查过的输入与文法中一个A后面跟一个a的出现是一致的。项目中占位符•的位置提供更多的信息。

- $[A\rightarrow\bullet\beta\ \gamma, a]$表示一个$A$也许是合法的。且在这一点识别一个$\beta$将向发现$A$的方向上迈进一步。我们称这样的项目为可能的（possibility），因为它表示对于已检查输入的一个可能完成的项目。
- $[A\rightarrow\beta\bullet\gamma, a]$表示语法分析器已经通过识别$\beta$，从$A$也许是合法的状态向前迈进了一步。$\beta$与识别$A$一致。下一步将要识别一个$\gamma$。我们称这样的项目为部分完整的（partially complete）。 121
- $[A\rightarrow\beta\ \gamma\bullet, a]$表示语法分析器已经在后面跟一个a的$A$也许是合法的前提下发现了$\beta\ \gamma$。如果向前看符号是a，那么语法分析器就可以将$\beta\ \gamma$归约到$A$，而且这个项目是句柄。这样的项是完整的（complete）。

在LR(1) 项目中，占位符•描绘左上下文，也就是迄今为止的分析历史。向前看符号描绘可能的右上下文。如果语法分析器把占位符移动到产生式的末端并发现下一个符号与项目的向前看符号匹配，那么语法分析器将使用该项目的产生式进行归约。

*SheepNoise*文法生成下列LR(1) 项目：

[*Goal* → • *SheepNoise*, *eof*]	[*SheepNoise* → • baa *SheepNoise*, *eof*]
[*Goal* → *SheepNoise* •, *eof*]	[*SheepNoise* → baa • *SheepNoise*, *eof*]
[*SheepNoise* → • baa, *eof*]	[*SheepNoise* → baa *SheepNoise* •, *eof*]
[*SheepNoise* → baa •, *eof*]	[*SheepNoise* → • baa *SheepNoise*, baa]
[*SheepNoise* → • baa, baa]	[*SheepNoise* → baa • *SheepNoise*, baa]
[*SheepNoise* → baa •, baa]	[*SheepNoise* → baa *SheepNoise* •, baa]

项目[*Goal*→*SheepNoise*•, *eof*]表示这样一个格局：语法分析器已经识别出归约到*Goal*的一个字符串，而且已经消耗尽输入，这由向前看符号*eof*表示。如果栈中*Goal*的下面是空的（也就是说，符号*invalid*在*Goal*的下面），那么分析成功。

3.5.2 构造规范集合

*CC*的构造法以构建语法分析器的初始状态的一个模型开始。这一状态由代表语法分析器的初始状态的项目集合组成，包括在初始状态必须成立的所有项目。为了简化构建这样的初始状态的任务，构造法要求文法有惟一的目标符号，就像*SN*那样。

项目[*Goal*→•*SheepNoise*, *eof*]描述语法分析器的初始状态。它表示这样一个格局：表明后面跟着*eof*的*SheepNoise*应该是一个合法的分析。这个项目是*CC*中记作CC_0的第一个状态的种子。如果对于目标符号文法有几个不同的产生式，那么它们中的每个都生成CC_0的初始种子中的一个项目。

1. 过程*closure*

122 为了计算分析器的初始状态CC_0，构造法从目标符号的每个可选右部的初始项目开始。为了完成这一状态，构造法必须把所有由这些初始项目蕴涵的项目加进来。过程闭包（*closure*）完成这一工作。

```
closure(s)
  while (s is still changing)
    for each item [A→β • Cδ,a] ∈ s
      for each production C→γ ∈ P
        for each b ∈ FIRST(δa)
          s ← s ∪ {[C→•γ,b]}
  return s
```

过程*closure*对集合*s*中的每个项目进行迭代。如果一个项目中的占位符•的后面紧跟某个非终结符*C*，那么对于每个左部为*C*的产生式，*closure*必须把一个或多个相应的项目加进来。占位符•出现于这些产生式右部的开始处。

这样做的原理是显然的。如果[*A*→*β*•*C* *δ*, a]∈*s*，那么左上下文的一个可能完整化是要找归约到*C*，且后面跟着*δ*a的一个字符串。这个完整化将引发一个到*A*的归约，因为它填充这一产生式的右部（*Cδ*）并且跟着一个有效的向前看符号。

对于产生式*C*→*γ*，*closure*过程在*γ*之前插入一个占位符，并把适当的向前看符号，也就是作为初始符号出现于*δ*a中的所有终结符加进来。这包括FIRST(*δ*) 中的每一个终结符。如果*ε*∈FIRST(*δ*)，它还包含a。在这一算法中，表记法FIRST(*δ*a) 用这种方法表示把FIRST集合扩展到字符串。如果*δ*是*ε*，这一集合退化成FIRST(a) = {a}。

对于*SN*，初始项目是{*Goal*→•*SheepNoise*, *eof*}。取它的闭包将两个项目加到该集合：从产生式2而来的{*SheepNoise*→•baa*SheepNoise*, *eof*}和从产生式3而来的{*SheepNoise*→•baa, *eof*}。因为每一个项目的占位符•之后是终结符baa，所以它们不再生成新项目。这三个项目形成第一个集合CC_0。

闭包（*Closure*）计算是另一个不动点计算。在每点，三重嵌套的循环或者把项目加到*s*上，或者保持*s*不变。它不从*s*中移去项目。因为集合LR(1) 的项目是有穷的，这一循环一定终止。三重嵌套循环看似代价很大。然而，仔细审查表明*s*中的每一项目只被处理一次。外部循环需要考虑那些在上次外部循环的迭代中加入的各项目。中间循环只迭代单一非终结符的可选右部的集合，而内部循环在FIRST(*δ*a)
123 上迭代。如果*δ*是*ε*，那么内部循环只检查一个符号a。因此，闭包计算实际上更快。

2. goto过程

这一构造法的第二个关键步骤是从CC_0得到语法分析器的其他状态。为了完成这一工作，对于每个

状态CC_i和每个文法符号x，我们计算当语法分析器在状态CC_i识别出x时产生的状态。过程*goto*完成这一工作。

```
goto(s,x)
  moved ← ∅
  for each item i ∈ s
    if the form of i is [α→β • xδ, a] then
      moved ← moved ∪ {[α→βx • δ, a]}
  return closure(moved)
```

过程*goto*有两个参数，一个是LR(1) 的项目集s，另一个是文法符号x。它在s的项目上进行迭代。如果它发现一个项目的占位符•紧跟在x的前面，那么它通过把•向右移动到x的后面而生成一个新项目。这个新项目表示识别x而得到的状态。过程*goto*把这些项目存放在集合*moved*中，并返回*closure*(*moved*)。

给定*SN*的集合CC_0，我们可以通过计算*goto*(CC_0，baa) 得到在分析一个初始终结符baa之后的语法分析器的状态。移动占位符•到baa的后面产生两个项目：从CC_0的项目2而来的[*SheepNoise*→baa• *SheepNoise*，*eof*]和从CC_0的项目3而来的[*SheepNoise*→baa•，*eof*]。把*closure*运用到这一集合加进来两个新项目：[*SheepNoise*→•baa*SheepNoise*，*eof*]和[*SheepNoise*→•baa，*eof*]。这两个项目都得自于使占位符位于*SheepNoise*之前的第一个项目。

我们的构造法使用*goto*去寻找直接来自于诸如CC_0等状态的状态集合。为了做到这一点，它对于出现于状态CC_0的项目中的占位符•后面的每一个文法符号x计算*goto*(CC_0，x)。这产生距CC_0一个符号的所有集合。为了计算完整的规范集合，我们只需要迭代这一过程到一个不动点。

3. 构造算法

为了构造LR(1) 项目集合的规范集合，算法计算初始状态CC_0，然后系统地寻找从CC_0可以达到的LR(1) 项目的所有集合。它反复把*goto*运用于*CC*中的新集合；goto使用closure。如果目标产生式是$S'\rightarrow s$，那么这一结构是：

```
cc0 ← closure({[S'→•S, eof]})
CC ← {cc0}
while (new sets are still being added to CC)
  for each unmarked set ccj ∈ CC
    mark ccj as processed
    for each x following a • in an item in ccj
      temp ← goto(ccj, x)
      if temp ∉ CC
        then CC ← CC ∪ {temp}
      record transition from ccj to temp on x
```

如前所述，初始化*CC*为含有CC_0的集合。然后，它通过寻找从*CC*中的状态到*CC*外面的状态的转换系统地扩展*CC*。它构造性地完成这一工作，通过构建每个可能的状态*temp*，并测试*temp*是否是*CC*的成员。如果*temp*是一个新成员，它就把*temp*加入到*CC*中。无论*temp*是否是新成员，它都记录从CC_j到*temp*的转换以便为后来构建*goto*表使用。

为了保证算法对每个集合CC_i只处理一次，算法使用一个简单的标记方案。它在无标记条件下生成每一个集合，而在处理这一集合时，对其做标记。这可以显著减少调用*goto*和*closure*的次数。

像这一构造法中其他计算一样，这一计算也是一个不动点计算。规范集合*CC*是LR(1) 项目的幂集2^{ITEMS}的一个子集。*while*循环是单调的；它只把新集合加入到*CC*中。因为*CC*不可能超过2^{ITEMS}，因此这一计算一定终止。

4. SN的规范集合

扩充的*SheepNoise*文法的规范集合的计算过程如下所示:

用*closure*([*Goal*→•*SheepNoise*, *eof*]) 计算CC_0,

$$CC_0 = \left\{ \begin{array}{l} [Goal \rightarrow \bullet\ SheepNoise, eof],\quad [SheepNoise \rightarrow \bullet\ \texttt{baa}\ SheepNoise, eof], \\ [SheepNoise \rightarrow \bullet\ \texttt{baa}, eof] \end{array} \right\}$$

因为每个项目的右部开头都有•，CC_0只包含可能的状态。这是合适的，因为CC_0是分析器的初始状态。

125 *while*循环的第一次迭代产生两个集合CC_1和CC_2。

goto(CC_0, *SheepNoise*) 是CC_1。

$$CC_1 = \{\ [Goal \rightarrow SheepNoise \bullet, eof]\ \}$$

如果向前看符号是*eof*，那么项目[*Goal*→*SheepNoise*•，*eof*]是一个句柄，而且分析器应该接受这一输入字符串。

goto(CC_0, baa) 是CC_2。

$$CC_2 = \left\{ \begin{array}{l} [SheepNoise \rightarrow \bullet\ \texttt{baa}\ \ SheepNoise, eof],\quad [SheepNoise \rightarrow \bullet\ \texttt{baa}, eof], \\ [SheepNoise \rightarrow \texttt{baa} \bullet SheepNoise, eof],\quad [SheepNoise \rightarrow \texttt{baa} \bullet, eof] \end{array} \right\}$$

*while*循环的第二次迭代试图从CC_1和CC_2中得到新集合。对于任意文法符号x，goto(CC_1，x) 生成空集。对于$x = eof$和$x = Goal$，*goto*(CC_2, x) 生成空集。对于$x = SheepNoise$，它生成一个新集合CC_3。

goto(CC_2, *SheepNoise*) 是CC_3。

$$CC_3 = \{\ [SheepNoise \rightarrow \texttt{baa}\ SheepNoise \bullet, eof]\ \}$$

最后，*goto*(CC_2，baa) 是CC_2，而CC_2已在*CC*中。因为第三次迭代没有把新集合加入到*CC*中，所以上述过程终止。

图3-17给出规范集合构造法的过程。第一列表示迭代的序号。构造法计算CC_0并使用*goto*构建*CC*的其他集合。第一次迭代对*SheepNoise*构建CC_1，对baa构建CC_2。第二次迭代没有从CC_1得到任何集合。从CC_2开始，它对*SheepNoise*构建CC_3，对baa构建CC_2。第三次迭代没有从CC_3得到任何集合。

	项目	目标	SheepNoise	baa	eof
1	CC_0	∅	CC_1	CC_2	∅
2	CC_1	∅	∅	∅	∅
	CC_2	∅	CC_3	CC_2	∅
3	CC_3	∅	∅	∅	∅

图3-17 *SheepNoise*文法上的LR(1) 构造的轨迹

126 规范集合代表句柄识别DFA的状态。这一集合中的每个集合元素成为DFA的一个状态。状态之间的转换遵循生成这一集合的动作。例如，*goto*(CC_2, *SheepNoise*) 是CC_3。这给出从状态CC_2沿着文法符号*SheepNoise*到状态CC_3的一个转换。图3-18给出我们为SN构建的句柄识别DFA。

3.5.3 填充表格

给定*SN*的LR(1) 项目集合的规范集合，分析器生成器可以通过在*CC*上迭代并对每个$CC_j \in CC$检查这些项目来填充*Action*表和*Goto*表。每一个CC_j成为分析器的一个状态。它的项目在*Action*表的一列中生成非空

元素；在CC的构造期间所记录的相应的转换描述*Goto*的非空元素。有三种情况产生*Action*表中的条目。

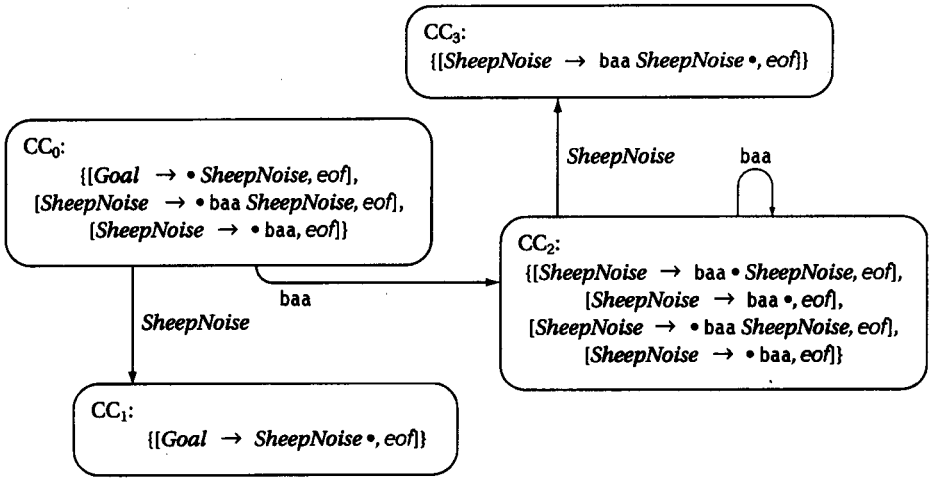


图3-18 *SheepNoise*文法*SN*的句柄识别DFA

1）[$A\rightarrow\beta\bullet c\gamma$, a]形式的项目表明遇到终结符号c将是向着发现非终结符号A的一个有效的下一步。因此，它在当前状态下生成一个对c的*移入*。语法识别器的下一个状态是通过计算对于当前状态和c的*goto* 127
所生成的状态。β或γ可以是ε。

2）[$A\rightarrow\beta\bullet$, a]形式的项目表示语法分析器已识别一个β，如果向前看符号是a，那么这个项是一个句柄。因此，在当前状态下，它对于a生成一个按$A\rightarrow\beta$的归约。

3）项目[$S'\rightarrow S\bullet$, *eof*]是惟一的。它表示语法分析器的接受状态；语法分析器已经识别归约到目标符号的输入流且向前看符号是*eof*。因此，这个项目生成当前状态下对*eof*的一个接受动作。

图3-19具体化这一过程。对于LR(1) 文法，图3-19将惟一定义*Action*表和*Goto*表中的非出错条目。

```
for each cc_i ∈ CC
  for each item I ∈ cc_i
    if I is [A→β • cγ,a] and goto(cc_i, c) = cc_j then
      Action[i,c] ← "shift j"
    else if I is [A→β•, a] then
      Action[i,a] ← "reduce A→β"
    else if I is [S'→S•, eof] then
      Action[i, eof] ← "accept"
  for each n ∈ NT
    if goto(cc_i, n) = cc_j then
      Goto[i, n] ← j
```

图3-19 LR(1) 表格填充算法

注意，表格填充算法实际上忽视占位符•在非终结符号前的那些项目。当•在终结符前时它生成移入动作。当•在产生式的右端时，它生成归约动作和接受动作。如果CC_i包含项目[$A\rightarrow\beta\bullet\gamma\delta$, a]，而$\gamma\in NT$时如何呢？虽然这一项目本身不生成任何表格中的条目，但是它迫使*closure*过程包含生成表格条目的各项目。当*closure*找到在非终结符号γ前的•时，对于以γ为左部的产生式，它在CC_i中具体化FIRST(γ)：*closure*寻找每一个$x\in$FIRST(γ) 并把相应的可能项目加入到CC_i中以生成对每个x的移入项目。

当然，表格填充动作可以合并到LR(1) 项目集合的规范集合的构造法中。 128

对于*SheepNoise*文法，LR(1) 项目集合的规范集合构造法生成以下两个表格：

Action表

状 态	eof	baa
0		s2
1	accept	
2	r3	s2
3	r2	

Goto表

状 态	SheepNoise
0	1
1	
2	3
3	

这些表格可以用于图3-13中的驱动器，生成*SN*的LR(1)分析器。

3.5.4 表构造法的出错

作为LR(1)表构造法的第二个例子，考虑典型的if-then-else结构的歧义文法，抽象控制表达式和所有其他语句的细节（通过把它们当作终结符）生成如下所示的四产生式文法：

这一文法有两个非终结符*Goal*和*Stmt*，六个终结符if、expr、then、else、assign以及伪式的*eof*。

1	*Goal*	→	*Stmt*
2	*Stmt*	→	if expr then *Stmt*
3		|	if expr then *Stmt* else *Stmt*
4		|	assign

构造法首先使用项目[*Goal*→•*Stmt*, *eof*]初始化CC_0同时通过*closure*产生第一个集合：

$$CC_0 = \left\{ \begin{array}{l} [Goal \rightarrow \bullet\, Stmt, eof],\ [Stmt \rightarrow \bullet\, \texttt{if expr then } Stmt, eof], \\ [Stmt \rightarrow \bullet\, \texttt{if expr then } Stmt \texttt{ else } Stmt, eof],\ [Stmt \rightarrow \bullet\, \texttt{assign}, eof] \end{array} \right\}$$

从这一集合出发，构造法逐渐得到LR(1)项目集合的规范集合的其余成员。

129 图3-20给出构造的过程。第一次迭代对于每一个文法符号检查从CC_0出发的转换。这从CC_0出发产生规范集合的三个新集合：对*Stmt*的转换得到的CC_1，对if的转换得到的CC_2，对assign的转换得到的CC_3。这些集合是：

$$CC_1 = \left\{ [Goal \rightarrow Stmt \bullet, eof] \right\},$$

$$CC_2 = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if} \bullet\, \texttt{expr then } Stmt, eof], \\ [Stmt \rightarrow \texttt{if} \bullet\, \texttt{expr then } Stmt \texttt{ else } Stmt, eof] \end{array} \right\}, \text{并且}$$

$$CC_3 = \left\{ [Stmt \rightarrow \texttt{assign} \bullet, eof] \right\}$$

130 第二次迭代检查从这三个新集合出发的转换。CC_1和符号expr的组合产生惟一的新集合：

$$CC_4 = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr} \bullet\, \texttt{then } Stmt, eof], \\ [Stmt \rightarrow \texttt{if expr} \bullet\, \texttt{then } Stmt \texttt{ else } Stmt, eof] \end{array} \right\}$$

第三次迭代检查从CC_2出发的转换并对符号then生成CC_5：

$$CC_5 = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr then} \bullet\, Stmt, eof], \\ [Stmt \rightarrow \texttt{if expr then} \bullet\, Stmt \texttt{ else } Stmt, eof], \\ [Stmt \rightarrow \bullet\, \texttt{if expr then } Stmt, \{eof, \texttt{else}\}], \\ [Stmt \rightarrow \bullet\, \texttt{assign}, \{eof, \texttt{else}\}], \\ [Stmt \rightarrow \bullet\, \texttt{if expr then } Stmt \texttt{ else } Stmt, \{eof, \texttt{else}\}] \end{array} \right\}$$

第四次迭代检查从CC_5出发的转换。对*Stmt*、if和assign分别生成如下集合：

$$CC_6 = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr then } Stmt\bullet, eof], \\ [Stmt \rightarrow \texttt{if expr then } Stmt \bullet \texttt{ else } Stmt, eof] \end{array} \right\},$$

$$CC_7 = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if} \bullet \texttt{expr then } Stmt, \{eof, \texttt{else}\}], \\ [Stmt \rightarrow \texttt{if} \bullet \texttt{expr then } Stmt \texttt{ else } Stmt, \{eof, \texttt{else}\}] \end{array} \right\}, \text{ 并且}$$

$$CC_8 = \{\ [Stmt \rightarrow \texttt{assign} \bullet, \{eof, \texttt{else}\}]\ \}$$

第五次迭代检查CC_6、CC_7和CC_8。虽然大多数组合生成空集合，但有两个组合导致新集合。从CC_6出发对else的转换得到CC_9，从CC_7对expr的转换生成CC_{10}：

$$CC_9 = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr then } Stmt \texttt{ else} \bullet Stmt, eof], \\ [Stmt \rightarrow \bullet \texttt{if expr then } Stmt, eof], \\ [Stmt \rightarrow \bullet \texttt{if expr then } Stmt \texttt{ else } Stmt, eof], \\ [Stmt \rightarrow \bullet \texttt{assign}, eof] \end{array} \right\}, \text{ 并且}$$

$$CC_{10} = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr} \bullet \texttt{then } Stmt, \{eof, \texttt{else}\}], \\ [Stmt \rightarrow \texttt{if expr} \bullet \texttt{then } Stmt \texttt{ else } Stmt, \{eof, \texttt{else}\}] \end{array} \right\}$$

当第六次迭代检查第五次迭代生成的集合时，它生成两个新集合，从CC_9对*Stmt*的转换得到的CC_{11}以及从CC_{10}对then的转换得到的CC_{12}。它还从CC_9开始生成两个已有的集合CC_2和CC_3： 131

$$CC_{11} = \{\ [Stmt \rightarrow \texttt{if expr then } Stmt \texttt{ else } Stmt \bullet, eof]\ \}, \text{ 并且}$$

$$CC_{12} = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr then} \bullet Stmt, \{eof, \texttt{else}\}], \\ [Stmt \rightarrow \texttt{if expr then} \bullet Stmt \texttt{ else } Stmt, \{eof, \texttt{else}\}], \\ [Stmt \rightarrow \bullet \texttt{if expr then } Stmt, \{eof, \texttt{else}\}], \\ [Stmt \rightarrow \bullet \texttt{if expr then } Stmt \texttt{ else } Stmt, \{eof, \texttt{else}\}], \\ [Stmt \rightarrow \bullet \texttt{assign}, \{eof, \texttt{else}\}] \end{array} \right\}$$

第七次迭代生成从CC_{12}对*Stmt*的转换得到的新集合CC_{13}，同时生成两个已有的集合CC_7和CC_8的复制：

$$CC_{13} = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr then } Stmt \bullet, \{eof, \texttt{else}\}], \\ [Stmt \rightarrow \texttt{if expr then } Stmt \bullet \texttt{ else } Stmt, \{eof, \texttt{else}\}] \end{array} \right\}$$

第八次迭代找到一个新集合：从CC_{13}对else的转换得到的新集合CC_{14}：

$$CC_{14} = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr then } Stmt \texttt{ else} \bullet Stmt, \{eof, \texttt{else}\}], \\ [Stmt \rightarrow \bullet \texttt{if expr then } Stmt, \{eof, \texttt{else}\}], \\ [Stmt \rightarrow \bullet \texttt{if expr then } Stmt \texttt{ else } Stmt, \{eof, \texttt{else}\}], \\ [Stmt \rightarrow \bullet \texttt{assign}, \{eof, \texttt{else}\}] \end{array} \right\}$$

第九次迭代从CC_{14}对*Stmt*的转换生成新集合CC_{15}，并生成两个已有的集合CC_7和CC_8：

$$CC_{15} = \{\ [Stmt \rightarrow \texttt{if expr then } Stmt \texttt{ else } Stmt \bullet, \{eof, \texttt{else}\}]\ \}$$

最后一次迭代检查CC_{15}。因为•位于CC_{15}中每个项目的最后，所以它只能生成空集。在这一点，没有新的项目集合加入到规范集合中，所以这一算法已达到不动点。它终止。

这一文法中的歧义性在表格填充算法中显现了出来。从状态CC_0到CC_{12}都不产生冲突。状态CC_{13}包含四个项目：

1) $[Stmt \rightarrow \texttt{if expr then } Stmt \bullet, \texttt{else}]$,

2) [*Stmt* → if expr then *Stmt* •, *eof*],

3) [*Stmt* → if expr then *Stmt* • else *Stmt*, else], 并且

132 4) [*Stmt* → if expr then *Stmt* • else *Stmt*, *eof*]。

项目1为CC_{13}和向前看符号else生成一个归约条目。项目3为表格中同样位置生成一个移入条目。显然，表格中的条目不能同时持有两个动作。这一移入归约冲突表明，该文法是歧义的。项目2和项目4对向前看符号*eof*生成类似的移入归约冲突。当表格填充算法遇到这样的冲突时，这一构造法就失败了。表格生成器应该向编译器设计者报告这一问题：在特定的LR(1)项目中产生式之间存在基本的歧义性问题。⊖

	项目	*Coal*	*Stmt*	if	expr	then	else	assign	*eof*
1	CC_0	∅	CC_1	CC_2	∅	∅	∅	CC_3	∅
2	CC_1	∅	∅	∅	∅	∅	∅	∅	∅
	CC_2	∅	∅	∅	CC_4	∅	∅	∅	∅
	CC_3	∅	∅	∅	∅	∅	∅	∅	∅
3	CC_4	∅	∅	∅	∅	CC_5	∅	∅	∅
4	CC_5	∅	CC_6	CC_7	∅	∅	∅	CC_8	∅
5	CC_6	∅	∅	∅	∅	∅	CC_9	∅	∅
	CC_7	∅	∅	∅	CC_{10}	∅	∅	∅	∅
	CC_8	∅	∅	∅	∅	∅	∅	∅	∅
6	CC_9	∅	CC_{11}	CC_2	∅	∅	∅	CC_3	∅
	CC_{10}	∅	∅	∅	∅	CC_{12}	∅	∅	∅
7	CC_{11}	∅	∅	∅	∅	∅	∅	∅	∅
	CC_{12}	∅	CC_{13}	CC_7	∅	∅	∅	CC_8	∅
8	CC_{13}	∅	∅	∅	∅	∅	CC_{14}	∅	∅
9	CC_{14}	∅	CC_{15}	CC_7	∅	∅	∅	CC_8	∅
10	CC_{15}	∅	∅	∅	∅	∅	∅	∅	∅

图3-20 If-then-Else文法的LR(1)构造法的过程

对于这种情况，冲突的出现是因为文法中的产生式2是产生式3的前缀。表格生成器可以按移入优先的方式解决这一冲突；这也迫使分析器识别较长的产生式并把else约束到最内的if。

歧义文法也能产生归约冲突。如果文法包含如下两个产生式：$A \rightarrow \gamma\delta$和$B \rightarrow \gamma\delta$，那么就产生归约冲突。如果某个状态包含项目[$A \rightarrow \gamma\delta$•，a]和[$B \rightarrow \gamma\delta$•，a]，那么它对于两个产生式的两个向前看符号a将生成两个冲突的归约动作。同样，这一冲突也反映文法中的一个基本的歧义性；编译器设计者必须重新制作文法以消去歧义性（参见3.6.3节）。

因为有许多自动化这一过程的分析器生成器，确定一个文法是否有LR(1)性质的一个方法是对这一文法调用LR(1)分析器生成器。如果这一调用过程成功，那么这一文法有LR(1)性质。

3.6 实践中的问题

即使是使用自动分析器生成器，编译器设计者也必须处理若干问题，以产生现实程序设计语言的健
133 壮、高效的语法分析器。本节讨论实践中发生的若干问题。

⊖ 通常，错误信息包括产生这一冲突的LR(1)项目。这是研究表构造法的另一个原因。

3.6.1 错误恢复

程序员通常要编译含有语法错误的代码。事实上，大多数程序员都认可编译器是发现这样的错误的最快方法。在这一应用中，编译器必须在对代码的一次语法分析中尽可能多地发现语法错误。这需要我们留意在错误状态处的语法分析器行为。

本章在给出的所有语法分析器当遇到语法错误时具有同样的行为：它们报告错误并停止。这防止编译器浪费时间去翻译不正确的程序。然而，这使编译器在每次编译至多发现一个语法错误。这样的编译器将使语法错误的发现成为一个漫长、痛苦的过程。

语法分析器应该在每次编译中尽可能多地发现可能存在的语法错误。这要求让语法分析器通过移动到可以继续分析的状态，从而从错误中恢复过来的机制。达到这一目标的一般方法是，选择语法分析器可以用于使输入与它的内部状态同步的一个或多个字。当语法分析器遇到一个错误时，它开始抛弃输入符号，直到它找到一个同步的字，然后重新设置它的内部状态，使其与同步字一致。

在分号作为语句分隔符的类Algol语言中，分号通常被用作同步字。当一个错误出现时，语法分析器反复调用扫描器，直到发现分号。然后，它把状态改变到一个成功识别完整语句的结果状态，而不是一个错误状态。

在递归下降分析器中，分析器代码可以简单地抛弃字直到发现分号。在发现分号的那一点，语法分析器可以将控制返回到分析语句的过程报告成功的地方。这可能包括操作运行时栈，或使用诸如C语言的`setjmp longjmp`结构的非局部跳转。

在LR(1) 分析器中，这种重新同步更加复杂。语法分析器抛弃输入，直到找到一个分号。接着，它向下扫描分析栈，直到发现一个使得*Goto*[*s*, *Statement*]是合法的、非错误条目的状态*s*。然后，错误恢复程序把状态*Goto*[*s*, *Statement*]压入栈并重新开始正常分析。

在表驱动分析器LL(1) 或LR(1) 中，编译器需要一个告诉语法分析器生成器同步地点的方法。使用错误产生式可以做到这一点。所谓的错误产生式就是它的右部包含表明错误同步地点的保留字和一个或多个同步记号。使用这样的结构，分析器生成器可以构造实现希望行为的错误恢复程序。 134

当然，错误恢复程序应该采取这样的步骤以保证编译器对语法上不正确的程序不尝试生成及优化代码。这要求在错误恢复设备与调用编译器各个部分的高层驱动器之间的一个简单的握手。

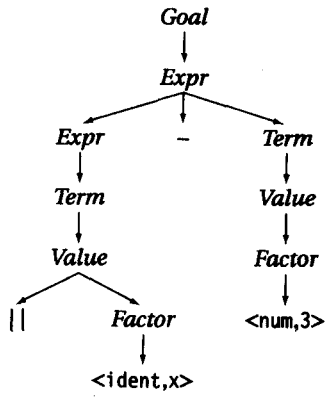
3.6.2 一元操作符

经典表达式文法只包含二元运算符。然而，标准代数表记既包含一元减、或否定操作符，又包含绝对值操作符。程序设计语言中还有其他一元操作符，包括布尔求补、自增、自减、类型转换、取地址和解除引用。把一元操作符加入到这种文法表达式中时需要小心。

考虑把一元绝对值操作符| |加入到经典表达式文法中。绝对值比 × 或 ÷ 有更高的优先权。然而，绝对值需要比*Factor*的优先权低以便在使用| |之前强制做括号表达式的评估。这导致以下文法：

$$
\begin{array}{lcl}
\textit{Expr} & \rightarrow & \textit{Expr} + \textit{Term} \\
 & | & \textit{Expr} - \textit{Term} \\
 & | & \textit{Term} \\
\textit{Term} & \rightarrow & \textit{Term} \times \textit{Value} \\
 & | & \textit{Term} \div \textit{Value} \\
 & | & \textit{Value} \\
\textit{Value} & \rightarrow & \mid\mid \textit{Factor} \\
 & | & \textit{Factor} \\
\textit{Factor} & \rightarrow & (\ \textit{Expr}\ ) \\
 & | & \texttt{num} \\
 & | & \texttt{ident}
\end{array}
$$

带着这些附加部分，文法仍然是LR(1) 的。它使程序员可以形成数的绝对值、标识符的绝对值以及括号
135 表达式的绝对值。字符串||x-3产生如下分析树:



这一分析树正确地表明这一代码必须在执行减法之前评估||x。这一文法不允许程序员书写|||x，因为它没有数学意义。然而，它却允许书写||(||x)，这与|||x同样没有意义。

不能书写|||x不会限定语言的表示能力。然而，对于其他一元操作符，这一问题似乎更严重。例如，C语言程序员需要书写**p对声明为char **p;的变量解除引用；我们也可以加入*Value*的解除引用产生式: *Value*→**Value*。结果文法仍是LR(1) 文法，即使我们在*Term*→*Term* × *Value*中用*中替换掉×，按C语言的方式重载“*”也是如此。也可以对一元减采用同样的做法。

3.6.3 处理上下文相关歧义性

使用一个字表示两个不同意义可以造成语法歧义性。在早期的若干程序设计语言，包括FORTRAN、PL/I和Ada的定义中出现了这一问题。在这些语言中，括号既将数组引用的下标表达式括起来，也将子程序或函数的参数列表括起来。给定一个文本引用，例如fee(i, j)，编译器无法知道fee是一个二维数组还是一个必须被调用的过程。为了分清这两种情况，我们需要了解fee的声明类型的信息。这一信息在语法上是不清楚的。对于任意一种情况，扫描器都毫无疑问地把fee归类为ident。函数调用和数组引
136 用可以出现于很多相同的情况中。

这些结构都不出现于经典表达式文法中。我们可以加入产生式，使其能够从*Factor*派生出来:

Factor	→	*FunctionReference*
	|	*ArrayReference*
	|	(*Expr*)
	|	num
	|	ident
FunctionReference	→	ident (*ExprList*)
ArrayReference	→	ident (*ExprList*)

因为最后两个产生式有相同的右部，这一文法是歧义的。这给LR(1) 表构建程序带来一个归约冲突。

解决这一歧义性需要语法之外的信息。在递归下降分析器中，编译器设计者可以简单地把*Function-Reference*和*ArrayReference*的代码结合起来，并加入检查这一ident的声明类型所需的额外代码。对于使用工具构建的表驱动分析器，该工具必须具有这样的解决方案。

多年来，我们使用两个不同的方法解决这一问题。编译器设计者可以重写文法，把函数调用和数组引用结合起来形成单一的产生式。在这样的方案中，这一问题被推迟到后期的翻译阶段，在那里可以通过声明的信息解决这一问题。语法分析器必须构造保存两种可能的所需信息的表示；后面的步骤将把这一引用重写成适当的形式：数组引用或函数调用。

另外，扫描器可以基于标识符的声明类型，而不是它们的微语法性质对其分类。这要求扫描器与语法分析器之间的某种握手；只要语言有“使用前定义”的规则，那么这样的合作就不是很困难。因为声明是在使用之前被分析的，所以语法分析器可以使它的内部符号表格对扫描器可用来解决把标识符分类成例如variable-name和function-name等不同类别的问题。相关的产生式变成

FunctionReference → function-name (*ExprList*)
ArrayReference → variable-name (*ExprList*)

137

按这种方式重写，文法是非歧义的。因为扫描器对于每种情况都返回一个不同的语法范畴，分析器可以区分这两种情况。

3.6.4 左递归与右递归

正如我们已经看到的那样，自顶向下分析器需要右递归文法而不是左递归文法。自底向上分析器可以适应于左递归文法和右递归文法两种情况。因此，编译器设计者在为自底向上分析器设计文法时可以在左递归和右递归之间做出选择。以下几个因素影响选择。

1. 栈的深度

一般地，左递归可以导致一个较浅的栈深度。考虑两个简单列表结构文法。（注意与*SheepNoise*文法的类似性。）

List	→	*List* elt	*List*	→	elt *List*
	|	elt		|	elt

使用两个文法的每个文法生成一个五元素的列表，我们有下面的派生：

List	*List*
List elt_5	elt_1 *List*
List $elt_4\ elt_5$	$elt_1\ elt_2$ *List*
List $elt_3\ elt_4\ elt_5$	$elt_1\ elt_2\ elt_3$ *List*
List $elt_2\ elt_3\ elt_4 elt_5$	$elt_1\ elt_2\ elt_3\ elt_4$ *List*
$elt_1\ elt_2\ elt_3\ elt_4\ elt_5$	$elt_1\ elt_2\ elt_3\ elt_4\ elt_5$

因为语法分析器以相反顺序构造这些序列，我们可以按从最底行向上读取每个派生来跟踪语法分析器的动作。

138

（1）左递归

这一文法把elt_1移入到它的栈中并马上把它归约成*List*。接着，这一文法把elt_2移入到栈中并把它归约成*List*。持续这一过程，直到已把每个elt_i移入到栈中并把它们归约到*List*。因此，栈达到的最大深度为2，且平均深度为$\frac{10}{6}=1\frac{2}{3}$。

（2）右递归

这一版本把所有五个elt_i都移入到栈中。接着，使用规则2把elt_5归约成*List*，使用规则1处理其余的

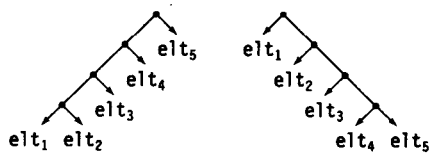
elt_i。因此，它的最大栈深为5，平均深度为$\frac{20}{6}=3\frac{1}{3}$。

右递归文法需要更多的栈空间；事实上，它的最大栈深被列表的长度惟一定界。相反，左递归文法的最大栈深度是文法的函数而不是输入流的函数。

对于短列表，栈空间不是问题。然而，如果列表表示很长的直线代码的语句列表，那么它可以有上百个元素。在这种情况下，空间的差异可能是巨大的。如果所有其他问题都一样的话，较小的栈深具有优越性。

2. 结合性

左递归自然地生成左结合性，右递归自然地生成右结合性。在某些情况下，评估的顺序产生很大的差异。考虑前面构造的五元素列表的抽象语法树（AST）。（AST可以消除分析树中的很多结点。）



左递归文法把elt_1归约成*List*，然后归约*List* elt_2，以此类推。这生成上图左边的AST。同样地，右递归文法生成上图右边所示的AST。

139

对于一个列表，这些顺序显然没有一个是显然正确的，尽管右递归的AST看起来可能更自然一些。然而，例如考虑按下面的文法用算术运算替换列表结构：

Expr	→	*Expr* + Operand	*Expr*	→	Operand + *Expr*
	|	*Expr* - Operand		|	Operand - *Expr*
	|	Operand		|	Operand

对于字符串$x_1 + x_2 + x_3 + x_4 + x_5$，左递归文法导致从左到右的评估顺序，而右递归文法导致从右到左的评估顺序。使用某个数系，如计算机上的浮点算术，这两种评估顺序可以产生不同的结果[⊖]。如果运算中的任意项是函数调用，那么评估顺序可能很重要。如果函数调用改变表达式中的变量值，那么评估顺序的改变也可能改变结果。

在带有减法的字符串，如$x_1 - x_2 + x_3$中，改变评估顺序可能产生不正确的结果。左结合在对树的后序遍历过程中将该字符串评估为$(x_1 - x_2) + x_3$，这是我们希望的结果。另一方面，右结合导致评估顺序$x_1 - (x_2 + x_3)$。当然，编译器必须保持语言定义指定的评估顺序。编译器设计者可以用如下两种方式保持指定的评估顺序：书写生成所希望的顺序的表达式文法，或者如4.5.2节所述，小心地生成中间表示以反映正确的评估顺序和结合性。

3.7 高级话题

为了构建一个令人满意的语法分析器，编译器设计者必须理解设计文法和语法分析器的基础。给定一个语法分析器，通常存在若干改进其性能的方法。本节考察语法分析器构造法中的两个话题，一个是优化文法以减小派生的长度（同时增加分析的速度）的方法，一个是减小LR(1) 分析器中*Action*表和*Goto*表大小的方法。

140

⊖ 因为浮点数相对于指数的范围而言具有较小的尾数，所以对于数量级相差很大的两个数，加法变成对于其中的较大数的恒等运算。例如，如果x_4远远小于x_5，那么处理器将计算$x_4 + x_5 = x_5$。对于参数的某些值，这一影响可能会叠加起来，导致从左到右和从右到左的评估给出不同的结果。

3.7.1 优化文法

虽然语法分析不再是消耗编译时间的主要因素，编译器不应在语法分析中无节制地浪费时间。文法的实际形式直接影响分析时所需的工作量。自顶向下和自底向上分析器都构建派生。自顶向下分析器对派生中的每个产生式执行一次展开。自底向上分析器对于派生中的每个产生式执行一次归约。产生较短派生的文法花费在分析的时间较少。

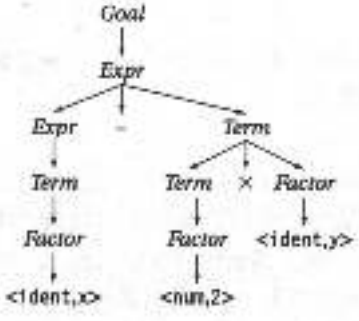
编译器设计者通常可以通过重写文法来降低分析树的高度。这可以减少自顶向下分析器中展开的数目，减少自底向上分析器中的归约数目。优化文法不能改变语法分析器的渐近行为：毕竟，分析树必须对输入流中的每个符号生成一个叶结点。然而，大幅度降低多用于文法部分，如表达式文法的（渐近）常数可以产生很大的差异。

再次考虑3.2.3节中的经典表达式文法。为了使各运算符拥有期望的优先权，我们加入了两个非终结符：*Term*和*Factor*，并且把这一文法重构成现在的形式：

```
1  Expr   →  Expr + Term
2         |  Expr - Term
3         |  Term
4  Term   →  Term × Factor
5         |  Term ÷ Factor
6         |  Factor
7  Factor →  ( Expr )
8         |  num
9         |  ident
```

这导致相当大的分析树，甚至是对于简单的表达式也是如此。正如我们在图3-10所看到的那样，x-2×y的分析树有十四个结点，其中五个是叶结点。（我们不能最优化掉这些叶子。文法的改变不能缩短输入程序！）

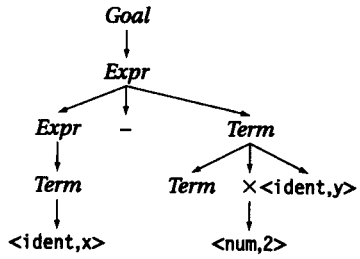
141



所有只有一个子结点的结点都是优化的候选。结点*Expr*到*Term*再到*Factor*再到<ident，x>这一序列对于输入流中的一个字使用了四个结点。通过在文法中向上折叠*Factor*的可能展开，我们至少可以消去*Factor*结点层。也就是说，使用*Factor*的三个选择取代在各产生式右部的*Factor*的各个出现。

```
Term  →  Term × ( Expr )  |  Term × ident  |  Term × num
      |  Term ÷ ( Expr )  |  Term ÷ ident  |  Term ÷ num
      |  ( Expr )         |  ident         |  num
```

这使*Term*的选择的数目增加2倍，但是，缩小一层分析树。



142

在自顶向下递归下降分析器中，这可以消去十四次过程调用中的三个。在LR(1) 分析器中，它消去九个归约动作中的三个，而移入的次数仍保持不变。

一般地，我们可以折叠任意右部为单一符号的产生式。这些产生式称为*无用产生式*（useless production）。有时，无用产生式出于某种目的：使文法更紧凑、也许是可读性更强，也许是迫使派生呈现某一特殊形式。（回想最简单的表达式文法接受x-2×y，但却不能在分析树描绘优先权。）正如我们将在第4章所看到的那样，编译器设计者可能单纯为了在派生中创建一个断点而包含无用产生式，以便在这样的断点实施特定的动作。

折叠无用产生式的这种变换也有它的代价。在LR(1) 分析器中，这种变换可以使表格增大。在我们的例子中，消除*Factor*使`Goto`表移出一列，但是，*Term*的额外产生式使*CC*的大小从32个集合增加到46个集合。因此，表格少一列，但是却增加14行。结果语法分析器拥有更少的归约（运行更快），但却有一张大表格。

在手工编码的递归下降分析器中，较大的文法可能增加在展开某个左部之前必须比较的选择的数目[⊖]。编译设计者也许能够通过结合各种情况来补偿这增加了的代价。例如，图3-7中的两个非平凡的*Expr'*展开的代码是相同的。编译器设计者可以用`word`与+或-的匹配检测把它们合并起来。另外，编译器设计者也可以设定+和-为相同语法范畴，让分析器检查这一语法范畴，并在需要时使用这个字的实际文本来区分二者。

3.7.2 减小LR(1) 表格的大小

正如图3-14和3-15所示，相对于较小的文法生成的LR(1) 表格也可能很大。有许多缩小这一表格的
143 技术。本节给出三种减小表格大小方法。

1. 合并行或列

如果表格生成器能够找到内容相同的两行或两列，那么表格生成器可以把它们合并起来。在图3-14中，对应于状态0，以及状态7到状态10的各行是相同的，它们是第4、14、21、22、24、25行。这一表格生成器可以实现这些集合各一次，然后重新映射这些状态。这将从表格移出九行，把它的大小减少28%。为了使用这一表格，框架分析器需要一个从分析器状态到`Action`表中的行下标的映射。这一表格生成器使用类似的方法合并相同的列。对于`Goto`表的独立检查将导致一个不同的状态组合集合，特别是只包含零的所有行将浓缩到一行中。

在某些情况下，表格生成器能发现两行或两列只在下面的情况下不同：其中的两行或两列中的一个有一个“错误”条目（在我们的图中记作空白。）在图3-14中，`eof`列和`num`列只在其中的一个有空白的地方不同。把这些列合并起来产生对正确的输入有相同行为的表格。在错误的输入上将改变分析器的行为，并有可能减弱分析器提供精确、有帮助的错误信息的能力。

⊖ 不要让这一讨论误导你认为使用这种方式修改的经典表达式文法是LL(1) 文法。回想这一文法需要修改以使它能够预测分析。同样的变换适用于这一文法的当前版本。

合并行和列可以直接减少表格的大小。如果这一空间的减少对每次表存取加入额外的间接处理，那么我们必须在这些内存操作的代价与内存的节约间进行直接的权衡。表格生成器也可以使用其他技术来表示稀疏矩阵。同样，实现者必须在存取代价的增加与内存大小间进行权衡。

2. 缩小文法

在很多情况下，编译器设计者可以重写文法来减小文法包含的产生式的数目。这通常导致较小的表格。例如，在经典表达式文法中，数字和标识符之间的差异与*Goal*、*Expr*、*Term*和*Factor*产生式无关。使用一个产生式*Factor*→val取代*Factor*→num和*Factor*→ident这两个产生式将使文法减少一个产生式。在*Action*表中，每个终结符都有自己的列。将num和ident折叠成单一符号val使*Action*表减少一列。在实践中，为了完成这一工作，对于num和ident，扫描器对num和ident必须返回到相同的语法范畴，或字。 144

类似的讨论也适于把×和÷合并成单一终结符muldiv，而把+和-合并成单一终结符addsub。这些替换中的每个都将消去一个终结符和一个产生式。以上在三个改变生成如下的缩小的表达式文法：

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* addsub *Term*
3		|	*Term*
4	*Term*	→	*Term* muldiv *Factor*
5		|	*Factor*
6	*Factor*	→	(*Expr*)
7		|	val

这一文法生成较小的CC，从表格中消去一些行。因为这一文法有较少的终结符，因此它也只有较少的列。

结果*Action*表和*Goto*表如图3-21所示。*Action*表包含132个条目，而*Goto*表包含66个条目，总共198个条目，与原来文法中的共计384个条目的表格相比非常好。改变文法将使表格的大小减小48%。然而，需要注意的是，这些表格仍然包含重复行。如*Action*表中的0、6和7行以及*Action*表中的4、11、15和17行是重复的。在*Goto*表中也有相同的行。如果我们对表格的大小非常关注的话，可以把这些技术结合起来使用。

对于其他方面的考虑也许会限制编译器设计者合并产生式的能力。例如，×操作符可能有多种用法，那么将它与÷合并起来就会变得不现实。同样地，语法分析器可能使用不同的产生式，以使语法分析器用不同的方法处理这两个语法上类似的结构。

3. 直接编码表格

作为最后一项改进，分析器生成器可以完全放弃表驱动框架分析器而支持手工编码实现。每个状态变成一个小的选择语句或一系列测试下一个字符的类型的if-then-else语句，以此来判断是移入、归约，接受还是报告错误。可以使用这种方法编写*Action*表和*Goto*表的条目。（2.5.2节已讨论了扫描器的类似转换。）

结果语法分析器回避我们在图中用空白表示的*Action*表和*Goto*表中的所有“无关”状态的直接表示。代码的增大可能抵消这一空间节约，因为此时每个状态都包含更多的代码。然而，新语法分析器没有分析表，不执行表查找，没有在框架分析器中的外部循环。虽然新语法分析器的结构变得令人几乎无法读懂，但是它要比相应的表驱动分析器运行得更快。使用适当的代码布局技术，结果语法分析器可以在结构缓冲器和分页系统两方面展示强大的局部化。典型例子是把表达式文法的所有过程放置在不会产生彼此冲突的单一页面上。 145 ~ 146

	Action表						Goto表		
	eof	addsub	muldiv	(	)	val	Expr	Term	Factor
0				s4		s5	1	2	3
1	acc	s6							
2	r3	r3	s7						
3	r5	r5	r5						
4				s11		s12	8	9	10
5	r7	r7	r7						
6				s4		s5		13	3
7				s4		s5			14
8		s15			s16				
9		r3	s17		r3				
10		r5	r5		r5				
11				s11		s12	18	9	10
12		r7	r7		r7				
13	r2	r2	s7						
14	r4	r4	r4						
15				s11		s12		19	10
16	r6	r6	r6						
17				s11		s12			20
18		s15			s21				
19		r2	s17		r2				
20		r4	r4		r4				
21		r6	r6		r6				

图3-21 缩小的表达式文法的表格

4. 使用其他构造算法

存在若干构造LR类型分析器的其他算法。其中包括SLR(1) 构造法和LALR(1) 构造法。SLR(1) 是简单(simple) LR(1) 的缩写，LALR(1) 是向前看 (lookahead) LR(1) 的缩写。这两个构造法生成的表格都比规范LR(1) 算法生成的表格小。

SLR(1) 算法接受的文法类比规范LR(1) 构造法接受的文法类小。这些文法局限于那些不需要LR(1) 项目中的向前看符号的文法。这一算法使用FOLLOW集合来区分分析器应该移入还是归约这两种情况。在实践中，这一机制足以处理很多实际的文法。通过使用FOLLOW集合，算法消除需要向前看符号的必要性。从而生成较小的规范集合以及相应的行数较少的表格。

LALR(1) 算法利用这样的观察：表示状态的集合中的某些项目是关键的，而其他项目则可以从那些关键项目派生出来。LALR(1) 的表结构只表示关键项目；从而生成与SLR(1) 构造法生成的规范集合等价的规范集合。表格的细节有些不同，但是表格的大小是一样的。

本章较早提出的LR(1) 构造法是这些表构造算法中最一般的算法。它生成最大的表格，但是接受最大的文法类。使用适当的表格减小技术，LR(1) 表格可以接近那些由有更多限制的技术生成的表格的大小。然而，作为一个非显然的结果，有LR(1) 文法的任意语言都有LALR(1) 文法和SLR(1) 文法。使用某种方法形成这些受到更多限制的文法形式将允许我们使用相应的构造算法来解决分析器何时该移入，何时该归约的问题。

3.8 概括和展望

几乎每个编译器都包含语法分析器。很多年来，语法分析是一个非常令人感兴趣的课题。从而导致

构建高效语法分析器的各种技术的发展。文法LR(1) 系列包含所有能够用确定的形式分析的上下文无关 147
文法。这些工具构建带有强大的错误检测功能的高效语法分析器。这些特性的结合，加之LR(1)、LALR(1)、SLR(1) 文法的诸多分析器生成器，减弱了人们对其他自动分析技术（例如，LL(1) 分析和算符优先权分析）的兴趣。

自顶向下、递归下降分析器有其自身的优越性。可以证论，它们是最易手工构建的语法分析器。它们提供发现和修改语法错语的最佳可能。编译器设计者较容易处理给LR(1) 分析器带来麻烦的源程序中的歧义性，诸如那些关键字名字可能作为标识符出现的语言中的歧义性。自顶向下、递归下降分析器是高效的；事实上，精心构造的自顶向下、递归下降分析器比表驱动LR(1) 分析器更快。(LR(1) 的直接编码设计可以战胜这一速度上的优势。）无论出于什么理由，想要构造手工编码分析器的编译器设计者都应该使用自顶向下、递归下降方法。

还有更一般的语法分析算法。然而在实践中，对于绝大多数程序设计语言，LR(1)、LL(1) 文法类对上下文无关文法的限制不会引起什么问题。

在LR(1) 和LL(1) 文法之间的选择，关键的问题是是否有合适的工具。事实上，很少有程序设计语言是处于LR(1) 文法与LL(1) 文法之间的。因此，以已有的适当分析器生成器开始总是要好过从头实现分析器生成器。

本章注释

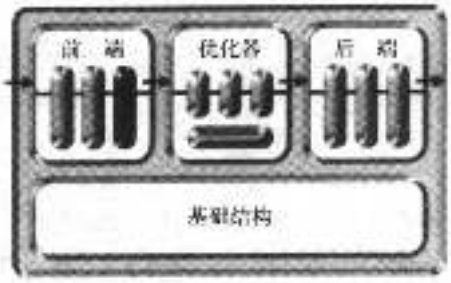
最早的编译器使用手写编码语法分析器[26，304，221]。Algol 60的丰富语法给早期的编译器设计者带来挑战。他们尝试各种方案来分析这一语言；Randell和Russell对各种Algol 60编译器所用的分析方法给出了迷人的概述[282，第1章]。

Irons是将语法从翻译中分离出来的第一人之一[191]。似乎是Lucas引入了递归下降分析的表记法[246]。Conway把类似的想法运用于COBOL的高效单遍编译器[91]。

LL和LR分析背后的思想出现于20世纪60年代。Lewis和Stearns描述了LL(k) 文法[236]；Rosenkrantz和Stearns更深入地描述了这些文法的性质[294]。Foster开发了把文法转换成LL(1) 形式的算法[144]。Wood形式化了提取左因子的概念并揭示了在把文法转换成LL(1) 形式的过程中涉及的理论问题 148
[336，337，338]。

Knuth奠定了LR(1)分析背后的理论[217]。随着SLR和LALR表格的出现，DeRemer等将这一理论付诸实践[113，114]。Waite和Goos说明了在LR(1) 表构造法算法中自动消除无用产生式的技术[326]。Penello提出直接将这些表格编写成可执行代码的方案[272]。对于LL和LR分析，Aho、Johnson和Ullman[9]是权威性的参考书。

分析任意上下文法无关文法的若干算法出现于20世纪60年代和20世纪70年代早期。Cocke和Schwartz[86]的算法、Younger[341]的算法、Kasami[201]的算法以及Earley[126]的算法都有类似的计算复杂性。Earley的算法值得一提，因为这一算法与LR(1) 表构造法算法类似。Earley算法在分析时，而不是在执行时得到可能分析状态的集合，而LR(1) 技术在分析器生成器中预先计算这些状态。从更高级的角度看，LR(1) 算法可以看成是Earley算法的一种自然的优化。 149



第4章
上下文相关分析

4.1 概述

编译器的终极任务是把输入程序翻译成能直接在目标机器上执行的形式。在它把这一输入程序翻译成目标机器的操作之前，它必须构建程序含有的基本信息。它必须知道如何表示值以及变量间的值的流向。它必须理解计算的结构。必须分析程序对外部文档和设备之间的相互作用。所有这些事实都可以从源代码以及某种上下文的信息中获取。然而，这要求编译器要做比扫描和语法分析更深入的调查。

考虑一个变量x。在编译器生成涉及x的计算的可执行目标机器代码之前，它必须回答很多问题。

- 存储在x中值的类型是什么？现代程序设计语言使用多种数据类型，包括若干类数、字符、布尔值，
151 指向其他对象的指针、文字量、集合（例如{red, yellow, green}）以及其他类型。大多数语言包括聚集各种值的复合对象：包括数组、结构体、集合和串。
- x有多大？因为编译器必须对x进行操作，它需要知道x在目标机器上的表示的长度。如果x是一个数，那么它的长度也许是一个字（整数或浮点数）、两个字（双精度浮点数或复数）或四个字（四倍精度浮点数或双精度复数）。对于数组和串，元素的数目可能在编译时固定下来，也可能在运行时确定。
- 如果x是一个过程，那么它需要什么样的参数？如果它返回值，那么返回值的类型是什么？在编译器生成调用过程的代码之前，它必须知道这个被调用的过程代码期望有多少个参数，在哪里找到这些参数，以及每个参数的值的类型。如果过程返回一个值，调用这一过程的代码在哪里可以找到这个值，这个值的类型是什么？（编译器必须保证调用程序用相容和安全的方式使用这个值。如果调用程序假设返回值是一个可以解除引用的指针，而被调用过程返回任意一个字符串，那么结果也许不能预测，不安全或不相容。）
- x的值必须保存多长时间？编译器必须保证x的值能够被所有可以合法引用x的计算的任意部分所存取。如果x是Pascal语言中的一个局部变量，那么编译器容易通过在声明x的过程的运行期间保存x的值而过多估算它的生存期。如果x是在任意位置都可被引用的一个全局变量，或者是由程序显式地分配的结构体的一个成员，那么编译器可能更难以决定它的生存期。编译器可以总是在整个计算中保存x的值；然而，关于x的生存期的更精确的信息可以使编译器能够对在生存期上与x不冲突的其他值复用x的空间。
- 谁负责分配x的空间？（以及是否初始化它？）是为x隐式地分配空间还是由程序显式地为它分配空间？如果是显式地分配空间，那么编译器必须假设在程序运行之前不知道x的地址。另一方面，
152 如果编译器是用它管理的运行时数据结构之一给x分配空间，那么关于x的地址编译器知道的更多。这一信息可能使编译器生成更有效的代码。

编译器必须从源程序和源程序的规则那里得到这些问题以及更多问题的答案。在类Algol语言，例如Pascal或C语言中，这些问题中的大多数可以通过检查x的声明而得到解答。如果语言没有声明，就像APL语言，编译器必须通过分析程序得到这类信息，或者它必须生成可以处理可能发生的所有情况的代码。

这些问题中的大多数超出了用上下文无关文法表示的源语言语法的范围。例如，x←y和x←z的分析树仅在赋值右侧的ident的文本是不同的。如果x和y是数而z是字符串，那么编译器也许需要对x←y产生不同于对x←z产生的代码。为了区别这种情况，编译器必须深入挖掘源程序的意义。扫描和语法分析仅处理程序的形式；意义的分析是*上下文相关分析*（context-sensitive analysis）的范围。

为了弄清楚语法与意义之间的差异，考虑在多数类Algol语言中的程序结构。这些语言要求每个变量在被使用之前必须被声明，而且变量的使用必须与其声明一致。

编译器设计者可以构造确保所有声明出现在可执行语句之前的语法。如下形式的产生式确保所有声明出现在可执行语句之前：

$$ProcedureBody \rightarrow Declarations\ Executables$$

其中，非终止符有显然的意义。这不对更深的规则进行检查，也就是说，不检查程序在可执行语句中第一次使用每一个变量之前是否声明它。它也不能处理诸如C++语言这样的简单情况，这类语言要求对某些范畴的变量在使用前先进行声明，但容许程序员穿插着书写声明和可执行语句。

强制要求“使用前声明”规则要求的信息层次更高，无法使用上下文无关文法来描绘。上下文无关文法处理的是语法范畴而不处理特定的字。因此，文法可以描述在表达式中变量名可能出现的位置，而且文法可以告知出现了变量名。然而，文法没有办法把一个变量名的一个实例与另外一个实例匹配；这一匹配要求文法描述更深层的分析，要求能够考虑上下文并能够检查和处理比上下文法无关语法更深层 153
信息的分析。

本章探讨需要上下文相关分析的若干问题，并且讨论执行上下文相关分析的框架。本章始终以类型检查为例来说明上下文相关分析的必要性以及在执行上下文相关分析时出现的问题。4.2节对类型系统和类型推理的必要性做概括性的介绍。为了执行类型推理和类似的关于上下相关语法的计算，我们引入两个框架：一个是4.3节中的属性文法形式，另一个是4.4节中的语法制导翻译的一个特定形态。高级话题一节概述在类型推理和类型检查中导致更困难问题的几种状况；它还给出语法制导翻译设计的例子。

4.2 类型系统概述

大多数程序设计语言都对每个数据值关联一系列性质。我们称这样的一系列性质为这个值的*类型*（type）。类型描述这一类型的所有值共有的一组性质。可以使用成员关系描述类型；例如，一个整数可能是在$-2^{31} \leqslant i < 2^{31}$范围内的任意的整数，red可能是定义为{red, orange, yellow, green, blue, brown, black, white}的枚举类型colors中的一个值。可以使用规则描述类型；例如，C语言中结构体的声明定义一个类型。在这里，类型包含在声明域中按声明顺序排列的任意对象；各声明域拥有描述值的允许范围和它们的解释的类型。（我们把一个结构体的类型表示成这一结构体的组成域的类型的按序乘积。）某些类型是程序设计语言预定义的；而程序员构造其他类型。程序设计语言中的类型集合，连同使用类型描述程序行为的规则，统称为*类型系统*（type system）。

4.2.1 类型系统的目的

程序设计语言设计者们引入类型系统，以便能够在比上下文无关文法更精确的层次上描述程序的行为。类型系统为描述合法程序的形式和行为生成另一种交流手段。根据类型系统的观点分析程序可以获 154
得使用扫描和语法分析技术所不能得到的信息。在编译器中，这一信息可以用于三个不同的目的：安全性、表示能力和运行时效率。

1. 确保运行时安全性

设计优良的类型系统有助于编译器检查并避免运行时错误。类型系统应该确保程序有良好的行为，也就是说编译器和运行时系统在执行一个引发运行时错误的操作之前可以识别所有不良程序。事实上，类型系统不能捕捉到所有不良程序；不良程序集合是不可计算的。诸如对一个超出范围的指针解除引用等的运行时错误有显然的影响（并且通常是灾难性的影响）。而诸如错误地把一个整数解释成浮点数等的运行时错误可能有微妙、累积性的影响。编译器应该使用类型检测技术尽可能多地消除运行时错误。

为了实现这一点，编译器必须首先为每个表达式推断它的类型。这些推断的类型暴露出某个值没有得到正确解释的一些情况，诸如在布尔值的地方使用了浮点数等。其次，编译器必须参照定义语言的规则检查每一个操作符的各操作数的类型是否是语言所允许的。在某些情况下，这些规则也许要求编译器把一个值从一种表示转换成另外一种表示。在其他情况下，这些规则也许禁止这样的转换，并仅仅声明程序是不良程序，因此是不可执行的。

在大多数语言中，编译器能够为每一个表达式推断一个类型。FORTRAN 77具有特别简单的、只有少数几个类型的类型系统。图4-1给出关于+操作符的所有情况。给定表达式a+b和a与b的类型，图4-1的表格描述a+b的类型。对于一个整数a和一个双精度数b，a+b产生一个双精度结果。

+	整 型	实 型	双 精 度	复 数
integer	integer	real	double	complex
real	real	real	double	complex
double	double	double	double	不合法
complex	complex	complex	不合法	complex

155

图4-1 FORTRAN 77中+运算的结果类型

相反，如果a是一个复数，那么a+b将是不合法的。编译器应该检查这一情况并在程序执行前报告这一错误，这就是一个简单的类型安全性的例子。

对于某些语言，编译器不能对所有的表达式推断其类型。例如，APL没有声明，它允许变量在任意赋值中改变类型，而且容许用户在输入提示下键入任意代码。虽然这些许可使得APL的功能和表示能力都很强大，但是我们需要它的实现必须做一定量的运行时类型推理和检测[⊖]。

安全性是使用类型语言的一个重要原因。在程序执行之前捕捉大多数类型相关的错误可以简化程序的设计和实现。可以对每一个表达式指定一个非歧义类型的语言称为*强类型语言*（strongly typed language）。可以在编译时对每一个表达式指定类型的语言称为*静态类型语言*（statically typed language）。只能在运行时才能对某些表达式指定类型的语言称为*动态类型语言*（dynamically typed language）。还有另外两种语言：*无类型*（untyped）语言，如汇编语言或BCPL，以及*弱类型*（weakly typed）语言，这一语言带用一个微弱的类型系统。

2. 改进表示能力

一个结构良好的类型系统使得语言设计者得以比上下文无关文法规则更精确地描述程序的行为。这种能力让语言设计者加入用上下文运无关文法不能描述的一些特征。一个非常好的例子就是操作符重载，它对操作符赋予上下文相关的意义。很多程序设计语言使用+表示几种加法。+的解释依赖于它的操作数的类型。在类型语言中，很多操作符被重载。而隐式类型语言则是对每一种情况提供词法上不同的操

⊖ 当然，这种选择假设程序行为良好且忽视这样的检测。通常当程序出错时，这导致不良的行为。在APL中，很多性能的改善高度取决于类型的有效性以及维数信息。

作符。

例如，在BCPL中，惟一的类型是“单元”。一个单元可以存放任意的位模式；这一位模式的解释由作用于该单元的操作符决定。因为单元本质上是无类型的，操作符不可被重载。因此，BCPL对整数加法使用+而对浮点加法使用#+。给定两个单元a和b，a+b和a#+b都是合法的表达式，它们都不对其操作数执行任意转换。

相反，即使是最古老的类型语言也都使用重载来描述复杂的行为。正如前面一节所述，FORTRAN有一个加法操作符+，它使用类型信息决定这一类型的实现。ANSI C使用函数原型，即函数参数的数目及其类型和函数的返回值类型的声明来把参数转换成适当的类型。类型信息决定C语言中自增指针的效应；指针的类型决定增加量。面向对象语言对每一次过程调用使用类型信息选择适当的实现。例如，Java语言通过检查构造器的参数列表在默认构造器和特定构造器之间做出选择。 156

3. 生成更好的代码

一个设计良好的类型系统为编译器提供程序中每一个表达式的详细信息，也就是可以用于生成更高效翻译的信息。考虑在FORTRAN 77中实现加法。编译器可以完全决定所有表达式的类型，所以它可以参考类似于图4-2所示的表格。右侧的代码显示加法的ILOC操作，连同对于混合类型表达式用FORTRAN标准所描述的转换代码。整个表格包括图4-1的所有情况。

a的类型	Type of b的类型	a+b的类型	代　码
integer	integer	integer	iADD r_a, r_b ⇒ r_{a+b}
integer	real	real	i2f r_a ⇒ r_{a_f} fADD r_{a_f}, r_b ⇒ r_{a_f+b}
integer	double	double	i2d r_a ⇒ r_{a_d} dADD r_{a_d}, r_b ⇒ r_{a_d+b}
real	real	real	fADD r_a, r_b ⇒ r_{a+b}
real	double	double	r2d r_a ⇒ r_{a_d} dADD r_{a_d}, r_b ⇒ r_{a_d+b}
double	double	double	dADD r_a, r_b ⇒ r_{a+b}

图4-2 FORTRAN 77中的加法的实现

在编译时不能决定所有类型的语言中，类型检查的某些部分可能要推迟到执行时进行。为了完成这一工作，编译器有必要发行类似于图4-3所示伪码的代码。 157

（这一代码给出与图4-2所示的各情况相同的子集。）虽然这保证运行时的安全，它却大幅度增加每一个操作的系统开销。编译时检测的一个目标就是在提供这样的安全性的同时在运行时不花费代价。

图4-3中的代码取决于运行代码可以决定操作数a和b的类型这一隐含的假设。为了实现这一点，编译器和运行时系统必须为每一个值标上它的类型，从而使得*type of* a成为对与存储a的真实值存放在一起的一个值的引用。在这个操作数被转换之后，这一代码还必须为结果值设置标签域。（我们在图4-3中省略了标签操作的代码。）标签操作带来的系统开销与加法的成本相比是相当大的。每一个操作都通过两个case选择语句。连同执行真实的加法一起，所选情况的代码执行所有必要的转换。这一代码读取两个标签写第三个标签。如果a和b储存于寄存器内，那么它们的标签应该也在寄存器内。这减小标签存取的代价，但却增加对寄存器的需求。（另一个选择是以a和b的空间的一部分为标签存储空间，并减小它们所能表示的值的范围。）

```
switch ( type of a ) {
  case integer:
    switch ( type of b ) {
      case integer: iADD ra,rb ⇒ ra+b
                    break
      case real:    i2f ra ⇒ raf
                    fADD raf,rb ⇒ raf+b
                    break
      case double:  i2d ra ⇒ rad
                    dADD rad,rb ⇒ rad+b
                    break
      default:      signal a run-time type error
                    break
    }
    break
  case real:
    switch ( type of b ) {
      case integer: i2f rb ⇒ rbf
                    fADD ra,rbf ⇒ ra+bd
                    break
      case real:    fADD ra,rb ⇒ ra+b
                    break
      case double:  f2d ra ⇒ rad
                    dADD rad,rb ⇒ rad+b
                    break
      default:      signal a run-time type error
                    break
    }
    break
  case double:
    switch ( type of b ) {
      case integer: i2d rb ⇒ rbd
                    dADD ra,rbd ⇒ ra+bd
                    break
      case real:    f2d rb ⇒ rbd
                    dADD ra,rbd ⇒ ra+bd
                    break
      case double:  dADD ra,rb ⇒ ra+b
                    break
      default:      signal a run-time type error
    }
    break
  default: signal a run-time type error
           break
}
```

158

图4-3 使用FORTRAN 77规则对+执行运行时检测和转换

编译时执行类型推断和检测消除这种系统开销。它可以使用图4-2所示的更快、更紧凑的代码取代图4-3的控制流和测试。在一般情况下，这些操作是不可避免的。(额外的信息可以进一步简化这一代码。例如，如果编译器知道加法的一个操作数是零，那么它就可以消除这一加法。)

4. 类型检测

为了获得这些益处，编译器必须分析程序并为每一个表达式赋予它所计算的类型。它必须检测这些类型以保证这些类型用在使它们合法的上下文中。这些活动通常统称为*类型检测*(type checking)。这是一个不恰当的名称，因为把赋值、推断、类型以及类型相关错误的检测这些不同的活动混在了一起。

程序员应该知道类型检测的执行方式。一个强类型、静态可检测语言也许是使用动态检测来实现的(或者甚至完全不做检测)。一个无类型语言可以用捕捉某种错误的方式实现。ML和Modula-3都是可以

使用静态检测的强类型语言的好例子。Common Lisp拥有必须使用动态检测的强类型系统。ANSI C是类型语言，但它的实现通常对违反类型做非常弱的检测。 159

类型系统的理论包括大而复杂的信息体系。本节对类型体系给出一个总体的认识，并引入检测中的一些简单问题。后续各节把类型推断的简单问题作为上下文相关计算的一个例子。

4.2.2 类型系统的组成部分

典型的现代语言的类型系统有四个主要组成成份：一组基本类型，即内建类型；从现存类型构造新类型的规则；决定两个类型是否等价或相容的方法；以及推断每一个源语言表达式的类型的规则。很多语言还包括基于上下文将值从一个类型隐式地转换到另外一个类型的规则。本节较详细地描述这四个组成部分，同时给出流行的程序设计语言的例子。

1. 基本类型

绝大多数程序设计语言或多或少包含下列基本类型：数、字符和布尔值。这些类型得到大多数处理器的直接支持。通常，数类型以若干种形式出现，例如整数类型和浮点数类型。有些语言加入其他基本类型。Lisp即包含有理数类型，还包含递归列表类型。有理数本质上是解释成比的整数对。列表定义为或者是特殊的nil，或者是序对（f，r），其中f是对象而r是列表。

这些基本类型的精确定义以及为它们所定义的操作符因语言不同而不同。一些语言细化这些基本类型以创造出更多的类型；例如，很多语言在它们的类型系统中区分若干种数的类型。而其他语言则缺少一个或多个这样的基本类型；例如，C缺少字符串类型。取而代之，C语言把字符串实现成字符数组的指针。几乎所有语言都包含从基本类型构造更复杂类型的装置。

（1）数

几乎所有程序设计语言都包含一种或多种数作为基本类型。典型地，这包括对有限区域的整数和通常称为浮点（floating-point）数的近似实数。很多程序设计语言通过为不同的硬件实现创建不同的类型以揭示相关的硬件实现。例如，C、C++和Java区分带符号整数和无符号整数。 160

FORTRAN、PL/I、Ada和C规定整数的长度。在PL/I中，程序员使用位来描述长度；然后，编译器把这一长度映射到整数的硬件表示上。PL/I的IBM 370实现把fixed binary(15)变量映射到16位整数，而把fixed binary(31)映射到32位整数。相反，C语言和FORTRAN使用相对术语描述长度。C语言的long的长度是short的长度的两倍。同样的关系对FORTRAN中的double和real也成立。然而，这两种语言的定义都把从长度描述到特定位长度的映射留给编译器。

某些语言详细地描述实现。例如，Java为长度为8、16、32和64位的带符号整数定义不同的类型。这些类型分别是byte、short、int和long。同样地，Java的float类型描述32位IEEE浮点数，而它的double类型描述64位IEEE浮点数。这种方法保证程序在不同的体系结构上有相同的行为。

Scheme采用不同的方式。这一语言的定义包含一个分层数类型，但让实现者选择支持的子集。然而，它的标准在精确数和非精确数之间画上一条严格的界线，并且规定某些操作在所有参数都是精确数时必须返回一个精确数。从而为实现者提供一定程度的自由度，同时又允许程序员推断什么时候在什么地方可以出现近似的情况。

（2）字符

很多语言包含字符类型。抽象地说，一个字符是一个单一字母。多年来，由于受到西方字母表的限制，这导致使用单一字节（8位）表示字符，通常被映射到ASCII字符集（或在IBM机器上的EBCDIC字符集。）最近，更多的实现，包括操作系统和程序设计语言都已开始支持由Unicode标准形式表示的更大字符集，这样的字符需要16位。大多数语言都假设字符集是有序的，这样，标准的比较操作符，如<、

=和>可以按字母顺序直观地进行工作。只有少数其他操作在字符数据上有意义。

（3）布尔类型

大多数程序设计语言包含取两个值true和false的布尔类型。布尔类型值的标准操作包括and、or、xor和not。布尔值或布尔值表达式通常用于决定控制流。C把布尔值认为是无符号整数的子区间，取值
161 限定于0（假）和1（真）。

2. 合成类型和结构类型

虽然程序设计语言的基本类型通常为直接受控于硬件的实际数据提供充分的抽象，但是作为表示程序所需的信息的符号，通常它们是不够的。程序经常要处理更复杂的数据结构，如图、树、表格、数组、列表以及栈。这些结构包含一个或多个对象，每个对象带有自身的类型。为这些合成对象或聚合对象构造新类型的能力是很多程序设计语言的本质特征。这一能力使程序员以新颖、程序特有的方式组织信息。把这些组织与类型系统结合增加编译器检测不良程序的能力。它也使语言可以表示高级操作，如对整个结构的赋值。

以Lisp为例，这一语言为程序设计提供列表的广泛支持。Lisp的列表是结构类型。一个列表或者是一个特殊值nil或者是（consfr），其中f是一个对象，r是一个列表，而cons是通过它的两个参数生成一个列表的构造器。Lisp的实现可以检查每一个cons调用以保证它的第二个参数是一个列表。

（1）数组

数组是最常见的聚合对象。数组把相同类型的多个对象组合起来，并给每一个对象一个不同的名字：虽然是隐式而非显式地、由程序员设计的名字。C的声明语句int a[100][200]；设置20 000个整数的空间，并且保证可以使用名字a对这些整数寻址。引用a[1][17]和a[2][30]存取不同且独立的内存单元。数组的本质性质是程序可以通过使用数（或其他有序，离散类型）来计算每一个元素的名字。

不同的语言对数组的操作的支持有巨大的差异。FORTRAN 90、PL/I和APL都支持对整个或部分数组的赋值。这些语言支持同时作用于数组各元素的操作。对于10 × 10的数组x、y和z，语句x=y+z使用y和z的相应元素的和重写x的每一个元素。APL比其他大多数语言在这一点上更先进；它包含内积、外积和若干种总括操作的操作符。（x的总括和，记作+/x，计算x的所有元素的和。）

数组可以看成是结构类型，因为我们通过描述数组元素的类型来构造它。因此，一个10 × 10的整数数组是具有类型整数的二维数组（two-dimensional array of integers）。一些语言在数组的类型中包含它
162 的维数；因此，一个10 × 10的整数数组与一个12 × 12的整数数组有不同的类型。这使得编译器作为类型错误捕捉维数不一致的数组操作。大多数语言允许任意基本类型的数组；而有些语言还允许结构类型的数组。

（2）串

一些程序设计语言把串看成是结构类型。例如，PL/I即有位串又有字符串。对于这两种类型所定义的各种性质、属性和操作都是相似的；它们都具有串的性质。位串与字符串在各位置上所允许的值的范围不同。因此，把它们分成位串（string of bit）和字符串（string of character）是合适的。（大多数支持串的语言局限于支持单一的串类型，也就是字符串。）其他语言通过把字符串当作字符数组来支持字符串。

真正的串类型与数组类型在若干重要方面是不同的。在串上有意义的操作，如联结、转换和长度计算等操作对数组来说也许没有相应的操作。从概念上说，串比较应该按照字典顺序进行，因此有“a” < “boo”及“fee” < “fie”。可以按自然的方式重载并使用标准比较操作符。字符数组的比较类似于数的数组或结构数组的相应比较，可能不同于串的比较。同样地，串的实际长度可能不同于分配给它的大小，而数组的大多数使用都是使用所有分配的元素。

（3）枚举类型

很多语言允许程序员创建包含一组特定常量值的类型。这一概念是在Pascal语言引入的，它允许程序员使用有意义的名字来记述一小组常量。典型例子包括一周中的日子和月份。使用C语言的语法，这些可能是：

```
enum WeekDay {Monday, Tuesday, Wednesday,
              Thursday, Friday, Saturday, Sunday};

enum Month {January, February, March, April,
            May, June, July, August, September,
            October, November, December};
```

编译器把枚举类型中的每一个元素映射到不同的值。枚举类型的元素是有序的，所以同类型的元素之间的比较是有意义的。在上述例子中，Monday<Tuesday及June<July都有意义。不同枚举类型间的操作没有意义，例如Tuesday>September将产生一个类型错误。Pascal保证每一个枚举类型的行为同整数的某 163
个子区域一样。例如，程序员可以声明以枚举类型的元素为下标的数组。

结构的另一种观点

对于结构的典型观点是把不同的结构体看成是不同的类型。结构类型的这一方式与其他聚合类型，例如数组和串的处理方式一致。这似乎很自然，对于程序员来说这样的区分很有用。例如，一个带有两个子结点的树结点可能应该与带有三个子结点的树结点有不同的类型；大概它们被用于不同的情况。把带有三个子结点的结点赋值给带有两个子结点的结点的程序应该对程序员生成一个类型错误和一个警告信息。

然而，从运行时系统的角度看，把每一个结构看成不同的类型使描述复杂化。对于不同的结构类型，堆包含来自于任意一组类型的任意一组对象。这使我们难以对诸如垃圾回收器等直接处理堆对象的程序进行推断。为了简化这样的程序，它们的作者有时对结构类型采用不同的处理方式。

这另一种模型把程序中所有的结构考虑成单一类型。各个结构声明各自创建类型*structure*的一个变形。类型*structure*本身是所有这些变形的并集。这一方式使程序把堆看成单一类型的一组对象，而不是很多类型的一组对象。这使得操作堆的代码更容易分析和优化。

（4）结构和变量

结构把任意类型的多个对象组合到一起。结构的元素，或成员通常被显式地命名。例如，在C语言中实现分析树可能需要带有一个或两个子结点的结点。 164

```
struct Node1 {
  struct   Node1 *left;
  unsigned Operator;
  int      Value
}

struct Node2 {
  struct   Node2 *left;
  struct   Node2 *right;
  unsigned Operator;
  int      Value
}
```

结构的类型是它的元素的类型的有序积。因此，我们可以把类型Node1描述成（Node1 *）× unsigned × int，而将Node2描述成（Node2 *）×（Node2 *）× unsigned × int。这些新类型具有与基本类型同样的基本性质。将指针通过自动增加指向Node1和通过类型转换指向Node1 *时的效果相同，它们的行为与基本类型的行为类似。

很多程序设计语言允许创建作为多个类型的联合的类型。例如，某个变量x可以有类型integer或boolean或WeekDay。Pascal语言使用可变记录来实现这一点。记录（record）是Pascal语言称呼结构的术

语。C语言则使用union来实现这一点。union类型是它的成份类型的并；因此，我们的变量x有类型integer∪boolean∪WeekDay。联合也可以包含不同类型的结构，各个类型的长度也可以不同。语言必须提供对每一个域进行非歧义引用的机制。

(5) 指针

为使程序员操作任意的数据结构，很多语言都提供指针类型。指针抽象地址。指针允许程序保存地址并在其后检查存放于该地址中的对象。当对象被创建时我们创建指针。(在Java语言中用new来创建对象，在C语言中用malloc来创建对象。)一些语言还包含返回对象的地址的操作符，例如C的&操作符。(当寻址操作符被用于一个类型*t*的对象时，它返回指向类型为*t*的指针的值。)

为了保护程序员远离某些与指针相关的错误，例如使用类型*t*的指针引用类型*s*的结构，一些程序设计语言将指针赋值限定在“等价”类型的范围。在这些语言中，赋值左边的指针必须与赋值右边的表达式有相同的类型。程序可以合法地把*指向整数的指针*(pointer to integer)赋给声明为指向整数的指针变量，但是却不能赋给声明为*指向整数指针的指针*(pointer to pointer to integer)变量或*布尔指针*(pointer to boolean)变量。

当然，创建新对象的机制应该返回相应类型的对象。因此，Java语言的new明确地创建一个具有类型的对象；其他语言使用以返回类型为参数的多态过程。ANSI C采用一种与众不同的方法处理这一问题；标准的分配过程malloc返回一个指向void的指针。这迫使程序员对malloc的每一次调用返回的值进行类型转换。

一些语言允许更复杂的指针操作。指针上的算术，包括自增指针和自减指针允许程序构建新指针。C语言使用指针的类型来决定增加或减小的幅度。程序员可以把一个指针设置为一个数组的开始；自增
165 指针使其从数组的一个元素指向下一个元素。

指针类型的安全性取决于一个隐式的假设：地址对应于具有类型的对象。创建新指针的能力使得这一假设变得有些含糊不清，而且严重降低了编译器和运行时系统对基于指针的计算进行推理的能力。(参见8.3节的例子)。

3. 类型等价

任何类型系统的重要组成部分都是它用于决定两个不同的类型声明是否等价的机制。考虑下面的C语言的两个声明：

```
struct TreeNode {
  struct TreeNode *left;
  struct TreeNode *right;
  int value
}
```

```
struct SearchTree {
  struct SearchTree *left;
  struct SearchTree *right;
  int value
}
```

TreeNode和SearchTree是一样的类型吗？它们等价吗？具有非平凡类型系统的任意程序设计语言必须包含对于所有类型回答这一问题的明确规则。

历史上，有两个一般方法。第一个方法是*名字等价*(name equivalence)，它主张两个类型等价当且仅当它们有相同的名字。从哲学的角度看，这一规则主张程序员可以为一个类型选择任意的名字；如果程序员选择不同的名字，语言及其实现应该尊重这种有意的行为。遗憾的是，一旦一个程序有多个作者或多个文件，维护名字的一致性实际上是不可行的。

第二个方法是*结构等价*(structural equivalence)，它主张两个类型等价当且仅当它们有相同的结构。从哲学的角度看，这一规则主张如果两个对象以相同的顺序组成相同的域集合，并且这些域都有等价的类型，那么这两个对象可以互换。结构等价检查定义这一类型的本质性质。

这两个策略都有其优点和缺点。名字等价假定相同的名字是有意的行为；在大型程序设计项目中，

这要求程序回避无意识的冲突。结构等价假定可交换对象可以安全地相互用于另一对象；如果某些值有“特殊”的意义，那么这可能引发问题。（想像两个假定的结构相同的类型。第一个类型有一个系统I/O控制块，而第二个类型有一组屏幕位映射图像的信息。把它们当作两个不同的类型使得编译器能 166
够检查到一个错误，即将I/O控制用于屏幕更新过程的错误，而把它们当作相同类型时则不会发生这个错误。）

4. 类型推断规则

通常对每一个操作符，类型推断规则描述操作数类型与结果类型之间的映射。对于某些情况，这一映射很简单。例如，赋值有一个操作数和一个结果。这一结果，或称左部必须有与操作数或右部类型相容的类型。（在Pascal中，区间1..100与整数是相容的，因为这一区间的任意元素都可以安全地被赋值给一个整型变量。）这一规则允许将整数值赋值给整型变量。它禁止将结构赋值给整型变量，除非明确地进行有意义的类型转换。

操作数类型与结果类型之间的关系通常描述成表达式树的类型上的递归函数。作为这一操作的操作数的类型的函数，这一函数计算操作的结果类型。这一函数可能描述成表格形式，类似于图4-1中的表格。有时候，操作数类型与结果类型之间的关系可以用简单的规则描述。例如在Java语言中，把不同精度的两个整数类型加起来产生一个较精确（较长）类型的结果。

类型推断规则可以识别类型错误。在FORTRAN表格中，某些组合是被禁止的，例如double与complex的组合。这一组合产生一个类型错误；这样的程序是不良的。Java语言禁止将一个数赋值给一 167
个字符。对于程序员，这两种类型错误都产生错误信息。

一些语言要求编译器修复混合类型表达式中的某些类型错误。当编译器找到这样的错误时，它必须插入生成相容类型值的转换。在FORTRAN 77中，整数和浮点数的加法要求在加法之前把整数强制类型转换成浮点值。同样地，关于不同精度的整数值间的加法的Java规则迫使将精度低的值转换成精度较高的形式。Java语言中的整数赋值可能需要强制类型转换。如果右部的精度较低，那么它就会被转换成左部更高精度的类型。然而，如果左部的精度比右部低，那么赋值产生一个类型错误（除非程序员插入明确的类型转换操作来改变这些类型。）

声明和推断

如前所述，很多程序设计语言包括一个“使用前声明”规则。由于强制声明，每一个变量都有明确定义的类型。编译器需要给常量指定类型的方法。通常有两种指定方法。一种是常量的形式示意特定的类型。例如，2表示一个整数而2.0表示一个浮点数，另一种是编译器根据它的使用推断常量的类型，例如，sin（2）表示2是一个浮点数，而对于x←2，若x是整型变量则表示2是一个整数[⊖]。使用变量的声明类型、已推断出的常量类型和一组完整的类型推断规则，编译器可以给变量和常量上的任意表达式指定类型。正如我们将看到的那样，函数调用使这一过程变得复杂。

一些语言完全不需要程序员书写任何声明。在这些语言中，类型推断的问题变得更复杂。4.5节描述相关的问题并给出编译器解决这些问题的一些常用的技术。

5. 表达式的类型推断

类型推断的目标是为出现在程序中的每一个表达式指定一个类型。类型推断的最简单的情况发生在编译器可以为表达式中的每一个基础元素指定类型的时候。这时，编译器可以为表达式的分析树的每一个叶子指定一个类型。类型推断需要所有变量的声明、所有常量的已推断出的类型，以及所有函数的类型信息。 168

⊖ 遗憾的是，这意味着“2”在不同的上下文中有不同的意思。经验表明程序员善于理解这样的重载。

类型系统分类

有很多术语被用于描述类型系统。在本书中，我们已介绍了*强类型*（strongly typed）语言、*无类型*（untyped）语言和*弱类型*（weakly typed）语言等术语。类型系统间的不同以及它们的实现间的不同也很重要。

1. 检查实现与无检查实现的对比

程序设计语言的实现可以选择进行充分的检查来发现并阻止由于类型的错误使用而引发的所有运行时错误。（这实际上可以排除一些特定值引发的错误，例如，以零为除数的除法。）这样的实现被称为*强检查*（strongly checked）。与强检查实现相对的是*无检查实现*（unchecked implementation），它假设程序是形式良好的。这两极之间的实现都是执行部分检查的*弱检查实现*（weakly checked implementations）。

2. 编译时活动与运行时活动的对比

强类型语言具有所有推断和检查都可在编译时进行的性质。在编译时实际完成所有这些工作的实现称为*静态类型*（statically typed）和*静态检查*（statically checked）。一些语言具有必须在运行时进行类型化并进行检查的结构。我们把这些语言称为*动态类型*（dynamically typed）和*动态检查*（dynamically checked）。更加混乱的是，编译器设计者可以用动态检查实现强类型、静态类型语言。Java语言是静态类型和静态检查语言的例子，只是它的执行模型不让编译器一下子就看到所有的源代码。这迫使我们在进行类型装入时执行类型推断并在运行时执行某些检查。

从概念上讲，对分析树进行一次简单的后序遍历，编译器就可以给表达式中每个值指定类型。这使编译器发现每一个对推断规则的违规，并*在编译时*报告这样的违规。如果这一语言缺乏某些特征而无法进行这样的简单推断的话，那么编译器将需要使用更复杂的技术。如果编译时类型推断变得太困难，编译器设计者也许有必要把一些分析和检查移到运行时去做。

对于表达式类型引用的简单情况，可以直接遵守表达式的结构。推断规则用源程序的术语描述这一
169 问题。分析树的评估策略是自底向上地进行的。基于这些原因，表达式类型推断已成为展示上下文相关分析的一个典型例子。

6. 类型推断中过程间的相关问题

本质上，表达式的类型推断取决于形成可执行成程序的其他过程。甚至在最简单的类型系统中，表达式也包含函数调用。编译器必须检查这些调用。编译器必须确保每一个实际参数与相应的形式参数是相容的。它必须决定在其后的推断所用的所有返回值的类型。

为了分析和理解过程调用，编译器需要为每一个函数做一个*类型署名*（type signature）。类型署名描述形式参数和返回值的类型。例如，C语言的标准函数库中的`strlen`函数取类型为`char *`的一个操作数，并返回包含以字节为单位的这个操作数的长度的`int`型的值，其中null终止符不计在内。在C中，程序员可以使用如下形式的*函数原型*（function prototype）来记录这一事实。

```
unsigned int strlen(const char *s);
```

这一函数原型声明`strlen`取类型为`char *`的参数，而且它不更改该参数，这由`const`表示。函数返回一个非负整数。用更抽象的表记法写出这一函数，我们可以写成：

```
strlen : const char * → unsigned int
```

上式读作“`strlen`是一个取一个常量值字符串并返回一个无符号整数的函数”。作为第二个例子，典型

的Scheme函数filter有类型署名:

filter: $(\alpha \rightarrow boolean) \times list\ of\ \alpha \rightarrow list\ of\ \alpha$

也就是说，filter是一个取两个参数的函数。第一个参数应该是一个把某一类型α映射到一个布尔类型，写作（$\alpha \rightarrow boolean$）的函数，而第二个参数应该是一个元素类型仍为$\alpha$的列表。给定这些类型的参数，filter返回一个元素类型为α的列表。函数filter展示参数多态性（parametric polymorphism）：它的结果类型是它的参数类型的函数。 170

为了执行精确的类型推断，编译器需要为每一个函数做一个类型署名。编译器能够以几种方法获取这一信息。编译器可以取消分块编译：需要把整个程序作为编译的一个单位。编译器可以要求程序员为每一个函数提供类型署名；这通常以强制函数原型的形式出现。编译器可以把类型检查推迟到链接时或运行时，当所有这些信息都可用时进行。最后，编译器设计者可以把编译器嵌入到程序开发系统中。这样的系统汇集必要的信息并在需要时将这些信息传给编译器。所有这些方法都可在现实的系统中看到。

4.3 属性文法框架

为执行上下文相关分析而提出的一种形式是*属性文法*（attribute grammar），或属性化上下文无关文法。属性文法在上下文无关文法的基础上增加了一组描述计算的规则。每一个规则使用其他属性的值定义一个值或*属性*（attribute）。规则将这一属性与特定的文法符号结合起来；出现在分析树中的文法符号的每一个实例都有这一属性的相应实例。因为属性实例与分析树结点之间的这样的关系，我们通常把属性域添加到分析树的结点上，以此来描述我们的实现。

为了使这些表记法具体化，考虑带符号二进制数的上下文无关文法。文法SBN=(T, NT, S, P) 的定义如下所示:

$$T = \{+, -, 0, 1\}$$

$$NT = \{Number, Sign, List, Bit\}$$

$$S = \{Number\}$$

$$P = \left\{ \begin{array}{ll} Number \rightarrow Sign\ List & \\ Sign \rightarrow + & Sign \rightarrow - \\ List \rightarrow List\ Bit & List \rightarrow Bit \\ Bit \rightarrow 0 & Bit \rightarrow 1 \end{array} \right\}$$

$L(SBN)$ 包含所有带符号二进制数，如，−101、+11、−01和+11111001100。它将无符号二进制数，如10排除在外。 171

从*SBN*出发，我们可以构建注解带符号二进制数所表示的值*Number*的属性文法。为了从上下文无关文法构建一个属性文法，我们必须决定每一个结点所需的属性，必须为产生式详细描述定义这些属性的值的规则。对于我们的*SBN*属性文法，我们需要下面的属性:

符　号	属　性
Number	*value*
Sign	*negative*
List	*position, value*
Bit	*position, value*

在这种情况下，终结符不需要属性。

图4-4给出带有属性规则的SBN的产生式。当某个符号多次出现在一个产生式中时，我们在相应文法符号上加上下标，以此来消除对规则中的符号引用的歧义性。因此，在产生式5和相应的规则中，*list*的出现都有下标。

	产 生 式	属 性 规 则
1	*Number* → *Sign List*	*List.position* ← 0
		if Sign.negative
		then Number.value ← - *List.value*
		else Number.value ← *List.value*
2	*Sign* → +	*Sign.negative* ← *false*
3	*Sign* → -	*Sign.negative* ← *true*
4	*List* → *Bit*	*Bit.position* ← *List.position*
		List.value ← *Bit.value*
5	$List_0 \rightarrow List_1\ Bit$	$List_1.position \leftarrow List_0.position + 1$
		$Bit.position \leftarrow List_0.position$
		$List_0.value \leftarrow List_1.value + Bit.value$
6	*Bit* → 0	*Bit.value* ← 0
7	*Bit* → 1	$Bit.value \leftarrow 2^{Bit.position}$

图4-4 带符号二进制数的属性文法

这些规则使用产生式中涉及的其他符号的属性以及文字常量来描述某个属性的值。从而允许规则从左到右传递信息，也允许另一方向的信息流传输。产生式4取决于两个方向的信息传输。第一个规则把*Bit.position*设置为*List.position*，而第二个规则把*List.value*设置为*Bit.value*。存在更简单的属性设计；我们特别选择这一设计以展示双向属性传输。

给定上下文无关文法中的一个串，属性规则把*Number*.*value*设置为二进制输入串的十进制值。例如，串－101引发图4-5左边所示的属性。（其中的属性名被缩写。）注意*Number.value*有值－5。

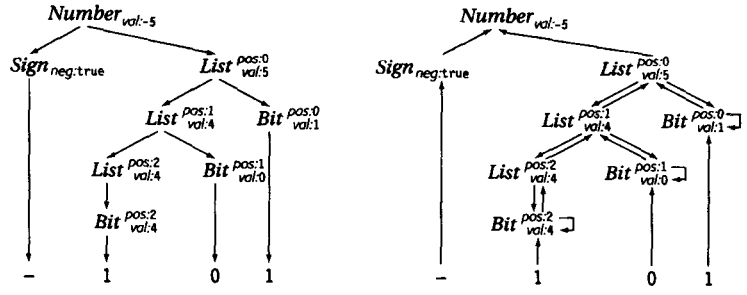


图4-5 带符号二进制数－101的属性树

为了评估*L*(*SBN*) 中的某个语句的属性分析树，我们使用各规则所描述的属性对分析树中各结点进行实例化。例如，这为每一个*List*结点中的value和position生成属性实例。每一个规则定义一组相关性；被定义的属性取决于规则的各个参数。处理整个分析树后，这些相关性形成一个*属性相关图*(attribute-dependence graph)。图中的边刻画规则评估中值的流向；从$node_i.field_j$到$node_k.field_l$的边表示

定义$node_k.field_l$的规则以$node_i.field_j$的值为它的一个输入。图4-5的右边给出串－101的分析树的相关图。

我们早前提到的值的双向传输（例如，产生式4中的双向传输）导致这一相关图中同时存在表示向根（*Number*）方向从下往上传输的弧和向叶方向从上往下传输的弧。*List*结点最清楚地给出这一效应。我们基于值的流向来区分属性。

（1）合成属性（synthesize attribute）

通常对于一个结点的属性，当它的值是由该结点的子结点的属性定义时，称为合成属性。通常，我们通过自底向上的传输计算分析树中合成属性的值。（叶结点不可能有合成属性。）

（2）继承属性（inherited attribute）

对于一个结点的属性，当它的值是由结点自身的属性、兄弟结点的属性以及父结点的属性定义时，称为继承属性。通常，我们通过自顶向下和横向的传输计算分析树中继承属性的值。（根结点不能有继承属性。）

在图4-5所示的例子中，属性*value*和*negative*是合成属性，而属性*position*是继承属性。

评估属性的任何设计框架都必须遵循蕴涵于属性相关图中的关系。每个属性必须由某个规则定义。如果这个规则取决于其他属性的值，那么直到所有这些值都被定义为止，不能评估这个规则。如果这一规则不取决于其他属性的值，那么规则必须根据某个常量或外部的信息生成属性值。只要不存在取决于其自身的值的规则，那么这些规则就应该惟一定义每一个值。

当然，规则可以直接或间接地引用其自身的结果。包含这样规则的属性文法称为循环（circular）文法。我们暂时忽略循环文法；4.3.2节将讨论这一问题。

相关图刻画这样一个值的传输：评估程序在评估属性树的实例时必须遵守值的传输方向。如果文法不是循环的，它在属性上附加一个偏序。这一偏序决定定义各属性的规则何时可以被评估。评估顺序与规则在文法中出现的顺序无关。

考虑最上方的*List*结点，即*Number*的右子结点相关的规则的评估顺序。这一结点是由产生式5即*List*→*List Bit*生成的；运用这个产生式把3个规则加入到评估中。为*List*结点的子结点设置继承属性的两个规则必须首先执行。它们依赖于*List.position*的值并为这一结点的子树设置属性*position*。设置*List*结点的*value*属性的第三个规则直到这两棵子树都已定义了*value*属性时才能执行。因为只有在*List*结点处的前两个规则被评估后这些子树才能被评估，因此在评估序列中，前两个规则的评估较早而第三个规则的评估较晚。

为了创建并使用属性文法，编译器设计者为文法中的每一个符号决定一组属性，并为计算这些属性的值设计一组规则。这些规则描述对任给合法分析树的计算。为了创建一个实现，编译器设计者必须创建一个评估程序；可以手工创建评估程序，也可以使用评估程序生成器。后者更具吸引力。评估程序生成器以属性文法的描述为输入。作为输出，它生成评估程序的代码。这就是属性文法对编译器设计者有吸引力的地方；这样的工具采用高级、非过程性的描述，并且自动地生成实现。

属性文法背后的一个重要的思想就是属性规则与上下无关文法的产生式相关联的表记方式。因为这些规则是函数式的，它们生成的值与评估顺序无关，只要顺序反映了蕴涵在属性相关图中的关系即可。在实践中对于所有规则来说，在它的所有输入都已被定义后才进行评估的任意评估顺序都满足这一相关性。

4.3.1 评估方法

仅当我们能够构建出通过解释这些规则的评估程序来自动地评估问题实例时，比如一个具体的分析树，属性文法才具有实际意义。文献已经非常详细地提出了很多属性评估技术。通常这些技术可以划分成下面三个类别。

1. 动态方法

这些技术使用特定属性分析树的结构来决定评估顺序。Knuth关于属性文法的原始论文提出按类似于数据流计算机体系结构的方式操作的评估程序：每个规则只要当它的所有操作数都可用时“被激活”。在实践中，可以使用一个已可评估属性的队列来实现这一方法。当一个属性被评估后，检查它在属性相关图中的所有后继是否已经准备就绪（参见12.3节中的“列表调度”的描述）。与此相关的设计框架构
175 建属性相关图，对其进行拓扑排序，并使用这一拓扑顺序来评估属性。

2. 遗忘方法

在这些方法中，评估顺序既独立于属性文法又独立于特定的属性分析树。大体上，系统的设计者选择一个相信是既适合于属性文法又适合于评估环境的方法。这一评估风格的例子包括反复从左到右扫描（直到所有属性都有值）、反复从右到左扫描，以及从左到右、从右到左交替扫描。这些方法具有简单的实现和相对较少的运行时负荷。当然，这些方法缺乏属性化了的特定树的信息所带来的任何改进。

3. 基于规则的方法

基于规则的方法通过对属性文法进行静态分析来构建一个评估顺序。在这一框架下，评估程序依赖于文法的结构；因此，分析树指导规则的运用。在带符号二进制数的例子中，产生式4的评估顺序应该是，使用第一个规则设置*Bit.position*，递归向下传递到*Bit*，然后在返回时使用*Bit.value*来设置*List.value*。同样地，对于产生式5，评估顺序应该是，先评估前两个规则来定义右部的属性*position*，然后递归向下到每一个子结点。返回时，它评估第三个规则来设置父结点*List*的*List.value*域。执行必要的离线静态分析的工具可以生成基于规则的快速评估程序。

4.3.2 循环性

循环属性文法可能引发循环属性相关图。当相关图含有循环时，我们的评估模型就会失败。编译器的这种失败会引发严重的问题，例如编译器也许不能为它的输入生成代码。相关图中的循环的这种灾难性的影响表明这一问题值得关注。

如果编译器使用属性文法，那么它必须有适当处理循环性问题的方法。有两个可行的方法。

1. 回避

编译器设计者可以把属性文法限制到一个不能引起循环相关图的集合上。例如，限制文法只使用合
176 成属性消除产生循环相关图的可能性。存在非循环属性文法的更一般集合；其中的某些文法，如强非循环属文法（strongly noncircular attribute grammar）具有多项式时间成员资格测试。

2. 评估

编译器设计者可以使用给每一个属性，甚至那些包含循环的属性指定值的评估方法。评估程序可以在循环上迭代直到这些值达到一个不动点。这样的评估程序可以回避由完全属性树失败引发的问题。

在实践中，大多数属性文法系统只注意非循环文法。如果属性文法是循环的，那么基于规则的评估方法也许不能构建一个评估程序。遗忘方法和动态方法会设法评估循环相关图；它们可能在定义某些属性实例时失败。

4.3.3 扩展例子

为了更好地理解作为描述语言语法上的计算的工具的属性文法的优缺点，我们仔细考察两个更详细的例子，一个为一个简单的类Algol语言的表达式树推断类型，另一个估测直线代码序列的执行时间。

1. 推断表达式类型

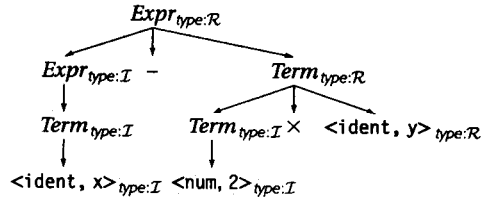
设法为类型语言生成有效代码的任何编译器都要面临为程序中每一个表达式推断类型的问题。本质上，这一问题依赖于上下文相关的信息；与ident或num相关的类型依赖于它们的身份，即它们的文本名字，而不是它们的语法范畴。

考虑由第3章给出的经典表达式文法而得到的表达式的类型推断问题的一个简化版本。假设这些表达式表示成分析树，而且表示ident或num的任意结点都已有*type*属性。（我们将在后面阐述把类型信息变成这些*type*属性的问题。）对于文法中的每一个算术操作符，我们需要一个把两个操作数的类型映射到结果类型的函数。我们分别称这些函数为F_+、F_-、$F_\times$和$F_\div$；这些函数描绘在表中可以找到如图4-1所示的信息。使用这些假设，我们可以写出对树中每个结点定义*type*属性的属性规则。图4-6给出这些属性规则。 177

产生式			属性规则
$Expr_0$	→	$Expr_1$ + *Term*	$Expr_0.type \leftarrow \mathcal{F}_+(Expr_1.type, Term.type)$
	|	$Expr_1$ - *Term*	$Expr_0.type \leftarrow \mathcal{F}_-(Expr_1.type, Term.type)$
	|	*Term*	$Expr_0.type \leftarrow Term.type$
$Term_0$	→	$Term_1$ × *Factor*	$Term_0.type \leftarrow \mathcal{F}_\times(Term_1.type, Factor.type)$
	|	$Term_1$ ÷ *Factor*	$Term_0.type \leftarrow \mathcal{F}_\div(Term_1.type, Factor.type)$
	|	*Factor*	$Term_0.type \leftarrow Factor.type$
Factor	→	(*Expr*)	*Factor.type* ← *Expr.type*
	|	num	num.*type is already defined*
	|	ident	ident.*type is already defined*

图4-6 推断表达式类型的属性文法

如果x有类型integer（记作I）且y有类型real（记作R），那么上面的框架对输入字符串x-2×y生成如下所示的属性分析树：



叶结点有正确初始化的*type*属性。其余的属性由图4-6的规则来定义，其中假设F_+、F_-、$F_\times$和$F_\div$反映FORTRAN 77的规则。

仔细观察属性规则表明，所有属性都是合成属性。因此，分析树中的所有相关性都是从子结点流向它的父结点。这样的文法有时被称为*S属性文法*（S-attributed grammar）。这种类型的属性具有简单、基于规则的评估框架。它与自底向上分析十分吻合；每一个规则可以在语法分析器通过相应的右部进行归约时得到评估。这一属性文法*范式*（paradigm）非常适合我们的问题。它的描述很短，也很容易理解。它导致一个高效的评估程序。 178

对属性表达式树的仔细观察显示两种情况：在这两种情况中，操作符的一个操作数的类型不同于操作结果的类型。在FORTRAN 77中，这要求编译器在该操作数与操作符之间插入一个转换操作。对于表示2与y的乘积的*Term*结点，编译器将把2从整数表达式转换成实数表达式。对于树的根结点处的*Expr*结

点，编译器将把x从整数转换成实数。遗憾的是，更改分析树不能很好地适应属性文法范式。

为了在属性树中表示这些转换，我们可以把一个保留转换后的类型的属性添加到每一个结点上，并加入设置该属性的相应规则。另外，我们还可以依赖于从分析树生成代码的过程在遍历的过程中比较父结点和子结点的类型，并插入必要的转换。前面的方法给属性评估遍增加一些工作，它局部化对一个分析树结点的转换所需要的所有信息。后面的方法将转换工作推迟到代码生成，它的代价是类型和转换的信息需要跨越编译器的两个不同的部分。这两个方法都有效；它们之间的差异主要是侧重点不同的问题。

2. 一个简单的执行时间估测器

作为第二个例子，考虑估测一系列赋值语句的执行时间问题。我们通过把三个新产生式加入到经典表达式文法来生成赋值语句序列。

$$
\begin{array}{lcl}
\textit{Block} & \rightarrow & \textit{Block Assign} \\
 & | & \textit{Assign} \\
 & & \\
\textit{Assign} & \rightarrow & \texttt{ident} = \textit{Expr;}
\end{array}
$$

其中*Expr*来自表达式文法。结果文法非常简单，因为它只允许把简单标识符当作变量且不含函数调用。但是，它也够复杂足以用于调查在估测运行时行为中引发的复杂性。

图4-7给出估测赋值语句块的执行时间的属性文法。属性规则估测这一语句块的循环计数，假设处理器一次执行一个操作。与推断表达式类型的属性文法类似，这一文法只使用合成属性。估测出现于分析树最上端*Block*结点的*cost*属性中。方法很简单。自底向上计算代价；读例子时，我们从*Factor*的产生式开始向上进行，直到*Block*的产生式。函数*cost*返回给定的ILOC操作的等待时间。

179

产 生 式			属性规则
$\textit{Block}_0$	→	$\textit{Block}_1$ *Assign*	{ $\textit{Block}_0$.*cost* ← $\textit{Block}_1$.*cost* + *Assign.cost* }
	|	*Assign*	{ $\textit{Block}_0$.*cost* ← *Assign.cost* }
Assign	→	`ident` = *Expr;*	{ *Assign.cost* ← *Cost*(`store`) + *Expr.cost* }
$\textit{Expr}_0$	→	$\textit{Expr}_1$ + *Term*	{ $\textit{Expr}_0$.*cost* ← $\textit{Expr}_1$.*cost* + *Cost*(`add`) + *Term.cost* }
	|	$\textit{Expr}_1$ − *Term*	{ $\textit{Expr}_0$.*cost* ← $\textit{Expr}_1$.*cost* + *Cost*(`sub`) + *Term.cost* }
	|	*Term*	{ $\textit{Expr}_0$.*cost* ← *Term.cost* }
$\textit{Term}_0$	→	$\textit{Term}_1$ × *Factor*	{ $\textit{Term}_0$.*cost* ← $\textit{Term}_1$.*cost* + *Cost*(`mult`) + *Factor.cost* }
	|	$\textit{Term}_1$ ÷ *Factor*	{ $\textit{Term}_0$.*cost* ← $\textit{Term}_1$.*cost* + *Cost*(`div`) + *Factor.cost* }
	|	*Factor*	{ $\textit{Term}_0$.*cost* ← *Factor.cost* }
Factor	→	(`Expr`)	{ *Factor.cost* ← *Expr.cost* }
	|	`num`	{ *Factor.cost* ← *Cost*(`loadI`) }
	|	`ident`	{ *Factor.cost* ← *Cost*(`load`) }

图4-7 估测执行时间的简单属性文法

3. 改进执行代价估测器

为了使这一例子更具实际意义，我们以通过编译器处理变量的方式改进它的模型。代价估测属性文法的初期版本，假设编译器为每一次变量的引用生成一个独立的load操作。对于赋值语句x=y+y，这一 180
模型对y做两次装入操作。很少有编译器对y生成多余的装入。更有可能的是，编译器将生成如下所示的只一次装入y的序列：

```
loadAI  r_arp,@y ⇒ r_y
add     r_y,r_y  ⇒ r_x
storeAI r_x      ⇒ r_arp,@x
```

为了更好地接近编译器的行为，我们可以修改属性文法，对于语句块中的每个变量只考虑进行一次装入的负荷。这需要更复杂的属性规则。

为了更精确地对装入计数，规则必须通过变量名跟踪对每一个变量的引用。这些名字超出了文法范围，因为文法跟踪语法范畴ident而不是跟踪各个名字，如x、y、z等。ident的规则应遵循如下一般框架：

```
if (ident has not been loaded)
   then Factor.cost ← Cost(load);
   else Factor.cost ← 0;
```

实现这一工作的关键是检测“ident *has not been loaded.*”

为了实现这一检测，编译器设计者可以加入一个属性来保存所有已装入变量的集合。产生式*Block*→*Assign*可以初始化这一集合。规则必须沿着表达式树将这一集合传递到每一个赋值。这意味着每一个结点带有两个集合*Before*和*After*。每个结点的*Before*集合包含较早出现在*Block*中的所有ident的名字；这些名字中的每一个必须是已装入的。一个结点的*After*集合包含在它的*Before*集合内的所有名字，再加上在以该结点为根的子树中应该被装入的所有ident。

*Factor*的扩展规则如图4-8所示。这一代码假设它可以得到每一个ident的文本名。派生（*Expr*）的第一个产生式把*Before*集合向下拷贝到*Expr*子树上，并且把*Expr*子树的*After*集合向上拷贝到*Factor*上。派生num的第二个产生式只将其父结点的*Before*集合拷贝到这一父结点的*After*集合。Num必须是树中的一个叶子；因此，不需要更进一步的动作。派生ident的最后一个产生式执行最关键的工作。它检测*Before*集合，决定是否需要一次装入，并相应修改父结点的*cost*和*After*属性。 181

产 生 式	属 性 规 则
Factor → (*Expr*)	{ *Factor.cost* ← *Expr.cost;* *Expr.Before* ← *Factor.Before;* *Factor.After* ← *Expr.After* }
| num	{ *Factor.cost* ← *Cost*(loadI); *Factor.After* ← *Factor.Before* }
| ident	{ *if* (ident.*name* ∉ *Factor.Before*) *then* *Factor.cost* ← *Cost*(load); *Factor.After* ← *Factor.Before* ∪ { ident.*name* } *else* *Factor.cost* ← *0;* *Factor.After* ← *Factor.Before* }

图4-8 跟踪Factor产生式中装入的规则

为了完成上面的描述，编译器设计者必须加入在分析树上拷贝*Before*集合和*After*集合的规则。有时被称为拷贝规则的这些规则把各个*Factor*结点的*Before*集合和*After*集合联系起来。因为属性规则只能引用局部的属性，即一个结点的父结点、兄弟结点和子结点的属性，这一属性文法必须显式地拷贝分析树上的值以确保它们的局部性。图4-9给出这一文法中其他产生式所需的规则。另外一个规则被加入进来；它将第一个赋值（*Assigh*）语句的*Before*集合初始化为空集。

产 生 式	属 性 规 则
$Block_0 \rightarrow Block_1\ Assign$	{ $Block_0.cost \leftarrow Block_1.cost +$ *Assign.cost*; *Assign.Before* $\leftarrow Block_1.After$; $Block_0.After \leftarrow$ *Assign.After* }
| *Assign*	{ $Block_0.cost \leftarrow$ *Assign.cost*; *Assign.Before* $\leftarrow \emptyset$; $Block_0.After \leftarrow$ *Assign.After* }
Assign → *Identifier* = *Expr*;	{ *Assign.cost* $\leftarrow$ *Cost*(store) + *Expr.cost*; *Expr.Before* $\leftarrow$ *Assign.Before*; *Assign.After* $\leftarrow$ *Expr.After* }
$Expr_0 \rightarrow Expr_1 + Term$	{ $Expr_0.cost \leftarrow Expr_1.cost +$ *Cost*(add) + *Term.cost*; $Expr_1.Before \leftarrow Expr_0.Before$; *Term.Before* $\leftarrow Expr_1.After$; $Expr_0.After \leftarrow$ *Term.After* }
| $Expr_1 - Term$	{ $Expr_0.cost \leftarrow Expr_1.cost +$ *Cost*(sub) + *Term.cost*; $Expr_1.Before \leftarrow Expr_0.Before$; *Term.Before* $\leftarrow Expr_1.After$; $Expr_0.After \leftarrow$ *Term.After* }
| *Term*	{ $Expr_0.cost \leftarrow$ *Term.cost*; *Term.Before* $\leftarrow Expr_0.Before$; $Expr_0.After \leftarrow$ *Term.After* }
$Term_0 \rightarrow Term_1 \times Factor$	{ $Term_0.cost \leftarrow Term_1.cost +$ *Cost*(mult) + *Factor.cost*; $Term_1.Before \leftarrow Term_0.Before$; *Factor.Before* $\leftarrow Term_1.After$; $Term_0.After \leftarrow$ *Factor.After* }
| $Term_1 \div Factor$	{ $Term_0.cost \leftarrow Term_1.cost +$ *Cost*(div) + *Factor.cost*; $Term_1.Before \leftarrow Term_0.Before$; *Factor.Before* $\leftarrow Term_1.After$; $Term_0.After \leftarrow$ *Factor.After* }
| *Factor*	{ $Term_0.cost \leftarrow$ *Factor.cost*; *Factor.Before* $\leftarrow Term_0.Before$; $Term_0.After \leftarrow$ *Factor.After* }

图4-9 跟踪装入的拷贝规则

这一模型比简单的模型复杂得多。它的规则是简单模型的三倍多；我们必须写出、理解、评估每一个规则。它既使用合成属性又使用继承属性；简单的自底向上评估策略不再适用。最后，处理*Before*和*After*集合的规则需要额外注意：我们希望通过使用基于高级描述的系统来回避这样的低级描述。

182 ~ 183

4. 再谈表达式类型推断

在最初的关于推断表达式类型的讨论中，我们假设ident.*type*和num.*type*属性是由某一外部机制

定义好的。为了使用属性文法填充这些值，编译器设计者需要为处理声明的文法部分开发一组规则。

这些规则需要为与声明语法相关的产生式中的每一个变量记录其类型信息。这些规则需要收集信息以便使用较少的属性包含所有已声明变量的必要信息。这些规则需要把这一信息传播到分析树中所有可执行语句的祖先结点上，然后向下拷贝到每个表达式中。最后，在每个ident或num叶结点处，规则需要从汇集的信息中提取正确的事实。

这样的规则与我们为跟踪装入所开发的规则有很多相似之处，但是在细节上这些规则更复杂。这些规则还创建巨大、复杂且必须在分析树上到处拷贝的属性。在朴素的实现中，拷贝规则的每一个实例都将创建一个新的拷贝。这其中一些拷贝是共享的，但是由多个子结点的信息而创建的大部分拷贝将不一致（因此，需要不同的拷贝）。在前面的例子中的*Before*集合和*After*集合也引发同样的问题。

5. 对执行代价估测器的最后改进

虽然跟踪装入改进了估测执行代价的真实度，还有可能对其做进一步的改进。例如，考虑这一模型上有限数目的寄存器的影响。至此，我们的模型一直假设目标计算机提供无限定数目的寄存器。在现实中，计算机提供较少的寄存器。为了给寄存器组的能力建模，估测器可以限制允许出现在*Before*集合和*After*集合中的值的数量。

作为第一步，我们必需替换*Before*和*After*的实现。之前它们被实现成任意大小的集合；在这一改进的模型中，寄存器组应该刚好拥有k个值，其中k是可以用来存放变量值的寄存器数。下一步，我们必须重写产生式*Factor*→ident的规则来建模寄存器的占有状况。如果一个值还没有被装入，而且寄存器是可用的，那么它负责这一简单的装入。如果一个装入是必须的，但是没有可用寄存器，那么它负责逐出某些寄存器的值并进行这一装入。选择逐出哪些值是非常复杂的；有关这一内容将在第13章讨论。因为*Assign*的规则总是负责存储，所以内存中的值一定是最近的。因此，当一个值要被逐出时，没有存储 184
的必要。最后，如果这个值已经装入且它仍在寄存器中，那么不需要代价。

这一模型使*Factor*→ident的规则集合变复杂，并要求稍微复杂化一些的初始条件（这是对*Block*→*Assign*的规则的要求）。然而，这一模型没有复杂化所有其他产生式的拷贝规则。因此，模型的精确性没有显著增加使用属性文法的复杂性。所有增加的复杂性只局限于直接处理这一模型的几个规则。

4.3.4 属性文法方法的问题

前面的例子展示在使用属性文法执行分析树上的上下文相关计算时引发的很多计算问题。其中的一些问题形成编译器中属性文法使用的特定问题。特别地，在编译器前端大多数属性文法的使用中都假设属性结果必须得到保存，通常是以属性分析树的形式进行保存。本节详细论述我们在前面例子中所看到的问题的影响。

1. 处理非局部信息

一些问题直接反映到属性文法范式，所有信息以相同方向传输的那些问题尤其如此。然而，带有复杂的信息传输模式的问题很难表示成属性文法。属性规则只能定义与出现于同一产生式的文法符号相关联的值；这迫使规则只使用附近或局部的信息。如果计算需要一个非局部值，那么属性文法必须包含能明确地把这些值拷贝到分析树上被使用的结点上。

这些拷贝规则能够增大属性文法的大小；将图4-7与图4-8和图4-9做比较就可清楚地看到这一点。实现者必须写出所有这些规则。在评估程序中，必须执行每一个规则，创建新属性并进行额外的工作。当信息是被整合在一起时，就如在使用前声明规则或估测执行时间框架中那样，每当规则改变整合值都需 185
要这一信息的一份新拷贝。这些拷贝规则在设计和评估属性文法的任务中又增加了一层工作。

2. 存储管理

对于现实的例子，评估产生大量属性。在分析树上使用移动信息的拷贝规则将成倍增加评估所创建的属性实例的数量。如果文法把信息聚集成复杂的结构，例如在分析树上传递声明信息，那么各个属性就会很大。评估程序必须为属性管理存储；不好的存储管理框架可能在评估程序的资源需求上产生不成比例的巨大的负面影响。

如果评估程序可以决定哪些属性值在评估后可以使用的话，那么它也许能够通过回收不再使用的属性值空间来复用部分属性存储。例如，将表达式树评估为一个值的属性文法也许把这个值返回到调用这个表达式的过程中。在这样的情况下，在内部结点处计算而得的中间值也许是废弃的，即不再使用，从而也就成了回收的候选。另一方面，如果关于属性的树结果需要被保留且在以后需要检查，例如类型推断的属性文法的情况，那么评估程序必须假设编译器的后面的分析必须遍历树并检查任意属性。在这样的情况下，评估程序不能回收这一属性任何实例的存储。

（这一问题实际上反映了属性文法范式的函数性本质与在编译器中对其进行命令式的使用之间的冲突。在编译器的后面阶段对属性的可能使用增加了属性的相关性，而这样的相关性是在属性文法中没有加以描述的。这歪曲了函数范式，丢弃了它的一个强项，即自动管理属性存储的能力。）

3. 实例化分析树

属性文法描述与文法上合法语句的分析树相关的计算。这一范式本质上取决于分析树的可用性。评估程序可以模拟分析树，但是它必须当作分析树存在那样行动。虽然分析树对分析的讨论很有用，但很
186 少有编译器会真正构建分析树。

有些编译器使用抽象语法树（AST）来表示被编译程序。AST具有分析树的本质结构，但是却消除很多表示文法中非终结符的内部结点（参见5.3.1节）。如果编译器构建AST，那么它可以使用依赖于AST文法的属性文法。然而，如果这个编译器在其他方面不使用AST，那么与构建并维持AST相关的程序设计努力和编译时代价将超过使用属性文法形式的益处。

4. 定位答案

上下文相关分析的属性文法框架的最后一个问题更加微妙。属性评估结果是一棵属性树。分析的结果以属性值的形式散布于树中。为了在以后的分析中使用这些结果，编译器必须遍历树来定位需要的信息。

编译器可以使用细心构建的遍历来定位特定结点；这仍然要求从分析树的根向下走到适当的位置，每一次处理都需要这样做。这使得代码既慢又很难书写，编译器必须执行每一个这样的遍历，而且编译器设计者必须逐个构造它们。另外一个方法是把重要的答案拷贝到在树上容易被发现一个点上，通常这一点是根。这将导致更多拷贝规则，使这一问题更加复杂。

5. 函数范式的崩溃

处理所有这些问题的一个方法就是添加属性的中心存储室。在这样的情况下，属性规则可以把信息直接记录到一个全局表上，在这个表上可以读取信息其他规则。这一混合方法可以消除非局部信息所引发的很多问题。因为这一全局表可以被任意属性规则处理，它具有为所有已得到的信息提供局部存取的效应。

添加中心存储室从另一方面使事情复杂化。如果两个规则通过某种机制而不通过属性规则进行沟通的话，那么二者之间隐含的依赖关系就从属性相关图中消失了。这消失的相关性应该限制评估程序来保证以正确的顺序处理这两个规则；没有这样的限制的话，评估程序就可能构建一个虽然对文法是正确的，
187 但却不是我们预期的行为顺序。例如，如果使用一个表在声明语法与可执行表达式之间传递信息，那么

评估程序可能在使用已声明变量的某些或全部表达式之后处理该声明。如果文法使用拷贝规则传递相同的信息，那么这些规则限制评估程序，使其遵守由这些拷贝规则规定的相关性。

6. 总结

属性文法并非是每一个上下文相关分析都适合的抽象。属性文法技术的倡导者们认为，属性文法的问题是可以解决的，而且高级、非过程化的描述的优越性超过了它们的问题。然而，由于诸多普遍的原因，属性文法方法从没有得到广泛的流行。诸如执行非局部计算的困难以及为发现简单问题的答案而必须遍历分析树等重大问题阻碍了对这些想法的采用。诸如短生存期属性的空间管理、评估程序的效率、缺乏高质量、廉价工具等诸多小问题也使这些工具和技术缺乏吸引力。

尽管如此，属性文法范式的简洁性还是具有吸引力的。如果能够把属性流传输方向控制到单一方向，或者是合成的或者是继承的，那么结果属性文法就很简单并且评估程序也很有效。例如，这一范式对计算器或解释器中表达式的评估就很有效。这时，值的传递方向是沿着分析树从叶子到根的，所以规则和评估程序都是直截了当的。同样地，只涉入局部信息的应用通常也有好的属性文法解决方案。

4.4 特定语法制导翻译

属性文法的基于规则的评估程序提出一个强有力的思想，这一思想用于很多编译器的上下文相关分析的特定技术的基础。在基于规则的评估程序中，编译器设计者描述一系列与文法中的产生式相关的动作。通过观察可知，上下文相关分析所需的动作是围绕着文法的结构组织起来的，这导致一个强有力的特定方法，这一方法把这种分析与分析上下文无关文法的过程结合起来。我们称这种方法为**特定语法制导翻译**（ad hoc syntax-directed translation）。 188

在这一方案中，编译器设计者提供在语法分析时执行的代码片段。每一个片段或*动作*（action）直接关联到文法中的一个产生式。每一次语法分析器识别出它处于文法中的某个特定的位置时，它便调用相应的动作来执行它的任务。为了在自顶向下递归下降分析器中实现这一工作，编译器设计者只是把适当的代码加入到分析程序中。编译器设计者完全控制何时执行动作。在自底向上移入归约分析中，每当语法分析器执行一个归约动作时执行动作。这受到更多的限制，但是仍然是可行的。

为了使这一过程更具体，考虑在特定语法制导翻译结构中重新公式化带符号二进制数的例子。图4-10给出这样的一个框架。每一个文法符号有一个与其相关的值，在代码片段中记作*val*。每一个规则的代码片段定义与这个规则左部符号相关的值。规则1简单地把*Sign*的值与*List*的值相乘。规则2和规则3为*Sign*设置适当的值，正如规则6和规则7为*Bit*的每一个实例设置值一样。规则4把*Bit*的值拷贝到*List*上。规则5做实际的工作，这一规则把前导bit的累积值（List.val）乘以2，然后再加上下一位。

	产 生 式	代码片段
1	$Number \rightarrow Sign\ List$	$Number.val \leftarrow Sign.val \times List.val$
2	$Sign \rightarrow +$	$Sign.val \leftarrow 1$
3	$Sign \rightarrow -$	$Sign.val \leftarrow -1$
4	$List \rightarrow Bit$	$List.val \leftarrow Bit.val$
5	$List_0 \rightarrow List_1\ Bit$	$List_0.val \leftarrow 2 \times List_1.val + Bit.val$
6	$Bit \rightarrow 0$	$Bit.val \leftarrow 0$
7	$Bit \rightarrow 1$	$Bit.val \leftarrow 1$

图4-10 带符号二进制数的特定语法制导翻译 189

至此，这看似与属性文法非常相似。然而，这一过程存在两个重要的简化。只存在一个方向上的值传输：从叶子到根。每一个文法符号只允许有一个值。即使如此，图4-10中的方案正确计算出带符号二进制数的值。它把这个值留在树的根部，就如同带符号二进制数的属性文法一样。

这两个简化使该评估方法得以与自底向上分析器，例如第3章所描述的LR分析器，很好地配合。

因为每一个代码片段都与特定产生式的右部相关，所以分析器可以在每一次用这个产生式进行归约时调用它的动作。这要求对如图3-13中所示的LR(1) 分析器驱动器中的归约动作做微小的修改。

```
else if Action[s,word] = "reduce A→β" then
  invoke the appropriate reduce action
  pop 2 × |β| symbols
  s ← top of stack
  push A
  push Goto[s,A]
```

分析器生成器可以将语法制导动作聚集到一起，把这些动作嵌入到一个在一组归约产生式间切换的选择语句中，并在它从栈中弹出产生式右部之前执行这个选择语句。

如图4-10所示的翻译方案比用于解释属性文法的方案简单。当然，我们可以写出运用相同策略的属性文法。它只使用合成属性。它将比如图4-4所示的文法有更少的属性规则和属性。我们选择更复杂的属性方案来展示合成和继承属性的使用。

4.4.1 实现特定语法制导翻译

为了使特定语法制导翻译奏效，语法分析器必须包含这样的机制：把一个动作中定义的值传递给另外一个动作使用，提供方便、一致的命名，并允许在分析过程中的其他点处执行的动作。本节描述自底向上移入归约分析器中处理这些问题的机制。我们采用YACC系统所使用的表记法，这一系统是随Unix操作系统发布的早期、流行的LALR(1) 分析器生成器。许多后期系统采用YACC表记法。

1. 动作之间的交流

为了在动作之间传递值，语法分析器必须有配置保存各个动作所产生的值的空间的方法。这一机制必须使动作能够找到它所使用的值。属性文法把这些值（属性）与分析树中的结点联系起来；将属性存储与树结点存储联系起来，使我们得以系统地找到属性值。在特定语法制导翻译中，语法分析器可以不构建这样的分析树。相反，语法分析器能够把值整合存储到它自身的机制中以跟踪语法分析的状态，这一机制就是它的内部栈。

190

回顾一下，框架LR(1) 分析器为每一个文法符号在栈上存储两个值：符号和相应的状态。当它识别句柄，例如与规则5的右部匹配的*List Bit*序列时，栈上的第一对值表示*Bit*。其下面的一对值表示*List*。我们可以用三元组<*value*，*symbol*，*state*>取代<*symbol*，*state*>。从而为每一个文法符号提供单一值的属性，这正是简化方案所需要的。为了管理这个栈，分析器压入和弹出更多的值。对于产生式$A\rightarrow\beta$ 的归约，它从栈中弹出$3\times|\beta|$ 个位置，而不是$2\times|\beta|$ 个位置。它压入值连同符号和状态。

这一方法把值存放于相对于栈顶容易计算的位置。每一次归约将它的结果作为表示左部的三元组的一部分压入到栈中。动作从栈中的相对位置读取右部的值；右部第i个符号的值在从栈顶数第i个三元组中。这些值的取值范围是固定的；在实践中，这一限制意味着我们应该使用指向结构的指针来传递更复杂的值。

为了节省存储，语法分析器可以在栈中省略实际的文法符号。语法分析所需的必要信息被编码于状态中。这可以缩小栈，而且通过消除压入和弹出这些符号的操作来加速分析。另一方面，文法符号有助于语法分析器中的错误报告和调试。这一权衡通常是在不修改工具所生成的语法分析器的条件下做出的，

因为在每一次重新生成分析器时，必须再一次进行这样的修改。

2. 命名值

为了简化基于栈的值的使用，编译器设计者需要为它们命名的表记法。YACC提出一种解决这一问题的简洁表记法。符号*$$*表示当前产生式的结果位置。因此，赋值*$$*=0；将把整数值0作为当前归约的结果压入到栈中。这一赋值可以实现图4-10中规则6的动作。对于规则右部，符号*$1*、*$2*、…、*$n*分别表示右部的第一个、第二个到第*n*个符号的位置。 191

使用这一表记法重写图4-10中的例子产生下面的描述：

	产 生 式	代码片段
1	*Number* → *Sign List*	*$$* ← *$1* × *$2*
2	*Sign* → +	*$$* ← *1*
3	*Sign* → -	*$$* ← −*1*
4	*List* → *Bit*	*$$* ← *$1*
5	$List_0 \rightarrow List_1\ Bit$	*$$* ← *2* × *$1* + *$2*
6	*Bit* → 0	*$$* ← *0*
7	*Bit* → 1	*$$* ← *1*

注意代码片段是多么紧凑。

这一方案具有高效的实现；符号直接翻译成距栈顶的偏移量。表记法*$* 1表示位于从栈顶往下第$3 \times |\beta|$个位置，而*$i*表示从栈顶往下第$3 \times (|\beta| - i + 1)$个位置。因此，这一位置表记法允许动作片段直接读写栈中的位置。（如果我们优化语法分析器，使其不压入实际的文法符号，那么乘数变成2而不是3。）

3. 语法分析其他点处的动作

编译器设计者也可能需要在产生式中间或在移入动作上执行一个动作。为了实现这一点，编译器设计者可以对文法进行转换，使其在需要动作的每一点执行一个归约。为了在产生式中间进行归约，可以将在应该执行动作之处把产生式分成两个部分。加入一个高层次的产生式，将第一个部分和第二个部分依次连结起来。当第一个部分进行归约时，语法分析器调用这一动作。为了迫使动作在移入时进行，编译器设计者或者把动作移到扫描器进行，或者加入一个产生式来支持这一动作。例如，为了在移入终结符*Bit*时执行一个动作，编译器设计者可以加入产生式： 192

$$ShiftedBit \rightarrow Bit$$

并用*ShiftedBit*取代*Bit*的每一次出现。从而为每一个终结符增加一个额外的归约。因此，额外代价直接与程序中终结符的数量成正比。

4.4.2 例子

为了理解特定语法制导翻译的工作方式，考虑使用这种方法重写执行时间估测器。属性文法解决方案的主要缺点在于在树的结点间拷贝信息的规则的增殖。这在描述中生成很多额外的规则，并在很多结点处复制属性值。

为了使用特定语法制导翻译方案来描述这些问题，编译器设计者通常为变量的信息引入一个中心存储室，如前所述。这消除在树的结点间拷贝值的必要性。它还简化继承值的处理。因为语法分析器决定评估顺序，我们无需担心破坏属性间的相关性。

大多数编译器构建和使用这样的一个存储室，称为*符号表*（symbol table）。符号表把名字映射到各种注释，诸如类型、运行时表示的大小等，以及生成运行时地址所需要的信息。符号表还存储若干类型

相关的域，诸如函数的类型署名、数组的维数以及边界等。5.7节和B.4将更深入讨论符号表的设计。

1. 再论装入跟踪

再一次考虑作为估测执行代价的一部分而提出的跟踪装入（load）操作的问题。属性文法中这一问题的大部分复杂性是由在分析树的结点间传送信息的需要引起的。在使用符号表的特定语法制导翻译方案中，这一问题变得容易处理。编译器设计者可以在这一表中设置一个域来保存表示标识符是否已被装入的布尔值。这个域最初被设置为*false*。关键代码是关于产生式*Factor*→ident的代码。如果ident的符号表条目表明还没有完成装入，那么代价被更新，而且这个域被设置为true。

图4-11给出这一情况，它还包括所有其他动作。因为这些动作可以包含任意代码，所以编译器可以
193 把*cost*累积在一个变量中，而不是在分析树的每一个结点处创建一个*cost*属性。

产	生	式	语法制导动作
$Block_0$	→	$Block_1$ *Assign*	
	|	*Assign*	
Assign	→	*Identifier* = *Expr* ;	{ *cost* = *cost* + *Cost*(store) }
$Expr_0$	→	$Expr_1$ + *Term*	{ *cost* = *cost* + *Cost*(add) }
	|	$Expr_1$ - *Term*	{ *cost* = *cost* + *Cost*(sub) }
	|	*Term*	
$Term_0$	→	$Term_1$ × *Factor*	{ *cost* = *cost* + *Cost*(mult) }
	|	$Term_1$ ÷ *Factor*	{ *cost* = *cost* + *Cost*(div) }
	|	*Factor*	
Factor	→	(*Expr*)	
	|	num	{ *cost* = *cost* + *Cost*(loadI) }
	|	ident	{ *if* ident's *symbol table field indicates that it has not been loaded then* *cost* = *cost* + *Cost*(load); *set the field to true* }

图4-11 使用特定语法制导翻译跟踪装入

对于最简单的执行模型，这一方案比属性规则需要更少的动作，尽管属性规则可以为更复杂模型提供精确性。

注意，有几个产生式没有动作。除iden引发的归约上的动作外，其余的动作是简单的。由跟踪装入引发的所有复杂性都落入这一动作中；这与属性文法方案中的动作不同，在那里*Before*和*After*集合间的传送任务支配着描述。特定语法制导翻译更清楚、更简单，这部分是因为这一问题非常适合于由移入归约分析器中的归约动作所指定的评估顺序。当然，编译器设计者必须实现这一符号表，或从某一数据结构实现的程序库中导入符号表。

当然，其中的某些策略也可以运用于属性文法框架。然而，它违反属性文法的功能属性。它迫使这
194 一工作的某些重要部分脱离属性文法框架而进入特定的设置。

图4-11的方案忽视一个关键问题：初始化*cost*。正如所看到的那样，这一文法不包含可以把*cost*适当初始化为0的产生式。如前所述，这一问题的解决方案是修改文法，使其生成一个初始化的地方。一个初始产生式，如*Start*→*CostInit Block*，还有*CostInit*→ε就可以完成这一工作。这一框架在从ε到*CostInit*的归约上执行赋值*cost*←0。

2. 重探表达式的类型推断

推断表达式类型的问题适合于属性文法框架，只要我们假设叶结点总是有类型信息即可。如图4-6所示这一问题的解答的简洁性来自于两个主要事实。第一，因为表达式类型是递归地定义于表达树上的，信息的自然流向自底向上从叶向根传输。这使其倾向于*S*-属性文法的解决方案。第二，表达式类型是使用源语言语法定义的。这也同样适合于属性文法框架，因为这隐式地要求分析树的存在。所有类型信息都可以与文法符号的实例相关联，而文法符号精确地对应于分析树上的结点。

如图4-12所示，我们可以在特定语法制导翻译的框架下重新公式化这一问题。结果框架看起来类似于同一目的的如图4-6所示的结果。这一特定框架并没有为这一问题提供真正的益处。 195

		产 生 式	语法制导动作
Expr	→	*Expr* + *Term*	{ $$ ← *F*+(*$1*,*$3*) }
	|	*Expr* − *Term*	{ $$ ← *F*−(*$1*,*$3*) }
	|	*Term*	{ $$ ← *$1* }
Term	→	*Term* × *Factor*	{ $$ ← *F*×(*$1*,*$3*) }
	|	*Term* ÷ *Factor*	{ $$ ← *F*÷(*$1*,*$3*) }
	|	*Factor*	{ $$ ← *$1* }
Factor	→	(*Expr*)	{ $$ ← *$2* }
	|	num	{ $$ ← *type of the* num }
	|	ident	{ $$ ← *type of the* ident }

图4-12 推断表达式类型的特定语法制导翻译框架

3. 构建抽象语法树

编译器前端必须为程序构建一个中间表示以供编译器的中间部分及后端使用。抽象语法树是树结构IR的一个通用形式（参见5.3.1节）。构建AST的任务非常适合于特定语法制导翻译框架。

假设编译器有一系列命名为$MakeNode_i$的过程，其中$0 \leq i \leq 3$。过程取惟一识别文法符号的常量为它的第一个参数，新结点将表示这一文法符号。其余i个参数是i棵子树的根结点。因此，$MakeNode_0$(*number*) 构造一个叶结点并标示它表示一个num。同样地， 196

$$MakeNode_2(Plus, MakeNode_0(number), MakeNode_0(number))$$

构建一个以*plus*为根并带有两个子结点的AST，其中每一个子结点是num的叶结点[⊖]。

为了构建抽象语法树，特定语法制导翻译框架遵循下面两个一般原则：

1）对于一个操作符，它创建一个以每个操作数为子结点的结点。因此2+3为+创建一个二叉结点，且结点2和3是它的子结点。

2）对于一个无用产生式，如*Term*→*Factor*，它复用*Factor*的动作结果为它自己的结果。

按这一方式，它避免构建表示语法变量的树结点，如*Factor*、*Term*和*Expr*等。图4-13给出具体表现这些思想的语法制导翻译方案。

4. 生成表达式的ILOC

作为处理表达式的最后一个例子，考虑生成ILOC而不是AST的特定框架。我们将做几个简化假设。

⊖ 过程$MakeNode_i$可以用任意合适的方式实现树。例如，它们可以把这一结构映射到一棵二叉树上。参见B.3.1节中的讨论。

这一例子只局限于整数；处理其他类型只会增加复杂性而不能有助于我们的理解。这一例子还假设所有值都可以存放于寄存器中：这些值可以适合寄存器的大小，而且ILOC实现提供的寄存器多于计算使用的寄存器。

产生式			语法制导动作
Expr	→	*Expr* + *Term*	{ *$$* ← *MakeNode*$_2$ (*plus*, *$1*, *$3*); *$$.type* ← *F* + (*$1.type*,*$3.type*) }
	|	*Expr* − *Term*	{ *$$* ← *MakeNode*$_2$(*minus*, *$1*, *$3*); *$$.type* ← *F* − (*$1.type*,*$3.type*) }
	|	*Term*	{ *$$* ← *$1* }
Term	→	*Term* × *Factor*	{ *$$* ← *MakeNode*$_2$(*times*, *$1*, *$3*); *$$.type* ← *F* × (*$1.type*,*$3.type*) }
	|	*Term* ÷ *Factor*	{ *$$* ← *MakeNode*$_2$(*divide*, *$1*, *$3*); *$$.type* ← *F* ÷ (*$1.type*,*$3.type*) }
	|	*Factor*	{ *$$* ← *$1* }
Factor	→	(*Expr*)	{ *$$* ← *$2* }
	|	num	{ *$$* ← *MakeNode*$_0$(*number*); *$$.text* ← *scanned text*; *$$.type* ← *type of the number* }
	|	ident	{ *$$* ← *MakeNode*$_0$(*identifier*); *$$.text* ← *scanned text*; *$$.type* ← *type of the identifier* }

图4-13 构建抽象语法树并推断表达式类型

代码生成要求编译器跟踪很多细节。为了把这些簿记细节抽象掉（并把一些更深层的问题推迟到以后的章节），这一例子的框架使用四个支援过程。

1）*Address*取一个变量名作为它的参数并返回一个寄存器号码。它保证该寄存器包含这一变量的地址。如果必要的话，它生成计算这一变量地址的代码。

2）*Emit*处理创建各个ILOC操作的具体表示的细节。它也许按特定格式把它们打印成文件。或者，它也可能为以后使用构建一个内部表示。

197 3）*NextRegister*返回一个新寄存器号码。一个简单的实现可能是递增一个全局计数器。

4）*Value*取一个数作为它的参数并返回一个寄存器号码。它保证该寄存器包含作为参数的数。如果有必要，它生成将这个数移到寄存器中的代码。

图4-14给出这一问题的语法制导框架。这些动作通过在分析栈中传递寄存器名来通信。在必要时这些动作把这些名字传送到*Emit*来创建实现输入表达式的操作。

5. 处理声明

当然，编译器设计者可以使用语法制导的动作填充属于符号表的大部分信息。例如，图4-15所示的文法片段描述C语言的变量声明的一部分语法。（它忽略类型定义（typedef）、结构（struct）、联合（union）、类型修饰词const、restrict和volatile以及初始化语法的细节。它还留下若干没有详细描述的非终结符。）考虑为每个变量声明构建符号表条目所需的动作。每个*Declaration*开始于一个或多个描
198 述变量类型和存储类型的修饰词。这些修饰词后面跟随由一个或多个变量名组成的序列；每一个变量名可以包含关于间接（一个或多个*的出现）、关于数组维数以及关于变量初始值等的描述。

产生式			语法制导动作
Expr	→	*Expr* + *Term*	{ *$$* ← *NextRegister*; *Emit*(*add*,*$1*,*$3*,*$$*)
	|	*Expr* − *Term*	{ *$$* ← *NextRegister*; *Emit*(*sub*,*$1*,*$3*,*$$*)
	|	*Term*	{ *$$* ← *$1* }
Term	→	*Term* × *Factor*	{ *$$* ← *NextRegister*; *Emit*(*mult*,*$1*,*$3*,*$$*)
	|	*Term* ÷ *Factor*	{ *$$* ← *NextRegister*; *Emit*(*div*,*$1*,*$3*,*$$*)
	|	*Factor*	{ *$$* ← *$1* }
Factor	→	(*Expr*)	{ *$$* ← *$2* }
	|	num	{ *$$* ← *Value*(*scanned text*); }
	|	ident	{ *$$* ← *Address*(*scanned text*); }

图4-14 为表达式发行ILOC

DeclarationList	→	*DeclarationList Declaration*
	|	*Declaration*
Declaration	→	*SpecifierList InitDeclaratorList* ;
SpecifierList	→	*Specifier SpecifierList*
	|	*Specifier*
Specifier	→	*StorageClass*
	|	*TypeSpecifier*
StorageClass	→	`auto`
	|	`static`
	|	`extern`
	|	`register`
TypeSpecifier	→	`void`
	|	`char`
	|	`short`
	|	`int`
	|	`long`
	|	`signed`
	|	`unsigned`
	|	`float`
	|	`double`
InitDeclaratorList	→	*InitDeclaratorList* , *InitDeclarator*
	|	*InitDeclarator*
InitDeclarator	→	*Declarator* = *Initializer*
	|	*Declarator*
Declarator	→	*Pointer DirectDeclarator*
	|	*DirectDeclarator*
Pointer	→	*
	|	* *Pointer*
DirectDeclarator	→	`ident`
	|	(*Declarator*)
	|	*DirectDeclarator* ( )
	|	*DirectDeclarator* (*ParameterTypeList*)
	|	*DirectDeclarator* (*IdentifierList*)
	|	*DirectDeclarator* [ ]
	|	*DirectDeclarator* [*ConstantExpr*]

图4-15 C语言声明语法的子集

例如，*StroageClass*产生式允许程序员描述有关变量值生存期的信息；auto变量的生存期与声明它的块的生存期一致，而static变量的生存期是程序的整个执行期间。register说明符建议编译器该值应该保存在能够快速存取的位置，即硬件寄存器上。extern说明符告知编译器不同编译单元中相同名字的声明需要作为一个对象来链接。

编译器必须保证每一个被声明的名字至多有一个存储类型属性。文法将说明符放置在由一个或多个名字的序列之前。编译器在处理说明符时可以把它们记录下来，并在后来遇到这些名字时对这些名字运用该说明符。文法容许任意多个*StroageClass*和*TypeSpecifier*关键字；语言的标准限制实际关键字可以结合的方式⊖。例如，每个声明只允许有一个*StorageClass*。编译器必须通过上下文相关检测来支持这一限制。类似的限制适用于*TypeSpecifier*。例如，short和int在一起是合法的，而和float在一起是不合法的。

为了处理声明，编译器必须从限定符收集属性，增加任意的间接、维数或初始化属性，并将变量加入表中。编译器设计者也许设置一个性质结构，这一性质结构的域与符号表条目的域相匹配。在一个*Declaration*的最后，它可以初始化这一结构中每个域的值。当它归约声明语法中的各产生式时，它可以相应调整这一结构中的值。

- 在auto到*StorageClass*的归约上，编译器检测存储类型域是否还没有被设置，然后，把这个域设置为auto。对static、extern和register的类似动作完成对名字的那些性质的处理。
- 类型说明符产生式将设置这一结构中的其他域。它们必须包含确保只出现合法组合的检测。

200

关于上下文相关文法

通过前几章所给出的思想的发展，似乎很自然地去考虑使用上下文相关语言来执行上下文相关检测，例如类型推断。总之，我们使用正则语言执行词法分析，使用上下文无关语言执行语法分析。一个自然的发展也许提出对上下文相关语言和它们的文法的研究。上下文相关文法表示的语言系列比上下文无关文法所能表示的语言系列更大。

然而，由于两个不同原因，上下文相关文法不是正确的答案。首先，分析上下文相关文法的问题是P空间完全的。因此，使用这样的技术的编译器会非常慢。其次，很多重要问题即使不是不可能，也是非常难以使用上下文相关文法来描绘。例如，考虑使用前声明的问题。将这一规则写成上下文相关文法将需要这一文法描绘声明变量的每一种不同的组合。如果名字空间足够小（例如，Dartmouth BASIC将名字限制为单一字母或一个字母后面跟着可选的一个数字），这也许是可行的；使用巨大名字空间的现代语言中，名字集合大得无法使用上下文相关文法描绘。

- 从ident到DirectDeclarator的归约将引发一个为该名字创建符号表新条目并将性质结构的当前设置拷贝到这一条目中的动作。
- 通过产生式

$$InitDeclaratorList \rightarrow InitDeclaratorList\ ,\ InitDeclarator$$

的归约可以重置与特定名字相关的性质域，包括由*Pointer*、*Initializer*和*DirectDeclarator*产生式所设置的性质域。

⊖ 这种类型的限制可以在文法上实现。标准的制定者不选择以这种方式处理它。增加这种限制将使已经很大的文法更加复杂。因为编译器必须在*TypeSpecifier*的组合上跟踪类似的限制，所以在语法之外进行这样的检测的负荷较小。

通过在声明语法的这些产生式间协调一系列动作，编译器设计者可以确保在处理每个名字时性质结构都包含适当的设置。

当语法分析器完成构建*DeclararionList*的工作时，它已为在当前作用域声明的每一个变量构建符号表的一个条目。此时，语法分析器也许需要执行某些常规杂务，诸如给已声明变量指派存储位置等。这一工作可以由归约*DeclararionList*的产生式的动作来完成。如果有必要的话，可以分解这个产生式，创建执行这一动作的合适位置。 201

4.5 高级话题

本章介绍了类型理论的基本概念，并把它们作为属性文法框架和特定语法制导翻译的启示性例子。对类型理论的更深层的处理和它的运用需要一整本书来讨论。

本节第一小节展示一些影响编译器执行类型推断和类型检测方式的语言设计问题。第二小节探讨实践中出现的若干问题：在构建中间表示的过程中重新安排计算。

4.5.1 类型推断中的较难问题

强类型化静态检测语言可以通过寻找大部分错误程序而帮助程序员生成合法的程序。同样地，揭示错误的特征可以增进编译器通过取消运行时检测来生成高效代码的能力，同时揭示出编译器在何处可以对某些结构成份特化特定情况、消除运行时不可能出现的情况。这些事实部分导致在现代程序设计语言中类型理论的作用日趋重要。

然而，我们的例子做了并非在所有程序设计语言中都成立的假设。例如，我们假设变量和过程都被声明，程序员为每一个名字写下紧凑、形成一个整体的描述。改变这些假设可能从根本上改变类型检测问题的性质以及编译器在实现语言时可用的策略的性质。

一些程序设计语言或者省略声明，或者把声明当作可选的信息。Scheme程序缺少变量声明。Smalltalk程序声明类，但是只有当程序实例化一个对象时，才决定这个对象的类。支持独立编译的语言，也就是可以独立地编译过程并在链接时把它们结合起来形成程序的语言，也许对独立编译的过程不要求声明。 202

在没有声明的情况下，类型检测更加困难，因为编译器必须取决于上下文线索来决定每个名字的合适类型。例如，如果i被用做某个数组a的下标，这也许迫使i具有数字类型。这一语言也许只允许整数下标；另外，它也可能允许可以转换到整数的任意类型。

通常可以用语言定义来描述类型规则。这些规则的特定细节决定它为每一个变量推断类型的难易程度。因而，这对编译器用于实现语言的策略有直接的影响。

1. 类型相容运用及固定函数类型

考虑一个需要变量和函数的相容运用的无声明语言。这时，编译器可以为每个名字指定一个一般类型，然后通过名字在上下文的使用来缩小类型。例如，`x←y*3.14159`这样的语句揭示x和y是数，x必须具有允许存放小数的类型。如果y还出现于期望整数的上下文中，例如出现于数组引用`a(y)`中，那么编译器必须在非整数（对于`y*3.14159`）和整数（对于`a(y)`）间做出选择。无论选择的结果如何，都需要对其中的一个运用进行类型转换。

如果函数具有已知且固定的返回类型，即一个总是返回相同类型的函数，那么编译器可以在使用类型构成的格上运用迭代不动点算法来对类型推断问题求解。

2. 类型相容运用及不知函数类型

如果一个函数的类型因函数的参数而发生变化，那么类型推断问题就变得更加复杂。例如，在

Scheme中就有这样的问题。Scheme的库过程map取一个函数和一个列表为参数。它返回将这一函数参数运用于列表的每一个元素所得的结果。也就是说，如果参数函数取类型$\alpha \rightarrow \beta$，那么map的类型为α的列表到β的列表[⊖]。我们将它的类型署名写作：

203

$$\text{map: } (\alpha \rightarrow \beta) \times \textit{list of } \alpha \rightarrow \textit{list of } \beta$$

因为map的返回类型依赖于它的参数类型（这称为参数多态性），推断规则必须包括类型空间上的等式。（等式包含已知、固定的返回类型和在类型空间上取返回值的函数。）由于这一情况，简单的迭代不动点类型推断是不够的。

检测这些更复杂体系的经典方法依赖于单一化，尽管精明的类型体系设计和类型表示可以使用更简单或更有效的技术。

3. 类型中的动态变化

如果变量的类型在执行期间可以改变，也许要求其他策略去发现类型在何处发生变化，并推断正确的类型。理论上，编译器可以重新命名变量使得每一个定义对应惟一的名字。然后，编译器可以基于定义每一个名字的操作所提供的上下文来为这些名字推断类型。

为了成功地推断类型，这样的体系需要处理代码中的一些点，不同控制流路径的汇集导致不同的定义必须在这些点处合并。就如静态单一赋值形式中的ϕ-函数那样（参见5.5节和9.3节）。如果这一语言包含参数多态化，那么类型推断机制也必须处理它。

实现带有动态改变类型的语言的经典方法是回到解释的方案。Lisp、Scheme、Smalltalk和APL都有类似问题。这些语言的标准实现实践包括解释操作符，使用数据类型给数据指定标签，并在运行时检查类型错误。

在APL中，程序员很容易写出这样一个程序，其中a × b在第一次执行时是整数乘法，而在下一次则是浮点数多维数数组的乘法。这导致对于消去的检测和对于动作的检测的实质性研究。最好的APL体系回避不成熟的解释器所需的大部分检测。

4.5.2 更改结合性

正如我们在3.6.4节中所看到的那样，在数字计算中结合性可以造成很大的不同。同样地，它可以改变构建数据结构的方式。我们可以使用语法制导动作构建反映与文法自然生成的结合性不同的结合性的表示。

204

一般地，左递归文法自然生成左结合性，而右递归文法自然生成右结合性。为了看到这一点，考虑图4-16上部的左递归列表文法和右递归列表文法，其中还加入了建立列表的语法制导动作。与每一个产生式相关联的这些动作构建列表的表示。假设$L(x, y)$是列表构造器；它可被实现成$MakeNode_2(cons, x, y)$。图4-16的下部给出将两个翻译方案运用到由五个elt组成的输入的结果。

在很多方面，这两棵树是等价的。两棵树的中序遍历以相同的顺序访问叶结点。如果我们加上括号来反映树的结构，那么左递归树是$((((elt_1, elt_2), elt_3), elt_4), elt_5)$，而右递归树是$(elt_1, (elt_2, (elt_3, (elt_4, elt_5))))$。左递归所产生的顺序对应于代数操作符的经典从左到右顺序。右递归所产生的顺序对应于Lisp和Scheme中的列表的表记法。

有时，对递归和结合性使用不同的方向很方便。为了从左递归文法构建右递归树，我们可以使用把相继的元素加到列表末端的构造器。这一思想的直接实现必将对每一次归约遍历列表，对于长度为n的列表，构造器本身的时间复杂度就是$O(n^2)$。为了避免这样的额外开销，编译器可以生成含有指向这一列

⊖ map也可以处理含多个参数的函数。为了做到这一点，它取多个参数列表并把它们依次作为各参数的列表。

表中的第一个结点和最后一个结点的指针的列表头结点。这对每个列表导入一个额外的结点。如果系统构造很多短列表，这一额外开销可能成为问题。

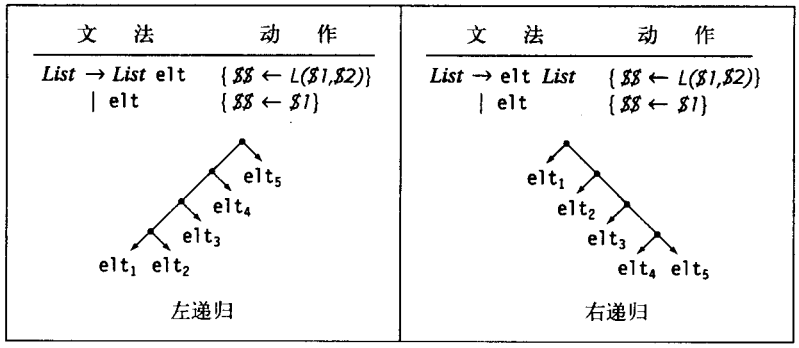


图4-16 递归与结合性的比较

一个特别吸引人的解决方案是在构建期间使用列表头结点，并在列表构建完成以后丢弃这一头结点。使用ε产生式重写文法就会使这一方案清楚地出现在我们的面前。 205

文 法			动 作
List	→	ε	{ $$ ← *MakeListHeader()* }
	|	*List* elt	{ $$ ← *AddToEnd($1,$2)* }
Quux	→	*List*	{ $$ ← *RemoveListHeader($1)* }

使用ε产生式的归约创建临时列表头结点；使用移入归约分析器，这一归约首先出现。产生式*List*→*List* elt调用构造器，它取决于临时头结点的存在。当在任意其他产生式的右部的*List*被归约时，相应的动作调用丢弃临时头结点并返回列表第一个元素的函数。

这一方法使得语法分析器能够在空间和时间上以小常量额外开销的代价反转结合性。这一方法对每个列表要求一个额外的由ε产生式引发的归约。修改后的文法接受空表，而原来的文法不接受空表。为了弥补这一差异，*RemoveListHeader*可以显式地检测空表并报告错误。

4.6 概括和展望

在第2章和第3章中,我们已看到编译器前端中的很多工作可以自动进行。正则表达式对词法分析很有效。上下文无关文法对语法分析很有效。在本章中，我们考察了两种执行上下文相关分析的方法：属性文法形式和一个特定方法。对于上下文相关分析，与扫描和语法分析不同，形式方法还不能取代这一特定方法。

使用属性文法的形式方法为书写产生适当高效可执行的高级描述带来了希望。尽管属性文法不是上下文相关分析中每一个问题的解决方案，它们已在从定理证明器到程序分析等若干领域得到了应用。如果问题中的属性流主要是局部的话，属性文法可以很好地工作。用属性文法处理使用一种属性可以完整刻画的问题，无论是合成属性还是继承属性，通常都可以产生清晰、直观的解决方案。(当我们需要使用拷贝规则沿树指引属性的流向时，)我们可能就到了走出属性文法的函数范式，来到引入中心存储室的时候了。 206

特定技术、语法制导翻译把任意的代码片段整合在语法分析器中，并让语法分析器来安排动作序列及在它们之间传值。由于这一方法有很大的灵活性，并且很多分析器生成器系统都包含该方法，这一方

法已得到广泛的拥护。这一特定方法回避非局部属性流以及管理属性存储的需要所引发的各种实际问题。值沿着它的状态的语法分析器内部表示的方向传输（就自底向上分析器的合成值和自顶向下分析器的继承值而言）。这些方案使用全局数据结构在其他方向上传送信息并处理非局部属性流。

在实践中，编译器设计者通常尝试同时解决若干问题，诸如构建中间表示，推断类型并指定存储位置。这倾向于创建许多两个方向上的属性流，迫使实现者采纳使用诸如符号表等的中心存储室存储事实的特定解决方案。在一遍中解决多个问题的理由通常是编译时的效率。然而，在不同的遍中解决问题通常可以产生更容易理解、更容易实现且更容易维护的解决方案。

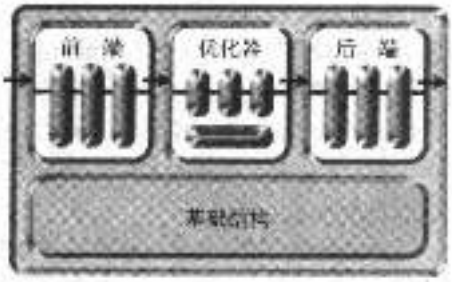
本章作为编译器必须执行的上下文相关分析的一个例子介绍了类型系统背后的思想。类型理论和类型系统设计的研究本身是具有重要学术意义的活动，同时具有深入的学术探讨。本章只粗略地概述了类型推断和类型检测等表面问题，更深层的研究问题则超出了本章的范畴。在实践中，编译器设计者需要彻底研究源程序的类型系统，并详细地工程化类型推断和类型检测的实现。本章的内容只是一个开始，实际的实现需要更深入的研究。

本章注释

从最初的FORTRAN编译器开始，类型系统就已成为程序设计语言的一个不可或缺的部分。虽然最早的类型系统反映机器的资源，但不久就在诸如Algol 86和Simula 67中出现了对于类型系统具有重要意义的抽象。类型系统理论几十年来得到了积极的研究，产生一系列包含重要原则的语言。这些包括Russell [43]（参数多态性），CLU [239]（抽象数据类型），Smalltak [157]（继承子类型）和ML [258]（彻底而完整地将类型作为第一类对象）。Cardelli给出了类型系统的精彩纵览[64]。APL组织发表了一系
207 列研究消除运行时检测技术的经典文章[1，32，257，331]。

与计算机科学中的很多思想一样，属性文法是由Knuth [218，219]首先提出来的。关于属性文法的文献集中研究了评估器[193，327]、循环检测[327]以及属性文法的应用[152，287]。属性文法充当了若干成功系统的基础，包括Intel 80286的Pascal编译器[135，136]、Cornell程序合成器[286]以及合成器生成器[188，288]。

特定语法制导翻译业已成为实际的语法分析器开发的一部分。Irons通过将语法分析器的动作从语法描述分离出来描述了语法制导翻译背后的基本思想[191]。毫无疑问，同样的基本思想在语法制导分析器出现之前就被用于手工编码优先顺序分析器。我们所描述的编写语法制导动作的风格是由Johnson在
208 YACC中引入的[195]。同样的表记法也已进军当前的系统中，包括Gnu项目的bison。



第5章
中间表示

5.1 概述

正如我们所看到的那样，编译器被组织成一系列的遍，其中每一遍扮演着不同的角色。这样的结构导致对被编译代码的中间表示的需求。因此，编译器必须使用某种内部形式，即中间表示或IR来表示被分析和被翻译的代码。大多数遍消耗IR；绝大多数遍生成IR。很多编译器在编译过程中使用多个IR。在这样的方案中，中间表示成为主要的或决定性的代码表示。

为了实现这一工作，IR必须有足够的表示能力来记录编译器各遍之间传输的所有有用事实。在翻译期间，编译器导到源代码中没有表示的各种事实，例如变量和过程的地址等。典型地，编译器为IR配置记录额外信息的一组表格。我们将这些表格考虑成IR的一部分。

209

在算法设计中，必须在线解决的问题与可以离线解决的问题之间有着关键的区别。通常编译器离线工作，也就是说，它们可以对被翻译代码做多遍处理。对代码做多遍处理可以改善编译器所生成的代码的质量。编译器可以在若干遍中汇集信息，并使用这些信息在后面的遍中做出决定。

为编译器设计选择适当的IR需要对源语言和目标计算机的理解以及编译器要编译的程序的性质的理解。因此，一个源语言到源语言的翻译器可能以相当接近源语言的形式保持它的内部信息。相反，为微控制器产生汇编代码的编译器也许使用与目标计算机的指令集合接近的内部形式。

设计特定的IR需要考虑编译器必须记录、分析和处理的信息种类。因此，C语言的编译器需要有关指针值的额外信息，而在Perl编译器中则不需要这样的信息。同样地，Java编译器需要有关类的层次结构的信息，而在C语言的编译器中却没有相应的信息。

最后，实现一个IR迫使编译器设计者考察若干实践上的因素。IR应该为执行编译器经常使用的操作提供廉价的方法。IR应该有表示编译过程中产生的结构的完整组织紧凑方法。最后，IR的实现应该为编译器设计者提供能够直接而简单地检查IR程序的机制。

编译器设计者绝不应忽视这最后一点。IR的清晰且可读的外部形式本身就是有价值的。有时，可以通过增加语法来改进可读性。ILOC中的⇒符号就是一个例子。它除了帮助读者把结果和操作数分离开来之外没有其他目的。

5.2 分类法

为了组织我们对IR的考虑，我们应该认识到在我们设置特定的设计时存在着两条主线。第一，IR有一个结构化的组织。大体说来，IR可以归类于三个组织范畴：

- 图示（graphical）IR以图的形式描绘编译器的信息。以结点和边、列表或树的形式表示算法。第3章中用于描述派生的分析树是一种图表式IR。

210

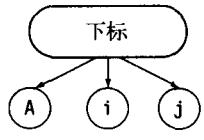
- 线性（linear）IR类似于某种抽象计算机的伪代码。算法在简单、线性的操作序列上迭代。本书所使用的ILOC代码就是一种线性IR。

- 混合型（hybrid）IR把结构式IR与线性IR的成份结合起来，其目的在于抓住两种IR的优势并避开它们的弱点。一个普通的混合型表示使用低级线性IR表示直线代码块，使用图形表示各代码块间的控制流。

IR的结构化组织对编译器设计者如何考虑分析、优化和代码生成有很大的影响。例如，树型IR自然导致把它们的操作嵌入到某种树遍历的遍。同样地，线性IR自然导致在对所有指令进行线性扫描过程中进行操作的遍。

IR分类法的第二条主线是IR表示操作的抽象层次。它的范围从以一个结点表示过程调用的贴近源语言的表示，直到必须将若干IR操作组合起来才能表示一个目标机器操作的非常低级的表示。

为了说明其可能性，考虑编译器是如何在源语言级树和ILOC中表示引用A[i, j]的。假设A是二维数组，且每一维有十个元素。



源代码级树

```
loadI 1      ⇒ r1
sub   rj, r1 ⇒ r2
loadI 10     ⇒ r3
mult  r2, r3 ⇒ r4
sub   ri, r1 ⇒ r5
add   r4, r5 ⇒ r6
loadI @A     ⇒ r7
add   r7, r6 ⇒ r8
load  r8     ⇒ rAij
```

ILOC代码

在源语言级AST中，编译器可以轻松地识别出计算是一个数组引用；检查低级代码，我们发现简单
211 的事实也相当含混不清。当编译器试图决定两个不同引用何时可能接触同一内存位置时，AST的高抽象度可能是很有价值的。相比之下，将上面的低级代码识别为一个数组引用是很难的，特别是当IR已进行了优化，而优化把若干操作移到了过程的其他部分，或者消除了一些代码时，识别就更加困难了。另一方面，如果编译器正在优化为数组地址计算而生成的代码，那么低级代码揭示出在AST中是隐藏着的操作。在这种情况下，较低级的抽象可以为地址计算生成更高效的代码。

高级抽象并非树型IR的固有性质；它隐式地存在于分析树的表记法中。然而，低级表达式树已被很多编译器用来表示诸如A[i, j]的地址计算等计算的所有细节。同样地，线性IR可以有相当高级的结构。例如，很多线性IR都包含把串到串拷贝编码成一个操作的字节拷贝操作。

在某些简单的RISC机器上，最好的串拷贝编码包含清除整个寄存器单元和一个做多字存储跟着一个多字装入的循环上的迭代。需要某些基本的逻辑来处理校正和重叠串的特殊情况。通过使用一个IR指令表示这一复杂操作，编译器设计者可以使优化器更容易地把拷贝移出循环或发现这一拷贝是冗余的。在编译的后期阶段，一个指令被适当地扩展成执行拷贝或对执行拷贝的某个系统程序或库程序的调用的代码。

生成和处理IR的代价应该是编译器设计者关心的问题，因为这些直接影响编译器的速度。不同IR需要的数据空间的变化范围很广。因为编译器通常要涉及所有被分配的空间，数据空间通常与执行时间有直接关系。为了使这一讨论更加具体，考虑用于我们在Rice大学构建的两个不同研究系统中的IR。

- *R*"程序设计环境为FORTRAN构建抽象语法树。树中每个结点占据92个字节。分析器对FORTRAN的每一个源代码行平均构建11个结点，因此对每一个源代码行，其大小刚好超过1 000个字节。
- 由我们小组所生成的研究用编译器使用ILOC的完整实现。（本书中的ILOC是它的一个简单子集。）一个ILOC操作占据23到25个字节。这一编译器对每一源代码行平均生成大约15个ILOC操作。优

化把这一数字降低到对每一个源代码行平均生成刚刚超过3个ILOC操作。

最后，编译器设计者应该考虑IR的表示能力，也就是它为编译器提供记录所需事实的能力。这其中 212
包含定义过程的一系列动作，以及静态分析的结果，前次运行的简档以及调试器所需的信息等。这些信息的表示应该使它们与IR中的特殊点间的关系清晰可见。

5.3 图示IR

很多IR用图形表示被翻译代码。在概念上，所有图示IR都是由结点和边组成的。它们之间的差异在于图与源语言程序之间的关系以及图的结构上。

5.3.1 与语法相关的树

第3章展示的分析树是表示程序的源代码形式的图。很多编译器使用树型IR，其中的树结构对应于源代码的语法。

1. 分析树

有时被称为*语法树*（syntax tree）的*分析树*（parse tree）是对应于输入程序的派生或分析的图形表示。图5-1的左边给出文法，右边给出x × 2+x × 2 × y的分析树。它表示完整的派生，每一个结点表示派生中的一个文法符号。编译器必须为每一个结点和每一条边分配内存。同样地，编译器将数次遍历所有结点和边。因此，值得考虑压缩分析树的方法。

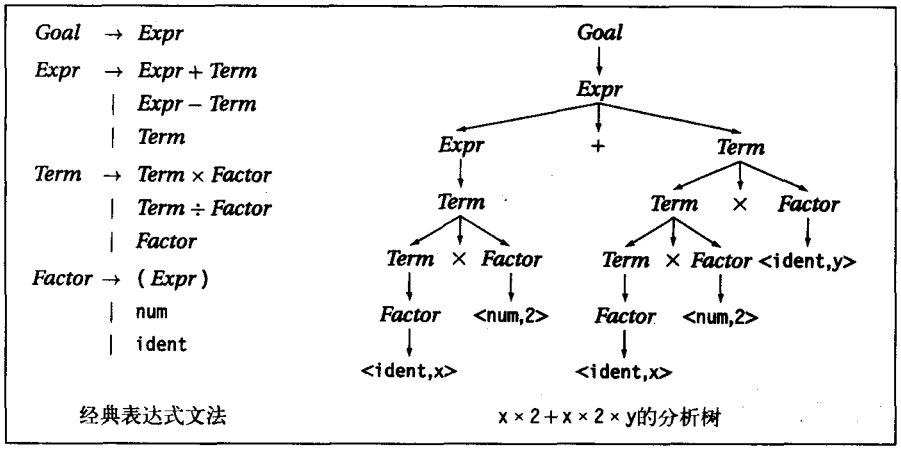


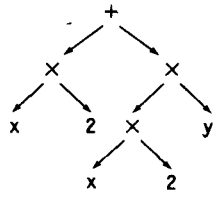
图5-1 使用经典表达式文法的分析树

如3.7.1节所述，文法上的微小转换可以消除派生中的某些步骤以及它们对应的语法树结点。更有效的技术是利用抽象剔出在编译器的余下部分中没有实质目的的那些结点。这一方法导致分析树的一个简化形式。

分析树主要用于语法分析和属性文法系统的讨论中，在那里分析树是主要IR。在需要源代码级树的 213
其他应用中，编译器设计者倾向于使用在本节其余部分描述的更紧凑的形式。

2. 抽象语法树

抽象语法树（abstract syntax tree，AST）保留分析树的本质结构而消除无关结点。表达式的优先度和意义仍然保留着，而无关的结点已不再出现。

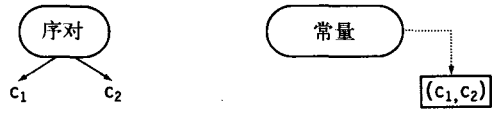


214 x × 2+x × 2 × y的抽象语法树

AST是一种贴近源代码级的表示。因为它大致对应于分析树，在分析器中很容易建立（参见4.4.2节）。

AST运用于很多实际的编译器系统。包括程序设计环境及自动并行化工具在内的源语言到源语言系统一般都依赖于AST，从AST可以容易地重新生成源代码。（这一过程通常被称为优质打印（pretty printing）；它在对AST的中序遍历过程中重新生成源代码文本。）在Lisp中的S-表达式和Scheme的实现本质上是AST。

即使当AST是被用作贴近源代码级的表示时，所选择的特定表示和所使用的抽象也都是需要考虑的问题。*R*″程序设计环境使用下图左子树来表示FORTRAN中的`complex`常量，记作（c_1，c_2）。这一选择适合于语法制导编辑器，在此程序员可以独立地改变c_1和c_2；`pair`结点对应于括号和逗号。



编辑的AST　　编译的AST

然而，这一抽象对编译器来说是有问题的。处理常量的编译器的每一个部分都需要包含处理复常量的特定选择代码。其他常量都有包含指针的一个`constant`结点，指针指向记录在表中的文本串。编译器也许使用如上图的右边所示的复常量表示会更好些。它可以通过消除很多特定选择代码来简化编译器。

在必须表示源代码级信息的系统中抽象语法树有广泛的用途。这些包括基于语言的编辑器、源代码到源代码翻译器以及解释程序。很多编译器以AST为IR；抽象的等级既可以很高也可以很低。如果编译器以另外一个程序设计语言作为它的输出目标，那么AST通常具有相当高级别的抽象。在为某种目标机
215 器生成汇编代码的编译器中，AST的最终版本通常是处于机器指令集的抽象等级或在其等级之下。

存储效率和图示表示

很多实际系统使用抽象语法树表示被翻译的源代码文本。在这些系统中所遇到的一个共同问题是相对于输入文本的AST的大小。巨大的数据结构会限制这些工具能够处理的程序的大小。

R″程序设计环境中的AST结点非常大，足以给20世纪80年代工作站的有限内存系统提出问题。树的磁盘I/O成本减慢了所有*R*″工具。

导致AST大小的急剧增加的原因是多样的。*R*″只有一种结点，这使得结点结构包含所有结点所需的所有域。这简化了结点分配，但是却增加了结点的大小。（大致有一半的结点是叶结点，它们不需要指向子结点的指针。）在其他系统中，结点伴随编译器中各遍所使用的多种较小的额外域的增加而生长。有时候，每当增加新的特性和遍时，结点的大小随之增加。

对AST的内容和形式加以关注可以缩小它的大小。在*R*″中，我们构建了分析AST内容和AST的使用状况的程序。我们把某些域结合起来而消除另外一些域。（在某些情况下，重新计算信息要比

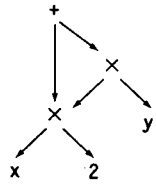
读写它更廉价。）在少数情况下，我们使用散列链表记录不常用的事实，即，使用存储每个结点类型的域中的一位来表明散列链表中存在额外的信息。（这避免了分配那些很少使用的域。）为了在磁盘上记录AST，我们通过前序遍历树把AST转换成线性表示；这消除了记录任意内部指针的必要性。

在R^n中，把所有这些技术结合起来可以使AST在内存中的大小减小约75%。在磁盘上，在移走指针后，文件的大小大约是它们的内存表示大小的一半。这些变化使R^n可以处理更大的程序并使工具更加迅速。

3. 有向无环图

尽管AST比语法树更紧凑，但是它却忠实地保留了原来的源代码的结构。例如，x × 2+x × 2 × y的AST包含表达式x × 2的两个不同拷贝。*有向无环图*（directed acyclic graph，DAG）是避免这一复制的AST的缩写形式。在DAG中，结点可以有多个父结点，相同的子树被复用。这使得DAG比相应的AST更紧凑。 216

对于没有赋值的表达式，文本上相同的表达式必须产生相同的值。x × 2+x × 2 × y的DAG反映了只使用x × 2的一个拷贝的事实，如下所示。

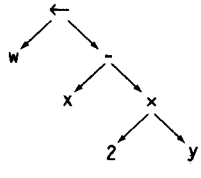


这个DAG在它的形状中描绘了这一表达式的一个明确提示。如果在x × 2的两次使用之间x的值不发生变化，那么编译器可以生成评估子树一次而使用该结果两次的代码。这将为整个表达式带来更好的代码。然而，编译器必须证明这个x的值不改变。如果表达式既不包含赋值也不包含对其他过程的调用，那么上述事实的证明很简单。因为无论是赋值还是过程调用都可能改变与某个名字相关的值，当子树的操作数发生变化时，这个DAG结构必须使子树无效。我们将在8.3.1节中对DAG结构加以描述。

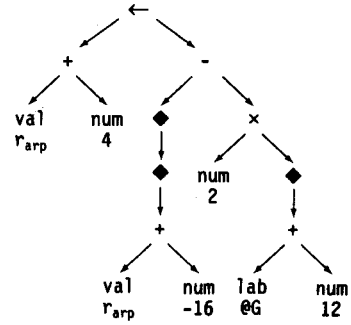
在实际系统中使用DAG有两个原因。有些系统受益于DAG的较小内存使用。特别是内存约束通常限制编译器处理的程序大小。DAG有助于减小这一限制。其他系统使用DAG揭示可能的冗余。这里，其效益在于更好的编译代码。这些系统倾向于把DAG用作衍生IR：构建DAG，转换衍生IR，然后丢弃DAG。

4. 抽象的级别

我们所有的树的例子都给出与源代码非常相近的树。也可以使用树来表示代码中低级别的细节。事实上，优化和代码生成的基于树的技术可能需要这样的细节。举个例子，考虑语句w←x-2 × y。源代码级的AST以简明的形式描述这一语句，如下所示： 217



源代码级AST



低级AST

然而，源代码级树缺少把这一语句翻译成汇编代码所需的很多节细。如上图右边所示的低级树可以使得这些细节清晰可见。这一树引入四个新结点类型。val结点表示已在寄存器中的值。num结点表示已知的常量。lab结点表示汇编级标签，典型地是一个可重定位符号。最后，◆是解除引用一个值的运算符；它把这个值当作一个内存地址并返回在那个地址处的内存内容。

这一低级树展示与这三个变量有关的地址计算。w被存储在离r_{arp}中的指针偏移为4的地方，r_{arp}存放指向当前过程的数据域的指针。x⊖的双重解除引用表明它是一个引用形式的形参，可以透过存储于在r_{arp}之前16个字节的指针存取。最后，y被存储在标签@G之后偏移为12的地方。

注意w评估到一个地址，因为它是赋值语句的左部。与此相反，由于操作符◆，x和y都评估到值。

5.3.2 图

尽管树为在分析过程中所发现的源代码的文法结构提供一种自然的表示，但是它们的呆板结构使得
218 它们对表示程序的其他性质用处不大。为了建模程序行为的这些方面，编译器通常以更一般的图为IR。在前面一节中引入的DAG就是图示IR的一个例子。

1. 控制流图

控制流图（control-flow graph，CFG）模型化程序中控制的流向。CFG是一个有向图，$G=(N, E)$。每一个结点$n \in N$对应于一个*基本块*（basic block）。基本块是总在一起执行的操作序列。控制总是从基本块的第一个操作进入该基本块并从它的最后一个操作离开。每一个边$e=(n_i, n_j) \in E$对应于从块n_i到块n_j的控制的可能转移。

为了简化第8章和第9章中的程序分析的讨论，我们假设每一个CFG有惟一的入口点n_0和惟一的出口点n_f。在一个过程的CFG中，n_0对应于这个过程的入口点。如果过程有多个入口，那么编译器可以插入一个n_0并加入从n_0到每一个实际入口点的边。同样地，n_f对应于这一个过程的出口点。多个出口比多个入口更一般，但是编译器很容易插入一个n_f并加入连结每一个真实出口到这个n_f的边。

CFG给出可能的运行时控制流路径的一种图形表示。图5-2给出两个例子：一个是循环，另一个是简单的条件控制。这与诸如AST这样的面向语法的IR不同，在这样的IR中边表示文法结构。因此，*while*循环的CFG包含一个环，而它的AST是无环的。条件控制的CFG如期望的那样是无环的。它表示控制总是从$stmt_1$和$stmt_2$流向$stmt_3$。在AST中，这一关系是隐式的，而不是显式的。

编译器通常与另一个IR一起使用CFG。CFG表示块之间关系，而使用诸如用表达式级AST、DAG或线性三地址码这样的IR来表示操作块内的操作。这种组合可以认为是一个混合型IR。

CFG实现中的一个权衡问题是决定每个块内包含多少代码。两个最一般的粒度是单一语句块和基本块。有时，我们使用单一语句块来简化分析和优化的算法。

使用单一语句块生成的CFG比使用基本块所生成的CFG大。很多优化和代码生成技术注释CFG中的结点和边。因为单一语句CFG有更多的结点和边，它可能需要更多的注释空间和更多的时间，并为构建这些注释并把它们从一个块拷贝到另外一个块而付出更多的努力。

219 使用基本块的CFG的结点和边要比单一语句的CFG的结点和边少。在这一模型中，编译器必须找到每一个块的开始和结束，而这对单一语句块来说是一个显然的问题⊜。这一模型减小了保存注释所需要

⊖ 原文中为y，应为x。——译者注

⊜ 一般地，基本块开始于带标签语句，因为可以沿着多条路径达到它们。一个过程中的第一个操作，无论是否带有标签，都是一个块的开始。块结束于一个分支或跳转。谓词操作也结束块，因为事实上它们在代码中创建两条路径。

的空间。然而，计算和使用注释的算法必须处理更大的块所拥有的聚集。可能需要某种解释来决定在一个块内的某一个特殊点处哪些事实为真。

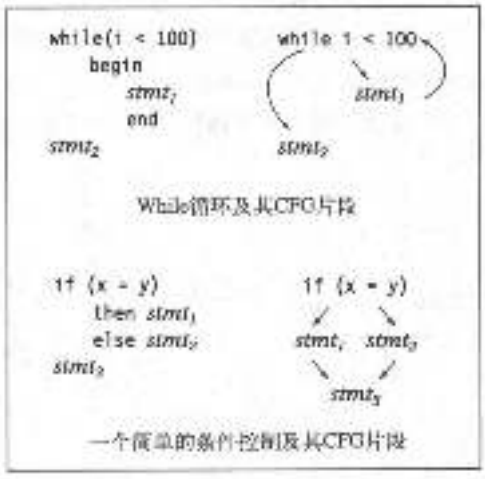


图5-2 控制流图片段

编译器中很多不同的活动或者显式地或者隐式地依赖于控制流图。支持优化的分析一般开始于控制流分析和CFG的构建（参见第9章）。指令调度需要CFG来理解各个块的已调度代码是如何在一起流动的（参见第12章）。全局寄存器分配依赖于CFG来理解每一个操作的执行频率以及放置在寄存器与内存之间移动值的操作的位置（参见第13章）。 220

2. 相关图

编译器使用图编码从一个值的创建点即定义点（definition point）到它的使用点（use point）之间的流向。数据相关图（date-dependence graph）直接编码这一关系。

数据相关图中的结点表示操作，大多数操作既使用值又定义新值。数据相关图中的边连结两个结点，其中一个结点使用另外一个结点定义的结果。我们使用从定义到使用的边画出相关图。

为了具体化这一讨论，图5-3再次给出图1-3的例子，并给出它的数据相关图。这个图对块中每一个语句有一个结点。每一条边表示一个值的流向。例如，从3到7的边反映出语句3中的r_3的定义以及其后它在语句7中的使用。r_{arp}的使用指出它在这一过程的开始处的隐式定义；我们用粗边表示这些使用。

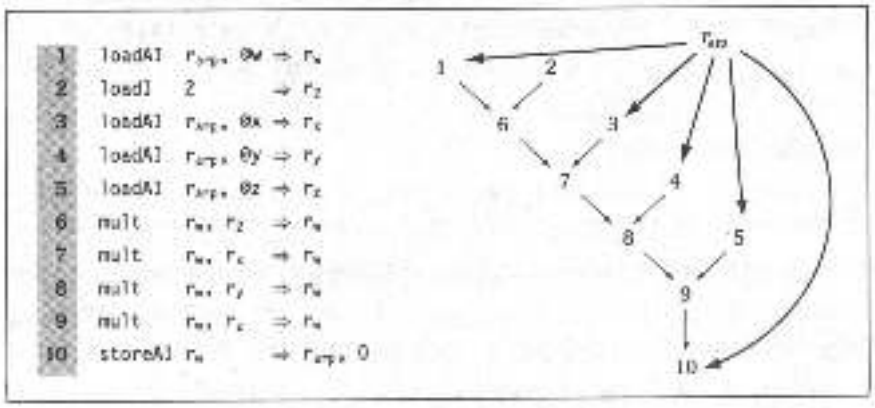


图5-3 一个ILOC基本块和它的相关图

图中的边表示对这一操作序列的实际限制。也就是说，一个值在它定义之前是不能使用的。然而，这一相关图没有充分描述这一程序的控制流。例如，这个图要求1和2领先于6。然而，并没有要求1或2领先于3。很多执行序列保持代码所示的相关性，包括<1，2，3，4，5，6，7，8，9，10>和<2，1，6，3，7，4，8，5，9，10>。这一偏序中的自由度正是"无序"处理器要利用的。

221 在更高层次上，考虑图5-4所示的代码片段。相关图表明，对a[i]的引用从代表a的先前定义的结点得到它们的值。这通过一个结点把a的所有使用都连接起来，不对下标表达式做复杂的分析，编译器是无法区分对各个数组元素引用的。

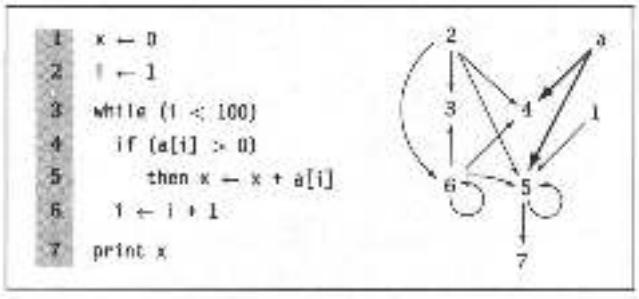


图5-4 控制流和相关图间的相互作用

这一相关图比前面的例子更复杂。结点5和结点6都依赖于它们自己，它们使用它们在前面的迭代中定义的值。例如，结点6既可能从结点2（在初始迭代中）也可能从它自己（在后面的迭代中）获取i的值。结点4和结点5获取的i的值也有两个不同的来源：结点2和结点6。

数据相关图经常用来派生IR，为了某个特殊任务从明确的IR构造出来，并在使用后丢弃。这些数据相关图在指令调度中起着重要的作用（参见第12章）。它们被运用于各种优化，特别是通过对循环重新排序来揭示并行性的转换，以及改进内存行为的优化中。这些应用通常都需要对数组下标做更复杂的分析，以便更精确地决定数组的存取模式。在数据相关图的更复杂的应用中，编译器可能对数组下标值执行大量的分析来决定何时对同一数组的引用可能重叠。

5.4 线性IR

图示IR之外的另一个形式是线性IR。一个汇编语言程序是线性代码的一种形式。它是由按照代码出222 现的顺序（或与这一顺序一致的顺序）执行的一系列指令组成的。指令可能包含多个操作；假如是这样的话，那么这些操作并行执行。用于编译器的线性IR与某个抽象机器的汇编代码相似。

使用线性形式背后的理论很简单。充当编译器输入的源代码是线性的。同样编译器所释放的目标机器代码也是线性的。若干早期编译器使用线性IR；对这些编译器的设计者来说这是很自然的表记法，因为他们在此之前是用汇编代码编程的。

线性IR为操作序列引入一个清晰而实用的顺序。例如，在图5-3中的代码与ILOC代码和数据相关图形成鲜明的对比。ILOC代码具有隐式的顺序；相关图引入一个允许许多不同执行顺序的偏序。

如果线性IR用作编译器中的明确的表示，那么它必须包含描绘程序中各点之间的控制转移的机制。线性IR中的控制流通常模型化目标机器上的控制流的实现。因此，线性代码通常包含条件分支和跳转。控制流划分出线性IR中的基本块；块结束于分支、跳转或带标签操作的前一条指令。

在本书使用的ILOC中，我们在每一个块的结束处包含一个分支或跳转。在ILOC中，分支操作为采用路径和拒绝路径指定标签。这消除在一个块的结束处的任意穿过路径。同时，这些约束使得寻找基本

块并重排序它们变得容易。

编译器中使用多种线性IR。有一些线性IR已不再使用，因为它们所模型化的体系结构已不再使用。单地址码模型化累加器机器的行为。这些代码向编译器揭示出计算机的有限名字空间，使得编译器为适应这些限制进行代码裁剪。同样地，二地址代码模型化带有破坏性操作的机器，在这些操作中一个变量既充当操作数又充当这一操作结果的目标。二地址代码为编译器给出明确选择破坏性操作数的IR。在计算机走出这些指令格式的同时，编译器从相应的IR走了出来。

本节描述用于现代编译器中的两个线性IR：栈机器代码和三地址代码。栈机器代码给出一个紧凑、高效存储的表示。在诸如在执行前需要在网络传输的Jave小程序这样IR的大小是重要因系的应用中，栈机器代码有意义。三地址代码模型化现代RISC机器的指令格式；它对两个操作数和一个结果有不同的名字。你已经对一种三地址代码非常熟悉了，即本书中所使用的ILOC。 223

5.4.1 栈机器代码

栈机器代码有时称为单地址代码，它假设操作数栈的存在。大多数操作从栈中取它们的操作数并把它们的结果压入到栈中。例如，整数的减运算将把栈顶的两个元素移出栈，并把这两个元素的差压入到栈中。栈的规则产生对某些新操作的需要。栈IR通常包括交换栈顶两个元素的swap操作。人们已经构建了若干种基于栈的计算机；这一IR似乎是应这些机器的编译要求而生的。图5-5左边一栏给出表达式x-2 × y的单地址代码。

栈机器代码	三地址代码
push 2	$t_1 \leftarrow 2$
push y	$t_2 \leftarrow y$
multiply	$t_3 \leftarrow t_1 \times t_2$
push x	$t_4 \leftarrow x$
subtract	$t_5 \leftarrow t_4 - t_3$

图5-5 x-2 × y的线性表示

栈机器代码很具紧凑。栈创建一个隐式名字空间并从IR中消去很多名字。这缩小IR形式的程序大小。然而，使用栈意味着所有的结果和参数都是暂时的，除非代码明确地把它们移到内存中去。

栈机器代码容易生成及执行。诸如Smalltalk 80和Java等语言使用字节码，而字节码是描绘程序的抽象栈机器代码。字节码或者在为目标机器实现的解释器上运行，或者在执行之前被翻译成目标机器代码。这生成一种系统，这一系统为程序发布提供紧凑程序，而且为把代码移植到新的目标机器（解释器的实现）提供简单的解决方案。

5.4.2 三地址代码

在三地址代码中，大多数操作具有形式x←y op z，它带有一个操作符（op）、两个操作数（y和z）和一个结果（x）。诸如立即装入和跳转等操作将需要较少的参数。有时候，需要带有超过三个地址的操作。图5-5最右栏给出表达式x-2 × y的三地址代码。 224

三地址代码因以下几个原因而具有吸引力。第一，不存在破坏性的操作符使得编译器自由地复用名字和值。程序到三地址代码的翻译引入一组新的由编译器生成的名字，这些名字保存各操作的结果。细心筛选名字空间可以揭示出改进代码的新因素。第二，三地址代码相当紧凑。大多数操作都是由四项组成：一个操作符和三个名字。操作符和名字都来自于有限集合。一般地，操作符需要1或2个字节。而名

字通常由整数或表索引来表示；无论哪种情况，4个字节通常就足够了。最后，因为很多现代处理器实现三地址操作，三地址代码可以很好地模型化它们的性质。

不同的三地址代码表示的特殊操作符的集合以及抽象级别有很大不同。通常，三地址IR将包含大部分低级操作，诸如跳转、分支和简单的内存操作等，此外还有较复杂的操作，如mvcl、max或min等。表示这些复杂操作将使这些操作容易分析和优化。

例如，mvcl(move charactera long) 取一个源地址，一个目标地址和一个字符记数。它把内存中源地址开始的特定数目的字符拷贝到目标地址开始的内存中。有些机器，例如像IBM 370，仅用一个指令(mvcl是IBM 370的一个操作码) 就可实现这一功能。在一台在硬件上没有实现这一操作的机器上，也许需要更多操作来执行这样的拷贝。

把mvcl加入三地址代码使得编译器紧凑地表示这一复杂的操作。它允许编译器在不考虑其内部工作的情况下分析、优化及移动这一操作。如果硬件支持与mvcl类似的操作，那么代码生成将把这个IR结构直接映射到该硬件操作上。如果硬件不支持与mvcl类似操作，那么编译器可以在最后的优化和代码生成之前先把mvcl翻译成一系列低级IR操作。

5.4.3 表示线性代码

我们已使用多种数据结构来实现线性IR。编译器设计者所做的这一选择会对IR代码上的各种操作的
225 代价产生影响。因为编译器花费大部分时间处理代码的IR形式，这些代价值得我们注意。尽管我们的讨论集中于三地址码，但是其中很多地方同样适用于栈机器代码（或任意其他线性形式）。

三地址代码通常被当作四元组来实现。每一个四元组由四个域表示：一个操作符，两个操作数（或源）和一个目标。为了形成块，编译器需要将各个四元组联系起来的机制。编译器以不同的方式实现四元组。

实际使用中的IR

在实践中，编译器使用各种IR。为了优化，IBM的FORTRANH编译器使用四元组和控制流图来表示代码。因为这一编译器是用FORTRAN写成的，所以存放三地址代码的数据结构是一个数组。

便携式C编译器PCC的分析器为控制结构释放汇编代码并为表达式构建低级树。它为表达式树细心地生成代码，这基于大部分时间都花费在表达式评估上的假设。

GCC长期以来依赖于一个非常低级的称为寄存器转移语言（RTL）的IR。RTL非常适合于地址计算和标量算术操作的详细优化。它隐藏诸如数组引用等高级内存优化所需的一些抽象。为了处理这些问题，GCC的G++前端在为更一般的优化和代码生成生成RTL形式之前，为C++的某些特殊优化使用AST。

IBM PL.8编译器使用比IBM 801处理器中的指令集合抽象级稍低的低级线性IR。这揭示出后来可以叠入到地址模式计算的一些计算。这一编译器有若干前端和后端；它们都使用相同的IR。HP PA-RISC编译器有类似的结构。

Open64编译器，这是IA-64体系结构的一个开源编译器，使用称为WHIRL的五个相关IR。分析器中的最初转换生成贴近源代码级的WHIRL。后继阶段把更多细节引入到WHIRL程序中，从而降低抽象等级靠近实际机器代码。这使得编译器对源代码文本为基于相关性的转换使用源代码级AST，
226 并且为优化和代码生成的后期阶段使用低级的IR。

图5-6给出三个实现图5-5的三地址代码的不同方案。此图左边所示的最简单的方案使用一个短数组

来表示每一个基本块。编译器设计者通常把数组放置在CFG的一个结点中。（这也许是混合IR的最一般形式。）图中中央所示的方案使用一个指针数组把四元组组织成一个块；这一指针数组可以包含在一个CFG结点中。图中右边所示的最后方案把四元组链接起来形成一个列表。这一方案在CFG的结点中需要的存储最少，它的代价是我们只能做顺序遍历。

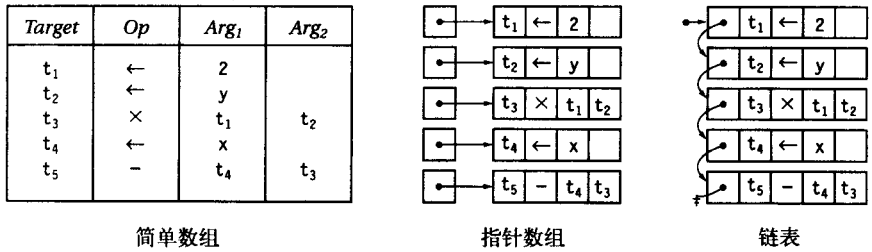


图5-6 三地址代码的实现

227

考虑在这一块中重新安排代码所产生的代价。第一个操作把一个常量装入到一个寄存器中；在大多数机器上，这直接翻译成一个立即装入操作。第二个和第四个操作从内存装入值，在大多数机器上，这一操作将带来多周期的等待，除非这些值已在主缓冲器中。为了隐藏一些这样的等待，指令调度器可能将y和x的装入移到在2的立即装入之前。

在简单数组方案中，把y的装入移到立即装入之前要求保存第一个操作的四个域，并把相应的域从第二个位置拷贝到第一个位置，然后用被保存的立即装入的值重写第二个位置的这些域。指针数组方案也要求同样的三步方法，只是只须改变指针的值。因此，编译器保存指向立即装入的指针，把指向y的装入的指针拷贝到这一数组中的第一个位置，并用已保存的指向立即装入的指针重写在这一数组的第二个位置。对于链表方案操作是类似的，不同的只是需要存储足够的指针来遍历链表。

现在，考虑当前端生成首轮IR时发生什么事情。使用简单数组形式和指针数组，编译器必须选定这个数组的大小，事实上，也就是编译器预期一个块中所含的四元组的数量。当它生成四元组时，它填充这一数组。如果数组过大，那么数组会浪费空间。如果数组过小，那么为了得到更大的数组，编译器必须重新分配数组并把过小的数组的内容拷贝到这一新的大数组中。然而，链表避免这些问题。扩展链表只需要分配新四元数组并设置链表的适当指针。

在多遍编译器中，使用不同实现来表示编译过程中不同点处的IR是有意义的。在前端，也就是致力于生成IR的地方，链表也许既可简化实现又可减小整体代价。在指令调度器中，由于它致力于重新安排操作，所以使用数组的两种实现可能更有意义。

注意，某种信息从图5-6中消失了。例如，图5-6没有给出标签，因为标签是块的性质而不是任意单个四元组的性质。在块中存储标签列表比在每个四元组中存储标签节省空间；它还突出标签只出现在基本块的第一个操作上的性质。让标签依附于块，当重新排列块内的操作时，编译器可以忽视这些标签，从而稍微简化操作。

228

5.5 静态单赋值形式

一个程序的*静态单赋值形式*（static single-assignment form，SSA）把有关控制流和数据流的信息加入到这一程序的文本中。SSA形式描绘代码的名字空间中的定义和使用的信息。每一个名字由代码中的一个操作定义，因此也就有了静态单赋值的命名。为了使单赋值规则与控制流的影响一致，SSA形式在控制流路径相遇的地点插入被称之为ϕ-函数的特殊操作。

当程序满足两个约束时它是SSA形式：（1）每个定义有各自不同的名字；（2）每次使用只涉及一个定义。为了把IR程序转换成SSA形式，编译器在不同的控制流路径合并的地点插入ϕ-函数，而且它重新命名变量使得单一赋值性质成立。

为了阐明这些规则的影响，考虑如图5-7的左边所示的小循环。右栏给出SSA形式的相同代码。变量名包括为每个定义创建不同名字的下标。ϕ-函数被插入到多个不同值能够达到块的开头的地点。最后，SSA形式使用两个不同的测试重写while结构来反映最初的测试引用x_0而循环结束测试引用x_2这一事实。

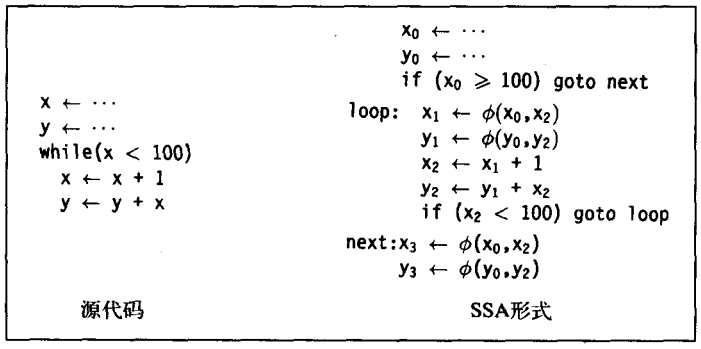


图5-7 SSA形式中的小循环

ϕ-函数的行为方式与众不同。它使用对应于控制进入块的边的参数的值定义它的目标SSA名字。因此，当控制流从上方进入循环时，循环体顶端的ϕ-函数把x_0和y_0的值分别复制到x_1和y_1中。当控制流从循环的底部测试进入循环时，ϕ-函数选择它们的另一个参数x_2和y_2。

在基本块的入口处，所有的ϕ-函数在执行其他语句之前同时执行。因此，这些ϕ-函数都读取适当参数的值，然后它们都定义它们的目标SSA名。使用这种方式定义ϕ-函数的行为使得处理SSA形式的算法忽视块顶端的ϕ-函数的顺序，这是一个重要的简化。正如我们将在9.3.4节中所看到的那样，它会使SSA形式翻译回可执行代码的过程复杂化。

我们为代码优化而设计了SSA形式。SSA形式中ϕ-函数的设置为编译器提供有关值的流向的信息；编译器可以使用这一信息来改进它所生成的代码的质量。这一名字空间消除任意与值的生存期相关的问题。因为每一个值刚好在一个指令中定义，它在从这一指令开始的任何路径上都可用。这两个性质简化并改进很多优化技术。

229 这一例子暴露一些值得解释的SSA形式的奇怪之处。考虑定义x_1的ϕ-函数。这一函数的第一个参数x_0是在领先于循环的块中定义的。它的第二个参数x_2是后来在包含这一ϕ-函数的块中定义的。因此，当这一ϕ-函数第一次执行时，它的一个参数还没有定义。在很多程序设计语言的上下文中，这会引发问题。因为ϕ-函数只读取一个参数，而且这个参数对应于最近在CFG中所取的边，所以它可以从不读取没有定义的参数。

构建SSA

静态单赋值形式是我们描述的IR中惟一没有显然构造算法的IR。9.3节给出算法的详情。然而，构造过程的略图可以阐明一些问题。假设输入程序是ILOC形式。为了将其转换成等价的线性SSA形式，编译器必须首先插入ϕ-函数，然后重新命名每一个ILOC虚拟寄存器。

插入ϕ-函数的最简单方法是在控制流图中有多于一个前驱的每个基本块的开始处，为每个ILOC虚拟寄存器加入一个ϕ-函数。这会插入许多无用的ϕ-函数；完整算法中最复杂的部分就是减少多余

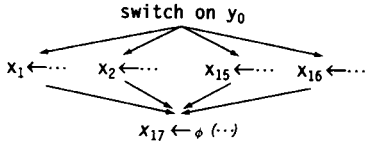
φ-函数的数目。

为了重新命名ILOC虚拟寄存器，编译器可以按深度优先的方式处理块。它为每个虚拟寄存器维护一个计数器。当编译器遇到r_i的一个定义时，编译器递增r_i的计数器，如果递增后的计数器值为k，那么它用名字r_{i_k}重写该定义。在编译器遍历块的过程中，它按r_i的计数器的当时值重写r_i的每一个使用。也就是说，它用r_{i_k}重写r_i，直到遇到另一个r_i的定义。（该定义使这一计数器跳到k+1。）在块的末尾，编译器查看控制流的每个边，并在每个有多个前驱的块中为r_i重写适当φ-函数的形参。

在重新命名之后，代码符合SSA形式的两条规则。每个定义生成惟一一个名字。每个使用只涉及一个定义。存在若干更好的SSA构造算法；它们比这一简单方法插入更少的φ-函数。

230

为了在三地址IR中使用SSA形式，编译器设计者必须包含一个表示任意大的操作的机制。考虑选择语句结尾处的块。



x_{17}的φ-函数必须对每一种情况有一个参数。φ-操作对每一个进入控制流的路径有一个参数；因此，它不适合固定参数数目、三地址方案。

在三地址代码的简单数组表示中，编译器设计者必须或者为这个φ-操作使用多个位置或者使用其他数据结构存放φ-操作的参数。图5-6中所给出的另外两个方案中，编译器可以插入可变大小的三元组。例如，装入和装入立即的三元组也许有恰好两个名字的空间，而φ-操作的三元组也许允许任意数目的名字。

5.6 把值映射到名字

特定IR和抽象等级的选择有助于确定编译器能够处理和优化的操作。例如，源代码级AST使得很容易找到对一个数组x的所有引用。同时，它把存取x的元素所需的地址计算的细节隐藏起来。相反，诸如ILOC这样的低级线性IR揭示出地址计算的细节，其代价是隐藏了与x相关的特定引用。这些影响是众所周知的。

对在执行期间计算的值指定内部名字的规则的选择也有类似的效应。对名字的生成、处理和使用方式的仔细关注可以揭示对优化有价值的因素。而忽视这一问题将导致这些因素含混不清。为了具体化这一讨论，考虑图5-8a所示的短块。

表达式的每一个操作数都是一个值，表达式的评估结果也是一个值。图5-8a中的块具有名字空间{a, b, c, d}。这一名字的选择把多个值映射到一个名字上。在这个块的开始处，我们可以假设b、c和d中的每一个都已被定义。b在第一个语句中的使用与b在第三个语句中的使用涉及不同的值，除非c=d。名字b在第二和第三个语句中的复用没有向编译器传达什么信息；事实上，它可能误导粗心的读者认为这一代码把a和c的值设为同一值。

231

然而，查看第二和第四个语句，这个块显然把b和d设置为同一值。与第一和第三个语句一样，赋值右侧的表达式从文本上是相同的。在第二和第四个语句中，它们使用相同的值。而在第一和第三个语句中，它们使用不同的值。

程序的IR形式通常比源代码包含更多细节。这一细节是由已分析输入程序到IR形式的翻译中生成的。我们的四语句例子可以按多种方法翻译。图5-8b和图5-8c给出其中两个方法。图5-8b使用按源代码中的

那些名字来使用名字，而图5-8c则为计算中的每个值指定新名字。

编译器生成名字的方法可以决定它能为优化找到什么样的因素。图5-8b使用较少的寄存器。它保持哪些计算必须生成相同结果的混淆。图5-8c使用更多的寄存器，但是它具有文本相同的表达式生成相同结果的性质。使用这一命名规则，这一代码显然使得a和c得到不同值，而b和d得到相同值。

	$t_1 \leftarrow b$	$t_1 \leftarrow b$
	$t_2 \leftarrow c$	$t_2 \leftarrow c$
	$t_3 \leftarrow t_1 + t_2$	$t_3 \leftarrow t_1 + t_2$
	$a \leftarrow t_3$	$a \leftarrow t_3$
	$t_4 \leftarrow d$	$t_4 \leftarrow d$
	$t_1 \leftarrow t_3 - t_4$	$t_5 \leftarrow t_3 - t_4$
	$b \leftarrow t_1$	$b \leftarrow t_5$
$a \leftarrow b + c$	$t_2 \leftarrow t_1 + t_2$	$t_6 \leftarrow t_5 + t_2$
$b \leftarrow a - d$	$c \leftarrow t_2$	$c \leftarrow t_6$
$c \leftarrow b + c$	$t_4 \leftarrow t_3 - t_4$	$t_5 \leftarrow t_3 - t_4$
$d \leftarrow a - d$	$d \leftarrow t_4$	$d \leftarrow t_5$
a) 源代码	b) 使用源命名	c) 使用值命名

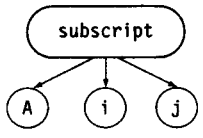
232

图5-8 命名导致不同的翻译

5.6.1 命名临时值

在把源代码翻译成IR的过程中，编译器必须为计算中的很多中间结果创造名字。名字空间的选择令人惊讶地强烈影响着编译器的行为。把名字映射到值的策略很大程序上决定哪个计算可以被分析和优化。

再次考虑我们前面的数组A[i, j]引用的例子。这两个IR片段在完全不同的抽象级别上表示同一个计算。



源代码级树

```
load  1       ⇒ r1
sub   rj, r1  ⇒ r2
loadI 10      ⇒ r3
mult  r2, r3  ⇒ r4
sub   ri, r1  ⇒ r5
add   r4, r5  ⇒ r6
loadI @A      ⇒ r7
add   r7, r6  ⇒ r8
load  r8      ⇒ rAij
```

ILOC代码

低级IR向编译器提供更多细节。这些细节可以从AST推断而得，而且可以生成代码。在一个从AST的直接翻译中，每一次对A[i, j]的引用都将产生独立于上下文的相同可执行代码。

在低级IR中，每一个中间结果有它自己的名字。使用不同的名字为分析和转换提供它们的结果。在实践中，编译器在优化中所能取得的大多数改进都来自于对上下文的利用。为了使这一改进成为可能，IR必须揭示上下文。当命名对许多不同值复用一个名字时，它可以隐藏上下文。如果它创造名字与值之间的一个对应时，它也可以揭示上下文。这一问题不是线性代码的特殊性质；编译器可以使用揭示完整地址计算的低级AST。

命名是SSA形式的一个关键部分。SSA形式使用严格的规则为代码所计算的每一个值生成一个名字，无论它是程序变量还是临时的中间值。这一统一的处理为分析和改进的研究提供很多细节。它描绘有关 233
每个值的生成地点的信息。它为这个值提供一个“永久”的名字，即使相应的程序变量被再定义。使用SSA名字空间可以改善分析和优化的结果。

命名的效应

20世纪80年代末，我们为FORTRAN语言构建了一个优化编译器。我们为前端尝试了若干种不同的命名方案。第一个版本通过增加下一寄存器计数器为每个计算生成一个新的临时寄存器。这生成巨大的名字空间，例如Forsythe、Malcolm和Moler的奇异值分解（singular value decomposition，SVD）的实现有985个名字，它的FORTRAN程序有210行代码。相对于程序的大小，这一名字空间似乎太大了。它引发了寄存分配器的速度和空间问题，在寄存器分配器中，名字空间的大小控制着很多数据结构的大小。（今天，我们已有更好的数据结构及带有更多内存的更快的机器。）

第二版本使用了一个简单分配/释放协议来保存名字。前端根据需要分配临时变量，并当使用完成时立即释放它们。这产生较小的名字空间；例如SVD大约使用60个名字。这加速了分配。例如，在使用SVD解决一个关键数据流问题寻找活变量时，它减小60%的时间。

遗憾的是，用一个临时名字与多个表达式相关联使数据流不清晰并降低了优化的质量。代码质量的下降超过了所有编译时间的效益。

进一步的实践导致了一组规则，它在产生强大的优化的同时还能减小名字空间的增长。

1）每个文本表达式得到惟一一个名字，该名字通过将操作符和操作数登录到一个散列表来决定。这保证一个表达式，例如$r_{17}+r_{21}$，的每一次出现都以同一个寄存器为目标。

2）在代码<op>r_i，$r_j \Rightarrow r_k$中，选择k使得i，j＜k成立。

3）对于寄存器拷贝操作（在ILOC中为i2i $r_i \Rightarrow r_j$），当j对应于一个标量程序变量时，我们允许i＞j。对应于这样一个变量的虚拟寄存器只通过移动操作来设置。表达式评估到它们的“自然”寄存器中，然后它们被移到一个变量。 234

4）只要出现到内存的存储操作（在ILOC中为store $r_i \Rightarrow r_j$），在其后立即加入一个从r_i到这个变量的虚拟寄存器的一个拷贝。（规则1表明从该位置的装入总是以同一寄存器为目标。这一规则保证每当变量的内存位置更新时，变量的寄存器也被更新。）

这种名字空间方案对SVD来说大约使用90个名字，但是给出了依据第一个名字空间方案而建立的所有优化。直到我们采用SSA形式为止，编译器一直使用这些规则，而SSA形式有它自己的命名规则。

5.6.2 内存模型

正如命名临时值的机制可以对程序的IR版本表示的信息产生影响一样，编译器对每个值选择存储位置的方法也产生影响。对于代码中计算的每一个值，编译器必须决定这个值的驻留位置。对于要执行的代码，编译器必须指定特定的位置，诸如寄存器r_{13}或距离标签L0089 16字节的位置等。然而，在代码生成的最后阶段之前，编译器可以使用描绘内存谱系中的层次，例如寄存器或内存的符号地址，而不是描绘那个层次中的特定位置。

考虑本书所使用的ILOC例子。符号内存地址由在它的前面的加上字符@来表示。因此，@x是x在包

含它的存储域中距离开始点的x的偏移量。因为r_{arp}存放活动记录指针，所以使用@x和r_{arp}计算地址的操作隐式取决于把变量x存储在为当前过程的活动记录而被保留的内存中的决策。

编译器通常工作于下面两个内存模型中的一个。

1）*寄存器到寄存器模型*（Register-to-Register Model）在这一模型下，编译器尽量把值保存在寄存器内，而忽视机器的物理寄存器单元所带来的任意限制。所有可以在其生存期的大部分时间都合法地保存在寄存器中的值都被保存在寄存器中。只有当程序的语义需要将一个值存储到内存中时，它才被存储到内存。例如，在过程调用时，地址被当作参数而传送给被调用过程的所有局部变量都必须被存回内存中。不是生命期的大部分时间都能保存在寄存器中的值被存储在内存中。对于这样的值，编译器生成在每一次它被计算使存储它的值并在每一次使用时装入它的值的代码。

235

2）*内存到内存模型*（Memory-to-Memory Model）在这一模型下，编译器假设所有值都存放在内存单元中。值在使用前从内存移到一个寄存器中。值在定义之后立即从寄存器移到内存中。与寄存器到寄存器模型相比，在这一模型中，代码的IR形式中命名的寄存器数目要少。在这一模型中，设计者也许会发现在IR中加入内存到内存操作的价值，例如内存到内存加法的价值。

内存模型的选择与IR的选择总体上是互不相关的。编译器设计者可以构建内存到内存AST或内存到内存ILOC，也可以构建相应的寄存器到寄存器版本，二者的难易程度相同。（栈机器代码和累加器机器的代码也许不在此列；它们包含它们独有的内存模型。）

内存模型的选择对编译器的其他部分产生影响。对于寄存器到寄存器模型，编译器典型地使用的寄存器数目比目标机器所能提供的数目多。因此，寄存器分配器必须把IR程序中的虚拟寄存器映射到目标机器所提供的物理寄存器上。这通常需要插入额外的装入、存储和拷贝等操作，因而使得代码变慢、变大。然而对于内存到内存模型，代码的IR版使用的寄存器数目通常比现代处理器所提供的寄存器数目少。在这里，寄存器分配器寻找可以长时间保存在寄存器中的基于内存的值。在这一模型中，通过消去装入和存储，分配器使得代码更快、更小。

RISC机器的编译器往往使用寄存器到寄存器模型，这基于两个原因。第一，寄存器到寄存器模型更能够反映RISC体系结构的程序设计模型。RISC机器不具备完整的内存到内存操作，相反，它们隐式地假设值可以保存在寄存器中。第二，寄存器到寄存器模型允许编译器在IR中直接描绘它得到的某些微妙的事实。值保存在寄存器内这一事实意味着编译器在早前已经证明了把这个值保存在寄存器是安全的[⊖]。除非编译器在IR中描绘这些事实，编译器将需要一再证明这一事实。

236

ILOC 9X中的内存操作层次结构

本书所用的ILOC是从称为ILOC 9X的IR中抽象出来的。ILOC 9X用于Rice大学的研究用编译器项目中。ILOC 9X包括编译器用于描绘关于值的信息的内存操作的层次结构。在这一层次结构的底部，编译器对值只有很少的信息或没有信息；在层次结构的顶部，编译器知道真正的值。这些操作如下所示：

⊖ 如果编译器能够证明只有一个名字对某个值提供存取，那么它就可以把这个值保存在寄存器中。如果存在多个名字，那么编译器必须小心，并把这个值保存在内存中。例如，一个局部变量x可以保存在寄存器中，除非它可以被另外一个作用域的代码引用。在诸如Pascal和Ada这样的支持嵌套作用域的语言中，这样的引用可以出现在嵌套过程中。在C语言中，如果程序取x的地址&x并通过这个地址存取这个值，就可能出现这一情况。在Algol和PL/I中，程序可以把x当作引用参数传送给另外一个过程。

操　作	意　义
立即装入	把已知的常量值装入到寄存器
无变化装入	装入一个在运行中不发生变化的值。编译器不知道这个值，但是可以证明它不是由程序的操作定义的
标量装入及存储	对标量的操作，被操作值不是数组元素、结构元素或基于指针的值
一般装入及存储	对可以是数组元素、结构元素或基于指针的值的操作。这是一般情况的操作

通过使用这一层次结构，前端可以把关于目标值的信息直接编码到ILOC 9X代码。在其他遍发现附加信息时，它们可以把值的较一般的装入操作重写成更特别的形式。如果编译器发现某个值是已知常量，它可以用这个值的立即装入来取代一般装入或标量装入。如果定义和使用的分析发现某个位置不能被任何可执行存储操作所定义，那么这个值的装入可以重写成无变化装入。

优化可以利用以这种形式描绘的信息。例如，无变化装入的结果与常量的比较本身一定是不变的。这是一个使用标量装入和一般装入难以证明或不能证明的事实。

237

5.7 符号表

作为翻译的一部分，编译器得到有关被翻译程序要处理的各个条目的信息。它必须发现并存储许多不同种类的信息。它遇到各种各样的名字，包括变量、被定义常量、过程、函数、标签、结构和文件。正如前面一节所讨论的那样，编译器还生成很多名字。对于一个变量，编译器需要知道它的数据类型、存储分类、名字和它的声明过程的词法级别以及在内存中的基地址和偏移量。对于一个数组，编译器还需要知道维数以及每一个维数的上界和下界。对于记录或结构，编译器需要知道域的列表，以及每个域的相关信息。对于函数和过程，编译器需要知道它的参数的数目和各参数的类型，以及返回值的类型；更精密的翻译也许要记录一个过程可以引用或修改的变量的信息。

编译器必须或者在IR中记录这一信息，或者根据需要重新得到它。出于效率的原因，大多数编译器记录事实而不是重新计算这些事实[⊖]。这些事实可以直接记录在IR中。例如，构建AST的编译器也许要作为注释（或属性）在表示变量声明的结点记录该变量的信息。这一方法的优点在于，它为被编译代码使用单一表示。它提供统一的存取方法和单一的实现。这一方法的缺点在于，这单一的存取方法可能是低效的，导航AST寻找适当的声明有其自身的代价。为了消除这种低效性，编译器可以遍历IR使得每一个引用都有一个指向相应声明的链结。这会增加IR的空间和IR构建器的开销。

238

如我们在第4章中已看到的那样，另一个选择是为这些事实创建一个中心存储室并提供对它的有效存取。被称为符号表的这一中心存储室成为编译器IR的一个主要部分。符号表局部化从源代码的可能不同部分得到的信息。符号表使得我们能够容易、高效地使用这样的信息，它简化必须引用在编译的早期阶段得到变量信息的所有代码的设计与实现。它避免通过搜寻IR来寻找表示变量声明的部分所带来的消耗；使用符号表通常可以消除直接在IR中表示声明的必要性。（源代码到源代码的翻译是一个例外。编译器可以为了效率而构建符号表，并在IR中保存声明的语法，这样编译器就能够生成与输入程序极为相似的输出程序。）符号表消除让每一个引用包含指向声明的指针的开销。它使用一个从文本名字到存储信息的计算映射来取代这些指针。因此，在某种程度上，符号表就是一种高效处理。

⊖ 这一规则的一个通用例外发生于把IR写到外部存储器时。与计算相比，这样的I/O活动是代价高昂的，而且当编译器读取信息时，它在IR上做一个完整的遍。对于某些信息来说，重新计算比把它们写到外部存储器然后再读回来代价更低。

在本书的很多地方，我们都使用“这一符号表（the symbol table）。”正如我们将在5.7.4节中看到的那样，编译器可以包含若干不同、特化的符号表。细心的实现可以对所有这些表格使用相同的存取方法。

符号表实现需要留意细节。因为几乎翻译的方方面面都引用符号表，存取的高效性是至关重要的。因为在翻译前编译器不能预测它将遇到的名字的数目，扩展符号表的操作必须既要适度又要高效。本节对在设计符号表中出现的问题提供高级别的处理。它从编译器的特定观点展示符号表的设计与使用。对于更深层的实现细节和设计选择请参见附录B的B.4节。

5.7.1 散列表

符号表的设计中最重要的问题是效率。编译器会经常存取符号表。因为散列表提供常量期待查找时间，所以散列表是实现符号表的选择。从概念上讲散列表是一流的。它们使用*散列函数*（hash function）h把名字映射到小整数上，并使用这个小整数作为这个表的索引。使用散列符号表，给定名字n，编译器在表中位置（槽）$h(n)$存放该名字的所有信息。图5-9给出一个简单的十槽散列表。这一散列表是记录的一个向量，每个记录保存编译器生成的单一名字的描述。名字a、b和c已经被插入表中。名字d正在被插入表中$h(d)=2$处。

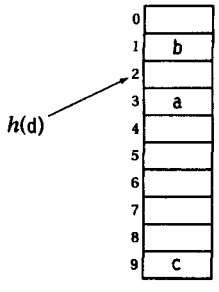


图5-9 散列表实现概念图

使用散列表的主要原因是提供以文本名字为关键字的常量期待时间查找。为了达到这一查找期待值，h的计算代价必须很低。给定一个适当的函数h，存取n的记录需要计算$h(n)$并索引到表中$h(n)$的位置。如果h把两个或多个符号映射到相同的小整数上，就会出现一个“冲突”。（图5-9中，如果$h(n)=3$，就会发生这一情况。）实现必须很好地处理这一情况，保留所需的所有信息并维持查找时间。本节假设h是完全散列函数，也就是说，这一函数从不产生冲突。另外，我们假设编译器事先知道做一个多大的表。B.4节更加详细地描述散列表的实现，包括散列函数、冲突处理和扩展散列表的方案。

散列表有时用作稀疏图的高效表示。给定两个结点x和y，对应于关键字xy的入口表示存在从x到y的边。（这需要一个从小整数对生成良好分布的散列函数，附录B的B.4节中所描述的常量乘积取模散列函数和通用散列函数都适用。）有良好实现的散列表可以提供快速插入和特定边存在性的快速测试。但回答诸如“什么样的结点与x相邻”等问题则需要额外的信息。

散列的替代

散列是组织编译器的符号表中最为广泛运用的方法。多重集判别是另一个吸引人的选择。它消除所有最坏情况行为的可能。多重集判别的关键思路是在扫描器中可以离线构建索引。

为了在符号表使用多重集判别，编译器设计者必须采用不同的扫描方法。编译器不是逐步地处理输入，而是扫描整个程序来寻找标识符的完全集合。当编译器发现一个标识符时，它创建一个二元组<*name*, *position*>，其中*name*是标识符的文本，而*position*是这一标识符在所有记号列表中的序位。编译器把所有这样的二元组放入一个大集合。

下一个步骤是按字典序对这一集合进行排序。它的效果是生成一个子集的集合，每一个子集对应一个标识符。这些子集中的每一个都拥有相应标识符的所有出现的二元组。因为每一个二元组通过它的*position*值与一个特定的记号相关联，所以编译器可以使用这个已排序的集合重写这一记号流。它线性扫描这个集合，按顺序处理每一个子集。编译器为整个子集分配一个符号表索引，然后重写

记号使其包含这一索引。这把标识符记号在符号表的索引附加到该记号中。如果编译器需要一个文本查找函数，可以使用二分查找，因为排序的结果是按字母顺序排列的。

使用这一技术的代价是对记号流的额外一次遍历，加上按字典序排序的代价。从复杂性的观点看，使用这一技术的优点是它避免了散列方法的最坏情况行为的可能，而且在分析前就使符号表的初始大小一目了然。在离线的解决方案可行的绝大多数应用中，这一技术都可以用于取代散列表。

5.7.2 构建符号表

符号表为编译器的其余部分定义两个接口过程。

(1) *lookUp*(*name*)

如果在表中*h*(*name*)处存在记录，则返回该记录。否则，返回一个表明没有找到*name*的值。 241

(2) *insert*(*name*,*record*)

把信息存储于表中*h*(*name*)处的*record*中。它可以扩充这个表以便容纳*name*的记录。

编译器可以分别实现*lookUp*和*insert*的功能，也可以通过向*lookUp*传送一个描述是否插入这个名字的标识来把它们结合起来。它需要保证未声明变量的*lookUp*将会失败，这是发现违背语法制导翻译的使用前声明规则的一个有用性质，也是支持嵌套词法作用域的一个有用性质。

这一简单的接口程序正好适合在第4章中所描述的特定语法制导翻译方案。在处理声明的语法中，编译器给每一个变量构建一个属性集合。当语法分析器识别声明某个变量的产生式时，语法分析器可以使用*insert*把这个名字及其属性加入到符号表。如果一个变量名只能在一个声明中出现，那么语法分析器可以首先调用*lookUp*去检查这个名字是否被重复使用。当语法分析器在声明的语法之外遇到变量名时，语法分析器使用*lookUp*从符号表中得到适当的信息。*lookUp*对于任意未声明的变量名返回失败。当然，编译器设计者也许需要加入初始化这个表、将表存储于外部设备、从外部设备重新得到表以及最后确定该表的函数。对于带有单一名字空间的语言，这一接口程序就足够了。

5.7.3 处理嵌套作用域

只有少数程序设计语言只提供一个集成名字空间。一个语言通常允许程序在多个层次上声明名字。这些层次中的每一个都有一个*作用域*（scope），即可以使用该名字的程序文本中的区域。这些层次中的每一个都有一个*生存期*（lifetime），即该名字具有值的程序执行中的一个期间。

如果源语言允许作用域彼此嵌套，那么前端需要把诸如*x*的引用转化到正确的作用域和生存期的机制。编译器用于实现这一转化的主要机制是一个作用域化了的符号表。

为了这一讨论目的，我们假设程序可以创建任意多个相互嵌套的作用域。我们对词法作用域的更深入的讨论推迟到6.3.1节；然而，大多数程序员对这里讨论的概念有足够多的经验。图5-10给出一个创建五个不同作用域的C语言程序。我们将用表示它们彼此间的嵌套关系的数字对这些辖域做标签。level 0的作用域是最外层的作用域，而level 3的作用域是最内层的作用域。 242

上图右侧的表格给出了在各作用域中声明的名字。在level 2*a*中的b的声明在生成level 2*a*的模块内部隐藏了level 1中的b的声明。在level 2*b*中，b的引用重新引用level 1的参数。同样地，level 2*b*内的a和x的声明隐藏它们早先（分别在level 1和level 0处）的声明。

这一上下文关系创建执行赋值语句的名字环境。用下标表示名字的层次，我们发现level 3的赋值意指：

$$b_1 = a_{2b} + b_1 + c_3 + w_0$$

```
static int w;      /* level 0 */
int x;

void example(int a, int b) {
  int c;           /* level 1 */
  {
    int b, z;      /* level 2a */
      ...
  }
  {
    int a, x;      /* level 2b */
      ...
      {
        int c, x; /* level 3 */
        b = a + b + c + w;
      }
  }
}
```

层次 (Level)	名　字
0	w, x, example
1	a, b, c
2a	b, z
2b	a, x
3	c, x

图5-10　C语言中简单词法作用域示例

注意这一赋值不能使用level 2*a*中声明的名字，因为这个块已经在level 2*b*打开之前关闭了。

为了编译包含嵌套作用域的程序，编译器必须把每个变量的引用映射到它的特定声明。这一过程被称为*名字分解*（name resolution），它把每个引用映射到它所声明的词法层次上。编译器用于完成这一名字分解的机制是在词法上作用域化了的符号表。本节其余部分将描述词法作用域化符号表的设计和实现。
243 相应的运行时机制将在6.5.2节描述，该机制将引用的词法层次转化成地址。作用域化符号表在代码优化中也有直接的应用。例如，8.5.1节描述的超局部值编号算法的效率取决于作用域化散列表。

1. 概念

为了管理嵌套作用域，分析器必须稍微改变一下它对符号表的管理方法。分析器每次进入一个新的词法作用域时，它可以为该作用域创建一个新的符号表。当它遇到这一作用域内的声明时，它把信息加入到当前的符号表。*Insert*在当前符号表上进行操作。当它遇到一个变量引用时，*lookUp*必须首先检查当前作用域的符号表。如果当前符号表没有这一名字的声明，那么它检查周围作用域的符号表[⊖]。通过这样对相继的较低词法层次的符号表依次进行搜索，分析器或者找到这一名字的最邻近的声明，或者在最外层的作用域内失败，失败表明这一变量在当前的辖域内没有可视的声明。

图5-11给出以这种方式为我们的示例程序构建的符号表，这时分析器位于赋值语句处。当编译器为
244 名字b调用修改后的*lookUp*函数时，它将在level 3失败，在level 2失败，并在level 1找到这个名字。这与我们对程序的理解一致，示例中b的最邻近的声明是level 1中的参数。因为在level 2中的第一个模块2*a*已经关闭，它的符号表不在搜索链中。找到这一符号的这一层次形成b的地址的第一部分，在此例中就是level 1。如果符号表中的记录包含每个变量的存储偏移，那么序对<*level*, *offset*>描述在内存的什么位置可以找到b：它在距离*level*作用域存储区开始位置*offset*的位置。我们称这个序对为b的静态坐标。

2. 细节

为了处理这一方案，需要两个额外的调用。为了每一个作用域，编译器需要一个初始化符号表和确定符号表的调用。

(1) *initializeScope*()

递增当前层次并为这一层次创建一个新的符号表。它把这个新表与前面层次的符号表连结起来并更新*lookUp*和*insert*所使用的当前层次指针。

⊖ 指的是当前符号表所涉及层次的上一层次的符号表。——译者注

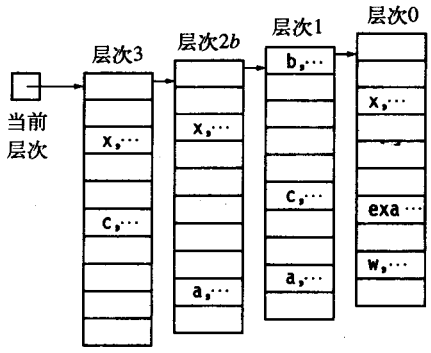


图 5-11

(2) *finalizeScope*()

改变当前层次指针使其指向包围当前层次的作用域的符号表，然后降低当前层次。如果编译器需要保存后来还要使用的层次相关的符号表，那么*finalizeScope*或者把这一符号表完全留在内存中，或者把它写到外部设备上并收回它的空间。

为了对词法作用域进行计数，每当分析器进入一个新的词法作用域时，它都要调用*initializeScope*，并当它离开一个词法作用域时调用*finalizeScope*。

使用这一接口，图5-10中的程序将产生下面一系列调用:

1. *InitializeScope*
2. *Insert*(w)
3. *Insert*(x)
4. *Insert*(example)
5. *InitializeScope*
6. *Insert*(a)
7. *Insert*(b)
8. *Insert*(c)
9. *InitializeScope*
10. *Insert*(b)
11. *Insert*(z)
12. *FinalizeScope*
13. *InitializeScope*
14. *Insert*(a)
15. *Insert*(x)
16. *InitializeScope*
17. *Insert*(c)
18. *Insert*(x)
19. *LookUp*(b)
20. *LookUp*(a)
21. *LookUp*(b)
22. *LookUp*(c)
23. *LookUp*(w)
24. *FinalizeScope*
25. *FinalizeScope*
26. *FinalizeScope*

245

当编译器进入每个作用域时，它调用*initializeScope*。编译器使用*insert*把每个名字加到该符号表上。当编译器离开给定作用域时，它调用*finalizeScope*丢弃该作用域的声明。对于示例中的赋值语句，编译器检查每一个遇到的名字。(*lookUp*调用的顺序会依赖于遍历赋值语句的方式而变化。)

如果*finalizeScope*将已中止层次的符号表保留在内存中，那么这些调用的实际结果将是图5-12给出的符号表。当前层次指针被设置成指向一个无效值。所有层次的符号表被留在内存中，并被链接到

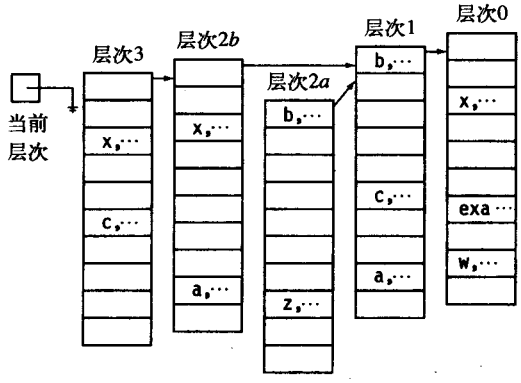


图5-12 示例程序的最终符号表

一起以反映词法嵌套。通过在每一个新层次的开始处在IR中存储指向相应符号表的指针，编译器可以使其后继遍存取相关符号表的信息。这样做的另一个好处是，IR中的标识符可以直接指向它们在符号表的入口。

5.7.4 符号表的多种运用

前面的讨论集中于一个中心符号表，虽然它也许是由几个符号表组成的。实际中，编译器为不同目
246 的构建多个符号表。

1. 结构表

用于对结构或记录中的域命名的文本串出现于与变量和程序的不同的名字空间。名字size可能出现于一个程序中的若干不同的结构中。诸如C语言或Ada语言这样把size用作结构中的域的很多程序设计语言不排除把size当作变量名或函数名来使用。

对于一个结构中的每个域，编译器需要记录它的类型、它的大小以及它在这一记录中的偏移。编译器通过使用处理变量声明同样的机制来收集来自这些声明的信息。它还必须确定这一结构的总体大小，通常这一大小被计算成各域大小的总和，再加上运行时系统所需要的系统负荷空间。

存在若干处理域名的名字空间的方法：

（1）分离表（separate table）

编译器可以为每个记录定义维护一个分离的符号表。概念上，这是一个非常清晰的想法。如果使用多个表的系统负荷很小，正如大多数面向对象的实现那样，那么使用分离表并把它们与这一结构名的符号表入口关联起来是有意义的。

（2）筛选器表（selector table）

编译器可以为域名字维护一个分离表。为了避免由于不同结构中使用相同名字的域造成的冲突，编译器必须使用限定名：将或者这一结构的名字或者惟一映射到这一结构的某种东西，如结构名的符号表索引等，与域名相联结。在这一方法中，编译器必须以某种方式把与每一个结构相关的各个域连结起来。

（3）统一表（unified table）

编译器可以通过使用限定名把域名存储在它的主符号表内。这减少表的数量，但是也意味着主符号
247 表必须支持变量和函数所需要的所有域，以及支持在结构中每一个域筛选器所需要的域。在以上这三个方法中，这个方法可能最不具吸引力。

分离表方法有这样的优势：例如，收回与结构相关的符号表时，它很自然地适合于主符号表的作用域管理框架。当发现一个结构时，它的内部符号表易于通过相应的结构记录存取。

在后面两个方案中，编译器将需要对作用域问题非常小心。例如，如果当前作用域声明一个结构fee，而外部作用域已定义了fee，那么作用域机制必须正确地把fee映射到这一结构上（以及它的对应域的入口）。这可能使得限定名的创建更加复杂。如果代码包含两个fee的定义，且每一个定义都有一个域名size，那么fee.size不是任意域入口的惟一关键字。通常可以通过把由全局计数器生成的惟一整数与每一个结构名相关联来解决这一问题。

2. 面向对象语言中的名字解析链表

在面向对象语言中，名字作用域规则在受到代码结构的控制的同时，在同等程度上受数据结构的控制。这导致一组更加复杂的规则；这也导致一组更复杂的符号表。例如，Java语言需要同时为被编译代码，为代码中已知和被引用的任意外部类，以及为包含这一代码的类的继承层次准备相应的符号表。

一个简单的实现为每个类付上一个符号表，该符号表带有两个嵌套层次，一个是各方法内在的词法

作用域层次，另一个是每个类的继承层次。因为一个类可以充当若干子类的超类，所以后面的层次比简单的捆表图更复杂。然而，它却很容易管理。

当编译类C中的一个方法m时，为了解析一个名字fee，编译器首先查阅m的词法作用域符号表。如果编译器在这一表中没有找到fee，那么它搜寻在这一继承层次中的各个类的作用域，从C开始沿着超类链向上搜索。如果这一查找没有找到fee，那么搜索将在全局符号表中搜索这个名字的类或符号表。全局符号表必须包含当前组件以及导入组件的信息。

因此，编译器需要每一个方法的词法作用域表。当它编译这一方法时构建这一词法作用域表。编译器需要每一个类的符号表，这一符号表向上链接继承层次。它需要到它的组件中其他类以及到组件级变量的符号表的链接。它需要处理每一个导入类的符号表。查找会更复杂，因为查寻必须以正确的顺序跟踪这些链接，而且只检查可视名字。然而，实现和处理这些表所需要的基本机制已经为人们所熟知。 248

5.8 概括和展望

中间表示的选择对编译器的设计、实现、速度和效率产生重要的影响。本章所描述的中间表示没有一个是对所有编译器或对给定编译中的所有任务来说是权威性的正确答案。当设计者选择一个中间形式、设计它的实现、增加诸如符号表或标签表等辅助数据结构时，他必须全面考虑编译器设计的整体目标。

当代编译器系统使用形形色色的中间表示，范围从分析树、抽象语法树（通常被用于源程序到源程序系统）直到比机器层次还低的线性代码（例如用于Gnu编译器系统）。很多编译器使用多个IR，它们构建第二个、第三个IR来执行特殊的分析或转换，然而再修改原来的IR，构建反映分析、转换结果的最终IR。

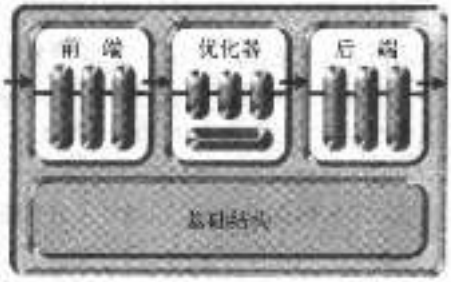
本章注释

关于中间表示及其使用经验的文献很少。这多少有些令人惊讶，因为IR对编译器的结构和行为有着非常重要的影响。典型的IR形式已在很多教科书中有描述[166，30，8，140]。像SSA这样较新的形式[104，267，47]都是在关于分析和优化的文献中描述的。Muchnick给出这一课题的一个现代的处理方法，还给出一个编译器中IR的多层次使用的展望[262]。

使用散列函数来从文本上识别文本相同操作的思想要追溯到Ershov[131]。它在Lisp体系中的特殊应用似乎出现于20世纪70年代初[116，159]；到了1980年，它变得太普通以致McCarthy没有引用直接提出了这一方法[252]。

Cai和Paige把多重集鉴别描述成散列法的一种替代[58]。他们的目的是要给出保障常量时间行为的高效查找机制。关于减小R^n的AST的工作是由David Schwartz和Scott Warren完成的。

实践中，IR的设计和实现对完整编译器的最终特征有非常大的影响。大而且复杂的IR似乎按它自己的想像来构造系统的形态。例如，用于20世纪80年代的早期的程序设计环境如R^n等中的大型AST限制了 249
能够分析的程序的大小。用于ILOC中的RTL形式有低级的抽象。因此，编译器完成管理细节的精细工作，诸如在代码生成中所需要的那些细节管理工作，但是它很少考虑那些需要源代码类知识的转换，诸如改进内存层次行为的循环模块化等等。 250



第6章
过程抽象

6.1 概述

过程是大多数现代程序设计语言的一个核心抽象。过程创建一个可控的执行环境。每一个过程都有属于它自己的命名存储单元。在过程内执行的语句可以存取私有存储单元内的私有或局部变量。过程在被其他过程（或操作系统）调用时执行。这个被调用的过程，即被调用者，可能把一个值返回到它的调用者。在这种情况下这个过程被称为函数（function）。过程间的这样的接口使得程序员得以独立地开发和试测过程；过程之间的这种独立提供了应付其他过程中的问题的某种绝缘层。

过程是多数编译器基本工作单位。少数系统要求一次为编译呈现一个完整的程序。反之，编译器可以处理过程的任意集合。这一特性，称为分块编译（separate compilation），使得构造和维护大型程序变得容易。想像一下在没有分块编译的情况下维护一个一百万行的程序。对源代码的任何改动都将需要一
251 个完整的再编译；在测试一个单行的改动之前，程序员需要等待一百万行代码的编译。

> **关于时机的问题**
>
> 本章研究编译时机制和运行时机制。编译时发生的事件和运行时发生的事件之间的差异常会引发混淆。编译器生成所有在运行时执行的代码。作为编译过程的一部分，编译器分析源程序并构建编码分析结果的数据结构来对分析结果进行编码。（回想5.7.3节中关于词法作用域符号表的讨论。）编译器决定程序在运行时使用的大部分内存布局。然后，它生成创建这一布局、在执行期间维护这一布局以及存取数据对象和内存中的代码所需要的代码。当被编译的代码运行时，它存取数据对象并调用过程或方法。编译时生成所有代码；执行时进行所有的存取。

过程对程序员开发软件以及编译器翻译程序的方式起着重要的作用。过程提供三个构建复杂程序的重要抽象。

（1）控制抽象（control abstraction）

过程向程序员提供简单的控制抽象：每一种语言都有一个调用过程，并把一组变量或参数从调用者的名字空间映射到被调用者的名字空间的标准机制。这一语言的标准返回机制允许过程把控制返回调用者，在调用后的地点继续执行。称为调用序列（calling sequence）或调用约定（calling convention）的这一机制使得分块编译成为可能：编译器可以生成调用任意过程的代码，而不必知道实现这一被调用过程的代码。

（2）名字空间（name space）

每一个过程都创建一个新的受到保护的名字空间：程序员可以声明名字，诸如变量或标签等。在过程的内部，这些声明取代编译器已看到的相同名字的其他声明。在过程的内部，参数通过它们的局部名
252 字引用，而不是通过它们的外部名字引用。因为过程有一个被隔离且受保护的名字空间，当被不同的上下文调用时，它都能正确地运行。

调用过程实例化它的名字空间。调用序列保留足够的空间来存放在这个过程的名字空间中声明的对象。这一空间的配置既是自动的又是高效的，这是调用这个过程的结果。

（3）外部接口（external interface）

过程在大型软件系统的各部分之间定义重要的接口。名字作用域、可寻址性以及运行时环境的有序保存等规则创建一个上下文，在这一上下文内程序员可以安全地调用他人编写的代码。这带来了图形用户界面、科学计算以及系统服务等的程序库的开发和使用。系统使用相同的接口开始用户代码的执行。在建立适当的运行时环境之后，系统代码调用一个指定的入口点，例如main。

在很多方面，过程是构成类Algol语言的基本程序设计抽象。在操作系统的协助下，过程与编译器和硬件环境共同创建一个精美的外表。过程创建命名变量，并把它们映射到虚拟地址；操作系统把虚拟地址映射到物理地址。过程建立名字和可寻址性的可视规则；硬件通常提供若干装入和存储的形式。过程使我们能够把大型软件分解成各个组成部分；链接器和装入器又把它们编织到一起以形成一个可执行程序，硬件通过递增进程序计数器和跟踪分支来执行这一程序。

编译器的一大任务是配置实现过程抽象的各个部分所需的代码。编译器必须规定内存的布局并在生成的程序中对那些布局进行编码。因为编译器可能在不同的时间编译程序的不同组成部分，而且不知道它们与其他组成部分的关系，因此这种内存布局及其引发的所有约定都必须被标准化并统一运用。编译器还必须使用操作系统提供的各种接口，处理输入和输出，管理内存并与其他进程通信。

本章集中讨论作为抽象的过程以及编译器用于建立控制抽象、名字空间及与外部世界接口的机制。 253

6.2 控制抽象

在类Algol语言中，过程有一整套简明的调用/返回规则。在一个过程的出口，控制返回到紧跟调用者调用这一过程的地点后面的地点。如果一个过程调用其他过程，那么这些过程以相同的方式返回控制。图6-1给出一个带有若干嵌套过程的Pascal程序。程序右侧的*调用图*（call graph）和*执行历史*（execution history）很好地概括这一Pascal程序执行时所发生的一切。

这一调用图给出各过程间可能的一组调用。图6-1的程序可以调用Fee两次：第一次是Foe中的调用，第二次是Fum中的调用。事实上，执行历史给出这一程序运行时上述所发生的一切。每一次对Fee的调用都创建Fee的一个不同的实例，即*活动*（activation）。在Fum被调用时，Fee的第一个实例不再是活动的。它由Foe中（事件3）的调用生成并把控制返回到Foe（事件4）。在Pascal中，没有允许控制返回到Fee的这一活动的机制；一旦控制返回（事件4），那么这一活动就终止。因此，当Fum调用Fee时（事件6），它创建Fee的一个新的活动。在控制返回到Fum（事件7）之后，这第二个活动终止。

当程序执行Fee的第一个调用中的赋值x:=1时，活动着的过程是Fee、Foe、Fie和Main。所有这些活动着的过程都位于调用图中从Main到Fee的一条路径上。同样地，当程序执行Fee的第二个调用时，活动着的过程（Fee、Fum、Foe、Fie和Main）都位于从Main到Fee的一条路径上。在执行期间的任意点，这些过程活动构成调用图中的某个有根路径。Pascal的调用和返回机制保证这一点。

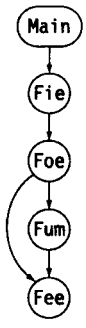
当编译器实现调用和返回时，它必须设法保存调用和返回正确操作所需的足够多的信息。因此，当Foe调用Fum时，调用机制必须保存控制返回到Foe时所需的信息。由于某个运行时错误、一个无限循环或是对某个不返回过程的子调用，Fum可能发散或不返回。另外，调用机制必须保存当Fum返回时恢复在Foe中的执行所需的足够多的信息。

这一简单的调用和返回行为可以用栈来模型化。当Fie调用Foe时，它把一个返回地址压入栈。当Foe返回时，它将这一地址从栈中弹出，并跳转到那个地址。如果所有过程都满足这一规则，那么从栈

中弹出返回地址将暴露出下一个适当的返回地址。

栈机制也可以很好地处理递归。事实上，这一调用机制打开穿过调用图中的环路，并为过程的每一
254 个调用创建一个不同的活动。只要递归终止，这一路径将是有限的，而且返回地址的栈将正确地捕获这一程序的行为。

```
program Main(input, output);
   var x,y,z: integer;
   procedure Fee;
      var x: integer;
      begin { Fee }
         x := 1;
         y := x * 2 + 1
      end;
   procedure Fie;
      var y: real;
      procedure Foe;
         var z: real;
            procedure Fum;
               var y: real;
               begin { Fum }
                  x := 1.25 * z;
                  Fee;
                  writeln('x = ',x)
               end;
         begin { Foe }
            z := 1;
            Fee;
            Fum
         end;
      begin { Fie }
         Foe;
         writeln('x = ',x)
      end;
   begin { Main }
      x := 0;
      Fie
   end.
```



调用图

1. Main calls Fie
2. Fie calls Foe
3. Foe calls Fee
4. Fee returns to Foe
5. Foe calls Fum
6. Fum calls Fee
7. Fee returns to Fum
8. Fum returns to Foe
9. Foe returns to Fie
10. Fie returns to Main

执行历史

图6-1 非递归Pascal程序

为使这一描述更具体，考虑如图6-2所示的阶乘的递归计算。当调用到计算（fact 5）时，这一递归
255 计算生成一系列递归调用：（fact 5）调用（fact 4）调用（fact 3）调用（fact 2）调用（fact 1）。

在这一点，cond语句执行分句（<= k 1），终止递归。这一递归以相反的顺序展开，（fact 1）的调用把值1返回到（fact 2）。它又把值2返回到（fact 3），而（fact 3）把值6返回到（fact 4）。最后，（fact 4）把值24返回到（fact 5），（fact 5）通过24乘以5返回答案120。这一递归程序展示了后进先出的行为方式，所以栈机制正确地跟踪所有的返回地址。

```
(define (fact k)
  (cond
    [(<= k 1) 1]
    [else (* (fact (sub1 k)) k)]
  ))
```

图6-2 Scheme中的阶乘的递归程序

更复杂的控制流

有些程序设计语言，例如Scheme，允许过程返回到过程及它的运行时上下文（通常称为闭包（closure））。当这一闭包被调用时，这一过程在它被返回的运行时上下文中执行。简单的栈不足以实现这一控制抽象。控制信息必须保存在诸如链表这样的更一般的结构中，在那里，返回并不意味着解除内

存单元分配（参见6.3.2）。如果这一语言允许引用生存期超过过程的活动期的局部变量，那么也会引发类似的问题。

6.3 名字空间

在大多数过程语言中，一个完整程序将包含多个名字空间。称为*作用域*（scope）的每一个名字空间把一组名字映射到一组值上，并把过程映射到程序中的一组语句上。作用域可以是整个程序、过程的某个集合、单一过程或者是一小组语句。在一个作用域内，程序员可以创建在这个作用域的外面不可存取的名字。这个作用域可从其他作用域继承某些名字。在一个作用域内创建一个名字fee可能会隐藏fee在周围作用域中的定义，事实上，这样做会使它们在这个作用域内不可存取。总之，作用域规则使得程序员可以控制程序存取信息的方式。 256

6.3.1 类Algol语言的名字空间

大多数传统的程序设计语言继承为Algol 60所定义的很多约定和规则。控制名字可见性的规则尤其如此。本节给出在类Algol语言中有效的命名表记法，并特别强调运用于这种语言中的层次作用域规则。

1. 嵌套词法作用域

大多数类Algol语言允许程序员把一个作用域嵌套在另外一个作用域内。大多数面向对象语言把词法作用域作为名字分解的一个机制；例如，由一个方法所定义的作用域通常存在于包含这一方法的类的作用域内。任意超类的作用域嵌套在这个类的外面。对象的实例变量存在于这一类的作用域内。在过程语言中，当块互相嵌套时就会产生作用域，就像在C或C++中那样。或者当过程互相嵌套时就会产生作用域，就像在Pascal中那样。

Pascal使嵌套过程广为流传。每一个过程定义一个新的作用域，而且程序员可以在每一个作用域内声明新变量和过程。Pascal使用最普通的称为*词法作用域*（lexical scoping）的作用域规则。词法作用域的一般规则很简单：

> *在给定作用域内，每个名字引用它在词法上最近的声明。*

因此，如果s用于当前的作用域内，那么如果在当前作用域内有s的声明则它将引用这个s。如果不存在这样的声明，那么它将引用出现在最近的外围嵌套的作用域中的s的声明。最外面的作用域包含全局变量。

为了使词法作用域更具体，考虑图6-3中给出的Pascal程序。这一程序包含五个不同的作用域，一个对应于程序Main，另外四个分别对应于过程Fee、Fie、Foe和Fum。每一个过程声明名字x、y和z的某个变量子集。图6-3给出的每个名字带有表示其层次数字的下标。在一个过程中声明的名字总是有一个比这个过程的名字层大1的层次。因此，如果如图所示Main是层次0，直接在Main内定义声明的名字如x、y、z、Fee和Fie都是层次1。 257

为了表示词法作用域化语言中的名字，编译器可以对每一个名字使用*静态坐标*（static coordinate）。静态坐标是一个序对$<l, o>$，其中l是符号表中的词法嵌套层次，而o是它在这个作用域的数据区的内存位置的偏移量。为了得到l，编译器前端使用词法作用域符号表，如5.7.3节所述。偏移量o应该连同这一名字及它在符号表中的层次一起存储。（可以在上下文相关分析过程中处理声明时指定偏移量。）图6-3右边的表给出每一过程中的每一变量名的静态坐标。

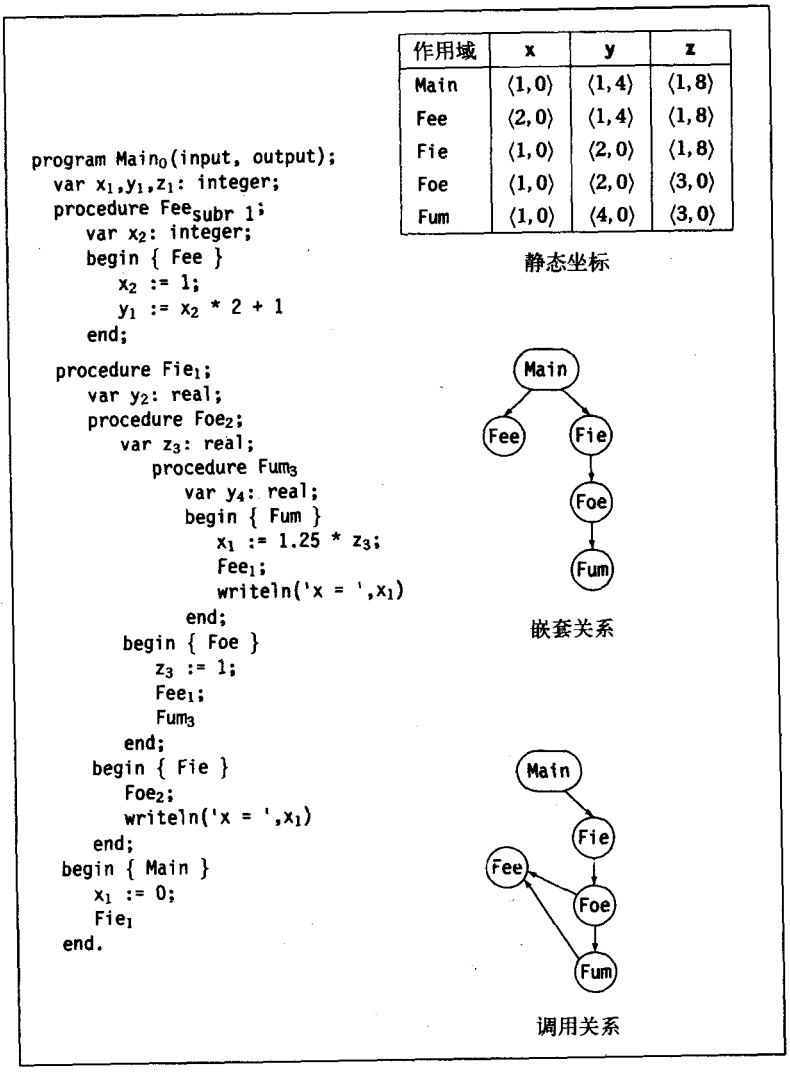


图6-3 Pascal中的嵌套词法作用域

258

在代码生成期间完成名字翻译的第二部分。编译器必须使用静态坐标对运行时的这个值进行定位。给定一个坐标<*l*, *o*>，代码生成器必须发行将*l*翻译成适当数据区域的运行时地址的代码。然后，编译器使用偏移量*o*计算对应于<*l*, *o*>的变量的地址。6.5.2节将给出完成这一任务的两个不同的方法。

2. 各种语言中的作用域规则

程序设计语言有很多不同的作用域规则。编译器设计者必须理解源语言的这些特定规则，而且必须改编一般的翻译方案以适用于这些特定规则。图6-4描述几种语言的名字作用域规则。

最古老的FORTRAN语言创建两种作用域：保存过程名字和公用块名字的全局作用域和一系列局部作用域，每个局部作用域对应于一个过程。一个公用块是由一个名字和一个变量列表组成的；这些公用块的元素只是全局变量。（FORTRAN允许在不同的文件中对一个公用块做不同的描述。这迫使编译器把公用块引用翻译成距离这个公用块开始位置的偏移量，从而保证处理的一致性。）在一个过程内部，程序员可以声明局部变量。如果局部名字与公用块元素名字发生冲突，局部名字覆盖公用块元素名字。在

默认情况下，一个过程的局部变量有与这个过程的调用相匹配的生存期。通过在save语句中提及过程局部变量，程序员可以迫使这个过程的局部变量有与这一程序的生存期相匹配的生存期。这使得这个局部变量成为一个静态（static）变量，在对这一过程的不同调用之间它的值被保留下来。所有全局变量都是静态的，它们的值总是被保留着。

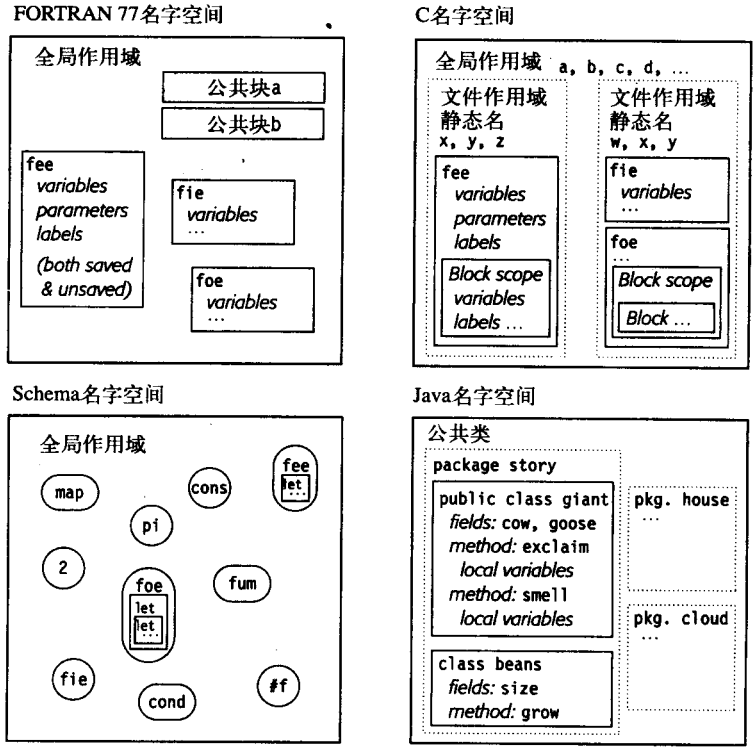


图6-4 几种语言中的名字空间

C语言有更复杂的规则。它创建一个全局作用域来保存所有过程名字以及所有全局变量的名字。每一个过程有它自己的变量、参数和标签的局部作用域。过程不能嵌套在另一个过程内（而Pascal中可以相互嵌套），但是一个过程可以包含一个创建独立局部作用域的块（块是由左大括号和右大括号包围的区域）。块可以嵌套。程序员通常使用块级作用域为预处理器的宏所生成的代码创建临时存储或创建一个作用域为循环体的局部变量。 259

C语言引入另外一个作用域级别，即文件范围作用域，这一作用域包含单一文件（或编译单位）中的所有过程。文件级作用域内的名字是在所有过程的外部使用static属性声明的。没有static属性，这些名字将是全局变量。这些名字对文件中的所有过程都是可视的，而对这一文件外的过程则是不可视的。变量和过程都可声明为静态的。

动态作用域

除词法作用域外，还有一种称为动态作用域的作用域。词法作用域与动态作用域之间的差异仅在于一个过程引用在其自身的作用域外部声明的变量时的行为不同，有时称这样的变量为*自由变量*（free variable）。

对于词法作用域，作用域规则简单且具有一致性：自由变量被绑定到词法上最接近这一所用名字的声明。也就是说，如果编译器从包含这一使用的作用域开始检查相继的周围作用域，那么这一变量就被绑定到编译器发现的第一个声明上。这一声明总是来自于包围这一引用的作用域。

对于动态作用域，规则也同样简单：自由变量被绑定到运行时由该名字最新创建的变量上。因此，当执行遇到一个自由变量时，它就把这一自由变量绑定到这一名字最后创建的实例上。早期的实现创建一个名字栈，每一个已创建的名字被压入到这一栈上。为了绑定一个自由变量，运行时系统自顶向下搜索这个名字栈，直到找到一个带有该名字的变量为止。后期的实现更加高效。

尽管很多早期的Lisp系统使用了动态作用域，但是词法作用域已成为人们选择的技术。动态作用域很容易在解释器中实现，但在某种程度上很难在编译器中高效地实现。它可能产生很难找到又很难理解的错误。动态作用域仍然出现在某些语言中；例如，Common Lisp仍然允许程序员强制变量遵循动态作用域规则。

Scheme有一组简单的作用域规则。Scheme中的几乎所有对象都在一个全局空间内。对象可以是数据或是可执行表达式。系统提供的函数，如cons，同用户编写的代码和数据项共存。由可执行表达式组成的代码可以通过使用let表达式创建私有对象。嵌套的let可以创建任意深度的嵌套词法作用域。

Java有一个限定的全局名字空间；只有那些被声明为“公有（public）”的类是全局的名字。每一个类居于一个组件中。一个组件可以包含多个类。一个类既可包含域（数据项）又可包含方法（代码）。公有类中的域和方法可以被声明成公有成员；这使得这些成员可被其他组件中的类的方法存取或调用。类中的域和方法可被相同组件中的所有类中的方法存取和调用，除非这一域或方法被明确地声明为“私有的（private）”。类可以嵌套在另外一个类中。

260 ~ 261

如果Java类giant声明一个域cow，这个声明在每一个giant的实例中都创建一个域cow。当程序创建新的giant时，每一个giant都有它自己的cow。类giant也可能需要某些公有域，这样的域对于整个类只有一个实例。为了声明这些*类变量*（class variable），程序员要声明一个*静态*（static）域。⊖

6.3.2 活动记录

一个新的独立名字空间的创建是过程抽象的一个重要部分。在一个过程内部，程序员可以声明在这一过程外不可存取的命名变量。这些命名变量可以被初始化到一个已知值。在类Algol语言中，局部变量有与声明它们的过程相匹配的生存期。因此，在这一调用的生存期间这些局部变量需要存储，而且它们的值只有当这一创建它们的调用是活动的时候才有意义。如果一个过程的多个调用在同一时间内都是活动的，那么每一个调用都需要这些局部变量自身的私有拷贝。

为了适用这一行为，编译器设法给一个过程的每一次活动留出一个内存区域。我们称这个块为*活动记录*（activation record，AR）。在大多数环境下，当某一个过程调用q时在运行时创建q的AR，而且当控制从q返回时AR被释放。q的这个AR包含q的局部变量所需要的所有存储以及维护q的状态所需要的所有其他数据。这一AR通常还保存这个过程的这一次调用的返回地址。更方便的是这一状态信息的生存期与局部变量的生存期相同。

q的这个AR把q的运行代码与这一程序的其他部分连结起来。当p调用q时，实现这一调用的代码序列必须既保存p的环境又为q创建一个新环境。因此，这一运行代码为q创建一个AR并把q的执行所需的

⊖ Static这一词的使用也许显得奇怪，除非你考虑到这样事实：一个类变量从一个类被装入直到执行停止的时间内生存。因此，静态变量从它第一次被提及直到执行的结束都保留它的值。

信息以及为重构p的环境而执行的返回序列所需要的信息存储在这个AR中。（其中的某些信息被直接插入到p和q的代码中。）图6-5展示如何布局AR的内容。整个AR通过一个*活动记录指针*（activation record pointer，ARP）来寻址，各域位于距离ARP的正或负的偏移处。 262

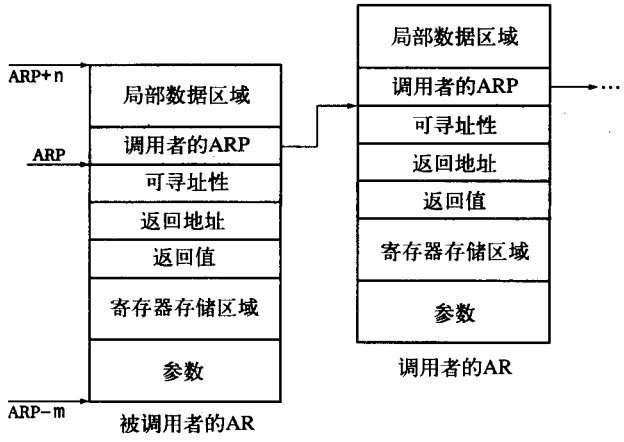


图6-5 典型的活动记录

参数区域保存从调用者到被调用者之间传输的参数。寄存器保存区域保存被调用者在调用序列中必须保存的值。如果需要的话，返回值槽保存用于从被调用者返回到调用者的通信数据，而返回地址槽保存当被调用者终止时执行应该恢复的运行时地址。让被调用者存取它周围词法作用域（不一定是调用者）内的变量的机制可以使用标签为“可寻址性”的槽。被调用者的ARP槽保存调用者的ARP，当被调用者终止时需要恢复它的环境。最后，局部数据区域保存被调用者的局部变量。出于高效性，这一AR的某些部分也许被存放于专用寄存器。

1. 局部存储

一个过程的AR保存它的局部数据和状态信息。这个过程的每一次调用都需要一个独立的AR。对一个过程的AR的所有存取都把这一过程的ARP作为出发点。因为过程经常存取它们的ARP，所以大多数编译器专门开辟一个硬件寄存器来保存当前过程的ARP。在ILOC中，我们把这一专用寄存器称为r_{arp}。 263

ARP指向这个AR中的一个指定位置。这个AR的中心部分有一个静态布局；所有的域都有已知的固定长度。这保证代码可以通过距离这个ARP的固定偏移量来存取那些项。AR的尾部被保留下来，作为其大小在每一次调用可能发生变化的存储区域。AR的一端通常保存参数存储区域，而另外一端保存局部数据区域。

（1）为局部数据保留的空间

每一个局部数据项都可能需要AR内的空间。编译器为每一个这样的项指定一个适当大小的区域，并记录当前的词法级和它在符号表中距离ARP的偏移量。词法级和偏移量这一序对成为此局部数据项的静态坐标。于是，使用类似于`loadAO`的操作，并以r_{arp}和这个偏移量作为这个操作的参数，我们可以对局部变量提供高效的存取。

对于某些局部变量，编译器在编译时不知道它们的大小。程序可能从外部设备读取一个数组的大小，或根据在计算的前期阶段所做的工作来决定数组的大小[⊖]。对于这样的变量，编译器可以在局部数据区

⊖ 例如，编译器的后面的遍可以经常使用更简单的数据结构，因为这些遍可以决定被编译代码的大小。前端必须有可适度扩展的数据结构。前端可以为这些结构记录适当的尺度，使得优化器和后端可以分配给这些结构以适当的大小。

域内为一个指向实际数据或指向一个数组的描述器的指针留出空间（参见7.5.3节）。于是，编译器在运行时在其他地方分配实际的存储空间。在这样的情况下，静态坐标给编译器指出指针的位置，而实际存取或者直接使用这一指针或者使用这一指针计算在可变长度数据区域内的适当地址。

（2）初始化变量

如果源语言允许程序给变量指定初始值，那么编译器必须设法使这个初始值出现。如果这个变量是被静态地分配的，也就是说，这一变量有独立于任意过程的生存期，而且这个初始值是在编译时已知的，那么通过装入器这一数据可以被直接插入到适当的位置。（静态变量通常被存储于所有AR的外面。这样的变量拥有一个实例，即跨越所有调用保存单一值，为我们提供了必要的语义。使用独立的静态数据区域，或者一个过程对应一个静态数据区域，或者整个程序对应一个静态数据区域，编译器可以使用一般在装入器中看到的初始化特性。）

264 另一方面，局部变量必须在运行时初始化。因为一个过程可以被调用多次，设置初始值的惟一可行方法是生成把必要值存储到适当位置的指令。作为结论，这些初始化是在每一次被调用时执行这一过程的第一个语句之前的赋值。

（3）保存寄存器值的空间

当p调用q时，它们中的一个必须保存p所需的寄存器值。也许需要保存所有寄存器的值；另一方面，也许一个子集合就足够了。在返回到p时，必须恢复这些被保存的值。因为p的每一次活动存储一组不同的值，保存寄存器值的空间是设置在AR内的。如果被调用者保存寄存器，那么寄存器的值被存储在被调用者的寄存器存储区域。同样地，如果调用者保存寄存器，那么这个寄存器的值被存储在调用者的寄存器存储区域。对于一个调用者p，在p的内部一次只能有一个调用是活动的。因此，在p的AR内的单一寄存器存储区域足够用于p所能做出的所有调用。

2. 分配活动记录

作为执行从p到q的调用的一部分，执行代码必须为q分配一个AR，并保证AR中的各个域都被填充适当的值。如果如图6-5所示的所有域都的确存储在内存中，那么这个AR必须对调用者p是可用的，这样p可以存储实参、返回地址、调用者的ARP和可寻址性信息。这迫使把q的AR分配植于p中，而在p中这个AR的局部数据区域的大小可能是未知的。另一方面，如果这些值都是在寄存器中传递的，那么这个AR的实际分配可以在被调用者q内完成。这使得q负责分配这个AR，包括局部数据区域所需要的所有空间。分配后，它可能把用寄存器传递的某些值存储到它的AR中。

编译器设计者对于分配活动记录有若干选择。选择既影响过程调用的代价也影响高级语言特性，诸如构建闭包等的实现代价。选择还影响活动记录所需的内存总量。

（1）活动记录的栈分配

在多数情况下，当一个过程的活动引发AR的创建时，这一AR的内容只有在这一过程的生存期才有意义。简而言之，大多数变量不能比创建它们的过程的生存期长，而且大多数过程活动不能比它们的调用者的生存期长。这诸多的限制使调用和返回达到平衡；它们遵循后进先出（last-in first-out，LIFO）
265 的规则。一个从p到q的调用最终要返回，而且在p到q的调用和q到p的返回之间出现的任意返回必须是由q（或者直接或者间接）所做的调用的结果。在这种情况下，活动记录也遵循LIFO顺序；因此，可以把这些活动记录分配到一个栈上。Pascal、c、Java通常都是由栈分配AR实现的。

在栈上保存活动记录有几个优点。分配和释放的代价都不高；每一个操作需要一个作用于标明栈顶的值上的算术操作。调用者可以开始设置被调用者的AR的过程。它可以分配局部数据区域之前的所有空间。然后，被调用者可以通过增加栈顶（top-of-stack，TOS）指针扩展AR来包含局部数据区域。被调用者可以使用相同的机制来逐步扩展当前AR以保存可变大小的对象，如图6-6所示。这里，被调用者

把TOS指针拷贝到A的局部数据区域中。然后通过A的大小增加TOS指针。最后，使用栈分配AR，调试器可以从栈顶到栈底遍历该栈以产生当前活着的过程的快照。

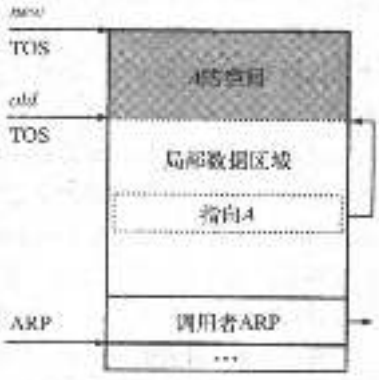


图6-6 动态尺度的数组的栈分配

（2）活动记录的堆式分配

如果一个过程的生存期可以比它的调用者长，那么分配AR的栈规则就不成立。同样地，如果一个过程可以返回诸如闭包这样的包含对其局部变量的显式或隐式引用的对象，那么栈分配不再适合；因为栈分配将留下悬挂指针。在这些情况下，可以将AR保存在堆存储区域中。Scheme和ML的实现都使用堆分配的AR。这样的分配方案要求代码拆除并释放不再需要的AR。在垃圾回收系统（garbage-collected system）中，回收器可以回收这一空间（参见6.7节）。

现代内存分配器可以把堆分配的代价维持在一个很低的水平（参见6.7.2节）。使用堆分配AR，可以把可变大小的对象作为独立的堆对象进行分配。这复杂化显式释放系统的释放；在隐式释放系统中，垃圾回收器将回收那些不可达的可变大小对象的空间。为了追踪整个执行路径，调试器必须从当前的AR向回追踪调用者ARP链。 266

（3）活动记录的静态分配

如果一个过程q不调用其他过程，那么q可能绝没有多个活动的调用。我们称q为一个叶过程（leaf procedure），因为它终止可能过程调用图的一条路径。编译器可以为叶过程静态地分配活动记录，从而消除AR分配的运行时代价。如果这一调用约定需要调用者保存它自己的寄存器，那么q的AR不需要寄存器保存区域。

如果这一语言有类似于Pascal的调用和返回机制，那么编译器可以比为每一个叶过程分配一个静态AR做得更好。在执行期间的任意点处，只有一个叶过程可以是活动的。（如果有两个这样的过程是活动的，那么第一叶过程将需要调用另外一个过程，所以它不是叶过程。）因此，编译器可以为所有的叶过程分配一个的静态AR。这一静态AR必须足够大以便适应这一程序的所有叶过程。在任意叶过程内声明的静态变量都可以在那个单一的AR内进行布局。叶过程单一静态AR所用的空间要小于为每一个叶过程分配独立的静态AR所需要的空间。

当然，禁止递归的任意程序设计语言都可以静态地分配所有的AR。FORTRAN 77具有这一性质。然而，给每个过程创建一个静态分配的数据区域可能增加程序的内存需要总量。除非某一执行路径上的每一过程都是活动的，完全静态解决方案的AR使用空间比基于栈的解决方案的使用空间大。作为一个选择，编译器也许要计算任意调用链所需要的空间最大量；然后，编译器在那个空间上覆盖AR。在实践中，这产生与AR的栈分配相同的代价和行为。

（4）合并活动记录

如果编译器发现一组过程总是以固定的顺序调用的，那么编译器能够把它们的活动记录组合起来。例如，如果从p到q的调用总是产生对r和s的调用，那么编译器也许会发现在同一时间为q、r和s分配AR是有利的。组合AR可以节省分配的代价；这一效益直接与分配代价相关。在实践中，这种优化受到分块编译和函数值参数的使用的限制。此二者都限制编译器确定在运行时实际出现的调用关系的能力。 267

6.3.3 面向对象语言的名字空间

有许多文献论述面向对象设计、面向对象程序设计和面向对象语言。诸如Smalltalk、C++、Self和Java等已开发成为支持面向对象程序设计的语言。许多其他语言已扩展成支持面向对象程序设计特性的

语言。遗憾的是，术语面向对象（object-oriented）已被赋予了过多不同的意义和实现，这使它成为非常宽泛的语言性质和设备的象征。

从根本上讲，面向对象是把面向过程方案的程序名字空间重新组织成面向数据方案的程序名字空间。本节以结果程序的名字空间的观点描述面向对象语言的性质。它把由面向对象语言创建的名字空间与类Algol语言中的名字空间相关联。

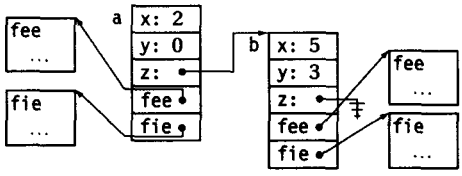
在过程语言中，作用域效应伴随着代码中的转换而出现，即伴随着进入一个过程或块以及离开一个过程或块而出现。在面向对象语言中，作用域规则和命名都围绕着程序中的数据组织起来，而不是围绕着代码组织起来。传统上控制命名规则的这些数据项被称为对象（object）。某些面向对象语言要求每一个数据项都是对象；而在另外一些面向对象语言中，对象和代码都可以包含与类Alogl语言中的变量功能类似的数据项。

从编译器设计者的角度看，面向对象语言不同于过程语言的地方在于它们需要额外的编译时和运行时支持。为了理解这些额外的机制，我们必须首先考虑对象抽象，然后再把它们与过程抽象联系起来。

1. 对象和类

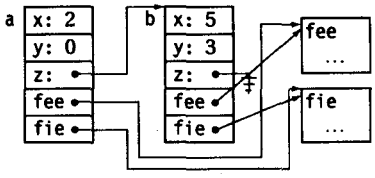
面向对象程序设计的核心是对象的概念。一个对象是一个抽象，它有一个或多个内部成员。这些成员可以是数据项、处理这些数据项的代码或其他对象。作为一个软件工程策略，对象被用于强制抽象、数据隐藏以及控制对存放于这一对象的成员内的信息的存取。

用图形表示，我们可以把对象表示成成员的数组或结构，且有这样的规定：成员可以是数据项、可
268 执行过程或其他对象。



这个图展示名为a和b的两个对象。这两个对象有相同的布局；每一个对象有五个成员。前两个成员x和y保存数。第三个成员z保存一个对象。后两个对象fee和fie保存可执行函数。为了可以统一地处理成员，我们给出的对象成员和代码成员是由指针实现的。

前述的图突出对象的简单模型的实现问题，为了实现fee和fie这一代码的不同拷贝浪费空间。如果代码成员是用指针实现的，那么a和b可以使用fee和fie的相同拷贝。

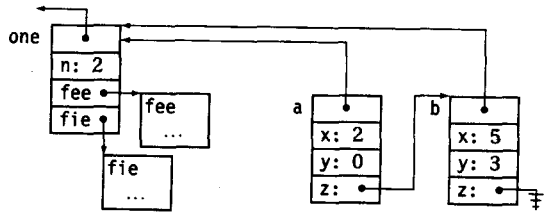


通过处理源语言中的指针来描述这种代码的复用也许是乏味的。为了支持代码的复用，大多数面向对象语言都引入类（class）的概念。一个类是把类似的对象组织到一起的抽象。一个类中的所有对象都有相同的布局。一个类中的所有对象使用相同的代码成员。一个类中的所有对象有它们自己的数据和对象成员。所有成员函数都是由类来描述的。通过对类的引用来创建或实例化各个对象。

通过要求每个对象都是某个类的成员，实现可以通过把类的代码成员存储在表示这个类的一个对象中来优化空间的使用。这给保存这个对象的类的每一个对象添加一个成员。它给每一个代码成员的引用

增加一个间接的层次。用图来表示，这可以表示成如下：

269



这里，one是a和b的类。它保存fee和fie的代码，a和b使用这一代码。a和b的表示已经改变，因为fee和fie的表示已移到了one中。另外，现在a和b的对象记录包含一个指向one的指针。类one有其自己的私有变量n。

2. 术语

这一实现图已变得相当复杂，使得我们有必要引入与各部分相关的术语。

（1）实例

实例（instance）是我们作为对象所考虑的事物。一个实例是一个属于某个类的对象。为了规整起见，我们假设每一个对象都是某个类的实例。

（2）对象记录

实例的具体表示就是它的对象记录（object record）。对象记录包含它的所有成员（或指向它们的指针）。

（3）实例变量

对象的数据成员被称为实例变量（instance variable）。在我们的图表中，a的实例变量x的值为2。在Java中，实例变量被称为域（field）。类one也有一个实例变量n。

（4）方法

某个类的对象共同拥有的每一个代码成员或过程称为一个方法（method）。方法是完整的过程。它们可以有参数、局部变量和返回值。

面向对象语言中的方法与Pascal语言中的过程之间的差异在于其命名的机制不同。只能针对于一个对象来命名一个方法。因此，在Java语言中程序员不能调用fee，而必须写成a.fee，其中a是实现fee的某个类的实例。

270

（5）接受者

我们针对某个对象调用方法，这个对象就是这个方法的接受者（receiver）。在实践中，实现给这个方法增加一个隐式的参数，而这个参数带有一个指向适当对象记录的指针。在这一方法的内部，可以使用一个指定的名字，如this或self来存取这一接受者。

Java对a.fee的调用中，a成为fee内部的接受者，而且可以通过名字this来对其进行存取。

（6）类

类（class）是描述其他对象性质的对象。特别地，类定义为这个类的每一个对象指定实例变量和方法。方法成为这个类的实例变量。

（7）类变量

每一个类可以有实例变量，它们通常被称为类变量（class variable）。它们提供一种永久的存储形式并使其对这个类中的所有方法可视，这种存储形式独立于当前接受者。

因为这与类Algol语言中的静态变量在行为及使用上的相似性，Java和C++都在类定义中把这些变量

声明为静态（static）变量。因此，类的局部变量成为实例变量，而类的静态变量成为类变量。在图中，类one的实例变量n是一个类变量。

3. 继承

大多数面向对象语言包含*继承*（inheritance）的概念。继承给出类上的一种祖先关系，每个类有一个或多个父类，通常被称为*超类*（superclass）。如果β是α的超类，那么α是β的*子类*（subclass），β的方法只要在α内是可视的，那么它对类α的对象也适用。

（相反，不能期待α的方法能对类β的对象适用，因为α的方法可以存取类α的对象所有的附加状态。当α的方法运用到类β的对象上时，这一附加状态将遗失。如果这一调用的确出现了，结果很可能是灾难性的：当这一方法试图存取这一遗失的状态时，将出现某种运行时错误。）

继承允许程序员通过把共用的方法放在超类中，并为具有不同行为的方法使用不同的子类来复用代码。当然，继承要求子类有它的超类指定的所有实例变量：超类的方法必须适用于子类对象的显然要求。
271 另外，把方法名转换成可执行过程的机制必须以一种自然的方式向上沿着超类链扩展。这一典型的规则的工作方式与词法作用域符号表类似；如果在对象自身的类中没有找到这一方法，那么就要按继承顺序搜寻这一对象的超类。调用者使用所发现的第一个实现。

图6-7给出一个例子。如前面一样，a和b是类one的成员。现在，one有超类two，而two有超类three。one、two和three都实现方法fee。另外，它们中的每一个都实现另外一个方法：one实现fie，two实现foe，three实现fum。上图给出第三个对象c，它是two的实例。它有实例变量x和y，但是类two的对象没有z。⊖

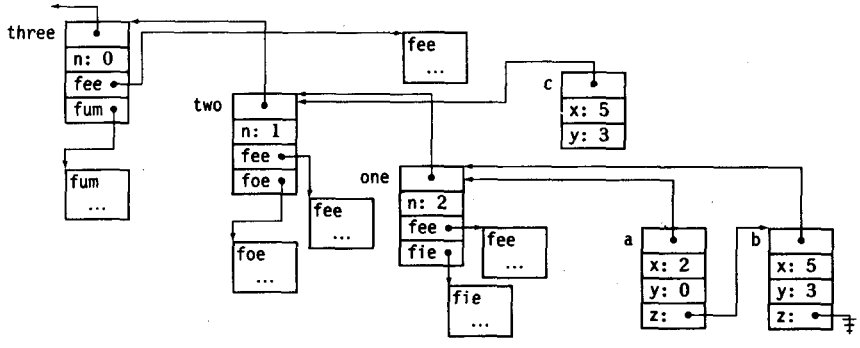


图6-7 在超类继承中寻找方法

4. 将名字映射到方法

在一个面向对象语言中，必须针对一个特定对象调用函数，这一特定对象成为接受者。再次考虑图6-7的结构。如果程序调用one.fee，那么它应该找到在类one中定义的方法fee。c.fee的一次调用应该找到在类two中定义的fee。最后b.fum的调用应该找到在类three中的实现。

272 从概念上，方法查找就如同过程调用的查找一样。对一个方法的搜索开始于接受者的类。如果这一搜索失败，它移升到接受者的超类中去。这一过程反复进行，移升到超类链直到这一搜索或者定位这一方法或者用尽这一超类链而失败。因此，搜索或者返回找到的第一个实现，或者报告失败。

这一方案的一个结果就是前面所提到的不会产生对方法的误用。方法查寻机制保证类α的对象永远

⊖ 这一图仅仅是一个图示；它在对象记录中使用同一个槽作为对象（a、b和c）的类指针和类（one、two和three）中的超类指针。在实际的实现中，超类指针将是每个类的对象记录中的一个独立域。类的类指针毫无疑问指向类class。

不是α的子类的方法的接受者。方法查寻沿超类链向上行进，而不是向下行进。这使我们回忆起强类型语言的规则：方法只在该方法有定义的接受者上执行。

如图所示，实现继承需要小心地生成运行时数据结构以支持方法的定位与调用，这一过程通常被称为调度（dispatching）。函数调用通常不能像类Algol语言那样进行。考虑为类one声明的方法fee内部发生的一切。如果fee的方法调用fie，也就是one所实现的另外一个方法，那么编译器直接生成调用fie的代码吗？如果one没有子类，那么对fie直接调用必定产生正确的行为。

然而，如果one有一个或多个子类，那么对fie的调用必须通过类层次间接地进行处理。如果fie的接受者是one的某个子类的元素而且那个子类实现了它自己的fie，那么这个间接调用将使用这个对象的类指针，这导致fie的适当实现。将fee的实现直接编译到one中将导致调用错误的fie。

为了使调度更高效，实现可以在每一个类中放置一个完整的方法表，如图6-8所示。如果类结构可以完全在编译时确定，那么这一方法表可以是静态的。构建这个表很容易。编译器仅仅把继承层次当作一个有序的作用域集合来处理（每一个类有一个符号表，从子类向超类链接，如5.7.4节所述。）在这样的作用域表内对一个方法名的查寻将把该方法名解析为出现于这个类的方法表的实现。

如果类结构是可变的，或它在编译时是不可知的，那么实现可以在类层次中使用完整查寻，它也可以使用完整的方法表并包含随时更新这些方法表的机制。前者的策略产生一个类似于如图6-7所示的运行时表示。调度包括向上追踪类层次到适当的层次，并调用在那里找到的方法。后者的策略使用如图6-8所示的运行时表示。例如，当通过用一个新类的定义取代某个类的定义而使类结构发生变化时，运行时系统必须为所有受到影响的类重建这个表。 273

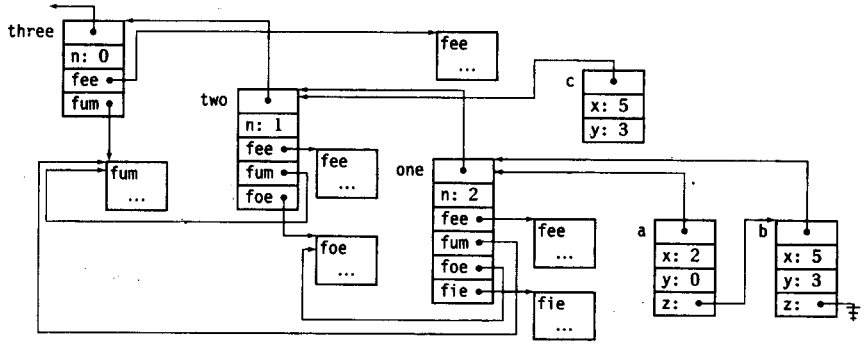


图6-8 在每一类中带有完整方法表的类层次示例

5. 名字可视性规则

从执行方法的角度看，面向对象语言和类Algol语言之间的主要差异在于方法所能存取的变量集合不同。在类Algol语言中，例如在Pascal中，作用域规则是以过程为中心的。每一个过程可以存取它自己的参数和变量、包围词法作用域中的那些参数和变量以及任意全局名字。在面向对象语言中，存取规则是以对象为中心的。

当程序调用方法one.fee时，这一方法能够存取什么呢？首先，fee可以存取在这一方法内局部声明的任意名字，包括被传递的任意参数。fee的代码可以包含词法嵌套作用域，这些作用域的行为方式与它们在类Algol语言中的行为方式一样。（这表明对一个方法的调用创建一个活动记录。）其次，fee可以存取它所知道的one的任意实例变量。（如果fee是一个超类方法，它也许看不到one的所有实例变量。）再次，这一方法可以存取定义fee的类的类变量，以及任意超类的类变量。最后，如果语言支持全局名字作用域，这一方法可以存取全局变量。

为了使上面的讨论更具体，回到图6-7。对于接受者b调用foe使得foe可以存取b的对象记录中的x和y的值，但不能存取z的值，因为foe是在类two中声明的，而two只声明了x和y。Foe可以存取two中的所有类变量，以及three中的所有类变量（以及这一链中其他超类中的类变量）。Foe不能存取one中的类变
274 量，因为它不能存取子类中的任意实现。

与此相对，对于接受者b调用fee使得fee可以存取b的所有实例变量，因为声明fee的类是one。Fee也可以存取one、two和three的类变量。

因为在一个方法中可视的名字集合很大程度上取决于这一语言的类结构，在面向对象语言中的参数传递比典型类Algol语言中的参数传递更为重要。在一个方法内，代码可以调用另一个方法仅当这一方法有适当的接受者对象；这些对象通常必须或者是全局变量或者是参数。

6.4 过程间的值传递

过程的概念的核心是抽象。程序员抽象出与一小组名字即*形参*（formal parameter）相关的公共操作，并把这些操作封装到一个过程中。为了使用过程，程序员通过把适当的值即*实参*（actual parameter）绑定到形参来调用它。被调用过程的执行通过形参名字存取实参传递的值。它还可以返回一个结果。

6.4.1 参数传递

在调用点参数绑定把实参映射到被调用过程的形参。这使程序员能够在没有过程运行的上下文信息的情况下编写这一过程。而且它使程序能够在没有过程的内部操作的信息的情况下，在不同的上下文调用这一过程。参数绑定对于编写抽象的模块化代码的能力是至关重要的。

大多数现代程序设计语言使用两个约定中的一个来在调用点把实参映射到在被调用过程内声明的形参，即*值调用*（call-by-value）绑定和*引用调用*（call-by-reference）绑定。尽管这些技术在行为上不同，它们之间的差异可以通过理解它们的实现而得到最好的解释。

1. 值调用

275 考虑下面用C语言写成的过程以及几个调用它的地点：

```
int fee(int x, int y) {
  x = 2 * x;
  y = x + y;
  return y;
}
```

```
c = fee(2,3);
a = 2;
b = 3;
c = fee(a,b);
a = 2;
b = 3;
c = fee(a,a);
```

如在C语言中那样，对于值调用调用者把实参的值拷贝到对应形参的适当位置：或者是一个寄存器，或者是被调用者的AR中的参数槽。仅有一个名字引用实参值，这就是形参的名字。形参名的初始值是由调用时对实参的评估来确定的。如果被调用者改变它的值，那么这一改变在被调用者中是可视的，但在调用者中是不可视的。

当使用值调用参数绑定来调用时，上面的三次调用产生下面的结果：

值调用	a		b		返回值
	in	out	in	out	
fee(2, 3)	–	–	–	–	7
fee(a, b)	2	2	3	3	7
fee(a, a)	2	2	3	3	6

使用值调用，绑定既简单又直观。

值调用的一个变形是值结果调用。在值结果方案中，作为从被调用者到调用者返回控制过程的一部分，形参的值被拷贝回实参。Ada语言包含值结果参数。值结果机制还满足FORTRAN 77语言定义中的规则。

2. 引用调用

对于引用调用参数传递，调用者在每一个参数的AR槽内存储一个指针。如果实参是一个变量，那么它把这个变量的地址存储在内存中。如果实参是一个表达式，那么调用者评估这个表达式，并把其评估结果存储在它自己的AR中，然后把指向这一结果的指针存储在被调用者的AR中的适当参数槽内。把常量作为表达式处理避免被调用者改变一个常量值的可能性。一些语言禁止把表达式作为实参传递给引用调用的形参。 276

名字调用参数绑定

Algol引入另外一种参数绑定机制，称为*名字调用*（call by name）。在名字调用绑定中，对形参的引用行为是：通过适当的重新命名，形参被实参位置上的文本所替换。这一简单的规则可能导致复杂的行为。考虑下面Algol 60中的人造例子：

```
begin comment Simple array example;
  procedure zero(Arr,i,j,u1,u2);
    integer Arr;
    integer i,j,u1,u2;
    begin;
      for i := 1 step 1 until u1 do
        for j := 1 step 1 until u2 do
          Arr := 0;
    end;
  integer array Work[1:100,1:200];
  integer p, q, x, y, z;
  x := 100;
  y := 200;
  zero(Work[p,q],p,q,x,y);
end
```

对zero的调用把0赋值给数组Work的每一个元素。为了明白这一点，请读者使用实参的文本重写zero。

尽管名字调用绑定很容易定义，但是它很难实现和理解。对每一个形参，编译器通常必须产生评估这个实参到返回一个指针的函数。这些函数被称为*形实转换程序*（thunks）。生成适当的形实转换程序很复杂；为每一个参数存取评估形实转换程序的代价很高。最终，这些缺点超过了名字调用参数绑定所提供的优点。 277

在被调用过程内部，对引用调用形参的每一次引用都包括一个额外的间接层次。引用调用在下面两个重要方面不同于值调用。第一，一个引用形参的任意再定义都在相应的实参中得到反映。第二，任意引用形参可以被绑定到由被调用过程内部的另外一个名字存取的一个变量上。当上述事件发生时，我们说这些名字是*别名*（aliase），因为它们引用相同的存储位置。别名化可能产生非直观的行为。

考虑前面的例子，使用FORTRAN 77重写它，它使用引用调用参数绑定。

```
integer function fee(x,y)
  integer x, y
  x = 2 * x
  y = x + y
  fee = y
  end
```

```
c = fee(2,3)
a = 2
b = 3
c = fee(a,b)
a = 2
b = 3
c = fee(a,a)
```

使用引用调用参数绑定，这一例子产生不同的结果。我们假设FORTRAN编译器用直观的方式处理别名参数，即使FORTRAN 77标准认为最后的调用fee(a, a)不是“标准相容的”。

引用调用	a		b		返回值
	in	out	in	out	
fee(2, 3)	–	–	–	–	7
fee(a, b)	2	4	3	7	7
fee(a, a)	2	8	3	3	8

注意第二次调用重新定义a和b；引用调用的行为倾向于将被调用过程中的更改传递到调用环境中。第三个调用使得x和y在fee中互为别名。第一条语句重新定义a，使其具有值4。下一条语句两次引用a的值，并把a的值加到它自身，而且重新定义a使其具有值8。这使得fee返回值8而不是6。

在值调用和引用调用中，表示参数所需要的空间都很小。因为每一个参数的表示必须在每一次调用时被拷贝到被调用过程的AR中，这对调用的代价产生影响。通过值传递较大的对象要承受拷贝整个对象的负担。有些语言允许通过引用传递数组和结构。而其他语言则包括一些装置让程序员得以指定通过引用传递特殊参数；例如，C语言中的const属性向编译器保证带有这一属性的参数不被修改。

278

6.4.2 返回值

为了从函数返回一个值，与改变它的一个实参的值相反，编译器必须为这一返回值设置空间。因为根据定义，这个返回值是在被调用过程终止后才被使用的，所以编译器需要把它存储在被调用过程的AR的外面。如果编译器设计者能够保证这个返回值有一个较小的固定大小，那么编译器或者把这个值存储在调用者的AR中或存储在一个指定的寄存器中。

我们所有的AR图都含有一个返回值槽。为了使用这个槽，调用者在它自己的AR内为这个返回值分配空间，并把指向那个空间的指针存储在它自己的AR的返回槽内。被调用者可以从调用者的返回值槽装入这一指针（使用在被调用者AR内的调用者ARP的拷贝）。被调用者可以使用这一指针存取为这个返回值而设置在调用者AR中的存储单元。只要调用者和被调用者都接受这个返回值的大小，这就是可行的。

如果调用者不知道返回值的大小，被调用者也许需要给这个返回值分配空间，这个空间通常是分配在堆上。在这种情况下，被调用者分配空间，把这个返回值存储在那里，并把这个指针存储在调用者AR的返回值槽中。这样，调用者可以使用在它的返回值槽中找到的指针来存取返回值。

如果调用者和被调用者都知道这一返回值较小，也就是说返回值槽的大小较小，那么它们可以消除这一间接的操作。对于较小的返回值，被调用者可以直接把这一值存储到调用者AR的返回值槽内。于是，调用者可以直接使用它的AR中的这个值。当然，这一改进要求调用者和被调用者能识别出这种情况并使用同样方式处理这种情况。

（这一改进导致在同一机器上对于同一语言的不同编译器之间不相容的最好例子。想像两个编译器，一个是实现这一改进的编译器，而另外一个是没有实现这一改进的编译器。如果一个可执行代码包含这

两个编译器编译的代码，那么它可能由于对返回值做指针的解除引用而陷入莫名其妙的运行时错误。这种灾难性失败的可能性通常阻止编译器设计者对连接约定做任何哪怕是微小的改动。）

279

6.5 建立可寻址性

作为连接约定的一部分，编译器必须保证每一个过程为所需引用的每一个变量生成一个地址。在类Algol语言中，过程可以引用全局变量、局部变量以及在周围词法作用域内声明的任意变量。地址计算一般由两部分组成：寻找包含这一值的作用域的内存块地址，称为数据区（data area）；以及寻找这一数据区内的偏移。

6.5.1 平凡基地址

对于很多变量，编译器都可使用一个或两个指令发行生成基地址的代码。最容易的情况是当前过程的局部变量。如果这个局部变量被存储在这一过程的AR中，那么编译器可把ARP当作它的基地址。尽管精确的操作序列取决于IR所能表示的寻址模式，但是存在多种选择。这其中包括使用单一的“寄存器+立即偏移”操作（像loadAI），如果这一偏移对立即域来说太大，则使用一个“地址+偏移”操作（像loadAO），或者使用一个三操作序列（loadAI,add,load）。对于所有的情况，地址计算都应该是快速的。

（有时候，局部变量没有被存储在距过程的ARP一个常量偏移的位置上。值也许在一个寄存器中，在这一情况下，装入和存储都是不必要的。如果变量的大小不可预测或不断改变，那么编译器将把这个值存储在用来保存可变大小对象的区域内，或者在AR的尾部或者在堆上。在这种情况下，编译器可以在AR中为一个指向变量真实位置的指针保留一个空间。然后，编译器需要生成一个额外的间接寻址来存取这一变量。）

对全局变量和静态变量的存取的处理是类似的，只是编译器可能需要把这个基地址装入到一个寄存器中。编译器可以为基地址发行一个符号化汇编级标签的立即装入。它必须使用一致性规则来生成这样的标签。编译器一般使用在源语言中不合法的字符给源代码名字加上一个前缀、后缀或者两者都加上。汇编器和装入器将使用正确的运行时值取代这一符号标签。

例如，全局变量fee可能导致一个标签&fee.，假设（&）和（.）都不能出现在源语言的名字中。编译器将发行适当的汇编语言伪操作来为fee保留空间，并把这一标签加到这一伪操作上。为了让fee的运行时地址进入一个寄存器中，编译器将发行诸如loadI &fee.⇒r_i的操作。下一个操作可以使用r_i来存取fee的内存位置。

280

编译器无需知道fee的存储位置。它使用一个可重定位标签来保证适当的运行时地址被写入到指令流中。立即装入操作保证r_i将含有这一适当的地址。汇编器、链接器和装入器都把符号&fee.转换成运行时地址。

全局变量可以各自加标签或在一个较大的组群加标签。例如，在FORTRAN中，语言把全局变量收集到共用块中。一个典型的FORTRAN编译器为每一个公用块创建一个标签。它给每一个公用块中的每一个变量赋一个偏移量，且生成相关于公用块标签的装入（load）和存储（store）操作。如果数据区域比“寄存器+偏移”操作所允许的偏移量大，那么也许把这个数据区域分成若干部分并使用多个标签更具优越性。

同样地，编译器可以把单一作用域内的所有静态变量都聚合到一个数据区内。这减小不希望的名字冲突的可能性；在链接和装入期间可能发现这样的冲突，而且这样的冲突可能给程序员带来困惑。为了避免这样的冲突，编译器可以把全局可视名字与作用域结合起来构造标签。这一策略还减小过程所使用

的基地址的数目，减少对寄存器的需求。使用过多的寄存器来保存基地址可能给整个运行时执行带来负面影响。

6.5.2 其他过程的局部变量

在一个词法作用域语言中，编译器必须提供把静态坐标映射到相应变量的硬件地址上的机制。为了实现一点，编译器必须适当地放置数据结构让编译器计算包围当前过程的词法作用域的AR的地址。

例如，假设fee处于词法层次m，且它引用处于层次n的fee的词法祖先fie内声明的变量a。分析器把这一引用转换成静态坐标<n，o>，其中o是fie在AR内的偏移。编译器可以计算出fee与fie之间的词法层次数目$m-n$。(坐标<$m-n$, o>称为*静态距离坐标*（static-distance coordinate)。这一坐标描述fee的AR与fie的AR之间的词法距离，以及a距fie的ARP的偏移。)

为了把<n，o>转换成一个运行时地址，编译器需要跟踪活动记录间的词法祖先的机制。编译器必须
281 发行保存当前运行时信息所需要的代码。然后，在对另一个作用域的一个局部变量的每一次引用时，编译器必须发行使用这一运行时数据结构来计算所需地址的代码。

有若干机制可以用于解决这一问题。我们将研究其中的两个机制，即存取链和全局显示。

1. 存取链

使用存取链，编译器保证每一个AR包含一个指向它的立即词法祖先的AR的指针。代码使用这一称为*存取链*（access link）或*静态链*（static link）的指针来存取非局部变量。存取链形成一个包含当前过程的所有词法祖先的链，如图6-9所示。因此，对当前过程为可视的另一个过程的所有局部变量被存储在存取链上的一个AR中。

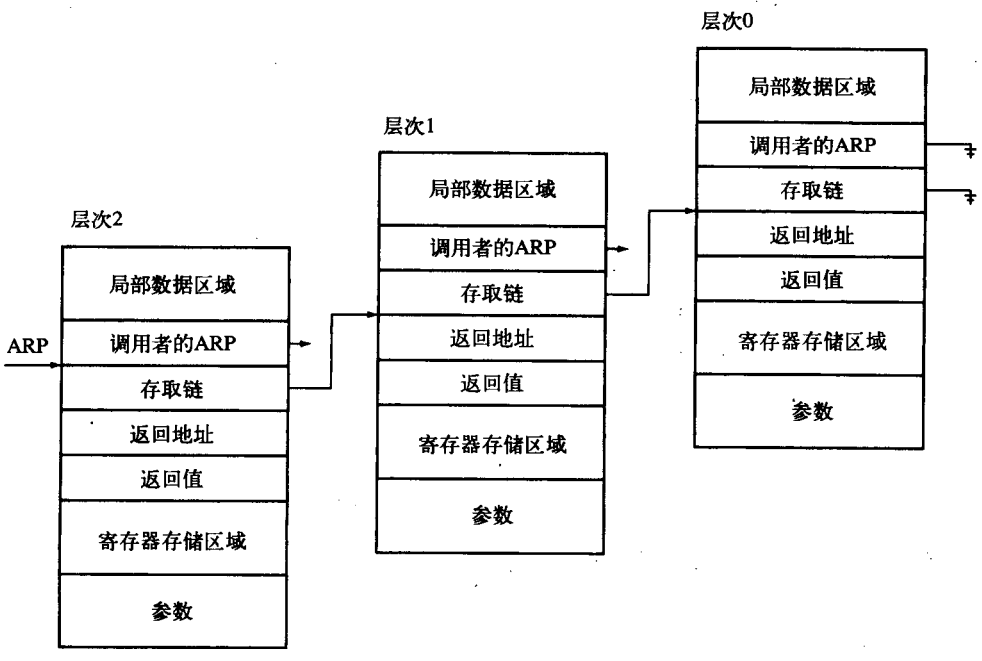


图6-9 使用存取链

282 为了从层次m的过程中存取一个变量<n，o>，编译器发行遍历这一链接链寻找层次n的ARP的代码。然后，编译器使用层次n的ARP和o发行一个装入。为了使这一讨论更具体，考虑如图6-9所示的程序。

假设m是2，且这一存取链存储在距离ARP偏移为-4的地方。下面的表给出编译器为三个不同的静态坐标所生成的ILOC代码。左栏给出静态坐标，右栏给出相应的ILOC代码。每一列把结果留在r_2中。

<2,24>	loadAI r_{arp},24	$\Rightarrow r_2$
<1,12>	loadAI r_{arp},-4	$\Rightarrow r_1$
	loadAI r_1,12	$\Rightarrow r_2$
<0,16>	loadAI r_{arp},-4	$\Rightarrow r_1$
	loadAI r_1,-4	$\Rightarrow r_1$
	loadAI r_1,16	$\Rightarrow r_2$

因为ILOC编译器有每一次引用的静态坐标，所以它可以计算静态距离（$m-n$）。这一距离告诉编译器要生成多少个持续的装入，所以编译器可以为每一个非局部引用发行正确的序列。地址计算的代价与静态距离成正比。如果程序显示较浅的词法嵌套，那么存取不同层次上的两个变量的代价差异也会相当小。

为了维护存取链，编译器必须给每一个过程调用添加寻找适当ARP并将其作为被调用者的存取链存储起来的代码。对于处于层次m的调用者和处于层次n的被调用者，可能发生下面三种情况。如果$n=m+1$，则被调用者嵌套在调用者内，且被调用者可以把调用者的ARP用作它的存取链。如果$n=m$，被调用者的存取链与调用者的存取链相同。最后，如果$n<m$，被调用者的存取链是调用者的$n-1$层次的存取链。（如果n是零，则存取链是空的。）编译器可以生成一个$m-n+1$个装入的序列来寻找这一ARP并把它作为被调用者的存取链存储这一指针。

2. 全局显示

在这一方案中，编译器分配一个被称为显示（display）的单一全局数组来保存在每一个词法层次上的一个过程的最近活动的ARP。对其他过程的局部变量的所有引用变成通过这一显示的间接引用。为了存取一个变量<n，o>，编译器使用这一显示的元素n的ARP，它把o用作偏移并生成适当的装入操作。图6-10给出这一结果。

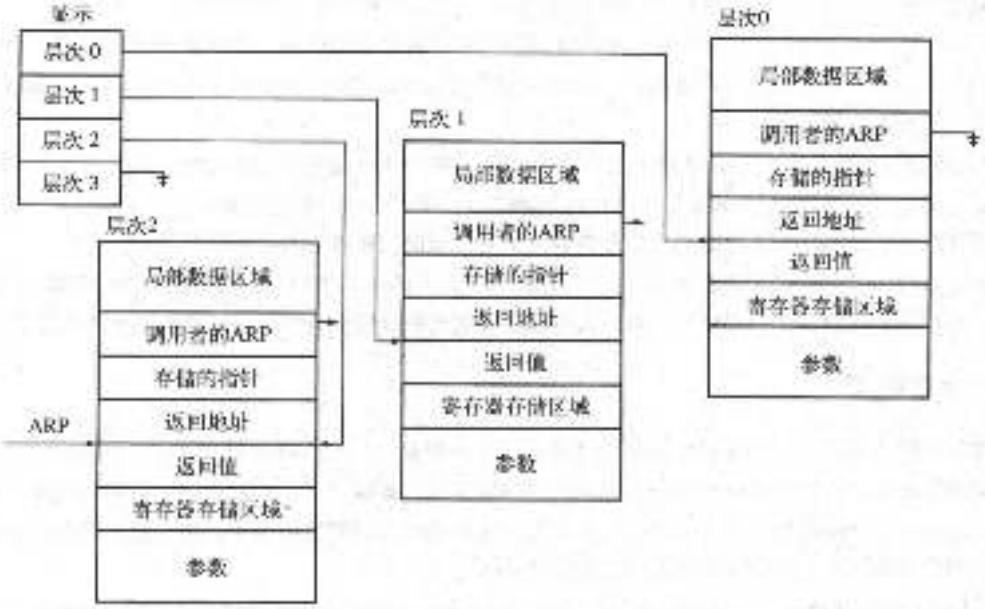


图6-10 使用全局显示

283 回到存取链的讨论中的静态坐标，下面的表给出编译器可能发行的基于显示的实现的代码。假设当前过程处于词法层次2，且标签_disp给出这一显示的地址。

<2,24>	loadAI r_{arp},24	$\Rightarrow r_2$
<1,12>	loadI _disp	$\Rightarrow r_1$
	loadAI r_1,4	$\Rightarrow r_1$
	loadAI r_1,12	$\Rightarrow r_2$
<0,16>	loadI _disp	$\Rightarrow r_1$
	loadAI r_1,16	$\Rightarrow r_2$

使用显示，非局部存取的代价是固定的。使用存取链，编译器生成$m-n$个装入的序列；使用显示，编译器把$n \times l$用作显示中的偏移，其中l是指针的长度（在这一例子中l是4）。局部存取仍然比非局部存
284 取廉价，但是使用显示，非局部存取的代价是常量，而不是变量。

当然，编译器必须插入维护这一显示所需要的代码。因此，当处于层次n的过程p调用某个处于层次$n+1$的过程q时，p的ARP成为层次n的显示入口。（q正在执行时这个入口不被使用。）保持显示最新的最简单方法是当控制进入p时，让p去更新层次n的入口，并在从p离开时恢复该入口。在入口处，p可以把层次n的显示入口拷贝到p的AR中保留的可寻址性槽中，并把它自己的ARP存储在这一显示的层次n的槽中。

当然，可以避免这些显示的大多数更新。只有过程p（直接或间接）调用的过程q可以使用由p存储的ARP，其中q嵌套于p的作用域内。因此，不调用被嵌套在其自身中的任意过程p无需更新这一显示。这消除了过程的所有更新以及其他很多更新。

3. 两种转换机制的选择

编译器只能实现这些技术中的一个。二者各有优缺点。

1）二者都增加代价。显示维护的代价是常量，即调用和返回时的一个装入和存储。存取链维护的代价是变动的，但是常见的层次n的过程调用层次$n+1$过程的情况是代价低廉的。

2）通过显示的引用有常量代价。如果非局部存取经常发生而成为一个足够关注的问题，那么显示指针可能不能放在它自己的寄存器内。然而在相同的情况下，它很可能被保留在缓冲器中，从而减小通过显示的间接寻址的实际代价。

3）通过存取链的引用导致变化的代价，因为这一链的开始端被存储在ARP中，无需寄存器指向这个链。因为编译器无法控制AR是如何映射到缓冲器的，从而遍历这个链可能导致缓冲错误。

在实践中，这两种方案之间的代价差异取决于对过程调用和返回的非局部引用的频率。

然而，对于若干情况，选择是显然的。例如，如果AR的生存期可以比创建它们的调用的生存期长，
285 那么存取链仍然有效，而全局显示无效。这使存取链成为把过程作为第一类对象的语言的共同选择。

6.6 标准链接

过程链接是编译器、操作系统和目标机器之间的一种契约，它清晰地划分命名、资源分配、可寻址性和保护等责任。过程链接保证用户代码之间的协同工作，如编译器翻译的代码，包括系统库、应用库和用其他程序设计语言写成的代码等从其他资源而来的代码之间的协同工作。用于给定的目标机器和操作系统的所有编译器一般都尽量使用相同的过程链接。

链接约定用于把每一个过程从调用过程的不同环境分离开来。假设过程p有一个整型参数x。对p的不同调用可能把x绑定到存储在调用过程的栈帧架中的一个局部变量上、绑定到一个全局变量上、绑定

到某个静态数组的一个元素上或者绑定到诸如y + 2这样的整型表达式的评估结果上。因为过程链接描述如何在调用过程中评估和存储传递给x的值，以及在被调用过程如何存取这个x，所以编译器能够给被调用过程本身生成这样的代码，这一代码忽视p的不同调用的运行时环境之间的差异。只要所有过程都遵循这一链接约定，这些细节将联合起来对由源语言描述所允许的值进行无缝传递。

当然，链接约定与机器相关。例如，链接约定隐式地取决于诸如目标机器上可用寄存器的数量等信息，而且还取决于执行调用和返回的机制。

图6-11展示标准过程链接的各部分是如何彼此相适应的。每一个过程都有一个*序言序列*（prologue sequence）和一个*结语序列*（epilogue sequence）。每一个调用地点包含一个*调用前序列*（precall sequence）和一个*返回后序列*（postreturn sequence）。

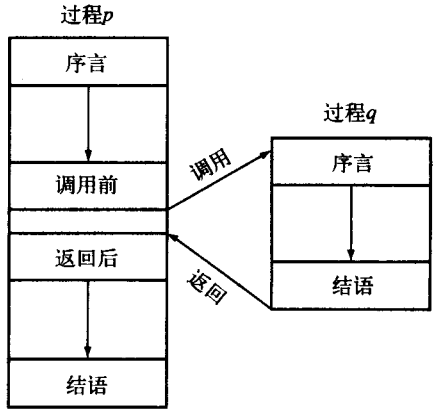


图6-11 标准过程链接

（1）调用前

调用前序列开始构造被调用环境的过程。它评估实参、决定返回地址，如果需要的话，还决定用来保存返回值的保留空间的地址。如果一个引用调用参数目前被分配给一个寄存器，那么调用前序列需要把这个参数存储到调用者的AR中，使得它可以把这一分配地址传递给被调用者。

如AR图所示的很多值都可以使用寄存器传递给被调用者。返回地址，即返回值的地址和调用者的ARP显然是这样做的候选者。前k个实参也可以使用寄存器传递，k的典型值是4。如果调用有多于k个参数，那么剩余的实参必须存储在被调用者的AR中或者调用者的AR中。

286

（2）返回后

返回后序列撤销调用前序列的动作。它必须恢复需要返回到寄存器的引用调用及值调用结果参数。它必须从寄存器存储区域恢复所有调用者保存的寄存器。它也许需要解除整个或部分被调用者的AR。

（3）序言

一个过程的序言序列完成构建被调用者运行时环境的任务。它可能在被调用者的AR中创建空间，把由调用者传递的某些值存储在寄存器中。它必须为局部变量创建空间，如需要的话，还需要初始化它们。如果被调用者引用过程固有的静态数据区，那么它也许需要把这个数据区的标签装入到一个寄存器中。

（4）结语

一个过程的结语序列开始拆除被调用者环境并重构调用者环境的过程。它可能参与解除被调用者AR的工作。如果这个过程返回一个值，结语可能负责把这一值存储到由调用者指定的地址上。（也可能是由为返回语句所生成的代码来执行这一任务。）最后，它恢复调用者的ARP并跳转到返回地址。

这是构建链接约定的一般框架。其中的很多工作可以在调用者和被调用者之间调整位置。移到序言和结语序列一般将产生更紧凑的代码。调用前和返回后序列是为每一个过程调用生成的，而序言和结语序列对每一个过程只出现一次。如果过程平均被调用超过一次，那么序言和结语序列的数量少于调用前和返回后序列的数目。

287

1. 保存寄存器

必须在调用序列的某个点处把调用者希望在调用后存活的所有寄存器值都保存在内存中。调用者可以执行这一任务；因为调用者精确地知道哪些值在调用后仍需存活，它保存的寄存器数目也许比被调者所保护的寄存器数目少。这一约定称为*调用者保存*（caller save）。被调用者也可以执行这一任务；因为被调用者精确地知道它将要使用哪些寄存器，它可能让某些值保留在它们的寄存器中而不被接触。这一

约定称为*被调用者保存*（callee save）。

两个约定都有受欢迎的理由。存储和恢复寄存器的过程可能要利用它自己的信息来避免某些存储和恢复。对于任意的特定劳动分工，我们都可以构造它适用的程序和它不适用的程序。很多现代系统采用一个折中的方案，对一部分寄存器进行调用者保存处理，而对另外的寄存器进行被调用者保存处理。在实践中，这似乎能很好地工作。它鼓励编译器把生存期长的值放置在被调用者保存的寄存器中，这样它们只在被调用者确实需要这个寄存器时才被存储。它鼓励编译器把生存期短的值放置在调用者保存寄存器中，这样可能在调用时可以避免存储这些值。

2. 分配活动记录

在大多数一般情况下，调用者和被调用者都需要存取被调用者的AR。遗憾的是，调用者一般都不知道构造多大的被调用者AR（除非编译器和链接器能够设法使链接器把适当的值粘贴到每一个调用地点。）

使用栈分配的AR时，有一个折中的方案。因为分配是由递增栈顶指针组成的，调用者可以通过跳转到栈顶并把值存储到适当的位置来开始创建被调用者的AR。当控制传递到被调用者时，它可能通过进一步递增栈顶来部分地扩展已构建的AR，从而为局部数据创建空间。在这一方案中，返回后序列可以重置栈顶指针，只用一步执行整个回收工作。

288 使用堆分配的AR时，递增地扩展被调用者的AR是不可能的。在这种情况下，调用者可以用寄存器传递AR所需的几乎所有的值；序言序列可以分配一个适当大小的AR并当需要时把值存储在其中。当过程有过多形参时，这一方案带来一个主要的复杂性问题。我们所有的图表都展示存储在被调用者AR中的实参；使用堆分配的AR，调用者可能不能把实参值保存在那里。在这一情况下，编译器设计者可能选择让调用者把过多的实参存储在自己的AR中。这消除调用者存取被调用者的AR的必要性，并使被调用者假定它全权负责分配和回收自己的AR。当然，存储在调用者AR中的参数的存取的成本会稍高一些，因为当控制在被调用者中时，调用者的ARP不总是在寄存器中。

更多关于时机的问题

在典型的系统中，在其发展的早期阶段，链接约定是要经过编译器实现者和操作系统实现者之间协商的。因此，诸如调用者保存寄存器与被调用者保存寄存器之间的差异等问题都是在设计时决定的。当编译器运行时，它必须为每一个过程发行过程序言序列和过程结语序列，并为每一个调用地点发行调用前序列和返回后序列。这些代码在运行时执行。因此，编译器不知道应该存储到被调用者AR的返回地址。（一般也不知道那个AR的地址。）然而，它却可以包含在链接时（使用可重定位汇编语言标签）或在运行时（使用距程序计数器的某个偏移）生成返回地址的机制，并把这个返回地址存储在被调用者AR中的适当位置。

同样地，在使用显示为其他过程的局部变量提供可寻址性的系统中，编译器不知道这一显示或AR的运行时地址。然而，它却可以发行维护显示的代码。实现这一目标的机制要求两样信息：当前过程的词法嵌套层次和全局显示的地址。前者在编译时可知；后者可以通过使用可重定位汇编语言标签在链接时确定下来。因此，序言可以读取进入它的AR的过程层次的当前显示入口（根据显示地址使用`loadA0`）并把它存储于该AR中（使用相对于ARP的`storeA0`）。

289

3. 管理显示和存取链

管理非局部存取的两个机制都需要在调用序列中做某些工作。使用显示，序言序列更新它自己的层次的显示记录，而结语序列则恢复显示记录。如果过程从不调用更深的嵌套过程，那么它会跳过这一步。

使用存取链，调用前序列必须定位被调用者的第一个存取链。工作量随着调用者与被调用者之间在词法层次上的差异而变化。只要被调用过程在编译时已知，两个方案都是相当高效的。如果被调用者不是已知的（例如，如果它是一个函数值参数），那么编译器也许需要发行特殊分支代码来执行适当的步骤。

6.7 管理内存

编译器设计者在实现过程时必须面对的另外一个问题是内存的管理。在大多数现代系统中，每一个程序在它自身的逻辑地址空间内执行。这一地址空间的布局、组织和管理需要编译器和操作系统之间的共同协助，以便在源语言和目标机器的规则和限制的范围内提供高效实现。

6.7.1 内存布局

编译器、操作系统和目标机器共同合作，保证多个程序可以在基于交叉存取（时间片）的平台上安全执行。很多关于如何布局、处理和管理程序地址空间的决策都超出编译器设计者的能力。然而，这些决策对必须生成的代码和这一代码的实现产生很大的影响。因此，编译器设计者必须对这些问题有广博的理解。

1. 设置运行时数据结构

在运行时，一个被编译的程序是由可执行代码和若干不同的数据范畴组成的。这一被编译的代码通常是固定大小的。某些数据区域的大小也是固定的；例如，诸如FORTRAN和C这类的语言中的全局和静态变量的数据区域在执行期间既不变大也不缩小。其他数据区域在整个执行过程中大小是变化的；例如，保存活动过程的AR的区域在程序执行时要扩大和缩小。 290

图6-12给出一个被编译程序所使用的地址空间的一个典型布局。固定大小的可执行代码位于地址空间的低端；在这一图表中被标有静态（static）的相邻区域保存静态和全局数据区域，以及某些编译器生成的数据。（依赖于程序和源语言，编译器也可能需要存储常量、开关语句的跳转表、方法表以及支持垃圾回收和调试的信息。）地址空间的其余部分都用于伴随执行扩展和压缩的数据区域；如果语言支持AR的栈分配，那么编译器需要为堆（heap）和栈（stack）两者保留空间。为了充分利用这一空间，堆和栈应该被放置在开放空间的相对两端，并可以向中间延伸。堆向着更高地址延伸；栈向较低地址延伸。

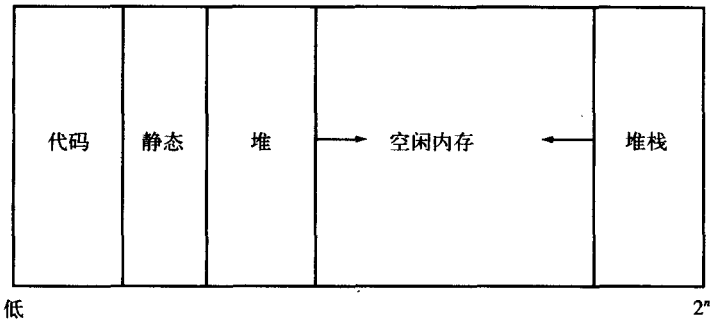


图6-12 逻辑地址空间布局

从编译器的观点看，逻辑地址空间是整个图案。然而，一般现代计算机系统以一种交叉存取形式同时执行很多程序。操作系统把若干不同的逻辑地址空间映射到由目标机器支持的单一地址空间。图6-13给出这一个更大的图案。每一个程序被分隔在它自己的逻辑地址空间内；每一个程序都能如同有自己的机器一样行动。

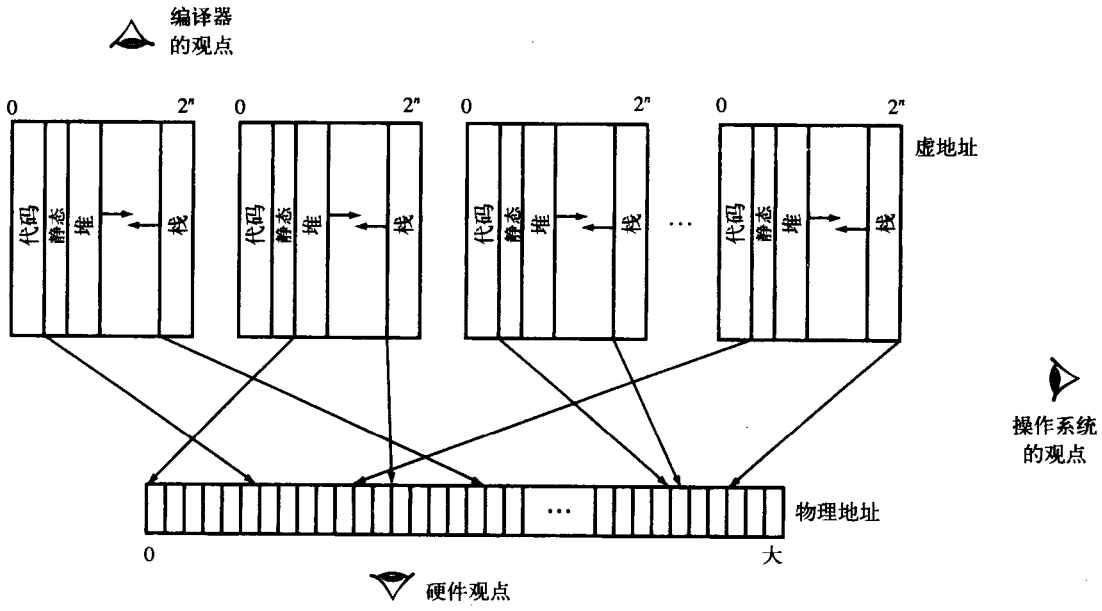


图6-13 地址空间的不同视点

单一逻辑地址空间可以跨越物理地址空间的分开的段（或页）；因此，程序的逻辑地址空间中的地址100 000和200 000在物理内存内不一定相隔100 000字节。事实上，与逻辑地址100 000相关联的物理地址可能比与逻辑地址200 000相关联的物理地址还大。从逻辑地址到物理地址的映射是由硬件和操作
291 系统共同维护的。在几乎各个方面，它都超出编译器的视野。

2. 内存模型对代码形态的影响

编译器设计者必须决定是把值尽量保存在寄存器里还是尽量保存在内存里。这一决策对编译器为各个语句所发行的代码有重要的影响。

使用内存到内存模型，编译器在目标机器上有限的寄存器单元内工作。编译器所发行的代码使用真实的寄存器名字。在语句到语句的基础上，编译器保证对寄存器的需要不超过目标机器可用的寄存器单元数。在这些情况下，寄存器分配成为改进代码的一种优化，而不是正确性所需的转换。

使用寄存器到寄存器模型，编译器假设它有一组虚拟寄存器，而不是目标机器的真实寄存器单元。这一虚拟寄存器单元大小不限。编译器把可以合法驻留在寄存器中的每个值与虚拟寄存器相关联；⊖只
292 有作为引用调用参数传送的值、作为返回值传送的值以及寄存器分配器造成溢出（参见第13章）的值才被存储到内存中。使用寄存器到寄存器内存模型，必须运行寄存器分配器来减少对寄存器的需要，并把虚拟寄存器名字映射到目标机器寄存器名字上。

3. 调整和填充

目标机器对数据项的存储位置有特殊的要求。一组典型的限制可以描述开始于字（32位）边界的32位整数和32位浮点数，开始于双字（64位）边界的64位浮点数据，以及开始于半字（16位）边界的串数据。我们称这些是*调整规则*（alignment rule）。

一些机器有一个用于实现过程调用的特殊指令；这一特殊指令可能保存寄存器或存储返回地址。这样的支持将进一步增加调整限制；例如，这一指令可能指定AR的某些部分的格式，并为每个AR的开始

⊖ 一般仅能用一个名字存取的值可以被保存在寄存器中。我们称这样的值为*非歧义值*（unambiguous value）。

点增加一个调整规则。DEC VAX计算机有一个独特的调用指令；这一指令自动存储寄存器和处理器状态的其他部分。⊖

为了遵守目标机器的调整规则，编译器也许需要浪费一些空间。为了在一个数据域内指定位置，编译器应该把变量编排成组，从带有最多的限制的变量到带有最少限制的变量（例如，双字调整的限制比全字调整的限制更多。）汇编程序一般有一个保证装入器开始于给定调整的数据区域，例如双字边界的命令。开始于这样的边界，编译器能够先给最严格的范畴内的所有变量赋偏移，然后给次严格类内的所有变量赋偏移，以此类推，直到所有变量都被赋偏移为止。因为调整规则几乎总是2的幂，每个范畴的末端自然适合下一个范畴的限制。

4. 相对偏移和高速缓冲存储器的性能

在现代计算机系统中高速缓冲存储器的广泛使用对内存中变量的布局有微妙的影响。如果两个值在代码中的使用很接近，那么编译器将倾向于保证它们在同一时间内驻留在高速缓冲存储器中。这可以用以下两个方法实现。 293

高速缓冲内存基础

计算机“建筑师”设法在处理器速度与内存速度之间的差距间搭设桥梁的一个方法是*高速缓冲存储器*（cache memory）的使用。高速缓冲存储器是置于处理器与主存储器之间的一个小的高速内存。高速缓冲存储器被分成一系列相同大小的*帧*（frame）。每个帧有一个地址域，称为它的*标签*（tag），这一地址域保存主存储器的地址。

硬件自动把内存位置映射到缓冲帧上。用于直接映射的高速缓冲存储器的最简单的映射把高速缓冲地址计算为主存储器地址模高速缓冲存储器的大小。这把内存分隔成一组线性块，每个块的大小是缓冲帧的大小。*一行*（line）是映射到一个帧上的一个内存块。在任意时刻，每个缓冲帧保存一块数据的一个拷贝。它的标签域保存数据所在内存的地址。

在每一次对内存的读存取中，硬件要查看被请求的字是否已在它的缓冲帧里。如果是这样的话，被请求的这些字节返回到处理器中。否则，在这一帧中的块被驱除，同时被请求的块被带入到这一缓冲存储器中。

有些高速缓冲存储器使用更复杂的映射。*集合关联*（set-associative）高速缓冲存储器对每个缓冲行使用多个帧，每一个行一般使用两到四个帧。*完整关联*（fully associative）缓冲存储器可以把任意一个块放置到任意帧中。这些方案都使用标签上的关联搜索来确定一个块是否在这一缓冲器内。关联方案使用一种策略来确定要驱除哪个块；常见的方案是随机置换和最近最少使用（least-recently-used，LRU）置换。

在实践中，高效的内存速度是由内存带宽、缓冲块长度、高速缓冲的速度与内存速度的比率和缓冲器内的存取击中率决定的。从编译器的角度看，前三个是固定的。以编译器为基础的对改进内存性能的努力集中在增加击中率。

一些体系结构提供一些指令，这些指令允许程序给高速缓冲提供线索：什么时候特定块应被放入内存中（*预取*），什么时候它们不再被需要（*清洗*）。 294

最好的情况是这两个值共享一个缓冲块，这保证可以一起将这两个值从内存提取到缓冲器中。如果它们不能共享一个缓冲块，那么编译器将倾向于保证这两个值映射到不同的缓冲行。通过控制这两个值

⊖ 对于每一个过程，编译器决定必须保存哪些寄存器和状态位。编译器把这一信息编码到一个位掩码并存储于在过程序言的前面。过程调用指令重新得到这个掩码并解释它，从而保存指定的寄存器和状态位。

的地址间的距离，编译器可以做到这一点。

如果我们只考虑两个变量，控制它们地址之间的距离似乎容易处理。然而，当考虑所有活动变量时，缓冲器的最优化布局问题是NP完全的。大多数变量都与其他变量有交互作用；这创建一个编译器无法同时满足的关系网络。如果我们考虑使用若干大数组的循环，解决互不干涉问题甚至会变得更糟。如果编译器能够发现这一循环中不同数组引用之间的关系，那么编译器可以在数组之间增加填充来增加引用击中不同缓冲行的可能性，因此也就不互相干涉。

正如我们在前面所看到的那样，程序的逻辑地址空间到硬件的物理地址空间的映射不一定保持特定变量之间的距离。作为这一想法的逻辑结论，读者也许会问编译器是如何保证与超出虚拟内存页的大小的相对偏移相关的所有事情的。处理器的物理高速缓冲器既可能使用虚设地址又可能使用它的标签域中的物理地址。虚拟地址高速缓冲器保持编译器创建的值之间的空间；使用这样的高速缓冲器，编译器也许能够设法使较大对象之间互不干涉。使用物理地址高速缓冲器，不同页中的两个位置之间的距离是由页映射决定的（除非高速缓冲器的大小不超过页的大小）。因此，编译器有关内存布局的决策很少产生效应，除非它们是在一个页内。在这样的情况下，编译器应该致力于使一起被引用的对象在相同页上，而且如有可能，使它们在相同的高速缓冲块中。

6.7.2 堆管理算法

很多程序设计语言处理动态创建和销毁的对象。编译器通常不能决定这样的对象的大小和生存期。为了处理这样的对象，编译器和操作系统创建一个动态可分配存储池，这一存储池一般被称为*运行时堆*（run-time heap）或堆。在堆的创建和管理中出现很多问题；其中一些问题能够被源语言程序员观察到，而另外一些问题只有系统软件的设计者才能够观察到。

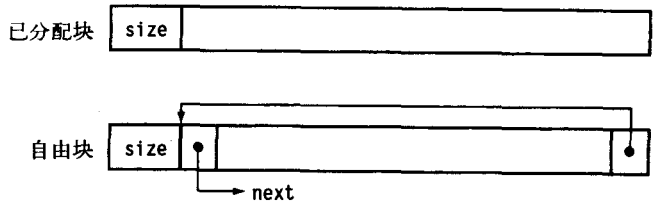
295 本节主要揭示用于堆管理的一些算法。本节考虑一个显式可管理堆的情况，在这一情况中，程序员必须分配并释放空间。下一节考虑执行隐式释放的算法。

我们假设堆有简单的接口，即例程`allocate(size)`和`free(address)`。`allocate`例程取整数参数`size`并返回包含至少`size`字节的堆中的一个空间块的地址。`free`例程取堆中在前面所分配的空间块的地址，并把这一地址返回到自由空间池。

显式堆管理的算法设计中所引发的关键问题是`allocate`和`free`的速度以及自由空间池破碎成小块的程度。我们首先考虑简单的分配模型：*首次拟合分配*（first-fit allocation）。

1. 首次拟合分配

首次拟合分配器的目标是快速分配和释放堆中的空间。首次拟合方案强调的不是内存利用而是速度。堆中的每一个块都有一个保存其大小的隐藏域，作为簿记的负荷。容量字段一般被定位于`allocate`返回的地址的前面的字中。可分配块居于称为*自由列表*（free list）的列表中。在必需的容量字段之外，自由列表中的每个块有指向自由列表中下一个块的指针（或在最后一个块的空指针）和位于这个块的最后字中的指向这个块本身的指针。



堆的初始条件是被置放在自由列表中的一个大块，以及包含指向这个块的开始的最后字和一个空的`next`域。

调用allocate(k)引发下面一系列事件。allocate例程遍历自由列表直到它发现一个大小大于或等于k加上size域的一个字的大小的块。假设allocate发现一个适当的块b_i。它把b_i从自由列表中移出。如果b_i比需要的大，那么allocate在b_i的末端剩余空间中创建一个新的自由块，并且把这一新的自由块放置在自由列表中。allocate例程返回一个指向b_i第二个字的指针。 296

如果allocate没有发现足够大的块，那么它尝试着扩充这个堆。如果它成功地扩充了堆，那么它从这个堆新分配的部分返回适当大小的块。如果扩充堆失败，那么它报告错误（一般通过返回一个空指针）。

为了释放一个块，程序对这个块b_j的地址调用free。free最简单的实现是把b_j加到自由列表的头部并返回。这生成一个快速的free例程。遗憾的是，这一做法导致在反复使用中把内存分成小块的分配器。

为了克服这一缺点，分配器可以在自由块的末端使用一个指针来连结邻接的自由块。free例程装入b_j的容量字段前面的字，这个字是内存中b_j的直接前驱块的块尾指针。如果这个字包含一个有效指针，而它指出一个匹配的块的首部（其地址加上容量字段指向b_j的开始的块），那么b_j和它的前驱都是自由的。free例程可以通过增加前驱的容量字段并在b_j的最后字中存储适当指针来把二者组合起来。把新的块连结到它的前驱上从而避免更新自由列表。

为了使这一方案运作起来，allocate和free必须维护块尾指针。每当free处理一个块时，它必须用这一块的头部地址更新块尾指针。allocate例程必须或者使next指针无效，或者使块结束指针无效，这样来阻止free把自由块与已分配块组合起来，这样的已分配块的这些域没有被复写。

free例程也可尝试把b_j与内存中它的后继b_k组合起来。free可以使用b_j的容量字段来定位b_k的开始。它还可以使用b_k的容量字段和块尾指针确定b_k是否是自由的。如果b_k是自由的，那么free可以把这两个块组合起来，从自由列表中移出b_k且把b_j加入自由列表中。为了使这一自由列表的更新高效，自由列表需要双重链接。因为，指针被存储在未分配块中，所以空间负荷是无关紧要的。更新双重链接自由列表所需要的额外的时间很小。

如上所述，这一组合方案依赖于这样的事实：自由块中的最后的指针与容量字段之间的关系在已分配块中不复存在。尽管分配器几乎是不可能把一个已分配块看成是自由的，但是这还是可能发生的。为了防止这一未必发生的事件出现，实现者可以使块尾指针既存在于已分配块又存在于自由块的一个域中。在分配时，这一指针设置为堆外面的一个地址，例如0。在释放时，这一指针被设置成这个块本身的地址。这一附加保障的代价是在每一个已分配块中有一个额外的域，且对每一次分配有一个额外的存储。

人们尝试了许多首次拟合分配的变形。这些尝试在allocate的代价、free的代价、由一系列分配所产生的碎片总量和返回比需要更大的空间所浪费的空间总量之间进行权衡。 297

基于实存块的分配

在编译器内部，编译器设计者也许发现使用一个特化了的分配器是可行的。编译器有面向阶段的活动。这非常适合于基于实存块的分配方案。

使用基于实存块的分配器，程序将在一个活动开始时创建一个实存块。这一程序使用这个实存块保存在使用上相关的已分配对象。在实存块中的分配对象的调用满足栈风格；一个分配包含递增指向实存块高水标的指针并返回指向新已分配块的指针。没有用于释放各个对象的调用；当包含这些对象的实存块被释放时，这些对象被释放。

基于实存块的分配器是传统的分配器和垃圾回收分配器之间的一种折中。使用基于实存块的分配器，对allocate的调用变得无足轻重（正如在现代分配器中那样）。不需要释放调用；当程序完成创建实存块的活动时，程序在一个调用下释放整个实存块。

2. 多池分配器

现代分配器使用一种简单的技术，它得自于首次拟合分配，及一组关于程序行为的观察。随着20世纪80年代早期的内存增长，如果浪费一些空间可以加快分配，那么这样做也变得合理。同时对程序行为的研究表明，实际的程序频繁地分配若干标准大小的内存，而不常分配大内存，也不常分配不常见大小的内存。

现代分配器对若干标准尺寸使用独立的内存池。所选的大小一般是2的幂，从小的块尺寸（例如16字节）到虚拟内存页的尺寸（一般是4096字节或8192字节）。每个池仅有一种块尺寸，所以allocate可以返回适当自由列表的第一个块，而free可以单纯地把块加到这个适当自由列表的头部。对于大于页尺寸的要求使用独立的首次拟合分配器。基于这些思想的分配器很快。这些分配器特别适用活动记录的堆分配。

298 这些改变简化allocate和free。allocate例程必须对空的自由列表进行检查，并且当它为空时对相应的池增加一个页。free例程仅把块插入相应尺寸的自由列表的头部。通过对照为每一个池所分配的内存区间检查自由块的地址巧妙的实现可以确定这一自由块的尺寸。另一个方案就是同前面一样使用容量字段，并且如果分配器把一个页上的所有存储都放入到一个池内，那么就把一个页中的块的尺寸存储在这个页的第一个字里。

3. 调试帮助

使用显式分配和释放编写的程序非常不容易调试。这表现为程序员很难决定什么时候释放堆分配的对象。如果分配器能够很快地区分已分配对象和自由对象的话，那么堆管理软件可以为程序员的调试提供某些帮助。

例如，为了组合相邻的自由块，分配器需要一个从一个块的尾端回到它的头部的指针。如果一个已分配块使这个指针设置到一个无效值，那么释放例程可以检查这个域，并当程序试图释放自由块或一个非法地址时，即指向某个已分配块的开始之外的其他任意事物时，释放例程报告一个运行时错误。

在适度的附加代价下，堆管理软件可以提供额外的帮助。通过把已分配块链接到一起，分配器可以创建一个内存分配调试环境。快照工具可以漫游已分配块列表。通过使用创建块的调用地点为这些块附标签使得这一工具可以揭示出内存泄漏。给已分配块打上时间印可以为程序员提供关于内存使用的详细信息。这种类型的工具可以对定位从不释放的块提供很大的帮助。

6.7.3 隐式释放

很多程序设计语言允许实现在内存对象不再使用时自动释放它们。这要求对分配器和编译代码的实现格外小心。为了执行隐式释放，即垃圾回收（garbage collection），编译器和运行时系统必须包含确定什么时候一个对象不再有用或已经死亡的机制，还必须包含回收和再利用死亡空间的机制。

299 垃圾回收的相关工作可以递增地施加于各个语句，也可以当自由空间池耗尽时由按面向批处理的任务要求来实施。引用计数（reference counting）是执行递增垃圾回收的一个经典方法。标记清理回收（mark-sweep collection）则是执行面向批处理回收的经典方法。

1. 引用计数

这一技术给每一个堆分配对象增设一个计数器。计数器追踪引用该对象的未完结指针的数目。当分配器创建该对象时，它将其引用计数设置为1。对一个指针变量的每一次赋值都调整两个引用计数。它递减指针的赋值前的值的引用计数，而递增指针的赋值后的值的引用计数。当一个对象的引用计数降到0时，不存在指向这一对象的指针，所以系统可以安全地释放这一对象。释放一个对象也许又会废弃指向其他对象的指针。这必定递减那些对象的引用计数。因此，废弃指向一个抽象语法树的最后的指针将

释放整棵树。当根结点的引用计数降到0时，它被释放且它的子孙的引用计数也被减少。因此，这将释放这些子孙，减少它们的子结点的引用计数。这一过程一直继续下去直到整个AST被释放。

由于在已分配对象中存在指针，这给引用计数方案带来如下所示的若干问题：

1）运行代码需要区分指针与其他数据的机制。这一机制或者为每一个对象把额外的信息存储于它的头部域，或者把指针限定到小于一个全字的范围，并使用空余的位“标签”这一指针。批处理回收器也面对着同样的问题，而且使用相同的解决方案。

2）每次减小引用计数所做的工作量可能相当大。如果外部限制要求有界的释放时间，那么运行时系统可以采用更复杂的协议来限制为每一个指针赋值时释放的对象的数量。通过维护必须释放对象的队列，并限制对每一次引用计数调整所做的处理数量，运行时系统可以在较大的操作集合上分布释放对象的工作。这把所有赋值的集合上的释放代价摊派到堆分配对象上，并限制每一次赋值时所做的工作。

3）程序可能形成指针的有环图。有环数据结构的引用计数不能减小到0。当最后的外部指针被废弃 300
后，这一环变得不可达且不可再利用。为了保证释放这样的对象，程序员必须在废弃指向环的最后指针之前切断这一环。（另外一个选择是在运行时对这些指针执行可达性分析，这将使引用计数的代价过于昂贵。）很多堆分配对象的范畴，如可变长度串和活动记录等都不能包含这样的环。

引用计数在每一个指针赋值上带来额外的代价。可以限制为特定赋值所做的工作量；在任意良好的设计中，总代价可以被限制到某个常量因子乘以被执行的指针赋值数量，加上分配对象的数量。引用计数的支持者们认为这些负荷是足够小的，而且认为引用计数体系中的复用模式产生良好的程序局部性。引用计数的反对者们认为实际程序比分配要做更多的赋值，因此垃圾回收对较少的工作总量要做较多的工作。

2. 批回收器

仅当自由空间池被耗尽时，批回收器才考虑释放。当分配器找不到所需要的空间时，它调用批回收器。回收器中止程序的执行，检查已分配内存池来发现不被使用的对象并回收这些对象的空间。当回收器终止时，自由空间池是非空的。[⊖]分配器可以完成它本身的工作并把新分配的对象返回给调用者。（在同引用计数一样，可以执行增量回收把代价摊派到更长的执行期间。）

批回收器逻辑上有两个阶段。第一个阶段发现如下这样的一组对象：从存储于程序变量中的指针和编译器生成的临时变量可达的对象。回收器适宜地假设以这种方式可达的任意对象是活的，而其余对象是死的。第二个阶段释放并再利用死对象。两个常用的技术是标记清理（mark-sweep）回收器和复制（copying）回收器。它们的差异在于回收的第二个阶段，即再利用的实现不同。 301

3. 识别活数据

回收分配器通过使用标记算法发现活对象。对堆中的每一个对象回收器需要一位，称为标记位（mark bit）。这个位和用于记录指针位置或对象大小信息的标签信息一同存储在对象的头部。另外，必要时回收器也可以为堆创建一个稠密的位映射。这一算法的第一步清除所有标记位并构建一个工作表（worklist），所有工作表包含存储于寄存器的指针及对应于当前或等待过程的活动记录中的指针。这一算法的第二个阶段从这些指针开始向前遍历，并标记出从这一组可见指针可达的每一个对象。

图6-14展示标记算法的一个高级概略。这是一个简单的不动点计算，因为堆是有限的而且标记阻止堆内的指针多次进入工作表（*Worklist*），因此算法终止。在最坏的情况下，标记的代价与包含在程序变量中的指针和临时变量数目加上堆的大小成正比。

⊖ 除非所有空间都已被使用。在这样的情况下，分配器设法从操作系统得到额外的空间，它把这一空间用作自由空间池。如果没有可用的额外的空间，分配失败。

```
清理所有标记
Worklist ← { pointer values from activation records & registers }
while (Worklist ≠ ∅)
   remove p from the Worklist
   if (p→object is unmarked)
      mark p→object
      add pointers from p→object to Worklist
```

图6-14 一个简单的标记算法

标记算法既可以是精确的，又可以是保守的。其差异在于算法如何确定*while*循环的最后一行中的特定数据值是指针。

- 在精确回收器中，编译器和运行系统了解每一个对象的类型和布局。这一信息可以记录在对象的头部，或者可以从类型系统间接地得知这一信息。无论是哪一种方法，使用精确的知识，标记阶段只跟踪实际的指针。
- 在保守的标记阶段，编译器和运行时系统不能肯定某些对象的类型和布局。因此，当一个对象被
302 标记时，运行时系统考虑可能是指针的每一个域。如果它的值是一个指针，那么它就被当作指针处理。不表示字对准地址的任意值都排除在外，因为它可能是落于堆的已知界线外面的值。

保守分配器具有局限性。它们无法回收精确回收器能够发现的某些对象。如果程序可以隐藏一个指针使标记阶段找不到它，那么这一回收器可能不正确地释放已分配的对象。但是，经过成功的改进，保守回收器已经被实现于诸如C语言等通常不支持垃圾回收的语言中。

当标记算法停止时，任何没有被标记的对象一定是程序无法达到的对象。因此，回收器的第二个阶段可以把这样的对象作为死对象处理。某些被标记为活对象也可能是死的。然而，回收器让它们活着，因为回收器无法证明它们是死的。当第二个阶段遍历堆来回收垃圾时，它可以把标记域重新设置为“不被标记”。这使得回收器避免在标记阶段对堆的初始遍历。

4. 标记清理回收器

标记清理（mark-sweep）回收器通过对堆进行线性扫描回收和再利用对象。回收器把每一未被标记对象加到自由列表中（或多个自由列表中的一个列表中），在这里分配器可以找到这个对象并复用它。使用一个自由列表时，可以运用首次拟合分配器中用于组合块的技术。如果希望压缩，那么通过在清理期间递增地向下移动活对象来实现压缩，也可以在清理后使用一个压缩遍来实现压缩。

5. 拷贝回收器

拷贝回收器把内存分成两个池，一个*旧*（old）池和一个*新*（new）池。分配器总是从旧池开始操作。拷贝回收器的最简单类型称为*停机拷贝*（stop and copy）回收器。当一次分配失败时，停机拷贝回收器把所有活数据从旧池拷贝到新池中，并交换新旧池的身份。拷贝活数据的动作压缩这一数据；回收后，整个自由空间在单一的连续块中。回收进行两个阶段的工作，与标记清除一样，它也可以在发现活数据
303 时增量地进行回收。一个增量方案是在拷贝旧池对象时对其做标记以避免把同一个对象拷贝多次。

拷贝回收器的一个重要的系列是*世代回收器*（generational collector）。这些回收器利用这样的观察：在一次回收中活着的对象很可能在后续很多回收中仍然活着。为了利用这一观察，世代回收器通过重新分割新、旧池中的自由空间使得后继回收只检查新的已分配对象。不同的世代方案在声明新的世代、冻结活着的对象以免除下一次回收，以及是否要周期性地再检查较旧世代等工作的时机上存在差异。

6. 各技术的比较

垃圾回收把程序员从对什么时候释放内存的担忧中解放出来，把他们从由于显式地管理分配和释放的工作所产生的不可避免的存储泄漏的跟踪中解放出来。各个方案都有它们的优点和缺点。对于大多数

运用来说，在实践中隐式释放各个方案的益处都超过任何方案的缺点。

引用计数比批回收更均匀地摊派回收代价于整个程序执行期间。然而，即使程序从不用尽自由空间，它也会增加涉及堆分配值的每一个赋值的代价。与此相对，在分配器能够找到所需空间之前，批回收器都不会带来任何代价。然而在这一点上，在回收过程中，程序要承受整个回收代价。因此，每一次分配都可能激活一次回收。

标记清理回收器检查整个堆，而拷贝回收器只检查活数据。拷贝回收器实际上移动每一个活的对象，而标记回收器却把活着的对象留在原来的位置上。这些代价之间的权衡随着应用的行为而变化，也随着各种内存引用的实际代价而变化。

引用计数的实现和保守批回收器在识别环结构上存在问题，因为它们不能区分环内引用和非环内引用之间的差异。标记清理回收器从一组外部指针开始，所以它们能够发现一个死的环结构是不可达的。开始于同样一组指针的拷贝回收器无法仅拷贝与环相关的对象。

拷贝回收器作为过程的一个自然部分压缩内存。回收器可以或者更新所有已存储的指针，或者要求为每一个对象的存取使用一个间接表。精确标记清理回收器也可以压缩内存。回收器可以把对象从内存的一端移到内存另一端的自由空间里。同样，这一回收器可以或者重写现存的指针，或者要求使用一个间接表。

304

好的实现器一般可以使得标记清理回收器和拷贝回收器都足够优秀，以至于对于大多数应用来说二者都是可接受的。在不能忍受不可预知负荷的应用中，例如实时控制器中，回收器必须以类似于摊派引用计数方案的模式增量地实施回收过程。这样的回收器被称为*实时回收器*（real-time collector）。

6.8 概括和展望

继汇编语言之后，基础理论提供了更抽象的程序设计模型，从而提高程序员的生产力和程序的可理解性。添加到程序设计语言中的每一个抽象要求在执行前翻译成目标机器的指令集合的翻译技术。本章揭示了翻译某些抽象的若干常用技术。

过程化程序设计发明于程序设计历史的早期。若干早期过程是早期计算机的调试例程；使用这些预先写好的例程，程序员能够了解错误程序的运行时状态。如果没有这样的例程，那么我们现在认为理所当然的任务，诸如变量内容检查或调用栈的踪迹等任务都要求程序员无误地键入长长的机器语言序列。

诸如Algol 60等语言中的词法作用域的引入对语言设计产生了长期的影响。大多数现代程序设计语言把Algol的某些思想带入命名和可寻址性。诸如存取链和显示等20世纪60年代和20世纪70年代发展起来的技术减少了这一抽象的运行时代价。这些技术至今仍在使用。

面向对象语言采用类Algol语言的作用域概念，并以数据制导方式对这些概念重定位。面向对象语言的编译器运用为词法作用域而发明的编译时和运行时结构来实现特定程序的继承层次带来的命名规则。

现代语言已添加了一些新手法。通过使过程成为第一类对象，诸如Scheme等语言创建了新的控制流范型。这些都要求对传统的实现技术做一些改革，例如活动记录的堆分配。同样地，隐式回收被人们日益接受，这要求对指针的暂时性保守处理。如果编译器能够更小心地运行并把程序员从以往的释放存储中再一次解放出来，这似乎是一个很好的折中方案。（长期的经验表明，程序员并不能很好地释放他们分配的所有存储。他们还释放保留着指针的对象。）

305

当新的程序设计范例开始流行时，这些范例也将引入要求更精密的思想和实现的新抽象。通过研究以往成功的技术并理解在真实实现中所涉及的代价和限制，编译器设计者可以开发通过使用高级抽象降低运行时损失的策略。

本章注释

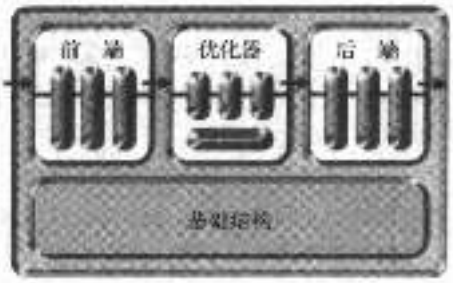
本章中的很多资料来自于编译器构造团队积累起来的经验。学习更多有关各种语言的名字空间结构的最好方法就是参考语言定义本身。这些文献是编译器设计者的必要藏书。

过程出现在最早的高级语言中，即出现在比汇编语言更抽象的语言中。FORTRAN [26]和Algol 60 [265]的过程都带有现代语言拥有的大多数性质。出现在20世纪60年代末的面向对象语言SIMULA 67 [269]被其后的Smalltalk 72 [223]所继承。

词法作用域是在Algol 60中引入的而且一直沿用至今。早期的Algol编译器引入了本章描述的很多支持机制，其中包括活动记录、存取链和参数传递技术。6.3节到6.6节的大部分素材都出现在这些早期的系统中[282]。优化迅速出现，如把块级作用域的存储叠入过程的活动记录中。早期的IBM 370链接约定考虑了叶过程和其他过程之间的差异；它们避免了为叶例程分配寄存器保存区。Murtagh采用了更完备、更系统化的方法来合并活动记录[264]。

内存布局的经典参考文献以及内存分配方案的经济学来自于Kunth的《计算机程序设计艺术（Art of Computer Programming）》[220，2.5节]。构建在常用大小的池上的现代分配器出现于20世纪80年代初期。

引用计数要追溯到20世纪60年代初，而且已被用于很多系统[90，117]。Cohen和后来的Wilson给出有关垃圾回收的文献的广泛概览[87，332]。保守回收器是由Boehm和Weiser引入的概念[44，112，42]。拷贝回收器是为回应虚拟内存系统而出现的[137，74]；它们在某种程度上很自然地导致今天广泛使用的
306 世代回收器 [238，324]。Hanson引入了基于实存块的分配的概念[171]。



第7章
代码形态

7.1 概述

在实践中，编译器能够在给定的目标机器上以多种方式实现某些源语言的构造。这些变形使用不同的操作和方法。其中有些实现比另外一些要快；有些使用较少内存；有些使用较少寄存器；有些在执行期间可能消耗较少能量。我们认为这些差异是代码形态（code shape）的问题。

代码形态对编译代码的行为、优化器的能力以及改进代码形态的后端有着很强的影响。例如，考虑C语言编译器实现通过单字节字符值进行切换的switch语句的方式。编译器可能会使用一系列级联if-then-else语句来实现这一switch语句，依据测试的布局，这将产生不同的结果。如果第一次测试是0，第二次测试是1，以此类推，那么这一方法将退化成一个256个关键字域上的线性搜索。如果字符是均匀分布的，那么字符搜索对每一个字符将需要平均128次测试和分支。这是实现选择语句的最昂贵方法。相反，如果测试执行二分搜索，那么平均情况将包含八次测试和分支，这个数字显然更好。为了以数据空间为代价提高速度，C语言编译器可以构造一个256个标签的表格，而且通过装入相应的表格项并跳转到这一表项来解释字符，那么对每个字符都有常量负荷。

所有这一切都是switch语句的合法实现。对特殊switch语句决定哪一个实现有意义要依赖于很多因素。特别地，各个情况的数量以及它们相对的执行频率非常重要，同样目标机器上的分支的代价结构的细节信息也非常重要。即使编译器无法得到它做出最好的选择所需要的信息，它也必须做出选择。可能实现之间的差异和编译器的选择是代码形态的问题。

作为另一个例子，考虑简单表达式x+y+z，其中x、y和z是整数。图7-1给出实现这一表达式的几个方法。在源代码形式中，我们可能把操作思成一个三元加法。如左边所示。然而，把这一理想化的操作映射到一系列二元加法揭示出评估顺序的影响。右边所示的三个译本给出三种可能的评估顺序，包括三地址代码和抽象语法树。（我们假设每个变量都在适当的命名寄存器中，而且假设源语言不给这样的表达式指定评估顺序。）因为整数加法满足交换律和结合律，因此这三个顺序是等价的；编译器必须选择其一来实现。

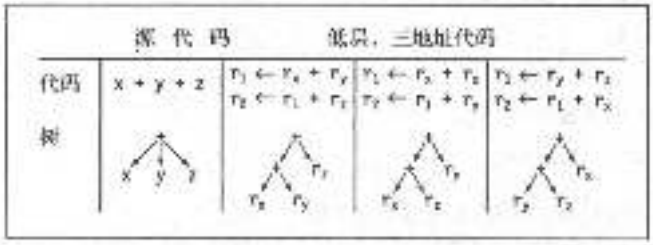


图7-1 x+y+z的不同代码形态

左结合将生成第一棵二叉树。由于左结合对应于我们从左到右的阅读风格，这棵树似乎很“自然”。

如果我们用文字常数2代替y而用3代替z，将发生什么呢？当然，x+2+3等价于x+5。编译器应该发现2+3
308 的计算，评估它并把结果直接叠入代码中。然而，在左结合形式中，2+3从不出现。顺序x+y+z把2+3隐藏起来。右结合译本揭示出改进的机会。然而对每一种可能的树，都存在使结果表达式不适于优化的x、y和z的变量和常量赋值。

同switch语句一样，在没有关于这一表达式所在的上下文的信息的条件下，这一表达式的最佳形态是不可知的。例如，如果表达式x+y最近已被计算，x的值和y的值都没有发生变化，那么使用最左边的形态将使编译器用对之前计算结果值的引用取代第一个操作$r_1 \leftarrow r_x + r_y$。在这种情况下，这三个评估顺序中的最佳选择可能依赖于这一代码周围的上下文。

本章探索在实现很多常用的源语言结构中引发的代码形态问题。这一章集中讨论为特定结构所生成的代码的形态问题，而在很大程度上忽视选出特定汇编语言指令所需的算法。指令筛选、寄存器分配和指令调度等问题将在后面章节中分别讨论。

7.2 指定存储位置

一个过程可能计算很多值。其中的一些值在源代码中具有名字；在类Algol语言中，程序员给每个变量提供一个名字。另外一些值没有明确的名字，例如表达式A[i-3, j+2]中的值i-3就没有名字。命名值具有固定的生存期。它们可能被其他过程修改。它们在调试器中也许是可视的。这些事实限制编译器放置它们的位置以及保存它们的期间。相反，编译器在处理未命名值，如i-3时却有更多的自由。编译器必须按照与程序的意义一致的方法处理它们，但是编译器在决定它们的驻留地和存放期间上却有相当大的回旋余地。

编译器对于命名值和非命名值的决策对它生成的最终代码有着很大的影响。特别地，关于非命名值的决策决定优化器可以分析和转换的值的集合。在为每一个值选择存储位置时，编译器必须遵守源语言和目标机器的内存层次的规则。编译器一般可以把标量值放到寄存器或内存中。可用内存可能被分割成
309 不同的子区域或数据区域，如第6章所述。

7.2.1 布局数据区

为了给类Algol语言中的变量指定存储类，编译器可能要使用类似于下面的规则：

```
若x在过程p中被局部声明，且
    它的值在p的不同调用间不被保存
        则将其赋值到过程局部存储空间
    若它的值在p的不同调用间被保存
        则将其赋值到过程静态存储空间
若x被声明为全局可视
    则将其赋值到全局存储空间
若x是在程序的控制下分配的
    则将其赋值到运行时堆
```

面向对象语言所遵循的规则与此不同，但复杂程度相似。

每个数据区都有其自身的限制。编译器可以把过程局部存储空间放置在这一过程的活动记录中，这是因为一个过程局部变量的生存期与这个过程的一次调用的生存期相匹配。因为代码总是需要寄存器中的ARP，这导致使用像loadAO和loadAI等操作的高效存取。频繁对AR的存取导致将其保留在高速缓冲

器中。

静态数据区和全局数据区将被存储在过程AR的外部，因为它们的生存期可能超过这次调用，直到程序的全部执行。这可能需要额外的loadI把可重定位符号（汇编语言的标签）的运行时地址引入到寄存器中来作为基地址使用。

存储在堆中的值具有编译器无法预测的生存期。可以通过两个不同机制把值放入堆中。程序员可以明确地分配堆的存储；编译器不能不考虑这一决定。当编译器发现一个值可能比创建它的过程生存期长时，它可以把这个值放入堆中。无论哪种情况，堆中的值是通过完整的地址来表示，而不是通过偏移量来表示的。

7.2.2 把值保存在寄存器中

除了给每一个值赋一个数据区和位置之外，编译器必须确定能否安全地把这个值保留在寄存器中。如果这个值可以安全地驻留在一个寄存器中，而且某个寄存器在这个值的整个生存期都是可用的，那么 310
这个值在内存中不需要空间。很多编译器给可以合法驻留在寄存器内的每个值指定一个*虚拟寄存器*(virtual register)，并依靠寄存器分配器把虚拟寄存器映射到目标机器的物理寄存器上。

如果编译器使用虚拟寄存器，那么它给每一个值或者指定一个虚拟寄存器或者指定一个内存位置，但是不能二者兼有。当分配器决定它不能把某个虚拟寄存器保留在寄存器集合中时，分配器把那个值存储到为溢出寄存器而保留的内存区域的空间中。分配器在这个值的每一次使用插入一个装入指令把这值移到一个临时寄存器，并且在每一次定义之后插入一个存储指令更新内存中的这个值。如果分配器把一个静态值保留在寄存器中，那么分配器必须在过程中第一次使用这个值之前装入这个值，并在离开这个过程（以上两个任务一个是在一个过程的出口，另一个在调用地点完成）之前把这个值存放回内存中。

为了确定是否将一个值保留在寄存器中，编译器必须知道代码可以通过多少个不同的名字来存取这个值。例如，一个局部变量将保存在寄存器中，只要从不提取它的地址、它的值从不用于嵌套的过程中而且它从不作为引用调用参数被传递给另一个过程。这些动作的每一个都为存取这一变量创建第二条路径。考虑下面C语言的代码片段：

```
void fee() {
  int a, *b;
    ...
  b = &a;
    ...
}
```

赋值b=&a创建代码可以用于存取a的另一个名字。编译器必须给a指定一个存储位置；否则编译器不能有意义地对a运用取地址操作符。然而，编译器甚至不能在a的连续引用之间把a的值保留在寄存器中，除非它能够证明这一期间代码不能给*b赋值。同样地，编译器也不能在引用之间把*b的值保留在寄存器中，除非它能证明这一期间代码不能给a赋值。

两个引用之间的代码一般可以包含其他基于指针的赋值或可能含有基于指针的赋值的过程调用。同样，过程中的多条路径可能给b指定其他地址。总之，这些复杂的情况使得对把a或*b保留在寄存器所需的分析更加困难。在实践中，很多编译器放弃这一分析并把a留在内存中。 311

一个可以保存在寄存器中的值有时被称为*非歧义性值*（unambiguous value）；可能有多个名字的变量被称为*歧义性值*（ambiguous value）。有若干引发歧义性的方式。存储在基于指针的变量中的值经常是歧义性的。引用调用形参与名字作用域规则之间的相互作用使得这样的形参具有歧义性。很多编译器把数组元素值当作歧义性值处理，因为编译器无法告知诸如A[i, j]和A[m, n]这样的两个引用会否涉及

同一位置。编译器可以执行分析消除其中的某些歧义性。遗憾的是，分析无法解决所有的歧义性问题。因此，编译器必须小心，做好正确地处理歧义性问题的准备。

编译器必须把任意值作为歧义性值处理，除非它能够证明这个值是非歧义性的。歧义性值被保存在内存中而不是寄存器中；必要时它们被装入和存储。对语言仔细推理可以帮助编译器。例如，在C语言中，从不被提取地址的所有局部变量都是非歧义性的。同样地，ANSI C标准要求通过指针变量的引用是类型相容的；因此，对*b的赋值只能改变作为合法指针的b的位置的值。遗憾的是，这一标准把字符指针从这一限制中除去，所以对字符指针的赋值能够改变任意类型的值。

ANSI C包含两个关键字，它们对编译器分析可能的歧义性值内容的能力产生影响。restrict关键字让程序员声明一个指针是非歧义性的。它通常被用于一个过程在一个调用地点直接传递一个地址的时候。volatile关键字让程序员声明一个变量的内容可能随意改变并且不必通知。这一关键字被用于硬件设备寄存器以及可能通过中断服务例程或应用中的其他控制线程修改的变量。

7.2.3 机器特性

对于任意处理器，编译器设计者将发现编译器必须遵守一组机器特有的规则。

体系结构可能要把寄存器单元分成不同的类。这些类可能是不相交的；一个常用的方案是把寄存器文件卷宗分隔成“浮点”寄存器和“通用”寄存器。这些类相互间可以部分重叠，如在“浮点”寄存器
312 和“双精度浮点”寄存器中所发生的那样。在一个类中，可能有附加的限制，如双精度浮点值必须占据一对相邻的寄存器这样的体系结构。其他常用寄存器类包括条件代码寄存器、谓词寄存器、分支目标寄存器以及段寄存器。

这一体系结构可能把一个寄存器类划分成多个不相交的寄存器文件卷宗。这样的机器在一组寄存器聚合一组功能单元。每个功能单元对它相邻的寄存器可以快速存取，而对其他寄存器组中的寄存器只能做受限存取。这一策略让体系结构设计者使用更多的功能单元。它给编译器带来配置操作和数据的负担。

内存常驻值也同样引发目标机器特有的问题。很多体系结构基于值的类型来对它的开始地址做限制。因此，它可能要求整数和单精度浮点数据始于一个字的边界（即字大小的整数倍的地址），而字符数据可能始于任意偶地址。双字和四倍字长边界也可能出现。

指定存储的细节问题将直接影响性能。随着内存层次变得更深且更复杂，局部化和复用的问题对运行时间有很大的影响。通过改变内存中代码和数据的布局和重排存取，编译器可以加强局部化和复用。

7.3 算术操作符

现代处理器为评估表达式提供广泛的支持。典型的RISC机器有完整的三地址操作，包括算术操作、逻辑移位和布尔操作。三地址形式让编译器对任意操作的结果命名，并将其保存以备后面的复用。三地址形式还消除二地址指令的主要复杂性之一，即破坏性操作。

为了生成诸如x+y这样的平凡表达式的代码，编译器首先发行确保x和y的值在如r_x和r_y这样的已知寄存器中的代码。如果x被存储在内存中的当前活动记录中偏移量为@x处，那么结果代码可能是：

```
loadI  @x       ⇒ r1
313 loadAO rarp, r1 ⇒ rx
```

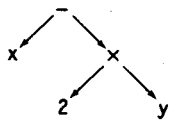
然而，如果x的值已经在一个寄存器中，编译器可以使用这个寄存器的名字代替r_x。编译器根据类似的决策链确保y在一个寄存器中。最后，编译器发行如下所示的执行加法的指令

```
add rx, ry⇒rt
```

如果这一表达式是用树表示的，那么上面的方案很自然地适合于树的后序遍历。图7-2a的代码通过把代码生成动作嵌入一个递归树遍历例程来实现这一代码的发行。它依赖于两个例程*base*和*offset*来隐藏某些复杂性。*base*例程返回保存一个标识符的基地址的寄存器名；如果需要，base发行把这一基地址放入一个寄存器中的代码。*offset*例程有类似的功能；它返回一个寄存器名，这个寄存器保存这个标识符相对于*base*返回的地址的偏移量。

```
expr(node) {
  int result, t1, t2;
  switch(type(node)) {
    case ×, ÷, +, −:
      t1 ← expr(LeftChild(node));
      t2 ← expr(RightChild(node));
      result ← NextRegister();
      emit(op(node), t1, t2, result);
      break;
    case IDENT:
      t1 ← base(node);
      t2 ← offset(node);
      result ← NextRegister();
      emit(loadAO, t1, t2, result);
      break;
    case NUM:
      result ← NextRegister();
      emit(loadI, val(node), none,
           result);
      break;
  }
  return result;
}
```

a) 树遍历代码生成器



b) x−2 × y的抽象语法树

```
loadI   @x       ⇒ r1
loadAO  rarp,r1  ⇒ r2
loadI   2        ⇒ r3
loadI   @y       ⇒ r4
loadAO  rarp, r4 ⇒ r5
mult    r3, r5   ⇒ r6
sub     r2, r6   ⇒ r7
```

c) 原始代码

图7-2 表达式的简单树遍历

314

同样的代码处理+、−、×和÷。从代码生成的角度看，这些操作符是可互换的（忽视交换性）。对如图7-2b所示的表达式x−2 × y的AST调用图7-2a中的例程expr生成如图7-2c所示的结果。这一例子假设无论x还是y都不是已在寄存器中，且假设二者都存储在当前的AR中。

注意代码生成的这一树遍历方法与如图4-14所示的专用语法制导翻译方案之间的类似性。这一树遍历算法使细节更加明了，包括终止符和在右子树之前评估左子树的处理。语法制导翻译方案把第一种情况抽象掉了。它没有评估顺序的控制，因为分析器为各个动作规定运用顺序。但是，这两个方案大体上生成等价的代码。

7.3.1 降低对寄存器的要求

很多问题影响着生成代码的质量。例如，存储位置的选择就对代码质量有直接的影响，甚至对这一简单表达式也是如此。如果y在全局数据区域内，把y放入一个寄存器所需要的指令序列可能要求一个附加的loadI以便得到保存这个值的基地址，并使用一个寄存器来保存这个值。另外，如果y在一个寄存器中，用于把y装入r_5的两个指令可以省略，而且编译器将在mult指令中使用保存y的寄存器名。把值保存在寄存器中既可以避免内存存取又可以避免任意的地址计算。如果x和y都在寄存器中，上面的七指令序列将被缩短到三指令序列（如果目标机器支持立即乘指令，则可缩短到二指令序列）。

被编入树遍历代码生成器的代码形态决策也产生效应。上图中的原始代码使用八个寄存器，包括r_{arp}。这一代码倾向于假设在编译的后期，寄存器分配器可以把寄存器的数量降到最少。例如，寄存器
315 分配器可能把这一表达式重写为：

```
loadI  @x       ⇒ r1
loadAO rarp, r1 ⇒ r1
loadI  2        ⇒ r2
loadI  @y       ⇒ r3
loadAO rarp, r3 ⇒ r3
mult   r2, r3   ⇒ r2
sub    r1, r2   ⇒ r2
```

这把寄存器的使用从八个降到四个，包括r_{arp}。这一代码把结果留在r_2中，这使得r_1中的x和r_3中的y都可以在后面使用。

然而，在计算2 × y之前装入x仍然浪费一个寄存器，这是由于在树遍历代码生成器中在右子结点之前先评估左子结点这一决策造成的。使用相反的顺序可以产生下面代码中左边所示的序列。寄存器分配器可以重写这一序列使其只使用两个寄存器（和r_{arp}），如右边所示：

```
loadI  @y       ⇒ r1          loadI  @y       ⇒ r1
loadAO rarp, r1 ⇒ r2          loadAO rarp, r1 ⇒ r1
loadI  2        ⇒ r3          loadI  2        ⇒ r2
mult   r3, r2   ⇒ r4          mult   r2, r1   ⇒ r1
loadI  @x       ⇒ r5          loadI  @x       ⇒ r2
loadAO rarp,r5  ⇒ r6          loadAO rarp,r2  ⇒ r2
sub    r6, r4   ⇒ r7          sub    r2, r1   ⇒ r1
首先评估2 × y                 寄存器分配后
```

这个分配器不能把x、y和x+2 × y都装入两个寄存器中。正如所写的那样，这一代码保存x，但不保存y。尽管谨慎的优化可以通过移动这一序列在计算的后期装入x从而把三寄存器代码改成二寄存器代码，但是生成这样的序列也许更容易且更直接，这一序列在优化和寄存器分配之后将产生更好的代码。

当然，首先评估右子结点不是一般的解决方案。对于表达式2 × y+x来说，合适的规则是“先评估左子结点”。诸如x+(5+y) × 7这样的表达式无法使用静态规则。限制寄存器使用的最佳评估顺序是先计算
316 5+y，再乘以7，最后加上x，在效果上是在右子结点和左子结点间交替计算。

生成立即地址装入指令

细心的读者也许注意到图7-2中的代码从不生成ILOC的立即地址装入指令loadAI。相反，它生成一个立即装入指令（loadI），后面跟一个地址偏移量装入指令（loadAO）：

```
loadI  @x       ⇒ r1
loadAO rarp,r1  ⇒ r2
```

而非 `loadAI rarp, @x ⇒ r2`

在本书中，我们已假设生成两操作序列比生单一操作更好。下面三个因素给出推荐这一做法的解释：

1）较长的代码序列为@x给出一个明确的名字。如果@x在loadAO指令之外的上下文中被复用，那么这个明确的名字是有用的。

2）偏移量@x对loadAI操作的立即域来说可能太大。如果这个偏移量不合适，那么编译器必须

或者生成使用loadI的序列或者把这个标签存储在内存中并使用一个完整的load重新得到它。

3）上述两指令序列在代码生成器中导致一个清晰的功能分解，如图7-2所示。把常量迭入子序列loadAO所需的逻辑是一种优化。

如果常量偏移量不被复用，那么子序列优化可以单纯地把这一两指令序列变换成单一的loadAI。例如，如果中间寄存器需要用于其他某个值，那么某些寄存器分配器可以自动地执行这一转换（参见第13章）。

如果编译器需要直接生成loadAI，有两个方法有意义。编译器设计者可以把包含于*base*和*offset*中的选择逻辑向上推向图7-2中的*IDENT*的选择中。这以不美观但有较少模块的代码为代价实现这一目的。另外，编译器设计者也可以让*emit*来维护一个小型指令缓冲器，并在指令被生成时对指令执行局部窥孔优化（参见11.4.1节）。保持缓冲器较小使得窥孔优化更实用。如果编译器遵守“多需求子树优先”规则，那么将在loadAO指令之前生成这个偏移量。在窥孔范例中，很容易识别出供给loadAO中的loadI。

317

为了给表达式树的子树选择一个最好的评估顺序，编译器需要知道有关每棵子树的详细信息。为了最小化寄存器的使用，编译器应该首先评估有更多需求的子树，即先评估需要更多寄存器的子树。代码必须在第二个子树的评估过程中保存第一棵子树中计算的值；因此，首先对需求较少的子树进行处理将比多需求子树优先多要求一个寄存器。这需要对代码做一个初始遍历以收集信息，其后再跟着一个发行真实代码的遍。

诸如专用语法制导翻译方案这样的单一遍可以发现需求更多的子树或者发行代码。但它不可以两者都做。该一般原理，即分析后面跟着翻译或转换，可以运用到代码生成和全局优化上。如Floyd于1961年发现的那样，如果我们让编译器在做最后的关于如何实现代码的决定之前检查代码，那么编译器可以生成更好的代码。

7.3.2 存取参数值

树遍历代码生成器隐含地假设存在对于所有标识符可行的单一存取方法。表示形式参数的名字可能需要不同的处理。被传递到AR中的值调用参数可以作为局部变量来处理。被传递到AR中的引用调用参数需要附加一个间接寻址。因此，对于引用调用参数x，编译器可能会生成下面的代码来得到x的值。

```
loadI  @x      ⇒ r1
loadAO rarp,r1 ⇒ r2
load   r2      ⇒ r3
```

318

前两个操作把这一参数值的内存地址移到r_2。最后的操作把这个值本身移到r_3。

许多链接约定把前面几个参数传递到寄存器中。如前所述，图7-2中的代码不能处理被永久保留在寄存器中的值。然而，必要的扩展很容易实现。

对于值调用参数，*IDENT*的选择分支必须检查这个值是否已经在寄存器中。如果是，那么它就给*result*指定寄存器号，或把这个值拷贝到一个新的虚拟寄存器中。否则，它使用标准机制来装入内存中的这个值。

对于通过寄存器传递地址的引用调用参数，编译器需要发行单一操作来从内存装入这个值。然而，这个值在每一个赋值的前后必须驻留在内存中，除非编译器可以证明它是非歧义性的。

7.3.3 表达式中的函数调用

至此，我们假设表达式中的所有操作数都是变量、常量和由其他子表达式所生成的临时值。函数调用也作为操作数出现在表达式中。为了评估函数调用，编译器简单地生成调用这个函数所需要的调用序列（参见6.6节和7.9节）并发行把这个返回值移到寄存器中所需的代码。这一链接约定限制了它对执行函数的调用过程的影响。

函数调用的存在可能会限制编译器改变表达式评估顺序的能力。这一函数可能具有修改表达式中所使用的变量的值的副作用。编译器必须尊重源表达式所含有的评估顺序，至少对于这一调用要如此。没有关于调用的可能副作用的相关知识，编译器不能在这一调用周围移动引用。编译器必须假设最坏的情况：函数既修改又使用它可以存取的每一个变量。改进上述的最坏情况的愿望促进了过程间分析领域的很多工作（参见9.4.2节）。

7.3.4 其他算术操作符

为了处理其他算术操作，我们可以扩展我们的简单模型。基础方案保持不变：把操作数放入寄存器
319 中，执行操作并在有必要的情况下存储结果。编入表达式文法中的优先度确保这一预定的顺序。诸如一元减等的一元操作符评估它们惟一的子树，然后执行这一特定的操作（指针的解除引用操作的行为也是如此）。某些操作符对它们的实现需要复杂的代码序列（例如，求幂、三角函数和简缩操作符。）这些也可能直接展开成直线式布局，或者被处理为对编译器或操作系统所提供的库例程的一个函数调用。

7.3.5 混合型表达式

许多程序设计语言所接受的一个复杂性是带有不同类型操作数的操作。（在这里，我们主要关心的是源语言中基类型的操作，而不是程序员定义的类型。）如4.2节所述，编译器必须识别这一情况并插入每个操作符的换算表所需要的变换代码。这一般包括把一个或两个操作数转换成更一般的类型并在这更一般的类型中执行操作。消耗结果值的操作可能需要把结果转换成另外一种类型。

有些机器提供直接执行这些转换的指令；而另外一些机器却希望编译器生成复杂而与机器相关的代码。无论在哪种情况下，编译器设计者都要在IR中提供转换操作符。这样的操作符压缩封装所有的转换细节，包括任意的控制流，而且让编译器负责统一优化它。因此，代码的动作可以在不关心循环的内部控制流的情况下把不变的转换移出该循环。

一般地，程序设计语言的定义为每一个转换规定一个特殊的公式。例如，在FORTRAN 77中，为了把`integer`转换成`complex`，编译器首先把`integer`转换`real`。编译器把结果数用作复数的实部，并把虚部设为`real`的0。

对于用户定义的类型，编译器没有定义每个特殊情况的转换表。然而，源语言仍定义表达式的意义。编译器的任务是实现那个意义；如果一个转换是不合法的，那么这个转换将被阻止。如第4章所述，很多不合法转换将会被发现，并在编译时被阻止。当编译时检测或者是不可能的或者是不完整的时候，编译器应该生成测试这些不合法情况的运行时检测。当代码尝试一个不合法转换时，这一运行时检测将会
320 引发一个运行时错误。

7.3.6 作为操作符的赋值

大多数类Algol语言实现满足下面简单规则的赋值：

1）评估一个赋值的右侧。

2）评估一个赋值的左侧。

3）把右侧值移到由左侧值所指定的位置。

因此，在诸如x←y的语句中，对两个表达式x和y进行不同的评估。因为y出现在赋值操作符的右侧，它被评估以产生一个值；如果y是一个整型变量，那么该值就是一个整数。因为x在赋值操作符的左侧，它被评估以产生一个位置；如果x是一个整型变量，那么该值就是一个整数的位置。这个位置可能是一个内存地址，或者是一个寄存器。为了区分这两种评估模式，我们有时称一个赋值的右侧的评估结果为*右值*（rvalue），而一个赋值左侧的评估结果称之为*左值*（lvalue）。

赋值可以包含指定与右值类型不同的位置的左值。根据这一类型，或者需要一个转换，或者发生一个类型错误。对于转换，典型的源语言规则让编译器评估这个右值到它的自然类型，即没有赋值上下文时生成的类型。然后，那个值被转换到由这个左值命名的位置的类型，并被存储在适当的位置上。

7.3.7 交换性、结合性以及数系

编译器通常可以利用各操作符的代数性质。加法和乘法满足交换律和结合律，布尔代数操作符也同样。因此，如果编译器看到一个计算x+y的代码片段，然后在不插入对x或y的赋值的前提下计算y+x，那么编译器知道它们计算同一个值。同样地，如果编译器看到表达式x+y+z和w+x+y，那么它应该知道x+y是一个共同的子表达式。如果它用严格按从左到右顺序评估这两个表达式，那么它无法识别这个公共子表达式，因为对于第二个表达式，它将先计算w+x，然后计算w+x+y。 321

编译器应该利用交换律和结合律来改进编译器生成的代码的质量。对表达式的计算进行重排序可以暴露出很多额外的转换机会。然而，重要的警告就是顺序。

> 由于精度上的限制，计算机上的浮点数只代表实数的一个子集，而且浮点操作不保持结合性。因此，编译器不应该对浮点计算重排序，除非语言定义特别允许它那样做。

考虑下面的例子。我们可以给x、y和z赋浮点值使得x、y < z、z-x=z、z-y=z，但是z-(x+y) ≠ z。在这样的情况下，这一结果依赖于评估顺序。计算（z-x)-y产生一个等于z的结果，而先评估x+y并从z中减去刚才的评估值产生一个不同于z的结果。

这一问题是由于浮点数的近似性质造成的；相对于指数的范围来说尾数较小。为了相加两个数，硬件必须把这两个数规格化；如果指数的差异比尾数的精度大，那么较小的数将被截短到零。围绕这一问题，编译器不能容易地按它的方式工作。因此，编译器应该注意这一警告，并避免对浮点计算重排序。

7.4 布尔操作符和关系操作符

大多数程序设计语言在比数更丰富的值集上进行操作。通常，包括布尔操作符和关系操作符，这二者都将产生布尔值。因为大多数程序设计语言有产生布尔值的关系操作符，我们把布尔操作符和关系操作符放到一起考虑。布尔表达式和关系表达式的一般应用是改变程序的控制流。现代程序设计语言的大部分能力都得自于计算和测试这样的值的能力。

为了引入布尔值，语言设计者们在标准表达式文法中加入若干产生式，如图7-3所示。（我们使用符号¬表示非（not），用∧表示与（and），用∨表示或（or），从而避免与相应ILOC操作产生混淆。）而编译器必须决定如何表示这些值以及如何计算它们。对于算术表达式，这样的设计决策很大程度上由目标机器的体系结构来规定，目标机器的体系结构提供数的格式和执行基础算术的指令。 322

幸运的是，处理器设计师似乎对如何支持算术已达成相当广泛的共识。同样地，大多数体系结构提供丰富的布尔操作。然而，对关系操作符的支持在各体系结构之间却有很大的不同。编译器设计者必须使用适当的评估策略，使得语言的需求与可用的指令集合相匹配。

Expr	→	¬ *OrTerm*		|	*RelExpr* > *NumExpr*
	|	*OrTerm*		|	*RelExpr* > *NumExpr*
OrTerm	→	*OrTerm* ∨ *AndTerm*		|	*NumExpr*
	|	*AndTerm*	*NumExpr*	→	*NumExpr* + *Term*
AndTerm	→	*AndTerm* ∧ *Bool*		|	*NumExpr* − *Term*
	|	*Bool*		|	*Term*
Bool	→	*RelExpr*	*Term*	→	*Term* × *Factor*
	|	true		|	*Term* ÷ *Factor*
	|	false		|	*Factor*
RelExpr	→	*RelExpr* < *NumExpr*	*Factor*	→	(*Expr*)
	|	*RelExpr* ≤ *NumExpr*		|	num
	|	*RelExpr* = *NumExpr*		|	ident
	|	*RelExpr* ≠ *NumExpr*			

图7-3 把布尔操作和关系操作加入表达式文法中

7.4.1 表示

传统上，有两种布尔值的表示：数编码和位置编码。前者对true和false赋特殊值，并使用目标机器的算术和逻辑操作来处理它们。后者的方法把表达式的值编码为可执行代码中的一个位置。它使用比较和条件分支来评估这一表达式；不同的控制流路径代表这一评估的结果。每一个方法对一些例子适用，
323 但是对另外一些例子不适用。

1. 数编码

当程序把一个布尔操作或关系操作的结果存储到一个变量时，编译器必须保证这个值有一个具体的表示。编译器设计者必须把数值赋给true和false，使得能很好地工作于诸如与、或和非等硬件操作。典型的是把0赋给false，而把1或一个（¬false）赋给true。

例如，如果b、c和d都在寄存器内，编译器将为表达式b ∨ c ∧ ¬d生成如下的代码：

```
not rd     ⇒ r1
and rc,r1  ⇒ r2
or  rb,r2  ⇒ r3
```

对于一个如x < y的比较，编译器必须生成比较x和y，并把适当的值赋给比较结果的代码。如果目标机器支持一个返回布尔值的比较操作的话，那么下面的代码是显然的：

```
cmp_LT rx, ry ⇒ r1
```

另一方面，如果比较定义一个需要读取并进行条件分支的条件代码，那么结果代码更长、更复杂。⊖这一比较的风格导致x < y的一个较混乱的实现。

```
    comp   rx, ry ⇒ cc1
    cbr_LT cc1    → L1,L2
L1: loadI  true   ⇒ r2
    jumpI         → L3
```

⊖ ILOC在这一点上比较灵活。它包含返回布尔的比较，例如例子中的cmp_LT。它包含返回条件代码的比较comp和一组条件分支操作符，如cbr-LT。条件分支操作符解释条件代码（参见A.5.1节）。

```
L2: loadI  false ⇒ r2
    jumpI        → L3
L3: nop
```

使用条件代码操作实现x<y需要的操作次数比使用返回布尔值的比较所需的操作次数更多。 324

2. 位置编码

在前面的例子中，L_1处的代码创建值true，而L_2处的代码创建值false。在其中的每一个点处，值既是固定的又是已知的。在某些情况下，代码不必为表达式的结果生成一个具体的值。相反，编译器可以在一个位置即代码中的位置，例如L_1，对值进行编码。

图7-4给出树遍历代码生成器为表达式a<b∨c<d∧e<f发行的代码。这一代码评估三个子表达式a<b、c<d和e<f，并使用布尔操作把评估结果值组合起来。遗憾的是，这生成每个路径都有11个操作的操作序列，包括三个分支和三个跳转。

```
     comp    ra, rb ⇒ cc1    // a < b
     cbr_LT  cc1    → L1,L2
L1:  loadI   true   ⇒ r1
     jumpI   → L3
L2:  loadI   false  ⇒ r1
     jumpI   → L3
L3:  comp    rc, rd ⇒ cc2    // c < d
     cbr_LT  cc2    → L4,L5
L4:  loadI   true   ⇒ r2
     jumpI   → L6
L5:  loadI   false  ⇒ r2
     jumpI   → L6
L6:  comp    re, rf ⇒ cc3    // e < f
     cbr_LT  cc3    → L7,L8
L7:  loadI   true   ⇒ r3
     jumpI          → L9
L8:  loadI   false  ⇒ r3
     jumpI          → L9
L9:  and     r2,r3  ⇒ r4
     or      r1,r4  ⇒ r5
```

图7-4 a<b∨c<d∧e<f的原始编码 325

使用true和false表示子表达式的决策导致这一代码的某些复杂性。以短路评估使用位置编码可以导致简单得多的操作序列，如图7-5所示。代码的这一版本只给最终的表达式生成一个值，并把这个值留在r_5中。子表达式的值被当作位置编入代码中。

直到一个操作需要一个值，位置编码避免对表达式赋实际值。例如，把结果赋给一个变量迫使代码提供具体的值。位置编码在代码的控制流路径中隐式地表示表达式的值。这通常导致执行较少操作的更简洁的代码。这对于带有使用条件代码的指令集来说更值得注意，如前所示。

如果表达式的结果从不被存储，那么位置编码有意义。当代码使用表达式的结果来决定*控制流*时，位置编码通常可以避免无关的操作。例如，在下面的代码片段中

```
      comp   ra, rb ⇒ cc1
      cbr_LT cc1    → L3,L1
L1:   comp   rc, rd ⇒ cc2
      cbr_LT cc2    → L2,L4
L2:   comp   re, rf ⇒ cc3
      cbr_LT cc3    → L3,L4
L3:   loadI  true   ⇒ r5
      jumpI         → L5
L4:   loadI  false  ⇒ r5
      jumpI         → L5
L5:   nop
```

图7-5 a<b∨c<d∧e<f的短路评估位置编码

```
if (x < y)
   then statement1
   else statement2
```

326

x<y的惟一用途就是来决定是执行$statement_1$还是执行$statement_2$。为x<y产生一个显式值没有意义，除非这个值可使控制流更有效。

在一台编译器必须使用比较和分支产生值的机器上，编译器可以简单地把$statement_1$和$statement_2$的代码分别放置在编译器将分别赋值到true和false的位置上。这样的位置编码的使用导致比使用数编码更简单且更快的代码。

```
      comp   rx, ry ⇒ cc1    // x < y
      cbr_LT cc1    → L1,L2
L1:   code for statement1
      jumpI         → L6
L2:   code for statement2
      jumpI         → L6
L6:   nop
```

这里，评估x<y的代码已与在$statement_1$和$statement_2$之间的选择代码结合到一起。这一代码把x<y的结果表示成一个位置：或者是L_1或者是L_2。

3. 短路评估

在很多情况下，子表达式的评估决定整个表达式的值。例如，在表达式a<b∨c<d∧e<f中，如果a<b则这一表达式的剩余部分的值就不再重要了。同样地，如果同时有a>b和c>d，则e<f的值就不再重要了。图7-5的代码使用这上面这些事实。如果a<b，那么代码直接分支到产生这一表达式的值true的语句序列。

这一改进布尔表达式和关系表达式的评估方法称为**短路评估**（short-circuit evaluation）。在短路评估中，代码只评估决定最终值所需的表达式部分，不做多余的评估。短路评估依赖于两个布尔等式：

$$\forall\ x,\ \text{false} \wedge x = \text{false}$$
$$\forall\ x,\ \ \text{true} \vee x = \text{true}$$

327 短路评估可以通过考虑表达式中操作数的运行时值降低评估表达式的代价。

某些程序设计语言，例如C语言，要求编译器使用短路评估。例如，C语言中的表达式

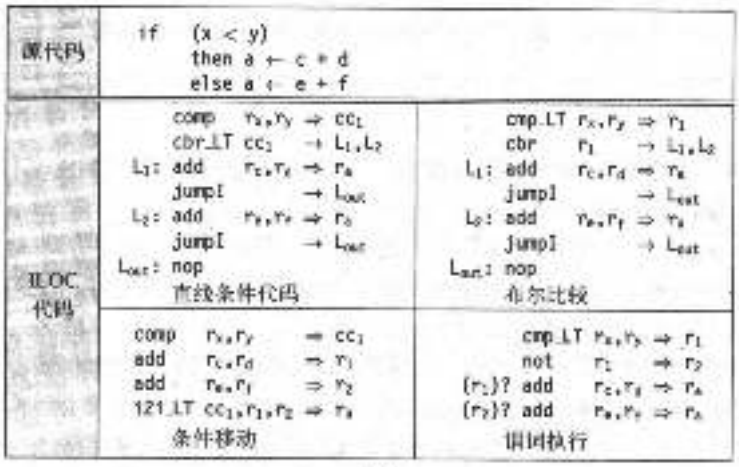
```
(x != 0 && y/x > 0.001)
```

依赖于短路评估来保证它的安全性。如果x是0，那么y/x无定义。显然，程序员想要避免由于除数为零而引发的硬件异常情况。语言定义要求：如果x有值0，则代码将不执行除法。

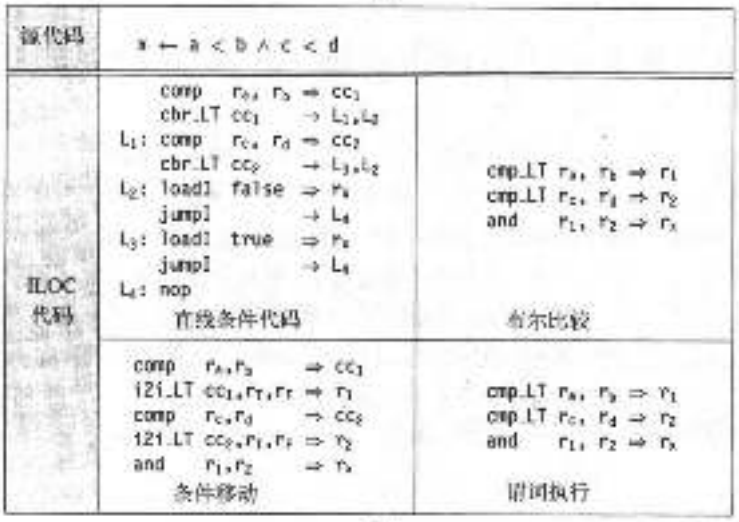
7.4.2 关系操作的硬件支持

目标机器的指令集合中特定的低级细节对于关系值表示的选择有着重要的影响。特别地，编译器设计者必须格外小心地对待条件代码、比较操作和条件移动操作的处理，因为这些操作对不同表示的相对代价有着重要的影响。我们将考虑支持关系表达式的四个方案：直线条件代码、增设条件移动操作的条件代码、取布尔值的比较以及谓词操作。上述每一种方案都是实际实现的理想版本。

图7-6给出两个源代码级的结构和在上述四个方案下的各自的实现。图的上半部分给出控制一对简单赋值语句的if-then-else，而下半部分给出一个布尔值的赋值。



a)



b)

图7-6 实现布尔操作符和关系操作符

1. 直线条件代码

在这一方案中，比较操作设置一个条件代码寄存器。解释这一条件代码的惟一指令是条件分支，它有在六个关系（小于（<）、小于等于（≤）、等于（=）、大于等于（≥）、大于（>）、不等于（≠））上分支的变形。对于不同类型的操作数，这些指令可能存在若干变形。

编译器必须使用条件分支来解释条件代码的值。如果其结果的惟一使用是决定控制流，如图7-6a所示，那么编译器用来读取这一条件代码的条件分支通常也可以实现源代码级别控制流的结构。如果结果被用于布尔操作中，或者被保存在一个变量中，如图7-6b所示，那么代码必须把这个结果转换成具体的布尔表示。图7-6b中的两个loadI操作就是做这一工作的。无论哪种情况，代码对于每一个关系操作符至少有一个条件分支。

条件代码之所以强大是因为处理器通常具有实现附属条件代码这一特性。一般情况下，这些处理器上的算术操作设置一个条件代码来反映它们的计算结果。如果编译器可以为必须执行的算术操作适当地设置条件代码，那么就可以省略比较操作。因此，这一体系结构风格的倡导者们认为，这一方案允许程序的更高效编码，代码执行的指令数小于把布尔值放入通用寄存器的比较器所执行的指令数。

作为优化的短路评估

短路评估的诱因是布尔表达式和关系表达式的值的位置编码。在使用条代码来记录比较结果并使用条件分支来解释条件代码的处理器上，短路评估是有意义的。

当处理器包含诸如条件移动、取布尔值的比较和谓词执行等特性时，短路评估的优势就逊色多了。随着分支等待的增长，短路评估所需的条件分支的代价也相应增长。当分支的代价超过避免评估所节省下来的代价时，短路评估将不再有意义。相反，全评估将更快。

当某些程序设计语言需要短路评估时，如C语言，编译器也许需要执行某些分析来决定何时可以安全地用全评估取代短路评估。因此，未来的C语言编译器也许要包含用全评估取代短路评估的分析和转换，正如过去的编译器执行用短路评估取代全评估的分析和转换那样。

2. 条件移动

这一方案把条件移动指令加入到直线条件代码模型中。在ILOC中，条件移动的形式如下所示。

330 i2i_LT $cc_i, r_j, r_k \Rightarrow r_m$

如果条件代码cc_i与LT匹配，那么r_j的值被拷贝到r_m。否则r_k的值被拷贝到r_m。

条件移动保留使用条件代码的主要优势，当较早的操作已设置这一条件代码时，避免比较操作。如图7-6a所示，条件移动让编译器使用分支编码简单的条件操作。这里，编译器推断评估两个加法。它使用条件移动来得到最后的赋值。只要两个加法不引发异常，这样做就是安全的。

如果编译器在寄存器中有真值和假值，例如真值在r_T中而假值在r_F中，那么编译器可以使用条件移动把条件代码转换成布尔值。图7-6b使用这一策略。它比较a和b并在r_1中创建这一布尔结果。它计算c < d的布尔值并把结果放入r_2。它计算r_1和r_2的逻辑与（and）来作为最后结果。

我们可以期望条件移动指令在一个循环内执行。这样，编译器可以避开分支，而条件移动的执行速度将更快。

3. 取布尔值的比较

这一方案完全避开条件代码。比较操作或者在通用寄存器中或者在专用的布尔寄存器中返回一个布尔值。条件分支取这个结果作为决定其行为的一个参数。

取布尔值的比较对图7-6a没有帮助。图7-6a的代码等价于直线条件代码方案。它需要比较、分支和跳转去评估if-then-else结构。

这一模型的优势如图7-6b所示。与条件移动一样，这一模型无需分支就可以评估关系操作符，如图7-6b中的例子所示。然而，它不需要把比较结果转换成布尔值。对本例来说，布尔值和关系值的统一表示使代码更紧凑、更高效。

这一模型的弱点是它需要显式比较。尽管条件代码模型有时可以通过利用算术操作设定条件代码从而避开比较，但是取布尔值的比较模型总是需要显式比较。

4. 谓词执行

在支持谓词执行的体系结构中，某些（或全部）操作根据取布尔值的操作数的值来决定这个操作是否生效，这种方法称为谓词执行（predicated execution）。这种技术使得编译器在很多情况下避开条件分支。在ILOC中，在谓词指令前包含一个谓词表达式来编写这一谓词指令。为了提醒读者这是谓词，我们把谓词放在一个括号中，并在其后面附加一个问号。例如 331

$(r_{17})?$ add $r_a, r_b \Rightarrow r_c$

表示当且仅当r_{17}包含值true时执行的加法操作（r_a+r_b）。

图7-6a中的例子展示谓词执行的优势。这一代码既简单又紧凑。它生成两个谓词r_1和r_2。它利用这两个谓词控制源程序结构中的then和else部分中的代码。对于图7-6b的例子，谓词生成与布尔比较方案相同的代码。

处理器可以使用谓词避开执行操作，它也可以执行操作并使用谓词避开赋值操作结果。只要空闲的指令不产生异常，这两个方法之间的差异与我们的讨论无关。我们的例子展示产生谓词和它的补所需的操作。为了避免这一情况，处理器可以提供定义两个寄存器的比较，一个是这个布尔值，另一个是这个布尔值的补。

7.4.3 选择表示

编译器设计者在决定何时使用这些表示时有一定的自由度。这一决定取决于硬件对比较的支持、分支（特别是错误预测的条件分支）的代价、短路评估的愿望以及周围代码对结果的使用方式。

比较图7-6中的四个实现表明这些方案之间的差异。考虑图7-6a中的if-then-else结构。四个方案都产生相同长度的执行路径，即四个操作。如果分支中断执行，那么分支方案（最上面的方案）的执行需要花更长时间。而无分支方案（最下面的方案）可以避开这样的中断；它们还产生更简洁的代码。

图7-6b中的布尔赋值展示比较的显示结果与隐式结果之间的差异。条件代码方案（左侧的方案）需要的代码比布尔比较方案（右侧的方案）需要的代码更多。这一赋值需要能够存储于r_x中的具体表示。直接产生这样的表示带来更清晰、更紧凑的代码。在短路评估中，谓词将比布尔比较产生更好的代码。 332

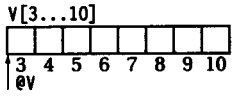
7.5 存储和存取数组

到此为止，我们一直假设存储在内存中的变量包含标量值。很多程序需要数组或类似的结构。定位和引用数组的元素所需要的代码惊人的复杂。本节给出在内存中布局数组的若干方案，并描述每个方案为数组引用生成的代码。

7.5.1 引用向量元素

数组的最简单形式是一维数组；我们称一维数组为向量（vector）。向量一般存储在连续的内存中，

因此第i个元素位于第$i+1$个元素的直接前趋位置。因此，向量V[3...10]生成下面的内存布局，其中每个单元下面的数字表示该单元在向量中的下标:



当编译器遇到如V[6]这样的引用时，它必须在向量中使用这个下标以及从V的声明所得的事实来生成V[6]的偏移量。然后，它的实际地址被计算为这一偏移量和指向V的开始位置的指针的和，我们把这一指针写作@V。

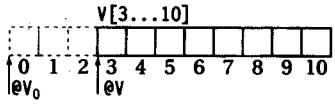
作为一个例子，假设V被声明为V[*low*...*high*]，其中*low*和*high*是向量的下界和上界。为了翻译引用V[i]，编译器既需要一个指向V的存储的开始位置的指针，又需要V中元素i的偏移量。这个偏移量是
333 $(i-low)\times w$，其中w是V的单个元素的长度。因此，如果*low*是3，i是6且w是4，那么元素i的偏移量是$(6-3)\times 4=12$。假设r_i保存i的值，下面的代码片段把V[i]的地址计算到r_4，并把值装入到r_v:

```
loadI @V     ⇒ r@v  // 获得V的地址
subI  r1, 3  ⇒ r1   // (偏移量 - 下界)
multI r1, 4  ⇒ r2   // × 元素长度，4
add   r@v,r2 ⇒ r3   // V[i]的地址
load  r3     ⇒ rv   // V[i]的值
```

注意文本上的简单引用V[i]引入三个算术操作。可以简化和改进这一操作序列。使用下界0消去减法。如果w是2的幂，乘法可以被算术移位取代；实际的程序设计语言中的很多基类型都有这一性质。对地址和偏移量求和似乎是无法避免的；也许这正好可以解释为什么大多数处理器都包含取基地址和偏移量的寻址模型，并在基地址+偏移量处存取这个位置。在ILOC中，我们把这个操作写作loadAO。

```
loadI   @V     ⇒ r@v  // 获得V的地址
subI    r1, 3  ⇒ r1   // (偏移量 - 下界)
lshiftI r1, 2  ⇒ r2   // × 元素长度，4
loadAO  r@v,r2 ⇒ rv   // V[i]的值
```

如果编译器知道V的下界，那么它可以把减去这个下界值的减法叠入@V中。我们不使用@V作为V的基址，而是使用$@V_0 = @V-low\times w$。我们称$@V_0$是V的*伪零*（false zero）。



使用$@V_0$并假设i在r_1内，存取V[i]的代码变成:

```
loadI   @V0      ⇒ r@V0  // 调整V的地址
lshiftI r1, 2    ⇒ r1    // × 元素长度，4
334 loadAO  r@V0, r1 ⇒ rv    // V[i]的值
```

这一版本的代码更短，因此应该也比较快。在手工编码的汇编程序中，这最后的序列可能更受欢迎。在编译器中，由于可以通过揭示诸如乘法和加法的细节以便优化，较长的序列可能产生更好的结果。诸如把乘法转换成移位并使用loadAO重写add-load序列等低级的改进可以在编译的后期完成。

如果编译器没有可用的数组上下界，那么它可能在运行时计算数组的伪零，并在每一次对数组的引用时复用这个值。例如，编译器可以在多次引用一个数组元素的过程的入口处计算这个数组的伪零。如

C语言中所使用的另外一个策略是强制使用0作为数组的下界。这使得@V_0=@V并简化所有的数组地址计算。然而，在不限制程序员选择下界的情况下，在编译器中对细节的注意也可以达到同样的结果。

7.5.2 数组存储布局

存取多维数组的元素需要更多的工作。在讨论编译器必须生成的代码序列之前，我们必须考虑编译器将如何把数组下标映射到内存位置。大多数实现使用下面三个方案之一：*行优先顺序*、*列优先顺序*或*间接向量*。源语言定义通常指定这三个映射中的一个。

存取数组元素所需的代码取决于数组映射到内存的方式。考虑数组A[1...2, 1...4]。从概念上，它的样子是：

A				
	1,1	1,2	1,3	1,4
	2,1	2,2	2,3	2,4

在线性代数中，一个二维矩阵的*行*（row）是它的第一维，*列*（column）是它的第二维。在行优先顺序中，A的元素被映射到连续的内存位置上，使得每一行的相邻元素占据连续的内存位置。这产生下面的布局：

1,1	1,2	1,3	1,4	2,1	2,2	2,3	2,4

335

下面的循环嵌套显示行优先顺序对内存存取模式的影响：

```
for i ← 1 to 2
   for j ← 1 to 4
      A[i,j] ← A[i,j] + 1
```

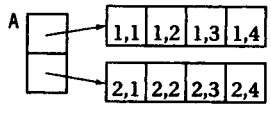
在行优先顺序中，赋值语句按顺序经过内存，开始于A[1, 1], A[1, 2], A[1, 3]，直到A[2, 4]。这一顺序存取也适用于大多数内存层次。把i循环移入j循环的内部产生在行之间跳转的存取序列，它依次存取A[1, 1], A[2, 1], A[1, 2], …, A[2, 4]。对于像A这样的小数组，这不是问题。对于比缓冲器大的数组，缺乏顺序存取将在内存层次中产生拙劣的性能。作为一般原则，当最右侧的下标（在上例中的j）变化最快时，行优先顺序产生顺序存取。

相对于行优先顺序的是列优先顺序。这一方式保持A的列在连续的位置上，产生下面的布局：

1,1	2,1	1,2	2,2	1,3	2,3	1,4	2,4

当最左侧的下标变化最快时，列优先顺序产生顺序存取。在我们的双重循环嵌套中，使i循环在外侧产生非顺序存取，而把i移到循环的内侧时则产生顺序存取。

第三个选择不是一目了然的，但却已用于若干语言中。这一方案使用间接向量把所有多维数组压缩到一个向量集合。对于我们的数组A，这一方案将产生下面的布局：



每一行都有自身的连续存储。在一行内，元素同向量一样寻址。为了允许行向量的系统寻址，编译器分配一个指针向量，并适当初始化这一指针向量。同样的方案可以创建列优先间接向量。

这一方案似乎很简单，但是它带来两种复杂性。第一，较之简单的行优先或列优先布局，它需要更多的存储。每一个数组元素有一个存储位置，同其他两个布局一样；除此之外，内部维数需要间接向量。

336

在向量中入口的数量可能以数组维数的平方增加。图7-7给出更复杂的数组B[1...2,1...3,1...4]的布局。第二，建立数组内部维数的所有指针需要大量的初始化代码。

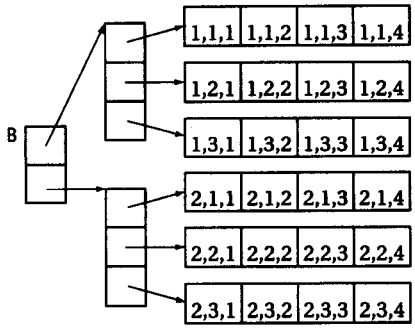


图7-7 B[1...2,1...3,1...4]的列优先顺序的间接向量

上述每一方案已用于现今流行的程序设计语言中。对于以连续存储的方式存储数组的语言，行优先顺序已成为典型的选择，一个著名的例外是FORTRAN，它使用列优先顺序。BCPL和Java都支持间接向量。

7.5.3 引用数组元素

使用数组的程序通常包含对数组的各个元素的引用。当使用向量时，编译器必须把一个数组引用翻译成这一数组存储的起始地址和这一元素相对于这一起始地址的偏移量。

本节描述在行优先顺序中被存储为连续块的数组的地址计算以及作为间接向量的集合的数组的地址计算。列优先顺序的相应地址计算也遵循与上述行优先顺序相同的基本方案，只是维数的顺序反过来了。
337 我们把这一部分留给读者思考。

1. 行优先顺序

在行优先顺序中，地址计算必须寻找行的开始，然后在这一行内生成一个偏移量，就如同它是一个向量一样。扩展用于描述向量边界的表记法，我们把下标加到描述各维的*low*和*high*上。因此，low_1指的是第一维的下界，而$high_2$指的是第二维的上界。在我们的例子A[1...2，1...4]中，low_1是1 $high_2$是4。

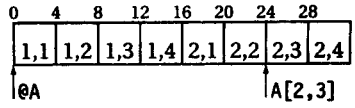
为了存取元素A[i，j]，编译器必须发行计算行i的地址的代码并计算元素j的偏移量，由7.5.1节可知，这一偏移量是 $(j-low_2)\times w$。每一行包含4个元素，这由$high_2-low_2+1$计算，其中$high_2$是最高列，而low_2是最低列，即它们分别是A的第二维的上界和下界。为了简化说明，设$len_k=high_k-low_k+1$，这是第k维的长度。因为行是被连续放置的，行i开始于距A的开始位置 $(i-low_1)\times len_2\times w$处。这表明地址计算是:

$$@A+(i-low_1)\times len_2\times w+(j-low_2)\times w$$

使用实际值取代i、j、low_1、$high_2$、low_2和w，我们发现A[2，3]位于距离A[1，1]偏移量为

$$(2-1)\times(4-1+1)\times 4+(3-1)\times 4=24$$

的地方（假设@A指向距A[1，1]偏移量为0的位置）。考虑内存中的A，我们发现A[1，1]+24的地址实际上是A[2，3]的地址。



在向量的情况下，当编译时上界和下界已知时，我们可以简化这一计算。运用相同的代数创建二维数组的伪零产生如下表达式: 338

$$@A + (i \times len_2 \times w) - (low_1 \times len_2 \times w) + (j \times w) - (low_2 \times w), \text{ or}$$

$$@A + (i \times len_2 \times w) + (j \times w) - (low_1 \times len_2 \times w + low_2 \times w)$$

上述表达式的最后一项（$low_1 \times len_2 \times w + low_2 \times w$）独立于i和j，所以这最后一项可以直接分解到基地址

$$@A_0 = @A - (low_1 \times len_2 \times w + low_2 \times w) = @A - 20$$

现在，数组引用只是

$$@A_0 + i \times len_2 \times w + j \times w$$

最后，我们可以重新分解并把w移出，节省一个额外的乘法:

$$@A_0 + (i \times len_2 + j) \times w$$

对于A[2, 3]的地址，这评估到:

$$@A_0 + (2 \times 4 + 3) \times 4 = @A_0 + 44$$

因为$@A_0$恰好是@A-20，这等价于@A-20+44=@A+24，与数组地址多项式的原来版本所建立的地址相同。

如果我们假设i和j在r_i和r_j内，且len_2是一个常量，那么这种多项式导致下面的代码序列:

```
loadI  @A0      ⇒ r@A0  // 调整A的基地址
multI  ri, len2 ⇒ r1    // i × len2
add    r1, rj   ⇒ r2    // + j
multI  r2, 4    ⇒ r3    // × 元素长度，4
loadAO r@A0,r3  ⇒ rA    // A[i,j]的值
```

在这一形式中，我们已把计算减少到两个乘法和两个加法（其中一个在loadAO内）。第二个乘法可以重写成移位。

如果编译器没有存取数组的边界，那么它必须或者在运行时计算伪零，或者使用更复杂的包含调整下界的减法的多项式。如果数组元素在一个过程中被存取多次，前者的选择可能更有益；在过程的入口 339
处计算伪零使得代码可以使用代价较小的地址计算。仅当数组不常被存取时，更复杂的计算才有意义。

二维数组的地址计算背后的思想可以推广到更高维数组。以列优先顺序存储的数组的地址计算可以以同样的方式得到。我们用于降低地址计算代价的优化同样也可用于其他种类的数组的寻址多项式上。

2. 间接向量

使用间接向量简化存取各个元素所生成的代码。因为最外面的维数被存储为一组向量，最后一步看似7.5.1节中所描述的向量存取。对于B[i, j, k]，最后一步计算距离k的偏移量，最外维的下界以及B的元素的长度。初始的几步可以通过在间接向量的结构中跟踪适当的指针得到这一向量的起始地址。

因此，为了存取图7-7所示的数组B的元素B[i, j, k]，编译器使用B、i和指针的长度来寻找对应于子数组B[i, *, *]的向量。接下来，编译器使用这一结果、j和指针长度来寻找对应于子数组B[i, j, *]的向量。最后，编译器使用下标k的向量地址计算和元素长度w在这个向量中寻找B[i, j, k]的地址。

如果i、j和k的当前值分别在寄存器r_i、r_j和r_k内，且$@B_0$是第一维的零调整地址，那么B[i, j, k]可按如下的形式引用:

```
loadI  @B0      ⇒ r@B0  // B的伪零
multI  ri,4     ⇒ r1    // 指针为4字节
```

```
loadAO r@B0,r1 ⇒ r2    // 获取@B[i,*,*]
multI  rj,4    ⇒ r3    // 指针为4字节
loadAO r2,r3   ⇒ r4    // 获取@B[i,j,*]
multI  rk,4    ⇒ r5    // 指针为4字节
loadAO r4,r5   ⇒ rB    // B[i,j,k]的值
```

340 这一代码假设在间接结构中的指针已被调整到非零下界。如果这些指针没有被调整，那么在r_j和r_k中的值必须对应于相应的下界而减少。在这一例子中，可以用移位取代乘法。

使用间接向量，引用恰好对每一维需要两个指令。这一性质使得对于内存存取相对于算术快捷的体系来说，数组的间接向量实现更高效。例如，1985年之前的大多数计算机上都是如此。若干编译器使用间接向量来操控地址算术的代价。随着内存存取的代价相对于算术的增加，这一方案已失去了它的优势。如果系统再一次呈现相对执行算术所需要的时间内存等待较小的话，间接向量也可能再次成为降低存取代价的一种实用方法。

对于基于高速缓冲器的机器来说，局部化对性能是至关重要的。当数组变得比高速缓冲器大很多的时候，存储顺序影响局部化。行优先和列优先存储方案为某些基于数组的操作产生良好的局部化。没有理由相信间接向量也产生良好的空间局部化。更可能的是这一方案生成随机出现的对内存系统的引用流。

3. 存取数组值参数

当数组作为参数传递时，大多数实现通过引用来传递它。即使是对其他参数都使用值调用的语言中，数组通常也是通过引用传递的。考虑值传递数组所需要的机制。调用过程需要把数组的每一个元素的值拷贝到被调用过程的活动记录中。作为引用参数传递数组可以大幅度降低每一次调用的代价。

如果编译器要在被调用过程中生成一个数组引用，那么它需要有关绑定到这一参数上的数组的维数信息。例如，FORTRAN要求程序员在声明数组时使用常量或其他形参来指定该数组的维数。因此，FORTRAN责成程序员负责向被调用过程传递对参数数组正确寻址所需的信息。

其他语言则把收集、组织和传递必要信息的任务交给编译器。如果数组的大小不能静态地确定，那么这一方法是必要的，因为需要在运行时进行分配。即使当数组的大小可以静态地确定时，这一方法也是有用的，因为它把使代码混淆的细节抽象掉。在这些环境下，编译器构建一个既包含指向数组开始的
341 指针又包含每一维的必要信息的描述器。描述器有一个已知的大小，即使数组的大小在编译时不可知时也是如此。因此，编译器可以在被调用过程的AR中为描述器分配空间。被传递到数组参数槽的值是一个指向这个描述器的指针，这一指针称为*内情向量*（dope vector）。

当编译器生成一个对作为参数传递的数组的引用时，它必须从内情向量中提取信息。它生成的地址多项式与对于局部数组引用所使用的地址多项式相同，只是必要的值是从内情向量装入的。作为一个重要的方针，编译器必须决定它将使用的地址多项式的形式。使用原始的地址多项式，内情向量必须包含一个指向数组开始的指针、每一维的下界以及除一个维之外的所有维的大小。使用基于伪零的地址多项式，下界的信息是不必要的。只要编译器总使用相同形式的多项式，它可以作为每一次过程调用的序言来生成构建内情向量的代码。

一个给定的过程可以从很多调用地点调用，每一个调用传递不同的数组。图7-8中的PL/I过程`main`包含两个调用`fee`的语句。第一语句传递数组`x`，而第二个语句传递数组`y`。在`fee`的内部，实参（`x`或`y`）被绑定到形参`A`。为了允许`fee`的代码引用适当的位置，它需要`A`的内情向量。各自的内情向量如此图的右侧所示，它们都包含伪零和在相应调用地点传递的数组的每一个维的长度。

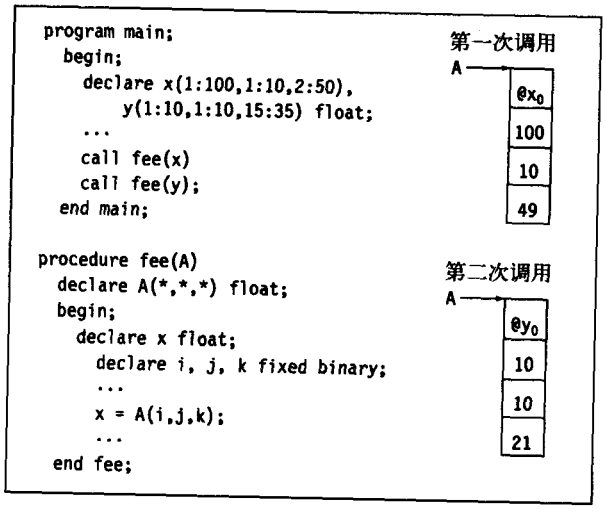


图7-8 内情向量

342

注意，存取数组值参数的代价要比存取局部声明的数组的代价大。最好的情况是，内情向量引入额外的内存引用来存取相应的入口。最坏的情况是，内情向量阻止编译器执行依赖于完备的数组声明的完整知识的优化。

7.5.4 范围检测

大多数程序设计语言的定义都显式或隐式地假设程序只引用在数组的固定边界内的数组元素。根据定义，引用界外元素的程序不是形式良好的。某些语言（例如Java和Ada）要求检查并报告界外存取。对于另外一些语言，很多编译器包含检查并报告界外数组存取的机制。

被称为*范围检测*（range checking）的最简单实现在每一次数组引用前插入一个检测。这一检测核实每一个下标值都落入它被使用的维的有效范围之内。在多用数组的程序中，这样的检测的额外开销成为重要的问题。对于这一简单方案可以做很多改进。在编译器中，最廉价的选择是证明给定的引用不能生成界外引用。

如果编译器打算为数组取值参数插入一个范围检测，那么编译器需要在内情向量中包含附加的信息。例如，如果编译器使用基于数组的伪零的地址多项式，那么它就要有每一维的长度信息，但是没有上界和下界的信息。它可以通过检测相对于数组总长的偏移量来执行不精确的检测。然而，为了执行精确的检测，编译器必须在内情向量中包含每一维的上界和下界的信息，并对照它们进行检测。

当编译器为范围检测生成运行时代码时，它插入很多这一代码的拷贝来报告范围外的下标。这一般包括到运行时错误例程的分支。在正常的执行期间，几乎从不到达这些分支（特别是如果错误处理器终止执行）。如果目标机器让编译器预测这一分支的可能方向，那么这些异常分支应被预测为不执行。编译器也许要利用这些分支导致非正常终止来注释它的IR。从而允许编译器的后续阶段区分“正常执行路径”和“错误路径”；编译器可能能够使用这一路径的差异知识产生更好的正常路径代码。

343

7.6 字符串

程序设计语言为字符数据提供的操作不同于为数值数据提拱的操作。程序设计语言对字符串的支持级别的范围非常之广，从C语言这样大部分处理都采用对库例程的调用形式，到PL/I这样把字符串赋值、形成字符串的任意子串、甚至是串连接作为第一类操作符的形式。为了表示在串实现中所出现问题，本

节讨论串赋值、串连接和计算串长度的实现。

串操作成本可能很高。诸如IBM S/370T和DEC VAX这样的老式CISC体系结构对串处理提供广泛的支持。现代RISC机器更加依赖于编译器来使用一组更简单的操作对这些复杂的操作进行编码。从一个位置到另一个位置的字节拷贝的基本操作出现在很多不同的上下文中。

7.6.1 串的表示

编译器设计者必须为串选择一个表示；这一表示的细节对串操作的成本有重大的影响。

为了看清这一点，考虑下面串b的两个表示。左边的表示是C语言的传统现实中的表示。它使用简单的字符向量，并用特定的字符（'\0'）作为结束符。图示符␣表示一个空格。右侧的表示除了存储这个串之外还存储串的长度（8）。很多语言的实现使用这种方法。



空结束　　显式长度字段

344 如果这个长度字段占据的空间比空结束符占据的空间大，那么存储这一长度将稍微增加内存中串的大小。（我们的例子假设这一长度是4个字节；在实践中，它可能会小一些。）然而，存储这一长度可以简化串上的若干操作。如果一种语言允许可变长度的串存储在被分配固定长度的串中，那么实现器也可能存储这个分配的长度以及这个串。编译器可以使用这一分配长度对赋值和连接做运行时边界检查。

7.6.2 串赋值

串赋值在概念上很简单。在C语言中，b的第三个字符到a的第二个字符的赋值可以编写成如下面左边所示的形式。

```
a[1] = b[2];
```

```
loadI    @b     ⇒ r@b
cloadAI  r@b,2  ⇒ r2
loadI    @a     ⇒ r@a
cstoreAI r2     ⇒ r@a,1
```

在带有字符尺度内存操作（cload和cstore）的机器上，这一赋值被翻译成右边所示的简单代码。（回想，a的第一个字符是a[0]，因为C语言使用零作为所有数组的下界。）

然而，如果底层硬件不支持面向字符的内存操作，那么编译器必须生成更复杂的代码。假设a和b开始于字边界，且一个字符占据1个字节，一个字占据4个字节，那么编译器会发行下面的代码：

```
loadI   @b             ⇒ r@b  // b的地址
load    r@b            ⇒ r1   // 获取第1个字
andI    r1,0x0000FF00  ⇒ r2   // 屏蔽其他
lshiftI r2,8           ⇒ r3   // 将其移动1字节
loadI   @a             ⇒ r@a  // a的地址
load    r@a            ⇒ r4   // 获取第1个字
and     r4,0xFF00FFFF  ⇒ r5   // 屏蔽第2个字符
or      r3,r5          ⇒ r6   // 放入新的第2个字符
store   r6             ⇒ r@a  // 将其放回至a中
```

345

这一代码装入包含b[2]的字，提取这个字符，把这个字符移位到位，再把它屏蔽到包含a[1]的字中的正确位置上，然后将这一结果存储回到应该放置的位置。（这一代码序列所增加的复杂性可以解释为什么

面向字符的装入和存储操作再次变得通用的原因。）

对于较长的串，代码是类似的。PL/I有串赋值操作符。程序员可以编写如a=b这样的语句；其中a和b已被声明为字符串。假设编译器使用显式长度表示。在带有面向字节的load和store操作的机器上，下面简单循环将完成字符串的移动。

```
                loadI   @b       ⇒ r@b
                loadAI  r@b,-4    ⇒ r1      // 获取b的长度
                loadI   @a        ⇒ r@a
                loadAI  r@a,-4    ⇒ r2      // 获取a的长度
                cmp_LT  r2,r1     ⇒ r3      // b能否装入a?
                cbr     r3        → Lsov,L1 // 引发溢出
            L1: loadI   0         ⇒ r4      // 计数器
a = b;          cmp_LT  r4,r1     ⇒ r5      // 是否需要更多的拷贝?
                cbr     r5        → L2,L3
            L2: cloadAO r@b,r4    ⇒ r6      // 从b获取字符
                cstoreAO r6       ⇒ r@a,r4  // 将其置入a中
                addI    r4,1      ⇒ r4      // 递增偏移
                cmp_LT  r4,r1     ⇒ r7      // 是否需要更多的拷贝?
                cbr     r7        → L2,L3
            L3: nop                         // 下一个语句
```

注意，这一代码检测a和b的长度以避免使a超界。标签L_{sov}表示串溢出条件的一个运行时错误处理器。

在带有空结束串的语言中，需要对这一代码做某种程度的改变。考虑将位于b的串拷贝到位于a的串的简单C语言循环。

```
                    loadI  @b    ⇒ r@b // 获取指针
                    loadI  @a    ⇒ r@a
                    loadI  NULL  ⇒ r1  // 结束符
do {
  *a++ = *b++;  L1: cload  r@b   ⇒ r2  // 获取下一个字符
} while (*b != '\0')  cstore r2    ⇒ r@a // 存储
                    addI   r@b,1 ⇒ r@b // 取代指针
                    addI   r@a,1 ⇒ r@a
                    cmp_NE r1,r2 ⇒ r4
                    cbr    r4    → L1,L2
                L2: nop                 // 下一个语句
```

346

如果目标机器支持load和store操作上的自增，那么上述循环中的两个add可以在cload和cstore操作中执行，而这将把这一循环减少到四个操作。（回想一下，C语言最初是为PDP/11设计的，而PDP/11支持自动后置增量。）如果没有自增，那么编译器可以对公共偏移使用cloadAO和cstoreAO来生成更好的代码。这在循环中只需要一个add操作。（注意这一代码改变$r_{@a}$和$r_{@b}$的值。）

如果没有面向字节的内存操作，那么代码将变得更加复杂。编译器将用早前给出的在循环体内屏蔽单一字符的方案来取代上面循环体中的load、store和add部分。取代后的结果代码是可用的，但却是一个很不美观且需要更多指令才能把b拷到a的循环。

另外一个选择是采用在某种程度上更复杂的方案，这一方案使字长度的load和store操作成为一个优势而不是负担。编译器可以使用面向字的循环，后面跟着一个循环后清理操作，循环后清理操作处理

串的尾端所剩余的字符。这一方案如图7-9所示。

```
     loadI    @b              ⇒ r@b
     loadAI   r@b,-4          ⇒ r1      // 获取b的长度
     loadI    @a              ⇒ r@a
     loadAI   r@a,-4          ⇒ r2      // 获取a的长度
     cmp_LT   r2,r1           ⇒ r3      // b能否装入a?
     cbr      r3              → Lsov,L1 // 跳转到错误例程
L1:  andI     r1,0xFFFFFFFC   ⇒ r4      // ⌊length(b)÷4⌋
     loadI    0               ⇒ r5      // 串中的偏移
     cmp_LT   r5,r4           ⇒ r7      // 是否需要更多循环?
     cbr      r7              → L2,L3
L2:  loadAO   r@b,r5          ⇒ r8      // 从b获取一个字
     storeAO  r8              ⇒ r@a,r5  // 将其存储到a中
     addI     r5,4            ⇒ r5      // 递增偏移
     cmp_LT   r5,r4           ⇒ r9      // 是否需要更多循环?
     cbr      r9              → L2,L3
L3:  cmp_LT   r5,r1           ⇒ r10     // 是否已完成?
     cbr      r10             → L4,L5
L4:  cloadAO  r@a,r5          ⇒ r11     // 获取下一个字符
     cstoreAO r11             ⇒ r@b,r5  // 存储下一个字符
     addI     r5,1            ⇒ r5      // 取代偏移
     cmp_LT   r5,r1           ⇒ r12     // 是否已完成?
     cbr      r12             → L4,L5
L5:  nop                                // 下一个语句
```

图7-9 使用全字操作的串赋值（a=b）

图7-9的代码要求更加复杂，因为它必须计算可以使用字操作移动的子串的长度（$\lfloor length(6) \div 4 \rfloor$。[⊖]）这一循环体包含五个指令。它的迭代次数是面向字符循环所使用的迭代次数的四分之一。

在面向字循环结束之后，面向字符循环处理剩余的一个、两个或三个字符。当然，如果编译器知道串长度（并且知道它们比较短），那么它可以完全避开生成循环，取而代之的是发行load和store的正确序列。这避免这一循环的额外开销，但是可能生成更大的程序。

存在更复杂的情况。这包括含有非字对齐的串或子串，以及赋值的源头和目标重叠的情况。面向字符的循环一般在这些情况下都可以正确地工作。更高效的面向字的循环需要额外的工作，诸如为了到达字边界的预循环，和使用一小块额外的空间来缓冲字符并填充源头和目标中不同的排列等。在实践中，很多编译器调用一个优化的库例程来实现这些非平凡的情况。

347 ~ 348

7.6.3 串连接

连接是一个或多个赋值序列的速记。连接有两种基本形式：把串b添加到串a的尾部；创建一个包含a且其后紧跟着b的新串。

前一种情况是在一个长度计算之后跟着一个赋值。编译器发行确定a的长度的代码。如果空间允许，那么编译器接着执行b到紧跟着a的内容的空间的一个拷贝。（如果没有足够的空间，代码将引发一个运

⊖ 原书的$\lfloor length(b) \div 4 \rfloor$是错的。——译者注

行时错误。）后一种情况要求拷贝a和b中的每一个字符。编译器把这种连接处理为一对赋值并生成如前节所示的代码。

无论在哪种情况，编译器将保证分配足够的空间来保存结果。在实践中，编译器或者运行时系统必须知道每个串的分配长度。如果编译器知道这些长度，那么它可以在代码生成期间执行检测并避开为运行时检测发行代码。在编译器不知道a和b的长度的情况下，编译器必须生成在运行时计算这些长度并执行适当的检测和分支的代码。

7.6.4 串长

处理串的程序通常需要计算字符串的长度。在C语言的程序中，标准程序库中的函数strlen取串为参数且返回表示为一个整数的串的长度。在PL/I中，内置函数length完成相同功能。前节所述的两种串表示将为串长度计算带来完全不同的代价。

（1）空结束的串（null-terminated string）

此时，长度计算必须开始于串的开始，并依次检查每一个字符，直到达到空字符。代码类似于C语言的字符拷贝循环。这一代码需要的时间与串的长度成正比。 349

（2）显式长度字段（explicit length field）

此时，长度计算是一个内存引用。在ILOC中，这一长度计算变成把这个串的起始地址装入一个寄存器的loadI，后面跟着一个loadAI以得到这一长度。其计算成本是常数而且很小。

这些表示间的权衡是简单的。空结束可以节省少量空间，但是对于长度计算需要更多代码和更长时间。显式长度字段对每一个串的代价是多一个字，但是可以使长度计算取常量时间。

串优化问题的一个典型例子是求两个串a和b的连接的长度。在带有串操作符的语言中，这可以写成length (a+b)，其中+表示串的连接。这一表达式有两个显然的实现：构造连接结果串并计算其长度（在C语言中是strlen(strcat(a, b))），以及求a和b的长度和（在C语言中是strlen(a)+strlen(b)）。当然，后者是理想的。使用显式长度字段，这一操作可以优化成使用两个load和一个add。

7.7 结构引用

出现在大多数程序设计语言中的另一种复杂的数据结构是结构或其某种变形。在C语言中，结构汇总各命名元素，它们通常具有不同的类型。例如，在C语言中，可以使用下面的结构来创建整数列表：

```
struct node {
  int value;
  struct node *next;
}
NilNode = {0, (struct node*) 0};
struct node *NIL = &NilNode;
```

每一个node包含一个整数和一个指向另一个node的指针。NilNode是一个带有值0并将next设为非法值（0）的node。

在C语言中使用结构要求使用*指针值*（pointer value）。例如，NIL的声明创建类型为指向node的*指针*的值，并且初始化这个值使得它指向NilNode（使用取地址操作符&）。指针的这样的使用给编译器带来两个不同的问题：*匿名值*（anonymous value）和*结构布局*（structure layout）。 350

7.7.1 装入和存储匿名值

C语言以两种方式创建结构实例。它可以声明结构实例；在前述的例子中，NileNode被声明为一个

node实例。另外，代码也可以动态地分配结构实例。对于被声明为指向node的指针变量fee，这一分配如下所示：

```
fee = (node *) malloc(sizeof(node));
```

对于这一新node的惟一存取是通过指针fee，因此，我们可以认为它是一个匿名值，因为它没有永久的名字。

因为匿名值的惟一名字是指针，所以编译器无法轻易地确定两个指针引用是否指向同一个内存位置。考虑下面的代码片段：

```
1  p1 = (node *) malloc(sizeof(node));
2  p2 = (node *) malloc(sizeof(node));
3  if (...)
4     then p3 = p1;
5     else p3 = p2;
6  p1->value = ...;
7  p3->value = ...;
8  ...   = p1->value;
```

前两行创建匿名node。第6行对经由p1能到达的node进行写入，而第7行经由p3进行写入。因为这两个语句经由if-then-else，p3既可能引用在第1行分配的node，也可能引用在第2行分配的node。最后，第8行引用p1->value。

为了实现第6行到第8行的赋值序列，编译器应该在寄存器中保存复用的值。遗憾的是，编译器无法轻易地确定第8行是引用在第6行生成的值还是引用在第7行生成的值。为了知道在第8行中引用了哪个node，编译器必须知道在第3行评估的条件表达式的值。尽管在某些特殊的实例中这是可能的（例如1>2），但在一般情况下编译器是无法确定的。除非编译器对这一条件表达式的值非常了解，它必须为这三个赋值发行保守的代码，它必须从内存装入第8行使用的值，即使最近在寄存器中已经有这个值。

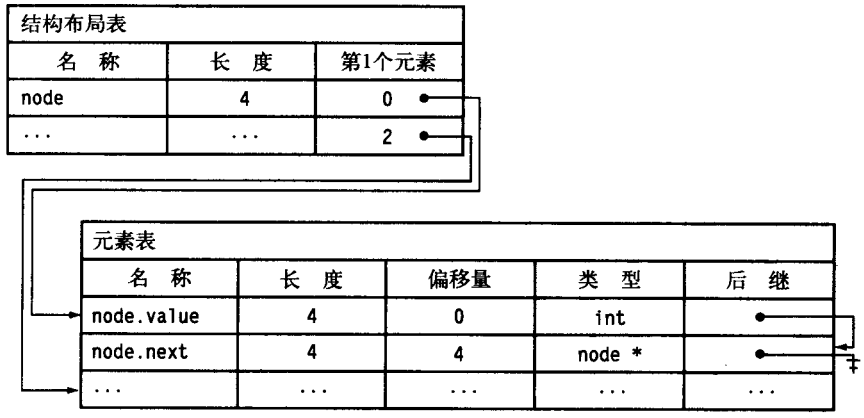
351

对于匿名对象引用的这一不确定性使得编译器无法把基于指针的引用所使用的值保存在寄存器中，从而使得包含基于指针引用的语句不如相应的非歧义性局部变量上的计算高效。分析指针值并使用分析的结果来消除歧义性引用，即用一种使得编译器能够保留某些寄存器的值的方法重写引用，是大量使用指针的程序的可能优化的主要来源。遗憾的是，消除歧义性指针引用所需的分析可能成本过高，因为它可能要求编译器检查整个程序。

大量使用数组的代码也存在类似的效应。除非编译器对数组下标执行深入的分析，编译器无法确定两个数组引用是否重叠。当编译器不能区分诸如a[i, j, k]和a[i, j, l]这样的两个引用时，它必须保守地处理这两个引用。消除数组引用歧义性的问题是具有挑战性的，但要比消除指针引用歧义性的问题容易。

7.7.2 理解结构布局

当编译器为结构引用发行代码时，它需要知道结构实例的起始地址和偏移量以及每个结构元素的长度。为了维护这一信息，编译器一般构建一个独立的结构布局表。这一编译时表必须包含每个结构元素的文本名称、它在这一结构的偏移量以及它在源语言中的数据类型。对于列表的例子，编译器可能构建下面的结构：



352

元素表中的各项使用完整的限定名。这样可以避免在若干不同结构中复用名字而引发的冲突。(很少有语言要求程序在不同的结构中使用不同的元素名字。)

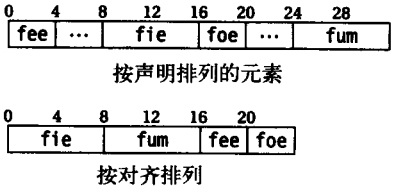
利用这样的信息，编译器可以容易地为结构引用生成代码。对于声明为node *的指针p1，引用p1→next可能翻译成如下ILOC序列:

```
loadI  4      ⇒ r1   // next的偏移
loadAO rp1,r1 ⇒ r2   // p1→next的值
```

在这里，通过从这一结构表中node项到元素表中node的项的链跟踪这一表，编译器找到next的偏移量。遍历这一node的项的链，编译器找到node.next的项和它的偏移量4。

在对结构进行布局并把偏移量赋给它的元素时，编译器必须遵守目标机器的排列规则。这可能迫使编译器在结构中留出未使用空间。当编译器对下面左边声明的结构进行布局时，它遇到这样的问题:

```
struct example {
  int fee;
  double fie;
  int foe;
  double fum;
} e1;
```



右上图给出当编译器被限制为按声明顺序放置元素时的结构布局。因为fie和fum必须双字对齐，所以编译器必须在fee和foe之后插入填充空间。如果编译器可以随意排列内存中元素，那么它可以使用上图右下方所示的布局，从而避免填充空间的需求。这是语言设计的问题: 语言定义指定结构的布局是否要向用户公开。

7.7.3 结构数组

很多程序设计语言允许用户声明结构数组。如果允许用户取数组的结构值元素的地址，那么编译器必须在内存中把这一数据布局为这一结构布局的多重拷贝。如果程序员不能取数组的结构值元素的地址，那么编译器可能把这一数据布局为由本身是数组的元素构成的结构。依赖于周围代码对数据的存取方式，这两个策略在带有缓冲内存的系统上可能性能完全不同。

353

为了对布局成结构的多重拷贝的结构数组寻址，编译器使用如7.5节中所述的数组地址多项式。这一结构的总长，包括所有所需的填充空间，成为地址多项式中的元素大小w。这一多项式生成这一结构实例的起始地址。为了得到特定元素的值，这个元素的偏移量被加到这一实例的起始地址上。

如果编译器把数据布局为元素是数组的结构，那么它必须使用偏移量表信息和数组维数计算元素数组的起始位置。使用适当的数组地址多项式，这一地址可以用作地址计算的起始点。

7.7.4 联合和运行时标签

联合及其变体展示额外的复杂性。为了发行对联合元素引用的代码，编译器必须把引用解析成特定的偏移量。联合可以包含多重结构定义。如果多重结构定义使用相同的名字，那么编译器需要一种把这一名字解析成运行时对象中预期位置的方法。

这一问题有一个语言学上的解决方案。程序设计语言可以迫使程序员使引用不具歧义性。考虑下面的C语言声明：

```
struct n1 {
  int kind;
  int value;
} i1;
struct n2 {
  int kind;
  float value;
} f1;
union one {
  struct n1 inode;
  struct n2 fnode;
} u1;
```

```
union two {
  struct {
    int kind;
    int value;
  } inode;
  struct {
    int kind;
    float value;
  } fnode;
} u2;
u1.inode.value = 1;
u2.fnode.value = 2.0;
```

354

两个联合one和two都可以保存一个带有整数值或浮点值的node。为了区分它们，程序员必须编写完整限定名，如上例末尾的两个赋值中所示的那样。

注意n1和n2本身也是结构声明。程序员可以分配并处理n1和n2。相反，two的定义创建并使用只在two的上下文中有意义的两个匿名结构。为了减轻编写这些完整限定名的负担，这一定义倾向于把区分两个变体的责任交给类型系统。然而在实践中，很难保证对联合中的元素的引用可以转换成单一的类型。

作为另外一个选择，某些系统依赖于运行时区分。在这里，联合中的每一变体有一个不同于其他变体的域。编译器可以发行检查那个域的值的代码，这本质上是基于区分域的值的选择语句，从而保证每一个对象得到正确的处理。语言可能要求程序员在声明中包含这个“标签”域和它的值；另外，编译器也可能自动地生成并插入标签。在这样的系统中，编译器有很强的类型检测的动机，消除类型相关的代码可以在很大程度上节省运行时间和代码空间。

当然，对于变体中相同的部分，语言可以允许对其使用不完整的限定名。很多语言认识到这一点并以某种特殊的方法处理变体的相同原始成分。程序可以通过把这些原始成分限定到特定的变体来引用它们。

7.8 控制流结构

一个基本块只是一个直线型、非判定代码的最大长度序列。对控制流没有影响的任意语句都可以出现在一个块内。所有控制流转移结束一个块，带标签语句也同样，因为它是分支的目标。当编译器生成代码时，它可以通过简单地汇集连续、无标签和非控制流操作来构建基本块。（我们假设带标签语句不是被无偿地加上标签，即每个带标签语句都是某个分支的目标。）基本块的表示无需太复杂。例如，如果编译器拥有存放于简单线性数组中的汇编式表示，那么块可以由序对<*first*，*last*>来描述，这一序对保存块开始和块结束指令的索引。（如果块索引是以递增数字序存储的，那么一个*first*数组就足够了。）

355

为了把一组块结合到一起使它们形成一个过程，编译器必须插入实现源程序控制流操作的代码。为

了刻画块之间的关系，很多编译器构建控制流图（CFG，参见5.3.2节和9.2.1节）并使用这一图做分析、优化和代码生成。在CFG中，结点表示基本块，边表示块之间的可能控制转移。CFG一般是一种派生表示，它包含对每一个块的更详细表示的引用。

实现控制流构造的代码居于基本块中，它在每一个块的尾端处或尾端的附近。（ILOC没有在分支中落空的情况，所以每一个块结束于一个分支或跳转。如果IR模型化等待槽，那么控制流操作可以不是块中的最后操作。）尽管很多语法约定被用于表示控制流，但是其基础概念却很少。本节列举现代程序设计语言中建立的大多数控制流结构。

7.8.1 条件执行

大多数程序设计语言提供某种版本的if-then-else结构。给定源程序文本：

```
if expr
   then statement1
   else statement2
statement3
```

编译器必须生成评估*expr*的代码以及基于这一*expr*的评估值到$statement_1$或$statement_2$的分支。实现$statement_1$和$statement_2$的ILOC代码必须结束于一个到$statement_3$的跳转。如7.4节所述，编译器对实现if-then-else有多种选择。

7.4节中的讨论集中于评估控制表达式。它指出基础指令集合是对处理控制表达式的策略以及在某些情况下处理被控制语句的策略的影响。

程序员可以把任意大的代码片段放置到then和else部分。这些代码片段的大小对编译器实现if-then-else结构的策略有直接的影响。对于如图7-6所示的平凡的then和else部分，编译器主要考虑的是使表达式的评估与底层硬件相匹配。当then和else部分增长时，then和else内部的高效实现的意义开始超出执行控制表达式的代价。 356

例如，在一台支持谓词执行的机器上，在then和else内部的较大模块使用谓词可能浪费执行周期。因为处理器必须把每个谓词指令传送到一个功能单元，所以带有假谓词的每一个操作也有一个机会成本，它被绑定到一个传送槽。对于then和else下的代码的较大模块，非执行指令的成本可能超过使用条件分支的负荷。

图7-10说明了这一权衡。图7-10假设then和else都包含十个彼此独立的ILOC操作，而且假设目标机器在每个周期可以发行两个操作。

左侧给出使用谓词可能生成的代码；它假设控制表达式的值在r_1内。这一代码在每个周期发行两个指令。在每个周期执行其中的一个。then部分的所有操作被发行到Unit 1，而else部分的所有操作被发行到Unit 2。这一代码避开所有分支。如果每一个操作需要一个周期，那么它需要十个周期来执行被控制语句，这与执行的分支无关。 357

右侧给出使用分支而生成的代码；它假设对于then部分控制流到L_1，而对于else部分控制流到L_2。因为指令是独立的，所以这一代码在每个周期发行两个指令。沿着then路径需要五个周期来执行对这一路径的操作，加上最终的跳转的代价。else路径的代价与此相同。

谓词版本避开了非谓词代码中所需要的初始分支（图中到L_1或者到L_2的分支），也避开了终端跳转（到L_3）。分支版本承受一个分支和一个跳转的额外负荷，但是可能执行的更快。每一个路径包含一个条件分支、五个操作的周期和终端跳转。（有些操作可能用于填充跳转上的等待槽。）谓词版本与分支版本的差异在于有效发行率，分支版本发行的指令数目大约为谓词版本发行的指令数目的一半。当then和else中的代码片段增大时，这一差异就明显增大。

使用谓词		使用分支	
Unit1	Unit2	Unit1	Unit2
比较⇒r_1		比较与分支	
(r_1) op_1	($\neg r_1$) op_{11}	L_1: op_1	op_2
(r_1) op_2	($\neg r_1$) op_{12}	op_3	op_4
(r_1) op_3	($\neg r_1$) op_{13}	op_5	op_6
(r_1) op_4	($\neg r_1$) op_{14}	op_7	op_8
(r_1) op_5	($\neg r_1$) op_{15}	op_9	op_{10}
(r_1) op_6	($\neg r_1$) op_{16}	jumpI → L_3	
(r_1) op_7	($\neg r_1$) op_{17}	L_2: op_{11}	op_{12}
(r_1) op_8	($\neg r_1$) op_{18}	op_{13}	op_{14}
(r_1) op_9	($\neg r_1$) op_{19}	op_{15}	op_{16}
(r_1) op_{10}	($\neg r_1$) op_{20}	op_{17}	op_{18}
		op_{19}	op_{20}
		jumpI → L_3	
		L_3: nop	

图7-10 谓词与分支的比较

用户所做的分支预测

对于编译器，有一个关于分支预测的传说。FORTRAN拥有一个算术if语句，该语句基于控制表达式被评估到一个负数、0还是一个正数来采取三个分支中的一个。一个早期编译器允许用户为每个标签提供一个权，以此反映采取该分支的相对概率。这一编译器使用这些权来对分支进行排序，使得分支的预期等待达到最小。

故事发生在这一编译器推出的一年之后：一位维护人员发现编译器是按相反顺序使用分支加权的，它最大化预期等待。在此之前没有任何人抱怨过。这个故事告诉我们，程序员关于他们所编写代码的行为的那些判断是没有意义的。（当然，没有报告显示有人改进了那个编译器，使其按正确的顺序使用分支加权。）

在使用分支还是谓词来实现if-then-else的选择上需要格外小心。应该考虑下面几个问题：

358 1）每个部分的期望执行频率。如果条件的一边的期望执行比另一边多很多，那么加速那一路径的执行的技术可能产生更快的代码。这一倾向性可以取谓词判断分支的形式、推测性地执行某些指令的形式或重排逻辑的形式。

2）代码的不平均量。如果结构的一条路径包含的指令数目比另一条路径包含的指令数目多很多，那么这可能倾向于谓词或谓词与分支的结合。

3）结构内的控制流。如果结构的某条路径包含非平凡的控制流，例如另一个if-then-else、一个循环、一个选择语句或一个过程调用，那么谓词可能不是最有效的选择。特别是嵌套的if结构创建更复杂的谓词表达式，同时降低被发行指令的实际执行率。

为了做出更好的决策，编译器必须考虑所有这些因素，同时还要考虑周围的上下文。这些因素在编译的早期可能难以评估；例如，优化可能使它们发生重大的改变。

7.8.2 循环和迭代

大多数程序设计语言包含执行迭代的循环结构。第一个FORTRAN编译器引入了do循环来执行迭代。今天，循环有多种形式。它们的主要部分有相似的结构。

以C语言的for循环为例。图7-11展示编译器如何对代码进行布局。for循环有三个控制表达式：提供初始化的e_1；评估为一个布尔值并控制循环执行的e_2；在每一次迭代的尾端执行并主要用于更新e_2中使用的值的e_3。我们将使用图7-11作为解释若干种循环的实现的基本方案。

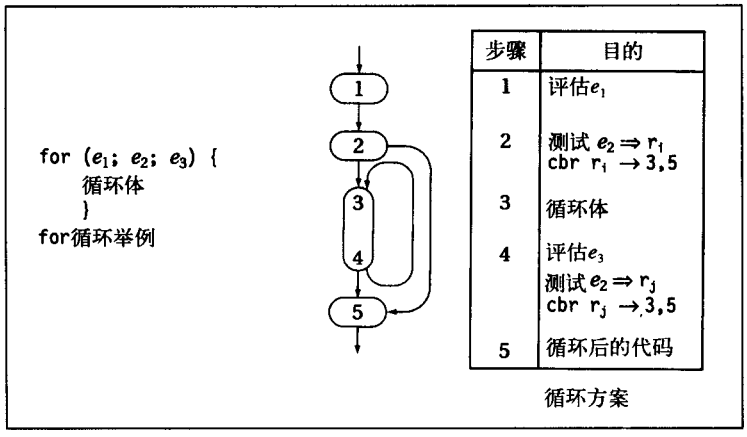


图7-11 C语言中for循环的代码布局

如果循环体是由单一基本模块组成的，即它不包含其他控制流，那么得自于这一方案的循环有一个初始分支，加上每次迭代的一个分支。编译器也许用下面两种方式之一隐藏这一分支的等待。如果体系结构允许编译器判断是否要采用这一分支，那么编译器应该在步骤四判断这一分支被采用（并开始下一次迭代）。如果体系结构允许编译器把指令移进这一分支的等待槽，那么编译器应该设法用循环体的指令填充等待槽。 359

1. for循环

为了把for循环映射到代码中，编译器遵循图7-11的方案。下面的例子通过展示产生于特定循环的ILOC代码做出五个具体步骤。步骤1和步骤2产生单一基本块，代码如下所示：

```
                        loadI   1       ⇒ r1     // 步骤 1
                        loadI   100     ⇒ r1     // 步骤 2
                        cmp_GT  r1,r1   ⇒ r2
for (i=1; i<=100; i++) {
                        cbr     r2      → L2,L1
循环体
}                   L1: 循环体                   // 步骤 3
后继语句                addI    r1,1    ⇒ r1     // 步骤 4
                        cmp_LE  r1,r1   ⇒ r3
                        cbr     r3      → L1,L2
                    L2: 后继语句                 // 步骤 5
```

在这种情况下，步骤1使用loadI把常量1移到一个寄存器中以供步骤4使用。如果循环体（步骤3）或者由单一基本模块组成，或者结束于单一基本模块，那么在步骤4中的更新和测试可以使用这一模块而得到优化。从而允许编译器改进这一代码，例如，编译器使用步骤3的尾端的操作填充步骤4的分支中 360

的等待槽。

编译器也可以使循环的形式只有一个测试的拷贝，即步骤2的测试。以这种形式，步骤4评估e_3然后跳转到步骤2。循环的这一形式比两测试形式少一个操作。(cmp-GT消失，而且cbr变成一个jump。)然而，它甚至为最简单的循环创建一个两模块的循环，而且使通过循环的路径至少加长一个操作。当代码大小是一个重要的考虑因素时，那么统一使用这一更简洁的循环形式是值得的。只要循环结束跳转是一个立即跳转，硬件可以最小化它可能引起的任意中断。

图7-11的这一规范的循环形态也为后期的优化设置平台。例如，如果e_1和e_2只包含已知常量，如在这一例子中那样，那么编译器可以把步骤1的值迭入步骤2的测试，同时或者消除比较和分支（如果控制进入循环），或者消除循环体（如果控制从不进入循环体）。在单一测试循环中，编译器无法做到这一点。这时，编译器发现到达这一测试的两条路径，一条路径从步骤1开始，另一个路径从步骤4开始。用于测试的值r_i沿着步骤4开始的边有可变值，所以测试的结果是不可预测的。

2. FORTRAN的do循环

在FORTRAN中，迭代循环是do循环。它类似于C语言的for循环，但是有更严格的形式。

```
                        loadI  1      ⇒ rj     // j ← 1
                        loadI  1      ⇒ ri     // 步骤 1
                        loadI  100    ⇒ r1     // 步骤 2
   j = 1                cmp_GT ri,r1  ⇒ r2
   do 10 i = 1, 100     cbr    r2     → L2,L1
      循环体
      j = j + 2      L1: 循环体                 // 步骤 3
10    continue           addI   rj,2   ⇒ rj     // j ← j + 2
   后继语句              addI   ri,1   ⇒ ri     // 步骤 4
                        cmp_LE ri,r1  ⇒ r3
                        cbr    r3     → L1,L2
                     L2: 后继语句               // 步骤 5
```

361 注释把部分ILOC代码映射回到图7-11的方案中。

如很多语言的定义一样，FORTRAN的定义有一些有趣的特点。其中的一个与do循环以及它们的索引变量相关。循环的迭代数量在执行进入循环之前是固定的。如果程序改变索引变量的值，那么这一改变不影响执行的迭代数量。例如，下面的循环应该执行100次迭代，即使这一循环递增i。

```
                        loadI  1      ⇒ ri     // 步骤 1
                        loadI  1      ⇒ ri     // 影子
                        loadI  100    ⇒ r2     // 步骤 2
                        cmp_GT r1,r2  ⇒ r3
   do 10 i = 1, 100     cbr    r3     → L2,L1
      循环体
      i = i + 2      L1: 循环体                 // 步骤 3
10    continue           addI   ri,2   ⇒ ri     // i ← i + 2
   后继语句              addI   ri,1   ⇒ ri     // 步骤 4
                        addI   r1,1   ⇒ r1     // 影子
                        cmp_LE r1,r2  ⇒ r4
                        cbr    r4     → L1,L2
                     L2: 后继语句               // 步骤 5
```

为了保证正确的行为，编译器也许需要生成一个称为*影子索引变量*（shadow index variable）的隐式归纳变量来控制迭代。在本例中，这一影子索引变量是r_2。

除非编译器可以确定循环体对这一归纳变量不做修改，否则编译器必须生成一个影子变量。如果循环包含对另一个过程的调用且把归纳变量作为引用调用参数传递，那么编译器必须假设被调用过程修改这一归纳变量，除非编译器能够证明它不被修改。

过程间分析，即整个程序的分析，处理上述问题。程序员把循环索引变量如i作为一个参数传递，这使得程序能够在它生成的输出中包含这一索引值。过程间分析可以轻松地识别何时传递循环索引变量不改变它的值。这使得编译器能够消除影子变量。 362

3. While循环

while循环也可以使用图7-11的循环方案来实现。与C语言的for循环或FORTRAN的do循环不同，while循环没有初始化。因此，这一循环的代码更简洁：

```
                        cmp_LT rx,ry ⇒ r1   // 步骤 2
                        cbr    r1    → L1,L2
while (x < y) {
  循环体                L1: 循环体            // 步骤 3
}
                        cmp_LT rx,ry ⇒ r2   // 步骤 4
后继语句
                        cbr    r2    → L1,L2
                        L2: 后继语句          // 步骤 5
```

复制步骤4的测试给出使用单一基本块创建循环的可能性。for循环中得到的成果同样也可运用于while循环。

4. Until循环

只要控制表达式为假，until循环就一直迭代下去。它在每一次迭代后检查控制表达式。因此，它总是进入循环并至少执行一次迭代。这产生一个特殊的简单循环结构，因为它避开了方案中的步骤1和步骤2。

```
                        L1: 循环体            // 步骤 3
{                           cmp_LT rx,ry ⇒ r2   // 步骤 4
  循环体                    cbr    r2    → L2,L1
} until (x < y)
后继语句                L2: 后继语句          // 步骤 5
```

C语言没有until循环。它的do结构类似于until循环，只是条件的意义被逆置。只要条件表达式评估为真，do循环就一直迭代下去，而只要条件评估为假until循环就一直迭代下去。 363

5. 把迭代表示成为尾递归

在类Lisp语言中，（程序员）常常使用递归风格的形式来实现迭代。如果一个函数执行的最后一个动作是一次调用，那么这一调用称为*尾调用*（tail call）。如果这一尾调用是一个自递归，那么这个调用称为*尾递归*（tail recursion）。例如，为了在Scheme中寻找一个列表的最后一个元素，那么程序员可以编写下面简单的函数：

```
(define (last alon)
  (cond
    ((empty? alon) empty)
    ((empty? (cdr alon)) (car alon))
    (else (last (cdr alon)))))
```

尾递归调用可以优化成一个返回过程头部的跳转，加上创建参数绑定效应的某些赋值。因此，这一代码可以避免过程调用中的大部分额外开销。它无需创建新的AR、评估参数、存储和恢复寄存器，以及调整显示或存取链接。大部分调用前序列消失；所有返回后序列消失。另外，它避免调用的标准实现中可能执行的所有返回（和结束语序列）。结果代码从效率上可以与for循环相匹敌。

7.8.3 选择语句

很多程序设计语言包含选择（case）语句的某种变形。FORTRAN有自己的计算转向（goto）语句。Algol-W引入了选择语句的现代形式。BCPL和C语言有switch结构。PL/I有映射到一组嵌套的if-then-else语句的一般结构。正如本章所介绍的那样，高效地实现选择语句很复杂。

考虑C语言的switch语句。其实现策略应该是（1）评估控制表达式；（2）分支到所选的选择；以及（3）执行这一选择的代码。步骤1和步骤3很容易理解；它们都遵循本章其他地方的讨论。这些选择通常结束于把控制转移到switch语句后面的语句的break语句。

364 实现选择语句的复杂部分是发行定位指定选择的有效代码。

1. 线性搜索

定位适当选择的最简单方法是把选择语句作为一组嵌套的if-then-else语句的描述处理。例如，在左下侧的switch语句可以翻译成右侧的嵌套if语句。

```
switch (b×c+d)                    t1 ← b×c+d
{                                 if (t1 = 0)
  case 0:  block0;                   then block0
           break;                 else if (t1 = 1)
  case 1:  block1;                   then block1
           break;                 else if (t1 = 2)
  ...                                then block2
  case 9:  block9;                ...
           break;                 else if (t1 = 9)
  default: block10;                  then block9
           break;                    else block10
}
```

这一翻译保持switch语句的意义，但是达到各选择的代价取决于各选项的编写顺序。这一代码本质上使用线性搜索来发现希望的选择。当选择的数量较小时，这一策略可能很有效。

2. 二分搜索

当选择的数量增加时，线性搜索的效率就成了问题。高效搜索的经典答案可以运用于这一情况。如果编译器能够在选择标签上赋予一个顺序，那么它可以使用二分搜索来得到一个对数时间复杂度搜索，而不是线性时间复杂度搜索。

想法很简单。编译器构建一个选择标签的紧凑的有序表，以及它们相应的分支标签。它使用二分搜索来发现一个匹配的选择标签，或发现不存在合适的选择。最后，它或者分支到相应的标签，或者分支到default选择。

对于前面所述的switch语句，可以使用右边的

365 跳转表和搜索例程：

值	标签
0	LB_0
1	LB_1
2	LB_2
3	LB_3
4	LB_4
5	LB_5
6	LB_6
7	LB_7
8	LB_8
9	LB_9

```
t1 ← b × c + d
down ← 0
up ← 10
while (down + 1 < up)
{
  middle ← (up + down + 1) ÷ 2
  if (Value[middle]   t1)
     then down ← middle
     else up ← middle
}
if (Value[down]= t1)
   then jump to Label[down]
   else jump to LBdefault
```

在这一方案中，每个模块的代码片段是相互独立的。块i开始于一个标签LB_i，结束于跳转到选择语句后面语句的跳转。defaul选择在块$LB_{default}$中。

如果要搜索的选择标签存在的话，二分搜索在$\log_2(n)$个迭代中发现该选择标签，其中n是选择的数量。如果这一标签不存在，二分搜索发现这一事实并跳转到default选择的块。

因为编译器插入搜索代码，所以它需要小心地选择操作。middle的计算需要两个加法和一个移位。if-then-else将花费更长时间。装入的地址计算很简单：一个移位和一个地址+偏移量的装入。使用条件移动或谓词执行（参见7.4.2节），编译器可以用两个操作更新up和down。没有这两个操作，更新取一个分支、一个拷贝和一个跳转。 366

3. 地址的直接计算

如果选择标签形成一个稠密集合，那么编译器可以比二分搜索做的更好。在上述例子中，switch语句对每个从零到九的整数各有标签。在这样的情况下，编译器可以构建一个包含模块标签LB_i的向量，并通过执行一个简单的地址计算找到适当的标签。

对于这个例子，可以如前所示计算t_1来发现标签，并把t_1当作进入这一表格的索引。在这样的情况下，实现选择语句的代码可能有如下形式：

```
t1 ← b × c + d
if (0 > t1 or t1 > 9)
   then jump to LBdefault
   else
     t2 ← memory(@Table + t1 × 4)
     jump to t2
```

这里假设标签的表示是4个字节大小。

对于标签的稠密集合，这一方案生成高效代码。代价很小且是常数。如果在标签集合中有几个洞，那么编译器使用default选择的标签填充这些槽。如果没有default选择，那么适当的行动要依赖于语言。例如，在C语言中，代码要分支到switch之后的第一个语句，所以编译器必须把该语句的标签放置到表中的每一个洞中。

4. 各方法之间的选择

对每个选择语句，编译器必须选择适当的实现方案。这一决策依赖于选择的数量和选择标签集的性质。对于少量的选择，嵌套的if-then-else方案就可以工作得很好。对于选择数量较大且选择值不形成稠密集合时，那么二分搜索是合理的选择。（然而，使用调试器对汇编码进行按步调试的程序员可能惊讶地发现while循环被嵌入到选择语句中！）当选择标签形成一个稠密集合时，使用跳转表的直接计算可能是最好的选择。

对于选择数量足够多的情况，编译器也许采用混合的策略，把两个或更多的方案结合到一起。对前面所述的跳转表的寻址伪码实际上就是实现了一个混合策略。这一代码使用简单的条件检查范围在0到9外面的值，并且或者跳转到$LB_{default}$，或者使用跳转表跳转到适当的标签。这使$LB_{default}$选择成为上述范围外的所有值对应的选择，并对于范围内的选择的稠密集合使用跳转表。 367

如果编译器设计者加入跳转表实现，那么应该尽可能使跳转的可能目标对程序的IR形式可见。在ILOC中，编译器可以生成tbl伪操作的适当集合。缺乏这样的线索将迫使后来的分析遍把伪边加到它们所建立的控制流图中。这些CFG中的错误将导致失去分析产生的信息的精确性。

7.8.4 中断语句

若干语言实现break语句或exit语句的变形。break语句是退出控制流结构的一种结构化方式。在一

个循环中，break把控制转移到循环后面的第一个语句。对于嵌套的循环，break一般退出最内侧的循环。一些语言，例如Ada和Java都允许在break语句上有可选的标签。这将引发break语句从由这个标签所描述的最近结构中退出。在嵌套的循环中，标签化的break允许程序一次退出若干循环。C语言还把break用于switch语句来把控制转移到switch语句后面的语句。

这些动作有简单的实现。每一个循环和选择语句应该结束于循环后面语句的标签。可以把break实现为到这一标签的立即跳转。某些语言包含skip或continue语句来跳转到循环的下一次迭代。同样可以把它们实现为到重新评估控制表达式并测试其值的代码的立即跳转。另外，编译器也可以在skip出现的地点简单地插入评估、测试和分支的拷贝。

7.9 过程调用

过程调用的绝大部分的实现是直截了当的。如图7-12所示，过程调用由调用者中的调用前序列和返回后序列，以及被调用者中的序言序列和结语序列组成。一个过程可以有若干调用点，每个调用点都有
368 自己的调用前序列和返回后序列。在大多数环境下，这一过程还包含一个序言序列和一个结语序列。⊖其中所涉及的大部分内容已在6.6节中描述。本节集中精力讨论影响编译器生成高效、简洁和相容的过程调用代码的能力的问题。

作为一般规则，把操作从调用前序列和返回后序列移到序言序列和结语序列可以减少最终代码的整体大小。如果如图7-12所示的从p到q的调用是在整个程序中对q的惟一调用，那么把操作从p中的调用前序列移到q中的序言序列（以及把操作从p中的返回后序列移到q中的结语序列）对代码的大小没有影响。然而，如果存在对q的其他调用，而且编译器（在每个调用点）把一个操作从调用者移到被调用者，那么通过使用单一操作替换该操作的多重拷贝，编译器可以减小整个代码的大小。当对单一过程的调用数量上升时，这种代码减小的程度就会增加。我们假设大多数过程是从若干个位置调用的；否则，程序员和编译器都应考虑在过程的惟一调用点处内嵌地包含这个过程。

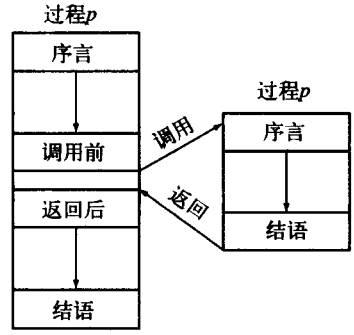


图7-12 标准一

7.9.1 评估实参

369 在构建调用前序列中，编译器必须发行评估实参或调用变量的代码。编译器把参数作为表达式处理。对于值调用参数，调用前序列评估实参，并把评估结果存储为这一参数指定的位置上，它或者在一个寄存器里，或者在被调用者的AR中。对于引用调用参数，调用前序列计算每个实参的地址，并把它存储在为这一参数指定的位置上。如果这一参数驻留在调用者的寄存器中，或者这一参数是一个表达式，那么编译器可能需要为这一参数的值创建一个位置，使得它拥有传递到被调用者的地址。

如果源程序指定实参的评估顺序，那么当然编译器必须遵守这一顺序。否则，编译器将使用一个相容的顺序：或者从左到右，或者从右到左。评估顺序对可能产生副作用的参数非常重要。例如，使用两个例程push和pop来处理栈的程序在从左到右和从右到左的评估顺序下为序列push(pop()-pop())产生不同结果。

过程可能有若干隐式参数。这些参数包括过程的ARP、调用者的ARP、返回地址以及在建立可寻址性中所需要的任何信息。在面向对象语言中，接受器的地址是一个隐式参数。这些隐式参数的一部分被

⊖ 如果语言允许过程有多个出口点，如FORTRAN和PL/I一样，那么过程可能包含多个序言序列。

传送到寄存器里；ARP和返回地址一般驻留在寄存器中。很多体系结构有像jsr $label_1 \Rightarrow r_1$这样的操作，这一操作把控制转移到$label_1$，并把下一个操作（jsr后面的操作）的地址放置到r_1中。其他隐式参数，诸如调用者的ARP，一般驻留在内存中。

7.9.2 过程值参数

当一个调用点把过程当作参数传递时，那个值需要特殊的处理。如果p调用q，作为一个参数传递过程r，p向q传递的信息必须比r的开始地址更多。特别是如果被编译代码使用存取链接来寻找非局部变量，那么被调用者需要r的词法级别，使得对r的系列调用可以找到r的级别的正确存取链接。编译器可以构造一个<地址，级别>对，并把它（或它的地址）传递到过程值参数的位置上。当编译器为过程值参数构建调用前序列时，它必须插入额外的代码来得到过程r的词法级别，并相应地调整存取链接。

7.9.3 保存和恢复寄存器

在所有的调用约定下，与过程调用相关的一个或两个过程必须保存寄存器的值。通常，链接约定使 370
用调用者保存寄存器和被调用者保存寄存器相结合的形式。随着内存操作的代价的增加和寄存器数量的上升，在调用点的寄存器保存和恢复的代价也上升，对于这一点值得特别留意。

在保存和恢复寄存器的策略选择上，编译器设计者必须既考虑效率又考虑代码的大小。目标机器的某些特性影响这一选择。将寄存器组的一部分取出的特征能够减小代码的大小。这样的范例包括SPARC机器上的寄存器窗口，PowerPC处理器上的多寄存器装入和存储操作以及VAX上的高级调用操作。它们都向编译器提供保存和恢复寄存器组的一部分的简洁方法。然而在利用这些特征时，编译器设计者也必须要考虑速度的问题。例如，在某些PowerPC模型上，一系列存储操作比等价的多元存储操作要快。

较大的寄存器组在增加代码存储和恢复的寄存器的数量的同时，一般也允许编译器设计者使用这些额外的寄存器改进结果代码的速度。使用较少的寄存器，编译器将被迫在代码的各处生成装入和存储；使用较多的寄存器，大部分的存取只出现在调用点。（较大的寄存器组可以减小代码中总的存取数量。）把保存和恢复集中在调用点，编译器有机会以比在整个过程的各处分散存取更好的方法来处理它们。

1. 使用多寄存器内存操作

在保存和恢复相邻的寄存器时，编译器通常可以使用多寄存器内存操作。很多体系结构支持双字和四倍字长的装入和存储操作。这可以减小代码的大小；也可以改进执行速度。泛式多寄存器装入和多寄存器存储操作有相同的效应。

2. 使用库例程

当寄存器保存和恢复操作的数量增加时，调用前序列和返回后序列的大小就成了问题。编译器设计者可以用对编译器提供的保存的恢复例程的调用来取代各内存操作的序列。对所有调用都这样做，则可以有效地减小代码的大小。因为保存和恢复例程只为编译器所知，编译器可以使用最小限度的调用序列来把运行时成本保持在较低的水平上。

保存和恢复例程可以取一个参数，该参数指定哪些寄存器必须被保存。生成一般情况的优化版本很有价值，例如保存所有调用者保存寄存器或被调用者保存寄存器。 371

3. 结合职责

为了进一步减小额外开销，编译器也许要把调用者保存寄存器和被调用者保存寄存器的工作结合起来。在这一方案中，调用者把一个指定哪些寄存器必须被保存的值传递给被调用者。而被调用者则把它

必须保存的寄存器加到这个值上，并调用编译器提供的适当保存例程。[⊖]结语序列把相同的值传递给恢复例程使得它能够恢复所需的寄存器。这一方法把额外开销限制在为保存寄存器的一个调用和为恢复它们的一个调用上。它利用调用例程的代价分离了调用者保存和被调用者保存的责任。

编译器设计者必须密切注意不同选择的代码大小和运行时速度。代码应该使用最快的保存和恢复操作。这要求仔细考察目标体系结构上单寄存器操作和多寄存器操作的代价。使用库例程执行保存和恢复可以节省空间；这些例程的精心实现可能减少调用它们的附加代价。

7.9.4 叶过程的优化

编译器可以通过叶例程不调用其他过程的事实而容易地识别出它们来。可以裁减叶过程的序言序列和结语序列来消除以建立序列调用为惟一目的的那些操作。这样的例子包括存储返回地址和更新显示；上述两种操作在叶过程中都是不需要的。（如果寄存器需要保存返回地址或存取链接，那么寄存器分配器腾出这一寄存器。）

编译器也许还能够简化寄存器存储的行为。特别是对于需要较少寄存器的较小叶过程，编译器可以把那些值放入调用者保存寄存器中，而不把它们保存在被调用者保存寄存器中。使用前面所提出的库保存和恢复方法，有可能做进一步的优化；如果叶过程知道调用者保存寄存器的值在该过程不被修改的话，那么它可以修改调用者对调用者保存寄存器的描述。在这一方案中，叶过程显式地保存它需要保存的寄
372 存器。这可以减少在调用中执行的保存和恢复的总数量，例如一个刚好使用四个寄存器的例程可以使用内联代码显式地保存它们，而不去调用保存和恢复例程。（这一方案可能完全避免调用者指定的保存和恢复。）

7.10 实现面向对象语言

正如我们在第6章中所看到的那样，面向对象语言与类Algol语言之间的主要差异在于把名字映射到运行时对象的机制不同。实现面向对象语言的主要困难是要把程序为寻找方法、数据成员和其他对象而进行的遍历的运行时结构链接在一起。

从生成各方法的代码的角度考虑这一问题。因为方法可以存取任意成为其接受器对象的任意成员，所以编译器必须给统一应用于每一个接受器的每一个成员建立一个偏移量，即为可以找到当前方法的每一个对象建立偏移量。当编译器处理类的声明时，它构建这些偏移量，类中的对象本身不包含代码。下面各小节讨论若干种选择。

7.10.1 单一类，无继承

最简单的情况发生于类结构在编译时可知并且不含有继承的时候。假设我们有一个带有方法fee、fie、foe和fum以及数据成员x和y的类giant，其中x和y都是数。为了简化使用giant的程序设计，这个类维护一个类变量n，它记录已创建的giant的数量。为了创建对象，每个类实现方法new，它定位于这个类的方法表中偏移量为0的地方。编译器必须为类giant的每一个实例布局对象记录，而且必须为（类class的实例）类giant布局一个对象记录。

giant的实例的对象记录是一个长度为三的向量。这一向量的第一个槽保存这个槽的类指针，这一指针有类giant的具体表示的地址。其余两个槽保存数据成员x和y。类giant的每一个对象有类似的对象

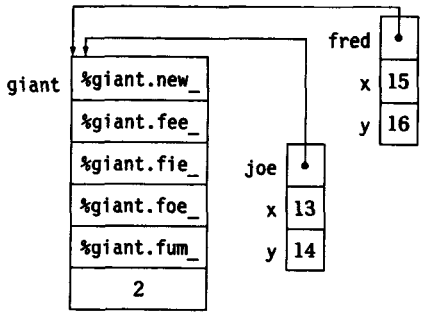
⊖ 在这一方案中，存储所有被保存的寄存器的自然地点是被调用者的AR。保存和恢复例程可以使用被调用者的AR来存取寄存器保存区域。

记录，这些记录是由giant的方法new创建的。

giant本身的类记录必须包含它的所有类变量的空间，在本例中就是n的空间，以及它的所有方法的空间。概念上，这一类记录包含两个成员：所有方法的入口点地址表，以及n的一个槽。因为这一方法表只有通过地址来引用，所以可以把它内联地分配成这一类记录的一部分。如果编译器总是把方法表放置在类记录中一个固定的偏移量处，那么代码就可以容易且低成本地找到这一方法表。例如，可以对类指针和适当的偏移量使用loadAO。对于giant，编译器可能在记录表中指定下面的偏移量： 373

	new	fee	fie	foe	fum	n
偏移量	0	4	8	12	16	20

在代码创建两个giant的实例，调用的fred和joe之后，与giant相关的运行时结构如下所示：



giant的类记录中的方法表是由指向每一个方法的代码的指针组成的。在上面图表中，我们不是显式地表示这些指针，而是给出每一方法名的混合形式。为了从过程名计算汇编级别标签，编译器通常加入程序员在源语言中不能使用的字符。上图使用完整限定名，并对这些名字加入一个百分号（%）作为前缀，加入下划线（-）作为后缀。这生成易计算、相容的标签。

数据成员的值被存储在适当的偏移量处。为了区分它们，我们分别把joe.x、joe.y、fred.x和fred.y设置为13、14、15和16。因为存在两个giant实例，giant.n包含值2。如果程序包含初始化数据成员的代码，那么编译器必须在new方法中包含这一代码。

为了弄清事实，考虑编译器为giant的方法new生成的代码。它必须为一个新对象记录分配空间、初始化每一个数据成员到一个适当的值、递增giant.n并返回指向新创建的对象记录的指针。因为new是一个完备的过程，它还需要实现链接约定；它可能需要保存某些寄存器值，创建一个活动记录，并在返回之前恢复环境。（如果new足够简单，它也许能够把所有状态保存在为链接而保留的寄存器中。）编译器使用它的混合标签%ginat.new_给new的入口赋标签。对于所有方法，编译器把接受器的指派名作为它的第一个隐式参数，如果没有明确指定，指派名是this。 374

当编译器看到一个包含giant的结构时，它生成的代码类似于用于类Algol语言中的代码。例如，诸如实例化giant的jane←jiant.new这样的语句将使用一个名字混合标签来获得giant的类记录的地址，并生成对存储于该地址中的地址的过程调用。⊖

同样地，为fum调用joe的方法的表达式将生成一个对%giant.fum_的调用并以joe为它的接受者。在joe.fum可以合法出现的所有上下文中，joe必须映射到一个已知的位置上，即一个地址可被计算的位

⊖ 如果编译器不能静态地分配giant的类记录，那么它可能需要一个多层次间接形式。编译器可以为giant创建一个静态的全局变量并让创建giant的类记录的代码把适当的地址存储到那个全局位置。

置上。那个位置或者是joe的对象记录的开始，或者包含指向joe的对象记录的指针。（它们必须能够由调用的上下中的类型信息来区分。）编译器生成把这个地址装入joe的对象记录中偏移量为零处的代码；这指向giant的类记录的开始。编译器生成把这个字装入类记录中偏移量为16的代码，在这里编译器发现%giant.fum_的地址。最后，编译器生成对这一地址的过程调用，并把joe的地址隐式地传递给参数this。使用过程链接的标准技术来处理其他参数。

为了创建其他类，编译器遵循同样的步骤。它在实例的对象记录和类记录中指定偏移量。使用这些偏移量和通过对适当的对象记录的间接引用，编译器编译方法。它使用名字混合来为每个过程和类记录创建不同的标签。它使用适当的标签进行静态初始化来填充方法表。

7.10.2 单一继承

继承在几个方面使面向对象语言的实现复杂化。每个类的类记录需要几个额外的成员。因为继承，
375 寻找和调用方法的机制必须以可存取的方式定位方法。最后，为使这些方法运作起来，每个实例的对象记录必须包含所有由超类指定的实例变量。类和对象记录中的额外成员的加入是显然的；加入额外的方法需要更多的工作。

使用继承，程序员可以使用不同的方法来实现类giant。图7-13给出一个这样的实现。由于继承，一个类的对象记录现在有一个类指针；这个类指针是类对象记录的第一个成员。所有类的对象记录的第二个成员是它的超类指针。这一超类指针显示giant是mc的一个子类，而mc又是sc的一个子类。考察图7-13可知每个类都实现new。除class外的所有类都实现fee。Giant从mc继承foe，从sc继承fum。

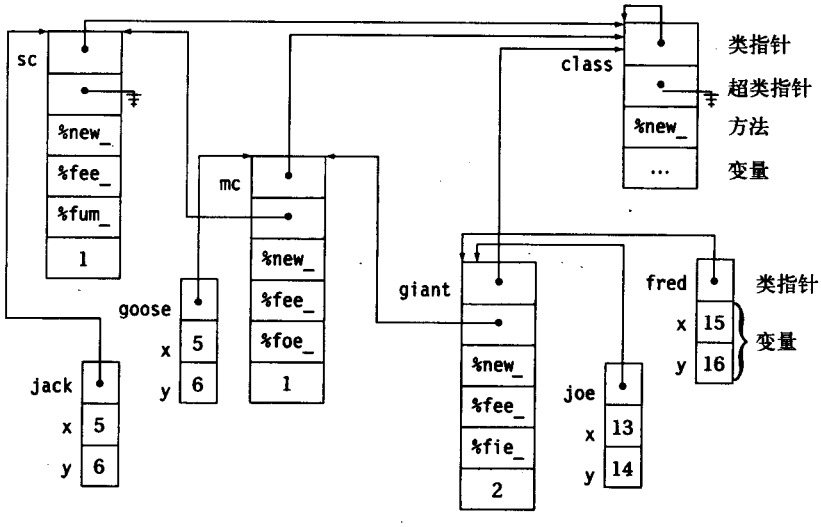


图7-13 实现继承

如果类的结构是固定且已知的，那么通过在每个类中加入一个完整的方法表，编译器可以简化方法调度的实现。在这一方案中，对于任意的方法，可以使用接受器的类指针和类的方法表中该方法的偏移量来调用该方法。这样避免遍历继承层次的额外间接寻址。如果类结构在运行时是可变的，那么
376 编译器仍可以设法使用完整的方法表。为了实现这一点，编译器必须提供在运行时构建方法表的机制。这一机制用于在执行的开始构建初始方法表。当类结构改变时它被重新调用（例如，在Java中类装入器装入新类时）。为了使这一工作更高效，编译器设计者应该避免重新计算类层次中没有发生变化的部分

的方法表。

如果编译器不能设法使用完整的方法表，那么代码可能需要把在类层次中进行查找的工作作为每个方法调度的一个部分。例如，如果语言允许在运行时创建新类，那么可能就需要进行动态调度。在这种情况下，编译器必须通过按祖先序搜索每个类的方法表来实现方法查找。这将戏剧性地增加每一次调用的代价。

为了执行动态方法查找，编译器必须把每个方法名映射到一个搜索键。这一映射可以很简单，使用这个方法的名字或这个方法名字的散列索引作为键。这一映射也可以很复杂，编译器可能使用链接时机制从某个紧集为每个方法名指定一个整数。无论哪种情况，编译器都必须生成一个在运行时可以搜索的表来寻找由这个接受器的类的最近祖先实现的这一方法的实例。

为了改进这一环境下的方法查找，运行时系统可以实现一个方法缓冲，即大多数现代处理器上的硬件数据缓冲的一个软件模拟。方法缓冲有较少的入口，例如1000个。缓冲入口由一个键、一个类和一个方法指针组成。键是包含类和方法搜索键的序对。方法查找首先在缓冲器中查找带有该方法的搜索键和接受器类的入口。如果该入口存在，查找返回被缓冲的方法指针。如果入口不存在，查找执行完整的搜索，搜索开始于接受器的类并沿着超类链向上搜索，直到它找到方法搜索键。在找到方法搜索键这一点，查找为当前搜索的结果创建一个新的缓冲入口，并返回方法指针。

当然，新入口的创建也许将迫使其他缓冲器入口的逐出。诸如最近最少使用或循环复用等标准缓冲替换策略可以用于选择替换的入口。更大的缓冲器保留更多的信息，但是需要更多内存，并且可能花费更长的时间来搜索。在任意改变类结构的操作点，我们要清除方法缓冲以防止查找找到不正确的结果。

1. 布局对象记录

继承的实现影响着编译器必须布局对象记录的方式。为使继承正常工作，对象必须能作为它自身的 377
类或它的超类中定义的方法的接受器正确地工作。这规定了这一对象的数据成员出现的顺序。如果fred是类mc的方法的接受器，那么这一方法只知道fred从mc和mc的祖先继承下来的实例变量。然而，为使那些继承来的方法工作，那些实例变量在giant的实例中和在mc的实例中的偏移量必须相同。否则，当使用fred或joe作为接受器调用mc的方法时，该方法将引用错误的值。

这一般表明各成员要以继承的顺序来布局。在类giant的对象jim的对象记录中的最前面的实例变量是从sc的定义而来的，后面跟着从mc的定义而来的实例变量，后面再跟着从giant的定义而来的实例变量。这导致下面的布局：

类指针	sc数据成员	mc数据成员	giant 数据成员

现在，存在于超类链中的每一个实例变量在每一个类中都有相同的偏移量。⊖

同样的想法也适用于方法表的布局。在对每一个类中都使用完整方法表的系统中，有一个从方法名到相对于层次链的偏移量的相容映射是至关重要的。任意类，如类new中出现的方法总有相同的偏移量。保证这一点的简单方法是遵循与前面所述的对数据成员的方法表的偏移量规则同样的规则。定义在当前类中的方法位于其方法表的末端，它的前面是立即超类的方法，如此沿继承链一直往上推。然而，当一

⊖ 这一观察的一个推论是每个类的对象记录必须包含类class的声明中定义的所有实例变量。这一结论并没有反映在上面的图中。

个类声明由其某个超类所定义的方法的一个新实现时，这一实现的方法指针必须存储于与该方法的前面实现相同的偏移量处。

2. 多重继承

在共享方法和数据时，为了向程序员提供更大的灵活性，一些面向对象语言允许一个类直接从多个超类继承而来。这些语言通常放松对单一遗传层次的包含要求，一个类可以从超类继承一部分而不是全
378 部方法。这需要一个语言学上的机制来精确地指定从一个超类中继承哪些成员。

多重继承进一步复杂化布局对象记录的问题。如果类*c*是从*a*和*b*继承而来的，但是*a*和*b*在继承层次上不相关，那么编译器必须设计一个能够对来自于这两个类的方法进行正确操作的对象记录。这要求额外的支持来调整环境，以使单一方法的实现既适用于类*b*的实例也适用于类*c*的实例。

假设类*c*实现fee，从*a*继承fie，且从*b*继承foe和fum。单一继承的对象布局适用于在三个类中的两个类，例如*a*和*c*。

类指针	*a*数据成员	*b*数据成员	*c*数据成员

我们的布局规则首先放置一个超类，在此例中为*a*，而最后放置目前的类，即此例中的*c*。当使用这一对象记录调用*a*的方法时，它在期望的偏移量处找到*a*的所有实例变量。因为该方法是用*a*的类定义而不是用*b*或*c*的类定义来编译的，所以它不能存取存在的所有实例变量，因为它们可能是从*b*继承而来或直接从类*c*的定义而来的。因此这一方法将正确地发挥功能。

同样地，当以这一对象记录为接受器调用作为*c*的一部分被编译的方法时，这一方法将发现所有实例变量都在它们所希望的位置上。因为*c*的描述包含它的继承的描述，所以编译器知道每个实例变量在对象记录中的位置。

当调用*b*的方法时发生问题。对象记录中的实例变量因为是从*b*继承而来的，所以它们是在错误的位置上；它们处于距对象记录的开始位置偏移量为*a*的实例变量的累加长度的地方。为了补偿这一偏移量，并让一个使用*b*编译的方法正确发挥功能，编译器必须插入代码来调整接受器指针，使得这一指针指向对象记录的中部，指向*b*的实例变量的适当位置。

为了调整偏移量，编译器有两个选择。它可以在对象记录中记录常量调整值，并使用总是把一个偏移量加到接受器指针this的链接约定。这对每次调用增加一个装入和一个加法，这是一个小代价。另外，编译器也可以为类*b*的每个方法创建一个所谓的*蹦床函数*（trampoline function），这是一个使this增加所需量，然后调用*b*的当前方法的函数。如果需要的话（即如果接受器是由引用传递的话），它也可能减少这一接受器指针。这把一个过程调用加入调用链上，这比装入和加法的代价要大。使用蹦床函数的*c*的类记录如图7-14所示。

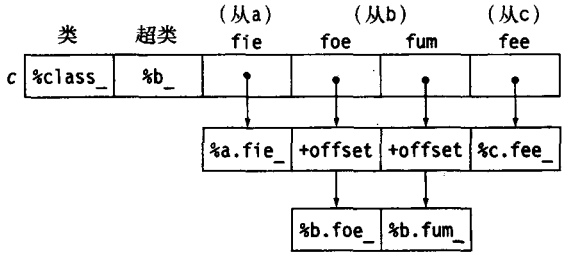


图7-14 使用蹦床函数的多重继承

有两个事实使蹦床函数值得考虑。第一，它们只在对由b继承而来的方法的调用上引发代价。而对a 379
和c的方法的调用不会引发代价。第二，蹦床函数可以比装入和加策略优化得更好。面向对象语言的优化通常包含大量的内联替换。因为蹦床函数是类特定的，所以偏移量以文字常量出现。这消除内存引用；使用内联替换，负荷减小到一个装入立即和一个加法，而这些只出现在对b的方法的调用中。与接受c中所有可视方法的所有调用相比，蹦床函数可能产生更快的代码。

作为最后的调整，编译器也可能需要插入一个类指针的副本。这样，由b编译而来的方法可以在它的b实例变量的前面立即找到这一类指针。这一类指针应该指向c，从而由c中的定义掩盖起来的b中的任意方法都可以解析到正确的方法指针。

3. 对象记录的形态

因为编译器必须在指定的偏移量上放置代码和数据成员，所以这一转换为类创建这样的对象记录，在其中方法和实例变量是交替出现的。这不会引发错误，因为编译器知道每一个引用的正确偏移量。如果c是从b继承而来的，而b又是从a继承而来的，那么类c的对象记录如下所示：

类指针	超类指针	a方法	a数据	b方法	b数据	c方法	c数据

其中，每个类的方法后面紧跟着那个类的类变量。 380

7.11 概括和展望

编译器设计者面临的更微妙的任务之一是为实现每一个源语言结构选择目标机器操作的模式。对几乎所有的源语言语句都有多种实现策略。在编译器设计时所做的特定选择对编译器生成的代码有重要的影响。

在非商用编译器中，即在调试编译器或学生编译器中，编译器设计者也许选择易于实现翻译并产生简洁代码的策略。而在优化编译器中，编译器设计者应该关注于那些给编译器的后期阶段，即低级优化、指令调度以及寄存器分配等展示尽可能多的信息的翻译。这两种不同的观点导致循环的不同形态，命名临时变量的不同规则以及表达式的不同评估顺序。

这一差异的典型例子可能是选择（case）语句。在调试编译器中，可以以联级`if-then-else`机构来实现它。而在优化编译器中，众多的测试和分支的低效率使得更复杂的实现方案更有价值。改进选择语句的努力必须在最初生成IR时就进行；很少有优化器把一系列联级条件语句转化成二分搜索或直接跳转表。

本章注释

本章所包含的素材大致属于两个范畴：为表达式生成代码和处理控制流结构。表达式评估在文献中已有很好的阐述。如何处理控制的讨论却很少；本章中关于控制流的大部分素材来自于非正式的交流、经验和对编译器输出的仔细阅读。

Floyd提出了第一个根据表达式树生成代码的多遍算法[143]。他指出冗余消除和代数重组有可能改进他的算法的结果。Sethi和Ullman[30]提出对于简单机器模型最优的两遍算法；Proebsting和Fischer扩展了这一工作，得到较小的内存等待[278]。Aho和Johnson[5]引入动态规划来寻找最小代价的实现。 381

科学计算程序中的数组计算的优势导致对数组寻址表达式的研究和改进它们的优化（如10.3.3节中

的强度减弱)。7.5.3节中所描述的计算来自于Scarborough 和Kolsky的研究[297]。

Harrison把串操作作为推广内联替换和特化的启发性例子[174]。7.6.4节结束时所提到的例子来自上述论文。

Mueller和Whalley描述不同循环形态对性能的影响[263]。Bernstein给出了选择语句的代码生成中的
382 各种选择的详细讨论[38]。调用约定在各处理器和各操作系统的操作手册中得到最好的阐述。



第8章
代码优化概述

8.1 概述

编译器中间部分，或优化器的目标是把由前端创建的IR程序转换成以更好的方式计算相同结果的IR程序。这里，“更好”可以有很多含义。它通常指更快的代码，但是它也可能指更简洁的代码。程序员也许想要优化器生成在运行时消耗更少电力的代码，或者在考虑资源的限制下运行成本更低的代码。所有这些目标都属于优化的领域。

优化的机会来自方方面面。为了使这一讨论更具体，考虑在实现源语言抽象时可能出现的低效问题。因为前端把源代码翻译成IR时通常不对周围上下文进行大量的分析，它通常生成的是处理某个结构的最一般情况的IR。优化器可以通过执行额外的分析来确定某些上下文；遗憾的是，某些上下文在编译时是不可知的。

考虑当编译器必须为诸如A[i, j]这样的数组引用生成代码时所发生的事情。在没有A、i、j及周围上下文的特别信息的情况下，编译器必须生成对二维数组进行寻址的完整表达式。正如我们在第7章中所看到的那样，这一计算的形式如下所示。

$$\begin{aligned} &address(A) \\ &+ (i - low_1(A)) \times (high_2(A) - low_2(A) + 1) \times size \\ &+ (j - low_2(A)) \times size \end{aligned}$$

其中，*address*是这一数组的第一个元素的运行时地址，$low_i(A)$和$high_i(A)$分别是A的第i维的下界和上界，*size*是A的元素的大小。这一计算成本很高。编译器改进这一代码的能力直接取决于它对这一代码和周围上下文的分析。

如果A是局部声明的，且有已知的边界和类型，那么正如7.5节中所讨论的那样，编译器通常可以在编译时执行这一地址计算的大部分任务，并在运行时使用这一结果。例如，项：

$$(high_2(A) - low_2(A) + 1) \times size$$

可以在编译时计算，并处理为运行时常量（例如，使用loadI）。上述地址计算的其余部分也可以得到改进，参见7.5节。

如果编译器能够识别出对A[i, j]的引用是发生在i和j以有序的方式变化的循环嵌套的内部，那么编译器可以用加法取代上述计算中的整数乘法。这一转换称为*操作并强度减弱*（operator strength reduction）。强度减弱使用序列i_0，i_1，i_2，…取代序列$i \cdot k$，$(i+1) \cdot k$，$(i+2) \cdot k$，…，其中，对于$m>0$，$i_0 = i \cdot k$且$i_m = i_{m-1} + k$。当程序运行时，它执行更多的加法和更少的乘法。如果乘法的代价比加法大，那么这一新序列就是一个改进。

有时候，编译器无法发现改进代码的事实。如果j的值是从诸如文档或键盘这样的外部设备读取的，那么编译器无法改进计算A[i, j]地址的代码。在这样的环境下，编译器必须生成可以处理任意合法情

384 况的代码。它可能还必须生成发现诸如$j > high_2(A)$的不合法情况的代码。

这一简单例子既展示了优化的动机，也展示了代码优化器的基本操作。

> 代码优化的目标是在编译时发现程序的运行时行为的信息，并使用这一信息改进编译器生成的代码。

改进可以采用多种形式。前面的例子都集中于优化的最一般目标：加速已编译代码的执行。然而对于某些应用，已编译代码的大小是重要的。这样的例子包括被提交给只读内存的代码，而内存的大小受到经济上的约束，或者在执行前要在限宽通信信道中传递的代码，而此时代码的大小直接影响完成工作的时间。对于这些应用，优化应该生成占据较少空间的代码。在某些情况下，用户想要按其他标准进行优化，例如寄存器使用、内存的使用、能量消耗或实时事件的响应时间等标准。

历史上，优化编译器主要聚焦于编译代码的运行时速度。我们关于优化的大部分讨论将考虑速度问题。对速度的强调常常带来空间成本的增加，由于复制操作而引发更大的代码。这就是算法设计中典型的时间和空间的权衡问题。鉴于在嵌入式和交互式计算中代码空间和数据空间的重要性的增加，我们将集中讨论某些减小程序对空间的需求，或者至少不显著增加空间需求的技术。

优化是一个巨大、细节性的课题。本章以及后面两章将对这一课题给出概述性的介绍。本章奠定基础工作。其他书籍对此给出了更深、更细致的讨论[262，260，19]。下一节考查LINPACK的一个例程，LINPACK是一个人们熟知的库，它为数值线性代数中的很多算法提供高效实现。下一节利用这一例程揭示深入研究这一问题及代码改进技术所需的一些概念。本章的其余部分给出代码优化中的一个典型问题，即寻找和消除冗余表达式的问题，并使用这一典型问题说明优化的机会和挑战。

接下来的两章更深入地研究程序分析和转换的问题。第9章给出静态分析的总论。它阐述优化编译器必须解决的分析问题，并给出解决这些问题时使用的实际技术。第10章展示单处理器机器上的优化分
385 类，并对各范畴使用一个实例转换来详述这一分类。

8.2 背景知识

直到20世纪80年代初期，很多编译器设计者把优化认为是只有当编译器的其他部分都能很好工作之后才应该加入的性质。这导致*调试编译器*（debugging compiler）与*优化编译器*（optimizing compiler）之间的差异。调试编译器以代码质量为代价强调快速编译。这些编译器对代码不做本质上的重新调整，所以源代码与执行代码之间保留着很强的对应关系。这简化了把运行时错误映射到源代码特定行的任务；因此就有了*调试*编译器的说法。与其相反，优化编译器致力于以编译时间为代价改进可执行代码的运行时间。在编译中花费更多的时间通常产生更好的代码。因为优化器通常要移动操作，所以从源代码到可执行代码的映射更加缺少透明度，而且相应地，调试更加困难。

随着RISC处理器推向市场（以及随着RISC的实现技术被应用于CISC的体系结构），改进运行时性能的大部分负担落到编译器的身上。为了增强性能，处理器设计师转向需要编译器更多支持的性质的研究上。这些性质包含分支上的等待槽、无阻塞内存操作、增设管道使用以及增加功能单元的数量。处理器对程序布局和结构这样的高层次问题以及调度和资源分配的细节这样的低层次问题都变得更加性能敏感。随着由优化所带来的性能差异的增加，对代码质量的期望值也在增加，以至于优化已经成为现代编译器的必要部分。

优化器的例程的加入反过来改变前端和后端的操作环境。优化进一步把前端从性能的考虑分离开来。在某种程度上，这可简化前端生成IR的任务。与此同时，优化改变后端处理的代码。现代优化器假设后端将处理资源分配；因此，它们一般都以对寄存器、内存和功能单元不加以限制的理想机器为目标。从

而，反过来这对用于编译器后端的技术产生更大的压力。

如果编译器要承担运行时效率的责任，那么它们必须包含优化器。正如我们将看到的那样，优化的工具还在编译器后端扮演重要的角色。出于这些原因，在讨论用于编译器后端的技术之前引入优化并揭示优化引发的问题是非常重要的。 386

8.2.1 LINPACK的一个例子

为了理解优化实际程序所引发的某些问题，考虑图8-1所给出的代码片段，这一代码片段来自于LINPACK数值库中的dmxpy例程的FORTRAN版本。这一嵌套隐藏着围绕单一长赋值的两个循环，而这一长赋值形成对向量x和y以及矩阵m计算y+x × m的例程的核心。从两个不同观点考虑这一代码是有益的：第一个观点是设计者尝试改进性能的手工转换；第二个观点是，编译器为了在特定处理器上高效运行而翻译这一循环嵌套时所面临的挑战。

```
      subroutine dmxpy (n1, y, n2, ldm, x, m)
      double precision y(*), x(*), m(ldm,*)
        ...
      jmin = j+16
      do 60 j = jmin, n2, 16
         do 50 i = 1, n1
            y(i) = ((((((((((((((( (y(i))
     $              + x(j-15)*m(i,j-15)) + x(j-14)*m(i,j-14))
     $              + x(j-13)*m(i,j-13)) + x(j-12)*m(i,j-12))
     $              + x(j-11)*m(i,j-11)) + x(j-10)*m(i,j-10))
     $              + x(j- 9)*m(i,j- 9)) + x(j- 8)*m(i,j- 8))
     $              + x(j- 7)*m(i,j- 7)) + x(j- 6)*m(i,j- 6))
     $              + x(j- 5)*m(i,j- 5)) + x(j- 4)*m(i,j- 4))
     $              + x(j- 3)*m(i,j- 3)) + x(j- 2)*m(i,j- 2))
     $              + x(j- 1)*m(i,j- 1)) + x(j) *m(i,j)
50       continue 60 continue
        ...
      end
```

图8-1 LINPACK中dmxpy的摘录

在设计者动手转换代码之前，这一循环嵌套执行一个上述计算的更简单形式，其计算如下：

```
   do 60 j = 1, n2
      do 50 i = 1, n1
         y(i) = y(i) + x(j) * m(i,j)
50    continue
60 continue
```

387

为了改进性能，外循环已经被展开（unrolled）。在原来循环体中的语句是y(i)=y(i)+x(j)*m(i，j)。对于j的不同值，这一循环体创建范围从j到j－15的16份拷贝。外循环的增量从1变成16。这16个循环体被嵌入到单一语句中，消除15个加法（y(i)+…的各个出现）以及y(i)的大部分装入和存储。

图8-1所示的循环嵌套的前面还有其他四个循环嵌套版本。这些循环嵌套处理当n2不是16的倍数的情况。具体的行为是，它们最多处理m的15行，使j成为一个使n2-j是16的整数倍的值。第一个循环处理m的单一行，对应于n2为奇数的情况。其他三个循环嵌套处理m的2、4和8行。这保证如图8-1所示的最后一个循环嵌套可以一次处理16行。

理想地，编译器应该能够把原来的循环嵌套转化成更高效的版本，或者转换成最适合于给定目标机器的某种版本。然而，这需要只有少数编译器所拥有的技术组合。对于dmxpy的情况，程序员重写代码

来保证最终代码的形态将使编译器生成高效的目标机器代码。

8.2.2 优化的各种考虑

程序员运用这些转换在于他们相信他们应该可以使程序运行的更快。程序员还必须相信他们将保留程序的意义。(毕竟，如果转换不需要保持意义，那么为什么不用一个nop取代整个过程呢？)

安全性和有效性这两个问题是每一种优化的核心。编译器必须有一种证明转换的每一次应用都是安全的机制，即它保持程序的意义。编译器必须有理由相信转换的应用是有效的，即，它改进程序的性能。如果这两个情况不都是真的，也就是说运用这一转换将改变程序的意义或将使得程序的性能更差的话，那么编译器就不应该运用这一转换。

1. 安全性

388 程序员是如何知道这一转换是安全的呢？也就是说为什么程序员相信转换后的代码产生与原代码相同的结果呢？对循环嵌套的仔细检查表明连续迭代之间的相互作用只通过y的成员出现。

定义安全性

正确性是编译器必须满足的最重要的准则，编译器所产生的代码必须与输入程序有相同的意义。优化器每次运用一个转换时，这一转换动作必须保持这一转换的正确性。

意义（meaning）一般定义为程序的可观察行为。对于一个批处理程序，当它停止后，可观察行为就是内存的状态以及这一程序生成的输出。如果这一程序终止，那么在这一程序停止之前的瞬间，所有可视变量的值应该在任何翻译方案下都是相同的。对于一个交互式程序，这一可视行为更复杂并更难刻画。

Plotkin把这一概念形式化为可观察等价（observational equivalence）。

> 对于两个表达式M和N，我们说M和N是可观察等价的，当且仅当在任意使M和N封闭的上下文C中，(也就是说，没有自由变量)，评估$C[M]$和$C[N]$或者产生相同的结果，或者都不终止[275]。

因此，两个表达式可观察等价，如果它们对于可视的外部环境的影响是相同的。

在实践中，编译器使用的等价概念比Plotkin的概念简单且宽松，即如果两个不同的表达式e和e'在它们实际的程序上下文中产生相同的结果，那么编译器就可以用e'取代e。

这一标准比Plotkin的标准宽松。它只考虑在程序中实际出现的上下文；使代码适合上下文是许多优化机会的源泉。这一标准不论及当一个计算出现错误，或发散时发生的事情。

在实践中，编译器留意不引起发散的情况，即源代码能够正确工作，而优化代码却试图作除数为零的除法或无穷循环的情况。相反的情况，即源代码发散而优化代码却不发散的情况。

389

- 计算为y(i)的值直到外循环的下一次迭代不被复用。内循环迭代相互独立，因为每一次迭代都刚好定义一个值，而且没有其他迭代引用这个值。因此，迭代可以以任意顺序执行。(例如，我们可以反转内循环，从n1到1运行它，而不改变结果。)
- 通过y的相互作用的效应受到限制。y的第i个元素累加内循环所有第i次迭代的和。这一累加的模式可以在展开的循环中安全地重新生成。

优化过程中所做的大部分分析都朝着证明转换的安全性方面努力。

2. 有效性

为什么程序员可以认为展开循环会改进性能呢？也就是说，为什么这一转换是有效的呢？展开循环的几种不同效应可能提高代码的速度。

- 循环迭代总数减小到了十六分之一。从而减小循环控制带来的额外操作：加法、比较、跳转和分支。如果循环执行成千上万次，那么这些节约将变得很重要。

这一论点暗示更大因数的展开。有限资源的限制也许要求我们选择16。例如，内循环对所有内循环迭代都使用x的16个值。很多处理器仅有32个可以保存浮点数的寄存器。用2的下一个幂32展开将保证x的这些“循环不变量”值不能保存在寄存器中。这将把内存操作加到内循环，有可能抵消展开带来的节省。

- 数组地址计算包含复制工作。考虑y(i)的使用。对于x和m的每一次相乘，原来的代码计算一次y(i)的地址；而转换后的代码是每16次乘法计算一次这一地址。展开代码只做1/16次y(i)的寻址工作。对m的16次引用和对x的更少引用将包含循环可以计算一次并复用的公共部分。 390
- 假设x值留在寄存器内，那么忽视地址计算，展开内循环对于17次装入和1次存储要执行16次乘法和16次加法。对于两次装入和1次存储原来的代码执行1次乘法和1次加法。这一转换后的循环似乎不可能达到内存边界。⊖这一循环有充足的独立算术来重叠操作并隐藏某些等待时间。

展开对其他机器相关的其他效应也有帮助。它增加内循环中代码总量；这可能使指令调度器更好地隐藏等待时间。如果循环末尾分支有较长的等待时间，那么展开可以使编译器填充那个分支的所有等待槽。在某些处理器上，未使用的等待槽必须由nop填充。在这种情况下，展开可以减少发行的nop总数量；从而减少总的内存冲突，而且有可能减少用于执行这一程序的能量。

3. 风险

如果改进性能的转换使得编译器更难生成好的程序代码，那么所有那些潜在的问题都应认为是有效性问题。在dmxpy上执行的手工转换给编译器带来如下新挑战：

- *对寄存器的要求*：原来的循环只需要少量的寄存器来保存它的活动值。只有x(j)和x、y和m的地址计算的某个部分，以及循环索引变量在循环迭代过程中需要寄存器，而y(i)和m(i, j)则暂时性地需要寄存器。相反，转换后的循环在循环期间把x的16个元素保存在寄存器中，另外还有m和y(i)的16个值暂时性地需要寄存器。
- *地址计算的形式*：原来的循环处理对应于y、x和m的3个地址。因为转换后的循环在每一次迭代引用更多的不同位置，所以编译器必须小心塑造地址计算的形式以避免重复计算和对寄存器的额外要求。在最坏的情况下，代码可能对x的所有16个元素、m的所有16个元素和y的一个元素使用独立的计算。 391

如果编译器适宜地塑造地址计算形式，它可以对m和x各使用一个指针，每个指针带有16个常量值偏移量。它可以重写循环以把这一指针用于尾端循环测试，回避对另一个寄存器的需要，并消除另一次更新。计划和优化在这两种情况下产生差异。

还存在与机器相关的其他问题。例如，在每次迭代中，需要小心地调度17次装入、1次存储、16次乘法、16次加法再加上地址计算以及循环负荷的操作。编译器可能需要在前一次迭代发行其中的若干装

⊖ 为了确定这一循环是否实际达到内存边界需要有关目标处理器的更加详细的知识，其中包括各操作的等待时间以及它在一个循环能够发行的操作种类和数量。

392 入操作，使得它能够适时地调度初始浮点操作。

8.2.3 优化的机会

正如我们所看到的那样，简单循环的优化任务就可能包含很多复杂的考虑事项。优化编译器一般利用来自于不同源头的机会：

1. 减小抽象的负荷

正如在本章一开始我们所看到的数组地址计算那样，程序设计语言所引入的数据结构和类型需要运行时支持。优化器使用分析和转换来减小这种负荷。

2. 利用特殊情况

通常，编译器可以使用操作执行的上下文信息来特化该操作。作为例子，C++编译器有时可以确定对于一个虚函数的调用总使用相同的实现。在这种情况下，它可以重新映射这一调用并减少每次调用的代价。

3. 匹配处理器资源

如果一个程序的资源需求不同于处理器所提供的资源，那么编译器可以转换这一程序使它的需求与处理器的能力更接近。运用于dmxpy的转换就有这种效应；它们减少对于每一个浮点操作的内存存取数量。

总而言之，这些问题涉及的范围很广。当我们在第9章和第10章中讨论特定分析和转换技术时，将使用更详细的例子来讨论这些领域的问题。

8.3 冗余表达式

作为一个具体的例子，考虑寻找并消除基本块内部的冗余表达式问题。一个基本块是直线式无谓词代码的最长片段。为了使问题简单化，我们的算法将忽视在基本块前后发生的任何事情。一个表达式x+y在一个块里是冗余的，如果它在这一块里已被计算过，而且没有再定义x或y的中间操作。如果编译器发现一个冗余表达式，那么它可以保存第一次计算时的那个值，并用对那个值的引用取代后来的评估。

我们可以在源代码级别上处理这一问题，如图8-2所示。图8-2a中的源代码两次计算2 × y。图8-2b中的代码给出它的重写代码以避免复制操作。这一重写代码有更多语句，但有较少的操作。当编译器把它翻译成目标代码时，较低的操作计数一般将产生较快的代码。程序员可以避免自己去编写包含这样的冗余表达式的代码。而预处理器及生成源代码的其他工具则比程序员更有可能创建这些冗余代码。这样的表达式还可能大量出现在如把源代码翻译成较低级IR期间所做的地址计算等计算中。编译器也可以把冗余消除技术用于低级IR上。

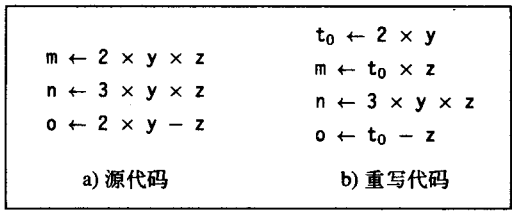
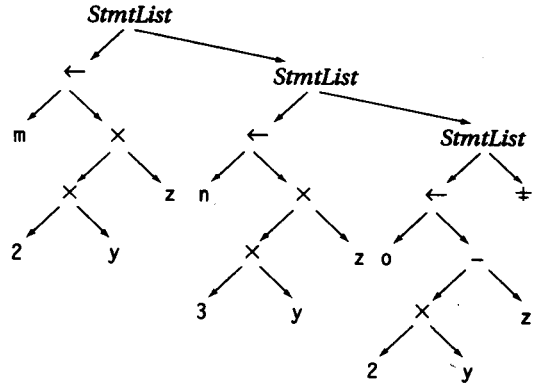


图8-2 源代码级冗余表达式

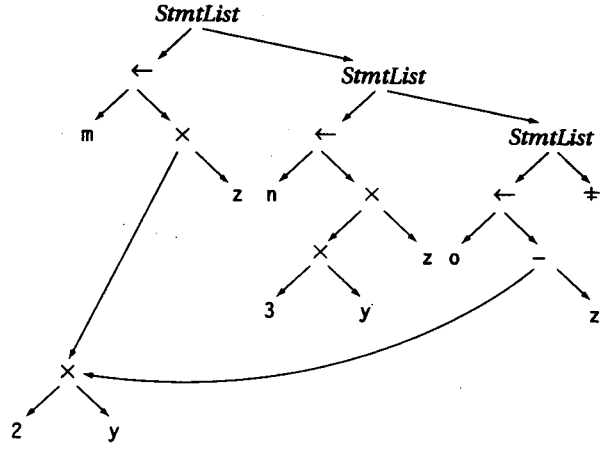
为了自动发现并消除冗余，编译器需要寻找冗余的算法以及重写无冗余代码的算法。我们将给出解决这一问题的两个不同方法：构建DAG和计算值编号。 393

8.3.1 构建有向无环图

明确地表示冗余计算的一个方法就是使用有向无环图（directed acyclic graph，DAG）。在AST中，每个结点至多有一个父结点。因此，图8-2a中的示例代码的AST形式如下：

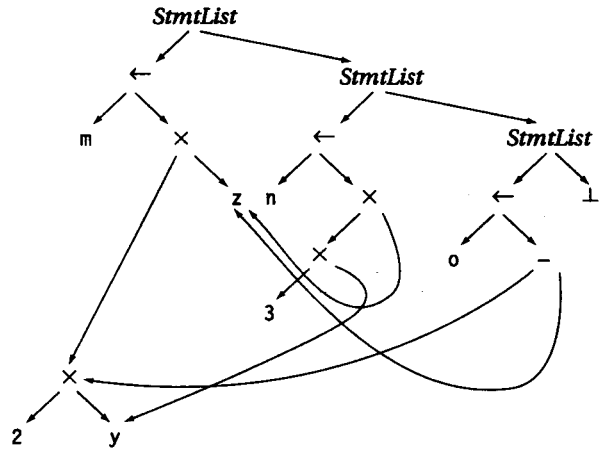


相反，DAG仅一次表示每个不同的表达式。在DAG中，一个结点可能有多个父结点。每一个父结点表示对这一结点表示的值有一个引用；带有多个父结点的任意结点一定是冗余表达式。由前面AST构建的DAG如下所示：



这一DAG揭示出这样的事实：子表达式2 × y在源代码中出现两次。表示2 × y的结点有两个父结点：第一个赋值中的乘法和第三个赋值中的减法。编译器可以生成利用这一事实的代码（与图8-2b类似）。把DAG用作冗余消除工具的关键是系统、高效地定位这些相同的表达式。 394

实现这一目标的最简单方法是修改编译器构造AST的例程。如果构造器使用散列来发现相同子树，那么它将构建每个不同表达式有一棵子树的DAG。这将产生理想的2 × y共享。它还将发现y的所有实例有相同值，而且z的所有实例也有相同值。因此，实际的DAG将比前面的DAG更复杂，如下图所示。



这一DAG使用更少的结点来表示相同的三个语句。分析树有24个结点，第一个DAG有21个结点，而最后的DAG有18个结点。（如果目标是缩小AST，那么这个思想发挥作用。）

在这种情况下，2 × y的两个实例有相同的值。然而，这是一种偶然而非精心设计的。基于散列的构造器对于相等使用文本概念，所以y等于y，与值无关。构造器保证两个操作共享一个表示，如果它们有
395 相同的操作符且它们的操作数有相同的表示。考虑如果我们把第二个赋值的左部从n改成y，那么将会发生什么呢？用于把y与y匹配的文本机制无法确定一个中间赋值是否改变了y的值。因此，修改后的例子的DAG仍然共享2 × y的表示，即使两次出现可能有不同的值。如果我们的目标是识别一定计算相同值的子表达式，那么我们需要考虑赋值影响机制。

为了使基于散列的DAG构造器反映赋值，我们可以跟踪每个变量的不同版本。这一机制很简单：把一个计数器与每个变量相关联，并在每个赋值处增加被重新定义变量的计数器。在基于散列的构造器中，在散列之前给每一个变量名附加一个计数器。通过这些修改，基于散列的构造器将构建一个DAG，在这个DAG中两个表达式有相同的表示当且仅当这两个表达式在文本上相同且用于这一表达式中任意变量在这一表达式的两次出现之间都不被重新定义。（我们使用5.5节和9.3节中描述的一种IR，即SSA来形式化这一计数器方案。）

指针赋值

给y赋一个新值的效应有限，所以下标方案能很好地进行模型化。指针赋值则有更加广泛的效应。诸如C语言中的*p=0这样的赋值必须递增它可能修改的每一个变量的下标。如果编译器对指针的值有详细的信息，那么它可能限制这一效应并递增一小组下标。然而，如果编译器对p指向何处知之甚少，那么它必须递增指针赋值可能修改的每一个变量的下标，这有可能是整个程序中的每一个变量的下标。

这似乎有些极端，但是这却反映了编译器能够得到的歧义性指针操作对事实集合的实际影响。如果源语言允许任意的指针算术，那么这就涉及存储在内存中的每一个变量。不允许任意的指针算术，编译器可能把影响限定于指针能够达到的那些变量上，有时这可以用这样的变量集合来近似，该集合中的变量的地址在程序中被使用。执行指针分析的一个主要动机是缩小这些集合的大小。

在对代码使用树型表示的编译器中，把DAG构建机制直接嵌入树构造器中具有优势。这产生更小的IR程序，以及伴随而来的对编译的好处。它使冗余被显式地表示出来：带有多个父结点的任意操作一定是冗余的。最后，它保证这一思想得到彻底的应用。当编译器的其余部分处理IR时，例如把2 × x × y重
396 写作x × y+x × y时，这一散列构造器将保证结果IR直接反映共用子表达式。

顺序的重要性

表达式的特定书写顺序对优化算法分析及转换这些表达式的能力有直接的影响。考虑用于说明DAG构造法的赋值序列。

$$m \leftarrow 2 \times y \times z$$
$$n \leftarrow 3 \times y \times z$$
$$o \leftarrow 2 \times y - z$$

基于散列的DAG构造算法发现$2 \times y$在第一和第三个赋值中重复出现。它没有注意到$y \times z$分别在第一和第二个赋值中出现两次。按左结合处理乘法隐藏了$y \times z$的重复使用。按右结合处理乘法揭示出 $y \times z$，但是隐藏$2 \times y$。

使用交换律、结合律和分配律对表达式重新排序可以改变优化的结果。例如，考虑表达式$3 \times a \times 5$。使用二元×的任意表示只有两种可能的组合方式:

$(3 \times a) \times 5$和

$3 \times (a \times 5)$。

无论哪种顺序都无法把两个常量组合到一起。因此，编译器在运行时评估的表达式3×5从不在考虑之列。

因为存在很多重排序较大表达式的方法，所以编译器使用启发式技术来搜索更好的表达式排序。例如，IBM FORTRAN H编译器生成它的所有数组地址计算，因为这可能用于改进其他优化。其他编译器对可交换操作的操作数排序，这一顺序对应于这些操作数被定义的循环嵌套层次。因为存在如此多的解决方案，所以编译器设计者必须尝试不同的想法以便确定对于特定的语言、编译器和编码风格什么是合适的。

397

8.3.2 值编号

把冗余编码到树结点的概念不适用于使用线性IR的编译器。对于线性IR，编译器设计者可能需要分析线性IR并重写IR以消除冗余的技术。实现这一目标的经典技术称为*值编号*（value numbering）。想法很简单。算法为运行时计算的每个值指定一个不同的数，使其具有这样的性质：两个表达式e_i和e_j有相同的值编号当且仅当e_i和e_j对于这两个表达式的所有可能的操作数都肯定相等。

值编号依赖于对DAG构造法的相同观察：编译器可以使用散列表来识别产生相同值的表达式。“冗余”的操作定义稍许不同；如果它们有相同的操作符且它们的操作数有相同的值编号，则两个表达式被认为是冗余的。为了把变量、常量和已计算的值映射到它们的值编号上，编译器使用一个散列表。对于一个变量或常量，编译器可以把它的扫描文本用作它的散列关键字。对于一个包含操作符的表达式，编译器可以从这个操作符和它的操作数的值编号构建散列关键字；相同的散列关键字必定产生相同的运行时值。

当计算新值时，这些值得到新的值编号。因为散列关键字使用操作数的值编号，而不是它们的名字，所以这一算法以一种自然的方式处理非歧义性赋值。在序列中，

$$x \leftarrow a + d$$
$$y \leftarrow a$$
$$z \leftarrow y + d$$

y得到与a相同的值编号，所以y+d得到与a+d相同的值编号。涉及指针或数组元素等的歧义性赋值仍然有着灾难性的效应，因此必须废除指针可能达到的所有变量的值编号。

398 图8-3给出基本的值编号算法，它假设以形式为$result_e \leftarrow operand_1\ op_e\ operand_2$的表达式为输入。这一算法开始于一个空的值表。为了得到一个操作数的值编号，算法在这一值表中查找这一操作数。如果存在一个条目，算法就使用已指定给这一条目的值编号。如果不存在这一条目，那么算法创建一个条目并给这个条目指定一个新的值编号。一旦它有了所有操作数的值编号，算法为上述整个表达式构造散列关键字，并使用这个关键字在表中查找这一表达式。如果存在一个条目，那么这一表达式e是冗余的。如果不存在这样的条目，那么这一表达式不是冗余的，因此算法使用表达式的新值编号在值表中登记这一表达式。它还在$result_e$的条目中记录这一值编号，$result_e$是由这一表达式定义的名字。把这一算法扩展到任意元的表达式是直截了当的。

for each expression e in the block of the form
$result_e \leftarrow operand_1\ op_e\ operand_2$

1. *get the value numbers for $operand_1$ and $operand_2$*
2. *construct a hash key from the operator and the value numbers for $operand_1$ and $operand_2$*
3. *if the hash key is already present in the table then*
   replace expression e with a copy operation and
   record the value number for $result_e$
   else
   insert the hash key into the table
   assign the hash key a new value number and
   record that value number for $result_e$

图8-3 值编号化一个块

为了弄明白这一算法是如何工作的，考虑下面的例子。列a给出值编号化之前的一个短基本块。

$a \leftarrow b + c$	$a^3 \leftarrow b^1 + c^2$	$a \leftarrow b + c$
$b \leftarrow a - d$	$b^5 \leftarrow a^3 - d^4$	$b \leftarrow a - d$
$c \leftarrow b + c$	$c^6 \leftarrow b^5 + c^2$	$c \leftarrow b + c$
$d \leftarrow a - d$	$d^5 \leftarrow a^3 - d^4$	$d \leftarrow b$
a) 源代码	b) 值编号代码	c) 重写代码

列b给出这一算法指定给每个名字的值编号。最初，对于空值表，b和c得到新的值编号。使用b的值编号（即1）和c的值编号（即2）以及操作符“+”，值编号算法给第一个表达式创建一个文本串“1+2”，并将其用作散列关键字，或某个散列函数的输入。把这个散列函数的输出作为索引，值编号算法在这个值表
399 中寻找这个散列关键字。寻找失败，因此算法为这一散列关键字创建一个条目，并给它指定值编号3。为了保证对a的后继引用发现它的正确值编号，算法还为a创建一个条目并给它指定值编号3。对每个操作依次重复这一过程产生上图中上标所示的值编号。

值编号正确地展示由于中间的b的再定义b+c的两次出现产生不同值。另一方面，a-d的两次出现产生相同的值，因为它们有相同的输入值编号和相同的操作符。这一算法发现这一事实并通过对b和d指定相同的值编号5来记录这一事实。利用这样的事实，编译器可以如列c所示那样重写代码。编译器的后继遍可能消除拷贝d←b。（我们可以给值编号算法增加一个机制，使用b取代对d的后继引用。这将不再需要拷贝操作。作为软件工程的一个核心问题，我们愿意插入一个拷贝并假设后来的编译器遍将消除它。）

1. 扩展值编号算法

值编号算法是执行其他若干局部优化的自然场所。因操作数的顺序的不同而不同的可交换操作应该得到相同的值编号。为了处理这一工作，编译器可以使用适当的方案对每个可交换操作符的操作数排序，例如用值编号排序这些操作数。同样地，如果算法知道某个操作的所有操作数是常量且算法有它们的值，那么它可以在编译时执行这一操作并把答案直接叠入代码中。为了处理这一工作，编译器设计者需要在值表中加入标明常量值的表记法。当编译器发现一个常量值表达式时，它就会评估这一操作，给其结果指定一个值编号，并用它来取代对该常量值的任意后继引用。

我们还可以扩展值编号算法来发现不产生影响的操作，算法运用某些代数等式。例如，x+0应该与x得到相同的值编号。为了处理代数等式，这一算法需要特殊情况代码来发现每一个等式。对应一个等式的一系列测试很容易变得太长，从而使值编号过程的速度慢得令人无法接受。取而代之的是，应该把这些测试组织成以操作符为第一个测试开关的一棵树，这可以保证检查等式的负荷较小。下面的表给出用这种方法可以处理的一些等式。 400

值编号的代数等式			
$a + 0 = a$	$a - 0 = a$	$a - a = 0$	$2 \times a = a + a$
$a \times 1 = a$	$a \times 0 = 0$	$a + 1 = a$	$a + a = 1, a \neq 0$
$a^1 = a$	$a^2 = a \times a$	$a \gg 0 = a$	$a \ll 0 = a$
a AND a = a	a OR a = a	MAX(a,a) = a	MIN(a,a) = a

一个聪明的实现器会发现其他等式，其中包括某些特定类型的等式。计算带有相同值编号的两个值的异或将产生适当类型的零。还有，在IEEE浮点格式中的数具有某些由∞和NaN（非数字）的显式表示引入的特殊情况；例如∞ − ∞ = NaN，∞ − NaN = NaN和∞ ÷ NaN = NaN。

图8-4给出带有这些扩展的算法。步骤2评估并叠入常量值操作。步骤3检查因实现代数等式而可以消除的操作。步骤4对可交换操作的操作数重新排序。步骤1、5和6是从原来的程序延续过来的。即使使用这些扩展，对每个IR操作的代价仍然维持在较低的水平。每一步骤都有高效的实现。

for each operation e in the block of the form
$result_e \leftarrow operand_1\ op_e\ operand_2$

1. *get the value numbers for $operand_1$ and $operand_2$*
2. *if all the operands to op_e are constant*
   evaluate it and replace later uses with references
3. *if the operation is an identity*
   replace it with a copy operation
4. *if op_e is commutative*
   sort the operands by their value numbers
5. *construct a hash key from the operator and*
   the value numbers for $operand_1$ and $operand_2$
6. *if the hash key is already present in the table then*
   replace operation e with a copy operation and
   record the value number found for $result_e$
   else
   insert the hash key into the table
   assign the hash key a new value number and
   record that value number for $result_e$

图8-4 扩展值编号 401

2. 命名的角色

变量和值的名字的选择可能会限制值编号的效果。考虑当上述算法用于下面的短块时所发生的情况:

源代码	值编号代码	重写代码
$a \leftarrow x + y$	$a^3 \leftarrow x^1 + y^2$	$a \leftarrow x + y$
$b \leftarrow x + y$	$b^3 \leftarrow x^1 + y^2$	$b \leftarrow a$
$a \leftarrow 17$	$a^4 \leftarrow 17^4$	$a \leftarrow 17$
$c \leftarrow x + y$	$c^3 \leftarrow x^1 + y^2$	$c \leftarrow x + y$

这一算法处理第一个运算并给它指定值编号3。当算法遇到第二个操作时，它发现对于（来自于x+y的）散列关键字“1+2”已存在一个表条目，所以它为b创建一个条目，并为它指定已赋的关键字3。这一操作被b←a取代。

接下来，算法处理第三个操作。为常量17和a都指定值编号4。最后，算法处理最后一个操作。它发现这一表达式是冗余的，并带有值编号3。然而，这个值在a中的已记录实例不再存在，因为第三个操作已经覆写了它。这一算法不能消除x+y的实例，因为它已经无法跟踪这个值的驻留地。

我们可以用两个不同的方法处理这一问题。算法可以构建并维护从值编号到名字的映射。每一个赋值必须更新这一映射。另外，编译器也可以按给每一个赋值制定不同名字的方式重写代码。为了满足惟一性，只要给每一个名字加上一个下标就可以了。（如同DAG构造法中赋值的版本号一样，这一方案类似于SSA形式的一个性质。）在这一新的命名规则下，上例的块变成:

$$
\begin{aligned}
a_0 &\leftarrow x_0 + y_0 \\
b_0 &\leftarrow x_0 + y_0 \\
a_1 &\leftarrow 17 \\
c_0 &\leftarrow x_0 + y_0
\end{aligned}
$$

402

使用这些新名字，代码对每个值刚好定义一次。因此，没有值被再定义或失去定义，或*杀死*（killed）。当我们把这一算法运用到这个块上时，它产生理想的结果。操作2和操作4被证明是冗余的；每一个都可以用从a_0的拷贝操作取代。然而，这一新的命名方案使我们可以做得更好。因为每一个值编号有惟一的名字，我们可以省略拷贝操作$b_0 \leftarrow a_0$和$c_0 \leftarrow a_0$。

优化中的这一改进也有相应的代价。在我们能够执行这一代码之前，我们必须与程序的旧意义协调这一新名字空间。特别地，编译器可能需要插入某些操作，以便在控制流路径合并点处协调名字。如果有两个路径进入一个块，a的沿着一条路径的最新定义是a_{13}而沿着另外一条路径是a_{17}，那么编译器也许需要创建一个新名字，例如a_{23}，来表示a_{13}和a_{17}的合并，并在一个前驱块的末尾插入拷贝操作$a_{23} \leftarrow a_{17}$，在另一个前驱块的末尾插入拷贝操作$a_{23} \leftarrow a_{13}$。我们将在9.3节中看到执行这种拷贝插入的算法。

8.3.3 冗余消除的经验

DAG构造法和值编号都分享代码改进的大多数基于编译器的技术的典型特性。这两个方法都必须阐明两个问题。

1. 发现机会

这两个技术中的第一步都是要检查块中的每一个表达式，并确定是否已查到一个等价的表达式。DAG构造法为每个由操作符（+，×等等）和它的操作数的名字组成的每一个操作创建一个抽象名字。它使用这一抽象名字为进入散列表的关键字。如果已经存在这一抽象名字的一个条目，那么这一表达式是冗余的。值编号使用类似的机制，但是在很大程度上又不相同。DAG构造法认为具有相同名字，即相

同拼写的操作数是相同的。值编号算法认为具有相同值编号的操作数是相同的。这使得它通过一个拷贝操作来跟踪一个值，如在序列x←a+d、y←a和z←y+d中那样。基于散列、使用词法相等概念的构造器无法发现a+d和y+d一定计算相同的值。

2. 转换代码

一旦编译器证明一个表达式是冗余的，它必须修改IR以便利用这一事实。在DAG构造法中，编译器直接把这一事实编码在DAG的结构中。因为目标就是保证冗余表达式的每个实例有一个结点，通过记录 403
适当的结点名，编译器在散列表中记录这一冗余。当后继操作引用这一操作时，它将找到这一实例。值编号通过重写操作来进行操作。当它发现冗余操作时，它用一个拷贝操作取代这一操作。

这种两部结构出现在很多优化中。这些技术有通过分析代码来寻找可以安全运用转换的机会的部分，以及通过修改IR执行这一转换的部分。下一节给出分析和转换讨论的背景。8.5节采用值编号技术并把它扩展到比基本块更大的区域上的操作。8.6节探讨完整过程上的冗余消除。这里不再扩展值编号，我们将使用一个计算可用表达式（available expression）的算法，并使用其结果来重写代码，可用表达式是数据流分析中的一个典型问题。这一方法发现与值编号不同的改进。

8.4 优化作用域

前面一节描述的两个技术都操作于基本块上。正如我们所看到的那样，这两个技术收集有关语句前面的上下文信息并利用这一信息试图改进程序。很多优化通过得到上下文知识并利用这一知识改进或特化代码。因此，一个技术所考虑的上下文总量在描述、实现和理解该技术中起着重要的作用。分析和转换一般都属于下面五个范畴，或作用域中的一个：局部、超局部、区域、全局及整个程序。

8.4.1 局部方法

这些方法把它们的注意力限制在基本块上。局部方法通常是最简单的分析和理解方法。图8-5给出包含七个基本块A、B、C、D、E、F和G的代码片段。每一个基本块都结束于一个跳转或分支。

在一个基本块内部有两个重要的特性。第一，按顺序执行语句。第二，如果任意语句被执行，则整 404
个基本块被执行。⊖这两个性质使编译器能够使用相对简单的分析证明比更大作用域所能证明的更强的事实。因此，局部方法有时可以产生更大作用域不能轻易得到的改进。然而，局部方法被局限于改进出现在相同块中的操作。

8.4.2 超局部方法

超局部方法操作于扩展的基本块上（EBB）。一个EBB β是一组块β_1，β_2，…，β_n，其中β_1可能有多个前驱，而其他β_i，$2 \leqslant i \leqslant n$在EBB中有惟一前驱。这些块$\beta_i \in \beta$形成一棵只能从它的根$\beta_1$进入的树。$\beta$可以有多个出口，这些出口是不以$\beta$中的块为目标的某个$\beta_i$尾部的分支或跳转。

图8-5中的例子有三个极大EBB：{A，B，C，D，E}，{F}和{G}。第一个EBB包含三条不同的路径，{A，B}、{A，C，D}和{A，C，E}。后两个EBB是由一个块组成的平凡EBB。然而，因为F和G都有多个前驱，所以它们不能包含在它们的前驱EBB中。

⊖ 运行时异常可以中断一个块的执行。典型地，异常促使到异常处理程序的控制转移。这个异常处理程序可能处理这一问题，把控制返回到这一个块，并再次执行引发这一异常的操作。另外，这个异常处理程序也可能终止执行。对于前一种情况，编译器必须明白哪些操作可能引发异常以及异常处理程序的副作用。后面的情况对优化程序来说是透明的，因为执行非正常结束。

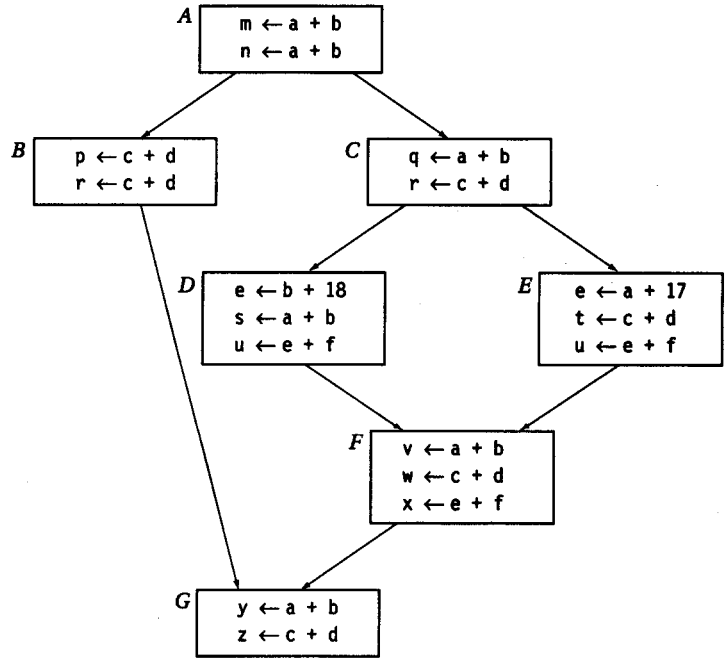


图8-5 代码片段举例

在一个EBB内部，编译器可以利用在前面的块中发现的事实改进后面块中的代码。超局部方法可以把通过EBB的各个路径处理成好像它们在一个块中那样。在这个例子中，超局部方法可以使用在*A*中发现的事实改进*B*、*C*、*D*或*E*。在值编号中，编译器可以把处理*A*的结果用作*B*和/或*C*的开始点。编译器可以把处理*A*和*C*的结果用作*D*和/或*E*的开始点。这些额外的事实揭示转换和改进的额外机会；较大的上下文通常导致较大的一组机会。

8.4.3 区域方法

405 这些方法操作的范围要比单一EBB大，但比整个过程小。在这一例子中，编译器可能发现把诸如所有7个块都考虑成一个区域{*A*，*B*，*C*，*D*，*E*，*F*，*G*}的优势，或者诸如{*A*，*B*，{*C*，*D*，*E*，*F*}，*G*}这样的层次嵌套分组区域的优势。这将允许编译器例如在考虑*F*时使用从*C*、*D*和*E*得来的事实。

区域方法集中于比过程小的作用域上。在某些情况下，这产生能够引发转换的更强大的分析。例如，在一个循环嵌套的内部，编译器也许能够证明一个频繁使用的指针是不变量（取单一值），即使它在过程中其他地方被修改。这样的知识可以带来诸如把这一指针所引用的值保留在寄存器内这样的优化。

编译器可以用很多方法选择区域。一个区域可能是由某个源代码控制结构来定义的，例如一个循环
406 嵌套；另外，它也可能是诸如支配关系这样的图论性质所定义的控制流图的一个子集（参见8.5.2节）。

- 基于循环的方法遵循一个简单的设计：把优化的努力集中于那些最频繁执行的区域上。平均说来，循环体比包围循环体的代码执行的次数要多，所以集中于循环体的优化更有意义。其他常用区域包括支配树的子树和静态单一赋值图中的环（参见9.3节和10.3节）。

区域方法和超局部方法在处理CFG中的合并点上是有差异的。在合并点处，区域方法必须具有合并在进入这一合并点的每一条路径上为真的事实集合的策略。区域方法与全局方法的差异在于区域方法只考虑过程的一个子区域。把优化集中于限定的区域可能是强有力的，也可能是有局限性的。

8.4.4 全局方法

这些方法也称为过程内（intraprocedural）方法，它检查整个过程。全局方法的动机很简单：局部化的优化决策可能在某些更大的上下文产生不好的结果。过程为分析和转换提供自然的边界。过程是封装和隔离运行时环境的抽象。同时，在许多系统中，它们又充当分块编译的边界。

全局转换几乎总是需要全局分析。数据流分析促使我们去迎接这一挑战。因此，全局技术通常包含某种过程内分析以收集事实，并通过运用这些事实来确定特定转换的安全性和有效性。全局方法发现局部方法无法发现的改进机会。

8.4.5 完整程序方法

这些方法也称为过程间方法（interprocedural method），它们把整个程序考虑为它们的作用域。正如从局部作用域到全局作用域的移动揭示出新机会那样，从单一过程到完整程序的移动也将揭示出新机会。它也带来新挑战。着眼于整个程序，编译器遭遇来自在单一过程内不存在的名字作用域规则和参数绑定所带来的复杂性和限制。

我们把包含多个过程的任意转换分类为过程间转换。在某些情况下，这些技术分析整个程序；在其 407
他情况下，编译器可能只检查源代码的一个子集。过程间优化的两个典型例子是内联替换，这样的替换使用被调用过程体的拷贝替换这一过程的调用，以及过程间常量传播，它在整个程序间传播和叠入有关常量的信息。

8.5 比基本块大的区域上的值编号

冗余可能出现在不同基本块的表达式中。8.3节中所给出的技术都不发现这些情况；两个技术都把注意力限制在一个块上。本节给出两个把值编号扩展到更大区域的方法：首先是扩展到扩展基本块，然后是扩展到更大的区域。对于值编号，转移到更大区域允许转换消除更多的冗余表达式。这一般导致更快的代码。

8.5.1 超局部值编号

为了改进值编号的结果，编译器可以把它的作用域从一个基本块扩展到EBB。为了运用这一算法，编译器对穿过EBB的每一条路径做值编号。在图8-5的代码片段中，它必须对{A，B}、{A，C，D}和{A，C，E}做值编号。考虑{A，C，D}。编译器在A上使用局部值编号，然后使用其结果散列表作为对C做值编号的初始状态。最后，编译器可以使用这一结果表开始处理D。事实上，它把{A，C，D}当作一个块处理。这一方法可以发现局部算法不能发现的冗余和常量值表达式。

为了把这一算法用于第二条路径{A，C，E}，编译器可以再一次从A开始，然后处理C，再处理E。分别处理EBB中的每一个路径，编译器可以达到理想的结果。然而，值编号的代价将会因为对像A这样是多个EBB的前缀的块的处理而毫无理由地增加。在超局部化算法中，为了尽可能少地增加编译时间，我们希望通过检查已增加的上下文而得到效益。为了实现这一效益，基于EBB的算法通常利用EBB的树结构。 408

过程内与过程间

编译中很少有术语像单词全局（global）那样带来太多的混淆。全局分析和优化是在过程作用域上操作。然而，现代英语的内涵暗示全包含作用域。

开始于IBM的PL/I优化编译器，并在20世纪80年代得到发展的跨越过程边界的分析和优化的兴趣导致对单一过程技术的过程内（intraprocedural）这一单词的频繁使用，也导致对跨越两个或更多过程的技术的过程间（interprocedural）这一单词的频繁使用。因为这些单词在拼写和发音上很接近，所以它们很容易混淆且很难使用。

Perkin-Elmer公司设法弥补这一缺陷，它引入了“通用（universal）”优化编译器；这一系统执行广泛的内联操作，接着在结果代码上执行积极的优化。“universal”这一术语并非正式用语。我们更喜欢使用术语全程序（whole program）并把它用在任何可能的地方。它传达正确的差异并提醒读者和听众注意：“global”不是“universal”。

为使在EBB上的值编号高效，编译器必须复用出现在EBB的多条路径上的块的结果。无论是以深度优先还是以广度优先方式处理这些块，它都需要撤销处理一个块的效应的方法。在处理{*A*，*C*，*D*}之后，它必须再次生成{*A*，*C*}结束时的状态，以便复用这一状态来处理*E*。编译器可以实现的众多方法中包括：

- 它可以记录每个块的边界处的表的状态并在需要时恢复这一状态。这有可能增加算法的空间需求，但是它使得算法对每个块只处理一次。
- 它可以通过向回遍历块来展开释放这一块的效应，并且在每一个操作处，撤销向前传送的工作。这可能引发丢失信息的问题：如果一个操作在向前传送中为*x*生成了一个新的值编号，那么*x*的早前的值编号是什么呢？这一方法要求向前传送记录某些额外的信息。
- 409 它可以使用为词法作用域散列表所开发的机制实现这一值表（参见5.7.3节）。当它进入一个块时，它创建一个新的作用域。在撤销一个块的效应时，它删除那个块的作用域。

所有三个方案都有效。作用域值表提供最低的代价，特别是当我们复用为管理作用域符号表所开发的机制时更是如此。因为编译器可以估测每个作用域所需的表的大小，所以编译器能够避开符号表实现中由于这一符号表变满而需要扩展这一表格所引发的复杂性。（在ILIC代码片段中，名字的最大数量是ILOC操作数量的三倍。）

使用作用域值表，仍然存在一种复杂性。如果一个名字在几个块中被定义，那么在一个块中的定义的效应可能被记录在与其他块相关的作用域中。在这种情况下，撤销表并删除与一个块相关的作用域不能消除与这个块相关的所有事实。值编号算法可能需修补值编号已被取代的表达式在其他作用域中的条目。⊖

为了避开这一复杂性，编译器可以在每个名字刚好被定义一次的表示上执行超局部值编号。这一命名规则通过保证一个块中的定义被记录在与这个块相关的值表的作用域中来简化值编号。正如在8.3.2节中所讨论的那样，这一命名规则还可以使值编号更高效。

我们已经看到拥有这一性质的IR，即SSA形式（参见5.5节）。SSA形式有两个重要性质。每个名字刚好由一个操作定义，以及一个值的每一次使用都刚好引用一个定义。前一个性质正是我们需要的性质，利用这一性质我们可以使超局部值编号的作用域值表版本高效地工作。图8-6给出我们例子的SSA形式。

在这一例子中，超局部算法将首先处理所有穿过EBB{*A*，*B*，*C*，*D*，*E*}的路径，然后是EBB{*F*}，最后是EBB{*G*}。（事实上，因为这一算法不依赖于每个EBB的头部之前的上下文，所以这一算法可以以任意顺序处理这些EBB。）具体地说，这些动作可能以下面的顺序出现：

⊖ 5.7.3节给出的“表束”实现可以避免这一问题，但是附录的B4.5节中给出的空间高效技术把不同块中的定义的存储结合在一起。

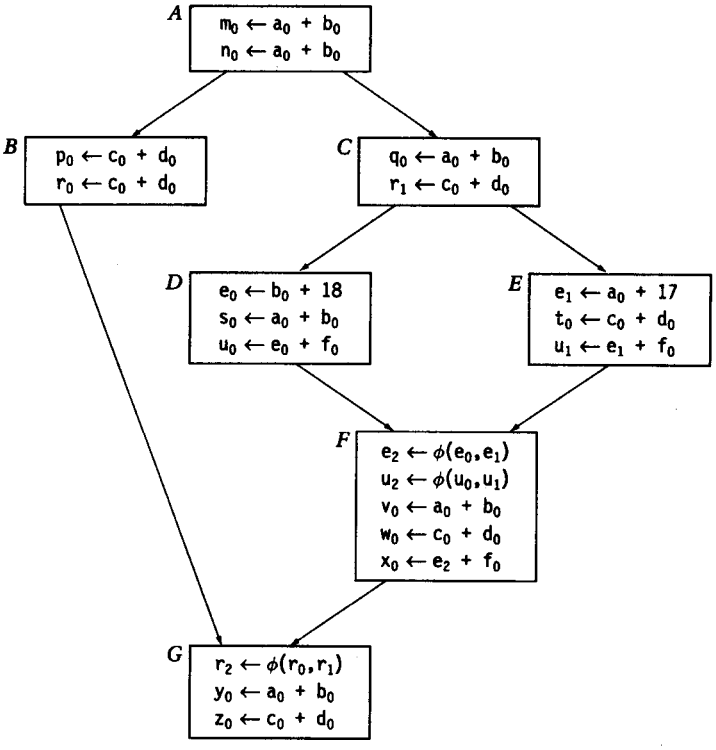


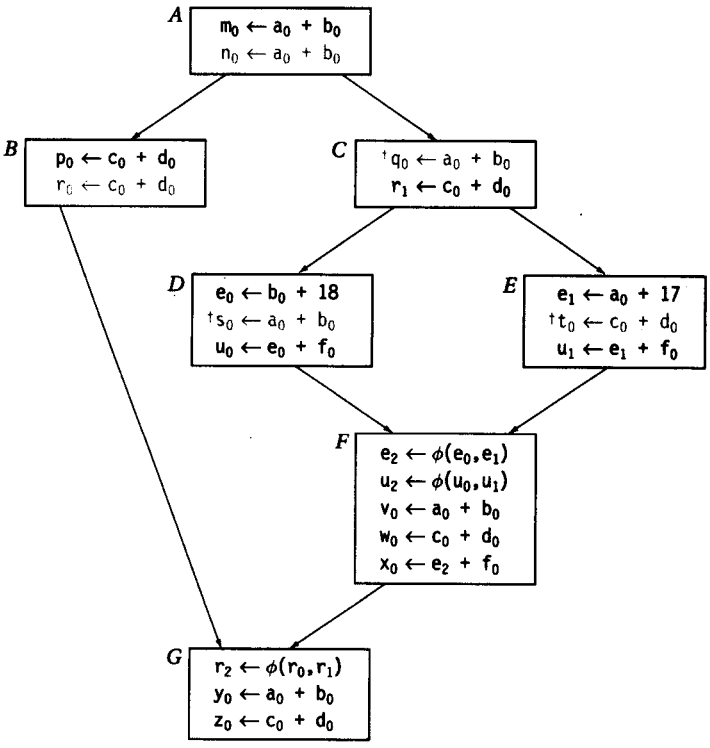
图8-6 SSA形式举例

410

1. 为*A*生成一个作用域
2. 值编号*A*
3. 为*B*生成一个作用域
4. 值编号*B*
5. 删除*B*的作用域
6. 为*C*生成一个作用域
7. 值编号*C*
8. 为*D*生成一个作用域
9. 值编号*D*
10. 删除*D*的作用域
11. 为*E*生成一个作用域
12. 值编号*E*
13. 删除*E*的作用域
14. 删除*C*的作用域
15. 删除*A*的作用域
16. 为*F*生成一个作用域
17. 值编号*F*
18. 删除*F*的作用域
19. 为*G*生成一个作用域
20. 值编号*G*
21. 删除*G*的作用域

对于每个基本块，超局部算法创建一个作用域，运用局部值编号，对当前EBB中的所有后继重复这一工作，然后删除这一作用域。通过使用处理作用域的一个简便的机制，编译器设计者可以保持超局部算法的代价接近于局部算法的代价。

就其有效性来说，超局部算法比局部算法消除更多的冗余计算。在图8-7中，这一算法消除的计算用灰色表示，其余部分用黑色表示。标以†的操作将被局部值编号遗漏，因为这些操作依赖于在其前驱块中计算的值。这一算法在一个EBB内工作得相当好。然而它遗失某些机会。因为*F*和*G*形成它们自己的EBB，在那些块中的a_0+b_0的计算不被处理为冗余，即使它们实际上是冗余的。同样地，这一算法不把块*F*和*G*中的c_0+d_0看成是冗余的。



411 ~ 412

图8-7 超局部值编号的结果

最后，F中的e_2+f_0的计算揭示出另一个问题。这一计算是冗余的；到达块F的所有路径都计算e+f，而且在F中的评估之前e和f都没有被再定义。然而，这些计算计算出不同的值，所以它们必须有不同的值编号。为了识别这一冗余，我们需要不同的等价概念，正如我们将在8.6节中看到的那样。

尽管有这些缺点，但是超局部算法还是值得考虑的。它以最小的附加代价捕获比局部算法更多的冗余。使用作用域散列表的高效实现，处理EBB的负荷可以维持到较小的范围。

并非所有的局部技术都可以清晰地推广到EBB上。值编号可以很好地工作，因为它使用等价的值取代冗余的计算。插入新操作或消除现存操作的技术可能更复杂。例如，在超局部指令调度中就会发生这样的问题。把一个操作从块b_j中移到它的前驱b_i中必然使b_i的其他后继发生变化。把一个操作从b_i移入b_j必然要在b_i的其他后继中插入这一操作的拷贝。

8.5.2 基于支配者的值编号

超局部值编号算法丢失一些机会，因为当它达到在CFG中有多个前驱的块时必定要忽视整个值表。扩展值编号的作用域的下一个步骤应该是能够寻找图8-6中的块F和G中的冗余的技术：操作是冗余的，因为它们已在早前的EBB中被计算过。为了达到这一点，我们需要在CFG中的连接点间传播信息的方法：在上例中是从D和E进入F及从B和F进入G的连接点。

超局部方法不能直接扩展到包含连接点的区域。为了值编号F，它不使用表示EBB{A，C，D}中的D的表，因为控制也可以从E进入F。当然，超局部方法也不使用表示{A，C，E}中的E的表，因为它未能考虑从D到F的路径。这一算法将设法整合D和E的值表。这一整合操作将需要整合沿不相交的路径所
413 指定的值编号。例如，D和E中的e+f的计算得到不同的值编号。同样重要的是，这一算法将需要检查沿

一条路径但不是沿其他路径计算的值。F中b+18的使用将找到沿着从D进入F的路径的冗余计算，而不是沿着从E进入F的路径的冗余计算。因此，b+18不是冗余的。为处理合并点而扩展值编号化算法时所引发的复杂性问题是令人沮丧的。

资源限制与效益

值编号也许是增加作用域直接导致更好结果的最清晰的例子。值编号的目标是发现冗余表达式并消除对它们的冗余评估。如8.5节所指出的那样，移动到较大的作用域导致算法发现并消除更多的冗余表达式。仍然存在的问题是，这能够导致更快的代码吗？机器相关的效应可能导致消除冗余产生较慢代码的情况。

例如，使用引用取代冗余评估可以延伸由第一个评估所产生的值的生存期。这可能会增加对寄存器的需求，在最坏的情况下，引发寄存器分配器引入额外的存储和装入操作来处理这一需求。当然，这一替换也可能减少对寄存器的需求，例如，如果两个操作数都是一个值的最后使用。

这一效应是微妙的。特别是它依赖于每一个操作的执行频率，既依赖于从程序中消除的那些操作的执行频率，又依赖于加入其中的那些操作的执行频率。计算特定替换的效率包含很多低级的细节，包括所有其他替换的影响。预测这一影响的困难已促使编译器设计者无视这样的效应并假定转换是有利的。这一实践要追溯到第一个FORTRAN编译器，在这一编译器上，Backus指示他的小组假设在优化期间有足够多的可用寄存器，并把从计算到寄存器的映射问题推迟到后期遍[25]。

然而存在一个表，算法可以使用它来处理F。到达F的两条路径有一个共同的前驱{A，C}。A是这一代码片段的惟一入口。从A到F的每一条路径都经过A和C。因此，编译器知道A中的每一个操作和C中的每一个操作都必须在F中的第一个操作之前执行。这一算法可以使用处理C而生成的表作为处理F的初始状态。 414

D和E中的赋值又如何呢？通过使用SSA形式，编译器可以避开在C和F之间值被再定义的问题。因为每一个名字都刚好是通过一个操作来定义的，所以D和E中的操作可以向值表添加信息，但是它们不能使其无效。如果F中的一个操作引用在D或E中所创建的值，那么这个值的名字不能在C的表中。事实上，SSA形式向编译器保证，如F中引用了在D或E中创建的值，那么该引用将使用在F的顶点处由φ函数定义的名字。（注意对于e的定义和在F中的后继使用所发生的事情，比较图8-5和图8-6。）

使用C的表初始化F的值编号步骤将允许编译器消除a_3+b_3和c_3+d_3的计算。但是它不让编译器发现a_3+f_3已在沿着到达F的每一条路径上计算过，因为e_2+f_2的值是在C和F之间计算的。

使用相同的原则，这一算法可以在处理G时使用A的表。从而使得它消除在G中对a_4+b_4的评估。因为原来的计算出现在A中。然而，它不能消除c_4+d_4的计算，因为早前的计算是在B和C中进行的。

1. 支配者

值编号算法需要发现沿着到达一个块的所有路径的最近公共祖先的方法。检查例子的CFG，我们看到下面的关系成立：

块	A	B	C	D	E	F	G
立即支配者	—	A	A	C	C	C	A

对于每一种情况，这一关系选择位于每一条从A到这一结点的路径上的最近的前驱。因为A没有前驱，所以A没有这样的结点。这一关系是支配者（dominator）关系的一种形式。

在CFG中，如果结点x出现在从图的入口到y的每一条路径上，那么我们说x支配y，记作$x \gg y$。根据定义，$x \gg x$。如果$x \gg y$，且$x \neq y$，那么x严格支配y，记作$x \gg y$。y的支配者的全集合记作DOM(y)。DOM(F) 就是{A，C，F}。y的立即支配者（immediate dominator）是最接近y的严格统治者。记作IDOM(y)。

这一例子表明一个结点可以有多个支配者。结点A和C位于从A到F的每一条路径上，所以$A \gg F$，
415 $C \gg F$且$F \gg F$。下面的表给出上面例子的支配者全集合。

块	A	B	C	D	E	F	G
$\gg$	—	{A}	{A}	{A, C}	{A, C}	{A, C}	{A}
DOM	{A}	{A, B}	{A, C}	{A, C, D}	{A, C, E}	{A, C, F}	{A, G}
IDOM	—	A	A	C	C	C	A

上表中的最后一行给出每个块的IDOM。这些集合与区域值编号算法有什么关系呢？

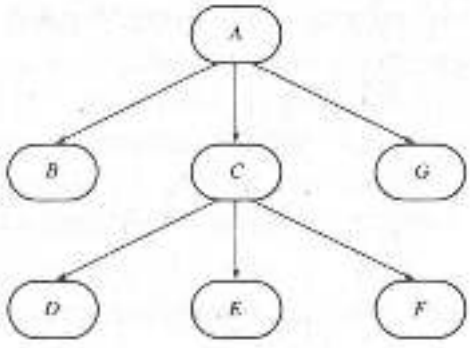
对于块b，DOM(b) − {b}中的每一个块必须在到达b的每一条路径上执行。因此，在对b的值编号中，编译器知道DOM(b) − {b}中的每个块在b之前已经执行。编译器可以使用DOM(b) − {b}中的任意块的值表初始化b的值表。这些块中哪个最好呢？

IDOM(b) 是DOM(b) − {b}中的块，且是最大的DOM集合，即如果i = IDOM(b)，那么，对于其他每一个j⊂DOM(b) − {b}，DOM(i) 包含j（以及DOM(j)）。编译器应该使用i的表来初始化b的值表，因为i出现在通向b的每一条路径上，且i的值表包含的信息比DOM(b) − {b}中其他任意结点所包含的信息都多。

为了运用这一观察，优化器可以把局部值编号方法运用到每一个块。对于块b，优化程序应该使用在块IDOM(b) 的末端运用的作用域值表。这与我们曾经用于超局部值编号的扩展和撤销作用域的基本框架相符。DOM和IDOM的计算将在9.3.1节中描述。

2. 对值编号使用IDOM

我们可以通过构建控制图的支配者树（dominator tree）来可视化IDOM关系。在这一树中，一个结
416 点的父结点是它的立即支配者。对于这一正在进行的例子，支配者树如下图所示：



对于给定结点，例如D，IDOM(D) 是它在支配者树中的父结点，而它的DOM集合包含其本身，以及这一树中它的所有祖先。因此，IDOM(D) 是C，而DOM(D) 是{D，C，A}。

基于支配者树的值编号算法前序遍历支配者树。这保证在访问一个块之前已经构建一个适当的表。这可能产生与直观相反的遍历顺序：例如，算法在F之前访问G。因为在访问G时这一算法可以使用的惟一事实是在处理A时发现的事实，它不仅没有给出F和G的相对顺序，而且它们也是不相关的。

8.6 全局冗余消除

值编号通过分析程序中值的使用，识别有相同值的表达式，并重写代码来消除冗余的计算。它集中处理值、而不是名字。可以用一种直接的方式扩展它的作用域以包含CFG的较大部分。无论是超局部值编号还是基于支配者的值编号都不能沿循环的后边传播信息。因此，它们失去发现某些冗余操作的机会。

超局部和基于支配者的值编号算法都在单一遍内执行CFG中的某个区域上的分析和转换。因为它们不能沿着CFG中的循环传播信息，所以它们在发现机会时重写代码。

处理整个循环的算法有可能找到更多的冗余操作。然而，这一算法必须在它能够向这一循环的先头的块传播信息之前处理整个循环。因此，在分析整个循环之前，它不能重写头部块。在分析和重写第一块之前，它不能重写第二块，以此类推。为了解决这一问题，大多数全局优化算法把分析与转换分离开来。对于冗余消除，这要包含一个识别出所有一定是冗余的表达式的阶段，其后再跟一个重写所有代码来消除多余评估的阶段。

发现冗余表达式的典型方法使用*数据流分析*（data-flow analysis）来计算在每一个块的入口处可用（available）的表达式集合。术语可用的定义如下所示：

- 一个表达式e在CFG中的点p处*被定义*（defined），如果它的值在p处被计算。我们称p是e的*定义场所*（definition site）。 417
- 一个表达式e在CFG中的点p处*被杀死*（killed），如果它的一个或多个操作数在p处被定义。我们称p是e的*杀死场所*（kill site）。
- 一个表达式e在CFG中的点p处*可用*（available），如果通向p的每一条路径都包含一个e的先前定义，且e不会在这个定义与p之间被杀死。

知道表达式e在p处可用本身不会使代码运行得更快。编译器必须重写代码以利用这一信息。最简单的重写方案是创建一个新的临时变量来保存e的值，需要时插入把e的早前评估拷贝到这个临时变量中的代码，并用临时变量的拷贝取代冗余评估。

我们按以下的方式分解全局冗余消除的处理。首先，我们给出计算在通向每一个块的入口处可用的表达式集合的算法。其次，我们描述一个简单的重写代码并取代冗余表达式的机制。最后小节把全局冗余消除的结果与各种值编号算法做一个比较。

名字的作用

就其核心，全局冗余消除依赖于冗余的语法概念。它找到e+f的重复评估，因为这些评估都使用e和f的名字。因此，它操作于程序原来的名字空间，而不是SSA的名字空间。它通过跟踪变量被杀死的地点来避免对译本名字的需求。因此，在序列

```
x ← e + f
e ← 5
y ← e + f
```

中，它识别出e+f的第二次出现不是冗余的，e的再定义杀死表达式e+f。

为了表示可能是冗余的表达式集合，编译器需要对每一个这样的表达式有一个不同的名字。编译器能够容易地构造出这样的名字。利用值编号的一个关键思想，编译器可以使用散列表构建从表达式到名字的映射。在对CFG的一次线性遍历中，编译器可以从每个操作符和它的操作数的名字构造出一个散列关键字，并给这一关键字指定一个独特的数。为了保证有一个简洁的名字空间，编译器可以通过递增计数器得到这些数。表达式名字的集合不会比变量的集合加上常量的集合小。它的大小没有CFG中的表达式的数量大。 418

8.6.1 计算可用表达式

全局冗余消除的第一步是计算在程序各点处的可用表达式集合。对于每一个块n，这一计算的结果是集合AVAIL(n)，这一集合包含在通向块n的入口处可用的所有表达式的名字。这一计算被形式化为涉及CFG中与各个块相关的集合的一系列联立方程。正如在经典代数中那样，这些方程包含一些已知的集合和一些未知的集合。编译器使用某个已知的算法求解这些方程。编译器的目标是对每一个原来未知的集合尽可能找到最精确的值。

为了计算AVAIL集合，方程对每个块需要两个集合，它们的值是由块中的操作惟一确定的。第一个集合DEEXPR(n) 包含n中的向下暴露的表达式的集合。也就是说，e∈DEEXPR(n) 当且仅当n评估e且e的任意操作数都不在块中e的最后一次评估与块的尾部之间定义。如果e∈DEEXPR(n)，那么e在n的出口处可用。第二个集合EXPRKILL(n) 包含被n中定义杀死的那些表达式的全体。一个表达式被杀死，如果它的一个或多个操作数在这一块中被再定义。如果e在通向n的入口处可用（e∈AVAIL(n)）且e不属于EXPRKILL(n)，那么e在n的尾部可用。

给定所有块的DEEXPR和EXPRKILL集合，编译器能够通过求解下面的方程组计算AVAIL集合。

$$\text{AVAIL}(n_0) = \emptyset$$

$$\text{AVAIL}(n) = \bigcap_{m \in pred(n)} (\text{DEEXPR}(m) \cup (\text{AVAIL}(m) \cap \overline{\text{EXPRKILL}(m)}))$$

这里，n_0是通向CFG的入口结点，对于每个块n，方程寻找在进入n的CFG的每一条边上都可用的表达式的集合。对于每一条边<m，n>，方程计算沿着这条边可用的表达式集合为DEEXPR(m) 和（AVAIL(m)∩ $\overline{\text{EXPRKILL}(m)}$）的并集。前面的项包含在$m$中被定义且在$m$的尾部未被杀死的表达式。后项包含在通向$m$的入口处可用且在$m$中不被杀死的表达式。这两项的并集保证AVAIL($n$) 中的表达式在进入$n$的所有边上都可用。

419

1. 计算局部集合

编译器在计算AVAIL集合之前需要产生每一个块的DEEXPR和EXPRKILL集合。计算DEEXPR是直截了当的。而计算EXPRKILL就更复杂一些。图8-8给出一个对基本块n计算这些集合的算法。

```
VARKILL ← ∅
DEEXPR ← ∅
for i = number of ops in n to 1
    assume operation o_i is "x ← y op z"
    VARKILL ← VARKILL ∪ {x}
    if (y ∉ VARKILL) and (z ∉ VARKILL)
      then add "y op z" to DEEPXR(n)
EXPRKILL(n) ← ∅
for each expression e in the procedure
    for each variable v ∈ e
        if v ∈ VARKILL
          then EXPRKILL(n) ← EXPRKILL(n) ∪ {e}
```

图8-8 为AVAIL计算局部信息

这一算法初始化集合VARKILL和DEEXPR(n)。接着，在从这个块的底部到其顶部的遍历中，它填充这些集合。对于每个操作x←y op z，它检查y和z是否是VARKILL的成员。如果二者都不在VARKILL中，那么y op z是向下暴露的且属于DEEXPR(n)。因为这一操作定义x，算法把x加到VARKILL中。

这一算法的第二部分从VARKILL集合计算EXPRKILL(*n*)。这一计算是直截了当的。对于在这一过程中计算的每一个表达式*e*，它查看*n*是否杀死*e*的任意操作数。如果是，它把*e*加到EXPRKILL(*n*) 中。

为了加速这一计算，编译器可能要预先计算从变量*v*到包含这一变量的表达式集合的映射M(*v*)。然后，它可以通过在VARKILL中的变量*v*上迭代并把M(*v*) 加入EXPRKILL(*n*) 中来构造EXPRKILL(*n*)。

2. 计算AVAIL

AVAIL的方程把AVAIL(*n*) 的内容指定为DEEXPR、EXPRKILL和CFG中的*n*的前驱的AVAIL集合的函数。我们可以使用如图8-9所示的简单迭代算法求出AVAIL。这一算法假设CFG中的块已被命名为b_0 420
到b_k。

```
for i = 0 to k
   Compute DEExpr(b_i) and ExprKill(b_i) as in Figure 8.8
   Avail(b_i) ← ∅
Changed ← true
while (Changed)
   Changed ← false
   for i = 0 to k
      OldValue ← Avail(b_i)
      Avail(b_i) = ⋂_{p ∈ pred(b_i)} (DEExpr(p) ∪ (Avail(p) ∩ ¬ExprKill(p)))
      if Avail(b_i) ≠ OldValue then
        Changed ← true
```

图8-9 可用表达式的简单迭代算法

初始化步骤计算每一个块的DEEXPR和EXPRKILL集合，并把所有AVAIL集合设置为空集。迭代步骤反复计算每个块的新AVAIL。当AVAIL集合停止变化时迭代停止。正如我们将在9.2节中所看到的那样，AVAIL方程的结构，连同其背后问题的性质以及迭代算法的性质保证算法将到达一个不动点并停止。

可用表达式是一个*全局数据流分析问题*（global data-flow analysis problem）。很多这样的问题都在关于代码优化的文献中得到陈述。编译器通过解决这些问题来在代码中寻找到可以安全地运用相关转换的位置。求解这样的方程的算法称为*全局数据流算法*（global data-flow algorithm）。目前已提出很多这样的算法。如图8-9所示的迭代算法具有稳健性和简洁性的优势（这一算法是不动点算法的另一个例子）。

8.6.2 取代冗余计算

一旦编译器得到CFG中每个块的AVAIL集合，它就可以使用蕴涵在这些集合中的信息转换代码。（信息本身并不能使代码运行得更快。）冗余消除的重写阶段必须使用在早前的评估中所保存下来的冗余表达式的值的拷贝操作取代该冗余表达式，而且它必须插入代码来保存这些早前的结果。我们的算法将使用一个拷贝操作取代实际的冗余评估。我们不集中精力消除这些拷贝，而是假设编译器的后继阶段将 421
消除大部分（如果不是全部的话）这些拷贝操作。这简化重写阶段并允许编译器对这一重写步骤的结果使用它最强大的拷贝消除工具。

为了简化这一过程，我们使用两个遍。第一遍识别出可用表达式可以被复用的位置，并用编译器新生成的临时变量的拷贝操作重写那些操作。第二遍遍历这一代码并插入必要的拷贝操作，这些拷贝操作定义在第一遍中编译器生成的临时变量。对应于一个实际的替换，这一方法只生成一个临时变量，并只

为那些临时变量插入拷贝操作。

（作为一个选择，编译器可以把每个值保存在一个适当的临时寄存器中。代码每次计算一个给定表达式时，编译器插入一个拷贝操作来保存它的值。这一方法生成很多伪拷贝操作。尽管那些额外的拷贝可以通过优秀的死亡代码消除器来消除，但是尽量避开插入它们会更好些。）

在第一遍，编译器可以在每一个块上执行局部值编号，并用AVAIL(n)中的表达式初始化块n的值表。如果局部值编号找到e在这一块中的一个评估，其中$e \in$AVAIL(n)，那么它使用新生成的名字$temp_i$的拷贝操作重写这一表达式，其中i是e在名字空间中的索引，即e的惟一整数标识符。无论值编号什么时候执行这样的替换，编译器都通过设置USED(i)为*true*来使用布尔数组USED记录这一事实。就如同局部值编号插入的其他拷贝一样拷贝操作取代冗余计算。

在第二遍，编译器为记录在USED数组中的每一个表达式插入拷贝操作。对于每一个块n，如果$e \in$DEEXPR(n)且e在USED中的条目为*ture*，那么它必须在n中e的最后定义之后插入一个把e的值移到$temp_i$中的拷贝。尽管这听起来很复杂，但是这一代码上的一次线性遍历就可以插入所有所需的拷贝操作。

拷贝插入的方法只加入拷贝来保存被包含在冗余评估中的表达式。然而它的确会而且也真的插入不需要的拷贝。如果USED(e)是*true*，那么重写阶段保存e的每一个剩余评估。这也将很有可能保存从来没有达到相应的临时变量的使用的e的某些评估。聪明的编译器设计者可以工程化重写策略，插入较少的拷贝，但是这必然要增加重写时的额外工作。由这一简单的技术所插入的额外拷贝是无用的，也就是
422 它们的值从不被使用。死亡代码消除将高效且自然地消除它们。

8.6.3 结合上述两个阶段

把上面两个阶段结合起来，即在计算AVAIL的分析之后加上使用AVAIL初始化局部值编号的重写，产生从被表示为CFG的整个过程中消除冗余表达式的方法。这一结果算法发现冗余操作的集合不同于任意值编号算法所发现的冗余操作集合。

图8-10给出我们例子的原名字空间形式（而不是SSA形式）。因冗余而被消除的表达式以灰色表示。这些表达式的右侧标签给出这一系列算法中可以发现和消除这一冗余的第一个算法。标签的含义如下
423 所示：

标　签	算　法	节
LVN	局部值编号	8.3.2
SVN	超局部值编号	8.5.1
DVN	基于支配者的值编号	8.5.2
GRE	全局冗余消除	8.6

前三个算法LVN、SVN和DVN形成严格的层次关系。DVN发现由SVN和LVN发现的每件事情。同样地，SVN找到LVN找到的每个冗余表达式。另一方面，GRE使用不同的方法且找到在某种程度上不同于其他算法的改进集合。例如，只有GRE能够发现F中的e+f的评估能够复用D和E中计算的值。

三个值编号算法都识别相同的值，所以它们能够发现如果a=c且b=d（或a=d且b=c）那么a+b和c+d产生同样的值。同样，集中于值的相等性的做法使得值编号算法识别出x+0等于x，或者2 × y等于y+y。因为GRE依赖于文本相等，它一般不能发现这样的相等关系。然而，因为它使用LVN来执行替换，所以它将发现基于局部值的相等性的替换。

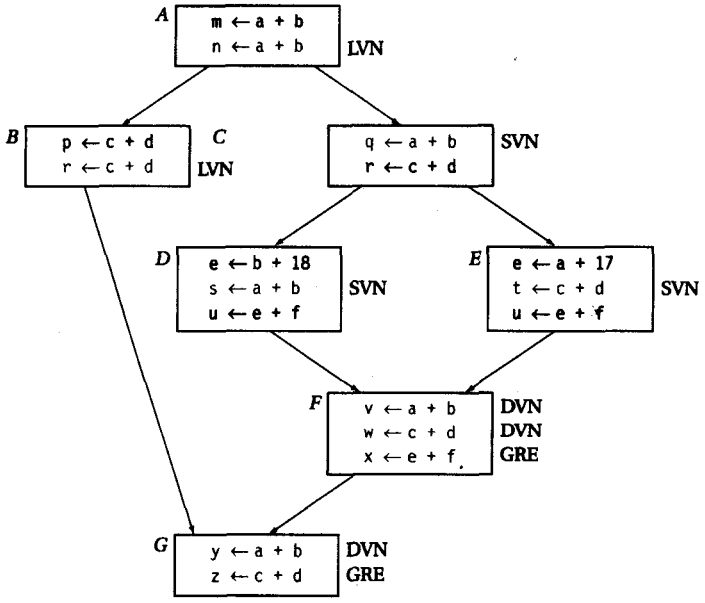


图8-10 比较算法

8.7 高级话题

通常人们认为较大作用域上的优化产生的代码好于较小作用域上的优化产生的代码。对于值编号，我们已经看到增加算法可用的上下文可使值编号找到更多的改进机会。另一方面，全局冗余消除发现显然不同的改进集合。全局冗余消除可以找到值编号方法无法找到的某些机会。但是它也可能无法发现某些局部值编号能够找到的机会。

然而有时候，增加优化可用的上下文不能改进结果。对于某些比全局或过程间问题中所看到的一般问题更简单的限定问题，局部和超局部方法更有效。在较小的作用内，编译器通常可以比在较大的作用域内证明更强的事实。在更大的作用域，分析也更加困难，控制流的相互作用和名字空间的处理更加复杂性。较大的作用域揭示出更多的优化机会；遗憾的是，这些机会只允许较弱的分析。 424

把分析和转换的作用域从单一块（或EBB）上扩展到包含CFG内连接点的较大区域上引发控制流的不确定性。在单一基本块内，编译器知道只有一条路径可以执行。穿过EBB的路径排斥连结点，所以编译器知道在这一EBB中哪些块总是先于给定的块。区域方法和全局方法包含连接点；这些块的代码必须作为通过代码的多条路径的一部分正确地发挥功能。其中有些路径可能是不可实行的，即它们从不执行。如果编译器能够发现不可实行路径，那么它就可以在分析期间忽视它们。如果编译器不能发现这些不可实行路径（这是一般情况），它必须假设它们是可以执行的。穿过CFG的所有路径是可执行的假设降低分析的精确性，而且限制编译器转换代码的能力。

把分析和转换的作用域从单一过程扩展到包含多个过程导致更复杂的情况。完整程序，或过程间方法必须模型化过程调用时名字空间改变的方式。值变得不可存取；值被重命名；值不再存在。当一个过程被若干过程调用时，每一个过程都带有自己的上下文，被调用过程的代码必须在其中每一个上下文内以及相应的名字空间内都能够正确地工作。这些影响在大多数单一过程分析问题中不会出现。

转换作用域的增加还增加它对资源分配产生广泛影响的潜在能力。例如，很多转换增加对寄存器的需求，这一需求有时称为*寄存器压力*（register pressure）。如果转换产生太多的寄存器压力，那么分配

器可能插入足够多的溢出代码而抵消原有的改进。因此，转移到更大的优化作用域需要坚信新方法的力量强大，且坚信编译器的其余部分处理结果代码的能力。这通常是可获利的。然而，有时候较小的优化作用域能获取更大的利益。

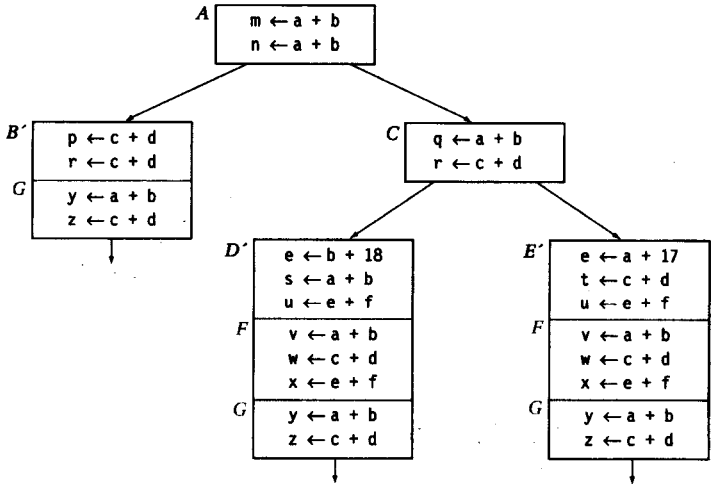
有时候，编译器复制代码来增加对优化可用的上下文，并创建新的优化机会。本节描述两个复制方法：在一个过程内复制块，以及在调用一个过程的地方用整个过程来做替换。这些技术给本章的主题，即我们能够通过修改算法来处理更大、更复杂的区域，提供一个相反的方向。相反，这些方法通过修改代码来创建带有简单控制流的更大区域。

8.7.1 通过复制增加上下文

425 各种值编号算法证明，在更大的作用域中使用优化可以带来额外的改进机会。然而，即使是基于支配者的技术也无法找到某些可能的替换。在上面的例子中，e+f在块F中的评估和c+d在块G中的评估显然是冗余的：两个表达式都在沿着从入口点出发的每一条路径上被评估。然而，所有值编号技术都无法发现它们并替换它们。⊖

这两个表达式都出现在在CFG中有多个前驱的块中。常常CFG中的这些合并点引发优化期间的信息遗失。为了解决这些问题，编译器有时候通过复制基本块来消除合并点。

为了复制一个块，编译器把这个块的拷贝附加到每一个前驱块的尾部。图8-11给出在我们的例子中复制块F和块G的结果。这创建三个新块B'、D'和E'。在这一图中，灰色部分表示原来块的边界。图中从G的每一个拷贝出发的向外的边提醒我们编译器必须把每一个拷贝与原来块G的每一个原来的后继连接起来。



426 图8-11 复制后的例子

以这样的方式复制块从下面三个主要方面改进优化结果。

1）这种复制方式创建较长的块。较长的块使局部优化处理更多的上下文。对于值编号的情况，超局部和支配者版本与局部版本一样强大。然而，对于某些技术，情况并非如此。例如，对于指令调度，超局部和支配者版本比局部方法更弱。在这一情况下，进行复制并随后使用局部优化可能产生更好的

⊖ 全局冗余消除有其自己的弱点。在序列x←a+d、y←a和z←y+d中，全局冗余消除不能发现x和z得到相同的值。

代码。

2）这种复制方式消除分支。组合两个块消除它们之间的分支。分支消耗执行时间。它们还破坏处理器中某些与性能紧密相关的机制，诸如指令抽取以及很多管道函数。消除分支的净效应是缩短执行时间，其手段是消除操作并使得预测行为的硬件机制更有效。

3）这种复制方式产生优化可能出现的地点。当复制消除一个合并点时，它可能在程序中创建新地点，编译器在这些点处可以得到更精确的运行时上下文的信息。例如，在块D'中，变量x得到值b+18+f，而在E'中，x得到值a+17+f。在原代码中的任意点都没有这样的性质。

当然，复制也有它自己的代价。复制之后，上述的例子有24个算术操作，而不是原来的17个。更大的代码可能运行得更快，因为它避开某些块末端的跳转。如果它的大小引发额外的指令缓冲失败，那么它也可能运行得更慢。它也可能允许更高效的优化；例如，局部值编号算法现在能够识别出表达式x←e+f的两次出现是冗余的。在用户更关心代码空间而不是速度的运用中，复制可能不具有效率。

8.7.2 内联替换

过程调用给优化带来严重的障碍。对另外一个过程的调用有直接的代价；程序必须执行调用序列中的所有操作。在某些情况下，调用序列比需要的更一般。对另外一个过程的调用有间接的代价；除非编译器有关于被调用者的行为的精确信息，否则它必须假设被调用者有不利的行为：修改它能够存取的每 427
一个值。在很多情况下，编译器不能得到这一信息，而且调用过程中的代码质量也因此受到影响。

类似的效应也发生在被调用者中。当编译器翻译一个过程时，它做有关运行时环境的假设：代码将继承它的调用者。在缺乏精确信息的情况下，编译器必须保证被调用者在任意合法的环境中都能正确操作。使用更好的信息，编译器通常可以生成更有效、特化的代码。

由于缺乏信息引发某些低效性。如果编译器能够在编译一个程序的任意部分之前分析整个程序，那么它就可以解决这一问题。其余的低效性则是过程型程序设计固有的。一个过程可以从不同的调用场所调用，每一个调用场所都带有不同的运行时条件集合。只要编译器必须构建对于所有调用场所都可行的单一可执行代码，那么为了一般性其结果代码必定在某些调用上下文中牺牲有效性。 428

为了抗击这些无效性，编译器可以执行内联替换（inline substitution），或内联（inlining）。这一转换使用被调用过程体替换调用该过程的调用场所，且在替换时使用适当的命名和拷贝来模拟原来调用场所的参数绑定效应。图8-12有两个调用者fee和fie的代码片段。这两个调用者都调用foe。对所有对foe的调用使用foe的过程体进行替换创建如图8-13所示的情况。fee和fie都有foe中的代码的私有拷贝。编译器可以使用包围这些拷贝的代码剪裁它们。与原来的三过程代码相比，替换的结果可能是有两个新过程的更好代码，其代价是更大的代码空间。

有若干可能的改进。

1）更有效的优化。消除调用把调用者和被调用者的代码同时暴露给同一个分析和转换。编译器可以得到更精确的信息，并把一信息运用于这一程序的更大部分，即结合起来的过程。

2）消除操作。内联foe的过程体消除调用序列中的大部分操作。这包括保存和恢复寄存器、建立可寻址性以及执行跳转或分支来转移控制。变量存取可能变得更简单（例如，通过内情向量存取数组值）。

当然，内联替换在改进代码方面的成功只取决于优化器和代码生成器能够利用这些机会的程度。其潜在的隐患包括由于代码增加以及寄存器需求的增加引发的问题。

（内联替换消除调用序列中的保存和恢复行为，这一行为破坏调用场所之间的值流向。这一“分裂”行动改变寄存分配问题。如果调用出现于寄存器需求高的区域，那么内联可能使事态更糟，而且诱发额外的寄存器溢出。由于寄存器溢出的某些突发事件，调用场所的需求可能诱发程序中其他部分的溢出。）

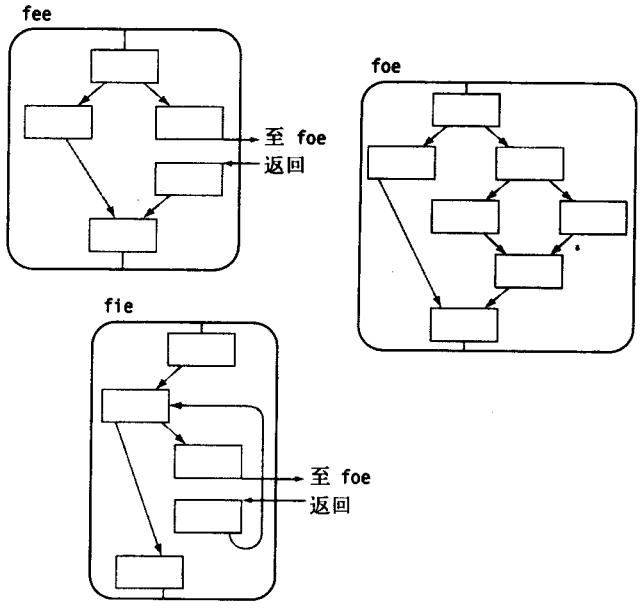


图8-12 内联替换之前

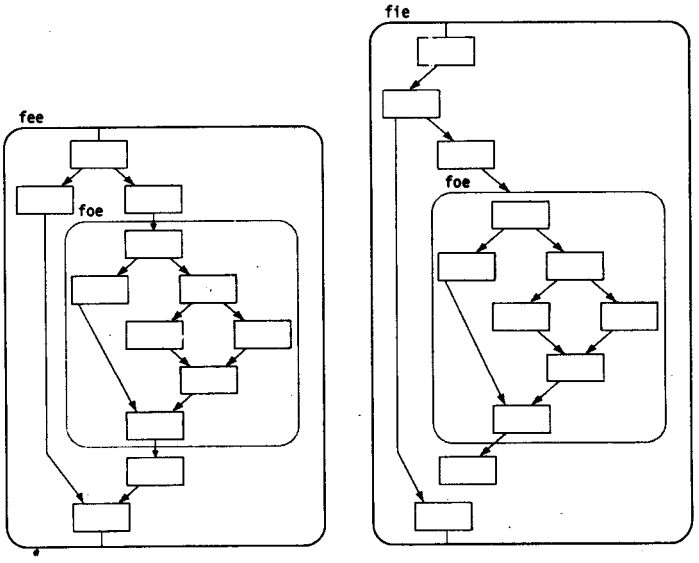


图8-13 内联替换之后

429

8.8 概括和展望

现代编译器中的优化器包含一系列试图改进编译代码性能的技术。大多数优化设法加速可执行代码。对于某些应用，诸如代码大小或能量消耗等其他度量非常重要。本章深入研究了冗余消除，并利用它来探究优化器中出现的一些问题。

优化是通过对照手边代码的特殊细节剪裁一般的翻译方案来改进程序的性能。优化器中的转换设法消除在支持源语言抽象中带来的负荷，其中包括数据结构、控制结构以及错误检查。这些转换设法识别

出有效实现的特殊情况，并重写代码来实现这些成本削减。对照处理器的实际可用资源，它们设法匹配程序对资源的需求，包括功能单元、内存层次上每一层次上的有限存储（寄存器、高速缓存、翻译后备缓冲器以及内存）、移动数据的各种带宽以及指令级并行等。

在优化器可以运用任意转换之前，它必须确定这一代码的计划重写是安全的：它保持代码原有的意 430
义。这通常要求优化器分析代码。在值编号方法中，构造散列关键字并执行查找的过程保证击中值表的操作必定是在这一块中或其前驱块中遇到的一个操作的拷贝。在全局冗余消除中，优化器通过计算可用表达式来发现哪些表达式在通向每个块的入口处一定是可用的，因而可以用拷贝操作替换这些表达式。

一旦优化器确定运用一个转换是安全的，那么它必须决定它是否改进代码。对于本章所描述的冗余消除的形式，编译器在没有深入分析的情况下假设它们改进代码。这一假设有三个重要成份：（1）拷贝至少与执行原来操作一样廉价；（2）冗余消除加入的大部分拷贝后来被消除；以及（3）冗余消除增加的对寄存器的需求不超过它的利益。如果所有这些假设成立，而且代码包含冗余表达式，那么任意形式的冗余消除都将改进代码。

本章介绍了优化中出现的很多术语和论点。对此课题的更多资料，有兴趣的读者可以参考数据流分析（第9章）和标量优化（第10章）。

本章注释

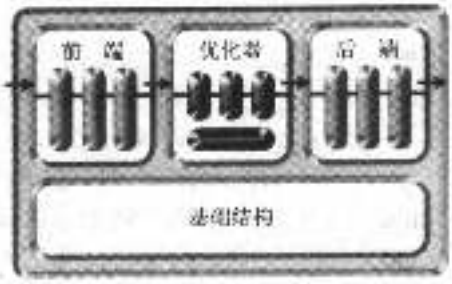
8.2节的例子来自于出现在LINPACK库中的基础线性代数子例程[123]。讨论暗示，循环展开有其他效应。Kennedy指出编译器可以使用循环展开来消除循环中的寄存器到寄存器拷贝[202]；Copper和Waterman指出循环展开可以改变某些处理器上的能源消耗[102]。

人们很早就认识到了通过寻找并消除冗余而带来的代码改进的机会。20世纪50年代Ershov在一个系统中实现了其中的某些效应[131]。Floyd提到了改进的可能性，以及使用交换性可能带来的影响[143]。

局部值编号通常归功于20世纪60年代后期的Balke[15，86]。到EBB的扩展很自然，而且毫无疑问已在很多编译器中得到了实现。我们给出的超局部和基于支配者的值编号的特殊算法归功于Simpson[306，50]。（参见第9章关于支配者的注释。） 431

Cocke描述了使用可用表达式来执行全局公共子表达式消除[82]。这一论文清楚地把优化分解成分析问题及其后的转换。后续的工作精炼并改进了基于可用性的冗余消除的概念。例如，参见10.3.2节中的惰性代码运动的描述。

代码复制是一个老想法[15]。块复制已用于改进调度[190]及冗余消除[41]。内联替换得到了广泛的应用与研究[111，290，93]。 432



第9章
数据流分析

9.1 概述

正如我们在第8章所看到的那样，优化是分析一个程序并对这一程序进行转换以改进它的运行时行为的过程。在编译器能够改变代码之前，它必须定位程序中的某些点，在这些点改变代码有可能改进代码，而且编译器必须证明在这些点处改变代码是安全的。与编译器的前端相比，这两项任务都要求编译器对代码的理解更深入。为了收集定位机会和证明优化所需要的信息，编译器使用静态分析的某种形式。

静态分析一般包括在编译时对运行时发生的情况加以推断，其中包括值在运行时是如何流动的，以及运行时变量取什么样的值。在简单的情况下，这可以产生精确的值，编译器能够精确地知道当代码执行时将发生什么。如果编译器能够得到精确的信息，那么它可以使用表达式或函数结果的一个立即装入取代其运行时评估。另一方面，如果代码从任意外部设备读取值，甚至包括一定量的控制流，或遇到任意的（从指针、数组引用或引用参数调用而来的）歧义性内存引用，那么分析代码就会变得更困难而且分析的结果将缺乏精确性。

在第8章，我们开发了一系列寻找和消除冗余，也就是寻找和消除必定产生相同结果的操作的技术。在某些情况下，编译器知道结果的值，并能够用它的值简单地取代计算。在另外一些情况下，编译器不知道这些值，但是它仍然可以通过消除冗余操作来改进代码。所有这些方法都得自于静态分析。

局部和超局部值编号算法得到直线代码中值流动的相当精确的信息（参见8.3.2节和8.5.1节）。它们依次检查操作并记录这些操作的可能效应。这些方法的精确性大部分直接来自于这些算法只处理有限种类的控制流的事实。同样的事实也限制它们操作的范围。

基于支配者的值编号（dominator-based value-numbering，DVN）算法分析并改进更大范围的代码（参见8.5.2节）。控制流是分析中的一个限制因素。当DVN处理带有多个前驱的块时，它必须放弃位于当前块和它的立即支配者之间的块的信息。因为同样的原因，DVN从不能识别出封闭一个循环的后边所携带的冗余。

基于可用表达式的全局公共子表达式消除采用不同的方法（参见8.6节）。它执行数据流分析并利用其分析结果，使用对早前计算的引用来取代冗余评估。它处理任意控制流，但是可以发现的冗余种类有更大的局限性。可用表达式的基于集合的分析无法跟踪赋值中的值。它无法凭借自己识别和简化算术等式。

正如我们所看到的那样，编译器可以在不同的作用域上执行静态分析。局部值编号执行局部分析。可用表达式中的EXPRKILL集合的计算也是局部的。很多特定的分析都在循环或者上执行。已知最好的静态分析技术是操作于整个过程之上的全局技术。更大的作用域是可能的：过程间分析技术得到执行过程调用对调用过程的运行时环境的影响的信息。某些问题需要过程间分析：例如，任意适当详细的指针值分析必须扫视过程边界。

本章详细给出静态分析的最常见形式：数据流分析。编译器使用数据流分析操纵优化和代码生成。9.2节给出解决数据流问题的迭代算法的更详细论述。与此同时，本节介绍编译器用于寻找安全且有效

机会的其他数据流问题。9.3节更加详细地讨论静态单一赋值形式，或SSA形式。一个程序的SSA形式以 434
简单而直观的形式编码控制流和数据流。它充当许多现代转换和代码生成算法的基础。本章高级话题一节研究非结构性的控制流、指针和数组的使用以及程序的模块分解等引发的数据流分析中的问题。

9.2 迭代数据流分析

在第8章，我们看到了从冗余消除的区域性算法到需要全局分析阶段的全局算法的迁移。编译器在转换任何块之前必须计算*控制流图*（control-flow graph，CFG）中每个块的AVAIL集合。转换的安全性取决于这个AVAIL集合的精确性，而这些集合则取决于分析器只有通过检查整个过程才能得到的事实。

计算可用表达式是*数据流分析*（data-flow analysis）的一种形式，即对运行时的值流向进行编译时推断。数据流分析就是求解在代码的图形表示上构建的方程组，它发现程序运行时可能发生的事实。在优化编译器中，数据流分析的主要用途是定位优化机会，也就是编译器可以使用转换的位置，而且要证明在代码的某些点上使用转换是安全的。

数据流分析还用于帮助程序员更好地理解他们的程序。定位数据流的不规则性的分析暗示程序中逻辑流的可能结果所造成的问题。其例子包括未定义变量的使用和从不被使用的变量定义。任意条件的出现都暗示程序可能出现逻辑问题。

9.2.1 活变量

作为数据流分析问题的第二个详细例子，考虑*寻找活变量*（live variable）。一个变量v在点p处是活的，当且仅当存在一个从p到v的使用的一个路径，而且沿着这一路径v不被重新定义。编译器以多种方式使用活变量信息。 435

静态分析与动态分析

静态分析的概念直接引出一个问题，那就是什么是动态分析？根据定义，静态分析设法在编译时评估运行时所发生的事情。在很多情况下，编译器不能告知将发生什么，即便是使用一个或多个运行时变量时答案是显然时也是如此。

例如，考虑下面的C语言片段：

```
x = y * z + 12;
* p = 0;
q = y * z + 13;
```

它包含一个冗余表达式（y*z）当且仅当p不包含y或z的地址。在编译时，p的值以及y和z的地址可能是未知的。在运行时，它们是已知的且可被测试。在运行时测试这些值将可以使代码避免重复计算y*z，对此编译时分析无法回答这一问题。

然而，测试p=&y还是p=&z还是二者都成立，以及根据测试结果所做动作的代价很可能超过重复计算y*z的代价。为使动态分析有意义，它必须是有效的，即节省必须超过执行分析的代价。在某些情况下，这会发生；而在大多数情况中，情况并非如此。相反，静态分析的代价可以被摊派在可执行代码的多次运行上，所以它一般更具有吸引力。

- 活变量信息在全局寄存器分配中扮演着重要的角色（参见13.5节）。寄存器分配器不需要把非活的值保存在寄存器中；当一个值经历从活到非活的过渡时，分配器可以为其他目的复用它的寄存器。

- 活变量信息用于改进SSA结构；一个值在它是非活的任意块中都不需要ϕ函数。以这种方式使用活
436 性信息可以大幅度减少编译器在构建程序的SSA形式时必须插入的ϕ函数的数量。
- 编译器可以使用活性信息来发现对未初始化变量的引用。如果某个变量v在通向这一过程的入口处是活的，那么存在从入口n_0到v的使用一个路径，沿着这一路径v不被赋值。如果v是一个局部变量，那么这一路径表示在v没有被赋值时对v进行了引用。
- 最后，活变量信息可以直接用于操纵转换。例如，一个不再是活的值不需要存回内存中（这是称为*无用存储消除*（useless-stroe elimination）的转换）。

无论是源代码级别的变量还是编译器生成的临时名字，活性是变量的一个性质，而可用性是表达式的一个性质。因此，活性分析的领域与可用表达式分析的领域是不同的。

1. 活变量方程

为了计算活变量信息，编译器可以使用一个集合LIVEOUT(n) 来注释CFG中的每一个结点n，而LIVEOUT(n) 集合包含在n的出口处是活的每一个变量的名字。这些集合由下面的方程定义：

$$\text{LIVEOUT}(n_f) = \emptyset$$

$$\text{LIVEOUT}(n) = \bigcup_{m \in succ(n)} \text{UEVAR}(m) \cup (\text{LIVEOUT}(m) \cap \overline{\text{VARKILL}(m)})$$

这里，UEVAR(m) 包含m中的向上暴露变量，即在m中重新定义之前使用的变量。VARKILL(m) 包含在m中定义的所有变量，且n_f是CFG的出口结点，诸如集合VARKILL中的上划线表示这一集合的逻辑补。

这一方程直观地编码上述定义。LIVEOUT(n) 恰好是在跟着n的某个后继块m的输入中活着的所有变量的全体。这一定义要求值在某一条路径上是活的，它不需要在所有路径上都是活的。因此，CFG中n的后继的贡献组合在一起形成LIVEOUT(n)。n的特定后继m的贡献是：

437 $$\text{UEVAR}(m) \cup (\text{LIVEOUT}(m) \cap \overline{\text{VARKILL}(m)})$$

变量v在满足两个条件之一的情况下在通向m的入口处是活的。这个变量在m中被重新定义之前可能在m中被引用，在这一情况下$v \in$UEVAR(m)。它可能在m的出口处是活的，并安然无恙地通过m，因为m不重新定义它，在这种情况下$v \in$LIVEQUT(m) $\cap$ $\overline{\text{VARKILL}(m)}$。运用 $\cup$ 把上面两个集合合并起来得到m对LIVEOUT(n) 的贡献。为了计算LIVEOUT(n)，分析器合并所有n的后继的贡献。

为了计算LIVEOUT集合，编译器可能使用下面三步骤算法。

1）构建一个CFG。这涉及遍历代码的IR版本，发现基本块，并示例结点和边来表示这些块以及它们之间的转换。图9-1给出这一算法。

2）收集每个块的初始信息。分析器一般必须遍历这个块并执行某些简单的计算。对于LIVEOUT，这包括以后序遍历每个块来计算UEVAR和VARKILL。这一计算显示在图9-2的左栏中。

3）使用一个迭代不动点算法在CFG中传播信息。在每个结点上，这一算法评估数据流方程。当信息停止改变时，它的评估终止。图9-2的右栏给出这样一个算法。

每一步骤需要更深层的解释。

2. AVAIL和LIVEOUT之间的差异

比较AVAIL和LIVEOUT的方程可以揭示出二者之间的某些基本差异。一个块的AVAIL集合依赖于这个块在CFG中的前驱，而一个块的LIVEOUT集合依赖于这个块在CFG中的后继。我们称可用表达式为*向前*（forward）问题，因为信息沿着CFG的边向前流动。同样地，活变量是*向后*（backward）问题，因为信息沿着CFG的边向后流动。

AVAIL方程使用求交运算合并不同路径的贡献。因此，一个表达式在n的入口处可用当且仅当它沿通向n的所有路径都可用。LIVEOUT集合的计算使用并运算合并不同路径的贡献。因此，沿着从块n出去的任意路径的活变量在n的出口也是活的。数据流问题有时被刻画成任意路径问题或所有路径问题。 438

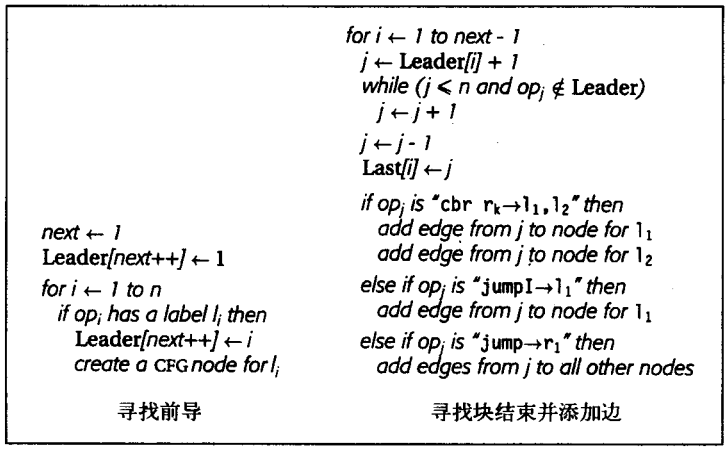


图9-1 构建控制流图

3. 构建控制流图

在最简单的情况下，CFG确定每一个基本块的开始端和尾端，并用描述块之间可能的控制转换的边把结果块连结起来。最初，假设CFG构建器作为输入得到一个表示典型类Algol语言中的简单、线性的IR。在本节的末尾，我们主要揭示在CFG的构建中引发的某些复杂性问题。

开始，编译器必须找到每一个基本块的开始端和尾端。在线性IR中，一个块的初始操作有时称为一个前导（leader）。一个操作是前导，如果它是这一过程中的第一个操作，或者它有可以成为分支目标的标签。编译器可以在对IR的一次遍历中识别所有的前导，如图9-1的左边所示。它在程序中的操作上依次进行迭代，发现带标签语句，并把它们记录为前导。⊖

第二次遍历寻找结束一个块的每个操作。它假设每个块都结束于一个分支或跳转，而且假设那些分支指定所有输出的标签："发生分支标签"和"不发生分支标签"。这简化块处理并使编译器后端来选择 439
分支的"落下"情况的路径。（目前，我们假设分支没有等待槽。）

为了找到每个块的尾端，这一算法以各块在前导数组的出现顺序迭代各块。它向前遍历IR，直到找到后继块的前导。那个前导的前一个操作结束当前块。算法把该操作的索引记录在Last[i]中，这样序对<leader[i]，last[i]>描述块i。它把所需的边加入CFG中。

正如我们在第5章所做的假设那样，CFG应该有惟一的入口结点n_0和惟一的出口结点n_f。代码应该有这样的形态。如果它没有这样的形态，这个图的一个简单的后序遍历可以创建n_0和n_f。

4. CFG构建中的复杂性

IR的性质、目标体系结构，甚至是源语言都可能使CFG的构建过程复杂化。例如，ILOC的jump操作这样的歧义性跳转可能迫使编译器把来自跳转的边加到过程中的每一个带标签的块。这可能把运行时从不出现的边加到CFG中，从而降低编译器得到的数据流信息的质量并减弱优化的效果。在目标已知的

⊖ 如果代码包含不作为分支目标的标签，那么这可能不必要地划分块。一个更复杂的算法可能只把分支和跳转的目标加入前导集合中。如果代码包含任意歧义性跳转，即一个到寄存器中的地址的跳转，那么它无论如何必须把所有带标签语句作为前导。

情况下，编译器设计者一般应该避免生成歧义性跳转，即使在IR生成期间这要求额外的分析。本书的ILOC示例避开使用跳转到寄存器操作jump，而使用立即跳转jumpI。

编译器设计者可以通过在IR中包含记录可能的跳转目标在一定程度上改进这一状况。ILOC引入tbl伪操作来让编译器记录歧义性跳转的可能目标。每当编译器生成一个jump，它都应该使用记录这一跳转的可能目标的tbl一组操作跟踪这一分支。当编译器构建CFG时，它可以使用这些线索限制假边的数量。跟在一组tbl后面的任意jump得到在这些tbl操作中命名的每个块的边。没有tbl的任意jump生成到每个带标签块的边。

在编译的后期阶段，编译器也许需要根据目标机器代码构建CFG。在这一点，目标的体系结构可能使这一过程复杂化。图9-1中的算法假设除了第一个前导外，所有的前导都带标签。如果目标机器有落下分支，那么必须扩展这一算法以识别在落下路径上接受控制的不带标签语句。如果代码已经完成调度
440 并且它模型化分支等待槽，那么问题就变得更困难。位于分支等待槽中的带标签语句是两个不同块的成员。编译器可以通过复制改变这一情况，创建这一等待槽中的操作的新（不带标签）拷贝。

等待槽还使寻找块尾端的过程复杂化。如果一个分支或跳转在分支等待槽中出现，那么GCF构建器必须从这个前导开始向前遍历来寻找块结束分支，也就是它遇到的第一个分支。一个块结束分支的等待槽中的分支自身，可能在通向目标块的入口处是未决的。它们可能分离目标块并迫使创建新块和新边。在这些环境下创建CFG所需要的分析更加复杂。

某些语言，例如Pascal和Algol，允许跳转到当前过程外面的标签。可以使用一个到为表示目标而创建的CFG结点的分支在当前过程中建模这一转换。这一复杂性也出现在这一分支的另一端，在此从一个未知源头的转换能到达一个块。出于这种原因，非逻辑转向通常被限定于一个词法嵌套过程中，在那里编译器可以一次查看所有相关过程的CFG。在这种情况下，编译器可以插入所需要的边并正确地建模这些效应。

5. 收集初始信息

对于LIVEOUT，分析器必须为每个块计算UEVAR和VARKILL集合。一次遍历可以同时计算这两项。对于每个块，分析器将这些集合初始化为空集。接着，分析器按从上到下的顺序遍历这个块并更新UEVAR和VARKILL来反映每一个操作的影响。图9-2的左边给出这一计算的详细过程。

```
for i ← 1 to number of operations
  assume op_i is "x ← y op z"
  if i ∈ Leader then
    b ← block number for i
    UEVAR(b) ← ∅
    VARKILL(b) ← ∅
  if y ∉ VARKILL(b) then
    UEVAR(b) ← UEVAR(b) ∪ {y}
  if z ∉ VARKILL(b) then
    UEVAR(b) ← UEVAR(b) ∪ {z}
  VARKILL(b) ← VARKILL(b) ∪ {x}
```

收集初始信息

```
N ← number of blocks − 1
for i ← 0 to N
  LIVEOUT(i) ← ∅
changed ← true
while (changed)
  changed ← false
  for i ← 0 to N
    recompute LIVEOUT(i)
    if LIVEOUT(i) changed then
      changed ← true
```

解方程

441 图9-2 迭代活分析

这一过程比AVAIL计算的相应部分简单，因为LIVEOUT使用VARKILL而不是EXPRKILL。从VARKILL计算EXPRKILL显著地增加AVAIL计算中的局部分析代价。（参见图8-8的比较）。

考虑如图9-3所示的代码片段。它由包含嵌套if-then-else结构的一个循环组成。这一代码中的很多

细节已被抽出。为了计算LIVEOUT集合，编译器应该首先构造CFG并计算每个块的UEVAR和 $\overline{\text{VARKILL}}$（VARKILL的补集）。通过观察，我们可以看到它将生成下面的集合：

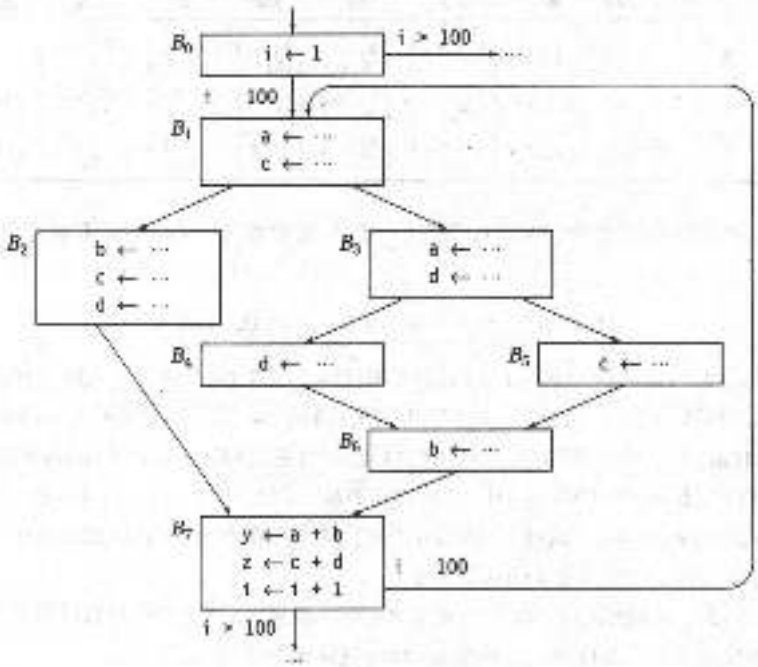


图9-3 LIVEOUT计算示例

442

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
UEVAR	∅	∅	∅	∅	∅	∅	∅	{a,b,c,d,i}
VARKILL	{a,b,c,d,y,z}	{b,d,i,y,z}	{a,i,y,z}	{b,c,i,y,z}	{a,b,c,i,y,z}	{a,b,d,i,y,z}	{a,c,d,i,y,z}	{a,b,c,d}

因为我们已省略了大部分操作的右部，所以UEVAR集合异常稀疏的。利用这些细节的话，大部分块的UEVAR集合将是非空的。

6. 求解LIVEOUT方程

为了求LIVEOUT集合的值，编译器设计者可以改编图8-9的迭代算法，可以使用LIVEOUT的等价代码取代AVAIL特化代码；其结果的迭代算法如早前的算法一样计算一个不动点。

图9-2右侧给出求解LIVEOUT方程的一个迭代方案。它把所有的LIVEOUT集合初始化为空集。接着，它按B_0到B_7的顺序计算每个块的LIVEOUT集合。它反复计算LIVEOUT集合，直到它们停止变化。这产生如下的LIVEOUT集合中的值。第一行是初始值。

迭代	LIVEOUT(n)							
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
0	∅	∅	∅	∅	∅	∅	∅	∅
1	∅	∅	{a,b,c,d,i}	∅	∅	∅	{a,b,c,d,i}	∅
2	∅	{a,i}	{a,b,c,d,i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{i}

（续）

迭 代	LIVEOUT(n)							
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
3	{i}	{a,i}	{a,b,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{i}
4	{i}	{a,c,i}	{a,b,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{i}
5	{i}	{a,c,i}	{a,b,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{i}

443

第1次迭代为每个块构造空的LIVEOUT集合，B_7的前驱除外。（只有B_7有非空的UEVAR集合。）计算

$$\text{UEVAR}(B_7) \cup (\text{LIVEOUT}(B_7) \cap \overline{\text{VARKILL}(B_7)})$$

产生集合{a，b，c，d，i}。LIVEOUT(B_2)和LIVEOUT(B_6)二者都在第1次迭代的尾部有这个值。第2次迭代填充更多的LIVEOUT集合。根据B_2，LIVEOUT(B_1)得到{a，i}。LIVEOUT(B_4)和LIVEOUT(B_5)从B_6都得到{a，c，d}最后，LIVEOUT(B_7)从B_3得到{i}；计算使用LIVEOUT(B_1)的最近的值。

第3次迭代把{i}传播到LIVEOUT(B_3)中，LIVEOUT(B_1)得到{a，c，d，i}。边<B_2，B_2>贡献出{a，c，i}，而边<B_1，B_2>添加{a，d，i}。因此，LIVEOUT(B_2)与LIVEOUT(B_4)和LIVEOUT(B_5)是相等的，而B_6和B_7都只传递它们自己的LIVEOUT集合的一个子集。

第4次迭代从边<B_1，B_3>把{c}加到LIVEOUT(B_1)中。最后，第5次迭代重新计算所有的LIVEOUT集合但不改变其中的任意一个。所以这一分析器从while循环跳出并停止。

9.2.2 迭代LIVEOUT解算器的性质

与LIVEOUT计算类似，数据流分析构成标量优化编译器的分析骨架。已经开发出很多解决数据流问题的技术。我们集中精力讨论迭代算法，因为这样的算法具有高速和稳健的行为，还因为我们对它的基础理论已经有深入的理解。本节概述LIVEOUT的迭代解算器（solver）的三个重要问题的解答：

1）分析终止吗？

2）分析计算什么答案？

3）分析有多快？

编译器设计者在设计分析追时都应该考虑这三个问题。

1. 终止性

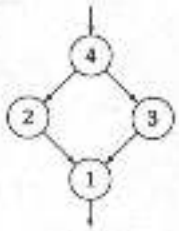
444 迭代的活变量分析终止。因为集合在整个计算中单调增长，算法初始化每个LIVEOUT集合为空集。对*while*循环的仔细推断表明LIVEOUT集合可以变得更大，但从不减小。任意LIVEOUT集合的大小由变量数|V|界定。在图9-3的例子中，V是{a，b，c，d，1，y，z}而且每一个LIVEOUT集合或者是V或者是V的某个真子集。

因为|V|是有界的，算法计算的LIVEOUT集合序列也是有界的。因此，迭代最后一定停止。当它停止时，解算器找到LIVEOUT计算的这一特殊实例的不动点。

2. 正确性

回想一下活变量的定义：一个变量v在点p处是活的，当且仅当存在一条从p到v的使用的路径，沿着这一路径v不被再定义。活性是根据图中的路径定义的。一定存在不包含v的定义的从p到v的使用的路径，我们称这条路径为v有效路径。

LIVEOUT(n) 包含v，当且仅当v在块n的尾部是活的。为了构成LIVEOUT(n)，算法计算在CFG中的n的每一个后继对LIVEOUT(n) 的贡献。它使用并集结合这些贡献，因为v∈LIVEOUT(n)，仅当v在任意离开n的路径上是活的。各个边上的局部计算是如何与所有路径上定义的活性相关联的呢？

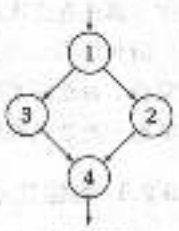


后序图

由这一迭代算法计算而来的LIVEOUT集合构成这些方程的一个不动点解。超出本节范围的迭代的数据流分析理论向我们保证对于这些特殊方程存在一个不动点，而且这一不动点是惟一的[199]。定义的所有路径解也是这些方程的不动点，称为在所有路径相遇解（meet-over-all-paths-solution）。不动点的惟一性确保由迭代算法计算而来的集合与所有路径相遇解相同。

3. 高效性

特定过程的LIVEOUT方程的不动点解与解算器计算各个集合的顺序无关。这是惟一性的一个结果。因此，编译器设计者可以自由地选择改进分析器运行时间的顺序。



逆后序图

图的逆后序遍历（reverse postorder，RPO）对迭代算法来说是特别高效的。后序遍历在访问一个结点之前以一致顺序尽可能多地访问该结点的子结点。（在循环图中，一个结点的子结点也可能是它的祖先。）RPO遍历恰好相反，在访问一个结点之前，它尽可能多地访问这个结点的前驱。一个结点的RPO编号是n + 1减去它的后序数，其中n是图中结点的数量。

```
N ← number of blocks −1
for i ← 0 to N
    LIVEOUT(i) ← ∅
changed ← true
while (changed)
    changed ← false
    for i ← 1 to N
    j ← Reverse Preorder[i]
    LIVEOUT(j) = ⋀_{k∈succ(j)} f_{j,k}(LIVEOUT(k))
        if LIVEOUT(j) has changed then
            changed ← true
```

图9-4 LIVEOUT的Round-Robin后序解算器

对于诸如AVAIL这样的向前问题，迭代算法应该使用在CFG上计算的RPO。而对诸如LIVEOUT这样的向后数据流问题，迭代算法应该使用在逆向CFG上计算的RPO。（逆向CFG就是把边逆置了的CFG，所以n_f是它的入口结点而n_0是它的出口结点。）逆向CFG的RPO等价于原来的CFG上的逆前序。

如果迭代算法使用适当的RPO遍历，它通常能够避开初始化某些集合的需要。例如，在活性分析中，初始化LIVEOUT(n_f) 为空集就足够了。在CFG的逆后序遍历中，对于每一个结点在这个结点的逆前序后继处的LIVEOUT集合计算仅使用这个结点的LIVEOUT集合的值，而解算器将在此计算之前计算这个结点的LIVEOUT集合，只有n_f除外。

为了查看顺序的影响，考虑图9-3的LIVEOUT计算上的RPO遍历的影响。逆向CFG的一个RPO编号是：

块	B_6	B_1	B_2	B_3	B_4	B_5	B_7	B_8
RPO编号	8	7	2	6	4	5	3	1

445

446

以这一顺序访问结点产生如下迭代

迭代	LIVEOUT(n)							
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
0	∅	∅	∅	∅	∅	∅	∅	∅
1	{i}	{a,c,i}	{a,b,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	∅
2	{i}	{a,c,i}	{a,b,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{i}
3	{i}	{a,c,i}	{a,b,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{i}

这一算法在三次迭代后停止，而由块编号严格规定的遍历需要五次迭代。比较这个表和早前的计算，我们就可以看到为什么会有这样的结果。在第一次迭代时，算法计算除B_7之外所有结点的正确LIVEOUT集合。算法需要在第二次迭代计算B_7的LIVEOUT集合，这是因为后边，即从B_3到B_1的边的存在。算法需要第三次迭代来识别这一算法已达到它的不动点。

9.2.3 数据流分析的局限性

编译器能够从数据流分析中了解的信息存在一定的局限。在某些情况下，这些局限来自于对分析所做的假设，而在另外一些情况下，这些局限来自于被分析语言的性质。为了做出可靠的决定，编译器设计者必须了解数据流分析能做什么以及它不能做什么。

当迭代算法计算CFG中的结点n的LIVEOUT集合时，它使用集合LIVEOUT、UEVAR以及n在CFG中的所有后继的VARKILL集合。这隐含假设执行可以达到所有这些后继；在实践中，这些后继中的一个或多个可能是不能达到的。考虑图9-5所示的代码片段，以及它的CFG。

B_0中对x的赋值是活的，因为x在B_1中使用。B_2中对x的赋值杀死这个值。如果B_1从不执行，那么x在B_3中的值在与y的比较后不是活的，且x∉LIVEOUT(B_0)。如果编译器能够证明测试（$y < x$）总是假的，
447 那么控制将从不转移到块B_1且从不执行对z的赋值。如果对f的调用没有副作用，那么整个语句是无用的且无需执行。因为测试的结果是已知的，编译器可以完全消除块B_0和B_1。

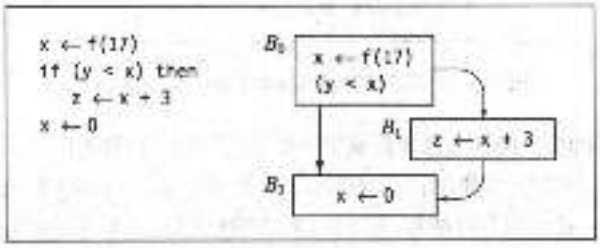


图9-5 控制流限制数据流分析的精确性

然而，LIVEOUT的方程取这一块的所有后继的并集，因此，分析器计算LIVEOUT(B_2)为：

$$\text{UEVAR}(B_1) \cup (\text{LIVEOUT}(B_1) \cap \overline{\text{VARKILL}(B_1)}) \cup$$
$$\text{UEVAR}(B_2) \cup (\text{LIVEOUT}(B_2) \cap \overline{\text{VARKILL}(B_2)})$$

所有数据流分析技术都假设CFG的所有路径实际上都可能被采用。因此，这些数据流分析技术计算的信息总括可能的数据流事件，假设每一条路径都可被采用。这限制了结果信息的精确度；我们说这一信息精确到“符号执行”，使用这一假设，x在B_2的出口是活的，而B_3和B_1必须被保存。

信息的非精确性殃及数据流的分析结果的另外一种方式来自于对数组、指针和过程调用的处理。诸如A[i，j，k]这样的数组引用将引用A的一个元素。然而，在没有揭示i、j和k的值的分析时，编译器不能告知存取的是哪一个元素。出于这一原因，编译器传统上把所有对数组A的引用结合到一起。因此数组A[x，y，z]的使用被看成是A的使用，而A[c，d，e]的定义被看成是A的定义。

然而，要注意尽可能避免做过强的推断。编译器知道它关于数组的信息不够精确，所以它必须保守地解释这一信息。因此，如果分析的目标是确定一个值在哪里不再是活的（也就是，这个值一定已经被杀死），那么A[i，j，k]的定义不杀死A的值。如果分析的目标是识别一个值也许不再存活的位置，那么A[i，j，k]的定义可能重新定义A的任意元素。 448

数据流方程中的集合命名

在书写经典问题的数据流方程时，我们对许多包含局部信息的集合进行了重新命名。原始论文使用更直观的集合名字。遗憾的是，在处理这一问题时这些名字之间会发生冲突。例如，可用表达式、活变量、可达定义以及忙碌表达式都使用KILL集合的概念。然而，这四个问题定义在三个不同的定义域上：表达式（AVAIL和VERYBUST）、定义点（REACHAES）以及变量（LIVEOUT）。因此，使用KILL或KILLED导致某些问题上的混乱。

我们使用的名字既编码定义域又编码有关集合意义的线索。因此，VARKILL(n) 包含在块n中被杀死的所有变量集合，而EXPRKILL(n) 包含在块n中被杀死的所有表达式。同样地，UEVAR(n) 包含在块n中向上暴露的所有变量，而UEEXPR(n) 包含向上暴露的所有表达式的集合。尽管这些名字有时候很难看，但它们却能鲜明地区分出用于可用表达式中的杀死概念和用于可达定义中的杀死（DEFKILL）概念之间的差异。

指针给静态分析的结果带来另外一层不精确性。指针上的显式算术把情况搞得更糟。没有跟踪指针值的特殊分析，编译器必须把对基于指针的变量的赋值解释为这一指针可能达到的每一个变量的可能定义。类型的安全性可能限制能够通过指针的赋值定义的对象集合；一个指向类型t的指针只用于修改类型t的对象。没有指针值的分析或类型安全性的保证，给一个基于指针的变量的赋值可能迫使分析器假设每一个变量都被修改了。在实践中，这通常阻止编译器在任意基于指针的赋值间把基于指针的变量的值保存在寄存器中。除非编译器可以特别地证明用于这一赋值中的指针不可能指向对应于寄存器中的值的内存位置，这是惟一的安全通道。

在实践中，指针分析的复杂性阻止很多编译器使用寄存器来保存基于指针的变量的值。有些变量通常被免除这种处理，例如地址从不被使用的局部变量就是这样的变量。另一种选择是执行以基于指针引用中的歧义性消除为目的的数据流分析：缩小在代码中每一点处指针可能引用的可能变量集合。 449

过程调用是不精确信息的最后一个源头。为了精确地建模当前过程中的数据流，编译器必须详细地了解被调用过程对当前过程和被调用过程都可存取的每一个变量所做的事情。而且，被调用过程可能调用其他过程。当然，这些过程将有自己的潜在影响。

除非编译器拥有总括程序中所有过程的效应的精确信息，否则它必须估测这些过程在最坏情况下的行为。尽管特定的假设因问题的不同而不同，但是一般的假设是被调用过程将引用并修改它能寻址的每一个变量，而且引用调用参数生成歧义性引用。因为很少有过程真的有这样的行为方式，所以这种假设通常过高估计过程调用的影响。这将进一步使数据流分析结果不精确。

9.2.4 数据流分析的其他问题

编译器使用数据流分析证明特殊情况下的转换的安全性。因此，已经有很多不同的数据流问题被提出，每一个问题都推动特殊的优化。

1. 可用表达式

第8章介绍了在代码中每一点发现可用表达式集合的问题。可用表达式被公式化为以在程序内计算的表达式为定义域的向前数据流问题。其结果的AVAIL集合用于驱动全局公共子表达式的消除。

2. 可达定义

在某些情况下，编译器需要知道给定操作的操作数的定义位置。如果在CFG中有多条路径导向这一操作，那么多个定义可以提供这一操作数的值。为了寻找到达到一个块的定义集合，编译器可以计算可
450 达定义（reaching definition）。REACHES的定义域是过程中定义地点的集合。某个变量v的一个定义d达到操作i，当且仅当i读取v的值且存在从d到i的v有效路径。

编译器使用集合REACHES(n) 注释CFG中的每一个结点n，该集合被计算为如下所示的向前数据流问题：

$$\text{REACHES}(n_0) = \emptyset$$

$$\text{REACHES}(n) = \bigcup_{m \in preds(n)} (\text{DEDEF}(m) \cup (\text{REACHES}(m) \cap \overline{\text{DEFKILL}(m)}))$$

DEDEF(m) 是m中向下暴露的定义集合，即m中的这些定义所定义的名字在m中不再重新定义。DEFKILL(m) 包含m中相同名字的定义所隐藏的所有定义点，即$d \in$DEFKILL(m) 如果d定义某个名字v且m包含同样定义v的定义。因此，$\overline{\text{DEFKILL}(m)}$ 即DEFKILL(m) 的补集由在m中不被隐藏的定义点组成。

DEDEF和DEFKILL二者都定义在定义点的集合上，但是计算其中任意一个都需要一个从名字（变量名字和编译器生成的临时变量名字）到定义点的映射。因此，收集可达定义的初始信息比收集活变量的初始信息更复杂。

3. 忙碌表达式

一个表达式e被认为在一个块n的出口忙碌（very busy），如果e在离开n的每一条路径上被评估且被使用，而且在n的尾部评估e产生的结果与沿着离开n的每一条路径对e的首次评估的结果相同。忙碌分析是表达式定义域上的一个向后数据流问题：

$$\text{VERYBUSY}(n_f) = \emptyset$$

$$\text{VERYBUSY}(n) = \bigcap_{m \in succ(n)} (\text{UEEXPR}(m) \cup (\text{VERYBUSY}(m) - \overline{\text{EXPRKILL}(m)}))$$

这里，UEEXPR(m) 是向上暴露的表达式集合，即那些在被杀死前在m中使用的表达式的集合。
451 EXPRKILL(m) 是在m中定义的表达式的集合；这与出现在可用表达式方程的集合是相同的。

编译器可以使用这一分析的结果定位代码提升（code hoisting）的机会。如果e在p处忙碌，那么编译器可以在p处插入一个e的评估，并删除离开p的每一条路径上的第一个e的评估。这一转换不缩短路径，但是减小整个程序中的操作数量。因此，它减小代码空间。

实现数据流框架

很多全局数据流问题的方程显示出令人惊讶的相似性。例如，可用表达式、活变量、可达定义以及忙碌表达式都有如下形式的传播函数

$$f(x) = c_1\ op_1\ (x\ op_2\ c_2)$$

其中，c_1和c_2是由实际代码所确定的常量，且op_1和op_2是诸如∪、∩和－等标准集合操作。这一相似性在这些问题有快速的迭代数据流框架的证明中显示出威力。这些相似性还应该在它们的实现中显示出威力。

编译器设计者可以容易地提取这些问题的不同细节并实现单一的参数化分析器。这一分析器需要计算c_1和c_2的函数，操作符的实现以及问题的方向指示。反之，它产生理想的数据流信息。

这一实现策略激励代码复用。它隐藏解算器的低级细节。同时，它创建编译器设计者可以对优化实现进行有效投资的环境。例如，允许这一框架实现$f(x)=a\ op_1(x\ op_2\ b)$为单一函数的方案可能胜过使用$f_1(x)=a\ op_1\ x$和$f_2(x)=x\ op_2\ b$并把$f(x)$计算为$f_1(f_2(x))$的实现。这一方案使得所有客户转换都受益于优化集合表示和操作符实现。

4. 常量传播

如果编译器可以证明某个变量v在代码的p点处总有值c，那么编译器可以用c替换在p处的v的使用。 452
这一替换使得编译器能够基于值c特化在p处的代码。例如，如果编译器知道一个循环的上界，那么它通常可以消除测试这一循环从不进入的情况所需要的比较和分支。为了证明v在某个点p处有值c，编译器可以执行全局常量传播。

这一问题的定义域是序对<v_i，c_i>的集合，其中v_i是一个变量且c_i或者是常量或者是表示未知值的特殊值⊥。分析使用集合CONSTANTS(n) 来注释CFG中每一个结点，这一集合包含编译器可以证明在通向n的入口处成立的所有变量和值的序对。CONSTANTS集合的定义如下所示:

$$\text{CONSTANTS}(n) = \bigwedge_{p \in preds(n)} F_p(\text{CONSTANTS}(p))$$

其中，∧在两个序对对集合上执行按对求交，且$F_p(x)$是在已知常量x的集合上建模p的效应的块特定函数。对这些需要更详细的解释。每一个CONSTANTS集合的初始值是空集。

上面的求交操作比较两个序对<v，c_1>和<v，c_2>并产生如下结果：如果$c_1=c_2$，那么<v，c_1>∧<v，c_2>是<v，c_1>；如果$c_1\neq c_2$，那么<v，c_1>∧<v，c_2>是<v，⊥>。(注意，如果c_1或c_2是⊥，那么这些规则产生<v，⊥>。) 因此，如果在一条进入n的路径上v有值3，而在另外一条路径上v有值5，那么编译器不能断定在通向n的入口处的v的值，所以分析使用值⊥来表明一个未知值。如果两条路径显示v有相同的值，比如说是13，那么∧产生序对<v，13>。

上述方程的另一个重要部分是块特定函数F_p。在我们已经看到的其他数据流框架中，块特定效应被建模为较小数量的集合操作。给定CONSTANTS(p)，算法通过如下方法建模这一块中的每一个操作来计算在p的尾部成立的集合:

$$\text{x} \leftarrow \text{y} \qquad \textit{if } \text{CONSTANTS}(p) = \{\langle \text{x}, c_1\rangle, \langle \text{y}, c_2\rangle, \ldots\} \textit{ then}$$
$$\text{CONSTANTS}(p) = (\text{CONSTANTS}(p) - \{\langle \text{x}, c_1\rangle\}) \cup \{\langle \text{x}, c_2\rangle\}$$

$$\text{x} \leftarrow \text{y}\ op\ \text{z} \quad \textit{if } \text{CONSTANTS}(p) = \{\langle \text{x}, c_1\rangle, \langle \text{y}, c_2\rangle, \langle \text{z}, c_3\rangle \ldots\} \textit{ then}$$
$$\text{CONSTANTS}(p) = (\text{CONSTANTS}(p) - \{\langle \text{x}, c_1\rangle\}) \cup \{\langle \text{x}, c_2\ op\ c_3\rangle\}$$

其中，减法表示从集合中消除一个项。依次建模块中的每一个操作产生在这个块的尾部成立的常量集合。
453 而这一结果被用在计算p的后继的CONSTANTS集合的∧操作中。

编译器设计者可以扩展这一模型使其包含特殊情况。如果一个操作中的一个操作数是未知的，而其他操作数已知是这个操作符的零元或单位元，那么这一模型可以确定结果值。很多操作符都有这一模型可以检查的零元或单位元。

CONSTANTS的定义域较大但是是有限的。因为特定块的CONSTANTS集合中的特定变量的值只能改变两次（从非特定的到c_i到⊥），所以在实践中，迭代算法在这一问题的实例上运行得很快。

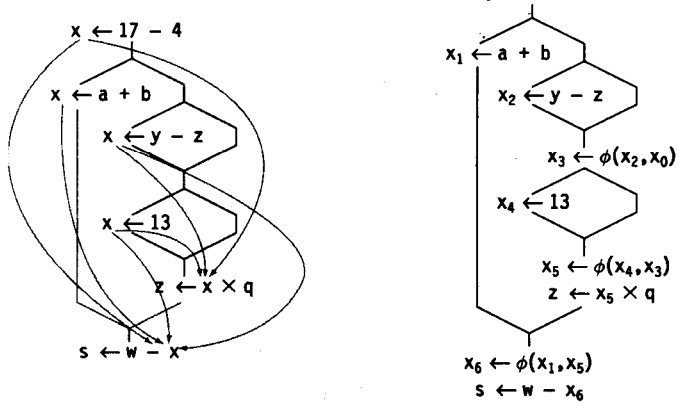
编译器可以直接把CONSTANTS集合中的信息用于改进代码。所有参数都是常量的操作可以在编译时评估；任意对这一评估结果值的引用都可以转换成使用一个文字值的引用。测试中的常量值可以消除分支。复杂计算中的常量值可以创建简化的机会。把常量叠入引用的位置是编译器可以运用的更有效的转换方法之一（参见10.3.3节和10.4.1节中更简单、更强大的算法。）

9.3 静态单一赋值形式

长久以来，很多不同的数据流问题已被公式化。如果每一个转换使用它自己特定的分析，那么花费在实现、调试和维护分析遍的时间和精力会毫无理由地增大。为了限制编译器设计者必须实现和编译器必须运行的分析量，理想的做法是使用单一分析实现多个转换。

实现这一“普遍”分析的一个策略是构建这一程序的一个变形，它把数据流和控制流直接编码到IR中。5.5节和8.5.1节引入的静态单一赋值（static single-assignment，SSA）形式具有这一性质。它可以充当许多转换的基础。根据把代码翻译成SSA形式的单一实现，编译器可以执行很多典型标量优化。

考虑图9-6中左边所示的代码片段中的变量x的不同使用。灰色的线给出可以达到的x的每一次使用的定义。而这一图的右侧给出相同的片段，只是经过重写把x变成了SSA形式。用下标给x的定义再命名
454 以确保每一个定义有惟一一个SSA名字。为简单起见，我们保留对其他变量的引用不变。



源代码片段　　将x变成SSA形式

图9-6 SSA：在数据流中编码控制流

上面代码的SSA形式包含（对x_3、x_5和x_6的）新赋值，这使x的不同SSA名字与（在对s和z的赋值中）x的使用一致。这些赋值保证沿着CFG中的每一条边x的当前值被赋有惟一的名字，且这一名字与控制进入这一边的路径无关。这些赋值的右部包含一个特殊的函数，ϕ函数，它把不同边的值结合起来。

ϕ函数取与进入这个块的每一个边相关的值的SSA名字作为参数。当控制进入一个块时，这个块中的所有ϕ函数并行执行。它们评估对应于控制进入这个块的边的参数。对应于从左到右的边，我们从左到右书写这些参数。在打印页面上，这很容易。在实现中，它则需要某种簿记。

SSA构造法在CFG中多条路径汇合的每一个点之后插入一个ϕ函数，即在每一个连接点之后插入ϕ函数。在连接点，不同的SSA名字必须被调整成单一名字。在整个过程被转换成SSA形式之后，两个规则成立：（1）这一过程中的每一个定义创建惟一的名字，（2）每一次使用引用一个定义。为了把一个过程转换成SSA形式，编译器必须为每一个变量把适当的ϕ函数插入代码中，而且它必须使用下标给变量再命名使得前两个规则成立。这一简单的两步设计产生基本SSA构造算法。 455

9.3.1 构建SSA形式的简单方法

为了构建一个程序的SSA形式，编译器必须在CFG中的连结点处插入ϕ函数，而且它必须重命名变量和临时变量以便与控制SSA名字空间的规则一致。算法遵从以下要点：

1）*插入ϕ函数*。在有多个前驱的每个块的开始处，为代码在当前过程中或定义或使用的每一个名字y插入诸如$y \leftarrow \phi(y, y)$这样的ϕ函数。ϕ函数应该对CFG中的每一个前驱块有一个参数。这一规则要求在每一个需要ϕ函数的地方插入ϕ函数。它也插入很多多余的ϕ函数。

算法可以以任意顺序插入ϕ函数。ϕ函数的定义要求一个块的顶点的所有ϕ函数并发执行，即它们同时读取它们的输入参数，然后再同时写出它们的输出值。这导致算法避免顺序带来的很多不重要的细节。

2）*重命名*。插入ϕ函数之后，编译器可以计算可达定义（参见9.2.4节）。因为这些已插入的ϕ函数是定义，它们确保对于任意使用只有一个定义可以达到。其次，编译器可以重新命名变量和临时变量的每一次使用以反映达到它的定义。编译器必须挑出达到每一个ϕ函数的定义，并使这些名字与这些定义到达的包含这一ϕ函数的块的路径相对应。这需要某种簿记，但相当直观。

这一算法构造出程序的正确的SSA形式。每一个变量只被定义一次，且每一次引用使用一个特定义的名字。然而，结果的SSA形式可能有许多不需要的ϕ函数。这些多余的ϕ函数成为一个问题。它们降低在SSA形式上执行的某些分析的精确性。它们占据空间，所以编译器浪费内存来表示或者是冗余的（即$x_j \leftarrow \phi(x_i, x_i)$）或者是非活的$\phi$函数。它们增加所有使用结果SSA形式算法的代价，因为这一算法必须遍历所有多余的ϕ函数。 456

我们称SSA的这种版本为“*极大SSA形式*（maximal SSA form）”。构建使用更少ϕ函数的SSA形式需要更多的工作；特别是编译器必须分析代码以确定不同的值可能在CFG中的汇合地点。这一计算依赖于我们在8.5.2节所介绍的支配的概念。

以下三小节详细地给出构建*半剪枝SSA形式*（semipruned SSA form）的算法，这是一个使用较少ϕ函数的算法版本。9.3.2节给出一个*快速算法*，这一算法计算插入ϕ函数所需要的支配信息。9.3.3节给出一个插入ϕ函数的算法，而9.3.4节给出如何重写变量名来完成SSA形式的构造。9.3.5节讨论把代码转换回到可执行形式时出现的困难。

9.3.2 支配

最大SSA形式的最基本问题是它包含太多的ϕ函数。为了减少ϕ函数的数量，编译器必须更仔细地确定需要它们的位置。高效且准确地实现这一点的关键是支配信息。本节开发一种数据流框架，它计算支配者并引入*支配边界*（dominance frontier）的概念。下一节使用支配边界改进ϕ函数的配置。

1. 计算支配者

支配是编译的一个最古老的思想。在CFG中，*结点i支配*（dominate）*结点j*，如果从入口结点到j的

每一条路径都通过结点i。已经提出了很多计算支配者的方法。在实践中，简单的数据流方法或者更复杂的算法都可完成这一工作。

为了把支配计算公式化为数据流问题，我们让编译器使用DOM集合注释每一个结点。形式上，$i \in \text{DOM}(j)$当且仅当i支配j。根据定义，一个结点支配其本身，所以$i \in \text{DOM}(i)$。DOM计算的数据流方程很简单：

$$\text{DOM}(n) = \{n\} \cup \left(\bigcap_{m \in \text{preds}(n)} \text{DOM}(m) \right)$$

457

使用初始条件$\text{DOM}(n_0) = \{n_0\}$且对任意的$n \neq n_0$，$\text{DOM}(n) = N$，其中$N$是CFG中的结点的集合。从实现的角度看，最初保留这些集合的初始值为空集而不是$\text{DOM}(n_0)$，且实现忽略空集的集合交运算符会更有效。

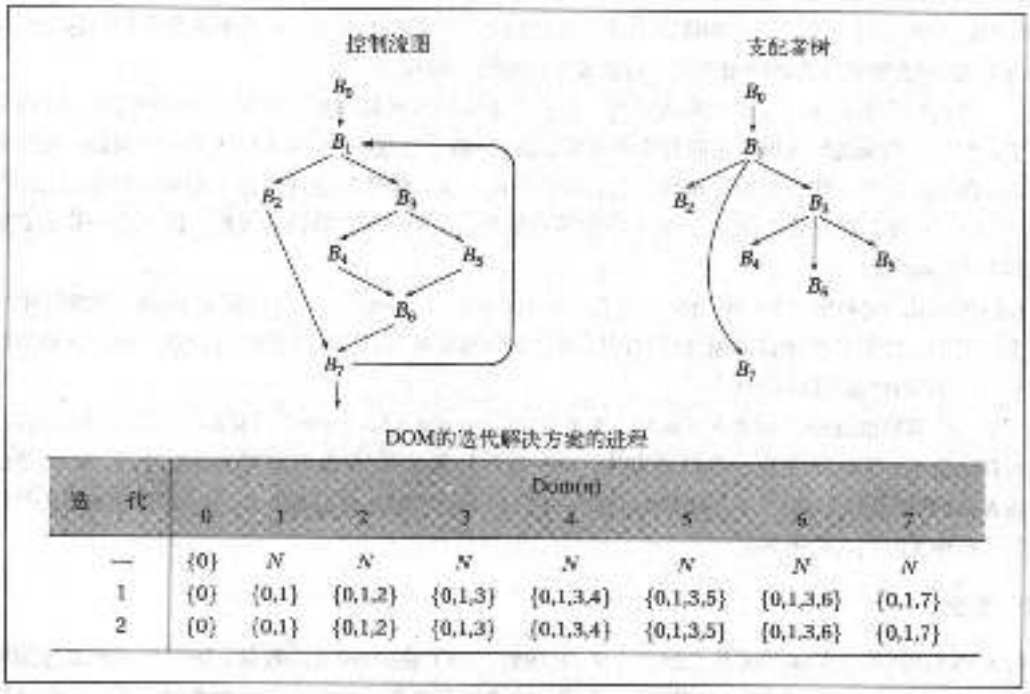
这一公式是直观的，$\text{DOM}(n)$是n的前驱的DOM集合的交集，再加上n本身。这一交集找到n的前驱的共同祖先。不在交集内的任意结点$m \neq n$不能位于从n_0到n的所有路径上。

可以使用迭代算法求解DOM的这些方程。算法很快就会发现这一问题的实例的惟一不动点解。

一个例子

458

考虑图9-3中的CFG。它的简图如图9-7的左上方所示。结点标签形成一个RPO编号，这是通过在访问左子结点之前访问右子结点的遍历计算而来的。使用迭代算法并使用上述的顺序取结点将产生上图底部的表所示的结果。



DOM的迭代解决方案的进程

迭代	Dom(n) 0	1	2	3	4	5	6	7
—	{0}	N	N	N	N	N	N	N
1	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,3,5}	{0,1,3,6}	{0,1,7}
2	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,3,5}	{0,1,3,6}	{0,1,7}

图9-7 LIVEOUT例子的支配关系

此图的简单结构使得算法在迭代标签为1的一遍就可找到正确的DOM集合。后边（即从较高编号结点到较低编号结点的边，本例中是$<B_7, B_3>$）对$\text{DOM}(B_3)$不添加任何东西，因为$\text{DOM}(B_3)$被初始化为N。

这一算法需要第二个遍来确认这些集合不再改变。

在图9-7的右上方所示的树是*支配者树*（dominator tree）的一个例子。（回想一下我们在8.8.2节中曾在基于支配者的值编号中使用过这一数据结构。）在这一支配者树中，结点n是它的立即支配者IDOM(n)的子结点。在这一例子中，B_4、B_5和B_6是B_3的子结点，即使控制通过B_4或B_5达到B_6。

2. 改进这一方法的效率

支配者的迭代框架既简单又直观。然而，由于DOM集合的稀疏性，直截了当的实现相对来说效率不高。我们可以通过注意到这一算法只需要存储每一个结点的IDOM这一事实改进迭代支配计算的效率。它可以从IDOM计算所有其他信息。

IDOM的关系即编码这一图的支配者树，又编码树中每一个结点的DOM集合。在图9-7中，IDOM(6)是3，IDOM(3) 是1，IDOM(1) 是0。6的DOM集合是{0，1，3，6}，这些结点刚好是位于支配者树中从6到0的路径上的结点。如果我们隐式地把结点自身包含在它自己的DOM集合中，那么我们可以通过从IDOM(n) 到根遍历支配者树重建DOM(n)。

为了把IDOM用作DOM集合的代理，我们需要计算两个DOM集合的交集的快捷方法。当我们遍历支配者树重新创建DOM集合时，我们以固定顺序遭遇DOM(n) 中的结点。把这一集合想像为一个列表。按这一顺序，两个DOM集合的交集将是它们的共同前缀。我们可以通过从每一个表的尾部开始向前端遍历各表，直到找到一个共同的结点为止来计算这个交集。这一表的其余部分一定是相同的，第一个共同结点可以表示由交运算产生的DOM集合。

现在出现一个复杂性问题。在计算DOM(i) ∩ DOM(j) 中，这一算法在每一点处都需要知道是从i，从j，还是从二者出发进一步向前遍历。如果我们使用结点RPO编号来命名这些结点，那么交集例程可以只比较这些数字。图9-8给出这一算法。这一算法使用两个指针沿树向上跟踪路径。当这两个指针相遇时，它们都指向代表交运算的结果结点。

```
intersect(i, j)
  finger1 ← i
  finger2 ← j
  while (finger1 ≠ finger2)
    while (finger1 > finger2)
      finger1 = IDOM(finger1)
    while (finger2 > finger1)
      finger2 = IDOM(finger2)
  return finger1
```

图9-8 通过代理求DOM集合的交集

因此，IDOM(n) 是高效寻找DOM(n) 的关键。给定每一个结点的IDOM，我们可以通过从n到n_0向后遍历立即支配者链来得到DOM(n)。这一方法节省空间。所有IDOM集合都只包含一个项，所以每个结点有一个单集。这一方法避免为一个结点分配并初始化DOM集合所需的代价。它最小化数据移动，交运算操作符从其DOM集合被结合的两个结点出发向上遍历，直到找到这两个结点的最近的共同支配者为止。这一操作的结果就是找到那个结点的名字。通过从第一个共同祖先开始向上遍历可以得到两个集合的交集的其他元素。它是高效的；交运算所花的时间与输入集合的大小成正比，而不与图的大小成正比。其结果是既简单又快速的实现。

更加困难的例子

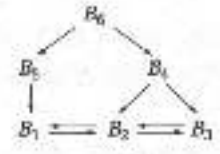
图9-8给出更加难以计算支配信息的例子。这个例子需要4次迭代。图的右侧给出算法在每个阶段的IDOM数组的内容。符号u表示未初始化集合。我们省略了第4次迭代，在这一迭代中IDOM的值不发生变化。

3. 支配边界

放置ϕ函数的关键在于了解在每一个连接点处哪些变量需要ϕ函数。为了有效而高效地解决这一问题，编译器可以对这一问题进行转换。对于每一个定义点，编译器可以确定需要由这个定义所创建的值的ϕ函数的连接点集合。支配在这一计算中起着重要的作用。

459

460



n	IDOM(n) 初始值	第1次迭代后	第2次迭代	第3次迭代
B_6	6	6	6	6
B_5	u	6	6	6
B_4	u	6	6	6
B_3	u	4	4	6
B_2	u	4	6	6
B_1	u	6	6	6

图9-9 形态更复杂的图

考虑CFG中的结点n中的一个定义。这个值可能无需ϕ函数就能达到满足$n \in \mathrm{DOM}(m)$的每一个结点m，因为达到m的每一条路径都经过n。这个值不能达到m的惟一方式是当相同名字的另一个定义出现在中间时，即这一定义出现在n与m之间的某个结点p处。在这种情况下，n中的定义不强制ϕ函数的存在；相反，在p处的重新定义强制ϕ函数的存在。

结点n中的定义迫使在n支配的CFG区域外部最近的连接点处有一个ϕ函数。更形式化地说，结点n中的定义迫使相应的ϕ函数存在于满足以下条件的任意连接点结点m处：（1）n支配m的一个前驱（$q \in \mathrm{preds}(m)$且$n \in \mathrm{DOM}(q)$），以及（2）n不严格支配m，即$n \notin \mathrm{DOM}(m) - \{m\}$。（增加“严格支配”的概念允许在一个块循环的开始处存在ϕ函数。在这种情况下，$n = m$且$n \notin \mathrm{DOM}(m) - \{m\}$。）我们称具有这一性质的结点的集合为$n$的**支配边界**（dominance frontier），记作$\mathrm{DF}(n)$。

非形式地，$\mathrm{DF}(n)$包含在离开n的每一条CFG路径上n不支配且从n开始可达的第一个结点的全体。在图9-7的例子中，B_3支配B_3、B_5和B_6，但是却不支配B_7。在每一条离开B_3的路径上，B_7是B_3不支配的第一个结点。因此，$\mathrm{DF}(B_3) = \{B_7\}$。

为了计算支配边界，考虑下面的观察。第一，支配边界中的结点一定是图中的连接点。第二，任意连接点j的前驱，除非支配j，否则必定使j在它们相应的支配边界集合中。这是前述支配边界的定义的直接结果。最后，j的前驱的支配者必定使j在它们的支配边界集合中，除非它们也支配j。

461

```
for all nodes, n
DF(n) ← φ
for all nodes, n
  if n has multiple predecessors then
    for each predecessors p of n
      runner ← p
      while runner ≠ IDOM(n)
        DF(runner) ← DF(runner) ∪ {n}
        runner ← IDOM(runner)
```

图9-10 计算支配边界

这些观察导致一个简单的算法。第一，我们识别每一个连接点j：带有多个入边的任意结点是一个连接点。第二，我们检查j的每一个CFG前驱p，并从p开始向上遍历支配者树，当达到j的立即支配者时，我们停止遍历：除了j的立即支配者之外，对于遍历中遇到的每个结点，j都在该结点的支配边界中。直观上，j的前驱分享j的所有其他支配者，因为它们支配j，所以它们不会使j在它们的支配边界中。

图9-10给出这一算法。需要少量的簿记来确保任意x只被加入到一个结点的支配边界一次。

例子

为了计算图9-7中的图的所有支配边界集合，我们需要CFG及CFG中每一个结点的IDOM。我们可以从如图9-7的右上方所示的早前构建的支配者树读取每一个结点的DOM和IDOM。

n	0	1	2	3	4	5	6	7
DOM(n)	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,3,5}	{0,1,3,6}	{0,1,7}
IDOM(n)	{—}	{0}	{1}	{1}	{3}	{3}	{3}	{1}

为了计算支配边界，分析器在CFG中选择一个连接点，并对这一连接点的每一个前驱向上遍历这一支配者树。本例的CFG有三个连接点：

1) B_2：分析器从B_4向后遍历到B_1，把B_2加到DF(B_2)中。它从B_4向后遍历到B_3，把B_3加到DF(B_4)中。

2) B_7：它从B_7向后直接遍历到B_3，把B_7加到DG(B_3)。它从B_3向后遍历到B_2再到B_1，把B_7加到DF(B_3)和DF(B_1)。

3) B_1：B_1没有立即支配者，所以$B_1 \notin$DF(B_1)。从B_1开始遍历，它发现B_1的立即支配者是B_0。因此，它把B_1加到DF(B_1)且不加入其他集合。

累积这些结果，我们得到下面的支配边界：

n	0	1	2	3	4	5	6	7
DF(n)	∅	∅	{7}	{7}	{6}	{6}	{7}	{1}

9.3.3 放置φ函数

朴素的算法在每一个连结结点的开始处为每一个变量放置φ函数。使用支配边界，编译器可以更精确地确定需要φ函数的位置。基本想法是简单的。块b中的x的定义在DF(b)中的每一个连接结点处强制一个φ函数。因为这个φ函数是x的一个新定义，所以它可能迫使多余的φ函数的插入。

编译器可以进一步缩小插入的φ函数集合。只在单一块中使用的变量从来没有活的φ函数。为了利用这一发现，编译器可以计算在多个块间活着的名字集合。我们称这个集合为全局名字（global name）。编译器可以为全局名字插入φ函数，同时忽视在多个块间从不是活着的所有名字。（这一限制把半剪枝SSA形式与其他类型SSA形式区别开来。）

编译器可以很廉价地找到全局名字。在每个块中，编译器检查具有向上暴露使用的名字，即活变量计算中得到的UEVAR集合。在一个或多个LIVEOUT集合中出现的任意名字一定在某个块的UEVAR集合中。所有这些UEVAR集合的并集给出在一个或多个块的入口处是活着的名字的集合，因此它是在多个块中活着的名字的集合。

图9-11左侧给出的算法得自于LIVEOUT分析所给的计算，这一算法构造单一集合*Globals*。在这一集合中，LIVEOUT的计算必须为每个块计算一个集合。在它构建*Globals*集合的同时，对于每个名字，它还构建包含这个名字的定义的所有块的列表。这些块列表充当φ函数插入算法的初始工作表（Worklist）。

插入φ函数的算法如图9-11右侧所示。对于每一个全局名字x，这一算法使用*Blocks*(x)初始化*Worklist*。对于*Worklist*中的每一个块b，算法在块b的支配边界中的每个块d的头部插入φ函数。根据定义，因为一个块中的所有φ函数并行执行，所以这一算法可以以任意顺序在d的头部插入这些φ函数。把x的φ函数加入d之后，算法把d加入以*Worklist*中以反映d中x的新赋值。

```
Globals ← ∅
Initialize all the Blocks sets to ∅
for each block b
    VARKILL ← ∅
    for each operation i in b, in order
        assume that op; is "x ← y op z"
        if y ∉ VARKILL then
            Globals ← Globals ∪ {y}
        if z ∉ VARKILL then
            Globals ← Globals ∪ {z}
        VARKILL ← VARKILL ∪ {x}
        Blocks(x) ← Blocks(x) ∪ {b}
```

查找名字

```
for each name x ∈ Globals
    WorkList ← Blocks(x)
    for each block b ∈ WorkList
        for each block d in DF(b)
            insert a φ-function for x in d
            WorkList ← WorkList ∪ {d}
```

插入φ函数

464

图9-11 插入φ函数

为了改进效率，编译器应该避免两种重复。第一，算法应该避免对每一个全局名字在工作表中多次放置任意块。它可以保存已处理过的块的清单。因为算法必须为每一个全局名字重置清单，所以这一实现应该使用稀疏集合或类似的结构（参见B.2.3节）。

第二，一个块可以出现在*WorkList*中多个结点的支配边界中。为了避免把变量i的φ函数重复插入一个块中，编译器可以维护已经包含了i的φ函数的块的清单。这要求一个稀疏集合，它与*WorkList*一同被再初始化。另一个选择是在这个块内搜索现存的φ函数；清单方法也许更快。

举例

把这一算法运用到图9-3和图9-7所示的例子的第一步是计算*Globals*集合和*Blocks*集合。*Globals*是{a，b，c，d，i}，而*Blocks*集合是：

名字	a	b	c	d	i	y	z
Blocks	{1,3}	{2,6}	{1,2,5}	{2,3,4}	{0,7}	{7}	{7}

注意，这一算法为y和z创建*Blocks*集合，即使y和z不在*Globals*内。把*Globals*的计算从*Blocks*的计算中分离出来可以避免实例化这些额外的集合，但要以代码上的另一次遍历为代价。

这一算法还需要CFG的支配边界。我们计算这一支配边界为：

x	0	1	2	3	4	5	6	7
DF(x)	∅	∅	{7}	{7}	{6}	{6}	{7}	{1}

使用这一信息，分析器可以对图9-11的右侧运用所示的算法。

考虑算法对变量a所做的工作。因为a在*Blocks*(a)=$\{B_1, B_3\}$有一个定义，这一算法必须在DF(B_1)=∅和DF(B_3)=(B_7)中的每一个结点处插入一个φ函数。把这个φ函数加入B_7并把B_7加入工作表。（B_7中的φ函数本身就是a的一个定义。）这一算法在DF(B_7)=$\{B_1\}$的每一个块中插入一个φ函数。因为B_1已在工作表中，所以这一算法既不插入重复的φ函数也不把DF(B_1)加到工作表中。这就完成算法对a的工作。对于*Globals*中的每一个名字，算法做同样的工作，产生如下的插入：

全局名字	a	b	c	d	i
有φ函数的块	{7,1}	{7,1}	{7,6,1}	{7,6,1}	{1}

465

结果代码如图9-12所示。

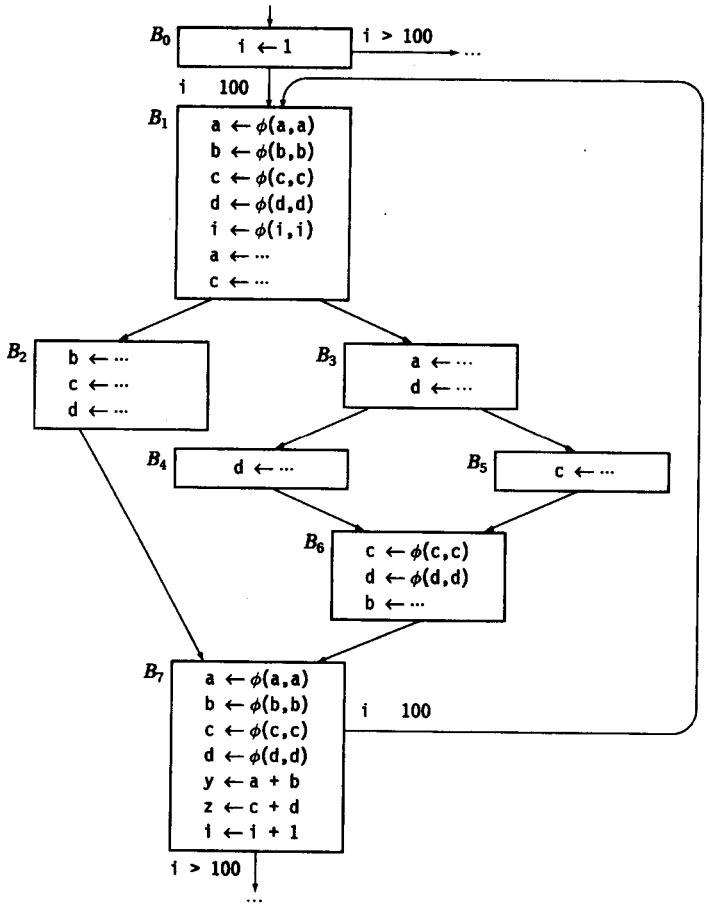


图9-12 使用φ函数的例子

把这一算法限制于全局名字上使其避免在块B_1中为x和y插入死φ函数。(B_1是DF(B_7)，而B_7包含x和y的定义。）然而，局部名字和全局名字之间的差异不足以避开所有死φ函数。块B_1中a和c的φ函数是死的，因为两个名字在它们的值被使用之前都被重新定义。为了避免插入这些死φ函数，编译器必须构造LIVEOUT集合，并把一个基于活性的试测加入φ函数插入算法的内循环中。

9.3.4 重命名

在对极大SSA形式的描述中，我们提到重命名变量在概念上是直观的。然而，其细节需要进一步解释。

在最终的SSA形式中，每个全局名字有单一基名字，而且我们通过在基名字的各个定义中加入数字下标来区分它们。对应于源语言变量，如x，的一个名字，算法把x用作基名字。因此，重命名算法在遇到x的第一个定义时将其命名为x_0，而遇到的x的第二个定义时将其命名为x_1。对于编译器生成的临时变量，算法必须生成不同的名字。

重命名算法在前序遍历支配者树的过程中重命名定义及其使用。在每个块中，它首先重命名在这个块的头部的φ函数定义的值，然后，它依次访问这个块中的每一个操作。它使用当前SSA的名字重写每一个操作的操作数，然后对这一操作的结果创建一个新的SSA名字。后者的动作使这个新名字为当前名

字。当块中所有操作都被重写之后，算法使用当前SSA的名字重写这一块的每一个CFG后继中的适当ϕ函数的参数。最后，它对支配者树中这个块的所有子结点重复这一工作。当它从这些递归调用返回时，它再把当前SSA名字的集合恢复到当前块被访问之前已存在的状态。

为了管理这一过程，编译器为每一个全局名字使用一个计数器和一个栈。名字栈保存这个名字的最近下标，即这个名字的当前SSA名字。在每一个定义处，算法通过把这一全局名字的当前计数器的值压入栈中并递增计数器为这个目标名字生成一个新下标。栈顶的值是新的SSA名字。作为处理一个块的最后一步，算法把在那个块中所生成的所有名字都从相应的栈弹出。这恢复在这个块的立即支配者中保存的名字集合，如果需要，可以在支配者树中这个块的兄弟结点复用这个栈。

466 栈和计数器用于不同的目的。当算法中的控制上下游走于支配者树时，栈被指定来模拟当前块中最近定义的生存期。另一方面，计数器单调递增以确保每一个后继对<*name*, *subscript*>映射到不同的定义。

SSA形式的不同风格

文献中已提出了SSA形式的若干不同风格。各个风格的不同在于它们插入ϕ函数的标准不同。对于给定程序，这些SSA形式产生不同的ϕ函数集合。

极小SSA 在原来相同的名字相遇的两个不同定义的所有连接点处插入一个ϕ函数。这是与SSA的定义一致的最小数目。然而，其中的一些ϕ函数可能是死的；定义并没有说明当值相遇时它们必须是活的。

剪枝SSA 给ϕ插入算法加上一个活性测试以确保只加入活的ϕ函数。这一构造法必须计算LIVEOUT集合，所以构建剪枝SSA的代价比构建最小SSA的代价要高。

半剪枝SSA 是最小SSA和剪枝SSA的折衷。在插入ϕ函数之前，这一算法消除所有在跨越块边界时非活的名字。从而缩小名字空间且在没有增加计算LIVEOUT集合的负荷的情况下减少ϕ函数的数量。这就是图9-11给出的算法。

当然，ϕ函数的数量依赖于被转换成SSA形式的特定程序。对于某些程序，通过半剪枝SSA和剪枝SSA得到的ϕ函数数量的减少是显著的。缩小SSA形式可以导致更快的编译，因为使用SSA形式的遍可以在包含较少操作，即含有较少ϕ函数的程序上的操作。

图9-13概括出这一算法。它初始化栈和计数器，然后，在支配者树的根部调用*Rename*，这是CFG的

```
for each global name i
  counter[i] ← 0
  stack[i] ← ∅
Rename(n0)

NewName(n)
  i ← counter[n]
  counter[n] ← counter[n] + 1
  push ni onto stack[n]
  return ni

Rename(b)
  for each φ-function in b, "x ← φ(···)"
    rename x as NewName(x)
  for each operation "x ← y op z" in b
    rewrite y as top(stack[y])
    rewrite z as top(stack[z])
    rewrite x as NewName(x)
  for each successor in the CFG
    fill in φ-function parameters
  for each successor s in the dominator tree
    Rename(s)
  for each operation "x ← y op z" in b
    and each φ-function "x ← φ(···)"
    pop(stack[x])
```

图9-13 ϕ函数插入后的重命名

入口结点。*Rename*重写这个块并在这一支配者树中的后继上进行递归调用。在结束这个块上的操作时，*Rename*弹出处理这个块时压入栈中的所有名字。函数*NewName*处理计数器和栈来创建所需的新名字。 467 ~ 468

还有最后一点细节。在块*b*的尾部，*Rename*必须重写*b*的每一个CFG后继中的ϕ函数的参数。编译器必须在那些*b*的ϕ函数中指定一个顺序参数槽。当我们画出SSA形式时，我们总是假设从左到右的顺序，这与绘制边时的从左到右顺序相匹配。在内部，编译器可以以任意产生理想结果的相容方式对边和参数槽进行编号。这需要构建SSA形式的代码与构建CFG的代码之间的合作。（例如，如果CFG实现使用离开每个块的边的列表，那么这一列表的顺序可以决定映射。）

1. 举例

为了完成我们的例子，让我们对图9-12运用重命名算法。假设a_0、b_0、c_0和d_0在B_0的入口被定义。图9-14给出这一过程中各个不同点处的全局名字计数器以及栈的状态。带标签“Before B_i”的图展示当*Rename*被块B_i调用时的计数器和栈。带标签“End of B_i”的图给出在处理这个块之后且*Rename*从各个栈弹出B_i的名字之前的状态。图9-14给出每个块的尾部的状态；为了清晰起见，它还包含B_0、B_3和B_7的入口状态。 469

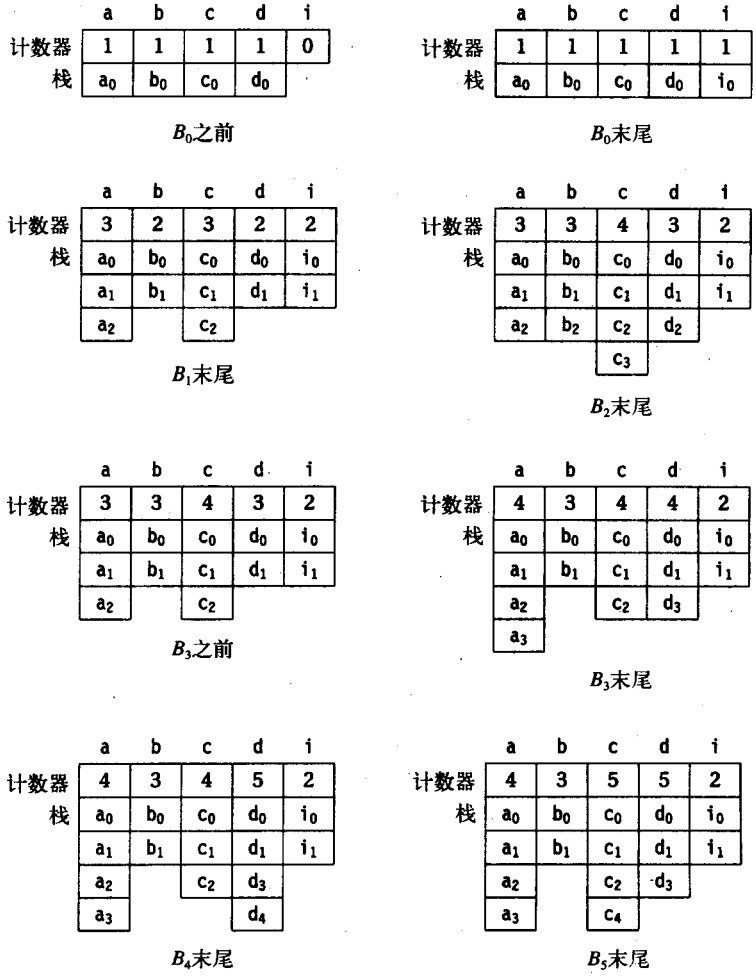


图9-14 重命名例子中的状态

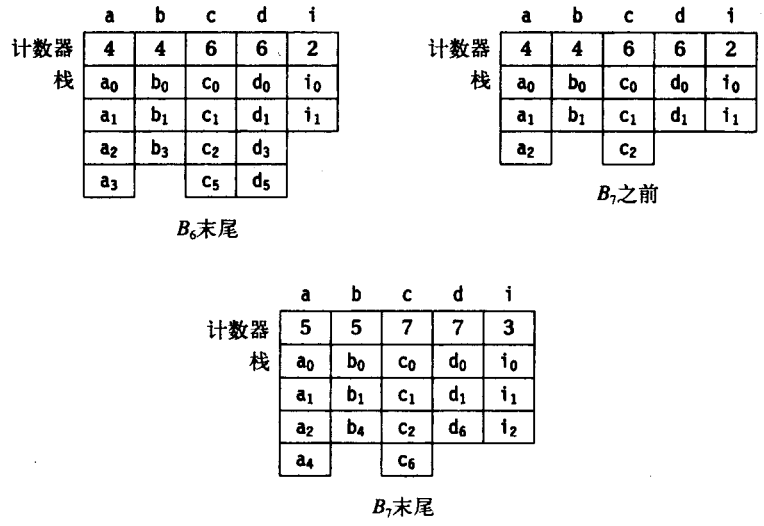


图9-14 （续）

这一算法前序遍历支配者树以从B_0到B_7的递增顺序访问各结点。当它处理各块时，它采取下面的动作:

1）B_0。这个块只包含一个操作。*Rename*使用i_0重写i，递增计数器，并把i_0压入i的栈。接下来，它用当前的状态a_0、b_0、c_0、d_0和i_0中适当的名字重写每一个ϕ函数的第一个参数。然后，它在B_0在支配者树中的子结点B_1上递归进行相同的操作。在这之后，它对i的相关栈做弹出操作并返回。

2）B_1。在进入B_1时，*Rename*使用新名字a_1、b_1、c_1、d_1和i_1重写ϕ函数的目标。接下来，它把a和c的定义重写为a_2和c_2的定义。B_1的CFG后继都没有ϕ函数。接下来，它在B_1的支配者树中的子结点B_2、B_3上递归进行相同的操作。它弹出这些栈并返回。

470 ~ 471

3）B_2。这一块没有需要重写的ϕ函数。接下来，*Rename*重写三个操作，创建b_2、c_3和d_2。然后，它使用保存在B_2的尾部的适当名字a_2、b_2、c_3和d_2取代B_2的CFG后继B_7中的每一个ϕ函数的第一个参数，以此来重写ϕ函数的参数。最后，它弹出这些栈并返回。

4）B_3。接下来，*Rename*再次运用在B_3上。检查标签为“Before B_3”的图。它拥有来自“End of B_1”的栈和来自于“End of B_2”的计数器。它使用a_3和d_3重写两个赋值。B_3的所有CFG后继都没有需要处理的ϕ函数。*Rename*递归运用与B_3在支配者树的子结点B_4、B_5和B_6上。它弹出这些栈并返回。

5）B_4。这个块有一个操作。*Rename*重写d为d_4。接下来，它使用相应的当前名字c_2和d_4重写B_6中每一个ϕ函数的第一个参数。它弹出d的栈并返回。

6）B_5。这个块也有一个操作。*Rename*重写c为c_4，然后重写B_6中每一个ϕ函数的第二个参数。这些参数都变成c_4和d_3。它弹出c的栈并返回。

7）B_6。Rename重写ϕ函数的目标为c_5和d_5。它重写给对b的赋值为对b_3的赋值。接着，它使用当前SSA名字（a_3、b_3、c_5和d_5）重写B_7中每一个ϕ函数的第二个参数。因为B_6在支配者树中没有子结点，所以它向上返回到B_3，在此它返回到B_2。它向下对B_2的最后一个支配者树的子结点B_7递归进行同样的操作。它弹出这些栈并返回。

8）B_7。首先，*Rename*重写ϕ函数的目标，创建a_4、b_4、c_6和d_6。它重写接下来的两个赋值中的全局名字的使用，但不重写它们的目标，因为y和z都不是全局名字。在最后一个赋值中，它使用i_1重写这一使用，然后为这个定义创建一个新名字i_2。

B_7只有一个CFG后继B_1。*Rename*使用它的当前SSA名字（a_4、b_4、c_6、d_6和i_2）重写B_1中每一个ϕ函

数的第二个参数。*Rename*弹出栈，返回到B_1，再到B_0，然后停止。

图9-15给出*Rename*停止后的代码。 472

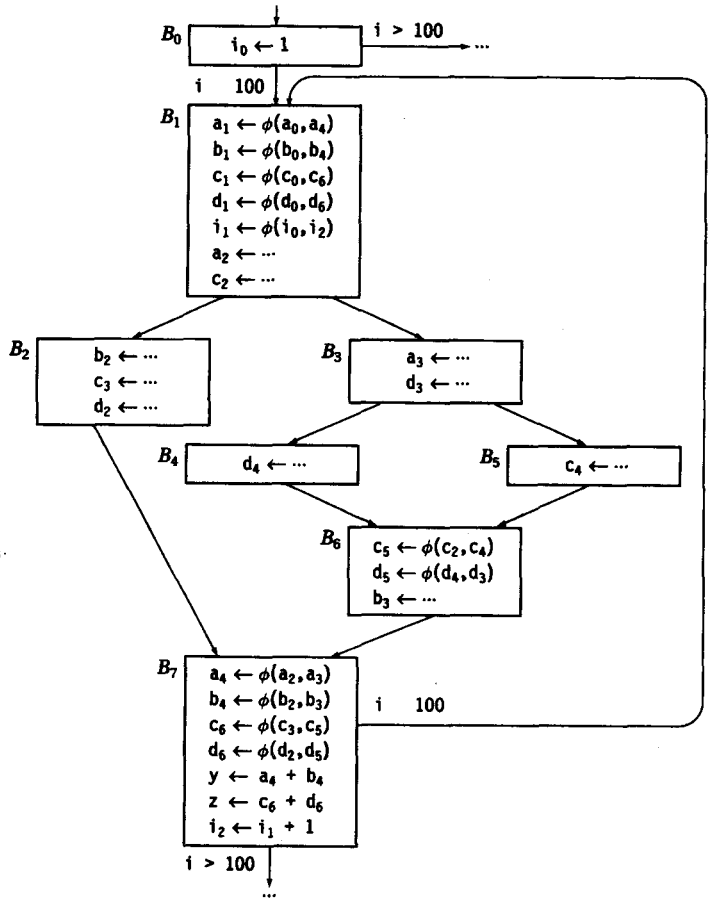


图9-15 重命名后的例子 473

2. 最后的改进

*NewName*的巧妙实现可以减少消耗在栈处理上的时间和空间。栈的主要使用是在块的出口处重新设置名字空间。如果一个块几次重新定义相同的基名字，那么*NewName*只需要保存最新的名字。这发生在本例中的块B_1中的a和c上。*NewName*可能在单一块内多次覆写相同的栈槽。

这使得栈的最大大小可以预测；没有栈能比支配者树的深度大。这降低整个空间的需求量，避免每次压入栈时进行溢出测试，并减少压入和弹出操作的数量。我们需要另一种机制来确定在一个块的出口对哪些栈做弹出操作。*NewName*可以线索化一个块的栈入口。*Rename*可以使用这一线索对适当的栈做弹出操作。

9.3.5 从SSA形式重新构造可执行代码

因为处理器不实现ϕ函数，所以编译器必须把SSA形式翻译回可执行代码。查看我们的例子，我们非常愿意相信编译器可以简单地消除名字的下标并删除ϕ函数。如果编译器仅构建SSA形式，并把SSA转换回可执行代码，那么这一方法可行。然而，依赖于SSA名字空间的转换可能致使这一简单的重命名过程生成不正确代码。

在局部值编号（LVN）中，我们已看到使用SSA名字空间使转换可以发现和消除更多的冗余。

初始名字空间		名字空间	
Before LVN	After LVN	Before LVN	After LVN
$a \leftarrow x+y$	$a \leftarrow x+y$	$a_0 \leftarrow x_0+y_0$	$a_0 \leftarrow x_0+y_0$
$b \leftarrow x+y$	$b \leftarrow a$	$b_0 \leftarrow x_0+y_0$	$b_0 \leftarrow a_0$
$a \leftarrow 17$	$a \leftarrow 17$	$a_1 \leftarrow 17$	$a_1 \leftarrow 17$
$c \leftarrow x+y$	$c \leftarrow x+y$	$c_0 \leftarrow x_0+y_0$	$c_0 \leftarrow a_0$

474 上表左侧的代码给出一个有四个操作的块，以及当LVN使用来自源代码的名字空间时，它所产生的结果。右侧代码给出使用SSA名字空间的相同的例子。因为SSA名字空间给a_0一个不同于a_1的名字，所以LVN可以使用对a_0的引用取代最后操作中的x_0+y_0的评估。

然而，请注意，去掉变量名字上的下标产生不正确的代码，因为它给c赋值17。诸如代码移动和拷贝叠入等更大胆的转换可能以某种方式重写SSA形式，而这种方式可能引发更微妙的问题。

为了避免这样的问题，编译器可以保存SSA名字空间的完整性，并使用一组拷贝操作取代每一个ϕ函数，即沿着每一个进入边使用一个拷贝操作。对于ϕ函数$x_{14} \leftarrow \phi(x_{12}, x_{13})$，编译器应该沿着携带值$x_{12}$的边插入$x_{14} \leftarrow x_{12}$并沿着携带值$x_{13}$的边插入拷贝$x_{14} \leftarrow x_{13}$。图9-16给出$\phi$函数被拷贝操作取代后的运行例子。$B_7$中的四个$\phi$函数已被$B_2$和$B_4$中的各自四个拷贝取代。同样地，$B_6$中的两个$\phi$函数引发出$B_4$和$B_5$中的各自一对拷贝。对于这两种情况，编译器可以把拷贝插入前驱模块中。

B_1中的ϕ函数揭示出更复杂的情况。因为它的前驱块有多个后继，所以编译器不能在它的前辈的尾部插入拷贝操作。把拷贝直接插入前驱中将促使这些拷贝在循环的出口路径（标签i＞100）中执行。没有关于沿着这些出口路径可以达到的代码的详细信息，我们不能告知插入的这些拷贝是否会产生不正确的行为。然而，它一般可能产生不正确的行为。编译器也不能在B_1的顶部插入拷贝操作；这些拷贝必须使用这样的名字，通过这些名字，在前驱块中这些值是可知的。

为了解决这一问题，编译器可以分离从B_0到B_1和从B_7到B_1的边，并在各边的中间插入一个块来保存这些拷贝，如图9-16所示。源头有多个后继且目标有多个前驱的边称为一个*临界边*（critical edge）。插入块B_8和B_9破坏这一临界边。插入拷贝之后，本例表面上出现多个多余的拷贝。幸运的是，编译器可以使用诸如拷贝叠入等一系列优化尽可能地（如果不是完全）消除它们（参见13.5.6节）。

分离临界边为拷贝操作创建必要的位置并消除拷贝插入期间引发的大部问题。然而，它却可能带来两个更微妙的问题。第一个问题称为*无效拷贝问题*（lost-copy problem），这一问题是大胆的程序转换和临界边的组合引发的。第二个问题称为*交换问题*（swap problem），这一问题是某些大胆的程序转换和
475 SSA形式的详细定义间的相互影响引发的。

1. 无效拷贝问题

很多基于SSA的算法要求分离临界边。然而，有时候编译器不能或不应该分离临界边。例如，如果这个临界边是反复执行的循环的后边，那么增加一个带有一个或多个拷贝操作和一个跳转的块可能会对执行速度有不利的影响。同样地，在编译的后期阶段增加块和边可能对非局部调度、寄存器分配和诸如代码替换等优化产生干扰。

无效拷贝问题发生于拷贝叠入和不可分离的临界边的组合。图9-17给出一个例子。左板面的上方给出源代码——一个简单的循环。在下一个板面中，编译器把这一循环转换成了SSA形式并对从i到y的拷贝进行了叠入，使用对i_1的引用取代y的惟一使用。右上方板面给出直接把拷贝插入ϕ函数的前驱块中所产生的代码。这改变循环后用于给z_0的赋值语句中的值。源代码赋给它i的倒数第二个值，而转换后的

代码赋给它i的最后值。左板面的下方表明分离临界边，即循环的后边，产生正确的行为。然而，它把一个跳转加到这一循环的每一次迭代中。

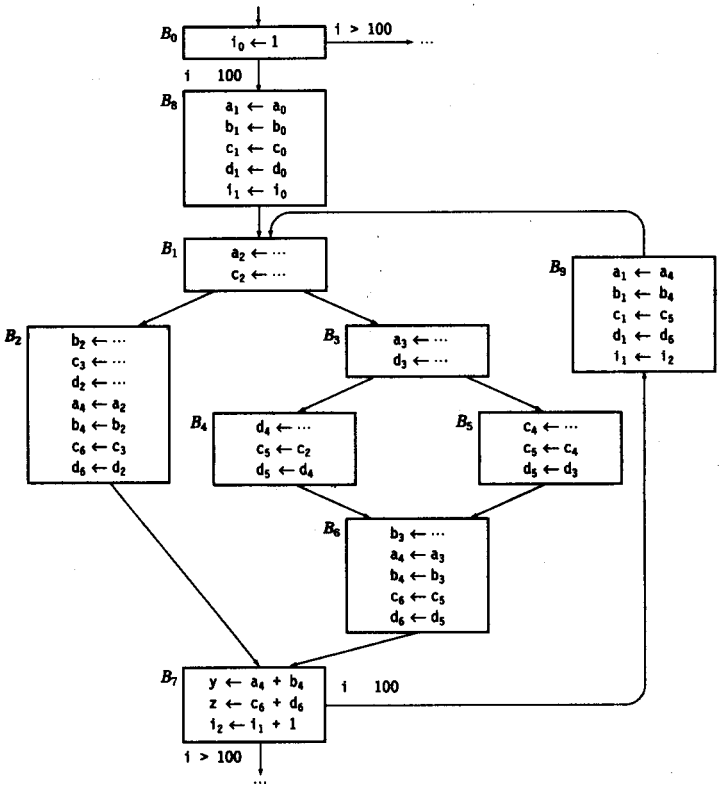
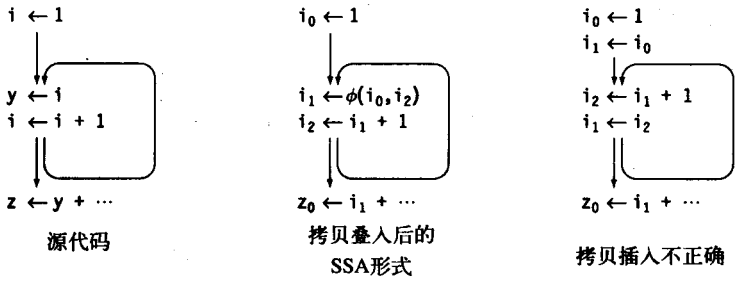


图9-16 拷贝插入后的例子

476



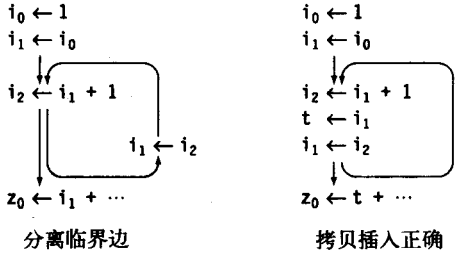


图9-17 无效拷贝问题的例子

477

这一问题是由于不可分离的临界边和名字空间的拷贝叠入处理的组合而产生的。拷贝叠入通过在尾随循环之后的块中把i_1叠入对y的引用来消除赋值$y \leftarrow i$。这延长了i_1的生存期。于是，拷贝插入算法使用那个块的每一个前驱中的拷贝操作取代循环体顶部的ϕ函数。这在这个模块的底部插入拷贝$i_1 \leftarrow i_2$，在这一地点i_1仍然活着。

为了避免无效拷贝，编译器必须发觉它在试图插入一个目标仍然是活的拷贝。当这一情况发生时，它必须把这个值拷贝到一个临时名字，并使用这个新临时名字重写已被覆写的名字的以后的使用。使用模拟SSA重命名算法的一个步骤可以实现这一重写步骤。图9-17右板面底部给出这一方法产生的代码。

2. 交换问题

ϕ函数的性质使得插入算法得以按任意顺序引入ϕ函数。这一性质诱发交换问题。当一个块执行时，它的所有ϕ函数都假设在这个块中的其他所有语句之前同时执行。这个块中的所有ϕ函数同时读取它们的适当的输入参数。然后，它们重定义其目标值，所有这些都在同一时间内进行。

图9-18给出交换问题的一个简单例子。图的左侧给出源代码，这是交换x和y的值的简单循环。图的
478 中央部分给出转换成SSA形式并进行了大胆的拷贝叠入的代码。在这一形式中，由于ϕ函数的评估规则，这一代码保留它原来的意义。当循环体执行时，ϕ函数参数在所有ϕ函数的目标被定义之前被读取。在第一次迭代中，代码在定义x_1和y_1之前读取x_0和y_0。在后继迭代中，它在重新定义x_1和y_1之前读取x_1和y_1。图中的右侧给出经过实施朴素的拷贝插入算法后的相同代码。因为拷贝是顺序执行，而不是同时执行的，所以x_1和y_1不正确地得到相同的值。

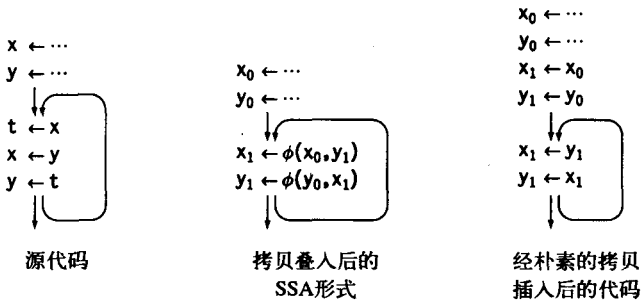


图9-18 交换问题示例

初看起来，似乎分离作为临界边的后边是有帮助的。然而，这只是以同样的顺序在另一个块中放置相同的两个拷贝。为了修复这一问题，编译器可以把这些值中的每一个拷贝到一个临时变量来模拟ϕ函数的规定行为。这在形成循环体的块中产生四个赋值的序列$t \leftarrow y_1$、$s \leftarrow x_1$、$x_1 \leftarrow t$和$y_1 \leftarrow s$。遗憾的是，它使循环中的操作的数量加倍，所以这不是这一问题的最好的解决方案。相反，编译器应该最小化它插入的拷贝数量。

事实上，交换问题不要求拷贝的循环集合；它所取的是一组这样的ϕ函数，这些ϕ函数的输入变量被定义为同一块中其他ϕ函数的输出。对于这样的无循环情况，编译器可以通过对已插入的ϕ函数作细心的排序来避免这一问题。

为了解决这一问题，编译器一般可以查明ϕ函数引用相同模块中其他ϕ函数的目标的情况。对于每一个引用循环，它必须插入一个到临时变量的拷贝来破坏循环。然后，它可以调度拷贝操作来反映ϕ函数所蕴涵的相关性。

9.4 高级话题

本章集中讨论了迭代数据流分析和SSA形式的构造法，这些内容将直接且高效地体现过程中诸多的数据流关系。流向图的一个性质对其他非迭代的数据流算法是重要的；无论这一图是否是可约的(reducible)。9.4.1节讨论流向图的可约性。

本章迄今为止，所有例子都来自于全局（过程间）数据流分析。9.4.2节介绍在把数据流分析的作用域从过程扩展到整个程序时出现的问题。它展示作用域大于单一过程的若干分析问题的主要观点。 479

9.4.1 结构数据流算法和可约性

9.2.2节给出迭代算法，因为这一算法一般在任意图上的合式方程集合上都可行。存在其他数据流算法；其中很多算法的工作模式都是首先得到被分析代码的控制流结构的简单模型，并使用这一模型求解方程。通常，这一模型是由一系列降低复杂性的得到这个图的转换构成的，即以细心的方式将结点或边结合起来。这种图归约过程是除迭代算法之外几乎所有数据流算法的核心。

非迭代数据流算法一般是通过把一系列转换运用到一个流向图来工作的，其中每一个转换选择一个子图并用一个表示这个子图的单一结点取代这一子图。这创建一系列派生图，其中每一个图与序列中它的前驱之间的差异是一个单一转换步骤的效应。当分析器转换这个图时，它计算数据流集合来寻找后继派生图中的新代表结点。这些集合总括被取代子图的效应。这些转换把行为良好的图归约到单一结点。然后，这一算法颠倒这一过程，从它最后的单一结点的派生图开始，返回到原来的流向图。当分析器把这个图展开回到它原来的形式时，它计算每一个结点的最终数据流集合。

本质上，归约阶段收集整个图的信息并把图合并起来，而展开阶段则是把合并起来的集合中的效应传播回到原来图的结点上。任意归约阶段成功的图被认为是可约的（reducible）。如果一个图不能被简化到单一结点，那么它是不可约的（irreducible）。

图9-19给出可以用于测试可约性并构建结构数据流算法的一对转换。T_1消除从一个结点返回自身的自循环边。此图给出作用于b的T_1，记作$T_1(b)$。T_2把只有一个前驱a的结点b叠入回a；它消除边$<a, b>$并使a成为原来离开b的所有边的源头。如果这一过程留下多个从a到某个结点n的边，那么T_2合并这些边。 480
此图给出作用于a和b的T_2，记作$T_2(a, b)$。任意通过反复运用T_1和T_2能够归约成单一结点的图都被认为是“可归约的”。

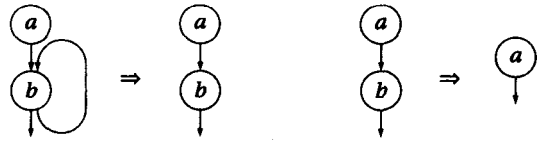


图9-19 转换T_1和T_2

为了理解这是如何工作的，考虑我们的例子的CFG。图9-20给出把这个CFG归约成单一结点图的一个T_1和T_2的使用序列。这一序列反复运用T_2，直到不再存在机会：$T_2(B_1, B_2)$、$T_2(B_3, B_4)$、$T_2(B_3, B_5)$、$T_2(B_3, B_6)$、$T_2(B_1, B_3)$ 和$T_2(B_1, B_7)$。接下来，它使用$T_1(B_1)$ 消除循环，其后跟着$T_2(B_0, B_1)$ 完成归约。因为这一序列把这一图归约成单一结点，所以原来的图是可约的。

其他的运用顺序也可约化此图。例如，从$T_2(B_1, B_3)$ 开始导致一个不同的转换序列。T_1和T_2有有限Church-Rosser性质，这一性质确保最后的结果与运用顺序无关且转换序列终止。因此，分析器可以适时地运用T_1和T_2：寻找图中可以使用其一的地方，并利用这一发现。

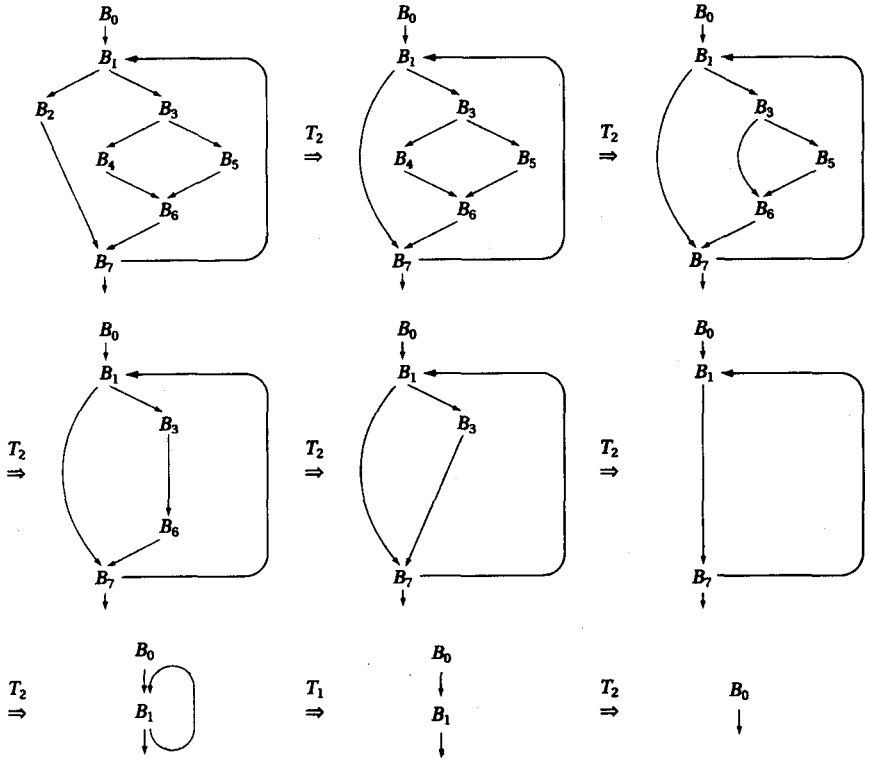


图9-7的示例图

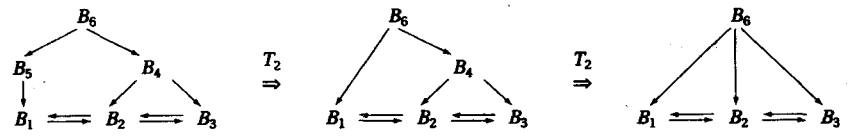


图9-9的示例图

图9-20 示例图的归约序列

图9-20的下半部分给出归约图9-9中的图的一个尝试。分析器使用$T_2(B_6, B_5)$并在其后跟着$T_2(B_6, B_4)$。然而，在这一点没有剩余结点或结点对能够成为T_1或T_2的候选者。因此，分析器不能进一步归约此图。(其他顺序也不能奏效。)此图不可归约到单一结点，它是不可约的。

T_1和T_2对此图归约的失败是由此图的基本性质所致。此图是不可约的，因为它包含一个循环，这一循环有在不同结点进入此结点的边。根据源语言，生成此图的程序有一个带有多个入口的循环。我们可以在这个图中看到这一点；考虑由B_1和B_2形成的循环。这个循环有从B_3、B_4和B_5进入它的边。同样地，由B_2和B_3形成的循环也有从B_1和B_2进入循环的边。

不可归约性对建筑在类似于T_1和T_2这样的转换的算法提出一个严峻的问题。如果归约序列不能产生单一结点图而完成归约，那么这一方法必须或者报告失败，或者通过分离一个或多个结点修改此图，或者使用一个迭代方法来解决已归约图上的系统。在结构上一般基于归约流向图的方法局限于可约图。⊖

⊖ 我们通篇集中精力于迭代不动点算法。我们强调迭代数据流分析，因为它对于不同的问题和不同的图是健壮的。因为它是一个不动点定点算法，所以它还提供(方法上的)良好的连续性，这可以追溯到第2章中的子集合构造法。

相反，迭代算法可以在非可约图上正确地工作。 481 ~ 482

为了把不可约图转化成可约图，分析器可以分离一个或多个结点。本例的图的最简单的分离复制必要的块来为边<B_2，B_1>和<B_2，B_3>重设目标。这创建只带有一个入口的通过B_2的循环。在原本通过B_1或B_3进入循环的路径上，这些块作为这一循环的序言执行。从而创建一个可约图，如图9-21所示。B_1和B_3都有惟一的前驱。B_2有五个前驱，但却形成由B_2、B_1'和B_3'形成的复杂循环的惟一入口。因为B_1'和B_3'都有惟一前驱，所以这一循环有惟一的入口点而且它自身是可约的。结点分离产生一个可约图，但要以复制两个结点为代价。

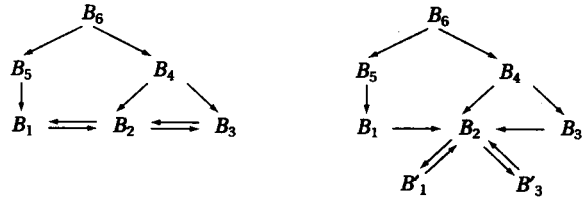


图9-21 结点分离

实践和研究表明，不可约图很少出现在全局数据流分析中。20世纪70年代结构化程序设计的出现使得程序员很少使用随意的控制转换，例如像在大多数程序设计语言中存在的goto语句。结构化循环结构如do、for、while和until等都不能产生不可约图。然而，把控制转出循环的转换（例如，C语言中的break语句）创建一个对于向后分析是不可约的CFG。（因为这样的循环有多个出口，逆向CFG也有多个入口。）同样地，不可约图可能更经常地出现在相互递归子例程的过程间分析中。例如，手工编码的递归下降分析器的调用图很可能有不可约子图。幸运的是，迭代分析器能够正确且高效地处理不可约图。

9.4.2 过程间分析

过程调用给全局数据流分析带来了不精确性。当编译器遇到一个过程调用，而且它没有关于被调用过程的详细信息时，它必须假设这一调用有最坏的效应。例如，被调用过程可能修改它能存取的任意值：全局变量、引用调用参数以及周围词法作用域的名字。 483

事实上，通常过程行为比最坏的假设所蕴涵的行为要好得多。为了避免每一个调用场所的信息丢失，编译器可以分析整个程序并记录概括了各个调用行为的信息。这样的分析通常被形式化为过程间数据流分析问题。这些数据流问题类似于相应的全局问题。取代控制流图，过程间问题通常使用调用图（call graph），或得自于调用图的一个图。调用图的结点表示过程，边表示过程调用。如果p多次调用q，那么每一次调用生成调用图中不同的边。

1. 控制流分析

在过程间分析中，编译器必须阐明的第一个问题是调用图的构建。对于最简单的情况，其中每一个过程调用调用一个已知的过程，这一过程的名字被指定为一个文字常量，此时这一问题是直截了当的。一旦程序员使用过程值变量，或者作为参数传递过程，那么这一问题就变得更加复杂。

如果源语言只允许程序员把过程作为参数传递给过程，那么可以通过构建一个近似的调用图并在这一近似的图中传播被传递过程的名字，同时在每次传播中增加相应的边，来解决这一问题。如果过程可以从调用中返回，即程序员可以编写过程值函数，那么就需要更加复杂的分析。最后，面向对象程序设计可能因名字解释依赖于类层次分析收集的运行时类型信息而进一步使调用图含混不清。

2. 概括问题

为了概括过程调用的副作用，编译器可以计算包含调用期间可能被修改或被引用的变量的集合。编译器可以使用这些“概括”集合来取代全局数据流分析中对调用场所的最坏假设。典型的过程间概括问题是可能修改（may modify）问题，它使用包含执行调用的可能被修改的所有变量名字的集合来注释每
484 一个调用场所。

可能修改问题是最简单的过程间数据流问题之一，但是它却展示可能出现的很多问题。这一问题由以下方程组描述：

$$\text{MOD}(p) = \text{LOCALMOD}(p) \cup \left(\bigcup_{e=(p,q)} unbind_e(\text{MOD}(q)) \right)$$

其中$e=(p, q)$是调用图中从p到q的一个边，而$unbind_e(q)$是一个函数，它根据对应于e的调用场所的绑定把q中的名字映射到p中的名字。LOCALMOD包含在p中被局部修改的变量。这个集合是在p中定义的名字集合与严格限制于p中的那些名字集合的差。为了计算被从p到q的特定调用可能修改的名字集合，编译器取MOD(q)的*unbind*，并将在通向q的入口处保存的所有别名添加到该结果中。类似公式可以找到可能引用（may reference）集合，它包含在调用中可能被引用的名字。

MOD信息可以显著改进全局分析的结果。例如，对调用场所所做的最坏情况假设促使全局常量传播可能丢失涉及参数和全局变量的值在内的信息。简单的MOD分析可以给出哪种信息可以安全保留。

3. 过程间常量传播

当全局变量和参数的已知常量值在调用图上传播时，过程间常量传播跟踪它们。从概念上看，这一问题类似于全局常量传播，但是它有额外一层复杂性。过程间常量传播必须跨越调用图的各边传播值，它必须理解可能出现在每一个调用的绑定（包括被返回值的值）。另外，它必须对经过一个过程p的本的过程体的值的传输进行建模：从p的入口，到p内的每一个调用场所，到p返回的任意值。因此，这一问题的过程内成份比概括问题更复杂，因为它可能是分析已经发现的信息的函数，而不是一个不变量集合。

若干编译器使用跳转函数（jump function）建模化过程内效应。编译器构造一个显式模型，模型把
485 过程入口（或从调用返回）处的已知常量映射到在其他调用和返回的常量值。数据流方程包含跳转函数的特殊调用；在传播阶段，分析器在必要时调用它们。跳转函数的实现多种多样，从在分析期间不发生变化的简单静态近似，通过小的参数化模型，直到执行完整的全局常量传播。在实践中，简单的模型似乎可以捕获大部分效应，至少从代码改进的角度看是这样的。

4. 指针分析

歧义性的内存引用干扰优化器改进代码的能力。歧义性的一个主要源头就是基于指针的值的使用。对于代码中的每一个指针，指针分析的目标是确定它可能引用哪些内存位置。没有这样的分析，编译器就必须假定每一个指针可以引用所有可寻址值，包括运行时在堆上分配的所有空间、有显式地址的所有变量以及作为引用调用参数而被传递的所有变量。

指针分析的一个共同形式是计算POINTSTO集合。这些集合包含这样的序对<*name*，*object list*>，其中*object list*是*name*可能存取的内存位置的列表。分析的目标是缩小*object list*。例如，编译器不能把歧义性值保存在寄存器中。然而，如果分析显示一个指针精确地引用一个对象，而且在代码的某个区域没有指向这一对象的其他指针，那么在那个区域，编译器可以把这个对象提升到一个寄存器中。

现今已有很多计算POINTSTO集合的技术。这些技术因它们在一个过程内的逼近行为的不同而不同，流非敏感（flow-insensitive）方法忽视过程内的控制流，而流敏感（flow-sensitive）方法则要考虑内部的控制流。某些技术通过把部分调用图的路径与每一个事实相关联来跟踪程序中的上下文。优化器可以

比较路径来确定两个事实是否可以同时成立。因为流敏感性和上下文敏感性都增加分析的代价，所以研究者设法量化来自于每一种技术的利益。在某些应用中，流非敏感、上下文非敏感的信息似乎就足够了。而对于其他的应用，更复杂的分析所带来的精确性是很有帮助的。

伴随引用调用形式参数的使用产生一定的歧义性。尽管可以使用更一般的POINTSTO公式来模型化这一情况，但是它还可以被作为一种复杂的概括信息来处理。这一方法为每一个过程计算一个*别名序对*（alias pair）集合。如果<x，y>∈ALIAS(p)，那么在某一条进入p的路径上，x和y可能引用相同的位置。 486

5. 另外一种选择：内联

作为过程间分析的另外一种选择，编译器可以执行内联替换（参见8.7.2节）。一个调用场所的内联展开把过程间问题转化成全局问题。这一方法很有吸引力，因为它给编译器设计者带来处理这一问题的最好的过程内方法。然而，内联也存在潜在的问题，包括代码增长、编译时间以及为大型程序生成好代码的困难。（特别是内联后的代码的名字空间可能从根本上比原来程序的名字空间大。这可能引发全局分析的空间问题以及全局寄存器分配的溢出问题。）

在使编译器的知识更精确的前提下，内联有彻底改进代码的潜力。例如，在面向对象代码中，内联通常可以减小给定调用场所的可能对象类型（类）集合的大小。这可以消除动态调度的需求，并使用简单的过程内控制流取代动态调度。这消除在实现面向对象语言中所面临的一个最大的负荷源头。这对有较小过程体的过程（方法）来说特别有优越性，它限制负效应的可能性。

6. 重编译

一旦编译器使用一个过程的知识来优化另一个过程，那么结果代码的正确性就依赖于这两个过程的状态。对一个过程的源代码的一系列改变可能会改变编译器曾用于证明其他过程内的优化的安全性时所用的事实；当这一情况发生时，编译器必须重编译使用了不合法优化的代码。因此，使用过程间分析或优化的系统必须密切注意这一问题。为了解决这一问题，编译器必须或者执行重编译分析来跟踪过程间数据流集合的变化并确定这些变化要求重编译的位置，或者采用一个结构来使编译器避免这样的问题。

重编译分析一般要求编译器跟踪过程间分析结果中的变化，并识别什么时候这些变化使优化变得不合法。对于编译器执行的每种过程间分析，它还执行发现重编译应该出现的时间的隐蔽分析。这增加分析量，但减少编译量。 487

另外一种方法是为每一次编译重新优化整个程序。称为*链接时优化器*（link-time optimizer）的系统把优化和代码生成的部分推迟到整个程序显现出来时进行。因为其结果只记录在可执行代码内，而这一可执行代码在下一次编译时被扔掉，所以这一策略绕开重编译的问题。这一方法增加可执行代码的每一个成份的编译和优化的次数，但是它避免跟踪重编译依赖关系的复杂性。

9.5 概括和展望

大多数优化把一般情况代码裁剪到编译代码中出现的特定上下文中。编译器裁剪代码的能力通常受限于它对程序的运行时行为的范围的知识。

数据流分析允许编译器在编译时对程序的运行时行为进行建模，并从这一模型中提取重要、特定的知识。已经提出很多数据流问题；本章展示了其中若干问题。其中的很多问题有导致高效分析的特征。特别地，可以用快速迭代框架表示的问题拥有使用简单迭代解算器的高效解。

静态单一赋值形式是把数据流信息和控制相关性信息都编码到程序的名字空间的一种中间形式。使用SSA形式通常简化分析和转换。很多现代转换依赖于代码的SSA形式。

本章注释

第一个数据流分析的荣誉通常被给予20世纪60年代早期在贝尔实验室的Vyssotsky[325]。在原来的FORTRAN编译器中的早期工作包括控制流图的构造法和为了估测执行频率在CFG上进行Markov风格的分析[25]。由Lois Haibt构建的这一分析器可以认为是一个数据流分析器。

使用tbl操作记录歧义性跳转线索已在Rice的巨大标量编译器项目中发挥了很好的作用。几乎所有的控制转换器都使用立即形式来编码；对于少数需要到寄存器的跳转的情况，前端生成一个简洁的tbl
488 集合。

Waterman描述了在CFG的构造法中等待槽可能带来的问题[95]。相关的问题也出现在过程间调用图的构造法中。Ryder[295]、Callahan等[60]以及Hall和Kennedy[170]都在FORTRAN程序的相对简单的框架下探讨了这些问题。Shivers在Scheme中向这些问题发起了进攻[305]。

迭代数据流分析已得到很好的研究。在关于这一课题的编译文献中，最有影响的论文是Kildall在1973年所写的论文[212]、Hecht和Ullman的协同工作[177]以及Kam和Ullman所写的两篇论文[199,200]。本章的处理方式依据Kam的工作。

本章集中精力讨论迭代数据流分析。人们提出了很多其他的解决数据流问题的算法[207]。Kennedy的路径列表算法预计算访问结点的顺序[205]。有兴趣的读者应该研究一下结构化技术，包括区间分析[16，17，56]、T_1-T_2分析[323，176]、Graham-Wegman算法[163，164]、平衡树、路径压缩[318，319]、图文法[208]以及分割变量技术[342]。

在文献中，支配具有很长的历史。Prosser在1959年引入了支配，但他没有给出计算支配者的算法[208]。Lowry和Medlock描述了用于他们的编译器中的这一算法[243]；这一算法的费时至少是$O(N^2)$，其中N是过程中语句的数量。有几位作者基于从CFG中消除结点开发出更快的算法[9，3，281]。Tarjan提出基于深度优先搜索和合并寻找的$O(N \log N+E)$算法[317]。Lengauer和Tarjan改进了这一算法时间边界[235]，其他人也一样[172，22，55]。支配者的数据流公式是从Aleen[13，16]那里得来的。迭代支配的快速数据结构来自于Cooper、Harvey和Kennedy的一篇尚未发表的论文。图9-10中的算法取自Ferrante、Ottenstein和Warren[138]。

SSA构造法基于Cytron等的工作[104]。而他们的工作是建筑在Shapiro Saint[303]、Reif[284，320]、Ferrante、Ottenstein和Warren[38]的工作之上的。9.3.3节中的算法构建半剪枝SSA形式[46]。重命名算法和重新构造可执行代码的算法的详细内容是Briggs等[47]给出的。由临界边引起的复杂性很早就受到了优化文献的重视[293，124，119，121，215]；我们不应对它们还出现在从SSA翻译为可执行代码的翻译中而感到惊讶。

IBM PL/I优化编译器是执行过程间数据流分析的一个最早的系统[313]。关于副作用问题[31，29，
489 96，97]和关于过程间常量传播问题[62，105，329]有很多文献。Burke和Torczon[57]公式化了一个分析，这一分析确定在一个较大程序中哪些块必须被重编译以反映程序的过程间信息的变化。指针分析本质上属于过程间分析；有很多文献对此做了描述[330，187，72，228，75，115，129，333，302，180，106，
490 181]。Ayers、Gottlieb和Schooler描述了一个实用系统，该系统分析和优化整个程序的某个子集[24]。



第10章
标量优化

10.1 概述

编译器中的代码优化通过分析和转换来试图改变编译器生成的代码质量。第9章详细讨论过的数据流分析能够使编译器发现转换的机会，并能证明运用转换的安全性。然而，分析只是转换的前奏：只有编译器通过重写代码才能改进代码的执行。

数据流分析是静态分析中典型问题的统一概念框架。许多问题可以看作是数据流框架问题。使用某种通用目的解算器可以解决程序中显现出来的这些问题实例。给出的结果可以是注释代码的某种形式的事实集合。有时候，可以把分析的信息直接编码成代码的IR形式，正如SSA形式所做的那样。

遗憾的是，优化不存在统一的结构框架。这一结构框架把特殊的分析与必要的重写相结合以达到理想的改进。优化消耗并产生编译器的IR；它们可以被看成是复杂的重写引擎。某些优化被作为详细的算法来描述；例如，基于支配者的值编号组建来自低层次细节的事实集合（参见8.5.2节）。而另外一些优化则通过高层次描述来说明；例如，全局冗余消除从数据流方程集开始操作（参见8.6节），而内联替换通常被描述为使用适当的实参替换形参并使用替换后的被调用过程的文本取代调用场所（参见8.7.2节）。用于描述和实现转换的技术多种多样。

现代编译器中的优化器一般被构造成一系列过滤器，每一个过滤器，即遍，以代码的IR形式为输入，作为它的输出每一个遍生成代码的IR形式的重写版本。优化器的这种结构因若干实际原因而得以发展。它把实现分解成较小的片段，避免在较大、集成的程序中所出现的某些复杂性问题。这种结构允许遍的独立实现和测试，从而简化开发、测试和维护。他允许编译器通过在各层次激活不同的遍来提供不同层次的优化。某些遍只执行一次；而其他遍可能在一个序列中执行多次。

于是，优化器设计中的一个重要问题就是选择要实现的一组遍以及运行这些遍的顺序。遍的选择决定在IR程序中发现哪样特定的低效性，以及为了缓解或消除这一低效性如何改进代码。执行顺序决定遍之间相互作用的方式。

例如，在适当的上下文（$r_2 \geqslant 0$且$r_3 = 4$）中，优化器可能把`mult r2, r3⇒r17`重写为`lshiftI r2, 2⇒r17`。这通过减小对寄存器的需求并使用更廉价的操作`lshiftI`取代成本高昂的操作`mult`来改进代码。在多数情况下，这是有益的。然而，如果下一个遍依赖于交换性来重新组织表达式，那么使用移位取代乘法则消除了一个机会（乘法是可交换的，而移位是不可交换的）。在某种程度上，它降低后面遍的效率，而且它可能损害整个代码的质量。推迟移位对乘法的取代可以避开这一问题，证明这一重写的安全性和高效性所需的上下文则可以在中间的遍中存留下来。

有时候，优化器应该多次重复一个遍。例如，消除死代码，或无用代码使编译器在几个方面受益。它缩小IR程序，所以后面的遍需要处理更少的代码。它消除某些定义和使用，所以它可以使数据流分析的结果更加明确。最后，它通过消除执行无用操作来改进结果代码，这才是它的真实目的。因为前两个效应，死代码消除通常在编译的早期运行。为了得到最后的效应，它应在编译后期运行。已知有些遍使代码无用，所以死代码消除还可能在这样的遍之后运行。因此，编译器通常在编译期间运行若干次死代码消除。

491
492

作为软件工程的优化

拥有独立的优化器可以简化编译器的设计和实现。这样的优化器简化前端；它可以生成通用目的的代码并忽视特殊情况。这样的优化器简化后端；它可以集中精力于把程序的IR版本映射到目标机器。没有优化器，前端和后端都必须参与寻找并开拓改进机会。

在遍结构式的优化器中，每个遍包含一个转换以及支持这一转换的分析。在原理上，优化器所执行的每一个任务都可以只实现一次。这提供了单一控制点，并使编译器设计者只实现一次复杂的功能，而不是多次实现复杂的功能。例如，从IR中删除一个操作可能很复杂。如果这个被删除的操作使得一个基本块为空（除了块结束分支和跳转之外），那么转换还应该删除这个块并适当地把这个块的前驱与其后继重新相连。把这一功能分离出来可以简化实现、理解和维护。

从软件工程的角度看，带有清晰的责任分离的遍结构是有意义的。它使每个遍只集中于一个任务。它提供清晰的责任分离，值编号忽视寄存器的压力，而寄存器分配器忽视公共子表达式。它使编译器设计者独立、全面地测试各遍，而且它简化错误隔离。

本章精选一组转换。在10.2节，我们围绕着转换的分类学将这些内容组织起来。10.3节给出在其他章节中没有得到很好陈述的转换类型的部分优化示例。高级话题一节简要讨论四个主题：组合优化以获
493 得更好的结果、操作符强度减弱、以速度以外的其他性能为目标的优化以及优化序列的选择。

10.2 转换分类

构建优化器的第一个障碍是概念上的障碍。关于优化的文献描述上百种改进IR程序的算法。编译器设计者必须选择这些转换的一个子集来使用。阅读原始论文对这一目的没有太大的帮助，因为多数作者都推荐使用他们自己的转换。

为了组织优化空间，我们使用一个简单的分类法来通过各转换对代码的效应对转换进行分类。分类必然是近似的。例如，有些转换有多种效应。在高层次上，我们把转换分成两个范畴：机器无关转换和机器相关转换。

机器无关转换（machine-independent transformation）是那些在很大程度上忽视目标机器细节的转换。在多数情况下，转换的效益实际上依赖于详细的机器相关论题，但是这类转换的实现却忽视这些细节。

例如，当局部值编号找到一个冗余的计算时，它使用一个引用取代这个冗余的计算。这消除一个计算，但它也许增加对寄存器的需求。如果增加需求迫使寄存器分配器把某个值移出到内存的话，那么内存操作的代价可能超过消除一个操作所带来的成本节约。然而，值编号却有意忽视这种效应，因为它不能精确地确定一个值是否一定要被移出。

机器相关转换（machine-dependent transformation）是那些明确考虑目标机器细节的转换。很多这样的转换属于代码生成领域，在这一领域中，编译器把代码的IR形式映射到目标机器上。然而，某些机器相关转换则属于优化器的范畴。（但是，大部分内容已超出本章的范畴。）其中的例子包括这样的转换：重排代码来改进缓冲的行为或设法揭示指令级并行。

尽管这两个范畴间的区分在某种程度上是人为的，但是这一区分长期以来一直被用作转换的第一级分类。

10.2.1 机器无关转换

494 事实上，编译器能够改进程序的机器无关方法很有限。我们考虑图10-1所示的五个效应。它们是：

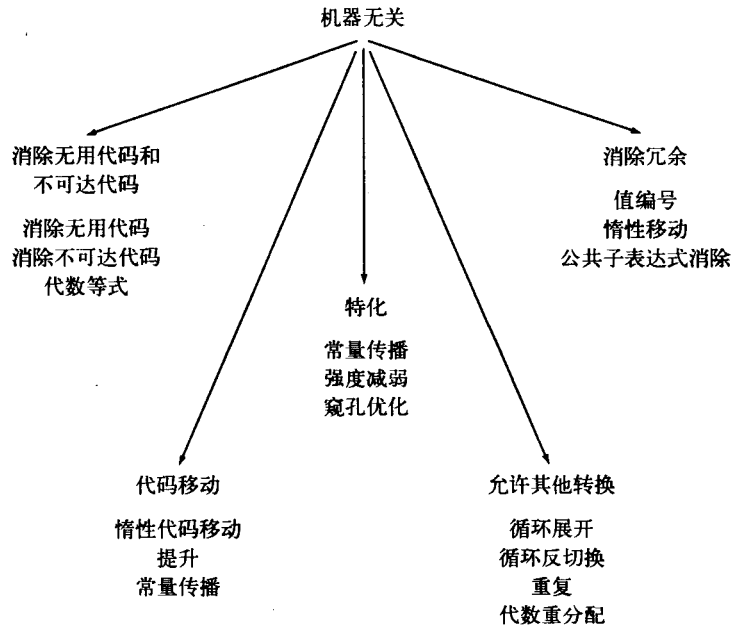


图10-1 机器相关转换

- *消除无用和不可达代码*：如果编译器能够证明一个操作或者是无用的或者是不可达的，那么它就可以消除这个操作。其方法包括无用代码消除和不可达代码消除（参见10.3.1节）、代数等式简化（8.3.2节中的局部值编号的一部分）和发现和消除某种不可达代码的稀疏条件常量传播（10.4.1节）。
- *把一个操作移到执行频率较低的地方*：如果编译器可以找到一个地方，在这里一个操作的执行频率较低且产生相同的答案，那么它可以把这个操作移到那里。其方法包括惰性代码移动（参见10.3.2节）和常量传播（参见9.2.4节、10.3.3节和10.4.1节），常量传播方法把运行时计算移到编译时进行。
- *特化计算*：如果编译器能够了解一个操作执行的特定上下文的话，那么它常常可以相对于这一上下文来特化操作的代码。其方法包括操作符强度减弱（参见10.4.2节）、常量传播（参见9.2.4节、10.3.3节和10.4.1节）以及窥孔优化（参见11.4.1节）。 495
- *激活其他转换*：如果编译器能够以某种方式重排序代码以便为其他转换揭示更多机会，那么它可以改进优化的整体效应。其方法包括内联替换（参见8.7.2节）、复制（参见8.7.1节和12.4.2节）以及代数重组（参见10.3.4节）。
- *消除冗余计算*：如果编译器能够证明一个计算是冗余的，那么它可以使用对前面已经计算的值的引用替换这个冗余计算。其方法包括局部值编号（参见8.3.2节）、超局部值编号（参见8.5.1节）、基于支配者的值编号（参见8.5.2节）以及全局公共子表达式消除（参见8.6节）。

我们已经看到了每个范畴的优化。10.3节将更加充分地给出转换分类中的机器无关转换，展示除冗余消除外每一个范畴的更多优化，而冗余消除已在第8章得到了相当深入的探讨。

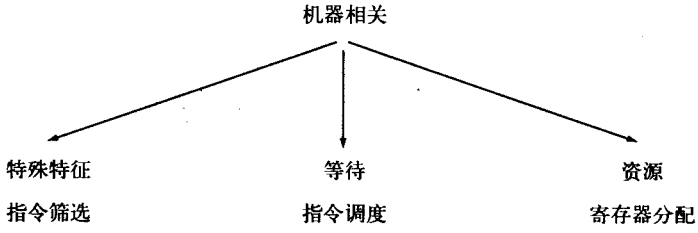
有些技术适合于多个范畴。惰性代码移动既可以达到代码移动的目的，又可以达到冗余消除的目的。常量传播可以达到某种形式的代码移动的目的，它把操作从运行时移到编译时，又可以达到特化的目的。它至少以一种形式（参见10.4.1节）区分和消除某种不可达代码。

10.2.2 机器相关转换

在机器相关转换中，编译器能够开发的效应受到更多的限制。它可以：

- *利用特殊的硬件特性*：通常，处理器设计者把他们相信对程序执行有帮助的特性包含到处理器中。这样的特性包括特化了的操作，如绕过缓冲层次的装入操作、告知分支预测硬件不跟踪其结果的分支操作，或咨询预取操作。如果编译器能够有效地利用这些特性，那么它们的确可以加速程序
496 的执行。识别利用这些特性的机会通常需要额外的工作。编译器设计者也许给优化器增加新的转换，或使用更复杂的指令筛选过程。其中的某些工作属于在第11章深入描述的指令筛选领域。
- *管理或隐藏等待时间*：在某些情况下，编译器可以以某种方式管理最终代码以便隐藏某些操作的等待时间。例如，内存操作可能有几十个乃至几百个周期的等待时间。如果目标机器或者支持预取操作或者支持无阻塞装入，那么编译器也许会找到一种调度在内存操作的使用之前发行这一内存操作，以此来隐藏等待时间。重新排序一个循环或一组嵌套循环的迭代顺序，或改变进入内存的值的填充都可以改进运行时缓冲位置并通过减少在缓冲中丢失的内存操作的数量来帮助管理等待时间。第12章详细讨论的指令调度处理其中的某些问题。然而，编译器设计者也许有必要增加直接处理这些问题的转换。
- *管理有限机器资源*：编译过程中的复杂性的另外一个源头来自于目标机器有有限资源，包括寄存器、功能单元、缓冲内存以及主内存的事实。编译器必须把所需的计算映射到机器提供的有限资源上，并且当这一计算需要的资源超过可用资源时，编译器必须重写代码以减少对资源的需求。例如，寄存器分配把计算值映射到目标机器的寄存器单元上；第13章详细描述这一问题。

图10-2给出本分类的机器相关部分。因为其效应的每一个例子都构成单独的一章，所以我们将它们从本章中省略。



497 图10-2 机器相关转换

10.3 优化示例

本节描述我们认定为机器无关转换的五个范畴中四个范畴中的额外优化。每一个优化都试图改进标量单处理器的性能。优化的选择是示意性的而不是详尽的。然而，每一个转换都是达到与机器无关转换范畴相关的特定效应的好例子。

10.3.1 消除无用和不可达代码

有时候，一个程序包含没有可视外部效应的计算。如果编译器能够确定一个给定操作具有这样的性质，那么它就可以从这一程序中消除这一操作。程序员一般不是有意识地创建这样的代码。然而，它却作为编译器优化的直接结果出现在大多数程序中，而且常常是在编译器前端的宏展开中出现。

两个不同的效应可以使一个操作符合消除的条件。这样的操作可以是*死的*（dead）或*无用的*（useless），即没有操作使用这一操作的结果，或其结果的所有使用都不能被外部探知。这一操作可能是

不可达的（unreachable），即不存在包含这一操作的有效控制流路径。如果一个计算属于其中任意一个范畴，那么它可以被消除。

消除无用或不可达代码生产更小的IR程序，这将导致更小的可执行程序和更快的编译。它还可能增加编译器改进代码的能力。例如，不可达代码可能存在于静态分析的结果中并阻碍某些转换的运用。在这种情况下，消除不可达块可以改变分析结果并允许进一步的转换（例如，参见10.4.1节的稀疏条件常量传播）。

冗余消除的某些形式也消除无用代码。例如，局部值编号运用代数等式来简化代码。例子包括x+0⇒x、y×1⇒y和max(zz)⇒z。这些表达式的任意一个都消除一个无用的操作，这时的无用操作指的是，消除这个操作对程序的外部可视行为不会带来任何不同。

本节的算法修改控制流图（CFG）。因此，它们把分支（cbr）与跳转（jump）区分开来。密切注意这一区分有助于读者理解本节中的算法。 498

1. 消除无用代码

消除无用代码的经典算法的操作方式类似于标记清理垃圾回收器的行为方式（参见6.7.3节）。同标记清理垃圾回收器一样，它们在代码上执行两遍。第一遍从消除所有标记域并标记“关键”操作开始。一个操作是关键的（critical），如果它为过程设置返回值，⊖它是一个输入/输出语句，或者它影响从这一过程外部可存取的存储位置中的值。关键操作的例子包括过程入口块和出口块以及对其他过程的调用的代码。接下来，算法向前跟踪关键操作的操作数到它们的定义，并把这些操作标记为有用操作。这一过程以一个简单的工作列表迭代方案持续下去，直到再没有操作可以标记为有用操作。第二遍遍历代码并消除所有没有被标记为有用的操作。

图10-3具体化这些想法。我们称为*Dead*的算法假设代码是SSA形式的。这简化上述过程，因为每一次使用引用一个定义。*Dead*由两个遍组成。第一个遍称为*MarkPass*，它发现有用操作集合。第二个遍称为*SweepPass*，它消除无用操作。*MarkPass*依赖于本节后面定义的反向支配边界。

```
Dead()
  MarkPass()
  SweepPass()

SweepPass()
  for each operation i
    if i is unmarked then
      if i is a branch then
        rewrite i with a jump
          to i's nearest marked
          postdominator
      if i is not a jump then
        delete i

MarkPass()
  WorkList ← ∅
  for each operation i
    clear i's mark
    if i is critical then
      mark operation i
      WorkList ← WorkList ∪ {i}
  while (WorkList ≠ ∅)
    remove i from WorkList
      (assume i is x ← y op z)
    if def(y) is not marked then
      mark def(y)
      WorkList ← WorkList ∪ {def(y)}
    if def(z) is not marked then
      mark def(z)
      WorkList ← WorkList ∪ {def(z)}
    for each block b ∈ RDF(block(i))
      let j be the branch that ends b
      if j is unmarked then
        mark j
        WorkList ← WorkList ∪ {j}
```

图10-3 无用代码消除

⊖ 这一情况可能以多种方式发生，包括给一个引用调用参数赋值、通过一个未知指针赋值或者一个真实的返回语句。

除了分支和跳转之外，操作的处理是直截了当的。标记阶段确定一个操作是否是有用的。而清除阶段消除没有被标记为有用的操作。

控制流操作的处理更为复杂。每一个跳转都被认为是有用的。只有当一个有用操作的执行依赖于一个分支的存在时，这一分支被认为是有用的。在标记阶段发现有用操作时，它还标记适当的分支为有用分支。为了把已标记操作映射到该操作认为有用的分支上，这一算法依赖于*控制相关*（control dependence）的概念。

控制相关的定义依赖于后支配。在一个CFG中，结点j后支配结点i，如果每一条从i到CFG出口结点的路径都通过j。⊖使用后支配，我们可以如下定义控制相关性：在一个CFG中，结点j与结点i控制相关，当且仅当：

1）存在一条从i到j的非空路径，使得j后支配这一路径上i之后的每一个结点。一旦执行在这一路径
499 上开始，那么这一执行一定通过j流到CFG的出口（根据后支配定义），而且

2）j结点不严格后支配i。存在另一个离开i的边，且控制可以沿一条路径到达不通向j的路径上的一个结点。一定存在一条开始于那条边，且不经过j就通向CFG的出口的路径。

换句话说，有两条或多条边离开块i。一条边通向j，而另外一条或多条边不通向j。因此，在块i的尾部分支处所做的决策可以决定j是否执行。如果j中有一个操作是有用的，那么i的尾部的分支也是有用的。

控制相关的概念可以通过j的反向支配边界RDF(j)来精确地刻画。*反向支配边界*（reverse dominance frontier）就是在反向CFG上计算的支配边界。当*MarkPass*把一个操作标记为有用时，它访问包含这个有用操作的块的反向支配边界中的每一个块，并把这个块的块结束分支标记为有用。当它标记这些分支时，
500 它把这些分支加到工作列表中。

对于所有未标记分支，*SweepPass*使用到包含已标记操作的第一个后支配者的跳转取代它。如果这个分支没有被标记，那么它的后继，即它的立即后支配者不包含有用操作。（否则，当这些操作被标记时，这一分支也将被标记。）如果这个立即后支配者不包含已标记操作，那么算法运用类似的讨论。为了找到最近的有用后支配者，算法可以向上遍历后支配者树，直到它找到包含有用操作的块。因为根据定义出口块是有用的，所以这一搜索一定停止。

*Dead*运行后，这一代码不包含无用计算。它可能包含空块，而这个空块可以通过下一个算法消除。

2. 消除无用控制流

优化可以改变程序的IR形式使得它有无用控制流。如果编译器包含以产生无用控制流为负作用的优化的话，那么编译器应该包含通过消除无用控制流来简化CFG的遍。这节给出一个处理这一任务的称为*Clean*的简单算法。

*Clean*在CFG上使用四个转换，如图10-4所示。这些转换以下面的顺序使用：

1）叠入冗余分支。如果*Clean*找到结束于分支的块，而且分支的两端都以相同块为目标，那么*Clean*应该使用到目标块的跳转取代这一分支。这一情况是作为其他简化的结果而出现的。例如，B_i也许有两个后继，每一个都带有一个到B_j的跳转。如果另一个转换消除这些块的所有计算，那么空块消除可能产生如图10-4第1部分所示的初始图。

2）消除空块。如果*Clean*找到一个只包含一个跳转的块，那么它就把这个块并入到它的后继中。当其他遍从B_i消除所有操作时，就会出现这种情况。考虑图10-4第2部分中的初始图。因为B_i只有一个后继B_j，这一转换为从进入B_i的边重定目标到B_j，并把B_i从B_j的前驱集合中删除。这简化此图。它也应该加快

⊖ 原文中这里是i。应该是j。——译者注

执行。在原图中，通过B_i到达B_j的路径需要两个控制流操作。在转换后的图中，这些到达B_j的路径使用一个操作。

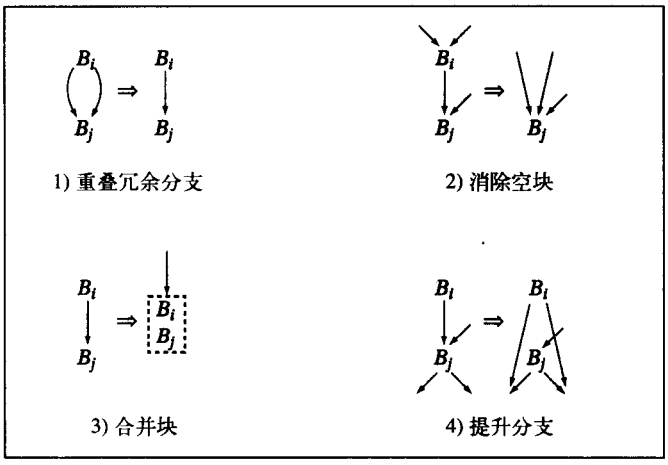


图10-4 *Clean*中使用的转换

3）合并块。如果*Clean*找到一个结束于到B_j的跳转的块B_i且B_j只有一个前驱，那么它可以把这两个块合并起来。这种情况以多种方式出现。另外一个转换可能消除其他进入B_j的边，或者B_i和B_j也许是叠入冗余分支的结果（如前所述）。无论哪种情况，这两个块都可以合并成一个块。从而消除B_i的尾部的跳转。

4）提升分支。如果*Clean*找到一个结束于到空块B_j的跳转的块B_i且B_j结束于一个分支，那么*Clean*可以使用B_j的拷贝取代结束于B_i的块结尾跳转。实际上，这把分支提升到B_i中。当其他遍消除B_j中的操作而只留下到一个分支的跳转时，就会出现这种情况。转换后的代码只使用一个分支就可实现相同的效应。这在CFG添加一个边。注意，B_i不能是空的，否则其他的空块消除将消除它。同样地，B_i不能是B_j的惟一前驱，否则其他*Clean*将合并这两个块。（提升后，B_j仍至少有一个前驱。）

实现这些转换需要某种簿记。某些修改是平凡的。为了在用ILOC和图式CFG表示的程序中叠入冗余分支，*Clean*使用一个跳转覆写这个块的尾部分支，并调整这些块的后继和前驱列表。而其他修改则更困难。合并两个块可能涉及为合并块分配空间、把操作拷贝到新块中、调整新块的前驱和后继列表及它在CFG中的邻居，以及丢弃原来的两个块。

*Clean*以一种系统的形式运用这四个转换。它后序遍历此图，这样B_i的后继在B_i之前被简化，除非这一后继位于对应于后序编号的后边上。对于这种情况，*Clean*将在访问后继之前访问前驱。在循环图中这是不可避免的。在前驱之前简化后继减少实现必须移动某些边的次数。

在某些情况下，可以使用多个这样的转换。各种情况的详细分析导致如图10-5所示的顺序。这一算法使用一系列*if*语句，而不是*if-then-else*语句，这使它在对一个块的单一访问中运用多个转换。

如果CFG包含后边，那么*Clean*的一遍可能会创建额外的机会，即优化沿着这个后边的未处理后继。而这些后继可以创建其他机会。出于这一原因，*Clean*迭代式地重复这一转换序列直到CFG停止变化。它必须在对*OnePass*的调用之间计算一个新的后序编号，因为每一遍都改变图。图10-5给出*Clean*的伪代码。

*Clean*不处理可能出现的所有情况。例如，它不可能依靠自己消除一个空循环。考虑图10-6a所示的CFG。假设块B_2是空的。*Clean*的任意转换都不能消除B_2。B_2尾部的分支不是冗余的。B_2不结束于一个跳

转，所以*Clean*不能将其与B_3合并。它的前驱结束于一个分支而不是跳转，所以*Clean*既不能合并B_2和B_1，也不能把B_2的分支叠入B_1中。

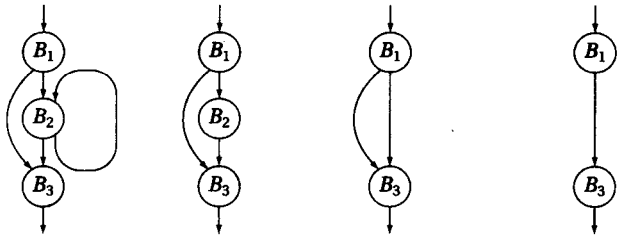
```
OnePass( )
  for each block i, in postorder
    if i ends in a conditional branch then
      if both targets are identical then
        replace the branch with a jump
    if i ends in a jump to j then
      if i is empty then
        replace transfers to i with transfers to j
      if j has only one predecessor then
        coalesce i and j
      if j is empty and ends in a conditional branch then
        overwrite i's jump with a copy of j's branch

Clean( )
  while the CFG keeps changing
    compute postorder
    OnePass( )
```

图10-5 *Clean*算法

然而，*Clean*和*Dead*间的合作可以消除空循环。*Dead*使用控制相关性来标记有用分支。如果B_1和B_3包含有用操作，而B_2不包含有用操作，那么*Dead*中的标记遍不会把结束B_2的分支标记为有用分支，因为$B_2 \notin \mathrm{RDF}(B_3)$。因为这一分支是无用的，所以计算这一分支条件的代码也是无用的。因此，*Dead*消除B_2中的所有操作且把结束B_2的分支转化成一个到它最近的有用后序支配者B_3的跳转。这消除原来的循环并产生如图10-6b所示的CFG。

以这种形式，*Clean*把B_2叠入B_1中，如图10-6c所示。这也使B_1尾部的分支成为冗余分支。*Clean*使用一个跳转重写这个分支，产生如图10-6d的CFG。在这一点，如果B_1是B_3惟一留下的前驱，那么*Clean*将把这两个块合并成一个单一块。



a) 源图 b) 使用*Dead*后的图 c) 移除B_2 d) 叠入分支

图10-6 消除空循环

这一合作比给*Clean*增加处理空循环的转换更简单且更高效。这样的转换可能识别从B_i到其自身的分支，对于空的B_i，使用到这一分支的其他目标的一个跳转重写B_i。问题在于确定什么时候B_i真正为空。如果B_i除了这个分支之外不包含其他操作，那么计算这一分支条件的代码必须位于循环之外。因此，只有自循环从不执行时这一转换才是安全的。推断自循环执行的次数需要比较的运行时值的信息，这一工作一般超出编译器的能力。如果这个块包含操作，但只是控制这一分支的操作，那么转换将需要识别模式匹配的情况。无论是哪一种情况，新的转换都将比*Clean*中的四个转换更加复杂。依赖于*Dead*和*Clean*

504

的合作可以以更简单、更模块化的形式达到适当的结果。

3. 消除不可达代码

有时候，CFG包含不可达代码。编译器应该发现不可达块并消除它们。一个块可能因为以下两个原因是不可达的：CFG中不存在通向这个块的路径，或者达到这个块的路径是不可执行的。例如，它被一个总是评估为假的条件所保护。

前者的情况比较容易处理。编译器可以在CFG上执行简单的标记清理式的可达性分析。从入口开始，编译器标记这一CFG中每一个可达结点。如果所有的分支和跳转都是非歧义性的，那么所有未标记的块都可以删除。对于歧义性的分支或跳转，编译器必须保存这样的分支或跳转可能达到的所有块。[⊖]这一分析简单且廉价。可以在其他目的的CFG遍历期间或在CFG本身的构造期间完成这一分析工作。

处理第二种情况更加困难。它需要编译器推断控制分支的表达式的值。10.4.1节给出寻找这样的不可达块的算法，通向这些块的路径是不可执行的。

10.3.2 代码移动

把一个计算移动到执行频率更低的位置应该可以减小运行程序的总操作计数。本节给出的第一个转换，*惰性代码移动*（lazy code motion）使用代码移动来加速执行。因为循环往往比包围它们的代码执行的次数更多，所以这一领域的大部分工作致力于把循环不变量表达式移出循环。惰性代码移动执行循环 505
不变量代码的移动。它把原本在可用表达式问题中公式化了的概念扩展到包含沿着某些路径冗余但不是在全部的路径冗余的操作上。它插入代码来使这些操作沿着所有路径都是冗余的，并消除这些最新的冗余表达式。

然而，有些编译器按其他标准进行优化。如果编译器关心可执行代码的大小，那么它可能考虑减少特定操作的拷贝数目的代码移动。本节给出的第二个转换，*提升*（hoisting），使用代码移动来减少指令的复制。它发现这样的情况：一个操作的插入使得相同操作的若干拷贝在不改变程序计算的值的前提下成为冗余。

1. 惰性代码移动

惰性代码移动（LCM）使用数据流分析来发现作为代码移动候选者的操作以及可以放置这些操作的位置。这一算法操作于程序的IR形式和它的CFG，而不在SSA形式上操作。这一算法由6个数据流方程集合和把结果解释为修改代码的指南的简单策略组成。

LCM把代码移动与冗余及部分冗余计算的消除结合起来。冗余性在第8章中做过介绍。一个计算在p处是部分冗余的，如果它出现在达到p的某些而不是全部路径上。图10-7给出表达式可能是部分冗余的两种形式。在第一个例子中，a←b × c出现在导向汇合点的一条路径上而不在另一条路径上。为使第二个计算冗余，LCM在另一条路径上插入a←b × c的评估。在第二个例子中，a←b × c沿着循环的后边是冗余的，但沿着进入这一循环的边不是冗余的。在循环的前面插入a←b × c的评估使得循环内的这一出现是冗余的。通过使循环不变量计算冗余并消除它，LCM把不变量计算移到循环之外。

为了实现这一点，优化器求解一系列数据流问题。它计算有关*可用性*（availability）的信息，这是在8.6节熟知的向前数据流问题，并计算有关*可前置性*（anticipability）的信息，这是作为向后数据流问题的相关概念。下一步使用这些分析的结果计算*最早放置*（earliest placement）；这使用一个集合注释每一条边，对于这一集合中的表达式，这个边是合法放置且在图中没有更早的合法放置。然后，算法寻 506

⊖ 如果源语言包含执行指针或标签上的算术的能力，那么每一个块都必须被保存。否则，编译器应该能够把被保存集合限制到其标签已被引用的那些块上。

找可能的*较晚放置*（later placement），即寻找把一个表达式从它的最早放置在CFG向前移动或在执行向后移动，且保持它产生与最早放置有相同效应的位置。最后，这一算法为每个边计算刻画沿着这条边插入的表达式的*插入集合*（insertion set），并为每一个结点计算包含从该块消除的表达式的*删除集合*（deletion set）。一个简单的后继遍解释插入集合和删除集合，并重写IR。

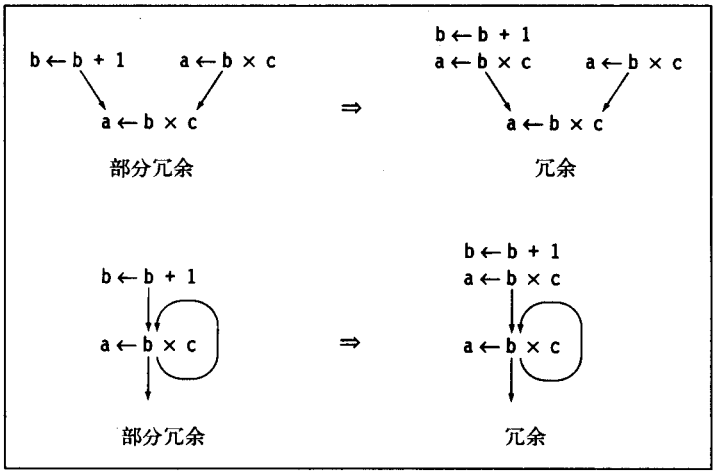


图 10-7

（1）背景　求解数据流方程的第一步是为每一个块计算局部谓词。LCM使用三种局部信息。

1）DEEXPR(*b*)。是块*b*中向下暴露的表达式*e*的集合。如果*e*∈DEEXPR(*b*)，那么在块*b*的尾部对*e*的评估所产生的值与在其原来位置的评估值相同。

2）UEEXPR(*b*)。是块*b*中向上暴露的表达式*e*的集合。如果*e*∈UEEXPR(*b*)，那么在块*b*的入口处对*e*的评估所产生的值与在其原来位置的评估值相同。

3）EXPRKILL(*b*)。是在块*b*中被杀死的表达式*e*的集合。如果*e*∈EXPRKILL(*b*)，那么*b*包含*e*的一个或
507 多个操作数的重定义。作为结果，在*b*的入口处对*e*的评估与在*b*的尾部对它的评估可能生成不同的值。

我们已在其他数据流问题中使用过这些集合。

LCM的方程依赖于若干个有关代码形态的隐含假设。它们假设文本相同的表达式总是定义相同的名字，这一假设提出名字的无限制集合，即每一个文本上不同的表达式有一个名字。（这是5.6.1节中描述的寄存器命名规则中的规则1。）因为r_k的每一个定义来自于表达式r_i+r_j且没有其他表达式定义r_k，所以优化器不需要寻找r_i+r_j的每一个定义，并把结果拷贝到一个临时位置以供后期使用。之后，它可以直接使用r_k。

LCM移动表达式，而不是移动赋值。文本上相同的表达式定义相同的虚拟寄存器的要求意味着程序变量是由寄存器到寄存器的拷贝操作设置的。图10-8中的代码具有这样的性质。通过把名字空间划分成变量和表达式，我们可以把LCM的定义域限制到表达式上。在这一例子中，变量是r_2、r_4和r_8中的每一个都是由一个拷贝操作定义的。所有其他名字，r_1、r_3、r_5、r_6、r_7、r_{20}和r_{21}代表表达式。下面的表给出本例中各块的局部信息：

	B_1	B_2	B_3
DEEXPR	$\{r_1, r_3, r_5\}$	$\{r_7, r_{20}, r_{21}\}$	…
UEEXPR	$\{r_1, r_3\}$	$\{r_6, r_{20}, r_{21}\}$	…
EXPRKILL	$\{r_5, r_6, r_7\}$	$\{r_5, r_6, r_7\}$	…

（2）可用表达式。LCM的第一步是计算可用表达式。

$$\text{AVAILIN}(n) = \bigcap_{m \in preds(n)} \text{AVAILOUT}(m), \quad n \neq n_0$$

$$\text{AVAILOUT}(m) = \text{DEEXPR}(m) \cup (\text{AVAILIN}(m) \cap \overline{\text{EXPRKILL}(m)})$$

以上方程的形式与8.6节中的方程形式略有不同。在这一形式中，这些方程为每一个块的入口和出口分别计算不同的集合AVAILIN和AVAILOUT。8.6节中的AVAIL集合与这里的AVAILIN集合相同。AVAILOUT集合表示一个块对其后继的AVAILIN集合的贡献。这一解算器必须初始化AVAILIN(n_0) 为空集，而设置AVAILIN和AVAILOUT集合为包含过程中所有表达式的集合。 508

对于图10-8中的例子，可用性方程产生下面的结果:

	B_1	B_2	B_3
AVAILIN	∅	{r_1, r_3}	{r_1, r_3}
AVAILOUT	{r_1, r_3, r_5}	{r_1, r_3, r_7, r_{20}, r_{21}}	…

在全局冗余消除中，x∈AVAILIN(b) 表示为沿着从n_0到b的每一条路径，x被计算，而且它的任意操作数都不在x被计算的点与b之间被重新定义。对理解LCM有帮助的另外一个观点是，x∈AVAILIN(b) 当且仅当编译器可以把x的评估放置在b的入口处，并得到由从n_0到b的任意控制流路径上的最新近评估产生的结果。就此而论，AVAILIN集合告知编译器，忽视x的任意使用，它可以把x的评估在CFG中向前移动的距离。

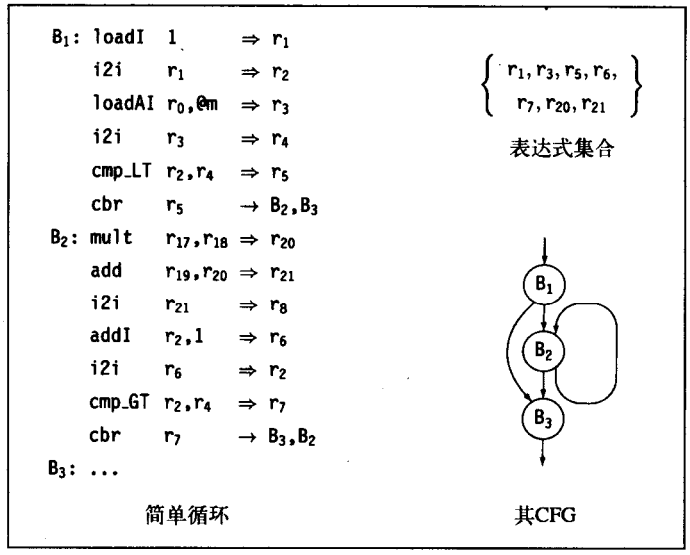


图10-8 惰性代码移动例子

（3）可前置表达式。可用性为LCM提供有关在CFG中把评估向前移动的信息。LCM还需要有关在CFG中把评估向后移动的信息。为了得到这一信息，LCM计算有关可前置表达式（anticipable expression）的信息，即在CFG中可以比它们的所在块更早评估的表达式。 509

UEEXPR集合局部地刻画这一概念。如果e∈UEEXPR(b)，那么b至少包含e的一个评估，而且这一评估使用在b的入口之前定义的操作数。因此，e∈UEEXPR(b) 蕴含着e在b的入口处的评估一定产生与b

中的e的第一次评估相同的值，所以编译器可以安全地把e的第一次评估移到块b的入口。

LCM的第二个数据流方程集合把可前置性概念推广到多个块之间。这是一个向后数据流问题。

$$\text{ANTOUT}(n) = \bigcap_{m \in succ(n)} \text{ANTIN}(m), \quad n \neq n_f$$

$$\text{ANTIN}(m) = \text{UEEXPR}(m) \cup (\text{ANTOUT}(m) \cap \overline{\text{EXPRKILL}(m)})$$

LCM必须初始化n_f的ANTOUT集合为空集，并保留ANTIN和ANTOUT集合为包含这一过程中所有表达式的集合。

ANTIN和ANTOUT为编译器提供把ANTOUT(b)中的表达式向后移动的相关信息。如果x∈ANTOUT(b)，那么编译器可以把x的评估放置在块b的尾部并产生与离开b的任意路径上的下一次评估相同的值。

对于我们的例子，可前置性方程产生下面的结果:

	B_1	B_2	B_3
ANTIN	$\{r_1, r_3\}$	$\{r_{20}, r_{21}\}$	$\emptyset$
ANTOUT	$\emptyset$	$\emptyset$	$\emptyset$

(4) 最早放置。给定编码在CFG中向前移动的相关信息的可用性，以及编码在CFG中向后移动的相关信息的可前置性，编译器能够为每一个表达式计算一个*最早放置*(earliest placement)。为了保持方程简单，把计算放置到CFG的边上而不是结点处要更容易些。这使得方程在不选择块的情况下计算放置。编译器可以推迟做如下决定: 把操作放置到边的源头的尾部、放置边的目的地的开始，还是在边的中间设置新块。(参见9.3.5节所讨论的临界边的概念。)

对于CFG中的边<i, j>，表达式e在EARLIEST(i, j)中当且仅当计算可以合法地移到边<i, j>且不能
510 移到CFG中任意更早的边上。EARLIEST的方程反映这一条件:

$$\text{EARLIEST}(i,j) = \text{ANTIN}(j) \cap \overline{\text{AVAILOUT}(i)} \cap (\text{EXPRKILL}(i) \cup \overline{\text{ANTOUT}(i)})$$

这些条件都有简单的解释。为使e在边<i, j>上合法且在CFG中不能进一步向上移动，下面的三个条件必须成立:

1) e∈ANTIN(j) 证明编译器可以把e移到j的头部。

2) e∉AVAILOUT(i) 证明在i的出口处没有e的早前计算是可用的。如果e∈AVAILOUT(i)，那么e在<i, j>上的计算将是冗余的。

3) e∈(EXPRKILL(i) ∪ $\overline{\text{ANTOUT}(i)}$) 证明$e$不能穿过$i$向上移动到更早的边。如果$e$∈EXPRKILL($i$)，那么$e$将在块$i$的头部产生一个不同的结果，所以它不能穿过$i$向上移动。如果$e$∈ $\overline{\text{ANTOUT}(i)}$ 的补集，那么e不能移到块i中。

因为LCM不能把表达式移动到比n_0更早的地方，所以对于$i = n_0$，LCM应该忽视最后一项(EXPRKILL(i) ∪ $\overline{\text{ANTOUT}(i)}$)。这一简化略微减小了计算EARLIEST的代价。

计算EARLIEST不需要迭代；它只依赖于早前的计算值。进而，EARLIEST被用于LATER集合的计算中。因为LATER计算需要迭代，为每个边预计算EARLIEST集合可能更快。编译器设计者也可以把EARLIEST(i, j)的定义的右端向前替换为方程中的下一个集合。

对于我们的例子，EARLIEST计算产生下面的集合:

	$<B_1, B_2>$	$<B_1, B_3>$	$<B_2, B_2>$	$<B_2, B_3>$
EARLIEST	$\{r_{20}, r_{21}\}$	$\emptyset$	$\emptyset$	$\emptyset$

（5）后放置。在达到相同效应的前提下，LCM中最后的数据流问题是确定一个最早放置是否可以在CFG中向前移动。

$$\text{LATERIN}(j) = \bigcap_{i \in pred(j)} \text{LATER}(i,j), \quad j \neq n_0$$

$$\text{LATER}(i,j) = \text{EARLIEST}(i,j) \cup \text{LATERIN}(i) \cap \overline{\text{UEEXPR}(i)}, \quad i \in pred(j)$$

511

这个解算器必须初始化LATERIN(n_0)为空集。

一个表达式e∈LATERIN(k)，当且仅当达到k的每一条路径包含一个边<p，q>使得e∈EARLIEST(p，q)，且从q到k的路径既不重新定义e的操作数也不包含e的评估（那个较早放置将是可前置的）。LATER方程中的EARLIEST项保证LATER(i，j)包含EARLIEST(i，j)。如果e可以从i开始向前移动（e∈LATERIN(i)）且i的入口的放置不前置i中的使用，那么这个方程的其余部分把e放到LATER(i，j)中。

一旦这一方程被解开，那么e∈LATERIN(i)意味着编译器可以穿过i向前移动e的评估而不损失任何利益，即在i中不存在较早评估是可前置的e的评估，而且e∈LATER(i，j)意味着编译器将把i中的e的评估向前移到j。

对于我们的例子，这些方程产生下面的集合：

	B_1	B_2	B_3
LATERIN	$\emptyset$	$\emptyset$	$\emptyset$

	$<B_1, B_2>$	$<B_1, B_3>$	$<B_2, B_2>$	$<B_2, B_3>$
LATER	$\{r_{20}, r_{21}\}$	$\emptyset$	$\emptyset$	$\emptyset$

（6）重写代码。执行惰性代码移动的最后一步是重写代码使得它利用得自于数据流计算的知识。为了简化这一过程，LCM计算两个额外的集合INSERT和DELETE。

对于每一条边，INSERT集合指定LCM应该插入这条边上的计算。

$$\text{INSERT}(i,j) = \text{LATER}(i,j) \cap \overline{\text{LATERIN}(j)}$$

如果i只有一个后继，那么LCM可以在i的尾部插入这些计算。如果j只有一个前驱，那么LCM可以在j的入口插入这些计算。如果上述两个条件都不成立，那么边<i，j>是一个临界边，而且编译器应该在这个边的中间插入一个块来分离<i，j>，以便保存INSERT(i，j)中指定的计算。

512

对每一个块，DELETE集合指定LCM应该从这一块中删除的那些计算。

$$\text{DELETE}(i) = \text{UEEXPR}(i) \cap \overline{\text{LATERIN}(i)}, \quad i \neq n_0$$

当然，DELETE(n_0)是空的，因为没有块在它之前。如果e∈DELETE(i)，那么在所有插入完成之后，e在i中的第一次计算是冗余的。在i中有向上暴露使用的e的任意评估序列也可以被消除，即当e的操作数在i的开始和这一评估间不被定义时，e的任意评估序列也可以被消除。因为e的所有评估定义相同的名字，所以编译器无需重写对已删除评估的后期引用。这些引用将直接引用那些LCM已证明产生相同值的e的

更早评估。

对于我们的例子，INSERT和DELETE集合很简单。

	<B_1, B_2>	<B_1, B_3>	<B_2, B_2>	<B_2, B_3>
INSERT	{r_{20}, r_{21}}	∅	∅	∅

	B_1	B_2	B_3
DELETE	∅	{r_{20}, r_{21}}	∅

INSERT和DELETE集合重写代码的解释如图10-9所示。LCM从B_2删除定义r_{20}、r_{21}的表达式，并把这些表达式插入从B_1到B_2的边上。

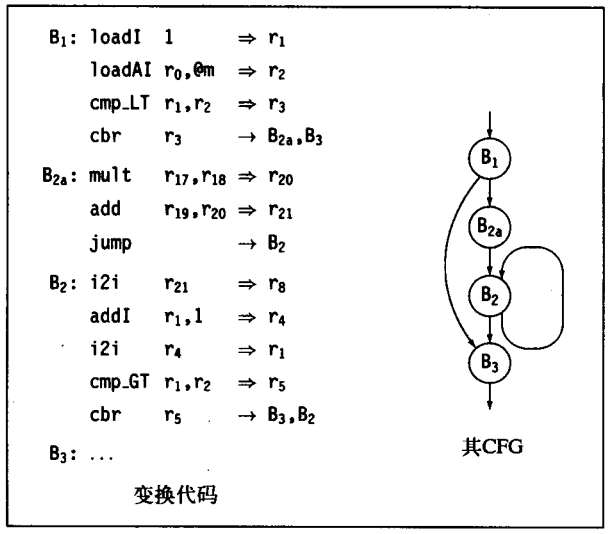


图10-9 惰性代码移动后的例子

因为B_1有两个后继而B_2有两个前驱，所以<B_1, B_2>是一个临界边。因此，LCM必须分离这个边，创建一个新块B_{2a}来保存插入的r_{20}的计算。分离边<B_1, B_2>给代码增加一个额外的jump。后继的代码生成几乎肯定地把B_{2a}中的jump实现为一个落下，从而消除与此相关的任何代价。

注意，LCM把B_2中定义的r_8的拷贝保留下来。LCM移动表达式，而不是赋值。（回想一下，r_8是一个变量，而不是一个表达式。）如果这一拷贝是不必要的，那么寄存器分配器将发现这一事实并将其合
513 并到其他操作中。

2. 提升

编译器也可以使用代码移动来缩小编译代码的大小。9.2.4节引入了忙碌表达式的概念。编译器可以使用VERYBUSY集合的计算结果来执行*代码提升*（code hoisting）。

这一想法很简单。对于某个块b，一个表达式$e \in$VERYBUSY(b)，当且仅当离开b的每一条路径都评估e，而且b的尾部的e的评估与沿着每一条离开b的路径上的e的下一个评估产生相同的值。（也就是说，e的任意操作数都不在b的尾部与沿着离开b的每一条路径的下一次评估之间被重新定义。）为了缩小代码，编译器可以在b的尾部插入e的评估，并使用一个引用来取代在离开b每一条路径上的e的第一次出现。

为了直接替换这些表达式，编译器将需要定位这些表达式。它可以插入e，然后求解另一个数据流

问题，证明从b到e的某一个评估的路径是e-clear的。另外，它也可以遍历离开b的每一条路径来寻找e被定义的第一个块，这可以通过在这个块的UEEXPR集合中的查找来实现。这些方法似乎都比较复杂。 514

一个更简单的方法是让编译器访问每一个块b，并对于每一个表达式$e \in$VERYBUSY(b) 在b的尾部插入e的一个评估。如果编译器使用一致的命名规则，就如在LCM讨论中所提到的那样，那么每一个评估将定义一个适当的名字。随后的LCM的运用或冗余消除将消除这一新的冗余表达式。

10.3.3 特化

在大多数编译器中，IR程序的形态是在对代码的任意详细分析之前由前端决定的。必然地，这将产生在运行程序可能遇到的所有上下文中都行得通的一般代码。然而，使用分析编译器通常能够充分缩小代码所处的上下文。这就为编译器利用代码执行的上下文信息来特化操作序列创造了机会。

作为一个例子，考虑常量传播。常量传播试图发现一个操作的参数所取的特殊值。对于诸如x←y × z这样的操作，如果编译器发现y总有值4，且z是非负的，那么它可以使用通常代价较小的移位操作取代这个乘法。如果它还发现z有值17，那么它可以使用68的立即装入取代这个操作。这些操作形成一个层次。乘法是普遍的；它适合于y和z的任意值（尽管它也许对它们的某些值会产生异常）。移位不如乘法普遍：它产生正确的结果，当且仅当y有值4且z是非负的。当然，如果y或z是零，那么x也是零。装入立即最不普遍：只有当操作数具有y × z的值是已知的这样的性质时它才有效。

特化的其他例子包括*窥孔优化*（peephole optimization）和*尾递归消除*（tail-recursion elimination）。窥孔优化在代码上滑动一个小“窗口”（窥孔）并在这个窗口内寻找简化。它源于在代码生成后执行某种最后局部优化的一个有效方法（参见11.4.1节）。

尾递归消除识别何时一个过程中执行的最终操作是一个自递归调用。尾递归消除使用一个到这一过程的第一个指令的跳转取代这个调用，这样可以复用活动记录并且回避完整过程链接约定的开销。在使用递归遍历数据结构或执行迭代计算的程序中出现这一重要的情况。下一节给出特化的详细例子：一个基于SSA的常量传播算法。 515

常量传播

使用SSA形式，我们可以使用比9.2.4节的方程直观得多的方法重新形式化常量传播。称之为*稀疏简单常量传播*（sparse simple constant propagation，SSCP）的这一算法如图10-10所示。

```
Worklist ← ∅
for each SSA name n
  initialize Value(n) by rules
    specified in the text
  if Value(n) ≠ ⊤ then
    Worklist ← Worklist ∪ {n}

while (Worklist ≠ ∅)
  remove some n from Worklist
  for each operation o that uses n
    let m be the SSA name that o defines
    if Value(m) ≠ ⊥ then
      t ← result of modelling m using meet rule
      if t ≠ Value(m) then
        Value(m) ← t
        Worklist ← Worklist ∪ {m}
```

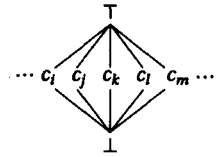
$\top \wedge x = x \quad \forall x$

$c_i \wedge c_j = c_i \quad \text{if } c_i = c_j$

$c_i \wedge c_j = \bot \quad \text{if } c_i \neq c_j,\ c_i \neq \top, c_j \neq \top$

$\bot \wedge x = \bot \quad \forall x$

交运算规则



格

图10-10 稀疏简单常量传播

SSCP算法由一个初始化阶段和一个传播阶段组成。初始化阶段在SSA名字上进行迭代。对于每个SSA名字n，初始化阶段根据一组简单的规则检查定义n的操作并设置$Value(n)$。如果n是由一个ϕ函数定义的，那么SSCP设置$Value(n)$为⊤。如果n的值是已知的常量c，那么SSCP设置$Value(n)$为c。如果n的值是不可知的，例如，它是由从外部媒介读取的一个值所定义的，那么SSCP设置$Value(n)$为⊥。最后，如果n的值是未知的，那么SSCP设置$Value(n)$为⊤。如果$Value(n)$不是⊤，算法把n加到工作列表中。

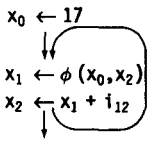
传播阶段是直观的。它从工作列表中移除一个SSA名字n。算法检查使用n的每一个操作o，其中o定义某个SSA名字m。如果$Value(m)$已经达到⊥，那么不再需要进一步的评估。否则，传播阶段通过在o的操作数的格点值上解释这一操作来建模o的评估。如果模型化的结果比$Value(m)$低，那么它相应地降低$Value(m)$并把m加到工作列表中。当工作列表为空时，这一算法终止。

在格点值上解释操作需要小心。对于ϕ函数，这一解释的结果就是所有ϕ函数的参数的格点值的交，即使一个或多个参数有值⊤也如此。对于其他种类的操作，编译器必须运用特定操作符的知识。如果任意操作数有格点值⊤，那么评估返回⊤。如果所有操作数都没有格点值⊤，那么这一模型应该产生适当的值。对于操作$x \leftarrow y \times z$，且$Value(y)=3$、$Value(z)=17$，这一模型应该产生值51。如果$Value(y)=\perp$，这一模型应该对$Value(z)=0$产生0，对任意其他格点值产生⊥。SSCP需要对IR中的每一个求值操作做类似的解释。

（1）复杂度　SSCP的传播阶段是一个典型的不动点方案。可以通过如图10-10所示的用于表示值的格的下降链的长度得到它的终止性和复杂度。与任意SSA名字相关的$Value$可以有三个初始值：⊤、某个不同于⊤和⊥的常量c_i以及⊥。传播阶段只能降低它的值。对于给定的SSA名字，这至多发生两次：从⊤到c_i再到⊥。只有当SSA名字的值发生变化时，SSCP才把这个名字加到工作列表中，所以每一个SSA名字至多在工作列表中出现两次。当一个操作的操作数从这一工作列表中被移除时，SSCP评估这一
516 操作。因此，评估的总次数至多是程序中使用的次数的两倍。

（2）覆盖面　SSCP发现9.2.4节中的数据流框架发现的所有常量。因为SSCP把未知值初始化为⊤，而不是⊥，所以它可以把某些值传播到图中的环，即CFG中的循环。开始于值⊤而不是⊥的算法通常被称为乐观（optimistic）算法。这一术语的直观印象是到⊤的初始化允许这一算法把信息传播到环区域，乐观地假设沿着后边的值将认可这一初始传播。被称之为悲观（pessimistic）的到⊥的初始化不允许有这种可能性。

为了明白这一点，考虑图10-11中的SSA片段。如果这一算法悲观地将x_1和x_2初始化为⊥，那么它将不会把值17传播到循环。当它评估x_1的ϕ函数时，它计算$17 \wedge \perp$得到⊥。使用设置到⊥的x_1，x_2也被设置到⊥，即使i_{12}有诸如0这样的已知值。



时序	SSCP格值					
	悲　观			乐　观		
	x_0	x_1	x_2	x_0	x_1	x_2
0	17	⊥	⊥	17	⊤	⊤
1	17	⊥	⊥	17	17	$17 \wedge i_{12}$

图10-11　乐观常量传播示例

另一方面，如果这一算法乐观地把未知值初始化为⊤，那么它可以把x_0的这个值传播到循环中。当它计算x_1的值时，它评估$17 \wedge \top$，并把结果17赋给x_1。因为x_1的值已发生变化，所以算法把x_1放置到工

作列表中。然后，这一算法重新评估x_2的定义。例如，如果i_{12}有值0，那么算法给x_2赋值17并把x_2加到工作列表中。当它重新评估ϕ函数时，它计算$17 \wedge 17$，并证明x_1是17。 517

考虑当i_{12}有值2时将发生什么。这时，当SSCP评估x_1+i_{12}时，它给x_2赋值19。现在，x_1得到值$17 \wedge 19$，即$\perp$。这传播回到x_2，产生与悲观算法相同的最终结果。

10.3.4 激活其他转换

通常，一个优化器包括主要目的是为其他转换创建或揭示机会的遍。在某些情况下，一个转换改变代码的形态使它更易于优化。而在另一些情况下，这个转换在代码中创建使其他转换安全的特定条件成立的地点。通过直接创建必要的代码形态，这些激活转换降低优化器对输入代码形态的敏感性。

本书的其他章节描述了若干激活转换。块复制（8.7.1节）复制各个块来消除分支并创建一种状态，在这种状态下，编译器可以得到一个块从它的CFG前驱继承而来的上下文的更精确信息。例如，12.4.2节描述块复制是如何改进指令调度的结果的。内联替换（8.7.2节）合并两个过程来消除过程调用的负荷，并为特化创建更大的上下文。本节给出三个简单的激活转换：*循环展开*（loop unrolling）、*循环反切换*（loop unswitching）和*重命名*（renaming）。 518

1. 循环展开

循环展开是最古老的激活转换之一。为了展开一个循环，编译器复制循环体，并调整控制被执行迭代的数目的逻辑。考虑如图10-12a所示的简单的循环。

如果编译器使用这一循环体的四个拷贝取代这个循环体，以4的因子展开循环，那么它就可以使用四分之一的比较和分支执行相同效应的工作。如果编译器知道n的值，比如说100，而且以整除n的因子展开循环，那么展开的循环有如图10-12b所示的简单形式。

```
do i = 1 to n by 1
   a(i) = a(i) + b(i)
   end
```

a) 源循环

```
do i = 1 to 100 by 4
   a(i)   = a(i) + b(i)
   a(i+1) = a(i+1) + b(i+1)
   a(i+2) = a(i+2) + b(i+2)
   a(i+3) = a(i+3) + b(i+3)
   end
```

b) 由4展开循环，n=100

```
i = 1
do while (i+3 < n)
   a(i)   = a(i) + b(i)
   a(i+1) = a(i+1) + b(i+1)
   a(i+2) = a(i+2) + b(i+2)
   a(i+3) = a(i+3) + b(i+3)
   i = i + 4
   end
do while (i < n)
   a(i) = a(i) + b(i)
   i = i + 1
   end
```

c) 由4展开循环，任意n

```
i = 1
if (mod(n,2) > 0) then
   a(i) = a(i) + b(i)
   i = i + 1
if (mod(n,4) > 1) then
   a(i)   = a(i) + b(i)
   a(i+1) = a(i+1) + b(i+1)
   i = i + 2
do j = i to n by 4
   a(j)   = a(j) + b(j)
   a(j+1) = a(j+1) + b(j+1)
   a(j+2) = a(j+2) + b(j+2)
   a(j+3) = a(j+3) + b(j+3)
   end
```

d) 由4展开循环，任意n

图10-12 展开一个简单循环 519

当循环边界未知时，展开需要某种额外的逻辑来支持满足mod(n，4)≠0的n的值。图10-12c中的循环版本给出处理这些情况的简单方法。图10-12d的循环版本使用稍微多一些的代码达到相同的结果。以较复杂的循环体为代价，图10-12d的循环版本允许对处理中间情况的两个迭代做某些改进。

如8.2.1节所示，LINPACK库中dmxpy的摘要使用图10-12d的方案。完整代码包含一次、两次、四次、八次和十六次迭代的情况。在每一种情况中，展开后的循环包含另一个循环，所以内循环的工作量证明大范围的外循环展开是适宜的。

循环展开减小程序所执行的总操作数。它也增加程序的大小。（如果循环体对指令缓冲而言变得太大时，那么结果的缓冲失误可能超出降低的循环负荷所带来的任何效益。）然而，循环展开的主要理由是为其他优化创建更好的代码形态。

展开拥有两个为其他转换创建机会的关键性效应。它增加循环体内的操作数量。相对于短循环和长分支等待时间，这可以产生更好的调度。特别是它可以给调度器带来更多可以同时执行的独立操作。它也可能给调度器带来足够多的操作来填充分支等待槽。它可以允许调度器把连续内存存取移到一起。这改进局部性并为一次可以处理更多数据的内存操作的使用提供可能性。

作为最后的注释，如果循环在一次迭代中计算在后面迭代中使用的值，这称为*循环进位数据相关*（loop-carried data dependence），而且如果需要拷贝操作为后期使用保存这个值，那么循环展开可以消除这些拷贝操作。对于拷贝的多种周期，由各个周期长度的最小公倍数所做的循环展开将消除所有的拷贝。⊖

2. 循环反切换

循环反切换把作为循环不变量的控制流操作提升到循环的外面。如果一个if-then-else结构中的谓词是循环不变量，那么编译器可以通过把if-then-else拉到循环外面，并在新的if-then-else的每个语句体内生成这个循环的裁剪拷贝来重写这一循环。图10-13给出一个短循环的这种转换。

```
do i = 1 to n                          if (x > y) then
   if (x > y)                             do i = 1 to n
      then a(i) = b(i) * x                   a(i) = b(i) * x
      else a(i) = b(i) * y                else
                                             do i = 1 to n
                                                a(i) = b(i) * y
```

源循环　　反切换版本

图10-13　反切换一个短循环

反切换是一种激活转换；它使得编译器以其他方法难以实现的方式裁剪循环体。反切换之后，剩余
520 的循环包含更少的控制流。它们执行更少的分支和其他支持那些分支的操作。这可以导致更好的调度、更好的寄存器分配和更快的执行。如果原来的循环包含if-then-else内的循环不变量代码，那么LCM将不能把它移到这一循环的外部。反切换之后，LCM很容易找到并消除这样的冗余。

反切换也有简单直接的改进程序的效应，它把控制循环不变量条件的分支逻辑移到循环的外面。把控制流移出循环是很困难的。基于数据流分析的各种技术，如LCM，在移动这样的结构时会遇到问题，因为这一转换改变分析所依赖的CFG。基于值编号的技术可以识别控制if-then-else结构的谓词是相等的某些情况，但是这些技术却不设法从循环中消除这一结构。

⊖ 译者的理解是，如果第一种拷贝的周期为4，即每4次迭代需要一次这样的拷贝，而第二种拷贝的周期为6，那么以12为因子的循环展开可以完全消除所有这两种拷贝。——译者注

3. 重命名

本章所给出的大多数转换都涉及对程序中的操作进行重写或重排序的工作。良好的代码形态可以揭示优化的机会。同样地，良好的名字集合也可以揭示优化的机会。

例如，在局部值编号中，我们看到了下面的例子:

$a \leftarrow x + y$	$a_0 \leftarrow x_0 + y_0$
$b \leftarrow x + y$	$b_0 \leftarrow x_0 + y_0$
$a \leftarrow 17$	$a_1 \leftarrow 17$
$c \leftarrow x + y$	$c_0 \leftarrow x_0 + y_0$
源代码	SSA形式

521

在原来的代码中，局部值编号能够识别出x+y的所有三个计算都产生相同的结果。然而，它不能替换x+y的最后出现，因为中间对a的赋值破坏了它识别出的x+y的拷贝。

把这一代码转变成SSA形式产生如图右边所示的名字空间。使用SSA名字空间，x_0+y_0在最终的操作时仍然可用，所以局部值编号可以使用一个对a_0的引用替换这一评估。(另一个方法是修改局部值编号算法使得它认可b为x+y的另一个拷贝。重命名是更简单、更一般的解决方案。)

一般地，名字的精心使用可以揭示出更多优化机会，它使更多的事实对分析可视并避免伴随存储复用的副作用。对于基于数据流的优化，如LCM，分析依赖于词法相等: 冗余操作必须有相同的操作，而且它们的操作数必须有相同的名字。⊖把诸如得自于值编号的某种值相等编码到名字空间的方案可以向LCM揭示出更多的冗余并让它消除这些冗余。

在指令调度中，名字创建限制调度器重组操作的能力的相关性。如果一个名字的复用反映了值的真实流向，那么这些相关性是正确性的重要组成部分。如果一个名字的复用出现的原因是寄存器分配器出于高效性而把两个不同的值放置在相同的寄存器内，那么这些相关性可能不必要地限制调度，从而在某些情况下，导致低效率的代码。

重命名是一个微妙的问题。SSA构造法根据特定的规则重命名程序中的所有值。结果名字空间在优化中是有帮助的。这些命名规则已在惰性代码移动及其补充说明中加以描述。命名的影响(参见第5章)简化很多转换的实现，它创建值的使用名字与计算这个值的操作的文本形式之间的一一映射。编译器设计者长久以来已认识到在控制流图中移动操作(而且，从事实上改变CFG本身)可能是有益的。同样，他们也应该认识到编译器无需受程序员或从源语言到特定IR的转换所引入的名字空间的制约。对手头的任务做适当的重命名可以改进很多优化的有效性。

522

10.3.5 冗余消除

第8章以冗余消除作为揭示优化的作用域的原始例子。它描述了局部值编号、超局部值编号和基于支配者的值编号，所有这些都是基于自底向上、面向细节并使用散列法来识别必须相等的值的方法。它指出使用可用表达式来执行全局公共子表达式消除可以作为全局分析和转换的一种方法，并指出全局优化一般需要把分析与转换分离开来。

在本章的前面部分，LCM是作为代码移动的例子出现的。它把以可用表达式为先导的数据流方法扩展到将代码移动和冗余消除统一起来的框架上。提升消除相同的操作以减小代码的大小；但不减少程序所执行的操作数量。

⊖ LCM不使用SSA名字，因为那些名字使文本相等含混。再创建文本相等带来额外的代价；因此，基于SSA的LCM比10.3.2节中所示的LCM版本运行得更慢。

10.4 高级话题

本章选取的大多数例子是用于说明编译器可以用来加速可执行代码的特殊效应的。有时候，一起执行两种优化可以产生分别使用这两个优化的任意组合都不可能得到的结果。下一小节给出这样一个例子：把常量传播与不可达代码消除相结合。10.4.2节给出另一个、更复杂的特化例子：使用线性函数测试替换的操作强度减弱。我们所给出的*OSR*算法比前面的算法简单，因为这一算法依赖于SSA形式的性质。10.4.3节简要描述编译器所考虑的某些其他目标功能。最后，10.4.4节讨论在选择优化器的转换集合的运用顺序中所产生的某些问题。

10.4.1 优化组合

有时候，同时形式化两个不同的优化并对它们连带求解可以产生由分别运行各优化的任意组合都得
523 不到的结果。作为一个例子，考虑10.3.3节描述的稀疏简单常量传播算法。它为程序的SSA形式中的每一个操作的结果指定一个格值。当它停止时，它使用格值⊤、⊥或一个常量对每一个定义赋标签。一个定义有值⊤，仅当它依赖于未初始化的变量，表明被分析代码中的一个逻辑问题。

稀疏简单常量传播给条件分支所使用的操作数指定一个格值。如果这个值是⊥，那么两个分支目标都是可达的。如果这个值即不是⊤也不是⊥，那么这个操作数必须有一个已知值，并且编译器可以使用到其两个目标中的一个的跳转重写分支，从而简化CFG。因为这从CFG中消除一个边，所以它可能消除进入标签为被消除分支目标的块中的最后的边，从而使那个块不可达。原理上，常量传播可以忽视不可达块的任意效应。稀疏简单常量传播没有利用这一知识的机制。

SSA图

在某些算法中，把代码的SSA形式看成是一个图可以简化讨论或实现。强度减弱算法把代码的SSA形式解释成一个图。

在SSA形式中，每一个名字有惟一的定义，因此一个名字指定代码中计算这个名字的值的特定操作。一个名字的每个使用出现在特定的操作中，所以这一使用可以解释为从这一使用到它的定义的链。因此，把名字映射到定义这些名字的操作上的简单查找表，创建一个从每个使用到相应定义的链。从定义到使用该定义的操作的映射要稍微复杂一些。然而，这一映射可以在SSA构造法的重名命阶段容易地构造出来。

在画SSA图时，我们以从使用到这个使用的相应定义为边。这表明SSA名字所蕴含的关系。编译器需要沿两个方向遍历这些边。强度减弱主要从使用向着定义移动。稀疏条件常量传播算法可以认为是在SSA图上从定义向着使用移动。编译器设计者能够容易地加入允许在两个方向上遍历所需的数据结构。

我们可以扩展稀疏简单常量传播算法，使其利用这些观察结果。其结果算法就是如图10-14a所示的
524 *稀疏条件常量传播*（sparse conditional constant propagation，sccp）算法。图10-14b勾勒出SCCP必须处理的各种操作的模型化规则。

为了避免包含不可达操作的效应，SCCP处理初始化和传播的方式不同于稀疏简单常量算法。第一，SCCP使用可达性标签标记每个块。最初，设置每个块的标签为不可达。第二，SCCP必须初始化每一个SSA名字的格值为⊤。早前的算法在初始化期间处理已知常量值的赋值（例如，对于$x_i \leftarrow 17$，它可以初始化*Value*(x_i)为17）。而SCCP不改变格值，直到它证明这一赋值是可达的。第三，这一算法需要两个

工作列表来进行传播：一个是CFG中的块的工作列表，另一个是SSA图中边的工作列表。最初，这一CFG的工作列表只包含入口结点n_0，而SSA工作列表是空的。迭代传播一直运行，直到它耗尽这两个工作列表（这是一个典型、复杂的不动点计算）。

```
SSAWorkList ← ∅
CFGWorkList ← {n0}
for each block b
  mark b as unreachable
  for each operation o in b
    Value(o) ← ⊤
while (CFGWorkList ≠ ∅ or
       SSAWorkList ≠ ∅)
  if CFGWorkList ≠ ∅ then
    remove some b from CFGWorkList
    mark b as reachable
    simultaneously model all the
      φ-functions in b
    model, in order, each
      operation o in b
  if SSAWorkList ≠ ∅ then
    remove some s = ⟨u,v⟩ from SSAWorkList
    let o be the operation that uses v
    if Value(o) ≠ ⊥ then
      t ← result of modelling o
      if t ≠ Value(o) then
        Value(o) ← t
        for each SSA edge e = ⟨o,x⟩
          if block(x) is reachable
            then add e to SSAWorkList
```

a) 算法

```
x ← c:                          (对常量c)
 Value(x) ← c

x ← φ(y,z):
 Value(x) ← Value(y) ∧ Value(z)

x ← y op z:
 if Value(y) ≠ ⊥ & Value(z) ≠ ⊥
   then Value(x) ← interpretation
        of Value(y) op Value(z)

cbr r1 → l1,l2:
 if r1 = ⊥ or r1 = TRUE
   and block l1 is marked as unreachable
   then add block l1 to CFGWorkList
 if r1 = ⊥ or r1 = FALSE
   and block l2 is marked as unreachable
   then add block l2 to CFGWorkList

jump → l1:
 if block l1 is marked as unreachable
   then add block l1 to CFGWorkList
```

b) 模型化规则

图10-14 稀疏条件常量传播

525

为了处理CFG工作列表中的块b，SCCP首先标记块b为可达。下一步，它模型化b中所有ϕ函数的效应，在重新定义所有ϕ函数的输出的格值之前，小心读取所有相关参数的格值。（回想一下，一个块中的所有ϕ函数是同时执行的。）下一步，SCCP以线性遍历这个块来模型化b中每一个操作的执行。图10-14b给出一组典型的模型化规则。这些评估可能改变由这些操作定义的SSA名字的格值。

SCCP改变一个名字的格值时，它必须检查SSA图中连结定义已改变值的操作与后继使用的边。对于每一个这样的边s=<u，v>，如果包含v的块是可达的，那么SCCP把这个边s加到SSA工作列表中。一旦SCCP发现这个块是可达的，那么在一个不可达块中的使用就会被评估。

b中的最后的操作一定是跳转或分支。如果这个操作是一个到标记为不可达的块的跳转，那么SCCP就把这个块加到CFG工作列表中。如果这是一个分支，那么SCCP就检查控制条件表达式的格值。这表明一个或两个分支目标是可达的。如果这选择了还没有标记为可达的目标，那么SCCP就把这个目标加到CFG的工作列表中。

传播步骤之后，我们需要一个最后遍来替换操作数带有⊥以外的*Value*标签的操作。它能够特化很多这样的操作。它还应该使用适当的跳转操作重写具有已知结果的分支。（这将导致后期遍消除它的代码并简化控制流，如10.3.1节所述。）只有在SCCP知道每一个定义的最后的格值时，它才能够重写代码，因为常量值*Value*标签在后面可能变成⊥。

1. 评估和重写操作的微妙问题

在模型化个别操作中有一些微妙的问题。例如，如果算法遇到操作数为⊤和⊥的乘法操作，那么它可能推断这个操作产生⊥。然而，这样做是不成熟的。后继分析可能把⊥降低到零，使得这个乘法产生值零。如果SCCP使用规则⊤ × ⊥→⊥，那么它将引入非单调行为的可能性：乘法值可能遵从序列⊤、
526 ⊥、0。这可能增加算法的运行时间，因为时间界限依赖于经过一个浅格的单调进程。同样重要的是，它可能不正确地引发其他值到达⊥。

为了解决这一问题，SCCP应该使用如下所示包括⊥在内的三个乘法规则：⊤ × ⊥→⊤，对于$\alpha \neq \top$且$\alpha \neq 0$，$\alpha \times \bot \rightarrow \bot$以及$0 \times \bot \rightarrow 0$。这些规则也适用于一个参数值可以完全确定结果的其他操作。其他例子包括多于一个字长的移位、与零的逻辑与以及与所有位为1的逻辑或。

某些重写具有无法预料的结果。例如，对于非负的x，使用一个移位取代4 × x将把一个可交换操作替换成一个不可交换的操作。如果编译器随后设法使用交换性重组表达式，那么这早前的重写将使编译器丧失一个机会。这种相互干涉可能对代码的质量造成明显的影响。为了选择编译器把4 × x改变成移位的时机，编译器设计者必须考虑运用优化的顺序。

2. 有效性

SCCP可以找到稀疏简单常量算法无法找到的常量。同样地，SCCP可以发现10.3.1节中所述的所有算法组合都无法发现的不可达代码。它的能力源自可达性分析与格值传播的结合。它可以消除某些CFG边，因为格值足以确定一个分支取哪一条路径。它可以忽视不可达操作（通过初始化这些定义为⊤）引发的SSA边，因为如果这一块的标记变为可达，那么这些操作将得到评估。SCCP的威力来自于这些想法间的相互作用，即常量传播和可达性间的相互作用。

如果可达性在确定格值的过程中没有起到作用，那么通过执行常量传播（并把取值常量的分支重写成跳转），后面跟随不可达代码消除，就可以实现相同的效应。如果常量传播在可达性中没有起到作用，那么通过另外一种顺序，即不可达代码消除后跟随常量传播，可以得到相同的效应。SCCP寻找简化的能力超过了这些结合，原因正是由于这两个优化相互依赖。

10.4.2 强度减弱

527 强度减弱是这样的转换，它使用计算相同结果的一系列低成本（“弱”）操作取代一系列重复进行的高成本（“强”）操作。典型的情况是把基于循环索引的整数乘法替换成等价的加法。这种特殊情况经常出现于循环中的数组和结构地址的展开。图10-15的左侧给出为下面的简单循环生成的ILOC代码：

```
sum ← 0
for i ← 1 to 100
  sum ← sum + a(i)
```

这一代码是半剪枝SSA形式；纯粹是局部的值（r_1、r_2、r_3和r_4）即没有下标，也没有ϕ函数。注意，对a(i)的引用是如何展开成四个操作，subI、multI、计算(i-1) × 4+@a的addI和定义r_4的load的。

对于每一次迭代，这一操作序列重新把a(i)的地址计算为循环索引变量i的函数。考虑r_1、r_{11}、r_2和
528 r_3的取值序列。

r_{11}: {1, 2, 3, ..., 100}
r_1: {0, 1, 2, ..., 99}
r_2: {0, 4, 8, ..., 396}
r_3: {@a, @a+4, @a+8, ..., @a+396}

r_1、r_2和r_3中的值只是为了计算load操作的地址。如果程序根据前一个值计算r_3的每一个值，那么

它可以消除定义r_1和r_2的操作。当然，r_3将需要一次初始化和一次更新。这将使它成为一个非局部名字，所以它也将在l_1和l_2处需要一个ϕ函数。

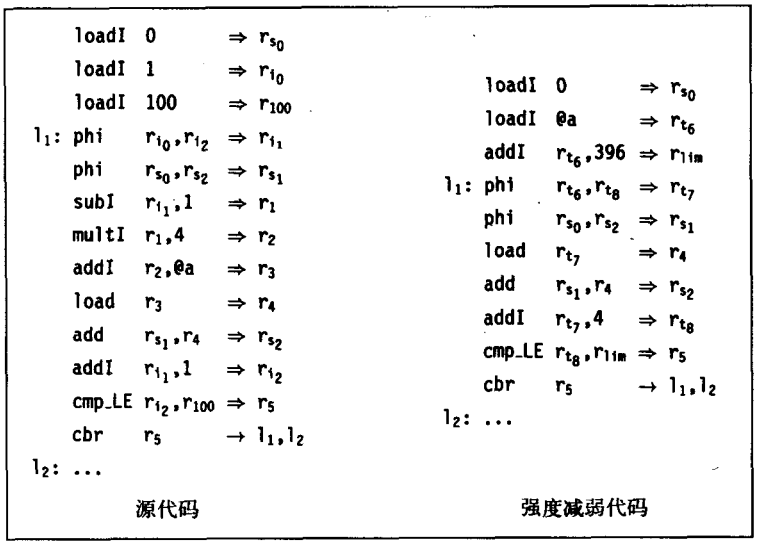


图10-15 强度减弱示例

图10-15的右侧给出经强度减弱、线性函数测试替换和死代码消除后的代码。它把原本在r_3中的那些值直接计算到r_{t_7}中，并在load操作中使用r_{t_7}。在原来代码中使用r_1的循环尾部测试已被修改成使用r_{t_8}。这使得r_1、r_2、r_3、r_{i_0}、r_{i_1}和r_{i_2}的所有计算都成为死计算。这些计算被消去来产生最终代码。现在，忽略ϕ函数，这一循环正好包含5个操作，而原来的代码包含8个操作。(在从SSA形式翻译回可执行代码时，ϕ函数变成寄存器分配器通常可以消除的拷贝操作。)

如果multI操作比addI操作的成本更高，那么上述的成本节省就会更大。历史上看，乘法的高代价已经证明了强度减弱的正当性。然而，即使乘法和加法有相同的代价，循环的强度减弱形式也许仍然受欢迎，因为它为后来的转换和代码生成创建更好的代码形态。特别地，如果目标机器具有自动递增寻址模式，那么循环中的addI操作可以叠入内存操作。对于原来的乘法，这一取舍是不存在的。

本节的其余部分给出强度减弱的一个简单算法*OSR*，其后给出线性函数测试替换的一个方案，此方案与*OSR*一起可以将循环尾部测试从操作于r_{i_2}替换成使用r_{t_8}。*OSR*操作于代码的SSA图上；图10-16给出此例中ILOC的SSA形式与SSA图之间的关系。

(1) 背景 强度减弱寻找如下上下文，在这样的上下文中，一个诸如乘法的操作在循环的内部执行，且这个操作的操作数是① 在那个循环内不发生变化的一个值，称为区域常量 (region constant)，以及② 随着迭代系统地发生变化的一个值，称为归纳变量 (induction variable)。当强度减弱发现这一情况时，它创建一个新的归纳变量，它以更高效的方式计算与原来的乘法相同的值序列。对于乘法操作的操作数 529
形式的限定保证这一新的归纳变量可以使用加法而不是乘法来计算。

我们称能够以这种方式减弱的操作为候选操作 (candidate operation)。为了简化*OSR*的表示，我们只考虑有下面形式之一的候选操作:

$$x \leftarrow c \times i \qquad x \leftarrow i \times c \qquad x \leftarrow i \pm c \qquad x \leftarrow c + i$$

其中，c是一个区域常量，而i是一个归纳变量。寻找和减弱候选操作的关键是高效地区分区域常量与归纳变量。一个操作是一个候选，当且仅当它有上述形式之一，包括对操作数的限制。

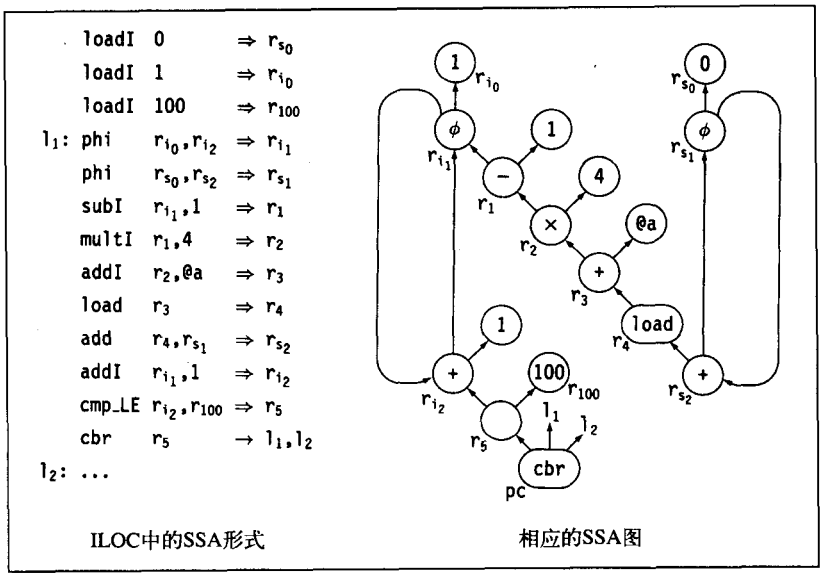


图10-16 ILOC中的SSA与SSA图的关系

530 区域常量可以是文字常量，例如10，也可以是循环不变量的值，即一个在循环内部不被修改的值。使用SSA形式的代码，编译器可以通过检查它的参数的惟一定义位置来确定它是否是循环不变量：它的定义必须支配定义这一归纳变量的循环入口。*OSR*可以在常量时间内检查这两个条件。在强度减弱之前执行惰性代码移动和常量传播可以揭示出更多可以作为区域常量的值。

直观地，归纳变量的值在循环内形成一个等差级数。对于这一算法的目的，我们可以使用更特殊、更局限的定义：归纳变量是满足下面条件的SSA图中的强连通成份（SCC）：在这样的SCC中，更新这一归纳变量的值的每一个操作是① 一个归纳变量加上一个区域常量，或者② 一个归纳变量减去一个区域常量，或者③ 一个ϕ函数，或者④ 从另一归纳变量的寄存器到寄存器拷贝。尽管这一定义不如常规定义那么一般，但是它完全能够让*OSR*算法寻找并减弱候选操作。为了确定归纳变量，*OSR*在SSA图中寻找SCC，并在其上进行迭代以确定SCC内的每一个操作是否是这四个类型中的一个。

因为*OSR*在SSA图中定义归纳变量及相对于CFG中的一个循环的区域常量，所以确定一个值是否是相对于这一包含特定归纳变量的循环的常量的测试是复杂的。考虑形如x←i × c的操作*o*，其中i是一个归纳变量。为使*o*成为强度减弱的候选者，*c*必须是相对于*i*发生变化的最外循环的区域常量。为了测试c是否有这一性质，*OSR*必须将SSA图中的i的SCC重新与CFG中的一个循环相关联。

*OSR*使用定义i的SCC中的最低逆向后序编号寻找SSA图中的结点。它把这一结点看成是这一SCC的头部，并把这一事实记录在SCC的每一个结点的标题字段。（SSA图中不成为归纳变量的一部分的任意结点设置它的标题字段为*null*。）在SSA形式中，这一归纳变量的头部是这一变量发生变化的最外循环的开始处的ϕ函数。在一个形如x←i × c的操作中，其中i是一个归纳变量，如果包含c的定义的CFG块支配着包含i的头部的块，那么c是一个区域常量。这一条件保证c在i发生变化的最外层循环内是一个不变量。为了执行这一测试，SSA结构必须产生从每一个SSA结点到得到它的CFG块的映射。

标题字段在确定一个操作能否被强度减弱的过程中起着重要的作用。当*OSR*遇到一个操作x←y × z时，它可以通过在SSA图中跟踪到y的定义的边，并检查它的标题字段来确定y是否是一个归纳变量。*null*标
531 题字段表明y不是一个归纳变量。如果y和z都有*null*标题字段，那么操作不能被强度减弱。

如果y或者z中有一个有非*null*标题字段，那么*OSR*使用那个标题字段来确定另外一个操作数是否是

一个区域常量。假设y的标题字段不是*null*。为了寻找y发生变化的最外循环的入口的CFG块，*OSR*以y的标题为索引查询SSA到CFG的映射。如果包含z的定义的CFG块支配y的头部块，那么z是相对于归纳变量y的区域常量。

（2）算法　为了执行强度减弱，*OSR*必须检查每一个操作并确认这一操作的一个操作数是一个归纳变量，而另一个操作数是一个区域常量。如果这一操作满足这些标准，那么*OSR*可以通过创建计算所需值的一个新的归纳变量，并使用这个新归纳变量的寄存器到寄存器拷贝替换这一操作来减弱这一操作。（*OSR*应该回避创建完全相同的归纳变量。）

基于前面的讨论，我们知道*OSR*可以通过寻找SSA图中的SCC来识别出归纳变量。OSR还可以通过检查值的定义来发现区域常量。如果这个定义得自于一个立即操作，或它的CFG块支配归纳变量的头部的CFG块，那么这个值是一个区域常量。关键是如何把这些想法结合起来形成一个高效的算法。

*OSR*使用Tarjan的强连通区域探测器来驱动整个过程。如图10-17所示，*OSR*取一个SSA图为它的参数，并对它反复使用强连通区域探测器*DFS*。（当*DFS*已访问*G*中的每一个结点时，这一过程停止。）

```
OSR(G)
  nextNum ← 0
  while there is an unvisited n ∈ G
    DFS(n)

DFS(n)
  n.Num ← nextNum++
  n.Visited ← true
  n.Low ← n.Num
  push(n)
  for each operand o of n
    if o.Visited = false then
      DFS(o)
      n.Low ← min(n.Low,o.Low)
    if o.Num < n.Num and
       o is on the stack
      then n.Low ← min(n.Low,o.Num)
  if n.Low = n.Num then
    SCC ← ∅
    until x = n do
      x ← pop( )
      SCC ← SCC ∪ { x }
    Process(SCC)

Process(N)
  if N has only one member n
    then if n is a candidate operation
           then Replace(n,iv,rc)
           else n.Header ← null
    else ClassifyIV(N)

ClassifyIV(N)
  IsIV ← true
  for each node n ∈ N
    if n is not a valid update for
       an induction variable
      then IsIV ← false
  if IsIV then
    header ← n ∈ N with the
             lowest RPO number
    for each node n ∈ N
      n.Header ← header
  else
    for each node n ∈ N
      if n is a candidate operation
        then Replace(n,iv,rc)
        else n.Header ← null
```

图10-17　操作符强度减弱算法

*DFS*在SSA图上执行深度优先搜索。对应于访问结点的顺序，它给每一个结点指定一个编号。它把每一个结点压入一个栈，并使用从这个结点的子结点可以达到的一个结点上的最低深度优先编号对这一结点赋标签。当*DFS*从处理子结点的工作返回时，如果从*n*可达的最低结点具有*n*的编号，那么*n*是一个SCC的头部。*DFS*从栈中弹出结点直到它达到*n*；所有这些结点都是SCC的成员。

*DFS*以简化*OSR*其余部分的顺序从栈中消除SCC。当一个SCC被从栈中弹出并被传送到*Process*时，*DFS*已经访问了它在SSA图中的所有子结点。如果我们把SSA图的边解释为从使用到定义，如图10-16中的SSA图所示，那么只有当候选操作的操作数已被传送到*Process*后，算法才会遇到这一候选操作。当*Process*遇到一个强度减弱的候选操作时，它的操作数已被分类。因此，*Process*可以在深度优先搜索中检查操作、识别出候选操作，并调用*Replace*来以强度减弱的形式重写这些候选操作。

*DFS*把每一个SCC传送给*Process*。如果这个SCC是由有以下候选形式（x←c×i、x←i×c、x←i+-c或x←c+i）的一个结点*n*组成的，那么*Process*把*n*连带它的归纳变量*iv*和它的区域常量*rc*，传送给*Replace*。*Replace*如下一节所述的那样重写这一代码。⊖如果这个SCC包含多个结点，那么*Process*把这个SCC传送给*ClassifyIV*以确定它是否是一个归纳变量。

对于SCC中的每一个结点，*ClassifyIV*参照归纳变量的有效更新集合检查这个结点。如果所有更新都是有效的，那么SCC是一个归纳变量，而且对于这一SCC中的每一个结点，*Process*把它的标题字段设置成包含带有最低逆向后序编号的SCC结点。如果SCC不是一个归纳变量，那么*ClassifyIV*重新访问SCC中的每一个结点来检测它是否是一个候选操作，根据检测结果或者把它传送到*Replace*，或者设置它的头部以显示它不是一个归纳变量。

532 ~ 533

（3）重写代码　*OSR*的剩余部分实现重写步骤。*Process*和*ClassifyIV*都调用*Replace*来执行重写。图10-18给出*Replace*的代码以及它的支持函数*Reduce*和*Apply*。

```
Replace(n, iv, rc)
  result ← Reduce(n.op, iv, rc)
  replace n with a copy from result
  n.header ← iv.header

Reduce(op,iv,rc)
  result ← Lookup(op, iv, rc)
  if result is "not found" then
    result ← NewName()
    Insert(op, iv, rc,result)
    newDef ← Clone(iv, result)
    newDef.header ← iv.header
    for each operand o of newDef
      if o.header = iv.header
        then rewrite o with
             Reduce(op, o, rc)
      else if op is × or
           newDef.op is φ
        then replace o with
             Apply(op, o, rc)
  return result

Apply(op, o1, o2)
  result ← Lookup(op, o1, o2)
  if result is "not found" then
    if o1 is an induction variable
       and o2 is a region constant
      then result ← Reduce(op, o1, o2)
    else if o2 is an induction variable
       and o1 is a region constant
      then result ← Reduce(op, o2, o1)
    else
      result ← NewName()
      Insert(op, o1, o2,result)
      Find block b dominated by the
        definitions of o1 and o2
      Create "op o1, o2 ⇒ result"
        at the end of b and set its
        header to null
  return result
```

图10-18　重写步骤的算法

*Replace*取三个参数，一个是SSA图的结点*n*，一个是归纳变量*iv*，一个是区域常量*rc*。后两个参数是*n*的操作数。*Replace*调用*Reduce*来重写*n*表示的操作。下一步，*Replace*使用它所产生的结果的拷贝操作替换*n*。它设置*n*的标题字段并返回。

*Reduce*和*Apply*做其中的大部分工作。它们使用一个散列表来避开插入重复的操作。因为*OSR*作用于SSA名字，所以一个全局散列表就足够了。在第一次调用*DFS*之前，这个散列表在*OSR*中被初始化。*Insert*把条目加到散列表中；*Lookup*查询这个表。

*Reduce*的计划很简单。它取一个操作码和两个操作数，它或者创建一个新的归纳变量以替换计算，或者返回前面为操作码和操作数的相同组合所创建的归纳变量的名字。它查询散列表以避免重复工作。

534

如果希望的归纳变量不在散列表中，那么它用一个两步过程创建这个归纳变量。首先，它调用*Clone*来拷贝*iv*的定义，*iv*是将被减弱的操作的归纳变量。下一步，它在这一新定义的操作数上重复这一工作。

⊖ 把*n*识别为一个候选的过程必然要把它的一个操作数识别为一个归纳变量*iv*，并把另一个操作数识别为一个区域常量。

这些操作数属于两个范畴。如果操作数是在SCC内定义的，那么它是*iv*的一部分，所以*Reduce*在这个操作数上递归运行。这通过在原来的归纳变量*iv*的SCC的周围复制这个操作数的行为而形成这个新的归纳变量。定义在SCC外部的操作数必定或者是*iv*的初始值或者是递增*iv*的值。而初始值一定是一个SCC外部的ϕ函数的参数；对于每个这样的参数，*Reduce*调用*Apply*。如果这一个候选操作不是一个乘法，那么*Reduce*可以不管归纳变量的增量。对于乘法，*Reduce*必须计算新的增量为旧增量和原来的区域常量*rc*的积。它调用*Apply*来生成这一计算。

*Apply*取一个操作代码和两个操作数，定位代码中的适当点并插入那个操作。它返回那个操作的结果的新SSA名字。其中一些细节需要进一步的解释。如果这一新操作本身是一个候选操作，那么*Apply*调用*Reduce*来处理这一新操作。否则，*Apply*得到一个新名字，插入这个操作，并返回结果。（如果*o1*和*o2*都是常量，*Apply*可以评估这个操作并插入一个立即装入。）它使用支配信息为这一新操作定位一个适当的块。直观地，这一新操作必须进入由定义这一操作的操作数的各块所支配的一个块。如果一个操作数是一个常量，那么*Apply*可以在定义另外一个操作数的块中复制这一常量。否则，两个操作数都必须有支配头部块的定义，而且一个操作数必须支配另一个操作数。*Apply*可以在后者的定义之后立即插入这个操作。

（4）回到我们的例子　考虑当*OSR*遇到图10-16中的例子时将发生什么情况。假设*OSR*开始于一个标签为r_{s_2}的结点，并假设它在访问右子结点之前访问左子结点。它沿着定义r_4、r_3、r_2、r_1和r_{i_1}的操作链向下递归处理。在r_{i_1}处，它在r_{i_2}上递归处理，然后在r_{i_0}上递归处理。它找到包含文字常量1的两个单结点SCC。二者都不是候选操作，所以*Process*通过设置它们的标题为*null*来标识它们为非归纳变量。

*DFS*发现的第一个非平凡SCC包含r_{i_1}和r_{i_2}。所有操作都是一个归纳变量的有效更新，所以*ClassifyIV*标识每一个结点为一个归纳变量，设置它的标题字段指向在SCC中带有最低深度优先编号的结点，即r_{i_1}的结点。

现在，*DFS*返回r_1的结点。它的左子结点是一个归纳变量而它的右子结点是一个区域常量，所以它调用*Reduce*来创建一个归纳变量。在这种情况下，r_1是$r_{i_1}-1$，所以新创建的归纳变量有一个比原来归纳变量的初始值小1的初始值0。增量相同。图10-19在标签“for r_1”的下方给出*Reduce*和*Apply*创建的SCC。最后，使用一个拷贝操作$r_1 \leftarrow r_{t_1}$替换r_1的定义。这一拷贝操作被标记为一个归纳变量。

535

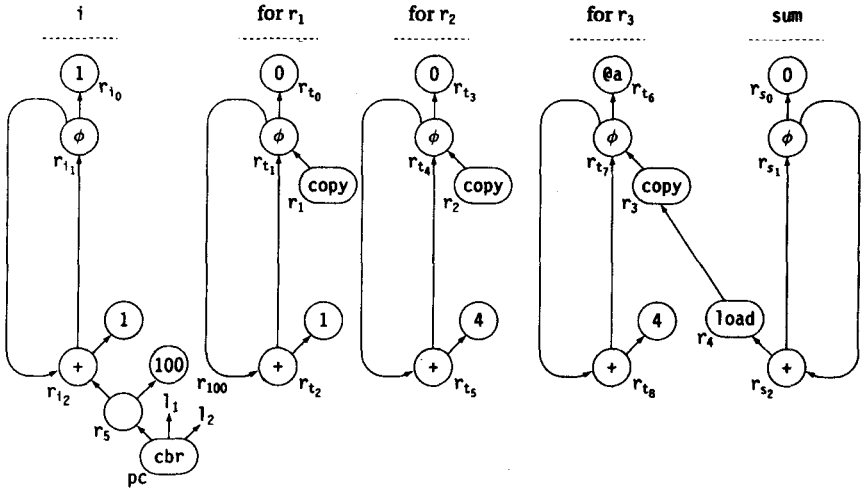


图10-19　示例的转换SSA图

下一步，*DFS*发现标签为r_2的结点组成的SCC。*Process*发现这个SCC是一个候选操作，因为它的左操作数（现在定义r_1的拷贝）是一个归纳变量，而它的右操作数是一个区域常量。*Process*调用*Replace*来创建一个值为$r_1 \times 4$的归纳变量。*Reduce*和*Apply*复制r_1的这个归纳变量，并调整增量，因为这个操作是一个乘法并把一个拷贝加到r_2。

下一步，*DFS*要把r_3的结点传送到*Process*。从而创建另一个以*@a*为初始值的归纳变量并把它的值拷贝到r_3。

*Process*处理*load*，其后跟着一个计算和的SCC。它发现这些操作都不是候选操作。

最后，*OSR*调用*DFS*以处理cbr的未访问结点。*DFS*访问这一比较，它是前面所标记的归纳变量，以及常量100。不再出现进一步的减弱操作。

536 图10-19中的SSA图给出此过程创建的所有归纳变量。标签为“for r_1”和“for r_2”的归纳变量是死的。i的归纳变量也将是死的，只有循环尾部测试仍然使用它。为了消除这一归纳变量，编译器可以运用线性函数测试替换来把这一测试转换成r_3的归纳变量。

线性函数测试替换

强度减弱通常消除归纳变量的所有使用，只有循环尾部测试的使用例外。对于循环尾部测试，编译器也许能够重写循环尾部测试使其使用循环中的另外一个归纳变量。如果编译器能够消除这最后的使用，那么它就可以消除作为死代码的原来的归纳变量。这一转换被称为线性函数测试替换（LFTR）。

为了执行LFTR，编译器必须（1）定位依赖于其他不需要的归纳变量的比较，（2）定位这一比较能够使用的一个适当的新归纳变量，（3）为这个被重写的测试计算正确的区域常量，以及（4）重写代码。使LFTR与*OSR*合作可以简化产生快速、高效转换的所有任务。

LFTR的目标操作是参照一个区域常量比较一个归纳变量的值。*OSR*检查程序中的每一个操作以确定这个操作是否是强度减弱的候选。它可以轻而易举地构建包含归纳变量的所有比较操作的列表。*OSR*完成它的工作之后，LFTR应该重访每一个比较。如果一个比较的归纳变量参数被*OSR*进行强度减弱，那么，LFTR应该重新定向这一比较来使用这一新的归纳变量。

为了使这一过程得以实现，*Reduce*可以记录用于得到它所创建的新归纳变量的数学关系。它可以插入从原来的归纳变量中的每一个结点到它的减弱副本的相应结点的特殊LFTR边，且使用负责创建新归纳变量的候选操作的操作码和区域常量对这个边赋标签。图10-20给出加到本例子的SSA图上的这些边。例子包含一系列减弱；这些减弱创建带有适当标签的边的链。从原来的归纳变量开始，我们可以找到标签－1、×4和＋@a。

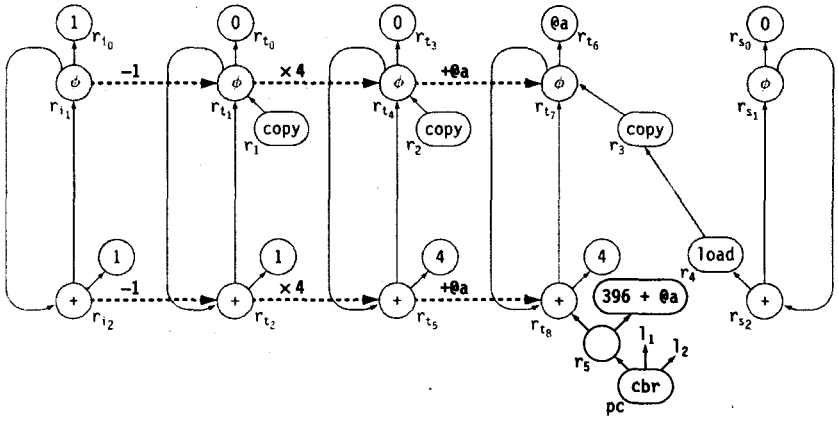


图10-20 LFTR后的示例

当LFTR找到应该被替换的比较时，它可以从它的归纳变量参数开始跟踪边链，直到得自于一个或多个减弱的链的最后归纳变量。这一比较应该在这一归纳变量和适当的新区域常量间进行。

LFTR所遍历的边上的标签描述为了得到新的区域常量而施加于原区域常量之上的转换。在本例子中，边的痕迹从r_{i_2}到r_{t_8}，并为被转换测试产生值 (100 − 1) × 4 + @a。图10-20给出这些边和被重写的测试。 537

LFTR的这一版本简单高效。它依赖于与*OSR*的密切合作来识别可能被重定向的比较并留下它所执行的减弱记录。使用这两个数据结构，LFTR能够找到重定向的比较，找到重定向它们的适当位置，找到比较的常量参数的必要转换。

10.4.3 优化的其他目标

1. 生成更小的代码

在某些应用中，编译代码的尺寸很重要。如果一个应用软件在其执行之前在相对缓慢的通信链之间传输，那么我们感觉到的运行时间是下载时间与运行时间之和。这使代码尺寸成为主要的考虑因素，因为在这一代码传输的时候用户需要等待。同样地，在很多嵌入式应用软件中，代码存储于一个永久、只读的内存（ROM）中。因为更大的ROM要花费更多的金钱，所以代码尺寸变成一个经济问题。 538

编译器设计者可以通过利用直接缩小代码的转换来解决这一问题。

- *提升*（hoisting）（如9.2.4节所述）通过使用单一等价操作替换多个相同操作来缩小代码。只要表达式在插入点是忙碌的，那么提升就不加长任意执行路径。
- *下沉*（sinking）在CFG中把若干公共的代码序列向前移动到只需这一序列的一个拷贝即可的地点。在下沉的一种特定形式，*交叉跳转*（cross jumping）中，编译器检查所有以相同标签为目标的跳转。如果在到这一标签的跳转之前是相同的操作，那么编译器可以把这一操作移动到这一标签并只保留这一操作的一个拷贝。这在不加长执行路径的前提下消除重复的操作。
- *过程抽象*（procedure abstraction）使用模式匹配技术来寻找重复的代码序列并使用对一个公共实现的调用替换这些重复代码。如果这一公共代码序列比这一调用所需要的代码序列长，那么上述做法可以节省空间。它使代码变慢，因为每一个抽象的代码序列被对这一抽象过程的调用以及一个跳转返回所取代。过程抽象可以运用于整个程序来寻找不同过程的公共序列。

作为另外一个方法，编译器可以避开加大代码的转换。例如，循环展开一般要加大代码。同样地，LCM可能由于插入新操作而加大代码。通过精心选择算法并避开引起代码增大的那些算法，编译器设计者可以构建总是生成简洁代码的编译器。

2. 避免页错误和指令缓冲失误

在某些环境中，页错误和指令缓冲失误所带来负荷使得我们值得以改进代码内存局部化的方式转换代码。编译器可以使用若干不同但又相关的效应来改进指令流的分页和缓冲行为。 539

- *过程放置*（procedure placement）：如果*A*反复调用*B*，那么编译器应该保证*A*和*B*在内存中占据相邻的位置。如果它们适合同一页面，那么这可以减少程序的工作设置。把它们放置在相邻位置还可以减少指令缓冲内冲突的可能性。
- *块放置*（block placement）：如果块b_i结束于一个分支且编译器知道这个分支常常把控制转到b_j，那么编译器就可以把b_i和b_j放置在连续的内存中。这使得b_j成为这一分支的落下情况（而且大多数处理器支持并欢迎这一落下情况）。它还增加指令缓冲中硬件预取机制的效率。
- *错误清除*（fluff removal）：如果编译器能够确定某一代码片段很少执行，即绝大部分时间它们是分支不采用的目标，那么编译器可以把它们移到较远的位置。这种很少执行的代码无需放在缓冲器中并降低带入处理器中的有用操作的密度。（值得把例外处理的代码都保留在同一页上，但应该

把它们放置不在同一道上。）

为了高效地实现这些转换，编译器需要了解代码中每一条路径的采用频率的精确信息。收集这些信息一般需要更复杂的编译系统，这样的编译系统收集执行简档数据并把这一数据反过来与代码相关联。用户编译一个应用软件并在“具有代表性的”输入上执行这一应用软件。在此之后，编译器使用来自于这些运行的简档数据在第二次编译中优化代码。与编译同样古老的另一个方法是构建CFG模型并使用合理转换可能性来估测执行频率。

10.4.4 优化序列的选择

一组特定转换的选取和运用这些转换的顺序是设计优化编译器的一项关键任务。对于任意特定问题，存在许多不同的技术。每一个技术适用于不同的情况。很多技术描述若干问题的某个局部。

为了使上述描述更具体，回想一下我们给出的消除冗余的方法。这些方法包括三个值编号技术（局
540 部、超局部和基于支配者的技术）和两个基于数据流分析（基于可用表达式的全局公共子表达式和惰性代码移动）的技术。值编号技术使用冗余的基于值的概念来寻找并消除冗余。这些技术的区别在于优化的作用域不同：覆盖单一块（局部）、扩展的基本块（超局部）以及整个CFG减去后边（基于支配者）。这些技术还执行常量传播并使用算术等式消除某些无用操作。基于数据流的技术使用冗余的词法概念：只有当两个操作有相同的名字时它们等价。全局公共子表达式消除只消除冗余，而惰性代码移动既消除冗余又消除部分冗余，并执行代码移动。

在设计优化编译器时，编译器设计者必须决定如何消除冗余。在上述这5种方法中的选择涉及决定哪些情况是重要的、实现的相对难度以及较之运行时利益相比，编译时代价如何等问题。使情况更为复杂的是，存在更多技术。

同样困难的是，不同效应的优化之间相互影响。一些优化可以创建另外一些优化的机会，正如常量传播和惰性代码移动通过把更多的值作为区域常量来改进强度减弱。对称地，一些优化可以降低另外一些优化的效果，如冗余消除可能通过使某些值在程序中的生存期更长来复杂化寄存器分配。两个优化的效应也可能会交迭。在实现稀疏条件常量传播的编译器中，值编号技术的常量叠入能力的重要性大打折扣。

在选择一组优化之后，编译器设计者必须选择运用这些优化的顺序。对这一顺序的某些约束是显而易见的；例如，值得在强度减弱之前运行常量传播和代码移动。而其他的约束却不是显然的；例如，循环反切换是否应该先于强度减弱？常量传播是否应该把$x \leftarrow y \times 4$，其中$y \geq 0$，转换成一个移位操作？这一决策依赖于哪些遍先于常量传播，哪些遍后于常量传播。最后，某些优化可能要执行多次。强度减弱之后，死代码消除进行清除的工作。在强度减弱之后，编译器可能再一次运行常量传播以试图把所有剩余的乘法转换成移位。另外，在判断已经获得了所有的全局效应的情况下，也可能使用局部值编号达到与上相同的效应。

如果编译器设计者希望提供不同层次的优化，那么他（她）需要设计若干编译序列，每一个编译都有自己的根本原因和自己的一组遍。更高层次的优化一般要增加更多的转换。它们还可能重复某些优化
541 来利用早前的优化所创建的机会。对于执行优化的顺序，可以参见Muchnick[262]。

10.5 概括和展望

优化编译器的设计和实现是一项复杂的工作。本章介绍了考虑转换的概念框架，即效应的分类。这一分类中的每一个范畴都给出了若干例子加以展示，这些例子或者是本章所给的例子，或者是本书其他

章节的例子。

编译器设计者所面临的挑战是选择一组能够良好地配合来生成良好代码的转换：代码必须满足用户的需求。在编译器中实现的特定转换在很大程度上决定能够产生良好代码的程序类型。

本章注释

尽管本章所给出的算法是现代的，但是很多基本思想早在20世纪60年代和20世纪70年代已是众所周知。死代码消除、代码移动、强度减弱和冗余消除都已由Allen（于1969年）[12]和Cocke、Schwartz[86]描述过。20世纪70年代以来的若干论文概括了这一领域的状况[15，28，308]。Morgan[260]和Muchnick[262]所著的书籍讨论了优化编译器的设计、结构和实现。Wolfe[335]、Allen和Kennedy[19]集中讨论基于相关性的分析和转换。

*Dead*以标记清除风格实现了由Kennedy[203，206]提出的死代码消除。它使人们想起Schorr-Waite的标记算法[299]。特别地，*Dead*是由Cytron等[104，参见7.1节]的工作编辑而成的。*Clean*是由Rob Shillner于1992年开发并实现的[245]。

LCM改进了Morel和Renvoise的经典部分冗余消除算法[259]。那篇原始论文激发了很多改进方法，包括[76，121，312，124]。Knoop、Rüthing和Steffen的惰性代码移动[215]改进了代码放置；10.3.2节给出的公式使用了Drechsler和Stadel[125]提出的方程。Bodik、Gupta和Soffa把这一方法与复制结合在一起来寻找并消除所有冗余代码[41]。

提升作为降低代码空间的一种技术出现于Allen-Cocke目录中。使用忙碌表达式的公式出现在若干地方，包括Fischer和LeBlanc[140]。下沉或交叉跳转是由Wulf等描述的[339]。 542

窥孔优化和尾递归消除开始于20世纪60年代初。窥孔优化最初是由McKeeman[253]所描述的。尾递归消除更加古老；传说McCarthy在1963年的一次讲座中在黑板上描述了这一方法。Steele的论文[314]是尾递归消除的经典参考资料。

稀疏简单常量算法SSCP归功于Reif和Lewis[285]。Wegman和Zadeck重新形式化SSCP以使用SSA形式并给出10.4.1节中的SCCP算法[328，329]。他们的工作阐明了乐观算法和悲观算法之间的差异；Click从集合构建的观点讨论了相同的问题[79]。

循环优化已得到广泛的研究[27，19]；Kennedy使用循环展开来避开循环尾部的拷贝操作[202]。Cytron、Lowreg和Zadeck提出了反切换的另一个有趣的方法[105]。Wolf和Lam把一组循环优化统一处理为称为*单一模块*（unimodular）的转换[334]。McKinley等人给出了内存优化对性能的影响的实践观点[89，254]。

组合优化，就像在SCCP中那样，通常导致独立运用原来的优化所不能得到的改进。值编号把冗余消除、常量传播和代数等式的简化结合起来[50]。LCM把冗余消除、部分冗余消除与代码移动结合起来[215]。Click和Cooper[81]把Alpern的划分算法[20]与SCCP结合起来[329]。很多作者把寄存器分配和指令调度结合起来[158，267，268，261，45，274，298]。

操作符强度减弱已有相当丰富的历史。强度减弱算法的一个系列就是来自于Allen、Cocke和Kennedy的工作[204，83，85，18，250]。*OSR*算法就是这一系列的成员。算法的另外一个系列来自于由LCM算法所展示的优化的数据流方法；有若干资料给出了这一系列中的技术[118，192，198，120，216，209，169]。10.3.3节中的*OSR*版本只减弱乘法。Allen等人展示了其他很多操作符的减弱序列[18]；扩展*OSR*以处理这些情况是直截了当的。

缩小现存代码或生成更小的程序是文献中永远的话题。Fabri给出通过存储覆盖的自动生成来减小数据内存需求的算法[134]。Marks构建了一个简洁、程序特化的指令集合并构建了该指令集合的一个解释

器[249]。Fraser、Myers和Wendt使用了基于后缀树的过程抽象[148]。最近的研究已集中于压缩代码以便网络传输 [130，145]并直接生成小代码：一些需要硬件解码器的帮助[233，234]，而另外一些无需这
543 样的帮助[99]。

代码放置的现代算法开始于Pettis和Hansen[273]。后来的工作包括分支调整[59，340]和代码布局[88，
544 73，156]。这些优化通过改进分层存储器系统中的代码内存行为来改进性能。它们还消除分支和跳转。



第11章
指令筛选

11.1 概述

当代码生成开始时，程序是以IR形式存在的。代码生成器必须把这一IR程序转换成在目标机器上可以运行的代码。编译器必须：（1）选择一个实现IR操作的目标机器操作序列，这是被称为指令筛选（instruction selection）的过程；（2）选择操作应该执行的顺序，这是被称为指令调度（instruction scheduling）的过程；（3）在程序每一点处决定哪些值应该驻留在寄存器中，这是被称为寄存器分配（register allocation）的过程。对于每一种情况，编译器必须重写代码以反映这些决策。大多数编译器分别处理这三种情况中的每一种。术语代码生成（code generation）经常指的是指令筛选。本章研究指令筛选中引发的各种挑战。下面两章将研究分析指令调度和寄存器分配。

在设计代码生成器时，编译器设计者必定要考虑代码的低级的细节和它们到指令集体系结构（ISA）上的映射，以及目标机器的硬件资源。如果IR程序已经表示某些或全部低级细节，那么指令筛选可以考虑这些细节并做出它相应的选择。如果IR在抽象的较高级别表示程序，那么指令筛选必须填补其缺或全部细节。（在编译的这一后期阶段机械地生成这样的细节可以导致带有低级细节定制的模板式代码。）很少执行或根本不执行优化的编译器直接从前端所产生的IR形式生成代码。

指令筛选的复杂性来自于这样的事实：在特定的目标机器上通常存在很多不同的执行给定计算的方法。我们暂时抛开指令调度和寄存器分配问题；我们将在下两章回到这些问题上来。如果每一个IR操作在目标机器上恰好只有一个实现，那么编译器可以简单地把每一个这样的操作映射到等价的机器操作序列上。然而，在大多数情况中，目标机器可以用很多方法实现每一个IR结构。

例如，在一台以ILOC为固有指令集的机器上，考虑把通用寄存器r_1中的值拷贝到另一个寄存器r_2中，正如我们将看到的那样，即使是ILOC也足以揭露出代码生成中很多问题的复杂性。$r_1 \rightarrow r_2$的一个显然的实现是使用i2i $r_1 \Rightarrow r_2$；通常，这样的寄存器到寄存器拷贝是处理器提供的最廉价的操作之一。然而，有很多其他实现。例如包括下面这些操作：

```
addI    r1,0 ⇒ r2        subI    r1,0 ⇒ r2
multI   r1,1 ⇒ r2        divI    r1,1 ⇒ r2
lshiftI r1,0 ⇒ r2        rshiftI r1,0 ⇒ r2
orI     r1,0 ⇒ r2        xorI    r1,0 ⇒ r2
```

还有一个使用andI的操作，其中，andI的一个操作数是所有位均为1的常数。如果处理器维护一个值总是零的寄存器，那么使用add、sub、lshift、rshift、or和xor的另一组操作也适用。更多两操作序列也适用，包括一个存储后面跟着一个装入，以及使用第3个寄存器的一对xor操作。很多其他多操作序列也是可能的。

程序员将快速否定绝大多数的可能序列。使用i2i简单、快速且显然。然而，自动过程也许需要考虑所有的可能性并做出适当的选择。特定指令集合用多种方法实现相同效应的能力增加了指令筛选的复杂性。对于ILOC，ISA为每一个特定效应只提供几个简单的低级操作。即使这样，它仍然支持很多实现

寄存器到寄存器拷贝的方法。

546 实际的处理器更加复杂。它们可能包含代码生成器应该考虑的高级操作和多寻址模型。尽管这些特性允许熟练程序员或精心设计的编译器创建更高效的程序，但是它们也增加指令筛选器所要面对的选择数量：它们使可能实现的空间更大。

每一个可选序列都有自己的成本。大多数机器实现诸如i2i、add和lshift这样的简单操作，这些操作在一个周期内执行。有一些操作，例如整数乘法和除法，所花的时间会更长。内存操作的速度取决于很多因素，其中包括计算机内存系统的当前状态细节。

在某些情况下，一个操作的实际代价可能依赖于上下文。例如，如果处理器有若干个功能单元，那么不使用寄存器到寄存器的拷贝而使用一个在未加以利用的功能单元上执行的操作更好。如果不使用这一操作时这个功能单元是空闲的，那么从效果上看，这一操作是免费的。把这一操作移到这个未加以利用的单元上实际上加速整个计算。如果代码生成器必须把拷贝重写成只在未加以利用的功能单元上执行的特定操作，那么这是一个筛选问题。否则就是一个指令调度问题。

在大多数情况下，编译器设计者想要后端产生快速运行的代码。然而，其他度量标准也是可能的。例如，如果最终的代码在以电池供电的设备上运行，那么编译器可能要考虑每一个操作的一般能源消耗。（各个操作可能消耗不同量的能源。）设法优化能源使用的系统中的成本可能根本不同于速度度量所使用的成本。能源成本很大程度依赖于硬件的细节，因此，这一成本也将随着处理器的实现而发生变化。同样地，如果代码空间是重要的度量标准，那么编译器设计者可能只基于序列长度指定成本。另外，编译器设计者也可能简单地拒绝与单一操作序列产生相同效应的多操作序列。

使问题进一步复杂化的是某些ISA对特定操作设置额外的限制。整数乘法可能需要从寄存器的一个子区域提取它的操作数。浮点操作可能需要它的操作数在偶数标号的寄存器中。内存操作可能只在处理器的特定功能单元上执行。浮点单元可能支持序列$r_i \times r_j + r_k$的单一操作实现，而这一实现比乘法和加法组成的序列运行得快。装入乘法和存储乘法操作也许需要相邻的寄存器。内存系统可能只对双字或四倍字长的装入发放最佳带宽和等待时间，而对单字装入则不发放最佳带宽和等待时间。诸如这样的约束限
547 制指令筛选。同时，它们也增加寻找在输入程序的每一点使用最佳操作的解决方案的重要性。

当IR和目标机器的抽象层次明显不同时，或者它们的计算模型不同时，指令筛选可以在衔接它们之间的这种差距中起重要的作用。指令筛选把IR程序中的计算高效地映射到目标机器的程度通常决定所生成代码的效率。例如，考虑由类ILOC的IR生成代码的三种模式：

1）简单、标量RISC机器 从IR到汇编语言的映射是直截了当的。代码生成器为每一个IR操作可能只考虑一个或两个汇编语言序列。

2）CISC处理器 为了有效使用CISC指令集合，编译器可能需要把若干IR操作聚合成一个目标机器操作。

3）栈机器 代码生成器必须把ILOC的寄存器到寄存器的计算风格翻译成基于栈的风格，这种风格带有隐式名字并在某些情况下带有破坏性操作。⊖

当IR和目标ISA之间的抽象层次的差距加大时，帮助构建代码生成器的工具的需要也加大。

尽管指令筛选在决定代码质量中起着重要的作用，但是编译器设计者必须牢记，指令筛选可能带来的巨大的搜索空间。正如我们将看到的那样，即使是中等大小的指令集合也可能产生包含数以百万计的状态搜索空间。显然，编译器无法承受以不周密或者穷举的方法对这样的空间进行探索。我们所描述的

⊖ 将单地址或栈IR（如字节码）移到三地址的机器上伴随类似的问题。代码生成器必须引入新的名字空间。到三地址形式的翻译可能引入复用值的机会，而这又引发诸如局部值编号这样的新的优化机会。

技术使用规范方法探索可选代码序列空间，并且或者限制它们的搜索，或者预计算足够的信息来使深层搜索更有效。

构建可重定位编译器

代码生成的系统方法已使构建*可重定位编译器*（retargetable compiler）成为可能。可重定位编译器一般支持多个指令集合的机制以及化简加入更多指令集合和的机制。 548

指令筛选与调度和分配之间的相互作用

后端的三个主要的过程是指令筛选、指令调度和寄存器分配。这三个过程都对生成代码的质量有直接的影响，而且它们互相作用。

指令筛选直接改变调度的过程。筛选既控制操作所需的时间又控制它在执行时使用的功能单元。调度可能影响指令筛选。如果代码生成器可以使用两个汇编操作之一实现一个IR操作，而且这些操作使用不同的资源，那么代码生成器可能需要了解最终的调度以保证做出最好的筛选。

筛选在若干方面与寄存器分配间相互影响。如果目标机器拥有统一的寄存器集合，那么指令筛选器可以假设寄存器的供给不受限制，并且依赖于分配器来插入使值适合寄存器集合的装入和存储操作。另一方面，如果目标机器拥有限制寄存器使用的规则，那么筛选器必须对特定的物理寄存器格外小心。这可能会使筛选复杂化，并且要预先决定一些或所有的分配决策。在这种情况下，代码生成器可能使用一个协同程序在指令筛选期间执行局部寄存器分配。

尽量分离筛选、调度和分配可以化简每一个过程的实现和调试。然而，因为这些过程的每一个都可能制约其他的过程，所以编译器设计者必须小心避免加入一些不必要的限制。

开发可重定位编译器的目标是最大化编译器中各组成部分的使用。理想的情况是，前端和优化器需要最小的变化，而后端的大部分也可复用。从而充分利用在构建、调试和维护这个编译器的公共部分上的投资。

可移植性的现代方法让指令筛选器负起处理不同的目标的责任。编译器对所有目标，在某些情况下 549
对所有源语言使用共同的IR。⊖这一方法基于在大多数目标机器上为真的一组假设优化中间形式。最后，编译器设计者设法分离并提取目标相关细节来创建后端。

（因为调度器和寄存器分配器操作于目标机器的汇编代码，所以编译器的这些部分需要目标相关的信息。这一信息包括操作等待时间、寄存器单元的大小、功能单元的数量和能力、调整限制、目标机器的调用约定以及其他种类的细节。但是，即便如此，编译器在整个平台上分享基础算法和实现。）

构建可重定位编译器的最重要一点在于指令筛选器的自动构造。图11-1给出一个可能的系统。这一指令筛选器是由与一组表格结合在一起的模式匹配引擎组成的，这些表格编码从IR到目标ISA的映射所需的必要信息。结果筛选器消耗编译器的IR并为目标机器产生汇编代码。在这样的系统中，编译器设计 550
者创建目标机器的描述并运行后端生成器，有时称为*代码生成器生成器*（code-generator generator）。而后端生成器使用这一描述得到模式匹配器所需的表格。像分析器生成器一样，后端生成器在编译器开发过程中脱机运行。因此，我们可以使用算法来创建比长时间的编译需要更多时间的这一表格。

尽管我们的目标是分离指令筛选器、调度器和寄存器分配器中的所有机器相关代码，但是现实几乎总不是那么理想。某些机器相关细节不可避免地渗透到编译器的早期部分。例如，活动记录的调整限制

⊖ 在实践中，新语言的加入通常意味着给IR增加一些新的操作。然而，目标是扩展IR而不是重新确立它。

可能因目标机器的不同而不同，这改变存储在ARS中的值的偏移。如果编译器要充分利用诸如预测执行、分支等待槽和多字内存操作，那么它也许需要显式地表现这些特性。然而，尽量把目标相关的细节注入指令筛选中可以减少编译器在其他地方必须做出的改变的数量，而且可以化简重定位的任务。

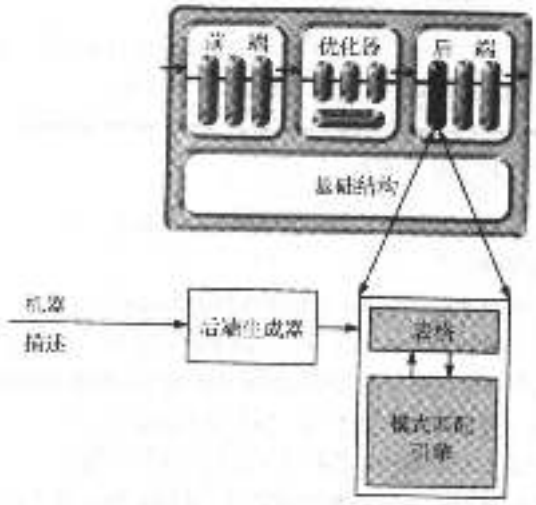


图11-1 一个可重定位的后端

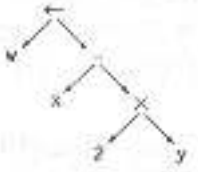
本章研究两个指令筛选器自动构造法。下一节介绍指令筛选的简单树遍历算法，并以此对这一问题给出详细的介绍。随后两节给出运用模式匹配技术把IR序列转换成汇编序列的各种方法。这两个方法都是基于描述的。编译器设计者以形式表记法描述目标机器的ISA，然后，我们使用一个工具构建编译时使用的模式匹配指令筛选器。二者都已用于成功的可移植编译器上。

第一个技术使用树模式匹配理论来创建自底向上的重写系统。至少在概念上，这一技术假设编译器使用类似于树的IR。编译器设计者使用类似于低级抽象语法树的树型模式描述目标机器的ISA。每一个模式采用以值和实现子树的相关汇编序列替换这一子树的重写规则的形式。模式匹配器发现把一个IR树归约成单一值的一个重写序列；第二遍跟踪这一重写序列并发行相关的代码。

第二个技术采用一个众所周知的优化技术，窥孔优化（peephole optimization），并把这一优化技术用于代码生成。窥孔优化器从一组模式库出发。它识别并改进这些模式的实例。至少在概念上，它们工作于线性IR上。它们把注意力局限于操作的一个小窗口——窥孔。这简化模式匹配并提高模式匹配的速度。它们也限制能够发现的匹配集合。基于窥孔的指令筛选从20世纪80年代早期已经开始使用。通过为
551 每一个IR结构构建一个将其重写成目标机器的汇编代码的规则，编译器设计者可以使窥孔优化器使用一个汇编程序替换IR程序。增加模式通常可以改进代码质量。11.4节将更详细地讨论这一方法。

11.2 简单的树遍历方案

为了使讨论更具体，考虑为诸如w←x−2×y这样的赋值语句生成代码时出现的问题。这一赋值语句可以表示成AST的形式，如下图左边所示，或表示成一个四元组表，如下图右边所示。



操作符	Arg_1	Arg_2	结果
×	2	y	t_1
−	x	t_1	w

指令筛选必须从上述两个IR表示产生一个汇编语言的程序。出于讨论的缘故，假设它必须生成如图11-2所示的ILOC子集中的操作。

算术运算		存储器操作	
add	$r_1, r_2 \Rightarrow r_3$	store	$r_1 \Rightarrow r_2$
addI	$r_1, c_2 \Rightarrow r_3$	storeAO	$r_1 \Rightarrow r_2, r_3$
sub	$r_1, r_2 \Rightarrow r_3$	storeAI	$r_1 \Rightarrow r_2, c_3$
subI	$r_1, c_2 \Rightarrow r_3$	loadI	$c_1 \Rightarrow r_3$
rsubI	$r_2, c_1 \Rightarrow r_3$	load	$r_1 \Rightarrow r_3$
mult	$r_1, r_2 \Rightarrow r_3$	loadAO	$r_1, r_2 \Rightarrow r_3$
multI	$r_1, c_2 \Rightarrow r_3$	loadAI	$r_1, c_2 \Rightarrow r_3$

图11-2 ILOC子集

在第7章中，我们已经看到一个可以从表达式的AST生成代码的简单树遍历例程。图7-2中的代码处理作用于变量和数字的二元操作符+、-、×和÷。它为表达式生成朴素代码并力图说明可以用于生成低级线性IR或简单RISC机器的汇编代码的方法。 552

代码布局

编译器在开始发行代码之前，它拥有在内存中对基本块进行布局的机会。如果IR中的每一个分支有两个明确的分支目标，如ILOC那样，那么编译器可以选择在内存中把一个块的其中一个逻辑后继放在其后。只使用一个显式分支目标，这通常称为落下分支（fall-through branch），对基本块进行重新布局需要更多的工作。

在体系结构上，有两种可以左右这一决策的考虑。对于某些处理器，采用分支需要比落下到下一个操作需要更多的时间。在带有高速缓冲存储器的机器上，一起执行的块应该放置在一起。两种考虑都欢迎同样的布局策略。如果块a结束于以b和c为目标的分支，那么编译器应该在内存中在a之后放置更频繁采用的目标。

当然，如果一个块在控制流图中有多个前驱，那么它们中只有一个能在内存中直接领先于它。而其他前驱将需要一个分支或跳转来达到它。

这一简单树遍历方法为特定AST结点类型的每一个实例生成相同的代码。尽管这产生正确的代码，但是它从不利用这一机会制作特定环境和上下文下的代码。如果编译器在指令筛选之后执行有意义的优化，那么这也许不成问题。然而，在没有后继优化的情况下，最终的代码很可能包含显然低效的地方。

例如，考虑简单树遍历例程处理变量和数字的方式。相应情况的代码是：

```
case IDENT:
  t1 ← base(node);
  t2 ← offset(node);
  result ← NextRegister();
  emit (loadAO, t1, t2, result);
  break;
```

```
case NUM:
  result ← NextRegister();
  emit (loadI, val(node),
        none, result);
  break;
```

对于变量，这一代码依赖于两个例程*base*和*offset*来得到进入寄存器的基地址和偏移。然后，它发行loadAO对基地址和偏移求和来生成一个有效地址并取回在内存这一有效地址处的内容。因为AST不区分 553

变量的存储种类，所以*base*和*offset*可能要参考符号表得到它们所需的额外信息。

把这一方案扩展到更现实的情况，包含有不同大小表示的变量、值调用和引用调用参数以及在整个生存期都驻留在寄存器中的变量，将需要写出检查每一次引用的每一种情况的明确代码。这将使得*IDENT*情况的代码更长（而且更慢）。它消除手工编码树遍历方案的大部分引人入胜的简洁性。

处理数的代码同样是不成熟的。这一代码假设一个数在每一个种情况下都应该被装入到寄存器中，并假设*val*能够从符号表中取得这个数的值。如果使用这一数的操作（它在树中的父结点）在目标机器上有立即形式，而且常量的值适合它的立即域，那么编译器应该使用这一立即形式，因为这样可以少用一个寄存器。如果这个数是立即操作不支持的类型，那么编译器必须设法在内存中存储这个值，并生成一个适当的内存引用把这个值装入一个寄存器中。从而，这将创建进一步改进的机会，例如将这个常量值保留在寄存器中。

为了使这一讨论具体化，考虑图11-3所示的三个乘法操作。符号表注释现显在树中叶结点的下方。对于标识符，它是由一个名字、一个基地址标签（或表明当前活动记录的ARP）以及距离基地址的偏移组成的。每一棵树的下方有两个代码序列：由简单树遍历评估器生成的代码序列和我们希望编译器生成的代码序列。在第一种情况的a × b中，低效性来自于树遍历方案不生成loadAI操作。*IDENT*情况中更复杂的代码可以解决这一问题。

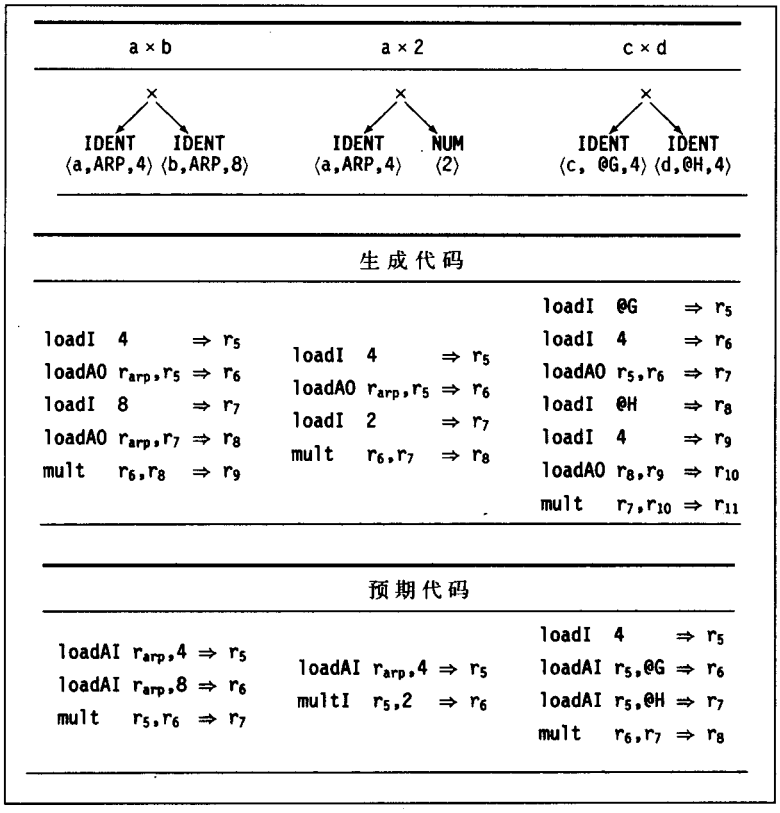


图11-3 乘法种类

第二种情况的a × 2比较麻烦。代码生成器可能使用multI操作实现乘法。然而，为了识别这一事实，代码生成器必须查看这一局部上下文之外的上下文。为了在树遍历方案中实现这一功能，对于 × 的这一

情况可能识别出评估到常量的一棵子树。或者，处理*NUM*结点的代码可能确定它的父结点可以使用立即操作来实现。无论哪种方法，它都需要非局部上下文，而非局部上下文破坏简单树遍历范例。

第三种情况的c × d有另一种非局部问题。× 的两棵子树都引用距基地址偏移为4处的变量。这一引用有不同的基地址。原来的树遍历方案为每一个常量生成一个明确的loadI操作：@G、4、@h和4。正如前面所提到那样，使用loadAI的修改版本将为@G和@H生成不同的loadI或者它将为4生成两个loadI。（当然，@G和@H的值的长度开始发挥作用。如果它们太长，那么编译器必须把4作为loadAI操作的立即操作数。） 554

第三个例子的基本问题在于最终的代码包含一个隐藏在AST中的公共子表达式。为了发现这个冗余并适当处理它，代码生成器将需要显式检查子树基地址和偏移值的代码，并为所有情况生成适当的序列。以这种方式对一种情况进行处理将是很笨拙的。对可能出现的所有类似情况进行处理将需要较少的附加编码。 555

处理这种冗余的更好方法是揭示IR中的冗余细节，并让编译器来消除它们。对例子中的赋值w←x-2 × y，前端会产生如图11-4所示的低级树。这棵树有若干种新结点。Val结点代表已知驻留在寄存器中的值，例如，r_{arp}中的ARP。Lab结点代表可重定位的符号，一般是用于代码或数据的汇编级标签。◆结点表示间接层次；它的子结点是一个地址而且它生成存储于这个地址的值。这些新结点类型要求编译器设计者指定更多的匹配规则。然而，反过来，附加的细节可以用于诸如在c × d中优化4的重复引用。

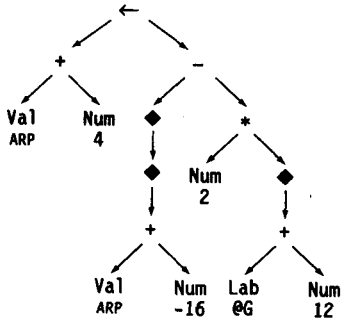


图11-4 w←x-2 × y的低级AST

上述这棵树的这一版本比目标LIOC指令集合揭示更低级别的细节。例如，检查这棵树揭示w是存储在距ARP偏移为4的局部变量，而 × 是一个引用调用参数（注意那两个◆结点），y被存储于距标签@G偏移为12的地方。另外，隐含于loadAI和storeAI操作中的加法显式地出现在这棵树上，作为◆结点的子树或←结点的左子结点。

在AST中揭示更多细节将导致更好的代码。增加代码生成器所考虑的目标机器的操作数量也将导致更好的代码。然而，把这些因素结合起来创建这样一种情况，在这里，代码生成器可以发现很多实现给定子树的不同方法。这一简单树遍历方案对每一种AST结点类型都有一个选项。为了高效使用目标机器指令集合，代码生成器将尽可能考虑更多可用的可能性。 556

最优代码的生成

筛选指令的树遍历方案每当遇到特定种类的AST结点时生成相同的代码序列。更现实的方案考虑多个模式并使用成本模型来在它们中间做出选择。很自然，这将引发这样一个问题，编译器能够做出最优选择吗？

如果每一个操作有相应的成本，而且我们忽视指令调度和寄存器分配的效应，那么最优指令筛选是可能的。11.3节中所述的树模式匹配代码生成器生成局部最优序列，即每一棵子树由最小代价的序列计算。

以单一成本数刻画运行时行为的困难引起对这一主张的重要性的疑问。执行顺序、有限硬件资源以及内存层次中的上下文相关行为的影响都将使确定任意特定代码序列的实际成本的问题变得复杂。

在实践中，大多数现代编译器在指令筛选期间很大程度上忽视调度和分配的效应。并假定与各个重写规则相关的代价是精确的。考虑到这些假定，编译器寻找局部最优序列，即最小化整棵子树的评估代价的序列。然后，编译器在由指令筛选所生成的代码上执行一遍或多遍调度和分配。

这增加的复杂性并不是由特定方法或特定的匹配算法引起的，而是由于它反映了实际情况：给定的机器可能提供很多实现IR结构的不同方法。当代码生成器考虑给定子树的多种可能匹配时，它需要在它们中间做出选择的方法。如果编译器设计者可以为每一个模式附加一个代价，那么这一匹配方案可以以最小化代价的方式选择模式。如果代价真实地反映性能，那么这种代价驱动指令筛选将导致更好的代码。

编译器设计者需要一个工具来帮助管理现实机器的代码生成复杂性。编译器设计者不编写出明确导
557 航IR并测试每一操作的可用性的代码，而是编写描述规则，而这些工具将生成把这些规则与代码的IR形式匹配所需的代码。下面两节讨论两种管理现代机器的指令集合中所出现的复杂问题的方法。下一节讨论树模式匹配技术的使用。这些系统把复杂性融入构建匹配器的过程中，正如扫描器把它们的选择融入DFA的转换表中一样。后面的一节将研究分析指令筛选中窥孔优化的使用。基于窥孔的系统把选择的复杂性转化成一个统一的方案，这一方案先做低级化简，然后寻找适当指令的模式匹配。为了保持匹配成本处于较低的水平，这些系统把它们的作用域局限在短的代码段上：一次只处理两三个操作。

11.3 通过树模式匹配实现指令筛选

编译器设计者可以使用树模式匹配工具处理指令筛选的复杂性。为了把代码生成转换成树模式匹配，程序的IR形式和目标机器指令集合必须被表示成树的形式。正如我们已看到的那样，编译器可以使用低级AST作为被编译代码的细节模型。它可以使用类似的树表示目标处理器上可用的操作。例如，ILOC的加法操作可以用如下形式的操作树来模型化：



通过系统地把操作树与AST的子树进行匹配，编译器可以发现这一子树的所有可能实现。

为了使用树模式，我们需要描述它们的更便利的表记法。使用前缀表记法，我们可以把add的操作树写成+(r_i, r_j)，而把addI的操作树写为+ (r_i, c_j)。这一操作树的叶子对操作数存储类型的相关信息加以编码。例如，在+(r_i, c_j) 中，符号r表示在寄存器中的一个操作数，而符号c表示一个已知常量操
558 作数。我们添加下标来保证惟一性，就如我们在属性文法的规则中所做的那样。如果我们以前缀形式重写图11-4的AST的话，它变成：

$$\leftarrow(+(Val_1,Num_1),\ -(\blacklozenge(\blacklozenge(+(Val_2,Num_2))),\ \times(Num_3,\ \blacklozenge(+(Lab_1,Num_4))))).$$

虽然画出树来更加直观，但是这一线性前缀形式包含相同的信息。

给定AST和操作树的集合，我们的目标是通过把操作树铺盖在AST上来把这个AST映到操作上。铺盖是序对<*ast-node*, *op-tree*>的集合，其中*ast-node*是AST中的一个结点，而*op-tree*是一棵操作树。在一个铺盖中序对<*ast-node*, *op-tree*>的存在意味着对应于*op-tree*的目标机器操作可以实现*ast-node*。当然，*ast-node*的实现的选择依赖于它的子树的实现。对于*ast-node*的每一棵子树，铺盖刻画与*op-tree*相关联的一个实现。

一个铺盖实现AST，如果它实现每一个操作而且每一块“瓦”与它的邻居“相连结”。对于每一个<*ast-node*, *op-tree*>，我们说一个块瓦与它的邻居连接，如果或者*ast-node*是AST的根，或者*ast-node*被这一铺盖中的另一个*op-tree*的叶子所覆盖。在两个这样的树交迭的地方（在*ast-node*），它们必须在公共结点的对应值的存储位置上达成一致。如果二者都假设它在寄存器中，那么这两个*op-tree*的代码序列是

相容的。如果一个假设它在内存中，而另一个假设它在寄存器中，那么这两个*op-tree*的代码序列是不相容的，因为它们不能正确地把由较低的树产生的值传送到上方树的叶子上。

给定实现AST的铺盖，编译器能够很容易地在自底向上的遍历中生成汇编语言。因此，使这一方法更实用的关键在于算法能够快速地发现AST的好铺盖。已出现若干对低级AST进行树模式匹配的技术。所有这些系统都为每一棵操作树关联一个代价，并生成最小代价铺盖。它们因用于匹配中的技术的不同而不同，这些技术包括树匹配、文本匹配以及自底向上的重写系统，还因它们的代价模型的一般性不同而不同，代价模型包括静态固定的代价模型以及在匹配过程中可以发生变化的代价模型。

11.3.1 重写规则

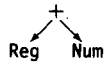
编译器设计者把AST中的操作树及子树之间的关系编码成一组**重写规则**（rewrite rule）。这一规则集合对这个AST中的每一种结点包含一个或多个规则。一个重写规则是由树文法中的一个产生式、一个代码模板和一个相关的代价组成的。图11-5给出使用ILOC操作铺盖低级AST的重写规则集合。 559

	产生式	代价	代码模板
1	Goal → Assign	0	
2	Assign → ← (Reg_1,Reg_2)	1	store r_2 ⇒ r_1
3	Assign → ← (+ (Reg_1,Reg_2), Reg_3)	1	storeAO r_3 ⇒ r_1,r_2
4	Assign → ← (+ (Reg_1,Num_2), Reg_3)	1	storeAI r_3 ⇒ r_1,n_2
5	Assign → ← (+ (Num_1,Reg_2), Reg_3)	1	storeAI r_3 ⇒ r_2,n_1
6	Reg → Lab_1	1	loadI l_1 ⇒ r_{new}
7	Reg → Val_1	0	
8	Reg → Num_1	1	loadI n_1 ⇒ r_{new}
9	Reg → ◆ (Reg_1)	1	load r_1 ⇒ r_{new}
10	Reg → ◆ (+ (Reg_1,Reg_2))	1	loadAO r_1,r_2 ⇒ r_{new}
11	Reg → ◆ (+ (Reg_1,Num_2))	1	loadAI r_1,n_2 ⇒ r_{new}
12	Reg → ◆ (+ (Num_1,Reg_2))	1	loadAI r_2,n_1 ⇒ r_{new}
13	Reg → ◆ (+ (Reg_1,Lab_2))	1	loadAI r_1,l_2 ⇒ r_{new}
14	Reg → ◆ (+ (Lab_1,Reg_2))	1	loadAI r_2,l_1 ⇒ r_{new}
15	Reg → + (Reg_1,Reg_2)	1	add r_1,r_2 ⇒ r_{new}
16	Reg → + (Reg_1,Num_2)	1	addI r_1,n_2 ⇒ r_{new}
17	Reg → + (Num_1,Reg_2)	1	addI r_2,n_1 ⇒ r_{new}
18	Reg → + (Reg_1,Lab_2)	1	addI r_1,l_2 ⇒ r_{new}
19	Reg → + (Lab_1,Reg_2)	1	addI r_2,l_1 ⇒ r_{new}
20	Reg → − (Reg_1,Reg_2)	1	sub r_1,r_2 ⇒ r_{new}
21	Reg → − (Reg_1,Num_2)	1	subI r_1,n_2 ⇒ r_{new}
22	Reg → − (Num_1,Reg_2)	1	rsubI r_2,n_1 ⇒ r_{new}
23	Reg → × (Reg_1,Reg_2)	1	mult r_1,r_2 ⇒ r_{new}
24	Reg → × (Reg_1,Num_2)	1	multI r_1,n_2 ⇒ r_{new}
25	Reg → × (Num_1,Reg_2)	1	multI r_2,n_1 ⇒ r_{new}

图11-5 使用ILOC铺盖低级树的重写规则

考虑规则16。它描述计算定位于Reg中的一个值与Num中的一个立即值的和的树。表的左边给出这一

规则Reg→+(Reg$_1$, Num$_2$)的树模式。中间一列列出它的代价1。右边一列给出一个实现这一规则的ILOC操作addI r_1, n_2⇒r_{new}。这一树模式中的操作数Reg$_1$和Num$_2$对应于这一代码模板中的操作数r_1和n_2。编译器必须用被分配保存这一加法结果的寄存器名字重写这一代码模板中的域r_{new}。而这一寄存器的名字又将成为连结到这一子树的子树的叶子。



这一树模式的线性形式所显示的直观性不如树表示。在下面的讨论中，我们常常使用图的形式来表示模式。例如，下图表示规则16，在＋结点处的它的结果隐式地是一个Reg。

图11-5给出的树文法与我们用于描述程序设计语言的语法的文法类似。每一个重写规则，或产生式的左部都是一个非终结符。在规则16中，这一非终结符是Reg。Reg表示树文法能够生成的子树的一个集合，对于这一情况，是使用规则6到规则25生成的子树集合。一个规则的右部是一个线性化了的树模式。对于规则16，这个线性化了的树模式是+(Reg$_1$, Num$_2$)，表示两个值Reg和Num的加法。

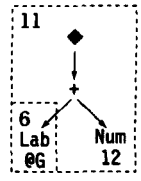
这一文法中的非终结符允许抽象。它们负责连结文法中的规则。它们还编码运行时对应值的存储位置及采用形式的信息。例如，Reg表示一个由一棵子树产生的且存储于一个寄存器的值，而Val则表示已经在寄存器中的一个值。Val可以是一个全局值，ARP就是如此。它也可能是在一棵不相交的子树上，即一个公共子表达式上，所执行的计算的结果。

与一个产生式相关联的代价应该为代码生成器提供运行时在这一模板中执行这一代码的真实估测。对于规则16，代价1表明使用需要一个执行周期的单一操作就可以实现这棵树。代码生成器使用这些代价在各种可能的选择中做出选择。某些匹配技术把代价限制到数值上。而其他匹配技术则允许代价在匹配期间发生变化，以反映前面的选择对当前选择的代价的影响。

560 ~ 561

树模式可以以简单树遍历代码生成器所无法做到的方式捕获上下文。规则10到规则14中的每一个都匹配两个操作符（◆和＋）。这些规则显示可以使用ILOC操作符loadAO和loadAI的条件。树遍历代码生成器一次只匹配一个操作符。与这些规则之一匹配的任意子树都能够通过与其他规则的结合而被铺盖。与规则10匹配的子树能够通过与产生一个地址的规则15和装入这个值的规则9结合而被铺盖。这一机动性使得重写规则集合具有歧义性。歧义性反映了这样的事实：目标机器具有若干实现这一特定子树的方法。

在图11-5中，Reg既作为终止符又作为非终结符出现。这反映本例中的一个缩写。规则的完整集合将包含用特定寄存器名重写Reg的产生式集合，诸如Reg→r_0、Reg→r_1、…、Reg→r_k。



⟨6,11⟩

为了把这些规则运用于一棵树，我们寻找把这棵树归约成一个符号的重写步骤序列。对于一个表示完整程序的AST，这个符号应该是目标符号。对于一个内部结点，那个符号一般表示以评估以这一表达式为根的子树所产生的值。这一符号还必须指定这个值存在的地方：一般是在一个寄存器中、在一个内存位置，或者是一个常量值。

例如，考虑表示对图11-4中的y的引用的子树，这一子树就是图11-6中最左边的版面所给出的图。（回想一下，y处于距标签@G偏移为12的地方。）图11-6的其余版面给出这一子树的一个归约序列。在这一序列中的第一个匹配发现左叶子（Lab结点）匹配规则6。这允许我们把这个左叶子重写成一个Reg。现在，这一重写了的树与规则11的右部◆（+(Reg, Num)）匹配，所以我们能够把以◆为根的整个子树重写为一个Reg。因此，重写序列<6, 11>把整个子树归约到一个Reg。

562

为了概括这样一个序列，我们将使用如上图左边所示的图。被点线框起来的盒子表示与这棵树匹配的特定右部，其中规则的标号记录在每一个盒子的左上角。图的下方的规则标号序列表示被运用的规则序列。重写序列使用最后规则的左部替换被框起来的子树。

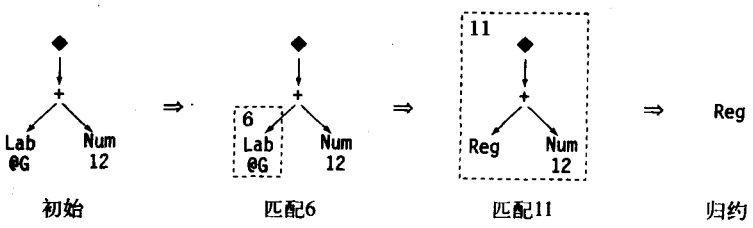


图11-6 一个简单的树重写序列

注意非终止符是如何保证操作树在它们交迭点处的适当连结的。规则6把Lab重写成Reg。规则11的左叶是Reg。把这一模式看成文法中的规则把发生于操作树边界处的所有考虑都叠入非终结符的标记中。

对于这个平凡子树，规则生成很多重写序列，这反映了这一文法的歧义性。图11-7给出8个这样的序列。在我们的方案中，除了规则1和规则7外，所有规则都有代价1。因为我们的重写序列都不使用这两个规则，所以它们的代价与它们的序列长度相等。这些序列根据代价分为三个范畴。第一个范畴是一对序列<6，11>和<8，14>，每一个都有代价2。第二个有四个序列<6，8，10>、<8，6，10>、<6，16，9>和<8，19，9>，每一个都有代价3。最后一个是两个序列<6，8，15，9>和<8，6，15，9>，每一个都有代价4。 563

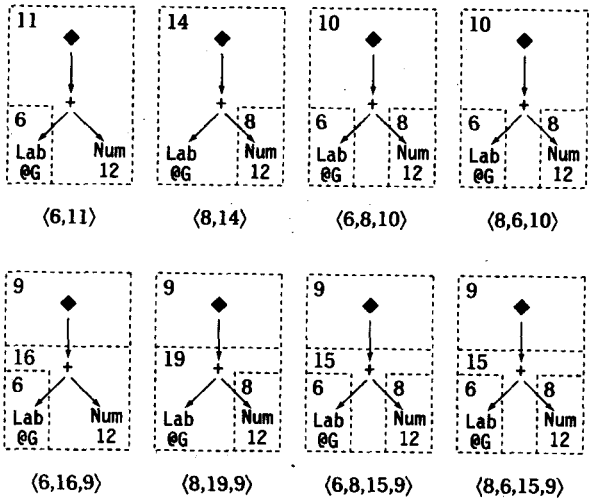


图11-7 可能的匹配

为了产生汇编代码，筛选器使用与每一规则关联的代码模板。一个规则的代码模板是由实现产生式所生成的子树的汇编代码操作序列组成的。例如，规则15把树模式+(Reg$_1$，Reg$_2$)映射到代码模板 add r_1，r_2⇒r_{new}。筛选器使用保存r_1和r_2所对应的子树的结果的寄存器名替换r_1和r_2。它为r_{new}分配一个新的虚拟寄存器名。AST的一个铺盖描述代码生成器应该使用的规则。代码生成器使用相关的模板来在自底向上遍历的过程中生成汇编代码。必要时，它提供名字来把存储位置结合到一起并发行对应于这一遍历的实例操作。

代码生成器应该选择产生最低代价汇编序列的铺盖。图11-8给出对应于每一个可能铺盖的代码。所有的寄存器名字已被适当地替换。<6，11>和<8，14>都产生最低代价2。它们导致不同但等价的代码序列。因为它们有相同的代价，所以筛选器可以在它们中间随意选择。如预见的那样，其他序列有更高的代价。

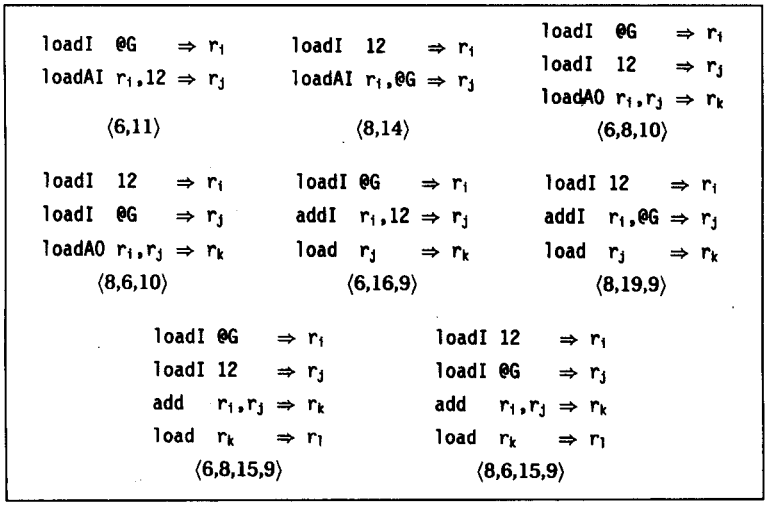


图11-8 上述匹配的代码序列

如果loadAI只接受有限范围内的参数，那么序列<8，14>可能无法工作，因为最终取代@G的地址对
564 于这一操作中的立即域来说可能太大。为了处理这种限制，编译器设计者可以把适当受限常量的概念引入到重写文法中。这可以采用一个新的终止符形式，这一终止符只表示给定范围内的整数，例如，对于12位的域，这可能是$0 \leqslant i < 4096$。利用这样的区别，以及通过检查整数的每一个实例来对其进行分类的代码，代码生成器可以回避序列<8，14>，除非@G落入loadAI的立即操作数的允许范围内。

代价模型操控代码生成器来筛选更好的序列。例如，注意序列<6，8，10>使用两个loadI操作，且其后跟随一个loadAO。代码生成器喜欢代价较低的序列，每一个这样的序列都避开一个loadI操作，并发行更少的操作。同样地，代价模型避开使用一个显式加法的四个序列，相反，更喜欢执行寻址硬件中的隐式加法。

11.3.2 寻找铺盖

为了把这些思想运用到代码生成中，我们需要能够构造好铺盖的算法，即构造一个产生高效代码的铺盖。给定一个规则集合，这一规则集合编码操作符树，并把它们与一个AST的结构相关联，代码生成器应该为特定的AST发现高效铺盖。构造这样的铺盖有若干有效技术。它们在概念上相似，但在细节上不同。

为了简化这一算法，我们做两个关于重写规则形式的假设。第一，每一个操作最多有两个操作数。扩展这一算法来处理一般情况是直截了当的，但是具体细节使我们的说明变得复杂。第二，规则的右部最多包含一个操作。这简化匹配算法，而又不失一般性。一个简单、机械式的过程可以把无限制情况转换成这种更简单的情况。对于产生式$\alpha \rightarrow op_1(\beta, op_2(\gamma,\delta))$，把它重写成$\alpha \rightarrow op_1(\beta, \alpha')$和$\alpha' \rightarrow op_2(\gamma, \delta)$，其中$\alpha'$是只在这两个规则中出现的新符号。这所引入的非终结符和产生式的数量是由原来的文法中出现的操作符的数目决定的。

为了使这一讨论更具体，考虑规则11，Reg→◆ (+(Reg_1, Num_1))。转换把它重写成Reg→◆ (R11P2) 和R11P2→+(Reg_1, Num_1)，其中R11P2是一个新符号。注意R11P2的新规则与描述addI的规则16相同。这一转换在文法中加入更多的歧义性。然而，独立地跟踪和匹配这两个规则使得模式匹配器考虑每一个规则的代价。取代规则11的规则应该有与原来规则相同的代价1。(每一个规则可以有小数代价，
565 或者它们中间的一个有零代价。) 这反映出使用规则16重写产生一个addI操作，而R11P2的规则却把加法叠入一个loadAI操作的地址生成中。当可能时，这一新规则通过利用上下文来特化匹配。

铺盖的目标就是对于每一个结点，使用编译器能用于实现这一结点的模式集合来对其赋标签。因为规则标号直接对应于右部的模式，所以代码生成器可以使用这些标号作为模式的缩写。编译器可以在对这棵树的后序遍历中为每一个结点计算规则标号序列，即模式序列。图11-9给出一个算法*Tile*的梗概，这一算法为AST中以结点*n*为根的树寻找一个铺盖。对于AST中的每一个结点*n*，这一算法使用*Label*(*n*)为其赋标签，其中*Label*(*n*) 由可以用于铺盖以结点*n*为根的树的所有规则标号组成。它在后序遍历中计算*Label*集合，以确保在为一个结点赋标签之前先为这个结点的子结点赋标签。

```
Tile(n)
  Label(n) ← ∅
  if n is a binary node then
    Tile(left(n))
    Tile(right(n))
    for each rule r that matches n's operation
      if left(r) ∈ Label(left(n)) and right(r) ∈ Label(right(n))
        then Label(n) ← Label(n) ∪ {r}
  else if n is a unary node then
    Tile(left(n))
    for each rule r that matches n's operation
      if left(r) ∈ Label(left(n))
        then Label(n) ← Label(n) ∪ {r}
  else /* n is a leaf */
    Label(n) ← { all rules that match the operation in n}
```

图11-9 计算铺盖AST的*Label*集合

考虑二元结点的内循环。为了计算*Label*(*n*)，这一内循环检查实现由*n*描述的操作的每一个规则*r*。它使用函数*left*和*right*来遍历AST和树模式（或者规则的右部）。⊖因为*Tile*已经为*n*的子结点赋了标签，它可以使用简单的关系测试把*r*的子结点与*n*的子结点对照。如果*left*(*r*)∈*Label*(*left*(*n*))，那么 566
*Tile*已经发现了它可以按与使用*r*实现*n*相适合的方式为*n*的左子树生成代码。同样的讨论也适用于*r*和*n*的右子树。如果两个子树都匹配，那么*r*属于*Label*(*n*)。

为了加速这一匹配过程，我们可以预计算所有可能的匹配，把它们存储在一个表中，并使用由结点*n*处的操作符和它的子结点的标签集合，为索引的表引用来替换二元结点和一元结点的内循环。这一标签集合是有界的。如果*R*是规则数量，那么$|Label(n)| < R$，且标签集合的数目不超过2^R。幸运的是，文法的结构划去很多这样的集合。如果每一个规则有一个运算符和两个子结点，那么查找表可以以这个操作符和两个子结点为索引。这导致一个大小为|操作树| × |标签集合|^{2}的表，这仍旧很大。对于拥有200个操作和带有1 000个标签集合的文法的机器，结果表有200 000 000个条目。幸运的是，为这一目的而构造的表是稀疏的，而且可以被高效地编码。（事实上，寻找有效地构建和编码这些表的方法是使得树模式匹配成为代码生成的实用工具的一个关键进步。）

寻找低代价的匹配

图11-9中的算法寻找模式集合中的所有匹配。事实上，我们是要代码生成器寻找最低代价的匹配。尽管它可以从所有匹配集合中得到最低代价匹配，但还是存在更高效的计算方法。

在自底向上遍历AST的过程中，代码生成器可以发现每一棵子树的最低代价匹配。自底向上的顺序保证代码生成器可以计算每一个可能匹配的代价，即匹配的规则代价加上相关的子树匹配的代价。原理

⊖ 回想一下，每一个规则描述一个操作符和至多两个子结点。因此，对于规则*r*，*left*(*r*) 和*right*(*r*) 都有明确的含义。

上看，它可以如图11-9所示的那样发现匹配，并保留最低代价匹配，而不是保留所有的匹配。在实践中，这一过程会更复杂一些。

代价函数本质上取决于目标处理器；不能从文法自动得到它。相反，它必须编码目标机器的性质并反映发生在汇编语言的各操作之间的相互作用，特别是从一个操作到另一个操作的值流的影响。

编译程序中的一个值可以有若干种不同的形式。各形式之间的差异构成指令筛选器的关键，因为这些差异将能够使用这个值的目标机器操作集合。在典型的机器上，一个值可以在寄存器中或在内存位置
567 中，或者它可能是一个编译时常量且足够小而适用于某些或全部立即操作。

当指令筛选器设法寻找特定子树的匹配时，它必须知道评估这一子树的每一个操作数的代价。如果这些操作数处于不同的存储分类中，如寄存器、内存单元或立即常量，那么代码生成器必须知道把这一操作数评估成各存储分类的代价。因此，它必须跟踪生成每一个存储分类的最低代价序列。当它做自底向上遍历来计算代价时，代码生成器可以轻松确定每一个存储分类的最低代价匹配。这只给这一过程增加少许工作量，但是这一增加量是由与存储分类数量相等的一个因数决定的，也就是整体上依赖于目标机器的一个数，而且它不依赖于重写规则的数量。

精心的实现可以在铺盖一棵树的同时累加这些代价。在每一次匹配时，如果代码生成器保留最低代价匹配，那么它将产生一个局部最优的铺盖。即，在每一个结点处，对于给定的规则集合和代价函数，不存在更好的选择。自底向上的代价累加为寻找最小代价铺盖提供一个动态规划解决方案。

如果我们要求代价是固定的，那么代价计算可以叠入模式匹配器的构建中。这把计算从编译时移到构建算法中，并几乎总是产生更快的代码生成器。如果我们允许代价发生变动且需考虑匹配时的上下文，那么代价比较必须在编译时完成。这也许会放慢代码生成器的速度，但是它使得代价函数有更多的灵活性，而且代价函数可以更精确地反映运行时行为。

11.3.3 工具

面向树、自底向上的代码生成方法导致高效指令筛选器。存在若干种编译器设计者实现基于这些原理的代码生成器的方法。

1）我们可以手工编写类似于Tile的匹配器，当这一匹配器铺盖树时显式地检查匹配规则。精心的实现可以限制对每一个结点必须检查的规则集合。这可避免较大的稀疏表并导致简洁的代码生成器。

2）因为这一问题是有限的，我们可以把它编码为一个有穷自动机，或树匹配自动机，并得到一个
568 DFA的最低代价行为。在这一方案中，查找表编码这一有穷自动机的转换函数，隐式地协同所有所需的状态信息。已有若干使用这一方法构建的系统，这些系统通常被称为自底向上重写系统（BURS）。

3）规则的文法形式表明可以使用分析技术。必须扩展分析算法来处理来自于机器描述的高度歧义性的文法并选择最小成本分析。

4）通过把树线性化成一个前缀串，这一问题可以转换成一个串匹配问题。然后，编译器可以使用串模式匹配算法寻找可能的匹配。

存在实现后三种方法的可用工具。编译器设计者生成目标机器指令集合的描述，而代码生成器生成器从这一描述创建可执行代码。

已自动化的工具在细节上是不同的。每一个被发行指令的代价随着技术的不同而不同。某些工具较快，某些工具较慢；但没有哪个工具足够慢使得它对结果编译器的速度产生重要的影响。这些方法允许不同的代价模型。某些系统限制编译器设计者对于每一个规则给予固定的代价；因此，它们可以在表生成期间执行某些或全部动态规划。而另外一些系统允许更一般的在匹配过程中可变的代价模型；这些系统在代码生成期间必须执行动态程序设计。然而，所有这些方法一般产生既有效又高效率的代码生成器。

11.4 指令筛选与窥孔优化

指令筛选核心的另一个执行匹配操作的技术是为后期阶段的优化而开发的称为*窥孔优化*（peephole optimization）的技术。为了避免代码生成器的复杂性，这一方法把低级IR上的系统局部优化与把这一IR与目标机器的操作匹配的简单方案结合起来。本节介绍窥孔优化，揭示其作为指令筛选的机制的使用，并描述已开发的用于自动化窥孔优化器结构的技术。 569

11.4.1 窥孔优化

窥孔优化背后的基本思想很简单：编译器可以通过检查邻近操作的短序列高效地发现局部改进。窥孔优化器最初是为了在编译的所有其他步骤完成之后运行而被提出的。它既消耗又生成汇编代码。这一优化器有一个滑动窗口，即“窥孔”，它可以在代码中移动。在每一步，窥孔优化器检查窗口内的操作，寻找它可以改进的特定模式。当它识别出一个模式时，它使用更好的指令序列重写这一模式。一个有限模式集合与一个关注的限定区域的结合导致快速的处理。

一个典型的模式示例是在存储之后跟随同一位置的装入。这个装入可以用一个拷贝来替换。

```
storeAI r1     ⇒ rarp,8          storeAI r1 ⇒ rarp,8
loadAI  rarp,8 ⇒ r15      ⇒      i2i     r1 ⇒ r15
```

如果窥孔优化器识别出这一重写使得存储操作死亡（也就是说，load只是为了存储于内存中的这个值而使用的），那么它还可能消除这一存储操作。然而，识别死亡存储一般需要超出窥孔优化器的作用域范围的全局分析。窥孔优化可以改进的其他模式包括简单的代数等式，例如，

```
addI r2,0 ⇒ r7
mult r4,r7 ⇒ r10    ⇒    mult r4,r2 ⇒ r10
```

以及目标本身是一个分支的分支

```
    jumpI → l10          jumpI → l11
l10: jumpI → l11   ⇒   l10: jumpI → l11
```

如果这消除最后到l_{10}的分支，那么开始于l_{10}的基本模块变成不可达且被消除。遗憾的是，对在l_{10}处的操作不可达的证明所花费的分析比窥孔优化期间的一般可行的分析要多（参见10.3.1节）。

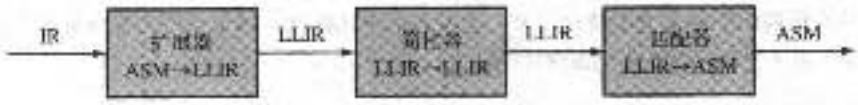
早期的窥孔优化器使用一个手工编码的模式的有穷集合。它们使用穷举搜索来匹配这一模式，但却可以快速运行，因为模式数量较小且有较小的窗口，一般地窗口中只有两到三个操作。 570

四元组上的树模式匹配

用于描述这些技术的术语，如*树模式匹配*和*窥孔优化*，隐含着可以用于这些技术的IR种类的假设。BURS理论研究树上的重写操作。这带来这样的一种印象，基于BURS的代码生成器需要树形IR。同样地，窥孔优化器最初是作为最后的汇编到汇编改进遍而提出的。移动指令窗口的思想暗示基于窥孔的代码生成器需要线性、低级的IR。

两个技术都适用于大多数IR。编译器可以把ILOC等的低级线性IR解释为树。每一个操作变成一个树结点；操作数的复用表示边。同样地，如果编译器为每一个结点赋一个名字，那么它能够通过执行后序遍历把树解释成为线性形式。聪明的实现器可以把在本章所给出的方法用于现实各种IR形式。

窥孔优化的进步已超出了匹配少数几种模式。复杂ISA的增加导致更系统化的方法。现代窥孔优化器把这一过程分解成三个不同的任务：扩展、简化和匹配。它系统地运用符号解释和简化来取代早前系统的模式驱动优化。



上图从结构上看起来像是一个编译器。扩展器识别IR形式的输入代码，并构建一个内部表示。简化器在那个IR上执行某些重写操作。匹配器把这一IR转换成目标机器代码，一般是汇编代码（ASM）。如果输入和输出语言相同，那么这一系统是一个窥孔优化器。输入和输出代码使用不同的语言时，相同的算法可以执行指令错选，如我们将在11.4.2节看到的那样。

扩展器逐个操作地把汇编代码重写成表示一个操作的所有直接效应的低级IR操作（LLIR）的序列。
571 这些效应至少包括影响程序行为的所有效应。如果操作add r_1，r_2⇒r_3设置条件代码，那么它的LLIR表示必须包含把r_1+r_2赋值给r_3的操作以及把条件代码设置为适当值的操作。扩展器一般具有简单结构。无需考虑上下文就可以逐一扩展操作。这一过程为每一个ASM操作使用一个模板，并适当替换这一模板内的寄存器名字、常量和标签。

简化器在整个LLIR上做一遍处理，检查LLIR上的一个小窗口内的操作，并设法系统地改进它们。简化的基本机制是前向替换、代数化简（例如，x+0⇒x）、评估常量值表达式（例如，2+17⇒19）以及消除诸如无用条件代码生成等无用效果。因此，简化器在这一窗口内对LLIR执行有限局部优化。这必须使在LLIR中揭示出的所有细节（地址算术、分支目标等等）服从于局部优化的标准。

在最后一步，匹配器参照模式库比较简化了的LLIR，寻找能够最好地刻画这一LLIR内的所有效应的模式。最终的代码序列可能产生超出LLIR序列所要求的效应，例如，它可能创建新的、尽管也许无用的条件代码值。然而，它必须保持正确性所需要的效应，它不能消除一个活着的值，无论这个值是被存储于内存中、寄存器中还是诸如条件代码等隐式设置的位置中。

图11-10给出这一方法是如何在11.2节的例子上工作的。在左上方，它开始于如图11-4所示的低级AST的四元组。（回想一下，这一AST计算w← x-2×y，其中w被存储在局部AR中偏移为4的地方，x作为一个引用参数被存储，这一参数的指针存储于距ARP偏移为+16的地方，y则存储于距标签@G偏移为12的地方。）扩展器创建如图中右上方所示的LLIR。简化器化简这一代码，生成图中右下方的LLIR代码。根据这一LLIR片段，匹配器构造出左下方的ILOC代码。

理解这一过程的关键在于简化器。图11-11给出窥孔优化器在处理本例中的低级IR时的窗口内的相继序列。假设这一窥孔优化器有三操作窗口。序列1显示带有前三个操作的窗口。没有化简的可能。优化器把定义r_{13}的第一个操作移出这一窗口，并带进r_{13}的定义。在这一窗口中，它向前把r_{13}替换成r_{12}的定义。因为这使r_{12}死亡，优化器忽视r_{12}的定义，并把另外一个操作加入这一窗口的底部。达到序列3。接下来，它把r_{13}叠入定义r_{14}的内存引用中，产生序列4。

序列4没有化简的可能，所以优化器把r_{11}的定义移出这一窗口。优化器不能化简序列5，所以它也把r_{14}的定义移出这一窗口。它能够通过向前将－16替换成定义r_{15}的加法来化简序列6。这一动作产生序列7。优化器继续这一行为，在可能时化简代码，当它不能化简代码时就前进。当达到序列13时，它停止，因为它不能进一步化简这一序列，而且它没有额外的代码可以带到这一窗口中。

回到图11-10，比较化简后的代码和源代码。化简后的代码是由离开这一窗口的顶部的那些操作，再加上当化简停止时留在这一窗口内的那些操作组成的。化简后，计算需要8个操作，而不是14个操作。它使用7个寄存器（r_{fp}除外），而不是13个寄存器。

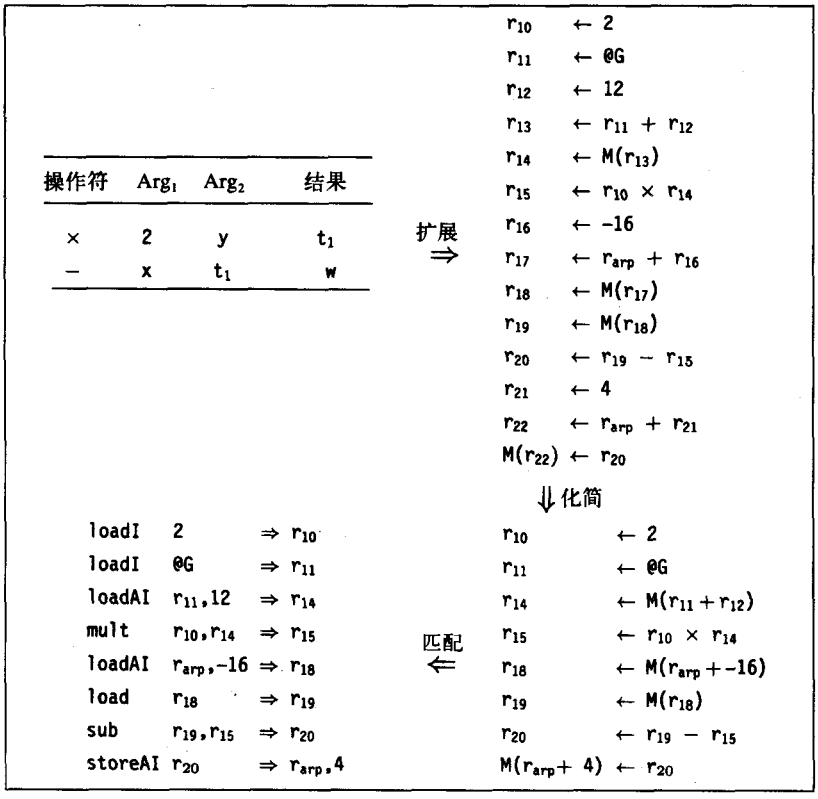


图11-10 对例子的扩展、化简和匹配

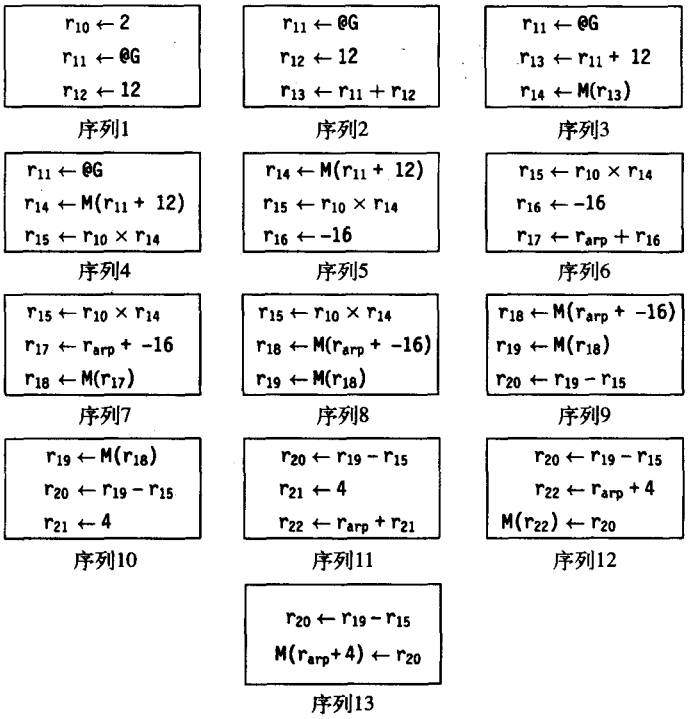


图11-11 化简器产生的序列

572 ~ 574

若干设计问题影响窥孔优化器改进代码的能力。检查一个值的死亡时间的能力在化简中起着重要的作用。控制流操作的处理决定在模块边界将发生的事情。窥孔窗口的大小限制优化器组合相关操作的能力。例如，更大的窗口将使得简化器把常量2叠入乘法操作中。下面3个小节探讨这些问题。

1. 死亡值

识别出表示为死亡值的无用效果的能力是这一过程的关键。回想一下，在代码的某个点p一个值是死亡的，如果不存在从p到这个值的任意使用的路径。遗憾的是，确定一个值的死亡一般需要对创建这个值的整个作用域的分析。那么，简化器是如何知道当它关注代码的一个小窗口时某些值是死亡的呢？

在扩展过程期间，优化器可以为每一个操作构造死亡变量表。下面两个观察可以实现这一点。第一，扩展器可以向后遍历代码来完成它的工作。当它遍历每一个块时，在处理前驱之前处理CFG的后继（当然，是自后继之外的后继），它可以建立一个死亡变量表。第二，大部分重要的死亡值是纯局部值，即在单一模块内定义并使用它们。这些值可以作为代码形态的内容来识别，编译器可以为这些值使用一个独立的名字或寄存器编号区域，或者优化器可以放置它能够发现的这些值。

需要被跟踪的频繁死亡的值的典型例子是条件代码。与ILOC的虚拟机器形成对比，在实际机器的描述中，会出现诸如条件代码这样的详细情况。某些计算机使用算术操作来设置条件代码；如果一个模块结束于对计算的值是否等于零的测试，那么可以通过使用前面计算的条件代码来避开这一比较。很自然，扩展器必须包含这些条件代码的效应。然而，除非死亡代码可以被检查并消除，否则这些不相关的条件代码赋值可能妨碍窥孔优化器组合那些它本能够组合的操作。

575 例如，考虑计算$r_i \times r_j + r_k$，如果×和+都设置条件代码，那么这一两操作序列可能生成如下的LLIR。

$$
\begin{aligned}
r_{t1} &\leftarrow r_i \times r_j \\
cc &\leftarrow f_x(r_x, r_g) \\
r_{t2} &\leftarrow r_{t1} + r_k \\
cc &\leftarrow f_+(r_{t1}, r_k)
\end{aligned}
$$

因为加法跟在乘法的后面，所以存储于条件代码内的第一个值是死的。

如果简化器消除为乘法$cc \leftarrow f_x(r_i, r_j)$设置条件代码的操作，那么它可以把余下的三个操作组合成一个乘-加操作，假设目标机器有这样的指令。然而，如果它不能消除$cc \leftarrow f_x(r_i, r_j)$，这将妨碍匹配器使用乘-加操作。

2. 控制流操作

控制流操作的存在使简化器变得复杂。处理这些控制流操作的最容易方法是当简化器达到分支、跳转或带标签指令时清除它的窗口。这防止简化器把效应移到效应不曾存在的路径上。

简化器可以通过检查分支周围的上下文达到更好的结果，但是它将给这一过程带来若干特殊的情况。如果输入语言编码带有单一目标和落下路径的分支，那么简化器应该跟踪并消除死亡的标签。如果它消除一个标签的最后使用而且前面的块有一个落下出口，那么它可以消除这个标签，组合这些块，并跨越老边界进行化简。如果输入语言编码带有两个目标的分支，或者前面的块结束于一个跳转，那么一个死亡标签意味着可以被完全消除的一个不可达块。无论哪种情况，简化器都应该跟踪每一个标签的使用数量，并消除不再被引用的标签。（扩展器可以计数标签的引用，以允许简化器使用简单的引用计数方案来跟踪余下的引用数量。）

一个更具吸引力的方法可能考虑一个在一个分支的两端的操作。某些化简有可能跨越这一分支，把这一分支的前一个操作的效应与这一分支目标处的那些操作的效应组合起来。然而，简化器必须考虑达到这个带标签操作的所有路径。

谓词操作需要某些相同的考虑。在运行时，谓词值决定哪些操作真的执行。事实上，谓词指定一条 576
经过一个简单CFG的路径，尽管这条路径可能是一条没有显式标签或分支的路径。简化器必须识别出这些效应并以处理带标签操作时使用的方式来处理它们。

3. 物理窗口和逻辑窗口

迄今为止，我们的讨论都集中在包含低级IR中邻近操作的窗口。这一概念有很好的物理直观性，而且使这一概念具体化。然而，低级IR中的邻近操作可能不是操作于相同值上。事实上，随着目标机器提供更多指令级并行，编译器的前端和优化器必须生成具有更多独立和交替计算的IR程序，以保持目标机器的功能单元忙碌。在这种情况下，窥孔优化器几乎不可能发现改进代码的机会。

为了改进这一状况，窥孔优化器可以使用逻辑窗口，而不是物理窗口。使用逻辑窗口，优化器考虑被这一代码内的值流连结起来的操作，也就是说，它一并考虑定义和使用相同值的操作。这创建组合及化简相关操作的机会，即使在这些操作之间存在其他的操作。

在扩展期间，优化器可以把每一个定义与在相同模块中这一定义的值的下一次使用链接起来。这使得简化器快速地把逻辑对放在一起。当优化器遇到操作i时，它设法在考虑i的逻辑后继的前提下化简这一操作，i的逻辑后继是块中下一次使用由i定义的值的操作。（因为化简在很大程度上依赖于向前替换，所以很少有理由考虑下一个物理操作，除非它使用i的结果。）在一个块内使用逻辑窗口可以使简化器更有效，同时减少所需的编译时间和化简后所余下的操作数量。在我们的例子中，逻辑窗口将使简化器把常量2叠入乘法中。

把这一想法扩展到更大的作用域会增加一些复杂性。编译器可以设法化简逻辑上邻近但又相距太远以致难以一同放入窥孔窗口内的操作：或者在同一个块内或者在不同的块内。这需要一个全局分析来确定每一个定义可以达到的使用（也就是说，根据9.2.4节的可达定义）。另外，简化器必须识别单一定义可能达到多个使用，而且单一使用可能引用由几个不同的定义计算而得的值。因此，简化器不能简单地把定义操作与一个使用组合起来，并留下余下的操作不管。它必须或者只考虑简单的情况，例如，单一定义和单一使用，或者带有单一定义的多次使用，或者它必须执行某些仔细的分析以确定一个组合是否 577
是既安全又有效益。这些复杂性表明在局部或超局部上下文中运用逻辑窗口。把逻辑窗口移到超出一个基本块的范围给简化器带来错综复杂的情况。

11.4.2 窥孔转换器

正如前节所述的那样，更多系统的窥孔优化器的出现带来了对目标机器的汇编语言的更完备的模式集合的需求。因为三步过程把所有操作翻译成LLIR并设法化简所有LLIR序列，匹配器需要有把任意LLIR序列翻译回目标机器的汇编代码的能力。因此，这些现代窥孔系统比早前系统有更大的模式库。随着计算机从16位指令转移到32位指令，不同汇编操作数量的爆增使得手工生成模式成为问题。为了处理这种爆增，大多数现代窥孔系统包含从目标机器的指令集合的描述自动生成匹配器的工具。

生成描述处理器指令集合所需的巨大模式库工具的出现，已使得窥孔优化成为指令筛选的一个具有竞争力的技术。一个最终的转变进一步化简这一情景。如果前端直接生成用于窥孔优化器的LLIR，那么编译器不再需要扩展器。同样地，化简器也无需处理无用效应。像设置一个无关的条件代码这样的效应的出现是因为扩展器缺少足够多的上下文来确定它们是否有用。这带来对扩展器计算死亡值列表的需求以及对简化器对它们进行计数的需求。如果前端直接生成LLIR，那么它也能生成它需要的效应并消除那些它不需要的效应。

这一方案还将简化把编译器的目标重设到另一个处理器时所需的工作。为了改变处理器，编译器设计者必须① 为模式生成器提供适当的机器描述，使得它能生成新的指令筛选器，② 改变由早前阶段生

成的LLIR序列，使得这些序列适合新的ISA，③ 修改指令调度器和寄存器分配器，以便反映新ISA的特性。尽管这包含大量的工作，但是描述、处理和改进LLIR序列的基础结构仍保持不变。另外，从根本上
578 不同的机器的LLIR序列必须刻画它们之间的差异；然而，编写这些序列的基础语言仍保持相同。这允许编译器设计者构建跨体系结构的一组工具，并且允许编译设计者通过为目标ISA生成适当低级IR并为窥孔优器提供一组适当的模式来产生机器特定的编译器。

这一方案的其他优势在于简化器。这一被拆分的窥孔转换器仍然包含简化器。代码的系统化简，即使是在限定的窗口内执行，也将比使用简单的手工编码遍历IR并将IR重写成汇编语言提供更有意义的优势。向前替换、简单算术等式的运用以及常量叠入可以产生更短、更高效的LLIR序列。而这些又将导致更好的目标机器代码。

RISC、CISC以及指令筛选

RISC体系结构的早期支持者认为RISC将导致更简单的编译器。像IBM 801这样的早期RISC机器拥有的寻址模式比同时代的CISC机器（如DEC的VAX-11）少得多。它们以寄存器到寄存器操作为特色，并具有把数据在寄存器和内存之间移动的装入和存储操作。与此相对照，VAX-11既提供寄存器操作数，又提供内存操作数；许多操作既支持二地址形式，又支持三地址形式。

RISC机器简化了指令筛选。它们为实现给定操作提供更少的方式。它们对寄存器的使用的限制更少。然而，它们的装入-存储结构增加寄存器分配的重要性。

与此相对照，CISC机器拥有把更多复杂功能封装到单一操作的操作。为了有效利用这些操作，指令筛选器必须在更大的代码片段中识别更大的模式。这增加系统化指令筛选的重要性；本章所述的自动技术对CISC机器来说更为重要，但是它们也同样可以运用于RISC机器。

若干重要的编译器系统已经在使用这一方法。最著名的编译器可能是Gnu编译器系统GCC。它的前端生成称为寄存器转移语言（register-transfer language，RTL）的低级IR。优化遍处理RTL来产生改良的RTL。后端消耗RTL并使用窥孔方案产生各个不同目标计算机的汇编代码。简化器是使用系统的符号
579 解释来实现的。窥孔优化器的匹配步骤实际上把RTL代码解释成为树，它使用从目标机器的描述而构建起来的简单树模式匹配器。诸如Davidson的VPO等其他系统把机器描述转换成文法，并生成以线性形式处理RTL的小型分析器来执行匹配步骤。

11.5 高级话题

基于BURS和基于窥孔的指令筛选器都是为编译时效率而设计的。然而，这两个技术都受到编译器设计者提供的模式中所包含的信息的限制。为了找到最好的指令序列，编译器设计者可以考虑使用搜索。这一想法很简单。有时候指令组合可以产生令人惊讶的效应。因为其结果是不可预料的，所以编译器设计者很少预测它们，因此也不把它们加入为目标机器所生成的描述中。

在文献中，有两个把穷举搜索用于改进指令筛选的不同方法。第一个方法设计基于窥孔的系统，这一方法在编译代码时发现并优化新模式。第二个方法涉及可能指令空间的蛮力搜索。

11.5.1 学习窥孔模式

在实现或使用窥孔优化器中所引发的一个重要问题是，描述目标机器指令集合所花费的时间与结果优化器或指令筛选器的质量和速度之间的权衡问题。使用完备的模式集合，化简和匹配的成本可以通过使用高效的模式匹配技术而维持到最小。当然，必须有人来生成所有这些模式。另一方面，在化简和匹

配期间解释规则的系统对每一个LLIR操作都有更大的负荷。这样的系统可以使用较小的规则集合来进行操作。这使得系统易于构建。然而，结果简化器和匹配器运行得更慢。

生成快速模式匹配窥孔优化器所需的显示模式表的一个高效方法是，把这一优化器与一个拥有符号简化器的优化器配对。在这一方案中，符号简化器记录它化简的所有模式。它每次化简一对操作时，便 580
记录初始对和化简了的对。⊖然后，它可以把结果模式记录在查找表中来生成快速模式匹配优化器。

通过在一个应用的训练集上运行符号简化器，优化器能够发现它所需的大部分模式。然后，编译器可以使用这个表作为快速模式匹配优化器的基础。这使得编译器设计者在设计期间消耗计算机时间来加速编译器的常规使用。它极大地降低必须描述的模式的复杂性。

两个优化器之间的相互影响的增加可以进一步改进代码质量。在编译时，快速模式匹配器将遇到在其表上没有匹配模式的LLIR对。当这一情况发生时，它能够调用符号简化器来寻找一个改进，这使得简化器只需在没有已存在模式的LLIR对上进行搜索。

为了使这一方法实用，符号简化器应该既记录成功又记录失败。这将允许它在没有符号解释负荷的情况下拒绝先前所看到LLIR对。当它成功改进一个对时，它应该把新的模式加入优化器模式表中，使得这个对以后的实例可以由更高效的机制来处理。

生成模式的这一学习方法有几个优势。它只倾力于前面没有看到的LLIR对。它填补目标机器的训练集合的覆盖范围中的空洞。在保持大多数模式导向系统的速度同时，它提供高成本系统所提供的完整功能。

然而，在使用这一方法时，编译器设计者必须决定符号优化器什么时候更新模式表以及如何调节这些更新。允许任意编译重写所有用户的模式表似乎是不明智的；这必定会引发同步和安全的问题。相反，编译器设计者可以选择周期性的更新，把新发现的模式存储起来，使得它们能够作为例行的维护动作加入表中。

11.5.2 生成指令序列

学习方法具有固有的偏见；它假设低级模式应该引导搜索等价的指令序列。某些编译器对于相同的基础问题采用穷举方法。取代设法从低级模型合成理想的指令序列，它们采用生成和测试方法。 581

想法很简单。编译器或编译器设计者识别一个应该得到改进的汇编语言指令的短序列。然后编译器生成所有代价为1的汇编语言序列，把原来的参数替换成这一生成的序列。它测试每一个参数来确定它是否与目标序列有相同的效应。当它耗尽了给定代价的所有序列时，它增加序列的代价并继续。这一过程一直持续，直到（1）它发现一个等价序列，（2）它达到原来的目标序列的代价，或者（3）它达到对代价或编译时的外部限制。

尽管这一方法本质上代价高昂，但是用于测试等价性的机制对测试每一个候选序列所需的时间有很强的影响。使用机器效应的低级模型，我们需要筛选出微妙的不正确匹配的形式方法，但是一个更快的测试可以捕获那些经常出现的不正确匹配。如果编译器简单地生成并执行这一候选序列，那么它可以把这一结果与得自于目标序列的结果相比较。运用于若干精选的输入，这一简单的方法应该以低成本测试消除大部分不适用的候选序列。

显然，这一方法成本太高，无法例行地使用或运用于较大的代码片段。然而，在某些情况下，它值得考虑。如果应用设计者或编译器可以识别代码中既小又对性能起决定因素的片段，那么优秀代码序列的成效可以证实穷举搜索的代价的正当性。例如，在某些嵌入式应用中，对性能起决定性因素的代码是

⊖ 符号化简器可能需要参照机器描述检查已提出的模式，以确保这一简化不过于一般。例如，如果它依赖于一个操作数的特殊常量值，那么输入模式必须编码这一限制。

由一个内循环组成的。为了改进速度或空间，使用穷举搜索寻找小的代码片段也许是值得的。

同样地，穷举搜索已用于把编译器的目标重新设定到新的体系结构的过程中。这一应用使用穷举搜索为编译器反复生成的IR序列发现特别高效的实现。因为当编译器被移植时，代价被带到移植后的编译器，所以编译器设计者可以在使用新编译器的很多编译上摊派成本来证实搜索的正当性。

11.6 概括和展望

就其核心来说，指令筛选是模式匹配问题。这一问题的难易程度依赖于编译器IR的抽象层次、目标
582 机器的复杂性以及对编译质量的期望。在某些情况下，简单树遍历方法将产生足够的结果。然而，对于这一问题的更加困难的实例，由树模式匹配或者窥孔优化引导的系统搜索可能产生更好的结果。创建能达到相同结果的手工树遍历代码生成器将需要更多的工作。尽管这两个方法在绝大部分的细节部分是不同的，但是它们享有一个共同的观念：运用模式匹配来在任意给定IR程序的各个可能的序列中寻找好的代码序列。

通过在每一个决策点采用低成本选择，树模式匹配器发现低成本铺盖。其结果代码实现由IR程序所描述的计算。窥孔转换器系统地化简IR程序并参照目标机器的模式集合匹配剩余的一切。因为它们缺乏显式代价模型，所以没有关于最优的论据。它们生成与IR程序有相同效应的计算代码，而不是IR程序的文字实现。由于这两个方法之间的这种细微差异，我们不能直接比较它们的行为。在实践中，每一种方法都可以得到优秀的结果。

这些技术的实际效益已在真实的编译器中得到展示。LCC和GCC都可以在很多平台上执行。前者使用树模式匹配；而后者使用窥孔转换器。在这两个系统中的自动工具的使用已使它们易于理解、易于重新设定目标并最终得到广泛的接受。

同样重要的是，读者应该认识到自动模式匹配器系列可以用于编译的其他问题。窥孔优化原本是作为改进编译器所生成的最终代码的技术。同样地，编译器可以运用树模式匹配来识别并重写AST中的计算。BURS技术可以提供特别有效的方法来识别并改进简单模式，包括由值编号识别出的代数等式。

本章注释

大多数早期编译器使用手工编码、专门的技术来执行指令筛选[25]。对于足够小的指令集合或足够大的编译器设计组来说，这是可行的。例如，BLISS-11编译器使用有限操作指令表为PDP-11生成优秀的代码[339]。早期计算机和小型计算机的小指令集合使研究者和编译器设计者忽视在现代机器上出现的
583 某些问题。

例如，Sethi和Ullman[301]以及后来的Aho和Johnson[5]考虑了为表达式树生成最优代码的问题。Aho、Johnson和Ullman把他们的思想扩展到表达式DAG[6]。基于这一工作的编译器为控制结构使用专门的方法，并为表达式树使用聪明的算法。

在20世纪70年代后期，体系结构中的两个不同趋势把指令筛选的问题推向编译器研究的前沿。从16位到32位体系结构的转移促成编译器必须考虑的操作和地址模型的数量的爆增。对于揭示更大可能性的编译器，它需要更加形式化且更强大的方法。与此同时，出色的Unix操作系统开始出现在多个平台上。这激起对C语言编译器的自然需求，并增加了对可重设目标编译器的兴趣[196]。指令筛选器能够轻松地重设目标的能力，在决定把编译器移植到新体系结构的轻松度上起着重要的作用。这两种倾向引发了始于20世纪70年代的关于指令筛选的丰富研究成果，并在进入20世纪90年代后仍得到良好的继续[155，122，67，161，66，276，277]。

自动扫描和分析的成功使得描述驱动指令筛选成为一个有吸引力的想法。Glanville和Graham把指令

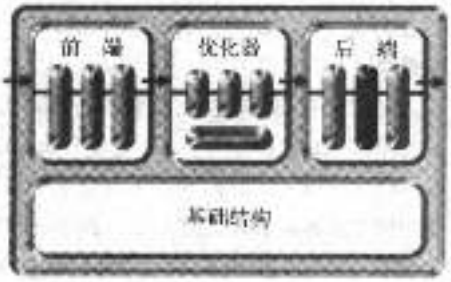
筛选的模式匹配映射到表驱动分析[155，160，162]。Ganapathi和Fischer使用属性算法对这一问题发起了进攻[151]。

树模式匹配代码生成器源于表驱动代码生成的早期工作[162，40，230，10，175]，以及树模式匹配的早期工作[182，71]。Pelegrí Llopart在自底向上重写系统（BURS）中形式化了很多概念[271]。后来的作者在这一工作的基础上构建了一系列实现、变形和表生成算法[147，146，276]。末梢（twig）系统把树模式匹配和动态规划结合到了一起[322，2]。

第一个窥孔优化器出现于McKeeman的系统[253]。Bagwell[28]、Wulf等人[339]和Lamb[227]描述了早期的窥孔系统。11.4.1节中所描述的扩展、化简和匹配循环来自于Davidson的工作[107，110]。Kessler也对直接从目标体系结构的低级目标描述得到窥孔优化器方面进行了研究[211]。Fraser和Wendt把窥孔优化用于执行代码生成[149，150]。11.5.1节所描述的机器学习方法是由Davidson和Fraser所描述的[108]。

Massalin提出了11.5.2节所描述的穷举方法[251]。Granlund和Kenner把这一方法部分运用于GCC[165]。

584



第12章
指令调度

12.1 概述

为了执行而呈现的操作的顺序，对执行这些操作序列所花费的时间长短有着重要的影响。不同的操作花费的时间长短不同。内存系统可能需要花费多于一个周期的时间来把操作数传送到寄存器单元或功能单元。功能单元本身也可能要花费几个执行周期来传送操作的结果。如果一个操作在一个结果准备就绪之前试图引用它，那么处理器一般都要推迟这一操作的执行，直到这个值准备就绪。也就是说，它停顿（stall）。用于某些处理器的另外一种选择是假设编译器可以预测这些停顿并重排序操作以避免这些停顿。如果不能插入有用的操作来推迟一个操作，那么编译器必须插入一个或多个nop。对于前者，对这一操作结果的过早引用将降低性能。而对于后者，硬件假设这一问题从不发生，所以当它发生时，计算产生不正确的结果。编译器应该设法对指令进行排序来避免第一个问题。它必须避免第二个问题。

很多处理器可以在每一个周期开始多个操作的执行。这些操作为了执行而出现的顺序可以决定在一个周期启动、或发行（issue）的操作数量。例如，考虑一个简单的处理器，它带有一个整数功能单元和一个浮点功能单元，以及一个由100个整数操作和100个浮点操作组成的已编译循环。如果编译器对这些操作进行安排，使得前75个操作是整数操作，那么浮点单元将保持空闲，直到处理器最终达到为它做某些工作为止。如果所有这些操作都是独立的（这是一个不真实的假设），那么最好的顺序将是在两个单元之间交替操作。

非形式地，指令调度是一个过程，编译器在这一过程中对已编译代码中的操作进行重排序以期望减少它的执行时间。概念上，一个指令调度器看起来呈如下形式：



指令调度器以指令的输序列表为输入；它产生由相同的操作集合构造而来的指令列表为输出。调度器假设有一个固定的操作集合；与寄存器分配器可能加入溢出代码不同，它不加入操作。调度器假设一个固定的名字空间；它不改变存储在寄存器中的值，尽管它可能对某些特殊的值重命名以便消除寄存器冲突。调度器通过改变操作序列来表示它的结果。

指令调度器的主要目标是保持作为输入而接受的代码的意义，且通过避免耗费在互锁和停顿中的浪费周期来最小化执行时间，以及避免由于变量的生存期的增加而引入额外的寄存器溢出。当然，这一调度器应该高效地操作。

如果处理器能够在单一周期发行多个操作，那么它必须拥有决定发行多少个操作的机制。在一台超长指令字（VLIW）机器上，处理器在每一个周期对于每一个功能单元发行一个操作，所有操作被收集到一个固定格式的指令中，一个指令在每一个槽中要么包含一个有用操作要么包含一个nop。压缩VLIW机器通过使用可变长度指令来避免其中的很多nop；指令流包含让处理器在提取和解码的过程中发现指

令边界的信息。

超标量处理器通过查看指令流中后k个操作来决定发行的操作数量，其中k是固定的小整数。(k一般等于或稍大于功能单元的数量。) 处理器按顺序检查每一个操作，并在可能时发行它。当处理器遇到由于资源的限制或操作数的可用性而不能发行的操作时，它停止发行操作并等待下一个周期。无序超标量 586
处理器查看指令流上的一个更大的窗口。它尽可能发行更多的操作，跳过在当前周期不能发行的操作。

机制的多样性使操作和指令之间的差异变得含混不清。在VLIW和压缩VLIW机器上，一个指令包含多个操作。在超标量机器上，我们通常把一个操作作为一个指令，并把这些机器描述为在每一个周期发行多个指令。本书中，我们一直在使用术语*操作*（operation）来描述编译器希望在单一功能单元执行的一个操作码和它的操作数。调度的讨论使用相同的术语。调度器对操作进行重排序。按照传统，我们仍称这一问题为*指令调度*（instruction scheduling），尽管称为*操作调度*（operation scheduling）会更精确。在VLIW或压缩VLIW体系结构上，调度器把操作包装成在给定的周期执行的指令。在超标量体系结构的机器上，无论是有序还是无序，调度器对操作进行重排序，使得处理器在一个周期尽可能多地发行操作。

本章研究调度以及编译器用于执行调度的工具和技术。以下几个小节给出讨论调度和理解算法及其影响所需要的背景信息。

12.2 指令调度问题

回想一下1.5节给出的指令调度的例子。图12-1再次生成它。标签“start”的那一列给出每一个操作开始执行的周期。假设处理器有单一的功能单元；内存操作需要三个周期；`mult`需要两个周期；所有其他操作都在一个周期内完成。在这些假设下，图左边所示的源代码需要20个周期。

图右边所示的调度后的代码要快得多。它把长等待时间操作从引用它们的结果的操作中分离出来。这一分离使得不依赖于这些结果的操作与它们同时执行。这一代码在前三个周期发行装入操作；其结果分别在周期4、5、6可用。这一调度需要一个额外的寄存器r_3来保存第三个同时执行的装入操作的结果，但是它允许处理器在等待第一个算法操作数到来时执行有用的工作。各操作之间的交迭有效地隐藏了内存操作的等待时间。在这个块中运用的相同思想隐藏了`mult`操作的等待时间。重排序把运行时间减少到13个周期，得到35%的改进。 587

并非所有块都适应于这种方式的改进。例如，考虑下面计算x^8的块：

开始		
1	loadAI	r_{arp},@x ⇒ r_1
4	mult	r_1,r_1 ⇒ r_1
6	mult	r_1,r_1 ⇒ r_1
8	mult	r_1,r_1 ⇒ r_1
10	storeAI	r_1 ⇒ r_{arp},@x

三个`mult`操作有长等待时间。遗憾的是，每一个指令都使用前一个指令的结果。因此，调度器对于改进这一代码没有什么可做的，因为它没有在`mult`执行的同时可以发行的独立指令。因为它缺少可以并行执行的独立操作，所以我们说这个块没有*指令级并行*（instruction-level parallelism，ILP）。给定充分的ILP，调度器可以隐藏内存等待时间和功能单元等待时间。

迄今为止我们所看到的所有例子都是隐式地考虑在每一个周期发行一个操作的目标机器。在这样一台机器上，操作和指令相同，因为每一个指令只含有一个操作。几乎所有现代计算机都有多个功能单元 588

以及在每一个周期发行若干操作的能力。我们将介绍单一发行机器的列表调度算法，并指出为了处理多操作指令而扩展这一基础算法所必须做的事情。我们继续使用术语操作来表示被发行到特定功能单元的单一操作码，且只有当设计在一个周期发行的所有操作的集合时使用指令这一词汇。在这样的观点下，称这一过程为操作调度也许更精确；然而，按照传统，我们继续称这一过程为指令调度。

为了更形式地定义调度，我们必须定义一个块的相关图（dependence graph）$D=(N, E)$，有时候称为这个块的优先图（precedence graph）。每一个结点$n \in N$是输入代码片段中的一个操作。存在边$e=\langle n_1, n_2\rangle \in E$，当且仅当$n_2$使用$n_1$的结果作为操作数。每一结点除了边之外，还有两个属性：类型（type）和等待（delay）。对于一个结点n，对应于n的操作必须在类型$type(n)$的功能单元上执行，而且完成它需要$delay(n)$个周期。图12-1的例子产生图12-2所示的相关图。

开始	源代码			开始	源代码		
1	loadAI	r_{arp}, @w	$\Rightarrow r_1$	1	loadAI	r_{arp}, @w	$\Rightarrow r_1$
4	add	r_1, r_1	$\Rightarrow r_1$	2	loadAI	r_{arp}, @x	$\Rightarrow r_2$
5	loadAI	r_{arp}, @x	$\Rightarrow r_2$	3	loadAI	r_{arp}, @y	$\Rightarrow r_3$
8	mult	r_1, r_2	$\Rightarrow r_1$	4	add	r_1, r_1	$\Rightarrow r_2$
9	loadAI	r_{arp}, @y	$\Rightarrow r_2$	5	mult	r_1, r_2	$\Rightarrow r_1$
12	mult	r_1, r_2	$\Rightarrow r_1$	6	loadAI	r_{arp}, @z	$\Rightarrow r_2$
13	loadAI	r_{arp}, @z	$\Rightarrow r_2$	7	mult	r_1, r_3	$\Rightarrow r_1$
16	mult	r_1, r_2	$\Rightarrow r_1$	9	mult	r_1, r_2	$\Rightarrow r_1$
18	storeAI	r_1	$\Rightarrow r_{arp}$,0	11	storeAI	r_1	$\Rightarrow r_{arp}$,0

图12-1 概述中的调度例子

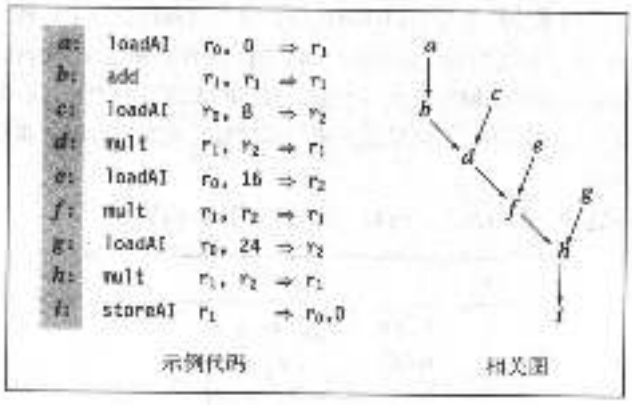


图12-2 例子的相关图

在D中没有前驱的结点，例如本例中的a、c、e和g，称为这个相关图的叶子（leave）。因为这些叶子不依赖于其他操作，所以它们可以尽早地被调度。在D中没有后继的结点，例如本例中的结点i，被称为此图的根（root）。一个相关图可以有多个根。在某种意义下，这些根是相关图中最受限制的结点，因为在它们的祖先全都执行完毕之前，它们不能执行。使用这一术语，就如我们是颠倒着画出D，至少相对于早前使用的分析树和抽象文法树是这样的。然而，把这些叶子置于图的顶部的画法使得图中的放置与调度代码中的最终放置之间有粗略的对应。叶子位于树的顶部，因为它在调度中可以早些执行。根位于树的底部，因为它必须在它的每一个祖先执行之后才能执行。

给定一个代码片段的相关图D，一个调度S把每一个结点$n \in N$映射到表示它应该被发行的周期的一个非负整数，假定第一个操作在周期1发行。这提供指令的清晰且精确的定义，也就是说，第i个指令是操作的集合$\{n|S(n)=i\}$。一个调度必须满足以下3个限制。

1）对于每一个$n \in N$，$S(n) \geqslant 1$。这一限制使得操作在执行开始之前不被发行。违反这一限制任意调度不是形式良好的。出于一致性的原因，这一调度也必须至少有一个操作n'有$S(n')=1$。 589

2）如果$<n_1, n_2> \in E$，那么$S(n_1)+delay(n_1) \leqslant S(n_2)$。这一限制保证正确性。任意一个操作在产生它的操作数的操作完成之前不能被发行。违反这一规则的调度改变代码中的数据流，而且可能产生不正确的结果。

3）每一个指令包含的每一个类型t的操作的数目不能超过目标机器在一个周期发行的操作数目。这一限制保证可行性，因为违反这一规则的调度包含目标不可能发行的指令。（在VLIW机器上，调度器必须使用nop填充指令中不被使用的槽。）

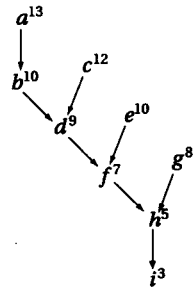
编译器应该只产生满足上述三个限制的调度。

给定一个既正确又可行的形式良好的调度，假定在周期1发行第一个指令，这一调度的长度就是最后的操作完成时的周期数。这可以计算如下：

$$L(S) = \max_{n \in N}(S(n) + delay(n))$$

假设$delay$刻画所有操作的等待时间，调度S应该在$L(S)$周期执行。调度长度的概念产生时间-最优 590
（time-optimal）调度的概念。一个调度S_i是时间-最优的，如果$L(S_i) \leqslant L(S_j)$对于所有其他包含相同操作集合的调度S_j都成立。

相关图刻画这一调度的重要性质。计算沿着通过这一图的路径的总等待，揭示关于这个块的额外细节。使用相应的累加等待时间的信息注释我们例子的相关图导致下面的图：



图中各结点的上标给出从这一结点到计算末端的路径长度。这些值明确指出路径$abdfhi$是最长的，它是决定本例整体执行时间的关键路径（critical path）。

那么编译器应该如何调度这一计算呢？当一个操作的操作数是可用的时候，它才能被调度到一个指令中。因为a、c、e和g在这一图中都没有前驱，所以它们是调度的初始候选者。a位于关键路径的事实强烈要求把a调度到第一个指令中。一旦a已被调度，剩余在D中的最长等待时间路径是$cdefhi$，这要求c作为第二个指令被调度。对于调度ac、b和e都是形成最长路径的结点。然而，b需要a的结果，直到第4个周期它才可用。这使得e成为在b后面的最好选择。继续这一方式，导致调度$acebdgfhi$。这与图12-1的左边所示的调度一致。

然而，编译器不能简单地把指令重新排序成所提议的顺序。回想一下，c和e都定义r_2，d使用c存储于r_2中的值。调度器不能把e移动到d之前，除非它要重命名e的结果以避免与c对r_2的定义相冲突。这一限制不是来自于由D中的边模型化的相关性的数据流。这是由于调度需要避免与在D中被模型化的相关

591 性相干扰。这些限制通常被称为反相关性（antidependence）。

调度器至少可以使用两种不同的方式产生正确代码。它可以发现存在于输入代码中的反相关性，并把它们反映到最终的调度中，或者它可以重命名值来避开它们。本例包含四个反相关性，即$e \rightarrow c$、$e \rightarrow d$、$g \rightarrow e$和$g \rightarrow f$。所有这些都涉及r_2的再定义。（也存在基于r_1的限制，但是每一个关于r_1的反相关性与一个基于值流的相关性重复。）

考虑反相关性改变编译器可以生成的调度集合。例如，它不能把*e*移到*c*或*d*之前。这迫使它生成诸如*acbdefghi*这样的调度，而这一调度需要18个周期。尽管这一调度对原来非调度代码（*abcdefghi*）改进了10%，但是它与通过重命名来生成的如图12-1右边所示的*acebdgfhi*而得到的35%的改进无法相比。

作为另一个可选的策略，调度器可以系统地重命名这一块中的值，以便在它调度代码之前消除反相关性。这一方法把调度器从反相关性强加的限制中解放出来，但是它也可能产生这样的问题：被调度代码是否需要溢出代码。重命名的动作不改变活变量的数目；它只是改变它们的名字并给调度器交叠操作的自由，而反相关性会妨碍这种自由。然而，增加交叠可能增加对寄存器的需求，并迫使寄存器分配器插入更多的溢出代码，这增加长等待时间操作并迫使另一轮调度。

最简单的重命名方案是把一个新名字赋给每一个它产生的值。在我们的例子中，这一方案产生如下
592 的代码。代码的这一版本的定义和使用有相同的模式。

然而，在代码中非歧义性地表示依赖关系。它不包含反相关性，所以不会出现命名限制。

```
a: loadAI  r0,0  ⇒ r1
b: add     r1,r1 ⇒ r2
c: loadAI  r0,8  ⇒ r3
d: mult    r2,r3 ⇒ r4
e: loadAI  r0,16 ⇒ r5
f: mult    r4,r5 ⇒ r1
g: loadAI  r0,24 ⇒ r6
h: mult    r5,r6 ⇒ r7
i: storeAI r7    ⇒ r0,0
```

12.2.1 调度质量的其他度量

调度可以用时间之外的其他方法度量。对于相同输入代码的两个调度S_i和S_j可能产生对寄存器的不同需求，也就是说，S_j中的活变量的最大数量可能比S_i少。如果处理器需要调度器为闲置的功能单元插入nop，那么S_i所拥有的操作数可能比S_j少，而且因此可能提取更少的指令。这不一定只依赖于调度长度。例如，在带有可变周期的nop的处理器上，把nop捆绑到一起可以产生较少的操作，因而也可能产生较少的指令。最后，在目标机器上执行S_j所需的资源可能比S_i少，因为它从不使用其中一个功能单元，提取较少的指令，或者它较少引发处理器的指令解码器中的位转换。

12.2.2 使调度困难的因素

调度中的基本操作是把操作收集到一起形成基于这些操作将开始执行的周期的小组。对于每一个操作，调度器必须选择一个指令和周期。对于每一个指令（和周期）它必须选择一组操作。在平衡这两个观点时，调度器必须确保只有当一个操作的操作数是可用时才能发行它。在一个给定的周期，可能有多个操作满足这一标准，所以调度器必须在其中做出选择。因为值从它的定义到它的最后使用是活的，所以调度器所做的决策可以通过把使用移近定义来缩短寄存器的生存期，或通过使用远离定义而加长寄存器的生存期。如果太多的值同时活着的话，那么调度将是不可行的，因为它需要太多的寄存器。在寻找

最优调度的同时，平衡这些因素使调度复杂化。

除对最简单的体系结构外，局部指令调度总体来说是NP完全的。编译器使用贪婪搜索法产生调度问题的近似解。在实践中，几乎所有用于编译器的调度算法都基于启发式搜索技术的一个系列，称为*列表调度*（list scheduling）。下面一节详细描述列表调度。接下来的几节将展示如何把这一范例扩展到更大的作用域。

593

指令调度与寄存器分配之间的关系

特定名字的使用可能引入反相关性，从而限制调度器重排序操作的能力。调度器可以通过重命名避开反相关性；然而，重命名却带来在调度后编译器执行寄存器赋值的需求。这只是指令调度和寄存器分配的相互作用的复杂方式的一个例子。

指令调度器的核心功能是重排序操作。因为大多数操作使用一个或两个值并且产生一个新值，改变这些操作的相对顺序会改变某些值的生存期。如果操作a定义r_1而其后的操作b定义r_2，那么把b移动到a之前可以有若干效应。它可以延长r_2的生存期一个周期，而缩短r_1的生存期。如果a的一个操作数是一个最后使用，那么把b移到a之前延长这个操作数的生存期。对称地，如是b的一个操作数是最后使用，那么重排序可以缩短那个操作数的生存期。

把b移到a之前的实际效应依赖于a和b的细节以及周围的代码。如果所有使用都不是最后使用，那么交换a和b不会对寄存器的需求产生实际影响。（每一个操作定义一个寄存器；交换操作的位置改变特定寄存器的生存期，但是对寄存器的总体需求没有改变。）

以类似的方式，寄存器分配器可以改变指令调度问题。寄存器分配器的核心功能是重命名引用并当对寄存器的需求比寄存器单元大时插入内存操作。这些功能影响调度器产生快速代码的能力。当分配器把较大的虚拟名字空间映射到目标机器寄存器的名字空间时，它可以引入限制调度器的反相关性。同样地，当分配器插入溢出代码时，它把操作加入到代码中，而这一代码本身必须调度到指令中。

数学上，我们知道一起解决这些问题可能产生通过运行调度器之后运行分配器，或者运行分配器之后运行调度器所得不到的解。然而，这两个问题都非常复杂，使得我们必须分别处理它们。在实践中，几乎所有现实世界的编译器也都是分别处理它们的。

594

12.3 列表调度

列表调度是在一个基本块中调度操作的一种贪婪、启发式搜索方法。自20世纪70年代后期它已成为指令调度的主要范例，这很大程度上是因为它发现可行的调度，并很容易适应计算机体系结构的改变。然而，列表调度描述的是一种方法而不一个特殊的算法。它在实现以及为了调度而为指令设定顺序的尝试上有多种形式。本节揭示列表调度的基本框架，以及基于这一思想的几组变形。

典型的列表调度操作于一个基本块。把我们的考虑限定在代码的直线序列上使得我们可以忽视可能复杂化调度的情况。例如，当调度器考虑多个块时，一个操作数可能依赖于多个前面的定义；这可能为这一操作的操作数的可使用时间带来不确定的成份。在它作用域中使用多个块，调度器可能把一个指令从一个块移到另外一个块中。这样的跨块间的代码移动可能会迫使调度器复制指令，把这一指令移到一个后继的块中可能导致每一个后继块都需要这一拷贝。同样地，它可能会引发一个指令在一条路径上执行，而在源代码中，这一指令不在此路径上执行：将它移到一个前驱块中把它放入通向那个块的每一个后继的路径上。把我们的考虑限制在单一模块的情况避免这些复杂性。

为了把列表调度运用于一个块，调度器遵照一个四步计划。

1）*重命名以避免反相关性*。为了减小对调度器的限制，编译器重命名值。每一个定义得到惟一名字。这一步不是严格必要的。然而，它却能使调度器找到反相关性阻止的调度。它还简化调度器的实现。

2）*构建一个相关图D*。为了构建相关图，调度器自底向上遍历块。对于每一个操作，调度器构建一个结点来表示新近创建的值。它增加从那个结点到使用这个值的每一个结点的边。使用当前操作的等待时间注释每一条边。（如果调度器不执行重命名，D还必须表示反相关性。）

3）*给每一个操作指定优先度*。当调度器在每一步从一组可用操作提取操作时，它利用这些优先度引导它自己。很多优先度方案已被用于列表调度中。调度器可以为每个结点计算若干不同的分数，使用
595 其中的一个作为主要顺序，而使用其他的分数来排除同级结点关系。最受欢迎的优先度方案使用从这个结点到*D*的一个根的最长等待时间加权路径的长度。我们将在本节后面描述其他优先度方案。

4）*迭代选择一个操作并对其进行调度*。列表调度算法的核心数据结构是在当前周期中可以合法执行的操作的列表，称为*就绪表*（ready list）。就绪表上的每一个操作的所有操作数都已可用。算法开始于块中的第一个周期，并尽可能多地挑选出在那个周期能发行的操作。然后它递增周期计数器，并更新就绪表以反映前面已发行的操作和时间的逝去。它重复这一过程，直到每一个操作已被调度。

重命名和构建*D*是直截了当的。优先度计算一般涉及*D*的一次遍历。算法的核心以及理解它的关键在于上述最后一步。图12-3给出这一步的框架，假定目标机器有单一功能单元。

```
Cycle ← 1
Ready ← leaves of D
Active ← ∅
while (Ready ∪ Active ≠ ∅)
  if Ready ≠ ∅ then
    remove an op from Ready
    S(op) ← Cycle
    add op to Active
  Cycle + +
  for each op ∈ Active
    if S(op) + delay(op) < Cycle then
      remove op from Active
      for each successor s of op in D
        if s is ready
          then add s to Ready
```

596 图12-3 列表调度算法

这一算法执行代码执行的一个抽象模拟。它忽视值和操作的细节而集中于由*D*的边所揭示出的时间限制。为了跟踪时间，它用变量*Cycle*维护一个模拟时钟。*Cycle*被初始化为1，即被调度代码中的第一个操作，而且在代码中每一个操作都被指定一个执行时间为止，这个时钟被依次递增。

这一算法使用两个列表来跟踪操作。第一个列表*Ready*保存在当前周期可以执行的所有操作。如果一个操作在*Ready*中，那么这一操作的所有操作数已被计算。最初，*Ready*包含*D*的所有叶子，因为这些叶子不依赖于其他任何操作。第二个列表*Active*保存在前面的一个周期已发行但还没有完成的所有操作。在每一时间步长处，调度器检查*Active*来发现已完成的操作。当一个操作完成时，调度器检查这些操作在*D*中的每一个后继（使用其结果的操作）来决定它的所有操作数现在是否都是可用的。如果它们都是可用的，那么它把后继*s*加到*Ready*中。

在每一时间步长处，算法遵循一个简单的规则。它计算在前一周期中完成的所有操作，然后为当前

周期调度一个操作。实现的细节将使这一结构略微含混，因为在第一个时间步长处总是有一个空的*Active*列表。while循环开始于第一个时间步长的中间。它从*Ready*中挑选出一个操作并调度它。下一步，它递增*Cycle*来开始第二个时间步长。它为新周期的开始更新*Active*和*Ready*，然后，环绕到循环的顶端，在那里，它重复选择、递增和更新这一过程。

当模拟时钟指出每一个操作的执行都已完成时，这一过程终止。在结尾处，*Cycle*包含块的模拟运行时间。如果所有操作都在由*delay*所指定的时间内执行，而且D的叶子的所有操作数在第一个周期可用，那么这一模拟运行时间应该与真实执行时间匹配。一个简单的后序遍可以重排序操作并在空周期插入nop。

此时，一个重要的问题是这一方法所生成的调度好到什么程度？总体上，这一答案依赖于这一算法在每一次迭代中是如何从*Ready*列表中挑选要消除的操作的。考虑最简单的情况，这里的*Ready*列表在每一次迭代至多包含一项。在这一限定的情况下，这一算法必将生成最优调度。只有一个操作可以在第一个周期执行。（在D中至少存在一个叶子，而且我们的限制保证刚好存在一个叶子。）在每一个后继周期，算法没有选择的余地：或者*Ready*包含一个操作而且算法调度它，或者*Ready*是空的而且算法不调度任何在那个周期发行的操作。当在这一过程的某一点处多个操作可用时就会出现困难。 597

在我们的例子中，D有四个叶子：a、c、e和g。只使用一个功能单元，调度器必须选出四个装入操作中的一个在第一个周期执行。这是优先度计算的作用：它给D中的每一个结点赋一个或多个调度器用于排序调度的等级。较前提出的度量，到D中一个根结点的最长等待时间权距离，对应于正在构建的调度中，总是选择位于当前周期的关键路径上的结点。（改变较早的选择可能改变当前关键路径。）在某种程度上，调度优先度的影响是可预测的，这一方案应该提供最长路径的平衡跟踪。

12.3.1 效率

如前所述，从*Ready*列表中选出一个操作需要*Ready*上的一个线性扫描。这使得创建并维护*Ready*的代价接近$O(n^2)$。使用优先队列取代这一列表可以降低维护的代价到$O(n \log_2 n)$，代价使实现难度的稍许增加。

类似的方法可以降低维护*Active*列表的代价。当调度器把一个操作加到*Active*时，它可以给这一操作指定一个等于这一操作完成的周期的优先度。寻找最小优先度的优先队列将把所有在当前周期完成的操作都推向队列的前端，相对于简单的列表实现的代价有稍许增加。

进一步改进Active的实现是可能的。调度器可以维护一个独立列表的集合，对应于每一个周期有一个存放在该周期可以完成的操作的列表。覆盖所有操作等待时间所需的列表数量是$MaxLatency = \max_{n \in D} delay(n)$。当编译器在*Cycle*调度一个操作$n$时，它把$n$加到*WorkList*[$(Cycle + delay(n)) \bmod MaxLatency$]中。当它要更新*Ready*队列时，其后继需要考虑的所有操作都在*WorkList*[$Cycle \bmod MaxLatency$]中。对于每一次到*WorkList*的插入，这一方案使用少量额外空间和稍多一点的时间。反之，它避免搜索*Active*的平方代价，而且使用线性遍更小的*Worklist*来取代搜索*Active*。[⊖] 598

警告

如前所述，局部列表调度器假设叶子的所有操作数在这个块的入口处可用。调度器必须接受在块的边界处出现的所有效应。如果处理器具有在一个操作数还不可用时停顿的互锁功能，那么调度器可以依赖于这些互锁来保证互锁相关性得到满足。如果处理器没有互锁功能，那么调度器必须保证每一个前驱

⊖ 这些*WorkList*中的操作数目等于*Active*中的操作数目，因此空间的额外代价刚好是拥有多个*WorkList*的代价。额外的时间来自于引入了*WorkList*上的下标计算，包括mod计算。

块在末端有足够的松弛时间使所有操作完成工作。使用更多上下文信息，调度器也许能够对叶子进行优先排序，以在后继块中隐藏这一松弛时间。

为了扩展这一算法使得它处理包含多个操作的指令，我们必须扩展*while*循环，使得它为每一个功能单元选择一个操作。如果一个给定的操作可以在若干不同的功能单元中执行，那么编译器设计者可能需要根据那些功能单元的能力进行操作选择。例如，如果只有一个功能单元可以执行内存操作，那么调度器必须把装入和存储优先地调度到那个单元中。如果寄存器组被划分（参见13.6.2节），那么将会产生类似的效应；调度器也许需要把操作放到这一操作的操作数驻留的单元中，或者在内部划分转换设备空闲的周期放置操作。

作为最后一点，内存操作通常因为内存层次的行为而带有不确定且可变的等待。在带有多级缓冲内存的机器上，装入操作可能有一个从零周期到几百个或几千个周期的等待范围。如果调度器假设最坏情况等待，那么它将冒着长期空运转处理器的危险。如果它假设最好情况等待，它将在缓冲错误上停顿处理器。在实践中，编译器可以根据可用于隐藏装入等待时间的指令级并行的数量来计算每一个装入的等待时间，以此来获取良好的结果。事实上，这是针对这一装入周围的代码来调度这一装入的，而不是针对执行的硬件来调度的。一旦这些可用的指令级并行被用尽，处理器将停顿，直到这个装入完成。

12.3.2 其他的优先度方案

局部指令调度的复杂性迫使编译器使用启发式搜索技术。在实践中，列表调度产生良好的调度，这
599 是一个近似最优的调度。但是，这些贪婪技术的行为很少是健壮的，输入的较小变化都可能在结果上产生很大的不同。

解决这一不稳定性的一个算法上的方法是细心的平局加赛。当两个或更多项有相同的等级时，实现应该基于另一个优先等级在它们中间做出选择。（相反，极大函数的实现一般都展示确定性的行为：它们或者保留第一个最大等级值或者保留最后最大等级值。这导致向列表中的较早结点或较晚结点的系统偏向。）一个好的列表调度器使用若干不同的优先度等级来进行平局加赛。文献中介绍的优先度方案包括：

- 一个结点的等级是包含这个结点的最长路径的总长。在每一步，这一方法偏好源代码中的关键路径，而忽视中间决策。这倾向于*D*的深度优先遍历。
- 一个结点的等级是它在*D*中所拥有的立即后继的数量。这支持调度器跟踪整个图中的很多不同路径，更接近于广度优先方法。它倾向于把更多的操作保存在*Ready*队列中。
- 一个结点的等级是它在*D*中所拥有的子孙的总数量。这放大前面等级的效应。为其他结点计算关键值的结点要早些调度。
- 如果一个结点有长等待时间，那么它的等级则较高。当更多的操作留下来可能用于覆盖结点的等待时间时，这倾向于早些调度块中长等待时间的结点。
- 如果一个结点包含一个值的最后使用，那么它的等级较高。这倾向于通过把最后使用移近它的定义来减小对寄存器的需要。

遗憾的是，其中没有一个调度方案在整体调度质量上胜于其他调度。每一个方案在某些例子上有优势，而在其他例子上则毫无优势。因此，关于使用哪个等级方案以及使用这些等级方案的顺序没有达成共识。

12.3.3 前向与向后列表调度

列表调度的另外一种形式以相反方向检查相关图，从根到叶子实施调度。被调度的第一个操作在块

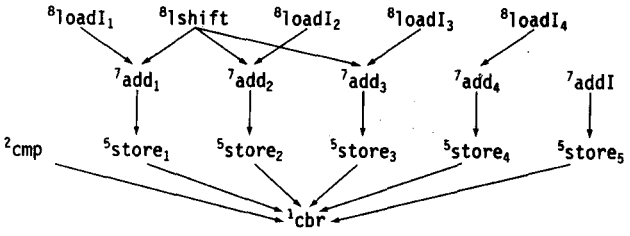
的最后周期执行，而被调度的最后一个操作则第一个执行。以这种形式，这一算法被称为向后列表调度 600
(backward list scheduling)，而原来版本的调度称为前向列表调度（forward list scheduling)。

编译器领域的一个标准实践是要在每一个块上尝试列表调度的若干版本，而且保留最短的调度。编译器一般既尝试前向列表调度，又尝试向后列表调度；它可能在每一个方向上尝试多个优先度方案。像很多技巧都来自于试验一样，这一技巧编码若干重要的观点。

第一，列表调度只占据编译器执行时间的一小部分。如果列表调度产生较好的代码，那么值得去尝试多个方案。注意，调度器可以复用大部分准备性工作：重命名、构建D以及某些已计算的优先度。因此，使用多个方案的代价相当于若干次重复调度器的迭代部分。

第二，实践表明，前向和向后调度都不是长胜不败的。前向列表调度和向后列表调度之间的差异在于它们各自考虑的操作的顺序。如果调度决定性地依赖于对某些短操作序列的精心排序，那么这两个方向上的调度可能产生显著不同的结果。如果关键的操作在叶子附近出现，向前调度似乎更有可能一起考虑它们，而向后调度必须遍历模块的其余部分来达到这些关键操作。对称地，如果关键操作出现在根的附近，那么向后调度有可能一起检查它们，而前向调度将以在块的末端开始时所做的决定指定的顺序检查它们。

为了使第二条的讨论更具体些，考虑图12-4给出的例子。这一例子给出SPEC基准程序go中的一个基本块的相关图。编译器把store操作的相关性加到块结束分支以确保内存操作在下一个块开始执行之前完成。（违反这一假设将产生一系列装入操作的不正确结果。）相关图上结点的上标给出这一结点到分支的等待时间；下标区分相似的操作。这一例子假设相关图下方的表格所示的操作等待时间。



操作码	loadI	lshift	add	addI	cmp	store
等待时间	1	1	2	1	1	4

图12-4 go的一个模块的相关图

这一例子展示前向和向后列表调度之间的差异。这来自于我们对列表调度的研究；编译器以带有两个整数功能单元和一个执行内存操作的单元的ILOC机器为目标。五个store操作花去这个块的大部分时间。最小化执行时间的调度必须尽可能早地开始执行各store操作。

使用到根的等待时间为优先度，除比较之外，向前列表调度按优先度顺序执行各操作。它按等级8 601
调度5个操作，然后按等级7调度5个操作。它开始处理等级为5的操作，将cmp沿着store操作移进来，因为cmp是叶子。如果按从左到右的顺序随意地进行平局加赛，那么这产生如图12-5左侧所示的调度。注意，内存操作在周期5开始，产生在周期13发行分支的调度。

对向后列表调度使用相同的优先度，编译器首先把分支放置到这一块的最后槽。cmp比它前$delay$(cmp)=1个周期。下一个被调度的操作是$store_1$（按左到右进行平局加赛规则)。它被指定到内存单元上$delay$(store)=4周期前的发行槽。调度器使用其他store操作相继填充这一内存单元上的更早的槽。它开始填充整数操作，因它们已准备就绪。第一个是add_1，在$store_1$的前两个周期。当算法终止时，它产生如图12-5右侧所示的调度。

	前向调度				向后调度		
	整 数	整 数	内 存		整 数	整 数	内 存
1	$loadI_1$	lshift	—	1	$loadI_4$	—	—
2	$loadI_2$	$loadI_3$	—	2	addI	lshift	—
3	$loadI_4$	add_1	—	3	add_4	$loadI_3$	—
4	add_2	add_3	—	4	add_3	$loadI_2$	$store_5$
5	add_4	addI	$store_1$	5	add_2	$loadI_1$	$store_4$
6	cmp	—	$store_2$	6	add_1	—	$store_3$
7	—	—	$store_3$	7	—	—	$store_2$
8	—	—	$store_4$	8	—	—	$store_1$
9	—	—	$store_5$	9	—	—	—
10	—	—	—	10	—	—	—
11	—	—	—	11	cmp	—	—
12	—	—	—	12	cbr	—	—
13	cbr	—	—				

图12-5 go中模块的调度

向后调度器所产生的调度所花的时间比前向调度器所产生的调度所花的时间少一个周期。向后调度把addI较早地放置在块里，允许$store_5$在周期4发行，这比向前调度中第一个内存操作提前一个周期。使用相同的基础优先度和平局加赛器，以不同的顺序考虑问题，向后算法得到不同的结果。

为什么会发生这样的情况呢？向前调度器必须在所有等级为7的操作之前把所有等级为8的操作放置到调度中。即使addI操作是叶子，其较低的等级促使前向调度器等待对它的处理。直到这一调度器用尽
602 等级8的操作时，等级为7的操作才可用。与此相对应，向后调度器在三个等级为8的操作之前把addI放置到调度中，这是向前调度器无法考虑的结果。

列表调度算法使用贪婪、启发式搜索构建这一问题的可行解决方案，但是等级函数不编码这一问题的完备信息。为了构建编码完备信息的等级，编译器也许需要考虑所有可能的调度，这使得等级过程自身成为NP完全问题。因此，除非P＝NP，最优等级函数是不切实际的。

12.3.4 使用列表调度的原因

多年来列表调度在指令调度算法中一直占据着统治地位。以它的前向和向后形式，这一技术使用贪
603 婪、启发式搜索方法给每一个操作在特定发行槽指定特定指令。在实践中，这对单一块产生卓越的效果。

关于无序执行

一些处理器包含无序（OOO）执行指令的硬件支持。我们称这样的处理器为*动态调度*（dynamically scheduled）机器。这并非新特性；例如，它出现在IBM 360/91。为了支持OOO执行，动态调度处理器检查指令流，来寻找那些可以在静态调度器指定的执行位置之前执行的操作。为了做到这一点，动态调度处理器在运行时构建并维护相关图的一个部分。它使用相关图的这一片段来发现每一个指令可以执行的时间，并在第一个合法机会处发行每一个指令。

无序处理器在什么时候可以改进静态调度呢？如果运行时环境比调度器假设的环境要好，那么OOO硬件可以在一个操作的静态调度位置前发行这一操作。如果一个操作的操作数在它的最坏情况时间之前可用，那么这一情况可能发生在块的边界处。因为OOO处理器知道真实的运行时地址，所

以它也可以消除某些调度器所不能消除的装入-存储所依赖的歧义性。

OOO执行不消除对指令调度的需求。因为向前看窗口是有限的，不好的调度可能不带来改进。例如，50个指令的向前看窗口将不会让处理器按交叉的<inter，floating-point>对的方式执行由100个整数指令串后面跟着100个浮点指令组成的指令序列。然而，它也许可以使较短的指令序列交叉运行，比如说长度为30的串。OOO执行通过改进好的但并非最优的调度来帮助编译器。

一个与此相关的处理器特性是动态寄存器重命名。与ISA允许编译器的命名相比，这一方案为处理器提供更多的物理寄存器。通过使用对编译器来说不可视的额外物理寄存器，来实现两个反相关的引用，这一处理器可以消除发生在它的向前看窗口内的反相关性。

列表调度是高效的。它一次从*Ready*队列移走一个操作。对于每一个操作和每一个进入该操作的边，列表调度做一次检查来决定是否把该操作加到*Ready*队列中。如果一个操作有m个操作数，那么调度器访问它的结点m次。每一次访问都检查它的每一个操作数，所以把这一操作放置到就绪队列中所涉及的 604
工作量是$O(m^2)$。然而，对于大多数操作，m是一或二，所以这一平方代价是平凡的。

列表调度形成众多在超过单一块的区域上执行调度的算法的基础。因此，了解它的长处和短处是重要的。任意对局部列表调度所做的改进都有改进区域调度算法的可能。

12.4 高级话题

20世纪90年代开发的体系结构增加了指令调度的重要性。带有多个功能单元的处理器的广泛使用、管道的增加和不断增长的内存等待时间结合在一起，使得实现的性能更多地依赖于执行顺序。这带来更具挑战性的调度方法的发展。12.4.1节给出3个区域调度方法。12.4.2节介绍改进调度的代码整形。

12.4.1 区域调度

同值编号一样，从单一基本块到更大作用域的转移可以改进编译器生成的代码的质量。对于指令调度，已经提出很多处理较之一个块大但又小于整个过程的区域的方法。本节列出派生于列表调度的三个方法。

1. 调度扩展基本块

在扩展的基本块（EBB）上使用列表调度算法使调度器考虑比它在单一块所能看到的更长的操作序列。穿越EBB的路径形成调度器可以按单一块处理的代码的直线片段。然而，这些路径在它第一个和最后一个操作之间存在临时出口，而且一些路径可能共享一个公用前缀。当EBB调度器跨越块边界移动操作时，它必须确保与该块间转换中相关的其他路径上的行为的正确性。

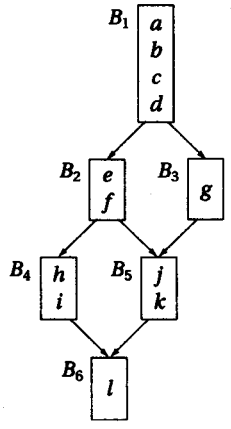


图12-6 扩展基本块的调度例子

考虑图12-6所示的简单代码片段。它有一个较大的EBB，{B_1，B_2，B_3，B_4}和两个平凡的EBB，{B_5}和{B_6}。这个较大的EBB有两个路径{B_1，B_2，B_4}和{B_1，B_3}。作为一个公共前缀，它们共享B_1。因为编译器只生成B_1的一个调度，所以它必须找到同时满足两个路径的调度。 605

B_1表示的这一冲突可以用两种方法展示它自己。调度器可能把一个操作，如c，从B_1移出经过B_1末端的分支进入B_2。在这种情况下，c将不再在从B_1的入口到B_3的入口的路径上，所以调度者需要把c的拷贝

插入B_3以便维护沿着路径$\{B_1, B_3\}$的正确性。另外，调度器可能把一个操作，如f，从B_2向上移到B_1。这把f放置在从B_1的入口到B_3的入口的路径上，此前它并不执行。

把f移入B_1中可能带来两个问题。它加长路径$\{B_1, B_3\}$，这伴随着沿着那条路径放慢执行的可能性。它也可能改变那条路径的正确性。为了避免后面的问题，编译器必须重命名值以遵从源代码中的非相关性。而这一重命名又可能会迫使编译器沿着控制流图中的某些边插入拷贝操作。（由此可能引发的问题类似于在把SSA转换成可执行代码中所出现的问题。参见9.3节中的讨论。）

编译器对于处理这些块间冲突有若干选择。为了处理向下移动中出现的问题，它可以把*补偿代码*
606 (compensation code) 插入这一路径的出口块中。每当把一个操作从B_i移到B_i的一个后继中时，它必须考虑在B_i的每一个其他立即后继中创建这一指令的拷贝。如果由这个被移动的指令所定义的值在后继中是活的，那么这一指令是必要的。为了处理向上移动所产生的问题，调度器可以阻止把一个操作从原来块移到一个前驱块中。这避免加长离开该前驱块的其他路径。

（还存在第三个选择。编译器也许要根据周围的上下文复制块。参见8.7.1节。在本例中，可以复制B_5来提供两个实现B_5'和B_5''，从B_2到B_5'及从B_3连结到B_5''各连一条边。这将创建包含新块的更长的EBB。这种转换超出了当前所讨论的内容。）

为了调度穿过一个EBB的一条路径，如果必要的话，编译器在这一区域上执行重命名。接下来，它构建整个路径的单一相关图，而忽视这一路径的任意出口。它为了在就绪操作间进行选择以及进行平局加赛计算优先度。最后，同单一模块的情况一样，它运用迭代调度方案。每当它把一个操作指定给这一调度的特定周期的特殊指令时，它必须插入这一选择所需的补偿代码。

编译器一般对每一个块做一次调度。因此，在我们的例子中，编译器可能首先调度路径$\{B_1, B_2, B_4\}$，因为它是最长的路径。接下来调度$\{B_1, B_3\}$，但是B_1已经有了一个固定的调度。因此，编译器可以使用B_1作为前缀来调度B_3。最后，它可以（以任意顺序）调度B_5和B_6。如果编译器有理由相信穿越一个EBB的一条路径比另外一条路径更频繁地执行，那么它应该使该路径优先并首先调度它，使得它保留调度的完整性。

2. 跟踪调度

跟踪调度使用程序的真实运行时行为的信息来选择调度的区域。它使用通过运行这一程序的测试版而收集的侧面信息，来决定哪个块更频繁地执行。根据这一信息，它构建一个表示最频繁执行路径的穿越控制流图的跟踪，或路径。

给定一个跟踪，调度器把列表调度算法运用于整个跟踪，这与EBB调度把列表调度算法运用于穿过某个EBB的一条路径的方式相同。对于任意的跟踪，出现另外一个额外的复杂性：这一跟踪可能有临时入口点和临时出口点。如果调度器把一个操作跨越一个临时入口（即CFG中的一个连接点）向上移动，
607 那么它需要把这个操作拷贝到进入这一连结点的非跟踪路径中。

为了调度整个过程，跟踪调度器构建一个跟踪并调度它。然后，它把这一跟踪中的块从考虑的对象移出，并选择下一个最频繁执行的跟踪。这一跟踪被调度，而且调度必须反映由前面的调度代码所引入的所有约束。这一过程持续下去，选择一个跟踪并调度它，并把它移出考虑之列，直到所有块都被调度为止。

EBB调度可以被认为是跟踪调度的一种退化形式，在这一调度中穿过这一EBB的各条路径恰好是由某个侧面数据的静态逼近所选择的跟踪。

3. 调度循环

因为循环在大多数计算密集的任务中起着关键性的作用，所以在关于编译的文献中，人们对它投入

了大量的关注。EBB调度可以处理循环体的一个部分，[⊖]但是它不能处理从循环底部回到循环顶部的“环绕”的等待时间。跟踪调度可以处理这些环绕问题，但是它需要通过创建在这一跟踪中出现多次的任意块的多次拷贝来做到这一点。这些缺陷导致若干技术的产生，这些技术直接处理这一问题，生成最内部循环的循环体的优秀调度。

专门的循环调度技术只有当缺省调度器无法为循环产生高效且简洁的代码时才有意义。调度之后，如果这一循环有不含停顿、互锁和nop的循环体，那么专门的循环调度器也不可能改进它的性能。同样地，如果这一循环的循环体足够长，使得块末端效应只是它的运行时间的微小部分，那么专门调度器也不大可能有帮助。

然而，很多小的计算密集的循环仍受益于循环调度。典型地，相对于操作的关键路径长度，这些循环含有太少的操作，以致于无法保持基础硬件忙碌。调度这些循环的关键是同时执行这一循环的多个迭代，例如，使用三次迭代执行，给定的周期可能从第i次迭代的开始、从第$i-1$次迭代的中间以及从第$i-2$次迭代的末端发行操作。为了创建这样的调度，调度器创建一个被称为核（kernel）的固定长度循环体，并使用模算术来调度它。事实上，这把单一迭代的执行叠入核上。这些循环通常被称为管道化循环（pipelined loop）。

为了使管道化循环正确执行，代码必须首先执行补足管道的序言部分。如果核执行来自于原来循环的三次迭代的操作，那么每一次核迭代大致处理原循环的每一个活动迭代的三分之一。为了开始执行，序言必须为迭代1的最后三分之一，迭代2的中间三分之一和迭代3的前三分之一做足够的准备工作。当核循环完成后，需要一个相应的结语（epilogue）来完成最后的迭代：空出管道。对独立的序言和结语部分的需求增加代码的尺寸。尽管这一特定的增加是循环和核同时执行的迭代次数的函数，但是序言和结束语使循环所需的代码量加倍的情况并非不常见。

608

下面的例子将使这一想法更具体。考虑下面用C语言写成的循环：

```
for (i=1; i < 200; i++)
    z[i] = x[i] * y[i];
```

图12-7给出在优化后编译器可能为这一循环所生成的代码。这时，已经运用了操作符强度减弱和线性函数测试替换（参见10.3.3节）。上图中的代码已经为一台带有一个功能单元的机器调度过。调度器假设装入和存储花三个周期，乘法花两个周期而其他所有操作花一个周期。第一列给出周期计数，以循环中的第一个操作为标准（在标签L_1处）。

609

预循环代码为每一个数组初始化一个指针，并计算循环末端测试在周期6所使用的$r_{@x}$的范围的上界r_{ub}。循环体装入x和y执行乘法，并把结果存储到z。调度器使用其他操作填充长等待操作的隐藏区的所有发行槽。在装入的等待时间，调度更新$r_{@x}$和$r_{@y}$。它在乘法的隐藏区执行比较。它使用$r_{@z}$的更新和分支填充存储后的槽。这为一个功能单元机器产生一个紧凑的调度。

考虑如果我们在一台带有两个功能单元和相同等待时间的超标量处理器运行代码的话会发生什么。假设装入和存储必须在功能单元零上执行，且假设功能单元停顿直到一个操作的操作数都已就绪，且处理器不能向停顿单元发行操作。图12-8给出循环的第一次迭代的执行。周期3的mult停顿，因为r_x和r_y都没有就绪。它在周期4停顿，等待r_y，在周期5重新开始执行，且在周期6的末端产生r_z。这迫使storeAO到周期7的开始一直停顿。假设硬件可以区别$r_{@z}$包含不同于$r_{@x}$和$r_{@y}$的地址，那么处理器可以在周期7为第二次迭代发行第一个loadAO。如果不能区别这一信息，那么处理器将停顿，直到这一存储完成。

⊖ 如果这一循环只包含一个模块，那么它无需扩展基本模块。如果它包含break式的到循环底部的跳转之外的任意内部控制流，那么一个EBB不可能包含它的所有模块。

周期	功能单元0				注释
-4		loadI	@x	⇒ $r_{@x}$	设置循环
-3		loadI	@y	⇒ $r_{@y}$	初始装入
-2		loadI	@z	⇒ $r_{@z}$	
-1		addI	$r_{@x}$,792	⇒ r_{ub}	
1	L_1:	loadAO	r_{arp},$r_{@x}$	⇒ r_x	得到x[i] & y[i]
2		loadAO	r_{arp},$r_{@y}$	⇒ r_y	
3		addI	$r_{@x}$, 4	⇒ $r_{@x}$	在装入的隐藏区
4		addI	$r_{@y}$, 4	⇒ $r_{@y}$	递增指针
5		mult	r_x,r_y	⇒ r_z	真正的工作
6		cmp_LT	$r_{@x}$,r_{ub}	⇒ r_{cc}	mult的隐藏区
7		storeAO	r_z	⇒ r_{arp},$r_{@z}$	存储结果
8		addI	$r_{@z}$, 4	⇒ $r_{@z}$	递增z的指针
9		cbr	r_{cc}	→ L_1,L_2	循环结束分支
	L_2:	...			

图12-7 一个功能单元的调度循环例子

周期	功能单元0				功能单元1			
-2		loadI	@x	⇒ $r_{@x}$	loadI	@y	⇒	$r_{@y}$
-1		loadI	@z	⇒ $r_{@z}$	addI	$r_{@x}$,792	⇒	r_{ub}
1	L_1:	loadAO	r_{arp},$r_{@x}$	⇒ r_x	无操作发行			
2		loadAO	r_{arp},$r_{@y}$	⇒ r_y	addI	$r_{@x}$, 4	⇒	$r_{@x}$
3		addI	$r_{@y}$, 4	⇒ $r_{@y}$	mult	r_x,r_y	⇒	r_z
4		cmp_LT	$r_{@x}$,r_{ub}	⇒ r_{cc}	r_y处停止			
5		storeAO	r_z	⇒ r_{arp},$r_{@z}$	addI	$r_{@z}$, 4	⇒	$r_{@z}$
6		r_z处停止			cbr	r_{cc}	→	L_1,L_2
7		…开始下次迭代…						

610

图12-8 两功能单元超标量处理器上的执行

转移到两功能单元改进了执行时间。它把预循环时间削减一半到两个周期。它减少三分之一循环体的时间，到6个周期。关键路径可以如我们所愿快速执行；乘法在r_y可用之前发行且尽可能早地执行。一旦r_z可用，存储就开始进行。某些释放槽被浪费了（在周期6时是单元0，周期1和4时是单元1）。

重排序线性代码可以改变执行调度。例如，如果把$r_{@x}$的更新移到从$r_{@y}$的装入的前面，那么处理器可以在与从寄存器$r_{@x}$和$r_{@y}$的装入的相同周期发行$r_{@x}$和$r_{@y}$的更新。这使得某些操作在调度中更早释放，但是它对加速关键路径没有什么帮助。最终结果是相同的：一个6周期循环。

管道化这一代码可以减少每一次迭代所需的时间。图12-9给出经编译器管道化之后的相同循环。调度器在这里已把这一循环叠成一半，使得它的核同时执行两次迭代。此循环的序言（周期－4到－1）同前面一样执行初始化代码，以及迭代1的上一半。核执行迭代i的下一半和迭代$i+1$的上一半。当核终止
611 时，结语（周期＋1到＋3）执行最后迭代的下一半。

为了理解管道化循环中的值流，考虑图12-10所示的执行。假设这一管道循环执行源循环的四次迭代，此图给出操作执行的顺序。箭头表示沿这一循环的关键路径的值流，两个装入、乘法和存储。

周期	功能单元0	功能单元1
-4	loadI @x ⇒ $r_{@x}$	loadI @y ⇒ $r_{@y}$
-3	loadAO r_{arp},$r_{@x}$ ⇒ $r_{x'}$	addI $r_{@x}$, 4 ⇒ $r_{@x}$
-2	loadAO r_{arp},$r_{@y}$ ⇒ r_y	addI $r_{@y}$, 4 ⇒ $r_{@y}$
-1	loadI @z ⇒ $r_{@z}$	addI $r_{@x}$,788 ⇒ r_{ub}
1	L_1: loadAO r_{arp},$r_{@x}$ ⇒ r_x	addI $r_{@x}$, 4 ⇒ $r_{@x}$
2	loadAO r_{arp},$r_{@y}$ ⇒ r_y	mult $r_{x'}$,r_y ⇒ r_z
3	cmp_LT $r_{@x}$,r_{ub} ⇒ r_{cc}	addI $r_{@y}$, 4 ⇒ $r_{@y}$
4	storeAO r_z ⇒ r_{arp},$r_{@z}$	i2i r_x ⇒ $r_{x'}$
5	cbr r_{cc} → L_1,L_2	addI $r_{@z}$, 4 ⇒ $r_{@z}$
+1	L_2: mult $r_{x'}$,r_y ⇒ r_z	无操作发行
+2	storeAO r_z ⇒ r_{arp},$r_{@z}$	无操作发行
+3	stall on r_y	无操作发行

图12-9 管道调度后的循环示例

612

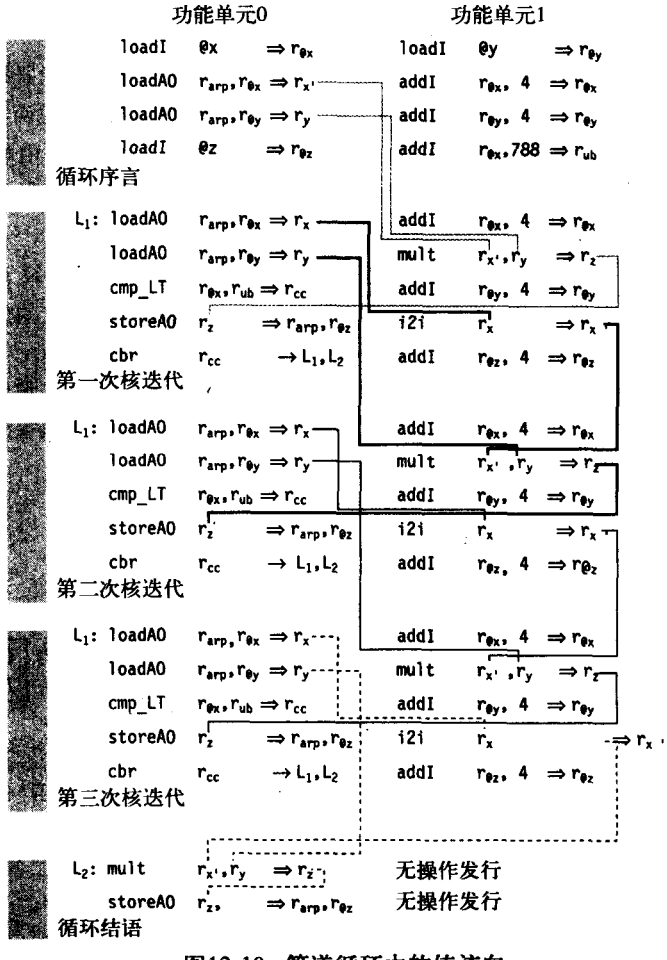


图12-10 管道循环中的值流向

- 第一次源循环迭代开始于管道循环的序言并完成于第一个核迭代。这些流向用细灰线表示。
- 第二次源循环迭代开始于管道循环的第一个核迭代并完成于第二次核迭代。这一流向用粗线表示。
- 第三次源循环迭代开始于管道循环的第二次核迭代并完成于第三次核迭代。这一流向用标准线表示。
- 最后一次源循环迭代开始于管道循环的第三次（最后一个）核迭代。循环的结语代码通过执行乘法和存储完成最后的迭代。这些值流用点线表示。

最后的存储操作停顿以等待乘法的结果。结语也包含某些空闲发行槽。如果调度器有这一循环周围的更多上下文，那么它能够通过从结语之后的基本块中提升操作来改进结语的调度。理想情况下，它应该在乘法和存储操作之间填充空闲发行槽并插入另外一组操作。

4. 回顾

本质上，管道调度器把循环变成一半来减少花在每一次迭代上的周期数。核的执行次数比原来的循环少一次。序言和结语都执行一次迭代的一半。管道循环通过把迭代i的乘法和存储与迭代$i+1$的装入交迭，每一次迭代它少使用两个周期。

产生管道循环的算法需要超出本章范围的分析技术。有兴趣的读者可以查询有关高级优化详细技术的教科书。

12.4.2 上下文的复制

在图12-6的例子中，穿过EBB(B_1，B_2，B_3，B_4) 的两条路径都包含B_1。编译器必须首先选择调度一条路径，或者是（B_1，B_2，B_4）或者是（B_1，B_3）。效应是把另外一条路径分裂成两半；如果编译器首先调度（B_1，B_2，B_4），那么它必须对应已调度的B_1调度B_3。本例中其他两个EBB是由单一块（B_5）和（B_6）组成的。因此，这些块的一半作为单一块来调度，任意得自于EBB调度的效应主要出现在路径（B_1，B_2，B_4）上。通过这一片段的其他每一条路径都遇到局部调度的代码。

如果编译器做出了正确的选择，且B_1、B_2、B_4是最频繁执行的路径，那么分裂（B_1，B_3）是合适的。相反，如果另一条路径，如（B_1，B_3，B_5，B_6）执行得更加频繁，那么EBB调度的效应被浪费掉了。跟踪调度可以捕获并调度这一路径。然而，跟踪调度也展现调度第一条路径的倾向。

为了增加可以被调度的多块路径的数量，编译器可以通过复制基本块来重新捕获上下文。复制可能通过创建调度器有更多改进代码的机会的上下文来提高性能。它也可以降低当编译器选择了错误的执行路径时的性能损失。使用复制，编译器复制有多个前驱的块来创建更大的EBB。

图12-11给出图12-6的例子上的这种复制的执行结果。块B_5被复制来创建从B_2的路径和从B_3的路径的独立实例。同样地，B_6被复制两次，创建进入其中的每一条路径的实例。

复制之后，整个图形成一个EBB。如果编译器选择（B_1，B_2，B_4）为紧迫路径，那么它将首先调度（B_1，B_2，B_4，B_6）。这留下要调度的其他两条路径。它可以使用已调度的（B_1，B_2）为上下文，调度路径（B_5，B'_6）。它也可以使用已调度的（B_1）为上下文，调度（B_3，B'_5，B'_6）。

把这一结果与简单EBB调度器相对比，后者对应于B_1调度B_3，且B_5和B_6都没有前上下文。因为B_5和B_6有多个前驱和不一致的上下文，所以EBB调度器不可能比局部调度做得更好。创建给调度

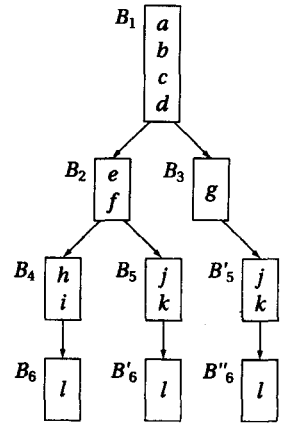


图12-11 通过复制增加调度上下文

器带来额外上下文的B_5和B_6的额外拷贝的代价是语句j和k的第二个拷贝以及语句l的两个额外的拷贝。在实践中，编译器可以把B_4和B_6、B_5和B_6'、以及B_5'和B_6''组合成对。这可以消除每个对中第一个块的块末端跳转。

当然，编译器设计者对这种类型的复制加入某些限制以避免过分的代码增长以及避免循环的展开。典型的实现可以在最内层的循环内复制块，当它达到循环关闭边或返回边时停止复制。这创建一种环境，在这一环境中，循环中仅有的多入口块是第一个块。所有其他路径只有单一入口块。

613 ~ 614

度量运行时性能

指令调度的主要目的是改进已生成代码的运行时间。性能的讨论使用很多不同的度量标准；其中有两个最为常用。

每秒指令数 通常用于告知计算机和比较系统性能的这一度量标准是一秒内执行的指令数量。这可以用于测量每秒发行的指令或每秒内退出的指令的数目。

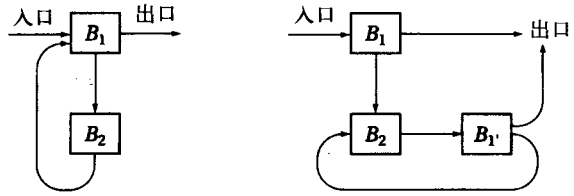
完成一项固定任务的时间 这一度量标准使用一个或多个行为已知的程序，并比较完成这些固定任务所需的时间。这一称为*基准测试*（benchmarking）的方法提供某一特定作业量上硬件和软件的总体系统性能的信息。

没有哪一个标准独自包含足够多的信息，使得编译器后端生成的代码的质量评估得到认可。例如，如果度量是每秒指令数，那么编译器获得关于把无关（但独立）的指令留在代码中的信息了吗？这一简单的测时标准不能提供给定程序可实现程度的信息。因此，它允许一个编译器比另外一个编译器做得更好，但是却不能展示已生成代码之间的差异，以及在目标机器上这一代码的最优性能。

编译器设计者想要测量的数目包括其结果的确被使用的已执行指令的百分比，以及花费在停顿和互锁中的周期的百分比。前者给出预测执行的某些方面的信息，而后者直接测量调度质量的某些方面的信息。

值得考虑的第二个问题出现于尾递归程序中。回想一下，根据7.8.2节，一个程序是尾递归的，如果它的最后动作是一个递归自调用。当编译器发现一个尾调用时，它可以把这一调用转换成到这一过程入口的一个向回跳转。从调度器的观点看，复制可能改进这种情况。 615

图12-12左侧给出尾递归例程在尾调用被转换成迭代之后的抽象化控制流。沿着两条路径进入块B_1，一条路径是这一过程的入口路径，另一条路径是从B_2的路径。这迫使调度器对领先B_1的块做最坏情况假设。如图右侧所示，通过复制B_1，编译器可以使控制只沿着一条边进入B_1'。这可能改进使用EBB调度器或循环调度器的区域调度结果。为了进一步简化这一情况，编译器可以把B_1'结合到B_2的末端，创建一个单一块循环体。可以使用局部调度器或循环调度器调度其结果循环。



经把尾递归转换成迭代后的源代码　　复制之后

图 12-12 616

12.5 概括和展望

为了在现代处理器上得到可行的性能，编译器必须精心地调度操作。几乎所有现代编译器都使用列表调度的某种形式。通过改变优先度方案、平局加赛规则以及甚至是调度的方向，这一算法很容易调整并参数化。它在范围广大的不同代码上产生良好的结果，在这一意义下，列表调度是健壮的。在实践中，它通常找到时间最优的调度。

回应现实的问题，操作于更大区域上的列表调度的变形已经得到发展。调度扩展基本块和循环的技术，本质上对应于编译器必须考虑的管道的数量以及它们各自的等待时间的增加。随着机器变得更加复杂，调度器也需要更多的上下文来发现足够多的指令级并行以保持机器忙碌。软件管道化为增加每一周期发行的操作数量提供一种方法，而且它减少执行循环的总时间。跟踪调度是为VLIW体系结构而发展起来的，为了它，编译器需要保持更多的功能单元忙碌。

本章注释

很多领域都引发调度问题，其范围涉及建筑、工业生产、服务传输以及在太空船上获得有效负载等领域。关于调度的文献也很多，其中包括很多这一问题的特化变形。指令调度从20世纪60年代开始就已作为独立问题得到广泛的研究。

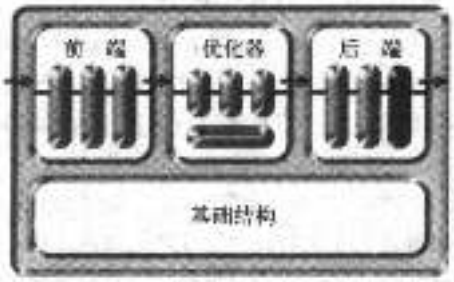
存在对于简单情况保证最优调度的算法。例如，在带有一个功能单元和统一的操作等待时间的机器上，Sethi-Ullman的标签算法为表达式树创建最优调度[301]。它可以运用于产生表达式DAG的好代码。
617 Fischer和Proebsting基于标签算法构建了为小内存等待时间产生最优或近似最优结果的算法[278]。遗憾的是，当等待时间上升或功能单元数量变大时，这一算法就有了麻烦。

关于指令调度的大部分文献研究本章所述列表调度算法的不同版本。Landskov等被公认为是做了列表调度的权威性工作[229]，但是这一算法至少要回溯到1961年的Heller的工作[178]。基于列表调度的其他文献包括Bernstein和Rodeh[37]、Gibbons和 Muchnick[154]，以及Hennessy和Gross[179]的文献。Krishnamurthy等发表了关于管道化处理器文献的高水平概览[224，311]。Kerns、Lo和Eggers作为使列表调度适合于不定内存等待时间的方法开发了平衡调度[210，240]。

很多作者描述了区域调度算法。第一个自动区域技术是Fisher的跟踪调度算法[141，142]。这一算法用于若干商业系统[128，242]和众多研究系统[310]。另外，Hwu等提出了超块（superblock）算法[190]；在一个循环内部，它按与图12-11所示的算法类似的形式复制块来避开连结点。Click基于全局值图的运用提出了全局调度算法[80]。若干作者提出了利用特定硬件特性的技术[310，292]。其他利用复制改进调度的方法包括Ebcioğlu和Nakatani[127]以及Gupta和Soffa[167]。Sweany和Beaty提出了基于支配信息选择路径的方法[316]；其他文献研究了这一方法的不同方面[315，189，98]。

软件管道化已经得到广泛的探究。Rau和Glaeser于1981年引进了这一思想[283]。Lam展示了一个软件管道化算法，这一算法包含处理循环中的控制流的分层方案[226]。在同一个学术会议上，Aiken和Nicolau提出了称之为理想管道化（perfect pipelining）的类似方法[11]。

图12-4中的向后与前向调度的例子是Philip Schielke提供给我们的[298]。该例子引自SPEC基准测试程序go。简明地说，它刻画一种效应，这一效应促使很多编译器设计者在他们的编译器后端同时加入前
618 向和向后调度器。



第13章
寄存器分配

13.1 概述

寄存器是内存层次结构中最快的场所。它们通常只是大多数操作可以直接存取的存储单元。寄存器与功能单元相邻而居的事实邻近使得寄存器的充分利用成为执行时性能的重要因素。在已编译代码中，充分利用目标机器的寄存器集合的责任落在寄存器分配器上。

在程序的每一点，寄存器分配器决定哪些值将驻留在寄存器中，哪个寄存器将保存这些值中的每一个。如果分配器在一个值的生存期不能把它保存在寄存器中，那么在这个值的整个生存期或某一期间，分配器必须将它存储于内存中。分配器可能因代码所包含的活值比目标机器寄存器集合所能保存的活值要多，而把值移交到内存中。另外，在这个值的使用期间，分配器也许因不能证明它可以安全地驻留在寄存器中而把它保存在内存中。

619

寄存器分配器概念上把使用任意数量寄存器的程序作为它的输入。它产生一个适合目标机器寄存器单元的等价程序作为输出。



当分配器不能把某个值保留在寄存器中时，它必须把这个值存储到内存，并当它再一次被需要时装入它。这一过程称为溢出（spilling）这个值到内存。

寄存器分配器的一般目标是高效地使用目标机器提供的寄存器单元。这包括最小化实现溢出所要执行的装入和存储操作的数量。然而，其他目标也是可能的。例如，在一个内存受限环境中，用户也许希望分配器最小化分配的内存影响，即最小化保存被溢出值的数据存储器和保存溢出操作的代码存储器。

寄存器分配中的一个不好的决定会使某些应该驻留在寄存器中的值溢出。这样的额外溢出代码的代价将提升内存等待时间。因此，在20世纪90年代内存速度和处理器速度的差异增加了寄存器分配对编译代码性能的影响。

下一节回顾创建寄存器分配器操作环境的某些背景问题。以后各节讨论寄存器分配的算法以及局部作用域和全局作用域内的赋值算法。

13.2 背景问题

寄存器分配器取几乎完全编译完毕的代码为输入。这一代码已经经过扫描、语法分析、检查、分析和优化，并被重写为目标机器代码。它还可能要经过了调度。分配器必须通过插入在寄存器和内存之间移动值的操作，使这一代码适合目标机器的寄存器单元。编译器的早期阶段所做的大部分决定影响分配器的任务，就如同目标机器的指令集合的性质影响分配器的任务一样。本节揭示在寄存器分配器的角色形成中起着重要作用的因素。

620

13.2.1 内存与寄存器

编译器设计者所选择的内存模型（参见5.6.2节）定义了分配器必须处理的问题的很多细节。在寄存器到寄存器模型中，通过使所有歧义性值驻留在虚拟寄存器中，编译器的早期阶段直接把它们关于内存引用的歧义性信息编码到IR程序的形态中。IR程序中基于内存的那些值被假设是歧义的（参见7.2节），所以分配器把这些值留在内存中。

在内存到内存模型中，分配器没有代码形态这一线索。IR程序把所有值保存在内存中，而且当它们被使用和定义时，它把这些值移进、移出寄存器。分配器必须决定哪些值是非歧义的，因而可以安全地被保存于寄存器中。然后，它必须决定把这些值保存在寄存器是否是有益的。在这一模型中，分配器作为输入得到的代码一般使用较少的寄存器，且与等价的寄存器到寄存器代码相比执行更多的内存操作。为了得到好性能，分配器需要尽它所能把更多的基于内存的值提升到寄存器中。

因此，内存模型的选择从根本上决定分配器的任务。在上述两种情况下，分配器的目标都是减少最终代码在寄存器和内存之间要执行的把值移回、移进的装入和存储操作的数量。在寄存器到寄存器的模型中，分配是产生合法代码过程的必要部分；它确保最终代码适合目标机器的寄存器单元。分配器插入装入和存储操作把某些基于寄存器的值移到内存中，一般是在对寄存器的需求超出处理器的供给的范围内进行。分配器设法最小化它所插入的装入和存储操作的影响。

相反，使用内存到内存模型的编译器中的分配器是改进合法程序性能的一种优化。分配器决定把某些基于内存的值保存在寄存器中，从而使程序中的某些装入和存储成为不必要的。分配器设法消除尽可能多的装入和存储，因为这可以显著地改进最终代码的性能。

因此，信息的缺乏，即编译器分析中的限制，会使编译器无法把一个变量分配到寄存器中。当单一代码序列沿着不同路径继承不同的环境时，也可能出现这一情况。有关编译器可能了解的信息的这些限制倾向于使用寄存器到寄存器模型。寄存器到寄存器模型为编译器的其他部分提供编码关于歧义性和惟一性信息的机制。这一信息可能来自于分析；它也可能来自于对复杂结构的翻译的理解；甚至可能来自
621 于分析器中的源文本。

13.2.2 分配与赋值

在现代编译器中，寄存器分配器解决两个不同的问题：寄存器分配和寄存器赋值，这在过去是分别处理的。这两个问题有关联但不相同。

1）*分配* 寄存器分配把名字的非限定集合映射到目标机器提供的寄存器的特定集合上。在寄存器到寄存器模型中，寄存器分配把虚拟寄存器映射到一组模型化物理寄存器集合的新名字集合上，并溢出不适合这一寄存器集合的值。在内存到内存模型中，寄存器分配把内存位置的某个子集映射到模型化物理寄存器集合的名字集合上。分配保证代码在每一条指令都适合目标机器的寄存器集合。

2）*赋值* 寄存器赋值把已分配的名字集合映射到目标机器的物理寄存器上。寄存器赋值假设分配已执行完毕，所以这一代码将适合目标机器提供的物理寄存器集合。因此，在已生成代码中的每一个指令处，被指定驻留于寄存器中的值不超过k个，其中k是物理寄存器的数量。赋值产生可执行代码所需的真实寄存器名字。

在几乎所有的现实的形式中，寄存器分配都是NP完全的。对于所有数据值都有相同存储长度的单一基本块，最优分配可以在多项式时间内进行，只要把值存放回到内存的代价是统一的。这一问题中几乎任意的额外复杂性都使它成为NP完全问题。例如，把数据项的存储长度增加到两种，例如增加保存双精度浮点数的寄存器对，可以使这一问题成为NP完全。另外，增加一个真实的内存模型，或增加某些值无需存储回到内存的事实也会使这一问题成为NP完全。扩展分配的作用域来包含控制流和多个块

也使这一问题成为NP完全。在实践中，任意现实系统的编译中都会出现一个或多个这样的问题。在很多情况下，这些问题都会出现。

在很多情况下，寄存器赋值可以在多项式时间内解决。给定基本块的可行分配，即在这一分配中每一个指令处，对物理寄存器的需要不超过物理寄存器的数量，使用一个与区间图着色类似的方法可以在线性时间内完成一个赋值。一个完整过程的相关问题可以在多项式时间内解决，即如果在每一个指令处 622
对物理寄存器的需要不超过物理寄存器的数量，那么编译器可以在多项式时间内构造一个赋值。

分配和赋值之间的差异是微妙且重要的。在寻求改进寄存器分配器的性能中，编译器设计者必须弄清楚是分配中存在弱点还是赋值中存在弱点，并直接在这一算法的相应部分下功夫。

13.2.3 寄存器分类

大多数处理器所提供的物理寄存器不能形成可互换资源的同质池。大多数处理器对不同种类的值有不同的寄存器类。

例如，大多数现代计算机既有*通用寄存器*（general-purpose register）又有*浮点寄存器*（floating-point register）。前者保存整数值和内存地址，而后者保存浮点值。该二分类法并不新鲜；早期的IBM360机器有16个通用寄存器和4个浮点寄存器。现代处理器可能增加更多的类。例如，IBM/Motorola PowerPC有条件代码的独立寄存器类，而Intel IA-64有谓词寄存器和分支目标寄存器的附加类。编译器必须把每一个值放置到适当类的寄存器中。

如果两个寄存器类之间的互相作用是受限的，那么编译器也许能够独立地为它们分配寄存器。这把问题分解成更小且相互独立的组成部分，减小数据结构的大小，而且可能产生更快的编译时间。然而，当两个寄存器类重叠时，这两个类都必须被模型化为单一分配问题。把双精度浮点数保存在一对单精度寄存器中，这样常见的体系结构实践就是这一问题的一个很好的例子。双精度值类和单精度值类都映射到硬件寄存器的一组相同的集合上。在不考虑其中一个类的情况下，编译器无法分配另外一个类，所以它必须解决这一联合分配问题。

即使不同的寄存器类在物理上和逻辑上是独立的，它们也通过引用多个类中的寄存器的操作而相互作用。例如，对于很多体系结构，溢出一个浮点寄存器的决定要求插入一个地址计算和某些内存操作；这些动作使用通用寄存器，并改变那一类寄存器的分配问题。因此，编译器可以对不同的类做出相互独立的分配决定，但是这些决定的结果可能对其他寄存器类的分配产生影响。溢出一个谓词寄存器或一个 623
条件代码寄存器也有类似的效应。因此，通用寄存器应该在其他寄存器类之后分配。

13.3 局部寄存器分配和赋值

作为对寄存器分配的介绍，考虑为单一基本块产生好分配的过程中出现的问题。在优化中，处理单一基本块的方法被称为*局部*（local）方法，所以这些算法是局部寄存器分配技术。分配器取带有寄存器到寄存器存储模型的单一基本块为输入。

为了简化讨论，我们假设程序开始并结束于这一块；它不从较早执行的块中继承值，而且从不为较晚执行的块留下任何值。我们的输入程序将只使用通用寄存器的单一类。我们的目标机器将支持k个通用寄存器的单一集合。

代码形态编码关于哪些值可以合法地驻留于一个寄存器一段时间等信息。可以合法驻留在寄存器中的任意值被保存在一个寄存器中。代码使用任意多寄存器名字来编码这一信息，所以它可能命名比目标机器所有的寄存器更多的寄存器。出于这一原因，我们称这些预分配寄存器为*虚拟寄存器*（virtual register）。对于一个给定的块，它所使用的虚拟寄存器的数量是*MaxVR*。

这一基本块是由N个三地址操作o_1、o_2、o_3、…、o_N的序列组成的。每一个操作o_i都有形式$op_i\ vr_{i_1}$，$vr_{i_2} \Rightarrow vr_{i_3}$。表记法vr表明这些是虚拟寄存器而不是物理寄存器。从高层次观点看，局部寄存器分配的目标是创建一个等价块，在其中对一个虚拟寄存器的每一次引用都被对一个特定物理寄存器的引用所取代。如果MaxVR > k，那么分配器可能需要插入装入和存储以使代码适合k个物理寄存器的集合。这一性质的另外一种陈述是，在块的任意点，输出代码不能有多于k个值在寄存器中。

我们将给出两个解决这一问题的方法。第一个方法计数块中引用一个值的次数，并使用这一频率计数来决定哪些值驻留在寄存器中。因为它依赖于频率计算这样的额外获取的信息才能做出决定，我们认为这是一个自顶向下的方法。第二个方法依赖于代码的详细、低层信息来做出它的决定。它遍历这个块，并在每一个操作处决定是否需要一个溢出。因为它综合并组合很多低层事实来驱动做出决定的过程，我
624 们认为这是一个自底向上的方法。

13.3.1 自顶向下的局部寄存器分配

自顶向下局部分配器的工作基于一个简单的原则：最常用的值应该驻留在寄存器中。为了实现这一启发式探索法，它计数这个块中每一个虚拟寄存器的出现次数，并把这些频率计数作为把虚拟寄存器分配到物理寄存器的优先度。

如果虚拟寄存器比物理寄存器多，那么分配器必须保存若干个物理寄存器以便用于涉及分配到内存中的值的计算。分配器必须有足够多的寄存器来对两个操作数寻址和装入，执行这一操作，并存储其结果。所需要寄存器的精确数量依赖于目标体系结构；在典型的RISC机器上，这一数量可以是2～4个寄存器。我们将称这一机器特定数为可行（feasible）值。

为了执行自顶向下局部分配，编译器可以运用如下简单的算法：

1）*为每一个虚拟寄存器计算优先度。*线性遍历这个块中的操作，分配器可以记录每一个虚拟寄存器出现的次数。虚拟寄存器的计数值就是它的优先度。

2）*将虚拟寄存器按优先度顺序分类。*如果块适当地小，那么它可以使用一个桶分类，因为这样的计数必将落到一个小范围之内，也就是零到块长度的一个小倍数之间。

3）*按优先度顺序指定寄存器。*前k个可行虚拟寄存器被指定给物理寄存器。

4）*重写代码。*在第二次遍历代码中，分配器可以重写代码。对赋值给物理寄存器的虚拟寄存器的引用，用物理寄存器名字进行重写。对任意没有赋值给物理寄存器的虚拟寄存器的引用，使用对一个已保存的临时寄存器的引用取代它；并适当地插入一个装入操作或一个存储操作。

这一方法的优点是它把常用的虚拟寄存器保存在物理寄存器中。它的主要弱点在于分配的方法：贯穿整个基本块，它给一个虚拟寄存器指定一个物理寄存器。因此，在这个块的前半部分频繁使用但在其后半部分不复用的值，在整个后半部分仍然占据着物理寄存器，即使已不复用它。下一节给出解决这一
625 问题的技术。它采用根本上不同的方法来分配：一个自底向上、递增的分配方法。

13.3.2 自底向上的局部寄存器分配

自底向上局部寄存器分配器背后的关键思想是：集中考虑在每一个操作执行时所出现的转换。它开始于所有寄存器都不被占据。对于每一个操作，分配器需要保证这一操作的操作数在其执行之前在寄存器中。它还必须给这一操作的结果分配一个寄存器。图13-1给出分配器的基本结构以及它所使用的三个支持例程。

这一自底向上分配器在块内的操作上迭代，按需求进行分配决策。然而，*却存在某些微妙的地方。*通过依次考虑vr_{i_1}和vr_{i_2}，分配器避免为诸如add r_y，$r_y \Rightarrow r_z$这样的带有重复操作数的操作使用两个物理

寄存器。同样地，在分配r_z之前尝试释放r_x和r_y，避免当这一操作实际上释放一个寄存器时溢出一个寄存器来保存结果。大多复杂性都被隐藏于例程*Ensure*、*Allocate*和*Free*之中。

```
/* code for the allocator */
for each operation, i, in order from 1
    to N where i has the form
        op vr_i1 vr_i2 ⇒ vr_i3
    r_x ← ensure(vr_i1, class(vr_i1))
    r_y ← ensure(vr_i2, class(vr_i2))
    if vr_i1 is not needed after i
      then free(r_x,class(r_x))
    if vr_i2 is not needed after i
      then free(r_y,class(r_y))
    r_z ←allocate(vr_i3, class(vr_i3))
    rewrite i as op_i r_x,r_y ⇒ r_z
    if vr_i1 is needed after i
      then class.Next[r_x] ← dist(vr_i1)
    if vr_i2 is needed after i
      then class.Next[r_y] ← dist(vr_i2)
    class.Next[r_z] ← dist(vr_i3)

free(i,class)
    if (class.Free[i] ≠ true) then
      push(i,class)
      class.Name[i] ← −1
      class.Next[i] ← ∞
      class.Free[i] ← true

ensure(vr,class)
  if (vr is already in class)
    then result ← vr's physical register
  else
    result ← allocate(vr,class)
    emit code to move vr into result
  return result

allocate(vr,class)
  if (class.StackTop ≥ 0)
    then i ← pop(class)
  else
    i ← j that maximizes class.Next[j]
    store contents of j
  class.Name[i] ← vr
  class.Next[i] ← −1
  class.Free[i] ← false
  return i
```

图13-1 自底向上局部寄存器分配器

626

例程*Ensure*在概念上是简单的。它取两个参数，一个保存理想值的虚拟寄存器*vr*和一个适当寄存器类的表示*class*。如果*vr*已经占据一个物理寄存器，那么*Ensure*的工作已经结束。否则，它要为*vr*分配一个物理寄存器并发行把*vr*的值移到那个物理寄存器中的代码。无论哪种情况，它返回一个物理寄存器。

*Allocate*和*Free*揭示分配问题的细节。为了理解这两个例程，我们需要寄存器类的一个具体表示：如图13-2左边所示用C语言所写的代码。一个类有`Size`个物理寄存器，每一个都用虚拟寄存器名字表示(`Name`)；整数`Next`表示到这一虚拟寄存器的下一次使用的距离；标记`Free`表示这一物理寄存器当前是否在使用中。此图右边所示的代码初始化这个类的结构，使用－1作为超出范围名字，使用∞作为最大可能距离。为使*Allocate*和*Free*高效，这个类还需要一个自由寄存器的列表，这是`Class`中的`Stack`。例程*push*和*pop*操纵*Stack*。

```
struct Class {
  int Size;
  int Name[Size];
  int Next[Size];
  int Free[Size];
  int Stack[Size];
  int StackTop;
}
```

```
initialize(class,size)
  class.Size ← size

>class.StackTop ← −1
  for i ← 0 to size − 1
    class.Name[i] ← −1
    class.Next[i] ← ∞
    class.Free[i] ← true
    push(i,class)
```

图13-2 用C语言表示寄存器类

使用这一细节层次，*Allocate*和*Free*的代码就是直截了当的。每一个类维护自由物理寄存器的一个栈。当*class*的自由列表中存在自由的物理寄存器时，*Allocate*从该列表返回一个物理寄存器。否则，它选择*class*中后面最远使用的值，存储这个值，并为*vr*重新分配这一物理寄存器。分配设置*Next*域为－1，保证不为当前操作的其他操作数选择这一寄存器。分配器的主循环在完成当前操作后，将把它的
627 *Next*域重新设置为适当的值。*Free*把寄存器压入*stack*，并重新设置它在*class*结构中的域。

函数*dist*(*vr*) 返回对*vr*的下一次引用在这一块中的索引。编译器可以通过在这一块上做一次向后遍历来使用适当的*dist*值注释块中的每一个引用。

这一自底向上技术的纯效应是直截了当的。最初，它假设物理寄存器没有被占据，并把它们放置在自由列表上。对于前几个操作，它满足从自由列表分配物理寄存器的请求。当分配器需要另一个寄存器并发现自由列表为空时，它必须从寄存器中溢出一个值到内存中。它挑选出下一次使用在后面最远的值。只要溢出一个值的代价对所有寄存器都是相同的，那么这一选择释放空置时期最长的寄存器。在某种意义上，它最大化因溢出代价而得到的效益。

在实践中，这一算法产生优秀的局部分配。的确，有些设计者认为它产生最优分配。然而，在实践中也引发一些复杂性。在分配的任意点处，寄存器中的某些值可能因溢出而需要存储，而另外一些值则不需要。例如，如果寄存器包含一个已知的常量值，那么这一存储将是多余的，因为分配器无需内存中的一个拷贝就可重新创建这个值。同样地，由从内存的装入所生成的值无需存储。无需存储的值称为干净（clean），而需要存储的值称为不净（dirty）。

为了选择最优的局部分配，分配器必须考虑溢出干净值和溢出不净值在成本上的差异。例如，考虑两寄存器机器上的分配，在这里值x_1和x_2已经在寄存器中。假设x_1是干净值而x_2是不净值。如果这个块剩余部分的引用串是$x_3x_1x_2$，那么分配器必须溢出x_1或x_2中的一个。因为x_2的下一次使用在后面更远处，所以自底向上局部算法将溢出它，产生如下图左边所示的内存操作序列。

store x_2	
load x_3	load x_3 （重写x_1）
load x_2	load x_1
溢出不净值	溢出干净值

相反，如果分配器溢出x_1，它将产生上图中右边所示的内存操作序列，它少一个内存操作。这一情况表明，分配器应该优先溢出干净值而非不净值。答案却不是这样简单。

考虑相同初始条件下的另一个引用串$x_3x_1x_3x_1x_2$。持续溢出干净值可以产生下图左边所示的四内存操
628 作序列。

load x_3	
load x_1	store x_2
load x_3	load x_3
load x_1	load x_2
溢出干净值	溢出不净值

相反，持续溢出不净值将产生上图右边所示的序列，这一序列至少需要一个内存操作。考虑干净值和不净值之间的差异使局部分配问题成为NP完全问题。然而，在实践中，自底向上局部分配器产生好的局部分配；它们往往比前面所描述的自顶向下分配器产生更好的局部分配。

自底向上局部分配器在处理各个值的方式上不同于自顶向下局部分配器。自顶向下分配器在整个块把物理寄存器分配给虚拟寄存器，而自底向上分配器则把物理寄存器只在虚拟寄存器的两个连续引用之间指定给这一虚拟寄存器。它在对*Allocate*的每一次调用重新考虑所做出的决定，即在每一次需要另一

个寄存器时重新考虑。因此，自底向上算法能够而且的确产生一个这样的分配：其中单一虚拟寄存器在其生存期的不同点保存在不同的物理寄存器中。也可以为自顶向下分配器制定类似的行为。

13.4 移到超出单一块的范围

我们已经看到如何为单一块构建好的分配器。以自顶向下的方式，我们得到频率计数分配器。以自底下上的方式，我们得到自底向上局部分配器。我们可以利用自底向上局部算法所得到的经验改进频率计数分配器。然而，局部分配不能捕获跨越多个块间的值的复用。在实践中，这一情况经常出现。因此，下一步是把分配的作用域扩展到超过单一基本块的范围。

遗憾的是，从单一块到多个块的移动增加了复杂性。例如，我们的局部分配器隐式假设值不能在块间流动。移动到更大分配作用域的主要原因是：要考虑块间的值流向并生成高效处理这样的流动的分配。分配器必须正确地处理在前面的块中计算的值，而且它必须保存下一个块中使用的值。为了完成这一任务，与局部分配器相比，分配器需要更加复杂的处理“值”的方法。 629

13.4.1 活性和存活范围

区域和全局分配器都试图以协调跨越多个块的值的使用的方式把值指定给寄存器。这些分配器依据反映定义的真实模式以及每个值的使用的新名字空间来工作。它们不是把变量或值分配给寄存器，而是分配*存活范围*（live range）。单一存活范围由一组彼此相关的定义和使用组成，因为它们的值一起流动。也就是说，一个存活范围包含一个定义的集合和一个使用的集合。这一集合在下面的意义下是自含的：能够达到一个使用的每一个定义与这个使用在相同的存活范围内，而且一个定义能够达到的每一个使用与这个定义在相同的存活范围内。

术语*存活范围*隐式依赖于*活性*（liveness）的概念，正如9.2.1节中对活变量问题的讨论所解释的那样。回想一下，一个变量v在点p是*活的*（live），如果它沿着从这一过程的入口到p的路径中被定义，而且存在一条从p到v的使用的路径，在这一路径上v不被重定义。在v存活的任意地点，它的值必须被保留，因为后继执行可能使用v。请记住，v既可能是源程序的一个变量，也可能是编译器生成的临时名字。

存活范围的集合不同于变量集合和值的集合。代码中计算的每一个值都是某个存活范围的部分，即使它在原来的源代码中没有名字。因此，由寻址计算而产生的中间结果有存活范围，程序员命名的变量、数组元素以及用作分支目标为使用装入的地址也是如此。一个特定程序员命名变量可能有多个不同的存活范围。使用存活范围的寄存器分配器，可以把这些不同的存活范围放到不同的寄存器中。因此，源语言变量在不同点处可能驻留在执行程序中的不同的寄存器中。

为了使这些思想更具体，考虑在单一基本块内寻找存活范围的问题。图13-3给出图1-3的块。我们加入了一个初始操作来定义r_{arp}。图13-3的右边所示的表格给出这个块中的不同存活范围。在直线代码中，我们可以把一个存活范围表示成一个区间。注意，每一个操作定义一个值，因此它开始一个存活范围。考虑r_{arp}。它是在操作1定义的。对r_{arp}的每一个其他引用都是一个使用。因此，这个块只使用r_{arp}的一个值。区间[1，11]表示这一存活范围。

相反，r_w有若干个存活范围。操作2定义r_w；操作7使用操作2定义的值。操作7、8、9和10中的每一个都定义一个新的r_w值；对于每一种情况，后面的操作使用新定义的值。因此，图中命名为r_w的寄存器对应于5个不同的存活范围：[2，7]、[7，8]、[8，9]、[9，10]和[10，11]。寄存器分配器无需在同一个物理寄存器中保存这些不同的存活范围。相反，它可以把块中的每一个存活范围作为独立的值来分配和赋值。 630

```
1   loadI   ...          ⇒ r_arp
2   loadAI  r_arp, 0     ⇒ r_w
3   loadI   2            ⇒ r_2
4   loadAI  r_arp, @x    ⇒ r_x
5   loadAI  r_arp, @y    ⇒ r_y
6   loadAI  r_arp, @z    ⇒ r_z
7   mult    r_w, r_2     ⇒ r_w
8   mult    r_w, r_3     ⇒ r_w
9   mult    r_w, r_y     ⇒ r_w
10  mult    r_w, r_z     ⇒ r_w
11  storeAI r_w          ⇒ r_arp, 0
```

定义寄存器	区间
r_{arp}	[1,11]
r_w	[2,7]
r_w	[7,8]
r_w	[8,9]
r_w	[9,10]
r_w	[10,11]
r_2	[3,7]
r_x	[4,8]
r_y	[5,9]
r_z	[6,10]

图13-3 基本模块中的存活范围

为了在比单一块大的区域内寻找存活范围，编译器必须执行活变量分析，以发现在每一块的出口是活着的值的集合。它可以使用得自于这一分析的集合LIVEOUT(b)以及包含在通向b的入口是活着的变量的集合LIVEIN(b)来注释每一个块b。而LIVEIN(b)恰好是UEVAR(b) ∪ (LIVEOUT(b) ∩ $\overline{\text{VARKILL}(b)}$)。

在代码的任意一点，不是活着的值不需要寄存器。同样地，只有在点p需要寄存器的那些值才是在点p活着的值，或是这些活着的值的某个子集。真实编译器中实现的局部寄存器分配器使用LIVEOUT集合来决定何时一个值必须在它在一个块的最后使用之前被保存在内存中。全局分配器使用类似的信息来发现存活范围并引导分配过程。

13.4.2 块边界处的复杂性

使用局部寄存器分配的编译器也许要为每一个块计算LIVEIN和LIVEOUT，来作为向局部分配器提供有关这一块的入口和出口的值的状态信息的必要前奏。当控制流从一个块流向另一个块时，这些集合的存在简化使各个块的分配正确行动的任务。例如，LIVEOUT(b)中的值在它在b中的最后一次定义之
631 后必须存储于内存中，以确保当这个值在后继块中被装入时，它将处于期望的位置上。相反，如果这个值不在LIVEOUT(b)中，那么它无需存储，除非是作为b内部后来使用的一个溢出。

由于考虑多个块而引入的某些效应复杂化赋值或分配，或者同时使二者复杂化。图13-4给出出现于全局赋值中的某些复杂性。考虑沿从块B_1到B_3的边上发生的转换。

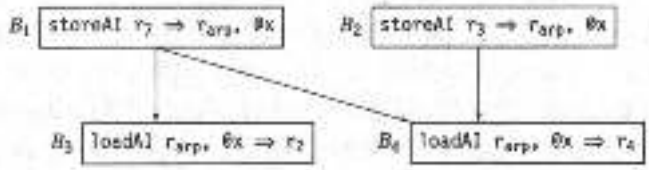


图13-4 多个块中存在的问题

B_1在r_7中有程序变量x的值，B_3在r_2中需要它。当局部分配器处理B_1时，它没有其他块所创建的上下文信息，所以它必须把x存储回x在内存中的位置（在r_{arp}中距ARP偏移为@x的位置）。同样地，当分配器处理B_3时，它没有关于B_1的行为信息，所以它必须从内存装入x。当然，如果它知道B_1上的分配结果，那么它可以把x赋给r_7并且使这一装入不再必要。在缺乏这方面的信息时，它必须生成这一装入。在B_2和

B_4中对x的引用进一步复杂化这一问题。在块间协调x的赋值的任意企图都必须考虑这两个块，因为B_4是B_1的后继，而且在B_4中对x的处理的任何变化都将对它的另一个前驱B_2产生影响。

分配也引发类似的效应。如果x在B_2中不被引用将如何呢？即使我们可以全局地协调赋值，保证使用x时它总是在r_7中，分配器还是需要在B_2的末端插入x的一个装入，使得B_4避开对x的初始装入。当然，如果B_2有其他后继，那么它们可能不引用x，而需要r_7中的另外一个值。

块边界处的这些效应可以变得越发复杂。它们不适合局部分配器，因为它们处理完全在局部分配器作用域之外的现象。如果分配器设法插入几个额外的指令来消除这一差异，那么它可能选择在错误的块中插入它们。例如，在一个形成内部循环体的块中，而不是在那个循环的头部块中插入它们。局部模型假设所有指令以相同的频率执行；拓宽这一模型处理更大区域也使这一假设无效。好坏分配之间的差异 632
可能在于代码中最经常执行块中的几个指令。

当我们设法把局部分配范例拓宽到超出单一块的范围时，引发第二个更微妙、更令人困惑的问题。考虑在块B_1上使用自底向上局部算法。仅使用一个块时，“最远的”下一次引用的概念很清晰。局部算法对每一个下一次引用有惟一的距离。对于多后继块，分配器必须考虑沿着所有离开B_1的路径的引用。对于B_1中的某个值y的最后引用，下一次引用或者是B_3中对y的第一次引用，或者是B_4中对y的第一次引用。这两次引用很少会在相对于B_1的尾部相同的地方。另外，B_3可能不包含对y的引用，而B_4却包含一个这样的引用。即使两个模块都使用y，而且引用是在输入代码中距离相同的地方发生的，在一个块中的局部溢出也可能使它们以某种不可预测的方式产生差异。当构成自底向上局部方法的基本度量变成多值时，算法的效应变得更难理解、更难证实。

所有这些问题表明，我们需要一个全新的方法来把局部分配转换到区域分配或全局分配。的确，成功的全局分配算法与局部分配算法很少有相似之处。

13.5 全局寄存器分配和赋值

寄存器分配器的目标是最小化它必须插入的指令所需的执行时间。分配器不能保证为这一问题提供最优解。从执行时间的角度看，同一基础代码的不同分配的差异在于分配器插入的装入、存储和拷贝操作的数量的不同，以及它们在代码中的放置不同。因为不同的块执行的次数不同，所以溢出的放置对执行溢出代码所需的时间总量产生强烈的影响。因为块执行频率随运行而发生变化，最佳分配的概念在某种程度是空洞的，它必须以执行频率的特定集合为条件。

全局分配在以下两个基本方面不同于局部分配。

1）存活范围的结构比局部分配器中的结构更复杂。在单一块中，存活范围只是线性操作串中的一个区间。在全局分配器中寻找存活范围更加复杂。一个全局存活范围是由定义和使用这两个关系的闭包 633
所建立的定义和使用的网络。对于存活范围内的每一个使用，它包含能够达到那个使用的每一个定义。对于活范围内的每一个定义，它包含那个定义能达到的每一个使用。

2）对于同一变量的不同引用执行的次数可能不同。在单一块中，如果任意操作执行，那么所有操作都执行（除非出现异常），所以溢出的代价是一致的。在更大的作用域内，每一个引用可能处于不同的块中，所以溢出的代价取决于引用的位置。当全局分配器必须溢出时，它应该考虑成为溢出候选的每一个存活范围的溢出代价。

任意全局分配器必须处理这两个问题。这使全局分配着实比局部分配更复杂。

为了处理复杂的存活范围问题，全局分配器显式创建一个名字空间，在这一空间内每一个不同的存活范围有不同的名字。然后，分配器把一个存活范围映射到一个物理寄存器或内存位置上。为了实现这一点，全局分配器首先构建存活范围并重命名代码中所有虚拟寄存器引用，以反映已构建存活范围的新

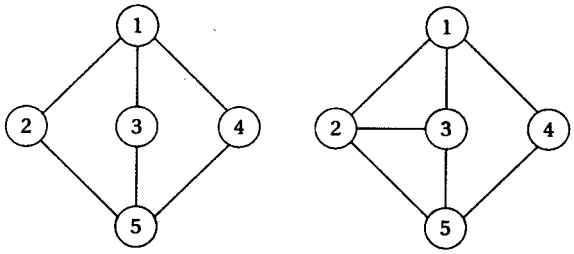
名字空间。为了处理执行频率问题，分配器可以使用一个评估执行频率来注释每一个引用或每一个基本块。这一评估可能来自于静态分析，或来自于程序的实际执行期间所收集的侧面数据。这个评估执行频率被用于引导分配器来做出关于分配和溢出的决策。

最后，全局分配器必须做出关于分配和赋值的决策。它们必须决定哪些存活范围将驻留在寄存器中。它们必须决定两个存活范围是否可以共享一个寄存器。它们必须把特定物理寄存器赋给被分配的每一个存活范围。

很多全局分配器使用图着色范例。它们构建一个图来模型化存活范围间的冲突，并为这个图寻找适当的着色。着色被用于模型化寄存器赋值；它使分配器知道什么时候两个值不能共享一个寄存器。不同的着色分配器以不同的方法处理分配（或溢出）。我们将审视使用高级信息做出分配决策的自顶向下分配器、以及使用低级信息做出那些决策的自底向上分配器。然而，在考虑这两个方法之前，我们揭示分
634 配器共有的某些次要问题：发现存活范围、估测溢出代价及构建冲突图。

图着色

很多全局分寄存器分配器使用图着色（graph coloring）作为范例模型化基本分配问题。对于任意的图G，G的一个着色给G中的每一个结点指定一种颜色，使得相邻结点对没有相同颜色。使用k种颜色的一种着色称为一个k着色，而且对于给定图的这样的最小的k称为这个图的着色数（chromatic number）。考虑下面的图：



左侧的图是2着色的。例如，给结点1和5指定蓝色，给结点2、3和4指定红色产生所需的结果。增加结点2和结点3之间的边，如右图所示使此图成为3着色，而不是2着色。（给结点1和结点5指定蓝色，结点2和结点3指定红色，给结点3指定白色。）

对于一个给定图，寻找它的着色数问题是NP完全的。类似地，对于某个固定的k，确定一个图是k着色的问题是NP完全的。使用图着色作为分配资源范例的算法利用近似方法来寻找适合可用资源集合的着色。

13.5.1 发现全局存活范围

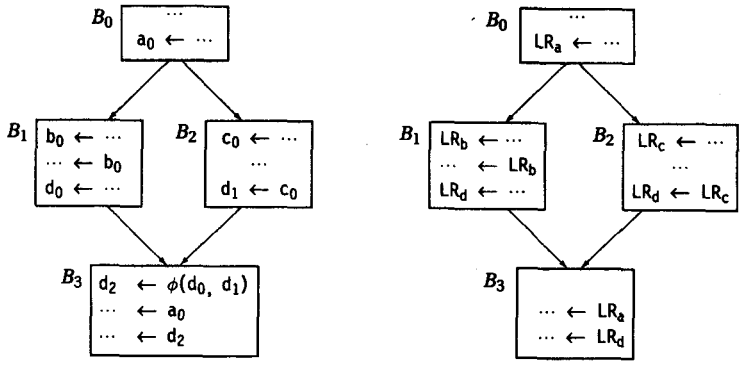
为了构建存活范围，编译器必须发现不同定义和使用之间存在的关系。分配器必须得到一个名字空间，这一空间必须是集成达到单一使用的所有定义以及单一定义所达到的所有使用的单一名字空间。这提出一个方法，使用这一方法编译器给每一个定义指定一个不同的名字并把达到一个公共使用的名字合
635 并到一起。

代码的SSA形式给出这一构造法的一个自然的开始点。回想一下，在SSA形式中，每一个名字只被定义一次，每一个使用只引用一个定义。为协调这两个规则插入的ϕ函数记录在控制流图中不同的路径

上的不同定义达到一个引用的事实。流向一个ϕ函数的两个或多个定义属于同一个存活范围，因为ϕ函数创建一个代表所有值的名字。引用ϕ函数结果的任意操作使用其中的一个值；这个值依赖于控制流如何达到这一ϕ函数。因为可以通过相同的使用引用所有定义，所以这些定义必须驻留在相同的寄存器中。因此，ϕ函数是构建SSA形式的代码的存活范围的关键。

为了从SSA形式构建存活范围，分配器使用不相交集合的并集寻找算法，并在代码之上做一次遍历。分配器把每一个SSA名字或定义作为算法中的一个集合来处理。分配器检查程序中的每一个ϕ函数，并把与每一个ϕ函数参数相关的集合与ϕ函数目标的集合合并起来。当所有这些ϕ函数已被处理过后，其结果集合代表代码中的存活范围。此时，分配器可以或者重写使用存活范围名字的代码，或者创建并维护SSA名字与存活范围之间的一个映射。

图13-5左侧给出包含源代码变量a、b、c和d的半剪枝SSA形式中的一个代码片段。为了寻找存活范围，分配器给每一个SSA名字指定一个包含这个名字的集合。它合并与在ϕ函数中使用的名字相关的集合，$\{d_0\} \cup \{d_1\} \cup \{d_2\}$。其结果给出四个存活范围的集合：$LR_a$包含$\{a_0\}$，$LR_b$包含$\{b_0\}$，$LR_c$包含$\{c_0\}$和$LR_d$包含$\{d_0, d_1, d_2\}$。该图的右侧给出使用这些存活范围重写名字而得的代码。 636



剪枝SSA形式中的代码片段　　使用存活范围重写之后

图13-5　发现存活范围

在9.3.5节，我们看到，运用于SSA形式的转换可能将复杂因素引入这一重写过程中。如果分配器构建SSA形式，并使用这一形式来发现存活范围，那么这一重写过程可以使用存活范围名字取代SSA名字。另一方面，如果分配器使用已经过转换的SSA形式，那么这一重写过程必须处理9.3.5节所述的复杂问题。因为大多数编译器在指令筛选和指令调度之后执行分配，所以分配器消耗的代码将不是SSA形式。这迫使分配器构建这一代码的SSA形式并保证重写过程是直截了当的。

13.5.2　评估全局溢出代价

为了做出有意义的溢出决定，全局分配器需要对溢出每一个值的代价做出评估。一个溢出的代价有三个组成部分：地址计算、内存操作和评估执行频率。

编译器设计者可以在内存中选择溢出值的驻留位置。这个溢出值一般被保存在当前的活动记录（AR）中，特别是如图6-5所示的寄存器存储空间，以便最小化地址计算的代价。在AR中存储溢出值使分配器为了溢出生成诸如`loadAI`或者`storeAI`等与r_{arp}相关的操作。这样的操作通常避免为了计算溢出地址对更多寄存器的需求。

内存操作的代价一般是不可避免的。如果目标机器有非缓冲片内局部内存，正如很多嵌入式处理器所拥有的那样，那么编译器可以为溢出使用那个内存。更一般地，编译器需要保存这个值，而当后面的

操作需要这个值时从内存中恢复它。随着内存等待时间的增加，所需的load和store操作的代价也增加。使情况在某种程度上变得更坏的是，只有当分配器绝对需要一个寄存器时，它才插入溢出操作。因此，很多溢出操作出现在对寄存器的要求很高的代码中，这将反过来阻止调度器把这些操作移到足够远的地方以隐藏内存等待时间。编译器设计者必须期望溢出位置停留在高速缓存中。（自相矛盾的是，仅当这
637 些位置被反复存取时它们才停留在高速缓存中以避免替换：这说明这一代码正在执行太多的溢出。）

1. 负溢出代价

对于包含一个装入、一个存储而且没有其他使用的存活范围，如果这一装入和存储引用相同的地址，那么这个存活范围应该得到一个负溢出代价。（这样的存活范围可能产生于为了改进代码的转换，例如，如果使用被优化掉，而且这一存储产生于一个过程调用而不是一个新值的定义。）有时候，溢出一个存活范围可以消除比溢出操作代价更高的拷贝操作；这样的存活范围也有一个负代价。带有负溢出代价的任意存活范围都应该溢出，因为这样做降低对寄存器的需求并且从代码中移出指令。

2. 无限溢出代价

有些存活范围很短而使得溢出它们没有任何帮助。考虑紧跟在其定义后的一个使用。溢出这一定义和使用将产生两个短存活范围。第一个存活范围包含这个定义后面跟着一个存储；第二个存活范围包含一个装入后面跟着这一使用。这两个新的存活范围的任意一个所使用的寄存器都不少于原来的存活范围所使用的寄存器，所以这一溢出不产生效益。分配器应该给原来的存活范围赋一个无限的溢出代价。一个存活范围一般应该有无限的溢出代价，如果没有其他存活范围在它的定义和使用之间结束。这一条件保证寄存器的可用性在这一定义和使用之间不发生改变。

3. 计数执行频率

为了计数控制流图中基本块的不同执行频率，编译器应该使用一个评估执行计数注释每一个块（如果不是每一个引用）。大多数编译器使用简单的启发式搜索来评估执行代价。一个通常的方法就是假设每一个循环执行十次。因此，它把计数10赋给循环内部的一个装入，并把计数100赋给两个嵌套循环内的一个装入。一个不可预测的if-then-else可以把执行计数降低到一半。在实践中，这些评估保证倾向于外循环的溢出而不是内循环的溢出。

为了评估溢出单一引用的代价，分配器对地址计算代价和内存操作代价求和，并将这个和与这一引
638 用的评估执行频率相乘。对于每一个存活范围，它求各个引用的代价和。这需要对这一代码中的所有块做一次遍历。分配器可以预先计算所有存活范围的代价，它也可以等待直到发现它必须至少溢出一个值。

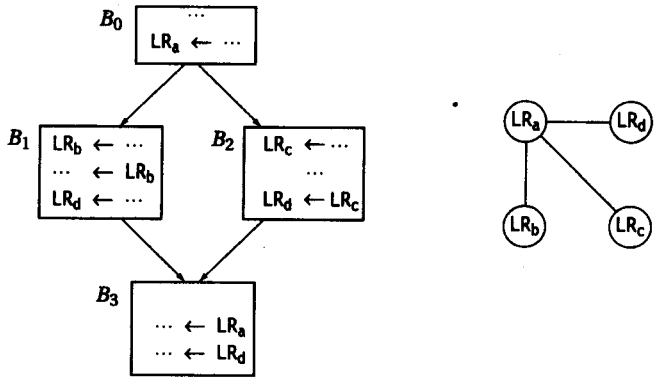
13.5.3 冲突和冲突图

一个全局寄存器分配器必须模型化的一个基本效应是：在目标寄存器集合内各值对空间的竞争。考虑两个不同的存活范围LR_i和LR_j。如果在程序中存在一个操作，在这一操作期间，LR_i和LR_j都是活的，那么它们不可能驻留在相同的寄存器中。（一个物理寄存器一般在一个时刻只能保存一个值。）我们说LR_i和LR_j冲突（interfere）。形式地，LR_i和LR_j冲突，如果其中一个在另一个的定义处是活着的且它们有不同的值。

为了模型化这一分配问题，编译器可以构建一个冲突图$I=(N, E)$，其中，N中的结点表示各个存活范围，而E中的边表示存活范围之间的冲突。因此，一个无向边$(n_i, n_j)\in I$存在，当且仅当相应的存活范围LR_i和LR_j冲突。图13-6给出图13-5的代码以及它的冲突图。LR_a与其他每一个存活范围都冲突。然而，其余的存活范围互不冲突。

如果编译器可以使用k或更少的颜色着色I，那么它可以把这些颜色直接映射到物理寄存器上来产生

合法分配。在本例中，LR_a不能与LR_b、LR_c和LR_d有相同的颜色，因为它与它们中的每一个冲突。然而，其他三个存活范围可以共享一种颜色，因为它们互不冲突。因此，冲突图是2着色的，而且可以重写代码使其恰好使用两个寄存器。 639



带有存活范围的代码片段　　冲突图

图13-6　存活范围和冲突

考虑，如果编译器的另一个阶段重排序B_1末端的两个操作将会发生什么。这使LR_b在LR_d的定义处是活着的。分配器必须把边（LR_b，LR_d）加到E中，使得仅使用两种颜色无法着色这个图。（因为这一图很小，因此通过枚举就可以证明这一点。）为了处理这个图，分配器有两个选择：使用三个寄存器，或者，如果目标机器只有两个寄存器，那么在B_1中LR_d的定义之前溢出LR_b或LR_a中的一个。当然，这一分配器也可能重新排序这两个操作，消除LR_b与LR_d之间的冲突。寄存器分配器一般不重排序操作。相反，分配器假设操作有一个固定的顺序，并将排序问题留给指令调度器（参见第12章）。

1. 构建冲突图

一旦分配器构建了全局存活范围，并使用它的LIVEOUT集合注释代码中的每一个基本块，它就能够通过对每一个块的简单线性遍历构造冲突图。图13-7给出这一基本算法。当分配器自底向上遍历这个块时，它计算在当前操作中活着的值的集合。因为在这个块的最后操作处LIVENOW必须等于这个块的LIVEOUT集合，所以分配器把LIVENOW初始化为LIVEOUT。随着分配器向后遍历这一块，它把适当的边加到E中并更新LIVENOW。

```
for each LR i
    create a node n_i ∈ N
for each basic block b
    LIVENOW ← LIVEOUT (b)
    for o_n, o_n-1, o_n-2, ... o_1 in b
        with form op_1 LR_a, LR_b ⇒ LR_c
        for each LR_i in LIVENOW
            add (LR_c, LR_i) to E
        remove LR_c from LIVENOW
        add LR_a & LR_b to LIVENOW
```

图13-7　构建冲突图

这一算法实现早前所给的冲突的定义：LR_i和LR_j冲突，当且仅当其中一个在另一个的定义处是活着的。这一定义允许编译器通过在每一个操作处在操作LR_c的目标与这一操作之后活着的每一个存活范围之间加入一个冲突来创建冲突图。 640

拷贝操作需要特殊处理。操作121　$LR_i \Rightarrow LR_j$不在LR_i与LR_j之间创建冲突，因为这些值可以占据相同的寄存器。这一操作不应该包含边（LR_i，LR_j）。如果其后的某个上下文迫使在这些存活范围之间有一个冲突，那么那个操作将创建这个边。相同的讨论适用于ϕ函数。用这种方法处理拷贝和ϕ函数使分配器得到何时LR_i和LR_j占据相同寄存器的精确信息。结果的冲突图可以充当组合存活范围，并消除不必要的拷贝的强有力的结合阶段的基础（参见13.5.6节）。

为了改进分配器后面工作的效率，编译器应该同时构建表示E的下三角位矩阵和邻接表集合。位矩

阵允许对冲突做常量时间测试，而邻接表使在结点的相邻结点上的迭代更高效。这比使用单一表示占据更多的空间，但是分配器高频率地执行这两个操作，使得这一代价很值得。编译器设计者也可以把不相交寄存器类当作独立的分配问题来处理，从而减小E的尺寸和整体分配时间。

2. 构建分配器

为了构建基于图着色范例的全局分配器，编译器设计者需要两个额外的机制。第一，分配器需要发现k着色的技术。遗憾的是，对于特定图确定它是否存在一个k着色是NP完全问题。因此，寄存器分配器使用一个无法保证找到k着色的快速近似方法。第二，分配器需要一个策略来处理对于一特定的存活范围没有剩余颜色的情况。大多数着色分配器通过重写代码改变这一分配问题来逼近这样的分配器。这样的分配器挑选出一个或多个存活范围加以修改。它或者溢出或者分割被选择的存活范围。溢出把被选择存活范围变成多个微小存活范围，围绕着原来的存活范围中的每一个定义和使用有一个这样的微小存活范围。分割把每一个存活范围分成更小是非平凡的存活范围。无论是哪一种情况，转换后的代码执行相同的计算，但是有不同的冲突图（包括更多的装入和存储）。如果这些改变有效，那么新的冲突图是k着
641 色的。如果这些改变无效，那么分配器必须溢出更多存活范围。

13.5.4 自顶向下着色

自顶向下图着色全局寄存器分配器使用低级信息给各个存活范围指定颜色，并使用高级信息选择它着色存活范围的顺序。为了寻找特定存活范围LR_i的颜色，分配器记录已经指定给LR_i在I中的邻居的颜色。如果邻居的颜色集合不是完全的，即有一种或多种颜色没有被使用，那么分配器可以把一种没有使用的颜色指定给LR_i。如果邻居的颜色集合是完全的，那么没有颜色适合LR_i，而且分配器必须对未着色存活范围使用它的策略。

自顶向下分配器设法以由某个等级函数所确定的顺序着色存活范围。基于优先度的自顶向下分配器给每一个结点指定一个等级，这一等级是由把该存活范围保存在寄存器中而积累起来的评估运行时间的节省。这些评估与13.5.2节所描述的溢出代价类似。自顶向下全局分配器为由这些等级定义的最重要的值使用寄存器。

分配器按等级顺序考虑存活范围，并试图为其中的每一个存活范围指定一种颜色。如果没有颜色适合一个存活范围，那么分配器调用溢出或分割机制处理这一未着色的存活范围。为了改进这一过程，分配器可以把存活范围划分成两个集合：限定存活范围和非限定活范围。一个存活范围是限定的(constrained)，如果它有k个或更多邻居，即它在I中的度$\geq k$。（我们记“LR_i的度”为LR_i^0，所以LR_i是限定的当且仅当$LR_i^0 > k$。）限定存活范围首先按等级顺序被着色。在所有限定存活范围被处理之后，非限定存活范围按任意顺序被着色。因为一个非限定存活范围的邻居少于k个，所以分配器总可以为它找到一种颜色；对它的邻居的颜色指定不能使用所有k种颜色。

通过首先处理限定活范围，分配器避免某些可能的溢出。以直接的优先顺序工作的另一个方法，可以使分配器把所有可能的颜色指定给非限定但是有较高优先度的LR_i的邻居。这可能迫使LR_i不被着色，即使一定存在它的非限定邻居的着色，使得存在适合LR_i的颜色。

1. 处理溢出

当自顶向下分配器遇到一个不能被着色的存活范围时，它必须或者溢出或者分割存活范围的某个集
642 合来改变这一问题。因为所有前面已着色的存活范围的等级高于没有着色的存活范围，溢出没有着色的存活范围比溢出前面已着色的存活范围更有意义。分配器可以考虑重新着色前面已着色的存活范围中的一个，但是它必须尽可能避免完全的一般性以及回溯的代价。

为了溢出LR_i，分配器在LR_i的每一个定义之后插入一个存储，并在LR_i的每一个使用之前插入一个装

入。如果内存操作需要寄存器，那么分配器可以保留足够的寄存器来处理它们。（例如，当一个溢出值在使用前被装入时，需要一个寄存器来保存这个溢出值。）出于这一目的所需的寄存器的数量是目标机器指令集合系统的一个函数。保留这些寄存器简化溢出。

为了溢出代码而保留寄存器的另一种选择是：在每一个定义和使用处寻找自由颜色；这一策略可以导致这样一种状况，在这一状况中分配器必须逆向溢出一个前面已着色的存活范围。⊖当然，自相矛盾的地方是：为溢出保留寄存器本身可能通过从效果上降低k而引发溢出。

2. 分割存活范围

溢出改变着色问题。一个未着色的存活范围被分割成一系列微小存活范围，它们是如此之小使得溢出它们是没有成效的。改变这一问题的一个相关方法是：取一个未着色存活范围并把它划分成比单一引用大的小块。如果与原来的存活范围相比，这些新的存活范围与较少的存活范围相互冲突，那么分配器可能为它们找到颜色。例如，如果新存活范围是非限制的，那么一定存在适合它们的颜色。被称为*存活范围分割*（live-range splitting）的这一过程可以导致一个分配，这一分配插入的装入和存储比溢出整个存活范围所需的要少。

由Chow构建的第一个自顶向下、基于优先度的着色分配器把未着色存活范围划分成单一块存活范围，记数每一个结果存活范围的冲突度。然后，当重新组合相邻块中的这些存活范围仍为非限制存活范围时，重新组合这些相邻块的存活范围。它对分割存活范围能够跨越的块数量设置一个任意的上限。它在每一个分割存活范围的开始点插入一个装入，并在这个存活范围的结束端插入一个存储。分配器溢出仍然未着色的任意分割存活范围。 643

13.5.5 自底向上着色

自底向上图着色寄存器分配器使用很多与自顶向下全局分配器相同的机制。这些分配器发现存活范围，构建一个冲突图，试图去着色它，并当需要时生成溢出代码。自顶向下和自底向上分配器的主要差异在于用于排序着色存活范围的机制不同。自顶向下分配器使用高级信息来选择着色顺序，而自底向上分配器根据关于冲突图的详细结构信息计算顺序。这样的分配器构造考虑存活范围的线性顺序，并按这一顺序制定颜色。

为了排序存活范围，自底向上图着色分配器依赖于非限定存活范围对颜色是平凡的这一事实。这一事实在构建颜色赋值顺序中起着重要的作用。图13-8给出一个计算这样的顺序的简单算法。分配器反复地从图中移出一个结点，并把这一结点放到栈上。它使用两个不同的机制来选择下一次移出的结点。第一个子语句取在移出的图中非限定的结点。因为这些结点是非限定的，所以它们移出的顺序不重要。移出一个非限定结点降低这一结点的每一个邻居的度，也有可能使它们也成为非限定的。第二个子语句只有当每一个余下的结点都是限定时才被使用，它使用某个外部标准挑选出一个结点。当这一循环停止时，此图是空的，而且栈包含按移出顺序排列的所有结点。

```
initialize stack to empty
while (N ≠ ∅)
  if ∃ n ∈ N with n° < k
    then node ← n
    else node ← n picked from N
  remove node and its edges from I
  push node onto stack
```

图13-8 计算自底下上顺序

为了着色这个图，分配器按由栈所表示的顺序重新构建冲突图，即按分配器从图中移出结点的相反顺序。它反复地从栈中弹出结点n，把n和它的边插入回I中，并寻找适合n的颜色。使用伪码的这一算法

⊖ 分配器将在每一个溢出场所重新计算冲突，并计算这一溢出场所的邻居的颜色集合。如果这一过程没有在每一个溢出场所发现可用的颜色，那么分配器溢出这一溢出场所的最低优先度的邻居。递归溢出已着色存活范围的可能性导致了自顶向下的、基于优先度的分配器的大多数实现保留溢出寄存器。

如下所示。

```
while (stack ≠ ∅)
  node ← pop(stack)
  insert node and its edges into I
  color node
```

为了着色结点n，分配器记录n在当前I的近似中的邻居的颜色，并给n指定一种未使用的颜色。[⊖]如
644 果没有颜色留给了n，那么n保持未着色。

当栈为空时，I已经重建完毕。如果每一个结点都有一种颜色，那么分配器就宣布成功并重写代码，使用物理寄存器替换存活范围名字。如果任意结点仍然未着色，那么分配器或者溢出相应的存活范围或者把它分割成小片。此时，典型的自底向上分配器重写代码以反映这样的溢出和分割，并重复整个过程：寻找存活范围，构建I并着色它。这一过程反复进行，直到I中每一个结点得到一种颜色。分配器一般在若干次迭代中停止。当然，自底向上分配器将为溢出保留寄存器，就如自顶向下分配器所描述的那样。这一策略允许它在一次遍历之后停止。

为什么这一方法起作用

自底向上分配器把每一个结点插回它被移出的图中。如果表示LR_i的结点因为当时它是非限定的而被从I中移出，那么当它被重新插回I的一个近似中去时，在这一近似中它仍然是非限定的。因此，当分配器插入LR_i时，它一定有一个可用于LR_i的颜色。得不到颜色的惟一结点是使用图13-8中的*else*子语句中的溢出度量被从I中移出的结点。这些结点被插入它们有k个或更多个邻居的图中。可能存在一种适合它们的颜色。当分配器把结点n插入I中时，假设$n^0 > k$。它的邻居不能都有不同的颜色，因为至多有k种颜色。如果它们刚好有k种颜色，那么分配器找不到可用于n的颜色。相反，如果它们使用的颜色小于k，那么分配器可以找到一种可用于n的颜色。

这一移出过程决定结点被着色的顺序。这一顺序非常重要，以这一顺序，它确定有无可用的颜色。对于那些由于是非限定的而从图中移出的结点，对于余下的结点来说顺序并不重要。这一顺序可能对于
645 已在栈中的结点很重要；毕竟，当前结点可能是限定的，直到早前的一些结点被移出。对于由图13-8中的*else*语句被从图中移出的结点，这一顺序是关键的。只有当每一个余下的结点都是限定结点时，这一语句才执行。因此，余下的结点形成一个或多个I的稠密连结的子图。

用于挑选结点的启发式搜索通常被称为*溢出度量*（spill metric）。由Chaitin等构建的原始的自底向上图着色分配器使用一种简单的溢出度量。它挑选出最小化比率*cost*/*degree*的结点，其中*cost*是估测溢出代价，而*degree*则是在当前图中的度。这一度量挑选出溢出成本相对较低的结点，但降低很多其他结点的度。人们还尝试了很多其他溢出度量，其中包括*cost*/*degree*^{2}，这一度量强调对邻居的影响；强调运行时速度的*cost*本身；以及最小化溢出操作数量，强调代码的大小的度量。前两个*cost*/*degree*和*cost*/*degree*^{2}试图平衡代价和影响，而后两个*cost*和溢出操作的目的在于优化特定标准。在实践中，没有哪一个启发式搜索占绝对优势。因为实际的着色过程相对于构建I更快，所以分配器可以尝试若干种着色，每一种使用不同的溢出度量，并保留最好的结果。

13.5.6 接合存活范围以降低度

编译器设计者可以构建一个强大的接合阶段，使用冲突图来决定何时通过拷贝相连的两个存活范围可以接合（coalesce）起来。考虑操作121 $LR_i \Rightarrow LR_j$。如果LR_i和LR_j不冲突，那么这一操作可以被消除，

⊖ 为了挑选出一个特定颜色，每一次它可以以一个相容的顺序进行搜索，或者它可以用循环的形式指定颜色。以我们的经验，用于颜色选择的机制没有什么实际意义。

而所有对LR_j的引用都可以被重写成对LR_i的引用。接合这些存活范围有若干有益的效应。它消除这一拷贝操作，使得代码更小且很有可能更快。它降低与LR_i和LR_j都冲突的任意LR_k[⊖]的度。它缩小存活范围的集合，使得I及许多与I相关的数据结构变小。（在Briggs的论文中，他给出接合消除三分之一存活范围的例子。）因为这些效应有助于分配，所以编译器通常在全局分配器中的着色阶段之前执行接合。

图13-9给出一个例子。源代码如图左边所示，代码右侧的直线表示每一个相关值LR_a、LR_b和LR_c活着的区域。即使LR_a与LR_b和LR_c交迭，它也不与它们中的任何一个冲突，因为一个拷贝的源头和目标是不冲突的。因为LR_b在LR_c的定义处是活的，所以二者冲突。两个拷贝操作都是接合的候选者。 646

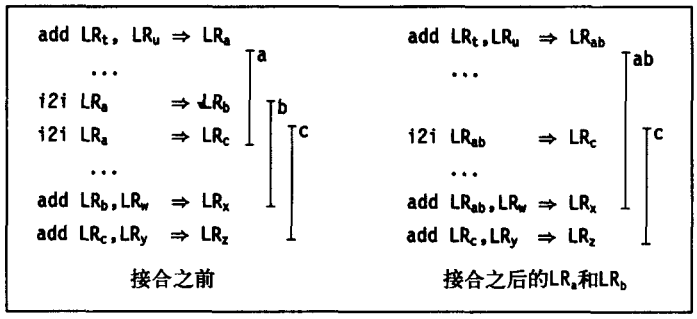


图13-9 接合存活范围

图中右侧的片段给出接合LR_a和LR_b生成LR_{ab}的结果。因为LR_c是由LR_{ab}的一个拷贝定义的，所以它们不冲突。结合LR_a和LR_b形成LR_{ab}降低了LR_c的度。一般地，接合两个存活范围不能增加它们任意邻居的度。它可以减小它们的度，或者保持它们的度不变，但是它不能增加它们的度。

为了执行接合，分配器遍历每一个块并检查这一块中的每一个拷贝操作。考虑一个拷贝i2i $LR_i \Rightarrow LR_j$。如果LR_i和LR_j不冲突（$(LR_i, LR_j) \notin E$），那么分配器接合它们，消除这一拷贝，并更新I以反映这一接合。分配器可以通过把从表示LR_j的结点出发的边移动为从表示LR_i的结点出发的边来保守地更新I：事实上，是把LR_i作为LR_{ij}。这一更新不是精确的，但是它使分配器得以继续接合。在实践中，分配器合并I允许接合的每一个存活范围，然后重写代码，再建I，并再次尝试。这一过程通常在几组接合之后停止。

本例展示在对I的保守更新中固有的不精确性。事实上，尽管LR_{ab}和LR_c之间不存在冲突，这一更新在LR_{ab}和LR_c间留下一个冲突。从转换代码重建I产生精确的冲突图，LR_{ab}和LR_c之间没有边，且允许分配器接合LR_{ab}和LR_c。

因为接合两个存活范围可能阻止其他存活范围的一系列接合，所以接合的顺序是关键的。理论上，编译器应该首先接合最频繁执行的拷贝。在实践中，分配器按发现的拷贝在块的循环嵌套的深度所决定的顺序接合这些拷贝。为了实现这一点，分配器可以按从最深层的嵌套到最浅嵌套的顺序考虑基本块。 647

13.5.7 回顾与比较

自顶向下和自底向上着色分配器有相同的基本结构，如图13-10所示。它们寻找存活范围，构建冲突图，接合存活范围，计算代码在接合版本上的溢出代价并尝试着色。构建接合的过程反复进行，直到它再也找不到机会。着色后，要出现两个状况中的一种。如果每一个存活范围都得到一种颜色，那么使用物理寄存器名字重写代码，分配结束。如果某些存活范围仍然未着色，那么插入溢出代码。

⊖ 原书中下标有误，应为k。——译者注

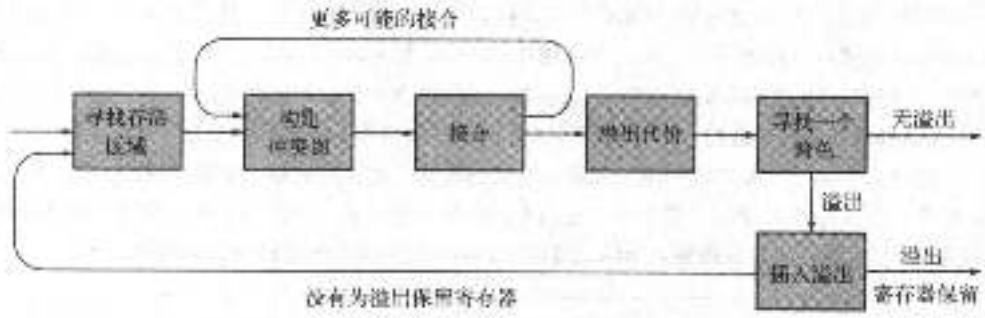


图13-10 着色分配器的结构

如果分配器已经保留了溢出寄存器，那么它在溢出代码中使用这些寄存器，使用它们的物理寄存器名字重写已着色的寄存器，这一过程终止。否则，分配器开发在溢出中使用的新的虚拟寄存器名字并为实现这一溢出插入必要的装入和存储。这稍微改变了着色问题，所以整个分配过程在转换代码上反复进行。当每一个存活范围有一种颜色时，分配器把颜色映射到真实的寄存器上，并把代码重写成最终形式。

当然，自顶向下的分配器能够采用用于自底向上分配器中的溢出迭代的基本原理。这将消除保留溢出寄存器的需要。同样地，自底向上分配器能够保留若干溢出寄存器，并消除迭代整个分配过程的需求。溢出迭代以额外的编译时间换取使用较少溢出代码的分配。保留寄存器产生可能包含更多溢出但是需要较少编译时间的分配。

648

自顶向下分配器使用它的优先度等级排序所有限定结点。它以任意顺序着色非限定结点，因为这一顺序不能改变它们得到一种颜色的事实。自底向上分配器构建一个顺序，对于这一顺序，图中大多数结点都在一个它们是非限定的图中被着色。在自底向上分配器中被分类为非限定的每一个结点都被自底向上分配器着色，因为它们在I的原来版本中是非限定的，而且在每一个从I中移出结点和边而得到的图中是非限定的。为移出结点和边而使用增量机制的自底向上分配器，把自顶向下分配器处理成限定结点的某些结点分类为非限定结点。这些结点在自顶向下分配器中仍可能被着色；在不实现它们的算法并运行它们的情况下，无法清晰地比较这两个分配器在这些结点上的性能。

真正难以着色的结点是自底向上分配器使用它的溢出度量从图中移出的那些结点。这一溢出度量只有当所有余下结点都是限定的时候才使用。这些结点形成I的一个强连通子图。在自顶向下分配器中，这些结点将被由它们的等级或优先度确定的顺序被着色。在自底向上分配器中，溢出度量使用相同的等级并做适当调整，调整的程度由该结点被选择时度被降低的结点数目决定。因此，自顶向下分配器选择溢出低优先度的限定结点，而自底向上分配器溢出在所有非限定结点已被移出后仍然是限定的那些结点。从后者的集合中，它挑选出最小化溢出度量的结点。

13.5.8 在冲突图中编码机器限制

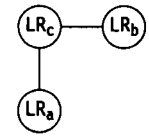
寄存器分配必须考虑目标机器和它的调用约定的特殊性质。在实践中所引发的某些限定可以在着色过程中被编码。

1. 多寄存器值

考虑如下形式的目标机器：对每一个双精度浮点值需要一个相邻寄存器的直线排列对和一个带有两个单精度存活范围LR_a和LR_b和一个双精度存活范围的LR_c的程序。

对于冲突（LR_a，LR_c）和（LR_b，LR_c），在13.5.3节所述的技术产生左边的图。三个寄存器r_0、r_1和r_2

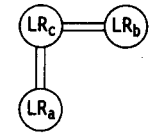
以及直线排列对（r_0，r_1）将足以构成此图。LR_a和LR_b可以共享r_2而把对（r_0，r_1）留给LR_c。遗憾的是，此图不足以表示分配上的真实限制。 649



给定$k=3$，自底向上着色分配器以任意顺序指定颜色，因为没有结点的度$\geq k$。如果这一分配器考虑LR_c，首先它会成功，因为（r_0，r_1）可以自由地保存LR_c。如果LR_a或LR_b首先被着色，那么分配器可能使用r_0或r_1，创建直线排列寄存器对LR_c不可用的状态。

为了强制一个适当的顺序，分配器可以插入两个边来表示要与一个需要两个寄存器的值的冲突。这产生左边的图。利用此图和$k=3$，自底向上分配器必须首先移出LR_a或LR_b中的一个，因为LR_c有度4。这保证两个寄存器对LR_c都可用。

双重边产生一个正确的分配，因为它们使用实际资源需求来搭配与LR_c冲突的结点的度。它不保证一个相邻对对LR_c可用。糟糕的赋值可以使LR_c没有可用对。例如，使用着色顺序LR_a、LR_c、LR_b，分配器可能把r_1赋给LR_a。编译器设计者可以通过在非限定结点间首先选择单寄存器值来使着色顺序向LR_c倾斜（图13-8中的*then*子语句）。分配器可以采用的另一种方法是：如果当它试图指定颜色时没有可用的适当序对，那么它在LR_c的邻居间执行局限性的重新着色。



2. 特定寄存器放置

寄存器分配器还必须考虑对存活范围的特定放置的需求。这些限制来自于几个方面。有些操作可能需要它们的操作数在特定的寄存器中；例如，Interl x86机器上的短无符号乘法总是把它的结果写到AX寄存器中。这一链接约定指明传送到寄存器的值的放置；这可以包含ARP，某些或所有实参以及返回值。

例如，如果存活范围LR_x在一个调用场所被作为实参传递，链接约定可能需要它在调用期间居留在寄存器r_3中。因此，我们希望分配器把LR_x赋给r_3。编译器设计者可以在冲突图中这样编码诸如此类的限制：对每一个物理寄存器向N中加入一个结点，并使用边来限制值的放置。例如，为了把LR_x强制到r_3，分配器可以加入一个从LR_x到其他每一个物理寄存器的结点的边。

当然，加入物理寄存器的边可能对冲突图施加过多的限制，迫使某些限定值溢出。如果LR_x和LR_y在不同的调用场所作为参数被传送，并且二者都必须驻留在r_3中，那么仅当LR_x和LR_y不冲突时分配器能够满足这一限制。分配器必须溢出一个或两个值来产生一个分配和赋值以把它们跨越这些值的调用放置在r_3中。 650

对于这一问题的另一个方法是，把这一限制编码到代码中。在这里，编译器在它把LR_x评估为一个参数的地点插入一个从LR_x到r_3的拷贝。当LR_y作为一个参数出现时，编译器采用相同的作法。这在调用场所创建一个小的存活范围，这些小存活范围可以被预先着色成r_3。（在指定任意颜色之前，分配器把这些限定的存活范围赋给它们要求的寄存器。）然后，分配器依赖于保守的连接或有倾向的着色来在可能的情况下消除这些拷贝。如果拷贝不能被消除，那么它可能允许分配器避免溢出一个完整的存活范围。

13.6 高级话题

因为在寄存器分配期间错误的代价会很高，所以寄存器分配算法已得到大量的关注。已出版了很多关于基本图着色分配技术。13.6.1节描述其中的几个方法。如本章所描述的那样，全局分配器也同样适用于很多程序和很多体系结构。13.6.2节给出寄存器分配中存在较困难问题的三个领域。

13.6.1 图着色分配的若干变形

文献中已出现很多关于这两个基本图着色寄存器分配的变形。本节描述其中若干个改进版本。其中

一些关注分配的代价，而另外一些关注分配的质量。

1. 非精确冲突图

Chow的自顶向下、基于优先度的分配器使用了冲突的一种不精确概念：存活范围LR_i和LR_j冲突，如果二者在同一基本块内是活的。这能够加快冲突图的构建。然而，此图的不精确特性过高评估某些结点的度，并阻止分配器使用冲突图作为接合的基础。（在一个不精确的图中，由一个有用拷贝连接的两个
651 存活范围是冲突的，因为它们在同一基本块内是活的。）分配器还包含一个预遍来在存活范围局限于在一个块的值上执行局部分配。

2. 把冲突图分割成为小片

如果冲突图能够被分成不连结的成份，那么那些不相交的成份可以独立着色。因为位矩阵的大小是$O(N^2)$，所以把这一位矩阵分割成独立的成份即可以节省空间也可以节省时间。分割冲突图的一个方法是分别考虑不重叠的寄存器类，如浮点寄存器和整数寄存器。对于较大过程的一个更复杂的选择是发现集团分离子，集团分离子是这样的连通子图，它们的消除把冲突图分成互不相交的若干片。对于足够大的图，使用散列表来取代位矩阵可以提高时空效率。

3. 保守接合

当分配器接合两个存活范围LR_i和LR_j时，新的存活范围LR_{ij}可能比LR_i或LR_j受到更多的限制。如果LR_i和LR_j有不同的邻居，那么$LR^0_{ij} > \max(LR^0_i, LR^0_j)$。如果$LR^0_{ij} < k$，那么创建$LR_{ij}$是绝对有益的。然而，如果$LR^0_i < k$且$LR^0_j < k$但$LR^0_{ij} \geq k$，那么，接合$LR_i$和$LR_j$使得$I$在不溢出的情况下更难着色。为了避免这一问题，一些编译器设计者使用被称为保守接合（conservative coalescing）的一种接合的限定形式。在这一方案中，只有当$LR^0_{ij} < k$时分配器才接合LR_i和LR_j。这保证LR_i和LR_j的接合不会使冲突图更难着色。

如果分配器使用保守接合，另一种改进是可能的。当分配器达到这样一个点，在此处每一个留下的存活范围是限定的，此时基本算法选择一个溢出候选。另外一个方法是在这一点处再次运用接合。由于结果存活范围的度而没有被接合的存活范围，可能在简化图中得到接合。接合可以降低与拷贝的源头和目标都冲突的结点的度。因此，这一*迭代接合*（iterated coalescing）可以消除额外的拷贝并降低结点的度。它可以创建一个或多个非限制结点并允许着色进行。如果它不创建任何非限制结点，那么溢出如前一样进行。

偏着色（biased coloring）是接合使图更难着色的拷贝的另外一种方法。在这一方法中，分配器设法给由一个拷贝连接的存活范围指定相同的颜色。在挑选LR_i的颜色时，分配器首先尝试这样的颜色，
652 这些颜色已赋给通过一个拷贝操作连结到LR_j的那些存活范围。如果分配器能够给二者赋相同的颜色，那么它消除这一拷贝。通过细心的实现，这对着色过程增加很少代价或不增加代价。

4. 溢出部分存活范围

如前所述，全局分配的两个方法都溢出整个存活范围。如果对寄存器的需求在整个存活范围的大部分区域较低而在一个较小的区域较高时，上述方法将导致过度溢出。更复杂的溢出技术寻找一个区域，在这个区域中，溢出存活范围是有价值的，也就是说，这一溢出在真正需要寄存器的区域释放寄存器。自顶下向分配器所描述的分割方案通过分别考虑已溢出存活范围内的每一个块而实现这一点。在自底向上分配器中，通过只在冲突出现的区域溢出可以得到类似的结果。一个被称为*冲突区域溢出*（ingterference-region spillin）的技术识别在高需求区域内冲突的一组存活范围，并把溢出限制在这一区域内。分配器可以评估冲突区域溢出的若干策略的代价，并对照标准的随处溢出方法比较这些代价。通过让这些选择在一个已评估代价的基础上进行竞争，分配器可以改进整个分配。

5. 存活范围的分割

把一个存活范围分割成小块可以改进基于着色的寄存器分配的结果。理论上，分割利用两个不同的效应。如果已分割存活范围的度比原来的存活范围的度低，那么它们可能更容易着色，甚至可能是非限制的。如果已分割存活范围中的一个有较高的度，因此溢出，那么分割可能阻止原来同一存活范围中有较低度的部分的溢出。作为最终的实际效应，分割在存活范围被分割的地点引入溢出。精心选择分割点可以控制某个溢出代码的放置，例如放在循环的外部而不是循环的内部。

人们已经尝试了很多分割方法。13.5.4节描述了一种方法，这一方法把一个存活范围分割成块，并在不改变分配器指定颜色的能力的前提下，把它们再次接合到一起。人们已经尝试了若干使用控制流图的性质来选择分割点的方法。Briggs指出很多方法是不相容的[45]；然而，两个特殊的技术很有前景。一个方法称为*零代价分割*（zero-cost splitting），这一方法利用指令调度中的nop来分割存活范围并改进分配和调度。另一个技术称为*被动分割*（passive splitting），这一技术使用有向冲突图来决定分割应该发 653
生的地点，并基于对分割和溢出的代价的评估在它们之间做出选择。

6. 重新实现

对于某些值，重新计算比溢出成本低。例如，可以使用一个装入立即重新创建一个小整数常量，而不是使用一个装入从内存中重新得到它。分配器能够识别出这样的值并重新实现它们，而不是溢出它们。

修改自底向上图着色分配器来执行重新实现需要几个小变动。分配器必须识别和标记可以被重新实现的SSA名字。例如，参数总是可用的任意操作是一个候选。分配器可以使用第10章给出的常量传播算法中的一个，在代码上传播这些重新实现标记。在形成存活范围的过程中，分配器应该只接合有相同重新实现标记的SSA名字。

编译器设计者必须使溢出代价评估正确处理重新实现标记，使得这些值有精确的溢出代价评估。溢出代价插入过程还必须检查这些标记并为可重新实现值生成适当的微小溢出。最后，分配器应该使用保守接合来永久地避免把存活范围与不同的重新实现标记组合起来。

13.6.2 寄存器分配中较困难的问题

本章已给出一些处理寄存器分配中的问题的算法。还有很多更加困难的问题。在若干前沿领域仍存在着改进的空间。

1. 整个程序的分配

本章所给出的算法都是在单一过程内考虑寄存器分配。当然，整个程序是由多个过程构建起来的。如果，通过考虑更大的作用域，全局分配产生的结果比局部分配的结果好，那么编译器设计者是否应该在整个程序上考虑执行分配呢？整个程序优化有把溢出跨越过程边界移动而得到定制过程调用约定和消除保存和恢复的效应。 654

然而，当把分配问题从单一过程扩展到整个程序时，它发生重大的改变。所有整个程序分配方案都必须处理下面的每一个问题。

为了执行整个程序分配，分配器必须存取整个程序的代码。事实上，这意味着在链接时执行整个程序分配。（当整个程序分析和转换在链接时之前可以运用时，通过在链接时执行这些技术可以有效地回避复杂化这样技术的实现中出现的很多问题。）

编译器需要对整个程序的所有各个部分的相对执行频率进行精确的评估。在一个过程内，静态评估（例如“一个循环执行10次”）已证明是真实行为的一个合理的近似。跨越整个程序，这样简单的评估器

可能无法提供真实行为的好的近似。

整个程序分配器还必须处理参数绑定机制。而引用调用参数可以把不同过程中独立的存活范围连结成一个单一过程间存活范围。由于必须对这一过程间存活范围设置太多的限制，这一效应将会复杂化分配。

2. 分割寄存器集合

新的硬件特性可能引发新的复杂性。例如，考虑在已划分寄存器集合的机器上所引发的不一致代价问题。当功能单元的数量增加时，保存操作数和结果所需的寄存器的数量也增加。在寄存器和功能单元之间移动值所需要的硬件逻辑中的限制也会增加。为了保持硬件代价可管理，一些设计师把寄存器集合划分成了更小的寄存器卷，并根据这些寄存器集合对功能单元分组群。为了保持一般性，处理器通常提供在这些组群间移动值的某些有限机制。图13-11给出这样的处理器的一个抽象形式。不失一般性，假设每一组群有一组除名字以外有相同的功能单元和寄存器集合。

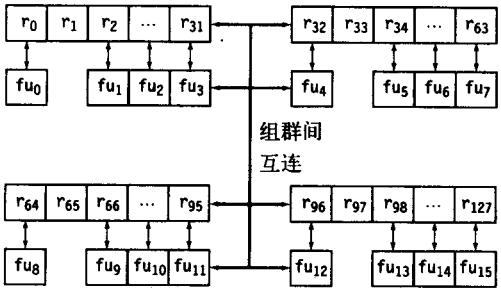


图13-11 组群寄存器集合机器

带有组群寄存器集合的机器把一组新的复杂性划分成不同层次的寄存器赋值问题。在决定LR_i在寄存器集合的位置的过程中，分配器必须处理每一个组群中的寄存器的可用性、组群间转换机制的代价和局部可用性，以及执行引用LR_i的操作的特定功能单元等问题。例如，每一个周期每一个组群只能有一个组群间转换，处理器可能允许每一个组群在每一个周期生成一个组群外寄存器引用。对于这样的限制，
655 分配器必须对操作的放置所在的组群和时间给予关注。（显然，这一问题需要受到指令调度器和寄存器分配器的关注。）有些体系结构允许在组群间移动值的寄存器到寄存器拷贝操作，并附带在单一周期移出移入每一个组群的值的数量的限制。在这样的体系结构中，一个值的跨组群使用需要额外的指令；如果在一个关键路径进行跨组群使用，这将延长整个执行时间。

3. 歧义值

在频繁使用歧义值的代码中，分配器是否有把这样的值保存在寄存器中的能力是一个非常严肃的性能问题。为了改进歧义值的分配，若干系统包含了重写代码来将非歧义值保存在标量局部变量中的转换，即使它们的“自然”的家是在一个数组元素内或是在基于指针的结构内。*标量替换*（scalar replacement）使用数组下标分析来识别数组元素值的复用，并引入保存复用值的标量临时变量。*寄存器提升*（register-promotion）使用指针值的数据流分析来决定何时一个基于指针的值在一个循环嵌套中可以安全地保存在寄存器中，并重写代码使这个值被保存在一个新的临时变量中。这两个转换把分析结果编码到代码形态中，使对寄存器分配器来说把这些值保存在寄存器中是显然的。事实上，提升太多的值可能产
656 生溢出代码，其代价超过转换试图回避的内存操作的代价：理想上，这些技术都应该集成到分配器中，以便其中对寄存器需求的真实评估可以用于决定提升多少个值。

13.7 概括和展望

因为寄存器分配器是现代编译器的重要组成部分，所以人们在文献中对它已经投入了很多的关注。对于局部分配、全局分配和区域分配都存在很强的技术。因为其基本问题几乎都是NP困难的，所以解决方案倾向对细小决策敏感，诸如如何打破相同等级的选择等。

寄存器分配中的进展来自于对于这些问题给予我们深远影响的范例的使用。因此，图着色分配器一直很受欢迎，这不是因为寄存器分配相当于图着色，而是因为着色捕获了全局分配问题的某些关键方面。事实上，大多数着色分配器的改进都来自于对着色范例没能精确反映这一问题的地方所做的进攻，诸如存活范围分割的更好代价模型和改进方法就是如此。

本章注释

作为一个问题，寄存器分配源自最早的编译器。Backus指出Best于20世纪50年代中期在开发原始FORTRAN编译器时发明了我们称之为“自底向上局部”的算法[26，25]。多年来，Best的算法在很多上下文中重新被发现并复用[33，173，109，237]。这一算法的最广为人知的具体形式可能是Belady的脱机页替换算法[33]。Horwitz[186]和Kennedy[202]描述了组合净值和不净值引发的复杂性问题。Liberatore等提出在溢出不净值之前溢出净值是一个可行的折中方法[237]。（13.3.2节中的）展示净值和不净值引发的问题的例子是由Ken Kennedy提出的。

文献中所描述的第一个图着色全局分配器是由Chaitin和他的同事在IBM的PL.8编译器上构建的自底向上分配器[70，68，69]。反过来，它构建于最初由Lavrov[232]提出的图着色和存储分配问题之间的关系上。Alpha编译器项目把这些想法用于把数据封装到内存中[132，133]。Schwartz描述了由Ershov和 657
Cocke分别提出的早期算法[300]；这些算法致力于减小所使用的颜色的数量并忽视溢出。

自顶向下图着色源于Chow[76,77,78]。他的实现是从内存到内存模型开始的，使用了一个不精确冲突图，并执行13.5.4节所述的存活范围分割。不精确冲突图不支持接合，所以编译器使用独立的优化遍来接合拷贝[76]。这一风格的分配已被用于MIPS、Silicon Graphics中的若干编译器中，并被用于Gnu编译器gcc中。Larus和Hilfinger为SPUR LISP构建了一个使用精确冲突图并操作在寄存器到寄存器模型上[231]的自顶向下、基于优先度的分配器。

13.5.5节中的自底向上分配器遵循由Briggs修改的Chaitin的设计 [48,49,53]。Chaitin的贡献包括冲突的基本定义和构建冲突图、接合及处理溢出的算法。Briggs通过把限定存活范围压入栈上，而不是直接溢出它们，修改了Chaitin的设计；这使Briggs分配器可以对带有很多使用很少颜色的邻居的结点进行着色。自底向上着色中其他有意义的改进包括更好的溢出度量、干净溢出、三中取最佳溢出[36]、冲突区域溢出[35]，重新实现简单值的方法[52]、迭代接合[153]、优化接合[270]、溢出部分存活范围的方法[35]以及诸如零代价分割[225]和被动分割[100]等存活范围分割的方法。精确冲突图的较大尺寸导致Gupta、Soffa和Steele对使用组群分离子分割图进行了研究[168]。Harvey提出了通过寄存器类分割图的方法[94]。Chaitin、Nickerson和Briggs都讨论了把边加入冲突图以模型化赋值上的特定限制[70，266，51]。

重写代码以改进寄存器分配的优化在减小内存冲突中显出了它的有效性。Carr的标量替换转换[61，65]使用数据相关性分析来寻找可以被重写成标量的基于数组引用。寄存器提升[244，296，241]以类似的方法使用指针数据流分析重写基于指针的引用。

本章集中讨论了局部和全局分配。一些作者审视了位于单一块与整个过程之间的区域上的分配。Koblenz和Callahan描述了Tera计算机的编译器中的层次着色分配器[63]。Knobe和Meltzer使用Compass编译器的类似性质构建了一个分配器；这一分配器构建一棵控制树，并在这棵树上执行两遍分配[214]。Proebsting和Fischer开发了概率方法[279]。基于接合的技术归功于Lueh、Adl-Tabatabai和Gross[247]。 658



附录A
ILOC

A.1 概述

ILOC是简单抽象RISC机器的线性汇编代码。本书所使用的ILOC是Rice大学用于巨型标量编译器项目（MSCP）的中间表示的一个简化版本。例如，本书中的ILOC不区分不同类型的数，为简单起见，它们假设所有的数都是整数。（有一些例子假设数据项是64位的，其他的例子假设数据项是32位的。从例子中可以明显看出这一差异。）

ILOC抽象机器的寄存器数量不限。它有三地址、寄存器到寄存器操作、装入和存储操作，比较操作和分支。它只支持几个简单的寻址模式：直接、地址+偏移、地址+立即以及立即。源操作数在操作的发行周期的开始读取。结果操作数在操作完成的周期的末端被定义。

除了它的指令集合外，机器的细节没有明确描述。大多数例子都假设一台简单机器，带有一个功能
659 单元，按ILOC操作出现的顺序执行这些操作。当使用其他模型时，我们再明确讨论它们。

一个ILOC程序是由指令的一个线性列表组成的。每一个指令之前可以附有一个标签。标签只是文本串；它与指令之间由冒号隔开。根据习惯，我们把标签的形式限定为[a-z]([a-z]|[0-9]|-)*。如果某个指令需要多个标签，那么我们在这一指令前面插入一个只包含一个nop的指令，并在这个nop上设置额外的标签。更形式地定义ILOC程序如下所示。

```
Iloc程序      → 指令列表
指令列表      → 指令
              | label : 指令
              | 指令   指令列表
```

每一个指令包含一个或多个操作。单一操作指令自身占一行，而多操作指令可以跨行书写。为了把多个操作组成单一指令，我们用方括号把这些操作封闭起来，并用分号把它们隔开。更形式地：

```
指令          → 操作
              | [操作列表]
操作列表      → 操作
              | 操作; 操作列表
```

一个ILOC操作对应于在单一周期被发行到一个功能单元的机器级指令。它有一个操作码，一个由逗号隔开的源操作数序列和一个由逗号隔开的目标操作数序列。源头与目标由符号⇒隔开。这一符号读作“进入”。

```
操作          → 一般操作
              | 控制流操作
一般操作      → 操作码 操作数列表 ⇒ 操作数列表
操作数列表    → 操作数
              | 操作数, 操作数列表
```

```
操作数      → register
            | num
            | label
```

660

非终结符*Opcode*（操作码）可以是除cbr、jump或jumpI之外的任意ILOC操作。遗憾的是，如同真实的汇编语言一样，操作码与它的操作数的形式之间的关系不是很系统。描述每一个操作码的操作数形式的最简单方法是用表格形式。在本附录后面出现的表格给出了本书所使用的每一个ILOC操作码的操作数的数量和它们的类型。

有三种类型的操作数：register（寄存器）、num（数）和label（标签）。每一个操作数的类型是由操作码和这一操作数在操作中的位置所决定的。在本书的例子中，我们使用数字（r_{10}）形式和符号（r_i）形式的寄存器名字。数字是简单的整数，如有必要的话可以是带符号整数。我们总是在标签的开始放上一个l来使其类型一目了然。这是一个约定而不是一个规则。ILOC模拟器和工具应该把上述的任意形式的串当作一个可能的标签处理。

大多数操作有一个目标操作数；某些store操作有多个目标操作数，分支也一样。例如，storeAI有一个源操作数和两个目标操作数。而源头必须是一个寄存器，且目标必须是一个寄存器和一个立即常量。因此，ILOC操作

```
storeAI r_i ⇒ r_j, 4
```

通过把4加到r_j的内容中计算一个地址，并把在r_i中发现的值存储到由这个地址所定义的内存位置上。换句话说，

$$\text{MEMORY}(r_j + 4) \leftarrow \text{CONTENTS}(r_i)$$

控制流操作有稍微不同的语法。因为它们不定义它们的目标，我们使用单箭头→而不是⇒书写这些控制流操作。

```
控制流操作  → cbr    register  → label , label
            | jumpI             → label
            | jump              → register
```

第一个操作cbr实现一个条件分支。其他两个操作是无条件分支，称为跳转。

661

A.2 命名约定

本书的例子中的ILOC代码使用一组简单的命名约定。

1）变量的内存偏移由在变量名字前加前缀符号@来表示。

2）用户可以假定寄存器供给是不受限制的。这些寄存器都是用简单的整数形式，如在r_{1776}中，或用符号形式，如在r_i中，来命名的。

3）寄存器r_0被保留为指向当前活动记录的指针。我们通常用r_{arp}取代r_0，作为一个提示。因此，下面的操作把变量x的内容装入r_1中。

```
loadAI r_0, @x ⇒ r_1
```

r_0的使用揭示x存储在当前活动记录中这一事实。当然，这与loadAI r_{arp}, @x⇒r_1相同。

ILOC的注释从串//开始直到行尾。我们假设这些都被扫描器除去；因此，它们可以出现指令的任意地方，而且不用语法来描述。

A.3 各种操作

本书的例子使用ILOC操作的一个限定的集合。在本附录末尾的表格给出本书中所用ILOC操作的全集，这里不包括第7章中用于讨论某些分支结构的分支语法的例子。

A.3.1 算术

662 为了支持算术，ILOC提供一组基本的三地址、寄存器到寄存器操作。

操作码	源操作数	目标操作数	含 义
add	r_1, r_2	r_3	$r_1 + r_2 \Rightarrow r_3$
sub	r_1, r_2	r_3	$r_1 - r_2 \Rightarrow r_3$
mult	r_1, r_2	r_3	$r_1 \times r_2 \Rightarrow r_3$
div	r_1, r_2	r_3	$r_1 \div r_2 \Rightarrow r_3$

这些操作都假设源操作数在寄存器中。它们把结果写回一个寄存器中。任意寄存器可以充当源操作数和目标操作数。

指定一个立即操作数常常很有用。因此，每一个算术操作都有一个立即形式的支持，对于非交换的算术操作则有两个立即形式的支持。

操作码	源操作数	目标操作数	含 义
addI	r_1, c_1	r_2	$r_1 + c_1 \Rightarrow r_2$
subI	r_1, c_1	r_2	$r_1 - c_1 \Rightarrow r_2$
rsubI	r_1, c_1	r_2	$c_1 - r_1 \Rightarrow r_2$
multI	r_1, c_1	r_2	$r_1 \times c_1 \Rightarrow r_2$
divI	r_1, c_1	r_2	$r_1 \div c_1 \Rightarrow r_2$
rdivI	r_1, c_1	r_2	$c_1 \div r_1 \Rightarrow r_2$

这些形式对表示特定优化的结果、更简明地写出例子，以及记录减少对寄存器的需求的简明方法都很有用。

注意，在使用ILOC的实际编译器中，我们将需要引入多个数据类型。这将带来类型化的操作码或多态操作码。我们的优先选择是类型化操作码系列：整数加法、浮点加法等等。ILOC起源的MSCP编译器对于整数、单精度浮点数、双精度浮点数、复数以及指针数据有不同的算术操作，但没有字符数据操作。

A.3.2 移位

663 ILOC支持一组算术移位操作，向左移位和向右移位，寄存器形式和立即形式。

操作码	源操作数	目标操作数	含 义
lshift	r_1, r_2	r_3	$r_1 \ll r_2 \Rightarrow r_3$
lshiftI	r_1, c_2	r_3	$r_1 \ll c_2 \Rightarrow r_3$
rshift	r_1, r_2	r_3	$r_1 \gg r_2 \Rightarrow r_3$
rshiftI	r_1, c_2	r_3	$r_1 \gg c_2 \Rightarrow r_3$

A.3.3 内存操作

为了在内存和寄存器之间移动值，ILOC支持一全套装入和存储操作。load和cload操作把数据项从内存移到寄存器。

操作码	源操作数	目标操作数	含 义
load	r_1	r_2	MEMORY(r_1) ⇒ r_2
loadAI	r_1, c_1	r_2	MEMORY(r_1 + c_1) ⇒ r_2
loadAO	r_1, r_2	r_3	MEMORY(r_1 + r_2) ⇒ r_3
cload	r_1	r_2	字符load
cloadAI	r_1, r_2	r_3	字符loadAI
cloadAO	r_1, r_2	r_3	字符loadAO

上表中的这些操作在它们所支持的寻址模式上不同。load和cload形式假设整个地址是在单一寄存器操作数中。loadAI和cloadAI形式把一个立即值加到寄存器的内容中。我们称这些为*地址立即*（address-immediate）操作。loadAO和cloadAO形式在执行load之前把两个寄存器内容加起来计算有效地址。我们称这些操作为*地址偏移*（address-offset）操作。

作为load的最后一种形式，ILOC支持简单的装入立即操作。它从指令流中取一个整数并把它放入一个寄存器中。

操作码	源操作数	目标操作数	含 义
loadI	c_1	r_2	c_1 ⇒ r_2

664

完整的类ILOC IR对于它所支持的每一种不同类型的值都应该有一个装入立即。

存储操作与装入操作相匹配。ILOC支持简单寄存器形式、地址立即形式和地址偏移形式的数字存储和字符存储。

操作码	源操作数	目标操作数	含 义
store	r_1	r_2	r_1 ⇒ MEMORY(r_2)
storeAI	r_1	r_2, c_1	r_1 ⇒ MEMORY(r_2 + c_1)
storeAO	r_1	r_2, r_3	r_1 ⇒ MEMORY(r_2 + r_3)
cstore	r_1	r_2	字符store
cstoreAI	r_1	r_2, r_3	字符storeAI
cstoreAO	r_1	r_2, r_3	字符storeAO

上表中没有存储立即操作。

A.3.4 寄存器到寄存器的拷贝操作

为了在寄存器之间移动值而不经过内存，ILOC包含一组寄存器到寄存器的拷贝操作。

操作码	源操作数	目标操作数	含 义
i2i	r_1	r_2	r_1 ⇒ r_2, 数值
c2c	r_1	r_2	r_1 ⇒ r_2, 字符
c2i	r_1	r_2	将字符转换为整数
i2c	r_1	r_2	将整数转换为字符

前两个操作i2i和c2c把一个值从一个寄存器拷贝到另一个寄存器，且没有转换。前者适用于整数值，而后者适用于字符。后两个操作执行字符和整数之间的转换，前者把字符转换成它在ASCII字符集合的
665 序位，后者使用相应的ASSCII字符取代一个整数。

A.4 例子

让我们利用如图A-1所示的第1章的例子使上述讨论更加具体。注意每一行右端的注释。在我们基于ILOC的体系中，编译器前端自动生成注释来使ILOC代码很适合我们阅读。因为本书中的例子的目的都主要是为了阅读，所以我们注释ILOC。

```
1.  loadAI  r_arp, @w ⇒ r_w     // w在自r_arp偏移为0处
2.  loadAI  2         ⇒ r_2     // 将常量2置入r_2
3.  loadAI  r_arp, @x ⇒ r_x     // x在偏移为8处
4.  loadAI  r_arp, @y ⇒ r_y     // y在偏移为12处
5.  loadAI  r_arp, @z ⇒ r_z     // z在偏移为16处
6.  mult    r_w, r_2  ⇒ r_w     // r_w ← w × 2
7.  mult    r_w, r_x  ⇒ r_w     // r_w ← (w × 2) × x
8.  mult    r_w, r_y  ⇒ r_w     // r_w ← (w × 2 × x) × y
9.  mult    r_w, r_z  ⇒ r_w     // r_w ← (w × 2 × x × y) × z
10. storeAI r_w       ⇒ r_arp, @w // 将r_w写回'w'
```

图A-1 介绍性例子

然而，记住编译器并不读取这一注释，因此，在如下操作中注释帮助我们，但是不帮助编译器确定标签跳转的目标是什么。

```
jump → r_32 // 返回至循环顶部
```

这一例子假设w、x、y和z都被存储在局部活动记录中距ARP一个固定的偏移处。第一个指令是loadAI操作，即装入地址立即（load address-immediate）操作。根据操作码表，我们可以看到这一指令把r_{arp}的内容与立即常量@w结合起来，并在那个地址的内存中找回值；这个值是w。它把这个找回的值放入r_w中。下一个指令是loadI操作，即装入立即（load immediate）操作。它把值2移到r_2中。（从效果上
666 看，它从这个指令从流中把一个常量读到一个寄存器中。）指令3到指令5把值x装入r_x中，把y装入r_y中并把z装入r_z中。

第6个指令把r_w和r_2的内容相乘，并将结果存储回r_w中。指令7把这个量与r_x相乘。指令8把这个量与r_y相乘，指令9把这个量与r_z相乘。从指令6到指令9的每一个指令都把这个值累加到r_w中。

最后的指令把r_w的值保存到内存中。这一指令使用storeAI，或存储地址立即（store address-immediate）操作把r_w的内容写入距r_{arp}偏移为@w的内存位置。正如第1章所指出的那样，这一序列评估表达式w←w × 2 × x × y × z。

A.5 控制流操作

一般地，ILOC的比较操作符取两个值并返回一个布尔值。如果它的操作数之间持有特殊的关系，
667 那么比较把目标寄存器设置为值true；否则目标寄存器得到值false。

操作码	源操作数	目标操作数	含义	
cmp_LT	r_1, r_2	r_3	true ⇒ r_3	if $r_1 < r_2$
			false ⇒ r_3	否则
cmp_LE	r_1, r_2	r_3	true ⇒ r_3	if $r_1 \leq r_2$
			false ⇒ r_3	否则
cmp_EQ	r_1, r_2	r_3	true ⇒ r_3	if $r_1 = r_2$
			false ⇒ r_3	否则
cmp_GE	r_1, r_2	r_3	true ⇒ r_3	if $r_1 \geq r_2$
			false ⇒ r_3	否则
cmp_GT	r_1, r_2	r_3	true ⇒ r_3	if $r_1 > r_2$
			false ⇒ r_3	否则
cmp_NE	r_1, r_2	r_3	true ⇒ r_3	if $r_1 \neq r_2$
			false ⇒ r_3	否则

可以通过使用条件分支操作把比较的结果用于改变控制流。

操作码	源操作数	目标操作数	含义
cbr	r_1	l_1, l_2	l_1 → PC if r_1 = true
			l_2 → PC if r_1 = false

条件分支操作取一个布尔值作为它的参数，并控制转换到两个目标标签中的一个。如果这一布尔值为true，那么选择第一个标签；如果这一布尔值为false，选择第二个标签。因为两个分支目标不是由指令定义的，所以我们可以稍微改变一下语法。我们用单箭头→写出这一分支，而不是使用箭头⇒。

在条件分支使用两个标签有两个优势。第一，代码在某种程度上更精确。在某种状态下，只带有一个标签的条件分支可能需要一个后继跳转。两个标签分支把这一相同的组合编码到单一操作中。第二，代码更容易处理。单一标签分支依赖于代码布局。它隐含地把包含这一分支的块与落下路径上的块连结起来。使用单一标签分支，编译器必须保留这些关系。

两标签条件分支使得这一隐式关联变成显式的，并消除任意可能的位置相关性。这就使编译器可以自由地以它所希望的最小化执行时间的顺序布局分支和块。最后，它简化控制流图的结构。

A.5.1 其他比较和分支语法

在第7章，我们讨论了当比较返回一个被写入到条件代码寄存器中的值时所发生的事情。这样的条件代码只能被更复杂的条件分支指令来解释。为了讨论这一机制，我们使用另外一组比较和条件分支操作。

668

操作码	源操作数	目标操作数	含义	
comp	r_1, r_2	cc_1	置 cc_1	
cbr_LT	cc_1	l_1, l_2	l_1 → PC	if cc_1 = LT
			l_2 → PC	否则
cbr_LE	cc_1	l_1, l_2	l_1 → PC	if cc_1 = LE
			l_2 → PC	否则
cbr_EQ	cc_1	l_1, l_2	l_1 → PC	if cc_1 = EQ
			l_2 → PC	否则
cbr_GE	cc_1	l_1, l_2	l_1 → PC	if cc_1 = GE

（续）

操作码	源操作数	目标操作数	含 义	
			$l_2 \rightarrow$ PC	否则
cbr_GT	cc_1	l_1, l_2	$l_1 \rightarrow$ PC	if cc_1 = GT
			$l_2 \rightarrow$ PC	否则
cbr_NE	cc_1	l_1, l_2	$l_1 \rightarrow$ PC	if cc_1 = NE
			$l_2 \rightarrow$ PC	否则

这里，比较操作符comp取两个值，并适当地设置条件代码。我们总是指定comp的目标为写成cc_i的条件代码寄存器。相应的条件分支有6个变形，其中每一个都对应一个比较结果。

A.5.2 跳转

ILOC包含两种跳转操作形式。几乎所有例子中使用的形式都是把控制转换成文字标签的立即跳转。第二种形式是跳转到寄存器操作，它取单一寄存器操作数。它把寄存器的内容解释为一个运行时地址，并把控制转换到那个地址上。

操作码	源操作数	目标操作数	含 义
jumpI	无	l_1	$l_1 \rightarrow$ PC
jump	无	r_1	$r_1 \rightarrow$ PC

669 跳转到寄存器形式是一个歧义控制流转换。一旦它被生成，编译器也许不能够推断出这一跳转的目标标签的正确集合。因为这一原因，编译器应该尽量避免使用到寄存器的跳转。

有时，避免使用到寄存器的跳转需要非常复杂的处理，以至于到寄存器的跳转变得更具吸引力，尽管它存在问题。例如，FORTRAN包含一个跳转到标签变量的结构；使用立即分支实现这一结构将需要类似于选择语句的逻辑：一系列立即分支以及把标签变量的运行时值与可能标签集匹配的代码。在这样的环境下，编译器有可能会使用到寄存器的跳转。

为了减小来自跳转到寄存器操作的信息损失，ILOC包含一个让编译器记录跳转到寄存器的可能标签集合的伪操作。tbl操作有两个参数，一个寄存器和一个立即标签。

操作码	源操作数	目标操作数	含 义
tbl	r_1, l_2	—	r_1 might hold l_2

tbl操作只能出现在jump操作之后。编译器把一个或多个tbl的集合解释成对寄存器所有可能标签的命名。因此，下面的序列表明跳转的目标是L01、L03、L05或L08中的一个。

```
jump          →r1
tbl r1, L01
tbl r1, L03
tbl r1, L05
tbl r1, L08
```

A.6 SSA形式的表示

当编译器根据程序的IR版本构造它的SSA形式时，它需要一个表示ϕ函数的方法。在ILOC中，书写

ϕ函数的自然的方法是把它作为一个ILOC操作。因此，我们有时候将把ϕ函数$r_m \leftarrow \phi(r_i, r_j, r_k)$写成phi r_i, r_j, $r_k \Rightarrow r_m$。因为SSA形式的性质，phi操作可能取任意多个源操作数。它总是定义单一目标。 670

ILOC操作码概括

操作码	源操作数	目标操作数	含 义
nop	无	无	使用其作为占位符
add	r_1, r_2	r_3	$r_1 + r_2 \Rightarrow r_3$
addI	r_1, c_1	r_2	$r_1 + c_1 \Rightarrow r_2$
sub	r_1, r_2	r_3	$r_1 - r_2 \Rightarrow r_3$
subI	r_1, c_1	r_2	$r_1 - c_1 \Rightarrow r_2$
mult	r_1, r_2	r_3	$r_1 \times r_2 \Rightarrow r_3$
multI	r_1, c_1	r_2	$r_1 \times c_1 \Rightarrow r_2$
div	r_1, r_2	r_3	$r_1 \div r_2 \Rightarrow r_3$
divI	r_1, c_1	r_2	$r_1 \div c_1 \Rightarrow r_2$
lshift	r_1, r_2	r_3	$r_1 \ll r_2 \Rightarrow r_3$
lshiftI	r_1, c_2	r_3	$r_1 \ll c_2 \Rightarrow r_3$
rshift	r_1, r_2	r_3	$r_1 \gg r_2 \Rightarrow r_3$
rshiftI	r_1, c_2	r_3	$r_1 \gg c_2 \Rightarrow r_3$
and	r_1, r_2	r_3	$r_1 \wedge r_2 \Rightarrow r_3$
andI	r_1, c_2	r_3	$r_1 \wedge c_2 \Rightarrow r_3$
or	r_1, r_2	r_3	$r_1 \vee r_2 \Rightarrow r_3$
orI	r_1, c_2	r_3	$r_1 \vee c_2 \Rightarrow r_3$
xor	r_1, r_2	r_3	r_1 *xor* $r_2 \Rightarrow r_3$
xorI	r_1, c_2	r_3	r_1 *xor* $c_2 \Rightarrow r_3$
loadI	c_1	r_2	$c_1 \Rightarrow r_2$
load	r_1	r_2	MEMORY $(r_1) \Rightarrow r_2$
loadAI	r_1, c_1	r_2	MEMORY $(r_1 + c_1) \Rightarrow r_2$
loadAO	r_1, r_2	r_3	MEMORY $(r_1 + r_2) \Rightarrow r_3$
cload	r_1	r_2	字符load
cloadAI	r_1, r_2	r_3	字符loadAI
cloadAO	r_1, r_2	r_3	字符loadAO
store	r_1	r_2	$r_1 \Rightarrow$ MEMORY (r_2)
storeAI	r_1	r_2, c_1	$r_1 \Rightarrow$ MEMORY $(r_2 + c_1)$
storeAO	r_1	r_2, r_3	$r_1 \Rightarrow$ MEMORY $(r_2 + r_3)$
cstore	r_1	r_2	字符store
cstoreAI	r_1	r_2, r_3	字符storeAI
cstoreAO	r_1	r_2, r_3	字符storeAO
i2i	r_1	r_2	$r_1 \Rightarrow r_2$
c2c	r_1	r_2	$r_1 \Rightarrow r_2$
c2i	r_1	r_2	将字符转换为整数
i2c	r_1	r_2	将整数转换为字符

671

ILOC控制流操作

操作码	源操作数	目标操作数	含 义
cbr	r_1	l_1, l_2	$r_1 =$ true $\Rightarrow l_1 \rightarrow$ PC $r_1 =$ false $\Rightarrow l_2 \rightarrow$ PC
jumpI	无	l_1	$l_1 \rightarrow$ PC
jump	无	r_1	$r_1 \rightarrow$ PC
cmp_LT	r_1, r_2	r_3	$r_1 < r_2 \Rightarrow$ true $\rightarrow r_3$ (否则，false $\rightarrow r_3$)

（续）

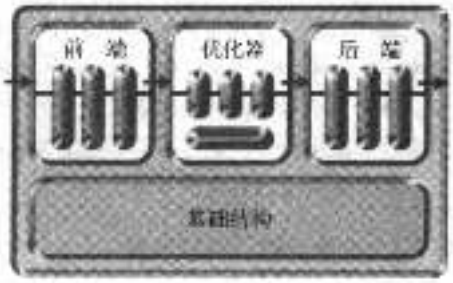
操作码	源操作数	目标操作数	含 义
cmp_LE	r_1, r_2	r_3	$r_1 \leq r_2 \Rightarrow$ true $\rightarrow r_3$
cmp_EQ	r_1, r_2	r_3	$r_1 = r_2 \Rightarrow$ true $\rightarrow r_3$
cmp_NE	r_1, r_2	r_3	$r_1 \neq r_2 \Rightarrow$ true $\rightarrow r_3$
cmp_GE	r_1, r_2	r_3	$r_1 \geq r_2 \Rightarrow$ true $\rightarrow r_3$
cmp_GT	r_1, r_2	r_3	$r_1 > r_2 \Rightarrow$ true $\rightarrow r_3$
tbl	r_1, l_2	—	r_1 might hold l_2

ILOC另一种分支语法

操作码	源操作数	目标操作数	含 义
comp	r_1, r_2	cc_1	置cc_1
cbr_LT	cc_1	l_1, l_2	$cc_1 =$ LT $\Rightarrow l_1 \rightarrow$ PC （否则 $l_2 \rightarrow$ PC）
cbr_LE	cc_1	l_1, l_2	$cc_1 =$ LE $\Rightarrow l_1 \rightarrow$ PC
cbr_EQ	cc_1	l_1, l_2	$cc_1 =$ EQ $\Rightarrow l_1 \rightarrow$ PC
cbr_GE	cc_1	l_1, l_2	$cc_1 =$ GE $\Rightarrow l_1 \rightarrow$ PC
cbr_GT	cc_1	l_1, l_2	$cc_1 =$ GT $\Rightarrow l_1 \rightarrow$ PC
cbr_NE	cc_1	l_1, l_2	$cc_1 =$ NE $\Rightarrow l_1 \rightarrow$ PC

672

附录B
数据结构



B.1 概述

精心制作成功的编译器需要注意很多细节。本附录揭示在编译器的设计和实现中引发的一些算法问题。在大多数情况下，这些细节应该不同于正文中的相关讨论。我们把它们收集起来形成本附录，在此它们会得到必要的考虑。

本附录集中讨论支持编译的基础结构。在设计和实现这些基础结构中引发很多工程问题；编译器设计者解决这些问题的行为方式对结果编译器的速度、易扩展性以及维护编译器都有着很大的影响。作为这些问题引出的一个例子，编译器无法知道它的输入代码的大小，直到它读完它们为止；因此，前端必须被设计成可以适度扩展它的数据结构的大小以适应较大的输入文件。然而，作为一个推论，在调用紧跟前端的遍之时，编译器应该知道它的大部分内部数据结构所需的大小。生成一个带有10 000个名字的IR程序后，编译器在第二遍的时候就不应该使用大小为1024个名字的符号表。包含IR的任意文件都应该从主要数据结构的粗略大小的描述开始。

同样地，编译器的后期遍可以假设传给它们的IR程序是由编译器生成的。虽然这些遍应该做完整的错误检查工作，实现者无需像前端那样花太多的时间解释错误，并设法改正这些错误。一般的策略是构建对IR程序做全面检查，并可以为调试目的插入信息的确认遍，而更少地依赖于错误检查及在不调试编译器时报告错误。然而，整个过程中，编译器设计者应该记住他们是最有可能去检查遍之间的代码的人。花费精力和时间去使IR程序的形式更容易阅读通常会回报那些对此投入精力和时间的人。

B.2 表示集合

编译的很多问题都可以用涉及集合的项来形式化。这些问题出现于本书的很多地方，包括子集构造法（第2章），LR(1) 项目的规范集合的构造法（第3章），数据流分析（第8章和第9章）以及诸如列表调度中就绪队列的工作表（第12章）。在每一个上下文，编译器设计者必须选择一个适当的集合表示。在很多情况下，算法的有效性依赖于集合表示的细心选择。（例如，在支配者计算中的*dom*数据结构表示使用一个紧凑的数组表示所有的支配者集合及立即支配者。）

构建编译器与构建其他种类系统软件，如操作系统，之间的基本差异是编译中的很多问题可以脱机解决，例如，13.3.2节中所描述的寄存器分配的自底向上局部算法是于20世纪50年代中期为原始的FORTRAN编译器提出的，它是以Belady的MIN算法而著称的脱机页替换算法，长期以来这一算法被用作评判在线页替换算法有效性的标准。在操作系统中，这一算法只是一个学术意义上的算法，因为它是一个脱机算法。因为操作系统不知道将要需要哪些页，它无法使用脱机算法。另一方面，脱机算法对编译器是实用的，因为编译器可以在做出决定之前检查整个块。

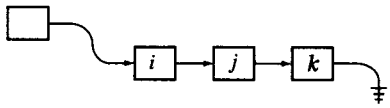
编译的脱机性质允许编译器设计者使用范围较宽的集合表示。现已提出有很多集合表示。特别地，脱机计算通常使我们把一个集合S的元素限制到一个固定大小的全域U内（$S \subseteq U$）。反过来，这又使我们可以使用比在动态发现U的大小的在线状况所适用的表示更有效的集合表示。

通用的集合操作包括*member*、*insert*、*delete*、*clear*、*select*、*cardinality*、*forall*、*copy*、*compare*、*union*、*intersect*、*difference*和*complement*。特定的应用一般只使用这些操作中的一小部分。各个集合操作的代价依赖于所选的特定表示。在为特定应用选择高效表示时，重要的是要考虑每一类操作的使用频率。要考虑的其他因素包括集合表示的内存需求和相对于U的S的期待稀疏性。

本节的其余部分集中于讨论用于编译器的三个高效的集合表示：有序链表、位向量和稀疏集。

B.2.1 把集合表示成有序表

在每一个集合都很小的情况下，使用简单的链表表示有时候很有意义。对于集合S，这一表示是由一个链表和指向这个列表的第一个元素的指针组成的。列表中的每一个结点包含一个S中元素的表示和一个指向列表中下一个元素的指针。列表中最后一个结点的指针指向一个表示列表结尾的标准值。使用链表表示，实现可以在元素（集合）上引入一个顺序来创建一个有序列表。例如，集合$S=\{i, j, k\}$，$i<j<k$的有序链接表可以有如下形式：



元素按升序保存。S的表示的大小与S中元素的个数成正比，而不与U的大小成正比。如果$|S|$比$|U|$小很多，那么仅表示存在于S中的元素的节约可以大大抵消每一个元素中的指针所带来的额外代价。

列表表示特别灵活。因为在列表中没有依赖于U的大小或依赖于S的大小的工作，它可以用于编译器发现U或S或两者的情况，例如图着色寄存器分配器的活性区域寻找部分。

675 图B-1中的表格给出使用这一表示的通用集合操作的渐近复杂度。有序链表上的大多数集合操作的复杂度是$O(|S|)$，因为为了执行这些操作，有必要遍历链表。如果释放存储单元操作无需遍历列表就可以释放各个元素的结点，如在某些垃圾收集系统或基于实存块的系统中那样，那么*clear*花费常量时间。

操 作	有序链表	位向量	稀疏集
member	$O(\|S\|)$	$O(1)$	$O(1)$
insert	$O(\|S\|)$	$O(1)$	$O(1)$
delete	$O(\|S\|)$	$O(1)$	$O(1)$
clear	$O(1)$	$O(\|U\|)$	$O(1)$
select	$O(1)$	$O(\|U\|)$	$O(1)$
cardinality	$O(\|S\|)$	$O(\|U\|)$	$O(1)$
forall	$O(\|S\|)$	$O(\|U\|)$	$O(\|S\|)$
copy	$O(\|S\|)$	$O(\|U\|)$	$O(\|S\|)$
compare	$O(\|S\|)$	$O(\|U\|)$	$O(\|S\|)$
union	$O(\|S\|)$	$O(\|U\|)$	$O(\|S\|)$
intersect	$O(\|S\|)$	$O(\|U\|)$	$O(\|S\|)$
difference	$O(\|S\|)$	$O(\|U\|)$	$O(\|S\|)$
complement	—	$O(\|U\|)$	$O(\|U\|)$

图B-1 集合操作的渐进时间复杂度

当全域未知时，这一想法的一个变形很有意义，集合可以合理地增大，如在冲突图构造法中那样（参见第13章）。使每个结点保存固定数量（大于1个）集合元素将显著减小空间和时间上的负担。每一

个结点保存k个元素，构建n个元素的集合需要$\lceil n/k \rceil$次分配和$\lceil n/k \rceil + 1$个指针，而单一元素结点集合将需要n次分配和$n+1$个指针。这一设计保留了列表表示的易扩展性，又减小了空间负担。与每一个结点对应于一个元素的集合相比，插入和删除要移动更多的数据。然而，它们的渐进复杂度仍然是$O(|S|)$。⊖ 676

用于支配计算（参见9.3.2节）中的*dom*数组，把列表表示巧妙地运用到一个特殊的情况。特别地，编译器知道全域的大小和集合的数量。使用有序集合，编译器还知道，如果$e \in S_1$且$e \in S_2$，那么S_1中e之后的每一个元素也都在S_2中这一特殊性质。因此，它们可以共享从e开始的元素。通过使用数组表示，元素名字可以用作指针。这使n个元素的单一数组把n个稀疏集合表示成有序列表。这还为那些集合生成快速插入操作符。

B.2.2 把集合表示成位向量

编译器设计者常常使用*位向量*来表示集合，特别是那些用于数据流分析（参见8.6节和9.2节）中的集合。对于一个有界全域U，包含于U的集合$S(S \subseteq U)$ 可以用长度$|U|$的位向量表示，这一表示被称为S的*特征向量*（characteristic vector）。对于每一个$i \in U$，$0 \leqslant i < |U|$，如果$i \in S$，特征向量的第i个元素等于1。否则它等于0。例如，包含于U的集合$S=\{i, j, k\}$，$i<j<k$的特征向量有如下形式：

0	…	i-1	i	i+1	…	j-1	j	j+1	…	k-1	k	k+1	…	|U|-1
0	…	0	1	0	…	0	1	0	…	0	1	0	…	0

位向量表示总是分配足够大的空间以表示所有U中的元素；因此，这一表示只能用于U是已知的应用中，也就是一个脱机应用。

图B-1中的表格列出使用这一表示的通用集合操作的渐进复杂度。尽管很多操作是$O(|U|)$，但是，如果U较小，它们仍然有较高的效率。一个字可以持有很多元素；相比每一个元素需要一个字的表示，这一表示可以得到常量倍的改进。因此，例如，使用32位大小的字，32个或少于32个元素的任意全域有单一字表示。

这一表示的紧凑性改进操作速度。使用单一字集合，很多集合操作变成单一机器指令；例如，*union*变成一个逻辑或操作，而*intersection*则变成一个逻辑与操作。即使这一集合需要多个字表示，很多集合操作的执行所需的机器指令的数量也可以按机器的字的大小减小。 677

B.2.3 表示稀疏集合

对于固定的全域U和包含于U的集合S，S是稀疏集，如果$|S|$远小于$|U|$。在编译中所遇到的某些集合是稀疏的。例如，用于寄存器分配的LIVEOUT集合是典型的稀疏集合。编译器设计者常常使用位向量来表示这样的集合，这归因于它们在时间和空间上的高效性。然而，只要有足够的稀疏性，那么更高效的时间表示是可能的，特别是当大部分操作的时间都在$O(1)$或$O(|S|)$时更是如此。相对而言，位向量集合在这些操作上花的时间要么是$O(1)$要么是$O(|U|)$。如果$|U|$与$|S|$的比大于一个字的大小，那么位向量可能是效率较低的选择。

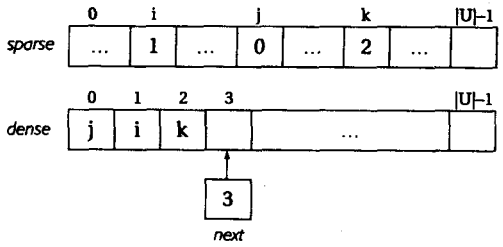
具有这些性质的稀疏集合表示，使用长度为$|U|$的两个向量和一个标量来表示这个集合。第一个向量*sparse*持有这一集合的稀疏表示；第二个向量*dense*持有这一集合的密集表示。标量*next*持有*dense*中这一集合的下一个新元素可以被插入的位置下标。当然，*next*也持有这一集合的势。

当创建一个稀疏集时，这两个向量不需要初始化；集合成员测试保证处理时每一入口的有效性。*clear*操作简单地把*next*设置回它的初始值0。为了把一个新元素$i \in U$加入S中，代码（1）把i存储到

⊖ 对于单一的链表，在列表的前面而不是后面保存额外的空间可以简化*insert*和*delete*。

*dense*中*next*的位置上；（2）把*next*的值存储在*sparse*中第*i*个位置上；（3）递增*next*，使其成为下一个元素在*dense*中的插入位置的索引。

如果我们从一个空稀疏集合S开始，且按j、i、k的顺序增加元素，其中$i<j<k$，那么这一集合的形式如下：



678 注意，稀疏集表示需要足够的空间以表示整个U。因此，它只能用于编译器知道U的大小的脱机状况。

因为*sparse*和*dense*中的每一个元素*i*的有效入口必须互相指向对方，所以可以使用下面的测试来确定成员关系。

$$0 \le sparse[i] < next \quad \text{and} \quad dense[sparse[i]] = i$$

图B-1中的表格列出通用集合操作的渐进复杂度。因为这一方案包含集合的稀疏表示和密集表示，所以这一方案具有两者的某些优势。集合中的各个元素可以通过*sparse*在时间O(1) 内得到处理，而必须遍历这一集合的集合操作可以使用*dense*得到O(|S|) 的复杂度。

当在位向量和稀疏集表示之间进行选择时，应该既考虑空间又考虑时间的复杂度问题。稀疏集表示需要两个长度为|U|的向量和一个标量。相反，位向量表示需要一个长度为|U|的向量。如图B-1所示，稀疏集表示在渐进时间复杂度方面优于位向量表示。然而，因为位向量集合操作的高效实现的可能性，所以在S不是稀疏的情况下位向量是首选。当在这两个表示之间选择时，重要的是要考虑被表示集合的稀疏性和被使用的集合操作的相对频率。

B.3 实现中间表示

在选择了特定类型的IR之后，编译器设计者必须决定如何实现它。乍看起来，这一选择似乎很显然。使用指针和堆分配数据结构，DAG很容易被表示成结点和边。四元组很自然地被表示成一个$4 \times k$的数组。然而，对于集合，选择最好的实现需要对编译器如何使用这一数据结构有更深的理解。

B.3.1 图式中间表示

如第5章所讨论的那样，编译器使用若干图式IR形式。使图的实现适合编译器的需求可以改进编译
679 器的时间和空间效率。本节描述使用树和图所发引发的一些问题。

1. 树的表示

在大多数语言中，树的自然表示是一组由指针连结起来的结点。在编译器构建树的时候，典型的实现是按需求分配结点。树可能包含不同尺寸的结点，例如改变结点的子结点的数量及某些数据域。另一种选择是使用单一种类的结点构建树，把每个结点分配成适合最大可能结点的结点。

表示相同的树的另一种方法是结点结构的数组。在这一表示中，指针被整数索引所取代，而且基于指针的引用变成标准的数组和结构引用。这一实现构建单一尺寸结点，而其他部分则与基于指针的实现类似。

这些方案各有优缺点。

- 指针方案处理任意大小的AST。当AST的大小超过最初分配的尺寸时，数组方案需要扩展数组的代码。
- 指针方案需要为每一个结点做一次分配，而数组方案只是递增记数器（除非它必须扩展数组）。像基于实存块的分配（参见第6章中的栏外标题：基于实存块的分配）那样，某些技术可以降低分配和回收的代价。
- 指针方案有完全依赖于运行时分配器行为的引用局部性。数组技术使用连续内存位置。特定系统可能更适合特定的方案。
- 指针方案更难优化，因为对指针密集代码的静态分析的质量相当不理想。相反，为密集的线性代数代码而开发的很多优化可以运用于数组方案。当编译器被编译时，这些优化可能产生比指针方案代码更快的数组方案代码。
- 指针方案与数组实现相比可能很难调试。程序员似乎发现数组索引比内存寻址更直接。
- 如果AST必须被写到外部媒介中，那么指针方案需要编码指针的方法。这大概包含按指针来遍历各结点。数组系统使用相对于数组开始位置的偏移，所以不需要进行翻译。在很多系统上，这可以通过大的模块I/O操作来实现。

680

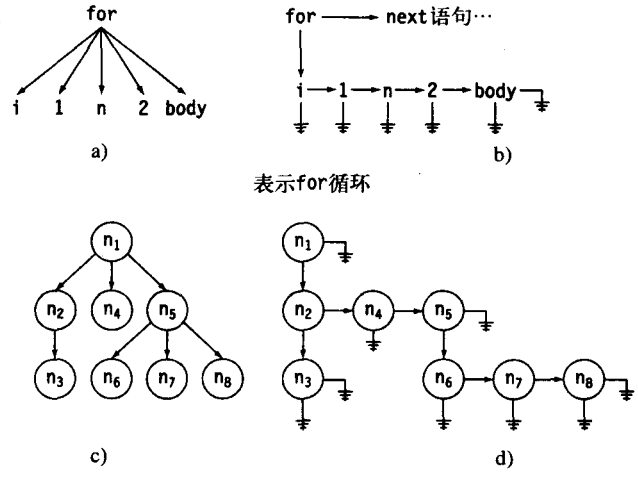
还存在其他很多衡量的尺度。每一种都要在上下文中来评估。

2. 把树映射到二叉树

抽象语法树的一个直截了当的实现也许支持带不同子结点的结点。例如，典型的for循环头部

```
for i=1 to n by 2
```

在AST中可能是如图B-2a) 所示带5个子结点的结点。其中，标签为body的结点表示for循环体子树。



图B-2 把任意树映射到二叉树

681

对于某些结构，无法固定子结点的数量。为了表示一个过程调用，AST必须或者基于参数的数量调整分配结点，或者使用带有参数列表的单一子结点。前者的方法复杂化遍历AST的所有代码；可变尺寸的结点必须保存指示有多少个子结点的数字，而遍历必须包含读取这些数字并相应修改遍历行为的代码。后者的方法把AST的实现从它对源头的强大的依附中分离出来，而使用列表这种容易理解的结构来表示

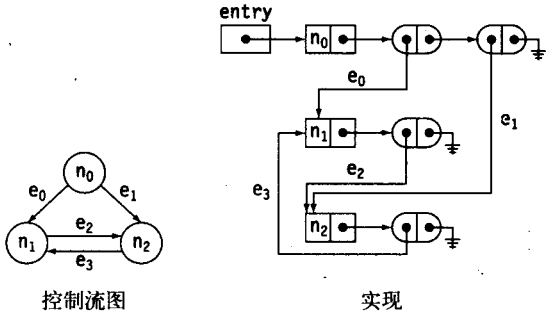
固定参数结点不适合的那些地方。

为了简化树的实现，编译器设计者可以进一步采用这种形式与含义的分离。可以把任意的树映射到二叉树上，其中二叉树的每一个结点都刚好有两个子结点。在这一映射中，左子结点指针指向最左边的子结点，右子结点指针指向当前水平上的下一个兄弟结点。图B-2b)给出映射到二叉树的带5个子结点的for结点。因为每一个结点都是二叉的，所以这棵树在每一个叶结点处有空指针。它还在for结点有一个兄弟指针；在左侧的版本中，这一指针出现在for结点的父结点中。此图B-2c)和图B-2d)给出更复杂的例子。

使用二叉树在树中引入额外的空指针，如这两个例子所示。但它以若干方式简化实现。内存分配可以简单地完成，使用基于实存块的分配器或定制分配器。编译器设计者也可以把树作为结构数组实现。处理二叉树的代码在某种程度上比处理可变数量结点树的代码更简单。

3. 表示任意图

编译器必须表示的若干结构是一般的图而不是树。其例子包括控制流图和数据优先图。一个简单实现可以使用堆分配结点，并用指针来表示边。图B-3左边给出一个简单的cfg。显然，它需要三个结点。边带来困难；每个结点需要多少个入边和出边？每个结点可以维护一个出边列表；这导致此图右边所示的一个实现。



图B-3 控制流图示例

在这一图表中，长方形表示结点。椭圆表示边。这一表示很容易沿着边的走向遍历此图。它不提供对结点的随机存取，为了对此作出补偿，我们可以加入一个结点指针数组，并以结点的整数名字为索引。
682 由于这一小小的增加（在图中没有显示出来），此图更适合解决前向数据流问题。它为寻找一个结点的所有后继提供快速方法。

遗憾的是，编译器通常需要沿着边反方向遍历这个cfg。例如，这一情况出现在向后数据流问题中，此时算法需要快速前驱操作。为使此图的结构适合向后遍历，我们需要增加另一个指向每个结点的指针，并创建第二个边结构集合来表示一个结点的前驱。这一方法的确可行，但是这一数据结构变得复杂从而难以画出、实现和调试。

与树类似，另一个方法是把此图表示成一对表格：一个是结点的表格，另一个是边的表格。结点表格有两个域：一个域是到后继的第一条边，另一个域是到前驱的第一条边。而边的表格有四个域：第一对域保存所表示边的源和目标，而另外一对域保存源头的下一个后继和目标的下一个前驱。使用这一方案，我们的示例cfg的表格如下所示。

结 点

名 称	后 继	前 驱
n_0	e_0	—
n_1	e_2	e_0
n_2	e_3	e_1

边

名　称	源	目　标	下一个后继	下一个前驱
e_0	n_0	n_1	e_1	e_3
e_1	n_0	n_2	—	e_2
e_2	n_1	n_2	—	—
e_3	n_2	n_1	—	—

683

这一表示提供对后继和前驱的快速存取，以及通过各个结点和边的名字对它们的快速存取（假设这些名字是由小整数表示的。）

表格表示对于遍历图和寻找前驱和后继都很有效。如果频繁地在这一图上运用其他操作，那么可以找到更好的表示。例如，图着色寄存器分配器中的支配操作在冲突图中试测一个边的存在，并在一个结点的邻居上进行迭代。为了支持这些操作，大多数实现使用两个不同的图表示（参见13.5.3节）。为了回答成员问题，即边（i, j）在图中吗？这些实现使用一个位矩阵。因为冲突图是无向的，所以下三角位矩阵就足够了，这大致可以节省完整位矩阵所需空间的一半。为了在一个结点的邻居上快速迭代，要使用一组邻接向量。

因为冲突图既大又是稀疏的，所以邻接向量的空间可能成为问题。某些实现使用两遍操作来构建此图，第一遍计算每一个邻接向量的尺寸，第二遍构建向量，每一个向量带最小所需尺寸。其他实现使用B.2.1节的集合列表表示的变形，构建此图只使用一遍操作，对邻接向量使用无序列表，且每个列表结点有多个边。

B.3.2 线性中间形式

如ILOC这样的线性中间形式在概念上吸引人的地方是，作为结构数组它们有一个简明的实现。例如，ILOC程序有一个到FORTRAN风格的数组的直接映射，n个ILOC操作映射到（$n \times 4$）元整数数组。操作码决定如何解释每一个操作数。当然，任何设计决策都有其优点和缺点，而且希望使用线性IR的编译器设计者应该考虑简单数组之外的其他表示。

1. FORTRAN风格数组

使用整数组数来保存IR保证对各个操作码和各操作数的快速存取，并降低分配和存取的负荷。处理IR的遍应该快速运行，因为使用标准分析和为了改进密集线性代数程序而开发的转换可以改进所有数组的存取。经过代码的一个线性遍拥有可预测内存位置；因为连续的操作占据连续内存位置，它们在缓冲器中不可能产生冲突。如果编译器必须把IR写到外部媒介中（例如，在两个遍之间），那么它可以使用有效的块I/O操作。

684

然而，数组实现也存在缺点。如果编译器需要把一个操作插入代码中，那么它必须为新操作创建空间。同样地，删除也将缩小代码。任意种类的代码移动都会遇上其中的某些问题。朴素的实现可以通过混洗操作来创建这一空间；采用这一方法的编译器通常在数组中在分支和跳转之后留有空槽，以便减少所需的混洗数量。

另外一个策略是使用detour操作符把对IR的任意遍历导向一个离线代码片段。这一方法使得编译器在离线代码片段中穿插控制，这样可以通过使用detour复写一个现存的操作来完成一个插入，这把被插入代码和被复写操作放到数组的末端，然后是一个detour，它在第一个detour之后返回这个操作。这一策略的最后一个阶段是间断性地线性化detour，例如，在每一遍的末尾，或当detour的数目超过某个限度时进行。

伴随着数组实现的另一个复杂性是由这样的间断性操作的需求引起的，例如对取可变操作数的ϕ函数的需求。在得到ILOC的编译器中，过程调用由一个复杂的操作表示。这一调用操作对每一个形参有一个操作数，（如果需要）有一个返回值的操作数，以及调用可能修改和使用的两个值列表操作数。这一操作不适合$n \times 4$个元素数组模型，除非操作数被解释成指向参数、被修改变量和已使用变量的列表的指针。

2. 结构列表

数组实现的另外一个方法是使用结构列表。在这一方案中，每一个操作有独立的结构和指向下一个操作的指针。因为可以为每一个操作分别分配结构，所以程序表示很容易扩展到任意尺寸。因为顺序是由链接操作的指针引入的，所以可以直接使用指针赋值插入和移除操作，而无需混洗或拷贝。如前所述的调用操作这样的可变长度操作可以使用不同的结构来处理；事实上，诸如loadI和jump这样的短操作，
685 也可以使用不同的结构处理以节省空间。

当然，使用分别分配的结构会增加分配负担，数组需要一次初始分配，而列表方案对每一个IR操作需要一次分配。列表指针增加所需的空间。因为所有处理IR的编译器遍必须包含很多基于指针的引用，这些遍的代码可能比使用简单数组实现的代码慢，因为基于指针的代码通常比数组密集代码更加难以分析和优化。最后，如果编译器在遍之间把IR写入外部媒介，那么在它写IR以及从外部媒介读取并再构建这一列表时，它必须遍历这一列表。这使I/O慢下来。

在某种程度上，可以通过在实存块或数组中实现结构列表来改善这些缺点。使用基于实存块的分配器，分配的代价降到在典型的情况下的一个测试和一个加法。[⊖]实存块也大致产生与简单数组实现相同的局部化。

在数组中实现列表可以达到相同的目标，并具有所有指针都变成整数索引的优点。经验表明这可以简化调试；也使得使用块I/O操作写和读IR成为可能。

B.4 实现散列表

散列表实现中的两个核心问题是保证散列函数（对它使用的所有尺寸的表）产生整数的均匀分布以及以高效的方式处理冲突。寻找好的散列函数很困难。好在，散列已经使用了很长时间，文献中已描述了许多优秀的散列函数。

本节的其余部分描述实现散列表中所引发的设计问题。B.4.1节给出实践中生产好结果的两个散列函数。下面的两节给出最为广泛使用的两个解决冲突的策略。B.4.2节描述*开放散列*（open hashing）策略（有时称为*桶式散列*（bucket hashing）），而B.4.3节给出另一个称为*开放寻址*（open addressing）即*再散列*（rehashing）的策略。B.4.4节讨论散列表的存储管理问题，而B.4.5节给出如何把词法作用域结合到
686 这些方案中。最后一节处理编译器开发环境中引发的特殊问题，即对散列表定义的频繁修改问题。

组织符号表

在设计符号表时，编译器设计者面临的第一个决策是符号表的组织和它的搜索算法。正如在其他很多应用中那样，编译器设计者有若干选择。

线性列表 线性表可以扩展到任意大小。其搜索算法是一个简单、小而紧凑的循环。遗憾的是，搜索算法对于每一次查表平均需要$O(n)$次探查，其中n是表中符号的数量。这一缺点几乎总是超过

⊖ 在第一遍之后的任意遍中，编译器应该有关于IR大小的相当精确的概念。因此，它可以分配既持有IR又持有若干附加空间的实存块，并避免扩展实存块的高昂代价。

实现和扩展的简单性。为了证实使用线性列表的正当性，编译器设计者需要强有力的证据，证明被编译的过程有非常少的名字，如面向对象语言中可能发生的那样。

二分搜索 为了在改进线性表的搜索时间的同时保留其易扩展性，编译器设计者可能使用平衡二叉树。理想地，平衡二叉树允许对于每一次查找使用$O(\log_2 n)$次探查；这是对线性表的一个相当可观的改进。已有很多平衡搜索树的算法。（通过使用有序表的二分搜索也可以达到类似的效率，但是这个表的插入和扩展更加困难。）

散列表 散列表可以最小化存取代价。它的实现直接根据名字计算表索引。只要计算产生好的索引分布，平均存取代价将变成O(1)。然而，最坏的情况可能是线性搜索。编译器设计者可以采用一系列步骤来降低最坏情况发生的可能性，但是，病态情况仍然要出现。很多散列表实现对于扩展来说有成本低方案。

多重集判定 为了避免最坏情况行为，编译器设计者可以使用称为多重集判定（multiset discrimination）的脱机技术。这一技术以在源文本上的一次额外的遍为代价，为每一个标识符创建不同的索引。这一技术避免了总是存在于散列中的病态行为。（详细描述参见第5章的附标题散列的替代。）

在这些组织中，最通常的选择是散列表。与线性表和二叉树相比，它提供更好的编译时行为，而且其实现技术也得到广泛的研究和传授。

687

B.4.1 选择散列函数

无论如何强调优秀散列函数的重要性都不过分。产生不好索引值分布的散列函数，直接增加把各项加插入表中并在后来寻找这些项的平均代价。幸运的是，文献中描述了很多好的散列函数，包括由Knuth给出的乘法散列函数和由Cormen等给出的通用散列函数。

1. 乘法散列函数

乘法散列函数（multiplicative hash function）看似简单。程序员选择一个常量C并把它用于下面的公式中：

$$h(key) = \lfloor TableSize \cdot ((C \cdot key) \bmod 1) \rfloor$$

其中C是这个常量，*key*是被用作进入这一表格的键值的整数，而*TableSize*显然是这一散列表的当前尺寸。Knuth给出了C的值 0.618 033 988 7 $\approx (\sqrt{5}-1)/2$。这一函数的效应是计算$C \cdot key$，使用模函数取其小数部分，并把结果与表的尺寸相乘。

2. 通用散列函数

为了实现*通用散列函数*（universal hash function），程序员设计一个可以以一小组常量为参数的函数族。在执行时随机地选取一组常量值：或者为这些常量值选取随机数字，或者通过随机选取下标在已经测试过的常量的集合中进行选取。（在使用散列函数的程序的每一次执行过程中使用相同的常量，但是，这组常量会随执行而变化。）通过改变每一次执行的散列函数，通用散列函数在程序的每一次运行中产生不同的分布。对于编译器，如果输入程序在某个特定编译中产生病态行为，那么很有可能在后继的编译中不产生相同的病态行为。为了实现乘法散列函数的通用版本，编译器设计者可以在编译开始时随机生成C的一个适当值。

688

拙劣的散列函数的危险

散列函数的选择对表的插入和查找代价产生关键的影响。这是一个少量的注意可能产生较大差异的例子。

很多年前，我们看到一名学生为字符串实现下面的散列函数：（1）把键值分成4字节块，（2）把它们异或到一起，（3）取其结果数字e模表的尺寸作为索引。这一函数相当快。它有一个直接而高效的实现。对于某些表尺寸，它产生适当的分布。

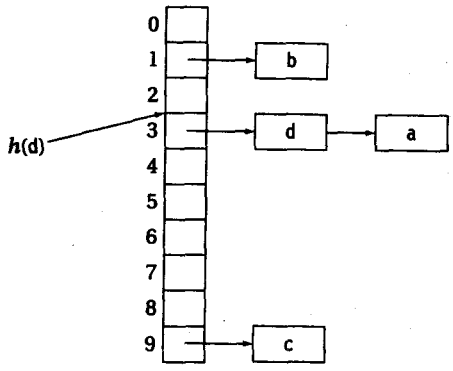
当这一学生把这一实现插入到一个在FORTRAN程序上执行源语言到源语言的转换时，若干独立的事实相结合带来了算法上的灾难。第一，这一实现的语言把字符串的右侧附上空格以达到4字节边界。第二，这个学生选择表的初始尺寸为2048。最后，FORTRAN程序员使用很多一个字符或两个字符的变量名，如i、j、k、x、y和z。

所有短变量名都适合单一字，避免了异或的任意效应。然而，取e模2048删去了除e的最后11个位之外的所有位。因此，所有短变量名产生相同的索引，一对空格的最后11位。这一散列搜索立即变成了线性搜索。在这一特殊散列函数与理想相差甚远的同时，把表的尺寸改成2047可以消除这最引人注目的负效应。

B.4.2 开放散列

开放散列（open hashing），也称为*桶式散列*（bucket hashing），假设散列函数h产生冲突。它依赖于h来把输入键值集合划分成一个固定数量的集合，即*桶*（bucket）。每一个桶包含一个记录的线性列表，每一个名字一个记录。*LookUp*(n) 遍历存储标签$h(n)$ 为下标的桶内的线性列表来寻找n。因此，*LooUp*需要$h(n)$ 的一个评估和一个线性列表的遍历。评估$h(n)$ 应该很快完成；而列表的遍历所花的时间与列表长度成正比。对于尺寸为S且有N个名字的散列表，每一次查找的代价大致为O(N/S)。只要h相当一致地分布名字且名字与桶的比率很小，那么查找代价近似于我们的目标：对于每一次存取，查找的次数为
689 O(1)。

图B-4给出使用这一方案实现的一个小型散列表。它假设h(a)＝h(d)＝3带来一个冲突。因此，a和d占据表中相同槽。这一列表结构把a和d链接起来。为了提高效率，*Insert*应该把数据添加在桶的前端。



图B-4 开放散列表

开放散列有几个优点。因为它为每一个插入的名字在链接列表之一创建一个新的结点，所以它可以

在不用尽空间的情况下，处理任意数量的名字。在一个桶内有大量的条目不影响其他桶内的存取代价。因为桶集合的具体表示通常是一个指针数组，所以增加S的负担很小：对每一个桶需要一个指针。（这使得维持S/N在较小的范围内的代价较小。每一个名字的代价是一个常量。）选择S为二的幂可以减小实现h所需的取模操作的代价。

开放散列的主要缺点直接与这些优点相关。概括起来有两点。

1）开放散列可能频繁地分配。每一个插入分配一个新记录。在内存分配很费时的系统上实现时，这可能很可观。使用不费时的分配机制，如基于实存块分配（参见第6章的附标题）可以缓解这一问题。

2）如果任意特定集合变大，那么*LookUp*则退化成线性搜索。使用合理行为的散列函数，当N比S大得多时，上述情况才有可能出现。实现应该发现这一问题，并扩大桶数组。典型地，这涉及分配一个新的桶数组，并把原表中每一条目重新插入新表中。 690

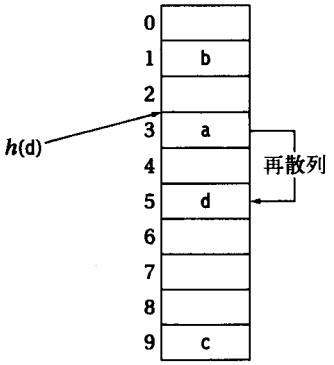
实现良好的开放散列表在空间和时间上以低负荷提供有效的存取。

为了改进在单一桶内执行线性搜索的行为，编译器可以动态地记录桶链。Rivest和其他人[291，309]给出两个有效的策略：在每一次查找中把查找到的结点提升一个位置，或在每一次查找中把它移到列表的前面。也可以使用更复杂的方案组织每一个桶。然而，编译器设计者在对这一问题投放更多努力之前，他们应该评估遍历一个桶所损失的时间总量。

B.4.3 开放寻址

开放寻址（open addressing），也称*再散列*（rehashing），当在正常情况下$h(n)$所在的槽已被占据时，为$h(n)$计算一个替换索引，以此来处理冲突。在这一方案中，*LookUp*(n)计算$h(n)$并检查那个槽。如果这个槽是空的，那么*LookUp*失败。如果*LookUp*找到n，它成功。当它找到一个名字但不是n时，它使用第二个函数$g(n)$计算这次搜索的增量。这导致它检查（$h(n)+g(n)$）模S处的表格，然后是（$h(n)+2\times g(n)$）模S处的表格，再然后是（$h(n)+3\times g(n)$）模S处的表格，依此类推，直到它或者找到n，或者发现一个空槽，或者再次返回到$h(n)$（这一表被从0标号到$S-1$，这保证模S将返回一个有效的表索引。）如果*LookUp*找到一个空槽，或再次返回$h(n)$，那么查找失败。

图B-5给出用这一方案实现的小型散列表。它使用与图B-4相同的数据。如从前一样，h(a)$=h$(d)$=3$，而h(b)$=1$且h(c)$=9$。当d被插入时，它与a产生一个冲突。第二个散列函数g(d)产生2，所以*Insert*把d放在表格中下标为5的地方。事实上，开放寻址构建类似于那些用于开放散列中的条目链。然而，在开放式寻址中，这一链被直接存储于表格之中，而且单一表格位置可以充当多个链的开始点，每一个带有由g产生的不同增量。



图B-5 开放地址表

这一方案进行了时间和空间之间的微妙权衡。因为每一个键都被存储于表中，S必须比N大。如果由于h和g产生良好分布而冲突不频繁，那么再散列链保持较短而且处理代价保持较低。因为它可以廉价地
691 再计算g，所以这一方案无需存储指针形成再散列链，可以节省N个指针。这一空间上的节省允许更大的散列表，而更大的表可以通过降低产生冲突的频率而改进执行。开放寻址的主要优点很简单：通过较短的再散列链降低存取代价。

开放寻址有两个主要的缺点。当N靠近S而表变满时，就会引发这两个缺点。

1）因为再散列链穿过索引表格，n和m之间的冲突可能会干扰某个其他名字p的后继插入。如果$h(n)=h(m)$且$(h(m)+g(m) \bmod S=h(p)$，那么插入n后再插入m填充表中p的槽。当这一方案行为良好时，这一问题的影响较小。当N靠近S时，它会变得显著。

2）因为S至少和N一样大，所以如果N太大时，必须扩展这个表。（同样地，当某个链变得太大时，实现也要扩展S。）扩展是正确性的需要；对于开放散列，这是有效性的关键。

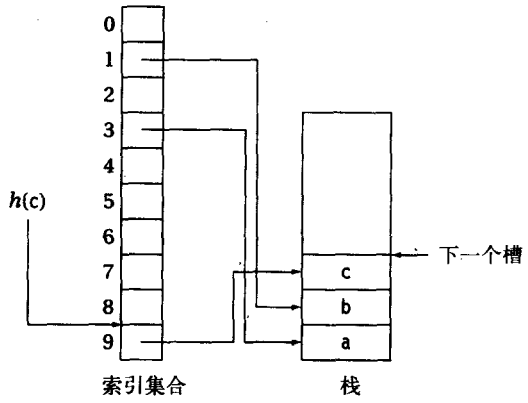
某些实现对g使用常量函数。这简化实现并减小计算第二索引的代价。然而，它为h的每一个值创建
692 单一再散列链，并在第二索引遇到一个已被占据的表格槽的地方产生合并再散列链的效应。这两个缺点超过了评估第二散列函数的代价。更合理的选择是使用两个带有不同常量的乘法散列函数：如果可能，在开始时从常量表做随机选取。

表格的尺寸S在开放寻址中起着重要的作用。*LookUp*必须识别它是否达到了已访问表槽；否则，它将不会停止或失败。为了使这有效，实现应该保证它最终返回到$h(n)$。如果S是一个素数，那么$0<g(n)<S$的任意选择生成一系列探查，p_1、p_2、…、p_S且有性质$p_1=p_S=h(n)$且对于任意满足$1<i<S$的i，$p_i \neq h(n)$。因为实现可能需要扩展表格，所以它应该包含适当素数大小的表。由于对程序大小和编译器可用内存的实际限制，一小组素数就足够了。

B.4.4 存储符号记录

开放散列和开放寻址都不直接处理如何给与每一个散列表条目相关的信息分配空间。对于开放散列，其诱惑是直接分配实现链的结点中的记录。对于开放寻址，其诱惑是避免指针并使索引表中的每一个条目是符号记录。这两个方法都有缺点。我们可以通过使用一个独立的已分配栈来保存这些记录而到达更好的结果。

图B-6描述这一实现。在开放散列实现中，链表本身可以在栈上实现。这降低分配各个记录的代价，当分配是一个耗时操作时尤为如此。在开放寻址实现中，再散列链仍隐含于索引集合中，这保持作为这一方案的动机的空间节省。



图B-6 保存记录的栈分配

当实际记录存储于栈中时，它们形成一个稠密表，它对外部I/O更适合。对于耗时的分配，这一方案把这一较大的分配代价平摊到很多记录上。对于垃圾回收器，它减少必须被标记和回收的对象数量。无论在哪种情况下，拥有稠密表都可以使表中符号上的迭代更有效：这是编译器用于执行诸如赋存储位置等任务的操作。

作为最后一个优点，这一方案大大简化扩展索引集合的任务。为了扩展索引集合，编译器放弃旧的 693
索引集合，分配一个更大的集合，然后从栈底到栈顶重新把记录插入新表中。这消除暂时在内存同时拥有新旧表格的需要。与跟踪指针来遍历开放散列中的列表相比，在稠密表上进行迭代的工作量一般更少。它避免在空表槽中的迭代，而这可能在开放寻址扩展索引集合以使链变短时发生。

编译器无需把整个栈作为单一对象来分配。相反，对于某个适当的k，这个栈可以作为含有k个记录的结点的链来实现。当一个结点变满时，这一实现分配一个新结点，把它加到链的末端，并继续下去。这为编译器设计者提供掌管分配代价和浪费空间之间的折中的细微控制。

B.4.5 增加嵌套词法作用域

5.7.3节描述了创建处理嵌套词法作用域的符号表中引发的问题。它描述创建符号表束的简单实现，每一层创建一个。虽然实现在概念上是清楚的，但是它把作用域的负荷推给*LookUp*，而不是*InitializeScope*、*FinalizeScope*和*Insert*。因为编译器调用*LookUp*的次数比它调用其他程序的次数多很多，所以其他实现也值得考虑。 694

再次考虑图5-10中的代码。它生成如下动作。

↑ ⟨w,0⟩ ⟨x,0⟩ ⟨example,0⟩ ↑ ⟨a,1⟩ ⟨b,1⟩ ⟨c,1⟩

↑ ⟨b,2⟩ ⟨z,2⟩ ↓ ↑ ⟨a,2⟩ ⟨x,2⟩ ↑ ⟨c,3⟩, ⟨x,3⟩ ↓ ↓ ↓ ↓

其中，↑表示对*InitializeaScope*的一个调用，↓表示对*FinalizeScope*的一个调用，而序对<name, *n*>表示对*Insert*的一个调用来把name加到层次n上。

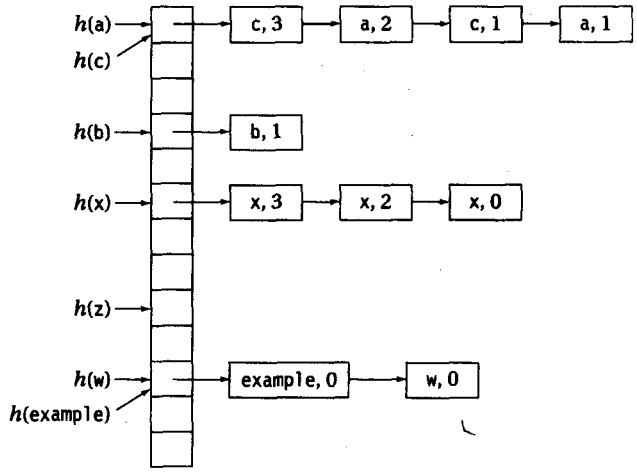
1. 把词法作用域加入到开放散列

考虑当我们把词汇等级域加到每一个名字的记录中，并在链的前面插入新名字时，在开放散列表中将发生什么。*Insert*可以通过比较名字和词法等级来检查重复。对于给定名字，*LookUp*返回它发现的第一个记录。*InitializeScope*将增加当前词法等级的计数器。这一方案把复杂性推向*FinalizeScope*，*FinalizeScope*不仅必须降低当前词法等级，而且还必须移除释放了的作用域中插入的所有名字的记录。

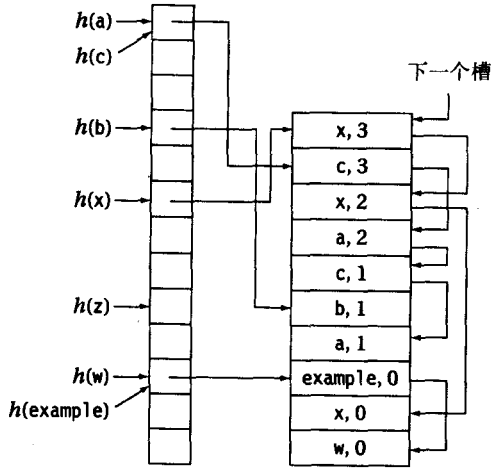
如果如图B-4所示开放散列是通过分配链中每一个结点来实现的，那么*FinalizeScope*必须找到要丢弃作用域中的所有记录，并把它们从相应的链中消除。如果它们后来不再在编译器中使用，那么*FinalizeScope*必须释放它们；否则，它必须把它们连到一起保存起来。图B-7给出这一方法在图5-10的赋值语句处产生的表。

使用栈分配记录，*FinalizeScope*可以从栈顶向下迭代，直到它达到被丢弃等级下面的某个等级的记录为止。对于每一个记录，它使用链中指向下一个条目的记录指针来更新索引集合入口。如果这一记录将被放弃，那么*FinalizeScope*把指针重新设置为指向下一个可用槽；否则，这些记录被保留在栈上。图B-8给出我们例子在赋值语句处的符号表。

无须太多考虑就可以把链的动态重排序加入这一方案中。因为*FinalizeScope*使用栈顺序，而不是链顺序，它仍然在栈的顶部找到所有顶级名字。为了对链进行重排序，编译器或者需要遍历这个链来移除每一个删除名字的记录，或者使用双链表来实现链以进行更快的删除。 695



图B-7 开放散列表中的词法作用域

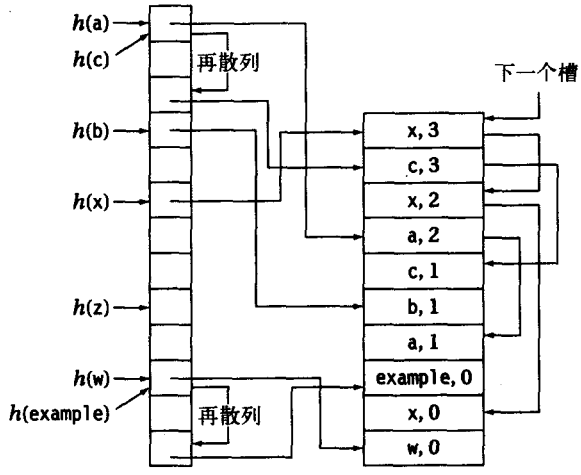


图B-8 栈分配开放散列表中的词法作用域

2. 把词法作用域加到开放寻址中

对于开放寻址表来说，情况可能会更复杂一些。表中的槽是关键的资源；当所有的槽都被填满时，在进一步的插入出现之前，必须扩展表格。在使用再散列的表中进行删除很困难；表的实现无法简单地告知再散列链的中间是否有已删除的记录。因此，把槽标记为空把通过该位置的所有链分割开来（而非在结尾处）。这证明在表中为每一个名字的每个变形存储离散的记录是不适宜的。相反，编译器应该为每一个名字只把一个记录链接到表中；它可以为陈旧变形创建替代记录链。图B-9描述我们例子的这一情况。

这一方案把大部分复杂性推给了*Insert*和*FinalizeScope*。*Insert*在栈的顶部创建新记录。如果它在索引表中发现了相同名字的陈旧声明，那么它用对新记录的引用替换这个引用，并把陈旧的引用链接到新记录上。*FinalizeScope*在栈上各顶端项间迭代，同开放散列一样。为了移除陈旧变形的记录，*FinalizeScope*简单地把索引表重新链接到陈旧记录上。为了移除记录的最后变形，它必须插入一个特殊的记录，它表示一个已删除的引用。*LookUp*必须通过在当前链占据一个槽来识别已删除引用。*Insert*必须知道它能够使用任意新插入的符号来取代已删除引用。



图B-9 开放寻址表中的词法作用域

本质上这一方案为冲突和再声明创建一个独立的链。冲突在索引集合被串起来。再声明在栈中被串起来。这将稍稍减小*LookUp*的代价，因为它避开为单一名字检查多个记录。

考虑开放散列中包含7个x的声明以及层次0处y的单一声明的桶。*LookUp*在找到y之前可能会遇到x的所有7个记录。使用开放寻址方案，*LookUp*遇到一个x的记录和一个y的记录。

B.5 一个灵活的符号表设计

大多数编译器使用符号表作为关于在源代码、在IR及在生成代码中出现的各种名字信息的中心储存仓。在编译器开发过程中，符号表中的域集合似乎在单调增大。域被加入表中以支持新遍，并使各遍历之间的信息通信。当对一个域的需求消失时，这个域可能从符号表定义中移除，也可能不移除。随着每一个域的加入，符号表的尺寸增大，且直接存取这一符号表的编译器的任意部分必须被重新编译。

我们在*R*"的实现和ParaScope程序设计环境中遇到了这一问题。这些系统的实验性质导致符号表域的增添和移除很普遍这样一种情况。为了处理这一问题，我们实现了一个更复杂但更灵活的符号表结构：一个*二维散列表*（two-dimensional hash table）。这消除几乎所有对符号表定义和实现的变更。

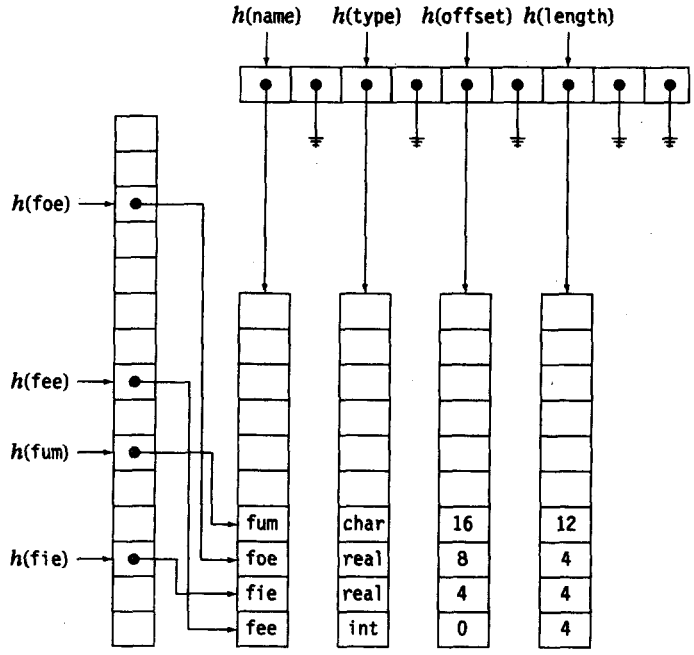
696 ~ 698

如图B-10所示的二维表使用两个不同的散列索引表。图左侧所示的第一个散列索引表对应于图B-6的稀疏索引表。实现把这个表用于符号名上的散列。图上部所示的第二个散列索引表是域名散列表。程序员通过各个域的文本名字和符号名字引用各个域；实现散列符号名来得到选择数据向量的索引和域名。理想的属性被存储在符号索引下面的向量中。它的行为就如每一个域有如图B-6那样实现的其自身的散列表一样。

这看来似乎很复杂，但它的成本并不特别高。每一个表存取需要两个散列计算，而不是一个。实现无需为给定域分配存储空间，直到有值被存储于其中；这避免未使用域的空间负荷。它允许开发者在不与其他程序员发生冲突的情况下创建和删除符号表域。

我们的实现为一个域（通过名字）设置初始值、为（通过名字）删除一个域以及为报告域使用统计等提供入口点。这一方案允许各程序员以合理而彼此独立的方式管理他们自己的符号表使用，而不受其他程序员和代码的干扰。

作为最后一个问题，实现相对于特定符号表来说应该是抽象的。也就是说，它应该总是把表的实例作为参数。这允许编译器在很多情况复用这一实现，例如第8章中的超局部或基于支配者的值编号算法。



699 图B-10 二维散列符号表

附录注释

编译器中的大多数算法都处理集合、映射、表格和图。它们的实现直接影响那些算法的时空效率，最终影响编译器本身的可用性[54]。算法和数据结构的教科书覆盖本附录总结的很多问题[220，4，185，103，39]。

我们的研究用编译器使用了这一附录中所描述的几乎所有数据结构。我们已从若干领域中数据结构的增长情况看到了若干性能问题。

- 如第5章附标题提到的抽象语法树可以大到不合理的程度。把任意树映射到二叉树的技术简化实现且似乎可以把额外负荷维持在很低的水平[220]。
- 使用后继和前驱列表的图的表格表示已经被反复重新发现。它对CFG特别适合，对CFG编译器要在后继和前驱上进行迭代。我们于1980年首次把这一数据结构运用于PFC系统。
- 数据流分析中的集合可以成长到占据几百兆字节。因为这样规模的分配和释放是性能的要因，所以我们通常使用Hanson的基于实存块的分配器[171]。

700

- 冲突图的尺寸和稀疏性使其成为值得精心考虑的另一个领域。我们为每一个结点使用多重集合元素的有序列表变形来管理空间额外负荷的同时，保持构建这样的图的代价较低[94]。

符号表在编译器保存和存取信息的方式中起着核心的作用。人们在这些表的组织上花费了大量的精力。再组建列表[291，309]、平衡搜索树[103，39]和散列[220，vol.3]都对使这些表高效中起着重要的

701 作用。Knuth[220，vol.3]和Cormen[103]详细描述了乘法散列函数。

参考文献

[1] Philip S. Abrams. *An APL machine.* PhD thesis, Stanford University, Stanford, CA, February 1970. (Technical Report SLAC-R-114, Stanford Linear Accelerator Center, Stanford University, February 1970.)

[2] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. On finding lowest common ancestors in trees. In *Conference Record of the Fifth Annual ACM Symposium on Theory of Computing* (STOC), pages 253–265, May 1973.

[4] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[5] Alfred V. Aho and Stephen C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, July 1976.

[6] Alfred V. Aho, Stephen C. Johnson, and Jeffrey D. Ullman. Code generation for expressions with common subexpressions. In *Conference Record of the Third ACM Symposium on Principles of Programming Languages*, pages 19–31, Atlanta, GA, January 1976.

[7] Alfred V. Aho, Steven C. Johnson, and Jeffrey D. Ullman. Deterministic parsing of ambiguous grammars. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 1–21, Boston, MA, October 1973.

[8] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[9] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling.* Prentice-Hall, Englewood Cliffs, NJ, 1973.

[10] Philippe Aigrain, Susan L. Graham, Robert R. Henry, Marshall Kirk McKusick, and Eduardo Pelegri-Llopart. Experience with a Graham-Glanville style code generator. *SIGPLAN Notices*, 19(6):13–24, June 1984. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.*

[11] Alexander Aiken and Alexandru Nicolau. Optimal loop parallelization. *SIGPLAN Notices*, 23(7):308–317, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[12] Frances E. Allen. Program optimization. In *Annual Review in Automatic Programming*, volume 5, pages 239–307. Pergamon Press, Oxford, England, 1969.

[13] Frances E. Allen. Control flow analysis. *SIGPLAN Notices*, 5(7):1–19, July 1970. *Proceedings of a Symposium on Compiler Optimization.*

[14] Frances E. Allen. The history of language processor technology in IBM. *IBM Journal of Research and Development*, 25(5):535–548, September 1981.

[15] Frances E. Allen and John Cocke. A catalogue of optimizing transformations. In R. Rustin, editor, *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, Englewood Cliffs, NJ, June 1972.

[16] Frances E. Allen and John Cocke. Graph-theoretic constructs for program flow analysis. Technical Report RC 3923 (17789), IBM Thomas J. Watson Research Center, Yorktown Heights, NY, July 1972.

[17] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communi-*

cations of the ACM, 19(3):137–147, March 1976.

[18] Frances E. Allen, John Cocke, and Ken Kennedy. Reduction of operator strength. In Steven S.Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 79–101. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[19] John R. Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, San Francisco, CA, October 2001.

[20] Bowen Alpern and Fred B. Schneider. Verifying temporal properties without temporal logic. *ACM Transactions on Programming Languages and Systems*, 11(1):147–167, January 1989.

[21] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, CA, January 1988.

[22] Stephen Alstrup, Dov Harel, Peter W. Lauridsen, and Mikkel Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–2132, June 1999.

[23] Marc A. Auslander and Martin E. Hopkins. An overview of the PL.8 compiler. *SIGPLAN Notices*, 17(6):22–31, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*

[24] Andrew Ayers, Robert Gottlieb, and Richard Schooler. Aggressive inlining. *SIGPLAN Notices*, 32(5):134–145, May 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

[25] John W. Backus. The history of Fortran I, II, and III. In Richard L. Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, New York, NY, 1981.

[26] John W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The FORTRAN automatic coding system. In *Proceedings of the Western Joint Computer Conference*, pages 188–198, Institute of Radio Engineers, New York, NY, February 1957.

[27] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[28] John T. Bagwell, Jr. Local optimizations. *SIGPLAN Notices*, 5(7):52–66, July 1970. *Proceedings of a Symposium on Compiler Optimization.*

[29] John Banning. An efficient way to find side effects of procedure calls and aliases of variables. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 29–41, San Antonio, TX, January 1979.

704

[30] William A. Barrett and John D. Couch. *Compiler Construction: Theory and Practice.* Science Research Associates, Inc., Chicago, IL, 1979.

[31] Jeffrey M. Barth. An interprocedural data flow analysis algorithm. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 119–131, Los Angeles, CA, January 1977.

[32] Alan M. Bauer and Harry J. Saal. Does *APL* really need run-time checking? *Software—Practice and Experience*, 4(2):129–138, 1974.

[33] Laszlo A. Belady. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[34] C. Gordon Bell and Allen Newell. *Computer Structures: Readings and Examples.* McGraw-Hill Book Company, New York, NY, 1971.

[35] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew T. O'Keefe. Spill code minimization via interference region spilling. *SIGPLAN Notices*, 32(5):287–295, May 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

[36] David Bernstein, Dina Q. Goldin, Martin Charles Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill code minimization techniques for optimizing compilers. *SIGPLAN Notices*, 24(7):258–263, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[37] David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. *SIGPLAN Notices*, 26(6):241–255, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[38] Robert L. Bernstein. Producing good code for the case statement. *Software—Practice and Experience*, 15(10):1021–1024, October 1985.

[39] Andrew Binstock and John Rex. *Practical Algorithms for Programmers.* Addison-Wesley, Reading, MA, 1995.

[40] Peter L. Bird. An implementation of a code generator specification language for table driven code generators. *SIGPLAN Notices*, 17(6):44–55, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*

[41] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant expressions. *SIGPLAN Notices*, 33(5):1–14, May 1998. *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation.*

[42] Hans-Juergen Boehm. Space efficient conservative garbage collection. *SIGPLAN Notices*, 28(6):197–206, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[43] Hans-Juergen Boehm and Alan Demers. Implementing Russell. *SIGPLAN Notices*, 21(7):186–195, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

[44] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, September 1988.

[45] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating register allocation and instruction scheduling for RISCS. *SIGPLAN Notices*, 26(4):122–131, April 1991. *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Systems.*

[46] Preston Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, Department of Computer Science, Houston, TX, April 1992. (Technical Report TR92–183, Computer Science Department, Rice University, 1992.)

[47] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software—Practice and Experience*, 28(8):859–881, July 1998.

[48] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. *SIGPLAN Notices*, 24(7):275–284, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[49] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Digital computer register allocation and code spilling using interference graph coloring. United States Patent 5,249,295, March 1993.

[50] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Software—Practice and Experience*, 27(6):701–724, June 1997.

[51] Preston Briggs, Keith D. Cooper, and Linda Torczon. Coloring register pairs. *ACM Letters on Programming Languages and Systems*, 1(1):3–13, March 1992.

[52] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. *SIGPLAN Notices*, 27(7):311–321, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[53] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[54] Preston Briggs and Linda Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1–4):59–69, March–December 1993.

[55] Adam L. Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery R. Westbrook. Linear-time pointer-machine algorithms for least common ancestors, *MST* verification, and dominators. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Com-*

705

puting, pages 279–288, Dallas, TX, 1998.

[56] Michael Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[57] Michael Burke and Linda Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, 15(3):367–399, July 1993.

[58] Jiazhen Cai and Robert Paige. Using multiset discrimination to solve language processing problems without hashing. *Theoretical Computer Science*, 145(1–2):189–228, 1995.

[59] Brad Calder and Dirk Grunwald. Reducing branch costs via branch alignment. *SIGPLAN Notices*, 29(11): 242–251, November 1994. *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems.*

[60] David Callahan, Alan Carle, Mary W. Hall, and Ken Kennedy. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, 16(4):483–487, April 1990.

706

[61] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. *SIGPLAN Notices*, 25(6):53–65, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[62] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. *SIGPLAN Notices*, 21(7):152–161, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

[63] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. *SIGPLAN Notices*, 26(6):192–203, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[64] Luca Cardelli. Type systems. In Allen B. Tucker, Jr., editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, Boca Raton, FL, December 1996.

[65] Steve Carr and Ken Kennedy. Scalar replacement in the presence of conditional control flow. *Software—Practice and Experience*, 24(1):51–77, 1994.

[66] Roderic G. G. Cattell. Automatic derivation of code generators from machine descriptions. *ACM Transactions on Programming Languages and Systems*, 2(2):173–190, April 1980.

[67] Roderic G. G. Cattell, Joseph M. Newcomer, and Bruce W. Leverett. Code generation in a machine-independent compiler. *SIGPLAN Notices*, 14(8):65–75, August 1979. *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction.*

[68] Gregory J. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices*, 17(6):98–105, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*

[69] Gregory J. Chaitin. Register allocation and spilling via graph coloring. United States Patent 4,571,678, February 1986.

[70] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, January 1981.

[71] David R. Chase. An improvement to bottom-up tree pattern matching. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 168–177, Munich, Germany, January 1987.

[72] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. *SIGPLAN Notices*, 25(6):296–310, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[73] J. Bradley Chen and Bradley D. D. Leupen. Improving instruction locality with just-in-time code layout. In *Proceedings of the First USENIX Windows NT Workshop*, pages 25–32, Seattle, WA, August 1997.

[74] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[75] Jong-Deok Choi, Michael Burke, and Paul R.Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, Charleston, SC, January 1993.

[76] Frederick C. Chow. *A Portable Machine-Independent Global Optimizer —Design and Measurements*. PhD thesis, Department of Electrical Engineering, Stanford University, Stanford, CA, December 1983. (Technical Report CSL-TR-83-254, Computer Systems Laboratory, Stanford University, December 1983.)

[77] Frederick C. Chow and John L. Hennessy. Register allocation by priority-based coloring. *SIGPLAN Notices*, 19(6):222–232, June 1984. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.*

[78] Frederick C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. ACM *Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[79] Cliff Click. *Combining Analyses, Combining Optimizations*. PhD thesis, Rice University, Department of Computer Science, Houston, TX, February 1995. (Technical Report TR95-252, Computer Science Department, Rice University, 1995.)

[80] Cliff Click. Global code motion/global value numbering. *SIGPLAN Notices*, 30(6):246–257, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.*

[81] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, 1995.

[82] John Cocke. Global common subexpression elimination. *SIGPLAN Notices*, 5(7):20–24, July 1970. *Proceedings of a Symposium on Compiler Construction.*

[83] John Cocke and Ken Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11):850–856, November 1977.

[84] John Cocke and Peter W. Markstein. Measurement of program improvement algorithms. In Simon H. Lavington, editor, *Information Processing 80*, North Holland, Amsterdam, Netherlands, pages 221–228, 1980, *Proceedings of IFIP Congress 80.*

[85] John Cocke and Peter W. Markstein. Strength reduction for division and modulo with application to accessing a multilevel store. *IBM Journal of Research and Development*, 24(6):692–694, 1980.

[86] John Cocke and Jacob T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, New York, NY, 1970.

[87] Jacques Cohen. Garbage collection of linked structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.

[88] Robert Cohn and P. Geoffrey Lowney. Hot cold optimization of large Windows/*NT* applications. In *Proceedings of the Twenty-Ninth Annual International Symposium on Microarchitecture*, pages 80–89, Paris, France, December 1996.

[89] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. *SIGPLAN Notices*, 30(6):279–290, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.*

[90] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[91] Melvin E. Conway. Design of a separable transition diagram compiler. *Communications*

707

708 *of the ACM*, 6(7):396–408, July 1963.

[92] Richard W. Conway and Thomas R. Wilcox. Design and implementation of a diagnostic compiler for PL/I. *Communications of the ACM*, 16(3):169–179, March 1973.

[93] Keith D. Cooper, Mary W. Hall, and Linda Torczon. An experiment with inline substitution. *Software—Practice and Experience*, 21(6):581–601, June 1991.

[94] Keith D. Cooper, Timothy J. Harvey, and Linda Torczon. How to build an interference graph. *Software—Practice and Experience*, 28(4):425–444, April 1998.

[95] Keith D. Cooper, Timothy J. Harvey, and Todd Waterman. Building a control-flow graph from scheduled assembly code. Technical Report 02-399, Department of Computer Science, Rice University, Houston, TX, June 2002.

[96] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. *SIGPLAN Notices*, 23(7):57–66, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[97] Keith D. Cooper and Ken Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 49–59, Austin, TX, January 1989.

[98] Keith D. Cooper and Philip J. Schielke. Non-local instruction scheduling with limited code growth. In *Proceedings of the 1998 ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems* (LCTES). *Lecture Notes in Computer Science 1474*, F. Mueller and A. Bestavros, editors, pages 193–207, Springer-Verlag, Heidelberg, Germany, 1998.

[99] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. *SIGPLAN Notices*, 34(7):1–9, July 1999. *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems* (LCTES), May 1999.

[100] Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In *Proceedings of the Seventh International Compiler Construction Conference, CC '98. Lecture Notes in Computer Science 1383*, pages 174–187, Springer-Verlag, Heidelberg, Germany, 1998.

[101] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. Operator strength reduction. *ACM Transactions on Programming Languages and Systems*, 23(5):603–625, September 2001.

[102] Keith D. Cooper and Todd Waterman. Understanding energy consumption on the C62x. In *Proceedings of the 2002 Workshop on Compilers and Operating Systems for Low Power*, pages 4–1 – 4–8, Charlottesville, VA, September 2002.

[103] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1992.

[104] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[105] Ron Cytron, Andy Lowry, and F. Kenneth Zadeck. Code motion of control structures in high-level languages. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 70–85, St. Petersburg Beach, FL,
709 January 1986.

[106] Manuvir Das. Unification-based pointer analysis with directional assignments. *SIGPLAN Notices*, 35(5):35–46, May 2000. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation.*

[107] Jack W. Davidson and Christopher W. Fraser. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2):191–202, April 1980.

[108] Jack W. Davidson and Christopher W. Fraser. Automatic generation of peephole opti-

mizations. *SIGPLAN Notices*, 19(6):111–116, June 1984. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.*

[109] Jack W. Davidson and Christopher W. Fraser. Register allocation and exhaustive peephole optimization. *Software—Practice and Experience*, 14(9):857–865, September 1984.

[110] Jack W. Davidson and Christopher W. Fraser. Automatic inference and fast interpretation of peephole optimization rules. *Software—Practice and Experience*, 17(11):801–812, November 1987.

[111] Jack W. Davidson and Ann M. Holler. A study of a C function inliner. *Software—Practice and Experience*, 18(8):775–790, August 1988.

[112] Alan J. Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 261–269, San Francisco, CA, January 1990.

[113] Frank DeRemer. Simple LR(k) grammars. *Communications of the* ACM, 14(7):453–460, July 1971.

[114] Frank DeRemer and Thomas J. Pennello. Efficient computation of LALR (1) look-ahead sets. *SIGPLAN Notices*, 14(8):176–187, August 1979. *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction.*

[115] Alain Deutsch. Interprocedural May-Alias analysis for pointers: Beyond k-limiting. *SIGPLAN Notices*, 29(6):230–241, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[116] L. Peter Deutsch. *An Interactive Program Verifier*. PhD thesis, Computer Science Department, University of California, Berkeley, Berkeley, CA, 1973. (Technical Report CSL-73-1, Xerox Palo Alto Research, May 1973.)

[117] L. Peter Deutsch and Daniel G Bobrow. An efficient, incremental, automatic, garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[118] Dhananjay M. Dhamdhere. On algorithms for operator strength reduction. *Communications of the ACM*, 22(5):311–312, May 1979.

[119] Dhananjay M. Dhamdhere. A fast algorithm for code movement optimisation. *SIGPLAN Notices*, 23(10):172–180, 1988.

[120] Dhananjay M. Dhamdhere. A new algorithm for composite hoisting and strength reduction. *International Journal of Computer Mathematics*, 27(1):1–14, 1989.

[121] Dhananjay M. Dhamdhere. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, April 1991. 710

[122] Michael K. Donegan, Robert E. Noonan, and Stefan Feyock. A code generator generator language. *SIGPLAN Notices*, 14(8):58–64, August 1979. *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction.*

[123] Jack J. Dongarra, James R. Bunch, Cleve B. Moler, and G. W. Stewart. *LINPACK User's Guide. SIAM*, Philadelphia, PA, 1979.

[124] Karl-Heinz Drechsler and Manfred P. Stadel. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies." *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, October 1988.

[125] Karl-Heinz Drechsler and Manfred P. Stadel. A variation of Knoop, Rüthing, and Steffen's "Lazy code motion." *SIGPLAN Notices*, 28(5):29–38, May 1993.

[126] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[127] Kemal Ebcioğlu and Toshio Nakatani. A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture. *Selected Papers of the Second Workshop on Languages and Compilers for Parallel Computing*, Pitman Publishing,

London, UK, pages 213–229, 1990.

[128] John R. Ellis. *Bulldog: A Compiler for* VLIW *Architectures*. The MIT Press, Cambridge, MA, 1986.

[129] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *SIGPLAN Notices*, 29(6):242–256, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[130] Jens Ernst, William S. Evans, Christopher W. Fraser, Steven Lucco, and Todd A. Proebsting. Code compression. *SIGPLAN Notices*, 32(5):358–365, May 1997. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

[131] Andrei P. Ershov. On programming of arithmetic expressions. *Communications of the ACM*, 1(8):3–6, August 1958. (The figures appear in volume 1, number 9, page 16.)

[132] Andrei P. Ershov. Reduction of the problem of memory allocation in programming to the problem of coloring the vertices of graphs. *Soviet Mathematics*, 3:163–165, 1962. Originally published in *Doklady Akademii Nauk S.S.S.R.*, 142(4), 1962.

[133] Andrei P. Ershov. Alpha—An automatic programming system of high efficiency. *Journal of the ACM*, 13(1):17–24, January 1966.

[134] Janet Fabri. Automatic storage optimization. *SIGPLAN Notices*, 14(8):83–91, August 1979. *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction.*

[135] Rodney Farrow. Linguist-86: Yet another translator writing system based on attribute grammars. *SIGPLAN Notices*, 17(6):160–171, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*

[136] Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. *SIGPLAN Notices*, 21(7):85–98, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

[137] Robert R. Fenichel and Jerome C. Yochelson. A *LISP* garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

711

[138] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[139] Charles N. Fischer and Richard J. LeBlanc, Jr. The implementation of run-time diagnostics in Pascal. *IEEE Transactions on Software Engineering*, SE-6(4):313–319, 1980.

[140] Charles N. Fischer and Richard J. LeBlanc, Jr. *Crafting a Compiler with C.* Benjamin/Cummings, Redwood City, CA, 1991.

[141] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[142] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel processing: A smart compiler and a dumb machine. *SIGPLAN Notices*, 19(6):37–47, June 1984. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.*

[143] Robert W. Floyd. An algorithm for coding efficient arithmetic expressions. *Communications of the ACM*, 4(1):42–51, January 1961.

[144] J. M. Foster. A syntax improving program. *Computer Journal*, 11(1): 31–34, May 1968.

[145] Christopher W. Fraser. Automatic inference of models for statistical code compression. *SIGPLAN Notices*, 34(5):242–246, May 1999. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation.*

[146] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[147] Christopher W. Fraser and Robert R. Henry. Hard-coding bottom-up code generation tables to save time and space. *Software—Practice and Experience*, 21(1):1–12, January

1991.

[148] Christopher W. Fraser, Eugene W. Myers, and Alan L. Wendt. Analyzing and compressing assembly code. *SIGPLAN Notices*, 19(6):117–121, June 1984. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.*

[149] Christopher W. Fraser and Alan L. Wendt. Integrating code generation and optimization. *SIGPLAN Notices*, 21(7):242–248, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

[150] Christopher W. Fraser and Alan L. Wendt. Automatic generation of fast optimizing code generators. *SIGPLAN Notices*, 23(7):79–84, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[151] Mahadevan Ganapathi and Charles N. Fischer. Description-driven code generation using attribute grammars. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 108–119, Albuquerque, NM, January 1982.

[152] Harald Ganzinger, Robert Giegerich, Ulrich Möncke, and Reinhard Wilhelm. A truly generative semantics-directed compiler generator. *SIGPLAN Notices*, 17(6):172–184, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*

[153] Lal George and Andrew W. Appel. Iterated register coalescing. In *Conference Record of the Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 208–218, St. Petersburg Beach, FL, January 1996. 712

[154] Phillip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *SIGPLAN Notices*, 21(7):11–16, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

[155] R. Steven Glanville and Susan L. Graham. A new method for compiler code generation. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 231–240, Tucson, AZ, January 1978.

[156] Nikolas Gloy and Michael D. Smith. Procedure placement using temporal-ordering information. *ACM Transactions on Programming Languages and Systems*, 21(5):997–1027, September 1999.

[157] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, MA, 1983.

[158] James R. Goodman and Wei-Chung Hsu. Code scheduling and register allocation in large basic blocks. *Proceedings of the Second International Conference on Supercomputing*, pages 442–452, July 1988.

[159] Eiichi Goto. Monocopy and associative operations in extended Lisp. Technical Report 74-03, University of Tokyo, Tokyo, Japan, May 1974.

[160] Susan L. Graham. Table-driven code generation. *IEEE Computer*, 13(8):25–34, August 1980.

[161] Susan L. Graham, Michael A. Harrison, and Walter L. Ruzzo. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2(3):415–462, July 1980.

[162] Susan L. Graham, Robert R. Henry, and Robert A. Schulman. An experiment in table driven code generation. *SIGPLAN Notices*, 17(6):32–43, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*

[163] Susan L. Graham and Mark Wegman. A fast and usually linear algorithm for global flow analysis. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages*, pages 22–34, Palo Alto, CA, January 1975.

[164] Susan L. Graham and Mark Wegman. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM*, 23(1):172–202, 1976.

[165] Torbjörn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the *GNU* C compiler. *SIGPLAN Notices*, 27(7):341–352, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[166] David Gries. *Compiler Construction for Digital Computers*. John Wiley and Sons, New York, NY, 1971.

[167] Rajiv Gupta and Mary Lou Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, April 1990.

[168] Rajiv Gupta, Mary Lou Soffa, and Tim Steele. Register allocation via clique separators. *SIGPLAN Notices*, 24(7):264–274, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[169] Max Hailperin. Cost-optimal code motion. *ACM Transactions on Programming Languages and Systems*, 20(6):1297–1322, November 1998.

[170] Mary W. Hall and Ken Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, September 1992.

[171] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software—Practice and Experience*, 20(1):5–12, January 1990.

[172] Dov Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing* STOC, pages 185–194, May 1985.

[173] William H. Harrison. A class of register allocation algorithms. Technical Report RC-5342, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1975.

[174] William H. Harrison. A new strategy for code generation—The general purpose optimizing compiler. *IEEE Transactions on Software Engineering*, SE-5(4):367–373, July 1979.

[175] Philip J. Hatcher and Thomas W. Christopher. High-quality code generation via bottom-up tree pattern matching. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 119–130, St. Petersburg Beach, FL, January 1986.

[176] Matthew S. Hecht and Jeffrey D. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM*, 21(3):367–375, July 1974.

[177] Matthew S. Hecht and Jeffrey D. Ullman. A simple algorithm for global data flow analysis problems. *SIAM Journal on Computing*, 4(4):519–532, 1975.

[178] J. Heller. Sequencing aspects of multiprogramming. *Journal of the ACM*, 8(3):426–439, July 1961.

[179] John L. Hennessy and Thomas Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[180] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, July 1999.

[181] Michael Hind and Anthony Pioli. Which pointer analysis should I use? *ACM SIGSOFT Software Engineering Notes*, 25(5):113–123, September 2000. In *Proceedings of the International Symposium on Software Testing and Analysis.*

[182] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1982.

[183] John E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz, editors, *Theory of Machines and Computations: Proceedings*, pages 189–196, Academic Press, New York, NY, 1971.

[184] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 1979.

[185] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, Inc., Potomac, MD, 1978.

[186] Lawrence P. Horwitz, Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. Index

register allocation. *Journal of the ACM*, 13(1):43–61, January 1966.

[187] Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence analysis for pointer variables. *SIGPLAN Notices*, 24(7):28–40, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.* 714

[188] Susan Horwitz and Tim Teitelbaum. Generating editing environments based on relations and attributes. *ACM Transactions on Programming Languages and Systems*, 8(4):577–608, October 1986.

[189] Brett L. Huber. Path-selection heuristics for dominator-path scheduling. Master's thesis, Computer Science Department, Michigan Technological University, Houghton, MI, October 1995.

[190] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing—Special Issue on Instruction Level Parallelism*, 7(1–2): 229–248, Kluwer Academic Publishers, Hingham, MA, May 1993.

[191] Edgar T. Irons. A syntax directed compiler for Algol 60. *Communications of the ACM*, 4(1):51–55, January 1961.

[192] J. R. Issaac and Dhananjay M. Dhamdhere. A composite algorithm for strength reduction and code movement optimization. *International Journal of Computer and Information Sciences*, 9(3):243–273, June 1980.

[193] Mehdi Jazayeri and Kenneth G. Walter. Alternating semantic evaluator. In *Proceedings of the 1975 Annual Conference of the ACM*, pages 230–234, 1975.

[194] Mark Scott Johnson and Terrence C. Miller. Effectiveness of a machine-level, global optimizer. *SIGPLAN Notices*, 21(7):99–108, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

[195] Stephen C. Johnson. Yacc—Yet another compiler-compiler. Technical Report 32 (Computing Science), AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[196] Stephen C. Johnson. A tour through the portable C compiler. In *Unix Programmer's Manual, 7th Edition*, volume 2b. AT&T Bell Laboratories, Murray Hill, NJ, January 1979.

[197] Walter L. Johnson, James H. Porter, Stephanie I. Ackley, and Douglas T. Ross. Automatic generation of efficient lexical processors using finite state techniques. *Communications of the ACM*, 11(12):805–813, December 1968.

[198] S. M. Joshi and Dhananjay M. Dhamdhere. A composite hoisting-strength reduction transformation for global program optimization. *International Journal of Computer Mathematics*, 11(1):21–44 (part I); 11(2): 111–126 (part II), 1982.

[199] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

[200] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[201] Tadao Kasami. An efficient recognition and syntax analysis algorithm for context-free languages. Scientific Report AFRCL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA, 1965.

[202] Ken Kennedy. *Global Flow Analysis and Register Allocation for Simple Code Structures.* PhD thesis, Courant Institute, New York University, October 1971.

[203] Ken Kennedy. Global dead computation elimination. SETL Newsletter 111, Courant Institute of Mathematical Sciences, New York University, New York, NY, August 1973. 715

[204] Ken Kennedy. Reduction in strength using hashed temporaries. SETL Newsletter 102, Courant Institute of Mathematical Sciences, New York University, New York, NY, March 1973.

[205] Ken Kennedy. Node listings applied to data flow analysis. In *Conference Record of the*

Second ACM Symposium on Principles of Programming Languages, pages 10–21, Palo Alto, CA, January 1975.

[206] Ken Kennedy. Use-definition chains with applications. *Computer Languages*, 3(3):163–179, 1978.

[207] Ken Kennedy. A survey of data flow analysis techniques. In Neil D. Jones and Steven S. Muchnick, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[208] Ken Kennedy and Linda Zucconi. Applications of graph grammar for program control flow analysis. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 72–85, Los Angeles, CA, January 1977.

[209] Robert Kennedy, Fred C. Chow, Peter Dahl, Shin-Ming Liu, Raymond Lo, and Mark Streich. Strength reduction via SSAPRE. In *Proceedings of the Seventh International Conference on Compiler Construction. Lecture Notes in Computer Science 1383*, pages 144–158, Springer-Verlag, Heidelberg, Germany, March 1998.

[210] Daniel R. Kerns and Susan J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. *SIGPLAN Notices*, 28(6):278–289, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[211] Robert R. Kessler. Peep—An architectural description driven peephole optimizer. *SIGPLAN Notices*, 19(6):106–110, June 1984. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.*

[212] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, Boston, MA, October 1973.

[213] Stephen C. Kleene. Representation of events in nerve nets and finite automata. In Claude E. Shannon and John McCarthy, editors, *Automata Studies. Annals of Mathematics Studies*, 34:3–41. Princeton University Press, Princeton, NJ, 1956.

[214] Kath Knobe and Andrew Meltzer. Control tree based register allocation. Technical report, COMPASS, 1990.

[215] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. *SIGPLAN Notices*, 27(7):224–234, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[216] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy strength reduction. *International Journal of Programming Languages*, 1(1):71–91, March 1993.

[217] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, December 1965.

[218] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[219] Donald E. Knuth. Semantics of context-free languages: Correction. *Mathematical Systems Theory*, 5(1):95–96, 1971.

716

[220] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.

[221] Donald E. Knuth. A history of writing compilers. *Computers and Automation*, 11(12): 8–18, December 1962. Reprinted in *Compiler Techniques*, Bary W. Pollack, editor, pages 38–56, Auerbach, Princeton, NJ, 1972.

[222] Dexter C. Kozen. *Automata and Computability*. Springer-Verlag, New York, NY, 1997.

[223] Glenn Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, Reading, MA, August 1983.

[224] Sanjay M. Krishnamurthy. A brief survey of papers on scheduling for pipelined processors. *SIGPLAN Notices*, 25(7):97–106, July 1990.

[225] Steven M. Kurlander and Charles N. Fischer. Zero-cost range splitting. *SIGPLAN Notices*, 29(6):257–265, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Program-*

ming Language Design and Implementation.

[226] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. *SIGPLAN Notices*, 23(7):318–328, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[227] David Alex Lamb. Construction of a peephole optimizer. *Software—Practice and Experience*, 11(6):639–647, June 1981.

[228] William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem taxonomy. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 93–103, Orlando, FL, January 1991.

[229] David Landskov, Scott Davidson, Bruce Shriver, and Patrick W. Mallett. Local microcode compaction techniques. *ACM Computing Surveys*, 12(3):261–294, September 1980.

[230] Rudolf Landwehr, Hans-Stephan Jansohn, and Gerhard Goos. Experience with an automatic code generator generator. *SIGPLAN Notices*, 17(6):56–66, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*

[231] James R. Larus and Paul N. Hilfinger. Register allocation in the SPUR Lisp compiler. *SIGPLAN Notices*, 21(7):255–263, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

[232] S. S. Lavrov. Store economy in closed operator schemes. *Journal of Computational Mathematics and Mathematical Physics*, 1(4):687–701, 1961. English translation in *U.S.S.R. Computational Mathematics and Mathematical Physics* 3:810-828, 1962.

[233] Charles Lefurgy, Peter Bird, I-Cheng Chen, and Trevor Mudge. Improving code density using compression techniques. In *Proceedings of the Thirtieth International Symposium on Microarchitecture*, pages 194–203, IEEE Computer Society Press , Los Alamitos, CA, December 1997.

[234] Charles Lefurgy, Eva Piccininni, and Trevor Mudge. Reducing code size with run-time decompression. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 218–227, IEEE Computer Society Press, Los Alamitos, CA, January 2000.

[235] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[236] Philip M. Lewis and Richard E. Stearns. Syntax-directed transduction. *Journal of the ACM*, 15(3):465–488, July 1968.

[237] Vincenzo Liberatore, Martin Farach-Colton, and Ulrich Kremer. Evaluation of algorithms for local register allocation. In *Eighth International Conference on Compiler Construction (CC 99)*, LNCS. *Lecture Notes in Computer Science 1575*, pages 137–152, Springer-Verlag, Heidelberg, Germany, 1999.

[238] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[239] Barbara Liskov, Russell R. Atkinson, Toby Bloom, J. Eliot B. Moss, Craig Schaflert, Robert Scheifler, and Alan Snyder CLU *Reference Manual. Lecture Notes in Computer Science 114*, Springer-Verlag, Heidelberg, Germany 1981.

[240] Jack L. Lo and Susan J. Eggers. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. *SIGPLAN Notices*, 30(6):151–162, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.*

[241] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. Register promotion by sparse partial redundancy elimination of loads and stores. *SIGPLAN Notices*, 33(5):26–37, May 1998. *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation.*

[242] P. Geoffrey Lowney, Stefan M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John. C. Ruttenberg. The Multiflow trace scheduling com-

717

piler. *The Journal of Supercomputing—Special Issue*, 7(1–2):51–142, March 1993.

[243] Edward S. Lowry and C. W. Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, January 1969.

[244] John Lu and Keith D. Cooper. Register promotion in C programs. *SIGPLAN Notices*, 32(5):308–319, May 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

[245] John Lu and Rob Shillner. Clean: Removing useless control flow. Unpublished. Department of Computer Science, Rice University, Houston, TX, June 1994.

[246] Peter Lucas. Die strukturanalyse von formelübersetzern. *Elektronische Rechenanlagen*, 3:159–167, 1961.

[247] Guei-Yuan Lueh, Thomas Gross, and Ali-Reza Adl-Tabatabai. Global register allocation based on graph fusion. In *Proceedings of the Ninth International Workshop on Languages and Compilers for Parallel Computing (LCPC '96). Lecture Notes in Computer Science 1239*, pages 246–265, Springer-Verlag, Heidelberg, Germany, 1997.

[248] Bohdan S. Majewski, Nicholas C. Wormald, George Havas, and Zbigniew J. Czech. A family of perfect hashing methods. *The Computer Journal*, 39(6):547–554, 1996.

[249] Brian L. Marks. Compilation to compact code. *IBM Journal of Research and Development*, 24(6):684–691, November 1980.

[250] Peter W. Markstein, Victoria Markstein, and F. Kenneth Zadeck. Reassociation and strength reduction. Unpublished book chapter, July 1994.

[251] Henry Massalin. Superoptimizer—A look at the smallest program. *SIGPLAN Notices*, 22(10):122–126, October 1987. *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems.*

718

[252] John McCarthy. Lisp—Notes on its past and future. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, pages v–viii, Stanford University, Stanford, CA, 1980.

[253] William M. McKeeman. Peephole optimization. *Communications of the ACM*, 8(7):443–444, July 1965.

[254] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):424–453, July 1996.

[255] Kathryn S. McKinley and Olivier Temam. A quantitative analysis of loop nest locality. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-7), pages 94–104, Cambridge, MA, September 1996.

[256] Robert McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, EC-9(1):39–47, March 1960.

[257] Terrence C. Miller. *Tentative Compilation: A design for an APL compiler.* PhD thesis, Yale University, New Haven, CT, May 1978. See also paper of same title in *Proceedings of the International Conference on APL: Part 1*, pages 88–95, New York, NY, 1979.

[258] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML—Revised. MIT* Press, Cambridge, MA, 1997.

[259] Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[260] Robert Morgan. *Building an Optimizing Compiler.* Digital Press (an imprint of Butterworth–Heineman), Boston, MA, February 1998.

[261] Rajeev Motwani, Krishna V. Palem, Vivek Sarkar, and Salem Reyen. Combining register allocation and instruction scheduling. Technical Report 698, Courant Institute of Mathematical Sciences, New York University, New York, NY, July 1995.

[262] Steven S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann, San Francisco, CA, August 1997.

[263] Frank Mueller and David B. Whalley. Avoiding unconditional jumps by code replication. *SIGPLAN Notices*, 27(7):322–330, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[264] Thomas P. Murtagh. An improved storage management scheme for block structured languages. *ACM Transactions on Programming Languages and Systems*, 13(3):372–398, July 1991.

[265] Peter Naur (editor), J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1):1–17, January 1963.

[266] Brian R. Nickerson. Graph coloring register allocation for processors with multi-register operands. *SIGPLAN Notices*, 25(6):40–52, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[267] Cindy Norris and Lori L. Pollock. A scheduler-sensitive global register allocator. In *Proceedings of Supercomputing '93*, pages 804–813, Portland, OR, November 1993. 719

[268] Cindy Norris and Lori L. Pollock. An experimental study of several cooperative register allocation and instruction scheduling strategies. In *Proceedings of the Twenty-Eighth Annual International Symposium on Microarchitecture*, pages 169–179, IEEE Computer Society Press, Los Alamitos, CA, December 1995.

[269] Kristen Nygaard and Ole-Johan Dahl. The development of the *simula* languages. In *Proceedings of the First ACM SIGPLAN Conference on the History of Programming Languages*, pages 245–272, ACM Press, New York, NY, January 1978.

[270] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. In *Proceedings of the 1998 International Conference on Parallel Architecture and Compilation Techniques* (PACT), pages 196–204, October 1998.

[271] Eduardo Pelegrí-Llopart and Susan L. Graham. Optimal code generation for expression trees: An application of BURS theory. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 294–308, San Diego, CA, January 1988.

[272] Thomas J. Pennello. Very fast LR parsing. *SIGPLAN Notices*, 21(7):145–151, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

[273] Karl Pettis and Robert C. Hansen. Profile guided code positioning. *SIGPLAN Notices*, 25(6):16–27, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[274] Shlomit S. Pinter. Register allocation with instruction scheduling: A new approach. *SIGPLAN Notices*, 28(6):248–257, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[275] Gordon D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[276] Todd A. Proebsting. Simple and efficient BURS table generation. *SIGPLAN Notices*, 27(7):331–340, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[277] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 322–332, San Francisco, CA, January 1995.

[278] Todd A. Proebsting and Charles N. Fischer. Linear-time, optimal code scheduling for delayed-load architectures. *SIGPLAN Notices*, 26(6): 256–267, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[279] Todd A. Proebsting and Charles N. Fischer. Probabilistic register allocation. *SIGPLAN Notices*, 27(7):300–310, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[280] Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Proceedings of the Eastern Joint Computer Conference*, pages 133–138, Institute of Radio Engineers, New York, NY, December 1959.

[281] Paul W. Purdom, Jr. and Edward F. Moore. Immediate predominators in a directed graph[H]. *Communications of the ACM*, 15(8):777–778, August 1972.

[282] Brian Randell and L. J. Russell. *Algol 60 Programs Implementation; The Translation and Use of Algol 60 Programs on a Computer.* Academic Press, London, England, 1964.

[283] Bob R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the Fourteenth Annual Microprogramming Workshop on Microprogramming*, pages 183–198, December 1981.

[284] John H. Reif. Symbolic programming analysis in almost linear time. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 76–83, Tucson, AZ, January 1978.

[285] John H. Reif and Harry R. Lewis. Symbolic evaluation and the global value graph. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 104–118, Los Angeles, CA, January 1977.

[286] Thomas Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 169–176, Albuquerque, NM, January 1982.

[287] Thomas Reps and Bowen Alpern. Interactive proof checking. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 36–45, Salt Lake City, UT, January 1984.

[288] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Springer-Verlag, New York, NY, 1988.

[289] Martin Richards. The portability of the BCPL compiler. *Software—Practice and Experience*, 1(2):135–146, April–June 1971.

[290] Steve Richardson and Mahadevan Ganapathi. Interprocedural analysis versus procedure integration. *Information Processing Letters*, 32(3): 137–142, August 1989.

[291] Ronald Rivest. On self-organizing sequential search heuristics. *Communications of the ACM*, 19(2):63–67, February 1976.

[292] Anne Rogers and Kai Li. Software support for speculative loads. *SIGPLAN Notices*, 27(9):38–50, September 1992. *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.*

[293] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 12–27, San Diego, CA, January 1988.

[294] Daniel J. Rosenkrantz and Richard Edwin Stearns. Properties of deterministic top-down grammars. *Information and Control*, 17(3):226–256, October 1970.

[295] Barbara G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, May 1979.

[296] A. V. S. Sastry and Roy D. C. Ju. A new algorithm for scalar register promotion based on SSA form. *SIGPLAN Notices*, 33(5):15–25, May 1998. *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation.*

[297] Randolph G. Scarborough and Harwood G. Kolsky. Improved optimization of FORTRAN object programs. *IBM Journal of Research and Development*, 24(6):660–676, November 1980.

[298] Philip J. Schielke. *Stochastic Instruction Scheduling.* PhD thesis, Rice University, Department of Computer Science, Houston, TX, May 2000. (Technical Report TR00-370, Computer Science Department, Rice University, 2000.)

[299] Herb Schorr and William M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[300] Jacob T. Schwartz. On programming: An interim report on the SETL project. Installment II: The SETL language and examples of its use. Technical report, Courant Institute of Mathematical Sciences, New York University, New York, NY, October 1973.

[301] Ravi Sethi and Jeffrey D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, October 1970.

[302] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of the Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 1–14, Paris, France, January 1997.

[303] Robert M. Shapiro and Harry Saint. The representation of algorithms. Technical Report CA-7002-1432, Massachusetts Computer Associates, February 1970.

[304] Peter B. Sheridan. The arithmetic translator-compiler of the IBM FORTRAN automatic coding system. *Communications of the ACM*, 2(2):9–21, February 1959.

[305] Olin Shivers. Control flow analysis in Scheme. *SIGPLAN Notices*, 23(7):164–174, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[306] L. Taylor Simpson. *Value-Driven Redundancy Elimination.* PhD thesis, Rice University, Department of Computer Science, Houston, TX, 1996.(Technical Report TR 96–308, Computer Science Department, Rice University, 1996.)

[307] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing Co., Boston, MA, December 1996.

[308] Richard L. Sites and Daniel R. Perkins. Universal P-code definition, version 0.2. Technical Report 78-CS-C29, Department of Applied Physics and Information Sciences, University of California at San Diego, San Deigo, CA, January 1979.

[309] Daniel Dominic Sleator and Robert Endre Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2): 202–208, February 1985.

[310] Michael D. Smith, Mark Horowitz, and Monica S. Lam. Efficient superscalar performance through boosting. *SIGPLAN Notices*, 27(9): 248–259, September 1992. *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.*

[311] Mark Smotherman, Sanjay M. Krishnamurthy, P. S. Aravind, and David Hunnicutt. Efficient DAG construction and heuristic calculation for instruction scheduling. In *Proceedings of the Twenty-Fourth Annual Workshop on Microarchitecture* (MICRO-24), pages 93–102, Albuquerque, NM, August 1991.

[312] Arthur Sorkin. Some comments on "A solution to a problem with Morel and Renvoise's 'Global optimization by suppression of partial redundancies.' " *ACM Transactions on Programming Languages and Systems*, 11(4):666–668, October 1989.

[313] Thomas C. Spillman. Exposing side-effects in a PL/1 optimizing compiler. In *Information Processing 71*, pages 376–381. North-Holland, Amsterdam, Netherlands, 1972. *Proceedings of IFIP Congress 71.*

[314] Guy Lewis Steele, Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, MIT Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, May 1978.

[315] Philip H. Sweany and Steven J. Beaty. Post-compaction register assignment in a retargetable compiler. In *Proceedings of the Twenty-Third Annual Workshop and Symposium on Microprogramming and Microarchitecture* (MICRO-23), pages 107–116, Orlando, FL, November 1990.

[316] Philip H. Sweany and Steven J. Beaty. Dominator-path scheduling—A global scheduling method. *ACM SIGMICRO Newsletter*, 23(1–2): 260–263, December 1992. *Proceedings of the Twenty-Fifth Annual International Symposium on Microarchitecture.*

[317] Robert Endre Tarjan. Testing flow graph reducibility. *Journal of Computer and System*

722

Sciences, 9(3):355–365, December 1974.

[318] Robert Endre Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, July 1981.

[319] Robert Endre Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, July 1981.

[320] Robert Endre Tarjan and John H. Reif. Symbolic program analysis in almost-linear time. *SIAM Journal on Computing*, 11(1):81–93, February 1982.

[321] Ken Thompson. Programming Techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[322] Steven W. K. Tjiang. TWIG reference manual. Technical Report CSTR 120, Computing Sciences, AT&T Bell Laboratories, Murray Hill, NJ, January 1986.

[323] Jeffrey D. Ullman. Fast algorithms for the elimination of common subexpressions. *Acta Informatica*, 2(3):191–213, 1973.

[324] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGSOFT Software Engineering Notes*, 9(3): 157–167, May 1984. *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.*

[325] Victor Vyssotsky and Peter Wegner. A graph theoretical FORTRAN source language analyzer. Manuscript, AT&T Bell Laboratories, Murray Hill, NJ, 1963.

[326] William Waite and Gerhard Goos. *Compiler Construction.* Springer-Verlag, New York, NY, 1984.

[327] Scott Kipling Warren. *The Coroutine Model of Attribute Grammar Evaluation.* PhD thesis, Department of Mathematical Sciences, Rice University, Houston, TX, April 1976.

[328] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 291–299, New Orleans, LA, January 1985.

[329] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[330] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, Las Vegas, NV, January 1980.

[331] Clark Wiedmann. Steps toward an *APL* compiler. *ACM SIGAPL APL Quote Quad*, 9(4):321–328, June 1979. *Proceedings of the International Conference on APL.*

723

[332] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management. Lecture Notes in Computer Science 637*, pages 1–42, Springer-Verlag, Heidelberg, Germany, 1992.

[333] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. *SIGPLAN Notices*, 30(6):1–12, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.*

[334] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[335] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, Redwood City, CA, 1996.

[336] D. Wood. The theory of left-factored languages, part 1. *The Computer Journal*, 12(4):349–356, November 1969.

[337] D. Wood. The theory of left-factored languages, part 2. *The Computer Journal*, 13(1):55–62, February 1970.

[338] D. Wood. A further note on top-down deterministic languages. *The Computer Journal,*

14(4):396–403, November 1971.

[339] William Wulf, Richard K. Johnsson, Charles B. Weinstock, Steven O. Hobbs, and Charles M. Geschke. *The Design of an Optimizing Compiler*. Programming Languages Series. American Elsevier Publishing Company, Inc., New York, NY, 1975.

[340] Cliff Young, David S. Johnson, David R. Karger, and Michael D. Smith. Near-optimal intraprocedural branch alignment. *SIGPLAN Notices*, 32(5):183–193, May 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

[341] Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.

[342] F. Kenneth Zadeck. Incremental data flow analysis in a structured program editor. *SIGPLAN Notices*, 19(6):132–143, June 1984. *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.*

724

练　习

第1章

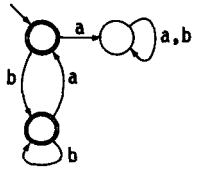
1. 考虑一个简单的Web浏览器，它取输入HTML格式的文本串并在屏幕上显示指定的图像。显示过程是编译或解释之一吗？
2. 在设计编译器时，你将面对很多权衡。作为一名用户，你考虑你购买的编译器中最重要的五个质量因素是什么？而当你是编译器设计者时，这一列表会变吗？就你将要实现的编译器，这一列表会告诉你什么？
3. 编译器被用于很多不同的环境。对于根据下面应用所设计的编译器，你期望的差异是什么？
   a. 用于翻译在网络上下载的用户界面代码的即时（just-in-time）编译器。
   b. 用于移动电话的嵌入式处理器的编译器。
   c. 用于高中程序设计导论性课程的编译器。
   d. 用于构建运行于（所有处理器都是相同的）巨型并行处理器的风洞模拟的编译器。
   e. 目标是运行于大量不同机器上的进行大量数值计算的程序的编译器。

第2章

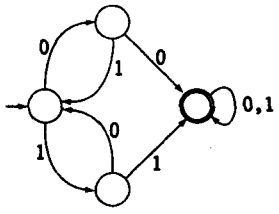
2.2节

1. 非形式地描述下面FA接受的语言：

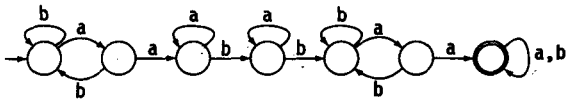
a)



b)



c)



2. 构造接受下面各语言的FA：
   a. $\{w \in \{a, b\}^* | w$以a开始且包含子串$baba\}$
   b. $\{w \in \{0, 1\}^* | w$包含子串111且不包含子串00$\}$

c. $\{w \in \{a, b, c\}^* | w$中$a$的数目模2与$b$的数目模3相等$\}$

3. 创建识别（a）表示复数的字，以及（b）表示以科学记数法格式表示的十进制数的字的FA。 726

2.3节

1. 不同的程序设计语言使用不同的方法表示整数。构建下面每一种整数的正则表达式:
   a. C语言中表示成以10和16为底的非负整数。
   b. VHDL中可以包含下划线的非负整数。（下划线不能出现在第一个字符或最后一个字符。）
   c. 在美元中，货币被表示四舍五入到最近的百分之一的正十进制数。这样的数开始于字符$，从小数点左边使用逗号把数字分成三位一组，并以小数点右边两位数字结束，例如，$8,937.43和$7,777,777.77。
2. 写出下列各语言的正则表达式。
   （提示: 并非所有描述刻画正则语言。）
   a. 给定字母表$\Sigma = \{0, 1\}$，L是交替0、1对串组成的集合。
   b. 给定字母表$\Sigma = \{0, 1\}$，L是包含偶数个0或偶数个1的0、1串组成的集合。
   c. 给定小写英语字母表，L是所有字母按字典顺序升序出现的串组成的集合。
   d. 给定字母表$\Sigma = \{a, b, c, d\}$，L是串$ryzwy$的集合，其中x和w是Σ中一个或多个字符组成的串，y是Σ中的任意单一字符，而z是就是字符z，它是从字母表外部取来的。
   （每一个串$xyzwy$都包含由字母表Σ中的字母构成的两个字xy和wy。这些字的结束字母都是y。它们被字母z分开。）
   e. 给定字母表$\Sigma = \{+, -, \times, \div, (,), \texttt{id}\}$，L是在id上使用加号、减号、乘号、除号和圆括号构成的代数表达式的集合。
3. 写出描述下列程序设计语言结构的正则表达式:
   a. 横向跳格和空格的任意序列（有时称为空白符（white space））。
   b. C程序设计语言中的注释。 727
   c. 串常量（不含转义字符）。
   d. 浮点数。

2.4节

1. 考虑下面三个正则表达式:

$$(ab \mid ac)^*$$
$$(0 \mid 1)^* 1\ 100\ 1^*$$
$$(01 \mid 10 \mid 00)^*\ 11$$

   a. 使用Thompson构造法构造上面每一个正则表达式对应的NFA。
   b. 把这些NFA转换成DFA。
   c. 最小化上面得到的DFA。
2. 证明两个RE等价的一个方法是构造它们对应的最小化DFA，然后比较它们。如果它们只是在状态名字上不同，那么这两个RE等价。使用这一技术检查下列的RE对，并说明它们是否等价。
   a. $(0 \mid 1)^*$ 和 $(0^* \mid 10^*)^*$
   b. $(ba)^+ (a^* b^* \mid a^*)$ 和 $(ba)^* ba^+ (b^* \mid \epsilon)$

3. 考虑下面的正则表达式

r0 | r00 | r1 | r01 | r2 | r02 | ··· | r09 | r10 | r11 | ··· | r30 | r31

运用结构法构建:

a. 对应于此RE的NFA。

b. 对应于上面NFA的DFA。

c. 对应于上面DFA的RE。

728 解释由源RE与（a）和（c）部分所产生的RE之间的差异。

比较你生成的DFA和第2章由下面RE构建的DFA。

r((0 | 1 | 2) ([0 ··· 9] | ε) | (4 | 5 | 6 | 7 | 8 | 9) | (3 (0 | 1 | ε)))

4. 最小化前一个练习所得的DFA。最小化后，使用更简单的寄存器描述在尺寸和速度方面的损失是什么?

5. 对于某些情况，可以把由ε转换连结的两个状态组合起来。

a. 在什么样的条件下，可以把两个由ε转换连接的状态组合起来?

b. 给出消除ε转换的算法。

c. 你的算法与用于实现子集构造法中使用的ε闭包的关系如何?

6. 证明正则语言的集合在交运算下封闭。

2.5节

1. 为下面C语言结构构造DFA，然后为它们中的每一个构建表驱动实现的相应表:

a. 整数常量。

b. 标识符。

c. 注释。

2. 对于上面练习中的每一个DFA，构建相应的直接编码扫描器。

3. 本章给出两种类型的DFA实现：一个是使用选择语句或开关语句的表驱动实现，另一个是使用分支和跳转的直接编码扫描器。第三个选择是使用交互递归函数来实现扫描器。讨论这样的实现的优点
729 和缺点。

第3章

3.2节

1. 为正则表达式的语法写出一个上下文相关文法。

2. 为上下文无关文法的Backus-Naur形式（BNF）写出一个上下文无关文法。

3. 在一次考试中，当被问及*非歧义性上下文无关文法*的定义时，两个学生给出了不同的答案。第一个学生把它定义为“一个文法，其中每一个句子都有由最左派生的惟一语法树。”而第二个学生把它定义为“一个文法，其中每一个句子都有根据任意派生的惟一语法树。”哪个是正确的?

3.3节

1. 下面的文法不适合于自顶向下预测分析器。找出其中的问题并通过重写文法修改它们。证明你的新文法满足LL(1)条件。

$$
\begin{array}{lcl}
L & \rightarrow & R\,\mathtt{a} \\
 & | & Q\,\mathtt{ba}
\end{array}
\qquad
\begin{array}{lcl}
R & \rightarrow & \mathtt{aba} \\
 & | & \mathtt{caba} \\
 & | & R\,\mathtt{bc}
\end{array}
\qquad
\begin{array}{lcl}
Q & \rightarrow & \mathtt{bbc} \\
 & | & \mathtt{bc}
\end{array}
$$

2. 考虑下面的文法:

$$
\begin{array}{lcl}
A & \rightarrow & B\,\mathtt{a} \\
B & \rightarrow & \mathtt{dab} \\
 & | & C\,\mathtt{b}
\end{array}
\qquad
\begin{array}{lcl}
C & \rightarrow & \mathtt{c}\,B \\
 & | & A\,\mathtt{c}
\end{array}
$$

这一文法满足LL(1)的条件吗？证明你的答案。如果它不满足LL(1)的条件，在不改变这一文法所描述的语言的前提下，修改这一文法使其成为LL(1)。 730

3. 可以在从左到右的线性扫描的过程中，通过向前看k个字符进行自顶向下分析的文法被称为LL（k）文法。在本书中，LL(1)的条件是用FIRST集合来描述的。你如何定义描述LL（k）的条件所需的FIRST集合？

4. 假设一个升降机是由两个命令控制的：↑把升降机向上移动一层，而↓把升降机向下移动一层。假设建筑物的高度是任意的，且这个升降机从第x层开始运行。

写出一个LL(1)文法，它生成满足下面条件的任何指令序列：① 从不引发升降机到第x层以下，② 在指令序列的结束时总是使升降机返回到第x层。例如，↑↑↓↓和↑↓↑↓是合法的指令序列，但是↑↓↓↑和↑↓↓不是合法的指令序列。为方便起见，你可以认为空序列是合法的。证明你的文法是LL(1)文法。

3.4节

1. 自底向上移入归约分析器的每一次移动称为一个*动作*（action）。这些动作包括移入、归约、接受和错误动作。当在分析中的给定点多个动作可能时，引发一个*分析冲突*（parsing conflict）。

考虑在分析器的各动作间可能出现的所有种类的冲突。哪些种类的冲突是可能的？哪些是不可能的？陈述每一个答案的正当性。

2. 自顶向下分析器和自底向上分析器以不同的顺序构建语法树。编写取一个语法树，并按构建的顺序打印结点的一对程序`TopDown`和`BottomUp`。`TopDown`应该显示自顶向下分析器的构建顺序，而`BottomUp`则应该给出自底向上分析器的构建顺序。

3.5节

1. ClockNoise语言（CN）是由下面的文法表示的: 731

$$
\begin{array}{lcl}
\mathit{Goal} & \rightarrow & \mathit{ClockNoise} \\
\mathit{ClockNoise} & \rightarrow & \mathit{ClockNoise}\ \mathtt{tick\ tock} \\
 & | & \mathtt{tick\ tock}
\end{array}
$$

a. CN的LR(1)项目是什么？

b. CN的FIRST集合是什么？

c. 构造CN的LR(1)项目集合的规范集合。

d. 得出ACTION和GOTO表。

2. 下面的文法描述匹配括号语言。

$$
\begin{array}{lcl}
\mathit{Goal} & \rightarrow & \mathit{Parens} \\
\mathit{Parens} & \rightarrow & (\ \mathit{Parens}\ )\ \mathit{Parens} \\
 & | & \epsilon
\end{array}
$$

a. 构造上面文法的LR(1)项目集合的规范集合。

b. 得出ACTION和GOTO表。

c. 这一文法是LR(1)文法吗?

3. 考虑下面的文法:

$$
\begin{array}{lcl}
Start & \rightarrow & S \\
S & \rightarrow & A\,\mathtt{a} \\
A & \rightarrow & B\,C \\
 & | & B\,C\,\mathtt{f} \\
B & \rightarrow & \mathtt{b} \\
C & \rightarrow & \mathtt{c}
\end{array}
$$

a. 构造上面的文法的LR(1)项目集合的规范集合。

b. 得出ACTION和GOTO表。

732 c. 这一文法是LR(1)文法吗?

4. 考虑接受两个命令的机械手: ▽把一个苹果放入包中, △从包中取出一个苹果。假设机械手是从空包开始的。

对于这一机械手的合法命令序列应该满足任意前缀中△命令的数目不超过▽命令的数目。作为例子, ▽▽△△和▽△▽都是合法指令序列, 而▽△△▽和▽△▽△△则不是。

a. 写出表示这一机械手的所有合法命令序列的LR(1)文法。

b. 证明这一文法是LR(1)文法。

3.6节

1. 为包括二元操作符(+和-)、一元减(-)、自增(++)和自减(--)且按惯例优先度运算的表达式写出一个文法。假设不允许出现重复的一元减, 但允许出现重复的自增和自减。

3.7节

1. 考虑程序设计语言方案构建分析器的任务。比较自顶向下归约递减分析器与表驱动LR(1)分析器所需的努力。(假设你已经有一个LR(1)表生成器。)
2. 本章描述了消除文法中无用产生式的一个手工操作技术。

a. 你能修改LR(1)表构造算法使其自动地消除无用产生式的负荷吗?

b. 即使一个产生式在语法上是无用的, 它也可能服务于特殊的目的。例如, 编译器设计者可以把一个语法制导动作与无用产生式联系起来(如第4章所示)。你的已修改算法应该如何处理与无用产生式

733 相关联的动作?

第4章

4.2节

1. 在Scheme中, +操作符是重载的。已知Scheme是一个动态类型语言, 描述对形如(+ab)的操作做类型检查的方法, 其中a和b可以是对+合法的任意类型。
2. 某些语言, 例如APL或RHP, 既不需要变量声明也不迫使对同一变量的赋值间的一致性。(一个程序可以给x赋一个整数10, 而后在相同的作用域内再给x赋串值"book"。)这种程序设计风格有时称为*类型欺骗*(type juggling)。

假设你有某一语言的已存在实现，这一实现没有声明但需要类型一致的使用。你将如何修改它使其允许类型欺骗呢？

4.3节

1. 基于下列评估规则，画出一棵注释分析树以明示a－(b＋c)的语法树是如何构建的。

产 生 式			评 估 规 则
E_0	→	E_1 + T	{ E_0.nptr ← mknode(+,E_1.nptr,T.nptr) }
E_0	→	E_1 - T	{ E_0.nptr ← mknode(-,E_1.nptr,T.nptr) }
E_0	→	T	{ E_0.nptr ← T.nptr }
T	→	(E)	{ T.nptr ← E.nptr }
T	→	id	{ T.nptr ← mkleaf(id,id.entry) }

2. 使用属性文法范例写出一个经典表达式文法的解释器。假设每一个ident有一个value属性和一个name属性。假设所有属性已被定义，且所有值将总有相同类型。 734
3. 写出一个文法来描述所有4的倍数的二进制数。把属性规则加到这一文法中，它用该二进制数的十进制值为value属性注释语法树的开始符号。
4. 使用前面练习所定义的文法，构建二进制数11100的语法树。
   a. 使用相应的值给出此树中所有的属性。
   b. 画出这一语法树的属性相关图并将所有属性分类综合或继承属性。

4.4节

1. 在Pascal中，程序员可以使用下面的语法来声明两个整型变量a和b。

```
var a, b: int
```

这一声明可以使用下面的文法来描述。

VarDecl → var *IDList* : *TypeID*
IDList → *IDList*, *ID*
　　| *ID*

其中*IDList*派生一个用逗号分隔的变量名列表，而*TypeID*派生一个合法的Pascal类型。你可能发现有必要重写这一文法。

a. 写出一个属性文法，使其给每一个声明变量赋一个正确的数据类型。
b. 写出一个专用语法制导翻译方案，使其给每一个声明变量赋一个正确的数据类型。
c. 这两个方案可以通过在语法树上的单一遍来操作吗？

2. 有时，编译器设计者可以跨越上下文无关和上下文相关分析间的边界移动一个问题。例如，考虑FORTRAN 77中函数调用和数组引用间发生的典型歧义性问题。使用下列产生式可以将这些构造加入到经典表达式文法中。 735

Factor → ident (*ExprList*)
ExprList → *ExprList* , *Expr*
　　| *Expr*

在这里，函数调用和数组引用之间的惟一差别在于如何声明ident。

在前一章，我们讨论了使用扫描器与分析器间的合作来解除这些构造中的歧义性。这一问题可以在上下文相关分析期间加以解决吗？哪种解决方案更好？

3. 有时，语言描述使用上下文相关机制检查可以使用上下文无关方法测试的性质。考虑图4-15的文法片段。事实上，虽然标准把声明限制在单一*StorageClass*说明符上，但这一文法片段允许任意数量的*StorageClass*说明符。

   a. 重写这一文法以在文法上实施这一限制。

   b. 同样地，这一语言只允许*TypeSpecifier*的特定组合。`long`只能与`int`或`float`组合；`short`只能与`int`组合。`signed`或者`unsigned`可以出现于`int`的任意形式。`signed`还可以出现在`char`上。这些限制可以写入文法中吗？

   c. 提出一个为什么作者这样构造这个文法的解释。（提示：扫描器为任意的*StorageClass*值返回单一记号类型，为任意的*TypeSpecifiers*返回另一个记号类型。）

   d. 你的这一文法的版本改变这一分析器的整体速度吗？在构建C语言的分析器中，你将使用如图4-15
736 的文法，还是更喜欢你的改编文法呢？给出你的答案的正当性。

4.5节

1. 面向对象语言允许操作符和函数重载。在这些语言中，函数名字不总是惟一的标识符，因为你可以有多个相关的定义，如用下面的形式：

```
void Show(int);
void Show(char *);
void Show(float);
```

为了查找的目的，编译器必须为每一个函数构造不同的标识符。有时候，这样的重载函数还有不同的返回值。你如何为这样的函数创建不同的标识符？

2. 继承可能为面向对象语言的实现带来问题。当对象类型*A*是对象类型*B*的一个双亲时，使用形如a←b的语法，程序可以把“指向*B*的指针”赋给“指向*A*的指针”。这将不会引发问题，因为*A*能做的每一件事*B*也能做。然而，你却不能把“指向*A*的指针”赋给“指向*B*的指针”，因为对象类*B*可以实现对象类*A*没有实现的方法。

设计一种机制，它可以使用专门语法制导翻译来决定这种指针赋值是否被允许。

第5章

5.3节

737 1. 说明如何用抽象语法树，控制流图和四元组表示下面的代码片段。讨论每一种表示的优点。对于什么样的应用，一种表示可能比其他表示更优越？

```
if (c[i] ≠ 0)
   then a[i] ← b[i] ÷ c[i];
   else a[i] ← b[i];
```

2. 检查下面的程序，画出它的CFG并以线性代码给出它的SSA形式。

```
...
x ← ···
y ← ···
a ← y + 2
b ← 0
```

```
while(x < a)
   if (y < x)
      x ← y + 1
      y ← b × 2
   else
      x ← y + 2
      y ← a + 2;
   w ← x + 2
   z ← y × a
   y ← y + 1
```

5.4节

1. 说明如何将表达式x-2×y翻译成抽象语法树、静态单一赋值形式、单地址代码、二地址代码和三地址代码。

5.5节

1. 考虑下面用C语言写成的三个过程:

738

```
static int max = 0;
void A(int b, int e)
{
  int a, c, d, p;
  a = B(b);
  if (b > 100) {
     c = a + b;
     d = c * 5 + e;
  }
  else
     c = a * b;
  *p = c;
  C(&p);
}
```

```
int B(int k)
{
  int x, y;
  x = pow(2, k);
  y = x * 5;
  return y;
}

void C(int *p)
{
  if (*p > max)
     max = *p;
}
```

a. 假设编译器是根据寄存器到寄存器内存模型工作的。编译器将迫使过程A、B和C中哪些变量存储于内存中？证明你的答案。

b. 假设编译器是根据内存到内存模型工作的。考虑if-else结构中if子语句的两个语句的执行。如果编译器在计算的那个地点有两个寄存器可用，那么在执行这两个语句期间，为了把值带到寄存器中并再把它们带回到内存，编译器将需要发行多少个装入和存储操作？如果有三个寄存器可用，那么结果将会如何？

2. 在FORTRAN中，使用equivalence语句可以迫使两个变量从同一个存储位置开始。例如，下面的语句迫使a和b共享存储:

```
equivalence (a,b)
```

如果一个局部变量在equivalence语句中出现，编译器能在整个过程中把这个局部变量保存在寄存器中吗？证明你的答案。

739

5.7节

1. 编译器的某个部分必须负责把每一个标识符放入符号表中。

a. 扫描器或分析器应该把标识符放入符号表中吗？它们都有机会这样做。

b. 这一问题，声明前使用原则，以及在含有FORTRAN 77这样的歧义性的语言中，函数调用中的下标的歧义性消除之间存在相互作用吗？

2. 编译器必须把信息存储于程序的IR版本中，使它可以取回每一个名字的符号表入口。编译器设计者可做的选择包括指向源字符串的指针和进入符号表的下标。当然，聪明的实现者也许会发现其他选择。

名字的两种表示方法各有什么优缺点？你将如何表示名字？

3. 你正在为你喜欢的词法作用域语言编写编译器。

740 考虑下面的示例程序：

```
1  procedure main
2     integer a, b, c;
3     procedure f1(w,x);
4        integer a,x,y;
5        call f2(w,x);
6        end;
7     procedure f2(y,z)
8        integer a,y,z;
9        procedure f3(m,n)
10          integer b, m, n;
11          c = a + b + m + n;
12          end;
13       call f3(c,z);
14       end;
15    ...
16    call f1(a,b);
17    end;
```

a. 画出这一程序的符号表和它在第11行的内容。

b. 当分析器进入一个新的过程以及当它离开一个过程时，符号表管理需要什么动作？

4. 决定下面C++程序的输出：

```
int i = 2;
  while (i == 2) {
    i--;
    int i = 2;
    i++;
    cout << i << " ";
}
cout << i << "\n";
```

5. 符号表最常用的实现技术是散列表，其中插入和删除的期望代价是O(1)。

a. 散列表中的插入和删除的最坏情况是什么？

b. 提出另外一个实现方案，它保证O(1)的插入和删除。

第6章

6.2节

1. 考虑下面的C程序。

```
int Sub(int i, int j) {
   return i - j;
}
int Mul(int i, int j) {
```

```
    return i * j;
}
int Delta(int a, int b, int c) {
    return Sub(Mul(b,b), Mul(Mul(4,a),c));
}
void main() {
    int a, b, c, delta;
    scanf("%d %d %d", &a, &b, &c);
    delta = Delta(a, b, c);
    if (delta == 0)
      puts("Two equal roots");
    else if (delta > 0)
      puts("Two different roots");
    else
      puts("No root");
}
```

741

给出它的调用树和执行历史。

2. 考虑下面的C程序。

```
void Output(int n, int x) {
    printf("The value of %d! is %s.\n", n, x);
}
int Fat(int n) {
    int x;
      if (n > 1)
    x = n * Fat(n - 1);
    else
      x = 1;
    Output(n, x);
    return x;
}
void main() {
    Fat(4);
}
```

给出它的调用树和执行历史。

742

6.3节

1. 某些程序设计语言允许程序员在局部变量的初始化中使用函数，但在全局变量的初始化中却不能使用函数。

   a. 是否存在解释语言定义中这看似不合情理现象的实现上的解释？

   b. 允许使用函数的调用结果来初始化全局变量所需要的机制是什么？

2. 编译器设计者可以用若干种方式优化AR分配。例如，编译器可以：

   a. 静态地分配叶过程（也就是那些无过程调用的过程）的AR。

   b. 把那些总是一起被调用的过程的AR结合起来。（当α被调用时，它总是调用β。）

   c. 取代AR的堆分配，使用实存块风格的分配器。

   对于每一个方案，考虑下面的问题：

   a. 哪些调用可以受益？最好的情况下如何？最坏的情况下又如何？

   b. 对运行时间空间运用产生什么影响？

3. 考虑下面的Pascal程序，其中只给出了过程调用和变量声明。

```
1  program Main(input, output);
2    var a, b, c : integer;
3    procedure P4; forward;
4    procedure P1;
5      procedure P2;
6        begin
7        end;
8      var b, d, f : integer;
9      procedure P3;
10       var a, b : integer;
11       begin
12         P2;
13         end;
14     begin
15       P2;
16       P4;
17       P3;
18       end;
19   var d, e : integer;
20   procedure P4;
21     var a, c, g : integer;
22     procedure P5;
23       var c, d : integer;
24       begin
25         P1;
26         end;
27     var d : integer;
28     begin
29       P1;
30       P5;
31       end;
32   begin
33     P1;
34     P4;
35     end.
```

743

a. 构造类似于图6-3的静态坐标表。

b. 构造表示程序中嵌套关系的图。

c. 构造表示程序中调用关系的图。

744

4. 画出编译器为支持如下定义的类型Dumbo的对象所需生成的结构：

```
class Elephant {
   private int Length;
   private int Weight;
   static int type;

   public int GetLen();
   public int GetTyp();
}

class Dumbo extends Elephant {
   private int EarSize;
   private boolean Fly;

   public boolean CanFly();
}
```

6.4节

1. 在程序设计语言中，不同变量引用同一个对象（内存区域）的情况被认为是不理想的。考虑下面通过引用传递参数的Pascal过程。

```
procedure mystery(var x, y : integer);
  begin
    x := x + y;
    y := x - y;
    x := x - y;
  end;
```

如果上述成下溢不在算术操作中出现，那么：

a. 当我们使用不同变量a和b调用mystery时，它将产生什么样的结果？

b. 如果我们在调用mystery时把一个变量同时传送到两个参数，那么期望的结果将是什么？在这种情况下，真实结果又是什么？

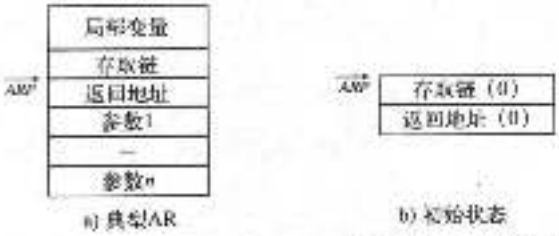
745

2. 考虑下面的类Pascal伪程序设计语言中的程序，模拟它在值调用、引用调用、名字调用和值结果调用规则下的执行。给出上述每种情况下打印语句的结果。

```
1  procedure main;
2    var a : array[1...3] of int;
3        i : int;
4    procedure p2(e : int);
5      begin
6        e := e + 3;
7        a[1] := 5;
8        i := 2;
9        e := e + 4;
10       end;
11   begin
12     a := [1, 10, 77];
13     i := 1;
14     p2(a[i]);
15     for i := 1 to 3 do
16       print(a[i]);
17     end.
```

6.5节

1. 假设一个Pascal语言实现的AR是以图表a所示形式分配的域。（为简单起见，我们省略了某些域。）ARP是指向AR的惟一指针，所以存取链是之前的ARP值。这个栈向本页顶部增长。图表b给出计算的初始AR。



a) 典型AR　　b) 初始状态

746

对于下面的Pascal程序，画出从函数F1返回之前在这一栈上的AR集合，把所有条目都放入这些AR中。使用行编号表示返回地址。画出存取链的有向弧。标清局部变量和参数的值。以每一个AR的

程序名标示这个AR。

```
1  program main(input, output);
2    procedure P1( function g(b: integer): integer);
3      var a: integer;
4      begin
5        a := 3;
6        writeln(g(2))
7        end;
8    procedure P2;
9      var a: integer;
10     function F1(b: integer): integer;
11       begin
12         F1 := a + b
13         end;
14     procedure P3;
15       var a:integer;
16       begin
17         a := 7;
18         P1(F1)
19         end;
20     begin
21       a := 0;
22       P3
23       end;
24   begin
25     P2
26     end.
```

2. 考虑下面的Pascal程序。假设AR遵从与前面问题相同的布局，具有相同的初始条件，只是实现使用全局显示而非存取链。

747

```
1  program main(input, output);
2    var x : integer;
3        a : float;
4    procedure p1();
5      var g:character;
6      begin
7        ...
8        end;
9    procedure p2();
10     var h:character;
11     procedure p3();
12       var h,i:integer;
13       begin
14         p1();
15         end;
16     begin
17       p3();
18       end;
19   begin
20     p2();
21     end
```

画出当这一程序达到过程p1中的第7行时，在运行时栈上的AR集合。

6.6节

1. 链接约定的概念与大程序结构之间有什么关系？与中间语言有什么关系？链接约定如何提供中间语言调用？
2. 假设编译器能够分析代码确定诸如“从这一点开始，变量v在这一过程中不再使用”或者“变量v在这一过程的下一次使用是在第11行，”等事实，且假设编译器把下面三个过程的所有局部变量都保存在寄存器中。 748

```
procedure main
   integer a, b, c
   b = a + c;
   c = f1(a,b);
   call print(c);
   end;
procedure f1(integer x, integer y)
   integer v;
   v = x * y;
   call print(v);
   call f2(v);
   return -x;
   end;
procedure f2(integer q)
   integer k, r;
   ...
   k = q / r;
   end;
```

   a. 过程f1中的变量x在两个过程调用间是活着的。为使编译代码的执行最快，编译器应该把这一变量保留在调用者存储寄存器还是被调用者存储寄存器呢？证明你的答案。
   b. 考虑过程main中的变量a和c。再次假设编译器尝试最大化编译代码的速度。这时，编译器应该把a和c保留在调用者存储寄存器还是被调用者存储寄存器呢？证明你的答案。 749

6.7节

1. 考虑带有局部变量和参数的尾递归函数。描述一个存储机制，它消除对栈分配活动记录的需求。
2. C语言的类型系统的什么性质使得自动垃圾回收变得困难？

第7章

7.2节

1. 内存布局影响指定给变量的地址。假设字符变量没有调整限制，短整数变量必须调整到半字（2字节）边界，整数变量必须调整到字（4字节）边界，长整数变量必须调整到双字（8字节）边界。考虑下面的声明集合。

```
char a;
long int b;
int c;
short int d;
long int e;
char f;
```

   画出这些变量的内存映射：

a. 假设编译器无法对这些变量做重排序。

b. 假设编译器可以为节省空间而对这些变量重排序。

2. 对于下面的每一类变量，说明编译器可以为这样的变量分配的内存空间位置。可能的答案包括寄存器、活动记录、（带有不同可视性的）静态数据区域和运行时堆。

a. 局部于过程的变量。

b. 全局变量。

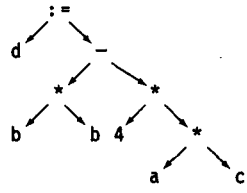
c. 动态分配的全局变量。

d. 形式参数。

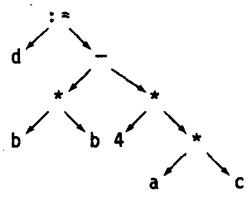
750 e. 编译器生成的临时变量。

7.3节

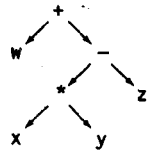
1. 使用7.3节的树遍历代码生成算法为下面的表达式树构建朴素代码。假设对寄存器数目没有限制。



2. 寻找使用ILOC指令集合评估下面的树所需要的寄存器的最小数目。对于每一个非叶子结点，指出为了得到寄存器的最小数目，哪一个孩子必须先评估。



a)



b)

3. 使用标准优先度和从左到右评估构建下面两个算术表达式的表达式树。计算使用ILOC指令集合评估每一个表达式树所需要的寄存器的最小数目。

a. `((a + b) + (c + d)) + ((e + f) + (g + h))`

751 b. `a + b + c + d + e + f + g + h`

7.4节

1. 为下面代码序列生成判定ILOC。（在解答中不应出现分支指令。）

```
if (x < y)
   then z = x * 5;
   else z = y * 5;
w = z + 10;
```

2. 如7.4节所述，下面的C语言的表达式的短路代码避开潜在的除零错误。

```
a != 0 && b / a > 0.5
```

如果源语言的定义不对布尔值表达式定义短路评估，那么编译器为了优化能够为这样的表达式生成短路代码吗？可能出现什么样的问题？

7.5节

1. 对于按行优先存储的字符数组A[10...12, 1...3]，在生成的代码中至多使用四个算术运算计算引用A[i, j]的地址。
2. 什么是内情向量？给出上面问题中的字符数组的内情向量的内容。为什么编译器需要内情向量？
3. 在实现C语言的编译器时，使编译器执行数组引用的区域检查是明智的吗？假设使用区域检查且C语言程序的所有数组引用都成功地通过这些检查，这个程序可能存取数组区域外的存储区域吗？例如，对于一个声明为下界为零、上界为N的数组A，存取A[-1]是否可能？ 752

7.6节

1. 考虑下面7.6.2节中的字符拷贝循环：

```
                    loadI  @b     ⇒ r@b  // get pointers
                    loadI  @a     ⇒ r@a
                    loadI  NULL   ⇒ r1   // terminator
                L1: cload  r@b    ⇒ r2   // get next char
                    cstore r2     ⇒ r@a  // store it
do {
  *a++ = *b++;      addI   r@b,1  ⇒ r@b  // bump pointers
} while (*b!='\0')  addI   r@a,1  ⇒ r@a
                    cmp_NE r1,r2  ⇒ r4
                    cbr    r4     → L1,L2
                L2: nop                  // next stmt
```

修改这一代码，使得它对于任意超出a的分配长度的企图分支到L_{sov}处的出错处理器。假设a的分配长度被作为无符号四字节整数存储于距a的开始偏移为－8的地方。

2. 考虑下面的串赋值a＝b的实现；如7.6节所示。对于串a和b，它假设串的实际长度存储于距该串的开始偏移为－4的地方，而且那个串的分配长度都存储于距该串的开始偏移为－8的地方。 753

```
                loadI    @b     ⇒ r@b
                loadAI   r@b,-4 ⇒ r1     // get b's length
                loadI    @a     ⇒ r@a
                loadAI   r@a,-8 ⇒ r2     // get a's length
                cmp_LT   r2,r1  ⇒ r3     // will b fit in a?
                cbr      r3     → Lsov,L1 // Lsov raises error
            L1: loadI    0      ⇒ r4     // counter
a = b;          cmp_LT   r4,r1  ⇒ r5     // more to copy ?
                cbr      r5     → L2,L3
            L2: cloadAO  r@b,r4 ⇒ r6     // get char from b
                cstoreAO r6     ⇒ r@a,r4 // put it in a
                addI     r4,1   ⇒ r4     // increment offset
                cmp_LT   r4,r1  ⇒ r7     // more to copy ?
```

```
        cbr     r7     → L2,L3
L3: nop                        // next statement
```

它使用两个cmp/cbr对来实现循环结束测试，一个在标签为L_1的块中，一个在标签为L_2的块中。

在一个编译代码的尺寸非常重要的环境中，编译器可能要用到L_1的jumpI取代循环尾部的cmp/cbr对。

这样的改变是如何影响循环的执行时间的？存在在其上运行更快的机器模型吗？存在在其上运行较慢的机器模型吗？

3. 图7-9给出如果使用面向字的内存操作来执行两个字对齐串的字符串赋值。任意赋值可能生成非对齐情况。

754

a. 写出你希望你的编译器为任意PL/I风格的字符赋值发行的ILOC代码，例如赋值为

```
fee(i:j) = fie(k:l);
```

其中j-i=l-k。这一语句把fie中从位置k到位置l的字符拷贝到串fee中从i到j的位置中。

包括使用面向字符的内存操作版本和使用面向字的内存操作版本。你可以假设fee和fie在内存内不交迭。

b. 程序员可以创建交迭的字符串。在PL/I中，程序员可以书写

```
fee(i:j) = fee(i+1:j+1);
```

或，甚至更恶劣地书写

```
fee(i+k:j+k) = fee(i:j);
```

这如何复杂化编译器为字符赋值必须生成的代码？

c. 存在编译器运用于各种字符拷贝循环以改进运行时行为的优化吗？它们有什么帮助？

7.7节

1. 考虑下面的C语言类型声明：

```
struct S2 {
  int i;
  int f;
};

union U {
   float r;
   struct S2;
};

struct S1 {
   int a;
   double b;
   union U;
   int d;
};
```

755

为S1构筑一个结构元素表。在其中包括所有编译生成对类型S1的变量元素的引用时所需的所有信息，包括每一个元素的名字、长度、偏移和类型。

2. 考虑下面的C语言声明：

```
struct record {
  int StudentId;
  int CourseId;
  int Grade;
} grades[1000];
int g, i;
```

说明编译器为了把变量g中的值作为grade的第i个元素的grade存储时所生成的代码，假设：

a. 数组grade被作为结构数组存储。

b. 数组grade被作为数组结构存储。

7.8节

1. 作为一名程序员，你关注于你所生成的代码的效率。你最近手工实现了一个扫描器。这个扫描器把大部分时间耗费在包含一个大选择语句的while循环上。
   a. 选择语句的不同实现技术是如何对你的扫描器的效率产生影响的？
   b. 你将如何改变你的源代码以便改进在选择语句的每一个实现策略下你的运行时性能？
2. 把下面的C语言中的尾递归函数转换成循环。 756

```
List * last(List *l) {
  if (l == NULL)
     return NULL;
  else if (l->next == NULL)
     return l;
  else
     return last(l->next); }
```

7.9节

1. 假设x是非歧义、局部整型变量，且x作为在它被声明过程中的一个引用调用实参被传递。因为它是局部且是非歧义的，所以在它的生存期内，编译器可能设法把它保存在一个寄存器中。因为它被作为一个引用调用参数传递，所以在调用点它必须有一个内存地址。
   a. 编译器应该把x存储在哪里？
   b. 在调用场所，编译应该如何处理x？
   c. 如果x是作为一个值调用参数被传递的，你的答案将做如何改变？
2. 链接约定是编译器与任意编译代码的外部调用者之间的一种契约。它创建可以用于调用一个过程并得到这一过程返回的任意结果（同时保护调用者的运行时环境）的已知接口。因此，只有当一个对链接约定的违规不能被编译代码的外部发现时，编译器才可以做这一违规操作。
   a. 在什么样的情况下编译器可以肯定使用不同的链接是安全的？给出一个真实程序设计语言中的例子。
   b. 在上述这些情况下，编译器可以怎样改变调用序列和链接约定？ 757

7.10节

1. 解释在类记录中第一域的目的，诸如图7-13中的sc、mc和giant。
2. 在Java语言中，数据成员的存取规则不同于方法成员的存取规则。例如，假设类b继承自类a，且这两个类都定义了名为foo的成员。进一步假设对象o是作为类b的一个实例分配的，但是当存取它的成员foo时，它被处理为类a中的对象。如果foo是一个方法成员，那么调用解析为类b中定义的方法。但是，如果foo是一个数据成员，那么将对类a中定义的域做存取。

   解释你如何实现支持这两种不同存取规则的Java编译器。
3. 多重继承的某些实现把“蹦床函数”用于调整对象记录指针，如7.10.2节所述。
   a. 什么时候代码需要调整在Java中用this表示的对象记录指针？
   b. 什么样的类需要多重蹦床函数，即带有不同偏移的蹦床函数？
4. 在以动态类结构为特征的程序设计语言中，必须动态分配的方法调用的数量可能很大，其中动态类结构是运行时可以变化的类结构。如7.10.2节中所述，方法缓冲可以绕过它们而减小这些查找的运行时代价。

   作为对全局方法缓冲的一种替代，实现可以在每一个调用场所维护一个入口方法缓冲。它记录该

调用场所最近发行的方法的地址，以及相关的类。

开发使用及维护这样的内联方法缓冲的伪代码。解释这个内联方法缓冲的初始化和对支持内联方
758 法缓冲所需的一般方法查找程序的修改。

第8章

8.3节

1. 很多优化和代码生成算法被设计成在DAG上操作。即使是使用低级IR的编译器，如ILOC优势也构建DAG。

   a. 草拟一个从ILOC表示的基本块构建DAG的算法。

   b. 将你的算法与图8-3所示的值编号算法做比较。

   c. 草拟一个从你的DAG重新生成ILOC的算法。

   d. 你能否扩展你的DAG构造算法使其包含值编号的某些其他属性，诸如常量叠入或处理代数等式?

2. 考虑下面的两个基本块:

```
a ← b + c        a ← b + c
d ← c            e ← c + c
e ← c + d        f ← a + c
f ← a + d        g ← b + e
g ← b + e        h ← b + c
h ← b + d
```

   a. 构筑每一个块的DAG。

   b. 值编号每一个块。

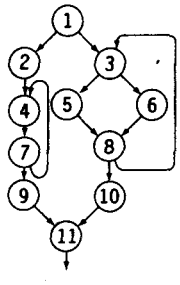
   c. 解释由以上两个技术在寻找冗余中的差异。

759 d. 在每一个块的尾端，f和g有相同的值。为什么这些算法很难发现这一事实?

3. 给定一个ILOC操作的线性列表，开发一个寻找基本块的算法。扩展你的算法构筑表示块间关系的控制流图。

8.4节

1. 考虑下面的控制流图:



   a. 识别上图中的扩展基本块。

   b. 识别出上图中的循环。

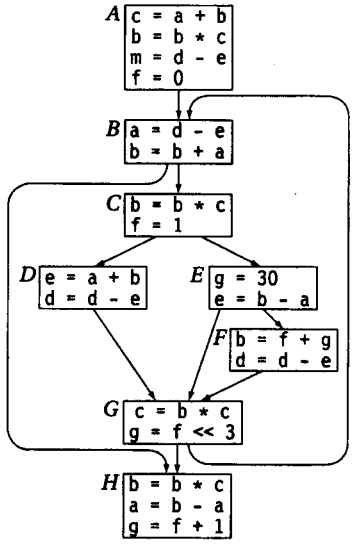
   c. 写出C语言中可以生成上面的控制流图的过程。

2. 给出一个算法，这一算法可以高效地确定被包含在一个循环L内的基本块集合。假设给你控制流图中

的循环结束边（也就是目标支配源头的边）。例如，上图中从7到4的边和8到3的边都是上一问题中CFG的循环结束边。

8.5节

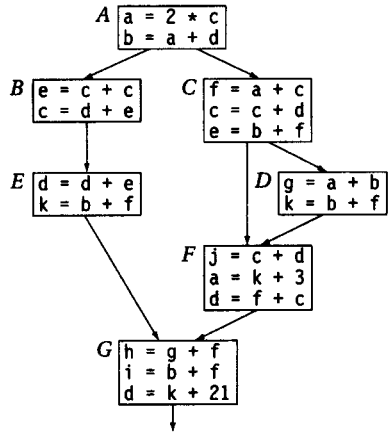
使用下面的控制流图解决问题1和2: 760



1. 使用上面所示的CFG:
   a. 在这一CFG中寻找扩展基本块。
   b. 寻找每一个基本块的支配者集合。
   c. 构筑这一CFG的支配树。
2. 使用上面练习的CFG:
   a. 将超局部值编号运用到这一CFG上。
   b. 将基于支配者的值编号运用于这一CFG上。

8.6节

使用下面的控制流图解决问题1和2。 761



1. 与值编号技术不同，全局冗余消除算法（GRE）使用可用表达式发现不同的冗余集合。
   a. 在前面的CFG中，GRE把哪些表达式作为冗余处理？
   b. 基于支配者的值编号找到哪些GRE无法找到的冗余表达式。
2. 计算上面CFG中的各块的DEEXPR、EXRPKILL和AVAIL集合。重写这一代码，使用拷贝表达式取代冗余表达式。
3. 如图8-8所示，EXPRKILL的计算比DEEXPR集合的计算更复杂。草拟算法
   a. 为表达式构造名字空间。
   b. 构造从VARKILL到EXPRKILL的映射。
   评估你的算法的渐近复杂度。你可以使它多快？
4. 8.6.2节给出插入全局冗余消除所需的拷贝操作的一种方案。这一方案插入一些无用的拷贝操作（这些操作将被死代码消除器消除）。开发另一个生成更少无用拷贝的插入拷贝技术。

762

8.7节

1. 本节描述两种代码复制的形式，复制和内联替换。
   a. 程序设计语言的什么属性阻碍编译器复制一个块或内联替换一个被调用过程？
   b. 复制和内联替换都既有效益又有负荷。给出编译器可以用于决定复制一个块的时机的策略。给出编译器可以用于决定内联替换一个调用的时机的策略。
2. 在某些环境中，代码复制总是有意义的。例如，只从一个调用场所被调用的过程是内联替换的强候选，除非内联使调用者太大而无法适合存储器层次的某个层次。
   a. 在什么样的环境下，编译器应该总是复制一个块？
   b. 在什么样的环境下，编译器应该总是内联替换一个调用？

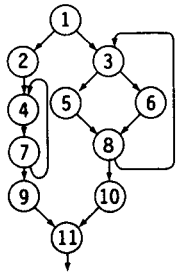
第9章

9.2节

1. 图9-2中的活性分析算法初始化每一个块的LIVEOUT集合为ϕ。有其他初始化的可能吗？它们改变分析的结果吗？证明你的答案。
2. 图9-2中的活性分析算法在CFG中的所有结点上迭代，但是它不特定处理块的顺序。这一顺序重要吗？
3. 在活变量分析中，编译器应该如何处理包含过程调用的块？这样的块的UEVAR集合应该包含什么？它的VARKILL集合应该是什么？

763

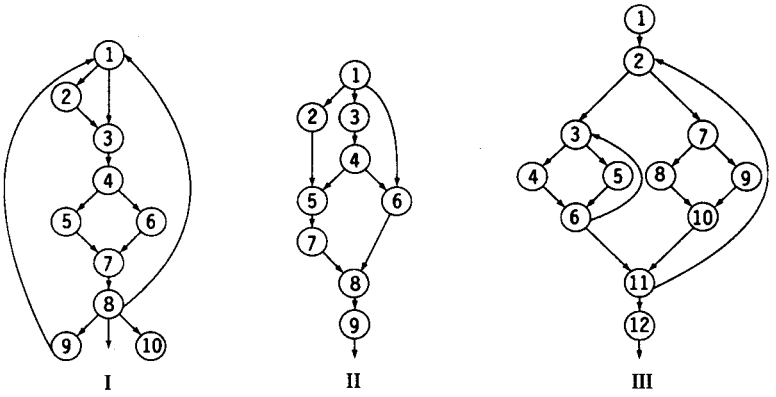
4. 考虑下面的控制流图：



   a. 为上图中的结点计算逆前序编号。
   b. 为上图中的结点计算逆后序编号。

9.3节

1. 考虑下面的控制流图:

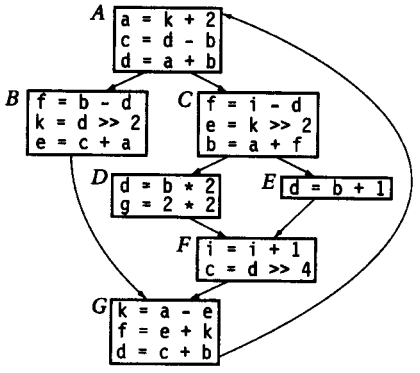


764

a. 计算CFG I、II和III的支配者树。

b. 分别计算CFG I中结点3和结点5，CFG II中的结点4和结点5，以及CFG III的结点3和结点11的支配边界。

2. 把下面的控制流图中的代码转换成SSA形式。给出ϕ函数插入和重命名后的最终代码。



3. 考虑所有这样的块的集合，由于某个块b中的赋值$x\leftarrow\cdots$它得到某个ϕ函数。图9-11中的算法在DF(b) 中的每一个块内插入一个ϕ函数。那些块中的每一个都被加入工作列表；它们反过来又可以把在它们的DF集合内的结点加入这一工作列表中。这一算法使用一个检查列表来避免把一个块多次加入工作列表中。称所有这些块的集合为$DF^+(b)$。

我们可以把$DF^+(b)$ 定义为下面序列的极限。

$$DF_1(b) = DF(b)$$

$$DF_2(b) = DF_1(b) \cup_{x \in DF_1(b)} DF(x))$$

$$\cdots$$

$$DF_{i+1}(b) = DF_i(b) \cup_{x \in DF_i(b)} DF_i(x))$$

765

使用这些扩展集合$DF^+(b)$ 导致插入ϕ函数的一个更简单的算法。

a. 开发一个计算$DF^+(b)$ 的算法。

b. 开发一个使用DF^+集合插入ϕ函数的算法。

c. 将你的新算法的整体代价，包括计算DF^+集合的代价，与9.3.3节给出的ϕ插入算法的代价做比较。

4. “极大”SSA构造法既简单又直观。然而，它插入的ϕ函数的数量可能比“半裁剪”算法多很多。特别是它既插入冗余ϕ函数（$x_1 \leftarrow \phi(x_j, x_j)$）又插入结果从不被使用的死$\phi$函数。

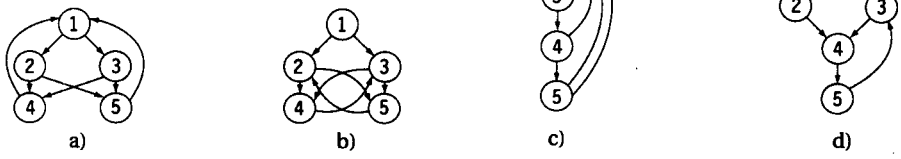
a. 给出一个检查并消除极大构造插入的额外ϕ函数的方法。

b. 你的方法能够把ϕ函数集合降低到半裁剪构造法所插入的那些ϕ函数吗？

c. 将你的方法的渐近复杂度与半裁剪构造法的渐近复杂度做对比。

9.4节

1. 对于下面各控制流图，说明它是否是可归约的。



a) b) c) d)

766

2. 证明可归约图的如下定义与使用转换T_1和T_2的定义等价：“一个图G是可归约的，当且仅当消除它的后边产生一个无环图G'。”（回想一下，后边是目标支配源头的边。）

3. 使用T_1和T_2给出一个简化图9-21中标签为“After”的图的简化序列。

第10章

10.3节

1. 优化器的主要功能之一是消除编译器在把源语言转化成IR时所带来的负荷。

a. 给出4个你希望编译器可以改进的低效率例子，并给出引发这些例子的源语言构造。

b. 给出4个你预期编译器错过的低效率例子，即使它们能够被改进。解释为什么优化器改进它们有困难。

2. 编译器可以用很多方法找到并消除冗余计算。其中包括值编号、基于可用表达式的全局公共子表达式（GCSE）消除和惰性代码移动（LCM）。

767

a. 给出两个由值编号消除的而GCSE或LCM却无法找到的冗余例子。

b. 给出LCM发现而值编号和GCSE却无法发现的例子。

c. GCSE可以发现LCM无法发现的冗余吗？给出一个例子，或证明你的答案。

3. 图10-3给出计算DEAD的算法。其标记遍是一个经典的不动点计算。

a. 解释为什么这一计算终止。

b. 这一算法找到的不动点是惟一的吗？证明你的答案。

c. 得出这一算法的严格时间边界。

4. 考虑10.3.1节的算法CLEAN。它消除无用控制流并简化CFG。

a. 为什么这一算法终止？

b. 给出这一算法的总体时间界限。

5. 对于一个表达式的多重评估，提升使用放置到CFG中它们“之上”的单一评估取代它们。一个称为下沉（sinking）的类似转换可以在代码中寻找这样的地点：（1）每一个到达块b的路径包含一个e的评估；（2）在b评估e产生相同结果；（3）e的结果在b之前不被使用。
   a. 形式化定位下沉机会所需的数据流问题并证明它的安全性。
   b. 概述一个（对于下沉）插入新评估并消除早前评估的高效算法。

10.4节

1. 操作符强度减弱的一个简单形式使用一系列执行代价较小的操作取代一个代价高的操作。例如，可以用移位和加法序列取代某些整数乘法操作。 [768]
   a. 必须满足什么样的条件才能使编译器安全地用单一移位操作取代整数操作x←y × z。
   b. 概述一个算法，对于已知常量和无符号整数的乘积且该常量不为2的幂的情况，它使用移位和加法的序列取代该乘法。
2. 编译器可以使用过程抽象来缩小它生成的代码尺寸。
   a. 为了编译器可以安全地运用过程抽象，重复代码序列必须满足什么样的条件？
   b. 你的编译器找到了两个有相同操作和寄存器名字的代码序列。然而，这一序列在中间有一个分支，而且两个实例分支到不同的目标。编译器将如何处理这样的情况？你能一般化你的想法吗？
   c. 过程抽象的负荷是什么？存在负荷超过转换效益的情况吗？

第11章

11.2节

1. 图7-2所示的树遍历代码生成器对每一个数使用一个`loadI`。重写树遍历代码生成器，使得它使用`addI`、`subI`、`rsub`、`multI`、`divI`和`rdivI`。解释你的代码生成器所需的任意额外例程或数据结构。

11.3节

1. 使用图11-5给出的规则，生成两个图11-4所示AST的覆盖。 [769]
2. 以图11-4中的树为模型，构建下面表达式的低级抽象语法树。
   a. `y ← a × b + c × d`
   b. `w ← x × y × z - 7`

   使用图11-5给出的规则构建这些树的覆盖并生成ILOC。
3. 树模式匹配假设它的输入是一棵树。
   a. 怎样扩展这些想法来处理结点可以有多个父亲DAG呢？
   b. 控制流是怎样适合这一范例的？
4. 在任意树遍历代码生成方案中，编译器都必须选择子树的评估顺序。即，在某个二叉结点n处，是先评估左子树还是右子树？
   a. 顺序的选择会影响评估整个子树所需的寄存器数量吗？
   b. 这一选择如何能够并入自底向上树模式匹配方案中？

11.4节

1. 真实的窥孔优化器必须处理控制流操作，包括条件分支、跳转和标号语句。
   a. 当窥孔优化器把一个条件分支带入优化窗口时，它应该做什么？

b. 当它遇到一个跳转时，情况有什么不同？

c. 对于标号语句情况又如何？

d. 为了改进这一状态优化器能够做什么？

2. 写出窥孔转换器执行简化和匹配功能的具体算法。

770 a. 你的每一个算法的渐近复杂度是什么？

b. 转换器的运行时间是如何受到较长输入程序、（为了简化和匹配的目的）更大的窗口和更大的模式集合的影响的？

3. 当窥孔转换器在选择代码的具体实现时简化代码。假设窥孔转换器在指令调度和寄存器分配之前运行，且假设这一转换器可以使用足够多的虚拟寄存器名字。

a. 这一窥孔转换器可以改变对寄存器的需求吗？

b. 这一窥孔转换器能够改变适合于调度器重排序代码的机会吗？

第12章

12.2节

1. 开发一个为基本块构筑优先图的算法。假设这一块是用ILOC写成的，且在这一块外部定义的任意值在这个块开始执行之前已准备就绪。

2. 如果优先图的主要用途是指令调度，那么目标机器上实际等待的精确模型化是至关重要的。

a. 优先图应该如何模型化由歧义内存引用所引发的非确定性？

b. 在某些管道化处理器中，读后写入的等待可能比写后读出的等待短。例如，下面的序列

```
[ add r10,r12 ⇒ r2 | sub r13,r11 ⇒ r10 ]
```

将在把sub的结果写入r_{10}之前，为了在add中的使用先读取r_{10}的值。对于这样的体系结构，编译器如何表示优先图中的反依赖性呢？

771 c. 某些处理器绕开内存以降低写后读出延迟。对于这样的机制，诸如下面的序列

```
storeAI r21    ⇒ rarp,16
loadAI  rarp,16 ⇒ r12
```

将（在这一序列开始处的r_{21}中）存储的值直接向前传给装入结果（r_{12}）。依赖图如何反映这种硬件上的绕过特性？

12.3节

1. 扩展如图12-3所示的局部列表调度算法来处理多功能单元。假设所有功能单元有相同的容量。

2. 所有调度算法的关键方面是设置初始优先度和当若干带有相同优先度的操作在同一周期准备就绪时进行平局加赛。文献中所给出的某些平局加赛包括：

a. 取优先图中带最多后代的操作。

b. 取最长等待时间的操作。

c. 取执行后活着的操作数最少的操作。

d. 取随机选取的操作。

e. 取任意计算之前的一个装入。

对于每一种平局加赛，给出一个理由：也就是对某人为什么要提议这一平局加赛给出猜测。你首选的平局加赛是哪一个？其次又是哪一个？证明（或解释）你的答案。

3. 大多数现代微处理器都在某个或所有分支操作上有等待槽（delay slot）。使用单一等待槽，紧跟在这一分支后面的操作在分支处理的同时执行；因此，调度一个分支的理想槽是在基本块的倒数第二个周期。（大多数处理器有不执行等待槽的分支版本，因此编译器可以避免在未填充等待槽生成nop指令。） 772
   a. 你将如何改编列表调度算法以改进它“填充”等待槽的能力。
   b. 概述一个填充等待槽的后调度遍。
   c. 提出不被有用操作填充的分支等待槽的建设性使用。

12.4节

1. 操作出现的顺序决定值创建的时间和它们最后一次使用的时间。这些综合效应决定值的寿命。
   a. 调度器如何减小对寄存器的需求？给出适合列表调度器的一个具体的平局加赛。
   b. 面向寄存器平局加赛和产生短调度的调度器能力之间的相互作用是什么？
2. 软件管道交迭循环迭代以产生与硬件管道相似的效应。
   a. 你希望软件管道对寄存器的需求有什么影响？
   b. 调度器如何使用预测执行来减小软件管道对代码空间的不利影响。

第13章

13.3节

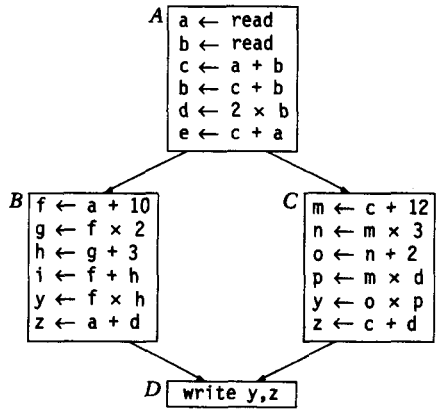
1. 考虑下面的ILOC基本块。假设r_{arp}和r_i在这个块的入口都是活的。 773

```
loadAI  r_arp,12 ⇒ r_a
loadAI  r_arp,16 ⇒ r_b
add     r_i,r_a  ⇒ r_c
sub     r_b,r_i  ⇒ r_d
mult    r_c,r_d  ⇒ r_e
multI   r_b,2    ⇒ r_f
add     r_e,r_f  ⇒ r_g
storeAI r_g      ⇒ r_arp,8
jmp              → L003
```

   a. 给出在这个块上使用自顶向下局部算法分配寄存器的结果。假设目标机器带有4个寄存器。
   b. 给出在这个块上使用自底向上局部算法分配寄存器的结果。假设目标机器带有4个寄存器。
2. 自顶向下局部分配器在对值的处理方面是相当朴素的。它把一个值在整个基本块分配到寄存器上。
   a. 一个改进的版本可能是计算在这个块内的活性区域，并在值的活性区域内为值分配寄存器。为完成这一任务要做什么必要的修改？
   b. 进一步的改进可能是，当一个值在其活性区域无法容纳于单一寄存器时，分割这一活性区域。给出（1）在寄存器不可用时分割指令（或指令区域）附近的活性区域；并（2）对这一活性区域的余下各个部分重新分配；所需的数据结构和算法修正。
   c. 使用这些修正，频率计数技术应该生成更好的分配。预期你的结果与使用自底向上局部算法比较时结果如何?证明你的答案。

13.4节

1. 考虑如下的控制流图： 774



假设read返回一个外部媒介的值而write把一个值传送给外部媒介。

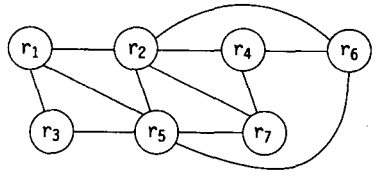
a. 计算每一个块的LIVEIN和LIVEOUT集合。

b. 将自底向上算法运用于每一个块A、B和C。假设有三个寄存器可以用于计算。如果块b定义一个名字n且$n \in$LIVEOUT(b)，那么分配器必须把n存储回内存中，使得它的值在后继块可用。同样地，如果块b在名字n的所有局部定义之前使用n，那么它必须从内存装入n的值。给出其结果代码，包括所有的装入和存储。

c. 给出允许LIVEOUT(A) 中的某些值保留在寄存器中以避免后继块中的初始装入的方案。

13.5节

775 1. 考虑下面的冲突图:



假设目标机器刚好有三个寄存器。

a. 把自底向上全局着色算法运用于此图。哪些虚拟寄存器将被溢出？哪些虚拟寄存器被着色？

b. 溢出结点的选择有区别吗？

c. 早前的着色分配器溢出选择时受到限制的所有活性区域。它们不是使用如图13-8所示的算法，而是使用如下的方法:

```
initialize stack to empty
while (N = φ)
  if ∃ n ∈ N with n° < k then
    remove n and its edges from I
    push n onto stack
  else
    pick a node n from N
    mark n to be spilled
```

如果这一方法标出任意溢出的结点，那么分配器插入溢出代码，并在修改后的程序上重复这一分配过程。如果没有溢出结点，它继续用自底向上全局分配器所描述的方法进行着色。

当这一算法被用于示例冲突图时将发生什么？用于选择溢出结点的机制改变结果吗？

2. 寄存器分配之后，仔细的代码分析可能会发现，在代码的某些伸长部分存在未使用的寄存器。在自底向上图着色全局分配器中，因为活性区域的溢出方法中细节上的缺点，这种情况将会出现。 776

a. 解释这一情况是如何出现的？

b. 编译器如何发现这一情况的发生及发生的地点？

c. 我们可以使用这些未使用的全局框架内部和外部的寄存器做些什么？

13.6节

1. 当图着色全局分配器达到没有适合特定活性区域的颜色的地点LR_i时，它溢出或分割这一个活性区域。另外，它也许设法对LR_i的一个或多个邻居重新着色。考虑$<LR_i, LR_j> \in I$且$<LR_i, LR_k> \in I$，但是$<LR_j, LR_k> \notin I$的情况。如果LR_j和LR_k已被着色，且得到了不同的颜色，那么分配器可以把其中一个重新着色为另一个活性区域的颜色，为LR_i释放一个颜色。

a. 概述一个发现是否存在LR_i的合法且实用的重新着色的算法。

b. 你的技术对寄存器分配器的渐近复杂度的影响是什么？

c. 你应该考虑对LR_k的邻居进行递归重新着色吗？解释你的理由。

2. 自底向上全局分配器的描述提出：为被分割活性区域的每一个定义和使用插入溢出代码。自顶向下全局分配器首先把这一活性区域分割成以块为单位的片，然后，当结果不受限制时把这些片组合起来，最后给它们指定颜色。

a. 如果给定块有一个或多个自由寄存器，多次溢出一个活性区域会造成浪费。对自底向上全局分配器的溢出方案提出一个避免这一问题的改进。

b. 如果给定块有太多的交迭活性区域，那么分割已溢出活性区域对处理这个块中的问题毫无帮助。提出一个（不同于局部分配的）机制来改进在块中对寄存器要求较高的模块内自顶向下全局分配器的行为。 777

索　引

索引中的页码为英文原书页码，与书中边栏页码一致。

φ-functions（φ函数），229-231，636
  arguments（φ函数变量），231，455
  behavior（φ函数行为），229
  block execution（φ函数块执行），478
  concurrent execution（φ函数并发执行），464
  dead（φ函数死亡）466
  defined（φ函数定义），228
  as definitions（φ函数作为定义）456
  duplicate（φ函数复制），avoiding（回避φ函数复制），464
  evaluation rules（φ函数评估规则），479
  example（φ函数例子），465-466，468
  execution（φ函数执行），230
  extraneous（多余φ函数），457
  implementation and（φ函数实现和），474
  insertion（φ函数插入），228，229，230，456，464
  insertion algorithm（φ函数插入算法），464
  placing（φ函数放置），463-466
  targets（φ函数目标），472，479

A

abstract syntax trees（抽象语法树）(AST)，139
  building（抽象语法树构建），196-197
  defined（抽象语法树定义），214
  exposing detail in（在抽象语法树中揭示更多细节），556
  illustrated（阐明抽象语法树），214
  low-level（低级抽象语法树），218，556，559
  mapping，to operations（抽象语法树到操作的映射），559
  node type（抽象语法树结点类型），557
  nodes（抽象语法树结点），216，394
  size reduction（抽象语法树大小的减小），216
  source-level（源代码级抽象语法树），211，217，218
  as source-level representation（作为源代码级表示的抽象语法树），215
  structure（抽象语法树的结构），187
  tiling（抽象语法树覆盖），559，564
  uses（抽象语法树的使用），215
  See also syntax-related trees
abstraction（抽象）
  concept（抽象的概念），275
  control（抽象控制），252，254-256
  grammar nonterminals and（抽象文法非终结符和），560
  level of（抽象级别），217-218，548
  overhead，reducing（降低抽象负荷），392
  procedure（抽象过程），539
  transition diagrams and（抽象转换图和），31
accept action（接受动作），117
access links（存取链接），282-283
  defined（存取链接定义），282
  global display vs.（存取链接全局显示与），285
  maintaining（存取链接维护），283，285
  managing（存取链接管理），290
  null（存取链接是空的），283
  overhead（存取链接负荷），285
  references through（存取链接通过…引用），285
  use illustration（存取链接使用说明），282
Action table（动作表格），116，117，118
  directly encoding（动作表格直接编码），145-147
  “don't care” states（动作表格“非关心”状态），146
  filling in（填充动作表格），127-129
  non-error entries（动作表格的非错误条目），128
  terminal symbols（动作表格中的终结符号），144
  See also LR(1) tables
actions（动作）
  accept（接受动作），117
  in accepting states（接受状态中的动作），65
  ad hoc（特定动作），115
  communicating between（动作之间的交流），190-191
  coordinating series of（协调一系列动作），201
  error（错误动作），117

placement of (动作放置), 67
reduce (动作简化), 117
scanner generator and (扫描器生成器和), 66
shift (移位动作), 117
specifying (描述动作), 65-67
syntax-directed (语法制导动作), 190
table-filling (表格填充动作), 128
activation (活动), 254
activation record pointers(ARPs) (活动记录指针), 17, 262-264
callee (被调用者活动记录指针), 263, 276, 289
caller (调用者活动记录指针), 263, 283
defined (活动记录指针定义), 262
as starting point (作为开始点的活动记录指针), 263
activation records(ARs) (活动记录), 262-267
allocating (活动记录分配), 265-267, 288-289
coalescing (活动记录接合), 267
combining (活动记录合并), 267
defined (活动记录定义), 262
heap allocation of (活动记录的堆分配), 266-267, 288-289
illustrated (插图说明的活动记录), 263
local storage (局部存储的活动记录), 263-265
stack allocation of (活动记录的栈分配), 265-266, 288
static allocation of (活动记录的静态分配), 267
See also name spaces
ad hoc syntax-directed translation (专用语法制导翻译), 188-202
ASTs building (AST构建的专用语法制导翻译), 196-197
communication between actions (动作间交流的专用语法制导翻译), 190-191
declaration processing (声明过程的专用语法制导翻译), 198-202
defined (专用语法制导翻译的定义), 188
examples (专用语法制导翻译的例子), 193-202
ILOC generation (ILOC生成的专用语法制导翻译), 197-198
implementing (实现专用语法制导翻译), 190-193
for inferring expression types (推断表达式类型的专用语法制导翻译), 195
load tracking with (带有专用语法制导翻译的装入跟踪), 193-195
naming values (命名值的专用语法制导翻译), 191-192
for signed binary numbers (带符号二进制数的专用语法制导翻译), 189
snippets (专用语法制导翻译方案中的代码片段), 189
`addI` operation (专用语法制导翻译中的`addI`操作), 529
address space (地址空间)
layout (地址空间布局), 291
logical (逻辑地址空间), 291
utilization (利用地址空间), 291
views (地址空间的观点), 292
addressability (可寻址性), 280-285
addresses (地址)
calculation cost (地址计算的代价), 283
calculation form (地址计算的形式), 391-392
directly computing (直接计算地址), 367
logical, mapping (地址逻辑, 映射), 291-292
receiver (接受者地址), 370
run-time (运行时地址), 281
trivial base (平凡基础地址), 280-281
address-immediate operations (地址立即操作), 664
address-offset operations (地址偏移操作), 664
Algol-like languages (Algol类语言)
name spaces in (Algol类语言中的名字空间), 257-259
object-oriented languages vs. (面向对象语言和Algol语言), 274-275
alias pairs (别名对), 486
alignment rules (对齐规则), 293
`allocate` routine (分配例程), 296, 297, 298-299
allocation. *See* register allocation
allocators (分配器)
arena-based (基于区域分配器), 298
bottom-up local register (自底向上局部寄存器分配器), 626-629
first-fit (首次拟合分配器), 296-298
garbage-collecting (垃圾回收分配器), 298
global (全局分配器), 641
multipool (多池分配器), 298-299
register (寄存器分配器), 316, 620, 626-629, 650-651
alternation (选择)
regular expressions (选择正则表达式), 37
transformations, applying (选择转换, 运用), 48-49
ambiguity (歧义性)
context-sensitive (上下文敏感的歧义性), 136-138
grammar (歧义性文法), 132
`if-then-else` (歧义的`if-then-else`), 81-82, 121
removal (歧义性消除), 82
ambiguous branches (歧义性的分支), 505
ambiguous grammars (歧义性的文法), 81

ambiguity removal（歧义性文法的歧义性消除），82
example（歧义性文法例子），81-82
if-then-else（歧义性文法的if-then-else），82，83
reduce-reduce conflict（歧义性文法归约冲突），133
table-filling algorithm and（表格填充和歧义性文法），132
ambiguous memory references（歧义性内存引用），486
ambiguous pointers（歧义性指针），396
ambiguous values（歧义性值），312，656-657
analysis（歧义性值分析），312
heavy use code（歧义性值频繁使用代码），656
in memory（内存中的歧义性值），312
analysis（分析），15-16
context-sensitive（上下文敏感分析），12，28
control-flow（控制流分析），484
data-flow（数据流分析），15，417，433-490
dependence（相关分析），15-16
dynamic（动态分析），436
interprocedural（过程内分析），362，434，483-488
in larger scopes（较大作用域内分析），424
local（局部分析），434
pointer（指针分析），396，449，486
recompilation（重编译分析），487-488
semantic（语义的分析），12
static（静态的分析），433，434，436
very busy（非常忙碌分析），451-452
anonymous values（匿名值），350
loading（装入匿名值），351-352
storing（存储匿名值），351-352
anticipability（可前置性），506
anticipable expressions（可前置表达式），509-514
earliest placement（可前置表达式最早放置），506，510-511
later placement（可前置表达式的后放置），506，511-512
antidependencies（反相关性）
avoiding（避免反相关性），595
breaking（破坏反相关性），604
constraints（反相关性的限制），592
defined（反相关性定义），591
introduction of（反相关性的引入），594
respecting（反映反相关性），592
arena-based allocation（基于实块分配），298
arithmetic operations（算术操作），662-663
arithmetic operators（算术操作符），9，313-322
assignment（算术操作符赋值），321
associativity（算术操作符的可结合性），321-322
commutativity（算术操作符的可交换性），321-322
number systems（算术操作符的数制），321-322
array elements（数组元素）
accessing（存取数组元素），338
referencing（引用数组元素），337-343
storage location（数组元素的存储位置），337
arrays（数组），162-163
address calculations for（数组的地址计算），337，340，390
address polynomials（数组的寻址多项式），340
bounds（数组边界），335，339
column-major order（列优先顺序使用数组），335，336
as constructed type（作为结构类型的数组），162
defined（数组定义），162
FORTRAN-style（FORTRAN风格），684-685
indirection vectors（间接向量），335，336-337
operations on，support for（支持数组上的操作），162
passing，as parameters（作为参数传输的数组），341
passing，by reference（以引用传输的数组），278-279
row-major order（行优先顺序使用数组），335，336
size determination（数组大小确定），341
storage layout（数组存储布局），335-337
string type vs.（串类型和数组），163
of structures（数组结构），353-354
subscripts（数组下标），352
two-dimensional，address computation（二维数组地址计算），340
See also vectors
array-valued parameters（数组值参数）
accessing（存取数组值参数），341-343
range checks for（数组值参数的范围），343
assignment（赋值）
allocation vs.（分配与赋值），622-623
boolean（布尔赋值），332
inside expressions（表达式内部的赋值），85
integer（整数赋值），168
lvalue（左值赋值），321
as operator（作为操作符赋值），321
pointer（指针赋值），396-398
register（寄存器赋值），594，622-623
rvalue（右值赋值），321

string（串赋值），345-349
associativity（结合性），139-140，321-322
changing（更改结合性），204-206
left（左结合性），139，140
operators（操作符的结合性），321-322
recursion vs.（左递归与结合性），205
for reordering expressions（重排序表达式的结合性），397
reversing（保持结合性），206
right（右结合性），139，140
attribute grammars（属性文法），171-188
advocates（属性文法的倡导者），188
building（构建属性文法），172
circular（循环属性文法），174，176-177
contents（属性文法的内容），171
creating/using（创建或使用属性文法），175
defined（定义属性文法），171
directing flow of（指引属性文法流），206
dynamic methods（关于属性文法的动态方法），175
evaluation methods（属性文法的评估方法），175-176
for execution-time estimator（执行时间评估器的属性文法），179-180
extended examples（属性文法的扩展例子），177-185
framework（属性文法的框架），171-188
handling nonlocal information（处理非局部信息的属性文法），185-186
to infer expression types（推断表达式类型的属性文法），177-179
instantiating parse tree（属性文法实例化分析树），186-187
oblivious methods（属性文法的遗忘方法），176
paradigm simplicity（属性文法范式的简洁性），188
problems with（属性文法的问题），185-188
rule-based evaluators（基于规则的评估器的属性文法），188
rule-based methods（基于规则的方法的属性文法），176
S-attributed（S-属性文法），178，195
storage management（属性文法的存储管理），186
strongly noncircular（强非循环属性文法），177
See also context-sensitive analysis
attributed parse trees（属性分析树）
analysis results（属性分析树的分析结果），187
evaluating（评估属性分析树），172
for signed binary number（带符号二进制数的属性分析树），174
See also parse trees
attribute-dependence graph（属性相关图），172，175
cyclic（循环属性相关图），176
formation（属性相关图的格式），172
attributes（属性）
association with nodes（与结点相关的属性），191
central repository for（属性中心存储室），187
for each grammar symbol（每种文法符号的属性），175
evaluating（评估属性），174
inherited（继承属性），173
large number of（大量属性），186
of node parent，siblings，children（父，兄和孩子结点的属性），182
synthesized（合成属性），173
attribution rules（属性规则），175，179，185
AVAIL（AVAIL）
defined（定义AVAIL），419
LiveOUT vs.（LiveOUT与AVAIL），438-439
sets，computing（计算AVAIL集合），419-420
available expressions（可用表达式），404
code speed and（代码速度和可用表达式），418
computing（计算可用表达式），419-421，508-509
defined（定义可用表达式），417-418
as forward data-flow problem（作为前向数据流问题的可用表达式），450
as global data-flow analysis problem（作为全局数据流分析问题的可用表达式），421
iterative algorithm for（可用表达式的迭代算法），421

B

back end（后端）
defined（定义后端），5
generator（后端生成器），551
processes（处理后端），549
retargetable（可重新设置目标的后端），550
See also compilers; front end
backtracking（回溯），92
additional，triggering（引发多余的回溯），93
eliminating need for（消除对回溯的需求），97-101
Backus-Naur form（Backus-Naur格式），74-75
backward list scheduling（向后列表调度），600-603
defined（定义向后列表调度），601
forward list scheduling vs.（前向列表调度与向后列表调度），601

operation order and (向后列表调度顺序和), 601
priority (向后列表调度的优先级), 602
schedules (调度向后列表调度), 602
See also list scheduling
base types (基本类型), 160-161
Booleans (布尔基本类型), 161
characters (字符基本类型), 161
definitions (定义基本类型), 160
numbers (数基本类型), 160-161
support (支持基本类型), 160
See also type systems;type(s)
batch collectors (批回收器), 301
defined (定义批回收器), 301
phases (阶段批回收器), 301
technique comparison (批回收器技术比较), 304
See also garbage collection
BCPL (BCPL)
cells (BCPL单元), 156
`switch` construct (`switch`结构), 364
benchmarking (基准测试), 615
biased coloring (偏着色), 652-653
binary search (二分搜索), 365-366, 687
binary trees (二叉树), 681-682
bit vectors (位向量), 677
blocks (块)
ϕ-function (块ϕ函数), 464, 478
boundary (块边界), 604
boundary complications (块边界处的复杂性), 631-633
cloning (块复制), 426-427
combining (结合块), 502
DVN processing (DVN处理块), 434
empty, removing (消除空块), 501
end, finding (寻找块的尾部), 440, 441
list scheduling application to (将列表调度运用于块), 595-596
multiple, problems (多个块的问题), 632
placement (放置块), 540
precedence graph (块的优先图), 589
state at end of (块的尾部状态), 469
unreachable (不可达块), 505
booleans (布尔), 322-333
adding, to expression grammar (把布尔加入表达式文法), 323
as base types (作为基本类型的布尔), 161
implementing (实现布尔), 329
numerical encoding (数编码布尔), 324-325
pointer to (指向布尔的指针), 165
positional encoding (位置编码布尔), 325-327
representations (表示布尔), 323-328
boolean-valued comparisons (布尔值比较), 331
bottom-up coloring (自底向上着色), 644-646
computing (计算自底向上着色), 645
functioning of (自底向上着色的功能), 645-646
live range ordering (自底向上着色排序活区域), 644
removal process (自底向上着色消除过程), 645
spill metric (使用溢出度量的自底向上着色), 646
See also global register allocation
bottom-up local register allocators (自底向上局部寄存器分配器), 626-629
defined (定义自底向上局部寄存器分配器), 626
illustrated (图解自底向上局部寄存器分配器), 626
net effect (自底向上局部寄存器分配器的网络效应), 628
top-down local allocator vs. (自底向上局部寄存器分配器与自顶向下局部分配器), 629
See also register allocation;register allocators
bottom-up parsers (自底向上语法分析器), 107-120
defined (定义自底向上语法分析器), 87
derivations (自底向上语法分析器中的派生), 107
functioning of (自底向上语法分析器的功能), 107
reduction (自底向上语法分析器中的归约), 141
shift-reduce (自底向上语法分析器中的移位归约), 190
states (自底向上语法分析器的状态), 110
bottom-up parsing (自底向上语法分析), 74, 107-120
critical step in (自底向上语法分析中的关键步骤), 108-109
failed (失败的自底向上语法分析), 107
handle-finding (自底向上语法分析中的发现句柄), 112-115
partial parse trees for (自底向上语法分析的部分分析树), 111
shift-reducing (自底向上语法分析中的移位归约), 108-112
successful (成功的自底向上语法分析), 107
unambiguous grammar and (非歧义性文法和自底向上语法分析), 108
branches (分支)
ambiguous (歧义分支), 505
conditional (条件分支), 328, 668
fall-through (落下分支), 553

hoisting（提升分支），502
prediction（判断分支），358
redundant，folding（叠入冗余分支），501
break statement（break语句），368
bucket hashing（桶散列），689-691
adding lexical scopes to（把词法作用域加到桶散列中），695-696
advantages（桶散列的优点），690
assumptions（桶散列假设），689
defined（定义桶散列），686
drawbacks（桶散列的缺点），690-691
table（桶散列表），690

C

for loop（for循环），359，360-361
scoping rules（作用域规则），259-260
switch construct（switch结构），364
cache memories（高速缓存），293-295
block sharing（高速缓存的块分享），295
defined（定义高速缓存），294
frames（高速缓存的帧），294
mappings（高速缓存的映射），294
multiple levels（多级高速缓存），599
primer（高速缓存的初步），294
virtually addressed（虚拟地址高速缓存），295
See also memory
call by name（名字调用），277
call by reference（引用调用），276-279
call by value vs.（值调用与引用调用），277-278
defined（定义引用调用），276
example（引用调用的例子），278
parameter（引用调用参数），318，319
space requirements（引用调用空间需求），278
call by value（值调用），275-276
call by reference vs.（引用调用与值调用），277-278
defined（定义值调用），276
parameters（值调用参数），370
results（值调用结果），276
space requirements（值调用空间需求），278
call graphs（调用图），254，484
distinct edge in（调用图中的不同边），484
graphs derived from（从调用图得到的图），484
callee saves（被调用者保存），288
callee-saves registers（被调用者保存的寄存器），288，371
combining with caller-saves（与被调用者保存的寄存器的结合），372
preserving（保持被调用者保存的寄存器），371
caller-saves registers（被调用者保存的寄存器），371，372
calling sequence（调用序列），252
candidate operations（候选操作），530
canonical collection of sets of LR(1) items（LR(1)项目集合的规范集合），120
algorithm（LR(1)项目集合的规范集合的算法），124-125
closure procedure（LR(1)项目集合的规范集合的closure过程），122-123
constructing（构造LR(1)项目集合的规范集合），122-127
goto procedure（LR(1)项目集合的规范集合的goto过程），124
for *SN*（*SN*的LR(1)项目集合的规范集合），125-127
states of handle-recognizing DFA（LR(1)项目集合的规范集合代表句柄识别DFA的状态），126
case statement（选择语句），364-368
binary search（选择语句的二分搜索），365-366
direct address computation（选择语句的直接地址计算），367
implementing（实现选择语句），364-368
linear search（选择语句的线性搜索），365
while loop in（选择语句内的while循环），367
central repository（中心存储库），187
characteristic vectors（字符向量），677
characters（字符）
as base types（作为基本类型的字符），161
escape（转义字符），40
extra memory references per（每个字符的额外内存引用），62
processing（处理字符），67
circular attribute grammars（循环属性文法），174，176-177
approaches（使用循环属性文法的方法），176-177
defined（定义循环属性文法），174
See also attribute grammars
CISC machines，instruction selection and（复杂指令集计算机（CISC）机器，指令筛选和），579
class variables（类变量），262，271，380
classes（类）
creating（创建分类），375
defined（定义分类），269，271
register（寄存器类），312，623-624
single，no inheritance（单一类，无继承），373-375

storage（类存储），568
structure（类结构），376
superclasses（超类），271
classic expression grammar（经典表达式文法），85
Action table for（经典表达式文法的action表格），118
defined（定义经典表达式文法），85
Goto table for（经典表达式文法的Goto表格），119
illustrated（图解经典表达式文法），85
parse tree using（使用经典表达式文法的分析树），214
parser for（经典表达式文法的语法分析器），86
right-recursive variant（经典表达式文法的右递归变量），95
See also grammars
Clean algorithm（Clean算法）
Dead cooperation with（Clean算法与Dead的合作），504
defined（定义Clean算法），501
illustrated（图解Clean算法），503
transformation application（Clean算法的转换运用），503
transformation illustration（Clean算法的转换说明），502
transformations（Clean算法的转换），501-502
clean values（净值），628
cloning（复制）
blocks（复制块），426-427
for context（为上下文复制），614-617
costs（复制代价），427
example illustration（复制例子说明），426
to increase context（通过复制增加上下文），425-427
for increasing scheduling context（为增加调度的上下文的复制），616
tail call（复制一个尾调用），616
closure（闭包）
Kleene（Kleene闭包），37，38
positive（正闭包），38
precedence（优先闭包），39
regular expressions（闭包的正则表达式），38
transformations，applying（运用闭包转换），48-49
under concatenation（连接下的闭包，封闭性），42
closure procedure（closure过程），122-123，128
coalescing live ranges（接合活区域），646-647
defined（定义接合活区域），646
illustrated（图解接合活区域），647
performing（执行接合活区域），647
two（接合两个活区域），647
See also live ranges
code（代码）
better，generating（生成较好代码），157-159
compensation（补偿代码），606
control-flow construct implementation（控制流结构实现代码），356
defined（定义代码），20
hoisting（提升代码），452，506，514-515
improving（改进代码），15-16
layout（代码结构），553
pipelining（流水线代码），611
smaller，generating（生成较小代码），538-539
stack-machine（栈机器代码），223，224
templates（模板代码），564
three-address（三地址代码），223，224-225
three-register（三寄存器代码），316
transition diagrams（转换图表代码），31
two-register（二寄存器代码），316
useless，eliminating（消除无用代码），495，498-505
writing（编写代码），9
code generation（代码生成），16-23
compilation problems during（代码生成期间的复杂问题），21
instruction scheduling（代码生成的指令调度），19-21
instruction selection（代码生成的指令筛选），16-18
interactions（代码生成的相互作用），21-22
optimal（代码生成的优化），557
register allocation（代码生成的寄存器分配），18-19
See also instruction selection
code generators（代码生成器），554，555
designing（设计代码生成器），545
efficient/effective（代码生成器的效率），569
implementation（实现代码生成器），568
lower-cost sequences and（较低代价序列和代码生成器），565
multiple addressing modes and（多寻址模型和代码生成器），546-547
tiling selection（代码生成器的铺盖选择），564
tree-walk（代码生成的树遍历），318，562
code motion（代码移动），496，505-515
hoisting use of（代码移动的提升使用），506
lazy（惰性代码移动），505，506-514

code optimization（代码优化），15，383-432
background（代码优化的背景），386-392
considerations（代码优化的考虑），388-392
context，increasing（增加上下文的代码优化），424-427
goal（代码优化的目的），385
improvement forms（多种改良形式代码优化），385
introduction to（介绍代码优化），383-385
levels of（代码优化层次），541
objectives（代码优化目标），538-540
opportunities（代码优化机会），383，392
performance gap（代码优化所带来的性能的差异），386
performance improvement（代码优化的执行改进），430
procedure calls and（过程调用和代码优化），427
profitability and（有效性和代码优化），390-391
repeating（重复代码优化），541
risk and（风险和代码优化），391-392
safety and（安全性和代码优化），388-390
scope（作用域和代码优化），404-408
selecting（选择代码优化），541
sequence，choosing（选择代码优化序列），540-541
See also optimization
code reuse（代码复用），269
data-flow frameworks and（数据流框架和代码复用），452
inheritance and（继承和代码复用），271
code sequences for matches（匹配的代码序列），564
code shape（代码形态），307-382
alternate（不同的代码形态），308
decisions（确定代码形态），315
defined（定义代码形态），307
examples（代码形态的例子），308-309
impact（代码形态的影响），307
memory model impact on（内存模型对代码形态的影响），292-293
memory-to-memory model and（内存到内存模型和代码形态），292
register-to-register model and（寄存器到寄存器模型和代码形态），292-293
code-space locality（代码空间的位置），104
column-major order（列优先顺序），335，336
commutative operations（可交换操作），400
commutativity（可交换性），321-322
operations（可交换性操作），321
for reordering expressions（重排序表达式的可交换性），397
comparison operator（比较操作符），668-669
compensation code（补偿代码），606
compilation（编译）
high-level view（编译的高级观点），8-23
offline nature of（编译的脱机属性），674
overview（编译总阅），1-26
principles（编译原理），4-5
problems during code generation（代码生成期间的编译问题），21
separate（分块编译），251
techniques（编译技术），22
compilers（编译器）
back end（编译器后端），5，549，550，551
construction（构造编译器），3-4
debugging（调试编译器），24，381，386
defined（定义编译器），1
desirable properties（编译器的理想性质），23-24
efficiency（高效编译器），24
as engineered objects（作为工程对象的编译器），4，10
feedback（编译器的反馈），23-24
front end（编译器的前端），5，25，578
fundamental role of（编译器的基础角色），3
GCC（GCC编译器），226
interpreters vs.（翻译和编译器），2-3
multipass（多遍编译器），228
Open64（Open64编译器），226
optimizing（优化编译器），7，386
PCC（PCC编译器），226
PL8（PL8编译器），226
principles（编译器原理），4-5
research（编译器研究），2
retargetable（可重定目标编译器），548-552
source-to-source translators（编译器是源语言到源语言翻译器），2
space（编译器的空间），23
speed（编译器的速度），23
structure（编译器的结构），5-8
successful（成功的编译器），4
target program（编译器到目标程序），2
three-phase（编译器的三阶段），6-8
compile-time efficiency（编译时效率），24
compound objects（复合对象），151-152
compound types（复合类型），162-166

computations (计算)
availability (可用性计算), 506
costs (计算代价), 568
direct address (直接地址计算), 367
fixed-point (不动点计算), 54, 55, 125
pointer-based (基于指针的计算), 166
priority (计算的优先级), 598
redundant, eliminating (冗余计算消除), 496
specializing (计算特化), 495-496
concatenation (连接)
closure under (连接下的封闭性), 42
regular expressions (正则表达式的连接), 38
string (串的连接), 349
transformations, applying (连接转换的运用), 48-49
conditional branches (条件分支), 328
operation (条件分支操作), 668
two-label (两标签条件分支), 668
conditional execution (条件执行), 356-359
conditional move (条件移动), 330-331
advantage (条件移动的优势), 331
example (条件移动示例), 330
execution (条件移动执行), 331
condition-code registers (条件代码寄存器), 624
conservative coalescing (保守接合), 652-653
constant function types (常量函数类型), 203
constant propagation (常量传播), 452-454
code motion (常量传播中的代码移动), 496
global (全局常量传播), 453
interprocedural (过程内常量传播), 485-486
reformulating (重新形式化常量传播), 516
sparse simple(SSCP) (稀疏简单常量传播), 516-518
CONSTANTS sets (CONSTANTS集合)
for code improvement (代码改进CONSTANTS集合), 454
defining (CONSTANTS集合定义), 453
domain (CONSTANTS集合的定义域), 454
constructed types (结构类型), 162-166
context-free grammars (上下文无关文法), 75-79
backtracking and (回溯和上下文无关文法), 90
construct recognition (上下文法的结构识别), 89
defined (上下文无关文法的定义), 75
example (上下文无关文法的示例), 75
languages defined by (由上下文无关文法定义的语言), 75
microsyntax in (上下文无关文法中的微语法), 89
quadruple (四元组上下文无关文法), 78
regular expressions vs. (正则表达式与上下文无关文法), 87-89
as rewrite system (作为重写系统的上下文无关文法), 79
scanners and (扫描器和上下文无关文法), 89
sentence construction (上下文无关文法中的语句构造), 79-83
strings in (上下文无关文法中的串), 172
subclasses (上下文无关文法中的子类), 75
subsets (上下文无关文法的子集), 89, 90
See also grammars
contexts (上下文)
cloning for (上下文的复制), 614-617
increasing (上下文的增加), 425-427
left (左上下文), 122
right (右上下文), 70, 122
scanning (上下文的扫描), 69-70
context-sensitive ambiguity (上下文相关的歧义性), 136-138
context-sensitive analysis (上下文相关分析), 12, 151-208
ad hoc syntax-directed translation (专用语法导制的翻译的上下文相关分析), 188-202
attribute grammars (属性文法的上下文相关分析), 171-188
defined (上下文相关分析的定义), 28, 153
type systems and (类型系统和上下文相关分析), 154-171
context-sensitive grammars (上下文相关文法), 201
control abstraction (控制抽象), 254-256
control-flow analysis (控制流分析), 484
control-flow constructs (控制流结构), 355-368
`break` statement (控制流结构中的`break`语句), 368
case statement (控制流结构中的选择语句), 364-368
conditional execution (控制流结构的条件执行), 356-359
implementation code (实现控制流结构的代码), 356
loops and iteration (控制流结构中的循环和迭代), 359-364
control-flow graphs(CFGs) (控制流图), 219-220
back edges (控制流图的后边), 503
basic blocks (控制流图中的基本块), 219-220
building (控制流图的构建), 438, 439-440
construction complications (控制流图构造的复杂性), 440-441
defined (控制流图的定义), 219

edges（控制流图的边），440
entry nodes（控制流图的入口结点），219
example（控制流图的示例），683
execution frequencies and（执行频率和控制流图），638
graphical representation（控制流图的图表示），219
illustrated（图解控制流图），220
implementation tradeoff（控制流图实现的权衡），219
irreducible（不可归约控制流图），480，481，483
reducible（可归约控制流图），480，481
successors（控制流图的后继），472
from target-machine code（目标机器代码的控制流图），440
for trail-recursive routine（尾递归例程的控制流图），616
uses（控制流图的使用），220
control-flow operations（控制流操作），576-577，667-670
predicated（预测控制流操作），576-577
presence of（控制流操作的存在），576
conversion algorithm（转换算法），66
copy folding（拷贝叠入），478
copy operations（拷贝操作），403，656
copy rules（拷贝规则）
attribute grammar size and（属性文法有大小的拷贝规则），185
defined（拷贝规则的定义），182
to track loads（跟踪装入的拷贝规则），183
copying collectors（复制回收器），301，303-304
defined（定义复制回收器），303
generational（生成的复制回收器），304
stop and copy（停止和拷贝复制回收器），303
technique comparison（复制回收器的技术比较），304
See also garbage collection
copy-insertion algorithms（拷贝插入算法），478
correctness（正确性），4
importance（重要的正确性），389
LiveOUT（LiveOUT的正确性），445
syntactic（语法的正确性），12
cost functions（代价函数），568
critical edges（临界边）
defined（临界边的定义），475
not splitting（不分离临界边），476
splitting（分离临界边），475，479
unsplit（不分离临界边），478
critical paths（关键路径），591
execution（关键路径的执行），611
execution time and（执行时间和关键路径），591
cycle of constructions（构造法的循环关系），44-45
consequences（构造法的循环关系的结论），47
illustrated（图解构造法的循环关系），45

D

data areas（数据区），309
constraints（数据区的限制），310
global variables（全局变量数据区），290-291，310
laying out（数据区布局），310
static variables（静态变量数据区），290-291，310
data structures（数据结构），673-701
doms（doms数据结构），674
hash table implementation（散列表实现数据结构），686-698
IR implementation（数据结构的IR实现），679-686
representing sets（数据结构的表示集合），674-679
sizes needed for（数据结构所需要的大小），673
data-flow analysis（数据流分析），417，433-490
assumptions（数据流分析假设），448
defined（定义数据流分析），15
global problem（数据流分析的全局问题），421
imprecision（数据流分析的不精确性），448
insights（观察数据流分析），491
iterative（迭代的数据流分析），435-454
limitations on（数据流分析的限制），447-450
live variables（活变量），435-444
naming sets in（数据流分析中的命名集合），449
precision limitation（数据流分析的精度限制），448
procedure calls and（过程调用和数据流分析），450
scope，extending（扩展数据流分析作用域），479
sets（数据流分析集合），700
SSA form（数据流分析的SSA形式），454-479
data-flow frameworks（数据流框架），491
code reuse and（数据流框架和代码复用），452
constants found by（数据流框架所发现的常量），517
implementing（数据流框架的实现），452
data-flow problems（数据流问题），450-454
available expressions（数据流问题的可用表达式），450
constant propagation（常量传播数据流问题），452-454
global，equations for（数据流问题的全局方程），452
reaching definitions（数据流问题的可达定义），450-451

very busy expressions（非常忙碌表达式数据流问题），451-452
dead values（死亡值），575-576
example（死亡值示例），575
frequently（频繁死亡值），575
See also values
dead variables（死亡变量），575
dead-code elimination（死亡代码消除），422，493
deallocation，implicit（隐式释放），299-305
debugging（调试），24，229
debugging compilers（调试编译器），24，381，386
defined（调试编译器的定义），386
optimizing compilers vs.（调试编译器和优化编译器），386
declarations（声明）
omission（声明忽略），202
processing（声明处理），198-202
syntax（语法声明），199，200
"declare before use" rule（"使用前声明"规则），153
definition points（定义点），221
delay（等待），597
best-case（最好情况等待），599
memory operation（内存操作等待），599
slots（槽等待），441
worst-case（最坏情况等待），599
deletion sets（删除集合），507
Delta operations（Delta操作），54
dependence analysis（相关性分析），15-16
dependence graphs（依赖图），221-222
building（依赖图构建），595
complexity（依赖图的复杂性），222
control flow interaction（控制流与依赖图之间互相作用），222
defined（依赖图的定义），221，589
edges（依赖图的边），221
example（依赖图的示例），590
illustrated（图解依赖图），221，602
node mapping（依赖图的结点映射），589
property capture（依赖图的性质刻画），591
roots（依赖图根），589
uses（依赖图的使用），222
derivations（派生）
bottom-up parser（自底向上语法分析器中的派生），107
constructing（派生构造），86
example sentence（派生例句），11
graphic depiction（派生图描述），80
leftmost（最左派生），81，91
order（派生顺序），108
parse tree equivalent（分析树等价派生），86
rightmost（最右派生），81，82，108
specific，discovering（特定派生发现），86-87
tabular form（表格形式），77
deterministic finite automata(DFAs)（确定性有穷自动机），44
with cyclic transition graph（转换图带有循环的DFA），59
defined（DFA的定义），46
deriving regular expression from（从DFA得到正则表达式），59
equivalent NFA（与NFA等价的DFA），47
handle-recognizing（句柄识别DFA），114
language acceptance by（DFA接受的语言），44
minimization algorithm（DFA最小化算法），56
minimizing（DFA最小化），56
NFA simulation（与NFA的相似性），60
numbered states（DFA的已编号状态），59-60
predictive parsers vs.（预测语法分析器和DFA），102
as recognizers（作为识别器的DFA），60-61
regular expression construction from（从DFA构造正则表达式），59-60
scanner implementation with（DFA与扫描器实现），62
seven-stage（七状态DFA），63
subset construction（DFA的子集构造），44，53，55-56
table generation from（从DFA的表格生成），63
table-driven implementation（DFA的表驱动实现），62
transition diagram（DFA的转换范式），59
transition tables（DFA转换表格），558
`detour` operator（`detour`操作符），665
direct-coded scanners（直接编码扫描器），64
directed acyclic graphs (DAGs)（有向无环图），216-217
defined（有向无环图的定义），216-217
hash-based constructor（有向无环图的基于散列的构造器），396，397
nodes（有向无环图的结点），217，394
nodes with multiple parents（带有多个父结点的有向无环图结点），394
redundancy elimination with（有向无环图和冗余消除），394-398
uses（有向无环图的使用），217

dirty values（非净值），628，629
disjoint-set union-find algorithm（不相交集合并集寻找算法），636
dispatching（分派），273
displays（显示）
  defined（显示的定义），283
  global（全局显示），283-285
  managing（映射显示），290
distributivity（分配性），397
do loop（do循环），361-362
  defined（do循环定义），361
  form（do循环的形式），361
  index variables（do循环的索引变量），362
DOM（DOM）
  finding（DOM寻找），460
  sets（DOM集合），459
  sets，intersecting（插入DOM集合），460
dominance（支配），457-463
  defined（支配定义），457
  for LiveOUT example（LiveOUT示例的支配），458
dominance frontiers（支配边界），457，460-463
  computing（计算支配），461-462
  defined（支配边界的定义），461
  example（支配边界的示例），462-463
  results（支配边界的结果），463
  reverse（反向支配边界），500
dominator trees（支配者树），416，459
  defined（支配者树的定义），416
  depth（支配者树的深度），474
  illustrated（图解支配者树），416，458
  preorder walk over（前序遍历支配者树），671
dominator-based value numbering（基于支配者的值编号），413-417
  block processing（基于支配者的值编号的块处理），434
  dominators（基于支配者的值编号中的支配者），415-416
  facts collection（基于支配者的值编号建立的事实集合），491
  IDom（基于支配者的值编号中的立即支配者），415，416-417
  See also value numbering
dominators（支配者），415-416
  computing（计算支配者），457-459
  relationship（支配者关系），415
dope vectors（内情向量），342-343
  defined（内情向量的定义），342
  illustrated（图解内情向量），342
dynamic analysis（动态分析），436
dynamic method lookup（动态方法查找），377
dynamic register renaming（动态寄存器的重命名），604
dynamic scoping（动态作用域），261
dynamically checked languages（动态检查语言），169
dynamically scheduled machines（动态调度机器），604
dynamically typed languages（动态类型语言），156，169

E

earliest placement（最早放置），506，510-511
  computing（计算最早放置），510
  equations（最早放置方程），510-511
  See also anticipable expressions;later placements
efficiency（效率）
  duplication and（重复和效率），464
  improving（效率改进），459-460
  list scheduling（列表调度的效率），598-599，604
  LiveOUT（LiveOUT的效率），445-447
enabling transformations（激活转换），496，518-522
  defined（激活转换的定义），518
  loop unrolling（循环展开激活转换），519-520
  loop unswitching（循环反切换激活转换），520-521
  renaming（重命名激活转换），521-522
  types of（激活转换的类型），518
  See also transformations
encoding（编码）
  naive（naive编码），325
  numerical（数字编码），324-325
  positional（位置编码），325-327
  recognizers into table set（识别器编码成一组表格），35-36
  syntactic structure（编码语法结构），78
enumerated types（枚举类型），163-164
epilogue sequences（结语序列），286，287
error(s)（错误）
  action（错误动作），117
  detection（错误发现），110
  recovery（错误恢复），134-135
  syntax（语法错误），110，134
  type（错误类型），167-168
escape characters（转换字符），40
evaluated parse trees（评估分析树），172
evaluator generator（评估器生成器），175

execution history（执行历史），254
execution-time estimator（执行时间估测器），179-180
attribute grammar（执行时间估测器的属性文法），180
final improvement to（对执行时间估测器的最后改进），184-185
improving（执行时间估测器的改进），180-183
exhaustive search（穷举搜索），582
explicit length field（显示长度字段），350
expressions（表达式）
anticipable（可前置表达式），509-514
available（可用表达式），404，417-418，419-421，504，508-509
boolean（布尔表达式），322
function calls in（表达式中的函数调用），319
ILOC generation fro（表达式的ILOC生成器），197-198
mixed-type（混合型表达式），320
names（表达式名字），419
operands（表达式操作数），231
predicate（谓词表达式），332
redundant（冗余表达式），393-404
relational（关系表达式），322
reordering（表达式重排序），397
tree-walk for（表达式的树遍历），314
trivial，code generation for（表达式的平凡代码生成），313
type inference for（表达式的类型引用），168-170，177-179，195
very busy（非常忙碌表达式），451-452
expressiveness（表示能力）
improving（改进表示能力），156-157
IR（IR表示能力），212-213
extended basic blocks (EBBs)（扩展的基本块），405
local technique extension to（局部技术到扩展的基本块的推广），413
maximal-size（极大扩展的基本块），405
scheduling（扩展的基本块的调度），605-607，608
scheduling paths through（通过扩展的基本块的调度路径），607
tree structure（扩展的基本块的树结构），408
trivial（平凡的扩展的基本块），605
external interfaces（外部接口），253

F

fall-through branch（落下分支），553
false zero（伪零），338
computing（伪零计算），340
defined（伪零定义），334
feedback（反馈），23-24
fields（字段），270
finite automata (FA)（有穷自动机），31-33
defined（有穷自动机定义），27，31
deterministic (DFAs)（确定性有穷自动机），44
encountering errors（有穷自动机遇见错误），33
five-tuple（五元组有穷自动机），31
formalism（有穷自动机的形式定义），32
initial state（有穷自动机的初始状态），32
merging（合并有穷自动机），46
nondeterministic (NFAs)（非确定性有穷自动机），44，45-50
number of states（有穷自动机的状态数量），42
operation cost（有穷自动机的操作代价），41
power（有穷自动机的威力），37
regular expressions and（正则表达式和有穷自动机），36，44
for signed integers（带符号整数的有穷自动机），35
simplifying（简化有穷自动机），34
string acceptance（有穷自动机的可接受串），32
for unsigned integers（无符号整数的有穷自动机），37
first-fit allocation（首次拟合分配），296-298
defined（首次拟合分配的定义），296
free list（首次拟合分配的自由列表），296
gola（首次拟合分配的目标），296
variations（首次拟合分配的变形），297-298
fixed-point computations（不动点计算），54，125
å-closure（å-closure的不动点计算），55
defined（不动点计算的定义），54
termination arguments（不动点计算的终止参数），54
floating-point numbers（浮点数），160
floating-point operations（浮点操作），547
floating-point registers（浮点寄存器），623
flow-insensitive methods（流非敏感方法），486
flow-sensitive methods（流敏感方法），486
fluff removal（错误清除），540
for loop（for循环），359，360-361
canonical shape（for循环的规范形态），361
code layout（for循环代码布局），360
mapping（for循环映射），360
two-block（两块的for循环），361
FORTRAN（FORTRAN）

arrays (FORTRAN的数组), 684-685
do loop (FORTRAN中的do循环), 361-362
properties (FORTRAN的性质), 67
right context and (右上下文和FORTRAN), 70
scanning (FORTRAN的扫描), 68
scoping rules (FORTRAN的作用域规则), 259
two-pass scanners (FORTRAN的两遍扫描器), 69-70
forward list scheduling (向前列表调度), 600-603
backward list scheduling vs. (向后列表调度与向前列表调度), 601
defined (向前列表调度的定义), 601
operation order and (操作顺序和向前列表调度), 601
priority (向前列表调度优先级), 601-602
See also list scheduling
forward substitution (向前替换), 579
frames, cache (高速缓冲帧), 294
free list (自由表), 296
free routine (free例程), 297, 299
free variables (自由变量), 261
front end (前端)
defined (前端定义), 5
focus (前端集中于), 25
LLIR generation (前端LLIR生成), 578
See also back end;compilers
function calls (函数调用), 319
functions (函数)
cost (函数代价), 568
defined (函数定义), 251
hash (散列函数), 688-689
jump (跳转函数), 485
returning values from (从函数的值返回), 279
side effects (函数的副作用), 319
trampoline (蹦床函数), 379-380
virtual (虚函数), 392

G

garbage collection (垃圾回收), 299-305
batch collectors (垃圾回收中的批回收器), 301
conservative collectors (垃圾回收中的保守回收器), 302-303
copying collectors (垃圾回收中的拷贝回收器), 303-304
defined (垃圾回收的定义), 299
mark-sweep collectors (标记清除回收器), 303
precise collectors (精确回收器的垃圾回收), 302
real-time collectors (实时回收器的垃圾回收), 305
reference counting (执行垃圾回收的引用计数), 300-301
techniques comparison (垃圾回收的技术比较), 304-305
work associated with (与垃圾回收相关的工作), 299-300
garbage-collecting allocators (垃圾回收分配器), 298
GCC compiler (GCC编译器), 226, 579
general-purpose registers (通用寄存器), 623
generational collectors (世代回收器), 304
global data-flow algorithms (全局数据流算法), 421
global display (全局显示), 283-285
access links vs. (存取链和全局显示), 285
current (当前全局显示), 285
defined (全局显示的定义), 283
maintenance (维护全局显示), 285
managing (管理全局显示), 290
overhead (全局显示的负担), 285
references through (通过全局显示的引用), 285
updates (更新全局显示), 285
use illustration (使用图示的全局显示), 284
global methods (全局方法), 407
global redundancy elimination (全局冗余消除), 417-424
algorithm comparison (全局冗余消除的算法比较), 423
available expression computation (可用表达式计算的全局冗余消除), 419-420
improvements (全局冗余消除的改进), 424
redundant computation replacement (取代冗余计算的全局冗余消除), 421-422
role of names (全局冗余消除中的名字作用), 418-419
global register allocation (全局寄存器分配), 633-651
bottom-up coloring (自底向上着色全局寄存器分配), 644-646
decisions (寄存器分配决策), 634
live ranges (全局寄存器分配中的活区域), 635-637
local allocation vs. (局部分配和全局寄存器分配), 633-634
spill cost estimation (全局寄存器分配的溢出代价评估), 637-638
top-down coloring (自顶向下着色全局寄存器分配), 642-643
See also register allocation

global spill costs（全局溢出代价），637-638
estimating（评估全局溢出代价），637-638
execution frequencies and（执行频率和全局溢出代价），638
infinite（无限全局溢出代价），638
negative（负全局溢出代价），638
See also spilling
global variables（全局变量）
access to（全局变量的存取），280
data areas（全局变量数据区），290-291，310
labeling（全局变量标签），281
goal symbols（目标符号），78，79
defined（目标符号定义），75
unique（惟一的目标符号），120
`goto` procedure（`goto`过程），124
`Goto` table（`Goto`表格），116，117，119
directly encoding（`Goto`表格的直接编码），145-147
"don't care" states（`Goto`表格中的"无关"状态），146
filling in（`Goto`表格填充），127-129
non-error entries（`Goto`表格中的非出错条目），128
See also LR(1) tables
grammar rules（文法规则），78
grammars（文法）
adding parentheses to（在文法中加入括号），85
ambiguous（歧义性文法），81，133
attribute（属性文法），171-188
backtrack-free（无回溯），94
classic expression（经典表达式文法），85
context-free（上下文无关文法），75-79
context-sensitive（上下文相关文法），201
defined（文法定义），10，75
engineering（设计方法），140
fixed（固定的文法），112
left-factoring（文法的左因子分解），101
left-linear（左线性文法），87
left-recursion（左递归文法），92
LL(1)（LL(1)文法），88，102
LR(1)（LR(1)文法），88
optimizing（文法优化），141-143
parser（语法分析器中的文法），70
for programming languages（程序设计语言的文法），10
reduced expression（缩小表达式文法），146
regular（正则文法），87，88
right-recursive expression（右递归表达式文法），97，98
shorter variations（文法的更短变形），141
shrinking（缩小文法），144-145
syntactic structure，encoding（语法结构文法编码），78
graph coloring（图着色），634-635
defined（图着色定义），634
dominant operations（图着色中的支配操作），684
illustrated（图解图着色），635
paradigm（图着色范例），634
variations（图着色变形），651-654
graphical IRs（图示IR），213-222
defined（图示IR定义），210
graphs（图示IR图），218-222
implementing（图示IR的实现），679-684
syntax-related trees（图示IR的语法相关树），213-218
See also intermediate representations(IRs)
graph-reduction process（图归约过程），480
graphs（图），218-222
arbitrary，representing（任意图表示），682-684
call（调用图），254
control-flow（控制流图），219-220，439-441，472，480-481
dependence（相关图），221-222，589
directed acyclic (DAGs)（有向无环图），216-217
interference（冲突），639-641，684
irreducible（不可归约图），480-483
precedence（优先图），589
tabular representation of（图的表格表示），700

H

handle-recognizing DFA（句柄识别DFA），114
for *SN*（*SN*的句柄识别DFA），127
states（句柄识别DFA的状态），126
handles（句柄）
complete（完整句柄），114
finding（寻找句柄），112-115
finite set of（句柄的有限集合），114
position fields（句柄的位置域），113
potential（可能的句柄），113，114
potential，set of（可能的句柄集合），114
representation（句柄表示），113
hash functions（散列函数），239，240，688-689
choosing（散列函数的选择），688，689
impact（散列函数的影响），689
multiplicative（乘法散列函数），688
poor（拙劣的散列函数），689

universal（通用散列函数），688
hash tables（散列表），239-241，534，687
defined（散列表定义），239-240
implementation（散列表的实现），240
index（散列索引），699
replacing（散列表取代），241
as representation for sparse graphs（作为稀疏图表示的散列表），240
two-dimensional（二维散列表），698-699
uses（使用散列表），240
See also symbol tables
hash-based constructor（基于散列的构造器），396，397
hashing（散列），241
bucket（桶散列），686，689-691
open（开放散列），686，689-691
perfect（完美散列），65
heap（堆）
defined（堆定义），295
explicitly managed（显式堆管理），295-296
first-fit allocation（首次拟合分配堆），296-298
management algorithms（管理堆算法），295-299
multipool allocators（堆的多池分配器），298-299
storage allocation（堆的存储分配），310
value lifetimes（堆中的值的生存期），310
heap allocated ARs（堆分配的AR），266-267，289
hoisting（提升），514-515，539
code motion（提升代码移动），506
code shrinkage（提升的代码缩小），539
defined（提升的定义），506
Hollerith constant（Hollerith常量），67
Hoperoft's algorithm（Hoperoft算法），55-59
hybrid IRs（混合型IR），211

I

identifiers（标识符）
classification based on declared type（基于声明类型的分类标识符），137
keywords as（作为标识符的关键字），65
tokens（记号标识符），241
IDom（IDom），415，416-417
array（IDom数组），460
defined（IDom定义），415
illustrated（图解IDom），416
as key to finding DOM（寻找DOM关键的IDom），460
using（IDom的使用），416-417
See also dominator-based value numbering
if-then-else ambiguity（if-then-else歧义性），81-82，121
if-then-else constructs（if-then-else构造），30，34，356
as ambiguous construct example（作为歧义性构造示例的if-then-else构造），81-82
control flow inside（if-then-else构造内的控制流），359
frequency of execution（if-then-else构造的执行频率），358-359
implementation strategy（if-then-else构造实现策略），356
nested（if-then-else构造嵌套），34，364
predication vs. branching（if-then-else构造的预测和分支），358-359
uneven amounts of code（if-then-else构造的代码不平均量），359
ILOC（ILOC），659-672
abstract machine（ILOC抽象机器），659
alternate branch syntax（ILOC的其他分支语法），668-669，672
arithmetic operations（ILOC的算术操作），662-663
basic block（ILOC的基本块），221
conditional move（ILOC的条件移动），330-331
control-flow operations（ILOC的控制流操作），667-670，672
defined（ILOC的定义），17，659
example（ILOC的示例），666-667
generating，for expressions（ILOC的表达式生成），197-198
jump operation（ILOC的跳转操作），669-670
load address-immediate instruction（ILOC的装入地址立即指令），317-318
memory operations（ILOC内存操作），237，664-665
naming conventions（ILOC的命名约定），662
opcode summary（ILOC操作码的总括），671
operands（ILOC的操作数），661
operations（ILOC操作），660，662-665
programs（ILOC程序），660
register-to-register copy（ILOC的寄存器到寄存器操作），665
shift operations（ILOC的移入操作），663-664
SSA form representation（ILOC的SSA形式表示），670
this book（本书中的ILOC），237
virtual registers，renaming（ILOC的虚拟寄存器的重命名），230

implicit deallocation（隐式释放），299-305
indirection vectors（间接向量），335，340-341
defined（间接向量的定义），336
reference requirement（间接向量的引用需求），341
in row-major order（行优先顺序中的间接向量），337
using（使用间接向量），340
See also arrays
induction variables（归纳变量），529，530，531，532
creation（归纳变量的创建），534-535
dead（死亡归纳变量），537
defining（归纳变量定义），531
in hash table（散列表中的归纳变量），534
initial value of（归纳变量的初始值），535
See also variables
inference（推理）
declarations and（声明和推理），168
for expressions（表达式推理），168-170
rules（推理规则），167-168
inheritance（继承），271-272
code reuse and（代码复用和继承），271
complication（继承的复杂性），375
defined（继承的定义），271
hierarchy（继承层次），376
implementing（继承的实现），376
multiple（多重继承），378-379
no（非继承），373-375
single（单一继承），375-377
See also object-oriented languages
inherited attributes（继承属性），173
inline substitution（内联替换），427-430
after（内联替换之后），429
before（内联替换之前），428
defined（内联替换的定义），429
improvements（内联替换的改进），429-430
as interprocedural analysis alternative（作为过程内分析选择的内联替换），487
operation elimination（内联替换的操作消除），430
optimization effectiveness（内联替换的优化效率），429
potential pitfalls（内联替换的潜在隐患），430
insertion sets（插入集合），507
instances（实例），270
instruction schedulers（指令调度器）
alternative strategy（指令调度器的选择策略），592
constraints，freeing（从限制中释放指令调度器），592
goals（指令调度器的目标），586
input（指令调度器的输入），586
multiple blocks and（多个块和指令调度器），595
instruction scheduling（指令调度），19-21，522，585-618
constraining（指令调度的限制），21
defined（指令调度的定义），21，545，586
dependences（指令调度的相关性），522
as difficult problem（作为困难问题的指令调度），21
difficulty（指令调度的困难），593
fundamental operation（指令调度基本操作），593
instruction selection and（指令筛选和指令调度），549
local（局部指令调度），593
problem（指令调度问题），587-594
quality measures（指令调度的质量度量），593
regional（区域指令调度），605-613
register allocation and（寄存器分配和指令调度），594
instruction selection（指令筛选），16-18，545-584
automatic construction（指令筛选的自动构造），550
choices（指令筛选的选择），18
CISC and（CISC和指令调度），579
complexity（指令调度的复杂性），546
defined（指令调度的定义），545
peephole-based（基于窥孔的指令调度），551，558，569-580
register allocation and（寄存器分配和指令调度），549
RISC and（RISC和指令调度），579
scheduling and（调度和指令调度），549
tree-walk scheme（指令调度的树遍历方案），552-558
via tree-pattern matching（通过树模式匹配的指令调度），558-569
instruction sequences（指令序列）
generating（指令序列生成），581-582
search for（指令序列搜索），581
synthesizing（指令序列合成），582
instruction set architecture (ISA)（指令集体系结构），545
complex（复杂的指令集体系结构），571
constraints（指令集体系结构的限制），547
target（目标指令集体系结构），579
instruction-cache misses（指令缓冲失误），539-540
instruction-level parallelism (ILP)（指令级并行），588
exhausted（指令级并行耗尽），599

memory latency and (内存等待时间和指令级并行), 588
instructions (指令)
idle (空闲指令), 332
ILOC program (ILOC程序指令), 660
operations and (操作和指令), 588-589
per second (每秒指令数), 615
sequential list of (指令的顺序表), 660
integers (整数)
assignment (整数赋值), 168
length of (整数长度), 161
pointer to (指向整数的指针), 165
signed (带符号整数), 35
storage-efficient representations (整数的高效存储表示), 66
two-dimensional array of (整数的二维数组), 162
unsigned (无符号整数), 34, 37, 39
interference graphs (冲突图) 639-641, 684
2-colorable (2着色冲突图), 640
adjacency vectors and (邻接向量和), 684
as basis for coalescing (作为接合基础的冲突图), 651
breaking, into smaller pieces (把冲突图切割成小冲突图), 652
building (冲突图的构建), 640-641
construction illustration (结构图解冲突图), 640
defined (冲突图的定义), 639
imprecise (非精确冲突图), 651-652
machine constraint encoding in (冲突图中的机器限制编码), 649-651
size/sparsity of (冲突图的大小/稀疏性), 700-701
interference-region spilling (冲突区域溢出), 653
interference (s) (冲突), 639-641
defined (冲突的定义), 640, 641
live ranges and (存活范围域和冲突), 639
intermediate representations (IRs) (中间表示), 5, 6, 209-250
in actual use (真实使用中的中间表示), 226
augmentation (配置中间表示), 209
builder (IR构建器), 239
choice of (IR的选择), 231, 249
data space requirements (IR对数据空间的需求), 212
defined (IR的定义), 209
designing (IR的设计), 210
expressiveness (IR的表示能力), 212-213
generation/manipulation cost (生成/处理IR的代价), 212
graphical (图示IR), 210, 213-222, 679-684
hybrid (混合型IR), 211
implementing (IR实现), 210, 679-686
linear (线性IR), 211, 222-228
low-level (低级IR), 233, 571-572
operation deletion from (从IR中消除操作), 493
selecting (选择IR), 210
static single-assignment form (SSA) (IR的静态单一赋值形式), 228-231
structural organization (IR的结构化组织), 210-213
tree-based (基于树的IR), 212
interpreters (解释器)
compilers vs. (编译器和解释器), 2-3
defined (解释的定义), 2
uses (使用解释器), 2
interprocedural analysis (过程间分析), 362, 434, 483-488
constant propagation (过程间分析的常量传播), 485-486
control-flow analysis (过程间分析的控制分析), 484
inlining alternative (过程间分析的内联选择), 487
pointer analysis (过程间分析的指针分析), 486
recompilation (过程间分析的重编译), 487-488
summary problems (过程间分析的概括问题), 484-485
interprocedural constant propagation (过程间的常量传播), 485-486
interprocedural methods (过程间方法), 407-408, 409
irreducibility (不可约性), 481
irreducible graphs (不可约图)
defined (不可约图的定义), 480
recursive-descent parser (不可约图的递归下降语法分析器), 483
reduction sequences (不可约图的归约序列), 482
transforming (不可约图的转换), 481-482
iterated coalescing (迭代接合), 652
iterative data-flow analysis (迭代数据流分析), 435-454
limitations (迭代数据流分析的限制), 447-450
live variables (迭代数据流分析中的活变量), 435-444
LiveOUT solver properties (迭代数据流分析的LiveOUT解算器的性质), 444-447
See also data-flow analysis
iterative live analysis (迭代活的分析), 441

J

Java, scoping rules (Java作用域规则), 261-262

jump（跳转）
functions（跳转函数），485
table implementation（跳转表实现），368
targets（跳转目标），440

K

keywords（关键字）
handling（处理关键字），65
as identifiers（作为标识符），65
regular expression（关键字的正则表达式），37
Kleene closure（Kleene闭包），37，38，44

L

LALR(1) construction（LALR(1)结构），147
languages（语言）
comments（语言注释），40
data types（语言的数据类型），151
declaration-free（无声明语言），203
defined（语言的定义），37
defined by context-free grammars（由上下文无关文法定义的语言），75
dynamically checked（动态检查语言），169
dynamically typed（动态类型语言），156，169
microsyntax（语言的微语法），28
name spaces（语言的名字空间），260
natural languages vs.（自然语言和语言），43
object-oriented（面向对象语言），157
regular（正则语言），42
regular expressions in（语言中的正则表达式），36
scoping rules（语言的作用域规则），259-262
statically checked（静态检查语言），169
statically typed（静态类型语言），156，169
strongly typed（强类型语言），156，159
untyped（无类型语言），156，159
weakly typed（弱类型语言），156
words in（语言中的字），43
last-in，first-out(LIFO) discipline（后进先出规则），265
latencies（等待时间）
hiding（隐藏等待时间），497，587
load（装入等待时间），610
memory（内存等待时间），588，637
`mult` operational（`mult`操作等待时间），588
operational（操作等待时间），590
later placements（较晚放置），506，511-512
defined（较晚放置定义），511-512
equations（较晚放置方程），512
See also anticipable expressions;earliest placement
lazy code motion (LCM)（惰性代码移动），506-514
anticipable expressions（惰性代码移动的可前置表达式），509-514
available expressions（惰性代码移动的可用表达式），508-509
background（惰性代码移动背景），507-508
defined（惰性代码移动的定义），505，506
DELETE set（惰性代码移动DELETE集合），513
equations（惰性代码移动的方程），508
example（惰性代码移动的示例），509，514
expression movement（惰性代码移动的表达式移动），508
INSERT set（惰性代码移动的INSERT集合），512-513
local information（惰性代码移动的装入信息），507-508
rewriting code（惰性代码移动的代码重写），512-514
See also code motion
leaders（头部），439
leaf procedures（叶过程），372-373
least-recently-used (LRU) replacement（最近最少使用置换），294
left associativity（左结合性），139，140
left recursion（左递归）
associativity（左递归的结合性），139-140
eliminating（左递归消除），94-96
general，removal of（一般左递归消除），96
immediate（立即左递归），95
indirect（间接左递归），95，96
productions（左递归产生式），92
right recursion vs.（右递归和左递归），138-140
stack depth（左递归的栈深度），138-139
termination problems（左递归的终止问题），94
See also recursion
left-factoring（左因子分解），101
left-linear grammars（左线性文法），87
left-recursion grammars（左递归文法），92
eliminating（左递归文法消除），94-96
left associativity（左递归文法的左结合性），204
termination problems and（终止问题和左递归文法），94
`lex` scanner generator（`lex`扫描器生成器），61
lexical scopes（词法作用域）
adding（增加词法作用域），694-698

example（词法作用域的示例），243
to open addressing（开放寻找的词法作用域），696-698
to open hashing（开放散列的词法作用域），695-696
See also scopes
linear IRs（线性IR），222-228，684-686
data structures（线性IR的数据结构），225-228
defined（线性IR的定义），211
as definitive representation（作为确定性表示的线性IR），223
implementing（线性IR的实现），225-228
logic（逻辑线性IR），223
ordering（线性IR排序），223
stack-machine code（栈机器代码的线性IR），223，224
three-address code（三地址代码的线性IR），223，224-225
See also intermediate representations(IRs),
linear lists（线性列表），687
linear representations（线性表示），225-228
linear search（线性搜索），365
linear-function test replacement(LFTR)（线性函数测试替换），537-538
after（LFTR之后），538
defined（LFTR的定义），537
edge labels（LFTR的边标签），537
features（LFTR的性质），538
performing（LFTR的执行），537
linkages（链接）
building（构建链接），287-288
convention（链接约定），286
illustrated（图解链接），287
negotiation（链接协商），289
standardized（标准的链接），286-290
link-time optimizers（链接时优化器），488
LINPACK（LINPACK）
defined（LINPACK的定义），385
dmxpy（dmxpy LINPACK），387
list of structures（结构列表），685-686
list representation（列表表示），675-677
list scheduling（列表调度），595-605
adaptation（列表调度的适应性），595
algorithm（列表调度算法），596，603
backward（向后列表调度），60-603
as basis（作为基础的列表调度），605
classic（典型的列表调度），595
defined（列表调度的定义），593，595
efficiency（列表调度的效率），598-599，604
execution time（列表调度的执行时间），601
forward（向前列表调度），600-603
greedy heuristic（列表调度的贪婪启发式搜索），603
local（局部列表调度），599
reasons to use（列表调度的使用理由），603-605
from roots to leaves（从根到叶的列表调度），600
variations（列表调度的变形），617
live range splitting（分离存活范围），643，653-654
passive（被动存活范围的分离），653-654
zero-cost（存活范围的零代价分离），653
live ranges（存活范围），630-631
in basic block（基本块中的存活范围），631
building, from SSA form（从存活范围构建SSA形式），636
coalescing（存活范围的接合），646-647
complex（复杂存活范围），634
defined（存活范围定义），630
discovering（存活范围的发现），636
finding（寻找存活范围），631，634
global（全局存活范围），635-637
in global allocator（全局分配中的存活范围），633-634
interference and（冲突与存活范围），639
ordering in bottom-up coloring（存活范围的自底向上着色的排序），644
partial，spilling（分离部分存活范围），653
set of（存活范围的集合），630
specific placement，of（存活范围的特定放置），650-651
spilling and（溢出和存活范围），641
live variables（活变量），435-444
as backward problem（作为向后问题的活变量），438
CFG（CFG中的活变量），439-441
defined（活变量的定义），435
equations for（活变量的方程），437-438
in global register allocation（全局寄存器分配中的活变量），436
in SSA construction（SSA结构中的活变量），436
uses（使用活变量），435-437
See also variables
LiveIN（LiveIN），631
liveness（活性），630
LiveNOW（LiveNOW），640
LiveOUT（LiveOUT），631
AVAILvs.（LiveOUT和AVAIL），438-439

correctness (LiveOUT的正确性), 445
dominance for (LiveOUT的支配关系), 458
efficiency (LiveOUT效率), 445-447
equations (LiveOUT方程), 437
example computation (LiveOUT示例计算), 442
properties (LiveOUT性质), 444-447
round-robin postorder solver for (LiveOUT的round-robin后序解算器), 446
sets, computing (LiveOUT的集合计算), 438
solving equations for (LiveOUT方程求解), 443-444
termination (LiveOUT的终止性), 444-445
LL(1) grammars (LL(1) 文法), 88, 102
defined (LL(1) 文法的定义), 104
usefulness (LL(1) 文法的有效性), 106
LL(1) parsers (LL(1) 语法分析器), 104
building table for (LL(1) 语法分析器表的构建), 105
skeleton (LL(1)语法分析器的框架), 106
table-driven (LL(1) 语法分析器的表驱动), 134
load address-immediate instruction (装入地址立即指令), 317-318
load latencies (装入等待时间), 610
load operation (load操作), 346, 347, 529, 665
load tracking (装入跟踪), 193-195
with ad hoc syntax-directed translation (带有专用语法制导翻译的装入跟踪), 193-195
copy rules for (装入跟踪的拷贝规则), 183
rules for (装入跟踪的规则), 182
loadAI operation (loadAI操作), 317, 318, 564, 664
loadAO operation (loadAO操作), 664
loadI operation (loadI操作), 317, 318, 330
local analysis, 434
local methods (局部方法), 404-405
defined (局部方法的定义), 624
join points (局部方法的连接点), 425
local register allocation (局部寄存器分配), 624-629
bottom-up (自底向上局部寄存器分配) 626-629
global allocation vs. (全局分配与局部寄存器分配), 633-634
optimal (优化的局部寄存器分配), 628
top-down (自顶向下局部寄存器分配), 625
See also register allocation
local value numbering (LVN) (局部值编号), 474
local variables (局部变量), 264
initializing (初始化局部变量), 264-265
of other procedures (其他过程的局部变量), 281-285
See also variables
logical windows (逻辑窗口), 577-578
lookup(s) (查找)
dynamic method (动态方法寻找), 377
failure (查找失败), 399-400
processes (查找处理), 248
loop scheduling (循环调度), 608-612
benefit (利益), 608
code reordering and (代码重排序的循环调度), 611
for one functional unit (一个功能单元的循环调度), 609
pipelined loop (管道循环), 608-612
techniques (循环调度技术), 608
loop unrolling (循环展开), 519-520
defined (循环展开的定义), 519
illustrated (图解循环展开), 519
key effects (循环展开的关键效应), 520
operation reduction (循环调度减少操作), 520
unknown bounds and (未知边界和循环展开), 520
See also enabling transformations
loop unswitching (循环反切换), 520-521
defined (循环反切换的定义), 520
effects (循环反切换的效应), 521
illustrated (图解循环反切换), 521
See also enabling transformations
loop-based methods (基于循环的方法), 407
loop-carried data dependence (循环进位数据相关), 520
loops (循环), 359-364
bounds, unknown (循环未知边界), 520
character-by-character (字符到字符的循环), 348
do (do循环), 361-362
empty, removing (消除空循环), 504
epilogue (循环的结语), 609
for (for循环), 359, 360-361
header block (循环的头部块), 633
iterations, order (循环的迭代顺序), 497
iterations, total number of (循环的迭代总量), 390
nest (嵌套循环), 387-388, 389, 497
pipelined (流水线循环), 608-612
prologue (循环的序言), 608, 611
scheduling (循环调度), 608-612
structure addresses in (循环中的结构地址), 528
unrolled (展开的循环), 388
unrolled inner (展开的内部循环), 391
unrolling (循环展开), 519-520
unswitching (循环的反切换), 520-521
until (until循环), 363

while (while循环), 34, 52, 54, 59, 302
word-oriented (面向字的循环), 347
lost-copy problem (无效拷贝问题), 476-478
arising of (无效拷贝问题引发的), 477
avoiding (回避无效拷贝问题), 478
defined (无效拷贝问题的定义), 475
example illustration (无效拷贝问题的示例图解), 477
low-cost matches (低代价匹配), 567-568
lower-level IR(LLIR) (低级IR)
front-end generation of (LLIR的前端生成), 578
operation overhead (LLIR的操作负担), 580
operations (LLIR操作), 571
sequences (LLIR序列), 578, 579
simplified, comparison (化简了的LLIR比较), 572
See also intermediate representations (IR)
low-level trees (低级树), 218
LR(1) grammars (LR(1) 文法), 88
LR(1) items (LR(1) 项目), 120, 121-122
canonical collection of sets for (LR(1) 项目的规范集合), 122-127
defined (LR(1) 项目的定义), 121
example (LR(1) 项目的示例), 122
finite (有穷LR(1) 项目), 123
left context (LR(1) 项目的左上下文), 122
position (LR(1) 项目的位置), 121-122
right context (LR(1) 项目的右上下文), 122
LR(1) parsers (LR(1) 语法分析器), 74, 115-120
build key (构建LR(1) 语法分析器的关键), 119
construction algorithm (LR(1) 语法分析器的结构算法), 114
defined (LR(1) 语法分析器的定义), 108
driver (LR(1) 语法分析器的驱动器), 116
generator system structure (LR(1) 语法分析器的生成器系统结构), 115
LALR(1)(LR(1) 语法分析器的LALR(1) 语法分析器), 115
one-symbol lookahead and (LR(1) 语法分析器与向前看一个符号), 114
properties (LR(1) 语法分析器的性质), 108
resynchronization (LR(1) 语法分析器中的重新同步), 134
rightmost derivation (LR(1) 语法分析器的最右派生), 108
SLR(1) (LR(1) 语法分析器的SLR(1) 分析器), 115
states (LR(1) 语法分析器的状态), 120
table-driven (LR(1) 语法分析器的表驱动), 134
See also bottom-up parsing
LR(1) tables (LR(1) 表格), 120-133
Action (LR(1) 表格中的Action), 116, 117, 118
building (LR(1) 表格构建), 120-133
canonical collection construction (LR(1) 表格的规范集合构造), 122-127
construction (LR(1) 表格构造), 116
construction algorithm (LR(1) 表格构造算法), 116
construction errors (LR(1) 表格构造错误), 129-133
construction trace (LR(1) 表格构造轨迹), 126, 130
directly encoding (LR(1) 表格的直接编码), 145-146
filling in (LR(1) 表格的填充), 127-128
Goto (LR(1) 表格中的Goto), 116, 117, 119
LR(1) items (LR(1) 表格中的LR(1) 项目), 121-122
for reduced expression grammar (缩小的表达式文法的LR(1) 表格), 146
rows/columns, combining (LR(1) 表格的行或列的合并), 144
size approximation (LR(1) 表格大小的近似), 147
size reduction (LR(1) 表格大小的减小), 143-147
See also LR(1) parsers

M

machine idiosyncrasies (机器特性), 312-313
machine-dependent transformations (机器相关转换) 496-497
bounded machine resource management (机器相关转换的有限机器资源的管理), 497
defined (机器相关转换的定义), 494
hardware features and (硬件特性和机器相关转换), 496-497
illustrated (机器相关转换的说明), 497
latency hiding (机器相关转换的等待时间的隐藏), 497
See also transformations
machine-independent transformations (机器无关转换), 494-496
computation specialization (机器无关转换的计算特化), 495-496
defined (机器无关转换的定义), 494
illustrated (机器无关转换的图解), 495
operation movement (机器无关转换的操作移动), 495
redundant computation elimination (机器无关转换的冗余计算的消除), 496
transformation enablement (机器无关转换的转换激活), 496

useless/unreachable code elimination（机器无关转换的无用/不可达代码的消除），495，498-505
See also transformations
marking algorithm（标记算法），302-303
conservative（保守的标记算法），302-303
illustrated（图解标记算法），302
precise（精确的标记算法），302
mark-sweep collectors（标记清理回收器），301，303
defined（标记清理回收器的定义），303
technique comparison（标记清理回收器的技术比较），304
matches（匹配）
code sequences for（匹配的代码序列），564
low-cost，finding（匹配的低代价寻找），567-568
precomputing（匹配的预计算），567
See also tree-pattern matching
maximal SSA form（极大SSA形式），457
may modify problem（可能修改问题），484-485
defined（可能修改问题的定义），484
equations（可能修改问题的方程），485
may reference sets（可能修改问题中的可能引用集合），485
meaning（意义）
checking（意义检查），12
defined（意义的定义），8，389
syntax vs.（语法与意义），153
meet-over-all-paths solution（所有路径相遇解），445
memory（内存）
cache（内存缓冲），293-295
latencies（内存等待时间），637
layout（内存布局），290-295
logical address-space layout（内存逻辑地址空间布局），291
managing（内存管理），290-305
multiregister operations（内存多寄存器操作），371
registers vs.（寄存器和内存），621
vector，layout（内存的向量布局），333
memory models（内存模型），235-238
choice of（内存模型的选择），236，621
impact on code shape（内存模型对代码形态的影响），292-293
memory-to-memory（内存到内存的内存模型），236，292
register-to-register（寄存器到寄存器的内存模型），235-236，292-293
memory operations（内存操作），664-665
delays（内存操作等待），599
executing（内存操作的执行），599
hierarchy（内存操作层次），237
multiword（多字内存操作），551
sequence（内存操作序列），628
speed（内存操作的速度），547
memory-to-memory model（内存到内存模型），236，292
code shape and（代码形态和内存到内存模型），292
register allocator in（内存到内存模型的寄存器分配器），621
values in memory（内存中的内存到内存模型的值），621
See also memory models
methods（方法）
defined（方法的定义），270
finding，in superclass hierarchy（在超类层次中寻找方法），272
lookups（方法查找），272-273，377
mapping names to（名字到方法的映射），272-274
tables（方法表格），378
microsyntax（微语法）
in context-free grammars（上下文无关文法中的微语法），89
defined（微语法的定义），28
syntax separation from（语法与微语法的分离），38-39
minimal SSA（极小SSA），468
mixed-type expressions（混合类型表达式），320
MSCP compiler（MSCP编译器），663
`multI` operation（`multI`操作），529
multipass compilers（多遍编译器），228
multiple inheritance（多重继承），378-379
complication（多重继承的复杂性），379
with trampoline function（带有蹦床函数的多重继承），379-380
See also inheritance
multiplicative hash functions（乘法散列函数），688
multipool allocators（多池分配器），298-299
alternate schemes（多池分配器的可选择方案），299
defined（多池分配器的定义），298
memory pools（多池分配器的内存池），298
multiregister values（多寄存器值），649-650
multiset discrimination（多重集判别），241

defined（多重集判别的定义），687
using（多重集判别的使用），241

N

name equivalence（名字等价），166，167
name resolution（名字分解）
defined（名字分解的定义），243
linked tables for（名字分解的链接表格），248
name spaces（名字空间），252-253，256-275
activation records（名字空间的活动记录），262-267
in Algol-like languages（类Algol语言中的名字空间），257-259
defined（名字空间的定义），252
in languages（各语言中的名字空间），260
of object-oriented languages（面向对象语言的名字空间），268-275
SSA（SSA名字空间），474，475
named values（命名值），309
names（名字）
compiler generation of（名字的编译器生成），232
declaring，at multiple levels（多层次的名字声明），242
expression（表达式的名字），419
full qualified（完整限定名字），374
impact of（名字的影响），234-235
inheritance（名字继承），256
level lifetime（名字的层次生存期），242
mapping，to methods（名字到方法映射），272-274
mapping values to（值到名字映射），231-238
name space management（名字空间的管理），247
register，arbitrary（任意寄存器名字），564
reusing（名字复用），231
set，in data-flow equations（控制流方程中的名字集合），449
SSA（SSA名字），454，455
static coordinate for（名字的静态坐标），259
temporary value（名字的临时值），233-235
translations and（翻译和名字），232
value numbering and（值编号和名字），402-403
visibility rules（名字可视性规则），274-275
natural languages（自然语言），43
nested scopes（嵌套作用域）
handling（嵌套作用域的处理），242-246
lexical（词法嵌套作用域），257-259
managing（嵌套作用域管理），244
in Pascal（Pascal中的嵌套作用域），258
See also scopes
nodes（结点）
AST（AST结点），216，394
attribute association with（与结点相关的属性），191
custom-allocating（结点定制分配），682
DAG（DAG结点），217，394
hard-to-color（难着色结点），649
long-latency（结点的长等待时间），600
with no predecessors（没有前驱的结点），589
rank（结点等级），600
root（根结点），589
sequence（结点序列），142
splitting（结点分离），483
systematic bias towards（结点的系统倾向），600
nondeterministic (NFAs)（非确定性有穷自动机），44，45-50
acceptance criteria（NFA）（NFA的可接受标准），47
behavior simulation（NFA的行为模拟），47
configuration（NFA的格局），47
defined（NFA的定义），46
deriving，from regular expression（从正则表达式得到NFA），48
equivalent DFA（与DFA等价的NFA），47
nondeterministic choices（NFA的非确定性选择），46
state（NFA的状态），47
states，transition for（NFA的状态转换），52
trivial，for RE operators（对应于RE操作符的平凡NFA），48
See also finite automata (FA)
noniterative data-flow algorithms（非迭代数据流算法），479，480
nonrecursive program（非递归程序），255
nonterminal symbols（非终结符），76
arbitrary order of（非终结符的任意顺序），96
defined（非终结符的定义），75
leftmost，expansion（非终结符的最左展开），80
operation trees and（操作树和非终结符），563
rewrite rules（非终结符的重写规则），560
set of（非终结符的集合），78
null-terminated strings（空终结串），346，349
number systems（数制），321-322
numbers（数）
as base types（作为基本类型的数），160-161
floating-point（浮动点数），160
real（实数），39，40

numerical encoding（数编码），324-325

O

object records（对象记录），270，375
  defined（对象记录定义），270
  laying out（对象记录布局），377-378
object-oriented languages（面向对象语言），157
  Algol-like languages vs.（类Algol语言和面向对象语言），274-275
  code reuse（面向对象语言的代码复用），269
  defined（面向对象语言的定义），268
  implementing（面向对象语言的实现），373-381
  inheritance（面向对象语言的继承性），271-272
  name spaces of（面向对象语言的名字空间），268-275
  single class，no inheritance（非继承面向对象语言的单一类），373-375
  single inheritance（单一继承面向对象语言），375-377
  terminology（面向对象语言的术语），270-271
objects（对象），268-270
  in classes（类中的对象），269
  defined（对象的定义），268
  instances（对象实例），270
  layout（对象布局），269
  as software-engineering strategy（作为软件工程策略的对象），268
observational equivalence（可观察等价），389
opcodes（操作码），660，661
Open64 compiler（open64编译器），226
open addressing（开放寻址），691-693
  adding lexical scopes to（增加开放寻址的词法作用域），696-698
  defined（开放寻址的定义），686
  drawbacks（开放寻址的缺点），692
  symbol records and（符号记录和开放寻址），693
  table（开放寻址表格），692
  table size（开放表格大小），693
open hashing（开放散列），689-691
  adding lexical scopes to（增加开放散列的词法作用域），695-696
  advantages（开放散列的优点），690
  assumptions（开放散列的假设），689
  defined（开放散列的定义），686
  drawbacks（开放散列的缺点），690-691
  symbol records and（符号记录和开放散列），693
  table（开放散列表格），690
operands（操作数），661
operation scheduling（操作调度）.*See* instruction scheduling
operation trees（操作树），559，563
operations（操作）
  address-immediate（地址立即操作），664
  address-offset（地址偏移操作），664
  arguments（操作参数），17
  arithmetic（算术操作），662-663
  candidate（候选操作），530
  commutative（可交换性操作），400
  conditional branch（条件分支操作），668
  control-flow（控制流操作），576-577，667-670
  copy（拷贝操作），403
  cost（操作代价），547
  defined（操作定义），587
  evaluation subtleties（评估操作中的微妙问题），526-527
  "false" dependence on（操作上的"假"依赖），21-22
  floating-point（浮点操作），547
  ILOC（ILOC操作），660
  instructions and（指令和操作），588-589
  issuing，per cycle（每周期的操作发行），589
  iteratively selecting（迭代地选择操作），596
  jump（跳转操作），669-670
  LLIR（LLIR操作），571
  long-latency（操作的长等待时间），20
  memory（内存操作），237，547，551，599，664-665
  moving（移动操作），495
  moving，from precall/postreturn sequences（从调用前序列/返回后序列移动操作），369
  multiple，at same time（同时的多个操作），597
  multiregister memory（多个寄存器内存操作），371
  operands（操作的操作数），17
  overlapping（操作交叠），87
  predicated（谓词操作），328
  priorities，assigning（操作优先级的指定），595-596
  register-to-register（寄存器到寄存器操作），579
  register-to-register copy（寄存器到寄存器拷贝操作），665
  rewriting（重写操作），526-527
  shift（移位操作），663-664
  string（串操作），344

three-address（三地址操作），313
three-address code（三地址代码操作），224-225
unreachable（不可达操作），498
variable-length（可变长度操作），685
See also specific operations
operator(s)（操作符）
assignment as（作为操作符的赋值），321
boolean（布尔操作符），322-333
closure（闭包操作符），37
comparison（比较操作符），668-669
overloading（操作符重载），156
relational（关系操作符），322-333
strength reduction（强度减弱操作符），384
strength reduction algorithm（强度减弱算法操作符），533
unary（一元操作符），135-136，320
whole-word（全字操作符），348
optimistic algorithms（优化算法），517
optimization scope（优化作用域），404-408
global methods（全局方法的优化作用域），407
interprocedural methods（过程间方法的优化作用域），407-408，409
intraprocedural methods（过程内方法的优化作用域），407，409
larger, as act of faith（坚信较大的优化作用域），425
local methods（局部方法的优化作用域），404-405
regional methods（区域方法优化作用域），405-407
superlocal methods（超局部方法的优化作用域），405
whole-program methods（整个程序方法优化作用域），407-408
See also code optimization;scopes
optimization(s)（优化），7
code（代码优化），15，383-432
combining（优化组合），523-527
defined（优化的定义），16
grammar（优化文法），141-143
interprocedural（过程间优化），408
leaf procedures（叶过程优化），372-373
peephole（窥孔优化），515，551，569-580
scalar（标量优化），491-544
short-circuit evaluation as（作为优化的短路评估），330
as software engineering（作为软件工程的优化），493
optimizers（优化器），492
design（优化器设计），492
interaction, increasing（增加优化器间的相互作用），581
link-time（链接时优化器），488
pass-structured（遍结构式优化器），493
peephole（窥孔优化器），552，569，570，575，580
repeating passes（重复遍优化器），492
separate（分离优化），493
optimizing compilers（优化编译器）
debugging compilers vs.（调试优化器和优化编译器），386
designing（优化编译器的设计），541
ordered lists（有序列表），675-677
OSR algorithm（OSR算法）
defined（OSR算法），529
header field（OSR算法中的标题字段），532
induction variable definition（OSR算法的归纳变量定义），531
Tarjan's strongly connected region finder（OSR算法中的强连通区域探测器），532
out-of-order execution（无序执行），604
overloading（重载），156

P

padding（填充），293
page faults, avoiding（避免页错误），539-540
parameter(s)（参数）
actual（实际参数），275
array-valued（数组值参数），341-343
binding（参数绑定），275
call-by-name binding（名字引用绑定参数），277
call-by-reference binding（引用调用绑定参数），276-279，318，319
call-by-value binding（值调用绑定参数），275-276
evaluating（参数评估），369-370
expansion（参数展开），70
formal（形式参数），275
passing（传递参数），275-279
procedure-valued（过程值参数），370
values, accessing（参数值存取），318-319
parametric polymorphism（参数多态性），170
parse trees（分析树），81，83，213-214
attributed, evaluating（评估属性分析树），172
with classic expression grammar（带有经典表达式文法的分析树），214
defined（分析树定义），213
derivation equivalent（分析树的派生等价），86

instantiating（分析树实例），186-187
large（大型分析树），141
partial（部分分析树），109，111
uses（分析树使用），213-214
See also syntax-related trees
parser generators（分析器生成器）
automatic（自动的分析器生成器），133
error-recovery routines（分析器生成器的错误恢复例程），134
LALR(1)（LALR(1)分析器生成器），190
syntax-directed actions and（语法制导动作和分析器生成器），190
parsers（语法分析器）
asymptotic behavior（语法分析器的渐近行为），141
bottom-up（自底向上语法分析器），86-87
for classic expression grammar（经典表达式文法语法分析器），86
engineering（语法分析器工程），140
as engines（作为工具的语法分析器），74
functioning of（语法分析器的功能），86
grammar（语法分析器中的文法），70
LL(1)（LL(1)语法分析器），104，105，106
LR(1)（LR(1)语法分析器），74，115-120
number of words and（字数和语法分析器），29
output（语法分析器的输出），86
predictive（预测语法分析器），102
responsibility（语法分析器的责任），73
size of（语法分析器的大小），28
syntax errors and（语法错误和语法分析器），134
table-driven（语法分析器的表驱动），104
top-down（自顶向下语法分析器），86-87，89-106
parsing（语法分析），73-149
bottom-up（自底向上语法分析），74，107-120
defined（语法分析定义），11，86
difficulty（语法分析的困难），28
goal（语法分析的目标），84
LR(1)（LR(1)语法分析），74
recursive-descent（递归下降语法分析），74
scanning and（扫描和语法分析），73
shift-reduce（移位归约语法分析），108-112
techniques（语法分析技术），74
top-down（自顶向下语法分析），74，89-106
partial parse trees（部分分析树），109，111
partitions（划分）
constructing DFA from（通过划分构建DFA），58
refining（划分细化），58
splitting（分离划分），57
Pascal（Pascal语言）
nested procedures（Pascal嵌套过程），257
nested scopes in（Pascal中的嵌套作用域），258
nonrecursive program（非递归程序），255
passes（遍）
code uselessness and（代码无用性和遍），492-493，499
repeating（遍重复），492
validation（确认遍），674
passive splitting（被动分割），653-654
pass-structured optimizers（遍结构式优化器），493
pattern matching techniques（模式匹配技术），539
pattern-driven optimization（模式驱动优化），571
patterns（模式）
peephole optimization（窥孔优化模式），580-581
priority specification（优先级描述模式），61
tree（模式树），558，562，563，566
PCC compiler（PCC编译器），226
peephole optimization（窥孔优化），515，551
algebraic identities（代数等式窥孔优化），570
as competitive technology（作为具有竞争力技术的窥孔优化），578
control-flow operations（窥孔优化中的控制流操作），576-577
dead values（窥孔优化中的死亡值），575-576
defined（窥孔优化的定义），569
for instruction selection（指令筛选的窥孔优化），558，569-580
library of patterns（模式库中的窥孔优化），551
logical windows（窥孔优化中的逻辑窗口），577-578
patterns, learning（窥孔优化中的学习模式），580-581
physical windows（窥孔优化中的物理窗口），577-578
window size（窥孔优化窗口大小），575
peephole optimizers（窥孔优化器），552
construction automation（窥孔优化器的自动构建），569
design issues（窥孔优化器的设计问题），575
early（早期的窥孔优化器），570
hand-coded patterns（窥孔优化器中的手工代码模式），570
interaction, increasing（增加窥孔优器间的相互作用），581
pattern-matching（窥孔优化器的模式匹配），580
running（窥孔优化器的运行），570
peephole transformers（窥孔转换器），578-580
perfect hashing（完美散列），65

performance（性能）
gap（性能差异），386
optimization improvement（优化性能的改进），430
run-time（运行时性能），386，615
pessimistic algorithms（悲观算法），517
PFC scanner（PFC扫描器），69
pipelined loops（管道化循环），608-612
algorithms（管道化循环算法），612
defined（管道化循环的定义），608
source-loop iterations（管道化循环的源循环迭代），612
value flow（管道化循环的值流），612
value flow illustration（管道化循环的值流说明），613
See also loop scheduling; loops
PL8 compiler（PL8编译器），226
pointer analysis（指针分析），486
complexity（指针分析的复杂性），449
motivation for（指针分析的动机），396
POINTSTO sets（指针分析的POINTSTO集合），486
pointer-based computations（基于指针的计算），166
pointers（指针）
activation record（指针活动记录），17，262
ambiguous（歧义性指针），396
assignments（指针的赋值），396-398
to boolean（指向布尔的指针），165
current-level，changing（指针当前层次的改变），245
defined（指针的定义），165
dereferences（指针的解除引用），320
in indirection structure（间接结构中的指针），340
to integer（指向整数的指针），165
list（指针列表），686
manipulation（指针处理），165
null（空指针），682
schemes（指针方法），680
static analysis and（静态分析和指针），449
storage-efficient representations（指针的高效存储表示），66
type safety with（指针类型的安全性），166
values（指针值），350，352
POINTSTO sets（POINTSTO集合），486
positional encoding（位置编码），325-327
assignment and（赋值和位置编码），326
benefits（位置编码的效益），327
with short-circuit evaluation（短路评估的位置编码），326
use of（位置编码的使用），326，327
positive closure（正闭包），38
postdominators（后支配者），501
postreturn sequences（后返回序列），287
defined（后返回序列的定义），287
moving operations from（从后返回序列中移动操作），369
powerset（幂集），47
preallocation registers（预分配寄存器），624
precall sequences（调用前序列），286-287
building（调用前序列的构建），369
call-by-value parameters（调用前序列中的值调用参数），370
defined（调用前序列的定义），286
moving operations from（从调用前序列中移动操作），369
precedence（优先权）
arithmetic（算术优先权），85
closure（闭包的优先权），39
graphs（优先图），589
levels of（优先级别），84
nonterminal association with（与优先权相关的非终结符），84
precise collectors（精确的回收器），302
predicated execution（谓词执行），331-332，551
defined（谓词执行的定义），332
example（谓词执行的示例），329
use of（谓词执行的使用），332
predication（谓词），331-332
branching vs.（分支与谓词），357
processor use of（处理器的谓词使用），332
predictive grammar（预测文法），103，105
predictive parsers（预测语法分析器）
cost of（预测语法分析器的代价），104
DFAs vs.（DFA与预测语法分析器的比较），102
importance（预测语法分析器的重要性），106
preorder walk（前序遍历），471
priority（优先级）
backward list scheduling（向后列表调度的优先级），602
computation（优先级计算），598
forward list scheduling（向前列表调度的优先级），601-602
ranking（优先级等级化），600
schemes（优先级方案），599-600
procedure abstraction（过程抽象），539
procedure calls（过程调用），170，368-373
data-flow analysis and（数据流分析和过程调用），450

implementing (过程调用的实现), 368
imprecision (过程调用的非精确性), 483
indirect costs (过程调用的间接代价), 427-428
leaf procedure optimizations (叶过程优化中的过程调用), 372-373
multiple sites (过程调用的多个场所), 368
as optimization barrier (作为优化障碍的过程调用), 427
parameter evaluation (过程调用的参数评估), 369-370
procedure-valued parameters (过程调用的值过程参数), 370
register save/restore (过程调用的寄存器保存或恢复), 370-372
summary problems (过程调用问题小结), 484-485
procedures (过程), 251-306
arguments (过程的参数), 152
called (调用的过程), 279, 290
defined (过程定义), 251
epilogue sequence (过程的结语序列), 286, 287
functions (过程的运行), 253
implicit arguments (过程的隐式参数), 370
interface between (过程间的接口), 251
invoking (过程调用), 152
leaf (叶过程), 372-373
linkage (过程链接), 286-290
nested (嵌套的过程), 257
placement (过程的放置), 540
prologue sequence (过程的序言序列), 286, 287
role (过程的作用), 252
value communication between (过程间的值传递), 275-279
procedure-valued parameters (值过程参数), 370
productions (产生式), 78, 79
breaking, into two pieces (将产生式分成两部分), 192
cost (产生式代价), 560
left-factoring (产生式的左因式分解), 101
left-recursive (产生式的左递归), 92
required rules for (产生式所需要的规则), 182
single symbol on right-hand side (右端为单一符号的产生式), 143
useless (无用产生式), 143
profitability (效益)
code optimization and (代码优化和效益), 390-391
resource constraints and (资源限制和效益), 414
programming languages (程序设计语言/编程语言) .*See* languages
prologue sequences (序言序列), 286, 287
pruned SSA (剪枝SSA), 468

R

random replacement (随机置换), 294
range checking (范围检测), 343-344
for array-valued parameters(数组值参数的范围检测), 343
defined (范围检测的定义), 343
run time code for (范围检测的运行时代码), 343
reaching definitions (可达定义), 450-451
computing (可达定义的计算), 456
defined (可达定义的定义), 450-451
as forward data-flow problem (作为向前数据流问题的可达定义), 451
initial information gathering for (可达定义的初始定义的收集), 451
read-only memory (ROM) (只读存储器), 23, 539
real numbers (实数)
complexity (实数的复杂性), 39
regular expression description of (实数的正则表达式的描述), 40
real-time collectors (实时回收器), 305
receivers (接收器)
address (接收器地址), 370
defined (接收器的定义), 271
recognizers (识别器)
construction with scanner generator (带有扫描器生成器的识别器构建), 65
DFAs as (作为识别器的DFA), 60-61
direct-coded (直接编码识别器), 64
encoding, into table set (到表集合的识别器编码), 35-36
implementing (识别器的实现), 34
regular expression specifications (识别器的正则表达式描述), 36
for unsigned integers (无符号整数的识别器), 34
recompilation (重编译), 487-488
analysis (重编译分析), 487-488
dependences (重编译的相关性), 488
recursion (递归)
associativity vs. (结合性与递归的比较), 205
left (左递归), 94-96, 138-140
right (右递归), 95-96, 138-140
tail (尾递归), 364

recursive programs（递归程序），256
recursive-descent parsers（递归下降语法分析器），101-106
code-space locality（递归下降语法分析器代码空间的位置），104
construction strategy（递归下降语法分析器的构造策略），103
error detection（递归下降语法分析器的错误检测），104
for expressions（表达式的递归下降语法分析器），105
hand-constructed（手工构造的递归下降语法分析器），102，143
irreducible subgraphs（递归下降语法分析器的不可归约子图），483
large grammar and（大型文法和递归下降语法分析器），143
for predictive grammar（预测文法的递归下降语法分析器），103，105
process automation（递归下降语法分析器过程的自动化），104-106
speed（递归下降语法分析器的速度），104
structure（递归下降语法分析器的结构），101
See also top-down parsers
recursive-descent parsing（递归下降语法分析），74
reduce action（归约动作），117
reduced expression grammar（归约表达式文法），146
reduce-reduce conflicts（归约归约冲突），133，137
reducible graphs（可约图），480
redundancy（冗余）
partial，converting（部分冗余的转换），507
syntactic notion of（冗余的语法表记法），418
redundancy elimination（冗余消除），393-404，523
with DAGs（使用DAG的冗余消除），394-398
global（全局冗余消除），417-424
lessons from（冗余消除的经验），403-404
useless code removal（冗余消除的无用代码消除），498
with value numbering（使用值编号的冗余消除），398-403
redundant expressions（冗余表达式），393-404
detecting/eliminating（检测或消除冗余表达式），393
names（冗余表达式的名字），419
potential set representation（可能的冗余表达式集合表示），418
at source level（源代码级的冗余表达式），393
reference counting（引用计数），300-301
arguments（引用计数的讨论），301
cost（引用计数的代价），301
defined（引用计数的定义），300
problems（引用计数问题），300-301
technique comparison（引用计数技术比较），304
See also garbage collection
region constants（区域常量），529，530
regional methods（区域方法），405-407
analysis（区域方法分析），406
focus（区域方法的焦点），406
join points（区域方法中的连接点），425
superlocal methods vs.（超局部方法与区域方法的比较），407
See also optimization scope
regional scheduling（区域调度），605-613
algorithms（区域调度算法），605
EBBs（EBB上的区域调度），605-607
loops（区域调度的循环），608-612
trace（跟踪区域调度），607-608
See also instruction scheduling
register allocation（寄存器分配），18-19，619-658
assignment vs.（赋值与寄存器分配），622-623
background issues（寄存器分配的背景问题），620-624
defined（寄存器分配的定义），545，619
global（全局寄存器分配），633-651
harder problems in（寄存器分配中的难题），654-657
instruction scheduling and（指令调度和寄存器分配），594
instruction selection and（指令筛选和寄存器分配），549
local（局部寄存器分配），624-629
multiregister values and（多寄存器值和寄存器分配），649-650
NP-complete（寄存器分配中的NP完全），622
scope（寄存器分配的作用域），629
specific placement（寄存器分配中的特定放置），650-651
register allocators（寄存器分配器），316
bad decisions in（寄存器分配中的不好的决策），620
bottom-up local（自底向上局部寄存器分配器），626-629
compiler decisions and（编译器的决策和寄存器分配器），620
concept（寄存器分配器的概念），620

core functions（寄存器分配器的核心功能），594
functioning of（寄存器分配器的功能），620
goal（寄存器分配器的目标），633
in memory-to-memory model（内存到内存模型中的寄存器分配器），621
for multiple blocks（多个块的寄存器分配器），629-633
in register-to-register model（寄存器到寄存器模型中的寄存器分配器），621
for single blocks（单一块的寄存器分配器），624-629
specific register placement（寄存器分配器的特定寄存器放置），650-651
spill code insertion（寄存器分配器的溢出代码的插入），594
top-down local（自顶向下局部寄存器分配器），625
value relegation（寄存器分配器的值的移交），619
virtual name space mapping（寄存器分配器对虚拟名字空间的映射），594
register assignment（寄存器赋值），594
allocation vs.（寄存器赋值与分配的比较），622-623
defined（寄存器赋值的定义），622
global（全局寄存器赋值），633-651
local（局部寄存器赋值），624-629
in polynomial time（多项式时间内的寄存器赋值），622
register classes（寄存器分类），623-624
disjoint（不相交的寄存器分类），312
physically/logically separate（物理上/逻辑上分离的寄存器分类），623
register pressure（寄存器压力），425
register sets（寄存器集）
clustered machine（组群寄存器集的机器），656
partitioned（分割的寄存器集），655-656
registers（寄存器）
assigning, in priority order（按优先度顺序指定寄存器），625
callee-saves（被调用者存储寄存器），288，371
caller-saves（调用者存储寄存器），371
condition-code（条件代码寄存器），328，624
demand, reducing（减小寄存器需求），315-318
demand for（对寄存器的需求），391
floating-point（浮点寄存器），623
general-purpose（通用寄存器），546，623
keeping values in（把值保存在寄存器中），310-312
memory vs.（内存与寄存器的比较），621
minimizing number of（寄存器数量的最小化），19
physical, demand for（对物理寄存器的需求），623
preallocation（寄存器的预分配），624
predicate（谓词寄存器），623-624
promotion（寄存器提升），656
refined specification, implementing（实现寄存器细化描述），63
reserving（保留寄存器），643
restoring（恢复寄存器），370-372
saved values（已保存的寄存器值），265
saving（存储寄存器），288，370-372
specific placement（寄存器的特定放置），650-651
speed（寄存器的速度），619
virtual（虚拟寄存器），18，292，311，624
register-to-register copy operations（寄存器到寄存器的拷贝操作），665
register-to-register model（寄存器到寄存器模型），235-236，292-293
code shape and（代码形态和寄存器到寄存器模型），292-293，621
operations（寄存器到寄存器模型的操作），579
register allocator in（寄存器到寄存器模型中的寄存器分配器），621
See also memory models
register-transfer language (RTL)（寄存器转换语言），226，579
code interpretation（寄存器转换语言的代码解释），580
defined（寄存器转换语言的定义），579
processing（寄存器转换语言的处理），580
regular expressions (REs)（正则表达式），36-44
alternation（正则表达式的选择），37
closed（封闭正则表达式），42
closure（正则表达式的闭包），38
complexity（正则表达式的复杂性），42
concatenation（正则表达式的连接），38
construction from DFA（从DFA构建正则表达式），59-60
context-free grammars vs.（上下文无关文法与正则表达式的比较），87-89
defined（正则表达式的定义），27，36
deriving, from DFA（从DFA得到正则表达式），59
deriving NFAs from（从NFA得到正则表达式），48
examples（正则表达式示例），39-42
finite automata and（有穷自动机和正则表达式），36，44
keywords（正则表达式的关键字），37
language（正则表达式语言），37
notation formalization（正则表达式的形式化表记法），

37-39
operations（正则表达式操作），37-38
priorities（正则表达式中的优先级），61
programming language（正则表达式的程序设计语言），36
properties（正则表达式的性质），42-44
real numbers description（实数的正则表达式描述），40
recognizer specifications（识别器的正则表达式描述），36
regular languages（正则表达式的正则语言），42
for six-character identifiers（六字符标识器的正则表达式），40
for syntactic categories（语法范畴的正则表达式），60
in virtual life（虚拟生活中的正则表达式），38
widespread use of（正则表达式的广泛使用），71
"zero or more occurrences,"（正则表达式的"零个或多个出现"）37
regular grammars（正则文法），87，88
defined（正则文法的定义），88
quadruple（四元组正则文法），87
use（正则文法的使用），88
regular languages（正则语言），42
rehashing（再散列），691-693
adding lexical scopes to（向…增加词法作用域），696-698
defined（定义再散列），686
drawbacks（再散列的缺点），692
table（再散列表），692
table size（再散列表的大小），693
relationals（关系），322-333
adding，to expression grammar（把关系操作加入到表达式文法中），323
hardware support（关系操作的硬件支持），328-332
implementing（实现关系操作），329
numerical encoding（编码数字关系），324-325
positional encoding（编码位置关系），325-327
representations（关系表示），323-328
rematerialization（重实现），654
renaming（重命名），466-474
algorithm（重命名算法），466，569
avoiding antidependencies and（避免反相关性和重命名），595
defined（重命名的定义），456
dynamic register（动态寄存器的重命名），604
as enabling transformation（作为激活转换的重命名），521-522
example（重命名的示例），469-473
register assignment and（寄存器赋值和重命名），594
states（重命名的状态），670-671
as subtle issue（作为微妙问题的重命名），522
`repeat until` loop（`repeat until`循环），112
representations（表示）
arbitrary graphs（任意图表示），682-684
boolean（布尔表示），323-328
compactness（表示的简洁性），677
linear（线性表示），225-228
linked list（链表表示），675
list（列表表示），675-677
sparse-set（稀疏集合的表示），678-679
string（串表示），344-345
symbol table（符号表表示），239
representing sets（表示集合），674-679
as bit vectors（作为位向量的表示集合），677
as ordered lists（作为有序列表的表示集合），675-677
sparse（稀疏的表示集合），678-679
representing trees（表示树），680-681
illustrated（图解表示树），681
methods（表示树方法），680
strengths/weaknesses（表示树的优点/缺点），680-681
resource (s)（资源）
bounded machine，managing（管理有限的机器资源），497
constraints（资源限制），414
`restrict` keyword（`restrict`关键字），312
retargetable compilers（可重定目标编译器），548-552
building（构建可重定目标编译器），548-552
defined（可重定目标编译器的定义），548
exhaustive search and（穷举搜索和可重定目标编译器），582
goal（可重定目标编译器的目标），549
reverse dominance frontier（反向支配边界），500
reverse postorder (RPO)（反向后序）
effectiveness（反向后序的高效性），445
for forward problem（前驱问题的反向后序），446
illustrated（图解反向后序），446
impact on LiveOUT computation（反向后序对LiveOUT计算的影响），446
rewrite rules（重写规则），559-565
applying，to trees（把重写规则运用于树），562
contents（重写规则的内容），559-560

defined（重写规则的定义），559
illustrated（图解重写规则），561
nonterminal symbols（重写规则中的非终结符号），560
for tiling（铺盖的重写规则），561
right associativity（右结合性），139，140
right context（右上下文），70，122
right recursion（右递归），95-96
associativity（右递归的结合性），139-140
left recursion vs.（左递归与右递归的比较），138-140
stack depth（右递归的栈深度），138-139
See also recursion
right-recursive expression grammar（右递归表达式文法），97
backtrack-free（右递归表达式文法的无回溯性），98
right associativity（右递归表达式文法的右结合性），204-205
stack depth（右递归表达式文法的栈深度），139
RISC machines（精简指令集计算机（RISC）机器）
compiler simplicity and（编译器简化和RISC机器），579
instruction selection and（指令筛选和RISC机器），579
roots，dependence graph（依赖图的根），589
row-major order（行优先顺序），335-336
address calculation（行优先顺序的地址计算），338
assignment statement（行优先顺序中的赋值语句），336
indirection vectors in（行优先顺序中的间接向量），337
layout（行优先顺序布局），335
See also arrays
rule-based evaluators（基于规则的评估器），188
run-time environment（运行时环境），13-14
run-time heap.*See* heap
run-time performance（运行时性能），386，615
run-time safety（运行时安全性），155-156
run-time tags（运行时标签），355

S

safety（安全性），388-390
defining（定义安全性），389
proving（证明安全性），390
run-time（运行时安全性），155-156
S-attributed grammars（S属性文法），178，195
scalar optimizations（标量优化），491-544
example（标量优化的示例），498-523
transformations（标量优化的转换），494-497
scalar replacement（标量取代），656
scanner generators（扫描器生成器）
actions and（动作和扫描器生成器），66
actions in accepting states（扫描器生成器的接受状态中的动作），65
lex（lex扫描器生成器），61
for recognizer construction（扫描器生成器的识别器构造），65
right context notation（扫描器生成器的右上下文记法），70
tools（扫描器生成器工具），61
scanners（扫描器），
automatic generation illustration（扫描器的自动生成说明），35
construction（扫描器的构造），28
construction automation（扫描器的自动构造），35-36
context-free grammars and（上下文无关文法和扫描器），89
with DFA，implementing（扫描器的DFA实现），62
direct-coded（扫描器的直接编译），64
for English（英语扫描器），43
functioning of（扫描器的功能），27
high-quality（高质量扫描器），39
implementation simplification（简化扫描器的实现），35
implementing（实现扫描器），61-67
overhead（扫描器的开销），29
PFC（PFC扫描器），69
for register names（寄存器名的扫描器），62
table-driven（扫描器的表驱动），62-64
two-pass（两遍扫描器），69-70
use of（扫描器的使用），28
well-implemented（优秀的扫描器），29
word categories（扫描器的字范畴），66
scanning（扫描），27-72
context（上下文扫描），69-70
defined（扫描的定义），11
FORTRAN（FORTRAN扫描），68
parsing and（语法分析和扫描），73
schedules（调度）
backward list scheduling（向后列表调度的调度），602
execution time（调度的执行时间），590
iteratively selecting（迭代地选择调度），*596*
length（调度长度），590-591

register demand（调度的寄存器需求），593
time-optimal（时间优化调度），591，593
well-formed（形式良好的调度），590
scheduling（调度）
algorithms（调度算法），593
defined（调度的定义），589
difficulty（调度的困难），593
EBBs（EBB上的调度），605-607，608
example（调度的示例），588
fundamental operation in（调度中的基本操作），593
instruction（指令调度），19-21，522，585-618
list（列表调度），593，595-605
loops（循环调度），608-612
regional（区域调度），605-613
trace（跟踪调度），607-608
Scheme（Scheme）
recursive program（Scheme递归程序），256
scoping rules（Scheme作用域规则），261
scoped value tables（作用域值表），410
scopes（作用域）
block，deleting（删除块作用域），409
current（当前作用域），244
defined（作用域的定义），256
inheritance from（作用域的继承），256
labeling（作用域标签），242
level（层次作用域），245
lexical（词法作用域），243，694-698
nested（嵌套的作用域），242-246，257-259
optimization（优化的作用域），404-408
in procedural languages（过程语言中的作用域），257
register allocation（作用域的寄存器分配），629
transformation，increasing（增加作用域转换），425
scoping rules（作用域规则），257
C（C作用域规则），259-260
FORTRAN（FORTRAN作用域规则），259
Java（Java作用域规则），261-262
Scheme（Scheme作用域规则），261
in various languages（各语言中的作用域规则），259-262
semantic analysis（语义分析），12
semantics（语义）
checking（语义检测），12
defined（语义的定义），8
syntax vs.（语法和语义的比较），153
semipruned SSA form（半剪枝SSA形式），457，468
sentences（句子）
constructing（构造句子），79-83
deriving（获得句子），75
grammatical structure and（文法结构和句子），81
sentential form（句型）
defined（句型的定义）76
obtaining（获得句型），77
separate compilation（分离编译），251
shadow index variables（影子索引变量），362
shift action（移入动作），117
shift operations（移入操作），663-664
shift-reduce parser（移入归约语法分析器），190
shift-reduce parsing（移入归约语法分析），108-112
algorithm（移入归约算法），112
defined（移入归约分析的定义），109
potential handles（移入归约分析的可能的句柄），113
shifts（移入归约分析的移位），112
short-circuit evaluation（短路评估），327-328
boolean identities（短路评估布尔等式），327
cost reduction（短路评估的代价减少），327
defined（短路评估的定义），327
as optimization（作为优化的短路评估），330
requirement（短路评估需求），330
use of（短路评估的使用），328
simulation clock（模拟钟），597
sinking（下沉），539
SLR(1) construction（SLR(1)构造），147
snapshot tool（快照工具），299
source-level trees（源代码级树），217，218
source-to-source translators（源语言到源语言翻译器），2
space（空间），23
sparse conditional constant propagation (SCCP)（稀疏条件常量传播），524-526
defined（稀疏条件常量传播的定义），524-525
effectiveness（稀疏条件常量传播的有效性），527
illustrated（图解稀疏条件常量的传播），525
initialization（稀疏条件常量传播的初始化），525-526
lattice value（稀疏条件常量传播的格值），526
multiplies rules（稀疏条件常量传播中的多条规则），527
nonmonotonic behavior and（非单调行为和稀疏条件常量传播），526
power（稀疏条件常量传播的威力），527
propagation（稀疏条件常量传播的传播），526
sparse simple constant propagation (SSCP)（稀疏简单常量传播），516-518

complexity (SSCP的复杂性), 516
coverage (SSCP的覆盖面), 517-518
defined (SSCP的定义), 516
illustrated (图解SSCP), 517
lattice value assignment (SSCP格值的赋值), 524
See also constant propagation
sparse-set representation (稀疏集合表示), 678-679
defined (稀疏集合表示的定义), 678
vectors requirement (稀疏集合表示的向量需求), 679
specialization (特化), 515-518
computation (特化计算), 495-496
constant propagation (特化的常量传播), 515, 516-518
peephole optimization (特化的窥孔优化), 515
tail-recursion elimination (特化的尾递归消除), 515
speed (速度), 23
available expressions and (可用表达式和速度), 418
memory operations (内存操作的速度), 547
recursive-descent parsers (递归下降语法分析器的速度), 104
registers (寄存器速度), 619
spill costs (溢出代价)
global, estimating (全局溢出代价, 评估), 637-638
infinite (无限溢出代价), 638
negative (负溢出代价), 638
spill metric (溢出度量), 646
spilling (溢出)
avoiding (避免溢出), 627
clean values (净值溢出), 628
cost (溢出代价), 628
defined (溢出定义), 620
dirty values (非净值溢出), 628, 629
floating-point registers (浮点寄存器溢出), 623
global, cost estimation (全局溢出代价估测), 637-638
interference-region (区域冲突溢出), 653
live ranges and (活范围和溢出), 641
partial live ranges (部分活范围的溢出), 653
predicate registers (谓词寄存器的溢出), 623-624
top-down coloring and (自顶向下着色和溢出), 642-643
splitting (分离)
live ranges (分离活范围), 643
nodes (分离结点), 483
partitions (分离划分), 57
SSA graphs (SSA图), 524
relating SSA to (关联SSA与SSA图), 530
transformed (转换后的SSA图), 536
See also static single-assignment (SSA) form
stack allocation (栈分配)
of activation records (活动记录的栈分配), 265-266
advantages (栈分配的优点), 266
of dynamically sized array (动态数组的栈分配), 266
symbol records (符号记录的栈分配), 694
stack depth (栈深度), 138-139
stack-machine code (栈机器代码), 224
defined (栈机器代码的定义), 223
generation/execution (栈机器代码生成/执行), 224
swap operation (栈机器代码的swap操作), 224
See also linear IRs
stack-relative notation (栈相关表记法), 113
start symbol (开始符号), 75, 78
static allocation (静态分配), 267
static analysis (静态分析)
dynamic analysis vs. (语法分析与静态分析的比较), 436
over different scopes (不同作用域上的静态分析) 434
pointers and (指针和静态分析), 449
reasoning (静态分析的推理), 433
See also data-flow analysis
static initialization (静态初始化), 375
static links (静态链接) .*See* access links
static single-assignment (SSA) form (静态单赋值形式), 228-231, 454-479, 488
build method (SSA的构建方法), 456-457
building (构建SSA), 230
building live ranges from (从SSA构建活范围), 636
in code optimization (代码优化中的SSA), 229
data-flow analysis (SSA的数据流分析), 454-479
defined (SSA的定义), 228
dominance (SSA的支配), 457-463
encoding control flow into data flow (SSA把控制流编码成数据流), 455
example in (SSA的示例), 410
executable code reconstruction from (从SSA重新构建可执行代码), 474-479
loop (SSA中的循环), 229
maximal (极大SSA), 457
minimal (极小SSA), 468
name space (SSA的名字空间), 474, 475

names（SSA名字），455
naming（命名SSA），233
oddities（SSA的怪僻），229-230
program constraints（SSA形式的程序的限制），228
properties（SSA的性质），411
pruned（剪枝SSA），468
rewriting（重写SSA），475
semipruned（半剪枝），457，468
in three-address IR（三地址IR的SSA形式），230
unique name（SSA形式的惟一名字），454
See also SSA graphs
static variables（静态变量）
access to（静态变量的存取），280
combining（组合静态变量），281
data areas（数据区内的静态变量），290-291，310
statically checked languages（静态检测语言），169
statically typed languages（静态类型语言），156，169
static-distance coordinate（静距离坐标），281
stop and copy collectors（停机拷贝回收器），303
storage layouts，array（数组的存储布局），335-337
storage locations（存储位置）
array elements（数组元素的存储位置），337
assigning（指定存储位置），309-313
assigning，to declared variables（指定已声明变量的存储位置），202
choice of（存储位置的选择），13
`store` operation（`store`操作），346，347，601，612，665
straight condition codes（直线条件代码），328-330
strength reduction（强度削弱），527-538
algorithm（强度削弱算法），532-533
background（强度削弱背景），529-532
defined（强度削弱定义），527-528
example illustration（强度削弱示例说明），528
linear-function test replacement（强度削弱中的线性函数测试替换），537-538
rewriting algorithm（强度削弱中的重写算法），534-535
string assignment（串赋值），345-349
concept（串赋值概念），345
with whole-word operators（使用全字操作的串赋值），348
string (s)（串），163，344-350
arrays vs.（数组和串的比较），163
as collection of symbols（作为符号集合的串），76
concatenation（连接串），349
as constructed type（作为结构类型的串），163
index（索引串），66
length（串长度），350
manipulation overhead（串处理的开销），66-67
null-terminated（串的空终止），346，349
operations（串操作），344
overlapping（串交叠），212
quoted character（被引用字符串），40
representations（串表示），66，344-345
sentential form（串句型），76，77
sets of（串集合），37
strongly checked implementation（强检测实现），169
strongly typed languages（强类型语言），156，159
structural data-flow algorithms（结构数据流算法），480-483
structural equivalence（结构等价），166，167
structure layouts（结构布局），350
illustrated（说明结构布局），353
multiple copies of（结构布局的多重拷贝），353-354
understanding（结构布局的理解），352-355
unused space（结构布局的未使用空间），353
user exposure（对用户公开的结构布局），353
structure references（结构引用），350-355
anonymous values（结构引用中的匿名值），351-352
arrays of structures（结构引用中的结构数组），353，354
code emission for（为结构引用的代码发行），352
structure layouts（结构引用的结构布局），352-353
structures（结构）
alternative view（结构的另一种观点），164
defined（结构的定义），164
as distinct types（作为不同类型的结构），164
type of（结构的类型），165
variants and（变量和结构），164-165
subset construction（子集构造法），51-55
ε-closure offline computation（子集构造法的ε-闭包脱机计算），55
algorithm steps（子集构造法算法步骤），54
applying，toDFA（把子集构造法运用于DFA），53
defined（子集构造法的定义），51
DFFA from（子集构造法的DFFA），55-56
example（子集构造法的例子），52-54
fixed-point computations（子集构造法不动点计算），54
illustrated（子集构造法的说明），51
subtrees evaluation order（子集构造法的子树评估顺序），318

superclasses (超类), 271
superlocal methods (超局部方法), 405
  defined (超局部方法的定义), 405
  regional methods vs. (超局部方法与区域方法的比较), 407
superlocal value numbering (超局部值编号), 408-413
  defined (超局部值编号的定义), 411
  effectiveness (超局部值编号的效率), 412-413
  example (超局部值编号的示例), 411-413
  results (超局部值编号的结果), 412
  scope creation (超局部值编号的作用域创建), 412
  See also value numbering
superscalar processors (超标量处理器), 586, 587
  operation determination (超标量处理器的操作决策), 586
  two-unit, executing (两功能单位超标量处理器的执行), 610
swap problem (交换问题), 478-479
  defined (交换问题的定义), 475
  example illustration (交换问题的示例说明), 478
  set of ϕ-functions (交换问题中的ϕ函数集合), 479
`switch` statement (`switch`语句), 308, 309
symbol records (符号记录)
  stack allocation (符号记录栈分配), 694
  storing (符号记录存储), 693-694
symbol tables (符号表), 238-248, 701
  building (构建符号表), 241-242
  as central repository (作为中心库的符号表), 698
  defined (符号表的定义), 193
  dependent-type fields (符号表中的类型相关域), 193
  flexible design (符号表的巧妙设计), 698-700
  hash (散列符号表), 239-241
  index (索引符号表), 241
  interface routines (符号表接口例程), 241-242
  linked (链接符号表), 248
  multiset discrimination for (符号表中多重级判别), 241
  organizing (符号表组织), 687
  representation (符号表表示), 239
  role (符号的作用), 701
  scoped (作用域符号表), 242-246
  selector (筛选器符号表), 247
  separate (分离表), 247
  unified (统一表), 247
  uses (符号表使用), 246-248
symbolic simplifier (符号简化器), 581
symbols (符号)
  current input (当前输入符号), 92
  designated (特定的当前符号), 109
  `eof` (`eof`符号), 121
  goal (目标符号), 75, 78, 79, 120
  nonterminal (非终结符号), 75, 76, 78, 560, 563
  start (开始符号), 78
  terminal (终结符号), 75, 76,, 78, 93
syntactic categories (语法范畴), 29
  of highest-priority pattern (最高优先级模式的语法范畴), 61
  regular expressions for (语法范畴的正则表达式), 60
syntax (语法)
  checking (语法检测), 9-11
  correctness (语法的正确性), 12
  declaration (语法声明), 199, 200
  defined (语法定义), 8
  errors (语法错误), 110, 134
  expressing (表示语法), 74-89
  meaning vs. (意义与语法的比较), 153
  microsyntax separation from (从语法中分离的微语法), 28-29
  recognizing (识别语法), 73
  specification-based approach (基于描述的语法), 10
syntax-related trees (与语法相关树), 213-218
  abstract syntax trees (ASTs) (与语法相关树中的抽象语法树), 214-215
  abstraction level (与语法相关树的抽象层次), 217-218
  directed acyclic graph (与语法相关树中的有向无环图), 216-217
  low-level (低级与语法相关树), 218
  parse trees (与语法相关树中的分析树), 213-214
  source-level (源语言级与语法相关树), 217, 218
synthesized attributes (合成属性), 173

T

table-driven scanners (表驱动扫描器), 62-64
table-filling algorithm (表填充算法), 128
  grammar ambiguity and (文法的歧义性和表填充算法), 132
  LR(1) (LR(1)表填充算法), 128
tables (表格)
  generating, from DFA description (从DFA描述生成表格), 63
  hash (散列表), 239-241
  LR(1) (LR(1)表格), 120-133

structure（结构表），247-248
symbol（符号表），193，238-248
tail calls（尾调用表），364
tail-recursion（尾递归）
defined（尾递归定义），364
elimination（尾递归消除），515
programs（尾递归程序），615-616
tbl operation（tbl操作），440，670
terminal symbols（终结符号），76，93
Action table（Action表格中的终结符号），144
arrayed（排列的终结符号），104
defined（终结符号的定义），75
removing（消除终结符），145
set of（终结符号的集合），78
Thompson's construction（Thompson构造法），48-50
applying（运用Thompson构造法），50
alb pattern（Thompson构造法中的*alb*模式），61
defined（Thompson构造法的定义），48
NFAs derived from（从Thompson构造法得到NFA），49，51
properties dependence（Thompson构造法的性质依赖），49
three-address code（三地址代码），224-225
benefits（三地址代码的好处），225
defined（三地址代码的定义），223
implementations（三地址代码的实现），227
operations（三地址代码的操作），224-225
as quadruples（作为四元组的三地址代码），226
SSA form in（三地址代码中的SSA形式），230
See also linear IRs
three-phase compilers（三阶段编译器），6
thunks（形实转换），277
tiling（铺盖），559
for AST（AST的铺盖），564
costs（铺盖代价），568
defined（铺盖定义），559
finding（寻找铺盖），565-568
goal（铺盖目标），566
implementation（实现铺盖），559
locally optimal（局部优化的铺盖），568
rewrite rules for（铺盖的重写规则），561
selection（选择铺盖），564
time-optimal schedules（时间优化调度），591，593
top-down coloring（自顶向下着色），642-643
defined（自顶向下着色的定义），642
live range splitting（自顶向下着色的活范围的分割），643
priority ranking（自顶向下着色的优先级等级），649
spill handling（自顶向下着色的溢出处理），642-643
spill-and-iterate philosophy（自顶向下着色的溢出迭代哲学），648
See also global register allocation
top-down local register allocation（自顶向下局部寄存器分配），625
top-down parsers（自顶向下语法分析器），89-106
defined（自顶向下语法分析器的定义），86-87
efficient（高效的自顶向下语法分析器），108
expansion forever（自顶向下语法分析器的永远展开），92
functioning of（自顶向下语法分析器的功能），89-90
recursive-descent（递归下降自顶向下语法分析器），101-106
See also parsers; parsing
top-down parsing（自顶向下语法分析），89-106
algorithm（自顶向下语法分析的算法），91
backtrack elimination（自顶向下语法分析的回溯消除），97-101
complications（自顶向下语法分析的复杂性），94
example（自顶向下语法分析的示例），90-93
left recursion elimination（自顶向下语法分析的左递归消除），94-96
trace scheduling（跟踪调度），607-608
defined（跟踪调度的定义），607
EBB scheduling and（EBB调度和跟踪调度），608
procedure schedule（跟踪调度的过程调度），608
trampoline function（蹦床函数），379-380
class-specific（类特定蹦床函数），380
defined（蹦床函数的定义），379
multiple inheritance with（使用蹦床函数的多重继承），380
transformations（转换），16
enabling（激活转换），496，518-522
left-factoring（转换的左因式分解），101
machine-independent（机器无关的转换），494-496，496-497
scope，increasing（增加转换的作用域），425
strength reduction（转换的强度削弱），527-538
taxonomy for（转换的分类），494-497
transition diagrams（转换图）
as code abstractions（作为代码抽象的转换图），31
DFA（DFA转换图），59
quintuple equivalency（转换图的四元组等价），32
tree patterns（树模式），566
context capture（树模式的上下文刻画），562

as grammar rules（作为文法规则的树模式），563
working with（使用树模式），558
tree-pattern matching（树模式匹配），551，558-569
cost computation（计算树模式匹配的代价），568
instruction selection via（通过树模式匹配实现指令筛选），558-569
on quads（四元组上的树模式匹配），571
process，speeding up（加速树模式匹配过程），567
rewrite rules（树模式匹配的重写规则），559-565
tools（树模式匹配工具），568-569
trees（树）
abstract syntax (ASTs)（抽象语法树），139，196-197，217-218，556，559
attributed parse（属性分析树），172，174，187
binary（二叉树），681-682
dominator（支配者树），416，458，459，474
implementation，simplifying（简化树的实现），682
low-level（低级树），218
mapping（映射树），681-682
operation（树操作），559，563
parse（分析树），81，83，172，186-187，213-214
representing（树表示），680-681
source-level（源语言级树），217，218
syntax-related（与语法相关树），213-218
tree-walk（树遍历），552-558
code generator（树遍历的代码生成器），318，562
for expressions（表达式树遍历），314
for instruction selection（指令筛选树遍历），552-558
recursive routine（递归树遍历历程），315
scheme（树遍历方案），552-558
trivial base addresses（平凡基地址），280-281
two-dimensional hash tables（二维散列表），698-699
defined（二维散列表定义），699
illustrated（二维散列表说明），699
two-pass scanners（两遍扫描器），69-70
first pass（两遍扫描器的第一遍），69
second pass（两遍扫描器的第二遍），69
type checking（类型检测），159-160
at compile time（编译时类型检测），159
declarations and（声明和类型检测），203
deferring（类型检测的延迟），171
defined（类型检测的定义），159
type equivalence（类型等价），166-167
name（名字类型等价），166，167
structural（结构式等价），166，167
type inference（类型推断），155-156
accurate（精确的类型推断），171
ad hoc framework for（类型推断的专用框架），195
for expressions（表达式的类型推断），168-170，195
harder problems（类型推断中的较难的问题），202-204
interprocedural aspects of（类型推断中过程间的相关问题），170-171
rules（类型推断的规则），167-168
successful（成功的类型推断），204
type signatures（类型署名）
defined（类型署名的定义），170
in symbol tables（符号表中的类型署名），193
type systems（类型系统），154-171
base types（类型系统中的基本类型），160-161
better code generation（生成较好代码的类型系统），157-159
classifying（类型系统的分类），169
components（类型系统的组成部分），160-171
compound/constructed types（混合类型系统/结构类型系统），162-166
defined（类型系统的定义），154
expressiveness（类型系统的表示能力），156-157
inference rules（类型系统的推断规则），167-168
purpose of（类型系统的目的），154-160
run-time safety（运行时安全的类型系统），155-156
secondary vocabulary（次要词汇表的类型系统），154
theory（类型系统理论），160
type equivalence（类型等价的类型系统），166-167
well-constructed（结构良好的类型系统），156
well-designed（设计良好的类型系统），157
See also context-sensitive analysis
type-consistent uses（类型相容运用），203-204
constant function types and（常量函数类型和类型相容运用），203
unknown function types and（未知函数类型和类型相容运用），203-204
type(s)（类型）
base（基本类型），160-161
cell（单元类型），156
compound（混合类型），162-166
constant function（常量函数类型），203
constructed（结构类型），162-166
declared（声明的类型），168
defined（类型定义），154
determination at compile time（编译时类型决策），157
dynamic changes in（类型中的动态变化），204
enumerated（枚举类型），163-164

errors（类型错误），167-168
Fortran result（Fortran中的结果类型），155
representing（类型表示），167
return（返回类型），203
specification（类型描述），154
unknown function（未知函数），203-204
typing rules（类型规则），203

U

unambiguous values（非歧义性值），312
unary operators（一元操作符），135-136，320
absolute value（绝对值一元操作符），135
adding（加入一元操作符），135
types of（一元操作符的类型），135
See also operators
unions（联合），354-355
element references（联合中的元素引用），355
multiple structure definitions（联合中的多重结构定义），354
universal hash functions（通用散列函数），688
unknown function types（未知函数类型），203-204
unreachable code（不可达代码）
defined（不可达代码的定义），498
eliminating（不可达代码的消除），505
See also useless code elimination
unrolling loops（循环展开），390，391
unsigned integers（无符号整数）
description（无符号整数的描述），39
finite automata for（无符号整数的有穷自动机），37
recognizer for（无符号整数的识别器），34
`until` loops（`until`循环），363
untyped languages（无类型语言），156，159
upper frontier（上边界），107
use points（使用点），221
useless code（无用代码），498
useless code elimination（无用代码消除），495，498-505
algorithm（无用代码消除算法），499-501
control dependencies definition（无用代码消除中的控制相关定义），499
control flow（无用代码消除中的控制流），501-505
illustrated（无用代码消除的图解），500
passes（无用代码消除的遍），499
redundancy elimination and（冗余消除和无用代码消除），498
useless control flow elimination（无用控制流消除），501-505
block combining（无用控制流消除中块结合），502
branch hoisting（无用控制流消除中分支提升），502
empty block removal（无用控制流消除中的空块消除），501
redundant branch folding（无用控制流消除中的冗余分支的叠入），501
useless productions（无用产生式），143
user-defined types（用户定义类型），320

V

validation passes（确认遍），674
value numbering（值编号），398-403
algorithm（值编号算法），398-399
algorithm extension（值编号算法的扩展），400-402
basis（值编号的倾向），398
code transformation（值编号中的代码转换），403-404
for commutative operations（可交换操作的值编号），400
cost（值编号的代价），408
defined（值编号的定义），398
dominator-based（基于支配者的值编号），413-417
example（值编号示例），399
with extensions（使用扩展的值编号），401
local（局部值编号），474
naming and（命名和值编号），402-403
opportunity discovery（值编号中的机会发现），403
over regions larger than basic blocks（大于基本块区域上的值编号），408-417
scope increase and（作用域增加和值编号），414
scope of optimization differences（优化作用域不同的值编号），541
superlocal（超局部值编号），408-413
values（值）
ambiguous（歧义性值），312，656-657
anonymous（匿名值），350，351-352
boolean（布尔值），322
clean（净值），628
dead（死亡值），575-576
dirty（非净值），628，629
initializing（值的初始化），200
keeping，in register（把值保存于寄存器中），310-312
lifetimes（值的生存期），310
mapping，to names（值到名字的映射），231-238
memory-resident（内存居住值），313
multiregister（多寄存器值），649-650

name choice（值的名字选择），402
named（命名的值），309
naming（值命名），191-192
parameter，accessing（存取参数值），318-319
pointer（指针值），350，352
position（值位置），241
returning（值返回），279
spilling（值溢出），620
stack-based（基于栈的值），191
temporary，naming（临时值命名），233-235
unambiguous（非歧义性值），312
unnamed（未命名的值），309
virtual register assignment（虚拟寄存器的赋值），311
variables（变量）
class（类变量），262，271，380
dead（死亡变量），575
different-sized representations（不同大小表示的变量），554
free（自由变量），261
global（全局变量），280，281，290-291
induction（归纳变量），529，530，531，532
initializing（变量初始化），264-265
instance（变量实例），270
live（活变量），435-444
local（局部变量），264，281-285
memory layout（变量的内存布局），293
name choice（变量名选择），402
pointer-based（基于的指针的变量），449
shadow index（影子索引变量），362
static（静态变量），280，290-291
storage classes（存储类变量），554
vectors（向量）
adjacency（邻接向量），684
bit（位向量），677
characteristic（特征向量），677
defined（向量的定义），333
dope（内情向量），342-343
elements，referencing（向量元素的引用），333-335
indirection（间接向量），335，336-337
lower/upper bounds（向量的下界/上界），333
memory layout（向量的内存布局），333
systematic addressing（向量系统寻址），336
See also arrays
very busy expressions（非常忙碌表达式），451-452
very-long instruction word (VLIW) machines（超长指令字机器），586
packed（压缩的VLIW），587
scheduler（VLIW机器的调度器），587
virtual functions（虚函数），392
virtual registers（虚拟寄存器），18，292
defined（虚拟寄存器的定义），624
feasible（可行的虚拟寄存器），625
number of（虚拟寄存器的数量），624
priority computation for（虚拟寄存器的优先级计算），625
single（单一虚拟寄存器），629
sorting（分类虚拟寄存器），625
value assignment（虚拟寄存器的值赋值），311
See also register allocation; registers
`volatile` keyword（`volatile`关键字），312

W

weakly typed languages，（弱类型语言）156
weakly-checked implementation，（弱检查实现）169
`while` loops（`while`循环），34，52，54，302
in case statement（选择语句中的`while`循环），367
code（`while`循环代码），363
implementation（`while`循环实现），363
iteration（`while`循环迭代），54，125
splitting sets（`while`循环分离集合），59
See also loops
whole-program allocation（整个程序分配），654-655
parameter-binding mechanisms and（参数绑定机制和整个程序分配），655
performing（执行整个程序分配），655
potential（整个程序分配潜能），654
whole-program methods（整个程序方法），407-408
words（字）
categories（字范畴），66
character-by-character formulation（按字符识别字公式），29-33
complex，recognizing（复杂字的识别），33-35
lexical structure of（字的词法结构），35
in programming languages（程序设计语言中的字），43
recognizing（识别字），29-36
scanner recognition of（字的扫描器识别），28
syntactic categories（字的语法范畴），29

Z

zero-cost splitting（零代价分割），653