计 算 机 科 学 丛 书

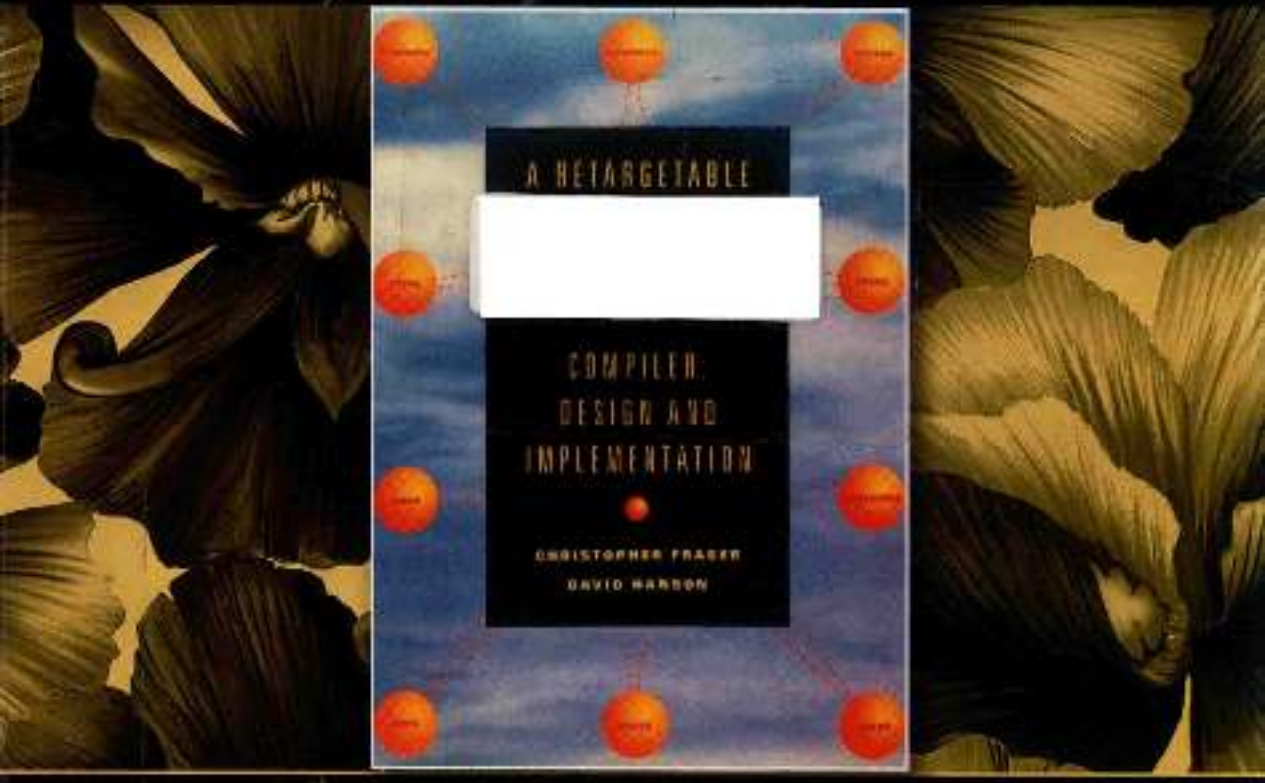
Pearson

可变目标C编译器

设计与实现

[美] 克里斯多夫 W. 弗雷泽（Christopher W. Fraser） 著
戴维 R. 汉森（David R. Hanson）
王挺 黄春 等译

A Retargetable C Compiler
Design and Implementation



机械工业出版社
China Machine Press

计 算 机 科 学 丛 书

可变目标C编译器

设计与实现

[美] 克里斯多夫 W. 弗雷泽（Christopher W. Fraser） 著
戴维 R. 汉森（David R. Hanson）
王甜 黄春 等译

A Retargetable C Compiler
Design and Implementation



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

可变目标C编译器：设计与实现/（美）克里斯多夫W.弗雷泽（Christopher W. Fraser），（美）戴维R.汉森（David R. Hanson）著；王甜等译．—北京：机械工业出版社，2016.11
（计算机科学丛书）
书名原文：A Retargetable C Compiler: Design and Implementation

ISBN 978-7-111-55258-1

I.可… II.①克…②戴…③王… III.C语言-编译器-程序设计 IV.TP312.8

中国版本图书馆CIP数据核字（2016）第260794号

本书版权登记号：图字：01-2011-2873

Authorized translation from the English language edition, entitled A Retargetable C Compiler: Design and Implementation (ISBN 978-0-8053-1670-4) by Christopher W. Fraser and David R. Hanson, published by Pearson Education, Inc., Copyright © 1995.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese simplified language edition published by Pearson Education Asia Ltd., and China Machine Press Copyright © 2016.

本书中文简体字版由 Pearson Education（培生教育出版集团）授权机械工业出版社在中华人民共和国境内（不包括香港、澳门特别行政区及台湾地区）独家出版发行。未经出版者书面许可，不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有 Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

本书系统地介绍了可变目标ANS C编译器lcc的设计与实现技术。lcc是一个实用的编译器，能够为不同的目标机器（如MIPS R3000、SPARC、Intel 386及其兼容机）生成代码。本书结合lcc的具体实例，详细地讲述了存储管理、符号表、词法分析、语法分析、中间代码生成、优化、目标代码产生与编译程序的各个部分。

本书特色鲜明、实用性强，适合作为高等院校计算机专业编译原理课程的参考书，对从事编译相关工作的技术人员也有很好的参考价值。

出版发行：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：刘锋泰　　责任校对：殷　虹
印　刷：北京诚信伟业印刷有限公司　　版　次：2016年11月第1版第1次印刷
开　本：184mm×260mm　1/16　　印　张：17.25
书　号：ISBN 978-7-111-55258-1　　定　价：79.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　购书热线：（010）68326294　88379649　68995259
投稿热线：（010）88379604　　读者信箱：hzjsj@hzbook.com

版权所有·侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所　韩光/邹晓东

文艺复兴以来，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举世性的优势；也正是这样的优势，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，不仅奠划了研究的范畴，还揭示了学术的源变，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和发展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到积极的推动作用，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公司较早意识到“出版要为教育服务”。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brian W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson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、研究及珍藏。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相助，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，有的还专门为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，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。其影印版“经典原版书库”作为姊妹篇也越来越多实施双语教学的学校所采用。

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。随着计算机科学与技术专业学科建设的不断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将步入一个新的阶段，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。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：

华章网站：www.hzbook.com
电子邮件：hzjsj@hzbook.com
联系电话：（010）88379604
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

华章教育
华章科技图书出版中心

译 者 序

A Retargetable C Compiler: Design and Implementation

编译器构造原理和技术可以说是计算机科学中理论与实践相结合的最好典范。到目前为止，大多数教材都是介绍编译器构造原理的，很少有详细介绍实用编译器构造的专门书籍。在编译原理课程的教学过程中，如何设计和组织实验一直是一个难题。这主要是因为，任何实用的编译器往往都是庞大的程序，而小的实验编译器又难以反映编译程序构造的许多重要技术。本书可以说是弥补了传统编译教材在实践方面的缺陷，如果希望向学生展示实用编译器是如何设计的，本书应该是最佳选择。

lcc 编译器是一个具有产品级质量的用于研究的 C 编译器，在 UNIX 界广为流行。本书深入到 lcc 编译器的内部，在代码级对该系统的设计和实现进行了详细的介绍。全书共分 19 章，详细讲述了存储管理、符号表、词法分析、语法分析、中间代码生成、优化、目标代码产生等编译程序的各个部分。本书特别针对 3 种目标机器（MIPS、SPARC 和 Intel 386）介绍了代码生成器的设计。通过学习上述内容，读者可以深入了解产品级编译程序设计中的许多关键技术，对于如何设计和实现一个实用的编译器有具体、真切的认识，这是其他教材无法达到的效果。

本书的两位作者都具有深厚的教学和研究背景。Christopher W. Fraser 从 1975 年就开始研究编译技术，尤其对于从紧凑规范自动产生代码生成器这一技术有深入的研究，在该领域发表了多篇论文。他提出了可变目标的窥孔优化方法，该方法被广为流行的 C 编译器——GCC 所采纳。从 1977 年到 1986 年，Fraser 在亚利桑那大学从事计算机科学（包括编译技术）的教学工作，1986 年以后，他在 AT&T 贝尔实验室主持计算技术的研究工作。David R. Hanson 是普林斯顿大学计算机科学教授，具有 20 多年的研究程序语言的经验，主持了与贝尔实验室的合作研究，是 lcc 的开发者之一。

本书是作者的教学、科研和开发思想以及经验的总结，读者可以从字里行间体会到两位作者在编译器的研究和设计方面的造诣。

国防科学技术大学计算机学院从事编译原理课程教学和科研工作的几位教师共同完成了本书的翻译工作。全书由王挺、黄春、刘金红、张晓燕和陈耀东负责翻译，由王挺和黄春通篇整理。由于时间和水平有限，翻译错误在所难免，恳请读者指正。

编译器是程序员使用的关键工具，程序员每天都在使用编译器，并且非常依赖于其正确性和可靠性。编译器必须接受程序语言的所有标准定义，以便源代码可以实现跨平台的可移植性。编译器必须生成高效的目标代码。但更重要的是，编译器必须生成正确的目标代码，只有可靠的编译器才能生成可靠的应用程序。

编译器本身是一个大而复杂的应用程序，值得我们深入分析研究。本书介绍了 ANSI C 语言编译器 lcc 的大部分实现，对编译器的介绍方式与 B. W. Kernighan 和 P. J. Plauger 合著的《Software Tools》（Addison-Wesley，1976）一书对文本处理（例如文本编辑和宏处理）的介绍类似。研究实用的工具软件，是学习软件设计和实现技术的最好方法。本书在代码级详细介绍了一个实用的编译器，该编译器的完整源代码可在 ftp.cs.princeton.edu（128.112.152.13）服务器的 pub/lcc 目录下，通过匿名 ftp 服务得到。

lcc 不是一个研究系统，而是一个实用的编译器产品。从 1988 年开始，lcc 就用于编译实际程序，现在每天都有数百名 C 程序员在使用它。由于本书详细分析了 lcc 编译器的设计与实现，因此用于介绍相关支撑材料的篇幅较少，仅展示了涉及的理论知识，而更为系统的编译技术的介绍可以参见其他教材。本书有意省略一些涉及通俗和重复实现的语言特征，而将这部分内容作为练习。

显然，本书将使读者对编译器的构造有更多的了解。然而只有少数程序员需要了解编译器的设计与实现，大多数程序员从事的是应用程序或其他系统程序的开发。但是，基于以下 4 个原因，大多数 C 程序员都可以从本书中受益。

第一，一般来说，如果程序员能够理解 C 编译器的工作原理，通常可以成为较好的程序员，特别是较好的 C 程序员。编译器设计者必须全面准确地理解 C 语言的每一个特性，程序员通过学习这些特性的实现，能够更好地掌握语言本身及其在现代计算机上的高效实现。

第二，大多数程序设计教材都是通过一些精简的示例来说明编程技巧的。但大多数程序员都是在从事大型程序的开发，在开发过程中需要不断修改程序，很少有带详细说明的示例可以作为大型程序设计的参考。lcc 不是完美的，但是本书详细说明了该程序的优缺点，可以作为大型程序开发的参考。

第三，编译器是计算机科学中理论与实践相结合的最好典范。lcc 展示了理论与实践的相互作用及其精美的结果，展示了实践需求牵引理论的发展，这些都可以清楚地从代码中找到。通过一个真实的程序来研究这些相互作用，可以帮助程序员理解何时、何地以及如何运用不同的技术。此外，lcc 也阐明了众多的 C 编程技术。

第四，这本书本身是一个文本程序（literate program），如同 D. E. Knuth 所著的《TeX: The Program》（Addison-Wesley，1986）一样，本书包括 lcc 的源代码及说明。为了方便读者理解，本书并未按源程序的顺序对程序代码进行讲解，而是有意进行了调整。

无论是对于在校学生还是专业技术人员，本书都非常适合自学使用。本书为 lcc 提供了说明完整的源代码，希望进行编译技术实践的人员，以及在需要使用或实现基于语言的工具和技术的应用领域（如用户接口）中工作的专业人员，将会对本书感兴趣。lcc 的相关信息可通过以

下载地址获得：www.cs.princeton.edu/software/lcc。

本书全面而真实地展示了一个大型软件系统，可作为软件工程课程的分析实例。

对于编译课程来说，本书弥补了传统编译教材的不足。本书介绍了C编译器的一种实现方法，而传统教材主要介绍编译过程中遇到的各种问题的解决算法，因此传统教材受篇幅限制只能介绍一些实验性的编译器，代码生成也通常面向较高的级别，以避免与具体的机器相关。

因此，许多教师要求学生完成接近实际的编译器项目，使学生获得实践经验。通常，教师必须从头开始编写编译程序，而学生复制其中的大部分，修改后利用其余的部分。然而，由于编译器只是实验性的，文档往往显得不够充分，这种情形使教学双方都不满意。本书通过对一个实际编译器的大部分程序进行文档说明，并提供源代码，为教师提供了一种新的选择。

本书介绍了完整的代码生成器，代码生成面向MIPS R3000、SPARC和Intel 386及其后续体系结构等不同的平台。本书利用了最新的研究成果，根据目标机器的紧缩规范（compact specification）生成代码生成器。这些方法使得我们能够针对多种机器展示完整的代码生成器，这是其他书籍无法做到的。通过介绍多个代码生成器，既避免了本书依赖于单一的机器，又有助于学生了解如何设计可变目标的软件。

教师布置的作业可以是增加编译器接受的语言特征、优化、改变目标机器等。本书如果与传统教材配合使用，也可以要求学生使用不同的算法代替现有的模块作为实践作业。如果以实现一个实验编译器作为实践作业，则可能在低级基础结构和重复的语言特征上花费大量的时间。采取上述方法，就能够更接近实际的编译器工程实践。本书的许多练习都涉及编译器工程问题。

除传统的编译目的外，lcc也有其他用途。例如，它可以用于构建一个C程序浏览器，或者根据声明来生成远程过程调用的桩函数（stub），也能用于语言扩充、新的计算机体系结构和代码生成技术的实验。

本书假设读者熟悉某种计算机的C和汇编语言，了解编译器的概念，理解编译器的工作原理，同时要求读者的数据结构和算法知识达到一般本科水平，例如，R. Sedgewick所著的《Algorithms in C》(Addison-Wesley，1990）一书中的内容对于理解lcc就足够了。

致谢

感谢众多lcc用户，他们来自于AT&T贝尔实验室、普林斯顿大学和其他地方，他们忍受了许多程序中的错误，并提供了有价值的反馈。感谢Hans Boehm、Mary Fernandez、Michael Gulan、Paul Haahr、Brian Kernighan、Doug McIlroy、Rob Pike、Dennis Ritchie和Ravi Sethi、Ronald Guilmette、David Kristol、David Prosser和Dennis Ritchie在ANSI标准的许多细节及其解释方面提供了非常有价值的信息。David Gay帮助我们改造了PFORT数值计算软件库，以作为lcc代码生成的测试用例，非常有价值。

Jack Davidson、Todd Proebsting、Norman Ramsey、William Waite和David Wall仔细阅读了我们的代码和文字，大大提高了二者的质量。还要感谢Steve Beck，他安装并改进了书中用到的字体；感谢Maylee Noah，他使用Adobe Illustrator制作完成了本书的图片。

Christopher W. Fraser
David R. Hanson

译者的话
中译序
自序

第 1 章 引论 …… 1
1.1 文本程序 …… 1
1.2 如何使用本书 …… 2
1.3 概述 …… 3
1.4 设计 …… 7
1.5 公共声明 …… 11
1.6 语法规范 …… 13
1.7 错误 …… 14
深入阅读 …… 15

第 2 章 存储管理 …… 16
2.1 内存管理接口 …… 16
2.2 分配区的表示 …… 17
2.3 空间分配 …… 18
2.4 空间释放 …… 20
2.5 字符串 …… 20
深入阅读 …… 23
练习 …… 23

第 3 章 符号管理 …… 26
3.1 符号的表示 …… 27
3.2 符号表的表示 …… 29
3.3 作用域的改变 …… 32
3.4 查找和建立标识符 …… 32
3.5 标号 …… 33
3.6 常量 …… 34
3.7 产生的变量 …… 37
深入阅读 …… 38
练习 …… 38

第 4 章 类型 …… 40
4.1 类型表示 …… 40
4.2 类型管理 …… 42
4.3 类型断言 …… 45
4.4 类型构造器 …… 46
4.5 函数类型 …… 48
4.6 结构和枚举类型 …… 49
4.7 类型检查函数 …… 52
4.8 类型映射 …… 56
深入阅读 …… 56
练习 …… 57

第 5 章 代码生成接口 …… 59
5.1 类型度量 …… 59
5.2 接口记录 …… 60
5.3 符号 …… 60
5.4 类型 …… 61
5.5 dag 操作 …… 61
5.6 接口标志 …… 65
5.7 初始化 …… 67
5.8 定义 …… 67
5.9 常量 …… 69
5.10 函数 …… 70
5.11 接口约定 …… 72
5.12 上行调用 …… 73
深入阅读 …… 75
练习 …… 75

第 6 章 词法分析器 …… 77
6.1 输入 …… 77
6.2 单词的识别 …… 83
6.3 关键字的识别 …… 85
6.4 标识符的识别 …… 86
6.5 数字的识别 …… 87
6.6 字符常量和字符串的识别 …… 92
深入阅读 …… 95
练习 …… 95

第 7 章　语法分析 …… 97
7.1　语言和语法 …… 97
7.2　二义性和分析树 …… 98
7.3　自上而下的语法分析 …… 100
7.4　FIRST 和 FOLLOW 集合 …… 102
7.5　编写分析函数 …… 104
7.6　处理语法错误 …… 106
深入阅读 …… 110
练习 …… 111

第 8 章　表达式 …… 112
8.1　表达式的表示 …… 112
8.2　表达式分析 …… 115
8.3　C 语言表达式的分析 …… 117
8.4　赋值表达式 …… 119
8.5　条件表达式 …… 121
8.6　二元表达式 …… 122
8.7　一元表达式和后缀表达式 …… 124
8.8　基本表达式 …… 127
深入阅读 …… 130
练习 …… 130

第 9 章　表达式语义 …… 132
9.1　转换 …… 132
9.2　一元操作符和后缀操作符 …… 136
9.3　函数调用 …… 141
9.4　二元操作符 …… 142
9.5　赋值操作 …… 150
9.6　条件操作 …… 154
9.7　常量折叠 …… 156
深入阅读 …… 165
练习 …… 165

第 10 章　语句 …… 167
10.1　代码的表示 …… 167
10.2　执行点 …… 170
10.3　语句的识别 …… 171
10.4　if 语句 …… 173
10.5　标号和 goto 语句 …… 174
10.6　循环 …… 176
10.7　switch 语句 …… 178
10.8　return 语句 …… 188
10.9　管理标号和跳转指令 …… 191
深入阅读 …… 194
练习 …… 194

第 11 章　声明 …… 196
11.1　转换单元 …… 196
11.2　声明 …… 197
11.3　声明符 …… 206
11.4　函数声明符 …… 210
11.5　结构说明符 …… 215
11.6　函数定义 …… 222
11.7　复合语句 …… 229
11.8　结束处理 …… 236
11.9　主程序 …… 238
深入阅读 …… 240
练习 …… 241

第 12 章　中间代码的生成 …… 243
12.1　消除公共子表达式 …… 244
12.2　构建节点 …… 248
12.3　控制流 …… 250
12.4　赋值语句 …… 256
12.5　函数调用 …… 259
12.6　强制计算顺序 …… 261
12.7　驱动代码生成 …… 263
12.8　删除多次引用的节点 …… 267
深入阅读 …… 272
练习 …… 273

第 13 章　构造代码生成器 …… 275
13.1　代码生成器的组织 …… 276
13.2　接口扩展 …… 277
13.3　上行调用 …… 279
13.4　节点扩展 …… 280
13.5　符号扩展 …… 282
13.6　帧的布局 …… 284
13.7　生成块复制的代码 …… 287
13.8　初始化 …… 289

深入阅读 …… 290
练习 …… 290

第 14 章 选择和发送指令 …… 291

14.1 规范 …… 292
14.2 标记树 …… 294
14.3 化简树 …… 295
14.4 代价函数 …… 302
14.5 测试 …… 303
14.6 发送器 …… 304
14.7 寄存器定位 …… 309
14.8 指令选择的协调 …… 313
14.9 共享规则 …… 314
14.10 编写规范 …… 315
深入阅读 …… 316
练习 …… 316

第 15 章 寄存器分配 …… 318

15.1 组织结构 …… 318
15.2 寄存器状态跟踪 …… 319
15.3 寄存器分配 …… 322
15.4 寄存器溢出 …… 327
深入阅读 …… 334
练习 …… 334

第 16 章 MIPS R3000 代码的生成 … 335

16.1 寄存器 …… 336
16.2 指令的选取 …… 339
16.3 函数的实现 …… 349
16.4 数据的定义 …… 353
16.5 块的复制 …… 359
深入阅读 …… 360
练习 …… 360

第 17 章 SPARC 代码的生成 …… 362

17.1 寄存器 …… 363
17.2 指令的选取 …… 366
17.3 函数的实现 …… 378
17.4 数据的定义 …… 384
17.5 块的复制 …… 386
深入阅读 …… 387
练习 …… 387

第 18 章 X86 代码的生成 …… 389

18.1 寄存器 …… 390
18.2 指令的选取 …… 394
18.3 函数的实现 …… 407
18.4 数据的定义 …… 409
深入阅读 …… 412
练习 …… 412

第 19 章 回顾 …… 413

19.1 数据结构 …… 413
19.2 接口 …… 414
19.3 句法和语义分析 …… 415
19.4 代码生成和优化 …… 416
19.5 调试和验证 …… 416
深入阅读 …… 417

参考文献 …… 419

引　论

编译器可以将源代码翻译成目标机器上的汇编代码或目标代码。可变目标编译器能够针对不同的目标机器生成代码。编译器中与机器有关的部分被独立成模块，针对不同的目标机器，这些模块可以方便地进行替换。

本书描述了一个可变目标的 ANSI C 编译器 lcc，重点介绍其实现。大多数编译器教材注重于编译算法，只附带介绍了实验性的编译器。而本书与其他教材不同，描述了一个完整的 ANSI C 实用编译器，包括针对 3 种目标机器的代码生成器。本书仅介绍了 lcc 用到的编译理论。

1.1　文本程序

本书不仅描述了 lcc 的实现全貌，其本身就是系统的实现。针对文字编程的 noweb 系统根据同一个文本源程序生成了文本和代码。该源程序中包括了交替出现的说明和带标记的代码片段。代码片段按照有利于描述程序的顺序出现，也就是说，本书说明代码的顺序并不与原来 C 语言程序中的一样。程序 noweave 以这种源程序为输入，生成了本书英文版原稿，包括大部分代码和所有文本。程序 notangle 按照书中要求的顺序抽取了所有的代码。

代码片段包括源代码和对其他代码片段的引用。代码片段的定义是以尖括号括起来的标记开始的，例如下列代码：

```
⟨a fragment label 1⟩=                                    1
  sum = 0;
  for (i = 0; i < 10; i++) ⟨increment sum 1⟩

⟨increment sum 1⟩=                                       1
  sum += x[i];
```

该段代码的功能是计算数组 x 的所有元素之和，其中 <increment sum 1> 显示了代码片段是如何被引用的。多个代码片段可以有相同的名字，程序 notangle 可以把这些名字相同的定义连接成一段代码。对于这些连续的程序段定义，程序 noweave 使用 += 而不是 = 来表示：

```
⟨a fragment label 1⟩+=                                   1
  printf("%d\n", sum);
```

程序段定义类似宏定义，程序 notangle 抽取整个程序的过程总是从一个程序段开始的。如果其定义引用了其他程序段，则把引用的程序段扩展进来，如此重复进行。

程序段定义有助于读者通读这些程序。每个程序段的名字的末尾标有一个数字，该数字是这个程序段开始定义的页码。如果没有数字，则表示该程序段未在本书中定义。每个后续的定义都指明了前一个定义，如果有后续定义，还将指明下一个定义。比如 14 指明当前定义的前一个定义在第 14 页，31 指明下一个定义在第 31 页上。这种标志实际上把一个程序段的所有定义连成了一个双向链表，向前指针指向前一个定义，向后指针指向下一个定义。链表中第一个定义的向前指针和最后一个定义的向后指针被省略。这些链表是完全的：如果一个程序段的部分定义在某页中出现，则可以通过这些页码引用找到相关的定义部分。

大多数程序段也指明了该程序段在哪些页中被使用，如上例中 <increment sum> 定义后面的数字 1，表明该程序段在第 1 页使用。下面可以看到，根程序段（定义模块的程序段）以及经常使用的程序段，则没有加这些标志。

程序 notangle 还实现了 C 语言的一个扩展。较长的字符串文本可以分成若干行，每行的末尾用下划线结尾。notangle 可以删除后续行的前导空格，把各行连成一个字符串。第 90 页中 error 的第一个参数就是这种扩展功能的例子。

1.2 如何使用本书

一般来说，应该从前往后阅读本书，但也有几种变通方法：

- 第 5 章介绍了编译器前端和后端的接口，这一章的内容尽可能做到独立。
- 第 13 ~ 18 章介绍了编译器的后端。只要读者了解了编译器前端和后端的接口，仅需简单了解前面的章节，就可以直接阅读这些内容。事实上，许多读者甚至能够替换编译器前端或后端，而不需要对另一部分内容做更多的了解。
- 第 16 ~ 18 章介绍了与 3 种目标机器 MIPS、SPARC、Intel 386 及其后续结构相关的模块。这 3 章相互独立，读者可以阅读其中的任意几章。如果读者阅读了多章的内容，就会发现在内容上有一些重复，但是由于大多数通用的代码已经分解出来并放在第 13 ~ 15 章中介绍，少许重复还是可以容忍的。

本书的部分内容采取自底向上的方式描述 lcc。例如，存储管理、字符串和符号表等章节介绍了一些最底层或接近最底层的函数，理解这些函数不需要太多的相关知识。

本书的另外一些内容则采取自顶向下的方式进行描述。例如，表达式分析、语句和声明等章节，从最顶层开始介绍，采取自顶向下的方式，先给出一些函数和程序段的使用及其功能简介，然后再介绍这些函数和程序段的细节。

本书还有一些地方采取了自顶向下和自底向上相结合的方式进行介绍。也许采取一致的方式进行介绍会更好，但是通常难以做到这一点。与大多数编译器一样，lcc 包括相互之间递归调用的函数。所以，试图完全先介绍调用程序再介绍被调用程序，或者先介绍被调用程序再介绍调用程序，都是不可能的。

有些程序段在读代码之前解释起来比较容易，而有些则在阅读代码后解释更容易一些。如果理解一个代码段有困难，不妨先看看该代码段前后的文字，可能会事半功倍。

lcc 的大多数代码都在教材中出现了，但有些程序段只是加以使用而并未给出定义。在这些程序段中，有些由于篇幅限制被省略了，有些则是因为它们实现的是语言扩展功能、可选的调试帮助或重复的划分。例如，只要了解了处理 C 语言的 for 语句的代码，处理 do-while 语句的代码就不必介绍得太多。唯一全部省略的是解释 lcc 对 C 语言的初始化限制的处理，这是因为它既冗长乏味，又无助于其他知识的理解，所以就省略了这部分内容。只使用而未给出定义的程序段很容易识别：这些程序段名字的后面没有页码。

另外我们还省略了断言。lcc 包含了几百条断言，大多数是关于参数或数据结构假设的断言。其中一个是 assert(0)，它表示程序不应执行到该状态。例如，如果分支语句需要确保测试表达式的所有值都有相应的真正的处理分支，那么，就可以在默认分支中加入 assert(0)。

1.3 概述

lcc 能够把源程序翻译成汇编程序代码。下面的例子说明了这一转换的各个阶段，介绍了 lcc 的主要组成和数据结构。每一阶段都把程序变换成另一种表示方式，例如：预处理后的源程序，单词、树、无环有向图及这些图的序列。例如最开始的源代码是：

```
int round(f) float f; {
        return f + 0.5;  /* truncates */
}
```

round 没有函数原型，所以参数 f 作为 double 类型的数据传送，round 在入口处将其转换成 float。然后 round 将 f 加上 0.5，再把结果截尾转换成整数并返回。

第一个阶段是由 C 的预处理程序进行宏扩展、引入头文件、选择条件编译代码等工作。虽然起源于 UNIX 系统，但目前 lcc 可以在 DOS 和 UNIX 系统下运行。与许多 UNIX 编译器一样，lcc 使用独立的预处理器，并且预处理器作为一个独立的进程执行，但这部分内容不在本书范围之内。我们经常使用 GNU C 编译器的预处理程序。

典型的预处理程序读取上面的源程序并产生如下代码：

```
# 1 "sample.c"
int round(f) float f; {
        return f + 0.5;
}
```

由于本例中没有使用预处理特性，所以预处理程序没有其他效果，只是去除了注解，并增加了一个 # 命令，把文件名和源代码的行号告诉编译器，以便诊断错误时使用。这段例子代码中的起始行号显然为 1，但是，如果源程序中包含多个 #include 指令，预处理后每个被包括进来的文件都用一对 # 指令括起来，指令中都包含各自的行号。

预处理程序工作完成后，编译器马上开始工作，首先由词法分析器（lexical analyzer）或扫描器（scanner）将输入的源程序分解成单词（token），见图 1-1。图中左边一栏是单词编码（token code），该编码是一个小的整数，右边一栏是附加值，附加值也可以为空。例如，关键字 int 的附加值是 inttype 的值，代表整数类型。单个字符构成的单词的编码就是该字符的 ASCII 码值，EOI 表示输入结束。词法分析器输出单词和单词的定义点位置，并处理 # 指令，编译器的其他部分无须再处理这些指令。lcc 的词法分析器将在第 6 章中介绍。

编译器的下一个阶段是根据 C 语言的语法规则对单词串进行分析（parse），同时也分析程序的语义（semantic）正确性。例如，检查加法运算中操作数的类型是否合法，以便进行隐式的类型转换。在上面例子的加法运算中，f 是 float 类型，0.5 是 double 类型，这是合法的组合，所得结果为 double 类型，由于返回类型是 int，求出的和被隐式转换成整数。

示例源程序经过分析阶段后，形成两棵抽象语法树（abstract syntax tree），见图 1-2。树中的每个节点代表一个基本运算。第一棵树将传入的 double 类型的参数转换成 float 类型。节点（INDIR+D）从调用程序中地址为 &f 的单元（ADDRF+P）取出 double 类型的值，节点（CVD+F）将该值转换成 float 值。节点（ASGN+F）把 float 值存入被调用程序的地址为 &f 的单元（ADDRF+P）。

第二棵树实现了例子中唯一的一条语句，并返回一个整数（RET+I）。节点（INDIR+F）从被调用程序中地址为 &f 的单元（ADDRF+P）取出 float 值，节点（CVF+D）将该值转换成 double 值，节点（ADD+D）把该值加上 double 常量 0.5（CNST+D），并将结果截尾成整数（CVD+I）。

```
INT     inttype
ID      "round"
'('
ID      "f"
')'
FLOAT   floattype
ID      "f"
';'
'{'
RETURN
ID      "f"
'*'
FCON    0.5
';'
'}'
EOI
```

图 1-1 示例对应的单词串

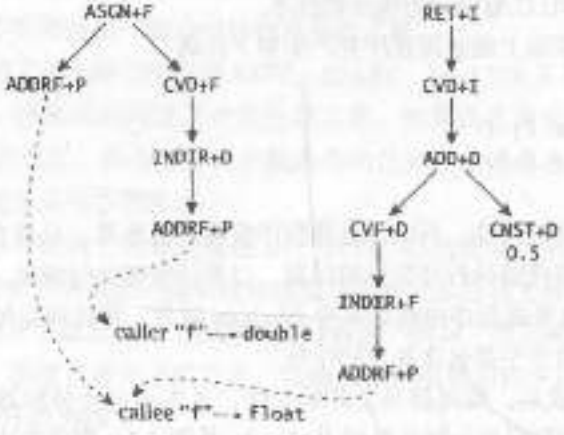


图 1-2 示例对应的抽象语法树

这些树使得隐含在源代码中的一些事实更加清晰。例如，上面的类型转换在源代码中并没有显式给出，但是在 ANSI 标准中是明确规定了的，所以在树中出现了类型转换指令。此外，树中明确给出了所有操作的类型，例如，源代码中的加法操作并没有显式给出类型，但是在树中与其对应的节点具有明显的类型信息。这些语义分析是 lcc 的分析器通过识别输入完成的，我们将在第 7 章到第 11 章进行介绍。

根据图 1-2 中的树，lcc 生成图 1-3 所示的无环有向图（directed acyclic graph，简称 dag），标号为 1 和 2 的 dag 是从图 1-2 中的树转换而来的。操作符标记中不再有 +。把树变换成 dag，使得一些隐含的事实明显化。例如，树中 CNST+D 节点的常量 0.5，在 dag 中成为名字为 "2" 的静态变量的值，CNST+D 操作符被替换成取地址操作符 ADDRGP 和取值操作符 INDIRD。

图 1-3 中的第三个 dag，定义了名字为 "1" 的标号，该标号出现在 round 的末尾，返回指令被翻译成跳转到该标号的指令，其他琐碎的细节不再详述。

在第 12 章中可以看到，从树变换成 dag 时，还能够删除相同的表达式（又称公共子表达式）。例如，若重复引用某 dag 代表的运算结果，则只要把该运算结果放入临时变量中，引用时直接使用这个临时变量即可实现删除公共子表达式。本书介绍的代码生成器就采取了这种措施。

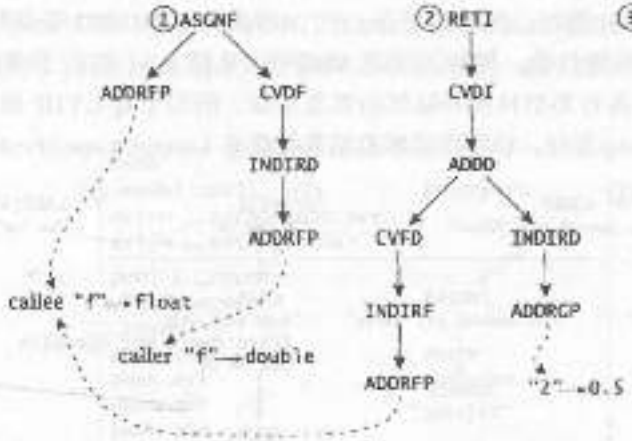


图 1-3 示例对应的 dag

这些 dag 的顺序与图 1-4 所示的代码表的执行顺序相同。列表的第一个入口是 Start，随后的每个入口代表 round 代码的一部分。入口 Defpoint 表示源代码的位置，入口 Blockbeg 和 Blockend 表示 round 代码中复合语句的边界。两个入口 Gen 分别表示执行图 1-3 的 dag 1 和 dag 2。入口 Label 表示执行 dag 3。代码表将在第 10 章和第 12 章中做详细介绍。

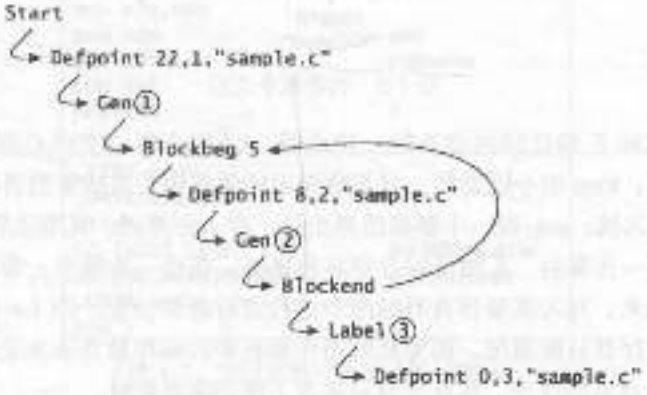


图 1-4 示例对应的代码表

此时，lcc 的与目标机器无关的前端将程序的表示结构传递给后端，由后端把这些结构翻译成目标机器上的汇编代码。我们可以针对特定机器手工编写一个后端程序实现代码的生成，这种代码生成器通常与目标机器紧密相关，如果目标机器发生变化，则需要全部进行更改替换。

本书介绍的代码生成器由表和树文法驱动，在第 13 章到第 18 章中我们将看到树文法能够把 dag 转换成指令序列。这种组织方式使得编译器的后端相对于目标机器具有部分的独立性，使得面对新的目标机器时，我们只需要替换后端的一部分。后端的其他部分也可以移入前端，以期支持针对不同目标机器的代码生成器，但这种方法会使得 lcc 不能方便地使用其他类型的代码生成器，所以我们没有采取这种方法。

代码生成器以对 dag 加注释的方式进行工作。首先为每个节点选择一个汇编代码模板——一条指令或一个操作数。从图 1-5 中可以看到，示例 dag 用 386 及其兼容、后续 X86 平台的汇编代码进行注释。"%n" 表示第 n 个子节点的汇编代码，最左子节点的编号为 0；"%字母" 表示该节点所指向符号表的入口。图中，实线连接了指令，而虚线连接部分指令，如将地址模式

和使用它的指令连接起来。例如，在第一个 dag 中，ASGNF 和 INDIRD 节点标有指令，而两个 ADDRGP 节点标有它们的类作数。同样，节点 ASGNF（见图 1-3）的右子节点 CVDF 消失了，因为 ASGNF 对应的指令具有类型转换和赋值的双重功能，所以节点 CVDF 被删除了。第 14 章介绍了指令选择机制和 lburg 程序，该程序能够根据紧缩规范（compact specification）生成代码。

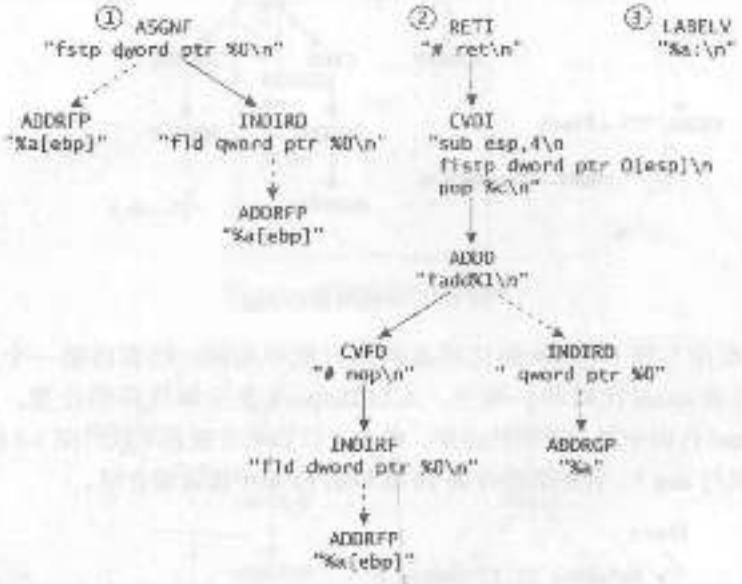


图 1-5 选择指令之后

下面为不了解 X86 汇编代码的读者做一些说明。fld 指令将一个浮点值装入栈；fstp 指令将栈顶的值弹出并存储；fistp 指令也类似，只是将弹出的值截尾变换成整数再存储；faddp 弹出两个值，再将它们的和压入栈；pop 将一个整数值弹出栈，存入寄存器。详细说明参见第 18 章。

在编译器完成下一步骤后，汇编代码会更容易理解。在这一步骤中，编译器将按照指令产生的顺序将节点连接起来，并为需要寄存器的指令进行寄存器的分配。图 1-6 说明了示例程序线性化后的指令序列和寄存器分配情况。因为此时指令模板中的操作数并未完全被替换（这些工作随后进行），图 1-6 有一点虚构成分，放在这里只是为了便于读者理解。

寄存器	汇编模板
eax	fld qword ptr %a[ebp]\n
	fstp dword ptr %a[ebp]\n
	fld dword ptr %a[ebp]\n
	#nop\n
	faddp qword ptr %a\n
	sub esp, 4\nfistp dword ptr 0[esp]\npop %a\n
	#ret\n
	%a:\n

图 1-6 分配寄存器之后

与许多源于 UNIX 系统的编译器一样，lcc 产生汇编代码，还需要和另外的汇编器和连接程序配合使用。与本书的后端配合工作的是：MIPS 和 SPARC 上的汇编器、DOS 下的微软 MASM 6.11 和 Borland 的 Turbo 汇编器 4.0。lcc 会为示例程序生成图 1-7 所示的汇编代码。图中用横线

将代码划分成若干部分，第一部分是为所有的程序都会生成的汇编指示命令样板。第二部分是 round 的入口指令序列，4 条 push 指令用于保存寄存器的值，mov 指令为本次调用 round 建立帧指针。

```
.486
.model small                    boilerplate
extrn __turboFloat:near
extrn __setargv:near
public _round
_TEXT segment
_round:
push ebx                        entry
push esi                        sequence
push edi
push ebp
mov ebp,esp
fld qword ptr 20[ebp]
fstp dword ptr 20[ebp]
fld dword ptr 20[ebp]           body of
fadd qword ptr L2               round
sub esp,4
fistp dword ptr 0[esp]
pop eax
L1:
mov esp,ebp
pop ebp                         exit
pop edi                         sequence
pop esi
pop ebx
ret
_TEXT ends
_DATA segment
align 4
L2 label byte                   initialized data
dd 00H,03fe00000H               & boilerplate
_DATA ends
end
```

图 1-7　为示例程序生成的汇编代码

第三部分代码是由图 1-5 中带注释的 dag 填充了符号表数据后产生的。第四部分是 round 的出口指令序列，恢复入口指令序列保存寄存器的值，并返回到调用程序。L1 是出口指令序列的标号。最后一部分包括初始数据和结束样板（concluding boilerplate）。对于 round 来说，这些数据包括常量 0.5，L2 是变量的地址，该变量初始化成 $000000003fe00000_{16}$，这是双精度常量 0.5 的 IEEE 64 位浮点数表示形式。

1.4　设计

对于 lcc 来说，并没有一个独立的设计阶段。lcc 刚开始只是针对 C 语言的一个子集的编译器，所以其最初的设计目标非常有限，仅定位于针对一般的编译器实现特别是代码生成的教学的需要。即使后来 lcc 演化成了适于实用的 ANSI C 编译器，这一设计目标仍然没有大的改变。

计算的代价越来越低，但是程序员的代价越来越高。如果我们被迫对设计方案做出选择，则通常会选择那种在保证产生令人满意的代码的基础上，能节省开发时间的设计方案。这种选择保证了 lcc 简单、快速，但可能在优化上比其他编译器略为逊色。lcc 面向多目标机器，应尽可能保

持系统的简单性。我们将与目标机器无关的代码独立出来，使得与目标机器相关的代码尽可能简单。我们在设计和实现上花费了相当大的精力，使得 lcc 更易于移植到新的目标机器上。

由于只有两个开发者，时间有限，lcc 必须简单，以节约开发时间并且易于后续完善修改。同时，通过本书，读者可以看到即使编写一个简单的编译器也是很困难的。

lcc 比大多数其他 ANSI C 编译器更小、更快。尽管有时在编译器设计时会忽视编译的速度，但快速仍是广为欣赏的特点，用户经常把速度作为他们使用 lcc 的原因之一。快速编译本身并不是设计的目标，它是追求简单特别是注意编译器中严重影响速度的组成部分的实现的自然结果。lcc 的词法分析（参见第 6 章）和指令选择（参见第 14 章）都特别快，对整个系统的速度贡献最大。

lcc 能够生成相当高效的目标代码，它以产生好的本地代码为设计目标，而其他优化编译器所具有的全局优化，不在 lcc 的设计目标之中。大多数现代的编译器，特别是 CPU 供应商开发的编译器，必须尽可能采取优化措施，以期在性能测试中突出他们的机器。这些编译器很复杂，一般需要数十人组成的小组进行开发。尽管这些高度优化的 C 编译器在启用优化选项时产生的代码比 lcc 产生的代码更高效，但数百名使用 lcc 作为日常主要 C 编译器的开发者发现，对于大多数应用来说，lcc 产生的代码已经足够快了。同时，由于 lcc 本身的速度非常快，也帮助他们节约了许多时间。另外，系统开发者如果需要对 lcc 进行修改，则会发现 lcc 很容易理解。

编译器并非存在于真空中，它必须与预处理程序、连接程序、装配程序、调试程序、汇编程序和操作系统配合工作，而所有这些软件又依赖于目标机器。要想处理好所有这些软件中与目标相关的部分是不现实的。lcc 的设计把这些不利的影响尽可能降到最低点。比如，依赖于目标的代码生成器产生汇编语言代码，并需要汇编程序帮它们转换目标代码；lcc 还依赖于一个独立的预处理程序。这些设计方法并非没有风险，比如，供应商提供的汇编程序存在着一些我们无法控制和回避的错误，给我们带来了很大的困难。

一个更为重要的例子是，必须根据目标机器的约定产生调用代码序列。这对于 lcc 是必须实现的，只有这样才能利用现有的库。当然，提供标准的 ANSI C 库可以简化这方面的任务，但即使 lcc 提供自己的库，它也仍然需要调用与目标相关的例程，比如提供系统调用的例程。调用第三方的库也有这些限制，这种方式日益重要，而且提供的库通常都是目标代码形式。

产生兼容的代码对于 lcc 中与目标无关的前端以及与目标相关的后端的设计都有明显的影响。在第 5 章中可以看到，前端和后端的接口中有很大一部分复杂性是直接源于这些设计上的限制和追求简单性、可变目标性的原则之间的冲突。接口中处理传递和返回结构的机制就是一个例子。

lcc 的前端大约有 9000 行代码，每种与目标相关的代码生成器有 700 行代码，还有约 1000 行与目标无关的后端代码被所有的代码生成器共享。

除了个别例外，lcc 的前端一般都使用成熟的编译技术。如上一节中介绍的，前端进行词法、句法和语义分析，它也删除公共子表达式（参见第 12 章）、进行常量折叠、实现许多简单的与机器无关的转换，以改进本地代码的质量（参见第 9 章），许多改进都是简单地对树进行转换，以期生成更好的代码。前端还对循环、switch 语句进行优化，以产生高效代码（参见第 10 章）。

lcc 的词法分析器和递归下降分析器都是通过手工编写的。使用编译程序构造工具，如利用分析器生成器，可能是一条实现这些模块的更为现代的途径，但是如果使用这些工具，将使得 lcc 依赖于一些特定的工具。相对于 lcc 刚面世时，现在看来这种依赖性已经不算什么问题了，但是，仍然没有发现有什么必要改变这种工作方式。理论上，使用这些工具会使得日后的改进和纠错变得简单，但是，对于 ANSI C 这种标准语言来说，适应变化的需求并不突出，系统的词法和语法错

误也非常少。事实上，lcc 的代码中大约不到 15% 是关于分析的，并且这部分程序的错误率也可以忽略不计。尽管分析在理论上的地位非常突出，但是在 lcc 和其他编译器中只是相对较小的模块，语义分析和代码生成才是主要的模块，占了代码的大部分，而大部分错误也出现于此。

后端是 lcc 最令人感兴趣的模块，原因之一就是它体现了我们旨在提高可变目标能力的设计选择。就可变目标来说，适应将来的变化（每种新的目标机器）的能力是非常重要的。由于代码生成中的错误几乎肯定会发生，改变目标的过程对于改正代码的出错必须是相对简单的。整个 lcc 中有许多针对可变目标性的设计上的考虑，其中最为重要的有两个：

第一，后端使用了代码生成器产生器 Iburg，该产生器可以根据紧缩规范产生代码生成程序。这些规范描述了 dag 如何映射到指令或指令的一部分（参见第 14 章）。由于 Iburg 完成了大多数枯燥的工作，这种方法简化了编写代码生成器、生成优化的本地代码等工作，也有助于避免错误。本书提供了一个 Iburg 的规范，针对新目标的规范可以在此基础上设计，因此，改变目标的过程无须从最基本的工作做起。

第二，只要可能，一些明显依赖于目标机器的函数尽可能移到前端实现。例如，前端完全实现了 switch 语句，通过综合运用移位（shifting）和掩码（masking）的结合实现了对位域的访问。这样做，避免了使用一些针对位域访问和 switch 语句的特殊指令，而支持这些指令的目标机器越来越少；简化改变目标的工作更为重要。前端还能完全实现传递和返回结构，为此采用了一些在依赖于目标机器的调用约定中经常使用的技术。这些功能都可以通过接口选项进行控制，因此，对于一些目标机器，后端可以通过设置这些选项，忽略代码生成在这些方面的考虑。

虽然 lcc 的总体设计目标并没有随着其发展而有大的变化，但是实现这些目标的方法却经常改变。大多数改变是把更多的功能移入前端。switch 语句就是一个例子。在 lcc 的早期版本中，代码生成器的接口包括由后端特殊提供的一些函数，这些函数为 switch 语句产生选择代码。随着新目标的增加，我们发现，这些函数的新版本几乎与原来已有目标中的版本完全相同。这种经验显示，只要对设计做比较简单的改变就可以把所有这些代码移入前端。这样做需要改变所有已有的后端，但是这些改变会移走部分代码，使得将来的新目标的后端变得更简单。

最近所做的最重要的设计变更包括 lcc 的打包方式。先前 lcc 只有一个后端，即针对目标机器 X 的后端和前端结合在一起形成了一个 lcc 实例，该实例在 X 上运行并生成 X 上的代码。大多数 lcc 的后端能够为多个操作系统产生代码。例如，MIPS 后端可以为 MIPS 芯片的计算机产生代码，操作系统包括 DEC 的 Ultrix 和 SGI 的 IRIX。可以构成两个 lcc 的实例。N 个目标机器和 M 个操作系统需要 N×M 个 lcc 实例才能完全满足。每个实例都是根据目标机器和操作系统对原来的模块做少许不同的修改而得到的。即使是较小的 N 和 M，构建 N×M 个编译器也是非常枯燥的，而且容易出错。

在为本书开发的 lcc 版本中，我们改变了代码生成器的接口（将在第 5 章中介绍），使得把所有后端结合到一个程序中成为可能。任意一个 lcc 的实例都是一个交叉编译器（cross-compiler），也就是说，它能为任何目标机器生成代码，而不论该机器上运行的操作系统是什么。通过命令行选项可以选择需要的目标。这种设计方法把所有与目标相关的数据封装在一个结构中，根据选项来选择相应的结构。前端通过这些结构和后端进行通信。这种变化需要修改所有业已存在的后端，但是这种修改只新增了少量的代码。这些努力是非常值得的：现在只需要 M 个 lcc 实例，并且它们都是根据同一组模块构造而成的。同时错误也更容易被发现，通常错误在所有面向不同目标的 lcc 实例中都会出现，可能包括那些专门用于帮助诊断错误的目标。如果节约空间变得非常重要，我们也可以构建一个单目标的 lcc 实例。

从 lcc 的源代码中可以看到数百种设计上的选择，这些选择在实现其他各式各样的软件时也必须加以考虑。lcc 和本书的源代码存放在 noweb 文件中，文本与代码相间，就像本书一样。代码是从 lcc 的模块中抽取出来的。表 1-1 说明了本书各章与程序模块的对应关系，并按照其主要功能将各模块进行了分组。有一些章的对应是一对一的，有一些章对应多个模块，也有较大的模块分在了 3 章中进行介绍。

表 1-1 章和模块的关系

功能	章	头文件	模块
公用定义	1	c.h	
基本部分和数据结构	2		alloc.c string.c
	3		sym.c
	4		types.c
			list.c
代码生成接口	5	ops.h	bind.c
			null.c symbolic.c
I/O 和词法分析	6	token.h	input.c lex.c
			output.c
语法分析和语义分析	7		error.c
	8		expr.c tree.c
	9		enode.c expr.c simp.c
	10		stmt.c
	11		decl.c main.c
			init.c
中间代码生成	12		dag.c
调试与执行剖面			event.c trace.c
			prof.c profio.c
与目标无关的指令选择和寄存器分配	13	config.h	
	13、14、15		gen.c
代码生成	16		mips.md
	17		sparc.md
	18		x86.md

没有对应章节编号的模块在本书中未做介绍。list.c 实现了练习 2.15 描述的列表处理函数，output.c 处理输出函数，init.c 分析和处理 C 的初始化机制，event.c 实现了 8.5 节中描述的事件钩子，trace.c 产生跟踪函数调用和返回的代码，prof.c 和 profio.c 产生执行剖面（profiling）代码。

为方便起见，每章都对其模块的实现进行了说明，程序段的形式如下：

```
⟨M 10⟩=
  #include "c.h"
  ⟨M macros⟩
  ⟨M types⟩
  ⟨M prototypes⟩
  ⟨M data⟩
  ⟨M functions⟩
```

其中 M 是模块名，如 alloc.c。<M macros>、<M types> 和 <M prototypes> 定义了本模块使用的宏、

类型和函数原型声明。<M data 和 <M functions> 包括了外部的和静态的数据及函数的定义（不是声明）。空的段可以省略。给定了模块名，如 alloc.c，notangle 就能抽取该模块，生成上面形式的程序段及其使用的程序段，所有这些程序段组成了该模块的代码。

上面的程序段中没有页码，这是因为这些程序段使用非常频繁，没必要列出一长串的页码，但仍然定义了程序段向前和向后的指针。

1.5 公共声明

每个模块都说明了输出哪些标识符供其他模块使用，输出标识符的声明通过下列程序段给出：<M typedefs>、<M exported macros>、<M exported types>、<M exported data> 和 <M exported functions>，其中 M 是模块名。头文件 c.h 把所有模块中的所有这些程序段进行集中声明，只是在程序段定义中不包括 M，其他定义与每个模块类似。因此所有的模块只要包括 c.h，就可以使用其他模块说明的标识符。与上节中的程序段相似，出于相同的原因，这些程序段也没有给出页码。

```
⟨c.h 11⟩=
  ⟨exported macros⟩
  ⟨typedefs⟩
  #include "config.h"
  ⟨interface 59⟩
  ⟨exported types⟩
  ⟨exported data⟩
  ⟨exported functions⟩
```

头文件 config.h 定义了 <interface> 中使用的与后端相关的类型，参见第 5 章。c.h 定义了 lcc 的全局结构和部分全局常量。

lcc 可以用 ANSI 前的编译器（pre-ANSI compiler）进行编译。目前，还有许多这样的 ANSI 前编译器，因此需要小心地维护与它们的兼容性。ANSI 标准增加了原型，这对于查错非常有帮助，我们可以随时利用这些原型。下面的程序段取自 output.c，它说明了 lcc 是如何工作的：

```
⟨output.c exported functions⟩=                                    U2
  extern void outs ARGS((char *));

⟨output.c functions⟩=                                             U2
  void outs(s) char *s; {
     char *p;
     for (p = bp; (*p = *s++) != 0; p++)
        ;
     bp = p;
     if (bp > io[fd]->limit)
        outflush();
  }
```

函数定义忽略了原型，因此老的编译器进行直接编译。函数声明出现在函数定义之前，把完整的 ANSI 参数类型列表作为参数交给宏 ARGS，ANSI 编译器必须预先定义 __STDC__，如果定义了 __STDC__，ARGS 则产生该类型列表，否则，忽略该声明。

```
⟨c.h exported macros⟩=                                            U2
  #ifdef __STDC__
  #define ARGS(list) list
  #else
  #define ARGS(list) ()
  #endif
```

对于 outs 来说，ANSI 前编译器认为其函数声明是：

```
extern void outs ();
```

而 lcc 和其他 ANSI C 编译器认为其函数声明是：

```
extern void outs (char *);
```

由于 outs 的声明出现在定义之前，ANSI 编译器处理定义时必须像定义中已经包括了原型一样，为所有对 outs 的调用进行参数合法性检查。

ANSI 也会改变可变参数（variadic）函数。宏 va_start 把最后声明的参数作为自己的参数，varargs.h 变成 stdarg.h:

```
⟨c.h exported macros⟩+=                                    11 12
  #ifdef __STDC__
  #include <stdarg.h>
  #define va_init(a,b) va_start(a,b)
  #else
  #include <varargs.h>
  #define va_init(a,b) va_start(a)
  #endif
```

ANSI C 中可变参数函数的定义也有所不同。ANSI C 用定义

```
void print(char *fmt, ...) { ... }
```

代替 ANSI C 前编译器的定义：

```
void print(fmt, va_alist) char *fmt; va_dcl; { ... }
```

lcc 的宏 VARARGS 根据 __STDC__ 的设置，获得 ANSI 参数列表或 ANSI 前参数列表：

```
⟨c.h exported macros⟩+=                                    12 13
  #ifdef __STDC__
  #define VARARGS(newlist,oldlist,olddcls) newlist
  #else
  #define VARARGS(newlist,oldlist,olddcls) oldlist olddcls
  #endif
```

output.c 中 print 的定义说明了如何使用 ARGS、va_init 和 VARARGS。

```
⟨output.c exported functions⟩+=                            11 73
  extern void print ARGS((char *, ...));

⟨output.c functions⟩+=                                      11
  void print VARARGS((char *fmt, ...),
  (fmt, va_alist),char *fmt; va_dcl) {
     va_list ap;

     va_init(ap, fmt);
     vprint(fmt, ap);
     va_end(ap);
  }
```

这个定义略显啰嗦，因为同样的信息用两种不同的形式给出，但是，lcc 很少使用 VARARGS，所以也没有必要做改进。

c.h 也包含了一些一般用途的宏，这些宏在其他地方使用。

```
⟨c.h exported macros⟩+≡                                    ↑12 13↓
  #define NULL ((void*)0)
```

NULL 代表空指针，是不依赖于机器的表达式，在整数和指针的存储空间大小不相同的情况下，f(NULL) 传递一个正确的指针，如果 f 没有原型，f(0) 传递的字节数就可能不对。lcc 产生的代码假设指针作为无符号整数存放。然而，即使某些编译器不支持这一假设，即指针所需的存储空间大于整数的空间，lcc 仍然能够被这些编译器编译。lcc 在调用时使用 NULL，这样即使在指针占的空间比无符号整数大的环境下也不会出错。所以在这些环境中 lcc 仍然可以被编译，也可以作为交叉编译器来使用。

```
⟨c.h exported macros⟩+≡                                    ↑13 73↓
  #define NELEMS(a) ((int)(sizeof (a)/sizeof ((a)[0])))
  #define roundup(x,n) (((x)+((n)-1))&(~((n)-1)))
```

NELEMS(a) 给出了数组 a 的元素的数目，roundup(x,n) 返回离 x 最近的 n 的倍数，n 是 2 的幂。

1.6 语法规范

本书使用文法（grammar）来描述语法（syntax），例如 C 的词法结构、语法和 lcc 的代码生成器产生器 lburg 使用的规范。

文法定义了一个语言，语言由一组句子组成，句子由字母表中的符号组成。这些符号称为终结符（terminal symbol）或单词（token）。文法规则，也称产生式，定义了语言中句子的结构（即语法）。产生式规定了如何从非终结符（nonterminal symbol）通过反复利用规则对非终结符进行替换而生成句子。

产生式说明了可以替换一个非终结符的文法符号序列，产生式的定义由非终结符、冒号、非终结符的替换串组成。如果一个非终结符有多个替换串，则这些替换串分行显示，或者用竖杠“|”分隔。可以出现或不出现的串用方括号（[…]）括起来，可重复 0 次或若干次的串用花括号（{…}）括起来，圆括号用来分组（group）。非终结符用斜体显示，终结符用等宽（typewriter）字体显示。记号“…之一”用于描述由终结符组成的候选序列。如果竖杠、圆括号、方括号和花括号需要作为终结符出现，则必须用单引号括起来以区别于作为产生式定义符号的出现。

例如，产生式

```
expr:
    term { ( + | - ) term }
term:
    factor { ( * | / ) factor }
factor:
    ID
    '(' expr ')'
```

定义了某语言的简单表达式。非终结符有 expr、term 和 factor，终结符有 ID、+、-、*、/、(、)。第一个产生式说明 expr 由 term 和后缀 0 个或若干个 +term 或 - term 组成。第二个产生式类似于乘除法运算说明。最后两个产生式说明了 factor 由一个 ID 或用括号括起来的 expr 组成。最后这两个产生式可以合并成：

factor: ID | '(' *expr* ')'

但是分行显示可以增加可读性。

我们还可以在文法中增加简单的函数调用，增加如下产生式：

factor: ID '(' *expr* { , *expr* } ')'

该产生式说明，factor 可以由 ID 开头，后面跟着用圆括号括起来的一个或多个表达式组成的列表，各表达式之间用逗号分隔。所有关于 factor 的三个产生式可以写成：

factor: ID ['(' *expr* { , *expr* } ')'] | '(' *expr* ')'

该产生式说明，factor 可以是 ID，或者以 ID 开头，后面跟着用圆括号括起来的以逗号分隔的表达式列表，还可以是用括号括起来的 expr。

这种描述语法规范的记号也叫扩充的巴克斯范式（Backus Naur form），简称 EBNF。7.1 节中说明了如何使用 EBNF 推导出语言中的句子。

1.7 错误

lcc 是一个大型复杂的程序。我们常会发现一些错误并加以修正。当我们开始编写本书的时候，一些错误已经暴露出来，随着写作的进行，发现了更多的错误。如果你认为你发现了错误，则可以采取如下措施：

1. 如果你是通过研究本书的代码发现错误的，你可能还没有一个会导致该错误的源程序。那么你首先应创建一个源程序。当然大多数情况下，程序员是在试图编译一个他们自己认为是正确的程序时发现错误的，此时，你可能已经有了一个能导致该错误的源程序。

2. 对源文件进行预处理，保存预处理的结果，抛弃原来的代码。

3. 在保证错误不被隐藏的前提下，尽可能化简你的程序。大多数错误只需要不多于 5 行的代码就可以显示出来。我们需要你化简程序，因为我们只有两个人，而读者却非常多。

4. 确信是在用发布的 lcc 版本处理源文件时出现错误的。如果你修改了 lcc，而该错误仅出现在你的版本中，那么你必须自己追踪错误，即使最后发现错误是由我们造成的，因为我们不能在你的代码上工作。

5. 对你的代码进行注解，说明为什么你认为 lcc 是错误的。如果 lcc 由于断言失败而终止，请告诉我们终止的地方。如果 lcc 崩溃，请尽可能报告最后的调用链。如果 lcc 拒绝一个你认为正确的程序，请告诉我们为什么你认为程序是正确的，并在《The C Programming Language》（Kernighan and Ritchie 1988）一书附录 A 的 ANSI 标准中找到支持你的部分，或者在《C: A Reference Manual》（Harbison and Steele 1991）中找到相应的章节，把页码告诉我们。如果 lcc 生成了不正确的代码，请在注解中附带上错误的汇编代码，并尽可能标出错误的指令。

6. 确信你的错误还没有被改正。最新的 lcc 版本可以在 ftp.cs.princeton.edu 服务器的 pub/lcc 目录下通过匿名 ftp 服务得到。LOG 文件记录了已经改正的错误及改正的时间。如果你报告的错误已被改正，你就可以找到解决方法。

7. 把你的程序以电子邮件的形式发往 lcc-bugs@cs.princeton.edu。请只发送正确的 C 程序，在程序注解中写上所有的评述，这样我们就能对报告进行半自动处理。

深入阅读

大多数编译器教材注重宽度，介绍了各种编译算法，而没有介绍实用的编译器，即日常用来编译实际程序的编译器。本书做了另一种折中，牺牲了教材的宽度，而深入地介绍了一个实用的编译器。对“宽度”和“深度”侧重不同的教材可以相互补充。例如，当你读到 lcc 的词法分析器的时候，可以考虑阅读 Aho，Sethi and Ullman（1986），Fischer and LeBlanc（1991），Waite and Goos（1984），从中学习相关的基础理论和其他方法。其他侧重“深度”的书还有 Holub（1990）及 Waite and Carter（1993）。

Fraser and Hanson（1991b）介绍了 lcc 的一个早期版本，包括 lcc 编译速度和生成的代码运行速度的测试结果。这篇论文还介绍了 lcc 设计上的多种选择及其跟踪（tracing）和产生执行剖面（profiling）的功能。

本章向读者介绍了使用本书所需要的 noweb 的知识，如果你需要深入了解 noweb 的基本设计原理和实现技术，可阅读 Ramsey（1994）。noweb 是 WEB（Knuth 1984）的后续产品。Knuth（1992）收集了他的多篇关于文本编程的论文。

ANSI（American National Standards Institute，Inc. 1990）是 C 程序设计语言的语法和语义的权威规范。与其他 C 编译器不同的是，lcc 只编译 ANSI C，它不支持被 ANSI 委员会遗弃的旧的特征。除了标准，Kernighan and Ritchie（1988）可称为 C 的编程参考。该书面世不久，C 标准就制定完成，因此稍显过时。Harbison and Steele（1991）是在标准制定后出版的，完全按照标准介绍了 C 的语法。Wirth（1977）介绍了 EBNF。

第 2 章 |
A Retargetable C Compiler: Design and Implementation

存储管理

复杂的程序大都需要动态分配内存，lcc 也不例外。C 语言中 malloc 函数分配内存，free 函数释放内存。lcc 也可以使用 malloc 和 free，但是我们优先考虑另一种方法，这种方法更有效、更易于编程实现、更适合在编译器中使用，同时，这种方法也更容易理解。

程序中如果调用了 malloc，就必须通过 free 释放分配的内存，这种释放工作的代价是明显的。更重要的是，程序员很容易忘记释放这些内存，而更糟的是，可能会释放一些正在被程序引用的内存。

在有些应用程序中，大多数释放工作是在同一时间进行的。以窗口系统为例，滚动条、按钮等窗口元素在窗口创建时建立并分配相应的内存空间，而在窗口撤销时释放占用的内存。编译器也是如此，如 lcc，它在处理到函数中的声明、语句和表达式时进行内存空间的分配，在语句和函数结束时释放这些内存。

大多数 malloc 的实现都采取基于对象大小的内存管理算法。而基于对象生存期（lifetime）的算法如果能做到内存及时释放，则更加高效一些。事实上，栈式分配方法可能最为高效，但该方法要求对象的生存期是嵌套的，而这种条件在编译器和许多其他应用程序中并不都能满足。

本章介绍了 lcc 基于对象的生存期的存储管理模式。在该模式中，内存分配比 malloc 更高效，释放的代价也非常小。该模式的真正优点是，它使得编程更简单。分配的代价小，使得编程人员可以采取更加面向应用的算法，而无须因为顾及空间效率而采用复杂的算法。同时，在该模式下，内存分配无须再释放，避免了忘记释放内存的问题。

2.1 内存管理接口

内存是在名为分配区（arena）的区域内进行分配的，整个分配区一起释放。生存期相同的对象在同一个分配区中分配。每个分配区都有一个非负的整数作为它的标识符，分配区在分配和释放时通过这个标识符进行标识：

```
⟨alloc.c exported functions⟩≡
  extern void  *allocate ARGS((unsigned long n, unsigned a));
  extern void deallocate ARGS((unsigned a));
```

许多分配具有下列形式：

```
struct T *p;
p = allocate(sizeof *p, a);
```

其中 T 是 C 语言的结构，a 是一个分配区，p 是一个指针，不论 p 是什么类型的指针，使用 sizeof*p 都可以正确地分配空间。如果采取另一种需要依赖于指针指向的类型的分配方式，那么当代码发生变化时就容易出错。例如：

```
p = allocate(sizeof (struct T), a);
```

只有当 p 确实是一个指向 struct T 的指针时，才是正确的；如果 p 改成指向其他结构的指针，那么上面语句分配空间还是一个 struct T，因此分配就不正确了。第一种方法在效率上并无多大损失，而第二种方法可能造成的错误却是灾难性的。

这种分配方式经常使用，因此我们将其定义成宏：

```
⟨alloc.c exported macros⟩=
  #define NEW(p,a) ((p) = allocate(sizeof *(p), (a)))
  #define NEW0(p,a) memset(NEW((p),(a)), 0, sizeof *(p))
```

宏 NEW 返回一个指向尚未初始化的空间的指针，一般来说，大多数使用者会马上初始化这块空间。宏 NEW0 在分配空间后，通过用 C 语言的 memset 将其初始化为 0。memset 将其第一个参数作为函数结果返回。注意，NEW 和 NEW0 对 p 都只计算一次，所以，调用这两个宏时，即使传递的参数表达式有副作用也无妨，如 NEW(a[i++])。

此外，sizeof 的结果是 size_t 类型，该类型必须是无符号整数的，要求能够表示可能声明的最大对象的大小。实际上，size_t 一般是 unsigned int 和 unsigned long 类型。allocate 的声明使用的是 unsigned long，所以总能表示 sizeof 的结果。

数组是另外一种常见的分配方式，下面的 newarray 在给定的分配区中为数组分配足够的空间，假设数组有 m 个元素，每个元素占用 n 个字节：

```
⟨alloc.c exported functions⟩+=                                    ↑16
  extern void *newarray
     ARGS((unsigned long m, unsigned long n, unsigned a));
```

2.2 分配区的表示

内存管理模块的实现如下：

```
⟨alloc.c 17⟩=
  #include "c.h"
  ⟨alloc.c types⟩
  #ifdef PURIFY
  ⟨debugging implementation⟩
  #else
  ⟨alloc.c data⟩
  ⟨alloc.c functions⟩
  #endif
```

如果 PURIFY 已定义，则采用 malloc 和 free 方式进行内存分配，这种方式适合查错，详情参见练习 2.1。

如前所述，每个分配区都是由一组很大的内存块构成的链表。每个内存块的块头的数据结构定义如下：

```
⟨alloc.c types⟩=                                                  ↓19
  struct block {
     struct block *next;
     char *limit;
     char *avail;
  };
```

紧接在这些分配区结构数据之后，直到 limit 所指的地址，都是可分配的空间。avail 指向块中可分配区域的首地址；地址小于 avail 的空间都是已经分配的区域，从 avail 开始，直到 limit 所指的地址为止，都是继续可分配的空间。next 指向链表中的下一块。程序实现中有一个分配区指针，指向链表中第一个还有可分配空间的内存块。下面可以看到，分配时，内存块动态地加入链表中。

图 2-1 说明了一个分配区在分配了 3 个块之后的情况，阴影部分表示已分配的空间。下面我们将会解释为什么图中第一个分配区中还有未分配的空间。

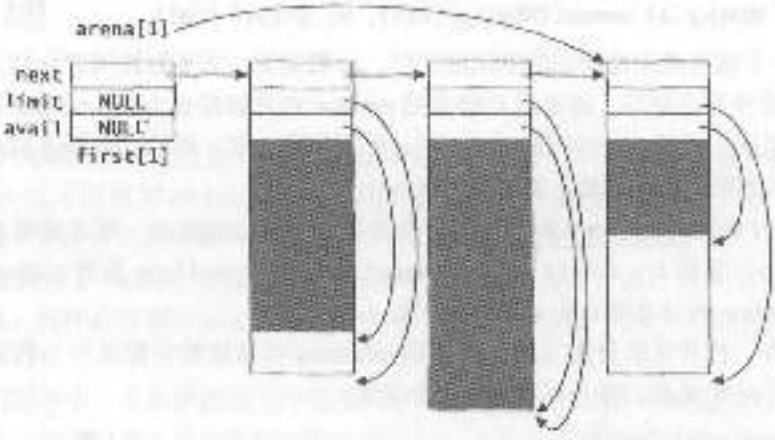


图 2-1 分配区的表示

假设有 3 个分配区，用整数 0、1、2 标识。用户在调用 allocate、deallocate 和 newarray 时，通常可以使用与这些标识等价的符号名（参见 5.12 节）。以分配区的标识数为下标定义一个指针数组，每个指针指向由一个元素组成的链表，分配区内存块长度为 0。在每个分配区中进行的第一次分配，将导致一个新的内存块增加到相应链表的末尾。

```
⟨alloc.c data⟩=                                                    19
  static struct block
      first[] = {  { NULL },  { NULL },  { NULL } },
     *arena[] = { &first[0], &first[1], &first[2] };
```

first 数组在初始化时仅提供元素的数目信息，由于初始化数据不完全，因此，3 个结构的剩余成员被初始化为空指针。虽然本例中只有 3 个分配区，但是只要修改 first 和分配区数组初始化数据的数目，就很容易推广到任意多个分配区。5.12 节介绍了 lcc 如何使用这 3 个分配区。

2.3 空间分配

大多数分配代码是非常琐碎的：按照内存边界对齐原则确定分配空间的大小，据此增加 avail 指针值，并返回该指针原来的值。

```
⟨alloc.c functions⟩=                                               20
  void *allocate(n, a) unsigned long n; unsigned a; {
     struct block *ap;

     ap = arena[a];
     n = roundup(n, sizeof (union align));
     while (ap->avail + n > ap->limit) {
        ⟨get a new block 19⟩
     }
     ap->avail += n;
     return ap->avail - n;
  }
```

⟨alloc.c types⟩+≡ ↑17 19
```
union align {
   long l;
   char *p;
   double d;
   int (*f) ARGS((void));
   };
```

与 malloc 类似，allocate 必须返回一个对齐后的指针，指针指向的空间可以存放任何类型的值。联合 align 给出了宿主机上最小对齐字节数。align 的成员规定了最严格的对齐要求。

上述代码中的 while 循环不断执行，直到 ap 所指的块至少有 n 个字节的可分配空间。对于大多数调用 allocate 的情况，这种块就是由 allocate 的第二个参数所指的分配区指向的块。

如果当前块的可分配空间不能满足分配请求，则必须分配一个新的块。在下面的代码中可以看到，deallocate 不会真正释放一个内存块，而是把释放的块放入空闲块列表，该链表由 freeblocks 指向。allocate 总是从该链表获得空闲的块，如果链表中没有空闲的块，则再分配产生一个新的内存块。

⟨alloc.c data⟩+≡ ↑18
```
static struct block *freeblocks;
```

⟨get a new block 19⟩= 18
```
if ((ap->next = freeblocks) != NULL) {
   freeblocks = freeblocks->next;
   ap = ap->next;
} else
   ⟨allocate a new block 19⟩
ap->avail = (char *)((union header *)ap + 1);
ap->next = NULL;
arena[a] = ap;
```

⟨alloc.c types⟩+≡ ↑19
```
union header {
   struct block b;
   union align a;
};
```

联合 header 保证 ap->avail 总是指向一个对齐后的地址。一旦 ap 指向了一个新的内存块，分配区指针就会调整指向这个新内存块，后续分配就在该块中进行。如果新的内存块来自 freeblocks，有可能太小不足以分配 n 个字节，则需要通过 allocate 的 while 循环继续寻找新的满足要求的内存块。

如果确实需要增加一个内存块，则新增的内存块除了包括块的头、n 个字节的需要分配的空间，还有 10 KB 的空闲待分配的空间：

⟨allocate a new block 19⟩= 19
```
{
   unsigned m = sizeof (union header) + n + 10*1024;
   ap->next = malloc(m);
   ap = ap->next;
   if (ap == NULL) {
```

```
    if (ap == NULL) {
        error("insufficient memory\n");
        exit(1);
    }
    ap->limit = (char *)ap + m;
}
```

如果当前块不能满足某个分配请求，那么当前块末尾剩余的空闲空间都被浪费了。图 2-1 中第一个分配区就说明了这种情况。

newarray 可以通过调用 allocate 简单地实现：

```
⟨alloc.c functions⟩+≡                                    ↑18 20↓
  void *newarray(m, n, a) unsigned long m, n; unsigned a; {
      return allocate(m*n, a);
  }
```

2.4 空间释放

一个分配区释放时，把它的所有内存块都加入空闲块列表中，自己重新初始化成指向单元素列表，而该元素的内存块长度为 0。由于被释放的内存块已经通过其 next 指针连成了一个链，所以只要通过简单的指针操作就可以把整个链中的所有块都加入 freeblocks 中：

```
⟨alloc.c functions⟩+≡                                    ↑20
  void deallocate(a) unsigned a; {
      arena[a]->next = freeblocks;
      freeblocks = first[a].next;
      first[a].next = NULL;
      arena[a] = &first[a];
  }
```

2.5 字符串

字符串可以用于标识符、常量和寄存器等，经常会需要进行字符串比较运算，例如在符号表中搜索一个标识符。

string.c 文件给出了最常用的字符串函数：

```
⟨string.c exported functions⟩=
  extern char * string ARGS((char *str));
  extern char *stringn ARGS((char *str, int len));
  extern char *stringd ARGS((int n));
```

其中每个函数返回一个指向永久占用的字符串的指针，string 函数对以 null 结尾的字符串 str 进行复制，stringn 对字符串 str 的前 len 个字节进行复制，stringd 把整数 n 按十进制转换成字符串形式，并返回。

这些函数对于每个不同的字符串只保存一份副本，所以这些函数返回的两个字符串可以通过比较它们的地址来检查是否相等。这种语义简化了比较操作，节约了空间，stringn 还可以处理内嵌 null 字符的字符串。

函数 string 调用了 stringn，可以用来说明 stringn 的使用方法：

```
⟨string.c functions⟩=                                    21
  char *string(str) char *str; {
     char *s;

     for (s = str; *s; s++)
        ;
     return stringn(str, s - str);
  }
```

stringd 把参数 n 转换成字符串，并存放在一个私有缓冲区中，然后调用 stringn 返回相应的字符串。

```
⟨string.c functions⟩+=                                   21 22
  char *stringd(n) int n; {
     char str[25], *s = str + sizeof (str);
     unsigned m;

     if (n == INT_MIN)
        m = (unsigned)INT_MAX + 1;
     else if (n < 0)
        m = -n;
     else
        m = n;
     do
        *--s = m%10 + '0';
     while ((m /= 10) != 0);
     if (n < 0)
        *--s = '-';
     return stringn(s, str + sizeof (str) - s);
  }
```

由于 ANSI C 允许不同的机器对于负数求模有不同的处理方式，因此代码使用了无符号算术运算。程序中首先把 n 的绝对值赋给 m，如果 n 是最小的负整数，由于任何 2 的补码符号整数都无法表示其绝对值，因此这种情况需要特殊处理。字符串的建立是从后向前的，最后一个数字优先。练习 2.10 说明了为什么局部数组 str 包含 25 个元素。INT_MIN 在头文件 limits.h 中定义。

stringn 对所有不同的字符串进行管理，它把所有字符串存放在一个字符串表（string table）中，保证每个不同的字符串只有一份副本，该表中的任何字符串不会被删除。字符串表是一个长度为 1024 的哈希（hash）表：

```
⟨string.c data⟩=
  static struct string {
     char *str;
     int len;
     struct string *link;
  } *buckets[1024];
```

每个哈希桶（bucket）都保存了一个字符串链表，链表中的字符串的哈希值都是相同的。考虑到字符串中可能包含 null 字符，每个字符串表项还包括了该串的长度（放在 len 域中）。

如果字符串不在字符串表中，则 stringn 把该字符串加入表中，并返回该字符串地址；否则直接返回其地址。

```
⟨string.c functions⟩+≡                                    21
  char *stringn(str, len) char *str; int len; {
     int i;
     unsigned int h;
     char *end;
     struct string *p;

     ⟨h ← hash code for str, end ← 1 past end of str 22⟩
     for (p = buckets[h]; p; p = p->link)
        if (len == p->len) {
           char *s1 = str, *s2 = p->str;
           do {
              if (s1 == end)
                 return p->str;
           } while (*s1++ == *s2++);
        }
     ⟨install new string str 22⟩
  }
```

h 为 str 对应的哈希链。stringn 对该链进行遍历，比较链上长度与 str 相等的字符串是否和 str 完全相同。end 指向 str 最后一个字符之后的字符。

理想的哈希函数最好能将字符串对应的函数值均匀分布在 0 到 NELEMS(buckets)−1 之间，使得每个哈希链具有相等的长度。代码

```
⟨h ← hash code for str, end ← 1 past end of str 22⟩≡      22
  for (h = 0, i = len, end = str; i > 0; i--)
     h = (h<<1) + scatter[*(unsigned char *)end++];
  h &= NELEMS(buckets)-1;
```

是一个相对较好的哈希函数，scatter 是包含 256 个随机数的静态数组，以帮助分布哈希函数值。直接把 end 所指的字符作为数组下标可能有问题，因为该字符也许会被按符号扩展而变成一个负整数，所以，我们先把 end 转换成一个指向无符号字符的指针以避免这种可能的错误。这段代码还将 end 指针置于 str 的最后一个字符后面的字符位置上，就像上面显示的，它用于比较和复制字符串。

传统的建议是，哈希表的大小应该取一个素数。而如果取成 2 的幂，lcc 会工作得更快，因为掩码（masking）运算比求模运算更快。

stringn 把新的字符串存放在永久分配的内存块中，内存块最少有 4 KB。PERM 表示永久的存储分配区。

```
⟨install new string str 22⟩≡                             22
  {
     static char *next, *strlimit;
     if (next + len + 1 >= strlimit) {
        int n = len + 4*1024;
        next = allocate(n, PERM);
        strlimit = next + n;
     }
     NEW(p, PERM);
     p->len = len;
     for (p->str = next; str < end; )
        *next++ = *str++;
```

```
        *next++ = 0;
        p->link = buckets[h];
        buckets[h] = p;
        return p->str;
    }
```

静态变量 next 指向当前内存块的下一个待分配的字节，strlimit 指向当前内存块最后一个字节之后的字节。如果有必要，上面的代码会分配一个新的内存块，并增加一个新的字符串，一边复制 str，一边增加 next，并把该字符串连接到相应的哈希链中。

深入阅读

存储管理是研究热点之一。Knuth（1973a）中的 2.5 节是权威的参考，其中列出了许多技巧，这些技巧有些是针对一般的应用，有些是针对特殊的应用，包括本章介绍的设计方法（Hanson 1990）。另一种有效的方法是“快速匹配”（quick fit）方法（Weinstock and Wulf 1988）。快速匹配分配方法对于最常分配的 N 种字节数目，分别维护 N 个待分配链表，通常这些块的字节数是较小而连续的，比如是 8 – 128 字节且为 8 的倍数，分配方法非常简单快速：根据你分配的字节数，从相应的链表中取第一个块即可。内存块释放时，再将其加入相应链表的链首。如果请求分配的字节数不是最常用的 N 个数字，则按照其他算法处理，如“最先匹配”（Knuth 1973a）。

对于 lcc 基于分配区的分配算法来说，其优点之一是，内存分配操作无须一一对应的释放操作，只要一个释放操作，就可以把通过多个分配操作得到的内存一起释放，使得编程更简单。垃圾收集方法将这种优点进一步发展，垃圾收集程序通过定期检查程序中的所有指针，查找出正在使用的内存，并将未被使用的空间释放。Appel（1991）和 Wilson（1994）研究了垃圾收集算法。通常需要程序设计语言、编译器和运行时系统（run-time system）对垃圾收集程序进行支持，使得后者能够定位可访问的内存。但是，也有一些算法能够不依赖于这些支持，Boehm and Weiser（1988）介绍了这样一种针对 C 的算法，该算法采取了一种保守的方法：所有貌似指针的元素都被当成指针。因此，收集程序会把一些不可访问的内存当成可以访问的，使得系统繁忙，但这种方法要比采取相反的方法好。

采取哈希函数把所有字符串存放到一个字符串表中，每个字符串只保存一个副本，这种方法是多年来在编译器和相关程序语言实现中被使用的一种模式，但是很少有介绍它的文档。例如，该方法用于 SNOBOL4（Griswold 1972），使得把字符串作为关联表的关键字变得更快速、更简单。还有一些相关的技巧，把字符串存放在单独的字符串空间，但是并未避免相同的字符串保留多个副本。这种方法可以简化某些字符串操作，如获取子串、字符串连接等（Hansen 1992；Hanson 1974；McKeeman，Horning and Wortman 1970）。

Knuth（1973b）是介绍哈希方法的权威。Aho，Sethi and Ullman（1986）介绍了哈希函数及其在编译器中的应用。

练习

2.1 使用 C 的库函数 malloc 和 free 修改 allocate 和 deallocate。

2.2 在 lcc 中，如果要在多个算法和设计中做出选择，唯一客观的途径就是实现这些算法和方法，并对其效果进行评测。将 lcc 用于编译其自身的源代码是一种很好的测量标准。请对基于分配区的算法和练习 2.1 所实现的采用 malloc 和 free 的方法的性能进行评测。

2.3 重新定义NEW，使其尽可能在分配区内部进行分配，即只有当分配区的空间不够时才调用allocate。测试其效果。为了实现内部分配方法，必须输出分配区的数据结构。

2.4 当allocate建立了一个新块时，提供了一个好时机，如果这个新块和该分配区中前一个块相邻，就可以将二者连成一个更大的块。实现并测试这种方法的改进效果。

2.5 当allocate从freeblocks中取走一个块时，有可能这个块太小。对分配程序进行插桩，看看这种情况是否经常发生，这个问题值得修改吗？

2.6 说明deallocate在分配区列表只有长度为0的块时也能正确工作。

2.7 deallocate从不真正释放块，如通过调用free。在某些输入情况下，lcc的分配区会临时膨胀，而已分配的块不会再被利用。修改deallocate以释放块，而不是将其加入freeblocks中。这种改变能使lcc运行得更快吗？

2.8 为lcc实现一个保守的垃圾收集程序，或者利用一个已有的垃圾收集程序。Boehm and Weiser (1988) 介绍的收集程序是公开的。大多数这类分配程序在进行分配时会调用收集程序或部分收集程序，因此，你可以去掉deallocate，或者将其定义为空的宏，并修改allocate以调用相应的分配函数。

2.9 通过stringn在字符串表中建立的字符串不会被删除。这种特点会带来问题吗？调用stringn看字符串表的大小分布情况。如果表格太大，如何修改字符串接口，使其允许删除字符串？

2.10 stringd将其参数格式化成字符串，存入长度为25的字符数组str中。请解释为什么25对于当前lcc运行并产生代码的计算机已经足够了。

2.11 许多传给stringd的整数很小，比如在−100到100之间。这些整数对应的字符串在编译时就可以进行预分配，stringd和stringn只要返回指向这些字符串的指针而无须再分配。实现这种优化措施，能使lcc运行加快吗？

2.12 stringn用较大的内存块来存放字符串中的字符，而不会为每个字符串都调用allocate。修改stringn使得它为每个字符串调用一次allocate。比较这两种方法在时间和空间上的差别，并解释这些差别。

2.13 stringn的哈希表的大小是2的幂，这种方法经常遭到反对。尝试将其大小设成某个素数并看看效果。请设计一种更好的哈希函数并考察其结果。

2.14 stringn比较字符串采取的是内联代码，而不是调用memcmp函数。请用调用memcmp代替内联代码并考察结果。为什么我们要采取内联方法？

2.15 lcc大量使用了指针链环列表，list.c模块的实现可视为使用分配宏（allocation macro）的例子，list.c输出了列表元素的类型和3个列表操作函数：

```
⟨list.c typedefs⟩=
  typedef struct list *List;

⟨list.c exported types⟩=
  struct list {
     void *x;
     List link;
  };

⟨list.c exported functions⟩=
  extern List append ARGS((void *x, List list));
  extern int  length ARGS((List list));
  extern void *ltov  ARGS((List *list, unsigned a));
```

List 保存了 0 或多个元素。每个元素存放在 list 结构的 x 域中。List 指向列表中最后一个 list 结构，空的 List 定义为空列表。append 函数把包含 x 的节点加入 list 列表的末尾并返回 list。length 函数返回列表中元素的数目。linv 函数把 list 中的 n 个元素复制到 a 所指的分配区中以空元素结尾的指针数组，释放列表结构并返回该数组。数组中有 n+1 个元素，包含一个空元素。请实现这种列表模块。

第3章

A Retargetable C Compiler: Design and Implementation

符号管理

符号表是编译器保存信息的中心库，编译器的各部分通过符号表进行交互，并访问符号表中的数据——符号。例如，词法分析器把标识符加入标识符表中，由分析器添加这些标识符的类型信息，代码生成器则为符号表的各入口添加与目标相关的信息，如局部变量和参数的寄存器分配信息。符号表也保存有关标号、常量和类型的信息。

符号表把各种名字映射到符号集合。常量、标识符和标号都是名字，不同的名字有不同的属性。例如，作为局部变量名的标识符包括变量的类型、该变量在其声明所在过程的栈帧中的位置以及存储类型；作为结构成员名的标识符则具有完全不同的属性集，包括成员的类型、所在的结构及其在结构中的位置。

符号信息都集中保存到符号表中，由符号表模块对符号和符号表进行管理。

符号管理不仅要处理符号本身，还必须遵循 ANSI C 标准规定的作用域（scope）或可见性（visibility）规则。标识符的作用域是指在程序文本中该标识符可见的部分，即在这些程序部分中，该标识符可以用于表达式等成分。C 的作用域可以嵌套，一个标识符的可见范围是从该标识符的声明点开始，直到其声明所在的复合语句或参数列表结束。在所有复合语句或参数列表以外声明的标识符具有文件作用域，其可见范围从声明点开始直到其所在的源文件结束。

内层声明的标识符 X 会隐藏外层声明的标识符 X，下面的程序说明了这种效果；行号是为了说明方便，并非程序的组成部分。

```
1  int x, y;
2  f(int x, int a) {
3     int b;
4     y = x + a*b;
5     if (y < 5) {
6        int a;
7        y = x + a*b;
8     }
9     y = x + a*b;
10 }
```

第 1 行声明了全局变量 x 和 y，它们的作用域开始于第 1 行，直到第 10 行。但是第 2 行参数 x 的声明打断了全局变量 x 的作用域，参数 x 和 a 的作用域起始于第 2 行，直到第 9 行。参数 a 的作用域也被第 6 行的局部变量 a 的声明打断。第 4 行的表达式中出现的各标识符按照 C 的作用域规则被绑定到各自的声明上。若使用 x:n 表示在第 n 行声明的标识符 x，那么第 4 行的表达式中的 y 绑定为 y:1，x 绑定为 x:2，a 绑定为 a:2，b 绑定为 b:3。第 7 行表达式中标识符的绑定情况类似，只是 a 应绑定为 a:6。

上面第 2 行 x 的声明和第 6 行 a 的声明使得外层声明的相同标识符的作用域出现了空洞。例如，a:6 的作用域是第 6 ~ 8 行，这正好是 a:2 的作用域的空洞，其作用域为第 2 ~ 5 行和第 9 ~ 10 行。符号管理功能必须能够处理这类情况。

在大多数语言中，如 Pascal，对于标识符有一个名字空间，也就是说，对于各种用途的标识符都在一个统一集合中，在程序的任何地方，对于给定的名字只有一个标识符可见。

ANSI C 按照用途对标识符的名字空间进行分类：语句标号、标记（tag）、成员、一般标识符。标记表示结构、联合和枚举。对于标号、标记和一般标识符分别有 3 个独立的名字空间，而对于每个结构和联合，其成员都有独立的名字空间。

在每个名字空间中，对于每个给定的名字，在程序的任何地方都只有一个可见的标识符。但是，如果标识符处于不同的名字空间中，在程序的任何地方就允许有多个标识符可见。下面的特意设计的、易于混淆的程序说明了这种情况：

```
1  struct list { int x; struct list *list; } *list;
2  walk(struct list *list) {
3     list:
4        printf("%d\n", list->x);
5        if ((list = list->list) != NULL)
6           goto list;
7  }
8  main() { walk(list); }
```

第 1 行声明了 3 个名为 list 的标识符，这些标识符从声明点开始都是可见的。这里 list 分别作为结构标记、结构域名和结构变量名。作为标记和变量名的 list 具有文件作用域；从技术上说，作为结构域名的 list 也如此，但它只能通过域引用运算符“.”和“->”来使用。第 2 行声明了参数 list，它的作用域是第 2 – 7 行。第 3 行声明了标号 list，它具有函数作用域（function scope），在函数 walk 的范围内都可见。第 4 – 8 行的 list 取决于使用的名字空间，第 4 行使用的是一般标识符，第 5 行前两个 list 使用的是一般标识符，最右边的 list 使用的是结构 list 的成员的名字空间，第 6 行使用的是标号名字空间，第 8 行再次使用了一般标识符的名字空间。

粗略地说，对于不同的名字空间有独立的符号表，每个符号表各自处理作用域。lcc 还使用独立的符号表来处理无作用域集合（unscoped collection），如常量。

3.1 符号的表示

内存分配和字符串模块可以在 lcc 之外独立使用，而符号表模块则是与 lcc 密切相关的，该模块对 lcc 的符号和符号表进行管理，并实现 ANSI C 规定的作用域规则和名字空间机制。

对于符号本身来说，与符号表模块没有什么关系，符号表模块只需要符号与作用域相关的属性，如符号名等。因此，很简单，符号结构中只需要包括名字和所有其他属性：

```
⟨sym.c typedefs⟩≡                                              28
  typedef struct symbol *Symbol;

⟨sym.c exported types⟩≡                                        28
  struct symbol {
     char *name;
     int scope;
     Coordinate src;
     Symbol up;
     List uses;
```

```
    int sclass;
    ⟨symbol flags 37⟩
    Type type;
    float ref;
    union {
        ⟨labels 34⟩
        ⟨struct types 50⟩
        ⟨enum constants 52⟩
        ⟨enum types 52⟩
        ⟨constants 35⟩
        ⟨function symbols 226⟩
        ⟨globals 206⟩
        ⟨temporaries 270⟩
    } u;
    Xsymbol x;
    ⟨debugger extension⟩
};
```

结构中在联合 u 之前的各域，对于所有符号表中的所有符号都是通用的，大多数符号表函数只读写 name、scope、src、up 和 uses 等域。针对常量和标号的函数则需要使用联合 u 和 <symbol flags>（后文详述）中的部分域。其他域则实现了与特定类符号相关的属性，这些属性由符号表模块的使用者进行初始化和修改。

name 域通常是符号表的关键域。对于标识符和表示类型名的关键字，该域保存了源代码中使用的名字；对于编译器生成的标识符，如无标记的结构，name 保存的是由数字组成的字符串。

scope 域说明了符号是常量、标号、全局变量、参数或局部变量：

```
⟨sym.c exported types⟩+≡                                    ↑27
  enum { CONSTANTS=1, LABELS, GLOBAL, PARAM, LOCAL };
```

在第 k 层中声明的局部变量，其 scope 域等于 LOCAL+k。

src 域指明了该符号在源代码中定义点的位置，如变量声明的所在位置，其类型 Coordinate 精确指明了符号在何处定义：

```
⟨sym.c typedefs⟩+≡                                          ↑27 29
  typedef struct coord {
    char *file;
    unsigned x, y;
  } Coordinate;
```

file 域指明了包含该定义的文件的名字，y 和 x 分别指明了定义出现的行号以及在定义行中的字符位置。

up 域把符号表中所有符号连成了一个链表，最后载入符号表的那个符号为链首。从后向前遍历该链表可以访问到当前作用域内的所有符号，包括因为嵌套内层作用域中的标识符声明而被隐蔽的符号。这种功能有助于编译后端产生调试器所用的符号表信息。

lcc 有一个选项，通过该选项可以保存每个符号的所有使用信息。如果该选项被设置，则 uses 域保存一个 Coordinate 列表，说明了符号的使用情况；如果该选项未被设置，uses 为 null。参见练习 3.4。

sclass 域说明了符号的扩展存储类型，可以是 AUTO、REGISTER、STATIC 或 EXTERN。sclass 还可以取值 TYPEDEF 表示 typedef，取值 ENUM 表示枚举常量，对于常量和标号该域未被使用，其值为 0。

type 域保存了变量、函数、常量、结构、联合和枚举等的类型信息。

对于变量和标号来说，ref 域保存了它们被引用的粗略次数，10.3 节解释了如何进行这种近似计数。

u 域是一个联合，为标号、结构和联合类型、枚举标识符、枚举类型、常量、函数、全局和静态变量、临时变量等提供了附加信息。对于每个符号来说，<symbol flags> 是一位属性标记。x 域和 <debugger extension> 包括一些仅由编译后端处理使用的域，如为变量分配的寄存器、为调试器产生数据流供所需要的信息。

Symbol 和 Coordinate 的类型定义说明了贯穿 lcc 的一种约定：大写字母开头的名字用于表示结构的类型名或指向结构的指针，而结构标记则采用小写字母。因此，Coordinate 表示 struct coord 的类型名，Symbol 表示 struct symbol* 的类型名。

3.2 符号表的表示

符号表只由符号表模块操作，该模块输出一个符号表类型 Table 和各种符号表实例：

```
⟨sym.c typedefs⟩+≡                                    ▲28 35▼
  typedef struct table *Table;

⟨sym.c exported data⟩≡                                  32▼
  extern Table constants;
  extern Table externals;
  extern Table globals;
  extern Table identifiers;
  extern Table labels;
  extern Table types;
```

上述符号表的子集实现了 ANSI C 的 3 种名字空间。identifiers 保存一般标识符；externals 存放声明为 extern 的标识符，用于警告外部标识符声明冲突；globals 是 identifiers 表的一部分，保存了具有文件作用域的标识符。

编译器定义的内部标号保存在 labels 中，类型标记存放在 types 中。

这些符号表都是由哈希表组成的列表，每个符号表对应一个作用域：

```
⟨sym.c types⟩≡
  struct table {
     int level;
     Table previous;
     struct entry {
        struct symbol sym;
        struct entry *link;
     } *buckets[256];
     Symbol all;
  };
  #define HASHSIZE NELEMS(((Table)0)->buckets)
```

对于 Table 类型的一个值，如 identifiers，指向了一个 table 结构，该结构将某作用域内的符号保存在一个哈希表中，level 域的值指明作用域；buckets 域是一个指针数组，每个指针指向一个哈希链；previous 域指向外层作用域对应的 table。

哈希链的每个入口保存了一个 symbol 结构和一个指向链中下一入口的指针。如果要查找一个符号，则根据关键字计算哈希函数值，找到相应的哈希链，然后通过遍历该链找到相应的符号；如果未发现该符号，则通过 previous 域在外层作用域的入口中进行查找。

在每个表结构中，all 域指向由当前及其外层作用域中所有符号组成的列表的头，该列表是通过 symbol 的 up 域连接起来的。

除了表 labels 外，符号表模块负责对所有向外导出的 Table 进行初始化：

```
⟨sym.c data⟩=                                                    32
  static struct table
     cns = { CONSTANTS },
     ext = { GLOBAL },
     ids = { GLOBAL },
     tys = { GLOBAL };
  Table constants   = &cns;
  Table externals   = &ext;
  Table identifiers = &ids;
  Table globals     = &ids;
  Table types       = &tys;
  Table labels;
```

globals 指向作用域为 GLOBAL 的标识符表，identifiers 指向当前作用域的表。types 将在第 4 章中介绍。funcdefn 会为每个函数创建一个标号表。

内层嵌套作用域的表都是动态创建的，并且与相应外层的表进行链接：

```
⟨sym.c functions⟩=                                               31
  Table table(tp, level) Table tp; int level; {
     Table new;

     NEW0(new, FUNC);
     new->previous = tp;
     new->level = level;
     if (tp)
        new->all = tp->all;
     return new;
  }
```

函数被编译完成后，所有动态分配的表都被释放，因此，动态表是在 FUNC 分配区中进行分配的。

图 3-1 说明了 lcc 在编译到本章第一个示例程序（第 26 页）的第 7 行时，涉及 identifiers 的 4 个符号表的情况。图中所有 entry 结构只显示了它们的 symbol 中 name 和 up 域以及它们的 link 域。实线显示了下列域信息：previous 域，把表连接起来；buckets 和 link 域，将各入口连接在一起；还有 name 域。虚线从 all 域和 symbol 的 up 域发出。

all 域初始化成指向外层表的 list，因此，从 table 的 all 域开始，就可以访问所有作用域的所有符号。foreach 函数使用该功能扫描一个表，并对指定作用域中的所有符号执行给定函数。

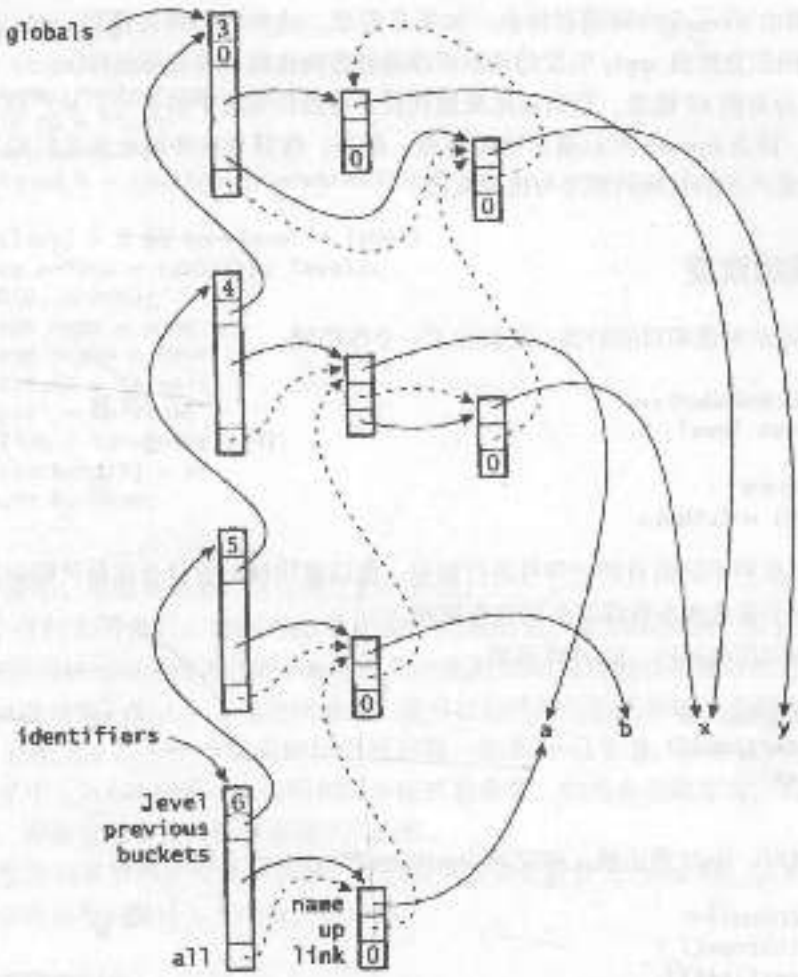


图 3-1 第 26 页示例程序编译到第 7 行时符号表的情况

```
⟨sym.c functions⟩+=                                          30 32
  void foreach(tp, lev, apply, cl) Table tp; int lev;
  void (*apply) ARGS((Symbol, void *)); void *cl; {
     while (tp && tp->level > lev)
        tp = tp->previous;
     if (tp && tp->level == lev) {
        Symbol p;
        Coordinate sav;
        sav = src;
        for (p = tp->all; p && p->scope == lev; p = p->up) {
         src = p->src;
         (*apply)(p, cl);
      }
      src = sav;
   }
}
```

while 循环查找与作用域对应的表，如果找到了，foreach 函数就把每个符号的定义位置保存在全局变量 src 中，并为该符号调用 apply 函数。cl 是一个指针，它指向与调用相关的数据

closure，该数据由 foreach 的调用者提供，如果有需要，这些数据将传递给 apply 函数以便其访问。src 中保存的信息使得 apply 引发的诊断程序能够指向正确的源程序坐标。

for 便环返历表的 all 链表，直到链尾或遇到较小层数作用域中的符号。从严格意义上说，all 并不是必需的，因为 foreach 可以遍历哈希链表，但是，按照与哈希地址无关的顺序为每个符号调用 apply，会使产生的代码的顺序与机器无关。

3.3 作用域的改变

全局变量 level 的值和对应的表一起表示了一个作用域：

```
⟨sym.c exported data⟩+≡                                    29 38
  extern int level;

⟨sym.c data⟩+≡                                             30
  int level = GLOBAL;
```

由于作用域按照不同的目的对符号进行划分，所以作用域的数目会比源代码中复合语句的数目多。比如，对于常量和参数都有不同的作用域。

进入一个新的作用域时，level 将递增：

```
⟨sym.c functions⟩+≡                                        31 32
  void enterscope() {
     ++level;
  }
```

退出作用域时，level 将递减，相应的 identifiers 和 types 表也随之撤销。

```
⟨sym.c functions⟩+≡                                        32 33
  void exitscope() {
     rmtypes(level);
     if (types->level == level)
        types = types->previous;
     if (identifiers->level == level) {
        ⟨warn if more than 127 identifiers⟩
        identifiers = identifiers->previous;
     }
     --level;
  }
```

当前作用域对应的表只有在必需时才会被创建。在 C 中，声明新符号的作用域是非常少的，所以采取懒惰的表分配方法可以节省时间，但是 exitscope 函数必须检查层数以决定是否有表需要撤销。对于在撤销的作用域中定义的带标记的类型，rmtypes 将从其类型缓冲中删除，参见 4.2 节。

3.4 查找和建立标识符

install 函数为给定的 name 分配一个符号，并把该符号加入与给定作用域层数相对应的表中，如果需要，还将建立一个新表。该函数返回一个指向符号的指针。

```
⟨sym.c functions⟩+≡                                    ↑32 33
  Symbol install(name, tpp, level, arena)
  char *name; Table *tpp; int level, arena; {
     Table tp = *tpp;
     struct entry *p;
     unsigned h = (unsigned)name&(HASHSIZE-1);

     if (level > 0 && tp->level < level)
        tp = *tpp = table(tp, level);
     NEW0(p, arena);
     p->sym.name = name;
     p->sym.scope = level;
     p->sym.up = tp->all;
     tp->all = &p->sym;
     p->link = tp->buckets[h];
     tp->buckets[h] = p;
     return &p->sym;
  }
```

name 存放了字符串，根据其地址可以计算它的哈希值。

tpp 是一个指向表的指针。如果 *tpp 指向某作用域的表，如 identifiers，并且目前没有与参数 level 给定的作用域相对应的表，则 install 将先为参数 level 给定的作用域分配一个表，并更新 *tpp；然后 install 分配一个入口，将该项清零，最后初始化符号的某些域，并把该入口加入哈希链表中。level 必须为 0 或不小于该表的作用域层数，如果 level 为 0，则表示 name 应该建立在 *tpp 表示的表格中。install 接受一个指明相应分配区的参数，如果有函数原型，则使其中的符号可以永久保存，即使它们是在内层作用域中声明的。

lookup 函数实现在表中查找一个名字，查找的关键字是符号的 name 域。如果找到了，该函数将返回一个指向符号的指针，否则返回空指针。

```
⟨sym.c functions⟩+≡                                    ↑33 34
  Symbol lookup(name, tp) char *name; Table tp; {
     struct entry *p;
     unsigned h = (unsigned)name&(HASHSIZE-1);

     do
        for (p = tp->buckets[h]; p; p = p->link)
           if (name == p->sym.name)
              return &p->sym;
     while ((tp = tp->previous) != NULL);
     return NULL;
  }
```

在代码中，内层循环扫描哈希链，外层循环扫描外层作用域。字符串模块保证当且仅当两个字符串完全相同时，它们才是同一个副本的，所以字符串的比较非常简单。

3.5 标号

符号表模块还提供管理标号和常量的函数。这些管理函数与 lookup 和 install 相似，但是不涉及作用域管理。查找标号和常量时，如果有必要，就会建立这些标号和常量，因此查找总会成功。查找的关键字是联合 u 中标号和常量特有的域。

编译器产生的标号和源程序中标号的内部表示都采取整数。函数 genlabel 通过累加计数器产生一个整数：

```
⟨sym.c functions⟩+≡                                    33 34
  int genlabel(n) int n; {
     static int label = 1;

     label += n;
     return label - n;
  }
```

genlabel 也可以用于产生唯一的、匿名的名字，如产生一个临时变量的名字。

对于每个标号将分配一个符号，u.l.label 保存了标号：

```
⟨labels 34⟩≡                                           28
  struct {
     int label;
     Symbol equatedto;
  } l;
```

如果两个或更多个内部标号指向相同的位置，则这些标号的 equatedto 域指向其中一个标号。

源程序中的每个标号都有相应的一个内部标号，这些内部标号和编译器产生的其他标号都保存在 labels 表中。对于每个函数都会建立一个这样的表（参见 11.6 节），并由 findlabel 函数进行管理。findlabel 函数的输入参数是一个标号数，并返回该标号对应的符号，如果需要，则会建立该符号、进行初始化并通知编译后端。

```
⟨sym.c functions⟩+≡                                    34 35
  Symbol findlabel(lab) int lab; {
     struct entry *p;
     unsigned h = lab&(HASHSIZE-1);

     for (p = labels->buckets[h]; p; p = p->link)
        if (lab == p->sym.u.l.label)
           return &p->sym;
     NEW0(p, FUNC);
     p->sym.name = stringd(lab);
     p->sym.scope = LABELS;
     p->sym.up = labels->all;
     labels->all = &p->sym;
     p->link = labels->buckets[h];
     labels->buckets[h] = p;
     p->sym.generated = 1;
     p->sym.u.l.label = lab;
     (*IR->defsymbol)(&p->sym);
     return &p->sym;
  }
```

generated 是一位二进制位域 <symbol flags>，表示一个产生的符号。对于产生的这些符号名字，某些编译后端可以利用特殊的格式以避免表在连接上的混乱。

3.6 常量

对表达式中作为类作数出现的编译时常量的引用，常处理成一个指向该常量的符号的指针。

这些符号保存在 constants 表中。与 labels 一样，这类表也是与作用域无关的：所有常量符号的 scope 域都取值 CONSTANTS。

常量的实际值用下面联合的实例表示：

```
⟨sym.c typedefs⟩+≡                                    ▲29
  typedef union value {
      /* signed */ char sc;
      short ss;
      int i;
      unsigned char uc;
      unsigned short us;
      unsigned int u;
      float f;
      double d;
      void *p;
  } Value;
```

按照常量的类型，其值存放在相应域中，例如，整数存放在 i 域，无符号字符存放在 uc 域，如此等等。

在 constants 表中建立一个常量时，该常量的类型（Type）存放在符号结构的 type 域，我们对 C 的数据类型进行了编码（参见第 4 章），其值存放在 u.c.v 中：

```
⟨constants 35⟩=                                       28
  struct {
      Value v;
      Symbol loc;
  } c;
```

在某些目标机器上，有些常量，主要是浮点数，不能存储在指令中，所以编译器产生一个静态变量并将其初始化成该浮点常量的值。对于这些常量，u.c.loc 指向了产生的变量所对应的符号。总体上，type 和 u.c 域保存了常量的所有已知信息。

对于每个常量来说，constants 中只保存了一个实例，例如，“hello world”在程序中出现了三次，所有这三次引用都指向了表中的同一个符号。constant 函数实现了在常量表中查找给定类型和值的常量，如果需要，将在表中增加该常量，函数返回一个指向符号的指针。常量不会从表中删除。

```
⟨sym.c functions⟩+≡                                    ▲34 36▼
  Symbol constant(ty, v) Type ty; Value v;{
      struct entry *p;
      unsigned h = v.u&(HASHSIZE-1);

      ty = unqual(ty);
      for (p = constants->buckets[h]; p; p = p->link)
          if (eqtype(ty, p->sym.type, 1))
              ⟨return the symbol if p's value == v 36⟩
      NEW0(p, PERM);
      p->sym.name = vtoa(ty, v);
      p->sym.scope = CONSTANTS;
```

```
    p->sym.type = ty;
    p->sym.sclass = STATIC;
    p->sym.u.c.v = v;
    p->link = constants->buckets[h];
    p->sym.up = constants->all;
    constants->all = &p->sym;
    constants->buckets[h] = p;
    (announce the constant, if necessary 36)
    p->sym.defined = 1;
    return &p->sym;
}
```

unqual 函数返回类型（Type）的未限定的（unqualified）形式，即去掉类型中的 const 和 volatile。eqtype 函数用于测试类型是否相同（参见 4.7 节）。如果 v 出现在表中，则返回指向其符号的指针；否则，在表中新增一个符号并初始化。name 域存放了 vtoa 函数返回的常量的字符串表示。

vtoa 返回的值只对整型和指向常量的指针有用，对于其他类型，返回的字符串并不能可靠地描述对应的数值。常量的查找是基于它们实际的值，而不是它们的字符串表示，因为有些浮点数不能直接自然地表示成字符串。例如，常量表达式（double）(float) 0.3 可以将 0.3 截尾成一个与机器相关的值，这种效果就不能用有效的字符串表示出来。

类型运算符决定了联合中被比较的域：

```
⟨sym.c macros⟩=
  #define equalp(x) v.x == p->sym.u.c.v.x

⟨return the symbol if p's value == v 36⟩=                    35
  switch (ty->op) {
  case CHAR:     if (equalp(uc)) return &p->sym; break;
  case SHORT:    if (equalp(ss)) return &p->sym; break;
  case INT:      if (equalp(i))  return &p->sym; break;
  case UNSIGNED: if (equalp(u))  return &p->sym; break;
  case FLOAT:    if (equalp(f))  return &p->sym; break;
  case DOUBLE:   if (equalp(d))  return &p->sym; break;
  case ARRAY: case FUNCTION:
  case POINTER:  if (equalp(p))  return &p->sym; break;
  }
```

constant 调用 defsymbol，通知编译后端这些常量可以出现在 dag（无环有向图）中：

```
⟨announce the constant, if necessary 36⟩=                    36
  if (ty->u.sym && !ty->u.sym->addressed)
    (*IR->defsymbol)(&p->sym);
```

对于基本类型，如整型和浮点类型，只有对 addressed 标志进行了测试，lcc 完成这些配置后才能出现在 dag 中。参见 4.2 节和 5.1 节。

在编译前端和后端中都有大量的整型常量。intconst 封装了建立和通知整型常量的功能。

```
⟨sym.c functions⟩+=                                          35 37
  Symbol intconst(n) int n; {
    Value v;

    v.i = n;
```

```
    return constant(inttype, v);
}
```

3.7 产生的变量

编译的前端会为各种目的产生许多局部变量。例如，可以产生静态的变量，用于保存超出一行的常量（如字符串）和 switch 语句的跳转表等，也可以产生一些局部变量用于函数传递和返回结构，以及用来保存条件表达式的结果和 switch 语句的测试值。genident 根据给定的类型、存储类别和作用域，产生一个标识符并初始化：

```
⟨sym.c functions⟩+≡                                          ↑36 37↓
  Symbol genident(scls, ty, lev) int scls, lev; Type ty; {
     Symbol p;

     NEW0(p, lev >= LOCAL ? FUNC : PERM);
     p->name = stringd(genlabel(1));
     p->scope = lev;
     p->sclass = scls;
     p->type = ty;
     p->generated = 1;
     if (lev == GLOBAL)
        (*IR->defsymbol)(p);
     return p;
  }
```

```
⟨symbol flags 37⟩≡                                           38  28↓
  unsigned temporary:1;
  unsigned generated:1;
```

其中 name 是由数字组成的字符串，generated 标志被设置为 1。参数和局部变量在其他地方通知编译后端，而产生的全局变量在此就通过调用后端的 defsymbol 接口函数通知后端。IR 指向一个数据结构，该数据结构连接前端和某个具体的后端，5.11 节说明了这种连接是如何初始化的。

临时变量是另外一类产生的变量，它们都具有 temporary 标志：

```
⟨sym.c functions⟩+≡                                          ↑37 37↓
  Symbol temporary(scls, ty, lev) Type ty; int scls, lev; {
     Symbol p = genident(scls, ty, lev);

     p->temporary = 1;
     return p;
  }
```

编译后端有时也需要产生临时变量，比如为了腾空寄存器。后端由于不知道类型系统，所以不能直接调用 temporary 函数。newtemp 接受一个类型后缀，通过调用 btot 将该后缀映射为相应的类型，再利用该类型调用 temporary。

```
⟨sym.c functions⟩+≡                                          ↑37
  Symbol newtemp(sclass, tc) int sclass, tc; {
     Symbol p = temporary(sclass, btot(tc), LOCAL);
```

```
        (*IR->local)(p);
        p->defined = 1;
        return p;
    }
```

⟨*symbol flags* 37⟩+≡ ↑37 137 28
```
    unsigned defined:1;
```

调用 newtemp 发生在代码生成的时候，如果像前端的临时变量那样进行通知，则为时已晚。因此，newtemp 调用 local 来通告它们。标志 defined 在通知完后端之后被置成 1。

深入阅读

lcc 的符号表模块仅仅实现了 C 必需的功能。针对其他语言还需要实现更多的功能，例如，在块结构化（block-structured）语言中允许过程嵌套，在同一时刻可能有多组参数和局部变量可见。最的面向对象的语言和具有显式作用域指令的语言中有更多的作用域。有些语言在同一时刻需要多个独立的符号表。

Fraser and Hanson（1991b）介绍了 lcc 符号表模块的发展情况。

Knuth（1973b）的 6.4 节给出了哈希方法的详细分析，介绍了好的哈希函数的特点。有许多建议有助于设计好的哈希函数，例如，Aho，Sethi and Ullman（1986）的 7.6 节。

练习

3.1 为符号表的哈希入口表设计一种更好的哈希函数。可使用 Aho，Sethi and Ullman（1986）的 7.6 节中的一个函数，用该函数能提高 lcc 的运行速度吗？

3.2 lcc 不会删除 constants 表中的入口，这种方法在什么时候会带来问题？设计并实现一种解决方法，检查该方法的效果。该方法带来的效果与其开销相比是否值得？

3.3 lcc 最初使用一个哈希表处理所有的符号表（Fraser and Hanson，1991b）。使用这种方法，映射到相同哈希桶的所有符号存储在哈希链中，并按照 scope 依降序排列。lookup 函数只需要搜索一个哈希表。采用这种方法，install 和 enterscope 函数非常容易，但是 exitscope 函数会变得更复杂，它必须扫描所有的哈希桶，删除当前作用域层的符号。现在 lcc 采取的设计方法在有些计算机上运行较快，但在某些机器上就不一定比原来的方法更快了。

请实现原来的设计方案，注意保证能够正确访问全局变量。该比较哪种方法更易理解，哪种方法更快。

3.4 sym.c 提供了数据和函数，为调试器生成标识符和符号表信息的交叉引用列表。-x 选项使 lcc 把每个符号的 uses 域置成一个指向 Coordinate 的指针列表，表示该符号的使用情况。sym.c 提供函数：

⟨*sym.c exported functions*⟩≡ ↓38
```
    extern void use ARGS((Symbol p, Coordinate src));
```

该函数把 src 加入 p->uses，还提供函数：

⟨*sym.c exported data*⟩+≡ ↑32
```
    extern List loci, symbols;
```

⟨*sym.c exported functions*⟩+≡ ↑38
```
    extern void locus ARGS((Table tp, Coordinate *cp));
```

loci 和 symbols 分别保存了指向 Coordinate 和 Symbol 的指针。symbols 表中的每个入口是一个链表的头，该链表由在 loci 对应的源代码位置上可见的符号组成，从入口的符号出发，通过 up 域可以访问所有在该点可见的符号。locus 函数把 tp->all 和 cp 加入 symbols 和 loci 中。tp->all 指向最近加入 *tp 表的符号，即当前可见符号列表的头。请实现 use 和 locus 函数，每个函数不超过 5 行代码。

类　　型

C 程序中具有丰富的数据类型，包括声明中显式定义的类型和用作表达式中间类型的导出类型。例如，下面的赋值语句包含 3 种不同类型：

```
int *p, x;
*p = x;
```

x 是一个存放整数的单元的地址，因此 x 的地址的类型（即它的左值）是“指向一个整数的指针”。x 的值的类型（即它的右值）是整数（从声明中可以得到）。同样，p 的左值的类型是指向一个整数的指针的指针，p 的右值的类型是指向一个整数的指针，*p 的类型是整数。lcc 在编译赋值语句时必须处理所有这些类型。

lcc 实现了类型表示和一组该表示之上的函数，本章将对此进行介绍。函数包括创建类型的类型构造函数（type constructor）以及测试类型的类型断言函数（type predicate）。lcc 还必须实现类型检查（type checking），确保声明和表达式遵守语言制定的规则。类型检查使用本章介绍的断言函数，详细信息参见第 9 章和第 11 章。

4.1　类型表示

C 语言的类型通常用英文的前缀形式描绘，所谓前缀形式，就是指类型操作数在类型操作符之后出现。例如，int *p 声明 p 是一个指向 int 的指针，就是 C 类型 int * 的前缀表示，指针是操作符，int 是操作数。同样，char *(*strings)[10] 将 strings 说明为：

```
一个指针，指向
  一个大小为 10 的数组，每个数组元素是一个
    指针，指向
      字符
```

操作数在它们各自的操作符下以阶梯方式缩进。

表示这种前缀类型规范的方法有很多。例如，一些老的 C 编译器使用位串来表示，类型操作符和基本类型用若干位编码。位串表示非常紧凑并易于操作，但是一般会对基本类型和操作符的数目做一定限制，并且不能拥有表示大小的数据，例如不能表示数组的大小。

lcc 通过能反映类型的前缀规范的链接结构来表示类型。类型节点定义如下：

```
⟨types.c typedefs⟩=                                          50
  typedef struct type *Type;

⟨types.c exported types⟩=                                    50
  struct type {
    int op;
    Type type;
    int align;
    int size;
```

```
    union {
        (types with names or tags 41)
        (function types 48)
    } u;
    Xtype x;
};
```

op 域存放整型的类型操作符编码，type 域存放类型操作数。操作符可以是下列全局枚举常量值：

```
CHAR        LONG        ARRAY       FUNCTION
INT         ENUM        STRUCT      CONST
UNSIGNED    FLOAT       UNION       VOLATILE
SHORT       DOUBLE      POINTER     VOID
```

操作符 CHAR、INT、UNSIGNED、SHORT、LONG 和 ENUM 定义了整数类型（integral type）；FLOAT、DOUBLE 定义了浮点类型（floating type）。这些类型都可以看作算术类型（arithmetic type）。除 ENUM 类型外，这些类型都没有操作数。ENUM 类型的操作数是与其兼容的整数类型，即枚举标识符的类型。对于 lcc 而言，枚举标识符的类型总是整型（int）（将在随后的 4.6 节解释）。

操作符 ARRAY、STRUCT、UNION 表示聚合类型（aggregate type）。STRUCT 和 UNION 没有操作数，它们的域存放在结构或联合标记的附加符号表入口中。ARRAY 的操作数是数组元素的类型（element type）。POINTER 和 FUNCTION 分别定义指针类型（pointer type）和函数类型（function type），它们的操作数分别指明被引用类型（referenced type）和返回类型（return type）。操作符 CONST 和 VOLATILE 说明限定类型（qualified type），它们的操作数就是类型的未限定形式。CONST 加上 VOLATILE 也是一个类型操作符，它说明一个既是 const 又是 volatile 的类型。VOID 操作符表示 void 类型，没有操作数。

align 域和 size 域以字节为单位给出类型的对齐字节数和该类型对象的大小。第 5 章的代码生成接口中规定，size 必须是 align 的整数倍。编译器后端必须为变量分配空间，使得变量地址是变量类型的对齐字节数的整数倍。

x 域扮演的角色与它在符号结构 symbols 中的一样：编译器后端通过定义 Xtype，向类型结构中加入与目标机器相关的域。这一机制通常用于对调试器的支持。

输出对 type 节点的声明可以展示 Type 的内部信息，后端可以读取 size 和 align 域，并且读写 x 域。lcc 约定，后端只允许检查这些域，而前端可以访问 Type 的所有域。

op、type、size 和 align 域给出了处理类型所需的大部分信息。对带有名字或标记的未限定类型，包括固有（built-in）类型、结构和联合类型以及枚举类型，u.sym 域指向符号表入口，符号表入口将给出更多与类型有关的信息。

```
(types with names or tags 41)≡                                    41
    Symbol sym;
```

符号表入口给出了类型名。此外，如果该类型的常量可以作为指令的一部分，u.sym->addressed 的值为 0。u.sym->type 反向指向类型自身，可用于将标记映射成类型。每个结构、联合和枚举类型都有一个符号表入口，每种基本类型对应一个符号表入口，所有指针类型共用一个符号表入口。这些入口都出现在 types 表中，参见 4.2 节中的详细描述。使用这种表示，sym.c 中定义的函数可用于管理 types。

可以用带括号的前缀形式描述类型，这种形式与前面介绍的英文前缀形式相似。例如，在MIPS上int类型表示为：

```
(INT 4 4 ["int"])
```

第一个4是对齐字节数，第二个4是大小，["int"]表示名为int的类型的指向符号表入口的指针。其他类型的描述相似，例如：

```
(POINTER 4 4 (INT 4 4 ["int"]) ["T*"])
```

表示指向int的指针。类型名T*表示用于所有指针类型对应的唯一的符号表入口。

如果随后在理解某些问题时不需要对齐字节数、大小和符号表指针，它们可以从解释中省略(但不能从代码中省略)。例如，本节开始给出的类型可以简写为：

```
(INT)
(POINTER (INT))
(POINTER (ARRAY 10 (POINTER (CHAR))))
```

最后一行描述的类型是指向一个数组的指针，该数组有10个元素，每个元素是指向char的指针。数组类型的size域总是保存数组的实际大小。元素的个数可以用数组的大小除以元素类型的大小而获得。因此，表示含有10个int元素的数组类型的更精确描述是：

```
(ARRAY 4 40 (INT 4 4 ["int"]))
```

但是，lcc通常约定描述为(ARRAY 10(INT))。不完全类型(incomplete type)是指类型大小未知，size域等于0的类型。不完全类型由省略了大小的声明得来，例如：

```
int a[];
extern struct table *identifiers;
```

不透明的指针(opaque pointer)，例如指向lcc的table结构的指针，是不完全类型。如果不完全类型的大小对识别不完全类型非常重要，在解释中就需要给出size域。

4.2 类型管理

类型检查的基本操作之一是判断两个类型是否等价。如果任意类型只有一份副本，等价测试就可以简化。这与字符串的比较是一个道理：任意字符串只保留一份副本，串的比较就很简单。

正如stringn函数处理字符串，type函数处理类型。type管理类型表typetable：

```
⟨types.c data⟩≡                                          44
  static struct entry {
     struct type type;
     struct entry *link;
  } *typetable[128];
```

typetable中的每个entry结构都保存一个类型。type函数在typetable中搜索指定类型或创建一个新的类型：

```
⟨types.c functions⟩≡                                     44
  static Type type(op, ty, size, align, sym)
     int op, size, align; Type ty; void *sym; {
     unsigned h = ⟨hash op and ty 43⟩&(NELEMS(typetable)-1);
     struct entry *tn;
```

```
    if (op != FUNCTION && (op != ARRAY || size > 0))
        ⟨search for an existing type 43⟩
    NEW(tn, PERM);
    tn->type.op = op;
    tn->type.type = ty;
    tn->type.size = size;
    tn->type.align = align;
    tn->type.u.sym = sym;
    memset(&tn->type.x, 0, sizeof tn->type.x);
    tn->link = typetable[h];
    typetable[h] = tn;
    return &tn->type;
}
```

type 总是为函数类型和不完全的数组类型创建新类型。创建新类型时，type 初始化参数指定的域，清空 x 域，将类型加入相应的哈希链中，并返回新类型。

type 在搜索 typetable 时，利用类型操作符和操作数类址的异或值作为哈希值，搜索相应的哈希链，寻找具有相同操作符、操作数、大小、对齐字节数和符号表入口的类型。

```
⟨hash op and ty 43⟩=                                    42
  (op^((unsigned)ty>>3))

⟨search for an existing type 43⟩=                       43
  for (tn = typetable[h]; tn; tn = tn->link)
     if (tn->type.op    == op   && tn->type.type  == ty
     &&  tn->type.size  == size && tn->type.align == align
     &&  tn->type.u.sym == sym)
        return &tn->type;
```

typetable 在初始化时，只具有固有类型和 void* 类型，这些类型也是 14 个全局变量的值：

```
⟨types.c exported data⟩=
  extern Type chartype;
  extern Type doubletype;
  extern Type floattype;
  extern Type inttype;
  extern Type longdouble;
  extern Type longtype;
  extern Type shorttype;
  extern Type signedchar;
  extern Type unsignedchar;
  extern Type unsignedlong;
  extern Type unsignedshort;
  extern Type unsignedtype;
  extern Type voidptype;
  extern Type voidtype;
```

前端使用这些变量引用特定的类型，避免为已知存在的类型搜索 typetable。typeInit 函数初始化这些全局变量和 typetable。

```
(types.c functions)+≡                                    ↑42 44
  void typeInit() {
     (typeInit 44)
  }
```

与 5.1 节将要描述的一样，每种基本类型用类型度量（type metric）来刻画。类型度量也可以看成三元式，描述类型大小、最小对齐字节数以及该类型的常量是否能出现在 dag（无环有向图）中。三元式的结构包括 size、align 和 outofline 三个域。

```
(typeInit 44)≡                                           44 ↓ 44
  #define xx(v,name,op,metrics) { \
     Symbol p = install(string(name), &types, GLOBAL, PERM);\
     v = type(op, 0, IR->metrics.size, IR->metrics.align, p);\
     p->type = v; p->addressed = IR->metrics.outofline; }
  xx(chartype,     "char",          CHAR,     charmetric);
  xx(doubletype,   "double",        DOUBLE,   doublemetric);
  xx(floattype,    "float",         FLOAT,    floatmetric);
  xx(inttype,      "int",           INT,      intmetric);
  xx(longdouble,   "long double",   DOUBLE,   doublemetric);
  xx(longtype,     "long int",      INT,      intmetric);
  xx(shorttype,    "short",         SHORT,    shortmetric);
  xx(signedchar,   "signed char",   CHAR,     charmetric);
  xx(unsignedchar, "unsigned char", CHAR,     charmetric);
  xx(unsignedlong, "unsigned long", UNSIGNED,intmetric);
  xx(unsignedshort,"unsigned short",SHORT,    shortmetric);
  xx(unsignedtype, "unsigned int",  UNSIGNED,intmetric);
  #undef xx
```

无符号整数类型和有符号整数类型具有相同的操作符、大小和对齐字节数，但具有不同的符号表入口，因此要为它们构建不同的类型。同样，lcc 假设 long 和 int、long double 和 double 具有相同结构，但每种都是单独的一种类型。测试某个类型是否表示长整数类型，只要将该类型与 longtype 进行比较。IR 指向后端提供的接口记录（参见 5.11 节）。类型 void 没有度量：

```
(typeInit 44)+≡                                          ↑44 46 ↓ 44
  {
     Symbol p;
     p = install(string("void"), &types, GLOBAL, PERM);
     voidtype = type(VOID, NULL, 0, 0, p);
     p->type = voidtype;
  }
```

typeInit 函数将符号表入口装载到 3.2 节定义的 types 表中。types 表包含了用标识符或标记命名的所有类型。基本类型由 typeInit 装载，且不会被删除。exitscope 函数将结构、联合、枚举类型的符号表入口从 types 中删除时，与结构、联合、枚举标记相关联的类型也必须从 typetable 中删除。exitscope 调用 rmtypes(lev) 从 typetable 中删除那些 u.sym->scope 大于或等于 lev 的类型：

```
(types.c data)+≡                                         ↑42 46 ↓
  static int maxlevel;

(types.c functions)+≡                                    ↑44 46 ↓
  void rmtypes(lev) int lev; {
     if (maxlevel >= lev) {
```

```
        int i;
        maxlevel = 0;
        for (i = 0; i < NELEMS(typetable); i++) {
            ⟨remove types with u.sym->scope >= lev 45⟩
        }
    }
}
```

maxlevel 的值是 typetable 中所有在符号表中有相关入口的类型的 u.sym->scope 的最大值。在调用 rmtypes 时，大多数情况下不存在某个符号表入口的 scope 大于或等于 lev，这时，rmtypes 可以利用 maxlevel 避免扫描 typetable。删除类型后需要重新计算 maxlevel：

```
⟨remove types with u.sym->scope >= lev 45⟩= 45
  struct entry *tn, **tq = &typetable[i];
  while ((tn = *tq) != NULL)
      if (tn->type.op == FUNCTION)
          tq = &tn->link;
      else if (tn->type.u.sym && tn->type.u.sym->scope >= lev)
          *tq = tn->link;
      else {
          ⟨recompute maxlevel 45⟩
          tq = &tn->link;
      }
⟨recompute maxlevel 45⟩= 45
  if (tn->type.u.sym && tn->type.u.sym->scope > maxlevel)
      maxlevel = tn->type.u.sym->scope;
```

对于函数类型来说，它们的 u.sym 被其他域覆盖，而没有 u.sym 域，一般应特殊处理。数组和限定类型也没有 u.sym 域，最后一个子句处理了这种情况。

4.3 类型断言

typeInit 初始化的全局变量可用于说明特殊类型或者测试特殊类型。例如，如果类型 ty 等于 inttype，ty 是 int 类型。下面列出了用宏实现的类型断言（以 is 开头），通过检查特定的类型操作符来测试类型集合。大多数操作作用于未限定类型，未限定类型可通过调用 unqual 获得：

```
⟨types.c exported macros⟩= 45
  #define isqual(t)     ((t)->op >= CONST)
  #define unqual(t)     (isqual(t) ? (t)->type : (t))

⟨types.c exported macros⟩+= 45 50
  #define isvolatile(t) ((t)->op == VOLATILE \
                      || (t)->op == CONST+VOLATILE)
  #define isconst(t)    ((t)->op == CONST \
                      || (t)->op == CONST+VOLATILE)
  #define isarray(t)    (unqual(t)->op == ARRAY)
  #define isstruct(t)   (unqual(t)->op == STRUCT \
                      || unqual(t)->op == UNION)
  #define isunion(t)    (unqual(t)->op == UNION)
  #define isfunc(t)     (unqual(t)->op == FUNCTION)
  #define isptr(t)      (unqual(t)->op == POINTER)
```

```
#define ischar(t)      (unqual(t)->op == CHAR)
#define isint(t)       (unqual(t)->op >= CHAR \
                     && unqual(t)->op <= UNSIGNED)
#define isfloat(t)     (unqual(t)->op <= DOUBLE)
#define isarith(t)     (unqual(t)->op <= UNSIGNED)
#define isunsigned(t)  (unqual(t)->op == UNSIGNED)
#define isdouble(t)    (unqual(t)->op == DOUBLE)
#define isscalar(t)    (unqual(t)->op <= POINTER \
                    || unqual(t)->op == ENUM)
#define isenum(t)      (unqual(t)->op == ENUM)
```

类型操作符的值在 token.h 中定义，可见上述断言将产生预期的结果。

4.4 类型构造器

type 函数可以构造任意类型，其他函数封装了对 type 的调用，以构造特定类型。例如，ptr 函数创建指针类型：

```
⟨types.c functions⟩+≡                                    44 46
  Type ptr(ty) Type ty; {
     return type(POINTER, ty, IR->ptrmetric.size,
        IR->ptrmetric.align, pointersym);
  }
```

给定类型 ty，ptr 返回 (POINTER ty)。与指针类型相应的符号表入口在初始化时赋给了 pointersym，类型 void* 调用 ptr 完成初始化：

```
⟨types.c data⟩+≡                                         44
  static Symbol pointersym;

⟨typeInit 44⟩+≡                                          44  44
  pointersym = install(string("T*"), &types, GLOBAL, PERM);
  pointersym->addressed = IR->ptrmetric.outofline;
  voidptype = ptr(voidtype);
```

ptr 创建指针类型，deref 间接访问指针，即 deref 返回引用类型。给定类型 (POINTER ty)，deref 返回 ty：

```
⟨types.c functions⟩+≡                                    46 47
  Type deref(ty) Type ty; {
     if (isptr(ty))
        ty = ty->type;
     else
        error("type error: %s\n", "pointer expected");
     return isenum(ty) ? unqual(ty)->type : ty;
  }
```

与下面将要介绍的某些类型构造器一样，deref 在操作数非法时产生错误。从技术上说，这些测试是类型检查的一部分，不属于类型构造，将这部分测试放到类型构造中，可以简化类型检查代码，避免疏漏。deref 的最后一行处理指向枚举的指针：间接访问枚举指针必须返回与它关联的未限定整数类型。unqual 的定义已经在 4.3 节中介绍了。

array(ty, n, a) 函数创建类型（ARRAY n ty），结果类型的对齐字节数为 a，如果 a 等于 0，结果类型的对齐字节数就是 ty 的对齐字节数。array 还检查非法操作数。

```
⟨types.c functions⟩+=                                                  46 47
  Type array(ty, n, a) Type ty; int n, a; {
      if (isfunc(ty)) {
         error("illegal type 'array of %t'\n", ty);
         return array(inttype, n, 0);
      }
      if (level > GLOBAL && isarray(ty) && ty->size == 0)
         error("missing array size\n");
      if (ty->size == 0) {
         if (unqual(ty) == voidtype)
            error("illegal type 'array of %t'\n", ty);
         else if (Aflag >= 2)
            warning("declaring type 'array of %t' is _
               undefined\n", ty);
      } else if (n > INT_MAX/ty->size) {
         error("size of 'array of %t' exceeds %d bytes\n",
            ty, INT_MAX);
         n = 1;
      }
      return type(ARRAY, ty, n*ty->size,
         a ? a : ty->align, NULL);
   }
```

C 语言中，不允许出现函数数组、void 类型的数组以及除 GLOBAL 外其他任意作用域层次的不完全数组（长度为 0 的数组）。array 不能表示超过 INT_MAX 字节的数组的大小，因此 array 也禁止大小超过 INT_MAX 字节的数组。如果 lcc 编译选项 -A 出现了两次，则将 Aflag 置为 2，表示 lcc 要对非 ANSI 用法报警。因此，当 Aflag 等于 2 时，如果声明的不完全数组的类型是不完全类型，array 应输出警告信息。格式代码 %t 打印相应类型参数的英文描述，参见练习 4.4。

在许多情况下，数组类型“退化”（decay）为指向数组元素类型的指针，例如当数组作为形参的类型时。atop 函数实现这种“退化”：

```
⟨types.c functions⟩+=                                                  47 47
  Type atop(ty) Type ty; {
     if (isarray(ty))
        return ptr(ty->type);
     error("type error: %s\n", "array expected");
     return ptr(ty);
  }
```

qual 和 unqual 分别构造和析构（deconstruct）限定类型。给定类型 ty，qual 检查非法的操作数，创建（CONST ty），（VOLATILE ty）或（CONST+VOLATILE ty）。

```
⟨types.c functions⟩+=                                                  47 48
  Type qual(op, ty) int op; Type ty; {
   if (isarray(ty))
      ty = type(ARRAY, qual(op, ty->type), ty->size,
         ty->align, NULL);
   else if (isfunc(ty))
      warning("qualified function type ignored\n");
```

```
        warning("qualified function type ignored\n");
    else if (isconst(ty)    && op == CONST
    ||       isvolatile(ty) && op == VOLATILE)
        error("illegal type '%k %t'\n", op, ty);
    else {
        if (isqual(ty)) {
            op += ty->op;
            ty = ty->type;
        }
        ty = type(op, ty, ty->size, ty->align, NULL);
    }
    return ty;
}
```

如果 ty 是类型（ARRAY ety），而限定是作用于数组元素类型的，那么 qual(op，ty) 就会创建（ARRAY(op，ety)）。如果 ty 已经是限定的，即是（CONST ty->type）或（VOLATILE ty->type），并且 op 是另一个限定符，那么 qual 创建（CONST+VOLATILE ty->type）。这一约定使 qual 的代码变得复杂，但是我们只使用一个类型节点而不是一或两个类型节点来描述限定类型，简化了 isqual 函数。

4.5 函数类型

函数类型的 type 域给出了函数返回值的类型，联合 u 保存了描述各参数类型的结构：

```
⟨function types 48⟩=                                    43
    struct {
        unsigned oldstyle:1;
        Type *proto;
    } f;
```

f.oldstyle 标记区分两种函数类型：1 表示旧风格（old-style）类型，即省略参数的类型；0 表示新风格（new-style）类型，即总是包含参数的类型。f.proto 指向以空指针（null）结尾的 Type 数组，f.proto[i] 是第 i+1 个参数的类型。因为旧风格的函数类型可能带有原型，需要 f.oldstyle 标记。但是，正如 ANSI 标准指出的一样，这些原型不用于检查函数调用时的实参类型。旧风格的函数定义之后，再使用新风格声明函数是很少见的。例如：

```
int f(x,y) int x; double y; { ... }
extern int f(int, double);
```

第一条语句定义了为旧风格函数，随后的语句声明 f 的原型。原型必须与定义保持一致，但不用于调用 f 时的检查实参类型。

func 函数创建类型（FUNCTION ty {proto}），ty 是返回值类型，花括号括住的是原型。func 初始化原型和 old-style 标记：

```
⟨types.c functions⟩+=                                   47 49
    Type func(ty, proto, style) Type ty, *proto; int style; {
        if (ty && (isarray(ty) || isfunc(ty)))
            error("illegal return type '%t'\n", ty);
        ty = type(FUNCTION, ty, 0, 0, NULL);
        ty->u.f.proto = proto;
        ty->u.f.oldstyle = style;
        return ty;
    }
```

freturn 对于函数类型的作用与 deref 对指针类型的作用一样。它以类型（FUNCTION ty）作为输入，间接访问（FUNCTION ty），生成函数返回值类型 ty。

```
{types.c functions}+=                                    48 49
  Type freturn(ty) Type ty; {
     if (isfunc(ty))
        return ty->type;
     error("type error: %s\n", "function expected");
     return inttype;
  }
```

ANSI C 支持不带参数的函数，这种函数在声明时用 void 作为参数列表。例如：

```
void f(void);
```

f 声明为不带参数的函数，并且没有返回值。从内部表示上看，不带参数的函数的原型是空的，仅由表示结束的空指针组成。f 的类型可以描述为：

```
(FUNCTION (VOID) {(VOID)})
```

ANSI C 支持参数数目可变的函数，例如 sprintf，它的声明为：

```
int sprintf(char *, char *, ...);
```

省略号表示参数列表的可变部分。可变参数的原型包括已声明的参数的类型，void 类型和表示结束的空指针。sprintf 的类型是：

```
(FUNCTION (INT)
   {(POINTER (CHAR))
    (POINTER (CHAR))
    (VOID)})
```

variadic 断言通过查找函数原型的末尾是否有 void 类型来测试函数类型是否带有长度可变的参数列表：

```
{types.c functions}+=                                    49 50
  int variadic(ty) Type ty; {
     if (isfunc(ty) && ty->u.f.proto) {
        int i;
        for (i = 0; ty->u.f.proto[i]; i++)
           ;
        return i > 1 && ty->u.f.proto[i-1] == voidtype;
     }
     return 0;
  }
```

参数数目可变的函数要求至少声明了一个参数，然后才是零个或多个可选参数。因此，判断时不会将在原型结尾处的 void 与不带参数的函数混淆，后者的原型只有一个元素，即 {(VOID)}。

4.6 结构和枚举类型

结构和联合类型是通过标记识别的，这些类型的 u.sym 域指向这些标记的符号表入口。这些域存放在符号表入口中，而不是在类型本身中。symbol 结构的相关域是 u.s：

```
{struct types 50}=                                                  28
  struct {
     unsigned cfields:1;
     unsigned vfields:1;
     Field flist;
  } s;
```

如果结构或联合类型的任意域带有常量（const）限定符或可变（volatile）限定符，那么 cfields 和 vfields 都为 1。flist 域指向用 link 域连接起来的 field 结构：

```
{types.c typedefs}+=                                                40
  typedef struct field *Field;

{types.c exported types}+=                                          40
  struct field {
     char *name;
     Type type;
     int offset;
     short bitsize;
     short lsb;
     Field link;
  };
```

name 保存域的名字，type 是域的类型，offset 是该域在结构的实例中以字节为单位的偏移量。

当 field 描述一个位域时，type 等于 inttype 或 unsignedtype，只有这两种类型才允许位域。lsb 域非零，用于下面的宏定义中。lsb 等于位域中最低有效位的位号加 1，最低有效位的位号从零开始。

```
{types.c exported macros}+=                                         45 56
  #define fieldsize(p)  (p)->bitsize
  #define fieldright(p) ((p)->lsb - 1)
  #define fieldleft(p)  (8*(p)->type->size - \
                          fieldsize(p) - fieldright(p))
  #define fieldmask(p)  (~(~(unsigned)0<<fieldsize(p)))
```

宏 fieldsize 返回 bitsize 域，bitsize 存放位域的以位为单位的长度。fieldright 返回位域最右边位的位号，可用于对位域进行移位，使位域从符号或无符号整数的最低有效位开始。类似地，fieldleft 返回域最左边位的位号，用于符号位域的符号扩展操作。fieldmask 是一个掩码，由 bitsize 个 1 组成，用于抽取位域时清除无用的位。值得注意的是，位域的表示不依赖于目标机器的字节顺序（endianness），不管是高位优先（big endian），还是低位优先（little endian），位域的表示都相同。

newstruct 创建新的类型（STRUCT["tag"]）或（UNION["tag"]），tag 是标记。声明或定义一个新的结构或联合时，无论有没有域列表，structdcl 函数都调用 newstruct。创建新的结构或联合类型时，类型的标记装入 types 表。对匿名的结构和联合，也就是没有标记的结构和联合，newstruct 为它们生成标记：

```
{types.c functions}+=                                               49 51
  Type newstruct(op, tag) int op; char *tag; {
     Symbol p;

     if (*tag == 0)
        tag = stringd(genlabel(1));
     else
```

```
        (check for redefinition of tag 51)
    p = install(tag, &types, level, PERM);
    p->type = type(op, NULL, 0, 0, p);
    if (p->scope > maxlevel)
        maxlevel = p->scope;
    p->src = src;
    return p->type;
}
```

将新的标记加入 types 表时，可能会创建作用域层数大于 maxlevel 的入口，因此，如果需要，应调整 maxlevel。结构类型指向它们的符号表入口，反过来符号表入口也指向结构类型，这样，标记可以映射到类型，类型也可以映射到标记。例如，当标记用在声明符中时，标记映射成类型，参见 structdcl。当 rmtypes 将类型从 typetable 中删除时，需要将类型映射成标记。

在同一作用域中多次定义相同的标记是非法的，但多次声明同一标记是合法的。如果给出了包含域的结构声明，那么就声明并定义了结构标记；如果使用不包含域的结构标记，那么只是声明标记而已。例如：

```
struct employee {
    char *name;
    struct date *hired;
    char ssn[9];
}
```

声明并定义了 employee，但只是声明了 date。定义一个标记时，其 defined 标记被设置，通过检查 defined 标记可以判断标记是否重复定义：

```
⟨check for redefinition of tag 51⟩=                                    51
    if ((p = lookup(tag, types)) != NULL && (p->scope == level
    || p->scope == PARAM && level == PARAM+1)) {
        if (p->type->op == op && !p->defined)
            return p->type;
        error("redefinition of '%s' previously defined at %w\n",
            p->name, &p->src);
    }
```

参数和参数类型的作用域为 PARAM，局部变量的作用域从 PARAM+1 开始。ANSI C 规定了参数与顶层局部变量的作用域相同，因此必须测试该作用域下局部变量是否重定义了参数中定义的标记。这种作用域的划分并不是 ANSI C 标准的要求，lcc 内部使用它来区分参数和局部变量，foreach 函数能够分别访问它们。

newfield 函数分配一个 field 结构，将该结构附加到结构类型 ty 的符号表入口的域列表中，从而在 ty 中加入一个类型为 fty 的域：

```
⟨types.c functions⟩+≡                                          50 52
    Field newfield(name, ty, fty) char *name; Type ty, fty; {
        Field p, *q = &ty->u.sym->u.s.flist;

        if (name == NULL)
            name = stringd(genlabel(1));
        for (p = *q; p; q = &p->link, p = *q)
            if (p->name == name)
                error("duplicate field name '%s' in '%t'\n",
```

```
            name, ty);
    NEW0(p, PERM);
    *q = p;
    p->name = name;
    p->type = fty;
    return p;
}
```

如果 name 为空，newfield 生成一个名字。field 函数使用这个功能处理无名位域。fieldref 函数搜索域列表，参见练习 4.6。

枚举类型和结构、联合类型相似，只是它们没有域，并且它们的 type 域给出了相关联的整数类型。lcc 总是使用 inttype。ANSI C 标准允许编译器使用任何能存放枚举值的整数类型，但许多编译器都使用 int。lcc 也这样处理，以保持兼容性。枚举类型具有 type 域，使 lcc 能够为不同的枚举使用不同的整数类型。以操作符 ENUM 为参数调用 newstruct 可以创建枚举类型，newstruct 返回类型 (ENUM["tag"])。

与结构或联合类型一样，枚举类型的 u.sym 域指向其标记的符号表入口，但是它使用 symbol 结构中的不同部分：

```
(enum types 52)=                                                    28
  Symbol *idlist;
```

对于枚举类型包含的枚举常量，idlist 指向以空结尾的 Symbol 数组。这些符号加载到 identifiers 表，每一个入口包含该枚举常量的值：

```
(enum constants 52)=
  int value;
```

枚举常量不是枚举类型的一部分。分析枚举常量时，将创建、初始化枚举常量，并将枚举常量打包在一个数组中。参见练习 11.9。

4.7 类型检查函数

判断两个类型是否兼容是类型检查的关键。本节描述的函数有助于实现 ANSI C 类型检查规则。

如果两个类型兼容，eqtype 函数返回 1，否则返回 0。

```
(types.c functions)+=                                               51 54
  int eqtype(ty1, ty2, ret) Type ty1, ty2; int ret; {
     if (ty1 == ty2)
        return 1;
     if (ty1->op != ty2->op)
        return 0;
     switch (ty1->op) {
     case CHAR: case SHORT: case UNSIGNED: case INT:
     case ENUM: case UNION: case STRUCT:   case DOUBLE:
        return 0;
     case POINTER:  (check for compatible pointer types 53)
     case VOLATILE: case CONST+VOLATILE:
     case CONST:    (check for compatible qualified types 53)
     case ARRAY:    (check for compatible array types 53)
     case FUNCTION: (check for compatible function types 53)
     }
  }
```

如果 ty1 或 ty2 是不完全类型，则 eqtype 返回第三个参数 ret 的值。

每个类型总是与自身兼容。type 函数保证大多数类型只有一个实例，这样许多兼容类型都可以通过 eqtype 的第一步测试。同样，许多类型不兼容都是因为测试类型具有不同的操作符，绝对是不兼容的，导致 eqtype 返回 0。

如果两个不同的类型具有相同的操作符 CHAR、SHORT、UNSIGNED 或 INT，但表示不同的类型，如 unsigned short 和 signed short，这两个类型也不兼容。同样，两个枚举类型、结构或联合类型，只有当它们是相同的类型时才是兼容的。

剩下的 case 语句遍历类型结构，判断兼容性。例如，如果两个指针类型引用的类型兼容，则指针类型兼容：

```
⟨check for compatible pointer types 53⟩=                    52
  return eqtype(ty1->type, ty2->type, 1);
```

如果两个相似的限定类型的非限定类型兼容，则它们也兼容：

```
⟨check for compatible qualified types 53⟩=                  52
  return eqtype(ty1->type, ty2->type, 1);
```

不完全类型不包括它所描述的对象的大小。例如：

```
int a[];
```

声明了一个大小未知的数组。类型由（ARRAY 0(INT)）给出，大小为 0 表示一个不完全类型。如果两个数组的元素类型兼容，并且大小也相等（如果给定了大小），则这两个数组类型是兼容的：

```
⟨check for compatible array types 53⟩=                      52
  if (eqtype(ty1->type, ty2->type, 1)) {
     if (ty1->size == ty2->size)
        return 1;
     if (ty1->size == 0 || ty2->size == 0)
        return ret;
  }
  return 0;
```

如果其中一个数组为不完全类型，eqtype 返回 ret，否则它们的类型仍兼容。当 eqtype 递归调用本身时，ret 总为 1，在别处调用时 ret 通常也为 1。一些操作符，如指针比较，要求操作数要么都是不完全类型，要么都是完全类型，这时 eqtype 的调用参数 ret 等于 0。上面代码中的第一个测试是处理两个数组的大小都未知的情况。

如果两个函数类型的返回类型和原型都兼容，那么这两个函数类型兼容：

```
⟨check for compatible function types 53⟩=                   52
  if (eqtype(ty1->type, ty2->type, 1)) {
     Type *p1 = ty1->u.f.proto, *p2 = ty2->u.f.proto;
     if (p1 == p2)
        return 1;
     if (p1 && p2) {
        ⟨check for compatible prototypes 54⟩
     } else {
        ⟨check if prototype is upward compatible 54⟩
     }
  }
  return 0;
```

当两个函数都有原型时，情况相对简单一些。原型必须具有相同数目的参数类型，每个原型中的类型的非限定形式必须兼容。

```
⟨check for compatible prototypes 54⟩=                                53
  for ( ; *p1 && *p2; p1++, p2++)
     if (eqtype(unqual(*p1), unqual(*p2), 1) == 0)
        return 0;
  if (*p1 == NULL && *p2 == NULL)
     return 1;
```

其他情况更复杂一些。如果其中一个函数类型有原型，则该函数类型的每个参数的类型必须与应用了默认参数提升（default argument promotion）得到的类型本身的非限定形式兼容。同时，如果具有原型的函数类型的参数个数可变，则两个函数类型不兼容。

```
⟨check if prototype is upward compatible 54⟩=                        53
  if (variadic(p1 ? ty1 : ty2))
     return 0;
  if (p1 == NULL)
     p1 = p2;
  for ( ; *p1; p1++) {
     Type ty = unqual(*p1);
     if (promote(ty) != ty || ty == floattype)
        return 0;
  }
  return 1;
```

默认参数提升规定浮点提升为双精度，小整数和枚举提升为整数或无符号数。上面的代码对浮点类型显式提升，为其他的调用 promote 函数。promote 实现整型提升：

```
⟨types.c functions⟩+=                                          52 54
  Type promote(ty) Type ty; {
     ty = unqual(ty);
     if (isunsigned(ty) || ty == longtype)
        return ty;
     else if (isint(ty) || isenum(ty))
        return inttype;
     return ty;
  }
```

两个兼容类型可以结合（combine）形成新的复合类型。例如，C 程序：

```
int x[];
int x[10];
```

第一条声明语句声明 x 具有类型（ARRAY 0(INT)），第二条形成新类型（ARRAY 10(INT)）。这两种类型结合形成的类型（ARRAY 10(INT)）变为 x 的类型。在类型结合时，复合类型的大小等于第二种类型的大小。另一个例子是结合具有原型的函数类型与不带原型的函数类型。

compose 以两个兼容类型为参数，返回复合（composite）类型。在结构上 compose 与 eqtype 相似，且在处理简单情况时也相似。

```
⟨types.c functions⟩+=                                          54 56
  Type compose(ty1, ty2) Type ty1, ty2; {
```

```
    if (ty1 == ty2)
        return ty1;
    switch (ty1->op) {
    case POINTER:
        return ptr(compose(ty1->type, ty2->type));
    case CONST+VOLATILE:
        return qual(CONST, qual(VOLATILE,
            compose(ty1->type, ty2->type)));
    case CONST: case VOLATILE:
        return qual(ty1->op, compose(ty1->type, ty2->type));
    case ARRAY:    { (compose two array types 55)     }
    case FUNCTION: { (compose two function types 55) }
    }
}
```

如果两个兼容的数组类型中有一个完全类型，则形成的新数组类型的大小等于完全类型的大小。

```
(compose two array types 55)=                                    55
  Type ty = compose(ty1->type, ty2->type);
  if (ty1->size && ty1->type->size && ty2->size == 0)
      return array(ty, ty1->size/ty1->type->size, ty1->align);
  if (ty2->size && ty2->type->size && ty1->size == 0)
      return array(ty, ty2->size/ty2->type->size, ty2->align);
  return array(ty, 0, 0);
```

两个兼容的函数类型所形成的复合类型的返回类型是这两个函数类型的返回类型的复合，参数类型是相应的参数类型的复合。如果其中一个函数类型没有原型，则复合类型的原型来自于另一个函数类型。

```
(compose two function types 55)=                                 55
  Type *p1  = ty1->u.f.proto, *p2 = ty2->u.f.proto;
  Type ty   = compose(ty1->type, ty2->type);
  List tlist = NULL;
  if (p1 == NULL && p2 == NULL)
      return func(ty, NULL, 1);
  if (p1 && p2 == NULL)
      return func(ty, p1, ty1->u.f.oldstyle);
  if (p2 && p1 == NULL)
      return func(ty, p2, ty2->u.f.oldstyle);
  for ( ; *p1 && *p2; p1++, p2++) {
      Type ty = compose(unqual(*p1), unqual(*p2));
      if (isconst(*p1)    || isconst(*p2))
          ty = qual(CONST, ty);
      if (isvolatile(*p1) || isvolatile(*p2))
          ty = qual(VOLATILE, ty);
      tlist = append(ty, tlist);
  }
  return func(ty, ltov(&tlist, PERM), 0);
```

上述代码中，数据结构 List 是指针列表，通过列表函数 append 和 ltov 对 List 进行操作。

4.8 类型映射

本章描述的类型表示和类型函数主要用于前端，后端可能检查类型的 size 和 align 域，但不能依赖于其他域。

然而，后端必须能够将 Type 映射为类型后缀，用来形成第 5 章中描述的类型相关的操作符。类型后缀是类型操作符的子集。ttob 函数将类型映射为相应的类型后缀：

```
⟨types.c functions⟩+=                                                  54 56
  int ttob(ty) Type ty; {
     switch (ty->op) {
     case CONST: case VOLATILE: case CONST+VOLATILE:
        return ttob(ty->type);
     case CHAR: case INT:   case SHORT: case UNSIGNED:
     case VOID: case FLOAT: case DOUBLE:  return ty->op;
     case POINTER: case FUNCTION:         return POINTER;
     case ARRAY: case STRUCT: case UNION: return STRUCT;
     case ENUM:                           return INT;
     }
  }
```

widen 的作用与 ttob 相似，只是将所有整数类型扩展为 int：

```
⟨types.c exported macros⟩+=                                            50
  #define widen(t) (isint(t) || isenum(t) ? INT : ttob(t))
```

btot 的功能和 ttob 相反，将类型操作符或类型后缀 op 转换为满足 optype(op)—ttob(btot(op)) 的类型：

```
⟨types.c functions⟩+=                                                  56
  Type btot(op) int op; {
     switch (optype(op)) {
     case F: return floattype;
     case D: return doubletype;
     case C: return chartype;
     case S: return shorttype;
     case I: return inttype;
     case U: return unsignedtype;
     case P: return voidptype;
     }
  }
```

5.5 节定义的枚举标识符 F、D、C…是相应的类型操作符的缩写。

深入阅读

lcc 的类型表示主要针对类型可由文法规定的语言，此类语言的类型可以通过链接结构表示，链接结构实际上是类型文法导出的表达式的抽象语法树。Aho，Sethi and Ullman（1986）详细描述了这种类型表示方法并且阐明如何进行类型检查。这种类型表示和检查不仅适用于 C，也适用于其他函数语言，如 ML（Ullman，1994）。Aho，Sethi and Ullman（1986）中的 6.3 节，特别是练习 6.13，介绍了可移植的 C 编译器 PCC（Johnson，1978）是如何使用位串表示类型的。

练习

4.1 给出下列声明中类型的带括号的前缀形式：

```
long double d;
char ***p;
const int *const volatile *q;
int (*r)[10][4];
struct tree *(*s[])(int, struct tree *, struct tree *);
```

4.2 给出一个C的结构定义的例子，使其引发4.6节所描述的标记重定义诊断错误。

4.3 实现断言：

```
(types.c exported functions)=                                    57
  extern int hasproto ARGS((Type));
```

如果ty不包含函数类型或者包含的所有函数类型都有原型，hasproto返回1，否则返回0。hasproto用于静当遗留的原型说明。由于分析结构时将域的类型作为参数显式调用hasproto，因此如果结构的某个域是函数指针，且函数没有原型，hasproto就不能发出警告信息。

4.4 在调试诊断中，lcc显示类型的英文解释。例如，4.5节所示的sprintf的类型（int sprintf(char*, char*, …)）和类型char *(*strings)[10]分别显示为：

```
int function(char *, char *, ...)
pointer to array 10 of pointer to char
```

printf风格的格式代码%t表示打印下一个Type参数，输出函数调用outtype函数完成对%t的处理：

```
(types.c exported functions)+=                                   57 57
  extern void outtype ARGS((Type));
```

实现outtype函数。

4.5 types.c 生输出3个函数格式化和打印类型。

```
(types.c exported functions)+=                                   57 57
  extern void printdecl  ARGS((Symbol p, Type ty));
  extern void printproto ARGS((Symbol p, Symbol args[]));
  extern char *typestring ARGS((Type ty, char *id));
```

typestring返回C的声明，该声明指定ty是标识符id的类型。例如，如果ty是

```
(POINTER (ARRAY 10 (POINTER (CHAR))))
```

且id是"strings"，typestring返回"char *(*strings)[10]"。lcc的-P选项通过在标准错误输出上打印函数的新风格原型和全局变量，帮助将ANSI之前的代码转换为ANSI C代码。printdecl假设函数p具有类型ty（通常ty等于p->type），打印p的声明；printproto打印函数p的说明，p的参数由args给出。printproto使用args创建参数类型，调用printdecl，printdecl再调用typestring。实现这些函数。

4.6 函数：

```
(types.c exported functions)+=                                   57
  extern Field fieldref ARGS((char *name, Type ty));
```

搜索 ty 的域列表，查找 name 指定的域，返回指向 field 结构的指针。如果没有名为 name 的域，返回 NULL。实现 fieldref。

4.7 解释为什么 lcc 诊断出下列 C 程序的赋值操作数含有非法类型。

```
struct { int x, y; } *p;
struct { int x, y; } *q;
main() { p = q; }
```

4.8 解释为什么 lcc 认为下列 C 程序中 f 的参数类型非法。

```
void f(struct point { int x, y; } *p) {}
struct { int x, y; } *origin;
main() { f(origin); }
```

4.9 解释为什么 lcc 认为下列 C 程序中 isdigit 的定义与 isdigit 的外部声明冲突。

```
extern int isdigit(char c);
int isdigit(c) char c; { return c >= '0' && c <= '9'; }
```

4.10 测试表里 types.c 中 <search for an existing type> 的 if 语句是 lcc 的执行热点之一。改写代码，判断最佳执行时间条件下的测试顺序。如果你发现了最佳顺序，用 lcc 编译自身以测试使用该顺序在时间上的改进。这种改进值得吗？

4.11 结构类型指向存放它们的域列表的符号，这些符号反过来也指向类型。重新设计这个显然比较笨拙的数据结构，使得类型完全与符号表无关。例如，结构类型可以在它的 u 的某个域中携带域列表。带标记的类型可以使用其他域来存储类型定义点的作用域级别。你需要修改诸如 newstruct 之类的函数，初始化这些域；提供函数将标记映射为类型，反之亦然。基本类型需要那些存放在符号表入口的数据，如 addressed 标记。比较修改后的设计，它是否明显比原有的先进？你的实现是否重复了其他地方提供的功能，如符号表模块中的功能？

代码生成接口

本章定义了与目标机器无关的前端和与目标机器相关的后端之间的接口。设计一个好的代码生成接口是很困难的。一个不恰当的接口可能迫使每个后端重复做一些前端能够一次完成的工作。如果接口太小，在开发全新的目标机器后端时可用的信息就会太少；如果接口太庞大，后端会被不必要地复杂化。这些矛盾的需求要求仔细规划代码生成接口，尤其在设计开发新的目标机器时，会暴露新的缺陷，可能需要重新规划接口。

lcc 的接口包括一些共享数据结构、18 个函数（其中大部分比较简单）和包括 36 个操作符的语言，该语言用于将可执行代码从源程序形式编码成 dag（无环有向图）形式。共享数据结构的某些域可以供前端和后端共享，而另一些域则只前端或后端私有。

我们在前面的第 3 章和第 4 章描述了两种共享数据结构：symbol 和 type。尽管编译器后端可以检查这些数据结构中的任何域，但 lcc 约定后端只访问部分域。本章列举了后端可能检查的域，集中描述了完整的接口，回顾了各个域表示的意义，其中省略了逻辑上对前端或后端可私有的域。

5.1 类型度量

类型度量（type metric）指定基本类型的大小和对齐字节数：

```
⟨interface 59⟩=                                          60  11
  typedef struct metrics {
     unsigned char size, align, outofline;
  } Metrics;
```

outofline 标记控制相关类型的常量的放置。如果 outofline 为 1，则该类型的常量不能出现在 dag 中，而是存放在一个匿名的静态变量中。对常量值的访问可通过对静态变量的存取获得。每个基本类型都有类型度量：

```
⟨metrics 59⟩=                                                60
  Metrics charmetric;
  Metrics shortmetric;
  Metrics intmetric;
  Metrics floatmetric;
  Metrics doublemetric;
  Metrics ptrmetric;
  Metrics structmetric;
```

ptrmetric 描述所有类型的指针。structmetric.align 指定结构的最小对齐字节数，结构的最大对齐字节数就是它的各个域的对齐字节数和 structmetric.align 中的最大值。structmetric 的 size 域未使用。只有某种类型的常量的值可以作为立即操作数出现在指令中时，编译器后端才将其类型度量的 outofline 标记置为 0。

字符的大小和对齐字节数都必须为 1。前端可以正确地将符号整数、无符号整数和长整数作为不同类型处理，但是它们都共享 intmetric。同样，double 和 long double 类型共享 doublemetric。每个指针必须能够存放在一个无符号整数中。

5.2 接口记录

交叉编译器是指在某个计算机平台上运行并为另一个平台产生代码的编译器。lcc 可以和多个不同目标机器的代码生成器链接，可以用作本地或交叉编译器。lcc 的接口记录存放了前端需要了解的目标机器信息，包括指向接口例程的指针、类型度量和接口标志。接口记录的定义为：

```
(interface 59)+≡                                          59 72 11
  typedef struct interface {
      (metrics 59)
      (interface flags 65)
      (interface functions 60)
      Xinterface x;
  } Interface;
```

lcc 为每种目标机器都形成一个独有的接口实例。x 域是对 interface 的扩展，后端使用它存放与目标机器相关的接口数据和函数，对后端是私有的。Xinterface 在 config.h 中定义。

接口记录包含指向本节稍后描述的 18 个接口函数的指针。本章中 <interface functions> 定义的函数通常只用函数名标识。例如我们使用 gen 来标识函数，而不用其精确但冗长的说法“接口记录的 gen 域所指的函数”。

接口记录还包含一些指向其他函数的指针，编译器前端使用这些函数为调试器产生符号表：

```
(interface functions 60)≡                                 67 60
  void (*stabblock) ARGS((int, int, Symbol*));
  void (*stabend)   ARGS((Coordinate *, Symbol, Coordinate **,
                      Symbol *, Symbol *));
  void (*stabfend)  ARGS((Symbol, int));
  void (*stabinit)  ARGS((char *, int, char *[]));
  void (*stabline)  ARGS((Coordinate *));
  void (*stabsym)   ARGS((Symbol));
  void (*stabtype)  ARGS((Symbol));
```

为了节省空间，本书将不再详细描述这些 stab 函数。

5.3 符号

符号表示变量、标号或常量，用 scope 域区分。对于变量和常量，后端可能查询 type 域，了解数据项的数据类型后缀。对于变量和标号，ref 域的浮点值用于估计变量和标号被引用的次数，非零的值表示变量或标号至少被引用一次。对于标号、常量和某些变量，联合 u 的域补充说明了一些额外的数据。

变量的 scope 域可以等于 GLOBAL、PARAM 或 LOCAL+k（k 为嵌套层次），sclass 域可以是 STATIC、AUTO、EXTERN 或 REGISTER。大多数变量的 name 域等于变量在源代码中使用的名字。对于临时变量和其他生成的变量，变量名是数字序列。对于全局和静态变量，u.seg 给出了变量定义点的逻辑段。如果接口标志 wants_dag 为 0，则前端生成显式的临时变量以保存那些使用多次的公共子表达式。前端设置这些临时变量符号的 u.t.cse 域，u.t.cse 可作为 dag 节点，计算所保存的值。

临时变量的 temporary 域和 generated 域都置为 1，而标号和其他生成的变量，如存放字符串文字的变量，只有 generated 置为 1。当接口标志 wants_argb 置为 1 时，structarg 标识结构参数。wants_argb 将在下面详细介绍。

标号的 scope 域等于 LABELS，u.l.label 是一个唯一标识该标号的数值，name 是值的字符串表示。标号没有 type 或 sclass。

常量的 scope 域等于 CONSTANTS，sclass 等于 STATIC。对于整型或指针常量，name 是 C 常量的字符串表示。对其他类型，name 没有定义。常量的实际值存放在 u.c.v 域（在 3.6 节中定义）中。如果需要生成一个变量来存放常量，那么 u.c.loc 指向该变量的符号表入口。

符号包含类型为 Xsymbol 的 x 域（Xsymbol 类型在 config.h 中定义），用于后端存放符号中与目标机器相关的数据，如局部变量的栈帧移量。x 域为后端私有，因此它的内容不是接口的一部分。第 13 章将进行详细介绍。

5.4 类型

符号中有一个 type 域。如果一个符号表示一个常量或变量，type 域就指向描述该数据项类型的结构。后端可以访问类型结构的 size 和 align 域，以了解类型的以字节为单位的大小和对齐限制。后端也可以传递 type 指针本身给类型断言函数，如 isarray 和 itob，在不检查其他域的情况下了解类型信息。

5.5 dag 操作

可执行代码用 dag 来描述。函数体是 dag 组成的序列或森林，每个 dag 都通过 gen 函数传递给编译器后端。dag 节点（有时也叫节点）定义如下：

```
⟨c.h typedefs⟩=
  typedef struct node *Node;

⟨c.h exported types⟩=                                    62
  struct node {
     short op;
     short count;
     Symbol syms[3];
     Node kids[2];
     Node link;
     Xnode x;
  };
```

kids 的元素指向操作数节点。一些 dag 操作也使用一个或两个符号表指针作为操作数，这些操作数存放在 syms 中。后端出于自身需要可能使用 syms 的第三个元素，前端在 dag 传递给后端之前也会临时使用它（参见 12.8 节）。link 指向森林中下一个 dag 的根。

count 记录了节点的值被使用或被其他节点引用的次数。只有来自 kids 的引用才被计数，来自 link 的引用不表示对节点值的使用，不被计数。事实上，link 只对根节点有意义，表示副作用而不是值的引用。如果接口标志 wants_dag 为 0，根的引用次数 count 总是等于 0。共享节点是指 count 大于 1 的节点，它生成的代码只对节点计算一次，但是节点的值被使用 count 次。

x 域是后端对节点的扩展。后端使用 config.h 中定义的类型 Xnode 来存放生成代码时需要的各节点的数据。第 13 章中将描述该域。

op 域存放 dag 操作符。每个操作符的最后一个字符是来自类型定义列表的类型后缀：

```
⟨c.h exported types⟩+≡                                    61 62
  enum {
      F=FLOAT,
      D=DOUBLE,
      C=CHAR,
      S=SHORT,
      I=INT,
      U=UNSIGNED,
      P=POINTER,
      V=VOID,
      B=STRUCT
  };
```

例如，通用操作符 ADD 有变体 ADDI、ADDU、ADDP、ADDF 和 ADDD。通过这种方式定义的后缀的取值为从 1 到 9。

操作符的定义：

```
⟨c.h exported types⟩+≡                                    62 68
  enum { ⟨operators 62⟩ };

⟨operators 62⟩≡                                          62
  CNST=1<<4,
      CNSTC=CNST+C,
      CNSTD=CNST+D,
      CNSTF=CNST+F,
      CNSTI=CNST+I,
      CNSTP=CNST+P,
      CNSTS=CNST+S,
      CNSTU=CNST+U,
  ARG=2<<4,
      ARGB=ARG+B,
      ARGD=ARG+D,
      ARGF=ARG+F,
      ARGI=ARG+I,
      ARGP=ARG+P,
```

<operators> 的其余部分定义了其他操作符。表 5-1 列举了每个通用操作符及其合法类型后缀。使用的 kids 和 syms 的数目；kid 的多个值指出了与类型相关的操作符变体，这些将在下文讲述。syms 栏的符号给出了 syms 值的数目和一个字母代码，说明对 syms 的使用情况：1V 表示 syms[0] 是指向变量的符号；1C 表示 syms[0] 是一个常量；1L 表示 syms[0] 是标号；1S 表示 syms[0] 是常量且常量值是字节大小；2S 增加 syms[1]，且 syms[1] 的值也是常量，表示对齐字节数。对大多数操作符，类型后缀表示要执行的操作类型和结果类型。ADDP 和 SUBP 是例外，其中 ADDP 是整型操作数 kids[1] 加到指针操作数 kids[0] 中，SUBP 是从指针操作数 kids[0] 中减去整型操作数 kids[1]。赋值、比较、参数和一些调用操作符没有返回结果，它们的类型后缀表明了要执行的操作类型。

表 5-1 节点操作符

syms	kids	操作符	类型后缀	操作
1V	0	ADDRF	P	参数地址
1V	0	ADDRG	P	全局地址
1V	0	ADDRL	P	局部地址
1C	0	CNST	CSILPFD	常量
	1	BCOM	U	位补码
	1	CVC	IU	从 char 转换
	1	CVD	I F	从 double 转换
	1	CVF	D	从 float 转换
	1	CVI	C SU D	从 int 转换
	1	CVP	U	从指针转换
	1	CVS	IU	从 short 转换
	1	CVU	CSI P	从 unsigned 转换
	1	INDIR	CSI PFDB	取
	1	NEG	I FD	非
	2	ADD	IUPFD	加
	2	BAND	U	位与
	2	BOR	U	位或
	2	BXOR	U	位异或
	2	DIV	IU FD	除
	2	LSH	IU	左移
	2	MOD	IU	模
	2	MUL	IU FD	乘
	2	RSH	IU	右移
	2	SUB	IUPFD	减
2S	2	ASGN	CSI PFDB	赋值
1L	2	EQ	I FD	等于转移
1L	2	GE	IU FD	大于或等于转移
1L	2	GT	IU FD	大于转移
1L	2	LE	IU FD	小于或等于
1L	2	LT	IU FD	小于转移
1L	2	NE	I FD	不等于转移
2S	1	ARG	I PFDB	参数
1	1 或 2	CALL	I FDBV	函数调用
	1	RET	I FD	从函数返回
	1	JUMP	V	无条件转移
1L	0	LABEL	V	标号定义

叶操作产生一个变量的地址或常量值。syms[0] 表示该变量或常量。单目操作除 INDIR 外，都是接受并产生一个数值，INDIR 接受一个地址得到该地址存放的值。没有 BCOMI 操作符，符号整型的补码利用 BCOMU 获得。二元操作接受两个数，生成一个数。

转换操作的类型后缀表示结果的类型。例如，CVUI 将一个无符号数（U）转换为符号整数（I）。无符号数和短整数（short）之间、无符号数和字符之间的转换属于无符号转换，而整数和短整数、整数和字符之间的转换属于符号转换。例如，CVSU 将无符号短整数转换为无符号整数，只要高位清零。CVSI 将符号短整数转换为符号整数，需要传播短整数的符号位以填充高位。

对于不在表 5-1 中的转换操作，前端会创建 dag 或组合转换以实现相应的功能。例如，将短整数转换为浮点，需要首先将短整数转换为整数，再将整数转换为双精度数。图 5-1 中用箭头表示了 16 种转换操作。沿着从源类型到目的类型的路径可以形成组合的转换操作。

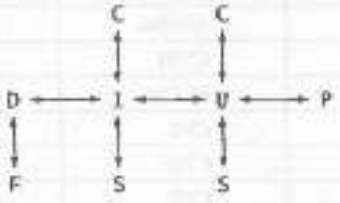


图 5-1 转换

ASGN 将 kids[1] 的值存入 kids[0] 指定的地址单元。syms[0] 和 syms[1] 分别指向表示 kids[1] 值的大小和对齐字节数的整型常量的符号表入口。这样的安排对于结构赋值和初始化自动数组的操作 ASGNB 尤其有用。

无条件跳转 JUMPV 的转移地址由 kids[0] 计算得来。对大多数转移操作，kids[0] 是常量 ADDRGP 节点，但是 switch 语句计算的是可变的目标地址，因此 kids[0] 可以是任意的计算。LABEL 定义的标号由 syms[0] 给出，否则是个空操作。对于比较操作，如果比较结果为真，syms[0] 指向将要转移的标号的符号表入口。因为等于测试不需要对符号位进行特殊考虑，无符号的等于和不等于操作使用符号比较来完成。

零个或多个 ARG 节点加上 CALL 节点表示函数调用。前端解决函数的嵌套调用：首先执行内层的函数调用，将它的值赋给一个临时变量，随后使用该临时变量。因此，ARG 节点总是与森林中的下一个 CALL 节点关联。如果 wants_dag 等于 1，CALL 节点总是作为森林的根节点出现。如果 wants_dag 等于 0，只有 CALLV 节点才能作为根出现，其他的 CALL 节点只能作为根节点 ASGN 的右操作数出现。

CALL 节点的 syms[0] 所指向的符号的唯一非空域是 type，表示被调用者的函数类型。

ARG 节点确立了 kids[0] 计算得出的值作为下一个参数。syms[0] 和 syms[1] 指向的符号表入口分别是参数的大小和对齐字节数的常量。

在 CALL 节点中，kids[0] 计算被调用者的地址。CALLB 节点用于调用返回值为结构的函数，kids[1] 计算存放返回值的临时变量的地址。CALLB 节点和函数头代码共同将 CALLB 的 kids[1] 存放到被调用者的第一个局部变量中。SPARC 接口程序 function 和 local 以及 CALLB 的产生过程展示了这种合作。由于在引用返回值时，前端实际引用的是临时变量，因此 CALLB 节点中 count 等于 0。操作符中没有 RETB，前端使用 ASGNB 给第一个局部变量指定的地址赋值。只有当接口标志 wants_callb 等于 1 时，CALLB 节点才能出现，参见 5.6 节。RET 节点中，kids[0] 计算返回值。

由于大部分机器要求传递的参数必须至少是整数，因此，字符和短整数实参总是被提升为相应的整数类型，即使在有原型的时候。在函数的入口，提升的值重新转换为形参声明的类型。例如，函数体

```
f(char c) { f(c); }
```

形成图 5-2 所示的两个森林。图中实线代表 kids 指针，虚线代表 link 指针。左边的森林保存了一个 dag，表示将变宽的实参变窄，转换为形参指定的类型。在左边的 dag 中，左边的 ADDRFP c 表示形参，在 INDIRC 之下的 ADDRFP c 表示实参。右边的森林包含两个 dag，第一个提升实参

c，使它作为整数传递，第二个 dag 调用 f。

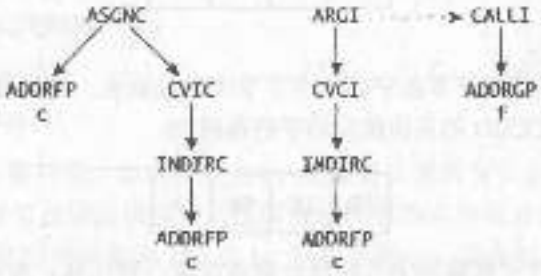


图 5-2 f(char c){f(c);} 的森林

ASGN、INDIR、ARG、CALL 和 RET 的无符号变体可以省略。符号和无符号整数的大小相同，可以使用相应的符号操作符来完成无符号操作。同理，操作符中也没有 CALLP 和 RETP，可以使用 CVPU 和 RETI 返回指针。因此，可以通过使用 CALLI 和 CVUP 来调用返回指针值的函数。

在表 5-1 中，ASGN 以及 ASGN 之后的操作符具有副作用，它们作为根节点出现在森林中，引用次数为 0。CALLD、CALLF 和 CALLI 可能产生一个值，此时它们作为 ASGN 节点的右操作数出现，且引用次数为 1。除了这种例外情况，所有具有副作用的操作符总是出现在 dag 森林的根节点，出现的顺序与它们被执行的顺序一致。前端通过对森林的 dag 的排序来表明对计算顺序的所有限制。如果 ANSI 规定 x 必须在 y 之前计算，则 x 的 dag 必须出现在 y 的 dag 之前的森林，或者 x 和 y 出现在同一 dag，但是 x 出现在以 y 为根的子树中。例如：

```
int i, *p; f() { i = *p++; }
```

f 的函数体代码产生的森林如图 5-3 所示。INDIRP 取 p 的值，ASGNP 将 P 的值修改为 INDIRP 与 4 求和的结果，ASGNI 将原来的 p 值指向的整数赋给 i。由于在森林中 INDIRP 出现在 p 被修改之前，所以保证了 INDIRI 使用的是 p 原来的值。

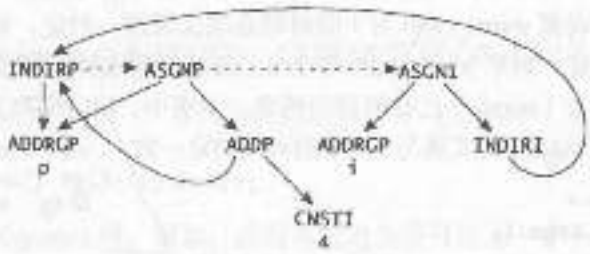


图 5-3 int i, *p; f(){i=*p++;} 的森林

5.6 接口标志

接口标志有助于为目标机器配置前端。

```
⟨interface flags 65⟩=                                    66  60
  unsigned little_endian:1;
```

如果目标机器是低位优先，little_endian 等于 1；如果是高位优先，则等于 0。如果每个字的最低有效位字节在字的所有字节中地址最小，则计算机是低位优先的。例如，32 位无符号值 0×AABBCCDD 的低位优先的字的布局为：



字节的地址从右到左递增。

如果每个字的最低有效位字节在字的所有字节中地址最大，则计算机是高位优先的。例如，32位无符号值0×AABBCCDD的高位优先的字的布局为：



换句话说，lcc前端以无符号整数的字节的地址顺序放置一列位域：在低位优先的机器上从最低有效位到最高有效位，在高位优先的机器上正好相反。ANSI支持两种顺序，但是跟随地址增长是一般惯例。

```
⟨interface flags 66⟩+≡                                    65 66 60
  unsigned mulops_calls:1;
```

如果由硬件实现乘、除和求余，mulops_calls应等于0。如果硬件没有实现，而是交给库例程实现，则mulops_calls等于1。前端将这类调用展开执行，因此，前端需要知道什么操作通过调用库例程模拟（emulate）计算。可能有必要推广这一特性，处理其他模拟指令，但迄今为止，还没有发现有目标机器需要更多这样的标志。

```
⟨interface flags 65⟩+≡                                    66 66 60
  unsigned wants_callb:1;
```

该标志通知前端产生CALLB节点以调用返回结构的函数。如果wants_callb为0，前端不生成CALLB节点，而是使用更简单的操作实现调用：首先传递一个额外的、位于第一位的隐藏参数，该参数指向一个临时变量，并以ASGNB dag作为每个结构函数的结束，复制返回值到临时变量。当调用者需要使用返回值时，就引用该临时变量。如果wants_callb等于1，前端产生CALLB节点，CALLB节点的kids[1]域计算存放返回值的空间的地址，且任意返回结构的函数的第一个局部变量都保存该地址。设置wants_callb为1的后端必须实现这一约定，例如，根据kids[1]初始化第一个局部变量的地址。如果wants_callb等于0，后端不能控制返回结构参数的函数的代码，于是后端通常也不能模仿（mimic）已有的调用约定。本书中，MIPS和X86代码产生器初始化wants_callb为0，前端CALLB的实现与MIPS的调用约定一致。

```
⟨interface flags 65⟩+≡                                    66 66 60
  unsigned wants_argb:1;
```

该标志告诉前端产生ARGB节点以传递结构参数。如果wants_argb等于0，前端不产生ARGB节点，而是使用更简单的操作实现结构参数：创建ASGNB dag，将结构参数复制到临时变量，传递指向该临时变量的指针，增加一个额外的对被调用函数的参数的间接引用，改变被调用函数的形参类型以反映这种转换。前端为这些结构参数设置structarg标志，以便与其他真正的结构指针区分开来。如果wants_argb为0，后端不能控制结构参数的代码，因此，通常后端不能模仿已有的调用约定。在本书中，SPARC代码产生器将wants_argb初始化为0，其他代码产生器将wants_argb初始化为1。前端ARGB的实现与SPARC调用约定一致。

```
⟨interface flags 65⟩+≡                                    66 67 60
  unsigned left_to_right:1;
```

该标志告诉前端按照从左到右的顺序计算和提交参数给后端。即，在 CALL 节点之前的 ARG 节点出现的顺序与源代码的参数的顺序一致。如果 left_to_right 为 0，则参数按照从右到左的顺序进行计算和传递。ANSI 允许两种顺序。

```
⟨interface flags 65⟩+≡                                    66  60
  unsigned wants_dag:1;
```

该标志告诉前端传递 dag 给后端。如果标志为 0，前端要处理所有节点的引用次数大于 1 的 dag 节点；创建临时变量，将节点赋给该变量，在所有使用原节点的地方改成使用该临时变量。当 wants_dag 等于 0 时，所有引用次数都为 0 或 1，只保留树——退化的 dag，而没有通常的 dag。本书的代码产生器使用的代码生成方法需要树，因此将 wants_dag 初始化为 0，但 lcc 的其他代码产生器是从 dag 中生成代码的。

5.7 初始化

在初始化过程中，前端调用

```
⟨interface functions 60⟩+≡                                60  67  60
  void (*progbeg) ARGS((int argc, char *argv[]));
```

argv[0..argc-1] 指向前端不能识别的程序参数。这些参数被认为是与特定目标机器相关的选项。progbeg 处理这些选项并且初始化后端。

在编译的最后，前端调用

```
⟨interface functions 60⟩+≡                                67  68  60
  void (*progend) ARGS((void));
```

使后端有机会完成输出。在一些目标机器上，progend 不含任何操作，为空。

5.8 定义

当前端定义一个 scope 为 CONSTANTS，LABELS 或 GLOBAL 的新符号，或一个静态变量时，要调用

```
⟨interface functions 60⟩+≡                                67  68  60
  void (*defsymbol) ARGS((Symbol));
```

使后端有机会初始化 Xsymbol 域。例如，后端可能想为符号定义一个不同的名字。在本书中一些目标机器上要约定要在全局变量名前加上前缀下划线 "_"。function 函数初始化 scope 域等于 PARAM 的符号的 Xsymbol 域，local 函数初始化 scope 为 LOCAL+k 的符号的 Xsymbol 域，而 address 函数则初始化表示地址计算的符号的 Xsymbol 域。

如果符号在一个模块中定义，在另外的模块中使用，那么该符号是输出符号；反之，若符号在一个模块中使用，在另外的模块中定义，则该符号是输入符号。前端调用下面的函数：

```
⟨interface functions 60⟩+≡                                67  68  60
  void (*export) ARGS((Symbol));
  void (*import) ARGS((Symbol));
```

宣告（announce）符号是输出或输入符号。只有非静态变量或函数才能够输出。前端总是在定义符号之前调用 export，但在符号使用前后的任何时候都可以调用 import。大多数目标机器需要调用 export 产生汇编指示。某些目标机器，如 MIPS 后端，不需要 import 做任何动作，因此，import 为空。

```
⟨Interface functions 60⟩+≡                                    67 68 60
  void (*global) ARGS((Symbol));
```

该函数产生定义全局变量的代码。在调用 global 之前，前端已经调用了 segment 函数（下面将要描述），为定义指定正确的逻辑段。前端将符号的 u.seg 设为该逻辑段，然后调用 global 函数和相应的数据初始化函数。global 必须产生必要的对齐指示并定义标号。

前端调用 local 函数宣告符号是局部变量：

```
⟨Interface functions 60⟩+≡                                    68 68 60
  void (*local) ARGS((Symbol));
```

local 宣告符号是临时变量的方式与此类似，将设置符号的 temporary 标志。local 必须初始化 Xsymbol 域，该域保存局部栈帧移量和寄存器号等数据。

前端调用

```
⟨Interface functions 60⟩+≡                                    68 68 60
  void (*address) ARGS((Symbol p, Symbol q, int n));
```

初始化符号 q，使 q 表示地址 x+n，其中 x 是 p 表示的地址，n 是正的或负的整数。与 defsymbol 一样，address 初始化 q 的 Xsymbol 域，但初始化基于 p 的 Xsymbol 和 n 的值。典型的 address 调用是初始化局部变量或参数符号，q 表示的地址是 p 的栈帧移量加上 n，q 的 x.name 是 p 的 x.name 连接上 +n 或 -n。例如，如果 n 等于 40，p 指向名为 array 的符号，如果后端通过加上前缀 "_" 形成名字，则 address 创建名字 _array+40，使得加操作可以由汇编器完成，而不占用运行时间。address 的参数可以是全局变量、参数和局部变量，但这些符号必须已经由 defsymbol、function 或 local 完成了初始化。

前端调用上述接口过程宣告符号，调用返回后，前端将 defined 标志设置为 1。defined 标志可以防止前端多次宣告同一个符号。

lcc 前端管理 4 个逻辑段，分隔代码、数据和文字：

```
⟨c.h exported types⟩+≡                                        62 73
  enum { CODE=1, BSS, DATA, LIT };
```

前端将可执行代码发送到 CODE 段，将未初始化的变量定义到 BSS 段，将已初始化的变量定义到 DATA 段并初始化，常量则定义并初始化到 LIT 段。前端调用

```
⟨Interface functions 60⟩+≡                                    68 69 60
  void (*segment) ARGS((int));
```

宣告说明段的改变。参数应属于上面定义的 4 种段之一。segment 将逻辑段映射到目标机器提供的段上。

CODE 和 LIT 可以映射到只读的段，BSS 和 DATA 段必须映射到可读写的段上。CODE 和 LIT 段能够映射到同一段，从而可以合并在一起。同样，BSS、DATA 和 LIT 可以任意组合合并。在只有一个段的目标机器上，CODE 段才能够与其他段合并。

5.9 常量

接口函数

```
(interface functions 60)+=                                  68 69 60
  void (*defaddress) ARGS((Symbol));
  void (*defconst)   ARGS((int ty, Value v));
```

对常量进行初始化。defconst 函数产生指示符以定义一个单元，并将该单元初始化为常量值。v 是常量值，ty 是常量的类型的编码，从而可以指出应访问 Value v 的那个元素，如下表所示：

ty	v 域	类型
C	v.uc	字符型
S	v.us	短整型
I	v.i	整型
U	v.u	无符号型
P	v.p	任意指针类型
F	v.f	浮点型
D	v.d	双精度型

编码 C、S、I…标识不同的操作符类型后缀。符号域 v.sc 和 v.ss 可以替换 v.uc 和 v.us，但是 defconst 只能初始化规定数目的位。如果 ty 等于 P，那么 v.p 保存了某些指针类型的数字常量。这来源于一些诸如 char *p=（char*）0×F0 的声明。defaddress 初始化涉及符号的指针常量，而不是涉及数字的指针常量。

第 16 章到第 18 章的 defconst 函数允许交叉编译，因此这些函数要适应不同的表示和字节顺序。例如，如果在低位优先的机器上编译生成高位优先的机器的代码（反之亦然）时，defconst 要交换双精度数据的两部分。

通常，ANSI C 编译器不能将对浮点常量进行编码的工作留给汇编器处理，很少有汇编器能够实现 C 的类型转换。例如：语句

```
double x = (float)0.3;
```

正确地初始化时最低有效位为 0。典型的汇编命令

```
.double 0.3
```

不能实现类型转换，因而错误地初始化 x，使 x 的最低有效位不为 0。因此，大多数 defconst 产生两个无符号数初始化双精度。

```
(interface functions 60)+=                                  69 69 60
  void (*defstring) ARGS((int n, char *s));
```

该函数产生代码将长度为 len 的字符串初始化为 s 中的字符。前端将转义序列（escape sequence）（如 \000）转换为相应的 ASCII 字符。s 中可能嵌有空字节（null），null 不能作为 s 的结束标志，所以 defstring 的参数中不仅包含 s，还包含字符串长度。

```
(interface functions 60)+=                                  69 70 60
  void (*space) ARGS((int));
```

emits code to allocate n zero bytes.

产生代码以分配 n 个值为 0 的字节。

5.10 函数

前端将函数编译为私有数据结构。在将函数的任意部分传递给后端之前，前端必须先对每个函数进行完整的分析。这样的组织允许某些特定的优化。例如，前端只有在处理完整的函数时，才能识别局部变量和没有分配地址的参数，只有这些变量才能分配到寄存器中。

三个接口函数和两个前端函数协同完成函数的编译。

```
⟨interface functions 60⟩+≡                                    69 71 60
  void (*function) ARGS((Symbol, Symbol[], Symbol[], int));
  void (*emit)     ARGS((Node));
  Node (*gen)      ARGS((Node));
⟨dag.c exported functions⟩=                                    243
  extern void emitcode ARGS((void));
  extern void gencode  ARGS((Symbol[], Symbol[]));
```

在每个函数的结尾，前端调用 function 函数生成并发送代码。function 的典型形式是：

```
⟨typical function 70⟩≡
  void function(Symbol f, Symbol caller[], Symbol callee[],
     int ncalls) {
     ⟨initialize⟩
     gencode(caller, callee);
     ⟨emit prologue⟩
     emitcode();
     ⟨emit epilogue⟩
  }
```

前端过程 gencode 遍历前端的私有数据结构，将 dag 的每个森林传递给后端过程 gen。gen 选择代码，在 dag 上添加注释以记录选择的结果，并返回一个 dag 指针。gencode 还可以调用 local 宣告新的局部变量，调用 blockbeg 和 blockend 宣告每个块的开始和结束等。前端过程 emitcode 再次遍历私有数据结构，将 gen 返回的指针传递给 emit 函数发送代码。

这样的组织方式使得后端可以灵活地生成函数头和结束部分。在调用 gencode 之前，function 初始化函数参数的 Xsymbol 域，完成其他必要的针对每个函数的初始化。调用 gencode 之后，过程活动记录（或帧）的大小以及需要保存的寄存器都已确定，这些信息通常在生成函数的开始部分时需要使用。在调用 emitcode 输出函数体代码之后，function 发送函数的结束部分。

function 的参数 f 指向当前函数的符号，ncalls 记录了当前函数调用其他函数的次数。ncalls 使得叶函数（即不调用其他函数的函数）可以得到特殊处理。

caller 和 callee 都是指向符号的指针数组，这两个数组以空指针标志结束。caller 中的符号是调用者传递给函数的参数，callee 中的符号是在函数中可见的参数。对许多函数而言，两个数组中的符号相同，但符号的 sclass 和 type 可能不同。例如：

```
single(x) float x; { ... }
```

调用 single 时可能将一个双精度数据作为实参，但在 single 中，x 是 float。即，caller[0]->type 等于 doubletype（表示双精度的前端全局变量），callee[0]->type 等于 floattype。又如：

```
int strlen(register char *s) { ... }
```

caller[0]->sclass 是 AUTO，而 callee[0]->sclass 为 REGISTER。即使没有寄存器属性说明，前端也将经常引用的参数指定成 REGISTER 方式，从而设置 callee 的 sclass 为 REGISTER。为了避免阻

器程序设计者的意图，只有当没有显式的寄存器局部变量时，才使用这种指定方式。

caller 和 callee 都传递给 gencode。如果 caller[i]->type 和 callee[i]->type 不同，或 caller[i]->sclass 和 callee[i]->sclass 的值不同，gencode 生成一条赋值语句，将 caller[i] 赋给 callee[i]。如果类型不相等，该赋值包括类型转换。例如，single 函数中对 x 的赋值包括从 double 到 float 的截取转换。对包含寄存器声明的参数，function 必须为其分配一个寄存器，并且相应地初始化 x 域，或者将 callee 的 sclass 改为 AUTO，避免不必要的从 caller[i] 到 callee[i] 的赋值。

当 function 函数希望给参数分配一个寄存器时，它可以将 callee[i]->sclass 由 AUTO 改变为 REGISTER。例如，MIPS 的调用约定规定某些参数可以通过寄存器传递，所以在叶函数中，function 将为相应的 callee 分配寄存器。相反，如果设置了 callee[i]->addressed，在函数体中需要使用参数的地址，那么在大多数机器上，该参数必须存储在内存中。

大多数后端为每个函数活动定义了一个参数创建区域（argument-build area），存储调用其他函数所需的参数。前端将嵌套的调用展开（unnest），因此，参数创建区域可用于所有的调用。后端应保证该区域充分大，足以保存最大的参数列表。当调用一个函数时，调用者的参数创建区域就成为被调用者的实参。

由于一些目标机器用寄存器传递某些参数，因此调用不能嵌套。如果我们试图为诸如 f(a,g(b)) 的嵌套调用生成代码，并且如果参数从左到右求值并建立，由于 a 和 b 都使用第一个参数寄存器，而 a 应在 b 之后放入寄存器，但生成的代码却首先将 a 加载到第一个寄存器，然后将 b 加载到同一寄存器从而破坏 a。

某些调用约定是将参数放入栈中，这样就可以处理嵌套调用，也不再一定需要参数创建区域。但将嵌套调用展开的方法还有另一个优点：栈溢出只会发生在函数的入口处，这对于那些需要在函数开始部分有显式代码检测栈溢出的目标机器非常有用。

对每个块，前端首先按声明的顺序宣告那些带显式寄存器声明的局部变量，使得程序员能够控制寄存器分配。然后，前端从它估计最常使用的变量开始宣告其余变量。前端甚至可以将那些地址未被使用且其使用次数可能超过两次的局部变量指定为 REGISTER 类别。声明的顺序以及修改 sclass 域，两种措施合作，使得大部分经常使用的局部变量即使没有用寄存器属性说明，也还能存放在寄存器中。

如果 p 的 sclass 是 REGISTER，local 也可能拒绝为它分配寄存器，并且将它的 sclass 改为 AUTO。如果后端已经将所有可用的寄存器分配给更有用的局部变量，后端没有其他选择，就只能修改 p 的 sclass。与参数处理一样，local 也能为 sclass 等于 AUTO 的局部变量指派寄存器，并将该变量的 sclass 改为 REGISTER，但是只有在变量符号的 addressed 域等于 0 时，才能这样做。

源语言块中的花括号规定了局部变量的生存期（lifetime），gencode 调用下面的函数宣告块的开始和结束：

```
⟨interface functions 60⟩+≡                                    ▲ 70  60
  void (*blockbeg) ARGS((Env *));
  void (*blockend) ARGS((Env *));
```

在 config.h 中定义的 Env 与目标机器相关，一般包括重用局部帧空间中与块关联的部分，以及释放块内指派给局部变量的寄存器所需要的数据。例如，blockbeg 在 Env 中记录帧的大小和块开始时已被占用的寄存器；如果新的块压栈深度超过最大允许值，blockend 恢复寄存器状态，修改栈。第 13 章将详细描述这种情况。

前端调用 gen 选择代码，参数为 dag 组成的森林。例如 5.5 节的图 5-3 给出了下面代码的森林：

```
int i, *p; f() { i = *p++; }
```

下表给出了按后序遍历的森林的线性表示。

节点#	op	count	kids	syms
1	ADDRGP	2		p
2	INDIRP	2	1	
3	CNSTI	1		4
4	ADDP	1	2、3	
5	ASGNP	0	1、4	
6	ADDRGP	1		i
7	INDIRI	1	2	
8	ASGNI	0	6、7	

这个森林包括3个dag，根节点分别为节点2、5、8。由于取p的值的INDIRP节点（节点2）在节点5之前执行，尽管节点5修改了p，随后的节点7仍使用p的原始值，读取该值所指向的整数。

gen遍历森林，选择代码，但并不产生任何代码，这是因为在产生函数的开始部分之前需要确定一些事情，如所需的寄存器等。因此gen只是在节点的x域加上注释，标识它所选择的代码，返回一个指针，该指针最终传递给后端emit，由emit输出代码。一旦前端调用gen，前端将不再使用节点的内容，gen可以自由地修改节点。

emit产生整个森林的代码。典型的处理是，遍历森林，根据操作代码或gen存放在节点中的相关值来输出代码。

5.11 接口绑定

编译选项 -target=name 指定为何种目标机器编译程序。可用目标机器的名字－接口对来存放在全局变量bindlings中。

```
(interface 59)+=                                    60 72  11
  typedef struct binding {
     char *name;
     Interface *ir;
  } Binding;

  extern Binding bindings[];
```

前端识别编译选项 -target 指定的目标机器，将指向该目标机器的接口记录的指针存放在变量IR中。

```
(interface 59)+=                                    72  11
  extern Interface *IR;
```

前端需要调用接口函数，或需要读取类型度量或接口标志时，都使用IR。

后端必须定义和初始化bindings，bindings包括名字和接口记录。例如，本书的后端在bind.c中定义了下列bindlings：

```
(bind.c 72)=
  #include "c.h"
  extern Interface nullIR,   symbolicIR;
  extern Interface mipsebIR, mipselIR;
  extern Interface sparcIR,  solarisIR;
```

```
extern Interface x86IR;
Binding bindings[] = {
   "symbolic",       &symbolicIR,
   "mips-irix",      &mipsebIR,
   "mips-ultrix",    &mipselIR,
   "sparc-sun",      &sparcIR,
   "sparc-solaris", &solarisIR,
   "x86-dos",        &x86IR,
   "null",           &nullIR,
   NULL,             NULL
};
```

MIPS、SPARC 和 X86 接口分别在第 16 章、第 17 章和第 18 章中介绍。null 和 symbolic 接口在练习 5.1 和练习 5.2 中介绍。

5.12 上行调用

前端和后端互为客户端。前端调用后端以执行代码生成和发送。后端调用前端完成输出、分配存储空间、查询类型以及管理节点、符号和串。下面对后端可能调用的前端函数进行总结，尽管其中一些函数在前面的章节中已经解释过，但仍包含在本节中，使得总结比较完整。

void *allocate(int n, int a) 在分配区 a 中分配了 n 个字节，a 可以是下面定义的枚举值之一：

```
⟨c.h exported types⟩+≡                                    68
  enum { PERM=0, FUNC, STMT };
```

allocate 返回一个指向所分配空间的第 1 个字节的指针，保证所分配空间的对齐方式适用于机器的大多数所需类型。在 PERM 分配区中分配的数据在编译结束时释放。在 FUNC 和 STMT 分配区中分配的数据在函数和语句编译结束后立即释放。

```
⟨input.c exported data⟩≡                                  78
  extern char *bp;
```

bp 指向输出缓冲区的下一个字符。语句 *bp++=c 将 c 附加到输出缓冲区（参见 1.5 节的 outs 函数）。每 80 个字符至少调用一次下面的输出函数之一。

```
⟨output.c exported functions⟩+≡                           12 74
  extern void fprint ARGS((int fd, char *fmt, ...));
```

该函数将第 3 个及其后的参数输出到文件描述符 fd 指定的文件中。格式的详细信息参见 print。如果 fd 不等于 1（标准输出），fprint 调用 outflush 将输出缓冲区内容输出到 fd。

如果 ty 的类型是函数，函数 Type freturn(Type ty) 得到函数类型 ty 的返回值类型。

```
⟨c.h exported macros⟩+≡                                   13 74
  #define generic(op) ((op)&~15)
```

宏 generic 是类型相关的 dag 操作符 op 的一般表示，即表达式 generic(op) 返回不含类型后缀的 op。

int genlabel(int n) 将生成的标识符的计数器加 n，并返回计数器原来的值。

int istype(Type ty) 是类型断言，如果类型 ty 属于下表的类型之一，istype 返回非零值。

断言	类型
isarith	算术
isarray	阵列
ischar	字符
isdouble	双精度
isenum	枚举
isfloat	浮点
isfunc	功能
isint	整数
isptr	指针
isscalar	标量
isstruct	结构或联合
isunion	联合
isunsigned	无符号

函数 Node newnode(int op，Node l，Node r，Symbol sym) 分配新的 dag 节点，将节点的 op 域初始化为 op，kids[0] 初始化为 l，kids[1] 初始化为 r，syms[0] 初始化为 sym，并且返回指向新节点的指针。

Symbol newconst(Value v，int t) 在符号表中创建一个常量，常量值为 v，类型后缀为 t，如果需要，返回指向符号表入口的指针。

Symbol newtemp(int sclass，int t) 创建一个临时变量，该临时变量的存储类别为 sclass，类型后缀为 t，返回指向该变量的符号表入口的指针。新的临时变量通过调用 local 宣告。

opindex(op) 得到操作符的编号，对操作符 op：

```
⟨c.h exported macros⟩+=                                    73 74
  #define opindex(op) ((op)>>4)
```

opindex 用于将通用操作符映射到一个连续范围内的整数。

```
⟨c.h exported macros⟩+=                                    74
  #define optype(op) ((op)&15)
```

是 dag 操作符 op 的类型后缀。

```
⟨output.c exported functions⟩+=                            73 75
  extern void outflush ARGS((void));
```

如果当前输出缓冲区不为空，该函数将输出缓冲区的内容写入标准输出。

void outs(char *s) 将字符串 s 附加到标准输出的输出缓冲区中，如果导致缓冲区指针指向了缓冲区末尾的 80 个字符内，则调用 outflush。

void print(char *fmt,…) 将第 2 个及随后的参数打印到标准输出。print 类似于 printf，但只支持格式 %c、%d、%o、%x 和 %s，省略了精度和域宽说明。print 支持 4 种 lcc 特定的格式代码。%s 打印指定长度的串，第 2 个和第 3 个参数分别给出串和串的长度。%k 打印相应参数给出的整数单词编码的英文翻译，%t 打印类型的英文翻译。%w 打印相应的参数给出的源坐标（即在源文件的行列号），此时参数应是指向 Coordinate 的指针。如果 print 打印来自 fmt 的换行符，到达输出缓冲区末尾的 80 个字符以内，那么 print 将调用 outflush。除 %c 外的每种格式都使用 outs 作为实际输出，使缓冲区排空。

int roundup(int n, int m) 增大 n，得到与 n 接近的下一个 m 的整数倍，其中 m 必须是 2 的幂。

char *string(char *s) 在字符串表中创建 s，如果需要，返回指向创建的副本的指针。

char *stringd(int n) 返回 n 的字符串表示，并且在字符串表中创建该串并将其返回。

```
⟨output.c exported functions⟩+=                                    74
  extern char *stringf ARGS((char *, ...));
```

将参数格式化为字符串，并将字符串加入字符串表中，返回指向该字符串的指针。格式信息参见 print。

int ttob(Type ty) 返回类型 ty 的类型后缀。

当类型 ty 表示一个变参函数时，int variadic(Type ty) 返回真。

深入阅读

Fraser and Hanson（1991a、1992）介绍了 lcc 代码生成接口的早期版本。本章基于 lcc 3.1 及以上版本，对 lcc 代码生成接口做了更详细的介绍。

一些编译器的接口产生抽象机代码（abstract machine code），一种类似于假想的机器的汇编代码（Tanenbaum，van Staveren and Stevenson，1982）。前端产生抽象机代码，后端读入抽象机代码并将其转换为目标机器的代码。抽象机使前端和后端分离，使优化遍的插入变得容易，但是额外的 I/O、结构分配和初始化需要消耗时间。lcc 紧密结合的接口使编译器高效并紧凑，但是由于前端的修改可能影响后端，使 lcc 的维护变得复杂。对标准语言来说（如 ANSI C），由于语言本身改变不会太大，因此维护的复杂性并不十分重要。

练习

5.1 如果代码生成器的接口记录所指向的函数实际上不做任何操作，lcc 可以仅作为语法和语义检查器。实现这种空代码生成器。

5.2 实现一个符号后端，生成接口函数的调用轨迹，并以可读的方式表示参数。例如 lcc 的符号后端为：

```
int i, *p; f() { i = *p++; }
```

生成的输出为：

```
export f
segment text
function f type=int function(void) class=auto ...
maxoffset=0
node#2 ADDRGP count=2 p
node'1 INDIRP count=2 #2
node#5 CNSTI count=1 4
node#4 ADDP count=1 #1 #5
node'3 ASGNP count=0 #2 #4 4 4
node#7 ADDRGP count=1 i
node#8 INDIRI count=1 #1
node'6 ASGNI count=0 #7 #8 4 4
1:
end f
segment bss
export p
global p type=pointer to int class=auto ...
```

```
space 4
export i
global i type=int class=auto scope=GLOBAL ref=1000
space 4
```

后续的所有接口例程都显示参数信息，其中一些还可以提供其他信息。例如，function 计算帧的大小，会打印 maxoffset 的值（如上例所示）。同样从上例中可以看到，gen 和 emit 合作打印 dag。gen 为每个森林节点编号，注释节点的 x 域，emit 打印节点操作数的节点号。例如第一个森林中的节点 1、3 和 6。emit 可以在节点号前加入前导符“=”来标识根。对于 LABELV 节点，emit 只打印标号和一个冒号。比较这种输出和 5.10 节的段件表示。

5.3 编写一个代码产生器，它化所有对其他模块可见的标识符的名字的发送过程，并报告未被使用的输入名字。

5.4 在设计 lcc 接口时，32 位整数是标准，因此使整数和长整数共享同一类型度量没有任何损失。实际，许多机器支持 32 位和 64 位整数，若在同一代码产生器中使用这两种类型，lcc 选取的捷径反而使情况复杂了。如何加入两个新的类型后缀 L 和 U，其中长整数使用 L，无符号长整数使用 U。从而改变 lcc 接口呢？考虑对类型度量、通用节点操作符，尤其是转换操作的影响，查阅图 5.1。哪些接口函数需要修改，如何修改？

5.5 设计一个与 lcc 接口一致的抽象机，并用它来分离 lcc 的前端和后端。编写一个代码产生器为该抽象机器输出代码。修改 lcc 后端读取抽象机器代码，重建后端需要使用的数据结构，调用已有后端生成代码。这一过程可能需要花费一个月左右的时间，但在读取抽象机代码、优化抽象机代码并将它输出给后端等方面所具有的灵活性，可以简化后面的优化实验。

词法分析器

词法分析器读入源程序，产生语言的基本词法单元——单词（token）。例如，表达式 *ptr=56; 包含 10 个字符或 5 个单词：*、ptr、=、56、;。词法分析器返回每个单词的单词编码、0 个或多个附加值（associated value）。单字符单词（如操作符和分隔符）的单词编码就是字符本身。包含一个或多个字符的单词，如标识符和常量，使用预定义的常量作为单词编码，预定义的常量值和有效字符的数值（numeric value）不能冲突。

例如，语句 *ptr=56; 产生下面一组单词：左边一列是单词编码，如果单词有附加值，则在右列显示。

单词编码	附加值	
'*'		
ID	"ptr"	"ptr" 对应的符号表入口
'='		
ICON	"56"	56 对应的符号表入口

操作符 * 和 = 的单词编码是操作符本身，分别是 * 和 = 的数值，没有附加值。标识符 ptr 的单词编码是预定义常量 ID 的值，附加值是标识符串本身的保留副本，即 stringn 函数返回的串以及该标识符的符号表入口。同样，整型常量 56 返回 ICON，附加值是串 "56" 和整型常量 56 的符号表入口。

关键字（如 "for"）都单独编码，以将它们与标识符区分开来。

词法分析器跟踪每个单词的源坐标，这些坐标（参见 3.1 节的定义）给出了文件名、行号和单词的第一个字符在该行中的位置。坐标用于标记发生错误的位置及记录符号的定义点。

词法分析器只是编译器的一部分，负责分解源程序的字符。词法分析器的执行时间通常不能超过编译器总的执行时间的一半，因此，速度非常重要。词法分析器的主要操作是移动字符，尽量减少字符移动的数量有助于提高速度。这可以通过将词法分析器划分为紧密结合的两个模块获得，一个是输入模块 input.c，以大块的方式将输入读入缓冲区；另一个是识别模块 lex.c，检查字符识别单词。

6.1 输入

在大多数程序设计语言中，以行的形式组织输入。尽管在原理上对行的长度很少进行限制，但是实际上行的长度是有限的。另外，在大多数语言中，单词不能跨行。因而，在检查某行时，如果能保证完整的一行都在内存中，就可以用很小的性能代价达到简化词法分析的目的。在 C 语言中，字符串文字是一个例外，可作为特例处理。

输入模块以大块方式读入源程序，块的大小通常超过单独的一行，这有助于在检查单词时，除标识符和字符串文字外，完整的单词都在输入缓冲区中。为了使访问输入的开销最小，输入模块可以输出允许直接访问输入缓冲区的指针：

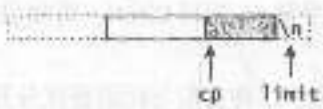
```
(input.c exported data) +=
    extern unsigned char *cp;
    extern unsigned char *limit;
```

cp 指向当前的输入字符，*cp 就是当前字符。limit 指向输入缓冲区的末尾字符的后一个字符，*limit 总是换行符（newline character），作为标记。这些指针都引用无符号字符，例如 *cp 不能对一个值大于 127 的字符进行符号扩展。

这种设计最重要的结果是大多数输入字符通过 *cp 访问，并且许多字符从不用移动。只有出现在可执行代码中的标识符（除了关键字）和字符串文字才从缓冲区复制到永久的存储区。只有在行的边界处才需要调用函数，与输入的字符数相比较，这种调用很少发生。尤其是词法分析器可以使用 *cp++ 读入一个字符并使 cp 加 1。如果 *cp++ 是一个换行符，则必须调用 nextline 函数，重置 cp 和 limit。在调用 nextline 后，如果 cp 等于 limit，则表明已经读到文件尾。

由于 *limit 总是换行符，并且在读到一个换行符后，必须调用 nextline 函数，因此对词法分析器来说，很少需要检查 cp 是否小于 limit。当读到的换行符就是 limit 指向的换行符时，nextline 调用 fillbuf 函数。词法分析器也可以显式地调用 fillbuf，例如在它想要确保一个完整的单词在缓冲区中时。大多数单词较短，少于 32 个字符，因此当 limit-cp 小于 32 时，词法分析器就可以调用 fillbuf。

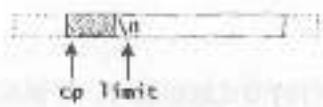
这种协议保证了 fillbuf 函数能够正确处理跨输入缓冲区的行。通常，每个输入缓冲区的结尾只包含某行的一部分（limit-cp 个字符）。为了使行看上去是连续的，减少不必要的搜索，fillbuf 将这部分 limit-cp 个字符移动到输入缓冲区的尚未处理的字符之前的存储空间，这样当输入缓冲区重填时，移动的字符可以与行的尾部字符连接。我们可以用一个例子来表述这一过程，假设输入缓冲区的状态如下：



阴影部分表示尚未处理的字符，\n 表示换行。如果调用 fillbuf，该函数移动输入缓冲区中尚未处理的尾部并重新填满缓冲区。缓冲区的结果状态如下：



深色阴影表示新读入的字符，浅色阴影表示被 fillbuf 移动的字符。当调用 fillbuf，到达输入末尾时，缓冲区状态变为：



最后，当读到最后一个 *limit 标记，调用 nextline 时，fillbuf 将 cp 和 limit 置为相同，标志文件的结束（在第一次调用 nextline 之后）。缓冲区的最终状态为：



input.c 输出的其余的全局变量包括：

```
⟨input.c exported data⟩+≡                                    78
  extern int infd;
  extern char *firstfile;
  extern char *file;
  extern char *line;
  extern int lineno;
```

infd 表示输入模块读入的源文件的文件描述符，默认值是 0，表示标准输入。file 是当前输入的文件名；如果文件被送到缓冲区，line 给出当前行的起始位置；lineno 表示当前行的行号。这样，以 cp 开始的单词的坐标（f，x，y），其中 f 是文件名，3 个参数分别由 file、cp-line、lineno 给出，每行中的字符从 0 开始计数。line 仅用于计算 x 坐标（tab 作为一个字符计数）。firstfile 给出输入的第一个源文件名，用于输出错误消息。

输入缓冲区局部于输入模块：

```
⟨input.c exported macros⟩≡
  #define  MAXLINE  512
  #define  BUFSIZE 4096

⟨input.c data⟩≡
  static int bsize;
  static unsigned char buffer[MAXLINE+1 + BUFSIZE+1];
```

BUFSIZE 是输入缓冲区的大小，输入缓冲区存放读入的字符。MAXLINE 表示在输入缓冲区中尚未处理的尾部可包含的最大字符数。如果 limit-cp 大于 MAXLINE，不会调用 fillbuf 函数。尽管 C 标准指出编译器不必处理超过 509 个字符的行，但是 lcc 可以处理任意长度的行，只是要求标识符和字符串文字的长度不能超过 512 个字符。

bsize 值的编码代表了 3 种不同的输入状态：如果 bsize 小于 0，表示没有任何字符被读入，遇到了读入错误；如果 bsize 等于 0，表示到达输入末尾；如果 bsize 大于 0，表示读入了 bsize 个字符。这一相对复杂的编码保证 lcc 可以被正确地初始化，而且读类作不会超过输入的末尾。

inputInit 函数初始化输入变量并填充输入缓冲区：

```
⟨input.c functions⟩≡                                         79
  void inputInit() {
     limit = cp = &buffer[MAXLINE+1];
     bsize = -1;
     lineno = 0;
     file = NULL;
     ⟨refill buffer 80⟩
     nextline();
  }
```

当 *cp++ 读到换行符时，调用 nextline 函数。如果 cp 大于或等于 limit，输入缓冲区为空。

```
⟨input.c functions⟩+≡                                        79 80
  void nextline() {
     do {
        if (cp >= limit) {
           ⟨refill buffer 80⟩
           if (cp == limit)
```

```
            return;
        } else
            lineno++;
        for (line = (char *)cp; *cp==' ' || *cp=='\t'; cp++)
            ;
    } while (*cp == '\n' && cp == limit);
    if (*cp == '#') {
        resynch();
        nextline();
    }
}
```

如果在填充缓冲区后 cp 仍等于 limit，则说明已经到达文件尾。do-while 循环使 cp 指向行中第一个非空白字符，换行标记作为空白看待。nextline 函数的最后 4 行代码检查预处理器产生的重同步（resynchronization）指导命令，参见练习 6.2。inputInnit 和 nextline 函数都调用 fillbuf 函数填充输入缓冲区：

```
⟨refill buffer 80⟩=                                    79
    fillbuf();
    if (cp >= limit)
        cp = limit;
```

如果没有新的输入，当 fillbuf 返回时，cp 仍然大于或等于 limit，各变量的状态如前面（见第 78 页）最后一幅图所示。fillbuf 函数完成所有的缓冲区管理和实际的输入操作：

```
⟨input.c functions⟩+≡                                  79
    void fillbuf() {
        if (bsize == 0)
            return;
        if (cp >= limit)
            cp = &buffer[MAXLINE+1];
        else
            ⟨move the tail portion 80⟩
        bsize = read(infd, &buffer[MAXLINE+1], BUFSIZE);
        if (bsize < 0) {
            error("read error\n");
            exit(1);
        }
        limit = &buffer[MAXLINE+1+bsize];
        *limit = '\n';
    }
```

fillbuf 读入 BUFSIZE（实际可能少于 BUFSIZE）个字符到缓冲区的 MAXLINE+1 以后的空间，重置 limit，存储换行标记。如果调用 fillbuf 时输入缓冲区为空，cp 指向读入的第一个新字符；否则，移动缓冲区尾部 limit-cp 个字符，使尾部最后一个字符放在 buffer[MAXLINE] 中，后面接着是新读入的字符。

```
⟨move the tail portion 80⟩=                            80
    {
        int n = limit - cp;
        unsigned char *s = &buffer[MAXLINE+1] - n;
        line = (char *)s - ((char *)cp - line);
```

```
    while (cp < limit)
        *s++ = *cp++;
    cp = &buffer[MAXLINE+1] - n;
}
```

注意 line 的计算：由于 line 考虑了当前行中已被处理的部分，所以 cp-line 给出了字符 *cp 正确的索引。

6.2 单词的识别

单词的识别有两个主要技术：创建一个有限自动机（finite automaton）或手工编写一个专门的识别器。大多数程序设计语言的词法结构可以由正则表达式（regular expression）描述，利用正则表达式可以构建一个确定的有限自动机（deterministic finite automaton）来识别及返回单词符号。这种方法的优点在于它的自动化。例如，程序 LEX 就使用了词法规则的正则表达式，生成一个自动机和相应的解释程序。

大多数语言的词法结构相当简洁，可以很容易地手工构造词法分析器。另外，自动生成的分析器，如 LEX 生成的分析器，比手工创建的分析器规模更庞大，速度更慢。但是诸如 LEX 的自动工具，对那些临时性的程序和具有复杂词法结构的应用非常有用。

例如 C 语言中单词分为 6 类，EBNF 文法定义分别为：

```
token:
    keyword
    identifier
    constant
    string-literal
    operator
    punctuator

punctuator:
    one of [ ] ( ) { } * , : = ; ...
```

空白包括空格、制表符、换行和注释，起到分隔单词的作用，例如分隔相邻的标识符。除了出现在字符串文字中外，其余地方出现的空白都被忽略。

词法分析器输出 2 个函数和 4 个变量：

```
⟨lex.c exported functions⟩=
    extern int getchr ARGS((void));
    extern int gettok ARGS((void));

⟨lex.c exported data⟩=
    extern int t;
    extern char *token;
    extern Symbol tsym;
    extern Coordinate src;
```

gettok 函数返回下一个单词，getchr 函数返回下一个待处理的非空白字符。gettok 返回的值可以是字符本身（对于单字符单词），或是枚举常量（对于关键字，如 IF），或是表示其他单词的下列预定义常量：

ID	标识符
FCON	浮点常量
ICON	整型常量
SCON	字符串常量
INCR	++
DECR	--
DEREF	->
ANDAND	&&
OROR	||
LEQ	<=
EQL	==
NEQ	!=
GEQ	>=
RSHIFT	>>
LSHIFT	<<
ELLIPSIS	...
EOI	输入结束

这些常量通过以下方式定义：

```
⟨lex.c exported types⟩=
  enum {
  #define xx(a,b,c,d,e,f,g) a=b,
  #define yy(a,b,c,d,e,f,g)
  #include "token.h"
     LAST
  };
```

文件 token.h 共有 256 行，每行的形式如下：

```
⟨token.h 82⟩=                                                    82
  yy(0,          0, 0,  0,    0,      0,      0)
  xx(FLOAT,      1, 0,  0,    0,      CHAR,   "float")
  xx(DOUBLE,     2, 0,  0,    0,      CHAR,   "double")
  xx(CHAR,       3, 0,  0,    0,      CHAR,   "char")
  xx(SHORT,      4, 0,  0,    0,      CHAR,   "short")
  xx(INT,        5, 0,  0,    0,      CHAR,   "int")
  xx(UNSIGNED,   6, 0,  0,    0,      CHAR,   "unsigned")
  xx(POINTER,    7, 0,  0,    0,      0,      0)
  xx(VOID,       8, 0,  0,    0,      CHAR,   "void")
  xx(STRUCT,     9, 0,  0,    0,      CHAR,   "struct")
  xx(UNION,     10, 0,  0,    0,      CHAR,   "union")
  xx(FUNCTION,  11, 0,  0,    0,      0,      0)
  xx(ARRAY,     12, 0,  0,    0,      0,      0)
  xx(ENUM,      13, 0,  0,    0,      CHAR,   "enum")
  xx(LONG,      14, 0,  0,    0,      CHAR,   "long")
  xx(CONST,     15, 0,  0,    0,      CHAR,   "const")
  xx(VOLATILE,  16, 0,  0,    0,      CHAR,   "volatile")

⟨token.h 82⟩+=                                                   82
```

```
yy(0,       42, 13, MUL,   multree,ID,      "*")
yy(0,       43, 12, ADD,   addtree,ID,      "+")
yy(0,       44, 1,  0,     0,       ',',    ",")
yy(0,       45, 12, SUB,   subtree,ID,      "-")
yy(0,       46, 0,  0,     0,       '.',    ".")
yy(0,       47, 13, DIV,   multree,'/',     "/")
xx(DECR,    48, 0,  SUB,   subtree,ID,      "--")
xx(DEREF,   49, 0,  0,     0,       DEREF,  "->")
xx(ANDAND,  50, 5,  AND,   andtree,ANDAND,  "&&")
xx(OROR,    51, 4,  OR,    andtree,OROR,    "||")
xx(LEQ,     52, 10, LE,    cmptree,LEQ,     "<=")
```

token.h 使用宏收集每个单词或符号常量的信息。token.h 的每行中，单词的 7 个有用信息作为 xx 或 yy 的参数。单词编码由第 2 列的值给出。在定义符号时读入 token.h，以单词为下标创建数组，保证定义的一致性。这种技术在汇编程序设计中非常普遍。

yy 行处理单字符单词，xx 行处理多字符单词和其他定义。xx 的第 1 列是枚举标识符。其他列分别表示标识符或字符值、单词操作符的优先次序（参见 8.3 节）、通用操作符（参见 5.5 节）、创建树的函数（参见 9.4 节）、单词的集合（参见 7.6 节）以及字符串表示。

通过重新定义宏 xx 和 yy 以及包含 token.h 文件，就可以根据不同的目的抽取不同的列。上述枚举定义展现了这种技术。在枚举定义中，通过定义宏 xx，每个宏展开就定义一个枚举成员。例如，xx 行将 DECR 扩展为：

```
DECR=48,
```

这样，就将 DECR 定义成值为 48 的枚举常量。yy 的定义为空，这种定义可以高效地忽略 yy 行。

通常使用全局变量 t 保存当前单词，因此大多数对 gettok 的调用都使用：

```
t = gettok();
```

变量 token、tsym 和 src 保存与当前单词相关的值。其中，token 是单词自身的原文本；tsym 为某些单词存放 Symbol，例如标识符和常量；src 表示当前单词在源程序中的坐标。

也可以让 gettok 返回一个包含单词编码和附加值的结构，或者返回一个指向该结构的指针。但是由于大多数调用只是检查单词编码，因此将这些值封装在一起并不能明显地提高性能，所以 lcc 中 gettok 只返回单词编码，用全局变量 token、tsym 和 src 存放其他相关值。另外，gettok 是 lcc 编译器最常调用的函数，简单的接口可以增加代码的可读性。

一般可以通过单词的第一个字符将单词分类，加上随后的字符形成完整的单词。gettok 函数通过第一个字符识别单词。对某些单词而言，这些字符可以由一个或多个通过 map 定义的集合给出。掩码 map[c] 可以将字符 c 归为下面 6 种集合中的一个或多个：

```
⟨lex.c types⟩=
  enum { BLANK=01,  NEWLINE=02, LETTER=04,
         DIGIT=010, HEX=020,    OTHER=040 };

⟨lex.c data⟩=                                          89
  static unsigned char map[256] = { ⟨map initializer⟩ };
```

如果 c 是一个非换行符的空白字符，则 map[c]&BLANK 非零。换行符被排除在外是由于换行符可能导致 gettok 调用 nextline 函数。其他枚举值表示其他字符子集：NEWLINE 是只包含换行符

的集合，LETTER 是包含大写和小写字母的集合，DIGIT 是包含数字 0 - 9 的集合，HEX 是包含数字 0 - 9、a - f 和 A - F 的集合，OTHER 是包含标准允许的、出现在源程序和执行字符集的其他 ASCII 字符。如果 map[c] 为零，则不能保证字符 c 能被所有 ANSI C 编译器接受，例如 $，@ 和 '。

gettok 是一个相当大的函数，但是使用 switch 语句根据单词的第一个字符进行分别处理，可以将 gettok 的代码划分为若干易处理的片段：

```
⟨lex.c macros⟩=
  #define MAXTOKEN 32

⟨lex.c functions⟩=                                              89
  int gettok() {
     for (;;) {
        register unsigned char *rcp = cp;
        ⟨skip white space 84⟩
        if (limit - rcp < MAXTOKEN) {
           cp = rcp;
           fillbuf();
           rcp = cp;
        }
        src.file = file;
        src.x = (char *)rcp - line;
        src.y = lineno;
        cp = rcp + 1;
        switch (*rcp++) {
        ⟨gettok cases 85⟩
        default:
           if ((map[cp[-1]]&BLANK) == 0)
              ⟨illegal character⟩
        }
     }
  }
```

gettok 首先忽略空白字符，然后检查输入缓冲区中是否至少包含一个单词。如果没有，则调用 fillbuf 补充缓冲区，保证其中至少包含一个单词。除标识符、字符串和数字常量外，其他所有单词的长度不超过 MAXTOKEN。超过 MAXTOKEN 的单词由为这些单词专门设计的代码显式处理。ANSI 标准允许编译器限制字符串长度小于 509，标识符少于 31 个字符。lcc 敢于违反这些限制，字符串和标识符长度分别可以达到 4096（BUFSIZE）和 512（MAXLINE），满足那些自动生成 C 程序的程序的需要，因为这些自动生成的 C 程序可能包含很长的标识符。

gettok 没有像 6.1 节建议的那样使用 cp，而是在函数入口处将 cp 复制到一个寄存器变量 rcp，在单词识别过程中使用变量 rcp。在函数返回前以及调用 nextline 和 fillbuf 之前，将 rcp 复制回 cp。使用 rcp 可以提高程序性能，使扫描循环更紧凑和快速。例如，省略空白的代码为：

```
⟨skip white space 84⟩=                                          84
  while (map[*rcp]&BLANK)
     rcp++;
```

使用寄存器变量作为 map 的下标，可以生成更高效的代码。这些类型的扫描检查输入的每一个字符，通过直接访问输入缓冲区检查字符，一些优化编译器能够做类似的局部改进，但不会超过潜在的别名赋值以及调用其他不相关的函数。

下面各节分别描述了 <gettok cases> 的各种情况的处理。本书省略的情况包括：

```
⟨gettok cases 85⟩=                                          85  84
  case '/': ⟨comment or /⟩
  case 'L': ⟨wide-character constants⟩
  ⟨cases for two-character operators⟩
  ⟨cases for one-character operators and punctuation⟩
```

gettok 遇到一个换行或等价于换行的字符时，调用 nextline 函数：

```
⟨gettok cases 85⟩+=                                    85 85  84
  case '\n': case '\v': case '\r': case '\f':
    nextline();
    if (⟨end of input 85⟩) {
      tsym = NULL;
      return EOI;
    }
    continue;

⟨end of input 85⟩=                                          85  94
  cp == limit
```

当控制遇到这一情况，进行如下处理：将 cp 指向换行符后的字符；当 nextline 返回后，cp 仍指向那个字符，且小于 limit。只有处理到文件结束时 cp 等于 limit。由于 *cp 总是一个换行符，对大多数单词符号而言，此时可以中止扫描，因此很少需要测试是否满足 cp—limit 的条件。

以下各节描述了其余的情况。单词符号的识别比较简单，但为单词符号计算附加值则使得各种情况的处理会复杂一些。

6.3 关键字的识别

共有 28 个关键字：

```
keyword: one of
   auto       double     int        struct
   break      else       long       switch
   char       extern     return     union
   const      float      short      unsigned
   continue   for        signed     void
   default    goto       sizeof     volatile
   do         if         static     while
```

关键字的识别可以通过查表完成，表中每个关键字的入口都包含它的单词编码，每个固有类型的入口包含它的类型。关键字还可以通过硬编码（hard-coded）的决策树识别，决策树的方法比搜索表更快，并且几乎同样简单。由于完整的关键字一定都在输入缓冲区中，可以根据关键字的起始小写字母测试关键字。例如，当起始小写字母为 i 时：

```
⟨gettok cases 85⟩+=                                    85 86  84
  case 'i':
    if (rcp[0] == 'f'
    && !(map[rcp[1]]&(DIGIT|LETTER))) {
      cp = rcp + 1;
      return IF;
```

```
}
if (rcp[0] == 'n'
&&  rcp[1] == 't'
&& !(map[rcp[2]]&(DIGIT|LETTER))) {
    cp = rcp + 2;
    tsym = inttype->u.sym;
    return INT;
}
goto id;
```

标号 id 标记的代码将在下一节描述，用于扫描标识符。如果单词符号是 if 或 int，则修改 cp，返回相应的单词编码；否则该单词就是标识符。如果是 int，tsym 包含了类型 int 的符号表入口。字符串 ahcdcfglrsuvw 的处理情况类似，该串由一个小型序自动生成。。

这些程序片段生成的目标代码短小且速度快。例如，在大多数机器上，int 的识别所需指令少于 12 条。即使使用完美的 hash 算法，通过查表的方式识别一个关键字，也需要更多的指令。

6.4 标识符的识别

标识符的语法如下：

identifier:
  nondigit { *nondigit* | *digit* }

digit:
  one of 0 1 2 3 4 5 6 7 8 9

nondigit:
  one of _
  a b c d e f g h i j k l m
  n o p q r s t u v w x y z
  A B C D E F G H I J K L M
  N O P Q R S T U V W X Y Z

lcc 标识符识别代码符合这一语法，但必须具有处理长于 MAXTOKEN 的标识符的能力，而这些标识符可能跨输入缓冲区。

```
(gettok cases 85 )+=                                    83 88  84
  case 'h': case 'j': case 'k': case 'm': case 'n': case 'o':
  case 'p': case 'q': case 'x': case 'y': case 'z':
  case 'A': case 'B': case 'C': case 'D': case 'E': case 'F':
  case 'G': case 'H': case 'I': case 'J': case 'K':
  case 'M': case 'N': case 'O': case 'P': case 'Q': case 'R':
  case 'S': case 'T': case 'U': case 'V': case 'W': case 'X':
  case 'Y': case 'Z': case '_':
  id:
     (ensure there are at least MAXLINE characters 87 )
     token = (char *)rcp - 1;
     while (map[*rcp]&(DIGIT|LETTER))
        rcp++;
     token = stringn(token, (char *)rcp - token);
     (tsym ← type named by token 87 )
     cp = rcp;
     return ID;
```

所有的标识符都保存在字符串表中。在本分支和所有分支情况下，cp 和 rcp 都加 1 跳过了单词的第一个字符。如果输入缓冲区的字符少于 MAXLINE，则首先 cp 回退一个字符，指向该标识符的第一个字符，调用 fillbuf 补充输入缓冲区，然后调整 cp 和 rcp，使二者如同先前一样，都指向标识符的第二个字符。

```
ensure there are at least MAXLINE characters 87 )=                86 88 91
 if (limit - rcp < MAXLINE) {
    cp = rcp - 1;
    fillbuf();
    rcp = ++cp;
 }
```

typedef 为一个类型创建一个标识符作为同义词，这些名字安装在标识符表中。如果标识符有符号表入口，gettok 设置 tsym 指向符号表的入口。

```
(tsym ← type named by token 87 )=                                  86
  tsym = lookup(token, identifiers);
```

如果某个单词命名了一个类型，tsym 就被设置为类型的符号表入口，设置 tsym->sclass 等于 TYPEDEF。否则，tsym 为空或标识符不是类型名。宏 istypename 测试当前单词是否为类型名：

```
(lex.c exported macros)=
  #define istypename(t,tsym) (kind[t] == CHAR \
     || t == ID && tsym && tsym->sclass == TYPEDEF)
```

类型名或者是命名类型的关键字（如 int），或者是使用 typedef 定义的类型标识符。全局变量 t 和 tsym 是 istypename 的唯一合法参数。

6.5 数字的识别

ANSI C 包含 4 种数字常量：

```
constant:
   floating-constant
   integer-constant
   enumeration-constant
   character-constant

enumeration-constant:
   identifier
```

上节展示的标识符的识别代码也能处理枚举常量，6.6 节的代码处理字符常量。对于枚举常量，词法分析器返回单词编码 ID，设置 tsym 指向枚举常量的符号表入口。调用者检查枚举常量，在枚举常量出现的地方用相应的整数替换。8.8 节的代码就是这种转换的一个实例。

整型常量分为 3 种：

```
integer-constant:
   decimal-constant [ integer-suffix ]
   octal-constant [ integer-suffix ]
   hexadecimal-constant [ integer-suffix ]

integer-suffix:
   unsigned-suffix [ long-suffix ]
   long-suffix [ unsigned-suffix ]
```

unsigned-suffix: u | U

long-suffix: l | L

整型常量的前几个字符可以帮助识别它的种类：

```
(gettok cases 85 )+=                                          86 91    84
  case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9': {
     unsigned int n = 0;
     (ensure there are at least MAXLINE characters 87 )
     token = (char *)rcp - 1;
     if (*token == '0' && (*rcp == 'x' || *rcp == 'X')) {
        (hexadecimal constant)
     } else if (*token == '0') {
        (octal constant)
     } else {
        (decimal constant 88 )
     }
     return ICON;
  }
```

与标识符识别一样，首先保证输入缓冲区中的字符不少于 MAXLINE 个，使得代码可以超前搜索，例如判断十六进制常量。

3 种整型常量都将常量值保存在 n 中，代码不仅能够识别常量，还要保证常量在可表示的整型范围之内。

下面以十进制常量的识别为例说明了识别过程。十进制常量的语法是：

decimal-constant:
  nonzero-digit { *digit* }

nonzero-digit:
  one of 1 2 3 4 5 6 7 8 9

代码反复执行乘法，累计的十进制值保存在 n 中：

```
(decimal constant 88 )=                                          88
  int overflow = 0;
  for (n = *token - '0'; map[*rcp]&DIGIT; ) {
     int d = *rcp++ - '0';
     if (n > ((unsigned)UINT_MAX - d)/10)
        overflow = 1;
     else
        n = 10*n + d;
  }
  (check for floating constant 89 )
  cp = rcp;
  tsym = icon(n, overflow, 10);
```

UINT_MAX 为可表示的最大的无符号数的值。在每一步，如果 10*n+d>UINT_MAX，则触发溢出。上面的代码对该不等式进行了变换，测试 n>((unsigned)UINT_MAX - d)/10，并在计算 n 的新值之前判断是否溢出。如果常量溢出，overflow 置为 1。icon 函数处理可选的后缀。

如果一个十进制常量的下一个字符是小数点或一个指数指示符，则该常量是一个浮点常量的前缀：

```
⟨check for floating constant 89⟩=                                    88
  if (*rcp == '.' || *rcp == 'e' || *rcp == 'E') {
     cp = rcp;
     tsym = fcon();
     return FCON;
  }
```

fcon 与 icon 类似，识别浮点常量的后缀。如果浮点常量的整体超过 UINT_MAX，overflow 置为 1，但是 n 和 overflow 都不传递给 fcon，fcon 再次检查单词符号，测试浮点溢出。

icon 识别可选的 U 和 L 后缀，警告发生溢出的值，用相应的类型和值初始化一个符号，返回指向该符号的指针。

```
⟨lex.c data⟩+≡                                                       83
  static struct symbol tval;
```

tval 的作用只是为 gettok 的调用者提供常量的类型和值。在再次调用 gettok 之前，调用者必须撤销相应的数据。

```
⟨lex.c functions⟩+≡                                                  84 90
  static Symbol icon(n, overflow, base)
  unsigned n; int overflow, base; {
     if ((*cp=='u'||*cp=='U') && (cp[1]=='l'||cp[1]=='L')
     ||  (*cp=='l'||*cp=='L') && (cp[1]=='u'||cp[1]=='U')) {
        tval.type = unsignedlong;
        cp += 2;
     } else if (*cp == 'u' || *cp == 'U') {
        tval.type = unsignedtype;
        cp += 1;
     } else if (*cp == 'l' || *cp == 'L') {
        if (n > (unsigned)LONG_MAX)
           tval.type = unsignedlong;
        else
           tval.type = longtype;
        cp += 1;
     } else if (base == 10 && n > (unsigned)LONG_MAX)
        tval.type = unsignedlong;
     else if (n > (unsigned)INT_MAX)
        tval.type = unsignedtype;
     else
        tval.type = inttype;
     if (overflow) {
        warning("overflow in constant '%S'\n", token,
           (char*)cp - token);
        n = LONG_MAX;
     }
     ⟨set tval's value 90⟩
     ppnumber("integer");
     return &tval;
  }
```

如果后缀中U和L都出现，则n的类型是unsigned long；若只出现U，n的类型是unsigned；如果只有L，则n的类型是long，除非n的值非常大（大于LONG_MAX），这时n的类型就是unsignedlong。如果十进制常量没有任何后缀，则当n的值非常大时其类型是unsigned long，否则是long。没有后缀的八进制和十六进制的常量类型通常为int，在常量值非常大时为unsigned。格式代码%S类似于printf的%s，打印一个字符串，但包含一个额外的参数指定字符串的长度。因此可以打印那些不是以空字符为结束的字符串。

类型int、long、unsigned应为不同的类型，但lcc处理时将它们的大小定为一样的。这一限制简化了上述测试。设置tval的值的代码可以简化为：

```
(set tval's value 90 )≡                                          89
  if (isunsigned(tval.type))
     tval.u.c.v.u = n;
  else
     tval.u.c.v.i = n;
```

放宽这一限制将会使这段代码及上述测试更复杂。例如，C标准指定无后缀十进制常量根据它的值可以是int、long、unsigned long。在lcc中，由于int和long表示相同范围的整数，一个无后缀的十进制常量只能是int或unsigned。

数字常量由预处理数字（preprocessing number）形成。预处理数字是C预处理器识别的数字常量。遗憾的是，语言标准规定的预处理数字是整型和浮点常量的超集，这样，合法的预处理数字很可能不是合法的数字常量。例如123.4.5，预处理器也可以处理这样的数，并可能把它们传递给编译器，编译器将它们作为一个单词符号处理，导致预处理数字不是合法的常量。

预处理数字的语法如下：

pp-number:
  [.] *digit* { *digit* | . | *nondigit* | E *sign* | e *sign* }
sign: - | +

预处理数字的前缀是合法的常量，这样即使处理的是错误的预处理数字，icon和fcon也可能在没有处理完整个预处理数字的情况下，成功返回。icon和fcon调用ppnumber函数处理这种情况。

```
(lex.c functions)+≡                                          ▲89 91▼
  static void ppnumber(which) char *which; {
     unsigned char *rcp = cp--;

     for ( ; (map[*cp]&(DIGIT|LETTER)) || *cp == '.'; cp++)
        if ((cp[0] == 'E' || cp[0] == 'e')
        &&  (cp[1] == '-' || cp[1] == '+'))
           cp++;
     if (cp > rcp)
        error("'%S' is a preprocessing number but an _
           invalid %s constant\n", token,
           (char*)cp-token, which);
  }
```

ppnumber回退一个字符，跳过那些可能包含预处理数字的字符。如果它扫描超过数字单词符号的结束点，则表明发生了错误。

fcon识别浮点常量的后缀，在两个地方被调用，一次出现在上述<check for floating constant>中，另一次出现在gettok处理'.'的分支情况中：

```
⟨gettok cases 85⟩+≡                                    ↑88 ↓93 84
  case '.':
     if (rcp[0] == '.' && rcp[1] == '.') {
        cp += 2;
        return ELLIPSIS;
     }
     if ((map[*rcp]&DIGIT) == 0)
        return '.';
     ⟨ensure there are at least MAXLINE characters 87⟩
     cp = rcp - 1;
     token = (char *)cp;
     tsym = fcon();
     return FCON;
```

浮点常量的语法为：

floating-constant:
  fractional-constant [*exponent-part*] [*floating-suffix*]
  digit-sequence exponent-part [*floating-suffix*]

fractional-constant:
  [*digit-sequence*] . *digit-sequence*
  digit-sequence .

exponent-part:
  e [*sign*] *digit-sequence*
  E [*sign*] *digit-sequence*

digit-sequence:
  digit [*digit*]

floating-suffix:
  one of f l F L

fcon 识别浮点常量，将 token 转换为双精度值，确定 tval 的类型和值。

```
⟨lex.c functions⟩+≡                                    ↑90
  static Symbol fcon() {
     ⟨scan past a floating constant 92⟩
     errno = 0;
     tval.u.c.v.d = strtod(token, NULL);
     if (errno == ERANGE)
        ⟨warn about overflow 91⟩
     ⟨set tval's type and value 92⟩
     ppnumber("floating");
     return &tval;
  }

⟨warn about overflow 91⟩≡                              91 92
  warning("overflow in floating constant '%S'\n", token,
     (char*)cp - token);
```

C 库函数 strtod 将它的第一个串参数解释为浮点常量，返回相应的双精度值。如果常量超过范围，strtod 将全局变量 errno 置为 ERANGE，这是 ANSI C 规范对 C 库的要求。

上述语法定义的浮点的识别如下：

```
(scan past a floating constant 92 )=                                    91
  if (*cp == '.')
     (scan past a run of digits 92 )
  if (*cp == 'e' || *cp == 'E') {
     if (*++cp == '-' || *cp == '+')
        cp++;
     if (map[*cp]&DIGIT)
        (scan past a run of digits 92 )
     else
        error("invalid floating constant '%S'\n", token,
           (char*)cp - token);
  }

(scan past a run of digits 92 )=                                        92
  do
     cp++;
  while (map[*cp]&DIGIT);
```

与语法规定一致，指数指示符后必须跟随至少一个数字。

浮点常量的后缀可以包含F或L（但二者不能同时出现），分别指定常量类型为float或long double。

```
(set tval's type and value 92 )=                                        91
  if (*cp == 'f' || *cp == 'F') {
     ++cp;
     if (tval.u.c.v.d > FLT_MAX)
        (warn about overflow 91 )
     tval.type = floattype;
     tval.u.c.v.f = tval.u.c.v.d;
  } else if (*cp == 'l' || *cp == 'L') {
     cp++;
     tval.type = longdouble;
  } else
     tval.type = doubletype;
```

6.6 字符常量和字符串的识别

转义序列（escape sequence）（如\n、\034、\xFF和\"）和宽字符常量（wide-character）使字符常量和字符串文字的识别变得复杂。lcc将所谓的宽字符作为普通的ASCII字符实现，类型wchar_t使用无符号字符实现。语法为：

character-constant:
  [L] '*c-char* [*c-char*] '

c-char:
  any character except ', \, or newline
  escape-sequence

escape-sequence:
  one of \' \" \? \\ \a \b \f \n \r \t \v
  \ *octal-digit* [*octal-digit* [*octal-digit*]]

\x *hexadecimal-digit* { *hexadecimal-digit* }

string-literal:
[L] "{ *s-char* }"

s-char:
any character except ", \, or newline
escape-sequence

利用换行符之前紧接一个反斜线符 /，字符串文字能够实现跨行。邻接的字符串文字自动结合在一起形成单个的字符串文字。在严格的 ANSI C 实现中，行的接合（splicing）和字符串文字的连接（concatenation）由预处理器处理，编译器看到的只是一个不间断的字符串文字。lcc 为字符串文字实现了行的接合和字符串文字的连接，可以和 ANSI 之前的预处理器一起使用。

实现这些特征意味着字符串文字的长度可能超过 MAXLINE 个字符，因此 <ensure there are at least MAXLINE characters> 不能用来保证邻接的整个字符串文字序列都出现在输入缓冲区中。相反，代码必须显式测试在 limit 的换行符，调用 nextline 函数，并且将文字复制到一个私有缓冲区中。

```
⟨gettok cases 85 ⟩+≡                                    91  84
  scon:
  case '\'': case '"': {
     static char cbuf[BUFSIZE+1];
     char *s = cbuf;
     int nbad = 0;
     *s++ = *--cp;
     do {
        cp++;
        ⟨scan one string literal 94 ⟩
        if (*cp == cbuf[0])
           cp++;
        else
              error("missing %c\n", cbuf[0]);
        } while (cbuf[0] == '"' && getchr() == '"');
        *s++ = 0;
        if (s >= &cbuf[sizeof cbuf])
           error("%s literal too long\n",
              cbuf[0] == '"' ? "string" : "character");
        ⟨warn about non-ANSI literals⟩
        ⟨set tval and return ICON or SCON 93 ⟩
     }
```

字符串文字可通过前导字符——双引号识别，外层 do-while 循环收集随后的字符串文字，复制到 cbuf 中，并报告长度过长的情况。前导字符也决定了附加值和 gettok 返回值的类型：

```
⟨set tval and return ICON or SCON 93 ⟩≡                  93
  token = cbuf;
  tsym = &tval;
  if (cbuf[0] == '"') {
     tval.type = array(chartype, s - cbuf - 1, 0);
     tval.u.c.v.p = cbuf + 1;
     return SCON;
  } else {
```

```
    if (s - cbuf > 3)
        warning("excess characters in multibyte character _
            literal '%S' ignored\n", token, (char*)cp-token);
    else if (s - cbuf <= 2)
        error("missing '\n");
    tval.type = inttype;
    tval.u.c.v.i = cbuf[1];
    return ICON;
}
```

转义序列 \0 使字符串文字可以包含空字符，因此文字长度由其类型给出：一个包含 n 个字符的文字的类型为 (ARRAY n (CHAR))(n 不包括双引号)。gettok 函数的调用者，例如 primary 函数，如果需要保存 tval 引用的字符串文字，可以通过调用 stringn 函数实现。

下列代码扫描一个字符串文字或字符常量，处理 4 种情况：limit 处有换行符、转义序列、非 ANSI 字符和超过 cbuf 大小的文字。

```
⟨scan one string literal 94⟩=                                    93
while (*cp != cbuf[0]) {
    int c;
    if (map[*cp]&NEWLINE) {
        if (cp < limit)
            break;
        cp++;
        nextline();
        if (⟨end of input 85⟩)
            break;
        continue;
    }
    c = *cp++;
    if (c == '\\') {
        if (map[*cp]&NEWLINE) {
            if (cp < limit)
                break;
            cp++;
            nextline();
        }
        if (limit - cp < MAXTOKEN)
            fillbuf();
        c = backslash(cbuf[0]);
    } else if (map[c] == 0)
        nbad++;
    if (s < &cbuf[sizeof cbuf] - 2)
        *s++ = c;
}
```

如果 *limit 是一个换行符，仅起到指示缓冲区末尾的作用，因此该换行符被忽略，除非后面没有输入。其他的换行符（当 cp 小于 limit 时出现的换行符）和文件尾的一个换行符将终止 while 循环，此时 cp 不会加 1。backslash 函数解释上面介绍的转义序列，参见练习 6.10。nbad 变量记录串中出现的非 ANSI 字符的个数。如果串中出现非 ANSI 字符或者串长度超过 ANSI 规定的 509 个字符，lcc 编译选项 -A -A 会产生警告信息。

深入阅读

lcc 的输入模块基于 Waite（1986）所介绍的设计。不同的地方在于 Waite 的算法一次只移动一个部分行，而不是可能的几个部分行或单词，而且该操作是在扫描到缓冲区中第一个换行符后进行的。如果一个部分行的长度超过固定的最大值，则该操作会覆盖缓冲区之前的存储空间。lcc 的算法避免了该问题，一次可以移动几个可能的部分行或单词，只是在识别每个单词时必须比较 limit-cp 与 MAXTOKEN。

词法分析器能够通过语言词法结构的正则表达式规范生成。UNIX 上的 LEX（Lesk，1975）是最为熟知的例子。Schreiner and Friedman（1985）在其编译器中利用 LEX，Holub（1990）详述了一个类似工具的实现。更多近期的生成器，如 flex、re2c（Bumbulis and Cowan，1993），以及 ELI 的扫描器生成器（Gray et al.，1992；Heuring，1986），它们产生的词法分析器比 LEX 产生的更快且更小。ELI 的一些技术在 lcc 的 gettok 中也得到应用。

"完美的" hash 函数应能将一个已知集合的每个字映射为不同的 hash 值（Cichelli，1980；Jaeschke and Osterburg，1980；Sager，1985）。一些编译器使用了完美的 hash 算法来识别关键字，但是通常这些 hash 算法本身比 lcc 识别关键字的 hash 算法使用了更多的指令。

lcc 依赖库函数 strtod 将浮点常量的字符串表示转换为相应的双精度值。尽可能精确地实现这一转换相当复杂。Clinger（1990）展示了在一些情况下，转换算法需要任意精度。许多 strtod 的实现就是基于 Clinger 算法。相反的问题是，将一个双精度值转换为它的字符串表示也很困难。Steele 和 White（1990）给出了细节。

练习

6.1 如果输入的一行多于 BUFSIZE 个字符，将会出现什么现象？lcc 能正确处理长度为 0 的行吗？

6.2 C 预处理器生成如下形式的代码行：

```
# n "file"
#line n "file"
#line n
```

这些代码行用于将当前行号和文件名分别重置为 n 和 file，使错误信息能够指向正确的文件和位置。在第二种形式中，只重置行号，文件名保持不变。nextline 函数调用 resynch 函数识别这些代码行，从而重置 file 和 lineno。请实现 resynch 函数。

6.3 在 C 的大多数实现中，预处理器作为单独的程序运行，预处理器的输出作为编译器的输入。将预处理器作为 input.c 整体的一部分实现，并测试性能的改善情况。注意：写一个预处理器是一项大的工作，可能有很多缺陷。ANSI 标准是预处理器的唯一定义规范。

6.4 实现 gettok 函数省略的片段。

6.5 当 lcc 读入的标识符长度大于 MAXLINE 时，将会出现什么现象？

6.6 实现函数 intgetchr（void）。

6.7 尝试实现更好的关键字哈希算法，并与 lcc 现在使用的算法进行比较。

6.8 八进制常量的语法为：

octal-constant:
  0 { *octal-digit* }

octal-digit:
  one of 0 1 2 3 4 5 6 7

编写 <octal constant>。注意：一个八进制常量是浮点常量合法的前缀，并且八进制常量也会溢出。

6.9 十六进制常量的语法为：

hexadecimal-constant:
(`0x` | `0X`) *hexadecimal-digit* { *hexadecimal-digit* }

hexadecimal-digit:
one of `0 1 2 3 4 5 6 7 a b c d e f A B C D E F`

编写 <hexadecimal constant>。注意处理溢出。

6.10 实现下面的函数：

(lex.c prototypes)+≡
```
static int backslash ARGS((int q));
```

该函数解释以 cp 开始的一个转义序列。其中，q 可以是单引号或双引号，以区分字符常量和字符串文字。实现该转义函数。

6.11 实现 <wide-character constants> 的代码。记住 wchar_t 是一个 unsigned char，常量 L'\377' 的值应为 255，而不是 −1。

6.12 利用 LEX 或类似的程序生成器重新实现词法分析器，比较两种实现方法。哪一个更快？哪一个更小？哪一个更容易理解并修改？

6.13 在你的机器中需要用多少条指令完成 <skip white space>？如果使用 cp 代替 rcp，需要多少条指令？

6.14 编写程序为 C 关键字生成 <gettok cases>。

6.15 lcc 假设 int 和 long（符号和无符号）大小相同。如果要取消这一不恰当的假设，如何修改 scan 函数？

语法分析

第6章描述的词法分析为语法分析提供了单词序列。语法分析过程确认输入是否符合语言的文法规则，并建立输入源程序的内部表示。随后的lcc程序遍历内部表示，生成特定目标机器的代码。

lcc采用递归下降（recursive-descent）的语法分析。它是传统手工构造分析方法的一种直接应用。这种方法能产生一个短小而高效的编译器，适于编译C和Pascal之类较为简单的语言。事实上，许多商业编译器的构造都采用了这种方法。

然而，对复杂的语言分析，使用语法分析器生成器更为可取。例如，递归下降分析能识别诸如C之类的语言，但对于像ADA的其他一些语言却无能为力。对于后者，必须使用功能更强大的分析方法，自底向上的分析就属于其中一种。手工构建这类语法分析器非常困难，必须使用自动生成的方法。

本章以形式语言理论为基础，采取语法制导翻译方法，处理程序中的错误，相关代码在随后的章节中给出。

7.1 语言和语法

正如前面章节所介绍的，我们可以用EBNF文法定义语言。大多数重要的语言都是无限的，比如程序设计语言。文法是一种用有限的规则定义无限集合的方法。

文法的产生式给出了生成句子的规则，通过反复使用非终结符的某个产生式的右部替换这个非终结符来产生句子。例如，EBNF文法

```
expr:
  expr + expr
  ID
```

定义了一种具有简单表达式的语言，expr为开始非终结符。在这个语言中，句子的推导过程是：以expr开始，反复用规则的右部替换左部的非终结符。本例中只有两条规则，因此，一种可能的替换是：

$$expr \Rightarrow expr + expr$$

这样的操作过程称为一个推导步（derivation step）。如果存在一组类似的推导步骤并最终推出一个句子，就称这组推导步是关于此句子的一个推导（derivation）。每个推导步都是一个非终结符被其产生式（称为候选式）之一的右部替换。例如，句子ID+ID+ID能通过以下步骤推出：

$$\begin{aligned} expr &\Rightarrow expr + expr \\ &\Rightarrow expr + \mathtt{ID} \\ &\Rightarrow expr + expr + \mathtt{ID} \\ &\Rightarrow \mathtt{ID} + expr + \mathtt{ID} \\ &\Rightarrow \mathtt{ID} + \mathtt{ID} + \mathtt{ID} \end{aligned}$$

第1步，使用产生式

```
expr: expr + expr
```

的右部替换左部的开始非终结符 expr。第 2 步，运用产生式 expr → ID 对第 1 步得到的句型中最右边的 expr 进行替换。后面的 3 步使用以上两种规则，最终得出 ID+ID+ID。可以看到，每一步推导都产生一个由终结符和非终结符组成的句型（sentential form）。句型和句子的区别在于：句型由终结符和非终结符构成，句子只由终结符构成。

在每步推导中，都可以选取句型中任意一个非终结符，然后用相关规则的右部进行替换。如果每一步推导都是句型中最左边的非终结符被替换，则称这样的推导为最左推导（leftmost derivation）。例如：

```
expr ⇒ expr + expr
     ⇒ ID + expr
     ⇒ ID + expr + expr
     ⇒ ID + ID + expr
     ⇒ ID + ID + ID
```

是关于句子 ID+ID+ID 的最左推导。分析器能对一个给定的句子（即输入的 C 程序）重新构造推导。lcc 分析器是一种自顶向下的分析器（top-down parser），它能重构输入中的最左推导。

7.2 二义性和分析树

考虑用下列文法定义的语言：

```
expr:
  expr + expr
  expr * expr
  ID
```

假设 a、b 和 c 为标识符，a+b、a+b+c 和 a+b*c 是该语言的句子。

我们既可以像上面那样写出句子的推导过程，也可以采用分析树（parse tree）的图示方法描述这个过程。例如，关于 a+b+c 的最左推导如下：

```
expr ⇒ expr + expr
     ⇒ expr + expr + expr
     ⇒ a + expr + expr
     ⇒ a + b + expr
     ⇒ a + b + c
```

对应的分析树如图 7-1 左边的树所示。分析树的中间节点用非终结符标记，叶节点用终结符标记，它的根用文法的开始符号标记。如果某个节点标记为非终结符 A，其直接子节点从左到右依次标记为 X_1，X_2，…，X_n，那么 $A: X_1X_2\cdots X_n$ 是一个产生式。

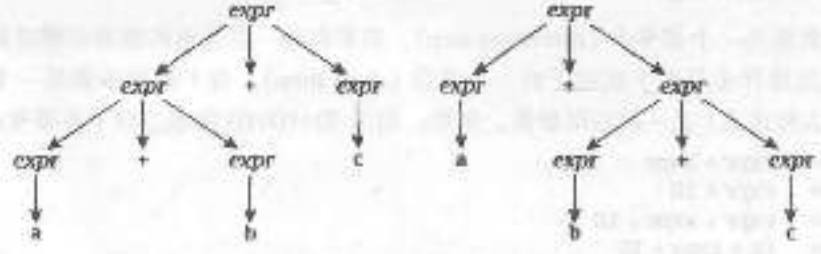


图 7-1 a+b+c 的两棵分析树

如果一个句子能够对应一棵以上的分析树，也就是同时存在几个不同的最左推导，则称该文法是二义的。例如，a+b+c 除上面描述的最左推导外，还有另外一种形式的最左推导，对应的分析树见图 7-1 右边的树。

上例之所以出现二义性问题是因为文法没有体现“+”的通常的左结合次序。a+b+c 的正确解释应该是（a+b）+c，其推导如上，语法分析树为图 7-1 左边的树。

使用 EBNF 的重复结构来重写文法就可以解决二义性问题，新的文法规定 a+b+c 只存在一种推导，即（a+b）+c：

```
expr:
  expr { + expr }
  expr { * expr }
  ID
```

改写文法后，a+b+c 只有一种最左推导。这里，我们首先必须弄清包含重复结构的 EBNF 产生式的含义，才能更好地理解新的推导规则。有如下形式的产生式：

$A: \beta \{ \alpha \}$

它表示 A 能推导出一个以 β 开头，且 β 后紧跟有零个或多个 α 的句型，这种文法也可以描述为：

$A: \beta X$

$X: \epsilon \mid \alpha X$

这里，X 能推出空串（记为 ε），或者 αX。先使用 A 的产生式，再反复使用 X 的产生式，就可以导出以 β 开头、后跟零个或多个 α 的句型。EBNF 的重复结构省略了许多像 X 那样的隐藏的非终结符，但要注意，这些隐藏的非终结符必须包含在语法分析树中。很容易对文法再次重写，引入这些非终结符。下面是加入非终结符后形成的新文法：

```
expr:
  expr X
  expr Y
  ID
X: ε | + expr X
Y: ε | * expr Y
```

修改后，a+b+c 只对应一种最左推导，即

$$
\begin{aligned}
expr &\Rightarrow expr\ X \\
&\Rightarrow a\ X \\
&\Rightarrow a + expr\ X \\
&\Rightarrow a + b\ X \\
&\Rightarrow a + b + expr\ X \\
&\Rightarrow a + b + c\ X \\
&\Rightarrow a + b + c\ \epsilon
\end{aligned}
$$

可以看出，分析器能够选择左结合的解释进行这种推导，当然，对于右结合的操作符也可以选择其他解释。

类似符“*”也存在同样的问题，解决方法与上面一样。另外，从一般意义上讲，“*”比起“+”具有更高的优先级，因而，文法的描述应有助于对类似 a+b*c 这样的句子做出正确的解释。然而，上面修改的文法却不能正确处理这类优先级问题。下面是对 a+b*c 的推导：

```
expr  ⟹  expr X
      ⟹  a X
      ⟹  a + expr Y
      ⟹  a + b Y
      ⟹  a + b * expr Y
      ⟹  a + b * c Y
      ⟹  a + b * c ϵ
```

在第4步中，推导产生的表达式被解释成了(a+b)*c，而不是a+(b*c)。

一种解决方法是，引入一个单独的非终结符用于推导含*的句子，并将这个非终结符设为"+"的操作数，这样就能体现"*"具有更高优先级。

```
expr: term X
term: ID Y
X: ϵ | + term X
Y: ϵ | * ID Y
```

用新的文法对a+b*c再次进行最左推导就变为：

```
expr  ⟹  term X
      ⟹  a Y X
      ⟹  a ϵ X
      ⟹  a ϵ + term X
      ⟹  a ϵ + b Y X
      ⟹  a ϵ + b * c Y X
      ⟹  a ϵ + b * c ϵ X
      ⟹  a ϵ + b * c ϵ ϵ
```

term推出一个包含b*c的句子，b*c在这里解释为加运算的右操作数。在第8章中我们将详细讨论这种方法，它能普遍运用于处理任意多层的优先级以及解决操作符的左右结合等各种问题。

上述文法的构造过程通常被省略，而直接写出相应的EBNF文法。例如，可用1.6节的表达式文法完成上面的文法功能。

其他的二义性问题也能通过重写文法解决，但还有一种专门的解决方法，它更为简单和直接，即选择唯一一种解释作为正确的解释，而将其他的解释视为错误的。例如if语句中悬挂else的二义性问题：

```
stmt:
  if ( expr ) stmt
  if ( expr ) stmt else stmt
```

嵌套if语句有两种推导：一是if语句中的else与最外层的if语句对应，另一种是其中的else与最内层的if对应。我们通常采用的是后一种解释方法。在第10章中，我们一遇到else就对它进行分析，以此实现后一种解释。

7.3 自上而下的语法分析

在一种语言中，文法定义了句子的生成规则，这些规则也可以用于识别句子。从前面的内容可知，一个语法分析器就是一个程序，它通过重建句子的推导过程来识别和理解给定语言的句子。在识别过程中，语法分析器重建句子的分析树，这棵树反映了句子的整个推导过程。当然，大多数语法分析器并不真正画出这样一棵树，而是构造与树等价的内部描述，或者仅当节点生成时执行某种语义处理(semantic processing)动作。

实际的语法分析器都是从左至右读取输入符号。但不同类型的语法分析器构造分析树的方式不同。自上而下的语法分析器是以开始非终结符为入口，不断猜测下一步推导，最终实现整个句子的最左推导。推导过程中，输入的下一个单词帮助选取适当的产生式完成下一步推导。例如，下面的文法定义了一种简单的语言 {cabd，cad}：

S: c A d

A: a b
   a

假定一个分析器分析输入 cad。首先输入的 c 表明应选择 S 的第一个产生式（唯一可选的产生式）。因此第一步推导是：

$$S \Rightarrow cAd$$

c 匹配后接着输入下一个符号，这时分析器必须使用 A 的产生式。由于下一个单词是 a，因而可以选择 A 的第一个产生式，推导过程为：

$$\begin{aligned} S &\Rightarrow cAd \\ &\Rightarrow cabd \end{aligned}$$

A 的第一个产生式中的符号 a 被匹配后，输入转入下一个单词。这时分析程序将停止工作，输入符 d 不能与当前推导使用的产生式的下一个符号 b 继续匹配。问题出在我们先前选错了 A 的产生式。分析器只能回退到上一步，并且输入也回退错误推导中读入的输入符号，重新选择 A 的另一个产生式：

$$\begin{aligned} S &\Rightarrow cAd \\ &\rightarrow cad \end{aligned}$$

输入回退到 a，与这次推导中剩余的符号继续匹配，后面的 d 亦获匹配，此时分析同告成功。

上述简单的例子说明，自上而下的语法分析利用输入单词选取适当的产生式，只要输入单词与推导中的终结符匹配，相应的输入单词就算处理完成。若在某步推导的右部遇到非终结符，程序自动选择这个非终结符的产生式作为下一步推导。这个例子也指出了自上而下分析的一个缺陷，错误使用产生式将导致不得不回溯前面的推导。对较复杂的语言来说，这种回溯将导致推翻前面已做的许多工作。尤为重要的是，大多数推导产生的负面影响是很难消除的。例如，需要随时备份输入数据、撤销对符号表的插入等。同时，回溯还使得分析过程变得非常缓慢。最坏的情况是，运行时间可能以输入单词数的指数倍增长。

自上而下的语法分析方法仅在能完全避免回溯的情况下才具有实际价值。这个限制条件使得它局限于处理特定的语言，即仅通过观察输入的下一个单词，就能选择一个正确的产生式完成下一个推导。幸运的是，包括 C 在内的许多程序语言都能满足这样的条件。

实现自上而下的语法分析的一般方法是，对文法中的每个非终结符写出相应的语法分析函数（parsing function），当要用到某个非终结符的产生式时，就调用相关的函数。由于这些非终结符之间可能递归引用，因而分析函数之间也必然会有递归调用。例如，对下面形式的推导：

$$A \rightarrow \ldots \rightarrow \alpha A \beta \Rightarrow \ldots$$

α、β 是文法符号串。使用这种策略编写的自上而下的分析器通常称为递归下降分析器。这种分析过程就像一棵向下生长的分析树，在树的每个节点处调用递归函数。

推导过程无须显式构造，处理递归函数调用的调用栈隐含记录了推导状态。实际上，每个非终结符的函数就是对非终结符的每个候选式的编码，组成一个比较和调用的序列。在产生式中，

终结符处理成与当前的输入单词比较，非终结符处理成调用相关的函数。例如，假设函数 gettok 返回上述语言的单词，产生式 S:cAd 对应的函数如下：

```
int S(void) {
   if (t == 'c') {
      t = gettok();
      if (A() == 0)
         return 1;
      if (t == 'd') {
         t = gettok();
         return 1;
      } else
         return 0;
   } else
      return 0;
}
```

如果函数 A 和 S 能识别从 A 和 S 导出的句子，则 A 和 S 返回 1，否则返回 0。分析开始时，主程序 main 先调用 gettok 进行初始化，并得到第一个输入单词，然后调用函数 S：

```
int t;
void main(void) {
   t = gettok();
   if (S() == 0)
      error("syntax error\n");
   if (t != EOI)
      error("syntax error\n");
}
```

EOI 代表输入结束的单词编码。只有当所有的输入组成一个该语言的句子时，才能说输入是合法的。

7.4 FIRST 和 FOLLOW 集合

为了给文法的每个非终结符编写语法分析函数，必须要根据输入的下一个待处理的单词选择适当的产生式。假设 α 是文法的符号串，FIRST(α) 是这样一个集合：它的元素是从 α 导出的所有句子的起始终结符。FIRST 集合能够帮助我们在推导中选取适当的产生式。

假定文法包括产生式 A:a 和 A:β。下一步推导是用 A 的某个产生式的右部替换 A，这时分析程序将调用 A 的分析函数选择适当的产生式。如果下一个单词属于 FIRST(α)，则选取产生式 A:α；反之，若下一个输入单词属于 FIRST(β)，则选取产生式 A:β。如果下一个输入单词不属于 FIRST(α) ∪ FIRST(β)，则表示有语法错。显然，FIRST(α) 和 FIRST(β) 不能相交。

当 α 只是一个非终结符时，FIRST(α) 就是指能由这个非终结符推导出的所有句子的起始终结符组成的集合。给定一个文法，其中所有的文法符号 X 的 FIRST 集合都能通过检查产生式获得。这种检查往往要迭代多次，直到所有的 FIRST 不再加入新元素时才停止。

如果 X 表示终结符 a，FIRST(X) 就是 {a}。如果 X 表示非终结符，而且有产生式 X:aα（a 是终结符），则 a 应加入 FIRST(X)。如果有产生式 X:[α] 或者 X:{α}，则 FIRST(α) 应加进 FIRST(X)，同时因为 ε 产生式能推导出空串，ε 也要加进 FIRST(X)。如果有下列形式的产生式：

$$
\begin{aligned}
X:\ & \alpha_1 \\
& \alpha_2 \\
& \vdots \\
& \alpha_k
\end{aligned}
$$

则

$$FIRST(\alpha_1) \cup FIRST(\alpha_2) \cup \cdots \cup FIRST(\alpha_k)$$

都应加进 FIRST(X)。如果产生式形如 $X:Y_1Y_2\cdots Y_k$，其中 Y_i 是文法符号，则 $FIRST(Y_1Y_2\cdots Y_k)$ 应加进 FIRST(X)。

$FIRST(Y_1Y_2\cdots Y_k)$ 的内容由 Y_1 到 Y_k 的 FIRST 集合确定。其初始状态为空。将 $FIRST(Y_1)$ 中除 ε 外的所有元素加进 $FIRST(Y_1Y_2\cdots Y_k)$。如果 $FIRST(Y_1)$ 包括 ε，则将 $FIRST(Y_2)$ 中除 ε 外的所有元素全加进 $FIRST(Y_1Y_2\cdots Y_k)$，如此重复下去。若 $FIRST(Y_{i-1})$ 包含 ε，就将 $FIRST(Y_i)$ 中除 ε 外的所有元素加进 $FIRST(Y_1Y_2\cdots Y_k)$。最后的结果是，$FIRST(Y_1Y_2\cdots Y_k)$ 包括了所有具有 ε- 透明的 FIRST 集合元素（ε- 透明指包含 ε）。如果从 $FIRST(Y_1)$ 到 $FIRST(Y_k)$ 都含有 ε，则 ε 也应加进 $FIRST(Y_1Y_2\cdots Y_k)$。

考虑 1.6 节给出的简单表达式的文法：

```
expr:
  term { + term }
  term { - term }
term:
  factor { * factor }
  factor { / factor }
factor:
  ID
  ID '(' expr { , expr } ')'
  '(' expr ')'
```

文法中，每一行表示非终结符的一个候选式。FIRST(expr) 等于：

$$FIRST(term\ \{\ +\ term\ \}) \cup FIRST(term\ \{\ -\ term\ \})$$

必须先计算 FIRST(term) 的值，才能得出 FIRST(expr)。同样，FIRST(term) 等于：

$$FIRST(factor\ \{\ *\ factor\ \}) \cup FIRST(factor\ \{\ /\ factor\ \})$$

它也必须等 FIRST(factor) 计算出来才能得到。这里 FIRST(factor) 容易计算，因为 factor 的产生式都以终结符为起始字符：

$$
\begin{aligned}
FIRST(factor) &= FIRST(\texttt{ID}) \cup FIRST(\texttt{ID '('}\ expr\ \{\ ,\ expr\ \}\ \texttt{')'}) \\
&\quad \cup FIRST(\texttt{'('}\ expr\ \texttt{')'}) \\
&= \{\texttt{ID (}\}
\end{aligned}
$$

现在可以计算出 FIRST(term)，FIRST(term) 为 {ID (}，因此，FIRST(expr) 也为 {ID (}。

有时仅知道 FIRST 集合还不足以确定选取哪个产生式。假设文法：

$$
\begin{aligned}
X:\ & A\ B \\
& C
\end{aligned}
$$

通常情况下，我们根据下一个输入单词是属于 FIRST(AB) 还是 FIRST(C) 来选取相应的产生式。现在如果 FIRST(AB) 包含 ε，即 AB 能推导出空句，这时通过这两个 FIRST 集合不能确定合适的产生式，还必须考虑能够跟在 X 后面的单词，这些单词组成 X 的 FOLLOW 集合。换言之，FOLLOW(X) 是一个终结符组成的集合，它的元素是所有含 X 的句型中紧跟在 X 后面的终结符。FOLLOW 集合为文法的非终结符提供了“右边的上下文”，可用于错误检测和文法构造，所以它适用于递归下降分析。在本例中，如果下一个输入单词属于 FIRST(AB) ∪ FOLLOW(X) 则选取 X:AB 做推导；若属于 FIRST(C) 则选取 X:C。当然，FIRST(AB) ∪ FOLLOW(X) 与 FIRST(C) 是不相交的。

FOLLOW 集合比 FIRST 集合更难计算。主要是因为它要扫描所有的含非终结符的产生式，而不只是考虑定义该非终结符的产生式。比如说对于产生式 X:αYβ，FIRST(β)−{ε} 应加进 FOLLOW(Y)。如果 FIRST(β) 为 ε 透明（即它包含 ε）则 FOLLOW(X) 要加进 FOLLOW(Y)。对于形如 X:αY 的产生式，FOLLOW(X) 应加进 FOLLOW(Y)。与 FIRST 集合的计算相同，FOLLOW 集合的生成也要反复扫描产生式，直至所有集合的元素不再增加。另外，要将文件结束符“⊣”加进开始非终结符的 FOLLOW 集合。

下面将说明如何计算 1.6 节表达式文法的 FOLLOW 集合。由于 expr 是开始符号，因此 FOLLOW(expr) 包括“⊣”。expr 仅出现在 factor 的产生式中，因而：

$$\begin{aligned} FOLLOW(expr) &= \{\dashv\} \cup FIRST(\{\ ,\ expr\ \}\ ')') \cup FIRST(')') \\ &= \{,\ )\ \dashv\} \end{aligned}$$

FIRST(')') 将“)”加入 FOLLOW(expr)。但是 FIRST({,expr}) 包含 ε，因此 FIRST({,expr}')') 将“)”加入 FOLLOW(expr)。

term 在两个 expr 的产生式中各出现两次，所以：

$$\begin{aligned} FOLLOW(term) &= FOLLOW(expr) \\ &\quad \cup FIRST(\{\ +\ term\ \}) \cup FIRST(\{\ -\ term\ \}) \\ &= \{,\ )\ \dashv\ +\ -\} \end{aligned}$$

同样，factor 在每个 term 产生式中出现两次。

$$\begin{aligned} FOLLOW(factor) &= FOLLOW(term) \\ &\quad \cup FIRST(\{\ *\ factor\ \}) \cup FIRST(\{\ /\ factor\ \}) \\ &= \{,\ )\ \dashv\ +\ -\ *\ /\} \end{aligned}$$

7.5 编写分析函数

编译器使用 EBNF 文法描述语言，为每个非终结符计算 FIRST 集合和 FOLLOW 集合，编写分析函数将非终结符的每个产生式翻译成可执行代码。它的基本思想是为每个非终结符 X 构造一个函数 X，按照 X 的产生式规则编写相应的代码。

翻译的规则可由文法产生式的可能形式导出。对每一种产生式形式 α，用 T(α) 表示 α 的翻译代码。在分析过程中的任意一点，全局变量 t 表示从词法分析器读入的当前单词，调用函数 gettok 可以获取下一个输入单词。

假设有产生式 X:α，它的翻译函数 X 为

```
X() { T(α) }
```

表 7-1 的左栏给出了每一种 α 的产生式形式，右栏给出了每种产生式对应的代码，即 T(α)。

$$D(\alpha)=\begin{cases}(FIRST(\alpha)-\{\epsilon\})\cup FOLLOW(X), & \text{如果}\epsilon\in FIRST(\alpha)\\ FIRST(\alpha), & \text{其他}\end{cases}$$

表 7-1 分析函数翻译

α	$T(\alpha)$
终结符 A	`if (t == A) t = gettok();` `else error`
非终结符 X	`X();`

$\alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$	`if (t` $\in D(\alpha_1)$`) T(`α_1`)`
	`else if (t` $\in D(\alpha_2)$`) T(`α_2`)`
	`...`
	`else if (t` $\in D(\alpha_k)$`) T(`α_k`)`
	`else error`

$\alpha_1\alpha_2\cdots\alpha_k$	$T(\alpha_1)\ T(\alpha_2)\ \cdots\ T(\alpha_k)$
[α]	`if (t` $\in D(\alpha)$`) T(`α`)`
{α}	`while (t` $\in D(\alpha)$`) T(`α`)`

当然，表 7-1 的代码也可用其他代码序列表示，例如 $T(\alpha_1|\alpha_2|\cdots|\alpha_k)$ 可以用 switch 语句表示。相反，机械地套用表 7-1 的代码有时会造成代码冗长。采取一些简单的转换可改进这个问题。比如用表 7-1 的规则导出产生式

parameter-list: [ID { , ID }]

的分析函数可分成以下 7 步：

1. *T*(*parameter-list*)
2. *T*([ID { , ID }])
3. `if (t == ID) { T(ID { , ID }) }`
4.
```
if (t == ID) {
    if (t == ID) t = gettok();
    else error("missing identifier\n");
    T([ , ID ])
}
```
5.
```
if (t == ID) {
    if (t == ID) t = gettok();
    else error("missing identifier\n");
    while (t == ',') { T(, ID) }
}
```
6.
```
if (t == ID) {
    if (t == ID) t = gettok();
    else error("missing identifier\n");
    while (t == ',') {
       if (t == ',') t = gettok();
       else error("missing ,\n");
       T(ID)
    }
}
```

```
7. if (t == ID) {
       if (t == ID) t = gettok();
       else error("missing identifier\n");
       while (t == ',') {
          if (t == ',') t = gettok();
          else error("missing ,\n");
          if (t == ID)  t = gettok();
          else error("missing identifier\n");
       }
   }
```

步骤 4 中的第二个 if 语句测试 t==ID 是多余的。如果控制执行到此 if 语句，则这个测试条件一定为真。类似地，步骤 6 中 while 循环体内的 if 语句（测试是否为逗号）也没必要。综上所述，函数可做如下简化：

```
void parameter_list(void) {
   if (t == ID) {
      t = gettok();
      while (t == ',') {
         t = gettok();
         if (t == ID) t = gettok();
         else error("missing identifier\n");
      }
   }
}
```

编写分析程序时，我们通常用提取左公因子（left factoring）的方法避免重写文法和增加新的非终结符。比如，A:αβ|αγ 即为 A:α(β|γ)，故 T(αβ|αγ) 可写成：

$$T(\alpha)\ T(\beta \mid \gamma)$$

在某些情况下，α 作为几个产生式的公共前缀出现，会涉及重要的语义处理。这时，我们通常引入一个新的非终结符，然后对相关的产生式提取左公因子，从而把 α 的语义处理单独封装为一个分析函数。

7.6 处理语法错误

FIRST 和 FOLLOW 集合及相关的子集不仅可以指导分析决策，而且能够用来检查错误。程序中经常存在两类错误：语法错误（syntax error）和语义错误（semantic error）。前者指输入不是语言的合法句子，后者指输入虽然合法，但无意义。例如，表达式 x+6，从语法角度分析是正确的，但若 x 没有提前声明，则表达式在语义上是错误的。

语义错误由分析函数根据指定结构的语义进行检查和处理。在分析函数的实现过程中会描述这些错误。

语法错误可以采用系统的方式解决，不依赖于出现的上下文。这些错误较易发现，通常出现在表 7-1 所示的翻译代码 error 中。但要修正错误并非易事。当然，不能说一发现错误就停止分析，这样做极不合理。错误处理的大部分工作用于进行错误恢复，使分析工作得以继续。

出现语法错误意味着出现的句子不属于给定的语言。语法错误的恢复方法是通过对输入添加遗漏的单词或者忽略某些单词将错误的输入转换为合法的句子。遗憾的是，选择合适的恢复方法非常困难。如果选择错误将导致分析程序乱套，甚至后面在文法上正确的输入也将产生大量语法错误。更糟的是，恢复某些小错误可能导致当前分析处理输入的中断。

递归下降分析器的结构有助于选取合适的错误恢复策略。分析器由许多函数组成，每个函数只完成整个分析任务的一小部分。因此，目标被划分成若干子目标，每个子目标调用相关的分析函数实现。为了保证分析的持续性，编写的每个函数都应该保下一个输入单词能够合法地跟在某个句型的非终结符之后。这样，如果检测到了错误，相应的分析函数就会马上报错，跳过非法输入单词，直至遇到能跟在给定非终结符后的单词才继续往下分析。

具体而言，假设构造了一个非终结符 X 的分析函数 X，如果下一个输入单词不属于非终结符 X 的 FOLLOW 集合，函数就略过它，反复执行直至遇到 X 的 FOLLOW 集合中的单词。这种思想的初衷是为了使分析程序与输入保持同步。在处理完属于 FOLLOW(X) 的单词后，要将所有合法的部分返回给函数 X 的调用者。但这种方法也存在局限性，它不能处理含有非终结符 X 的句型。例如，非终结符 X 出现在句型 αXβ 中。函数 X 将会略过 D(β) 中的元素。由于 D(β) 通常比 FOLLOW(X) 小，当分析误判出一个属于 FOLLOW(X)，但不属于 D(β) 的输入单词时，程序丢弃该单词，X 停止继续执行。这种判断显然为之过早。此外，函数 X 的调用程序还会报出其他不必要的语法报错。可见，如果 D(β) 已知，函数必须使用 D(β)，否则使用 FOLLOW(X)。比如，调用表达式的分析函数 expr() 分析 for 语句的第三个表达式时，函数就利用了集合 {;,)} 恢复表达式存在的语法错误。

下面 error.c 中的输出函数采用了上述策略：

```
⟨error.c exported functions⟩≡                                    107
  extern void test ARGS((int tok, char set[]));
```

该函数检查下一个单词是否等于 tok。如果不等，则发出提示信息，并跳过当前单词，反复执行直至遇到一个属于 {tok} ∪ set 的单词。set 集合包含了所有不能忽略的元素，保证输入不会无限地忽略。set 只是一个以空指针为结束标记的单词编码数组。

```
⟨error.c functions⟩≡                                            108
  void test(tok, set) int tok; char set[]; {
     if (t == tok)
        t = gettok();
     else {
        expect(tok);
        skipto(tok, set);
        if (t == tok)
           t = gettok();
     }
  }
```

函数 test 调用函数 expect 报错，调用函数 skipto 跳过错误的单词。expect 和 skipto 的代码见下文。

如果编译器发现输入错误，而采取跳过当前输入的方法是合适的对策时，test 采用的策略将能良好地工作。但是，如果遇到一个预期的单词在输入中被漏掉，则 test 无法做出正确的处理。这时，可以采用另一种更为有效的方法进行报错和处理，即假定预期的单词存在，然后继续分析。这种方法能够有效插入遗漏的单词，由于所遗漏的通常是具有简单句法功能的单词，例如分号或者逗号，因此这种方法能够很好地工作。下面提供了其余的实现代码：

```
⟨error.c exported functions⟩+≡                                  107 109
  extern void expect ARGS((int tok));
```

该函数检查下一个单词，即 t 的当前值是否与 tok 相等，如果是，则转入下一个输入单词。

```
⟨error.c functions⟩+≡                                    107 108
  void expect(tok) int tok; {
     if (t == tok)
        t = gettok();
     else {
        error("syntax error; found");
        printtoken();
        fprint(2, " expecting '%k'\n", tok);
     }
  }
```

当 expect 被函数 test 调用时，第一个测试条件肯定不满足，调用的目的只是为了进行错误诊断和报错。当需要一个特定的预期单词时，其他的分析函数也会调用 expect，expect 处理该预期的单词。如果预期的单词被漏掉了，expect 进行诊断报错，然后返回，输入停在当前位置。这样做的效果如同实际存在这样一个预期的标识符。

expect 调用函数 error 开始显示错误信息，接着调用静态函数 printtoken 显示当前输入单词（即由 t 给定的单词和 token），最后调用 fprint 显示错误信息的最后结论。例如，调用 expect 处理输入串 "int x[5;" 显示的诊断信息为：

```
syntax error; found ';' expecting ']'
```

从 expect 的代码可以知道，函数 error 初始化错误信息，error 的输入参数是一个 printf 风格的格式串和其他参数。除了输入的信息外，error 还可以打印由 gettok 得到的当前单词在源程序中的位置，并将错误信息的总数保存在变量 errcnt 中。

```
⟨error.c functions⟩+≡                                    108 110
  void error VARARGS((char *fmt, ...),
  (fmt, va_alist),char *fmt; va_dcl) {
     va_list ap;

     if (errcnt++ >= errlimit) {
        errcnt = -1;
        error("too many errors\n");
        exit(1);
     }
     va_init(ap, fmt);
     if (firstfile != file && firstfile && *firstfile)
        fprint(2, "%s: ", firstfile);
     fprint(2, "%w: ", &src);
     vfprint(2, fmt, ap);
     va_end(ap);
  }
⟨error.c data⟩=                                          109
  int errcnt   = 0;
  int errlimit = 20;
```

如果 errcnt 的值太大，error 会终止程序的执行。函数 warning 与 error 很相似，但显示的是警告信息，不会增大 errcnt 的值。函数 fatal 也与 error 类似，但在显示错误信息后会立刻停止编译，因此只有当编译程序本身有错时才调用 fatal。

最后一个与出错处理相关的函数是：

```
⟨error.c exported functions⟩+≡                                  107
  extern void skipto ARGS((int tok, char set[]));
```

skipto 不断跳过输入单词，直至遇到单词 t（t 要么与 tok 相等，要么使得 kind[t] 包含在无效终结符数组 set 中）。数组 kind 的定义如下：

```
⟨error.c exported data⟩≡
  extern char kind[];
```

kind 以单词编码为下标，把单词编码划分为不同的集合。数组定义时包括 token.h，并且抽取 token.h 的第 6 栏。token.h 见 6.2 节：

```
⟨error.c data⟩+≡                                                108
  char kind[] = {
  #define xx(a,b,c,d,e,f,g) f,
  #define yy(a,b,c,d,e,f,g) f,
  #include "token.h"
  };
```

kind[t] 是一个单词编码，它意味着一个含有 t 的集合。例如编码 ID 表示集合 FIRST(expression)（expression 的定义见 8.3 节）。因而，对每个单词 t ∈ FIRST(expression)，kind[t] 与 ID 相等。语句 kind[t] == ID 测试 t 是否属于 FIRST(expression)。利用数组 {ID, 0} 作为函数 skipto 的第二个参数，指示函数跳过若干个无关的单词直至找到一个属于 FIRST(expression) 的元素。

下面概括了所有的 kind 值。左边的单词编码代表包含该单词编码的集合，右边列出了集合中的单词：

```
ID      FCON ICON SCON SIZEOF & ++ -- * + - ~ ( !
CHAR    FLOAT DOUBLE SHORT INT UNSIGNED SIGNED
        VOID STRUCT UNION ENUM LONG CONST VOLATILE
STATIC  EXTERN AUTO REGISTER TYPEDEF
IF      BREAK CASE CONTINUE DEFAULT DO ELSE
        FOR GOTO RETURN SWITCH WHILE {
```

对于上面未提及的单词，kind[t] 就等于 t，例如 kind[';'] 等于 ';'。由 kind 定义的集合与 7.4 节描述的 FIRST 集合有如下关系：

```
kind[ID]                    =  FIRST(expression)
kind[ID] ∪ kind[IF]         ⊃  FIRST(statement)
kind[CHAR] ∪ kind[STATIC]   ⊂  FIRST(declaration)
kind[STATIC]                ⊂  FIRST(parameter)
```

上面列出的非终结符分别在第 8、10 和 11 章中定义。

因为 skipto 的第二个参数是数组，如果一些额外单词的 kind 值与自身相等，例如前面提到的 ';'，那么这个数组可以表示集合的超集。在一些情况下，这些超集与 FOLLOW 集合相关。例如，某个 statement 后面必须紧跟一个 } 或 FIRST(statement) 的单词，statement 的分析函数将包含 IF、ID 和 } 的数组作为参数传递给 skipto。

当 skipto 跳过单词时，通知前 8 个和最后一个跳过的单词：

```
(error.c functions)+≡                                    108
  void skipto(tok, set) int tok; char set[]; {
     int n;
     char *s;

     for (n = 0; t != EOI && t != tok; t = gettok()) {
        for (s = set; *s && kind[t] != *s; s++)
           ;
        if (kind[t] == *s)
           break;
        if (n++ == 0)
           error("skipping");
        if (n <= 8)
           printtoken();
        else if (n == 9)
           fprint(2, " ...");
     }
     if (n > 8) {
        fprint(2, " up to");
        printtoken();
     }
     if (n > 0)
        fprint(2, "\n");
  }
```

如果 t 与 tok 相等，或者 t 属于 kind[t]，skipto 不跳过任何单词，也不会发出出错信息。假设 bug.c 只有一行：

```
fprint(2, " expecting '%k'\n", tok);
```

它的语法错误在于这一行代码应该属于某个函数。虽然对 fprint 的调用看上去是一个函数定义的开头，但 lcc 能很快发现这个错误，函数 test 调用 expect 和 skipto 显示下面的信息：

```
bug.c:1: syntax error; found '2' expecting ')'
bug.c:1: skipping '2' ',' '"' expecting '%k'\12" ',' 'tok'
```

值得注意的是右括号没有被跳过。

深入阅读

许多书都讲述了构造一个编译器的理论和实践方法，例如 Aho，Sethi and Ullman（1986）、Fischer and LeBlanc（1991）、Waite and Goos（1984）、Davie and Morrison（1981）以及 Wirth（1976）阐述了递归下降编译器的设计与实现。

自底向上的分析器重新构建输入串的最右推导，按照从叶节点到根节点的顺序建立分析树。这种分析器能处理多种类型的语言，而且文法编写也容易，因而普遍应用于各种编译器。大多数自底向上的分析器都采用 LR 的分析策略，或是它的各种变形。LR 分析法最初由 Aho and Johnson（1974）发表，Aho，Sethi and Ullman（1986）对这种分析法做了详细介绍。以后相继出现了许多种分析器产生器，它们也能处理语言的句法规则（这种规则的形式大多同练习 7.2 的规则相似），并能自动生成分析程序。YACC（Johnson，1975）就是一个用于 UNIX 平台的分析器产生器。YACC 和 LEX 同时工作时能大大简化编译器的实现。Aho，Sethi and Ullman（1986），Kernighan and Pike（1984）以及 Schreiner and Friedman（1985）讲述了 YACC 和 LEX 的许多实例。

Holub（1990）也介绍了另一个分析器产生器的实现。

其他的分析器产生器都是基于属性文法的，Waite and Goos（1984）介绍了属性文法的概念以及相关的分析器产生器。

lcc 的出错处理机制与 Stirling（1985）中提及的方法相似，Wirth（1976）应用了这种方法。Burke and Fisher（1987）还提出了一种新的方法，它或许是处理 LR 和 LL 分析表错误的最佳方法。

练习

7.1 使用前几章的词法分析器和符号表模块搭建一个简单的分析器，使之能识别由下面文法定义的表达式，并画出分析树。

```
expr:
   term { + term }
   term { - term }

term:
   factor { * factor }
   factor { / factor }

factor:
   ID
   ID '(' expr { , expr } ')'
   '(' expr ')'
```

7.2 编写一个程序，使之能计算 EBNF 文法的 FOLLOW 和 FIRST 集合，并报告哪些因素影响递归下降分析。针对与 EBNF 相似的文法设计一种输入表示方法。例如，假设给定一个自由格式的文法，文法中非终结符用插入 "-" 符号的小写字母表示，终结符用大写字母表示，或者用单引号或双引号括起来，产生式都以分号结尾。例如，练习 7.1 中的文法可表示为：

```
expr    : term { ( '+' | '-' ) term } ;

term    : factor { ( '*' | '/' ) term } ;

factor  : ID [ '(' expr { , expr } ')' ]
        | '(' expr ')'
        ;
```

针对输入的语法，给出一个 EBNF 说明，采用本章描述的技术编写递归下降分析器来识别输入。

表 达 式

C 的表达式构成了一种子语言（sublanguage），可以直接根据这种子语言写出表达式的分析函数。这一特点使得我们从表达式出发，能够更好地描述 lcc 中 8 个分析和处理输入的源程序的模块。这些函数将建立源程序的内部表示，例如抽象语法树和 1.3 节描述的代码表。

用于语法分析和描述表达式的模块有 4 个：expr.c 实现语法分析函数，识别和翻译表达式；tree.c 实现管理分析树的基本函数，分析树是表达式的内部中间表示；enode.c 实现类型检查（type-checking）函数，保证表达式语义的有效性，输出建立和操作分析树的函数；simp.c 实现与树转换相关的函数，例如常量折叠（constant folding）等。

广泛地说，本章的重点是解释 tree.c 和 expr.c 两个模块，描述用于表示表达式的抽象语法树的形式。在大部分介绍中，我们采用了自顶向下的方式浏览语法分析函数，了解它们如何建立分析树。第 9 章将具体结合 C 语言的语义规则讨论分析树表示的意义，并采用与建立类型检查树过程一致的自底向上的方式浏览语义函数。本章最后一节则是一个例外，它既描述了抽象语法树中表示常量和标识符的叶节点的各种形式，又介绍了如何处理这些叶节点所表示的意义。

8.1 表达式的表示

编译器不仅具有识别和分析表达式的能力，还应该能建立表达式的中间表示，从而验证表达式的有效性，生成目标代码。我们通常采用抽象语法树（或其他简单的树结构）作为表达式的中间表示。抽象语法树是一类特殊的分析树，没有表示非终结符的节点，也没有表示无用的终结符的节点，其节点代表操作符，子节点表示操作数。例如，表达式 (a-b)-b*(a+b) 的抽象语法树表示如图 8-1 所示，图中的节点不是语法分析的非终结符，也不表示单词“(”和“)”。由于图中的操作符（例如 ADD+I 和 MUL+I）包含了相关的信息，因此图中也没有节点直接表示单词 + 和 *。ADDRG+P 节点表示计算其操作数指定的标识符的地址。

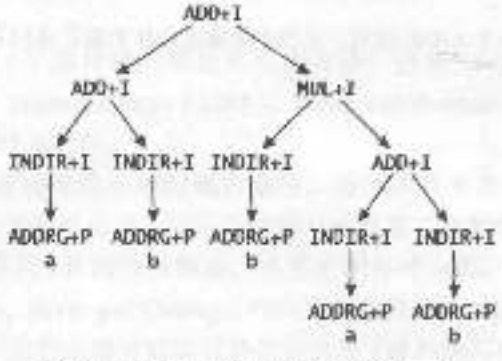


图 8-1 (a+b)-b*(a-b) 的抽象语法树

抽象语法树中的操作符通常不出现在源语言中。例如，节点 INDIR+I 表示按操作数指定的地址取整数，但在 C 语言中，没有显式的“取”操作符。另外还有由于隐式转换引入的转换操作符

和作为语义规则的结果引入的一些操作符。某些这样的操作符并不会产生对应的运行指令，引入它们是为了便于编译。

与类型表示一样，抽象语法树也能写成以括号开头的形式。比如表达式 (a+b)+b*(a+b) 的抽象语法树（见图 8-1）可以用下面的式子表示：

```
(ADD+I
   (ADD+I (INDIR+I (ADDRG+P a)) (INDIR+I (ADDRG+P b)))
   (MUL+I
      (INDIR+I (ADDRG+P b))
      (ADD+I (INDIR+I (ADDRG+P a)) (INDIR+I (ADDRG+P b)))
   )
)
```

分析表达式的过程也是生成抽象语法树的过程。对树节点，我们给出如下定义：

```
⟨tree.c typedefs⟩≡
  typedef struct tree *Tree;

⟨tree.c exported types⟩≡
  struct tree {
     int op;
     Type type;
     Tree kids[2];
     Node node;
     union {
        ⟨u fields for Tree variants 128⟩
     } u;
  };
```

op 域表示操作符的代码，type 域指向 Type，表示运行时该节点计算结果的类型，kids 域表示操作数，node 域用于建立与树对应的 dag（无环有向图）（详见 12.2 节）。某些操作符对应的树含有附加信息，这些信息都放在联合 u 中。

抽象语法树的各类操作符形成了第 5 章定义的节点操作符（见表 5-1）的超集，这些操作符用于树时，它们的写法与前面不同。表中的通用操作符加上表示类型的后缀就形成了分析树中的操作符。例如，ADD+I 代表整数加法。省略了 +，就是节点操作符 ADDI。这种约定使我们能在图和上下文中很好地分清分析树和 dag。类型后缀表见 5.5 节，表 5-1 给出了每个操作符可搭配的后缀形式。

表 8-1 列出了除表 5-1 以外抽象语法树中可以出现的其他 6 类操作符。AND、OR 和 NOT 分别代表 &&、|| 和 ! 操作。逗号表达式生成 RIGHT 树，通过定义可知，RIGHT 从左至右计算其参数的值，RIGHT 的结果就是其最右边操作数的值。RIGHT 也能生成逻辑上具有两个以上操作数的树，例如 COND 操作符。COND 能表示形如 $c?e_1:e_2$ 的条件表达式，COND 树的第一个操作数是 c，第二个操作数是一棵含 e_1 和 e_2 的 RIGHT 树。RIGHT 树也可表示 e++ 这样的表达式。这些操作符仅在编译器前端使用，因而不需要也不能带类型后缀。FIELD 操作符表示对位域的引用。

表 8-1 树操作符

syms	kids	操作符	操作
	2	AND	逻辑与
	2	OR	逻辑或
	1	NOT	逻辑非
IV	2	COND	条件表达式

（续）

syms	kids	操作符	操作
	1或2	RIGHT	左个
	1	FIELD	结构段访问

抽象语法树与 dag 有很多相同的操作符，但是表 5-1 中总结的关于操作数个数和符号的规则只针对 dag。前端生成树时不受这些规则的限制，并且经常使用 dag 所没有的其他操作数和符号。例如，为函数调用的参数生成树时，使用 ARG 节点中的 kid[1] 域建立参数列表，并将生成的树存入 kid[0]。

函数 tree 实现树的分配、初始化和返回：

```
⟨tree.c functions⟩=                                          114
  Tree tree(op, type, left, right)
  int op; Type type; Tree left, right; {
     Tree p;

     NEW0(p, where);
     p->op = op;
     p->type = type;
     p->kids[0] = left;
     p->kids[1] = right;
     return p;
  }

⟨tree.c data⟩=                                               118
  static int where = STMT;
```

树的空间分配在由变量 where 指定的分配区中，where 总指向 STMT 分配区。每分析完成一条语句，STMT 中就有大量的数据被释放。但是，在某些情况下，即使超出当前语句的编译范围，也必须将整个表达式的树存储在 STMT 中以备后用。比如 for 循环中的自增表达式就是一个例子。这些表达式的分析都可通过调用 texpr 函数实现，调用时带上指向分配区的参数。

```
⟨tree.c functions⟩+=                                         114 118
  Tree texpr(f, tok, a) Tree (*f) ARGS((int)); int tok, a; {
     int save = where;
     Tree p;

     where = a;
     p = (*f)(tok);
     where = save;
     return p;
  }
```

函数 texpr 先保存 where，然后将 where 设置成 a，接着调用分析函数 (*f)(tok)，再恢复 where 先前的值，最后返回树 *f。

tree.c 还包括其他函数，它们用于构建、测试和处理树及操作符。因为对少数操作符而言，前端只生成 dag，并不建立树，因此这些函数不是修改已有的树，而是重新生成新的树。rightkid(p) 返回一组嵌套的 RIGHT 树中最右边的非 RIGTH 操作数。当 p->type==ty 时，retype(p,ty) 返回 p，否则返回 ty 类型的 p 的副本。当 p 包含一个 CALL 树时，hascall(p) 返回 1，否则为 0。

generic(op)返回op的一般操作符；optype(op)返回op的类型后缀；opindex(op)返回op的操作符索引，即将通用操作符映射到一个连续范围内的整数，以标识该操作符。

8.2 表达式分析

C语言共有41种操作符，分成15个优先级。如果采用7.2节建议的EBNF文法，即为每个优先等级引入一个非终结符，然后为每个非终结符编写分析程序，这样做虽然正确，但显得过于繁杂。本节将介绍一种重要的简化方法，它能精简文法以及最后生成的代码。

考虑下面关于7.4节文法的简化表述，它描述了C语言的一小部分表达式：

expr: term { + *term* }

term: factor { * *factor* }

factor: ID | '(' *expr* ')'

可以根据表7-1直接写出该文法的分析函数。例如，我们采取下面的步骤对非终结符expr的分析函数expr（不带语义）进行简化：

```
T(expr)
T(term { + term })
T(term) T({ + term })
term(); T({ + term })
term(); while (t == '+') { T(+ term) }
term(); while (t == '+') { T(+) T(term) }
term(); while (t == '+') { t = gettok(); T(term) }
term(); while (t == '+') { t = gettok(); term(); }
```

同样，非终结符term的分析函数term是：

```
factor(); while (t == '*') { t = gettok(); factor(); }
```

factor是一个基础函数，用来处理基本表达式：

```
void factor(void) {
   if (t == ID)
      t = gettok();
   else if (t == '(') {
      t = gettok();
      expr();
      expect(')');
   } else
      error("unrecognized expression\n");
}
```

这个文法具有两个优先级。一般来说，n个优先级就对应n+1个非终结符，每个非终结符对应一个优先级，再加上一个非终结符对应基本项——文法的最小单位，不能再细分。这样必然要构造n+1个函数与非终结符一一对应。假设二元运算符都采用左结合，这些函数实际上非常相似。比较两个函数expr和term就可以看出，函数之间的主要区别仅在于处理的操作符与调用的子函数不同。假设优先级为k的非终结符对应的分析函数为k，则函数k将调用函数k+1。

利用这种相似性，我们能编写一个单独的函数以及一张按等级递增排列的操作符表，替代前面的函数1到函数n。lcc利用单词编码作为索引，将优先级存储在一个数组中。表8-2列出了C语言所有操作符的优先级和结合顺序。prec[t]代表单词编码为t的操作符的优先级。例如，prec['+']等于12，prec[LEQ]等于10。利用数组prec并假定它只包含-、+、*、/、%，那么上

面给出的 expr 和 term 就能用下面的函数替换：

```
void expr(int k) {
   if (k > 13)
      factor();
   else {
      expr(k + 1);
      while (prec[t] == k) {
         t = gettok();
         expr(k + 1);
      }
   }
}
```

函数中的 13 来自表 8-2：二元运算符 + 和 - 的优先级为 12，*、/ 和 % 的优先级为 13。当 k 超过 13 时，函数 expr 调用 factor 来分析 factor 的产生式。分析受限文法时，首先调用的是 expr(12)，factor 中对 expr 的调用必须改成调用 expr(12)。

表 8-2 操作符优先级、结合顺序和分析函数

优先级	结合顺序	操作符	目的	分析函数
1	左结合	,	组合	expr
2	右结合	= -= += *= /= %= &= ^= |= <<= >>=	赋值	expr1
3	右结合	?:	条件	expr2
4	左结合	||	逻辑或	expr3
5	左结合	&&	逻辑与	expr3
6	左结合	|	位或	expr3
7	左结合	^	位异或	expr3
8	左结合	&	位与	expr3
9	左结合	== !=	等价	expr3
10	左结合	< > <= >=	关系	expr3
11	左结合	<< >>	移位	expr3
12	左结合	+ -	加	expr3
13	左结合	* / %	乘	expr3
14	右结合	* & - + ! ~ ++ -- sizeof 类型转换	一元前缀	unary
15		++ --	一元后缀	postfix

用 expr 和 factor 能处理任何形如 expr: expr⊗expr | factor 的表达式，⊗ 表示任何二元左结合的操作符。如果要在文法中增加一些操作符，只需对 prec 做适当的初始化。

函数 expr 中的 while 循环可处理左结合操作符，而在 EBNF 文法中，这是通过 expr 和 term 的产生式来说明的。右结合的操作符（比如赋值），在 EBNF 中的产生式形式为：

asgn: expr = asgn

前面的方法也可处理这两种语句，但是在 expr 的 while 循环中应该调用 expr(k) 而不是 expr(k+1)。假定每个优先级的所有操作符具有相同的结合属性，那么我们就能以表格的形式确定每个优先级到底是调用 expr(k) 还是 expr(k+1)，接着只要分别写出左结合和右结合操作的分析函数，或者为每个操作符编写一段显式的测试代码就可以了。

一元操作符也适用于这种方法。幸运的是，C 语言中的一元操作符都具有高优先级，因此它们像 factor 一样出现在函数 n+1 中，否则在函数 expr 入口处就必须先对第 k 级的一元操作符进行检查。

使用这种方法，与大多数中间优先级的操作符对应的非终结符都可省略，因此也能简化描述表达式的文法。

8.3 C 语言表达式的分析

下面是有关 C 语言表达式的完整语法：

expression:
  assignment-expression { , *assignment-expression* }

assignment-expression:
  conditional-expression
  unary-expression assign-operator assignment-expression

assign-operator:
  one of = += -= *= /= %= <<= >>= &= ^= |=

conditional-expression:
  binary-expression [? *expression* : *conditional-expression*]

binary-expression:
  unary-expression { *binary-operator unary-expression* }

binary-operator:
  one of || && '|' ^ & == != < > <= >= << >> + - * / %

unary-expression:
  postfix-expression
  unary-operator unary-expression
  '(' *type-name* ')' *unary-expression*
  sizeof *unary-expression*
  sizeof '(' *type-name* ')'

unary-operator:
  one of ++ -- & * + - ~ !

postfix-expression:
  primary-expression { *postfix-operator* }

postfix-operator:
  '[' *expression* ']'
  '(' [*assignment-expression* { , *assignment-expression* }] ')'
  . *identifier*
  -> *identifier*
  ++
  --

primary-expression:
  identifer
  constant
  string-literal
  '(' *expression* ')'

文法中共有7个以-expression结尾的非终结符，例如assignment-expression，每个非终结符对应一个表达式分析函数。其中，binary-expression可以采用8.2节介绍的技术，该技术能够处理任何4到13级之间（见表8-2）的二元操作符。

每个函数对有效的表达式进行分析，生成表达式的语法分析树，并且对树做类型检查，最后返回生成的树。函数中用到3个数组，每个数组以单词编码作为下标索引，指导函数的各种操作。prec[t]给出了单词编码t对应的操作符的优先级（具体参见8.2节）。oper[t]给出了单词编码t对应的通用操作符树。optree[t]指向为t对应的操作符构造分析树的函数。例如，prec['+']等于12，oper['+']是ADD，optree['+']表示函数addtree。这些数组与大多数optree会引用到的函数一样，都包含在enode.c中。prec和oper的定义引入了token.h，它们选用其中的第3栏和第4栏。

```
⟨tree.c data⟩+≡                                          ↑114 129
  static char prec[] = {
  #define xx(a,b,c,d,e,f,g) c,
  #define yy(a,b,c,d,e,f,g) c,
  #include "token.h"
  };
  static int oper[] = {
  #define xx(a,b,c,d,e,f,g) d,
  #define yy(a,b,c,d,e,f,g) d,
  #include "token.h"
  };
```

token.h见6.2节。

每个函数都可用7.5节描述的规则导出。实际上，构造树代码以及对树进行检查的代码交叉出现在分析代码中。非终结符expression的分析代码具有代表性，而且也是最简单的。

```
⟨tree.c functions⟩+≡                                     ↑114 119
  Tree expr(tok) int tok; {
     static char stop[] = { IF, ID, ']', 0 };
     Tree p = expr1(0);

     while (t == ',') {
        Tree q;
        t = gettok();
        q = pointer(expr1(0));
        p = tree(RIGHT, q->type, root(value(p)), q);
     }
     ⟨test for correct termination 119⟩
     return p;
  }
```

expr首先调用expr1，expr1分析assignment-expression，并返回表达式的分析树（具体参见8.4节）。while循环处理assignment-expression产生式的

{ , *assignment-expression* }

部分，为每个逗号操作符生成一棵RIGHT树。函数pointer和value检查作为参数的树的语义正确性，或者返回参数树的变换结果（后面将给出函数的代码）。在练习12.9中函数root可以处理那些只起副作用的树。

当 expr 的参数不为零时，它表示可以跟在当前 expression 之后的单词编码。

```
⟨test for correct termination 119⟩=                    118 120
  if (tok)
     test(tok, stop);
```

如果 tok 不为零，但表达式后跟有其他文法符号，那么函数 test 会跳过若干个输入符号直至下一个 tok 或是遇到 stop 中的单词，集合 stop 为 tok ∪ {IF ID}（参见 7.6 节）。许多分析函数都使用这种约定，能有效地检查和处理错误。每个 expression 后面必须跟着它的 FOLLOW 集合中的单词。但在多数应用中，能够跟在 expression 后面的单词只有一个。例如，for 循环的增量表达式后面只能跟有一个右括号。因此，expr 不是对 FOLLOW 集合中的单词逐个检查，而只是查找集合中的某个元素，这种方法更精确一些。在具体上下文中，如果有几个单词能跟在某个 expression 后面，就调用 expr(0) 检查下一个输入单词的合法性。

语句级表达式，比如赋值和函数调用，会产生副作用。

```
⟨tree.c functions⟩+≡                                  118 119
  Tree expr0(tok) int tok; {
     return root(expr(tok));
  }
```

expr0 调用 expr 分析表达式，将结果分析树作为参数传递给函数 root，root 只返回具有副作用的树。例如，语句 a+f() 中的加法运算是无用的，lcc 消除了这种无用的加法运算（即使加法运算可能导致溢出）。如果将该表达式的分析树传递给 root，则 root 返回 f() 的分析树。函数 root 见练习 12.9。

8.4 赋值表达式

assignment-expression 的第二个候选式中包含右递归，使赋值操作为右结合。比如它将 a=b=c 这样的多重赋值语句解释为 a=(b=c)。而用产生式

assignment-expression:
  unary-expression [*assign-operator conditional-expression*]

则是错误的，因为它将多重赋值表达式解释为左结合，即 (a=b)=c。这种解释导致赋值的结果不是左值。

函数 expr1 分析赋值表达式：

```
⟨tree.c functions⟩+≡                                  119 120
  Tree expr1(tok) int tok; {
     static char stop[] = { IF, ID, 0 };
     Tree p = expr2();

     if (t == '='
     || (prec[t] >=  6 && prec[t] <=  8)
     || (prec[t] >= 11 && prec[t] <= 13)) {
        int op = t;
        t = gettok();
        if (oper[op] == ASGN)
           p = asgntree(ASGN, p, value(expr1(0)));
        else
           ⟨augmented assignment 120⟩
     }
```

```
    ⟨test for correct termination 119⟩
    return p;
}
```

函数 expr2 分析 conditional-expression：

conditional-expression:
binary-expression [? *expression* : *conditional-expression*]

函数 expr1 的代码并不是严格地遵照文法：assignment-expression 的两个候选式都需要调用函数 expr2 处理，即使处理第二个候选式时要调用函数 unary。expr2 最终会调用 unary，上面的分析代码可以识别所有正确的表达式，也能接受不正确的表达式，但是通过函数 asgntree 的语义分析可以找出错误的表达式。这种赋值表达式的分析方法具有强大的错误处理能力。举例来说，a-b=c 中的 a+b 不是 unary-expression，较严格的分析器分析时将在 + 处标记错误，由于不能完整地分析出整个表达式，因而可能还会出现其他的错误标记。虽然 lcc 可以接受该表达式而不报告语法错误，但是如果赋值语句的左部不是一个左值，lcc 会进行警告提示。

函数 expr1 的第一个 if 语句测试赋值号（=）或者扩展赋值运算符的首字符（见表 8-2）。oper[op] 代表这些字符所对应的通用树操作符，比如，oper['+'] 表示 ADD。expr1 能够处理由两个单词构成的扩展赋值运算符，如 +=。

```
⟨augmented assignment 120⟩=                                    119
{
    expect('=');
    p = incr(op, p, expr1(0));
}
```

每个扩展赋值操作符都是一个单词，但上述代码将它们视为两个单词。下面介绍的函数 expr3 就能避免这类错误的解释，函数只有确认了像 - 这样的单词后没有跟随等号时才认为它们是二元操作符。因而，expr1 能够解释 a+=b 中的扩展赋值操作符，并通过 expr3 找出 a+=b 中存在的错误。

函数 incr 为形如 v⊗=e 的表达式生成树，这里 ⊗ 表示任意二元操作符，v 表示左值，e 表示右值。

```
⟨tree.c functions⟩+=                                          119 121
Tree incr(op, v, e) int op; Tree v, e; {
    return asgntree(ASGN, v, (*optree[op])(oper[op], v, e));
}
```

前端调用 incr 并不生成树，而是建立一个 dag。图 8-2 表示 incr 为表达式 *f()+=b 生成的分析树。*f() 仅计算一次，但它计算的结果作为左值却使用了两次，一次是取对应的右值，一次是作为赋值的目标。lcc 为 *f() 单独建立一棵树就反映了这一语义。需要指出的是，这类扩展赋值语句的树需要用到临时变量，这些临时变量在将树转换成节点时生成，详见第 12 章。

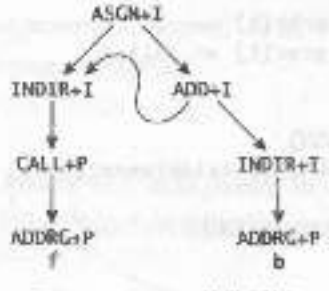


图 8-2 *f()+=b 的分析树

也可以不用 dag，而使用额外的树操作符表示扩展赋值语句。但这样会增加树操作符的数目，使得相关的二元操作符的语义分析过程复杂化。例如执行 + 的语义分析的函数 addtree，必须同时处理 + 和 +=。在其他地方也会使用 dag，例如 9.3 节描述的关于嵌套函数的处理。

8.5 条件表达式

条件表达式的语法形式为：

conditional-expression:
  binary-expression [? *expression* : *conditional-expression*]

当 binary-expression 不为 0 时，条件表达式的值即是 expression 的值，否则它的值等于第三个操作数的值。第三个操作数本身也可以是 conditional-expression，lcc 用 expr2 实现这个语法分析过程：

```
(tree.c functions)+≡                                    ⟨20 122⟩
  static Tree expr2() {
     Tree p = expr3(4);

     if (t == '?') {
        Tree l, r;
        Coordinate pts[2];
        if (Aflag > 1 && isfunc(p->type))
           warning("%s used in a conditional expression\n",
              funcname(p));
        p = pointer(p);
        t = gettok();
        pts[0] = src;
        l = pointer(expr(':'));
        pts[1] = src;
        r = pointer(expr2());
        p = condtree(p, l, r);
        if (events.points)
           ⟨plant event hooks for ?:⟩
     }
     return p;
  }
```

expr2 首先调用 expr3 来分析优先级大于 4 的 binary-expression，最后调用 condtree 生成 COND 树（见图 9-6）。

if 语句和条件表达式常见的一个错误是用函数名替代函数调用。例如，使用表达式 test?a:b 替代表达式 test(a, b)?a:b。这两个表达式都合法，但程序员很少会真正有意使用第一个表达式。lcc 的 -A 选项能对这类可疑的使用发出警告。Aflag 记录 -A 选项的数目，多次出现 -A 选项将引发更多的警告。

lcc 能够在源程序的控制流分支点处执行事件钩子（event hook）。使用这一功能，能够插入树实现表达式级的执行剖面（profiling），或者为源代码级的调试程序插入数据。改变控制流的操作符有 3 种，?: 是其中的一种。建立事件钩子的函数参数应包含表达式中 then 和 else 部分的源坐标，在上面的程序中，pts[0] 和 pts[1] 保存了这些坐标。

条件表达式也可以用来将控制流表示的关系表达式转换为一个值。例如对于表达式 a<b<c，如果 b<c，则 a 为 1，否则置 a 为 0。函数 value 实现这种转换，生成与表达式 c?1:0 对应的 COND 树。

```
(tree.c functions) +≡                                    ↑121 123↓
  Tree value(p) Tree p; {
     int op = generic(rightkid(p)->op);

     if (op==AND || op==OR || op==NOT || op==EQ || op==NE
     ||  op== LE || op==LT || op== GE || op==GT)
        p = condtree(p, consttree(1, inttype),
            consttree(0, inttype));
     return p;
  }
```

lcc 的接口程序指定了两种类别的比较操作符：一种是条件判断，总会伴有跳转指令；另一种用于求值，像 a<b<c，得到 0 或 1。这种设计的优点在于 lcc 可以直接通过指令获得比较的结果，避免隐含在 c?1:0 中的跳转。但是只有当目标机包含这类指令，并且跳转严重影响性能时，才可以从这种方法中获得好处，其他的目标机必须考虑增加操作符带来的影响。只指定条件形式的比较操作可能会牺牲一点灵活性，但可以获取更多的可重定目标性。

8.6 二元表达式

所有优先级在 4 到 13 之间的二元操作符（见表 8-2）的表达式都可定义为如下产生式：

```
binary-expression:
  unary-expression { binary-operator unary-expression }
binary-operator:
  one of || && '|' ^ & == != < > <= >= << >> + - * / %
```

用 8.2 节介绍的函数就能分析此类表达式。使用该技术，binary-expression 的分析函数（不含生成树的代码）为：

```
void expr3(k) int k; {
   if (k > 13)
      unary();
   else {
      expr3(k + 1);
      while (prec[t] == k) {
         t = gettok();
         expr3(k + 1);
      }
   }
}
```

函数 unary 分析 unary-expression。上述代码除了能正确分析 binary-expression 外，还能完成其他工作。expr2 调用 expr3(4)，这是在 expr3 本身之外对 expr3 唯一的外部调用。在第一次调用 unary 之前，从 expr3(5) 到 expr3(14)，执行 10 次递归。在分析源表达式时，这 10 个递归调用按优先级从高到低依次展开。当有某个 k 与 prec[t] 相等时，即 t 是跟在由 unary 分析的表达式后的单词，代码进入 while 循环。多次递归调用 expr3 只是为了测试当前的 k 是否等于 prec[t]，只有一次调用能满足条件。

例如，表达式 a|b 的递归调用序列是：

```
expr3(4)
 expr3(5)
  expr3(6)
   expr3(7)
    expr3(8)
     expr3(9)
      expr3(10)
       expr3(11)
        expr3(12)
         expr3(13)
          expr3(14)
           unary()
   expr3(7)
    ...
   expr3(14)
    unary()
```

在第一次调用 unary 之前（unary 分析 a）的所有调用中，只有 expr3(6) 在 unary 返回后执行有意义的动作。在 while 循环中产生的对 expr3 的递归调用，会导致第二次调用 unary，但所有这些递归调用都不执行有意义的动作。

从此递归序列可看出 expr3 的作用：首先分析 unary-expression，然后按照优先级 13，12，…，4 的顺序处理 binary-expression。采用递减计数的方法，我们可以去掉一些递归。从上面的执行情况可知，只有当优先级与 prec[t] 相同时程序才真正有作用，因而计数就从满足此条件的地方开始：

```
void expr3(k) int k; {
   int k1;
   unary();
   for (k1 = prec[t]; k1 >= k; k1--)
      while (prec[t] == k1) {
         t = gettok();
         expr3(k1 + 1);
      }
}
```

采用这种方法，大部分对 expr3 的递归调用都可以删除。这时，a|b 的分析过程变为：

```
expr3(4)
 unary()
 expr3(7)
  unary()
```

如果往 expr3 中加入一些代码，使程序具有生成树和验证树有效性的功能，并且解决遗留的扩展赋值以及 && 与 || 操作符这两个小问题后，形成了最终的 expr3 程序：

(tree.c functions)+≡ ↑122 125

```
static Tree expr3(k) int k; {
   int k1;
   Tree p = unary();

   for (k1 = prec[t]; k1 >= k; k1--)
      while (prec[t] == k1 && *cp != '=') {
```

```
            Tree r, l;
            Coordinate pt;
            int op = t;
            t = gettok();
            pt = src;
            p = pointer(p);
            if (op == ANDAND || op == OROR) {
                r = pointer(expr3(k1));
                if (events.points)
                    (plant event hooks for && ||)
            } else
                r = pointer(expr3(k1 + 1));
            p = (*optree[op])(oper[op], p, r);
        }
    return p;
}
```

与条件表达式一样，操作符 && 和 || 可以改变控制流，也必须提供事件钩子功能。

从技术上说，操作符 && 和 || 是左结合的，必要时才计算右操作数。如果将它们视为右结合操作符，就可以简化节点的生成。这两个操作符所属的两个优先级中都没有其他操作符。因此，使 && 成为右结合，将很容易产生一棵右重（right-heavy）的 ANDAND 树，而不是左重（left-heavy）的。在 12.3 节将看到，这种显然的错误不仅能在节点产生过程中得到修正，而且与左重的树相比，它还可以为 && 和 || 生成更好的短路（short-circuit）计算代码。要使操作符 || 能右结合，必须在 while 循环中调用 expr3(4) 而不是 expr3(5)。对操作符 &&，则应该 expr3(5) 而不是 expr3(6)。总之，在 while 循环中使用 expr3(k1) 替换 expr3(k1+1) 就可以正确处理这两个操作符。

最后的问题是扩展赋值。expr1 通过识别双单词序列来判断是否为扩展赋值操作符。但这些操作符都是以单个单词组成的，并不是真正的双单词序列。例如，+= 表示相加赋值，而 + = 是一个语法错误。expr1 中用到的识别方法只有在 += 总不被认为是 +− 的情况下才有效。expr3 从另一方向确保了这个条件成立，即当且仅当二元操作符后面没有紧跟 = 号时，才认为它是一个二元操作符。因此，在 a+−b 中的 + 不是一个二元运算符，lcc 能有效地检测出这种错误。

8.7 一元表达式和后缀表达式

下面将介绍处理如下产生式的分析函数：

unary-expression:
  postfix-expression
  unary-operator unary-expression
  '(' *type-name* ')' *unary-expression*
  sizeof *unary-expression*
  sizeof '(' *type-name* ')'

unary-operator:
  one of ++ -- & * + - ~ !

postfix-expression:
  primary-expression { *postfix-operator* }

postfix-operator:
  '[' *expression* ']'
  '(' [*assignment-expression* { , *assignment-expression* }] ')'

```
. identifier
-> identifier
++
--
```

其中 primary-expression 的产生式见下节。由于这些产生式都不复杂，因此相应的分析函数也比较简单。比如分析 unary-expression，大多数一元操作的处理仅需 3 步：处理操作符、分析操作数、生成树。

```
⟨tree.c functions⟩+≡                                    ▲123 126▼
  static Tree unary() {
     Tree p;

     switch (t) {
     case '*':    (p ← unary 125) ⟨indirection 138⟩ break;
     case '&':    (p ← unary 125) ⟨address of 137⟩ break;
     case '+':    (p ← unary 125) ⟨affirmation⟩ break;
     case '-':    (p ← unary 125) ⟨negation 136⟩ break;
     case '~':    (p ← unary 125) ⟨complement⟩ break;
     case '!':    (p ← unary 125) ⟨logical not⟩ break;
     case INCR:   (p ← unary 125) ⟨preincrement 125⟩ break;
     case DECR:   (p ← unary 125) ⟨predecrement⟩ break;
     case SIZEOF: t = gettok(); { ⟨sizeof 126⟩ } break;
     case '(':
        t = gettok();
        if (istypename(t, tsym)) {
           ⟨type cast 138⟩
        } else
           p = postfix(expr(')'));
        break;
     default:
        p = postfix(primary());
     }
     return p;
  }
⟨p ← unary 125⟩≡                                        125
  t = gettok(); p = unary();
```

函数的大部分都是做语义检查，下一章将详细介绍语义问题。前面提到的 3 步，即处理运算符、分析操作数和生成树在这个函数里都能实现。表达式 ++e 在语义上等价于扩展赋值 e+=1，因而函数 incr 能为 += 生成树。

```
⟨preincrement 125⟩≡                                     125
  p = incr(INCR, pointer(p), consttree(1, inttype));
```

前自减（predecrement）处理与上面类似。

sizeof('type-name')' 得到一个 size_t 类型的常量，它给出 type-name 的一个实例所占的字节数。在 lcc 中，size_t 是无符号数。同样，unary-expression 中 sizeof unary-expression 也仅能用于提供大小已知的类型；unary-expression 并不在运行时计算。分析 sizeof 的代码主要的工作就是区分上面这两种 sizeof 形式，并找出相应的类型。注意，如果操作数是 type-name，则必须加上括号。

```
⟨sizeof 126⟩≡                                                    125
  Type ty;
  p = NULL;
  if (t == '(') {
     t = gettok();
     if (istypename(t, tsym)) {
        ty = typename();
        expect(')');
     } else {
        p = postfix(expr(')'));
        ty = p->type;
     }
  } else {
     p = unary();
     ty = p->type;
  }
  if (isfunc(ty) || ty->size == 0)
     error("invalid type argument '%t' to 'sizeof'\n", ty);
  else if (p && rightkid(p)->op == FIELD)
     error("'sizeof' applied to a bit field\n");
  p = consttree(ty->size, unsignedtype);
```

由程序可知，sizeof的参数不能是函数、不完全类型和从位域中导出的类型。

在 unary 和 <sizeof> 中，左括号代表一个 primary-expression，或者如果它的下一个单词是类型名，则表示开始进行类型转换。

如果左括号后不引入类型转换，此时调用函数 primary 分析括号内的表达式就晚了，因此 unary 必须能够处理这种情况。这就解释了为什么 postfix 希望它的调用者首先调用 primary，将得到的结果树作为参数传送给 postfix，而不是在 postfix 中调用 primary。

```
⟨tree.c functions⟩+≡                                    125 127
  static Tree postfix(p) Tree p; {
     for (;;)
        switch (t) {
        case INCR:  ⟨postincrement 126⟩ break;
        case DECR:  ⟨postdecrement⟩ break;
        case '[':   ⟨subscript 139⟩ break;
        case '(':   ⟨calls 143⟩ break;
        case '.':   ⟨struct.field⟩ break;
        case DEREF: ⟨pointer->field 139⟩ break;
        default:
           return p;
        }
  }
```

同样，postfix 的大部分代码段是在检测操作数的语义以及生成相应的树（详见下章）。但后自增（postincrement）以及后自减（postdecrement）的树能通过 incr 生成：

```
⟨postincrement 126⟩≡                                            126
  p = tree(RIGHT, p->type,
     tree(RIGHT, p->type,
        p,
        incr(t, p, consttree(1, inttype))),
```

```
        p);
    t = gettok();
```

后缀的 ++ 使操作数加 1，但返回的是加 1 前的值。因此 ++ 的树是一个 dag。例如，表达式 i++ 生成的树如图 8-3 所示。两个 RIGHT 操作符保证了计算顺序的正确性，整个表达式的值是 i 的右值，低层的 RIGHT 树确保在 i 通过 ASGN+I 树加 1 前对 i 的右值进行计算和保存。p++ 的树的结构也是相同的，这里 p 是一个指针。注意，p++ 表示将 p 增加，增加的量为它指向的对象的大小。

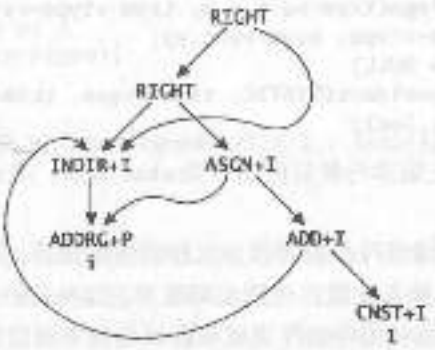


图 8-3 i++ 对应的树

8.8 基本表达式

处理表达式的最后一个分析函数是 primary：

primary-expression:
  identifier
  constant
  string-literal
  '(' *expression* ')'

它与 8.2 节描述的简单表达式语法的分析函数 factor 类似，剩下需要处理的只有常量和标识符：

```
⟨tree.c functions⟩+≡                                    ↑126 128↓
  static Tree primary() {
     Tree p;

     switch (t) {
     case ICON:
     case FCON: ⟨numeric constants 128⟩ break;
     case SCON: ⟨string constants 128⟩ break;
     case ID:   ⟨an identifier 130⟩ break;
     default:
        error("illegal expression\n");
        p = consttree(0, inttype);
     }
     t = gettok();
     return p;
  }
```

CNST 树的 u.v 域保存了整数和浮点常量的值：

```
⟨numeric constants 128⟩=                                          127
  p = tree(CNST + ttob(tsym->type), tsym->type, NULL, NULL);
  p->u.v = tsym->u.c.v;
⟨u fields for Tree variants 128⟩=                                 128 113
  Value v;
```

字符串常量可视为只读变量的简写，这些只读变量被初始化为字符串常量的值：

```
⟨string constants 128⟩=                                           127
  tsym->u.c.v.p = stringn(tsym->u.c.v.p, tsym->type->size);
  tsym = constant(tsym->type, tsym->u.c.v);
  if (tsym->u.c.loc == NULL)
     tsym->u.c.loc = genident(STATIC, tsym->type, GLOBAL);
  p = idtree(tsym->u.c.loc);
```

生成的变量及其初始化过程在编译的最后由函数 finalize 完成，表示字符串的树与表示生成的标识符的树一样。

idtree(p) 生成一棵树用于访问符号表入口 p 所标识的标识符。标识符根据它们的作用域、生存期（参数、自动局部变量和静态变量，包括全局变量）以及它们的类型（数组、函数及非数组对象）来分类。函数 idtree 根据标识符的作用域和存储类别来决定需要的地址操作符。然后根据其类型来决定访问它的树的结构，并且在树的 u.sym 域存储一个指向符号表入口 p 的指针：

```
⟨u fields for Tree variants 128⟩+=                                128 141 113
  Symbol sym;

⟨tree.c functions⟩+=                                              127 129
  Tree idtree(p) Symbol p; {
     int op;
     Tree e;
     Type ty = p->type ? unqual(p->type) : voidtype;

     p->ref += refinc;
     if (p->scope  == GLOBAL
     ||  p->sclass == STATIC || p->sclass == EXTERN)
        op = ADDRG+P;
     else if (p->scope == PARAM) {
        op = ADDRF+P;
        if (isstruct(p->type) && !IR->wants_argb)
           ⟨return a tree for a struct parameter 129⟩
     } else
        op = ADDRL+P;
     if (isarray(ty) || isfunc(ty)) {
        e = tree(op, p->type, NULL, NULL);
        e->u.sym = p;
     } else {
        e = tree(op, ptr(p->type), NULL, NULL);
        e->u.sym = p;
        e = rvalue(e);
     }
     return e;

  }

⟨tree.c data⟩+=                                                   118
  float refinc = 1.0;
```

p->ref 就是对 p 所标识的标识符引用次数的估计值，其他函数可以通过改变 refinc 的值来调节对 p 的引用的权值。所有外部、静态和全局标识符都通过 ADDRG 寻址，参数通过 ADDRF 寻址，而局部变量通过 ADDRL 寻址。

数组和函数不能直接用作左值或右值，所以只有专用的地址操作符才能引用它们。其他类型的树引用标识符的右值。例如图 8-3 所描述的 i 的右值树。函数 rvalue 加上 INDIR：

```
⟨tree.c functions⟩+≡                                    128 129
  Tree rvalue(p) Tree p; {
     Type ty = deref(p->type);

     ty = unqual(ty);
     return tree(INDIR + (isunsigned(ty) ? I : ttob(ty)),
        ty, p, NULL);
  }
```

rvalue 的参数可以是任何一棵表示指针值的树，而函数 lvalue 的参数只能是表示右值（一个可寻址位置的内容）的树。由函数 rvalue 生成的 INDIR 树也表明了这棵树是一个有效的左值，并且左值的地址可以通过消除 INDIR 获得。函数 lvalue 实现了这种检查和转换：

```
⟨tree.c functions⟩+≡                                    129 132
  Tree lvalue(p) Tree p; {
     if (generic(p->op) != INDIR) {
        error("lvalue required\n");
        return value(p);
     } else if (unqual(p->type) == voidtype)
        warning("'%t' used as an lvalue\n", p->type);
     return p->kids[0];
  }
```

结构参数的树还依赖于接口域 wants_argb 的值。如果 wants_argb 等于 1，则上面所示的代码就构建相应的分析树。如果参数为 x，这棵树的形式就是 (INDIR+B(ADDRF+P x))。如果 wants_argb 等于 0，则编译前端是这样实现结构参数的：即通过在调用中复制参数副本，并传递指向该副本的指针。这样，对一个结构参数的引用就需要通过另一个间接访问来访问结构本身：

```
⟨return a tree for a struct parameter 129⟩=                 128
  {
     e = tree(op, ptr(ptr(p->type)), NULL, NULL);
     e->u.sym = p;
     return rvalue(rvalue(e));
  }
```

对于参数 x，上段代码构建树：

```
(INDIR+B (INDIR+P (ADDRF+P x)))
```

如果需要使用标识符树，就要用到函数 idtree，例如对于字符串常量和标识符：

```
⟨an identifier 130⟩=                                              127
  if (tsym == NULL)
    ⟨undeclared identifier⟩
  if (xref)
    use(tsym, src);
  if (tsym->sclass == ENUM)
    p = consttree(tsym->u.value, inttype);
  else {
    if (tsym->sclass == TYPEDEF)
      error("illegal use of type name '%s'\n", tsym->name);
    p = idtree(tsym);
  }
```

如果 tsym 为空，则标识符没有声明，如果它又不是一个函数调用，这就会报错（见练习 8.5）。枚举标识符代表的是常量，它将产生常量树，而不是标识符树。

深入阅读

在编译器领域，用一个分析函数代替 n 个函数来处理 n 级优先级是非常流行的，但是对于这种技术的解释却很少。Hanson（1985）介绍了这种技术及其在 lcc 中的使用情况。Holzmann（1988）在其图像处理语言 pico 中使用了类似的技术。从技术上说，这种技术等价于在 BCPL（Richards and Whitby-Strevens，1979）中使用的技术，但是在 BCPL 中，操作符以及操作符的优先级和结合方式都分散在 BCPL 的整个代码中，而不是存储在表格中。

练习

8.1 实现

```
⟨tree.c exported functions⟩=                                      130
  extern Tree retype ARGS((Tree p, Type ty));
```

如果 p->ty==ty，就返回 p，否则返回 ty 类型的 p 的副本。记住所有的树操作函数都可以使用该函数。

8.2 实现

```
⟨tree.c exported functions⟩+=                                     130 130
  extern Tree rightkid ARGS((Tree p));
```

该函数返回以 p 为根节点的最深 RIGHT 树中最右边的非 RIGHT 操作数。不要忘记 RIGHT 节点可以有一个或者两个操作数（但不能是 0 个）。

8.3 实现

```
⟨tree.c exported functions⟩+=                                     130
  extern int hascall ARGS((Tree p));
```

如果 p 包含一棵 CALL 树，返回值为 1，否则返回值为 0。不要忘记接口标志 mulops_calls。

8.4 利用 8.6 节开始部分介绍的直接方式重新实现 expr3 函数，并测试它的性能。你认为去除递归调用后获得的好处是否值得？

8.5 完成 8.8 节所用的 <undeclared identifier> 代码。如果标识符被用作一个合法函数，当前作用域和 externals 表就隐式地声明了该标识符，否则就会对未声明的标识符报错。因此对标识符的隐式声明十分有用，可以使得编译继续进行。

8.6 如 8.4 节所介绍的，incr 函数返回的树是 dag。请你为扩展赋值类作符增加新的树操作符，并利用新增加的操作符重写 incr 函数，避免使用 dag。你需要更改函数 listnodes，可能还得改变 enode.c 中的语义函数。

表达式语义

表达式必须在语法和语义上都是正确的。上一章介绍的分析函数所处理的是语法问题和一些简单的语义问题，例如为常量和标识符构建抽象语法树。本章专门介绍为表达式构建树时必须要做的语义分析。语义分析必须处理难度相当的 3 个子问题：隐式转换（implicit conversion）、类型检查（type checking）和计算次序（order of evaluation）。

隐式转换就是那些不在源程序中出现，但是根据标准 C 的语义规则，必须被编译器添加进来的转换。例如，对于表达式 a+b，如果 a 是整型而 b 是浮点型，a+b 在语义上是正确的，但是必须添加一个将 a 的值转换为浮点类型的隐式转换。

类型检查是为了确保操作符的操作数的类型是合法的，并确定结果类型以及应使用哪一个与类型相关的操作。例如，对 a+b 的类型检查，首先验证 a 和 b 的类型是不是合法的算术类型组合，然后根据 a 和 b 的类型来确定结果类型，而结果类型应该是一种数值类型。另外，还要确定使用哪一个与类型相关的加法操作。在表达式 a+b 的例子中，经过类型检查后，表达式等价于 (float) a+b，并确定使用浮点加法操作。

编译器必须遵守标准 C 的计算顺序规则来生成树。对于大多数操作符，计算顺序都没有规定。例如，在 a[i++]=i 中，无论 i 是先自增再赋值还是先赋值再自增都没有规定。只有少数的操作符规定了计算顺序，例如在 f()&&g() 中，f 必须先于 g 被调用；如果 f 返回值为 0，就没有必要再调用 g 了。类似地，在 (f(), g()) 中，f 必须先于 g 被调用。正如在函数 expr 中所描述的，RIGHT 树很好地定义了计算顺序，可以用来强制指定计算顺序。

9.1 转换

转换函数接受一个或多个类型并返回一个结果类型，或者接受一棵树和一个类型，返回一棵经过相应类型转换的树。promote(Type ty) 就是前一类转换的例子：它实现了整数的提升。如果有必要，它就将一个整数类型 ty 扩展为整型、无符号整型或长整型。标准 C 规定，整数类型的提升应保持值，包括符号，而不是以无符号整型方式保持值。例如，一个无符号字符被提升为整型，而不是无符号整型。如果一个更小的整型（或一个位域）的所有值都可以用整型表示，则这个更小的整型（或位域）就被提升为整型。否则，就会被提升为无符号整型。在 lcc 编译器中，整型必须能够表示更小整数类型的所有值，所以在函数 promote 中最后一个 if 语句返回 inttype。

函数 binary 实现了常规算术转换（usual arithmetic conversion），它以两个算术类型为参数，返回任意二元算术操作结果的类型：

```
⟨tree.c functions⟩+≡                                    129 133
  Type binary(xty, yty) Type xty, yty; {
      if (isdouble(xty) || isdouble(yty))
          return doubletype;
      if (xty == floattype || yty == floattype)
          return floattype;
      if (isunsigned(xty) || isunsigned(yty))
```

```
        return unsignedtype;
    return inttype;
}
```

lcc 编译器假定 double 和 long double 所需字节大小相同，long 和 int（包括无符号和有符号）所需字节大小也相同。这些假设简化了标准规定的常规算术转换，因而简化了函数 binary。在通常情况下，long double 比 double 所需字节多，而 long 又比 unsigned int 所需字节多，下表总结了通常情况下的标准规定：

long double
double
float
unsigned long int
long int
unsigned int
int

出现在这张表最上面的操作数类型就是其他操作数可以转换成的类型。如果没有指定这些类型中的某一个，那么操作数就被转换为整型。基于 lcc 编译器的假设，函数 binary 中的第一条 if 语句处理前两种类型，第二条 if 语句处理 float 类型；由于 lcc 的 signed long 不能表示所有无符号类型的值，所以需要第三条 if 语句处理 4 种整型。

函数 pointer 用于处理第二类隐式转换，即输入参数为一棵树，返回可能经过转换后形成的树。当数组和函数类型被用于表达式时，需要转换为指针类型。例如，(ARRAY T) 和 (POINTER T) 转换为 (FUNCTION T) 和 (POINTER(FUNCTION T))。

```
⟨tree.c functions⟩+≡                                    ↑132 133↓
Tree pointer(p) Tree p; {
    if (isarray(p->type))
        p = retype(p, atop(p->type));
    else if (isfunc(p->type))
        p = retype(p, ptr(p->type));
    return p;
}
```

函数 rvalue、lvalue 和 value 也可以看作一种转换。而函数 cond 就是函数 value 的逆转换：它处理一棵可表示某个值的树，添加一个与 0 进行比较的操作，从而将它转换为一棵表示条件的树。

```
⟨tree.c functions⟩+≡                                    ↑133 134↓
Tree cond(p) Tree p; {
    int op = generic(rightkid(p)->op);

    if (op == AND || op == OR || op == NOT
    ||  op == EQ  || op == NE
    ||  op == LE  || op == LT || op == GE || op == GT)
        return p;
    p = pointer(p);
    p = cast(p, promote(p->type));
    return (*optree[NEQ])(NE, p, consttree(0, inttype));
}
```

条件没有值，仅能用于通过其结果影响控制流，如在 if 语句中。当输入参数的值非零时，函数 cond 就返回一棵结果为真的树。

函数 cond 调用函数 cast 将它的参数转换为其提升类型给出的基本类型。cast 实现了图 9-1 所描述的转换。图 9-1 中的每一个箭头都表示了一种转换操作。例如，从 I 到 D 的箭头表示了从整型到双精度型的转换，即 CVI+D，而相反方向的箭头则表示了从双精度型到整型的转换，即 CVD+I。I 上面的 C 表示有符号字符类型，而 U 上面的 C 表示无符号字符类型；图中两个 S 的解释也同它们一样。

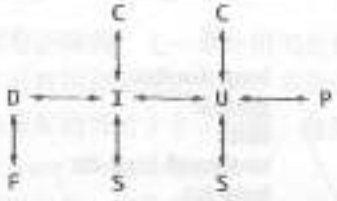


图 9-1 类型转换

没有箭头连接的转换可以通过组合已有的转换操作来实现。例如，要将一个有符号短整型 s 转换为浮点型，可以通过下面的操作组合来实现：首先将它转换为整型，然后转换为双精度型，最后转换为浮点型。树为 (CVD+F(CVI+D(CVS+I s)))。对无符号整型和双精度之间转换的处理有所不同，这将在下面详述。

对应于上面列出的 3 步，函数 cast 可分为 3 部分。首先，p 被转换为它的父类型（supertype），即 D、I 或 U。如果有必要，接着转换为目标类型的父类型。最后，转换为目标类型。

```
⟨tree.c functions⟩+≡                                    (133) 140
  Tree cast(p, type) Tree p; Type type; {
     Type pty, ty = unqual(type);

     p = value(p);
     if (p->type == type)
        return p;
     pty = unqual(p->type);
     if (pty == ty)
        return retype(p, type);
     ⟨convert p to super(pty) 134⟩
     ⟨convert p to super(ty) 135⟩
     ⟨convert p to ty 136⟩
     return p;
  }
```

如代码所示，这些转换针对参数类型的非限定形式进行。函数 super 返回其参数的父类型。

cast 的第 1 步就是将符号整型转换为整型，浮点转换为双精度，指针转换为无符号整型：

```
⟨convert p to super(pty) 134⟩=                                  134
  switch (pty->op) {
  case CHAR:    p = simplify(CVC, super(pty), p, NULL); break;
  case SHORT:   p = simplify(CVS, super(pty), p, NULL); break;
  case FLOAT:   p = simplify(CVF, doubletype, p, NULL); break;
  case INT:     p = retype(p, inttype);                 break;
  case DOUBLE:  p = retype(p, doubletype);              break;
  case ENUM:    p = retype(p, inttype);                 break;
  case UNSIGNED:p = retype(p, unsignedtype);            break;
  case POINTER:
     if (isptr(ty)) {
```

```
            (pointer-to-pointer conversion 135)
        } else
            p = simplify(CVP, unsignedtype, p, NULL);
        break;
    }
```

函数 simplify 像函数 tree 一样构建常，但是如果有可能，它还要进行常量折叠；并且，如果它的第一参数是一个通用操作符，simplify 就根据它的第一、第二操作数构造与类型相关的操作符。lcc 编译器规定指针可以用无符号整型表示，因此指针操作可以使用无符号操作符，这样就减少了操作符数量。有一个特例：没有指针到指针的转换 CVP+U，因为该转换没有任何用处。

```
(pointer-to-pointer conversion 135)=                                    135
    if (isfunc(pty->type) && !isfunc(ty->type)
    || !isfunc(pty->type) &&  isfunc(ty->type))
        warning("conversion from '%t' to '%t' is compiler _
            dependent\n", p->type, ty);
    return retype(p, type);
```

如果出现对象指针与函数指针之间的转换，lcc 编译器将会给出警告信息。这是因为标准 C 允许不同类型的指针大小不同，而 lcc 编译器却假定它们有相同的大小。

cast 的第 2 步就是转换 p，这时 p 是一个双精度型、整型或者无符号整型，如果需要，p 就被转换为这 3 种类型中的任意一种，也就是 ty 的父类型：

```
(convert p to super(ty) 135)=                                           134
    {
        Type sty = super(ty);
        pty = p->type;
        if (pty != sty)
            if (pty == inttype)
                p = simplify(CVI, sty, p, NULL);
            else if (pty == doubletype)
                if (sty == unsignedtype) {
                    (double-to-unsigned conversion)
                } else
                    p = simplify(CVD, sty, p, NULL);
            else if (pty == unsignedtype)
                if (sty == doubletype) {
                    (unsigned-to-double conversion 136)
                } else
                    p = simplify(CVU, sty, p, NULL);
    }
```

注意在图 9-1 中 D 与 U 之间没有直接的转换箭头。大多数机器都有指令实现有符号整型与双精度之间的转换，但是很少有机器有指令实现无符号整型与双精度之间的转换，所以没有 CVU+D 或 CVD+U。取而代之的是，编译前端构建树来实现这种转换，而前提是假定整型和无符号整型有相同的大小。

通过构建等价于

```
2.*(int)(u>>1) + (int)(u&1)
```

的表达式可以将一个无符号整型u转换为双精度型。u>>1（等价于u/2）腾空符号位，这样，就可以通过整型到双精度型的转换，实现无符号整型到双精度型的转换。表达式中的浮点乘和浮点加计算出正确的值。下面的代码为该表达式构建树：

```
⟨unsigned-to-double conversion 136⟩=                                    135
  Tree two = tree(CNST+D, doubletype, NULL, NULL);
  two->u.v.d = 2.;
  p = (*optree['+'])(ADD,
     (*optree['*'])(MUL,
        two,
        simplify(CVU, inttype,
           simplify(RSH, unsignedtype,
              p, consttree(1, inttype)), NULL)),
     simplify(CVU, inttype,
        simplify(BAND, unsignedtype,
           p, consttree(1, unsignedtype)), NULL));
```

注意这棵树是一个dag：它引用了两次p。函数optree用来处理浮点乘和浮点加操作，因此无符号整型到双精度型的转换需要调用optree。

编译前端通过构建相应表达式树来实现双精度型和无符号整型之间的转换。练习9.2给出了具体的实现方法。

现在得到的树表示了一个ty的父类型的值，cast的第3步就是将树转换成目标类型。这一步其实就是super函数的逆转换：

```
⟨convert p to ty 136⟩=                                                  134
  if (ty == signedchar || ty == chartype || ty == shorttype)
     p = simplify(CVI, type, p, NULL);
  else if (isptr(ty)
  || ty == unsignedchar || ty == unsignedshort)
     p = simplify(CVU, type, p, NULL);
  else if (ty == floattype)
     p = simplify(CVD, type, p, NULL);
  else
     p = retype(p, type);
```

9.2 一元操作符和后缀操作符

上一节的转换函数为每一个操作实现了机器需要的语义检查。标准还规定了操作数的限制（如它们的类型）和操作的语义（如结果类型）。典型的例子如一元操作符-，标准规定如下：

一元操作符-的操作数应当具有算术类型。

一元操作符-的结果就是操作数取反。因为对操作数进行了整型的提升，因此结果类型是提升类型。

每一个操作对应的代码实现了标准的规定，包括检查操作数的树是否符合这种限制并为运算结果构建相应的树。例如，一元操作符-的代码如下：

```
⟨negation 136⟩=                                                         125
  p = pointer(p);
  if (isarith(p->type)) {
     p = cast(p, promote(p->type));
     if (isunsigned(p->type)) {
```

```
            warning("unsigned operand of unary -\n");
            p = simplify(NEG, inttype, cast(p, inttype), NULL);
            p = cast(p, unsignedtype);
         } else
            p = simplify(NEG, p->type, p, NULL);
      } else
         typeerror(SUB, p, NULL);
```

函数 typeerror 负责对一元或二元操作符的非法操作数进行诊断。例如，如果 pi 是一个整型指针，那么 pi 不是算术类型，因此 -pi 是非法的，函数 typeerror 产生下列信息：

```
operand of unary - has illegal type 'pointer to int'
```

如果一元操作符 - 使用无符号操作数，根据标准规定，要给出警告信息，这有助于检查可能的错误。即使 lcc 编译器支持有符号长整型包含所有负的无符号整型，这种警告信息也是合适的，因为整型的提升不会产生任何长整型。

对于一元操作符 &，标准规定如下：

一元操作符 & 的操作数或者是一个函数指示（function designator）或者是一个对象的左值，该对象不能是位域，也不能声明为具有寄存器存储类别。

一元操作符 & 处理类型为 T 的操作数并返回它的地址（类型为 (POINTER T)）。大多数情况下，上述语义是通过函数 lvalue 实现的，lvalue 函数返回 INDIR 节点下的地址树。函数和数组是个例外，它们没有 INDIR 节点：

```
〈address of 137〉≡                                              125
   if (isarray(p->type) || isfunc(p->type))
      p = retype(p, ptr(p->type));
   else
      p = lvalue(p);
   if (isaddrop(p->op) && p->u.sym->sclass == REGISTER)
      error("invalid operand of unary &; '%s' is declared _
         register\n", p->u.sym->name);
   else if (isaddrop(p->op))
      p->u.sym->addressed = 1;

〈tree.c exported macros〉≡
   #define isaddrop(op) \
      ((op)==ADDRG+P || (op)==ADDRL+P || (op)==ADDRF+P)

〈symbol flags 37〉+≡                                    ◆ 163  28 ▼
   unsigned addressed:1;
```

正如上面所说的，一元操作符 & 的操作数不能是寄存器变量或位域。位域的树没有 INDIR，所以 lvalue 可以区分出它们。对于经常被引用的局部变量和参数，编译前端会在它们传到编译后端之前将它们的存储类型改为 REGISTER，但是前端不能改变那些地址被使用的变量的存储类型，也就是那些具有 addressed 标记的符号。

一元操作符 * 是一元操作符 & 的逆运算；它处理具有类型（POINTER T）的操作数，返回 INDIR 分析树，该树表示类型 T 的右值。同样，大部分的工作由函数 rvalue 完成，而数组和函数的指针需要进行特殊处理。

```
⟨indirection 138⟩≡                                              125
  p = pointer(p);
  if (isptr(p->type)
  && (isfunc(p->type->type) || isarray(p->type->type)))
     p = retype(p, p->type->type);
  else {
     if (YYnull)
        p = nullcheck(p);
     p = rvalue(p);
  }
```

YYnull 和 nullcheck 捕获空指针错误，参见练习 9.5。

类型转换说明了显式的转换。某些类型转换，比如指针到指针的转换，不会生成代码，只需简单地指定表达式的类型。其他类型转换，比如整型到浮点型的转换，会生成代码，影响运行时转换。下面的代码和函数 cast 实现了标准所规定的转换规则。

标准规定，类型转换的目标类型必须是限定的或未限定的标量类型或者 void，操作数的类型（即源类型）必须是一个标量类型。类型转换的语义分析分成 3 部分：计算和检查目标类型，分析操作数，计算和检查源类型。函数 typename 分析类型声明符，返回结果 Type，除了对限定枚举类型进行处理外，typename 的大部分工作就是计算目标类型：

```
⟨type cast 138⟩≡                                    138  125
  Type ty, ty1 = typename(), pty;
  expect(')');
  ty = unqual(ty1);
  if (isenum(ty)) {
     Type ty2 = ty->type;
     if (isconst(ty1))
        ty2 = qual(CONST, ty2);
     if (isvolatile(ty1))
        ty2 = qual(VOLATILE, ty2);
     ty1 = ty2;
     ty = ty->type;
  }
```

上面的代码就是用来计算目标类型 ty1 和它的未限定变体 ty。枚举类型转换的目标类型是与其相关联的整数类型（在 lcc 编译器中就是整型），而不是枚举类型。这样，在分析操作数之前，ty1 和 ty 必须被重新计算。

```
⟨type cast 138⟩+≡                                   138 138  125
  p = pointer(unary());
  pty = p->type;
  if (isenum(pty))
     pty = pty->type;
```

如果目标类型和源类型都是合法的，则分析树转换为未限定类型 ty：算术类型和枚举类型可以相互转换；指针类型可以转换为其他指针类型；指针类型还可以转换为整数类型，反之亦然，但是如果类型之间的 size 不同，结果就不确定了；而任何类型都可以被转换为 void 类型。

```
⟨type cast 138⟩+≡                                   138 139  125
  if (isarith(pty) && isarith(ty)
  ||  isptr(pty)   && isptr(ty))
```

```
    p = cast(p, ty);
  else if (isptr(pty) && isint(ty)
  ||       isint(pty) && isptr(ty)) {
    if (Aflag >= 1 && ty->size < pty->size)
        warning("conversion from '%t' to '%t' is compiler _
          dependent\n", p->type, ty);
    p = cast(p, ty);
  } else if (ty != voidtype) {
    error("cast from '%t' to '%t' is illegal\n",
       p->type, ty1);
    ty1 = inttype;
  }
```

记住，对于对象指针和函数指针之间的转换，cast会给出警告信息。

最后一步就是根据需要将p转换为可能的限定类型：

```
(type cast 138)+≡                                    ▲ 138  125
  p = retype(p, ty1);
  if (generic(p->op) == INDIR)
    p = tree(RIGHT, ty, NULL, p);
```

转换不是一个左值，所以如果p是一棵INDIR树，它隐含在一棵RIGHT树中，从而防止函数lvalue将它作为一个左值。

标准C规定e[i]形式的表达式等价于*(e+i)，其中的一个操作数必须是指针而另一个必须是整数类型。一旦e和i确定了，加法操作的语义函数就可以完成大部分的处理工作：

```
(subscript 139)≡                                     126
  {
     Tree q;
     t = gettok();
     q = expr(']');
     if (YYnull)
        if (isptr(p->type))
           p = nullcheck(p);
        else if (isptr(q->type))
           q = nullcheck(q);
     p = (*optree['+'])(ADD, pointer(p), pointer(q));
     if (isptr(p->type) && isarray(p->type->type))
        p = retype(p, p->type->type);
     else
        p = rvalue(p);
  }
```

最后一个if语句处理n维数组：例如，如果x声明为int x[10][20]，则x[i]表示第i行，它的类型是(ARRAY 20(INT))，但x[i]不是一个左值。i[x]的解释与x[i]的类似，二者可以等价，但很少使用i[x]。

对域的引用类似于数组下标，生成的是引用该域右值的树，因此是左值；或者对于数组域，生成引用域地址的树。这种分析是显而易见的：

```
(pointer->field 139)≡                                126
  t = gettok();
  p = pointer(p);
```

```
if (t == ID) {
   if (isptr(p->type) && isstruct(p->type->type)) {
      if (YYnull)
         p = nullcheck(p);
      p = field(p, token);
   } else
      error("left operand of -> has incompatible _
         type '%t'\n", p->type);
   t = gettok();
} else
   error("field name expected\n");
```

field 调用 fieldref，fieldref 返回一个 Field，它包含给定域的类型和位置。

```
⟨tree.c functions⟩+≡                                    134
  Tree field(p, name) Tree p; char *name; {
     Field q;
     Type ty1, ty = p->type;

     if (isptr(ty))
        ty = deref(ty);
     ty1 = ty;
     ty = unqual(ty);
     if ((q = fieldref(name, ty)) != NULL) {
        ⟨access the field described by q 140⟩
     } else {
        error("unknown field '%s' of '%t'\n", name, ty);
        p = rvalue(retype(p, ptr(inttype)));
     }
     return p;
  }
```

field 必须能够处理限定的结构类型。如果结构类型声明为 const 或者 volatile，对于域的引用也必须是类似限定的，即使在域的声明符中不允许有限定符。q->type 是域的类型，而 q->offset 就是域的字节偏移量。

```
⟨access the field described by q 140⟩=                  141  140
  if (isarray(q->type)) {
     ty = q->type->type;
     ⟨qualify ty, when necessary 140⟩
     ty = array(ty, q->type->size/ty->size, q->type->align);
  } else {
     ty = q->type;
     ⟨qualify ty, when necessary 140⟩
     ty = ptr(ty);
  }
  p = simplify(ADD+P, ty, p, consttree(q->offset, inttype));

⟨qualify ty, when necessary 140⟩=                        140
  if (isconst(ty1) && !isconst(ty))
     ty = qual(CONST, ty);
  if (isvolatile(ty1) && !isvolatile(ty))
     ty = qual(VOLATILE, ty);
```

函数 simplify 返回域地址的树，或者返回表示存放位域的无符号数的地址的树。如果 q->lsb 非零，那么 q->lsb 指向位域的最低有效位加 1 的位置，lsb 说明了该域是位域。位域通过 FIELD 树被引用，且不是左值。

```
⟨access the field described by q 140⟩+≡                    ⟨140⟩ 140
  if (q->lsb) {
    p = tree(FIELD, ty->type, rvalue(p), NULL);
    p->u.field = q;
  } else if (!isarray(q->type))
    p = rvalue(p);

⟨u fields for Tree variants 128⟩+≡                          ⟨128⟩ 113
  Field field;
```

FIELD 树的 u.field 域指向 Field 结构（4.6 节中定义），该结构描述了位域。

表达式 e.name 等价于 (&e)->name，所以函数 field 也被 <struct.field> 代码段调用。该代码段为 "." 的左操作数（e）的地址构建树，并将该树传递给 field 函数。

9.3 函数调用

函数调用的语法分析很容易但是语义分析比较复杂。语义分析不仅要处理新风格的函数，还必须处理旧风格的函数，而标准对它们所规定的语义对于转换和参数检查有影响。语义分析必须处理参数的计算顺序（依赖于接口标志 left to right），实现以传值方式传递和返回结构（依赖于接口标志 wants_argb 和 wants_callb），还要处理实参中包括其他函数调用的情况。所有这些不同的处理方式都是由 lcc 编译器接口产生的，而不是标准中的规则。这些不同的处理方式都可以去掉。但是，一旦去掉，就使得 lcc 编译器不能生成可模拟一个或多个目标机的调用序列的代码。这种复杂性就是为与现存的函数调用约定相兼容而付出的代价。

例如：

```
char *str;
struct node { ... } a;
struct node f(struct node x, char c, int i) { ... }
main () { f(a, '\n', atoi(str)); }
```

这段没有什么实际意义的程序可以展示几乎所有的复杂性。调用函数 f 的树如图 9-2 所示，我们假定 wants_argb 等于 1。CALL+B 的右操作数将在下面解释。图中的所有 RIGHT 树合作实现了期望的计算次序。CALL 树的左操作数就是一棵计算参数（ARG 子树）和函数自身的 RIGHT 树。图 9-2 中最左端的 RIGHT 树就是一个例子。图 9-2 中以阴影标记的 RIGHT 节点为根节点的树表示了对函数 atoi 的嵌套调用。当遍历这棵树生成代码时，就可以保证对 atoi 的调用代码出现在计算 f 的参数之前。通常，对于每一个包含函数调用的参数都有一棵 RIGHT 树，而且如果函数名本身是含函数调用的表达式，也会有一棵 RIGHT 树。

实参用 ARG 树表示。首先表示的是最右参数；ARG 树的右操作数就表示对其余实参的计算。ARG 树可以有两个操作数。最上面的 ARG+I 树表示参数 atoi(str)，它的左操作数指向 CALL+I 子树。这里出现的 RIGHT 子树会使编译后端将 atoi 函数返回的值存储在一个临时变量中，再通过 atoi 中从 ARG+I 到 CALL+I 的引用，将该值传递给函数 f。

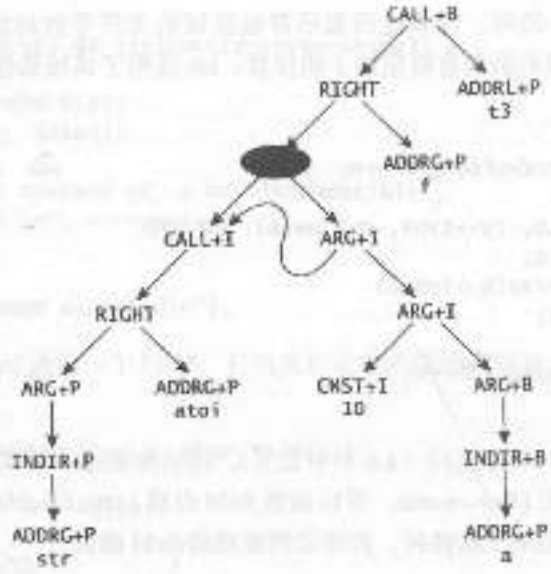


图 9-2 f(a,"\n",atoi(str)) 的树

第二个 ARG+I 表示第二个参数，即换行符。f 有一个原型，是一个新风格的函数，所以它要求整数常量'\n'转换为字符来传递。大多数机器都有限制，比如栈对齐，这种强制使得大小不到一个字的类型要作为字来传递。即使没有这些限制，将不到一个字的类型作为字来传递效率也更高。所以 lcc 编译器在生成代码时，它就将所传递的短整型（short）参数和字符参数扩大为整型来传递，对于新风格的函数，lcc 生成代码在函数入口处将这些参数还原为 short 和字符。如果全局字符 ch 作为 f 函数的第二个参数传递，对应的树为：

```
(ARG+I (CVC+I (INDIR+C (ADDRG+P ch))))
```

最下面的 ARG+B 树以传值的方式将结构 a 传递给函数 f。这里提供了 ARG+B 树，因此编译后端可以使用与目标机相关的调用序列。第 16 章就给出了它是如何用在 MIPS 机器上的。如果 wants_argb 等于 0，编译前端就完全实现结构的传值。首先调用者将实参复制到一个局部临时变量中，然后传递该临时变量的地址。被调用者对实参的引用处理成通过新加的间接引用来获取结构实参（函数 idtree 实现）。图 9-3 显示了当 wants_argb 等于 0 时，将 a 传递给函数 f 的 ARG 树。该 RIGHT 树生成代码将 a 赋值给临时变量 t2，然后传递 t2 的地址。

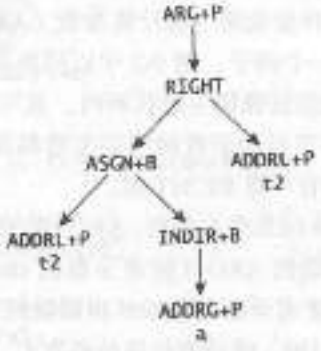


图 9-3 当 wants_argb 等于 0 时，通过传值来传递结构

CALL+B 的右操作数就是调用程序中存放返回值的临时变量的地址，例如图 9-2 中的 t3。当接口标志 wants_callb 等于 1 时，编译后端就必须将该地址传递给调用者。当 wants_callb 等于 0 时，编译前端将该地址作为隐含的第一个参数来传递，并将 CALL+B 改为 CALL+V；这种情况下，编译后端就永远见不到 CALL+B 节点。当函数调用的分析树被转换为编译后端的节点森林时，函数 listnodes 将 CALL+B 转换为 CALL+V。

listnodes 返为调用树时，检查接口标志 left_to_right 的值。如果 left_to_right 等于 1，那么遍历参数子树时，首先访问 ARG 树的右操作数，这就生成了从左至右计算参数的代码。如果 left_to_right 等于 0，就会首先访问左操作数，这样就生成了从右至左计算参数的代码。

函数 posttfix 检查函数表达式的类型，调用函数 call 完成大部分处理工作：

```
⟨calls 143⟩≡                                                    126
  {
     Type ty;
     Coordinate pt;
     p = pointer(p);
     if (isptr(p->type) && isfunc(p->type->type))
        ty = p->type->type;
     else {
        error("found '%t' expected a function\n", p->type);
        ty = func(voidtype, NULL, 1);
     }
     pt = src;
     t = gettok();
     p = call(p, ty, pt);
  }
```

函数 call 使用局部变量来处理上述每个语义问题。变量 n 用来对实参进行计数。args 表示参数树的根节点，而 r 是 RIGHT 树的根节点，该 RIGHT 树保存了参数或含有函数调用的函数表达式。例如在图 9-2 中，r 指向 CALL+I 树。分析完参数后，如果 r 非空，则 r 和 args 一起封装在 RIGHT 树中传递，这里的 RIGHT 树就是图 9-2 中根节点为带阴影的 RIGHT 树的那棵子树。如果参数树中包含一棵 CALL 子树，函数 hascall 返回一个非零值。如果函数 f 计算函数的地址，那么函数 funcname 返回字符串 "a function"，否则 funcname 返回 f 的函数名。

```
⟨enode.c functions⟩≡                                            146
  Tree call(f, fty, src) Tree f; Type fty; Coordinate src; {
     int n = 0;
     Tree args = NULL, r = NULL;
     Type *proto, rty = unqual(freturn(fty));
     Symbol t3 = NULL;
     if (fty->u.f.oldstyle)
        proto = NULL;
     else
        proto = fty->u.f.proto;
     if (hascall(f))
        r = f;
     if (isstruct(rty))
        ⟨initialize for a struct function 144⟩
     if (t != ')')
        for (;;) {
```

```
            ⟨parse one argument 144⟩
            if (t != ',')
               break;
            t = gettok();
         }
      expect(')');
      if (⟨still in a new-style prototype? 144⟩)
         error("insufficient number of arguments to %s\n",
            funcname(f));
      if (r)
         args = tree(RIGHT, voidtype, r, args);
      if (events.calls)
         ⟨plant an event hook for a call⟩
      return calltree(f, rty, args, t3);
   }
```

f是表示该函数的表达式，rty是返回类型，而proto的值分两种情况：如果是旧风格的函数（即使它有原型，参见4.5节），proto等于NULL；如果是新风格的函数，proto保存的是函数原型。非空的proto不断累积每一个实参，这些实参与新风格原型中的形参对应。如果有函数原型，下面的代码就用来测试proto是否指向一个形参类型：

```
⟨still in a new-style prototype? 144⟩=                                144
  proto && *proto && *proto != voidtype
```

处理到函数原型的末端并不等于把实参分析完了，例如，在允许有可变参数的新风格函数中，允许有多余的参数。

如果函数返回一个结构，t3就是生成的用来保存返回值的临时变量：

```
⟨initialize for a struct function 144⟩=                               143
   {
      t3 = temporary(AUTO, unqual(rty), level);
      if (rty->size == 0)
         error("illegal use of incomplete type '%t'\n", rty);
   }
```

t3就是图9-2中所示的临时变量。上面的初始化工作可以在参数分析完成后再进行，但是lcc将它放在参数分析之前处理，以便在调试诊断时显示的源程序坐标就是参数列表的起始位置。

一个实参就是一个赋值表达式：

```
⟨parse one argument 144⟩=                                             144
   Tree q = pointer(expr1(0));
   if (⟨still in a new-style prototype? 144⟩)
      ⟨new-style argument 145⟩
   else
      ⟨old-style argument 145⟩
   if (!IR->wants_argb && isstruct(q->type))
      ⟨pass a structure directly 147⟩
   if (q->type->size == 0)
      q->type = inttype;
   if (hascall(q))
      r = r ? tree(RIGHT, voidtype, r, q) : q;
   args = tree(ARG + widen(q->type), q->type, q, args);
   n++;
```

```
if (Aflag >= 2 && n == 32)
    warning("more than 31 arguments in a call to %s\n",
        funcname(f));
```

代码开始处的 if 语句区分新风格与旧风格的函数类型，并处理对含有可变参数（例如 printf）或有多余参数的新风格的函数的调用。如果函数原型规定了一个可变长度的参数列表（以,... 结束），那么，在原型数组中至少有两种类型，并且最后一种是 voidtype。最后一个显式声明的参数之后的形参所对应的实参的传递方式与旧风格函数传递参数的方式一样。

新风格的参数传递就像把实参赋值给形参。当然，参数是通过 ARG 树传递的，因此实际上并没有进行赋值。但是可以调用函数 assign（该函数对赋值进行类型检查）对参数进行类型检查：

```
⟨new-style argument 145⟩= 144
  {
      Type aty;
      q = value(q);
      aty = assign(*proto, q);
      if (aty)
          q = cast(q, aty);
      else
          error("type error in argument %d to %s; found '%t' _
              expected '%t'\n", n + 1, funcname(f),
           q->type, *proto);
      if ((isint(q->type) || isenum(q->type))
      && q->type->size != inttype->size)
          q = cast(q, promote(q->type));
      ++proto;
  }
```

第二个对函数 cast 的调用，将传递的短整型（short）参数和字符参数提升为整型来传递。

旧风格函数的参数遵循默认的参数提升规则（default argument promotion），包括整数类型提升以及将浮点类型提升为双精度类型。

```
⟨old-style argument 145⟩= 144
  {
      if (!fty->u.f.oldstyle && *proto == NULL)
          error("too many arguments to %s\n", funcname(f));
      q = value(q);
      if (q->type == floattype)
          q = cast(q, doubletype);
      else if (isarray(q->type) || q->type->size == 0)
          error("type error in argument %d to %s; '%t' is _
              illegal\n", n + 1, funcname(f), q->type);
      else
          q = cast(q, promote(q->type));
  }
```

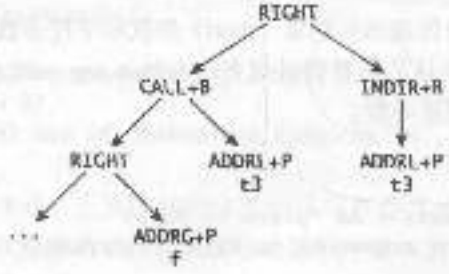
仅仅检查 f.proto 是否非空是不够的，因此代码段首先检查 f.oldstyle。正如 4.5 节所述，旧风格函数可以有原型，但是这些原型不能用来对实参进行类型检查。

实际的 CALL 树是通过函数 calltree 构建的，该函数的参数包括：函数表达式的树、返回类型以及参数树，如果函数返回一个结构，还包括存放返回值的临时变量。所有这些树连接起来形成了图 9-2 中的 CALL+B 树：

```
<enode.c functions>+=                                    143 147
  Tree calltree(f, ty, args, t3)
  Tree f, args; Type ty; Symbol t3; {
     Tree p;

     if (args)
        f = tree(RIGHT, f->type, args, f);
     if (isstruct(ty))
        p = tree(RIGHT, ty,
           tree(CALL+B, ty, f, addrof(idtree(t3))),
           idtree(t3));
     else {
        Type rty = ty;
        if (isenum(ty))
           rty = unqual(ty)->type;
        else if (isptr(ty))
           rty = unsignedtype;
        p = tree(CALL + widen(rty), promote(rty), f, NULL);
        if (isptr(ty) || p->type->size > ty->size)
           p = cast(p, ty);
     }
     return p;
  }
```

整型、无符号类型和指针都需要使用操作符 CALL+I，所以 calltree 函数的大部分工作是保证类型的正确。CALL+B 树永远在 RIGHT 树下，该 RIGHT 树返回保存返回值的临时变量的地址。图 9-2 省略了这棵 RIGHT 树；实际上，由 calltree 构建，并由函数 call 返回的树的开始部分为：



CALL+B 本身没有返回值，它的存在只是为了允许编译后端为这些函数生成与目标机器相关的调用序列。

addrof 是 lvalue 的内部形式，并不需要 INDIR 树（尽管在 calltree 使用 addrof 时有一棵 INDIR 树）。addrof 在 RIGHT，COND 和 ASGN 的操作数以及 INDIR 树中找到计算其参数指定的地址的树。如果有必要，addrof 返回一棵 RIGHT 树，代表原来的树和参数指定的地址。例如，如果 α 是 p 下面的代表操作数的树，用来计算地址，那么 addrof(p) 返回（RIGHT root(p) α）；如果 p 本身计算地址，addrof(p) 返回 p。

结构总是以传值方式进行传递，但如果 wants_argb 等于 0 并且参数是一个结构，那么它必须复制到一个临时变量。如果函数返回一个结构，有一种优化方法可以改进传递该结构的代码。例如：

```
f(f(a, '\n', atoi(str)), '0', 1);
```

由内层的 f 调用返回的 node 被传递给外层的调用。在这类情况下，可以避免实参的副本，因为该实参已经存在于一个临时变量中。这种模式是：

```
(RIGHT
   (CALL+B ... )
   (INDIR+B (ADDRL+P temp))
)
```

这里 temp 是一个临时变量。iscallb 函数负责寻找这种模式：

```
(enode.c functions)+≡                                    146 148
   int iscallb(e) Tree e; {
      return e->op == RIGHT && e->kids[0] && e->kids[1]
         && e->kids[0]->op == CALL+B
         && e->kids[1]->op == INDIR+B
         && isaddrop(e->kids[1]->kids[0]->op)
         && e->kids[1]->kids[0]->u.sym->temporary;
   }

(pass a structure directly 147)=                         144
   if (iscallb(q))
      q = addrof(q);
   else {
      Symbol t1 = temporary(AUTO, unqual(q->type), level);
      q = asgn(t1, q);
      q = tree(RIGHT, ptr(t1->type),
         root(q), lvalue(idtree(t1)));
   }
```

asgn(Symbol t, Tree e) 就是一个赋值语句的内部形式，构建并返回一棵树，该树代表将 e 赋值给符号 t。

9.4 二元操作符

在函数 expr3 中通过全局变量 optree 间接调用二元操作符的语义函数。二元操作符的语义函数接受通用操作符和代表两个操作数的树，返回代表二元表达式的树。表 9-1 列出了这些语义函数以及它们所处理的操作符。在表中，操作符按组排列，每一组操作符的语义都是相似的。

表 9-1 操作符语义函数

函数	操作符
incr	++ -- *= /= %= <<= >>= &= ^= |=
asgntree	=
condtree	? :
andtree	|| &&
bittree	| ^ & %
eqtree	== !=
cmptree	< > <= >=
shtree	<< >>
addtree	+
subtree	-
multree	* /

全局变量 optree 的定义使用了文件 token.h（参见 6.2 节），抽取它的第 5 列，其中存放的就是构建二元表达式的树的函数的名字：

```
⟨enode.c data⟩=
  Tree (*optree[]) ARGS((int, Tree, Tree)) = {
  #define xx(a,b,c,d,e,f,g) e,
  #define yy(a,b,c,d,e,f,g) e,
  #include "token.h"
  };
```

下面以加法二元操作符为例介绍这些语义函数。语义函数必须对操作数进行类型检查，并根据它们的类型来构建相应的树。最简单的情形就是两个操作数都具有算术类型：

```
⟨enode.c functions⟩+=                                    ↑147 149↓
  static Tree addtree(op, l, r) int op; Tree l, r; {
     Type ty = inttype;

     if (isarith(l->type) && isarith(r->type)) {
        ty = binary(l->type, r->type);
        ⟨cast l and r to type ty 148⟩
     } else if (isptr(l->type) && isint(r->type))
        return addtree(ADD, r, l);
     else if ( isptr(r->type) && isint(l->type)
     && !isfunc(r->type->type))
        ⟨build an ADD+P tree 148⟩
     else
        typeerror(op, l, r);
     return simplify(op, ty, l, r);
  }
```

```
⟨cast l and r to type ty 148⟩=                            148 149 150
  l = cast(l, ty);
  r = cast(r, ty);
```

加法操作还可以按任意顺序处理一个指针和一个整型数。上面对函数 addtree 的递归调用交换了参数次序，这样就使得接下来的语句对两种次序都可以处理。编译前端可以将整型操作数置于 ADD dag 的左侧，以帮助编译后端实现对地址模式的加操作。

C 的标准区分了对象指针和函数指针。大多数操作符，比如加法操作符，仅处理对象指针。指数可以与对象指针相加，但是这一操作还隐含了乘以对象大小的乘法操作：

```
⟨build an ADD+P tree 148⟩=                                148
  {
     int n;
     ty = unqual(r->type);
     ⟨n ← *ty's size 149⟩
     l = cast(l, promote(l->type));
     if (n > 1)
        l = multree(MUL, consttree(n, inttype), l);
     return simplify(ADD+P, ty, l, r);
  }
```

```
{n = *ty's size 149}=                                    148
  n = ty->type->size;
  if (n == 0)
     error("unknown size for type '%t'\n", ty->type);
```

函数 consttree 为任意类型（具有整型或无符号型的附加值）的常量构建一棵树：

```
{enode.c functions}+=                                    148 149
  Tree consttree(n, ty) unsigned n; Type ty; {
     Tree p;

     if (isarray(ty))
        ty = atop(ty);
     p = tree(CNST + ttob(ty), ty, NULL, NULL);
     p->u.v.u = n;
     return p;
  }
```

关系比较操作符也只能接受对象指针并返回整数，但是它们对指针操作数的限制比较松。

```
{enode.c functions}+=                                    149 149
  static Tree cmptree(op, l, r) int op; Tree l, r; {
     Type ty;

     if (isarith(l->type) && isarith(r->type)) {
        ty = binary(l->type, r->type);
        {cast l and r to type ty 148}
     } else if (compatible(l->type, r->type)) {
        ty = unsignedtype;
        {cast l and r to type ty 148}
     } else {
        ty = unsignedtype;
        typeerror(op, l, r);
     }
     return simplify(op + ttob(ty), inttype, l, r);
  }
```

这两个指针必须指向兼容的对象类型（或兼容的不完全类型）的限定或未限定形式。换句话说，当对对象进行类型检查时，任何 const 或 volatile 限定符必须被忽略，这就是函数 compatible 所做的工作：

```
{enode.c functions}+=                                    149 150
  static int compatible(ty1, ty2) Type ty1, ty2; {
     return isptr(ty1) && !isfunc(ty1->type)
         && isptr(ty2) && !isfunc(ty2->type)
         && eqtype(unqual(ty1->type), unqual(ty2->type), 0);
  }
```

eqtype 函数的第三个参数为 0，表示它的两个类型参数都是对象类型或者都是不完全类型。

相等比较操作符类似于关系操作符，但是指针操作数的处理更加复杂。这些操作符以及其他操作符对于 void 指针和 null 指针必须区别对待，void 指针指向限定或未限定的 void 类型，而 null 指针是值为 0 的整型常量表达式，或者是将这些值为零的表达式转换为 void * 得到的结果。下面的代码定义了 void 指针和 null 指针：

```
⟨enode.c macros⟩=
  #define isvoidptr(ty) \
     (isptr(ty) && unqual(ty->type) == voidtype)

⟨enode.c functions⟩+≡                                    ⟨149 150⟩
  static int isnullptr(e) Tree e; {
     return (isint(e->type) && generic(e->op) == CNST
           && cast(e, unsignedtype)->u.v.u == 0)
        || (isvoidptr(e->type) && e->op == CNST+P
           && e->u.v.p == NULL);
  }
```

除了算术类型是通过调用函数 cmptree 来处理的以外，函数 eqtree 接受一个指针和一个空指针、一个对象指针和一个 void 指针参数，或者两个指向兼容类型的限定或非限定形式的指针。eqtree 中的第一条 if 语句就是对左、右操作数的这 3 种组合进行条件测试，该函数的递归调用会在适当的时候对右、左操作数组合进行重复测试。

```
⟨enode.c functions⟩+≡                                    ⟨150 151⟩
  Tree eqtree(op, l, r) int op; Tree l, r; {
     Type xty = l->type, yty = r->type;

     if (isptr(xty) && isnullptr(r)
     ||  isptr(xty) && !isfunc(xty->type) && isvoidptr(yty)
     ||  ⟨xty and yty point to compatible types 150⟩) {
        Type ty = unsignedtype;
        ⟨cast l and r to type ty 148⟩
        return simplify(op + U, inttype, l, r);
     }
     if (isptr(yty) && isnullptr(l)
     ||  isptr(yty) && !isfunc(yty->type) && isvoidptr(xty))
        return eqtree(op, r, l);
     return cmptree(op, l, r);
  }

⟨xty and yty point to compatible types 150⟩=                    150 151 155
  (isptr(xty) && isptr(yty)
  && eqtype(unqual(xty->type), unqual(yty->type), 1))
```

如果 eqtype 的第三个参数等于 1，则表示 eqtype 允许前两个参数可以是兼容对象或不完全类型的任意组合。对于声明：

```
int (*p)[10], (*q)[];
```

eqtype 函数的第三个参数允许 p==q 而不允许 p<q。

9.5 赋值操作

赋值表达式、函数参数、返回语句或者初始化代码的合法性都依赖于将右值赋值给左值所表示的地址的合法性。assign(xty, e) 对任何赋值都进行必要的类型检查，它检查将树 e 赋值给一个保存 xty 类型值的左值的合法性，如果该赋值是合法的就返回 xty 类型，否则就返回 null。在进行赋值之前，必须将 e 转类成返回值 xty 类型。

```
⟨enode.c functions⟩+≡                                          ↑150 152↓
  Type assign(xty, e) Type xty; Tree e; {
     Type yty = unqual(e->type);

     xty = unqual(xty);
     if (isenum(xty))
        xty = xty->type;
     if (xty->size == 0 || yty->size == 0)
        return NULL;
     ⟨assign 151⟩
  }
```

C 语言标准针对赋值规定了 5 种约束，assign 的函数体对这 5 种约束进行测试。前两种约束是针对算术和结构类型的赋值：

```
⟨assign 151⟩≡                                                  ↓151  151
  if ( isarith(xty) && isarith(yty)
  ||  isstruct(xty) && xty == yty)
     return xty;
```

其他 3 种约束都涉及指针。空指针可以赋值给任何指针：

```
⟨assign 151⟩+≡                                                 ↑151 151↓  151
  if (isptr(xty) && isnullptr(e))
     return xty;
```

只要左操作数指向的类型有右操作数指向的类型所具有的所有限定符，任何指针就都可以赋值给 void 指针，反之亦然。

```
⟨assign 151⟩+≡                                                 ↑151 151↓  151
  if ((isvoidptr(xty) && isptr(yty)
    || isptr(xty)     && isvoidptr(yty))
  && ⟨*xty has all of *yty's qualifiers 151⟩)
     return xty;

⟨*xty has all of *yty's qualifiers 151⟩≡                                  151
  (  (isconst(xty->type)    || !isconst(yty->type))
  && (isvolatile(xty->type) || !isvolatile(yty->type)))
```

一个指针可以赋值给另一个指针，仅当它们指向兼容的类型，并且左值具有右值的所有限定符，像上面一样。

```
⟨assign 151⟩+≡                                                 ↑151 151↓  151
  if (⟨xty and yty point to compatible types 150⟩
  &&  ⟨*xty has all of *yty's qualifiers 151⟩)
     return xty;
```

最后，如果上面的情形都不满足，那么该赋值操作就是错误的，assign 函数返回一个空指针。

```
⟨assign 151⟩+≡                                                 ↑151  151
  return NULL;
```

函数 assign 应用于函数 asgntree，为赋值操作生成树：

```
⟨enode.c functions⟩+≡                                          151 154
  Tree asgntree(op, l, r) int op; Tree l, r; {
     Type aty, ty;

     r = pointer(r);
     ty = assign(l->type, r);
     if (ty)
        r = cast(r, ty);
     else {
        typeerror(ASGN, l, r);
        if (r->type == voidtype)
           r = retype(r, inttype);
        ty = r->type;
     }
     if (l->op != FIELD)
        l = lvalue(l);
     ⟨asgntree 152⟩
     return tree(op + (isunsigned(ty) ? I : ttob(ty)),
        ty, l, r);
  }
```

当赋值非法时，assign 函数返回空指针，asgntree 函数必须为赋值的结果选择一种类型。如果右操作数的类型不是 void 类型，结果类型就使用该类型；否则，结果类型为整型。这段代码也展示了发生语义错误时，需要从错误中恢复以使编译继续下去。

下面 <asgntree> 中列出的代码段是 asgntree 函数的主体，它主要进行以下工作：检查是否有试图改变常量区域的值的代码，改变位域赋值的整数右值使得它们符合标准规范，以及转换一些结构赋值语句以生成更好的代码。

如果对象类型是用 const 限定的或者结构类型是 const 限定的，则其左值就表示常量区域。结构类型是否具有 const 限定符，可以通过它的 u.s.cfields 域来判断。

```
⟨asgntree 152⟩≡                                                153 152
  aty = l->type;
  if (isptr(aty))
     aty = unqual(aty)->type;
  if ( isconst(aty)
  ||  isstruct(aty) && unqual(aty)->u.sym->u.s.cfields)
     if (isaddrop(l->op)
     && !l->u.sym->computed && !l->u.sym->generated)
        error("assignment to const identifier '%s'\n",
           l->u.sym->name);
     else
        error("assignment to const location\n");
```

aty 被设置为存放在左值表示的地址中的值类型。如果 aty 具有 const 限定符或者它是一个具有一个或多个用 const 限定了的域的结构类型，那么该赋值不合法。调试信息对那些在源程序中没有名字的左值单独处理。

赋值操作的结果就是其左操作数的值，结果类型是左操作数的限定形式。函数 asgntree 开始的类型转换就是将 r 设置为正确的树，将 ty 设置为 r 对应的类型，这样 ty 就表示结果，所以 ASGN 的结果就是它的右操作数。遗憾的是，这种处理对于位域并不起作用。位域赋值的结果应是赋值后从位域中提取的值，这个值可能与 r 表示的值并不相同。因此，如果位域所占的空间不到一个完整的无符号数所占的空间，那么处理对该位域的赋值时，asgntree 必须将 r 改变为仅计

算位域值的树。

```
(asgntree 152)+≡                                    152 153  152
  if (l->op == FIELD) {
     int n = 8*l->u.field->type->size - fieldsize(l->u.field);
     if (n > 0 && isunsigned(l->u.field->type))
        r = bittree(BAND, r,
           consttree(fieldmask(l->u.field), unsignedtype));
     else if (n > 0) {
        if (r->op == CNST+I)
           r = consttree(r->u.v.i<<n, inttype);
        else
           r = shtree(LSH, r, consttree(n, inttype));
        r = shtree(RSH, r, consttree(n, inttype));
     }
  }
```

如果位域是无符号数，那么结果就是 r 去掉多余的有效位后得到的值。如果位域是有符号数，并且有 m 位，则第 m-1 位就是位域的符号位，符号位用来对该值进行符号扩展，符号扩展可以通过算术移位实现：首先左移 r，将第 m 位移到符号位，然后再右移相同的位数，并在右移过程中使用符号位填充。例如，图 9-4 给出了下面代码块中两条赋值语句的树：

```
struct { int a:3; unsigned b:3; } x;
x.a = e;
x.b = e;
```

在赋值 x.a=e 中，r 赋值为一棵树，该树表示通过移位对 e 最右三位进行符号扩展；对于 x.b=e，r 赋值为一棵表示将 7 和 e 相与的树。如果 r 是一个常量，左移操作形式执行，以避免进行常量折叠时导致溢出。

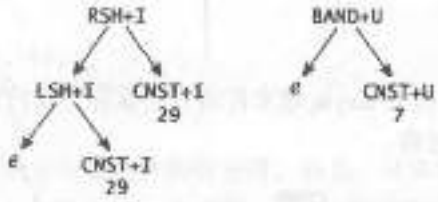


图 9-4 表示 x.a=e 和 x.b=e 结果的树

一般来说，编译后端会为结构赋值生成块移动（block move）操作。为这些赋值生成好的代码主要取决于编译后端，但是，前端也可以减少这种无用的代码块移动的数目，它类似于 9.3 节中描述的通过 call 函数对结构参数进行的优化。在 x=f() 中，f 返回一个结构，调用程序将生成一个临时变量以保存 f 的返回值，调用返回后将临时变量赋值给 x。图 9-5 的左图给出了结果树，其中 x 代表 x 对应的树。在临时变量的位置直接使用 x 代替，可以避免复制操作。

当 x 是一个直接地址，并且有一个保存 f 返回值的临时变量时，这种改进才可以实现：

```
(asgntree 152)+≡                                    153  152
  if (isstruct(ty) && isaddrop(l->op) && iscallb(r))
     return tree(RIGHT, ty,
        tree(CALL+B, ty, r->kids[0]->kids[0], l),
        idtree(l->u.sym));
```

图 9-5 的右图给出了这种转换返回的树。

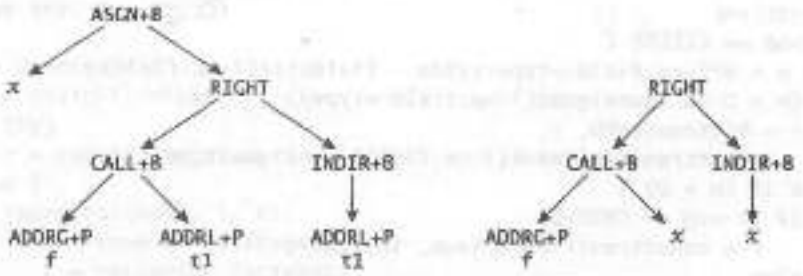


图 9-5 x=f() 的分析树

9.6 条件操作

条件表达式复杂的语义综合了比较操作符、二元操作符、赋值和类型转换四者的语义。COND 是唯一一个处理 3 个操作数的操作符：表达式 e?l:r 生成如图 9-6 所示的树，该树是由函数 condtree 构建的：

```
⟨enode.c functions⟩+≡                                          152▲
  Tree condtree(e, l, r) Tree e, l, r; {
     Symbol t1;
     Type ty, xty = l->type, yty = r->type;
     Tree p;

     ⟨condtree 155⟩
     p = tree(COND, ty, cond(e),
        tree(RIGHT, ty, root(l), root(r)));
     p->u.sym = t1;
     return p;
  }
```

其中临时变量 t1 通过 COND 树的 u.sym 域来传递，它保存了运行时条件表达式的结果。如果结果是 void 类型的，t1 就会被忽略。

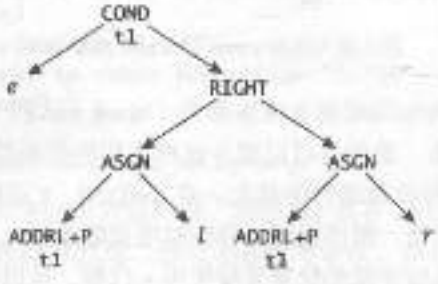


图 9-6 e?l:r 的树

上面的代码调用 cond(e) 对第一个操作数进行类型检查，第一个操作数必须是一个标量类型。对于第二个和第三个操作数的类型有 6 种合法的组合。最简单的 3 种组合就是：二者都是算术类型、二者是兼容的结构类型、二者都是 void 类型。所有这 3 种组合都可以被下面两条 if 语句所覆盖：

```
⟨condtree 155⟩=                                          155  154
  if (isarith(xty) && isarith(yty))
     ty = binary(xty, yty);
  else if (eqtype(xty, yty, 1))
     ty = unqual(xty);
```

第一个 if 语句处理算术类型，第二个 if 语句处理结构类型和 void 类型。

剩余的 3 种情况都涉及指针类型。如果操作数中有一个是空指针而另一个是指针，则结果类型就是那个非空指针的类型：

```
⟨condtree 155⟩+=                                         155 155  154
  else if (isptr(xty)   && isnullptr(r))
     ty = xty;
  else if (isnullptr(l) && isptr(yty))
     ty = yty;
```

如果一个操作数是 void 指针，而另一个是指向对象或不完全类型的指针，则结果类型就是 void 指针：

```
⟨condtree 155⟩+=                                         155 155  154
  else if (isptr(xty) && !isfunc(xty->type) && isvoidptr(yty)
  ||       isptr(yty) && !isfunc(yty->type) && isvoidptr(xty))
     ty = voidptype;
```

如果两个操作数都是指向兼容类型的限定或未限定形式的指针，则它们中任何一个都可以作为结果类型：

```
⟨condtree 155⟩+=                                         155 155  154
  else if (⟨xty and yty point to compatible types 150⟩)
     ty = xty;
  else {
     typeerror(COND, l, r);
     return consttree(0, inttype);
  }
```

上述类型检查代码不考虑限定类型指针的限定符。但是，结果指针类型必须包含两个操作数引用类型的所有限定符；所以，如果 ty 是一个指针，则 lcc 使用相应的限定符来重新构建 ty：

```
⟨condtree 155⟩+=                                         155 156  154
  if (isptr(ty)) {
     ty = unqual(unqual(ty)->type);
     if (isptr(xty) && isconst(unqual(xty)->type)
     ||  isptr(yty) && isconst(unqual(yty)->type))
        ty = qual(CONST, ty);
     if (isptr(xty) && isvolatile(unqual(xty)->type)
     ||  isptr(yty) && isvolatile(unqual(yty)->type))
        ty = qual(VOLATILE, ty);
     ty = ptr(ty);
  }
```

如果条件式 e 是一个常量，那么条件表达式的结果就是其他两个操作数中的一个：

```
⟨condtree 155⟩+≡                                    155 156 154
  if (e->op == CNST+D || e->op == CNST+F) {
     e = cast(e, doubletype);
     return cast(e->u.v.d != 0.0 ? l : r, ty);
  }
  if (generic(e->op) == CNST) {
     e = cast(e, unsignedtype);
     return cast(e->u.v.u ? l : r, ty);
  }
```

这种常量折叠并不只是一种优化技术，它还是必需的，这是因为条件表达式可以用于需要常量表达式的上下文中。

最后，如果结果类型不是 void 类型，就生成临时变量 t1，并且将 l 和 r 转变为对该临时变量进行的赋值操作：

```
⟨condtree 155⟩+≡                                    156 154
  if (ty != voidtype && ty->size > 0) {
     t1 = temporary(REGISTER, unqual(ty), level);
     l = asgn(t1, l);
     r = asgn(t1, r);
  } else
     t1 = NULL;
```

9.7 常量折叠

任何需要常量的地方都允许使用常量表达式。例如，数组大小、case 标号、位域的宽度以及初始化操作。

constant-expression: conditional-expression

常量表达式可以用下面的代码分析：

```
⟨simp.c functions⟩≡                                 157
  Tree constexpr(tok) int tok; {
     Tree p;

     needconst++;
     p = expr1(tok);
     needconst--;
     return p;
  }

⟨simp.c data⟩≡
  int needconst;
```

函数 expr1 分析赋值表达式。从技术上来讲，函数 constexpr 应当调用分析条件表达式的函数 expr2，但是合法的赋值操作不是常量，会导致语义错误。因为 expr1 完整地处理了一个赋值操作，并且避免了对同一语法错误的多次检查，因此调用 expr1 来处理语法错误通常效果更好。如果 constexpr 返回的树不是一棵 CNST 树，但该返回树却用于需要常量表达式的上下文中，那么 constexpr 的调用者就会报告一个错误。分析整型常量表达式的函数 intexpr 就是这样的例子：

```
⟨simp.c functions⟩+≡                                    156 157
  int intexpr(tok, n) int tok, n; {
     Tree p = constexpr(tok);

     needconst++;
     if (generic(p->op) == CNST && isint(p->type))
        n = cast(p, inttype)->u.v.i;
     else
        error("integer expression must be constant\n");
     needconst--;
     return n;
  }
```

全局变量 needconst 控制常量折叠（由函数 simplify 处理）。如果 needconst 不等于 0，simplify 函数就会对溢出的常量表达式给出警告信息并进行常量折叠，否则不执行常量折叠。

常量折叠不只是一个简单的优化。C 语言标准在对一些语言成分的定义中规定，常量表达式的值必须在编译时计算，因此需要常量折叠。数组大小和位域的宽度就是这样的例子。与函数 tree 相同，函数 simplify 也返回其参数指明的树：

```
⟨simp.c functions⟩+≡                                    157 159
  Tree simplify(op, ty, l, r) int op; Type ty; Tree l, r; {
     int n;
     Tree p;

     if (optype(op) == 0)
        op += ttob(ty);
     switch (op) {
        ⟨simplify cases 157⟩
     }
     return tree(op, ty, l, r);
  }
```

simplify 完成函数 tree 未处理的 3 类操作：如果传递的参数是一个通用操作符，它就形成与类型相关的操作符；当两个操作数都是常量时，它就计算操作的结果；在它构建所需的树时，会把一些树转换为更简单的树，以生成更优的代码。

simplify 中 switch 语句的每一个 case 分支分别处理一种与类型相关的操作符。如果两个操作数都是常量，该代码就为结果值构建并返回一棵 CNST 树；否则，它就跳转到 switch 语句的结尾，构建并返回相应的树。检查常量操作数和构建结果 CNST 树的代码对于每一个 case 语句几乎都是相同的；只有类型后缀、Value 域名、操作符以及返回类型不同，所以这部分代码就被包装为一组宏。最典型的一种情况就是无符号数的加法操作：

```
⟨simplify cases 157⟩≡                                   158 ← 157
  case ADD+U:
     foldcnst(U,u,+,unsignedtype);
     commute(r,l);
     break;
```

这个 case 实现了如下转换：

$$\texttt{(ADD+U (CNST+U } c_1\texttt{) (CNST+U } c_2\texttt{))} \Rightarrow \texttt{(CNST+U } c_1 + c_2\texttt{)}$$

函数 foldcnst 检查两个操作数是否都是 CNST+U 树，如果是，它就返回一棵新的 CNST+U 树，且新建的树的 u.v.u 域就是 l->r.v.u 与 r->r.v.u 的和：

```
⟨simp.c macros⟩=                                                    158
  #define foldcnst(TYPE,VAR,OP,RTYPE) \
     if (l->op == CNST+TYPE && r->op == CNST+TYPE) {\
        p = tree(CNST+ttob(RTYPE), RTYPE, NULL, NULL);\
        p->u.v.VAR = l->u.v.VAR OP r->u.v.VAR;\
        return p; }
```

对于可交换的操作，函数 commute 确保如果它的参数中有一个是常量，则该常量就作为 commute 函数的第一个参数。这种转换减少了编译后端必须进行的分析操作，允许编译后端依赖出现在特定地方的可交换操作的常量操作数。

```
⟨simp.c macros⟩+=                                                158 158
  #define commute(L,R) \
     if (generic(R->op) == CNST && generic(L->op) != CNST) {\
        Tree t = L; L = R; R = t; }
```

必要时，commute 交换它的两个参数，使得 L 指向常量操作数。例如，对于上述的 ADD+U，如果有一个操作数是常量，commute(r,l) 就保证了 r 引用该常量操作数。在下文可以看到，这种转换也使得 simplify 的一些转换更加简单。

C 语言标准规定对无符号数的操作没有溢出，因此无符号数的加法很简单。然而，有符号的操作必须处理溢出。例如，如果 ADD+I 的操作数是常量，而且它们的和发生溢出，那么该表达式就不是一个常量表达式，除非其上下文要求它是常量的。有符号操作符的处理需要使用函数 xfoldcnst，该函数与 foldcnst 类似，只是 xfoldcnst 要对溢出进行检查。

```
⟨simplify cases 157⟩+=                                           157 159 157
  case ADD+I:
     xfoldcnst(I,i,+,inttype,add,INT_MIN,INT_MAX,needconst);
     commute(r,l);
     break;
```

仅当 c_1+c_2 没有溢出或者 needconst 等于 1 时，才实现如下转换：

```
(ADD+I (CNST+I c1) (CNST+I c2))  →  (CNST+I c1 + c2)
```

xfoldcnst 还有 4 个其他参数：一个相关的函数，结果允许的最小和最大值，以及一个标记，当需要常量时，该标记为非零值。

```
⟨simp.c macros⟩+=                                                158 159
  #define xfoldcnst(TYPE,VAR,OP,RTYPE,FUNC,MIN,MAX,needconst)\
     if (l->op == CNST+TYPE && r->op == CNST+TYPE\
     && FUNC((double)l->u.v.VAR,(double)r->u.v.VAR,\
        (double)MIN,(double)MAX, needconst)) {\
        p = tree(CNST+ttob(RTYPE), RTYPE, NULL, NULL);\
        p->u.v.VAR = l->u.v.VAR OP r->u.v.VAR;\
        return p; }
```

lcc 编译器假定双精度类型的有效位足够表示所有整数类型，因此该相关函数的实参被转换为双精度类型。检查常量操作数以及构建结果 CNST 树的代码与函数 foldcnst 中的代码相同，但是需要调用指定函数来检查操作的有效性。如果操作发生溢出，函数返回 0，否则返回 1。函数的参数包括操作数的值、最小和最大值以及一个是否需要常量的标记。除了标记外，其他的都被转换

为双精度类型。对于整数加法操作，溢出的测试很简单：如果 x+y 小于 INT_MIN 或大于 INT_MAX，就会发生溢出，这里 INT_MIN 和 INT_MAX 表示的是有符号整数的最小或最大 ANSI 值。函数 add 可以处理所有的类型，由于加法可能产生溢出，add 并不真正计算 x+y，相反，它只测试发生溢出的条件：

```
(simp.c functions)+≡                                    157 163
  static int add(x, y, min, max, needconst)
  double x, y, min, max; int needconst; {
     int cond = x == 0 || y == 0
     || x < 0 && y < 0 && x >= min - y
     || x < 0 && y > 0
     || x > 0 && y < 0
     || x > 0 && y > 0 && x <= max - y;
     if (!cond && needconst) {
        warning("overflow in constant expression\n");
        cond = 1;
     }
     return cond;
  }
```

在给出警告信息后，如果 needconst 非零，那么 needconst 强制 add 函数返回 1。函数 sub、mul 和 div 也都类似。

类型转换可以分为必须对溢出进行检查的转换以及可以忽略溢出的转换。从较小类型向较大类型转换、无符号类型之间的转换、无符号类型与指针类型之间的转换以及从整数向无符号类型的转换都可以忽略溢出。下面的转换包含了这 4 种情形的处理，它们实现了类似于下面的转换：

```
(CVC+I (CNST+C c))  ⇒  (CNST+I c')
```

其中，c' 就是 c 可能的符号扩展值，对于无符号类型之间的转换，c'=c。

```
(simplify cases 157)+≡                                  158 160  157
  case CVC+I:
     cvtcnst(C,inttype, p->u.v.i =
        (l->u.v.sc&0200 ? (~0<<8) : 0)|(l->u.v.sc&0377));
     break;
  case CVU+S:
     cvtcnst(U,unsignedshort,p->u.v.us = l->u.v.u); break;
  case CVP+U:
     cvtcnst(P,unsignedtype, p->u.v.u  = (unsigned)l->u.v.p);
     break;
  case CVI+U:
     cvtcnst(I,unsignedtype, p->u.v.u  = l->u.v.i); break;

(simp.c macros)+≡                                       158 160
  #define cvtcnst(FTYPE,TTYPE,EXPR) \
     if (l->op == CNST+FTYPE) {\
        p = tree(CNST+ttob(TTYPE), TTYPE, NULL, NULL);\
        EXPR;\
        return p; }
```

处理CVC+I的情况时，赋值操作必须首先显式地将字符操作数用符号位进行符号扩展。由于编译器本身可能通过其他C编译器编译，所以不能假设字符已经经过了符号扩展。在一些情况下，也可以通过下面的表达式替换传递给cvtconst的赋值操作：

```
((int)l->u.v.sc<<(8*sizeof(int) - 8))>>(8*sizeof(int) - 8)
```

然而>>操作是否能够复制符号位还依赖于编译lcc的编译器。

从大类型向小类型的4种转换必须检查溢出。

```
(CVI+C (CNST+I c))  ⇒  (CNST+C c)
```

如果c可以用小类型表示或者needconst等于1，就实现下面的转换：

```
(simplify cases 157)+≡                                    159 160  157
  case CVI+C:
     xcvtcnst(I, chartype,l->u.v.i,SCHAR_MIN,SCHAR_MAX,
        p->u.v.sc = l->u.v.i); break;
  case CVD+F:
     xcvtcnst(D,floattype,l->u.v.d, -FLT_MAX,FLT_MAX,
        p->u.v.f  = l->u.v.d); break;
  case CVD+I:
     xcvtcnst(D,  inttype,l->u.v.d,  INT_MIN,INT_MAX,
        p->u.v.i  = l->u.v.d); break;
  case CVI+S:
     xcvtcnst(I,shorttype,l->u.v.i, SHRT_MIN,SHRT_MAX,
        p->u.v.ss = l->u.v.i); break;

(simp.c macros)+≡                                         159 160
  #define xcvtcnst(FTYPE,TTYPE,VAR,MIN,MAX,EXPR) \
     if (l->op == CNST+FTYPE) {\
        if (needconst && (VAR < MIN || VAR > MAX))\
           warning("overflow in constant expression\n");\
        if (needconst || VAR >= MIN && VAR <= MAX) {\
           p = tree(CNST+ttob(TTYPE), TTYPE, NULL, NULL);\
           EXPR;\
           return p; } }
```

除了计算常量表达式，simplify还对某些操作树进行转换以生成更优的代码，包括去掉标识符和其他简单的情况。例如：

```
(simplify cases 157)+≡                                    160 161  157
  case BAND+U:
     foldcnst(U,u,&,unsignedtype);
     commute(r,l);
     identity(r,l,U,u,(~(unsigned)0));
     if (r->op == CNST+U && r->u.v.u == 0)
        return tree(RIGHT, unsignedtype, root(l),
           consttree(0, unsignedtype));
     break;

(simp.c macros)+≡                                         160 161
  #define identity(X,Y,TYPE,VAR,VAL) \
```

```
if (X->op == CNST+TYPE && X->u.v.VAR == VAL)\
   return Y
```

函数 identity 和接下来的 if 语句来实现下面的转换：

```
(BAND+U e (CNST+U -0))  ⇒  e
(BAND+U e (CNST+U  0))  ⇒  (e, (CNST+U 0))
```

在第二种情况中，e 可能产生副作用，不能被删除。从代码中可以看到，如果只有 r 是一个常量，使用 commute(r,l) 进行检查将很有必要。

对于某些操作符 simplify 还实现了强度削弱（strength reduction），即用计算结果相同、代价更低的操作符代替原操作符。例如，一个无符号数乘以 2 的 k 次幂可以用左移操作来代替：

```
(MUL+U (CNST+U 2^k) e)  ⇒  (LSH+U e (CNST+I k))
```

这段代码也使用函数 foldcnst 对常量操作数进行检查。

```
{simplify cases 157}+=                                    160 161  157
  case MUL+U:
     commute(l,r);
     if (l->op == CNST+U && (n = ispow2(l->u.v.u)) != 0)
        return simplify(LSH+U, unsignedtype, r,
           consttree(n, inttype));
     foldcnst(U,u,*,unsignedtype);
     break;
```

k>0 时，如果 u 等于 2^k，那么 ispow2(u) 就返回 k。

位域经常使用诸如 p->x!=0 的表达式进行检查，该表达式将生成以 FIELD 和 CNST 树为操作数的 NE 树。位域的抽取通常涉及移位和掩码操作，因此对位域的检查可以被更简单的代码替换：首先获得包含位域的字，然后和相应的屏蔽位进行与操作，检查与操作的结果：

```
{simplify cases 157}+=                                    161 162  157
  case NE+I:
     cfoldcnst(I,i,!=,inttype);
     commute(r,l);
     zerofield(NE,I,i);
     break;
{simp.c macros}+=                                              160 162
  #define zerofield(OP,TYPE,VAR) \
     if (l->op == FIELD\
     &&  r->op == CNST+TYPE && r->u.v.VAR == 0)\
        return eqtree(OP, bittree(BAND, l->kids[0],\
           consttree(\
              fieldmask(l->u.field)<<fieldright(l->u.field),\
              unsignedtype)), r);
```

上面针对 NE+I 的代码实现了下面的转换：

```
(NE+I (FIELD e) (CNST+I 0))  ⇒
    (NE+I (BAND+U (e (CNST+U M))) (CNST+I 0))
```

其中，M 是位域左移 m 位后得到的长度为 s 的掩码，s 是位域的长度，该位域与存放位域的无符号数或整数的最低有效位之间隔 m 位。cfoldcnst 是 foldcnst 的另一个版本，针对关系操作符。

```
⟨simp.c macros⟩+≡                                    161 162
  #define cfoldcnst(TYPE,VAR,OP,RTYPE) \
     if (l->op == CNST+TYPE && r->op == CNST+TYPE) {\
        p = tree(CNST+ttob(RTYPE), RTYPE, NULL, NULL);\
        p->u.v.i = l->u.v.VAR OP r->u.v.VAR;\
        return p; }
```

在函数 simplify 中，指针加法是最有趣但也是最复杂的一种情况，需要实现许多转换以便生成更优的代码。生成高效的寻址是生成高效代码的关键，所以指针加法的转化对于所有目标机器都有价值。最简单的情况就是处理常量和标识符：

```
⟨simplify cases 157⟩+≡                               161 157
  case ADD+P:
     foldaddp(l,r,I,i);
     foldaddp(l,r,U,u);
     foldaddp(r,l,I,i);
     foldaddp(r,l,U,u);
     commute(r,l);
     identity(r,retype(l,ty),I,i,0);
     identity(r,retype(l,ty),U,u,0);
     ⟨ADD+P transformations 162⟩
     break;

⟨simp.c macros⟩+≡                                    162
  #define foldaddp(L,R,RTYPE,VAR) \
     if (L->op == CNST+P && R->op == CNST+RTYPE) {\
        p = tree(CNST+P, ty, NULL, NULL);\
        p->u.v.p = (char *)L->u.v.p + R->u.v.VAR;\
        return p; }
```

由于 ADD+P 的操作数是不对称的——一个是指针而另一个是整数或者无符号数，所以需要用 4 种方式调用函数 foldaddp。对 foldaddp 的这种用法实现了下面的转换：

```
(ADD+P (CNST+P c1) (CNST+I c2))   ⇒   (CNST+P c1 + c2)
```

使用函数 identity 实现下面的转换：

```
(ADD+P e (CNST+I 0))   ⇒   e
```

还实现了类似的无符号常量的转换。

针对 ADD+P 树的其余转换或者是生成更简单、更优的树，或者是为其他转换进行准备。转换

```
⟨ADD+P transformations 162⟩≡                         164 162
  if (isaddrop(l->op)
  && (r->op == CNST+I || r->op == CNST+U))
     return addrtree(l, cast(r, inttype)->u.v.i, ty);
```

去掉了用常量标识的已知地址的下标寻址，例如数组引用 a[5] 和域引用 x·name。这些表达式生成形如

```
(ADD+P n (CNST+x c))
```

的树。其中，n 表示一个标识符地址的树，x 等于 U 或 I，而 c 是一个常量。该树被转换为 n′，n′ 是被绑定到寻址目标位置的标识符的树。函数 addrtree 创建了一个新的标识符，并为它的地址构建树。新创建的标识符的地址为 l 加上一个常量偏移量。

```
⟨simp.c functions⟩+≡                                          159
  static Tree addrtree(e, n, ty) Tree e; int n; Type ty; {
     Symbol p = e->u.sym, q;

     NEW0(q, FUNC);
     q->name = stringd(genlabel(1));
     q->sclass = p->sclass;
     q->scope = p->scope;
     q->type = ty;
     q->temporary = p->temporary;
     q->generated = p->generated;
     q->addressed = p->addressed;
     q->computed = 1;
     q->defined = 1;
     q->ref = 1;
     ⟨announce q 163⟩
     e = tree(e->op, ty, NULL, NULL);
     e->u.sym = q;
     return e;
  }

⟨symbol flags 37⟩+≡                                    137 228 28
  unsigned computed:1;
```

与其他标识符一样，编译前端必须将这个新的标识符通知编译后端。因为它的地址是基于另外一个标识符 p 的地址的，前端通过调用接口函数 address 通知后端。新的标识符的 computed 标志表示它是一个基于其他符号的符号。但是有一个先后问题：p 必须在 q 之前通知，但是如果 p 是一个局部变量或者参数，则此时它还未通过 local 或者 function 传递到编译后端。因此，addrtree 仅对全局变量和静态变量调用 address，而对于局部变量和参数延迟处理：

```
⟨announce q 163⟩≡                                             163
  if (p->scope  == GLOBAL
  ||  p->sclass == STATIC || p->sclass == EXTERN) {
     if (p->sclass == AUTO)
        q->sclass = STATIC;
     (*IR->address)(q, p, n);
  } else {
     Code cp;
     addlocal(p);
     cp = code(Address);
     cp->u.addr.sym = q;
     cp->u.addr.base = p;
     cp->u.addr.offset = n;
  }
```

代码表人口 Address 将在 10.1 节介绍。对于全局变量和静态变量 lcc 编译器不会推迟调用 address，因为像 &a[5] 一类的表达式是常量，而且可以出现在初始化中。

接下来的转换可改善诸如 b[i].name 的表达式，生成一棵形如（ADD+P(ADD+P i n)(CNST+x c)）的树。而其中，i 是一个整数表达式的树，n 和 c 的定义已经在上文中给出了。该树可以转换

为（ADD+P i (ADD+P n (CNST+x c))），内层的 ADD+P 树可以通过上面的转换变为（ADD+P i n'），以简化寻址。

```
⟨ADD+P transformations 162⟩+≡                                  162 164  162
  if (l->op == ADD+P && isaddrop(l->kids[1]->op)
  && (r->op == CNST+I || r->op == CNST+U))
return simplify(ADD+P, ty, l->kids[0],
   addrtree(l->kids[1], cast(r, inttype)->u.v.i, ty));
```

从技术上来讲，仅当 (i+n)+c 等于 i+(n+c) 时，这种转换才是安全的。而这要在运行时才能确定，但是 C 语言标准允许在编译时进行这种计算的重排。

类似地，树（ADD+P(ADD+I i(CNST+x c))n）可被转换为（ADD+P i n'）；如果 SUB+I 出现在 ADD+I 的位置上，这种转换也适用：

```
⟨ADD+P transformations 162⟩+≡                                  164 164  162
  if ((l->op == ADD+I || l->op == SUB+I)
  && l->kids[1]->op == CNST+I && isaddrop(r->op))
     return simplify(ADD+P, ty, l->kids[0],
        simplify(generic(l->op)+P, ty, r, l->kids[1]));
```

接下来的分支合并常量并实现了转换：

```
(ADD+P (ADD+P x (CNST c1)) (CNST c2))          ⇒
   (ADD+P x (CNST c1 + c2))
(ADD+P (ADD+I x (CNST c1)) (ADD+P y (CNST c2)))  ⇒
   (ADD+P x (ADD+P y (CNST c1 + c2)))
```

当 x 和 y 是标识符树时，这些转换会触发其他转换。

```
⟨ADD+P transformations 162⟩+≡                                  164 164  162
  if (l->op == ADD+P && generic(l->kids[1]->op) == CNST
  && generic(r->op) == CNST)
     return simplify(ADD+P, ty, l->kids[0],
        (*optree['+'])(ADD, l->kids[1], r));
  if (l->op == ADD+I && generic(l->kids[1]->op) == CNST
  &&  r->op == ADD+P && generic(r->kids[1]->op) == CNST)
     return simplify(ADD+P, ty, l->kids[0],
        simplify(ADD+P, ty, r->kids[0],
        (*optree['+'])(ADD, l->kids[1], r->kids[1])));
```

最后一个转换处理 RIGHT 树，对其操作数使用 ADD+P 转换。

```
⟨ADD+P transformations 162⟩+≡                                  164  162
  if (l->op == RIGHT && l->kids[1])
     return tree(RIGHT, ty, l->kids[0],
        simplify(ADD+P, ty, l->kids[1], r));
  else if (l->op == RIGHT && l->kids[0])
     return tree(RIGHT, ty,
        simplify(ADD+P, ty, l->kids[0], r), NULL);
```

这些测试实现了转换：

```
(ADD+P (RIGHT x y) e)  ⇒  (RIGHT x (ADD+P y e))
(ADD+P (RIGHT x) e)    ⇒  (RIGHT (ADD+P x e))
```

第一个 if 语句是对 f(), x 一类的表达式形成的树进行测试；该调用返回一个临时变量，所以如果要对临时变量的域进行引用，上面第一个 ADD+P 转换的价值就体现出来了。第二个 if 语句是对作为转换结果的 RIGHT 树中的表达式进行测试。

表 9-2 列出了 <simplify cases> 中其余的转换。

表 9-2 其他的 simplify 转换

```
(AND+I (CNST+I 0) e)                    ⇒  (CNST+I 0)
(AND+I (CNST+I 1) e)                    ⇒  e
(OR+I (CNST+I 0) e)                     ⇒  e
(OR+I (CNST+I c) e), c ≠ 0              ⇒  (CNST+I 1)
(BCOM+U (BCOM+U e))                     ⇒  e
(BOR+U (CNST+U 0) e)                    ⇒  e
(BXOR+U (CNST+U 0) e)                   ⇒  e
(DIV+I e (CNST+I 1))                    ⇒  e
(DIV+U e (CNST+U c)), c = 2^k           ⇒  (RSH+U e (CNST+I k))
(GE+U e (CNST+U 0))                     ⇒  (e, (CNST+I 1))
(GE+U (CNST+U 0) e)                     ⇒  (EQ+I e (CNST+I 0))
(GT+U (CNST+U 0) e)                     ⇒  (e, (CNST+I 0))
(GT+U e (CNST+U 0))                     ⇒  (NE+I e (CNST+I 0))
(LE+U (CNST+U 0) e)                     ⇒  (e, (CNST+I 1))
(LE+U e (CNST+U 0))                     ⇒  (EQ+I e (CNST+I 0))
(LT+U e (CNST+U 0))                     ⇒  (e, (CNST+I 0))
(LT+U (CNST+U 0) e)                     ⇒  (NE+I e (CNST+I 0))
(LSH+I e (CNST+I 0))                    ⇒  e
(LSH+U e (CNST+I 0))                    ⇒  e
(MOD+I e (CNST+I 1))                    ⇒  (e, (CNST+I 0))
(MOD+U e (CNST+I c)), c = 2^n           ⇒  (BAND+U e (CNST+U c - 1))
(MUL+I (CNST+I c1) (ADD+I e (CNST+I c2)))  ⇒
    (ADD+I (MUL+I (CNST+I c1) e) (CNST+I c1 × c2))
(MUL+I (CNST+I c1) (SUB+I e (CNST+I c2)))  ⇒
    (SUB+I (MUL+I (CNST+I c1) e) (CNST+I c1 × c2))
(MUL+I (CNST+I c) e), c = 2^k           ⇒  (LSH+I e (CNST+I k))
(NEG+D (NEG+D e))                       ⇒  e
(NEG+F (NEG+F e))                       ⇒  e
(NEG+I (NEG+I e)), e ≠ (CNST+I INT_MIN)  ⇒  e
(RSH+I e (CNST+I 0))                    ⇒  e
(RSH+U e (CNST+I 0))                    ⇒  e
(SUB+P e (CNST+I c))                    ⇒  (ADD+P e (CNST+I -c))
(SUB+P e (CNST+U c))                    ⇒  (ADD+P e (CNST+U -c))
(SUB+P e1 (ADD+I e2 (CNST+I c)))        ⇒
    (SUB+P (SUB+P e1 (CNST+I c)) e2)
```

深入阅读

lcc 编译器的类型检查方法与第 6 章中列出的 Aho，Sethi and Ullman（1986）方法类似。simplify 转换类似于 Hanson（1983）描述的转换。类似更彻底的转换也可以通过其他优化或者在代码生成阶段进行，但是它们通常都会带来额外的开销。函数 simplify 仅仅实现了那些使所有程序都获益的转换。更彻底的转换需要更系统的方法，参见练习 9.8。

练习

9.1 实现如图 9-1 所示的 Type super(Type ty)。注意考虑枚举类型以及 long、unsigned long 和 long double 类型。

9.2 只利用双精度→符号整数（double-to-signed integer）转换如何将双精度转换为一个无符号整数？按照你的方法实现 cast 中的 <double-to-unsigned conversion> 代码段。

9.3 在 lcc 编译器中，所有的枚举类型都是通过整数来表示的，而大多数 C 编译器也都是这样做的。但是 C 语言标准允许枚举类型用任何一种整数类型来表示，只要选择的类型可以保存所有的值。例如，无符号字符可以用来表示枚举值在 0 ～ 255 的枚举类型。试修改 cast 将如何改变才能适应这种模式。lcc 早期版本实现了这种模式。

9.4 完成函数 unary 和 postfix 中省略的代码段。

9.5 在 C 程序中，对空指针进行间接访问（dereference）是常见的程序错误。lcc 编译器的 -n 选项就是用来捕获这种错误的。通过 -n，lcc 在每一个源文件结束时生成如下代码：

```
static char *_YYfile = "file";
static void _YYnull(int line) {
   char buf[200];
   sprintf(buf,"null pointer dereferenced @%s:%d\\n",
      _YYfile, line);
   write(2, buf, strlen(buf));
   abort();
}
```

其中，file 就是源文件名，全局变量 YYnull 指向函数 _YYnull 的符号表表项。当需要构建树来对指针 p 进行间接访问时，如果 YYnull 非空，则调用函数 nullcheck 来构建等价于 ((t1=p)||YYnull(lineno),t1) 的树，其中 t1 是一个临时变量，而 lineno 是一个常量，它给出了对指针进行间接访问的代码在源程序中的行数，并且 lineno 等于全局变量 lineno 的值。这样，在运行时试图对空指针进行间接访问将导致调用函数 _YYnull 报错。实现函数 nullcheck。

9.6 函数 bittree 为 &、|、^ 和 % 构建树，函数 multree 为 * 和 / 构建树，函数 shtree 为 >> 和 << 构建树，函数 subtree 为二元操作符 - 构建树。试实现这些函数。函数 subtree 中的指针减法代码和函数 bittree 中的模操作符 % 代码是最复杂的。函数 subtree 大约有 25 行代码，而其他的每一个函数都少于 20 行。

9.7 对于如下文件作用域的声明，对于表达式 x[10].tble[1].count 会构建什么样的 ADDRP 树？注意使用 simplify 的转换。

```
int i;
struct list {
   char *name;
   struct entry table {
      int age;
      int count;
   } table[10];
} x[100];
```

9.8 函数 simplify 使用专门的技术来实现常量折叠，但它仅实现了其中的一部分转换。研究使用函数 lburg 来实现常量折叠及所有转换的可能性，lburg 将在第 14 章介绍。

语　句

C 语言的语句语法格式如下：

```
statement:
  ID : statement
  case constant-expression : statement
  default : statement
  [ expression ] ;
  if '(' expression ')' statement
  if '(' expression ')' statement else statement
  switch '(' expression ')' statement
  while '(' expression ')' statement
  do statement while '(' expression ')' ;
  for '(' [ expression ] ; [ expression ] ; [ expression ] ')'
      statement
  break ;
  continue ;
  goto ID ;
  return [ expression ] ;
  compound-statement
compound-statement:
  '{' { declaration } { statement } '}'
```

compound-statement 的实现参见 11.7 节。有些语言（如 Pascal）使用分号来将语句分开。而在 C 语言中，分号表示语句结束，所以分号出现在表达式、do-while、break、continue、goto 以及返回语句的产生式中，而不出现在复合语句的产生式中。

10.1　代码的表示

语句的语义包括表达式的计算，可能还夹杂着跳转和标号以实现控制的转换。如 1.3 节所述以及将在第 12 章中详细介绍的，表达式首先被编译为分析树然后转换为 dag（无环有向图）。跳转和标号也通过 dag 来表示。每个函数的这些 dag 在代码表（code list）中被串在一起，代码表代表了函数的代码。编译前端为函数构造代码表并调用接口函数 function。编译后端则调用 gencode 函数和 emitcode 函数来生成和发送代码，这些将在 11.6 节中介绍；这些函数都需要遍历代码表。

代码表是一个 code 结构类型的双向链表：

```
⟨stmt.c typedefs⟩=                                          129
  typedef struct code *Code;

⟨stmt.c exported types⟩=
  struct code {
    enum { Blockbeg, Blockend, Local, Address, Defpoint,
           Label,    Start,    Gen,   Jump,    Switch
```

```
    } kind;
    Code prev, next;
    union {
        (Blockbeg 169)
        (Blockend 170)
        (Local 169)
        (Address 169)
        (Defpoint 170)
        (Label, Gen, Jump 170)
        (Switch 188)
    } u;
};
```

u 的每个域对应于上面 kind 枚举值中除 Start 以外的一个值，Start 不需要 u 的域。Blockbeg 和 Blockend 入口标识复合语句的边界。Local 和 Address 标识必须通过 local 和 address 接口函数来通知编译后端的局部变量。Defpoint 入口定义了执行点的位置，例如，程序中调试器可以设断点的地方。Label、Gen 和 Jump 入口为表达式、标号和跳转传递 dag。Switch 入口传递生成 switch 语句代码所需要的数据。

代码表从 Start 入口开始。codelist 总是指向表中最后一个入口：

```
⟨stmt.c data⟩≡                                                    185
  struct code codehead = { Start };
  Code codelist = &codehead;
```

图 10-1 中上面的图给出了代码表的初始状态。随着函数中的语句被编译，入口被不断地添加到代码表中，代码表不断增大。code 函数负责分配入口，并将其链接到 codelist 所指向的入口之后，然后更新 codelist 使其指向新的入口，这样新入口就成为代码表的最后一个入口。code 函数返回一个指向新入口的指针：

```
⟨stmt.c functions⟩≡                                               169
  Code code(kind) int kind; {
      Code cp;
      ⟨check for unreachable code 168⟩
      NEW(cp, FUNC);
      cp->kind = kind;
      cp->prev = codelist;
      cp->next = NULL;
      codelist->next = cp;
      codelist = cp;
      return cp;
  }
```

图 10-1 中下面的图给出了添加两个入口后的代码表。

用于标识代码表入口的枚举常量的值很重要。大于 Start 的值会生成可执行代码；而小于 Label 的值并不生成代码，仅用于声明编译后端感兴趣的信息。这样，如果在一个无条件跳转指令后添加一个 kind 大于 Start 的入口，将生成不能执行到的代码，code 函数能够检测这样的入口：

```
⟨check for unreachable code 168⟩≡                                 168
  if (kind > Start) {
      for (cp = codelist; cp->kind < Label; )
```

```
        cp = cp->prev;
    if (cp->kind == Jump || cp->kind == Switch)
        warning("unreachable code\n");
}
```

控制并不会使 switch 语句失败，它们更像无条件跳转指令，这将在 10.7 节详细介绍。



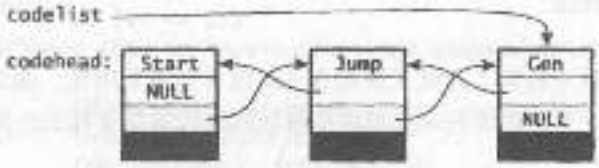


图 10-1 初始代码表和添加两个入口之后的代码表

除非局部变量已经定义过了，否则 addlocal 函数为其添加一个 Local 入口：

```
(Local 169)=                                                    168
  Symbol var;

(stmt.c functions)+=                                            168 170
  void addlocal(p) Symbol p; {
      if (!p->defined) {
          code(Local)->u.var = p;
          p->defined = 1;
          p->scope = level;
      }
  }
```

addrtree 函数说明了 addlocal 函数的用法，还说明了 code 函数如何添加 Address 入口。Address 入口为 gencode 函数调用接口函数 address 传递必要的数据。

```
(Address 169)=                                                  168
  struct {
      Symbol sym;
      Symbol base;
      int offset;
  } addr;
```

当 gencode 函数处理 Address 入口时，它将 sym、base 和 offset 3 个域的值作为 address 函数的 3 个参数。

Blockbeg 入口中存储了编译复合语句所需的信息：

```
(Blockbeg 169)=                                                 168
  struct {
      int level;
      Symbol *locals;
```

```
    Table identifiers, types;
    Env x;
} block;
```

level 表示的是与代码块相关的 level 值，locals 是一个以 null 结尾的数组，该数组中存放了代码块中声明的局部变量的符号表指针。x 表示代码块在编译后端中的 Env 值。当代码块被编译时，identifiers 和 types 记录 identifiers 和 types 表；本书中忽略了使用这些数据的代码，这些代码用来生成选项 -g 规定的调试器符号表信息。Blockend 入口仅指向与它匹配的 Blockbeg：

```
⟨Blockend 170⟩=                                                    168
  Code begin;
```

Label、Gen 和 Jump 入口都存放了指向森林的指针：

```
⟨Label, Gen, Jump 170⟩=                                            168
  Node forest;
```

每个入口都通过枚举常量来标识，因此无须检查 dag 就可知道其用途。例如，code 函数就使用枚举常量来识别跳转，10.9 节中也用它来去掉那些跳转到跳转的指令和无法执行到的跳转指令。

10.2 执行点

执行点可以出现在本章一开始描述的语法中的每个表达式之前、&& 和 || 的操作数之前、操作符？：的第二操作数和第三操作数之前、每个复合语句的开始和结束处，以及每个函数的入口和出口。它们给编译后端为调试器生成代码和符号表信息的机会，这些编译后端实现了 5.2 节提及的 stab 接口函数。例如，调试器允许在执行点设置断点。

执行点和事件还可以用来实现 lcc 编译器产生执行剖面的功能。-b 选项使得 lcc 编译器可以生成代码以计算每个执行点的执行次数并将这些次数写入文件。-a 选项使得该文件在编译时可以读取并且可以用来计算 refinc 的值，给出了精确的执行频率而不是一个估计的执行频率。

执行点入口记录了标识该执行点的源程序位置和一个唯一编号：

```
⟨Defpoint 170⟩=                                                    168
  struct {
     Coordinate src;
     int point;
  } point;
```

definept 函数将 Defpoint 入口添加到代码表中，并填上一个明确的坐标值或 src 的当前值：

```
⟨stmt.c functions⟩+≡                                          169 171
  void definept(p) Coordinate *p; {
     Code cp = code(Defpoint);

     cp->u.point.src = p ? *p : src;
     cp->u.point.point = npoints;
    ⟨reset refinc if -a was specified⟩
    if (events.points)
       ⟨plant event hook⟩
}
```

通常，调用 definept 时使用一个空指针作为参数，但是循环和 switch 语句会在语句结束时生成测试和赋值代码，因此，生成的代码中执行点的顺序与它们在源代码中的顺序是不同的。对于这些执行点，当分析表达式时，相关的坐标值就会保存下来，并在生成该表达式的代码时将保存的坐标值传递给 definept 函数；例如，forstmt 函数中对 definept 函数的调用。

10.3 语句的识别

语句分析函数使用当前单词来标识语句的种类，并根据不同的种类切换到相应的代码处理：

```
⟨stmt.c functions⟩+≡                                          170 173
  void statement(loop, swp, lev) int loop, lev; Swtch swp; {
     float ref = refinc;

     if (Aflag >= 2 && lev == 15)
        warning("more than 15 levels of nested statements\n");
     switch (t) {
     case IF:       ⟨if statement 173⟩ break;
     case WHILE:    ⟨while statement⟩ break;
     case DO:       ⟨do statement⟩ ⟨semicolon 171⟩
     case FOR:      ⟨for statement 176⟩ break;
     case BREAK:    ⟨break statement 180⟩ ⟨semicolon 171⟩
     case CONTINUE: ⟨continue statement 176⟩ ⟨semicolon 171⟩
     case SWITCH:   ⟨switch statement 180⟩ break;
     case CASE:     ⟨case label 181⟩ break;
     case DEFAULT:  ⟨default label 181⟩ break;
     case RETURN:   ⟨return statement 189⟩ ⟨semicolon 171⟩
     case '{':      compound(loop, swp, lev + 1); break;
     case ';':      definept(NULL); t = gettok(); break;
     case GOTO:     ⟨goto statement 175⟩ ⟨semicolon 171⟩
     case ID:       ⟨statement label or fall thru to default 174⟩
     default:       ⟨expression statement 172⟩ ⟨semicolon 171⟩
     }
     ⟨check for legal statement termination 171⟩
     refinc = ref;
  }
⟨semicolon 171⟩≡                                                171
  expect(';');
  break;

⟨check for legal statement termination 171⟩≡                     171
  if (kind[t] != IF && kind[t] != ID
  && t != '}' && t != EOI) {
     static char stop[] = { IF, ID, '}', 0 };
     error("illegal statement termination\n");
     skipto(0, stop);
  }
```

statement 函数有 3 个参数：loop 表示的是最内层的 for、while 或 do-while 循环的标号；swp 表示的是一个指针，该指针指向保存有属于最内层 switch 语句的所有数据的 swtch 结构（参见 10.7

节)；而 lev 表示的是目前语句的嵌套层数。如果当前语句没有被嵌套在任何循环中，loop 就等于 0；如果它不在任何 switch 语句中，swp 就为空。在生成 break 和 continue 语句的代码时要用到 loop，生成 switch 语句的代码需要用到 swp，而 lev 仅仅用于 statement 函数开始时产生警告信息。每一种语句的代码都将这些值传递给嵌套的 statement 调用，并做相应的修改。

loop 循环中使用的标号都是局部标号，它们通过调用 genlabel(n) 生成。genlabel(n) 返回 n 个标号中的第一个。findlabel(n) 返回标号 n 在符号表中的入口。

对于标识符的每次引用，idtree 函数都会将该标识符的 ref 域增加 refinc。ref 域的值与该标识符的引用次数几乎是成比例的。statement 函数及其调用的函数可以改变 refinc 的值以实现标识符在不同地方的引用具有不同的权值。例如，对于出现在 if 语句分支中的引用，refinc 等于 refinc 除以 2，而对于 loop 循环体中的引用，refinc 等于 refinc 乘以 10。ref 域的值用来帮助标识那些适合指派给寄存器的局部变量和参数。此外，局部变量也是按照 ref 值的降序通知编译后端的。

switch 语句的默认分支处理作为语句的表达式：

```
⟨expression statement 172⟩≡                                    171
  definept(NULL);
  if (kind[t] != ID) {
    error("unrecognized statement\n");
    t = gettok();
  } else {
    Tree e = expr0(0);
    listnodes(e, 0, 0);
    if (nodecount == 0 || nodecount > 200)
      walk(NULL, 0, 0);
    deallocate(STMT);
  }
```

listnodes 和 walk 这两个函数从分析树生成 dag。它们的实现将在第 12 章详细解释，但是为了理解编译前端是如何实现语句的语义分析的，现在必须解释一下它们的用法。

listnodes 函数的第一个参数是一棵分析树，如第 5 章介绍的，listnodes 函数根据该分析树生成 dag，并将该 dag 添加到由其维护的不断增长的森林中。这样，上面对 listnodes 函数的调用，就根据 expr0 函数返回的分析树生成 dag，并将其添加到森林中。对于如下输入：

```
c = a + b;
a = a/2;
d = a + b;
```

<expression statement> 代码段会被执行 3 次，每次处理一条语句，因此 listnodes 函数就被调用了 3 次。第一次调用将 c= a+b 的 dag 添加到初始的空森林中，第二次和第三次调用通过添加第二个和第三个语句的 dag 使得森林不断增大。在 12.1 节将会看到，如果有可能，listnodes 函数会重用公共子表达式；例如，在赋值语句 d=a+b 中，它会重用第一个赋值语句形成的 a 的左值和 b 的右值的 dag。但因为第二个赋值语句改变了 a，所以不能重用 a 的右值。

listnodes 函数的第二个和第三个参数是标号，它们的作用将在下一节介绍；上面 listnodes 函数调用中的 0 表示没有标号。listnodes 函数还可以接受空树，直接返回空。

listnodes 函数保持森林直到 walk 函数被调用，walk 函数接受的参数与 listnodes 函数的参数相同。walk 函数的处理分两步：第 1 步，它将参数传递给 listnodes 函数，所以调用 walk 函数与调用 listnodes 函数有相同的作用。第 2 步，也是最重要的，walk 函数分配一个 Gen 代码表入口，

将森林保存在该入口中，再将该入口添加到代码表中，然后清除森林。一旦将森林添加到代码表中，它的 dag 就不能被 listnodes 函数重用了。

walk(NULL,0,0) 真效地只执行第 2 步，如果当前森林非空，就将它添加到代码表中。只要需要将当前森林添加到代码表中，就可以调用该函数。例如其他一些可执行的代码表入口要添加进来，或者两个或多个分离的控制流要合并。在上面的代码中，当 nodecount 等于 0 或者超过 200 时就会执行该调用。nodecount 表示森林中可以重用的节点数。当森林没有可重用的节点或者当森林变大时，walk 函数就会被调用。前一个条件将不含共享公共子表达式的 dag 添加到不同的森林中，而后一个条件是限制森林的大小的；两种结果对编译后端都很有用。

deallocate 函数释放 STMT 分配区中的所有空间，而 STMT 分配区也就是为分析树分配内存空间的地方。walk 函数也释放 STMT 分配区的空间。

10.4 if 语句

为 if 语句生成的代码有如下的形式：

```
        if expression == 0 goto L
        statement1
        goto L + 1
L:      statement2
L + 1:
```

如果省略了 else 部分，那么 goto L+1 语句也随之忽略。处理代码是：

```
⟨if statement 173⟩=                                                    171
  ifstmt(genlabel(2), loop, swp, lev + 1);

⟨stmt.c functions⟩+=                                                   ↑171 174↓
  static void ifstmt(lab, loop, swp, lev)
  int lab, loop, lev; Swtch swp; {
     t = gettok();
     expect('(');
     definept(NULL);
     walk(conditional(')'), 0, lab);
     refinc /= 2.0;
     statement(loop, swp, lev);
     if (t == ELSE) {
        branch(lab + 1);
        t = gettok();
        definelab(lab);
        statement(loop, swp, lev);
        if (findlabel(lab + 1)->ref)
           definelab(lab + 1);
     } else
        definelab(lab);
  }
```

ifstmt 函数的第一个参数是 L，genlabel(2) 生成 if 语句中使用的两个标号。ifstmt 的另外 3 个参数就是 statement 的 3 个参数。conditional 函数通过调用 expr 函数来分析表达式，并保证结果树是一个条件表达式，该表达式的值只用来改变控制流。条件表达式分析树的根是下列条件操作符中的一个：

AND、OR、NOT 或者常量。conditional 函数的参数就是在调用 conditional 函数的上下文中紧接着表达式的单词。

```
⟨stmt.c functions⟩+≡                                    173 125
  static Tree conditional(tok) int tok; {
     Tree p = expr(tok);

     if (Aflag > 1 && isfunc(p->type))
        warning("%s used in a conditional expression\n",
           funcname(p));
     return cond(p);
  }
```

listnodes 和 walk 函数的第二个和第三个参数是表示 true 或 false 目标的标号。walk(e，tlab，flab) 将其参数传递给 listnodes 函数。而正如上一节所说的，listnodes 函数根据 e 生成 dag 并将它添加到森林中，并且还将一个保存了森林的 Gen 入口添加到代码表中。如果 e 是一个条件表达式的分析树，那么 tlab 或 flab 不等于 0。如果 tlab 不等于 0，若 e 的结果不等于 0，那么 listnodes 函数就生成将控制转移到 tlab 的 dag；否则，如果 e 计算结果等于 0，则 listnodes 函数生成跳转到 flab 的 dag。即使 tlab 和 flab 都非零，也只能有一次用非零值调用 listnodes 和 walk 函数；对于其他情况控制就会失败。

对于 if 语句，在上面的生成代码中，walk 函数是用与 L 对应的非零值 flab 来调用的。definelab 和 branch 函数为标号定义和跳转生成代码表项。只有需要时才定义 L+1。每当标号被用作分支的目标时，它的 ref 域就会增加。例如，如果没有转到 L+1 的分支，标号 L+1 也就没有必要了，下面的代码会出现这种情况：

```
if (...)
   return;
else
   ...
```

返回语句的作用类似于一个无条件跳转指令，所以调用 branch(lab+1) 不会产生分支代码。

在 idtree 函数中，每次对标识符的引用，都会使该标识符的 ref 增加 refinc。lcc 估计 if 语句的每一个分支被执行的次数大致相同，所以在分析 if 语句的分支之前，refinc 被二等分。这样做的结果是，在每个分支中对标识符引用的次数是 if 语句之前和之后的语句中引用次数的一半。由于 statement 函数已经恢复了 refinc 的值，所以 ifstmt 函数就不必再恢复了。

10.5 标号和 goto 语句

对于一个以标识符开始的语句，如果它后面紧接着一个冒号，那么该标识符就是一个标号，否则，它就是一个表达式的开始。

```
⟨statement label or fall thru to default 174⟩≡                171
  if (getchr() == ':') {
     stmtlabel();
     statement(loop, swp, lev);
     break;
  }
```

getchr 函数恰好处理到下一个单词符号的开始字符并返回该字符。该字符被用来向前搜索一个字符，以检查是否为冒号。由于标识符既可以是一个标号，同时也可以是一个变量，所以需要一个独立的表 stmtlabs 保存源语言标号：

```
⟨stmt.c exported data⟩=
  extern Table stmtlabs;
```

像其他表格一样，stmtlabs 也由 lookup 函数和 install 函数管理。它将源语言中的标号映射为内部标号，内部标号存储在符号的 u.l.label 域中。

```
⟨stmt.c functions⟩+≡                                    174 177
  static void stmtlabel() {
     Symbol p = lookup(token, stmtlabs);

     ⟨install token in stmtlabs, if necessary 175⟩
     if (p->defined)
        error("redefinition of label '%s' previously _
           defined at %w\n", p->name, &p->src);
     p->defined = 1;
     definelab(p->u.l.label);
     t = gettok();
     expect(':');
  }
```

definelab(n) 构建了一个用于定义标号 n 的 LABELV dag，分配一个保存该 dag 的 Label 代码表入口，并将该 Label 入口添加到代码表中。

标号可在被引用之前定义，反之亦可，因此，当它们作为一个语句标示出现，或者出现在 goto 语句中时，都可以建立到表中。

```
⟨install token in stmtlabs, if necessary 175⟩=                    175
  if (p == NULL) {
     p = install(token, &stmtlabs, 0, FUNC);
    p->scope = LABELS;
    p->u.l.label = genlabel(1);
    p->src = src;
  }
```

标号的 ref 域用来对标号的引用次数进行计数，由 install 函数初始化为 0。标号每引用一次，它的 ref 域就被加 1：

```
⟨goto statement 175⟩=                                             171
  walk(NULL, 0, 0);
  definept(NULL);
  t = gettok();
  if (t == ID) {
     Symbol p = lookup(token, stmtlabs);
     ⟨install token in stmtlabs, if necessary 175⟩
     use(p, src);
     branch(p->u.l.label);
     t = gettok();
  } else
     error("missing label in goto\n");
```

branch(n) 为转到标号 n 的分支构建了一个 JUMPV dag，并分配 Jump 代码表入口以保存该 dag，然后将该 Jump 入口添加到代码表中。同时还将 n 的 ref 域加 1。

funcdefn 函数在函数定义结束时调用 checklab 函数，以发现并通知那些未定义的标号（也就是那些在 goto 语句中引用的，但从来没定义过的标号）。

10.6 循环

为 3 种不同循环生成的代码都有类似的结构，它们都涉及 3 个标号：L 表示循环头，L+1 表示循环的条件测试部分，L+2 表示循环出口。例如，while 循环生成的代码如下：

```
        goto L + 1
L:      statement
L + 1:  if expression != 0 goto L
L + 2:
```

这种代码结构要比下面的代码好：

```
L:
L + 1:  if expression != 0 goto L + 2
        statement
        goto L
L + 2:
```

当循环体执行 n 次时，前一种代码执行了 n+2 个分支指令，而第二种结构很明显执行了 2n +1 个分支指令。

continue 语句的代码会跳转到 L+1，而 break 语句的代码会跳转到 L+2。L 是循环句柄，它被传递到 statement 及其调用的函数（这与 ifstmt 函数需要 L 作为参数一样）。例如，仅当存在一个循环句柄时，continue 语句才是合法的：

```
⟨continue statement 176⟩=                                    171
  walk(NULL, 0, 0);
  definept(NULL);
  if (loop)
     branch(loop + 1);
  else
     error("illegal continue statement\n");
  t = gettok();
```

for 循环语句有 4 个标号，其中前 3 个与 while 循环中的标号意义相同。当 for 循环的 3 个控制表达式都存在时，生成的代码结构如下所示：

```
        expression1
        goto L + 3
L:      statement
L + 1:  expression3
L + 3:  if expression2 != 0 goto L
L + 2:
```

expression$_1$、expression$_2$ 和 expression$_3$ 分别称为初始化表达式、条件测试表达式和增量表达式。

分析函数的复杂性主要表现在处理可选的表达式，确定可执行点的正确位置，以及实现至少执行一次循环体的循环的优化。

```
⟨for statement 176⟩=                                         171
  forstmt(genlabel(4), swp, lev + 1);
```

```
⟨stmt.c functions⟩+≡                                          175 180
  static void forstmt(lab, swp, lev)
  int lab, lev; Swtch swp; {
     int once = 0;
     Tree e1 = NULL, e2 = NULL, e3 = NULL;
     Coordinate pt2, pt3;

     t = gettok();
     expect('(');
     definept(NULL);
     ⟨forstmt 177⟩
  }
```

首先分析初始化表达式，并添加到代码表中：

```
⟨forstmt 177⟩≡                                                177 177
  if (kind[t] == ID)
     e1 = texpr(expr0, ';', FUNC);
  else
     expect(';');
  walk(e1, 0, 0);
```

接下来分析条件测试表达式，但是只有等到循环体编译完后，才能将它传递给 walk 函数。赋值语句 pt2=src 保存了条件测试表达式在源程序中的位置，在将条件测试表达式的分析树传递给 walk 函数之前，需要该位置调用 definept 函数。

```
⟨forstmt 177⟩+≡                                          177 177 177
  pt2 = src;
  refinc *= 10.0;
  if (kind[t] == ID)
     e2 = texpr(conditional, ';', FUNC);
  else
     expect(';');
```

walk 函数有一种重要的副作用：释放 STMT 分配区。tree 函数在该分配区内为分析树分配内存空间。texpr 函数在 FUNC 分配区内为条件测试表达式的分析树分配内存空间，这些空间直到循环体编译完、调用 walk 函数时一直有效。处理增量表达式时也要用到 texpr 函数：

```
⟨forstmt 177⟩+≡                                          177 178 177
  pt3 = src;
  if (kind[t] == ID)
     e3 = texpr(expr0, ')', FUNC);
  else {
     static char stop[] = { IF, ID, '}', 0 };
     test(')', stop);
  }
```

pt3 保存了增量表达式在源程序中的位置，将来调用 definept 函数需要该位置。

lcc 估计循环体执行的次数是循环外的语句执行次数的 10 倍，因此 refinc 等于 10 乘以 refinc，这便相应地设定了循环体内引用标识符的权值。

大多数 for 循环都类似于下面的代码：

```
sum = 0;
for (i = 0; i < 10; i++)
   sum += x[i];
```

这类循环的循环体至少会被执行一次，刚开始的 goto L+3 语句可以省略，下面的代码实现了这种情况：

```
⟨forstmt 177⟩+≡                                          177 178  177
  if (e2) {
     once = foldcond(e1, e2);
     if (!once)
        branch(lab + 3);
  }
```

foldcond 函数检查初始化表达式和条件测试表达式的分析网，以判断循环体是否至少被执行一次；参见练习 10.3。e1 传递给 foldcond 函数，所以上面的代码中需要使用 texpr 函数分析 e1。

forstmt 函数的其余部分用来编译循环体并生成上述的标号和表达式。

```
⟨forstmt 177⟩+≡                                          178  177
  definelab(lab);
  statement(lab, swp, lev);
  definelab(lab + 1);
  definept(&pt3);
  if (e3)
     walk(e3, 0, 0);
  if (e2) {
     if (!once)
        definelab(lab + 3);
     definept(&pt2);
     walk(e2, lab, 0);
  } else {
     definept(&pt2);
     branch(lab);
  }
  if (findlabel(lab + 2)->ref)
     definelab(lab + 2);
```

生成的标号的符号表入口是通过 findlabel 函数加载到 labels 表中的。像其他标号一样，如果生成的标号是跳转目标，则该标号的 ref 域就不等于 0。

10.7 switch 语句

C 语言的 switch 语句与其他语言（如 Pascal）的 case 语句有很大的不同。任何语句都可以跟在 switch 子句的后面，switch 语句的语法并没有明确规定 case 标号和 default 标号的位置。另外，在执行完某个 case 标号标识的分支语句后，控制就会转换到下一个语句，该语句可能有另外一个 case 标号标识。case 标号和 default 标号只是简单的标号，并没有其他语义。例如：

```
switch (n%4)
   while (n > 0) {
   case 0: *x++ = *y++; n--;
```

```
case 3: *x++ = *y++; n--;
case 2: *x++ = *y++; n--;
case 1: *x++ = *y++; n--;
}
```

该代码的功能为当 n 大于等于 1 时，从 y 中复制 n 个值到 x 中。如果该循环被展开，则每次循环复制 4 个值。switch 语句复制前 n%4 个值，而 n/4 次迭代复制其余的值。这个例子虽然不自然，但也还是合法的。

对于一个具有 n 个 case 分支和一个 default 分支的 switch 语句，生成的代码具有如下所示的形式：

```
        t1 ← expression
        select and jump to L1,...,Ln,L
        code for statement
L+1:
```

这里 t1 是一个与 switch 语句相关的临时变量，而 L+1 是一个出口标号。每一个 case 标号为它所生成的标号 L 生成一个定义，默认标号为 L 生成一个定义，switch 语句中的每个 break 语句都生成一个跳转到出口标号的代码：

```
goto L+1
```

如果没有默认分支，L 就和 L+1 代表相同的地址。

分析 switch 语句、case 标号、default 标号以及 break 语句都比较简单，困难的是为 <select and jump> 程序段生成好的代码。每一个 case 标号都与一个整数值关联。这些值 - 标号对用来根据 expression 的值选择跳转到相应的分支。这些数据同其他数据一起存放在与 switch 语句相关的 swtch 结构中：

```
⟨stmt.c typedefs⟩+≡                                    ↑167
  typedef struct swtch *Swtch;

⟨stmt.c types⟩=
  struct swtch {
     Symbol sym;
     int lab;
     Symbol deflab;
     int ncases;
     int size;
     int *values;
     Symbol *labels;
  };
```

sym 中保存的是临时变量 t1 的值，lab 保存了 L 的值，如果有 default 标号，deflab 就指向 default 标号在符号表中的入口。values 和 labels 指向保存了值 - 标号对的数组。这两个数组都具有 size 个元素，其中 ncases 个元素被使用，且这 ncases 个元素按照 values 的升序排列。指向当前 switch 语句的 swtch 结构（也就是一个 switch 句柄）的指针，会被传递给 statement 函数及其调用的函数。

case 标号和 default 标号的处理与 break 和 continue 语句的处理非常类似：case 标号和 default 标号与最内层或当前的 switch 语句相关，当 switch 句柄为空时，即 case 标号和 default 标号出现在 switch 语句之外，就会导致报错。break 语句的代码通过检查循环句柄和 switch 句柄来判断它是否与某个循环或 switch 语句关联：

```
⟨break statement 180⟩=                                          171
  walk(NULL, 0, 0);
  definept(NULL);
  if (swp && swp->lab > loop)
    branch(swp->lab + 1);
  else if (loop)
    branch(loop + 2);
  else
    error("illegal break statement\n");
  t = gettok();
```

随着标号的生成，标号的值不断增加，所以如果存在一个 switch 句柄且它的 L 值比循环句柄大，那么该 break 语句与 switch 语句相关。

分析一个 switch 语句要涉及以下几个方面：对表达式的分析和类型检查、生成临时变量、在代码表中添加一个 Switch 占位符、初始化一个新的 switch 句柄并将它传递给 statement 函数、生成结束标号和选择代码等。

```
⟨switch statement 180⟩=                                         171
  swstmt(loop, genlabel(2), lev + 1);

⟨stmt.c macros⟩=                                                185
  #define SWSIZE 512
⟨stmt.c functions⟩+=                                    177 182
  static void swstmt(loop, lab, lev) int loop, lab, lev; {
    Tree e;
    struct swtch sw;
    Code head, tail;

    t = gettok();
    expect('(');
    definept(NULL);
    e = expr(')');
    ⟨type-check e 180⟩
    ⟨generate a temporary to hold e, if necessary 181⟩
    head = code(Switch);
    sw.lab = lab;
    sw.deflab = NULL;
    sw.ncases = 0;
    sw.size = SWSIZE;
    sw.values = newarray(SWSIZE, sizeof *sw.values, FUNC);
    sw.labels = newarray(SWSIZE, sizeof *sw.labels, FUNC);
    refinc /= 10.0;
    statement(loop, &sw, lev);
    ⟨define L, if necessary, and L + 1 183⟩
    ⟨generate the selection code 184⟩
  }
```

生成选择代码时，代码表中的 Switch 入口占位符就会被一个或多个 Switch 入口取代。switch 表达式必须是整数类型，并可能被提升：

```
⟨type-check e 180⟩=                                             180
  if (!isint(e->type)) {
```

```
        error("illegal type '%t' in switch expression\n",
            e->type);
        e = retype(e, inttype);
    }
    e = cast(e, promote(e->type));
```

临时变量也具有 e->type 类型，但是在有些情况下可以不需要临时变量。如果 switch 表达式是一个简单的标识符，并且类型正确又不是可变类型，那么就可以直接使用它。否则，该表达式就赋值给临时变量：

```
⟨generate a temporary to hold e, if necessary 181⟩=                    180
    if (generic(e->op) == INDIR && isaddrop(e->kids[0]->op)
    && e->kids[0]->u.sym->type == e->type
    && !isvolatile(e->kids[0]->u.sym->type)) {
        sw.sym = e->kids[0]->u.sym;
        walk(NULL, 0, 0);
    } else {
        sw.sym = genident(REGISTER, e->type, level);
        addlocal(sw.sym);
        walk(asgn(sw.sym, e), 0, 0);
    }
```

一旦 switch 句柄被初始化，case 标号和 default 标号就可以将数据添加到句柄中。例如，如果一个句柄的 deflab 域还没有被填充，则 default 标号就会填充该域：

```
⟨default label 181⟩=                                                   171
    if (swp == NULL)
        error("illegal default label\n");
    else if (swp->deflab)
        error("extra default label\n");
    else {
        swp->deflab = findlabel(swp->lab);
        definelab(swp->deflab->u.l.label);
    }
    t = gettok();
    expect(':');
    statement(loop, swp, lev);
```

case 标号与此类似，标号的值被转换为 switch 表达式的提升类型，同时生成和定义与该值关联的标号：

```
⟨case label 181⟩=                                                      171
    {
        int lab = genlabel(1);
        if (swp == NULL)
            error("illegal case label\n");
        definelab(lab);
        while (t == CASE) {
            static char stop[] = { IF, ID, 0 };
            Tree p;
            t = gettok();
            p = constexpr(0);
            if (generic(p->op) == CNST && isint(p->type)) {
                if (swp) {
                    needconst++;
```

```
            p = cast(p, swp->sym->type);
            needconst--;
            caselabel(swp, p->u.v.i, lab);
        }
      } else
        error("case label must be a constant _
          integer expression\n");
      test(':', stop);
    }
    statement(loop, swp, lev);
  }
```

在调用 cast 函数之前 needconst 会自增，这样即使发生溢出，simplify 函数也能合并转换。例如对如下输入：

```
int i;
switch (i)
case 0xffffffff: ;
```

因为该 case 值是一个无符号数，不能表示成整数，所以会产生如下诊断信息：

```
warning: overflow in constant expression
```

值得注意的是，即使一个 case 标号出现在 switch 语句的外面，它也会被处理。这样就可以防止 case 标号引起其他的语法错误。

caselabel 函数负责将值和标号添加到 switch 句柄中的 values 和 labels 数组中。它还负责对重复标号进行检查：

```
⟨stmt.c functions⟩+≡                                    ↑180 185↓
  static void caselabel(swp, val, lab)
  Swtch swp; int val, lab; {
     int k;

     if (swp->ncases >= swp->size)
        ⟨double the size of values and labels⟩
     k = swp->ncases;
     for ( ; k > 0 && swp->values[k-1] >= val; k--) {
        swp->values[k] = swp->values[k-1];
        swp->labels[k] = swp->labels[k-1];
     }
     if (k < swp->ncases && swp->values[k] == val)
        error("duplicate case label '%d'\n", val);
     swp->values[k] = val;
     swp->labels[k] = findlabel(lab);
     ++swp->ncases;
     if (Aflag >= 2 && swp->ncases == 258)
        warning("more than 257 cases in a switch\n");
  }
```

上面的 for 循环将新标号和值插入 values 和 labels 数组的合适位置，使数组元素按值的升序来排序，这将有助于检查重复的标号值和生成好的选择代码。如果有必要，这些数组大小可以加倍以适应新的值－标号对。

在 statement 函数返回到 swstmt 函数之后，如果没有显式的 default 标号，就定义 default 标号，如果引用了出口标号，则定义出口标号 L+1：

```
(define L, if necessary, and L + 1 183)=                    180
  if (sw.deflab == NULL) {
     sw.deflab = findlabel(lab);
     definelab(lab);
     if (sw.ncases == 0)
        warning("switch statement with no cases\n");
  }
  if (findlabel(lab + 1)->ref)
     definelab(lab + 1);
```

因为选择代码可能会引用 default 标号，所以即使 default 标号未被引用，也会被定义。

直到所有的 case 分支语句都被检查完，才能生成选择代码。对语句进行编译将在代码表中添加入口，而选择代码的入口应该出现在表达式的入口之后、语句的入口之前。如果插入了分支语句，那么选择代码就可以出现在语句的后面，这样选择代码仍可以在语句之前执行。对于这个问题，有一种更简单的可以生成更好的代码的解决方法：重新安排代码表。

图 10-2 最上面的图给出了出口标号定义之后的代码表。其中，实心圆表示的是表达式的入口，空心圆表示 Switch 占位符，空心方块表示语句的入口，包括对 case 标号和 default 标号的定义以及由 break 语句生成的跳转指令。head 指向占位符，codelist 指向最后一条语句的入口。

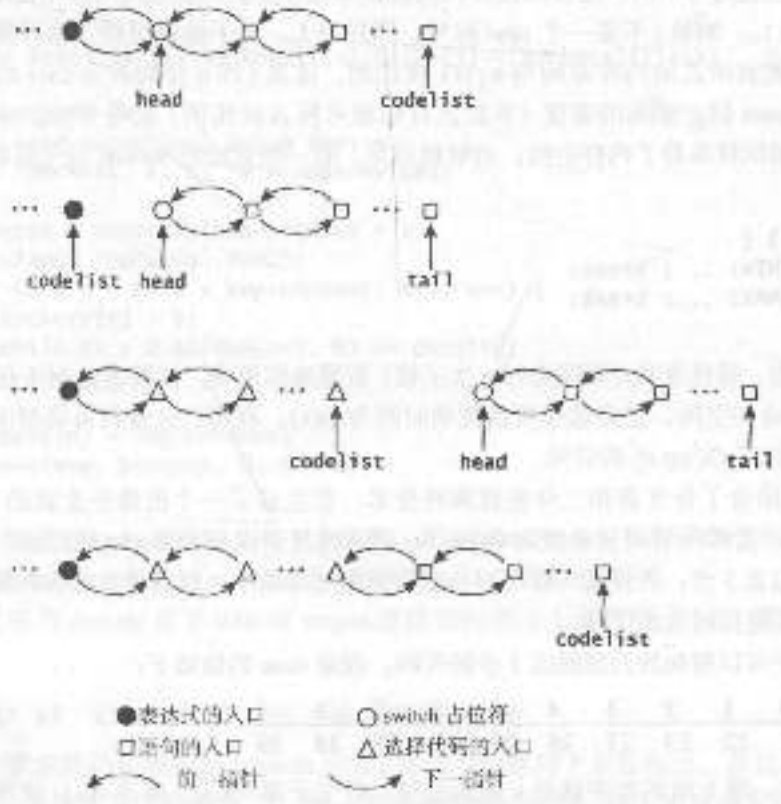


图 10-2 生成 switch 选择代码的代码表处理图

生成选择代码的第一步就是构建实心圆作为代码表的结尾：

```
(generate the selection code 184)=                    184  180
  tail = codelist;
  codelist = head->prev;
  codelist->next = head->prev = NULL;
```

图 10-2 的第二幅图给出了这些语句的结果。head 和 tail 分别指向占位符和语句的入口，codelist 指向表达式的入口。生成选择代码时，它的入口就被添加到适当的地方：

```
(generate the selection code 184)+=                   184 184  180
  if (sw.ncases > 0)
     swgen(&sw);
  branch(lab);
```

图 10-2 的第三幅图给出了添加选择代码入口以后的代码表，其中选择代码的入口用三角形表示。最后一步就是将由 head 和 tail 保存的整个列表添加到代码表中，并把 codelist 设置为 tail:

```
(generate the selection code 184)+=                   184  180
  head->next->prev = codelist;
  codelist->next = head->next;
  codelist = tail;
```

图 10-2 的最后一幅图给出了最后的结果，图中忽略了占位符。

当 case 分支超过 3 个时，最快的选择代码就是采用分支表：表达式的值用作该表的索引，第 i 个入口保存了 L，如果 i 不是一个 case 标号，则保存 L。对于这种组织，选择操作所消耗的时间是固定的。该表所占的内存空间与 u−l+1 成比例，这里 l 和 u 分别表示 case 的最小值和最大值。对于 n 个 case 值，该表的密度（非默认目标标号所占的比例）就等于 n/(u−l+1)。如果该密度太小，这种组织就浪费了内存空间。更糟糕的是，有一些合法的 switch 语句采取这种方法并不实际：

```
switch (i) {
case INT_MIN: ...; break;
case INT_MAX: ...; break;
}
```

另一种极端方法，线性查找（顺序进行 n 次比较）更紧凑但更慢。这种方法对于任意 case 标号集合都只需要 O(n) 的空间，但是选择所消耗的时间为 O(n)。利用二分查找可将时间复杂性降低到 O(log n)，但增加了 O(log n) 的空间。

lcc 编译器结合了分支表和二分查找两种技术：它生成了一个密集分支表的二分查找代码。如果有 m 个表，选择的时间复杂度为 O(log m) 而空间复杂度与 n+log m 成正比。通过这种方法生成选择代码包含 3 步：先将值 − 标号对分配到密集分支表中，然后将这些表转换为与二分查找对应的树，最后遍历树生成代码。

下面的例子可以帮助我们理解这 3 步的代码。假定 case 的值如下：

i	0	1	2	3	4	5	6	7	8	9
$v[i]$	21	22	23	27	28	29	36	37	38	39

v 是 values 数组。线上面的数字就是 v 的索引号。对于子集 v[i..j]，密度 d(i,j) 就等于数组的元素个数除以这些值的范围：

$$d(i,j) = (j - i + 1)/(v[j] - v[i] + 1)$$

例如：

$$d(0,9) = (9-0+1)/(39-21+1) = 10/19 = 0.53$$
$$d(0,5) = (5-0+1)/(29-21+1) = 6/9 = 0.67$$
$$d(6,9) = (9-6+1)/(39-36+1) = 4/4 = 1.0$$

density 的值就是分支表中最小的密度值：

```
⟨stmt.c data⟩+≡                                          168
  float density = 0.5;
```

默认的密度值等于 0.5，因为 d(0, 9)>0.5，使得上面的例子存放在一个表格中。lcc 编译器的 -dx 选项可以将 density 的值改为 x。如果 density 等于 0.66，那么该例子就生成两个表格（v[0..5] 和 v[6..9]），而如果 density 等于 0.75 就生成 3 个表格（v[0..2]、v[3..5] 和 v[6..9]）。如果 density 等于 1.0，那么就会有 n 个表格，每个表格只有一个元素，这些表格与一个二分查找相对应。

一个简单的贪婪算法可以实现这种划分：如果当前表格为 v[i..j]，且 d(i, j+1) ≥ density，就将表格扩展为 v[i..j+1]。表格被扩展后，如果该表格与其前驱表格合并后的密度值大于 density，则将它们合并。swgen 函数是这样实现以上两步的，将单元素 v[j+1] 当作表格 v[j+1..j+1] 来处理，如果有可能，就将它与其前驱合并。在下面的代码中，buckets[k] 就是第 k 个表的第一个值在 v 中的索引，也就是说，表 k 是 v[buckets[k]..buckets[k+1]-1]。对于 n 个 case 值，最多有 n 个表格，所以 buckets 数组有 n+1 个元素。

```
⟨stmt.c macros⟩+≡                                        180
  #define den(i,j) ((j-buckets[i]+1.0)/(v[j]-v[buckets[i]]+1))

⟨stmt.c functions⟩+≡                                     182 186
  static void swgen(swp) Switch swp; {
     int *buckets, k, n, *v = swp->values;

     buckets = newarray(swp->ncases + 1,
        sizeof *buckets, FUNC);
     for (n = k = 0; k < swp->ncases; k++, n++) {
        buckets[n] = k;
        while (n > 0 && den(n-1, k) >= density)
           n--;
     }
     buckets[n] = swp->ncases;
     swcode(swp, buckets, 0, n - 1);
  }
```

当 swgen 函数调用 swcode 函数时，有 n 个表格，而 buckets[0..n-1] 保存了每个表格的第一个值在 v 中的索引，且 buckets[n] 等于 n，也就是假定的第 n+1 个表格的索引。

下面将说明当 density 等于 0.66 时 swgen 函数如何划分上面的例子。for 循环第一次循环结束时：

```
v[i]  21  22  23  27  28  29  36  37  38  39
```

表中第一个元素左边的竖线表示 buckets 的值。k 对应的值用下划线标出。所以，第一次循环结束时，k 等于 0 并且它对应的值为 21，唯一的表格就是 v[0..0]。接下来的两次循环将 buckets[1] 分别设置为 1 和 2，将单个元素的表格 v[1..1] 和 v[2..2] 与它们的前驱 v[0..0] 和 v[0..1] 合并。第三次循环结束时的状态如下：

```
v[i] |21  22  23  27  28  29  36  37  38  39
```

只有表格 v[0..2]。第四次循环不能将包含 27 的 v[3..3] 与 v[0..2] 合并，因为密度 d(0,3)=4/7=0.57 太小了，所以状态变为：

```
v[i] |21  22  23 |27  28  29  36  37  38  39
```

接下来，v[4..4]（28）可以与 v[3..3] 合并，但是 v[3..4] 不能与 v[0..3] 合并，因为 d(0..4)=5/8=0.63。

检查 29 的循环是最有趣的。在 while 循环之前，n 等于 2 且状态为：

```
v[i] |21  22  23 |27  28 |29  36  37  38  39
```

while 循环将 v[3..4] 与 v[5..5] 合并，并将 n 减至 1；因为 d(0,5)=6/9=0.67，可以将 v[0..2] 与刚刚形成的 v[3..5] 合并，并将 n 减至 0。while 循环后的状态为：

```
v[i] |21  22  23  27  28  29  36  37  38  39
```

该过程结束时分为两个表：在调用 swcode 函数之前的状态如下（最右边的竖线表示 buckets[n] 的值）：

```
  i  | 0   1   2   3   4   5 | 6   7   8   9 |
v[i] |21  22  23  27  28  29 |36  37  38  39 |
```

n 等于 2，而 buckets 中保存索引 0、6 和 10。

最后两步就是将上面 buckets 描述的表转换为树，然后对树进行转换，为每个表生成选择代码。swcode 函数使用分治算法同时处理这两步。swgen 函数调用 swcode 函数，提供 switch 句柄、buckets、buckets 的下界和上界以及表格的数目作为参数。buckets 在其最后一个元素后面还有一个标记，从而简化了对最后一个表格中最后一个 case 值的访问。

swcode 函数对 b[lb..ub] 给定的 ub−lb+1 个表格生成代码。它选择中间的表格作为查找树的根节点，为其生成代码，并递归地调用自身为根节点两边的表格生成代码。

```
⟨stmt.c functions⟩+≡                                    185 188
  static void swcode(swp, b, lb, ub)
  Swtch swp; int b[]; int lb, ub; {
     int hilab, lolab, i, u, k = (lb + ub)/2;
     int *v = swp->values;

     ⟨swcode 186⟩
  }
```

当只有一个表格时，如果 switch 表达式的值不在表格覆盖范围之内，控制将会转到 default 标号。需要对表格进行二分查找时，如果 switch 表达式在当前表格范围之外，控制流将转到相应的子表中。

```
⟨swcode 186⟩≡                                           187  186
  if (k > lb && k < ub) {
     lolab = genlabel(1);
     hilab = genlabel(1);
  } else if (k > lb) {
```

```
        lolab = genlabel(1);
        hilab = swp->deflab->u.l.label;
    } else if (k < ub) {
        lolab = swp->deflab->u.l.label;
        hilab = genlabel(1);
    } else
        lolab = hilab = swp->deflab->u.l.label;
```

如果 switch 表达式小于根的最小值或者大于根的最大值，则 lolab 和 hilab 指明控制应当转到的地方。如果搜索树既有左子表又有右子表，那么 lolab 和 hilab 就标识它们的代码序列。当没有右子表时，default 标号就用作 hilab，没有左子表时它就用作 lolab。如果根节点是唯一的表格，那么 default 标号既可以用作 hilab 也可以用作 lolab。

最后，生成根节点表格的代码：

```
⟨swcode 186⟩+≡                                    186 187 186
  l = b[k];
  u = b[k+1] - 1;
  if (u - l + 1 <= 3)
      ⟨generate a linear search⟩
  else {
      ⟨generate an indirect jump and a branch table 187⟩
  }
```

swcode 函数会被递归调用，生成左、右子表的代码。

```
⟨swcode 186⟩+≡                                    187 186
  if (k > lb) {
      definelab(lolab);
      swcode(swp, b, lb, k - 1);
  }
  if (k < ub) {
      definelab(hilab);
      swcode(swp, b, k + 1, ub);
  }
```

每个分支表处理两个条件比较和一个间接跳转指令——至少处理 3 条指令。对于大多数目标机器，如果表格中的值超过 3 个，那么使用分支表就很适合。否则简短的线性查找会更好，参见练习 10.8。

通过分支表为间接跳转指令生成的代码具有如下形式：

```
if t1 < v[1] goto lolab
if t1 > v[u] goto hilab
goto *table[t1-v[1]]
```

这里 v[1]、v[u]、lolab 和 hilab 都会被 swcode 函数计算的相应值替代。分支表是一个静态指针数组，间接跳转指令的跳转目标对应的树与数组索引的树相同：

```
⟨generate an indirect jump and a branch table 187⟩≡          188 187
  Symbol table = genident(STATIC,
      array(voidptype, u - l + 1, 0), LABELS);
  (*IR->defsymbol)(table);
  cmp(LT, swp->sym, v[1], lolab);
  cmp(GT, swp->sym, v[u], hilab);
```

```
walk(tree(JUMP, voidtype,
   rvalue((*optree['+'])(ADD, pointer(idtree(table)),
      (*optree['-'])(SUB,
         cast(idtree(swp->sym), inttype),
         consttree(v[l], inttype)))), NULL), 0, 0);
```

cmp 函数是为比较表达式：

```
if p ⊗ n goto L
```

构建分析树，并将它转换为 dag。p 是一个标识符，⊗ 表示一个关系运算符，n 是一个整数常量：

```
⟨stmt.c functions⟩+≡                                    186 189
  static void cmp(op, p, n, lab) int op, n, lab; Symbol p; {
     listnodes(eqtree(op,
           cast(idtree(p), inttype),
           consttree(n, inttype)),
        lab, 0);
  }
```

cmp 函数还可用来生成一个线性查找，参见练习 10.8。

分支表是这样生成的，定义静态变量 table，为表格中的每个标号调用接口函数 defaddress，但是只有发送生成代码时，才会执行该过程，所以相关的数据存储在代码表的 Switch 入口中：

```
⟨Switch 188⟩≡                                           168
  struct {
     Symbol sym;
     Symbol table;
     Symbol deflab;
     int size;
     int *values;
     Symbol *labels;
  } swtch;
```

```
⟨generate an indirect jump and a branch table 187⟩+≡    187  187
  code(Switch);
  codelist->u.swtch.table = table;
  codelist->u.swtch.sym = swp->sym;
  codelist->u.swtch.deflab = swp->deflab;
  codelist->u.swtch.size = u - l + 1;
  codelist->u.swtch.values = &v[1];
  codelist->u.swtch.labels = &swp->labels[1];
  if (v[u] - v[l] + 1 >= 10000)
     warning("switch generates a huge table\n");
```

该表通过 emitcode 函数产生。

10.8 return 语句

void 函数的 return 语句中没有表达式，其他函数的 return 语句都含有表达式。如果在不该返回表达式时返回了表达式，就会报错；应该返回表达式而漏掉了表达式，则仅给出警告信息：

```
⟨return statement 189⟩=
  {
     Type rty = freturn(cfunc->type);
     t = gettok();
     definept(NULL);
     if (t != ';')
        if (rty == voidtype) {
           error("extraneous return value\n");
           expr(0);
           retcode(NULL);
        } else
           retcode(expr(0));
     else {
        if (rty != voidtype
        && (rty != inttype || Aflag >= 1))
           warning("missing return value\n");
        retcode(NULL);
     }
     branch(cfunc->u.f.label);
  }
```

retcode 函数对其参数分析树进行类型检查，调用 walk 函数来构造相应的 RET dag。这些将在下面详细介绍。紧接该 dag 的是一个跳转到 cfunc->u.f.label 的跳转指令，而 cfunc->u.f.label 标识了当前函数的结束点；cfunc 指向当前函数在符号表中的入口（可以通过调用 branch 函数去掉该跳转指令）。最后编译后端加入函数尾代码以结束函数的处理——如果有必要，函数尾代码应恢复已保存的值，并将函数返回到调用它的程序。

除非 lcc 编译器指定了 -A 选项，否则上面的代码对于漏掉返回值的函数不会给出警告信息。因为程序员经常将返回整数的函数当作 void 函数来使用，也就是使用

```
f(double x) { ... return; }
```

来取代更准确的

```
void f(double x) { ... return; }
```

所以，对于许多程序来说，漏掉返回值的警告信息太多，以至于会淹没其他更重要的警告信息。

对于 void 函数，retcode 函数除了可能产生一个事件钩子外，不会进行其他操作：

```
⟨stmt.c functions⟩+≡                                    188 191
  void retcode(p) Tree p; {
     Type ty;

     if (p == NULL) {
        if (events.returns)
           ⟨plant event hook for return⟩
        return;
     }
     ⟨retcode 190⟩
  }
```

如果返回类型不是 void，retcode 函数构造并转换 RET 树。该 RET 操作符仅标识了返回值，所以编译后端可以根据目标机器的调用约定将它放在适当的地方，比如规定的寄存器。

如果有返回表达式，retcode 函数就对它进行类型检查，并把它转换为返回类型，就像把该值赋给具有函数返回类型的变量一样，将它包装在相应的 RET 树中：

```
⟨retcode 190⟩≡                                                    190 189
  p = pointer(p);
  ty = assign(freturn(cfunc->type), p);
  if (ty == NULL) {
     error("illegal return type; found '%t' expected '%t'\n",
        p->type, freturn(cfunc->type));
     return;
  }
  p = cast(p, ty);
```

整数、无符号数、浮点数以及双精度数作为返回值时，将返回自身。字符和短整型如同它们出现在参数列表中一样，将转换为返回类型的提升类型。由于没有 RET+P 指令，所以指针类型转换成无符号数，并通过 RET+I 返回。调用这类函数使用的是 CALL+I 指令，它们的值再通过 CVU+P 转换回指针。

```
⟨retcode 190⟩+≡                                                   190 189
  if (retv)
     ⟨return a structure 190⟩
  if (events.returns)
     ⟨plant an event hook for return p⟩
  p = cast(p, promote(p->type));
  if (isptr(p->type)) {
     ⟨warn if p denotes the address of a local⟩
     p = cast(p, unsignedtype);
  }
  walk(tree(RET + widen(p->type), p->type, p, NULL), 0, 0);
```

返回局部变量的地址是一种常见的程序错误，lcc 编译器会对这种简单情况进行检查并给出警告信息；参见练习 10.9。

因为没有 RET+B 指令，返回结构是通过将结构赋值给变量来实现的。如 9.3 节所示，如果 wants_callb 等于 1，该变量就是调用程序的 CALL+B 的第二操作数，也是被调用程序的第一个局部变量，编译后端必须按照具体目标机器的约定传递其地址。如果 wants_callb 等于 0，编译前端就将该变量作为隐含的第一参数来传递，而不会将 CALL+B 传递给编译后端。对于这两种情况，实现复合语句的 compound 函数就将 retv 指向该变量的符号表入口，成为指向该变量的指针。返回结构就是对 *retv 进行赋值：

```
⟨return a structure 190⟩≡                                         190
  {
     if (iscallb(p))
        p = tree(RIGHT, p->type,
           tree(CALL+B, p->type,
              p->kids[0]->kids[0], idtree(retv)),
           rvalue(idtree(retv)));
     else
        p = asgntree(ASGN, rvalue(idtree(retv)), p);
     walk(p, 0, 0);
     if (events.returns)
        ⟨plant an event hook for a struct return⟩
```

```
    return;
}
```

对于 ASGN+B（参见 9.5 节）和 ARG+B（参见 9.3 节），有一种可以减少对

```
return f();
```

进行复制的方法，其中函数 f 与当前函数返回的结构相同，所以当前函数的 retv 可以用作调用函数 f 的临时变量。如果上面的代码经过 iscallb 函数调用发现满足这种条件，就用 retv 取代临时变量对 CALL+B 树进行重新构建。

10.9 管理标号和跳转指令

标号是通过 definelab 函数定义的，而跳转到标号的跳转指令则由 branch 函数生成。这些函数还要合作以删除死的跳转指令，即接在无条件跳转指令或 switch 语句后的跳转指令，避免跳转到跳转指令的跳转指令，并且避免跳转到紧接着的标号的跳转指令。采用的方法与 code 函数中检查不可达代码的方法类似。

definelab 函数将标号定义添加到代码表中，然后检查前一个可执行入口是否是一个跳转到该新标号的跳转指令：

```
⟨stmt.c functions⟩+≡                                    ↑189 192↓
  void definelab(lab) int lab; {
     Code cp;
     Symbol p = findlabel(lab);

     walk(NULL, 0, 0);
     code(Label)->u.forest = newnode(LABELV, NULL, NULL, p);
     for (cp = codelist->prev; cp->kind <= Label; )
        cp = cp->prev;
     while (⟨cp points to a Jump to lab 191⟩) {
        p->ref--;
        ⟨remove the entry at cp 192⟩
        while (cp->kind <= Label)
           cp = cp->prev;
     }
  }
```

这里，newnode 函数为 LABELV 构建一个 dag，且 sym[0] 等于 p。for 循环用来向后遍历代码表，直到遇到第一个表示可执行代码的入口，而 while 循环是用来去掉一个或多个跳转到 lab 的跳转指令。如果 *cp 是一个 Jump 入口，那么 cp 就是一个跳转到 lab 的跳转指令，它的节点计算 lab 的地址：

```
⟨cp points to a Jump to lab 191⟩=                        ←191
     cp->kind == Jump
  && cp->u.forest->kids[0]
  && cp->u.forest->kids[0]->op == ADDRGP
  && cp->u.forest->kids[0]->syms[0] == p
```

把这种 Jump 入口从代码表中去掉，就实现了对无用跳转指令的删除：

```
⟨remove the entry at cp 192⟩≡                                    191 192
  cp->prev->next = cp->next;
  cp->next->prev = cp->prev;
  cp = cp->prev;
```

当 definelab 函数删除一条跳转指令时，还要将其目标标号的 ref 域减 1。这样做是因为在构建跳转指令的 dag 时对目标标号的 ref 域进行了加 1 操作：

```
⟨stmt.c functions⟩+≡                                             ↑191 192↓
  Node jump(lab) int lab; {
     Symbol p = findlabel(lab);

     p->ref++;
     return newnode(JUMPV, newnode(ADDRGP, NULL, NULL, p),
        NULL, NULL);
  }
```

jump 函数由 branch 函数调用，在 Jump 入口保存了 JUMPV dag 并将它添加到代码表中。

branch 函数还要删除跳转到跳转指令的跳转指令以及死的跳转指令。首先，它使用一个 Label 占位符将跳转指令添加到代码表中。跳转指令不是标号，但使用 Label 占位符，使代码段 <check for unreachable code> 不会报警，如果代码表中最后一个可执行入口是一个无条件跳转指令，该程序就会报告它。

```
⟨stmt.c functions⟩+≡                                             ↑192 193↓
  static void branch(lab) int lab; {
     Code cp;
     Symbol p = findlabel(lab);

     walk(NULL, 0, 0);
     code(Label)->u.forest = jump(lab);
     for (cp = codelist->prev; cp->kind < Label; )
        cp = cp->prev;
     while ((cp points to a Label ≠ lab 192)) {
        equatelab(cp->u.forest->syms[0], p);
        ⟨remove the entry at cp 192⟩
        while (cp->kind < Label)
           cp = cp->prev;
     }
     ⟨eliminate or plant the jump 193⟩
  }
```

branch 函数的 for 循环退回到占位符之前的第一个可执行入口或 Label 入口。while 循环则寻找具有如下模式的标号 L' 的定义：

```
L':
     goto L
```

这里 goto L 就是占位符处的跳转指令。

```
⟨cp points to a Label ≠ lab 192⟩≡
     cp->kind == Label
  && cp->u.forest->op == LABELV
  && !equal(cp->u.forest->syms[0], p)
```

如果 L' ≠ L，L' 就等价于 L；跳转到 L' 的指令可以直接跳转到 L，这样 L' 的 Label 入口就可以删除。

```
⟨stmt.c functions⟩+≡                                    192 193
  void equatelab(old, new) Symbol old, new; {
     old->u.l.equatedto = new;
     new->ref++;
  }
```

这段程序为 old 构建一个同义符号 new。在代码生成过程中，对 old 的引用可以替换成由 equatedto 域形成的列表末尾的标号。因为 old 等价于 new 后，new 可能等价于其他符号，所以将这些域构成一个列表。ref 域负责计算跳转指令或其他标号的 u.l.equated 域对该标号的引用次数，所以 equatelab 函数要对 new->ref 加 1。

当两个标号相等时，这些同义符号使得条件测试很复杂。当 L' 等于跳转的目标时，代码段 <cp points to a Label ≠ lab> 一定会失败，而删除如下代码就不会出错，而不管它是多么无意义：

```
top:
  goto top;
```

仅仅测试 L' 是否等于目标 p 还不够；如果 L' 等于 p 或者等于任何一个与 p 同义的符号，那么这两个标号就是等价的。equal 函数实现了这种更复杂的条件测试：

```
⟨stmt.c functions⟩+≡                                    193
  static int equal(lprime, dst) Symbol lprime, dst; {
     for ( ; dst; dst = dst->u.l.equatedto)
        if (lprime == dst)
           return 1;
     return 0;
  }
```

如果 cp 以 Jump 或 Switch 结束，那么该分支就是不可达的，所以占位符就可以删除。否则，占位符就变成一个 Jump:

```
⟨eliminate or plant the jump 193⟩=                      192
  if (cp->kind == Jump || cp->kind == Switch) {
     p->ref--;
     codelist->prev->next = NULL;
     codelist = codelist->prev;
  } else {
     codelist->kind = Jump;
     if (cp->kind == Label
     &&  cp->u.forest->op == LABELV
     &&  equal(cp->u.forest->syms[0], p))
        warning("source code specifies an infinite loop");
  }
```

将对像上面所示的无限循环发出警告信息。

深入阅读

Baskett（1978）描述了为什么要进行循环的生成代码的布局。

lcc 编译器的执行点可以用来生成调试器的符号表以及用于性能执行剖面的生成。Ramsey 和 Hanson（1992）介绍了可重定目标的调试器 ldb 如何使用执行点来定位断点以及如何为搜索调试器符号表提供开始点。Ramsey（1993）还详细介绍了如何使用关键接口函数来生成符号表数据，描述了如何使用 lcc 来计算调试时输入的 C 语言表达式。Fraser and Hanson（1991b）介绍了 lcc 的与目标机器无关的性能执行剖面的实现方法，该功能通过 -b 选项来实现。

许多论文和编译器方面的教材都介绍了如何生成 switch 语句的选择代码。Hennessy, Mendelsohn（1982）和 Bernstein（1985）介绍了与 lcc 编译器类似的技术。贪婪算法可以按线性时间复杂度将 case 值分组到密集表格中，但并不保证这种分组的表格数最少。Kannan and Proebsting（1994）用一页论文给出了一种实现这种功能的简单二次项算法。

练习

10.1 实现 do 语句。

10.2 实现 while 语句。

10.3 实现下面的函数：

```
⟨stmt.c prototypes⟩=
  static int foldcond ARGS((Tree e1, Tree e2));
```

它会被 forstmt 函数调用。提示：构建一棵分析树，在适当的时候，用 e1 替换测试表达式 e2 的左操作数。如果该分析树的操作数都是常量，simplify 函数就会返回一棵 CNST 树，CNST 树决定循环体是否至少执行一次。

10.4 在 <case label> 代码段中有一个 while 循环，但在 case 标号的文法中并没有重复构件。请解释原因。

10.5 证明 swgen 函数中分组算法运行的时间复杂度关于 n 是线性的，n 为 case 值的数目。

10.6 下面是 swgen 函数分组算法的另一种实现方法（由 Arthur Watson 提出）。

```
while (n > 0) {
   float d = den(n-1, k);
   if (d < density
   || k < swp->ncases - 1 && d < den(n, k+1))
      break;
   n--;
}
```

与 10.7 节 swgen 使用的贪婪算法的区别在于，如果一个表格可以和 v[k+1] 构成一个更密集的表格，则该表格和它的前驱不进行合并。例如，当 density 等于 0.5 时，贪婪算法就将值 1、6、7、8、11 和 15 分为 3 个表格：{1，6~8}、{11} 和 {15}；而这种向前看的方法给出了两个表格：{1}、{6 ~ 8，11，15}。分析并解释这种方法。你能证明在什么条件下它可以得到比贪婪算法更少的表格？

10.7 用 Kannan and Proebsting（1994）介绍的最优分组算法来改写 swgen 函数。当 density 等于 0.5 时，该最优算法将值 1、6、7、8、9、10、15 和 19 分为两个表格：{1}、{6 ~ 10，15，19}；而贪婪算法及前面练习中所提到的向前看的方法生成 3 个表格：{1，6 ~ 10}、{15} 和 {19}。请找一个真实的程序，针对该程序的最优分组算法得到的表格比贪婪算法更少。能找到它们在执行时间上的差别吗？

10.8 实现 swcode 函数的 <generate a linear search> 代码段，使得生成的代码具有如下形式：

```
if t1 = v[l] goto L_l
    ⋮
if t1 = v[u] goto L_u
if t1 < v[l] goto lolab
if t1 > v[u] goto hilab
```

使用 cmp 函数来实现条件比较，避免生成不必要的跳转到 lolab 和 hilab 的跳转指令。

10.9 实现 <warn if p denotes the address of a local> 代码，该代码是对 p 进行检查，若 p 是局部变量或参数的地址，则给出警告信息。这个测试可以捕获这类程序错误中的一部分，但不是全部。试给出一个这种方法不能检查出来的错误例子。是否有一种方法能够在编译时捕获所有的错误？在执行时可以吗？

10.10 swcode 函数要接受 b[lb..ub] 中的 ub−lb+1 个表格作为参数，并选择 b[(lb+ub)/2] 处的表格作为生成二分查找树的根节点。其他选择也可以，比如可以选择最长的表格，或者执行剖面（profiling）数据能够提供每一个 case 值出现的频率，这些数据可以查明最可能覆盖 switch 值的表格。此外，我们还可以假定 case 值具有特定的概率分布。假设在 v[b[lb]..b[ub+1]−1] 范围中的所有（甚至那些没有 case 标号的）值都以相同的概率出现。对于这种分布，作为根节点的表格就应当是该范围中最接近平均值的那个 case 值对应的表格。请通过正确计算 swcode 函数的 k 值来实现这种方法。注意：可能没有表格覆盖平均值，所以要选择最接近的一个。

10.11 某些系统支持动态链接和加载。当加载新的代码时，动态链接器必须识别和更新其中所有的可重定位地址。这个过程要花费一些时间，所以位置无关的（position-independent）地址可以为动态链接带来好处，这种地址与使用它的指令在执行过程中程序计数器的值有关系。例如，如果在地址 200 的指令跳转到地址 300 处，传统的可重定位代码将地址 300 存储在指令中，而与位置无关的代码存储的是 300−200，即 100。扩展 lcc 编译器的接口，使它可以为 switch 语句产生位置无关的代码。在第 5 章中定义的接口不能产生位置无关的代码，因为它对 switch 语句使用相同的 defaddress 函数来初始化指针数据，但这肯定不是位置无关的。

第 11 章 |
A Retargetable C Compiler: Design and Implementation

声　明

声明用于规定标识符的类型，定义结构和联合类型，给出函数的代码。对声明的分析可以看成是将类型的文本表示转换为第 4 章介绍的内部表示，并且生成 1.3 节所描述的代码表。

声明是 C 语言中最难分析的一部分，这主要有两个原因。第一，声明的语法是为了说明标识符的用法，例如，声明 int *x[10] 表示 x 是一个含有 10 个整型指针的数组。这里声明的目的就是说明 x 的用法，比如 *x[i] 是整型。遗憾的是，类型信息分散在声明中使得分析更加复杂。

第二个原因来自于对全局变量、局部变量和参数声明的限制。例如，全局和局部变量可以声明为静态的，而参数却不可以。同样，不管是函数声明还是函数定义都可以在文件作用域中出现，但是只有函数声明可以出现在局部作用域中。当然我们可以写一个包含这些限制的语法，但是这会导致出现一系列只有细微差别的产生式。本章将介绍另一种声明语法，即只规定最通用的语法，然后在分析过程中，依据上下文利用语义检查来强制实现相应的限制。由于关于 3 种标识符的重复声明的规则各不相同，所以这种语义检查在任何情况下都是必要的。

本章的内容和代码也体现了上述困难：本章是全书最长的一章，其中的一些代码十分复杂并难以理解，它必须处理很多甚至很细微的问题，而且有一些函数相互递归调用，实现多种功能，所以对它们进行反复解释是不可避免的。

本章前 5 节主要描述了如何对声明进行分析以及将它变换为上一章介绍的前端数据结构中的内部表示。后 4 节介绍了函数定义、复合语句、结束处理（finalization）和 lcc 的主程序。这几节内容对于理解编译前端和后端的交互很有帮助，或许是本书最重要的一部分。例如 11.6 节介绍前端如何调用后端的 function 接口程序，11.9 节显示了针对特定目标机器的接口记录是如何与前端联系起来的。

11.1　转换单元

C 语言的转换单元（translation unit）由一个或多个声明或者函数定义组成。

translation-unit:
　external-declaration { *external-declaration* }

external-declaration:
　function-definition
　declaration

program 函数是转换单元的分析函数，它是 decl.c 文件输出的 5 个函数之一，decl.c 文件用于处理所有的声明。在分析转换单元时允许空输入，对于空输入的情况，lcc 只给出警告信息。

```
⟨decl.c functions⟩=                                          198
  void program() {
    int n;

    level = GLOBAL;
    for (n = 0; t != EOI; n++)
```

```
        if (kind[t] == CHAR || kind[t] == STATIC
        || t == ID || t == '*' || t == '(') {
            decl(dclglobal);
            ⟨deallocate arenas 197⟩
        } else if (t == ';') {
            warning("empty declaration\n");
            t = gettok();
        } else {
            error("unrecognized declaration\n");
            t = gettok();
        }
    if (n == 0)
        warning("empty input file\n");
}
```

函数定义看上去就像一个函数声明后跟一个复合语句，因此 decl 函数既是函数定义又是函数声明的分析函数。decl 函数的参数可以是 dclglobal、dcllocal 或者 dclparam。decl 及其相关函数完全分析完一个标识符的声明之后，将调用 dclX 函数验证标识符，并把它安装在适当的符号表中。这些 dclX 函数将全局变量、局部变量和参数在语义上相互区分开。

上面循环体中的两个 else 分支处理两种错误情况。C 标准规定每个声明至少要声明一个标识符、一个结构或枚举标记，或者一个或多个枚举成员。第一个 else 警告出现的空声明，第二个 else 诊断有语法错误的声明。

声明可以在任何分配区分配空间，函数定义在 PERM 和 FUNC 分配区分配空间，变量声明则使用 STMT 分配区的空间存放表示初始表达式的树。这样，FUNC 和 STMT 分配区的空间都在声明和定义结束后释放。

```
⟨deallocate arenas 197⟩≡                                    197
    deallocate(STMT);
    deallocate(FUNC);
```

11.2 声明

声明的语法如下：

declaration:
  declaration-specifiers init-declarator { , *init-declarator* } ;
  declaration-specifiers ;

init-declarator:
  declarator
  declarator = *initializer*

initializer:
  assignment-expression
  '{' *initializer* { , *initializer* } [,] '}'

declaration-specifiers:
  storage-class-specifier [*declaration-specifiers*]
  type-specifier [*declaration-specifiers*]
  type-qualifier [*declaration-specifiers*]

storage-class-specifier:

```
    typedef | extern | static | auto | register
  type-specifier:
    void
    char | float  | short | signed
    int  | double | long  | unsigned
    struct-or-union-specifier
    enum-specifier
    identifier
  type-qualifier: const | volatile
```

声明规定了标识符的类型和其他属性，比如它的存储类型。定义声明了一个标识符并为该标识符预留存储空间。带初始化的声明就是定义；那些没有初始化的声明被称为尝试性定义（tentative definition），尝试性定义将在 11.8 节中介绍。

声明包含一个或多个说明符，这些说明符可以按任意顺序排列，例如：

```
short const x;
const short x;
const short int x;
int const short x;
```

上述声明都说明 x 是一个不能改变的短整数。存储类型说明符、类型说明符和类型限定符可以以任意顺序出现，但是一种说明符只能出现一次。这种灵活性使得声明说明符的分析函数 specifier 十分复杂。

```
⟨decl.c functions⟩+≡                                    196 200
  static Type specifier(sclass) int *sclass; {
     int cls, cons, sign, size, type, vol;
     Type ty = NULL;

     cls = vol = cons = sign = size = type = 0;
     if (sclass == NULL)
        cls = AUTO;
     for (;;) {
        int *p, tt = t;
        switch (t) {
        ⟨set p and ty 199⟩
        default: p = NULL;
        }
        if (p == NULL)
           break;
        ⟨check for invalid use of the specifier 199⟩
        *p = tt;
     }
     if (sclass)
        *sclass = cls;
     ⟨compute ty 200⟩
     return ty;
  }
```

如果 specifier 函数的参数 sclass 非空，则它指向一个变量，该变量的值就是存储类型的单词编码。局部变量 cls、vol、cons、sign、size 和 type 分别记录了相应说明符的符号码：

```
⟨set p and ty 199⟩=                                          199 198
  case AUTO:
  case REGISTER: if (level <= GLOBAL && cls == 0)
                    error("invalid use of '%k'\n", t);
                 p = &cls;  t = gettok();       break;
  case STATIC: case EXTERN:
  case TYPEDEF:  p = &cls;  t = gettok();       break;
  case CONST:    p = &cons; t = gettok();       break;
  case VOLATILE: p = &vol;  t = gettok();       break;
  case SIGNED:
  case UNSIGNED: p = &sign; t = gettok();       break;
  case LONG:
  case SHORT:    p = &size; t = gettok();       break;
  case VOID: case CHAR: case INT: case FLOAT:
  case DOUBLE:   p = &type; ty = tsym->type;
                            t = gettok();       break;
  case ENUM:     p = &type; ty = enumdcl();     break;
  case STRUCT:
  case UNION:    p = &type; ty = structdcl(t); break;
```

这些变量都初始化为零，只有遇到相应的说明符时变量值才发生改变。因此，若任何一个变量非零，则表明它的说明符已经出现过，这有助于检测错误：

```
⟨check for invalid use of the specifier 199⟩=                198
  if (*p)
    error("invalid use of '%k'\n", tt);
```

一旦所有声明的说明符都分析完，sign、size 和 type 的值就描述相应的类型，函数 enumdcl 和 structdcl 分别分析枚举说明符和结构或联合说明符。

如果 sclass 为空指针，则未指定变量的存储类型，因此将 cls 初始化为存储类型 AUTO，这样就可以捕捉错误。因此，这种灵活性是必需的。因为当分析抽象声明时会调用 specifier 函数，而抽象声明没有存储类型，参见 11.3 节及练习 11.3。

上面的 switch 语句将指针 p 指向相应的局部变量，并且如果单词是类型说明符，则将 ty 设置为 Type。类型定义（typedef）的名字会作为 ID 符号处理，它只能有一个存储类型或限定符。

```
⟨set p and ty 199⟩+≡                                         199 198
  case ID:
    if (istypename(t, tsym) && type == 0
    && sign == 0 && size == 0) {
      use(tsym, src);
      ty = tsym->type;
    p = &type;
    t = gettok();
  } else
    p = NULL;
  break;
```

在对声明说明符分析之后将确定相应的类型 Type，这些类型被编码在 sign、size 和 type 的值中，Type 是 specifier 函数的返回值。默认情况下，

```
⟨compute ty 200⟩=                                          200  198
  if (type == 0) {
     type = INT;
     ty = inttype;
  }
```

short const x 将 x 声明为一个短常数。其他情形根据 sign、size 和 type 来决定相应的类型：

```
⟨compute ty 200⟩+=                                    200 200  198
  if (size == SHORT && type != INT
  ||  size == LONG  && type != INT && type != DOUBLE
  ||  sign && type != INT && type != CHAR)
     error("invalid type specification\n");
  if (type == CHAR && sign)
     ty = sign == UNSIGNED ? unsignedchar : signedchar;
  else if (size == SHORT)
     ty = sign == UNSIGNED ? unsignedshort : shorttype;
  else if (size == LONG && type == DOUBLE)
     ty = longdouble;
  else if (size == LONG)
     ty = sign == UNSIGNED ? unsignedlong : longtype;
  else if (sign == UNSIGNED && type == INT)
     ty = unsignedtype;
```

在 CHAR 的条件测试中明确包含 sign 是必要的，sign 用来区别无符号、有符号字符型与普通字符型，普通字符型是一种特殊类型。经过以上代码，enumdcl 函数或 structdcl 函数计算后，结果 Type 还可能受 const 或 volatile 限定符（或二者同时）的限定。

```
⟨compute ty 200⟩+=                                     200  198
  if (cons == CONST)
     ty = qual(CONST, ty);
  if (vol  == VOLATILE)
     ty = qual(VOLATILE, ty);
```

声明的分析函数 dcl 首先调用 specifier 函数：

```
⟨decl.c functions⟩+=                                   198 202
  static void dcl(dcl)
  Symbol (*dcl) ARGS((int, char *, Type, Coordinate *)); {
     int sclass;
     Type ty, ty1;
     static char stop[] = { CHAR, STATIC, ID, 0 };

     ty = specifier(&sclass);
     if (t == ID || t == '*' || t == '(' || t == '[') {
        char *id;
        Coordinate pos;
        ⟨id, ty1 = the first declarator 201⟩
        for (;;) {
           ⟨declare id with type ty1 202⟩
           if (t != ',')
              break;
```

```
            t = gettok();
            ⟨id, ty1 ← the next declarator 201⟩
         }
      } else if (ty == NULL
      || !(ty is an enumeration or has a tag))
         error("empty declaration\n");
      test(';', stop);
   }
```

函数 dclr 用来分析声明符，我们在下一节详细介绍。第一个以及后续声明符的处理较为简单：

```
⟨id, ty1 ← the next declarator 201⟩= 201
   id = NULL;
   pos = src;
   ty1 = dclr(ty, &id, NULL, 0);
```

函数 dclr 接受基本类型（specifier 函数的结果）并且返回一个 Type、一个标识符，或者参数列表。上述代码中的 ty 就是基本类型，它是函数 dclr 的第一个参数，接着出现的两个参数分别存放标识符和参数列表的变量的地址。函数 dclr 返回一个完整的 Type 值。如果函数 dclr 的第三个参数是空指针，参数列表将不能出现在上下文中。在 11.3 节中可以看到，如果 dclr 的第四个参数非零，dclr 可以分析抽象声明符。在声明标识符时，pos 用来存放声明符的开始在源文件中的位置。

由于 dclr 还要识别出函数定义，而函数定义在文件作用域中只可能同第一个声明符混淆，所以第一个声明符的处理与其他声明符是不同的：

```
⟨id, ty1 ← the first declarator 201⟩= 200
   id = NULL;
   pos = src;
   if (level == GLOBAL) {
      Symbol *params = NULL;
      ty1 = dclr(ty, &id, &params, 0);
      if (⟨function definition? 201⟩) {
         ⟨define function id 202⟩
         return;
      } else if (params)
         exitparams(params);
   } else
      ty1 = dclr(ty, &id, NULL, 0);
```

由于第一个声明符可能是一个函数定义，所以要把参数列表的非空地址作为第三个参数传给 dclr 函数。若该声明符包含一个函数及其参数列表，则 params 将设置为符号表入口数组。如果参数列表非空且又不是函数定义的一部分，则调用 exitparams 函数关闭为该参数列表打开的作用域。否则，由于处理参数列表的分析函数还不能区别函数声明和函数定义，到达参数列表的末尾时，该作用域不会关闭，详细论述参见 11.4 节。

如果第一个声明符说明了一个函数类型并包含一个标识符，并且接下来的单词是复合语句或参数声明列表的开始，那么该声明就是函数定义：

```
⟨function definition? 201⟩= 201
   params && id && isfunc(ty1)
   && (t == '{' || istypename(t, tsym)
   || (kind[t] == STATIC && t != TYPEDEF))
```

decl 通过调用 funcdefn 函数来处理函数定义：

```
⟨define function id 202⟩=                                    201
  if (sclass == TYPEDEF) {
     error("invalid use of 'typedef'\n");
     sclass = EXTERN;
  }
  if (ty1->u.f.oldstyle)
     exitscope();
  funcdefn(sclass, id, ty1, params, pos);
```

这里调用 exitscope 函数来关闭 parameters 打开的作用域，当 funcdefn 函数分析参数声明时会重新打开该作用域。

decl 函数的语义部分相当于声明在声明符中给出的标识符。正如上面所介绍的，decl 的参数 dclX 函数进行语义处理（类型定义除外）。

```
⟨declare id with type ty1 202⟩=                              200
  if (Aflag >= 1 && !hasproto(ty1))
     warning("missing prototype\n");
  if (id == NULL)
     error("missing identifier\n");
  else if (sclass == TYPEDEF)
     ⟨declare id a typedef for ty1 202⟩
  else
     (void)(*dcl)(sclass, id, ty1, &pos);
```

类型定义是最简单的情形，此时语义处理只检查错误的重复声明，将标识符 id 存放到 identifiers 表中并且填充 id 的类型和存储类属性。

```
⟨declare id a typedef for ty1 202⟩=                          202
  {
     Symbol p = lookup(id, identifiers);
     if (p && p->scope == level)
        error("redeclaration of '%s'\n", id);
     p = install(id, &identifiers, level,
        level < LOCAL ? PERM : FUNC);
     p->type = ty1;
     p->sclass = TYPEDEF;
     p->src = pos;
  }
```

3 个 dclX 函数复杂得多。每个 dclX 函数处理一种类别所特有的声明的语义，dclglobal 和 dcllocal 还要分析初始化表达式。dclglobal 函数是 3 个 dclX 函数中最复杂的，它必须处理有效的重复声明。

例如：

```
extern int x[];
int x[10];
```

两次声明 x 都是有效的。第二个声明还将 x 的类型从 (ARRAY(INT)) 变成了 (ARRAY 40 4(INT))。

```
⟨decl.c functions⟩+=                                  200 205
  static Symbol dclglobal(sclass, id, ty, pos)
  int sclass; char *id; Type ty; Coordinate *pos; {
     Symbol p, q;
```

```
    (dclglobal 203)
    return p;
}
```

decl 函数接受任何句法上合法的说明符和声明符组合，因此 dclX 函数必须检查说明符在特定语义环境下的合法性，而且还要检查重复声明。例如 dclglobal 函数，要求存储类型必须是外部的、静态的或者省略存储类型：

```
⟨dclglobal 203⟩=                                    203  203
  if (sclass == 0)
    sclass = AUTO;
  else if (sclass != EXTERN && sclass != STATIC) {
    error("invalid storage class '%k' for '%t %s'\n",
      sclass, ty, id);
    sclass = AUTO;
  }
```

没有存储类型或存储类型非法的全局变量 dclglobal 函数都会给它一个 AUTO 存储类型，这样所有的标识符都有一个非空的存储类型，简化了其他程序的错误检查。

接下来，dclglobal 函数检查错误的重复声明。

```
⟨dclglobal 203⟩+=                                   203 204  203
  p = lookup(id, identifiers);
  if (p && p->scope == GLOBAL) {
    if (p->sclass != TYPEDEF && eqtype(ty, p->type, 1))
      ty = compose(ty, p->type);
    else
      error("redeclaration of '%s' previously declared _
        at %w\n", p->name, &p->src);
    if (!isfunc(ty) && p->defined && t == '=')
      error("redefinition of '%s' previously defined _
        at %w\n", p->name, &p->src);
    ⟨check for inconsistent linkage 204⟩
  }
```

如果两个声明的类型是兼容的，则重复声明是合法的。类型是否兼容由 eqtype 函数决定，结果类型是两种类型的组合。例如上面的例子，经过组合，x 的类型从 (ARRAY(INT)) 变成了 (ARRAY404(INT))。有一些重复声明是合法的，但是重定义（由一个非零 defined 标记和一个初始量标志）在任何情况下都是非法的。

一个标识符具有 3 种链接特性之一。外部链接的标识符可以被其他独立的编译单元所引用。那些具有内部链接特性的标识符只可以被它所在的编译单元引用。参数和局部变量没有链接特性。

没有存储类型或者第一次声明为外部标识符的全局变量具有外部链接特性，而那些声明为静态的全局变量具有内部链接特性。对于其他的声明，省略存储类型或者声明为外部的解释稍有不同。如果是省略存储类型，那么它就具有外部链接特性；但是如果存储类型是外部的，那么其链接类型就与该标识符在文件作用域中前一次声明的链接特性相同。这样，

```
static int y;
extern int y;
```

就是合法的声明并且 y 具有内部链接特性。但是，

```
extern int y;
static int y;
```

是非法声明，第二个声明要求 y 具有内部链接特性，而第一个声明已经说明 y 具有外部链接特性了。多次声明具有相同内部链接特性或者外部链接特性是允许的。

下面这张表总结了 p->sclass（已有声明的存储类型）与 sclass（当前声明的存储类型）之间的关系。AUTO 表示没有存储类型。

		sclass		
		EXTERN	STATIC	AUTO
p->sclass	EXTERN	√	×	√
	STATIC	√	√	×
	AUTO	√	×	√

√表示合法的组合，而 × 表示链接特性错误的组合。dclglobal 函数代码就源自于这张表：

```
⟨check for inconsistent linkage 204⟩=                                    203
  if (p->sclass == EXTERN && sclass == STATIC
  ||  p->sclass == STATIC && sclass == AUTO
  ||  p->sclass == AUTO   && sclass == STATIC)
      warning("inconsistent linkage for '%s' previously "
        declared at %w\n", p->name, &p->src);
```

对于上面的第二个例子，该 if 语句将输出警告信息。

接下来，全局变量被加载到 globals 表中，如果有必要，它的属性值也会被初始化或者重写。

```
⟨dclglobal 203⟩+≡                                          203 203  203
  if (p == NULL || p->scope != GLOBAL) {
      p = install(id, &globals, GLOBAL, PERM);
      p->sclass = sclass;
      if (p->sclass != STATIC) {
          static int nglobals;
          nglobals++;
          if (Aflag >= 2 && nglobals == 512)
              warning("more than 511 external identifiers\n");
      }
      (*IR->defsymbol)(p);
  } else if (p->sclass == EXTERN)
      p->sclass = sclass;
  p->type = ty;
  p->src = *pos;
```

新的全局变量就会传到后端的 defsymbol 接口函数，从而初始化它的 x 域。如果已有的全局变量具有 extern 存储类型，并且该声明没有存储类型或者规定为静态的，则该全局变量的 sclass 就变为 STATIC 或者 AUTO，以确保它在 finalize 函数中定义。若该声明规定为外部的，则对 sclass 的赋值不会有效果。lcc 的选项 -A 使得系统对非 ANSI 标准的用法给出警告。例如，标准 C 并不要求每个 C 语言的实现支持在一个编译单元允许超过 511 个外部标识符，因此，如果使用了 -A-A 选项，lcc 对于太多的外部标识符就会给出警告信息。

标准 C 允许编译器接受如下声明：

```
f() { extern float g(); ... }
int g() { ... }
h() { extern double g(); ... }
```

而不去检测第一个 g 的声明和它的定义（也可视为一种声明）是否相冲突，也不会去检测最后一个 g 的声明和前面两个是否相冲突。从技术上说，每一个 g 的声明都会引入一个不同的标识符，而这些标识符都限制在声明的复合语句的作用域范围内。但是上述 3 个关于 g 的声明都有外部链接特性，这样在执行时必须引用相同的函数。lcc 使用外部声明符表来给出这一类错误的警告信息。dellocal 函数负责将具有外部链接类型的标识符添加到外部声明符表中，而且函数 dellocal 和 dclglobal 都可用来检查不一致：

```
⟨dclglobal 203⟩+≡                                    204 205  203
  {
     Symbol q = lookup(p->name, externals);
     if (q && (p->sclass == STATIC
               || !eqtype(p->type, q->type, 1)))
        warning("declaration of '%s' does not match previous _
           declaration at %w\n", p->name, &q->src);
  }
```

如果有相应的初始化值，dclglobal 函数最后将分析初始化值：

```
⟨dclglobal 203⟩+≡                                    205  203
  if (t == '=' && isfunc(p->type)) {
     error("illegal initialization for '%s'\n", p->name);
     t = gettok();
     initializer(p->type, 0);
  } else if (t == '=')
     initglobal(p, 0);
  else if (p->sclass == STATIC && !isfunc(p->type)
  && p->type->size == 0)
     error("undefined size for '%t %s'\n", p->type, p->name);
```

上面最后一个 else if 语句是对具有内部链接特性和不完全类型的标识符声明进行测试，这种标识符的声明是非法的；一个很好的例证如下：

```
static int x[];
```

如果有初始值或者 initglobal 的第二个参数非零，则 initglobal 函数分析初始值，并且根据它的第一个参数定义该全局变量。initglobal 函数在适当的段中声明该全局变量，分析其初始值，调整其类型，并标记该全局变量为已定义。

```
⟨decl.c functions⟩+≡                                 202 206
  static void initglobal(p, flag) Symbol p; int flag; {
     Type ty;

     if (t == '=' || flag) {
        if (p->sclass == STATIC) {
           for (ty = p->type; isarray(ty); ty = ty->type)
              ;
           defglobal(p, isconst(ty) ? LIT : DATA);
        } else
           defglobal(p, DATA);
        if (t == '=')
           t = gettok();
```

```
        ty = initializer(p->type, 0);
        if (isarray(p->type) && p->type->size == 0)
            p->type = ty;
        if (p->sclass == EXTERN)
            p->sclass = AUTO;
        p->defined = 1;
    }
}
```

initializer 函数分析初始值，在本书中不做详细介绍。如果 p 的类型是一个未知大小的数组，那么初始化会规定数组大小并完成该类型。初始化就是一种定义，这时外部存储类型就等于没有存储类型，因此必要的时候 sclass 是要改变的。这种改变是为了防止 doextern 函数在编译最后为 p 调用后端的 import 函数。

defglobal 函数通过调用相应的接口函数来声明参数的定义。

```
⟨decl.c functions⟩+≡                                    205 207
  void defglobal(p, seg) Symbol p; int seg; {
      p->u.seg = seg;
      swtoseg(p->u.seg);
      if (p->sclass != STATIC)
          (*IR->export)(p);
      (*IR->global)(p);
  }

⟨globals 206⟩≡                                          28
  int seg;
```

具有外部链接特性的标识符通过调用 export 接口函数来声明，global 函数声明实际的定义。swtoseg(n) 函数通过调用 segment 接口函数来转向段 n（BSS、LIT、CODE 或 DATA 段中的一种），但是如果当前段就是段 n，它就不会调用 segment 函数。defglobal 函数在全局的 u.seg 域中记录该段。

11.3 声明符

在 decl 函数中，第一个声明符的分析 <parse the first declarator> 被当作一种特殊情况来看待，它也是识别声明的过程中最复杂的地方之一。下面我们将定义什么是声明符（声明的对象）。声明符的分析非常复杂，主要困难在于声明中基本类型出现在修饰符的前面。例如，int *x 说明 x 的类型为（POINTER(INT)），但是当分析该声明符时，若采取从左至右的顺序进行类型构建，则会产生无意义的类型（INT(POINTER)）。操作符 [ ] 和（ ）的优先级也会导致类似的困难。例如：

```
int *x[10], *f();
```

这里 x 和 f 的类型分别是：

```
(ARRAY 10 (POINTER (INT)))
(POINTER (FUNCTION (INT)))
```

* 在单词流中的位置相同，但是在类型表示中却处于不同的位置。

这些例子都说明，在分析过程中构建一个临时的逆序类型是比较容易的，所以 dclr 函数首先构建逆序类型，然后从后向前遍历该逆序类型以构建相应的顺序类型。dclr 函数的第一个参数是基本类型，即 specifier 函数返回的类型值。

```
(decl.c functions)+≡                                    206 207
  static Type dclr(basety, id, params, abstract)
  Type basety; char **id; Symbol **params; int abstract; {
     Type ty = dclr1(id, params, abstract);
   for ( ; ty; ty = ty->type)
      switch (ty->op) {
      case POINTER:
         basety = ptr(basety);
         break;
      case FUNCTION:
         basety = func(basety, ty->u.f.proto,
            ty->u.f.oldstyle);
         break;
      case ARRAY:
         basety = array(basety, ty->size, 0);
         break;
      case CONST: case VOLATILE:
         basety = qual(ty->op, basety);
         break;
      }
    if (Aflag >= 2 && basety->size > 32767)
      warning("more than 32767 bytes in '%t'\n", basety);
    return basety;
}
```

dclr1 函数分析声明符并返回一个逆序类型，根据这个类型，dclr 就可以构建并返回一个正常的 Type。参数 id 和 params 设置为声明的标识符和参数列表。练习 11.3 将介绍 abstract 参数。dclr1 函数用 Type 结构存放逆序类型的各元素，调用 tnode 函数为各元素分配存储空间并进行初始化：

```
(decl.c functions)+≡                                    207 208
  static Type tnode(op, type) int op; Type type; {
     Type ty;

     NEW0(ty, STMT);
     ty->op = op;
     ty->type = type;
     return ty;
  }
```

dclr1 是声明符的分析函数。声明符的语法格式为：

declarator:
  pointer direct-declarator [*suffix-declarator*]

direct-declarator:
  identifier
  '(' *declarator* ')'

suffix-declarator:
  '[' [*constant-expression*] ']'
  '(' [*parameter-list*] ')'

pointer: [* [*type-qualifier*]]

分析声明符与分析表达式类似。"*""("和"["是操作符，而标识符和参数列表是操作数。操作符产生逆序类型的元素，操作数设置 id 或 params 参数。

```
⟨decl.c functions⟩+≡                                    207 211
  static Type dclr1(id, params, abstract)
  char **id; Symbol **params; int abstract; {
     Type ty = NULL;

     switch (t) {
     case ID:                    ⟨ident 208⟩ break;
     case '*': t = gettok(); ⟨pointer 208⟩ break;
     case '(': t = gettok(); ⟨abstract function 210⟩ break;
     case '[': break;
     default:  return ty;
     }
     while (t == '(' || t == '[')
        switch (t) {
        case '(': t = gettok(); { ⟨concrete function 209⟩ }
                  break;
        case '[': t = gettok(); { ⟨array 209⟩ } break;
        }
     return ty;
  }
```

如果 id 非空，则它指向存储标识符的地址，否则表明该声明符没有包含标识符。

```
⟨ident 208⟩≡                                            208
  if (id)
     *id = token;
  else
     error("extraneous identifier '%s'\n", token);
  t = gettok();
```

指针可能夹杂任意数量的 const 或 volatile 限定符。例如：

```
int *const *const volatile *p;
```

声明 p 为一个指向常 volatile 指针的指针，而该常 volatile 指针又指向了一个常指针，这个常指针则指向一个整数。p 和 ***p 是可以改变的，而 *p 和 **p 不能改变，但 *p 是 volatile 的，可以通过某些外部途径改变。dclr1 返回上述声明符的逆序类型：

```
[POINTER [CONST [POINTER [VOLATILE [CONST [POINTER]]]]]]
```

其中方括号标识了逆序类型的元素。dclr 最后返回的类型为：

```
(POINTER (CONST+VOLATILE (POINTER (CONST POINTER (INT)))))
```

分析指针的代码如下：

```
⟨pointer 208⟩≡                                          208
  if (t == CONST || t == VOLATILE) {
     Type ty1;
     ty1 = ty = tnode(t, NULL);
```

```
    while ((t = gettok()) == CONST || t == VOLATILE)
        ty1 = tnode(t, ty1);
    ty->type = dclr1(id, params, abstract);
    ty = ty1;
} else
    ty = dclr1(id, params, abstract);
ty = tnode(POINTER, ty);
```

对 dclr1 的递归调用，使得 dclr1 中的其他部分没必要把它们的逆序类型添加到指针类型上（如果出现了指针）。练习 11.2 将做详细叙述。

dclr1 执行完第一个 switch 语句后，ty 等于指针或函数的逆序类型，或者为空。后缀操作符"["和"("将 ty 包在逆序类型的相应元素中。对于数组的处理如下所示：

```
⟨array 209⟩≡                                                    208
  int n = 0;
  if (kind[t] == ID) {
      n = intexpr(']', 1);
      if (n <= 0) {
          error("'%d' is an illegal array size\n", n);
          n = 1;
      }
  } else
      expect(']');
  ty = tnode(ARRAY, ty);
  ty->size = n;
```

圆括号可能是对声明符分组，也可能是说明函数类型。如果圆括号出现在后缀声明符之中，就肯定是说明函数类型：

```
⟨concrete function 209⟩≡                                        208
  Symbol *args;
  ty = tnode(FUNCTION, ty);
  ⟨open a scope in a parameter list 209⟩
  args = parameters(ty);
  if (params && *params == NULL)
      *params = args;
  else
      exitparams(args);

⟨open a scope in a parameter list 209⟩≡                          209 210
  enterscope();
  if (level > PARAM)
      enterscope();
```

函数类型中的参数列表打开一个新的作用域，因此，在这种情况下需要调用 enterscope 函数。第二次调用 enterscope 函数是为了处理一种特殊情况，即参数列表本身又包含另外一个作用域。例如，在如下声明中：

```
void f(struct T { int (*fp)(struct T { int m; }); } x) {
    struct T { float a; } y;
}
```

f函数的参数列表打开一个新的作用域，定义了结构标记T。该结构唯一的域fp是一个指向函数的指针；该函数的参数列表又打开另外一个新的作用域，并定义了一个不同的结构标记T。这种声明是合法的。但第二行是错误的声明，因为它重复定义了标记T——f的参数x。标记T与f的局部变量y及标记T都在同一个作用域范围中。

对于在顶层参数作用域范围内声明的标识符，lcc使用PARAM作用域，而对于类似于y的标识符使用LOCAL作用域，LOCAL等于PARAM+1。这种分法只是为了使用方便。例如，可以使foreach函数只访问参数。但是，在检查重复声明时，必须检查LOCAL标识符是不是错误地重复声明了PARAM标识符。

上面的例子是不需要检查重复声明的情况之一。代码中嵌套的参数列表的作用域层数至少是PARAM+2。这种作用域的"空洞"避免了错误的重复声明诊断。例如，fp的参数标记T的作用域是PARAM+2，而x的标记T的作用域是PARAM，这样就不会导致重复声明错误。

函数调用或调用enterscope函数而打开的作用域，在合适的地方，必须通过调用相应的exitscope函数来关闭。参数列表可以看作是函数定义的一部分，或者只看作声明的一部分。如果参数列表出现在函数定义中，params为非空，并且未预先定值，则decl函数的调用者必须在适当的时候调用exitscope函数。例如，在decl函数的<id, tyl ← the first declarator>段中，就调用了exitparams函数。exitparams函数检查旧风格的参数列表中是否有错，并调用exitscope函数。如果params是空指针或者已经有参数列表，则该参数列表不能作为函数定义的一部分，必须立即调用exitscope函数。

抽象声明符使得用圆括号分组更加复杂，它的处理将在练习11.3中介绍。

```
⟨abstract function 210⟩=                                    208
  if (abstract
  && (t == REGISTER || istypename(t, tsym) || t == ')')) {
     Symbol *args;
     ty = tnode(FUNCTION, ty);
     ⟨open a scope in a parameter list 209⟩
     args = parameters(ty);
     exitparams(args);
  } else {
     ty = dclr1(id, params, abstract);
     expect(')');
     if (abstract && ty == NULL
     && (id == NULL || *id == NULL))
        return tnode(FUNCTION, NULL);
  }
```

如果dclr函数的第四个参数非零，意味着dclr函数当前分析的是抽象声明符。如果左括号后跟有新风格的参数列表或非空的声明符以及匹配的右括号，那么该左括号标记了一个参数列表。由于抽象声明符不在函数定义中出现，所以在分析完参数列表后可立即调用exitparams函数。

11.4 函数声明符

C语言标准允许函数声明和函数定义包含旧风格或新风格的参数列表，其语法格式为：

parameter-list:
  parameter { , *parameter* } [, ...]
  identifier { , *identifier* }

parameter:
declaration-specifiers declarator
declaration-specifiers [*abstract-declarator*]

旧风格的参数列表其实就是标识符列表，而新风格的参数列表可以是：声明符列表（每个参数一个声明符）；至少一个参数后面接着逗号和省略号（,...），以表示一个参数数目可变的函数；一个类型说明符 void，以说明一个没有参数的函数。在函数声明和定义中，这两种风格以及它们之间的相互作用使得识别和分析参数列表非常复杂。

parameters 函数可以分析这两种风格的参数列表。它将每个参数登记在当前作用域层的 identi-fiers 表中，从上一节可以看到，当前作用域由 parameters 的调用者通过 enterscope 函数建立。parameters 返回一个指向以 null 结尾的符号数组的指针，函数的每个参数都对应数组的一个元素。参数列表的第一个单词可以识别其风格类型：

```
⟨decl.c functions⟩+≡                                    208 212
  static Symbol *parameters(fty) Type fty; {
     List list = NULL;
     Symbol *params;

     if (kind[t] == STATIC || istypename(t, tsym)) {
        ⟨parse new-style parameter list 213⟩
     } else {
        ⟨parse old-style parameter list 211⟩
     }
     if (t != ')') {
        static char stop[] = { CHAR, STATIC, IF, ')', 0 };
        expect(')');
        skipto('{', stop);
     }
     if (t == ')')
        t = gettok();
     return params;
  }
```

parameters 函数还使用参数信息注释函数类型 fty，如下所示。

旧风格的参数只是简单地被收集到 List 中，在识别所有参数后，List 转换为一个以 null 结尾的数组。

```
⟨parse old-style parameter list 211⟩=                    211
  if (t == ID)
     for (;;) {
        Symbol p;
        if (t != ID) {
           error("expecting an identifier\n");
           break;
        }
        p = dclparam(0, token, inttype, &src);
        p->defined = 0;
        list = append(p, list);
        t = gettok();
        if (t != ',')
```

```
            break;
        t = gettok();
    }
    params = ltov(&list, FUNC);
fty->u.f.proto = NULL;
fty->u.f.oldstyle = 1;
```

lcc 调用 delparam 函数，将参数装载到 identifiers 表中。此时参数的类型还是未知的，所以暂且作为缺型。如果参数列表是函数定义的一部分（必须是），在调用 funcdefn 函数处理声明时，这些符号就会丢弃，再重新加入表中。此时将它们加入表中，仅是为了检测重复参数。lcc 通过将 defined 位设为 0 来标识旧风格的参数。函数类型 fty 也记录了它是旧风格的。

在下面会看到，如果参数列表不是处在函数定义中，当处理到其末尾时，新风格的参数在构造完原型之后就退出其作用域。但此时使用旧风格的参数列表则是错误的。例如声明：

```
int (*f)(int a, float b);
int (*g)(a, b);
```

第 1 行是一个合法的新风格声明，声明了类型：

```
(POINTER (FUNCTION (INT) ((INT) (FLOAT))))
```

但第 2 行则是对类型

```
(POINTER (FUNCTION (INT)))
```

非法的旧风格声明，因为其参数列表不是处在一个函数定义的上下文中。exitparams 函数报告了这种错误：

```
⟨decl.c functions⟩+=                                    211 214
  static void exitparams(params) Symbol params[]; {
     if (params[0] && !params[0]->defined)
        error("extraneous old-style parameter list\n");
     ⟨close a scope in a parameter list 212⟩
  }

⟨close a scope in a parameter list 212⟩=                    212
  if (level > PARAM)
     exitscope();
  exitscope();
```

正如练习 2.15 所提到的，ltov 函数返回的数组至少有一个以 null 结尾的元素，所以如果 params 来自于 parameters，那么它一定是非空的。

新风格的参数列表可以有几种变体，更为复杂。参数列表可以包含标识符，也可以不包含，这取决于它是不是处在函数定义中。各种变体都可能以 "，…" 结束，而且仅包含 void 的参数列表既可以在函数定义中出现，也可以在声明中出现。另外，新风格的声明为函数类型提供了一个原型，该原型必须保留下来，用来检查函数调用以及该函数的其他声明的一致性。如果有函数定义，还用于检查其定义。正如 4.5 节所介绍的，一个没有参数的新风格函数的原型长度为零。一个参数数目可变的函数原型至少有两个元素，且最后一个元素的类型是 void。用 void 表示参数数目可变是一种编码技巧（其作用值得怀疑），void 不会出现在源码中，也不会在原型中出现，因此不会与 void 参数相混淆。

```
⟨parse new-style parameter list 213⟩=                                    211
  int n = 0;
  Type ty1 = NULL;
  for (;;) {
    Type ty;
    int sclass = 0;
    char *id = NULL;
    if (ty1 && t == ELLIPSIS) {
      ⟨terminate list for a varargs function 213⟩
      t = gettok();
      break;
    }
    if (!istypename(t, tsym) && t != REGISTER)
      error("missing parameter type\n");
    n++;
    ty = dclr(specifier(&sclass), &id, NULL, 1);
    ⟨declare a parameter and append it to list 213⟩
    if (ty1 == NULL)
      ty1 = ty;
    if (t != ',')
      break;
    t = gettok();
  }
  ⟨build the prototype 214⟩
  fty->u.f.oldstyle = 0;
```

ty1 是第一个参数的 Type，如上所示，用来检查对 void 和省略号的错误使用。每一个参数都是一个声明符，所以使用 specifier 和 dclr 函数中的方法来分析参数，但是参数仅允许是 register 存储类型。如果出现 void 类型，那么它必须单独且第一个出现：

```
⟨declare a parameter and append it to list 213⟩=                         213
  if ( ty == voidtype && (ty1 || id)
  ||  ty1 == voidtype)
    error("illegal formal parameter types\n");
  if (id == NULL)
    id = stringd(n);
  if (ty != voidtype)
    list = append(dclparam(sclass, id, ty, &src), list);
```

省略的标识符被赋予一个整数名字，如果该声明是函数定义的一部分，则 dclparam 函数将对省略标识符的情况报错。

如果参数列表的长度可变，则在列表的末尾加上一个静态分配的符号，该符号名字为空，类型是 void。

```
⟨terminate list for a varargs function 213⟩=                             213
  static struct symbol sentinel;
  if (sentinel.type == NULL) {
    sentinel.type = voidtype;
    sentinel.defined = 1;
  }
  if (ty1 == voidtype)
    error("illegal formal parameter types\n");
  list = append(&sentinel, list);
```

在分析完新风格的参数列表之后，list 将按参数出现的顺序保存参数符号。这些符号构成了 parameters 函数返回的 params 数组，它们的类型构成了该函数类型的原型：

```
⟨build the prototype 214⟩=                                    213
  fty->u.f.proto = newarray(length(list) + 1,
     sizeof (Type *), PERM);
  params = ltov(&list, FUNC);
  for (n = 0; params[n]; n++)
     fty->u.f.proto[n] = params[n]->type;
  fty->u.f.proto[n] = NULL;
```

dclparam 函数对于新风格和旧风格的参数都可以声明。对于每一个参数，lcc 会调用 dclparam 函数两次。第一次是在 parameters 函数中调用，第二次是在 funcdefn 函数中调用。如果参数列表不是处在函数定义中，lcc 则调用 exitscope 函数（在 exitparams 函数中）来撤销 dclparam 在符号表中建立的记录。

```
⟨decl.c functions⟩+≡                                    212 216
  static Symbol dclparam(sclass, id, ty, pos)
  int sclass; char *id; Type ty; Coordinate *pos; {
     Symbol p;

     ⟨dclparam 214⟩
     return p;
  }
```

声明参数与声明全局变量不同，声明参数更简单一些。首先，类型（ARRAY T）和（FUNCTION T）退化为（POINTER T）和（POINTER(FUNCTION T)）：

```
⟨dclparam 214⟩≡                                         214 214
  if (isfunc(ty))
     ty = ptr(ty);
  else if (isarray(ty))
     ty = atop(ty);
```

虽然参数只允许是 register 存储类型，但是 lcc 函数内部使用 auto 来标识非 register 类型的参数。

```
⟨dclparam 214⟩+≡                                    214 215 214
  if (sclass == 0)
     sclass = AUTO;
  else if (sclass != REGISTER) {
     error("invalid storage class '%k' for '%t%s\n",
        sclass, ty, ⟨id 214⟩);
     sclass = AUTO;
  } else if (isvolatile(ty) || isstruct(ty)) {
     warning("register declaration ignored for '%t%s\n",
        ty, ⟨id 214⟩);
     sclass = AUTO;
  }

⟨id 214⟩≡                                                214
  stringf(id ? " %s'" : "' parameter", id)
```

参数只允许声明一次，因而参数的重复声明检查很容易进行：

```
(dclparam 214)+≡                                    214 215  214
  p = lookup(id, identifiers);
  if (p && p->scope == level)
     error("duplicate declaration for '%s' previously _
        declared at %w\n", id, &p->src);
  else
     p = install(id, &identifiers, level, FUNC);
```

最后，dclparam 函数对 p 的其他域进行初始化，检查并处理非法的初始化：

```
(dclparam 214)+≡                                    215  214
  p->sclass = sclass;
  p->src = *pos;
  p->type = ty;
  p->defined = 1;
  if (t == '=') {
     error("illegal initialization for parameter '%s'\n", id);
     t = gettok();
     (void)expr1(0);
  }
```

参数经过声明后，就通过 function 接口函数通知了编译后端，因此将其视为已定义，详细介绍见 11.6 节。

11.5 结构说明符

从语法上来说，结构、联合和枚举说明符都与通过 int、float 等关键字说明的类型是一样的。然而从语义上来说，它们定义了新的类型。结构或联合说明符定义了一个由有名域组成的聚合类型，枚举说明符定义了一个类型以及相关的有名整型常量集合。练习 11.9 将详细介绍枚举说明符。结构和联合说明符的语法如下：

struct-or-union-specifier:
  struct-or-union [*identifier*] '{' *fields* { *fields* } '}'
  struct-or-union identifier

struct-or-union: `struct` | `union`

fields:
  { *type-specifier* | *type-qualifier* } *field* { , *field* } ;

field:
  declarator
  [*declarator*] : *constant-expression*

标识符是结构或者联合的标记，只有当说明符包含域的列表时它才是可选的。如果结构或联合说明符 struct-or-union-specifier 包含域，或者单独出现在声明中并且在同一个作用域中没有相同标记的结构、联合或枚举类型定义，那么它就定义了一种新的类型。语法的最后一类定义是相互递归的结构声明。例如：

```
struct head { struct node *list; ... };
struct node { struct head *hd; struct node *link; ... };
```

上面的定义说明了 head 的一个实例的 list 域指向链表中的节点，且每一个 node 节点都指向链表头 head。该链表通过 link 指针链接起来。但是，如果在外层作用域已经声明 node 为某结构或联

合的标记，list就成为指向该类型的指针，而不是指向这里声明的node结构类型。如果接下来把指向node的指针赋给list就会发生错误。如果交换上述两行代码来解决list的问题，则head又会遇到相同的问题。解决的办法应该是在定义head之前定义新类型：

```
struct node;
struct head { struct node *list; ... };
struct node { struct head *hd; struct node *link; ... };
```

单独的struct node，在其所处作用域范围内定义了一个新的不完整类型，其标记为node，以屏蔽外层作用域中其他名为node的标记（如果有的话）。如果在同一个作用域中还有一个形如struct node定义的结构标记node，那么后一个声明将不起作用。

structdcl是结构或联合说明符的分析函数，负责处理它们的标记和定义，并调用fields函数来分析各域以及为每个域计算偏移量。除了对域偏移量计算不同外，联合和结构的处理是相同的。

```
⟨decl.c functions⟩+≡                                    214 218
  static Type structdcl(op) int op; {
     char *tag;
     Type ty;
     Symbol p;
     Coordinate pos;

     t = gettok();
     pos = src;
     ⟨structdcl 216⟩
     return ty;
  }
```

structdcl函数首先处理标记，若标记省略，则使用空串：

```
⟨structdcl 216⟩≡                                        216  216
  if (t == ID) {
     tag = token;
     t = gettok();
  } else
     tag = "";
```

如果标记后面有域列表，那么该说明符就定义了一个新的标记：

```
⟨structdcl 216⟩+≡                                       216 217  216
  if (t == '{') {
     static char stop[] = { IF, ',', 0 };
     ty = newstruct(op, tag);
     ty->u.sym->src = pos;
     ty->u.sym->defined = 1;
     t = gettok();
     if (istypename(t, tsym))
        fields(ty);
     else
        error("invalid %k field declarations\n", op);
     test('}', stop);
  }
```

newstruct 函数用来检查标记的重复声明并定义新的类型。如果标记是空的，newstruct 函数就调用 genlabel 函数来生成一个标记，newstruct 函数还用来处理枚举说明符，详见练习 11.9。

如果 struct-or-union-specifier 没有 fields 且该标记已被某类型使用（用 op 的值来标识），则该说明符就指向那个类型。

```
⟨structdcl 216⟩+≡                                    ↑216 ↓217  216
  else if (*tag && (p = lookup(tag, types)) != NULL
  && p->type->op == op) {
     ty = p->type;
     if (t == ';' && p->scope < level)
        ty = newstruct(op, tag);
  }
```

这种情形也可处理上面描述的类型重复定义的异常：如果一个标记在外层作用域中已定义，并且说明符独自出现在一个声明中，则这个说明符就定义了一种新的类型。正如第 3 章中所介绍的，标记有它们自己的名字空间，由 types 表来管理。

如果上面的这些情形未出现，则必定有一个标记且该说明符定义一种新的类型：

```
⟨structdcl 216⟩+≡                                    ↑217  216
  else {
     if (*tag == 0)
        error("missing %k tag\n", op);
     ty = newstruct(op, tag);
  }
  if (*tag && xref)
     use(ty->u.sym, pos);
```

上面最后一个 else 语句处理的情形是：当一个说明符单独出现在一个声明中，且该标记在外层作用域中已经由于其他原因定义过了，例如：

```
enum node { ... };
f(void) {
   struct node;
   struct head { struct node *list; ... };
   struct node { struct head *hd; struct node *link; ... };
   ...
}
```

上面的 else 语句就是处理第二行中的 struct node。

处理结构和联合说明符的复杂性主要在于分析它们所包含的域以及计算其偏移量，尤其是涉及位域的说明符。域必须按照它们在 fields 表中出现的次序存放，偏移量取决于其类型和类型的对齐限制。位域分配在可编址的存储单元中，如果 N 个位域可以存放在一个存储单元中，那么它们必须按照声明次序来存放，但是该次序可以按低位到高位或者高位到低位排列。传统上都是按照地址递增的次序来存放：在低位优先的目标机器上按照低位到高位的次序（从右到左）存放，而在高位优先的目标机器上则是按照高位到地位（从左到右）的次序存放。但是编译器不会强制分割位域，使其跨存储单元，并且它会为位域选择任何存储单元。lcc 使用无符号整数存储位域，这样位域就可以通过整数的读取、存储及屏蔽操作来获取并存储。

图 11-1 展示了一个结构定义及该结构在低位优先的 MIPS 机器中的存放。无符号整数是 32 位的，整数和无符号整数按照 4 个字节的边界对齐。由数组 a 的元素分布可以看出，地址是从右到左依次增加的，图右侧的数据说明了偏移量从上到下依次增加的情况。阴影区域反映了对齐

约束产生的空洞，深阴影区域为 26 位无名位域。这个例子有助于我们理解 fields 的复杂性以及 fields 的分析函数。

```
struct {
   char a[5];
   short s1, s2;
   unsigned code:3, used:1;
   unsigned :26;
   int amt:7, last;
   short id;
} x;
```

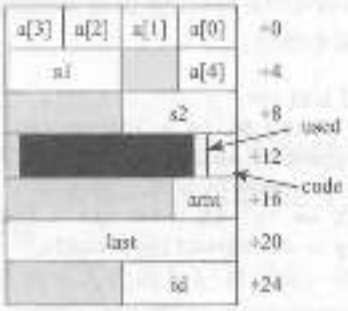


图 11-1 低位优先的结构布局示例

fields 函数分析域列表并且构建一个由 ty->u.sym->u.s.flist fields 引导的 field 结构列表。field 结构在 4.6 节中有详细介绍。它的 name、type 和 offset 域分别给出了域名、类型和它的偏移量，偏移量为距离结构首地址的字节数。对于位域来说，bitsize 给出了位域的位数，lsb 等于位域最低位的位号加 1，所有目标机器上最低位的位号都从 0 开始计算。位域可以用一个非零值的 lsb 来标识。field 结构列表通过 link 域链接在一起。对于图 11-1 所示的例子，下面的表格列出了保存 field 的列表。

name	type	offset	bitsize	lsb
a	chartype	0		
s1	shorttype	6		
s2	shorttype	8		
code	unsignedtype	12	3	1
used	unsignedtype	12	1	4
amt	inttype	16	7	1
last	inttype	20		
id	shortype	24		

fields 函数首先构建一个 fields 列表，然后通过遍历该列表来计算偏移量和位域的位置：

```
⟨decl.c functions⟩+≡                                    216 223
  static void fields(ty) Type ty; {
     { ⟨parse fields 218⟩ }
     { ⟨assign field offsets 220⟩ }
  }
```

分析 fields 列表时，lcc 调用 specifier 函数处理域的说明符，然后分析每个域：

```
⟨parse fields 218⟩≡                                     218
  int n = 0;
  while (istypename(t, tsym)) {
     static char stop[] = { IF, CHAR, '}', 0 };
     Type ty1 = specifier(NULL);
     for (;;) {
        Field p;
        char *id = NULL;
        ⟨parse one field 219⟩
```

```
        n++;
        if (Aflag >= 2 && n == 128)
            warning("more than 127 fields in '%t'\n", ty);
        if (t != ',')
            break;
        t = gettok();
    }
    test(';', stop);
}
```

变量 n 表示域的数目，主要用于当声明的域的数目超过标准所允许的最大数目时发出警告，该功能可以通过 lcc 的 -A 选项进行选择。

域的分析类似于分析声明中的声明符，dclr 函数完成了大部分工作：

```
⟨parse one field 219⟩=                                    219  218
    p = newfield(id, ty, dclr(ty1, &id, NULL, 0));
```

newfield 函数分配一个 field 结构，使用 dclr 函数返回的 id 值和 Type 类型初始化 field 结构的 name 域和 type 域，清除其他域，并将其添加到 ty->u.sym->u.s.flist 中。newfield 从头至尾访问列表以检查域名是否重复。

接下来如果是冒号表示了当前域是一个位域，fields 函数还必须检查域的类型，分析域的宽度并检查域宽的合法性：

```
⟨parse one field 219⟩+=                                   219 220  218
    if (t == ':') {
        if (unqual(p->type) != inttype
        &&  unqual(p->type) != unsignedtype) {
            error("'%t' is an illegal bit-field type\n",
                p->type);
            p->type = inttype;
        }
        t = gettok();
        p->bitsize = intexpr(0, 0);
        if (p->bitsize > 8*inttype->size || p->bitsize < 0) {
            error("'%d' is an illegal bit-field size\n",
                p->bitsize);
            p->bitsize = 8*inttype->size;
        } else if (p->bitsize == 0 && id) {
            warning("extraneous 0-width bit field '%t %s' _
                ignored\n", p->type, id);
            p->name = stringd(genlabel(1));
        }
        p->lsb = 1;
    }
```

位域必须是限定或未限定的整型或无符号整型，编译器可以将整数位域当作有符号或无符号整数，lcc 将其处理为有符号的。一个未命名的位域表示填充域，它暂时被赋予唯一的整数作为名字，并像其他域一样添加到列表中，但是在计算完偏移量后，它就会从列表中删除。类似地，lsb 域也暂时被置为 1 以表示是一个位域，在计算完偏移量后，它就会变为正确的值。

除了没有检查缺少域名和非法类型的错误，newfield 函数完成了有关域的所有工作：

```
⟨parse one field 218⟩+≡                                    219 220 218
  else {
     if (id == NULL)
        error("field name missing\n");
     else if (isfunc(p->type))
        error("'%t' is an illegal field type\n", p->type);
     else if (p->type->size == 0)
        error("undefined size for field '%t %s'\n",
           p->type, id);
  }
```

如果一个域或位域声明为 const，则禁止对该域赋值。对结构赋值也不允许。例如，对于定义：

```
struct { int code; const int value; } x, y;
```

x.code 和 y.code 的值可以改变，但是 x.value 和 y.value 的值是不能改变的。对于类似 x=y 的赋值也是非法的，asgntree 函数中通过检查结构类型的 cfields 标记来发现这些错误。cfields 标记以及记录可变域的 vfields 标记都在分析域时设置：

```
⟨parse one field 218⟩+≡                                    220 218
  if (isconst(p->type))
     ty->u.sym->u.s.cfields = 1;
  if (isvolatile(p->type))
     ty->u.sym->u.s.vfields = 1;
```

至此，图 11-1 中的例子的域列表有 9 个元素；在 218 页表格中列出的 8 个元素再加上在 used 与 amt 之间的一个 bitsize 等于 26 的域。code、used 和 amt 元素的 lsb 域都等于 1，而所有的 offset 域都为 0。

接下来，field 函数遍历域列表计算偏移量。该函数同时计算结构的对齐字节数，重建域列表，并删除那些仅起填充作用的 field 结构，这些结构以整数作为名字。

```
⟨assign field offsets 220⟩=                                222 218
  int bits = 0, off = 0, overflow = 0;
  Field p, *q = &ty->u.sym->u.s.flist;
  ty->align = IR->structmetric.align;
  for (p = *q; p; p = p->link) {
     ⟨compute p->offset 221⟩
     if (p->name == NULL
     || !('1' <= *p->name && *p->name <= '9')) {
        *q = p;
        q = &p->link;
     }
  }
  *q = NULL;
```

off 等于 p 所指向的域之前的所有域（不包含 p 所指的域）占用的字节数的累加和。bits 是 p 之前的位域序列占用的超出 off 个字节的位数加 1。这样，如果前一个域是一个位域，则 bits 非零，而且它绝对不会超过 unsignedtype->size 的值。fields 函数必须能够处理偏移量计算时的溢出情况。它使用宏 add 来实现对 off 的增加：

```
⟨decl.c macros⟩=                                                    221
  #define add(x,n) (x > INT_MAX-(n) ? (overflow=1,x) : x+(n))
  #define chkoverflow(x,n) ((void)add(x,n))
```

如果 x+n 溢出，宏 chkoverflow 通过调用宏 add 来设置 overflow 标志。如果在 fields 函数处理结束后 overflow 等于 1，则说明该结构太大。

如果域出现在联合中，则定义其偏移量为 0：

```
⟨compute p->offset 221⟩=                                            221 220
  int a = p->type->align ? p->type->align : 1;
  if (p->lsb)
    a = unsignedtype->align;
  if (ty->op == UNION)
    off = bits = 0;
```

变量 a 的值表示的是域的对齐字节数。如果有必要，它也可用来增加结构的对齐字节数，它还用来按照对齐边界要求对 off 的值进行向上舍入：

```
⟨compute p->offset 221⟩+=                                           221 221 220
  else if (p->bitsize == 0 || bits == 0
  || bits - 1 + p->bitsize > 8*unsignedtype->size) {
     off = add(off, bits2bytes(bits-1));
     bits = 0;
     chkoverflow(off, a - 1);
     off = roundup(off, a);
  }
  if (a > ty->align)
     ty->align = a;
  p->offset = off;

⟨decl.c macros⟩+=                                                   221
  #define bits2bytes(n) (((n) + 7)/8)
```

如果 p 不是一个位域，且它前面的域不在一个无符号整数的中间结束，或者 p 是一个位域，但是又太大，不能放入它之前的位域占用的剩余空间内，那么 off 就必须进行向上舍入。在向上舍入之前，off 必须增加到超过前面的位域所占用的无符号整数，直到碰到一个普通域或者到达链表的末尾，才会计算这些增加的部分。变量 bits 的值就是该空间的位数加 1，所以通过用 bits-1 除以 8 并向上取整就将它转换为字节数。甚至当 bits 为 0 时，这种计算也是正确的。

在图 11-1 中，当处理 s1 域时，off 等于 5 且 bits 等于 0。s1 的对齐数为 2，所以上面的代码将 off 设置为 6，这也就是 s1 的偏移量。当处理 amt 时，off 等于 12，这就是保存 code 和 bits 的无符号整数，bits 为 3+1+26+1=31，amt 需要 7 位，而这不能满足，所以 off 设置为 12+((31-1)+7)/8=16，这正好符合由 a 所指出的按 4 字节边界对齐的要求，位数设置为 0。接着，处理 last 域，off 等于 16 且 bits 为 7+1=8。因为 last 不是一个位域，off 设置为 16+((8-1)+7)/8=17，然后上舍入为 20。

一旦计算并存储了 p 的偏移量，off 就加上 p 的类型大小（位域除外）。如果 p 是一个位域，则计算 p->lsb，bits 加上位域的宽度：

```
⟨compute p->offset 221⟩+=                                           221 220
  if (p->lsb) {
     if (bits == 0)
        bits = 1;
```

```
        if (IR->little_endian)
            p->lsb = bits;
        else
            p->lsb = 8*unsignedtype->size - bits + 1
                   - p->bitsize + 1;
        bits += p->bitsize;
    } else
        off = add(off, p->type->size);
    if (off + bits2bytes(bits-1) > ty->size)
        ty->size = off + bits2bytes(bits-1);
```

bits 等于按地址顺序的位偏移量加 1，但 lsb 不是按地址顺序的，它是距该位域右端的位数加 1。在一个低位优先的目标机上，bits 和 lsb 是相同的。但是在一个具有 32 位无符号整数的高位优先的目标机上，例如，一个宽 m 的位域到右端的位数是 32-(bits-1)-m，这里 bits-1 就是它的前一位域所占的位数。上面代码的最后一条语句更新 ty->size，该代码无论对结构还是对联合都起作用，因为对于联合的域已将 off 设置为 0。对于联合而言，包含由 bits 所占用的空间是很关键的；如果忽略它，那么

```
union { int x; int a:31, b:4; };
```

的大小是 4 而不是 8，因为 b 所占的 4 位记录在 bits 上，将不被计算。

处理 code 域时，off 等于 10，bits 为 0。由于位域必须从一个适合的无符号整数的边界开始，所以上面代码中向上舍入操作使 off 增加为 12 并作为 code 的偏移量。bits 和 code 的 lsb 都设为 1。如果图 11-1 中的例子在一个 32 位高位优先的目标机上编译，code 的 lsb 就等于 32-1+1=31-0=30；也就是说，在一个高位优先的目标机上到 code 的右端有 29 位。处理到 used 时，off 仍然等于 12，所以它的 lsb 变为 4 并且 bits 变为 5。在 used 和 code 之间的填充使得位数增加为 5+26=31。在为 amt 分配的偏移量为 12 的无符号整数中没有空白，所以如上所述，off 和 bits 分别变为 16 和 0，amt 的 lsb 值变为 1。

结构还可以出现在数组中；一个结构的结束地址边界必须是具有最严格的对齐字节数的域的对齐字节数的整数倍，这样便可以通过增加指针值，使指针从指向数组元素 n 变为指向数组元素 n+1。例如，如果一个结构含有一个 double 类型的域且 double 的对齐字节数为 8，则该结构的对齐字节数也为 8。如上所示，fields 函数负责处理结构的对齐字节数，ty->align 等于或大于其域的对齐字节数，但是如果需要，还必须填充该结构使其为对齐字节数的整数倍：

```
(assign field offsets 220)+≡                                  220  218
    chkoverflow(ty->size, ty->align - 1);
    ty->size = roundup(ty->size, ty->align);
    if (overflow) {
        error("size of '%t' exceeds %d bytes\n", ty, INT_MAX);
        ty->size = INT_MAX&(~(ty->align - 1));
    }
```

对于图 11-1 的例子，<assign field offsets> 中的循环结束时 ty->size 等于 26，最后 off 的值并不是 4（ty->align 的值）的倍数，最后这段代码使得 ty->size 增加到了 28。

11.6 函数定义

函数定义就是一段没有分号结束符的声明，后面接着一些复合语句。在旧风格的函数定义中，还可以插入一些声明列表。

function-definition:
  declaration-specifiers declarator { *declaration* }
    compound-statement

funcdefn 函数分析函数定义。当遇到一个函数定义时，decl 函数就会调用 funcdefn。

```
⟨decl.c functions⟩+≡                                    218 224↓
  static void funcdefn(sclass, id, ty, params, pt) int sclass;
  char *id; Type ty; Symbol params[]; Coordinate pt; {
     int i, n;
     Symbol *callee, *caller, p;
     Type rty = freturn(ty);

     ⟨funcdefn 223⟩
  }
```

funcdefn 函数需要处理许多事情。它必须分析旧风格函数中可选的声明，使得新风格的函数声明与旧风格的函数定义相一致，并对编译前端进行初始化，以分析构成函数的代码表的复合语句。一旦处理到复合语句，当编译后端调用 gencode 和 emitcode 函数时，funcdefn 必须完成对代码表的转换，为接口过程 function 安排正确的参数，在函数的代码生成后还要重新初始化编译前端。

funcdefn 函数的参数 sclass、id 和 ty 分别给出了存储类型、函数名和函数类型（由 decl 函数分析的函数声明符得到）。pt 是该声明符开始的源代码位置，params 是由 parameters 函数构建的符号数组——每个参数对应一个符号。如果参数列表以一个省略号结束，则还有一个未命名的符号。因为该未命名符号仅用于函数原型中，funcdefn 函数开始时会排除这个额外的未命名符号，funcdefn 函数还检查非法返回类型：

```
⟨funcdefn 223⟩≡                                         223 223↓
  if (isstruct(rty) && rty->size == 0)
     error("illegal use of incomplete type '%t'\n", rty);
  for (n = 0; params[n]; n++)
     ;
  if (n > 0 && params[n-1]->name == NULL)
     params[--n] = NULL;
```

params 数组帮助 funcdefn 函数构建两个类似的指向符号表入口的指针数组。callee 是函数自身可见的参数入口数组，而 caller 是函数调用者可见的参数入口数组。通常，这两个数组中相应的入口是相同的，但当因为参数提升使得调用者的参数类型与被调用函数的参数类型不同时，它们也会不同，参见 1.3 节。调用者和被调用者的参数的存储类型也可能不相同，例如，被调用函数的某个参数声明为 register 存储类型，但是调用者却通过栈来传递该参数。构建 callee 和 caller 的详细情况还取决于函数定义是旧风格还是新风格的：

```
⟨funcdefn 223⟩+≡                                        223 226↓ 223
  if (ty->u.f.oldstyle) {
     ⟨initialize old-style parameters 224⟩
  } else {
     ⟨initialize new-style parameters 224⟩
  }
  for (i = 0; (p = callee[i]) != NULL; i++)
     if (p->type->size == 0) {
        error("undefined size for parameter '%t %s'\n",
           p->type, p->name);
```

```
        caller[i]->type = p->type = inttype;
    }
```

与旧风格的函数定义相比，新风格的函数定义容易处理一些。由于 parameters 函数已经做了大部分的处理工作，所以 params 可以直接用作 callee。调用者的参数一般是相应的被调用者参数的副本，但是它们的类型可以被提升，它们的存储类型可以是 AUTO 以表明它们是通过内存传递的。

```
⟨initialize new-style parameters 224⟩=                                    225
    callee = params;
    caller = newarray(n + 1, sizeof *caller, FUNC);
    for (i = 0; (p = callee[i]) != NULL && p->name; i++) {
        NEW(caller[i], FUNC);
        *caller[i] = *p;
        caller[i]->type = promote(p->type);
        caller[i]->sclass = AUTO;
        if ('1' <= *p->name && *p->name <= '9')
            error("missing name for parameter %d to _
                function '%s'\n", i + 1, id);
    }
    caller[i] = NULL;
```

parameters 函数使用参数编号来表示参数中省略的标识符，所以 funcdefn 函数必须检查这类标识符。在声明中标识符可以省略，但是在函数定义中不允许省略标识符。

对于旧风格的函数定义，parameters 简单收集了参数列表中的标识符并且检查了副本。funcdefn 函数必须分析它们的声明符，并把结果的标识符与 params 中的标识符进行匹配。funcdefn 函数直接将 params 用作 caller，并复制一个作为 callee，然后调用 decl 函数来分析这些声明。

```
⟨initialize old-style parameters 224⟩=                          224 223
    caller = params;
    callee = newarray(n + 1, sizeof *callee, FUNC);
    memcpy(callee, caller, (n+1)*sizeof *callee);
    enterscope();
    while (kind[t] == STATIC || istypename(t, tsym))
        decl(dclparam);
```

对参数声明的分析将为每一个标识符在 identifiers 表中增加一个符号表入口。这些声明可能忽略掉数参数，也可能声明不在 callee 中的标识符。funcdefn 函数通过访问每一个具有 PARAM 作用域的符号来检查第二种情况，并将 callee 指向那个符号：

```
⟨initialize old-style parameters 224⟩+=                    224 225 223
    foreach(identifiers, PARAM, oldparam, callee);

⟨decl.c functions⟩+=                                              223 229
    static void oldparam(p, cl) Symbol p; void *cl; {
        int i;
        Symbol *callee = cl;

        for (i = 0; callee[i]; i++)
            if (p->name == callee[i]->name) {
                callee[i] = p;
                return;
            }
```

```
        error("declared parameter '%s' is missing\n", p->name);
    }
```

处理完旧风格的声明后，callee 中的一些入口就指向了真实的参数符号，其他的则指向 param-eters 函数创建的占位符号。对于 defined 标志等于 0 的符号，如果它们显式地声明为参数，则它们此时会初始化。同时，相应的 caller 符号被 identifiers 中的真实符号覆盖。

```
(initialize old-style parameters 224) +=                    224 225 223
    for (i = 0; (p = callee[i]) != NULL; i++) {
        if (!p->defined)
            callee[i] = dclparam(0, p->name, inttype, &p->src);
        *caller[i] = *p;
        caller[i]->sclass = AUTO;
        if (unqual(p->type) == floattype)
            caller[i]->type = doubletype;
        else
            caller[i]->type = promote(p->type);
    }
```

调用旧风格函数时会遇到默认参数的困扰，所以 caller 函数中的参数类型做了相应的改变。例如：在

```
f(c,x) char c; float x; { ... }
```

中，callee 的两个参数的类型分别是 CHAR 和 FLOAT，但 caller 的参数类型是 INT 和 DOUBLE。如 1.3 节所示，这种差异导致在函数 f 的入口处调用者的参数值需要赋值给被调用函数的参数。

标准 C 允许混合使用旧风格的定义和新风格的声明，本书的代码反映了这一点，但是函数定义必须与声明相一致，反之亦然。如果一个新风格的函数声明后接着一个旧风格的函数定义，则该函数就会被认为是一个新风格的函数，并且旧风格的函数定义所提供的参数列表的类型必须与函数声明一致。

```
(initialize old-style parameters 224) +=                    225 226 223
    p = lookup(id, identifiers);
    if (p && p->scope == GLOBAL && isfunc(p->type)
    && p->type->u.f.proto) {
        Type *proto = p->type->u.f.proto;
        for (i = 0; caller[i] && proto[i]; i++)
            if (eqtype(unqual(proto[i]),
                unqual(caller[i]->type), 1) == 0)
                break;
        if (proto[i] || caller[i])
            error("conflicting argument declarations for _
                function '%s'\n", id);
    }
```

由于没有兼容的旧风格函数定义，所以新风格的声明不能以“,...”结束。上面的代码用来检查 caller 的类型是否与相应的新风格声明兼容。这样，与上面的 f 函数唯一兼容的声明是：

```
extern int f(int, double);
```

声明

```
extern int f(char, float);
```

的参数类型与上面定义的f函数的参数c和x相同，因此看上去二者兼容。但是，出于兼容的考虑，参数应当是提升后的类型。

如果一个新风格的函数声明跟在一个旧风格的函数定义的后面，则该函数依然是一个旧风格函数，但是如上所示，声明必须是兼容的。为了实现这种检查，lcc对旧风格函数构建一个原型，并通过包含该原型来改变函数的类型。该函数类型的oldstyle标记等于1，所以这种原型仅被eqtype用来进行这样的检查。

```
⟨initialize old-style parameters 224⟩+≡                    225  223
  else {
   Type *proto = newarray(n + 1, sizeof *proto, PERM);
   if (Aflag >= 1)
      warning("missing prototype for '%s'\n", id);
   for (i = 0; i < n; i++)
      proto[i] = caller[i]->type;
   proto[i] = NULL;
   ty = func(rty, proto, 1);
}
```

接下来出现新风格函数声明时，重声明代码将调用eqtype函数，并用该原型检查兼容性。

一旦caller和callee构建好后，funcdefn函数就会为函数本身定义符号，因为其他函数，比如statement函数和retcode函数，需要访问当前函数。设该符号为全局变量：

```
⟨decl.c exported data⟩≡                                    227
  extern Symbol cfunc;
```

函数的其他信息存放在符号的u.f域：

```
⟨function symbols 226⟩≡                                    28
  struct {
     Coordinate pt;
     int label;
     int ncalls;
     Symbol *callee;
  } f;
```

pt表示的是函数入口点的源代码位置，label表示退出点的标号，ncalls表示函数中出现的函数调用的次数，callee域是函数funcdefn的局部变量callee的副本。

```
⟨funcdefn 223⟩+≡                                           223 227  223
  p = lookup(id, identifiers);
  if (p && isfunc(p->type) && p->defined)
     error("redefinition of '%s' previously defined at %w\n",
        p->name, &p->src);
  cfunc = dclglobal(sclass, id, ty, &pt);
  cfunc->u.f.label = genlabel(1);
  cfunc->u.f.callee = callee;
  cfunc->u.f.pt = src;
  cfunc->defined = 1;
  if (xref)
     use(cfunc, cfunc->src);
```

此时，funcdefn 函数为最后处理函数体做好了准备。它为内部标号和语句标号初始化符号表，把 refinc 初始化为 1，将 code 列表设为唯一的 Start 入口，为函数的入口点添加一个执行点，并调用 compound 函数。compound 函数将在下一节中介绍。

```
⟨funcdefn 223⟩+=                                          226 227  223
  labels   = table(NULL, LABELS);
  stmtlabs = table(NULL, LABELS);
  refinc = 1.0;
  regcount = 0;
  codelist = &codehead;
  codelist->next = NULL;
  definept(NULL);
  if (!IR->wants_callb && isstruct(rty))
     retv = genident(AUTO, ptr(rty), PARAM);
  compound(0, NULL, 0);

⟨decl.c data⟩=                                                  229
  static int regcount;

⟨decl.c exported data⟩+=                                        226
  extern Symbol retv;
```

regcount 是显式声明为 register 的局部变量的个数。正如 10.8 节详细叙述的，如果接口标志 wants callb 等于 0，则编译前端就完全实现返回结构的函数。这样就创建了一个指向存储返回值地址的隐式参数，并通过 retv 为该参数传递符号。它也会在调用时传递该参数的值，参见 9.3 节。

随着复合语句的分析，代码表不断增长。当 compound 函数返回时，如果有必要，funcdefn 函数会为返回语句在代码表中增加一棵树。该代码类似于增加了一个跳转：只有当程序控制能够执行到函数尾时，该返回语句才是必要的。

```
⟨funcdefn 223⟩+=                                          227 228  223
  {
     Code cp;
     for (cp = codelist; cp->kind < Label; cp = cp->prev)
        ;
     if (cp->kind != Jump) {
        if (rty != voidtype
        && (rty != inttype || Aflag >= 1))
           warning("missing return value\n");
        retcode(NULL);
     }
  }
  definelab(cfunc->u.f.label);
  definept(NULL);
```

对 definelab 函数的调用将增加一个退出点标号，而 definept 函数则增加一个相应的执行点。lcc 编译器对于以下两种情况给出警告信息：一种是返回类型为非整数的函数采取了隐式返回方式，另一种是具有 -A 选项而非 void 类型的函数采取了隐式返回方式。函数分析的最后一步就是关闭 compound 函数所打开的作用域并检查未引用的参数：

```
(funcdefn 223)+≡                                    227 228  223
  exitscope();
  foreach(identifiers, level, checkref, NULL);
```

checkref 函数将在下一节中详细介绍。

现在函数的代码表已经构造完成（除了在 gencode 函数中进行必要的转交）。funcdefn 函数已准备调用接口过程 function。但是在调用之前，必须根据接口标志 wants_callb 和 wants_argb 的值，对 caller 和 callee 进行两种不同的变换。如果 wants_callb 等于 0，则通过 retv 指针指向的隐含参数必须插入 callee 的开始，其副本也必须插到 caller 的开始：

```
(funcdefn 223)+≡                                    228 228  223
  if (!IR->wants_callb && isstruct(rty)) {
     Symbol *a;
     a = newarray(n + 2, sizeof *a, FUNC);
     a[0] = retv;
     memcpy(&a[1], callee, (n+1)*sizeof *callee);
     callee = a;
     a = newarray(n + 2, sizeof *a, FUNC);
     NEW(a[0], FUNC);
     *a[0] = *retv;
     memcpy(&a[1], caller, (n+1)*sizeof *callee);
     caller = a;
  }
```

如果 wants_argb 等于 0，编译前端将完全实现结构参数，参见 8.8 节和 9.3 节。例如，当 wants_argb 等于 0 时，idtree 函数对于结构参数另外生成一个间接地址，因为参数是该结构的真正地址。但是这种错误在编译后端必须修正。编译后端通过改变 caller 和 callee 的参数类型并且设置一个 structarg 变量来标识这种标识符进行这样的修正。

```
(symbol flags 37)+≡                                 163  28
  unsigned structarg:1;

(funcdefn 223)+≡                                    228 228  223
  if (!IR->wants_argb)
     for (i = 0; caller[i]; i++)
        if (isstruct(caller[i]->type)) {
           caller[i]->type = ptr(caller[i]->type);
           callee[i]->type = ptr(callee[i]->type);
           caller[i]->structarg = callee[i]->structarg = 1;
        }
```

最后，如果有必要，funcdefn 就输出函数并将控制权交给编译后端：

```
(funcdefn 223)+≡                                    228 229  223
  if (cfunc->sclass != STATIC)
     (*IR->export)(cfunc);
  swtoseg(CODE);
  (*IR->function)(cfunc, caller, callee, cfunc->u.f.ncalls);
```

funcdefn 函数结束时输出结果，对未定义的语句标号进行检查，有选择地生成一个函数结束的事件钩子，关闭 PARAM 作用域，并读入函数复合语句的结束括号。

```
(funcdefn 223)+≡                                    ↑228  223
  outflush();
  foreach(stmtlabs, LABELS, checklab, NULL);
  exitscope();
  expect('}');
```

checklab 类似于 checkref，参见练习 11.4。

11.7 复合语句

复合语句的语法如下：

compound-statement:
'{' { *declaration* } { *statement* } '}'

compound 是复合语句的分析函数。它添加一个 Blockbeg 入口到代码表中，然后打开一个新的作用域，分析可选的声明和语句，然后添加一个 Blockend 入口地址到代码表中。compound 函数的参数有循环句柄、switch 句柄和结构化语句的嵌套层数。

```
(decl.c functions)+≡                                ↑224 231↓
  void compound(loop, swp, lev)
  int loop, lev; struct swtch *swp; {
     Code cp;
     int nregs;
     walk(NULL, 0, 0);
     cp = code(Blockbeg);
     enterscope();
     (compound 230)
     cp->u.block.level = level;
     cp->u.block.identifiers = identifiers;
     cp->u.block.types = types;
     code(Blockend)->u.begin = cp;
     if (level > LOCAL) {
        exitscope();
        expect('}');
     }
  }
```

compound 函数会在 statement 函数和 funcdefn 函数中调用。两种调用唯一的不同之处在于：源于 statement 函数的调用需要 compound 关闭作用域。如上节所示，funcdefn 函数会关闭 compound 函数打开的作用域，这样 funcdefn 就可以在关闭作用域之前调用接口过程 function。

compound 函数中的语义处理大多涉及在语句块中声明的局部变量。dcllocal 函数处理每一个局部变量并根据它的：显式存储类型将它添加到下面的一个列表中：

```
(decl.c data)+≡                                     ↑227
  static List autos, registers;
```

没有存储类型的局部变量添加到 autos 列表中，而静态局部变量的处理与全局变量类似。

如果 compound 函数是被 funcdefn 函数调用的，则它必须处理接口标志 wants_callb。当此标志为 1 时，编译后端就会处理返回结构的函数其返回值的传递。编译前端在调用方为该返回值建立存储空间，但它不知道如何将该存储空间的地址传递给被调用函数。它假定编译后端会以一种与目标机器相关的方式传递该地址，并将它存储在第一个局部变量中。所以，compound 函数将生成第一个局部变量，并在 retv 中保存该变量的符号表入口：

```
⟨compound 230⟩=                                          230  229
  autos = registers = NULL;
  if (level == LOCAL && IR->wants_callb
  && isstruct(freturn(cfunc->type))) {
     retv = genident(AUTO, ptr(freturn(cfunc->type)), level);
     retv->defined = 1;
     retv->ref = 1;
     registers = append(retv, registers);
  }
```

即使 retv 为 AUTO 类型它也会添加到 registers 列表中，以确保它作为第一个局部变量传递到编译后端。

编译前端根据 wants_callb 的值选择两种方式之一使用 retv。当 wants_callb 为 1 时，retv 就保存返回值地址的局部变量在符号表中的入口。当 wants_callb 为 0 时，就没有这种局部变量。编译前端会把该地址作为隐含的第一个参数的值来传递，这时，retv 就是该参数的符号表入口地址。至于涉及 retcode 时，retv 就是存放该地址的变量在符号表中的入口，无须关心它是如何得到的。

接下来，compound 函数分析可选块层次中的声明：

```
⟨compound 230⟩+=                                    230 230  229
  expect('{');
  while (kind[t] == CHAR || kind[t] == STATIC
  || istypename(t, tsym) && getchr() != ':')
     decl(dcllocal);
```

compound 调用 getchr 函数来检查一些很少使用但是合法的代码，例如：

```
typedef int T;
f() { T: ...; goto T; }
```

若 istypename(t) 为真，则 T 是一个类型定义，但是在函数 f 中，T 是一个标号。通过查询下一个输入符号，可以避免解释错误。

局部变量分析完后，autos 列表中的那些局部变量就会添加到 registers 列表中，然后它们就转换为以空值结尾的数组，并被赋给 Blockbeg 所指的代码表入口的 u.block.locals 域。

```
⟨compound 230⟩+=                                    230 231  229
  {
     int i;
     Symbol *a = ltov(&autos, STMT);
     nregs = length(registers);
     for (i = 0; a[i]; i++)
        registers = append(a[i], registers);
     cp->u.block.locals = ltov(&registers, FUNC);
  }
```

cp->u.block.locals [0..nregs-1] 是寄存器局部变量，而自动局部变量从 cp->u.block.locals[nregs] 开始。这种存放顺序确保寄存器局部变量在自动局部变量之前通知了编译后端。

接下来 compound 处理语句：

```
⟨compound 230⟩+≡                                    230 231 229
  while (kind[t] == IF || kind[t] == ID)
     statement(loop, swp, lev);
  walk(NULL, 0, 0);
  foreach(identifiers, level, checkref, NULL);
```

编译语句时，idtree 函数就增加语句所使用的标识符的 ref 域。这样，在语句编译结束时，ref 域指明了最常被访问的变量。下面描述的 checkref 函数将引用次数超过 3 次的变量的存储类型改为 REGISTER，除非其地址被使用。compound 函数从 cp->u.block.locals[nregs] 开始按 ref 值降序排列这些局部变量。

```
⟨compound 230⟩+≡                                    231 229
  {
     int i = nregs, j;
     Symbol p;
     for ( ; (p = cp->u.block.locals[i]) != NULL; i++) {
        for (j = i; j > nregs
           && cp->u.block.locals[j-1]->ref < p->ref; j--)
           cp->u.block.locals[j] = cp->u.block.locals[j-1];
        cp->u.block.locals[j] = p;
     }
  }
```

现在，这些局部变量中的一部分具有 REGISTER 存储类型，由于这些变量按照估算的使用频率排序，编译后端可以直接为它们分配寄存器而无须再做分析。在 cp->u.block.locals [0..nregs −1] 中的局部变量被引用的次数可能会比其他变量少，但是由于程序已明确声明它们为寄存器变量，所以它们最先提交给编译后端。

在复合语句结束时，对于 identifiers 表中的每一个符号，compound 都会调用 checkref 函数，checkref 函数所要做的不只是改变存储类型，

```
⟨decl.c functions⟩+≡                                229 232
  static void checkref(p, cl) Symbol p; void *cl; {
     ⟨checkref 231⟩
  }
```

checkref 还要设置可变的局部变量和参数的 addressed 标记，以防止它们被放置到寄存器变量表中：

```
⟨checkref 231⟩≡                                     231 231
  if (p->scope >= PARAM
  && (isvolatile(p->type) || isfunc(p->type)))
     p->addressed = 1;
```

当 lcc 的 -A 选项出现两次时，checkref 函数就会对未引用的静态变量、参数以及局部变量给出警告信息：

```
⟨checkref 231⟩+≡                                    231 232 231
  if (Aflag >= 2 && p->defined && p->ref == 0) {
     if (p->sclass == STATIC)
        warning("static '%t %s' is not referenced\n",
           p->type, p->name);
     else if (p->scope == PARAM)
        warning("parameter '%t %s' is not referenced\n",
```

```
            p->type, p->name);
    else if (p->scope >= LOCAL && p->sclass != EXTERN)
        warning("local '%t %s' is not referenced\n",
            p->type, p->name);
}
```

除了上面所提到的情况外，还有许多参数或者局部变量的存储类型需要从 AUTO 变为 REGISTER。仅当没有被明确声明为寄存器变量时，参数的存储类型才会改变。否则的话，有可能把寄存器分配给参数，而不是想要分给的局部变量。

```
⟨checkref 231⟩+≡                                    231 232  231
  if (p->sclass == AUTO
  && (p->scope  == PARAM && regcount == 0
   || p->scope  >= LOCAL)
  && !p->addressed && isscalar(p->type) && p->ref >= 3.0)
     p->sclass = REGISTER;
```

在任何块中明确声明为寄存器类型的局部变量都会使 dcllocal 函数增加 regcount 的值。

checkref 函数还帮助管理 externals 表。如下所示，dcllocal 函数将那些声明为 externals 的局部变量安装到 externals 和 identifiers 中。当局部变量超出了作用域范围时，checkref 函数就会把 identifiers 符号中 ref 域的值加到它的 externals 符号的 ref 域：

```
⟨checkref 231⟩+≡                                    232 232  231
  if (p->scope >= LOCAL && p->sclass == EXTERN) {
     Symbol q = lookup(p->name, externals);
     q->ref += p->ref;
  }
```

这样，externals 表中标识符的 ref 值就汇总了所有函数中对该标识符的引用。

最后，在编译结束时，finalize 函数调用 checkref 查看当前作用域层数，以检查未定义的静态变量和函数。

```
⟨checkref 231⟩+≡                                    232  231
  if (level == GLOBAL && p->sclass == STATIC && !p->defined
  && isfunc(p->type) && p->ref)
     error("undefined static '%t %s'\n", p->type, p->name);
```

lcc 对那些声明过但从未定义和未引用的静态函数不会报错，因为标准 C 并没有规定这种声明是错误的。

当 compound 为每个局部变量调用 decl 函数时，decl 函数调用最后一个 dclX 函数——dcllocal 函数。

```
⟨decl.c functions⟩+≡                                231 236
  static Symbol dcllocal(sclass, id, ty, pos)
  int sclass; char *id; Type ty; Coordinate *pos; {
     Symbol p, q;

     ⟨dcllocal 233⟩
     return p;
  }
```

与 dclglobal 和 dclparam 函数类似，dcllocal 函数开始先检查无效的存储类型：

```
⟨dcllocal 233⟩≡                                          233 232
  if (sclass == 0)
    sclass = isfunc(ty) ? EXTERN : AUTO;
  else if (isfunc(ty) && sclass != EXTERN) {
    error("invalid storage class '%k' for '%t %s'\n",
      sclass, ty, id);
    sclass = EXTERN;
  } else if (sclass == REGISTER
  && (isvolatile(ty) || isstruct(ty) || isarray(ty))) {
    warning("register declaration ignored for '%t %s'\n",
      ty, id);
    sclass = AUTO;
  }
```

局部变量可以具有任何存储类型，但函数一定没有存储类型或者是 extern。可变局部变量和那些具有聚合类型的局部变量可以声明为寄存器类型的，但是 lcc 将它们看作自动类型的。

接下来，dcllocal 函数对重复声明进行检查：

```
⟨dcllocal 233⟩+≡                                  233 233 232
  q = lookup(id, identifiers);
  if (q && q->scope >= level
  ||  q && q->scope == PARAM && level == LOCAL)
    if (sclass == EXTERN && q->sclass == EXTERN
    && eqtype(q->type, ty, 1))
      ty = compose(ty, q->type);
    else
      error("redeclaration of '%s' previously "
        declared at %w\n", q->name, &q->src);
```

对于参数和函数复合语句中的局部变量，lcc 使用不同的作用域，但是标准 C 将它们的作用域看成是一个。这样，如果在同一作用域中已经有一个标识符或者参数具有相同的名字，再声明一个作用域为 LOCAL 的同名局部变量，该局部变量的声明就是重复声明。代码：

```
f() { extern int x[]; extern int x[10]; ... }
```

说明了对于一个局部变量允许多次声明的情况，即当它们声明为 extern 时。这里，第二个 extern 声明给出了 x 的更多信息，即它的大小。

接下来 dcllocal 函数安装标识符，初始化它的域，并根据它的存储类型进行不同的处理。

```
⟨dcllocal 233⟩+≡                                  233 235 232
  p = install(id, &identifiers, level, FUNC);
  p->type = ty;
  p->sclass = sclass;
  p->src = *pos;
  switch (sclass) {
  case EXTERN:   ⟨extern local 234⟩ break;
  case STATIC:   ⟨static local 234⟩ break;
  case REGISTER: ⟨register local 234⟩ break;
  case AUTO:     ⟨auto local 234⟩ break;
  }
```

自动的和寄存器局部变量比较简单，它们只是简单地添加到相应的列表中：

```
⟨register local 234⟩=                                    233
  registers = append(p, registers);
  regcount++;
  p->defined = 1;

⟨auto local 234⟩=                                        233
  autos = append(p, autos);
  p->defined = 1;
```

regcount是在函数的任何地方显式声明为寄存器类型的局部变量的数目，在上面的 checkref 函数中使用。与全局变量不同的是，局部变量在 gencode 函数中传到编译后端之前，即在它声明时，其 defined 标志被设置了。局部变量以这种方式处理是因为它们在给定的作用域中只能声明一次，而且它们的声明也就是定义。

处理静态局部变量的大部分工作就是进行可选的初始化，这与 initglobal 函数中处理全局变量是相同的：

```
⟨static local 234⟩=                                      233
  (*IR->defsymbol)(p);
  initglobal(p, 0);
  if (!p->defined)
     if (p->type->size > 0) {
        defglobal(p, BSS);
        (*IR->space)(p->type->size);
     } else
        error("undefined size for '%t %s'\n",
           p->type, p->name);
  p->defined = 1;
```

如果没有初始化，当 initglobal 函数返回时，p->defined 的值就等于 0。dcllocal 函数必须为静态局部变量分配存储空间。与没有初始化的全局变量一样，未初始化的静态变量在 BSS 段中定义。

声明为 extern 类型的局部变量要受到第 204 页表中 EXTERN 一栏所总结的规则的约束：如果一个标识符在文件作用域中有一个可见的声明，那么局部变量就引用该声明。因为除了作用域不同外，局部变量与全局变量没有区别。所以任何时候，局部变量都是通过接口函数 defsymbol 来通知的。

```
⟨extern local 234⟩=                                  235  233
  if (q && q->scope == GLOBAL && q->sclass == STATIC) {
     p->sclass = STATIC;
     p->scope = GLOBAL;
     (*IR->defsymbol)(p);
     p->sclass = EXTERN;
     p->scope = level;
  } else
     (*IR->defsymbol)(p);
```

正如该代码所示，相同名字的静态说明符若存在一个可见的文件作用域声明，则需要进行特殊处理。编译后端的 defsymbol 函数会对 statics 和 externs 进行不同的处理，例如，对其与目标相关的名字进行不同的变换。所以，在 defsymbol 函数调用期间，dcllocal 函数会改变存储类型和作用

域。这段代码并没有对两个标识符的类型兼容性进行检查，这种检查会在下面进行。

如 11.2 节所述，外部局部变量也要安装到 externals 表中，用来对代码块层次中 extern 声明的一致性进行检查。

```
⟨extern local 234⟩+≡                                    ↑234  233
  {
     Symbol r = lookup(id, externals);
     if (r == NULL) {
        r = install(p->name, &externals, GLOBAL, PERM);
        r->src = p->src;
        r->type = p->type;
        r->sclass = p->sclass;
        q = lookup(id, globals);
        if (q && q->sclass != TYPEDEF && q->sclass != ENUM)
           r = q;
     }
     if (r && !eqtype(r->type, p->type, 1))
        warning("declaration of '%s' does not match previous
           declaration at %w\n", r->name, &r->src);
  }
```

如果在 externals 表中已经存在该标识符的符号，则它必须具有兼容的类型。否则，该标识符就安装到 externals 表中。在第 233 页 dcllocal 函数对重复声明的检查代码中，有一种复杂的情况没有涉及。例如，在下面的代码中：

```
int x;
f(int x) { ... { extern float x; ... } }
```

f 函数中对 x 的外部声明与 x 的文件作用域中的声明相冲突，因为它们对相同的 x 规定了不同的类型。在重复声明检查代码中对 lookup 函数的调用会返回一个指向参数 x 符号的指针，并将该指针赋值给 q，在 ⟨extern local⟩ 的开始正是使用该值对文件作用域的标识符进行检查；参数 x 隐藏了文件作用域中的 x，但是后者正需要检测这种冲突。这样，dcllocal 函数在 globals 表中检查标识符，如果找到，就对它进行类型相容性的检查。如果中间没有隐藏文件作用域标识符的声明，则第二次调用 Lookup 函数设置 q 为已存在的值，这是最常见的一种情况。前面的例子一般很少见，但是总可能发生，尤其在大型程序中。

dcllocal 函数最后分析选择性的初始化。与在 initglobal 函数中不同，在某些情况下初始值可能是任意的表达式。如果局部变量具有标量类型，它的初始值可以是一个表达式或者用大括号括起来的表达式。如果该局部变量是一个结构或者联合，它的初始值可以是一个单个的表达式或者是用大括号括起来的常量表达式列表。如果该局部变量是一个数组，则它的初始值只能是一个用大括号括起来的常量表达式列表。数组必须有明确的大小或者能够根据初始值确定其大小。dcllocal 函数通过生成对该局部变量的赋值来处理所有这些情况：

```
⟨dcllocal 233⟩+≡                                        ↑233  232
  if (t == '=') {
     Tree e;
     if (sclass == EXTERN)
        error("illegal initialization of 'extern %s'\n", id);
```

```
        t = gettok();
        definept(NULL);
        if (isscalar(p->type)
        ||  isstruct(p->type) && t != '{') {
            if (t == '{') {
                t = gettok();
                e = expr1(0);
                expect('}');
            } else
                e = expr1(0);
        } else {
            (generate an initialized static t1 236)
            e = idtree(t1);
        }
        walk(root(asgn(p, e)), 0, 0);
        p->ref = 1;
    }
    if (!isfunc(p->type) && p->defined && p->type->size <= 0)
        error("undefined size for '%t %s'\n", p->type, id);
```

对于具有标量类型、结构类型或者联合类型的局部变量，如果它的初始值是一个简单表达式，则初始化就是用初始表达式对局部变量进行赋值。对于具有聚合类型的局部变量，如果有用大括号括起来的初始表达式，lcc会生成一个无名的静态变量，并根据说明的初始化值进行初始化。一个简单的结构赋值可以初始化局部变量，甚至数组。

```
(generate an initialized static t1 236)=                                    236
    Symbol t1;
    Type ty = p->type, ty1 = ty;
    while (isarray(ty1))
        ty1 = ty1->type;
    if (!isconst(ty) && (!isarray(ty) || !isconst(ty1)))
        ty = qual(CONST, ty);
    t1 = genident(STATIC, ty, GLOBAL);
    initglobal(t1, 1);
    if (isarray(p->type) && p->type->size == 0
    && t1->type->size > 0)
        p->type = array(p->type->type,
            t1->type->size/t1->type->type->size, 0);
```

这个静态变量将永远不会改变，因此，const限定符会加到它的类型中，导致initglobal函数将它定义在LIT段中。

11.8 结束处理

正如上一节中介绍的，对于每一个具有文件作用域的标识符，在编译结束时要调用checkref函数，例如，具有GLOBAL作用域的标识符。该调用源自于finalize函数，finalize函数还处理外部变量和全局变量。

```
(decl.c functions) +=                                                  232 237
    void finalize() {
        foreach(externals,  GLOBAL,  doextern, NULL);
```

```
    foreach(identifiers, GLOBAL,    doglobal, NULL);
    foreach(identifiers, GLOBAL,    checkref, NULL);
    foreach(constants,   CONSTANTS, doconst,  NULL);
}
```

finalize 4行代码的每一行处理符号表中的一组符号，这组符号在 foreach 调用中列出。第一行处理的是 externals 表中的标识符。dellocal 函数将声明为 extern 的变量安装在 externals 表中。这些声明中有一部分也引用了文件作用域中声明的标识符，并在 identifiers 中存入口。但是，另一部分引用了在其他翻译单元中声明的标识符。出现 extern 声明的翻译单元都需要引入这些标识符。doextern 函数通过调用接口函数 import 来引入它们：

```
⟨decl.c functions⟩+≡                                    ↑236 237↓
  static void doextern(p, cl) Symbol p; void *cl; {
    Symbol q = lookup(p->name, identifiers);

    if (q)
      q->ref += p->ref;
    else {
      (*IR->defsymbol)(p);
      (*IR->import)(p);
    }
  }
```

当 dellocal 函数遇到一个 extern 声明时，就会调用 import 函数。因为局部变量声明能够出现在文件作用域的定义之前，lcc 不能为这些局部变量的标识符调用 import 函数。

第二次调用 foreach 函数来最终确定临时定义和文件作用域的 extern 声明。没有初始值的对象如果没有存储类型或者具有静态存储类型，则它的文件作用域声明就是一个临时定义。对于一个标识符可以有不止一个这样的声明，只要这些声明类型指定相容类型。例如，输入：

```
int x;
int x;
int x;
```

是有效的，每个声明都是 x 的一个临时定义。一个具有初始值的文件作用域声明是一个外部定义，可能只有一个这样的定义。

在翻译单元结束时，那些只有临时定义的文件作用域标识符必须最终确定。假定该标识符在翻译单元中有一个文件作用域的外部定义，且具有初始值 0。例如，x 最终确定是基于假定：

```
int x = 0;
```

这样，没有初始化的文件作用域对象会根据定义初始化为 0。

doglobal 函数处理 identifiers 表中的每个标识符。

```
⟨decl.c functions⟩+≡                                    ↑237 238↓
  static void doglobal(p, cl) Symbol p; void *cl; {
    if (!p->defined && (p->sclass == EXTERN
    || isfunc(p->type) && p->sclass == AUTO))
      (*IR->import)(p);
    else if (!p->defined && !isfunc(p->type)
    && (p->sclass == AUTO || p->sclass == STATIC)) {
      if (isarray(p->type)
```

```
        && p->type->size == 0 && p->type->type->size > 0)
            p->type = array(p->type->type, 1, 0);
        if (p->type->size > 0) {
            defglobal(p, BSS);
            (*IR->space)(p->type->size);
        } else
            error("undefined size for '%t %s'\n",
                p->type, p->name);
        p->defined = 1;
    }
    ⟨print an ANSI declaration for p 238⟩
}
```

如果一个外部标识符或者非静态的函数没有定义，它会引用一个在其他翻译单元中给出的定义，在其他翻译单元中定义的标识符将被引入。未定义的对象（即那些只有临时定义的对象）会在 BSS 段中定义。编译后端必须确保该段在执行之前清除。数组要进行特殊的处理：如果未说明数组的大小，则会按照只有一个元素来定义。

lcc 的 -P 选项使得 doglobal 函数在标准错误输出设备上输出 ANSI 风格的声明。

```
⟨print an ANSI declaration for p 238⟩=                                  238
    if (Pflag
    && !isfunc(p->type)
    && !p->generated && p->sclass != EXTERN)
        printdecl(p, p->type);
```

对于函数来说，即使函数是变旧风格定义的，输出也包含函数原型。编辑该输出有助于将旧的程序转换为 ANSI C。参见练习 4.5。

在编译过程中，大部分常量都以 dag 形式结束，并嵌入在机器指令代码中。正如 5.1 节显示的配置标准所规定的，一些常量不能出现在指令中，字符串文字（string literals）永远不能出现在指令当中。对每个这样的常量，都会生成一个匿名的静态变量，doconst 函数会把该变量初始化为常量的值。

```
⟨decl.c functions⟩+=                                                    ↑237
    void doconst(p, cl) Symbol p; void *cl; {
        if (p->u.c.loc) {
            defglobal(p->u.c.loc, LIT);
            if (isarray(p->type))
                (*IR->defstring)(p->type->size, p->u.c.v.p);
            else
                (*IR->defconst)(ttob(p->type), p->u.c.v);
            p->u.c.loc->defined = 1;
            p->u.c.loc = NULL;
        }
    }
```

constants 表中符号的 u.c.loc 域指向了匿名静态变量的符号。

11.9 主程序

main.c 文件中的 main 函数通过调用 program 函数和 finalize 函数来开始和结束编译过程，它还调用了接口函数 progbeg 和 progend 使得编译后端进行初始化和最终处理。

```
⟨main.c functions⟩=
  int main(argc, argv) int argc; char *argv[]; {
     ⟨main 239⟩
     return errcnt > 0;
  }
```

errcnt 是编译中检测到的错误数目，因此，如果有错误 lcc 就返回 1。在大多数系统中，这种结束代码终止了编译系统，而不继续执行后续处理程序，如汇编器和链接器。

main 函数调用初始化函数之前，它必须将 IR 指向合适的接口记录，参见 5.11 节。编译后端初始化 bindings 数组以保存名字和指向相应接口记录的指针对。main 函数使用编译选项中最右端的 -target=name 选项来选择所需要的接口记录：

```
⟨main.c data⟩=                                                    240
  Interface *IR = NULL;

⟨main 239⟩=                                                  239  239
  {
     int i, j;
     for (i = argc - 1; i > 0; i--)
        if (strncmp(argv[i], "-target=", 8) == 0)
           break;
     if (i > 0) {
        for (j = 0; bindings[j].name; j++)
           if (strcmp(&argv[i][8], bindings[j].name) == 0)
              break;
        if (bindings[j].ir)
           IR = bindings[j].ir;
        else {
           fprint(2, "%s: unknown target '%s'\n", argv[0],
              &argv[i][8]);
           exit(1);
        }
     }
  }
  if (IR) {
     int i;
     fprint(2, "%s: must specify one of\n", argv[0]);
     for (i = 0; bindings[i].name; i++)
        fprint(2, "\t-target=%s\n", bindings[i].name);
     exit(1);
  }
```

如果没有给出 -target 选项，lcc 则列出所有可能的目标并退出。IR 一旦指向一个接口记录，编译前端就会被绑定到一个目标机器上，而且在该翻译单元的处理过程中这种绑定不会改变。

接下来，main 函数初始化编译前端的类型系统并且分析其他编译选项：

```
⟨main 239⟩+=                                                 239 240  239
  typeInit();
  argc = doargs(argc, argv);
```

除了处理编译前端能够理解的参数外，doargs 函数还要把 infile 和 outfile 分别设置为第一个和第二个非选择性参数。这两个值就是输入的源文件名和输出的汇编语言文件名。如果给出了这两个文件中的一个（或者两个），main 函数就打开文件并设置相应的文件描述符。

```
⟨main.c data⟩+≡                                                    239
  static char *infile, *outfile;

⟨main 239⟩+≡                                                  239 240  239
  if (infile && strcmp(infile, "-") != 0)
     if ((infd = open(infile, 0)) < 0) {
        fprint(2, "%s: can't read '%s'\n",
           argv[0], infile);
        exit(1);
     }
  if (outfile && strcmp(outfile, "-") != 0)
     if ((outfd = creat(outfile, 0666)) < 0) {
        fprint(2, "%s: can't write '%s'\n",
           argv[0], outfile);
        exit(1);
     }
  inputInit();
  outputInit();
```

初始化文件描述符后，上面的 Init 函数就会初始化输入和输出模块，接着初始化编译后端：

```
⟨main 239⟩+≡                                                  240  240  239
  t = gettok();
  (*IR->progbeg)(argc, argv);
```

doargs 函数修改 argv，只保存那些前端不能理解的选项，它们被认为是编译后端的选项。doargs 函数返回这些选项的数目，并赋给上面的 argc 参数。program 函数编译源代码：

```
⟨main 239⟩+≡                                                  240  240  239
  program();
```

最后，main 函数调用 finalize 函数和接口过程 progend，并输出结果，结束编译：

```
⟨main 239⟩+≡                                                       240  239
  finalize();
  (*IR->progend)();
  outflush();
```

深入阅读

Ritchie（1993）详细介绍了 C 语言的发展历史，说明了其声明语法的起源和特点，这也是 C 语言的突出特点之一，同时也是经常受到批评的地方。Sethi（1981）总结了各种设计思想，提出了一种声明符的替代语法：用 Pascal 中的后缀 ^ 代替 C 中的前缀 * 来表示指针。如果他的替代语法被采纳，则 dclr 函数和 dclrl 函数可以大大简化。

与大多数高级语言一样，C 语言要求标识符在被使用之前先进行声明（函数除外）。这条规则强迫语言设计者允许多次声明，并且引入一些规则，例如 C 语言中尝试性定义的规则。在 dclX 函数、doglobal 函数和 doextern 函数中的大部分代码都用来处理这些设计规则。Modula-3（Nelson，

1991）是极少数允许声明和使用以任意顺序出现，从而避免了顺序规则的语言之一，因此语言的理解变得简单许多。这种设计思想在编译领域有其影响，但是这种影响并没有超过 C 语言中多次声明这一规则的影响。

练习

11.1 dclrl 函数接受错误声明 int *const const*p，但 lcc 仍会给出预期的检测信息：

```
illegal type 'const const pointer to int'
```

该错误是在哪里以及如何被检测到的？

11.2 dclrl 的实现看起来很特殊。第 207 页的语法规范建议 dclrl 函数以一个循环开始，该循环通过分析剩余声明符来计算指针。使用该方法重写一个 dclrl 函数。你会发现需要把逆序类型的指针部分前加到通过分析剩余声明符而构造的逆序类型中。通过程序变换，把你的实现改成与 dclrl 类似的情况。

11.3 类型名用于类型转换，也可作为 sizeof（参见 <type cast> 和 <sizeof> 的操作数。类型定义的语法格式如下：

```
type-name:
    { type-specifier | type-qualifier } [ abstract-declarator ]
abstract-declarator:
    * { type-qualifier }
    pointer '(' abstract-declarator ')'
    | suffix-abstract-declarator }
    pointer [ suffix-abstract-declarator ]
suffix-abstract-declarator:
    '[' [ constant-expression ] ']'
    '(' parameter-list ')'
```

抽象声明符（abstract- declarator）就是一个没有内嵌标识符的声明。实现：

```
⟨main.c exported functions⟩=                                    241
  extern Type typename ARGS((void));
```

用它来分析 type-name。当 dclr 函数的参数 abstract 等于 1 时，它将分析抽象声明符，所以 typename 函数可以让 dclr 做大部分工作，而函数自身不会超过 10 行代码。

11.4 实现：

```
⟨main.c exported functions⟩+=                                   241 242
  extern void checklab ARGS((Symbol p, void *cl));
```

对 stmtlabs 中的每个符号都调用 checklab 函数。如果 p 是一个未定义的标号，checklab 将处理这种错误。

11.5 dcllocal 函数调用 initglobal 函数来分析静态局部变量的初始化，但是它也可以分析一个可选的初始化，即使这样，lcc 还是拒绝类似于下面的输入：

```
f() { static int x = 2  = 3; }
g() { static int y = 2; = 3; }
```

请解释一下原因。

11.6 在fields函数中，具有最大对齐数的域决定了整个结构的对齐数。只因为基本类型的大小和对齐数都必须是2的幂，所以该结论成立。修改fields函数，使得对于基本类型的大小和对齐数的任何正值，该结论都是正确的。

11.7 位域声明（例如unsigned:0）会导致接下来的位域被存放在下一个可编址的存储单元，即使当前单元还有空间。例如，如果图11-1的声明改为：

```
struct {
    char a[5];
    short s1, s2;
    unsigned code:3, :0, used:1;
    int amt:7, last;
    short id;
}
```

code域存放在无符号整数中，偏移量为12，used域存放在amt域的右边，并与code存放在同一无符号整数中，偏移量为16。解释fields函数是如何处理这种情况的。

11.8 fields函数的代码很难读懂。写一个新的（假定是更好的）版本并比较两个版本，看看是不是你的版本更容易理解一些？对于它的正确性你有没有足够的信心？

11.9 枚举说明符的语法格式如下：

enum-specifier:
  enum [*identifier*] '{' *enumerator* [, *enumerator*] '}'
  enum *identifier*

enumerator:
  identifier
  identifier = *constant-expression*

为枚举说明符enum-specifier实现分析函数：

⟨*main.c exported functions*⟩ +≡ 241
```
extern Type enumdcl ARGS((void));
```

enumdcl函数类似于structdcl函数，但是更简单一些，而且对于单独出现在声明中的枚举说明符enum-specifier没有特殊的规则。枚举定义不允许相互递归，因此一个具有enumerator的枚举说明符enum-specifier必定不会引用已经存在的枚举类型。enumdcl函数可以使用newstruct函数来定义一个新的枚举类型，并且它会在的identifiers表中建立具有ENUM存储类型的枚举常量。枚举常量的整数值存储在该符号的u.value域中。

中间代码的生成

lcc 编译器前端的剩余代码将分析树转换为 dag（无环有向图），并将其添加到代码表中。编译后端通过 function 接口过程调用 gencode 函数和 emitcode 函数来遍历代码表。这部分代码在 dag.c 文件中，dag.c 文件输出 gencode 和 emitcode 函数（参见 5.10 节）。

```
⟨dag.c exported functions⟩+≡                                    ▲70
  extern void walk      ARGS((Tree e, int tlab, int flab));
  extern Node listnodes ARGS((Tree e, int tlab, int flab));
  extern Node newnode   ARGS((int op, Node left, Node right,
                              Symbol p));
```

walk 和 listnodes 函数操作处理 dag 森林（5.5 节中定义）。森林序列表示函数的代码。该森林序列由代码表中的 Gen、Jump 和 Label 入口构成。10.3 节概述了 listnodes 函数如何逐步构建森林序列，将分析树 e 转换为 dag 并将 dag 添加到森林中。图 5-2 和图 5-3 给出了森林的例子。

walk 函数通过调用 listnodes 函数将树 e 转换为 dag，并将森林添加到代码表的 Gen 入口中，然后为新的森林重新初始化编译前端的一些变量。将树转换为 dag 的复杂工作主要由 listnodes 函数承担，因此 walk 函数很简单：

```
⟨dag.c functions⟩≡                                              245▼
  void walk(tp, tlab, flab) Tree tp; int tlab, flab; {
     listnodes(tp, tlab, flab);
     if (forest) {
        code(Gen)->u.forest = forest->link;
        forest->link = NULL;
        forest = NULL;
     }
     reset();
     deallocate(STMT);
  }

⟨dag.c data⟩≡                                                   245▼
  static Node forest;
```

forest 指向当前森林的最后一个节点，森林在构建的时候是一个循环链表，该链表通过节点的 link 域连接起来，因此 forest->link 就是森林的第一个节点。当 walk 函数将森林添加到代码表中时，就将该链表转换为一个非循环链表。

传递给 listnodes 函数和 walk 函数的 tlab 和 flab 的值表示标号的数值，当 e 为一个条件表达式（比如“比较”）时才用到它们的值。如果 tlab 是非零值，那么当 e 也非零时，listnodes 函数就生成跳转到 tlab 的代码。如果 flab 非零而 e 等于 0；则 listnodes 函数生成跳转到 flab 的代码。10.4 节给出了如何用这些标号来生成 if 语句的代码。tlab 和 flab 的值只有一个可以是非零值。

newnode 函数为节点分配内存空间并用它的参数值来初始化节点的域。例如，newnode 函数可以被编译前端的 definelab 和 jump 函数调用；由于后端必须构建 dag 溢出寄存器，因此 newnode 也可以被编译后端调用。

listnodes 函数还要负责删除公共子表达式，即用来计算冗余值的表达式。例如，图 8-1 给出了表达式（a+b）+b*（a+b）的分析树。a+b 的值被计算了两次，b 的右值也计算了两次；第一次是计算 a+b，第二次是计算乘法表达式。尽管 b 的右值是一个简单的计算，但还是有一次冗余。图 12-1 给出了删除这些公共子表达式后的 dag。左值也可能是公共表达式，例如图 5-3 森林中的 p 的左值。这些表和其他 dag 图中的操作符与表 5-1 中的一样。在后缀的省略“+”号以区别分析树和 dag。

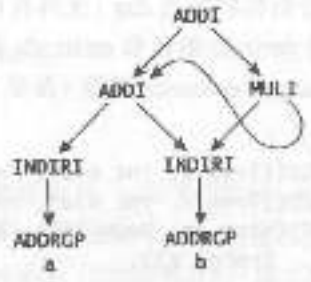


图 12-1 （a+b）+b*（a+b）的 dag 图

编译前端构建的某些分析树，已经在源语言中通过扩充的赋值和后缀操作符反映了 dag 的结构，实际上就是 dag 了。图 8-2 中表达式 a+=b 的分析树和图 8-3 中 i++ 的分析树就是很好的例子。为了产生有效的中间代码，listnodes 函数必须检查这种习惯用法，以便按照标准的规定生成计算这些操作的操作数的中间代码。前缀、后缀和扩充的赋值操作的操作数只能计算一次。

分析树包含的操作符并不是表 5-1 中列出的接口指令表的一部分。listnodes 函数生成表 8-1 中操作符的实现代码，然后消除这些操作符。例如，通过使用具有标号的比较操作，以及插入跳转和定义必要的标号，listnodes 函数实现了 AND。同样，结合移位和屏蔽操作实现了指定位域析取或赋值操作 FIELD。

基本块是只有单入口和单出口的顺序执行的指令序列；如果代码块中的一条指令被执行，则所有指令都会被执行。基本块一般以跳转的目标指令、条件或非条件跳转指令的后继指令开始。编译器通常使用流图来表示函数。流图中的节点就是基本块，而有向箭头则表示了基本块之间可能的控制流。lcc 的代码表不是流图，Gen 入口中的森林可以包含跳转指令和标号，也不是基本块。它们所表示的可以称为扩展基本块：单个入口点，但有多个出口和多个内部执行路径。从下面要介绍的 listnodes 函数的实现可以看到，在某些情况下，这种设计使得公共表达式可以在整个扩展基本块内考虑。这样 listnodes 函数可以花费很小的代价使这些子表达式的生命期超越基本块的范围。

12.1 消除公共子表达式

listnodes 函数根据树 t 构建相应的节点 n。分析树在 8.1 节中定义，节点在 5.5 节中定义。n->op 来自于 t->op。n->syms 的元素来自于 t->u.sym 或者是在 constants 表中建立的 t->u.v，或者由 listnodes 函数根据其他常量构造。n->kids 的元素来自于 t->kids 对应元素的节点。n 还有一个 count 域，它表示引用 n 作为操作数的节点的数目。

节点自底向上构建：n 通过按后序遍历 t 来构造，等价于

```
l = listnodes(t->kids[0], 0, 0);
r = listnodes(t->kids[1], 0, 0);
n = node(t->op, l, r, t->u.sym);
```

node 函数为一个新的节点分配存储空间并用它的参数来初始化域。为了删除公共子表达式，node 函数要检查所要的节点是否已经构建，也就是说，如果已经有一个各域相同的节点，就不必构建一个新的节点了。

node 函数要维护一张已构建节点的表格，在为一个新节点分配内存之前需要查询这张表。当它为一个新节点分配了内存空间后，就将该节点添加到这张表中。图 12-1 展示了由图 8-1 中的树构建 dag 的过程，并说明这张表是如何构建和使用的。表 12-1 给出了 node 函数的调用序列以及每一步构建的节点（如果需要构建新节点）和返回值。中间一列表示了通过 node 函数形成表的过程。节点是用数字表示的。第一次调用是对图 8-1 左下角的 ADDRG+P 树进行处理；此时 node 的表是空的，所以它就为 ADDRG+P 构建了一个节点并返回该节点。接下来的 4 次调用与此类似，都遍历 a+b 树的剩余部分。每一步都构建了相应的节点并将其返回。节点返回后，它们被其他节点用作操作数。当对 MUL+I 左操作数的叶节点 ADDRG+P 调用 node 函数时，它会发现该节点在表中（节点 3）并返回它。类似地，它也会发现节点 4 对应于（INDIRI 3）。node 函数不断在表中寻找节点，其中就包括公共子表达式 a+b 对应的节点。

表 12-1 为 (a+b) +b* (a+b) 调用 node 函数

调用参数				构建	返回
ADDRG+P			a	1=(ADDRGP a)	1
INDIR+I	1			2=(INDIRI 1)	2
ADDRG+P			b	3=(ADDRGP b)	3
INDIR+I	3			4=(INDIRI 3)	4
ADD+I	2	4		5=(ADDI 2 4)	5
ADDRG+P			b		3
INDIR+I	3				4
ADDRG+P			a		1
INDIR+I	1				2
ADDRG+P			b		3
INDIR+I	3				4
ADD+I	2	4			5
MUL+I	3	5		6=(MULI 3 5)	6
ADD+I	5	6		7=(ADDI 5 6)	7

上表第 2 列描述的节点存储在哈希表中：

```
⟨dag.c data⟩+≡                                    243 260
  static struct dag {
     struct node node;
     struct dag *hlink;
  } *buckets[16];
  int nodecount;
```

dag 结构包含一个节点和指向同一哈希桶中的另一个 dag 结构的指针。nodecount 表示 buckets 中的节点总数。哈希表中的节点数很少超过几十个，因此只需要 16 个哈希桶。node 函数在 buckets 中搜索具有相同操作符、操作数和符号的节点，如果找到就返回该节点；否则就构建一个新的节点，并将新节点添加到 buckets 中，然后返回该新节点。

```
⟨dag.c functions⟩+≡                               243 246
  static Node node(op, l, r, sym)
  int op; Node l, r; Symbol sym; {
     int i;
```

```
    struct dag *p;

    i = (opindex(op)^((unsigned)sym>>2))&(NELEMS(buckets)-1);
    for (p = buckets[i]; p; p = p->hlink)
        if (p->node.op       == op && p->node.syms[0] == sym
        &&  p->node.kids[0] == l  && p->node.kids[1] == r)
            return &p->node;
    p = dagnode(op, l, r, sym);
    p->hlink = buckets[i];
    buckets[i] = p;
    ++nodecount;
    return &p->node;
}
```

dagnode 函数分配并初始化新的 dag 和它包含的节点。如果有操作数节点，它还要负责增加该操作数节点的 count 域。

```
⟨dag.c functions⟩+≡                                    245 246
  static struct dag *dagnode(op, l, r, sym)
  int op; Node l, r; Symbol sym; {
      struct dag *p;

      NEW0(p, FUNC);
      p->node.op = op;
      if ((p->node.kids[0] = l) != NULL)
          ++l->count;
      if ((p->node.kids[1] = r) != NULL)
          ++r->count;
      p->node.syms[0] = sym;
      return p;
  }
```

listnodes 函数调用 node 函数查找一个公共子表达式的节点或者建立一个新节点，调用 newnode 函数跳过查表过程，直接构建一个新节点，但新节点不添加到 buckets 中：

```
⟨dag.c functions⟩+≡                                    246 247
  Node newnode(op, l, r, sym) int op; Node l, r; Symbol sym; {
      return &dagnode(op, l, r, sym)->node;
  }
```

只有 newnode 函数可以被编译后端用来构造它们所需要的节点，比如生成溢出寄存器的代码。

只要节点表示的值是有效的，节点就会出现在 buckets 中。赋值或者函数调用会使 buckets 中的部分或所有节点无效。例如，在下面的代码中：

```
c = a + b;
a = a/2;
d = a + b;
```

在第 3 行中计算的 a+b 的值与第 1 行计算的值是不相同的。第 2 行对 a 的赋值失效节点（INDIR1 a），其中 a 表示的是变量 a 的左值的节点。有副作用的操作符，如 ASGN 和 CALL，必须清除其失效的节点。对于这样的操作符，受影响的节点各不相同，而 lcc 仅按两种情况处理：对标识符的赋值要删除标识符的右值节点，而其他有副作用的操作符要清除所有节点。kill 函数处理赋值：

```
⟨dag.c functions⟩+≡                                    246 248
  static void kill(p) Symbol p; {
     int i;
     struct dag **q;

     for (i = 0; i < NELEMS(buckets); i++)
        for (q = &buckets[i]; *q; )
           if (⟨*q represents p's rvalue 247⟩) {
              *q = (*q)->hlink;
              --nodecount;
           } else
              q = &(*q)->hlink;
  }
```

p 的右值显然可以是形式为（INDIR(ADDRxP p)）的 dag，这里 ADDRxP 是任何一个地址操作符；也可以是形式为（INDIRα）的 dag，此处 α 是任意地址计算，甚至可能是计算 p 的地址。这两种情况都由下面的代码进行检查：

```
⟨*q represents p's rvalue 247⟩=                                247
  generic((*q)->node.op) == INDIR &&
  (!isaddrop((*q)->node.kids[0]->op)
   || (*q)->node.kids[0]->syms[0] == p)
```

只有 INDIR 节点必须被删除，这足以使得后续的查表工作都失败。例如，在经过赋值语句 a = a/2 后，a+b 的节点仍保存在 buckets 中。但是当把 a+b 赋值给 d 时，不会找到该 a+b 节点。对 a 右值的引用会构建新的节点，使得必须也为 a+b 构建一个新的节点。这个例子接下来对 node 函数的调用全都在表 12-2 中显示。因为赋值语句 a=a/2 已经删除了节点 3，a 的右值在表达式 d=a+b 中会重用左值，但是要构建一个新的 INDIRI 节点。赋值通过调用 newnode 函数来构建节点，所以不会出现在 buckets 中。

表 12-2　为 c=a+b; a=a/2; d=a+b 调用 node 函数

调用参数			构建	删除	返回
ADDRG+P		c	1=(ADDRGP c)		1
ADDRG+P		a	2=(ADDRGP a)		2
INDIR+I	2		3=(INDIRI 2)		3
ADDRG+P		b	4=(ADDRGP b)		4
INDIR+I	4		5=(INDIRI 4)		5
ADD+I	3	5	6=(ADDI 3 5)		6
ASGN+I	1	6			7
ADDRG+P		a			2
INDIR+I	2				3
CNST+I		2	8=(CNSTI 2)		8
DIV+I	3	8	9=(DIVI 3 8)		9
ASGN+I	2	9		3	10
ADDRG+P		d	11=(ADDRGP d)		11
ADDRG+P		a			2
INDIR+I	2		12=(INDIRI 2)		12
ADDRG+P		b			4
INDIR+I	4				5
ADD+I	12	5	13=(ADDI 12 5)		13
ASGN+I	11	13			14

reset 函数通过清除 buckets 和 nodecount 来删除 buckets 中的所有节点：

```
⟨dag.c functions⟩+≡                                    247 248
  static void reset() {
     if (nodecount > 0)
        memset(buckets, 0, sizeof buckets);
     nodecount = 0;
  }
```

12.2 构建节点

listnodes 函数为作为其参数的树构建节点。实现时，根据树的操作数来递归调用自身，根据操作符调用 node 函数或者 newnode 函数，有必要的话还要调用 kill 或者 reset 函数。

```
⟨dag.c functions⟩+≡                                    248 251
  Node listnodes(tp, tlab, flab) Tree tp; int tlab, flab; {
     Node p = NULL, l, r;

     if (tp == NULL)
        return NULL;
     if (tp->node)
        return tp->node;
     switch (generic(tp->op)) {
     ⟨listnodes cases 248⟩
     }
     tp->node = p;
     return p;
  }
```

tp->node 指向树 tp 的节点。这个域标识了 listnodes 函数访问过的树，并确保 listnodes 函数为就是 dag 的树返回正确的节点。例如图 8-2 和图 8-3 中所示的树，在这些用法中多次引用的子树会不止一次被访问；第一次访问时构建节点，而接下来的访问只是返回该节点。

在 listnodes 函数中的 switch 语句将操作符分组，同一组中的操作符具有相同的遍历和构建节点代码：

```
⟨listnodes cases 248⟩=                                 248
  case AND:   { ⟨AND 252⟩ } break;
  case OR:    { ⟨OR⟩ } break;
  case NOT:   { ⟨NOT 251⟩ }
  case COND:  { ⟨COND 254⟩ } break;
  case CNST:  { ⟨CNST 255⟩ } break;
  case RIGHT: { ⟨RIGHT 261⟩ } break;
  case JUMP:  { ⟨JUMP 250⟩ } break;
  case CALL:  { ⟨CALL 260⟩ } break;
  case ARG:   { ⟨ARG 261⟩ } break;
  case EQ:  case NE: case GT: case GE: case LE:
  case LT:    { ⟨EQ..LT 251⟩ } break;
  case ASGN:  { ⟨ASGN 256⟩ } break;
  case BOR: case BAND: case BXOR:
  case ADD: case SUB:  case RSH:
  case LSH:   { ⟨ADD..RSH 249⟩ } break;
```

```
case DIV: case MUL:
case MOD:   { (DIV..MOD) } break;
case RET:   { (RET) } break;
case CVC: case CVD: case CVF: case CVI:
case CVP: case CVS: case CVU: case BCOM:
case NEG:   { (CVx,NEG,BCOM 249) } break;
case INDIR: { (INDIR 249) } break;
case FIELD: { (FIELD 250) } break;
case ADDRG:
case ADDRF: { (ADDRG,ADDRF 249) } break;
case ADDRL: { (ADDRL 249) } break;
```

最长的操作符组是一元操作符组。遍历代码访问一元操作符唯一的操作数并构建节点：

```
(CVx,NEG,BCOM 249)≡                                          249
  l = listnodes(tp->kids[0], 0, 0);
  p = node(tp->op, l, NULL, NULL);
```

二元操作符的遍历代码也类似：

```
(ADD..RSH 249)≡                                              248
  l = listnodes(tp->kids[0], 0, 0);
  r = listnodes(tp->kids[1], 0, 0);
  p = node(tp->op, l, r, NULL);
```

DIV、MUL 和 MOD 并不包含在这类情况里。如果设置了接口标志 mulops_calls，它们就必须被当作函数调用来处理，参见练习 12.5。

3 个地址操作符为其所引用的符号的左值构建节点：

```
(ADDRG,ADDRF 249)≡                                           249
  p = node(tp->op, NULL, NULL, tp->u.sym);

(ADDRL 249)≡                                                 249
  if (tp->u.sym->temporary)
    addlocal(tp->u.sym);
  p = node(tp->op, NULL, NULL, tp->u.sym);
```

如果一个局部变量是临时的，则不可能出现在代码表中，如果有必要，addlocal 函数为临时变量增加一个 Local 代码表入口。因为有些临时变量从来不会用到，也从来不需要通知后端，所以不能预先增加这些入口地址。直到最后必要的时刻才会生成 Local 代码表入口地址，从而保证效率，抛弃无用的临时变量。

INDIR 树为右值构建节点，但是声明为 volatile 的地址要进行特殊处理：

```
(INDIR 249)≡                                                 249
  Type ty = tp->kids[0]->type;
  l = listnodes(tp->kids[0], 0, 0);
  if (isptr(ty))
    ty = unqual(ty)->type;
  if (isvolatile(ty)
  || (isstruct(ty) && unqual(ty)->u.sym->u.s.vfields))
   p = newnode(tp->op, l, NULL, NULL);
else
   p = node(tp->op, l, NULL, NULL);
```

如果左值的类型是（POINTER T），INDIR 就会按其他一元操作符一样处理。但是如果左值的类型是（POINTER (VOLATILE T)），则在源代码中对右值的每一次读取必须在执行时实际读右值。这种限制意味着右值不能被当作公共子表达式处理，所以该节点应由 newnode 函数构建。这种限制也适用于具有类型（POINTER(STRUCT…)）和（POINTER(UNION…)）的左值，这里结构或联合有一个或多个域声明为 volatile。

位域通过 FIELD 树引用。对位域的赋值是作为一棵 ASGN 树出现的，而该 ASGN 树的左操作数是 FIELD 树；将在 12.4 节中介绍的针对 ASGN 的 case 分支，会检测这种用法。在其他树中，FIELD 的出现表示位域的右值。FIELD 操作符只能在树中出现，所以 listnodes 函数必须用其他操作合成位域的抽取操作，比如移位和屏蔽操作。

从无符号整数或整数右端的 m 位开始抽取具有 s 位的位域要考虑两种情况：如果该位域是无符号的，它可以通过右移 m 位、然后再把它和 s 位的屏蔽位相与获得；如果该位域是有符号的，其最高有效位作为符号位，抽取域时必须进行扩展。这样，一个有符号域可以通过类似于下面的代码获得：

```
((int)((*p)<<(32 - m)))>>(32 - s)
```

这里假设一个字有 32 位，p 指向保存该域的字。表达式先将该字左移，使得域的最高有效位为符号位，然后做算术右移，使移入的位均为符号位。该表达式同样适用于无符号的情形，只不过是用无符号类型转换替换整数类型转换，这也就是 listnodes 函数对于两种情形的处理方法：

```
⟨FIELD 250⟩≡                                                    249
  Tree q = shtree(RSH,
     shtree(LSH, tp->kids[0],
        consttree(fieldleft(tp->u.field), inttype)),
     consttree(8*tp->type->size - fieldsize(tp->u.field),
        inttype));
  p = listnodes(q, 0, 0);
```

这里宏 fieldleft 有运算结果等于 32-m，consttree 的第一个参数是 32-s。内层的 shtree 调用构建的树的类型依赖于 tp->kids[0] 的类型，可能是整数或者无符号整数，这就使得外层的 shtree 调用生成 RSH+I 或者 RSH+U。

12.3 控制流

上一节所介绍的一元或二元操作将节点加入节点表中。这些节点会被其他节点引用，但是，除了 INDIR 节点，它们从来不会作为森林的根节点出现。如果一个节点具有副作用或者它必须在森林的下层 dag 中的节点之前计算，那么该节点就可以作为根节点。表 5-3 中 INDLRP 节点作为根结点的例子就属于第二种情况。赋值、调用、返回、标号、跳转以及条件转移都属于第一种情况。一部分这些操作符改变了控制流，也会影响到节点表。跳转是最简单的情形：

```
⟨JUMP 250⟩≡                                                     248
  l = listnodes(tp->kids[0], 0, 0);
  list(newnode(JUMPV, l, NULL, NULL));
  reset();
```

处理跳转时，节点表中的表达式不会被跳转后面的代码用到，因此必须重置节点表。上述代码调用 newnode 构建 JUMPV 节点，通过 list 函数将其作为根节点添加到森林中：

```
⟨dag.c functions⟩+≡                                    248 252
  static void list(p) Node p; {
     if (p && p->link == NULL) {
        if (forest) {
           p->link = forest->link;
           forest->link = p;
        } else
           p->link = p;
        forest = p;
     }
  }
```

forest 是一个循环链表。如果 forest 不为空，forest 指向链表中的最后一个节点；否则，list 函数就会将其初始化。link 域也可以标识出节点，而 list 函数列出的根节点不会超过一个。

比较操作符说明了 listnodes 函数的 tlab 和 flab 参数的用法。tlab 和 flab 中只有一个可以是非零值。如果比较操作的结果为真，则操作就跳转到 tlab 标识的指令；如果比较结果为假，则跳转到 flab 标识的指令。比较操作的节点将目标作为标号放在 syms[0] 域中。该标号就是比较结果为真时的目标。结果为假时，没有方法规定目标地址。所以，当 flab 非零时，使用原比较操作的逆操作作为比较操作。

```
⟨EQ..LT 251⟩=                                          248
  Node p;
  l = listnodes(tp->kids[0], 0, 0);
  r = listnodes(tp->kids[1], 0, 0);
  if (tlab)
     list(newnode(tp->op, l, r, findlabel(tlab)));
  else if (flab) {
     int op = generic(tp->op);
     switch (generic(op)) {
     case EQ: op = NE + optype(tp->op); break;
     case NE: op = EQ + optype(tp->op); break;
     case GT: op = LE + optype(tp->op); break;
     case LT: op = GE + optype(tp->op); break;
     case GE: op = LT + optype(tp->op); break;
     case LE: op = GT + optype(tp->op); break;
     }
     list(newnode(op, l, r, findlabel(flab)));
  }
  if (forest && forest->syms[0])
     forest->syms[0]->ref++;
```

listnodes 函数还要处理只出现在树中的控制流操作符：AND、OR、NOT 以及 COND。NOT 操作通过在递归调用 listnodes 函数时将真、假标号对调来实现：

```
⟨NOT 251⟩=                                             248
  return listnodes(tp->kids[0], flab, tlab);
```

AND 和 OR 操作符使用短路计算（short-circuit evaluation）的方法来实现：一旦结果确定就马上停止计算。例如，语句：

```
if (i >= 0 && i < 10 && a[i] > max) max = a[i];
```

如果 i 小于 0 或者 i 大于或等于 10，a[i] 就不用再计算了。AND 和 OR 的操作数总是条件表达式或常量（andtree 为每一个操作数调用 cond 函数），所以处理这些操作符的 case 分支只要定义适当的真和假的标号，并在为这些操作符调用 listnodes 函数时将标号作为参数来传递。

假定 tlab 等于 0 而 flab 等于 L；则为 e_1&&e_2 生成的短路代码具有如下形式：

```
if e1 == 0 goto L
if e2 == 0 goto L
```

换句话说，如果 e_1 等于 0，就转到 L 处继续执行而 e_2 就不会被计算。否则，计算 e_2，如果 e_2 非零且 e_2 等于 0，也转到 L 处继续执行。仅当 e_1 和 e_2 都是非零值时控制才失败。当 tlab 等于 L 而 flab 等于 0 时，如果 e_1 或者 e_2 有一个为 0 控制就失败；只有当 e_1 和 e_2 都不为 0 时，才转到 L 处继续执行。生成的代码具有如下形式：

```
        if e1 == 0 goto L'
        if e2 != 0 goto L
L':
```

对于这种情况，如果 e_1 等于 0，则不必计算 e_2，控制就失败。处理 AND 的 listnodes 代码如下：

```
⟨AND 252⟩=                                                    248
  if (depth++ == 0) reset();
  if (flab) {
     listnodes(tp->kids[0], 0, flab);
     listnodes(tp->kids[1], 0, flab);
  } else {
     listnodes(tp->kids[0], 0, flab = genlabel(1));
     listnodes(tp->kids[1], tlab, 0);
     labelnode(flab);
  }
  depth--;
```

处理 OR 的代码类似，参见练习 12.2。练习 12.15 解释了静态类型变量 depth 及调用 reset 函数的作用。

labelnode 函数将一个 LABELV 节点添加到森林中：

```
⟨dag.c functions⟩+=                                           ↑251 255↓
  static void labelnode(lab) int lab; {
     if (forest && forest->op == LABELV)
        equatelab(findlabel(lab), forest->syms[0]);
     else
        list(newnode(LABELV, NULL, NULL, findlabel(lab)));
     reset();
  }
```

如果森林的最后一个节点是标号，就不用添加另外一个标号节点；新的标号 lab 是作为已存在标号的一个同义词（参见 10.9 节）。由于可能有多个路径转到标号的后续代码，节点表中的公共子表达式必须在标号处被清除，因此 labelnode 函数会调用 reset 函数。

正如 8.6 节中介绍的，OR 和 AND 被处理为右结合的操作符，因此，如图 12-2 所示，类似于 e_1&&e_2&&…&&e_n 的表达式会构建一个右重（right-heavy）树。

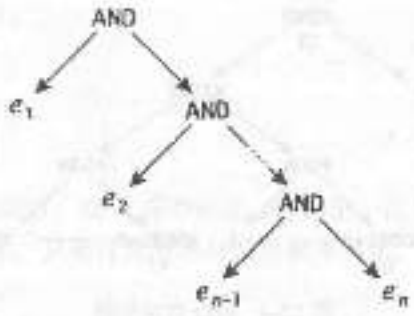


图 12-2 e_1 && e_2 &&…&& e_n 的树

这样的处理保证了在上述代码中递归调用 listnodes 函数时能按照正确的顺序来访问表达式 e，以产生短路计算代码。同时，这还有助于清除出现在 e_i 中的公共子表达式。例如，在下面的语句中：

```
if (a[i] && a[i]+b[i] > 0 && a[i]+b[i] < 10) ...
```

其中 a 和 b 是整型数组，地址计算 4*i，a[i] 和 b[i] 的右值以及 a[i]+b[i] 之和都被计算一次，并在必要时重用。AND 树被传递给了 listnodes 函数，其 flab 参数等于 2，在递归下降分析树时，使用 2 作为 flab 参数调用 listnodess 函数。中间没有调用 reset 函数，因此第二个和第三个子表达式可以重用第一个和第二个子表达式的值。这个语句的森林如图 12-3 所示。比较操作符下面的 2 以及 LABELV 下面的 2 表示其 sym [0] 域中指向标号 2 的符号表指针。

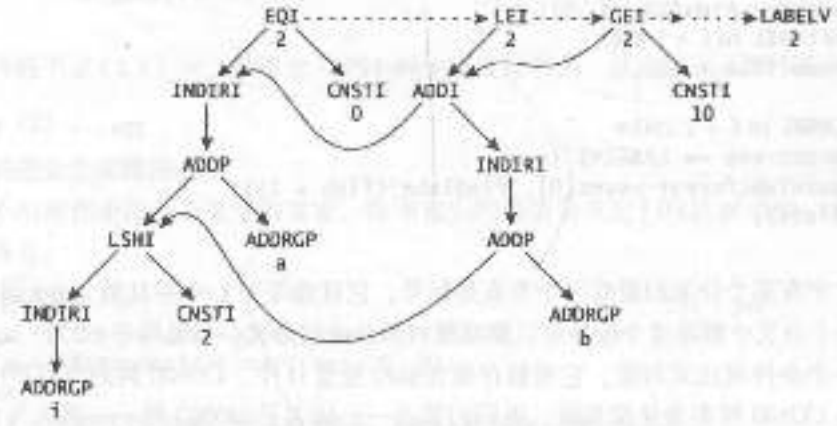


图 12-3 a[i] && a[i]+b[i]>0 && a[i]+b[i]<10 的森林

表达式 e?l:r 生成的 COND 树如图 12-4 所示，其中 RIGHT 树只起到传递两个赋值语句树的作用。生成的代码类似于 if 语句代码：

```
        if e == 0 goto L
        t1 = l
        goto L+1
L:      t1 = r
L+1:
```

t1 的右值是 COND 表达式的结果。如果该条件表达式的值没有被使用或者 l 和 r 都是空表达式，那么对 t1 的赋值就会被忽略。COND 的代码开始先为 t1 增加一个 LOCAL 代码表的入口地址，然后生成 L 和 L+1；接着遍历以 L 为假标号的条件表达式 e。

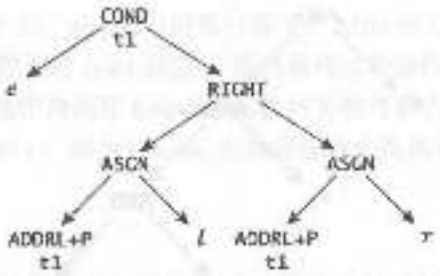


图 12-4 e?l:r 对应的树

```
⟨COND 254⟩≡                                                    254  248
  Tree q = tp->kids[1];
  if (tp->u.sym)
    addlocal(tp->u.sym);
  flab = genlabel(2);
  listnodes(tp->kids[0], 0, flab);
  reset();
```

接下来的代码为第一个赋值、跳转、L、第二个赋值和 L+1 生成节点：

```
⟨COND 254⟩+≡                                                   254 254  248
  listnodes(q->kids[0], 0, 0);
  ⟨equate LABEL to L + 1 254⟩
  list(jump(flab + 1));
  labelnode(flab);
  listnodes(q->kids[1], 0, 0);
  ⟨equate LABEL to L + 1 254⟩
  labelnode(flab + 1);

⟨equate LABEL to L + 1 254⟩≡                                   254
  if (forest->op == LABELV) {
    equatelab(forest->syms[0], findlabel(flab + 1));
    unlist();
  }
```

如果两个分支中有某个分支的最后一个节点是标号，它可能等于 L-1 并且被 unlist 函数从森林中删除。在第一个分支中删除这个标号可以删减跳转到分支的分支，参见练习 12.7。

如果有一个条件表达式的值，它就被存放在临时变量 t1 中。COND 树的节点产生为 t1 赋值的代码，但是 COND 树本身并没有值。返回的节点——代表了 COND 树——就是为 t1 的右值生成的节点：

```
⟨COND 254⟩+≡                                                   254  248
  if (tp->u.sym)
    p = listnodes(idtree(tp->u.sym), 0, 0);
```

在子表达式中，条件操作数可以在其他操作数之前计算，所以遍历 e 的树 tp->kids[0] 后，调用 reset 函数在遍历 l 和 r 之前清除公共子表达式。例如，赋值语句：

```
n = a[i] + (i>0?a[i]:0);
```

尽管先遍历树的左操作数，但条件表达式仍在加法的左类作数之前计算。由于没有调用 reset 函数，所以当遍历 (i>0?a[i]:0) 的树时，公共子表达式 a[i] 的节点已经在节点表中，生成的代码如下：

```
        if i <= 0 goto L1
        t2 = a[i]
        t1 = t2
        goto L2
L1:     t1 = 0
L2:     n = t2 + t1
```

其中 t1 保存的是条件表达式的值，而 t2 保存的是 a[i] 的值。t2 只在条件表达式的 then 分支中被计算，但是都要使用它来求和。只要计算顺序与遍历顺序不同，或者生成的代码可能有多条执行路径，就必须调用 reset 函数。

大多数常量都是作为一元和二元操作符的操作数出现的。但是常量折叠可以使一个微型常量作为比较运算的第一操作数，甚至用于 COND。例如，语句：

```
if (2.5) ...
```

导致 conditional 函数构建表达式 2.5 != 0.0，simplify 函数为这样的表达式产生一棵代表整型常量 1 的树。ifstmt 函数将该树传递给参数 tlab 非零的 listnodes 函数。对于一个整型 CNST 树，如果 tlab 和常量都是非零值或者 flab 非零而常量为零，则 listnodes 函数生成跳转：

```
⟨CNST 255⟩=                                                    255  248
  Type ty = unqual(tp->type);
  if (tlab || flab) {
     if (tlab && tp->u.v.i != 0)
        list(jump(tlab));
     else if (flab && tp->u.v.i == 0)
        list(jump(flab));
  }
```

对于上面的例子 if（2.5）…，没有为 CNST 树生成任何东西，这也体现了程序员的意图。对于像

```
while (1) ...
```

一样的代码就会生成跳转。

对于不出现在条件上下文中的常量，除非它们的类型要求它们应该被消去，否则它们就会生成 CNST 节点：

```
⟨CNST 255⟩+=                                                   255  248
  else if (ty->u.sym->addressed)
     p = listnodes(cvtconst(tp), 0, 0);
  else
     p = node(tp->op, NULL, NULL, constant(ty, tp->u.v));
```

如果某类型的常量不能出现在指令中，typeInit 函数就把它的基本类型的符号表入口 addressed 标记设置为 1。这样，设置了 addressed 标记的类型的常量存放在一个变量中，对该值的引用就由对该变量右值的引用代替。在图 1-2 中的常量 0.5 就是一个例子：它出现在树中，但最终在图 1-3 中是存放在一个变量中的。如果有必要，cvtconst 函数会生成一个匿名的静态变量，并返回该变量的右值树：

```
⟨dag.c functions⟩+=                                            252  263
  Tree cvtconst(p) Tree p; {
     Symbol q = constant(p->type, p->u.v);
     Tree e;
```

```
    if (q->u.c.loc == NULL)
        q->u.c.loc = genident(STATIC, p->type, GLOBAL);
    if (isarray(p->type)) {
        e = tree(ADDRG+P, atop(p->type), NULL, NULL);
        e->u.sym = q->u.c.loc;
    } else
        e = idtree(q->u.c.loc);
    return e;
}
```

编译结束时，finalize 函数调用 doconst 函数对这些变量进行初始化。

12.4 赋值语句

赋值语句的节点都会被列出，没有返回值。但是赋值语句的分析树反映了C语言中赋值的语义，即返回它的左操作数的值。listnodes 函数处理赋值（ASGN）的 case 分支遍历操作数，构建并列出赋值节点。首先处理操作数：

```
(ASGN 256)=                                                    256  248
  if (tp->kids[0]->op == FIELD) {
      (l, r ← for a bit-field assignment 257)
  } else {
      l = listnodes(tp->kids[0], 0, 0);
      r = listnodes(tp->kids[1], 0, 0);
  }
  list(newnode(tp->op, l, r, NULL));
  forest->syms[0] = intconst(tp->kids[1]->type->size);
  forest->syms[1] = intconst(tp->kids[1]->type->align);
```

ASGN 的 syms 域指向常量的符号表入口，这些常量分别给出了值的大小和对齐方式。位域的赋值将在下面介绍。

赋值语句使得节点表中那些依赖于左操作数的先前值的节点失效。lcc 仅处理两种情形：

```
(ASGN 256)+=                                                   256 256  248
  if (isaddrop(tp->kids[0]->op)
  && !tp->kids[0]->u.sym->computed)
      kill(tp->kids[0]->u.sym);
  else
      reset();
```

如果左操作数是一个源语言变量或者临时变量的地址，则赋值语句只删除其右值的节点。如果左操作数是一个已计算变量的地址或者一个已计算的地址，赋值语句就清空节点表。已计算的变量表示变量的地址加上一个常量，例如域的引用，由 addrtree 函数生成。对已计算变量的赋值类似于对数组元素的赋值，即对一个元素赋值会清除数组的所有元素。更精密的措施需要更复杂的分析；那些带来最多益处的措施，比如全局公共子表达式的删除，就需要对整个函数进行数据流分析，lcc 的设计并没有提供这些功能。

赋值语句的值就是其左操作数的新值，而左操作数的新值可能是对右操作数的值进行转换后得到的，所以 listnodes 函数对表示 ASGN 树的节点进行了注释：

```
(ASGN 256)+=                                                   256  248
  p = listnodes(tp->kids[1], 0, 0);
```

tp->kids[1]已经被访问过，它表示前面赋值给r的节点。因此p通常等于r，对位域的赋值是例外。在位域赋值中，对r的计算有所不同，也根本不会访问tp->kids[1]，下面将详细介绍。

作为ASGN树的左操作数的FIELD树标识了对位域的赋值。对位域的赋值被编译成适当的移位和位逻辑操作序列。例如，考虑多重赋值w= x.amt=y，其中x在图11-1中定义，w和y是全局整型变量。第一条赋值x.amt=y被编译为如下等式：

```
*β = ((*β)&0xFFFFFF80) | (y&0x7F);
```

β表示地址x+16。读取x+16单元保存的字，清除对应于amt域的位，这些清除的位和y的最右7位进行或运算，结果存回x+16单元。这个表达式不是很准确：赋给w的值，即x.amt=y的值，不是y的值，而是x.amt的新值。该值一般等于y，除非它的最高有效位是1。如果最高有效位等于1，且赋值语句的结果将被使用，该位就必须按符号扩展。即如果y等于255，则w等于-1。

listnodes函数通过构建一棵ASGN树来处理这种多重赋值语句，该ASGN树的右操作数用来计算正确的值。对语句：w= x.amt=y，listnodes函数构建的右操作数是（y<<25）>>25，这也就是应当出现在上面对*β赋值的y位置上的值。图12-5给出了这个多重赋值语句完整的树。

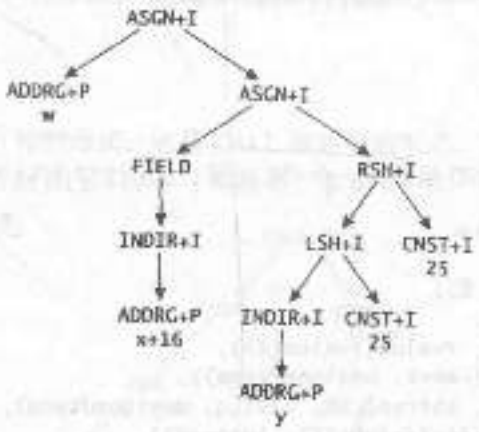


图12-5 语句w=x.amt=y的树

位域赋值的处理代码，将为上面的表达式构建一棵树并调用listnodes函数来遍历该树。

```
⟨1, r ← for a bit-field assignment 257⟩=                    256
  Tree  x = tp->kids[0]->kids[0];
  Field f = tp->kids[0]->u.field;
  reset();
  l = listnodes(lvalue(x), 0, 0);
  if (fieldsize(f) < 8*f->type->size) {
     unsigned int fmask = fieldmask(f);
     unsigned int  mask = fmask<<fieldright(f);
     Tree q = tp->kids[1];
     ⟨q ← the r.h.s. tree 258⟩
     r = listnodes(q, 0, 0);
  } else
     r = listnodes(tp->kids[1], 0, 0);
```

在 ASGN 树下层的 FIELD 子树的 u.sym 域，指向了描述该位域的 field 结构。对于 amt 域，fmask 和 mask 都是 $7F_{16}$；mask 的补码是 $FFFFFF80_{16}$。listnodes 函数对位域赋值语句的处理类似于对数组元素赋值语句的处理，也清空节点表。

当将常量赋值给位域时，有两种情况需要进行特殊处理。如果常量值等于 0，赋值语句就清空域，这可以通过将该字同 mask 的补码进行与操作来实现：

```
⟨q ← the r.h.s. tree 258⟩+≡                                    258  257
  if (q->op == CNST+I && q->u.v.i == 0
  ||  q->op == CNST+U && q->u.v.u == 0)
     q = bittree(BAND, x, consttree(~mask, unsignedtype));
```

如果常量值等于 2^s-1（这里 s 表示位域的大小），赋值语句就会设置该域的所有位，这可以通过将该字同 mask 进行或操作来实现：

```
⟨q ← the r.h.s. tree 258⟩+≡                                    258 258  257
  else if (q->op == CNST+I && (q->u.v.i&fmask) == fmask
  ||       q->op == CNST+U && (q->u.v.u&fmask) == fmask)
     q = bittree(BOR, x, consttree(mask, unsignedtype));
```

这些改进使得给 1 位位域赋常量值同更复杂的逻辑运算一样高效。例如，赋值语句 x.used=1 被编译为下面的等式：

```
*α = *α | 0x8;
```

这里 α 表示地址 x + 12。

一般情况下需要两个 AND 运算和一个 OR 运算，如第 257 页对 *β 的赋值。

```
⟨q ← the r.h.s. tree 258⟩+≡                                    258  257
  else {
     listnodes(q, 0, 0);
     q = bittree(BOR,
        bittree(BAND, rvalue(lvalue(x)),
           consttree(~mask, unsignedtype)),
        bittree(BAND, shtree(LSH, cast(q, unsignedtype),
           consttree(fieldright(f), inttype)),
           consttree(mask, unsignedtype)));
  }
```

图 12-6 给出了多重赋值语句 w=x.amt=y 的森林，这属于一般情况。在所有 3 种情况中，q 是位域赋值语句右方的子树，而 tp->kids[1] 表示赋值语句的值。例如，图 12-6 中的 RSHI 节点表示的是图 12-5 中位域 ASGN+I 树，这样它就用作对 w 赋值的右操作数。

在 <q ← the r.h.s tree> 中的最后一个 else 子句对存放该域的字的右值进行了重新构建，因为该字可能已经被 tp->kids[1] 改变了。例如：

```
struct { int b:4, c:4; } x;
x.c = x.b++;
```

在上面的语句中，x.b++ 改变了保存 x、c 的字。作为被 bittree 的第二个参数返回的树，上面代码块中的 rvalue(lvalue(x)) 使得该字被重新读取。如果用的是 x，则对 x.c 的赋值就会使用 x.b 被改变之前该字的值，x.b 的新值将会丢失。

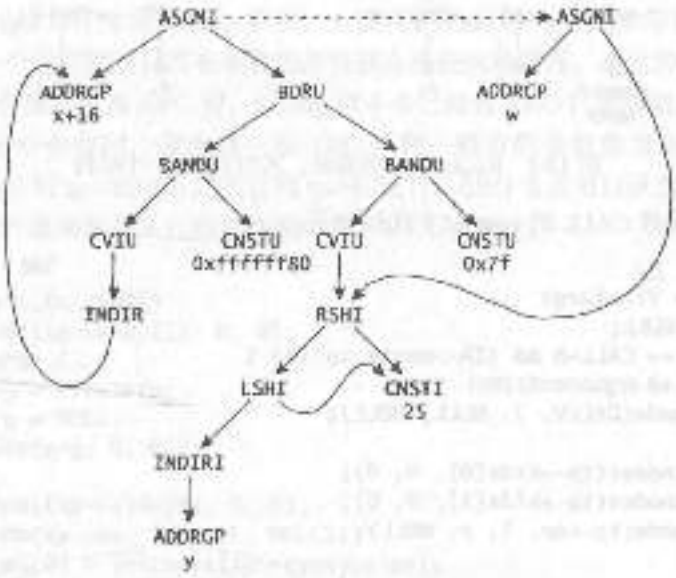


图 12-6 语句 w=x.and=y 的森林图

12.5 函数调用

图 12-7 给出了 CALL+B 树的形式，这是 CALL 最通用的形式。其中，右操作数是保存返回值的临时变量的地址。其他的 CALL 只有一个操作数。参见 9.3 节。

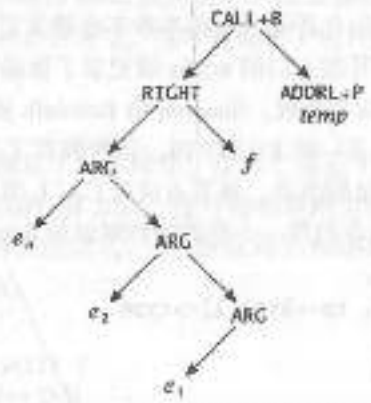


图 12-7 $f(e_1,e_2,\ldots,e_n)$ 的分析树，其中 f 返回一个结构

CALL+B 使得 listnodes 函数对 CALL 的处理更加复杂。如果接口标志 wants_callb 等于 0，则右操作数会作为隐含的第一个操作数传递，使用 CALLV 节点而不是 CALLB 节点。另一个使 CALL 的处理复杂化的原因是 ARG 子树可以出现在 CALL 树左操作数的下面，但是相应的节点又在列表中，且 CALL 节点的左操作数是函数的地址（见 5.5 节）。最后，接口标志 left_to_right 也使 listnodes 复杂化：如果该标志等于 1，则从左到右计算参数；如果该标志为 0，则是从右到左计算参数。当 wants_callb 等于 0 时，这种计算顺序也适用于隐含的第一个参数。图 12-8 给出了当 wants_callb 等于 0 且 left_to_right 等于 1 时，图 12-7 中的树所生成的森林。第一个 ARGP 节点就是隐含的第一个参数。



图 12-8 f(e1,e2,...,en) 的森林，其中 f 返回一个结构

listnodes 函数处理 CALL 的 case 分支代码如下：

```
⟨CALL 260⟩=                                                    248
  Tree save = firstarg;
  firstarg = NULL;
  if (tp->op == CALL+B && !IR->wants_callb) {
     ⟨list CALL+B arguments 260⟩
     p = newnode(CALLV, l, NULL, NULL);
  } else {
     l = listnodes(tp->kids[0], 0, 0);
     r = listnodes(tp->kids[1], 0, 0);
     p = newnode(tp->op, l, r, NULL);
  }
  list(p);
  reset();
  cfunc->u.f.ncalls++;
  firstarg = save;

⟨dag.c data⟩+=                                                 ↑245 268↓
  static Tree firstarg;
```

如果有必要，firstarg 传递隐含的第一个参数的树。代码段首先保存 firstarg 并将其重新初始化为空。在代码段末尾将其恢复，使包含其他调用的参数不会覆盖它。一个调用总会被列出，它会删除节点表中的所有节点。函数符号表入口的 ncalls 域记录了该函数所执行的 CALL 的次数。这个值为接口过程 function 提供了第四个参数，function 由 funcdefn 函数调用。

如图 12-7 所示，tp->kids[0] 是一棵 RIGHT 树，该树保存了参数和函数地址树。遍历这棵树就会列出所有参数并返回函数地址的节点，该节点成为 CALL 节点的左操作数。

对于 CALL+B 树，将代表隐含的第一个参数的树赋值给 firstarg：

```
⟨list CALL+B arguments 260⟩=                                   260
  Tree arg0 = tree(ARG+P, tp->kids[1]->type,
     tp->kids[1], NULL);
  if (IR->left_to_right)
     firstarg = arg0;
  l = listnodes(tp->kids[0], 0, 0);
  if (!IR->left_to_right || firstarg) {
     firstarg = NULL;
     listnodes(arg0, 0, 0);
  }
```

如果 left_to_right 等于 1，则当 listnodes 函数遍历 tp->kids[0] 时，在参数被访问之前需要先使用 firstarg 得到隐含的第一个参数的树，以便隐含参数在其他参数之后被列出。当 left_to_right 等于 0 时，隐含参数总是被最后列出，因此不需要使用 firstarg。

如果 left_to_right 等于 1 且 firstarg 非空，上面代码的最后一个 if 语句也会遍历隐含参数的树。例如，调用一个返回结构且没有参数的函数就会出现这种情况。在这种情况下，对于该函数调用，tp->kids[0] 中没有包含 ARG 树，firstarg 就不会已经被 ARG 代码所遍历。

当从左到右分析参数时，会构建一棵 ARG 子树，最右的参数作为 ARG 子树的根，如图 12-7 所示。通过在访问 tp->kids[0] 之前访问 tp->kids[1]，ARG 节点可以从左到右被一一列出；通过其他顺序访问操作数则按照从右到左的顺序将 ARG 节点列出。

```
(ARG 261)=                                                    248
  if (IR->left_to_right)
     listnodes(tp->kids[1], 0, 0);
  if (firstarg) {
     Tree arg = firstarg;
     firstarg = NULL;
     listnodes(arg, 0, 0);
  }
  l = listnodes(tp->kids[0], 0, 0);
  list(newnode(tp->op, l, NULL, NULL));
  forest->syms[0] = intconst(tp->type->size);
  forest->syms[1] = intconst(tp->type->align);
  if (!IR->left_to_right)
     listnodes(tp->kids[1], 0, 0);
```

与 ASGN 节点类似，ARG 节点的 syms 域也指向常量的符号表入口，这些常量分别给出了参数的大小和对齐方式。

当 left_to_right 标记等于 1 时，第一次执行到测试 firstarg 的时机就是在转换第一个参数的 ARG 树时，即图 12-7 中 e_1 所对应的 ARG 被遍历时。如果 firstarg 非空，它就会在第一个参数的树之前被列出，且 firstarg 被重新设置为空，这样就保证了它只被转换一次。

12.6 强制计算顺序

在标准 C 中只有少数操作规定了计算顺序。标准 C 规定了 AND 和 OR 操作为短路计算，规定 COND 操作遵循通常 if 语句的计算方式。逗号操作符的左操作数必须在右操作数之前计算。lcc 用树（RIGHT e_1, e_2）来表示表达式 e_1, e_2。所以对于 RIGHT 树，listnodes 函数首先计算左操作数，然后计算并返回右操作数：

```
(RIGHT 261)=                                                  248
  if (⟨tp is a tree for e++ 263⟩) {
     ⟨generate nodes for e++ 263⟩
  } else if (tp->kids[1]) {
     listnodes(tp->kids[0], 0, 0);
     p = listnodes(tp->kids[1], tlab, flab);
  } else
     p = listnodes(tp->kids[0], tlab, flab);
```

正如第 8 章、第 9 章及上面的代码所介绍的，RIGHT 树并不仅仅用于逗号操作符，它可能有一个或者两个操作数。RIGHT 树的值就是它的最右操作数的值。

例如，RIGHT 树可以用来展开嵌套调用，即将函数调用作为参数的函数调用。call 函数将所有这样的参数提升为 RIGHT 树的左操作数，这样它们就能在 ARG 树（参数出现在 ARG 中）之前被列出。图 12-9 给出了 $f(e_1,g(e_2),e_3)$ 的树。

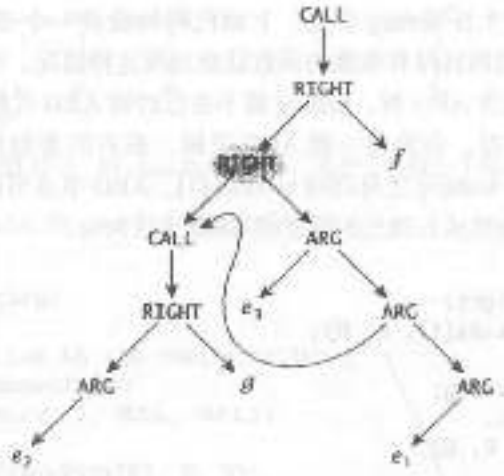


图 12-9 $f(e_1,g(e_2),e_3)$ 的树

将该图与图 12-7 做比较可以发现，图 12-9 中操作数作为一个额外的 RIGHT 节点的右操作数出现。而该 RIGHT 节点用阴影标识，它的左操作数就是嵌套调用 $g(e_2)$ 的树。f 函数的第二个 ARG 节点引用了调用 g 的值。

在 12.5 节描述的处理 CALL 的代码中，listnodes 函数遍历这棵树。当最上面的 CALL 唯一的左操作数被转换时，RIGHT 树使得对 g 的嵌套调用在 f 的所有参数之前被遍历和列出。图 12-10 给出了结果森林。

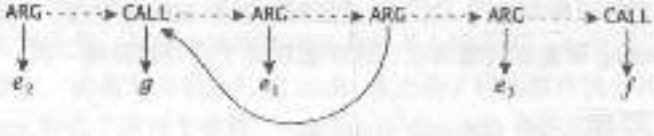


图 12-10 $f(e_1,g(e_2),e_3)$ 的森林

RIGHT 树也用来强制实现表达式 e++ 和 e-- 的正确语义。图 12-11 给出了 postfix 函数为 i++ 构建的树。这些 RIGHT 节点合作以实现在增加 i 之前返回 i 的值，但是实现并不容易。为了强制先计算 INDIR+I，在对 i 的赋值之前，遍历该树，在森林中必须列出它的节点，并且该节点必须表示为 RIGHT 树。列出这个 INDIR 节点就是图 12-11 中下层 RIGHT 节点表示的需要特殊处理的 RIGHT 用法。

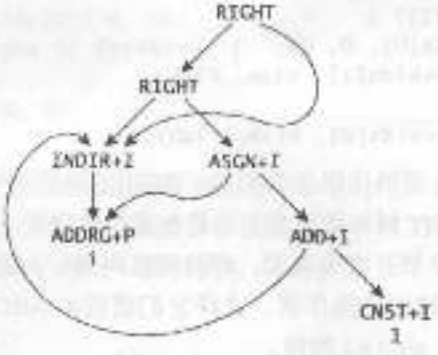


图 12-11 i++ 的树

```
⟨tp is a tree for e++ 263⟩≡                                    261
    tp->kids[0] && tp->kids[1]
  &&  generic(tp->kids[1]->op) == ASGN
  && (generic(tp->kids[0]->op) == INDIR
  && tp->kids[0]->kids[0] == tp->kids[1]->kids[0]
  || (tp->kids[0]->op == FIELD
  &&  tp->kids[0] == tp->kids[1]->kids[0]))
```

这段测试代码表明，对于一个位域的后自增或后自减操作，是用 FIELD 节点代替 INDIR 节点，而且该 FIELD 节点是赋值的目标。

当 e 不是一个位域时，遍历 INDIR 树，其节点在遍历 RIGHT 树的第二个参数之前被列出。

```
⟨generate nodes for e++ 263⟩≡                                  263  261
  if (generic(tp->kids[0]->op) == INDIR) {
     p = listnodes(tp->kids[0], 0, 0);
     list(p);
     listnodes(tp->kids[1], 0, 0);
  }
```

图 5-3 给出了赋值语句 i=*p++ 的森林。计算 p 的右值的 INDIR 节点出现在对 p 的赋值之前。

位域的处理需要解决的问题很多。因为在 dag 中没有 FIELD 节点，FIELD 操作符只在树中出现，listnodes 函数不能列出 FIELD 节点。相反，listnodes 函数必须在 FIELD 树下面找到读取位域所在字的 INDIR 树，然后遍历该树并列出它的节点：

```
⟨generate nodes for e++ 263⟩+≡                                 263  261
  else {
     list(listnodes(tp->kids[0]->kids[0], 0, 0));
     p = listnodes(tp->kids[0], 0, 0);
     listnodes(tp->kids[1], 0, 0);
  }
```

12.7 驱动代码生成

一旦函数的代码表完成，funcdefn 函数就调用接口过程 function 来生成和发送代码。5.10 节介绍了 function 接口函数在编译前端进行的两次函数调用：调用 gencode 函数生成代码，然后调用 emitcode 函数发送它所生成的代码。每个函数都扫描代码表，对每一个代码表入口调用相应的接口函数。

funcdefn 函数会构建两个指向符号表入口的指针数组：callee 数组保存函数内部所见的参数，caller 数组保存的是函数调用者所见的参数。这些数组被传递给 function 函数，function 又将它们传递给 gencode 函数：

```
⟨dag.c functions⟩+≡                                            255  265
  void gencode(caller, callee) Symbol caller[], callee[]; {
    Code cp;
    Coordinate save;

    save = src;
    ⟨generate caller to callee assignments 264⟩
    cp = codehead.next;
```

```
    for ( ; errcnt <= 0 && cp; cp = cp->next)
        switch (cp->kind) {
        case Address:  (gencode Address 265) break;
        case Blockbeg: (gencode Blockbeg 264) break;
        case Blockend: (gencode Blockend 265) break;
        case Defpoint: src = cp->u.point.src; break;
        case Gen: case Jump:
        case Label:    (gencode Gen,Jump,Label 265) break;
        case Local:    (*IR->local)(cp->u.var); break;
        case Switch:   break;
        }
    src = save;
}
```

这里需要对 src 进行赋值，使得在代码生成时进行的诊断能够包含错误表达式的位置。

在对代码表扫描一遍之前，gencode 函数检查 caller 数组和 callee 数组中的符号。对于大多数函数，这些数组中的符号描述了相同的变量。然而，对于字符和短整型参数，编译前端会将它们提升为整数或无符号整数；5.5 节曾给出了这种情况的一个例子。当这种提升发生时，caller 符号类型与 callee 符号类型就不同了，gencode 函数必须生成将 caller 赋值给 callee 的赋值指令。如果 caller 和 callee 的存储类型不同，也要生成这类赋值，例如当编译后端为遵循目标机器的调用约定而改变了 caller 或者 callee 的存储类型时，就会发生这种情况。

这些赋值指令改变了代码表，它们必须插入函数体的第一个入口之前。

```
⟨generate caller to callee assignments 264⟩=                    263
    {
        int i;
        Symbol p, q;
        cp = codehead.next->next;
        codelist = codehead.next;
        for (i = 0; (p = callee[i]) != NULL
                 && (q = caller[i]) != NULL; i++)
            if (p->sclass != q->sclass || p->type != q->type)
                walk(asgn(p, idtree(q)), 0, 0);
        codelist->next = cp;
        cp->prev = codelist;
    }
```

在循环语句之前对 codehead 和 codelist 进行处理的代码将代码表分为两部分：codelist 指向代码表唯一的 Start 入口，而 cp 指向代码表的其余部分。调用 walk 函数将每一个上述赋值指令添加到 codelist 指针所指向的代码表中。赋值指令添加完后，cp 指向的代码表的其余部分重新连接在后面。对代码表的这些操作与在 10.7 节的同法类似，如图 10-2 所示，在恰当的位置为 switch 语句插入选择指令代码。

Blockbeg 和 Blockend 代码表入口通知源代码中复合语句的开始和结束。

```
⟨gencode Blockbeg 264⟩=                                         264
    {
        Symbol *p = cp->u.block.locals;
        (*IR->blockbeg)(&cp->u.block.x);
        for ( ; *p; p++)
            if ((*p)->ref != 0.0)
```

```
        (*IR->local)(*p);
}
```

```
⟨gencode Blockend 265⟩=                                    264
  (*IR->blockend)(&cp->u.begin->u.block.x);
```

与代码块相关的 Env 值的地址被传递给接口函数 blockbeg 和 blockend。编译后端可以用这个值来存储那些必须在语句块结束后恢复的值，例如寄存器组和帧偏移量。

Blockbeg 入口包含一个指针数组，每个指针都指向块中声明的局部变量的符号表入口。这些局部变量是由接口函数 local 通知的。其他局部变量，比如临时变量，出现在 Local 入口中，与上面一样由 local 声明。

Address 入口传递必要的信息，以定义那些依赖局部变量或参数地址的符号和由 address 函数创建的符号。这些符号通过调用接口函数 address 通知：

```
⟨gencode Address 265⟩=                                     264
  (*IR->address)(cp->u.addr.sym, cp->u.addr.base,
     cp->u.addr.offset);
```

对于局部变量来说，这些入口在代码表中位于 Blockbeg 或 Local 入口之后，Blockbeg 或 Local 入口传递它们所依赖的符号。一旦后面这些符号被通知给编译后端，接口函数 address 就会被调用，并在 Address 入口定义符号。这些入口也可以定义依赖已通知的参数的符号。

Gen、Jump 和 Label 入口分别负责传递表示表达式、跳转和标号定义的代码的森林。这些森林被传递给接口函数 gen：

```
⟨gencode Gen,Jump,Label 265⟩=                              264
  if (!IR->wants_dag)
     cp->u.forest = undag(cp->u.forest);
  fixup(cp->u.forest);
  cp->u.forest = (*IR->gen)(cp->u.forest);
```

gen 返回一个指向节点的指针。通常，gen 注释 forest 中的节点，但可能重新组织并返回森林，但是 gen 也可以返回其他信息，只要这些信息可以表示为指向节点的指针。对于森林中的指令，本书所有的编译后端都返回一个指向节点列表的指针。如果 gen 的返回值为空，就不会调用接口函数 emit，这将在下面介绍。

正如上一节所介绍的，在 Gen 入口中的森林可以含有被多次引用的节点，因为它们表示的是公共子表达式。如果接口标志 wants_dag 等于 1，gen 就会传递这种类型的节点。然而，如果 wants_dag 等于 0，undag 函数就会生成把公共子表达式存储在临时变量中的赋值指令，并用对临时变量的引用替换对计算表达式节点的引用。这将在 12.8 节详细介绍。

比较操作符和跳转的 syms[0] 域指向标号在符号表中的入口。这些标号可能是真正标号的同名标号，这在 10.9 节已经介绍过。fixup 函数查找这些节点，并将其 syms[0] 域指向真正的标号。

```
⟨dag.c functions⟩+≡                                  263 266
  static void fixup(p) Node p; {
     for ( ; p; p = p->link)
        switch (generic(p->op)) {
        case JUMP:
           if (p->kids[0]->op == ADDRG+P)
              p->kids[0]->syms[0] =
                 equated(p->kids[0]->syms[0]);
```

```
        break;
      case EQ: case GE: case GT: case LE: case LT: case NE:
        p->syms[0] = equated(p->syms[0]);
      }
}
```

若 equatelab 函数使得 L_1 作为 L_2 的同名标号，就设置 L_1 符号表入口的 u.l.equatedto 域为 L_2 的符号表入口。equated 函数沿着由该域构成的符号表进行搜索，如果存在，在表的最后就会找到真正的标号：

```
⟨dag.c functions⟩+=                                    265 266
  static Symbol equated(p) Symbol p; {
     while (p->u.l.equatedto)
        p = p->u.l.equatedto;
     return p;
  }
```

JUMP 和比较操作符总是作为根节点出现，因此只有在森林的根节点 fixup 函数才需要进行检查。

一旦 gencode 函数返回，接口过程 function 就获得了它需要的所有信息，如帧的大小和已使用寄存器的数目，以此来生成函数的头代码。当它准备发送已生成的代码时，它就调用 emitcode 函数：

```
⟨dag.c functions⟩+=                                    266 268
  void emitcode() {
     Code cp;
     Coordinate save;

     save = src;
     cp = codehead.next;
     for ( ; errcnt <= 0 && cp; cp = cp->next)
        switch (cp->kind) {
        case Address: break;
        case Blockbeg: ⟨emitcode Blockbeg⟩ break;
        case Blockend: ⟨emitcode Blockend⟩ break;
        case Defpoint: ⟨emitcode Defpoint 266⟩ break;
        case Gen: case Jump:
        case Label:    ⟨emitcode Gen,Jump,Label 267⟩ break;
        case Local:    ⟨emitcode Local⟩ break;
        case Switch:   ⟨emitcode Switch 267⟩ break;
        }
     src = save;
  }

⟨emitcode Defpoint 266⟩=                                  266
  src = cp->u.point.src;
```

Defpoint、Blockbeg、Blockend 和 Local 的代码表入口并不发送代码。但是，如果指定了 lcc 的 -g 选项，就会调用 stab 接口函数为调试器发送符号表信息。

Gen、Jump 和 Label 入口存放了由接口函数 gen 返回的森林，emitcode 函数将非空森林传给 emit 接口函数：

```
⟨emitcode Gen,Jump,Label 267⟩=                                    266
  if (cp->u.forest)
     (*IR->emit)(cp->u.forest);
```

Switch 代码表入口存放了由 swcode 函数生成的 switch 语句的分支表。这些入口中的 u.swtch.values 和 u.swtch.labels 数组保存了数目为 u.swtch.size 的值－标号对，这些值－标号对就构成了分支表。emitcode 函数为分支表生成一个全局变量，该全局变量的符号表入口在 u.swtch.table 域中，同时还将该分支表初始化为标号对应的地址。

```
⟨emitcode Switch 267⟩=                                           266
  {  int i;
     unsigned k = cp->u.swtch.values[0];
     defglobal(cp->u.swtch.table, LIT);
     for (i = 0; i < cp->u.swtch.size; i++, k++) {
        for ( ; k < (unsigned)cp->u.swtch.values[i]; k++)
           (*IR->defaddress)(equated(cp->u.swtch.deflab));
        (*IR->defaddress)(equated(cp->u.swtch.labels[i]));
     }
     swtoseg(CODE);
  }
```

在 u.swtch.values 和 u.swtch.labels 中的值－标号对按照值的升序排列。但这些值可能不是连续的。没有的值使用 u.swtch.deflab 的缺省标号。

12.8 删除多次引用的节点

编译前端构建分析树，其中有一些树是 dag。listnodes 函数获得这些树并构建 dag，这样就可以消除公共子表达式。这一节将介绍 undag 函数，它接受 dag 并将其变换成合适的树，尽管变换结果仍然称为 dag。由于 lcc 对相关术语的不恰当混用，我们最好记住“树”表示编译前端建立和处理的中间表示，而“dag”表示传递给后端并由后端处理的中间表示。

我们可以删除 listnodes 函数，但是这也将会牺牲公共子表达式的消除，而删除公共子表达式对于生成的代码的质量有重要的影响。lcc 早期版本采用的就是相反的做法：编译前端直接构建 dag。因为 dag 使得代码的转换（例如，simplify 函数执行的转换）更加复杂，也使得维护引用次数的工作容易出错，目前的方案已经放弃了这种做法。

对于表示公共子表达式的节点，在同一个森林中至少有两个其他节点的 kids 数组元素指向它，它的 count 域记录了指向它的指针的个数。编译后端可以将通过接口函数 gen 传递来的森林的每个 dag 直接转换生成代码。但这些多次引用的节点一般会使代码生成和别名寄存器分配复杂化。因此，一些编译器就删除了这些节点。这种工作可由编译前端完成，也可以由代码生成器完成。编译器生成了将它们的值赋给临时变量的代码，并只把对它们节点的引用替换成对临时变量的引用。在 12.7 节中曾提及，将接口标志 wants_dag 设置为 0，会使 lcc 的编译前端生成这类赋值指令，从而删除多次引用的节点。如果 wants_dag 的值等于 0，编译前端也为 CALL 的返回值生成赋值语句，即使它们只被引用一次，因为列出 CALL 节点就导致代码表的一个隐式引用。本书中所有的代码生成器都设置 wants_dag 为 0。

在将森林传递给接口函数 gen 之前，gencode 为代码表中的每一个森林调用 undag 函数。undag 函数构建并返回一个新的森林。当 undag 访问老森林中的每一个节点时，将增加必要的赋值语句到新森林。

```
⟨dag.c data⟩+≡                                                    260
  static Node *tail;
⟨dag.c functions⟩+≡                                               256 269
  static Node undag(forest) Node forest; {
     Node p;

     tail = &forest;
     for (p = forest; p; p = p->link)
        if (generic(p->op) == INDIR
        || iscall(p) && p->count >= 1)
           visit(p, 1);
        else {
           visit(p, 1);
           *tail = p;
           tail = &p->link;
        }
     *tail = NULL;
     return forest;
  }
```

if语句的两个分支分别处理那些在新森林中不作为根节点出现的节点以及作为根节点出现的节点。列出的INDIR节点以及被其他节点引用的调用节点，在新森林中都被对临时变量的赋值指令所替换。其他列出的节点，比如用于比较的节点，仅起副作用的JUMP、LABEL、ASGN和CALL节点，都被添加到新森林中。这里，如果接口标志mulops_calls等于1，调用中就包含乘法操作符：

```
⟨dag.c macros⟩=
  #define iscall(p) (generic((p)->op) == CALL \
     || IR->mulops_calls \
     && ((p)->op==DIV+I || (p)->op==MOD+I || (p)->op==MUL+I \
     ||  (p)->op==DIV+U || (p)->op==MOD+U || (p)->op==MUL+U))
```

visit函数遍历dag，查找那些被多次引用的节点，也就是那些count域大于1的节点。当第一次遇到这样的节点时，visit函数就生成一个临时变量，然后构建赋值将该节点赋给临时变量，并将赋值添加到新森林中。当再一次遇到该节点时，不管是在同一个dag还是在接下来的dag中，visit函数都会将对该节点的引用替换成一个新节点，新节点引用相应的临时变量。这样，在新森林中，对临时变量的赋值只出现在第一次引用它的dag的根节点之前。

下面的例子有助于理解visit函数。下列语句的森林如图12-1所示：

```
register int n, *q;
n = *q++ = f(n, n);
```

图12-12中有5个公共子表达式，也就是说有5个多次引用的节点：q和n的左值、q和n的右值以及对f函数的调用。图12-13给出了undag函数返回的森林。这些公共子表达式中只有两个被临时变量所取代：q的右值赋值给t2，调用f函数的返回值赋值给t3。重新计算q和n的左值十分简单，它们没有对应的临时变量。所以图12-13中有两个（ADDRLP q）节点和三个（ADDRLP n）节点。如下所示，n是一个寄存器，复制引用n的INDIR节点的代价很小，n的右值也没有对应的临时变量，所以在图12-13中有两个标识为（INDIRI（ADDRLP n））的dag。如果上面的语句被改为下面的语句，图12-13中所示的就是可能生成的森林：

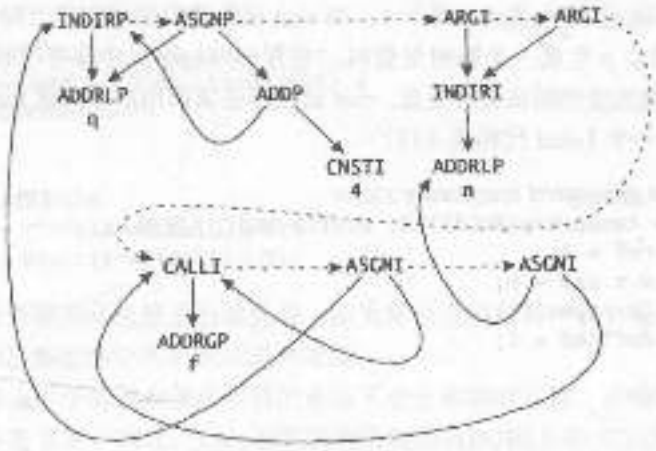


图 12-12 n=*q++=f(n,n)的森林

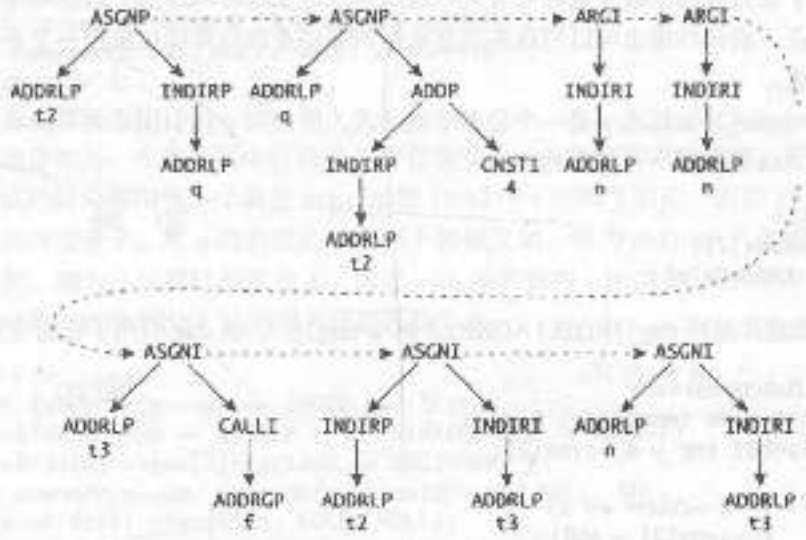


图 12-13 当 wants_dag 等于 0 时，n=*q++=f(n,n)的森林

```
register int n, *q, *t2, t3;
t2 = q;
q = *t2 + 1;
t3 = f(n, n);
*t2 = t3;
n = t3;
```

visit 函数遍历以 p 为根的 dag，返回 p 或者返回保存 p 的值的临时变量所对应的节点：

```
(dag.c functions)+≡                                    ↑268 270↓
  static Node visit(p, listed) Node p; int listed; {
      if (p)
          (visit 270);
      return p;
  }
```

当 undag 函数调用 visit 函数时，listed 等于 1。当 visit 函数递归调用它自己时 listed 等于 0。

当 visit 函数为节点 p 生成一个临时变量时，它在 p->syms[2] 中保存了该临时变量的符号表入口；否则，编译前端无法使用该临时变量。visit 函数还必须调用接口函数 local 通知该临时变量，就好像该临时变量有一个 Local 代码表入口：

```
⟨p->syms[2] = a generated temporary 270⟩=                    272
  p->syms[2] = temporary(REGISTER, btot(p->op), LOCAL);
  p->syms[2]->ref = 1;
  p->syms[2]->u.t.cse = p;
  (*IR->local)(p->syms[2]);
  p->syms[2]->defined = 1;

⟨temporaries 270⟩=                                           28
  struct {
     Node cse;
  } t;
```

保存公共子表达式的临时变量符号表入口通过非空的 u.t.cse 域标识。这些域指向了表示临时变量的值的节点。编译后端也可以用这些信息来标识那些重新计算要比重用一个寄存器代价更小的公共子表达式。

非空的 p->syms[2] 也标志 p 是一个公共子表达式，所以对 p 的引用必须替换成对临时变量的引用，这就是 visit 函数第一步要做的：

```
⟨visit 270⟩=                                          271  269
  if (p->syms[2])
     p = tmpnode(p);
```

tmpnode 函数构建并返回 dag (INDIR (ADDRLP p->syms[2]))，该 dag 引用了临时变量的右值：

```
⟨dag.c functions⟩+=                                   269 272
  static Node tmpnode(p) Node p; {
     Symbol tmp = p->syms[2];

     if (--p->count == 0)
        p->syms[2] = NULL;
     p = newnode(INDIR + ⟨type suffix for tmp->type 270⟩,
        newnode(ADDRLP, NULL, NULL, tmp), NULL, NULL);
     p->count = 1;
     return p;
  }

⟨type suffix for tmp->type 270⟩=                      270 272
  (isunsigned(tmp->type) ? I : ttob(tmp->type))
```

p->count 表示的是引用 p 的次数。对每一次引用，tmpnode 函数都将 p->count 的值减 1，如果 p->count 等于 1，则清空 p->syms[2]。将 p->syms[2] 的值重新初始化为空，以便编译后端的使用。

对于那些仅被引用了一次的节点以及有副作用的函数调用，visit 函数遍历并重写它们的操作数：

```
⟨visit 270⟩+≡                                          270 271  269
  else if (p->count <= 1 && !iscall(p)
  ||       p->count == 0 &&  iscall(p)) {
    ⟨visit the operands 271⟩
  }
```

```
⟨visit the operands 271⟩≡                              271 272
  p->kids[0] = visit(p->kids[0], 0);
  p->kids[1] = visit(p->kids[1], 0);
```

被其他节点引用的函数调用在这里不会被处理，因为即使它们只有一次引用，也会替换成赋值指令。如下所示，它们也被当作公共子表达式来处理。

由以上说明可见，对于局部变量和参数的地址不会生成临时变量，通常重新计算它们的地址比用一个寄存器保存更划算。因此，visit 函数就为所有的 ADDRLP 和 ADDRFP 节点构建并返回一个新节点：

```
⟨visit 270⟩+≡                                          271 271  269
  else if (p->op == ADDRLP || p->op == ADDRFP) {
    p = newnode(p->op, NULL, NULL, p->syms[0]);
    p->count = 1;
  }
```

类似地，通常在另一个寄存器中存储某个寄存器变量的右值是非常浪费的。最好为每一个对寄存器 rvalue 值的引用都构造一个新的 dag。如图 12-13 中 n 的两个引用。在图 12-13 中，q 的值被复制到一个临时变量中。对 q 的右值的引用则不能被复制，因为 INDIRP 已经被列出，这表明该值必须被复制，因为 q 可能已被改变了。因此 visit 函数查找 (INDIR (ADDRxP v)) 模式，其中 v 是一个寄存器，清除 INDIR 已经被列出的那些节点：

```
⟨visit 270⟩+≡                                          271 271  269
  else if (generic(p->op) == INDIR && !listed
  && (p->kids[0]->op == ADDRLP || p->kids[0]->op == ADDRFP)
  &&  p->kids[0]->syms[0]->sclass == REGISTER) {
    p = newnode(p->op, newnode(p->kids[0]->op, NULL, NULL,
      p->kids[0]->syms[0]), NULL, NULL);
    p->count = 1;
  }
```

这种情况也说明了为什么 undag 函数不能早一点调用；例如，不能在 walk 函数中调用。直到编译后端处理到该函数，其局部变量和参数的存储类型才可以确定。一旦处理到该函数，funcdefn 函数就调用 checkref 函数，checkref 会把经常存取的局部变量和参数的存储类型改成 REGISTER。如果 undag 函数在 walk 函数中被调用，它就会为自动局部变量和参数生成临时变量，而它们可能会在后面变成寄存器存储类型的。

最后两种情况覆盖了 INDIRB 节点和公共子表达式的节点。寄存器不可能保存结构，所以就没有将它们复制到临时变量中；visit 函数只复制了 INDIRB 节点：

```
⟨visit 270⟩+≡                                          271  269
  else if (p->op == INDIRB) {
    --p->count;
    p = newnode(p->op, p->kids[0], NULL, NULL);
```

```
        p->count = 1;
        ⟨visit the operands 271⟩
    } else {
        ⟨visit the operands 271⟩
        ⟨p->syms[2] ← a generated temporary 270⟩
        *tail = asgnnode(p->syms[2], p);
        tail = &(*tail)->link;
        if (!listed)
            p = tmpnode(p);
    }
```

else 子句是处理第一次遇到公共子表达式的情况。在遍历了操作数之后，如上文所介绍的，visit 函数就生成一个临时变量，并调用：

```
⟨dag.c functions⟩+≡                                          270
    static Node asgnnode(tmp, p) Symbol tmp; Node p; {
        p = newnode(ASGN + ⟨type suffix for tmp->type 270⟩,
            newnode(ADDRLP, NULL, NULL, tmp), p, NULL);
        p->syms[0] = intconst(tmp->type->size);
        p->syms[1] = intconst(tmp->type->align);
        return p;
    }
```

为该临时变量生成一个赋值指令。然后将该赋值指令添加到新的森林中。这些代码就是负责生成图 12-13 中对 t2 和 t3 的赋值。如果 listed 等于 0，p 被其他节点所引用，所以 visit 函数必须返回一个对临时变量的引用。否则，p 只在旧森林中被引用，该引用并未包含在 p->count 中，也就不会代表一次对临时变量的引用。

深入阅读

使用代码表来表示函数代码是 lcc 编译器的一大特点。流图是传统的函数代码表示方式。流图在传统的编译器教材中都将阐述，例如 Aho，Sethi and Ullman（1986）。流图中的节点就是基本块，而边则表示从基本块到后续块的分支。流图这种表示方式通常用来进行优化，但是，lcc 编译器不进行优化。例如，许多过程内优化算法发现和改善循环代码都是通过流图进行的。

node 函数用来发现公共子表达式的自底向上哈希表算法也被称为 value numbering 算法，自从 20 世纪 50 年代后期以来，就在编译器中使用。表 12-1 和表 12-2 中的节点的数值就是其相关联的节点的数目。value numbering 算法也用在数据流算法中，以计算流图中可获得的表达式信息。这些信息可以用来删除那些被多个基本块使用的公共子表达式。

12.3 节中为 && 和 || 操作符生成短路代码的方式，与 Logothetis and Mishra（1981）介绍的算法类似。该方法和 lcc 的方法都传播真、假标号。另一种称为回填（backpatching）的方法，传播需要回填指令的列表——即目标未定的跳转。一旦目标明确了，就遍历这些列表填充跳转指令。这种方法对于自底向上的直接语法分析器（Aho, Sethi and Ullman，1986）特别有效。

大多数编译器都是从树生成代码，但是也有一些使用 dag；Aho，Sethi and Ullman（1986）都介绍了从树和 dag 生成代码的相关算法，并权衡了它们的优缺点。早期版本的 lcc 编译器中包含了接受 dag 的代码生成器。这些代码生成器使用一种紧缩的程序语言来描述指令选择，该语言是专门为从 lcc 的 dag 中产生代码而设计的（Fraser，1989）。这种语言可以为 VAX、Motorola 68030、SPARC 和 MIPS 系统编写代码生成器。本书中的所有代码生成器都使用树。

练习

12.1 kill 函数在 buckets 中搜索 p 的右值时，在找到并删除第一个之后，还继续找。试写一段 C 代码，说明在 buckets 中对于 p 为什么同时会有多种 INDIR 节点。提示：考虑类型变换。

12.2 实现 listnodes 函数中处理 OR 操作符的代码 <OR>。

12.3 画出下面语句生成的森林，其中 a 和 b 是整型数组：

```
while (a[i] && a[i]+b[i] > 0 && a[i]+b[i] < 10) ...
```

12.4 实现 <RET>；RET 节点总是根节点。

12.5 实现 <DIV..MOD>；确保你的代码能正确处理接口标志 mulops_calls。

12.6 实现在 12.3 节中介绍的 unlist 函数。

12.7 给出一个条件表达式的例子，处理该条件表达式时，<COND> 调用 equatelab 函数和 unlist 函数来删除转移到分支的分支。提示：嵌套条件表达式。

12.8 对以下形式的代码：

```
if (1) S1 else S2
```

lcc 编译器生成：

```
        S1
        goto L+1
L:      S2
L+1:
```

修改 lcc 编译器，删除 goto 语句和无用的 S_2 代码。

12.9 图 12-5 是赋值语句 w=x.amt=y；的分析树。低层的 ASGN+I 树是单个赋值 x.amt=y 的树。如果对位域所赋的值没有被用到，那么 asgntree 函数为计算赋值语句结果的右操作数而构建的树就浪费了，而且生成了没有必要的代码。一旦编译前端发现树的值没有被用到，它就传递该树给 root 函数，并使用函数返回的树来代替原来的树。参见 expr0 和 expr1。研究并扩展 root 函数，以便尽可能简化位域赋值语句的右端。

12.10 12.4 节中的 asgntree 函数和 listnodes 函数代码合作，通过必要的符号扩展或屏蔽来计算位域赋值语句的结果。其他赋值语句也有类似情况。例如：

```
int i;
short s;
i = s = 0xFFFF;
```

在具有 16 位短整数和 32 位整数的目标机器上，设置 i 为 −1。没有专门的代码完成这类赋值，但是 lcc 能为该赋值生成正确的代码。请解释 lcc 是如何做到的。

12.11 如果 left_to_right 等于 0，画出图 12-7 中树的森林。

12.12 画出下面扩展赋值语句的树和森林：

```
struct { int b:4, c:4; } x;
x.c += x.b++;
```

位域 b 和 c 在同一个字中，所以该字分别被读取和存储两次。

12.13 管理标号及其同名标号是一个 union-find 问题。union-find 问题在 Sedgewick（1990）的著作的第 30 章有详细介绍。通常用压缩路径算法来解决 union-find 问题。试用该算法代替 equatelab 函数。fixup

函数和 equated 函数，考察 lcc 执行时间的改进情况。如果没有明显改进，请解释原因。

12.14 为什么 visit 函数对 ADDRGP 节点的处理与对 ADDRLP，ADDRFP 节点的处理不同？

12.15 如果 depth 等于 0，<AND> 代码开始时会调用 reset 函数。<AND> 在开始时使 depth 的值加 1，结束时将 depth 的值减 1。OR 对应的 case 分支代码也包含相同的操作。这样，depth 就对嵌套的 AND 或 OR 操作符进行了计数。与在 COND 中必须调用 reset 一样，这里也必须调用 reset 函数。参见第 254 页和第 255 页后做的解释。如果不调用 reset，对于下面的赋值语句，lcc 将生成怎样的错误代码？

```
x[i] = (n || (n=i), n);
```

如图 12-3 所介绍的，对于通常使用的 && 和 ||，调用 reset 函数却没有作用，请解释其原因。

构造代码生成器

代码生成器为编译前端提供接口函数，这些接口函数选用与目标机器相关的指令来实现机器无关的中间代码。接口函数也为变量以及临时变量指派寄存器、固定的存储单元或者栈空间（也在存储器中）。

在 lcc 编译后端的设计过程中，始终优先考虑的是全局的简单性。很少有编译教材包含产品代码生成器，但是本书介绍了 3 种代码生成器。一般来说，人工编写一个规模中等的代码生成器需要 1000 到 1500 行的 C 代码。如果我们尽可能地远离与目标机器相关的特性，这个数字就会锐减一半。尽管这样做的代价增加了大约 1000 行与目标机器无关的代码，但是，只要有两个目标机器，就能从这种方法中获得益处。更重要的是，如果我们尽可能多地使用已有（即与机器无关）代码，开发一个新的代码生成器就变得更加容易了。

lcc 将大部分与机器无关的函数重组到一个大的与机器无关的程序中，该程序调用较小的与目标机器相关的程序，这样 lcc 就分离了一部分与目标机器相关的特性。lcc 还通过表格分离了另外一些与目标机器相关的特性。例如，lcc 寄存器分配程序的大部分与机器无关，lcc 将与目标机器相关的数据存放在一个与目标机器无关的结构中，这样，lcc 寄存器分配程序对目标相关的数据的处理就转化成了对存放该数据的目标无关的表格结构的处理。此外，lcc 还使用专门描述特性的语言来描述与目标相关的特性，以分离部分与目标相关的特性，例如，lcc 使用了一种非常适合表达指令选择的语言，该语言包括驱动代码生成器的子语言。

对于代码生成器中与目标机器无关的部分来说，目标机器相关的操作就好比是烧红的煤块儿，只能用钳子间接处理。例如，若某个与机器无关的程序必须发送一条存储指令，那么该时，该程序不能直接调用 print，它必须创建一个 ASGN 的 dag(无环有向图)，然后为该 dag 生成代码；或者该程序时间某个目标相关的函数来发送指令；或者发送一个预定义的目标相关的模板。所有这些方法都需要更多的代码，而不能只调用 print，但是这样做也有好处，可使 lcc 移植到新的目标机器时变得简单。例如，如果寄存器溢出器采用较小的目标无关的部分加上分别对应 3 种目标机器的目标相关部分，整体上可能比 lcc 的机器无关的溢出器需要更少的代码，但是，溢出器调试相当困难，因此，调试一个机器无关的溢出器所花的时间就会比调试 3 个较简单的目标相关的溢出器更少。

下一章将涵盖指令选择、寄存器分配以及其他与机器相关的内容。本章介绍代码生成器的整体组织及其数据结构，也会涉及一些机器无关的松散部分，但是不会深入到指令选择或者寄存器分配的内容。

在本书剩下的章节中，使用术语“树”表示存储在 node 记录中的树结构，而前面的章节中使用了术语 dag 表示从 node 构造的结构。更糊涂的是，前面的章节中还使用了术语“树”来表示多次引用其他节点的结构，这些并不是真正的“树”。在整本书的中间改变术语容易使人迷惑，但其他的方法更糟。最初 lcc 使用了基于 dag 的代码生成器，但是在本书中，代码生成器针对树结构。如果输入不是纯粹的“树”，一些算法就会失败。如果在后面的章节中使用了 dag，那就是错误的。lcc 仍然构建 dag 来消除公共子表达式，但是，本书中的代码清除了 wants_dag 标记。

13.1 代码生成器的组织

表13-1以调用图的形式描述了编译后端的整体组成。程序名的缩进表示程序之间的调用关系。在该表以及本节中，省略了许多细节甚至许多程序。下面的内容只能简单地引导大家，并不能解决所有的问题。

表13-1 简化的编译后端的调用树

例程名	目的
function	产生函数头代码和尾代码，调用genode
genode	解释代码表，并且将树传递给gen
gen	驱动rewrite、prune、linearize和ralloc
rewrite	驱动prelabel、_label和ralloc
prelabel	修改树，以适应寄存器变量和特殊的目标机器
_label	用所有可能的实现标记树
reduce	选择代价最小的实现
prune	从树中删除某些子指令
linearize	为输出排序指令
ralloc	分配寄存器
emitcode	解释代码表，并将节点传递给emit
emit	遍历指令列表，驱动emitasm
requate	删除寄存器到寄存器的复制
moveself	删除将寄存器复制到寄存器自身的指令
emitasm	解释汇编模板，输出大多数指令
emit2	输出由于过于复杂，不适应模板的指令

编译前端调用接口过程function为一个例程生成代码。function决定了如何接收和存储形参，然后调用前端的genode。genode调用编译后端的gen处理代码表中的各个森林。当genode结束返回的时候，后端已经看到整个程序，并计算了栈空间大小以及所要使用的寄存器。进而，function开始产生过程的头代码，调用前端的emitcode，emitcode为代码表中的每个森林调用后端的emit。当emitcode结束返回时，function开始产生程序尾代码并返回。

gen协调程序完成选择指令以及为这些指令分配寄存器临时变量的工作，这些程序包括：rewrite、prune、linearize和ralloc。rewrite为单个树选择相应的指令。prune从树中删除某些子指令（即地址模式可以计算的操作）。这些指令不需要寄存器，消除这些指令可以简化寄存器分配程序。linearize用来安排输出剩余的指令。ralloc接收一个节点并为其分配一个目标寄存器，同时释放那些不再需要的源寄存器。

rewrite协调选择指令的程序：prelabel、_lablel以及reduce。prelabel为每个节点标出适合的寄存器集合，并修改一些树以更明确地标识出读写寄存器变量的节点。_lablel是根据描述目标机器指令的语法自动生成的，它对一个树进行标记，以记录所有可能的使用目标指令的实现方法。reduce选择最为经济的实现方式。

emit协调发送指令的程序以及标识不必发送的指令的程序：emitasm、requate和moveself。requate标识了一些不必要的寄存器到寄存器的数据复制，moveself标识了那些将某个寄存器复制到该寄存器本身的指令。emitasm解释那些类似于printf格式的字符串的汇编程序模板。对于一些太复杂而不适合模板的指令，emitasm则交给目标相关的emit2程序处理。

13.2 接口扩展

编译后端的程序可以为两类：目标相关的或者机器无关的，后端私有的或者前端可见的。这两种分类合起来就将编译后端程序分成4种。下表给出了每种类型的一个例子（例程从表13-1中选出）。

例程名	私有的	目标相关的
gen	否	否
function	否	是
rewrite	是	否
_label	是	是

第5章介绍了公共接口。本节将概括介绍后端的私有内部接口；第16章到第18章提供了实现这些私有接口的例子，有助于回答更深入一些的问题。

公共接口的4个例程 blockbeg、blockend、emit 和 gen 都是与目标机器无关的。它们可以移到编译前端去，但是这样做会使编译前端在使用不同的代码生成技术时变得复杂。我们可以重写公共接口中的所有例程或是重写除 blockbeg、blockend、emit 与 gen 之外的所有例程并实现私有接口，将 lcc 重定向到新的目标机器。

Xinterface 结构扩展了接口记录：

```
⟨config.h 277⟩=                                                    279
  typedef struct {
    ⟨Xinterface 277⟩
  } Xinterface;
```

该类型收集了所有与机器相关的数据和例程，这些数据和例程都是编译后端的与目标无关的部分生成代码时所需要的。对于编译后端来说，该类型对于目标无关部分的意义，就如同接口记录主体对于编译前端的意义一样。

首先处理那些有助于为 ASGNB 和 ARGB 生成高效代码的特性。这些特性需要复制内存块。若需要复制大量的块，lcc 通过生成循环来完成；否则，由于循环的开销要比复制8个字节块的开销大，lcc 将短循环展开。块复制生成程序一部分与机器相关，一部分与机器无关。机器相关部分是一个小整数和3个过程：

```
⟨Xinterface initializer 277⟩=                                  295  336 363 390
  blkfetch, blkstore, blkloop,
```

代码生成器不需要调用块复制生成程序，例如，第18章中的代码生成器使用 X86 块复制指令，因此，它只实现了上述例程的基本部分（stub）。

整数 x.max_unaligned_load 给出了目标机器按非对齐方式可以存取的最大字节数：

```
⟨Xinterface 277⟩=
  unsigned char max_unaligned_load;
```

例如，SPARC 结构中没有实现非对齐方式取数，因此，x.max_unaligned_load 等于1，即只有取字节指令不需要对齐。但是，MIPS 结构支持双字节和四字节的非对齐方式取数，因此，x.max_unaligned_load 等于4。

过程 x.blkfetch 产生从给定单元取数到寄存器的代码：

```
⟨Xinterface 277⟩+=                                          277 278  277
  void (*blkfetch) ARGS((int size, int off, int reg, int tmp));
```

blkfetch 产生的代码首先将寄存器 reg 的值与偏移常量 off 相加得到一个地址，然后从该地址单元中取出 size 字节的数据，载入寄存器 tmp 中。过程 x.blkstore 产生代码，将某个寄存器数据存储到给定单元：

```
(Xinterface 277)+≡                                          277 278  277
  void (*blkstore) ARGS((int size, int off, int reg, int tmp));
```

blkstore 产生的代码将 size 字节的数据从寄存器 tmp 中存储到给定单元中，单元地址由寄存器 reg 与偏移量 off 相加得到。

过程 x.blkloop 生成一个循环来复制内存中的块：

```
(Xinterface 277)+≡                                          278 278  277
  void (*blkloop)  ARGS((int dreg, int doff,
                         int sreg, int soff,
                         int size, int tmps[]));
```

x.blkloop 产生一个循环复制内存中 size 字节的数据。源地址为寄存器 sreg 与偏移量 soff 相加之和，目标地址为寄存器 dreg 与偏移量 doff 相加之和。tmps 是一个由 3 个整数组成的数组，表示实现该循环可用的寄存器。

在块复制生成程序的接口之后，是指令选择程序的接口：

```
(Xinterface 277)+≡                                          278 278  277
  ⟨interface to instruction selector 295⟩
```

这段程序包含了与机器无关的 gen 和 emit 例程所需的大部分与目标机器相关的代码和数据。这段程序是根据一个紧缩规范（specification）自动生成的。这样，重定向时只需要书写新的规范，而不必直接编写接口代码和数据。如果没有预备知识，不论是规范，还是指令选择程序的接口，都难以描述清楚。第 14 章的引言部分将详细介绍这一点。

x.emit2 产生那些不能通过产生简单的指令模板来处理的指令：

```
(Xinterface 277)+≡                                          278 278  277
  void (*emit2) ARGS((Node));
```

每种机器以及许多调用约定（calling convention），都包含一些没有 emit2 的转义子句就很难处理的特性。

x.doarg 计算分配给下一个参数的寄存器或者栈单元：

```
(Xinterface 277)+≡                                          278 278  277
  void (*doarg) ARGS((Node));
```

编译后端要对树组成的森林进行多遍扫描。第一遍扫描时，每当遇到 ARG 节点就调用 x.doarg。lcc 需要 doarg 来产生符合复杂的调用约定的代码。

x.target 标记那些必须送到特定寄存器中计算的叶节点：

```
(Xinterface 277)+≡                                          278 279  277
  void (*target) ARGS((Node));
```

例如，返回值必须存放到返回寄存器中，还有一些机器需要把除数及余数送到固定寄存器中。对这些节点的标记是通过对节点的 syms[RX] 赋值来实现的，syms[RX] 登记了存放该节点的计算结果的寄存器。将会在 13.5 节中详细介绍。

x.clobber 使被某条给定指令破坏的所有寄存器溢出到存储器，在以后需要这些寄存器的值时重新加载：

```
(Xinterface 277)+≡                                    278  277
  void (*clobber) ARGS((Node));
```

该过程通常根据节点的操作码进行分别处理；每种情况都要调用 spill，spill 是一个与机器无关的过程，它保存或者恢复指定的寄存器集合。

13.3 上行调用

正如编译后端要使用编译前端的一些代码和数据一样，编译后端中与目标机器相关的代码也会使用编译后端中的一些与目标机器无关的代码和数据。大部分通过上行调用（upcall）访问的编译前端中的程序都很简单，这些程序是在（或靠近）调用图的叶节点处，所以在第 5 章中对它们的解释很容易。编译后端内部的类似情况则较复杂，一般来说需要利用上下文来描述它们。本书在这里对它们进行总结，以便将 lcc 重定向到新的目标机器时可以在一个地方找到所有的程序，也可以查阅第 16 章到第 18 章中使用的例子。事实上，对 lcc 重定向最好的方式就是通过对某个已存在的代码生成器进行修改来实现；有一个完整的上行调用例子的集合是很吸引人的。

```
⟨config.h 277⟩+≡                                      277 280
  extern int      askregvar  ARGS((Symbol, Symbol));
  extern void     blkcopy    ARGS((int, int, int,
                                   int, int, int[]));
  extern int      getregnum  ARGS((Node));
  extern int      mayrecalc  ARGS((Node));
  extern int      mkactual   ARGS((int, int));
  extern void     mkauto     ARGS((Symbol));
  extern Symbol   mkreg      ARGS((char *, int, int, int));
  extern Symbol   mkwildcard ARGS((Symbol *));
  extern int      move       ARGS((Node));
  extern int      notarget   ARGS((Node));
  extern void     parseflags ARGS((int, char **));
  extern int      range      ARGS((Node, int, int));
  extern void     rtarget    ARGS((Node, int, Symbol));
  extern void     setreg     ARGS((Node, Symbol));
  extern void     spill      ARGS((unsigned, int, Node));

  extern int      argoffset, maxargoffset;
  extern int      bflag, dflag;
  extern int      dalign, salign;
  extern int      framesize;
  extern unsigned freemask[], usedmask[];
  extern int      offset, maxoffset;
  extern Symbol   rmap[];
  extern int      swap;
  extern unsigned tmask[], vmask[];
```

13.4 节点扩展

代码生成器的主要操作就是对编译前端的节点进行注释扩展。注释记录了诸如指令选择和寄存器分配的数据。在 node 结构中扩展的域被命名为 x 并具有 Xnode 类型：

```
⟨config.h 277⟩+≡                                    279 282
  typedef struct {
      ⟨Xnode flags 281⟩
      ⟨Xnode fields 280⟩
  } Xnode;
```

指令选择器标识了可以实现该节点的指令和寻址模式，并且使用 x.state 来记录结果：

```
⟨Xnode fields 280⟩≡                                  280  280
  void *state;
```

第 14 章将详细介绍 x.state 域指向的结构所表示的信息。

由指令实现的节点需要寄存器，但是那些通过地址分配实现的节点则不需要寄存器，所以区分这两种类型十分有用。指令选择器必须标识它们。编译后端使用 x.inst 来标识通过指令实现的节点：

```
⟨Xnode fields 280⟩+≡                                 280 280  280
  short inst;
```

如果节点是通过指令实现的，则 x.inst 非零，该值可以帮助标识指令。

x.kids 域存放编译后端形成的指令树：

```
⟨Xnode fields 280⟩+≡                                 280 280  280
  Node kids[3];
```

这棵树与前端的 kids 类似。但是，通过地址模式子指令计算的节点被排除了，如图 1-5 所示。也就是说，x.kids 存储的是图 1-5 中的实线；而 kids 存储的是所有的线。

x.kids 有 3 个元素。因为 lcc 要产生读 3 个源寄存器的 SPARC 和 X86 指令，也就是这些指令通过两个寄存器相加得到的一个地址，再将另一个寄存器存储到该地址所指的单元中。如果 lcc 生成 VAX 代码，只使用的是有 3 个操作数的指令，而每一个操作数都要用到两个寄存器（一个基址寄存器和一个索引寄存器），这样该版本的编译器在 x.kids 中就有 6 个元素。

在某些情况下，代码生成器必须将输出的指令进行排序。编译后端以后序方式遍历排除寻址模式的指令后的指令树，并通过 x.prev 和 x.nex 按照指令执行顺序形成双向链表：

```
⟨Xnode fields 280⟩+≡                                 280 280  280
  Node prev, next;
```

例如，图 13-1 给出了图 1-5 的指令链表。图中忽略了通过 kids 和 x.kids 构成的树。

寄存器分配器使用 x.prevuse 来链接读写相同临时单元的节点：

```
⟨Xnode fields 280⟩+≡                                 280 280  280
  Node prevuse;
```

某些调用约定只使用寄存器集合中前面很少的几个寄存器传递参数，因此编译后端通过在 ARG 节点的 x.argno 域中记录参数的数目处理参数传递：

```
⟨Xnode fields 280⟩+≡                                 280  280
  short argno;
```



图 13-1 转化后的图 1-5

每一个节点扩展都保存了若干表示节点属性的标记。例如，森林中的根节点需要一些特殊处理（例如寄存器分配），因此编译后端使用 x.listed 来标识它们：

```
〈Xnode flags 281〉≡                                    281  280
  unsigned listed:1;
```

寄存器分配器和代码产生器可以对一些节点进行多次遍历，但是它们只能在第一次遍历时分配寄存器和产生该节点的代码，因此它们设置 x.registered 和 x.emitted 的值来防止多次处理：

```
〈Xnode flags 281〉+≡                                  281 281  280
  unsigned registered:1;
  unsigned emitted:1;
```

为了删除某些指令，lcc 会对计算表达式的临时单元进行重新安排。为了实现这些优化措施，编译后端使用 x.copy 来标记所有在寄存器之间进行复制的指令，并使用 x. equatable 来标识将寄存器复制到存放公共子表达式的临时单元的指令：

```
〈Xnode flags 281〉+≡                                  281 281  280
  unsigned copy:1;
  unsigned equatable:1;
```

某些公共子表达式的计算代价非常小，不值得为其分配一个寄存器。为了节省这样的寄存器分配，编译后端使用 x.mayrecalc 来标识那些可被安全地重新计算的公共子表达式节点。

```
〈Xnode flags 281〉+≡                                  281  280
  unsigned mayrecalc:1;
```

编译后端为 node 结构增加两个通用操作码。LOAD 表示从寄存器到寄存器的复制。当一个父节点需要从一个寄存器中输入，而子节点产生了一个不同的寄存器时，编译后端就插入一个

LOAD 节点。例如，如果一个函数被调用并且函数返回值需要赋给一个寄存器变量时，子节点 CALL 就产生一个返回寄存器，而父节点需要 LOAD 将返回结果从返回寄存器复制到该寄存器变量。

如果编译后端为一个局部变量或形参指派了寄存器，它就将该变量的所有 ADDRFP 或 ADDRLP 操作码替换为 VREG。寄存器和存储器引用需要不同的代码，而不同的操作码将告诉我们需要产生什么代码。可以确定的是，在 VREG 节点上一定是 ASGN 或 INDIR 节点；否则，程序就会计算寄存器变量的地址，而这是不允许的。即使程序直接获得了寄存器变量，INDIR 节点也不会从寄存器中删除它来。

目标无关的 Regnode 结构描述了一个目标相关的寄存器：

```
⟨config.h 277⟩+≡                                    ⟨280 282⟩
  typedef struct {
     Symbol vbl;
     short set;
     short number;
     unsigned mask;
  } *Regnode;
```

如果该寄存器被分配以保存一个变量（而不是一个临时变量集），vbl 就指向该变量的符号结构。set 表示寄存器所属的寄存器集合。set 可以处理数目很大的寄存器集合，但是在 lcc 支持的所有当前目标机器中，set 只处理两个寄存器集合：IREG 和 FREG：

```
⟨config.h 277⟩+≡                                    ⟨282 282⟩
  enum { IREG=0, FREG=1 };
```

IREG 和 FREG 用来区别通用寄存器和浮点寄存器。number 中保存的是寄存器号；即使寄存器是用一个名字而不是一个寄存器号（如 X86 汇编程序中）来标识，通常也有相应的数字编码方法，以便二进制代码产生器和调试器使用。mask 的每一位都表示相应的底层硬件寄存器的占用情况。大多数单字节宽度的寄存器只用一位表示，而大多数双字节宽度寄存器正好由两位表示。例如，mask 为 1 表示单字节宽度的寄存器 0，而 mask 为 6 表示的是占用了单字节宽度寄存器 1 和 2 的双字节宽度寄存器。X86 体系结构有一字节、二字节和四字节整型寄存器。因此它的 mask 值有 1、2 或 4 个位为 1。这种表示对于大多数寄存器已经足够了，但并不能表示所有的寄存器集合；参见练习 13.2。

13.5 符号扩展

编译后端还扩展了 symbol 结构。该域被命名为 x 并具有 Xsymbol 类型：

```
⟨config.h 277⟩+≡                                    ⟨282 283⟩
  typedef struct {
     char *name;
     int offset;
     ⟨fields for temporaries 283⟩
     ⟨fields for registers 283⟩
  } Xsymbol;
```

x.name 就是编译后端为该符号产生的名字。对于全局变量，在某些目标机器上，它可能等于 name。对于局部变量和形参，它是一个数字串，其值等于栈偏移量 x.offset。局部变量的偏移量永远是负值，但参数的偏移量可能是正值。因此 x.offset 是有符号整数。

如果当前符号是一个编译前端存储公共子表达式的计算结果的临时变量，那么编译后端将通过 x.lastuse 把所有读或写该表达式的节点连接起来，并用 x.usecount 计算读写的次数：

```
⟨fields for temporaries 283⟩=                                    282
  Node lastuse;
  int usecount;
```

在初始化过程中，编译后端为每一个可分配的寄存器分配一个寄存器符号，表示寄存器分配给了一个带该符号的节点，因此产生器可以采取产生保存在 syms 中的标识符和常量的引制，输出寄存器名和数字。这些寄存器符号使用两种特殊的域：

```
⟨fields for registers 283⟩=                                  283  282
  Regnode regnode;
```

编译后端将 x.regnode 指向描述寄存器的结构，并将 x.name 设置为寄存器名或寄存器号。当后端把一个寄存器分配给某个节点 p 时，就在 p->syms[RX] 中存储相应的寄存器符号。编译后端将 RX 设置为 2，以避免改变编译前端在 syms[0] 和 syms[1] 中传递过来的值：

```
⟨config.h 277⟩+=                                             282 285
  enum { RX=2 };
```

但是，一旦编译前端调用 function 函数，syms 的所有元素就都变成编译后端的属性。编译前端将适应这种改变，编译后端只要认为合适就可以改变这些属性。后端的大部分改变都是针对 Xsymbol 域和 syms[RX] 的，但也有一些改变是针对其他域的。

mkreg 创建并初始化寄存器符号：

```
⟨gen.c functions⟩=                                               284
  Symbol mkreg(fmt, n, mask, set)
  char *fmt; int n, mask, set; {
     Symbol p;

     NEW0(p, PERM);
     p->x.name = stringf(fmt, n);
     NEW0(p->x.regnode, PERM);
     p->x.regnode->number = n;
     p->x.regnode->mask = mask<<n;
     p->x.regnode->set = set;
     return p;
  }
```

stringf 用来创建一个包含寄存器号的寄存器名。例如，如果 i 等于 7，那么 mkreg ("r%d", i, 1, IREG) 就创建一个名为 r7 的寄存器。通常第 29 个寄存器称为 sp 而不是 r29，将使用像 mkreg ("sp", 29, 1, IREG) 这样的调用。

编译后端还要表示寄存器的集合。例如，如果一个节点必须被计算并存放到某个特殊的寄存器中，那么编译后端将使用寄存器来标识该节点；但是如果该节点可以被计算并存放到某个寄存器集合中的任何一个寄存器中，那么编译后端就会用一个表示这个集合的值来标识它。编译后端通过在特定通配符（wildcard symbol）的 x.wildcard 域中存储一个指针向量来表示一个寄存器集合，指针向量的每个元素分别指向包含在寄存器集合内的寄存器符号：

```
⟨fields for registers 283⟩+=                                 283  282
  Symbol *wildcard;
```

例如，如果一个机器有 32 个整数寄存器，编译后端就会分配 32 个寄存器符号，并将它们存储在一个具有 32 个元素的向量中。然后分配一个通配符，并将该向量的地址存储在通配符的 x.wildcard 域中。mkwildcard 函数创建一个寄存器集合符号：

```
⟨gen.c functions⟩+≡                                    283 285
  Symbol mkwildcard(syms) Symbol *syms; {
    Symbol p;

    NEW0(p, PERM);
    p->x.name = "wildcard";
    p->x.wildcard = syms;
    return p;
  }
```

x.name 的值 "wildcard" 不会在 lcc 的输出中出现，但是不管怎样，x.name 都会被初始化，因此产生器不会失效，即使出错；至少也会产生一个有效的寄存器名。

13.6 帧的布局

过程活动记录（也称为帧）保存了一个过程被调用时需要的所有状态信息，包括自动变量、返回地址以及保存的寄存器。栈中为每一个活动过程调用保存了一个帧，这个栈向下增长，即向低地址方向增长。例如，如果 main 函数调用 f 函数，而 f 函数又递归调用它自身一次，这时栈的结构就如图 13-2 中所示。栈结构朝阴影区域增长。

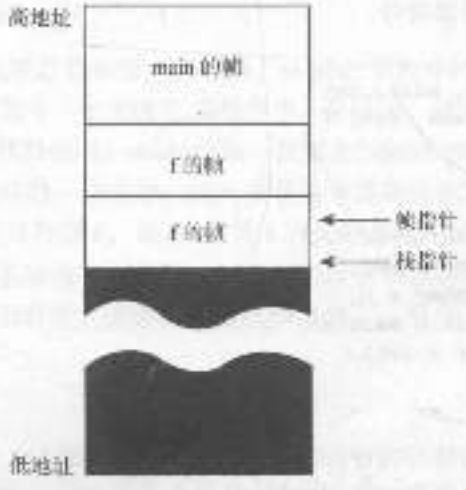


图 13-2 3 个栈帧

一个逻辑帧指针指向一个栈帧内的某个地方。对于所有的目标机器，相对于帧指针来说，局部变量具有负的偏移量。形参和其他数据根据目标机器的约定可能有正的或负的偏移量。图 13-3 给出了一个典型的帧结构。

某些目标机器把帧指针保存在一个物理寄存器中；例如，在图 18-1 中，X86 的帧指针存储在寄存器 ebp 中，而它指向了存储在该帧中的某个寄存器。其他目标机器仅保存栈指针，帧指针表示为栈指针和某个常量的和；例如，在 MIPS 代码生成器中，如果某个程序的帧有 80 字节，它的虚拟帧指针就是地址 80($sp)（即 80 加上栈指针 $sp 的值），而 -4+80($sp) 引用了偏移量为 -4 的局部变量（参见图 16-1）。

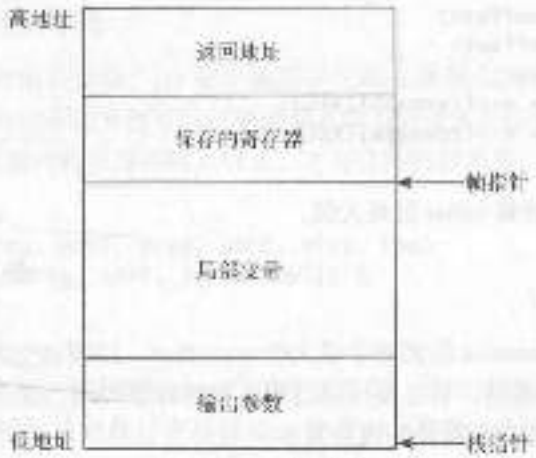


图 13-3 典型的帧

offset 就是最后一个自动变量的栈偏移量的绝对值，mkauto 函数将栈空间进行对齐，以便安排下一个帧：

```
⟨gen.c data⟩≡                                                    285
  int offset;
⟨gen.c functions⟩+≡                                              284 285
  void mkauto(p) Symbol p; {
     offset = roundup(offset + p->type->size, p->type->align);
     p->x.offset = -offset;
     p->x.name = stringd(-offset);
  }
```

使用绝对值就避免了对负整数运行除操作时的取整问题，而且我们并不假设所有的偏移量都是负值，因为对于一些形参来说，它们也确实不是负值。

在每一个块的开始，编译前端都调用 blockbeg 来保存当前栈的偏移量以及每一个寄存器的分配状态：

```
⟨config.h 277⟩+≡                                                283 284
  typedef struct {
     int offset;
     unsigned freemask[2];
  } Env;

⟨gen.c functions⟩+≡                                              285 285
  void blockbeg(e) Env *e; {
     e->offset = offset;
     e->freemask[IREG] = freemask[IREG];
     e->freemask[FREG] = freemask[FREG];
  }
```

blockend 在块结束时会恢复保存的值：

```
⟨gen.c data⟩+≡                                                   285 286
  int maxoffset;

⟨gen.c functions⟩+≡                                              285 286
  void blockend(e) Env *e; {
```

```
    if (offset > maxoffset)
        maxoffset = offset;
    offset = e->offset;
    freemask[IREG] = e->freemask[IREG];
    freemask[FREG] = e->freemask[FREG];
}
```

blockend 还要为当前程序计算 offset 的最大值。

```
(gen.c data)+≡                                          285 286
  int framesize;
```

接口过程 function 设置 framesize 的值等于或大于 maxoffset，以留出空间来存储诸如被调用程序必须保存的寄存器之类的数据，它还会生成过程头和过程尾代码，这些代码根据 framesize 调整栈指针，以分配或释放该程序所有块的栈空间。

每一个程序的栈帧都包含一个参数构建区，这是一片输出参数的存储块，如图 13-3 所示。lcc 可以通过将参数压栈来传递参数，压栈指令就隐含地分配存储块。然而，当前的 RISC 机器没有压栈指令，用多条指令进行模拟也很慢。对于这些机器，lcc 分配一块存储块，并将每个参数移进该存储块的相应单元中。lcc 为每个程序创建一个存储块，使得该块足够满足最大的输出参数集合。

计算参数构建区的偏移量和块大小的代码及数据类似于前面介绍的管理自动变量的代码及数据。argoffset 表示下一个可用的块偏移量，mkactual 将它按照规定的对齐方式取整，返回该结果并更新 argoffset 的值：

```
(gen.c data)+≡                                          286 286
  int argoffset;

(gen.c functions)+≡                                     285 286
  int mkactual(align, size) int align, size; {
      int n = roundup(argoffset, align);

      argoffset = n + size;
      return n;
  }
```

处理每一个参数列表结束的 CALL 节点时都要调用 docall 函数。该函数为下一个参数集合清除 argoffset 值，计算出最大的输出参数块的大小并存于 maxargoffset 中：

```
(gen.c data)+≡                                          286 288
  int maxargoffset;
(gen.c functions)+≡                                     286 287
  static void docall(p) Node p; {
      p->syms[0] = intconst(argoffset);
      if (argoffset > maxargoffset)
          maxargoffset = argoffset;
      argoffset = 0;
  }
```

docall 函数将当前调用参数块的大小记录在 p->syms[0] 中，因此必要时调用者可以将它弹出栈。X86 代码生成器就使用了这种机制。

13.7 生成块复制的代码

ASGNB 和 ARGB 复制存储块。lcc 会生成循环代码来复制大的存储块，但是它将较短的循环展开为线性代码，因为循环的开销可以抵消诸如 8 字节的块复制所带来的数据移动的开销。

blkcopy 函数是块复制代码生成器的入口点。它与目标机器无关，与 blkloop 的参数相同：

```
⟨gen.c functions⟩+≡                                          ↑286 288↓
  void blkcopy(dreg, doff, sreg, soff, size, tmp)
  int dreg, doff, sreg, soff, size, tmp[]; {
     ⟨blkcopy 287⟩
  }
```

blkcopy 产生代码，完成在主存中复制 size 个字节的功能。源地址是通过寄存器 sreg 的值与偏移量 soff 相加得到的，目标地址则通过寄存器 dreg 的值与偏移量 doff 相加得到。tmps 给出了 3 个寄存器中可以被产生的代码来存放临时变量的寄存器的数目。

对于较长的块，blkcopy 将调用 blkloop，但是对于 16 字节或更少字节的块，则是把循环展开；在考虑了其他一些编译器使用的限制值之后，我们选择了 16 作为限制值，虽然可能有些武断。blkcopy 是递归的，所以开始时，它要确认还需要复制的块：

```
⟨blkcopy 287⟩≡                                               ↓287  287
  if (size == 0)
     return;
```

如果剩下的块少于 4 个字节，blkcopy 将调用 blkunroll 以产生复制它们的代码：

```
⟨blkcopy 287⟩+≡                                              ↑287 287↓  287
  else if (size <= 2)
     blkunroll(size, dreg, doff, sreg, soff, size, tmp);
  else if (size == 3) {
     blkunroll(2, dreg, doff,   sreg, soff,   2, tmp);
     blkunroll(1, dreg, doff+2, sreg, soff+2, 1, tmp);
  }
```

如果该块有 4 到 16 个字节，blkcopy 就将 size 向下取整为 4 的倍数（用 size&~3），并调用 blkunroll 以每次复制 4 字节的方式复制那些字节。然后它递归调用自身来处理余下的 0 到 3 个字节：

```
⟨blkcopy 287⟩+≡                                              ↑287 287↓  287
  else if (size <= 16) {
     blkunroll(4, dreg, doff, sreg, soff, size&~3, tmp);
     blkcopy(dreg, doff+(size&~3),
                   sreg, soff+(size&~3), size&3, tmp);
  }
```

循环用来复制超过 16 字节的块：

```
⟨blkcopy 287⟩+≡                                              ↑287  287
  else
     (*IR->x.blkloop)(dreg, doff, sreg, soff, size, tmp);
```

除了多出一个开头的整数参数 k 外，blkunroll 的参数与 blkcopy 和 blkloop 相同。k 表示一次可以复制的字节数，它的取值一定是 1、2 或 4：

```
⟨gen.c functions⟩+≡                                    287 289
  static void blkunroll(k, dreg, doff, sreg, soff, size, tmp)
  int k, dreg, doff, sreg, soff, size, tmp[]; {
     int i;

     ⟨reduce k?288⟩
     ⟨emit unrolled loop 288⟩
  }
```

理想情况下，blkunroll 会交替调用 blkfetch 和 blkstore，以每次复制一个 k 字节的存储块。这种情况下，源或目标地址的对齐数可能就不是 k 的整数倍，而在某些目标机器上，如果该地址不是 k 的整数倍，将不能加载或存储 k 字节单元。blkcopy 的最初调用者将全局变量 salign 和 dalign 设置为源和目标块的对齐数：

```
⟨gen.c data⟩+≡                                         286 289
  int dalign, salign;
```

如果编译器对源或目标的对齐数一无所知，那么它就设置 salign 或 dalign 等于 1，因为所有块的地址都能被 1 整除。使用全局变量 salign 和 dalign 是折中的方法：将它们作为参数进行传递可能更合适，但是该过程的参数已经太多了，而且将这些参数打包为一个结构只会使结果更糟糕。blkunroll 使用这些值和 x.max_unaligned_load 来减小 k 的值。这样如果 k 超过了非对齐取数与源或目标地址对齐取数的最大值，则可以复制较小的块：

```
⟨reduce k?288⟩=                                        288
  if (k > IR->x.max_unaligned_load
  && (k > salign || k > dalign))
     k = IR->x.max_unaligned_load;
```

因此，对于目标地址按照 32 位对齐、源地址按照 16 位对齐的情况来说，只要一次复制 16 位就可以了。复制头 16 位后，源区域剩余的字节将按照 32 位对齐，但是它会接着将余下的目标区域变成 16 位对齐。所以单独这一步并不能帮我们生成更优的代码；参见练习 13.3。

当一个取操作正好在使用其结果的指令之前时，就会引起停顿。blkunroll 解决此类停顿的办法是：先产生两条取数指令，然后再产生两条存数指令。这样，存数指令就不会紧跟在对应的取数指令之后：

```
⟨emit unrolled loop 288⟩=                              288  288
  for (i = 0; i+k < size; i += 2*k) {
     (*IR->x.blkfetch)(k, soff+i,   sreg, tmp[0]);
     (*IR->x.blkfetch)(k, soff+i+k, sreg, tmp[1]);
     (*IR->x.blkstore)(k, doff+i,   dreg, tmp[0]);
     (*IR->x.blkstore)(k, doff+i+k, dreg, tmp[1]);
  }
```

for 循环的每次循环产生取数和存数指令对。当没有剩余指令对时，它就退出循环。如果调用次数为奇数，就再产生一对指令：

```
⟨emit unrolled loop 288⟩+≡                             288  288
  if (i < size) {
     (*IR->x.blkfetch)(k, i+soff, sreg, tmp[0]);
     (*IR->x.blkstore)(k, i+doff, dreg, tmp[0]);
  }
```

图 13-4 给出了 lcc 生成用于复制一个 20 字节的结构体的 MIPS 代码，源地址和目标地址按照 4 字节对齐。第 1 列列出了对上述过程的调用。第 2 列给出了相应的产生代码。tmps 的初始值为 {3,9,10}。第 16 章将介绍 MIPS 指令以及针对 MIPS 的 blkloop、blkfetch 和 blkunroll 过程。MIPS 的 blkloop 过程每次复制 8 个字节。在循环之前，会递归调用 blkcopy 过程来复制剩下的 4 个字节。

```
blkcopy(25, 0, 8, 0, 20, {3,9,10})
 blkloop(25, 0, 8, 0, 20, {3,9,10})          addu $8,$8,16
                                              addu $10,$25,16
  blkcopy(10, 0, 8, 0, 4, {3,9,10})
   blkunroll(4, 10, 0, 8, 0, 4, {3,9,10})
    blkfetch(4, 0, 8, 3)                        lw $3,0($8)
    blkstore(4, 0, 10, 3)                       sw $3,0($10)
   blkcopy(10, 0, 8, 0, 0, {3,9,10})
                                              L.3:
                                              addu $8,$8,-8
                                              addu $10,$10,-8
  blkcopy(10, 0, 8, 0, 8, {3,9,10})
   blkunroll(4, 10, 0, 8, 0, 8, {3,9,10})
    blkfetch(4, 0, 8, 3)                        lw $3,0($8)
    blkfetch(4, 4, 8, 9)                        lw $9,4($8)
    blkstore(4, 0, 10, 3)                       sw $3,0($10)
    blkstore(4, 4, 10, 9)                       sw $9,4($10)
                                              bltu $25,$10,L.3
```

图 13-4 生成结构体复制的代码

13.8 初始化

parseflags 识别那些影响代码生成的命令行选项。例如，-d 选项表示调试输出，它有助于将 lcc 移植到新的目标机器上。本书忽略了产生调试输出信息的函数。

```
⟨gen.c data⟩+≡                                          288 290
  int dflag = 0;

⟨gen.c functions⟩+≡                                     288 297
  void parseflags(argc, argv) int argc; char *argv[]; {
     int i;

     for (i = 0; i < argc; i++)
        if (strcmp(argv[i], "-d") == 0)
           dflag = 1;
  }
```

lcc 可以运行在某个机器（宿主机）上，为另一个机器（目标机器）产生代码。其中一个机器可以是高位优先的机器，而另一个可以是低位优先的机器，这就使得产生 double 类型的常量时变得有些复杂，但是这样会为初始化过程带来一些好处。

lcc 假定它是在具有 IEEE 浮点运算的机器上运行，并为该类机器编译代码。宿主机和目标机器不需要是相同的机器，但是它们都必须采用 IEEE 浮点运算。这种假定虽然是一种限制，但是牺牲很小。

第 5 章中关于接口过程 defconst 的讨论说明了 C 的代码生成器必须自己对浮点数进行编码。也就是说，它们必须产生等价的十六进制常量，并且要避免将浮点常量的文本表示转换为机器内部表示的汇编指令。

lcc 可以为每一个单精度浮点数产生一个字，但是对于双精度浮点数则必须产生两个字。如果 lcc 运行在一个低位优先的机器上，并且为低位优先的机器进行编译，或者如果两台机器都是高位优先的机器，那么对浮点数的编码都是相同的，而代码生成器可以顺序产生组成双精度的两个字。但是，如果一个是高位优先，另一个是低位优先，那么一个要求先产生高位字，而另一个要求先产生低位字。产生这些数据的时候，defconst 必须交换高位和低位。

接口标志 little_endian 对目标机器进行分类，但是接口中没有对宿主机进行分类的信息。lcc 在初始化过程中自动对宿主机进行分类：

```
⟨gen.c data⟩+≡                                          289 307
  int swap;

⟨shared progbeg 290⟩≡                                   337 364 390
  {
     union {
        char c;
        int i;
     } u;
     u.i = 0;
     u.c = 1;
     swap = (u.i == 1) != IR->little_endian;
  }
```

低位优先的机器利用 u.i 的低位的定义 u.c，所以上面的代码在将 u.c 赋值为 1 的同时也将 u.i 设置成了 1。高位优先的机器利用 u.i 的高位的定义 u.c，所以对于 lcc 的 32 位目标机，在将 u.c 赋值为 1 的同时也将 u.i 设置成了 0x01000000。

深入阅读

本书从这一章开始的内容有助于了解最新的计算机体系结构。例如，如果不理解当前机器取数和存数操作是如何相互作用的，那么就无法理解 blkunroll 的取数 取数 存数 存数指令组合的意义。Patterson and Hennessy（1990）介绍了计算机体系结构。

练习

13.1 部分 lcc 程序假设目标机器最多有两个寄存器集合。找出这些程序并将它们适用化，使其可以处理更多的寄存器集合。

13.2 部分 lcc 程序假设目标机器的每一个寄存器集合中最多有 N 个寄存器，这里 N 是一个无符号数。找出这些程序并将它们适用化，使其可以处理更大的寄存器集合。

13.3 图 13-4 的第 1 列给出了在源和目标地址可被 4 整除的情况下，

```
blkcopy(25, 0, 8, 0, 20, {3, 9, 10})
```

的调用步骤。当源和目标地址可被 2 而不是 4 整除时，给出类似的调用步骤。

13.4 lcc 会把那些复制 16 字节或更少字节的结构体的循环进行展开。这种限制显得有些不合理。设计试验方案以确定更适合你的机器的限制值。

选择和发送指令

本书介绍的指令选择器是由程序 lburg 根据紧缩规范自动生成的，lburg 是代码生成器的生成器。lcc 还含有另外一些指令选择器，其中有些是手写的，有些是由另一些代码生成器的生成器产生的，本书不讨论这些选择器。多次遍历节点可以造成 lburg 的代码生成器不能正常运行。因而，尽管树元素的类型是“struct node”，而不是“struct tree”，本书所有的编译后端仍将 wants_dag 置为 0，只对树进行操作。

lburg 接受紧缩规范并产生一个用 C 语言编写的树分析程序，该程序为目标机器选择指令。就像编译前端的分析程序把输入分割成语句、表达式之类的单元一样，树分析程序接受中间代码的主题树，并把它分割成与目标机器指令相对应的程序块，这种分割称为树覆盖。本章称生成的树分析程序为 BURM。lcc 要求每个目标机器都有一个分析程序，所以它在 mips.c、sparc.c 和 x86.c 中各生成一个 BURM。

lburg 规范的核心是树文法。与常规的文法类似，树文法也是一个规则列表，每条规则的左边是一个非终结符，右边是终结符（中间代码中的操作符）和非终结符组成的模式。

典型规则将每种模式与一个寻址形式或者与执行该模式中的操作的指令相关联。常规的模式如同线性的信息串，但树模式是一种结构树，所以树模式必须描述所匹配的操作以及模式中那些操作的相对位置。lburg 的规范就用一个函数符号和圆括号描述了这种结构。例如，模式：

```
ADDI(reg, con)
```

如果节点 ADDI 的第一个子节点递归地匹配非终结符 reg，第二个子节点递归地匹配非终结符 con，那么该模式就在节点 ADDI 处匹配一棵树。规则

```
addr: ADDI(reg, con)
```

规定了非终结符 addr 与上述模式相匹配。规则

```
stmt: ASGNI(addr, reg)
```

规定了只要 ASGNI 节点的每个子节点递归地与 addr 和 reg 相匹配，非终结符 stmt 就与该 ASGNI 节点匹配。

生成的代码生成器，即代码生成器的生成器 lburg 输出的程序，产生一个树覆盖。树覆盖使用文法规则的模式完全覆盖了每一棵输入树，这些文法规则满足了每一个模式在终结符和非终结符上的限制。例如图 14-1 给出了下面的树的覆盖：

```
ASGNI(ADDP(INDIRP(ADDRLP(p)),CNSTI(4)),CNSTI(5))
```

该覆盖使用了上述 addr 和 stmt 两个规则及图中显示的另外一些规则。每个节点旁边的规则识别该覆盖，每个阴影区域对应大多数机器上的一条指令。

描述指令集的树文法经常是有歧义的。例如：典型的寄存器自增可以通过直接给寄存器加 1 实现，也可以通过把 1 放到另一个寄存器中，然后与第一个寄存器相加来完成。我们更希望选择代价最低的实现，所以 lcc 为每个规则设定了一个代价值，并选择总代价最小的树分析。14.2 节就显示了标记有代价的树。

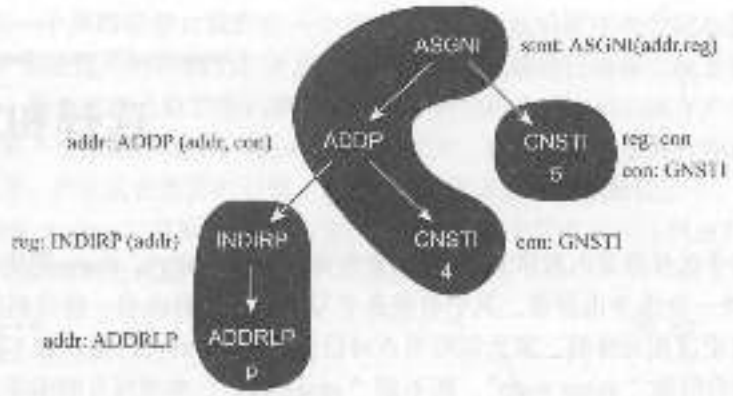


图 14-1 ASGNI (ADDP(INDIRP(ADDRLP(p)), CNSTI(4)), CNSTI(5)) 的覆盖

有时在树的下层，局部覆盖代价较少，但是从根到局部覆盖的代价却可能较大，使得全树的完整覆盖的代价变大。当匹配一棵子树时，我们并不知道那种匹配是全局匹配中较好的，所以代码生成器在每个节点处为每个非终结符记录了最佳匹配。这样在树的较高层处就可以任意选择所有可用的非终结符，甚至有可能是那些在较低层次上代价较高的非终结符。这种记录一组方案然后从中选择一个的技术称为动态规划。

生成的代码生成器对每个主题树进行两遍扫描：第一遍可视为一个采用自底向上方式的标记程序，得到一个能以最小代价覆盖所有子树的模式集；第二遍可视为一个采用自顶向下方式的化简程序，它从第一遍标记程序记录的模式集中选取代价最小的覆盖。这样，代码生成器就可使用代价最小的模式生成代码。

14.1 规范

下面的文法描述了 lburg 规范。term 和 nonterm 分别代表终结符和非终结符。

```
grammar:
  '%{' configuration '%}' { dcl } %% { rule } [ %% C code ]

dcl:
  %start nonterm
  %term | term = integer }

rule:
  nonterm : tree template [ C expression ]

tree:
  term [ '(' tree [ , tree ] ')' ]
  nonterm

template:
  " { any character except double quote } "
```

lburg 规范是按行组织的。单词 "%{"、"%}" 和 "%%" 必须单独一行。每个 dcl 或 rule 必须出现在一行上。configuration 是 C 语言代码，它被原封不动地复制到 BURM 的开头。如果有第二个 "%%"，那么它之后的正文也被原封不动地复制到 BURM 的末尾。

配置是 BURM 与被分析的树之间的接口，它把 NODEPTR_TYPE 定义成可见的指针类型，该指针指向主题树的节点。BURM 调用函数或宏 OP_LABEL(p)、LEFT_CHILD(p) 和 RIGHT_CHILD(p) 读取指针 p 指向的节点的操作符和子节点。

BURM 在主题树的每一个节点处计算并存储一个 void 类型的状态指针。配置段定义了宏 STATE_LABEL(p)，以访问 p 指向的节点的状态域，使用宏是因为 Iburg 需要将它用作左值，另外的一些配置操作也可以作为宏或函数实现。

本书的所有 Iburg 规范共享一种配置：

```
⟨Iburg prefix 293⟩≡                                    335 362 389
  #include "c.h"
  #define NODEPTR_TYPE Node
  #define OP_LABEL(p) ((p)->op)
  #define LEFT_CHILD(p) ((p)->kids[0])
  #define RIGHT_CHILD(p) ((p)->kids[1])
  #define STATE_LABEL(p) ((p)->x.state)
```

%start 指示符用来命名每棵树的“根”必须匹配的非终结符，如果没有 %start 指示符，那么默认的开始符号就是第一条规则定义的非终结符。

%term 声明了终结符，即主题树中的操作符，每个终结符对应一个唯一的正整数操作码。OP_LABEL(p) 返回节点 p 的有效操作码。每个终结符都有固定的数字，这是 Iburg 从使用该终结符的规则中推出的。Iburg 限定终结符最多只能有两个子节点。例如 lcc 的终结符声明包括：

```
⟨terminal declarations 293⟩≡                           335 363 389
  %start stmt
  %term ADDD=306 ADDF=305 ADDI=309 ADDP=311 ADDU=310
  %term ADDRFP=279
  %term ADDRGP=263
  %term ADDRLP=295
  %term ARGB=41 ARGD=34 ARGF=33 ARGI=37 ARGP=39
```

图 14-2 给出了 lcc 的 Iburg 规范的一部分以及大部分机器指令集合的子集。第 2 行和第 3 行声明了终结符。

```
1  %start stmt
2  %term ADDI=309 ADDRLP=295 ASGNI=53
3  %term CNSTI=21 CVCI=85 INDIRC=67
4  %%
5  con:   CNSTI                "1"
6  addr:  ADDRLP               "2"
7  addr:  ADDI(reg,con)        "3"
8  rc:    con                  "4"
9  rc:    reg                  "5"
10 reg:   ADDI(reg,rc)         "6"  1
11 reg:   CVCI(INDIRC(addr))   "7"  1
12 reg:   addr                 "8"  1
13 stmt:  ASGNI(addr,reg)      "9"  1
```

图 14-2 Iburg 规范示例

规则定义树模式时，采用了带括号的前缀形式。每个非终结符代表一棵树。链规则的模式是另一个非终结符，在图 14-2 中，规则 4、5 和 8 都是链规则。

规则描述了目标机器提供的指令集和寻址方式。每条规则有一个汇编代码模板，模板是一个带引号的串，说明了当使用该规则时发送什么样的代码。14.6 节描述了这些模板的格式。在图 14-2 中模板仅仅是规则号。

规则的末尾是一个可选的代价值。链规则的代价值必须是常量，而其他规则就可以使用任意的C表达式，在表达式中$a代表被匹配的节点。例如，规则：

```
con: CNSTU  ""  ($a->syms[0]->u.c.v.u < 256 ? 0 : LBURG_MAX)
```

说明了无符号常量若只占一个字节则代价值为0，否则为LBURG_MAX。所有代价值必须在从0到LBURG_MAX之间（包括0和LBURG_MAX在内）。LBURG_MAX定义为最大的短整数：

```
⟨config.h 277⟩+≡                                              285
  #define LBURG_MAX SHRT_MAX
```

默认的代价值为0。推导的总代价值为推导过程中所应用规则的代价值的总和。树分析程序寻找主题树代价最低的分析，它可以尝试任意组合。

在图14-2中，con匹配常量；addr匹配可被地址计算器计算的树，如ADDRLP或一个寄存器与一个常量求和；rc匹配常量或reg，reg匹配任何可被计算并放进寄存器中的树；规则6描述了一个加法指令，第一个操作数必须为寄存器，第二个操作数必须是寄存器或常量，结果存储在寄存器里；规则7描述了一个取字节指令，扩展符号位，并把结果放到寄存器中；规则8描述了一个取地址到寄存器的指令；stmt匹配用于副作用的树，包括赋值；规则9描述了把寄存器值存到内存单元中的指令，该单元按某种寻址方式寻址。

14.2 标记树

BURM从标记主题树开始，按照自底向上、从左到右的顺序计算以最小代价覆盖树的规则。图14-3显示了下面代码中赋值语句的树：

```
{ int i; char c; i = c + 4; }
```

图14-3中的其他注释描述了标记。（N,C,M）表示规则号为N的规则M的模式与节点匹配的代价是C。C是规则右边的非终结符、相应模式或链规则的代价之和。

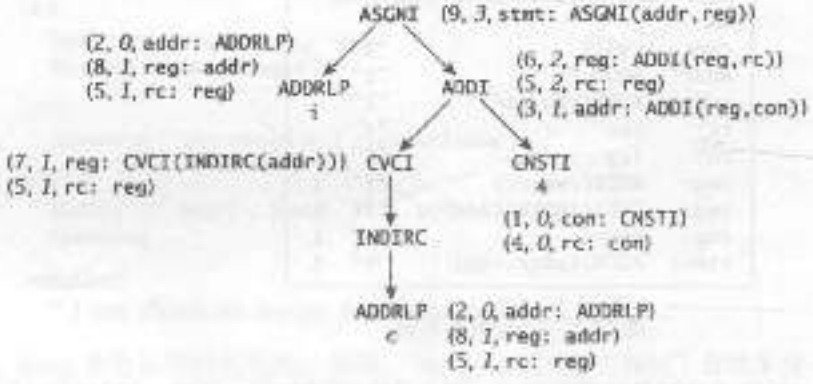


图14-3 标记后的i=c+4的树

例如，图14-2中的规则2与ADDRLP i节点匹配，代价为0，所以该节点标记为（2，0，addr:ADDRLP）。规则8说明只要与addr匹配就可以与reg匹配，但代价值加1，所以该节点也被标记为（8，1，reg:addr）。规则5说明只要与reg匹配就能与rc匹配，且没有额外的代价值，

所以该节点又可被标记为（5,1,rc:reg）。然后，树中更高层的匹配需要 addr，所以不需要链规则。需要链规则的是 CNSTI 节点，它只能与 con 直接匹配，但是它的父节点需要 rc，只有链规则记录了每个 con 同时也是 rc。采用自下而上的匹配程序，下层并不清楚更高层需要哪种匹配，所以程序记录了所有的匹配，由自上而下的化简程序选择需要的匹配。

模式除了描述直接子节点之外还可以描述子树。例如，图 14-2 中的规则 7 就涉及了 CVCI 节点的孙节点，虽然没有单独的模式与 INDIRC 节点匹配，但规则 7 的模式覆盖了这个节点，其代价是 ADDRLPc 与 addr 匹配（使用规则 2）的代价加 1。

只有当 C 的重比规则 M 中非终结符的所有先前匹配都小的时候，节点才有标记（N，C，M）。例如，ADDI 节点用规则 6 匹配 reg，其总代价是 2；它也用规则 3 匹配了 addr，所以禁用链规则 8 得到第二种与 reg 的匹配，其总代价也是 2，而这些与 reg 的匹配中只有一个会被记录。lburg 可以尝试任意组合，所以想要预测哪个匹配被记录下来并不容易，但由于其总代价都一样，所以关系不大。

lburg 生成了函数：

```
⟨BURM signature 295⟩=                                    296
  static void _label ARGS((NODEPTR_TYPE a));
```

函数 _label 对 a 指向的整个主题树进行标记。用零状态来标记未匹配的树，这些树可能有语法错误或不合文法。所有 lburg 生成的名字都用下划线开始，以避免与 BURM 的 C 程序开头和结尾中的名字冲突。这些标识符都是静态的，其地址存放在一个接口记录中，所以 lcc 可以包含多个代码生成器。下面的程序片段为结构声明符收集了所有标识符的声明：

```
⟨interface to instruction selector 295⟩=                 278
  void (*_label) ARGS((Node));
```

另一个片段为记录了静态名字的 C 初始化表达式收集名字：

```
⟨Xinterface initializer 277⟩+≡                    277  336 363 390
  _label,
```

由 lburg 在 BURM 中定义的其他标识符，在上面两个程序片段中都有相应的入口，本书不再重复介绍。

14.3 化简树

BURM 的标记程序自底向上遍历主题树。由于无法预测上层节点将匹配哪条规则，也不知道规则需要哪些非终结符，因此，BURM 采用动态规划的方法，为所有的非终结符都记录了最佳匹配。标记对规则号进行了编码，形成一个向量，每个非终结符对应一个向量。lburg 创建了一个 _state 结构类型，函数 _label 将每个非终结符的最佳（N,C,M）存储在 _state 结构中：

```
struct _state {
   short cost[MAX_NONTERMINALS];
   short rule[MAX_NONTERMINALS];
};
```

cost 向量为每个非终结符存储了最佳匹配的代价，而 rule 向量存储了实现该代价的规则号。（其实上述声明中的一部分与实际情况不一样，但也没有什么问题：lburg 实际上是使用位域压缩了 rule 域，但是 lburg 又提供了函数来提取这些域，所以就不必浪费时间来分析这些编码了。）

lburg 提供了函数 _rule，_rule 接受树的状态标记和一个表示非终结符的整数作为参数：

```
⟨BURM signature 295⟩+≡                                    ↑295 296↓
  static int _rule ARGS((void *state, int nt));
```

该函数根据给定的非终结符，从标记编码的规则号向量中提取某个左边带有该非终结符的规则的序号，如果没有规则与给定的非终结符匹配，则返回 0。

BURM 的第二遍扫描程序，也称为化简程序，自上而下遍历主题树，所以它能够处理标记程序遗漏的上下文信息。树的根必须匹配起始非终结符，所以，化简程序从根节点标记编码的规则号向量中，为起始非终结符提取最佳规则。如果规则模式包括非终结符，那么它们会识别要继续化简的新边界以及与该边界匹配的非终结符。这样，从根开始不断递归，就能找到整棵树的最佳覆盖。下面显示了图 14-3 的这个过程：

```
_rule(root, stmt) = 9
 _rule(root->kids[0], addr) = 2
 _rule(root->kids[1], reg) = 6
  _rule(root->kids[1]->kids[0], reg) = 7
   _rule(root->kids[1]->kids[0]->kids[0]->kids[0], addr) = 2
  _rule(root->kids[1]->kids[1], rc) = 5
   _rule(root->kids[1]->kids[1], con) = 1
```

每个规则的模式都为所有的递归调用识别出主题子树和非终结符。在这里，子树并不一定是当前节点的直接子节点。带有内部操作符的模式将导致化简程序跳过相应的主题节点，而直接到达其孙节点、曾孙节点等。另一方面，每条规则使化简程序以新的非终结符再次访问当前主题节点，所以 x 也可视为其本身的子树。

lburg 用 1 来表示起始非终结符，所以为了初始的根结点层面调用 _rule 时，nt 必须是 1。BURM 定义并初始化了一个数组，该数组标识用于嵌套调用的值：

```
⟨BURM signature 295⟩+≡                                    ↑296 297↓
  static short *_nts[];
```

_nts 是一个由规则号索引的数组。每个元素指向一个以 0 结尾的短整数向量，按照从左到右的顺序表示规则模式的非终结符。例如，下面的代码实现了图 14-2 的 _nts：

```
static short _r1_nts[] = { 0 };
static short _r3_nts[] = { 4, 1, 0 };
static short _r4_nts[] = { 1, 0 };
static short _r5_nts[] = { 4, 0 };
static short _r6_nts[] = { 4, 3, 0 };
static short _r7_nts[] = { 2, 0 };
static short _r9_nts[] = { 2, 4, 0 };

short *_nts[] = {
   0,        /* (no rule zero) */
   _r1_nts,  /* con:  CNSTI */
   _r1_nts,  /* addr: ADDRLP */
   _r3_nts,  /* addr: ADDI(reg,con) */
   _r4_nts,  /* rc:   con */
   _r5_nts,  /* rc:   reg */
```

```
    _r6_nts,  /* reg:  ADDI(reg,rc) */
    _r7_nts,  /* reg:  CVCI(INDIRC(addr)) */
    _r7_nts,  /* reg:  addr */
    _r9_nts,  /* stmt: ASGNI(addr,reg) */
};
```

写一个完整的化简程序只需要 _rule 和 _nts，但是通过增加 _kids 则可以简化许多应用：

```
⟨BURM signature 295⟩+≡                                    296 303
  static void _kids
    ARGS((NODEPTR_TYPE p, int rulenum, NODEPTR_TYPE kids[]));
```

_kids 接受树 p 的地址、规则号以及一个由指向树的指针组成的空向量。该过程假设 p 与所给的规则匹配，并将 p 的子树（其意义如前所述）填入向量，这些子树必须是递归化简后的。kids 不以 null 结束。

下面的代码显示了最小的化简程序，它自下而上，从左至右遍历了最佳覆盖，但是在遍历的过程中没有任何操作。parse 函数先对树做标记，然后开始化简。reduce 函数从 _rule 中获得匹配的规则号，从 _kids 中获得匹配的边界，从 _nts 中得到用于递归调用的非终结符。

```
parse(NODEPTR_TYPE p) {
    _label(p);
    reduce(p, 1);
}

reduce(NODEPTR_TYPE p, int nt) {
    int i, rulenum = _rule(STATE_LABEL(p), nt);
    short *nts = _nts[rulenum];
    NODEPTR_TYPE kids[10];

    _kids(p, rulenum, kids);
    for (i = 0; nts[i]; i++)
        reduce(kids[i], nts[i]);
}
```

这个特殊的化简程序没有对任何节点进行操作。如果节点按照前序进行了处理——如产生代码或分配了一个寄存器，那么处理代码将从化简程序的开头运行。后序处理代码在化简程序的末尾开始运行，顺序代码将自动在 reduce 的递归调用之间运行。化简程序可以以任何顺序递归遍历子树，也可以在递归遍历中穿插执行其他任何操作。

可以以多种方式实现化简程序，比如多遍扫描算法或者独立的单遍扫描算法。lcc 有 3 个化简程序，一个用来识别需要寄存器的节点，第二个产生代码，第三个在调试时输出树的覆盖。它们全都使用 getrule，该函数以断言（assertion）方式将 _rule 打包，并通过 IR 封装间接地址：

```
⟨gen.c functions⟩+≡                                       289 298
  static int getrule(p, nt) Node p; int nt; {
      int rulenum;

      rulenum = (*IR->x._rule)(p->x.state, nt);
      return rulenum;
  }
```

第一个化简程序是为寄存器分配准备的，它对最小的化简程序进行了扩充，以标记那些由指令计算的节点，这些节点可能需要寄存器：

```
⟨gen.c functions⟩+≡                                    297 299
  static void reduce(p, nt) Node p; int nt; {
     int rulenum, i;
     short *nts;
     Node kids[10];

     p = reuse(p, nt);
     rulenum = getrule(p, nt);
     nts = IR->x._nts[rulenum];
     (*IR->x._kids)(p, rulenum, kids);
     for (i = 0; nts[i]; i++)
        reduce(kids[i], nts[i]);
     if (IR->x._isinstruction[rulenum]) {
        p->x.inst = nt;
        ⟨count uses of temporaries 299⟩
     }
  }
```

这里，x.isinstruction 中的 lburg 标记规定产生指令的规则，这些规则与那些产生子指令（比如寻址方式指令）的规则不同，这些功能都是通过测试汇编程序模板实现的，这将在 14.6 节介绍。

上面的 x.inst 不只是一个标记，它还指明负责处理该标记的非终结符。寄存器分配程序将指令树线性化，产生器每次化简一条指令，所以产生器需要记录用于指令化简的非终结符。

reduce 与 reuse 合作恢复过度删除掉的公共子表达式。前端为公共子表达式分配了临时变量，通过引用这些临时变量来避免重复计算。但这在某些方面增加了代价。举个例子，MIPS 的寻址硬件可以无任何代价地给寄存器增加一个 16 位常量，所以当这样的和只是作为一个地址来使用的时候（即用于访问存储器的指令时），把和放进寄存器里只是增加一条指令并多用了一个寄存器。

所以 lburg 扩充了标记程序，以寻找读取寄存器的树，INDIRx(VREGP)。如果寄存器中有一个公共子表达式并且表达式可能更适合重新计算，那么标记程序就用额外匹配来扩展标记。这些匹配就是分配给临时变量的表达式的所有自由匹配的集合。

例如，考虑代码 p->b=q->b，p 在寄存器 23 中，q 在寄存器 30 中，域 b 的偏移量为 4。图 14-4 显示了树的中间代码。

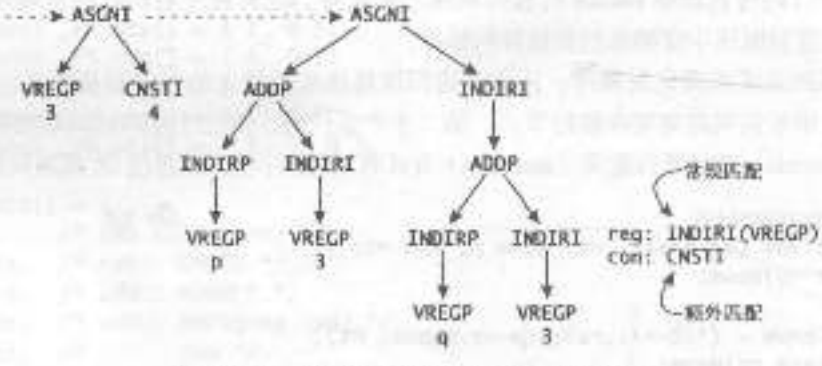


图 14-4 p->b=q->b 中过度删除的公共子表达式

第一棵树把公共子表达式 4 复制到一个临时变量，第二棵树再一次使用临时变量来完成语句。INDIRI 节点的第一个标记由一个典型的模式匹配产生，但第二个标记就是由一个额外匹配

产生。若没有这个额外匹配，lcc 的代码生成器将产生 5 条指令：

```
la   $25,4          取常量 4 到寄存器 25
add  $24,$30,$25    计算 q->b 的地址并存到寄存器 24
lw   $24,($24)      取 q->b 的值到寄存器 24
add  $25,$23,$25    计算 p->b 的地址并存到寄存器 25
sw   $24,($25)      存 q->b 的值到 p->b
```

该额外匹配启用了另外几个匹配，总共节省了 3 条指令和一个寄存器：

```
lw $24,4($30)       取 x[0] 的值到寄存器 24
sw $24,4($23)       存储寄存器 24 到 x[0]
```

lcc 的化简程序调用 reuse(p,nt) 来检查使用非终结符 nt 的 p 节点的化简是否用到额外匹配，如果是，reuse 就返回一个公共子表达式而不是 p。此时，化简程序就对公共子表达式进行后处理，并忽略临时变量：

```
⟨gen.c functions⟩+≡                                    298 300
  static Node reuse(p, nt) Node p; int nt; {
     struct _state {
        short cost[1];
     };
     Symbol r = p->syms[RX];

     if (generic(p->op) == INDIR && p->kids[0]->op == VREG+P
     && r->u.t.cse && p->x.mayrecalc
     && ((struct _state*)r->u.t.cse->x.state)->cost[nt] == 0)
        return r->u.t.cse;
     else
        return p;
  }
```

第一个返回语句有效地忽略了树 p，重用公共子表达式的定义。如果 p 使用了公共子表达式，那么该子表达式肯定已经被标记过。所以调用 reuse 的化简程序并不会漫无目标。如果要获得非终结符 nt 与树进行匹配的代价，上述 _state 的类型转换是不可避免的措施。除了这里，本书其他地方没有再介绍 state 记录的形式，它只用于根据 Iburg 规范而自动生成的代码。只要理解了标记就很容易理解它了。实际的、与目标相关的代价向量的长度在这里并不知道，也不需要知道。所以声明中可假设长度为 1。

reduce 还计算了每个临时变量剩余的使用次数：

```
⟨count uses of temporaries 299⟩≡                       298
  if (p->syms[RX] && p->syms[RX]->temporary) {
     p->syms[RX]->x.usecount++;
  }
```

如果 reuse 留下了不再使用的临时变量，那么寄存器分配程序将删去装载该临时变量的代码。

最早的 reuse 版本是通过每次处理一个类型后缀来实现的，这说明了至少在一些 C 程序中什么是真正重要的。lcc 有一个测试平台，该测试平台包含 18 个程序，大概 9000 行，并且还保存了这些程序标准的汇编程序代码，以便每次改变 lcc 时与新代码做比较。reuse 第一次消减只消除

带有类型后缀I的空闲公共子表达式，在MIPS测试平台上它减少了58条指令。增加后缀C、S、D、F、B并不能减少任何指令，但是增加后缀P就可节约382条指令。

即使其中一个输入改变了，公共子表达式也不能重新计算。在允许一个额外匹配之后，标记程序调用mayrecalc进一步证实该公共子表达式可以被重新计算，并把结果记录在x.mayrecalc中：

```
⟨gen.c functions⟩+≡                                    299 300
  int mayrecalc(p) Node p; {
     Node q;

     ⟨mayrecalc 300⟩
  }
```

如果节点不表示公共子表达式，mayrecalc就会返回假：

```
⟨mayrecalc 300⟩≡                                       300 300
  if (!p->syms[RX]->u.t.cse)
     return 0;
```

如果森林中靠前的任何一棵树改变了公共子表达式的输入，mayrecalc也返回假：

```
⟨mayrecalc 300⟩+≡                                      300 300 300
  for (q = head; q && q->x.listed; q = q->link)
     if (generic(q->op) == ASGN
     && trashes(q->kids[0], p->syms[RX]->u.t.cse))
        return 0;
```

如果两个条件都不满足，那么公共子表达式就能安全地被重新计算了：

```
⟨mayrecalc 300⟩+≡                                      300 300
  p->x.mayrecalc = 1;
  return 1;
```

trashes(p, q)遍历公共子表达式q，并报告赋值的目标p是否在q中被读取：

```
⟨gen.c functions⟩+≡                                    300 300
  static int trashes(p, q) Node p, q; {
     if (!q)
        return 0;
     else if (p->op == q->op && p->syms[0] == q->syms[0])
        return 1;
     else
        return trashes(p, q->kids[0])
            || trashes(p, q->kids[1]);
  }
```

reduce及其辅助程序执行完毕后，gen调用prune。它使用x.inst标记为x.kids域中的最新指令构造一棵树。该者运行寄存器分配程序，只有指令需要寄存器，其余节点不需要寄存器（例如，ADDP节点由寻址模式自动计算），所以lcc不把这些节点放在寄存器分配程序可见的树中。初始的树仍保存在kids域中。调用prune之后执行一个化简程序，但prune本身并不是化简程序：

```
⟨gen.c functions⟩+≡                                    300 302
  static Node *prune(p, pp) Node p, pp[]; {
     ⟨prune 301⟩
  }
```

pp 指向某个节点的 x.kids 向量中的一个元素，也就是下一个即将被填充的元素。p 指向将被剪枝的树。如果 p 表示一条指令，那么 prune 就把指令存储到 *pp 中，并返回 pp+1，即指向下一个空单元。否则，prune 什么都不存，返回 pp，也不向前走。

如果树 p 是空的，prune 的工作如下：

```
⟨prune 301⟩≡                                                    301  300
  if (p == NULL)
    return pp;
```

否则，prune 清除节点的 x.kids 域中的无用元素：

```
⟨prune 301⟩+≡                                                301  301  300
  p->x.kids[0] = p->x.kids[1] = p->x.kids[2] = NULL;
```

如果 p 不是一条指令，prune 将会在子树中寻找指令，从第一个子节点开始：

```
⟨prune 301⟩+≡                                                301  301  300
  if (p->x.inst == 0)
    return prune(p->kids[1], prune(p->kids[0], pp));
```

每次递归调用都能存储 0 条或多条指令。上面的嵌套调用确保 prune 返回 pp 上的累积结果。

如果 p 是一个设置临时变量的指令，而且临时变量的 x.usecount 小于 2，那么临时变量将被该指令置值，但这个值不会再使用，因此可以忽略该指令，按上述方式继续遍历树：

```
⟨prune 301⟩+≡                                                301  301  300
  else if (p->syms[RX] && p->syms[RX]->temporary
  && p->syms[RX]->x.usecount < 2) {
    p->x.inst = 0;
    return prune(p->kids[1], prune(p->kids[0], pp));
  }
```

记住，reduce 刚刚计算了 x.usecount 的值。

如果上述条件没有一个满足，p 就是一条必需的指令，prune 把它存储在 *pp 中，并返回下一个待设置的元素的地址。prune 还须去该节点的子树，并将其所有指令存储在 p 的 x.kids 域中，这是因为在这条指令之后的任何一条指令一定是 p 的子树，而不是 pp 指向的比 p 更高层的节点。

```
⟨prune 301⟩+≡                                                301  300
  else {
    prune(p->kids[1], prune(p->kids[0], &p->x.kids[0]));
    *pp = p;
    return pp + 1;
  }
```

prune 增加了 pp 的值，然后将另一个 p 存放到 pp 所指的单元中。这个过程不会执行过头，因为 x.kids 的长度足以处理目标指令能访问的寄存器的最大值，该值与任意指令（即任何代码）可拥有的子节点数相同。理论上 prune 遵循这种断言，但是如果要做检查，还需要增加至少一个供断言读取的参数。

图 14-5 中的虚线表示 prune 在图 14-3 中的树上增加的 x.kids，而 ASGNI、ADDI、CVCI 是指令，剩余节点是子指令，这在当前许多机器中是存在的：CVCI 取字节并扩展符号位，ADDI 加 4，ASGNI 存储结果。实线是 kids。

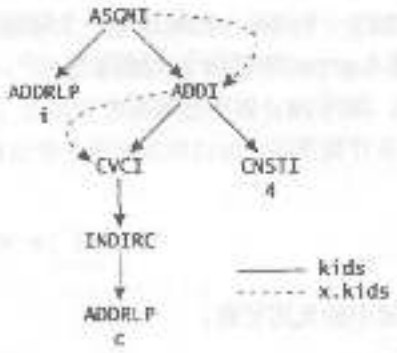


图 14-5 图 14-3 剪枝后

下面显示了对 prune 的调用过程，这些调用在虚线被创建时进行，p 为 0 时不进行调用，并且用节点的操作码来命名节点，从而减少了混乱：

```
prune(ASGNI, &dummy) called
 prune(ADDRLP, &ASGNI->x.kids[0]) called
 prune(ADDI, &ASGNI->x.kids[0]) called
  prune(CVCI, &ADDI->x.kids[0]) called
   prune(INDIRC, &ADDI->x.kids[0]) called
    prune(ADDRLP, &ADDI->x.kids[0]) called
  prune(CVCI, &ADDI->x.kids[0]) points the ADDI at the CVCI
  prune(CNSTI, &ADDI->x.kids[1]) called
 prune(ADDI, &ASGNI->x.kids[0]) points the ASGNI at the ADDI
prune(ASGNI, &dummy) points dummy at the ASGNI
```

gen 调用 prune 并提供一个哑元（dummy）单元来接收指向最高层指令的指针。因为根是作为副作用而被执行的，故仅需要一个哑元单元就足够了，而且根一定是一条指令，gen 不必检测 dummy 就可以知道根在那里。

14.4 代价函数

在 lburg 规范中大多数代价值都是常量，但也有一些代价值依赖于被匹配节点的性质。例如，把一个常量加到另一个操作数的指令，其代价值就取决于这个常量是否能够存储在指定位中。对于存储常量的节点 p 来说（节点 ADDRL 和 ADDRF 存有线偏移常量，节点 CNST 保存数字常量），range(p, lo, hi) 决定常量是否在整数 lo 和 hi 之间（包括 lo 和 hi）。如果在 lo 和 hi 之间，range 就返回一个零代价值，否则返回一个较大的代价值，促使树分析程序使用其他匹配。在一个 lburg 代价表达式中，a 表示匹配的节点，也就是计算代价表达式时传递给 _label 的参数，一个典型的应用是：

```
con8: CNSTI  "%a"  range(a, -128, 127)
```

上述规则匹配所有的 CNSTI 节点，但是如果常量不能放入 8 位有符号域中，那么代价值就不为 0。range 的实现如下：

```
⟨gen.c functions⟩+≡                                    300 303
  #define ck(i) return (i) ? 0 : LBURG_MAX
```

```
int range(p, lo, hi) Node p; int lo, hi; {
   Symbol s = p->syms[0];

   switch (p->op) {
   case ADDRFP: ck(s->x.offset >= lo && s->x.offset <= hi);
   case ADDRLP: ck(s->x.offset >= lo && s->x.offset <= hi);
   case CNSTC:  ck(s->u.c.v.sc >= lo && s->u.c.v.sc <= hi);
   case CNSTI:  ck(s->u.c.v.i  >= lo && s->u.c.v.i  <= hi);
   case CNSTS:  ck(s->u.c.v.ss >= lo && s->u.c.v.ss <= hi);
   case CNSTU:  ck(s->u.c.v.u  >= lo && s->u.c.v.u  <= hi);
   case CNSTP:  ck(s->u.c.v.p  == 0  && lo <= 0 && hi >= 0);
   }
   return LBURG_MAX;
}
```

对无符号字符常量来说，range 应该将 u.c.v.uc 的值进行零扩展，而不是对 u.c.v.sc 进行符号扩展。但是 range 的简化处理并不会有问题，因为节点 CNSTC 只出现在 ASGNC 的右边，ASGNC 忽略了扩展位。如果不进行简化处理，就需要将 CNSTC 按照有符号变量和无符号变量两种情况分别处理。无符号短常量的处理与上面一样。

14.5 调试

lburg 根据经过编码的输入规范，对树分析程序进行扩充。从严格意义上讲，这种扩充并非是必需的，但它有助于产生调试所需的信息。例如，向量 _opname 和 _arity 为每一个终结符分别保存了子节点的名字和编号：

```
⟨BURM signature 295⟩+≡                                    297 304
  static char *_opname[];
  static char  arity[];
```

_opname 和 _arity 都是按照终结符的整数操作码进行索引的。lcc 在函数 dumptree 中使用了这些数据，dumptree 打印操作符和所有子树，子树显示在圆括号中，并用逗号隔开：

```
⟨gen.c functions⟩+≡                                       302 304
  static void dumptree(p) Node p; {
     fprint(2, "%s(", IR->x._opname[p->op]);
     if (IR->x._arity[p->op] == 0 && p->syms[0])
        fprint(2, "%s", p->syms[0]->name);
     else if (IR->x._arity[p->op] == 1)
        dumptree(p->kids[0]);
     else if (IR->x._arity[p->op] == 2) {
        dumptree(p->kids[0]);
        fprint(2, ", ");
        dumptree(p->kids[1]);
     }
     fprint(2, ")");
  }
```

对叶节点来说，如果 p->syms[0] 存在，dumptree 就添加它。对于图 14-3 中的树，其输出如下：

```
ASGNI(ADDRLP(i), ADDI(CVCI(INDIRC(ADDRLP(c))), CNSTI(4)))
```

lcc 使用了 dumptree，因其意义不大，本书省略了相关调用。但是 dumptree 本身值得介绍一下，看看 lburg 是如何支持调试的。

向量 _string 保存了每个规则的文本。

```
(BURM signature 295)+≡                                    303 305
  static char *_string[];
```

它是按规则号进行索引的。化简程序 dumpcover 扩充了最小的化简程序，并利用 _string 逐层缩进来显示树覆盖。

```
(gen.c functions)+≡                                       303 305
  static void dumpcover(p, nt, in) Node p; int nt, in; {
     int rulenum, i;
     short *nts;
     Node kids[10];

     p = reuse(p, nt);
     rulenum = getrule(p, nt);
     nts = IR->x._nts[rulenum];
     fprint(2, "dumpcover(%x) = ", p);
     for (i = 0; i < in; i++)
        fprint(2, " ");
     dumprule(rulenum);
     (*IR->x._kids)(p, rulenum, kids);
     for (i = 0; nts[i]; i++)
        dumpcover(kids[i], nts[i], in+1);
  }

  static void dumprule(rulenum) int rulenum; {
     fprint(2, "%s / %s", IR->x._string[rulenum],
        IR->x._templates[rulenum]);
     if (!IR->x._isinstruction[rulenum])
        fprint(2, "\n");
  }
```

当编译图 14-3 的 MIPS 代码时，dumptree 显示如下：

```
dumpcover(1001e9b8) = stmt: ASGNI(addr, reg) / sw $%2,%1
dumpcover(1001e790) =  addr: ADDRLP / %a($sp)
dumpcover(1001e95c) =  reg: addr / la $%c,%1
dumpcover(1001e95c) =   addr: ADDI(reg, con) / %2($%1)
dumpcover(1001e8a4) =    reg: CVCI(INDIRC(addr)) / lb $%c,%1
dumpcover(1001e7ec) =     addr: ADDRLP / %a($sp)
dumpcover(1001e900) =    con: CNSTI / %a
```

下一节将解释 x._templates 和每个规则后的汇编程序模板。

14.6 发送器

lcc 发送器（emitter）的作用是为目标机器输出汇编程序代码。发送器不依赖于目标机器，由两个描述与机器相关数据的数组驱动。lburg 为每个 BURM 生成一些 C 程序代码，用来声明并初始化这两个数组。两个数组都是通过规则号来索引的，其中一个数组为规则生成模板：

```
⟨BURM signature 295⟩+≡                                    304 305
  static char *_template[];
```

另一个数组标记与指令对应的模板，以区别于子指令（如寻址方式）：

```
⟨BURM signature 295⟩+≡                                    305 316
  static char _isinstruction[];
```

lburg 从 1 开始为规则编号，并通过返回规则号来报告匹配情况，这样当需要的时候就可以找到相应的模板。如果模板以一个换行字符结尾，那么 lburg 就假设它是一条指令，否则就必然是某条指令的一部分，比如是一个操作数。

emitasm 对规则结构及其汇编程序代码模板进行了解释：

```
⟨gen.c functions⟩+≡                                       304 307
  static unsigned emitasm(p, nt) Node p; int nt; {
      int rulenum;
      short *nts;
      char *fmt;
      Node kids[10];

      p = reuse(p, nt);
      rulenum = getrule(p, nt);
      nts = IR->x._nts[rulenum];
      fmt = IR->x._templates[rulenum];
      ⟨emitasm 305⟩
      return 0;
  }
```

emitasm 是另一个化简程序，它处理部分线性化的树。列表的元素是指令子树的根。emitasm 递归调用自身以处理地址计算之类的子指令。emitasm 的遍历从一个指令开始。当递归到达为该指令提供值的指令时结束。也就是说，emitasm 的化简是追踪指令内的树分析，这个分析与寻址方式及指令内的其他计算相对应。emitasm 由 emit 调用，emit 确保 emitasm 以正确的顺序来处理这些指令，这样便可处理指令间的顺序。

当产生节点的代码时，emit 设置 x.emitted 来标记已生成的节点。当 emitasm 遇到已生成的指令时，仅生成存放有该指令结果的寄存器名。对所有产生值的节点来说，寄存器分配程序已经在 p->syms[RX] 中记录了目标寄存器。

```
⟨emitasm 305⟩≡                                            305 305
  if (IR->x._isinstruction[rulenum] && p->x.emitted)
      outs(p->syms[RX]->x.name);
```

如果模式以 # 开头，生成器就调用 emit2，它是一个与目标机器相关的过程：

```
⟨emitasm 305⟩+≡                                           305 305 305
  else if (*fmt == '#')
      (*IR->x.emit2)(p);
```

lcc 需要这种转义符为诸如结构参数之类的复杂情况生成各种代码，否则，emitasm 就需要一些解释来产生模板：

```
⟨emitasm 305⟩+≡                                           305 305
  else {
      ⟨omit leading register copy? 306⟩
```

```
    for ((*IR->x._kids)(p, rulenum, kids); *fmt; fmt++)
        if (*fmt != '%')
            *bp++ = *fmt;
        else if (*++fmt == 'F')
            print("%d", framesize);
        else if (*fmt >= '0' && *fmt <= '9')
            emitasm(kids[*fmt - '0'], nts[*fmt - '0']);
        else if (*fmt >= 'a' && *fmt < 'a' + NELEMS(p->syms))
            outs(p->syms[*fmt - 'a']->x.name);
        else
            *bp++ = *fmt;
}
```

bp 是指向 output.c 模块中输出缓冲区的指针。%F 使得 emitasm 生成 framesize，这样有助于生成相对于帧大小的局部偏移量。%digit 形式的子串使得 emitasm 递归生成与模板的第 digit 个非终结符相对应的子树，子树从 0 开始计数，从左到右，并忽略嵌套。%x 形式的子串让 emitasm 生成节点的 p->syms['x' -'a']->x.name。例如，%c 生成 p->syms[2] ->x.name。表 14-1 总结了这些约定。

表 14-1 产生器模板语法

模板	产生内容
%%	一个 %
%F	framesize
%digit	对应规则的第 digit 个非终结符的子树
%letter	p->syms[letter-'a']->x.name
任何其他字符	字符本身
以 # 打头	调用 emit2 以产生代码
以? 打头	如果源和目标寄存器相同，忽略第一条指令

所以发送器是这样来解释字符串 "lw r%c, %1\n" 的：先生成 "lw r"，然后是目标寄存器的名字（通常是一个数字串），接着是一个逗号。如果 nts[1] 保存了表示非终结符 addr 的整数，那么递归生成 p->kids[1] 作为一个 addr。最后，emitasm 生成一个换行符。

许多目标机器采用三操作数指令，这种指令有两个独立的源操作数并产生一个独立的目的操作数。其他目标机器通过使用两操作数指令以节省存储指令的空间，这种两操作数指令将目的操作数定为第一个源操作数。由于第一个源操作数可能还要使用，所以 lcc 把双指令模板用于诸如 ADDI 之类的操作码。第一条指令复制第一个源操作数到目的操作数，第二条指令把第二个源操作数加到目的操作数上。如果第一个源操作数不再有用，寄存器分配程序通常安排目的操作数和第一个源操作数共享相同的寄存器，所以第一条指令复制寄存器到其自身是多余的。生成器最后可以很容易地删除这些冗余指令。每个规范在这类指令的开头标记一个 "?"，如果源操作数寄存器和目的操作数寄存器是同一个，emit 将跳过这类指令。

```
⟨omit leading register copy? 306⟩=                                    305
  if (*fmt == '?') {
    fmt++;
    if (p->syms[RX] == p->kids[0]->syms[RX])
      while (*fmt++ != '\n')
        ;
  }
```

接口程序 emit 遍历了指令列表，并且每次生成一条指令：

```
⟨gen.c functions⟩+≡                                    305 307
  void emit(p) Node p; {
     for (; p; p = p->x.next) {
        if (p->x.equatable && requate(p) || moveself(p))
           ;
        else
         (*emitter)(p, p->x.inst);
       p->x.emitted = 1;
     }
  }
```

大多数接口例程对每个目标机器都有一个对应的实现，但是 emit 的实现只有一个，emit 与目标相关的部分已被分离出来，成为汇编代码模式。

上面的间接调用使得 lcc 可以调用另一个生成器。例如，利用这一特性，可以把本书介绍的生成器替换成一个直接产生二进制目标代码的生成器。emitter 被初始化成 emitasm：

```
⟨gen.c data⟩+≡                                         290 310
  unsigned (*emitter) ARGS((Node, int)) = emitasm;
```

emit 实现了两种最终优化措施。moveself 删除了那些复制寄存器到自身的指令：

```
⟨gen.c functions⟩+≡                                    307 307
  static int moveself(p) Node p; {
     return p->x.copy
     && p->syms[RX]->x.name == p->x.kids[0]->syms[RX]->x.name;
  }
```

相等测试利用了这样一个事实，就是字符串功能模块对于每个不同的字符串只保存一个副本。x.copy 由代价函数 move 来设置，该函数通过选择“寄存器—寄存器”移动的规则调用：

```
⟨gen.c functions⟩+≡                                    307 307
  int move(p) Node p; {
     p->x.copy = 1;
     return 1;
  }
```

emit 的其他优化措施消除了一些寄存器—寄存器复制，比如通过使用源寄存器来代替目的寄存器。如果指令 p 把寄存器 src 复制到临时的寄存器 tmp，作为公共子表达式来使用，那么寄存器分配程序将设置 x.equatable 标记。如果设置了 x.equatable 标记，生成器就调用 requate，requate 从 p 开始向前扫描：

```
⟨gen.c functions⟩+≡                                    307 310
  static int requate(q) Node q; {
     Symbol src = q->x.kids[0]->syms[RX];
     Symbol tmp = q->syms[RX];
     Node p;
     int n = 0;
     for (p = q->x.next; p; p = p->x.next)
        (requate 308)
     for (p = q->x.next; p; p = p->x.next)
        if (p->syms[RX] == tmp && readsreg(p)) {
           p->syms[RX] = src;
           if (--n <= 0)
```

```
            break;
        }
    return 1;
}
```

第一个 for 循环包含了几个返回 0 的语句，它们使生成器继续生成指令，除非 moveself 介入。生成器只在 requate 退出第一个循环的时候删除寄存器－寄存器复制指令，然后进入第二个循环，最后返回值为 1。第二个循环将所有从 tmp 读数替换成从 src 中读数，第一个循环已经将这些读操作的数目记录在 n 中。

如果指令把 tmp 复制到 src，由于 tmp 的值改变了，所以 moveself 会将其删除，循环将继续检查，看是否可能还有其他的改变：

```
⟨requate 308⟩≡                                                    308  307
  if (p->x.copy && p->syms[RX] == src
  &&  p->x.kids[0]->syms[RX] == tmp)
      p->syms[RX] = tmp;
```

如果没有这个测试，return f() 将从返回寄存器把 f 的值复制到一个临时单元，然后再为当前函数把该值复制到返回寄存器。

如果找到一条指令是以 src 为目标的，且该指令不是将 src 赋值给其自身，指令也不只是读取 src，那么 requate 将会失效，因为一般来说，tmp 和 src 此后的值是不相同的：

```
⟨gen.c macros⟩≡                                                    322
  #define readsreg(p) \
      (generic((p)->op)==INDIR && (p)->kids[0]->op==VREG+P)
  #define setsrc(d) ((d) && (d)->x.regnode && \
      (d)->x.regnode->set == src->x.regnode->set && \
      (d)->x.regnode->mask&src->x.regnode->mask)

⟨requate 308⟩+≡                                                308  308  307
  else if (setsrc(p->syms[RX]) && !moveself(p) && !readsreg(p))
      return 0;
```

例如，当 p 在寄存器 r1 中时，c=*p++ 产生了下面的伪指令。目的操作数是最右边的操作数。

```
move r1,r2      存储 p 值
add r2,1,r1     递增 p 值
loadb (r2),r3   取字符
storeb r3,c     存字符
```

requate 将 add 指令改成使用 r1 而不是 r2，但是不能对后面的指令做这种改变，因为在加法之后 r1 和 r2 就不相等了。

如果 requate 遇到一条使 tmp 溢出的指令，它也会结束：

```
⟨requate 308⟩+≡                                                308  309  307
  else if (generic(p->op) == ASGN && p->kids[0]->op == ADDRLP
  && p->kids[0]->syms[0]->temporary
  && p->kids[1]->syms[RX]->x.name == tmp->x.name)
      return 0;
```

程序没有对 genspill 插入的节点给出明确的标志，但是通过上述条件能找到它们。

如果 requate 遇到调用指令，它也将停止，除非该调用是森林的结尾，因为 src 可能是调用程序存储的寄存器，它调用了 clobber。

```
(requate308)+≡                                          308 309  307
  else if (generic(p->op) == CALL && p->x.next)
    return 0;
```

通常，src 是个被调用程序存储的寄存器变量，所以 requate 可能在终止之前需要确认寄存器是调用程序存储的。但对于数以千行计的源代码而言，这种检查毫无意义，因此程序略去了这种检查。

如果 requate 遇到标号，它也将停止，除非该标号是森林的结尾，因为 src 此后可能会有不同的值：

```
(requate308)+≡                                          309 309  307
  else if (p->op == LABEL+V && p->x.next)
    return 0;
```

如果上述测试都没有成功，且 tmp 和 src 有同样的值，那么倘若这个节点读取了 tmp，该指令就被计数，并继续循环以判断后面对 tmp 的使用能否被 src 取代：

```
(requate308)+≡                                          309 309  307
  else if (p->syms[RX] == tmp && readsreg(p))
    n++;
```

如果一个节点写入 tmp，或者如果 requate 处理完了指令，那么森林关于 tmp 的处理完毕，requate 退出第一个循环：

```
(requate308)+≡                                          309  307
  else if (p->syms[RX] == tmp)
    break;
```

现在 requate 的第二个循环用 src 的读操作取代了 tmp 的所有读操作，然后 requate 返回 1，生成器忽略最初对 tmp 的赋值。

到目前为止，大多数复杂的寄存器－寄存器复制操作，一般来自于那些在使用初值的情形中的后增指令，如 c = *p++。在某种情形下，可以通过重新安排指令来避免这些代码，对于这类模式，lcc 的代码以一个复制指令开始。例如，一个比较好的优化程序能够将上面的 4 条扔指令缩减为：

```
loadb (r1),r3    取字符
add r1,1,r1      递增 p
storeb r3,c      存字符
```

现在，在标准 lcc 测试平台中，寄存器－寄存器传送指令大概占到 MIPS 和 SPARC 所有指令的 5%。在 MIPS 代码中，大约有一半的复制是从寄存器变量或零（这种寄存器－寄存器复制的源寄存器由硬件置零）到寄存器变量、参数或返回寄存器，这样的传送指令是不容易被删除的。剩下的一些（但不是全部）可能被删除，这样，我们就接近简单寄存器复制优化的极限了。

14.7 寄存器定位

有些节点能够在众多寄存器中任选一个进行计算，但是有些节点就要烦琐一些。例如，大多数计算机能够在任何通用寄存器中计算整数之和，但是大多数函数调用都规定返回值只能存储在一个特定的寄存器中。

如果节点需要子节点在一个固定寄存器中，那么寄存器定位器序就会尽力将该子节点的计算放在那个寄存器中。如果不能做到，那么代码生成器必须在树中的父节点与子节点之间插入一个寄存器－寄存器复制指令。例如，在下面的代码中：

```
f(a, b) { return a + b; }
```

返回操作是烦琐的，但求和并不复杂，所以代码能够直接在返回寄存器中求和。相反，

```
f() { register int i = g(); }
```

g 通常通过某个寄存器返回结果，而寄存器变量 i 会分配在另一个寄存器中，所以寄存器－寄存器复制指令是必不可少的。

下一章将介绍变量和临时变量的实际的寄存器分配，但是寄存器－寄存器复制是一种指令，这种指令只有在表示为节点时，才能像其他指令一样得到处理。为此，prelabel 在标记树前先对树进行一次遍历：

```
⟨gen.c functions⟩+≡                                    ↑307 311↓
  static void prelabel(p) Node p; {
    ⟨prelabel 310⟩
  }
```

prelabel 对那些烦琐的节点标以它们需要的寄存器，对其余的节点（至少是那些产生结果而不是起副作用的节点）标以表示有效寄存器组的 wildcard 符号。prelabel 还在那些可能要用到寄存器－寄存器复制的地方插入 LOAD 节点。

preload 以从左到右遍历子树的方式开始：

```
⟨prelabel 310⟩≡                                        310↓ 310
  if (p == NULL)
    return;
  prelabel(p->kids[0]);
  prelabel(p->kids[1]);
```

然后就为需要在寄存器中存放结果的节点指明寄存器类别：

```
⟨prelabel 310⟩+≡                                    ↑310 311↓ 310
  if (NeedsReg[opindex(p->op)])
    setreg(p, rmap[optype(p->op)]);
```

NeedsReg 测试把具有副作用的节点与那些需要寄存器存放结果的节点区分开来。NeedsReg 按照通用操作码索引，并标记产生值的操作码：

```
⟨gen.c data⟩+≡                                         ↑307 313↓
  static char NeedsReg[] = {
    0,                      /* unused */
    1,                      /* CNST */
    0, 0,                   /* ARG ASGN */
    1,                      /* INDIR  */
    1, 1, 1, 1,             /* CVC CVD CVF CVI */
    1, 1, 1, 1,             /* CVP CVS CVU NEG */
    1,                      /* CALL */
    1,                      /* LOAD */
    0,                      /* RET */
```

```
    1, 1, 1,                    /* ADDRG ADDRF ADDRL */
    1, 1, 1, 1, 1,              /* ADD SUB LSH MOD RSH */
    1, 1, 1, 1,                 /* BAND BCOM BOR BXOR */
    1, 1,                       /* DIV MUL */
    0, 0, 0, 0, 0, 0,           /* EQ GE GT LE LT NE */
    0, 0,                       /* JUMP LABEL   */
};
Symbol rmap[16];
```

rmap 按照类型后缀索引，保存了表示存放该类型值的寄存器组的 wildcard。例如，rmap[I] 中存放了表示通用寄存器的通配符，rmap[D] 中存放了表示双精度浮点寄存器的通配符。每个寄存器组都与目标机器相关，所以目标机器的 progbeg 初始化 rmap。setreg 在节点中记录了来自 rmap 的值，以支持寄存器定位和分配：

```
⟨gen.c functions⟩+≡                                    310 312
  void setreg(p, r) Node p; Symbol r; {
     p->syms[RX] = r;
  }
```

由于 setreg 对于前面的断言和断点作用很大，所以将其设计为一个函数。

prelabel 调用 setreg，为所有带有相同类型后缀的操作码指派相同的通配符。接下来，prelabel 校正了烦琐节点。

寄存器变量能够影响目标寄存器定位，所以 prelabel 下一步要找出读写寄存器变量的节点。前端的符号可以区分寄存器变量与非寄存器变量（判断符号的 sclass 域是否是 REGISTER），但是前端节点并不能做到这一点。后端必须生成不同的代码来访问这两种存储类别，所以 prelabel 改变了一些访问寄存器变量的操作码。如果引用的符号是个寄存器变量，就用 VREG 来代替 ADDRL 和 ADDRF，并且利用分配给该变量的唯一寄存器代替在 VREG 上面的 INDIR 中的通配符：

```
⟨prelabel 310⟩+≡                                       310 312  310
  switch (generic(p->op)) {
  case ADDRF: case ADDRL:
     if (p->syms[0]->sclass == REGISTER)
        p->op = VREG+P;
     break;
  case INDIR:
     if (p->kids[0]->op == VREG+P)
        setreg(p, p->kids[0]->syms[0]);
     break;
  case ASGN:
     ⟨prelabel case for ASGN 311⟩
     break;
  }
```

prelabel 将赋值指令的右子节点分配给寄存器变量，尽可能计算其值并直接存入该寄存器变量：

```
⟨prelabel case for ASGN 311⟩=                                  311
  if (p->kids[0]->op == VREG+P) {
   rtarget(p, 1, p->kids[0]->syms[0]);
  }
```

最后，prelabel 调用一个与目标机器相关的程序，该程序为烦琐的操作码调整寄存器类别：

```
(prelabel310)+≡                                          311  310
  (IR->x.target)(p);
```

rtarget(p,n,r) 保证了在该计算 p->kids[n] 的结果并保存到寄存器 r 中：

```
⟨gen.c functions⟩+≡                                      311  313
  void rtarget(p, n, r) Node p; int n; Symbol r; {
     Node q = p->kids[n];

     if (r != q->syms[RX] && !q->syms[RX]->x.wildcard) {
        q = newnode(LOAD + optype(q->op),
           q, NULL, q->syms[0]);
        if (r->u.t.cse == p->kids[n])
           r->u.t.cse = q;
        p->kids[n] = p->x.kids[n] = q;
        q->x.kids[0] = q->kids[0];
     }
     setreg(q, r);
  }
```

如果子节点已被定位到另一个寄存器变量或者返回寄存器之类的特殊寄存器，那么 rtarget 在树中父节点和子节点之间插入一个 LOAD 节点，并定位 LOAD 节点（而不是原来的子节点）。代码生成器为 LOAD 生成寄存器－寄存器复制代码。如果子节点未被定位，那么 q->syms[RX] 含有一个通配符。因为 r 必须是通配符对应的寄存器组中的一员，最后调用 setreg 来保证这一点。如果不是，lcc 将产生代码，把一个寄存器组中的寄存器复制到另一个寄存器组中的寄存器，假若没有显式转类节点，这种情况便不会发生。

图 14-6 显示了在 rtarget 之前和之后的 3 棵示例树。它们均假定 r0 是返回寄存器，r2 是寄存器变量。第一棵树有一个无约束的子节点，所以 rtarget 没有插入 LOAD1；第二棵树有一个 INDIRI，INDIRI 在 RETI 之下，生成 r2，但 RETI 需要的是 r0，所以 rtarget 插入了一个 LOADI；第三棵树有一个 CALLI，CALLI 在 ASGNI 之下，产生 r0，但 ANGNI 需要 r2，所以 rtarget 也插入了一个 LOADI。

prelabel 和 rtarget 通过寄存器定位程序取出和存入寄存器变量，所以 lcc 针对这类操作的模板在任何机器上都不为这两种操作生成代码。所有的机器都使用如下规则：

```
⟨shared rules 312⟩≡                                      314  335 363 389
  reg:  INDIRC(VREGP)      "# read register\n"
  reg:  INDIRD(VREGP)      "# read register\n"
  reg:  INDIRF(VREGP)      "# read register\n"
  reg:  INDIRI(VREGP)      "# read register\n"
  reg:  INDIRP(VREGP)      "# read register\n"
  reg:  INDIRS(VREGP)      "# read register\n"
  stmt: ASGNC(VREGP,reg)   "# write register\n"
  stmt: ASGND(VREGP,reg)   "# write register\n"
  stmt: ASGNF(VREGP,reg)   "# write register\n"
  stmt: ASGNI(VREGP,reg)   "# write register\n"
  stmt: ASGNP(VREGP,reg)   "# write register\n"
  stmt: ASGNS(VREGP,reg)   "# write register\n"
```

注释模板不生成任何代码，但是它出现在调试器的输出中，所以描述性注释还是有作用的。

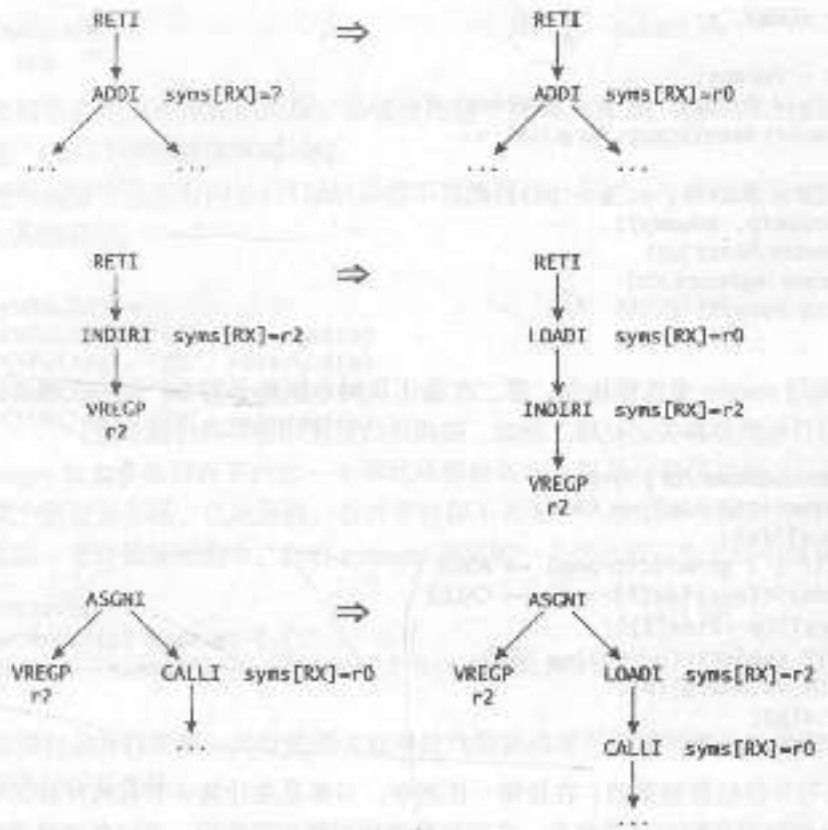


图 14-6 target 示例

14.8 指令选择的协调

13.1 节介绍了 rewrite 和 gen 怎样协调处理本章所描述的某些过程，下面做进一步的详细介绍。rewrite 完成了单棵树的寄存器定位和指令选择：

```
⟨gen.c functions⟩+≡                                    312 313
  static void rewrite(p) Node p; {
    prelabel(p);
    (*IR->x._label)(p);
    reduce(p, 1);
  }
```

接口函数 gen 从前端接收森林，然后对这些树进行多次遍历。

```
⟨gen.c data⟩+≡                                         310 319
  Node head;

⟨gen.c functions⟩+≡                                    313 315
  Node gen(forest) Node forest; {
    int i;
    struct node sentinel;
```

```
    Node dummy, p;

    head = forest;
    for (p = forest; p; p = p->link) {
        ⟨select instructions for p 314⟩
    }
    for (p = forest; p; p = p->link)
        prune(p, &dummy);
    ⟨linearize forest 323⟩
    ⟨allocate registers 323⟩
    return forest;
}
```

第一次遍历调用 rewrite 来选择指令，第二次遍历从树中删除了指令。第一次遍历为参数和过程调用完成了与目标机器相关的处理。例如，按调用约定使用寄存器传递参数：

```
⟨select instructions for p 314⟩=                                    314
    if (generic(p->op) == CALL)
        docall(p);
    else if (   generic(p->op) == ASGN
    && generic(p->kids[1]->op) == CALL)
        docall(p->kids[1]);
    else if (generic(p->op) == ARG)
        (*IR->x.doarg)(p);
    rewrite(p);
    p->x.listed = 1;
```

只有 doarg 是与目标机器相关的。在任何一棵树中，只要是先计算子节点再计算父节点，代码生成器就能自由地以最佳顺序计算节点。有些函数调用可能有副作用，所以前端将调整所有对该森林的调用，来修改产生副作用的调用的顺序。如果调用没有返回值或者忽略了返回值，那么调用本身就出现在森林中；第一个 if 语句用来识别这种模式。否则，调用指令将出现在对一个临时变量赋值的指令下面，该临时变量用于以后需要返回值的地方；第二个 if 语句用来识别这种模式。

第一次遍历还对列出的节点进行标记。第 15 章将详细介绍这一点以及 gen 遍历的其他处理。

14.9 共享规则

本书中有些规则对所有目标机器都是相同的。这些规则被分离出来放在一个与目标机器无关的片段中，既可以节省空间，又可以保证 lcc 发生变化时保持一致。其中一些公共规则与些数常量匹配：

```
⟨shared rules 312⟩+=                                    ↑312 ↓15  335 363 389
    con: CNSTC  "%a"
    con: CNSTI  "%a"
    con: CNSTP  "%a"
    con: CNSTS  "%a"
    con: CNSTU  "%a"
```

本书中，所有 Iburg 规范共享这样的约定，该约定包括：非终结符 reg 匹配所有在寄存器中产生结果的计算；非终结符 stmt 匹配所有的根，这是为某些副作用而执行的，主要是改变存储器或程序计数器。

```
⟨shared rules 312⟩+≡                                    314 315  335 363 389
  stmt: reg  ""
```

当产生寄存器的节点作为根出现的时候，必须使用这个规则。例如，如果 CALLI 的调用者忽略了它的返回值，CALLI 就需要该规则匹配。

下面的规则说明当前所有的 lcc 目标机器都不需要通过计算把一个整型或指针类型转换为具有同样大小的其他类型：

```
⟨shared rules 312⟩+≡                                    315  335 363 389
  reg: CVIU(reg)  "%0"  notarget(a)
  reg: CVPU(reg)  "%0"  notarget(a)
  reg: CVUI(reg)  "%0"  notarget(a)
  reg: CVUP(reg)  "%0"  notarget(a)
```

代价函数 notarget 在大多数情况下产生一个零代价值的匹配，但是如果寄存器定位程序已经将节点安排到了某个固定寄存器，也就是说，目的寄存器不再是一个表示寄存器红的通配符，那么就可能需要寄存器－寄存器复制指令，此时 notarget 将返回一个代价值以中止这个规则：

```
⟨gen.c functions⟩+≡                                     313 320
  int notarget(p) Node p; {
    return p->syms[RX]->x.wildcard ? 0 : LBURG_MAX;
  }
```

每个规范都包含并行规则，并行规则生成单位代价值的寄存器复制指令，这种规则在节点有固定目标寄存器的时候使用。

14.10 编写规范

第 16 章至第 18 章给出了一些完整的 lburg 输入。尽管编写自己的 lburg 规范最容易的方法可能是以本书解释的某个规范为基础开始编写，但是了解一些通用的准则还是有用的。16.2 节就说明了这一点。

一般来说，每条指令对应一条规则，每个寻址方式也对应一条规则。在规范中，模板给出了汇编程序语法，模式使用中间语言操作符树描述指令的作用。

通常遇到等价操作符时需要复制规则。例如，针对 ADDI 的规则一般与针对 ADDU 和 ADDP 的规则类似。

为每个树操作符编写额外规则时，可以将树操作符作为通用操作符的特例来处理。例如，ADDRLP 可以通过给帧指针添加常量来实现，所以在写一个匹配常量与寄存器求和的规则时，稍做修改就可以编写与 ADDRLP 匹配的规则。

要为通用操作符的每种退化情形编写额外的规则。例如，如果一些寻址方式与常量和寄存器之和匹配，那么当常量为 0 的时候，它也能做一些简单的间接寻址。

要为中间语言中不能由单个指令实现的操作符编写额外的规则生成多条指令。例如，许多机器没有直接实现 CVCI 的指令，所以它们的规范中就实现了一个规则，该规则的模板有两条移位指令。这些指令通过对字节先进行逻辑或者算术左移再算术右移的方式来传播符号位。

应当用一个非终结符来推导所有能产生值的树。当对应于规则模式的指令需要读取寄存器时使用该非终结符。本书就是按照这种方法使用非终结符 reg 的。这种方法的一种变形能够发现更多的错误。即一个非终结符用于通用寄存器，另一个非终结符用于浮点寄存器（如 freg）。例如，

只使用一个寄存器非终结符的规则会接受类似 NEGI (INDIRI (...)) 在语法上有错误的树。当然，这种特殊的错误是很少见的。

类似地，可用一个非终结符来推导所有只起副作用的树，例如 ASGN 和 ARG。本书用 stmt 表示副作用树。我们可以编写一个 Iburg 规范，合并 reg 和 stmt，但是寄存器分配程序假定起副作用的树都是根，而具有值的树都是内部节点和叶节点。如果违反了这种假定，Iburg 就会生成错误的代码，这是编译器编写人员最不愿见到的。如果违反这些假定，只能将 reg 从 stmt 中分离出来以保证代码生成器正确实现目标。

确保为中间语言中的每个操作符提供至少一种生成代码的方法。其中较容易的一种方法就是为每个操作符写一个寄存器－寄存器规则：

```
reg:  LEAF
reg:  UNARY(reg)
reg:  OPERATOR(reg,reg)
```

这样的规则确保了 lcc 对每个节点至少有一种匹配方法，为每个节点生成一条汇编指令代码。

编写 Iburg 规范时需要查阅目标机器的体系结构手册，并清楚支持各种中间代码操作的指令或寻址方式，编写各种规则，这些规则的模式能够匹配指令的计算功能。图 14-2 中的规则 3 和规则 7 就是这样的例子。如果看一下完整的“寄存器－寄存器”规则组，将会发现这些较大的规则不是必要的，但一般说来，它们生成的代码更短、更快。跳转指令和寻址方式是非常特殊的，很难想象它们会用于 C 程序（或 C 语言编译器）。

使用非终结符来对规范进行组织。如果你发现某个子模式经常被使用，那么为其设计一个规则，并用一个非终结符作为规则名字。

深入阅读

lcc 的指令选择程序基于 Aho and Johnson（1976）提出的算法。接口是根据 burg（Fraser, Henry and Proebsting, 1992）改编而成的，其实现也是根据相应的程序 iburg（Fraser, Hanson and Proebsting, 1992）改编的。iburg 在编译时采用了动态规划方法。处理规范时，burg 使用 BURS 理论（Pelegri-Llopart and Graham, 1988；Proebsting, 1992）来完成动态规划，所以速度比较快，但在某种程度上少了一些灵活性。

可变目标的编译器 gcc（Stallman, 1992）给出指令选择的另一种方法。它使用了一个本地代码生成器和一个彻底可变目标的窥孔优化程序，该程序由目标机器描述来驱动。Davidson 和 Fraser（1984）描述了这种方法的基础。

练习

14.1 如果把代价的类型从 short 改为 int，将会带来哪些问题？

14.2 严格来说，kids 并不是必需的。描述一下如果没有它将怎样实现伪码程序。

14.3 Iburg 使用从 1 开始的整数编码表示每个非终结符，1 表示开始非终结符。以 0 结尾的向量 _ntname 按照这些数进行索引，并存放相应非终结符的名字：

```
⟨BURM signature 295⟩+≡                                    305
  static char *_ntname[];
```

利用它编写过程 void dumpmatches（Node p），该过程显示节点 p 和所有与它匹配的规则，典型的输出可能是：

```
dumpmatches(0x1001e790)=ADDRLP(i):
   addr: ADDRLP / %a($sp)
   rc: reg / $%1
   reg: addr / la $%c,%1
```

dumpmatches 不是化简程序。

14.4 树分析程序在处理无环有向图（dag）时会出错。那么标记程序遍历 dag 时会出错吗？为什么？化简程序也会出错吗？为什么？

14.5 在任意的机器上，用 -d 或 -Wf, -d 选项编译代码：

```
f(i) { return (i-22)>>22; }
```

dumpcover 中的程序行可以识别自身。哪些行与 reuse 的额外匹配对应？哪些行与随后出现的额外匹配启动的匹配对应？

14.6 在你的机器上编译 lcc 时，moveself 优化节省了多少条指令？

14.7 优化的一个标准是对编译器本身进行编译时是否有效。例如，requate 优化需要花费时间，你能计算出在你的机器上需要多少时间吗？如果在 lcc 的编译自身时采用这种优化，就能生成一个更快速的编译器，这样可以节省时间。考察这种改进在你的机器上的效果，是否值得进行 requate 优化呢？

寄存器分配

寄存器分配包括两个部分：一是分配，决定哪些值占用寄存器；二是指派，为每个值指派某个特定寄存器。第 14 章的指令选择确定某个子表达式需要占用寄存器，因而隐含地分配了临时变量或中间值。本章主要描述寄存器变量的分配以及所有对寄存器的指派。

与所有的寄存器分配器一样，lcc 的寄存器分配器也包含多个任务：它必须跟踪哪些寄存器空闲、哪些寄存器正忙；必须在变量或中间值生存期开始时就为其分配一个寄存器，而且在其生存期结束时释放该寄存器以备它用；最后，当寄存器分配器用完所有寄存器时，必须生成代码将其中某些寄存器的值保存到存储器中（称为寄存器的溢出），使这些寄存器可以分配给其他变量或中间值，以后需要此值时再重新读取到寄存器中。

lcc 提供寄存器变量，即使没有明确声明，也可以为某些变量指派寄存器，但这是全局寄存器分配的一个扩展。lcc 没有过程间的寄存器分配，临时变量的分配只局限在森林内。

现在有许多更复杂的寄存器分配器，但是 lcc 的分配器产生的代码令人满意，与现在广泛使用的其他编译器产生的代码相比，也具有竞争力。lcc 的溢出器尤其简单。在典型的编译中溢出是非常少的，因此我们把更多的精力投入其他方面可能更有效。功能强大的寄存器分配器能够在寄存器中更长时间地保存更多的值，这样将会增加对寄存器的需求，因而也会增加溢出的数量。lcc 的寄存器分配器非常简单，所以它的溢出器也是很简单的。

设计 lcc 的寄存器分配器时首先要考虑的问题是，要有足够的灵活性来适应现有的各种寄存器使用约定，以便 lcc 的代码能与公共的已有的 ANSI C 库一起工作。也就是说，我们要做到不必编写、维护或编译 lcc 特有的库。

设计上第二个要考虑的问题是总体上的简单化，特别是使得与目标机器相关的代码最小化。这两个目标是互相冲突的。例如，lcc 的溢出器独立于目标机器，因此必须间接构造用于溢出、重取值的指令。也就是说，要创建一个中间代码树并通过代码生成器来遍历它。这将是比较麻烦的，因为我们正处在代码生成过程中，且寄存器已经用完。与目标机器相关的溢出器更简单，因为它可以简单地产生目标指令完成溢出和重取寄存器，但是我们不得不为每个目标机器编写并调试新的溢出器。即便使用 lcc 这样简单的溢出器，溢出也很少发生，这意味着好的用于测试溢出器的例子是很复杂的，也很难找到，因此溢出器很难调试。尽管与目标机器相关的溢出器比 lcc 的溢出器更简单，但是从长远的角度来看，这种方法的益处不大。

15.1 组织结构

表 15-1 显示了寄存器分配调用过程中最重要的部分，以说明寄存器分配器的整体组织结构。表中的缩进说明了调用和被调用的关系。该表的抽象层次较高，它指导我们如何深入更低的层次。

表 15-1 后端简化的调用树

例程名	目的
linearize	为输出一棵指令树排序
ralloc	为一条指令释放和分配寄存器
putreg	释放一个忙寄存器

（续）

例程名	目的
getreg	发现和分配一个寄存器
askreg	发现和分配一个空闲寄存器
asktfixedreg	尝试分配一个指定寄存器
spillee	标记一个寄存器溢出
spill	溢出一个或多个寄存器
spillr	溢出一个寄存器
genspill	产生代码溢出一个寄存器
genreload	产生代码重载一个被溢出的寄存器的值
reprune	当 genreload 更新 x.kids 后，更新 kids

在后端选择指令并将子指令从树（树是通过 x.kids 数组链接而成的）中分离出来后，linearize 采用前序方法遍历分离后的树，并按照最后执行的顺序链接指令。gen 遍历此表，将每条指令传递给 ralloc 函数，ralloc 一般首先调用 putreg 来释放不再被其子节点使用的寄存器，然后调用 getreg 函数为自身分配一个寄存器。对于临时变量，ralloc 在首次赋值的时候为它分配一个寄存器，在最后一次使用的时候释放该寄存器。

如果 getreg 发现没有适合指令的空闲寄存器，那么 getreg 调用 spillee 函数以识别将来最远使用的寄存器，然后调用 spill 函数生成代码把这个寄存器的值溢出到存储器，以后需要时再重取该值。genspill 函数引发溢出，genreload 函数用表示从存储器中取值的节点来替换所有还未被处理的、对该溢出寄存器的使用。genreload 调用 reprune 函数重建 kids 与 x.kids 之间的关系，而这种关系在溢出改变森林之前是由 prune 函数建立的。

对这些例程而言，ralloc 不是唯一的入口点。clobber 可以直接调用 spill 来溢出和重取寄存器，就像调用程序和被调用程序之间保存寄存器一样。并且，每个目标机器的接口程序 local 都能通过调用 askregvar 来调用 askreg，askreg 为寄存器变量分配寄存器。

15.2 寄存器状态跟踪

掩码记录了例程编译过程中哪些寄存器空闲，哪些已被使用。freemask 跟踪哪些寄存器是空闲的，告诉寄存器分配器哪个寄存器可以被分配；usedmask 跟踪所有被当前例程使用的寄存器，告诉 function 函数哪些寄存器在过程开始时必须保存，哪些在过程结束时必须恢复。这两个掩码都用向量表示，向量中的每个元素对应一个寄存器组。

```
⟨gen.c data⟩+≡                                          313 319
  unsigned freemask[2];
  unsigned usedmask[2];
```

每个目标机器的 function 接口过程将这些掩码初始化成没有寄存器被使用，即所有的寄存器都是空闲的：

```
⟨clear register state319⟩≡                              350 379 407
  usedmask[0] = usedmask[1] = 0;
  freemask[0] = freemask[1] = ~(unsigned)0;
```

每个 progbeg 函数将设置相似的两个掩码 tmask 和 vmask。tmask 表示用于临时变量的寄存器，vmask 表示分配给寄存器变量的寄存器。

```
⟨gen.c data⟩+≡                                          319
  unsigned tmask[2];
  unsigned vmask[2];
```

不可分配的寄存器既不属于tmask也不属于vmask，例如栈指针。

freemask和usedmask的值由低层的例程putreg、getreg、askreg和askregvar维护，这些程序分配并释放某个寄存器。putreg释放符号r表示的寄存器。只有freemask可以区别空闲与被使用的寄存器，所以除修改freemask外，putreg不需要改变别的什么。

```
⟨gen.c functions⟩+≡                                    ↑315 320↓
  static void putreg(r) Symbol r; {
     freemask[r->x.regnode->set] |= r->x.regnode->mask;
  }
```

如果有可能，askfixedreg分配固定寄存器r，如果该寄存器忙，askfixedreg返回一个空指针，否则该函数调整寄存器状态记录并返回r。

```
⟨gen.c functions⟩+≡                                    ↑320 320↓
  static Symbol askfixedreg(s) Symbol s; {
     Regnode r = s->x.regnode;
     int n = r->set;

     if (r->mask&~freemask[n])
        return NULL;
     else {
        freemask[n] &= ~r->mask;
        usedmask[n] |=  r->mask;
        return s;
     }
  }
```

askreg的参数可以是一个表示固定寄存器的符号或寄存器组的通配符。askreg的第二个参数是对通配符进行限制的掩码。如果寄存器是固定的，askreg只需要调用askfixedreg；否则，askreg寻找一个符合掩码并由通配符表示的寄存器组的空闲寄存器：

```
⟨gen.c functions⟩+≡                                    ↑320 321↓
  static Symbol askreg(rs, rmask)
  Symbol rs; unsigned rmask[]; {
     int i;

     if (rs->x.wildcard == NULL)
        return askfixedreg(rs);
     for (i = 31; i >= 0; i--) {
        Symbol r = rs->x.wildcard[i];
        if (r != NULL
        && !(r->x.regnode->mask&~rmask[r->x.regnode->set])
        && askfixedreg(r))
           return r;
     }
     return NULL;
  }
```

使用寄存器掩码对寄存器组中寄存器的数量设定上限，这个上限即为编译器宿主机上无符号整型掩码的位数。到目前为止，该值在每个目标机器上都已经达到32，所以将askregvar循环次数初始化为32次较为合理。但是如果定义更小的寄存器组，lcc最新的代码生成器（在X86上）的编译速度将会更高。当然有些机器拥有更大的寄存器组，例如现在已经有了具有32个64位整型寄

存器的机器，但这种64位的机器不符合我们当初设计本方法的初衷。我们可以用一个适应大小可变的集合的结构来表示寄存器组，改进现有的方法。

getreg 函数请求分配一个寄存器。若 askreg 无法找到合适的寄存器，getreg 调用 spillee 选出一个寄存器溢出，然后调用 spill 修改森林，加入指令，把该寄存器的值存储到存储器并在需要的时候重取出。这样第二次调用 askreg 就保证能找到一个寄存器了。

```
⟨gen.c functions⟩+≡                                          320 321
  static Symbol getreg(s, mask, p)
  Symbol s; unsigned mask[]; Node p; {
     Symbol r = askreg(s, mask);
     if (r == NULL) {
        r = spillee(s, p);
        spill(r->x.regnode->mask, r->x.regnode->set, p);
        r = askreg(s, mask);
     }
     r->x.regnode->vbl = NULL;
     return r;
  }
```

如果一个寄存器已经分配给一个变量，那么 x.regnode->vbl 则指向表示该变量的符号；getreg 的默认设定是寄存器未分配给变量，所以清空了 vbl 域。

askregvar 试图为一个局部变量或一个形式参数分配寄存器。如果分配成功就返回 1，否则返回 0：

```
⟨gen.c functions⟩+≡                                          321 322
  int askregvar(p, regs) Symbol p, regs; {
     Symbol r;

     ⟨askregvar 321⟩
  }
```

如果变量是聚合类型，或者变量没有寄存器存储类别，则 askregvar 拒绝为其分配寄存器：

```
⟨askregvar 321⟩=                                             321  321
  if (p->sclass != REGISTER)
     return 0;
  else if (!isscalar(p->type)) {
     p->sclass = AUTO;
     return 0;
  }
```

如果设置了 u.t.cse，那么该变量就是一个存储公共子表达式的临时变量。askregvar 将会延迟分配，直到寄存器分配器处理了这个表达式：

```
⟨askregvar 321⟩+≡                                            321 322  321
   else if (p->temporary && p->u.t.cse) {
      p->x.name = "?";
      return 1;
   }
```

延迟处理可以帮助 lcc 将一个寄存器用于多个临时变量。为了在调试编译器时区分这些变量，askregvar 在这些临时变量的 x.name 域中临时设置了一个问号标识。

如果上述条件没有一个满足，askfcgvar 就向 askreg 申请寄存器。若请求成功，符号 r 指向所申请到的寄存器：

```
⟨askregvar 321⟩+≡                                          321 322 321
  else if ((r = askreg(regs, vmask)) != NULL) {
    p->x.regnode = r->x.regnode;
    p->x.regnode->vbl = p;
    p->x.name = r->x.name;
    return 1;
  }
```

否则，变量就被强制放到栈上：

```
⟨askregvar 321⟩+≡                                          322 321
  else {
    p->sclass = AUTO;
    return 0;
  }
```

15.3 寄存器分配

寄存器分配过程是从构造指令执行顺序开始的。linearize(p, next) 把以 p 为根的指令树线性化，得到的序列通过 x.next 和 x.prev 形成双向链表。参数 next 指向当前构造的序列末尾的一个标识点，linearize 函数在图 14-5 中添加点线，将其转换为图 15-1。

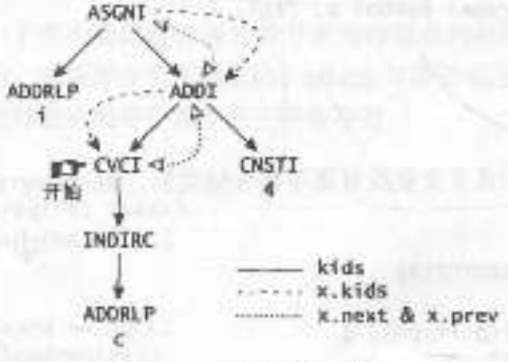


图 15-1 将使用节点排序

```
⟨gen.c macros⟩+≡                                           308
  #define relink(a, b) ((b)->x.prev = (a), (a)->x.next = (b))

⟨gen.c functions⟩+≡                                        321 325
  static void linearize(p, next) Node next, p; {
    int i;

    for (i = 0; i < NELEMS(p->x.kids) && p->x.kids[i]; i++)
      linearize(p->x.kids[i], next);
    relink(next->x.prev, p);
    relink(p, next);
  }
```

linearize 按照前序遍历树，所以它从递归处理子树开始，然后在不断增长的序列上添加 p，相当于将 p 插入 next 和 next 的前驱之间。第一个 relink 把 next 的前驱的向后指针指向 p，p 的向前指针指向 next 的前驱。第二个 relink 对 p 和 next 的操作与前相同，即 p 的向后指针指向 next，next 的向前指针指向 p。

gen 调用 relink 对序列做初始化，形成只含一个标记点的循环链表：

```
⟨linearize forest 323⟩=                                    323  314
  relink(&sentinel, &sentinel);
```

然后，gen 沿着森林扫描，调用 linearize 把每个列出的树线性化，并将该树链接到不断增长的序列中，放在标记点前：

```
⟨linearize forest 323⟩+=                                   323 323  314
  for (p = forest; p; p = p->link)
    linearize(p, &sentinel);
```

循环结束时，gen 将 forest 指向序列开头，该节点也就是循环列表中跟在标记点之后的节点：

```
⟨linearize forest 323⟩+=                                   323 323  314
  forest = sentinel.x.next;
```

最后，gen 将第一个 x.prev 和最后一个 x.next 清零，中断循环链表：

```
⟨linearize forest 323⟩+=                                   323  314
  sentinel.x.next->x.prev = NULL;
  sentinel.x.prev->x.next = NULL;
```

寄存器分配器对森林进行 3 遍扫描。第一遍对所有使用临时变量的节点建立一个列表。该列表指明了临时变量节点的最后一次使用，也就是释放临时变量的时刻。该列表还标识出当某个临时变量溢出到存储器时需要发生变化的节点。如果 p->syms[RX] 指向一个临时变量，那么 p->syms[RX]->x.lastuse 的值就指向使用 p 的最后一个节点，而该节点的 x.prevuse 指向前一个使用 p 的节点，依次类推。这个列表包括了所有读写临时变量的节点：

```
⟨allocate registers 323⟩=                                  324  314
  for (p = forest; p; p = p->x.next)
    for (i = 0; i < NELEMS(p->x.kids) && p->x.kids[i]; i++) {
      if (p->x.kids[i]->syms[RX]->temporary) {
        p->x.kids[i]->x.prevuse =
          p->x.kids[i]->syms[RX]->x.lastuse;
        p->x.kids[i]->syms[RX]->x.lastuse = p->x.kids[i];
      }
    }
```

这段程序使用了嵌套循环，首先对指令进行循环，然后对每条指令的子节点进行循环，确保按照指令执行顺序访问这些使用节点。用一个非嵌套的单层循环访问森林是一种吸引人的方法，但却不正确：

```
for (p = forest; p; p = p->x.next)
  if (p->syms[RX]->temporary) {
    p->x.prevuse = p->syms[RX]->x.lastuse;
    p->syms[RX]->x.lastuse = p;
  }
```

该程序访问的使用节点与前面的程序一样，但是对于某些输入来说，访问顺序可能有误。例如，a[i]=a[i]+1 两次用到 a[i] 的地址，需要将 a[i] 的地址赋给一个临时变量。上面错误的代码将会首先访问 INDIR 节点，为左边的 a[i] 引用取得临时变量值，这样为右边的 a[i] 取临时变量的 INDIR 就会作为最后一次使用节点出现。取完数后，临时变量就会被释放，用于存放其他的值，这时后面的存储指令将会用到一个错误的地址。图 15-2 就显示了这个例子在 x.prevuse 链上循环嵌套的作用。

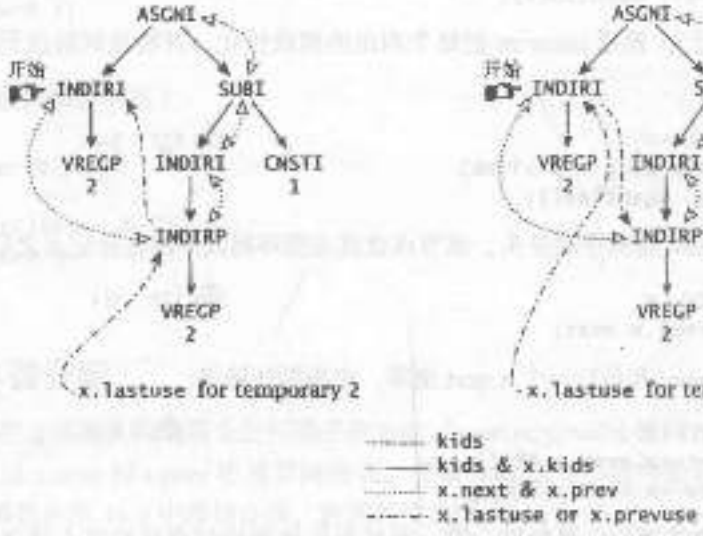


图 15-2 将使用节点排序，左图显示了单层循环的错误结果，右图显示了嵌套的双重循环的正确结果

第二遍对森林的扫描删去了一些用于寄存器间复制的指令，实现方式是把计算源寄存器的表达式重新定向，使用目的寄存器。如果源寄存器是个公共子表达式，若两条指令之间是顺序执行的，且不改变目的寄存器，我们就用目的寄存器保存该公共子表达式：

```
⟨allocate registers 323⟩+≡                                  323 325  314
  for (p = forest; p; p = p->x.next)
  if (p->x.copy && p->x.kids[0]->syms[RX]->u.t.cse) {
     Symbol dst = p->syms[RX];
     Symbol temp = p->x.kids[0]->syms[RX];
     Node q;

     for (q = temp->u.t.cse; q; q = q->x.next)
        if (p != q && dst == q->syms[RX]
        || (⟨changes flow of control? 325⟩))
           break;
     if (!q)
        for (q = temp->x.lastuse; q; q = q->x.prevuse)
           q->syms[RX] = dst;
  }
```

第一个内层循环扫描森林剩余的部分，如果目的寄存器是在程序块的后面部分被设置，或者某个节点改变了控制流，就会提早退出循环。内层循环也可以在临时变量释放时退出，但是这个附加的逻辑控制在测试时，25 000 条指令中仅减少了 5 条指令，所以我们不采用这种控制。如果没有其他的节点设置目的寄存器，那么将这个寄存器用于公共子表达式是安全的。第二个内层循环把

所有公共子表达式的实例替换成目的寄存器。一旦公共子表达式被计算后放进 dst，最初的“寄存器－寄存器”复制就把 dst 复制到自身。代码产生器与 moveself 合作删掉了这样的指令。

调用不会改变寄存器变量，因此，只要目的寄存器不存有寄存器变量，就可以把调用看成是顺序执行代码中的一个中断：

```
(changes flow of control? 325)≡                                    324
  q->op == LABELV || q->op == JUMPV || generic(q->op)==RET ||
  generic(q->op)==EQ || generic(q->op)==NE ||
  generic(q->op)==LE || generic(q->op)==LT ||
  generic(q->op)==GE || generic(q->op)==GT ||
  (generic(q->op) == CALL && dst->sclass != REGISTER)
```

最后一遍扫描森林为每个节点分配一个寄存器。rmap 是一个以类型后缀进行索引的向量，它的每个元素都是一个表示寄存器组的寄存器通配符。这些寄存器组适合具有相应类型的未定位节点。

```
(allocate registers 323)+≡                                  324  314
  for (p = forest; p; p = p->x.next) {
     ralloc(p);
     if (p->x.listed && NeedsReg[opindex(p->op)]
     && rmap[optype(p->op)]) {
        putreg(p->syms[RX]);
     }
  }
```

当父节点执行到 ralloc 时，释放寄存器。但是有一些节点（如 CALLI）即使寄存器值不再被使用，由于该节点没有父节点，它还是占用了寄存器。上面的 if 语句将释放分配给这些节点的寄存器。现有的目标机器只把上述代码用于处理 CALL 和 LOAD。

ralloc(p) 释放了所有 p 的子节点不再需要的寄存器，然后，如果 p 需要一个寄存器，且之前未被处理，就给 p 分配一个寄存器。最后，ralloc 调用目标机器的 clobber，溢出所有因这个节点而被溢出的寄存器。

```
(gen.c functions)+≡                                         322  329
  static void ralloc(p) Node p; {
     int i;
     unsigned mask[2];

     mask[0] = tmask[0];
     mask[1] = tmask[1];
     (free input registers 326)
     if (!p->x.registered && NeedsReg[opindex(p->op)]
     && rmap[optype(p->op)]) {
        (assign output register 326)
     }
     p->x.registered = 1;
     (*IR->x.clobber)(p);
  }
```

如果子节点产生一个寄存器变量，或者寄存器存有公共子表达式，且该子表达式还有其他使用，那么这个寄存器就不能被释放，下面的 if 语句正好捕获了这些例外：

```
⟨free input registers 326⟩≡                                    325
  for (i = 0; i < NELEMS(p->x.kids) && p->x.kids[i]; i++) {
     Node kid = p->x.kids[i];
     Symbol r = kid->syms[RX];
     if (r->sclass != REGISTER && r->x.lastuse == kid)
        putreg(r);
  }
```

r->x.lastuse 指向 r 的最后一次使用。大多数表达式的临时变量通常只被使用一次，但是分配给公共子表达式的临时变量会被多次使用。

现在 ralloc 为该节点分配寄存器。prelabel 函数在 p->syms[RX] 中存储了一个寄存器或通配符，表示 p 能接受的寄存器。但是，askregvar 已把公共子表达式的 p->syms[RX] 指向了一个还没有分配寄存器的寄存器变量，使情况变得更为复杂。因此，我们需要用到两个值 sym 和 set，其中 sym 等于 p->syms[RX]，set 是适合 p 的寄存器组：

```
⟨assign output register 326⟩≡                                 326 325
  Symbol sym = p->syms[RX], set = sym;
  if (sym->temporary && sym->u.t.cse)
     set = rmap[optype(p->op)];
```

如果 p 不需要寄存器，那么 ralloc 就执行完毕；否则，ralloc 向 getreg 申请一个寄存器并且把它存储在该节点或者需要它的所有节点中：

```
⟨assign output register 326⟩+≡                                 326 325
  if (set->sclass != REGISTER) {
     Symbol r;
     ⟨mask out some input registers 327⟩
     r = getreg(set, mask, p);
     ⟨assign r to nodes 327⟩
  }
```

ralloc 在分配输出寄存器之前要释放输入寄存器，这样输入寄存器可以当作输出寄存器重复使用。当节点用单指令实现时，这种节省总是安全的；但是如果节点由指令序列来实现，这种节省就不安全了：如果输出寄存器是输入寄存器之一，而且指令序列在读相应的输入寄存器之前改变了输出寄存器，那么读操作取出的是一个错误的值。因此要注意，所有产生指令序列的规则只能在完成所有对输入寄存器的读操作之后才能设置输出寄存器。大多数模板仅产生一条指令，所以前面我们假定节点是单指令实现的可以作为系统的默认值，但也要充分考虑多指令序列的情况。

对于需要输出寄存器是输入寄存器之一的指令来说，这个规则很不实用。例如 X86 的加法指令只取两个操作数，它把第二操作数加到第一操作数上，并将结果存于第一操作数中。如果第一操作数还没有失效，生成的代码就必须把结果放到一个空闲的寄存器中，因此加法首先就要把第一操作数复制到该寄存器中，这样代码模板就有两条指令：第一条是将第一操作数复制到目的寄存器中，第二条是计算和。例如，X86 的加法模板是：

```
reg: ADDI(reg,mrl1)  "mov %c,%0\nadd %c,%1\n"  2
```

这样的模板在完成所有对输入寄存器的读操作之前改变了输出寄存器，所以破坏了上述规则。

为了处理两操作数指令，我们在其代码模板的开头用一个问号做标记。上述规则的完整形式是：

```
reg: ADDI(reg,mrl1)  "?mov %c,%0\nadd %c,%1\n"  2
```

当 ralloc 看到这样的规则时，就修改了 mask 以防止对其他所有输入寄存器（除了第一个）进行重新分配。因此下面的循环从 1 而不是从 0 开始：

```
⟨mask out some input registers 327⟩=                                    326
  if (*IR->x._templates[getrule(p, p->x.inst)] == '?')
     for (i = 1; i < NELEMS(p->x.kids) && p->x.kids[i]; i++) {
        Symbol r = p->x.kids[i]->syms[RX];
        mask[r->x.regnode->set] &= ~r->x.regnode->mask;
     }
```

代码生成器必须注意没有节点与其子节点（除了第一个子节点）定位在相同的寄存器上。

寄存器一旦被分配，ralloc 就把它存入使用它的节点中：

```
⟨assign r to nodes 327⟩=                                                326
  if (sym->temporary && sym->u.t.cse) {
     Node q;
     r->x.lastuse = sym->x.lastuse;
     for (q = sym->x.lastuse; q; q = q->x.prevuse) {
        q->syms[RX] = r;
        q->x.registered = 1;
        if (q->x.copy)
           q->x.equatable = 1;
     }
  } else {
     p->syms[RX] = r;
     r->x.lastuse = p;
  }
```

如果该节点不是一个公共子表达式，else 从句就会把 r 存进 p->syms[RX]，并在 r->x.lastuse 中记录下这个使用。如果 sym 是一个公共子表达式，则 x.lastuse 已经标明了使用点。程序段需要遍历使用列表，存储 r，并给节点做上已被寄存器分配器处理过的标记。另外，如果公共子表达式已经存在于其他一些寄存器中，则设置节点的 x.equatable。

15.4 寄存器溢出

当寄存器分配器用完寄存器时，需要生成代码溢出一个寄存器，将其值存回到存储器中，并将那些未被处理的使用该寄存器的节点替换成从存储器中重取该值的节点。实际上还有更多有创意的方法（参见练习 15.6 和练习 15.7），但是 lcc 没有采纳这些方法。因为溢出并不常出现，所以 lcc 的溢出程序在不影响与目标机器无关性的情况下已经尽可能简单。花精力去优化调试那些很少使用的代码是一种无谓的浪费。现实中很难找到测试这些代码的程序，也很难将这些测试程序独立出来，要彻底地测试一个复杂的溢出实现是非常困难的事情。

当寄存器分配器用完寄存器时，最优选择是把最远使用的寄存器溢出到存储器中。溢出器将未被处理的使用该寄存器的节点替换成从存储器中重取该值的节点，然后释放寄存器以满足当前要求。

溢出由几个例程合作处理：spillee 识别被适合溢出的寄存器，spill 调用 genspill 来插入溢出代码，genreload 插入重取代码。图 15-3 说明了针对下面的程序，这几个例程是如何操作的：

```
int i;
main() { i = f() + f(); }
```

这是大多数目标机器上都会导致输出的最简单的程序。第一次调用的值从返回寄存器中溢出，这样该返回值就不会被第二次调用破坏了。

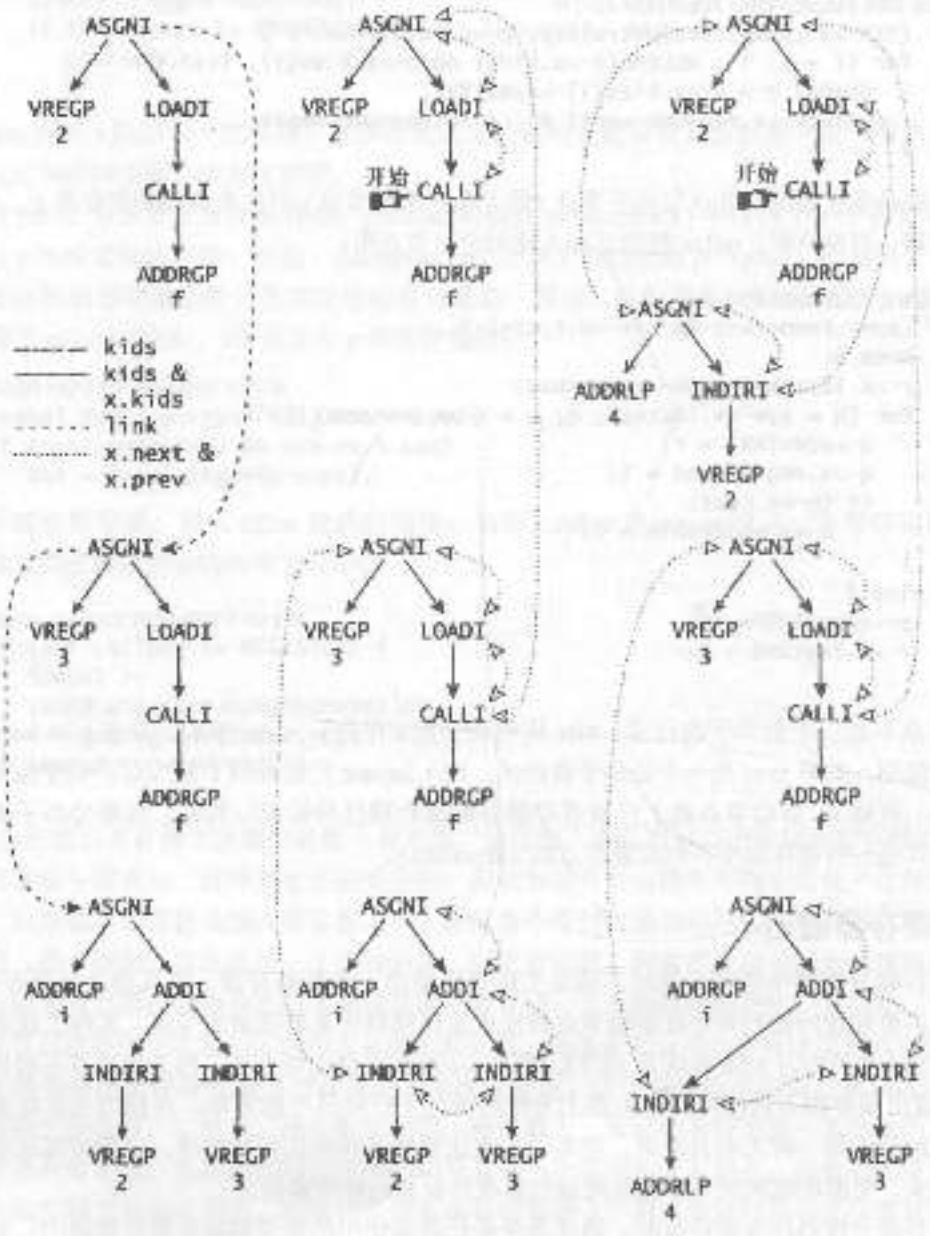


图15-3 f()中的溢出

图中第1列显示了代码生成前的森林，即来自编译器前端的森林经prelabel函数处理后形成的森林，prelabel把访问临时寄存器变量的ADDRL替换成VREG，并插入LOAD以写入这样的寄存器。第2列显示了线性化之后的森林，它假定带有空心箭头的连续连接的节点是指令（尽管读写寄存器的节点INDIR和ASGN只是典型的注释指令），剩下的是地址计算之类的子指令。最后一列展示了插入的溢出和重取操作，这里使用了ADDRLP(4)。后两列的黑色箭头显示了kids列x.kids，这是将子指令从树中删除后剩下的连接。

getreg 用完寄存器后，调用 spillee(set,here)，在 set 中寻找使用点离 here 最远的寄存器：

```
⟨gen.c functions⟩+≡                                    325 329
  static Symbol spillee(set, here) Node here; Symbol set; {
     Symbol bestreg = NULL;
     int bestdist = -1, i;

     if (!set->x.wildcard)
        return set;
     for (i = 31; i >= 0; i--) {
        Symbol ri = set->x.wildcard[i];
        if (ri != NULL && ri->x.lastuse
        && ri->x.regnode->mask&tmask[ri->x.regnode->set]) {
           Regnode rn = ri->x.regnode;
           Node q = here;
           int dist = 0;
           for (; q && !uses(q, rn->mask); q = q->x.next)
              dist++;
           if (q && dist > bestdist) {
              bestdist = dist;
              bestreg = ri;
           }
        }
     }
     return bestreg;
  }
```

如果 set 不是一个通配符，那么就表示一个单一的寄存器，也只有这个寄存器可以被溢出，spillee 就简单地返回这个寄存器。否则，set 表示一个相应的寄存器组，spillee 从中找出一个最远使用的元素。spillee 调用 uses 检查节点 p 是否读取了一个给定的寄存器：

```
⟨gen.c functions⟩+≡                                    329 329
  static int uses(p, mask) Node p; unsigned mask; {
     int i;
     Node q;
     for (i = 0; i < NELEMS(p->x.kids)
        && (q = p->x.kids[i]) != NULL; i++)
        if (q->x.registered
        && mask&q->syms[RX]->x.regnode->mask)
           return 1;
     return 0;
  }
```

spillr(r,here) 溢出寄存器 r，并且把 here 之后 r 的每个使用都替换成一个重取操作：

```
⟨gen.c functions⟩+≡                                    329 330
  static void spillr(r, here) Symbol r; Node here; {
     int i;
     Symbol tmp;
     Node p = r->x.lastuse;
     ⟨spillr 330⟩
  }
```

spillr把寄存器溢出到存储器，有时也可能溢出到另一个寄存器，但是这样可能会引发另一个溢出，从而导致无限递归，把逻辑复杂化了。spillr找到第一个使用点（x.prevuse链以第一个使用点结束），即对r进行定值的使用点：

```
⟨spillr 330⟩=                                          330  329
  while (p->x.prevuse)
    p = p->x.prevuse;
```

r可以存放一个简单的只被使用一次的表达式临时变量，也可能保存一个使用多次的公共子表达式。二者都是由一条指令完成赋值的。spillr找到这个变量后，把它送给genspill，genspill在森林的赋值点上嵌入一个溢出节点：

```
⟨spillr 330⟩+≡                                  330 330  329
  tmp = newtemp(AUTO, optype(p->op));
  genspill(r, p, tmp);
```

溢出可以在定值节点与here之间的任何位置进行，但在赋值的位置进行是很安全的，这也解释了为什么表15-3中的溢出发生在最后一列的第一棵树。

接下来，spillr将所有剩余的读取r的节点改成从溢出单元取值；最后是释放r：

```
⟨spillr 330⟩+≡                                          330  329
  for (p = here->x.next; p; p = p->x.next)
    for (i = 0; i < NELEMS(p->x.kids) && p->x.kids[i]; i++) {
      Node k = p->x.kids[i];
      if (k->x.registered && k->syms[RX] == r)
        genreload(p, tmp, i);
    }
  putreg(r);
```

基于一些错综复杂的原因，从here->x.next开始搜索读取r的节点，而不是从here开始。here能将自身的一个子节点溢出。例如，代码(*f)()可把一个指针的值写入调用程序保存的寄存器，然后使用一个间接调用指令，这样clobber一定会溢出。大多数指令模板只含有一条指令，所以必须在破坏之前完成读取输入寄存器的操作。例如，间接调用在利用完其值之前不会破坏地址寄存器，除非该地址寄存器又被调用指令之后的其他指令访问，这些指令都是在here->x.next之后的操作。

另外，genreload不会调用ralloc为插入的节点分配寄存器。genreload把每个重取指令嵌入指令序列中，就在使用重取值的指令之前。这些指令已经在here或在其之前访问了溢出值，但是genreload把它们修改成使用here之后的重取指令。我们只是把对新指令的寄存器分配，推迟到ralloc在余下的指令序列中遇到它们的时候。

genspill(r,last,tmp)在last处把对r的赋值溢出到tmp：

```
⟨gen.c functions⟩+≡                                     329 332
  static void genspill(r, last, tmp)
  Symbol r, tmp; Node last; {
    Node p, q;
    Symbol s;
    unsigned ty;

    ⟨genspill 331⟩
  }
```

genspill 合成一个具有适当类型的寄存器变量用于溢出：

```
(genspill 331)=                                          331  330
  ty = optype(last->op);
  if (ty == U)
     ty = I;
  NEW0(s, FUNC);
  s->sclass = REGISTER;
  s->x.name = r->x.name;
  s->x.regnode = r->x.regnode;
  s->x.regnode->vbl = s;
```

被溢出的寄存器不是一个寄存器变量，但由于 INDIRx（VREGP）不会产生任何指令，所以我们假定它是一个寄存器变量，以保证不会有生成计算溢出值的指令。溢出值已经被计算出来了，不需要额外的指令。接着 genspill 创建节点把寄存器溢出到存储器：

```
(genspill 331)+=                                         331 331  330
  q = newnode(ADDRLP, NULL, NULL, s);
  q = newnode(INDIR + ty, q, NULL, NULL);
  p = newnode(ADDRLP, NULL, NULL, tmp);
  p = newnode(ASGN + ty, p, q, NULL);
```

现在，genspill 选择指令，删除子指令并对得到的指令树进行线性化：

```
(genspill 331)+=                                         331 331  330
  rewrite(p);
  prune(p, &q);
  q = last->x.next;
  linearize(p, q);
```

最后，genspill 用寄存器分配器遍历新产生的节点：

```
(genspill 331)+=                                         331  330
  for (p = last->x.next; p != q; p = p->x.next) {
     ralloc(p);
  }
```

如果是因为 ralloc 用完了寄存器而引起对 genspill 的调用，并且这些调用确实想要分配一个寄存器，那么这些调用可能会导致无穷递归。我们必须注意代码生成器可以在不分配新寄存器的情况下使一个寄存器溢出。溢出就是存储，通常只需要一条指令，因而不需要额外的寄存器。但是有些机器限制了地址计算中常量部分的大小，这样就需要两条指令和一个临时寄存器来将数据存储到任意地址。所以我们必须确保这些存储所使用的寄存器不会被 ralloc 做另外的分配。MIPS R3000 体系结构中有这样的限制条件，但它是通过为汇编程序预留临时寄存器，由汇编程序解决这个问题的。SPARC 是到目前为止唯一需要关注代码生成器的目标机器。17.2 节将对此进行详述。

genspill 对 ralloc 的调用不能分配任何寄存器，但是 ralloc 还负责分配寄存器以外的工作，因此 genspill 仍将调用它。例如，ralloc 可以调用目标机器的 clobber。当然，用一个简单的存储操作就能使 clobber 完成任何事情，似乎不太现实，但是将来会有一些目标机器能做到这一点。所以如果不调用 ralloc，genspill 将掩盖一个潜在的错误。编译器后端对所有其他的节点都调用 rewrite、prune、linearize 和 ralloc，因此对溢出节点而言，忽略任何一步都是不明智的。

genreload(p,tmp,i) 修改了 p->x.kids[i] 以读取 tmp，而不是读取已被溢出的寄存器：

```
⟨gen.c functions⟩+≡                                    330 333
  static void genreload(p, tmp, i)
  Node p; Symbol tmp; int i; {
     Node q;
     int ty;

     ⟨genreload 332⟩
  }
```

genreload 将目标节点改成一棵读取 tmp 的树，并为其选择指令以及剔除了指令：

```
⟨genreload 332⟩=                                       332  332
  ty = optype(p->x.kids[i]->op);
  if (ty == U)
     ty = I;
  q = newnode(ADDRLP, NULL, NULL, tmp);
  p->x.kids[i] = newnode(INDIR + ty, q, NULL, NULL);
  rewrite(p->x.kids[i]);
  prune(p->x.kids[i], &q);
```

接着，genreload 线性化重取指令，在调用 prune 对指令树剪枝之后这样做是很正常的，但首先需要增加额外的同步：

```
⟨genreload 332⟩+≡                                      332  332
  reprune(&p->kids[1], reprune(&p->kids[0], 0, i, p), i, p);
  prune(p, &q);
  linearize(p->x.kids[i], p);
```

在大多数情况下，x.kids 的每个入口都是通过 prune 从某些 kids 的入口复制的，但是 genreload 只改变 x.kids[i]，并不修改任意 kids 中相应的入口。由于产生器要使用 kids，所以 genreload 必须找到相应的入口并进行更新。genreload 调用 reprune 完成这项工作，并且第二次调用 prune 对 p 指向的节点做了类似的修改。

当 p->x.kids[n] 被修改后，genreload 调用 reprune(pp,k，n，p) 重建 kids 与 x.kids 之间的连接，也就是说，reprune 必须使得重取指令看起来就像一开始就在森林中一样。这样，reprune 是 prune 的一个扩充版本：prune 为一棵完整的树在 kids 与 x.kids 之间建立了对应，而 reprune 是在其中一个发生改变之后重建这种对应，即其中一个与重取相关。图 15-4 显示了 reprune 怎样修复图 15-3 中最后的树。

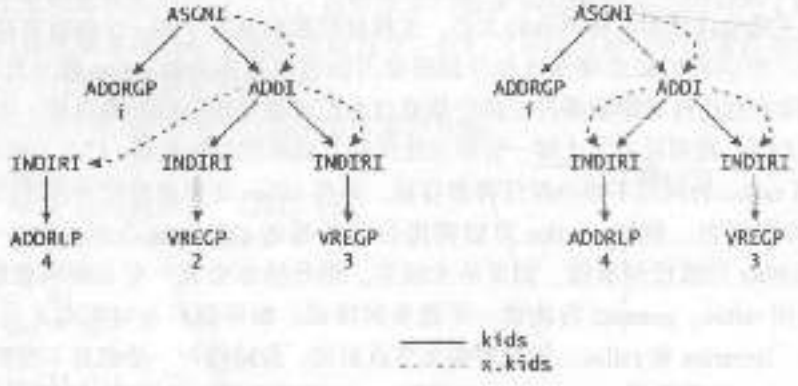


图 15-4 图 15-3 的重取在调用 reprune 之前和之后的情形

起初，在根层次上调用 reprune 时有一个指针 pp，它指向第一个可能需要修改的 kids 入口：

```
(gen.c functions)+≡                                          332 333
  static int reprune(pp, k, n, p) Node p, *pp; int k, n; {
     struct node x, *q = *pp;
      if (q == NULL || k > n)
         return k;
      else if (q->x.inst == 0)
         return reprune(&q->kids[1],
            reprune(&q->kids[0], k, n, p), n, p);
      if (k == n) {
         *pp = p->x.kids[n];
         x = *p;
         (IR->x.target)(&x);
      }
      return k + 1;
   }
```

kids 链表原来的树，x.kids 链接指令树。第 2 步是对第 1 步做删除，但删除的节点数并不确定，所以对树进行递归搜索才能找到与 p->x.kids[i] 对应的 kids 入口。对 reprune 的递归调用与对 prune 的递归调用类似。这些调用会干扰 k 的值，该值从 0 开始，当 prune 找到一条指令，并在 x.kids 中设置下一个入口时，让 p 继续前进。故当 k 到达 n 时，reprune 已经找到了 kids 入口进行更新，而且 x 保留了对重取的改动。

gctreg 函数和每个目标机器的 clobber 函数都调用 spill(mask,n,here) 来溢出寄存器组 n 中所有的忙寄存器，这些寄存器由 mask 指定。一个典型的应用就是 CALL 节点，一般说来，函数调用总会引发某些寄存器的冲突，它们在调用之前必须溢出，调用之后必须重新读取。spill 将这些寄存器标记为已使用，然后遍历森林的其余部分，寻找需要溢出的活动寄存器。比较经济的做法是先确认是否存在需要溢出的寄存器。

```
(gen.c functions)+≡                                          333
  void spill(mask, n, here) unsigned mask; int n; Node here; {
     int i;
     Node p;

     usedmask[n] |= mask;
     if (mask&~freemask[n])
        for (p = here; p; p = p->x.next)
           (spill333)
  }
```

下面的内层循环识别了需要溢出的活动寄存器，并调用 spillr 来溢出它们：

```
(spill333)≡                                                  333
  for (i = 0; i < NELEMS(p->x.kids) && p->x.kids[i]; i++) {
     Symbol r = p->x.kids[i]->syms[RX];
     if (p->x.kids[i]->x.registered && r->x.regnode->set == n
     && r->x.regnode->mask&mask)
        spillr(r, here);
  }
```

溢出操作使 lcc 可以释放调用程序保存的寄存器，这也是用于处理一些古怪指令的机制的特例，这些古怪指令会破坏某些固定寄存器组。

深入阅读

指令的执行顺序决定了指令的输出顺序。大多数语言只是部分受约束。例如，ANSI 规定必须按顺序地计算赋值语句，但是又不关心先计算赋值的哪个操作数。lincarize 使用一个固定的顺序，也可能有更好的选择。例如，Sethi-Ullman 标记（numbering）算法（Aho，Sethi and Ullman，1986）采用的办法就是先计算需要最多个寄存器的子节点以节省寄存器。

指令调度和寄存器分配相互影响。在使用指令结果之前早早地开始执行较慢的指令是有好处的，但是这样将要占用结果寄存器更长的时间，因而需要更多寄存器。Proebsting and Fischer（1991）解决了从一个方面进行简单的取舍。Krishnamurthy（1990）综述了有关指令调度的许多文献。

许多功能强大的寄存器分配器使用了图着色方法。编译器建立一张图，其中节点是计算得到的值。当且仅当两个值同时有效时把这两个节点链接起来，这意味着它们不能共用一个寄存器，也就是不能标为同一种颜色。Chaitin et al.（1981）描述了这个过程。

选择一个寄存器溢出与操作系统中的页替换是紧密相关的。虚拟存储系统并不知道哪一个是最近使用的页，但溢出器能决定哪一个是最近使用的寄存器（Freiburghouse，1974）。

练习

15.1 15.3 节描述了 lcc 所没有使用的优化措施，该措施在测试时 25 000 条指令中只省掉了 5 条指令。请实现这样一个优化程序，看看是否能发现优化效果更好的有用程序。

15.2 用 Sethi-Ullman 标记算法改写 lcc，看看对应于你的程序，lcc 编译产生的代码的效率提高了多少？

15.3 构造一些输入程序以检验溢出器

15.4 对 spillcc 进行改造，使其溢出最近最少使用的寄存器。通过计时，你能发现改造前后在编译速度上或在生成的代码质量上所发生的变化吗？

15.5 最简单的编译器没有溢出器，发生问题时会退出并启动诊断程序。删除 lcc 的溢出器并修改 clobber。这个新的编译器能够简化到什么程度？在你常用的 C 语言程序中，在任意一个程序发生编译错误之前，已经编译了多少程序？如果将 CALL 节点改成从结果寄存器复制到任意一个寄存器，以避免 8)(4) 中的溢出，那么前面一问中的程序的数目将怎样改变？

15.6 有些溢出发生在寄存器仍然有用的时候。例如，表达式 f()+f()，尽管其他寄存器可能是空闲的，但第二次调用需要同一个返回寄存器，因此必须从返回寄存器中溢出第一个返回值。修改 lcc 的溢出器以将溢出的值存入另一个寄存器，这种改变对你常用的 C 语言程序代码能起多大改进？这样做值得吗？

15.7 lcc 为溢出之后处理的每一次引用都生成了一个重取操作，其实用较少的几个重取操作就足够了。修改 lcc 的溢出器以避免无意义的重取。这种改变对你常用的 C 语言程序代码能起多大改进？这样做值得吗？

MIPS R3000 代码的生成

MIPS R3000 体系结构与配套的 R3010 浮点功能部件组成了 RISC，它们各有 32 个 32 位寄存器，另外包含一个 32 位的紧缩指令集和一种寻址模式。MIPS R3000 和 R3010 只能通过显式存取指令来访问存储器。函数调用约定规定通过寄存器传递某些参数。

我们本可以先完整地描述 MIPS 汇编语言，但本章不打算作为 MIPS 汇编语言参考手册来讲，而是从表 16-1 中列举的示例指令入手。这样可以避免一些重复的工作，比如前面已经单独描述了加法指令，在后面讲述 ADDI 规则时又会介绍一次加法指令。在这里描述 MIPS 汇编语言的目的是想帮助读者理解汇编语言的一般语法，即寄存器、地址、操作码和汇编指示等的特征及使用。在此基础上，加上描述规则的模板和文本，以及重复规则的对应构件，我们将能明确对目标机器的认识。

表 16-1　MIPS 汇编器输入样例

汇编指令	意义
move $10, $11	将寄存器 10 的值设置为寄存器 11 的值
subu $10,$11, $12	将寄存器 10 的值设置为寄存器 11 的值减去寄存器 12 的值
subu $10, $11, 12	将寄存器 10 的值设置为寄存器 11 的值减去常量 12
lb $10, 11($12)	将寄存器 10 的值设置为地址为寄存器 12 的值加上寄存器 11 的字节单元的值
sub.d $f12, $f14, $f16	将寄存器 12 的值设置为寄存器 14 的值减去寄存器 16 的值，使用双精度浮点寄存器和运算
sub.s $f12, $f14, $f16	将寄存器 12 的值设置为寄存器 14 的值减去寄存器 16 的值，使用单精度浮点寄存器和运算
b L1	跳转到标号为 L1 的指令处
j $31	跳转到寄存器 31 所指的单元
blt $10, $11, L1	如果寄存器 10 的值小于寄存器 11 的值，则转移到标号 L1 处
.byte 0x20	将内存中下一个字节单元初始化成十六进制数 20

文件 mips.c 包含了所有 MIPS 代码生成器的代码和数据。下面是带接口例程的 Iburg 规范，其中接口例程在文法的后面：

```
⟨mips.md 335⟩=
  %{
  ⟨mips.c macros⟩
  ⟨Iburg prefix 293⟩
  ⟨interface prototypes⟩
  ⟨MIPS prototypes⟩
  ⟨MIPS data 337⟩
  %}
  ⟨terminal declarations 293⟩
  %%
  ⟨shared rules 312⟩
  ⟨MIPS rules 339⟩
  %%
  ⟨MIPS functions 337⟩
  ⟨MIPS interface definition 336⟩
```

最后的程序段（接口定义）配置前端，并指向后端的 MIPS 的代码和数据。大多数目标机器只有一个接口记录，但是 MIPS 结构的机器能进行高位优先或低位优先两种配置，所以 lcc 需要两个接口记录来对应这两种配置：

```
⟨MIPS interface definition 336⟩= 335
  Interface mipsebIR = {
     ⟨MIPS type metrics 336⟩
     0, /* little_endian */
     ⟨shared interface definition 336⟩
  }, mipselIR = {
     ⟨MIPS type metrics 336⟩
     1, /* little_endian */
     ⟨shared interface definition 336⟩
  };
```

数据设备公司（Digital Equipment Company）采用 Ultrix 操作系统，采取低位优先配置并使用 mipselIR。SGI 公司采用 IRIX 操作系统，采取高位优先配置并使用 mipsebIR。两个系统共享同一种类型度量：

```
⟨MIPS type metrics 336⟩= 336
  1, 1, 0,  /* char */
  2, 2, 0,  /* short */
  4, 4, 0,  /* int */
  4, 4, 1,  /* float */
  8, 8, 1,  /* double */
  4, 4, 0,  /* T * */
  0, 1, 0,  /* struct */
```

它们还共享其余的接口记录：

```
⟨shared interface definition 336⟩= 336
  0,  /* mulops_calls */
  0,  /* wants_callb */
  1,  /* wants_argb */
  1,  /* left_to_right */
  0,  /* wants_dag */
  ⟨interface routine names⟩
  0, 0, 0, stabinit, stabline, stabsym, 0,
  {
     4, /* max_unaligned_load */
     ⟨Xinterface initializer 277⟩
  }
```

我们忽略了一些符号表处理程序。像许多编译器一样，lcc 假设所有用于调试程序的数据都能使用汇编指示符编码。MIPS 编译器采用这种方式对文件名和程序行号编码，但有关标识符的类型和位置的信息被编码后放入另外的文件，lcc 并不发送该文件。这样，MIPS 的调试程序能够准确地报告 lcc 生成的可执行文件中的错误出现的位置，但却不能报告或改变标识符的值；参见练习 16.5。

16.1 寄存器

MIPS R3000 处理器内含 32 个 32 位寄存器，在汇编程序中用 $i 表示。MIPS R3010 的浮点协处理器在此基础上又添加了 32 个 32 位寄存器，用 $f6 表示，它们通常当成 16 个 64 位寄存器使用，这些 64 位的寄存器用偶数号码标识，例如 $f2。

在硬件上通常存在一些限制，比如寄存器 $0 的值恒为 0，跳转链接指令的返回地址总是存入 $31。MIPS 结构的 lcc 还必须遵守更多其他编译器使用的约定，以便共享标准库和调试程序。表 16-2 列举了这些约定。

表 16-2 MIPS 寄存器约定

寄存器	用途
$0	0，并保持不变
$1	为汇编器保留
$2 - $3	函数返回值
$4 - $7	开头的各个过程参数
$8 - $15	临时寄存器
$16 - $23	寄存器变量
$24 - $25	临时寄存器
$26 - $27	为操作系统保留
$28	全局指针，也称为 $gp
$29	栈指针，也称为 $sp
$30	寄存器变量
$31	过程返回地址
$f0 - $f2	函数返回值
$f4 - $f10	临时寄存器
$f12 - $f14	开头的两个浮点参数
$f16 - $f18	临时寄存器
$f20 - $f30	寄存器变量

汇编程序预留 $1 以实现伪指令。例如，在地址计算中，硬件只允许 16 位的偏移量，但汇编程序允许 32 位的偏移量，其做法是在原指令上插入额外的指令，利用 $1 形成大于 16 位的偏移量。lcc 使用了一些伪指令，但为了简化直接产生二进制目标代码的过程，它也舍弃了许多伪指令。

按照约定，返回值存入寄存器 $2 - $3 和 $f0 - $f2。但 lcc 只用到每个寄存器的前半部分字节，后半部分专门存储 Fortran 程序语言的复数算术类型。虽然 C 语言没有复数算术类型，但 C 编译器为了能与 Fortran 代码进行互操作，也遵循这种约定。

progend 在 MIPS 目标机器中没有作用。progbeg 按表 16-2 的约定初始化寄存器分配器的数据结构。

```
⟨MIPS functions 337⟩=                                          339  335
  static void progbeg(argc, argv) int argc; char *argv[]; {
     int i;

     ⟨shared progbeg 290⟩
     print(".set reorder\n");
     ⟨parse -G flag 358⟩
     ⟨initialize MIPS register structures 338⟩
  }
```

progbeg 首先产生一个“无害”的指示符——MIPS 汇编程序忽略空输入，接着分析与目标机器相关的标记，然后初始化寄存器符号组成的向量。

```
⟨MIPS data 337⟩=                                               338  335
  static Symbol ireg[32], freg2[32], d6;
```

数组 ireg 的每个元素表示一个通型寄存器，数组 freg2 的每个元素表示由相邻浮点寄存器组成的寄存器对。d6 表示寄存器对 $6-$7。

在 MIPS 结构的机器中，每种类型实际上只有 31 个寄存器对，但数组声明却提供了 32 个，这样可以保证 askreg 中的循环边界的正确性。

```
⟨initialize MIPS register structures 338⟩=                    338  337
  for (i = 0; i < 31; i += 2)
     freg2[i] = mkreg("%d", i, 3, FREG);
  for (i = 0; i < 32; i++)
     ireg[i]  = mkreg("%d", i, 1, IREG);
  ireg[29]->x.name = "sp";
  d6 = mkreg("6", 6, 3, IREG);
```

mkreg 为每个寄存器赋予一个数字作为名字。为了便于记忆，栈指针 $29 重命名为 $sp。汇编程序没有使用寄存器 $gp，否则还要对 $28 重命名。

rmap 存储通配符，这些通配符用于标识各种类型的默认寄存器类：

```
⟨initialize MIPS register structures 338⟩+=                   338 338  337
  rmap[C] = rmap[S] = rmap[P] = rmap[B] = rmap[U] = rmap[I] =
     mkwildcard(ireg);
  rmap[F] = rmap[D] = mkwildcard(freg2);
```

tmask 标识临时寄存器，vmask 标识寄存器变量：

```
⟨mips.c macros⟩=                                              346
  #define INTTMP 0x0300ff00
  #define INTVAR 0x40ff0000
  #define FLTTMP 0xDDDfDff0
  #define FLTVAR 0xfff00000

⟨initialize MIPS register structures 338⟩+=                   338 338  337
  tmask[IREG] = INTTMP; tmask[FREG] = FLTTMP;
  vmask[IREG] = INTVAR; vmask[FREG] = FLTVAR;
```

ARGB 和 ASGNB 复制数据块时，除了需要一个源地址寄存器和一个目的地址寄存器外，还需要 3 个临时寄存器。$3 专门留做临时寄存器，我们还需要另外找两个，而且这 5 个寄存器必须互不相同。一种解决方法是将源寄存器映射到一个寄存器三元组：$8 用作源寄存器，$9 和 $10 用作临时寄存器。tmpregs 为 blkcopy 列出了 3 个临时寄存器：$3，$9 和 $10。

```
⟨MIPS data 337⟩+=                                             337 338  335
  static int tmpregs[] = {3, 9, 10};
```

blkreg 是源寄存器三元组：

```
⟨MIPS data 337⟩+=                                             338 338  335
  static Symbol blkreg;

⟨initialize MIPS register structures 338⟩+=                   338  337
  blkreg = mkreg("8", 8, 7, IREG);
```

mkreg 的第三个参数是由 3 个 1 组成的寄存器掩码，标识寄存器 $8、$9 和 $10。lcc 以后产生的代码就用 $8 作为源寄存器，$9 和 $10 用作临时寄存器。

target 调用 setreg 来标记来需要使用特殊寄存器的节点，调用 rtarget 以标记需要于节点在特定寄存器中的节点。

```
⟨MIPS functions 337⟩+≡                                  337 339 335
  static void target(p) Node p; {
     switch (p->op) {
     ⟨MIPS target 340⟩
     }
  }
```

如果某条指令改写寄存器，clobber 便调用 spill 预先将这些寄存器的值保存起来，处理完当前操作后再恢复原值。

```
⟨MIPS functions 337⟩+≡                                  339 346 335
  static void clobber(p) Node p; {
     switch (p->op) {
     ⟨MIPS clobber 346⟩
     }
  }
```

上述 target 和 clobber 的分支情况及相关指令将在下一节介绍。

16.2 指令的选取

表 16-3 概括了 MIPS 代码生成器的 lburg 规范中的非终结符，给出了面向 MIPS 的 lcc 树文法的组织概貌。

表 16-3 MIPS 的非终结符

名字	匹配对象
acon	地址常量
addr	针对内存读写指令的地址计算
ar	寄存器中的标号和地址
con	常量
rc	寄存器和常量
rc5	寄存器和常量（取 5 位）
reg	计算结果在寄存器中的运算
stmt	副作用产生的运算

一些汇编程序指令通过后缀来标识指令所操作的数据类型。后缀 s 和 d 分别表示单精度浮点指令和双精度浮点指令，b、h 和 w 分别表示 8 位、16 位和 32 位整型指令。可选后缀 u 表示无符号数指令，如果没有 u，汇编指令在默认情况下为有符号数指令。

常量和标识符在汇编程序中直接表示：

```
⟨MIPS rules 339⟩≡                                       339 335
  acon: con      "%0"
  acon: ADDRGP   "%a"
```

访存指令需要使用硬件的地址计算单元，由硬件对指令域与整型寄存器的值求和。与访存指令对应的汇编程序的语法为常量后面跟着用圆括号括住的寄存器：

```
⟨MIPS rules 339⟩+≡                                      339 340 335
  addr: ADDI(reg,acon)   "%1(%%0)"
  addr: ADDU(reg,acon)   "%1(%%0)"
  addr: ADDP(reg,acon)   "%1(%%0)"
```

地址计算过程中有时用到常量0或寄存器$0作为加数，我们称之为退化求和，这种运算主要提供绝对地址和间接地址。此时，常量0或寄存器$0可能被忽略：

```
(MIPS rules 339)+≡                                    339 340  335
  addr: acon   "%0"
  addr: reg    "($%0)"
```

前面已经提到，硬件只能实现16位偏移量的地址计算，但是汇编程序能够利用寄存器$1和额外的指令综合实现更大的偏移量计算，所以lcc可以不受硬件限制，至少在MIPS机器上是这样。

ADDRFP和ADDRLP给栈指针加上一个常量偏移量：

```
(MIPS rules 339)+≡                                    340 340  335
  addr: ADDRFP   "%a+%F($sp)"
  addr: ADDRLP   "%a+%F($sp)"
```

%a产生p->syms[0]->x.name，%F产生帧的大小。例程开始执行时，$sp减去帧的大小，于是%F($sp)重设为$sp的初始值。局部变量的偏移量为负，其地址小于$sp的初始值。形参的偏移量为正，其地址大于$sp的初始值。在调用程序的帧中，16.3节对此有详细阐述。

addr说明了14.10节中提到的一些指导原则，其中包括寻址模式的规则、退化和规则、等价操作符的复制规则，以及普通运算的特殊操作的实现规则。为了简单起见，规范并未在上述每个规则中都使用addr。

伪指令la执行地址运算，并把计算后的结果存入寄存器。例如，指令la $2,x($4)，表示寄存器$4的内容与地址x相加，结果存入$2。

```
(MIPS rules 339)+≡                                    340 340  335
  reg: addr   "la $%c,%0\n"   1
```

%c产生p->syms[RX]->x.name。con表示addr，当lcc需要读取常量放入寄存器时，用到指令la。因为$0总为0，我们无须用指令计算0值：

```
(MIPS rules 339)+≡                                    340 341  335
  reg: CNSTC   "# reg\n"   range(a, 0, 0)
  reg: CNSTS   "# reg\n"   range(a, 0, 0)
  reg: CNSTI   "# reg\n"   range(a, 0, 0)
  reg: CNSTU   "# reg\n"   range(a, 0, 0)
  reg: CNSTP   "# reg\n"   range(a, 0, 0)
```

我们知道，估算代价表达式时，a代表已被标记的节点。这里a表示常量值，我们需要测试a是否为0，如果为0，target为这些节点返回$0：

```
(MIPS target 340)≡                                    345  339
  case CNSTC: case CNSTI: case CNSTS: case CNSTU: case CNSTP:
     if (range(p, 0, 0) == 0) {
        setreg(p, ireg[0]);
        p->x.registered = 1;
     }
     break;
```

分配$0是没有意义的，所以target只是标记节点以防为其分配寄存器。

指令1和s分别表示读取存储器和存入存储器。这些指令包括类型后缀、整型寄存器和addr。例如，指令sw $4，x表示把$4中的32位整数存入存储单元x。sb和sh对寄存器的低8

位和低 16 位也做类似处理。lb、lh 和 lw 是 sb、sh 和 sw 的逆过程，表示从存储器中读取 8 位、16 位和 32 位的值：

```
⟨MIPS rules 339⟩+≡                                          340 341 335
  stmt: ASGNC(addr,reg)  "sb $%1,%0\n"  1
  stmt: ASGNS(addr,reg)  "sh $%1,%0\n"  1
  stmt: ASGNI(addr,reg)  "sw $%1,%0\n"  1
  stmt: ASGNP(addr,reg)  "sw $%1,%0\n"  1
  reg:  INDIRC(addr)     "lb $%c,%0\n"  1
  reg:  INDIRS(addr)     "lh $%c,%0\n"  1
  reg:  INDIRI(addr)     "lw $%c,%0\n"  1
  reg:  INDIRP(addr)     "lw $%c,%0\n"  1
```

lb 和 lh 用符号位填充寄存器的高位部分，它们实现了类型转换 CVCI 和 CVSI。lbu 和 lhu 用零值填充寄存器的高位部分，它们实现了类型转换 CVCU 和 CVSU：

```
⟨MIPS rules 339⟩+≡                                          341 341 335
  reg: CVCI(INDIRC(addr))  "lb $%c,%0\n"  1
  reg: CVSI(INDIRS(addr))  "lh $%c,%0\n"  1
  reg: CVCU(INDIRC(addr))  "lbu $%c,%0\n" 1
  reg: CVSU(INDIRS(addr))  "lhu $%c,%0\n" 1
```

这些规则同时也说明了 14.10 节的另一个指导原则：若指令需要计算多个中间语言的操作码，则写一条能够匹配多个节点的规则。

l. 和 s. 分别表示读取浮点值和存入浮点值。所有的浮点指令都以点号分隔操作码与类型后缀：

```
⟨MIPS rules 339⟩+≡                                          341 341 335
  reg:  INDIRD(addr)     "l.d $f%c,%0\n"  1
  reg:  INDIRF(addr)     "l.s $f%c,%0\n"  1
  stmt: ASGND(addr,reg)  "s.d $f%1,%0\n"  1
  stmt: ASGNF(addr,reg)  "s.s $f%1,%0\n"  1
```

整数乘法指令需要两个源寄存器，计算结果放入一个目的寄存器。左右操作数跟在目的寄存器之后。例如，div$4，$5，$6 表示 $5 除以 $6，结果存入 $4。

```
⟨MIPS rules 339⟩+≡                                          341 341 335
  reg: DIVI(reg,reg)  "div $%c,$%0,$%1\n"   1
  reg: DIVU(reg,reg)  "divu $%c,$%0,$%1\n"  1
  reg: MODI(reg,reg)  "rem $%c,$%0,$%1\n"   1
  reg: MODU(reg,reg)  "remu $%c,$%0,$%1\n"  1
  reg: MULI(reg,reg)  "mul $%c,$%0,$%1\n"   1
  reg: MULU(reg,reg)  "mul $%c,$%0,$%1\n"   1
```

其余的二元整型指令都具有立即数形式，这种形式的右操作数通常是一个常量指令域：

```
⟨MIPS rules 339⟩+≡                                          341 342 335
  rc:  con              "%0"
  rc:  reg              "$%0"

  reg: ADDI(reg,rc)     "addu $%c,$%0,%1\n"  1
  reg: ADDP(reg,rc)     "addu $%c,$%0,%1\n"  1
  reg: ADDU(reg,rc)     "addu $%c,$%0,%1\n"  1
  reg: BANDU(reg,rc)    "and $%c,$%0,%1\n"   1
```

```
reg: BORU(reg,rc)   "or $%c,$%0,%1\n"    1
reg: BXORU(reg,rc)  "xor $%c,$%0,%1\n"   1
reg: SUBI(reg,rc)   "subu $%c,$%0,%1\n"  1
reg: SUBP(reg,rc)   "subu $%c,$%0,%1\n"  1
reg: SUBU(reg,rc)   "subu $%c,$%0,%1\n"  1
```

在立即数移位指令中，立即数的取值必须在 0 至 31 之间：

```
(MIPS rules 339)+=                                    341 342   335
  rc5: CNSTI          "%a"                  range(a,0,31)
  rc5: reg            "$%0"

  reg: LSHI(reg,rc5)  "sll $%c,$%0,%1\n"  1
  reg: LSHU(reg,rc5)  "sll $%c,$%0,%1\n"  1
  reg: RSHI(reg,rc5)  "sra $%c,$%0,%1\n"  1
  reg: RSHU(reg,rc5)  "srl $%c,$%0,%1\n"  1
```

对于一元操作指令，操作数只能是寄存器：

```
(MIPS rules 339)+=                                    342 342   335
  reg: BCOMU(reg)  "not $%c,$%0\n"   1
  reg: NEGI(reg)   "negu $%c,$%0\n"  1
  reg: LOADC(reg)  "move $%c,$%0\n"  move(a)
  reg: LOADS(reg)  "move $%c,$%0\n"  move(a)
  reg: LOADI(reg)  "move $%c,$%0\n"  move(a)
  reg: LOADP(reg)  "move $%c,$%0\n"  move(a)
  reg: LOADU(reg)  "move $%c,$%0\n"  move(a)
```

前面提到，move 函数返回值为 1，同时把节点标记为“寄存器－寄存器”的移动，使节点可用于某些优化。

浮点指令也只有寄存器形式：

```
(MIPS rules 339)+=                                    342 342   335
  reg: ADDD(reg,reg)  "add.d $f%c,$f%0,$f%1\n"  1
  reg: ADDF(reg,reg)  "add.s $f%c,$f%0,$f%1\n"  1
  reg: DIVD(reg,reg)  "div.d $f%c,$f%0,$f%1\n"  1
  reg: DIVF(reg,reg)  "div.s $f%c,$f%0,$f%1\n"  1
  reg: MULD(reg,reg)  "mul.d $f%c,$f%0,$f%1\n"  1
  reg: MULF(reg,reg)  "mul.s $f%c,$f%0,$f%1\n"  1
  reg: SUBD(reg,reg)  "sub.d $f%c,$f%0,$f%1\n"  1
  reg: SUBF(reg,reg)  "sub.s $f%c,$f%0,$f%1\n"  1
  reg: LOADD(reg)     "mov.d $f%c,$f%0\n"       move(a)
  reg: LOADF(reg)     "mov.s $f%c,$f%0\n"       move(a)
  reg: NEGD(reg)      "neg.d $f%c,$f%0\n"       1
  reg: NEGF(reg)      "neg.s $f%c,$f%0\n"       1
```

很少有指令专门用于类型转换。CVCI 和 CVSI 通过先左移再右移实现符号扩展。CVCU 和 CVSU 通过对寄存器的高位部分做逻辑与来实现零扩展：

```
(MIPS rules 339)+=                                    342 343   335
  reg: CVCI(reg)  "sll $%c,$%0,24; sra $%c,$%c,24\n"  2
  reg: CVSI(reg)  "sll $%c,$%0,16; sra $%c,$%c,16\n"  2
  reg: CVCU(reg)  "and $%c,$%0,0xff\n"                1
  reg: CVSU(reg)  "and $%c,$%0,0xffff\n"              1
```

这些规则说明了 14.10 节中的另一个指导原则：当没有指令能够直接实现一个操作时，则写一条规则，使用若干指令完成该操作。

剩下的有关整型和指针类型的转换，指令不执行操作。由于前端建立的树不会用到缩小后的值的高位，缩小类型的转换操作（比如 CVIC）不必清除寄存器的高位。如果对现存寄存器进行缩小类型的转换操作，那么共享的规则和下面的规则将不产生代码：

```
(MIPS rules 339)+≡                                          ↑342 343↓ 335
  reg: CVIC(reg)  "%0"  notarget(a)
  reg: CVIS(reg)  "%0"  notarget(a)
  reg: CVUC(reg)  "%0"  notarget(a)
  reg: CVUS(reg)  "%0"  notarget(a)
```

如果指令已定位某个特定寄存器，则一些开销大的规则会生成“寄存器－寄存器”的复制：

```
(MIPS rules 339)+≡                                          ↑343 343↓ 335
  reg: CVIC(reg)  "move $%c,$%0\n"  move(a)
  reg: CVIS(reg)  "move $%c,$%0\n"  move(a)
  reg: CVIU(reg)  "move $%c,$%0\n"  move(a)
  reg: CVPU(reg)  "move $%c,$%0\n"  move(a)
  reg: CVUC(reg)  "move $%c,$%0\n"  move(a)
  reg: CVUI(reg)  "move $%c,$%0\n"  move(a)
  reg: CVUP(reg)  "move $%c,$%0\n"  move(a)
  reg: CVUS(reg)  "move $%c,$%0\n"  move(a)
```

cvid.s 将浮点数转换成双精度数，cvt.s.d 执行相反的操作：

```
(MIPS rules 339)+≡                                          ↑343 343↓ 335
  reg: CVDF(reg)  "cvt.s.d $f%c,$f%0\n"  1
  reg: CVFD(reg)  "cvt.d.s $f%c,$f%0\n"  1
```

cvt.d.w 将整数转换成浮点数，这里的整数必须存在于浮点寄存器中，故 CVID 首先执行 mtc1，把一个整数值从整数单元复制到浮点单元：

```
(MIPS rules 339)+≡                                          ↑343 343↓ 335
  reg: CVID(reg)  "mtc1 $%0,$f%c; cvt.d.w $f%c,$f%c\n"  2
```

CVID 设置了目标寄存器的值两次：第一次设置为未转换的整数值，第二次设置为等价的双精度数值。参见练习 16.6。

trunc.w.d 截取双精度数，把整数结果放入浮点寄存器，所以 trunc.w.d 后面有一个 mfc1，它把浮点单元中的值复制到整数单元，该整数单元已由 CVDI 的使用者事先指定：

```
(MIPS rules 339)+≡                                          ↑343 343↓ 335
  reg: CVDI(reg)  "trunc.w.d $f2,$f%0,$%c; mfc1 $%c,$f2\n"  2
```

trunc.w.d 需要一个临时的浮点寄存器保存转换后的值，这里使用 $f2。调用约定保留 $f2 作为第二个返回寄存器，所以 lcc 不分配 $f2 另做他用。

标号定义时后跟一个冒号：

```
(MIPS rules 339)+≡                                          ↑343 344↓ 335
  stmt: LABELV  "%a:\n"
```

b 指令无条件跳转至固定地址，j 指令无条件跳转至寄存器指定的地址：

```
(MIPS rules 339)+≡                                          343 344 335
  stmt: JUMPV(acon)  "b %0\n"   1
  stmt: JUMPV(reg)   "j $%0\n"  1
```

用分支转移表实现的 switch 语句需要用到 j 指令。其他无条件分支转移都使用 b 指令。

整数条件分支指令首先比较两个寄存器值，如果所示条件满足则进行转移：

```
(MIPS rules 339)+≡                                          344 344 335
  stmt: EQI(reg,reg)  "beq $%0,$%1,%a\n"   1
  stmt: GEI(reg,reg)  "bge $%0,$%1,%a\n"   1
  stmt: GEU(reg,reg)  "bgeu $%0,$%1,%a\n"  1
  stmt: GTI(reg,reg)  "bgt $%0,$%1,%a\n"   1
  stmt: GTU(reg,reg)  "bgtu $%0,$%1,%a\n"  1
  stmt: LEI(reg,reg)  "ble $%0,$%1,%a\n"   1
  stmt: LEU(reg,reg)  "bleu $%0,$%1,%a\n"  1
  stmt: LTI(reg,reg)  "blt $%0,$%1,%a\n"   1
  stmt: LTU(reg,reg)  "bltu $%0,$%1,%a\n"  1
  stmt: NEI(reg,reg)  "bne $%0,$%1,%a\n"   1
```

硬件不能直接实现所有这些指令，但汇编程序进行了补充。例如，有关 GE、GT、LE 和 LT 的硬件指令都假定第二个比较数为 0，但是必要时汇编程序可以计算两个比较操作数的差值并存入 $1，再由硬件实现 $1 与 0 的比较，这样综合伪指令实现上述分支指令。另外的问题是，只有 j 指令接受 32 位的地址，而上述伪指令打破了硬件地址计算的限制。汇编出任意长度的地址。因此，当 lcc 使用伪指令时，它并不知道汇编程序产生什么样的实际指令，所以上述代价估算只是近似的，但无论如何，对于这些中间语言操作，只有一种方法生成代码，因而估算的偏差不会对所产生代码的质量造成影响。

浮点条件分支对一个条件标记进行测试，该条件标记由另一条独立的比较指令设定。例如，当 $f0 中的双精度数小于 $f2 中的双精度数时，指令 c.lt.d $f0，$f2 设置条件标记，并执行 bclt 分支。如果标记为 0，则执行 bclf 分支。

```
(MIPS rules 339)+≡                                          344 344 335
  stmt: EQD(reg,reg)  "c.eq.d $f%0,$f%1; bc1t %a\n"  2
  stmt: EQF(reg,reg)  "c.eq.s $f%0,$f%1; bc1t %a\n"  2
  stmt: LED(reg,reg)  "c.le.d $f%0,$f%1; bc1t %a\n"  2
  stmt: LEF(reg,reg)  "c.le.s $f%0,$f%1; bc1t %a\n"  2
  stmt: LTD(reg,reg)  "c.lt.d $f%0,$f%1; bc1t %a\n"  2
  stmt: LTF(reg,reg)  "c.lt.s $f%0,$f%1; bc1t %a\n"  2
```

浮点比较指令只实现了小于、小于或等于以及等于 3 种比较计算。lcc 通过关系意义转化，再执行 bclf，就可以实现其他的比较运算：

```
(MIPS rules 339)+≡                                          344 345 335
  stmt: GED(reg,reg)  "c.lt.d $f%0,$f%1; bc1f %a\n"  2
  stmt: GEF(reg,reg)  "c.lt.s $f%0,$f%1; bc1f %a\n"  2
  stmt: GTD(reg,reg)  "c.le.d $f%0,$f%1; bc1f %a\n"  2
  stmt: GTF(reg,reg)  "c.le.s $f%0,$f%1; bc1f %a\n"  2
  stmt: NED(reg,reg)  "c.eq.d $f%0,$f%1; bc1f %a\n"  2
  stmt: NEF(reg,reg)  "c.eq.s $f%0,$f%1; bc1f %a\n"  2
```

例如，lcc 不能使用：

```
c.gt.d $f0,$f2
bc1t L
```

而是用下面的指令代替它：

```
c.le.d $f0,$f2
bc1f L
```

jal 指令将程序计数器保存在 $31 中，并跳转至常量指令域或寄存器所示的地址处。

```
(MIPS rules 339)+≡                                    344 345 335
  ar:   ADDRGP    "%a"

  reg:  CALLD(ar)  "jal %0\n"  1
  reg:  CALLF(ar)  "jal %0\n"  1
  reg:  CALLI(ar)  "jal %0\n"  1
  stmt: CALLV(ar)  "jal %0\n"  1
```

CALLV 不产生结果，因而匹配 stmt 而不是 reg。大多数调用都跳转到某个标号继续执行，但像 (*p)( ) 这样的间接调用需要使用寄存器形式：

```
(MIPS rules 339)+≡                                    345 345 335
  ar: reg    "$%0"
```

一些设备驱动程序通常跳到固定数值的地址。jal 要求地址必须是 28 位的：

```
(MIPS rules 339)+≡                                    345 345 335
  ar: CNSTP  "%a"   range(a, 0, 0x0fffffff)
```

如果常量不是 28 位，lcc 只有采用其他代价更高的规则，先将任意一个 32 位的常量载入寄存器，然后通过该寄存器实现间接跳转。另外，MIPS 的汇编程序完成了大多数的范围检查，但有些版本的汇编程序把边界检查留给了编译器来完成。

编译前端和例程 function 与 target 协同将返回值存入返回寄存器，并把地址存入程序计数器，所以 RET 节点不必生成任何代码：

```
(MIPS rules 339)+≡                                    345 346 335
  stmt: RETD(reg)  "# ret\n"  1
  stmt: RETF(reg)  "# ret\n"  1
  stmt: RETI(reg)  "# ret\n"  1
```

CALLD 和 CALLF 产生 $f0，CALLI 产生 $2。每个 RET 的子节点执行计算，并将计算结果存入相应的寄存器。

```
(MIPS target 340)+≡                                   340 347 339
  case CALLD: case CALLF: setreg(p, freg2[0]);      break;
  case CALLI:             setreg(p, ireg[2]);       break;
  case RETD: case RETF:   rtarget(p, 0, freg2[0]); break;
  case RETI:              rtarget(p, 0, ireg[2]);   break;
```

setreg 为正在操作的节点设置结果寄存器，rtarget 为该节点的子节点设置结果寄存器。但如果在子节点上直接调用 setreg，则很可能会破坏一些值，所以还要使用 rtarget。详细描述参见有关 setreg 的资料。

临时和返回寄存器不能跨过程调用来保留，所以除了调用者本身使用的结果寄存器外，其他任何活动寄存器必须溢出和重取。

```
⟨mips.c macros⟩+≡                                                    338
  #define INTRET 0x00000004
  #define FLTRET 0x00000003

⟨MIPS clobber 346⟩≡                                                  339
  case CALLD: case CALLF:
     spill(INTTMP | INTRET, IREG, p);
     spill(FLTTMP,          FREG, p);
     break;
  case CALLI:
     spill(INTTMP,          IREG, p);
     spill(FLTTMP | FLTRET, FREG, p);
     break;
  case CALLV:
     spill(INTTMP | INTRET, IREG, p);
     spill(FLTTMP | FLTRET, FREG, p);
     break;
```

浮点值存入双精度寄存器 $f0 中返回，其他类型的值存入 $2 中返回。target 和 clobber 配合使用。例如 CALLI，它在 target 中生成寄存器 $2，函数调用前，clobber 先调用 spill，保存所有其他由调用程序保存的寄存器，函数调用完成后再恢复这些寄存器的原值。

参数传递的规则需要 target 和 emit2 的协作完成：

```
⟨MIPS rules 339⟩+≡                                          345 348  335
  stmt: ARGD(reg)  "# arg\n"  1
  stmt: ARGF(reg)  "# arg\n"  1
  stmt: ARGI(reg)  "# arg\n"  1
  stmt: ARGP(reg)  "# arg\n"  1

⟨MIPS functions 337⟩+≡                                      339 346  335
  static void emit2(p) Node p; {
     int dst, n, src, ty;
     static int ty0;
     Symbol q;

     switch (p->op) {
     ⟨MIPS emit2 348⟩
     }
  }
```

一般情况下，根据 MIPS 的调用约定，将参数的头 4 个字传入寄存器 $4-$7（包括确保对齐的间隙字节）。但是，如果第一个参数是浮点型或双精度型，则通过 $f12 传递；如果第二个参数也是一个浮点型或双精度型，则通过 $f14 传递。argreg 实现了上述规则：

```
⟨MIPS functions 337⟩+≡                                      346 347  335
  static Symbol argreg(argno, offset, ty, ty0)
  int argno, offset, ty, ty0; {
     if (offset > 12)
        return NULL;
     else if (argno == 0 && (ty == F || ty == D))
        return freg2[12];
```

```
    else if (argno == 1 && (ty == F || ty == D)
    && (ty0 == F || ty0 == D))
       return freg2[14];
    else if (argno == 1 && ty == D)
       return d6;  /* Pair! */
    else
       return ireg[(offset/4) * 4];
}
```

argno 表示参数个数，offset 和 ty 分别表示参数的偏移量和类型，ty0 表示第一个参数的类型，它影响第二个参数的存储位置。如果参数通过某个寄存器传递，那么 argreg 返回该寄存器，否则返回空。

gen 调用 doarg 来计算 argreg 需要的 argno 和 offset：

```
(MIPS functions 337)+≡                                    346 349  335
  static void doarg(p) Node p; {
     static int argno;
     int size;

     if (argoffset == 0)
        argno = 0;
     p->x.argno = argno++;
     size = p->syms[1]->u.c.v.i < 4 ? 4 : p->syms[1]->u.c.v.i;
     p->syms[2] = intconst(mkactual(size,
        p->syms[0]->u.c.v.i));
  }
```

docall 在每个 CALL 开始时清除 argoffset，所以若 doarg 遇到 argoffset 等于 0，它将重设静态参数计数器。mkactual 使用参数的大小和对齐字节数（必要时对 4 进行舍入计算，将较小参数加宽），并返回参数的偏移量。

target 利用 argreg 和 rtarget 计算 ARG 的子节点（如果有子节点），将其存入参数寄存器：

```
(MIPS target 340)+≡                                       345 348  339
  case ARGD: case ARGF: case ARGI: case ARGP: {
     static int ty0;
     int ty = optype(p->op);
     Symbol q;

     q = argreg(p->x.argno, p->syms[2]->u.c.v.i, ty, ty0);
     if (p->x.argno == 0)
        ty0 = ty;
     if (q &&
     !((ty == F || ty == D) && q->x.regnode->set == IREG))
        rtarget(p, 0, q);
     break;
     }
```

该程序段也记录第一个参数的类型，有助于为后面的参数选择寄存器。若参数是浮点数却通过整数寄存器传递，则程序不执行 rtarget。lcc 规定浮点操作码只产生浮点寄存器，所以浮点数存入整数寄存器之前必须先做整型转换。emit2 执行这一过程并处理各种访存的参数：

```
⟨MIPS emit2 348⟩≡                                          348 346
  case ARGD: case ARGF: case ARGI: case ARGP:
     ty = optype(p->op);
     if (p->x.argno == 0)
        ty0 = ty;
     q = argreg(p->x.argno, p->syms[2]->u.c.v.i, ty, ty0);
     src = getregnum(p->x.kids[0]);
     if (q == NULL && ty == F)
        print("s.s $f%d,%d($sp)\n", src, p->syms[2]->u.c.v.i);
     else if (q == NULL && ty == D)
        print("s.d $f%d,%d($sp)\n", src, p->syms[2]->u.c.v.i);
     else if (q == NULL)
        print("sw $%d,%d($sp)\n", src, p->syms[2]->u.c.v.i);
     else if (ty == F && q->x.regnode->set == IREG)
        print("mfc1 $%d,$f%d\n", q->x.regnode->number, src);
     else if (ty == D && q->x.regnode->set == IREG)
        print("mfc1.d $%d,$f%d\n", q->x.regnode->number, src);
     break;
```

如果 argreg 返回空指针，则调用程序通过存储器传递参数。emit2 使用 doarg 计算的偏移量存储参数值。最后两个 if 条件语句生成的指令实现了在整数寄存器中传递浮点参数。mfc1 x,y 表示从浮点寄存器 y 中复制一个单精度值到整数寄存器 x。mfc1.d 表示对双精度值的类似操作，操作目标是一个寄存器对。

emit2 与 target 合作实现块的复制：

```
⟨MIPS rules 339⟩+≡                                         346 335
  stmt: ARGB(INDIRB(reg))        "# argb %0\n"       1
  stmt: ASGNB(reg,INDIRB(reg))   "# asgnb %0 %1\n"   1
```

下面是 emit2 处理 ASGNB 分支的情况：该分支首先设置全局变量记录源块和目的块的对齐字节数，然后调用 blkcopy 完成剩下的工作：

```
⟨MIPS emit2 348⟩+≡                                         348 349 346
  case ASGNB:
     dalign = salign = p->syms[1]->u.c.v.i;
     blkcopy(getregnum(p->x.kids[0]), 0,
        getregnum(p->x.kids[1]), 0,
        p->syms[0]->u.c.v.i, tmpregs);
     break;
```

图 13-4 的调用序列从 ASGNB 分支开始。tmpregs 中存有 3 个临时寄存器的编号，它们组成了一个寄存器三元组，progbeg 将三元组分配给 blkreg。ARGB 和 ASGNB 将它们的源地址寄存器定位到预留的 blkreg 中：

```
⟨MIPS target 340⟩+≡                                        347 339
  case ASGNB: rtarget(p->kids[1], 0, blkreg); break;
  case ARGB:  rtarget(p->kids[0], 0, blkreg); break;
```

由于插进来的子节点是形式上的 INDIRB，所以源地址来自于一个孙节点。emit2 处理 ARGB 的分支与处理 ASGNB 的分支类似。

```
(MIPS emit2 348)+≡                                    348 346
  case ARGB:
     dalign = 4;
     salign = p->syms[1]->u.c.v.i;
     blkcopy(29, p->syms[2]->u.c.v.i,
        getregnum(p->x.kids[0]), 0,
        p->syms[0]->u.c.v.i, tmpregs);
     n   = p->syms[2]->u.c.v.i + p->syms[0]->u.c.v.i;
     dst = p->syms[2]->u.c.v.i;
     for ( ; dst <= 12 && dst < n; dst += 4)
        print("lw $%d,%d($sp)\n", (dst/4)+4, dst);
     break;
```

由代码可知，分配给输出参数的栈空间总是按照 4 的倍数对齐，所以这里的 dalign 与前面的不同，blkcopy 和它的辅助程序经常这样用。因为 $sp 是目标基准寄存器，所以 blkcopy 的第一个参数是 29，第二个参数是目的块的栈偏移量，偏移量由 doarg 计算。若 ARGB 覆盖参数的头 4 个字，那么程序中 for 循环语句将覆盖的部分复制到相应的参数寄存器，这样做才符合调用规则。

16.3 函数的实现

编译前端调用 local 通知每个新的局部变量：

```
(MIPS functions 337)+≡                                347 350 335
  static void local(p) Symbol p; {
     if (askregvar(p, rmap[ttob(p->type)]) == 0)
        mkauto(p);
  }
```

与目标机器无关的例程做了大部分工作。如果当前有合适的可分配的寄存器，askregvar 将分配一个寄存器给 p，否则 mkauto 指派一个栈帧偏移量给 p。图 16-1 显示了 MIPS 栈帧的结构布局。

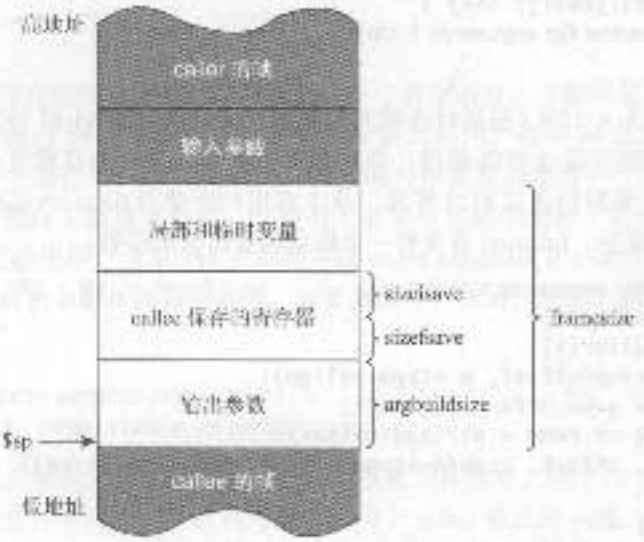


图 16-1 MIPS 栈帧

前端调用 function 通知新的例程。大部分的后端程序由 function 驱动。function 调用 gencode，gencode 调用 gen，再由 gen 调用标记程序（labeller）、化简程序（reducer）、线性化程序

〈linearizer〉和寄存器分配器〈register allocator〉。function 还调用前端的 emitcode，emitcode 再调用后端的代码产生器。

```
⟨MIPS functions 337⟩+≡                                    ↑349 355  335
  static void function(f, caller, callee, ncalls)
  Symbol f, callee[], caller[]; int ncalls; {
     int i, saved, sizefsave, sizeisave, varargs;
     Symbol r, argregs[4];

     ⟨MIPS function 350⟩
  }
```

前端向 function 传递 3 个参数，分别是：代表例程的符号、表示调用程序和被调用程序的参数视图符号向量，以及记录例程引发的调用次数的计数器。function 首先释放所有寄存器，然后清除用于跟踪帧并跟踪输出参数的复制区域的变量：

```
⟨MIPS function 350⟩≡                                      350  350
  ⟨clear register state 319⟩
  offset = maxoffset = maxargoffset = 0;
```

接下来，function 判断例程是否带可变参数，因为参数是否可变将关系到后面生成的代码：

```
⟨MIPS function 350⟩+≡                                     ↑350 350  350
  for (i = 0; callee[i]; i++)
     ;
  varargs = variadic(f->type)
     || i > 0 && strcmp(callee[i-1]->name, "va_alist") == 0;
```

MIPS 体系结构的机器规定，函数必须要么有原型，要么最后的参数应该命名为 va_alist。function 根据这一规定来确定某些输入参数的位置：

```
⟨MIPS function 350⟩+≡                                     ↑350 352  350
  for (i = 0; callee[i]; i++) {
     ⟨assign location for argument i 350⟩
  }
```

前面曾提到，参数的头 4 个字（包括对齐填充的间隙）通过寄存器 $4-$7 传递，但如果第一个参数是浮点或双精度，则应通过 $f12 传递；如果第二个参数是浮点或双精度，那么第二个参数通过 $f14 传递，第一个参数仍通过 $f12 传递。这个调用约定使得 function 实现起来很复杂，特别是上面的循环体非常复杂。function 首先将一个栈偏移量指派给参数：

```
⟨assign location for argument i 350⟩≡                     351  350
  Symbol p = callee[i];
  Symbol q = caller[i];
  offset = roundup(offset, q->type->align);
  p->x.offset = q->x.offset = offset;
  p->x.name = q->x.name = stringd(offset);
  r = argreg(i, offset, ttob(q->type), ttob(caller[0]->type));
  if (i < 4)
     argregs[i] = r;
  offset = roundup(offset + q->type->size, 4);
```

程序也在栈中给那些通过寄存器传递并驻留在寄存器中的参数预留了存储位置。事实上，argreg 通过偏移量就能确定哪个寄存器中有参数。argregs[i] 记录了 argreg 关于第 i 个参数的结果，以备后用。因为代码通过间接寻址访问变量例程的所有参数，所以变参例程所有的参数均存在栈内。

```
⟨assign location for argument i 350⟩+≡                    350 351  350
  if (varargs)
     p->sclass = AUTO;
```

对于一个通过寄存器传递的参数，若例程不包含可能重写该寄存器的调用，这时参数只要满足下面 3 个条件就可以驻留在寄存器中：(1) 不是一个结构，(2) 不是间接访问，(3) 不是通过整型寄存器传递的浮点数。

```
⟨leave argument in place? 351⟩≡                                  351
  r && ncalls == 0 &&
  !isstruct(q->type) && !p->addressed &&
  !(isfloat(q->type) && r->x.regnode->set == IREG)
⟨assign location for argument i 350⟩+≡                    351 351  350
  else if (⟨leave argument in place? 351⟩) {
     p->sclass = q->sclass = REGISTER;
     askregvar(p, r);
     q->x = p->x;
     q->type = p->type;
  }
```

这里 r 不可能因其他目的而被分配，所以 askregvar 总能执行成功。代码中有一个隐含的断言来确认这个判断。参数的 type 域和 sclass 域保持一致能避免前端生成复制或转换参数的代码。最后还必须给那些先前没有分配或不能驻留在原寄存器中的参数分配寄存器：

```
⟨assign location for argument i 350⟩+≡                          351  350
  else if (⟨copy argument to another register? 351⟩) {
     p->sclass = q->sclass = REGISTER;
     q->type = p->type;
  }
```

当且仅当参数通过寄存器传递，并且必须移到另一个寄存器时，才能满足上面代码的 if 条件。例如，如果某个参数通过 $4 传递，但是例程要执行新的调用，这时 $4 必须存储输出参数。如果输入参数只用到一个寄存器，那么上述代码将把输入参数复制到另一个寄存器中。较特殊的情况是，浮点参数可能存入整数寄存器，而前端不能表示这样的操作，这时上述代码根据 type 和 scalss 通知前端不生成代码，由第 354 页的程序段 <save argument in a register> 生成复制指令。

要满足程序最后的 else-if 语句的条件，需要测试 3 个从句。首先，askregvar 必须能够为该参数分配寄存器：

```
⟨copy argument to another register? 351⟩≡                       352  351
  askregvar(p, rmap[ttob(p->type)])
```

如果分配失败，参数只有存入存储器。如果参数不在栈里，程序段 <save argument in stack>（见第 354 页）会把它压入栈中。在这种情况下，两个 sclass 域已经一致，但我们不希望两个 field 域一致，因为这样可能需要一个转换才行。例如，在高位优先中，一个新风格函数的字符参数需要转换，它只能以整数方式传递，所以它的值存入参数字中最低有效位，但是它在后面必须以字符的形式访问，所以在高位优先的机器中，它的值必须移到字的最高有效位。

第二个条件进一步确认参数已存入寄存器：

```
⟨copy argument to another register? 351⟩+≡                    351 352  351
  && r != NULL
```

如果条件不满足，这时参数通过存储器传递，需要将该参数读取到由 askregvar 确定的寄存器中。例如，这种参数可能是 5 个整型参数中的最后一个，这意味着它将通过存储器传递。如果这个参数使用频繁，那么还应该将其存入寄存器。askregvar 设置 p->sclass 为 REGISTER，而 q->sclass 绝不会是 REGISTER。因此，如果条件不满足且 p->sclass 与 q->sclass 的值不同，前端就会生成读取类作。

第三个条件（即最后一个条件）是确认不需要转换：

```
⟨copy argument to another register? 351⟩+≡                    352  351
  && (isint(p->type) || p->type == q->type)
```

例如，如果 q（调用程序）是双精度数，p 是浮点数；那么必须用到 CVDF。这时 p 和 q 的 sclass 和 type 不相等，条件的不满足导致前端生成类型转换操作。

为所有参数指派存储位置后，function 调用 gencode 为例程体选取代码并分配寄存器：

```
⟨MIPS function 350⟩+≡                                         350 352  350
  offset = 0;
  gencode(caller, callee);
```

gencode 返回时，usedmask 标识了例程所用的寄存器。function 为 usedmask 再加上保存返回地址的寄存器（除非该例程没有包含调用），并且删除调用程序保存的寄存器。

```
⟨MIPS function 350⟩+≡                                         352 352  350
  if (ncalls)
    usedmask[IREG] |= ((unsigned)1)<<31;
  usedmask[IREG] &= 0xc0ff0000;
  usedmask[FREG] &= 0xfff00000;
```

然后 function 计算参数建立区的大小：

```
⟨MIPS function 350⟩+≡                                         352 352  350
  maxargoffset = roundup(maxargoffset, 4);
  if (maxargoffset && maxargoffset < 16)
    maxargoffset = 16;
```

调用约定要求输出参数块的大小必须为 4 的倍数，而且参数块若不为空，则至少为 16 个字节。function 计算帧以及帧内部用来存放浮点寄存器和整数寄存器的块的大小。

```
⟨MIPS function 350⟩+≡                                         352 353  350
  sizefsave = 4*bitcount(usedmask[FREG]);
  sizeisave = 4*bitcount(usedmask[IREG]);
  framesize = roundup(maxargoffset + sizefsave
    + sizeisave + maxoffset, 8);
```

bitcount 计算无符号整数中为 1 的位的数目。图 16-1 说明了这些数值的意义。我们遵循的约定保证了栈按双字对齐。

现在 function 得到了开始产生例程所需要的数据。函数头代码转到代码段，确保按字对齐，并生成了 MIPS 例程的起始代码的样板：

```
⟨MIPS function 350⟩+≡                                    352 353  350
  segment(CODE);
  print(".align 2\n");
  print(".ent %s\n", f->x.name);
  print("%s:\n", f->x.name);
  i = maxargoffset + sizefsave - framesize;
  print(".frame $sp,%d,$31\n", framesize);
  if (framesize > 0)
     print("addu $sp,$sp,%d\n", -framesize);
  if (usedmask[FREG])
     print(".fmask 0x%x,%d\n", usedmask[FREG], i - 8);
  if (usedmask[IREG])
     print(".mask 0x%x,%d\n",  usedmask[IREG],
        i + sizeisave - 4);
```

lcc 的代码只使用标号和 addu，addu 为例程分配帧。其余的指示符是为满足其他程序的需要（如调试程序和性能分析）而描述例程。.ent 通知程序入口点；.frame 声明栈指针、帧大小和返回地址寄存器；.fmask 和 .mask 分别标识被保存的寄存器及其在栈中的位置。

接下来的函数头代码保存由被调用程序所保存的寄存器，参见表 16-2：

```
⟨MIPS function 350⟩+≡                                    353 353  350
  saved = maxargoffset;
  for (i = 20; i <= 30; i += 2)
     if (usedmask[FREG]&(3<<i)) {
        print("s.d $f%d,%d($sp)\n", i, saved);
        saved += 8;
     }
  for (i = 16; i <= 31; i++)
     if (usedmask[IREG]&(1<<i)) {
        print("sw $%d,%d($sp)\n", i, saved);
        saved += 4;
     }
```

然后保存通过寄存器传递的参数：

```
⟨MIPS function 350⟩+≡                                    353 353  350
  for (i = 0; i < 4 && callee[i]; i++) {
     r = argregs[i];
     if (r && r->x.regnode != callee[i]->x.regnode) {
        ⟨save argument i 354⟩
     }
  }
```

对于变参例程来说，由于每次对它的调用所使用的参数个数都可能不同，所以 lcc 还应保存其余的形型参数寄存器：

```
⟨MIPS function 350⟩+≡                                    353 355  350
  if (varargs && callee[i-1]) {
     i = callee[i-1]->x.offset + callee[i-1]->type->size;
     for (i = roundup(i, 4)/4; i <= 3; i++)
        print("sw $%d,%d($sp)\n", i + 4, framesize + 4*i);
     }
```

该循环从它前面的循环没有处理的参数开始执行，直到存储完最后一个整型参数寄存器 $7。

对于非变参例程来说，函数头代码只保存那些被使用但不能继续驻留的参数寄存器：

```
⟨save argument 1354⟩=                                          353
  Symbol out = callee[i];
  Symbol in  = caller[i];
  int rn = r->x.regnode->number;
  int rs = r->x.regnode->set;
  int tyin = ttob(in->type);

  if (out->sclass == REGISTER
  && (isint(out->type) || out->type == in->type)) {
     ⟨save argument in a register 354⟩
  } else {
     ⟨save argument in stack 354⟩
  }
```

函数头代码可区分通过寄存器传递的参数与通过存储器传递的参数。&& 之后的子句匹配前面第 351 页的 <leave argument in place?> 中的条件，它决定了在这里应该生成什么代码。

如果通过寄存器传递的参数已经分配到了某个寄存器，但不能继续驻留在该寄存器中，那么 function 将产生代码把输入参数寄存器的值复制到其他寄存器中：

```
⟨save argument in a register 354⟩=                             354
  int outn = out->x.regnode->number;
  if (rs == FREG && tyin == D)
     print("mov.d $f%d,$f%d\n", outn, rn);
  else if (rs == FREG && tyin == F)
     print("mov.s $f%d,$f%d\n", outn, rn);
  else if (rs == IREG && tyin == D)
     print("mtc1.d $%d,$f%d\n", rn,   outn);
  else if (rs == IREG && tyin == F)
     print("mtc1 $%d,$f%d\n",   rn,   outn);
  else
     print("move $%d,$%d\n",    outn, rn);
```

如果参数已被指派到存储器中，那么函数头代码会将参数写入程序的活动记录中：

```
⟨save argument in stack 354⟩=                                  354
  int off = in->x.offset + framesize;
  if (rs == FREG && tyin == D)
     print("s.d $f%d,%d($sp)\n", rn, off);
  else if (rs == FREG && tyin == F)
     print("s.s $f%d,%d($sp)\n", rn, off);
  else {
     int i, n = (in->type->size + 3)/4;
     for (i = rn; i < rn+n && i <= 7; i++)
        print("sw $%d,%d($sp)\n", i, off + (i-rn)*4);
  }
```

最后的 else 子句内的 for 循环通常只执行一次，保存一个整型参数，但它也可以处理通过整型寄存器传递的浮点数。for 循环还能推广到处理双精度和结构参数，这些参数一般占用多个整型寄存器。当循环处理完所有参数或参数寄存器后即终止，不论二者哪个先处理完。

在产生函数头代码之后，function 产生程序体：

```
⟨MIPS function 350⟩+≡                                    353 355 350
  emitcode();
```

函数尾代码重新读取由被调用程序保存的寄存器。首先载入浮点寄存器：

```
⟨MIPS function 350⟩+≡                                    355 355 350
  saved = maxargoffset;
  for (i = 20; i <= 30; i += 2)
    if (usedmask[FREG]&(3<<i)) {
      print("l.d $f%d,%d($sp)\n", i, saved);
    saved += 8;
  }
```

然后是通用寄存器：

```
⟨MIPS function 350⟩+≡                                    355 355 350
  for (i = 16; i <= 31; i++)
    if (usedmask[IREG]&(1<<i)) {
      print("lw $%d,%d($sp)\n", i, saved);
      saved += 4;
    }
```

再把帧弹出栈：

```
⟨MIPS function 350⟩+≡                                    355 355 350
  if (framesize > 0)
    print("addu $sp,$sp,%d\n", framesize);
```

并返回：

```
⟨MIPS function 350⟩+≡                                    355 350
  print("j $31\n");
  print(".end %s\n", f->x.name);
```

16.4 数据的定义

defconst 产生汇编指示符以分配一个标量，并把标量初始化成常量：

```
⟨MIPS functions 337⟩+≡                                   350 356 335
  static void defconst(ty, v) int ty; Value v; {
    switch (ty) {
    ⟨MIPS defconst 355⟩
    }
  }
```

不同的整数类型将产生大小相关的指示符及相应的常量域：

```
⟨MIPS defconst 355⟩≡                                     356 355
  case C: print(".byte %d\n",   v.uc); return;
  case S: print(".half %d\n",   v.ss); return;
  case I: print(".word 0x%x\n", v.i);  return;
  case U: print(".word 0x%x\n", v.u);  return;
```

数字地址常量分支将地址常量当成无符号整数处理：

```
⟨MIPS defconst 355⟩+≡                                    355 356 355
  case P: print(".word 0x%x\n", v.p); return;
```

defaddress 处理符号地址常量：

```
⟨MIPS functions 337⟩+≡                                   355 356 335
  static void defaddress(p) Symbol p; {
     print(".word %s\n", p->x.name);
  }
```

汇编程序的 .float 和 .double 指示符不能表示通过计算表达式得到的浮点常量（例如类型转换得到的浮点常量），所以 defconst 产生十六进制的浮点常量：

```
⟨MIPS defconst 355⟩+≡                                    356 356 355
  case F: print(".word 0x%x\n", *(unsigned *)&v.f); return;
```

如果 lcc 在低位优先的机器上运行，却为高位优先的机器编译生成代码，则一定要交换每个双精度数的高位和低位两部分，反之亦然：

```
⟨MIPS defconst 355⟩+≡                                    356 355
  case D: {
     unsigned *p = (unsigned *)&v.d;
     print(".word 0x%x\n.word 0x%x\n", p[swap], p[!swap]);
     return;
     }
```

要避免这种潜在的交换，程序必须约定宿主机与目标机的浮点编码方式一致。现在大多数目标机都使用 IEEE 浮点，这种约定无须强制。

defstring 生成处理字节序列的指示符：

```
⟨MIPS functions 337⟩+≡                                   356 356 335
  static void defstring(n, str) int n; char *str; {
     char *s;

     for (s = str; s < str + n; s++)
        print(".byte %d\n", (*s)&0377);
  }
```

由于 ANSI 换码允许字符串中嵌有空字节，所以 defstring 通过计数找到字符串尾。

export 通过使用汇编指示符使得本模块的符号在其他模块中可见：

```
⟨MIPS functions 337⟩+≡                                   356 356 335
  static void export(p) Symbol p; {
     print(".globl %s\n", p->x.name);
  }
```

相应地，import 使用配套的指示符使其他模块中的符号在本模块中可见：

```
⟨MIPS functions 337⟩+≡                                   356 357 335
  static void import(p) Symbol p; {
     if (!isfunc(p->type))
        print(".extern %s %d\n", p->name, p->type->size);
  }
```

MIPS 编译器的约定将忽略这些用于函数的指示符。

前端调用 defsymbol 通知新的符号，并提示后端初始化 x.name 域：

```
⟨MIPS functions 337⟩+≡                                    356 357 335
  static void defsymbol(p) Symbol p; {
     ⟨MIPS defsymbol 357⟩
  }
```

defsymbol 给每个静态局部变量定义唯一的标识，防止与其他静态局部变量重名：

```
⟨MIPS defsymbol 357⟩≡                                     357 357
  if (p->scope >= LOCAL && p->sclass == STATIC)
     p->x.name = stringf("L.%d", genlabel(1));
```

MIPS 编译器规则约定，这类符号以 L 和点号开头，如果生成的符号未按上述规则进行转换，那么 name 域已经保存一个数字串：

```
⟨MIPS defsymbol 357⟩+≡                                    357 357 357
  else if (p->generated)
     p->x.name = stringf("L.%s", p->name);
```

其他情况下，符号的名字在前端和后端中都相同：

```
⟨MIPS defsymbol 357⟩+≡                                    357 357
  else
     p->x.name = p->name;
```

许多 UNIX 汇编程序通常忽略符号表中以 L 开头的符号，所以我们将临时变量的名字都以 L 开头，这样编译器就能节省目标文件的空间。

address 的功能与 defsymbol 处理表示其他符号的符号一样，但名字中增加了偏移常量。

```
⟨MIPS functions 337⟩+≡                                    357 358 335
  static void address(q, p, n) Symbol q, p; int n; {
     q->x.offset = p->x.offset + n;
     if (p->scope == GLOBAL
     || p->sclass == STATIC || p->sclass == EXTERN)
        q->x.name = stringf("%s%s%d", p->x.name,
           n >= 0 ? "+" : "", n);
     else
        q->x.name = stringd(q->x.offset);
  }
```

对栈内的变量而言，address 只计算了调整后的偏移量。而对于通过标记访问的变量，address 将 x.name 设置成 name ± n 的字符串形式。如果偏移量是正的，那么“+”产生操作符，如果偏移是负，则由 %d 产生操作符。

MIPS 编译器约定对全局变量做了划分，从而能更快地存取较小的全局变量。通常情况下，MIPS 机器将寄存器值与有符号 16 位指令域相加形成地址，因此存取一个 32 位地址不得不使用多条指令。为了减少对这种指令序列的需求，翻译程序把小的全局变量放到一个大小为 64 KB 的 sdata 段内，另外再将 sdata 的基地址存入寄存器 $gp，这样我们就能用一条指令访问在这 64 KB 中的全局变量了。-Gn 选项设置阀值 gnum：

```
⟨MIPS data 337⟩+≡                                    338 358  335
  static int gnum = 8;

⟨parse -G flag 358⟩≡                                          337
  parseflags(argc, argv);
  for (i = 0; i < argc; i++)
     if (strncmp(argv[i], "-G", 2) == 0)
        gnum = atoi(argv[i] + 2);
```

前端调用接口过程 global 通知新的全局符号：

```
⟨MIPS functions 337⟩+≡                               357 359  335
  static void global(p) Symbol p; {
     if (p->u.seg == BSS) {
        ⟨define an uninitialized global 358⟩
     } else {
        ⟨define an initialized global 358⟩
     }
  }
```

global 将已初始化的小全局符号放入 sdata，其余的放入 data：

```
⟨define an initialized global 358⟩≡                           358
  if (p->u.seg == DATA
  && (p->type->size == 0 || p->type->size > gnum))
     print(".data\n");
  else if (p->u.seg == DATA)
     print(".sdata\n");
  print(".align %c\n", ".01.2...3"[p->type->align]);
  print("%s:\n", p->x.name);
```

当 p->type->size 未定时，其值为 0。这种情况通常发生在数组声明时没有注明数组边界。global 程序最后产生一个对齐指示符和一个标号。".01.2…3"[x] 是一个关于对齐字节数 x 的以 2 为底的对数的压缩表达式，这是 .align 指示符的数据处理格式。

对于未初始化的全局符号而言，指示符一般隐式定义了标号，预留了空间，并根据大小选取了合适的段：

```
⟨define an uninitialized global 358⟩≡                        358
  if (p->sclass == STATIC || Aflag >= 2)
     print(".lcomm %s,%d\n", p->x.name, p->type->size);
  else
     print(".comm %s,%d\n", p->x.name, p->type->size);
```

.comm 输出符号并进行了标记，这样即使其他模块也为相同标识符产生 comm 指示符，装配程序也只生成一个公用的全局符号。.lcomm 则不同，lcc 用 .lcomm 处理静态符号，防止输出静态符号。当多个模块定义同一全局符号时，如果使用了编译选项 -A-A（即 Aflag>=2），lcc 也会使用 .lcomm 生成装配对象。ANSI 之前的 C 允许多次定义同一符号，但从技术上讲，ANSI C 要求每个符号只定义一次，其他的模块应使用 extern 声明代替定义。

cseg 跟踪当前段：

```
⟨MIPS data 337⟩+≡                                    358  335
  static int cseg;
```

DATA 和 BSS 段中的符号完成各自的段分支处理，segment 只为文本段和文字段产生指示符：

```
⟨MIPS functions 337⟩+≡                                    358 359 335
  static void segment(n) int n; {
     cseg = n;
     switch (n) {
     case CODE: print(".text\n"); break;
     case LIT:  print(".rdata\n"); break;
     }
  }
```

因为 global 要为 BSS 的符号分配空间，所以除非符号不在 BSS 段中，否则 space 将产生指示符预留存储块：

```
⟨MIPS functions 337⟩+≡                                    359 359 335
  static void space(n) int n; {
     if (cseg != BSS)
        print(".space %d\n", n);
  }
```

.space 将块清零，这是 C 语言标准的要求。

16.5 块的复制

blkloop 产生循环从源地址处复制 size 个字节到目的地址处，其中源地址通过寄存器 sreg 和偏移量 soff 相加而成，目的地址通过寄存器 dreg 和偏移量 doff 相加而成。图 13-4 显示了 blkloop、blkfetch 和 blkstore 的运行情况。

```
⟨MIPS functions 337⟩+≡                                    359 360 335
  static void blkloop(dreg, doff, sreg, soff, size, tmps)
  int dreg, doff, sreg, soff, size, tmps[]; {
     int lab = genlabel(1);

     print("addu $%d,$%d,%d\n", sreg, sreg, size&~7);
     print("addu $%d,$%d,%d\n", tmps[2], dreg, size&~7);
     blkcopy(tmps[2], doff, sreg, soff, size&7, tmps);
     print("L.%d:\n", lab);
     print("addu $%d,$%d,%d\n", sreg, sreg, -8);
     print("addu $%d,$%d,%d\n", tmps[2], tmps[2], -8);
     blkcopy(tmps[2], doff, sreg, soff, 8, tmps);
     print("bltu $%d,$%d,L.%d\n", dreg, tmps[2], lab);
  }
```

tmps 命名 3 个寄存器，将它们用作临时寄存器。每次迭代都复制 8 个字节。初始代码将 sreg 和 tmps[2] 指向要复制的块的末端。如果块大小不是 8 的倍数，则第一次调用 blkcopy 会复制余下的几位。接着循环递减寄存器 sreg 和 tmps[2] 的值，并调用 blkcopy 复制 sreg 和 tmps[2] 当前所指的 8 个字节，循环直到寄存器 tmps[2] 中的值与寄存器 dreg 中的值相等才终止。

寄存器 reg 和偏移量 off 相加形成一个地址，blkfetch 以 1、2 或 4 个字节的形式将该地址单元的内容读取到寄存器 tmp：

```
⟨MIPS functions 337⟩+≡                                    359 360 335
  static void blkfetch(size, off, reg, tmp)
  int size, off, reg, tmp; {
     if (size == 1)
        print("lbu $%d,%d($%d)\n",  tmp, off, reg);
     else if (salign >= size && size == 2)
        print("lhu $%d,%d($%d)\n",  tmp, off, reg);
     else if (salign >= size)
        print("lw $%d,%d($%d)\n",   tmp, off, reg);
     else if (size == 2)
        print("ulhu $%d,%d($%d)\n", tmp, off, reg);
     else
        print("ulw $%d,%d($%d)\n",  tmp, off, reg);
  }
```

如果源寄存器对齐字节数（由 salign 给出）不小于要读取的单元的大小，那么 blkfetch 使用一般对齐方式读取，否则 blkfetch 用汇编伪指令读取未对齐的单元。对读取字节来说，对齐是没有实际意义的。blkstore 是 blkfetch 的逆过程：

```
⟨MIPS functions 337⟩+≡                                    360 335
  static void blkstore(size, off, reg, tmp)
  int size, off, reg, tmp; {
     if (size == 1)
        print("sb $%d,%d($%d)\n",  tmp, off, reg);
     else if (dalign >= size && size == 2)
        print("sh $%d,%d($%d)\n",  tmp, off, reg);
     else if (dalign >= size)
        print("sw $%d,%d($%d)\n",  tmp, off, reg);
     else if (size == 2)
        print("ush $%d,%d($%d)\n", tmp, off, reg);
     else
        print("usw $%d,%d($%d)\n", tmp, off, reg);
  }
```

深入阅读

Kane and Heinrich（1992）提供了 MIPS R3000 系列的参考手册。lcc 的 MIPS 代码生成器能在基的 MIPS R4000 系列机上运行，但是它不能利用 R4000 的 64 位指令。

练习

16.1 为什么小的全局数组不能进入 sdata?

16.2 为什么所有非零的参数建立区域必须至少是 16 字节长?

16.3 请解释当变参例程的第一个参数是浮点型或双精度型时，MIPS 的调用约定为什么不能处理该变参例程。

16.4 请解释在变长参数列表的未声明后缀中，为什么 MIPS 的调用约定不能实现结构的可靠传递，怎样才能解决这个问题?

16.5 对 lcc 进行扩展，使得 lcc 能产生有关标识符的类型和位置的信息，当调试程序报告和改变标识符的值时需要这些信息。

16.6 15.3 节介绍了 ralloc 的假设：在处理完所有寄存器之前，所有模板不能更改目标寄存器。lcc 的 CVID MIPS 模板（见第 343 页）通过两种方法达到了这一要求。请说明这两种方法。

16.7 以 MIPS 代码生成器为模式，为诸如 DEC Alpha 或 Motorola PowerPC 的 RISC 机器编写代码生成器。请先阅读 19.2 节。

第 17 章

A Retargetable C Compiler: Design and Implementation

SPARC 代码的生成

SPARC 也是 RISC 结构。SPARC 结构包括 32 个 32 位通用寄存器，32 个 32 位浮点寄存器，1 个 32 位紧缩指令集和两种寻址模式。与 MIPS 一样，访存时必须显式地使用存取指令。

SPARC 与 MIPS 最大的不同之处在于 SPARC 增加了 1 个寄存器窗口，能在调用和返回时自动存储和恢复寄存器。与此相关的调用约定也要求对 function 做许多改变。真正最简单的 function 实例请查看 X86 的 function。

lcc 的生成目标是汇编语言，而不是机器语言。MIPS 和 SPARC 的汇编程序也大不相同，因此两者的代码生成器也不一样。例如，对大多数 RISC 机器而言，一条指令就能实现寄存器值小常量的自增。稍大常量的自增则需要花费多条指令，通常先将常量扩展成 32 位存入临时变量，然后再与寄存器相加。但 MIPS 汇编程序屏蔽了这些特征，也就是说，我们可以使用任意大小的常量，汇编程序在必要时生成多条指令实现。SPARC 汇编程序的翻译工作相对而言更忠实于代码生成器生成的代码，它要求代码生成器为大小不同的常量生成不同的代码。另外，MIPS 汇编程序能够调度指令，而 SPARC 不能。

SPARC 汇编指令中，源操作数位于目的操作数之前。寄存器名字之前有一个“%”。表 17-1 列举的指令对于我们理解 SPARC 代码生成已经足够了。

表 17-1　SPARC 汇编器输入样例

汇编指令	意义
mov %i0, %o0	将寄存器 o0 的值设置为寄存器 i1 的值
sub %i0, %i1, %o0	将寄存器 o0 的值设置为寄存器 i0 的值减去寄存器 i1 的值
sub %i0, 1, %o0	将寄存器 o0 的值设置为寄存器 i0 的值减去 1
ldsb [%i0+4], %o0	将寄存器 o0 的值设置为地址为寄存器 i0 的值加上 4 的字节单元的值
ldsb [%i0+%i4],%o0	将寄存器 o0 的值设置为地址为寄存器 i0 的值加上寄存器 i4 的值的字节单元的值
fsubd $f0, $f2, $f4	将寄存器 f4 的值设置为寄存器 f0 的值减去寄存器 f2 的值，使用双精度浮点运算
fsubs $f0, $f2, $f4	将寄存器 f4 的值设置为寄存器 f0 的值减去寄存器 f2 的值，使用单精度浮点运算
ba L1	跳转到标号为 L1 的指令处
jmp [%i0]	跳转到寄存器 i0 所指的单元
cmp %i0, %i1	比较寄存器 i0 和 i1 的值，比较结果保存在条件标志中
bl L1	如果上次比较结果为小于，转移到标号 L1 处
byte 0x20	将内存中下一个字节单元初始化成十六进制数 20

文件 sparc.c 包含所有与 SPARC 结构相关的代码和数据。下面是 Iburg 规范，其中接口例程在文法的后面：

```
(sparc.md 362)≡
  %{
  (Iburg prefix 293)
  (interface prototypes)
  (SPARC prototypes)
  (SPARC data 365)
  %}
```

```
⟨terminal declarations 293⟩
%%
⟨shared rules 312⟩
⟨SPARC rules 367⟩
%%
⟨SPARC functions 364⟩
⟨SPARC interface definition 363⟩
```

最后的程序段 <SPARC interface definition 363> 对前端进行配置，并指向后端的 SPARC 的例程和数据：

```
⟨SPARC interface definition 363⟩= 363
  Interface sparcIR = {
     ⟨SPARC type metrics 363⟩
     0,  /* little_endian */
     1,  /* mulops_calls */
     1,  /* wants_callb */
     0,  /* wants_argb */
     1,  /* left_to_right */
     0,  /* wants_dag */
     ⟨interface routine names⟩
     stabblock, 0, 0, stabinit, stabline, stabsym, stabtype,
     {
        1,  /* max_unaligned_load */
        ⟨Xinterface initializer 277⟩
     }
  };

⟨SPARC type metrics 363⟩= 363
  1, 1, 0,  /* char */
  2, 2, 0,  /* short */
  4, 4, 0,  /* int */
  4, 4, 1,  /* float */
  8, 8, 1,  /* double */
  4, 4, 0,  /* T * */
  0, 1, 0,  /* struct */
```

因为某些 SPARC 处理器是用代码而不是硬件实现乘法和除法，所以 mulops_calls 的值为 1。

SPARC 和 MIPS 关于结构参数和返回值的约定是相反的。MIPS 约定使用 ARGB，但不使用 CALLB，SPARC 的约定与之相反。

SPARC 删除了符号表生成器。在 stab 例程中，两个 0 表示无须为特定目标生成代码。当在其他机器上建立 SPARC 代码生成器以形成交叉编译器时，由于其他 stab 例程的代码包含或引用了 SPARC 系统特有的头文件和标识符，所以使用 #defined 将其他 stab 例程也定义为 0。

17.1 寄存器

SPARC 汇编程序设计者必须了解 SPARC 的 32 个 32 位通用寄存器，其中大多数寄存器组织成可重叠的寄存器窗口栈。大部分的例程通过分配新的窗口来存储局部变量、临时变量和输出参数（调用约定规定有些参数通过寄存器传递），返回时释放窗口。

每个通用寄存器至少对应两个名字，见表 17-2。一个是 r0 – r31，另一个名字编码主要标识

该寄存器在寄存器窗口中的使用和位置。硬件将 g0 置为 0。指令可以写 g0，但实际上并不能改变 g0。因此读取 g0 时，g0 的值总为 0。

表 17-2 SPARC 通用寄存器

基本名	等价名	解释
r0 – r7	g0 – g7	固定的全局寄存器，不在栈中
r8 – r15	o0 – o7	输出参数，在栈中
r16 – r23	l0 – l7	局部寄存器，在栈中
r24 – r31	i0 – i7	输入参数，在栈中

SPARC 机器组织寄存器窗口使得调用程序的物理寄存器 o0 – o7 与被调用程序的 i0 – i7 指向相同的寄存器。图 17-1 显示了 f 返回之前各个寄存器窗口的状态：

```
main() { f(); }
f() { return; }
```



图 17-1 main 调用 f

SPARC 包括 32 个通用寄存器。但是由于 g0 – g7 不能用于栈，所以每次调用只用到 16 个通用寄存器。阴影表示调用程序的物理寄存器 o0 – o7 与被调用程序的 i0 – i7 相同。

SPARC 接口程序 progend 不起作用，progbeg 分析与目标机器相关的编译选项 -p 和 -pg，lcc 以此为 SPARC 的性能分析工具生成代码，本节对此不做描述。progbeg 也初始化用于描述寄存器组的结构。

```
⟨SPARC functions 364⟩=                                    366  363
  static void progbeg(argc, argv) int argc; char *argv[]; {
      int i;

      ⟨shared progbeg 290⟩
      ⟨parse SPARC flags⟩
      ⟨initialize SPARC register structures 365⟩
  }
```

progbeg 用数组 greg 的每个元素描述一个通用寄存器：

```
⟨SPARC data 365⟩=                                          365 362
  static Symbol greg[32];
  static Symbol *oreg = &greg[8], *ireg = &greg[24];
```

与表 17-2 相应的初始化代码为：

```
⟨initialize SPARC register structures 365⟩=                365 364
  for (i = 0; i < 8; i++) {
    greg[i +  0] = mkreg(stringf("g%d", i), i +  0, 1, IREG);
    greg[i +  8] = mkreg(stringf("o%d", i), i +  8, 1, IREG);
    greg[i + 16] = mkreg(stringf("l%d", i), i + 16, 1, IREG);
    greg[i + 24] = mkreg(stringf("i%d", i), i + 24, 1, IREG);
  }
```

SPARC 机器还有 32 个 32 位浮点寄存器，它们在汇编语言中标识为 f0 ~ f31。这些寄存器不包含在通用寄存器栈中。偶－奇寄存器对可以用作双精度浮点寄存器。progbeg 用数组 freg 的每个元素表示一个单精度浮点寄存器，数组 freg2 的每个偶数元素表示一个双精度浮点寄存器：

```
⟨SPARC data 365⟩+=                                  365 381 362
  static Symbol freg[32], freg2[32];

⟨initialize SPARC register structures 365⟩+=               365 365 364
  for (i = 0; i < 32; i++)
    freg[i]  = mkreg("%d", i, 1, FREG);
  for (i = 0; i < 31; i += 2)
    freg2[i] = mkreg("%d", i, 3, FREG);
```

rmap 存储通配符，这些通配符标识每种数据类型使用的默认寄存器类：

```
⟨initialize SPARC register structures 365⟩+=               365 365 364
  rmap[C] = rmap[S] = rmap[P] = rmap[B] = rmap[U] = rmap[I] =
    mkwildcard(greg);
  rmap[F] = mkwildcard(freg);
  rmap[D] = mkwildcard(freg2);
```

g0 ~ g7，i6 ~ i7，o0 或 o6 ~ o7 不用于存放一般变量和临时变量。调用约定规定调用过程不保存 g0 ~ g7、o6 保存栈指针，也称为 sp。i6 保存帧指针，也称为 fp。每个调用程序都把返回地址放入 o7，在被调用程序里对应的是 i7（见图 17-1）。每个函数的返回值都存入 i0，从调用程序来看为 o0。一个例程调用其他例程时总是假设被调用程序会破坏 o0 ~ o7。浮点值通过 f 或 f0-f1 寄存器对返回。

lcc 将临时变量存入其余的通用寄存器 i0 ~ i5、l0 ~ l7 和 o1 ~ o5 中：

```
⟨initialize SPARC register structures 365⟩+=               365 365 364
  tmask[IREG] = 0x3fff3e00;
```

寄存器变量只能使用这些寄存器中的一半，即 l4 ~ l7 和 i0 ~ i5：

```
⟨initialize SPARC register structures 365⟩+=               365 366 364
  vmask[IREG] = 0x3ff00000;
```

在第 16 章曾提到 tmask 和 vmask。tmask 标识的寄存器存储计算表达式过程中产生的临时变量，vmask 标识的寄存器存储寄存器变量。对于临时变量和寄存器变量，在一定程度上可不做区分，但一般情况下两个集合是互斥的：存储寄存器变量的寄存器在例程的函数头代码溢出。这

样可避免所有的调用点都溢出活跃的寄存器变量；存储临时变量的寄存器可在调用点溢出，因为活跃的临时变量为数不多，寄存器分配器很容易就能识别出它们。在 SPARC 结构的机器中，当进入一个例程时，寄存器栈会自动保存许多寄存器，所以最好是允许临时变量可以存储在所有 tmask 标识的寄存器中。我们将寄存器变量占用的寄存器数限制在临时变量可用的寄存器数的一半左右，是因为寄存器变量的第一次改写都发生在寄存器中，所以它们占用的寄存器数目不宜太少；而另一方面，如果给临时变量留下的可用寄存器太少，则又可能引发大量溢出或者完全破坏寄存器分配器。

调用约定规定调用过程中不保存浮点寄存器，这些寄存器仅用于存储临时变量：

```
⟨initialize SPARC register structures 365⟩+≡                    ↑365  364
  tmask[FREG]  = ~(unsigned)0;
  vmask[FREG]  = 0;
```

target 调用 setreg 对需要特定寄存器的节点进行标记，调用 rtarget 对需要于节点存放在特定寄存器中的节点进行标记：

```
⟨SPARC functions 364⟩+≡                                  ↑364 366↓  363
  static void target(p) Node p; {
     switch (p->op) {
     ⟨SPARC target 370⟩
     }
  }
```

如果某条指令将改写寄存器，clobber 会先调用 spill 把当前寄存器保存起来，以后再恢复原值：

```
⟨SPARC functions 364⟩+≡                                  ↑366 367↓  363
  static void clobber(p) Node p; {
     switch (p->op) {
     ⟨SPARC clobber 373⟩
     }
  }
```

我们将在下一节介绍有关 target 和 clobber 的 case 分支处理及相关指令。

17.2 指令的选取

表 17-3 总结了 SPARC 代码生成器的 Iburg 规范中出现的非终结符，此表概括了 lcc 的树文法的组织。

表 17-3 SPARC 的非终结符

名字	匹配对象
addr	针对内存读写指令的地址计算
addrg	ADDRG 节点
base	addr 减去（寄存器 + 寄存器）地址模式
call	call 指令的操作数
con	常量
con13	寄存器和常量（取带符号的 13 位）
rc	寄存器和常量
reg	计算结果在寄存器中的运算
stk	局部变量和形式参数的地址
stk13	局部变量和形式参数的地址（取带符号的 13 位）
stmt	副作用产生的运算

SPARC 的汇编语言中，% 的作用与 lcc 的（用于模板转义字符）不同。例如伪指令 set，它的作用是将寄存器设置为整数常量或地址，所以 ADDRGP 的规则是：

```
⟨SPARC rules 367⟩≡                                          367 363
  reg: ADDRGP  "set %a,%%%c\n"  1
```

模板子串 %%% 产生一个 %，模板子串 %c 产生目的寄存器的名字，所以模式子串 %%%c 指示代码产生器在所生成的代码中的寄存器名字前加上百分号。这种做法并不可取，但它与 print 和 printf 一致。如果我们选择了不同的转义符，在其他目标机器上可能需要额外的处理。

SPARC 的指令若含有立即数域，则存储一个有符号的 13 位常量，所以一些指令使用与目标机器相关的代价函数 imm，如果 p 的常量值能够放入 13 位的常量中，imm 返回 0，否则返回一个较大值：

```
⟨SPARC functions 364⟩+≡                                     366 373 363
  static int imm(p) Node p; {
     return range(p, -4096, 4095);
  }
```

例如，假设能用 13 位表示 ADDRFP 和 ADDRLP 节点中的有符号偏移量，那么用一条指令就能把地址存入寄存器：

```
⟨SPARC rules 367⟩+≡                                         367 367 363
  stk13: ADDRFP  "%a"                    imm(a)
  stk13: ADDRLP  "%a"                    imm(a)
  reg:   stk13   "add %0,%%fp,%%%c\n"  1
```

否则，必须花费多条指令：

```
⟨SPARC rules 367⟩+≡                                         367 367 363
  stk: ADDRFP  "set %a,%%%c\n"                                  2
  stk: ADDRLP  "set %a,%%%c\n"                                  2
  reg: ADDRFP  "set %a,%%%c\nadd %%%c,%%fp,%%%c\n"  3
  reg: ADDRLP  "set %a,%%%c\nadd %%%c,%%fp,%%%c\n"  3
```

如果某个常量不能由一条指令载入，则伪指令 set 生成两条指令；如果一条指令可以载入，那么交由 stk13 处理。前一章 MIPS 的代码生成器已经用到这种方法，只是 MIPS 的汇编程序彻底隐藏了常量大小的检测，在 SPARC 中最好也使用这个功能。但是 SPARC 的汇编程序将该问题的一部分留给了程序员或编译器处理，因此我们也没有其他选择。

上述 4 个规则看起来与下面的相当：

```
stk: ADDRFP  "set %a,%%%c\n"        2
stk: ADDRLP  "set %a,%%%c\n"        2
reg: stk     "add %0,%%fp,%%%c\n"  1
```

但上面较短的规则会出现错误，原因是它们导致函数 reduce 在一个 x.inst 中存储了两个不同的值。回顾前面的内容，如果节点有标识匹配规则的非终结符，那么 x.inst 作为一条指令记录该非终结符。上述较短规则的问题是，ADDRLP 或 ADDRFP 的 x.inst 域不能标记 stk 和 reg。

非终结符 con13 匹配较小的整型常量：

```
⟨SPARC rules 367⟩+≡                                         367 368 363
  con13: CNSTC  "%a"  imm(a)
  con13: CNSTI  "%a"  imm(a)
  con13: CNSTP  "%a"  imm(a)
```

```
con13: CNST5  "%a"  imm(a)
con13: CNSTU  "%a"  imm(a)
```

指令读写存储器单元时首先进行地址计算，将某个寄存器值与一个13位有符号常量相加形成地址：

```
⟨SPARC rules 367⟩+≡                                    367 368 363
  base: ADDI(reg,con13)  "%%0+%1"
  base: ADDP(reg,con13)  "%%0+%1"
  base: ADDU(reg,con13)  "%%0+%1"
```

如果常量是0，或者寄存器是g0，那么相加将退化成简单的间接或直接寻址：

```
⟨SPARC rules 367⟩+≡                                    368 368 363
  base: reg     "%%0"
  base: con13   "%0"
```

如果寄存器存储的是帧指针，那么相加将产生形参的地址或局部变量的地址：

```
⟨SPARC rules 367⟩+≡                                    368 368 363
  base: stk13   "%%fp+%0"
```

硬件也能实现两个寄存器相加以形成地址：

```
⟨SPARC rules 367⟩+≡                                    368 368 363
  addr: base            "%0"
  addr: ADDI(reg,reg)   "%%0+%%1"
  addr: ADDP(reg,reg)   "%%0+%%1"
  addr: ADDU(reg,reg)   "%%0+%%1"
  addr: stk             "%%fp+%%0"
```

大多数存取类作使用上述寻址模式：

```
⟨SPARC rules 367⟩+≡                                    368 368 363
  reg:  INDIRC(addr)     "ldsb [%0],%%%c\n"  1
  reg:  INDIRS(addr)     "ldsh [%0],%%%c\n"  1
  reg:  INDIRI(addr)     "ld [%0],%%%c\n"    1
  reg:  INDIRP(addr)     "ld [%0],%%%c\n"    1
  reg:  INDIRF(addr)     "ld [%0],%%f%c\n"   1
  stmt: ASGNC(addr,reg)  "stb %%%1,[%0]\n"   1
  stmt: ASGNS(addr,reg)  "sth %%%1,[%0]\n"   1
  stmt: ASGNI(addr,reg)  "st %%%1,[%0]\n"    1
  stmt: ASGNP(addr,reg)  "st %%%1,[%0]\n"    1
  stmt: ASGNF(addr,reg)  "st %%f%1,[%0]\n"   1
```

ldd和std指令完成双精度数的存取，存取地址必须是8的倍数。由于调用约定只保证参数和全局变量的地址为4的倍数，所以ldd和std只能操作局部变量：

```
⟨SPARC rules 367⟩+≡                                    368 369 363
  addrl: ADDRLP          "%%%fp+%a"          imm(a)

  reg:   INDIRD(addrl)   "ldd [%0],%%f%c\n"  1
  stmt:  ASGND(addrl,reg) "std %%f%1,[%0]\n"  1
```

伪指令 ld2 和 st2 生成指令对存取按 4 字节对齐的双精度数。当地址由两个寄存器的值相加形成时，某些 SPARC 的汇编程序可能产生有潜在错误的代码。所以调用规则在这些伪指令中加入非终结符 base，它的作用是避过“寄存器 + 寄存器”类型的寻址：

```
⟨SPARC rules 367⟩+≡                                    368 369 363
  reg:  INDIRD(base)      "ld2 [%0],%%f%c\n"  2
  stmt: ASGND(base,reg)   "st2 %%f%1,[%0]\n"  2
```

对于汇编程序的这个小问题，定义了 base 和 addr 的规则可以合起来定义一个单独的非终结符。

当没有空闲的可分配寄存器时，溢出程序必须生成代码存储其中一个寄存器。当偏移量不能放入 SPARC 的立即数域时，ASGN 规则生成多条指令。这些指令需要一个寄存器用来通信，这样违反了溢出程序的假设。lcc 解决这个问题的方法是在 ASGN 规则中加入第二次复制。第二次复制时，用不可分配的寄存器 g1 帮助保存局部变量，因为 lcc 只溢出局部变量，而且局部变量不是立即寻址的。

```
⟨SPARC rules 367⟩+≡                                    369 369 363
  spill:  ADDRLP          "%a" !imm(a)

  stmt: ASGNC(spill,reg)  "set %0,%%g1\nstb %%%1,[%%fp+%%g1]\n"
  stmt: ASGNS(spill,reg)  "set %0,%%g1\nsth %%%1,[%%fp+%%g1]\n"
  stmt: ASGNI(spill,reg)  "set %0,%%g1\nst %%%1,[%%fp+%%g1]\n"
  stmt: ASGNP(spill,reg)  "set %0,%%g1\nst %%%1,[%%fp+%%g1]\n"
  stmt: ASGNF(spill,reg)  "set %0,%%g1\nst %%f%1,[%%fp+%%g1]\n"
  stmt: ASGND(spill,reg)  "set %0,%%g1\nstd %%f%1,[%%fp+%%g1]\n"
```

规则有一个虚设的代价底线值 0，这样一旦匹配就会成功，但这种情况并不常见。这些规则也适用于存储没有溢出的寄存器，这些情况下使用 0 值是无害的。参见练习 17.7。

ldsb 和 ldsh 扩展载入单元的符号位，它们以最低代价实现了 CVCI 和 CVSI。ldub 和 lduh 清除最高位，并以最低代价实现了 CVCU 和 CVSU。

```
⟨SPARC rules 367⟩+≡                                    369 369 363
  reg: CVCI(INDIRC(addr))  "ldsb [%0],%%%c\n"  1
  reg: CVSI(INDIRS(addr))  "ldsh [%0],%%%c\n"  1
  reg: CVCU(INDIRC(addr))  "ldub [%0],%%%c\n"  1
  reg: CVSU(INDIRS(addr))  "lduh [%0],%%%c\n"  1
```

整型变换中，在转换后的类型不会加宽的情况下也生成“寄存器 – 寄存器”移动指令。在第 16 章中曾提到，move 指令返回 1，并标记可以被 requate 和 moveself 优化的节点。

```
⟨SPARC rules 367⟩+≡                                    369 370 363
  reg: CVIC(reg)  "mov %%%0,%%%c\n"  move(a)
  reg: CVIS(reg)  "mov %%%0,%%%c\n"  move(a)
  reg: CVIU(reg)  "mov %%%0,%%%c\n"  move(a)
  reg: CVPU(reg)  "mov %%%0,%%%c\n"  move(a)
  reg: CVUC(reg)  "mov %%%0,%%%c\n"  move(a)
  reg: CVUI(reg)  "mov %%%0,%%%c\n"  move(a)
  reg: CVUP(reg)  "mov %%%0,%%%c\n"  move(a)
  reg: CVUS(reg)  "mov %%%0,%%%c\n"  move(a)
```

如果节点没有定位到特定寄存器，那么下面的规则不生成任何指令：

```
⟨SPARC rules 367⟩+≡                                    369 370 363
  reg: CVIC(reg)  "%0"  notarget(a)
  reg: CVIS(reg)  "%0"  notarget(a)
  reg: CVUC(reg)  "%0"  notarget(a)
  reg: CVUS(reg)  "%0"  notarget(a)
```

第二个列表看起来比前面的短了许多，实际上加入 14.7 节中与目标机器相关的段 <shared rules> 就相同了。

LOAD 也生成寄存器副本：

```
⟨SPARC rules 367⟩+≡                                    370 371 363
  reg: LOADC(reg)  "mov %%%0,%%%c\n"  move(a)
  reg: LOADI(reg)  "mov %%%0,%%%c\n"  move(a)
  reg: LOADP(reg)  "mov %%%0,%%%c\n"  move(a)
  reg: LOADS(reg)  "mov %%%0,%%%c\n"  move(a)
  reg: LOADU(reg)  "mov %%%0,%%%c\n"  move(a)
```

如果这些规则也能共享就很好了，但模板是与机器相关的。

寄存器 g0 由硬件设定为 0，所以值为 0 的整型节点 CNST 不生成代码：

```
⟨SPARC rules 367⟩+≡                                    370 372 363
  reg: CNSTC  "# reg\n"  range(a, 0, 0)
  reg: CNSTI  "# reg\n"  range(a, 0, 0)
  reg: CNSTP  "# reg\n"  range(a, 0, 0)
  reg: CNSTS  "# reg\n"  range(a, 0, 0)
  reg: CNSTU  "# reg\n"  range(a, 0, 0)
```

前面曾提到，在计算代价表达式的宏时，a 表示被标识的节点，但在这里，a 表示常量值；代码判断 a 是否为 0，如果为 0，target 为这些节点返回 g0：

```
⟨SPARC target 370⟩≡                                    372 366
  case CNSTC: case CNSTI: case CNSTS: case CNSTU: case CNSTP:
     if (range(p, 0, 0) == 0) {
        setreg(p, greg[0]);
        p->x.registered = 1;
     }
     break;
```

对寄存器 g0 进行分配是没有意义的，所以 target 标识节点以避免分配 g0。

set 伪指令能把任何常量载入寄存器：

```
⟨SPARC rules 367⟩+≡                                    370 373 363
  reg: con  "set %0,%%%c\n"  1
```

如果常量可以用 13 位来表示，则 set 只生成一条指令，否则生成两条。这些细节都由汇编程序完成，我们无须关心。

二元操数运算指令的第二操作数可以是寄存器或 13 位的常量：

```
⟨SPARC rules 367⟩+≡                                    370 374 363
  rc: con13  "%0"
  rc: reg    "%%%0"
```

但指令的第一操作数和结果必须是寄存器：

```
⟨SPARC rules 367⟩+≡                                          370 371 363
  reg: ADDI(reg,rc)   "add %%0%0,%1,%%%c\n"  1
  reg: ADDP(reg,rc)   "add %%0%0,%1,%%%c\n"  1
  reg: ADDU(reg,rc)   "add %%0%0,%1,%%%c\n"  1
  reg: BANDU(reg,rc)  "and %%0%0,%1,%%%c\n"  1
  reg: BORU(reg,rc)   "or %%0%0,%1,%%%c\n"   1
  reg: BXORU(reg,rc)  "xor %%0%0,%1,%%%c\n"  1
  reg: SUBI(reg,rc)   "sub %%0%0,%1,%%%c\n"  1
  reg: SUBP(reg,rc)   "sub %%0%0,%1,%%%c\n"  1
  reg: SUBU(reg,rc)   "sub %%0%0,%1,%%%c\n"  1
```

移位指令只接受 0–31 之间的常量作为第二类作数：

```
⟨SPARC rules 367⟩+≡                                          371 371 363
  rc5: CNSTI  "%a"    range(a, 0, 31)
  rc5: reg    "%%%0"
```

第一操作数和结果必须是寄存器：

```
⟨SPARC rules 367⟩+≡                                          371 371 363
  reg: LSHI(reg,rc5)  "sll %%0%0,%1,%%%c\n"  1
  reg: LSHU(reg,rc5)  "sll %%0%0,%1,%%%c\n"  1
  reg: RSHI(reg,rc5)  "sra %%0%0,%1,%%%c\n"  1
  reg: RSHU(reg,rc5)  "srl %%0%0,%1,%%%c\n"  1
```

3 种布尔操作都有变体形式补全第二操作数：

```
⟨SPARC rules 367⟩+≡                                          371 371 363
  reg: BANDU(reg,BCOMU(rc))  "andn %%0%0,%1,%%%c\n"  1
  reg: BORU(reg,BCOMU(rc))   "orn %%0%0,%1,%%%c\n"   1
  reg: BXORU(reg,BCOMU(rc))  "xnor %%0%0,%1,%%%c\n"  1
```

一元操作只能使用寄存器作为操作数：

```
⟨SPARC rules 367⟩+≡                                          371 371 363
  reg: NEGI(reg)   "neg %%0%0,%%%c\n"  1
  reg: BCOMU(reg)  "not %%0%0,%%%c\n"  1
```

对有符号字符和有符号短整数的正宽转换，可以通过算术左移和右移实现符号位的扩展：

```
⟨SPARC rules 367⟩+≡                                          371 371 363
  reg: CVCI(reg)  "sll %%0%0,24,%%%c; sra %%%c,24,%%%c\n"  2
  reg: CVSI(reg)  "sll %%0%0,16,%%%c; sra %%%c,16,%%%c\n"  2
```

无符号数的转换需要使用 and 指令清除高位：

```
⟨SPARC rules 367⟩+≡                                          371 372 363
  reg: CVCU(reg)  "and %%0%0,0xff,%%%c\n"                          1
  reg: CVSU(reg)  "set 0xffff,%%g1; and %%%0,%%g1,%%%c\n"          2
```

CVSU 需要一个 16 位掩码，它与 CVCU 不同。

SPARC 结构中，所有的无条件跳转和条件分支都同到“单指令”延迟槽（delay slot）。跳转和分支后的指令（称这条指令在延迟槽中）总会执行，就像它们在跳转和转移之前就已经执行了一样。就目前而言，可暂时用无害的 nop 填充每一个延迟槽。ba 指令处理常量地址，余下的由 jmp 指令完成，即 jmp 处理 switch 语句的目标地址。

```
⟨SPARC rules 367⟩+≡                                          371 372 363
  addrg: ADDRGP          "%a"
  stmt:  JUMPV(addrg)    "ba %0; nop\n"   2
  stmt:  JUMPV(addr)     "jmp %0; nop\n"  2
  stmt:  LABELV          "%a:\n"
```

整型关系的比较在寄存器间或是寄存器与常量间进行：

```
⟨SPARC rules 367⟩+≡                                          372 372 363
  stmt: EQI(reg,rc)  "cmp %%%0,%1; be %a; nop\n"   3
  stmt: GEI(reg,rc)  "cmp %%%0,%1; bge %a; nop\n"  3
  stmt: GEU(reg,rc)  "cmp %%%0,%1; bgeu %a; nop\n" 3
  stmt: GTI(reg,rc)  "cmp %%%0,%1; bg %a; nop\n"   3
  stmt: GTU(reg,rc)  "cmp %%%0,%1; bgu %a; nop\n"  3
  stmt: LEI(reg,rc)  "cmp %%%0,%1; ble %a; nop\n"  3
  stmt: LEU(reg,rc)  "cmp %%%0,%1; bleu %a; nop\n" 3
  stmt: LTI(reg,rc)  "cmp %%%0,%1; bl %a; nop\n"   3
  stmt: LTU(reg,rc)  "cmp %%%0,%1; blu %a; nop\n"  3
  stmt: NEI(reg,rc)  "cmp %%%0,%1; bne %a; nop\n"  3
```

call 指令处理常量地址或者某个已计算的地址：

```
⟨SPARC rules 367⟩+≡                                          372 372 363
  call: ADDRGP           "%a"
  call: addr             "%0"
  reg:  CALLD(call)      "call %0; nop\n"                  2
  reg:  CALLF(call)      "call %0; nop\n"                  2
  reg:  CALLI(call)      "call %0; nop\n"                  2
  stmt: CALLV(call)      "call %0; nop\n"                  2
  stmt: CALLB(call,reg)  "call %0; st %%%1,[%%sp+64]\n"    2
```

CALLB 指令将返回块的地址存入栈，传送地址，存储指令占用延迟槽。

编译前端为每个 RET 节点后增加一个转移到程序尾代码的跳转，所以 RET 节点不生成代码，只用来帮助后续定位返回寄存器：

```
⟨SPARC rules 367⟩+≡                                          372 373 363
  stmt: RETD(reg)  "# ret\n"  1
  stmt: RETF(reg)  "# ret\n"  1
  stmt: RETI(reg)  "# ret\n"  1
```

函数把返回值存入 f0、f0 ~ f1 或 o0（被调用程序将 o0 视为 i0）。target 遵循这个约定：

```
⟨SPARC target 370⟩+≡                                         370 372 366
  case CALLD: setreg(p, freg2[0]);      break;
  case CALLF: setreg(p, freg[0]);       break;
  case CALLI:
  case CALLV: setreg(p, oreg[0]);       break;
  case RETD:  rtarget(p, 0, freg2[0]); break;
  case RETF:  rtarget(p, 0, freg[0]);  break;
```

RETI 分支标记节点以防为其分配寄存器，避免产生不一致：

```
⟨SPARC target 370⟩+≡                                         372 373 366
  case RETI:
```

```
    rtarget(p, 0, ireg[0]);
    p->kids[0]->x.registered = 1;
    break;
```

如果某个例程的第一参数是整型则驻留在 i0。如果某个函数返回值是整数，i0 也保存这个函数返回值。lcc 的寄存器分配器溢出临时变量但不溢出形参，如果请求它分配 i0 给 RETI，分配将失败；而且，形参在返回时就已清除，所以我们仅标记该节点已分配，这样就能将 i0 分给 RETI 了，同时阻止了寄存器分配器的任何操作，包括溢出形参。

在调用时，寄存器钱能自动存储并恢复通用寄存器，所以只有浮点寄存器（除去返回寄存器）需要显式地存储和恢复：

```
⟨SPARC clobber 373⟩=                                          375  366
  case CALLB: case CALLD: case CALLF: case CALLI:
     spill(~(unsigned)3, FREG, p);
     break;
  case CALLV:
     spill(oreg[0]->x.regnode->mask, IREG, p);
     spill(~(unsigned)3, FREG, p);
     break;
```

在第 16 章提到，给节点分配寄存器后，ralloc 将调用目标机器的 clobber。

doarg 将一个整型常量符号存储到每个 ARG 节点的 syms[RX] 域，这个整型常量等于参数偏移量除以 4。对大多数参数而言，该整型常量命名了输出 o- 寄存器：

```
⟨SPARC functions 364⟩+=                                  367 324  363
  static void doarg(p) Node p; {
     p->syms[RX] = intconst(mkactual(4,
        p->syms[0]->u.c.v.i)/4);
  }
```

ARG 节点的执行没有副作用，通常并不使用 syms[RX]。但是 SPARC 的调用约定是通过寄存器定位和赋值实现 ARG 节点的，所以使用 RX 很自然。

对于输出参数，寄存器定位程序计算参数的头 24 个字节，并将它们存入寄存器。实现过程是 target 调用 rtarget 将子节点存入相应的 o- 寄存器，然后将 ARG 改变为读取同一个寄存器的 LOAD，emit 和 moveself 可对该过程进行优化：

```
⟨SPARC target 370⟩+=                                     372 326  366
  case ARGI: case ARGP:
     if (p->syms[RX]->u.c.v.i < 6) {
        rtarget(p, 0, oreg[p->syms[RX]->u.c.v.i]);
        p->op = LOAD+optype(p->op);
        setreg(p, oreg[p->syms[RX]->u.c.v.i]);
     }
     break;
```

若调用所带的参数过多，则必须将多出的一部分通过存储器传递。为了实现通过存储器传递参数，汇编程序模板撤销了除法（存储的是参数偏移量除 4），并加上 68：

```
⟨SPARC rules 367⟩+=                                      372 324  363
  stmt: ARGI(reg)  "st %%%0,[%%sp+4*%c+68]\n"  1
  stmt: ARGP(reg)  "st %%%0,[%%sp+4*%c+68]\n"  1
```

sp 指向 16 个字单元（64 字节），若当前的寄存器窗口已被耗尽而必须溢出寄存器；这时操作系统将用上面的 16 字单元来保存例程的 i- 寄存器和 l- 寄存器。下一个字预留作为返回块结构的地址（如果有的话）。再将它后面的字预留作为输出参数，即使通过 i0 – i5 传递的参数在这里也有预留空间。所以参数偏移量为 n 的参数是 %sp+n+68，这正好可以解释上述模板的含义。图 17-2 显示了 SPARC 的帧。



图 17-2 SPARC 帧布局

可变参数例程的代码只能使用偏移量为 20 的单元，所以浮点参数也必须通过 o0 – o5 传递。注意对比 MIPS 调用约定的相关部分。lcc 假设浮点操作码产生浮点寄存器，所以一棵树不能将没有转换的浮点值存入整型寄存器。emit2 必须处理这个特殊的 ARG：

```
⟨SPARC rules 367⟩+≡                                    373 375  363
  stmt: ARGD(reg)  "# ARGD\n"  1
  stmt: ARGF(reg)  "# ARGF\n"  1

⟨SPARC functions 364⟩+≡                                373 378  363
  static void emit2(p) Node p; {
     switch (p->op) {
     ⟨SPARC emit2 374⟩
     }
  }
```

ARGF 必须从浮点寄存器中取出一个值并放到 o- 寄存器或栈中。每个输出参数都预留一个栈槽，而从浮点寄存器到通用寄存器的唯一路径就是经过存储器，所以 emit2 把浮点寄存器复制到栈中，然后再从栈槽读取到 o- 寄存器，除非我们超过了 o5：

```
⟨SPARC emit2 374⟩≡                                     375  374
  case ARGF: {
     int n = p->syms[RX]->u.c.v.i;
     print("st %%f%d,[%%sp+4*%d+68]\n",
```

```
        getregnum(p->x.kids[0]), n);
    if (n <= 5)
        print("ld [%%sp+4*%d+68],%%o%d\n", n, n);
    break;
}
```

ARGD 的执行情况与 ARGF 类似，但它需要两次存储和最多两次读取：

```
⟨SPARC emit2 374⟩+≡                                    374 377  374
  case ARGD: {
      int n = p->syms[RX]->u.c.v.i;
      int src = getregnum(p->x.kids[0]);
      print("st %%f%d,[%%sp+4*%d+68]\n", src, n);
      print("st %%f%d,[%%sp+4*%d+68]\n", src+1, n+1);
      if (n <= 5)
          print("ld [%%sp+4*%d+68],%%o%d\n", n, n);
      if (n <= 4)
          print("ld [%%sp+4*%d+68],%%o%d\n", n+1, n+1);
      break;
  }
```

如果一个双精度浮点参数前有 5 个整数，那么将浮点参数的前半部分存入 o5，后半部分存入栈。这样分离传输双精度数看起来不可思议，但是对于可变参数的例程只能这样处理，例程的尾代码将这两部分重新合起来。

要求寄存器分配器为浮点节点分配通用寄存器似乎不很明智，所以 clobber 调用 spill 确保参数寄存器中的所有活跃值都在浮点型 ARG 之前被存储，以便后来恢复：

```
⟨SPARC clobber 373⟩+≡                                  373 376  366
  case ARGF:
      if (p->syms[2]->u.c.v.i <= 6)
          spill((1<<(p->syms[2]->u.c.v.i + 8)), IREG, p);
      break;
  case ARGD:
      if (p->syms[2]->u.c.v.i <= 5)
          spill((3<<(p->syms[2]->u.c.v.i + 8))&0xff00, IREG, p);
      break;
```

MIPS 代码生成器避开了这一步，因为它从来不会因为其他目的而分配参数寄存器。但是对于 SPARC 约定，若 o1 ~ o5 中没有输出参数，它们将存储临时变量。

第一个 SPARC 系统没有提供乘法、除法和求余指令，所以标准库提供了等价的函数。现在舍弃这些系统还为时过早，所以 lcc 设置了标志 mulops_calls，并且在已经提供了乘法指令的新机器上保留这些函数（参见练习 17.1）：

```
⟨SPARC rules 367⟩+≡                                    374 376  363
  reg: DIVI(reg,reg)  "call .div,2; nop\n"   2
  reg: DIVU(reg,reg)  "call .udiv,2; nop\n"  2
  reg: MODI(reg,reg)  "call .rem,2; nop\n"   2
  reg: MODU(reg,reg)  "call .urem,2; nop\n"  2
  reg: MULI(reg,reg)  "call .mul,2; nop\n"   2
  reg: MULU(reg,reg)  "call .umul,2; nop\n"  2
```

target使用o0和o1传递操作数，并把结果放进o0：

```
⟨SPARC target 370⟩+≡                                    373 366
  case DIVI: case MODI: case MULI:
  case DIVU: case MODU: case MULU:
     setreg(p, oreg[0]);
     rtarget(p, 0, oreg[0]);
     rtarget(p, 1, oreg[1]);
     break;
```

库函数只是溢写o1-o5，并不分配新的寄存器窗口：

```
⟨SPARC clobber 373⟩+≡                                   375 366
  case DIVI: case MODI: case MULI:
  case DIVU: case MODU: case MULU:
     spill(0x00003e00, IREG, p); break;
```

二元浮点指令只接受寄存器：

```
⟨SPARC rules 367⟩+≡                                     375 376 363
  reg: ADDD(reg,reg)  "faddd %%f%0,%%f%1,%%f%c\n"  1
  reg: ADDF(reg,reg)  "fadds %%f%0,%%f%1,%%f%c\n"  1
  reg: DIVD(reg,reg)  "fdivd %%f%0,%%f%1,%%f%c\n"  1
  reg: DIVF(reg,reg)  "fdivs %%f%0,%%f%1,%%f%c\n"  1
  reg: MULD(reg,reg)  "fmuld %%f%0,%%f%1,%%f%c\n"  1
  reg: MULF(reg,reg)  "fmuls %%f%0,%%f%1,%%f%c\n"  1
  reg: SUBD(reg,reg)  "fsubd %%f%0,%%f%1,%%f%c\n"  1
  reg: SUBF(reg,reg)  "fsubs %%f%0,%%f%1,%%f%c\n"  1
```

大多数浮点一元操作符都是类似的：

```
⟨SPARC rules 367⟩+≡                                     376 376 363
  reg: NEGF(reg)   "fnegs %%f%0,%%f%c\n"  1
  reg: LOADF(reg)  "fmovs %%f%0,%%f%c\n"  1
  reg: CVDF(reg)   "fdtos %%f%0,%%f%c\n"  1
  reg: CVFD(reg)   "fstod %%f%0,%%f%c\n"  1
```

双精度数与整数之间的转换各需要3条指令，因为必要的转换指令只使用浮点寄存器（即使整型操作数也是如此）。fdtoi把一个双精度数转换为整数，但是把结果放在一个浮点寄存器里。CVDI的父节点需要一个通用寄存器，所以模板用存储器中的临时单元把结果复制到通用寄存器里，这是唯一可行的方法：

```
⟨SPARC rules 367⟩+≡                                     376 376 363
  reg: CVDI(reg)  "fdtoi %%f%0,%%f0; st %%f0,[%%sp+64]; _
        ld [%%sp+64],%%%c\n"  3
```

CVID的过程与CVDI相反：

```
⟨SPARC rules 367⟩+≡                                     376 377 363
  reg: CVID(reg)  "st %%%0,[%%sp+64]; ld [%%sp+64],%%f%c; _
        fitod %%f%c,%%f%c\n"  3
```

CVDI 和 CVID 需要使用预留给被调用程序的单元，被调用程序用该单元来保存返回块结构的地址。这个单元只用在 call 指令的分支延迟槽以及被调用程序的分配新栈帧的函数头代码指令之间。而 CVDI 和 CVID 不可能出现在这样的间隔里。

浮点比较指令在分支之后有一个延迟槽，比较之后也有一个延迟槽：

```
⟨SPARC rules 367⟩+≡                                          376 377 363
  rel: EQD(reg,reg)  "fcmped %%f%0,%%f%1; nop; fbue"
  rel: EQF(reg,reg)  "fcmpes %%f%0,%%f%1; nop; fbue"
  rel: GED(reg,reg)  "fcmped %%f%0,%%f%1; nop; fbuge"
  rel: GEF(reg,reg)  "fcmpes %%f%0,%%f%1; nop; fbuge"
  rel: GTD(reg,reg)  "fcmped %%f%0,%%f%1; nop; fbug"
  rel: GTF(reg,reg)  "fcmpes %%f%0,%%f%1; nop; fbug"
  rel: LED(reg,reg)  "fcmped %%f%0,%%f%1; nop; fbule"
  rel: LEF(reg,reg)  "fcmpes %%f%0,%%f%1; nop; fbule"
  rel: LTD(reg,reg)  "fcmped %%f%0,%%f%1; nop; fbul"
  rel: LTF(reg,reg)  "fcmpes %%f%0,%%f%1; nop; fbul"
  rel: NED(reg,reg)  "fcmped %%f%0,%%f%1; nop; fbne"
  rel: NEF(reg,reg)  "fcmpes %%f%0,%%f%1; nop; fbne"

  stmt: rel  "%0 %a; nop\n"  4
```

有些操作符不能由固定的汇编程序模板实现，而必须通过 emit2。SPARC 结构的指令不能将双精度的寄存器值复制到另一个寄存器内，所以 lcc 为每个 LOADD 生成了两条单精度指令：

```
⟨SPARC rules 367⟩+≡                                          377 377 363
  reg:  LOADD(reg)  "# LOADD\n"  2

⟨SPARC emit2 374⟩+≡                                          375 377 374
  case LOADD: {
     int dst = getregnum(p);
     int src = getregnum(p->x.kids[0]);
     print("fmovs %%f%d,%%f%d; ", src,   dst);
     print("fmovs %%f%d,%%f%d\n", src+1, dst+1);
     break;
  }
```

NEGD 的做法与此类似。一条指令复制第一个字并改变符号位，另一条指令复制第二个字：

```
⟨SPARC rules 367⟩+≡                                          377 378 363
  reg:  NEGD(reg)  "# NEGD\n"  2

⟨SPARC emit2 374⟩+≡                                          377 378 374
  case NEGD: {
     int dst = getregnum(p);
     int src = getregnum(p->x.kids[0]);
     print("fnegs %%f%d,%%f%d; ", src,   dst);
     print("fmovs %%f%d,%%f%d\n", src+1, dst+1);
     break;
  }
```

最后，emit2 调用 blkcopy 生成代码，以复制存储器块：

```
⟨SPARC rules 367⟩+≡                                    ↑377  363
  stmt:  ASGNB(reg,INDIRB(reg))  "# ASGNB\n"

⟨SPARC emit2 374⟩+≡                                    ↑377  374
  case ASGNB: {
     static int tmpregs[] = { 1, 2, 3 };
     dalign = salign = p->syms[1]->u.c.v.i;
     blkcopy(getregnum(p->x.kids[0]), 0,
             getregnum(p->x.kids[1]), 0,
             p->syms[0]->u.c.v.i, tmpregs);
     break;
  }
```

图 13-4 跟踪显示了 MIPS 机器的“块复制生成器”的运行情况。SPARC 的代码只是在外观上与 MIPS 不同。SPARC 指令组没有无符号数的存取指令，这一点颇有争议，因为图中的例子也没使用 MIPS 的存取指令。回顾前面的 salign、dalign 和 x.max_unaligned_load，它们合起来甚至能复制无符号块，所以与目标机器相关的代码可忽略这种复杂性。g- 寄存器当前未被使用，因此 g1-g3 可做临时寄存器。MIPS 的代码比较难处理，因为它的约定很难做到一次获得大量寄存器。

在 SPARC 结构中 wants_argb 为 0，所以 emit2 省去了 ARGB 分支。

17.3 函数的实现

编译前端调用 local 通知新的局部变量。与其他目标机器上的对应操作一样，SPARC 的函数 local 调用 askregvar 尽可能地把局部变量存入寄存器，如果 askregvar 不能做到这一点，它将调用 mkauto：

```
⟨SPARC functions 364⟩+≡                          ↑374 379  363
  static void local(p) Symbol p; {
     ⟨structure return block? 379⟩
     ⟨put even lightly used locals in registers 378⟩
     if (askregvar(p, rmap[ttob(p->type)]) == 0)
        mkauto(p);
  }
```

通常情况下，除非前端估计某个局部变量将使用 3 次以上，否则它不会将该局部变量的 sclass 设置为 REGISTER。这一规定使得一些很少使用的局部变量仍留在存储器中，对于这些很少使用的局部变量，前端无法判断是否应该在函数头代码中溢出它们的寄存器，以及在函数尾代码重新载入。然而，SPARC 的寄存器窗口自动为局部变量指派通用寄存器，所以即使局部变量只用到一两次，我们的代码也可能使用寄存器变址：

```
⟨put even lightly used locals in registers 378⟩=                    378
  if (isscalar(p->type) && !p->addressed && !isfloat(p->type))
     p->sclass = REGISTER;
```

SPARC 代码生成器将接口标志 wants_callb 置为 1，以匹配 SPARC 关于返回结构的约定。wants_callb 等于 1 时，编译前端执行下面 3 个操作：

1. 生成 CALLB 节点，以调用返回结构的函数。

2. 把每个 CALLB 的第二个子节点设置成一个计算块地址的节点。这些块地址将用来存储被调用程序的返回结构。

3. 在每次返回前执行 ASGNB，将返回的子节点标识的存储块复制到由第一个局部变量标识的存储块。

与其他局部变量一样，前端也通知这个局部变量，后端将这个局部变量存入为返回的块结构而预留的栈空间：

```
⟨structure return block?379⟩≡                                    378
  if (retstruct) {
     p->x.name = stringd(4*16);
     p->x.offset = 4*16;
     retstruct = 0;
     return;
  }
```

如果当前函数返回一个结构或联合，那么 function 将 retstruct 设置成 1。

前端调用接口程序 function 通知每个新的例程。function 驱动大部分的后端程序。function 调用 gencode，gencode 通过 gen 调用标记程序、化简程序、线性化程序及寄存器分配器。function 也调用前端的 emitcode，该函数调用后端的产生程序。SPARC 的前端同样传递给 function 4 个参数：符号，表示新例程；两个符号向量，分别表示调用程序和被调用程序的参数视图；计数器，记录由该新例程引发的调用的次数。

```
⟨SPARC functions 364⟩+≡                                  378 383  363
  static void function(f, caller, callee, ncalls)
  Symbol f, callee[], caller[]; int ncalls; {
     int autos = 0, i, leaf, reg, varargs;

     ⟨SPARC function 379⟩
  }
```

leaf 标记简单的叶例程，varargs 标记可变参数例程，autos 记录存储器中的参数数目，varargs 和 autos 都可用于计算 leaf。只有 varargs 可立即算出：

```
⟨SPARC function379⟩≡                                          379  379
  for (i = 0; callee[i]; i++)
     ;

  varargs = variadic(f->type)
     || i > 0 && strcmp(callee[i-1]->name,
        "__builtin_va_alist") == 0;
```

SPARC 的约定允许声明例程是可变参数例程，或者使用宏将最后一个参数命名为 __builtin_va_alist，来表明例程是可变参数例程。

function 清除后端的关于正在使用的寄存器的记录：

```
⟨SPARC function 379⟩+≡                                   379 380  379
  ⟨clear register state 319⟩
  for (i = 0; i < 8; i++)
     ireg[i]->x.regnode->vbl = NULL;
```

上面的 for 循环在 MIPS 的代码产生器中没有相对应的部分。因为在 MIPS 中，lcc 不会给参数寄存器分配局部变量，但是在 SPARC 中可以，所以在 SPARC 中，x.regnode->vbl 可能保存了一些代码生成器编译的上一个例程遗留的垃圾。

offset 最初保存了这个例程中下一个形参的帧偏移量。function 初始化 offset，每个帧至少包含一个字，该字记录了用于返回结构的函数的目标块的地址，当寄存器窗口必须溢出时，每个帧还应加上 16 个字以保存 i0 ~ i7 和 l0 ~ l7：

```
⟨SPARC function 379⟩+≡                                    379 381  379
  offset = 68;
```

maxargoffset 保存输出参数所占栈块的大小。function 至少要为 o0 ~ o5 预留空间：

```
⟨SPARC function 379⟩+≡                                    380 380  379
  maxargoffset = 24;
```

函数头代码将输入的浮点参数存入 function 预留的空间，因为浮点参数很少通过 i- 寄存器进行传递。面对一些可变参数的函数（如 printf），这个空间存储了所有输入参数寄存器，因为在它们的函数头代码中并不知道需要多少参数，并且必须在运行时通过地址计算存取参数，所需的寄存器数在编译时不能确定。

function 为每个输入参数指定 i- 寄存器或栈偏移量。在下面的 for 循环中，每次迭代开始时，offset 都存有为下一个参数预留的栈偏移量；如果参数通过寄存器传递，则 reg 存有下一个参数的寄存器号或寄存器对号。栈按照 4 字节对齐，所以我们在对每个参数进行处理之前将舍入参数的大小使之成为 4 的倍数。参数将消耗栈空间中 size 大小的字节和 size/4 个寄存器。这种方式不包括结构参数，结构参数以引用方式传递，只消耗一个 i- 寄存器。

```
⟨SPARC function 379⟩+≡                                    380 381  379
  reg = 0;
  for (i = 0; callee[i]; i++) {
     Symbol p = callee[i], q = caller[i];
     int size = roundup(q->type->size, 4);
     ⟨classify SPARC parameter 380⟩
     offset += size;
     reg += isstruct(p->type) ? 1 : size/4;
  }
```

因为 wants_argb 为 0，function 能够只关注标量形参。

如果参数是一个浮点值，或者参数寄存器已经用完，那么该参数将会存入存储器。例程需要一个存储器段来存储这个参数：

```
⟨classify SPARC parameter 380⟩≡                                381  380
  if (isfloat(p->type) || reg >= 6) {
     p->x.offset = q->x.offset = offset;
     p->x.name = q->x.name = stringd(offset);
     p->sclass = q->sclass = AUTO;
     autos++;
  }
```

第一种情况下，如果参数在寄存器内，由于 lcc 的中间代码不可能保存整数寄存器中的浮点值，function 将自己生成代码来保存参数，前端对此无能为力。

如果参数是整型并且已在 i- 寄存器内，而且例程中会使用它的地址或者该例程为可变参数例程，那么它仍需存入存储器：

```
(classify SPARC parameter 380)+≡                    380 381 380
  else if (p->addressed || varargs)
     (arrives in an i-register, belongs in memory 381)

(arrives in an i-register, belongs in memory 381)=                  381
  {
     p->x.offset = offset;
     p->x.name = stringd(p->x.offset);
     p->sclass = AUTO;
     q->sclass = REGISTER;
     askregvar(q, ireg[reg]);
     autos++;
  }
```

function 将调用程序和被调用程序的 sclass 域设置为不同值，这样前端将产生一个赋值来存储寄存器。

对于通过寄存器传递的参数，如果它是原型参数，且所在例程不使用它的地址，例程也不是可变参数例程，那么该参数将驻留在寄存器内：

```
(classify SPARC parameter 380)+≡                    381 380
  else {
     p->sclass = q->sclass = REGISTER;
     askregvar(p, ireg[reg]);
     q->x.name = p->x.name;
  }
```

现在 function 调用前端的 gencode，gencode 调后端的 gen。function 首先清空 offset，表示没有局部变量被指派到栈上；接着 function 清空 maxoffset 以跟踪记录 offset 的最大值。因为 local 必须对它的第一个局部变量做特殊处理，故 function 对每个返回聚合类型的函数进行标记：

```
(SPARC data 365)+≡                                  365 385 362
  static int retstruct;

(SPARC function 379)+≡                              380 381 379
  offset = maxoffset = 0;
  retstruct = isstruct(freturn(f->type));
  gencode(caller, callee);
```

gencode 完成第一遍代码生成并且返回后，function 就能够计算出帧和参数构建块的大小，输出参数排列在该区域中。参数构建块的大小必须是 4 的倍数，否则有些段就不能对齐。帧大小必须是 8 的倍数，它包括局部变量的存储空间、参数构建区空间以及存放 l0 ~ l7 和 i0 ~ i7 的 16 个字，另外用一个字存放任意聚合返回块的地址：

```
(SPARC function 379)+≡                              381 381 379
  maxargoffset = roundup(maxargoffset, 4);
  framesize = roundup(maxoffset + maxargoffset + 4*(16+1), 8);
```

对于不需要新帧或者寄存器窗口的例程，function 不给它们分配新帧或寄存器窗口，这样可以节省处理时间：

```
(SPARC function 379)+≡                              381 382 379
  leaf = ((is this a simple leaf function? 382));
```

成为叶例程的限制有很多，首先该例程不会调用其他函数：

```
⟨is this a simple leaf function?382⟩=                    382 381
  !ncalls
```

存储器中没有局部变量或形参：

```
⟨is this a simple leaf function?382⟩+=                   382 382 381
  && !maxoffset && !autos
```

函数不返回结构，因为这样的函数将使用帧指针存取含有返回块位置信息的单元：

```
⟨is this a simple leaf function?382⟩+=                   382 382 381
  && !isstruct(freturn(f->type))
```

没有寄存器需要保存：

```
⟨is this a simple leaf function?382⟩+=                   382 381
  && !(usedmask[IREG]&0x00ffff01)
  && !(usedmask[FREG]&-(unsigned)3)
```

这意味着该例程的参数将限制在输入参数寄存器 o0 – o7 中。例程既不必调试也不必做性能分析，本书省略了这部分检查。如果所有这些条件都满足，那么该例程可以不需要帧。

所有的函数头代码都从公用的模板文件开始：

```
⟨SPARC function379⟩+=                                    381 382 379
  print(".align 4\n.proc 4\n%s:\n", f->x.name);
```

大多数函数头代码紧跟着一个 save 指令分配新的寄存器窗口，并将一个寄存器或常量与另一个寄存器相加。save 指令最常见的用法是把一个负值常量加到 sp 上，实现在向下增长的栈中分配新的帧：

```
⟨SPARC function379⟩+=                                    382 383 379
  if (leaf) {
     ⟨emit leaf prologue382⟩
  } else if (framesize <= 4095)
     print("save %%sp,%d,%%sp\n", -framesize);
  else
     print("set %d,%%g1; save %%sp,%%g1,%%sp\n", -framesize);
```

如果常量不能存入 SPARC 立即数域，函数头代码将首先计算该常量，并存入寄存器 g1。

符合叶优化的例程无须函数头代码，但在这种情况下，代码生成器仍将参数、局部变量和临时变量存入 i- 寄存器。如果我们已经决定不生成帧和寄存器窗口，就应该用相应的 o- 寄存器代替 i- 寄存器。lcc 的后端没有专门的成批重命名功能，所以最好的解决方法是：function 临时改变那些保存每个 i- 寄存器的名字和编号的结构，重新命名为 o- 寄存器。具体操作过程是：从 caller 的参数向量入手，function 在初始 for 循环中把 i- 寄存器的名字复制到参数的 x.name 域，接着修正 x.name 域：

```
⟨emit leaf prologue382⟩=                                 382
  for (i = 0; caller[i] && callee[i]; i++) {
     Symbol p = caller[i], q = callee[i];
     if (p->sclass == REGISTER && q->sclass == REGISTER)
        p->x.name = greg[q->x.regnode->number - 16]->x.name;
     }
  rename();
```

函数 rename 完成余下的工作：

```
⟨SPARC functions 364⟩+≡                                    379 384 363
  #define exch(x, y, t) ((t) = x), ((x) = (y)), ((y) = (t)))

  static void rename() {
     int i;

     for (i = 0; i < 8; i++) {
        char *ptmp;
        int itmp;
        if (ireg[i]->x.regnode->vbl)
           ireg[i]->x.regnode->vbl->x.name = oreg[i]->x.name;
        exch(ireg[i]->x.name, oreg[i]->x.name, ptmp);
        exch(ireg[i]->x.regnode->number,
           oreg[i]->x.regnode->number, itmp);
     }
  }
```

rename 交换对应的 i- 寄存器和 o- 寄存器的名字和编号，function 结尾处通过另一个交换来恢复函数开始时的样子。如果寄存器分配器已将寄存器指派给了变量，rename 也要修正此变量在符号结构中记录的名字。由于 rename 导致的改变在当前例程的结尾处必须还原，因此 rename 用交换来实现；但是由于在当前例程之外，caller 和寄存器变量不再有用，因此，它们的改变可用简单赋值实现。

function 下一步生成函数头代码以保存已存在于寄存器中但不能驻留的参数。由于函数头代码并不知道可变参数的例程实际包含的参数数目，因此可变参数的例程必须保存 i0 – i5：

```
⟨SPARC function 379⟩+≡                                     382 384 379
  if (varargs)
     for (; reg < 6; reg++)
        print("st %%i%d,[%%fp+%d]\n", reg, 4*reg + 68);
  else
     ⟨spill floats and doubles from i0-i5 383⟩
```

函数头代码也保存通用寄存器内的浮点值，因为指令不能在通用寄存器中对浮点值进行处理：

```
⟨spill floats and doubles from i0-i5 383⟩≡                 383
  offset = 4*(16 + 1);
  reg = 0;
  for (i = 0; caller[i]; i++) {
     Symbol p = caller[i];
     if (isdouble(p->type) && reg <= 4) {
        print("st %%r%d,[%%fp+%d]\n",
              ireg[reg++]->x.regnode->number, offset);
        print("st %%r%d,[%%fp+%d]\n",
              ireg[reg++]->x.regnode->number, offset + 4);
     } else if (isfloat(p->type) && reg <= 5)
        print("st %%r%d,[%%fp+%d]\n",
           ireg[reg++]->x.regnode->number, offset);
     else
        reg++;
     offset += roundup(p->type->size, 4);
  }
```

isfloat处理浮点数和双精度数，所以上面的第一个else子句不仅保存浮点数也将双精度数的前半部分保存在i5中，后半部分已经在存储器里了，调用程序通常这样做。

最后，function产生性能分析代码（这里没有显示）以及当前例程的函数体和函数尾代码。函数尾代码一般由ret指令和restore指令组成，ret指令返回到调用程序，restore指令在ret的延迟槽内，用于还原函数头代码中的save指令。如果例程没有用到寄存器窗口和栈帧，那么也就没必要还原save指令了，但是需要另一个rename函数还原前面被改动的i-寄存器的名字和编号：

```
⟨SPARC function 379⟩+≡                                    ↑383  379
  ⟨emit profiling code⟩
  emitcode();
  if (!leaf)
     print("ret; restore\n");
  else {
     rename();
     print("retl; nop\n");
  }
```

伪指令ret和retl都使用含返回地址的寄存器来生成间接分支指令。由于没有压栈操作，ret必须使用i7，retl必须使用o7命名同一个寄存器，所以ret、retl需要有不同的名字。

17.4 数据的定义

SPARC中的defconst、defaddress、defstring和address与MIPS对应的实例相同。参见第16章中的代码。

前端调用export将本模块的符号输出给其他模块，这是SPARC汇编程序指示符.global的作用：

```
⟨SPARC functions 364⟩+≡                                   ↑383 384↓  363
  static void export(p) Symbol p; {
   print(".global %s\n", p->x.name);
}
```

前端调用import使得当前模块能访问其他模块中定义的符号。SPARC汇编器假设未定义的符号都是外部符号，所以SPARC的import不做任何工作：

```
⟨SPARC functions 364⟩+≡                                   ↑384 384↓  363
  static void import(p) Symbol p; {}
```

前端调用defsymbol通知新的符号，并提示后端初始化x.name域。SPARC的约定要为每个局部静态符号命名，在以后的程序中就用这个名字。SPARC链接编辑器将L开头的符号从符号表中删除，所以defsymbol将L放在生成的符号之前。对于其他的符号则在它们之前加下划线，这也是SPARC的约定：

```
⟨SPARC functions 364⟩+≡                                   ↑384 385↓  363
  static void defsymbol(p) Symbol p; {
     if (p->scope >= LOCAL && p->sclass == STATIC)
        p->x.name = stringf("%d", genlabel(1));
     else
        p->x.name = p->name;
```

```
    if (p->scope >= LABELS)
        p->x.name = stringf(p->generated ? "L%s" : "_%s",
            p->x.name);
}
```

具有文件作用域的静态符号保留了它们的名字。该套块中的静态符号使用编号标识以避免与其他例程的静态符号冲突。

接口例程 segment 产生 .seg"name"，表示一个新段的开始：

```
⟨SPARC functions 364⟩+≡                                    384 385 363
  static void segment(n) int n; {
      cseg = n;
      switch (n) {
      case CODE: print(".seg \"text\"\n"); break;
      case BSS:  print(".seg \"bss\"\n");  break;
      case DATA: print(".seg \"data\"\n"); break;
      case LIT:  print(".seg \"text\"\n"); break;
      }
  }
```

接口例程 space 需要 segment 将当前段保存在 cseg 中，space 生成汇编程序指示符 .skip 为已初始化的全局或静态符号预留 n 字节的空间：

```
⟨SPARC data 365⟩+≡                                         381 362
  static int cseg;

⟨SPARC functions 364⟩+≡                                    385 385 363
  static void space(n) int n; {
      if (cseg != BSS)
          print(".skip %d\n", n);
  }
```

根据标准的要求，.skip 清空它所分配的空间。

如果是在 BSS 程序段，接口程序 global 可以定义标号，并预留所需空间，外部符号用 .common，其他符号用 .reserve：

```
⟨SPARC functions 364⟩+≡                                    385 386 363
  static void global(p) Symbol p; {
      print(".align %d\n", p->type->align);
      if (p->u.seg == BSS
      && (p->sclass == STATIC || Aflag >= 2))
          print(".reserve %s,%d\n", p->x.name, p->type->size);
      else if (p->u.seg == BSS)
          print(".common %s,%d\n",  p->x.name, p->type->size);
      else
          print("%s:\n", p->x.name);
  }
```

global 也可以生成一个对齐指示符，对于已初始化的全局名还产生标记。.common 输出符号，并标识它们，这样即使其他模块用 .common 命令输出同样的符号，装配程序也只生成一个公用的全局名。.reserve 的做法不同，静态名使用 .reserve 以避免输出。如果使用了编译选项 -A-A（即 Aflag>=2），对于全局符号，global 也使用 .reserve。当多个模块定义了同一个全局名时，装配程

序将产生警告信息。ANSI 前的 C 允许重复定义，但从技术上说 ANSI C 要求单一定义，其他模块应使用 extern 声明相同的变量。

17.5 块的复制

blkfetch 产生代码读取内存块，即产生代码首先将寄存器 reg 和偏移量 off 相加得到地址，然后从相应地址单元中取出 k 个字节的数据载入寄存器 tmp。k 可以取至 1、2 或 4：

```
⟨SPARC functions 364⟩+≡                                    385 386  363
  static void blkfetch(k, off, reg, tmp)
  int k, off, reg, tmp; {
     if (k == 1)
        print("ldub [%%r%d+%d],%%r%d\n", reg, off, tmp);
     else if (k == 2)
        print("lduh [%%r%d+%d],%%r%d\n", reg, off, tmp);
     else
        print("ld [%%r%d+%d],%%r%d\n",   reg, off, tmp);
  }
```

SPARC 指令只载入对齐后的值，所以 blkfetch 无须判断载入的值是否对齐，但 MIPS 的 blkfetch 就要做这个判断。blkunroll 用 x.max_unaligned_load 选取了一个块大小，并且保证对齐字节数不小于块大小。blkfetch 只能选择读取 8 位字节、16 位半字或 32 位字。blkstore 与 blkfetch 的做法一样：

```
⟨SPARC functions 364⟩+≡                                    386 386  363
  static void blkstore(k, off, reg, tmp)
  int k, off, reg, tmp; {
     if (k == 1)
        print("stb %%r%d,[%%r%d+%d]\n", tmp, reg, off);
     else if (k == 2)
        print("sth %%r%d,[%%r%d+%d]\n", tmp, reg, off);
     else
        print("st %%r%d,[%%r%d+%d]\n",  tmp, reg, off);
  }
```

所有 SPARC 的 blk 过程都使用类似 r9 的通用寄存器名。如果我们试图像在其他地方一样，在 blk 中使用以 g、i、l、o 为首字母的名字，那么就需要改变 blk 程序之间的接口，以直接传送寄存器名而不是编号，但这也增加了不少麻烦，例如直接导致 lcc 二进制目标代码的产生变得更加复杂。

blkloop 产生循环从源地址（寄存器 sreg 和偏移量 soff 之和）复制 size 大小的字节到目的地址（寄存器 dreg 和偏移量 doff 之和）：

```
⟨SPARC functions 364⟩+≡                                    386  363
  static void blkloop(dreg, doff, sreg, soff, size, tmps)
  int dreg, doff, sreg, soff, size, tmps[]; {
     ⟨SPARC blkloop 387⟩
  }
```

tmps 给 3 个寄存器命名以用作临时寄存器。每次迭代复制 8 个字节。初始化时将 sreg 指向源块的末端，tmps[2] 指向目标块的末端。这个程序段有两个子程序块。如果块大小能够用有符号的 13 个有效数位的域表示，则可以直接处理。否则，先计算块的大小并存入寄存器 tmps[2] 中，然后把该寄存器加到输入源地址和目的地址上。

```
⟨SPARC blkloop 387⟩=                                    387  386
  if ((size&-7) < 4096) {
     print("add %%r%d,%d,%%r%d\n", sreg, size&~7, sreg);
     print("add %%r%d,%d,%%r%d\n", dreg, size&~7, tmps[2]);
  } else {
     print("set %d,%%r%d\n", size&-7, tmps[2]);
     print("add %%r%d,%%r%d,%%r%d\n", sreg, tmps[2], sreg);
     print("add %%r%d,%%r%d,%%r%d\n", dreg, tmps[2], tmps[2]);
  }
```

如果块的大小不是 8 的倍数，则第一个 blkcopy 将复制除以 8 后的余数字节：

```
⟨SPARC blkloop 387⟩+=                                   387 387  386
  blkcopy(tmps[2], doff, sreg, soff, size&7, tmps);
```

循环每次迭代都会使寄存器 sreg 和 tmps[2] 减 8。tmps[2] 立即减 8，而 sreg 的递减要推迟到循环末端的分支延迟槽中：

```
⟨SPARC blkloop 387⟩+=                                   387 387  386
  print("1: dec 8,%%r%d\n", tmps[2]);
```

tmps[2] 减 8 后，循环调用 blkcopy，从源地址处复制 8 个字节到目的地址。由于 sreg 不能现在递减，因此只能调节源地址偏移量，blkcopy 的第 4 个参数使用 soff-8：

```
⟨SPARC blkloop 387⟩+=                                   387 387  386
  blkcopy(tmps[2], doff, sreg, soff - 8, 8, tmps);
```

最后，如果还有剩余字节则循环继续进行：

```
⟨SPARC blkloop 387⟩+=                                   387  386
  print("cmp %%r%d,%%r%d; ", tmps[2], dreg);

  print("bgt 1b; ");
  print("dec 8,%%r%d\n", sreg);
```

深入阅读

SPARC 参考手册详细说明了这种机器的结构（SPARC International，1992）。Patterson and Hennessy（1990）解释了各种延迟槽的作用。Krishnamurthy（1990）综述了有关指令在延迟槽中调度的文献。

练习

17.1 添加一个标记，使后端不必通过函数调用而直接生成指令来对有符号和无符号整数做乘法。

17.2 改写 lcc 的 SPARC 代码生成器使之能更好地利用寄存器 g1 – g7，并将浮点变量保存在浮点寄存器中。回顾前面的调用约定，所有已编译的库例程都没有维持这些寄存器。

17.3 找出无条件跳转后的延迟槽的一些用法。例如，可以将跳转的目标指令复制到延迟槽内，跳转的目标

可以改写为原目标指令的下一条指令；有些优化程序需要缓冲代码，并用额外的优化遍来处理它们。MIPS R3000也有这样的延迟槽，但是标准的汇编程序会重新对指令进行排序并用更有用的指令填充延迟槽，所以在MIPS R3000上，我们可以忽略这个问题。

17.4 找出条件分支后的延迟槽的一些用法。比如设计一个取消位（annul bit），它说明了延迟槽中的指令没有作用，除非该分支是条件分支，而且条件成立。在操作码中添加a来设置取消位（例如be，a L4）。

17.5 对于某些SPARC的处理器，若一条load指令恰好在要用该值的指令之前，必然造成至少一个时钟周期的处理器延迟，其执行速度与目标代码在load之后再执行一个空操作一样（虽然要占用一个字长）。对已生成的汇编指令重新排序以消除一些延迟。Proebsting and Fischer（1991）描述了一种指令排序方案。

17.6 一些叶例程尽管不需要寄存器窗口，但由于需要一个新指针，所以仍然没有进行叶优化。例如，一些返回结构的函数不需要窗口，但要用到帧指针。修改lcc，使这些例程仅生成一个帧而不生成寄存器窗口。

17.7 SPARC的代码生成器包含一些特殊代码，它们保证了当寄存器全都分配完时，溢出程序可以生成代码对其中一个寄存器进行保存。设计一个小的测试程序来跟踪这段代码。

X86 代码的生成

为了说明 Intel 386 结构的代码生成具有兼容性，本书统一使用 X86 代表与其兼容的机器。X86 包括 Intel 486 和奔腾系列，以及 AMD 和 Cyrix 生产的系列兼容机。X86 的 lburg 规范采用近似的 Intel 486 的指令开销周期。对于与 486 兼容的结构而言，大部分情况下可以得到最优结果，但也不是绝对的。许多不必要的开销被忽略掉了。例如，如果一条规则只与某一个操作符匹配，那么就没必要花费代价打破它们的导出关系。

X86 结构属于 CISC，即复杂指令集机器。它包含大量变长指令和寻址模式。X86 结构有 8 个 32 位通用寄存器和 8 个 80 位浮点寄存器，浮点寄存器组成一个浮点栈。

X86 结构的 C 语言编译器有很多种，这些编译器使用的约定各不相同（比如函数调用规则和返回值规则）。本章采用的代码生成器遵循 Borland C++ 4.0 的约定，它使用 Borland 的标准头文件、库和链接器，而使用其他 X86 环境的 lcc 可能需要做些改动。

各种 X86 汇编程序使用的语法不同。lcc 采用微软的 MASM 6.11 和 Borland 的 Turbo Assembler 4.0，所以 lcc 生成的代码上述两种汇编程序都能接受。这两种汇编程序的指令都将目的操作数放在源操作数之前，而且使用寄存器名而不是寄存器号。表 18-1 描述了部分指令的示例。

表 18-1 X86 汇编器输入样例

汇编指令	意义
mov al, byte ptr 8	将寄存器 al 的值设置为地址为 8 的字节单元的值
mov dword ptr 8 [edi*4],1	将地址为寄存器 edi 的值乘以 4 再加上 8 的 32 位字的值设置为 1
subu eax, 7	将寄存器 eax 的值减去 7
fsub qword ptr x	将浮点栈栈顶的值减去标号为 x 的单元中的双精度浮点值
jmp L1	跳转到标号为 L1 的指令处
cmp dword ptr x, 7	将标号为 x 的 32 位字的值与 7 进行比较，比较结果保存在条件标志中
jl L1	如果上次比较结果为小于，转移到标号 L1 处
dword 0208H	将内存中下一个字节单元初始化成十六进制数 20

文件 x86.c 包含了所有与 X86 相关的代码和数据。下面是 lburg 规范，其中接口例程在文法的后面：

```
⟨x86.md 389⟩≡
  %{
  ⟨X86 macros 390⟩
  ⟨lburg prefix 293⟩
  ⟨interface prototypes⟩
  ⟨X86 prototypes⟩
  ⟨X86 data 391⟩
  %}
  ⟨terminal declarations 293⟩
  %%
  ⟨shared rules 312⟩
```

```
⟨X86 rules 395⟩
%%
⟨X86 functions 390⟩
⟨X86 interface definition 390⟩
```

最后的程序段配置前端，并指向后端的关于 X86 的例程和数据：

```
⟨X86 interface definition 390⟩=                                    390
  Interface x86IR = {
      1, 1, 0,  /* char */
      2, 2, 0,  /* short */
      4, 4, 0,  /* int */
      4, 4, 1,  /* float */
      8, 4, 1,  /* double */
      4, 4, 0,  /* T * */
      0, 4, 0,  /* struct; so that ARGB keeps stack aligned */
      1,        /* little_endian */
      0,        /* mulops_calls */
      0,        /* wants_callb */
      1,        /* wants_argb */
      0,        /* left_to_right */
      0,        /* wants_dag */
      ⟨interface routine names⟩
      ⟨symbol-table emitters 390⟩
    {1, ⟨Xinterface initializer 277⟩}
};
```

MIPS 和 SPARC 的约定都是从左向右计算参数，而 X86 恰好相反，所以它的接口标志 left_to_right 为 0。

X86 的编译器可以采取不同方式为调试器对汇编程序的符号表进行编码，X86 的约定并不提供标准方式，所以 lcc X86 的后端没有符号表产生器：

```
⟨symbol-table emitters 390⟩=                                       390
  0, 0, 0, 0, 0, 0, 0,
```

18.1 寄存器

X86 结构包括 8 个通用寄存器。汇编程序对每个寄存器命名，分别是 eax、ecx、edx、ebx、esp、ebp、esi 和 edi。由于 lcc 的寄存器分配器需要一个数来计算寄存器掩码的移位距离，所以 lcc 借用了一些指令的二进制编码表示方法：

```
⟨X86 macros 390⟩=                                                  389
  enum { EAX=0, ECX=1, EDX=2, EBX=3, ESI=6, EDI=7 };
```

X86 的约定要求预留 ebp 保存帧指针，预留 esp 保存栈指针，所以 lcc 不分配这两个寄存器。

progbeg 建立了描述寄存器的结构：

```
⟨X86 functions 390⟩=                                          393  390
  static void progbeg(argc, argv) int argc; char *argv[]; {
      int i;

      ⟨shared progbeg 290⟩
      parseflags(argc, argv);
```

```
    intreg[EAX] = mkreg("eax", EAX, 1, IREG);
    intreg[EDX] = mkreg("edx", EDX, 1, IREG);
    intreg[ECX] = mkreg("ecx", ECX, 1, IREG);
    intreg[EBX] = mkreg("ebx", EBX, 1, IREG);
    intreg[ESI] = mkreg("esi", ESI, 1, IREG);
    intreg[EDI] = mkreg("edi", EDI, 1, IREG);
    ⟨X86 progbeg 391⟩
}
```

汇编程序代码采用不同的名字对全 32 位寄存器以及它的低 8 位和低 16 位子寄存器命名。例如，汇编程序代码使用 eax 命名第一个 32 位寄存器，使用 ax 命名该寄存器的低 16 位子寄存器，使用 al 命名它的末尾字节。这个规则要求为短整数和字符初始化单独的寄存器向量：

```
⟨X86 data 391⟩=                                              393  389
  static Symbol charreg[32], shortreg[32], intreg[32];
  static Symbol fltreg[32];

⟨X86 progbeg 391⟩=                                           391  391
  shortreg[EAX] = mkreg("ax", EAX, 1, IREG);
  shortreg[ECX] = mkreg("cx", ECX, 1, IREG);
  shortreg[EDX] = mkreg("dx", EDX, 1, IREG);
  shortreg[EBX] = mkreg("bx", EBX, 1, IREG);
  shortreg[ESI] = mkreg("si", ESI, 1, IREG);
  shortreg[EDI] = mkreg("di", EDI, 1, IREG);

⟨X86 progbeg 391⟩+=                                          391 392  391
  charreg[EAX]  = mkreg("al", EAX, 1, IREG);
  charreg[ECX]  = mkreg("cl", ECX, 1, IREG);
  charreg[EDX]  = mkreg("dl", EDX, 1, IREG);
  charreg[EBX]  = mkreg("bl", EBX, 1, IREG);
```

由于没有指令寻址 esi 和 edi 的末尾字节，所以汇编程序并未对它们的低 8 位子寄存器命名。字节指令能寻址每个 16 位寄存器的高 8 位，但用这些字节寄存器会使代码生成过程更为复杂，因此 lcc 没有使用它们。例如，如果操作数处在低位字节，则 CVCI 需要生成一个指令序列；当操作数在高位字节上时，CVCI 需要生成另一个指令序列。表 18-2 概括了所有可分配的寄存器。

表 18-2 可分配的 X86 寄存器

Int	Short	Char
eax	ax	al
ecx	cx	cl
edx	dx	dl
ebx	bx	bl
esi	si	
edi	di	

X86 的浮点寄存器组织成一个栈。有些指令的操作数能存入任意一个浮点寄存器中（自上而下），但也有一些重要的指令要求栈式操作。例如，浮点加法指令的所有变体都需要至少一个操作数在栈顶。在汇编程序里，操作数 st 表示栈顶，st(1) 表示 st 下面的值。把一个值压入栈就是使 st 表示一个新的单元，这时 st(1) 表示的是刚才 st 所指向的单元。

lcc 采用固定名字的寄存器，因此不能随栈大小的增减而改变寄存器的名字。X86 的浮点寄存器不符合这一规定，所以 lcc 的寄存器分配器不对 X86 的浮点寄存器进行管理，而是由指令来

操作这类寄存器。例如，取指令将一个值压入栈，这样就实现了寄存器分配。加法指令一次从栈中弹出两个操作数，然后将相加的结果压入栈，显然加法指令能实现同时释放两个寄存器并重新分配一个寄存器。

通过简单地清除 rmap 中的入口，并不能禁止寄存器分配器为浮点和双精度分配寄存器。如果某个节点产生一个值，那么寄存器分配器会假定它需要一个寄存器来保存该值，并且假设节点的 sym[RX] 将给出所需寄存器的类型。所以我们需要对浮点寄存器进行表示。但是这种表示法不会破坏原有的寄存器分配器。一个简单的方法就是创建具有零掩码的寄存器，它使 getreg 总能执行成功并且不改变关键的编译器状态：

```
⟨X86 progbeg 391⟩+≡                                    391 392  391
  for (i = 0; i < 8; i++)
    fltreg[i] = mkreg("%d", i, 0, FREG);
```

这种巧妙的设计使得寄存器分配器能正常工作，但它还不是最好的。这个方法说明了可重定位编译器设计中常见的折中考虑。我们尽可能合理地把代码移入编译器中与目标机器无关的部分，并将这些部分固定下来。在设计上，我们不可能考虑到所有种类机器的全部功能部件，因而某些机器的代码生成器可能需要采取特定的工作方式，它们生成的代码也只是次优的。

rmap 存储通配符，这些通配符标识了每种数据类型的默认寄存器类型：

```
⟨X86 progbeg 391⟩+≡                                    392  392  391
  rmap[C] = mkwildcard(charreg);
  rmap[S] = mkwildcard(shortreg);
  rmap[P] = rmap[B] = rmap[U] = rmap[I] = mkwildcard(intreg);
  rmap[F] = rmap[D] = mkwildcard(fltreg);
```

tmask 和 vmask 分别标识存储临时变量和寄存器变量的寄存器。lcc 只能使用 X86 的 6 个通用寄存器，其中一些可以在函数调用、块复制和其他一些特殊指令或指令序列中溢出。如果公共子表达式太多，lcc 简单的寄存器分配器将会生成一些处理寄存器的代码，这些代码的效果就像对虚拟存储页频繁地换进换出一样。因而保守的方案是将全部 6 个通用寄存器预留给临时变量，而不是把它们分配给一般变量。

```
⟨X86 progbeg 391⟩+≡                                    392  392  391
  tmask[IREG] = (1<<EDI) | (1<<ESI) | (1<<EBX)
              | (1<<EDX) | (1<<ECX) | (1<<EAX);
  vmask[IREG] = 0;
```

lcc 对浮点寄存器的处理类似。

```
⟨X86 progbeg 391⟩+≡                                    392  392  391
  tmask[FREG] = 0xff;
  vmask[FREG] = 0;
```

progbeg 生成一些固定的指令，以便对产生的代码进行汇编和链接：

```
⟨X86 progbeg 391⟩+≡                                    392  393  391
  print(".486\n");
  print(".model small\n");
  print("extrn __turboFloat:near\n");
  print("extrn __setargv:near\n");
```

上面对外部符号的引用将指导链接程序安排一个特殊的浮点包和代码，以便在每个 main 例程中设置 argc 和 argv。

从一个段切换到另一个段需要两个指示符：ends 和 segment。前者结束当前段，后者开始一个新段：

```
⟨X86 data 391⟩+≡                                    391 399 389
  static int cseg;

⟨X86 functions 390⟩+≡                               390 393 390
  static void segment(n) int n; {
     if (n == cseg)
        return;
     if (cseg == CODE)
        print("_TEXT ends\n");
     else if (cseg == DATA || cseg == BSS || cseg == LIT)
        print("_DATA ends\n");
     cseg = n;
     if (cseg == CODE)
        print("_TEXT segment\n");
     else if (cseg == DATA || cseg == BSS || cseg == LIT)
        print("_DATA segment\n");
  }
```

export 要求指示符必须出现在两个段之间。CODE、DATA、LIT 和 BSS 都是用正值表示的，所以 export 可以用 segment(0) 关闭正在使用的段，但不打开新段。

progbeg 将 cseg 置为 0，以记录后端正位于两个段之间：

```
⟨X86 progbeg 391⟩+≡                                 392 399 391
  cseg = 0;
```

progend 产生固定指令关闭当前程序段和整个汇编程序：

```
⟨X86 functions 390⟩+≡                               393 393 390
  static void progend() {
     segment(0);
     print("end\n");
  }
```

target 记录需要特定寄存器的操作符。clobber 调用 spill 溢出并重取被少数操作符重写的扩寄存器：

```
⟨X86 functions 390⟩+≡                               393 394 390
  static void target(p) Node p; {
     switch (p->op) {
     ⟨X86 target 399⟩
     }
  }

  static void clobber(p) Node p; {
     static int nstack = 0;

     nstack = ckstack(p, nstack);
     switch (p->op) {
     ⟨X86 clobber 402⟩
     }
  }
```

上述 switch 语句中与操作符紧密相关的分支处理部分将在下一节中介绍。clobber 用 nstate 记录

浮点寄存器栈的高度。当 progbeg 不允许分配这些寄存器时，同时也禁止溢出这些寄存器，所以 X86 的代码生成器必须自己处理浮点寄存器的溢出。ckstack 调整 nstate 来记录当前指令的结果：

```
(X86 functions 390)+≡                                    393 397  390
  #define isfp(p) (optype((p)->op)==F || optype((p)->op)==D)

  static int ckstack(p, n) Node p; int n; {
     int i;

     for (i = 0; i < NELEMS(p->x.kids) && p->x.kids[i]; i++)
        if (isfp(p->x.kids[i]))
           n--;
     if (isfp(p) && p->count > 0)
        n++;
     if (n > 8)
        error("expression too complicated\n");
     return n;
  }
```

for 循环弹出源寄存器。随后的 if 语句将计算结果压栈。与赋值和条件分支一样，浮点指令作为副作用执行，没有压栈操作。当表达式过于复杂（n>8）时，ckstack 通知程序员简化表达式以避免溢出寄存器。由于这种溢出非常少，lcc 仅报告错误，这样并不激怒用户。但如果 lcc 完全忽略了这个问题，对于一些程序可能产生错误代码，这一点是不可接受的。练习 18.8 和练习 18.9 说明了相关情况。

18.2 指令的选取

表 18-3 总结了 X86 的 Iburg 规范中的非终结符。此表概括了 lcc 树文法的组织。

表 18-3 X86 的非终结符

名字	匹配对象
acon	地址常量
addr	针对内存读写指令的地址计算
addrj	针对转移指令的地址计算
base	未索引的地址计算
compf	浮点比较的操作数
con	常量
con1	整型常量 1
con2	整型常量 2
con3	整型常量 3
flt	浮点操作数
index	带索引的地址计算
mem	一般操作符使用的内存单元
memf	浮点操作符使用的内存单元
mr	内存单元和寄存器
mr0	内存单元、寄存器和内存代价为 0 的常量
mr1	内存单元、寄存器和内存代价为 1 的常量
mr4	内存单元、寄存器和内存代价为 3 的常量
rc	寄存器和常量
rc5	寄存器 ecl 和 0 到 31 之间常量（不包括 0 和 31）
reg	计算结果在寄存器中的运算
stmt	副作用产生的运算

整数和地址常量都直接表示：

```
⟨X86 rules 395⟩=                                    395  390
  acon: ADDRGP  "%a"
  acon: con     "%0"
```

基地址可以是 ADDRGP，也可以是 acon 与通用寄存器的和。汇编程序的语法规定寄存器名必须用方括号括起来：

```
⟨X86 rules 395⟩+=                                   395 395  390
  base: ADDRGP          "%a"
  base: reg             "[%0]"
  base: ADDI(reg,acon)  "%1[%0]"
  base: ADDP(reg,acon)  "%1[%0]"
  base: ADDU(reg,acon)  "%1[%0]"
```

如果寄存器是帧指针，那么计算形参地址的方法与计算局部变量的方法相同：

```
⟨X86 rules 395⟩+=                                   395 395  390
  base: ADDRFP  "%a[ebp]"
  base: ADDRLP  "%a[ebp]"
```

有些地址使用一个索引，表示寄存器乘上 1、2、4 或 8：

```
⟨X86 rules 395⟩+=                                   395 395  390
  index: reg "%0"
  index: LSHI(reg,con1)  "%0*2"
  index: LSHI(reg,con2)  "%0*4"
  index: LSHI(reg,con3)  "%0*8"

  con1:  CNSTI  "1"  range(a, 1, 1)
  con1:  CNSTU  "1"  range(a, 1, 1)
  con2:  CNSTI  "2"  range(a, 2, 2)
  con2:  CNSTU  "2"  range(a, 2, 2)
  con3:  CNSTI  "3"  range(a, 3, 3)
  con3:  CNSTU  "3"  range(a, 3, 3)
```

回顾前面，在计算代价表达式的值时，a 表示被标记的节点，在这里 a 是一个与小整数进行比较的常量值。无符号数的左移操作等价于整型移位：

```
⟨X86 rules 395⟩+=                                   395 395  390
  index: LSHU(reg,con1)  "%0*2"
  index: LSHU(reg,con2)  "%0*4"
  index: LSHU(reg,con3)  "%0*8"
```

一般地址可以是基地址或基地址与索引的和。前端将索引操作放在左边（参见 9.7 节）：

```
⟨X86 rules 395⟩+=                                   395 395  390
  addr: base              "%0"
  addr: ADDI(index,base)  "%1[%0]"
  addr: ADDP(index,base)  "%1[%0]"
  addr: ADDU(index,base)  "%1[%0]"
```

如果基地址是 0，那么索引本身就作为地址：

```
⟨X86 rules 395⟩+=                                   395 396  390
  addr: index  "[%0]"
```

许多指令接受存储器操作数。许多机器的汇编程序将数据类型编码在指令操作码中，但在这里是由类作数说明符来表示数据类型的。word 表示一个 16 位操作数，dword 表示一个 32 位类作数。

```
⟨X86 rules 395⟩+≡                                    395 396 390
  mem: INDIRC(addr)  "byte ptr %0"
  mem: INDIRI(addr)  "dword ptr %0"
  mem: INDIRP(addr)  "dword ptr %0"
  mem: INDIRS(addr)  "word ptr %0"
```

X86 的有些指令可以接受寄存器或立即数类作数，有些指令可以接受在寄存器或存储器中的操作数。有些指令 3 种形式都能接受：

```
⟨X86 rules 395⟩+≡                                    396 396 390
  rc:   reg  "%0"
  rc:   con  "%0"

  mr:   reg  "%0"
  mr:   mem  "%0"

  mrc0: mem  "%0"
  mrc0: rc   "%0"
```

最后一类中的某些指令访问存储器时无任何开销，有些指令也许要花费一个时钟周期，某些甚至花费 3 个时钟周期：

```
⟨X86 rules 395⟩+≡                                    396 396 390
  mrc1: mem  "%0"  1
  mrc1: rc   "%0"

  mrc3: mem  "%0"  3
  mrc3: rc   "%0"
```

lea 指令将地址载入寄存器，mov 指令读取寄存器、常量或存储器单元：

```
⟨X86 rules 395⟩+≡                                    396 396 390
  reg: addr          "lea %c,%0\n"  1
  reg: mrc0          "mov %c,%0\n"  1
  reg: LOADC(reg)    "mov %c,%0\n"  move(a)
  reg: LOADI(reg)    "mov %c,%0\n"  move(a)
  reg: LOADP(reg)    "mov %c,%0\n"  move(a)
  reg: LOADS(reg)    "mov %c,%0\n"  move(a)
  reg: LOADU(reg)    "mov %c,%0\n"  move(a)
```

mov 指令在访问存储器时不会造成额外开销，所以它使用 mrc0。前几章提到，代价函数 move 返回 1，而且将节点标记为“寄存器—寄存器”的复制。emit、requate 和 moveself 可以合作删除一些已被标记的指令。

如果进型加法和减法指令访问存储器，则将要花费一个时钟周期，所以使用 mrc1：

```
⟨X86 rules 395⟩+≡                                    396 397 390
  reg: ADDI(reg,mrc1)  "?mov %c,%0\nadd %c,%1\n"  1
  reg: ADDP(reg,mrc1)  "?mov %c,%0\nadd %c,%1\n"  1
  reg: ADDU(reg,mrc1)  "?mov %c,%0\nadd %c,%1\n"  1
```

```
reg: SUBI(reg,mrc1)  "?mov %c,%0\nsub %c,%1\n"  1
reg: SUBP(reg,mrc1)  "?mov %c,%0\nsub %c,%1\n"  1
reg: SUBU(reg,mrc1)  "?mov %c,%0\nsub %c,%1\n"  1
```

位操作指令也类似：

```
⟨X86 rules 395⟩+≡                                    396 397  390
  reg: BANDU(reg,mrc1)  "?mov %c,%0\nand %c,%1\n"  1
  reg: BORU(reg,mrc1)   "?mov %c,%0\nor %c,%1\n"   1
  reg: BXORU(reg,mrc1)  "?mov %c,%0\nxor %c,%1\n"  1
```

如果当前指令重用第一个子节点的目的寄存器，汇编程序模板内起始的问号标记将通知 emit 忽略模板中的第一条指令。也就是说，如果 %c 是 eax，%0 是 ebx，%1 是 ecx，那么 SUBU 模板将产生下列指令：

```
mov eax,ebx
sub eax,ecx
```

但如果 %c 是 eax，%0 是 eax，%1 是 ecx，那么该模板就产生下列指令：

```
sub eax,ecx
```

二元操作指令会破坏指令的第一操作数，所以通常开始时必须复制第一操作数到目的寄存器，但在某些情况下复制是多余的。上面所说的开销只是估计值，因为只有在分配了寄存器后，lcc 才知道是否需要 mov 指令。这时再选择指令为时已晚。这是一个经典的阶段顺序（phase-ordering）问题：编译器必须选择指令以分配寄存器，但是又必须分配寄存器后才能准确计算指令的开销。

上述二元操作有一种变体可以修改存储器单元。有些操作将另一个操作数固定为 1。例如，

```
inc dword ptr i
```

使 i 加 1。

```
⟨X86 rules 395⟩+≡                                    397 398  390
  stmt: ASGNI(addr,ADDI(mem,con1))  "inc %1\n"  memop(a)
  stmt: ASGNI(addr,ADDU(mem,con1))  "inc %1\n"  memop(a)
  stmt: ASGNP(addr,ADDP(mem,con1))  "inc %1\n"  memop(a)
  stmt: ASGNI(addr,SUBI(mem,con1))  "dec %1\n"  memop(a)
  stmt: ASGNI(addr,SUBU(mem,con1))  "dec %1\n"  memop(a)
  stmt: ASGNP(addr,SUBP(mem,con1))  "dec %1\n"  memop(a)
```

单个操作数同时标识源操作数和目的操作数。memop 确定树的形式为 ASGNa(x,b(INDIR(x),c))：

```
⟨X86 functions 390⟩+≡                                394 398  390
  static int memop(p) Node p; {
     if (generic(p->kids[1]->kids[0]->op) == INDIR
     && sametree(p->kids[0], p->kids[1]->kids[0]->kids[0]))
        return 3;
     else
        return LBURG_MAX;
  }
```

memop 确定树的整体形状，sametree 判断目的操作数与第一源操作数是否相同：

```
⟨X86 functions 390⟩+≡                                    397 400  390
  static int sametree(p, q) Node p, q; {
     return p == NULL && q == NULL
     || p && q && p->op == q->op && p->syms[0] == q->syms[0]
        && sametree(p->kids[0], q->kids[0])
        && sametree(p->kids[1], q->kids[1]);
  }
```

二元操作的其他变体允许第二操作数是寄存器或常量：

```
⟨X86 rules 395⟩+≡                                        397 398  390
  stmt: ASGNI(addr,ADDI(mem,rc))   "add %1,%2\n"  memop(a)
  stmt: ASGNI(addr,ADDU(mem,rc))   "add %1,%2\n"  memop(a)
  stmt: ASGNI(addr,SUBI(mem,rc))   "sub %1,%2\n"  memop(a)
  stmt: ASGNI(addr,SUBU(mem,rc))   "sub %1,%2\n"  memop(a)

  stmt: ASGNI(addr,BANDU(mem,rc))  "and %1,%2\n"  memop(a)
  stmt: ASGNI(addr,BORU(mem,rc))   "or %1,%2\n"   memop(a)
  stmt: ASGNI(addr,BXORU(mem,rc))  "xor %1,%2\n"  memop(a)
```

每个单型一元操作都将破坏它唯一的操作数：

```
⟨X86 rules 395⟩+≡                                        398 398  390
  reg: BCOMU(reg)  "?mov %c,%0\nnot %c\n"  2
  reg: NEGI(reg)   "?mov %c,%0\nneg %c\n"  2

  stmt: ASGNI(addr,BCOMU(mem))  "not %1\n"  memop(a)
  stmt: ASGNI(addr,NEGI(mem))   "neg %1\n"  memop(a)
```

移位指令与其他二元整型指令类似，只是移位的距离必须是常量或是在字节寄存器 cl 内，即在寄存器 ecx 的低位字节内：

```
⟨X86 rules 395⟩+≡                                        398 399  390
  reg: LSHI(reg,rc5)  "?mov %c,%0\nsal %c,%1\n"  2
  reg: LSHU(reg,rc5)  "?mov %c,%0\nshl %c,%1\n"  2
  reg: RSHI(reg,rc5)  "?mov %c,%0\nsar %c,%1\n"  2
  reg: RSHU(reg,rc5)  "?mov %c,%0\nshr %c,%1\n"  2

  stmt: ASGNI(addr,LSHI(mem,rc5))  "sal %1,%2\n"  memop(a)
  stmt: ASGNI(addr,LSHU(mem,rc5))  "shl %1,%2\n"  memop(a)
  stmt: ASGNI(addr,RSHI(mem,rc5))  "sar %1,%2\n"  memop(a)
  stmt: ASGNI(addr,RSHU(mem,rc5))  "shr %1,%2\n"  memop(a)

  rc5: CNSTI  "%a"  range(a, 0, 31)
  rc5: reg    "cl"
```

值得注意的是，移位的常量必须在 0 ~ 31 之间。因为 X86 的汇编程序有很多种，所以有些汇编程序也许会对未定义的移位报错。rtarget 把所有 0 ~ 31（包含 0 和 31）以外的常量放入 cl 中。

```
⟨is p->kids[1] a constant common subexpression? 398⟩=          399
     generic(p->kids[1]->op) == INDIR
  && p->kids[1]->kids[0]->op == VREG+P
  && p->kids[1]->syms[RX]->u.t.cse
  && generic(p->kids[1]->syms[RX]->u.t.cse->op) == CNST
```

```
⟨X86 target 399⟩=                                          399  393
  case RSHI: case RSHU: case LSHI: case LSHU:
     if (generic(p->kids[1]->op) != CNST
     && !(⟨is p->kids[1] a constant common subexpression? 398⟩)) {
        rtarget(p, 1, intreg[ECX]);
        setreg(p, intreg[EAX]);
     }
     break;
```

调用 setreg 确保该节点不会定位到 ecx。否则模板启始的 mov 指令将会重写 ecx，这时 cl 的值还没有被使用。eax 不是唯一可接受的寄存器，但我们在程序测试中发现，不是常量的移位很少见，所以不值得为这些移位设计一个不包含 ecx 的通配符。

乘法指令 imul 表示有符号整数的乘法。imul 有一个指令变体表示寄存器与寄存器、常量或存储器相乘：

```
⟨X86 rules 395⟩+=                                          398 399  390
  reg: MULI(reg,mrc3)  "?mov %c,%0\n\imul %c,%1\n"  14
```

另一种指令变体使用 3 个操作数，它将常量和寄存器或存储器单元的乘积放入寄存器：

```
⟨X86 rules 395⟩+=                                          399 399  390
  reg: MULI(con,mr)    "imul %c,%1,%0\n"  13
```

剩下的乘法指令限制更多。例如，mul 指令表示无符号整数的乘法。

```
⟨X86 rules 395⟩+=                                          399 400  390
  reg: MULU(reg,mr)  "mul %1\n"  13
```

mul 指令默认第一操作数在 eax 中，指令结果存入双寄存器 edx-eax。其中 eax 放结果的低位字节，除非操作溢出，否则 eax 存放的就是结果。在溢出的情况下，ANSI 称其为结果未定义，eax 仍然可以作为结果：

```
⟨X86 target 399⟩+=                                         399 400  393
  case MULU:
     setreg(p, quo);
     rtarget(p, 0, intreg[EAX]);
     break;
```

quo 和 rem 代表寄存器对 eax-edx，它们保存无符号乘法操作的结果；在除法运算中，它们先保存被除数。除法完成后，eax 保存商，edx 保存余数。

```
⟨X86 data 391⟩+=                                           393  389
  static Symbol quo, rem;

⟨X86 progbeg 391⟩+=                                        393  391
  quo = mkreg("eax", EAX, 1, IREG);
  quo->x.regnode->mask |= 1<<EDX;
  rem = mkreg("edx", EDX, 1, IREG);
  rem->x.regnode->mask |= 1<<EAX;
```

div 指令表示整数除法。指令默认第一个参数在寄存器对 edx-eax 中，操作完成后商放入 eax，余数放入 edx：

```
〈X86 target 399〉+≡                                    399 402  393
  case DIVI: case DIVU:
     setreg(p, quo);
     rtarget(p, 0, intreg[EAX]);
     rtarget(p, 1, intreg[ECX]);
     break;
  case MODI: case MODU:
     setreg(p, rem);
     rtarget(p, 0, intreg[EAX]);
     rtarget(p, 1, intreg[ECX]);
     break;
```

xor 指令清空 edx，为无符号除法做准备：

```
〈X86 rules 395〉+≡                                     399 400  390
  reg: DIVU(reg,reg)  "xor edx,edx\ndiv %1\n"
  reg: MODU(reg,reg)  "xor edx,edx\ndiv %1\n"
```

cdq 指令将 eax 的符号位扩展到 edx 中，为有符号除法做准备：

```
〈X86 rules 395〉+≡                                     400 400  390
  reg: DIVI(reg,reg)  "cdq\nidiv %1\n"
  reg: MODI(reg,reg)  "cdq\nidiv %1\n"
```

上面的第一条指令会改写 edx，所以应该将除数放到其他地方。一种方法是放入 ecx。这种限制其实没有必要，只是因为整数除法和求模运算并不常见。

整型和指针类型之间的转换没有什么意义，可通过 mov 指令实现。move 标记这些指令，希望将来能够删除它们：

```
〈X86 rules 395〉+≡                                     400 400  390
  reg: CVIU(reg)  "mov %c,%0\n"  move(a)
  reg: CVPU(reg)  "mov %c,%0\n"  move(a)
  reg: CVUI(reg)  "mov %c,%0\n"  move(a)
  reg: CVUP(reg)  "mov %c,%0\n"  move(a)
```

movsx、movzx 与 mov 类似，不同之处在于，它们分别按符号扩展或零扩展对输入进行加宽：

```
〈X86 rules 395〉+≡                                     400 400  390
  reg: CVCI(INDIRC(addr))  "movsx %c,byte ptr %0\n"  3
  reg: CVCU(INDIRC(addr))  "movzx %c,byte ptr %0\n"  3
  reg: CVSI(INDIRS(addr))  "movsx %c,word ptr %0\n"  3
  reg: CVSU(INDIRS(addr))  "movzx %c,word ptr %0\n"  3
```

movsx 和 movzx 也能操作寄存器，但是源操作数必须是 8 位或 16 位子寄存器，因此它们需要借助 emit2：

```
〈X86 rules 395〉+≡                                     400 401  390
  reg: CVCI(reg)  "# extend\n"  3
  reg: CVCU(reg)  "# extend\n"  3
  reg: CVSI(reg)  "# extend\n"  3
  reg: CVSU(reg)  "# extend\n"  3

〈X86 functions 390〉+≡                                 398 402  390
  static void emit2(p) Node p; {
```

```
    ⟨X86 emit2 401⟩
  }

⟨result 401⟩=                                                    401
  p->syms[RX]->x.name

⟨X86 emit2 401⟩=                                          401  401
  #define preg(f) ((f)[getregnum(p->x.kids[0])]->x.name)

  if (p->op == CVCI)
     print("movsx %s,%s\n", ⟨result 401⟩, preg(charreg));
  else if (p->op == CVCU)
     print("movzx %s,%s\n", ⟨result 401⟩, preg(charreg));
  else if (p->op == CVSI)
     print("movsx %s,%s\n", ⟨result 401⟩, preg(shortreg));
  else if (p->op == CVSU)
     print("movzx %s,%s\n", ⟨result 401⟩, preg(shortreg));
```

将整型变窄的转换也需要专门的处理：

```
⟨X86 rules 395⟩+=                                        400 401  390
  reg: CVIC(reg)  "# truncate\n"  1
  reg: CVIS(reg)  "# truncate\n"  1
  reg: CVUC(reg)  "# truncate\n"  1
  reg: CVUS(reg)  "# truncate\n"  1
```

模板 "?mov %c,%0\n" 和代价函数 movc(a) 将输入转移到输出，当源寄存器和目标寄存器相同时省略转移过程。mov 假设源操作数与目的操作数的字节大小相同，所以需要 mov 指令时，emit2 将产生一条 mov 指令，但这条 mov 指令的源寄存器和目标寄存器都是 16 位的。这样可减轻汇编程序，并且处理整型变窄转换时能保证复制足够的位：

```
⟨X86 emit2 401⟩+=                                          401  401
  else if (p->op == CVIC || p->op == CVIS
        || p->op == CVUC || p->op == CVUS) {
     char *dst = shortreg[getregnum(p)]->x.name;
     char *src = preg(shortreg);
     if (dst != src)
        print("mov %s,%s\n", dst, src);
  }
```

mov 指令的存数操作与取数操作类似：

```
⟨X86 rules 395⟩+=                                        401 401  390
  stmt: ASGNC(addr,rc)  "mov byte ptr %0,%1\n"   1
  stmt: ASGNI(addr,rc)  "mov dword ptr %0,%1\n"  1
  stmt: ASGNP(addr,rc)  "mov dword ptr %0,%1\n"  1
  stmt: ASGNS(addr,rc)  "mov word ptr %0,%1\n"   1
```

ARGI、ARGP 分别与 ASGNI、ASGNP 类似，但是 ARGI 和 ARGP 的目标指向栈顶的新单元。它们使用 push 指令将参数压栈：

```
⟨X86 rules 395⟩+=                                        401 402  390
  stmt: ARGI(mrc3)  "push %0\n"  1
  stmt: ARGP(mrc3)  "push %0\n"  1
```

尽管和直觉不同，上面规则中使用 mrc3 是正确的。即使下面两条指令只需要 2 个时钟周期，push 0 也要花费 4 个时钟周期。

```
mov eax,0
push eax
```

doarg 调用 mkactual 为下一个实参计算栈偏移量，同时更新 maxargoffset。与 RISC 结构的机器不同，X86 结构有一个 push 指令，不需要 mkactual 计算栈偏移。但由于在调用完成后，call 指令需要 maxargoffset 从栈中弹出实参，因此 doarg 仍然会调用 mkactual 来计算 maxargoffset，docall 将 maxargoffset 存入 CALL 节点。

```
⟨X86 functions 390⟩+≡                                    400 403  390
  static void doarg(p) Node p; {
     mkactual(4, p->syms[0]->u.c.v.i);
  }
```

ASGNB 复制存储器块，movsb 指令根据 esi 内的地址值从对应的内存单元中取一个字节，然后放入 edi 地址值对应的内存单元，同时两个寄存器都加 1。串指令前缀 rep 将后缀指令重复 ecx 次，所以 rep movsb 从 esi 对应的内存单元复制 ecx 个字节到 edi 对应的内存单元。target 计算源地址和目的地址，然后放入 esi 和 edi：

```
⟨X86 target 399⟩+≡                                       400 402  393
  case ASGNB:
     rtarget(p, 0, intreg[EDI]);
     rtarget(p->kids[1], 0, intreg[ESI]);
     break;
```

ASGNB 的模板将块的大小存入 ecx，并产生 rep movsb：

```
⟨X86 rules 395⟩+≡                                        401 402  390
  stmt: ASGNB(reg,INDIRB(reg))  "mov ecx,%a\nrep movsb\n"
```

ARGB 的操作类似。源操作数是 ARGB 唯一的子节点：

```
⟨X86 target 399⟩+≡                                       402 406  393
  case ARGB:
     rtarget(p->kids[0], 0, intreg[ESI]);
     break;
```

目的操作数固定为栈顶，所以模板开始时在栈顶分配一个块，并使 edi 指向栈顶：

```
⟨X86 rules 395⟩+≡                                        402 403  390
  stmt: ARGB(INDIRB(reg))  "sub esp,%a\nmov edi,esp\n_
                           mov ecx,%a\nrep movsb\n"
```

fep 修改 ecx，movsb 修改 esi 和 edi：

```
⟨X86 clobber 402⟩=                                       406  393
  case ASGNB: case ARGB:
     spill(1<<ECX | 1<<ESI | 1<<EDI, IREG, p);
     break;
```

这里不需要 blk 程序：

```
⟨X86 functions 390⟩+≡                                    402 407  390
  static void blkfetch(k, off, reg, tmp)
  int k, off, reg, tmp; {}
  static void blkstore(k, off, reg, tmp)
  int k, off, reg, tmp; {}
  static void blkloop(dreg, doff, sreg, soff, size, tmps)
  int dreg, doff, sreg, soff, size, tmps[]; {}
```

程序段 <interface routine names> 需要静态 blk 程序，它出现在 x86IR 中，所以必须定义这些 blk 例程，但是这些例程不会被调用，因此它们没有任何操作。

浮点指令使用由 8 个 80 位寄存器组成的栈。所有的临时变量值都是 80 位。ANSI C 允许计算操作使用额外的精度，所以代码生成器不需要产生额外的代码。

有些浮点指令的某个操作数可以来自存储器。操作数（不是操作符）必须指明类型：

```
⟨X86 rules 395⟩+≡                                        402 403  390
  memf: INDIRD(addr)          "qword ptr %0"
  memf: INDIRF(addr)          "dword ptr %0"
  memf: CVFD(INDIRF(addr))    "dword ptr %0"
```

fld 指令表示从存储器中读取浮点值并压入浮点栈：

```
⟨X86 rules 395⟩+≡                                        403 403  390
  reg: memf  "fld %0\n"  3
```

fstp 指令表示从浮点栈中弹出一个值并存入存储器：

```
⟨X86 rules 395⟩+≡                                        403 403  390
  stmt: ASGND(addr,reg)         "fstp qword ptr %0\n"  7
  stmt: ASGNF(addr,reg)         "fstp dword ptr %0\n"  7
  stmt: ASGNF(addr,CVDF(reg))   "fstp dword ptr %0\n"  7
```

浮点参数保存在存储器栈里，所以应使用减法指令为这些参数分配空间，然后使用 fstp 指令填充参数空间：

```
⟨X86 rules 395⟩+≡                                        403 403  390
  stmt: ARGD(reg)  "sub esp,8\nfstp qword ptr [esp]\n"
  stmt: ARGF(reg)  "sub esp,4\nfstp dword ptr [esp]\n"
```

一元操作改变浮点栈栈顶的元素。例如，fchs 指令将栈顶元素取反：

```
⟨X86 rules 395⟩+≡                                        403 403  390
  reg: NEGD(reg)  "fchs\n"
  reg: NEGF(reg)  "fchs\n"
```

二元操作处理浮点栈栈顶的两个元素，或者处理一个栈顶元素和一个存储器操作数：

```
⟨X86 rules 395⟩+≡                                        403 404  390
  flt: memf  " %0"
  flt: reg   "p st(1),st"
```

例如，指令：

```
fsubp st(1),st
```

从栈顶下的第一个元素 st(1) 中减去栈顶值 st，然后将 st 弹出（因为有后缀 p），消息被减值。指令：

```
fsub qword ptr x
```

从栈顶减去 64 位的 x 值。这里没有后缀 p，所以浮点栈的高度不会变。其他二元操作也类似：

```
〈X86 rules 395〉+≡                                          403 404  390
  reg: ADDD(reg,flt)  "fadd%1\n"
  reg: ADDF(reg,flt)  "fadd%1\n"
  reg: DIVD(reg,flt)  "fdiv%1\n"
  reg: DIVF(reg,flt)  "fdiv%1\n"
  reg: MULD(reg,flt)  "fmul%1\n"
  reg: MULF(reg,flt)  "fmul%1\n"
  reg: SUBD(reg,flt)  "fsub%1\n"
  reg: SUBF(reg,flt)  "fsub%1\n"
```

由于浮点寄存器已经是 80 位的宽度，不需要再加宽，所以浮点到双精度的转换不改变浮点寄存器。CVFD 不需要其他操作：

```
〈X86 rules 395〉+≡                                          404 404  390
  reg: CVFD(reg)  "# CVFD\n"
```

X86 结构中没有指令能直接把双精度数缩小成浮点数，所以必须先把这个值存入临时浮点单元，把值缩小，然后将浮点值重新取入浮点寄存器，尽管这将再次加宽值。但是已经丢弃了多余的精度：

```
〈X86 rules 395〉+≡                                          404 405  390
  reg: CVID(reg)  "push %0\n_
                   fild dword ptr 0[esp]\nadd esp,4\n"  12
```

从双精度数到整数的转换类似。指令 fistp 从浮点栈弹出双精度值，转换成整型值存入存储器：

```
〈X86 rules 395〉+≡                                          404 404  390
  stmt: ASGNI(addr,CVDI(reg))  "fistp dword ptr %0\n"  29
```

如果代码用到（通用）寄存器中的整型结果，则需要在存储器栈上创建、使用、释放一个临时变量：

```
〈X86 rules 395〉+≡                                          404 404  390
  reg: CVDI(reg)  "sub esp,4\n_
                   fistp dword ptr 0[esp]\npop %c\n" 31
```

fild 指令表示取一个整数，将其转换成 80 位的浮点值并压入浮点栈：

```
〈X86 rules 395〉+≡                                          404 404  390
  reg: CVID(INDIRI(addr))  "fild dword ptr %0\n"  10
```

如果操作数来自（通用）寄存器，则需要在存储器栈上创建、使用、释放另一个临时变量：

```
〈X86 rules 395〉+≡                                          404 404  390
  reg: CVDF(reg)  "sub esp,4\nfstp dword ptr 0[esp]\n_
                   fld dword ptr 0[esp]\nadd esp,4\n"  12
```

jmp 指令表示无条件跳转，它的操作数可以是标号、寄存器或存储器单元：

```
⟨X86 rules 395⟩+≡                                    404 405  390
  addrj: ADDRGP  "%a"
  addrj: reg     "%0"  2
  addrj: mem     "%0"  2
  stmt:  JUMPV(addrj)  "jmp %0\n"  3
  stmt:  LABELV        "%a:\n"
```

条件分支指令首先比较两个值，当条件满足时执行分支。cmp 指令实现比较并有多种变体。一种是存储器单元与寄存器或常量做比较。有符号整数总共有 6 种关系：

```
⟨X86 rules 395⟩+≡                                    405 405  390
  stmt: EQI(mem,rc)  "cmp %0,%1\nje %a\n"   5
  stmt: GEI(mem,rc)  "cmp %0,%1\njge %a\n"  5
  stmt: GTI(mem,rc)  "cmp %0,%1\njg %a\n"   5
  stmt: LEI(mem,rc)  "cmp %0,%1\njle %a\n"  5
  stmt: LTI(mem,rc)  "cmp %0,%1\njl %a\n"   5
  stmt: NEI(mem,rc)  "cmp %0,%1\njne %a\n"  5
```

因为 EQI 和 NEI 也适用于无符号整数，所以无符号整数只有 4 种关系：

```
⟨X86 rules 395⟩+≡                                    405 405  390
  stmt: GEU(mem,rc)  "cmp %0,%1\njae %a\n"  5
  stmt: GTU(mem,rc)  "cmp %0,%1\nja  %a\n"  5
  stmt: LEU(mem,rc)  "cmp %0,%1\njbe %a\n"  5
  stmt: LTU(mem,rc)  "cmp %0,%1\njb  %a\n"  5
```

cmp 的另一种变体是寄存器与常量、存储器单元或另一个寄存器做比较，为此，我们重复上面有关有符号和无符号的规则：

```
⟨X86 rules 395⟩+≡                                    405 405  390
  stmt: EQI(reg,mrc1)  "cmp %0,%1\nje %a\n"   4
  stmt: GEI(reg,mrc1)  "cmp %0,%1\njge %a\n"  4
  stmt: GTI(reg,mrc1)  "cmp %0,%1\njg %a\n"   4
  stmt: LEI(reg,mrc1)  "cmp %0,%1\njle %a\n"  4
  stmt: LTI(reg,mrc1)  "cmp %0,%1\njl %a\n"   4
  stmt: NEI(reg,mrc1)  "cmp %0,%1\njne %a\n"  4

  stmt: GEU(reg,mrc1)  "cmp %0,%1\njae %a\n"  4
  stmt: GTU(reg,mrc1)  "cmp %0,%1\nja %a\n"   4
  stmt: LEU(reg,mrc1)  "cmp %0,%1\njbe %a\n"  4
  stmt: LTU(reg,mrc1)  "cmp %0,%1\njb %a\n"   4
```

指令 fcomp x 从浮点栈中弹出一个元素，将它与存储器中的操作数 x 做比较。变体 fcompp 的两个比较操作数都来自浮点栈。非终结符 cmpf 允许一条规则产生两个变体：

```
⟨X86 rules 395⟩+≡                                    405 406  390
  cmpf: memf  " %0"
  cmpf: reg   "p"
```

与 cmpf 类似的非终结符 ftt（定义见第 403 页）并不这样做，因为汇编程序需要 st(1)，st 用于二元操作，但禁止 st(1)，st 用于 fcomp。fcomp 将比较的结果存入某些机器标记。指令 fststw ax 将标记存入 eax 的底部，然后指令 sahf 将它们载入由条件分支指令检测的标记中：

```
⟨X86 rules 395⟩+≡                                          405 406 390
  stmt: EQD(cmpf,reg)  "fcomp%0\nfstsw ax\nsahf\njc %a\n"
  stmt: GED(cmpf,reg)  "fcomp%0\nfstsw ax\nsahf\njbe %a\n"
  stmt: GTD(cmpf,reg)  "fcomp%0\nfstsw ax\nsahf\njb %a\n"
  stmt: LED(cmpf,reg)  "fcomp%0\nfstsw ax\nsahf\njae %a\n"
  stmt: LTD(cmpf,reg)  "fcomp%0\nfstsw ax\nsahf\nja %a\n"
  stmt: NED(cmpf,reg)  "fcomp%0\nfstsw ax\nsahf\njne %a\n"

  stmt: EQF(cmpf,reg)  "fcomp%0\nfstsw ax\nsahf\njc %a\n"
  stmt: GEF(cmpf,reg)  "fcomp%0\nfstsw ax\nsahf\njbe %a\n"
  stmt: GTF(cmpf,reg)  "fcomp%0\nfstsw ax\nsahf\njb %a\n"
  stmt: LEF(cmpf,reg)  "fcomp%0\nfstsw ax\nsahf\njae %a\n"
  stmt: LTF(cmpf,reg)  "fcomp%0\nfstsw ax\nsahf\nja %a\n"
  stmt: NEF(cmpf,reg)  "fcomp%0\nfstsw ax\nsahf\njne %a\n"
```

clobber 记录了浮点条件分支对 eax 的破坏：

```
⟨X86 clobber 402⟩+≡                                        402 406 393
  case EQD: case LED: case GED: case LTD: case GTD: case NED:
  case EQF: case LEF: case GEF: case LTF: case GTF: case NEF:
     spill(1<<EAX, IREG, p);
     break;
```

指令 call 把下一条指令的地址压入栈，然后跳转至指令操作数所指定的地址：

```
⟨X86 rules 395⟩+≡                                          405 406 390
  reg:  CALLI(addrj)  "call %0\nadd esp,%a\n"
  stmt: CALLV(addrj)  "call %0\nadd esp,%a\n"
```

调用完成后，指令 add 将参数弹出栈。前端把每个调用节点的 syms[0] 指向一个符号，该符号等于实参的字节数，%a 表示产生该数。返回值存入 eax：

```
⟨X86 target 399⟩+≡                                         402 393
  case CALLI: case CALLV:
     setreg(p, intreg[EAX]);
     break;
  case RETI:
     rtarget(p, 0, intreg[EAX]);
     break;
```

浮点函数将返回值存入浮点寄存器栈顶：

```
⟨X86 rules 395⟩+≡                                          406 407 390
  reg: CALLF(addrj)  "call %0\nadd esp,%a\n"
  reg: CALLD(addrj)  "call %0\nadd esp,%a\n"

⟨X86 clobber 402⟩+≡                                        406 393
  case CALLD: case CALLF:
     spill(1<<EDX | 1<<EAX, IREG, p);
     break;
```

与通常情况一样，存在返回节点，但返回节点主要用于指导寄存器定位，并不产生代码：

```
⟨X86 rules 395⟩+≡                                    406  390
  stmt: RETI(reg)  "# ret\n"
  stmt: RETF(reg)  "# ret\n"
  stmt: RETD(reg)  "# ret\n"
```

18.3 函数的实现

前端调用 local 通知局部变量和前端生成的临时变量。代码生成器不给浮点局部变量指派寄存器，甚至不给浮点临时变量指派寄存器，所以 local 一开始就把这些变量压入栈：

```
⟨X86 functions 390⟩+≡                                403  407  390
  static void local(p) Symbol p; {
    if (isfloat(p->type))
      p->sclass = AUTO;
    if (askregvar(p, rmap[ttob(p->type)]) == 0)
      mkauto(p);
  }
```

因为代码生成器会给整型临时变量指派寄存器，所以浮点局部变量与整型局部变量的处理过程是不同的。代码生成器也不给其他的局部变量指派寄存器，但 prugbeg 清空 vmask[IREG]，指导 askregvar 保存真正的寄存器变量。

前端调用接口过程 function 通知一个新的例程：

```
⟨X86 functions 390⟩+≡                                407  409  390
  static void function(f, caller, callee, n)
  Symbol f, callee[], caller[]; int n; {
    int i;
    ⟨X86 function 407⟩
  }
```

function 产生函数头代码，包括一个标记以及存储 ebx、esi、edi、ebp 的指令：

```
⟨X86 function 407⟩≡                                  407  407
  print("%s:\n", f->x.name);
  print("push ebx\n");
  print("push esi\n");
  print("push edi\n");
  print("push ebp\n");
  print("mov ebp,esp\n");
```

函数头代码也更新 ebp。图 18-1 显示了 X86 的帧。

接下来，function 清除寄存器分配器的状态，并计算每个输入参数的栈偏移量。第一个输入参数距离当前 ebp 20 个字节：16 个字节用于存储寄存器，4 个字节用于存储返回地址。

```
⟨X86 function 407⟩+≡                                 407  408  407
  ⟨clear register state 319⟩
  offset = 16 + 4;
  for (i = 0; callee[i]; i++) {
    ⟨assign offset to argument 408⟩
  }
```



图 18-1 X86 的帧

offset 给出了下一个参数到 ebp 的偏移。它决定了被调用程序和调用程序的参数视图的 x.offset 和 x.name 域：

```
⟨assign offset to argument i 408⟩=                                    407
  Symbol p = callee[i];
  Symbol q = caller[i];
  p->x.offset = q->x.offset = offset;
  p->x.name = q->x.name = stringf("%d", p->x.offset);
  p->sclass = q->sclass = AUTO;
  offset += roundup(q->type->size, 4);
```

sclass 域设置为 AUTO，以记录没有参数被指派给寄存器。然后调整 offset 处理下一个参数，并保持栈对齐。

接着，function 调用 gencode 处理例程的主体。它首先重置 offset 和 maxoffset，记录还未分配局部变量：

```
⟨X86 function 407⟩+≡                                          407 408  407
  offset = maxoffset = 0;
  gencode(caller, callee);
  framesize = roundup(maxoffset, 4);
  if (framesize > 0)
     print("sub esp,%d\n", framesize);
```

当 gencode 返回时，在 gencode 的生存周期内 offset 的最大值保存在 maxoffset 中，所以分配剩余帧的代码现在可以放进函数头代码里。function 接着调用 emitcode 产生例程的主体，emitcode 直接调用 print 生成函数尾代码。函数尾代码只执行恢复函数头代码的操作：

```
⟨X86 function 407⟩+≡                                          408  407
  emitcode();
  print("mov esp,ebp\n");
  print("pop ebp\n");
  print("pop edi\n");
  print("pop esi\n");
```

```
print("pop ebx\n");
print("ret\n");
```

18.4 数据的定义

前端调用 defsymbol 函数通知每个新的符号：

```
⟨X86 functions 390⟩+≡                                    407 409 390
  static void defsymbol(p) Symbol p; {
    ⟨X86 defsymbol 409⟩
}
```

每个静态局部符号都有唯一的生成名字，以避免与其他静态局部符号重名：

```
⟨X86 defsymbol 409⟩≡                                     409 409
  if (p->scope >= LOCAL && p->sclass == STATIC)
     p->x.name = stringf("L%d", genlabel(1));
```

生成的符号已经有了唯一的数字名，defsymbol 在名字前面加上一个字母前缀，使之成为合法的汇编标识符：

```
⟨X86 defsymbol 409⟩+≡                                    409 409 409
  else if (p->generated)
     p->x.name = stringf("L%s", p->name);
```

根据约定，defsymbol 在输出的全局符号名字前加一条下划线：

```
⟨X86 defsymbol 409⟩+≡                                    409 409 409
  else if (p->scope == GLOBAL || p->sclass == EXTERN)
     p->x.name = stringf("_%s", p->name);
```

十六进制的常量名必须重定格式。如前端使用 0xff，X86 汇编程序则用 0ffH：

```
⟨X86 defsymbol 409⟩+≡                                    409 409 409
  else if (p->scope == CONSTANTS
  && (isint(p->type) || isptr(p->type))
  && p->name[0] == '0' && p->name[1] == 'x')
     p->x.name = stringf("0%sH", &p->name[2]);
```

对于剩余的符号，如十进制常量和静态全局符号，前端和后端共用一个名字：

```
⟨X86 defsymbol 409⟩+≡                                    409 409
  else
     p->x.name = p->name;
```

接口过程 address 使用偏移量算法（如 _up-28）计算符号的偏移量，defsymbol 对普通符号也这么做：

```
⟨X86 functions 390⟩+≡                                    409 410 390
  static void address(q, p, n) Symbol q, p; int n; {
     if (p->scope == GLOBAL
     || p->sclass == STATIC || p->sclass == EXTERN)
        q->x.name = stringf("%s%s%d",
           p->x.name, n >= 0 ? "+" : "", n);
     else {
        q->x.offset = p->x.offset + n;
```

```
            q->x.name = stringd(q->x.offset);
        }
    }
```

对于栈上的变量，address 仅计算调整后的偏移量。对于以标记寻址的变量，address 将 x.name 设置成形如 name±n 的字串。如果偏移量是正的，那么由文字 "+" 产生操作符，如果偏移量是负的，由 %d 产生操作符。

前端调用 defconst 产生汇编指示符，该指示符分配并将一个标量初始化成常量。参数 ty 识别联合 v 中相应的成员：

```
⟨X86 functions 390⟩+≡                                    409 410  390
  static void defconst(ty, v) int ty; Value v; {
     switch (ty) {
        ⟨X86 defconst 410⟩
     }
  }
```

大部分的 case 分支只是发送一个成员到汇编指示符，它的作用是分配并初始化一个 ty 类型的单元：

```
⟨X86 defconst 410⟩≡                                      410  410
  case C: print("db %d\n",   v.uc); return;
  case S: print("dw %d\n",   v.ss); return;
  case I: print("dd %d\n",   v.i ); return;
  case U: print("dd 0%xH\n", v.u ); return;
  case P: print("dd 0%xH\n", v.p ); return;
```

汇编程序的 real4 和 real8 指示符不能表示由任意表达式计算产生的浮点常量，例如类型转换得到的浮点常量，所以 defconst 不使用它们，而是产生十六进制的浮点常量：

```
⟨X86 defconst 410⟩+≡                                     410 410  410
  case F:
     print("dd 0%xH\n", *(unsigned *)&v.f);
     return;
```

如果 lcc 在低位优先的机器上运行，但为高位优先的机器编译代码，则双精度数的两个部分必须交换，反之亦然：

```
⟨X86 defconst 410⟩+≡                                     410  410
  case D: {
     unsigned *p = (unsigned *)&v.d;
     print("dd 0%xH,0%xH\n", p[swap], p[1 - swap]);
     return;
     }
```

接口过程 defaddress 给指针分配空间，并用符号地址将其初始化：

```
⟨X86 functions 390⟩+≡                                    410 411  390
  static void defaddress(p) Symbol p; {
     print("dd %s\n", p->x.name);
  }
```

defconst 的指针分支将指针初始化为数字地址。

接口过程 defstring 产生指示符，初始化一组字节：

```
⟨X86 functions 390⟩+≡                                    410 411  390
  static void defstring(n, str) int n; char *str; {
     char *s;

     for (s = str; s < str + n; s++)
        print("db %d\n", (*s)&0377);
  }
```

因为 ANSI C 的转义码允许字符串中包含空字节，所以 defstring 通过计数找到字符串的结尾。

前端调用 export 来将当前模块中的符号展示给其他模块。汇编指示符 public 的作用也是如此：

```
⟨X86 functions 390⟩+≡                                    411 411  390
  static void export(p) Symbol p; {
     print("public %s\n", p->x.name);
  }
```

extern 指示符使当前模块可以访问其他模块的输出符号，但是该指示符不能出现在段内，所以接口过程 import 需要临时从当前段切换出来：

```
⟨X86 functions 390⟩+≡                                    411 411  390
  static void import(p) Symbol p; {
     int oldseg = cseg;

     if (p->ref > 0) {
        segment(0);
        print("extrn %s:near\n", p->x.name);
        segment(oldseg);
     }
  }
```

指示符 near 声明外部符号可以直接寻址。平坦内存模式和它的 32 位地址允许为任何符号直接寻址，所以不必理解 near 和相关指示符，除非是正在生成分段的代码，这里加困难。

X86 中 segment 的实现必须要注意 segment(0) 的调用，segment(0) 切换出当前段但并不进入新的段。因为有些 X86 目标代码连接器禁止不必要的 extrns，所以只在符号会被使用的时候，import 才检查符号的 ref 域，生成指示符。

前端调用接口过程 global 来定义一个新的全局符号。如果这个符号被初始化，则前端接着调用 defconst，所以 global 只为那些没有初始化的全局符号分配空间，这些全局符号都在 BSS 段中：

```
⟨X86 functions 390⟩+≡                                    411 412  390
  static void global(p) Symbol p; {
     print("align %d\n",
        p->type->align > 4 ? 4 : p->type->align);
     print("%s label byte\n", p->x.name);
     if (p->u.seg == BSS)
        print("db %d dup (0)\n", p->type->size);
  }
```

前端调用接口过程 space 以定义初始化为 0 的全局数据块：

```
{X86 functions 390} +=                                    411  390
  static void space(n) int n; {
     if (cseg != BSS)
        print("db %d dup (0)\n", n);
  }
```

深入阅读

有多种参考手册详细描述了 X86 机器的结构（Intel Corp，1993）。微软的 MASM 和 Borland 的 Turbo Assembler 的汇编程序手册介绍了 X86 的汇编语言，特别是控制不同存储器模式的汇编指示符。

练习

18.1 浏览 X86 的参考手册，注意那些在 lcc 中没用到的指令。加入一些规则产生这些指令。在每次修改之前和之后测试编译器，评价你所做的改动。

18.2 一些 lcc 的操作码是可交换的，这意味着对于每条规则：

```
reg: ADDI(reg,mrc1)  "mov %c,%0\nadd %c,%1\n"  2
```

都有规则：

```
reg: ADDI(mrc1,reg)  "mov %c,%1\nadd %c,%0\n"  2
```

增加一些交换规则，哪些规则会带来明显的不同？哪些规则因为在前端不可能生成而不会有区别？

18.3 某些非可交换的操作有对称的规则，这些规则用于交换操作数。例如，规则：

```
stmt: GTI(reg,mrc1)  "cmp %0,%1\njg %a\n"  2
```

有对称的规则：

```
stmt: GTI(mrc1,reg)  "cmp %1,%0\njl %a\n"  2
```

这是因为 x>y 当且仅当 y<x。请找出有用的 X86 对称规则。

18.4 rep movsb 每次复制 ecx 个字节，而 rep movsw 每次复制 ecx 个 16 位单元，后者比前者差不多快两倍，rep movsd 每次复制 ecx 个 32 位单元，又比 rep movsw 几乎快两倍。尽可能利用这些指令，修改块复制的代码。

18.5 即便程序不使用 ebx、esi 和 edi，lcc 的函数头代码和函数尾代码也会保存并恢复它们。修正这一缺陷，看这是否值得。

18.6 保留一个寄存器，并将其分配给最有利的局部变量。评价一下改进的程度。重复该实验，保留更多的寄存器。这类寄存器数目取什么值具有最好的效果。

18.7 对于 f(i+1) 中的加法，lcc 会产生代码 lea edi,1[edi]。我们更希望产生代码 inc edi，但是想让 X86 的代码生成器为这类特殊情况产生该代码非常困难。请解释其中的原因。

18.8 构造一个 C 的小程序，引发 ckstack 的诊断信息。

18.9 修改 X86 的代码生成器，使其不依靠程序员的帮助就能溢出和恢复浮点寄存器。参看关于 ckstack 的讨论。

回　顾

lcc 是构造 C 编译器的一种方法。在 lcc 的设计与实现过程中对数以百计的技术策略进行了选择。这些策略中的许多都是可行的方法。在前面章节的练习中我们也给出了一些不同的策略。本章主要是回顾 lcc 的设计并讨论一些全局的设计策略，这些策略对当前的编译器实现起了重要作用。这些策略具有很好的实际效果，可能是目前的优先选择。

lcc 中运用了许多编程技巧，比如第 2 章的存储分配器和 2.5 节的字符串管理，它们都可应用于更广泛的领域。第 3 章的符号表模块，虽然只用在 lcc 中，但只需较少的改动，它就能应用于功能相似的其他领域。6.1 节的输入模块能用于任何需要高速输入的地方。

第 8 章讲述的语法分析技术可以用于需要语法分析和计算表达式的应用中，例如电子数据表。除了第 14 章的选择指令外，Iburg 还有更广的应用。由 Iburg 生成的匹配器与 lcc 的节点联系并不紧密，它们可以用于树的匹配模式问题。Iburg 所体现出的思想，即根据重要属性的简短说明来生成程序，已获得广泛应用。其他编译器也通常采用这种思想，借助 LEX、YACC 之类的工具生成词法分析器和语法分析器。

19.1　数据结构

由于共享的数据结构不多，因此我们能很好地处理前端与代码生成器之间数据结构的共享。这种方法的一个不足之处在于：比较其他简单的设计方法，这些结构更为复杂。例如，所有表示标识符的符号都跨过了接口。符号有许多域，其中有些只与前端相关，对它们的访问只能通过约定进行规范。有些符号仅使用某些域，比如标号就只用到 name 域和 u.l 中的域。如果采用专门适合于标号的数据结构可能更容易理解一些。

许多人认为 C 语言导致了这一复杂性：要描述所有可能的情况需要一个比 C 更为庞大的类型系统。用定义单独结构的方法可以减少某些复杂程度，例如，每种符号用一种结构来描述，用另一种结构来描述私有的前端数据，但这样做使数据结构变得更大而更复杂了。带继承的类型系统简化了结构变体的定义，也不增加使用那些变体的复杂性。对面向对象的程序语言来说，如 Oberon-2、Modual-3 和 C++，它们的类型系统必须用到这种机制。在这些语言中，我们定义了一个基本的符号类型，它只包含所有符号都具有的域，然后我们为每种符号设定不同类型。这些类型通过继承来对基本类型进行扩展，增加特殊的域。

比如在 Modula-3 中，基本类型定义如下：

```
TYPE Symbol = OBJECT
   name: TEXT
END;
```

这里定义了一个对象类型，它只有一个域 name，存放字符串。如果要定义标号类型，只需在基本类型中加入特定的域：

```
TYPE Label = Symbol OBJECT
   label: INTEGER;
   equatedto: Label
END;
```

这里定义的 Label 是一种对象类型，它除了具有 Symbol 的所有域外，还有两个标号特有的域。因为 Label 也是 Symbol，处理 Symbol 的过程也可以处理 Label。

这种机制在其他数据结构中也有用，如类型、树和节点。例如后端的扩展（symbol 和 node 中的 x 域）是不必要的，因为后端能定义附加类型，这些类型可以对前端的类型进行扩展，增加与目标机器相关的域。

使用面向对象的程序语言支持方法，即与特定类型的值相关、对这些值进行操作的过程，可以替代某些接口函数。此外，由于这些方法只能应用于特定的类型，所以它们可以消除 switch 语句，我们在 defconst 的实现中遇到过这些 switch 语句。

19.2 接口

lcc 删除了许多冗余部分并做了简单性的假定，因此它的代码生成接口很紧凑（compact）。但是这些省略和假定限制了接口在其他语言和机器上的应用能力。

lcc 的接口假定有符号和无符号整数以及长整数都具有相同的长度。这种假定使得 lcc 只要处理 9 种类型后缀和 108 种与类型相关的操作，但在一些 64 位机器上会变得更加复杂。如果需要多次使用，我们或许应该用类型后缀区分有符号字符和无符号的字符、长整数、整数，以及浮点数、双精度浮点数和长双精度浮点数。我们有时还考虑将这些类型返回 lcc，尽管这样做很烦琐。例如，为长双精度浮点数定义一个后缀，就要在前端和后端中增加 19 种操作和代码对其进行处理。这种改进不需要在很多地方增加代码，只在需要的地方加入若干行代码即可。另一种方法是用后缀描述数据类型，而非字长，然后再增加一个后缀来表示字长。比如 ADDI2 和 ADDI4 分别表示 2 字节和 4 字节的整数加法。字长标识也可以表示在 node 中，而不是表示在操作符的名字中。

接口假定所有指针的表示相同。这种假定使得在按字寻址的目标机器上 lcc 处理会变得复杂，因为在这些机器上，当要指向某个小于字长的单元，如字符时，就需要用额外的位来表示字内的单元。为了区分字和字符指针，需要另外增加后缀和至少 13 种操作。我们不认为这样规定有何不妥，但是到目前为止，我们还未使用按字寻址的目标机器。

指令系统中省去了可以通过简单操作组合实现的复杂操作。比如位域能通过移位和掩码操作，而不必设置专门的位域操作。对于具有位域指令的机器来说，这样做会使开发更趋于复杂。另一方面，前端将一位的位域按特例处理，并产生高效的掩码运算的 dag（无环有向图），这样生成的代码优于使用位域指令的代码。

经过不断修正，通过将功能不断地向前端转移，减少接口的数量，接口已经变得非常简化。例如，早期版本都有一个接口函数和操作符以实现 switch。每一次修正都减小了后端，但也留下了瑕疵。

一方面，前端过于集成化，比如在“操作符 × 类型”的矩阵中，曾经有像 indiru、retp 和 cvud 这样的操作，读去冗余的指令后能够节省一些代码，但采用更正规的操作集更易于学习。再比如，代码表对后端虽然不可见，但可通过 gencode 和 emitcode 遍历它。有些人利用 lcc 来研究全局优化，需要对遍历进行更精细的控制。为此，他们使代码表对后端可见，即将代码表作为接口，将功能更强大的 gencode 和 emitcode 函数转移到后端的与目标无关的部分。这样的改进实际上是同等价的代码表入口代替了 local 这样的接口函数。提供代码表或是流图的接口，再加上 gencode 和 emitcode 的标准实现，可以帮助用户在简单性和灵活性之间做出选择。

另一方面，接口还可以进一步简化。比如 asgn 和 call 与类型相关的变体可有不同数目的操

作数，这种变化使得原来只需要检查一般操作的某些策略变得更复杂。另外一个例子是，有些操作总生成很少的目标代码；有些操作在部分目标机器上不会生成任何代码；而有些操作，如 CVUI 和 CVIU，在当前任何可以想象的目标机器上都不会生成代码。像本书所描述的那样，产品级编译器的后端，应尽力避免产生无谓的寄存器—寄存器型移数操作。同样，描述 cv(ui) × (cs) 的转换也是毫无意义的，最好省去。

如果没有遵从某些接口约定，将会导致微妙的错误。例如，接口函数 local、function 以及操作符 calib 的代码，三者合作就可以为返回结构的函数产生代码。后端中的 3 个部分必须很好地配合，否则编译器很容易产生错误代码。其实前端完全能处理这些函数，可以去掉接口标志 wants_calib，但这样会排除一些已经建立好的调用序列。函数 ARGB 和标志 wants_argb 也与此类似。综合考虑产生相容的调用序列代码，形成了一个较复杂的代码生成接口。

lcc 的接口设计是针对单片集成编译器（monolithic compiler）的，这种编译器的前后端连接成一个单独的程序。这种做法很难将前端和后端分离成单独的程序。有些接口是双向的，接口函数 function 上行调用 gencode 和 emitcode 就是这样。通过这些上行调用，前端可以生成函数入口的转换代码。后端检查源语言类型表示中的若干域，它使用前端的函数（如 isstruct）查询类型。要将后端分离成单独的程序，就必须转换类型数据以供查询。同时，后端还必须实现函数的入口转换。

19.3 句法和语义分析

lcc 中的语法分析和语义分析是穿插进行的。许多编译器都采用这种设计方法，它们基于 20 世纪 60 年代就广为使用的经典的递归下降分析方法。这种分析方法易理解，手工实现也较容易，可以产生高效的编译器。

许多语言（如 C）是设计成一遍编译的，即处理源程序的同时生成代码，像 lcc 一样。大多数语言都遵从先声明后使用的规则：标识符在使用之前一定先声明，除非在一些特殊的情形下，并且有某种机制保证程序员可以理解。例如，C 语言的声明语句

```
extern Tree (*optree[])(int, Tree, Tree);
```

声明了 optree，但不是对它进行定义，这样声明后我们就能在定义它之前使用它。其他的例子如 11.5 节一开始提到的向前结构声明，Pascal 也具有这样的功能。这种声明的唯一目的就为了使一遍编译成为可能。

当代程序语言，如 Modula-3 和 ML，没有这类规则。在 Modula-3 中，声明就是定义，可以为常量、类型、变量、异常和过程命名，它们能以任意的顺序出现。它们出现的顺序只会影响初始化代码执行时的顺序。这种灵活性初看起来容易使人混淆，但是与 C 语言相比，Modula-3 的语法规则和特殊情形更少，从长远角度看 Modula-3 更容易理解。

因为必须处理完整个源代码才能解决声明之间的相关性之类的问题，具有这些特征的语言需要多遍扫描的编译器。这类编译器的语义分析和语法分析必须是分离的。第一遍扫描程序通常会为整个源程序建立一个 AST（抽象语法树），后续的扫描将多次遍历 AST，每遍都会加入特定的注释。例如，Modula-3 编译器的声明处理遍只分析说明、建立符号表，并用指向符号表入口的指针对节点进行注释。然后到代码生成遍时，就可以访问过程节点及其子节点，生成相应的代码，用与 lcc 的代码表等价的标记对过程节点进行注释。

lcc 的一遍扫描策略有其优点。与 AST 比较，一遍扫描所需存储空间更小，而且由于它构造

简单，不存在建立和遍历 AST 的额外开销，所以编译速度很快。对大型数组进行初始化时就能体现出这些优势，lcc 对其进行编译时需要的空间与最大的初始化代码成正比，并能处理任意大小的初始化操作。而使用 AST 的编译器，通常需要为所有的初始化表达式建立一棵树，为了避免过多地占用内存，必须限制初始化代码的大小。

对当代计算机来说，一遍扫描编译器带来的时间和空间上的高效，相对于 AST 策略带来的灵活性来说，已不再显得那么重要了。对 AST 进行多遍处理，可以简化每遍的代码。这种策略可以简化 lcc 的用于分析声明、表达式和语句的模块，使得本书的相关章节更容易理解。

AST 有助于将 lcc 用于其他目的。lcc 的某些部分可用于构造浏览器、为其他后端构造前端、为其他前端构造后端、构造连接时（link-time）和运行时（run-time）的代码生成器等，还能用于从解释器和调试器中生成代码。起初设计 lcc 时并没有预见到这些用途，如果 lcc 能够构造 AST 并能让用户对 AST 做扫描和注释，那么上述工作实现起来将更容易。

19.4 代码生成和优化

代码生成需要综合平衡各种因素。功能强大的优化器能产生更好的代码，但它们太大而且速度慢。介绍一个大型的编译器会花费我们过多时间，而且这样的编译器仅用一本书来讲述也是不够的；而一个速度较慢的编译器将花费我们和程序员很多时间，对于这些用户来说，编译花费的时间往往是瓶颈。由此来看，lcc 能生成令人满意的代码，但其他编译器在这方面可以做得更出色。

就每棵树独立来看，lcc 的指令选择是最佳的，但相邻树的代码的边界处就差一些。lcc 可以在最后使用的窥孔优化遍解决上述不足。在各种优化方法中，窥孔技术可能是其中最简单的一种。按照我们过去的经验，它所生成的代码最少。

lcc 的接口只支持代码生成，并不直接支持构造流图和其他有利于全局优化的结构。许多精心设计的 function 和 gen 版本能够合作实现相关结构的建立、执行优化、调用更简单的 gen，但从 AST 中生成流图是更为一般的解决途径。

lcc 的寄存器分配器还较为原始。它能为某些变量和局部公共子表达式分配寄存器，但在其他方面它的能力极为有限。目前采用图着色方法的寄存器分配器的能力更为出色。我们之所以没有采用功能更强的寄存器分配器，主要是因为这样做的话，估计至少要增加超过 1000 行的代码（差不多占编译器的 10%），因此不得不从本书中略去。

目前 lcc 的 SPARC 代码需要指令调度，将来调度可能也会用于其他目标机器上。理想情况下，调度与寄存器分配是相互影响的，但在寄存器分配遍后考虑调度器可能更简单，能更好地满足 lcc 的需要。

19.5 测试和验证

lcc 测试的第一级是将产生的汇编代码及汇编程序的输出与保存的标准汇编程序代码及输出进行比较。有时我们希望对汇编程序代码做修改，因此第一级比较就失去了作用，但它仍具有价值，因为如果比较测试发生了预期之外的不同，我们就知道某个对编译器的改动造成了错误。

测试时有时候也使用 ANSI C 编译器的 PHVS（Plum-Hall Validation Suite）中语言一致性的部分。还用到从 Fortran 语言翻译成的一系列数值程序。数值程序比起其他测试程序来含有更多的变量、更长的表达式、更多公共子表达式，这使得寄存器分配器工作更为繁忙，因此经常用于测试溢出。由于溢出是比较少见的，因此溢出器（spiller）测试起来通常也很困难。

lcc 的测试组件包括被称为故障报告的材料，但我们希望能保存更多信息。自 1988 年起 lcc

就在 AT&T 贝尔实验室、普林斯顿大学和其他许多地方使用，报告了很多错误、诊断和修正。电子新闻组对每处修正进行了总结，为贝尔实验室和普林斯顿大学的使用者提供了方便，这样他们可以判断是否需要抛弃旧版本的二进制程序。我们记录了所有的新闻信息，今后将进一步记录。

首先，我们记录了能够暴露故障的最小的输入数据。如果能找出这种输入，可以说工作完成了一半。有些故障报告只是记录 lcc 为程序生成了错误的代码，还给出了指向源代码目录的指针。如果只有一个庞大而费解的源程序和几千行的目标代码，想找出编译错误是很难的。我们通常从修整程序入手，直到某个修改使故障消除。我们发现，几乎所有的故障最终都可由不超过 5 行的样本代码展示出来。将来，我们会把这些程序和输入输出样本一起保存下来，形成测试材料，以便今后能够自动地重复进行测试。有的人认为某些难以想象的故障不会重复出现，我们必须消除这样的侥幸心理。有时当我们修正一个故障时有可能引发旧的故障，因此不得不再一次跟踪修改旧故障。如果这种情况出现一两次，上面测试材料的作用就体现出来了。

我们也将一些故障和修正故障的代码链接起来。lcc 最初不是文本程序，英语文本是后来加入代码中的。因此遇到某些编译器的程序片段时，我们不可能马上弄明白。这些程序片段主要用于解决故障，但是如果以前记录了大量的故障样本（即源代码和输入输出样本），并在旁边加上注释或是加上现在已经去掉了的文本程序部分，就可以节省许多时间。

另一类测试组件用于重定目标机器。如果需要为新的目标机器编写后端，我们必须先测试，然后才能实现整个目标代码生成器。或者，先实现一个编译器能够编译下面小程序：

```
main() {
   printf("Hello world\n");
}
```

当 lcc 能对其正确进行编译时，我们可以相信所有的简单函数调用都能正确处理（或许有点天真），并使用它们测试其他的基本特征，而这些基本特征又是其他大多数测试所必需的。整数赋值是典型的第 2 步：

```
main() {
   int i = 0;
   printf("%d\n", i);
}
```

我们用一组类似的程序继续进行这样的测试。每个都很简单，都完成一个新特征的测试，并且尽可能少地用到其他特征，为了尽量减少汇编程序代码大小和编译路径长度，必须看看测试程序是否失败。我们也许不会花时间来收集碎碎的测试程序作为将来重定位的指导，但从长远的眼光来看，这样做可以为我们节省许多时间，而当你需要为你经常使用的机器编写一个 lcc 的后端时，它能帮你节约不少时间。

深入阅读

Schreiner and Friedman（1985）介绍了如何使用 LEX（Lesk，1975）和 YACC（Johnson，1975）来构造一个小型语言的实验编译器。Holub（1990）和 Gray et al.（1992）介绍了这些编译器工具的新版本及其实现过程。

Budd（1991）介绍了面向对象程序设计和面向对象程序设计语言，包括 SmallTalk、C++、ObjectPascal 和 Objective-C。各种语言的参考手册都是关于这些语言的权威资料，如 C++（Ellis 和 Stroustrup，1990）、Oberon-2（Mössenbock and Wirth，1991）和 Modula-3（Nelson，1991）。

Ramsey（1993）将 lcc 改造成可重定位调试器 ldb 的表达式服务器。该服务器接收在调试过

程中遇到的 C 的表达式和符号表，按照提供的符号表对该表达式进行编译和计算。Ramsey 实现了一个后端，它生成 PostScript 而不是汇编语言。ldb 内嵌的 PostScript 解释器对生成的代码进行计算，并计算表达式的值。他还对 lcc 进行了修改以生成 ldb 的符号表。

Appel（1992）介绍了一种研究型的 ML 编译器，该编译器构造 AST，在编译过程中对 AST 扫描 30 多遍。

我们曾发表过论文介绍了早期的 lcc 版本（Fraser and Hanson，1991b），在大小、运行速度和编译器所生成的代码的运行速度等方面，将 lcc 与提供商的编译器做了比较，同时还与 VAX、Motorola、SPARC、MIPS R3000 上使用的 gcc 做了比较。lcc 生成的代码通常优于没有经过优化的商业编译器所生成的代码。另一篇相关的论文给出了一些测试数据，这些测试支持我们的直觉，即寄存器溢出是很少发生的（Fraser and Hanson，1992）。

Lamb（1981）介绍了一种典型的窥孔优化器。窥孔优化器 copt 非常简单，它能在网站 ftp.cs.princeton.edu 的目录 pub/lcc/contrib 下通过匿名 ftp 进行访问。Davidson and Fraser（1984）介绍了一种由目标机器的形式化描述所驱动的窥孔优化器。

Chaitin et al.（1981）通过图着色算法介绍了寄存器分配器，Krishnamurthy（1990）综述了许多有关指令调度的文献。Proebsting and Fischer（1991）介绍了一种将寄存器分配器与指令调度结合在一起的方法，这也是最简单的方法之一。

参考文献

Aho, A. V., and S. C. Johnson. 1974. LR parsing. *ACM Computing Surveys* 6(2), 99–124.

———. 1976. Optimal code generation for expression trees. *Journal of the ACM* 23(3), 488–501.

Aho, A. V., R. Sethi, and J. D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison Wesley.

American National Standards Institute, Inc. 1990. *American National Standard for Information Systems, Programming Language C ANSI X3.159-1989*. New York: American National Standards Institute, Inc.

Appel, A. W. 1991. Garbage collection. In P. Lee, Ed., *Topics in Advanced Language Implementation Techniques*, 89–100. Cambridge, MA: MIT Press.

———. 1992. *Compiling with Continuations*. Cambridge: Cambridge University Press.

Baskett, F. 1978. The best simple code generation technique for while, for, and do loops. *SIGPLAN Notices* 13(4), 31–32.

Bernstein, R. L. 1985. Producing good code for the case statement. *Software—Practice and Experience* 15(10), 1021–1024.

Boehm, H. J., and M. Weiser. 1988. Garbage collection in an uncooperative environment. *Software—Practice and Experience* 18(9), 807–820.

Budd, T. A. 1991. *An Introduction to Object-Oriented Programming*. Reading, MA: Addison Wesley.

Bumbulis, P., and D. D. Cowan. 1993. RE2C: A more versatile scanner generator. *ACM Letters on Programming Languages and Systems* 2(1–4), 70–84.

Burke, M. G., and G. A. Fisher. 1987. A practical method for LR and LL syntactic error diagnosis. *ACM Transactions on Programming Languages and Systems* 9(2), 164–197.

Chaitin, G. J., M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. 1981. Register allocation via coloring. *Journal of Computer Languages* 6, 47–57.

Cichelli, R. J. 1980. Minimal perfect hash functions made simple. *Communications of the ACM* 23(1), 17–19.

Clinger, W. D. 1990. How to read floating-point numbers accurately. *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 25(6), 92–101.

Davidson, J. W., and C. W. Fraser. 1984. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems* 6(4), 505–526.

Davie, A. J. T., and R. Morrison. 1981. *Recursive Descent Compiling*. New York: John Wiley & Sons.

Ellis, M. A., and B. Stroustrup. 1990. *The Annotated C++ Reference Manual*. Reading, MA: Addison Wesley.

Fischer, C. N., and R. J. LeBlanc, Jr. 1991. *Crafting a Compiler with C*. Redwood City, CA: Benjamin/Cummings.

Fraser, C. W. 1989. A language for writing code generators. *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 24(7), 238–245.

Fraser, C. W., and D. R. Hanson. 1991a. A code generation interface for ANSI C. *Software—Practice and Experience* 21(9), 963–988.

———. 1991b. A retargetable compiler for ANSI C. *SIGPLAN Notices* 26(10), 29–43.

———. 1992. Simple register spilling in a retargetable compiler. *Software—Practice and Experience* 22(1), 85–99.

Fraser, C. W., D. R. Hanson, and T. A. Proebsting. 1992. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems* 1(3), 213–226.

Fraser, C. W., R. R. Henry, and T. A. Proebsting. 1992. BURG—Fast optimal instruction selection and tree parsing. *SIGPLAN Notices* 27(4), 68–76.

Freiburghouse, R. A. 1974. Register allocation via usage counts. *Communications of the ACM* 17(11), 638–642.

Gray, R. W., V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. 1992. Eli: A complete, flexible compiler construction system. *Communications of the ACM* 35(2), 121–131.

Griswold, R. E. 1972. *The Macro Implementation of SNOBOL4*. San Francisco: W. H. Freeman.

Hansen, W. J. 1992. Subsequence references: First-class values for substrings. *ACM Transactions on Programming Languages and Systems* 14(4), 471–489.

Hanson, D. R. 1974. A simple technique for representing strings in Fortran IV. *Communications of the ACM* 17(11), 646–647.

———. 1983. Simple code optimizations. *Software—Practice and Experience* 13(8), 745–763.

———. 1985. Compact recursive-descent parsing of expressions. *Software—Practice and Experience* 15(12), 1205–1212.

———. 1990. Fast allocation and deallocation of memory based on object lifetimes. *Software—Practice and Experience* 20(1), 5–12.

Harbison, S. P., and G. L. Steele, Jr. 1991. *C: A Reference Manual* (third edition). Englewood Cliffs, NJ: Prentice Hall.

Hennessy, J. L., and N. Mendelsohn. 1982. Compilation of the Pascal case statement. *Software—Practice and Experience* 12(9), 879–882.

Heuring, V. P. 1986. The automatic generation of fast lexical analyzers. *Software—Practice and Experience* 16(9), 801–808.

Holub, A. I. 1990. *Compiler Design in C*. Englewood Cliffs, NJ: Prentice Hall.

Holzmann, G. J. 1988. *Beyond Photography*. Englewood Cliffs, NJ: Prentice Hall.

Intel Corp. 1993. *Intel486 Microprocessor Family Programmer's Reference Manual*. Intel Corp.

Jaeschke, G., and G. Osterburg. 1980. On Cichelli's minimal perfect hash function method. *Communications of the ACM* 23(12), 728–729.

Johnson, S. C. 1975. YACC—Yet another compiler compiler. Technical Report 32, Murray Hill, NJ: Computing Science Research Center, AT&T Bell Laboratories.

———. 1978. A portable compiler: Theory and practice. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, Tucson, AZ, 97–104.

Kane, G., and J. Heinrich. 1992. *MIPS RISC Architecture*. Englewood Cliffs, NJ: Prentice Hall.

Kannan, S., and T. A. Proebsting. 1994. Correction to 'producing good code for the case statement'. *Software—Practice and Experience* 24(2), 233.

Kernighan, B. W., and R. Pike. 1984. *The UNIX Programming Environment*. Englewood Cliffs, NJ: Prentice Hall.

Kernighan, B. W., and D. M. Ritchie. 1988. *The C Programming Language* (second edition). Englewood Cliffs, NJ: Prentice Hall.

Knuth, D. E. 1973a. *The Art of Computer Programming: Volume 1, Fundamental Algorithms* (second edition). Reading, MA: Addison Wesley.

———. 1973b. *The Art of Computer Programming: Volume 3, Searching and Sorting*. Reading, MA: Addison Wesley.

———. 1984. *The TeXBook*. Reading, MA: Addison Wesley.

———. 1992. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford, CA: Center for the Study of Language and Information, Stanford University.

Krishnamurthy, S. M. 1990. A brief survey of papers on scheduling for pipelined processors. *SIGPLAN Notices* 25(7), 97–106.

Lamb, D. A. 1981. Construction of a peephole optimizer. *Software—Practice and Experience* 11(12), 639–648.

Lesk, M. E. 1975. LEX – A lexical analyzer generator. Technical Report 39, Murray Hill, NJ: Computing Science Research Center, AT&T Bell Laboratories.

Logothetis, G., and P. Mishra. 1981. Compiling short-circuit Boolean expressions in one pass. *Software—Practice and Experience* 11(11), 1197–1214.

McKeeman, W. M., J. J. Horning, and D. B. Wortman. 1970. *A Compiler Generator*. Englewood Cliffs, NJ: Prentice Hall.

Mössenböck, H., and N. Wirth. 1991. The programming language Oberon-

2. *Structured Programming* 12(4), 179–195.

Nelson, G. 1991. *Systems Programming with Modula-3*. Englewood Cliffs, NJ: Prentice Hall.

Patterson, D. A., and J. L. Hennessy. 1990. *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann.

Pelegri-Llopart, E., and S. L. Graham. 1988. Optimal code generation for expression trees: An application of BURS theory. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, San Diego, CA, 294–308.

Proebsting, T. A. 1992. Simple and efficient BURS table generation. *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 27(6), 331–340.

Proebsting, T. A., and C. N. Fischer. 1991. Linear-time, optimal code scheduling for delayed-load architectures. *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 26(6), 256–267.

Ramsey, N. 1993. *A Retargetable Debugger*. Ph.D. diss., Princeton University, Princeton, NJ.

———. 1994. Literate programming simplified. *IEEE Software* 11(5), 97–105.

Ramsey, N., and D. R. Hanson. 1992. A retargetable debugger. *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 27(7), 22–31.

Richards, M., and C. Whitby-Strevens. 1979. *BCPL—The Language and Its Compiler*. Cambridge: Cambridge University Press.

Ritchie, D. M. 1993. The development of the C language. *Preprints of the Second ACM SIGPLAN History of Programming Languages Conference (HOPL-II), SIGPLAN Notices* 28(3), 201–208.

Sager, T. J. 1985. A polynomial time generator for minimal perfect hash functions. *Communications of the ACM* 28(5), 523–532.

Schreiner, A. T., and H. G. Friedman, Jr. 1985. *Introduction to Compiler Construction with UNIX*. Englewood Cliffs, NJ: Prentice Hall.

Sedgewick, R. 1990. *Algorithms in C*. Reading, MA: Addison Wesley.

Sethi, R. 1981. Uniform syntax for type expressions and declarators. *Software—Practice and Experience* 11(6), 623–628.

SPARC International. 1992. *The SPARC Architecture Manual, Version 8*. Englewood Cliffs, NJ: Prentice Hall.

Stallman, R. M. 1992. Using and porting GNU CC. Technical report, Cambridge, MA: Free Software Foundation.

Steele, Jr., G. L., and J. L. White. 1990. How to print floating-point numbers accurately. *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 25(6), 112–126.

Stirling, C. 1985. Follow set error recovery. *Software—Practice and Experience* 15(3), 239–257.

Tanenbaum, A. S., H. van Staveren, and J. W. Stevenson. 1982. Using peephole optimization on intermediate code. *ACM Transactions on Programming Languages and Systems* 4(1), 21–36.

Ullman, J. D. 1994. *Elements of ML Programming*. Englewood Cliffs, NJ: Prentice Hall.

Waite, W. M. 1986. The cost of lexical analysis. *Software—Practice and Experience* 16(5), 473–488.

Waite, W. M., and L. R. Carter. 1993. *An Introduction to Compiler Construction*. New York: Harper Collins.

Waite, W. M., and G. Goos. 1984. *Compiler Construction*. New York: Springer-Verlag.

Weinstock, C. B., and W. A. Wulf. 1988. Quick fit: An efficient algorithm for heap storage management. *SIGPLAN Notices* 23(10), 141–148.

Wilson, P. R. 1994. Uniprocessor garbage collection techniques. *ACM Computing Surveys* 27, to appear.

Wirth, N. 1976. *Algorithms + Data Structures = Programs*. Englewood Cliffs, NJ: Prentice Hall.

———. 1977. What can be done about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM* 20(11), 822–823.

可变目标C编译器 设计与实现

A Retargetable C Compiler Design and Implementation

lcc编译器是一个具有产品级质量的可变目标C编译器，由AT&T贝尔实验室和普林斯顿大学为ANSI C编程语言设计，在UNIX界广为流行。本书称得上是一个文本程序（literate program），包括lcc的源代码及说明，在代码级对系统的设计和实现进行了详细的介绍。本书全面而真实地展示了一个大型软件系统，介绍了C编译器的一种实现方法，而不是只针对编译过程中遇到的问题给出解决方案，非常适合自学使用，也适合在需要使用或实现基于语言的工具和技术的应用领域（如用户接口）工作的专业人员使用。

本书特点

- 只给出了理解lcc的实现所需的编译器理论，重点关注实际应用问题。
- 从编译器设计者的角度讲解C语言的特性，引导C程序员更好地掌握语言本身及其在现代计算机上的高效实现。
- 介绍了完整的代码生成器，代码生成面向MIPS R3000、SPARC和Intel 386及其后续产品等不同的平台。
- 提供了lcc编译器的完整源代码、3个后端以及代码生成器。

作者简介

克里斯多夫 W. 弗雷泽（Christopher W. Fraser）从1975年就开始研究编译技术，尤其对于从描述规范自动产生代码生成器这一技术有深入的研究，在该领域发表了多篇论文。他提出了可变目标的窥孔优化方法，该方法被广为流行的C编译器——GCC所采纳。从1977年到1985年，他在亚利桑那大学从事计算机科学（包括编译技术）的教学工作，1986年以后，他先后在AT&T贝尔实验室、微软研究院和谷歌公司任职，目前致力于独立项目的咨询和研究工作。

戴维 R. 汉森（David R. Hanson）谷歌公司软件工程师，于2012年退休。在加入谷歌公司之前，曾任职于微软研究院、普林斯顿大学、亚利桑那大学和耶鲁大学。他主持了与贝尔实验室的合作研究，是lcc的开发者之一。他的研究兴趣包括编程语言、编译器、软件工具和编程环境。

Pearson
www.pearson.com

ISBN 978-7-111-55258-1

定价：79.00元