"十二五"普通高等教育本科国家级规划教材

编译原理

(第3版)

陈意云　张　昱　编著

高等教育出版社

“十二五”普通高等教育本科国家级规划教材

编译原理

Bianyi Yuanli

（第3版）

陈意云　张　昱　编著

高等教育出版社·北京

内容提要

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

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

图书在版编目(CIP)数据

编译原理/陈意云，张昱编著. --3版. --北京：高等教育出版社，2014.9

ISBN 978-7-04-040491-3

Ⅰ.①编… Ⅱ.①陈… ②张… Ⅲ.①编译程序-程序设计-高等学校-教材 Ⅳ.①TP314

中国版本图书馆CIP数据核字(2014)第160816号

策划编辑 刘 艳　　责任编辑 张海波　　封面设计 于文燕　　版式设计 童 丹

插图绘制 郝 林　　责任校对 陈旭颖　　责任印制 朱学忠

出版发行 高等教育出版社
社 址 北京市西城区德外大街4号
邮政编码 100120
印 刷 北京鑫海金澳胶印有限公司
开 本 787mm×1092mm 1/16
印 张 28.5
字 数 650千字
购书热线 010-58581118
咨询电话 400-810-0598
网 址 http://www.hep.edu.cn
http://www.hep.com.cn
网上订购 http://www.landraco.com
http://www.landraco.com.cn
版 次 2003年8月第1版
2014年9月第3版
印 次 2014年9月第1次印刷
定 价 39.00元

本书如有缺页、倒页、脱页等质量问题，请到所购图书销售部门联系调换

版权所有 侵权必究

物 料 号 40491-00

第3版前言

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

编译器技术近年来的主要进展在面向新型计算机体系结构的优化方面,例如并行化和低功耗。本教材第2版已增加一章专门介绍依赖于机器的优化,再增加这方面内容会使得教材各部分的比例失调。另一方面,对高可信软件的需求也在推动编译器技术的研究,例如出具证明的编译器(certifying compiler)和经过验证的编译器(verified compiler),但相关成果尚未成熟到值得写入教材。

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

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

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

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

虽经多次改版,书中仍难免存在不妥和错误之处。若发现本书有错误或对本书提出建议,欢迎发送电子邮件(yiyun@ustc.edu.cn,yuzhang@ustc.edu.cn)批评指正,作者将与出版社联系,在再次印刷或再版时改正错误和采纳合理的建议。

作　者
于中国科学技术大学
2013年8月

第2版前言

本书改版的动力主要来自下列几个方面:

(1) 编译器技术的发展,特别是计算机体系结构的发展对编译器技术的推动;

(2) 第1版教材在使用中得到的积极反馈和建议;

(3) 作者在教学实践中的心得和体会,在相关科研活动中的进展与收获。

第2版增加了有关新理论和新技术的介绍,重写了因理论和技术发展而需要调整的部分,改写了原先有疏漏而需要改善的地方,并且继承和发扬了前一版的特色。改动主要体现在:

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

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

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

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

本次改版主要参考了文献[9],在此向作者表示衷心的感谢。

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

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

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

作　者

于中国科学技术大学

2008年3月

第1版前言

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

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

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

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

编译技术在软件安全、程序理解和软件逆向工程等方面有着广泛的应用。

作为一本教材，本书有如下一些特点：

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

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

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

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

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

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

(7) 为了便于读者学习，本书配有习题解答（见参考文献8）。

本书的多数章节是参考了参考文献1和参考文献7编写的，部分习题取自参考文献8。在此向有关作者表示感谢。

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

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

由于作者水平有限，书中难免存在一些缺点和错误，恳请广大读者批评指正。

作　者

于中国科学技术大学

2003 年 5 月

郑重声明

高等教育出版社依法对本书享有专有出版权。任何未经许可的复制、销售行为均违反《中华人民共和国著作权法》，其行为人将承担相应的民事责任和行政责任；构成犯罪的，将被依法追究刑事责任。为了维护市场秩序，保护读者的合法权益，避免读者误用盗版书造成不良后果，我社将配合行政执法部门和司法机关对违法犯罪的单位和个人进行严厉打击。社会各界人士如发现上述侵权行为，希望及时举报，本社将奖励举报有功人员。

反盗版举报电话 (010)58581897 58582371 58581879

反盗版举报传真 (010)82086060

反盗版举报邮箱 dd@hep.com.cn

通信地址 北京市西城区德外大街4号 高等教育出版社法务部

邮政编码 100120

目　　录

第1章　引论 …… 1
1.1　编译器概述 …… 1
1.1.1　词法分析 …… 1
1.1.2　语法分析 …… 3
1.1.3　语义分析 …… 3
1.1.4　中间代码生成 …… 4
1.1.5　代码优化 …… 4
1.1.6　代码生成 …… 5
1.1.7　符号表管理 …… 5
1.1.8　阶段的分组 …… 6
1.1.9　解释器 …… 7
1.2　编译器技术的应用 …… 7
1.2.1　高级语言的实现 …… 8
1.2.2　针对计算机体系结构的优化 …… 9
1.2.3　新计算机体系结构的设计 …… 10
1.2.4　程序翻译 …… 10
1.2.5　提高软件开发效率的工具 …… 11
习题1 …… 12
第2章　词法分析 …… 13
2.1　词法记号及属性 …… 13
2.1.1　词法记号、模式、词法单元 …… 14
2.1.2　词法记号的属性 …… 15
2.1.3　词法错误 …… 16
2.2　词法记号的描述与识别 …… 16
2.2.1　串和语言 …… 16
2.2.2　正规式 …… 18
2.2.3　正规定义 …… 19
2.2.4　状态转换图 …… 20
2.3　有限自动机 …… 23
2.3.1　不确定的有限自动机 …… 24
2.3.2　确定的有限自动机 …… 25
2.3.3　NFA到DFA的变换 …… 26
2.3.4　DFA的化简 …… 29
2.4　从正规式到有限自动机 …… 31
2.5　词法分析器的生成器 …… 34
习题2 …… 37
第3章　语法分析 …… 40
3.1　上下文无关文法 …… 40
3.1.1　上下文无关文法的定义 …… 41
3.1.2　推导 …… 42
3.1.3　分析树 …… 43
3.1.4　二义性 …… 44
3.2　语言和文法 …… 45
3.2.1　正规式和上下文无关文法的比较 …… 45
3.2.2　分离词法分析器的理由 …… 46
3.2.3　验证文法产生的语言 …… 46
3.2.4　适当的表达式文法 …… 47
3.2.5　消除二义性 …… 48
3.2.6　消除左递归 …… 49
3.2.7　提左因子 …… 50
*3.2.8　非上下文无关的语言构造 …… 51
*3.2.9　形式语言鸟瞰 …… 52
3.3　自上而下分析 …… 54
3.3.1　自上而下分析的一般方法 …… 54
3.3.2　LL(1)文法 …… 55
3.3.3　递归下降的预测分析 …… 56
3.3.4　非递归的预测分析 …… 57

3.3.5　构造预测分析表 …… 60
3.3.6　预测分析的错误恢复 …… 61
3.4　自下而上分析 …… 65
3.4.1　归约 …… 65
3.4.2　句柄 …… 65
3.4.3　用栈实现移进-归约分析 …… 66
3.4.4　移进-归约分析的冲突 …… 68
3.5　LR 分析器 …… 69
3.5.1　LR 分析算法 …… 69
3.5.2　LR 文法和 LR 分析方法的特点 …… 73
3.5.3　构造 SLR 分析表 …… 74
3.5.4　构造规范的 LR 分析表 …… 81
3.5.5　构造 LALR 分析表 …… 85
3.5.6　非二义且非 LR 的上下文无关文法 …… 88
3.6　二义文法的应用 …… 89
3.6.1　使用算符的优先级和结合性来解决冲突 …… 90
3.6.2　使用其他约定来解决冲突 …… 92
3.6.3　LR 分析的错误恢复 …… 93
3.7　语法分析器的生成器 …… 95
3.7.1　分析器的生成器 Yacc …… 95
3.7.2　用 Yacc 处理二义文法 …… 98
3.7.3　Yacc 的错误恢复 …… 101
习题 3 …… 103
第 4 章　语法制导的翻译 …… 109
4.1　语法制导的定义 …… 109
4.1.1　语法制导定义的形式 …… 110
4.1.2　综合属性 …… 111
4.1.3　继承属性 …… 111
4.1.4　属性依赖图 …… 112
4.1.5　属性计算次序 …… 113
4.2　*S* 属性定义的自下而上计算 …… 114
4.2.1　语法树 …… 115
4.2.2　构造语法树的语法制导定义 …… 115
4.2.3　*S* 属性的自下而上计算 …… 117
4.3　*L* 属性定义的自上而下计算 …… 119
4.3.1　*L* 属性定义 …… 120
4.3.2　翻译方案 …… 120
4.3.3　预测翻译器的设计 …… 124
4.3.4　用综合属性代替继承属性 …… 126
4.4　*L* 属性的自下而上计算 …… 126
4.4.1　删除翻译方案中嵌入的动作 …… 127
4.4.2　分析栈上的继承属性 …… 127
4.4.3　模拟继承属性的计算 …… 130
习题 4 …… 132
第 5 章　类型检查 …… 135
5.1　类型在编程语言中的作用 …… 135
5.1.1　执行错误和安全语言 …… 136
5.1.2　类型化语言和类型系统 …… 136
5.1.3　类型化语言的优点 …… 139
5.2　类型系统的描述语言 …… 139
5.2.1　定型断言 …… 140
5.2.2　定型规则 …… 141
5.2.3　类型检查和类型推断 …… 142
5.3　一个简单类型检查器的规范 …… 143
5.3.1　一个简单的语言 …… 143
5.3.2　类型系统 …… 143
5.3.3　类型检查 …… 145
5.3.4　类型转换 …… 147
*5.4　多态函数 …… 148
5.4.1　为什么要使用多态函数 …… 148
5.4.2　类型变量 …… 149
5.4.3　一个含多态函数的语言 …… 151
5.4.4　代换、实例与合一 …… 153
5.4.5　多态函数的类型检查 …… 154
5.5　类型表达式的等价 …… 158
5.5.1　类型表达式的结构等价 …… 158
5.5.2　类型表达式的名字等价 …… 159

5.5.3 记录类型 …… 160
5.5.4 类型表示中的环 …… 161
5.6 函数和算符的重载 …… 162
5.6.1 子表达式的可能类型集合 …… 162
5.6.2 缩小可能类型的集合 …… 163
习题5 …… 164
第6章 运行时存储空间的组织和管理 …… 171
6.1 局部存储分配 …… 172
6.1.1 过程 …… 172
6.1.2 名字的作用域和绑定 …… 172
6.1.3 活动记录 …… 174
6.1.4 局部数据的布局 …… 175
6.1.5 程序块 …… 175
6.2 全局栈式存储分配 …… 177
6.2.1 运行时内存的划分 …… 177
6.2.2 活动树和运行栈 …… 178
6.2.3 调用序列 …… 180
6.2.4 栈上可变长度数据 …… 183
6.2.5 悬空引用 …… 183
6.3 非局部名字的访问 …… 184
6.3.1 无过程嵌套的静态作用域 …… 185
*6.3.2 有过程嵌套的静态作用域 …… 185
*6.3.3 动态作用域 …… 189
6.4 参数传递 …… 190
6.4.1 值调用 …… 190
6.4.2 引用调用 …… 191
6.4.3 换名调用 …… 191
*6.5 堆管理 …… 192
6.5.1 内存管理器 …… 192
6.5.2 计算机内存分层 …… 193
6.5.3 程序局部性 …… 195
6.5.4 手工回收请求 …… 195
习题6 …… 196
第7章 中间代码生成 …… 212
7.1 中间语言 …… 213
7.1.1 后缀表示 …… 213
7.1.2 图形表示 …… 213
7.1.3 三地址代码 …… 214
7.1.4 静态单赋值形式 …… 216
7.2 声明语句 …… 217
7.2.1 过程中的声明 …… 217
7.2.2 作用域信息的保存 …… 218
7.2.3 记录的域名 …… 220
7.3 赋值语句 …… 221
7.3.1 符号表中的名字 …… 221
7.3.2 数组元素的地址计算 …… 222
7.3.3 数组元素地址计算的翻译方案 …… 223
7.3.4 类型转换 …… 225
7.4 布尔表达式和控制流语句 …… 226
7.4.1 布尔表达式 …… 227
7.4.2 控制流语句的翻译 …… 227
7.4.3 布尔表达式的控制流翻译 …… 229
7.4.4 开关语句的翻译 …… 231
7.4.5 过程调用的翻译 …… 233
习题7 …… 234
第8章 代码生成 …… 241
8.1 代码生成器设计中的问题 …… 241
8.1.1 目标程序 …… 241
8.1.2 指令选择 …… 242
8.1.3 寄存器分配 …… 243
8.1.4 计算次序选择 …… 243
8.2 目标语言 …… 244
8.2.1 目标机器的指令集 …… 244
8.2.2 指令代价 …… 245
8.3 基本块和流图 …… 247
8.3.1 基本块 …… 248
8.3.2 基本块的优化 …… 249
8.3.3 流图 …… 250
8.3.4 下次引用信息 …… 251

8.4 一个简单的代码生成器 …… 252
8.4.1 寄存器描述和地址描述 …… 252
8.4.2 代码生成算法 …… 253
8.4.3 寄存器选择函数 …… 254
8.4.4 为变址和指针语句产生代码 …… 255
8.4.5 条件语句 …… 256
习题 8 …… 257
*第 9 章 独立于机器的优化 …… 267
9.1 优化的主要种类 …… 267
9.1.1 优化的主要源头 …… 268
9.1.2 一个实例 …… 268
9.1.3 公共子表达式删除 …… 270
9.1.4 复写传播 …… 272
9.1.5 死代码删除 …… 273
9.1.6 代码外提 …… 274
9.1.7 强度削弱和归纳变量删除 …… 274
9.2 数据流分析介绍 …… 276
9.2.1 数据流抽象 …… 277
9.2.2 数据流分析模式 …… 278
9.2.3 到达-定值 …… 279
9.2.4 活跃变量 …… 284
9.2.5 可用表达式 …… 285
9.2.6 小结 …… 287
9.3 数据流分析的基础 …… 288
9.3.1 半格 …… 289
9.3.2 迁移函数 …… 291
9.3.3 一般框架的迭代算法 …… 293
9.3.4 数据流解的含义 …… 295
9.4 常量传播 …… 297
9.4.1 常量传播框架的数据流值 …… 297
9.4.2 常量传播框架的迁移函数 …… 298
9.4.3 常量传播框架的单调性 …… 298
9.4.4 常量传播框架的非分配性 …… 299
9.4.5 结果的解释 …… 300
9.5 部分冗余删除 …… 301
9.5.1 冗余的根源 …… 301
9.5.2 能否删除所有的冗余 …… 303
9.5.3 惰性代码移动问题 …… 304
9.5.4 预期表达式 …… 305
9.5.5 惰性代码移动算法 …… 305
9.6 流图中的循环 …… 310
9.6.1 支配结点 …… 311
9.6.2 回边和可归约性 …… 312
9.6.3 流图的深度 …… 314
9.6.4 自然循环 …… 314
9.6.5 迭代流图算法的收敛速度 …… 315
习题 9 …… 317
*第 10 章 依赖于机器的优化 …… 329
10.1 处理器体系结构 …… 330
10.1.1 指令流水线和分支延迟 …… 330
10.1.2 流水化的执行 …… 331
10.1.3 多指令发射 …… 331
10.2 代码调度的约束 …… 332
10.2.1 数据相关 …… 332
10.2.2 发现内存访问中的相关性 …… 333
10.2.3 寄存器使用和并行执行之间的折中 …… 334
10.2.4 寄存器分配和代码调度的次序安排 …… 335
10.2.5 控制相关 …… 336
10.2.6 投机执行的支持 …… 336
10.2.7 一个基本的机器模型 …… 337
10.3 基本块调度 …… 338
10.3.1 数据依赖图 …… 338
10.3.2 基本块的表调度 …… 340
10.3.3 区分优先级的拓扑次序 …… 341
10.4 全局代码调度 …… 341
10.4.1 简单的代码移动 …… 342
10.4.2 向上的代码移动 …… 343
10.4.3 向下的代码移动 …… 344

10.4.4 更新数据相关 …… 345
10.4.5 全局调度的其他问题 …… 345
10.4.6 静态调度器和动态调度器的相互影响 …… 346
10.5 软件流水 …… 346
10.5.1 引言 …… 347
10.5.2 循环的软件流水 …… 348
10.5.3 寄存器分配和代码生成 …… 350
10.5.4 Do-Across 循环 …… 351
10.5.5 软件流水的目标和约束 …… 352
10.5.6 软件流水算法 …… 354
10.5.7 无环数据依赖图的调度 …… 355
10.6 并行性和数据局部性优化概述 …… 356
10.6.1 多处理器 …… 357
10.6.2 应用中的并行性 …… 358
10.6.3 循环级并行 …… 359
10.6.4 数据局部性 …… 360
10.6.5 矩阵乘法算法 …… 362
10.6.6 矩阵乘法算法的优化 …… 364
习题 10 …… 365
第 11 章 编译系统和运行时系统 …… 369
11.1 C 语言的编译系统 …… 369
11.1.1 预处理器 …… 370
11.1.2 汇编器 …… 371
11.1.3 连接器 …… 371
11.1.4 目标文件的格式 …… 373
11.1.5 符号解析 …… 375
11.1.6 静态库 …… 376
11.1.7 可执行目标文件及装入 …… 378
11.1.8 动态连接 …… 379
11.1.9 处理目标文件的一些工具 …… 381
11.2 Java 语言的运行时系统 …… 381
11.2.1 Java 虚拟机语言简介 …… 382
11.2.2 Java 虚拟机 …… 382
11.2.3 即时编译器 …… 384
*11.3 无用单元收集 …… 386
11.3.1 标记和清扫 …… 386
11.3.2 引用计数 …… 388
11.3.3 复制收集 …… 388
11.3.4 分代收集 …… 390
11.3.5 渐增式收集 …… 391
11.3.6 编译器与收集器之间的相互影响 …… 391
习题 11 …… 394
***第 12 章 面向对象语言的编译** …… 399
12.1 面向对象语言的概念 …… 399
12.1.1 对象和对象类 …… 399
12.1.2 继承 …… 400
12.1.3 信息封装 …… 402
12.2 方法的编译 …… 402
12.3 继承的编译方案 …… 405
12.3.1 单一继承的编译方案 …… 406
12.3.2 多重继承的编译方案 …… 408
习题 12 …… 412
***第 13 章 函数式语言的编译** …… 415
13.1 函数式编程语言简介 …… 415
13.1.1 语言构造 …… 415
13.1.2 参数传递机制 …… 417
13.1.3 变量的自由出现和约束出现 …… 418
13.2 函数式语言的编译简介 …… 419
13.2.1 几个受启发的例子 …… 420
13.2.2 编译函数 …… 422
13.2.3 环境与约束 …… 422
13.3 抽象机的体系结构 …… 423
13.3.1 抽象机的栈 …… 424
13.3.2 抽象机的堆 …… 425
13.3.3 名字的寻址 …… 426
13.3.4 约束的建立 …… 427

13.4 指令集和编译 …………………… 428
13.4.1 表达式 …………………… 428
13.4.2 变量的引用性出现 ………… 430
13.4.3 函数定义 …………………… 431
13.4.4 函数应用 …………………… 432
13.4.5 构造和计算闭包 ………… 436
13.4.6 letrec 表达式和局部变量 …… 437
习题 13 ……………………………… 439
参考文献 ……………………………… 441

第 1 章

从理论上说,构造专用计算机来直接执行用某种高级语言编写的程序是可能的。但实际上,目前的计算机能执行的都是非常低级的**机器语言**。那么,一个基本问题是:用高级语言编写的程序是怎样变成能在计算机上执行的机器语言程序的?

能够完成从一种语言到另一种语言的保语义变换的软件称为**翻译器**,这两种语言分别称为该翻译器的**源语言**和**目标语言**。**编译器**是一种翻译器,它的特点是目标语言比源语言低级。

本章通过简要描述编译器的各个组成部分以及编译器技术的各种应用来介绍编译这个课题。该课题涉及编程语言、计算机体系结构、形式语言理论、类型论、算法和软件工程等方面的知识。

1.1 编译器概述

编译器的工作可以分成若干阶段,每个阶段把源程序从一种表示变换成另一种表示。编译过程的一种典型分解见图 1.1,图中左边一列的每个方框表示它的一个阶段。

本节以赋值语句

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

的翻译(假定变量都是实型)为例,简要介绍编译的各个阶段。

1.1.1 词法分析

词法分析阅读构成源程序的字符流,按编程语言的词法规则把它们组成词法**记号**(token)流。对于一个词法单元,词法分析产生的记号是

〈记号名,属性值〉

二元组。记号名是同类词法单元共用的名称,而属性值是一个词法单元有别于同类中其他词法单元的特征值。赋值语句(1.1)的字符流在词法分析时被依次组成下面这些记号。

(1) 标识符 position 形成的记号是〈**id**, 1〉,其中 **id** 是标识符的总称,1 代表 position 在符号

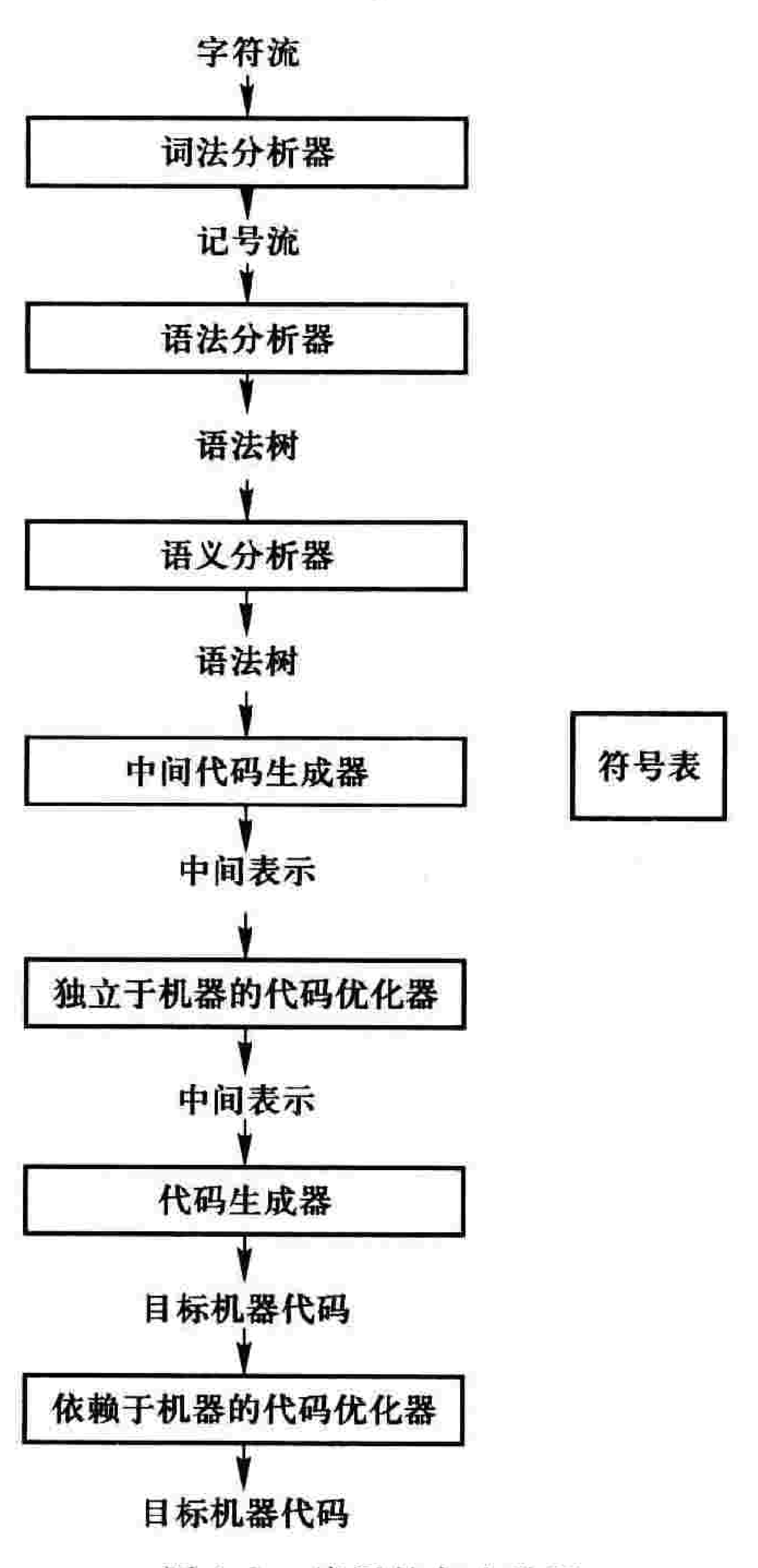


图 1.1 编译的各个阶段

表中的条目，符号表的条目用来存放标识符的各种属性，如它的名字和类型。

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

(3) 标识符 initial 形成的记号是〈**id**, 2〉。

(4) 加号 + 形成的记号是〈 + 〉。

(5) 标识符 rate 形成的记号是〈**id**, 3〉。

(6) 乘号 * 形成的记号是〈 * 〉。

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

分隔单词的空格通常在词法分析时被删去。

于是,赋值语句(1.1)在词法分析后形成的记号流是

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

第 2 章讨论词法分析,词法分析也称为**线性分析**或**扫描**。

1.1.2 语法分析

语法分析(syntax analysis)简称**分析**(parsing),它检查词法分析输出的记号流是否符合编程语言的语法规则,并依据这些规则所体现出的语言**构造**(construct,如函数、语句、表达式等)的层次性,用各记号的第一元建成一种树形的中间表示,这个中间表示用抽象语法的方式描绘了该记号流的语法情况。一种典型的中间表示是**语法树**,其中内部结点表示运算,它们的子结点代表该运算的运算对象。为记号流(1.2)建立的语法树见图 1.2(a)。

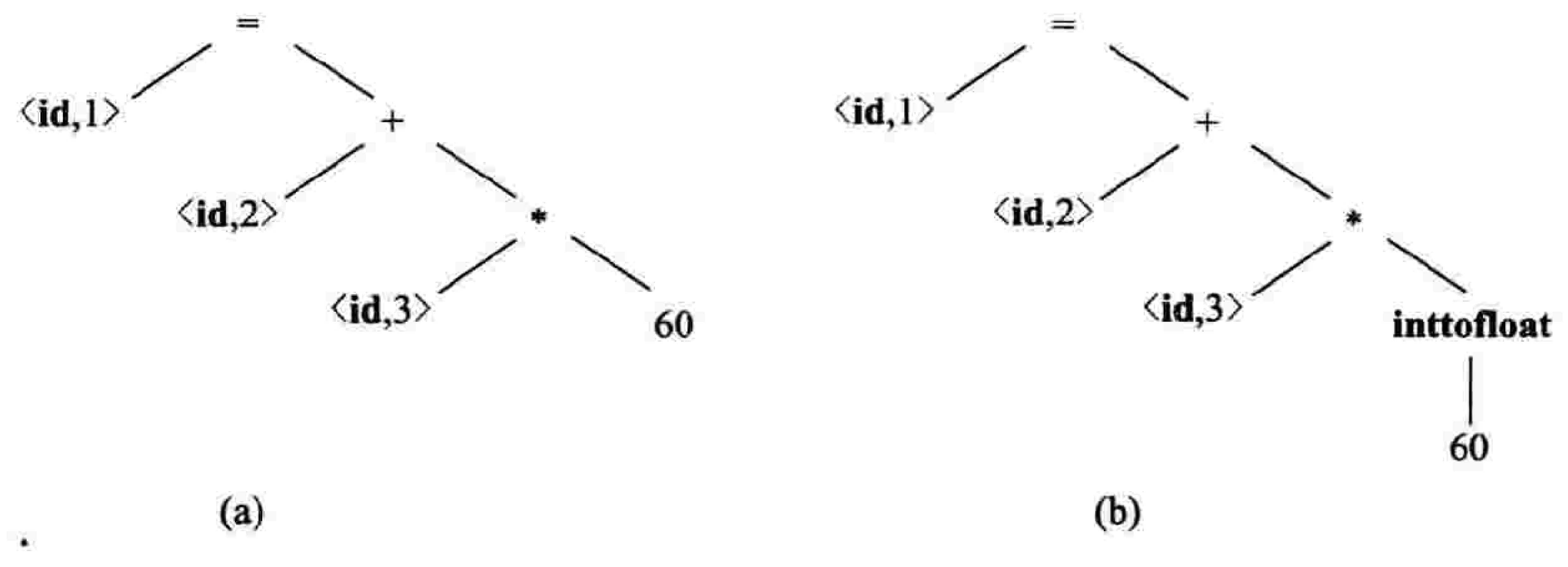


图 1.2 语义分析插入了类型转换

编译器后面各个阶段使用这种语法树进一步分析源程序和生成目标代码。第 3 章专门讨论语法分析算法,图 1.2 这种形式的语法树将在 4.2 节讨论。在第 4 章中,还将详细讨论编译器如何利用输入所含的层次结构来产生语法树。

1.1.3 语义分析

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

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

例 1.1 在机器内部,整数的二进制表示和实数的二进制表示是有区别的,不论它们是否有

相同的值。在图 1.2 中,所有的变量都是实型,另外,由 60 本身可知它是整数。对图 1.2(a)进行类型检查会发现 * 作用于实型变量 rate 和整数 60,可以建立一个额外的算符结点 **inttofloat** [见图 1.2(b)],它显式地把整数转变为实数。 □

类型检查和语义分析将在第 5 章讨论。

1.1.4 中间代码生成

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

第 7 章主要采用一种称为**三地址代码**的中间表示形式,它像一种抽象机器的汇编语言,这种机器中存储单元的作用类似于寄存器。三地址代码由三地址指令序列组成,每条三地址指令最多有三个操作数,语句(1.1)的三地址代码如下:

```
t1 = inttofloat(60)
t2 = id3 * t1                                    (1.3)
t3 = id2+t2
id1 = t3
```

这种中间形式有它的特点。首先,除了赋值算符外,每条指令至多还有一个算符,因此,在生成这些指令时,编译器必须决定运算完成的次序,语句(1.1)的乘优先于加。其次,编译器必须产生临时变量名,用以保留每条指令的计算结果。第三,某些三地址指令的运算对象不足三个,例如指令序列(1.3)的第一条和最后一条指令。

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

1.1.5 代码优化

独立于机器的代码优化阶段试图改进中间代码,以便产生较好的目标代码。通常,"较好"是指执行较快,但也可能期望其他目标,如目标代码较短或目标代码执行时能耗较低。

如果中间代码生成算法比较简单,那么它就给代码优化留下了很多机会。例如,一个比较自然的中间代码生成算法,为语法树上的每个算符产生一条指令,因而得到(1.3)这样的中间代码。用这样的中间代码生成算法再跟进一个代码优化阶段是一种可行的办法,因为产生较优代码问题可以在代码优化阶段得以解决。比如,代码优化器会推断出,把 60 从整数转变为浮点数可以在编译时完成,从而用 60.0 代替 60 就可以把 **inttofloat** 运算删去。还有,t3 只被引用一次,就是取它的值传给 id1,因此用 id1 代替 t3,把(1.3)的最后一条指令删除也是可以的。这样,优

化器可以得到如下结果：

$$t1 = id3 * 60.0$$
$$id1 = id2 + t1 \tag{1.4}$$

不同的编译器所实现的优化程度是不同的，能完成大部分优化的编译器称为"优化编译器"，但这时编译时间中相当可观的一部分都消耗在这种优化上。简单的优化也可以使目标程序的运行时间大大缩短，而编译速度并没有降低太多。第9章和第10章分别讨论独立于机器和依赖于机器的优化，第8章代码生成也会涉及代码优化。

1.1.6 代码生成

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

例如，使用寄存器R1和R2，(1.4)的中间代码可以翻译成：

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

每条指令的第一个和第二个操作数分别代表源和目的操作数，每条指令的"F"告知指令处理浮点数。(1.5)的代码把地址id3的内容取入寄存器R2（在此认为指令中id3代表对象id3的地址。变量的存储分配将在第6章讨论），然后把它乘上实数60.0，#号代表60.0作为立即数处理。第三条指令把id2取入寄存器R1，第四条指令再把寄存器R2的值加上去，最后一条指令把寄存器R1的值存入地址id1。这样，该段代码实现了语句(1.1)的赋值。代码生成将在第8章讨论。

1.1.7 符号表管理

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

符号表是为每个变量名字保存一个记录的数据结构，记录的域是该名字的属性。该数据结构应该设计成允许编译器迅速地找到一个名字的记录，并在此记录中迅速地存储和读取数据。符号表将在第7章讨论。

语句(1.1)的编译过程总结在图1.3中。

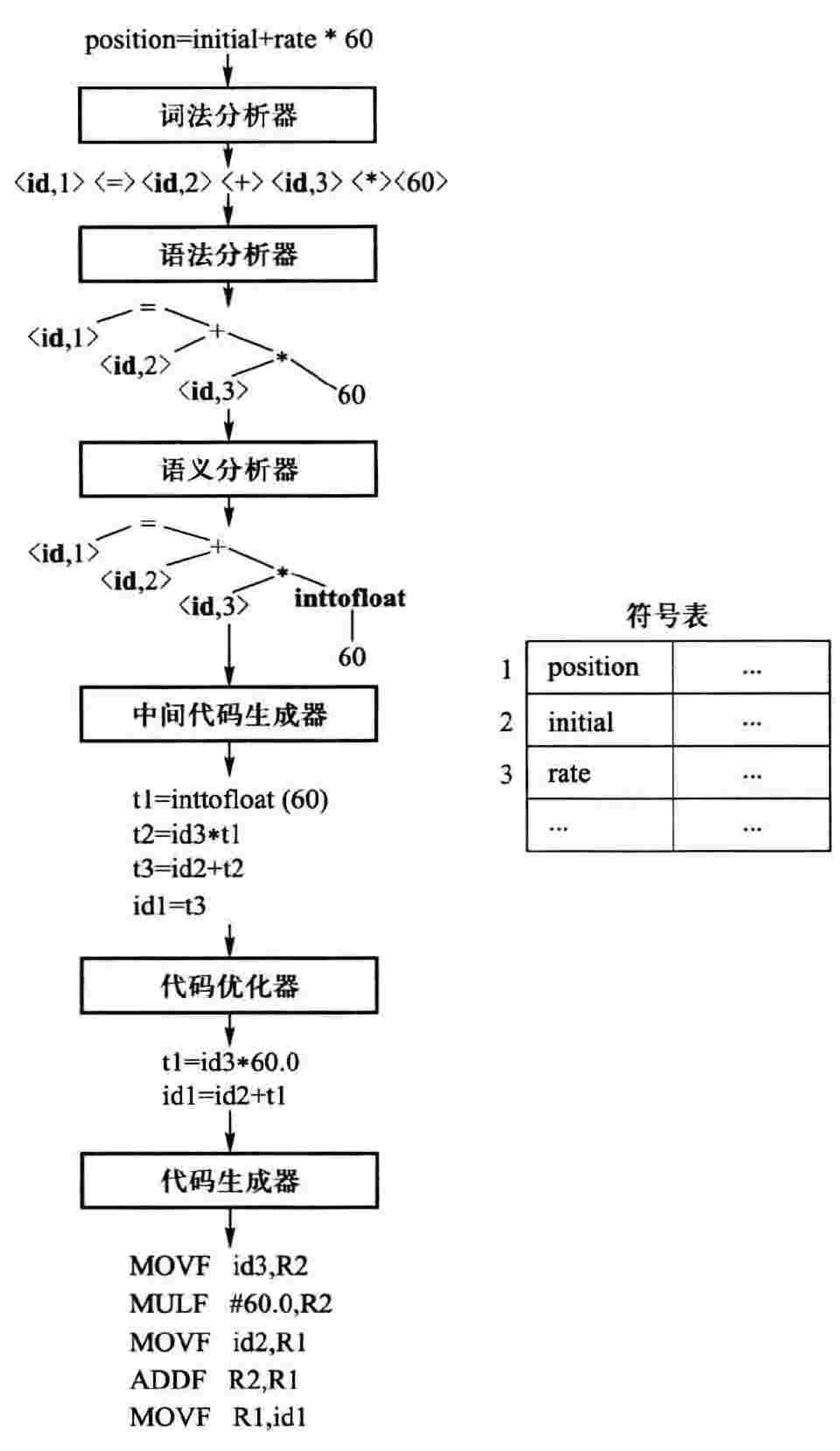


图 1.3 一个语句的翻译

1.1.8 阶段的分组

上面的介绍把编译器从逻辑上分成了7个阶段。最粗略的看法是把编译器分成分析和综合两大部分。分析部分揭示源程序的基本元素和它们所形成的层次结构,决定它们的含义,建立起源程序的中间表示,分析部分经常被称为**前端**。综合部分根据源程序的中间表示建立起和源程

序等价的目标程序,它经常被称为**后端**。

在实际的编译器中,源于几个阶段的活动可以组合在一起,各阶段之间的中间表示也无须显式构造。源于几个阶段的活动常用一**遍**(pass)扫描来实现,一遍扫描包括读一个输入文件和写一个输出文件。例如,前端的词法分析、语法分析、语义分析和中间代码生成可以组成一遍,独立于机器的代码优化可以单独作为可选择的一遍,然后为特定目标机器的代码生成可以作为一个后端遍。

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

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

1.1.9 解释器

解释器是不同于编译器的另一类语言处理器。解释器不像编译器那样通过翻译来生成目标程序,而是直接执行源程序所指定的运算。解释器也有和编译器类似的地方,它也需要对源程序进行词法分析、语法分析和语义分析等,这样它才有可能知道源程序指定了一些什么运算。

解释执行的效率比编译器生成的机器代码的执行效率低。对于编译方式来说,对源程序的词法分析、语法分析和语义分析只要进行一次。而对于解释执行来说,每次执行到源程序的某个语句,都要对它进行一次词法分析、语法分析和语义分析,确定了这个语句的含义后,才能执行它指定的运算。显然,反复分析循环体降低了解释执行的效率,所以解释执行要寻找一种适合于解释的中间语言,以缩短反复分析源程序需要的时间。

在 20 世纪 80 年代的 BASIC 语言阶段,解释器的功能是这样介绍的:它将高级语言的源程序翻译成一种中间语言程序,然后对中间语言程序进行解释执行。在那个年代,解释器的两个功能(编译和解释)是合在一个程序中的,因此这个程序被统称为解释器。进入 Java 语言年代,解释器的上述两个功能分离在两个程序中,前一个程序称编译器,它把 Java 语言的程序翻译成一种中间语言程序,这种中间语言叫做字节码;后一个程序称解释器,它对字节码程序进行解释执行。

1.2 编译器技术的应用

真正从事主流编程语言编译器设计的虽然只是极少数人,但是编译器技术还有着其他重要的应用,这是学习编译器技术的主要理由。此外,编译器的设计还影响着计算机科学的其他一些领域。本节回顾编译器设计与计算机科学其他主要领域的相互影响和编译器技术的重要应用。

1.2.1 高级语言的实现

一种高级语言定义了一种编程抽象:程序员用这种语言表达算法,编译器将程序翻译为目标语言。一般来说,高级语言易于编程,但是所得程序运行较慢。用低级语言编程时可以在程序中实施更有效的控制方式,原则上说,能得到更有效的代码。不幸的是,低级语言程序难编写、易出错;更糟糕的是,它难维护、缺乏移植性。优化编译器包括很多改进所生成代码性能的技术,从而弥补了高级抽象导入的低效。

历史上,流行编程语言的大多数改变都是朝着提高抽象级别的方向发展。20 世纪 80 年代 C 语言占据了系统编程的统治地位,90 年代启动的很多项目选择了 C++,1995 年问世的 Java 语言在 90 年代后期迅速流行。每一轮编程语言新特征的出现都促进了对编译器优化的新研究。下面回顾激励编译器技术显著推进的主要语言特征。

所有通用编程语言,包括 C、FORTRAN 和 COBOL,都支持用户定义的聚合数据类型,如数组和记录,也都支持高级控制流,如循环和过程调用。如果逐句把程序直接翻译成机器代码,则结果程序非常低效。作为编译器优化主体的数据流分析和优化,其功能是分析数据通过程序时的流动情况并进行多种优化,所生成代码的效率相当于有经验的程序员用低级语言写出程序的效率。数据流优化至今仍在被广泛研究。

面向对象的概念于 1967 年首次出现在 Simula 67 语言中,以后逐步被加入到各种语言中,如 Smalltalk、C++、C#和 Java。面向对象的主要概念是数据抽象和性质继承,它们都使得程序更加模块化并易于维护。面向对象的程序不同于用其他语言写的程序,其中类的定义中可能包含许多较小的过程(在面向对象语言中称之为方法)。因此,对于源程序中跨越过程边界的操作,编译器优化必须作适当处理才能保证效率。在这个场合,用过程体替换过程调用的过程内联(inlining)特别有用。加速虚方法调度(dispatch)的优化也一直在开发。

Java 有很多特征使编程变得更容易,其中大部分已被应用于之前的语言中。Java 语言是类型安全的,这是指一个对象不会被当作一个不相关类型的对象来使用。所有数组访问都会被检查以保证这些访问在相应数组的边界内。Java 中没有指针,也不允许指针作算术运算。它采用无用单元收集(俗称垃圾收集)机制来自动地回收那些不再被使用的变量所占据的内存。这些特征虽然使编程变得容易了,但是它们会引起运行开销的增加。人们也关注于研究有关降低开销的编译器优化技术,例如,删除不必要的边界检查,把离开过程后不再访问的对象分配到栈上而不是堆上。极小化无用单元收集开销的算法也一直在开发。

此外,Java 支持代码移植和代码移动。程序被分发成 Java 字节码的形式,字节码必须被解释或动态地编译成本地代码。在运行时能动态地抽取信息的场合,动态编译可用来生成较好优化的代码。在动态优化中,很重要的一点是极小化编译所需时间,因为它是执行开销的一部分。现在通用的一种技术是仅编译和优化程序中经常执行的部分。

1.2.2 针对计算机体系结构的优化

计算机体系结构的迅速演化引起不满足于现有编译器技术的新的需求，几乎所有的高性能系统都在利用两种基本技术：并行化和内存分层。并行性可分别在指令级和处理器级挖掘；而内存分层则面临以下基本局限：构造非常快的存储器或者非常大的存储器是可能的，但是构造不出既快又大的存储器。

所有现代微处理器都开发了指令级的并行。这种并行对程序员是透明的，程序员理解为串行执行的指令序列，会被硬件动态地检查其中的相关性，并尽可能并行发射这些指令。在有些情况下，机器包括一个硬件调度器，它可以改变指令的排序以增加程序中的并行。不管硬件是否重排指令的次序，编译器总可以重新整理指令，使得指令级的并行更有效。

指令级并行也可以显式出现在指令集中，超长指令字机器中有能够发射并行乘法运算的指令，如 Intel 的 IA64 就是这种体系结构的一个著名例子。所有高性能通用微处理器也包括能同时对数据向量进行运算的指令。自动地为串行程序生成这类机器代码的编译器技术也一直是研究热点。

多处理器已经开始流行，甚至个人计算机也有多个处理器。程序员可以为多处理器写多线程代码，或者编译器可以从传统的串行程序自动生成并行代码。这样的编译器对程序员隐藏了它发现程序并行性、把计算分布到多个处理器、极小化处理器之间的同步和通信等具体细节。许多科学计算和工程应用都是计算密集型的，它们可以从并行处理中大大获益。并行化技术已经用于自动地把串行科学计算程序翻译成多处理器代码。

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

内存分层出现在所有的机器上。处理器通常有规模在几百字节的少数几个寄存器，规模在几千字节到几百万字节的几层缓存，规模在几百万字节到几十亿字节的物理存储器，还有规模在几十亿字节甚至更大的二级存储。相对应地，相邻两层之间的访问速度会区别 2 ~ 3 个数量级。一个系统的性能经常不是受处理器速度的限制，而是受内存子系统性能的限制。编译器的优化历来集中在优化处理器的执行上，但是现在更强调要使内存分层更有效。与并行一样，内存分层也能改进机器潜在的性能，但必须有合适的编译器才能把性能改进真正落实到应用上。

在优化一个程序时，寄存器的有效使用是关键问题。和寄存器由软件来显式管理不同，缓存和物理内存对指令集是隐蔽的，它们由硬件来管理。人们已经发现，由硬件实现的缓存管理策略在某些情况下并不怎么有效，尤其在使用大数据结构(最典型的是数组)的科学计算代码中。通过改变数据的布局或改变指令访问数据的次序来提高内存分层的效率是可能的，还可以通过改变代码的布局来改进指令缓存的效率。

1.2.3 新计算机体系结构的设计

在早期的计算机体系结构设计中,编译器的开发是在造出计算机之后进行的。现在,这种情况已经改变,因为计算机系统的性能不仅仅取决于它的原始速度(raw speed),还取决于编译器是否能生成充分利用其特征的代码。因此,在现代计算机体系结构的研究中,在处理器的设计阶段就已着手开发编译器,并将编译生成的代码在模拟器上运行,以评价拟采用体系结构的性能。

编译器技术影响计算机体系结构设计的一个著名例子是精简指令集计算机(RISC)的发明。在此发明之前,当时的倾向是开发日益增大的复杂指令集,以使汇编编程变得容易,这种复杂指令集架构被称为复杂指令集计算机(CISC)。例如,CISC 指令集包含支持数据结构访问的复杂内存寻址模式,还有在栈上保存寄存器和传递参数的过程调用指令。

编译器优化通过删除复杂指令序列中的冗余,可以把这些指令精简到数目不多的较简操作。于是,建立简单指令集是有希望的,编译器能够有效地利用它们,并且硬件也很容易优化它们。

许多通用处理器体系结构,包括 PowerPC、SPARC、MIPS、Alpha 和 PA-RISC,都是基于 RISC 概念的。虽然最流行的微处理器 x86 体系结构有 CISC 指令集,但是研究 RISC 机器的许多想法都落实在实现这种处理器上。而且,使用一台高性能 x86 机器的最有效方式就是只用它的简单指令。

过去 30 年提出了很多体系结构概念,包括数据流机、向量机、超长指令字机器 VLIW、单指令多数据阵列处理器、脉动阵列(systolic array)、带共享内存的多处理器和带分布内存的多处理器。相应编译器技术的研究和开发都伴随着上述每一种体系结构概念的发展。

上述一些概念已经应用到嵌入式机器的设计中。由于完整的系统可以集成在单个芯片上,处理器不再是一个包装好的商品单元,它能够为特定应用而量身定做,以获得更好的成本效益。于是,通用处理器靠经济规模促进计算机体系结构趋于一致,而专用处理器则展现了体系结构的多样性。编译器技术不仅要能支持这些体系结构上的编程,还要能对拟采用体系结构的设计进行评价。

1.2.4 程序翻译

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

一是二进制翻译。编译器技术可用于把一种机器的二进制代码翻译成另一种机器的代码,以运行原先为别的指令集编译的代码。许多计算机公司用这种方式增强它们机器上的软件的可用性。一个典型例子是,由于 x86 在个人计算机市场上占据支配地位,大部分软件产品都是 x86 的代码,因此很早就开发出了用于把 x86 的代码转换成 Alpha 和 SPARC 代码的二进制翻译器。全美达(Transmeta)公司的 Crusoe 处理器是一种 VLIW 处理器,它们不是在硬件上直接执行复杂

的 x86 指令集,而是利用二进制翻译技术把 x86 指令翻译成本地 VLIW 代码。二进制翻译也可用于提供反向的兼容性。1994 年,当 Apple Macintosh 的处理器从 Motorola MC 68040 改为 PowerPC 时,二进制翻译使得 PowerPC 处理器可以运行历史遗留的 MC 68040 代码。

二是数据库查询解释器。数据库查询语言,例如结构化查询语言 SQL,其查询由一些谓词组成,而这些谓词又由包括关系运算在内的布尔表达式组成,它们可以被解释执行,也可以被编译成搜索数据库的命令,以寻找满足这些谓词的记录。

1.2.5 提高软件开发效率的工具

可以说,程序是最复杂的人工制品,它们包含许许多多的细节,要求每个细节在程序真正投入使用后不能出错。测试是定位程序中错误的基本技术。

一种很有前景并且和测试互补的方式是用数据流分析来静态地定位错误。数据流分析能发现任何可能执行路径上的错误,而不仅仅是那些由测试数据集合引起的执行路径上的错误。起源于编译器优化的许多数据流分析技术,可以用来构造一些工具,在软件工程任务中给程序员以帮助。

找出程序中所有错误是一个不可判定问题。数据流分析可以设计成将所有可能是错误的情况都报告给程序员,但是若其中大部分警告都是假的,则程序员将不会使用这样的工具。通常,实际的错误探测器确实既不可靠也不完备,即它们不能保证报告给程序员的都是真正的错误,并且也发现不了程序中所有的错误。尽管如此,各种静态分析工具仍然层出不穷并且能够在实际查找错误时发挥一定的作用,例如查找对空指针或已被释放指针的脱引用(dereferencing)。错误探测器的不可靠使得它们和编译器优化有显著区别,优化器必须是稳妥的,在任何情况下不能改变程序的语义。

源于编译器中代码优化技术的程序分析有助于改进软件开发效率,下面扼要介绍三个方面,其中最重要的是静态发现程序可能有安全漏洞的技术。

(1) 类型检查。类型检查是一种捕捉程序中不一致性的成熟并且有效的技术。例如,它能发现运算对象的类型不满足某个运算对其类型的要求,能发现实在参数和相应形式参数的类型不匹配。通过分析程序的数据流,程序分析能发现除了类型错误以外的更多错误。例如,把一个指针赋值为 null 后紧接着对它进行脱引用操作,显然这是一个程序错误。

类型检查技术可用于捕捉多种安全漏洞,例如程序草率使用的字符串。攻击者会通过提供这样的字符串来发起攻击。程序分析的一种解决办法是,把用户提供的字符串都加类型标记“危险的”。如果程序没有对一个字符串进行格式检查,则它一直维持为“危险的”。如果该字符串会影响某程序点的控制流,则有潜在的安全缺陷。

(2) 边界检查。当程序员用较低级的语言编程时很容易犯错误。例如,系统中很多安全缺口就是由于 C 程序中的缓冲区溢出引起的,因为 C 语言不做数组边界检查,它要求程序员自己保证对数组的访问没有越界。对用户提供的数据不做检查也可能导致缓冲区溢出,进而程序可

能破坏或误用缓冲区以外的用户数据。攻击者就是利用给程序提供这样数据的机会,来致使程序行为不正常并危及系统安全。查找程序缓冲区溢出的很多技术一直在开发,但是都只取得了有限的成功。

删除冗余边界检查的数据流分析技术也可以用来定位缓冲区溢出,它与边界检查的主要区别是,未删除冗余的边界检查仅仅是增加了一点点运行开销,而不能定位潜在缓冲区溢出则可能危及系统的安全。因此,虽然使用简单的技术对优化边界检查是足够的,但是为了获得高质量的错误探测工具,需要使用更复杂的分析技术,例如对指针的值进行跨过程跟踪。

(3) 内存管理。无用单元收集是在易于编程、软件可信赖和执行效率之间的一个折中。自动的内存管理删除内存泄漏等所有内存管理错误,这些错误是C和C++程序中问题的主要根源。目前,有很多帮助程序员发现内存管理错误的工具,例如Purify就是一个广泛使用的工具,它能动态地捕获程序运行时出现的内存管理错误。还有一些可静态标识这些错误的程序分析工具。

习 题

1.1 解释下列名词:

源语言、目标语言、翻译器、编译器、解释器

1.2 典型的编译器可以划分成几个主要的逻辑阶段?各阶段的主要功能是什么?

第2章

词法分析

词法分析器的任务是把构成源程序的字符流翻译成词法记号流。构造词法分析器的一种简单办法是用状态转换图来描述源语言词法记号的结构,然后手工把这种状态转换图翻译成为识别词法记号的程序。用这种方式可以产生高效的词法分析器。

本章重点围绕词法分析器的自动生成展开,先介绍与之有关的正规式和有限自动机概念,以及词法分析器的自动生成方法,最后介绍一个词法分析器自动生成工具 Lex。

2.1 词法记号及属性

词法分析是编译的第一阶段,它的主要任务是扫描输入字符流,产生用于语法分析的词法记号序列。第 1 章曾提到,编译器中一些阶段的活动会交错进行,图 2.1 给出了词法分析器和语法分析器的一种典型关系,即把词法分析器作为语法分析器的一个子程序来实现。当收到来自语法分析器的“取下一个记号”命令时,词法分析器扫描剩余输入字符,直到它能够确认一个词法记号为止。

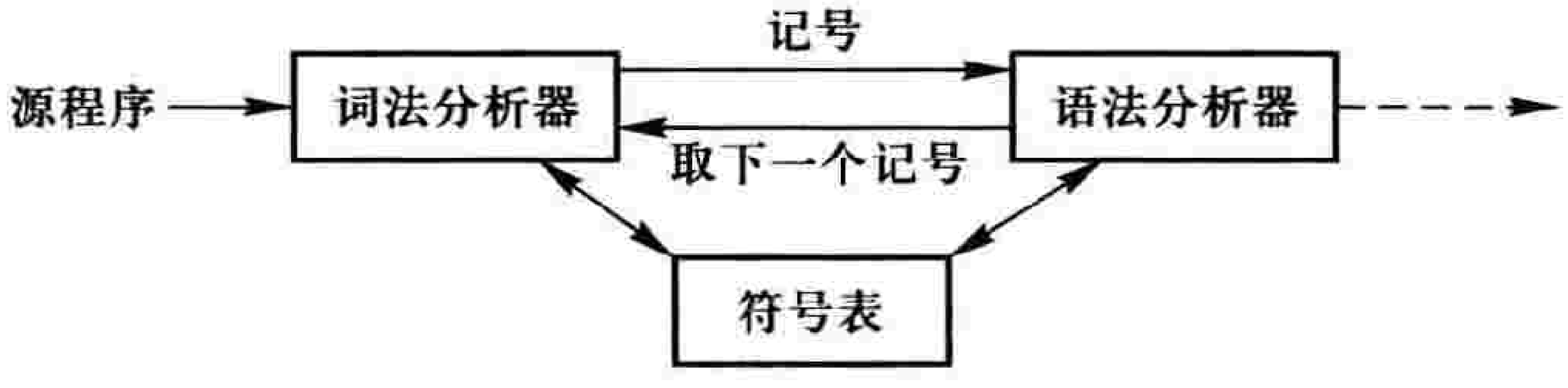


图 2.1　词法分析器和语法分析器的相互作用

词法分析器是编译器中扫描源程序的部分,因而它还可以完成和用户接口的一些其他任务。其一是剥去源程序的注解和(由空格、制表或换行符等引起的)空白。其二是把来自编译器各个阶段的错误信息和源程序联系起来,例如,词法分析器记住当前处理的字符行的行号,从而可以把一个错误信息和发现错误时的那行源程序联系在一起。在某些编译器中,复制源程序并把错误信息嵌在其中也是词法分析器的工作。通常在词法记号流中已经没有行的概念,因此这样的事情一般由词法分析器来完成。如果源语言支持宏定义,那么对它们的预处理也可以在词法分

析时实现。

2.1.1 词法记号、模式、词法单元

在谈论词法分析时，使用三个相关但有区别的术语。

(1) 词法记号。词法记号(简称记号)是由记号名和属性值构成的二元组，属性值不是必需项。记号名是代表一类词法单元的抽象名字，例如标识符、某个特定的关键字。记号名是语法分析的输入符号，下文中记号名均以加粗形式表示，并且经常直接用记号名来引用记号。

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

(3) 词法单元。词法单元(lexeme)，又称单词，是源程序中匹配一个记号模式的字符序列，它由词法分析器识别为该记号的一个实例。

表2.1给出了一些典型的记号名、实例及其非形式描述模式。例如，在C语句

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

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

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

表2.1 记号的例子

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

某些语言的一些规定给词法分析带来了困难。例如，在对待空格上，不同语言有不同规定。在早先的一些语言(如FORTRAN和ALGOL 68)中，空格无意义(文字串中的除外)，而不是作为词法单元的分隔符，它可以随便加入，以改变程序的可读性。空格的这种约定增加了处理标识符记号的复杂性，典型例子是FORTRAN的DO语句，在语句

```
DO 8 I=3.75
```

中，词法分析器只有看见了小数点后，才能确定DO在这里不是关键字，而DO8I是一个标识符。但是在表面上类似的语句

DO 8 I=3,75

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

另一个例子是,关键字是否保留。保留字是语言预先确定了含义的词法单元,程序员不可以对这样的词法单元重新声明它的含义,如 Pascal 语言的 var 和 begin 等是保留字。

很多语言使用关键字概念,并且关键字是保留的,因此它和上面的保留字概念没有区别,如 C 语言和 Java 语言。但是 FORTRAN 语言的关键字不保留,如 IF,当它作为语句的第一个词法单元时,很可能是关键字,因为这是该关键字可能出现的地方,但也不排除它是一个程序显式声明的标识符。若 IF 出现在语句的其他地方,它一定是程序显式或隐式声明的标识符。这就给词法分析带来很大困难,因为识别一个记号和该记号所处的上下文有关了。

顺便需要区分有些语言使用的标准标识符概念,标准标识符也是预先确定了含义的标识符,但程序可以重新声明它的含义。在该声明的作用域内,程序声明的含义起作用,而预先确定的含义消失,在其他地方则都是预先确定的含义起作用,如 Pascal 语言的 integer 和 true 等。词法分析器对标准标识符没有什么特别的处理,由符号表管理来解决这件事。

2.1.2 词法记号的属性

从上一小节知道,C 语言中的 6 个关系算符都属于记号 **relation**。因为从程序的语法是否正确的角度看,使用哪一个关系算符都一样。但是从翻译成目标代码来考虑,不同的关系算符,其翻译结果不一样。因此词法分析器需要给记号以属性,用属性来记住记号的附加信息,以便需要时使用它们。概括地说,记号名影响语法分析的决策,属性影响记号的翻译。

例 2.1 语句

position = initial+rate * 60

的记号用二元组序列表示如下:

〈**id**,指向符号表中 position 条目的指针〉
〈**assign_op**〉
〈**id**,指向符号表中 initial 条目的指针〉
〈**add_op**〉
〈**id**,指向符号表中 rate 条目的指针〉
〈**mul_op**〉
〈**number**,整数值 60〉

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

2.1.3 词法错误

词法分析难以发现源程序的错误,因为词法分析器对源程序采取非常局部的观点。在C语言的语句

fi (a==f(x))…

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

有些语言要求实型常量的小数点后面必须有数字,如果程序中出现小数点后面没有数字情况,词法分析器会报错。词法分析器也就只能发现这样的错误了。

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

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

(1) 删除一个多余的字符;

(2) 插入一个遗漏的字符;

(3) 用一个正确的字符代替一个不正确的字符;

(4) 交换两个相邻的字符。

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

2.2 词法记号的描述与识别

上一节提到,字符串集合由称为模式的规则来描述。正规式是表示这些规则的一种重要方法,因此本节围绕正规式来介绍记号的描述与识别。在介绍正规式前,先给“语言”一个形式定义。

2.2.1 串和语言

术语**字母表**表示符号的有限集合,符号的典型例子有英文字母和标点符号。集合{0, 1}是二进制字母表;ASCII是字母表的一个重要例子,它用于很多软件系统中;Unicode是另一个重要例子,它包含了取自全世界各个字母表的大约十万个字符。

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

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

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

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

有一些重要的运算可以作用于语言。对词法分析而言,感兴趣的运算是和、连接、闭包,它们定义在表2.2中。幂运算也可用于语言 L,定义 L^0 是 $\{\varepsilon\}$,L^i 是 $L^{i-1}L$,即 L^i 是 L 连接它自己 $i-1$ 次。

表 2.2 语言运算的定义

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

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

(1) $L \cup D$ 是字母和数字的集合。

(2) LD 是所有由一个字母后跟随一个数字组成的串的集合。

(3) L^6 是所有由6个字母组成的串的集合。

(4) L^* 是所有字母串(包括 ε)的集合。

(5) $L(L \cup D)^*$ 是以字母开头的所有字母数字串的集合。

(6) D^+ 是不含空串的数字串的集合。 □

从这个例子可以看出,从基本集合开始,利用语言上的运算可以定义新的语言。下面将用更有利于计算机处理的形式来定义一类简单的语言。

2.2.2 正规式

正规式(又称正规表达式、正则表达式)是按照一组定义规则,由较简单的正规式构成的,每个正规式 r 表示一个语言 $L(r)$。这些定义规则说明 $L(r)$ 是怎样从 r 的子正规式所表示的语言中构造出来的。

下面是定义字母表 Σ 上正规式的规则,和每条规则相联系的是被定义的正规式所表示的语言的描述。

(1) ε 是正规式,它表示 $\{\varepsilon\}$。

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

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

正规式表示的语言叫做**正规语言**或**正规集**。

如果约定:

(1) 闭包运算(算符是 *)有最高的优先级并且是左结合的运算,

(2) 连接运算(两个正规式并列)的优先级次之且也是左结合的运算,

(3) 选择运算(算符是 $|$)的优先级最低且仍然是左结合的运算,

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

例 2.3 令字母表 $\Sigma=\{a, b\}$,那么:

(1) 正规式 $a|b$ 表示集合 $\{a, b\}$。

(2) 正规式 $(a|b)(a|b)$ 表示 $\{aa, ab, ba, bb\}$,即由 a 和 b 构成的所有长度为2的串集。表示同样集合的另一个正规式是 $aa|ab|ba|bb$。

(3) 正规式 a^* 表示仅由字母 a 构成的所有串的集合,包括空串。

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

如果两个正规式 r 和 s 表示同样的语言,就说 r 和 s 等价,写作 $r=s$。例如,$(a|b)=(b|a)$。

正规式遵守一些代数定律,它们可用于正规式的等价变换,表 2.3 列出了正规式 r、s 和 t 遵守的部分代数定律。

表 2.3 正规式的代数定律

定律	描述
$r\|s=s\|r$	| 是可交换的
$r\|(s\|t)=(r\|s)\|t$	| 是可结合的

续表

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

2.2.3 正规定义

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

如果 Σ 是基本符号的字母表,那么**正规定义**是形式为

$d_1 \to r_1$

$d_2 \to r_2$

...

$d_n \to r_n$

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

为了区别名字和符号,本书在正规定义中用黑体字表示名字。

例 2.4 C 语言的标识符是由字母、数字和下划线组成的串,下面是 C 语言标识符的正规定义。

letter_ → *A* | *B* | ⋯ | *Z* | *a* | *b* | ⋯ | *z* | _

digit → 0 | 1 | ⋯ | 9

id → **letter_**(**letter_** | **digit**)* □

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

digit → 0 | 1 | ⋯ | 9

digits → **digit digit***

optional_fraction → . **digits** | ε

optional_exponent → (E (+ | - | ε) **digits**) | ε

number → **digits optional_fraction optional_exponent**

从这个定义可以知道,无符号数由整数部分、小数部分和指数部分三部分组成,其中小数部分和指数部分都是可能出现或可能不出现的。如果出现指数部分,是E及可能有的+或-号,再跟上一个或多个数字。注意小数点后面至少有一个数字,所以 **number** 能匹配2.0,但不能匹配2.。 □

在正规式中,某些结构频繁出现,为方便起见,用缩写表示它们。

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

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

number → **digit**$^+$(. **digit**$^+$)? (E (+ | -)? **digit**$^+$)?

来描述无符号数。

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

letter_ → [A-Za-z_]

digit → 0 | 1 | ⋯ | 9

id → **letter_**(**letter_**|**digit**)*

描述标识符。

2.2.4 状态转换图

本节以下面正规定义描述的语言为例,讨论怎样识别记号。

例 2.6 考虑下面的正规定义:

while → while

do → do

relop → < | <= | = | <> | > | >=

letter → [A-Za-z]

id → **letter** (**letter** | **digit**)*

number → **digit**$^+$(. **digit**$^+$)? (E (+ | -)? **digit**$^+$)?

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

delim → **blank** | **tab** | **newline**

ws → **delim**$^+$

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

记号,然后返回到分析器。

词法分析器输出的是〈记号名,属性值〉二元组序列,表2.4给出了各正规式描述的记号的这种二元组。注意,正规式 **ws** 没有对应的记号,另外,关系算符的属性值由符号常量 LT、LE、EQ、NE、GT 和 GE 给出。 □

表 2.4 正规式、记号名和属性

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

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

图2.2是识别记号 **relop** 的转换图,可以用这张图来解释有关转换图的概念。转换图上的圆圈表示**状态**,状态由有向**边**连接,边上有指示输入字符的标记,标记通常是一个字符。若离开状态 s 的某个边上有标记 **other**,则它表示离开 s 的其他边所指示的字符以外的任意字符。这一节讨论的转换图是确定的,即不可能出现某一字符和离开一个状态的两条边上的标记都匹配的情况。这个条件在后面会放宽。

在状态转换图中,有一个状态标记为开始状态,这是转换图的初始状态。当开始识别记号时,控制进入开始状态。当控制进入某个状态时,读输入串的下一个字符,如果离开这个状态的一条边上的标记和该输入字符匹配,控制就进入由这条边指向的状态,否则识别过程失败。某些状态有两个圆圈,表示它们是接受状态,控制进入这样的状态表明识别了一个记号。接受状态可以有动作,控制到达接受状态时执行它的动作。

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

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

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

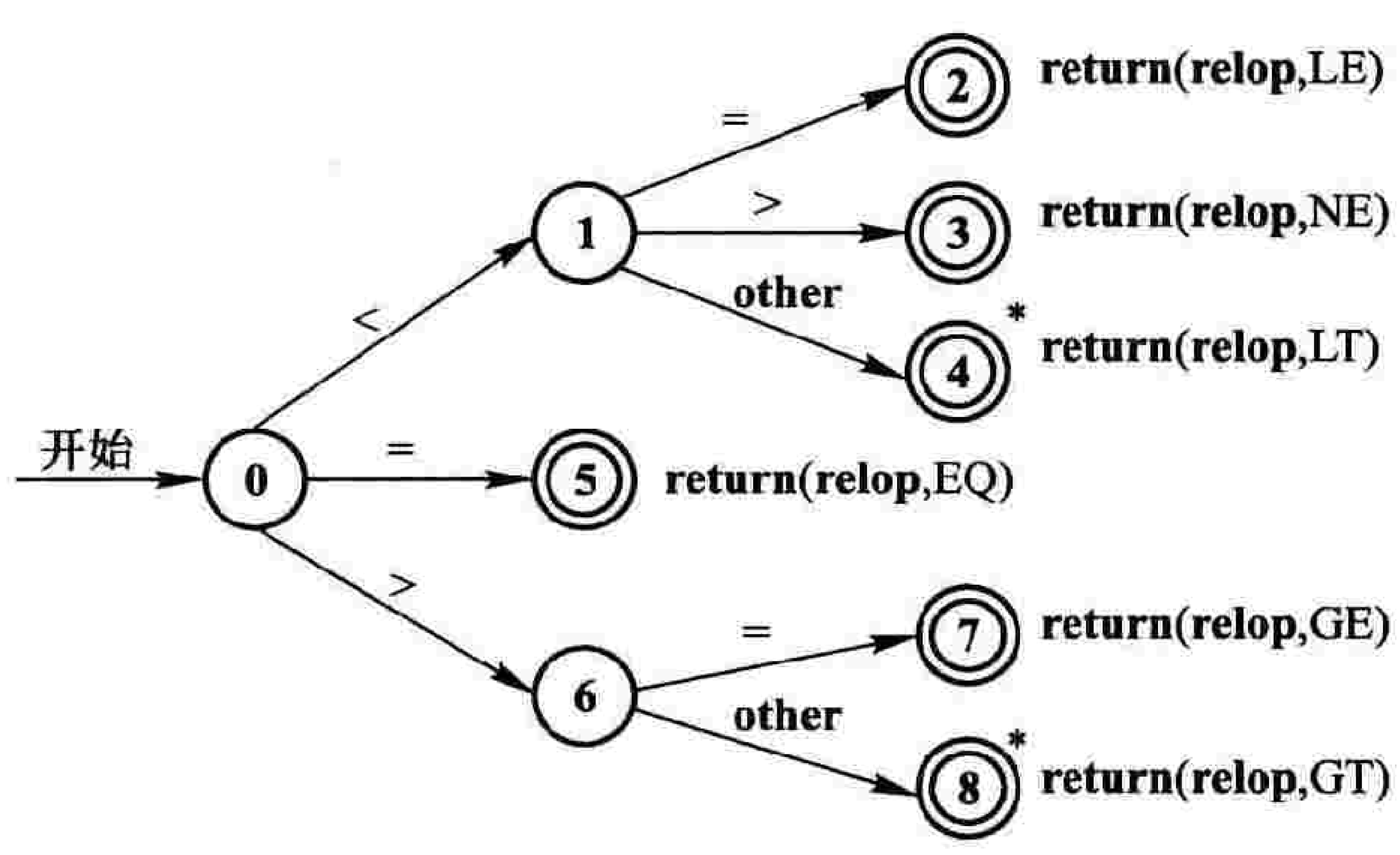


图 2.2　关系算符的转换图

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

如果要识别的关键字有所变化，无须修改转换图，只需给关键字表重新置初值即可。为关键字单独构造转换图是可能的。对典型的编程语言来说，这么做会使词法分析器的状态数多达几百个。而用上面的方法，不到 100 个状态可能就够了。 □

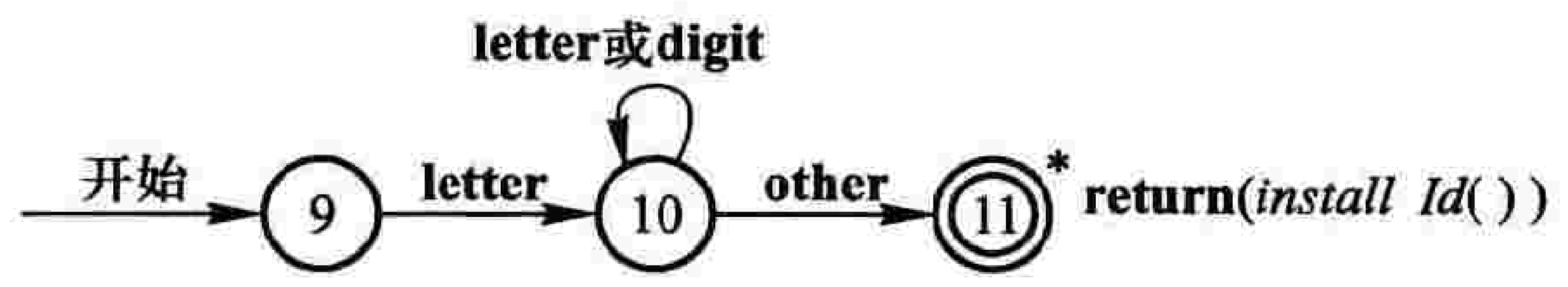


图 2.3　标识符和关键字的转换图

例 2.8　下面根据正规定义

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

为无符号数构造识别器。注意，这个定义中，小数部分(. $\textbf{digit}^{+}$)和指数部分($\mathrm{E}\,(+\mid -)?\,\textbf{digit}^{+}$)是可选的。图 2.4 是它的状态转换图。

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

把图 2.2、图 2.3 和图 2.4 的三个开始状态 0、9 和 12 合并成一个开始状态，就可以把这三个转换图合并成一个转换图。

剩下的问题是空白。代表空白的 **ws** 的处理和上面讨论的代表各种记号的正规式的处理有

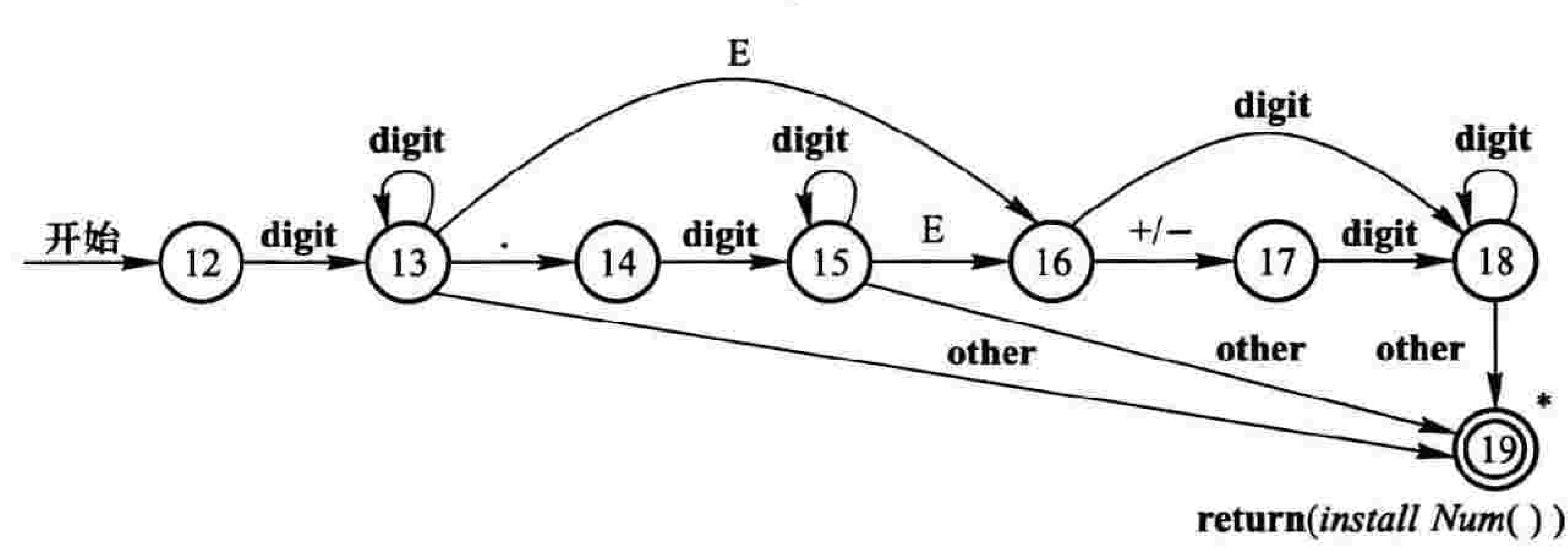


图 2.4　无符号数转换图

所不同,因为在输入串中发现空白时,并没有任何东西返回给语法分析器。识别 **ws** 的转换图如图 2.5 所示。把这个转换图和其他几个转换图合并成一个转换图后,到达接受状态 22 的动作就是回到开始状态,识别下一个记号。

根据正规式构造出转换图后,基于这样直观的转换图,编写一个程序来识别这些正规式描述的记号就不是件困难的事情了。当然,直接编写词法分析器并非一定要基于转换图概念,本节介绍转换图是为了便于大家接受有限自动机的概念。

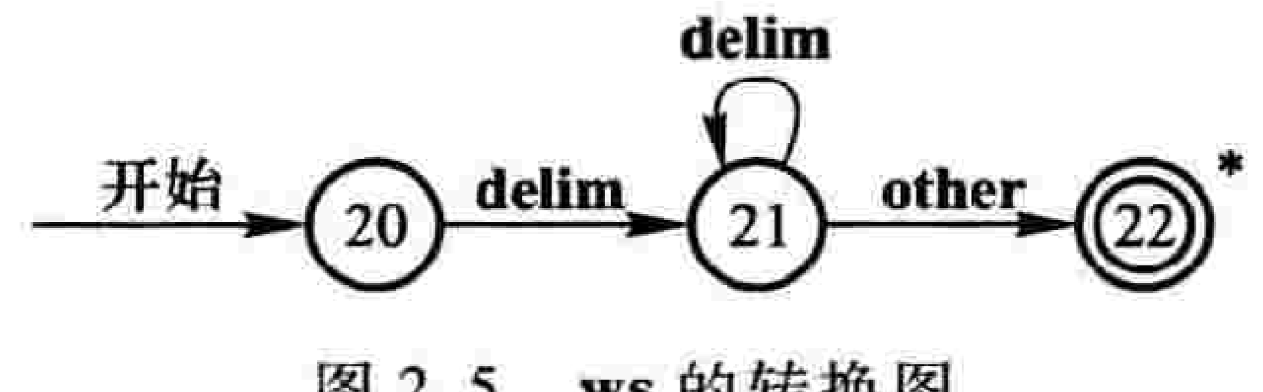


图 2.5　**ws** 的转换图

2.3　有限自动机

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

有限自动机分成确定的和不确定的两种情况。"不确定"的含义是,存在这样的状态,对于某个输入符号,它存在不止一种转换。

确定的和不确定的有限自动机都正好能识别正规集,也就是它们能识别的语言正好是正规式所能表达的语言。但是,它们之间存在着时空权衡问题:从确定的有限自动机得到识别器,比从等价的不确定的有限自动机得到识别器要快得多;但是,确定的有限自动机可能比等价的不确定有限自动机占用更多的空间。由于把正规式变成不确定的自动机更直接一些,因此下面首先讨论这类自动机。

本节和下一节的基本例子是正规式 $(a|b)^*ab$ 表示的语言。类似的语言在实际应用中也有,例如,表示所有以.o结尾的文件名的正规式是(. | o | c)*. o 的形式,其中 c 代表除 . 和 o 以外

的任何字符。另一个例子是,C 语言的注释是以/ * 开始和以 * /结束的任意字符串,但它的任何前缀(本身除外)不以 * /结尾。

2.3.1 不确定的有限自动机

不确定的有限自动机(Non-deterministic Finite Automata,NFA)是一个数学模型,它包括:

(1) 一个有限的状态集合 S;

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

(3) 一个转换函数 $move: S \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(S)$ [$P(S)$ 指 S 的幂集],它把状态和符号(可以是 ε)两元组映射到一个状态集合;

(4) 状态 s_0 是唯一的开始状态;

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

NFA 可以用带标记的有向图表示,即状态转换图,结点表示状态,有标记的边代表转换函数。这种转换图和上一节所讲的略有区别,在这里,可以把同样的符号标记在出自同一个状态的多条边上。另外,边可以由输入符号标记,也可以由特殊符号 ε 标记。

可以识别语言 $(a|b)^*ab$ 的 NFA 的转换图如图 2.6 所示。这个 NFA 的状态集合是 $\{0, 1, 2\}$,输入符号表是 $\{a, b\}$,状态 0 是开始状态,接受状态 2 用双圈表示。

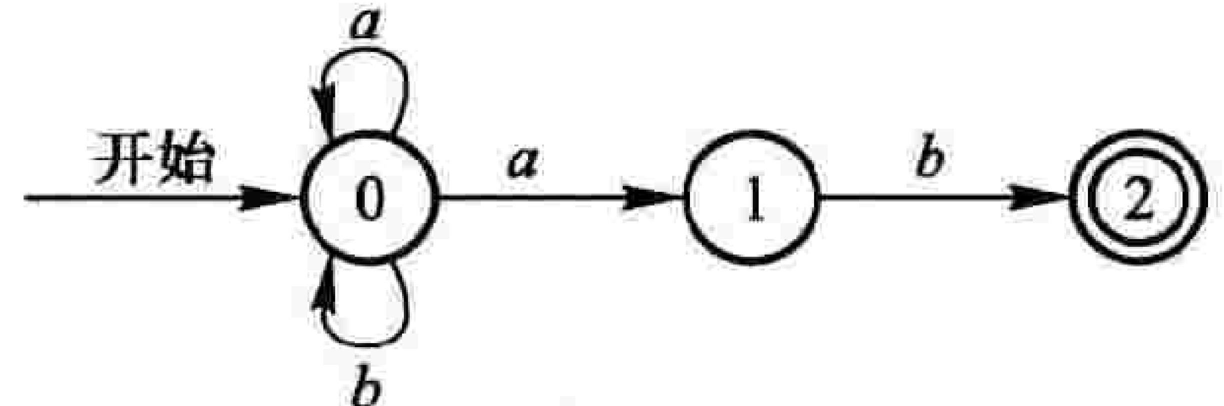


图 2.6 识别 $(a|b)^*ab$ 的 NFA

NFA 可以用这种转换图来描述,在计算机上它可以用不同的方法实现。最简单的办法是**转换表**,每个状态一行,每个输入符号和 ε(如果需要的话)各占一列,表的第 i 行中对应符号 a 的条目是一个状态集合(说得更实际一些,是状态集合的指针),表示 NFA 在输入是 a 时,状态 i 所能到达的状态集合。表 2.5 是对应图 2.6 的 NFA 的转换表。

表 2.5 图 2.6 的 NFA 的转换表

状态	输入符号	
	a	b
0	{0,1}	{0}
1	∅	{2}
2	∅	∅

转换表的优点是可以快速访问给定状态和字符的状态集。它的缺点是,当输入字母表较大,并且大多数转换是空集时,占用了大量空间。显然,很容易把有限自动机的一种实现转变成另一种实现。

NFA 接受输入串 x,当且仅当转换图中存在从开始状态到某个接受状态的路径,该路径各边

上的标记可拼成 x。图 2.6 的 NFA 接受输入串 $ab, aab, bab, aaab, \cdots\cdots$ 例如,从状态 0 开始,沿着标记为 a 的边再到状态 0,然后沿着标记分别为 a 和 b 的边先后到达状态 1 和 2 构成的路径,接受 aab。对一个输入,可能有多条路径可以到达接受状态。

由 NFA 定义的语言是它接受的输入串集合,不难看出图 2.6 的 NFA 可识别(也称接受)$(a|b)^*ab$。

例 2.9 图 2.7 是识别 $aa^*|bb^*$ 的 NFA,串 aaa 由通过 0,1,2,2 和 2 的路径来接受,相应边上的标记分别是 ε, a, a 和 a,它们拼成 aaa,ε 在拼接中"消失"。 □

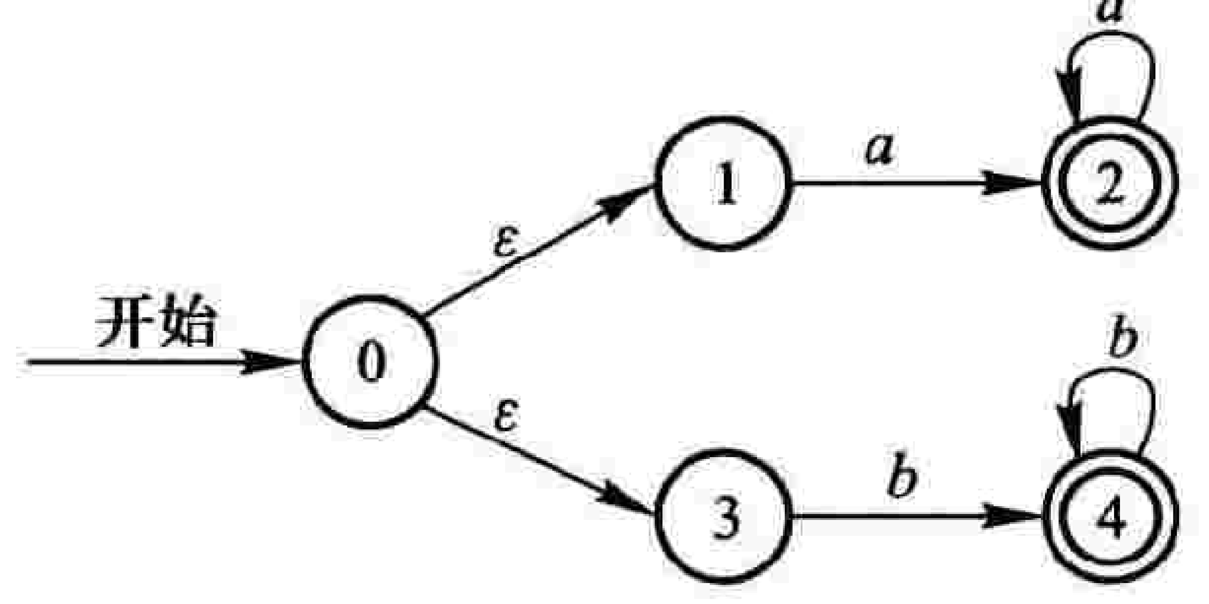


图 2.7 识别 $aa^*|bb^*$ 的 NFA

2.3.2 确定的有限自动机

确定的有限自动机(Deterministic Finite Automata,DFA)是不确定的有限自动机的特殊情况。其中:

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

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

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

算法 2.1 模拟 DFA。

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

输出 如果 D 接受 x,则回答"yes",否则回答"no"。

方法 把图 2.8 的算法施加于输入串 x。函数 $move(s, c)$ 给出一个状态,它是面临输入符号 c、状态 s 的转换。函数 $nextChar(\ )$ 返回输入串 x 中的下一个字符。 □

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

图 2.8 模拟 DFA

例 2.10 如图 2.9 所示的转换图表示一个 DFA，它和图 2.6 所示的 NFA 识别同样的语言 $(a|b)^*ab$。用这个 DFA 和输入串 *abab* 作为算法 2.1 的输入，根据算法方法可知状态沿着 0，1，2，1 和 2 移动，并返回"yes"。 □

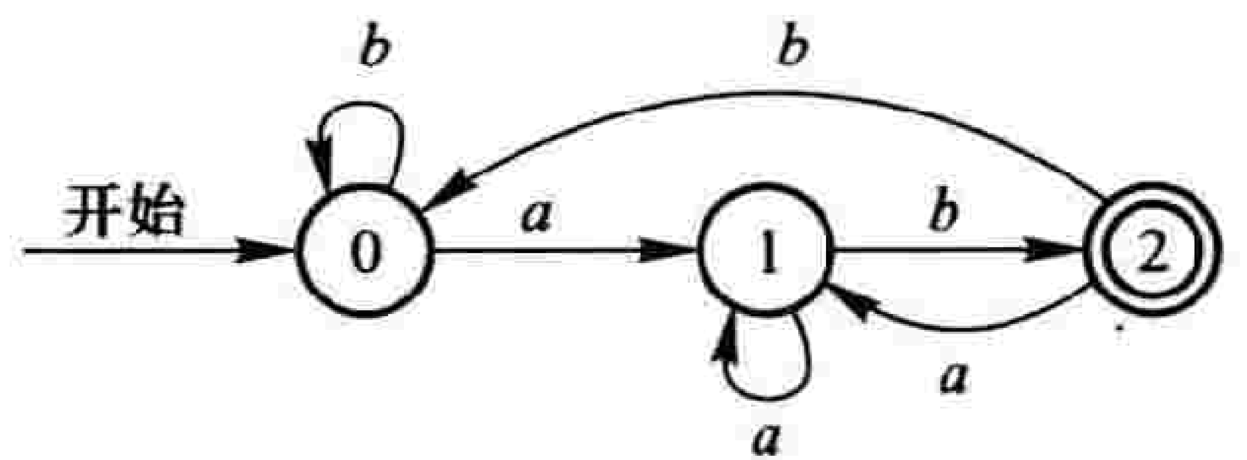


图 2.9 识别 $(a|b)^*ab$ 的 DFA

2.3.3 NFA 到 DFA 的变换

图 2.6 的 NFA 在状态 0 下、输入为 *a* 时有两个转换，它可以进入状态 0 或 1。类似地，图 2.7 的 NFA 在状态 0 下，不接受任何输入（即面对 ε）也有两个转换。有些场合还会出现既可以根据 ε 也可以根据一个实际输入符号进行转换的情况，这同样也会引起二义。这种转换函数多值的情况，使得很难用计算机程序模拟 NFA。"接受"的定义是说必须存在从开始状态到某个接受状态的一条路径，该路径的标记正好拼成输入串。这样，在找到一条接受路径或确定没有这样的路径前，可能不得不同时考虑所有路径。

下面给出一个算法，它从 NFA 构造出可识别同样语言的 DFA，这个算法通常称为**子集构造法**。一个和它密切相关的算法是下一章构造 LR 分析器的基础。

首先概述子集构造法的思想。在 NFA 的转换表里，每个条目是一个状态集；在 DFA 的转换表中，每个条目只有一个状态。从 NFA 构造等价 DFA 的一般思想是让新构造的 DFA 的每个状态对应到该 NFA 的一个状态集，即这个 DFA 在读取输入 $a_1 a_2 \cdots a_n$ 后到达的状态，对应于该 NFA 从开始状态沿着那些标有 $a_1 a_2 \cdots a_n$ 的路径能到达的所有状态的集合。显然，这个 DFA 的状态数和该 NFA 的状态数是成指数变化的，但实际上，这种最坏情况很少发生。

算法 2.2 从 NFA 到 DFA 的子集构造法。

输入 一个 NFA *N*。

输出 一个接受同样语言的 DFA *D*。

方法 为 *D* 构造转换表 *Dtran*，表中的每个状态是 *N* 的状态集合，*D*"并行"地模拟 *N* 面对输入串的所有可能的移动。

表 2.6 定义的三个函数描述了在 *N* 状态上的基本运算，它们用在图 2.10 的算法中。注意 *s* 代表 *N* 的单个状态，*T* 代表 *N* 的一个状态集合。

表 2.6 在 NFA 状态上的运算

运算	描述
ε-*closure*(*s*)	从 NFA 的状态 *s* 出发，只用 ε 转换能到达的 NFA 状态集合

续表

运算	描述
ε-closure(T)	*NFA* 的状态集合 $\cup_{t \in T}$ε-closure(t)
move(T, a)	*NFA* 的状态集合 $\cup_{t \in T}$ move(t, a)(move 被拓展成多态函数)

在读第一个输入符号前,N 可以处于集合 ε-*closure*(s_0)的任何状态,其中 s_0 是 N 的开始状态。假定集合 T 是从 s_0 出发,面临某个输入串所能到达的所有状态的集合,令 a 是下一个输入符号,那么看见 a 时,N 可以移动到集合 *move*(T, a)中的任何一个状态。由于允许 ε 转换,看见 a 后,N 可以处于 ε-*closure*(*move*(T, a))中的任何一个状态。

按图 2.10 的算法就是按子集构造法思想来构造 D 的状态集合 *Dstates* 和转换表 *Dtran*。D 的开始状态是 ε-*closure*(s_0)。如果 D 的某个状态所对应的状态集至少含 N 的一个接受状态,那么它是 D 的一个接受状态。

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

图 2.10　子集构造法

ε-*closure*(T)的计算是从给定的结点集合出发,在状态转换图上搜索可达结点的典型过程。该图只包含 NFA 的含 ε 标记的边,T 是给定的结点集合。计算 ε-*closure*(T)的简单算法是用栈来保存那些边还没有完成 ε 转换检查的状态。图 2.11 描述了这样的过程。 □

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

图 2.11　ε-*closure*(T)的计算

例2.11 图2.12是接受语言$(a|b)^*ab$的另一个NFA N,之所以用它做例子,是因为下一节"机械地构造NFA"的算法也是以此为例。对N执行算法2.2,其等价的DFA的开始状态是$\varepsilon\text{-}closure(0)$,记为$A=\{0, 1, 2, 4, 7\}$,因为它们正好是从状态0出发,经过标记都是$\varepsilon$的路径所能到达的所有状态。由于路径可以没有边,所以0也是经这样的路径从0能到达的状态。

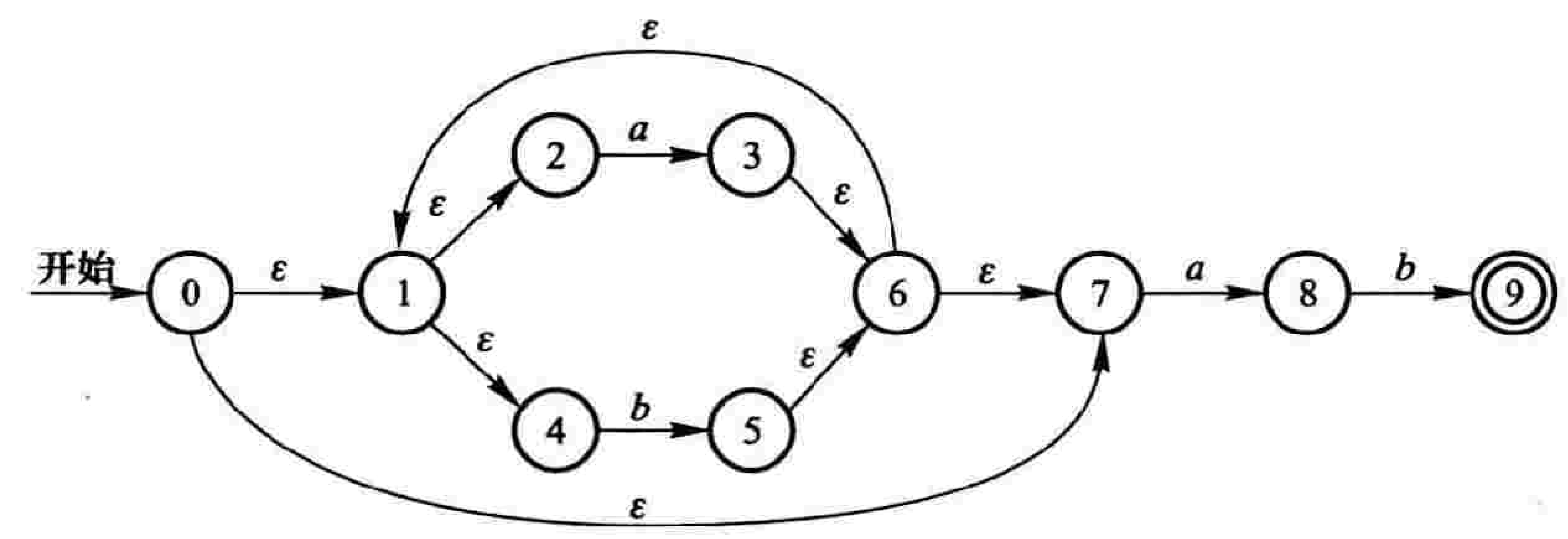


图2.12 识别$(a|b)^*ab$的NFA

这里的输入字母表是$\{a, b\}$。根据图2.10的算法,首先标记A,然后计算

$$\varepsilon\text{-}closure(move(A, a))$$

由于在$A=\{0, 1, 2, 4, 7\}$中,只有状态2和7能发生a转换,分别转变为状态3和8,因此$move(A, a) = \{3, 8\}$。

所以

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

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

在A中,只有状态4发生b转换到达状态5,所以状态A的b转换到达

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

令该集合为C,于是,$Dtran[A, b] = C$。

用尚未标记的新集合B和C继续这个过程,最终会达到这样一点:所有的集合(即DFA的所有状态)都已标记。因为10个状态的集合的不同子集只有2^{10}个,一个集合一旦标记就永远是标记的,所以计算终止是肯定的。本例实际构造出的4个不同状态集合是:

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

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

表 2.7　DFA 的转换表 *Dtran*

状态	输入符号	
	a	b
A	B	C
B	B	D
C	B	C
D	B	C

这个 DFA 的转换图见图 2.13。必须注意，图 2.9 的 DFA 也接受 $(a|b)^*ab$，并且少一个状态。下一小节将讨论 DFA 的化简问题。□

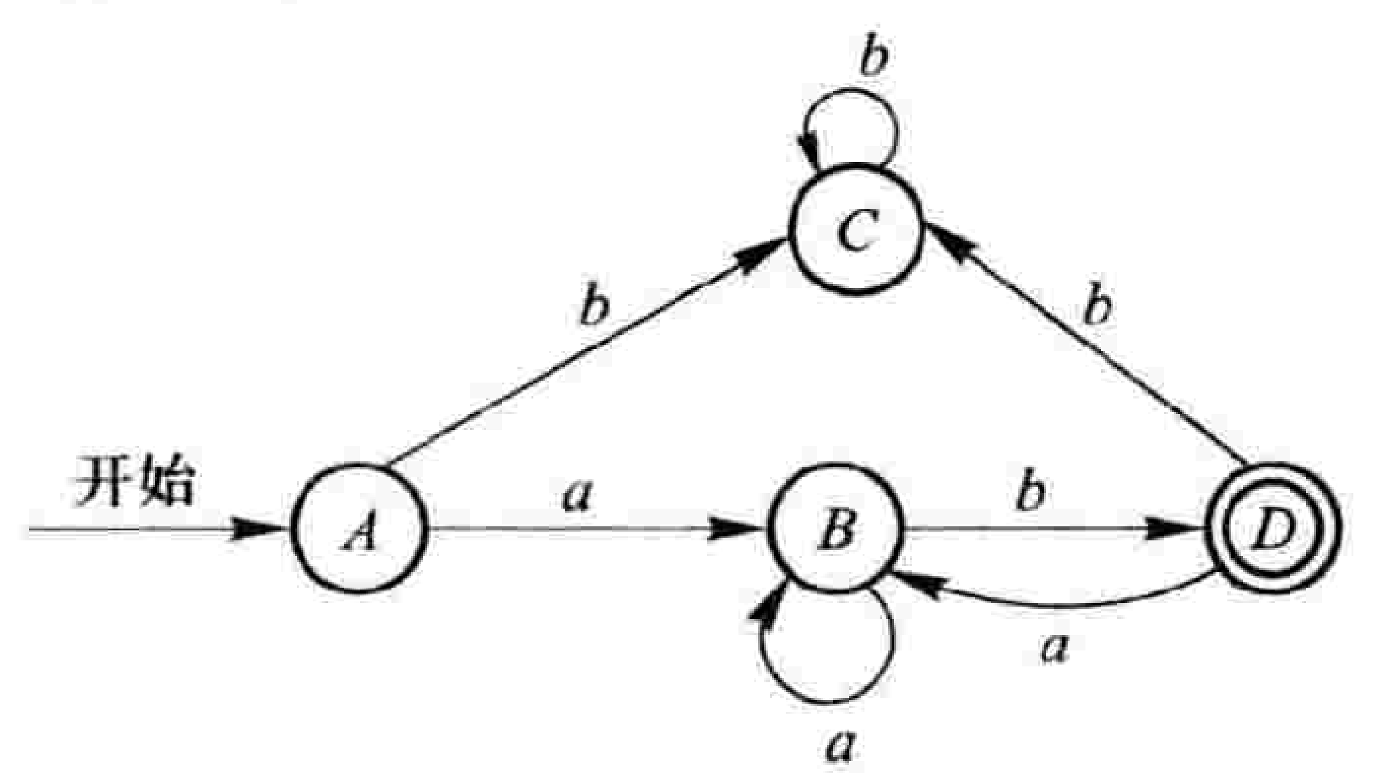


图 2.13　子集构造法用于图 2.12 的结果

2.3.4　DFA 的化简

理论上的一个重要结论是，每一个正规集都可以由一个状态数最少的 DFA 识别，这个 DFA 是唯一的（因状态名不同的同构情况除外）。本小节介绍如何把一个 DFA 化简到状态数最少，并且识别同样的语言。

这里所用方法基于转换函数是全函数。如果一个 DFA 的转换函数不是全函数，可以引入一个“死状态”s_d，s_d对所有输入符号都转换到 s_d本身。如果状态 s 对符号 a 没有转换，那么加上从 s 到 s_d的 a 转换。显然，加入死状态后的 DFA 和原来的 DFA 等价。由此可知，DFA 也可以基于 *move* 函数是全函数来定义。

对于 DFA M，如果从状态 s 出发，在面临输入串 w 时，最终停在一个接受状态，而从状态 t 出发，面临 w 时，它停在一个非接受状态，或者反过来，则串 w 可用来**区别**状态 s 和 t。如果找不到任何串来区别 s 和 t，那么 s 和 t 是不可区别的。例如，任何接受状态和非接受状态可用空串 ε 来区别。在图 2.13 中，A 和 B 可由输入 b 来区别，因为对输入 b，A 走到非接受状态 C，而 B 走到接受状态 D。

极小化 DFA 状态数的算法就是把它的状态分成一些不相交的子集，每一子集的状态都是不可区别的，不同子集的状态都是可区别的。每个子集合并成一个状态。

最初，这个划分包括两个子集，即接受状态子集和非接受状态子集，因为它们可用空串来区别。然后，检查每一个子集，看其中的状态是否还可区别。例如，对状态子集 $A = \{s_1, s_2, \cdots, s_k\}$ 和输入符号 a，检查 $s_1, s_2, \cdots, s_k$ 面临 a 的转换，如果这些转换所到的状态落入当前划分的两个或更多的状态子集中，那么 A 必须进一步划分，使得 A 每个子集的 a 转换能落入当前划分的一个状态子集中。例如，若 s_1 和 s_2 的 a 转换分别到达 t_1 和 t_2，并且 t_1 和 t_2 在当前划分的不同子集中，那么 A 至少要分成两个子集，一个含 s_1，另一个含 s_2。注意，如果 t_1 和 t_2 是可由某个串 w 区别的，那么 s_1 和 s_2 一定是可由串 aw 区别的。

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

算法 2.3 极小化 DFA 的状态数。

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

输出 一个 DFA M'，它和 M 接受同样的语言，且状态数最少。

方法 (1) 构造状态集合的初始划分 Π：分成两个子集，接受状态子集 F 和非接受状态子集 $S-F$。

(2) 应用下面的过程对 Π 构造新的划分 Π_{new}：

for (Π 中的每个子集 G) {
　　把 G 划分成若干子集，G 的两个状态 s 和 t 在同一子集中，当且仅当对任意输入符号 a，s 和 t 的 a 转换是到 Π 的同一子集中
　　在 Π_{new} 中，用 G 的划分代替 G
}

(3) 如果 $\Pi_{new} = \Pi$，则让 $\Pi_{final} = \Pi$，再执行步骤(4)，否则，令 $\Pi = \Pi_{new}$，转(2)。

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

(5) 如果 M' 有死状态，则去掉它。从开始状态不可达的状态也要删除。从任何其他状态到死状态的转换都改成无定义。 □

再次提请注意，使用这个算法时，其输入 DFA 的状态转换函数必须是全函数，否则有可能导

致得到的新 DFA 和原来的 DFA 接受不同语言。前面介绍的增加死状态,其目的就是把转换函数变成全函数。

例 2.12 重新考虑图 2.13 代表的 DFA。初始划分 Π 包括两个子集:接受状态子集{D}和非接受状态子集{A, B, C}。为了构造 Π_{new},首先考虑{D},因为这个子集只包含一个状态,它不能再划分,所以在 Π_{new} 中仍是{D}。然后考虑{A, B, C},对于输入 a,这些状态都转换到 B,但对于输入 b,A 和 C 都转换到状态子集{A, B, C}的一个成员,而 B 转换到 D,是另一个子集的成员。于是,在 Π_{new} 中,状态子集{A, B, C}必须分成两个新子集{A, C}和{B},Π_{new} 成了{A, C}、{B}和{D}。

再次扫描,只有{A, C}有划分的可能。但是对于输入 a 和 b,它们都分别转换到 B 和 C,因而不必再划分。即这一遍扫描后,$\Pi_{new} = \Pi$。所以,Π_{final} 是{A, C}、{B}和{D}。

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

表 2.8 最简 DFA 的转换表

状态	输入符号	
	a	b
A	B	A
B	B	D
D	B	A

例如,在这个最简自动机中,D 的 b 转换到 A,因为在原来的自动机中,D 的 b 转换到 C,并且 A 是 C 所在子集的代表。类似的变化也发生在状态 A 面临输入 b 时。其余的都是从图 2.13 中复制过来。该图没有死状态,并且所有的状态都是从开始状态 A 可到达的。 □

2.4 从正规式到有限自动机

有很多办法可以从正规式建立它所定义语言的识别器,每种办法都有它的长处和短处。本书介绍的是从正规式构造 NFA,然后用上节的子集构造法把 NFA 变成 DFA,并把它化简。本节将给出从正规式构造 NFA 的一个算法。

该算法是语法制导的,它用正规式语法结构来指导构造过程。首先构造识别 ε 和字母表中一个符号的自动机,然后构造识别主算符为选择、连接或闭包的正规式的自动机。例如,对于正规式 $r \mid s$,从 r 和 s 的 NFA 中归纳构造出它的 NFA。

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

算法 2.4 从正规式构造 NFA。

输入 字母表 Σ 上的正规式 r。

输出 接受 $L(r)$ 的 NFA N。

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

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

(1) 对于 ε，构造如图 2.14 所示的 NFA，其中 i 是开始状态，f 是接受状态。很明显，这个 NFA 识别 $\{\varepsilon\}$。

(2) 对 Σ 中的每个符号 a，构造如图 2.15 所示的 NFA。同样，i 是开始状态，f 是接受状态。这个 NFA 识别 $\{a\}$。

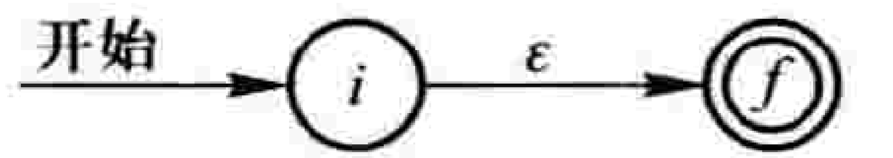


图 2.14 识别正规式 ε 的 NFA

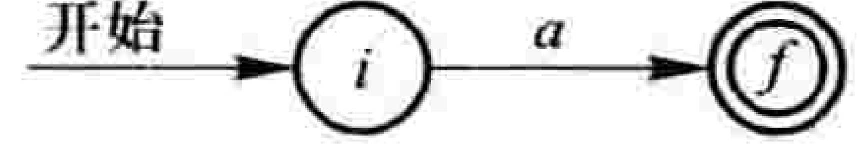


图 2.15 识别正规式 a 的 NFA

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

① 对于正规式 $s \mid t$，构造合成的 NFA $N(s \mid t)$，结果如图 2.16 所示。这里 i 是新的开始状态，f 是新的接受状态。从 i 到 $N(s)$ 和 $N(t)$ 的开始状态有 ε 转换，从 $N(s)$ 和 $N(t)$ 的接受状态到 f 也有 ε 转换。$N(s)$ 和 $N(t)$ 的开始和接受状态不是 $N(s \mid t)$ 的开始和接受状态。这样，从 i 到 f 的任何路径必须排他地通过 $N(s)$ 或 $N(t)$。这个合成的 NFA 识别 $L(s) \cup L(t)$。

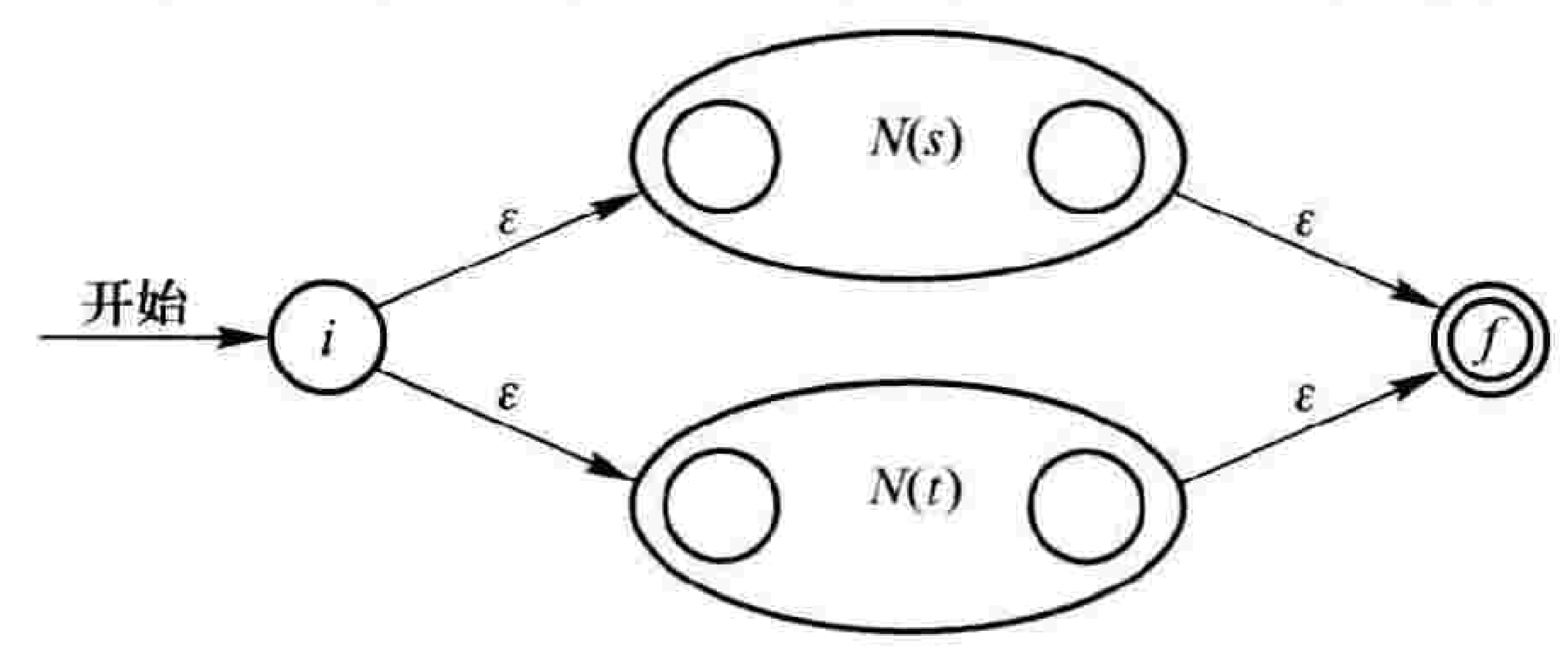


图 2.16 识别正规式 $s|t$ 的 NFA

② 对于正规式 st，构造合成的 NFA $N(st)$，结果如图 2.17 所示。$N(s)$ 的开始状态成为合成后的 NFA 的开始状态，$N(t)$ 的接受状态成为合成后的 NFA 的接受状态，$N(s)$ 的接受状态和 $N(t)$ 的开始状态合并，也就是 $N(t)$ 开始状态的所有转换成为 $N(s)$ 的接受状态的转换。合并后的这个状态不作为合成后的 NFA 的接受状态或开始状态。从 i 到 f 的路径必须首先经过 $N(s)$，然后经过 $N(t)$，所以这种路径上的标记拼成 $L(s)L(t)$ 的串。因为没有边进入 $N(t)$ 的开始状态

或离开 $N(s)$ 的接受状态，所以在 i 到 f 的路径中不存在从 $N(t)$ 回到 $N(s)$ 的现象，故合成的 NFA 识别 $L(s)L(t)$。

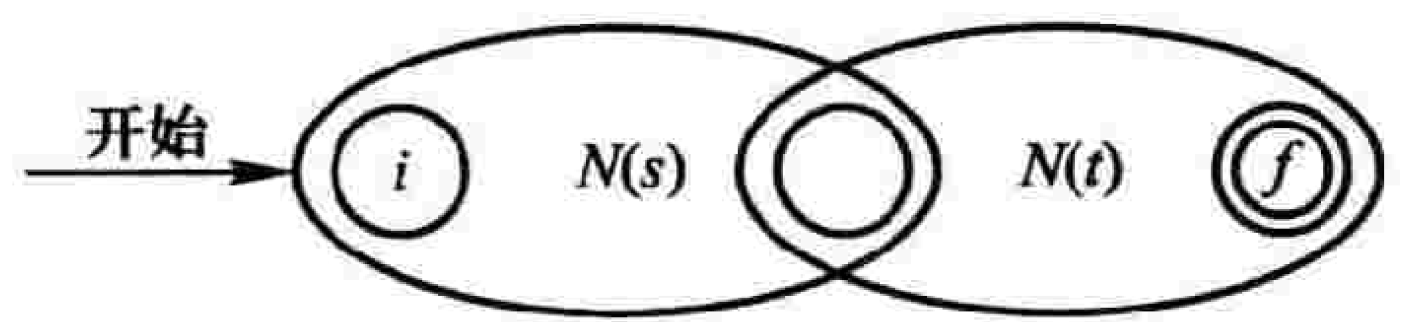


图 2.17 识别正规式 st 的 NFA

③ 对于正规式 s^*，构造合成的 NFA $N(s^*)$，结果如图 2.18 所示。同样，i 和 f 分别是新的开始状态和接受状态。在这个合成的 NFA 中，可以沿着 ε 边直接从 i 到 f，这代表 ε 属于 $(L(s))^*$，也可以从 i 经过 $N(s)$ 一次或多次。显然，这个 NFA 识别 $(L(s))^*$。

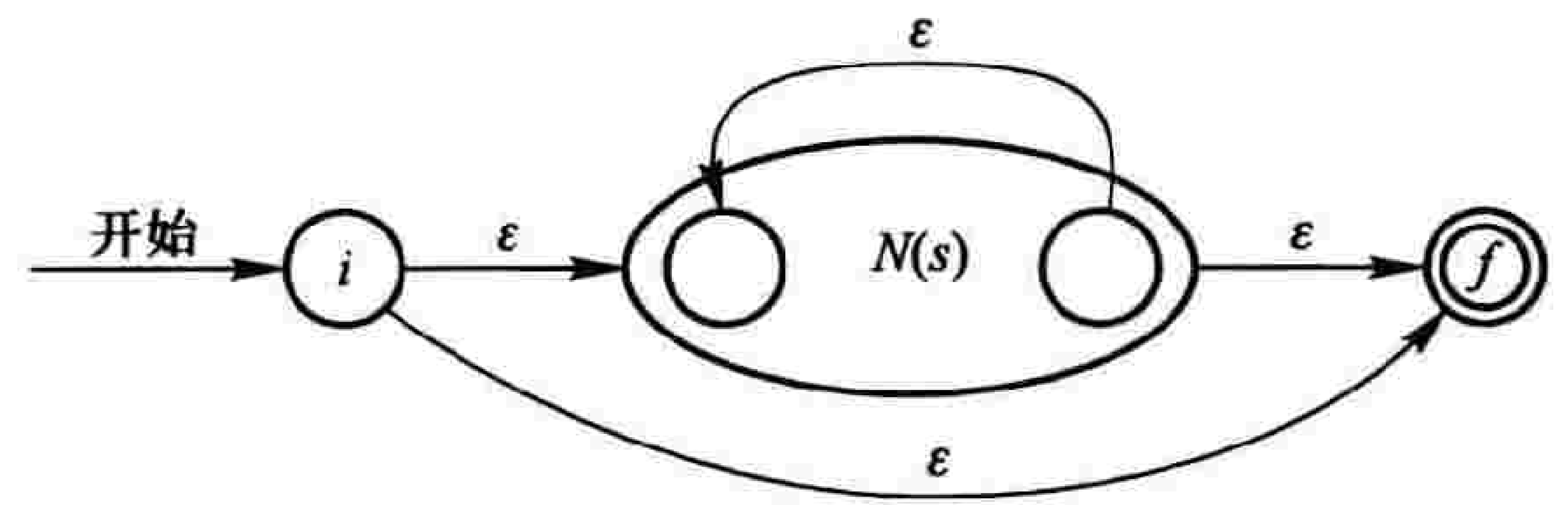


图 2.18 识别正规式 s^* 的 NFA

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

对于每次构造的新状态都赋予不同的名字。这样，所有的状态都有不同的名字。 □

可以检验，算法 2.4 的每一步都构造产生识别对应语言的 NFA。此外，产生的 NFA 有下列性质。

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

(2) $N(r)$ 只有一个接受状态，接受状态没有向外的转换。

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

例 2.13 用算法 2.4 构造正规式 $r = (a|b)^*ab$ 的 NFA $N(r)$。图 2.19 是 r 的分析树。对于其中的 r_1 和 r_2，构造它们的 NFA，再用选择规则组合 $N(r_1)$ 和 $N(r_2)$，得到 $r_3 = r_1 | r_2$ 的 NFA。(r_3) 的 NFA 和 r_3 的一样，再构造 $(r_3)^*$ 的 NFA。这样依次下去，最后得到 $r = (a|b)^*ab$ 的 NFA 如图 2.20 所示，它和图 2.12 一致。 □

从手工构造 NFA 的角度看，算法 2.4 的缺点是引入了大量的 ε 转换，使得后面手工将 NFA 转换为 DFA 时，容易因疏忽而出错。因此在手工构造 NFA 时，应避免引入 ε 转换，例如图 2.6 识别 $(a|b)^*ab$ 的 NFA 就比图 2.20 简单得多。

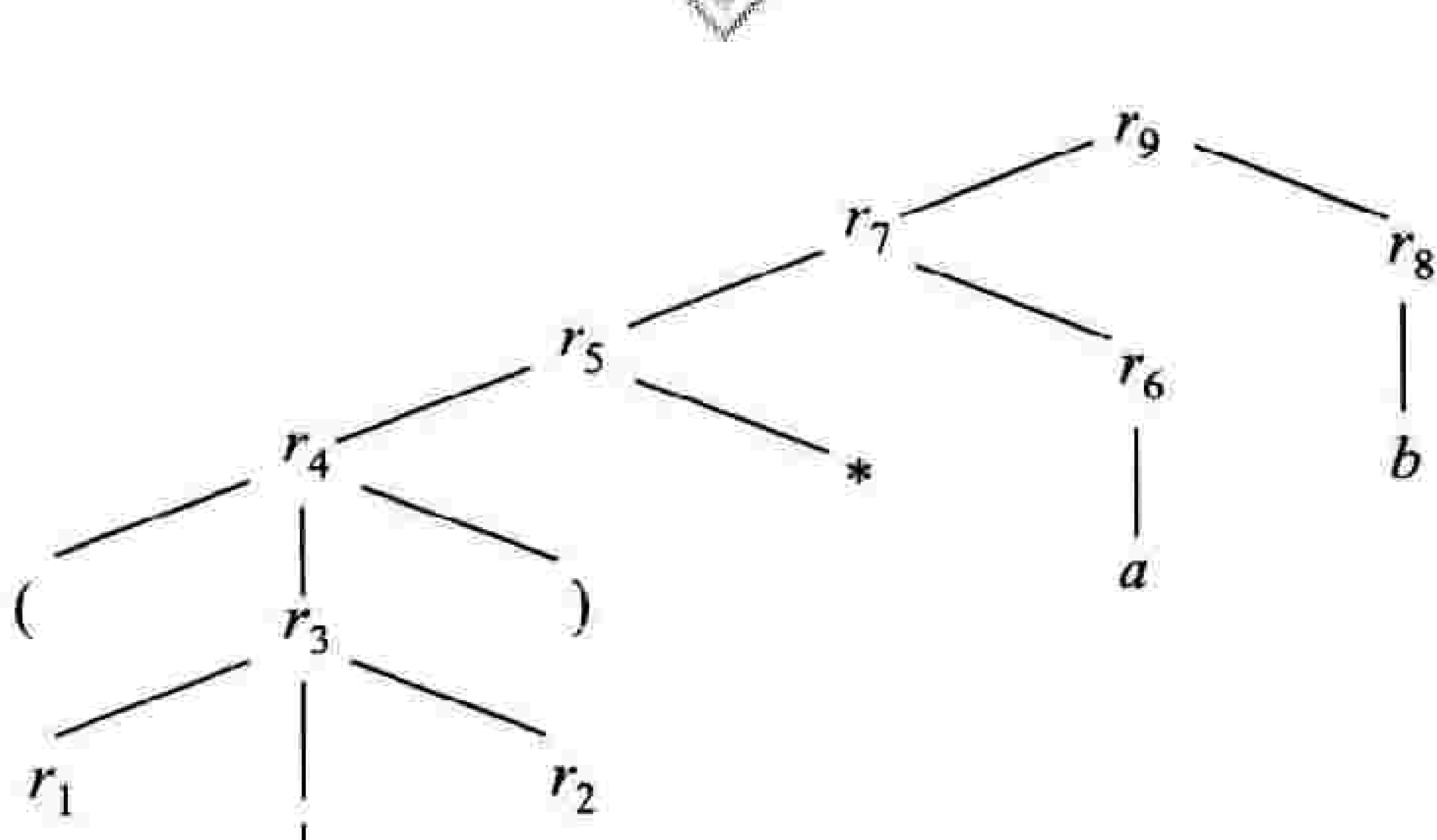


图 2.19 $(a|b)^*ab$ 的分解

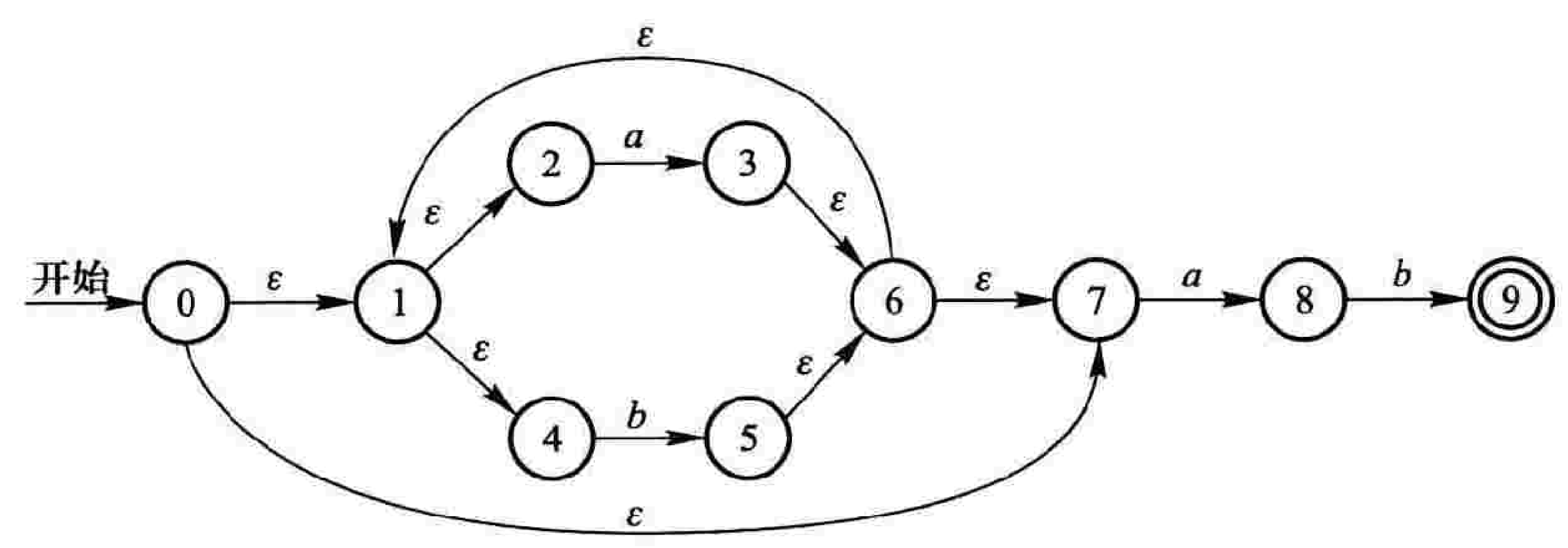


图 2.20 识别$(a|b)^*ab$ 的 NFA

2.5 词法分析器的生成器

本节描述一个特殊的工具——Lex,它依据基于正规式的描述来构造词法分析器,并且已广泛用于描述各种语言的词法分析器。这个工具也称为 Lex 编译器,它的输入是用 Lex 语言编写的。通过讨论这个工具可以明白,要求词法分析器完成的动作(例如,在符号表中增加新条目)是怎样和基于正规式的模式说明组织在一起的,从而形成词法分析器的规范。即使没有可用的 Lex 编译器,这样的规范也还是有用的,因为可以根据 2.2 节介绍的转换图技术手工地构造出词法分析器。

Lex 通常按图 2.21 描绘的方式使用。首先,词法分析器的说明用 Lex 语言表达在程序 lex.l 中,然后 lex.l 通过 Lex 编译器,产生 C 语言程序 lex.yy.c。程序 lex.yy.c 包括从 lex.l 的正规式构造出的转换图(用表格形式表示)和使用这张转换图识别词法单元的标准子程序。在 lex.l 中,和正规式相关联的动作是用 C 语言代码表示的,它们被直接复制到 lex.yy.c 中。最后,lex.yy.c 被编译成目标程序 a.out,它就是把输入串识别成记号序列的词法分析器。

Lex 程序包括三个部分:

图 2.21 用 Lex 建立词法分析器

声明
%%
翻译规则
%%
辅助函数

声明部分包括变量声明、常量定义和正规定义。这里的正规定义和 2.2 节介绍的方式类似，可以作为正规式出现在翻译规则中。

每条翻译规则的形式如下：

模式　　　　　{动作}

其中，每个模式是一个正规式，每个动作描述该模式匹配词法单元时，词法分析器应执行的程序段。在 Lex 中，动作是用 C 语言写的代码段，尽管已经出现了很多使用其他语言的 Lex 变种。

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

由 Lex 建立的词法分析器通常作为语法分析器的一个子程序。词法分析器每次被语法分析器调用时，逐个字符地读它的剩余输入，直到它在剩余输入中发现能和某个模式 P 匹配的最长前缀为止。然后执行和 P 对应的动作 A。典型地，动作 A 将把控制返回语法分析器。如果不是这样，例如 P 描述空白或注释，则词法分析器就继续寻找后继的词法单元，直到有一个动作引导控制回到语法分析器为止。

词法分析器仅返回一个值（记号名）给语法分析器，记号的属性值通过共享整型变量 yylval 传递。

例 2.14 图 2.22 是识别表 2.4 中记号的 Lex 程序，此程序体现出 Lex 的一些重要特征。

声明部分定义了一些在翻译规则中使用的常量，这些声明由特别的括号%{和%}包围。出现在括号中的任何东西都被直接复制到词法分析器 lex.yy.c 中，不作为正规定义和翻译规则的一部分。第三部分中的辅助函数也按同样方式处理。图 2.22 有两个函数 installId() 和 installNum()，它们被复制到 lex.yy.c 中。

声明部分还包括一些正规定义，每个定义由一个名字和该名字指示的正规式组成。例如，第

一个定义的名字是 delim，它代表字符类[\t\n]，也就是空格、制表符(\t)和换行符(\n)这三个字符中的任意一个。第二个是空白定义，由名字 ws 表示，空白是一个或多个分界符的序列。

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

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

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

图 2.22 识别表 2.4 记号的 Lex 程序

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

原来的含义。数的正规定义中，十进制小数点由\.表示，因为 Lex 及许多处理正规式的 UNIX 程序中，点可以代表除了换行符以外的任何一个字符。在字符类[+\-]中，减号的前面放了反斜线，因为减号也是元符号，用于表示范围，如[A-Z]。

还有别的方法可让作为元符号的字符表示原来的含义，就是把它们放在引号中。在翻译规则部分有这样的例子，6 个关系算符都由引号包围。

在翻译规则中，第一条规则表明，如果看见 ws，也就是空格、制表符和换行符的串，则没有任何动作发生，控制也不返回分析器。词法分析器将继续去识别记号，直至所识别记号的动作引起返回为止。注意，在 Lex 中，ws 必须由花括号包围，以区别由两个字母 ws 组成的模式，下面的 id 和 number 也是这样。

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

在 id 的规则中，有关的动作含两个语句。第一个语句调用函数 installId，该函数定义在第三部分，该函数的返回值赋给变量 yylval。变量 yylval 的定义出现在 Lex 的输出 lex.yy.c 中，它对分析器也是可用的，它的作用是保存返回记号的属性值，因为动作的第二个语句 return(ID)只能返回记号名。

图 2.22 代码中略去了 installId 的详细代码，可以猜想，它在符号表中查找由模式 id 匹配的词法单元。注意，它和 2.2 节介绍的 installId 在功能上有区别，因为在这里关键字是用另外的正规式定义的。通过两个变量 yytext 和 yyleng，Lex 使得词法单元的拼写对出现在第三部分的函数是可用的，变量 yytext 是指针，它指向词法单元的开始字符，变量 yyleng 是整数，它指出词法单元的长度。

再下一条规则以类似方式处理数。最后 6 条规则都返回记号 RELOP，而用 yylval 的值来区别不同的关系算符。

如果下一个要匹配的词法单元是 while，那么模式 while 和{id}都匹配这个词法单元，而且这时它们都不能匹配更长的串。这时应该选哪一个呢？Lex 对这个冲突的解决方法是选择排在前面的模式。由于图 2.22 中关键字 while 的模式先于标识符的模式，因此发生冲突时选择排在前面的关键字。

如果要读的前两个字符是<= ，这时模式<匹配第一个字符，但它不是匹配输入中最长前缀的模式。由于 Lex 的策略是选择匹配某模式的最长前缀，这使解决<和<=的冲突变得容易，并且是按所期望的方式选择<=作为下一个记号。 □

习 题 2

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

(a) C

(b) C++

(c) C#

(d) Java

(e) SQL

2.2 在下面的C函数中,按序列出所有的记号,并给每个记号以合理的属性值。

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

2.3 叙述由下列正规式描述的语言。

(a) $0\ (0 \mid 1)^*\ 0$

(b) $(\ (\varepsilon \mid 0)\ 1^*)^*$

(c) $(0 \mid 1)^* 0\ (0 \mid 1)\ (0 \mid 1)$

(d) $0^* 10^* 10^* 10^*$

(e) $(00 \mid 11)^* (\ (01 \mid 10)\ (00 \mid 11)^* (01 \mid 10)\ (00 \mid 11)^*\ )^*$

*2.4 为下列语言写出正规定义。

(a) 包含5个元音的所有字母串,其中每个元音只出现一次且按顺序排列。

(b) 按词典序排列的所有字母串。

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

(d) 相邻数字都不相同的所有数字串。

(e) 最多只有一处相邻数字相同的所有数字串。

(f) 由偶数个0和偶数个1构成的所有0和1的串。

(g) 由偶数个0和奇数个1构成的所有0和1的串。

(h) 所有不含子串011的0和1的串。

(i) 字母表$\{a, b\}$上,a不会相邻出现的所有串。

2.5 说明习题2.1中各种语言的数值常数的词法形式。

2.6 说明习题2.1中各种语言的标识符的词法形式。

2.7 用算法2.4为下列正规式构造不确定有限自动机,给出它们处理输入串$ababbab$的状态转换序列。

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

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

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

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

2.8 用算法2.2把习题2.7的NFA变换成DFA。给出它们处理输入串 *ababbab* 的状态转换序列。

2.9 根据表2.4中记号的转换图构造DFA。

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

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

2.12 为下列正规式构造最简的DFA。

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

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

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

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

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

2.15 构造一个最简的DFA,它接受所有大于101的二进制整数。

2.16 修改算法2.4,使之尽可能少用 ε 转换,并保持所产生的NFA只有一个接受状态。

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

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

其中,x^R 表示 x 的逆。

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

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

误写成

```
/* then part
```

那么该编译器发现不了遗漏分号的错误。这是为什么?

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

第3章

语法分析

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

编译器常用的语法分析方法有自上而下和自下而上两种。正如它们的名字所示,自上而下分析器按从根结点到叶结点的次序来建立分析树,而自下而上分析器恰好相反。它们的共同点是从左向右地扫描输入,每次一个符号。

最有效的自上而下和自下而上的分析法都只能处理上下文无关文法的子类。这些子类足以描述编程语言的大多数构造和它们的语法特征,其中LL文法的分析器通常用手工实现,而LR文法的分析器通常利用自动工具构造。

本章阐述编译器采用的典型语法分析方法。首先提出有关上下文无关文法的基本概念,然后介绍适合于手工实现的预测分析技术,最后给出自动工具采用的LR分析算法。由于程序员准备的代码经常会出现一些语法错误,因此本章还扩展所介绍的分析方法,使之能从常见的错误中恢复过来。

3.1 上下文无关文法

本节首先说明语法分析器(syntax analyzer)在编译器模型中的位置,然后介绍上下文无关文法。语法分析器简称分析器(parser)。如图3.1所示,分析器读取词法分析器提供的记号流,检查它是否能由源语言的文法产生,输出分析树的某种表示。另外,总是希望分析器能以易理解的形式报告语法错误,并从发现语法错误的状态中恢复过来,使之能继续对剩余程序进行分析。

事实上,还有一些其他任务可以在分析的同时完成,例如把各种记号的信息存入符号表中,完成类型检查和其他语义检查,并产生中间代码。所有这些都包罗在图3.1的"前端的其余部分"一框中,在下面几章将详细讨论它们。图3.1中的虚线表示分析树是概念上的东西,并不一定要真正生成。

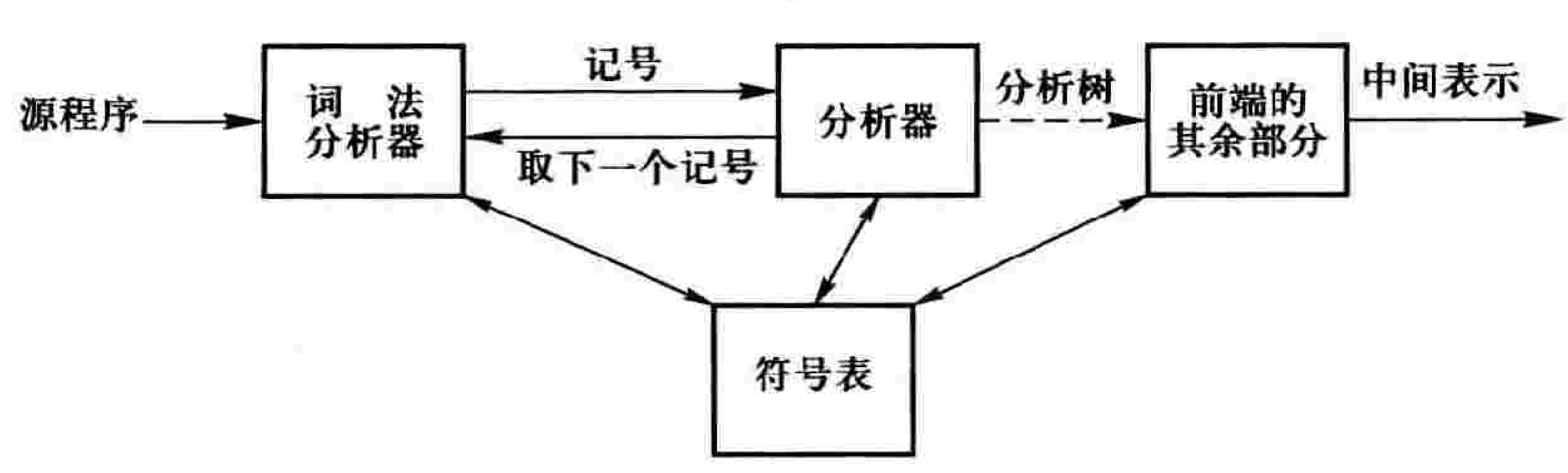


图 3.1　分析器在编译器模型中的位置

3.1.1　上下文无关文法的定义

第 2 章用正规式来定义一些简单的语言,但是很多较复杂的语言不能用正规式表达。例如,正规式不能描述配对或嵌套的结构,具体的例子有,由配对括号构成的串的集合不能用正规式描述,语句的嵌套结构也不能用正规式描述。还有,很多重复串也不能用正规式表示,例如集合

$\{wcw \mid w$ 是 a 和 b 的串$\}$

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

本节定义描述功能比正规式更强的上下文无关文法,并介绍一些与分析有关的术语。

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

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

(2) V_N是一个非空有限集合,其元素称为**非终结符**,并有 $V_T \cap V_N = \varnothing$。在下面的例 3.1 中,*expr* 和 *op* 是非终结符。非终结符用来帮助定义由文法决定的语言,一个非终结符定义终结符串的一个集合。非终结符还在语言中强加了层次结构,这种层次结构对语法分析和翻译都是有用的。

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

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

例 3.1　文法($\{$**id**, +, *, -, (,)$\}$, $\{expr, op\}$, $expr$, P)定义了包含加、乘和一元减的算术表达式。P 由下列产生式组成:

$expr \to expr\ op\ expr$　　　　$expr \to \mathbf{id}$

$expr \to (expr)$　　　　$op \to +$

$expr \to -\ expr$　　　　$op \to *$　　□

为了表示上的简洁,本书将采用下列约定来表示文法。

(1) 下列符号是终结符:

① 字母表前面的小写字母,如 a,b,c;

② 黑体串,如 **id** 或 **while**;

③ 数字 0, 1, …, 9;

④ 标点符号,如分号、逗号等;

⑤ 运算符号,如+、-等。

(2) 下列符号是非终结符:

① 字母表前面的大写字母,如 A,B,C;

② 字母 S,并且它通常代表开始符号;

③ 小写字母组成的名字,如 *expr* 和 *stmt*。

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

(4) 字母表后面的小写字母,主要是 $u, v, \cdots, z$,代表终结符号串。

(5) 小写希腊字母,例如 α,β 和 γ,代表文法的符号串。

(6) 如果 $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \cdots, A \rightarrow \alpha_k$是所有以 A 为左部的产生式(称它们为 A 产生式),则可以把它们写成 $A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$的形式,称这些产生式的右部 $\alpha_1,\alpha_2,\cdots,\alpha_k$是 A 的**选择**。

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

例 3.2 使用这些简写,例 3.1 的文法可以重新表示如下:

$$E \rightarrow EAE \mid (E) \mid -E \mid \mathbf{id}$$

$$A \rightarrow + \mid *$$

其中,E 和 A 都是非终结符,E 是开始符号,其余符号都是终结符。 □

3.1.2 推导

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

$$E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid \mathbf{id} \qquad (3.1)$$

产生式 $E \rightarrow E + E$ 意味着两个表达式相加仍然是表达式。这个产生式允许用 $E + E$ 代替 E 的任何出现,从简单的表达式产生更复杂一些的表达式。如果用 $E + E$ 代替单个 E,这个动作可以用式子

$$E \Rightarrow E + E$$

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

从开始符号 E 开始,不断使用产生式,可以得到一个代换序列,如

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

这个代换序列被称为从 E 到 **id** + **id** 的**推导**,这个推导表明了串 **id** + **id** 是表达式的实例。

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

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

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

类似地,用$\Rightarrow^+$表示“一步或多步推导”。

对于开始符号为 S 的文法 G,可以用$\Rightarrow^+$关系来定义 G 产生的语言 $L(G)$, $L(G)$的串仅包含 G 的终结符。终结符串 w 在 $L(G)$中,当且仅当 $S \Rightarrow^+ w$。若串 w 是 $L(G)$的句子,则也可以称它为文法 G 的句子。由上下文无关文法产生的语言叫做**上下文无关语言**。如果两个文法产生同样的语言,则称这两个文法**等价**。

如果 $S \Rightarrow^* \alpha$,α 可能含非终结符,则把 α 称为 G 的**句型**。句子是只含终结符的句型。

例 3.3 串-(**id** + **id**)是文法(3.1)的句子,因为存在着推导

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

出现在这个推导中的 E, $-E$, $-(E)$, …, -(**id** + **id**)都称为这个文法的句型。

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

如果在推导过程中出现的句型有两个或多个非终结符,那么就需要决定下一步推导代换哪个非终结符。例如,例 3.3 的推导在得到-(E + E)后,可以如下进行:

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

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

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

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

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

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

3.1.3 分析树

分析树是推导的图形表示。分析树上的每个分支结点都由非终结符标记,它的子结点由该非终结符本次推导所用产生式的右部的各符号从左到右依次来标记。分析树的叶结点由非终结符或终结符标记,所有这些标记从左到右构成一个句型。例如,表达式-(**id**+**id**)最左推导的分析

树(包括推导过程中的分析树)如图 3.2 所示。

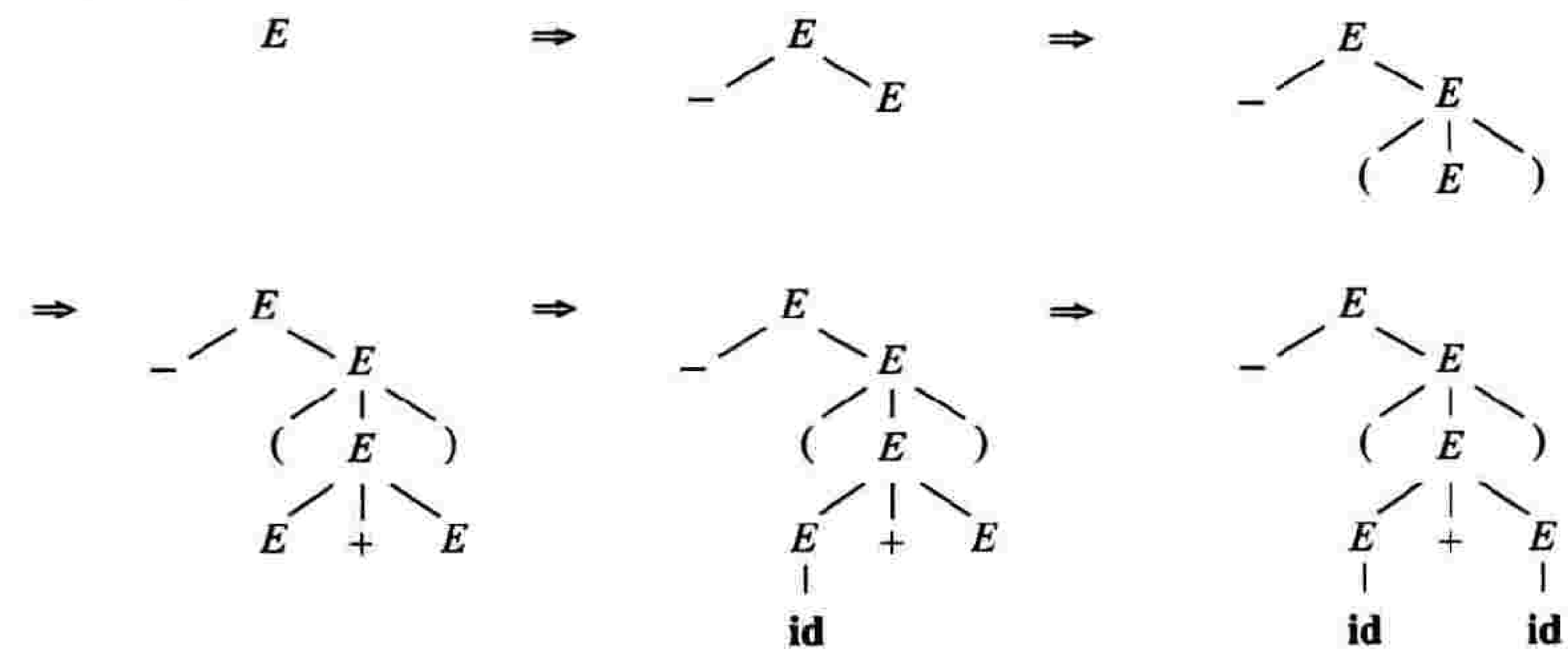


图 3.2 推导(3.2)的分析树

显然,表达式 $-(\mathbf{id}+\mathbf{id})$ 最左推导和最右推导的最终分析树是一样的,也就是分析树忽略了不同的推导次序。不难看出,每棵分析树都有和它对应的最左推导和最右推导。

3.1.4 二义性

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

例 3.4 考虑算术表达式文法(3.1),句子 $\mathbf{id} * \mathbf{id} + \mathbf{id}$ 两种不同的最左推导如下:

$$
\begin{array}{ll}
E \Rightarrow E * E & E \Rightarrow E+E \\
\quad \Rightarrow \mathbf{id} * E & \quad \Rightarrow E * E+E \\
\quad \Rightarrow \mathbf{id} * E+E & \quad \Rightarrow \mathbf{id} * E+E \\
\quad \Rightarrow \mathbf{id} * \mathbf{id}+E & \quad \Rightarrow \mathbf{id} * \mathbf{id}+E \\
\quad \Rightarrow \mathbf{id} * \mathbf{id}+\mathbf{id} & \quad \Rightarrow \mathbf{id} * \mathbf{id}+\mathbf{id}
\end{array}
$$

因而有两棵不同的分析树,见图 3.3。 □

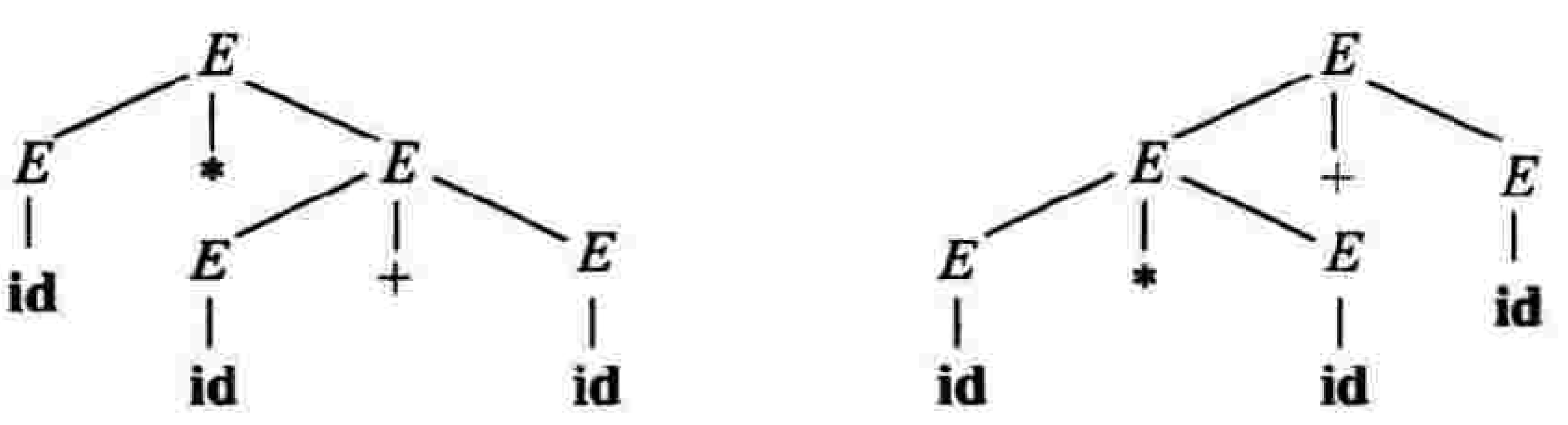


图 3.3 $\mathbf{id} * \mathbf{id}+\mathbf{id}$ 的两棵分析树

注意,图 3.3 右边的分析树反映了+和 * 通常的优先关系,而图 3.3 左边的分析树则不是。也就是,习惯上 * 的优先级高于+,因而表达式 $a * b+c$ 看成 $(a * b)+c$,而不是 $a * (b+c)$。

一个文法,如果存在某个句子有不止一棵分析树与之对应,那么称这个文法是**二义**的。也可以这么说,二义文法是至少存在一个句子有不止一个最左(最右)推导的文法。有些类型的分析

器,它希望处理的文法是无二义的,否则它不能唯一确定对某个句子应选择哪棵分析树。出于某些需要,也可以构造允许二义文法的分析器,不过这样的文法要附带**消除二义性的规则**,以便分析器扔掉不希望的分析树,为每个句子只留一棵分析树。

注意,文法二义并不代表语言一定是二义的。只有当产生一个语言的所有文法都是二义时,这个语言才称为二义的。

3.2 语言和文法

在语言(今后通常就是指编程语言)的设计和编译器的设计方面,文法都提供了很多的优点:

(1) 文法为语言给出了精确的、易于理解的语法说明。

(2) 对于某些文法类,可以为其中的文法自动产生高效的分析器。额外的好处是,分析器的自动构造过程可以揭示文法的二义性和难以分析的构造,而这些问题在语言及其编译器的最初设计阶段很可能未被发现。

(3) 如果文法设计得当,则其中很多非终结符可对应到语言构造(过程、语句和表达式等),这种对应对于把源程序翻译成为正确的目标代码和对于错误诊断都是有用的。把以文法为基础的翻译描述变换为相应编译器或翻译器的工具也是存在的。

(4) 语言也是逐渐完善的,增加新构造以完成新任务的情况时有发生。如果存在以文法为基础的语言的实现,语言新构造的加入就显得更方便。

但是,必须注意,上下文无关文法只能描述编程语言语法上的大部分规定,而不是所有的规定。对输入串的上下文有关的限制,例如要求标识符的声明先于它们的使用,就不可能用上下文无关文法来描述。因此,语法分析之后的阶段必须进一步分析语法分析器的输出,以保证输入串符合分析器无法检查的那些规定,这些事情通常在静态语义检查时完成。

本节首先考虑词法分析器和语法分析器的区别。由于每种分析方法只能处理某类文法,因此有时需要改写文法,以便使它对某种方法来说是可分析的。所以,本节还考虑文法变换的一些规则,以便产生适合于自上而下分析的文法。本节还讨论一些不能用上下文无关文法描述的语言构造,最后简单给出形式语言的小结。

3.2.1 正规式和上下文无关文法的比较

正规式可以描述的语言都能用上下文无关文法来描述。例如正规式$(a|b)^*ab$和上下文无关文法

$$A_0 \to aA_0 \mid bA_0 \mid aA_1$$
$$A_1 \to bA_2$$

$A_2 \rightarrow \varepsilon$

描述同样的语言。

可以机械地把一个非确定的有限自动机变换成一个上下文无关文法,它产生的语言和这个自动机识别的语言相同。上述文法是从图2.6的NFA用下列规则构造的:首先确定终结符号集合,这是简单的。再为NFA的每个状态 i 引入非终结符 A_i,其中 A_0 是开始符号,因为0是开始状态。如果状态 i 有一个 a 转换到状态 j,则引入产生式 $A_i \rightarrow aA_j$;如果是 ε 转换,则引入 $A_i \rightarrow A_j$。如果 i 是接受状态,再引入 $A_i \rightarrow \varepsilon$。

3.2.2 分离词法分析器的理由

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

(1) 语言的词法规则非常简单,不必用功能更强的上下文无关文法描述它。

(2) 对于词法记号,正规式给出的描述比上下文无关文法给出的更简洁且易于理解。

(3) 从正规式自动构造出的词法分析器比从上下文无关文法构造出的更有效。

上面这些理由也决定了词法分析和语法分析的分离。从软件工程的角度看,它们的分离有如下好处:

(1) 编译器的效率会改进。词法分析的分离可以简化词法分析器的设计,允许构造专门的和更有效的词法分析器。编译时相当一部分时间消耗在读源程序和把它分成一个个记号上,专门的读字符和处理记号的技术可以加快编译速度。

(2) 编译器的可移植性加强。输入字符集的特殊性和其他与设备有关的不规则性可以被限制在词法分析器中处理。

(3) 把语言的语法结构分成词法和非词法两部分,为编译器前端的模块划分提供了方便的途径。

哪些应作为词法规则,哪些应作为语法规则,并没有严格的准则。正规式是描述诸如标识符、常数和关键字等词法结构的最有力武器。上下文无关文法是描述括号配对、begin 和 end 配对、语句嵌套、表达式嵌套等结构的最有力武器,这些结构不可能用正规式来描述。

可能还会有这样一个问题,能否把词法分析并入到语法分析中,直接根据字符流进行语法分析,即把语言字母表上的字母作为语法分析的终结符?这是非常困难的。如果词法分析和语法分析合在一起,则必须将语言的注解和空白的规则反映在文法中,这将使文法复杂度大大提高。要自己来处理注解和空白的分析器,比注解和空格已被词法分析器删除的分析器要复杂得多。

3.2.3 验证文法产生的语言

例3.5 考虑文法

$$S \rightarrow (S)S \mid \varepsilon \tag{3.4}$$

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

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

$$S \Rightarrow (S)S \Rightarrow^* (x)S \Rightarrow^* (x)y$$

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

下一步证明任何配对括号串都可由 S 产生，按串长进行归纳。归纳基是指，空串可从 S 经一步推导得到。

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

$$S \Rightarrow (S)S \Rightarrow^* (x)S \Rightarrow^* (x)y$$

从而证明了 $w = (x)y$ 可以由 S 推导出来。 □

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

3.2.4 适当的表达式文法

3.1 节构造的表达式文法有二义性，一个句子的不同分析树体现了不同的算符优先关系和算符结合性。下面构造非二义的有+和 * 运算的表达式文法，该文法和通常的算符优先关系和算符结合性相对应。

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

$$factor \rightarrow \textbf{id} \mid (expr)$$

然后考虑二元算符 *，它有较高的优先级，又是左结合的算符，因而产生式如下：

$$term \rightarrow term * factor \mid factor$$

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

$$expr \rightarrow expr + term \mid term$$

这个表达式文法是无二义的。句子 **id * id * id** 和 **id+id * id** 的分析树如图 3.4 所示。

上面两棵分析树所表现出的算符优先关系和结合性与通常的规定是一致的。可以看出，如果语言语义所规定的算符优先关系和结合性不是这样的话，则文法可能需要重新设计，否则所得

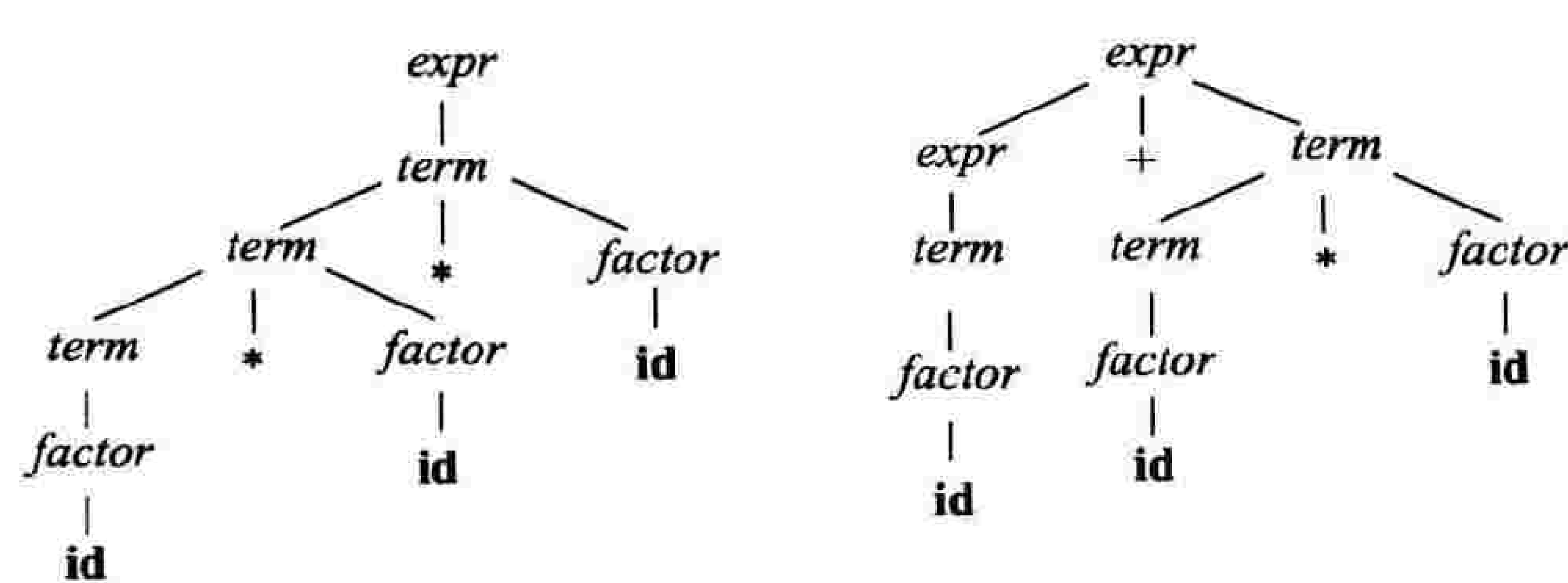


图 3.4 **id** * **id** * **id** 和 **id**+**id** * **id** 的分析树

到的分析树就不能很方便地用于语义分析和中间代码生成等阶段。例如，如果规定 * 和+是右结合的运算，那么文法应该如下：

expr → *term*+*expr* | *term*

term → *factor* * *term* | *factor*

factor → **id** | (*expr*)

比较图 3.5 和图 3.4 的分析树，应该不难看出它们的区别。

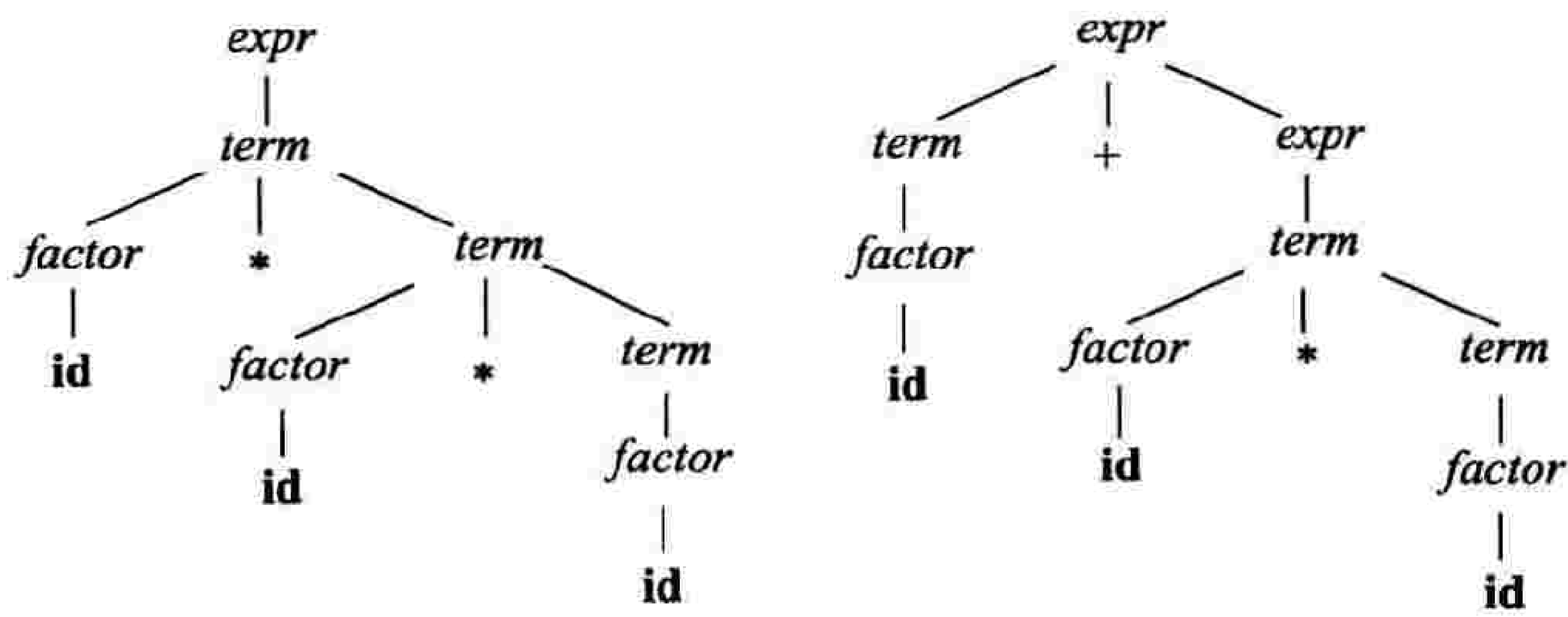


图 3.5 文法修改后的分析树

3.2.5 消除二义性

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

stmt →**if** *expr* **then** *stmt*

| **if** *expr* **then** *stmt* **else** *stmt* (3.5)

| **other**

这里的 **other** 代表任何其他语句。按照这个文法，形式为

if *expr* **then** **if** *expr* **then** *stmt* **else** *stmt* (3.6)

的嵌套条件语句有两个最左推导：

stmt ⇒ **if** *expr* **then** *stmt* ⇒ **if** *expr* **then** **if** *expr* **then** *stmt* **else** *stmt*

$stmt \Rightarrow$ **if** *expr* **then** *stmt* **else** *stmt* $\Rightarrow$ **if** *expr* **then if** *expr* **then** *stmt* **else** *stmt*

因此文法(3.5)是二义的。

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

stmt → *matched_stmt*

| *unmatched_stmt*

matched_stmt → **if** *expr* **then** *matched_stmt* **else** *matched_stmt*

| **other** (3.7)

unmatched_stmt → **if** *expr* **then** *stmt*

| **if** *expr* **then** *matched_stmt* **else** *unmatched_stmt*

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

也许你会问,为什么大多数编程语言都不用无二义的文法(3.7),而采用二义文法(3.5)呢?这是因为,文法(3.7)失去了简洁性。定义语言语法的文法有二义性并不可怕,只要有消除二义性的规则就可以了。

3.2.6 消除左递归

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

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

$A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$

来代替,它们没有改变从 A 推导出的串集。

例 3.6 考虑下面的算术表达文法:

$E \to E+T \mid T$

$T \to T * F \mid F$

$F \to (E) \mid$ **id**

消除 E 和 T 的直接左递归,可以得到

$E \to TE'$

$$E' \to +TE' \mid \varepsilon$$
$$T \to FT' \qquad (3.8)$$
$$T' \to *FT' \mid \varepsilon$$
$$F \to (E) \mid \mathbf{id}$$
□

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

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

其中 β_i 都不以 A 开始,α_i 都非空,然后用

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

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

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

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

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

删除其中的直接左递归,得到如下的文法:

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

由此可见,写一个删除文法左递归的算法并不是件困难的事情。

3.2.7 提左因子

提左因子也是一种文法变换,它用于产生适合于自上而下分析的文法。在自上而下的分析中,当不清楚应该用非终结符 A 的哪个选择来替换它时,可以通过重写 A 产生式来推迟这种决定,推迟到看见足够多的输入,能帮助正确决定所需选择为止。

例如,条件语句有两个产生式:

$$stmt \to \mathbf{if}\ expr\ \mathbf{then}\ stmt\ \mathbf{else}\ stmt$$
$$\mid \mathbf{if}\ expr\ \mathbf{then}\ stmt$$

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

一般来说,如果 $A \to \alpha\beta_1 \mid \alpha\beta_2$ 是 A 的两个产生式,输入串的前缀是从 α 推导出的非空串时,则不知道是用 $\alpha\beta_1$ 还是用 $\alpha\beta_2$ 来扩展 A。但是可以通过先扩展 A 到 $\alpha A'$ 来推迟这个决定。然后,看完了从 α 推出的输入后,再扩展 A' 到 β_1 或 β_2。这就是**提左因子**,原来的产生式成为:

$A \to \alpha A'$

$A' \to \beta_1 \mid \beta_2$

例 3.7 对于悬空 **else** 的文法

stmt →**if** *expr* **then** *stmt* **else** *stmt*

| **if** *expr* **then** *stmt*

| **other**

提左因子后的文法成为

stmt →**if** *expr* **then** *stmt optional_else_ part*

| **other**

optional_else_ part →**else** *stmt*

|ε

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

*3.2.8 非上下文无关的语言构造

在很多编程语言中,仅用上下文无关文法难以实现其中一些语言构造的规范。本节将给出一些这样的构造,并用简单的抽象语言来说明其中的困难。

例 3.8 考虑抽象语言 $L_1 = \{wcw \mid w$ 属于$(a \mid b)^*\}$。L_1的句子的特点是,其前后是由 a 和 b 组成的相同的串,中间由 c 把它们隔开,例如 $aabcaab$。这个抽象语言是对程序中标识符的声明应先于其引用的抽象,wcw 中的第一个 w 代表标识符 w 的声明,第二个代表它的引用。可以证明该语言不是上下文无关语言,但是这个证明超出了本书的范围。这个例子意味着 C 和 Java 都不是上下文无关语言,因为它们都要求标识符的声明先于引用,并且允许标识符任意长。

由于这一点,描述这些语言语法的文法只是用 **id** 这样的记号来代表所有的标识符,而在这些语言的编译器中,由语义分析阶段检查标识符的声明必须先于引用。 □

例 3.9 语言 $L_2 = \{a^n b^m c^n d^m \mid n \geq 0, m \geq 0\}$不是上下文无关语言。$L_2$是正规式 $a^* b^* c^* d^*$ 所表示语言的子集,其要求是 a 和 c 的个数相等,b 和 d 的个数相等。它是对过程声明中形参个数和该过程引用的实参个数应该相同这一问题的抽象,a^n 和 b^m 代表两个过程声明的形参表中分别有 n 和 m 个参数,c^n 和 d^m 分别代表这两个过程调用的实参表。

语言中过程声明和引用的语法并不涉及参数的个数。例如 FORTRAN 的 call 语句可描述为:

stmt→ **call id** (*expr_list*)

expr_list →*expr_list*, *expr*

| *expr*

实参和形参个数的一致性检查也是放在语义分析阶段完成的。 □

例 3.10 语言 $L_3 = \{a^n b^n c^n \mid n \geqslant 0\}$ 也不是上下文无关语言，它是 $L(a^* b^* c^*)$ 中含 a，b 和 c 三个字符个数相等的串。它是对早先排版描述的一个现象的抽象。 □

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

$S \to aSa \mid bSb \mid c$

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

$S \to aSd \mid aAd$

$A \to bAc \mid bc$

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

$S \to AB$

$A \to aAb \mid ab$

$B \to cBd \mid cd$

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

$S \to aSb \mid ab$

值得注意的是，L_3' 是不能用正规式描述的语言的一个范例。证明这一点并不困难。假定 L_3' 可以由某个正规式描述，那么就可以构造一个 DFA D，它接受 L_3'。D 的状态数必定有限，设为 k，设 D 读完 ε，a，aa，…，a^k 分别到达状态 s_0，s_1，…，s_k，也就是 D 读 i 个 a 后到达状态 s_i。

因为 D 只有 k 个不同的状态，那么在序列 s_0，s_1，…，s_k 中至少有两个状态相同，例如是 s_i 和 s_j($i<j$)。从状态 s_i 出发，D 可以接受 i 个 b 到达一个接受状态 f，因为 $a^i b^i$ 属于 L_3'。同时，D 还存在着一条从状态 s_0 到状态 s_i 再到 f 的路径，该路径的标记为 $a^j b^i$，如图 3.6 所示。于是，D 也接受 $a^j b^i$，但它不在 L_3' 中，这和 D 接受的语言是 L_3' 的假设矛盾。

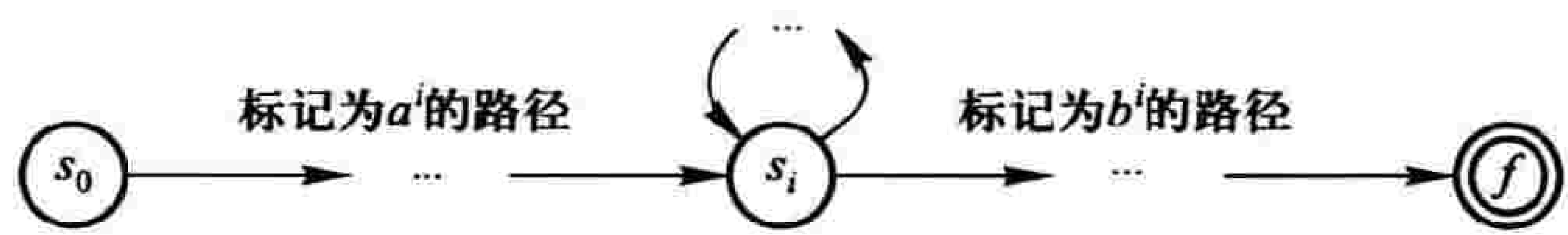


图 3.6 接受 $a^i b^i$ 和 $a^j b^i$ 的 DFA D

通俗地说，有限自动机不能计数，即有限自动机不能接受像 L_3' 这样的语言，它要求在接受 b 以前数出 a 的个数。类似地，上下文无关文法可以计两项的数量，但不能计三项的数量，即可用它定义 L_3'，但不能定义 L_3。

*3.2.9 形式语言鸟瞰

是否可以用功能更强的文法描述上面那些不能用上下文无关文法描述的语言呢？回答是肯

定的。

乔姆斯基(Chomsky)把文法分成四种类型,即0型、1型、2型和3型,0型的描述能力强于1型,1型的强于2型,2型的强于3型。这几类文法的差别在于对产生式施加不同的限制。

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

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

如果对0型文法加上以下第 i 条限制,就可以得到 i 型文法:

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

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

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

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

2型文法也就是**上下文无关文法**,非终结符的替换不必考虑上下文。

3型文法等价于正规式,因而也称**正规文法**。

前面提到的语言 $L_3=\{a^n b^n c^n \mid n \geqslant 1\}$ 可以用上下文有关文法来定义,其产生式如下:

$S \to aSBC$ $\quad bB \to bb$

$S \to aBC$ $\quad bC \to bc$

$CB \to BC$ $\quad cC \to cc$

$aB \to ab$

$a^n b^n c^n$ 的推导过程如下:

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

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

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

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

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

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

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

由此可见,上下文有关文法的能力强于上下文无关文法。

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

由于下面只涉及上下文无关文法,因而把它简称为文法。

3.3 自上而下分析

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

3.3.1 自上而下分析的一般方法

自上而下分析的宗旨是，对任何输入串，试图用一切可能的办法，从文法开始符号（根结点）出发，自上而下，从左到右地为输入串建立分析树。或者说，为输入串寻找最左推导。这种分析过程本质上是一种试探过程，是反复使用不同的产生式谋求匹配输入串的过程。

例 3.11 若有文法

$S \rightarrow aCb$

$C \rightarrow cd \mid c$

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

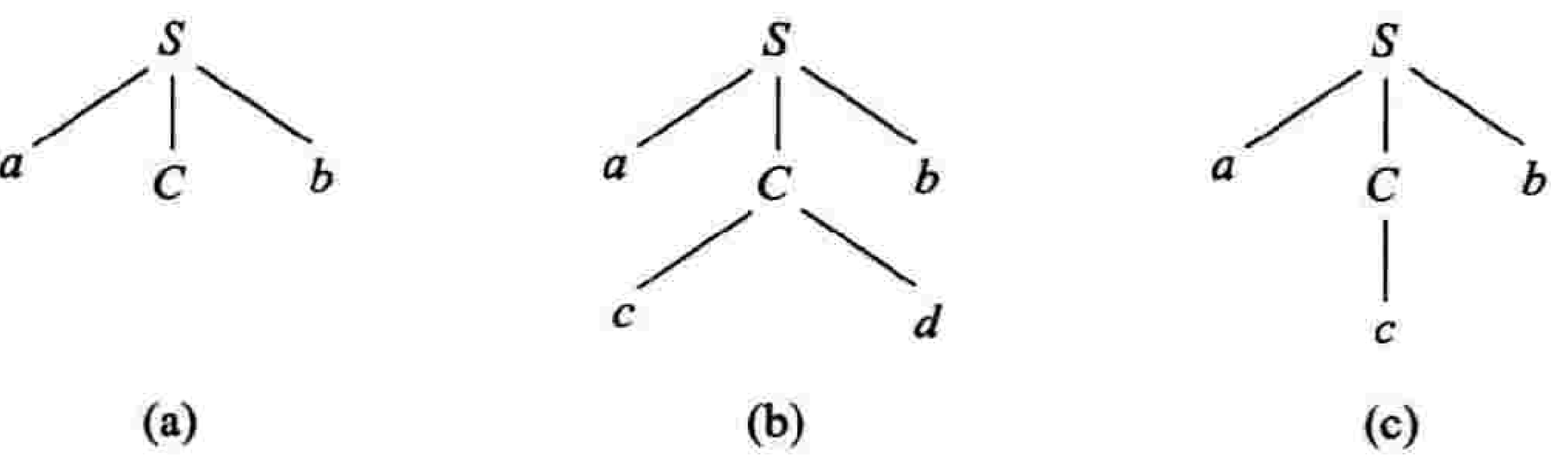


图 3.7 自上而下分析的试探过程

树中最左边的叶子标记为 a，匹配 w 的第一个符号。于是，推进输入指针到 w 的第二个符号 c，并考虑分析树中下一个叶子 C，它是非终结符。用 C 的第一个选择来扩展 C，得到图 3.7(b)的树。现在第二个输入符号 c 能匹配，再推进输入指针到 b，把它和分析树中的下一个叶子 d 比较。因为 b 和 d 不匹配，回到 C，看它是否还有别的选择尚未尝试。

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

上述这种自上而下分析法存在着困难和缺点。首先，如果存在非终结符 A，并且有

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

这样的左递归，当试图用 A 去匹配输入串时有可能使分析过程陷入无限循环（若输入串不是一个

句子，则一定陷入无限循环），因为可能在没有吃进任何输入符号的情况下，又得要求用下一个 A 去进行新的匹配。因此，使用自上而下分析法时，文法应该没有左递归。

其次，当非终结符用某个选择匹配成功时，这种成功可能仅是暂时的。由于这种虚假现象，需要使用复杂的回溯技术。

最后，试探与回溯是一种穷尽一切可能的办法，效率低、代价高，它只有理论意义，在实践中价值不大。

3.3.2 LL(1)文法

为构造不带回溯的自上而下分析算法，首先要消除文法的左递归，并找出避免回溯的充分必要条件。消除左递归的方法已介绍了，下面讨论如何避免回溯。

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

在讨论不得回溯的前提对文法有什么限制之前，先定义两个和文法有关的函数。一个文法的符号串 α 的开始符号集合 $FIRST(\alpha)$ 是

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

特别是，$\alpha \Rightarrow^* \varepsilon$ 时，规定 $\varepsilon \in FIRST(\alpha)$。如果对 A 的任何两个不同的选择 α_i 和 α_j，有

$$FIRST(\alpha_i) \cap FIRST(\alpha_j) = \varnothing$$

那么，当要求 A 匹配输入串时，A 就能根据它所面临的第一个输入符号 a，准确地指派某一个选择前去执行任务。这个选择就是满足 $a \in FIRST(\alpha)$ 的那个 α。

已经介绍过，用提取左因子的办法，可以把文法改造成对任何非终结符 A，A 的所有选择的开始符号集合两两不相交。

如果 ε 属于 A 的某个选择的开始符号集合，那么问题就比较复杂，需要定义文法非终结符的后继符号集合后才能解释。非终结符 A 的后继符号集合 $FOLLOW(A)$ 是所有在句型中可以直接出现在 A 后面的终结符的集合，也就是

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

如果 A 是某个句型的最右符号，那么 \$ 也属于 $FOLLOW(A)$。

例 3.12 考虑(3.8)的文法，把它重复如下：

$$E \to TE' \qquad T' \to *FT' \mid \varepsilon$$
$$E' \to +TE' \mid \varepsilon \qquad F \to (E) \mid \mathbf{id}$$
$$T \to FT'$$

那么

$$FIRST(E) = FIRST(T) = FIRST(F) = \{\ (,\ \mathbf{id}\ \}$$
$$FIRST(E') = \{+,\ \varepsilon\}$$

$FIRST(T') = \{*, \varepsilon\}$

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

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

$FOLLOW(E) = FOLLOW(E') = \{), \$ \}$

$FOLLOW(T) = FOLLOW(T') = \{ +,), \$ \}$

$FOLLOW(F) = \{ +, *,), \$ \}$ □

设计一个算法来计算开始符号集合和后继符号集合是件简单的事情。

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

这样，要想不出现回溯，需要文法的任何两个产生式 $A \rightarrow \alpha \mid \beta$ 都满足下面两个条件：

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

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

把满足这两个条件的文法称为 **LL(1) 文法**，其中的第一个“L”代表从左向右地扫描输入，第二个“L”表示产生最左推导，“1”代表在决定分析器的每步动作时需要向前查看下一个输入符号(即输入指针所指向的符号)。除了没有公共左因子外，LL(1)文法还有一些明显的性质，它不是二义的，也不含左递归。还有一些性质将在 3.3.5 节构造预测分析表时再介绍。很明显，(3.8)的表达式文法是 LL(1)的。

3.3.3 递归下降的预测分析

所谓预测分析是指能根据当前的输入符号为非终结符确定采用哪一个选择，LL(1)文法是满足这个要求的。递归下降的预测分析是指为每一个非终结符写一个分析过程，由于文法的定义是递归的，因此这些过程也是递归的。在处理输入串时，首先执行的是开始符号所对应的过程，然后根据产生式右部出现的非终结符，依次调用相应的过程，这种逐步下降的过程调用序列隐含地建立了输入的分析树。

还是通过一个例子来说明如何构造递归下降的预测分析程序。下面的文法产生 Pascal 语言的类型子集，用记号 **dotdot** 表示“..”以强调这个字符序列作为一个词法单元。

type → *simple*
  | ↑ **id**
  | **array** [*simple*] **of** *type*
simple → **integer**
  | **char**

| **num dotdot num**

显然,该文法是 LL(1)的。

图 3.8 是上面类型定义文法的递归下降预测分析器。这个分析器包括处理非终结符 *type* 和 *simple* 的过程以及附加的过程 match()。使用 match()是为了简化 type()和 simple()的代码,如果它的参数匹配当前面临的符号,它就调用函数 nextToken(),取下一个记号,并更新变量 lookahead的值。

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

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

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

图 3.8 预测分析器的代码

3.3.4 非递归的预测分析

如果显式地维持一个栈,而不是隐式地通过递归调用,那么可以构造非递归的预测分析器。预测分析的关键问题是,在扩展一个非终结符时怎样为它选择合适的产生式,图 3.9 中的非递归的预测分析器通过查分析表来决定产生式。

表驱动的预测分析器有一个输入缓冲区、一个栈、一张分析表和一个输出流。输入缓冲区包含被分析的串,后面跟一个符号 \$,它是输入串的结束标记。栈中存放文法的符号串,栈底符号是 \$ 。初始时,栈中含文法的开始符,它在 \$ 的上面。分析表是一个二维数组 $M[A, a]$, A 是非

输入 | a | + | b | $

栈 X Y Z $　预测分析程序　输出

分析表M

图 3.9　非递归的预测分析器的模型

终结符，a 是终结符或 \$ 。

现在说明这个分析器的工作过程。预测分析程序根据当前的栈顶符号 X 和输入符号 a 决定分析器的动作，它有四种可能：

（1）如果 $X = a = \$$ ，分析器宣布分析完全成功而停机。

（2）如果 $X = a \neq \$$ ，分析器弹出栈顶符号 X，并推进输入指针，使之指向下一个符号。

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

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

算法 3.1　非递归的预测分析。

输入　串 w 和文法 G 的分析表 M。

输出　如果 w 属于 $L(G)$，则输出 w 的最左推导，否则报告错误。

方法　初始时分析器的格局是：\$$S$在栈里，其中 S 是开始符号并且在栈顶；w\$ 在输入缓冲区中，图 3.10 是用预测分析表 M 对输入串进行分析的程序。 □

例 3.13　考虑文法(3.8)，该文法的预测分析表见表 3.1，表中空白表示出错，非空白指示一个产生式，用来替换栈顶的非终结符。

表 3.1　文法(3.8)的分析表

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

续表

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

如果输入是 **id** * **id**+**id**,分析过程中各部分的变化则如表 3.2。输入指针指在输入串最左边的符号。仔细观察分析器的输出动作可知,分析器跟踪的是输入的最左推导,也就是输出最左推导的那些产生式。已匹配的输入符号加上栈中的文法符号(从顶到底),构成最左推导的句型。 □

```
让 ip 指向 w $ 的第一个符号;
令 X 等于栈顶符号;
while (X != $ ) { /* 栈非空 */
    if (X 是 a) 把 X 从栈顶弹出并把 ip 推进到指向下一个符号;
    else if(X 是终结符) error ( );
    else if(M[X, a]是出错入口) error ( );
    else if(M[X, a] = X →Y1 Y2 ··· Yk) {
        输出产生式 X → Y1 Y2 ··· Yk;
        从栈中弹出 X;
        把 Yk, Yk-1, ···, Y1 依次压入栈,Y1 在栈顶;
    }
    令 X 等于栈顶符号;
}
```

图 3.10 预测分析程序

表 3.2 预测分析器接受输入 id * id + id 的动作

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

续表

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

3.3.5 构造预测分析表

对于非递归的预测分析来说，剩下的问题是如何构造预测分析表。下面的算法为文法 G 构造预测分析表 $M[A, a]$，其中 A 是非终结符，a 是终结符或 \$ 。这个算法的思想如下：如果 $A \to \alpha$ 是产生式且 a 在 $FIRST(\alpha)$ 中，那么在当前输入符号为 a 时，分析器选择用 α 展开 A。唯一的复杂情况是 $\alpha \Rightarrow^* \varepsilon$，在这种情况下，如果当前输入符号（包括 \$ ）在 $FOLLOW(A)$ 中，仍应用 α 展开 A。

算法 3.2 构造预测分析表。

输入　文法 G。

输出　分析表 M。

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

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

(2) 如果 ε 在 $FIRST(\alpha)$ 中，对 $FOLLOW(A)$ 的每个终结符 b（包括 \$ ），把 $A \to \alpha$ 加入 $M[A, b]$（包括 $M[A, \$]$）。

M 剩下的条目没有定义，都是出错条目，通常用空白表示。 □

例 3.14 把算法 3.2 用于文法(3.8)，产生的分析表见表 3.1。因为 $FIRST(TE') = FIRST(T) = \{(, \textbf{id}\}$，因此产生式 $E \to TE'$ 加入 $M[E, (]$ 和 $M[E, \textbf{id}]$。

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

算法 3.2 可用于任何文法 G 来产生分析表 M。然而对某些文法,M 可能含有一些多重定义的条目。例如 G 是左递归或二义的,则 M 至少含一个多重定义的条目。

例 3.15 以例 3.7 的条件语句为例,为方便起见,将文法重写如下:

stmt → **if** *expr* **then** *stmt* *e_part* | **other** /* *e_part* 指 *optional_else_part* */

e_part→ **else** *stmt* | ε (3.9)

expr → *b*

它的分析表见表 3.3。

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

可以证明,一个文法的预测分析表没有多重定义的条目,当且仅当该文法是 LL(1)的。还可以证明,算法 3.2 为 LL(1)文法 G 产生的分析表能分析 $L(G)$的所有句子,也仅能分析 $L(G)$的句子。

剩下的一个问题是,当分析表有多重定义的条目时应该怎么办。一种办法是求助于文法变换,消除左递归和提取所有可能的左因子,以期得到的新文法的分析表没有多重定义的条目。遗憾的是,有些文法不论怎么变化也不能产生 LL(1)文法,文法(3.9)是一个这样的例子,它的语言没有 LL(1)文法。正如所看见的那样,让 M[*e_part*, **else**] = {*e_part*→ **else** *stmt* },仍可以用预测分析器对(3.9)进行分析。但是,一般来说,没有一个普遍适用的规则来删除多重定义的条目,使其成为单值而不影响分析器可识别的语言。

表 3.3 文法(3.9)的预测分析表

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

使用预测分析的主要困难在于为源语言写一个能构造出预测分析器的文法。虽然左递归的消除和提左因子是简单的,但它们使得结果文法难于理解而且不易于翻译。3.5 节介绍的 LR 分析器可以克服这些问题。

3.3.6 预测分析的错误恢复

在讨论预测分析的错误恢复前,先对编译器的错误处理作一个概述。

如果编译器只处理正确的程序,它的设计和实现可以大大简化,但是程序员往往不是一次就能把程序编写正确的,好的编译器应能帮助程序员识别和定位错误。虽然错误是那样容易发生,但是几乎所有编程语言的规范都没有给出编译器应该怎样处理语法错误的描述。

在设计编译器时,如果从一开始就规划错误处理,那么就可以简化编译器的结构,并且改进它对错误的响应。

普通的编程错误会出现在不同的层次上:

(1) 词法错误。如标识符、关键字或算符的拼写错,遗漏字符串两端的引号。在第2章已经提到,这些基于程序员观点的词法错误并非一定是在词法分析阶段发现的。

(2) 语法错误。如算术表达式的括号不配对。这些错误在语法分析阶段可以发现。另外一些情况,例如在C和Java中,case语句如果没有外围的switch语句,则是一个语法错误。由于这样的约束并没有体现在语言的文法中,因此分析器通常发现不了这种错误,但是它会在编译器试图产生代码时或在其他场合被捕获。

(3) 语义错误。如算符和运算对象之间类型不匹配。具体例子有,在Java方法中,不带表达式的return语句要求相应方法的返回类型是void,否则就是错误。

(4) 逻辑错误。如在C程序中用赋值号“=”代替了比较算符“==”。这时,从编译器看,整个程序可能仍然是合法的,但是它未能正确反映程序员的意图。普通的编译器难以发现这样的错误。

之所以强调分析期间的错误恢复,是因为大多数错误是语法错误。少数语义错误,如类型不匹配,也能够被有效地诊断出来;但是一般而言,在编译时精确地诊断语义错误和逻辑错误是一件困难的事情。

分析器对错误处理的基本目标是:

(1) 清楚而准确地报告错误的出现;

(2) 迅速地从每个错误中恢复过来,以便诊断剩余程序的错误;

(3) 它不应该使处理正确程序的速度降低太多。

这些目标的有效实现是非常困难的。幸好,常见的错误是简单的,直截了当的错误处理机制一般就够用了。但是,在有些场合中,发生错误的实际位置远远先于发现它的位置,并且这种错误的准确性质也难以推断。而在另一些困难的场合,错误处理程序甚至还需要猜想程序员的本意。

错误处理程序应怎样报告错误?至少,它应该报告源程序的错误被检测到的位置(它可能偏离错误的真正位置)。很多编译器采用的办法是,打印出错的程序行,指出检测到错误的地方。如果能够知道实际错误很可能是什么的话,这时编译器还会附带一个诊断信息,如“此处漏了分号”。

下面回到语法分析阶段,考虑分析器的错误恢复办法。分析方法的准确性使得语法错误的诊断非常有效,本章所介绍的分析方法都能及时诊断语法错误。一旦查出错误,分析器应如何恢复?后面会结合各种语法分析方法介绍几种一般性的策略,但没有哪一种策略占明显优势。最

简单的方式是检测到一个错误时,分析器就暂停分析并给出有关该错误的信息。然后,如果分析器能够到达一个状态,从该状态可以对剩余输入继续分析,并且很有希望继续给出有意义的诊断信息,则分析器从该状态恢复,使后续程序的错误还能被揭示出来。如果错误信息一大堆,则编译器最好的办法是,在错误信息数目超过某个上限时放弃骚扰程序员的雪崩式伪错误。

现在讨论预测分析的错误恢复。非递归预测分析器的栈使得分析器希望和剩余输入匹配的终结符和非终结符变得明显,在下面的错误恢复讨论中将引用该栈中的符号。这种技术也可用到递归下降的分析中。

当栈顶的终结符和下一个输入符号不匹配,或者栈顶是非终结符 A,输入符号是 a,而 $M[A, a]$ 指示出错,则预测分析器发现一个错误。

对于非递归的预测分析,这里介绍**紧急方式的错误恢复**,这是最简单的方法。发现错误时,分析器每次抛弃一个输入记号,直到输入记号属于某个指定的同步记号集合为止。该方法适用于大多数分析方法。该方法的效果依赖于同步记号集合的选择,编译器的设计者必须选择适当的同步记号,使分析器能迅速从错误中恢复过来。这种方法的缺点是常常会放弃一段输入记号,不检查其中是否有其他错误。有关同步记号选择的一些提示如下。

(1) 首先考虑的是,至少可以把 $FOLLOW(A)$ 的所有终结符放入非终结符 A 的同步记号集合。即出错时,若栈顶是 A,则放弃一些记号,直到看见 $FOLLOW(A)$ 的元素为止,然后把 A 弹出栈,分析一般可以继续下去。这样做本质上是放弃对 A 推出部分的分析,在其后恢复分析。

(2) 仅使用 $FOLLOW(A)$ 作为 A 的同步记号集合是不够的。例如,分号在 C 语言中作为语句的结束符,那么作为语句开始符号的关键字没有出现在表达式非终结符的 $FOLLOW$ 集合中。这样,仅按上面(1)来设定同步记号集合,作为赋值结束的分号被遗漏时,会引起下一语句的开始关键字被放弃,甚至该语句整个被放弃。

一种语言的各种构造往往构成一种层次结构,如表达式出现在语句中,语句出现在程序块中等。可以把高层构造的开始符号加到低层构造的同步记号集合中,例如,可以把语句开始的关键字加入到表达式非终结符的同步记号集合。

(3) 如果把 $FIRST(A)$ 的终结符加入 A 的同步记号集合,恢复关于 A 的分析是可能的,只要 $FIRST(A)$ 的终结符出现在输入中。

(4) 如果一个非终结符可以推导出空串,则把能推导出空串的产生式作为默认产生式,即出错时若栈顶是这样的非终结符,则使用这种默认产生式。这样做会延迟错误的发现,但不会遗漏,好处是可以减少错误恢复要考虑的非终结符数。

(5) 如果终结符在栈顶而不能匹配,简单的办法是,除了报告错误外,弹出此终结符,继续分析。效果上,这种方式等于把所有其他的记号作为该终结符的同步记号集合。

例 3.16 按照文法(3.8)分析表达式时,使用 $FOLLOW$ 符号和 $FIRST$ 符号作为同步记号是合情合理的。该文法的分析表(表 3.1)重复于表 3.4,并用 synch 来指示从非终结符的 $FOLLOW$ 集合中得到的同步记号。非终结符的 $FOLLOW$ 集合从例 3.12 可得。

表 3.4 的使用如下:如果分析器查找条目 $M[A, a]$,发现它是空的,则跳过输入符号 a;如果

条目是 synch，则调用同步过程并把栈顶的非终结符弹出，恢复分析；如果栈顶的记号与输入符号不匹配，则从栈顶弹出该记号，如上面所提到的那样。

表3.4的分析器和错误恢复机制面临有语法错误的输入+**id** * +**id** 的行为见表3.5。 □

表3.4 同步记号加到表3.1的分析表上

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

表3.5 由预测分析器产生的分析和错误恢复动作

栈	输入	输出
$E	+**id** * +**id** $	出错，跳过+
$E	**id** * +**id** $	**id** 属于 *FIRST*(*E*)
$E'T	**id** * +**id** $	
$E'T'F	**id** * +**id** $	
$E'T' **id**	**id** * +**id** $	
$E'T'	* +**id** $	
$E'T'F *	* +**id** $	
$E'T'F	+**id** $	出错；“+”正好在 *F* 的同步记号集合中，无须跳过任何记号；*F* 被弹出
$E'T'	+**id** $	
$E'	+**id** $	
$E'T+	+**id** $	
$E'T	**id** $	
$E'T'F	**id** $	
$E'T'**id**	**id** $	
$E'T'	$	
$E'	$	
$	$	

上面讨论的紧急方式恢复没有涉及错误信息这个重要问题。一般来说,报告错误的信息必须由编译器的设计者提供。

3.4 自下而上分析

本节介绍自下而上分析的一般风格,称做移进-归约分析。编译器常用的移进-归约分析方法称为 LR 分析,将在 3.5 节讨论。

3.4.1 归约

移进-归约分析为输入串构造分析树是从叶结点开始的,朝着根结点方向逆序前进。可以把这个过程看成是把输入串**归约**成文法的开始符号。在每一步归约中,一个子串和某个产生式的右部匹配,然后用该产生式的左部符号代替这个子串。如果每步都能恰当地选择子串,那么它实际跟踪的是最右推导过程的逆过程。

例 3.17 考虑文法

$S \to aABe$

$A \to Abc \mid b$

$B \to d$

句子 $abbcde$ 可以按如下步骤归约成 S。首先扫描 $abbcde$,寻找能够匹配某产生式右部的子串,子串 b 和 d 都可以。选择最左边的 b,用 A 代替(因为有 $A \to b$)它,得到 $aAbcde$。现在子串 Abc, b 和 d 分别都匹配一个产生式的右部,其中 Abc 和 b 都是匹配 A 产生式一个右部的最左子串,用 A 代替子串 b 会使归约进行不下去,而用 A 代替子串 Abc(有 $A \to Abc$)则可以,因此得到 $aAde$。然后,因有 $B \to d$,那么用 B 代替 d,得 $aABe$,再用 S 代替此串。这样,归约序列

$abbcde, aAbcde, aAde, aABe, S$

表示从 $abbcde$ 到 S 的归约。事实上,这些归约刻画出 $abbcde$ 的最右推导过程

$S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcde$

的逆过程。 □

3.4.2 句柄

非形式地说,句型的**句柄**(handle)是该句型中和一个产生式右部匹配的子串,并且,把它归约成该产生式左部的非终结符代表了最右推导过程的逆过程中的一步。在很多情况下,句型中能和产生式 $A \to \beta$ 右部匹配的最左子串 β 就是句柄;但并非总是这样,有的时候用这个产生式归约后得到的串不能归约到开始符号。对于例 3.17 的第二个句型 $aAbcde$,如果用 A 代替 b, 得到

$aAAcde$,那它就不能归约成 S。基于这一点,必须给句柄更精确的定义。

形式地说,右句型(最右推导可得到的句型)γ 的句柄是一个产生式的右部 β 以及 γ 中的一个位置,在这个位置可找到串 β,用 A 代替 β(有产生式 $A\to\beta$)得到最右推导的前一个右句型。即,如果 $S \Rightarrow^{*}_{rm} \alpha Aw \Rightarrow_{rm} \alpha\beta w$,那么在 α 后的 β 是 $\alpha\beta w$ 的句柄。句柄右边的 w 仅含终结符。注意,如果文法二义,那么句柄可能不唯一,因为一个句子可能不止一个最右推导。只有文法无二义时,它的每个右句型才有唯一的句柄。

在上面的示例中,$abbcde$ 是右句型,它的句柄是 $A \to b$ 的右部 b,并且在位置 2。同样,$aAbcde$ 也是右句型,它的句柄是 Abc(有产生式 $A \to Abc$),并且也在位置 2。

例 3.18 考虑文法

$$E\to E+E \qquad E \to (E)$$
$$E\to E*E \qquad E \to \mathbf{id} \tag{3.10}$$

和最右推导:

$$\begin{aligned} E &\Rightarrow_{rm} \underline{E*E} \\ &\Rightarrow_{rm} E*\underline{E+E} \\ &\Rightarrow_{rm} E*E+\underline{\mathbf{id}_3} \\ &\Rightarrow_{rm} E*\underline{\mathbf{id}_2}+\mathbf{id}_3 \\ &\Rightarrow_{rm} \underline{\mathbf{id}_1}*\mathbf{id}_2+\mathbf{id}_3 \end{aligned}$$

为方便起见,给 **id** 以下标,并给每个右句型的句柄加下划线。例如,$\mathbf{id}_1$ 是右句型 $\mathbf{id}_1*\mathbf{id}_2+\mathbf{id}_3$ 的句柄。注意,句柄右边的串仅含终结符。

文法(3.10)是二义的,该句子还存在着另一个最右推导:

$$\begin{aligned} E &\Rightarrow_{rm} \underline{E+E} \\ &\Rightarrow_{rm} E+\underline{\mathbf{id}_3} \\ &\Rightarrow_{rm} \underline{E*E}+\mathbf{id}_3 \\ &\Rightarrow_{rm} E*\underline{\mathbf{id}_2}+\mathbf{id}_3 \\ &\Rightarrow_{rm} \underline{\mathbf{id}_1}*\mathbf{id}_2+\mathbf{id}_3 \end{aligned}$$

考虑右句型 $E*E+\mathbf{id}_3$,在这个推导中 $E*E$ 是句柄,而在上一个推导中 $\mathbf{id}_3$ 是句柄。

此例的两个最右推导类似于例 3.4 中的两个最左推导。第一推导体现了+的优先级高于 *,而第二个则反过来。 □

3.4.3 用栈实现移进-归约分析

如果使用移进-归约的方式分析句子,有两个问题必须解决。第一个是怎样确定右句型中将要归约的子串;第二个是若被归约子串碰巧是多个产生式的右部,如何确定选择哪一个产生式。在讨论这些问题之前,首先看一下移进-归约分析器所使用数据结构的类型。

实现移进-归约分析的一种便利办法是用栈保存文法符号,用输入缓冲区保存要分析的串

w,用 \$ 标记栈底,也用它标记输入串的右端。起初,栈是空的,串 w 在输入中,如下所示:

栈　　　　输入

\$　　　　$w\$$

分析器移动若干个(包括零个)输入符号入栈,直到句柄 β 在栈顶为止,再把 β 归约成恰当的产生式左部。分析器重复这个过程,直到它发现错误或者栈中只含开始符号并且输入串为空为止:

栈　　　　输入

$\$S$　　　　\$

进入这个格局后,分析器停机并宣告分析完全成功。

例 3.19　逐步观察移进-归约分析器分析输入串 $\mathbf{id}_1 * \mathbf{id}_2 + \mathbf{id}_3$ 时的动作,其中文法按(3.10),并采用例 3.18 的第一种最右推导过程的逆过程。动作序列见表 3.6。注意,由于该输入有两种最右推导,所以还存在分析器可取的另一个动作序列。□

分析器的基本动作有移进和归约,实际可能的动作还有两种:接受和报错。

(1) **移进**动作:把下一个输入符号移进栈。

(2) **归约**动作:分析器知道句柄的右端已在栈顶,然后它确定句柄的左端在栈中的位置,再决定用什么样的非终结符代替句柄。

(3) **接受**动作:分析器宣告分析成功。

(4) **报错**动作:分析器发现语法错误,调用错误恢复例程。

有一个重要的事实说明在移进-归约分析中栈的使用是合理的:句柄最终总是出现在栈顶而不是在栈的里面。从表 3.6 可以看出,这个事实是明显的,在第一步归约前和每一步归约后,分析器必须移进若干个(包括零个)符号以使下一个句柄进栈,但它决不需要深入栈中查找句柄。由此可知,使用栈对实现移进-归约是特别方便的。当然还必须解释怎样选取动作才能使移进-归约分析器正常工作,马上要讨论的 LR 分析就是一种这样的技术。

表 3.6　移进-归约分析器对于输入 $\mathbf{id}_1 * \mathbf{id}_2 + \mathbf{id}_3$ 的格局

栈	输入	动作
\$	$\mathbf{id}_1 * \mathbf{id}_2 + \mathbf{id}_3\$$	移进
$\$\mathbf{id}_1$	$* \mathbf{id}_2 + \mathbf{id}_3\$$	按 $E \to \mathbf{id}$ 归约
$\$E$	$* \mathbf{id}_2 + \mathbf{id}_3\$$	移进
$\$E *$	$\mathbf{id}_2 + \mathbf{id}_3\$$	移进
$\$E * \mathbf{id}_2$	$+ \mathbf{id}_3\$$	按 $E \to \mathbf{id}$ 归约
$\$E * E$	$+ \mathbf{id}_3\$$	移进
$\$E * E +$	$\mathbf{id}_3\$$	移进
$\$E * E + \mathbf{id}_3$	\$	按 $E \to \mathbf{id}$ 归约
$\$E * E + E$	\$	按 $E \to E + E$ 归约
$\$E * E$	\$	按 $E \to E * E$ 归约
$\$E$	\$	接受

3.4.4 移进-归约分析的冲突

有些上下文无关文法不能使用移进-归约分析。这种文法的移进-归约分析器会到达以下这样的格局:它根据栈中所有的内容和下一个输入符号难以决定是移进还是归约(**移进-归约冲突**),或难以决定按哪一个产生式进行归约(**归约-归约冲突**)。现在给出一些语法构造的例子,它们就属于这类文法。从技术上讲,这些文法不属于3.5节定义的LR(k)类,称它们为非LR文法。

例3.20 二义文法绝不是LR的,例如考察悬空else文法(3.5)

stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| **other**

如果移进-归约分析器处于格局

栈	输入
⋯ **if** *expr* **then** *stmt*	**else** ⋯ $

则不知道 **if** *expr* **then** *stmt* 是否为句柄,产生移进-归约冲突,所以这个文法不是LR(1)。更一般地说,没有一种二义文法是LR(k)的(对任何k)。 □

不过,必须指出,移进-归约分析还是可以用来分析某些二义文法的,如上面的if-then-else文法。当为包括条件语句两个产生式的文法构造这样的分析器时,存在着上面所讲的冲突,如果采用优先移进的策略来解决这个冲突,分析器的行为就自然了。3.6节将会讨论这种文法的分析器。

出现非LR文法的另一种情况是,知道了句柄,但根据栈里的内容和下一个输入符号不足以决定按哪个产生式归约。下面的例子说明这种情况。

例3.21 假定词法分析器对任何标识符都回送记号**id**而不管它是如何使用的,假如语言的过程调用是给出它们的名字和参数表,并且数组元素的引用也用同样的语法。因为过程调用的参数和数组引用的下标的翻译是不一样的,因此需要用不同的产生式来产生实参表和下标表。这样,文法可以有下面一些产生式:

(1) *stmt* → **id** (*parameter_list*)
(2) *stmt* → *expr* = *expr*
(3) *parameter_list* → *parameter_list*, *parameter*
(4) *parameter_list* → *parameter*
(5) *parameter* → **id**
(6) *expr* → **id** (*expr_list*)
(7) *expr* → **id**
(8) *expr_list* → *expr_list*, *expr*
(9) *expr_list* → *expr*

由 $p(i, j)$ 开始的语句经词法分析后,变为记号流 **id** (**id**, **id**)进入分析器。把前三个记号移进栈后,分析器的格局是:

栈　　　　　　输入

… **id** (**id**　　　, **id**)…

很明显,栈顶的 **id** 必须归约,但是按哪个产生式归约?如果 p 是过程,应按产生式(5)归约;如果 p 是数组,应按产生式(7)归约。但是栈中的信息不能告诉应按哪个产生式归约,必须使用符号表中有关 p 的信息。

解决这个问题的一种办法是把产生式(1)的记号 **id** 改为 **procid**,并且使用更聪明一点的词法分析器,它识别出作为过程名的标识符时,返回记号 **procid**。当然,这样做要求词法分析器在返回记号前访问符号表。

这样修改后,处理 $p(i, j)$ 时分析器处于如下格局:

栈　　　　　　输入

…**procid** (**id**　　　, **id**) …

或处于先前的那个格局。前者用产生式(5) 归约, 后者用产生式(7)归约。注意,栈中的第三个符号用来决定归约用的产生式,虽然它本身不包含在这个归约中。移进-归约分析可以深入栈里取信息来指导分析。 □

3.5 LR 分析器

本节提出一种高效的、自下而上的语法分析技术,它能适用于一大类上下文无关文法的分析。这种技术称为 LR(k)分析技术,L 是指从左向右扫描输入,R 是指构造最右推导的逆,k 是指在决定分析动作时向前查看的符号个数。(k)省略时,表示 k 是 1。

在讨论 LR 分析算法后,本书提出构造 LR 分析表的三种技术。第一种方法称为简单的 LR 方法(简称 SLR),它最容易实现,但功能最弱。对某些文法,用另外两种方法能成功地产生分析表,但用它却失败。第二种方法称为规范的 LR 方法,它功能最强,但代价也最大。第三种方法称为向前搜索的 LR 方法(简称 LALR),它的功能和代价处于另外两者之间。LALR 方法可用于大多数编程语言的文法,可以高效地实现。最后例举非 LR 的上下文无关文法。

3.5.1 LR 分析算法

LR 分析器的模型见图 3.11,它包括输入、输出、栈、驱动程序和含**动作**和**转移**两部分的分析表。驱动程序对所有的 LR 分析方法都一样,用不同的分析方法构造的分析表不同。驱动程序每次从输入缓冲区读一个符号,它使用栈存储形式为 $s_0X_1s_1X_2s_2\cdots X_ms_m$ 的串,s_m 在栈顶。X_i 是文法符号,s_i 是叫做**状态的**符号,状态符号概括了栈中它下面部分所含的信息。栈顶的状态符号和

当前的输入符号用来检索分析表,以决定移进-归约分析的动作。真正实现时,文法符号不必出现在栈里,不过本书的讨论基本上总是包含它们以帮助解释 LR 分析的行为。

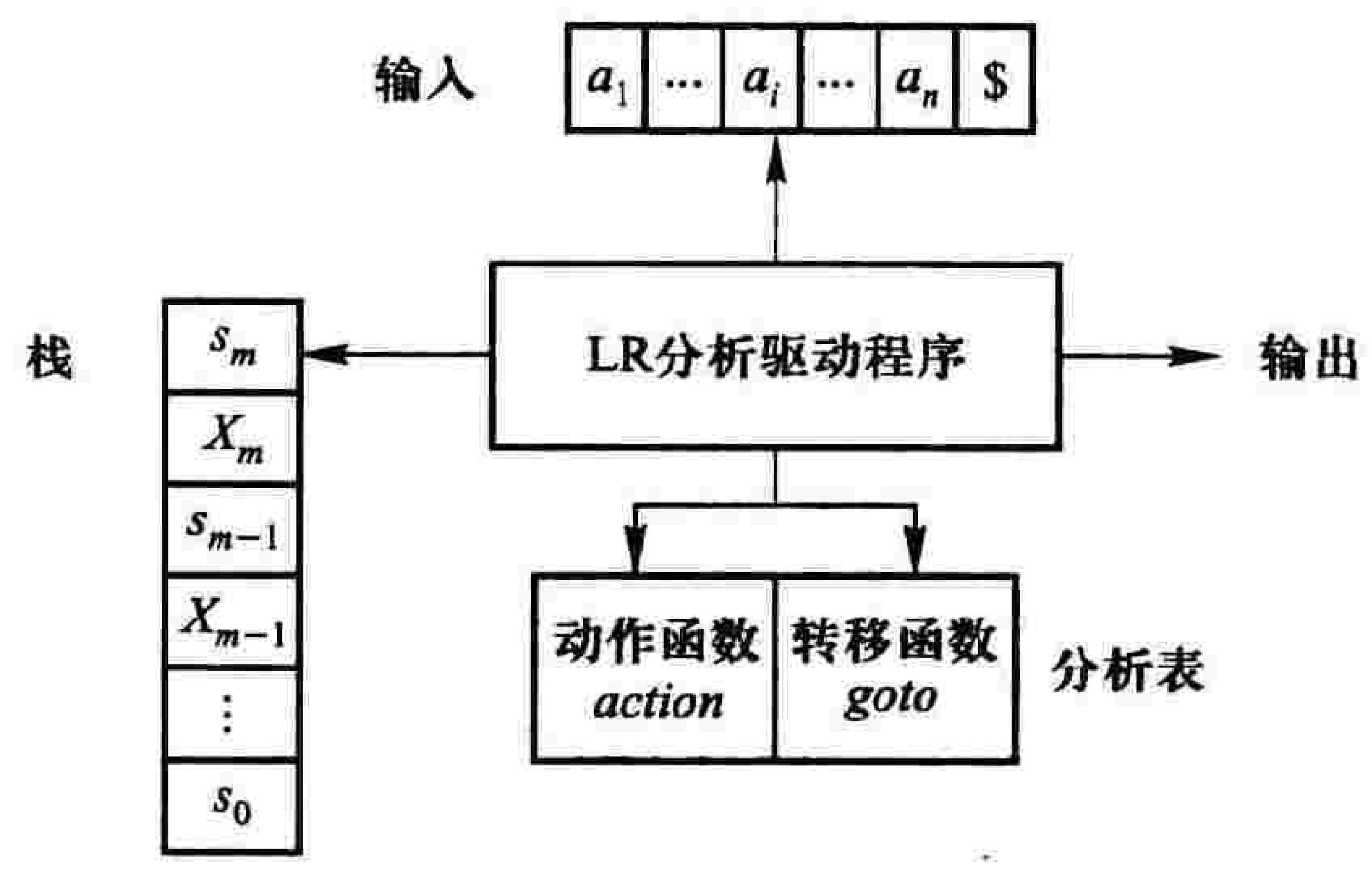


图 3.11 LR 分析器的模型

分析表由两部分组成:动作函数 *action* 和转移函数 *goto*。LR 分析驱动程序的行为是:它根据栈顶当前的状态 s_m 和当前的输入符号 a_i,访问 $action[s_m, a_i]$,它可能的 4 种值如下:

(1) 移进 s,其中 s 是一个状态。

(2) 按文法产生式 $A \to \beta$ 归约。

(3) 接受。

(4) 出错。

转移函数 *goto* 取状态和文法符号作为变元,产生一个状态。

从 3.5.3、3.5.4 和 3.5.5 节介绍的分析表构造方法可以看出,用 SLR、规范的 LR 和 LALR 方法从文法 G 构造的分析表的 *goto* 函数,都是识别 G **活前缀**的确定有限自动机的转换函数。这个 DFA 的开始状态是初始时置于 LR 分析器栈中的状态。

文法 G 的活前缀是它的右句型的前缀,该前缀不超过该右句型的最右句柄的右端。由这个定义可知,在活前缀的右端加上一些终结符后可以使它成为右句型,因此只要输入串的已扫描部分可以归约成一个活前缀,那就意味着已经扫描的部分没有错误。LR 分析栈中的文法符号总是构成活前缀。

LR 分析器的**格局**是二元组,它的第一个成分是栈的内容,第二个成分是尚未扫描的输入。

$$(s_0X_1s_1X_2s_2\cdots X_ms_m,\ a_ia_{i+1}\cdots a_n\$)$$

这个格局代表右句型

$$X_1X_2\cdots X_ma_ia_{i+1}\cdots a_n$$

从这里可以看出,它本质上和一般的移进-归约分析器一样,只有栈中的状态是新出现的。

分析器的下一个动作是用当前输入符号 a_i 和栈顶状态 s_m 访问分析表条目 $action[s_m, a_i]$,4 种不同的动作引起的格局变化如下。

(1) 如果 $action[s_m, a_i]$ = 移进 s,则分析器执行移进动作,进入格局

$$(s_0X_1s_1X_2s_2\cdots X_ms_ma_is,\ a_{i+1}\cdots a_n\$)$$

即分析器把当前输入符号 a_i 和下一个状态 s 移进栈，a_{i+1} 成为当前输入符号。

(2) 如果 $action[s_m, a_i]$ = 归约 $A \to \beta$，则分析器执行归约动作，进入格局

$$(s_0 X_1 s_1 X_2 s_2 \cdots X_{m-r} s_{m-r} A s, \quad a_i a_{i+1} \cdots a_n \$)$$

其中 r 是 β 的长度，$s = goto[s_{m-r}, A]$。这里，分析器首先从栈中弹出 $2r$ 个符号，即 r 个状态符号和 r 个文法符号，这些文法符号刚好匹配产生式右部 β，这时栈顶暴露出状态 s_{m-r}。然后把产生式左边的符号 A 和 $goto[s_{m-r}, A]$ 的状态 s 压入栈。在归约动作时，当前输入符号没有改变。

LR 分析器的输出由归约时执行与归约产生式有关的语义动作来产生，现在暂且认为输出就是打印归约产生式。

(3) 如果 $action[s_m, a_i]$ = 接受，则分析完成。

(4) 如果 $action[s_m, a_i]$ = 出错，则分析器发现错误，调用错误恢复例程。

LR 分析算法总结在算法 3.3 中，所有 LR 分析器的行为都遵从该算法，唯一的区别是分析表的内容不一样。

算法 3.3 LR 分析算法。

输入 输入串 w 和文法 G 的 LR 分析表。

输出 若 $w \in L(G)$，则输出 w 自下而上分析的归约步骤，否则报错。

方法 分析器初始情况是：初始状态 s_0 在分析器的栈顶，$w\$$ 在输入缓冲区。然后分析器执行图 3.12 的程序。 □

```
令 a 是 w$ 的第一个符号;
while(1)    /* 始终重复 */
    令 s 是栈顶的状态;
    if (action[s, a] == 移进 t ) {
        把 a 和 t 依次压入栈;
        令 a 是下一个输入符号;
    } else if (action[s, a] == 归约 A → β ) {
        栈顶退掉 2 * |β| 个符号;
        令 t 是现在的栈顶状态;
        把 A 和 goto[t, A] 压入栈;
        输出产生式 A→ β;
    } else if (action[s, a] == 接受) break;/* 分析完成 */
    else 调用错误恢复例程;
}
```

图 3.12 LR 分析程序

例 3.22 表 3.7 给出一个算术表达式文法的 LR 分析表，该算术表达式的文法如下：

(1) $E \to E+T$ (4) $T \to F$

(2) $E \to T$ (5) $F \to (E)$

(3) $T \to T * F$ (6) $F \to \mathbf{id}$

动作表中各类动作的含义是：

(1) *si* 表示移进，把当前输入符号和状态 *i* 压进栈；

(2) *rj* 表示按第 *j* 个产生式进行归约；

(3) *acc* 表示接受；

(4) 空白表示出错。

表 3.7 表达式文法的分析表

状态	动作						转移		
	id	+	*	(	)	$	*E*	*T*	*F*
0	*s*5			*s*4			1	2	3
1		*s*6				*acc*			
2		*r*2	*s*7		*r*2	*r*2			
3		*r*4	*r*4		*r*4	*r*4			
4	*s*5			*s*4			8	2	3
5		*r*6	*r*6		*r*6	*r*6			
6	*s*5			*s*4				9	3
7	*s*5			*s*4					10
8		*s*6			*s*11				
9		*r*1	*s*7		*r*1	*r*1			
10		*r*3	*r*3		*r*3	*r*3			
11		*r*5	*r*5		*r*5	*r*5			

注意，对于终结符 *a*，状态转移动作可以在动作表的条目 *action*[*s*, *a*]中找到，所以转移表中仅给出非终结符 *A* 的 *goto*[*s*, *A*]。

面对输入 **id** * **id**+**id**，栈内容和输入内容的变化序列在表 3.8 中给出。例如，在第一行，LR 分析器处于状态 0，当前输入符号是 **id**。表 3.7 的第 0 行和 **id** 列的动作是 *s*5，它的含义是移进 **id**，再把状态 5 压进栈，如表 3.8 第二行所示。然后，* 成为当前输入符号，状态 5 面对输入 * 的动作是按 $F \rightarrow$ **id** 归约。这时两个符号（状态符号 5 和文法符号 **id**）弹出栈，状态 0 显露出来。因为 *goto*[0, *F*]是 3，因此把 *F* 和 3 压进栈，到达第三行所示的格局。剩余的动作也类似地决定。 □

表 3.8 LR 分析器对于输入 $\mathbf{id}_1 * \mathbf{id}_2 + \mathbf{id}_3$ 的格局变化和相应动作

栈	输入	动作
0	**id** * **id** + **id** $	移进
0 **id** 5	* **id** + **id** $	按 $F \to$ **id** 归约
0 F 3	* **id** + **id** $	按 $T \to F$ 归约
0 T 2	* **id** + **id** $	移进
0 T 2 * 7	**id** + **id** $	移进
0 T 2 * 7 **id** 5	+ **id** $	按 $F \to$ **id** 归约
0 T 2 * 7 F 10	+ **id** $	按 $T \to T * F$ 归约
0 T 2	+ **id** $	按 $E \to T$ 归约
0 E 1	+ **id** $	移进
0 E 1 + 6	**id** $	移进
0 E 1 + 6 **id** 5	$	按 $F \to$ **id** 归约
0 E 1 + 6 F 3	$	按 $T \to F$ 归约
0 E 1 + 6 T 9	$	按 $E \to E + T$ 归约
0 E 1	$	接受

3.5.2 LR 文法和 LR 分析方法的特点

一个文法,如果能为它构造出所有条目都唯一的 LR 分析表,就说它是 **LR 文法**。直观上说,当句柄出现在栈顶时,如果自左向右扫描的移进-归约分析器能及时识别它,那么文法一定是 LR 的。

LR 分析器不需要扫描整个栈就可以知道句柄是否出现在栈顶,因为栈顶的状态符号包含了确定句柄所需要的一切信息。学习了下面的分析表构造方法后就不难理解这句话了。能够用来帮助 LR 分析器作出移进-归约决策的另一个信息源是剩余输入的前 k 个符号,最感兴趣的是 $k=0$ 或 $k=1$ 的情况,并且仅讨论 $k \leq 1$ 的情况。例如,表 3.7 仅向前搜索一个符号。如果最多向前搜索 k 个符号就可以决定动作的话,那么用这样的 LR 分析器所分析的文法称为 **LR(k) 文法**。

LR 分析器富有吸引力的原因如下。

(1) LR 分析器能够被构造来识别所有能用上下文无关文法写出的编程语言构造。3.5.6 节讨论非 LR 的上下文无关文法,典型的编程语言构造都能避免出现这种情况。

(2) LR 分析方法是已知的最一般的无回溯的移进-归约方法,它能和其他移进-归约方法一样有效地实现。

(3) LR 方法能分析的文法类是预测分析法或者说 LL 方法能分析的文法类的真超集。

(4) 在自左向右扫描输入的前提下,LR 分析器能尽可能快地发现语法错误。

LL方法和LR方法有明显的区别。LR(k)分析器必须在看见一个产生式的右部推出的所有东西并加上k个向前搜索符号后,才能识别该右部的出现。这个要求远不如LL(k)那样严峻,LL(k)文法要求在看见一个产生式右部推出的前k个符号时就确定使用该产生式。所以LR文法比LL文法能描述更多的语言。

例3.23 现有句型$\gamma l\beta bw$,其规范推导是$S \Rightarrow_{rm} \cdots \Rightarrow_{rm} \gamma Abw \Rightarrow_{rm} \gamma l\beta bw$,最后一步用的产生式是$A \rightarrow l\beta$。LL(1)方法在看见右部$l\beta$的第一个符号$l$时就必须决定用这个产生式,而LR(1)方法在看见右部$l\beta$的后继符号b时决定用这个产生式,见图3.13。显然,和LL(1)方法相比,LR(1)方法是在掌握了更多的信息后才决定用哪个产生式,因此能力较强。 □

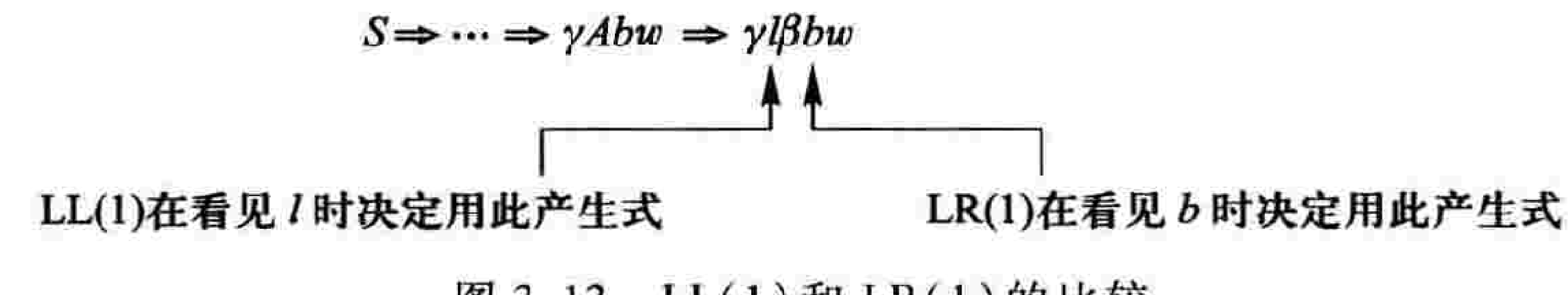


图3.13 LL(1)和LR(1)的比较

LR方法的主要缺点是,对真正的编程语言文法,手工构造LR分析器的工作量太大,因而需要专门的LR分析器的生成器。幸好有很多这样的工具可用,本章最后将介绍其中之一Yacc的设计和使用。这些生成器自动为上下文无关文法产生分析器。如果文法含二义的或者很难通过从左到右扫描进行分析的构造,这些生成器能定位这些构造,并提供详细的诊断信息。

3.5.3 构造SLR分析表

本小节开始介绍怎样从文法构造LR分析表,一共给出三种方法,它们的功能和实现的难易程度不同。第一种叫做简单的LR方法,简称SLR方法,它功能最弱但最易实现,根据这种方法构造的分析表叫做SLR分析表,使用SLR分析表的LR分析器叫做SLR分析器,能够为之构造SLR分析器的文法叫做SLR文法。另两种方法用更准确的向前搜索信息来增强SLR文法,所以SLR方法是研究LR分析的理想起点,了解了SLR方法也就了解了LR方法的基本思想。

文法G的**LR(0)项目**(简称**项目**)是在其右部的某个地方加点的产生式。如从产生式$A \rightarrow XYZ$可得到如下四个项目:

$$A \rightarrow \cdot XYZ \qquad A \rightarrow XY \cdot Z$$

$$A \rightarrow X \cdot YZ \qquad A \rightarrow XYZ \cdot$$

对于产生式$A \rightarrow \varepsilon$,只能得到一个项目$A \rightarrow \cdot$。直观地讲,项目表示在分析过程的某一点,已经看见了产生式的多大部分(点的左边部分)和下面期望看见的部分(点的右边部分)。例如,$A \rightarrow \cdot XYZ$表示期望下一步从输入中看见由XYZ推出的串,$A \rightarrow X \cdot YZ$表示刚从输入中看见了由X推出的串,下面期望看见由YZ推出的串。也可以这么说,点的左边代表历史信息,而右边代表展望信息。

SLR方法的主要思想是首先从文法构造识别文法活前缀的DFA。所有项目被按一定方法组

成集合，这些集合对应到 SLR 分析器的状态，这些状态也是这个 DFA 的状态。这样的一族项目集，称做 LR(0)项目集的规范族，它提供了构造 SLR 分析表的基础。为了构造文法的 LR(0)项目集规范族，先定义拓广文法及两个函数 *closure* 和 *goto*。

如果文法 G 的开始符号是 S，那么 G 的**拓广文法** G'是在 G 的基础上增加了新的开始符号 S' 和产生式 $S' \to S$。新增产生式的目的是用来指示分析器什么时候应该停止分析并宣布接受输入。也就是当且仅当分析器执行归约 $S' \to S$ 时，意味分析成功。

下面先讲闭包函数 *closure*。

如果 I 是文法 G 的一个项目集，那么 *closure*(I)是用下面两条规则从 I 构造的项目集：

(1) 初始时，I 的每个项目都加入 *closure*(I)。

(2) 如果 $A \to \alpha \cdot B\beta$ 在 *closure*(I)中，且 $B \to \gamma$ 是产生式，那么如果项目 $B \to \cdot \gamma$ 还不在 *closure*(I)中的话，则把它加入。运用这条规则，直到没有更多的项目可加入 *closure*(I)为止。

直观上，*closure*(I)中的 $A \to \alpha \cdot B\beta$ 表示在分析过程的某一点，下一步从输入中看见的可能是由 $B\beta$ 推出的串。因为 $B \to \gamma$ 是产生式，也可以认为从输入中首先可能看见由 γ 推出的子串，出于这个原因，把 $B \to \cdot \gamma$ 加入 *closure*(I)。

例 3.24 考虑拓广的表达式文法

$$E' \to E \qquad T \to T * F \mid F$$

$$E \to E + T \mid T \qquad F \to (\ E\ ) \mid \mathbf{id} \tag{3.11}$$

如果 I 是项目集 $\{[E' \to \cdot E]\}$，那么 *closure*(I)含下列项目：

$$E' \to \cdot E \qquad T \to \cdot F$$

$$E \to \cdot E + T \qquad F \to \cdot (E)$$

$$E \to \cdot T \qquad F \to \cdot\ \mathbf{id}$$

$$T \to \cdot T * F$$

这里，由规则(1)，$E' \to \cdot E$ 被加入 *closure*(I)。因为 E 紧挨在点的右侧，由规则(2)，把对应 E 产生式的、点在左端的项目 $E \to \cdot E + T$ 和 $E \to \cdot T$ 都加入。同样道理，把 $T \to \cdot T * F$ 和 $T \to \cdot F$ 都加入。最后，再把 $F \to \cdot (E)$ 和 $F \to \cdot\ \mathbf{id}$ 也都加入。再没有其他项目可由规则(2)加入。 □

函数 *closure* 可以如图 3.14 那样计算。

```
setOfItems closure(I) {
    repeat
        for (I 的每个项目 A → α · Bβ )
            for (G 的每个产生式 B →γ )
                if (B → · γ 不在 I 中)
                    把 B → · γ 加入 I;
    until 在一次重复中没有项目加入 I;
    return I;
}
```

图 3.14 *closure* 的计算

从计算 *closure* 的算法知道，如果 B 在点右邻的一个项目加入 *closure*(I)，那么与 B 各产生式对应且点在左端的项目都加入该闭包。因此，在某些场合下，并不需要实际列出由闭包运算加入的所有项目 $B \rightarrow \cdot\gamma$，只要列出这样的非终结符 B 就足够了。项目集的项目可以分成两类：

（1）核心项目：它包括初始项目 $S' \rightarrow \cdot S$ 和所有那些点不在左端的项目。

（2）非核心项目：除了项目 $S' \rightarrow \cdot S$ 以外，所有点在左端的项目。

如果想用较少的空间来存储项目集，则可以扔掉项目集中所有的非核心项目，因为非核心项目可以由闭包过程重新生成。

下面再看转移函数 *goto*(I, X)，其中 I 是项目集，X 是文法符号。*goto*(I, X) 的定义是，满足 $[A \rightarrow \alpha \cdot X\beta]$ 属于 I 的所有项目 $[A \rightarrow \alpha X \cdot \beta]$ 的集合的闭包。直观上讲，如果 I 是对某个活前缀 γ 有效的项目集，那么 *goto*(I, X) 是对活前缀 γX 有效的项目集。

例 3.25 如果 I 是项目集 $\{[E' \rightarrow E \cdot], [E \rightarrow E \cdot + T]\}$，那么 *goto*$(I, +)$ 包括项目：

$E \rightarrow E + \cdot T$ $\qquad$ $F \rightarrow \cdot (E)$

$T \rightarrow \cdot T * F$ $\qquad$ $F \rightarrow \cdot \mathbf{id}$

$T \rightarrow \cdot F$

这是通过考察 I 的项目，看+是否紧挨在点的右边来计算 *goto*$(I, +)$ 的。$E' \rightarrow E \cdot$ 不是这样的项目，但是 $E \rightarrow E \cdot + T$ 是。把点移过+得到 $\{[E \rightarrow E + \cdot T]\}$，然后求它的闭包。 □

现在可以构造拓广文法 G' 的 LR(0) 项目集的规范族了，算法如图 3.15 所示。

```
void items(G') {
    C = {closure ({[S'→ · S ]})};
    repeat
        for(C 的每个项目集 I )
            for(每个文法符号 X)
                if(goto (I, X)非空并且不在 C 中)
                    把 goto (I, X)加入 C 中;
    until  在一次重复中没有新的项目加入 C;
}
```

图 3.15 LR(0)项目集规范族的计算

例 3.26 例 3.24 文法(3.11)的 LR(0)项目集规范族见图 3.16，这些项目集的 *goto* 函数以及接受活前缀的确定有限自动机 D 的转换图显示在图 3.17 中。 □

如果图 3.17 中 D 的每个状态都是接受状态且 I_0 是初态，那么 D 识别的正是文法(3.11)的活前缀。这绝不是偶然的，可以证明，对每个文法 G，项目集规范族的 *goto* 函数定义了一个 DFA，它识别 G 的活前缀。

事实上，也可以构造一个识别活前缀的 NFA N，它的状态就是项目本身，从 $A \rightarrow \alpha \cdot X\beta$ 到 $A \rightarrow \alpha X \cdot \beta$ 有转换标记 X，从 $A \rightarrow \alpha \cdot B\beta$ 到 $B \rightarrow \cdot\gamma$ 有转换标记 ε。项目集（也就是 N 的状态集）I

$I_0: E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

$I_1: E' \to E \cdot$
$E \to E \cdot + T$

$I_2: E \to T \cdot$
$T \to T \cdot * F$

$I_3: T \to F \cdot$

$I_4: F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

$I_5: F \to \mathbf{id} \cdot$

$I_6: E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

$I_7: T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathbf{id}$

$I_8: F \to (E \cdot)$
$E \to E \cdot + T$

$I_9: E \to E + T \cdot$
$T \to T \cdot * F$

$I_{10}: T \to T * F \cdot$

$I_{11}: F \to (E) \cdot$

图 3.16 文法(3.11)的 LR(0)项目集规范族

的 *closure*(I)正好是 2.3 节定义的 NFA 状态集的 ε 闭包。若从 N 用子集构造法构造 DFA,那么 *goto*(I, X)正是这个 DFA 的 *move*(I, X)。按照这种观点,图 3.15 的 *items*(G')正是子集构造法本身运用于从 G'构造的这个 NFA N。

如果 $S' \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta_1\beta_2 w$,则说项目 $A \to \beta_1 \cdot \beta_2$对活前缀 $\alpha\beta_1$是**有效的**。因为该项目的期望部分是β_2,在活前缀 $\alpha\beta_1$右边添上β_2后仍然构成活前缀,因此说该项目对 $\alpha\beta_1$有效。一般而言,一个项目可能对好几个活前缀都是有效的。对任何活前缀 $\alpha\beta_1$,从项目 $A \to \beta_1 \cdot \beta_2$ 有效这个事实可以知道,当 $\alpha\beta_1$ 在分析栈的栈顶时,分析器应该执行移进还是归约。如果 $\beta_2 \neq \varepsilon$,那么它暗示句柄还没有完全进栈,应该移进;如果 $\beta_2 = \varepsilon$,那么 β_1是句柄,应该按产生式 $A \to \beta_1$归约。

另一方面,一个活前缀可能有多个有效项目。一个活前缀 γ 的有效项目集就是从上述 DFA 的初态出发,沿着标记为 γ 的路径到达的那个项目集(状态),这是 LR 分析理论的一条基本定理。这样,当活前缀 γ 在栈顶时,其有效项目集(也就是栈顶的状态)概括了所有可以从栈中收集到的有用信息。在此不打算证明这个定理,而是用例子来阐明。两个不同的有效项目可能要

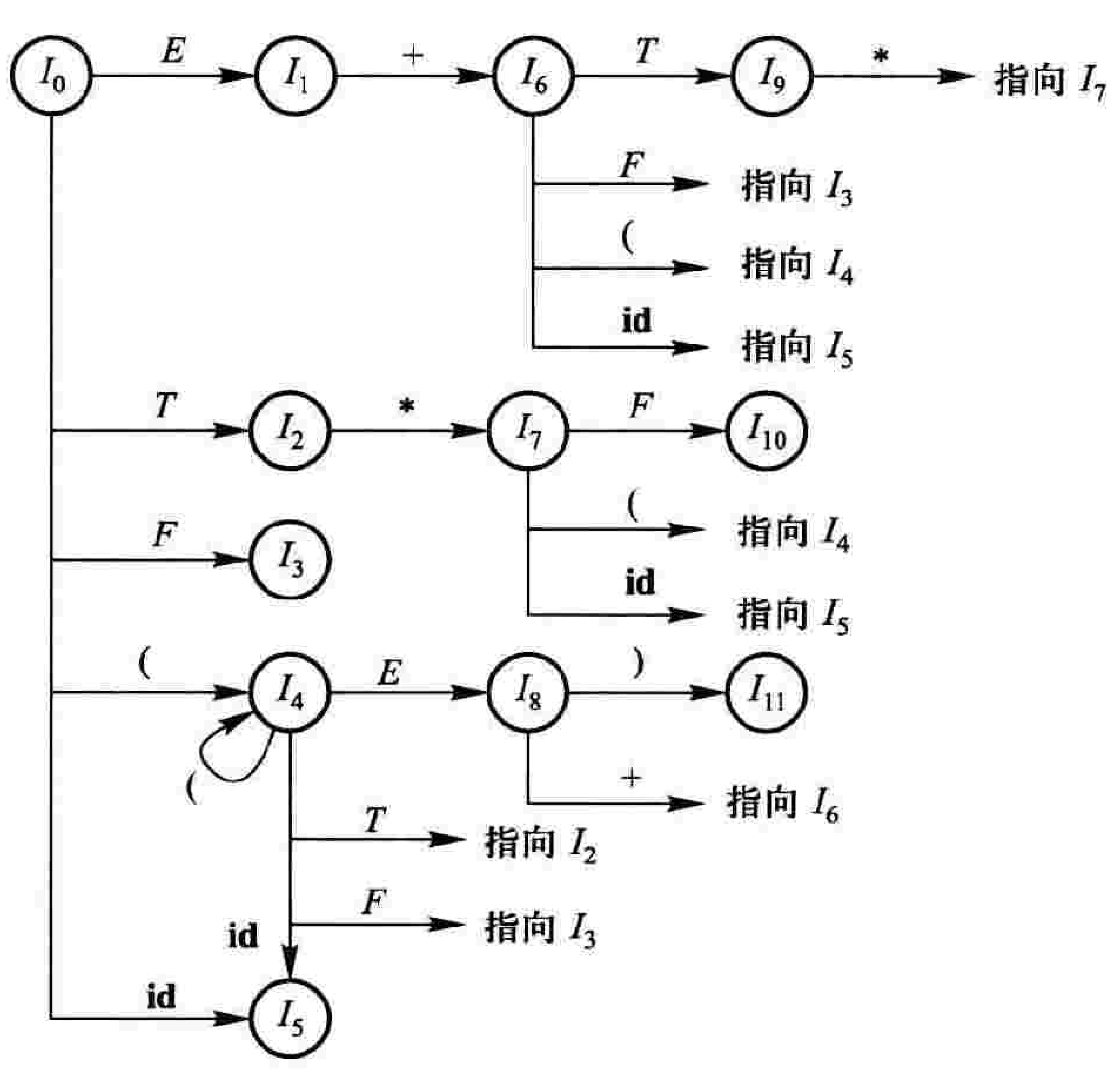


图 3.17 接受活前缀的 DFA D 的转换图

求分析器采取不同的动作，对这样的冲突其中一部分可以由向前搜索下一个输入符号来解决，还有一些可以由下一小节介绍的方法解决。不过，当 LR 方法用于构造任意文法的分析表时，不要期望所有分析动作的冲突都可以解决。

例 3.27 再次考察文法(3.11)，它的项目集和 *goto* 函数显示在图 3.16 和图 3.17 中。显然，串 $E+T*$ 是(3.11)的活前缀。在读完 $E+T*$ 后，图 3.17 的自动机处于状态 I_7，它包含项目

$T \to T * \cdot F$

$F \to \cdot (E)$

$F \to \cdot \mathbf{id}$

它们都对 $E + T*$ 有效，下面三个最右推导明白无误地说明了这一点。

$E' \Rightarrow E$	$E' \Rightarrow E$	$E' \Rightarrow E$
$\Rightarrow E+T$	$\Rightarrow E+T$	$\Rightarrow E+T$
$\Rightarrow E+T*F$	$\Rightarrow E+T*F$	$\Rightarrow E+T*F$
$\Rightarrow E+T*\mathbf{id}$	$\Rightarrow E+T*(E)$	$\Rightarrow E+T*\mathbf{id}$
$\Rightarrow E+T*F*\mathbf{id}$		

这三个推导分别展示了 $T \to T * \cdot F$、$F \to \cdot (E)$ 和 $F \to \cdot \mathbf{id}$ 的有效性。可以证明，不存在对 $E+T*$ 有效的其他项目。 □

以上完成了构造 SLR 分析表的第一步，即从文法构造识别活前缀的 DFA。现在进行第二步，怎样从识别活前缀的 DFA 构造 SLR 分析表的动作函数和转移函数。该算法不可能为任何文

法都产生 SLR 分析表,即所有条目内容都唯一的分析表,但是它对许多编程语言的文法是成功的。给定文法 G,从拓广文法 G'构造了项目集规范族 C 后,使用算法 3.4 可以构造分析表的动作函数 *action* 和转移函数 *goto*。这个算法需要知道每个非终结符的后继符号集合(见 3.3 节)。

算法 3.4 构造 SLR 分析表。

输入 拓广文法 G'。

输出 G'的 SLR 分析表的 *action* 函数(动作表)和 *goto* 函数(转移表)。

方法 (1) 构造 G'的 LR(0)项目集规范族 $C = \{I_0, I_1, \cdots, I_n\}$。

(2) 状态 i 从 I_i构造。*action* 函数在状态 i 的值按如下方法确定:

(a) 如果$[A\to\alpha \cdot a\beta]$在 I_i中,并且 $goto(I_i, a) = I_j$,那么置 $action[i, a]$为 sj,其含义是把 a 和状态 j 移进栈。此时 a 必须是终结符。

(b) 如果$[A\to\alpha \cdot]$在 I_i中,那么对 $FOLLOW(A)$中的所有 a,置 $action[i, a]$为 rj, j 是产生式 $A\to\alpha$ 的编号。这个动作的意思是按产生式 $A\to\alpha$ 归约。此时 A 不可以是 S'。

(c) 如果$[S'\to S \cdot]$在 I_i中,那么置 $action[i, \$]$为接受 acc。

如果由上面规则产生的动作有冲突,那么该文法就不是 SLR(1)的。在这种情况下,该算法流产,不产生分析器。

(3) 使用下面规则构造 *goto* 函数在状态 i 的值:

对所有的非终结符 A,如果 $goto(I_i, A) = I_j$, 那么 $goto[i, A] = j$。

(4) 不能由规则(2)和(3)定义的条目都是空白,表示出错。

(5) 分析器的初始状态是包含$[S'\to \cdot S]$的项目集对应的状态。 □

由算法 3.4 决定的动作函数和转移函数组成的分析表叫做文法 G 的 SLR(1)分析表,使用 G 的 SLR(1)分析表的 LR 分析器叫做 G 的 SLR(1)分析器,有 SLR(1)分析表的文法叫做 SLR(1)文法。通常省略 SLR 后面的(1),因为本书不讨论向前搜索两个或多个符号的分析器。

例 3.28 为文法(3.11)构造 SLR 分析表,它的 LR(0)项目集规范族已在图 3.16 给出。首先考虑项目集 I_0:

$$E'\to \cdot E \qquad T\to \cdot F$$
$$E\to \cdot E+T \qquad F\to \cdot (E)$$
$$E\to \cdot T \qquad F\to \cdot \mathbf{id}$$
$$T\to \cdot T * F$$

项目 $F\to \cdot (E)$使得条目 $action[0, (] = s4$, 项目 $F\to \cdot \mathbf{id}$ 使得条目 $action[0, \mathbf{id}] = s5$。$I_0$的其他项目不产生动作。现在考虑项目集 I_1:

$$E'\to E \cdot$$
$$E\to E \cdot +T$$

第一个项目使得 $action[1, \$] = acc$, 第二个项目使得 $action[1, +] = s6$。再考虑 I_2:

$$E\to T \cdot$$
$$T\to T \cdot * F$$

因为 $FOLLOW(E)=\{\$,+,)\}$，因此第一个项目使得 $action[2,\$]=action[2,+]=action[2,)]=r2$（因为 $E \rightarrow T$ 的序号为2）。第二个项目使得 $action[2,*]=s7$。以这种方法继续下去可得到表3.7的动作表和转移表。 □

每个SLR(1)文法都不是二义的，但是有很多非二义的文法，它们都不是SLR(1)的，这说明SLR(1)文法的描述能力有限。

例3.29 看下面文法：

$$
\begin{aligned}
&S \rightarrow V = E \\
&S \rightarrow E \\
&V \rightarrow * E \\
&V \rightarrow \mathbf{id} \\
&E \rightarrow V
\end{aligned}
\qquad (3.12)
$$

可以把 V 和 E 想象成分别代表左值和右值，左值表示一个存储单元，右值是一个可存储的值。* 是表示"取单元内容"的算符。文法(3.12)的LR(0)项目集规范族如图3.18所示。

$I_0: S' \rightarrow \cdot S$
　$S \rightarrow \cdot V=E$
　$S \rightarrow \cdot E$
　$V \rightarrow \cdot * E$
　$V \rightarrow \cdot \mathbf{id}$
　$E \rightarrow \cdot V$

$I_1: S' \rightarrow S \cdot$

$I_2: S \rightarrow V \cdot =E$
　$E \rightarrow V \cdot$

$I_3: S \rightarrow E \cdot$

$I_4: V \rightarrow * \cdot E$
　$E \rightarrow \cdot V$
　$V \rightarrow \cdot * E$
　$V \rightarrow \cdot \mathbf{id}$

$I_5: V \rightarrow \mathbf{id} \cdot$

$I_6: S \rightarrow V= \cdot E$
　$E \rightarrow \cdot V$
　$V \rightarrow \cdot * E$
　$V \rightarrow \cdot \mathbf{id}$

$I_7: V \rightarrow * E \cdot$

$I_8: E \rightarrow V \cdot$

$I_9: S \rightarrow V=E \cdot$

图3.18 文法(3.12)的规范LR(0)族

考察项目集 I_2，这个集合的第一项目使得 $action[2,=]=s6$。因为 $FOLLOW(E)$ 包含=（因为 $S \Rightarrow V=E \Rightarrow *E=E$），第二项目使得 $action[2,=]$ 为按 $E \rightarrow V$ 归约。这样，条目 $action[2,=]$ 是多重定义的条目，它既含有移进条目又含有归约条目，因此状态2在输入符号是=时有移进-归约冲突。

文法(3.12)不是二义的，项目集 I_2 的冲突应该是可以避免的。=虽然是 E 的一个后继符号，但通过 $S \Rightarrow V = E \Rightarrow *E = E$ 等推导来考察=作为 E 后继符号的场合可以知道，=不是而只有 \$ 是 $E \to V$ 归约时的后继符号。这说明 SLR 方法在把项目集 I_2 的归约项目填入分析表时，掌握的信息不够精确。改进的办法是增加状态，使每个状态比 SLR 的状态包含更准确的信息。这就是下面要讨论的规范 LR 方法。一般来说，规范 LR 方法导致状态数大大增加，而 LALR 方法是通过损失一点精度来达到减少状态数的目的。 □

3.5.4 构造规范的 LR 分析表

回顾上面讲的 SLR 方法，如果 I_i 包含项目 $[A \to \alpha \cdot]$ 且 a 在 $FOLLOW(A)$ 中，那么状态 i 要求面临 a 时按 $A \to \alpha$ 归约。但是在有些场合下，当状态 i 出现在栈顶、活前缀 $\beta\alpha$ 在栈中并且 a 是当前输入符号时，用 $A \to \alpha$ 来归约却不合适，因为在任何右句型中，a 不可能跟随在 βA 的后面。

例 3.30 再来看例 3.29。状态 2 有项目 $E \to V \cdot$，它对应上面的 $A \to \alpha \cdot$，=对应上面的 a，它在 $FOLLOW(E)$ 中。于是 SLR 分析器在状态 2 且面临输入=时，要求按 $E \to V$ 归约。但是项目 $S \to V \cdot = E$ 也在状态 2 中，因此分析器在状态 2 且面临=时也要求移进。然而，例 3.29 的文法并不存在以 $E = \cdots$ 开始的右句型，因此，分析器此时不应该把 V 归约成 E。 □

让状态含有更多的信息，使之能够剔除上述那些无效归约是完全可能的。可以设想，必要时，对状态进一步细分，使得 LR 分析器的每个状态能够确切地指出，在 α 之后出现哪些终结符时才允许把 α 归约为 A。

重新定义项目，使之包含一个终结符作为第二个成分，这样就把更多的信息纳入了状态。项目的一般形式也就成为 $[A \to \alpha \cdot \beta,\ a]$，其中 $A \to \alpha\beta$ 是产生式，a 是终结符号或 \$，这种项目叫做**LR(1)项目**，1 是第二个成分的长度，该成分叫做项目的**搜索符**。搜索符对 β 非空的项目 $[A \to \alpha \cdot \beta,\ a]$ 是不起什么作用的；但形式为 $[A \to \alpha \cdot,\ a]$ 的项目要求，只有下一个输入符号是 a 时，才按 $A \to \alpha$ 归约。因此，分析器只有在输入符号是 a 时才按 $A \to \alpha$ 归约，其中 $[A \to \alpha \cdot,\ a]$ 在栈顶状态的LR(1)项目集中。这样的 a 的集合是 $FOLLOW(A)$ 的子集，完全可能是真子集，如例3.30那样。

形式地说，LR(1)项目 $[A \to \alpha \cdot \beta,\ a]$ 对活前缀 γ **有效**，如果存在着推导 $S \Rightarrow^*_{rm} \delta A w \Rightarrow_{rm} \delta\alpha\beta w$，其中：

(1) $\gamma = \delta\alpha$；

(2) a 是 w 的第一个符号，或者 w 是 ε 且 a 是 \$。

例 3.31 考虑文法

$$S \to BB$$
$$B \to bB \mid a$$

它有一个最右推导 $S \Rightarrow^*_{rm} bbBba \Rightarrow_{rm} bbbBba$。可以看出，项目 $[B \to b \cdot B,\ b]$ 对活前缀 $\gamma = bbb$ 是有效的。根据上面定义，只需令 $\delta = bb$，$A = B$，$w = ba$，$\alpha = b$ 且 $\beta = B$ 即可。

再看该文法的另一个最右推导 $S \Rightarrow^*_{rm} BbB \Rightarrow_{rm} BbbB$，可以看出项目 $[B \to b \cdot B,\ \$]$ 对活前缀 Bbb 是有效的。 □

构造 LR(1)项目集规范族的方法本质上和构造 LR(0)项目集规范族的方法是一样的,只需要修改 *closure* 函数和 *goto* 函数。

为了理解 *closure* 运算的新定义,考虑对活前缀 γ 有效的项目集中的项目 $[A \to \alpha \cdot B\beta,\ a]$。该文法必定存在一个最右推导 $S \Rightarrow_{rm}^{*} \delta Aax \Rightarrow_{rm} \delta\alpha B\beta ax$,其中 $\gamma = \delta\alpha$。假定 βax 推出终结符串 by,那么对每个形式为 $B \to \eta$ 的产生式,有推导 $S \Rightarrow_{rm}^{*} \gamma Bby \Rightarrow_{rm} \gamma\eta by$,于是 $[B \to \cdot\ \eta,\ b]$ 对 γ 有效。注意,b 可能是从 β 推出的第一个终结符,或者在推导 $\beta ax \Rightarrow^{*} by$ 中,β 推出 ε,b 就成了 a, 总结这两种可能性,可以说 b 是 $FIRST(\beta ax)$ 中的任何终结符。注意,x 不可能含 by 的第一个终结符,所以 $FIRST(\beta ax) = FIRST(\beta a)$。下面给出 LR(1)项目集的构造算法。

算法 3.5 构造 LR(1)项目集。

输入 拓广文法 G'。

输出 LR(1)项目集,其中每个项目集都是对 G' 的若干个活前缀有效的项目集。

方法 构造项目集的函数 *closure* 和 *goto* 及主例程 *items* 列在图 3.19 中。 □

```
setOfItems closure (I) {
    repeat
        for(I 的每个项目[A→α · Bβ, a])
            for(G′中的每个产生式 B→γ)
                for(FIRST(βa)的每个终结符 b)
                    if([B→ · γ, b]不在 I 中)
                        把[B→ · γ, b]加到 I;
    until 在一次重复中没有项目加入 I;
    return I;
}

setOfItems goto(I, X) {
    置 J 为空集;
    for(I 中的每个项目[A→α · Xβ, a])
        把项目[A→α X · β, a]加到集合 J;
    return closure(J);
}

void items(G′) {
    置 C 的初值为{closure({[S′ → · S, $]})};
    repeat
        for(C 的每个项目集 I)
            for(每个文法符号 X)
                if(goto(I, X)非空且不在 C 中)
                    把 goto(I, X )加入 C 中;
    until 在一次重复中没有项目集加入 C 中;
}
```

图 3.19 构造文法 G' 的 LR(1)项目集

例 3.32 拓广文法

$$S' \to S$$
$$S \to BB \tag{3.13}$$
$$B \to bB \mid a$$

的 LR(1)项目集规范族见图 3.20,*goto* 函数已在图中给出。其中,$[B \to \cdot bB,\ b/a]$是两个项目$[B \to \cdot bB,\ b]$和$[B \to \cdot bB,\ a]$的缩写。 □

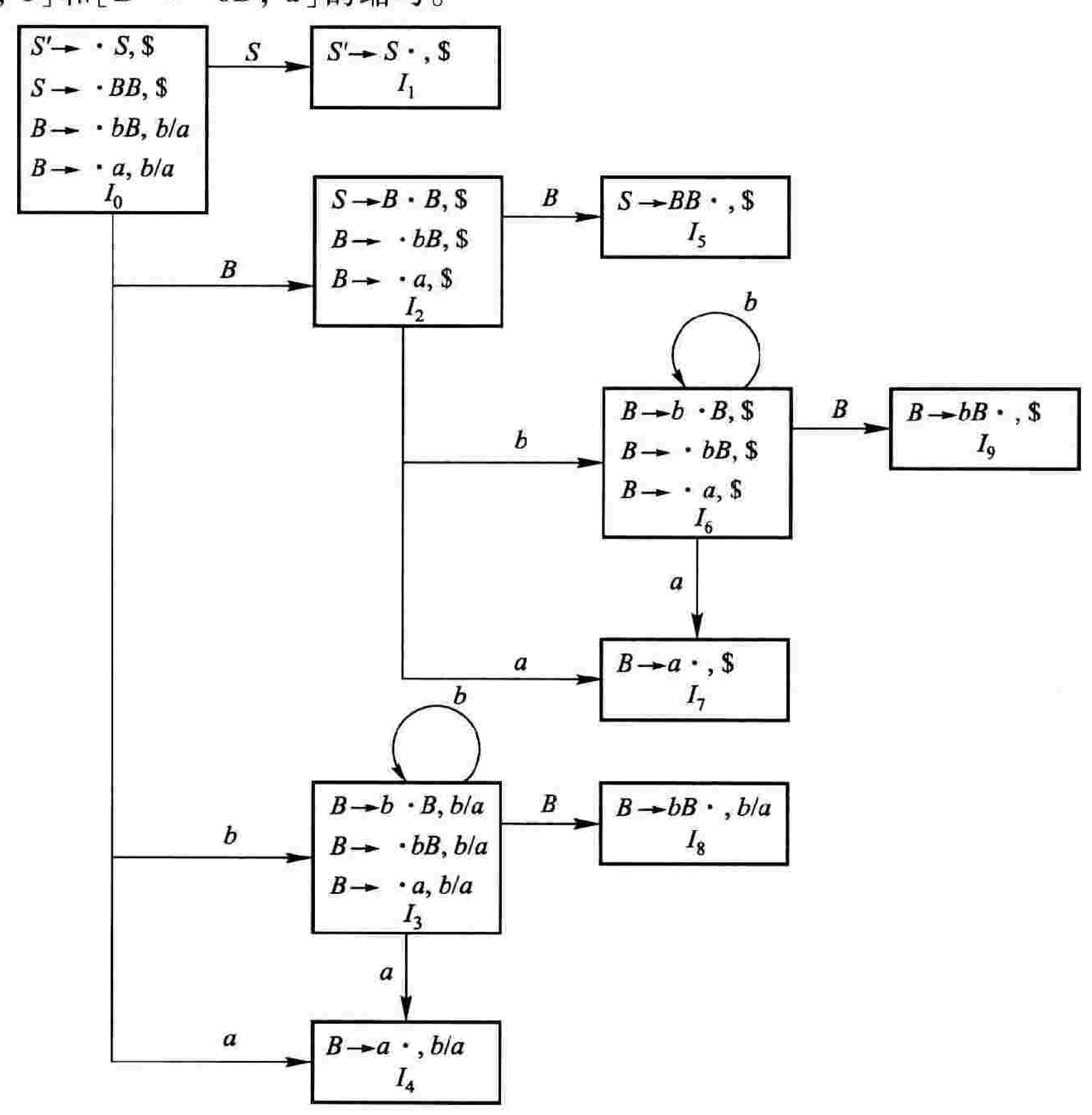


图 3.20 文法(3.13)的状态转换图

下面给出从 LR(1)的项目集构造 LR(1)分析的动作函数和转移函数的规则。

算法 3.6 构造规范的 LR 分析表。

输入 拓广文法 G'。

输出 文法 G'的规范 LR 分析的 *action* 函数和 *goto* 函数。

方法 (1) 构造 G'的 LR(1)项目集规范族 $C = \{I_0, I_1, \cdots, I_n\}$。

(2) 从 I_i 构造分析器的状态 i, *action* 函数在状态 i 的值按如下方法确定。

(a) 如果 $[A \to \alpha \cdot a\beta, b]$ 在 I_i 中,且 $goto(I_i, a) = I_j$,那么置 $action[i, a]$ 为 sj, 这里,a 一定是终结符。

(b) 如果 $[A \to \alpha \cdot, a]$ 在 I_i 中,且 $A \neq S'$,那么置 $action[i, a]$ 为 rj,j 是产生式 $A \to \alpha$ 的序号。

(c) 如果 $[S' \to S \cdot, \$]$ 在 I_i 中,那么置 $action[i, \$] = acc$。

如果用上面规则构造出现了冲突,那么该文法就不是 LR(1)的。算法对此文法失败。

(3) *goto* 函数在状态 i 的值按如下方法确定:

如果 $goto(I_i, A) = I_j$, 那么 $goto[i, A] = j$。

(4) 用规则(2)和(3)未能定义的所有条目都是空白,表示出错。

(5) 分析器的初始状态是包含 $[S' \to \cdot S, \$]$ 的项目集对应的状态。 □

用算法 3.6 产生的动作函数和转移函数构成的表叫做**规范的** LR(1)分析表,使用这种表的 LR 分析器叫做规范的 LR(1)分析器。如果动作函数没有多重定义的条目,那么该文法叫做 **LR(1)文法**。和前面一样,如果不会引起误解的话,省略“(1)”。

例 3.33 文法(3.13)的规范 LR 分析表见表 3.9,产生式 1,2 和 3 分别是 $S \to BB$,$B \to bB$ 和 $B \to a$。 □

所有的 SLR(1)文法都是 LR(1)文法,但对于 SLR(1)文法,规范的 LR 分析器可能比同一文法的 SLR 分析器有更多的状态。上面例子的文法是 SLR 文法,它的 SLR 分析器只有 7 个状态,而图 3.20 有 10 个状态。

表 3.9 文法(3.13)的规范 LR 分析表

状态	动作			转移	
	b	*a*	$	*S*	*B*
0	*s*3	*s*4		1	2
1			*acc*		
2	*s*6	*s*7			5
3	*s*3	*s*4			8
4	*r*3	*r*3			
5			*r*1		
6	*s*6	*s*7			9
7			*r*3		
8	*r*2	*r*2			
9			*r*2		

对于例 3.29,当构造它的规范 LR(1)分析表时,可以看到不存在任何冲突。

3.5.5 构造 LALR 分析表

本小节介绍最后一种分析器的构造方法——LALR(lookahead LR)方法。实际的编译器经常使用这种方法,因为由它产生的分析表比规范 LR 的分析表要小得多,而且对大多数一般编程语言来说,其语法构造都能方便地由 LALR 文法表示。同样的结论对 SLR 文法几乎也是对的,但是有少数构造不能方便地由 SLR 技术处理(例 3.29 便是一个例子)。

就分析器的大小而言,SLR 和 LALR 的分析表对同一个文法有同样多的状态,而规范 LR 分析表要大得多。例如,对 C 这样的语言,SLR 和 LALR 的分析表有几百个状态,而规范 LR 分析表有几千个状态。所以,使用 SLR 和 LALR 的分析表比使用规范 LR 分析表要经济得多。

再次考虑文法(3.13),它的 LR(1)项目集见图 3.20。取一对类似的状态,如 I_4 和 I_7,它们都只有一个项目,并且第一成分都是 $B \to a\cdot$,I_4的搜索符是 b 或 a,I_7的搜索符是 \$ 。

观测一下分析器中 I_4和 I_7的不同作用。注意,文法(3.13)产生的是正规语言 b^*ab^*a。当读输入 $bb\cdots babb\cdots ba$ 时,分析器把第一组 b 和它后面的 a 移进栈,进入状态 4。随后,如果下一个输入符号是 b 或 a 的话,分析器按产生式 $B\to a$ 归约,因为 b 或 a 属于第二个 b^*a 的开始符号。如果下一个输入符号是 \$,那么整个输入实际是 $bb\cdots ba$ 的形式,它不属于这个语言,这时状态 4 能正确地指出错误。

在读过第二个 a 后,分析器进入状态 7。这时分析器必须看见输入结束标记 \$,否则输入串就不是 b^*ab^*a 的形式。所以,合乎道理的做法是,面临 \$ 时,状态 7 应按 $B \to a$ 归约,而面临 b 或 a 时则报告错误。

现在把状态 I_4和 I_7合并为 I_{47},并把它们的搜索符也合并起来,成为[$B \to a\cdot$, $b/a/$ \$]。从 I_0,I_2,I_3和 I_6到达 I_4或 I_7的 a 转移现在都进入 I_{47},状态 I_{47}的动作是不管面临任何符号都归约。修改后的分析器的行为本质上和原来的一样,但它会把输入 bba 的 a 归约成 B,把输入 $babab$ 的第二个 a 归约成 B,而原来的分析器对这两种情况都是报错的。幸好,这些错误最终还是会被修改后的分析器捕获,而且是在移进下一个输入符号前被捕获。

更一般地说,需要寻找的是**同心的 LR(1)**项目集,即略去搜索符后它们是相同的集合,并把这些同心集合并成一个项目集。例如在图 3.20 中,I_4和 I_7,I_3和 I_6以及 I_8和 I_9分别是同心的项目集。注意,一般而言,一个心是对应文法的一个 LR(0)项目集;另外,LR(1)文法可能会使多个项目集同心。

因为 $goto(I, X)$的心仅依赖于 I 的心,被合并集合的 $goto$ 函数的结果集合也可以合并,所以项目集合并时带来的 $goto$ 函数修改不会引起问题。动作函数应做相应的修改,使得它能够反映合并前所有项目集的非出错动作。

对 LR(1)文法,如果把所有的同心集合并,有可能导致冲突。但是这种冲突不会是移进-归约冲突。因为,如果存在这种冲突,则意味着,面对当前的输入符号 a,有一项目[$A\to\alpha\cdot$, a]要求采取归约动作,同时又有另一项目[$B\to\beta\cdot a\gamma$, b]要求把 a 移进。这两个项目既然同处于合并之

后的项目集中，则意味着在合并前，必有某个 c 使得 $[A \to \alpha \cdot,\ a]$ 和 $[B \to \beta \cdot a\gamma,\ c]$ 同处于合并前的某一集合中。然而，这又意味着，原来的 LR(1) 项目集就已经存在着移进–归约冲突，从而文法不是 LR(1) 的了。因此，同心集的合并不会带来新的移进–归约冲突。

但是，同心集的合并有可能产生新的归约–归约冲突，例 3.34 是说明该问题的一个例子。

例 3.34 考虑文法

$$S' \to S \qquad\qquad A \to c$$

$$S \to aAd \mid bBd \mid aBe \mid bAe \quad B \to c$$

它只产生四个串：*acd*、*bcd*、*ace* 和 *bce*。构造该文法的 LR(1) 项目集不出现冲突，它是 LR(1) 文法。在它的项目集中，对活前缀 *ac* 有效的项目集为 $\{[A \to c\cdot,\ d],\ [B \to c\cdot,\ e]\}$，对活前缀 *bc* 有效的项目集为 $\{[A \to c\cdot,\ e],\ [B \to c\cdot,\ d]\}$，这两集合都不含冲突，它们是同心的。然而，它们合并后变成 $\{[A \to c\cdot,\ d/e],\ [B \to c\cdot,\ d/e]\}$，出现归约–归约冲突，因为面临 e 或 d 时，不知道应该用哪个产生式进行归约。 □

下面给出构造 LALR 分析表的算法，其基本思想是，首先构造 LR(1) 项目集规范族，如果它不存在冲突，则把同心集合并，再按合并后的项目集构造分析表。这个方法可以作为描述 LALR (1) 文法的基本定义。由于该方法先构造完整的 LR(1) 项目集规范族，需要较多的时间和空间，因此该方法只有概念上的意义，实际中通常使用避免构造完整的 LR(1) 项目集规范族的算法，有兴趣的读者可以查阅相关书籍。

算法 3.7 一个简易但耗空间的 LALR 分析表构造法。

输入 拓广文法 G'。

输出 G' 的 LALR 分析表的 *action* 函数和 *goto* 函数。

方法 (1) 构造 LR(1) 项目集规范族 $C = \{I_0,\ I_1,\ \cdots,\ I_n\}$。

(2) 寻找 LR(1) 项目集规范族中同心的项目集，用它们的并集代替它们。

(3) 令 $C' = \{J_0,\ J_1,\ \cdots,\ J_m\}$ 是合并后的 LR(1) 项目集族。*action* 函数在状态 i 的值以和算法 3.6 同样的方式从 J_i 构造。如果出现分析动作的冲突，则算法不能产生分析表，此文法不是 LALR(1) 的。

(4) *goto* 函数按如下方式构造：如果 J 是若干个 LR(1) 项目集的并集，即 $J = I_1 \cup I_2 \cup \cdots \cup I_m$，那么 $goto(I_1,\ X)$，$goto(I_2,\ X)$，…，$goto(I_m,\ X)$ 也同心，因为 $I_1,\ I_2,\ \cdots,\ I_m$ 都同心。记 K 为所有和 $goto(I_1,\ X)$ 同心的项目集的并集，那么 $goto(J,\ X) = K$。 □

由算法 3.7 产生的表叫做 G 的 **LALR 分析表**。如果没有分析动作的冲突，那么该文法叫做 **LALR(1) 文法**。在第(3)步构造的项目集族叫做 **LALR(1) 项目集族**。

例 3.35 再次考虑文法(3.13)，它的转换图见图 3.20。如已提到的那样，有三对项目集可以合并，I_3 和 I_6 合并成

$$I_{36}:\ B \to b \cdot B,\ b/a/\$$$

$$B \to \cdot bB,\ b/a/\$$$

$$B \to \cdot a,\ b/a/\$$$

I_4和 I_7合并成

$I_{47}: B \to a\cdot, \ b/a/\$$

I_8和 I_9合并成

$I_{89}: B \to bB\cdot, \ b/a/\$$

压缩项目集后的 LALR 动作函数和移转函数如表 3.10 所示。

表 3.10 文法(3.13)的 LALR 分析表

状态	动作			转移	
	b	*a*	$	*S*	*B*
0	*s*36	*s*47		1	2
1			*acc*		
2	*s*36	*s*47			5
36	*s*36	*s*47			89
47	*r*3	*r*3	*r*3		
5			*r*1		
89	*r*2	*r*2	*r*2		

为了理解转移函数是怎样计算的,考查一下 $goto(I_{36}, B)$。在原来的 LR(1)项目集中,$goto(I_3, B) = I_8$,而 I_8现在是 I_{89}的一部分,因此置 $goto(I_{36}, B) = I_{89}$。如果考查 I_{36}的另一部分 I_6,可以得到同样的结论,即 $goto(I_6, B) = I_9$,而 I_9现在是 I_{89}的一部分。再看 $goto(I_2, b)$,它指出了在面临 b 执行了 I_2的移进动作之后的转移方向。原来的 LR(1)项目集中,$goto(I_2, b) = I_6$,因为 I_6现在是 I_{36}的一部分,所以 $goto(I_2, b) = I_{36}$。于是,分析表中状态 2 面临 b 的条目是 *s*36,其含意是移进 b,再把状态 36 压在栈顶。 □

当输入串为 b^*ab^*a 时,不论是表 3.9 的 LR 分析器还是表 3.10 的 LALR 分析器,都给出了同样的移进-归约序列,其差别只是状态名不同而已。对于 LALR 文法,这种关系总是保持,即只要输入串正确。LR 和 LALR 分析始终形影相随。

当输入串有错误时,LALR 分析可能比 LR 分析多做了一些不必要的归约,但 LALR 分析决不会比 LR 分析移进更多的符号。即就准确地指出输入串的出错位置而言,LALR 分析和 LR 分析是等效的。例如,输入串是 *bba* 时,LALR 分析的动作序列见表 3.11。

表 3.11 LALR 分析器对于输入 *bba* 的格局

	栈	符号	输入	动作
(1)	0		*bba* $	移进
(2)	0 36	*b*	*ba* $	移进

续表

	栈	符号	输入	动作
(3)	0 36 36	$b\ b$	$a\,\$$	移进
(4)	0 36 36 47	$b\ b\ a$	$\$$	按 $B \to a$ 归约
(5)	0 36 36 89	$b\ b\ B$	$\$$	按 $B \to bB$ 归约
(6)	0 36 89	$b\ B$	$\$$	按 $B \to bB$ 归约
(7)	0 2	B	$\$$	报告错误

而如果用 LR 分析,分析器将把状态

0 3 3 4

压进栈,并在状态 4 发现错误,因为状态 4 面临 \$ 的条目内容是空白。两者的区别是 LR 分析能及时发现错误。

从 SLR、规范 LR 和 LALR 分析表的构造和使用可以看出,只要把状态压入栈就可以了,文法符号进栈是多余的,因为分析过程中从不根据栈中的文法符号来决定动作。先前的介绍中把文法符号压栈,纯粹是为了便于直观理解。表 3.11 的动作序列就是按文法符号不进栈给出的。第三列的符号也可以不要,放在这里还是为了直观理解。从现在开始,只讲状态进栈,而不讲符号和状态都进栈。

例 3.34 的文法是规范的 LR(1)的,但不是 LALR(1)的。例 3.29 的文法是 LALR(1)的,但不是 SLR(1)的。这两个例子说明这三类文法的集合是不同的。

3.5.6 非二义且非 LR 的上下文无关文法

在 3.5.2 节曾经提到,若自左向右扫描的移进-归约分析器能及时识别出现在栈顶的句柄,那么相应的文法就是 LR 的。而二义文法一定不是 LR 的,那么是否存在非二义并且非 LR 的文法呢? 回答是肯定的,可以用例子来说明。

例 3.36 语言 $L = \{ww^{R} \mid w \in (a \mid b)^{*}\}$的文法

$S \to aSa \mid bSb \mid \varepsilon$

不是 LR 的。直观上说,对于该语言的任何句子,如 *abaaba*,扫描前一半字符时应该压栈,扫描后一半字符时先做一次空归约(ε 归约),然后将剩余字符和栈中的字符通过归约进行比较,以保证后一半是前一半的逆。问题是,向前搜索一个字符无法判断是否已到达串的中点。因此该文法不是 LR(1)的,构造分析表时肯定会出现移进-归约冲突。事实上,对于任意大的 k,总能找到一个句子,即使是向前搜索 k 个字符也无法判断是否应该做空归约了。因此该文法不是 LR(k)的。 □

例 3.37 为语言

$$L = \{ a^m b^n \mid n > m \geqslant 0 \}$$

写三个文法。它们分别是 LR(1)的、二义的和非二义且非 LR(1)的。

该语言的句子是 $aa\cdots abb\cdots b$ 的形式,但后面 b 的个数比前面 a 的个数多。为了保证 b 出现在 a 的后面,并且 b 的个数不少于 a 的个数,应该有形如 $S \rightarrow aSb$ 这样的产生式,得到 a 和 b 个数相同的句型。为了能推导出更多的 b,应该有形如 $S \rightarrow Sb$ 的产生式。前一个产生式用来把 b 和前面的 a 进行配对,由于 b 的个数比 a 的多,配对方式可以有多种,不同配对方式形成不同的文法。

如果将 $a^m b^n$ 的前 m 个 b 和 a^m 配对,如图 3.21(a)所示,那么按此特点写出的文法

$S \rightarrow AB$

$A \rightarrow aAb \mid \varepsilon$

$B \rightarrow Bb \mid b$

是 LR(1)文法。

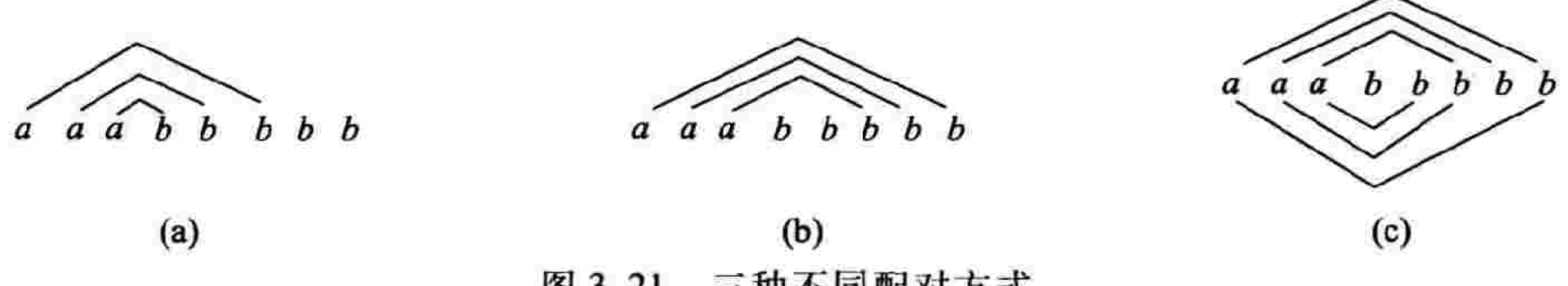


图 3.21 三种不同配对方式

如果将 $a^m b^n$ 的后 m 个 b 和 a^m 配对,如图 3.21(b)所示,那么按此特点写出的文法

$S \rightarrow aSb \mid B$

$B \rightarrow Bb \mid b$

是非二义且非 LR(1)文法。因为在分析时,当把所有的 a 压进栈后,要先将中间若干个 b 归约成 B,使得剩下的 b 和前面的 a 一样多。向前搜索一个符号是不可能确定将栈顶的 B 归约成 S,还是移进下一个 b 并把 Bb 归约成 B。

如果让 $a^m b^n$ 中的 a 和 b 有不止一种配对方式,如图 3.21(c)所示,那么它的文法就是二义的,如

$S \rightarrow aSb \mid Sb \mid b$ □

3.6 二义文法的应用

任何二义文法绝不是 LR 文法,因而不属于上节所讨论的任何一类文法。但是,正如在这一节将要看到的,某些二义文法对规范和实现语言是有用的。像表达式这样的语言构造,二义文法提供的规范比任何其他非二义文法提供的都要简短些,更自然些。另外,为了便于对某个语法构造的特殊情况安排特别的语义处理,需要在文法中增加针对这种特殊情况的产生式,以便把它从

该语法构造的一般情况中分离出来，这种产生式的加入会使文法二义，这是二义文法的另一应用。

必须强调，虽然使用的文法是二义的，但若对所有二义情况都有消除二义的规则，保证每个句子正好只有一棵分析树，那么在此全面规范下，对应语言仍然是无二义的。此时有可能为该语言设计一个 LR 分析器，该分析器消除二义的选择和语言规范中消除二义的规则一致。

3.6.1 使用算符的优先级和结合性来解决冲突

考虑编程语言的表达式。下面含有二元算符+和 * 的算术表达式文法是二义的，因为它没有指出算符+和 * 的结合性和优先级：

$$E \to E+E \mid E*E \mid (E) \mid \mathbf{id} \tag{3.14}$$

无二义的文法

$$\begin{aligned} E &\to E+T \mid T \\ T &\to T*F \mid F \\ F &\to (E) \mid \mathbf{id} \end{aligned} \tag{3.15}$$

产生同样的语言，但它给+以较低的优先级，并让两个算符都具有左结合性。使用文法(3.14)有优于使用文法(3.15)的地方。首先，如下面将要看到的那样，算符+和 * 的结合性和优先级可以方便地改变而无须修改文法(3.14)，也不会改变分析器的状态数。其次，文法(3.15)的分析器要花一部分时间来完成产生式 $E \to T$ 和 $T \to F$ 的归约，而文法(3.14)的分析器不会消耗时间在归约这样的**单非产生式**(右部只有一个非终结符产生式)上。

文法(3.14)用 $E' \to E$ 拓广后的 LR(0)项目集见图 3.22。因为文法(3.14)二义，因此从这些项目集产生 LR 分析表时，肯定会出现分析动作的冲突，冲突出现在项目集 I_7 和 I_8 对应的状态。假如用 SLR 方法来构造动作表，I_7 引起的冲突是在面临+和 * 时，移进该符号还是按 $E \to E+E$ 归约，因为+和 * 都在 $FOLLOW(E)$ 中。I_8 引起的冲突是在面临+和 * 时，移进该符号还是按 $E \to E*E$ 归约。事实上，用任何一种 LR 分析表的构造方法都会产生这些冲突。

这些冲突可以用有关+和 * 的优先级和结合性的信息来解决。考虑输入 **id**+**id** * **id**，基于图 3.22 的分析器在扫描过 **id**+**id** 后进入状态 7，形成如下格局：

栈	符号	输入
0 1 4 7	**id**+**id** (归约成了 $E+E$)	* **id** \$

如果 * 的优先级高于+，分析器应把 * 移进栈，准备归约 * 和它两侧的 **id** 到一个表达式。这正是基于该语言无二义文法的 SLR 分析器(表 3.7)要做的。另一方面，如果+的优先级高于 *，那么分析器应该归约。这样，根据+和 * 的优先关系，可以解决在状态 7 面临 * 时，移进 * 和按 $E \to E+E$ 归约之间的冲突。

如果输入是 **id**+**id**+**id**，分析器扫描完 **id**+**id** 后到达的格局和上面的唯一区别是，下一个输入符号是+而不是 *。在状态 7 面临+时仍有移进-归约冲突，现在由算符+的结合性来解决冲突。

如果+是左结合,正确的动作是按 $E \rightarrow E+E$ 归约,即第一个+及其前后的 **id** 应看成一组。这个选择和例 3.24 文法的 SLR 分析器的动作是一致的。

总之,假如+是左结合的,那么在状态 7 面临+时应该按 $E \rightarrow E+E$ 归约;如果 * 的优先级高于+,那么在状态 7 面临 * 时应该移进。可以类似地讨论状态 8,最后得出如下结果:如果 * 是左结合且优先级高于+,那么不论面临+还是 *,分析器在状态 8 的动作都是按 $E \rightarrow E*E$ 归约。

按这种方式处理,可以得到表 3.12 的 LR 分析表,产生式(1)到产生式(4)分别是 $E \rightarrow E+E$,$E \rightarrow E*E$,$E \rightarrow (E)$ 和 $E \rightarrow$ **id**。有趣的是,类似的动作表可以从表 3.7 的 SLR 表中删去文法(3.15)的单非产生式 $E \rightarrow T$ 和 $T \rightarrow F$ 的归约得到。在构造 LALR 分析表和规范的 LR 分析表时,也可用类似的方法来处理(3.14)这样的二义文法。

I_0: $E' \rightarrow \cdot E$
$E \rightarrow \cdot E+E$
$E \rightarrow \cdot E*E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot$ **id**

I_1: $E' \rightarrow E \cdot$
$E \rightarrow E \cdot +E$
$E \rightarrow E \cdot *E$

I_2: $E' \rightarrow (\cdot E)$
$E \rightarrow \cdot E+E$
$E \rightarrow \cdot E*E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot$ **id**

I_3: $E \rightarrow$ **id** $\cdot$

I_4: $E \rightarrow E+ \cdot E$
$E \rightarrow \cdot E+E$
$E \rightarrow \cdot E*E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot$ **id**

I_5: $E \rightarrow E* \cdot E$
$E \rightarrow \cdot E+E$
$E \rightarrow \cdot E*E$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot$ **id**

I_6: $E \rightarrow (E \cdot)$
$E \rightarrow E \cdot +E$
$E \rightarrow E \cdot *E$

I_7: $E \rightarrow E+E \cdot$
$E \rightarrow E \cdot +E$
$E \rightarrow E \cdot *E$

I_8: $E \rightarrow E*E \cdot$
$E \rightarrow E \cdot +E$
$E \rightarrow E \cdot *E$

I_9: $E \rightarrow (E) \cdot$

图 3.22 文法(3.14)拓广后的 LR(0)项目集

表 3.12 文法(3.14)的分析表

状态	动作						转移
	id	+	*	(	)	$	*E*
0	*s*3			*s*2			1
1		*s*4	*s*5			*acc*	
2	*s*3			*s*2			6
3		*r*4	*r*4		*r*4	*r*4	
4	*s*3			*s*2			7
5	*s*3			*s*2			8
6		*s*4	*s*5		*s*9		
7		*r*1	*s*5		*r*1	*r*1	
8		*r*2	*r*2		*r*2	*r*2	
9		*r*3	*r*3		*r*3	*r*3	

3.6.2 使用其他约定来解决冲突

考虑条件语句文法

stmt → **if** *expr* **then** *stmt* **else** *stmt*

| **if** *expr* **then** *stmt*

| **other**

该文法是二义的,因为它没有解决悬空 else 的二义性问题。可以肯定,构造 LR 分析表时,会碰到移进-归约冲突,即,当 **if** *expr* **then** *stmt* 在栈顶,并且 **else** 是下一个输入符号时,究竟是将 **if** *expr* **then** *stmt* 归约还是将 **else** 移进。根据语言关于 else 配对的约定,可以知道,对于这种移进-归约冲突,应该忽略归约,采用移进,即优先移进。

再举一个实际例子。如果需要引入额外的产生式,来表示由其余产生式产生的语法构造的一种特殊情况时,文法会因加入了这额外的产生式而引起二义性,从而引起分析动作的冲突。先举一个这种文法的例子。

历史上,数学公式编排预处理器 EQN 中使用了特殊情况产生式,这是一个有趣的应用。在 EQN 中,描述数学表达式的文法用算符 **sub** 表示下角标并用算符 **sup** 表示上角标,这个文法的片段见(3.16)。其中花括号由预处理器用来表示复合表达式,*c* 作为表示任意正文串的记号。

(1) *E* → *E* **sub** *E* **sup** *E*

(2) *E* → *E* **sub** *E*

(3) $E \to E$ **sup** E (3.16)

(4) $E \to \{E\}$

(5) $E \to c$

文法(3.16)是二义的。该文法没有说明算符 **sub** 和 **sup** 的结合性和优先级。即使由它们引起的二义性解决了，比如规定这两个算符的优先级相同并且都是右结合的，该文法仍然是二义的。这是因为产生式(1)分离出了由产生式(2)和(3)产生的表达式的一种特例，即形式为 E **sub** E **sup** E的表达式。没有产生式(1)，该文法产生的语言是一样的。把这种形式的表达式处理为一种特殊情况的理由是，像 a **sub** i **sup** 2 这样的表达式应该排版成 a_i^2，而不是 ${a_i}^2$ 或 a_{i^2}的形式。只有加上特殊情况产生式后，EQN 才能够产生这样特殊的输出。

构造该文法的 LR 分析表时会发现，存在移进-归约冲突和归约-归约冲突。其中移进-归约冲突可以根据 **sub** 和 **sup** 的优先级和结合性来解决。归约-归约冲突发生在产生式

$E \to E$ **sub** E **sup** E

$E \to E$ **sup** E

之间。即当 E **sub** E **sup** E 出现在栈顶时，是按前一个产生式将 E **sub** E **sup** E 归约，还是按后一个产生式将 E **sup** E 归约。显然，应该优先特殊情况产生式，即按前一个产生式归约。这样，和该特殊情况产生式相联的语义动作才可能用更专门的措施来输出特定的格式。

写一个无二义文法，提取出语法构造的特殊情况，这是非常困难的。为了体会这是何等的困难，请读者为文法(3.16)构造等价的无二义文法，它分离形式为 E **sub** E **sup** E 的表达式。

3.6.3 LR 分析的错误恢复

LR 分析器在访问动作表时若遇到空白条目，那么它就发现了错误。但是在访问转移表时它绝不会遇到空白条目。只要已扫描的输入出现一个不正确的后继，LR 分析器便立即报告错误，绝不会把不正确的后继移进栈。规范的 LR 分析器甚至在报告错误之前决不做任何无效归约，但 SLR 和 LALR 分析器在报告错误前有可能执行几步无效归约。

在 LR 分析中，可按如下方法实现紧急方式的错误恢复：从栈顶开始退栈，直至出现状态 s，它有预先确定的非终结符 A 的转移；然后抛弃若干个（可以是零个）输入符号，直至找到符号 a，它能合法地跟随 A。分析器再把 A 和状态 $goto[s, A]$压进栈，恢复正常分析。A 的选择可能不唯一，一般来说 A 应是代表主要语法构造的非终结符，如表达式、语句或程序块。例如，若 A 是非终结符 *stmt*，那么 a 可以是分号或}，后者标记一个语句序列的结束。

这种错误恢复方法的实质是试图忽略含语法错误的短语。分析器认为由 A 推出的串含有一个错误，这个串的一部分已经处理，该处理的结果是若干个状态已压到栈顶。这个串的其余部分仍在剩余输入中。分析器试图跳过这个串的其余部分，在剩余输入中找到一个符号，它能合法地跟随 A。通过从栈中弹出一些状态，跳过若干输入符号，把 $goto[s, A]$推进栈，分析器装扮成已发现了 A 的一个实例，并恢复正常分析。

错误恢复的另一种方式叫做**短语级恢复**。当发现错误时,分析器对剩余输入作局部纠正,它用可以使分析器继续分析的串来代替剩余输入的前缀。典型的局部纠正有:用分号代替逗号,删除多余的分号,或插入遗漏的分号等。编译器的设计者必须仔细选择替换的串,以免引起死循环。死循环是可能的,例如,总是在当前输入符号的前面插入一些东西。

这种替换可以纠正任何输入串,它的主要缺点是很难应付实际错误出现在诊断点以前的情况。

对 LR 分析来说,短语级恢复的实现过程是,考察 LR 分析表的每一个空白条目并根据语言的使用情况,决定最可能进入该条目的输入错误,然后为该条目编一个适当的错误恢复过程。

可以在分析表动作域的每个空白条目中填入一个指示器,它指向编译器设计者为之设计的错误处理例程。该例程的动作包括在栈顶和/或输入中插入或删除状态或符号,或者改变或置换输入符号等。要注意,所做的选择不应引起 LR 分析器进入无限循环。保证至少有一输入符号被删除或最终被移进,或者在到达输入末尾时保证栈最终会缩短的策略是一种安全的策略。应该避免从栈中弹出对应非终结符的状态,因为这种修正删除了已经被成功分析的一个构造。

例 3.38 考虑表达式文法

$E \rightarrow E+E \mid E*E \mid (E) \mid \mathbf{id}$

表 3.13 给出了该文法的 LR 分析表,它是在表 3.12 的基础上加了错误诊断和恢复。有的状态对某些输入符号的动作是按某个产生式进行归约,现在把该状态下其他空白条目都改成了归约,这种修改会推迟错误的发现,多执行了一步或几步归约,但错误仍在移进下一个符号前被捕获。表 3.12 的其余空白条目已经改成了调用错误处理例程。

表 3.13 有错误处理例程的 LR 分析表

状态	动作						转移
	id	+	*	(	)	$	*E*
0	*s*3	*e*1	*e*1	*s*2	*e*2	*e*1	1
1	*e*3	*s*4	*s*5	*e*3	*e*2	*acc*	
2	*s*3	*e*1	*e*1	*s*2	*e*2	*e*1	6
3	*r*4	*r*4	*r*4	*r*4	*r*4	*r*4	
4	*s*3	*e*1	*e*1	*s*2	*e*2	*e*1	7
5	*s*3	*e*1	*e*1	*s*2	*e*2	*e*1	8
6	*e*3	*s*4	*s*5	*e*3	*s*9	*e*4	
7	*r*1	*r*1	*s*5	*r*1	*r*1	*r*1	
8	*r*2	*r*2	*r*2	*r*2	*r*2	*r*2	
9	*r*3	*r*3	*r*3	*r*3	*r*3	*r*3	

错误处理例程如下。

*e*1:/* 分析器处于状态 0,2,4 或 5 时,期望输入符号为运算对象首符,即 **id** 或左括号,现在遇到的是+,* 或 $,调用此例程。*/

把状态 3 压进栈(状态 0,2,4 和 5 面临 **id** 时的转移);给出诊断信息“缺少运算对象”。

*e*2:/* 分析器处于状态 0,1,2,4 或 5,遇到右括号时调用此例程。*/

从输入中删除右括号;给出诊断信息“不配对的右括号”。

*e*3:/* 分析器处于状态 1 或 6,期望运算符,但遇到的却是 **id** 或左括号时,调用此例程。*/

把状态 4 压进栈(对应符号+);给出诊断信息“缺少算符”。

*e*4:/* 分析器处于状态 6,遇到 $ 时调用此例程。*/

把状态 9 压入栈(对应右括号);给出诊断信息“缺少右括号”。

读者可以用有语法错误的简短表达式为例,例如 **id**+),体会该错误恢复方法的效果。 □

3.7 语法分析器的生成器

本节介绍分析器的生成器如何用来帮助构造编译器的前端。该介绍将 LALR 分析器的生成器 Yacc(Yet Another Compiler-Compiler)作为讨论的基础,因为它实现了前两节讨论的许多概念,并且使用广泛。Yacc 是 20 世纪 70 年代初期分析器的生成器盛行时的产物,它已经被用来帮助实现了许多产品编译器,现在它仍然是 UNIX 和 Linux 等系统下的一个好工具。

3.7.1 分析器的生成器 Yacc

一个翻译器可用 Yacc 按图 3.23 表示的方式构造出来。首先,用 Yacc 语言将翻译器的规范建立于一个文件(例如 translate.y)中。UNIX 系统的命令

```
yacc translate.y
```

把文件 translate.y 翻译为 C 语言文件,叫做 y.tab.c,它使用的是 LALR 方法。程序 y.tab.c 包含用 C 写的 LALR 分析器和其他由用户准备好的 C 语言例程。为了使 LALR 分析表少占空间,使用紧凑技术来压缩分析表的大小。

然后,再用命令

```
cc y.tab.c -ly
```

编译 y.tab.c,其中的选项 ly 表示使用 LR 分析器的库(名字 ly 随系统而定),它包含 LR 分析的驱动程序。编译的结果是目标程序 a.out,该目标程序能完成上面的 Yacc 规范指定的翻译。如果还需要其他过程的话,它们可以和 y.tab.c 一起编译和连接。

用 Yacc 写的规范由三部分组成:

声明

%%
翻译规则
%%
用C语言编写的支持例程

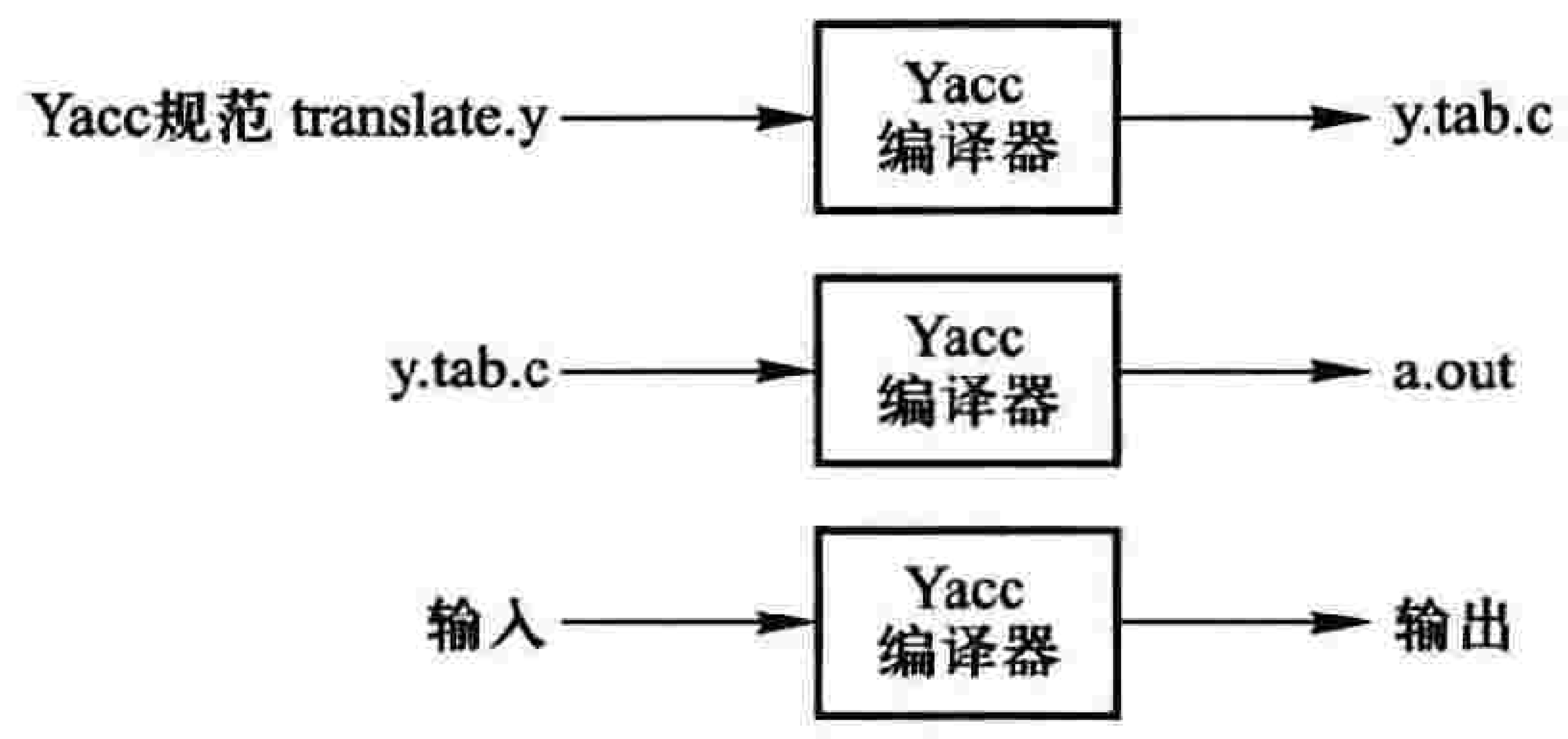


图3.23 用Yacc建立翻译器

例3.39 以构造一个简单的计算器为例，说明怎样准备Yacc规范。该计算器读一个算术表达式，计算并打印它的值。构造该计算器从下面算术表达式的文法开始：

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid$ **digit**

记号 **digit** 是0~9之间的单个数字。基于这个文法的计算器的Yacc规范见图3.24。 □

Yacc规范的声明部分有两节，它们都是可选的。第一节处于分界符%{和%}之间，它是一些普通的C语言声明，这里声明的常量和变量等提供给第二部分和第三部分的翻译规则和过程使用。图3.24中，这一节只有一个包含语句

include <ctype. h>

因为这个文件含有谓词isdigit。

声明部分的第二节是文法终结符(即词法记号)的声明，图3.24的语句

%token DIGIT

声明DIGIT是记号。这一节声明的记号可用于Yacc规范的第二部分和第三部分。如果词法分析器是由Lex建立的，这些记号声明对该词法分析器也是可用的。

Yacc规范的第二部分位于第一个%%后面，该部分放置翻译规则，每条规则由一个文法产生式及和它相联系的语义动作组成。产生式集合

左部 → 选择1 | 选择2 | … | 选择 *n*

在Yacc中写成

```
左部    : 选择1 {语义动作1}
        | 选择2 {语义动作2}
        …
        | 选择n {语义动作n}
        ;
```

```
%{
#include <ctype.h>
%}
%token DIGIT
%%
line    : expr '\n'          {printf ("%d\n", $1);}
        ;
expr    : expr '+'term       {$$ = $1 + $3;}
        | term
        ;
term    : term '*'factor     {$$ = $1 * $3;}
        | factor
        ;
factor  : '('expr')'         {$$ = $2;}
        | DIGIT
        ;
%%
yylex () {
  int c;
  c = getchar ();
  if (isdigit (c)) {
    yylval = c - '0';
    return DIGIT;
  }
  return c;
}
```

图 3.24　简单计算器的 Yacc 规范

在 Yacc 产生式中,没有加引号的字母数字串,若没有被声明为记号,则被看成非终结符;加引号的单个字符,例如'+',被看成是由这个字符+代表的记号。右部的各个选择及其语义动作之间用竖线隔开,最后一个选择的后面用分号,表示该产生式集合的结束。第一个左部非终结符是开始符号。

Yacc 的语义动作是 C 语句序列。在语义动作中,符号 $\$\$$ 表示引用左部非终结符的属性值,而 $\$i$ 表示引用右部第 i 个文法符号的属性值。每当归约一个产生式时,执行与之关联的语义动作,所以语义动作一般是从各 $\$i$ 的值决定 $\$\$$ 的值。在图 3.24 的 Yacc 规范中,两个 E 产生式

$$E \rightarrow E + T \mid T$$

及和它们相关的语义动作写成

```
expr    : expr '+'term    { $$ = $1 + $3;}
```

```
            | term
            ;
```

注意,在第一个产生式中,非终结符 term 是右部的第三个文法符号,'+'是第二个文法符号。第一个产生式的语义动作是把右部 expr 的值和 term 的值相加,把结果赋给左部非终结符 expr,作为它的值。第二个产生式的语义动作的描述被省略,因为当右部只有一个文法符号时,默认的语义动作就是{ $$ = $1;},这正是所希望的动作。

注意,一个新的开始产生式

```
line    : expr '\n'   {printf ("%d\n", $1);}
```

被加到这个 Yacc 规范中。该产生式的意思是,这个计算器的输入是一个表达式后面跟一个换行字符。它的语义动作是打印表达式的十进制值并且换行。

Yacc 规范的第三部分是一些用 C 语言写的支持例程。必须提供名字为 yylex()的词法分析器(当然也可以用 Lex 来产生 yylex()),其他的过程,如错误恢复例程,需要的话也可以加上。

词法分析器 yylex()返回记号,它由记号名和属性值两部分组成。如果返回的是 DIGIT 这样的记号名,该名字还必须声明在 Yacc 规范第一部分的第一节中,给它一个区分于其他记号的值。属性值是通过 Yacc 定义的变量 yylval 传给分析器的。

图 3.24 的词法分析器是非常粗糙的。它用 C 语言的函数 getchar()每次读一个输入字符,如果是数字字符,取它的值存入变量 yylval,返回记号 DIGIT,否则把字符本身作为记号返回。

3.7.2 用 Yacc 处理二义文法

修改上节的 Yacc 程序,使计算器功能更强。首先,让计算器计算一系列表达式,每行一个,也允许表达式之间有空白行。为此,将第一个翻译规则改为

```
lines   : lines expr '\n'      {printf ( "%g \n" , $2 );}
        | lines '\n'
        | /* empty */
        ;
```

按照 Yacc 的规定,第三行的空选择表示 ε。

其次,允许表达式使用由多个数字组成的数,并增加算符包括+,-(一元和二元),*和/。使用二义文法

$$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid (E) \mid -E \mid \textbf{number}$$

来描述表达式。最终的 Yacc 程序见图 3.25。

因为图 3.25 的 Yacc 程序的文法是二义的,LALR 算法将产生分析动作的冲突。Yacc 会报告产生了多少个分析动作的冲突。可以用编译选项-v 来获得产生的项目集(仅含核心项目)和分析动作冲突的描述,该选项将这些信息放到另一个文件 y.output 中,该文件还包含了 LR 分析表的可读表示,它表明了 Yacc 是怎样解决这些分析动作的冲突的。

```
%{
# include <ctype.h>
# include <stdio.h>
# define YYSTYPE double/* 将 Yacc 栈定义为 double 类型 */
%}

%token NUMBER
%left '+''-'
%left '*''/'
%right UMINUS
%%
lines    : lines expr '\n'            {printf ( "%g \n", $2 );}
         | lines '\n'
         | /* empty */
         ;
expr     : expr '+'expr               { $$ = $1 + $3;}
         | expr '-'expr               { $$ = $ 1- $3;}
         | expr '*'expr               { $$ = $1 * $3;}
         | expr '/'expr               { $$ = $1 / $3;}
         | '('expr')'                 { $$ = $2;}
         | '-'expr %prec UMINUS       { $$ = - $2;}
         | NUMBER
         ;
%%
yylex ( ) {
  int c;
  while ( ( c = getchar ( ) ) == '');
  if ( ( c == '. ') | | (isdigit (c) ) ) {
         ungetc (c, stdin);
         scanf ("%lf",&yylval);
         return  NUMBER;
  }
  return c;
}
```

图 3.25　更高级的计算器的 Yacc 规范

当 Yacc 报告它发现分析动作冲突时，明智的做法是建立和查阅文件 y.output，以明白为什么会出现分析动作的冲突以及它们是否已经得到恰当的解决。

除非另有说明，否则 Yacc 按下面两条规则解决分析动作的冲突。

(1) 对于归约-归约冲突,选择在 Yacc 规范中最先出现的那个冲突产生式。按此规则,为了正确解决排版文法(3.16)的冲突,只要把产生式(1)放在产生式(3)的前面就足够了。

(2) 对于移进-归约冲突,优先移进。这条规则正确解决了悬空 else 的移进-归约冲突。

由于这些默认的规则并不总是编译器设计者想要的,因而 Yacc 允许编译器设计者提供一些解决移进-归约冲突的说明。在声明部分,可以为终结符指定优先级和结合性。声明

```
%left  '+'  '-'
```

表示+和-有同样的优先级并且它们是左结合的。声明

```
%right '^'
```

表示算符^是右结合的。还可以用声明来限制二元算符为不可结合的(即不允许该算符相邻出现),如

```
%nonassoc '<'
```

终结符的优先级按它们在声明部分出现的次序而定,先出现的优先级低,同一声明中的终结符有相同的优先级。这样,图 3.25 的声明

```
%right UMINUS
```

使得 UMINUS 的优先级高于前面 5 个终结符。

Yacc 解决移进-归约冲突时,首先参考这个冲突涉及的产生式和终结符的优先级与结合性。通常,产生式的优先级与结合性同它最右边的终结符一致,在大多数情况下这是合理的决策。

如果 Yacc 必须在移进输入符号 a 和按产生式 $A \to \alpha$ 归约这两个动作之间进行选择的话,那么当这个产生式的优先级高于 a,或者优先级相同但产生式左结合时,选择归约动作,否则选择移进。例如,给定产生式

$$E \to E+E \mid E*E$$

若下一个输入符号是+,归约项目的产生式是 $E \to E+E$,那么优先作归约,因为该产生式右部的+和下一个输入符号有同样的优先级,而+是左结合的。如果下一个输入符号是 *,那么选择移进,因为 * 的优先级高于该产生式右部+的优先级。

在那些最右边的终结符不能给产生式以适当优先级和结合性的情况下,可以增加一个标签

```
%prec 终结符
```

来强制产生式,使得它的优先级和结合性同该标签终结符的一样。这个终结符仅仅是个占位符,就像图 3.25 的 UMINUS 那样,它不由词法分析器返回,仅通过声明来确定一个产生式的优先级和结合性。图 3.25 中,声明

```
%right UMINUS
```

给记号 UMINUS 指定高于 * 和/的优先级。在翻译规则部分,标签

```
%prec UMINUS
```

出现在产生式

```
expr : '-'expr
```

的后面,它使得该产生式的一元减算符的优先级高于任何其他算符。

Yacc 不报告用上述优先级和结合性能解决的移进-归约冲突。

3.7.3 Yacc 的错误恢复

Yacc 使用一种叫做**错误产生式**的形式来进行错误恢复。首先,它要求编译器设计者决定为哪些“主要的”非终结符设置与它们相关联的错误恢复处理,这些非终结符的典型选择是那些用于产生表达式、语句、程序块和过程的非终结符。然后编译器设计者把形式为 $A \rightarrow$ **error** α 的错误产生式加到文法上,其中 A 是主要非终结符,α 是文法符号串,可以是空串,**error** 是 Yacc 的保留字。Yacc 将从这样的规范产生分析器,把错误产生式当作普通产生式来处理。

当 Yacc 产生的分析器遇到错误时,它用特别的方式来处理其项目集含错误产生式的状态。遇到错误时,分析器从状态栈中逐个弹出状态,直到发现栈顶状态的项目集包含形式为 $A \rightarrow \cdot$ **error** α 的项目为止。然后分析器把虚构的记号 **error**“移进”栈,好像它在输入中看见了这个记号。

当 α 为 ε 时,分析器立即进行对 A 的归约并执行产生式 $A \rightarrow$ **error** 的语义动作(它可能是编译器设计者指定的错误恢复例程),然后它抛弃若干输入符号,直到发现一个能回到正常处理的输入符号为止。

如果 α 非空,分析器就在输入上向前寻找能够归约为 α 的子串。如果 α 含的都是终结符,那么它在输入上直接寻找这样的串,把其中的符号移进栈,使得 **error** α 在分析器的栈顶。随后分析器把 **error** α 归约为 A 并执行产生式 $A \rightarrow$ **error** 的语义动作,恢复正常分析。

上一节介绍的短语级错误恢复方法显然较难用于 Yacc 自动生成分析器的情况。使用 Yacc 的编译器设计者通常用紧急方式的错误恢复思想。例如,错误产生式

$stmt \rightarrow$ **error** ;

要求分析器看见错误时跳过下一个分号,好像这个语句已经被看见一样。

例 3.40 图 3.26 的 Yacc 程序在图 3.25 的基础上增加了错误产生式。错误产生式为

lines : **error** '\n'

当输入行有语法错误时,分析器从栈中弹出状态,直至碰到一个有移进 **error** 动作的状态。状态 0 是唯一符合条件的这种状态,因为它的项目集包含

lines $\rightarrow \cdot$ **error** '\n'

状态 0 总是在栈底。分析器把终结符 **error** 移进栈,抛弃剩余输入符号,直至发现换行字符。然后分析器把换行符移进栈,把 **error** '\n'归约成 *lines*, 输出诊断信息“重新输入上一行”。专门的 Yacc 例程 yyerrok 用来重置分析器回到正常操作方式。 □

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double/* 将 Yacc 栈定义为 double 类型 */
%}

%token  NUMBER
%left '+''-'
%left '*''/'
%right  UMINUS
%%
lines     : lines expr '\n'            {printf ( "%g \n", $2 );}
          | lines '\n'
          |  /* empty */
          |error '\n'                  {yyerror( "reenter previous line:" ); yyerrok;}
          ;
expr      : expr '+'expr               { $$ = $1 + $3;}
          | expr '-'expr               { $$ = $1 - $3;}
          | expr '*'expr               { $$ = $1 * $3;}
          | expr '/'expr               { $$ = $1 / $3;}
          | '('expr ')'                { $$ = $2;}
          | '-'expr %prec UMINUS       { $$ = - $2;}
          | NUMBER
          ;
%%
yylex ( ) {
  int c;
  while ((c = getchar ()) == '');
  if ((c == '.') || (isdigit (c))) {
          ungetc (c, stdin);
          scanf ( "%lf",&yylval);
          return  NUMBER;
  }
  return c;
}
```

图 3.26　有错误恢复的计算器

习 题 3

3.1 考虑文法

$S \to (L) \mid a$

$L \to L, S \mid S$

(a) 建立句子$(a, (a, a))$和$(a, ((a, a), (a, a)))$的分析树。

(b) 为(a)的两个句子构造最左推导。

(c) 为(a)的两个句子构造最右推导。

(d) 这个文法产生的语言是什么?

3.2 考虑文法

$S \to aSbS \mid bSaS \mid \varepsilon$

(a) 为句子 *abab* 构造两个不同的最左推导,以此说明该文法是二义的。

(b) 为 *abab* 构造对应的最右推导。

(c) 为 *abab* 构造对应的分析树。

(d) 这个文法产生的语言是什么?

3.3 下面的二义文法描述命题演算公式,为它写一个等价的非二义文法。

$S \to S$ **and** $S \mid S$ **or** $S \mid$ **not** $S \mid$ **true** $\mid$ **false** $\mid (S)$

3.4 文法

$R \to R\ '|'R \mid R\ R \mid R^* \mid (R) \mid a \mid b$

产生字母表$\{a, b\}$上所有不含ε的正规式。注意,第一条竖线加了引号,它是正规式的或运算符号,而不是文法产生式右部各选择之间的分隔符,另外*在这儿是一个普通的终结符。该文法是二义的。

(a) 证明该文法产生字母表$\{a, b\}$上的所有正规式。

(b) 为该文法写一个等价的非二义文法。它给予算符*、连接和 | 的优先级和结合性同2.2节中定义的一致。

(c) 按上面两个文法构造句子$ab \mid b^*a$的分析树。

3.5 条件语句文法

stmt → **if** *expr* **then** *stmt* | *matched_stmt*

matched_stmt → **if** *expr* **then** *matched_stmt* **else** *stmt* | **other**

试图消除悬空 *else* 的二义性,请证明该文法仍然是二义的。

3.6 为字母表$\Sigma = \{a, b\}$上的下列每个语言设计一个文法,其中哪些语言是正规的?

(a) 每个a后面至少有一个b跟随的所有串。

(b) a和b的个数相等的所有串。

(c) a和b的个数不相等的所有串。

(d) 不含 abb 作为子串的所有串。

*(e) 形式为 xy 且 $x \neq y$ 的所有串。

3.7 可以在文法产生式的右部使用类似正规式的算符。方括号可以用来表示产生式的可选部分,例如可以用

$stmt \rightarrow$ **if** $expr$ **then** $stmt$ [**else** $stmt$]

表示 else 子句是可选的。通常,$A \rightarrow \alpha\ [\beta\ ]\gamma$ 等价于两个产生式 $A \rightarrow \alpha\beta\gamma$ 和 $A \rightarrow \alpha\gamma$。

花括号用来表示短语可重复出现若干次(包括零次),例如

$stmt \rightarrow$ **begin** $stmt$ { ; $stmt$ } **end**

表示处于 **begin** 和 **end** 之间的、由分号分隔的语句表。通常,$A \rightarrow \alpha\{\beta\}\gamma$ 等价于 $A \rightarrow \alpha B\gamma$ 和 $B \rightarrow \beta B \mid \varepsilon$。

概念上,[β]代表正规式 $B \mid \varepsilon$,{β }代表 β^*,现在把它们推广为允许文法符号的任何正规式出现在产生式的右部。

(a) 修改上面的 $stmt$ 产生式,使得每个语句都以分号终止的语句表出现在产生式右部。

(b) 给出上下文无关的产生式,它和 $A \rightarrow B^* a\ (C \mid D\ )$ 产生同样的串集。

(c) 说明如何用一组有限的上下文无关产生式来代替产生式 $A \rightarrow \gamma$,其中 γ 是正规式。

3.8 (a) 消除习题 3.1 文法的左递归。

(b) 为(a)的文法构造预测分析器。

3.9 为习题 3.3 的文法构造预测分析器。

3.10 构造下面文法的 LL(1) 分析表。

$D \rightarrow TL$

$T \rightarrow$ **int** | **real**

$L \rightarrow$ **id** R

$R \rightarrow$, **id** $R \mid \varepsilon$

3.11 构造下面文法的 LL(1) 分析表。

$S \rightarrow aBS \mid bAS \mid \varepsilon$

$A \rightarrow bAA \mid a$

$B \rightarrow aBB \mid b$

3.12 下面的文法是否为 LL(1) 文法,说明理由。

$S \rightarrow AB \mid PQx \qquad A \rightarrow xy \qquad B \rightarrow bc$

$P \rightarrow dP \mid \varepsilon \qquad Q \rightarrow aQ \mid \varepsilon$

*3.13 证明左递归的文法不是 LL(1) 文法。

*3.14 证明 LL(1) 文法不是二义的。

3.15 证明没有 ε 产生式的文法,只要每个非终结符的各个选择以不同的终结符开始,那么它就是 LL(1) 的。

3.16 (a) 用习题 3.1 的文法构造$(a, (a, a))$的最右推导,说出每个右句型的句柄。

(b) 给出对应(a)的最右推导的移进-归约分析器的步骤。

(c) 对照(b)的移进-归约,给出自下而上构造分析树的步骤。

3.17 给出接受文法

$S \to (L) \mid a \qquad L \to L, S \mid S$

的活前缀的一个 DFA。

3.18 为习题 3.3 的文法构造 SLR 分析表。

3.19 考虑下面的文法:

$E \to E+T \mid T$

$T \to TF \mid F$

$F \to F^{*} \mid a \mid b$

(a) 为此文法构造 SLR 分析表。

(b) 构造 LALR 分析表。

3.20 证明下面的文法:

$S \to SA \mid A$

$A \to a$

是 SLR(1) 文法,但不是 LL(1) 文法。

3.21 (a) 证明下面文法:

$S \to AaAb \mid BbBa$

$A \to \varepsilon$

$B \to \varepsilon$

是 LL(1) 文法,但不是 SLR(1) 文法。

*(b) 证明所有 LL(1) 文法都是 LR(1) 文法。

3.22 证明下面文法:

$S \to Aa \mid bAc \mid dc \mid bda$

$A \to d$

是 LALR(1) 文法,但不是 SLR(1) 文法。

3.23 说明每个 SLR(1) 文法都是 LALR(1) 文法。

3.24 证明下面文法:

$S \to Aa \mid bAc \mid Bc \mid bBa$

$A \to d$

$B \to d$

是 LR(1) 文法,但不是 LALR(1) 文法。

3.25 一个非 LR(1) 的文法如下:

$L \to MLb \mid a$

$M \to \varepsilon$

请给出所有含移进-归约冲突的规范 LR(1)项目集,以说明该文法确实不是 LR(1)的。

3.26 (a) 通过构造识别活前缀的 DFA 和构造分析表,来证明文法 $E \to E+\mathbf{id} \mid \mathbf{id}$ 是 SLR(1)文法。

(b) 下面左右两个文法都和(a)的文法等价:

$E \to E+M\ \mathbf{id} \mid \mathbf{id}$　　　　$E \to ME + \mathbf{id} \mid \mathbf{id}$

$M \to \varepsilon$　　　　$M \to \varepsilon$

请指出其中有几个文法不是 LR(1)文法,并给出它们不是 LR(1)文法的理由。

3.27 文法 G 的产生式如下:

$S \to I \mid R$　　　　$I \to d \mid I\,d$　　　　$R \to WpF$

$W \to Wd \mid \varepsilon$　　　　$F \to Fd \mid d$

(a) 令 d 表示任意数字,p 表示十进制小数点,那么非终结符 S, I, R, W 和 F 在编程语言中分别表示什么?

(b) 该文法是 LR(1)文法吗? 为什么?

3.28 下面文法不是 LR(1)的,对它略作修改,使之成为一个等价的 SLR(1)文法。

PROGRAM→ **begin** DECLIST semicolon STATELIST **end**

DECLIST→ d semicolon DECLIST | d

STATELIST→ s semicolon STATELIST | s

3.29 (a) 为下面文法构造规范 LR(1)分析表,画出像图 3.20 这样的状态转换图就可以。

$S \to V = E \mid E$

$V \to *E \mid \mathbf{id}$

$E \to V$

(b) 上述状态转换图有同心项目集吗? 若有,合并同心项目集后是否会出现动作冲突?

3.30 描述文法

$S \to aSbS \mid aS \mid \varepsilon$

产生的语言,并为此语言写一个 LR(1)文法。

3.31 下面两个文法中哪一个不是 LR(1)文法,并对非 LR(1)的那个文法给出含有移进-归约冲突的规范的 LR(1)项目集。

$S \to aAc$　　　　$S \to aAc$

$A \to Abb \mid b$　　　　$A \to bAb \mid b$

3.32 现有字母表 $\Sigma = \{a\}$,写一个和正规式 a^* 等价的上下文无关文法,要求所写的文法既不是 LR 文法,也不是二义文法。

3.33 为语言

$L = \{a^m b^n \mid 0 \leq m \leq 2n\}$(即 a 的个数不超过 b 的个数的两倍)

写三个文法,它们分别是 LR(1)的、二义的和非二义且非 LR(1)的。

3.34 为语言

$L = \{ w \mid w \in (a \mid b)^*$ 并且在 w 的任何前缀中，a 的个数不少于 b 的个数$\}$
写三个文法，它们分别是 LR(1)的、二义的和非二义且非 LR(1)的。

3.35 为习题 3.4 的文法构造 SLR(1)分析表，分析动作冲突的解决要保证正规式能以正常的方式分析。

3.36 由于文法二义引起的 LR(1)分析动作冲突，可以依据消除二义的规则而得到该文法的 LR(1)分析表，根据此表可以正确识别输入串是否为相应语言的句子。对于非二义非 LR(1)文法引起的 LR(1)分析动作的冲突，是否也可以依据一定的规则来消除这种分析动作的冲突而得到 LR(1)分析表，并且根据此表识别相应语言的句子？若可以，你是否能给出这样的规则？

3.37 下面是一个二义文法：

$S \rightarrow AS \mid b$

$A \rightarrow SA \mid a$

如果为该文法构造 LR 分析表，则一定存在某些有分析动作冲突的条目，它们是哪些？假定分析表这样来使用：出现冲突时，不确定地选择一个可能的动作。给出对于输入 *abab* 所有可能的动作序列。

3.38 下面是类型表达式的语法：

type → **integer** | **boolean** | **array**[**num**] **of** *type* | **record** *field_list* **end** | ↑ *type*

field_list → **id** : *type* | **id** : *type* ; *field_list*

若规定：在记录类型中不能出现数组类型（包括不能出现数组的指针类型）。请重新设计一个文法，把该约束体现在文法中，即它和上述文法的区别就是所定义的语言满足该约束。

*3.39 为排版文法(3.16)构造一个等价的 LR 文法，它能把形式如 *E* **sub** *E* **sup** *E* 的表达式处理为一种特殊情况。

3.40 写一个 Yacc 程序，它把输入的算术表达式翻译成对应的后缀表达式输出。

3.41 写一个 Yacc“计算器”程序，它计算布尔表达式。

3.42 写一个 Yacc 程序，它取正规式作为输入，产生它的分析树作为输出。

3.43 对于例 3.16 和例 3.38 的预测分析器和 LR 分析器，追踪它们面临下面有语法错误的输入时的动作。

(a) (**id**+(* **id**)

(b) * +**id**)+(**id** *

*3.44 为下面的文法构造有短语级错误恢复的 LR 分析器。

stmt → **if** *e* **then** *stmt*
  | **if** *e* **then** *stmt* **else** *stmt*
  | **while** *e* **do** *stmt*
  | **begin** *list* **end**
  | *s*

list → *list* ; *stmt*

| *stmt*

3.45 对于一段C语言代码:

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

基于LALR(1)方法的一个编译器的报错情况如下:

如果缺少第1行的逗号,编译器扫描第1行时报告 expected ';', ',' or ')' before 'long'。如果缺少第2行的右括号,编译器扫描第3行时报告 expected ')' before 'return'。这两个示例表明LALR(1)方法能及时发现错误,且不会把出错点后面的符号移进分析栈(活前缀性质)。

如果第2行的if误写成fi,编译器扫描第3行时报告 expected ';' before 'return',扫描第4行时报告'else' without a previous 'if'。此时是否违反了活前缀性质?

第 4 章

语法制导的翻译

在 3.7 节用 Yacc 写的例子中，大家看到了一种有用的描述形式：将语法构造的属性附加在代表语法构造的文法符号上，这些属性值由伴随着文法产生式的语义动作来计算，而语义动作的计算是在对应产生式被归约时进行，由此得到结果。这种描述形式可用来描述编译器的语义分析和中间代码生成等，因此本章系统地研究这种被称为“语法制导的语言翻译”的描述方法及其实现。它的语义动作（有时称为语义规则）的计算可以产生代码、把信息存入符号表、显示出错信息或完成其他工作。语义规则的计算结果就是所期望的记号流的翻译。

本章讨论语义规则和产生式相联系的两种方式：语法制导的定义和语法制导的翻译方案（后者简称翻译方案）。语法制导定义是比较抽象的翻译规范，它隐蔽了一些实现细节；而翻译方案多陈述了一些实现细节，主要是指明各语义规则的计算时机。在第 5 章规范语义检查和第 7 章描述中间代码生成时，大量使用这两种方法。

本章还讨论语法制导定义和翻译方案的实现方法。概念上的方法是，首先分析输入的记号串，建立分析树，然后从分析树得到描述结点属性间依赖关系的有向图，从这个依赖图得到语义规则的计算次序，然后进行计算，最终得到翻译的结果。实际的实现并不需要按上面步骤逐步进行，本章将讨论几种不同场合下的实现方法。

4.1 语法制导的定义

语法制导的定义是带属性和语义规则的上下文无关文法，其中每个文法符号有一组属性，每个产生式有一组语义规则。如果 X 是文法符号，a 是它的一个属性，则 $X.a$ 指称 X 的属性 a 的值。若分析树的结点用记录或对象来实现，那么 X 的属性 a 就可以用代表 X 的记录或对象的数据域 a 来实现。显然，分析树上不同的 X 结点有各自的 a 属性。属性可以表示任何东西，如串、数值、类型、表引用，或其他想表示的东西。分析树结点的属性由该结点所用产生式的语义规则来定义。在语法制导定义中，其中的文法被称为**基础文法**。

本节介绍语法制导定义的形式及其概念上的实现模型。

4.1.1 语法制导定义的形式

在语法制导定义中,每个文法符号有一组属性,每个文法产生式 $A\to\alpha$ 有一组形式为 $b=f(c_1,c_2,\cdots,c_k)$ 的语义规则,其中 f 是函数,b 和 $c_1,c_2,\cdots,c_k$ 是该产生式的文法符号的属性,该规则定义属性 b,并且:

(1) 如果 b 是 A 的属性,$c_1,c_2,\cdots,c_k$ 是产生式右部文法符号的属性或 A 的其他属性,那么 b 称为 A 的**综合属性**。

(2) 如果 b 是产生式右部某个文法符号 X 的属性,$c_1,c_2,\cdots,c_k$ 是 A 的属性或右部文法符号的属性,那么 b 称为 X 的**继承属性**。

在这两种情况下,都说属性 b 依赖于属性 $c_1,c_2,\cdots,c_k$。每个文法符号的综合属性集和继承属性集的交集应为空。一般来说,一个结点的综合属性的值是通过分析树中其子结点的属性值来计算;继承属性的值由结点的兄弟结点、父结点和自己的属性值来计算。终结符只有综合属性,它由词法分析器提供,即记号的属性。因此在语法制导定义中没有计算终结符的语义规则。

语义规则的函数 f 通常是表达式。在实际使用的语法制导定义中,经常还会出现一些规则的目的是打印值、输出中间代码或修改全程量等,它们是一些产生副作用的操作。这样的语义规则写成过程调用或程序段。可以把它们想象成定义了产生式左部非终结符的一个虚拟综合属性,这个虚拟属性和赋值符号=在该规则中没有显式表示出来。

属性文法是指语义规则函数无副作用的语法制导定义。本书大部分语法制导定义都使用有副作用的语义规则。

例 4.1 表 4.1 的语法制导定义表示一个计算器程序。这个定义分别给非终结符 E、T 和 F 一个存放整数值的综合属性 *val*。对每个 E、T 和 F 产生式,语义规则从产生式右部非终结符的属性值 *val* 或 **digit** 的 *lexval* 来计算产生式左部非终结符的属性值 *val*。

表 4.1 简单计算器的语法制导定义

产生式	语义规则
$L\to E\mathbf{n}$	$print(E.val)$
$E\to E_1+T$	$E.val=E_1.val+T.val$
$E\to T$	$E.val=T.val$
$T\to T_1*F$	$T.val=T_1.val\times F.val$
$T\to F$	$T.val=F.val$
$F\to(E)$	$F.val=E.val$
$F\to\mathbf{digit}$	$F.val=\mathbf{digit}.lexval$

记号 **digit** 有综合属性 *lexval*,它的值由词法分析器提供。在产生式 $L \to E\mathbf{n}$ 中,**n** 是表达式的结束标记(如换行符)。该产生式的语义动作比较特殊,它是打印由 E 产生的算术表达式的值,这被看成定义了 L 的一个虚拟属性。这个计算器的 Yacc 规范在图 3.24 已给出,目的是用来描述 LR 分析时的翻译。 □

4.1.2 综合属性

综合属性在实践中有广泛应用。仅仅使用综合属性的语法制导定义称为 ***S* 属性定义**。对于 S 属性定义,分析树各结点属性的计算可以自下而上地完成:从叶结点到根结点,通过计算语义规则而得到结点的属性。

每个结点的属性值都标注出来的分析树叫做**注释分析树**(annotated parse tree),计算各结点属性值的过程叫做分析树的**注释**或**修饰**。

例 4.2 例 4.1 的 S 属性定义规范一种计算器。例如,若输入是表达式 8+5*2,并跟随一个换行符 **n** 时,那么该计算器打印值 18。图 4.1 是 8+5*2**n** 的注释分析树,在树的根结点打印 18。

为了明白属性值是怎么计算的,考虑最左边最底下的分支结点,它使用的产生式是 $F \to \mathbf{digit}$,对应的语义规则是 $F.val = \mathbf{digit}.lexval$。从该规则得到此 F 结点的属性 *val* 为 8,因为它的子结点 **digit** 的 *lexval* 是 8。同样地,该结点的父结点的属性 $T.val$ 也为 8。

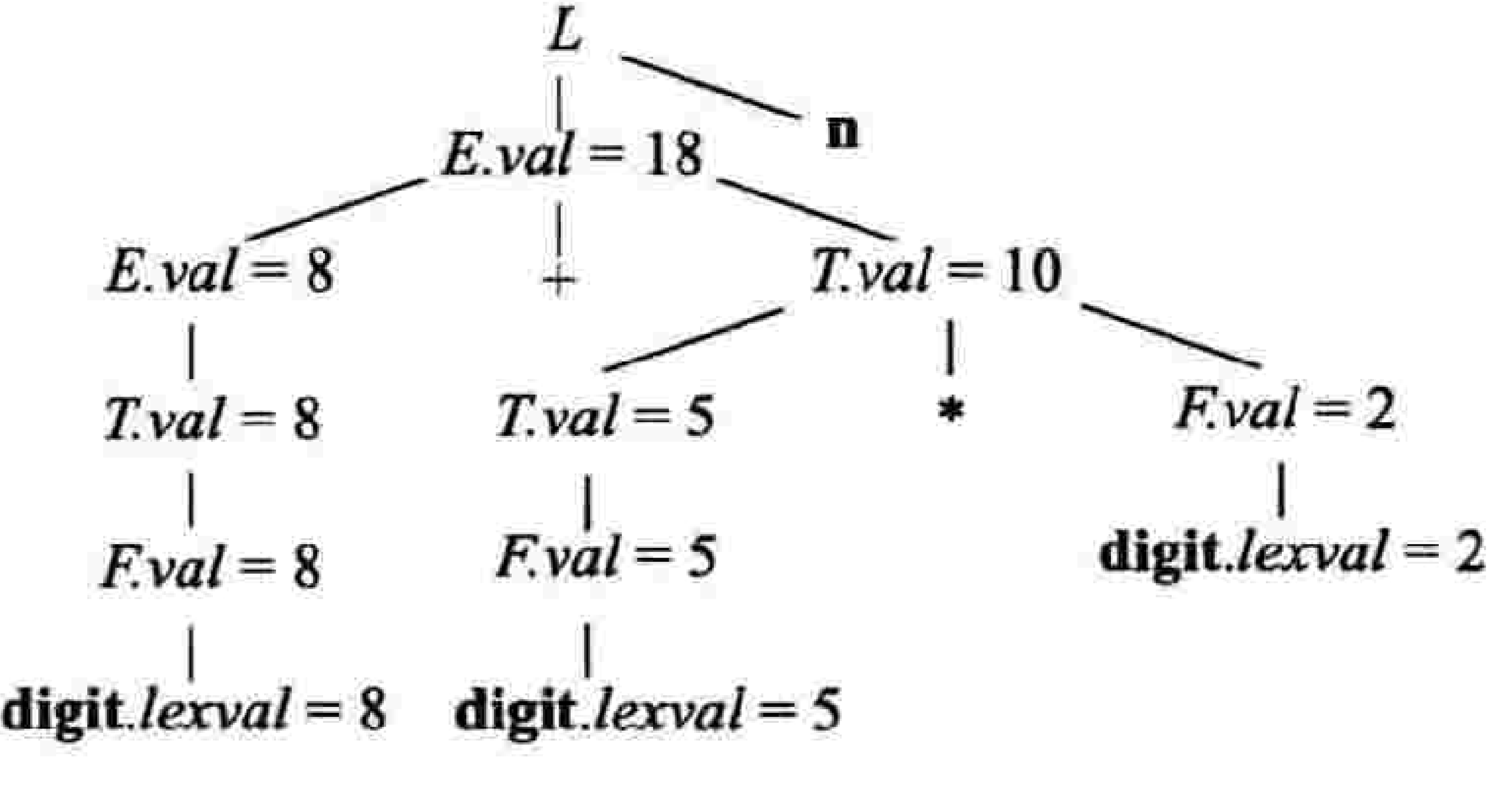


图 4.1 8+5*2**n** 的注释分析树

再以产生式 $E \to E + T$ 的结点为例,左部 E 结点的属性 *val* 由产生式 $E \to E_1 + T$ 的语义规则 $E.val = E_1.val + T.val$ 来定义。当在这个结点运用该语义规则时,子结点 E_1 和 T 的 *val* 分别为 8 和 10,故在此结点求得 $E.val$ 的值为 18。 □

4.1.3 继承属性

在分析树中,一个结点的继承属性是由它的兄弟结点、父结点和自己的属性来定义的。编程语言的一些构造的属性依赖于它们所在的上下文,此时使用继承属性是方便的。在下面的例子中,继承属性将类型信息传递给一张声明表中的各个标识符。

例 4.3 在表 4.2 的语法制导定义中,非终结符 D 产生的声明由关键字 **int** 或 **real** 及一张标识符表组成。非终结符 T 有综合属性 *type*,它的值由声明中的关键字决定。产生式 $D \to TL$ 的语义规则置 L 的继承属性为声明中的类型。产生式 $L \to L_1, \mathbf{id}$ 的语义规则 $L_1.in = L.in$ 把继承属性 *in* 沿分析树向下传递类型。L 产生式的规则调用过程 *addType*,把类型信息加到符号表中各个标识符的条目(属性 *entry* 给出条目的入口)中。

表 4.2 有继承属性的语法制导定义

产生式	语义规则
$D \to TL$	$L.in = T.type$
$T \to$ **int**	$T.type = integer$
$T \to$ **real**	$T.type = real$
$L \to L_1$, **id**	$L_1.in = L.in$　　$addType$ (**id**. $entry$, $L.in$)
$L \to$ **id**	$addType$ (**id**. $entry$, $L.in$)

图 4.2 给出句子 **int** $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$ 的注释分析树(仅注释了部分属性)。这些属性值的计算次序是,首先计算根的左子结点的属性 $T.type$,然后在根的右子树中自上而下地计算三个 L 结点的 $L.in$。在每个 L 结点还调用过程 $addType$,在符号表中将 **id** 子结点上标识符的类型记为整型。 □

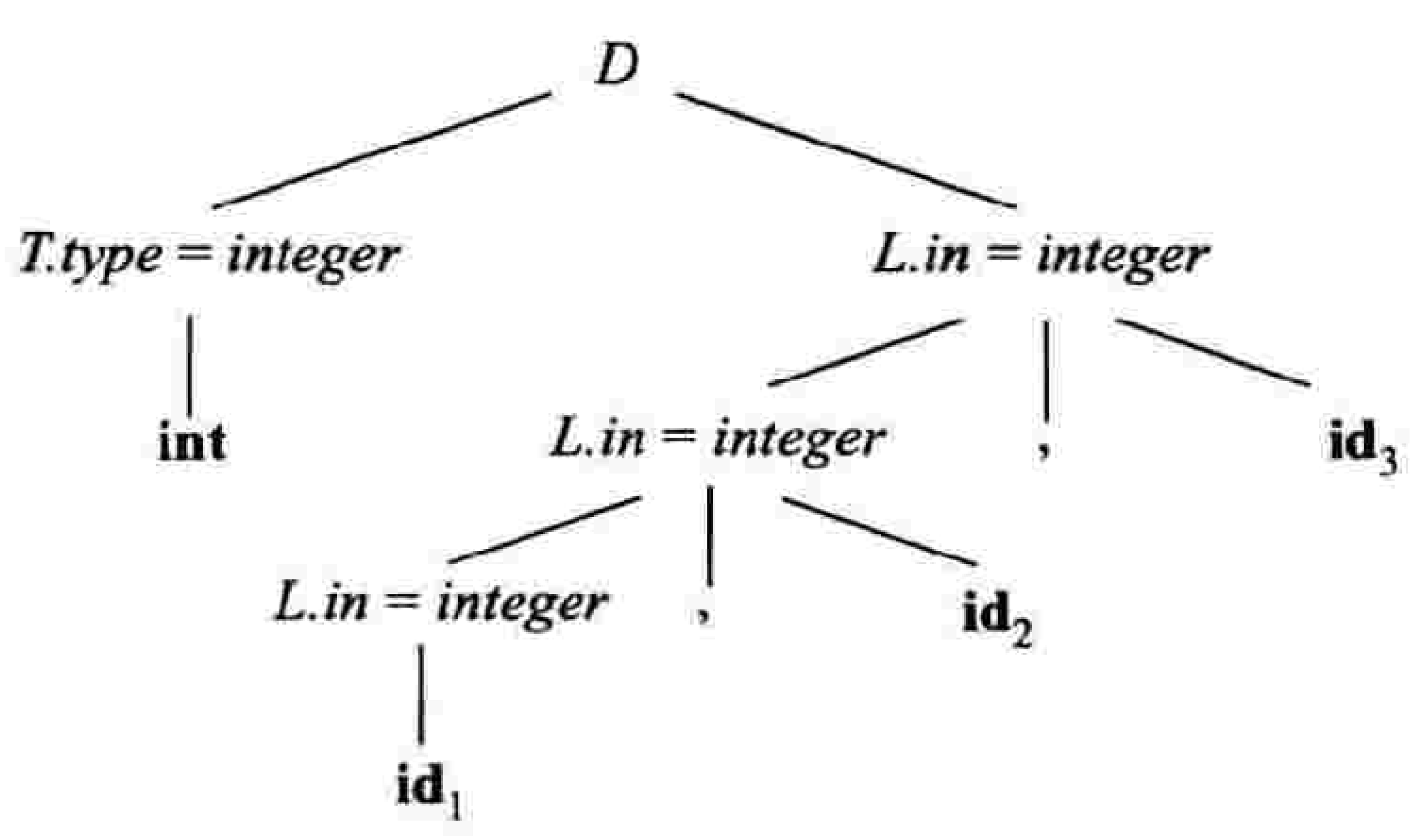


图 4.2 **int** $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$ 的注释分析树

使用继承属性的地方很多,例如,可以用继承属性来记住标识符是出现在赋值号的左边还是右边,以便决定是需要它的地址还是需要它的值。

重写语法制导定义使之仅使用综合属性总是可能的,但有时会使得重写后的文法失去了简洁和直观,反而不如使用带继承属性的语法制导定义自然。

4.1.4 属性依赖图

如果分析树上一个结点的属性 b 依赖某个结点的属性 c,那么定义属性 b 的语义规则的计算必须在定义属性 c 的语义规则的计算之后。分析树结点的属性之间的互相依赖可以用一种称为**依赖图**的有向图来描绘。

在构造分析树的依赖图之前,先为由过程调用组成的语义规则引入虚拟综合属性 b,使得每条语义规则都能写成 $b = f(c_1, c_2, \cdots, c_k)$ 的形式。依赖图的组成是这样的:分析树上每个结点的每个属性在依赖图上都有一个结点,如果属性 b 依赖于属性 c,那么从 c 的结点到 b 的结点有一条有向边。

例如,若 $S.s = f(A.a, B.b, C.c)$ 是产生式 $S \to ABC$ 的语义规则,它定义了依赖于属性 $A.a$、$B.b$ 和 $C.c$ 的综合属性 $S.s$。如果该产生式出现在分析树上,那么,分析树上它的每个实例在依赖图中都有对应的 4 个结点 $S.s$、$A.a$、$B.b$ 和 $C.c$,并且结点 $A.a$、$B.b$ 和 $C.c$ 分别有边到结点 $S.s$。

如果产生式 $S \to ABC$ 的语义规则是 $A.a = f(S.s, B.b, C.c)$,那么在依赖图上,结点 $S.s$、$B.b$

和 $C.c$ 分别有边到结点 $A.a$,因为 $A.a$ 依赖于 $S.s$、$B.b$ 和 $C.c$。

例 4.4 图 4.3 给出了图 4.2 分析树的依赖图。在图 4.3 中,虚线表示的是分析树;依赖图的结点用数字表示,边用实线表示。从结点 4 的 $T.type$ 到结点 5 的 $L.in$ 有一条边,因为根据产生式 $D \to TL$ 的语义规则 $L.in = T.type$,$L.in$ 依赖于 $T.type$。产生式 $L \to L_1$,**id** 的语义规则 $L_1.in = L.in$ 导致 $L_1.in$ 依赖于 $L.in$,因此依赖图上有分别到达结点 7 和 9 的两条向下的边。L 产生式的语义规则 *addType*(**id**.*entry*, $L.in$)导致对应虚拟属性的结点,结点 6,8,10 是这样的虚拟属性结点。 □

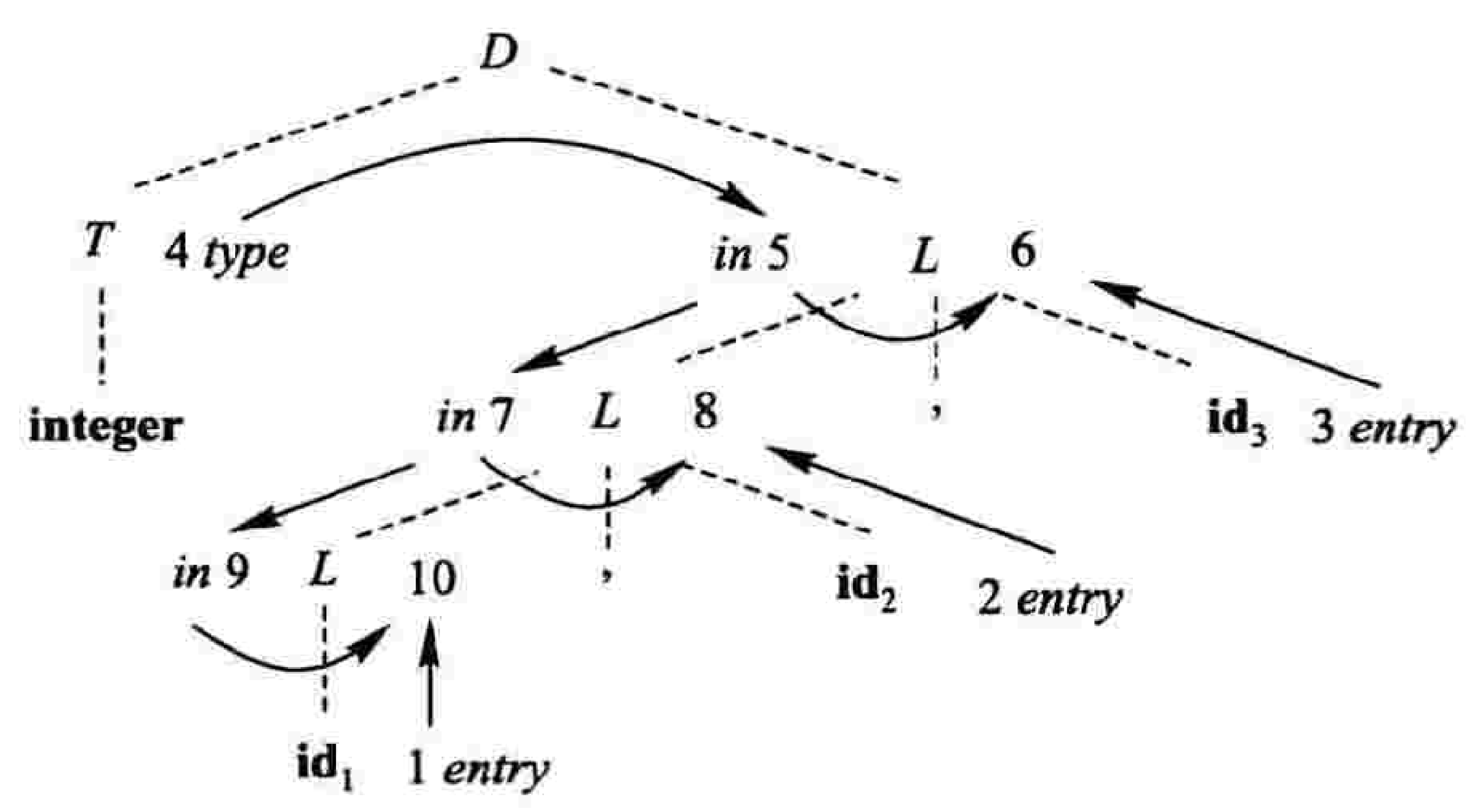


图 4.3 图 4.2 分析树的依赖图

4.1.5 属性计算次序

拓扑排序是有向无环图的结点的一种排序 $m_1, m_2, \cdots, m_k$,它使得有向边只会从这个排序中先出现的结点到后出现的结点,也就是若 $m_i \to m_j$ 是从 m_i 到 m_j 的边,那么在此排序中 m_i 先于 m_j。

显然,依赖图的任何拓扑排序都是分析树中结点属性计算的一个正确次序,即按拓扑排序进行计算的话,用语义规则 $b = f(c_1, c_2, \cdots, c_k)$ 计算 b 时,属性 $c_1, c_2, \cdots, c_k$ 已经计算出来了。

这样,由语法制导定义规范的翻译可以准确地按下述步骤完成:

(1) 首先根据基础文法构造输入的分析树;

(2) 按上面讨论的方法构造属性依赖图;

(3) 对依赖图的结点进行拓扑排序,得到语义规则的计算次序;

(4) 按这个次序计算属性,得到输入串的翻译。

例 4.5 在图 4.3 所示的依赖图中,每条边从序号较低的结点到序号较高的结点,因此结点 1,2,⋯,10 构成该图的一个拓扑排序。把依赖图中序号为 n 的结点的属性写成 a_n,那么从这个拓扑排序可以得下面的程序:

$a_4 = integer$;

$a_5 = a_4$;

$addType\ (\mathbf{id}_3.entry, a_5)$;

$a_7 = a_5$;

$addType\ (\mathbf{id}_2.entry, a_7)$;

$a_9 = a_7$;

$addType\ (\mathbf{id}_1.entry, a_9)$;

这些语义规则的计算把类型 *integer* 存于符号表中每个标识符的条目中。 □

语义规则的这种计算方法称为**分析树方法**。若依赖图有环,则这种方法失败。这种方法的缺点是编译速度很慢,因为它是在编译过程中确定属性的计算次序。显然,要想提高编译速度,必须在编译前,即在构造编译器的时候就把属性计算次序确定下来,避免编译时构造依赖图和拓扑排序等工作。下面概述的两种方法分别是从手工构造编译器和自动生成编译器的角度考虑的改进。

在构造编译器时,用专门的工具或用手工对产生式的语义规则进行分析,对每个产生式,得到与它相关联的一组语义规则的计算次序。据此,把计算次序在编译前就确定下来。编译时,分析树上结点属性的计算就按事先确定的次序进行。这种方法称为**基于规则的方法**,它适用于手工构造编译器。这种方法的缺点是,一些属性依赖关系复杂的语法制导定义很难事先确定属性计算次序,因此这种方法对语法制导定义的种类有限制。

Yacc 等编译器生成工具都有确定的属性计算策略,如果编译器的设计者使用这样的工具来生成编译器,则必须按所选工具规定的计算策略去写语义规则。如果所写的语义规则不符合该计算策略对语义规则的约束,则属性计算不能正确完成。这种方法称为**忽略规则的方法**(oblivious method),即它不是根据语义规则的特点来选择计算策略,而是根据计算策略来限定编译器设计者所提供的语义规则的形式。这种方法大大限制了能够实现的语法制导定义的种类,但是能够得到高效的编译器。

基于规则的方法和忽略规则的方法都不必在编译时显式构造依赖图,和分析树方法相比,它们使编译器的时空效率大大改进。从4.2节开始介绍的各种具体翻译方法都属于这两种方法之一。其中4.2节介绍的综合属性自下而上计算方法最容易理解,也是编译器生成工具提供的基本方法。

4.2 *S*属性定义的自下而上计算

4.1节初步介绍了如何用语法制导定义来规范翻译;从本节开始,将一边熟悉语法制导定义,一边研究其实现。对任意语法制导定义都适用的翻译器很难建立,但对于语法制导定义的一些常用形式,它们的翻译器很容易构造。本节以构造语法树为例,来熟悉 *S* 属性定义,并研究如何在自下而上的分析过程中完成属性计算。

4.2.1 语法树

语法树是分析树的浓缩表示。在语法树中,算符和关键字不是作为叶结点,而是作为分支结点。例如,对于产生式 $S \rightarrow$ **if** B **then** S_1 **else** S_2,可以把它的右部看成是有一个算符 **if-then-else** 和三个运算对象 B、S_1 和 S_2 的语言构造,它的语法树见图 4.4(a)。语法树中另一个简单的地方是单非产生式链可能消失,图 4.4(b)是表达式 8+5*2 的语法树。这两个例子的特点是,语法树上的内部结点都代表运算,其子结点都是它的运算对象。

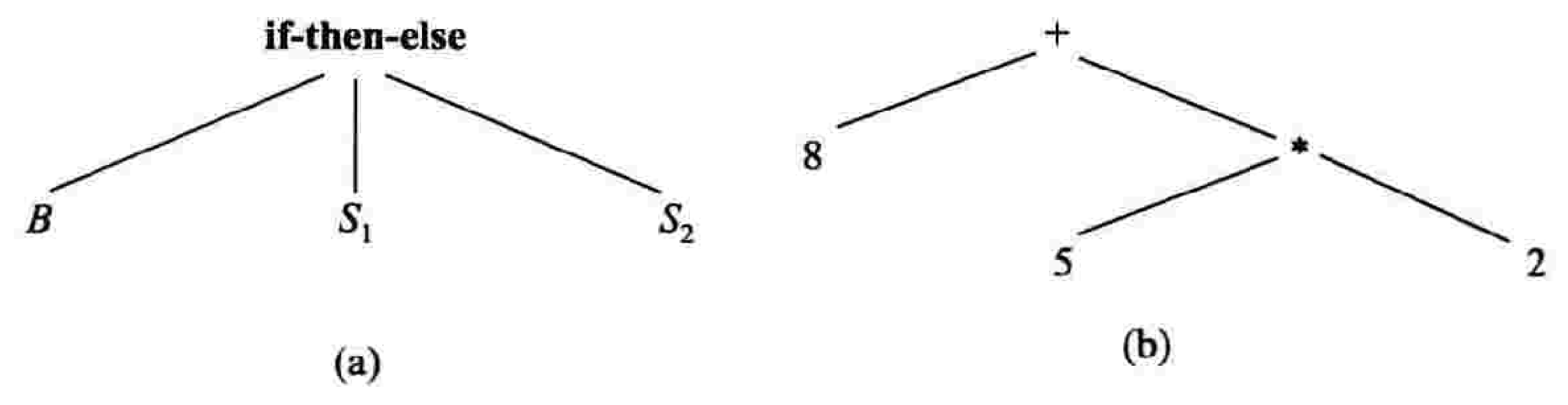


图 4.4 语法树的例子

语法树作为一种中间表示,允许把翻译从分析中分离出来,形成先分析后翻译的方式,即先分析生成语法树,然后再基于语法树进行翻译。即使是边分析边翻译,语法树作为一种概念上的中间表示,也是有用的。C 和 Java 的编译器通常显式构造语法树。

语法制导翻译可以基于分析树,也可以基于语法树,方法是一样的。如同在分析树中那样,在语法树中也可以把属性附加到结点。

4.2.2 构造语法树的语法制导定义

本节给出构造表达式的语法树的语法制导定义。首先介绍一下数据结构,语法树的结点可以用有若干域的记录来实现。对于算符结点,一个域存放算符,该域作为该结点的标记,其余两个域存放指向运算对象的指针。对于基本运算对象结点,一个域存放运算对象类型,另一个域存放其值。当真正用于翻译时,语法树的结点可能还有其他域来保存附加在该结点的其他属性值(或属性值的指针)。

下面解释语义规则中用到的函数,这些函数用来建立语法树的叶结点和分支结点,每个函数都返回新建结点的指针。

(1) *mkLeaf* (**id**, *entry*)。它建立标记为 **id** 的标识符结点;结点另有一个域,其值等于 *entry*,它是符号表中该标识符条目的指针。

(2) *mkLeaf* (**num**, *val*)。它建立标记为 **num** 的整数结点;结点另有一个域,其值等于 *val*,它是该整数的值。

(3) *mkNode* (*op*, *left*, *right*)。它用来建立标记为 *op* 的算符结点;结点另有两个域,其值分别

等于 *left* 和 *right*,它们是该结点左右子树的指针。

表4.3是为含+和 * 的表达式构造语法树的 S 属性定义。它给文法的产生式添加语义规则来安排对函数 *mkNode* 和 *mkLeaf* 的调用,以便构造语法树。E,T 和 F 的综合属性 *nptr* 用来记住函数调用返回的指针。

表4.3 构造表达式语法树的语法制导定义

产生式	语义规则
$E \to E_1 + T$	$E.nptr = mkNode('+', E_1.nptr, T.nptr)$
$E \to T$	$E.nptr = T.nptr$
$T \to T_1 * F$	$T.nptr = mkNode('*', T_1.nptr, F.nptr)$
$T \to F$	$T.nptr = F.nptr$
$F \to (E)$	$F.nptr = E.nptr$
$F \to$ **id**	$F.nptr = mkLeaf$ (**id**, **id**.*entry*)
$F \to$ **num**	$F.nptr = mkLeaf$ (**num**, **num**.*val*)

例4.6 图4.5给出了表达式 $a+5*b$ 的注释分析树和执行语义规则后所构造出的语法树,分析树由带点的线表示。标有非终结符 E,T 和 F 的分析树结点,其综合属性 *nptr* 指向由该非终结符推出的表达式对应的语法树的根结点。

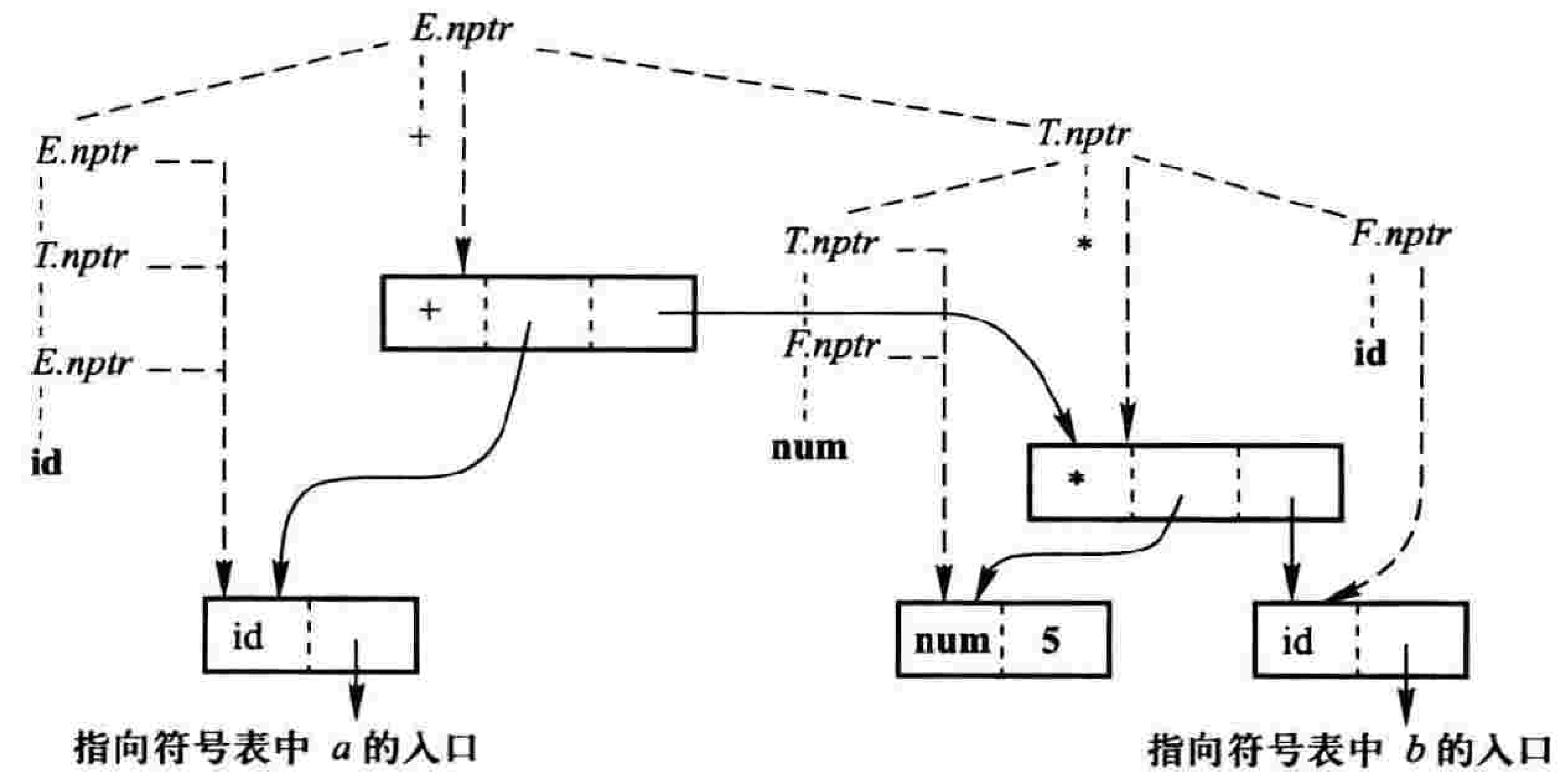


图4.5 $a+5*b$ 的语法树的构造

产生式 $F \to$ **id** 和 $F \to$ **num** 的语义规则定义的属性 $F.nptr$ 也是指针,它们分别指向代表标识符和整数的叶结点。**id**.*entry* 和 **num**.*val* 是对应记号的属性值。

在图4.5中,当表达式 E 只有一项,也就是使用产生式 $E \to T$ 时,属性 $E.nptr$ 和 $T.nptr$ 的值一样。当使用产生式 $E \to E_1 + T$ 的语义规则 $E.nptr = mkNode('+', E_1.nptr, T.nptr)$ 时,先前的规则

已把 $E_1.nptr$ 和 $T.nptr$ 分别置为指向代表 a 的叶结点和代表 $5*b$ 的分支结点。

处于图4.5的下部，由记录形成的树是构成输出的真正语法树；而虚线表示的树是分析树，它可能仅有象征意义。 □

根据表4.3的语法制导定义，对于输入 $a+5*b$，实际执行的一系列函数调用如下：

(1) $p_1 = mkLeaf$ (**id**, *entrya*);

(2) $p_2 = mkLeaf$ (**num**, 5);

(3) $p_3 = mkLeaf$ (**id**, *entryb*);

(4) $p_4 = mkNode$ ('*', p_2, p_3);

(5) $p_5 = mkNode$ ('+', p_1, p_4);

可以看出，写语法制导定义比直接用编程语言写程序更困难。对程序而言，整个程序的执行流程是显式地用语句描述的。而语法制导定义中语义规则的执行受输入的语法制导，当输入串中的一个语法构造被识别时（可以看成发生一个事件），其相应的语义规则被执行，因此本质上这是一种事件驱动的程序设计方法。

4.2.3 S属性的自下而上计算

综合属性可以由自下而上的分析器在分析输入的同时完成计算。分析器可以把文法符号的综合属性放在栈中，每当归约时，根据出现在栈顶的产生式右部符号的属性来计算左部符号的综合属性。下面说明如何扩展分析器的栈，使之能够保存综合属性。在4.4节将看到这种实现也支持一些有继承属性的情况。

S属性定义的翻译器可以借助LR分析器的生成器来实现，例如3.7节讨论的Yacc。分析器的生成器根据S属性定义，可以构造出翻译器，它在分析输入的同时计算属性。

自下而上分析器用栈来保存已分析子树的信息。可以在分析栈中增加一个域来保存综合属性，图4.6给出了一个例子。拓展后分析栈的每个栈元素由状态域 *state* 和属性域 *val* 组成，或者说该栈由 *state* 和 *val* 两部分组成。为直观起见，图4.6的 *state* 部分用文法符号来代替状态，这个符号就是图3.11所描述的分析栈中被状态直接压住的那个符号。*val* 部分为文法符号存放综合属性。如果 *state* 部分某个位置的符号是 A，那么 *val* 部分相应位置保存分析树上对应这个 A 的结点的属性值。

栈顶由指针 *top* 指示，并假定综合属性刚好在每步归约前计算完成。若产生式 $A \rightarrow XYZ$ 的语义规则是 $A.a = f(X.x, Y.y, Z.z)$，那么在 XYZ 归约成 A 之前，属性 $Z.z$ 的值在 *stack*[*top*].*val*（指栈顶元素的 *val* 域），$Y.y$ 的值在 *stack*[*top*-1].*val*（指次栈顶元素的 *val* 域，以下类推），$X.x$ 的值在 *stack*[*top*-2].*val*。如果某个符号没有属性，那么其对应的 *val* 域没有定义。归约时，覆盖 A

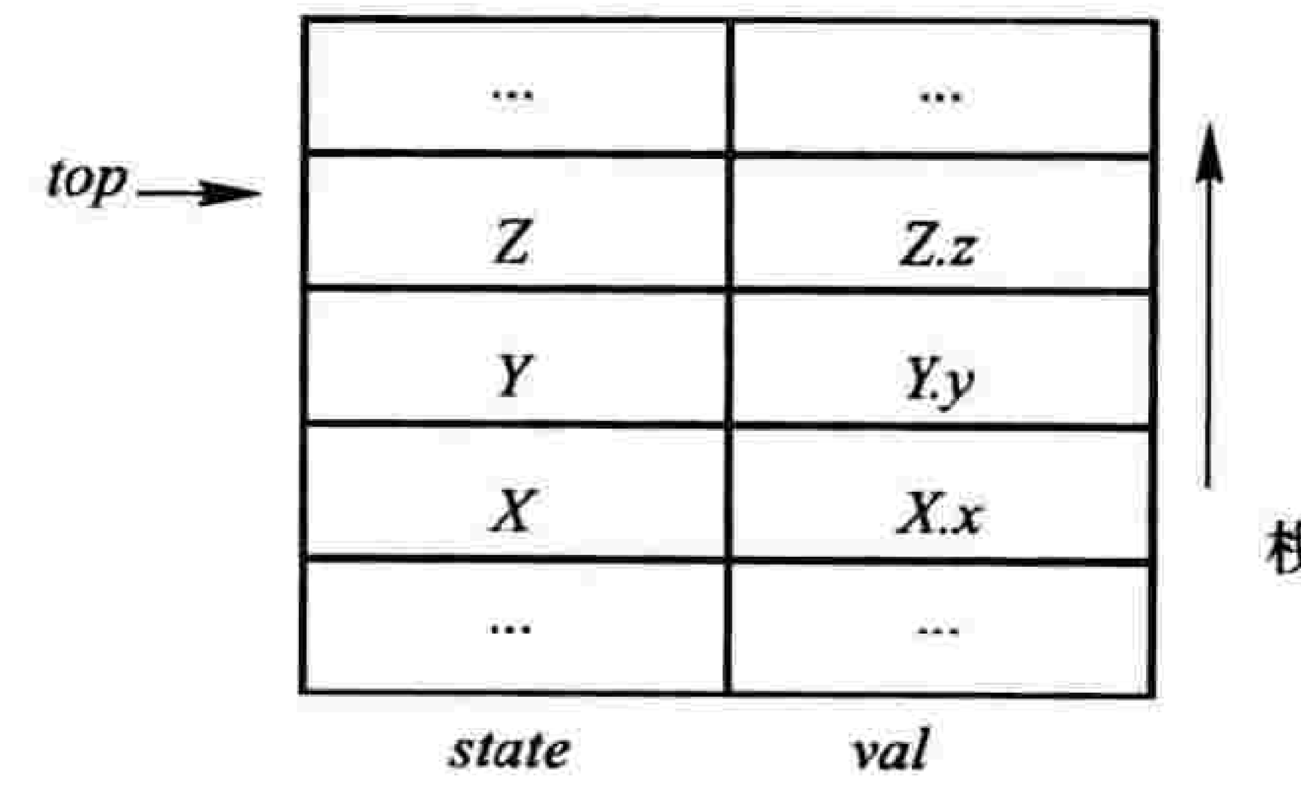


图4.6 有综合属性域的分析栈

的状态放进 *stack*[*top*−2].*state*(即 *X* 原来的位置),综合属性的值 *A*.*a* 放进 *stack*[*top*−2].*val*,然后 *top* 的值减 2。

例 4.7 再次考虑表 4.1 计算器的语法制导定义。用 3.5 节的技术构造基础文法的 LR 分析器。为了计算属性,需要修改分析器,让它在归约前执行表 4.1 的语义规则,这些语义规则可以翻译成表 4.4 的栈操作代码段。这些代码段实际上就是通过用属性在栈中的位置代替表 4.1 语义规则中的属性而得到的。

表 4.4 用 LR 分析器实现计算器

产生式	表 4.1 的语义动作翻译成的代码段
$L \to E$**n**	*print* (*stack*[*top*−1].*val*);
$E \to E_1 + T$	*stack*[*top* −2].*val*=*stack*[*top* −2].*val* + *stack*[*top*].*val*;
$E \to T$	
$T \to T_1 * F$	*stack*[*top* −2].*val*=*stack*[*top* −2].*val* × *stack*[*top*].*val*;
$T \to F$	
$F \to (E)$	*stack*[*top* −2].*val*=*stack*[*top* −1].*val*;
$F \to$ **digit**	

表 4.4 中的代码段中,属性计算的结果都置入 *top*−2 的位置,因为刚好对应产生式的右部都是三个符号。这里的 *top* 指归约前的栈顶,归约后分析器会根据右部符号的多少调整它的值。注意,对应产生式 $E \to T$、$T \to F$ 和 $F \to$ **digit** 没有代码段,因为相应语义规则的翻译结果是 *stack*[*top*].*val*=*stack*[*top*].*val*,显然可以忽略。

表 4.5 展示了面临输入 8+5 * 2**n**(注释分析树在图 4.1)时分析器的动作序列。每步动作后分析栈的 *state* 域和 *val* 域的内容都在表中给出,仍然用对应的文法符号代替状态。为直观起见,还用实际的输入数字代替记号 **digit**。

考虑看见第一个输入符号 8 时的动作序列。首先分析器把对应记号 **digit** 的状态和属性值移进栈(状态由 8 表示,属性值 8 在 *val* 域)。第二步,分析器用产生式 $F \to$ **digit** 归约并执行表 4.4 中对应的代码段。第三步,分析器按 $T \to F$ 归约,执行对应的代码段。第四步,分析器按 $E \to T$ 归约,执行对应的代码段。注意,没有代码段和这三个产生式相关联,所以 *val* 域一直不变,但每步归约后栈顶 *val* 域代表归约后产生式左部符号的属性。 □

表 4.5 翻译器面临 8+5 * 2n 时的动作

输入	*state*	*val*	所用产生式
8+5 * 2**n**	−	−	
+5 * 2**n**	8	8	

续表

输入	*state*	*val*	所用产生式
+5 * 2**n**	*F*	8	*F*→**digit**
+5 * 2**n**	*T*	8	*T*→*F*
+5 * 2**n**	*E*	8	*E*→*T*
5 * 2**n**	*E*+	8+	
* 2**n**	*E*+5	8+5	
* 2**n**	*E*+*F*	8+5	*F*→**digit**
* 2**n**	*E*+*T*	8+5	*T*→*F*
2**n**	*E*+*T* *	8+5 *	
n	*E*+ *T* * 2	8 + 5 * 2	
n	*E*+ *T* * *F*	8 + 5 * 2	*F*→**digit**
n	*E*+ *T*	8 + 10	*T*→*T* * *F*
n	*E*	18	*E*→*E* + *T*
	E **n**	18-	
	L	18	*L*→*E* **n**

在上面的实现轮廓中,代码段刚好在归约前执行。归约提供一种"挂钩",任何代码段组成的动作都可以悬挂在上面,即允许用户把代码和产生式联系起来,当按此产生式归约时执行这些代码。

本节实际上揭示了3.7节介绍的生成器Yacc的语义动作的实现方法,对今后使用Yacc和类似的生成器会有很大帮助。

4.3 *L*属性定义的自上而下计算

从4.2节可知,将LR分析器进行拓展,很容易把*S*属性定义所要求的翻译嵌在分析过程中完成。那么,这种边分析边翻译的方式能否适用于有继承属性的情况呢?

若翻译在分析时发生,属性的计算次序一定受到分析方法所限定的分析树结点建立次序的限制。不管是自上而下分析还是自下而上分析,一个共同的特点是,分析树的结点总是自左向右地生成。如果属性信息是自左向右流动,那么就有可能在分析的同时完成属性计算。下面按这种想法来确定*L*属性定义,其中*L*表示左,因为属性信息是从左向右流动的。

4.3.1 L 属性定义

语法制导定义是 L 属性的，如果每个产生式 $A \to X_1\ X_2 \cdots\ X_n$ 的每条语义规则计算的属性是 A 的综合属性；或者计算的是 X_j 的继承属性（$1 \leqslant j \leqslant n$），它仅依赖：

（1）该产生式中 X_j 左边符号 $X_1, X_2, \cdots, X_{j-1}$ 的属性；

（2）A 的继承属性。

显然，S 属性定义属于 L 属性定义，因为限制（1）和（2）仅对继承属性进行限制。

例 4.3 有关变量类型声明的语法制导定义是一个 L 属性定义，其中类型信息从左向右流。为了说明问题，把这个语法制导定义重复写在这里，见表 4.6。

表 4.6　有继承属性的语法制导定义

产生式	语义规则
$D \to TL$	$L.in = T.type$
$T \to$ **int**	$T.\ type = integer$
$T \to$ **real**	$T.\ type = real$
$L \to L_1$, **id**	$L_1.in = L.in$　　$addType$ (**id**. $entry$, $L.in$)
$L \to$ **id**	$addType$ (**id**. $entry$, $L.in$)

在这个例子中，如果语义规则都在归约时才执行的话，肯定会出问题。例如产生式 $D \to TL$ 的语义规则是 $L.in = T.type$，在分析由 L 推出的标识符表时，需要用 $L.in$ 的值，等归约该产生式时才对 $L.in$ 赋值的话肯定太晚了。由此可知，需要考虑语义规则的执行时机，以及反映语义规则执行时机的描述方法。下一小节将介绍这种描述方法。

4.3.2 翻译方案

语法制导的翻译方案和语法制导定义不同之处是它的**语义动作**（在此不叫语义规则）放在括号{}内，并且可以插入到产生式右部的任何地方。这是一种动作和分析交错的表示法，以表达动作的执行时机。若 $A \to \alpha\{\cdots\}\beta$，那么{…}中语义动作的执行在 α 的推导（或向 α 的归约）结束以后，在 β 的推导（或向 β 的归约）开始之前。可以把{}之间的语义动作想象成一个文法符号，在分析过程中对该符号进行推导（或归约）的时候，就是该语义动作执行的时候。

本章也使用翻译方案作为描述分析期间翻译的一种方法。

例 4.8　下面是一个简单的翻译方案，它把含加和减算符的中缀表达式翻译成后缀表达式。

$$E \to TR$$
$$R \to \textbf{addop}\ T\ \{print\ (\textbf{addop}.lexeme);\}\ R_1 \mid \varepsilon \qquad (4.1)$$
$$T \to \textbf{num}\ \{print\ (\textbf{num}.val);\}$$

如果输入是 8+5-2,该翻译方案的输出是 8 5 + 2 -。第 2 行的动作 *print* (**addop**. *lexeme*) 必须放在 T 和 R_1 之间,移到别的位置都会导致不正确的结果。 □

设计翻译方案时,必须保证动作在引用属性时其值已经可用,也就是要保证动作不会引用还没有计算出值的属性。

只有综合属性的情况最简单。此时,为每条语义规则建立一个赋值动作,把该动作放在对应产生式右部的末端,由此可以得到翻译方案。

如果同时还有继承属性,则必须仔细斟酌。下面是三条限制,它们的确定是受 L 属性定义的启发:

(1) 产生式右部符号的继承属性必须在先于这个符号的动作中计算。

(2) 一个动作不能引用该动作右边符号的综合属性。

(3) 左部非终结符的综合属性只能在它所引用的所有属性都计算完后才能计算。计算该属性的动作通常放在产生式右部的末端。

对 L 属性语法制导定义,构造满足上面三个限制的翻译方案总是可能的。下面的例子说明这一点,它基于早期的数学排版语言 EQN。对于输入

E sub 1 .val

EQN 编排的 E、1 和 .*val* 的相对位置和相对大小见图 4.7,下标 1 以较小的字体印刷,它的位置也低于 E 和 .*val*。

例 4.9 根据表 4.7 的 L 属性定义,构造图 4.8 的翻译方案。表中非终结符 B 代表公式编排单元,产生式 $B \to BB$ 代表两个单元的并置,$B \to B$ **sub** B 表示第二个单元作为第一个单元的下标,它的编排尺寸比第一个单元的小,并且位置较低。

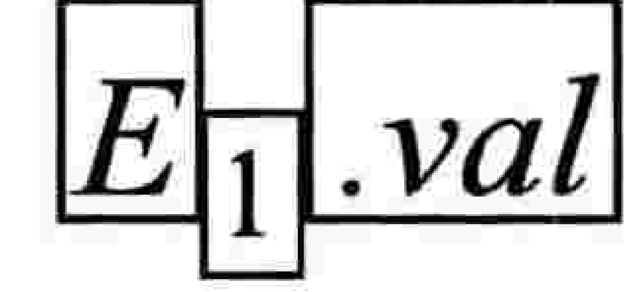


图 4.7 编排单元的排版结果

表 4.7 定义编排单元大小和高度的语法制导定义

产生式	语义规则		
$S \to B$	$B.ps = 10$	$S.ht = B.ht$	
$B \to B_1\ B_2$	$B_1.ps = B.ps$	$B_2.ps = B.ps$	$B.ht = max(B_1.ht, B_2.ht)$
$B \to B_1$ **sub** B_2	$B_1.ps = B.ps$	$B_2.ps = shrink(B.ps)$	$B.ht = disp\ (B_1.ht, B_2.ht)$
$B \to$ **text**	$B.ht = \textbf{text}.h \times B.ps$		

继承属性 *ps* 表示点的大小,它会影响公式的高度。产生式 $B \to$ **text** 的规则用正文的正常高度乘以点的大小以得到正文的实际高度。**text** 的属性 h 可以根据 **text** 的性质查表得到。在产生式 $B \to B_1 B_2$ 的语义规则中,通过复写规则,B_1 和 B_2 从 B 继承了点的大小。B 的高度由综合属性 *ht* 表示,它取 B_1 和 B_2 高度的较大值。

$S \to$ $\{B.ps = 10;\}$
　　B $\{S.ht = B.ht;\}$

$B \to$ $\{B_1.ps = B.ps;\}$
　　B_1 $\{B_2.ps = B.ps;\}$
　　B_2 $\{B.ht = max(B_1.ht, B_2.ht);\}$

$B \to$ $\{B_1.ps = B.ps;\}$
　　B_1
　　sub $\{B_2.ps = shrink(B.ps);\}$
　　B_2 $\{B.ht = disp(B_1.ht, B_2.ht);\}$

$B \to$ **text** $\{B.ht = \textbf{text}.h \times B.ps;\}$

图 4.8　从表 4.7 构造的翻译方案

在产生式 $B \to B_1$ **sub** B_2的语义规则中,函数 *shrink* 将 B_2的点缩小 30%。函数 *disp* 在计算 B 的高度时,把单元 B_2向下偏置。输出编排结果的语义动作在图 4.8 中没有给出。

表 4.7 中,唯一的继承属性是非终结符 B 的 *ps*,定义 *ps* 的语义规则仅依赖产生式左部非终结符的继承属性,所以该定义是 L 属性定义的。

图 4.8 的翻译方案是根据上面三点要求,通过把对应于表 4.7 语义规则的语义动作插入产生式而得到的。为了可读性,把

$$S \to \{B.ps = 10;\}\ B\ \{S.ht = B.ht;\}$$

写成了

$S \to$ $\{B.ps = 10;\}$
　　B $\{S.ht = B.ht;\}$

注意,给继承属性 $B_1.ps$ 和 $B_2.ps$ 赋值的动作刚好在产生式右部 B_1和 B_2的前面。□

大多数算术运算符是左结合的,因此用左递归文法表示表达式是自然的。但是在构造预测翻译器时,必须消除文法中的左递归,而左递归的消除可能会引起继承属性的出现。下面的例子说明了这一点。

例 4.10　如果把表 4.3 构造语法树的语法制导定义变成翻译方案,那么 E 的产生式和语义动作成为

$E \to E_1 + T$ $\{E.nptr = mkNode('+', E_1.nptr, T.nptr);\}$
$E \to T$ $\{E.nptr = T.nptr;\}$

从这个方案消除左递归时,变换后的翻译方案见图 4.9,其中 F 的产生式和语义动作类似于表 4.3 最初定义中的那些产生式和语义动作。

$E \rightarrow$	T	$\{R.i = T.nptr;\}$
	R	$\{E.nptr = R.s;\}$
$R \rightarrow$	$+$	
	T	$\{R_1.i = mkNode\,('+', R.i, T.nptr);\}$
	R_1	$\{R.s = R_1.s;\}$
$R \rightarrow$	ε	$\{R.s = R.i;\}$
$T \rightarrow$	F	$\{W.i = F.nptr;\}$
	W	$\{T.nptr = W.s;\}$
$W \rightarrow$	$*$	
	F	$\{W_1.i = mkNode\,('*', W.i, F.nptr);\}$
	W_1	$\{W.s = W_1.s;\}$
$W \rightarrow$	ε	$\{W.s = W.i;\}$
$F \rightarrow$	(E)	$\{F.nptr = E.nptr;\}$
$F \rightarrow$	**id**	$\{F.nptr = mkLeaf\,(\textbf{id}, \textbf{id}.entry);\}$
$F \rightarrow$	**num**	$\{F.nptr = mkLeaf\,(\textbf{num}, \textbf{num}.val);\}$

图 4.9　构造语法树的翻译方案

图 4.10 给出了图 4.9 的动作是怎样为 $a*5*b$ 构造语法树的。为使图简单起见，在图中略去了 $E \Rightarrow TR \Rightarrow T$ 的部分。综合属性在文法符号的右边，继承属性在文法符号的左边。语法树的叶结点由与产生式 $F \rightarrow$**id** 和 $F \rightarrow$**num** 相关联的动作构造，和例 4.6 一样。最左边的 F，其属性

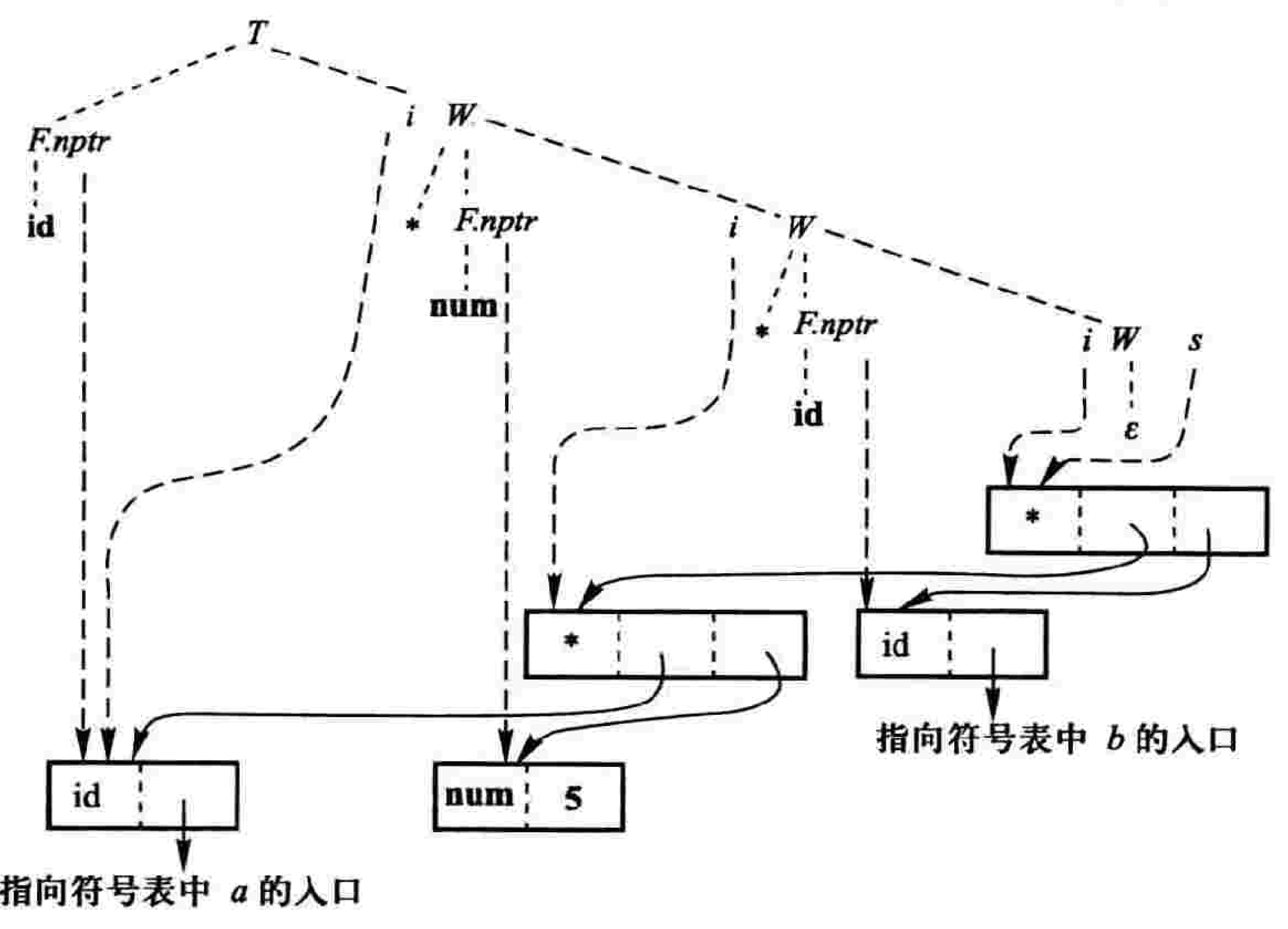


图 4.10　$a*5*b$ 的语法树的构造

F. nptr 指向叶结点 *a*，指向 *a* 的指针由 $T \to FW$ 右部 *W* 的属性 *W. i* 继承。

当产生式 $W \to * FW_1$ 用于根的右子结点时，*W. i* 指向 *a* 结点，*F. nptr* 指向结点 5。现在 *mkNode* 作用于乘算符和这两个指针，构造出对应 $a * 5$ 的结点。

最后使用的产生式是 $W \to \varepsilon$ 时，*W. i* 指向整棵树的根。该指针通过 *W* 结点的 *s* 属性返回（图 4.10 没有画出），直至把它赋给 *T. nptr*（最终赋给 *E. nptr* 不在图 4.10 的范围中）。 □

原来的左递归文法没有继承属性，为什么把文法从左递归改成右递归后会出现继承属性？可以这样直观地解释：对于表 4.3 的文法，属性信息的流动方向和归约方向是一致的，因此一般来说只用综合属性就可以了；而图 4.9 翻译方案的属性信息（语法树结点指针）是从左向右流动的，归约却是从右向左进行的，这种不一致性导致了继承属性的出现。

4.3.3 预测翻译器的设计

本节讨论在预测分析的同时完成 *L* 属性定义。为能明显看出语义动作执行的时机，本节基于翻译方案而不是语法制导定义来讨论。

下面的算法把预测分析器的构造方法推广到翻译方案的实现，该翻译方案的文法必须适用于自上而下分析。

算法 4.1 构造语法制导的预测翻译器。

输入 语法制导的翻译方案，其基础文法适于预测分析。

输出 语法制导翻译器的代码。

方法 修改预测分析器的构造技术。

(1) 为每个非终结符 *A* 构造一个函数，*A* 的每个继承属性声明为该函数的一个形式参数，*A* 的综合属性作为它的返回值（可以是记录，或者是每个属性占一个域记录的指针）。为简单起见，假定每个非终结符正好只有一个综合属性。*A* 的函数还为 *A* 产生式中的其他每个文法符号的每个属性声明一个局部变量。

(2) *A* 函数的代码框架和预测分析过程的一致，主要是根据当前的输入决定使用什么产生式。

(3) 和每个产生式相关联的代码按下面方法生成。产生式右部的记号、非终结符和语义动作被从左向右地依次考虑。

① 对于有属性 *x* 的记号 *X*，把 *x* 的值保存在为 *X. x* 声明的变量中。然后产生匹配记号 *X* 的调用，并推进输入指针。

② 对于非终结符 *B*，产生赋值 $c = B(b_1, b_2, \cdots, b_k)$，它的右部是函数调用，其中 $b_1, b_2, \cdots, b_k$ 是对应 *B* 继承属性的变量，*c* 是代表 *B* 综合属性的变量。

③ 对于每个语义动作，把代码复制到函数体中，把对属性的引用改成对相应的局部变量的引用。 □

例 4.11 图 4.9 的文法是 LL(1) 的，因而适合于自上而下分析。从该文法的非终结符的属性，

可以得到函数 E,R,T,W 和 F 的首部如下。因为 E,T 和 F 没有继承属性,所以它们没有变元。

```
syntaxTreeNode * E( )
syntaxTreeNode * R( syntaxTreeNode * i)
syntaxTreeNode * T( )
syntaxTreeNode * W( syntaxTreeNode * i)
syntaxTreeNode * F( )
```

R 的代码基于图 4.11 的分析过程。如果向前看符号是+,那么使用产生式 $R \to +TR$,先用 match 过程去读+后面的下一个输入记号,然后调用过程 T 和 R。否则该产生式什么也不做,使用的是产生式 $R \to \varepsilon$。

```
void R( ) {
    if (lookahead == '+') {
        match ('+'); T( ); R( );
    }
    else /* 什么也不做 */
}
```

图 4.11　产生式 $R \to +TR \mid \varepsilon$ 的分析过程

图 4.12 的 R 函数含计算属性的代码,有这样几步:记号'+'的值 lexval 存于 addoplexeme,匹配+,调用 T,用 nptr 保存它的结果。变量 i1 对应继承属性 $R_1.i$,s1 对应综合属性 $R_1.s$。**return** 语句返回 s。其他函数可以效仿这个函数去构造。 □

```
syntaxTreeNode * R (syntaxTreeNode * i) {
    syntaxTreeNode  * nptr, * i1 , * s1 , * s;
    char addoplexeme;

    if (lookahead == '+') { /* 产生式 R→+TR */
        addoplexeme = lexval;
        match('+');
        nptr = T( );
        i1 = mkNode( addoplexeme, i, nptr);
        s1 = R (i1);
        s = s1;
    }
    else   s = i; /* 产生式 R→ε */
    return s;
}
```

图 4.12　语法树的递归下降构造

4.3.4 用综合属性代替继承属性

从先前的例子已经看出，S 属性定义比 L 属性定义易于理解。从例4.10还知道，改变文法可能会引入继承属性。同样，改变文法有时可以避免使用继承属性，后者对于构造只允许综合属性的翻译方案极其有用。

例如，Pascal语言的变量声明由标识符表加类型组成，类型出现在变量表的后面，如 m,n : integer。这种声明可以用下面的文法描述：

$D \to L : T$

$T \to$ **integer** | **char**

$L \to L,$ **id** | **id**

在这个文法中，标识符表由 L 产生，但类型不是 L 的子树，因此不能仅靠使用综合属性把类型和标识符联系起来。事实上，在第一个产生式中，由于非终结符 L 从它右边的 T 获得类型信息，所写的语法制导定义将不可能是 L 属性定义的，所以基于它的翻译也不可能在分析期间完成。

这个问题可以通过重新构造文法来解决，把类型作为标识符表的最后一个成分：

$D \to$ **id** L

$L \to ,$ **id** L | $: T$

$T \to$ **integer** | **char**

现在，类型作为 L 的综合属性 $L.type$，当标识符由 L 产生时，它的类型可以填入符号表，具体的翻译方案如下：

$D \to$ **id** L { $addType$ (**id**.$entry$, $L.type$); }

$L \to ,$ **id** L_1 { $L.type = L_1.type$; $addType$ (**id**.$entry$, $L_1.type$); }

$L \to : T$ { $L.type = T.type$; }

$T \to$ **integer** { $T.type = integer$; }

$T \to$ **char** { $T.type = char$; }

可以说，在Pascal声明中，类型信息是从右向左流动的。本小节开始给出的文法是按从左向右归约设计的，因此不可能在分析期间完成把类型信息填入符号表的计算。而修改后的文法是按从右向左归约设计的，类型信息流向和归约方向的一致性使得属性计算可以在分析期间完成。

4.4 *L* 属性的自下而上计算

本节提出在自下而上分析的框架中实现 L 属性定义的方法。它能实现任何基于LL(1)文法的 L 属性定义，也能实现许多(但不是所有的)基于LR(1)的 L 属性定义。该方法是对4.2.3节自下而上翻译技术的推广。本节所讲的一些技术已经应用于Yacc或类似的生成器中。

4.4.1 删除翻译方案中嵌入的动作

4.2.3 节的自下而上翻译方法适用于 S 属性定义，S 属性定义的语义动作都可以放在产生式的右端，在归约时执行。虽然 L 属性定义的继承属性计算需要嵌在产生式右部的不同地方，但是可以通过修改文法而把翻译方案中所有的嵌入动作都变换成只出现在产生式的右端，当嵌入动作只涉及虚拟属性时，这种变换尤其容易。

这种变换在文法中加入推出空串的**标记非终结符**，让每个嵌入动作由不同标记非终结符 M 代表，并把该动作放在产生式 $M \to \varepsilon$ 的右端。例如，使用标记非终结符 M 和 N，(4.1)翻译方案

$E \to TR$

$R \to +T$ {*print* ('+');} R_1 | $-T$ {*print* ('-');} R_1 | ε

$T \to$ **num** {*print* (**num**.*val*);}

变换成

$E \to TR$

$R \to +TMR_1$ | $-TNR_1$ | ε

$T \to$ **num** {*print* (**num**.*val*);}

$M \to \varepsilon$ {*print* ('+');}

$N \to \varepsilon$ {*print* ('-');}

这两个翻译方案中的文法接受同样的语言，并且对任何输入，它们执行的语义动作是一样的。变换后的翻译方案中，动作都在产生式的右边，所以它们可以在自下而上分析过程中归约产生式右部的时候完成。

4.4.2 分析栈上的继承属性

对产生式 $A \to XY$，自下而上分析器从它的栈顶移开 Y 和 X，并用 A 代替它们，完成产生式右部的归约。如果 X 有综合属性 $X.s$，4.2.3 节的实现把该属性放入栈中与 X 对应的地方。

因为在 Y 以下的任何子树归约前，$X.s$ 的值已经在栈中，所以它的值可以被 Y 继承。如果 Y 的继承属性由复写规则 $Y.i = X.s$ 定义，那么在需要使用 $Y.i$ 的地方，可以取 $X.s$ 的值来代替。如果能静态地确定 $X.s$ 在栈中的位置，那么这个想法很容易实现。

例 4.12 考虑由表 4.6 的语法制导定义改写的翻译方案：

$D \to$ T {$L.in = T.type$;}

L

$T \to$ **int** {$T.type = integer$;}

$T \to$ **real** {$T.type = real$;}

$L \to$ {$L_1.in = L.in$;}

L_1, **id**　　{ *addType* (**id**. *entry*, *L. in*); }

L→ **id**　　{ *addType* (**id**. *entry*, *L. in*); }

该翻译方案使用继承属性,并且继承属性由复写规则传递。下面以输入 int p,q,r 为例,考察在需要 *L. in* 值的地方,能否静态地确定 *T. type* 在栈中的位置。

先用图4.13给出该输入的分析树及 *type* 和 *in* 属性之间的依赖关系,以增强直观性。表4.8给出分析器面临该输入时的动作序列。为清楚起见,仍然用对应的文法符号代替状态,用实际的标识符代替记号 **id**。在表4.8中,每次归约 *L* 产生式右部时,*T* 在栈中刚好处于该右部的下面,因此 *T. type* 在栈中的位置可以静态地确定。这个事实可用来访问属性 *T. type*。

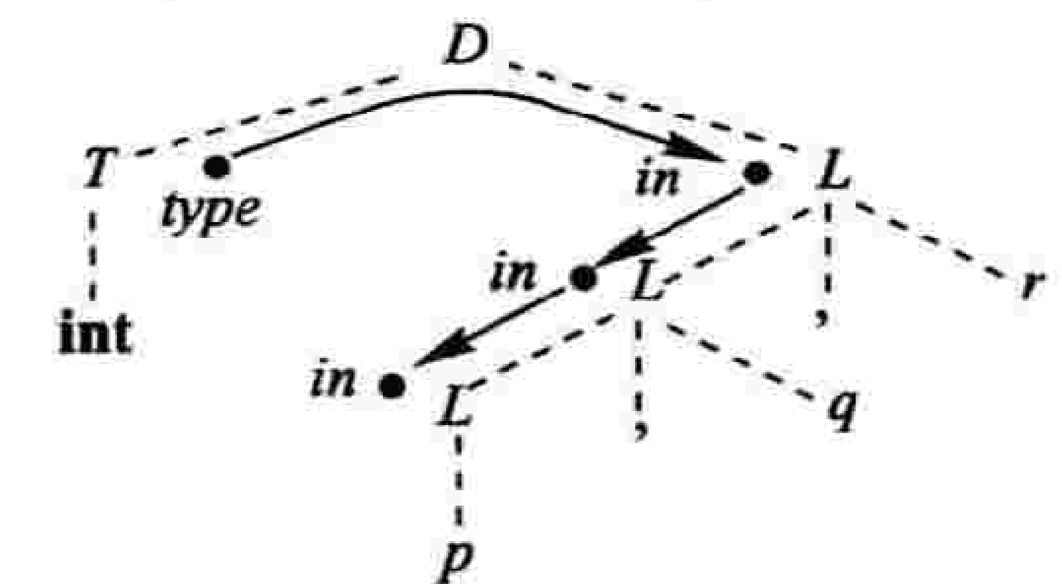


图4.13　在每个 *L* 结点上,*L. in* = *T. type*

表4.8　分析器面临 int p,q,r 时的动作序列

状态	输入	所用产生式
-	**int** p,q,r	
int	p,q,r	
T	p,q,r	*T* →**int**
*T*p	,q,r	
TL	,q,r	*L* →**id**
TL,	q,r	
TL,q	,r	
TL	,r	*L* →*L*,**id**
TL,	r	
TL,r		
TL		*L* →*L*,**id**
D		*D*→*TL*

按照4.2.3节的假设,分析栈由 *state* 和 *val* 两部分来实现,并且如果 *stack*[*i*]. *state* 对应文法符号 *X*,那么 *stack*[*i*]. *val* 保存 *X* 的综合属性 *s*。表4.9的实现利用了属性 *T. type* 在栈中的位置已知这个事实。

表 4.9 **$T.type$ 的值可用来代替 $L.in$**

产生式	代码段
$D \to TL$	
$T \to$ **int**	*stack*[*top*]. *val* = *integer*;
$T \to$ **real**	*stack*[*top*]. *val* = *real*;
$L \to L_1$, **id**	*addType*(*stack*[*top*]. *val*, *stack*[*top*−3]. *val*);
$L \to$ **id**	*addType*(*stack*[*top*]. *val*, *stack*[*top*−1]. *val*);

当使用产生式 $L \to$ **id** 时，**id**. *entry* 在 *val* 栈顶，$T.type$ 刚好在它的下面。所以 *addType*(*stack*[*top*]. *val*, *stack*[*top*−1]. *val*)等价于 *addType*（**id**. *entry*, $L.in$）。同样，因为产生式 $L \to L$, **id** 右部有三个符号，所以归约时 $T.type$ 出现在 *stack*[*top*−3]. *val*。其他和 $L.in$ 有关的动作是复写规则，它们完全可以省略掉。 □

如果属性的位置不能预测，可以尝试增加标记非终结符，使属性的位置变得可以静态确定。下面通过一个例子来说明。

例 4.13 考虑下面的语法制导定义：

产生式	语义规则
$S \to aAC$	$C.i = A.s$
$S \to bABC$	$C.i = A.s$
$C \to c$	$C.s = g(C.i)$

(4.2)

通过复写规则，$C.i$ 继承综合属性 $A.s$。由于在栈中，B 可能在、也可能不在 A 和 C 之间，因此在用产生式 $C \to c$ 归约时，$C.i$ 的值（也就是 $A.s$ 的值）在 *stack*[*top*−2]. *val* 或者在 *stack*[*top*−1]. *val*。

增加一个标记非终结符 M，插在(4.2)的第二个产生式右部的 C 之前，并增加了属性复写规则。修改后的语法制导定义如下：

产生式	语义规则
$S \to aAC$	$C.i = A.s$
$S \to bABMC$	$M.i = A.s$ $C.i = M.s$
$C \to c$	$C.s = g(C.i)$
$M \to \varepsilon$	$M.s = M.i$

产生式 $S \to bABMC$ 和 $M \to \varepsilon$ 的语义规则保证 $C.i = M.s = M.i = A.s$，也就是对于第二个产生式 $S \to bABMC$，$C.i$ 间接地通过 $M.i$ 和 $M.s$ 继承 $A.s$ 的值。这样，不管在 aAC 还是 $bABMC$ 的情况下，当使用 $C \to c$ 时，$C.i$ 的值都可从 *stack*[*top*−1]. *val* 找到。在使用 $M \to \varepsilon$ 时，$M.i$ 的值可以在 *stack*[*top*−1]. *val* 找到。 □

4.4.3 模拟继承属性的计算

如果继承属性并不直接等于某个综合属性,而是它的一个函数,是否能够利用上一小节的办法呢?回答是可以的。在这种情况下,可以用标记非终结符来模拟继承属性的计算。例如,考虑

产生式	语义规则
$S \to aAC$	$C.i=f(A.s)$
$C \to c$	$C.s=g(C.i)$

这里,定义 $C.i$ 的规则不是简单的复写规则。如果不完成这个计算,则在栈中取不到 $C.i$ 的值。解决办法是增加一个标记非终结符,把 $f(A.s)$ 的计算移到对标记非终结符归约时进行,如下所示:

产生式	语义规则
$S \to aANC$	$N.i=A.s \quad C.i=N.s$
$N \to \varepsilon$	$N.s=f(N.i)$
$C \to c$	$C.s=g(C.i)$

当按 $N \to \varepsilon$ 归约时,$N.i$ 的值可以在放 $A.s$ 的地方($stack[top].val$)找到。当按 $C \to c$ 归约时,$C.i$ 的值可以在对应于 N 的地方($stack[top-1].val$)找到。

例 4.14 对于表 4.7 的排版语言例子,可以增加三个标记非终结符 L,M 和 N,以保证当 B 的子树被归约时,所需的继承属性 $B.ps$ 的值出现在分析栈中已知的位置。修改后的语法制导定义见表 4.10。

标记非终结符 L 的作用是完成 $S \to LB$ 中的属性 $B.ps$ 的初始化。$L \to \varepsilon$ 的语义规则 $L.s=10$ 使得 $B.ps$ 的初值 10 在栈中有存放的地方,以便后面引用。

标记非终结符 M 和 N(还有 L)的一个共同作用是,保证了在任何情况下向 B 归约时,$B.ps$ 的值总是正好在右部的下面。这个结论的证明留作一个练习。

N 的另一个作用是模拟继承属性的计算。在表 4.7 中,产生式 $B \to B_1$ **sub** B_2 的语义规则 $B_2.ps=shrink(B.ps)$ 表明 $B_2.ps$ 不是简单地由复写规则定义的。产生式 $N \to \varepsilon$ 的动作完成这个计算,并将计算结果保留在栈上。

表 4.10 由复写规则设置所有继承属性

产生式	语义规则
$S \to LB$	$B.ps=L.s \qquad S.ht=B.ht$
$L \to \varepsilon$	$L.s=10$
$B \to B_1MB_2$	$B_1.ps=B.ps \quad M.i=B.ps \qquad B_2.ps=M.s \quad B.ht=max(B_1.ht, B_2.ht)$
$M \to \varepsilon$	$M.s=M.i$

续表

产生式	语义规则
$B\rightarrow B_1$ **sub** NB_2	$B_1.ps=B.ps$ $N.i=B.ps$ $B_2.ps=N.s$ $B.ht=disp\ (B_1.ht, B_2.ht)$
$N\rightarrow \varepsilon$	$N.s\ =shrink(N.i)$
$B\rightarrow$**text**	$B.ht=$**text**$.h \times B.ps$

实现表 4.10 语法制导定义的代码段列在表 4.11。在表 4.10 中,所有的继承属性都是由复写规则来赋值,因而只要知道它们在栈中的位置,就可以获得它们的值。 □

表 4.11 表 4.10 的语法制导定义的实现

产生式	语义规则
$S\rightarrow LB$	$stack[top-1].val=stack[top].val;$
$L\rightarrow \varepsilon$	$stack[top+1].val\ =10;$
$B\rightarrow B_1MB_2$	$stack[top-2].val=\ max(stack[top-2].val, stack[top].val);$
$M\rightarrow \varepsilon$	$stack[top+1].val\ =stack[top-1].val;$
$B\rightarrow B_1$ **sub** NB_2	$stack[top-3].val=disp\ (stack[top-3].val, stack[top].val);$
$N\rightarrow \varepsilon$	$stack[top+1].val\ =shrink(stack[top-2].val);$
$B\rightarrow$**text**	$stack[top].val=stack[top].val \times stack[top-1].val;$

可以证明,对于基于 LL(1)文法的 L 属性定义,系统地引入标记非终结符后,可以在 LR 分析期间完成属性计算。因为每个标记非终结符至多只有一个产生式,因此标记非终结符加入后文法仍然是 LL(1)的。任何 LL(1)文法一定是 LR(1)文法,因此加标记非终结符后不会引起 LR 分析动作的冲突。

但是对于基于 LR(1)文法的 L 属性定义,情况就不这么简单,标记非终结符的加入有可能使得文法不再是 LR(1)的了。

例 4.15 文法 $L\rightarrow Lb \mid a$ 是 LR(1)的,加了标记非终结符 M 后的文法

$L\rightarrow MLb \mid a$

$M\rightarrow \varepsilon$

不是 LR(1)的。

该语言的句子是 $abb\cdots b$ 的形式。根据这个文法,空归约的个数和 b 的个数一样多。在面临 a 时,后面有多少个 b 是不知道的,因此一定有移进-归约冲突。所以这个加标记非终结符的文法不是 LR(1)的。 □

除 4.1 节所讨论的分析树方法以外,本章具体讨论的都是一边分析一边进行属性计算的方

法。它们的优点是效率较高,缺点是访问结点的次序受到分析方法的限制,因而只能完成 L 属性计算。

对于非 L 属性定义,可以把分析和属性计算分开,先建立分析树(也可以用语法树),然后遍历分析树来计算属性。可以把4.3节的预测分析技术进行推广,仍然为每个非终结符构造一个函数,但是这些函数不含语法分析部分,它们是通过遍历分析树来完成属性计算的递归函数。每个函数对分析树上相应结点的子结点的访问次序由所用产生式的语义规则决定,但不一定是从左到右的。

这种方法不受分析方法的限制,属性计算的次序可在构造编译器时通过分析语法制导定义来得出,而且它可以推广到对分析树需要进行多遍扫描的翻译。

习 题 4

4.1 根据表4.1的语法制导定义,为输入表达式5*(4*3+2)构造注释分析树。

4.2 构造表达式((a*b)+(c))的分析树和语法树:

(a) 根据表4.3的语法制导定义。

(b) 根据图4.9的翻译方案。

4.3 为文法

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

(a) 写一个语法制导定义,它输出括号的对数。

(b) 写一个语法制导定义,它输出括号嵌套的最大深度。

4.4 下面的文法定义语言 $L=\{a^n b^n c^m \mid m, n \geq 1\}$。写一个语法制导定义,其语义规则的作用是:对不属于语言 L 的子集 $L_1=\{a^n b^n c^n \mid n \geq 1\}$ 的句子,打印出错信息。

$S \rightarrow DC$

$D \rightarrow aDb \mid ab$

$C \rightarrow Cc \mid c$

4.5 为下面文法写一个语法制导的定义,它完成一个句子的 **while-do** 最大嵌套层次的计算并输出这个计算结果。

$S \rightarrow E$

$E \rightarrow$ **while** E **do** $E \mid$ **id** $:= E \mid E + E \mid$ **id** $\mid (E)$

4.6 下列文法产生由+作用于整常数或实常数的表达式。两个整数相加时,结果是整型,否则是实型。

$E \rightarrow E + T \mid T$

$T \rightarrow$ **num.num** $\mid$ **num**

(a) 给出决定每个子表达式类型的语法制导定义。

(b) 扩充(a)的语法制导定义,既决定类型,又把表达式翻译成后缀表示。使用一元算符 **inttoreal** 把整数变成等价的实数,使得后缀式中+的两个对象有同样的类型。

4.7 给出对表达式求导数的语法制导定义,表达式由+和 * 作用于变量 x 和常数组成,如 $x*(3*x+x*x)$,并假定没有任何化简,例如将 $3*x$ 翻译成 $3*1+0*x$。

*4.8 给出把中缀表达式翻译成没有冗余括号的中缀表达式的语法制导定义。例如,因为+和 * 是左结合,$((a*(b+c))*(d))$ 可以重写成 $a*(b+c)*d$。

4.9 用 S 的综合属性 *val* 给出下面文法中 S 产生的二进制数的值。例如,输入 101.101 时,$S.val=5.625$。

$S \to L.L \mid L$

$L \to LB \mid B$

$B \to 0 \mid 1$

(a) 仅用综合属性决定 $S.val$。

(b) 用 L 属性定义决定 $S.val$。在该定义中,B 的唯一综合属性是 c(还需要继承属性),它给出由 B 产生的位对最终值的贡献。例如,101.101 的最前一位和最后一位对值 5.625 的贡献分别是 4 和 0.125。

4.10 重写例 4.3 语法制导定义的基础文法,然后重新设计语法制导定义,使得仅用综合属性就能把类型信息填入符号表。

4.11 由下列文法产生的表达式包括赋值表达式。

$S \to E$

$E \to E=E \mid E+E \mid (E) \mid$ **id**

表达式的语义和 C 语言的一样,即 $b=c$ 是把 c 的值赋给 b 的赋值表达式,而且 $a=(b=c)$ 把 c 的值赋给 b,然后再赋给 a。构造一个语法制导定义,它检查赋值表达式的左部是否为左值。

4.12 文法如下:

$S \to (L) \mid a$

$L \to L,S \mid S$

(a) 写一个翻译方案,它输出每个 a 的嵌套深度。例如,对于句子$(a,(a,a))$,输出的结果是1 2 2。

(b) 写一个翻译方案,它打印出每个 a 在句子中是第几个字符。例如,当句子是$(a,(a,(a,a),(a)))$时,打印的结果是2 5 8 10 14。

4.13 语句的文法如下:

$S \to$ **id** := E | **if** E **then** S | **while** E **do** S | **begin** S;S **end** | **break**

写一个翻译方案,其语义动作的作用是:若发现 **break** 不是出现在循环语句中,及时报告错误。

4.14 程序的文法如下:

$P \to D$

$D \to D;D \mid$ **id** : T | **proc id**;D;S

(a) 写一个语法制导定义,打印该程序一共声明了多少个 **id**。

(b) 写一个翻译方案,打印该程序每个变量 **id** 的嵌套深度。

4.15 下面是构造语法树的一个 S 属性定义。将这里的语义规则翻译成 LR 翻译器的栈操作代码段。

$E \to E_1 + T$	$E.nptr = mkNode$ ('+', $E_1.nptr$, $T.nptr$)
$E \to E_1 - T$	$E.nptr = mkNode$ ('-', $E_1.nptr$, $T.nptr$)
$E \to T$	$E.nptr = T.nptr$
$T \to (E)$	$T.nptr = E.nptr$
$T \to$ **id**	$T.nptr = mkLeaf$ (**id**, **id**.$entry$)
$T \to$ **num**	$T.nptr = mkLeaf$ (**num**, **num**.val)

4.16 参照4.3.3节,给出例4.10中对应非终结符 T 的翻译函数。

*4.17 假定有 L 属性定义,它的基础文法是LL(1)的,或者是一个能解决其二义性并且为之构造预测分析器的文法,证明可以把继承属性和综合属性放在由预测分析表驱动的自上而下分析器的分析栈中。

*4.18 证明在LL(1)文法的任何地方加上可区别的标记非终结符后,结果文法仍然是LR(1)的。

第5章

类型检查

编译器必须检查源程序是否满足源语言在语法和语义两个方面的约定。这种检查称为**静态检查**(以区别在目标程序运行时的**动态检查**),它诊断和报告程序错误。静态检查的例子如下。

(1) 类型检查。如果算符作用于不相容的运算对象,编译器报告错误。例如数组变量和函数变量相加。

(2) 控制流检查。对于任何引起控制流离开一种程序构造的语句,程序中必须有该控制转移可以转到的地方。例如,C 的 break 语句引起控制离开最小包围的 while、for 或 switch 语句,如果这样的包围语句不存在,则是一个错误。

(3) 唯一性检查。有些场合,对象必须正好被定义一次。例如,在 C 的函数中,局部变量必须唯一地声明,switch 语句的分情形常量必须有区别,枚举类型的元素不能重复。

(4) 关联名字检查。有时,同样的名字必须出现两次或更多次。例如,在 Ada 语言中,循环或程序块可以有名字出现在它的开头和结尾。编译器必须检查两个地方是否使用同样的名字。

本章集中于类型检查。上面的列举表明,大多数其他静态检查都是一些简单的工作,可以用上章所讲的技术实现。其中有些工作可以并入编译器的其他部分,例如,在把名字的信息填入符号表时,可以检查该名字声明的唯一性。许多编译器把静态检查和中间代码生成组织在分析的同时完成。对于更复杂的构造,如 Ada 中的一些构造,在分析和中间代码生成之间加一遍类型检查可能会方便一些,见图 5.1。

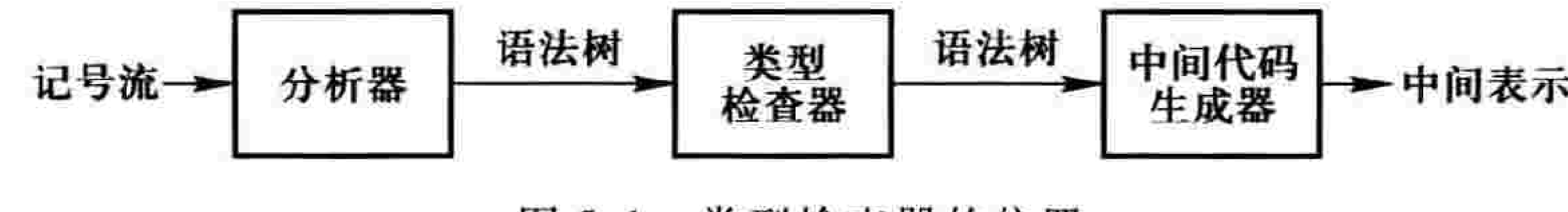


图 5.1　类型检查器的位置

5.1　类型在编程语言中的作用

本节非形式地给出执行错误、安全语言、类型系统和其他有关概念,讨论所期望的类型系统

的性质和好处，并且概述类型系统的形式化。本节所用的术语并非绝对标准，因为来源不同的术语原来就不一致。

5.1.1 执行错误和安全语言

本节先介绍一些和程序运行有关的概念。

程序运行时出现的错误称为**执行错误**(execution error)。有些执行错误，如非法指令错误、非法内存访问错误和除数为零错误，机器都会报告错误，因为在许多计算机体系结构上，这样的错误使得计算立即停止，并且操作系统报告发现错误的位置和错误性质。然而，还有一些难以捉摸的执行错误，它们引起数据遭到破坏，但操作系统发现不了因而也不会立即报告错误。一个例子是不适当地访问一个合法的地址，例如，在没有运行时越界检查的情况下访问超越数组边界的数据。另一个例子是跳转到一个错误的地址，该地址开始的内存正好代表一个指令序列，使得该错误在一段时间内都不会引起可被捕捉的事件。因此执行错误可以分成两类：一类引起计算立即停止；另一类当时未引起会被捕捉的事件，然后可能引发难以预料的行为。前者称为**会被捕获的错误**(trapped error)，后者称为**不会被捕获的错误**(untrapped error)。区别这两类错误是有用的。

如果一个程序的运行不可能引起不会被捕获错误的出现，那么就称该程序是**良行为的**(well behaved)。所有合法程序都是良行为的语言称为**安全语言**(safe language)。保证语言安全性的通常做法是为语言设计一个类型系统，通过静态(编译时)检查、动态(运行时)检查或静态检查与运行检查混合的方式来拒绝一切有潜在不安全性的程序。

虽然安全性是程序一个至关重要的性质，但是对一种类型化语言(typed language)来说，很少关心类型系统是否恰好排除不会被捕获的错误，因为设计一个类型系统正好排除这类错误是非常困难的。因此，安全语言类型系统的设计结果常常是在排除所有不会被捕获错误的同时，也排除部分会被捕获的错误。

这样，对任何一种语言，可以指定所有可能执行错误集合的一个子集作为**禁止错误**(forbidden error)。若是安全语言，则禁止错误应该包括所有不会被捕获的错误，再加上部分会被捕获的错误。良行为的程序和安全语言也可以基于禁止错误来定义。

5.1.2 类型化语言和类型系统

一个程序变量在程序执行期间可取的值有一定的限制，这个限制的依据就是该变量的类型。例如，boolean 类型的变量 x 在程序运行时只能取布尔值。若 x 是 boolean 类型变量，那么表达式 not(x)在程序每次运行时都有意义，因为 not 函数接受 boolean 类型的参数，返回布尔类型的值。若语言的规范为其每种运算都定义了各运算对象和运算结果所允许的类型，则该语言称为**类型化语言**。

一种类型化语言由一个实质上存在的类型系统来类型化，而不在乎类型是否实际出现在程

序的语法中。在一种语言中,若函数和变量的类型必须显式声明,则该语言被称为**显式类型化语言**。Java 和 C++都是显式类型化语言。类型声明不是必不可少的语言被称为**隐式类型化语言**。例如,函数式语言 ML 和 Haskell 支持编写忽略部分类型信息的程序段,这些语言的类型系统会自动地给这些程序段指派类型,5.4.5 节有这样的例子。

在类型化语言中,**类型系统**(type system)由一组**定型规则**(typing rule)构成,这组规则用来给构成一个程序的各种语言构造(例如变量、表达式、函数和模块等)指派类型。非形式描述的定型规则的例子有:若 M 和 N 都是整型表达式,则 $M+N$ 也是整型表达式。

编译器基于语言的类型系统对程序进行**类型检查**(type checking),类型检查就是根据定型规则来确定程序中各语法构造的类型,类型检查的目的是拒绝那些有**类型错误**的程序,例如含表达式 true+3 的程序。不能通过类型检查的程序不是合法的程序,严格地说,它们不能称为程序。完成这种类型检查的算法称为类型检查器。能够通过类型检查的程序称为**良类型的**(well typed)程序。在一个语言定义中,除了文法外,若上下文有关的所有限制都体现在类型系统中,则良类型的程序就是合法程序。从本章引言所举的一些上下文有关约束知道,控制流和唯一性等上下文有关约束通常都不是表达在类型系统中的。与类型检查相近的类型推断概念在 5.2.3 节介绍。

5.1.1 小节已经提到,为语言设计类型系统的根本目的是防止程序运行时出现禁止错误,以保证其运行是良行为的。如果良类型程序一定是良行为的,则称该语言是**类型可靠的**(type sound)。类型可靠的语言一定是安全语言。执行错误、良行为程序和安全语言是基于程序运行时特征来定义的,它们是动态的概念;而类型错误、良类型程序和类型化语言是静态的概念。类型可靠是联系两者的一个重要概念,类型检查的目的就是保证程序运行时的良行为。

和类型化语言有关的常用概念还有强类型语言和弱类型语言,它们的含义在不同的文献中差别较大,有的把它们当成静态的概念,另外一些把它们看成像类型可靠这样联系静态和动态的概念。本书避免使用这两个概念。

对于类型可靠的语言来说,良类型的程序有下面几点性质:

- 不会出现不会被捕获的错误(即是安全的);
- 不会出现已列入禁止错误的会被捕获的错误;
- 有可能出现其他会被捕获的错误,避免这些错误是程序员的责任。

类型检查通常是在编译时静态地完成,以减少运行时开销。静态类型检查的语言有 ML 和 Java 等。但是,静态类型检查的语言通常难以避免一些运行时的测试,因为有些表达式的类型很难静态确定的。例如表达式 a[e]的类型除了取决于变量 a 是否为数组类型外,还要看 e 的值是否在上下界范围内,而这种越界检查通常是动态测试的。

语言若不限制变量的取值范围,则称之为**无类型语言**(untyped language),它们没有类型,或者说仅有一个包含所有值的泛类型。在这样的语言中,一个运算可以作用于任意的运算对象,其结果可能是一个有意义的值、一个错误、一个异常或一个语言未加定义的结果。汇编语言是典型的无类型语言,并且它是令人讨厌的不安全语言。

大多数无类型高级语言也是安全的,它们可以通过彻底的运行时详细检查来排除所有的禁止错误。例如,它们检查所有下标变量的访问是否越界,检查所有的除法操作,当禁止错误出现时,产生异常。这些语言的检查过程称为动态检查,LISP 语言是一个这样的例子。这些语言虽然既没有静态检查,也没有类型系统,但它们是安全的。

倘若没有编译时和/或运行时检查来避免数据遭破坏的话,程序设计将是令人沮丧的事情。

现在实际使用的众多语言中,有些语言即使是类型化的语言,也不能保证安全性,因为它们的禁止错误集合并没有包含所有不会被捕获的错误。部分非安全运算会被静态地检查出来,但还有一些难以检查。C 语言就有很多不安全的并且被广泛使用的特征,如指针算术运算和类型强制。很有趣的是,人们为 C 程序员总结的戒律中约一半是直接针对 C 的不安全性的。C 的这些问题一部分已经在 C++中得以缓解,更多一些问题在 Java 中已得到解决。Modula-3 支持一些不安全特征,但是仅限于显式标记为不安全的模块中,以防止安全模块输入不安全的接口。ML 是一个类型化的安全语言。

在语言设计的历史上,对安全性考虑不足是出于效率上的原因,最典型的是 C 语言。C 以处理低级构造的灵活性著称,它的设计为了保证编程的灵活性而牺牲了安全性,它鼓励在安全的边缘进行编程。这使得 C 语言程序的效率很高,但是也使得它们缺乏安全性。为了排除程序中的安全漏洞,需要进行大量的调试工作。

对于安全性不足的语言,可以通过运行时的检查来提高安全性,但是运行时检查的代价有时是非常昂贵的。另一方面,有些安全语言,即使对程序做了细致的静态分析,要想获得彻底的安全性也需要一点运行时的代价。例如,上面已经提到,数组越界检查是不可能完全在编译时删除的。

为获得安全性所花的代价是值得的。安全性使得程序出现执行错误就会停止执行,这可以减少调试的时间。而且安全性保证运行时存储空间结构的完整性,因而使得无用存储单元收集(garbage collection,俗称垃圾收集,见第 11 章)可以完成。而无用存储单元收集大大降低代码开发的时间和代码的规模,虽然它需要一点运行时的代价。

从历史上看,安全和不安全语言之间的选择可能最终与开发时间和执行时间之间的权衡相关。但是,随着国家、社会、经济和日常生活越来越依赖于信息技术,关键的基础设施如交通运输、通信、金融市场、能源分配和卫生保健等,都将非常危险地依赖于不是一个管理机构范围内的计算基础和资源。而当我们越来越依赖于这些基础构造的同时,提高相关信息系统对恶意攻击和有漏洞软件的破坏的抵抗能力显得越来越重要。软件系统的编程语言是否安全也成为其中一个重要的考虑因素。例如,由于 C 语言有一些不安全因素,对于用 C 编写的系统来说,利用缓冲区溢出对它进行的攻击已占目前各种攻击的 50%。另外还有根据 C 程序格式串的安全漏洞而进行的攻击等。因此,近年来以及今后,在安全和不安全语言之间的选择还与系统应达到的安全级别有关。

5.1.3 类型化语言的优点

编程语言是否应该有类型？该问题仍然是某些讨论的一个主题。从可维护性的观点看，很明显，不安全的类型化语言优于安全的无类型语言（例如，C 和 LISP 的比较）。几乎不用怀疑，用无类型语言开发的产品代码维护起来非常困难。从工程的观点看，类型化语言有下面一些优点。

（1）开发的实惠。有了类型系统可以较早发现错误，例如整数和串相加。若类型系统设计得很好，类型检查可以发现大部分常见的程序设计错误，剩下的错误也很容易调试，因为大部分错误已经被排除了。

对于大规模软件开发来说，接口和模块有方法学上的优点，类型信息在这里可以组织到程序模块的接口中。程序员可以一起讨论接口，然后分头编写各自要实现部分的代码。这些代码之间的相互依赖最小，并且代码可以局部地重新安排而不必担心对全局造成影响。

程序中的类型信息还具有文档作用。程序员声明标识符和表达式的类型，也就是告诉了所期望的值的部分信息，这对阅读程序是很有用的。

（2）编译的实惠。程序模块可以相互独立地编译，例如 Modula-2 和 Ada 的模块，每个模块仅依赖于其他模块的接口。这样，大系统的编译可以更有效，因为改变一个模块并不会引起其他模块的重新编译，至少在接口稳定的情况下是这样。

（3）运行的实惠。在编译时收集类型信息，保证了在编译时就能知道数据占用空间的大小，因而可得到更有效的空间安排和访问方式，提高了目标代码的运行效率。例如，像 C 的结构体和 C++的对象，其域或成员的偏移可以根据它们的类型信息静态地确定。

另外，一般来说，精确的类型信息在编译时可以保证运行时的运算都作用于适当的对象并且不需要昂贵的运行时的测试，从而提高程序运行的效率。例如在 ML 中，精确的类型信息可以删除在指针脱引用（dereference）中的 nil 检查。

上面提到，类型信息具有文档作用，但是它和其他形式的程序标注不同。一般来说，关于程序行为的标注可以包含从非形式的注解一直到精确表述的形式规范。类型信息处在该范围的中间：它们比程序注解精确，比形式规范容易理解。

5.2 类型系统的描述语言

类型系统是一种逻辑系统，它主要用来描述编程语言的定型规则，即类型系统中的推理规则。对类型系统的研究可以采用形式方法（formal method）或非形式的方法。非形式的描述通常不能把语言的类型结构详细描述到不会出现歧义实现的程度。历史上曾经出现过同一个语言的不同编译器实现了略有区别的类型系统，5.5 节将给出这样的例子。类型系统的形式化就是先给出符号表示的定型断言，然后再用这样的定型断言来描述定型规则，包括定型公理。形式的类

型系统应该是类型化编程语言的定义的一部分。形式的类型系统提供了概念上的工具,用它们可以评判一个语言定义中的某些重要方面是否适当。依靠形式的类型系统,才有可能证明语言是否类型可靠。

类型系统独立于类型检查算法。这类似于形式文法可用来描述编程语言的语法,但这种描述独立于语法分析算法。把类型系统从类型检查算法中分离出来是有益的:类型系统属于语言定义,而类型检查算法属于编译器。用类型系统(而不是用编译器的算法)很容易解释语言在类型方面的规定。而且,不同的编译器对同一个类型系统可能使用不同的类型检查算法。

从技术上说,通常设计类型系统的一个重要目标是存在有效的类型检查算法。但也有这样的类型系统,它只存在难以实施的类型检查算法,或者完全没有类型检查算法,本章介绍的一些简单类型系统不涉及这种情况。

类型系统的基本概念实质上可用于所有的计算范型(paradigms),包括函数式语言、命令式语言和并行语言等。一些定型规则还可以不作修改地用于不同的范型,例如对函数来说,不管是函数式语言还是命令式语言,不管是值调用还是换名调用,其基本定型规则都一样。

形式的类型系统是编程语言手册中非形式类型系统的一种数学描述。鉴于它的精确性,本章尽量采用形式方法来描述类型系统。

5.2.1 定型断言

和逻辑系统一样,类型系统的形式描述从描述**断言**(assertion)开始。一个典型的断言具有形式

$\Gamma \vdash S$　　　　S 的所有自由变量都声明在 Γ 中

这里的 Γ 是一个**静态定型环境**(static typing environment),其形式是 n 个不同名字的变量 $x_1, \cdots, x_n$ 和它们所属类型 $T_1, \cdots, T_n$ 的有序表 $x_1 : T_1, \cdots, x_n : T_n$。这相当于程序的类型声明语句和编译器的符号表。在 Γ 中声明的这组变量 $x_1, \cdots, x_n$ 用 $dom(\Gamma)$ 表示。变量个数为零的空环境用 $\varnothing$ 表示。S 的形式随断言形式的不同而不同,但是 S 的所有自由变量必须在 Γ 中声明。

断言有三种形式:环境断言、语法断言和定型断言。例如,下面这个环境断言直接声称定型环境 Γ 是**良形的**(well formed),即它是适当地构造的:

$\Gamma \vdash \diamond$

后面将用环境推理规则来定义定型环境的语法。

语法断言用来表示类型表达式是良形性。语言构造的类型可以用"类型表达式"来表示,基本类型是类型表达式,把类型构造符应用到类型表达式而形成的表达式也是类型表达式。例如:

$\Gamma \vdash nat$

表示在静态定型环境 Γ 下,nat 是一个类型表达式。后面将用语法推理规则来定义类型系统中的类型表达式的语法。在编程语言的语法中,类型表达式是用文法描述的(见后面图 5.2)。但是在设计类型系统时,通常都用本节的方式来描述,其优点是能够用 Γ 来表示上下文有关的

约束。

最重要的断言是**定型断言**,其形式是:

$\Gamma \vdash M : T$

它声称在静态定型环境 Γ 下,M 具有类型 T(M 的自由变量都在出现 Γ 中),其中 M 是表达式和语句这样的程序构造。例如:

$\varnothing \vdash true : boolean$ 和 $x : nat \vdash x + 1 : nat$

第一个定型断言告知 *true* 是 *boolean* 类型的。因为 *true* 是常量,因此在空定型环境下也能判断它的类型。第二个定型断言告知,当 x 具有类型 *nat* 时,$x + 1$ 的类型是 *nat*。后面将用形式的定型规则来定义程序构造的类型。

一个断言可能是**有效的**(valid)或**无效的**(invalid)。例如,$\Gamma \vdash true : boolean$ 是一个有效断言,因为 *true* 在布尔值集合中;显然,$\Gamma \vdash true : nat$ 是无效断言,因为 *true* 不在自然数集合中。

5.2.2 定型规则

作为一个逻辑系统,类型系统也有它的推理规则。它们是在一组已知有效的断言的基础上,声称某个断言的有效性。它们的一般形式是

$$\frac{\Gamma_1 \vdash S_1, \cdots, \Gamma_n \vdash S_n}{\Gamma \vdash S}$$

在每条推理规则中,横线上面是一组称为**前提**的断言,横线下面是一个称为**结论**的断言。推理规则的含义是,当所有的前提都有效时,则结论也有效。前提个数可以为零,这样的规则叫做**公理**,即横线下的断言是有效的。

为便于引用,每条规则有一个规则名。其名称的第一个词由结论断言来确定。例如,形式为(Exp…)的规则名用于这样的规则:其结论是一个表达式的定型断言。需要时,可以规定规则使用的限制条件,还可以在规则中使用一些缩写,它们都可以列在规则名或规则的右边,称之为注释。因此,下面按照

(规则名)(注释)　　　　推理规则　　　　(注释)

的形式来介绍推理规则,其中注释可以为空。

针对三种不同的断言形式,有三类推理规则。例如环境规则:

(Env $\varnothing$)　　　　$\overline{\varnothing \vdash \diamond}$

表示空环境是良形的,它是一个公理。

下面的语法规则本质上是说,在任何良形环境 Γ 下,*boolean* 是一个类型表达式。

(Type Bool)　　　　$\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash boolean}$

推理规则的结论是定型断言的话,则称之为**定型规则**。例如,下面的第一条规则是说,在任何良形环境下,任何一个自然数都是类型为 *nat* 的表达式。第二条规则是说,两个 *nat* 类型的表

达式 M 和 N 相加所得表达式 $M+N$ 仍是一个 nat 类型表达式，而且 M 和 N 的环境 Γ 继续作为 $M+N$ 的环境。

(Val n) (n 可以是 0,1,…) $$\frac{\Gamma \vdash \diamond}{\Gamma \vdash n : nat}$$

(Exp +) $$\frac{\Gamma \vdash M : nat, \Gamma \vdash N : nat}{\Gamma \vdash M+N : nat}$$

一个语言的定型规则（还有它的环境规则和语法规则）的集合构成该语言的（**形式的**）**类型系统**。从技术上讲，类型系统符合**形式证明系统**的一般框架，一组定型规则用于执行一步步的演绎推理，类型系统中的这种演绎推理用于对程序或程序构造定型。

5.2.3 类型检查和类型推断

在一个可靠的类型系统中，通过演绎推理得到的断言一定是该类型系统的**有效断言**，也就是说，一个有效断言是通过正确地应用类型规则可以得到的断言。形式的演绎推理在其他课程中学习过，在此不作介绍，将在 5.4 节举例。

在一个类型系统中，如果存在一个类型 T 使得 $\Gamma \vdash E : T$ 是一个有效断言，那么 E 对环境 Γ 来说是良类型的，并且它的类型是 T。用语法制导的方式，根据上下文有关的定型规则来判定程序构造是否良类型的过程，通常被称作类型检查。例如，$\Gamma \vdash E : T$ 的任何推导必须依据 E 的语法结构，因此类型检查通常是语法制导的。

若程序构造的形式是函数 $f(x : t) = E$，则该函数首部规定了 x 的类型是 t。此时，对 E 进行类型检查是在已知 x 类型的情况下进行的，命令式语言通常都是这样。但是，如果允许程序员忽略函数首部的参数类型，把该函数写成 $f(x) = E$ 的形式，那么就不清楚 x 是什么类型了。此时需要对 E 进行更加复杂的分析，才能判断 x 应该是什么类型。这种在类型信息不完全情况下的定型判定称为**类型推断**(type inference)。区别类型推断和类型检查是有用的，但是精确区分它们是困难的。

如果不存在 E 的任何定型推导，就说 E 是不可定型的，通常说它有类型错误或者类型缺陷。

程序构造的类型检查或类型推断问题，与所使用的类型系统密切相关。算法可能非常容易，也可能非常困难，甚至不可能实现，这取决于所使用的类型系统。另外，如果能够找到这样的算法，它可能非常有效，也可能慢得令人失望。显然，类型系统的实际效用依赖于是否存在好的类型检查或类型推断算法。对于 C 和 Java 这样的显式类型化命令式语言，类型检查问题还算容易解决。对于像 ML 这样的隐式类型化函数式语言，类型推断问题相当困难，其基本算法已被很好地理解和广泛地使用，但是实际使用的算法相当复杂，一些问题仍然在研究中。

5.3 一个简单类型检查器的规范

本节描述一个简单语言的类型系统和语法制导的类型检查器。在此语言中,每个标识符的类型必须在该标识符使用前先声明。这个类型检查器是一个翻译方案,它根据子表达式的类型来确定表达式的类型。该类型检查器能够处理数组、指针、语句和函数。

5.3.1 一个简单的语言

图 5.2 是该语言的文法,其中 P 表示程序,它由声明 D 和随后的语句 S 组成。为集中于类型检查器的介绍,这里采用了一个比较简洁的二义文法,并且允许直接声明函数标识符的类型而无须给出具体的函数定义。

$P \to D ; S$
$D \to D ; D \mid$ **id** : T
$T \to$ **boolean** | **integer** | **array** [**num**] **of** $T \mid \uparrow T \mid T$ '→' T
$S \to$ **id** := $E \mid$ **if** E **then** $S \mid$ **while** E **do** $S \mid S ; S$
$E \to$ **truth** | **num** | **id** | E **mod** $E \mid E$ [E] $\mid E \uparrow \mid E$ (E)

图 5.2 源语言文法

图 5.2 第三行是描述类型的产生式。经从 T 开始的语法推导得到的终结符号串都是类型表达式,它们可以像算术表达式那样用树来描述。每个类型表达式都是该语言的一个类型,因此经从 T 开始的推导,可以得到该语言所有的类型。在 5.3.2 节的类型系统中,类型表达式用下面更接近抽象的语法:

$T \to$ *boolean* | *integer* | *array* (N, T) | *pointer*(T) | T '→' T | *void*

其中 N 表示自然数,*void* 在类型系统中作为语句的类型。在 5.3.3 节的类型检查器中,还需要增加 *type_error*,作为出现类型错误的程序构造的类型。

有关图 5.2 文法的其他一些解释,穿插在下面各小节。由该文法产生的程序实例有

```
i : integer;
j : integer;
j: =i mod 2000
```

在该程序段中,选择 mod 作为二元算符的代表。

5.3.2 类型系统

事实上,语言的设计和其类型系统的设计是同时进行并相互影响的。类型系统的设计包括

三部分。首先,最简单的是环境规则的设计。表达式类型的确定和环境有关,因为表达式中可能有变量。两条环境规则同图5.2文法中的类型声明有关,第一条规则是说空环境∅是良形的,第二条规则是说,程序中一个新的变量声明引起在环境中增加一个变量到类型的映射。

(Env ∅) $\dfrac{}{\varnothing \vdash \diamond}$

(Decl Var) $\dfrac{\Gamma \vdash T,\ \mathrm{id} \notin \mathrm{dom}\ (\Gamma)}{\Gamma,\ \mathrm{id}: T \vdash \diamond}$

类型系统设计的第二部分是设计在定型规则中使用的类型表达式的语法,即良形类型表达式的定义。5.3.2节已经提到,这里采用类型表达式的抽象语法,它和5.3.3节类型检查算法中所用类型表达式的语法一致。

下面三条语法规则表示,在任何良形环境下,*boolean*、*integer* 和 *void* 都是类型表达式。其中 *void* 类型本质上表示语句是没有类型的语法构造。

(Type Bool) $\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash boolean}$

(Type Int) $\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash integer}$

(Type Void) $\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash void}$

下面三条语法规则说明构造类型的语法。*pointer*、*array* 和→都是类型构造符。

规则(Type Ref)说明,如果 T 是类型表达式,那么类型表达式 $pointer(T)$ 表示指向类型为 T 的对象的指针类型。在图5.2的具体语法中,类型声明中的前缀算符↑用于指针类型,表达式中的后缀算符↑用于脱引用。

规则(Type Array)说明,如果 T 是类型表达式,那么 $array(N,T)$ 表示成分类型为 T 的数组类型。所有数组的下标都假定从0开始,数组的下标集合是 $0..N-1$。

规则(Type Fun)说明,类型表达式 $T_1 \to T_2$ 表示定义域类型为 T_1 和值域类型为 T_2 的函数类型。为简单起见,图5.2的语言只考虑了一元函数。

(Type Ref) $(T \neq void)$ $\dfrac{\Gamma \vdash T}{\Gamma \vdash pointer(T)}$

(Type Array) $(T \neq void)$ $\dfrac{\Gamma \vdash T, \Gamma \vdash N: integer}{\Gamma \vdash array(N,T)}$ $(N>0)$

(Type Fun) $(T_1, T_2 \neq void)$ $\dfrac{\Gamma \vdash T_1, \Gamma \vdash T_2}{\Gamma \vdash T_1 \to T_2}$

类型系统设计的第三部分是设计定型规则,也就是给各种形式的表达式和语句指派类型。这是类型系统最重要的部分,因此下面详细给出针对各种形式的表达式和语句的定型规则,首先针对表达式的7种语法形式,有7条定型规则用来确定表达式的类型。

(Exp Truth) $\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \mathrm{truth} : boolean}$

(Exp Num) $\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \text{num} : integer}$

(Exp Id) $\dfrac{\Gamma_1, \text{id} : T, \Gamma_2 \vdash \diamond}{\Gamma_1, \text{id} : T, \Gamma_2 \vdash \text{id} : T}$

(Exp Mod) $\dfrac{\Gamma \vdash E_1 : integer, \Gamma \vdash E_2 : integer}{\Gamma \vdash E_1 \text{ mod } E_2 : integer}$

(Exp Index) $\dfrac{\Gamma \vdash E_1 : array(N, T), \Gamma \vdash E_2 : integer}{\Gamma \vdash E_1[E_2] : T}$ $(0 \leqslant E_2.val \leqslant N-1)$

(Exp Deref) $\dfrac{\Gamma \vdash E : pointer(T)}{\Gamma \vdash E\uparrow : T}$

(Exp FunCall) $\dfrac{\Gamma \vdash E_1 : T_1 \rightarrow T_2, \Gamma \vdash E_2 : T_1}{\Gamma \vdash E_1(E_2) : T_2}$

上述规则(Exp Id)表示,如果 **id** 出现在环境 Γ 中,则 **id** 的类型就是它在 Γ 中的类型。

通常认为语句是没有类型的。在类型系统的设计中,可以给它们以特殊的基本类型 *void* 并使用下面这些定型规则,这样便于判断整个程序可执行的语句部分是否有类型错误。为简单起见,这里没有设计把声明部分和语句部分结合起来(即整个程序)的定型规则。

(State Assign) ($T = boolean$ or $T = integer$) $\dfrac{\Gamma \vdash \text{id} : T, \Gamma \vdash E : T}{\Gamma \vdash \text{id} := E : void}$

(State If) $\dfrac{\Gamma \vdash E : boolean, \Gamma \vdash S : void}{\Gamma \vdash \text{if } E \text{ then } S : void}$

(State While) $\dfrac{\Gamma \vdash E : boolean, \Gamma \vdash S : void}{\Gamma \vdash \text{while } E \text{ do } S : void}$

(State Seq) $\dfrac{\Gamma \vdash S_1 : void, \Gamma \vdash S_2 : void}{\Gamma \vdash S_1; S_2 : void}$

5.3.3 类型检查

根据上一小节的类型系统来设计类型检查器是直截了当的。在下面的翻译方案中,文法符号的 *type* 属性给出它的类型,编译器的符号表对应类型系统中的环境 Γ。*addtype* 函数将 **id** 的类型添加到符号表中,若出现重复定义则报错;*lookup* 函数取符号表中 **id** 的类型,若找不到 **id** 的类型则报错,返回 *type_error*。这里的类型表达式和类型系统中的一致,当然也可以另选一种表示形式。

类型错误不像语法错误那样直观,也不像语法错误那样容易被及时发现,但是对类型检查器同样要求它除了报告错误的性质和位置外,至少还能从错误中恢复过来,以便检查剩余输入中的各种错误。因此这里的类型检查器增加一个特别的类型 *type_error*,作为出现类型错误的语法构造的类型,但是没有考虑如何报告类型错误。类型检查器和类型系统的一个重要区别是,类型系

统的设计只考虑对无类型错误的语法构造定型,而类型检查器的设计还必须对付出现类型错误的情况。

先来看声明语句的语义动作。这些语义动作构造类型表达式,并将它们保存在 T 的综合属性 *type* 中。

$D \rightarrow D ; D$	
$D \rightarrow$ **id** : T	{ *addtype* (**id**. *entry*, T. *type*); }
$T \rightarrow$ **boolean**	{ T. *type* = *boolean*; }
$T \rightarrow$ **integer**	{ T. *type* = *integer*; }
$T \rightarrow \uparrow T_1$	{ T. *type* = *pointer*(T_1. *type*); }
$T \rightarrow$ **array** [**num**] **of** T_1	{ T. *type* = *array*(**num**. *val*, T_1. *type*); }
$T \rightarrow T_1$ '→' T_2	{ T. *type* = T_1. *type* → T_2. *type*; }

在表达式的语义动作中,E 的综合属性 *type* 给出了 E 产生的表达式的类型表达式。如果碰到类型错误时,该表达式的类型表达式是 *type_error*。如果表达式 E 的某个子表达式的类型是 *type_error*,那么 *type_error* 从这个子构造一直传到 E。

$E \rightarrow$ **truth**	{ E. *type* = *boolean*; }
$E \rightarrow$ **num**	{ E. *type* = *integer*; }
$E \rightarrow$ **id**	{ E. *type* = *lookup*(**id**. *entry*); }
$E \rightarrow E_1$ **mod** E_2	{ **if** (E_1. *type* == *integer* && E_2. *type* == *integer*) E. *type* = *integer*; **else** E. *type* = *type_error*; }
$E \rightarrow E_1 [E_2]$	{ **if** (E_2. *type* == *integer* && E_1. *type* == *array*(*s*, *t*)) E. *type* = *t*; **else** E. *type* = *type_error*; }
$E \rightarrow E_1 \uparrow$	{ **if** (E_1. *type* == *pointer*(*t*)) E. *type* = *t*; **else** E. *type* = *type_error*; }
$E \rightarrow E_1 (E_2)$	{ **if** (E_2. *type* == *s* && E_1. *type* == *s* → *t*) E. *type* = *t*; **else** E. *type* = *type_error*; }

对于数组引用 $E_1[E_2]$,下标表达式必须有 *integer* 类型。此时,结果类型是从 E_1 的类型 *array*(*s*, *t*) 中得到的元素类型 t。这里没有用数组的下标集合 s,因为数组引用的越界检查在运行时完成。

对于函数调用 $E_1(E_2)$,E_1 的类型必须是从 E_2 的类型 s 到某个值域类型 t 的函数类型 $s \rightarrow t$,$E_1(E_2)$ 的类型是 t。这里只考虑一元函数,后面介绍了积类型后,把类型检查推广到多元函数不是件难事。

从下面各种语句的语义规则可以看出,在没有类型错误时,整个程序可执行的语句部分的类型是 *void*,否则是 *type_error*。

$S \to$ **id** := E { **if** (**id**. *type* == *E*. *type* && *E*. *type* ∈ { *boolean*, *integer* })
   S. *type* = *void*;
  else *S*. *type* = *type_error*; }

$S \to$ **if** E **then** S_1 { **if** (*E*. *type* == *boolean*) *S*. *type* = S_1. *type*;
  else *S*. *type* = *type_error*; }

$S \to$ **while** E **do** S_1 { **if** (*E*. *type* == *boolean*) *S*. *type* = S_1. *type*;
  else *S*. *type* = *type_error*; }

$S \to S_1 ; S_2$ { **if** (S_1. *type* == *void* && S_2. *type* == *void*)
   S. *type* = *void*;
  else *S*. *type* = *type_error*; }

5.3.4 类型转换

考虑表达式 $x+i$,其中 x 是实数,i 是整数。因为在计算机中实数和整数有不同的表示,并且对实数和整数有不同的机器指令,因此编译器可能首先把其中一个运算对象进行类型转换,以保证相加的两个对象有同样的类型。

语言的类型系统会指出哪些不同类型的对象在一起运算是允许的,语言的实现会考虑是否要进行类型转换。例如,当将整数表达式赋给实型变量时,应该把赋值号右边对象的类型转换成左边对象的类型。在表达式中,通常是把整数转换成实数,然后在一对实型对象上进行实数运算。编译器的类型检查器可用来在源程序的中间表示中插入这些转换操作。例如,$x+i$ 的后缀式可以是

x *i* **inttoreal** real+

这里,首先由 **inttoreal** 算符把 i 从整数转换成实数,然后由 real+将该转换结果和 x 进行实数加。

如果从一个类型转换到另一类型可以由编译器自动完成,这样的转换称为**隐式的**。例如,在 C 语言的算术表达式中,编译器把 ASCII 字符强制(即隐式转换)为 0 ~ 127 之间的整数。在许多语言中,要求隐式转换原则上不丢失信息,例如,整数可以转变为实数,但反过来不行。不过,当实数和整数用同样位数表示时,还是有可能丢失信息的。

如果转换必须由程序员写出,那么这种转换称为**显式的**。Ada 的所有转换都是显式的。显式转换就是一种函数调用,因此它们不会给类型检查器提出新问题。

例 5.1 考虑把算术算符 **op** 作用于常数和标识符形成的表达式,如图 5.3 的文法那样。假定有实数和整数两个类型,必要时整数转换成实数。非终结符 *E* 的属性 *type* 可以是 *integer* 或 *real*,类型检查在图 5.3 中给出(读者应该能写出相关的定型规则)。 □

$E \to \textbf{num}$ $\{E.type = integer;\}$

$E \to \textbf{num.num}$ $\{E.type = real;\}$

$E \to \textbf{id}$ $\{E.type = lookup(\textbf{id}.entry);\}$

$E \to E_1 \textbf{ op } E_2$ $\{\textbf{if } (E_1.type == integer \ \&\& \ E_2.type == integer)$

$E.type = integer;$

$\textbf{else if } (E_1.type == integer \ \&\& \ E_2.type == real)$

$E.type = real;$

$\textbf{else if } (E_1.type == real \ \&\& \ E_2.type == integer)$

$E.type = real;$

$\textbf{else if } (E_1.type == real \ \&\& \ E_2.type == real)$

$E.type = real;$

$\textbf{else } E.type = type_error;\}$

图 5.3 从整数到实数的类型转换

常数的隐式转换可以在编译时完成,并且常常可以使目标程序的运行时间明显缩短。

*5.4 多态函数

普通的函数要求变元有唯一的类型,即它体中的语句只能在变元类型固定的情况下执行。多态函数允许变元有不同的类型,即它体中的语句适用于不同的调用有不同变元类型的情况。术语"多态"也可用于以不同类型的变元执行的代码段,所以这里既谈多态函数,也谈多态算符。

内部定义的算符,如数组索引、函数应用和通过指针的间接访问等通常都是多态的,因为它们并没有限于特定类型的数组、函数和指针。例如,C 语言的参考手册关于指针 & 的论述是:"如果运算对象的类型是'…',那么结果类型是指向'…'的指针"。因为任何类型都可以代替"…",所以 C 的算符 & 是多态的。

本节讨论为支持多态函数的语言设计类型检查器时出现的问题。为了处理多态性,类型表达式集合需要拓广,增加含类型变量的表达式。类型变量的引入又产生和类型表达式等价有关的一些有趣的算法问题。

5.4.1 为什么要使用多态函数

多态函数的吸引力在于它便于实现那些操作于某种数据结构,而不必顾及其成分类型的算法。例如,用多态函数很容易写出求链表长度的程序而不必管表元的类型。

像 Pascal 这样的语言,它们要求给出函数参数类型的完整说明,所以确定整数链表长度的函数不能用于实数链表。求整数链表长度的 Pascal 代码见图 5.4。函数 length 顺着链表的 next 链

直至 nil 为止。虽然函数代码不以任何方式依赖表元信息的类型,但是 Pascal 要求在编写函数 length 时声明 info 域的类型。

```
type link = ↑ cell;
    cell = record
        info : integer;
        next : link
    end;
function length(lptr : link) : integer;
    var len : integer;
    begin
        len := 0;
        while lptr <> nil do begin
            len := len + 1;
            lptr := lptr↑. next
        end;
        length := len
    end;
```

图 5.4 求表长的 Pascal 程序

在支持多态函数的语言,例如 ML 中,函数 length 可以编写成适用于任何类型的表,如图 5.5 所示。关键字 fun 表示 length 是函数。函数 null 和 tl 是预定义的,null 测试表是否为空,tl 返回拿开第一个表元后剩下的部分。用图 5.5 的定义,函数 length 的下列应用都产生值 3。

```
length (["sun","mon","tue"]);
length ([10,9,8]);
```

在第一个应用中,length 作用于串表;在第二个应用中,它作用于整数表。

要想完成多态函数的类型检查,首先要能给出它们的类型描述。像 length 函数,其参数表的表元类型可以任意,显然应该用变量表示。这样,在类型表达式中需要引入类型变量。允许使用类型变量并引入全称量词 ∀ 后,length 的类型可以写成 $\forall\alpha.\ list(\alpha) \rightarrow integer$。

```
fun length(lptr) =
    if null(lptr) then 0
    else length(tl(lptr)) + 1;
```

图 5.5 求表长度的 ML 程序

5.4.2 类型变量

下面允许类型表达式包含变量,变量的值是类型表达式。本节的其余部分用希腊字母 α 和

β 等作为类型变量，用于类型表达式中。

类型变量除了可出现在多态函数的类型表达式中外，还可以用在其他方面，例如使用类型变量便于讨论未知类型。在不要求标识符的声明先于使用的语言中，类型变量的一个重要运用是检查标识符使用的一致性。类型变量代表没有声明的标识符的类型，查看程序可以知道没有声明的程序变量的使用情况，如果它在某个语句作为整型使用，而在另一语句作为数组使用，那么可以报告使用不一致的错误。相反，如果变量总是作为整型使用，那么不仅能保证它使用的一致性，而且能推断出它的类型必定是整型。

例 5.2 类型推断技术可用于 C 和 Pascal 这样的语言，在编译时补上程序中欠缺的类型信息。图 5.6 的 Pascal 代码段给出过程 mlist，它有一个参数 p，p 本身是个过程。从过程 mlist 的第一行仅能知道 p 是一个过程，不能确定 p 的变元个数和它们的类型。C 和 Pascal 早先的参考手册允许这种类型不完整的 p 声明。

```
type link = ↑ cell;
procedure mlist (lptr : link; procedure p);
begin
    while lptr <> nil do begin
        p (lptr);
        lptr := lptr ↑. next
    end
end;
```

图 5.6 带过程参数 p 的过程 mlist

过程 mlist 把参数 p 用于链表的每个表元。例如，p 可以用于给表元中整型变量置初值或打印其值。尽管 p 的变元类型没有指明，但是从表达式 p(lptr) 中 p 的使用可以推断出 p 的类型必须是

$$link \rightarrow void$$

对于 mlist 的任何调用，如果过程参数不是这个类型，则是一个错误。过程可以看成是没有返回值的函数，所以它的结果类型是 void。 □

类型推断技术和类型检查技术有很多共同的地方。在这两种情况下都要处理含变量的类型表达式。类似于下例的推导在本节将用一个类型检查器来推断变量代表的类型。

例 5.3 在下面假想的程序中，可以推断出多态函数 deref 的类型。函数 deref 和 Pascal 指针的脱引用算符 ↑ 有同样的作用。

```
function deref (p);
begin
    return p ↑
end;
```

看见第一行

function deref (p);

时,对 p 的类型一无所知,因而用类型变量 β 代表它的类型。由定义,后缀算符 ↑ 作用于一个指针,返回该指针指向的对象。因为 ↑ 算符在表达式 p↑ 中作用于 p,因此 p 必定是指向某个对象的指针,让该对象的类型用含另一个类型变量 α 的类型表达式来表示,则可以断定

β = pointer (α)

而且,表达式 p↑ 有类型 α,因此函数 deref 的类型表达式可以写成

$\forall \alpha.$ pointer (α) $\rightarrow \alpha$ □

5.4.3 一个含多态函数的语言

到目前为止,多态函数是指可以用"不同类型"的变元来执行的函数。多态函数的参数所允许的类型集合的精确定义可以用全称量词符号 ∀ 表示,其含义是"对任何类型"。非形式地,含有符号 ∀ 的类型表达式称为"多态的类型"。

$P \rightarrow D; E$

$D \rightarrow D; D$ | **id** : Q

$Q \rightarrow \forall$ **type_variable**. Q | T

$T \rightarrow T$ '→' T

　| $T \times T$

　| **unary_constructor** (T)

　| **basic_type**

　| **type_variable**

　| (T)

$E \rightarrow E$ (E) | E, E | **id**

图 5.7　有多态函数的语言的文法

用于检查多态函数的抽象语言由图 5.7 的文法定义。之所以称为抽象语言,是因为它像图 5.2 的文法那样,忽略了函数定义的函数体,而且还让非终结符 T 直接产生抽象类型表达式,以简化语言和突出要讨论的问题。

该文法对图 5.2 的文法还做了一些简化。首先,程序是由类型声明和表达式(而不是语句)组成,这样可以免去考虑语句,因为语句部分的类型检查不会因加入多态类型而有什么变化。表达式也做了简化,略去了和问题无关的一些选择。用 **basic_type** 类型来表示像 **boolean** 和 **integer** 这样的基本类型,用 **unary_constructor** 表示一元构造符,它允许写出像 *pointer* (*integer*) 和 *list* (*integer*) 形式的类型。

该文法增加了多态类型函数的声明,还增加了笛卡儿积类型的声明。笛卡儿积类型可以用

来表示多元函数类型，产生式 $E \to E, E$ 用以推导多个实参表达式的情况。$T \to (T)$ 仅用来组合类型。

由该文法产生的一个程序实例在图5.8。

```
deref : ∀α. pointer (α) →α;
q : pointer (pointer (integer));
deref (deref (q))
```

图5.8 由图5.7文法产生的一个程序

再来看引入类型变量、笛卡儿积类型和多态函数后增加的推理规则。

下面的环境规则(Env Var)用来把一个类型变量加到静态定型环境中，这样，环境就不再仅仅是程序变量到类型的映射了。

(Env Var) $$\frac{\Gamma \vdash \diamond, \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma, \alpha \vdash \diamond}$$

语法规则增加了4条。规则(Type Var)用来从环境中得到一个类型变量，规则(Type Product)说明积类型的语法，规则(Type Parenthesis)表示类型表达式外加括号后得到的仍然是类型表达式。(Type Forall)是为多态类型准备的规则，它表示加全称量词以形成多态类型。

(Type Var) $$\frac{\Gamma_1, \alpha, \Gamma_2 \vdash \diamond}{\Gamma_1, \alpha, \Gamma_2 \vdash \alpha}$$

(Type Product) $$\frac{\Gamma \vdash T_1, \Gamma \vdash T_2}{\Gamma \vdash T_1 \times T_2}$$

(Type Parenthesis) $$\frac{\Gamma \vdash T}{\Gamma \vdash (T)}$$

(Type Forall) $$\frac{\Gamma, \alpha \vdash \mathrm{T}}{\Gamma \vdash \forall \alpha. T}$$

下面关于积的定型规则

(Exp Pair) $$\frac{\Gamma \vdash E_1 : T_1, \Gamma \vdash E_2 : T_2}{\Gamma \vdash E_1, E_2 : T_1 \times T_2}$$

比较直观，如果两个表达式 E_1 和 E_2 的类型分别是 T_1 和 T_2 的话，那么 E_1, E_2 的类型是 $T_1 \times T_2$。它用于多元函数的情况。

多态类型的函数标识符在使用时要去掉全称量词，用一个新的类型变量代替约束变量。记号 $[\alpha_i/\alpha]\ T$ 表示 T 中自由出现的 α 用 α_i 代换后的结果。

(Exp Id Fresh) $$\frac{\Gamma_1, \mathrm{id} : \forall \alpha. T, \Gamma_2, \alpha_i \vdash \diamond}{\Gamma_1, \mathrm{id} : \forall \alpha. T, \Gamma_2, \alpha_i \vdash \mathrm{id} : [\alpha_i/\alpha]T}$$

对于非多态的函数标识符，仍然用5.3节的Exp Id定型规则。

多态函数调用的定型规则非常复杂，为避免复杂，让一个代换(类型变量到类型表达式的映

射)S 直接出现在类型表达式中,$S(T)$表示代换结果的类型表达式。

(Exp FunCall)(S 是 T_1 和 T_3 最一般的合一代换)

$$\frac{\Gamma \vdash E_1: T_1 \to T_2,\ \Gamma \vdash E_2: T_3}{\Gamma \vdash E_1(E_2): S(T_2)}$$

在上面这条定型规则中,对 S 的限制是,它是 T_1 和 T_3 最一般的合一代换。非多态的函数调用可以统一使用本定型规则。下面先解释代换与合一等概念,然后再给出多态函数检查的翻译方案。

5.4.4 代换、实例与合一

类型表达式中的类型变量用其所代表的类型表达式替换,称之为**代换**。下面的递归函数 *subst*(t)阐明了应用代换 S 去替换不含量词的类型表达式 t 中所有类型变量的概念,函数类型构造符在 *subst* 中被作为“典型”的类型构造符。

```
typeExpression subst (typeExpression t) {
    if (t 是基本类型) return t;
    else if(t 是类型变量) return S(t);
    else if (t 是 t1→t2) return subst(t1) →subst(t2);
}
```

为方便起见,仍用 $S(t)$表示 *subst* 用于 t 后所得的类型表达式,即把 S 拓广为类型表达式到类型表达式的映射,这个结果 $S(t)$叫做 t 的**实例**。如果代换 S 对变量 α 没有指定类型表达式,则假定 $S(\alpha)$仍是 α,即 S 在这样的类型变量上是恒等映射。

例 5.4 下面给出一些类型表达式实例的例子,其中 $s < t$ 用来表示 s 是 t 的实例。

$pointer(integer) < pointer(\alpha)$

$pointer(real) < pointer(\alpha)$

$integer \to integer < \alpha \to \alpha$

$pointer(\alpha) < \beta$

$\alpha < \beta$

但是,下面左边的类型表达式不是右边的实例,原因列在括号中:

integer	*real*	(代换不能用于基本类型)
$integer \to real$	$\alpha \to \alpha$	(α 的代换不一致)
$integer \to \alpha$	$\alpha \to \alpha$	(α 的所有出现都应该代换)

□

如果存在某个代换 S 使得 $S(t_1) = S(t_2)$,那么称类型表达式 t_1 和 t_2 能够**合一**(unify),S 是合一代换。实际感兴趣的是**最一般的合一代换**(the most general unifier),它是对类型表达式中的变量限制最少的代换。更精确地说,类型表达式 t_1 和 t_2 最一般的合一代换是具有下列性质的代换 S:

(1) $S(t_1) = S(t_2)$;

(2) 对任何其他满足 $S'(t_1)=S'(t_2)$ 的代换 S'，代换 $S'(t_1)$ 是 $S(t_1)$ 的实例。

下面所说的合一都是指最一般的合一代换。

5.4.5 多态函数的类型检查

多态函数的类型检查在三个方面和5.3节的普通函数的类型检查有区别。在说明多态函数的类型检查前，用图5.8程序的表达式 deref (deref (q)) 来说明这些不同。这个表达式的语法树在图5.9给出。每个结点有两个标记：第一个标记指出结点代表的子表达式，第二个标记是指派给该表达式的类型表达式。下标 i 和 o 分别用来区分 deref 的里外两次出现。

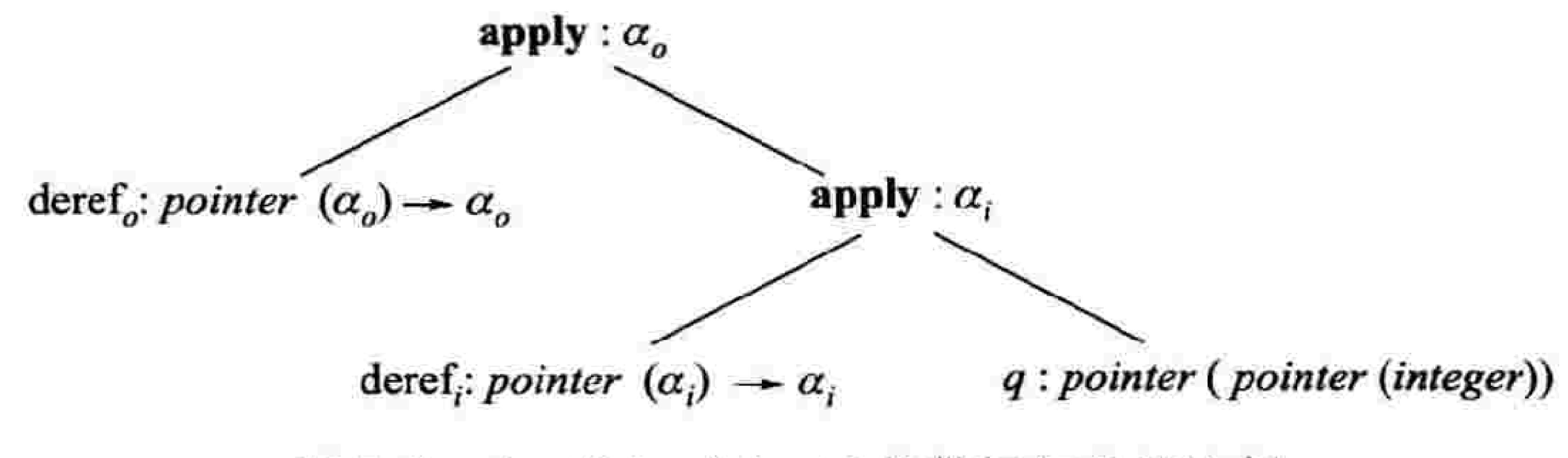


图5.9 deref(deref (q))的带标记的语法树

多态函数的类型检查和普通函数的类型检查的区别如下。

(1) 表达式中同一多态函数的不同出现无须变元有相同类型。在表达式 $\text{deref}_o(\text{deref}_i(q))$ 中，deref_i 拿掉一层指针间接访问，所以 deref_o 和 deref_i 作用于不同类型的变元。这个性质来源于对 deref 函数的类型表达式中的 $\forall\alpha$ 的解释：对任何类型 α。deref 的每次出现对约束变量 α 代表什么类型有它自己的观点。所以对 deref 的每次出现指派一个类型表达式，这个类型表达式是用新的变量代替 deref 的类型表达式中的 α 并且移开全称量词 $\forall$ 后形成的，也就是使用 Exp Id Fresh 定型规则。在图5.9中，新的变量 α_o 和 α_i 分别用于里外两个 deref 的类型表达式中。

(2) 由于变量可以出现在类型表达式中，在某些场合，必须把类型相同的概念推广到类型合一。假如类型为 $s \to s'$ 的 E_1 作用于类型为 t 的 E_2，不是简单地确定 s 和 t 是否相同，而是要看它们能否合一。例如，图5.9右子树上那个标有 **apply** 的结点，如果 α_i 用 *pointer*(*integer*) 代替的话，那么就能使得 *pointer*(α_i) 和 *pointer*(*pointer*(*integer*)) 相同。

(3) 要有办法记录两个子表达式的合一代换。通常，类型变量可以出现在几个类型表达式中。如果 s 和 s' 的合一使得变量 α 代表类型 t，那么在类型检查的过程中 α 必须继续代表 t。例如，图5.9中，α_i 是 deref_i 的值域类型，所以它代表 $\text{deref}_i(q)$（也就是右子树上那个 **apply** 结点）的类型。deref_i 定义域类型和 q 的类型的合一使得 α_i 代表 *pointer*(*integer*)。图5.9的另一个类型变量 α_o 是 $\text{deref}_o(\text{deref}_i(q))$ 的类型，合一运算使得 α_o 代表 *integer*。

现在可以考虑多态函数的类型检查算法了。由图5.7文法产生的表达式的类型检查的翻译方案见图5.10，图中略去了声明语句和其他无关语句的翻译方案。

和5.3节的翻译方案不同的是，当由非终结符 T 和 Q 产生的类型表达式被扫描时，它产生语

法树形式的类型表达式,也就是这里的类型表达式都用语法树来表示。类型表达式图中的叶结点和内部结点可用类似于4.2节的*mkLeaf*和*mkNode*的操作来构造。图5.10的翻译方案还根据下面的操作定义。

(1) *fresh*(t):它把t所指类型表达式中的约束变量用新的变量代替,返回指向替换后的类型表达式的指针。它对应 Exp Id Fresh 定型规则。

(2) *unify*(m,n):它合一由m和n所指向的类型表达式。它有副作用,记住使类型表达式相同的代换S,也就是 Exp FunCall 定型规则中提到的代换S。如果m和n所指类型表达式不能合一,整个类型检查过程失败。本书不打算介绍合一算法。

函数应用$E \to E_1(E_2)$的语义动作需要解释一下,它受到$E_1.type$和$E_2.type$都是类型变量情况的启发。例如,若$E_1.type=\alpha$并且$E_2.type=\beta$,那么α必须是函数,使得存在某个未知类型γ满足$\alpha=\beta\to\gamma$。在图5.10中,对应γ的新类型变量由*newTypeVar*调用获得,并且$E_1.type$和$E_2.type\to\gamma$需要合一而不是相等。合一后,γ代表结果类型。该语义动作和 Exp FunCall 定型规则有些区别,但本质上一样。

$E \to E_1(E_2)$ { $p = mkLeaf(newTypeVar())$;
  $unify(E_1.type, mkNode('\to', E_2.type, p))$;
  $E.type = p$; }
$E \to E_1, E_2$ { $E.type = mknode('\times', E_1.type, E_2.type)$ }
$E \to$ **id** { $E.type = fresh(lookup(\textbf{id}.entry))$; }

图5.10 检查多态函数的翻译方案

用一个简单的例子来说明图5.10的语义动作。表5.1中给出了指派到每个子表达式的类型表达式,总结了这个算法的执行结果。在每一次函数应用时,*unify*操作都有副作用,它记录类型变量代表的类型表达式。这样的副作用由表5.1的代换栏表示。

表5.1 自下而上确定类型的小结

表达式:类型	代换
$q: pointer(pointer(integer))$	
$deref_i: pointer(\alpha_i) \to \alpha_i$	
$deref_i(q): pointer(integer)$	$\alpha_i = pointer(integer)$
$deref_o: pointer(\alpha_o) \to \alpha_o$	
$deref_o(deref_i(q)): integer$	$\alpha_o = integer$

下面一个例子把ML多态函数的类型推断同5.3.3节和图5.10的类型检查联系起来。ML的函数定义的语法由

fun $\mathbf{id}_0(\mathbf{id}_1,\cdots,\mathbf{id}_k)=E$;

给出。其中 $\mathbf{id}_0$代表函数名，$\mathbf{id}_1,\cdots,\mathbf{id}_k$代表它的参数。为简单起见，假定表达式 E 的语法如图5.7所示，并且 E 中的标识符都是函数(包括内部函数)名或参数名。这里的方式是例5.3方法的形式化。在例5.3中，类型推断用来决定 deref 的多态类型；现在，先为函数和它的变元构造新的类型变量。内部函数通常是多态类型的，出现在这些类型中的任何类型变量都由全称量词∀约束。然后检查表达式 $\mathbf{id}_0(\mathbf{id}_1,\cdots,\mathbf{id}_k)$的类型和 E 的类型是否匹配。匹配成功，则得出该函数的类型。最后，将所得函数类型的任何变量都用∀量词约束，得到该函数的多态类型。

例5.5 再次写出图5.5的确定表长度的 ML 函数。

```
fun length(lptr) =
    if null(lptr) then 0
    else length(tl(lptr)) + 1;
```

为 length 和 lptr 分别引入类型变量 β 和 γ 来表示它们的类型。在形式证明中，可以发现 length(lptr)的类型和函数体的类型能够匹配，并且得到 length 的类型是

$$\forall \alpha.\ list(\alpha) \rightarrow integer$$

更详细一点的话，建立图5.11的程序，以便把5.3.3节和图5.10的翻译方案用于它(或者说，把前面所给出的推理规则用于它)。该程序的声明把新类型变量 β 和 γ 同 length 和 lptr 联系起来，并把内部函数的类型显式化。按图5.7的风格，把多态函数 if 作用于三个运算对象，它们分别代表被测试的条件、then 部分和 else 部分的类型。这个声明指出 then 和 else 部分可以和任何类型匹配，它也是结果的类型。

```
length : β;
lptr : γ;
if : ∀α. boolean ×α ×α →α;
null : ∀α. list (α) →boolean;
tl : ∀α. list (α) →list (α);
0 : integer;
1 : integer;
+ : integer × integer →integer;
match : ∀α.α ×α→α;
match (
    length (lptr),
    if (null (lptr),0,length (tl (lptr)) +1)
)
```

图5.11 类型声明及要检查的表达式

显然，length(lptr)必须和函数体有同样的类型，这个检查用运算 match 表示。match 的使用

是一种技术上的方便,它使得所有的检查都可以用图 5.7 风格的一个程序完成。

把图 5.10 的类型检查用于图 5.11 的结果见表 5.2。这里只给出了自下而上分析表达式 match (…)的情况,略去了类型检查用于声明语句的情况。该表的第三列是所用的合一代换,第四列是得出该行定型断言所使用的规则。Exp Id Fresh 定型规则(或者说操作 *fresh*)引入的用于多态的内部函数的新类型变量由 α 的下标区别。

从第(3)行可以知道 length 必须是从 γ 到某个未知类型 δ 的函数。然后,检查子表达式 null(lptr)时,在第(6)行确定 γ 和 $list(\alpha_n)$合一,其中 α_n是未知类型。在该点,length 的类型必须是

$$\forall \alpha_n . list\ (\alpha_n) \to \delta$$

最后,在第(15)行检查加运算时,确定 δ 和 *integer* 合一。为清楚起见,把"+"写在两个变元之间。当类型检查完毕时,类型变量 α_n仍留在函数 length 的类型中。因为对类型 α_n没有任何限制,所以使用该函数时可用任何类型代换它。让它成为约束变量,并写成

$$\forall \alpha_n . list\ (\alpha_n) \to integer$$

这就是 length 的类型。 □

表 5.2 length 的类型推导过程

行	定型断言	代换	规则
(1)	lptr : γ		(Exp Id)
(2)	length : β		(Exp Id)
(3)	length(lptr) : δ	$\beta = \gamma \to \delta$	(Exp FunCall)
(4)	lptr : γ		从(1)可得
(5)	null : $list(\alpha_n) \to boolean$		(Exp Id Fresh)
(6)	null(lptr) : $boolean$	$\gamma = list(\alpha_n)$	(Exp FunCall)
(7)	0 : $integer$		(Exp Num)
(8)	lptr : $list(\alpha_n)$		从(1)可得
(9)	tl : $list(\alpha_t) \to list(\alpha_t)$		(Exp Id Fresh)
(10)	tl(lptr) : $list(\alpha_n)$	$\alpha_t = \alpha_n$	(Exp FunCall)
(11)	length : $list(\alpha_n) \to \delta$		从(2)可得
(12)	length(tl(lptr)) : δ		(Exp FunCall)
(13)	1 : $integer$		(Exp Num)
(14)	+ : $integer \times integer \to integer$		(Exp Id)
(15)	length (tl(lptr)) +1 : $integer$	$\delta = integer$	(Exp FunCall)
(16)	if : $boolean \times \alpha_i \times \alpha_i \to \alpha_i$		(Exp Id Fresh)

续表

行	定型断言	代换	规则
(17)	if (…) : *integer*	$\alpha_i = integer$	(Exp FunCall)
(18)	match : $\alpha_m \times \alpha_m \to \alpha_m$		(Exp Id Fresh)
(19)	match (…) : *integer*	$\alpha_m = integer$	(Exp FunCall)

5.5 类型表达式的等价

5.3节的类型检查有许多“如果两个类型表达式相同,那么返回某个类型,否则返回*type_error*”的情况。到目前为止,关于类型表达式相同的概念是清楚的,它就是下面5.5.1节定义的类型结构等价的概念。

在有些编程语言中,可以给类型表达式命名,并且这些名字可用于随后的类型表达式中。类型名字的使用会出现一些潜在的二义性,关键问题是类型表达式中的名字是代表它自己(则不同名字代表不同类型)还是作为另一个类型表达式的缩写(则不同名字有可能代表相同的类型),由此引出区别于结构等价的名字等价概念。

本节讨论这些等价,这些讨论依据类型表达式的图形表示。在把名字看成类型表达式的缩写的情况下,递归定义的类型将会导致类型表达式的图形表示中出现环。

注意,类型名字的引入只是类型表达式的一个语法约定问题,它并不像引入类型构造符或类型变量那样能丰富所能表达的类型。另外,本节讨论的类型表达式中没有类型变量,即没有考虑多态类型。

5.5.1 类型表达式的结构等价

如果类型表达式仅由类型构造符作用于基本类型组成,则两个类型表达式等价的自然想法是**结构等价**,即两个表达式要么是同样的基本类型,要么是同样的类型构造符作用于结构等价的类型。也就是,两个类型表达式结构等价,当且仅当它们完全相同。例如,类型表达式 *integer* 仅等价于 *integer*,因为它们是同样的基本类型。类似地,*pointer*(*integer*)仅等价于 *pointer*(*integer*),因为它们是把同样的构造符 *pointer* 作用于等价的类型。

在实际使用中,结构等价的概念常常需要修改以反映源语言的实际类型检查规则。例如,当数组作为参数传递时,可能不希望数组的界作为类型的一部分。

图5.12是测试结构等价的算法,假定仅有的类型构造符是数组、积、指针和函数。为简单起见,类型表达式(而不是它的树形表示)直接出现在算法中。这个算法递归地比较类型表达式的

结构而不检查环，所以它能用于类型表达式的树形表示。

```
(1)  boolean sequiv (typeExpression s, typeExpression t) {
(2)      if (s 和 t 是相同的基本类型)
(3)          return true;
(4)      else if (s == array(s1, s2) && t == array(t1, t2))
(5)          return sequiv(s1, t1) && sequiv(s2, t2);
(6)      else if (s == s1 × s2 && t == t1 × t2)
(7)          return sequiv(s1, t1) && sequiv(s2, t2);
(8)      else if (s == pointer (s1) && t == pointer(t1))
(9)          return sequiv(s1, t1);
(10)     else if (s == s1 → s2 && t == t1 → t2)
(11)         return sequiv(s1, t1) && sequiv(s2, t2);
(12)     else return false;
(13) }
```

图 5.12　两个类型表达式 s 和 t 的结构等价测试

如果图 5.12 第(4)和第(5)行的数组等价测试重写为

```
else if (s == array(s1, s2) && t == array(t1, t2))
    return sequiv(s2, t2);
```

那么表示忽略数组的界。

5.5.2　类型表达式的名字等价

在某些语言中，类型可以命名。例如，在 Pascal 的程序段

```
type link = ↑ cell;
var next : link;
    last : link;                    (5.1)
    p : ↑ cell;
    q, r : ↑ cell;
```

中，标识符 link 声明为类型 ↑ cell 的一个名字。由此引出的问题是，变量 next、last、p、q 和 r 都有相同的类型吗？

为了讨论这个问题，允许给类型表达式命名，并且允许这些名字出现在类型表达式中基本类型可以出现的地方。例如，如果 cell 是类型表达式的名字，那么 *pointer*(cell)是类型表达式。暂且假定没有带环的类型表达式定义。

当名字允许出现在类型表达式中时，类型等价的两种不同概念出现了，它们取决于如何看待

名字。按结构等价的观点，先把两个类型表达式中所有的类型名字用它们定义的类型表达式代换，完成代换后的两个类型表达式结构等价的话，则认为原来的两个类型表达式结构等价。按**名字等价**的观点，不同的类型名看成不同的类型，因此两个类型表达式名字等价当且仅当这两个类型表达式不做名字代换就结构等价。历史上，由于对类型等价的不同解释，造成 Pascal 语言的不同编译器实现了略有不同的类型系统。

例 5.6 下面列出了和声明(5.1)的各变量联系的类型表达式。

变量	类型表达式
next	link
last	link
p	*pointer* (cell)
q	*pointer* (cell)
r	*pointer* (cell)

在名字等价下，变量 next 和 last 有相同的类型，因为它们有同样的类型表达式。变量 p，q 和 r 也有相同的类型，但是 p 和 next 类型不相同，因为它们的类型表达式不同。在结构等价下，所有这 5 个变量都有相同的类型，因为 link 是类型表达式 *pointer*(cell)的名字。 □

在不同语言中，标识符和类型通过声明联系的规则是不同的，在解释这些规则时，结构等价和名字等价是两个有用的概念。

5.5.3 记录类型

在介绍递归类型前，先介绍记录类型。

记录类型从某种意义上来说是它各域类型的积，记录和积之间的主要区别是记录的域被命名。把记录构造符 *record* 作用于域名及其类型的二元组序列，就形成记录类型表达式，由此可以写出与记录有关的定型规则如下：

(Type Record) (l_i是有区别的) $$\frac{\Gamma \vdash T_1, \cdots, \Gamma \vdash T_n}{\Gamma \vdash record(l_1 : T_1, \cdots, l_n : T_n)}$$

(Val Record) (l_i是有区别的)

$$\frac{\Gamma \vdash M_1 : T_1, \cdots, \Gamma \vdash M_n : T_n}{\Gamma \vdash record(l_1 = M_1, \cdots, l_n = M_n) : record(l_1 : T_1, \cdots, l_n : T_n)}$$

(Val Record Select) ($1 \leqslant i \leqslant n$) $$\frac{\Gamma \vdash M : record(l_1 : T_1, \cdots, l_n : T_n)}{\Gamma \vdash M.l_i : T_i}$$

根据这些定型规则写一个翻译方案，以完成记录类型和记录域访问的类型检查是很容易的。例如，C 的程序段

```
typedef struct {
    intaddress;
```

```
    char lexeme [15];
} row;
```

定义类型名 row 代表类型表达式

record(address : *int*, lexeme : *array*(15, *char*))

5.5.4 类型表示中的环

链表和树这样的基本数据结构经常是递归定义的,例如,一个链表可以是空表或者是由一个表元和一个链表指针组成。这样的数据结构通常用记录实现,这种记录含指向同类型的记录的指针。在定义这样的记录类型时,类型名起着重要的作用。

考虑链表的每个表元含整型信息和指向下一个表元的指针的情况。用 C 语言来描述的话,表元 cell 的声明可以是

```
typedefstruct _cell {
    int info;
    struct_cell * next;
} cell;
```

cell 的类型表达式如图 5.13(a)所示。如果愿意在类型图中引入环,递归定义的类型名可以替换掉,即用图 5.13(b)的环,则可以删除图 5.13(a)中 *record* 结点以下出现的 cell 结点。

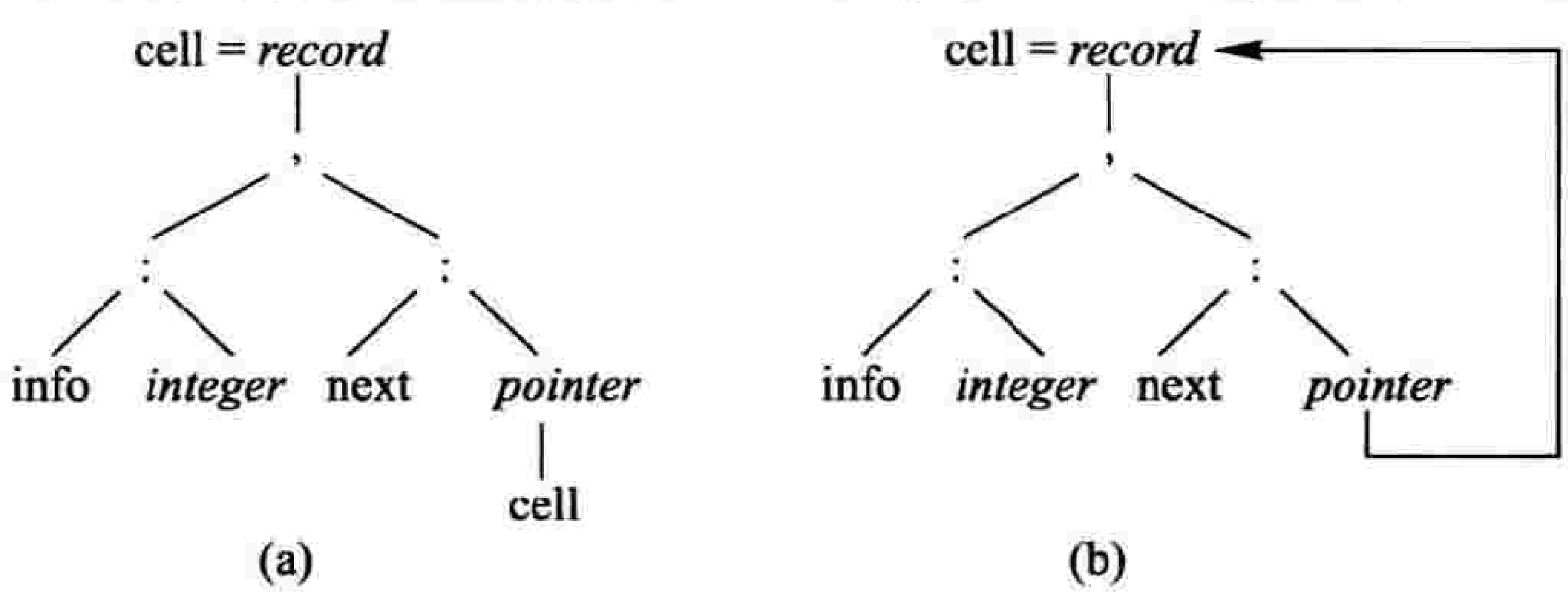


图 5.13 递归定义的类型名

例 5.7 C 语言对除记录(结构体)以外的所有类型使用结构等价,而对记录类型使用的是名字等价,以避免类型图中的环,因为环的出现会使结构等价的判断大大复杂。当碰到记录构造符时,结构等价的测试停止,被比较的类型或者由于它们是同样的命名记录类型而等价,或者它们不等价。因此在效果上,C 使用的是图 5.13(a)的无环表示形式。 □

5.6 函数和算符的重载

和多态性容易混淆的一个概念是符号(包括函数名)的重载,尤其是有些没有多态性的语言,把类似于本书重载的概念定义为多态,使得它们更难区别。**重载(overload)符号**是指该符号有多个含义,但在该符号的每个引用点,其含义可以依赖上下文来唯一地确定。在数学中,加法算符+是重载的,因为当 A 和 B 是整数、实数、复数或矩阵时,$A+B$ 中的+有不同的含义。在 Ada 语言中,括号( ) 是重载的,表达式 $A(I)$ 可能是数组 A 的第 I 个元素的引用,也可能是用变元 I 调用函数 A。多态的符号只有一个含义,但是它允许参数类型在一定的范围内变化。

在重载符号的引用点,若其含义能唯一地确定就称为重载的**消除**。例如,如果+可以表示整数加或实数加,那么在 $x+(i+j)$ 中,+的两个出现可以表示不同形式的加,它取决于 x,i 和 j 的类型。重载的消除有时也称做**算符的辨别**,因为它确定运算符号指称哪个运算。

在大多数语言中,算术算符是重载的。不过,它们的重载可以由仅看它们变元的类型而消除。它们唯一含义的确定类似于图 5.3 中 $E \rightarrow E_1$ **op** E_2 的语义规则,即 E 的类型可以由看 E_1 和 E_2 的类型来确定。

注意,重载的函数和符号的引入使得程序员可以用一个名字或符号表示多个不同类型的函数或运算,它并不像引入类型构造符或类型变量那样能丰富所能表达的类型。

5.6.1 子表达式的可能类型集合

并不总是只看函数的变元就可以消除重载。如下例所示,一个子表达式本身有一个可能类型的集合,而不只是一个类型。在 Ada 中,上下文必须提供足够的信息将这个集合缩小到单元素集合,也就是唯一类型。

例 5.8 在 Ada 中,算符 * 的一个标准(即内部定义)解释是整数集合上的二元函数。加入下面这样的声明:

```
function " * " (i, j : integer) return complex;
function " * " (x, y : complex) return complex;
```

会使得 * 重载。在上述声明后,* 可能的类型包括:

$integer \times integer \rightarrow integer$

$integer \times integer \rightarrow complex$

$complex \times complex \rightarrow complex$

如果从 2、3 和 5 的字面知道它们是整型常量,那么在上述声明的环境下,子表达式 3 * 5 是整型或复型,到底是哪一个则取决于它的上下文。如果完整的表达式是 2 * (3 * 5),那么 3 * 5 必须是整型,因为 * 的两个变元要么都是整型,要么都得复型。相反,如果表达式是(3 * 5) * z 并且 z 是复

型,那么 3 * 5 必须是复型。 □

5.3 节假定每个表达式有唯一的类型,所以函数应用的类型检查是

$$E \rightarrow E_1(E_2) \qquad \{\textbf{if}\ (E_2.type == s\ \&\&\ E_1.type == s \rightarrow t)\ E.type = t;\ \textbf{else}\ E.type = type_error;\}$$

表 5.3 把这条规则自然地推广到有类型集合的情况。表 5.3 仅有的运算是函数应用,表达式中其他算符的类型检查仍类似于前面的检查。重载的标识符可能有几个声明,所以假定符号表的条目包含可能类型的集合,这个集合由 *lookup* 函数返回。开始非终结符 E'产生完整的表达式,它的作用将在下面澄清。

表 5.3 确定表达式可能类型的集合

产生式	语义规则
$E' \rightarrow E$	$E'.types = E.types$
$E \rightarrow \textbf{id}$	$E.types = lookup(\textbf{id}.entry)$
$E \rightarrow E_1(E_2)$	$E.types = \{t \mid E_2.types$ 中存在一个 s,使得 $s \rightarrow t$ 属于 $E_1.types\}$

如果用自然语言叙述,表 5.3 第 3 行的语义规则可以这么说:如果 s 是 E_2的一个类型,并且 E_1的一个类型能把 s 映射到 t,那么 t 是 $E_1(E_2)$的一个类型。函数应用的类型不匹配会使集合 $E.types$ 为空,可以用它作为通知类型错误的条件。

例 5.9 考虑表达式 3 * 5,设算符 * 的声明如例 5.8 所示的那样。即根据上下文,* 可以把一对整数或复数映射到整数或复数。子表达式 3 * 5 的可能类型集合在图 5.14 中,其中 i 和 c 分别是 *integer* 和 *complex* 的缩写。除了解释表 5.3 的语义规则外,这个例子还提示了如何将这种方法应用于其他程序构造。 □

图 5.14 表达式 3 * 5 可能的类型集合

5.6.2 缩小可能类型的集合

Ada 要求完整的表达式有唯一的类型。根据上下文确定了表达式唯一类型,然后据此可以缩小每个子表达式的类型选择。如果这个过程不能使每个子表达式的类型唯一确定,那么宣布该表达式有类型错误。

表 5.4 的语法制导定义由表 5.3 的定义加上定义 E 继承属性 *unique* 的语义规则得到。E 的综合属性 *code* 在下面讨论。因为整个表达式由 E'产生,因此希望 $E'.types$ 是单个类型的集合。这个类型由 $E.unique$ 继承。基本类型 *type_error* 仍是通知错误出现。

如果函数应用 $E_1(E_2)$返回类型 t,那么可以找到类型 s,它对变元 E_2是可行的,同时,$s \rightarrow t$ 对

该函数是可行的。表 5.4 中语义规则的集合 S 用来检查具有这个性质的唯一类型 s。

表 5.4　缩小表达式的类型集合

产生式	语义规则
$E' \to E$	$E'.types = E.types$ $E.unique =$ **if** $(E'.types == \{t\})$ **then** t **else** $type_error$ $E'.code = E.code$
$E \to$ **id**	$E.types = lookup($**id**$.entry)$ $E.code = gen($**id**$.lexeme, ':', E.unique)$
$E \to E_1(E_2)$	$E.types = \{s' \mid E_2.types$ 中存在一个 s,使得 $s \to s'$ 属于 $E_1.types\}$ $t = E.unique$ $S = \{s \mid s \in E_2.types$ and $s \to t \in E_1.types\}$ $E_2.unique =$ **if** $(S == \{s\})$ **then** s **else** $type_error$ $E_1.unique =$ **if** $(S == \{s\})$ **then** $s \to t$ **else** $type_error$ $E.code = E_1.code \parallel E_2.code \parallel gen($'**apply**'$, ':', E.unique)$

表 5.4 的语法制导定义可以用第 4 章结束时扼要介绍的先分析后计算的方式来完成,通过对表达式语法树的两次深度优先遍历实现。在第一次遍历时,属性 *types* 被自下而上地综合。第二次扫描时,属性 *unique* 被自上而下传播,当从结点返回时,*code* 属性被综合。*code* 属性是生成的中间代码,这里用的是后缀表示。在中间表示中,每个标识符和 **apply** 算符的实例都有一个类型。函数 *gen* 表示产生字符串形式的代码,"‖"表示串的连接运算。

习　题 5

5.1　(a) 写一个简单的有控制流错误的 C 程序并提交编译,确认编译器是否报告了这个控制流错误。

(b) 写一个简单的不满足唯一性要求的 C 程序并提交编译,确认编译器是否进行了唯一性检查并报告错误。

5.2　语句的文法如下:

$S \to$ **id** $:= E$ | **if** E **then** S | **while** E **do** S | **begin** $S;S$ **end** | **break**

写一个翻译方案,其语义动作的作用是:若发现 **break** 不是出现在循环语句中,及时报告错误。

5.3　下面是一个 C 语言程序,其中的形参采用传统的声明方式。虽然 main 函数中两次调用函数 gcd 的参数个数都不对,但是该程序能够通过编译并且连接而形成一个目标程序。试问为什么 C 编译器和连接器没能发现这样的错误?

```
long gcd(p,q) long p,q;{
```

```
    if (p% q==0)
        return q;
    else
        return gcd(q,p% q);
}
main() {
    printf("% ld,% ld\n",gcd(5),gcd(5,10,20));
}
```

5.4 为下列类型写类型表达式：

(a) 指向实数的指针数组,数组的下标从 0 到 99。

(b) 二维数组(即元素类型为一维数组的数组),它的行下标从 0 到 9,列下标从 0 到 19。

(c) 函数,它的定义域是从整数到整数指针的函数,它的值域是由一个整数和一个字符组成的记录。

5.5 假如有下列 C 的声明：

```
typedef struct {
    int a,b;
} CELL, * PCELL;
CELL foo[100];
PCELL bar(x,y) int x;CELL y;{…}
```

为变量 foo 和函数 bar 的类型写出类型表达式。

5.6 下列文法定义字面常量表的表。符号的解释和图 5.2 文法的那些相同,增加了类型 **list**,它表示类型 T 的元素表。

$P \rightarrow D;E$

$D \rightarrow D;D \mid$ **id** : T

$T \rightarrow$ **list of** $T \mid$ **char** $\mid$ **integer**

$E \rightarrow (L) \mid$ **literal** $\mid$ **num** $\mid$ **id**

$L \rightarrow E,L \mid E$

写一个类似 5.3 节中的翻译方案,以确定表达式(E)和表(L)的类型。

5.7 把产生式

$E \rightarrow$ **nil**

加入习题 5.6 的文法。含义是表达式可以是空表。修改习题 5.6 的答案,把 **nil** 看成空表,其元素可以是任何类型。

5.8 修改 5.3.3 节的翻译方案,发现错误时打印描述信息,并继续检查,好像所期望的类型已经看见。

5.9 修改 5.3.3 节的翻译方案,使之能处理：

(a) 各种有值语句。赋值语句的值是赋值号右边的表达式的值,条件语句或循环语句的值是语句体的值,语句表的值是表中最后一个语句的值。

(b) 布尔表达式。加上逻辑算符 **and**,**or** 及 **not** 和关系算符的产生式。然后给出适当的翻译规则,它们检查这些表达式的类型。

5.10 推广在5.3节给出的一元函数类型检查,使之能处理 n 元函数。

5.11 C语言是一种类型化语言,但它不是类型可靠的语言,因为运行前的类型检查不能保证所接受的程序没有不会被捕获的错误。例如,编译时的类型检查一般不能保证运行时共用体(union)中数据的使用不出现类型问题。请你再举一个例子说明C语言不是类型可靠的语言。

5.12 拓展5.3.3节的类型检查,使之能包含记录。有关记录部分的类型和记录域引用表达式的语法如下:

T→**record** *fields* **end**

fields →*fields*;*field* | *field*

field →**id** : *T*

E →*E*. **id**

5.13 在文件 stdlib. h 中,关于 qsort 的外部声明如下:

```
extern void qsort(void * ,size_t,size_t,int (*)(const void * ,const void *));
```

下面C程序所在的文件名是 type. c,用某个C编译器编译时,错误信息如下:

```
type. c:18: warning: passing argument 4 of 'qsort' from incompatible pointer type
```

请对该程序略作修改,使得该警告错误能消失,并且不改变程序的结果。

注:程序中关于变量 astHypo 和 n 的赋值以及其他部分被略去。

```
#include <stdlib.h>
typedef struct{
    int       Ave;
    double    Prob;
}HYPO;
HYPO      * astHypo;
int         n;
int HypoCompare(HYPO * stHypo1,HYPO * stHypo2) {
    if (stHypo1->Prob>stHypo2->Prob){
        return(-1);
    }else if (stHypo1->Prob<stHypo2->Prob) {
        return(1);
    }else{
        return(0);
    }
```

```
}/* end of function HypoCompare */
main() {
    qsort (astHypo,n,sizeof(HYPO),HypoCompare);
}
```

5.14　使用类型变量表示下列函数的类型：

(a) 函数 *ref*,它取任意类型的对象作为变元,返回这个对象的指针。

(b) 函数 *arrayderef*,它以一个数组为变元,数组的下标是整型,数组的元素是某类型的指针,返回一个数组,它的元素是变元数组的元素所指向的对象。

5.15　找出下列表达式的最一般的合一代换。

(a) $(pointer\ (\alpha)) \times (\beta \rightarrow \gamma)$

(b) $\beta \times (\gamma \rightarrow \delta)$

如果(b)的 δ 是 α 呢?

5.16　对下面的每对表达式,找出最一般的合一代换。

(a) $\alpha_1 \rightarrow (\alpha_2 \rightarrow \alpha_1)$

(b) $array\ (\beta_1) \rightarrow (pointer\ (\beta_1) \rightarrow \beta_2)$

(c) $\gamma_1 \rightarrow \gamma_2$

(d) $\delta_1 \rightarrow (\delta_1 \rightarrow \delta_2)$

5.17　效仿例 5.5,推导下面 map 的多态类型：

$$\text{map} : \forall \alpha.\ \forall \beta.\ ((\alpha \rightarrow \beta) \times list\ (\alpha)) \rightarrow list\ (\beta)$$

map 的 ML 定义是

```
fun map (f,l)=
    if null (l) then nil
    else cons (f (hd (l)),map (f,tl (l)));
```

在这个函数体中,内部定义的标识符的类型是：

$\text{null} : \forall \alpha.\ list\ (\alpha) \rightarrow \text{boolean}$;

$\text{nil} : \forall \alpha.\ list\ (\alpha)$;

$\text{cons} : \forall \alpha.\ (\alpha \times list\ (\alpha)) \rightarrow list\ (\alpha)$;

$\text{hd} : \forall \alpha.\ list\ (\alpha) \rightarrow \alpha$;

$\text{tl} : \forall \alpha.\ list\ (\alpha) \rightarrow list\ (\alpha)$;

5.18　对于下面的 C 语言程序,在 x86/Linux 系统上,某版本的 GCC 编译器报告第 11 行有错误：

incompatible types in return

在 C 语言中,数组和结构体都是构造类型,为什么下面第二个函数有类型错误,而第一个函数没有?

```
typedef int A1[10];
```

```
typedef int A2[10];
A1 a;
typedef struct {int i;} S1;
typedef struct {int i;} S2;
S1 s;
A2 * fun1() {
    return(&a);
}
S2 fun2() {
    return(s);
}
```

5.19 一个C语言程序如下：

```
void fun(struct {int x;double r;} val) { }
main() {
    struct {int x;double r;} val;
    fun(val);
}
```

该程序在x86/Linux系统上用某版本的GCC编译器编译时，报告的错误信息如下：

```
4: error: incompatible type for argument 1 of 'fun'
1: note: expected 'struct <anonymous>' but argument is not of type 'struct <anonymous>'
```

请问，报告错误的原因是什么？如何修改程序，使得编译时不再出现这个错误信息。

5.20 推广表5.4的算法，使表达式有类型构造符 *array* 和 *pointer*。

5.21 使用例5.9的规则，确定下列哪些表达式有唯一类型（假定z是复数）。

(a) 1*2*3

(b) 1*(z*2)

(c) (1*z)*z

5.22 在C语言的教材上，称&为地址运算符，&a为变量a的地址。但是教材上没有说明表达式&a的类型是什么。另外，教材上说，数组名代表数组的首地址，但是也没有说明这个值的类型。它们所带来的一个问题是，如果a是一个数组名，那么表达式a和&a的值都是数组a的首地址，但是它们的使用是有区别的，初学时很难掌握。

下面给出4个C文件，请根据某版本的GCC编译器报错信息和程序运行结果，判断出表达式a和&a的类型。若能明白它们的类型，那么它们的区别就清楚了，从而可以正确使用它们。

(1) 文件1：

```
typedef int A[10][20];
A a;
```

```
A * fun() {
    return(a);
}
```

该函数在 Linux 上用 GCC 编译时,报告的类型错误如下:

第 4 行:warning: return from incompatible pointer type

(2) 文件 2:

```
typedef   int   A[10][20];
A   a;
A * fun() {
    return(&a);
}
```

该函数在 Linux 上用 GCC 编译时,没有类型方面的错误。

(3) 文件 3:

```
typedef   int   A[10][20];
typedef   int   B[20];
A   a;
B * fun() {
    return(a);
}
```

该函数在 Linux 上用 GCC 编译时,没有类型方面的错误。

(4) 文件 4:

```
typedef   int   A[10][20];
A   a;
fun() {
    printf("%d,%d,%d\n",a,a+1,&a+1);
}
main() {
    fun();
}
```

该程序的运行结果是:

134518112,134518192,134518912

5.23　下面是一个 C 语言文件的内容。

```
long a1[10][10][10];
long a2[ ][10][10];
long a3[ ][ ][10];
```

```
f (a4) long a4[ ][10][10];{
    a1[2][2][2]=a4[2][2][2];
}
main() {
    f (a1);
}
```

把它提交给 GCC 编译器(版本(GNU) 4.2.3 (Debian 4.2.3-5))时,编译器对第 2 行给出警告:array ‘a2’ assumed to have one element。对第 3 行报告错误:array type has incomplete element type。

(a) 为什么第 2 行给出警告,而第 3 行报告错误?

(b) 第 4 行的 a4 和第 2 行的 a2 的描述一样,为什么编译器对 a4 没有警告?

(c) 若在 f 函数中增加输出表达式 sizeof(a2)和 sizeof(a4)的值,你认为编译时会报错吗?

第6章

运行时存储空间的组织和管理

本章把过程和函数这样的程序单元统称为过程,程序运行时过程的一次执行称为过程的一次活动(activation)。过程的每次活动都需要可执行代码和存放所需数据的存储空间,所需的局部数据通常用一块连续的存储区来存放,称之为**活动记录**。过程 p 一次活动的**生存期**(lifetime)是从过程体开始执行到执行结束的时间,包括消耗在执行被 p 调用的过程所需的时间,以及再由这样的过程调用过程所花的时间等。在考虑代码生成之前,需要把静态的程序正文和运行时的活动联系起来,考察静态的名字和运行时数据对象之间的绑定关系。

由于过程可以递归,在程序运行中的某一时刻,可能有多个存活的活动与一个过程对应。因此递归过程中的一个变量可以关联于目标机器上不同的数据对象,虽然任何时刻只有一个这样的对象是可访问的。因此,本章不仅要讨论一个活动记录中的数据布局,还要讨论程序执行过程中,所有存活过程活动的活动记录的组织方式。

当过程执行结束时,一般而言,其局部变量不再可访问。然而,许多语言允许创建对象或其他数据,它们的生存期没有约束在创建它们的过程活动的生存期内。本章还需要讨论这类数据的存储分配。

过程能否递归和程序能否动态创建数据对象都是影响存储分配策略的重要语言特征,影响存储分配策略的主要语言特征如下:

(1) 过程能否递归;

(2) 当控制从过程活动返回时,局部变量的值是否要保留;

(3) 过程能否访问非局部变量;

(4) 过程调用的参数传递方式;

(5) 过程能否作为参数被传递;

(6) 过程能否作为结果值被传递;

(7) 存储块能否在程序控制下动态地分配;

(8) 存储块是否必须显式地回收。

编译器组织运行时的存储空间和把名字绑定到数据单元的方式,在很大程度上取决于对上述问题的回答,这也是本章重点要解决的问题。

面向对象语言和函数式语言特有的一些特征也会影响存储分配策略,其中面向对象语言的

特征将在第12章介绍。

从编译器设计者的观点看,目标程序运行在它自己的逻辑地址空间中,该逻辑地址空间的组织和管理由编译器、操作系统和目标机器共同参与。操作系统把逻辑地址映射到物理地址,后者通常遍及整个内存。因此,本书的存储分配是在逻辑地址空间中讨论。

6.1 局部存储分配

本节讨论一个过程活动所需局部信息的存储分配,先回顾和这个存储分配有关的语言概念,然后介绍活动记录中的数据布局,最后介绍对过程中并列的程序块实行重叠分配的办法。

6.1.1 过程

过程定义是一个声明,它的最简单形式是将一个名字和一个语句联系起来。该名字是**过程名**,而这个语句是**过程体**。在大多数语言中,返回值的过程称为**函数**,完整的程序也可以看作一个过程。

过程名出现在调用语句中则称这个过程在该点**被调用**。过程调用就是执行被调用过程的过程体。过程调用也可以出现在表达式中,这时称为函数调用。

出现在过程定义中的某些名字是特殊的,它们被称为该过程的**形式参数**,简称形参。过程调用语句中的**实在参数**(简称实参)传递给被调用过程,它们取代过程体中的形参。建立实参和形参对应的方法在6.4节讨论。图6.1是用C语言写的快速排序程序的概略,除main外,它还有3个过程。

6.1.2 名字的作用域和绑定

语言中的声明是把信息联系到名字的一种语法构造。在程序的不同部分可能有同一名字的互相独立的声明,语言的**作用域规则**规定了一个名字在程序中出现时,该名字的哪个声明应用到这个出现。在图6.1的程序中,i在两个过程中都有声明,但它们的使用互相独立。

一个声明起作用的程序部分称为该声明的**作用域**。过程中出现的名字,如果是在该过程中这个名字的某个声明的作用域内,那么称这个出现局部于该过程;否则称为非局部的。局部和非局部的区分也适用于其他任何可包含声明的语法构造。作用域是名字声明的一个性质。为简单起见,通常用“名字x的作用域”来代替“用于名字x这个出现的x声明的作用域”。

即使一个名字在程序中只声明一次,该名字在程序运行时也可能代表不同的数据对象。非正式的术语“数据对象”指的是保存值的存储单元。

在编程语言的语义中,由于上面所讲的原因,通常用环境和状态来表示变量名字到值的映

```
int a[11];
void readArray(){/* Reads 9 integers into a[1],…,a[9]. */
    int i;
    …
}
int partition(int m,int n){
    /* Picks a separator value v,and partitions a[m..n] so that a[m..p-1] are less
       than v,a[p]=v,and a[p+1..n] are equal to or great than v. Returns p. */
    …
}
void quickSort(int m,int n){
    int i;
    if(n>m){
        i=partition(m,n);
        quickSort(m,i-1);
        quickSort(i+1,n);
    }
}
main(){
    readArray();
    a[0]=-9999;
    a[10]=9999;
    quickSort(1,9);
}
```

图 6.1　快速排序程序的概略

射。术语**环境**表示将名字映射到存储单元的函数,术语**状态**表示将存储单元映射到它所保存值的函数,如图 6.2 所示。也可以说,环境把名字映射到其左值,而状态把名字的左值映射到名字的右值。左值和右值的概念源于 C 语言,分别指变量的地址和值,因为对于赋值号左右两边的变量,被关注的分别是它们的地址和值。区别状态和环境是有意义的,赋值改变状态,但不改变环境;过程调用改变环境。

图 6.2　从名字到值的两步映射

如果环境将名字 x 映射到存储单元 s,就说 x 被**绑定**(binding)到 s。术语“存储单元”是象征性的,因为如果 x 不是基本类型的话,那么 x 的存储单元 s 可能是内存中的若干个字。

在学习本章时，应注意区别一些静态的概念和它们的动态对应物，见表6.1。尤其要注意，在某一时刻，递归过程可以有不止一个活动存活，递归过程的局部变量名字在该过程的不同活动中绑定到不同的存储单元。

表6.1 静态概念和动态概念的对应

静态概念	动态概念
过程的定义	过程的活动
名字的声明	名字的绑定
声明的作用域	绑定的生存期

6.1.3 活动记录

过程一次执行所需局部信息用一块连续的存储区来管理，这块存储区叫做活动记录或帧(frame)，它由图6.3的各个域组成。不同的语言，以及同一语言的不同编译器，它们所使用的域可能不同，这些域在活动记录中的布局也可能不同，另外，寄存器往往可以取代它们中的一个或多个域。

活动记录各个域的用途如下。

(1) 临时数据。保存临时值，例如寄存器不足以存放表达式计算的中间结果时，可以把中间结果存放在这儿。

(2) 局部数据。保存过程的局部数据，这个域的具体布局在后面还会讨论。

(3) 保存的机器状态。保存刚好在过程调用前的机器状态信息，典型信息包括返回地址(程序计数器的值，它是被调用过程必须返回的地址)，还有调用过程使用并且在返回时必须恢复的寄存器的内容。

图6.3 一般的活动记录

(4) 访问链。有些语言需要通过访问链来访问非局部数据，6.3节将详细介绍。

(5) 控制链。用来指向调用者的活动记录。

(6) 返回值。用于存放被调用过程返回给调用过程的值。为提高效率，这个值也常常用寄存器返回。

(7) 参数。存放调用过程提供的实在参数，由被调用过程使用。为提高效率，实际上常常用寄存器传递参数。6.4节介绍各种参数传递方式的实现。

每个域的长度都可以在过程调用时确定。事实上，几乎所有域的长度都可以在编译时确定。一个例外是，如果过程中存在只有在过程开始执行后才能确定大小的局部数组，那么只有在执行到过程

调用时才能确定局部数据域的大小,6.2.4 节将讨论活动记录中可变长度的数据的分配方法。

活动记录没有包含过程一次执行所需的全部信息,例如非局部数据就不在这个活动记录中,6.2 节和 6.3 节介绍非局部数据的布局和访问方法。还有,在程序控制下动态地分配的数据对象通常放在堆上,6.5 节讨论堆分配。

6.1.4 局部数据的布局

假定运行时存储空间是连续字节区域,其中字节是可编址内存的最小单位。一个字节为 8 位,几个字节形成一个机器字。多字节数据对象存储于连续的字节中,并以第一个字节的地址作为该对象的地址。

变量所需的存储空间可以由它的类型确定。一个基本类型的数据,如字符、整数或实数,可以用几个连续字节保存。对于数组,在分配给它的存储区内,数组元素依次存放,以便计算下标变量的地址。对于记录,在分配给它的存储区内,它的域通常按类型声明时出现的次序存放。

在编译时,一个过程所声明的局部变量按它们声明时出现的次序,在活动记录的局部数据域中依次分配空间。这些局部数据的地址可以用**相对于**活动记录中某个位置的相对地址来表示,例如相对于活动记录的开始点,或者相对于活动记录中部的某个特定单元。相对地址(或者称**偏移**)就是指数据对象和这个位置的地址差,活动记录中其他域的访问也可以用相对于这个位置的偏移来处理。

数据的存储布局还受目标机器寻址限制的影响。例如,整数加的指令可能要求被相加的整数必须**对齐**(alignment),即它们被放在内存中满足一定条件的位置,例如能被 4 整除的地址。例如,10 个单字符的数组只需要 10 个字节,假如下面紧接着安排一个整数的话,编译器会跳过 2 个字节后再进行分配,以保证该整数的地址是 4 的倍数。由于考虑对齐而引起的无用空间称为**衬垫区**(padding)。如果空间很宝贵,编译器可能紧凑数据布局,使得没有任何衬垫区出现,但是运行时可能要执行一些额外的指令来取出这些紧凑布局的数据,然后才能对它们进行操作。

6.1.5 程序块

程序块(block,又翻译成分程序)是本身含有局部变量声明的语句。程序块的一个特点是它的嵌套结构,即不可能出现程序块 B_1 先于 B_2 开始执行,又先于 B_2 结束执行的情况,这种嵌套性又称作程序块结构。程序块结构中声明的作用域是按最接近的嵌套规则给出的。图 6.4 是有嵌套程序块的 C 程序,其中变量声明的作用域见表 6.2。4 个打印语句按照 B_2, B_3, B_1 和 B_0 的次序执行,即控制离开这些程序块的次序。在这些程序块中,a 和 b 的值分别为:

```
2    1
0    3
0    1
0    0
```

```
main( ) {                                    /* begin of B0 */
    int a=0;
    int b=0;
    {                                    /* begin of B1 */
        int b=1;
        {                                /* begin of B2 */
            int a=2;
            printf("%d %d\n",a,b);
        }                                /* end of B2 */
        {                                /* begin of B3 */
            int b=3;
            printf("%d %d\n",a,b);
        }                                /* end of B3 */
        printf("%d %d\n",a,b);
    }                                    /* end of B1 */
    printf("%d %d\n",a,b);
}                                            /* end of B0 */
```

图 6.4 C 程序的程序块

表 6.2 图 6.4 中各声明的作用域

声明	作用域
int a=0;	B_0-B_2
int b=0;	B_0-B_1
int b=1;	B_1-B_3
int a=2;	B_2
int b=3;	B_3

如果过程中有嵌套程序块,那么编译器在存储分配时,也要为内嵌程序块中声明的变量留出所需的空间。对于图 6.4 的程序,可以按图 6.5 所示来分配存储空间,局部变量 a 和 b 的下标用来标识它们的声明所在的程序块。注意,a_2和 b_3可以重叠分配,因为它们所在的程序块不会同时活跃。

有一点需要注意,在确定过程所需局部存储空间时,是稳妥地假设过程运行时会走遍所有的控制路径,即认为条件语句的 then 和 else 部分都会执行,循环语句中循环体的各语句也都会执行。因此,过程中有嵌套的程序块,不会影响活动记录中局部数据域大小的静态确定。

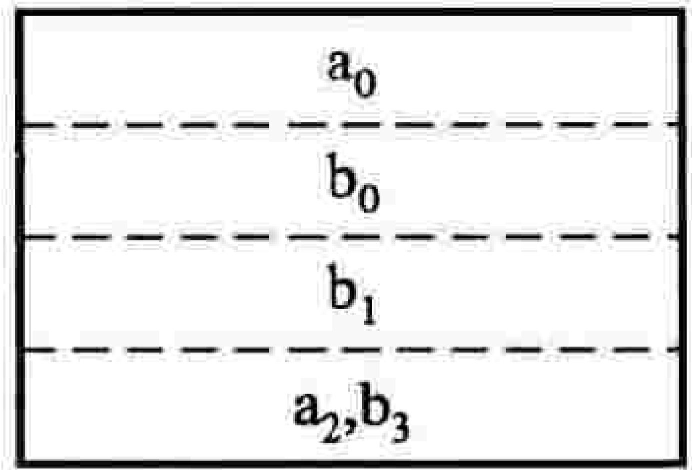


图 6.5　图 6.4 的程序所声明变量的存储单元

6.2　全局栈式存储分配

上节介绍的是单个活动记录内数据的布局，本节介绍程序运行时所有活动记录在存储空间的组织，并描述过程的目标代码怎样访问绑定到局部名字的存储单元。本节所讨论的栈式存储分配可用于 C、FORTRAN 和 Pascal 等面向过程的语言，也可用于 Java 和 C++等面向对象的语言。

6.2.1　运行时内存的划分

在逻辑地址空间，一个程序运行时所用空间像图 6.6 那样，由程序区和若干个数据区组成。例如，在 Linux 操作系统上，C 编译器就是按这种方式将内存细分的。

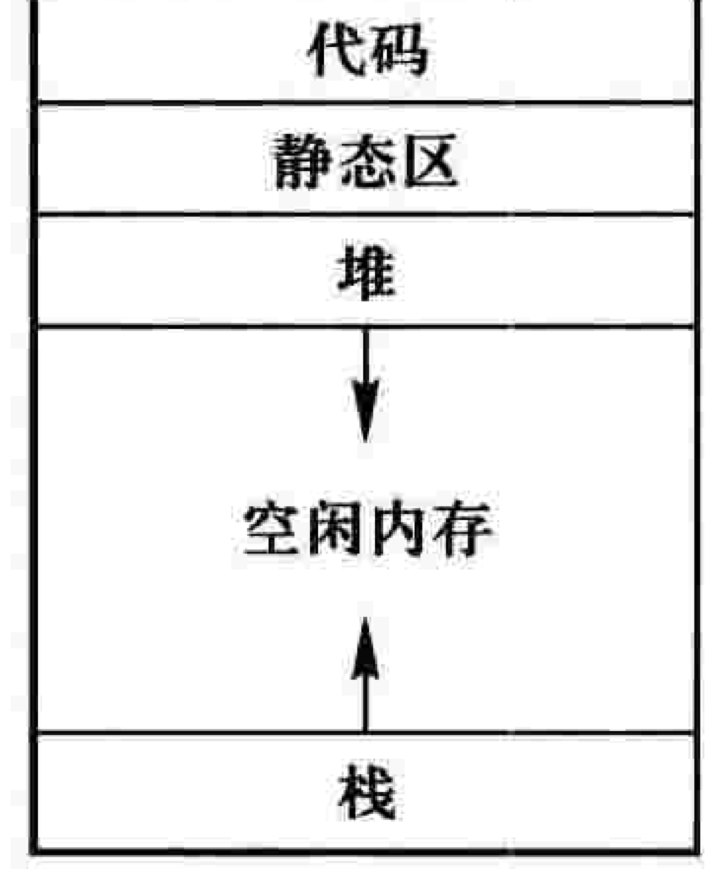


图 6.6　运行时内存空间的划分

目标代码的长度在编译时即可确定，并且运行时不会改变，通常安排在内存的低地址区。一些程序数据，例如全局常量，还有编译器产生的数据，例如支持垃圾收集的信息，它们的存储大小在编译时都可以确定，因此把它们安排在静态确定的数据区中。尽可能把数据安排在静态区的一个理由是，这些数据的地址可以编译到目标代码中去，以提高运行时对这些数据的访问速度。在 FORTRAN 语言的早期版本中，所有数据都可以静态分配，因为它们不允许递归过程。

为了在运行时充分利用存储空间，栈和堆一般安排在地址空间剩余部分的两端。它们是动态的，在程序运行过程中它们占用的存储空间不断变化，相向增长。在实际机器上，栈向低地址方向增长，堆向高地址方向增长。

像 C、FORTRAN 和 Pascal 这样的语言，由于过程递归的次数一般不是静态可确定的，因此活动记录不可能静态分配，虽然每个活动记录的大小可以静态确定。另一方面，由于一个过程活动终止时其活动记录不再需要，又由于过程活动的生存期要么嵌套，要么无重叠，因此可以把当前所有存活过程活动的活动记录组织成一个栈。这种存储分配策略从 6.2.2 节开始讨论。

许多编程语言允许程序员在程序控制下分配和释放存储块，如 C 语言的函数 malloc 和 free 用于此目的。这些存储块中数据对象的生存期可能比要求分配这些存储块的过程更长，因此它们的生存期不遵守栈式规则。堆用来管理这类长命数据。数据放在栈上比放在堆上的开销要小一些，这是由它们对数据的分配和回收方式决定的，6.5 节专门讨论堆管理。

第 10 章还会对运行时内存空间的组织做更细一点的介绍。

6.2.2 活动树和运行栈

6.2.1 节已经提到，对于常见的面向过程的语言，如果 *a* 和 *b* 都是过程活动，那么它们的生存期或者嵌套，或者无重叠。也就是说，如果控制在离开 *a* 之前进入 *b*，那么控制在离开 *b* 之后才能离开 *a*。由此可知，可以用树来描绘控制进入和离开活动的次序，这样的树称为**活动树**。在活动树中：

（1）每个结点代表一个过程活动；

（2）根结点代表主过程的活动；

（3）结点 *a* 是结点 *b* 的父结点，当且仅当控制流从活动 *a* 进入活动 *b*；

（4）结点 *a* 处于结点 *b* 的左边，当且仅当活动 *a* 的生存期先于活动 *b* 的生存期。

因为结点和活动一一对应，所以当控制处于某结点代表的活动时，就直接说控制处于这个结点。

例 6.1 图 6.1 程序的活动树画在图 6.7 中，其中过程名都用它的第一个字母表示，括号中的数据是活动的实参。程序的控制流对应于活动树的后根遍历。 □

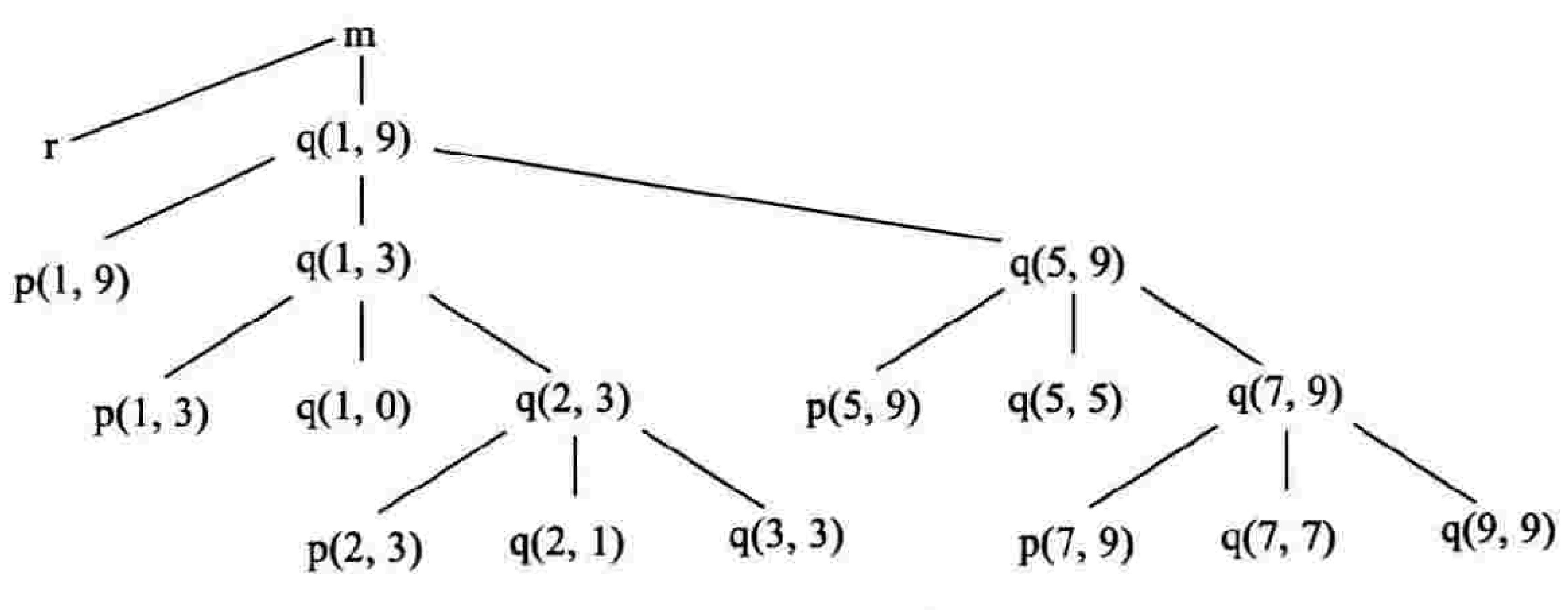


图 6.7 图 6.1 程序的活动树

例 6.2 图 6.8 给出了控制进入 q(2,3) 代表的活动时，已经执行完毕和正在执行的活动。其中已经执行完毕的活动在虚线的下端，正在执行、尚未结束的活动在根结点到结点 q(2,3) 的这条实线路径上。尚未开始的活动在图中没有画出。 □

从图 6.8 可以看出，当前活跃的过程活动可以保存在一个栈中。当活动开始时，把这个活动的结点压入栈中，当它结束时，把它的结点从栈中弹出。这样的栈被称为**控制栈**。对于图 6.8 的情况，控制栈的内容从栈底到栈顶依次为：

m, q(1,9), q(1,3), q(2,3)

如果控制栈中的信息包含过程活动的活动记录，控制栈就成了活动记录栈，通常称之为**运行栈**。当一个过程被调用时，它的一个新活动记录被压入栈，局部变量被绑定到其中的存储单元；当对应这次调用的活动终止时，该活动记录被退栈，局部变量不再绑定到其中的存储单元，恢复原来的绑定（如果有的话）。由于一个过程的每次调用都会引起一个新的活动记录进栈，所以过程的每次活动都会把局部变量绑定到新的存储单元。

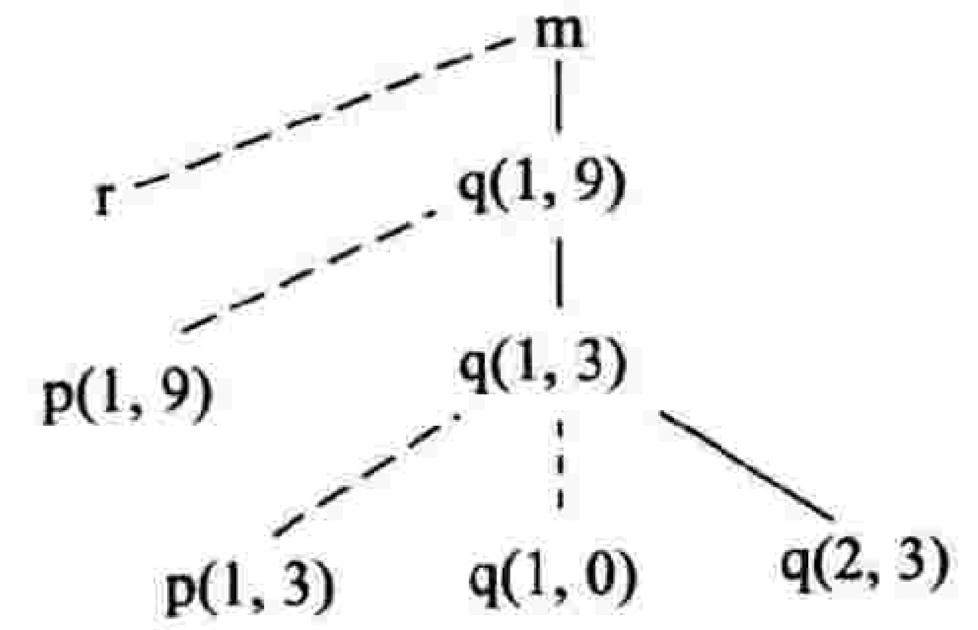


图 6.8　控制栈包含实线上的结点

例 6.3　图 6.9 表示当控制流通过如图 6.7 所示的活动树时，活动记录压入运行栈和从运行栈退出的几个瞬像（snapshot，也称快照），树上的虚线仍然引向已经结束的活动。注意，因为数组 a 是全局的，它分配在静态区，因此没有出现在该图上。程序执行开始时有过程 m 的活动，当控制到达 m 体中第一个调用时，过程 r 被激活，它的活动记录被压入栈。当控制从该活动返回时，栈顶活动记录被退栈，栈中仅剩下 m 的活动记录。在 m 的继续执行过程中，当控制到达以 1 和 9 为实参的对过程 q 的调用时，q 的一个活动记录被压入栈。

活动树	栈上的活动记录	注释
m	m	m的活动记录
m r	int i r m	r 被激活
m r　q(1, 9)	int i q(1, 9) int m, n m	r 的活动记录出栈 q(1, 9)的活动记录进栈
m r　q(1, 9) p(1, 9)　q(1, 3) p(1, 3)　q(1, 0)	int i q(1, 3) int m, n int i q(1, 9) int m, n m	控制返回到 q(1, 3)

图 6.9　活动记录在栈中的分配（栈向上长）

图6.9的最后两个瞬像之间有多个活动发生。在最后一个瞬像中，活动p(1,3)和q(1,0)已经开始并终止在q(1,3)的生存期里中，它们的活动记录也都已经压栈并退栈，因此只留下q(1,3)的活动记录在栈顶。 □

6.2.3 调用序列

从先前的介绍可以知道，过程调用和过程返回都需要执行一些代码来管理运行栈：分配和回收活动记录、保存或恢复机器状态等。在过程调用时执行的分配活动记录，以及把信息填入其域中的代码被称为**过程调用序列**；而在过程返回时执行的恢复机器状态，回收活动记录，以及让调用过程继续执行的代码被称为**过程返回序列**。

即使是同一种语言，过程调用序列、过程返回序列和活动记录的布局也会因实现而异。过程调用序列的代码常常分成两部分，分别位于调用过程和被调用过程中。过程调用序列在这两者之间的划分也不是唯一的。源语言、目标机器和操作系统强加的约束可能使得某种方法比另一些方法更合适。过程返回序列也是如此。

下面一些原则对设计调用序列、返回序列和活动记录布局是有益的。

(1) 调用者和被调用者之间交流的数据一般放在被调用者活动记录的开始处，尽可能靠近调用者的活动记录。这样做便于调用者把实参的值放到自身活动记录的顶端，而不需要此刻就建立被调用者完整的活动记录，甚至不用知道它的布局。这样做还允许使用参数个数可变的过程，如C语言的标准库函数printf，这个问题后面再讨论。被调用者也知道返回值应该放在靠近调用者活动记录的地方，不管有多少个实参，它们都正好在返回值的下面。

(2) 固定长度的项通常放在活动记录的中间。它们一般包括控制链、访问链和机器状态域。如果每次调用时正好保存机器状态中同样的成分，那么可以执行同样的代码来完成机器状态的保存和恢复。而且，如果机器状态信息是标准化的，那么出现错误时，调试器等程序可以很轻松地解读运行栈的内容。

(3) 对编译时不能及时知道其大小的一些项，放在活动记录的末端。大多数局部变量有固定的长度，这些长度能够由编译器在检查变量类型时确定。但是，有些局部变量的大小在程序运行前一直不能确定，最常见的例子是动态大小的数组(其大小依赖于过程参数的数组)。还有，所需的临时数据空间的大小通常依赖于代码生成阶段怎样尽可能地把临时数据存放在寄存器中。因此，虽然临时数据空间的大小在编译时最终可以知道，但它可能不是在中间代码生成时就知道的。

作为一个例子，图6.10给出一种调用者和被调用者怎样协作管理栈的建议(无须访问链的语言)。寄存器top_sp指向栈顶活动记录的末端；另一个寄存器base_sp指向栈顶活动记录中控制链所在的位置，它作为访问栈顶活动记录中内容的基地址。假定过程p调用过程q，调用序列如下。

(1) p计算实参，依次将它们压入栈顶，也就是放到q的活动记录中，并在栈顶上留出放返

回值的空间。在这些操作过程中,top_sp 的值在不断减小。

(2) p 把返回地址压入 q 的活动记录,把控制转到 q。

(3) q 将要保存的寄存器(base_sp 除外)的值和其他机器状态信息压入栈,然后把 base_sp 的当前值(p 的 base_sp)压入栈,并把当前 top_sp 的值作为自己 base_sp 的值。

(4) q 根据局部数据域和临时数据域的大小来减小 top_sp 的值,也就是进行局部数据和临时数据的空间分配,并初始化它的局部数据,开始执行过程体,如图 6.10 所示。

在上述第(3)步中,把 p 的 base_sp 压入栈就是建立控制链,由于控制链是按运行时过程调用关系从被调用者的活动记录指向调用者的活动记录,因此控制链也称为**动态链**。

返回序列如下。

(1) q 把返回值置入活动记录中存放返回值的地方。

(2) 与调用序列的步骤(4)相对应,q 增加 top_sp 的值。

(3) q 恢复寄存器(包括 base_sp)和机器状态,把控制转到 p。

(4) 与调用序列的步骤(1)相对应,p 增加 top_sp 的值,并取出返回值。

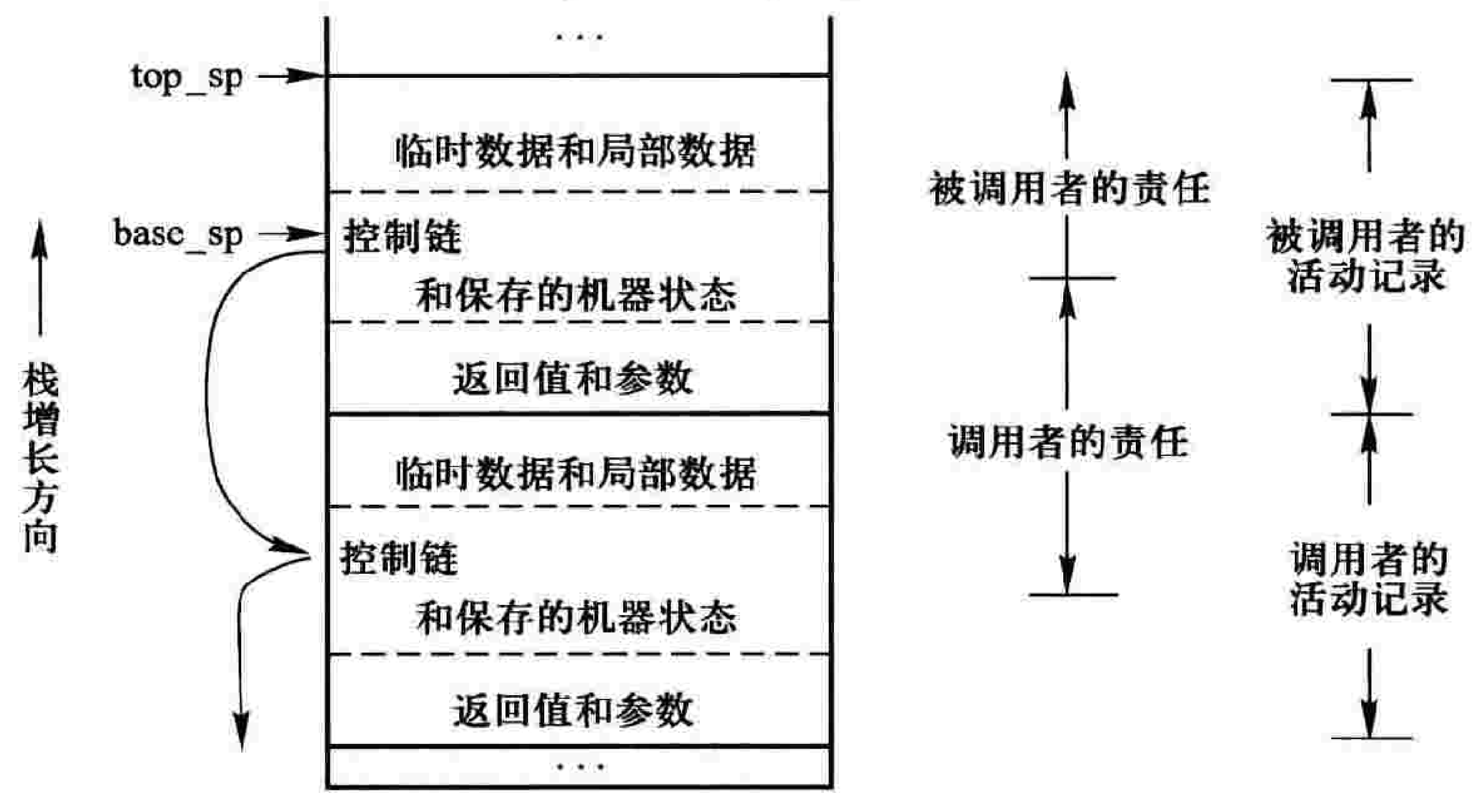


图 6.10 调用者和被调用者之间的任务划分

例 6.4 一个 C 语言函数如下:

```
func(i)long i;{
    long j;
    j=i-1;
    func(j);
}
```

为使生成的汇编代码尽量少,用了这个不会终止的递归函数。在 x86/Linux 系统上用版本标识为 GCC:(GNU)egcs-2.91.66 19990314/Linux(egcs-1.1.2 release)的 GCC 编译器,为该函数生成的汇编代码(略去与问题无关的部分)及添加的注释如下:

```
func:
      pushl  %ebp                  //将老的基地址指针压栈
      movl  %esp,%ebp              //将当前栈顶指针作为基地址指针
      subl  $4,%esp                //为局部变量j分配空间
      movl 8(%ebp),%edx            //取形参i的值到寄存器,i的地址是8(%ebp)
      decl  %edx                   //i-1
      movl  %edx,-4(%ebp)          //i-1的值赋给j,分配给j的地址是-4(%ebp)
      movl-4(%ebp),%eax            //和下一条指令一起,完成将实参j的值压栈
      pushl  %eax
      call  func                   //函数调用,将返回地址压栈,修改程序计数器
      addl  $4,%esp                //栈顶指针恢复到参数压栈前的位置
.L1:
      leave                        //相当于movl %ebp,%esp和popl %ebp
      ret                          //相当于popl %eip,将esp恢复到调用前参数压栈后
                                   //的位置,返回调用者。eip存放下步执行指令的地址
```

从上面的汇编代码可以分析出该函数一个活动记录的内容,如图6.11所示。其中esp是栈顶指针寄存器(执行movl 8(%ebp),%edx指令时栈顶指针所指的位置就是图中所标明的位置),ebp是基地址寄存器。

在这个例子中,调用序列的代码在调用者的部分是pushl %eax和call func,在被调用者的部分是pushl %ebp、movl %esp,%ebp和subl $4,%esp。返回序列的代码在被调用者的部分是leave和ret,在调用者的部分是addl $4,%esp。

注意,现在流行的较高版本GCC编译器,所分配的局部数据空间比局部变量的实际需求略大一些,一个原因是为了栈对齐,另一个原因是预留一点空间。有兴趣的读者可以参考相关资料。还需要注意一点,为提高参数传递的效率,不是用push指令将参数逐个压栈,而是先修改esp的值,空出所需的参数区,然后用movl指令将实参逐个进栈。 □

图6.11 活动记录的内容及相关信息

上面的调用序列和返回序列可用于过程的参数个数可变的情况,以C语言的标准库函数printf为例。在一个程序中,每次printf调用的实参个数是清楚的,编译器产生将这些实参逆序压栈的代码,被调用函数printf虽然不知道参数的个数,但是它能准确地知道第一个参数的位置。

因此 printf 的实现首先取第一个参数——格式控制字符串,然后分析它的格式控制要求,根据格式控制中的格式说明,到栈中取第二、第三个参数等。

6.2.4 栈上可变长度数据

在现代编程语言中,通常把编译时不能确定所需空间的数据分配到堆上。但是有些情况下把数组、对象和其他未知大小的结构分配到栈上也是可能的,尽可能把它们分配在栈上而不是堆上的理由是避免垃圾收集器收集它们的开销。注意,只有那些局部于一个过程并在该过程返回后不可访问的数据对象才能分配到栈上。本节以动态数组为例,介绍编译时如何将不能确定所需空间的数据分配到栈上,该方案也可用于局部于过程并且大小依赖于过程参数的其他类型的数据。

在图 6.12 中,局部数组的大小要等到过程激活时才能确定。过程 p 有两个不能静态确定大小的局部数组,在编译时,在活动记录中为这两个数组分别分配一个存放数组指针的单元。运行时,这两个数组的大小能确定后,在栈顶为这些数组分配空间,并把起始地址置入存放数组指针的单元。这样对这些数组的访问是通过活动记录中的数组指针间接进行的。

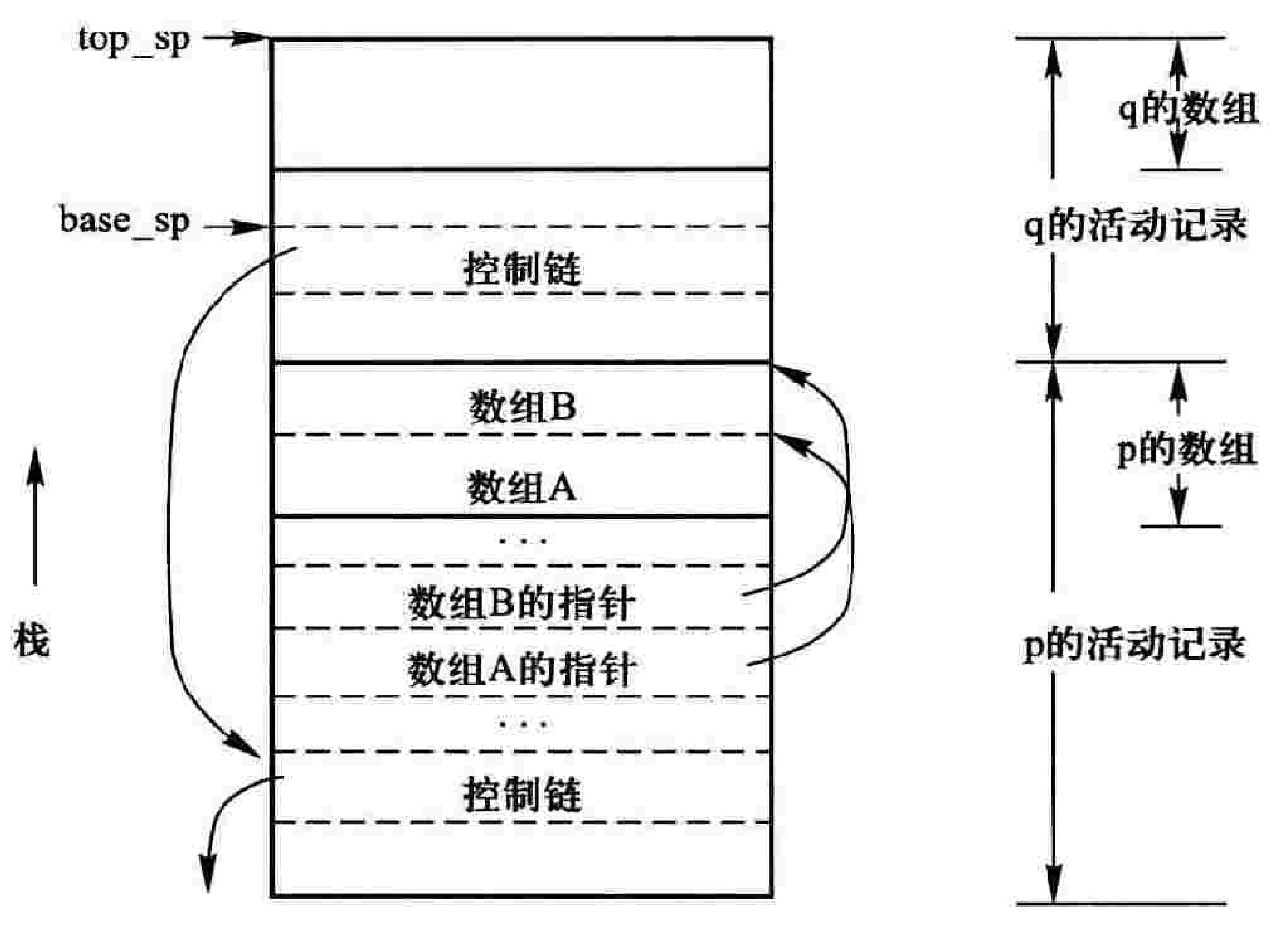


图 6.12 访问动态分配的数组

6.2.5 悬空引用

只要存储空间可以回收,就有可能出现悬空引用问题。引用某个已被回收的存储单元就称为**悬空引用**(dangling reference)。使用悬空引用是一种逻辑错误,因为按大多数语言的语义,已

被回收的存储单元的值是没有定义的。更糟糕的是,已被回收的存储单元可能随后被分配用来存放其他数据,因此有悬空引用错误的程序会出现难以理解的、不会被捕获的错误。悬空引用在栈上和堆上都有可能出现,下面是一个在栈上出现悬空引用的例子。

例 6.5 在图 6.13 的 C 语言程序中,过程 dangle 返回一个指向绑定到局部名 j 的存储单元的指针。当控制从 dangle 返回到 main 时,dangle 的活动记录已经回收,并可能已另有安排。因为 main 中 q 的值是这个存储单元的地址,q 被称为**悬空指针**,对 q 指向的对象的访问就是一个悬空引用。 □

```
int * dangle( ) {
    int j=20;
    return &j;
}
main( ) {
    int *q;
    q=dangle( );
}
```

图 6.13 q 指向已经回收了的存储单元

6.3 非局部名字的访问

语言的作用域规则规定了如何处理非局部名字的访问。一种常用的规则是**词法作用域**或**静态作用域**规则,它仅根据程序正文静态地确定适用于一个名字的某个引用性出现的该名字的声明(简单说为用于名字的声明)。许多语言,如 C、Java 和 Pascal 等,都使用静态作用域规则。本节首先考虑 C 语言那样的非局部名字。由于 C 语言不允许嵌套的过程声明,因此所有的非局部名字都可以静态地绑定到所分配的存储单元。

像 Pascal 和 ML 等语言,它们允许过程嵌套,并使用静态作用域,确定用于名字的声明需要根据过程的嵌套层次来决定。和 C 语言不同的是,Pascal 语言的非局部名字不一定就是全局名字。当运行时访问非局部名字,首先要确定该非局部名字被绑定到的活动记录,本节讨论利用访问链寻找该活动记录的一种方法。

另一种作用域规则是动态作用域规则,它是在运行时根据当前存活的过程活动来确定用于名字的声明。使用动态作用域的语言很少,如 LISP。作为比较,6.3.3 节将讨论动态作用域的实现。

6.3.1 无过程嵌套的静态作用域

由于C语言不允许过程嵌套,C语言的程序由变量和函数的声明序列组成,如果在某个函数中对名字a有非局部引用,那么a必须作为外部变量,声明在所有函数的外面,例如图6.1的数组a。函数外声明的作用域是该声明后的所有函数体,但是某个函数体中有该名字的重新声明的情况要除外。在图6.1中,readArray、partition和main中对a的非局部引用都是引用第一行声明的数组a。

由于没有过程嵌套,6.2节中局部名的栈式分配策略可以直接用于像C这样的静态作用域语言。声明在过程外面的所有变量都可以分配在静态区,它们的存储位置在编译时都可以确定。所以,过程体中的非局部引用可以直接使用静态确定的地址。任何其他变量必定局部于栈顶的活动记录,可以通过base_sp指针来访问。过程嵌套会使这种方法失败,因为对非局部名字的访问需要深入到栈中访问数据,这个问题在6.3.2节讨论。

对非局部名字进行静态分配的一个重要好处是,程序中声明的过程可以作为参数来传递,也可以作为结果来返回(C语言传递和返回的都是过程指针)。这是因为在静态作用域并且无嵌套过程的情况下,一个过程的任何非局部名字也是所有过程的非局部名字,该名字的静态地址可以被所有过程使用,而不用管这些过程是怎样被激活的。同样,如果过程作为结果返回,该被返回的过程中对非局部名字的引用,仍然是引用静态分配给这些名字的地址。

*6.3.2 有过程嵌套的静态作用域

在Pascal语言的过程p中若没有名字a的声明,则适用于p中a出现的a的声明在p的某个外围过程q中,并且从静态程序正文看,q中的这个a声明比任何其他a声明更靠近p中的这个a出现。

为说明问题,把图6.1快速排序程序用Pascal语言重写在图6.14中。在图6.14中,partition函数声明在quickSort过程的里面,并增加了exchange过程。修改后的快速排序程序中,过程声明的嵌套由下面的阶梯表示:

在图6.14中,第(15)行的a出现在函数partition中,最接近这个a的外嵌a声明在第(2)行,它属于构成整个程序的过程。最接近的嵌套规则也同样用于过程名。第(17)行中由函数partition调用的过程exchange对partition来说是非局部的,应用这个规则,首先检查exchange是否定义在quickSort中;因为不是,所以在主程序sort中寻找它。

在实现静态作用域时,需要过程**嵌套深度**概念。设主程序的嵌套深度为1,从一个过程进入一个被包围的过程时,嵌套深度加1。因此第(11)行的quickSort过程的嵌套深度是2,而第(13)行的partition过程的嵌套深度是3。对名字的每次出现,把它的声明所在过程的嵌套深度作为该名字的嵌套深度。在partition的第(15)行到(17)行的a,v和i的嵌套深度分别为1,2和3。

```
sort
    readArray
    exchange
    quickSort
        partition
(1)         program sort(input,output);
(2)             var a:array[0..10]of integer;
(3)             x::integer;
(4)             procedure readArray;
(5)                 var i:integer;
(6)                 begin…a…end{readArray};
(7)             procedure exchange(i,j:integer);
(8)                 begin
(9)                     x:=a[i];a[i]:=a[j];a[j]:=x
(10)                end{exchange};
(11)            procedure quickSort(m,n:integer);
(12)                var k,v:integer;
(13)                function partition(y,z:integer):integer;
(14)                    var i,j:integer;
(15)                    begin    …a…
(16)                             …v…
(17)                             …exchange(i,j);…
(18)                    end{partition};
(19)                begin…end{quickSort};
(20)            begin…end{sort}.
```

图6.14 有过程嵌套的Pascal程序

过程嵌套的静态作用域的直接实现是在每个活动记录中增加一个叫做访问链的指针。如果过程p直接嵌在过程q中,那么过程p活动记录的访问链直接指向最靠近的那个属于过程q的活动记录的访问链。

图6.14的程序运行时,运行栈的瞬像在图6.15给出。为了节约空间,图中仍然只给出过程名的第一个字母。sort活动的访问链为空(图中用nil表示),因为它不再有外围的过程。quick-Sort的每个活动的访问链都指向sort的活动记录。图6.15(c)中,partition(1,3)活动记录的访问链指向最靠近的那个quickSort活动记录的访问链,即quickSort(1,3)活动记录的访问链。

假定过程p的嵌套深度为n_p,它引用一个嵌套深度为n_a的变量a,$n_a \leq n_p$,则a的存储单元可以如下找到。

(1) 当控制在p中时,p的一个活动记录肯定在栈顶。首先从栈顶的活动记录开始,追踪访

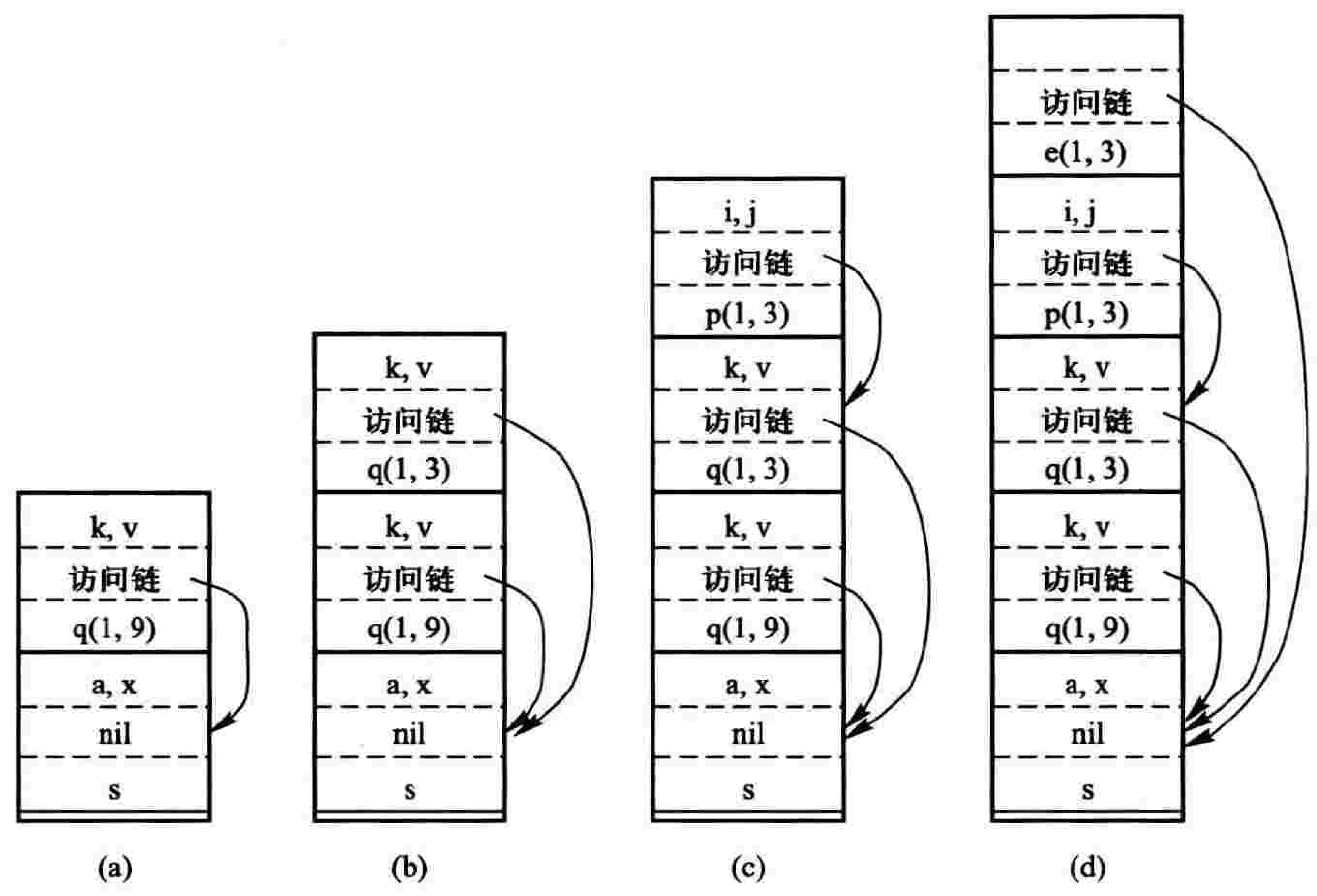


图 6.15　寻找非局部名字存储单元的访问链

问链 n_p-n_a次(n_p-n_a的值可以在编译时计算)。如果一个活动记录的访问链正好指向另一个活动记录的访问链,那么访问链的追踪用间接操作就可以完成。

(2) 追踪访问链 n_p-n_a次后,到达 a 的声明所在过程的活动记录。根据 6.1 节和 6.2 节的讨论,它的存储单元是在相对该活动记录中某个位置的固定偏移处。

因此,过程 p 对变量 a 访问时,a 的地址由下面的二元组表示:

(n_p-n_a,a 在活动记录中的偏移)

其中第一个分量给出追踪访问链的次数。

例如,在图 6.14 中的(15)和(16)行中,过程 partition 的嵌套深度为 3,它所引用的非局部变量 a 和 v 的嵌套深度分别为 1 和 2。包含这些非局部变量存储单元的活动记录可以从 partition 的活动记录分别追踪访问链 3-1=2 次和 3-2=1 次而找到。访问链是按照过程的静态嵌套关系建立的,因此它也称**静态链**。

建立访问链的代码是过程调用序列的一部分。假定嵌套深度为 n_p的过程 p 调用嵌套深度为 n_x的过程 x,建立被调用过程访问链的代码取决于被调用过程是否嵌在调用过程的里面。

(1) $n_p<n_x$的情况。这表明被调用过程 x 比 p 嵌得更深,而且 x 肯定就声明在 p 中,否则 p 不能访问 x。图 6.15(a)中 sort 调用 quickSort 和图 6.15(c)中 quickSort 调用 partition 都属于这种情况。此时,被调用过程的访问链必须指向栈中刚好在它下面的调用过程的活动记录的访问链。

(2) $n_p \geqslant n_x$的情况。根据作用域规则,p 和 x 的嵌套深度分别为 1,2,…,n_x-1 的外围过程肯定相同。图 6.15(b)中 quickSort 调用本身和图 6.15(d)中 partition 调用 exchange 都属于这种情

况。从调用过程追踪访问链 n_p-n_x+1 次,即到达了静态包围 x 和 p 的并且离它们最近的那个过程的最新活动记录。所到达的这个访问链就是被调用过程 x 的活动记录中的访问链应该指向的那个访问链。同样,n_p-n_x+1 的值可以在编译时计算。

当过程作为参数传递,尤其是被嵌套过程作为参数传递的情况,怎样在该过程被激活时建立它的访问链呢?以图6.16的 Pascal 程序为例说明之。在该程序的第(8)行,c 的过程体把 f 作为参数传递给 b。在 b 的体中,语句 writeln(h(2))激活 f,因为形参 h 代表的是 f,即 writeln 打印调用 f(2)的结果。可以看出,f 在 b 的体中被激活,但是从 b 的访问链难以建立 f 的访问链。

怎样解决这个问题呢?在过程作为参数传递时,必须把它的访问链和它一起传递。如图6.17所示,当过程 c 传递 f 时,它确定 f 的访问链,就好像 f 被调用一样,该链和 f 一起传递给 b。随后,当 f 在 b 中被激活时,该链用来建立 f 活动记录的访问链,而不按上面常规方法建立访问链。

```
(1)  program param(input,output);
(2)      procedure b(function h(n:integer):integer);
(3)          begin writeln(h(2))end{b};
(4)      procedure c;
(5)          var m:integer;
(6)          function f(n:integer):integer;
(7)              begin f:=m+n end{f};
(8)          begin m:=0;b(f)end{c};
(9)      begin
(10)         c
(11)     end.
```

图6.16 过程作为参数

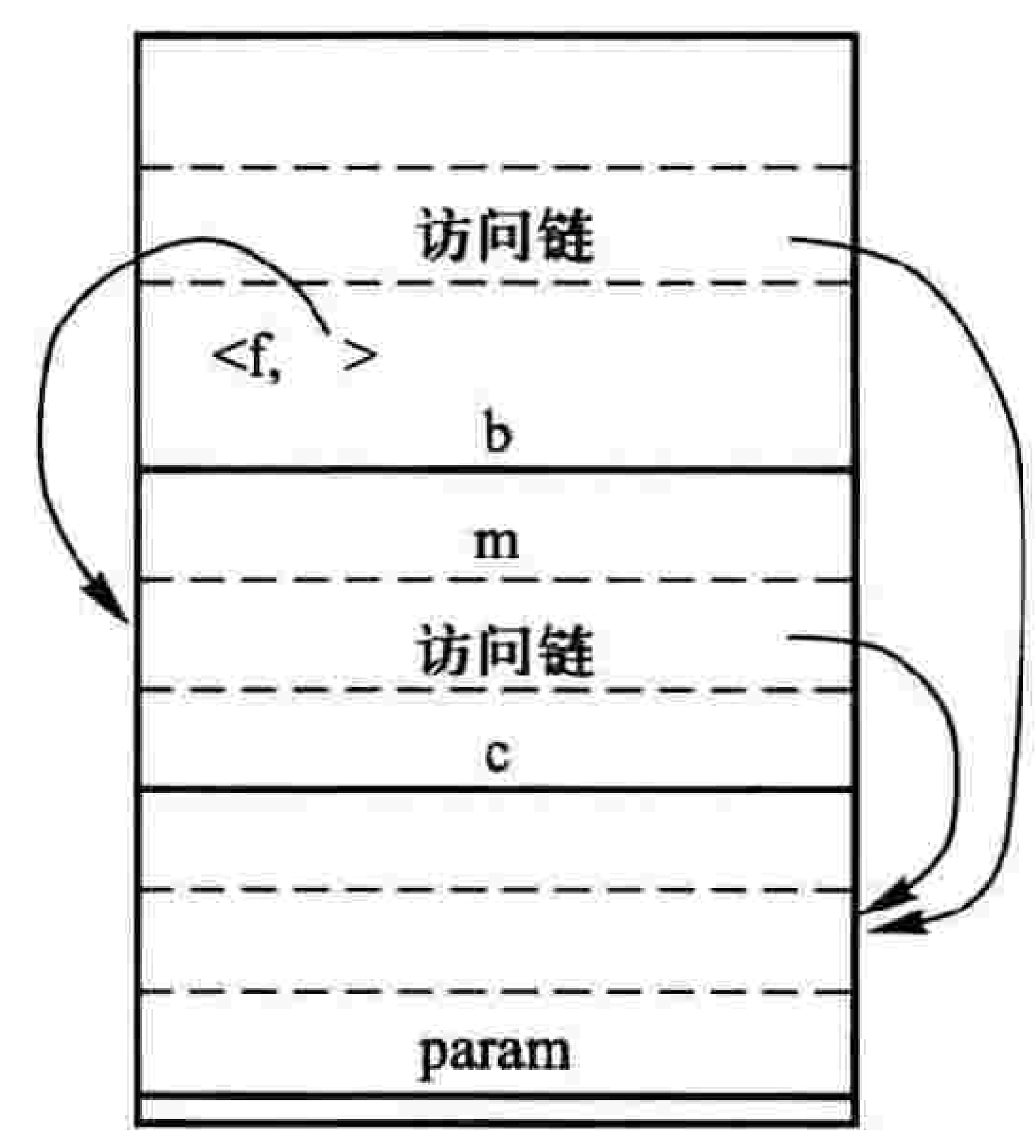


图6.17 实过程 f 带着它的访问链一起传递

对于 Pascal 语言来说，如果允许一个函数的返回值是函数，那么有可能出现作为返回值的函数执行时，需访问的非局部数据已经不复存在，习题 6.21 可以说明这一点。所以 Pascal 语言不允许函数类型作为函数的返回值类型。

前面已经提到，C 语言的函数声明不能嵌套，因此 C 的函数不论在什么情况下激活，在执行它的代码时要访问的数据分成两种情况：

(1) 非静态局部变量（包括参数），它们分配在运行栈顶的那个活动记录中；

(2) 外部变量（包括定义在其他源文件中的外部变量）和静态的局部变量，它们都分配在静态数据区。

因此，C 语言不会出现 Pascal 语言碰到的那种问题。

*6.3.3 动态作用域

在动态作用域下，被调用过程 q 的非局部名字 a 到存储单元的绑定，与 q 的调用过程 p 中对 a 的绑定是一致的，即 q 的非局部名字 a 和 p 中的 a 引用的是同样的存储单元。在 q 被调用时，新的绑定仅为 q 的局部名字建立，这些名字在 q 的活动记录中占用存储单元。也可以这么说，一个名字的声明在运行时实施它的影响，直至过程调用时在被调用过程中遇到该名字的一个新的声明为止；被调用过程停止时，该影响恢复。

图 6.18 的程序可用来说明动态作用域的概念。第(3)和(4)两行的过程 show 输出非局部量 r 的值。按照 Pascal 的静态作用域规则，非局部量 r 在第(2)行声明的作用域中，所以程序的输出是：

0.250　0.250

0.250　0.250

如果是动态作用域，则它的输出是：

0.250　0.125

0.250　0.125

当主程序在第(10)和(11)两行调用 show 时，写出 0.250，因为使用的是局部于主程序的 r。但是在第(7)行从 small 调用 show 时，输出 0.125，因为使用的是局部于 small 的变量。之所以在第(11)行重复第(10)行的调用，目的是想说明，第(10)行的 small 调用结束后，第(6)行 r 声明的影响结束，第(2)行 r 声明的影响恢复。所以第(11)行再次调用 show 时，又输出 0.250。

下面介绍两种实现动态作用域的方法。

(1) 深访问。概念上，如果访问链指向的活动记录和控制链指向的活动记录一样，那就实现了动态作用域。因此一种简单的实现是省略访问链，并用控制链搜索运行栈，寻找包含所需非局部名字的第一个活动记录。深访问的意思是，搜索可能要深入运行栈中，搜索的深度可能取决于程序的输入，编译时无法确定。

(2) 浅访问。为程序中每个名字在静态数据区分配空间，用于保存它的当前值。当过程 p

的新活动出现时,p 的局部名字 n 使用在静态数据区分配给 n 的存储单元。n 先前的值保存在 p 的活动记录中,当 p 的活动结束时再恢复。

```
(1)  program dynamic(input,output);
(2)      var r:real;
(3)      procedure show;
(4)          begin write(r:5:3)end;
(5)      procedure small;
(6)          var r:real;
(7)          begin r:=0.125;show end;
(8)      begin
(9)          r:=0.25;
(10)         show;small;writeln;
(11)         show;small;writeln
(12)     end.
```

图 6.18　用于说明作用域的一个程序

深访问对非局部名字需要较长的访问时间,但是它在活动的开始和结束处没有额外开销;浅访问则相反,它可以直接访问非局部名字,但在活动的开始和结束处需要花费时间来维护这些值。当函数作为参数传递和作为结果返回时,深访问可以较直截了当地实现。

6.4 参数传递

过程调用时,调用者和被调用者之间交换信息的办法通常是通过非局部名字和被调用过程的参数来实现。本节讨论形参和实参联系的几种一般方法,它们是值调用、引用调用和换名调用。了解语言(或编译器)的参数传递方法是很重要的,因为程序的结果依赖于所使用的方法。参数传递方法的区别在于传递实参的右值、左值还是实参本身的正文。

6.4.1 值调用

值调用是最简单的传递参数的方法。调用者计算实参,并把它的值(右值)传给被调用者。C 语言和 Java 语言使用值调用,值调用也是 C++和 Pascal 等很多语言参数传递方式的一种选择。本章到目前为止的所有程序都是按这种方式传递参数的。值调用可以如下实现。

(1) 把形参当作所在过程的局部名看待,形参的存储单元在该过程的活动记录中。

(2) 调用者计算实参,并把它的值放入形参的存储单元中。

值调用的显著特征是,对形参的任何操作不会影响调用者实参的值。

但是需要注意,在C语言中,指向一个变量的指针可以传递给被调用过程,导致该变量有可能在被调用过程中被修改。同样地,在C、C++和Java中,作为参数传递的数组名或对象名本质上传递给被调用过程的是关于自身的一个指针或引用。例如,如果a是调用者的一个数组名,它的值传给被调用者的形参x,那么x[i]=2这样的赋值会改变数组元素a[i]。

6.4.2 引用调用

引用调用(也称为**地址调用**)的参数传递方式是,调用者把实参存储单元的地址(即实参的左值)传给被调用者,被调用者对形参的任何访问就是对对应实参的访问。引用调用可以如下实现。

(1) 如果实参是有左值的名字或表达式,则把该左值放入形参的存储单元。如果实参是a+b或2这样没有左值的表达式,则把它的值计算出来后放入新的存储单元,然后传递这个单元的地址。

(2) 在被调用过程的目标代码中,任何对形参的访问都是通过传给该过程的地址来间接访问实参的。

引用调用的显著特征是,对形参的任何赋值都会影响调用者的实参。

Pascal的var参数是按引用调用方式传递的,C++语言的ref参数也使用引用调用,引用调用也是其他许多编程语言的一个选项。当形参是较大数据对象时,例如数组或对象,是否采取引用调用则是一个很重要的问题。其原因是,严格的值调用需要调用者把整个实参复写到属于对应形参的空间,当这个参数很大时复写的开销也很大。6.4.1节已经提到,Java和C++等语言的解决办法是,不是复写整个数据对象,而是复写它的一个引用。因此,从本质效果来说,Java对整型和实型等基本类型采用值调用,对其他类型采取引用调用。

6.4.3 换名调用

换名调用出自一种计算模型——λ演算,它用于早期的编程语言ALGOL 60。换名调用可以用ALGOL的复制规则来定义,具体如下。

(1) 把过程当作宏来对待,也就是在调用点,用被调用过程的体来替换调用者的调用,但是形参用对应的实参正文来代换。这种正文替换方式称为**宏展开**或**内联展开**。

(2) 被调用过程的局部名与调用过程的名字保持区别。可以认为在宏展开前,被调用过程的局部名字都系统地被重新命名成可区别的名字。

(3) 为保持实参的完整性,实参可以由括号包围。

例6.6 对于过程

```
procedure swap(var x,y:integer);
    var temp:integer;
    begin
```

```
        temp: =x;
        x: =y;
        y: =temp
    end
```

调用 swap(i,a[i])在换名调用下的实现好像它是

```
temp: =i;
i: =a[i];
a[i]: =temp
```

从这里可以看出换名调用和其他方式的重要区别。第 2 行引用的 a[i]和第 3 行被赋值的 a[i]可能是不同的数据单元,因为 i 的值在第 2 行可能被改变了。 □

换名调用比较复杂,因此是一种不再受欢迎的机制。它的实现也比较复杂。

虽然换名调用主要由于理论上的兴趣,但是概念上的内联展开暗示了可以缩短程序运行时间。建立过程的活动,包括活动记录的空间分配、机器状态的保存、链的建立和控制的转移等,都需要一定的代价。如果过程体较小,那么过程调用序列的代码可能会超过过程体的代码。此时把过程体的代码内联展开到调用者的代码中,即便程序的代码会稍长一些,效率也会提高。出于这样的考虑,C++和 Java 在 C 的基础上增加了内联函数。

*6.5 堆 管 理

当过程终止时,一般而言,其局部变量不再可访问。然而,许多语言允许创建对象或其他数据,它们的生存期没有被约束在创建它们的过程活动的生存期之内。例如,C++和 Java 允许程序员用 new 创建对象,这些对象,或者说是指向它们的指针,可以从一个过程传递到另一个过程。因此,创建它们的过程终止后,它们可能继续长期存在。这样的对象存放在堆上,堆就是用来存放这种生存期不确定或一直到程序显式释放的数据。

本节讨论在堆上分配和回收空间的内存管理器,它是应用程序和操作系统之间的一个接口。像 C 和 C++这样由程序员显式释放存储块(例如通过 free 或 delete 调用)的语言,实现内存回收也是内存管理器的责任。

无用单元收集是一个在堆上寻找空间的过程,它寻找程序不再使用因而可以重新分配给其他数据项的空间。对 Java 语言来说,存储块的释放靠无用单元收集器来完成,因此无用单元收集器是 Java 运行系统中完成内存管理的重要子系统,它将在 11.3 节介绍。

6.5.1 内存管理器

内存管理器所掌握的基本信息是堆中的空闲空间,它执行的基本函数是以下两个。

(1) 分配函数。当程序请求空间以存放数据时,内存管理器按所需大小分配一块连续的堆块。只要有可能,它总是用堆中的空闲空间来满足分配请求;如果堆中没有满足大小要求的空闲块,它就从操作系统申请虚拟内存的连续字节以增加堆空间。如果空间被耗尽,内存管理器将此信息反馈给用户程序。

(2) 回收函数。内存管理器将回收的空间返回到空闲空间池,使得它可重新使用以满足其他分配请求。内存管理器的典型做法是不把内存返回给操作系统,甚至在程序堆的使用情况下降时也是如此。

在下面两种情况下,内存管理器实现非常简单。

(1) 所有分配请求需要同样大小的存储块。LISP 语言属于这种情况。纯 LISP 仅使用一种数据元素,它是含两个指针的存储单元,所有数据结构都是在这样的数据元素上构建。

(2) 存储的回收是可以预见的,例如先分配的先回收。在某些场合下这种情况会出现,最常见的就是可以分配在运行栈上的数据。

然而对大多数编程语言来说,上面两种情况都不满足。也就是说,需要分配不同大小的数据对象,而且没有好的方法来预测它们的生存期。因此,内存管理器必须应对任何次序、任何大小的分配和释放请求,小到 1 个字节,大到整个程序地址空间。

人们希望内存管理器具有下列性质。

(1) 空间有效性。内存管理器应该极小化程序需要的堆空间总量。这样做允许较大的程序运行在固定虚拟地址空间上。空间有效性可以通过极小化碎片来获得。

(2) 程序有效性。内存管理器应该很好地利用内存子系统,使得程序能运行得快一些。在 6.5.2 节将看到,一条指令的执行时间将在相当大的范围内变化,它取决于所涉及对象在内存的什么地方。幸好,程序通常展示出较好的局部性,即程序访问内存是以一种非随机的聚集方式(6.5.3 节讨论这种现象)实现的。通过关注数据对象在内存中的布局,内存管理器可以较好地利用空间,有希望使程序运行得较快。

(3) 低开销。因为内存分配和释放操作在许多程序中频繁出现,因此这些操作要尽可能有效,使得分配和回收操作所花时间在整个程序执行时间中所占比例尽量小。注意,分配的代价被小数据对象的分配请求所支配,而管理较大数据对象的开销不是很重要,因为它通常可以从这个数据对象上的大量计算中得到分摊。

由于数据结构的教材上通常都介绍堆空间的分配和回收算法,包括怎样减少碎片,因此本书略去这些算法的介绍。

6.5.2 计算机内存分层

了解内存的行为表现是做出较好的内存管理设计的一个前提。现代计算机都设计成程序员不用关心内存子系统的细节就可以写出正确的程序,但是,一个程序的效率不仅取决于被执行的指令数,还取决于执行每条指令需要多长时间。不同指令的执行时间的差别非常可观,因为访问

内存不同部分所需时间在纳秒级到微秒级之间变化。数据密集型程序可以从恰当利用内存子系统中显著获益。在6.5.3节将看到,它们通常借助一种叫做“局部性”现象实现,它是典型程序的一种非随机行为。

内存访问时间差别大的原因是硬件技术的基本局限。可以构造小容量高速存储器,也可以构造大容量低速存储器,但是构造不了大容量的高速存储器。例如,现在不可能构造像高性能处理器运行一样快的纳秒访问时间的千兆字节存储器。因此,现代计算机都把存储器进行分层安排,图6.19是一种内存分层(memory hierarchy),它由一系列的存储元件组成,其中最小、最快的最“靠近”处理器,较大、较慢的则较远一些。

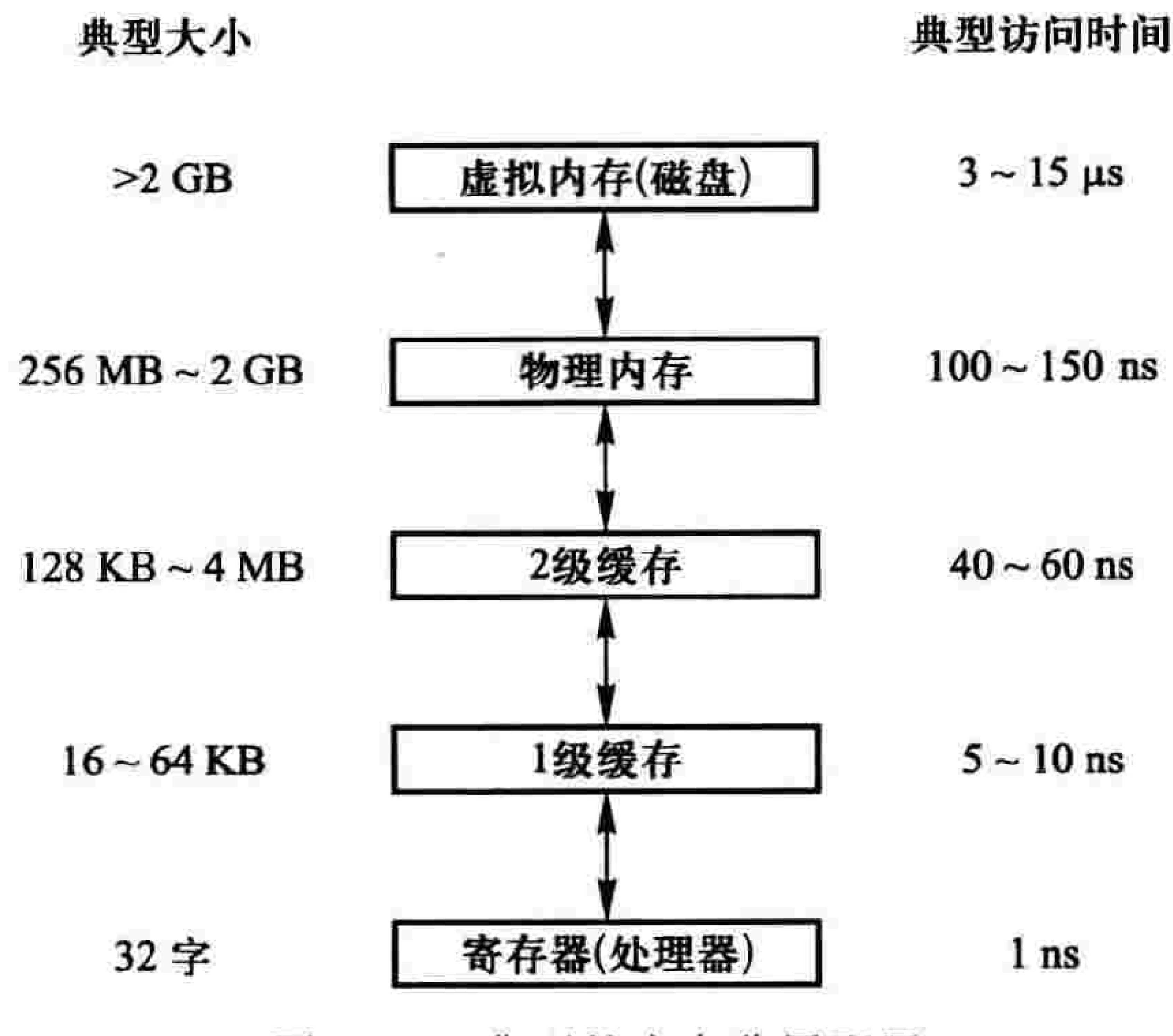


图6.19 典型的内存分层配置

典型情况是,一个处理器有少数寄存器,它们的内容由软件控制。往上一层是一级或多级缓存,它们通常由静态RAM构成,大小在几千字节到几兆字节。再往上一层是物理内存(主存),由几百兆或几千兆动态RAM构成。物理内存由虚拟内存支持,虚拟内存由千兆字节的磁盘实现。对于一个内存访问,机器从最底层开始,逐层查找,直到定位数据为止。

寄存器个数很少,为了使之得到充分利用,它们由编译器生成的代码来管理。内存分层中所有其他层次都被自动管理。按此方式,不仅编程简化,而且同一个程序在不同内存配置的机器上同样能有效地运行。缓存由硬件管理,以跟上相对快的RAM访问时间。磁盘相对较慢,因此虚拟内存在一种叫做翻译后备缓冲器(translation lookaside buffer)的硬件结构帮助下,由操作系统来管理。

数据以块为单位在相邻层次之间进行传送。为分摊访问代价,较大的块用在内存分层的较慢层次之间。在主存和缓存之间,数据传送的块称为缓存行(cache line),长度在32 ~ 256 B之间。在虚拟内存和主存之间,数据传送的块称为页,大小在4 KB ~ 64 KB之间。

6.5.3 程序局部性

大多数程序表现出相当高的局部性,即它们大部分时间在执行一小部分代码,并且仅涉及一小部分数据。如果程序访问的内存单元在很短的时间内可能再次被程序访问,则称该程序具有**时间局部性**。如果毗邻被访问单元的内存单元在很短的时间内会再次被访问,则称该程序具有**空间局部性**。

传统的说法是,程序 90% 的时间消耗在执行 10% 的代码上,其原因如下。

(1) 程序经常包含许多决不会执行的指令。由组件和库构建的程序经常仅使用所提供功能的一小部分。还有,随着需求变化和程序演化,遗留系统(legacy system)通常包含许多不再使用的指令。

(2) 在程序的典型执行中,通常只有一小部分代码可能被真正执行。例如,处理非法输入和例外情况的指令虽然对程序的正确性来说是至关重要的,但是它们很少被执行。

(3) 典型程序的大部分时间消耗在程序中最内层循环和深度递归的执行上。

现代计算机的内存分层可以很好地被程序局部性利用。把最经常使用的指令和数据放在速度快但空间小的层次上,把其余的放在大而慢的层次上,这样可以大大降低程序的平均内存访问时间。

一般来说,从代码本身很难看出哪部分代码会被频繁使用,尤其是对某个特定的输入。即使知道了哪些指令会被频繁执行,最快的缓存也可能没有大到足以把它们同时纳入其中。于是,必须动态调整最快缓存的内容,让它保存在不久的将来会频繁使用的指令。

怎样最优化利用内存分层?把最近使用的指令保存在缓存中往往是一种较好的策略,换句话说,过去使用情况一般是内存未来使用的一个有益预测。当一条新的指令被执行,很可能下一条指令也将被执行,这个现象是空间局部性的一个例子。改进指令空间局部性的一种有效技术是,编译器尽可能把可能毗邻执行的基本块(基本块是一个顺序执行的指令序列,见第 8 章)放在同一页中,甚至放在同一缓存行上。属于同一个循环或同一个函数的代码也很可能一起执行。

改变数据布局或计算次序也可以改进程序数据访问的时间和空间局部性。例如,对于需重复访问大批数据的计算来说,每次遍历数据完成一小部分计算的方式则效率不高。较好的做法是每次从较慢层取部分数据到较快层(如从磁盘到主存),在这些数据驻留较快层期间完成对它们的所有计算。同样的思想可用于在主存、缓存和寄存器中的数据。

6.5.4 手工回收请求

C 和 C++语言要求程序员在程序中显式释放堆块来达到回收堆块的目的,称之为**手工回收**;本质上,这就是让程序员介入内存管理。理想的情况是,任何不再被访问的存储都应该被释放,反过来,任何有可能被访问的存储一定不能释放。不幸的是,要想让程序维持这两个性质是非常

困难的。

手工内存管理很容易出错。常见错误有两种形式，一种是没有释放已经引用不到的堆块，称之为**内存泄漏**错误；另一种是引用已经被释放的堆块，称之为悬空引用错误。后者也可能发生在栈上，6.2.5 节已经介绍过。注意，虽然内存泄漏可能会因内存使用的不断增加而导致程序运行速度降低，但是只要内存没有用尽，它就不会影响程序的正确性。许多程序容忍内存泄漏，尤其是当泄漏速度很慢时。然而，对长时间运行的程序，特别是不间断运行的操作系统或服务器代码来说，不出现内存泄漏是至关重要的。

自动无用单元收集通过回收所有无用单元来避免内存泄漏。但即使是这样，程序仍然可能占用了一些不再使用的内存。例如，一个对象在程序中不再使用，虽然目前还存在指向它的引用；这导致该对象无须保留却又不能自动回收。在这种情况下，程序员应该移开对该对象的引用，以便它能被自动回收。

过分热心地释放数据对象会导致比内存泄漏更严重的悬空引用问题，因为悬空引用容易导致不会被捕获的错误。因此，程序员在不能确保不再使用一个数据对象前，更倾向于不释放该对象。

一个相关的编程错误是访问非法地址。常见的例子有对空指针（null pointer）进行脱引用（dereferencing）操作和越界访问数组元素。这样的错误宜及早发现，因为这些安全违例有可能导致黑客控制程序和机器。一种防止这类错误的做法是让编译器插入检查代码以保证每次访问都在界内。编译器的优化器可以发现其中没有必要的检查并把它们删除，因为优化器可以推导出一些访问一定落在界内。

习 题 6

6.1 使用 Pascal 的作用域规则，确定下面程序中用于名字 a 和 b 的每个出现的声明。该程序的输出是整数 1,2,3,4。

```
program a(input,output);
    procedure b(u,v,x,y:integer);
        var a:record  a,b:integer end;
            b:record  b,a:integer end;
        begin
            with a do begin a:=u;b:=v end;
            with b do begin a:=x;b:=y end;
            writeln(a.a,a.b,b.a,b.b)
        end;
    begin
        b(1,2,3,4)
```

end.

6.2 一个C程序的三个文件 head.h、file1.c 和 file2.c 的内容分别如下。

head.h:	file1.c:	file2.c:
short int a=10;	#include "head.h"	#include "head.h"
	main(){	
	}	

在 x86/Linux 系统上，使用某版本的 GCC 编译器，编译命令如下：

```
cc file1.c file2.c
```

编译结果报错的主要信息如下：

```
multiple definition of 'a'
```

试分析为什么会报这样的错误。

*6.3 考虑下面的C语言程序：

```
main(){
    char *cp1, *cp2;
    cp1 = "12345";
    cp2 = "abcdefghij";
    strcpy(cp1,cp2);
    printf("cp1=%s\ncp2=%s\n",cp1,cp2);
}
```

(a) 该程序经早先某些C编译器的编译，其目标程序的运行结果是：

```
cp1=abcdefghij
cp2=ghij
```

试分析，为什么 cp2 所指向的串被修改了？

(b) 在 x86/Linux 系统上经任何近期版本的 GCC 编译器编译后，该程序运行时，操作系统报告段错误(segmentation fault)并终止运行，请分析原因。

6.4 一个C语言程序如下：

```
typedef struct    _a{
    char          c1;
    long          i;
    char          c2;
    double        f;
}a;
typedef struct    _b{
    char          c1;
    char          c2;
```

```
        long        i;
        double      f;
    } b;
    main() {
        printf("Size of double,long,char=%d,%d,%d\n",
                              sizeof(double),sizeof(long),sizeof(char));
        printf("Size of a,b=%d,%d\n",sizeof(a),sizeof(b));
    }
```

该程序在早先的 SPARC/Solaris 系统上的运行结果如下：

```
Size of double,long,char=8,4,1
Size of a,b=24,16
```

在 x86/Linux 系统上经编译器 GCC:(GNU) egcs-2.91.6619990314/Linux(egcs-1.1.2 release)编译后，运行时第 2 行输出不一样，是

```
Size of a,b=20,16
```

结构体类型 a 和 b 的域都一样，仅次序不同，为什么它们需要的存储空间不一样？为什么在不同的机器上情况又不一样？

*6.5 一个 C 语言程序如下：

```
typedef struct_a{
    short  i;
    short  j;
    short  k;
}a;
typedef struct_b{
    long   i;
    short  k;
}b;
main(){
    printf("Size of short, long,a and b=%d,%d,%d,%d\n",
        sizeof(short),sizeof(long),sizeof(a),sizeof(b));
}
```

该程序在 Ubuntu 12.04.2 LTS(GNU/Linux 3.2.0-42-generic x86_64)系统上，经过编译器 GCC:(Ubuntu/Linaro 4.6.3-1ubuntu5)4.6.3 编译后，运行结果如下：

```
Size of short,long,a and b=2,8,6,16
```

已知 short 类型和 long 类型分别对齐到 2 的倍数和 8 的倍数。试问，为什么类型 b 的 size 会等于 16？

6.6 下面是C语言两个函数f和g的概略(它们不再有其他的局部变量):

```
int f(int x){int i;…return i +1;…}
int g(int y){int j;…f(j+1);…}
```

请按照图6.11的形式,画出函数g调用f,f的函数体正在执行时,活动记录栈的内容及相关信息,并按图6.10左侧箭头方式画出控制链。假定函数返回值是通过寄存器传递的。

*6.7 下面给出C语言程序及其在x86/Linux系统上的编译结果(编译器版本见汇编代码最后一行)。根据所生成的汇编程序来解释程序中四个变量的存储分配、作用域、生存期和置初值方式等方面的区别。

```
static long aa=10;
short bb=20;
func(){
    static long cc=30;
    short dd=40;
}
```

该C语言程序生成的汇编代码:

```
        .file"static.c"
        .version"01.01"
gcc2_compiled.:
.data
        .align 4
        .type aa,@object
        .size aa,4
aa:
        .long 10
.globl bb
        .align 2
        .type bb,@object
        .size bb,2
bb:
        .value 20
        .align 4
        .type cc.2,@object
        .size cc.2,4
cc.2:
        .long 30
```

```
.text
    .align 4
.globl func
    .type func,@function
func:
    pushl %ebp
    movl %esp,%ebp
    subl $4,%esp
    movw $40,-2(%ebp)
.L1:
    leave
    ret
.Lfe1:
    .size func,.Lfe1-func
    .ident "GCC:(GNU)egcs-2.91.66 19990314/Linux(egcs-1.1.2 release)"
```

6.8 假定使用:(a)值调用、(b)引用调用、(c)换名调用,下面的程序打印的结果是什么?

```
program main(input,output);
var a,b:integer;
procedure p(x,y,z:integer);
    begin
        y:=y+1;
        z:=z+x;
    end;
begin
    a:=2;
    b:=3;
    p(a+b,a,a);
    print a;
end.
```

6.9 C语言函数f的定义如下:

```
int f(int x,int *py,int **ppz){
    **ppz+=1;*py+=2;x+=3;return x+*py+**ppz;
}
```

变量a是指向b的指针,变量b是指向c的指针,c是整型变量并且当前值是4。那么执行f(c,b,a)的返回值是多少?

6.10 一个C语言程序如下：

```
func(i1,i2,i3)long i1,i2,i3;{
    long j1,j2,j3;
    printf("Addresses of i1,i2,i3=%o,%o,%o\n",&i1,&i2,&i3);
    printf("Addresses of j1,j2,j3=%o,%o,%o\n",&j1,&j2,&j3);
}
main(){
    long i1,i2,i3;
    func(i1,i2,i3);
}
```

该程序在x86/Linux系统上，经某编译器编译后的运行结果如下：

```
Addresses of i1,i2,i3=27777775460,27777775464,27777775470
Addresses of j1,j2,j3=27777775444,27777775440,27777775434
```

从上面的结果可以看出，func函数的三个形式参数的地址依次升高，而三个局部变量的地址依次降低。试说明为什么会有这个区别。注意，输出的数据是八进制的。

*6.11 下面的C语言程序中，函数printf的调用仅含格式控制字符串一个参数，该程序在x86/Linux系统上，经某编译器编译后，运行时输出三个整数。试从运行环境和printf的实现来分析，为什么此程序会有三个整数输出。

```
main(){
    printf("%d,%d,%d\n");
}
```

6.12 下面是一个C语言程序：

```
long f1(i)long i;{
    return(i*10);
}
long f2(long i){
    return(i*10);
}
main(){
    printf("f1=%d,f2=%d\n",f1(10.0),f2(10.0));
}
```

其中函数f1和f2仅形式参数的描述方式不一样。该程序在x86/Linux系统上，用编译器GCC：(GNU)egcs-2.91.66 19990314/Linux(egcs-1.1.2 release)编译后，运行结果如下：

```
f1=0,f2=100
```

请解释为什么用同样的实在参数调用这两个函数的结果不一样。

6.13　一个 C 语言程序如下：

```
int fact(i)int i;{
    if(i==0)
        return 1;
    else
        return i * fact(i-1);
}
main(){
    printf("%d\n",fact(5));
    printf("%d\n",fact(5,10,15));
    printf("%d\n",fact(5.0));
    printf("%d\n",fact());
}
```

该程序在 x86/Linux 系统上，用编译器 GCC：(GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release) 编译后，运行结果如下：

```
120
120
1
Segmentation fault(core dumped)
```

请解释下面问题：

(a) 第二个 fact 调用：结果为什么没有受参数过多的影响？

(b) 第三个 fact 调用：为什么用浮点数 5.0 作为参数时结果变成 1？

(c) 第四个 fact 调用：为什么没有提供参数时会出现 Segmentation fault？

6.14　一个 C 文件 array.c 仅有下面两行代码：

```
char a[][4]={"123","456"};
char *p[]  ={"123","456"};
```

在 x86/Linux 系统上编译生成的汇编代码列在下面(编译器版本见汇编代码最后一行)，从中可以看出，对数组 a 和指针 p 的存储分配是不同的。试依据这里的存储分配，为置了初值后的数组 a 和指针 p 写出类型表达式。

```
    .file "array.c"
.globl a
    .data
    .type a,@object
    .size a,8
a:
```

```
        .string "123"
        .string "456"
        .section .rodata
.LC0:
        .string "123"
.LC1:
        .string "456"
.globl p
        .data
        .align 4
        .type p,@object
        .size p,8
p:
        .long .LC0
        .long.LC1
        .section        .note.GNU-stack,"",@progbits
        .ident  "GCC:(GNU)3.3.5(Debian 1:3.3.5-13)"
```

*6.15 下面给出一个 C 语言程序及其在早先的 SPARC/SunOS 系统上经某编译器编译后的运行结果。从运行结果看，函数 func 中 4 个局部变量 i1,j1,f1,e1 的地址间隔和它们类型的大小是一致的，而四个形式参数 i,j,f,e 的地址间隔和它们类型的大小不一致，试分析不一致的原因。注意，输出的数据是八进制的。

```
func(i,j,f,e) short i,j;float f,e;{
    short i1,j1;float f1,e1;
    printf("Address of i,j,f,e=%o,%o,%o,%o\n",&i,&j,&f,&e);
    printf("Address of i1,j1,f1,e1=%o,%o,%o,%o\n",&i1,&j1,&f1,&e1);
    printf("Sizes of short,int,long,float,double=%d,%d,%d,%d,%d\n",
        sizeof(short),sizeof(int),sizeof(long),sizeof(float),sizeof(double));
}
main(){
    short i,j;float f,e;
    func(i,j,f,e);
}
```

运行结果是：

```
Address of i,j,f,e=35777772536,35777772542,35777772544,35777772554
Address of i1,j1,f1,e1=35777772426,35777772424,35777772420,25777772414
```

Sizes of short, int, long, float, double = 2, 4, 4, 4, 8

*6.16 一个C语言的函数

```
func(c,l) char c;long l;{
    func(c,l);
}
```

在x86/Linux系统上编译生成的汇编代码如下(编译器版本见汇编代码最后一行):

```
        .file"parameter.c"
        .version"01.01"
gcc2_compiled.:
.text
        .align 4
.globl func
        .type func,@function
func:
        pushl %ebp              //将老的基地址指针压栈
        movl %esp,%ebp          //将当前栈顶指针作为基地址指针
        subl $4,%esp            //分配空间
        movl 8(%ebp),%eax
        movb %al,-1(%ebp)
        movl 12(%ebp),%eax
        pushl %eax
        movsbl-1(%ebp),%eax
        pushl %eax
        call func
        addl $8,%esp
.L1:
        leave                   //和下一条指令一起完成恢复老的基地址指针,将栈顶
        ret                     //指针恢复到调用前参数压栈后的位置,并返回调用者
.Lfe1:
        .size func,.Lfe1-func
        .ident "GCC:(GNU)egcs-2.91.66 19990314/Linux(egcs-1.1.2 release)"
```

请说明字符型参数和长整型参数在参数传递和存储分配方面有什么区别(小于长整型size的整型参数的处理方式和字符型参数的处理方式是一样的)。

6.17 C程序设计的教材上说,可以用两种形式表示字符串:其一是用字符数组存放一个字符串,另一种是用字符指针指向一个字符串。教材上同时介绍了这两种形式的很多共同点和不

同点,但是有一种可能的区别没有介绍。下面是一个包含这两种形式的 C 程序:

```
char c1[]="good!";
char *c2="good!";
main(){
    c1[0]='G';
    printf("c1=%s\n",c1);
    c2[0]='G';
    printf("c2=%s\n",c2);
}
```

该程序在 Ubuntu 12.04.2 LTS(GNU/Linux 3.2.0-42-generic x86_64)系统上,经过编译器 GCC:(Ubuntu/Linaro 4.6.3-1ubuntu5)4.6.3 编译后,运行时的信息如下:

```
c1=Good!
Segmentation fault(core dumped)
```

请问,出现 Segmentation fault 的原因是什么?

*6.18 下面是一个 C 语言程序:

```
main(){
    long i;
    long a[0][4];
    long j;
    i=4;j=8;
    printf("%d,%d\n",sizeof(a),a[0][0]);
}
```

虽然出现 long a[0][4]这样的声明,但在 x86/Linux 系统上,用编译器 GCC:(GNU)egcs-2.91.66 19990314/Linux(egcs-1.1.2 release)编译时,该程序能够通过编译并生成目标代码。请回答下面两个问题:

(a) sizeof(a)的值是多少,请说明理由。

(b) a[0][0]的值是多少,请说明理由。

6.19 从例 6.4 可以看到,C 程序执行时只用到了控制链,不需要使用访问链。为什么 Pascal 程序执行时需要使用访问链,而 C 程序不需要。

6.20 下面是求阶乘的 Pascal 程序。画出程序第三次进入函数 factor 时的活动记录栈和静态链。

```
program fact(input,output);
    var f,n:integer;
    function factor(n:integer):integer;
        begin
```

```
            if  n=0  then  factor:=1
            else  factor:=n * factor(n-1)
        end;
    begin  n:=5;f:=factor(n);write(f)
    end.
```

*6.21 在下面假想的程序中，第(11)行语句 f:=a 调用函数 a,a 传递函数 addm 作为返回值。

(a) 画出该程序执行的活动树。

(b) 假定非局部名字使用静态作用域，为什么该程序在栈式分配情况下不能正确工作？

(c) 在堆分配策略下，该程序的输出是什么？

```
(1)     program  ret(input,output);
(2)         var  f:function  (integer):integer;
(3)         function  a:function(integer):integer;
(4)             var  m:integer;
(5)             function  addm(n:integer):integer;
(6)                 begin  return  m+n  end;
(7)             begin  m:=0;return  addm  end;
(8)         procedure  b(g:function(integer):integer);
(9)             begin  writeln(g(2))end;
(10)        begin
(11)            f:=a;b(f)
(12)        end.
```

*6.22 为什么 C 语言允许函数类型(的指针)作为函数的返回值类型，而 Pascal 语言却不允许？

6.23 一个 C 语言程序如下：

```
int  n;
int  f(g)int  g();{
    int  m;
    m=n;
    if(m==0)return  1;
    else{
        n=n-1;return  m * g(g);
    }
}
main(){
```

```
        n=5;printf("%d factorial is %d\n",n,f(f));
    }
```

该程序在 Ubuntu 12.04.2 LTS(GNU/Linux 3.2.0-42-generic x86_64)系统上,经过编译器 GCC:(Ubuntu/Linaro 4.6.3-1ubuntu5)4.6.3 编译后,运行结果不是所期望的

    5 factorial is 120

而是

    0 factorial is 120

试说明原因。

*6.24 下面程序在早先的 SPARC/SunOS 系统上运行时陷入死循环,试说明原因。如果将第 5 行的 long *p 改成 short *p,并且将第 15 行 long k 改成 shortk 后,loop 中的循环体执行一次便停止了。试说明原因。

```
main(){
    addr();
    loop();
}
long *p;
loop(){
    long i,j;
    j=0;
    for(i=0;i<10;i++){
        (*p)--;
        j++;
    }
}
addr(){
    long k;
    k=0;
    p=&k;
}
```

*6.25 一个 C 语言程序如下:

```
main(){
    func();
    printf("Return from func\n");
}
func(){
```

```
    char s[4];
    strcpy(s,"12345678");
    printf("%s\n",s);
}
```

在 Ubuntu 12.04.2 LTS(GNU/Linux 3.2.0-42-generic x86_64)系统上,该程序经过编译器 GCC:(Ubuntu/Linaro 4.6.3-1ubuntu5)4.6.3 编译后,运行结果如下:

```
12345678
Return from func
```

若将语句 strcpy(s,"12345678")改成 strcpy(s,"123456789"),目标程序 a.out 的运行结果及运行失败的报告如下:

```
123456789
*** stack smashing detected ***:a.out terminated
=======Backtrace:=========
注:Backtrace 的具体内容略去
=======Memery map:=======
注:Memery map 的具体内容略去
Aborted(core dumped)
```

试分析为什么会出现这样的运行情况。

*6.26 通常,当函数调用的返回值是简单类型时,用寄存器传递函数值。当返回值是结构体类型时需要采用别的方式。下面是一个 C 语言文件和它在 x86/Linux 系统上编译(编译器版本见汇编代码最后一行)生成的汇编代码。(备注,该汇编码略经修改,以便于阅读。该修改没有影响结果。)

(a) 请分析这些代码,总结出函数返回值是结构体类型时,返回值的传递方式。

(b) 若 m 函数的语句 s=f(10)改成 s.i=f(10).i+f(20).i+f(30).i,你认为 m 函数的局部存储分配应该怎样修改,以适用该语句的计算。

源文件 return.c 的内容如下:

```
typedef struct{long i;}S;
S f(k)long k;{
    S s;
    s.i=k;
    return s;
}
m(){
    S s;
    s.i=20;
```

```
    s=f(10);
}
```

汇编文件 return. s 的内容如下：

```
    .file"return.c"
    .text
.globl f
    .typef,@function
f:
    pushl   %ebp
    movl    %esp,%ebp
    subl    $4,%esp
    movl    8(%ebp),%eax
    movl    12(%ebp),%edx
    movl    %edx,-4(%ebp)
    movl    -4(%ebp),%edx
    movl    %edx,(%eax)
    leave
    ret
    .sizef,.-f
.globl m
    .typem,@function
m:
    pushl   %ebp
    movl    %esp,%ebp
    subl    $4,%esp
    movl    $20,-4(%ebp)
    leal    -4(%ebp),%eax
    pushl   $10
    pushl   %eax
    call f
    addl    $8,%esp
    leave
    ret
    .sizem,.-m
    .section.note.GNU-stack,"",@progbits
```

```
.ident "GCC:(GNU)3.3.5(Debian 1:3.3.5-13)"
```

*6.27 一个C语言的文件如下:

```
func(long i,long j,long k){
        k=(i+j)-(i-j-f(k));
}
```

经x86/Linux系统上编译(编译器版本见汇编代码最后一样)得到的汇编代码分两列在下面给出:

```
        .file   "call.c"
        .text
.globl func
        .type   func,@function
func:
        pushl   %ebp
        movl    $esp,%ebp
        pushl   %esi
        pushl   %ebx
        subl    $4,%esp
        movl    12(%ebp),%eax
        movl    8(%ebp),%esi
        addl    %eax,%esi
        movl    12(%ebp),%edx
        movl    8(%ebp),%eax
        movl    %eax,%ebx
        subl    %edx,%ebx
movl    16(%ebp),%eax
movl    %eax,(%esp)
call    f
subl    %eax,%ebx
movl    %ebx,%eax
subl    %eax,%esi
mov     %esi,%eax
movl    %eax,16(%ebp)
addl    $4,%esp
popl    %ebx
popl    %esi
popl    %ebp
ret
.size func,.-func
.section    .note.GNU-stack,"",@progbits
.ident  "GCC:(GNU) 3.4.6"
```

请根据上述汇编代码进行总结:为适应函数调用引起的跨函数执行,该编译器在寄存器的值的保护方面有些什么约定?

6.28 下面是一个以函数作为参数的C语言程序:

```
int f(int g()){return g(g);}
main(){f(f);}
```

(a) 对于函数类型的形式参数,调用时的参数传递传什么?

(b) 该程序执行时,系统报告 Segmentation fault,请回答是什么原因。

6.29 有人认为,下面 C 程序中结构体类型 record 的定义方式可用来动态生成其中 a 数组的大小不一样的结构体,以适应某些编程场合的需要。你认为这样的程序能够通过 C 编译器的类型检查吗?请说明理由。

```
#include<malloc.h>
typedef struct{double r;int n;float a[];}record;
main(){
    record *p;
    p=malloc(sizeof(record)+sizeof(float)*5);
    p->n=5;p->a[4]=100.0;…
}
```

6.30 (a) Java 语言的编译器通常把数组分配在堆上。Java 数组一般不能静态确定大小应该不是将它分配在堆上的原因,因为图 6.12 给出了一种将不能静态确定大小的数组动态地分配在活动记录栈上的方法。Java 数组一般不能分配在活动记录栈上的原因是什么?

(b) 将数组分配在活动记录栈上和分配在堆上给程序运行带来什么区别?

*6.31 Java 语言的实现通常把对象和数组都分配在堆上,把指向它们的指针分配在栈上,依靠运行时的垃圾收集器来回收堆上那些从栈不可达的空间(垃圾)。这种方式提高了语言的安全性,但是增加了运行开销。编译时能否采用一些技术,以降低垃圾收集所占运行开销?概述你的方案。

第7章

中间代码生成

第1章已经介绍,编译器的前端把源程序翻译成中间表示,后端从中间表示产生目标代码,与目标语言有关的细节尽可能限制在后端。使用独立于机器的中间表示的好处有如下两点。

(1) 再目标(retargeting,指生成另一种机器的目标代码)比较容易。把针对新机器的后端与现成的前端组合起来,就可以得到另一种机器的编译器。

(2) 独立于机器的代码优化器可以作用于这种中间表示。第9章将介绍这种代码优化。因此,虽然可以把源程序直接翻译并生成目标代码,但编译器一般都采用中间表示形式。

本章将用第4章的语法制导定义方法来描述编程语言的构造怎样被翻译成独立于机器的中间表示。为简单起见,假定对源程序的分析和静态检查已经完成,如图7.1表示的那样。本章大多数语法制导定义可以用第4章的技术在自下而上或自上而下的分析期间实现,所以,如果愿意的话,中间代码生成可以在分析阶段完成。

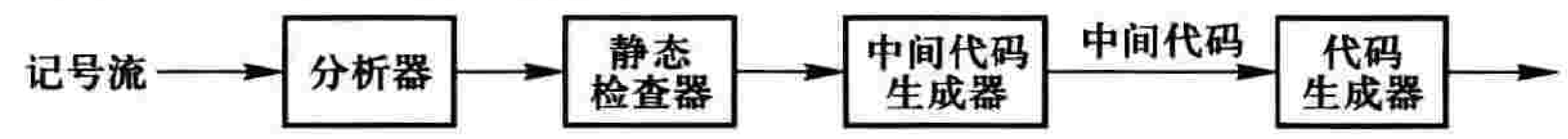


图7.1 中间代码生成器的位置

实际的编译器可能构造一系列的中间表示。高级一点的中间表示接近源语言,语法树就是其中一种,它们适合于完成静态类型检查等任务。低级的中间表示接近目标机器,它们适合于完成依赖于机器的任务,例如寄存器分配和指令选择。本章介绍的三地址代码对高级和低级中间表示都适用,区别主要在于所选择的算符。对于表达式,语法树和三地址代码之间的区别是肤浅的。对于控制流语句,例如循环语句,语法树表示该语句的各个成分,而三地址代码包含标号和跳转指令,以体现控制流,就像机器语言中那样。

中间表示设计的选择随编译器不同而不同。中间表示可以是一种实际的语言,也可以是编译各阶段共享的内部数据结构。C是一种编程语言,但它经常被当作一种中间形式,这是因为它灵活,能生成高效的机器代码,并且它的编译器到处可用。例如,最初的C++编译器就是由一个生成C的前端和作为后端的C编译器组成。

7.1 中间语言

4.2 节介绍的语法树是一种图形化的中间表示,本节再介绍几种常用的中间表示:后缀表示、其他图形表示和三地址代码。本章主要使用三地址代码。从编程语言的各种构造产生三地址代码的语义规则类似于产生语法树或后缀表示的那些规则。

7.1.1 后缀表示

表达式 E 的**后缀表示**可以如下归纳定义:

(1) 如果 E 是变量或常数,那么 E 的后缀表示就是 E 本身。

(2) 如果 E 是形式为 $E_1\ op\ E_2$ 的表达式,其中 op 是任意的二元算符,那么 E 的后缀表示是 $E'_1E'_2op$,其中 E'_1 和 E'_2 分别是 E_1 和 E_2 的后缀表示。

(3) 如果 E 是形式为 (E_1) 的表达式,那么 E_1 的后缀表示也是 E 的后缀表示。

后缀表示不需要括号,因为算符的位置及其运算对象的个数使得后缀表示仅有一种解释。例如,(8-4)+2 的后缀表示是 8 4 - 2 +,而 8-(4+2)的后缀表示是 8 4 2 + -。

上面的定义很容易拓广到含一元算符的表达式。

后缀表示的最大优点是便于计算机处理表达式。利用一个栈,自左向右扫描表达式的后缀表示。每碰到运算对象,就把它压进栈;每碰到运算符,就从栈顶取出相应个数的运算对象进行计算,再将结果压进栈。最终的结果留在栈顶。

后缀表示也可以拓广到表示赋值语句和控制语句,但很难用栈来描述控制语句的计算。

7.1.2 图形表示

语法树是一种图形化的中间表示,它是分析树的浓缩表示,描绘了源程序在语义上的层次结构。后缀表示是语法树的一种线性表示,例如,赋值语句 a=(-b+c*d)+c*d 的语法树如图 7.2(a)所示,对应的后缀表示为

有向无环图(directed acyclic graph, DAG)也是一种中间表示。和语法树相比,它以更紧凑的方式给出同样的信息,因为公共子表达式标识出来了。在图 7.2(b)的 DAG 中,公共子表达式 c*d 不止一个父结点。在考虑代码优化时,有向无环图比语法树更适用。

表 7.1 的语法制导定义构造赋值语句的语法树,它是 4.2 节构造表达式语法树的语法制导定义的一个拓展,其中 *mkUNode*(*op*,*child*)是构造一元运算结点的函数。这里用的是二义文法,假定算符的结合性和优先关系与通常的一样,虽然没有把它们加入文法。这个定义从输入 a=(-b+c*d)+c*d 构造出图 7.2(a)的语法树(按照该定义,叶结点的形式应该像图 4.5 的叶结点

那样)。

a b uminus c d * +c d * +assign

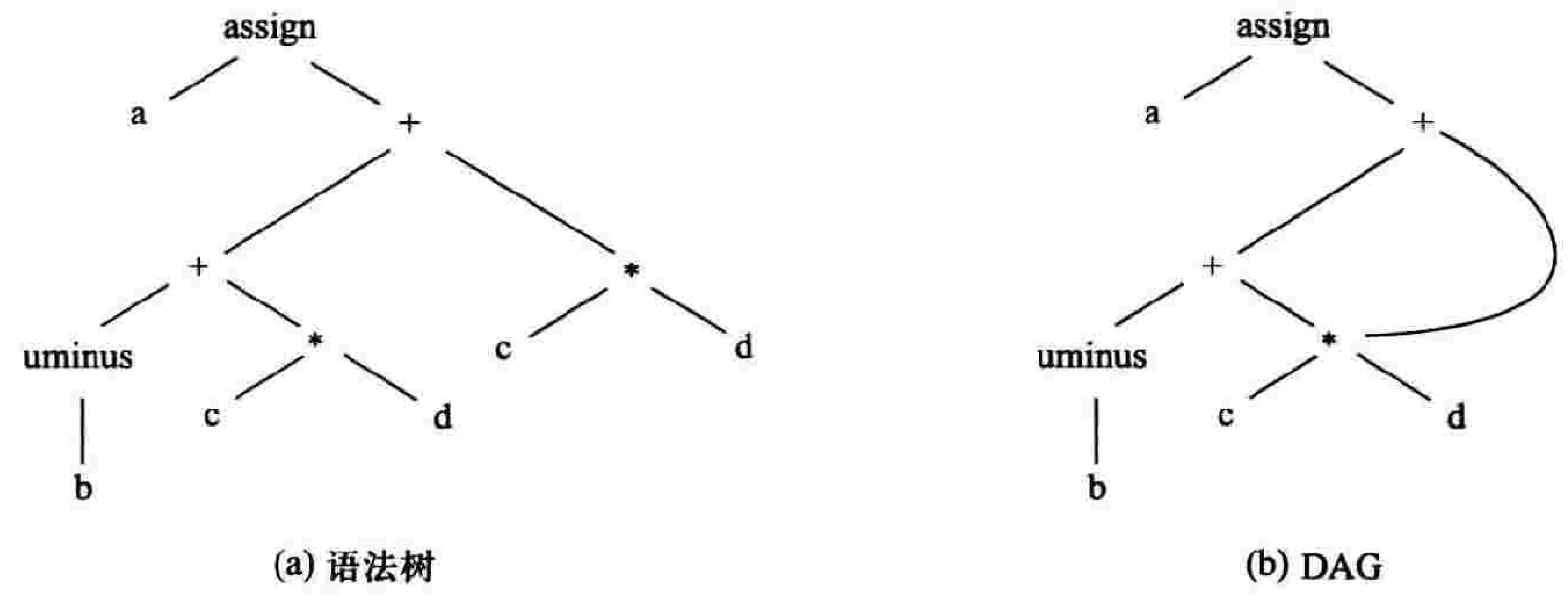


图 7.2 a=(-b+c * d)+c * d 的图形表示

表 7.1 构造赋值语句语法树的语法制导定义

产生式	语义规则
$S \to \mathbf{id} = E$	$S.nptr = mkNode$('assign', $mkLeaf$(**id**, **id**.$entry$), $E.nptr$)
$E \to E_1 + E_2$	$E.nptr = mkNode$('+', $E_1.nptr$, $E_2.nptr$)
$E \to E_1 * E_2$	$E.nptr = mkNode$('*', $E_1.nptr$, $E_2.nptr$)
$E \to -E_1$	$E.nptr = mkUNode$('uminus', $E_1.nptr$)
$E \to (E_1)$	$E.nptr = E_1.nptr$
$E \to \mathbf{id}$	$E.nptr = mkLeaf$(**id**, **id**.$entry$)

修改构造结点的函数,仍然用这个语法制导定义,可以构造出有向无环图。如果构造结点的函数首先检查是否已经有相同的结点存在,有则返回先前构造的这种结点的指针而不是构造新结点,这样就可以得到 DAG。这个语法制导定义从输入 a=(-b+c * d)+c * d 构造出的 DAG 见图 7.2(b)。

7.1.3 三地址代码

三地址代码是一般形式为

$$x = y \ op \ z$$

的指令(也称语句)序列,其中 x、y 和 z 是名字、常数或编译器产生的临时变量,op 代表算符,如定

点或浮点算术算符,或是对布尔类型数据操作的逻辑算符等。之所以叫做三地址代码,是因为每条指令通常包含三个地址,即两个运算对象的地址和一个结果的地址。因为每条指令的右边只有一个算符,所以源语言的表达式 x+y * z 翻译成的三地址指令序列是

$t_1 = y * z$

$t_2 = x + t_1$

其中 t_1 和 t_2 是编译器产生的临时名字。采用临时名字保存中间结果,可以比较容易地为三地址代码重新安排计算次序;而在后缀表示上要改变计算次序是很困难的。

三地址代码是语法树或 DAG 的一种线性表示,其中新增加的临时名字对应图的内部结点。图 7.2 的语法树和 DAG 分别由图 7.3(a)和图 7.3(b)中的三地址指令序列表示。

(a) 对应语法树的代码	(b) 对应 DAG 的代码
$t_1 = -b$	$t_1 = -b$
$t_2 = c * d$	$t_2 = c * d$
$t_3 = t_1 + t_2$	$t_3 = t_1 + t_2$
$t_4 = c * d$	$t_4 = t_3 + t_2$
$t_5 = t_3 + t_4$	$a = t_4$
$a = t_5$	

图 7.3　对应图 7.2 的语法树和 DAG 的三地址代码

三地址指令类似于汇编代码。指令可以有符号标号,而且可以有控制流指令。下面是本书常用的三地址指令,其中符号标号 *L* 是三地址指令序列中某条三地址指令的索引。

(1) 形式为 $x = y\ op\ z$ 的赋值指令,其中 *op* 是二元算术或逻辑算符。

(2) 形式为 $x = op\ y$ 的赋值指令,其中 *op* 是一元算符。一元算符主要包括一元减、逻辑否定、移位和类型转换算符。

(3) 形式为 $x = y$ 的复写指令。

(4) 无条件转移 goto *L*。标号为 *L* 的指令是下一步将要执行的三地址指令。

(5) 形如 if *x* relop *y* goto *L* 的条件转移。这种指令用于 *x* 和 *y* 的关系运算(<、== 和 >= 等)。如果关系成立,执行标号为 *L* 的指令;否则,与通常的顺序一样,执行 if *x* relop *y* goto *L* 的下一个三地址指令。

(6) param *x* 和 call *p*,*n* 用于过程调用,其中 *n* 表示实参个数。return *y* 用于过程返回,*y* 代表返回值,它是可选的。过程调用的典型使用是

param x_1

param x_2

….

param x_n

call p,n

这些指令是过程调用 $p(x_1,x_2,\cdots,x_n)$ 的中间代码。

(7) 形式为 $x=y[i]$ 和 $x[i]=y$ 的索引赋值。第一条指令把 y 的存储单元之下的第 i 个存储单元的值赋给 x。第二条指令把 y 的值赋给 x 的存储单元之下的第 i 个存储单元。在这些指令中,x,y 和 i 都代表数据对象。

(8) 形式为 $x=\&y$,$x=*y$ 和 $*x=y$ 的地址和指针赋值。第一条指令把 y 的存储单元地址赋给 x,可以猜想 y 是名字(或临时变量),它表示像 a 和 A[i][j]这样有左值的表达式,x 是指针名字或临时变量,即 x 的右值是某个对象的左值。在第二条指令中,y 是一个指针或临时变量,它的右值是一个存储单元,x 的右值被置成等于那个存储单元的内容。最后,$*x=y$ 把 y 的右值置入 x 指向的存储单元。

选译适当的算符集合是中间代码设计的重要问题。很显然,它必须大到足以实现源语言的操作。较小的算符集合易于在新的目标机器上实现,但是,较小的算符集合可能迫使前端对源语言的某些运算产生较长的三地址指令序列,此时如果想产生好代码的话,优化器和代码生成器的工作会比较艰巨。

三地址指令是中间代码的抽象形式。在编译器中,三地址指令可以用记录来实现,这种记录有算符域和运算对象域,不同的编译器对这种记录的设计可能是不一样的。

7.1.4 静态单赋值形式

静态单赋值形式(static single-assignment form, SSA)是一种便于某些代码优化的中间表示。SSA 有两个显著特点可区别它和三地址代码。第一个特点是,SSA 中所有赋值指令都是对不同变量的赋值,所以才有静态单赋值这个术语。图 7.4 是用三地址代码和静态单赋值形式写的同一个中间语言程序,注意,在 SSA 表示中,使用不同下标的 p 和 q 代表不同的变量。

(a) 三地址代码	(b) 静态单赋值形式
p = a+b	$p_1 = a+b$
q = p−c	$q_1 = p_1 - c$
p = q * d	$p_2 = q_1 * d$
p = e−p	$p_3 = e - p_2$
q = p+q	$q_2 = p_3 + q_1$

图 7.4 用三地址代码和 SSA 写的中间语言程序

在一个程序中,同一个变量可能在不同的控制流路径上都有定值。例如,源程序

```
if(flag) x = -1; else  x = 1;
y = x * a;
```

有定值 x 的两条不同控制流路径。如果在该条件语句的两个分支为 x 使用不同的名字,那么在赋值 y=x * a 中应该为 x 使用哪个名字?这就是 SSA 第二个特点所在。它使用一种记号上的约

定，叫做 ϕ 函数，以组合 x 的两个定值。如

$$\text{if(flag)}\ x_1=-1;\text{else}\ x_2=1;$$

$$x_3=\phi(x_1,x_2);$$

在这里，如果控制流通过条件为真的部分，则 $\phi(x_1,x_2)$ 的值是 x_1，如果控制流通过条件为假的部分，则 $\phi(x_1,x_2)$ 的值是 x_2。也就是说，$\phi(x_1,x_2)$ 返回它某个变元的值，取决于到达 $\phi(x_1,x_2)$ 时所通过的控制流路径。

7.2 声明语句

在分析过程或程序块的声明序列时，为局部名字建立符号表条目（考虑到作用域等，词法分析不将名字放进符号表，返回⟨**id**，lexeme⟩），并为它分配存储单元。这样，符号表包含各名字的类型和分配给它们的存储单元的相对地址等信息，相对地址是对静态数据区基址的偏移或是对活动记录中某个基址的偏移。

前端分配地址时必须考虑目标机器的一些特点，目标机器的指令系统可能偏爱数据对象和它们地址的某种布局。这里的讨论忽略数据对象的对齐等问题。

7.2.1 过程中的声明

C、Java、Pascal 和 FORTRAN 这些语言的语法允许一个过程中的所有声明集中在一起处理。在这种情况下，可用全局变量，例如 *offset*，来记住下一个可用的相对地址。

在图 7.5 的翻译方案中，非终结符 *P* 产生形式为 **id**:*T* 的声明序列，再产生可执行语句 *S*，本节先考虑声明序列。在检查第一个声明之前，*offset* 置为 0。每次为一个名字分配存储单元时，它的偏移等于 *offset* 的当前值，同时 *offset* 增加由该名字指示的数据对象的宽度。*offset* 初值可能不是 0，并且可能每次都减小，例如第 6 章所介绍的栈分配情况。

过程 *enter*(*name*，*type*，*offset*) 为名字 *name* 建立符号表条目，该名字的类型是 *type*，它在数据区的相对地址是 *offset*。综合属性 *type* 和 *width* 用来表示非终结符的类型和宽度（该类型的对象所需的字节数）。属性 *type* 代表从基本类型 *integer* 和 *real* 等，应用类型构造器 *pointer* 和 *array* 构造出的类型表达式。如果类型表达式用图形表示，那么属性 *type* 可以是一个指针，指向代表类型表达式的结点。

在图 7.5 中，整数宽度是 4，实数宽度是 8，数组的宽度由每个元素的宽度乘以数组元素的个数而得到，每个指针的宽度假定为 4。

$P \to$ { *offset* = 0; }

  $D; S$

$D \to D; D$

$D \to$ **id** : T { *enter*(**id**. *lexeme*, *T. type*, *offset*); *offset* = *offset* + *T. width*; }

$T \to$ **integer** { *T. type* = *integer*; *T. width* = 4; }

$T \to$ **real** { *T. type* = *real*; *T. width* = 8; }

$T \to$ **array** [**num**] **of** T_1 { *T. type* = *array*(**num**. *val*, T_1. *type*);

  T. width = **num**. *val* × T_1. *width*; }

$T \to \uparrow T_1$ { *T. type* = *pointer*(T_1. *type*); *T. width* = 4; }

图 7.5 计算被声明名字的类型和相对地址

7.2.2 作用域信息的保存

现在考虑像 Pascal 这样允许过程嵌套的语言。为简单起见,仅讨论无参过程,并且认为过程不会递归。所讨论语言的文法如下:

$P \to D; S$

$D \to D; D \mid$ **id** : $T \mid$ **proc id** ; $D; S$ (7.1)

在这样的语言里,局部于每个过程的名字仍然可以用图 7.5 的方式来为其分配相对地址,每个过程建立单独的符号表。这样,每个过程要有自己的符号表指针和自己的相对地址 *offset*。

在一边扫描一边建立符号表和完成存储分配的情况下,当碰到过程嵌套时,对外围过程声明的处理需要暂时停止,等被嵌套过程处理完后再继续。可以用两个栈分别保存尚未处理完的过程的符号表指针和它们的相对地址 *offset*,这两个栈的栈顶元素分别是正在处理的过程的符号表指针和相对地址 *offset*。

按照这种方式,基于文法(7.1)的翻译方案见图 7.6,其中 T(类型)产生式的语义动作和图 7.5 的一致,在此略去;非终结符 S(语句)的语义动作在 7.3 节开始介绍。图 7.6 的语义动作根据下面的操作定义。

$P \to M\ D; S$ { *addWidth*(*top*(*tblStack*), *top*(*offsetStack*));

  pop(*tblStack*); *pop*(*offsetStack*); }

$M \to \varepsilon$ { *t* = *mkTable*(*nil*); *push*(*t*, *tblStack*); *push*(0, *offsetStack*); }

$D \to D_1; D_2$

$D \to$ **proc id** ; $N\ D_1; S$ { *t* = *top*(*tblStack*); *addWidth*(*t*, *top*(*offsetStack*));

  pop(*tblStack*); *pop*(*offsetStack*);

  enterProc(*top*(*tblStack*), **id**. *lexeme*, *t*); }

$D \to$ **id** : T { *enter*(*top*(*tblStack*), **id**. *lexeme*, *T. type*, *top*(*offsetStack*));

  top(*offsetStack*) = *top*(*offsetStack*) + *T. width*; }

$N \to \varepsilon$ { *t* = *mkTable*(*top*(*tblStack*)); *push*(*t*, *tblStack*);

  push(0, *offsetStack*); }

图 7.6 处理嵌套过程中的声明

(1) *mkTable*(*previous*):建立新的符号表,并返回新符号表的指针。变元 *previous* 指向先前建立的符号表,可以猜想,这是直接外围过程的符号表。指针 *previous* 放在新建符号表的首部,首部除了包括直接外围过程的符号表的指针外,还包含该符号表中所有局部变量所需存储单元的总数等信息。

(2) *enter*(*table*,*name*,*type*,*offset*):在 *table* 指向的符号表中为变量名 *name* 建立新条目。和前面一样,*enter* 把类型 *type* 和相对地址 *offset* 置于该条目的域中。如果要说得再详细一些的话,本过程还需要完成检查名字是否重复定义等事情。

(3) *addWidth*(*table*,*width*):把 *table* 指向的符号表中所有局部变量条目的累加宽度记录在该符号表的首部。

(4) *enterProc*(*table*,*name*,*newTable*):在 *table* 指向的符号表中为过程名 *name* 建立新条目。变元 *newTable* 指向过程 *name* 本身的符号表。

再对图 7.6 的语义动作做一些解释。对于过程声明 $D \to$ **proc id**;$N\ D_1$;S,在扫描 D_1之前,需要建立新的符号表,让它指向直接外围过程的符号表,并将新符号表指针压入栈 *tblStack*,将 0 压入栈 *offsetStack*,这些动作是通过标记非终结符 N 的动作完成的。然后将 D_1 中声明的名字的条目建立在该新符号表中,同时根据新的 *offset* 进行存储分配。在结束内嵌过程的扫描,执行 $D \to$ **proc id**;$N\ D_1$;S 右边的动作时,由 D_1产生的所有声明的宽度在 *offsetStack* 的栈顶上,用 *addWidth* 把它记录在符号表中,然后栈 *tblStack* 和 *offsetStack* 的顶元退栈,再把过程名 **id** 的条目建立在直接外围过程的符号表中。这些都结束后,继续分析外围过程的声明。

非终结符 M 的动作完成一些初始化的工作,它用操作 *mkTable*(*nil*)建立最外层作用域的符号表,并用两个 *push* 操作来完成栈 *tblStack* 和 *offsetStack* 的初始化。

按图 7.6 的翻译方案,为图 6.14 的 Pascal 程序生成的符号表在图 7.7 给出(没有包括形式参数)。过程的符号表之间用双向链连接,从这个双向链可以知道过程的嵌套关系,例如,过程 readArray、exchange 和 quickSort 的符号表逆向指向直接外围过程 sort 的符号表,而 sort 的符号表中有三个指针指向这三个过程的符号表。

当遇到对名字的引用时,若在本过程的符号表中找不到该名字,那么就需要到它的外围过程的符号表中去找,符号表中的逆向指针可用于这个目的。若语言有预定义的标准标识符(指程序员可以重新定义其含义的标识符),如 Pascal 语言的 integer,true 等,那么主过程符号表的逆向指针应该指向标准标识符的符号表,以便确定程序员没有定义的标识符是不是一个标准标识符。在此实际上已经解释了 7.3 节用到的符号表查找函数 *lookup* 的处理过程。

如果允许过程递归的话,图 7.6 的翻译方案需要修改。在产生式 $D \to$ **proc id**;$N\ D_1$;S 的语义动作中,该过程 **id** 是在处理完该过程体后才进入符号表的,这样,在 S 中若有直接递归调用,在符号表查找该 **id** 时会报告没有定义。

若是含有参数的过程,对构造符号表来说,形式参数的处理和其他局部名字的处理没有很大的区别,但是填在条目中的属性,如存储分配信息等,会有些区别。

如果是一遍扫描的编译器,每个过程被扫描后,它的目标代码已经生成,若无其他需要,该过

程的符号表可以释放。这时编译器可以将符号表组织成一个栈，碰到一个过程声明时将在该过程中声明的名字进栈，对该过程扫描结束时将它的符号全部出栈。

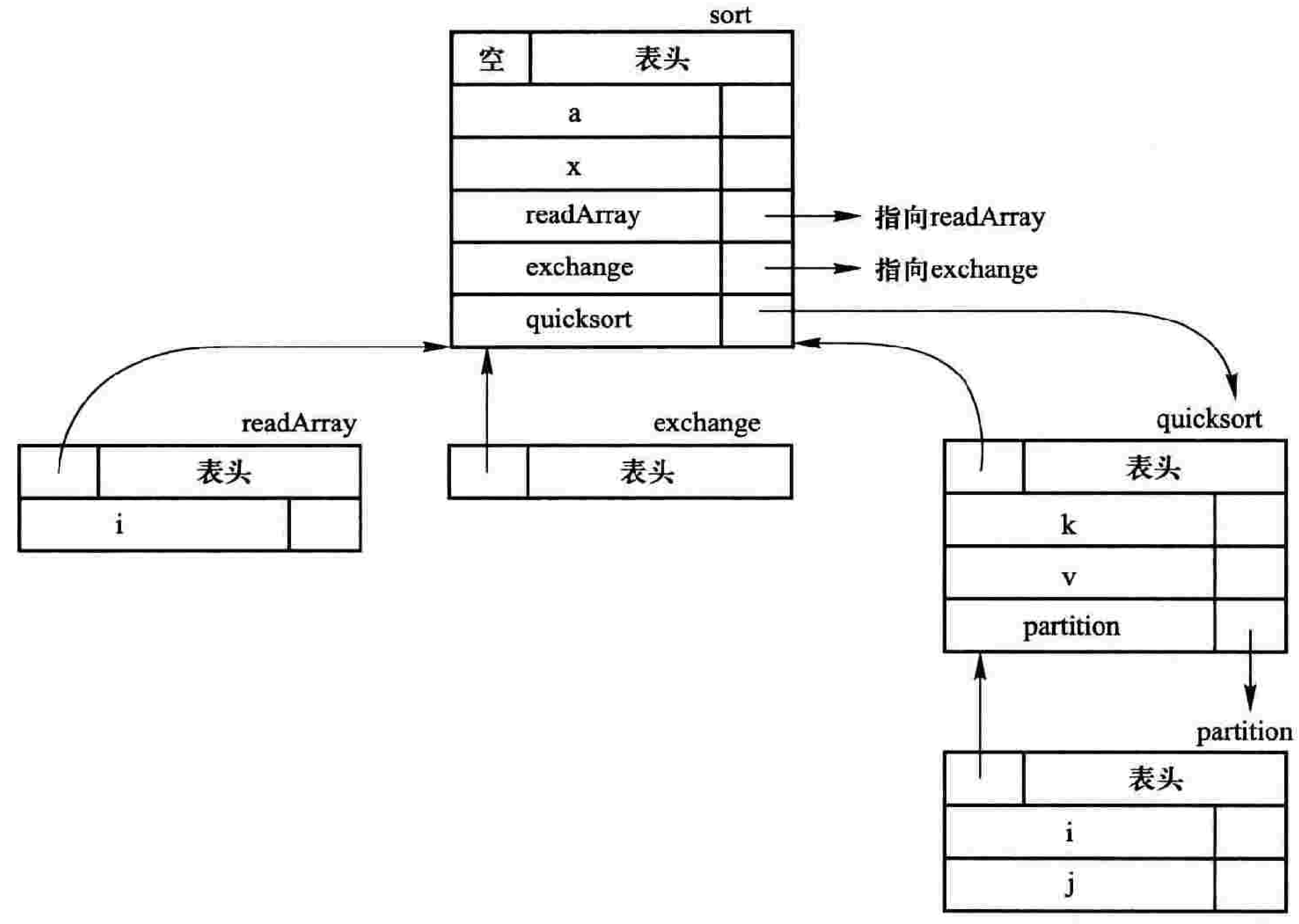


图 7.7 嵌套过程的符号表

7.2.3 记录的域名

在图 7.5 文法上增加一个产生式

T→**record** *D* **end**

使得非终结符 *T* 除了可以产生基本类型、指针和数组外，还能产生记录类型。在图 7.8 的翻译方案中，语义动作强调了记录类型中域的存储布局同过程中局部名字在活动记录中布局的相似性。因为过程定义不影响图 7.6 的宽度计算，因此把上面的产生式放宽到允许过程定义出现在记录类型里面，这只是为了使翻译方案简洁、易懂。

看见关键字 **record** 后，标记非终结符 *L* 的动作为域名建立新的符号表。该符号表的指针入栈 *tblStack*，相对地址 0 入栈 *offsetStack*。产生式 *D*→**id**:*T* 的动作把域名 **id** 的信息加入记录类型的符号表。在记录类型的域分析完后，*offsetStack* 栈顶保存记录类型中所有对象的总宽度。在图 7.8 中，**end** 后的动作把总宽度作为综合属性 *T*. *width* 返回。把类型构造器 *record* 作用于该记录的符号表指针就相当于得到 *T*. *type*。

$T \to$ **record** L D **end** { *T. type* = *record*(*top*(*tblStack*)) ; *T. width* = *top*(*offsetStack*) ;
pop(*tblStack*) ; *pop*(*offsetStack*) ; }

$L \to \varepsilon$ { *t* = *mkTable*(*nil*) ; *push*(*t* , *tblStack*) ; *push*(0 , *offsetStack*) ; }

图 7.8 为记录中的域名建立符号表

记录类型定义和过程声明的处理还是有区别的。处理记录类型时并未真正为哪个变量分配存储单元，只是决定了该类型的宽度和每个域在记录存储空间中的相对位置。在处理记录类型的变量声明时，才真正在活动记录中为记录类型的变量分配存储单元。

7.3 赋值语句

本节中表达式的类型可以是整型、实型和数组。为使翻译方案简洁，采用二义的表达式文法，并选择加运算作为二元运算的代表。作为赋值语句翻译成三地址指令的一部分工作，下面说明怎样从符号表中查找名字，怎样访问数组的元素和记录的域。

7.3.1 符号表中的名字

在 7.1 节介绍中间语言时，为直观起见，让名字本身直接出现在三地址代码中，实际上应该把名字理解为它们在符号表中位置的指针。编译器在处理表达式、赋值语句等构造中的名字时，需要在符号表中查找它的定义，获得它的属性，然后在生成的三地址代码中使用它在符号表中位置的指针。

在图 7.9 的翻译方案中，名字 **id** 的 *lexeme* 属性代表组成该名字的字符序列，*lookup*(**id**. *lexeme*)用于根据名字的拼写检查符号表中是否存在该名字的条目。如果有，返回该条目的指针，否则 *lookup* 返回 *nil*，以表示没有找到。是否有过程嵌套，会影响 *lookup* 函数的设计，但不影响图7.9 的翻译方案使用该函数来查符号表。

$S \to$ **id** := E { *p* = *lookup*(**id**. *lexeme*) ;
if(*p* != *nil*) *emit*(*p* , ' = ' , *E. place*) ;
else *error* ; }

$E \to E_1 + E_2$ { *E. place* = *newTemp*( ) ;
emit(*E. place* , ' = ' , E_1. *place* , ' + ' , E_2. *place*) ; }

$E \to -E_1$ { *E. place* = *newTemp*( ) ;
emit(*E. place* , ' = ' , 'uminus ' , E_1. *place*) ; }

$E \to (E_1)$ { *E. place* = E_1. *place* ; }

$E \to$ **id** { *p* = *lookup*(**id**. *lexeme*) ;
if(*p* != *nil*) *E. place* = *p* ;
else *error* ; }

图 7.9 为赋值语句产生三地址代码的翻译方案

E 的属性 *place* 用来记住符号表条目的地址。函数 *newTemp* 用来产生一个新的临时变量的名字,把该名字也存入符号表,并返回该条目的地址。过程 *emit* 将其参数写到输出文件上,*emit* 的参数构成一条三地址指令。

如果碰到像 p.info 这样对记录域的访问,如何查表找到所需的属性呢?首先,从过程的名字表中可以找到 p,它应该是记录类型的变量。根据 7.2.3 节可以知道,p 的类型是 *record*(*tblptr*),其中 *tblptr* 是该记录类型的符号表,从该表中可以找到 info 的条目,从而可以得到所需的属性。

7.3.2 数组元素的地址计算

一个数组的所有元素通常按一定的次序存于连续的存储块中,这样可以迅速访问这些元素。一维数组的元素一般是顺序存放,如果每个数组元素的宽度是 w,那么一维数组 A 的第 i 个元素从地址

$$base+(i-low)\times w \tag{7.2}$$

开始,其中 *low* 是下标的下界,*base* 是分配给该数组的地址(可能是活动记录中的相对地址),即 *base* 是 A[*low*]的地址。

如果把表达式(7.2)重写成

$$i\times w+(base-low\times w)$$

那么可以在编译时完成该表达式后一部分的计算,从而减少了运行时的计算。

二维数组通常用两种形式之一存储:行为主(一行接一行)或列为主(一列接一列)。FORTRAN 系列语言使用列为主形式,C、Java 和 Pascal 都使用行为主形式。对于一个 2×3 的数组 A,若是行为主,其元素的存放次序是:

A[1,1],A[1,2],A[1,3],A[2,1],A[2,2],A[2,3]

若是列为主,其元素的存放次序是:

A[1,1],A[2,1],A[1,2],A[2,2],A[1,3],A[2,3]

在行为主的情况下,因为 A[i,j]等价于 A[i][j],因此可以说是各一维数组 A[i]依次连续存放。

在行为主的二维数组情况下,A[i_1,i_2]的地址可以由公式

$$base+((i_1-low_1)\times n_2+(i_2-low_2))\times w$$

计算,其中 low_1 和 low_2 分别是这两维的下界,n_2 是第二维的大小。即,如果 $high_2$ 是 i_2 的上界,那么 $n_2=high_2-low_2+1$。假定 i_1 和 i_2 都是编译时不能确定的值,可以把上面表达式重写成

$$((i_1\times n_2)+i_2)\times w+(base-((low_1\times n_2)+low_2)\times w) \tag{7.3}$$

同样,该表达式的后一项可以在编译时计算。

可以推广行为主或列为主的形式到多维数组。行为主形式的存储可以这样理解,当朝高地址扫描存储块时,最右边的下标变化最快,就像里程计上的数字。表达式(7.3)的推广使得 A[$i_1,i_2,\cdots,i_k$]的地址表达式如下:

$$((\cdots((i_1\times n_2+i_2)\times n_3+i_3)\cdots)\times n_k+i_k)\times w$$

$$+base-((\cdots((low_1\times n_2+low_2)\times n_3+low_3)\cdots)\times n_k+low_k)\times w \tag{7.4}$$

因为对所有的 $j,n_j=high_j-low_j+1$ 是固定的,(7.4)式第二行的项可以在编译器分析数组声明时计算,存于符号表的 A 条目中。列为主的形式的布局相反,最左边的下标变化最快。

某些语言允许数组的大小在过程调用时动态确定,这种数组在运行栈上的分配在 6.2 节已经讨论了,其元素的访问公式和固定大小的数组相同,但由于上下界在编译时不知道,因此地址计算全部在运行时完成。

7.3.3 数组元素地址计算的翻译方案

根据(7.4)式知道,k 维数组引用 $A[i_1,i_2,\cdots,i_k]$ 的三地址代码主要是完成

$$(\cdots((i_1\times n_2+i_2)\times n_3+i_3)\cdots)\times n_k+i_k \tag{7.5}$$

的计算,然后乘以 w,再加上(7.4)式第二行的值。(7.5)式的值可以由递推计算

$$e_1=i_1$$

$$e_m=e_{m-1}\times n_m+i_m \tag{7.6}$$

来完成,一直到 $m=k$ 为止。

根据(7.6)式的递推计算,除了第一个下标表达式 i_1 外,对其他每一个下标表达式,需要产生乘和加两条三地址指令(没有包括下标表达式本身值计算的三地址指令),因此在处理下标表达式时需要访问符号表中 A 的条目,以得到 A 各维的大小。

如果用下列产生式

L→**id**[*Elist*] | **id**

Elist→*Elist*, *E* | *E*

的非终结符 *L* 取代图 7.9 中 **id**,那么数组引用可以出现在赋值语句中。如果仅用综合属性,在处理 *Elist*→*E* 和 *Elist*→*Elist*, *E* 时,访问不到符号表中 A 的条目,因为这是数组 **id** 的属性。加标记非终结符也解决不了这个问题,为此将产生式重写为

L→*Elist*] | **id**

Elist→*Elist*, *E* | **id**[*E*

即数组名和最左边一个下标表达式连在一起。这个文法虽然不直观,但是从下面的翻译方案可以知道它能解决上面所提到的问题。

使用 *Elist* 的综合属性 *array* 来传递符号表中数组名条目的指针,并使用 *Elist. ndim* 来记录已分析过的下标表达式的个数。函数 *limit*(*array*, *j*)返回 n_j,它是 *array* 指向的数组中第 *j* 维的大小。函数 *invariant*(*array*)从 *array* 所指符号表条目中取静态可计算的值,即(7.4)式第二行的值,函数 *width*(*array*)从 *array* 所指符号表条目中取数组元素的宽度。最后,*Elist. place* 指示临时变量,该变量保存根据 *Elist* 的下标表达式计算的值。

左值 *L* 有两个属性:*place* 和 *offset*。*L. offset* 等于 **null** 时,则表示该左值是一个简单名字,此时 *L. place* 是该名字的符号表条目指针(直接用 **id**. *palce* 表示)。*L* 是数组引用时这两个属性的

用途见下面的解释。和图 7.9 一样,非终结符 E 有 *place* 属性,并且含义一样。

将语义动作加到下面文法上:

(1) $S \to L := E$

(2) $E \to E + E$

(3) $E \to (E)$

(4) $E \to L$

(5) $L \to Elist\,]$

(6) $L \to \mathbf{id}$

(7) $Elist \to Elist, E$

(8) $Elist \to \mathbf{id}[E$

和 7.3.1 节的表达式一样,三地址代码由调用过程 *emit* 产生。

如果 L 是简单名字,则产生正常的赋值;否则产生对由 L 的两个属性确定的存储单元的索引赋值:

```
(1) S→L:=E      {if(L.offset==null) emit(L.place,'=',E.place);
                                                    /* L是简单变量 */
                 else  emit(L.place,'[',L.offset,']','=',E.place);}
```

算术表达式的代码和图 7.9 的完全一样:

```
(2) E→E1+E2     {E.place=newTemp();emit(E.place,'=',E1.place,'+',E2.place);}
(3) E→(E1)      {E.place=E1.place;}
```

如果 E 产生数组元素 L,则需要 L 的右值,可用索引得到存储单元 *L.place*[*L.offset*]的内容:

```
(4) E→L         {if(L.offset==null) E.place=L.place /* L是简单变量 */
                 else  begin
                     E.place=newTemp();
                     emit(E.place,'=',L.place,'[',L.offset,']');
                 end}
```

L.place 和 *L.offset* 都是新的临时变量,前者对应(7.4)式的第二项;后者保存 w 乘以 *Elist.place* 的值,对应(7.4)式的第一项:

```
(5) L→Elist]    {L.place=newTemp();emit(L.place,'=',invariant(Elist.array));
                 L.offset=newTemp();
                 emit(L.offset,'=',Elist.place,'*',width(Elist.array));}
```

offset 为空时表示简单名字:

```
(6) L→id        {L.place=id.place;L.offset=null;}
```

当看见下一个下标表达式时,使用递推公式(7.6)。在下面的动作中,$Elist_1.place$ 对应(7.6)式的 e_{m-1},*Elist.place* 对应 e_m。如果 $Elist_1$ 有 $m-1$ 个成分,那么产生式左边的 *Elist* 有 m 个成分:

(7) $Elist \to Elist_1$, E　　{ $t = newTemp(\,); m = Elist_1.ndim + 1;$
$emit(t, '=', Elist_1.place, '*', limit(Elist_1.array, m));$
$emit(t, '=', t, '+', E.place); Elist.array = Elist_1.array;$
$Elist.place = t; Elist.ndim = m;$ }

下面的 $E.place$ 保存表达式 E 的值,也是(7.6)式 $m=1$ 时的值:

(8) $Elist \to$ **id** [E　　{ $Elist.place = E.place; Elist.ndim = 1; Elist.array =$ **id**. $place;$ }

例 7.1　设 A 是 10×20 的数组, $n_1 = 10$ 且 $n_2 = 20$, 取 $w = 4$。赋值语句 x: = A[y,z]的注释分析树如图 7.10 所示。该赋值语句翻译成下列三地址指令序列

$t_1 = y * 20$

$t_1 = t_1 + z$

$t_2 = c$　　　/* 常量 $c = base_A - 84$ */

$t_3 = t_1 * 4$

$t_4 = t_2[t_3]$

$x = t_4$

对每个变量, **id**. $place$ 已经由它的名字代替了。 □

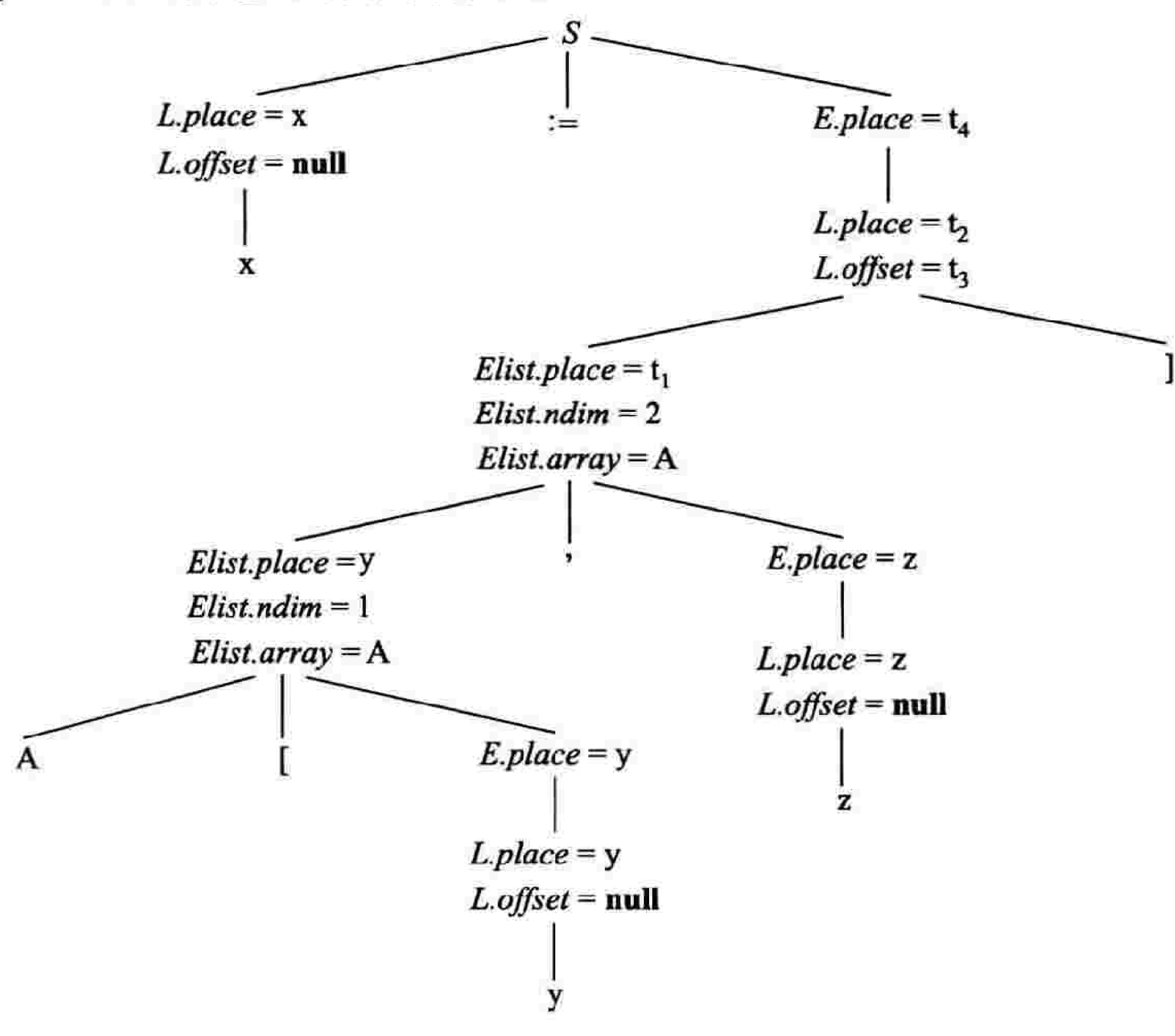


图 7.10　x: = A[y, z]的注释分析树

7.3.4 类型转换

到目前为止的中间代码生成,都忽略了可能有的数据对象的类型转换操作。考虑上述赋值语句的文法,假定有整数和实数两种类型,必要时整数可转为实数。这里直接使用在5.3节已经熟悉的属性 *E. type*,它的值是 *real* 或 *integer*,并且还忽略类型错误的检查。

E→*E*+*E* 和大多数其他产生式的语义动作都可以修改成必要时产生 x = inttoreal y 的三地址指令,它的作用是把整数 *y* 转换成值相等的实数,再赋给 *x*,还必须给算符一个标记,用以标明这是定点还是浮点算术运算。产生式 $E \to E_1 + E_2$ 的完整语义动作列在图7.11中。

例如,假定 x 和 y 的类型是 real,i 和 j 的类型是 integer,对于输入

x = y+i * j

根据图7.11,输出的三地址指令序列是

t_1 = i int * j

t_2 = inttoreal t_1

t_3 = y real+t_2

x = t_3

图7.11的语义动作为非终结符 *E* 使用了两个属性 *E. place* 和 *E. type*。

```
E.place = newTemp();
if(E1.type == integer) && (E2.type == integer) begin
    emit(E.place, '=', E1.place, 'int+', E2.place);
    E.type = integer;
end else if(E1.type == real) && (E2.type == real) begin
    emit(E.place, '=', E1.place, 'real+', E2.place);
    E.type = real;
end else if(E1.type == integer) && (E2.type == real) begin
    u = newTemp(); emit(u, '=', 'inttoreal', E1.place);
    emit(E.place, '=', u, 'real+', E2.place); E.type = real;
end else if(E1.type == real) && (E2.type == integer) begin
    u = newTemp(); emit(u, '=', 'inttoreal', E2.place);
    emit(E.place, '=', E1.place, 'real+', u); E.type = real;
end else
    E.type = type_error;
```

图7.11 $E \to E_1 + E_2$ 的语义动作

7.4 布尔表达式和控制流语句

在编程语言中，布尔表达式有两个基本目的，第一是用于计算逻辑值，第二而且更经常的是作为条件表达式，用于控制流语句，如 if-then、if-then-else 和 while-do 语句。因此本节将布尔表达式和控制流语句一起讨论。

7.4.1 布尔表达式

布尔表达式也可以像算术表达式那样归纳地定义。下面是本节所用的布尔表达式文法，其中 **relop** 是关系算符。

$B \to B$ **or** B | B **and** B | **not** B | (B) | E **relop** E | **true** | **false**

按惯例，**or** 和 **and** 都是左结合的，**or** 的优先级最低，然后是 **and**，最后是 **not**。

在有些情况下，只要完成了布尔表达式的部分计算就可以知道它的真假，那么它剩余部分是否还要计算？这由编程语言的语义决定的。C 和 Java 采用部分计算，称为短路计算。短路计算有严格的语义规定，把 B_1 **or** B_2定义成

if B_1 **then true else** B_2

把 B_1 **and** B_2定义成

if B_1 **then** B_2 **else false**

如果 B_1或 B_2是有副作用的表达式（例如含修改某个全局变量的函数调用），则完全计算和短路计算的结果可能是有区别的。

表示布尔表达式的值有两种主要方法。第一种方法是把真和假数值化，使布尔表达式的计算类似于算术表达式的计算，常常用 1 表示真，用 0 表示假。当然还有其他编码方式，例如，用非 0 表示真，用 0 表示假；或者用非负数表示真，用负数表示假。显然，在这种情况下，布尔表达式的中间代码生成和算术表达式没有多少区别。

实现布尔表达式的第二种方法是借助控制流，即用程序中的位置来表示布尔表达式的值，它适用于短路计算的情况，用这种方式实现控制流语句中的布尔表达式尤其方便。因为对于控制流语句来说，只要能根据布尔表达式的结果跳转到正确的位置继续执行就可以了，对它的具体值无须再关心。7.4.3 节介绍布尔表达式在这种方式下的中间代码生成。

7.4.2 控制流语句的翻译

现在考虑 if-then，if-then-else，while-do 和顺序语句的三地址指令翻译，这些语句由下面的文法产生：

$S \rightarrow$ **if** B **then** S_1

　　|**if** B **then** S_1 **else** S_2

　　|**while** B **do** S_1

　　|S_1; S_2

在这些产生式中，B 是布尔表达式。图 7.12 用图形表示出这四个语句的三地址代码的结构。图中 *B. code* 和 *S. code* 分别表示 B 和 S 的三地址代码，它们在图中的次序就表示了它们在整个代码中的次序。例如在 if-then 的结构图中，*B. code* 在 *S. code* 的上面，它表示 B 的三地址代码在 S 的三地址代码前面。

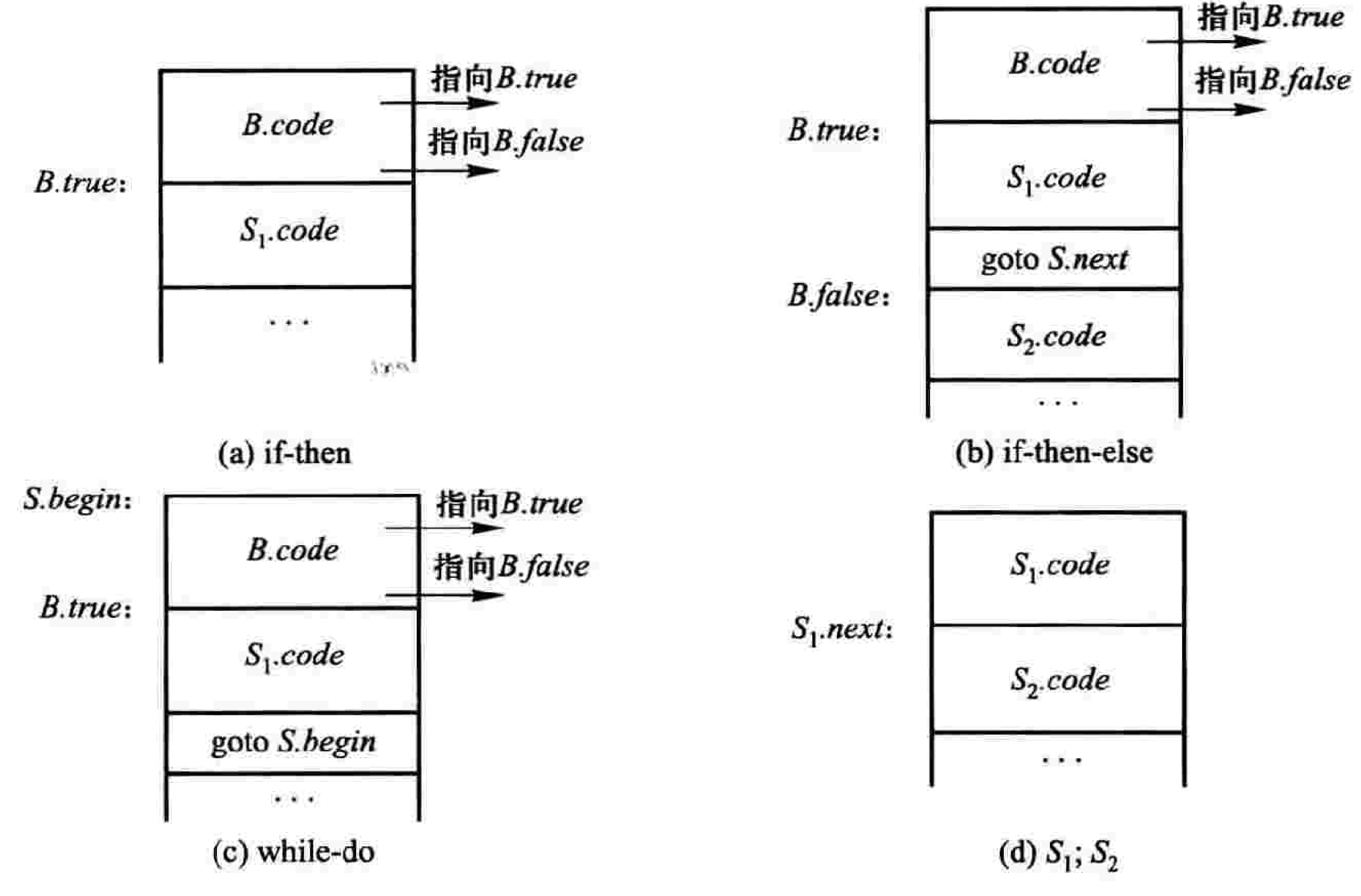


图 7.12　if-then，if-then-else，while-do 和顺序语句的代码

图 7.12 中的 *B. true*，*B. false*，*S. begin* 和 *S. next* 都是三地址指令的标号，它们都是继承属性。图 7.12(a) 为 if-then 的结构图，由于 *B. code* 究竟有多少条指令需等 B 翻译结束才能知道，因此在翻译 B 的过程中难以知道 *B. true* 在三地址指令序列中的准确位置，为便于翻译，采用给三地址指令加标号的方式，函数 *newLabel* 每次调用时返回一个新的标号。

对布尔表达式 B，用两个标号 *B. true* 和 *B. false* 分别表示 B 为真和为假时控制流应该转向的标号，这两个属性由 B 的上下文决定。*S. begin* 是 S 第一条三地址指令的标号，*S. next* 表示执行完 S 后应该执行的第一条三地址指令的标号，它们都由 S 的上下文决定。对于 S_1;S_2的情况，S_1. *next* 显然应该定位在 S_2的第一条三地址指令处，如图 7.12(d) 所示。对于其他语句，情况就不这么简单了，见图 7.12(b) 所示的 if-then-else 语句。在 S_1. *code* 的后面有 goto *S. next*，这条指令是需要的，因为 S_1执行结束意味着 S 执行结束，并且当 S_1是赋值语句时肯定会执行这条指令。问题是 S_1. *next* 应该定位在什么地方？它可以定位在指令 goto *S. next* 处，但这意味着如果 S_1中有

三地址指令跳转到 $S_1.next$ 的话，那么紧接着再执行 goto $S.next$。这种连续跳转显然应该避免并且可以避免，具体做法是暂不定位 $S_1.next$，让它等于 $S.next$。出于同样的原因，$S.next$ 不一定定位在 S_2的后面并且紧挨着 S_2的地方。

控制流语句的语法制导定义在表 7.2。和赋值语句的翻译方案不一样的是，用函数 *gen* 代替了过程 *emit*，*gen* 把形成的代码串作为函数值，而不是输出到文件上。还有，‖ 是作为串的连接算符。

表 7.2 没有给出布尔表达式的语法制导定义，它在下一小节另行给出。

表 7.2　控制流语句的语法制导定义

产生式	语义规则
$S \to$ **if** B **then** S_1	$B.true = newLabel()$ $B.false = S.next$　　$S_1.next = S.next$ $S.code = B.code \parallel gen(B.true, ':') \parallel S_1.code$
$S \to$ **if** B **then** S_1 **else** S_2	$B.true = newLabel()$　$B.false = newLabel()$ $S_1.next = S.next$　　$S_2.next = S.next()$ $S.code = B.code \parallel gen(B.true, ':') \parallel S_1.code \parallel$ 　　$gen('goto', S.next) \parallel gen(B.false, ':') \parallel S_2.code$
$S \to$ **while** B **do** S_1	$S.begin = newLabel()$　　$B.true = newLabel()$ $B.false = S.next$　　$S_1.next = S.begin()$ $S.code = gen(S.begin, ':') \parallel B.code \parallel gen(B.true, ':') \parallel$ 　　$S_1.code \parallel gen('goto', S.begin)$
$S \to S_1 ; S_2$	$S_1.next = newLabel()$　　$S_2.next = S.next()$ $S.code = S_1.code \parallel gen(S_1.next, ':') \parallel S_2.code$

7.4.3　布尔表达式的控制流翻译

现在讨论上一节的 $B.code$，即为布尔表达式 B 产生中间代码，把 B 翻译成一串条件转移和无条件转移的三地址指令序列。

设计这个翻译的基本想法是，假定 B 是 $a<b$ 的形式，那么生成的代码形式为：

if $a<b$ goto $B.true$

goto $B.false$

设 B 是 B_1 **or** B_2的形式。如果 B_1为真，那么可以知道 B 本身为真，即 $B_1.true$ 和 $B.ture$ 一样。如果 B_1为假，那么 B_2必须计算，可以把 $B_1.false$ 定位在 B_2代码第一条指令处。B_2的 *true* 和 *false* 分别与 B 的 *true* 和 *false* 一样。

类似的考虑可用于 B_1 **and** B_2 的翻译。形式为 **not** B_1 的表达式无须产生新的代码，只要交换 B 的 *true* 和 *false* 就得到 B_1 的 *true* 和 *false*。按这种方法为布尔表达式生成三地址代码的语法制导定义见表 7.3。

例 7.2 考虑布尔表达式

a<b or c<d and e<f

假定整个表达式为真和为假的出口分别置为 L_{true} 和 L_{false}。那么用表 7.3 的定义可以得到下列代码：

```
    if a<b goto Ltrue
    goto L1
L1: if c<d goto L2
    goto Lfalse
L2: if e<f goto Ltrue
    goto Lfalse
```

生成的代码还不是最优的，因为删掉第二条指令不会改变代码的值。这种形式的冗余指令不难在以后的处理中删除。避免产生这些冗余转移的另一个办法是把形式为 $E_1<E_2$ 的关系表达式翻译成 if $E_1 \geq E_2$ goto $B.false$，即条件为真时执行正文中紧跟着它的代码，而条件为假时跳转，简称为假转方式。 □

表 7.3 布尔表达式的语法制导定义

产生式	语义规则
$B \to B_1$ **or** B_2	$B_1.true = B.true$　　$B_1.false = newLabel()$ $B_2.true = B.true$　　$B_2.false = B.false()$ $B.code = B_1.code \parallel gen(B_1.false, ':') \parallel B_2.code$
$B \to B_1$ **and** B_2	$B_1.true = newLabel()$　　$B_1.false = B.false$ $B_2.true = B.true$　　$B_2.false = B.false$ $B.code = B_1.code \parallel gen(B_1.true, ':') \parallel B_2.code$
$B \to$ **not** B_1	$B_1.true = B.false$　　$B_1.false = B.true$　　$B.code = B_1.code$
$B \to (B_1)$	$B_1.true = B.true$　　$B_1.false = B.false$　　$B.code = B_1.code$
$B \to E_1$ **relop** E_2	$B.code = E_1.code \parallel E_2.code$ $\parallel gen($'if'$, E_1.place,$ **relop**$.op, E_2.place,$'goto'$, B.true)$ $\parallel gen($'goto'$, B.false)$
$B \to$ **true**	$B.code = gen($'goto'$, B.true)$
$B \to$ **false**	$B.code = gen($'goto'$, B.false)$

例 7.3 考虑语句

```
while a<b do
    if c<d then
        x:=y+z
    else
        x:=y-z
```

表 7.3 的语法制导定义、赋值语句的翻译方案和控制流语句的语法制导定义合在一起，为该语句产生下列代码：

L_1: if a < b goto L_2
    goto L_{next}
L_2: if c < d goto L_3
    goto L_4
L_3: t_1 = y+z
    x = t_1
    goto L_1
L_4: t_2 = y−z
    x = t_2
L_{next}: goto L_1

改变条件测试的方向可以删除前两个 goto 语句。 □

7.4.4 开关语句的翻译

在许多语言中都有“开关”语句或者“分情况”语句，这里讨论的开关语句如下：

switch E
    begin
        case V_1: S_1
        case V_2: S_2
            …
        case V_{n-1}: S_{n-1}
        default: S_n
    end

它的执行流程是：

（1）计算开关表达式 E 的值。

（2）分支测试，即在 $n-1$ 个常量 $V_1, V_2, \cdots, V_{n-1}$ 中寻找和 E 值相同的常量。若存在，则认为该常量和 E 匹配，否则认为默认值 **default** 和 E 匹配。

(3) 执行相匹配常量后的分支的语句。

(4) 执行了这三步后就代表开关语句执行结束。

步骤(2)的分支测试有多种实现方法。如果分支数不是很多,比方说少于10,那么把开关语句翻译成下面的条件转移序列是合理的。

```
        t=E 的代码
        if t! =V1 goto L1
        S1的代码
        goto next
L1:     if t! =V2 goto L2
        S2的代码
        goto next
L2:…
        …
Ln-2:if t! =Vn-1 goto Ln-1
        Sn-1的代码
        goto next
Ln-1:Sn的代码
next:
```

该方式的缺点是,由于分支测试的代码散在各处,很难对它们进行专门的优化处理。为提高效率,程序员应该把经常出现的分支放在前面。

另一种翻译方式是将分支测试的代码集中在一起,放在该语句代码的后部。

```
        t=E 的代码
        goto test
L1:     S1的代码
        goto next
L2:     S2的代码
        goto next
        …
Ln-1:Sn-1的代码
        goto next
Ln:Sn的代码
        goto next
test:if t==V1 goto L1
        if t==V2 goto L2
        …
```

```
        if  t == V_{n-1} goto  L_{n-1}
        goto L_n
    next:
```

把分支测试序列的代码放在该语句代码的前部对一遍扫描的编译器来说是不方便的，因为它不可能在扫描每个 S_i 前就产生这样的测试代码，而扫描每个 S_i 时又必须产生 S_i 的代码。

为了便于识别从标号 test 开始的测试是开关语句的分支测试序列，以利于代码生成器对它进行特别处理，可以给中间代码增加一种 case 指令，将分支测试序列的中间代码改成下面的形式：

```
    test: case V_1   L_1
          case V_2   L_2
          …
          case V_{n-1} L_{n-1}
          case t   L_n
    next:
```

其中 case V L 和 if t==V goto L 的含义一样。

代码生成器实现这个测试序列的一种紧凑办法是建立一张常量值和标号的二元组表。通过一个循环，将开关表达式的值和表中的各个值相比较，如果没有其他的值可匹配，最后的默认条目肯定匹配。

如果常量值的数目较多，那么用散列表效率会更高一些。

对一种经常出现的特殊情况，可以用更有效的办法实现 n 个分支。如果所有的常量值落在一个较小的区间，比方说 i_{min} 和 i_{max} 之间，并且 $n/(i_{max}-i_{min})$ 的值较大，那么可以构造标号表，让与常量 j 对应的标号放在偏移是 $j-i_{min}$ 的条目中，空白的条目全填上默认语句的标号。执行该语句时，先计算开关表达式，获得它的值 m，检查 m 是否在 i_{min} 和 i_{max} 之间，若不在其中，则直接执行默认语句；若在其中，则跳转到偏移为 $m-i_{min}$ 的条目所指地址去执行。

为把开关语句翻译成上面形式的代码，编译器看见保留字 **switch** 时，产生两个新的标号 test 和 next 以及新的临时变量 t，在分析表达式 E 时，产生计算 E 到 t 的代码，并产生跳转语句 goto test。然后为每个 case V_i：S_i 产生新建的标号 L_i，后面跟着 S_i 的代码，再后面是跳转 goto next 指令。与此同时，将标号 L_i 加入符号表，并将这个条目的指针和常量 V_i 放入专用于存储 V_i 和 L_i 对应关系的队列中。当看见终止开关体的保留字 **end** 时，用这个队列中的信息产生分支测试的指令序列。

7.4.5 过程调用的翻译

第 6 章已详细介绍了过程调用的实现，包括存储空间的组织、过程调用序列和返回序列等，本节用下面的文法：

S→**call id** (*Elist*)

Elist→*Elist*, *E*

Elist→*E*

简单介绍过程调用语句的中间代码生成。

过程调用的不同实现方式产生不同的过程调用序列和返回序列,为了让中间表示能方便地用于不同的实现方式,特别为它增设一种 param 指令,专用于指示实在参数,就像开关语句的中间代码 case 指示分支测试一样。过程调用 **id**($E_1,E_2,\cdots,E_n$)的中间代码结构如下:

$E_1.place=E_1$的代码

$E_2.place=E_2$的代码

…

$E_n.place=E_n$的代码

param $E_1.place$

param $E_2.place$

…

param $E_n.place$

call **id**.*place*, *n*

根据上面的代码结构可知,在生成 $E_i.place=E_i$的代码后,需要保存 $E_i.place$ 的值,队列是保存这些值的一种适当的数据结构。下面是采用这种数据结构的翻译方案:

S→**call id**(*Elist*)	{为长度为 *n* 的队列中每个 *E.place*,执行 *emit*('param',*E.place*); *emit*('call',**id**.*place*,*n*)}
Elist→*Elist*,*E*	{把 *E.place* 放入队列末尾}
Elist→*E*	{将队列初始化,并让它仅含 *E.place*}

在用 Yacc 等生成器来描述时,这个队列一般声明成一个全局的数据结构。

对于返回语句,产生它的中间代码是一件直截了当的事情。

习 题 7

7.1 把算术表达式-(a+b) * (c+d)+(a+b+c)翻译成:

(a) 语法树;

(b) 有向无环图;

(c) 后缀表示;

(d) 三地址代码。

7.2 把 C 程序

```
main() {
    int i;
```

```
    int a[10];
    while(i<=10)
        a[i]=0;
}
```

的可执行语句翻译成:

(a) 语法树;

(b) 后缀表示;

(c) 三地址代码。

*7.3 证明:如果所有算符都是二元的,那么算符和运算对象的串是后缀表达式,当且仅当

(1) 算符个数正好比运算对象个数少一个;

(2) 在这个表达式的每个非空前缀中,算符数少于运算对象数。

7.4 修改图 7.5 中计算声明名字的类型和相对地址的翻译方案,允许名字表而不是单个名字出现在形式为 $D \rightarrow \mathbf{id}:T$ 的声明中。

7.5 修改图 7.5 中计算声明名字的类型和相对地址的翻译方案,*offset* 不是全局变量,而是文法符号的继承属性。

7.6 算符 θ 作用于表达式 $e_1, e_2, \cdots, e_k$ 的前缀形式是 $\theta\ p_1 p_2 \cdots p_k$,其中 p_i 是 e_i 的前缀形式。

(a) 写出 a * -(b+c) 的前缀形式。

*(b) 证明:所有的语义动作都是打印,并且所有的动作都出现在产生式右部末端的翻译方案不可能把中缀表达式翻译成前缀表达式。

(c) 给出把中缀表达式翻成前缀形式的语法制导定义。

7.7 编一个程序,实现表 7.3 的翻译布尔表达式到三地址代码的语法制导定义。

7.8 表 7.3 的语法制导定义把 $B \rightarrow E_1 < E_2$ 的控制转移部分翻译成两条指令

if $E_1.place < E_2.place$ goto …

goto …

它们可以用一条指令

if $E_1.place >= E_2.place$ goto …

来代替,当 B 为真时执行后继代码。修改表 7.3 的语法制导定义,使之产生这种性质的代码。

*7.9 下面的 C 语言程序

```
main() {
    int i,j;
    while((i || j)&&(j>5)) {
        i=j;
    }
}
```

在 x86/Linux 系统上编译生成的汇编代码如下(编译器版本见汇编代码最后一行):

```
        .file"bool.c"
        .version"01.01"
gcc2_compiled.:
.text
        .align 4
.globl main
        .type main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp
        nop
        .p2align 4,,7
.L2:
        cmpl $0,-4(%ebp)
        jne .L6
        cmpl $0,-8(%ebp)
        jne.L6
        jmp.L5
        .p2align 4,,7
.L6:
        cmpl $5,-8(%ebp)
        jg.L4
        jmp.L5
        .p2align 4,,7
.L5:
        jmp.L3
        .p2align 4,,7
.L4:
        movl-8(%ebp),%eax
        movl %eax,-4(%ebp)
        jmp.L2
        .p2align 4,,7
.L3:
.L1:
```

```
        leave
        ret
.Lfe1:
        .size  main,.Lfe1-main
        .ident"GCC:(GNU)egcs-2.91.66 19990314/Linux(egcs-1.1.2 release)"
```

在该汇编代码中有关的指令后加注释，将源程序中的操作和生成的汇编代码对应起来，以判断确实是用短路计算方式来完成布尔表达式计算的。

7.10 下面是一个C语言程序和在x86/Linux系统上编译（版本较低的GCC编译器，并且未使用优化）该程序得到的汇编代码（为便于理解，略去了和讨论本问题无关的部分，并改动了一个地方）。

(a) 为什么会出现一条指令前有多个标号的情况，如.L2和.L4，还有.L5、.L3和.L1？从控制流语句的中间代码结构加以解释。

(b) 每个函数都有这样的标号.L1，它可能的作用是什么，为什么本函数没有引用该标号的地方？（习题7.9的汇编代码也有这个现象。）

```
main(){
        long i,j;
        if(j)
                i++;
        else
                while(i)j++;
}
```

编译产生的汇编代码如下：

```
main:
        pushl  %ebp
        movl  %esp,%ebp
        subl  $8,%esp
        cmpl  $0,-8(%ebp)
        je  .L2
        incl  -4(%ebp)
        jmp.L3
.L2:
.L4:
        cmpl  $0,-4(%ebp)
        jne.L6
        jmp.L5
```

```
.L6:
    incl-8(%ebp)
    jmp.L4
.L5:
.L3:
.L1:
    leave
    ret
```

7.11 编一个程序,实现表7.2给出的控制流语句的语法制导定义。

7.12 用7.3节的翻译方案,把赋值语句A[x,y]:=z翻译成三地址代码(其中A是10×5的数组)。

*7.13 C语言的for语句有下列形式:

for(e_1 ; e_2 ; e_3) *stmt*

它和

e_1 ;
while(e_2) **do** **begin**
  stmt;
  e_3
end

有同样的含义。构造一个语法制导定义,把C语言风格的for语句翻译成三地址代码。

7.14 Pascal标准定义语句

for v := *initial* **to** *final* **do** *stmt*

和下面代码序列

begin
  t_1 := *initial*;
  t_2 := *final*;
  if $t_1 \leqslant t_2$ **then** **begin**
    v := t_1 ;
    stmt;
    while $v \neq t_2$ **do** **begin**
      v := *succ* (v);
      stmt
    end
  end
end

有同样的含义。

(a) 考虑下面的 Pascal 程序:

```
program forloop(input,output);
    var i,initial,final:integer;
    begin
        read(initial,final);
        for i:=initial to final do
            writeln(i)
    end.
```

当 *initial*=MAXINT-5 并且 *final*=MAXINT 时,该程序的行为是什么? 其中 MAXINT 是目标机器的最大整数。

*(b) 构造语法制导的定义,为 Pascal 的 for 语句产生正确的三地址代码。

7.15 语句

for i:=1 **step** 10-j **until** 10 * j **do** j:=j+1

可以有三种不同的语义定义。一种可能的含义是,步长表达式 10-j 和终值表达式 10 * j 都仅在循环前计算一次。这样,如果在循环前 j=5,那么该循环体执行 10 次。第二种语义完全不同,要求每次通过循环体时,都要执行步长表达式和终值表达式。例如,若在循环前 j=5,那么该循环将不会终止,C 语言的 for 语句属于这种情况,其表达式 e_2 和 e_3 是要重复计算的(见习题 7.13)。第三种含义由 ALGOL 这样的语言给出。当步长是负数时,该循环终止的测试是 i<10 * j,而不是 i>10 * j。分别写出这三种定义下的中间代码结构。

7.16 赋值语句 A[x,y]:=z(其中 A 是 10×5 的数组)的注释分析树见图 7.13。请根据 7.3.3 节的翻译方案,把图 7.13 中的属性值都补上(像图 7.10 那样),并且把每步归约产生的中间代码写在相应产生式的旁边。

7.17 C 语言和 Java 语言的数组声明和数组元素引用的语法形式同 7.3. 节讨论的不一样,例如 float A[10][20] 和 A[i+1][j-1],并且每一维的下界都是 0。若适应这种情况的赋值语句的文法如下:

$S \to L := E$

$E \to E+E \mid (E) \mid L$

$L \to L[E] \mid \mathbf{id}$

(a) 重新设计 7.3.2 节数组元素的地址计算公式,以方便编译器产生数组元素地址计算的中间代码。不要忘记每一维的下界都是 0。

(b) 重新设计数组元素地址计算的翻译方案。只需写出产生式 $L \to L[E] \mid \mathbf{id}$ 的翻译方案,但要能和 7.3.3 节中产生式 $S \to L := E$ 和 $E \to E+E \mid (E) \mid L$ 的翻译方案衔接。若翻译方案中引入新的函数调用,要解释这些函数的含义。

7.18 (a) 图 7.6 的翻译方案不能用于允许递归过程的情况(见 7.2.2 节中的解释)。对图

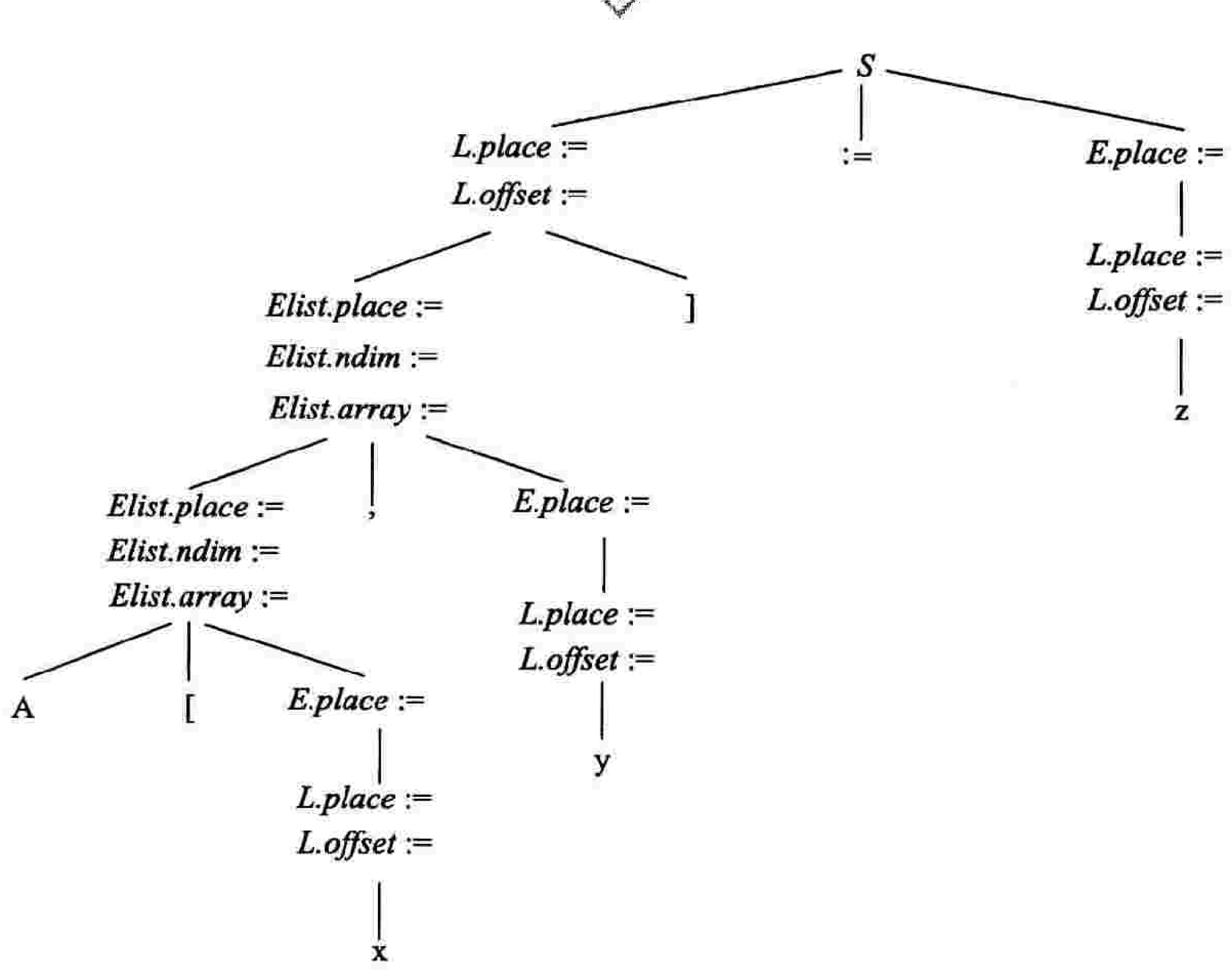


图 7.13 A[x, y]:= z 的注释分析树

7.6 的文法及其翻译方案作最少修改(不改变该文法定义的语言),使得修改后的翻译方案能够用于有直接递归调用的情况(可以修改或增加语义动作)。

(b) 对于允许过程嵌套的语言,结合图 7.6 和图 7.9 的翻译方案,像图 2.10 和图 2.11 那样给出函数 *lookup*(*name*)的描述。

第 8 章

编译器的最后一个阶段的工作是代码生成,它取源程序的中间表示作为输入,产生等价的目标程序作为输出,如图 8.1 所示。不管代码生成阶段的前面是否有代码优化阶段,本章提出的代码生成技术都是适用的。

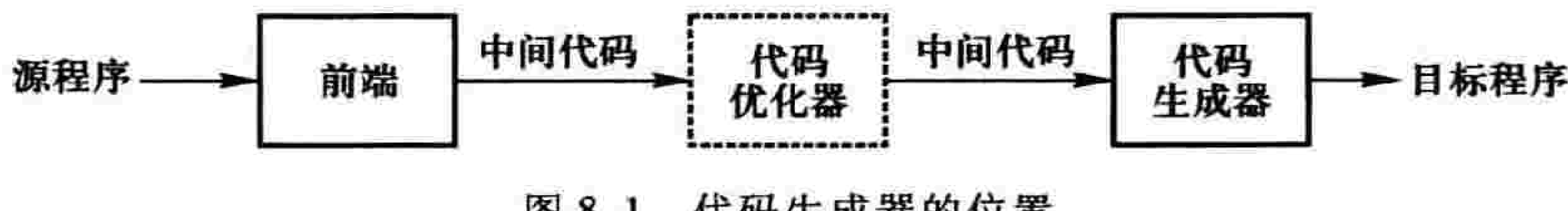


图 8.1　代码生成器的位置

对代码生成器的要求是严格的。首先,所生成的目标代码必须确保源程序的语义,并且是高质量的,高质量的含义是目标代码应该有效地利用目标机器的资源。其次,代码生成器本身应该高效地运行。此外,易于实现、测试和维护也是重要的设计目标。

从理论上讲,产生最优代码问题是不可判定的,在实践中,有很多技术能够产生令人满意的代码(虽不是最优的)。本章仅介绍一个简单的代码生成算法,以便对代码生成技术有一个大致的了解。

8.1　代码生成器设计中的问题

虽然代码生成器的具体细节依赖于目标机器和操作系统,但很多问题,如存储管理、指令选择、寄存器分配和指派、计算次序选择等是几乎所有的代码生成器都会碰到的问题。本节考察代码生成器设计中的一些公共问题,其中存储管理在第 6 章已经详细介绍。

代码生成器的输入包括由前端产生的中间表示和符号表信息,符号表信息用来决定中间表示中名字所代表的数据对象的运行地址。本章采用的中间表示是三地址代码。

8.1.1　目标程序

代码生成器的输出称为目标程序。像中间代码那样,目标程序的形式也有多种:绝对机器语

言程序、可重定位机器语言程序或者汇编语言程序。

可重定位机器语言程序是指代码装入内存的起始地址可以任意，代码中有一些重定位信息，以适应重定位的要求。绝对机器语言程序装入内存的起始地址是固定的。

产生绝对机器语言程序作为输出的好处是，目标程序可以放在内存的固定地方并且立即执行。因此程序可以被迅速地编译和执行。

产生可重定位机器语言程序（经常被称为目标模块）作为输出允许子程序分别地编译。通过一个连接装入程序，一组可重定位目标模块可以被连接在一起并被装入机器等待运行。虽然产生可重定位目标模块必须增加额外的开销来进行连接，但它带来的好处是灵活性。因为这种方式允许程序模块或子程序分别编译，允许从目标模块中调用其他先前编译好的程序模块。如果目标机器不能自动地管理重定位，编译器必须提供显式的重定位信息，供连接装入程序连接分别编译的程序模块。第 11 章将详细介绍目标模块中的重定位信息和连接信息。

产生汇编语言程序作为输出使得代码生成的过程变得容易，因为可以产生符号指令并可利用汇编器的宏机制来帮助生成代码，所付出的代价是代码生成后的汇编工序。

为了和前后几章的汇编程序实例基本一致，本章使用 CISC 风格的目标机器并以汇编代码作为目标语言。需要说明的是，只要地址可以根据符号表中的偏移和其他信息计算，那么产生名字的重定位地址或绝对地址，和产生它的符号地址一样容易。

8.1.2 指令选择

代码生成器必须把中间表示程序映射到能够在目标机器上执行的指令序列。这个映射的复杂性取决于这种中间表示的层次、指令集架构的性质和所生成代码的质量期望等因素。

如果中间表示的层次较高，则代码生成器可以依据代码模板把中间代码的每个语句翻译成一个机器指令序列。然而这样逐个语句产生目标代码的方式总是产生需要进一步优化的低劣代码。如果中间表示反映下面目标机器的低层细节，那么代码生成器可以利用这些信息生成较有效的代码序列。

目标机器指令集的性质决定了指令选择的难易程度。指令集的统一性和完备性是其中两个重要的因素。如果目标机器不能以统一的方式支持各种数据类型，那么每种例外的数据类型都需专门的处理。例如在某些机器上，浮点运算需要使用分离寄存器。

指令速度和机器方言（machine idiom）是另外两个重要的因素。如果不考虑目标程序的效率，指令选择是直截了当的。对每一类三地址语句，可以设计所生成的目标代码的框架。例如，形式为 x=y+z 的三地址语句，若 x，y 和 z 都是静态分配，那么它可以翻译成如下代码序列：

```
MOV    y,    R0     /* 把 y 装入寄存器 R0 */
ADD    z,    R0     /* z 加到 R0 上 */
MOV    R0,   x      /* 把 R0 存入 x 中 */
```

遗憾的是，这种逐条语句产生代码的方式常常得到质量低劣的代码。例如语句序列

a=b+c

d=a+e

将翻译成

```
MOV    b,    R0
ADD    c,    R0
MOV    R0,   a
MOV    a,    R0
ADD    e,    R0
MOV    R0,   d
```

显然,第四条指令是多余的。如果能够知道 a 以后不再使用的话,那么第三条也多余。

所生成的代码的质量通常取决于它的长度和执行速度。在大多数机器上,一个中间表示程序可以由许多不同的指令序列来实现,并且不同的实现之间的代价差别相当可观。因此对中间代码进行简单的翻译能产生正确的、但效率可能难以接受的目标代码。

例如,若目标机器有加 1 指令(INC),那么三地址语句 a=a+1 的高效实现是一条指令 INC a,而不是下面的指令序列:

```
MOV    a,    R0
ADD    #1,   R0
MOV    R0,   a
```

为设计好的代码序列,需要知道指令的代价。不幸的是精确的代价信息通常很难得到。决定哪个指令序列对给定的三地址构造是最优的,可能还要用到该构造出现的上下文知识。

8.1.3 寄存器分配

运算对象处于寄存器中的指令通常比运算对象处于内存的指令要短一些,执行也快一些。因此,充分利用寄存器对生成高质量的代码尤其重要。寄存器的使用可以分成两个子问题:

(1) 寄存器分配:为程序各点选择驻留在寄存器中的一组变量。

(2) 寄存器指派:为驻留寄存器的变量选择保存其值的具体寄存器。

寻找最优的寄存器指派方案是困难的,即便是单寄存器机器也是这样。这个问题是 NP 完全的。这个问题还会进一步复杂,因为目标机器的硬件和/或操作系统可能要求寄存器的使用遵守一些约定。

本章介绍的简单代码生成算法将寄存器分配和指派合在一起,统称为寄存器分配。

8.1.4 计算次序选择

计算的执行次序会影响目标代码的效率。例如,对某个表达式,一种计算次序可能会比其他

次序占用较少的寄存器来保存中间结果。选择最佳计算次序也是一个 NP 完全问题。本章只讨论按照中间代码生成器所生成的三地址语句的次序来生成目标代码,但是在习题中将给出一些体现计算次序选择的例子。第 10 章将研究在一个时钟周期内能执行几条指令的流水线机器上的代码调度。

8.2 目标语言

熟悉目标机器和它的指令集是设计一个良好代码生成器的先决条件。遗憾的是,在代码生成的一般性讨论中,不能对目标机器描述到足够详细的程度,因而难以为该机器完整的机器语言产生高效的代码。本章选择一种简化的多寄存器机器作为目标计算机,所提出的一些技术可用于其他许多类机器。

8.2.1 目标机器的指令集

该目标机器是字节寻址机器,4 个字节组成 1 个字。有 n 个通用寄存器 R0,R1,…,Rn-1。它有形式为

op 源,目的

的二地址指令,其中 *op* 是操作码,源和目的都是数据域。该机器有如下的操作码:

MOV /* 源传到目的 */
ADD /* 源加到目的 */
SUB /* 目的减去源 */

其他的指令等用到时再介绍。

由于源和目的这两个域没有长到足以保存内存地址,所以它们的某些位用来指明指令的下一个字或下两个字包含运算对象或地址。于是,一条指令的源和目的由带地址模式的寄存器和内存单元的组合来指明。在下面的描述中,*contents*(*a*)表示由 *a* 代表的寄存器或内存单元的内容。

地址模式和它们的汇编语言形式及附加代价见表 8.1。

表 8.1 地址模式和汇编语言形式及附加代价

地址模式	汇编语言形式	地址	附加代价
绝对地址	M	M	1
寄存器	R	R	0
变址	*c*(R)	*c*+*contents*(R)	1

续表

地址模式	汇编语言形式	地址	附加代价
间接寄存器	*R	*contents*(R)	0
间接变址	*c(R)	*contents*(c+*contents*(R))	1

8.2.2 节将解释附加代价。

当用作源或目的时，内存单元 M 和寄存器 R 都代表本身。例如，指令

MOV R0,M

把寄存器 R0 的内容存入内存单元 M。

寄存器 R 的值加上偏移 c 表示的地址写成 c(R)。例如，指令

MOV 4(R0),M

把值

contents(4+*contents*(R0))

存入内存单元 M。

上述两种模式的间接形式由加前缀 * 来指明。例如，指令

MOV *4(R0),M

把值

contents(*contents*(4+*contents*(R0)))

存入内存单元 M。

最后一种地址模式允许源是常数：

模式	形式	常数	附加代价
立即数	#c	c	1

例如，指令

MOV #1,R0

把 1 保存到寄存器 R0 中。

8.2.2 指令代价

在这个简化的机器上，指令的代价是

1+源地址模式的附加代价+目的地址模式的附加代价

其中附加代价见表 8.1 的最后一列。这个代价对应指令以字计算的长度。寄存器地址模式的代价是 0，而那些含内存单元或常数的地址模式的代价是 1，因为这样的运算对象必须和指令存放在一起。

如果空间是至关重要的，应该使指令的长度尽可能短。这样做有一个额外的重要好处，对大

多数机器和大多数指令来说,从内存取指令的时间会超过执行指令的时间,因而,极小化指令的长度也使得指令的执行时间趋于最小。下面是几个例子。

(1) 指令 MOV R0,R1 把寄存器 R0 的内容复写到寄存器 R1,这条指令的代价是 1,因为它仅占 1 个字的内存。

(2) 指令 MOV R5,M 把寄存器 R5 的内容复写到内存单元 M。这条指令的代价是 2,因为内存单元的地址 M 存于指令的第二个字中。

(3) 指令 ADD#1,R3 把常数 1 加到寄存器 R3 的内容上。这条指令的代价是 2,因为常数 1 出现在指令的第二个字中。

(4) 指令 SUB 4(R0), * 12(R1)把值

contents(*contents*(12+*contents*(R1)))−*contents*(4+*contents*(R0))

存入目的地址 * 12(R1)。这条指令的代价是 3,因为常量 4 和 12 存于指令的第二个和第三个字中。

为这种机器产生代码的困难可以从下面的例子略知一二。考虑为形式是 a=b+c 的三地址语句产生代码,其中 a、b 和 c 都是简单变量,它们分配在静态数据区,直接用它们的名字表示相应的单元。这个语句可以由许多不同的指令序列来实现,例如:

```
(1)        MOV b,R0
           ADD c,R0        代价=6
           MOV R0,a
(2)        MOV b,a
           ADD c,a         代价=6
```

若 R0,R1 和 R2 分别含 a,b 和 c 的地址,则可以使用:

```
(3)        MOV *R1, *R0
           ADD *R2, *R0    代价=2
```

若 R1 和 R2 分别含 b 和 c 的值,并且 b 的值在这个赋值后不再需要,则可以使用:

```
(4)        ADD R2,R1
           MOV R1,a        代价=3
```

可以看出,要想为这种机器产生好的代码,必须有效地使用它的寻址能力。可能的话,尽量把名字的左值或右值保存在寄存器中,以便在不久的将来使用。

例 8.1 图 8.2 是 C 语言赋值表达式 a[i]=b+1 的中间代码树,其中 a[i],i 和 b 都是 long 类型,并且 a 和 i 都是动态栈式分配的,b 是静态分配的。为简单起见,假定机器是按字编址,并且 long 类型的值占一个字。和上一章的中间代码不同的是,图 8.2 形式的中间代码是一种层次较低的中间代码,它体现了很多和机器有关的特征,如动态栈式分配变量的地址描述和左右值的区别,因此它向代码生成提供了丰富的信息。

图 8.2 中结点标记的解释如下:

(1) reg_{sp}表示栈的基址寄存器 SP;

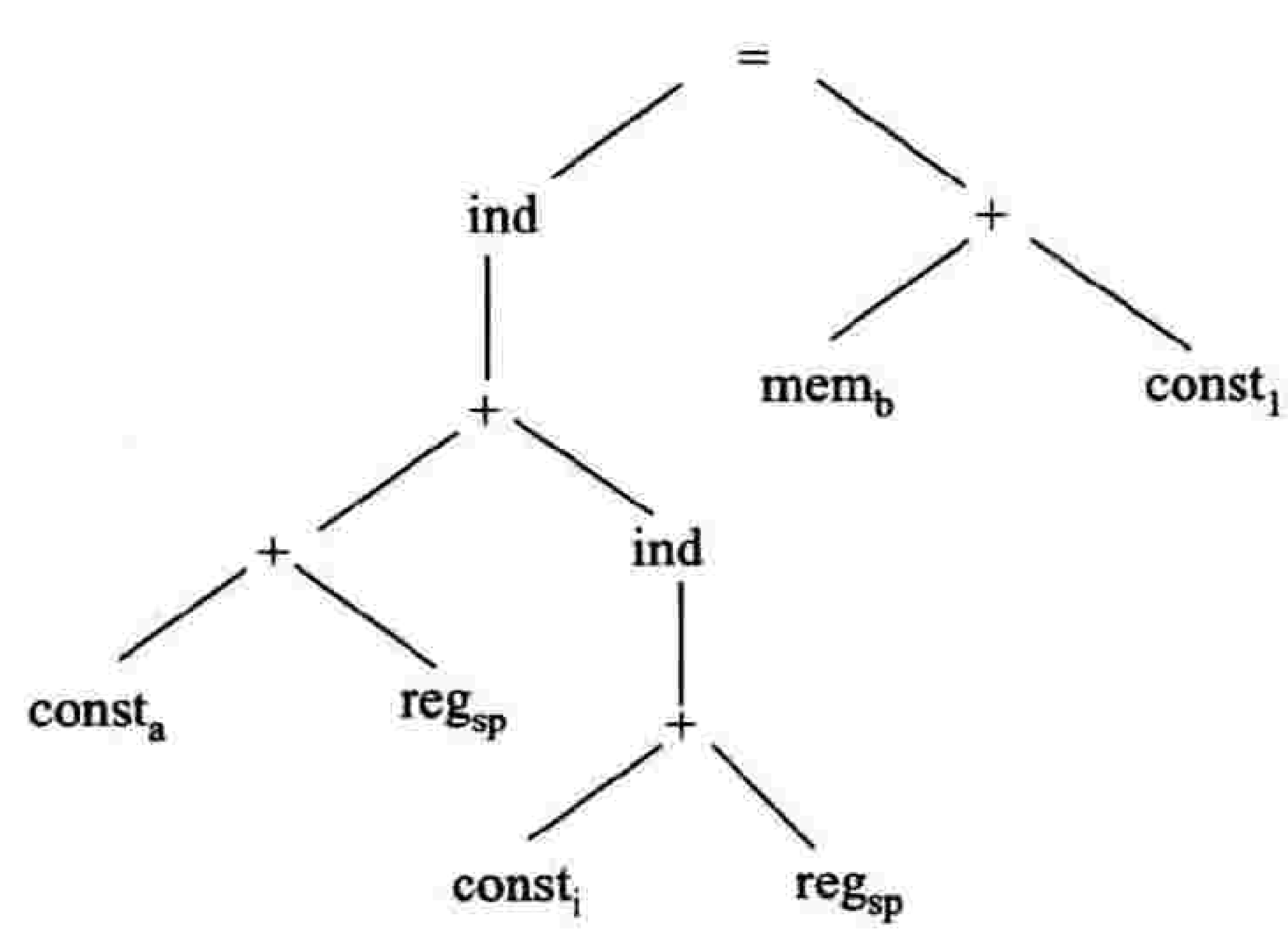


图 8.2　赋值表达式 a[i]=b+1 的中间代码树

(2) const 表示常数,其中 $const_1$ 表示常数 1,$const_a$ 和 $const_i$ 分别表示 a 和 i 在活动记录中的相对地址,它们是常数;

(3) mem 表示内存地址,mem_b 表示分配给 b 的内存地址;

(4) ind 表示它的子树是一个左值表达式,当需要左值时,直接用 ind 下面的子树,当需要右值时,把 ind 下面的子树作为地址,取该地址的内容,因此 ind 又可表示间接访问的意思。

有了这个解释,就不难理解表达式树的含义了。$mem_b+const_1$ 表示 b+1,$const_i+reg_{sp}$表示 i 的地址,$const_a+reg_{sp}$表示 a 的起始地址。($const_a+reg_{sp}$)+ind($const_i+reg_{sp}$)表示 a[i]的地址,注意这是一个左值(a 的起始地址)和一个右值(i 的值)相加,得到的是一个左值。ind(($const_a+reg_{sp}$)+ind($const_i+reg_{sp}$))在赋值号的左边,因此生成代码时直接用最外层括号中的左值表达式。

用本节介绍的各种指令和一种叫做树重写的指令选择算法(本书不介绍),可为该中间代码树生成如下的代码序列:

```
MOV #a,R0
ADD SP,R0
ADD i(SP),R0
MOV b,R1
INC R1
MOV R1, *R0
```

□

8.3 基本块和流图

三地址语句序列的一种图形表示叫做流图。流图的结点代表一个顺序计算的三地址语句序列,边代表控制流。即使代码生成算法不明显构造流图,流图对于理解代码生成算法也是有用的。下一章将充分利用流图作为从中间代码收集信息的媒介。

8.3.1 基本块

基本块是一个连续的三地址语句序列,控制流从它的开始进入,并从它的末尾离开,没有停止或分支的可能性(末尾除外)。下面三地址语句序列形成一个基本块:

$$
\begin{array}{ll}
t_1 = a * a & t_4 = t_1 + t_3 \\
t_2 = a * b & t_5 = b * b \\
t_3 = 2 * t_2 & t_6 = t_4 + t_5
\end{array}
\tag{8.1}
$$

下面的算法可用于把三地址语句序列分成基本块。

算法 8.1 划分基本块。

输入 三地址语句序列。

输出 基本块列表,每个三地址语句仅在一个基本块中。

方法 (1) 首先确定所有的**入口语句**(基本块的第一个语句)。规则如下:

(a) 该中间代码的第一个语句是入口语句。

(b) 任何作为条件转移或无条件转移目标的语句是入口语句。

(c) 紧跟在条件转移或无条件转移之后的语句是入口语句。

(2) 对于每个入口语句,它所在的基本块由从它开始一直到程序结束为止或下一个入口语句为止(但不含该入口语句)的所有语句组成。 □

例 8.2 考虑图 8.3 的源代码段,它计算两个长度为 20 的向量 ***a*** 和 ***b*** 的点积。在目标机器上完成这个计算的三地址语句序列见图 8.4。

```
prod=0;
i=1;
do{
    prod=prod+a[i] * b[i];
    i=i+1;
}while(i<=20);
```

图 8.3 计算点积的程序

(1) prod=0		(7) $t_5 = t_2 * t_4$
(2) i=1		(8) $t_6 = prod + t_5$
(3) $t_1 = 4 * i$		(9) $prod = t_6$
(4) $t_2 = a[t_1]$	/* 计算 a[i] */	(10) $t_7 = i+1$
(5) $t_3 = 4 * i$		(11) $i = t_7$
(6) $t_4 = b[t_3]$	/* 计算 b[i] */	(12) if i<=20 goto(3)

图 8.4 计算点积的三地址代码

把算法 8.1 应用到图 8.4 的三地址代码可知,语句(1)和(3)都是入口语句。语句(3)为入口语句的原因是最后一个语句可以转到它。这样,语句(1)和(2)构成一个基本块,其余的语句

形成一个基本块。 □

8.3.2 基本块的优化

三地址语句 x=y+z 被称为**引用** y 和 z 的值并对 x **定值**。一个名字的值在基本块的某点之后还要引用的话(包括在后继基本块的引用),则称这个名字在该点是**活跃**的。

两个基本块是**等价**的,若在它们的出口点有同样的活跃变量集合,并且在这两个出口点,对该集合中的任何一个活跃变量,代表该变量值的两个表达式可证明为相等。

有很多等价变换可用于基本块,这些变换改进基本块代码的质量,称之为**局部优化**。这里介绍两类可用于基本块的局部优化,它们是保结构变换和代数变换。下一章介绍的一些全局优化技术,如删除公共子表达式,也可用在局部优化中。作为简单介绍,假定基本块没有数组、指针和过程调用,并且不同名字代表不同变量,即没有别名问题。

先介绍三种保结构变换。

1. 删除局部公共子表达式

考虑基本块:

```
a = b+c
b = a-d
c = b+c
d = a-d
```

第二个语句和第四个语句计算同样的表达式,即 b+c-d,因此该基本块可以变换成等价的基本块:

```
a = b+c
b = a-d
c = b+c
d = b
```

虽然第一个语句和第三个语句有同样的 b+c 出现在右部,但由于第二个语句给 b 定值,使得第一和第三个语句引用的 b 可能有不同的值,所以两个 b+c 的值可能不同。

2. 删除死代码

语句 x=y+z 给 x 定值,若 x 此后不再被引用,则称 x 为死变量。删除这样的语句是基本块的一种等价变换。

3. 交换相邻的独立语句

如果基本块有两个相邻的语句:

$t_1 = b+c$

$t_2 = x+y$

则交换这两个语句不会影响基本块所计算的表达式。

有许多代数变换可用来对基本块实施等价变换,其中有价值的是那些可以简化表达式或用较快运算代替较慢运算的变换。例如,像

x = x+0

或

x = x * 1

这样的语句可以从基本块删除,这是基本块的等价变换。语句

x = y * * 2

的指数运算通常需要用函数调用实现。使用代数变换,该语句可以由快速、等价的语句

x = y * y

代替。

8.3.3 流图

可以把控制流信息加到基本块集合,形成一个有向图,用以表示程序,这样的有向图称为**流图**。流图的结点是基本块(简称块),有一个特殊的结点称为**初始结点**,它的入口语句是程序的第一个语句。对两个基本块 B_1 和 B_2,如果在程序的某个执行序列中 B_2 紧跟 B_1,那么 B_1 到 B_2 有一条有向边。也就是,如果:

(1) B_1 的最后一个语句是条件转移或无条件转移到 B_2 的第一个语句,或者

(2) 按程序正文的次序 B_2 紧跟 B_1,并且 B_1 不是结束于无条件转移,那么 B_1 到 B_2 有一条有向边。称 B_1 是 B_2 的**前驱**,B_2 是 B_1 的**后继**。

例 8.3 图 8.4 程序的流图见图 8.5。B_1 是初始结点。注意,最后一个语句原来是条件转移到语句(3),现已由转到 B_2 开始的等价语句代替。 □

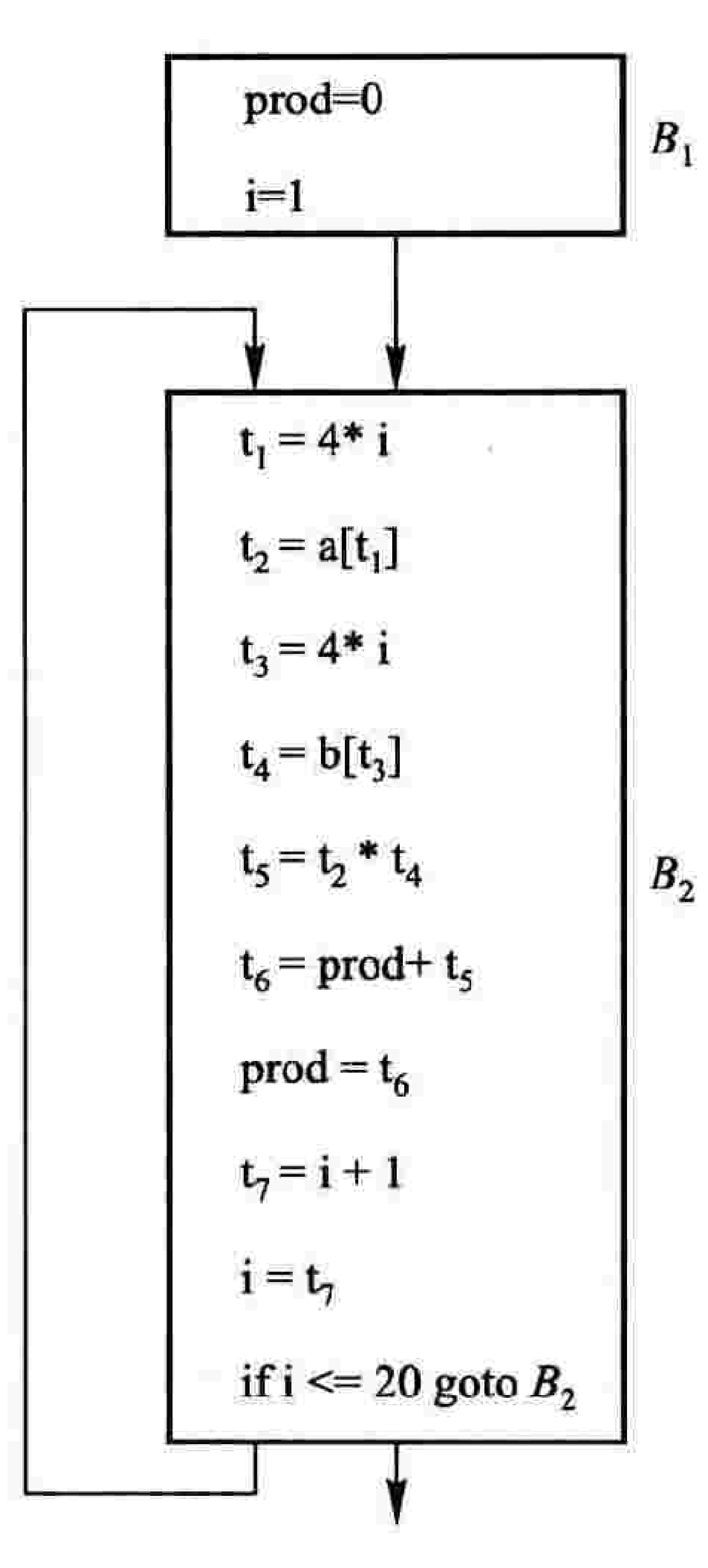


图 8.5 图 8.4 程序的流图

在流图中,什么是循环?怎样找出所有的循环?大多数情况下,这些问题是容易回答的。例如,图 8.5 存在一个循环,它由块 B_2 组成。对更一般的情况需等下一章详细讨论之后才能作答。目前,只要知道循环是流图中满足下面条件的一组结点就行了:

(1) 该组中所有结点是**强连通**的,即从循环中任一个结点到另一个结点都有一条路径,路径上的所有结点都在这组结点中;

(2) 这组结点有唯一的**入口**结点,从循环外的结点到达循环中任一个结点的唯一方式是首先通过入口。

不包含其他循环的循环叫做**内循环**。

8.3.4 下次引用信息

下面介绍如何收集基本块中名字的下次引用(next-use)信息,8.4节的简单代码生成器需要该信息来完成寄存器分配。这是因为,如果知道了其值存于一个寄存器的某个名字没有下次引用,那么该寄存器可用来存放其他名字的值。

在一个三地址语句中,名字的**引用**定义如下:假定三地址语句 i 对 x 定值,如果语句 j 用 x 作为运算对象,控制可以从 i 流到 j,并且在这条路径上没有对 x 的其他定值,那么称 j 引用 x 在 i 的定值。

目前需要为基本块中每个三地址语句 $x=y\ op\ z$ 决定 x、y 和 z 在该块中的下次引用信息。暂时不考虑它们在基本块外的被引用情况,若需要这种信息,可用第9章的活跃变量分析技术获得。

对每个基本块,收集下次引用信息的算法从最后一个语句反向扫描到第一个语句。在反向扫描一个基本块前,根据下一章的活跃变量分析,在符号表中把在出口点活跃的变量都置以活跃标记,并置以没有下次引用(指在本块中没有下次引用)。而对于在出口点不活跃的变量,都置不活跃也没有下次引用。

假如反向扫描到达三地址语句 i:$x=y\ op\ z$,执行下面几步:

(1) 从符号表中找出 x、y 和 z 的下次引用信息和活跃标记,并把它们加到语句 i 上;

(2) 在符号表中把 x 置成不活跃和没有下次引用;

(3) 在符号表中,置 y 和 z 活跃,并且置它们的下次引用信息为 i。

注意,(2)和(3)的次序不能颠倒,因为 x 可能就是 y 或 z。

如果三地址语句 i 是 $x=y$ 或 $x=op\ y$ 的形式,其步骤同上,但忽略 z。

利用下次引用信息,也可以压缩临时变量需要的空间。一般地说,如果两个临时变量不同时活跃(生存期不重叠)的话,可以把它们压缩在同一单元中。因为大多数临时变量都在同一块中定义和引用,因而下次引用信息就可用来紧缩临时变量占用空间。对于穿越多个基本块的引用问题,放在下一章数据流分析时讨论。

临时变量存储单元的分配可以这样进行:依次检查临时变量区域的单元,找到第一个不含活跃临时变量的单元,把它分配给待分配的临时变量。如果没有这样的单元,则在活动记录的临时变量区域加一个单元。在许多情况下,临时变量可以压缩到只使用寄存器而不需要内存单元,见8.4节的讨论。

例如,基本块(8.1)有6个临时变量,可压缩为只需要两个临时变量,它们是 t_1 和 t_2。

$t_1 = a * a$

$t_2 = a * b$

$t_2 = 2 * t_2$

$t_1 = t_1 + t_2$

$t_2 = b * b$

$t_1 = t_1 + t_2$

8.4 一个简单的代码生成器

本节的代码生成算法为一个基本块产生目标代码。它依次考虑该块中每个语句,根据寄存器当前的使用情况(什么值在寄存器中,在哪个寄存器中),为其产生代码,并根据产生的代码修改寄存器的使用情况。这样,它可以避免产生不必要的装入寄存器和存回内存单元的指令。

代码生成期间的一个基本问题是怎样最佳地使用寄存器。寄存器的四种主要用途如下。

(1) 在大多数机器架构(machine architecture)上,为完成某种运算,它的部分或所有的运算对象都必须在寄存器中。

(2) 寄存器优选作为临时单元,即在计算一个大表达式时,作为保存子表达式值的地方,或者更广一点,作为保存一个基本块中还要使用的值的地方。

(3) 寄存器用来保存在一个基本块中计算而在另一个基本块中使用的全局值。例如在循环体的一次执行中,循环控制变量在某处被修改,可能在多处被引用。

(4) 寄存器经常用于运行时存储空间的管理。例如作为栈顶指针和活动记录基地址指针。

本节的算法假定有部分寄存器可用于保存基本块中使用的值,假定基本块已完成 8.3 节所提到的一些优化变换,还假定三地址语句中出现的每种运算都有对应的目标机器指令。

另外本算法将计算结果尽量长时间地保留在寄存器中,只有在下面两种情况下才把它存入内存:

(1) 寄存器要用于其他计算;

(2) 到了基本块的出口点。

在基本块的出口点,以后还要引用的值都必须保存起来。原因是,本算法的寄存器分配以基本块为单位,不考虑用寄存器保留值来穿越基本块。因为一个基本块可能有若干个后继,也可能有若干个前驱,因而没有经过额外的努力,就认为一个基本块引用的某个数据在入口点一定处于某个寄存器中是不稳妥的。

8.4.1 寄存器描述和地址描述

对三地址语句 a=b+c,如果寄存器 Ri 含 b,Rj 含 c,且 b 在此语句后不再活跃,那么可以为它产生代价为 1 的代码 ADD Rj,Ri,结果 a 在 Ri 中。

如果 Ri 含 b,但 c 在内存单元,b 仍然不再活跃,那么可以产生代价为 2 的代码

```
ADD c,Ri
```

或代价为 3 的代码序列

```
MOV c,Rj
ADD Rj,Ri
```

如果 c 的值以后还要用的话,则该代码序列比较有吸引力,因为可以从寄存器 Rj 中取 c 的值。

还有很多的情况可考虑,这取决于 b 和 c 的值当前在什么地方以及 b 的值以后是否还要用。还必须考虑 b 和 c 中一个或两个都是常数的情况。如果+运算是可交换的话,需考虑的情况还会增加。所以,可以看出,代码生成包含了对大量情况的考察,哪种情况占优势依赖于三地址语句出现的上下文。

从上面的例子可以看出,在代码生成过程中,需要跟踪寄存器的内容和名字的地址。本节的代码生成算法使用寄存器和名字的描述来跟踪寄存器的内容和名字的地址。

(1) 寄存器描述:它记住每个寄存器当前存的是什么。初始时寄存器描述表明供基本块使用的寄存器都空闲。随着对基本块的代码生成逐步前进,在任何一点,每个寄存器保存若干个(包括零个)名字的值。寄存器的这些信息可以单独用一张寄存器表来描述。

(2) 名字的地址描述:它记住每个名字的当前值可以在哪个场所找到。这个场所可以是寄存器、栈单元、内存地址,甚至是它们的某个集合,因为复写时值仍然留在原来的地方。这些信息可以存于符号表中。

8.4.2 代码生成算法

该代码生成算法以构成一个基本块的三地址语句序列作为输入,对每个三地址语句 x = y *op* z 完成下列动作:

(1) 调用函数 *getReg*,决定存放 y *op* z 计算结果的场所 L。L 通常是寄存器,但也可能是内存单元。8.4.3 节将简要描述 *getReg* 的算法。

(2) 查看 y 的地址描述,确定存放 y 当前值的一个场所 y′。如果 y 的值既在内存单元又在寄存器中,优先选择寄存器作为 y′,特别是 y 的值所在的寄存器正好是 L 的时候。如果 y 的值还不在 L 中,则产生指令 MOV y′,L,把 y 的值复写到 L 中。

(3) 产生指令 *op* z′,L,其中 z′是存放 z 当前值的场所之一。同上面一样,如果 z 的值既在寄存器又在内存单元中,就优先取前者。修改 x 的地址描述,以表示 x 在场所 L,如果 L 是寄存器,修改它的描述,以表示它含 x 的值。

(4) 如果 y 和/或 z 的值在块内不再引用,在块的出口点也不活跃,并且还在寄存器中,那么修改寄存器描述,以表示在执行了 x = y *op* z 以后,这些寄存器分别不再含 y 和/或 z 的值。

如果三地址语句有一元算符,步骤同上面的类似,在此略去。一个重要的特例是复写语句 x = y。如果 y 在寄存器中,只要改变寄存器和地址描述,记住 x 的值当前只能在存 y 值的寄存器中找到;进一步,如果 y 在块内不再引用,并且在块出口点也不活跃,那么这个寄存器不再保存 y 的值。如果 y 的值仅在内存中,原则上,可以记住 x 的值在 y 的内存单元,但是这样会使算法复杂,

因为以后若要改变 y 的值就必须先保存 x 的值。所以,如果 y 在内存中,可用 *getReg* 来找到一个存放 y 的寄存器,并记住此寄存器也是保存 x 的场所。另一种办法是产生指令 MOV y,x。尤其是 x 在块中不再引用时,这样做比较好。值得注意的是,如果应用下一章的各种优化,尤其是复写传播算法,大多数复写指令可以删去。

一旦处理完一个基本块的所有三地址语句,在基本块出口点,用 MOV 指令把那些值尚不在它们内存单元的活跃名字的值都存入它们的内存单元。为完成这件事,用寄存器描述来决定什么名字的值仍在寄存器中,再用地址描述来决定其中哪些名字的值不在它们的内存单元,最后用活跃变量信息来决定其中需要存储的名字。如果基本块之间的数据流分析没有计算活跃变量信息,只好认为所有用户定义的名字在基本块末尾都是活跃的。

8.4.3 寄存器选择函数

函数 *getReg* 以三地址语句 x=y *op* z 为参数,返回所选择的保存 x 值的场所 L。本节算法的很多努力都消耗在实现这个函数上,以产生对 L 的较好选择。本小节讨论基于下次引用信息的一个简单易行的办法。

(1) 如果名字 y 在寄存器中,此寄存器不含其他名字的值(注意,x=y 这样的复写语句会使得寄存器可能同时保存多个变量的值),并且在执行 x=y *op* z 后 y 在块内不再引用,那么返回当前保存 y 值的这个寄存器作为 L。

(2) 当(1)失败时,返回一个空闲寄存器(如果有的话)。

(3) 当(2)不能成功时,如果 x 在块中还会被引用,或者 *op* 是必须用寄存器的运算,如变址操作,那么找一个已被占用的寄存器 R。如果 R 保存的是变量 w 的值,并且还没有保存到 w 的内存单元,则生成 MOV R,w 指令,把 R 的值存入 w,并修改 w 的地址描述,返回 R。如果 R 保存着几个变量的值,那么对于每个需要存储的变量都产生 MOV 指令。怎样恰当地选择这个寄存器呢?可优先选择其数据在将来最晚使用的寄存器,或者其数据同时在内存的寄存器。很难证明哪种选择方法是最佳的。

(4) 如果 x 在块内不再引用,或者找不到适当的被占用寄存器,可选择 x 的内存单元作为 L。

更复杂的 *getReg* 函数在决定存放 x 值的寄存器时要考虑 x 随后的使用情况和算符 *op* 的交换性等。

例 8.4 赋值语句 d=(a-b)+(a-c)+(a-c)可以翻译成下面的三地址语句序列:

$t_1 = a - b$

$t_2 = a - c$

$t_3 = t_1 + t_2$

$d = t_3 + t_2$

假定只有 d 在基本块出口活跃。本节的代码生成算法为该三地址语句序列产生如表 8.2 的代码序列。表中给出代码生成过程中相关的寄存器描述和地址描述,但是忽略了 a,b 和 c 的值总是

在内存中这样一个事实。

getReg 的第一次调用返回 R0 作为保存 t_1 的场所。因为 a 不在 R0,因此产生 MOV a,R0 和 SUB b,R0 的指令。修改寄存器描述表示 R0 含 t_1。

代码生成以这种方式前进,直到最后一个三地址语句处理完。注意,这时 R1 为空,因为 t_2 不再引用。最后在基本块的出口点产生 MOV R0,d,存储活跃变量 d。

表 8.2 生成的代码的代价是 12。可以把它缩减到 11,在第一条指令后立即产生指令 MOV R0,R1,删去指令 MOV a,R1,但是这需要更复杂的代码生成算法。代价能减小的原因是从 R1 取到 R0 比从内存取到 R0 要低廉一些。 □

表 8.2 目标代码序列

语句	生成的代码	寄存器描述	名字地址描述
		寄存器空闲	
$t_1=a-b$	MOV a,R0 SUB b,R0	R0 含 t_1	t_1 在 R0 中
$t_2=a-c$	MOV a,R1 SUB c,R1	R0 含 t_1 R1 含 t_2	t_1 在 R0 中 t_2 在 R1 中
$t_3=t_1+t_2$	ADD R1,R0	R0 含 t_3 R1 含 t_2	t_3 在 R0 中 t_2 在 R1 中
$d=t_3+t_2$	ADD R1,R0	R0 含 d	d 在 R0 中
	MOV R0,d		d 在 R0 和内存中

8.4.4 为变址和指针语句产生代码

变址与指针运算的三地址语句的处理和二元算符的处理相同。表 8.3 给出了为变址语句 a=b[i]和 b[i]=a 产生的代码序列,假定 b 是静态分配的。

i 当前所在场所决定了代码序列。表 8.3 给出三种情况,分别是 i 在寄存器 Ri 中、i 在内存单元 Mi 中、i 在栈中。对于后者,偏移为 Si,且 i 所在活动记录的基地址寄存器是 Rs。寄存器 R 是调用函数 *getReg* 时返回的寄存器,对于赋值 a=b[i],如果 a 在块中有下次引用,并且寄存器 R 是可用的,宁愿把 a 留在寄存器 R 中。

表8.3 变址语句的代码序列

语句	i在寄存器Ri中		i在内存Mi中		i在栈中	
	代码	代价	代码	代价	代码	代价
a=b[i]	MOV b(Ri),R	2	MOV Mi,R MOV b(R),R	4	MOV Si(Rs),R MOV b(R),R	4
b[i]=a	MOV a,b(Ri)	3	MOV Mi,R MOV a,b(R)	5	MOV Si(Rs),R MOV a,b(R)	5

表8.4给出了为指针赋值a= *p和 *p=a产生的代码序列,它由p当前所在场所决定。同上面一样,这里也给出了三种情况。寄存器R也是调用函数 *getReg* 返回的寄存器。

表8.4 指针语句的代码序列

语句	p在寄存器Rp中		p在内存Mp中		p在栈中	
	代码	代价	代码	代价	代码	代价
a= *p	MOV *Rp,a	2	MOV Mp,R MOV *R,R	3	MOV Sp(Rs),R MOV *R,R	3
*p=a	MOV a, *Rp	2	MOV Mp,R MOV a, *R	4	MOV a, *Sp(Rs)	3

8.4.5 条件语句

机器实现条件转移有两种方式。一种方式是根据寄存器的值是否为下面6个情况之一进行分支:负、零、正、非负、非零和非正。在这样的机器上,if x<y goto L的实现通常是把x减y的值存入寄存器R,如果R的值为负,则跳转到L。

第二种方式是用条件码来表示计算的结果或装入寄存器的值是负、零还是正。这种方法适用于很多机器。通常,比较指令(本章机器模型上是CMP)有这样的性质,它设置条件码标志而并不真正计算值。例如,若x>y,那么CMP x,y置条件码标志为正。条件转移指令根据指定的条件<,==,>,<=,!=或>=是否满足来决定是否转移。指令CJ<=L用来表示如果条件码标志是负或者零则跳转到L。例如,if x<y goto L可以由

```
CMP     x,y
CJ<     L
```

来实现。

产生代码时,记住条件码标志的描述是有用的。这个描述告知设置当前条件码标志的名字

或比较的名字对。于是可以用

```
MOV     y,    R0
ADD     z,    R0
MOV     R0,   x
CJ<     L
```

来实现

```
x = y+z
if x<0 goto L
```

因为根据条件码标志的描述可以知道在 ADD z,R0 之后,条件码标志是由 x 设置的。

习 题 8

8.1 为下列 C 语句产生 8.2 节目标机器的代码,假定所有的变量都是静态的,并假定有三个寄存器可用于保存计算结果。

(a) x = 1

(b) x = y

(c) x = x+1

(d) x = a+b * c

(e) x = a/(b+c) - d * (e+f)

8.2 重复习题 8.1,假定所有的变量都是自动变量(分配在栈上)。

8.3 为下列 C 语句产生 8.2 节目标机器的代码,假定所有的变量都是静态的,并假定有 3 个寄存器可用于保存计算结果。

(a) x = a[i]+1

(b) a[i] = b[c[i]]

(c) a[i] = a[i]+b[j]

8.4 使用 8.4 节的算法重做习题 8.1。

* 8.5 在早先的 SPARC/SunOS 系统上,经某编译器编译后,下面程序的运行结果是 120。但是如果把第 10 行的 abs(1)改成 1 的话,则程序结果是 1。试分析为什么会有这样不同的结果。

```
int fact(){
    static int i=5;
    if(i==0){
        return(1);
    }
    else{
        i=i-1;
```

```
        return((i+abs(1)) * fact());
    }
}
main(){
        printf("factor of 5=%d\n",fact());
}
```

*8.6 一个C语言程序如下:

```
main(){
    long i;
    i=0;
    printf("%ld\n",(++i)+(++i)+(++i));
}
```

该程序在x86/Linux系统上,编译后的运行结果是7(编译器版本是GCC:(GNU)egcs-2.91.66 19990314/Linux(egcs-1.1.2 release)),而在早先的SPARC/SunOS系统上编译后的运行结果是6。试分析运行结果不同的原因。

*8.7 一个C语言程序如下,运行时输出105。

```
main(){
    long i;
    i=10;
    i=(i+5)+(i=i*5);
    printf("%d\n",i);
}
```

该程序在x86/Linux系统上编译后生成的汇编代码如下(编译器版本见汇编代码最后一行),从生成的汇编代码看出,表达式(i+5)+(i=i*5)的右子树先计算,你能猜测出有关的代码生成策略吗?

```
        .file"expression.c"
        .version"01.01"
gcc2_compiled.:
.section.rodata
.LC0:
        .string"%d\n"
.text
        .align 4
.globl main
        .type main,@function
```

```
main:
    pushl %ebp
    movl %esp,%ebp
    subl  $4,%esp
    movl  $10,-4(%ebp)
    movl -4(%ebp),%edx
    movl %edx,%eax
    sall  $2,%eax
    addl %edx,%eax
    movl %eax,%edx
    movl %edx,-4(%ebp)
    leal 5(%edx),%eax
    addl %eax,-4(%ebp)
    movl -4(%ebp),%eax
    pushl %eax
    pushl  $.LC0
    call printf
    addl  $8,%esp
.L1:
    leave
    ret
.Lfe1:
    .size main,.Lfe1-main
    .ident"GCC:(GNU)egcs-2.91.66 19990314/Linux(egcs-1.1.2 release)"
```

8.8 一个C语言程序如下:

```
main(){
    int i;
    i=50;
    switch(i*i){
        case 10:i=10;break;
        case 80:i=80;break;
        case 50:i=50;break;
        case 70:i=70;break;
        case 20:i=20;break;
        default:i=40;
```

```
    }
    switch(i * i) {
        case 7:i=7;break;
        case 1:i=1;break;
        case 6:i=6;break;
        case 9:i=9;break;
        case 5:i=5;break;
        case 10:i=10;break;
        case 2:i=2;break;
        default:i=40;
    }
}
```

它在 x86/Linux 系统上编译后生成的汇编代码如下(编译器版本见汇编代码最后一行),请根据所生成的汇编代码写出程序中两个 switch 语句的目标代码结构的特点。

```
    .file "switch.c"
    .version"01.01"
gcc2_compiled.:
.text
    .align 4
.globl main
    .type main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    subl $4,%esp
    movl $50,-4(%ebp)
    movl -4(%ebp),%eax
    imull -4(%ebp),%eax
    cmpl $50,%eax
    je.L5
    cmpl $50,%eax
    jg.L10
    cmpl $10,%eax
    je.L3
    cmpl $20,%eax
```

```
    je. L7
    jmp. L8
    . p2align 4, ,7
. L10:
    cmpl  $70,%eax
    je. L6
    cmpl  $80,%eax
    je. L4
    jmp. L8
    . p2align 4, ,7
. L3:
    movl  $10,-4(%ebp)
    jmp. L2
    . p2align 4, ,7
. L4:
    movl  $80,-4(%ebp)
    jmp. L2
    . p2align 4, ,7
. L5:
    movl  $50,-4(%ebp)
    jmp. L2
    . p2align 4, ,7
. L6:
    movl  $70,-4(%ebp)
    jmp. L2
    . p2align 4, ,7
. L7:
    movl  $20,-4(%ebp)
    jmp. L2
    . p2align 4, ,7
. L8:
    movl  $40,-4(%ebp)
. L2:
    movl -4(%ebp),%edx
    imull -4(%ebp),%edx
```

```
      leal -1(%edx),%eax
      cmpl  $9,%eax
      ja. L19
      movl. L20(,%eax,4),%eax
      jmp * %eax
      . p2align 4,,7
. section. rodata
      . align 4
      . align 4
. L20:
      . long. L13
      . long. L18
      . long. L19
      . long. L19
      . long. L16
      . long. L14
      . long. L12
      . long. L19
      . long. L15
      . long. L17
. text
      . p2align 4,,7
. L12:
      movl  $7,-4(%ebp)
      jmp. L11
      . p2align 4,,7
. L13:
      movl  $1,-4(%ebp)
      jmp. L11
      . p2align 4,,7
. L14:
      movl  $6,-4(%ebp)
      jmp. L11
      . p2align 4,,7
. L15:
```

```
	movl $9,-4(%ebp)
	jmp .L11
	.p2align 4,,7
.L16:
	movl $5,-4(%ebp)
	jmp .L11
	.p2align 4,,7
.L17:
	movl $10,-4(%ebp)
	jmp .L11
	.p2align 4,,7
.L18:
	movl $2,-4(%ebp)
	jmp .L11
	.p2align 4,,7
.L19:
	movl $40,-4(%ebp)
.L11:
.L1:
	leave
	ret
.Lfe1:
	.size main,.Lfe1-main
	.ident"GCC:(GNU)egcs-2.91.66 19990314/Linux(egcs-1.1.2 release)"
```

8.9 一个 C 语言程序如下：

```
extern int a;
static int b;
int c;
main(){
	b=a;
}
```

它在 x86/Linux 系统上编译后生成的汇编代码如下(编译器版本见汇编代码最后一行),请说明编译时对 extern 变量的处理和外部变量的处理有什么区别?

```
	.file "extern.c"
	.version "01.01"
```

```
gcc2_compiled.:
.text
    .align 4
.globl main
    .type main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    movl a,%eax
    movl %eax,b
.L1:
    leave
    ret
.Lfe1:
    .size main,.Lfe1-main
    .local    b
    .comm     b,4,4
    .comm     c,4,4
    .ident    "GCC:(GNU)egcs-2.91.66 19990314/Linux(egcs-1.1.2 release)"
```

8.10 在C语言中，若a和b是相同的结构体类型，那么赋值a=b是可以的。但是编译器对这种赋值的实现方式可能和它们的字节数(size)有关。下面是两个C语言程序及其在x86/Linux系统上，经某版本的GCC编译器编译生成的汇编代码(略去了和本题目无关的部分)。扼要叙述这些汇编代码中体现出的实现方式上的区别(在下面汇编程序中给出的注释仅供参考)。

第一个C程序及对应汇编程序：

```
struct{long i,j;double m;}a,b;
main(){
    a=b;
}
```

```
main:
    pushl %ebp
    movl %esp,%ebp
    movl $b,%eax
    movl $a,%edx
    movl (%eax),%ecx
```

```
        movl %ecx,(%edx)
        movl 4(%eax),%ecx
        movl %ecx,4(%edx)
        movl 8(%eax),%ecx
        movl %ecx,8(%edx)
        movl 12(%eax),%eax
        movl %eax,12(%edx)
    .L1:
        leave
        ret
```

第二个C程序及对应汇编程序：

```
    struct{long i,j;double m,n;}a,b;
    main(){
        a=b;
    }

    main:
        pushl %ebp
        movl %esp,%ebp
        pushl %edi
        pushl %esi
        movl  $a,%edi
        movl  $b,%esi
        cld                 //设定方向标志,使地址指针自动增量
        movl  $6,%ecx       //设定重复次数
        rep                 //表示重复下面的指令
        movsl               //传送指令
    .L1:
        leal-8(%ebp),%esp
        popl %esi
        popl %edi
        leave
        ret
```

8.11　在C语言中，若a和b是相同的结构体类型，那么赋值a=b是可以的。这句话对下面程序中出现的赋值p=q是否有效，即编译器会报错吗？若编译器不报错，程序的输出是什么？

```
typedef struct{int n;float a[];}record;
record p={2,{1.0,2.0}},q={3,{1.0,2.0,3.0}};
main(){
    p=q;
    printf("%d,%f,%f,%f\n",p.n,p.a[0],p.a[1],p.a[2]);
    printf("%d,%f,%f,%f\n",q.n,q.a[0],q.a[1],q.a[2]);
}
```

*第 9 章

独立于机器的优化

如果简单地把高级语言的每个构造独立地翻译成机器代码,那么这种方式会引起较大的运行开销。在编译过程中,通过删除目标代码中不必要的指令,用速度较快并完成同样事情的代码序列代替较慢的代码序列等变换来降低所生成代码的运行开销,称之为**代码改进**或**代码优化**。

第8章已经介绍了一些在基本块内的局部优化,本章介绍全局代码优化,它的信息收集和代码改进变换都不局限于一个基本块。大多数全局优化基于**数据流分析**,各种数据流分析都由一些收集程序信息的算法构成。数据流分析的结果都有同样的形式:对程序中的每条指令,它们描述该指令每次执行时程序必定会保持的某些性质。不同的分析计算不同的性质。例如,常量传播分析是对每个程序点,确定每个变量在该程序点是否维持一个不变的值。再例如,活跃变量分析是对每个程序点,确定各变量在该点保持的值在下次引用前是否肯定被覆盖。若是,则在该程序点,无论是寄存器还是内存单元都不需要保存这样的值。

9.1节通过一个实例来介绍代码改进的一些主要机会。9.2节介绍的数据流分析包括几个重要的实例,它们全局地收集一些重要信息,随后将这些信息用于代码改进变换。9.3节介绍数据流分析的理论基础和一般框架。9.2节所介绍的几种数据流分析都是该框架的实例,这些数据流分析用的是本质上相同的算法,因而可以用同样方式来评估这些数据流分析的性能和证明它们的正确性。9.4节介绍一种在此框架上能力较强的数据流分析。9.5节介绍部分冗余删除(partial redundancy elimination),它是能力更强的优化技术,它用于确定程序中计算每个表达式的恰当位置。这个问题的解决需要借助多种不同数据流问题的解。9.6节讨论程序中循环的识别和分析。循环的识别引领求解数据流问题的另一类算法,它们基于可归约程序的循环层次结构。

9.1 优化的主要种类

编译器的优化必须保证程序的语义。除了非常特殊的情况外,对于程序员选择并实现的算法,编译器不可能对相应程序理解到能够用另一个本质上不同的更有效算法来代替。编译器只知道怎样实施一些相对低级的语义变换,例如,利用像代数恒等 $i+0=i$ 这样的一般事实,或者利用像相同的运算作用于相同的值得到相同的结果这样的程序语义。

9.1.1 优化的主要源头

在一个典型的程序中,存在许多冗余的运算。有时,这些冗余在源级是可见的。例如,程序员发现重复某些计算可能更直接和方便,因而把识别它们并优化成仅计算一次的任务留给编译器。更经常的情况是,冗余是用高级语言写程序的一种副作用。在大多数编程语言中(不包括允许指针算术的 C 和 C++),程序员除了通过像 A[i][j]和 X.f1 的方式引用数组元素和结构体的域外,没有其他更有效的方式。但是随着程序被编译,这些对高级数据结构的访问展开成多步低级算术运算,例如数组 A 第(i,j)个元素的地址计算。在一个程序中,对同一个数据结构的多次访问经常共享许多公共的低级运算。程序员不知道这些低级运算,他们自己没有办法删除这些冗余。事实上,从软件工程的观点看,也宁可程序员只能用源程序中的名字访问数据元素,而不希望他们能使用低级的操作。这样的程序易于编写,更重要的是易于理解和进化。让编译器来删除这些冗余,可以达到两全其美:程序既高效又容易维护。

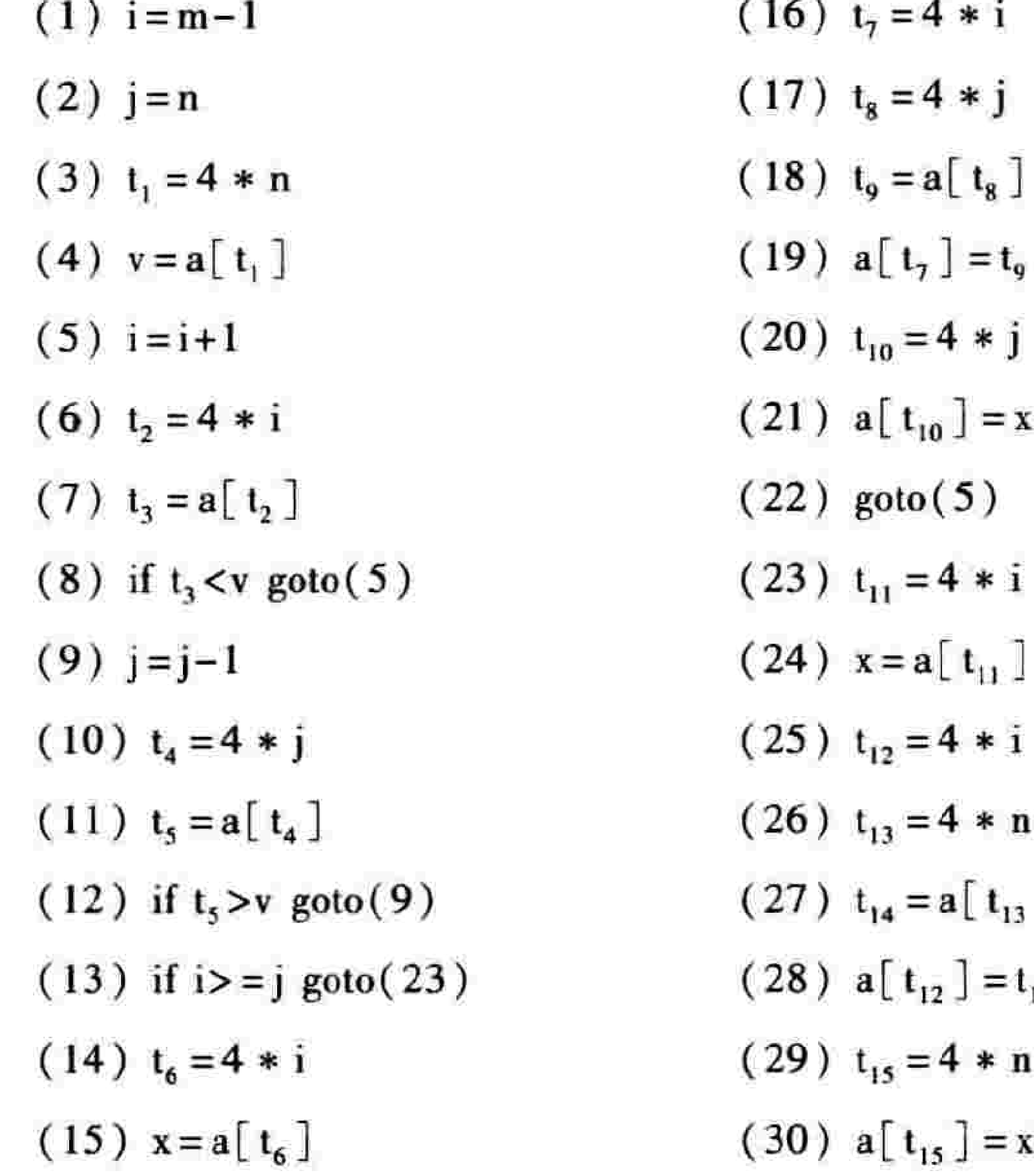
9.1.2 一个实例

下面用快速排序程序 quickSort 来介绍公共子表达式删除、复写传播、死代码删除、强度削弱和归纳变量删除等几种重要的代码改进变换。图 9.1 是它的 C 代码。为了这个程序能正常工作,a[0]和 a[max]应分别是被排序的最小元素和最大元素。

```
void quickSort(int m,int n){
    int i,j;
    int v,x;
    if(n<=m)return;
    /* 下面讨论所关心的程序段从这里开始 */
    i=m-1;j=n;v=a[n];
    while(1){
        do i=i+1;while(a[i]<v);
        do j=j-1;while(a[j]>v);
        if(i>=j)break;
        x=a[i];a[i]=a[j];a[j]=x;      /* 交换 a[i]和 a[j] */
    }
    x=a[i];a[i]=a[n];a[n]=x;      /* 交换 a[i]和 a[n] */
    /* 下面讨论所关心的程序段到这里结束 */
    quickSort(m,j);  quickSort(i+1,n);
}
```

图 9.1 快速排序的 C 代码

在通过优化删除图 9.1 程序中冗余的地址计算前，首先将这些地址计算分解成低级的算术运算以揭示其中的冗余。本章剩余部分假定采用三地址语句方式的中间表示，其中临时变量用来保存所有中间表达式的结果。为图 9.1 中被标注程序段产生的中间代码见图 9.2。



(1) $i=m-1$
(2) $j=n$
(3) $t_1=4*n$
(4) $v=a[t_1]$
(5) $i=i+1$
(6) $t_2=4*i$
(7) $t_3=a[t_2]$
(8) if $t_3<v$ goto(5)
(9) $j=j-1$
(10) $t_4=4*j$
(11) $t_5=a[t_4]$
(12) if $t_5>v$ goto(9)
(13) if $i>=j$ goto(23)
(14) $t_6=4*i$
(15) $x=a[t_6]$
(16) $t_7=4*i$
(17) $t_8=4*j$
(18) $t_9=a[t_8]$
(19) $a[t_7]=t_9$
(20) $t_{10}=4*j$
(21) $a[t_{10}]=x$
(22) goto(5)
(23) $t_{11}=4*i$
(24) $x=a[t_{11}]$
(25) $t_{12}=4*i$
(26) $t_{13}=4*n$
(27) $t_{14}=a[t_{13}]$
(28) $a[t_{12}]=t_{14}$
(29) $t_{15}=4*n$
(30) $a[t_{15}]=x$

图 9.2　图 9.1 部分程序的三地址代码

在这个例子中，假定每个整数占 4 个字节。赋值 $x=a[i]$ 按 7.3.3 节的方式翻译成两个三地址语句（见图 9.2 的(14)和(15)）：

$t_6=4*j$

$x=a[t_6]$

类似地，$a[j]=x$ 也翻译成两个三地址语句（见图 9.2 的(20)和(21)）：

$t_{10}=4*j$

$a[t_{10}]=x$

正因为源程序中每个数组访问都翻译成两步（下标乘以 4 和数组索引操作），导致这段较短的源代码翻译成相当长的三地址语句序列。

图 9.3 是图 9.2 程序的流图，块 B_1 是初始结点。程序所有的条件转移和无条件转移在图 9.3中都改成转移到目标语句所在的基本块。图 9.3 的流图有三个循环：块 B_2 和 B_3 分别单独构成一个循环；块 B_2、B_3、B_4 和 B_5 一起形成一个循环，入口结点是块 B_2。

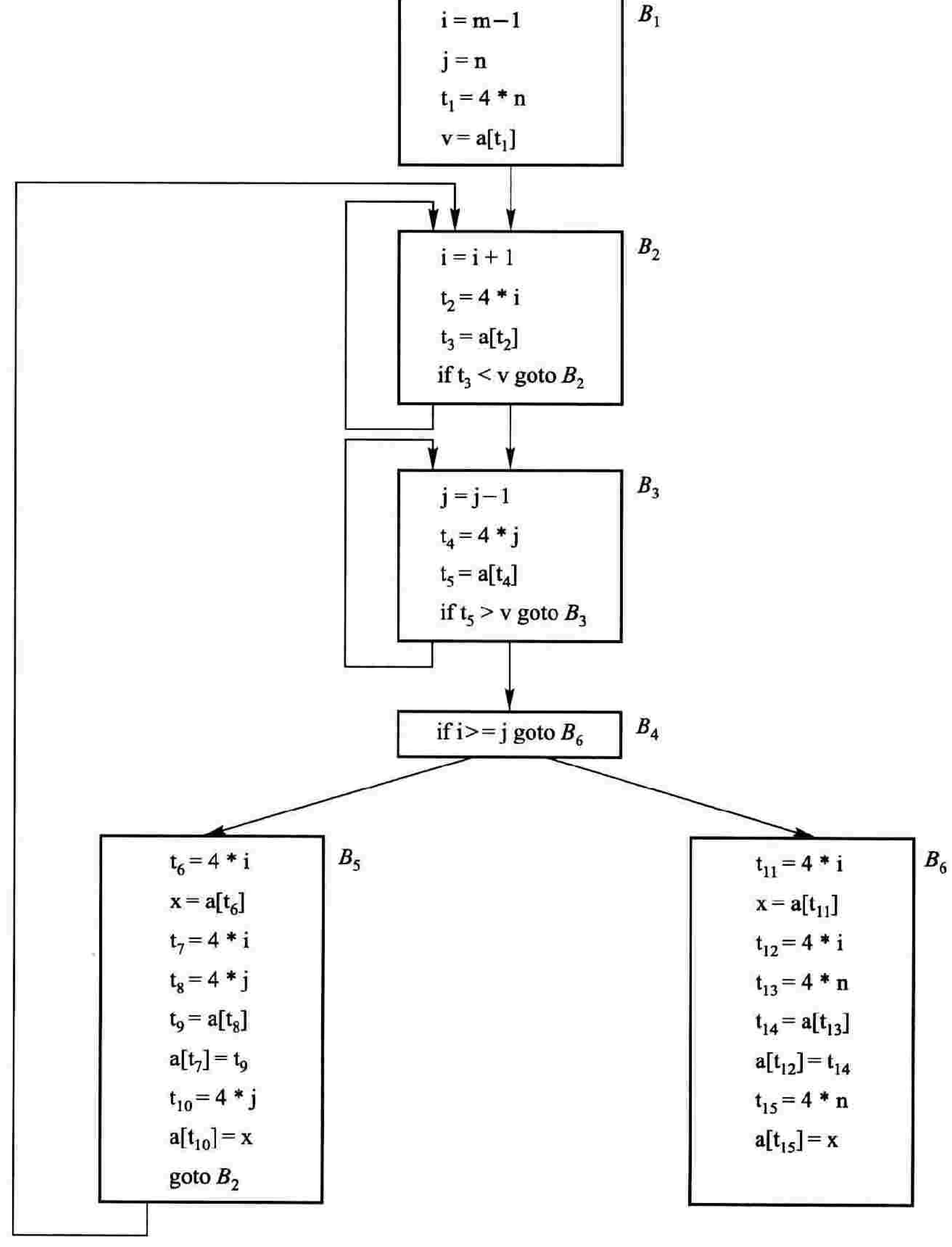


图 9.3　图 9.2 程序的流图

9.1.3　公共子表达式删除

如果表达式 E 先前已计算，并且从先前的计算到 E 的再次出现，E 中变量的值没有改变，那么 E 的这个再次出现称为**公共子表达式**。如果能够利用先前的计算结果，就可以避免表达式的

重复计算。

例 9.1 在图 9.3 的基本块 B_5 中(见图 9.4(a)),对 t_7 和 t_{10} 赋值的语句分别有公共子表达式 4 * i 和 4 * j 出现在它们的右部。用 t_6 代替 t_7,用 t_8 代替 t_{10},这些公共子表达式得以删除,删除后该基本块的代码如图 9.4(b)所示。这是仅考察基本块内部可完成的公共子表达式的删除。 □

例 9.2 在图 9.3 的流图中,块 B_5 和 B_6 中全局公共子表达式和局部公共子表达式删除后的结果在图 9.5 给出。这里重点讨论 B_5 的变换。

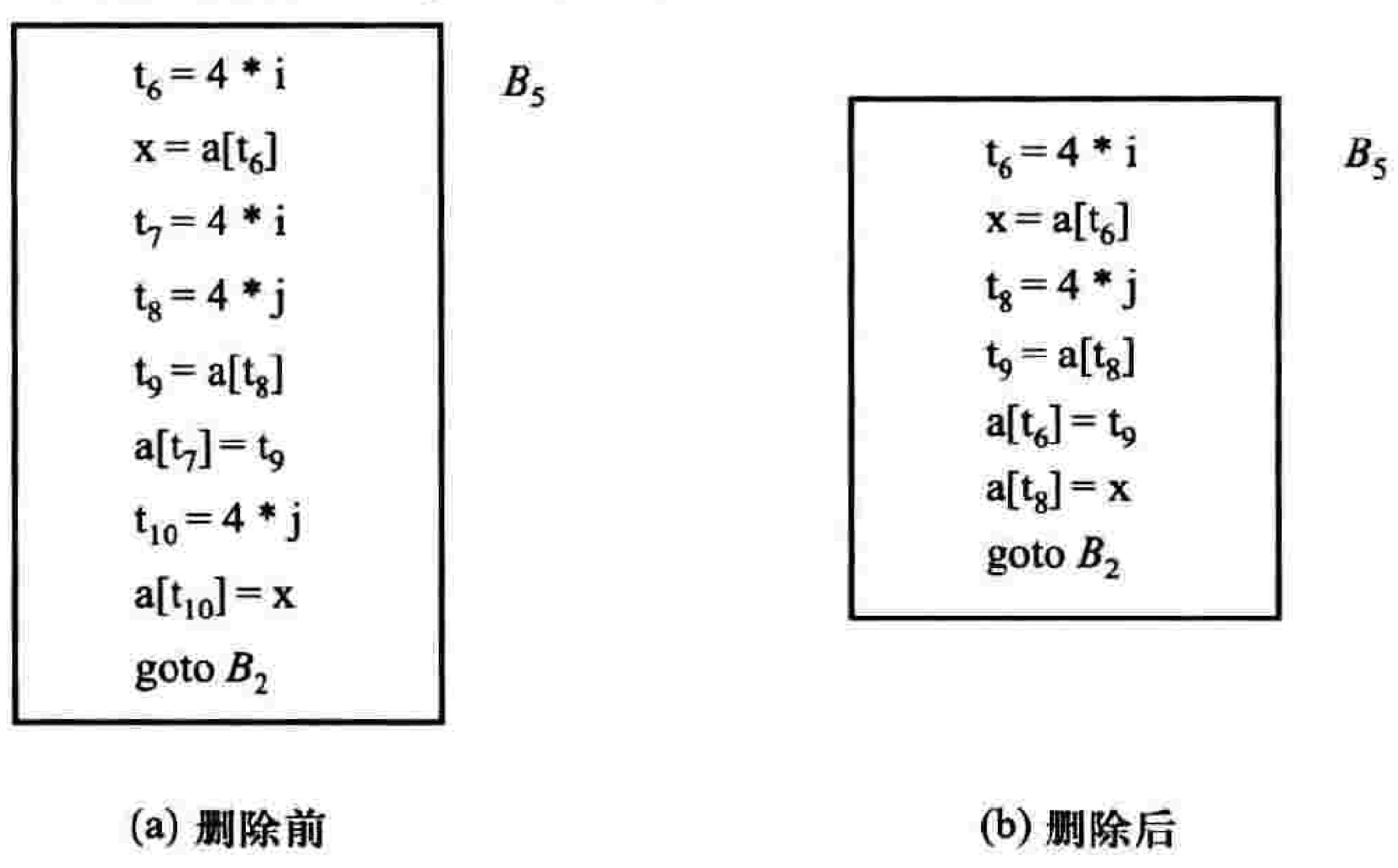


图 9.4 局部公共子表达式删除

删除了局部公共子表达式后,B_5 仍然计算 4 * i 和 4 * j,从全局看,它们仍然是公共子表达式。B_5 中三个语句

$t_8 = 4 * j$ $\quad$ $t_9 = a[t_8]$ $\quad$ $a[t_8] = x$

可以由

$t_9 = a[t_4]$ $\quad$ $a[t_4] = x$

代替。因为从图 9.5 可以看到,t_4 在 B_3 中计算,当控制从 B_3 中 4 * j 的计算流到 B_5 时,中间没有改变 j 的值,所以在 B_5 中需要 4 * j 的值时可以引用 t_4。

t_4 代替 t_8 后,B_5 的另一个公共子表达式变得清楚了。它是表达式 $a[t_4]$,对应于源代码中 a[j]。不仅控制离开 B_3 和进入 B_5 时 j 的值没有变,而且 a[j]的值也不变(a[j]的值计算在临时变量 t_5 中),因为在这段区间中没有对 a 的元素赋值。这样,B_5 的语句

$t_9 = a[t_4]$ $\quad$ $a[t_6] = t_9$

可以由 $a[t_6] = t_5$ 代替。

类似地,图 9.4(b)的 B_5 中对 x 赋的值和 B_2 中对 t_3 赋的值一样。删掉图 9.4(b)中对应到源代码表达式 a[i]和 a[j]的公共子表达式后,其结果就是图 9.5 的 B_5。

图 9.5 的 B_6 也是完成了一系列类似变换后的结果。图 9.5 的 B_1 和 B_6 中表达式 $a[t_1]$不能看作公共子表达式,虽然 t_1 在两个地方都使用,但是因为控制离开 B_1 进入 B_6 之前,它可以通过

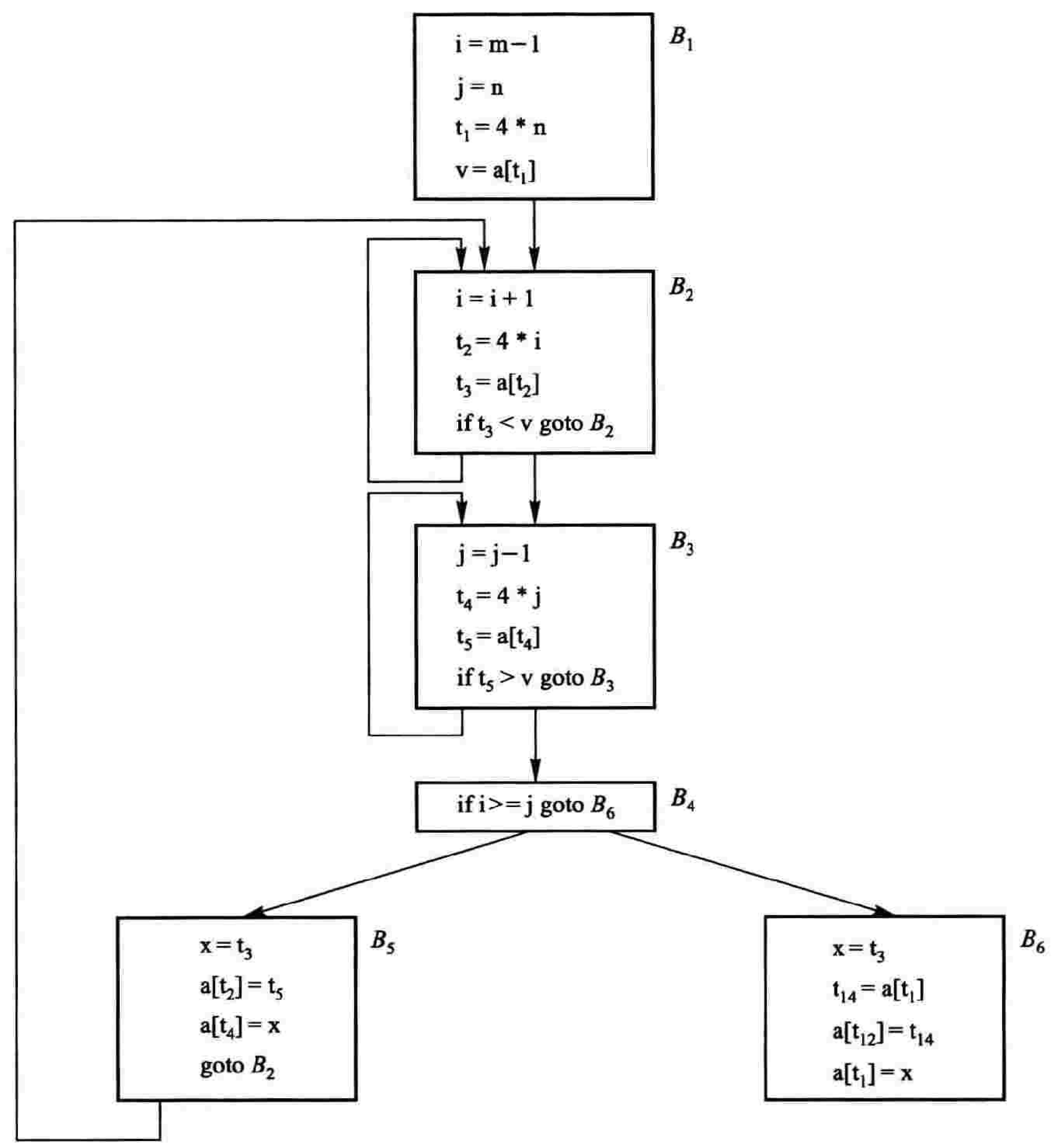


图 9.5 删除公共子表达式后的 B_5 和 B_6

B_5,B_5 有对 a 的赋值,因此在到达 B_6 时,a[t_1]的值可能和离开 B_1 时的值不一样,把 a[t_1]作为公共子表达式不是稳妥的。 □

9.1.4 复写传播

图 9.5 的 B_5 可以通过使用两种新的变换来删除 x 而进一步化简。一种变换是下一小节介绍的死代码删除;另一种变换和形式为 f=g 的赋值有关,这种赋值叫做**复写语句**,简称为**复写**。复写传播变换是指在复写语句 f=g 的后面,尽可能用 g 代表 f。例 9.2 其实已经用到了复写,不仅删除公共子表达式会引入复写,而且利用复写传播会得到新的公共子表达式。某些场合下的

删除公共子表达式可能会引入更多的复写。例如,当删除图 9.6 的公共子表达式 c=d+e 时,需要使用新的变量 t 来保存 d+e 的值。因为控制到达 c=d+e 可能会在对 a 的赋值之后,也可能会在对 b 的赋值之后,因此用 c=a 或 c=b 来代替 c=d+e 都不是稳妥的。

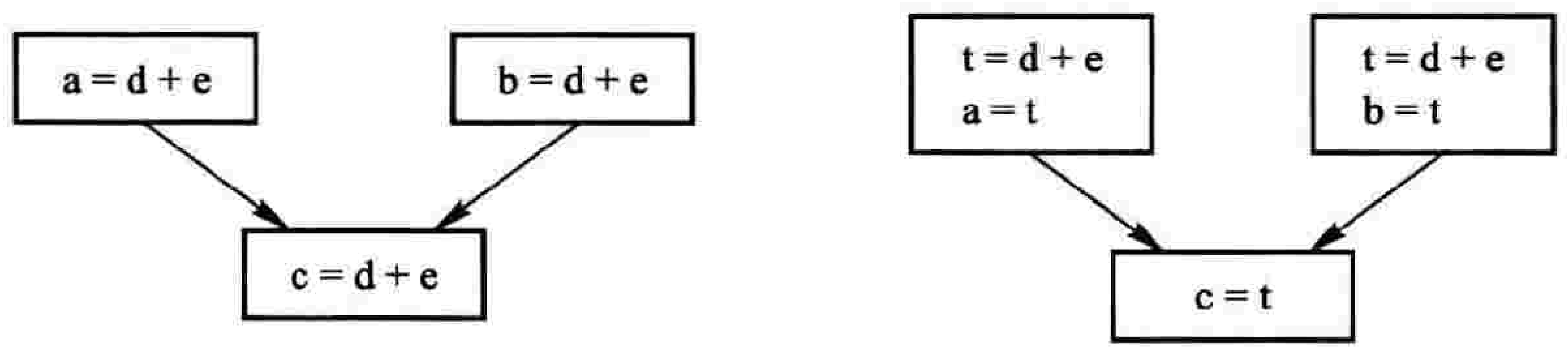


图 9.6　删除局部公共子表达式时引入复写

图 9.5 中 B_5 的赋值 $x=t_3$ 是一个复写。把复写传播运用于 B_5 后,它的四个语句是

$x=t_3$

$a[t_2]=t_5$

$a[t_4]=t_3$

goto B_2　(9.1)

这看起来似乎没有改进,但在 9.1.5 节会看到,它增加了删除对 x 赋值的机会。

复写传播还可能引起一些其他的优化机会。若在编译时能推断出一个表达式的所有运算对象都是常量,则可以在编译时完成该表达式的计算,并用计算结果代替该表达式,这种变换叫做**常量合并**。复写传播可能会引入一些常量合并的机会。9.1.2 节的例子没有可以实施常量合并的地方。

9.1.5 死代码删除

如果一个变量的值以后还要引用,则称它在该程序点是活跃的,否则它在该点已死亡。**死代码**或**无用代码**就是指计算的结果决不被引用的语句,删除它们不会影响程序的语义。虽然程序员一般不大可能引入死代码,但是上面介绍的变换可能会引起死代码。事实上例 9.2 已经使用了死代码删除。

复写传播的优点之一是经常导致复写语句成为死代码,(9.1)的代码可以进一步优化。删掉(9.1)中的死代码 $x=t_3$,它变成

$a[t_2]=t_5$

$a[t_4]=t_3$

goto B_2

这段代码是对图 9.5 中 B_5 的进一步改进。

有时,程序员也会利用优化编译器删除死代码的特点来方便自己调试程序。例如,在程序中增加像

if (debug) print…　　(9.2)

这样的条件打印语句,当调试结束时并不把它们从程序中删除,而是将程序开头的

debug = TRUE

改成

debug = FALSE

当复写传播用 FALSE 代替 debug 时,条件打印语句的条件为假,打印语句就成了死代码,条件判断和打印语句都可以从目标代码中删除。

9.1.6 代码外提

本小节和下一小节简单介绍一个非常重要且值得优化的地方——循环,尤其是消耗程序运行大部分时间的内循环。如果内循环的指令数量得以减少,即使这时外循环的指令数增加了,程序的运行时间仍可能缩短。减少循环中代码数量的一种重要办法是**代码外提**。它把**循环不变计算**,即运算结果独立于循环体执行次数的表达式,放到循环之前计算。例如,语句

```
while(i<=limit-2){/* 不改变 limit 的语句 */}
```

中 limit-2 是循环不变计算。代码外提的结果是

```
t = limit-2;
while(i<=t){/* 不改变 limit 的语句 */}
```

9.1.2 节的例子没有可以实施代码外提的地方。

9.1.7 强度削弱和归纳变量删除

循环的另一种优化是寻找循环中的**归纳变量**,并优化对它们的计算。变量 x 称为归纳变量,如果存在一个正或负的常量 c,使得每次对 x 赋值时,它的值实质上增加 c。例如,i 和 t_2 是图 9.5 中包含 B_2 的循环中的归纳变量。对 t_2 来说,虽然语句是 $t_2 = 4 * i$,但实际上 t_2 的值是在原来基础上加 4。在每次循环迭代时,归纳变量可以用加或减一个常量来计算。用一个较廉价的运算代替一个花费较大的运算称为**强度削弱**,例如用加代替乘。

归纳变量不仅有时可用来实施强度削弱,而且对一组在循环中步伐一致地改变它们值的归纳变量,经常可以仅保留其中一个归纳变量,删除其余的归纳变量。

在处理循环时,合适的做法是"从里向外",即从最内层循环开始,沿直接外围循环逐层向外。把这种思想用于快速排序程序,从块 B_3 构成的最内层循环开始来进行优化。从图 9.5 可以看出 j 和 t_4 形成一组归纳变量,因为它们的值步伐一致地变化,每次 j 的值减 1 后,$4 * j$ 赋给 t_4,t_4 的值实质上减 4。但是仅看由 B_3 构成的内循环,j 和 t_4 都不能删去,因为 t_4 在 B_5 中引用,j 在 B_4 中引用。然而可以用它们来举例说明强度削弱,而这个强度削弱又为

删除归纳变量创造了机会。最终,当考察由 B_2,B_3,B_4 和 B_5 构成的外循环时,j 可以被删除。

例 9.3 在图 9.7(a)中,对由 B_3 构成的内循环,若不考虑第一次进入 B_3,关系 $t_4=4*j$ 在 B_3 的入口一定保持,在 j=j-1 后,关系 $t_4=4*j+4$(即 $4*j=t_4-4$)也保持,那么 $t_4=4*j$ 可以用 $t_4=t_4-4$ 代替。要进行这个变换的唯一问题是第一次进 B_3 时 t_4 没有初值,所以在给 j 置初值的那个基本块末尾添加对 t_4 置初值 4 * j 的语句。在图 9.7(b)中,这个语句放在块 B_1 的最后。在许多机器上,乘运算比加或减需要更多时间,这种变换会加快目标代码的运行速度,虽然在 B_1 的最后增加了一个语句。 □

下面再举一个删除归纳变量的例子来作为本节的结束,该例在外循环 B_2,B_3,B_4 和 B_5 的上下文中处理 i 和 j。

例 9.4 把强度削弱用于 B_2 和 B_3 的内循环后,i 和 j 的作用仅在于决定 B_4 的测试结果。由于 j 和 t_4 满足关系 $t_4=4*j$,i 和 t_2 满足关系 $t_2=4*i$,测试 $t_2>=t_4$ 等价于 i>=j。一旦作出这种替换,B_2 的 i 和 B_3 的 j 就成了死变量,在这些块中对它们的赋值也就成了死代码,可以删除,最终结果在图 9.8 中给出。 □

本节快速排序程序的代码改进变换的效果是明显的。在图 9.8 中,B_2 和 B_3 的指令数都从图 9.3 最初流图的 4 条减到 3 条,B_5 从 9 条减到 3 条,B_6 从 8 条减到 3 条。虽然 B_1 从 4 条增加到 6 条,但它在这段程序中仅执行一次,所以总的运行时间几乎不受 B_1 大小的影响。

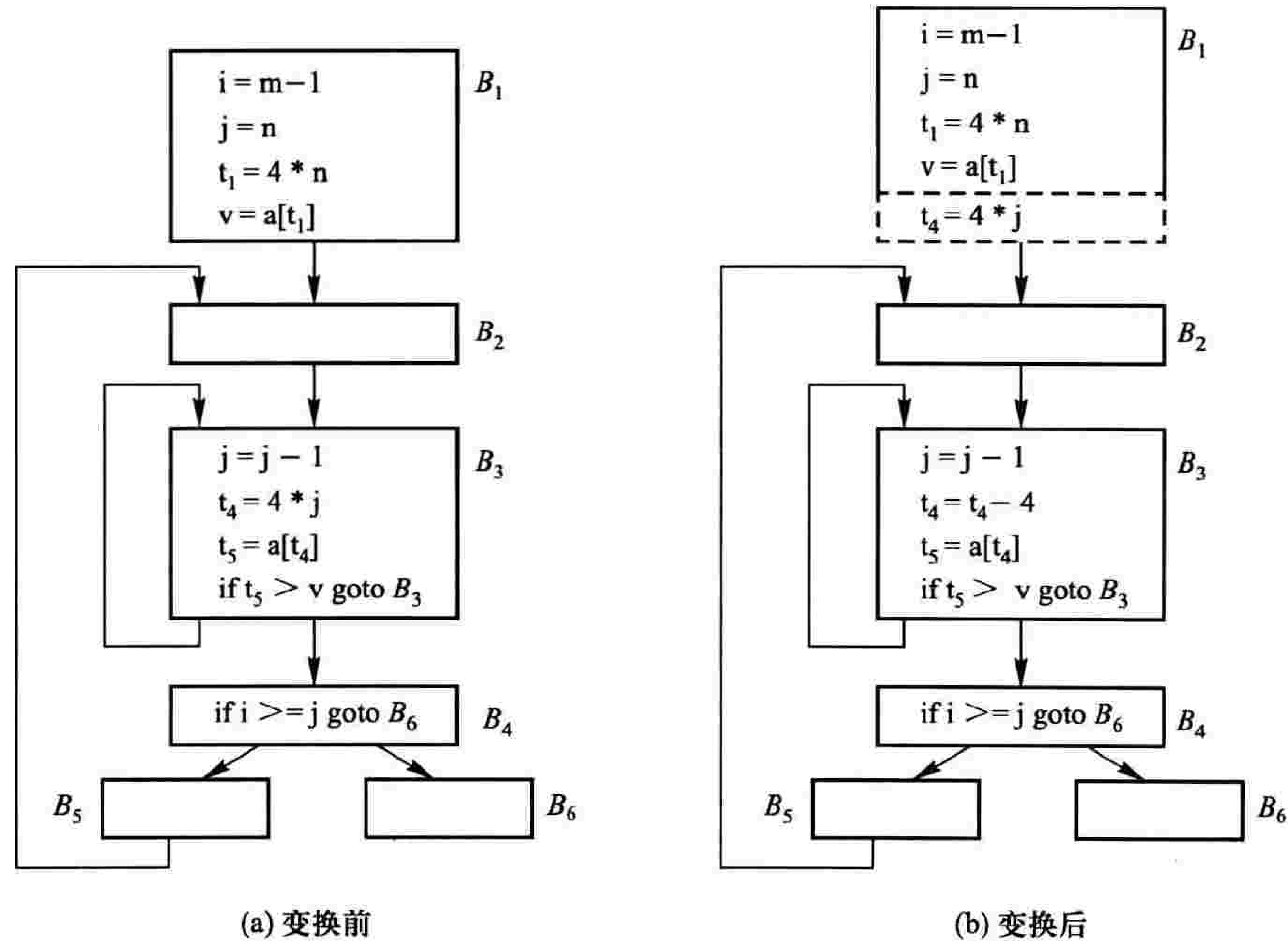


图 9.7 强度削弱用于块 B_3 的 4 * j

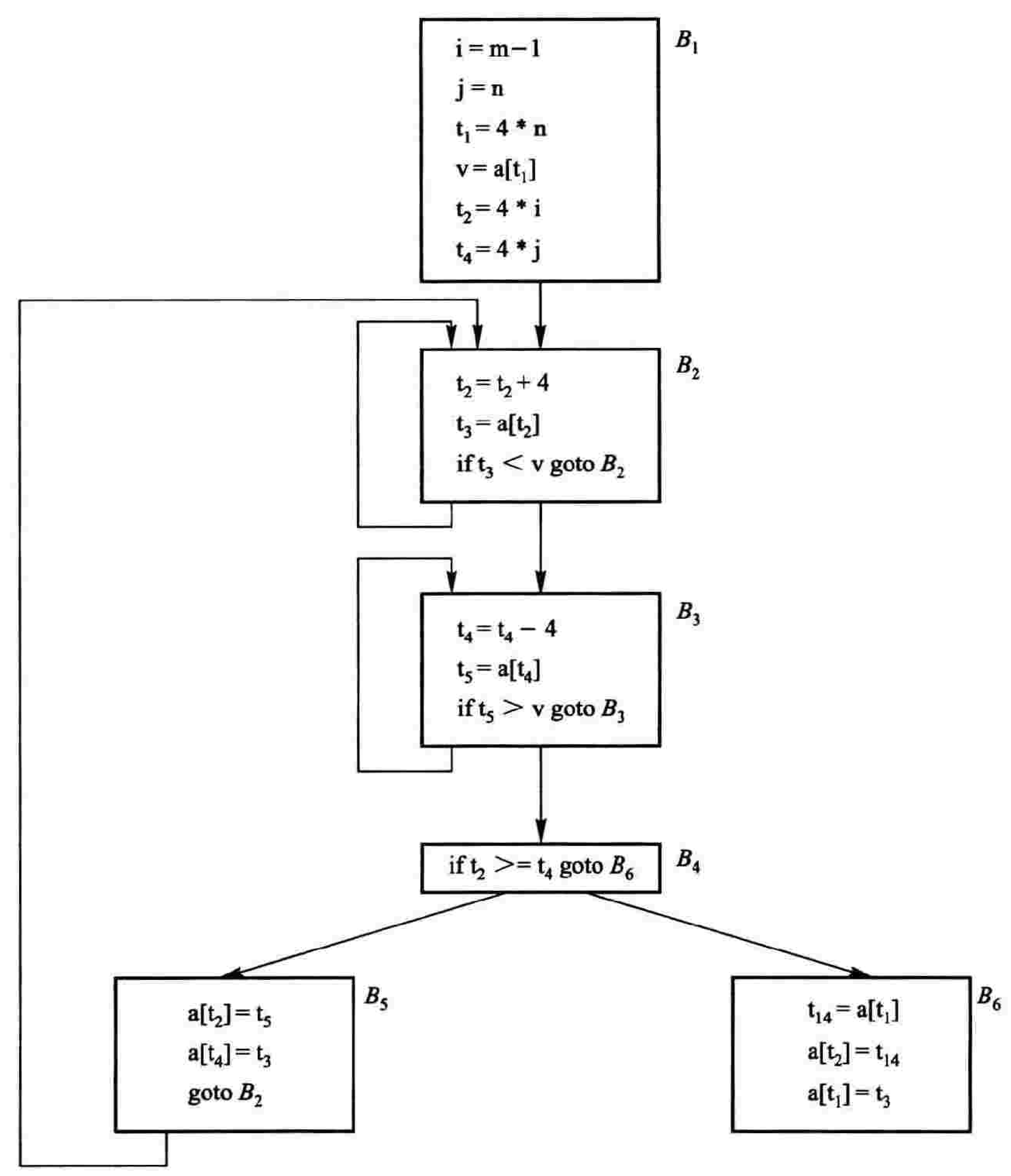


图 9.8 删除归纳变量后的流图

9.2 数据流分析介绍

9.1 节介绍的所有优化都依赖于**数据流分析**。数据流分析推导的是数据沿着程序执行路径流动的信息。例如,实现全局公共子表达式删除的一种方式是,需要判断在程序的任何执行路径上,两个正文上完全一样的表达式计算得到相同的值。还有,如果一个赋值的结果在随后的任何执行路径上都不使用,则可以把该赋值作为死代码删除。这些问题,还有其他很多问题都可以通

过数据流分析来回答。

9.2.1 数据流抽象

根据 6.1.2 节的介绍知道,程序状态由程序所有单元的值组成,包括运行栈的栈顶活动记录之下的那些活动记录中的值。一个程序的执行可以看成是在程序状态上的一系列变换。每个三地址语句的执行将一个输入状态转换为一个输出状态。输入状态和输出状态分别联系到该语句前后的那两个**程序点**。

在分析一个程序的行为时,必须在其流图上考虑该程序所有可能的执行路径(程序点序列),然后从每个程序点的可能状态中抽取解决特定数据流分析所需要的信息。在更复杂的分析中,例如调用语句或返回语句被执行时,还需要考虑路径在多个流图之间的跳转。本章作为对优化的初步介绍,只集中在单个流图(对应一个过程)的路径上。

现在来看从一个流图可以得到的执行路径。先看点之间的一些特征。

(1) 在一个基本块内,一个语句之后的程序点就是下一个语句之前的程序点。

(2) 如果有一条边从块 B_1 到块 B_2,那么 B_1 最后一个语句之后的那个程序点(B_1 的出口点)可以由 B_2 第一个语句之前的那个程序点(B_2 的入口点)直接跟随。

这样,从点 p_1 到点 p_n 的**执行路径**(简称**路径**)定义为点序列 $p_1, p_2, \cdots, p_n$,并且每个 $i=1,2,\cdots,n-1$ 满足:

(1) p_i 是一个语句前的那个点,p_{i+1} 是该语句后的那个点,或者

(2) p_i 是某块的出口点,p_{i+1}是该块一个后继块的入口点。

通常,从一个流图得到的程序执行路径的数目无限,并且不存在执行路径长度的有限上界。程序分析从一个程序点的所有可能状态中为该点总结出一组有限的事实,不同的程序分析可能提炼出不同的信息,通常来说,没有任何程序分析收集的信息必须是状态的完整描绘。

例 9.5 即使是图 9.9 这样简单的程序,它的执行路径数也是无上界的。不进入循环且最短的完整执行路径由程序点(1,2,3,4,9)组成。下一条最短路径执行循环的一次迭代,它由程序点(1,2,3,4,5,6,7,8,3,4,9)组成。例如,第一次到达点(5)时,a 的值是 1,它是由 d_1 定值的。因此说第一次迭代时,d_1 到达点(5)。在随后的迭代中,d_3 到达点(5),a 的值是 243。

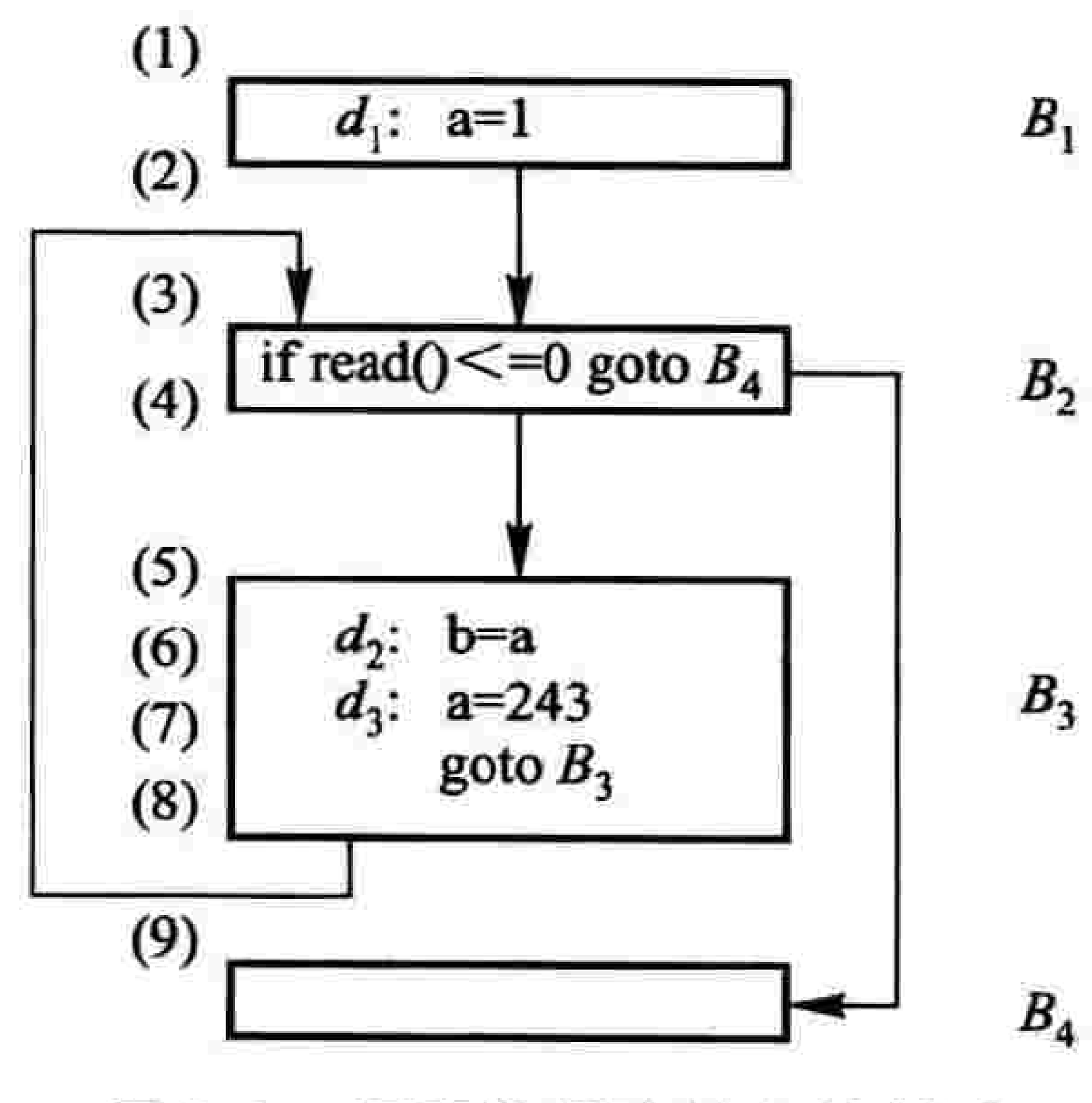


图 9.9 阐明数据流抽象的例子

通常,明了所有路径上的所有程序状态是不可能的。数据流分析不区分到达一个程序点的不同路径,而且也不掌握完整的状态,但是它提炼出某些细节,以获取可用于分析的数据。下面两个例子说明怎样从一个点的状态中提炼

出不同的信息。

(1) 为了帮助程序员调试程序,可能希望找出一个变量在某个程序点的所有可能取值,以及它们是在何处定值的。例如,对图 9.9 点(5)的所有程序状态可以这样总结:a 的值是{1,243}中的一个,并且由{d_1,d_3}中的一个语句定值。沿着某条路径可以到达一个程序点的定值叫做该点的**到达-定值**。

(2) 考虑常量合并的实现。如果 x 的一个引用仅由它的一个定值到达,并且该定值给 x 赋一个常量,那么可以直截了当地用这个常量来代替 x 的这个引用。反过来,如果 x 的几个定值都可以到达一个程序点,那么就不可能对 x 执行常量合并。因此对于常量合并问题,希望的是找到这样的定值,它们是相应变量到达一个程序点的唯一定值,而不管该定值经过怎样的路径。例如,对于图 9.9 的点(5),不存在 a 的某个定值,它是到达该点的 a 的唯一定值。即使某变量在某点有唯一定值,该定值还必须给这个变量赋常量才能用于常量合并。因此对于图 9.9 的点(5),可以简单地描述 a"不是常量",而不是去收集所有可能的值或所有可能的定值。

从这两个例子可以看出,从同样的信息可以提炼出不同的总结,这取决于程序分析的不同目的。 □

9.2.2 数据流分析模式

在数据流分析的每种应用中,总把每个程序点和一个**数据流值**联系起来,数据流值代表在该点能观测到的所有可能程序状态集合的一个抽象。可能数据流值的集合叫做该应用的**论域**,例如,对于到达-定值,数据流值的论域就是该程序中所有定值组成集合的幂集,其中一种具体应用是,联系到每个程序点的数据流值正好是能够到达该点的定值集合。如先前所讨论的那样,信息提炼的选择依赖于数据流分析的目的;为了使分析高效,数据流分析只跟踪记录相关的信息。

在每个语句 s 前后两个点的数据流值分别用 IN[s]和 OUT[s]来表示。**数据流问题**就是对所有语句 s 的 IN[s]和 OUT[s]上的约束寻找一个解。这些约束分成两组,分别是基于语句语义的约束(迁移函数)和基于控制流的约束。

先讨论迁移函数。一个语句前后两点的数据流值受该语句的语义约束。例如,假定数据流分析是用来确定变量在程序点的值。如果变量 a 在语句 b=a 执行前有值 v,那么该语句执行后 a 和 b 的值都是 v。该赋值语句前后两点的数据流值之间的联系称为**迁移函数**。

迁移函数有两种风格:信息沿执行路径正向传播或者逆向传播。在一个正向流问题中,一个语句 s 的迁移函数(通常用 f_s 来表示)以该语句前那个点的数据流值为变元,产生该语句后那个点的数据流值,即

$$\text{OUT}[s]=f_s(\text{IN}[s])$$

反过来,在一个逆向流问题中,迁移函数以该语句后那个点的数据流值为变元,产生该语句前那个点的数据流值,即

$$\text{IN}[s]=f_s(\text{OUT}[s])$$

再讨论控制流约束。数据流值上的第二组约束从控制流推导出来。以正向流问题为例。在基本块内部,控制流是简单的。如果基本块 B 由语句$s_1, s_2, \cdots, s_n$依次组成,那么一定有

$$IN[s_{i+1}] = OUT[s_i], \quad i=1,2,\cdots,n-1$$

然而,基本块之间的控制流边引出在基本块出口点和后继基本块入口点之间更加复杂的约束。例如,如果收集到达程序点的所有定值,那么到达基本块入口点的定值集合,是到达该块各前驱块出口点的定值集合的并集。下面更仔细地讨论数据在基本块之间的流动。

由于基本块内的控制流是简单的,没有中断,也没有分支等,因此可以将基本块内各语句的迁移函数作复合运算,形成基本块的迁移函数。仍以正向流问题为例,若基本块 B 由语句 $s_1, s_2, \cdots, s_n$ 依次组成,则 B 的迁移函数 f_B 是(逆向流问题的 f_B 可类似地得到)

$$f_B = f_{s_n} \circ \ldots \circ f_{s_2} \circ f_{s_1}$$

在建立控制流约束时不用关心基本块内部的情况,因此称 $IN[B]$ ($=IN[s_1]$)和 $OUT[B]$ ($=OUT[s_n]$)分别为基本块 B 入口和出口的数据流值(因而也不区分块入口和块入口点概念,对块出口也是这样)。这样,基本块的出口和入口之间数据流值的联系可以用下面的等式表达

$$OUT[B] = f_B(IN[B])$$

现在可以来写控制流约束了。例如,如果数据流值是赋给某个变量的常量集合,那么该正向流问题的控制流约束可以用等式表示:

$$IN[B] = \bigcup_{P\text{是}B\text{的前驱}} OUT[P]$$

对于逆向流问题,例如活跃变量分析,其等式类似,但是 IN 和 OUT 的作用颠倒,即

$$IN[B] = f_B(OUT[B])$$

$$OUT[B] = \bigcup_{S\text{是}B\text{的后继}} IN[S]$$

和线性算术方程不一样,由上述数据流等式形成的方程组通常解不唯一。求解的目标是要找到满足这两组约束(控制流约束和迁移约束)的最“精确”解。也就是需要一个鼓励合法代码改进的解,而不是可能引起不安全变换(改变程序所计算的东西)的解。下面的几小节讨论一些可以用数据流分析解决的重要问题。

9.2.3 到达-定值

到达-定值是一种最常用的数据流模式。如果控制到达每个程序点 p 时,能够知道每个变量 x 是在程序的哪些地方被定值的,那么就可以确定关于 x 的许多事情。拿已经举过的两个例子来说,编译器可以知道 x 在点 p 是否为常量,调试器可以告知 x 在点 p 是否为没有定值的变量。

如果存在从对 x 的定值 d 之后那个点到点 p 的一条路径,并且在这条路径上没有对 x 的定值,那么称定值 d 到达点 p。如果在这条路径上其他某个地方有对 x 的定值,那么称变量 x 在 d 的定值被**注销**。直观上说,如果某个变量 x 在 d 的定值到达点 p,并且运行时在点 p 引用 x,则 d 可能是 x 最近一次定值的位置。

在有别名的情况下,变量 x 的定值是一个语句,它给 x 赋值或可能给 x 赋值。因为过程参

数、数组访问、间接引用等都可能引起别名，因此很难确定一个语句是否联系到某个特定变量 x。程序分析必须是稳妥的，如果不能确定语句 *s* 是否给变量 x 赋值，那么必须认为它可能给 x 赋值。也就是在 *s* 之后，x 的值可能是原来的，也可能是 *s* 赋给它的新值。为简单起见，本章其余部分仅考虑变量无别名的情况，即只考虑大多数语言都有的局部标量，并且排除 C 和 C++语言能够获取局部变量地址的情况。

例 9.6 图 9.10 是有 7 个定值的流图。重点考察到达块 B_2 的定值。B_1 的所有定值都可以到达 B_2 的入口。B_2 的 d_5：j = j−1 也可以到达 B_2 的入口，因为在回到 B_2 的循环中不存在其他对 j 的定值。然而这个定值注销了定值 d_2：j = n，阻止它到达 B_3 或 B_4。语句 d_4：i = i+1 不能到达 B_2 的入口，因为变量 i 总是在 d_7：i = u3 被重新定值。最后，定值 d_6：a = u2 总能到达 B_2 的入口。 □

从到达−定值的定义可以看出，数据流分析有时是不精确的。例如，到达−定值的定义基于流图中的每条边都会经过，这个假设和实际情况是有出入的。例如，无论 a 和 b 取任何值都不能使控制流真正到达下面程序段的 *statement*2，即流图上的一条执行路径并不一定能成为一条运行时的执行路径。

```
if (a==b) statement1;else if(a==b) statement2;
```

一般而言，流图中每条路径是否为一条真正的运行路径，这是一个不可判定的问题，因此数据流分析简单地假设流图中每条路径总会在程序的某次执行中经过。这个假设是安全的或稳妥的，它是指该假设决不会导致实施改变程序所计算东西的变换，但是有可能导致没有发现某些能保持程序含义的变换。在到达−定值的大多数应用中，假定一个定值会到达某点而实际上它不一定到达是稳妥的。例如，当到达−定值信息用于常量合并时，它有可能导致某些常量没有合并，但它决不会导致实施改变程序含义的变换。

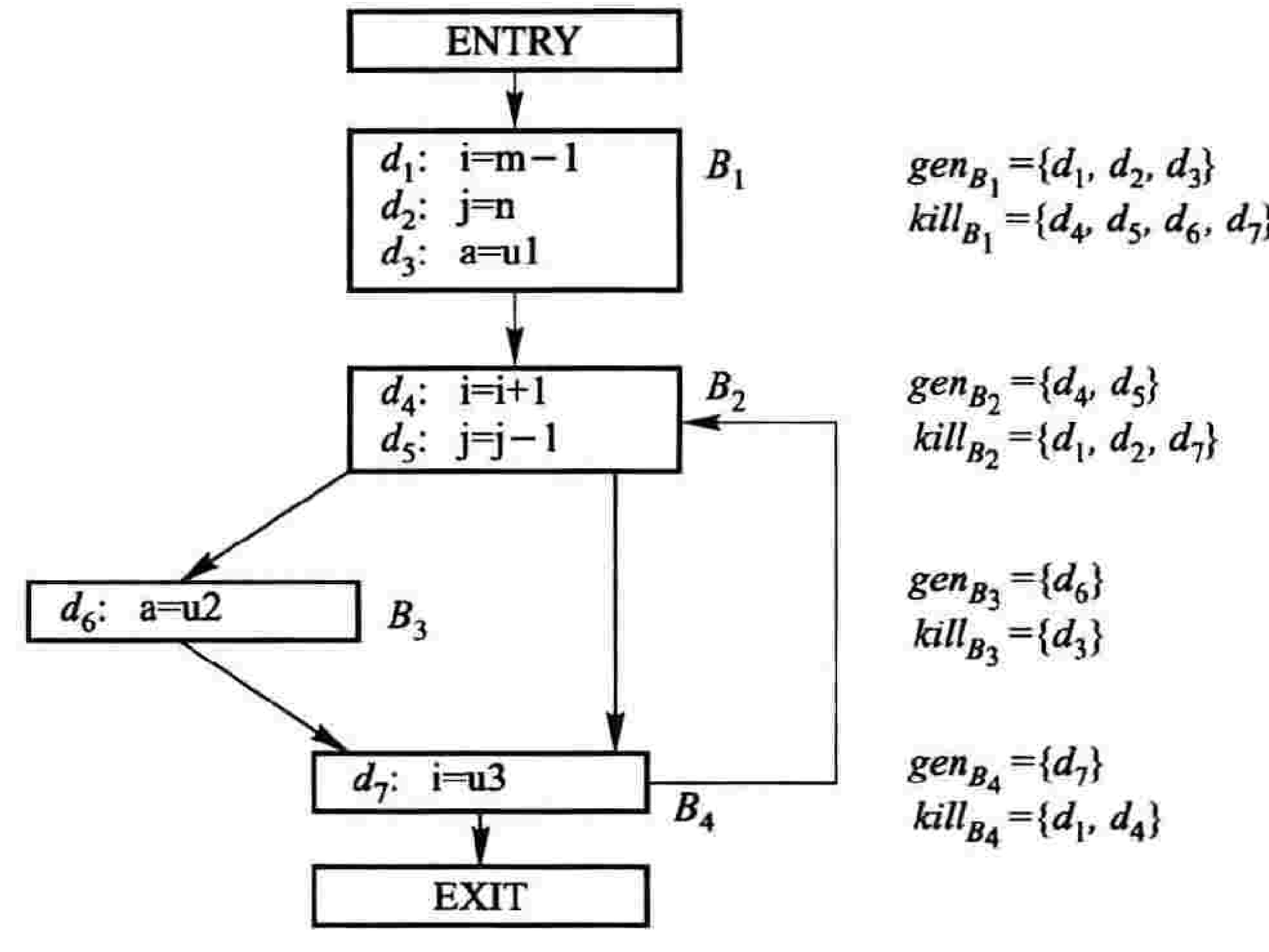


图 9.10 说明到达−定值的流图

现在开始建立到达-定值问题的约束等式,首先是迁移等式。由考察一个语句开始。考虑定值

d: u = v+w

“+”经常作为二元算符的代表用于下面的讨论。

该语句产生变量 u 的一个定值 d,注销程序中 u 的所有其他定值,并且维持其他变量的定值没有变化。因此定值 d 的迁移函数可以表达为

$$f_d(x) = gen_d \cup (x - kill_d) \qquad (9.3)$$

其中 $gen_d = \{d\}$,即由该语句产生的定值集合,$kill_d$ 是该程序中 u 所有其他定值的集合。

如 9.2.2 节所讨论的那样,一个基本块的迁移函数由该块中各语句迁移函数复合而成。下面可以看到,(9.3)形式的函数的复合结果仍然是这种形式,称这种形式为“产生-注销形式”。假定有两个函数 $f_1(x) = gen_1 \cup (x - kill_1)$ 和 $f_2(x) = gen_2 \cup (x - kill_2)$,那么

$$\begin{aligned} f_2(f_1(x)) &= gen_2 \cup (gen_1 \cup (x - kill_1) - kill_2) \\ &= (gen_2 \cup (gen_1 - kill_2)) \cup (x - (kill_1 \cup kill_2)) \end{aligned}$$

这个规则可以延伸到一个块包含任意数目的语句上。假定块 B 有 n 个语句,它们的迁移函数分别是 $f_i(x) = gen_i \cup (x - kill_i)$ $(i = 1, 2, \cdots, n)$,那么块 B 的迁移函数可以写成

$$f_B(x) = gen_B \cup (x - kill_B)$$

其中

$$kill_B = kill_1 \cup kill_2 \cup \cdots \cup kill_n$$

并且

$$\begin{aligned} gen_B = gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \\ \cdots \cup (gen_1 - kill_2 - kill_3 - \cdots - kill_n) \end{aligned}$$

于是和一个语句类似,一个块也是产生一组定值并且注销一组定值。*gen* 集合包含该块中所有在该块出口“可见”的定值集合,称它们为**向下暴露**的定值。一个定值在块中向下暴露,是指它没有被该块中随后的定值注销。一个基本块的 *kill* 集合直接就是该块中各语句注销集合的并集。注意,一个定值 d 可能同时出现在一个块 B 的 gen_B 和 $kill_B$ 中。如果是这样,定值 d 在 gen_B 中的事实优先,因为在产生-注销形式中,减去 $kill_B$ 的运算先于和 gen_B 求并集的运算,因此定值 d 一定出现在 f_B 的结果中。

例 9.7 基本块

d_1: a=3

d_2: a=4

的 *gen* 集合是 $\{d_2\}$,因为 d_1 没有向下暴露。*kill* 集合包含 d_1 和 d_2,因为 d_1 注销 d_2,d_2 也注销 d_1。该块迁移函数的结果总会包含定值 d_2。 □

讨论完迁移等式后,下面考虑从基本块之间的控制流推导出来的一组约束等式。因为只要存在一条路径使得一个定值能沿着它到达某个程序点,就称该定值能到达该点,因此只要有控制流边从 P 到 B,那么 $\mathrm{OUT}[P] \subseteq \mathrm{IN}[B]$。另一方面,若不存在任何路径使得一个定值能沿着它到

达某个程序点,就称该定值不能到达该点,因此 IN[B]也没有必要大于它所有前驱块出口的到达-定值集合的并集。于是,取

$$\text{IN}[B] = \cup_{P\text{是}B\text{的前驱}} \text{OUT}[P]$$

是安全的。

这里的集合并算符称为到达-定值的**交算符**(meet operator)。在任何数据流模式中,交算符是用来总结汇聚到同一点的各条路径的贡献,因此本章称它为**汇合算符**。

建立起到达-定值的数据流等式后,下面讨论相应方程组的迭代求解。为方便起见,假定每个流图都有两个空的基本块,一个称 ENTRY,代表流图的起点;另一个称 EXIT,代表流图的终点,见图 9.10。对于到达-定值问题,显然 OUT[ENTRY]是空集。

这样,到达-定值问题由下面的方程组定义:

$$\text{OUT}[\text{ENTRY}] = \varnothing$$

和对所有块 B(ENTRY 除外)有

$$\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$$

$$\text{IN}[B] = \cup_{P\text{是}B\text{的前驱}} \text{OUT}[P]$$

该方程组可以用下面的算法 9.1 求解。该算法所得解是该方程组的最小不动点,也就是该解给各基本块赋予的 IN 和 OUT 值包含于该方程组其他任何解对应的值中。该算法的解是可以接受的,因为对这些 IN 和 OUT 集合中的每个集合来说,它的任何一个定值,肯定到达相应的点。这是所希望的解,因为它不包含任何不能保证到达的定值。

算法 9.1 到达-定值的迭代求解。

输入 一个流图及各块 B 的 $kill_B$ 和 gen_B。

输出 该流图中每个块 B(ENTRY 除外)的 IN[B]和 OUT[B]。

方法 迭代从所有的 OUT[B]都等于空集开始,逐步收敛到所期望的 IN 和 OUT 值为止。因为必须迭代到 IN 收敛(从而 OUT 也收敛),因此可以使用一个布尔变量 exchange 来记录一遍通过这些块时,是否有任何 OUT 发生变化。为简洁起见,本算法和后面描述的一些类似算法都忽略了把握是否发生变化的准确机制。

该算法概述在图 9.11。前两行给各个 OUT 置初值。第(3)行启动迭代到收敛的循环,第(4)到(6)行的内循环把数据流等式应用到除 ENTRY 以外的所有块。 □

直观上可以看出,算法 9.1 传播定值到尽可能远的地方,只要它们没有被注销,因此它模拟程序所有可能的执行。算法 9.1 是终止的,因为在迭代过程中,任何块 B 的 OUT[B]集合不会缩小;一旦某个定值被加入,它就一直在里面。由于一个程序的定值是有限的,最终总有一遍 while 循环没有向任何 OUT 添加任何定值,于是算法终止。这个终止是安全的,因为当 OUT 没有改变时,下一遍的 IN 也不会改变。如果 IN 不变,那么 OUT 也不变,所有以后各遍都不会有任何改变。

(1) OUT[ENTRY] = ∅;
(2) **for**(除了 ENTRY 以外的每个块 B)　OUT[B] = ∅;
(3) **while**(任何一个 OUT 出现变化)
(4)　　**for**(除了 ENTRY 以外的每个块 B) {
(5)　　　IN[B] = $\cup_{P\text{是}B\text{的前驱}}$ OUT[P];
(6)　　　OUT[B] = $gen_B \cup$ (IN[B] $- kill_B$);
(7)　　}

图 9.11　计算到达-定值的迭代算法

流图中的结点数是 while 循环体执行次数的上界。因为如果一个定值到达某点,那么它能够沿着无环路径继续到达其他点,流图的结点数目是该流图中任何一个无环路径的结点数目的上界。while 循环的循环体每执行一次,一个定值就会沿着正在考察的路径至少前进一个结点,并且经常是前进多个结点,这取决于这些结点被访问的次序。

经验表明,对第(4)行 for 循环中的结点适当地排序,可以使 while 循环的平均迭代次数小于 5。定值集合可以用位向量表示,其上的运算可以由位向量的逻辑运算完成。因此算法 9.1 在实际使用中效率很高。

例 9.8　对图 9.10 中的 7 个定值 $d_1, d_2, \cdots, d_7$ 用位向量来表示,从左边开始的第 i 位表示定值 d_i。集合的并集用对应位向量上的逻辑或运算来计算,两个集合 S 和 T 的差($S-T$)由先对 T 的位向量求补,然后和 S 的位向量进行逻辑与运算来完成。

表 9.1 列出了执行算法 9.1 时各个块的 IN 和 OUT 值的变化。初值由加上角标 0 的 OUT $[B]^0$表示,它们由算法 9.1 的第(2)行赋值,都是空集。第一遍执行循环体后的值由 IN $[B]^1$ 和 OUT$[B]^1$ 表示,以此类推。

假定第(4)到(6)行 for 循环的执行按 B_1, B_2, B_3, B_4, EXIT 的次序处理各基本块。当 $B = B_1$ 时,因为 OUT[ENTRY]和 IN$[B_1]^1$ 都是空集,因此 OUT$[B_1]^1$ 等于 gen_{B_1}。该值和 OUT$[B_1]^0$ 有区别,因此知道将还有下一遍。

当 $B = B_2$ 时,

$$\begin{aligned}\text{IN}[B_2]^1 &= \text{OUT}[B_1]^1 \cup \text{OUT}[B_4]^0 \\ &= 111\ 0000 + 000\ 0000 = 111\ 0000 \\ \text{OUT}[B_2]^1 &= gen_{B_2} \cup (\text{IN}[B_2]^1 - kill_{B_2}) \\ &= 000\ 1100 + (111\ 0000 - 110\ 0001) = 001\ 1100\end{aligned}$$

第二遍扫描后,OUT 集合不再有什么变化,因此在第三遍扫描后算法终止,各 IN 和 OUT 的值就是表 9.1 最后两栏的值。 □

表 9.1　IN 和 OUT 的计算

块	OUT$[B]^0$	IN$[B]^1$	OUT$[B]^1$	IN$[B]^2$	OUT$[B]^2$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000

续表

块	$OUT[B]^0$	$IN[B]^1$	$OUT[B]^1$	$IN[B]^2$	$OUT[B]^2$
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111

9.2.4 活跃变量

一些代码改进变换依赖从程序流图逆向计算得到的信息,现在考虑其中的**活跃变量分析**。在活跃变量分析中,对变量 x 和点 p,希望知道 x 的值是否可能在从点 p 开始的某个路径上被引用,如果可能被引用,就说 x 在点 p 是**活跃**的,否则称 x 在点 p 是**死亡**的。

活跃变量信息的一种重要应用就是基本块的寄存器分配,8.3 和 8.4 节对此已有简单的介绍。一个保存在寄存器中的值,如果在该块出口是死亡的,那么离开该块时这个值不必存储。还有,如果所有寄存器都被占用,此时需要释放一个寄存器,则首先选择保存死亡值的寄存器,因为这个值不必存储。

数据流等式的定义仍然直接根据 IN[B]和 OUT[B],它们分别代表基本块 B 入口和出口的活跃变量集合。这些等式也是由先定义单个语句的迁移函数并把它们复合成基本块的迁移函数。对于单个语句 d:u=v+w 来说,它产生 v 和 w 的活跃性,注销 u 的活跃性,因此迁移函数也是产生-注销形式。逆向复合单个语句的迁移函数,得到基本块的迁移函数如下:

(1) use_B 是在块 B 中有引用并且在该引用前 B 中无定值的变量集合,

(2) def_B 是在块 B 中有定值的变量集合。

它们分别对应到达-定值中的 gen_B 和 $kill_B$。

例 9.9 图 9.10 的 B_2 对 i 重新定值前引用 i,对 j 也是这样。因此 $use_{B_2}=\{i,j\}$。另外 B_2 为 i 和 j 定值,因此 $def_{B_2}=\{i,j\}$。 □

根据这两个定义,use_B 中的任何变量在 B 的入口应该是活跃的,而在 def_B 中的变量在 B 的入口应该是死亡的,除非它也出现在 use_B 中。

将 def 和 use 联系到未知的 IN 和 OUT 的方程组定义如下:

$$IN[EXIT]=\varnothing$$

和对所有块 B(EXIT 除外)有

$$IN[B]=use_B\cup(OUT[B]-def_B)$$

$$OUT[B]=\cup_{S是B的后继}IN[S]$$

第一个等式描述了边界条件:没有任何变量在程序终点活跃。第二个等式说,如果一个变量

在块中定值前有引用，或者在该块出口活跃并且没有被该块定值，那么它在该块入口活跃。第三个等式指出，一个变量在块的出口活跃，当且仅当它在该块某个后继块入口活跃。

必须注意活跃性等式和到达-定值等式之间的联系。

(1) 它们都以集合并算符作为它们的汇合算符。因为在这两种数据流模式中，信息都是沿着路径传播，并且仅关心是否有一条具备所关心性质的路径存在，而不是关心某个性质是否在所有路径上都保持。

(2) 活跃性的信息流是逆控制流方向遍历，因为在这个问题中，要保证的是，变量 x 在点 p 之后可能被引用的信息要沿路径逆向传递到点 p 的前驱点，除非 x 正好在这两点之间被定值。

为求解该逆向问题，注意它和到达-定值问题的联系和区别。这里初始化 IN[EXIT]而不是 OUT[ENTRY]，IN 和 OUT 集合的作用相互交换，*use* 和 *def* 分别代替了 *gen* 和 *kill*。同样，方程组的解也不一定唯一，所要的也是最小解。用于求这个最小解的算法本质是算法 9.1 的逆向版本。

算法 9.2 活跃变量的迭代求解。

输入 一个流图及各块 B 的 def_B 和 use_B。

输出 该流图中每个块 B(EXIT 除外)的 IN[B]和 OUT[B]。

方法 执行图 9.12 的程序。 □

```
IN[EXIT] = ∅;
for(除了 EXIT 以外的每个块 B)  IN[B] = ∅;
while(任何一个 IN 出现变化)
    for(除了 EXIT 以外的每个块 B){
        OUT[B] = ∪_{S是B的后继} IN[S];
        IN[B] = use_B ∪ (OUT[B]-def_B);
    }
```

图 9.12 计算活跃变量的迭代算法

9.2.5 可用表达式

如果从流图起点到点 p 的每条路径上都计算 x+y，并且在到达 p 之前的最后一次计算 x+y 后，一直到 p 为止都没有对 x 或 y 赋值，那么称表达式 x+y 在点 p **可用**。

在可用表达式数据流分析模式中，对于单个语句 d:z=x+y 来说，它产生可用表达式 x+y，注销含 z 的可用表达式，因此迁移函数仍然是产生-注销形式。复合单个语句的迁移函数，得到基本块 B 的迁移函数如下。

(1) e_gen_B 是在块 B 中产生的可用表达式集合。

一个基本块产生可用表达式 x+y 是指它计算 x+y，并且随后没有对 x 或 y 的定值。

(2) e_kill_B 是在块 B 中注销的可用表达式集合。

一个基本块注销含 z 的可用表达式，若它有对 z 定值的语句。

注意,可用表达式的产生和注销概念同到达-定值这些概念不完全一样,然而,它们在这里的行为同它们在到达-定值中的行为本质上是一致的。

可用表达式的基本应用是寻找全局公共子表达式。例如,在图 9.13 中,若表达式 4 * i 在块 B_3 的入口可用,那么块 B_3 的 4 * i 是公共子表达式。这有两种可能,一是块 B_2 没有对 i 定值,如图 9.13(a)所示;另一种如图 9.13(b)所示,块 B_2 对 i 定值后又重新计算 4 * i。

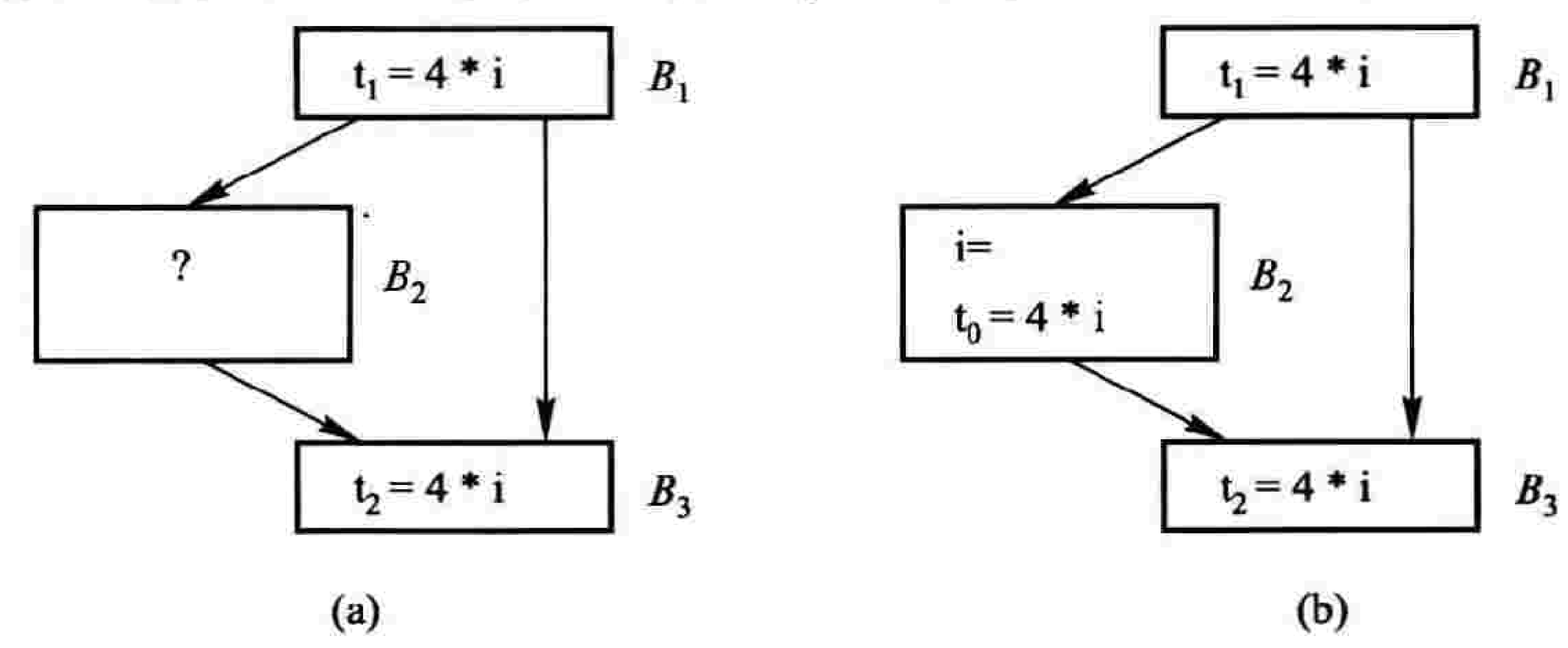


图 9.13 穿越块的公共子表达式

可用类似于计算到达-定值集合的方式寻找可用表达式。假定 U 代表程序中出现在语句右部的所有表达式的集合。对每个块 B,令 IN[B]和 OUT[B]分别是在块 B 入口和出口的可用表达式集合,下列方程组把未知的 IN 和 OUT 同已知的 e_gen 和 e_kill 联系起来:

$$\mathrm{OUT}[\mathrm{ENTRY}] = \varnothing$$

和对所有块 B(ENTRY 除外)有

$$\mathrm{OUT}[B] = e_gen_B \cup (\mathrm{IN}[B] - e_kill_B)$$

$$\mathrm{IN}[B] = \cap_{P\text{是}B\text{的前驱}} \mathrm{OUT}[P]$$

该方程组看起来和到达-定值方程组几乎一样。并且和到达-定值一样,边界条件也是 OUT[ENTRY] = ∅,因为在 ENTRY 的出口不可能有可用表达式。最重要的区别是,汇合算符不是集合并算符而是集合交算符。用集合交运算是合适的,因为当一个表达式在某块所有前驱的出口都可用时,才能说它在该块入口可用。而到达-定值的情况相反,一个定值只要能到达某块一个前驱的出口,则它就能到达该块的入口。

使用∩而不是∪使得可用表达式方程的行为和到达-定值方程的行为有区别。虽然这两种方程组都不是唯一解,但对到达-定值方程组而言,要的是最小集合的解。为了得到这个解,由假设没有任何东西可到达任何地方开始,然后逐步增大到这个解。按这种方式,除非真能找到把 d 传播到 p 的一个路径,否则决不会获得定值 d 能到达点 p 的信息。相反,对可用表达式方程,想要的是最大集合的解,因此从足够大的近似开始,然后逐步缩小到所要的解。

迭代计算从假定任何东西(即集合 U)在任何地方都可用开始,然后逐步删掉那些不可用的表达式。删除一个表达式的依据是,能够找到一条路径,沿着这条路径该表达式是不可用的。最终获得真正的可用表达式集合。

在可用表达式情况下,产生可用表达式精确集合的一个子集是稳妥的,因为利用这些信息是为了用先前计算的值来代替可用表达式的计算,把一个其实可用的表达式当成不可用仅仅是阻止进行这种优化。

例 9.10 把注意力集中在单个基本块,如图 9.14 的块 B_2,来说明 OUT[B_2]的初始近似对 IN[B_2]的影响。令 G 和 K 分别是 $e_gen_{B_2}$ 和 $e_kill_{B_2}$ 的缩写,块 B_2 的数据流方程是:

$$\text{IN}[B_2] = \text{OUT}[B_1] \cap \text{OUT}[B_2]$$

$$\text{OUT}[B_2] = G \cup (\text{IN}[B_2] - K)$$

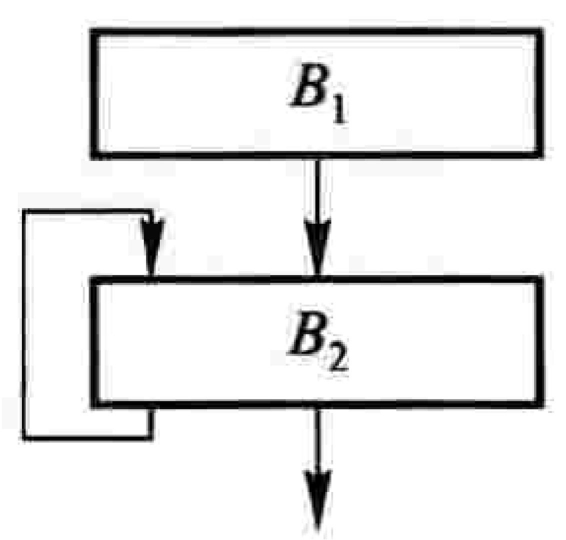


图 9.14 OUT 初值为空集则限制太紧

用 I^j 和 O^j 分别表示 IN[B_2]和 OUT[B_2]的第 j 次近似,这些方程可以重写成递推的形式

$$I^{j+1} = \text{OUT}[B_1] \cap O^j$$

$$O^{j+1} = G \cup (I^{j+1} - K)$$

以 $O^0 = \varnothing$ 开始,则得到 $I^1 = \text{OUT}[B_1] \cap O^0 = \varnothing$。但是,如果从 $O^0 = U$ 开始,则得到 $I^1 = \text{OUT}[B_1] \cap O^0 = \text{OUT}[B_1]$,这才是应该得到的。直观上说,以 $O^0 = U$ 开始,所得到的解更有希望,因为它正确反映了一个事实:OUT[B_1]中没有被块 B_2 注销的表达式在块 B_2 的出口可用。 □

算法 9.3 可用表达式的迭代求解。

输入 一个流图及各块 B 的 e_gen_B 和 e_kill_B。

输出 该流图中每个块 B(ENTRY 除外)的 IN[B]和 OUT[B]。

方法 执行图 9.15 的算法,每步的解释和图 9.11 的类似。 □

```
OUT[ENTRY] = ∅;
for(除了 ENTRY 以外的每个块 B)  OUT[B] = U;
while(任何一个 OUT 出现变化)
    for(除了 ENTRY 以外的每个块 B){
        IN[B] = ∩ P是B的前驱 OUT[P];
        OUT[B] = e_gen_B ∪ (IN[B] - e_kill_B);
    }
```

图 9.15 计算可用表达式的迭代算法

9.2.6 小结

本节讨论了三个重要的数据流问题:到达-定值、活跃变量和可用表达式。如表 9.2 总结的那样,每个问题由数据流值的论域、数据流的方向、迁移函数、边界条件、汇合算符和数据流方程组成。汇合算符一般用 ∧ 表示。

表 9.2 最后一行是迭代算法所用的初值。初值这样选择可以保证迭代算法找到所求解方程组的最精确解。严格地说,这种选择不是数据流问题定义的一部分,因为它是用迭代算法求解时

所需要的。还有其他办法可用来求解这些问题。例如,从本节已经知道怎样从单个语句的迁移函数复合得到基本块的迁移函数,类似方式的复合可用来计算整个过程的迁移函数,或者从过程入口到任何程序点的迁移函数。

表 9.2 三个数据流问题总结

	到达-定值	活跃变量	可用表达式
论域	定值集合	变量集合	表达式集合
方向	正向	逆向	正向
迁移函数	$gen_B \cup (x - kill_B)$	$use_B \cup (x - def_B)$	$e_gen_B \cup (x - e_kill_B)$
边界	$\text{OUT}[\text{ENTRY}] = \varnothing$	$\text{IN}[\text{EXIT}] = \varnothing$	$\text{OUT}[\text{ENTRY}] = \varnothing$
汇合算符($\wedge$)	$\cup$	$\cup$	$\cap$
方程	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \wedge_{P\text{是}B\text{的前驱}} \text{OUT}[P]$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \wedge_{S\text{是}B\text{的后继}} \text{IN}[S]$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \wedge_{P\text{是}B\text{的前驱}} \text{OUT}[P]$
初始化	$\text{OUT}[B] = \varnothing$	$\text{IN}[B] = \varnothing$	$\text{OUT}[B] = U$

9.3 数据流分析的基础

在展示了数据流抽象的一些实例后,现在可以把各种数据流模式作为一个整体来抽象地研究,然后可以形式地回答数据流算法的下列几个基本问题:

(1) 在什么情况下数据流分析中使用的迭代算法是正确的?

(2) 该迭代算法所得解的精度如何?

(3) 该迭代算法是否收敛?

(4) 数据流方程的解的含义是什么?

9.2 节首先针对到达-定值问题,非形式地讨论了上面这些基本问题,然后基于和已经讨论过问题的类似性来解释新问题,而不是针对每个新问题简略地回答这些基本问题。本节给出一种一般方式,它可以对一大类数据流问题回答这些基本问题。首先确定数据流模式被期望的性质,然后证明这些性质蕴涵数据流算法的正确性、精确性和收敛性,最后说明解的含义。这样,为了理解老算法或阐明新算法,只需要证明所定义的数据流问题具有某些性质就行了,然后立即可以回答上面这些基本问题。

对一类数据流模式有一个共同理论框架的概念也具有实现上的意义。在软件设计过程中,该框架能帮助确认可复用的算法组件。这样不仅降低编码的工作量,而且也减少编程错误,因为不需要为雷同的细节重新编码。

一个**数据流分析框架**$(D, V, \wedge, F)$包括:

(1) 数据流分析的方向 D,它可以是正向或逆向。

(2) 包括数据流值的论域 V 和其上汇合算子 $\wedge$ 的一个半格。

(3) 一族从 V 到 V 的迁移函数 F。这族函数必须包括适合于边界条件的函数,这些边界条件是用于流图中像 ENTRY 和 EXIT 这样特殊结点的常函数。

9.3.1 半格

从离散数学课程知道,**半格**是一个集合 V 和一个二元交运算(仍称为汇合运算)$\wedge$,并且该汇合运算满足下面三点性质:

(1) 幂等性:对 V 中所有的 x,$x \wedge x = x$;

(2) 交换性:对 V 中所有的 x 和 y,$x \wedge y = y \wedge x$;

(3) 结合性:对 V 中所有的 x,y 和 z,$x \wedge (y \wedge z) = (x \wedge y) \wedge z$。

通常用二元组$(V, \wedge)$表示半格。半格有顶元,它用 $\top$ 来指称,使得下式成立:

对 V 中的所有 x,$\top \wedge x = x$。

半格可能还有底元,用$\perp$表示,使得下式成立:

对 V 中所有的 x,$\perp \wedge x = \perp$。

从离散数学课程知道,集合 V 上的关系$\leqslant$若满足下面三点性质,则它是一种偏序关系:

(1) 自反性:对 V 中所有的 x,$x \leqslant x$;

(2) 反对称性:对 V 中所有的 x 和 y,如果 $x \leqslant y$ 并且 $y \leqslant x$,那么 $x = y$;

(3) 传递性:对 V 中所有的 x,y 和 z,如果 $x \leqslant y$ 并且 $y \leqslant z$,那么 $x \leqslant z$。

二元组$(V, \leqslant)$称为**偏序集合**。在偏序集合上建立一个关系$<$对今后的使用是方便的,其定义如下:

$x < y$ 当且仅当$(x \leqslant y)$并且$(x \neq y)$。

半格$(V, \wedge)$的汇合运算$\wedge$确定了半格值集 V 上一种偏序$\leqslant$,它的定义如下:

对 V 中所有的 x,y 和 z,$x \leqslant y$ 当且仅当 $x \wedge y = x$。

因为汇合运算$\wedge$具有幂等性、交换性和结合性,因此所定义的关系$\leqslant$具有自反性、反对称性和传递性,是一种偏序关系。其证明如下。

(1) 自反性:由$\leqslant$的定义和$\wedge$的幂等性立即可得 $x \leqslant x$。

(2) 反对称性:由$\leqslant$的定义,$x \leqslant y$ 意味着 $x \wedge y = x$,$y \leqslant x$ 意味着 $y \wedge x = y$,由$\wedge$的交换性,$x = (x \wedge y) = (y \wedge x) = y$。所以,如果 $x \leqslant y$ 并且 $y \leqslant x$,那么 $x = y$。

(3) 传递性:由$\leqslant$的定义,$x \leqslant y$ 并且 $y \leqslant z$ 意味着 $x \wedge y = x$ 并且 $y \wedge z = y$,由$\wedge$的结合性,$(x \wedge z) = ((x \wedge y) \wedge z) = (x \wedge (y \wedge z)) = (x \wedge y) = x$。所以,如果 $x \leqslant y$ 并且 $y \leqslant z$,那么 $x \leqslant z$。

例 9.11 在 9.2 节三种数据流分析模式中使用的汇合运算是集合并和集合交,它们都具有幂等性、交换性和结合性。对于集合并,空集$\varnothing$是顶元,所有元素的集合 U 是底元,因为对 U 的任何子集 x,有$\varnothing \cup x = x$ 并且 $U \cup x = U$。对于集合交,情况反过来,U 是顶元,$\varnothing$是底元。无论将

集合并还是集合交作为汇合运算，相应半格的值论域 V 是 U 的所有子集的集合，即 U 的幂集，用 2^U 来指称。

对 V 中所有的 x 和 y，$x \cup y = x$ 蕴涵 $x \supseteq y$，因此由集合并强加的偏序是集合包含 $\supseteq$。对应地，由集合交强加的偏序是集合包含于 $\subseteq$，这时有较少元素的集合在该偏序中被视为较小的。而对于集合包含作为偏序，有较多元素的集合在该偏序中被视为较小的，这虽然违反直觉，但在上面这些定义下是不可避免的。

如 9.2 节所讨论的那样，数据流方程组通常有很多解，但是按偏序 $\leqslant$ 意义上的最大解是最精确的。例如，在到达-定值分析中，其数据流方程最精确的解是含最少定值的那个解，它对应于这时的汇合运算 $\cup$ 所定义偏序中的最大元素。在可用表达式分析中，最精确的解是含最多表达式的那个解。同样，它也对应于这时的汇合运算 $\cap$ 所定义偏序中的最大元素。 □

在半格的汇合运算和由它所确定的偏序关系之间存在另一种有用的联系。假定 $(V, \wedge)$ 是一个半格，V 中元素 x 和 y 的**最大下界**是 V 中的元素 g，满足

(1) $g \leqslant x$，

(2) $g \leqslant y$，并且

(3) 如果存在元素 z 也满足 $z \leqslant x$ 和 $z \leqslant y$，则 $z \leqslant g$。

显然，x 和 y 汇合运算的结果就是它们唯一的最大下界。为了明白这一点，令 $g = x \wedge y$，则可以看出：

(1) $g \leqslant x$，因为利用汇合运算的结合性、交换性和幂等性，可以得到

$$g \wedge x = (x \wedge y) \wedge x = x \wedge (y \wedge x) = x \wedge (x \wedge y) = (x \wedge x) \wedge y = x \wedge y = g$$

(2) 同理可得 $g \leqslant y$。

(3) 假定元素 z 也满足 $z \leqslant x$ 和 $z \leqslant y$，则有 $z \leqslant g$，因为

$$z \wedge g = z \wedge (x \wedge y) = (z \wedge x) \wedge y = z \wedge y = z，从而\ z \leqslant g$$

因此 g 是 x 和 y 的唯一最大下界。

用格图画出论域 V 对讨论问题是很有帮助的。格图的结点是 V 的元素；如果 $y \leqslant x$，则有从 x 朝下到 y 的有向边。例如，图 9.16 给出一个只有三个定值 d_1，d_2 和 d_3 的到达-定值数据流模式的集合 V。因为 $\leqslant$ 是 $\supseteq$，因此从这三个定值的一个子集到它的每个超集都有一条边。因为 $\leqslant$ 是传递的，因此当图中存在从 x 到 y 的其他路径时，省略从 x 到 y 的边是方便的。例如，虽然 $\{d_1, d_2, d_3\} \leqslant \{d_1\}$，但图中没有画相应的边，因为它可以由通过 $\{d_1, d_2\}$ 的路径来代表。

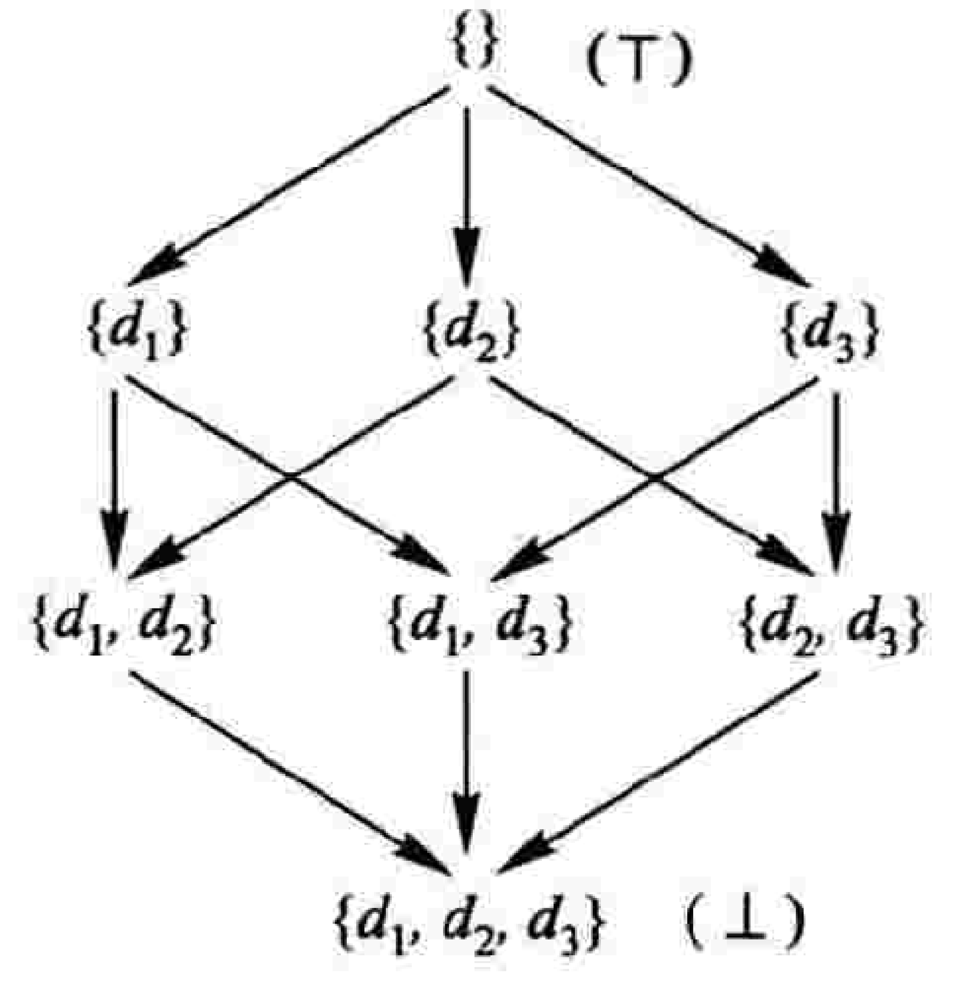


图 9.16 定值子集形成的格

从图 9.16 很容易看出汇合运算的结果。因为 $x \wedge y$ 是 x 和 y 最大下界，因此它总是那个从 x 和 y 向下的路径能够到达并且最高的那个 z。例如在图 9.16 中，$\{d_1\}$ 和 $\{d_2\}$ 的汇合结果是 $\{d_1, d_2\}$，因为这时汇合算符是集合并。顶元 $\top$ 出现在格图的顶部，从它到任何元素都有路径。同样地，底元 $\perp$ 在格图的底部，任何元素

到它都有路径。

本课程中半格概念就够用了,因此没有进一步定义格。在图9.16的例子中,如果把并(join)运算,或者说最小上界运算(对应集合的∩运算)考虑进去,它实际上形成一个格。

图9.16仅涉及三个定值,而一个典型程序的格图可能非常大。数据流值的集合是定值集合的幂集,如果一个程序中有 n 个定值,那么数据流值集包含 2^n 个元素。然而,一个变量的定值是否能到达一个程序点独立于其他变量定值的可达性。因此,可以用从每个变量的一个简单半格构造出的"积半格"(product semilattice)来表示整个定值半格。例如,若图9.16中的定值 d_1, d_2 和 d_3 正好是程序中某个变量的仅有定值,则图9.16是该变量的定值半格。若对另一个变量仅有 d 一个定值的话,那么相应的半格只有两个元素。空集{}是顶元,{d}是底元。

积半格可以形式地定义如下。假定(A, $\wedge_A$)和(B, $\wedge_B$)是半格,它们的**积半格**定义如下:

(1) 积半格的论域是 $A \times B$;

(2) 积半格的汇合运算 $\wedge$ 这样定义:如果(a,b)和(a',b')是积半格论域的元素,那么

$$(a,b) \wedge (a',b') = (a \wedge_A a', b \wedge_B b')。 \quad (9.4)$$

根据 A 和 B 的偏序 $\leqslant_A$ 和 $\leqslant_B$ 来表达积半格上的偏序 $\leqslant$ 是直截了当的:

$$(a,b) \leqslant (a',b') \text{当且仅当} a \leqslant_A a' \text{且} b \leqslant_B b' \quad (9.5)$$

很容易明白(9.5)是从(9.4)得到的,因为若询问在什么情况下有

$$(a,b) \wedge (a',b') = (a,b) [\text{即}(a,b) \leqslant (a',b')]$$

则可知只有在 $a \wedge_A a' = a$ 且 $b \wedge_B b' = b$ 时才会出现,而这两个条件和 $a \leqslant_A a'$ 且 $b \leqslant_B b'$ 一样。

半格的积是可结合的运算,因此可以证明,规则(9.4)和(9.5)可以推广到任意数目的半格。给定半格(A_i, $\wedge_i$), $i=1,2,\cdots,k$,那么按此次序,这 k 个半格的积论域是 $A_1 \times A_2 \times \cdots \times A_k$,其汇合运算定义为

$$(a_1,a_2,\cdots,a_k) \wedge (b_1,b_2,\cdots,b_k) = (a_1 \wedge_1 b_1, a_2 \wedge_2 b_2, \cdots, a_k \wedge_k b_k)$$

其偏序定义为

$$(a_1,a_2,\cdots,a_k) \leqslant (b_1,b_2,\cdots,b_k) \text{当且仅当对所有的} i \text{有} a_i \leqslant_i b_i$$

从研究相关半格的高度可以获悉一些关于一种数据流分析算法收敛速度的信息。在偏序集合(V, $\leqslant$)中,一个上升链是一个序列 $x_1 < x_2 < \cdots < x_n$。一个半格的高度就是其中最长上升链中 $<$ 的个数,或者说等于该链中元素个数减1。例如,在一个有 n 个定值的程序中,到达-定值的高度是 n。

如果半格的高度有限,则证明一个数据流分析迭代算法的收敛是非常容易的。显然,如果半格的值论域有限,则它的高度一定有限;对于值论域无限的半格,其高度有限也是可能的。9.4节仔细讨论的常量传播算法中使用的半格就是一个这样的例子。

9.3.2 迁移函数

在一个数据流分析框架中,迁移函数族 $F: V \to V$ 有下列性质:

(1) F 有一个恒等函数 I,即对 V 中所有的 x,有 $I(x)=x$。

(2) F 封闭于复合,即对 F 中任意两个函数 f 和 g,由 $h(x)=g(f(x))$ 定义的函数 h 在 F 中。

例 9.12 在到达-定值中,F 有恒等函数,其中 *gen* 和 *kill* 都是空集。对复合的封闭性已展示在 9.2.3 节,把它简洁地重复在这里。假如 F 中有两个函数

$$f_1(x)=G_1\cup(x-K_1) \text{ 和 } f_2(x)=G_2\cup(x-K_2)$$

那么

$$f_2(f_1(x))=G_2\cup((G_1\cup(x-K_1))-K_2)$$

等号的右边代数地等价于

$$(G_2\cup(G_1-K_2))\cup(x-(K_1\cup K_2))$$

如果令 $K=K_1\cup K_2$ 并且 $G=G_2\cup(G_1-K_2)$,那么 f_1 和 f_2 的复合 f 为

$$f=G\cup(x-K)$$

f 同 f_1 和 f_2 的形式一样,因此它也是 F 的一个成员。

如果考虑可用表达式,则根据类似于到达-定值的理由同样表明,F 有一个恒等函数并且封闭于复合。 □

为了使数据流分析的迭代算法正常工作,还需要数据流分析框架满足单调性。形式地说,一个数据流分析框架$(D,V,\wedge,F)$是单调的,如果

对 V 中所有的 x 和 y 以及 F 中所有的 f,$x\leqslant y$ 蕴涵 $f(x)\leqslant f(y)$ (9.6)

单调性也可以等价地定义为

对 V 中所有的 x 和 y 以及 F 中所有的 f,$f(x\wedge y)\leqslant f(x)\wedge f(y)$ (9.7)

(9.6)和(9.7)这两个定义的差异较大,因此它们各自都有优于对方的使用场合。

下面给出这两个定义等价的证明概略。首先证明(9.6)蕴涵(9.7)。因为 $x\wedge y$ 是 x 和 y 的最大下界,因此

$$x\wedge y\leqslant x \text{ 并且 } x\wedge y\leqslant y$$

由(9.6),

$$f(x\wedge y)\leqslant f(x) \text{ 并且 } f(x\wedge y)\leqslant f(y)$$

因为 $f(x)\wedge f(y)$ 是 $f(x)$ 和 $f(y)$ 的最大下界,因而得到(9.7)。

再证明(9.7)蕴涵(9.6)。假定 $x\leqslant y$,由定义知 $x\wedge y=x$。由(9.7)知

$$f(x\wedge y)\leqslant f(x)\wedge f(y)$$

由假定得

$$f(x)\leqslant f(x)\wedge f(y)$$

因为 $f(x)\wedge f(y)$ 是 $f(x)$ 和 $f(y)$ 的最大下界,因此 $f(x)\wedge f(y)\leqslant f(y)$。于是

$$f(x)\leqslant f(x)\wedge f(y)\leqslant f(y)$$

因而得到(9.6)。

一个框架经常遵循比(9.7)更强的条件,称之为分配性条件:

对 V 中所有的 x 和 y 以及 F 中所有的 f,$f(x\wedge y)=f(x)\wedge f(y)$

的确,如果 $a=b$,那么由幂等性,$a \wedge b=a$,所以 $a \leqslant b$。因此分配性蕴涵单调性,但反过来不成立。

例 9.13 令 x 和 y 是到达-定值框架中的定值集合,令 f 是对某两个定值集合 G 和 K 的迁移函数 $f=G \cup (x-K)$。可以由检查下面的等式

$$G \cup ((y \cup z)-K)=(G \cup (y-K)) \cup (G \cup (z-K))$$

来验证到达-定值框架满足分配性条件。首先考虑 G 中的定值,它们肯定既在等号左边的集合又在等号右边的集合中。然后考虑不属于 G 的定值,此时可以把 G 从等号的两边都删掉,也就是检验下面等式是否成立

$$(y \cup z)-K=(y-K) \cup (z-K)$$

该等式很容易从集合论的文氏图(Venn diagram)得到。 □

9.3.3 一般框架的迭代算法

现在把算法 9.1 推广到适用于很多种数据流分析问题。

算法 9.4 一般数据流分析框架的迭代求解。

输入 由下面几部分组成的数据流分析框架

(1) 带特殊结点 ENTRY 和 EXIT 的数据流图;

(2) 数据流方向 D;

(3) 数据流值集 V;

(4) 汇合算符 $\wedge$;

(5) 迁移函数集合 F,其中 f_B 是块 B 的迁移函数;

(6) V 中一个常量 v_{ENTRY} 或 v_{EXIT},它们分别代表正向或逆向框架的边界条件。

输出 该流图中每个块 B(ENTRY 或 EXIT 除外)的数据流值集 IN[B]和 OUT[B]。

方法 正向和逆向数据流问题的求解算法分别在图 9.17(a)和图 9.17(b)中。和 9.2 节的那些数据流迭代算法一样,用逐次逼近法计算每个块 IN 和 OUT。 □

重写算法 9.4 的正向和逆向版本,使得实现汇合运算的函数作为参数是可能的,流图本身和边界条件也可以作为参数。按这种方式,编译器的实现者可以避免为编译器优化阶段使用的每一种数据流分析框架存档基本迭代算法。

使用讨论到现在为止的抽象框架,可以证明迭代算法的一些有用性质:

(1) 如果算法 9.4 收敛,则结果是数据流方程组的一个解。

(2) 如果框架单调,则所求得的解是数据流方程组的最大不动点(MFP)。最大不动点解具有这样的性质:

对任何其他解,其 IN[B]和 OUT[B]的值 $\leqslant$ 最大不动点解中相应的值。

(3) 如果框架单调并且半格的高度有限,那么可以保证算法收敛。

现在基于正向框架来讨论这几点。逆向框架的情况本质上是一样的。第一个性质很容易证明。如果 while 循环体的一轮执行使得所有的 OUT 都不发生变化,则所有的 IN 也不再发生变

化，这就意味着方程组得到了满足。

为了证明第二个性质，首先证明对任何 B，在算法迭代时，IN[B]和 OUT[B]的值仅能减小（在≤关系的意义上）。这可以由归纳来证明。

```
(1) OUT[ENTRY] = v_ENTRY;
(2) for(除了 ENTRY 以外的每个块 B)   OUT[B] = ⊤;
(3) while(任何一个 OUT 出现变化)
(4)      for(除了 ENTRY 以外的每个块 B) {
(5)           IN[B] = ∧_{P是B的前驱} OUT[P];
(6)           OUT[B] = f_B(IN[B]);
(7)      }
```

(a) 正向数据流问题的迭代算法

```
(1) IN[EXIT] = v_EXIT;
(2) for(除了 EXIT 以外的每个块 B)   IN[B] = ⊤;
(3) while(任何一个 IN 出现变化)
(4)      for(除了 EXIT 以外的每个块 B) {
(5)           OUT[B] = ∧_{S是B的后继} IN[S];
(6)           IN[B] = f_B(OUT[B]);
(7)      }
```

(b) 逆向数据流问题的迭代算法

图 9.17 迭代算法的正向和逆向版本

归纳基 基本情况是证明在 while 循环第一轮迭代后 IN[B]和 OUT[B]的值不大于初值。这是简单的，因为对不是 ENTRY 的所有块 B，IN[B]和 OUT[B]都被初始化为⊤（在图 9.17(a)中，在 while 循环前增加将所有 IN[B]都初始化为⊤的语句不会影响结果）。

归纳假设 假设经过 k 次迭代后，IN[B]和 OUT[B]的值都不大于 $k-1$ 次迭代后对应的值。

归纳证明 要证明同样情况也出现在 $k+1$ 次迭代和 k 次迭代之间。图 9.17(a)的第(5)行有

$$\mathrm{IN}[B] = \wedge_{P\text{是}B\text{的前驱}} \mathrm{OUT}[P]$$

用记号 $\mathrm{IN}[B]^i$ 和 $\mathrm{OUT}[B]^i$ 分别表示经过 i 次迭代后 IN[B]和 OUT[B]的值。由归纳假设，$\mathrm{OUT}[B]^k \leqslant \mathrm{OUT}[B]^{k-1}$，根据汇合算符的性质，可以知道 $\mathrm{IN}[B]^{k+1} \leqslant \mathrm{IN}[B]^k$。下一步，第(6)行是

$$\mathrm{OUT}[B] = f_B(\mathrm{IN}[B])$$

因为 $\mathrm{IN}[B]^{k+1} \leqslant \mathrm{IN}[B]^k$，根据单调性，有 $\mathrm{OUT}[B]^{k+1} \leqslant \mathrm{OUT}[B]^k$。

注意，IN[B]和 OUT[B]值的任何变化都必须满足方程组，而汇合运算总是返回它们输入的最大下界，迁移函数总是返回和本身块以及输入一致的唯一解。因此，如果迭代算法终止，则所求得解中的值都至少和其他解对应的值一样大，即算法 9.4 的解是方程组的最大不动点。

最后考虑第三点，数据流分析框架单调并且高度有限时迭代算法收敛。因为每个 IN[B]和 OUT[B]值有变化时一定减小，并且如果某一轮迭代中没有变化则算法终止，因此该算法保证在

迭代次数不大于

框架高度×流图中结点数

情况下收敛。

9.3.4 数据流解的含义

从9.3.3节知道,使用迭代算法得到的解是最大不动点;从程序语义的观点考察,这样的解代表什么?为了理解数据流分析框架($D,F,V,\wedge$)的解,首先描述一个框架的理想解是什么样的,然后说明理想解在一般情况下是得不到的。算法9.4的结果是理想解的一种稳妥近似。

不失一般性,仍然假定所讨论的是正向数据流分析框架。考虑一个基本块B的入口,理想解的讨论从找出由ENTRY到B入口所有**可能的**执行路径开始。只有存在程序的某次执行准确跟随某条路径时,这条路径被称为是"可能的"。先前也说明过,流图上的路径并不一定都能成为程序运行时所走的路径,即可能的执行路径集合是执行路径集合的子集。然后,理想解计算每条可能路径末端的数据流值,并将汇合运算施加于这些数据流值,以得到它们的最大下界。这个最大下界就是理想解。因此,程序的任何执行不可能为该程序点产生小于理想解的数据流值。此外,这个界是紧的,由最大下界唯一性知道,不存在较大的数据流值,它可以作为沿流图中每条到达B的可能路径计算出的数据流值的最大下界。

现在可以形式地定义理想解了。对流图中任何块B,令f_B是B的迁移函数。考虑从起点ENTRY到某个块B_k的任意路径

$$P = \text{ENTRY} \to B_1 \to B_2 \to \cdots \to B_{k-1} \to B_k$$

由于程序路径可能有环,因此一个基本块可能在路径P中多次出现。定义P的迁移函数f_P为$f_{B_1}, f_{B_2}, \cdots, f_{B_{k-1}}$的复合。注意,$f_{B_k}$不是该复合的一部分,因为这条路径到达$B_k$的入口为止。这样,执行这条路径建立的数据流值是$f_P(v_{\text{ENTRY}})$,其中$v_{\text{ENTRY}}$是起点ENTRY的迁移常函数的结果。于是,块$B$的理想解是

$$\text{IDEAL}[B] = \wedge_{P\text{是从ENTRY到}B\text{的一条可能路径}} f_P(v_{\text{ENTRY}})$$

根据所讨论框架的格论偏序$\leqslant$,可以认定

(1) 任何大于理想解IDEAL的回答一定是不对的;

(2) 任何小于或等于IDEAL的值是稳妥的,也就是安全的。

直观上讲,在稳妥的值中,越接近IDEAL的值越精确。为什么解必须小于或等于IDEAL?注意,对任何块,任何大于IDEAL的解可以通过忽略某些可能路径而得到,但是可能路径是不能忽略的,因为一般来说,无法保证在这些路径上不存在这样的影响:它们导致基于该解所进行的程序变换无效。反过来说,任何小于IDEAL的解可以看成是包括了某些在流图中不存在的路径,或者虽然存在于流图但在程序执行时决不会走到的路径。该较小解允许对程序所有可能执行都正确的变换,但是禁止了某些基于IDEAL允许的变换。

然而,先前已介绍过,寻找所有可能执行路径是不可判定的,从而必须采取近似的办法。在

数据流抽象中，假定的是流图中的每条路径都会走到，因而为块 B 定义的是所有路径上的汇合 (meet over paths) 解

$$\text{MOP}[B] = \bigwedge_{P\text{是从ENTRY到}B\text{的一条路径}} f_P(v_{\text{ENTRY}})$$

MOP 解考虑的所有路径是 IDEAL 解考虑的所有可能路径的超集。因此，MOP 解不仅汇集了所有可能路径的数据流值，而且还包括了那些不可能被执行路径的数据流值。显然，对所有的块 B，$\text{MOP}[B] \leqslant \text{IDEAL}[B]$，简写成 MOP ≤ IDEAL。

注意，如果一个流图包含环的话，MOP 解所考虑的路径数是无界的。因此 MOP 的定义并没有通向一个直接算法。图 9.17(a)的迭代算法不是首先找出到达一个块的所有路径，然后再使用汇合运算，而是

(1) 访问每个基本块，并且不一定按照程序执行时的次序。

(2) 在每个汇合点，把汇合运算作用于到目前为止得到的数据流值上。其中所用的一些值是在初始化时人工引入的，它们并不代表从程序起点开始的任何执行的结果。

那么，MOP 解和由算法 9.17(a)得到的最大不动点解 MFP 之间具有什么联系？

首先讨论结点的访问次序。在一次迭代中，可能先访问某个基本块而后访问它的某些前驱。如果前驱是 ENTRY 结点，OUT[ENTRY] 已经被适当地初始化为一个常量值。如果前驱是其他结点，它们都被初始化为 $\top$，这是一个不小于最终解的值。根据单调性，通过使用 $\top$ 作为输入得到的解不小于所期望的解。从某种意义上说，可以认为 $\top$ 代表没有任何信息。

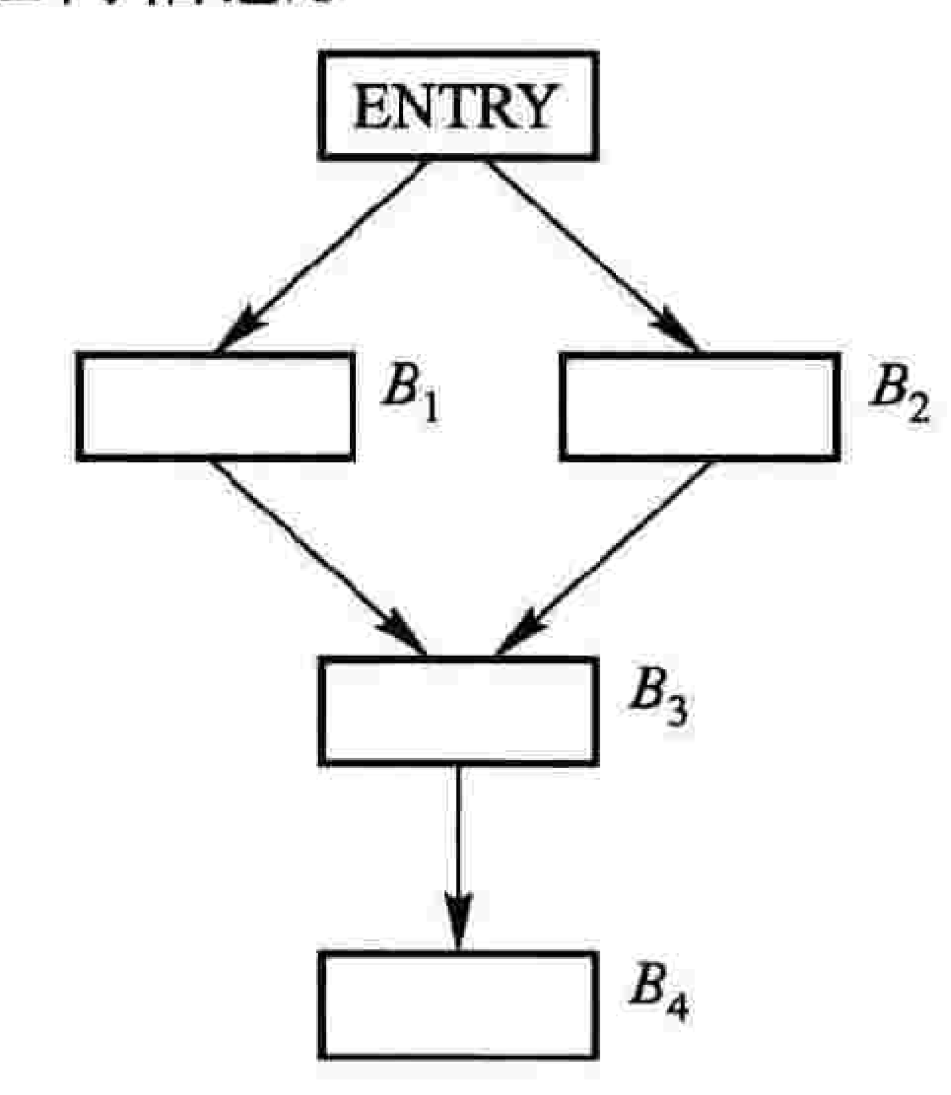


图 9.18 说明路径上较早汇合之影响的流图

较早地使用汇合运算会带来什么影响？考虑图 9.18 的简单例子并假定所关心的是 $\text{IN}[B_4]$ 的值。由 MOP 的定义，有

$$\text{MOP}[B_4] = (f_{B_3} \circ f_{B_1})(v_{\text{ENTRY}}) \wedge (f_{B_3} \circ f_{B_2})(v_{\text{ENTRY}})$$

在迭代算法中，如果按 B_1, B_2, B_3 和 B_4 的次序访问结点，那么

$$\text{IN}[B_4] = f_{B_3}(f_{B_1}(v_{\text{ENTRY}}) \wedge f_{B_2}(v_{\text{ENTRY}}))$$

汇合运算在 MOP 定义的最后使用，但它在迭代算法中使用得较早一些。当数据流分析框架具有分配性时，它们的结果是一样的。如果框架具有单调性而没有分配性时，仍然有 $\text{IN}[B_4] \leqslant \text{MOP}[B_4]$。如果能对所有的 B 证明 $\text{IN}[B] \leqslant \text{MOP}[B]$，则很容易得到对所有的 B，$\text{IN}[B]$ 是稳妥的了。

现在可以简短概述为什么由迭代算法得到的 MFP 解总是安全的。对迭代次数 i 进行归纳可以证明，i 次迭代后得到的值总小于或等于长度不大于 i 的所有路径的汇合运算结果。但是，只有在到达所得结果肯定同无界次迭代的结果一样时，迭代算法才会终止。因此迭代结果不会大于 MOP 的结果，即 MFP ≤ MOP。因为前面已经说明，MOP ≤ IDEAL，因此 MFP ≤ IDEAL。所以由迭代算法得到的 MFP 解是安全(稳妥)的。

9.4 常量传播

9.2 节讨论的数据流模式实际上都是高度有限且具有分配性的框架的简单例子，对它们使用算法 9.4 的正向或逆向版本能够得到 MOP 解。本节仔细考察常量传播框架，它是一种具备一些特殊性质的数据流分析框架。

和已经讨论过的数据流分析问题相比，下面讨论的常量传播框架的区别如下：

(1) 即使是对一个确定的流图，它数据流值的集合也是无界的，并且

(2) 它没有分配性。

常量传播是一个正向数据流问题。

9.4.1 常量传播框架的数据流值

数据流值的集合是一个积半格，对应程序中每个变量，该积中有一个成员。对于单个变量的半格，其论域如下：

(1) 变量的类型所允许的所有常量。

(2) 值 NAC，它表示不是常量。如果一个变量的值被确定为不是常量，则该变量映射到 NAC。被确定为不是常量的具体情况有，变量被赋予一个输入值，或者取决于不等于常量的变量，或者沿到达同一个程序点的不同路径被赋予不同的常量。

(3) 值 UNDEF，它表示没有定义。如果有关一个变量的值无任何可证实的消息，则该变量映射到 UNDEF。具体情况有，该变量没有任何定值到达所关心的程序点。

NAC 和 UNDEF 是不同的，可以说本质上是相反的。一个变量在某程序点的值为 UNDEF，表示没有该变量的定值到达该点；而值为 NAC 表示有定值到达该点，但不能确定它是常量。

图 9.19 表示单个整型变量可能“值”的半格

整型变量的半格见图 9.19。顶元是 UNDEF，底元是 NAC，它们分别是该偏序的最大值和最小值。整数之间无偏序，但是它们都小于 UNDEF 并且都大于 NAC。两个值之间的汇合是它们的最大下界。因此对任何 v，

$$\text{UNDEF} \wedge v = v \text{ 并且 } \text{NAC} \wedge v = \text{NAC}$$

对任何常量 c，

$$c \wedge c = c$$

对任何两个不同的常量 c_1 和 c_2，

$$c_1 \wedge c_2 = \text{NAC}$$

上面讨论的是单个变量的半格。常量传播框架的数据流值是程序中各变量的半格的积,即该框架的数据流值是一个映射,它把程序中每个变量映射到整型变量值半格的一个值。若变量 v 在映射 m 中的值用 $m(v)$ 表示,那么 $m \leqslant m'$ 当且仅当对所有的变量 v 有 $m(v) \leqslant m'(v)$。用另一种方式表示则定义映射上的汇合运算为:$m \wedge m' = m''$,若对所有的变量 v 有 $m''(v) = m(v) \wedge m'(v)$。

9.4.2 常量传播框架的迁移函数

假定每个基本块只有一个语句。因为基本块有多个语句时,其迁移函数可以通过复合其中每个语句的迁移函数来构造。

集合 F 由迁移函数组成,每个迁移函数以 9.4.1 节最后介绍的变量到值的映射为变元,并且返回一个这样的映射。F 包含恒等函数,还包含用于 ENTRY 结点的迁移常函数。对任何映射,该常函数返回 m_0;m_0 的定义是:对所有变量 v,$m_0(v)$ = UNDEF。该边界条件是有意义的,因为在程序的所有语句开始执行前,任何变量都还没有定值。

令 f_s 是语句 s 的迁移函数,令 m 和 m' 代表满足 $m' = f(m)$ 的数据流值。f_s 的描述可以根据 m 和 m' 之间的联系确定:

(1) 如果 s 不是赋值语句,则 f_s 是恒等函数。

(2) 如果 s 对变量 x 赋值,那么对所有 $v \neq x$,$m'(v) = m(v)$;$m'(x)$ 的值由下面几种情况决定:

- 如果语句 s 的右部是一个常量 c,则 $m'(x) = c$。
- 如果语句 s 的右部是 $y+z$,那么

$$m'(x) = \begin{cases} m(y)+m(z), \text{如果 } m(y) \text{ 和 } m(z) \text{ 都是常量值} \\ \text{NAC}, \text{如果 } m(y) \text{ 或 } m(z) \text{ 是 NAC} \\ \text{UNDEF}, \text{否则} \end{cases}$$

9.4.3 常量传播框架的单调性

下面证明常量传播框架是单调的。首先考虑 f_s 在一个变量上的影响。除了 s 的右部是 $y+z$ 种情况以外,对于其他所有情况,f_s 不改变 $m(x)$ 的值,或者改变这个映射到返回一个常量。在这些情况下,f_s 肯定是单调的。

现在看 s 的右部是 $y+z$ 这种情况,f_s 的影响列在表 9.3 中。前两列是 y 和 z 可能的值,最后一列是 x 的值,它们的值按从最大到最小的次序出现在各列或子列中。不难看出 f_s 函数是单调的。

表 9.3 语句 $x=y+z$ 的常量传播迁移函数

$m(y)$	$m(z)$	$m'(x)$
UNDEF	UNDEF	UNDEF
	c_2	UNDEF
	NAC	NAC
c_1	UNDEF	UNDEF
	c_2	c_1+c_2
	NAC	NAC
NAC	UNDEF	NAC
	c_2	NAC
	NAC	NAC

9.4.4 常量传播框架的非分配性

常量传播框架不具备分配性。也就是迭代算法的解 MFP 是安全的,但可能小于 MOP 解。可以用一个例子来证明该框架无分配性。

例 9.14 在图 9.20 的程序中,块 B_1 分别给 x 和 y 赋常量 2 和 3,块 B_2 给它们置的值相反。显然,不管取什么路径,在块 B_3 的出口,z 的值都是 5。然而,算法 9.4 未能发现这个事实。因为该迭代算法在块 B_3 的入口而不是出口使用汇合运算,使得 x 和 y 在块 B_3 入口的值都是 NAC,从而在块 B_3 的出口 z 等于 NAC。这个结果虽不精确,但是安全的。

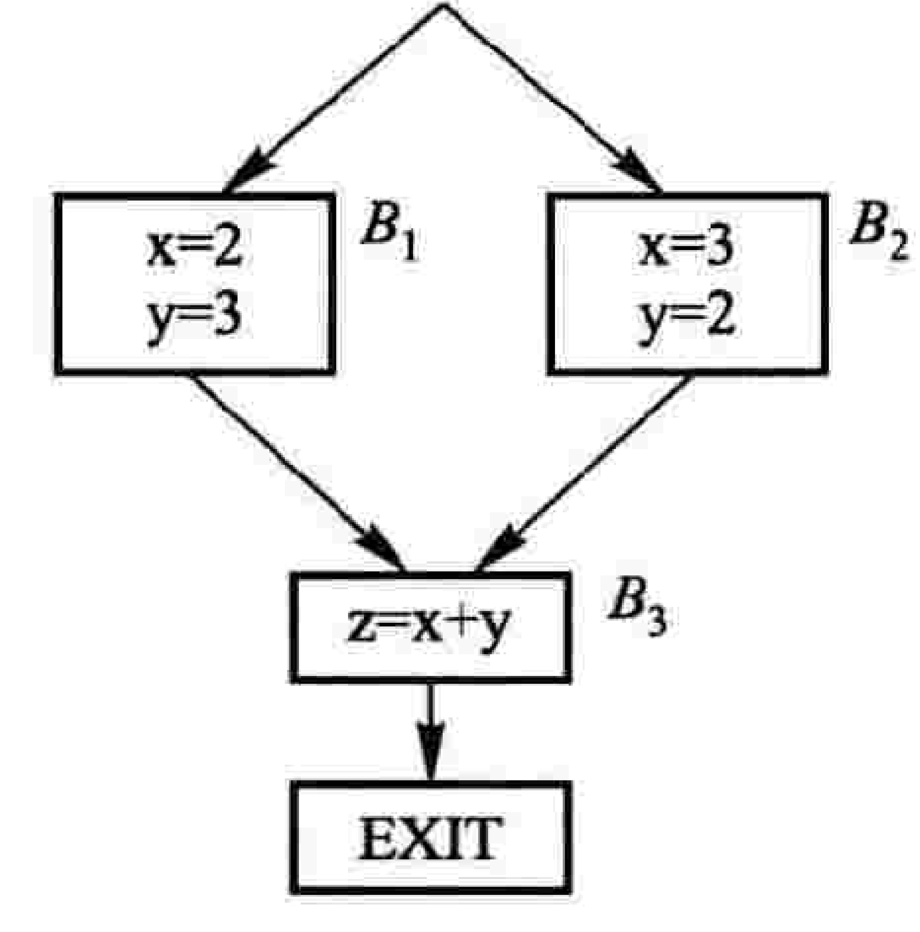


图 9.20 说明常量传播框架没有分配性的例子

算法 9.4 不精确是因为它不能明了 x 等于 2 以及 y 等于 3 和 x 等于 3 以及 y 等于 2 之间的相关性。使用更加复杂的框架有可能明确所有涉及程序变量的表达式之间的相等性,但是代价将大大增加。

理论上讲,可以把这种精确性的丢失归结到常量传播框架的无分配性。令 f_1,f_2 和 f_3 分别代表 B_1,B_2 和 B_3 的迁移函数,表 9.4 体现了

$$f_3(f_1(m_0) \wedge f_2(m_0)) < f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$$

成立,即该框架的无分配性。 □

表 9.4 无分配性的迁移函数的例子

m	$m(x)$	$m(y)$	$m(z)$
m_0	UNDEF	UNDEF	UNDEF
$f_1(m_0)$	2	3	UNDEF
$f_2(m_0)$	3	2	UNDEF
$f_1(m_0) \wedge f_2(m_0)$	NAC	NAC	UNDEF
$f_3(f_1(m_0) \wedge f_2(m_0))$	NAC	NAC	NAC
$f_3(f_1(m_0))$	2	3	5
$f_3(f_2(m_0))$	3	2	5
$f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$	NAC	NAC	5

9.4.5 结果的解释

UNDEF 在算法 9.4 中有两个用途,给 ENTRY 块置初值和在迭代前给程序其他块置初值。这两种情况的含义略有区别。第一种情况表示,在程序开始执行前所有的变量都没有定值;第二种情况表示,在迭代开始前缺乏信息的情况下,把解先近似为顶元 UNDEF。

在迭代过程的结束,变量在 ENTRY 块出口的值仍然都是 UNDEF,因为 OUT[ENTRY]没有改变。UNDEF 也可能出现在程序其他点的迭代结果中,它表示沿任何到达该点的路径上没有对某个变量的定值。注意,按照汇合运算的定义方式,只要存在给某个变量定值的一条路径到达一个程序点,变量在该点的值就肯定不是 UNDEF。这样,如果某变量到达一个点的所有定值都有同样的常量值,即使沿某条到达该点的路径上没有该变量的定值,该变量在该点也被看成常量。

例 9.15 在图 9.21 中,在块 B_2 和 B_3 的出口,x 的值分别为 10 和 UNDEF。因为 UNDEF $\wedge$ 10 = 10,x 在块 B_4 入口的值是 10。于是,块 B_5 中对 x 引用可以用 10 来代替。但是,如果执行路径是 $B_1 \to B_3 \to B_4 \to B_5$ 的话,那么 x 到达 B_5 的值是 UNDEF。因此,用 10 代替 x 似乎不正确。

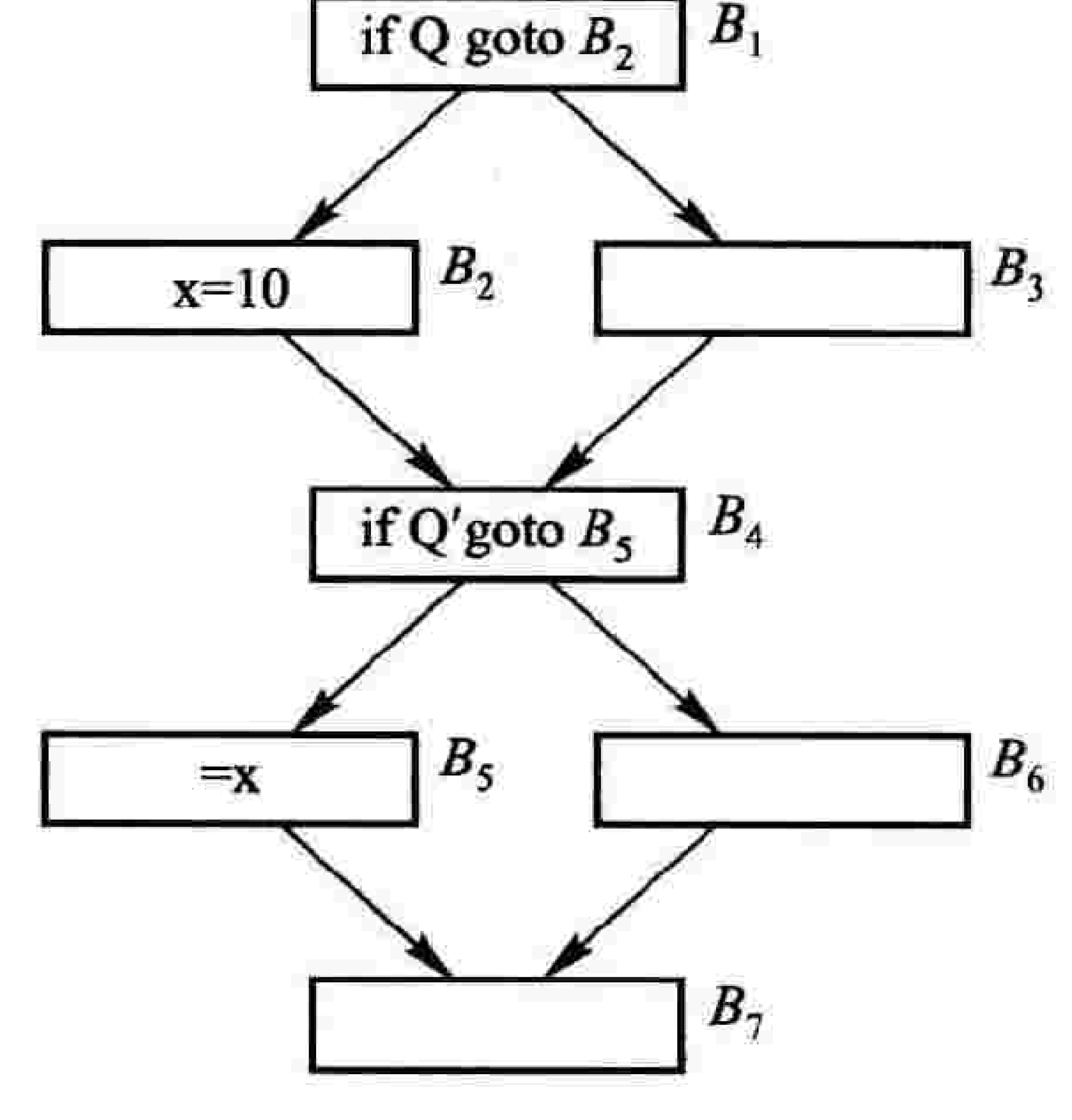


图 9.21 UNDEF 和一个常量的汇合

但是,如果程序正确的话,所有变量一定是先定值后引用,则这条路径在运行时根本不会被执行,因为该路径引用了没有定值的 x。因此在程序正确的情况下,认为 x 在块 B_5 入口的值只能为 10 是合适的。也许程序员能够知道断言 Q 为假而 Q′为真是不可能出现的,因而知道这条路径不可能执行;可惜一

般来说它已经超出了任何数据流分析的判断能力。不过,即使程序不正确,该路径可能执行,选择 10 作为 x 的值并不比 x 是某个随机值更坏。因此,块 B_5 中对 x 引用可以用 10 来代替。 □

从例 9.15 看出,该算法能找出更多的常量。其做法是,为了使程序更有效,它方便地为那些可能没有定值的变量选择一个初值。在多数编程语言中,这样做是合法的,因为允许给没有定值的变量任取一个值。

如果语言的语义要求给所有没有定值的变量一个特定的值,则常量传播问题的形式化需要做相应的修改。如果兴趣在于找出程序中可能没有定值的变量,那么可以形式化另一个数据流分析问题来求解。

9.5 部分冗余删除

本节详细讨论怎样最小化表达式计算的次数。就是考虑一个流图中的所有路径,查看像x+y这样表达式的计算次数。通过考察计算表达式 x+y 之处的四周,并在值得优化时把 x+y 的计算结果保存在一个临时变量中,往往可以减少 x+y 在很多路径上的计算次数,并且没有增加任何路径上 x+y 的计算次数。虽然这种优化有可能使得流图中计算 x+y 的地方增加了,但相对来说这并不重要,只要 x+y 的计算次数减少就可以了。

应用本节介绍的代码变换可以改进代码的性能,因为这里的策略是:一个计算尽量不做,除非它必须得做。每个优化编译器实现了类似这里所描述的变换,虽然它使用的算法可能不像本节算法那样“积极进取”。

本节选择删除冗余以达到最小化表达式计算次数问题来讨论,还有另一个动机,那就是找出流图中计算每个表达式的恰当位置需要四种不同的数据流分析,这有助于增强对数据流分析在编译器中作用的理解。

程序中的冗余以两种形式存在,一种是以公共子表达式的形式出现,另一种是以循环不变表达式的形式出现。冗余也可能是部分的,它可能在一部分路径上出现,但并非在所有路径上都存在。公共子表达式和循环不变表达式可以看成是部分冗余的特殊情况,于是,一个部分冗余删除算法可以衍生到删除各种形式的冗余。

下面首先讨论冗余的不同形式,以建立对问题的直观认识,然后描述广义的冗余删除问题,最后提出算法。该算法特别有意义,因为它涉及求解多个正向或逆向的数据流问题。

9.5.1 冗余的根源

图 9.22 列举了三种形式的冗余:公共子表达式,循环不变表达式和部分冗余表达式。该图同时给出了优化前后的代码。

在图 9.22(a)中,块 B_4 中表达式 b+c 是公共子表达式,它的计算冗余。可以在 B_2 和 B_3 中

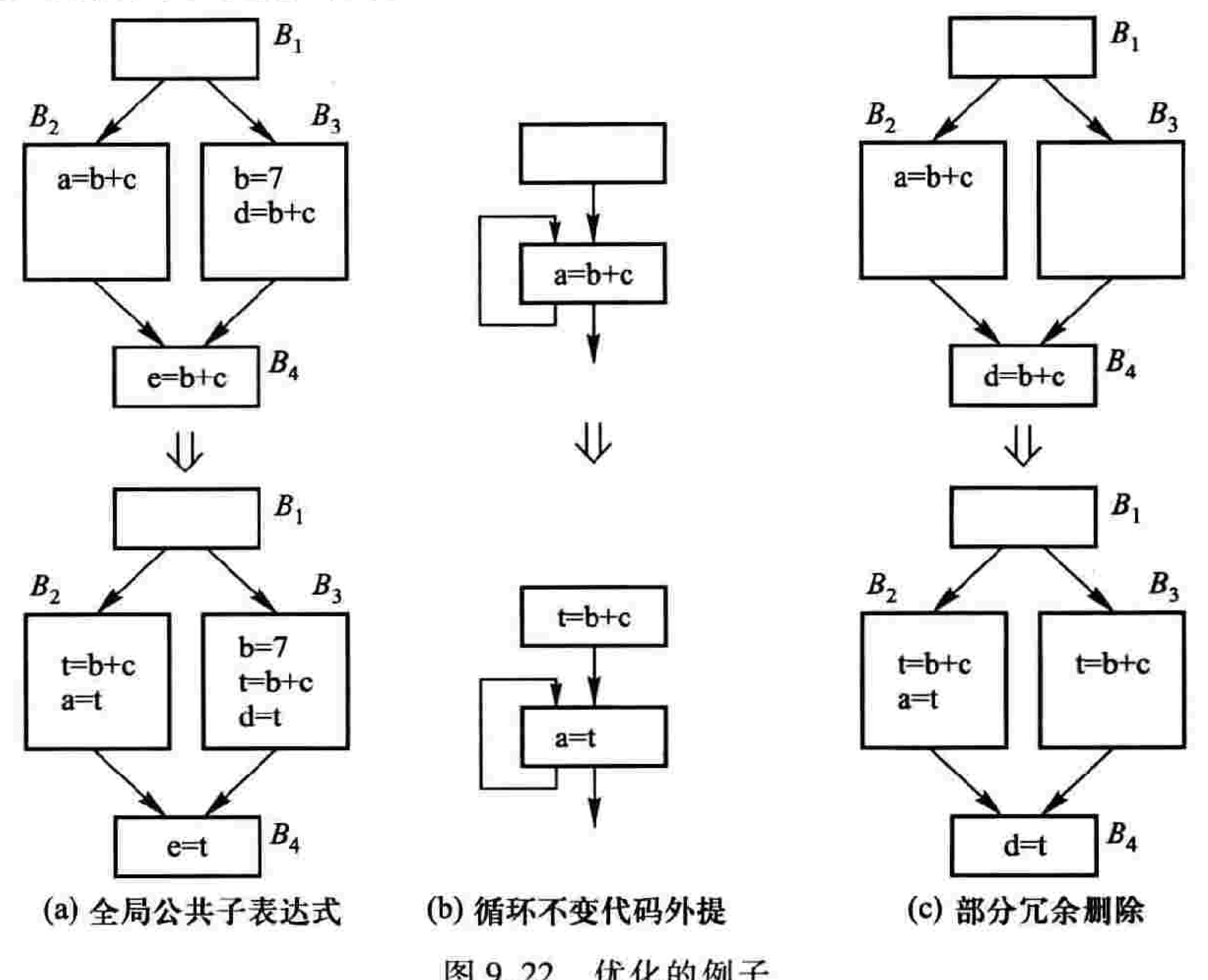
把 b+c 的结果保存在同样的临时变量 t 中，删除 B_4 中的 b+c 计算，把 t 赋给 e。

形式地说，表达式 b+c 在点 p（完全）**冗余**，如果它是 9.2.5 节概念上的可用表达式，也就是说，在到达点 p 的所有路径上都计算 b+c，并且在最后一次计算 b+c 后没有对 b 或 c 重新定值。

注意，使用可用表达式分析只能识别出正文上完全一样的冗余表达式。例如，公共子表达式删除可以识别代码段

```
t1 = b+c
a = t1 +d
t2 = b+c
e = t2 +d
```

的 t1 和 t2 有相同的值，只要 b 和 c 在它们之间没有重新定值。但是它不能识别出 a 和 e 的值也一样。当然，通过反复使用公共子表达式删除，直到没有新的公共子表达式被发现为止，是可以删除这种“深层”的公共子表达式的。



(a) 全局公共子表达式　(b) 循环不变代码外提　(c) 部分冗余删除

图 9.22　优化的例子

图 9.22(b)是循环不变表达式的例子。表达式 b+c 是循环不变计算，只要 b 和 c 在循环中都没有重新定值。可以这样来优化，将 b+c 外提到循环外面计算一次，赋给临时变量 t，循环中的 b+c 用 t 来代替。

循环不变表达式的删除需要将表达式从循环的里面移到循环的外面，而不像公共子表达式删除那样简单地忽略冗余计算，因此这种优化通常被称为**循环不变代码外提**。循环不变代码外

提也可能需要反复使用,因为一旦某个变量被确定为具有循环不变的值,则引用该变量的表达式也有可能成为循环不变的。

图 9.22(c)是一个部分冗余表达式的例子。块 B_4 的表达式 b+c 在路径 $B_1 \to B_2 \to B_4$ 上是冗余的,但在路径 $B_1 \to B_3 \to B_4$ 上不冗余。可以在块 B_3 中添加 b+c 的计算来删除前一条路径上的冗余。这样,同循环不变代码外提一样,部分冗余删除相当于移动了一个 b+c 的计算。

9.5.2 能否删除所有的冗余

是否每条路径上的所有冗余计算都能删除?回答是否定的,除非允许在流图中增加新的块。

例 9.16 在图 9.23(a)的例中,如果程序的执行路径是 $B_1 \to B_2 \to B_4$,则块 B_4 的表达式 b+c 是冗余的。然而,不能简单地把 b+c 的计算移到块 B_3,因为这么做会导致路径 $B_1 \to B_3 \to B_5$ 上增加了一个额外的 b+c 计算。

要想删除这个部分冗余,则 b+c 的计算只能沿块 B_3 到块 B_4 的边放置,具体做法可以像图 9.23(b)那样增加块 B_6,并把 b+c 计算放置在块 B_6 中。 □

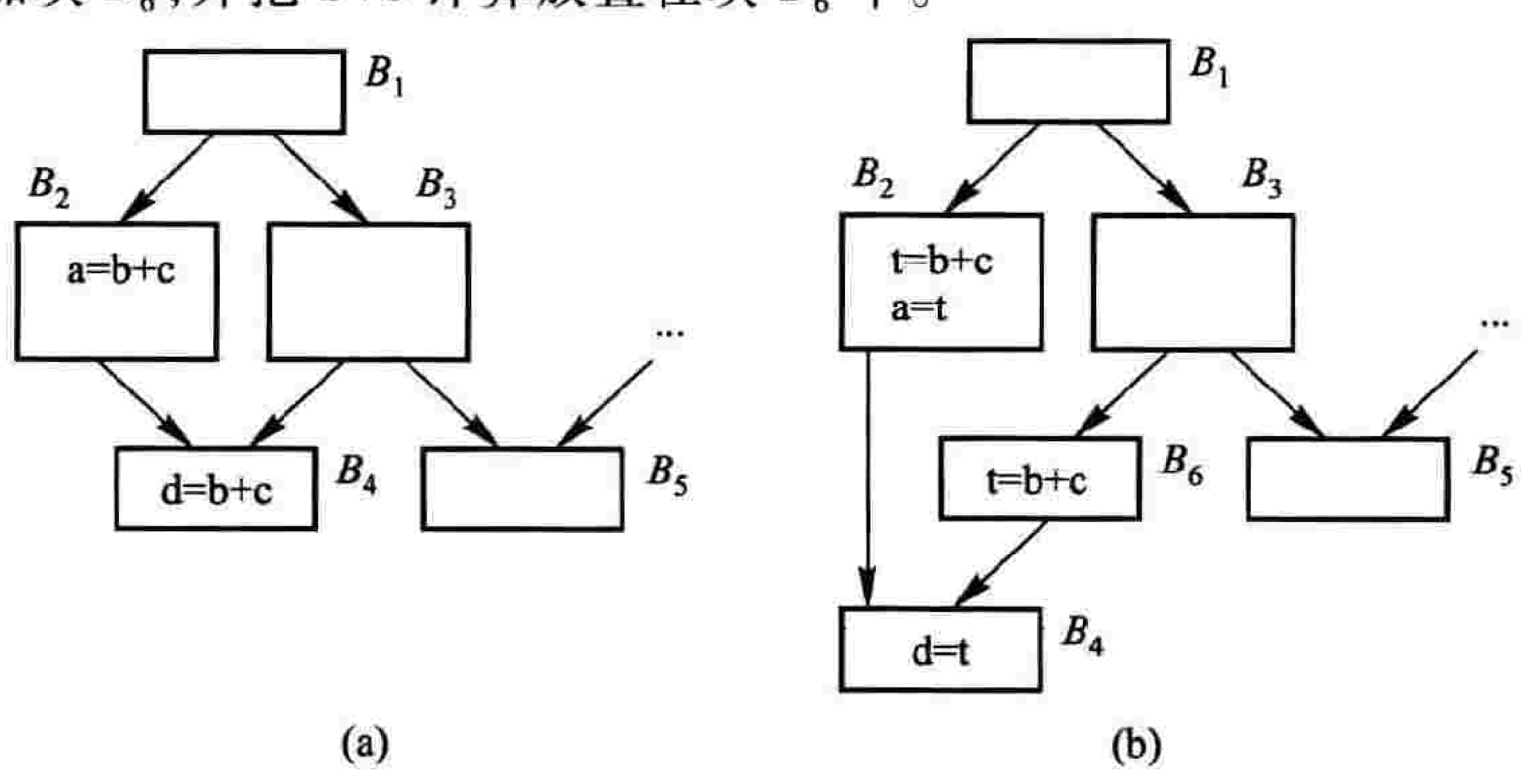


图 9.23 $B_3 \to B_4$ 是一条关键边

流图中的一条边,若其源结点有多个后继结点并且其目的结点有多个前驱结点,则称它为**关键边**,例如图 9.23(a)块 B_3 到块 B_4 的边。通过在关键边的地方增加新块,总能适应放置表达式的需求。

但是增加新块并不是删除所有冗余计算的充分条件。如下面例 9.17 所示,有时需要复制代码以分离发现冗余的路径。

例 9.17 在图 9.24(a)的例中,表达式 b+c 在路径 $B_1 \to B_2 \to B_4 \to B_6$ 上被冗余地计算。期望的优化是将这条路径上块 B_6 的冗余计算 b+c 删除,而仅在路径 $B_1 \to B_3 \to B_4 \to B_6$ 上放置一个b+c 计算。然而,在路径 $B_1 \to B_3 \to B_4 \to B_6$ 的任何地方放置 b+c 计算,都会改变其他某条路径上出现的计算。为了做到不影响其他路径,需要复制块 B_4 和 B_6,使得块 B_4 和 B_6 通过 B_2 到达,而它们

的复制块 B_4' 和 B_6' 通过 B_3 到达,像图 9.24(b)那样。 □

因为路径数同程序中分支数呈指数关系,删除所有的冗余可能导致优化后的代码规模大大增加。因此,以下仅考虑增加基本块的冗余代码技术,而不考虑复制基本块的情况。

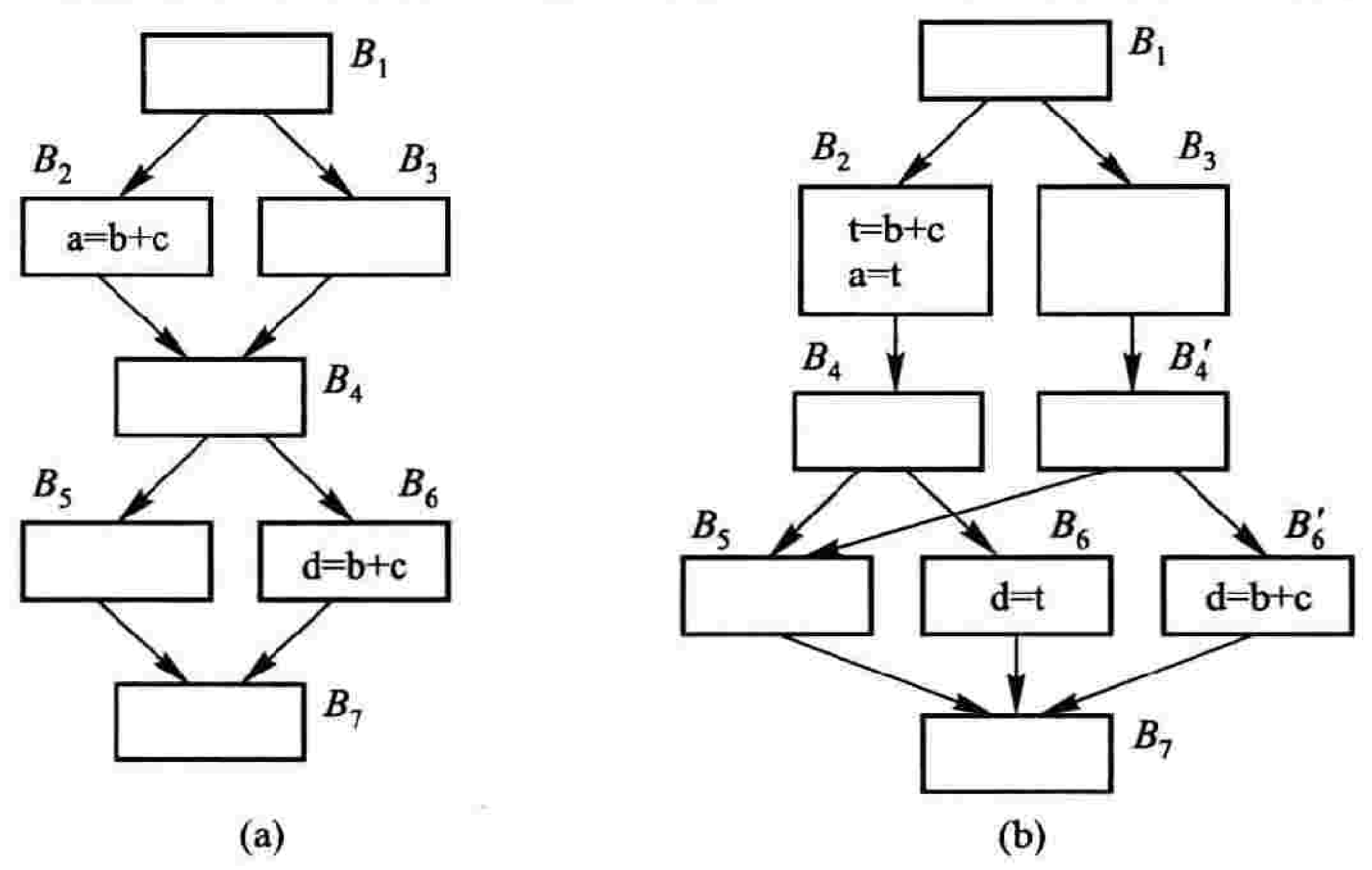


图 9.24 复制代码以删除冗余

9.5.3 惰性代码移动问题

实施部分冗余删除算法,使优化后的程序具有下列性质是可能的:

(1) 不需要通过复制代码就可以删除的表达式冗余计算都被删除,

(2) 优化后的程序不会执行不出现在原来程序中的任何计算,

(3) 表达式的计算尽可能推迟。

最后一个性质是重要的,因为被用来避免冗余计算的表达式的值通常保存在寄存器中,一直到它最后一次被使用。尽可能延迟一个值的计算就是最小化它的生存期——从该值的生成到它最后一次被引用的期间,因而也就是最小化它占用寄存器的时间。这种兼顾尽可能延迟计算的删除部分冗余优化被称为**惰性代码移动**。

为建立对这个问题的直观认识,首先讨论怎样推理有关单个表达式沿单条路径的部分冗余。为方便起见,在该讨论的其余部分假定每个语句构成一个只包含其自身的基本块。在讨论之前,再次认识一下完全冗余和部分冗余。

块 B 的表达式 e 是完全冗余的,如果沿所有到达块 B 的路径,e 都被求值,并且 e 的运算对象随后都没有被重新定值。令 S 是这样的基本块集合,其中每个块都含表达式 e,它们致使 B 中的 e 冗余。S 中各块出边的集合一定形成一个**割集**,如果将该割集删除,则从程序起点到 B 的路径全部被割断。而且,从 S 中任何块到 B 的路径都没有重新给 e 的运算对象定值。

如果块 B 的表达式 e 仅是部分冗余,则惰性代码移动算法试图通过在流图中放置该表达式

的副本致使 B 中的 e 完全冗余。如果这种尝试成功,则修改后的流图也存在基本块集合 S,每个块都包含表达式 e,并且它们的出边构成程序起点和 B 之间的一个割集。像完全冗余情况一样,从 S 中任何块到 B 的路径都没有重新给 e 的运算对象定值。

9.5.4 预期表达式

为了保证没有额外的运算被执行,需要对被添加的表达式强加一个约束。一个表达式的副本只能放置在该表达式被预期的程序点。表达式 b+c 在点 p 被认为是**预期的**,如果所有从点 p 开始的路径上最终都从该点可用的 b 和 c 的值计算表达式 b+c 的值。

现在讨论怎样删除一条无环路径 $B_1 \to B_2 \to \cdots \to B_n$ 上的部分冗余。假定表达式 e 仅在块 B_1 和 B_n 计算,并且 e 的运算对象在这条路径上没有重新定值。这条路径上有入边进入,也有出边离开。e 在块 B_i 的入口不是所期望的,当且仅当存在离开 $B_j(i \leqslant j \leqslant n)$ 的出边,它通向一条不使用 e 值的路径。于是,预期约束限制了表达式能放置的最早位置。

如果 e 在 B_i 的入口是可用的或者是预期的,那么可以组成包含边 $B_{i-1} \to B_i$ 的一个割集,并且该割集致使 B_n 的 e 是冗余的。如果 e 在 B_i 的入口是预期的但不是可用的,则必须沿 B_i 的入边放置 e 的副本。

由于流图中通常存在满足所有要求的若干个割集,因此放置表达式 e 副本的位置也有多种选择。在上面所述方式中,e 沿正在关注路径的入边放置,它保证没有引入冗余,并且 e 副本的计算已经尽量靠近引用。注意,放置的这些 e 副本本身可能成为程序中其他路径上 e 的部分冗余。这样的部分冗余可以由进一步移动这些计算来删除。

总的来说,表达式的预期用来约束表达式的最早放置,表达式的放置不能再提早到它不具备预期性质的位置。表达式的放置越提早,则可以删除的冗余可能越多。在删除同样冗余的所有放置中,最迟计算该表达式的位置是保存该表达式值的寄存器之生存期最短的位置。

9.5.5 惰性代码移动算法

由上面的讨论可构建一个四步算法。第一步使用预期来决定表达式可以放置在什么地方。第二步在那些能删除尽可能多的冗余计算并且不必复制代码也不会引入任何多余计算的割集中,找出最早的割集。该步把计算放置在它们的值最早被预期的程序点。第三步将割集尽量朝下推,一直到再推就会改变程序语义或引入冗余为止。第四步是一遍简单的扫描,通过删除对仅引用一次的临时变量的赋值来整理代码。每步的完成都需要数据流分析,第一和第四步是逆向数据流问题,第二和第三步是正向数据流问题。

1. 预处理步

下面给出完整的惰性代码移动算法。为了使算法简洁,假定最初每个语句构成一个只包含其自身的基本块,并且只在块的入口放置表达式计算的副本。为了保证这种简化不会降低这种

技术的效力,将新基本块添加到一条边的源结点和目的结点之间,如果该目的结点的前驱多于一个。这样做明显考虑了程序中所有的关键边。

每个块的语义用两个集合来抽象,e_use_B 是 B 中计算的表达式集合,e_kill_B 是 B 注销的表达式集合,即程序中至少有一个运算对象在 B 中定值的表达式。下面的例 9.18 用来贯穿该算法所涉及的四种数据流分析的讨论,这四种数据流分析的定义列在表 9.5。

表 9.5 部分冗余删除中的四个数据流问题

参数	预期表达式	可用表达式
论域	表达式集合	表达式集合
方向	逆向	正向
迁移函数	$f_B(x)=e_use_B \cup (x-e_kill_B)$	$f_B(x)=(anticipated[B].in \cup x)-e_kill_B$
边界	$IN[EXIT]=\varnothing$	$OUT[ENTRY]=\varnothing$
汇合算符	$\cap$	$\cap$
方程	$IN[B]=f_B(OUT[B])$ $OUT[B]=\wedge_{S是B的后继} IN[S]$	$OUT[B]=f_B(IN[B])$ $IN[B]=\wedge_{P是B的前驱} OUT[P]$
初始化	$IN[B]=U$	$OUT[B]=U$
参数	可延迟表达式	引用表达式
论域	表达式集合	表达式集合
方向	正向	逆向
迁移函数	$f_B(x)=(earliest[B] \cup x)-e_use_B$	$f_B(x)=e_use_B \cup (x-latest[B])$
边界	$OUT[ENTRY]=\varnothing$	$IN[EXIT]=\varnothing$
汇合算符	$\cap$	$\cup$
方程	$OUT[B]=f_B(IN[B])$ $IN[B]=\wedge_{P是B的前驱} OUT[P]$	$IN[B]=f_B(OUT[B])$ $OUT[B]=\wedge_{S是B的后继} IN[S]$
初始化	$OUT[B]=U$	$IN[B]=\varnothing$

例 9.18 在图 9.25(a)中,表达式 b+c 出现 3 次。因为块 B_9 是循环的一部分,该表达式可能计算许多次。块 B_9 的这个计算不仅是循环不变计算,它也是一个冗余表达式,因为它的值在块 B_7 中已经使用。在本例中,b+c 仅需要计算两次,一次在块 B_5,另一次在块 B_2 到块 B_7 的路径上。惰性代码移动算法把 b+c 的计算放置在块 B_4 和 B_5 的入口。 □

2. 预期表达式

在图 9.25(a)中,所有在入口预期表达式 b+c 的基本块都用浅灰色表示,即 b+c 被块 B_3,B_4,B_5,B_6,B_7 和 B_9 预期。它在块 B_2 入口不是预期的,因为 c 的值在该块中重新计算,从而在块 B_2

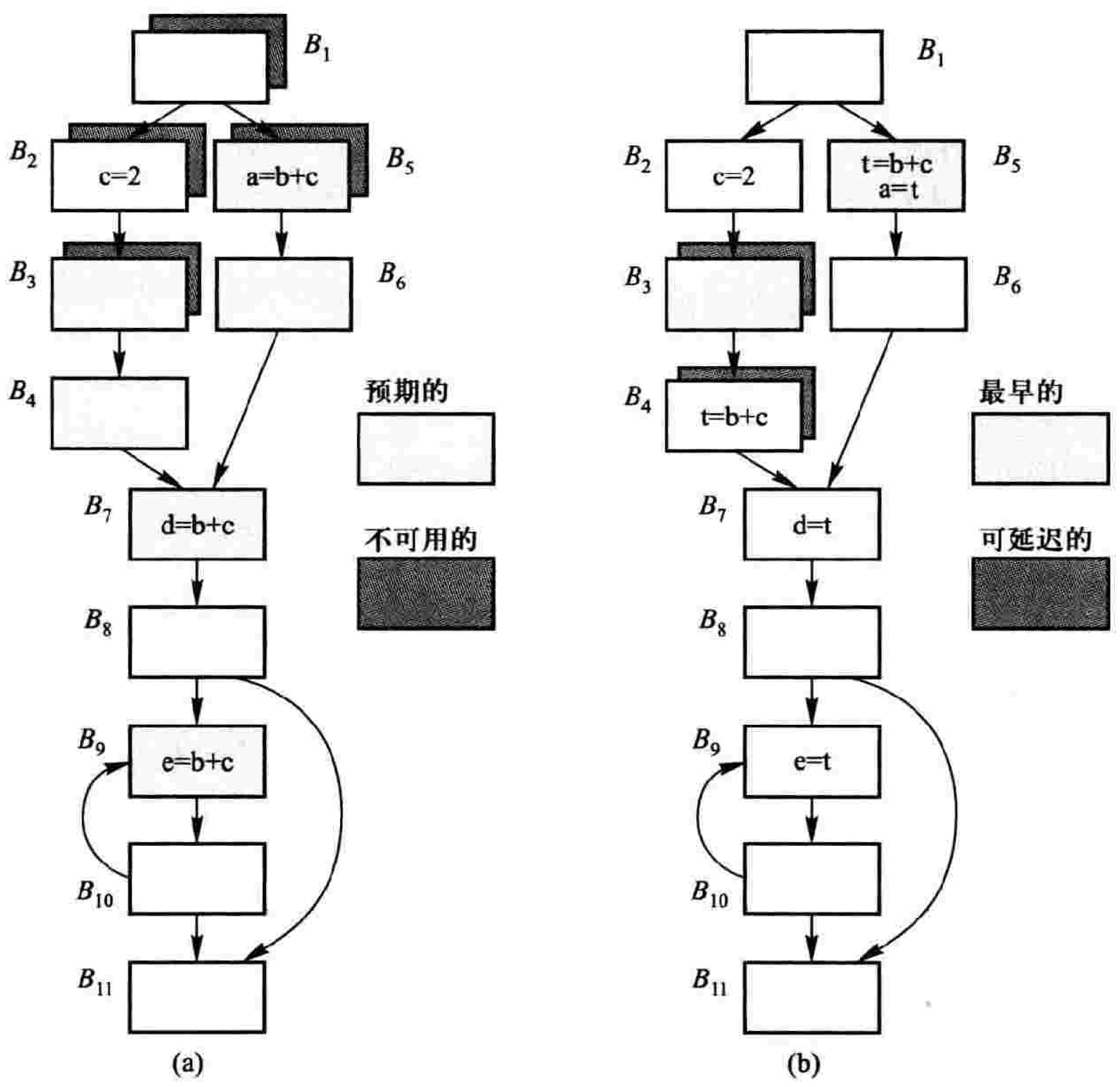


图 9.25　例 9.18 的流图

入口如果计算 b+c 的话,其值是不能在任何路径上使用的。b+c 在块 B_1 入口也不是预期的,因为 c 在此处的值是不能沿从 B_1 经 B_2 的路径继续被引用(虽然 b+c 沿路径 $B_1 \rightarrow B_5 \rightarrow B_6$ 可以被引用)。类似地,该表达式在块 B_8 的入口不是预期的,因为有从 B_8 到 B_{11} 的分支。一个表达式的预期性可以沿路径摆动,这可以用路径 $B_7 \rightarrow B_8 \rightarrow B_9$ 来说明。

预期表达式问题的数据流方程展示在表 9.5 的"预期表达式"列。它是逆向分析的。块 B 出口的预期表达式,如果它不在 e_kill_B 中,则也是 B 入口的预期表达式。还有,块 B 产生 e_use_B 中表达式的新的预期。在程序的终点,没有任何表达式是预期的。因为兴趣在于找到沿每条随后的路径都被预期的表达式,所以汇合算符是集合交。这样,除了 EXIT 外,其他块必须被初始化为全集 U,就像 9.2.5 节讨论可用表达式那样。

3. 可用表达式

在第二步的结尾,表达式的副本将被放置在该表达式最早被期望的程序点。要做到这一点,需要定义可用表达式数据流问题。一个表达式在点 p **可用**,如果在到达点 p 的所有路径上它都被期望。这里定义的问题和 9.2.5 节描述的可用表达式类似,区别在于迁移函数。一个表达式

在一个基本块的出口可用，如果该表达式在入口可用或者在入口的预期表达式集合中(即如果选择在这里计算它，则它就成为可用的了)，并且没有被该基本块注销。

可用表达式的数据流方程展示在表9.5的“可用表达式”列。为了避免和IN混淆，引用先前分析结果时，在先前分析名字后面加上“[B].in”。

按照最早放置策略，在块B放置的表达式集合由下式定义：

$$earliest[B]=anticipated[B].in-avaiable[B].in$$

即预期表达式中那些尚未可用的表达式。

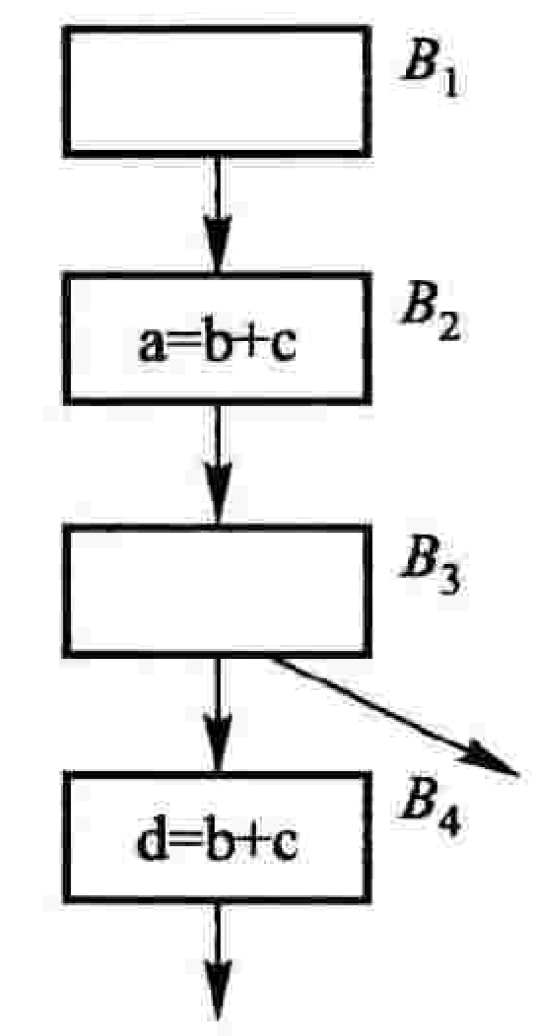


图9.26　例9.19解释可用性的流图

例9.19　在图9.26的流图中，表达式b+c在块B_3的入口不是预期的，但是它在块B_4的入口是预期的。然而，它无须在块B_4中计算，因为由于块B_2，该表达式已经可用。□

例9.20　在图9.25(a)中，带深灰色阴影的块表示表达式b+c在入口不可用，它们是B_1，B_2，B_3和B_5。最早放置由带深灰阴影的灰色块表示，它们是B_3和B_5。例如，b+c在B_4的入口是可用的，因为存在路径$B_1 \to B_2 \to B_3 \to B_4$，并且b+c在$B_3$是被预期的，还有从$B_3$的入口开始，b和c都没有被重新计算。□

4. 可延迟表达式

第三步是在保源程序语义并且最小化冗余的情况下，尽可能延迟表达式的计算。例9.21说明这一步的重要性。

例9.21　在图9.27中，表达式b+c沿路径$B_1 \to B_5 \to B_6 \to B_7$计算两次。b+c在块$B_1$的入口甚至也是预期的。如果按照最早放置策略，就会在B_1计算b+c。若这样，b+c的结果从执行的一开始就要保存，并通过由B_2和B_3构成的循环，一直到使用它的B_7。显然可以缩短该结果的生存期，只要把b+c的计算延迟到B_5的入口和控制流将要从B_4转换到B_7的时候。□

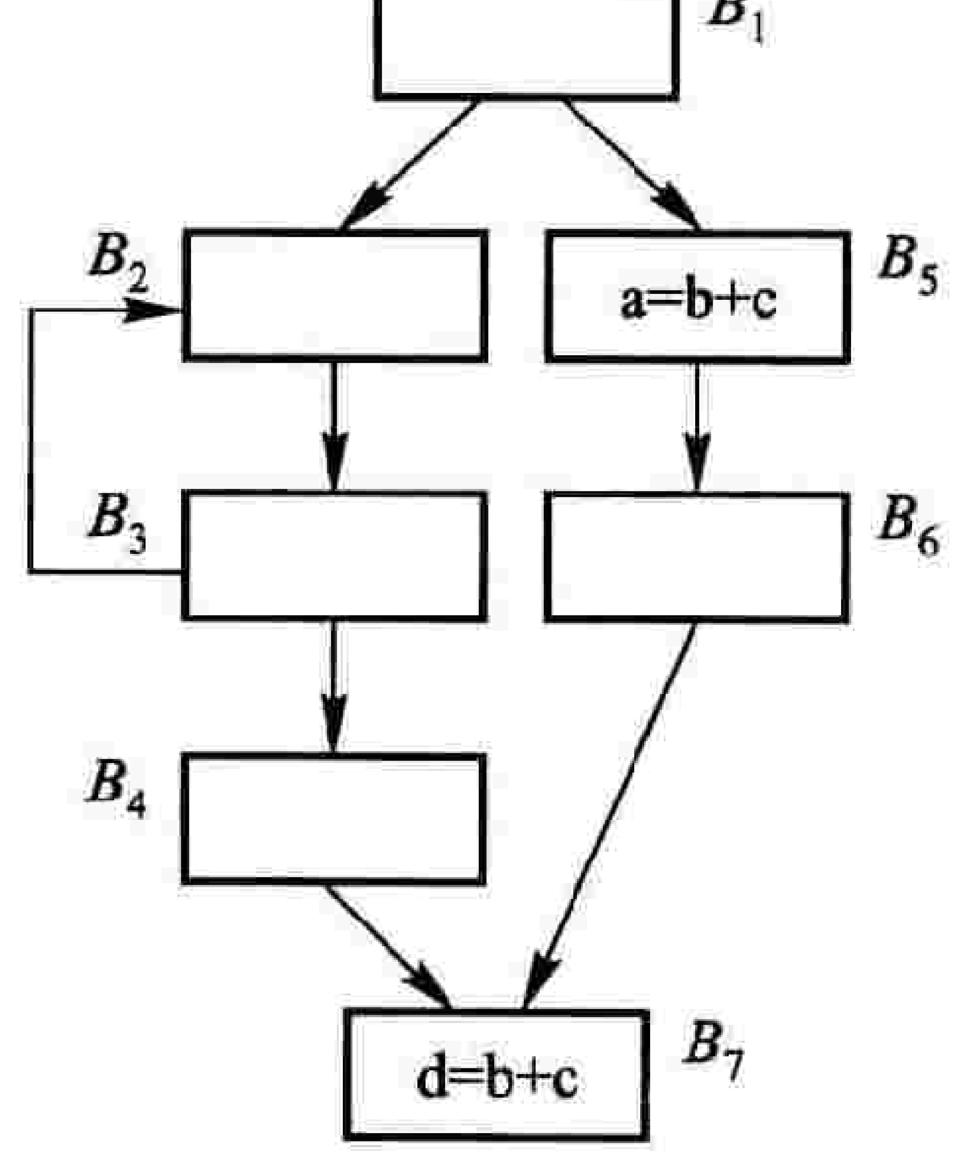


图9.27　例9.21说明延迟表达式的流图

形式地说，表达式x+y可以**延迟**到点p，如果x+y的一个较早放置会遇到以下情况：从起点到p的每条路径都遇到这个放置，并且最后一次遇到该放置之后到p没有x+y的引用。

例9.22　再次考虑图9.25的表达式b+c。b+c的两个最早位置是B_3和B_5，它们是图9.25(a)中仅有的带深灰阴影的灰色块，表示b+c对这些块的入口来说是预期的和不可用的。不能把b+c从B_5延迟到B_6，因为b+c在B_5中有引用，但是可以把它从B_3延迟到B_4。

不能把b+c从B_4再延迟到B_7，因为虽然b+c在B_4中无引用，但是把b+c的计算放置在B_7将导致在路径$B_5 \to B_6 \to B_7$上出现b+c的冗余计算。下面将看到，B_4是计算b+c的最晚位置。

□

可延迟表达式问题的数据流方程展示在表 9.5 的“可延迟表达式”列。它是正向分析的，由于不可能把一个表达式“延迟”到程序的起点，因此 OUT[ENTRY] = ∅。一个表达式可以延迟到块 B 的出口，如果它在 B 入口是可延迟的或者在 $earliest[B]$ 中，并且在 B 中没有引用。一个表达式不可能延迟到一个块的入口，除非这个表达式在该块所有前驱的出口都是可延迟的。因此汇合算符是集合交，并且除了 ENTRY 外，其他块初始化为该半格的顶元——全集。

粗略地说，为尽量延迟，一个表达式放置在从可延迟到不可延迟的边界上。具体地说，一个表达式 e 可以放置在块 B 的入口，只要它在 B 的 *earliest* 或 B 入口的 *postponable* 集合中。除此以外，要想 B 是 e 的可延迟边界，当且仅当下面两条件成立：

(1) e 不在 $postponable[B].out$ 中，换句话说，e 在 e_use_B 中。

(2) e 不能延迟到 B 的一个后继。换句话说，存在 B 的一个后继，使得 e 不在该后继的 *earliest* 集合中，也不在该后继入口的 *postponable* 集合中。

由于该算法预处理步添加了适当的新基本块，因此，如果上面两个条件之一成立，则表达式 e 就放置在 B 的入口。

对于上面可延迟和不可延迟边界的非形式描述，可以给出 *latest* 集合的精确定义：

$$latest[B] = (earliest[B] \cup postponable[B].in) \cap \\ (e_use_B \cup \neg(\bigcap_{S\text{是}B\text{的后继}}(earliest[S] \cup postponable[S].in)))$$

其中¬表示关于该程序计算的所有表达式集合的补集。

例 9.23 图 9.25(b)表明了分析结果。浅灰色块表示它的 *earliest* 集合包含 b+c。深灰阴影表示相应块入口的 *postponable* 集合包含 b+c。因此 b+c 最迟放置在 B_4 和 B_5 的入口，因为

(1) b+c 在 B_4 入口但不在 B_7 入口的 *postponable* 集合中，并且

(2) B_5 的 *earliest* 集合包含 b+c 并且它引用 b+c。

b+c 的值在 B_4 和 B_5 存入临时变量 t，然后 t 在其他地方用来代替 b+c，如图 9.25 中显示的那样。 □

5. 引用表达式

最后，一种逆向分析用来确定临时变量的引用是否超出给它们定值的块。一个表达式在点 p 被引用，如果存在从 p 开始的一条路径，该路径在重新计算该表达式前引用它。这本质上是活跃性分析，不过是面向表达式而不是先前介绍的面向变量。

引用表达式问题的数据流方程展示在表 9.5 的“引用表达式”列。它是逆向的。块 B 出口的引用表达式在 B 入口仍然是引用表达式，只要它不在 *latest* 集合中。一个基本块产生的新引用集合就是 e_use_B 集合。在程序的终点，没有表达式可引用。因为兴趣在于找出在随后任意一条路径上都有引用的表达式，所以汇合算符是集合并。因此，除了 EXIT 以外，其他块初始化为该半格的顶元——空集。

6. 小结

该算法的所有步骤总结在算法 9.5 中。

算法 9.5 惰性代码移动。

输入 一个流图及各块 B 的 e_use_B 和 e_kill_B。

输出 一个被修改后的流图,满足 9.5.3 节介绍的惰性代码移动的三个性质。

方法 按下面八个步骤执行。

(1) 在所有不止一个前驱的基本块的所有入边上插入一个空块。

(2) 按表 9.5"预期表达式"列的定义,找出所有块 B 的 $anticipated[B].in$。

(3) 按表 9.5"可用表达式"列的定义,找出所有块 B 的 $available[B].in$。

(4) 为所有的块 B 计算最早放置:

$$earliest[B] = anticipated[B].in - avaiable[B].in$$

(5) 按表 9.5"可延迟表达式"列的定义,找出所有块 B 的 $postponed[B].in$。

(6) 为所有的块 B 计算最迟放置:

$$latest[B] = (earliest[B] \cup postponable[B].in) \cap$$
$$(e_use_B \cup \neg(\bigcap_{S\text{是}B\text{的后继}}(earliest[S] \cup postponable[S].in)))$$

(7) 按表 9.5"引用表达式"列的定义,找出所有块 B 的 $used[B].out$。

(8) 对程序计算的每个表达式,例如 x+y,执行下面步骤:

- 为 x+y 确定一个新临时变量,例如 t。
- 对满足 x+y 属于 $latest[B] \cap used[B].out$ 的所有基本块 B,在 B 的入口加上 t=x+y。
- 对满足 x+y 属于 $e_use_B \cap (\neg latest[B] \cup used[B].out)$ 的所有基本块 B,把原来的 x+y 都用 t 代替。 □

该算法介绍了以一种统一的算法找出多种不同形式的冗余计算,同时还展示了怎样把多个数据流问题用于找出最佳的表达式放置。

9.6 流图中的循环

到目前为止,尚未把循环作为一类特殊的控制流来处理。然而,循环是非常重要的,因为程序执行的大部分时间消耗在循环上,改进循环性能的优化会对程序执行产生显著影响。因此标识循环并对它们专门处理是至关重要的。

循环也会影响程序分析的运行时间。如果程序不包含任何循环,只要对程序进行一遍扫描就可以回答数据流问题,例如,一个正向数据流问题可以由按拓扑次序访问所有结点一遍来解决。

本节介绍支配结点、深度优先排序、回边、图的深度和可归约性等概念,这些概念用在随后的寻找循环和迭代数据流分析收敛速度的讨论中。

9.6.1 支配结点

流图中结点 d **支配**结点 n,如果从流图起点开始,每条到达 n 的路径都要经过 d。写成 $d\ dom\ n$,称 d 是 n 的支配结点,也称 d 是 n 的**必经结点**。根据这个定义,每个结点支配自身。

例 9.24 考虑图 9.28 的流图,它的起点是 1。起点支配所有结点(这句话对所有流图都是对的)。结点 2 仅支配它自身,因为控制可以沿 1→3 开始的路径到达任何其他结点。结点 3 支配除 1 和 2 以外的所有结点。结点 4 支配除 1、2 和 3 以外的所有结点,因为从 1 出发的所有路径必须由 1→2→3→4 或 1→3→4 开始。结点 5 和 6 分别仅支配自身,因为控制流可以经过这两者之一而跳过另一个。最后,7 支配 7、8、9 和 10;8 支配 8、9 和 10;9 和 10 分别仅支配自身。 □

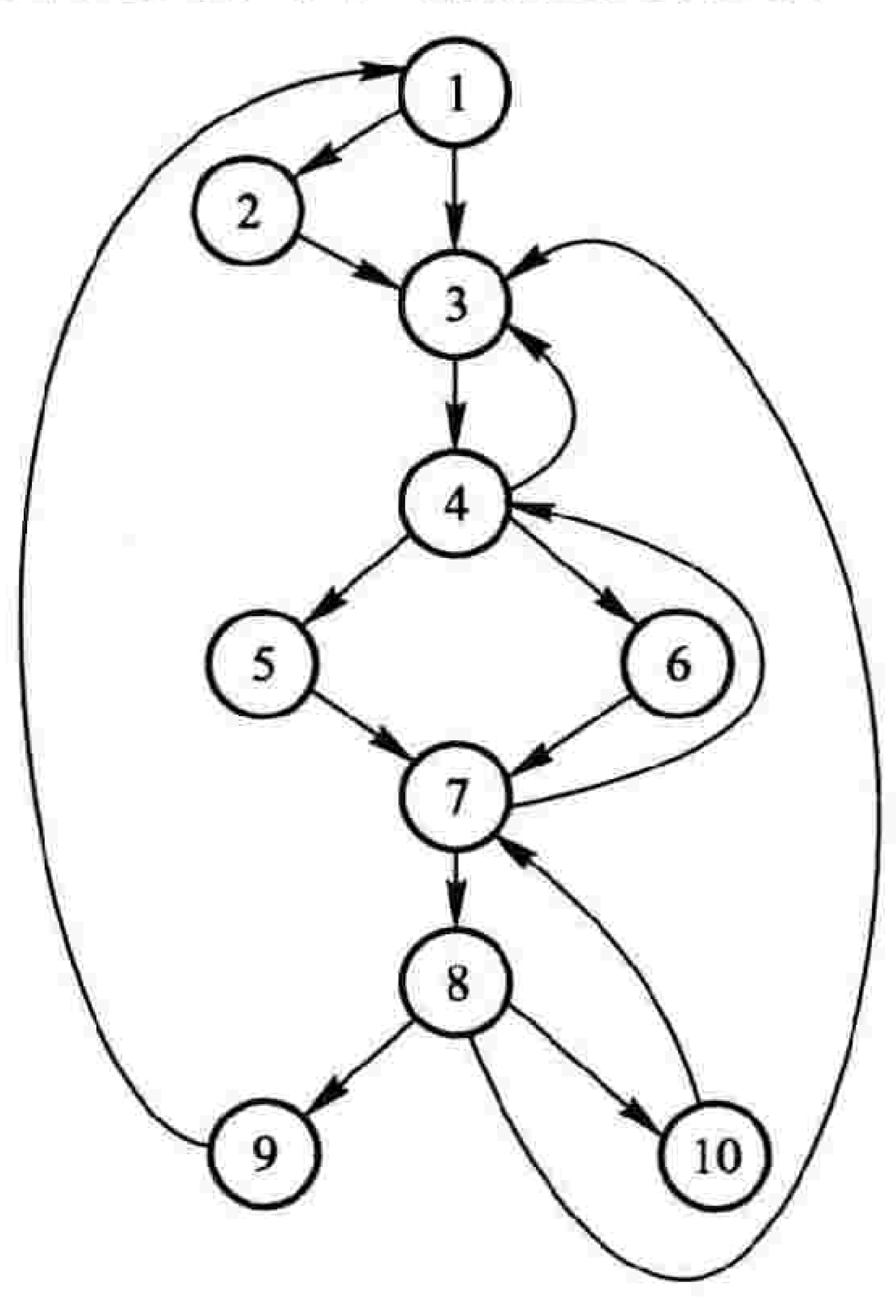


图 9.28 流图

下面介绍一个算法,它计算流图中每个结点 n 的支配集,即 n 的所有支配结点。它依据的原理是,如果 $p_1, p_2, \cdots, p_k$ 是 n 的所有前驱,并且 $d \neq n$,那么 $d\ dom\ n$,当且仅当对所有的 i $(0 \leqslant i \leqslant k)$ 有 $d\ dom\ p_i$。该问题可以形式化为一个正向数据流分析。数据流值是基本块集合。一个结点的支配集是它各前驱支配集的交集再加上它自身,于是汇合运算是集合交。基本块 B 的迁移函数直接把 B 自身加到输入结点集合上。边界条件是:ENTRY 结点支配它自身。最后,除了 ENTRY 以外,其他结点初始化为全集,也就是所有结点的集合。

算法 9.6 计算支配集。

输入 一个流图,其结点集为 N,起点为结点 ENTRY。

输出 N 中所有结点 n 的支配集 $D(n)$。

方法 以表 9.6 所列参数求解数据流问题。对 N 中所有结点,$D(n) = \text{OUT}[n]$。 □

表 9.6 计算支配集的数据流问题

参数	支配集
论域	结点集合 N 的幂集
方向	正向
迁移函数	$f_B(x) = x \cup \{B\}$
边界	$\text{OUT}[\text{ENTRY}] = \{\text{ENTRY}\}$
汇合算符	$\cap$

续表

参数	支配集
方程	$OUT[B]=f_B(IN[B])$ $IN[B]=\wedge_{P\text{是}B\text{的前驱}} OUT[P]$
初始化	$OUT[B]=N$

例 9.25 考虑图 9.28 的流图,并且假定图 9.17(a)程序第(4)到(6)行的 for 循环是按数值序访问结点。令 $D(n)$ 等于 $OUT[n]$。结点 1 是起点,第(1)行赋值语句使 $D(1)=\{1\}$。结点 2 的唯一前驱是结点 1,因此 $D(2)=\{2\}\cup D(1)=\{1,2\}$。结点 3 有前驱 1、2、4 和 8。由于除了结点 1 以外,其他结点都初始化为$\{1,2,\cdots,10\}$,因此

$$D(3)=\{3\}\cup(\{1\}\cap\{1,2\}\cap\{1,2,\cdots,10\}\cap\{1,2,\cdots,10\})=\{1,3\}$$

其余结点的计算如下:

$$D(4)=\{4\}\cup(D(3)\cap D(7))=\{4\}\cup(\{1,3\}\cap\{1,2,\cdots,10\})=\{1,3,4\}$$

$$D(5)=\{5\}\cup D(4)=\{5\}\cup\{1,3,4\}=\{1,3,4,5\}$$

$$D(6)=\{6\}\cup D(4)=\{6\}\cup\{1,3,4\}=\{1,3,4,6\}$$

$$D(7)=\{7\}\cup(D(5)\cap D(6)\cap D(10))$$

$$=\{7\}\cup(\{1,3,4,5\}\cap\{1,3,4,6\}\cap\{1,2,\cdots,10\})=\{1,3,4,7\}$$

$$D(8)=\{8\}\cup D(7)=\{8\}\cup\{1,3,4,7\}=\{1,3,4,7,8\}$$

$$D(9)=\{9\}\cup D(8)=\{9\}\cup\{1,3,4,7,8\}=\{1,3,4,7,8,9\}$$

$$D(10)=\{10\}\cup D(8)=\{10\}\cup\{1,3,4,7,8\}=\{1,3,4,7,8,10\}$$

因为第二次迭代时(图 9.17(a)从第(3)到(6)行的外循环),这些值都不变,因此上述这些值就是该支配集问题的结果。 □

9.6.2 回边和可归约性

一个图的深度优先搜索访问该图所有结点各一次,它从图的起点开始并且尽快地访问离起点尽可能远的结点。深度优先搜索的搜索路线形成深度优先生成树。从数据结构课程知道,先序遍历访问一个结点先于它任何子结点,然后按从左到右的次序递归地访问它的子结点。后序遍历在访问一个结点以前,先按从左到右的次序递归地访问它的子结点。

对于程序流图来说,深度优先排序是一种很重要的排序,它是后序遍历的逆。深度优先排序先访问一个结点,然后从它的最右子结点开始,逐步向左遍历各结点,直到最左子结点。在为流图构建相应的树之前,需要确定流图中每个结点的后继中,哪个被看成该树上它的最右子结点,哪个次之,直到哪个是最左子结点。

例 9.26 图 9.28 流图的一种深度优先表示在图 9.29 给出。实线边形成相应的树,虚线边

是该流图的其他边。该树的深度优先遍历由 1→3→4→6→7→8→10 开始，然后回到 8，进到 9；再次回到 8，继续回到 7，6 和 4，再进到 5；从 5 回到 4，继续回到 3 和 1，进到 2；最后回到 1，完成了整个树的遍历。

先序遍历的结点序列是

1，3，4，6，7，8，10，9，5，2

后序遍历的结点序列是

10，9，8，7，6，5，4，3，2，1

深度优先排序是后序序列的逆，即

1，2，3，4，5，6，7，8，9，10 □

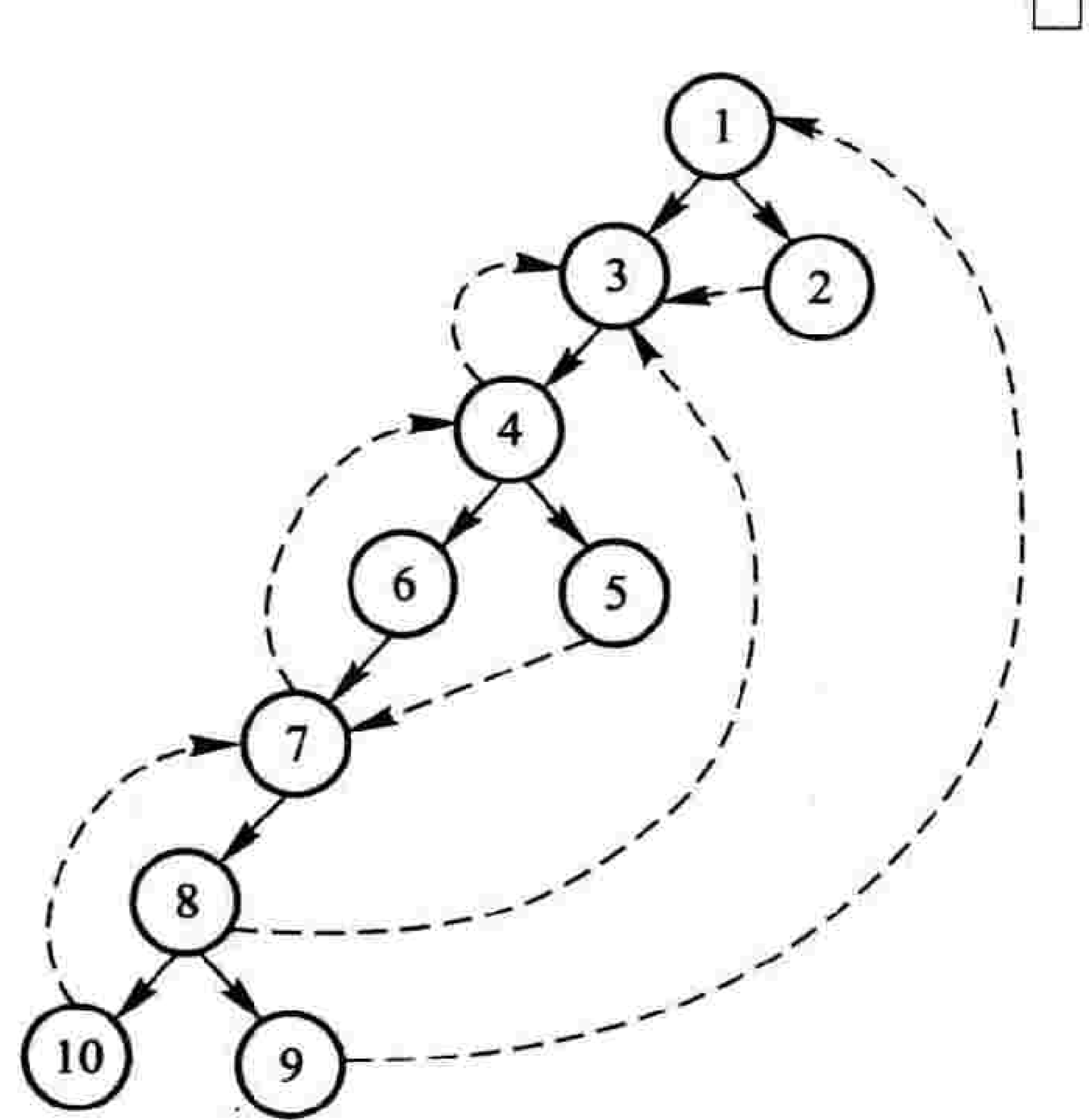


图 9.29　图 9.28 流图的深度优先表示

发现一个图的深度优先生成树和深度优先排序的算法可参考相关数据结构教材。图 9.29 的深度优先生成树就是对图 9.28 应用这样的算法得到的结果。

在构造一个流图的深度优先生成树时，该流图的边分成三类。

(1) 从结点 m 到 n 的边称为前进边（advancing edge），如果 n 在深度优先生成树上是 m 的真后代。在深度优先生成树上的所有边都是前进边。图 9.29 没有其他的前进边。但是，如果有边 4→8 的话，则它就是前进边。

(2) 从结点 m 到 n 的边称为后撤边（retreating edge），如果 n 在深度优先生成树上是 m 的祖先（包括就是 m 本身）。例如在图 9.29 中，4→3、7→4、10→7、8→3 和 9→1 都是后撤边。

(3) 从结点 m 到 n 的边称为交叉边（crossing edge），如果 m 和 n 在深度优先生成树上互不为对方的祖先。在图 9.29 中，2→3 和 5→7 是仅有的交叉边。交叉边的一个重要性质是，如果在画深度优先生成树时，一个结点的子结点从左到右画出的次序同它们加到该树上的次序一样的话，则所有的交叉边都从右到左。

流图的边 $a \to b$ 称为**回边**（back edge），如果 b 支配 a。对任何流图，回边一定是后撤边，但后撤边不一定是回边。一个流图称为**可归约**的，如果在它任何深度优先生成树上，所有的后撤边都是回边。换句话说，如果一个流图可归约，那么它所有的深度优先生成树都有同样的后撤边，并且它们正好就是流图的回边。如果流图不可归约，在它任何深度优先生成树上，回边都是后撤边，但每个深度优先生成树上都可能有不是回边的后撤边。这些后撤边可能因深度优先生成树的不同而不同。因此，如果把一个流图中所有回边删掉后，剩余的图有环，则原来的流图是不可归约的，反之也成立。

实际出现的流图几乎都是可归约的。程序若仅使用结构化控制流语句，如 if-then-else、

while-do、continue 和 break 语句等,其中间代码对应的流图都是可归约的。甚至使用 goto 语句的程序,只要程序员是合乎逻辑地基于循环和分支来思考的,那么它们也经常是可归约的。

图 9.28 的流图是可归约的,该图中的后撤边都是回边。

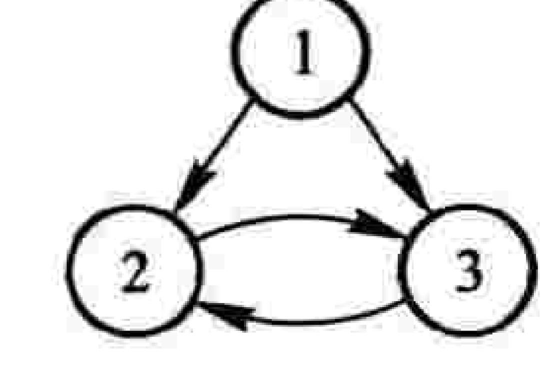


图 9.30 非归约流图

例 9.27 考虑图 9.30 的流图,它的起点是 1。结点 1 支配结点 2 和 3,但结点 2 和 3 相互不支配,因此该流图没有回边。该图有两种可能的深度优先生成树,一种情况下边 3→2 是后撤边但不是回边,另一种情况下边 2→3 是后撤边但不是回边。直观上说,它不可归约的原因是结点 2 和 3 构成的循环可以从两个不同的地方(结点 2 和 3)进入。 □

9.6.3 流图的深度

给定一个流图的一棵深度优先生成树,其深度就是在任何可能无环路径上的最大后撤边数。可以证明,深度决不会大于直观上认为的该流图中循环嵌套的层数。如果流图是可归约的,则可以用回边取代后撤边来定义深度,因为在任何深度优先生成树中,后撤边正好就是回边。这时,深度的概念就独立于所选择的深度优先生成树,也可以说是流图的真正深度,而不是流图联系到某棵深度优先生成树的深度。

例如,在图 9.29 中,流图的深度是 3,因为存在一条路径

$$10 \to 7 \to 4 \to 3$$

它有 3 条后撤边。该图不存在包含更多后撤边的无环路径。在这里最深路径碰巧仅含后撤边;一般情况下,最深路径可能是后撤边、前进边和交叉边的混合。

9.6.4 自然循环

在源语言一级,循环可以用多种方式来表达,例如 for 循环、while 循环和 repeat 循环,甚至可以用标号和 goto 语句方式。从程序分析的观点,循环在源程序中以哪种方式出现是无关紧要的。要紧的是它们是否具有易于分析的一些性质。尤其关注的是循环是否只有一个入口结点,如果是这样,编译器的优化分析可以假定在该循环每次迭代的开始,某些初始条件一定成立。由此产生一种称为“自然循环”的循环定义。

自然循环由下面两个本质性质来定义:

(1) 循环必须有唯一的入口结点,称之为**首结点**。首结点支配该循环中所有结点,否则它就不是该循环的唯一入口。

(2) 至少存在一条回边进入该循环首结点。否则,控制流不可能从该“循环”回到首结点,即这根本就不是一个循环。

给定一条回边 $n \to d$,该边确定的自然循环是 d 加上不经过 d 能到达 n 的所有结点。结点 d 是该循环的首结点。

算法 9.7 构造回边的自然循环。

输入 流图 G 和回边 $n \to d$。

输出 由回边 $n \to d$ 确定的自然循环中所有结点的集合 *loop*。

方法 令 *loop* 初值是 $\{n, d\}$。标记 d 为“已访问”，以保证搜索不会超出 d。从结点 n 开始，完成对流图 G 的逆向流图的深度优先搜索，把搜索过程中访问的所有结点都加入 *loop*。该过程找出不经过 d 能到达 n 的所有结点。 □

例 9.28 图 9.28 的流图有 5 条回边：10→7、7→4、4→3、8→3 和 9→1。这些正好是直观上认为该流图中形成循环的边。

回边 10→7 确定自然循环{7,8,10}，因为 8 和 10 是不经过 7 能到达 10 的仅有结点。回边 7→4确定自然循环{4,5,6,7,8,10}，它包含了自然循环{7,8,10}，因此相对来说，{7,8,10}被认为是内循环。

回边 4→3 和 8→3 确定的循环有同样的首结点，并且碰巧有同样的结点集合{3,4,5,6,7,8,10}，显然应该把它们看成是一个循环。上面提到的两个循环是该循环的内循环。

最后，回边 9→1 确定的自然循环是整个流图，并且它是最外循环。该例中，4 个循环正好依次一个嵌套在下一个的里面。然而，两个循环相互不嵌套才是更经常的情况。 □

在归约流图中，因为所有的后撤边都是回边，可以把自然循环联系到后撤边来定义。但这句话对非归约流图不成立。例如，在图 9.30 的非归约流图中，结点 2 和 3 构成一个环，但它们都不是回边，因此该环不符合自然循环的定义。不把该环标识为自然循环，当然也就不会对它进行循环优化。这样做是可接受的，因为假定所有循环都只有一个入口可以简化循环分析，而非归约流图在实际中又很少出现。

由于只把自然循环作为“循环”，因此流图还有这样的性质：除非两个循环有同样的首结点，否则它们要么不相交，要么一个嵌套在另一个中。

如果两个循环有同样的首结点，如图 9.31 那样，这时很难说哪一个是内循环。因此当两个循环有同样的首结点，并且并非一个包含另一个时，把它们合在一起看成一个循环。

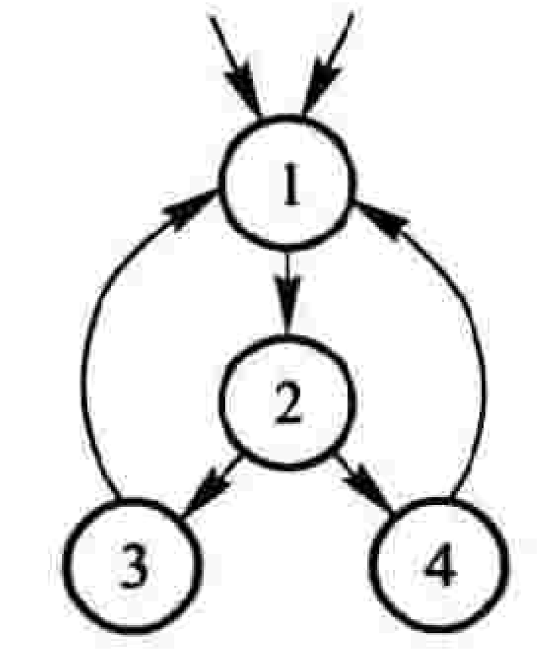


图 9.31 有相同首结点的两个循环

例 9.29 在图 9.31 中，回边 3→1 和 4→1 确定的自然循环分别是{1,2,3}和{1,2,4}。它们被合成一个循环{1,2,3,4}。

如果该图还有另外一条回边 2→1 的话，那么它确定的自然循环是{1,2}，结点 1 成为第三个循环的首结点。{1,2}真包含于{1,2,3,4}，因此{1,2}不合并到自然循环{1,2,3,4}中，而是作为一个内循环来处理。 □

9.6.5 迭代流图算法的收敛速度

现在可以讨论迭代算法的收敛速度了。如 9.3.3 节所讨论的那样，该算法迭代的最大次数

可以取格的高度和流图中结点数的积。对许多数据流分析,通过给结点的计算排序,使得该算法以一个小得多的迭代次数收敛是可能的。所关心的性质是,在一个结点处所有有意义结果是否都可以通过无环路径传播到另一个结点那儿。在讨论到目前为止的数据流分析中,到达-定值、可用表达式和活跃变量都有这个性质,但是常量传播没有这个性质。具体来说分以下三种情况。

(1) 如果定值 d 在 IN[B]中,那么存在某条从包含 d 的块到块 B 的无环路径,使得 d 在这条路径上所有的 IN 和 OUT 中。

(2) 如果表达式 x+y 在块 B 的入口不是可用的,那么存在一条从起点开始的无环路径,该路径上没有产生 x+y,或者存在一条从注销 x+y 的某块开始的无环路径,该路径随后没有产生 x+y。

(3) 如果 x 在某块 B 的出口是活跃的,那么存在一条从 B 开始到 x 一个引用的无环路径,在这条路径上没有 x 的定值。

对上面这三种情况都必须检查,带环的路径没有加入任何东西。例如,如果 x 的一个引用是它的某个定值 d 沿带环的路径可以到达,那么 d 沿着无这些环的相应路径也可以到达。

相反,常量传播没有这个性质。考虑只有一个基本块并且该块单独构成一个循环的简单程序

```
L: a = b
   b = c
   c = 1
   goto L
```

第一次访问该基本块时,发现 c 有常量值 1,但是 a 和 b 都没有定值;第二次访问该块时,发现 b 和 c 都有常量值 1;通过对该块的三次访问,给 c 赋的常量值 1 到达 a。

如果所有有用信息都沿无环路径传播,那么就有机会在迭代数据流算法中选择合适的结点访问次序,使得可以用相对少的遍数来保证信息已经通过所有无环路径。

从 9.6.2 节可以知道,只有边 $a \to b$ 是一条后撤边时,b 的深度优先序号小于 a 的深度优先序号。对于正向数据流问题,按照深度优先排序来访问结点是所希望的。具体说,将图 9.17(a)中算法第(4)行(访问流图中各基本块)替换为

for(除了 ENTRY 以外,按深度优先次序的每个块 B){

例 9.30 假如定值 d 沿一条路径传播,例如

$3 \to 5 \to 19 \to 35 \to 16 \to 23 \to 45 \to 4 \to 10 \to 17$

其中整数代表这条路径上基本块的深度优先序号。按照 9.17(a)的算法,第一次通过第(4)到(6)行的循环时,d 从 OUT[3]到 IN[5],再到 OUT[5],如此最后到 OUT[35]。在这一遍 d 到达不了 IN[16],因为 16 先于 35,在 d 被加入 OUT[35]时,IN[16]已经计算过了。第二次通过第(4)到(6)行的循环时,d 将被包含在 IN[16]中,因为它在 OUT[35]中。定值 d 传播到 OUT[16],继续到 IN[23],最后一直到 OUT[45],同第一遍一样,在这里它必须等待,因为 IN[4]在这一遍已经计算过了。第三次通过第(4)到(6)行的循环时,d 从 IN[4]到 OUT[4]、IN[10]、OUT[10],最后到 IN[17]。这样,经过三遍扫描,d 到达块 17。 □

从这个例子不难总结出一般原则。如果图 9.17(a)的算法使用深度优先次序,那么沿任何无环路径传播任何定值需要的遍数不会超过后撤边数加 1,即流图的深度加 1。当然,在发现新一遍扫描没有引起任何变化前,算法 9.4 并没有去判断所有的定值是否到达了它们可以到达的地方。因此,使用深度优先排序的算法 9.4,其迭代遍数的上界是深度加 2。研究表明,典型流图的平均深度大约 2.75。因此该算法会很快收敛。

对于逆向问题,例如活跃变量问题,则以深度优先次序的逆序来访问结点。因此,块 17 中一个变量引用沿路径

$$3\to5\to19\to35\to16\to23\to45\to4\to10\to17$$

逆向第一遍到 IN[4],第二遍到 IN[16],第三遍到 IN[3]。同样,采用了深度优先次序的逆序后,沿任何无环路径传播任何变量引用需要的遍数不会超过流图的深度加 1。

到目前为止描述的上界是一类问题的上界,这类问题的特点是,有环路径没有为分析加入任何信息。对于一些特别问题,例如支配集计算,算法收敛得更快。在这个问题中,如果采用一个按深度优先排序访问结点的数据流算法,则当输入流图可归约时,每个结点的支配集可以在第一次迭代时得到。如果不知输入流图是否可归约,则需要一次额外的迭代来确定是否已经收敛。

习 题 9

9.1 对于图 9.32 流图:

(a) 识别该流图的循环。

(b) 块 B_1 中的语句(1)和(2)都是复写语句,并且它们给 a 和 b 赋的都是常量。可以对 a 和 b 的哪些引用实施复写传播并将这些引用替换成对常量的引用?

(c) 识别每个循环的全局公共子表达式。

(d) 识别每个循环的归纳变量,不要忘记把(b)的复写传播引入的常量考虑进去。

(e) 识别每个循环的循环不变计算。

9.2 为图 9.33 计算向量 A 和 B 点积的中间代码完成下列优化:删除公共子表达式、归纳变量上的强度削弱和尽量删除归纳变量。

9.3 对图 9.32 的流图,计算:

(a) 为到达-定值分析,计算每个块的 *gen*、*kill*、IN 和 OUT 集合。

(b) 为可用表达式分析,计算每个块的 *e_gen*、

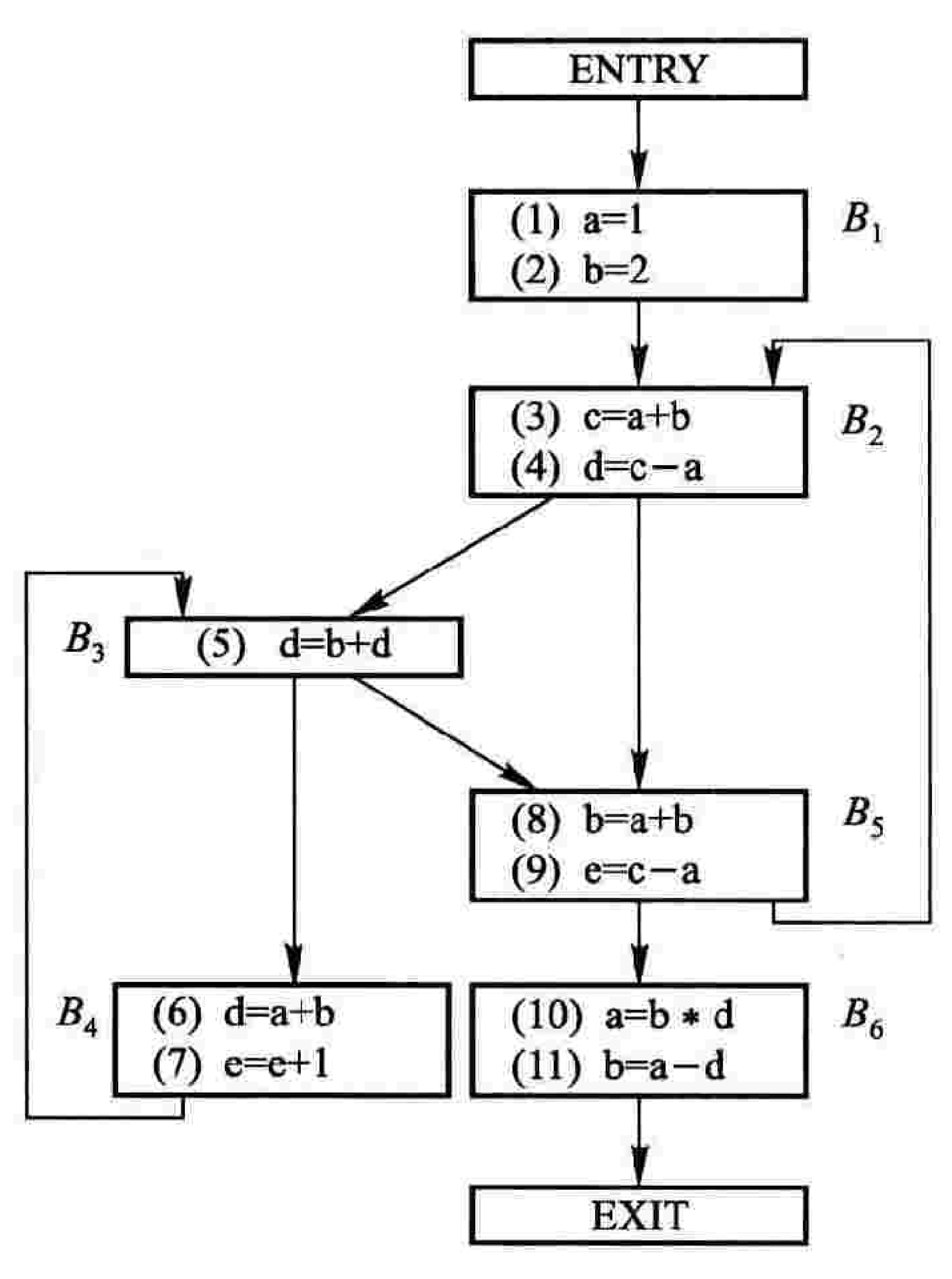


图 9.32 一个流图

e_kill、IN 和 OUT 集合。

```
      dp=0
      i=0
L:    t1=i*8
      t2=A[t1]
      t3=i*8
      t4=B[t3]
      t5=t2*t4
      dp=dp+t5
      i=i+1
      if i<n goto L
```

图 9.33　计算点积的中间代码

(c) 为活跃变量分析,计算每个块的 *def*、*use*、IN 和 OUT 集合。

*9.4　通过对算法 9.1 第(4)到(6)行 for 循环的迭代次数的归纳来证明,IN 和 OUT 集合不会缩小,即一旦某个定值在某一轮迭代中被加入,则在随后的继续迭代中决不会消失。

*9.5　证明算法 9.1 的正确性,即证明:

(a) 如果定值 *d* 加入 IN[*B*]或 OUT[*B*],那么存在一条从 *d* 开始分别到 *B* 入口或出口的路径,*d* 定值的变量在该路径上没有重新定值。

(b) 如果定值 *d* 没有加入 IN[*B*]或 OUT[*B*],那么不存在任何从 *d* 开始分别到 *B* 入口或出口的路径,*d* 定值的变量在该路径上没有重新定值。

*9.6　对算法 9.2 证明:

(a) IN 和 OUT 集合决不会缩小。

(b) 如果变量 x 加入 IN[*B*]或 OUT[*B*],那么存在一条分别从 *B* 入口或出口开始的路径,x 在该路径上被引用。

(c) 如果变量 x 没有加入 IN[*B*]或 OUT[*B*],那么不存在分别从 *B* 入口或出口开始的任何路径,x 在该路径上被引用。

*9.7　对算法 9.2 证明:

(a) IN 和 OUT 集合决不会扩大,即在迭代过程中,这些集合后续的值是先前值的子集(并非一定是真子集)。

(b) 如果表达式 *e* 从 IN[*B*]或 OUT[*B*]中删除,那么存在一条从流图起点分别到 *B* 入口或出口的路径,*e* 在该路径上没有计算,或者在最后一次计算后它的某个运算对象重新定值。

(c) 如果表达式 *e* 一直留在 IN[*B*]或 OUT[*B*]中,那么对于从流图起点分别到 *B* 入口或出口的任何一条路径,*e* 在该路径上被计算,并且在最后一次计算后,它的任何运算对象都没有重新定值。

9.8　假定 *V* 是复数集合。下面哪个运算可以作为 *V* 上一个半格的汇合运算?

(a) 加：$(a+\mathrm{i}b) \wedge (c+\mathrm{i}d) = (a+b)+\mathrm{i}(c+d)$

(b) 乘：$(a+\mathrm{i}b) \wedge (c+\mathrm{i}d) = (ac-bd)+\mathrm{i}(ad+bc)$

(c) 取较小成员：$(a+\mathrm{i}b) \wedge (c+\mathrm{i}d) = \min(a,c)+\mathrm{i}\min(b,d)$

(d) 取较大成员：$(a+\mathrm{i}b) \wedge (c+\mathrm{i}d) = \max(a,c)+\mathrm{i}\max(b,d)$

9.9 在9.3节中提到，如果块 B 由 n 个语句组成，并且每个语句的产生和注销集合分别是 gen_i 和 $kill_i$，那么块 B 迁移函数的产生和注销集合 gen_B 和 $kill_B$ 由下面的等式给出

$$kill_B = kill_1 \cup kill_2 \cup \cdots \cup kill_n$$

和

$$gen_B = gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \cdots \cup (gen_1 - kill_2 - kill_3 - \cdots - kill_n)$$

基于对 n 的归纳证明上面等式。

9.10 每个定值 $d_i(i=1,2,3)$ 构成一个格，为这三个格的积构造格图。所构造的格图和图9.16的格图有什么联系？

9.11 在9.3.3节提到，如果框架是有限高度，则迭代算法收敛。本题是高度无限并且迭代算法不收敛的例子。令集合 V 是非负实数集合，并且汇合运算是取极小值。三个迁移函数如下：

(1) 恒等函数 $f_I(x)=x$

(2) 取半函数 $f_H(x)=x/2$

(3) 恒等于1的常函数 $f_O(x)=1$

迁移函数集合 F 由上述三个函数加上它们的任意复合构成。

(a) 描述集合 F。

(b) 在该框架中，$\leqslant$ 关系是什么？

(c) 给出一个带指定迁移函数的流图，使得算法9.4不收敛。

(d) 该框架单调吗？它是否可分配？

9.12 假定想检查到达某变量一个引用点的所有路径中是否存在未初始化该变量的路径，试问应该怎样修改9.4节的框架？

9.13 对图9.34的流图：

(a) 为每个基本块的入口和出口计算 *anticipated*。

(b) 为每个基本块的入口和出口计算 *available*。

(c) 计算每个块的 *earliest*。

(d) 为每个基本块的入口和出口计算 *postponable*。

(e) 为每个基本块的入口和出口计算 *used*。

(f) 计算每个块的 *latest*。

(g) 引入临时变量 t，并说明它在哪里定值和在哪里引用。

9.14 为图9.32重复习题9.13。可以把你的分析限制到表达式 a+b、c-a 和 b * d。

9.15　对图 9.32 的流图：

(a) 计算支配关系。

(b) 找出一种深度优先排序。

(c) 对(b)的结果，标明前进边、后撤边和交叉边。

(d) 该流图是否可归约。

(e) 计算该流图的深度。

(f) 找出该流图的自然循环。

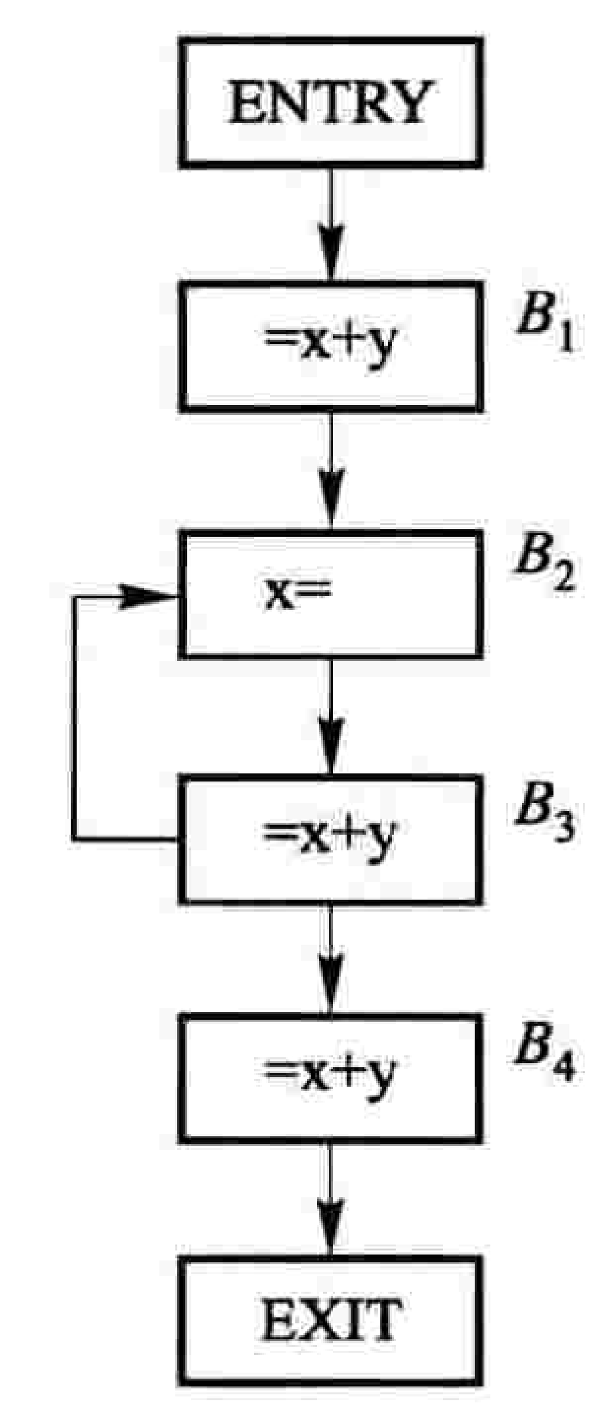


图 9.34　习题 9.13 的流图

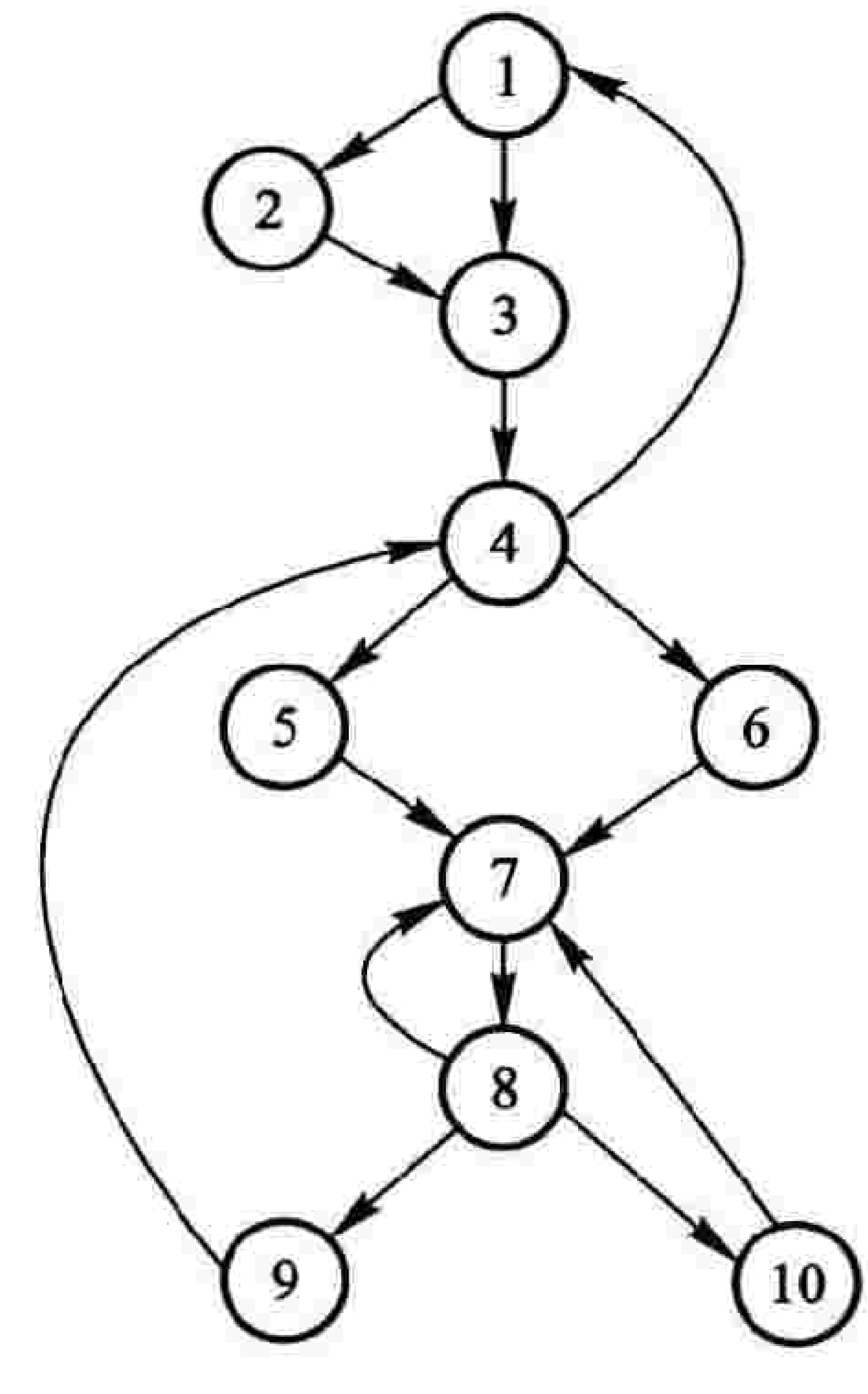


图 9.35　习题 9.17 的流图

9.16　对图 9.3 的流图重复习题 9.15。

9.17　对图 9.35 的流图重复习题 9.15。

9.18　一个 C 语言程序如下：

```
main() {
    int i,j,k;
    i=5;
    j=1;
    while(j<100) {
        k=i+1;
        j=j+k;
    }
}
```

在 x86/Linux 系统上经优化编译后（编译器版本见汇编代码最后一行），生成的代码如下：

```
    .file "optimize.c"
```

```
    .version "01.01"
gcc2_compiled.:
.text
    .align 4
.globl main
    .type main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    movl  $1,%eax
    movl  $6,%edx
    .p2align 4,,7
.L4:
    addl %edx,%eax
    cmpl  $99,%eax
    jle.L4
    leave
    ret
.Lfe1:
    .size main,.Lfe1-main
    .ident"GCC:(GNU)egcs-2.91.66 19990314/Linux(egcs-1.1.2 release)"
```

试说明编译器对这个程序作了哪些种类的优化(只需要说复写传播、删除公共子表达式等,不需要说怎样完成这些优化的)。

*9.19 下面是用C语言写的求最大公约数的函数

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

其中的递归调用称为尾递归(即return后的表达式是一个递归调用,而其他地方没有递归调用)。对于尾递归,编译器可以产生和一般的函数调用不同的代码,使得目标程序运行时,这种递归调用所需的存储空间大大减少,也缩短了运行时间。对于尾递归,编译器应怎样产生代码,简述你的想法。(若用源语言一级的优化来回答此问题,则不合题目要求。)

9.20 一个C语言程序

```
main( ) {
    long i,j;
    while(i) {
        if(j) { i=j; }
    }
}
```

的编译结果如下：

```
    .file "control.c"
    .version "01.01"
gcc2_compiled.:
.text
    .align 4
.globl main
    .type main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    subl  $8,%esp
    nop
    .p2align 4,,7
.L2:
    cmpl  $0,-4(%ebp)
    jne.L4
    jmp.L3
    .p2align 4,,7
.L4:
    cmpl  $0,-8(%ebp)
    je.L5
    movl -8(%ebp),%eax
    movl %eax,-4(%ebp)
.L5:
    jmp.L2
    .p2align 4,,7
.L3:
.L1:
```

```
        leave
        ret
.Lfe1:
        .size main,.Lfe1-main
        .ident  "GCC:(GNU)egcs-2.91.66 19990314/Linux(egcs-1.1.2 release)"
```

它的优化编译结果如下：

```
        .file "control.c"
        .version "01.01"
gcc2_compiled.:
.text
        .align 4
.globl main
        .type main,@function
main:
        pushl %ebp
        movl %esp,%ebp
.L7:
        testl %eax,%eax
        je.L3
        testl %edx,%edx
        je.L7
        movl %edx,%eax
        jmp.L7
        .p2align 4,,7
.L3:
        leave
        ret
.Lfe1:
        .size main,.Lfe1-main
        .ident  "GCC:(GNU)egcs-2.91.66 19990314/Linux(egcs-1.1.2 release)"
```

请分析优化编译器所做的控制流优化。

9.21 UNIX 下的 C 编译命令 cc 的选择项 g 和 O 的解释如下，其中 dbx 的解释是"dbx is an utility for source-level debugging and execution of programs written in C"。试说明为什么用了选择项 g 后，选择项 O 便被忽略。

-g　Produce additional symbol table information for dbx(1) and dbxtool(1) and pass-lg

option to ld(1)(so as to include the g library, that is:/usr/lib/libg.a). When this option is given, the-O and-R options are suppressed.

-O[level]　　Optimize the object code. Ignored when either-g, -go, or-a is used....

9.22　请利用代码优化的思想(代码外提和强度削弱等),改写下面C语言程序中的循环,得到优化后的C语言程序。

```
main(){
    int i,j;
    int r[20][10];

    for(i=0;i<20;i++){
        for(j=0;j<10;j++){
            r[i][j]=10*i*j;
        }
    }
}
```

9.23　C语言程序引用sizeof(求字节数运算符)时,该运算是在编译该程序时完成,还是在运行该程序时完成?说明理由。

9.24　某优化编译器对下面程序的局部变量i和j不分配空间,为什么?

```
main(){
    long i,j;
    i=5;
    j=i*2;
    printf("%d\n",i+j);
}
```

9.25　一个C语言的函数如下:

```
func(i)long i;{
    long j;
    j=i-1;
    func(j);
}
```

下面左右两边的汇编代码是两个不同版本GCC编译器为该函数产生的代码。左边的代码在调用func之前将参数压栈,调用结束后将参数退栈(见例6.4)。右边代码对参数传递的处理方式没有实质区别。请叙述右边代码对参数传递的处理方式并推测它带来的优点。

```
func:
        pushl   %ebp
        movl    %esp,%ebp
        subl    $4,%esp
        movl    8(%ebp),%edx
        decl    %edx
        movl    %edx,-4(%ebp)
        movl    -4(%ebp),%eax
        pushl   %eax
        call    func
        addl    $4,%esp
        leave
        ret
```

```
func:
        pushl   %ebp
        movl    %esp,%ebp
        subl    $8,%esp
        movl    8(%ebp),%eax
        decl    %eax
        movl    %eax,-4(%ebp)
        movl    -4(%ebp),%eax
        movl    %eax,(%esp)
        call    func
        leave
        ret
```

9.26 考虑一个类 Pascal 的语言，其中所有的变量都是整型（不需要显式声明），并且仅包含赋值语句、读语句、写语句、条件语句和循环语句。下面的产生式定义了该语言的语法（其中 **lit** 表示整型常量；**OP** 的产生式没有给出，因为它和所讨论的问题无关）。

Program→*Stmt*

Stmt→**id**:=*Exp*

Stmt→read(**id**)

Stmt→write(Exp)

Stmt→*Stmt*;*Stmt*

Stmt→**if**(*Exp*)**then begin** *Stmt* **end else begin** *Stmt* **end**

Stmt→**while**(*Exp*)**do begin** *Stmt* **end**

Exp→**id**

Exp→**lit**

Exp→*Exp* **OP** *Exp*

定义 *Stmt* 的两个属性：*Def* 表示在 *Stmt* 中一定会定值且在该定值前没有引用的变量集合，*MayUse* 表示在 *Stmt* 中有引用且在该引用前可能没有定值的变量集合。

(a) 写一个语法制导定义或翻译方案，它计算 *Stmt* 的 *Def* 和 *MayUse* 属性。

(b) 基于上面的计算，程序可能未赋初值的变量集合从哪儿可以得到？

可能未赋初值的变量是这样定义的：若存在从程序开始点到达变量 a 某引用点的一条路径，在这条路径上没有对变量 a 赋值，则变量 a 属于程序可能未赋初值的变量集合。

9.27 考虑一个简单语言，其中所有的变量都是整型（不需要显式声明），并且仅包含赋值

语句、读语句和写语句。下面的产生式定义了该语言的语法（其中 **lit** 表示整型常量；**OP** 的产生式没有给出，因为它和所讨论的问题无关）。

Program→*StmtList*

StmtList →*Stmt StmtList* | *Stmt*

Stmt→**id**: =*Exp*; | read(**id**); | write(*Exp*);

Exp→**id** | **lit** | *Exp* **OP** *Exp*

把不影响 write 语句输出值的赋值（包括通过 read 语句来赋值）称为无用赋值，写一个语法指导定义，它确定一个程序中出现过赋予无用值的变量集合（不需要知道无用赋值的位置）和没有置初值的变量集合（不影响 write 语句输出值的未置初值变量不在考虑之中）。

非终结符 *StmtList* 和 *Stmt* 用下面三个属性（你根据需要来定义其他文法符号的属性）：

（1）*uses_in*：在本语句表或语句入口点的引用变量集合，它们的值影响在该程序点后的输出。

（2）*uses_out*：在本语句表或语句出口点的引用变量集合，它们的值影响在该程序点后的输出。

（3）*useless*：本语句表或语句中出现的无用赋值变量集合。

9.28 下面左边的函数被 GCC：(GNU)3.3.5(Debian 1:3.3.5-13)优化成右边的代码。若认为该优化结果不对或该优化不合理，则阐述理由。若认为该优化结果是对的，则请说明实施了哪些优化，并解释循环优化结果的合理性。

```
f(a,b,c,d,x,y,z)int a,b,c,d,x,y,z;{
    while(a<b){
        if(c<d)
            x=x+z;
        else
            x=y-z;
    }
}
```

```
f:
        pushl       %ebp
        movl        %esp,%ebp
        movl        8(%ebp),%edx
        movl        12(%ebp),%eax
        cmpl        %eax,%edx
        jge .L9
.L7:
        jl. L7
.L9:
        popl        %ebp
        ret
```

9.29 下面左栏的 C 语言程序计算 5!，结果是 120。经优化编译器（编译器版本见汇编代码最后一行）生成的汇编程序跟在源程序的后面（先左栏后右栏），其中在省略号处略去了 main 函数的汇编程序。请问优化编译器对 factorial 函数进行了怎样的优化？

```
factorial(long n,long res){
    if(n==0)return res;
    else return factorial(n-1,n*res);
}
main(){
    printf("%d\n",factorial(5,1));
}
```

汇编程序:

```
    .file"recursion.c"
    .text
    .p2align 4,,15
.globl factorial
    .type     factorial,@function
factorial:
    pushl     %ebp
    movl      %esp,%ebp
    movl      8(%ebp),%edx
    movl      12(%ebp),%eax
    testl     %edx,%edx
    je        .L3
    .p2align 4,,7
.L6:
    imull     %edx,%eax
    subl      $1,%edx
    jne       .L6
.L3:
    popl      %ebp
    ret
    .size     factorial,.-factorial
    …
    .ident    "GCC:(GNU)4.2.3(Debian 4.2.3-5)"
    .section .note.GNU-stack,"",@progbits
```

9.30 对于例6.4的程序(见下面C代码),用编译器GCC:(GNU)4.2.3(Debian 4.2.3-5)编译后所生成的优化代码(见下面中间栏的汇编代码)同例6.4给出的代码(见下面左栏的汇编代码)略有不同。请指出该较新版本编译器完成的优化(或采用的不同代码生成策略)。给一点提示:若把调用语句改成func(j,j),则所生成的优化代码和下面中间的汇编代码的区别见下面右栏的说明。

```
func(i)long i;{
    long j;j=i-1;func(j);
}
```

func:	func:	
pushl %ebp	pushl %ebp	
movl %esp,%ebp	movl %esp,%ebp	
subl $4,%esp	subl $4,%esp	该指令改成 subl $8,%esp
movl 8(%ebp),%edx	movl 8(%ebp),%eax	
decl %edx	subl $1,%eax	在该指令和下条指令间
movl %edx,-4(%ebp)	movl %eax,(%esp)	增加 movl %eax,4(%esp)
movl -4(%ebp),%eax	call func	

```
    pushl   %eax
    call    func
    addl    $4,%esp
.L1:
    leave
    ret
```

```
    leave
    ret
```

*9.31 下面的C程序分别经非优化编译和2级以上(含2级)的优化编译后,生成的两个目标程序运行时的表现不同(编译器是GCC:(GNU) 4.2.3 (Debian 4.2.3-5))。请回答它们运行时的表现有何不同,并说明原因。

```
int f(int g()){
    return g(g);
}
main(){
    f(f);
}
```

* 第 10 章

依赖于机器的优化

现代高性能处理器能够在一个指令周期内执行多个操作。在这种指令级并行的机器上，一个程序的运行能快到什么程度？答案依赖于下面几个因素：

(1) 程序中潜在的并行；

(2) 处理器上可用的并行；

(3) 从串行程序提取并行的能力；

(4) 在给定的调度约束下发现最佳并行调度的能力。

如果一个程序中的所有运算高度地相互依赖，那么不管是硬件还是并行化技术都不能使程序并行地快速运行。典型的非数值应用有许多固有的相关性，例如，这样的程序有许多依赖于数据的分支，它们使得预测将要被执行的指令都变得非常困难，更不要说确定可以并行执行的操作。因此在该领域的研究集中在放宽调度约束，包括引入新的体系结构特征，而不是调度技术本身。

数值应用，例如科学计算和信号处理，一般有更多的并行。这些应用处理大批量数据，这些数据具有一定的数据结构，而且该结构中每个元素上的运算相互独立，因而在不同元素上的运算可以并行执行。高性能通用计算机和数字信号处理器上都有附加的硬件资源，它们可以利用这种并行。该领域的程序一般有比较简单的控制结构和规整的数据处理模式，并且许多从这类程序中提取并行的静态技术已经研发出来。这类应用的代码调度是重要且有意义的，因为这些应用提供了可以映射到大量计算资源上的大量独立运算。

并行的提取和并行执行的调度都可以静态地在软件中或动态地在硬件中完成。事实上，即使机器有硬件调度，它也可以从软件调度中得到帮助。本章从解释使用指令级并行的基础问题开始(忽略该并行是由软件还是硬件管理)，然后讨论提取并行的数据相关性分析，接下来给出代码调度的基本概念。随后，本章描述基本块调度的技术、应对通用程序中高度数据相关的控制流的方法和用于调度数值程序的软件流水线技术。

本章最后一节简要介绍使用数组的计算密集型程序在多处理器系统上的优化问题。许多科学、工程和商业应用对计算能力的渴求有无法满足的胃口，例如天气预报和设计药物的蛋白质折叠。加速计算的一种方式是采用并行。不幸的是，研发利用并行机特点的软件并不是件容易的事情。把一个大型计算任务分成若干个在不同处理器上并行执行的计算单元是一件有相当难度

的事情,而且它还不能保证一定得到加速的效果。另外,还必须考虑让处理器之间的通信最少,因为通信开销很容易使得并行代码运行得比串行执行还要慢。

最小化通信问题可以看成改进程序**数据局部性**的一种特殊情况。简单地说,处理器如果经常访问它最近使用过的数据,则认为相应程序有良好的数据局部性。可以肯定地说,并行机上的一个处理器如果有良好的数据局部性,则它和其他处理器之间不需要频繁通信。因此并行和数据局部性需要联合考虑。另外,数据局部性对一个内存分层的现代处理器的性能来说也是重要的,就像6.5.3节讨论的程序局部性那样。本章最后一节概述并行化和数据局部性优化的概念和方法,不介绍具体实现算法。

本章使用CISC风格的目标机器,其常用指令的形式如下:

(1) OP 目的,源1,源2

(2) LD 目的,地址

(3) ST 地址,源

10.1 处理器体系结构

在考虑指令级并行时,通常想象成一个处理器在单个时钟周期内发射多个操作。事实上,在每周期内发射一个操作是可能的,而指令级并行的获得是通过使用流水线的概念实现的。下面首先解释流水线,然后讨论多指令问题。

10.1.1 指令流水线和分支延迟

实际上,每种处理器,不管是高性能的超级计算机还是普通的机器,都使用**指令流水线**。采用指令流水线技术时,当先前的指令仍在继续通过流水线时,每个时钟周期可以取一条新指令。图10.1给出一个简单的5级流水线:取指令(IF)、译码(ID)、执行操作(EX)、访问内存(MEM)和回写结果(WB),显示了指令 i、$i+1$、$i+2$、$i+3$ 和 $i+4$ 同时执行的情况。每行对应于一个时钟周期,每列说明每条指令在每周期内占用的流水线级。

如果指令的结果数据在随后的指令需要该数据时已经可用,那么处理器可以在每周期发射一条指令。但是分支指令会引起问题,因为一直到这条指令被取出、译码和执行后,处理器才知道下面执行哪条指令。许多处理器投机地取出并译码分支指令的直接后继指令,但是如果发现应该执行一个分支而不是直接后继时,则投机失败,指令流水线被清空,转向该分支目的地址去取指令。因此转向一个分支时会引起取分支目标地址指令的延迟并引起指令流水线"打嗝"。高级的处理器通过使用硬件,根据各分支的执行历史来预测选取分支的结果并根据预测的目的地址预取指令。即使这样,分支延迟不可避免,因为分支预测会发生偏差。

	i	$i+1$	$i+2$	$i+3$	$i+4$
1.	IF				
2.	ID	IF			
3.	EX	ID	IF		
4.	MEM	EX	ID	IF	
5.	WB	MEM	EX	ID	IF
6.		WB	MEM	EX	ID
7.			WB	MEM	EX
8.				WB	MEM
9.					WB

图 10.1　5 级指令流水线中的 5 条连续指令

10.1.2　流水化的执行

有些指令的执行需要几个周期。一个简单的例子是读取内存指令。即使内存访问命中缓存,通常缓存也需要几个周期才能返回数据。一条指令的执行被称为流水化的,如果不依赖该指令结果的随后指令在该结果产生前就被允许执行。因此,即使处理器每周期只能发射一个操作,也可能出现几个操作同时出现在它们的执行级上。如果最长的执行流水线是 n 级,n 个操作同时进行的可能性是存在的。然而并非所有的指令都被完全流水化。例如,虽然浮点加和乘经常可以完全流水化,但是更加复杂而很少执行的浮点除经常是不行的。

大多数通用处理器动态地检查相继指令之间的依赖性,并且如果它们的操作数不可用则自动地延迟指令的执行。某些处理器,特别是嵌入在手持设备中的那些处理器,把数据相关性的检查交给软件,以保持硬件简单并且功耗低。在这种情况下,如果要求保证结果在需要时一定可用,则编译器插入什么也不做的 NOP 指令。

10.1.3　多指令发射

通过每周期发射多个操作,处理器可以让更多的操作同时进行。同时被执行的操作的最大数目可以是指令发射的宽度乘以执行流水线中的平均级数。

像流水线一样,多指令发射机器上的并行可以由软件或硬件来管理,依靠软件管理这种并行的机器称为超长指令字机器,而依靠硬件的称为超标量机器。超长指令字机器,正如该名字所暗示的,有比正常指令字更宽的指令字,它能将若干个操作编码在单周期中发射出去。编译器需要确定哪些操作可以并行发射,并用机器代码显式地编码这些信息。另一方面,超标量机器有按普通顺序执行语义的正规指令集。超标量机器自动检查指令之间的相关性,并且在它们的操作数可用时就发射它们。有些处理器同时包括超长指令字和超标量功能。

简单的硬件调度器按指令被取出的顺序执行指令,如果它碰巧遇到相关指令,那么该指令和所有后续指令都必须等待,直到该相关性被解决(也就是所需要的数据可用)为止。这样的机器显然会从安排相互不相关的指令相继执行的静态调度器中获益。

更复杂的代码调度器能够"乱序"执行指令。在这样的机器上,操作会自动地延迟,直到所依赖的所有值都产生时再继续执行。就是这样的调度器也可以从静态调度中获益,因为硬件调度器只有有限的空间来缓冲必须延迟的操作。静态调度可以把相互独立的操作靠近以获得较好的硬件利用率。更重要的是,不管动态调度器怎么复杂,它总不能执行尚未取出的指令。当处理器不得不取未意料到的分支时,它只能在新取指令中间去发现并行。编译器可以通过保证这些新取指令能够并行执行来增强动态调度器的性能。

10.2 代码调度的约束

代码调度是一种作用于代码生成器所产生的机器代码上的程序优化。代码调度受到下面三类约束。

(1) 控制相关约束:在原程序中执行的所有操作都必须在优化代码中执行。

(2) 数据相关约束:优化程序中的操作产生的结果必须同原程序对应操作的结果一样。

(3) 资源约束:调度必须不过分地占用机器资源。

这些调度约束保证优化程序和原程序产生同样的结果。但是,因为代码调度会改变操作执行的次序,因此某一个程序点的内存状态可能与顺序执行的任何内存状态都不匹配。如果程序的执行由于引发异常或碰到用户设定的断点等情况而被中断,那么这种不相匹配就是一个问题,它导致优化程序很难调试。当然这个问题不是代码调度专有的,其他的优化,如部分冗余删除和寄存器分配等也会引起同样问题。

10.2.1 数据相关

很容易明白,如果两个操作不涉及任何相同的变量,那么它们执行次序的改变不会影响结果;甚至它们读同一个变量时,交换它们的次序也无妨。只有当一个操作写由另一个操作读或写的变量时,改变它们的执行次序才会改变它们的结果。这样的一对操作被称为共享一个**数据相关**,它们的相对执行次序必须保持。存在以下三类数据相关。

(1) 真相关:如果对同一个单元先写后读,那么这个读依赖于所写的值,这种相关称为真相关。

(2) 反相关:如果对同一个单元先读后写,那么就说存在一个从这个读到这个写的一个反相关。这个写本质上不依赖于这个读,但是如果这个写碰巧先于这个读被执行,那么这个读操作将取到一个错误的值。反相关是命令式语言的副产品,因为在命令式语言中,一个内存

单元通常用来存储不同的值。反相关不是一种“真”相关,因为它可以通过把值存在不同的单元来删除。

(3) 输出相关:如果对同一个单元先后写两次,则称为输出相关。如果该相关性被违反,则在这两个操作都完成后,该内存单元将有一个错误的值。

反相关和输出相关统称为和内存单元联系的相关。它们不是真正的相关,因为可以通过使用不同的单元保存不同值来删除它们。注意,数据相关概念可同时用于内存访问和寄存器访问。

10.2.2 发现内存访问中的相关性

为了检查两个内存访问是否共享一个数据相关,需要知道它们是否引用同一个单元,但不用知道真正访问哪个单元。例如,可以知道 *p 和(*p)+4 这两个访问肯定不引用同一个单元,虽然不知道 p 究竟指向哪个单元。一般来说,数据依赖在编译时是不可判定的。因此编译器必须假定不同操作可能引用同一个单元,除非能证明它们引用不同的单元。

例 10.1 对于下面的代码序列

(1) a=1

(2) *p=2

(3) x=a

除非编译器知道 p 不可能指向 a,否则它只能总结出以上三个操作必须串行执行。因为语句(1)和(2)可能构成一个输出相关,语句(1)和(3)以及语句(2)和(3)可能构成两个真相关。 □

数据依赖分析是和写程序所用编程语言密切相关的。例如,像 C 和 C++这样的非类型可靠语言,它们的指针可以被强制为指向任何种类的对象,因此需要复杂的分析来证明任何一对基于指针的内存访问之间的独立性。这件事甚至还牵扯到局部和全局的标量,因为它们可以被间接地访问,除非能够证明它们的地址没有被程序中任何指令存放在任何地方。在像 Java 这样的类型可靠语言中,不同类型的对象必定相互区别。类似地,在栈上的局部简单变量不可能是任何通过其他名字访问的变量的别名。因此对于 Java 语言,相应的分析要简单得多。

正确发现数据相关需要几种不同形式的分析。下面先简要说明编译器要想找出程序中存在的所有相关的话,必须解决的主要问题,然后主要介绍怎样把这些信息应用到代码调度中去。本书不介绍怎样完成这些分析。

1. 数组数据相关分析

它是指在数组元素的访问中,消除下标表达式值之间的二义。例如,循环

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

将数组 A 中每个奇元素复写到先于并紧靠它的偶元素。因为在该循环中,读写的单元相互区别,因此在这些访问之间不存在相关,因而该循环中所有迭代可以并行执行。数组数据相关分析(通常简称数据相关分析),在数值应用的优化中是非常重要的。

2. 指针别名分析

在编程语言中,两个不同的名字(包括有左值的表达式)如果代表运行时的同一个内存单元,则它们互为**别名**(alias)。别名通常由指针、引用调用、下标变量和C语言的联合体等引起。上面所说的消除下标表达式值之间的二义问题也就是分析两个下标变量是否互为别名问题。两个指针的值相等的话,则它们会引起很多的别名。例如,若p和q相等,则*p和*q、p->next和q->next、p->data和q->data等都分别互为别名(如果它们都是程序中合法的表达式)。由此,两个指针的值相等,即它们指向同一个单元时,称它们为**别名化指针**(aliased pointer)。指针别名分析就是通过分析指针运行时可能的值(也称为指针分析)来确定可能别名化指针或者指针可能指向的单元集合(也称为point-to分析)。指针别名分析是非常困难的,因为程序中存在许多潜在的别名化指针,并且其中每个指针在程序运行过程中都可以随着时间的推移而指向不计其数的动态对象。为了提高指针分析的精度,指针分析还必须穿越一个程序的所有过程。因为没有过程间的分析,则只能假定被调用过程会改变所有可访问指针变量的内容,导致过程内的指针分析也变得低效。

3. 过程间分析

对于采用引用调用的语言,过程间分析用来确定同一个变量是否作为两个或更多的不同变元被传递(习题6.8是一个例子)。这样的别名会导致在表面上不同的形参访问之间建立了相关。类似地,当全局变量作为实参时,也会建立形参访问和全局变量访问之间的相关。过程间分析对确定这样的别名是必需的。

10.2.3 寄存器使用和并行执行之间的折中

本章假定在翻译源程序得到的独立于机器的中间表示中,使用个数不限的伪寄存器来表示可以分配到寄存器中的变量。这些变量包括源程序中不能通过其他名字引用的标量,还有编译器为了保存表达式中间结果的临时变量。不像内存单元那样,寄存器是唯一命名的,因此很容易为寄存器访问产生精确的数据相关约束。

中间表示采用的个数不限的伪寄存器,最终都必须映射到目标机器上可用的少数物理寄存器上。把几个伪寄存器映射到同一个物理寄存器上会人为导致存储相关,限制了指令级并行。反过来,并行执行指令会要求更多的存储单元来保存同时计算的值。因此,极小化寄存器使用个数和极大化指令级并行这两个目标是相冲突的。下面的例10.2和10.3说明寄存器使用和并行执行之间的典型折中。

例10.2 下面的代码使用伪寄存器t1和t2,把内存变量a和c的值分别复写到内存变量b和d。

```
LD  t1,a
ST  b,t1
LD  t2,c
```

```
ST d,t2
```

如果所有被访问的内存单元相互是有区别的,那么这两个复写可以并行执行。但是如果把 t1 和 t2 指派到同一个寄存器,那么寄存器的使用是极小化了,但是这两个复写就只能串行执行了。□

例 10.3 传统的寄存器分配技术瞄准极小化寄存器的使用个数。考虑下面的表达式

(a+b)+c+(d+e)

其语法树如图 10.2 所示。完成这个计算可以只使用三个寄存器,其代码如下:

```
LD R1,a
LD R2,b
ADD R1,R1,R2
LD R2,c
ADD R1,R1,R2
LD R2,d
LD R3,e
ADD R2,R2,R3
ADD R1,R1,R2
```

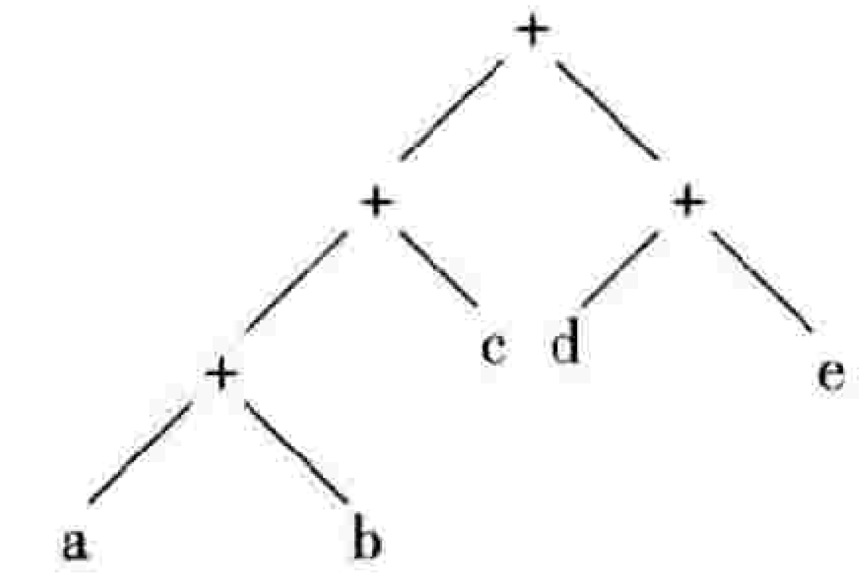
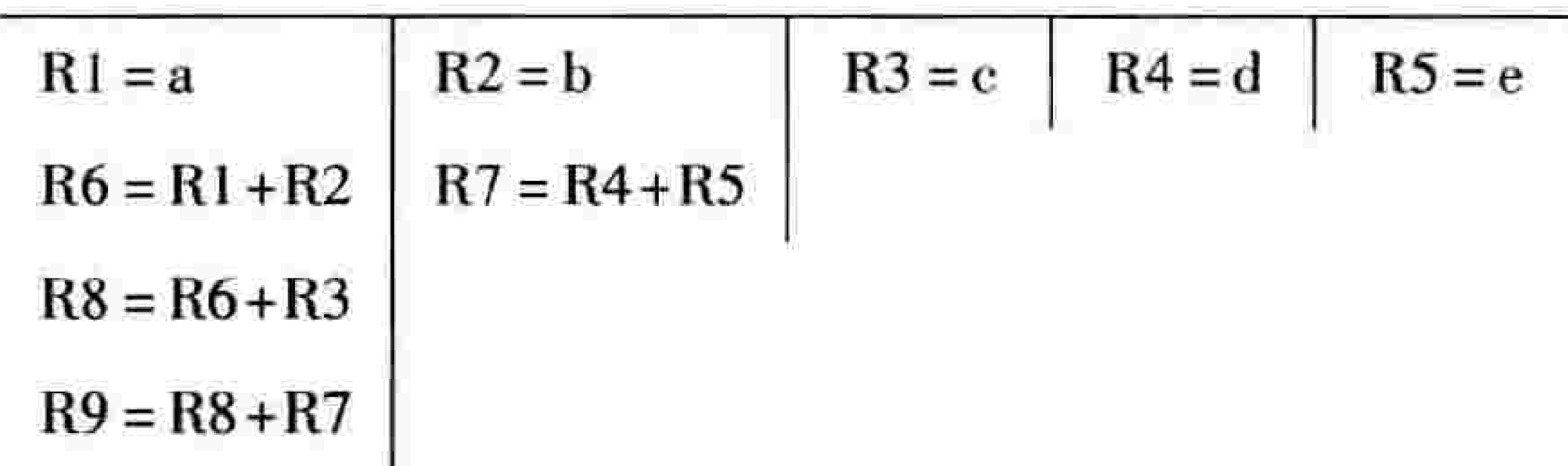


图 10.2 例 10.3 的表达式树

重复利用寄存器导致计算的串行,仅允许的并行操作是同时把 a 和 b 分别取到寄存器 R1 和 R2,还有同时取 d 和 e 的操作。因此并行完成整个计算需要 7 步。

如果对每个中间结果使用不同寄存器的话,则完成该表达式的计算只需要四步,它和图10.2 中表达式的高度是一致的。具体的并行方式见图 10.3。□



R1 = a	R2 = b	R3 = c	R4 = d	R5 = e
R6 = R1+R2	R7 = R4+R5			
R8 = R6+R3				
R9 = R8+R7				

图 10.3 图 10.2 表达式的并行计算

10.2.4 寄存器分配和代码调度的次序安排

如果在代码调度之前进行寄存器分配,则结果代码中会有很多存储相关,它们限制了代码调度。另一方面,如果代码调度在寄存器分配之前,那么代码调度会导致需要许多寄存器的代码,寄存器溢出会抵消指令级并行的优点。寄存器溢出是指把寄存器的内容存入内存单元,使得寄存器可以用于其他目的。由此,对于寄存器分配和代码调度,编译器究竟应该先做哪个?还是编译器应该同时考虑这两个问题?

为回答这个问题,必须考虑被编译程序的特点。许多非数值应用没有很多可用的并行,考虑

用较少的寄存器来保存表达式的中间结果就可以了。这时可以先分配寄存器,然后调度代码。但是这种方式不适用于有许多大表达式的数值应用。对它们可以从最内层循环开始,由里向外逐层地优化代码。在假定每个伪寄存器就是物理寄存器的情况下,首先调度指令。然后进行寄存器分配,把处理寄存器溢出的代码附加在必要的地方,并再次进行代码调度。对较外层循环的代码重复这个过程。当同时考虑处于一个较外循环的多个内循环时,同样的变量可能被指派了不同的寄存器。可以调整寄存器的指派来避免在寄存器之间进行值的复写。10.5 节将讨论在特定调度算法中,寄存器分配和代码调度之间的相互影响。

10.2.5 控制相关

基本块内的代码调度相对容易,因为当控制流到达该块入口时,则该块所有指令肯定要被执行一次。因此基本块中的指令可以任意重新排序,只要所有的数据相关得到满足就可以。不幸的是,基本块非常小,尤其是非数值程序中的基本块,平均一个基本块只有五条指令。此外,一个基本块中的操作通常高度相关,几乎不能并行。因此调查跨基本块的并行就变得至关重要了。

优化程序必须执行原程序中的所有操作。它可以比原程序执行更多的指令,只要额外的指令不改变程序的语义就可以。为什么执行额外的指令会加快程序的执行?如果知道一条指令很可能被执行并且有空闲的资源可“免费”用于完成该指令的操作,那么可以投机地执行该指令。如果投机成功,则程序可以运行得快一些。

指令 i_1 被认为控制相关于指令 i_2,如果 i_2 的结果决定 i_1 是否被执行。例如,在条件语句

```
if (c) s1;else s2;
```

中,s1 和 s2 都控制相关于 c。类似地,在循环语句

```
while (c) s;
```

中,循环体 s 控制相关于 c。

例 10.4 在代码片段

```
if (a>t)
    b=a*a;
d=a+c;
```

中,语句 b=a * a 和 d=a+c 同该片段的其他部分没有数据相关。语句 b=a * a 依赖于比较 a>t 的结果,但是语句 d=a+c 不依赖于该比较并且可以在任何时候执行。假定乘 a * a 不会产生任何副作用,则它可以投机地执行,只要在发现 a 大于 t 之后才把 a * a 的结果写入 b 就可以了。 □

10.2.6 投机执行的支持

内存读取是一类能从投机执行大大获益,并且使用相当频繁的指令。它们有相对长的输入延迟,读取指令中使用的地址通常是事先可用的,并且读取的结果可以存放于新的临时变量而不

会破坏任何其他变量的值。不幸的是,如果地址非法的话,则内存读取会引发异常,因此投机访问非法地址可能会引起正确程序意料不到的停止。此外,由于预测不正确发生的内存读取会引起额外的缓存未命中和缺页,这个代价是很高的。

例 10.5 在代码片段

```
if( p != null)
    q= * p;
```

中,投机地对 p 脱引用,将引起该正确代码片段因试图取 null 指向对象的错误而停止。 □

许多高性能处理器提供专门的特性来支持投机地访问内存,下面介绍其中一些最重要特性。

(1) 预取。预取指令被设计来在数据使用前,将它从内存取到缓存。预取指令告诉处理器,程序在不久将来很可能使用某个特定的内存单元。如果该单元无效或者如果访问它会引起缺页,则处理器简单地忽略该操作;否则,处理器将数据从内存取到缓存,如果它还不在缓存的话。

(2) 抑制位。另一种叫做抑制位(poison bit)的体系结构特征被设计用来允许投机地从内存将数据读取到寄存器堆。该处理器上的每个寄存器增加了抑制位。如果出现非法内存访问或者所访问的页不在内存,该处理器并不马上引发异常,而只是设置目标寄存器的抑制位。只有当已经设置了抑制位的寄存器的内容被使用时,才会引发异常。

(3) 判定执行。由于分支的花费很大,尤其是在分支预测出现错误时,因此采用判定指令(predicated instruction)来减少程序中的分支数。判定指令与普通指令类似,但是有一个额外的判定操作数来看守它的执行。只有在判定为真的情况下,判定指令才执行。

作为一个例子,条件传送指令 CMOVZ R2,R3,R1 只有在 R1 为零的情况下,才将寄存器 R3 的内容移到寄存器 R2 中。假定 a、b、c 和 d 分别被分配了寄存器 R1、R2、R4 和 R5,代码片段

```
if (a==0)
    b=c+d;
```

可以用两条机器指令实现:

```
ADD R3,R4,R5
CMOVZ R2,R3,R1
```

这个转换将一个控制相关的指令序列变换为只有数据相关的指令序列。这些指令于是可以用来将相邻基本块组合成一个更大的基本块。更重要的是,使用这样的代码可以保证指令流水的平滑运行,因为处理器不会出现预测错误。

判定执行伴随着一些代价的发生。判定指令被预取和译码,即使它们有可能最后不被执行。静态调度必须预留所有需要用于它们执行的资源,并保证所有潜在的数据相关都满足。应谨慎使用判定执行,除非机器有足够多的资源。

10.2.7 一个基本的机器模型

许多机器可以用下面的简单模型表示。机器 $M=(R,T)$ 由 R 和 T 两部分组成:

(1) 操作类型的集合 T,例如读取、存储和算术运算等。

(2) 代表硬件资源的向量集合 $R=[r_1,r_2,\cdots]$,其中 r_i代表第 i 类资源中可用的部件数目。典型的资源类型包括内存访问部件、算术运算部件和浮点功能部件等。

每个操作有一组输入操作数、一组输出操作数和一个资源需求。和每个输入操作数相关的是一个输入延迟,用于指示什么时候输入值必须可用(相对于该操作的启动)。典型的输入操作数是零延迟的,就是说该值在该操作被发射的那个周期就需要。类似地,和每个输出操作数相关的是一个输出延迟,它指示什么时候结果可用(相对于该操作的启动)。

对每种机器操作类型 t,资源使用由一张二维资源预留表 RT_t来建模。该表的宽度是该机器上资源种类数,该表的长度是资源被该类型操作使用的持续时间。条目 $RT_t[i,j]$是 t 类型的一个操作在它被发射 i 时钟周期后,使用第 j 种资源的部件数。为了表示上的简单,如果 i 引用该表一个不存在的条目(例如,i 大于该操作执行所需的时钟周期数),则假定 $RT_t[i,j]=0$。当然,对任何 t、i 和 j,$RT_t[i,j]$必须小于或等于 $R[j]$,$R[j]$是该机器拥有的第 j 类型资源数目。

典型的机器操作在它被发射时仅占用一个资源部件。有些操作可能使用的功能部件多于一个。例如,一个乘加操作可以在第一个时钟周期使用一个乘法器,在第二个时钟周期使用一个加法器。全流水的操作是那些能够在每个时钟周期都发射的操作,即使它们的结果在几个时钟周期后才可用。不需要为一条流水线各级的资源显式地建模,只需要为代表第一级的资源部件建模,占据一条流水线第一级的任何操作被保证在随后各时钟周期正确处理随后各级。

10.3 基本块调度

现在开始讨论代码调度算法。先从最简单的调度基本块中的操作开始,该问题的最优解是一个 NP 完全问题。但是在实际中,一个典型的基本块只有几个高度受约束的操作,只需要简单的调度技术就足够了。本节介绍一个称为**表调度**的简单高效算法。

10.3.1 数据依赖图

机器指令的每个基本块由数据依赖图 $G=(N,E)$来表示,其中结点集合 N 表示该块的机器指令中的操作集合,有向边集合 E 表示这些操作之间的数据相关约束。G 的结点集 N 和边集 E 按如下两步构造。

(1) N 中的每个操作 n 有一张资源预留表 RT_n,它的值直接就是 n 的操作类型的资源预留表。

(2) 边集 E 中的每条边 e 都标示有延迟 d_e,它表示 e 的目的结点必须在它源结点发射 d_e个时钟周期之后才可以发射。假定操作 n_1 由操作 n_2跟随,并且它们访问同一个单元,前者的输出延迟和后者的输入延迟分别是 l_1和 l_2。也就是该单元的值在第一条指令发射 l_1周期后产生,并且

在第二条指令发射 l_2周期后被需要(通常 $l_1=1$ 且 $l_2=0$)。那么边 $n_1 \to n_2$标有延迟 l_1-l_2。

例 10.6 考虑一种简单的机器,它每周期可以执行两个操作。一个操作必须是分支操作或算术逻辑部件 ALU 的操作,另一个操作必须是读取(LD)或存储(ST)操作。读取操作是全流水的并且需要 2 周期,读取操作可以直接由写到该读取单元的存储操作跟随。其他所有操作都在 1 周期内完成。

图 10.4 是一个基本块的依赖图和它的资源需求。可以把 R1 想象成一个栈指针,通过像 0 或 12 这样的偏移来访问栈上的数据。由于第一条指令读取数据到 R2 需要 2 周期,因此从它到第二和第五条指令的边上的标记都是 2。类似地,从第三条指令到第四条指令的边上的标记也是 2。

由于不知道 R1 和 R7 的值是怎样的关系,则不得不认为 8(R1)和 0(R7)有可能是同一地址。这样,第七条指令存储的地址可能同第三条指令读取的地址相同。根据所用的机器模型,存储一个单元的指令可以在读取同一个单元的指令发射 1 周期后就发射,虽然这时值还需 1 周期才能取到寄存器。所以从第三条指令到第七条指令的边的标记为 1。第一条指令和第二条指令到第七条指令的边的标记也为 1 是同样的道理。 □

通常用灰白两色的方格来形象化资源预留表。每列对应到机器的一种资源,每行对应到相应操作执行过程中的 1 周期。假定各操作对任何一种资源最多只需要一个部件,则可以用灰色方块表示 1,用白色方块表示 0。此外,如果操作是全流水的,那么则只需要显示在第一行使用的资源,这样资源预留表就可以简化成单行的表。

例如,例 10.6 使用的就是这种表示。在图 10.4 中,资源预留表用方块行表示,两个加运算需要 ALU 资源,读取和存储指令需要内存资源。

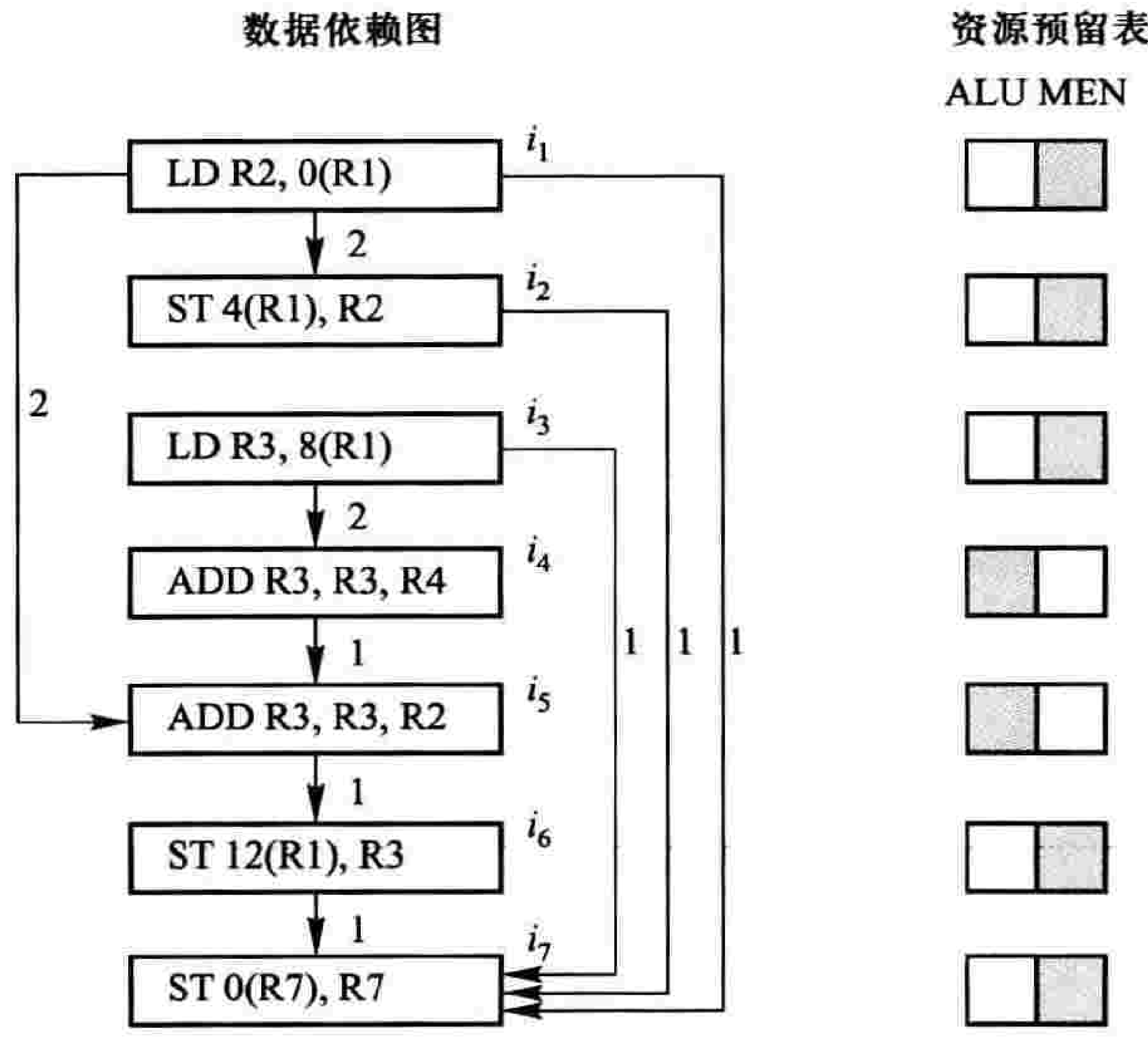


图 10.4 例 10.6 的数据依赖图

10.3.2 基本块的表调度

调度基本块的最简单方式，涉及用区分优先级的拓扑次序(prioritized topological order)来访问数据依赖图中各个结点。因为数据依赖图无环，因此至少存在这些结点的一种拓扑次序。然而在这些可能的拓扑次序中，某些次序可能比其他次序优越。10.3.3节将讨论选择拓扑次序的某些策略，但是现在假定存在算法选择一个较优的次序。

下面描述的表调度算法按所选定的区分优先级的拓扑次序来访问结点。结点调度的次序和它们被访问的次序可能相同，也可能不相同。不过，指令总归是尽可能早地放进调度表，因此一种趋势是指令以近似被访问的次序被调度。

更具体一点，该算法根据每个结点同先前已经被调度的各结点之间的数据相关约束，来计算该结点可以执行的最早时间槽。然后，该结点所需资源根据一张资源预留表进行检查，该资源预留表收集了所有到目前为止被占用的资源。该结点的调度按有足够资源的最早时间槽来安排。

算法 10.1 基本块的表调度。

输入 机器资源向量 $R=[r_1,r_2,\cdots]$，其中 r_i 是机器第 i 种资源的可用部件数目；另外还有数据依赖图 $G=(N,E)$。N 中的每个操作 n 都带有它的资源预留表 RT_n(根据上面的讨论，简化为只有一行的表)，E 中每条边 $e=n_1\rightarrow n_2$ 标有 d_e，表示 n_2 必须在 n_1 开始 d_e 周期后才能开始。

输出 一张调度表 S，它把 N 中的每个操作映射到一个时间槽，该操作在该时间槽可以启动，满足所有数据和资源的约束。

方法 执行图10.5的程序。有关其中“区分优先级的拓扑次序”的讨论在10.3.3节中。 □

```
RT = 一张空的资源预留表;
for(N 中按区分优先级的拓扑次序的每个 n){
    s = max_{e=p→n in E}(S(p)+d_e);      /* 查找该指令在所有前驱指令都启动后，它能够启动的最早时间 */
    while (存在一个 i 使得 RT[s,i]+RT_n[i]>R[i])
        s = s+1;                         /* 延迟指令直到所需资源可用 */
    S(n) = s;
    for (所有的 i)
        RT[s,i] = RT[s,i]+RT_n[i];
}
```

图10.5 一个表调度算法

注意，其中 $RT[s,i]$ 表示第 i 种资源在时间槽 s 的预留数，$RT_n[i]$ 表示依赖图上第 n 条指令对第 i 种资源的需求数，$R[i]$ 表示第 i 种资源的可用部件数。p 是已经进入调度表 S 的第 n 条指令的前驱指令。

10.3.3 区分优先级的拓扑次序

表调度算法不需要回溯,它调度每个结点都正好一次。在调度中它使用启发式优先级函数,从可被调度的结点中选择一个结点。该启发式调度的一些原则如下。

(1) 在没有资源约束的时候,最短调度由**关键路径**给出,关键路径是数据依赖图上的最长路径。作为优先级函数的一个有用尺度是结点的高度,也就是该图中源自该结点的最长路径的长度。

(2) 另一方面,如果所有操作都是相互独立的,那么调度的长度就只受资源约束。关键资源是所有可用资源中使用频率最高的资源,那些使用较多关键资源的操作可以给予较高优先级。

(3) 最后,可以使用原先的代码排序来打破操作之间调度的平局,在原先代码中较早出现的操作将优先被调度。

例 10.7 对于图 10.4 的数据依赖图,关键路径包括最后五个结点,该路径边上延迟的总和是 5,最后一个操作需要的时间是 1 周期,因此该路径是 6 周期。

使用高度作为优先级函数,算法 10.1 找出一种优化调度见图 10.6。注意,第 3 条指令由于具有最高高度而首先被调度,下一步原本可以调度第 4 条指令,但是由于第 3 条指令读取操作的延迟是 2,因此必须等到第 3 周期才能调度第 4 条指令。该基本块经过表调度后,原来 7 条指令能够在 7 周期内完成,包括执行最后 1 条指令需要的 1 周期。 □

调度表

	LD R3, 8(R1)
	LD R2, 0(R1)
ADD R3, R3, R4	
ADD R3, R3, R2	ST 4(R1), R2
	ST 12(R1), R3
	ST 0(R7), R7

资源预留表

ALU	MEN
	■
	■
■	
■	■
	■
	■

图 10.6 表调度用到图 10.4 中例子的结果

10.4 全局代码调度

对于有适度指令级并行的机器,仅考虑紧凑单个基本块的调度会引起许多资源空闲。为了更好地利用机器资源,需要考虑把指令从一个基本块移到另一个基本块的代码生成策略。每次考虑多于一个基本块的调度策略被称为**全局调度**算法。为了正确地进行全局调度,不仅需要考虑数据相关,还需要考虑控制相关。必须保证:

(1) 原来程序中所有的指令在优化程序中都被执行,并且

(2) 当该优化程序可以投机地执行额外指令时,这些指令肯定不能有任何多余的副作用。

10.4.1 简单的代码移动

先围绕一个例子来研究将操作在基本块之间移动涉及的问题。

例 10.8 假定机器可以在一个时钟周期执行任意的两个操作。除读取操作外,每个操作的执行有 1 周期的延迟,读取操作有 2 周期的延迟。为简单起见,假定该例中所有内存访问操作都是有效的,并且能命中缓存。图 10.7(a)给出只有三个基本块的一个流图,图 10.7(b)是翻译成机器操作后的代码。其中 BEQZ R6,L 是条件转移指令,表示如果 R6 的值为 0 则跳转到 L 去执行,否则执行后继指令。由于数据相关,每个基本块中内的指令必须串行执行,并且还必须插入代表空操作的 NOP 指令。

假定变量 a、b、c、d 和 e 的地址是有区别的,它们的地址分别保存在寄存器 R1 到 R5。这样,不同基本块之间的操作没有数据相关。可以看出,不管分支测试最终选择什么路径,B_3的所有操作肯定都要执行,因而它们可以和 B_1的操作并行执行。但是不能把 B_1的操作移动到 B_3,因为它们是用来确定分支测试结果的。

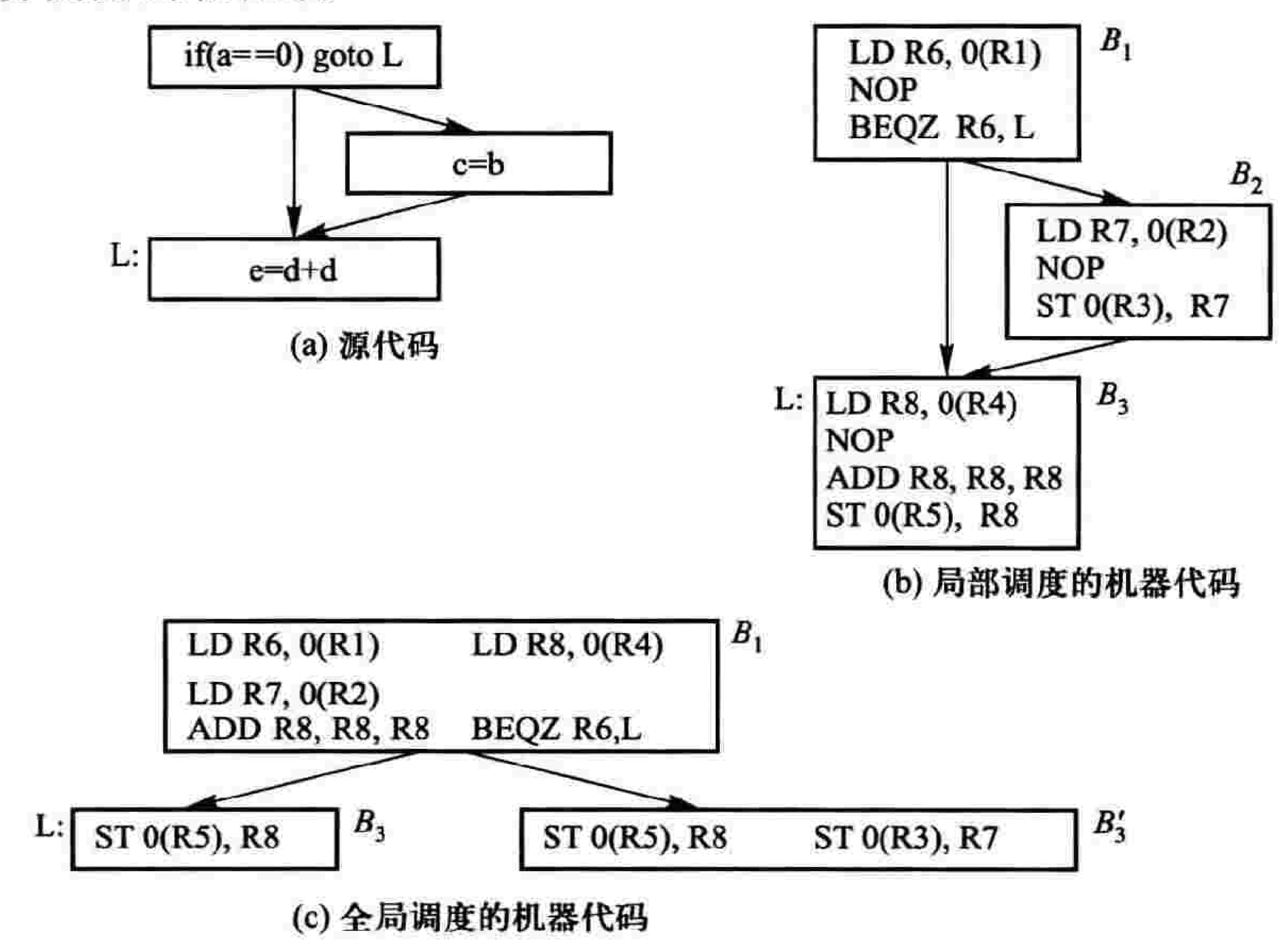


图 10.7 例 10.8 中全局调度前后的流图

块 B_2的操作是控制依赖于块 B_1测试结果的。可以在 B_1投机地完成 B_2的读取操作,若测试结果导致 B_2被执行,则此举可以节约 2 周期的执行时间。

存储操作不能投机地完成,因为它覆盖了一个存储单元的旧值,但是延迟一个存储操作是可以的。由于 B_2并不一定执行,因此不能把 B_2的存储操作直接放到 B_3中,但是可以把该存储操作

放到 B_3的一个副本中。图 10.7(c)展示了一个优化的调度,其中用逗号隔开的并排两条指令同时执行。该优化代码执行只需要 4 周期,它和单独执行原先 B_3的时间是一样的。 □

例 10.8 表明,把操作沿执行路径上下移动是可能的。该例中的每对基本块有不同的"支配关系",因而指令在每对基本块之间何时和怎样移动的考虑是不同的。9.6 节已经介绍过,如果每条从控制流图入口到达块 B'的路径都要通过块 B 的话,那么称 B 支配 B'。类似地,如果每条从 B'到达该图出口的路径都要通过 B 的话,那么称 B 后支配(postdominate)B'。如果 B 支配 B' 并且 B'后支配 B,那么称 B 和 B'是**控制等价**的,意思说它们中一块被执行当且仅当另一块被执行。例如,假定图 10.7 中 B_1和 B_3分别是入口和出口,

(1) B_1和 B_3控制等价,B_1支配 B_3,B_3后支配 B_1;

(2) B_1支配 B_2,但是 B_2并非后支配 B_1;

(3) B_2不支配 B_3,但是 B_3后支配 B_2。

显然,一条路径上一对基本块相互之间既无支配关系也无后支配关系也是可能的。

10.4.2 向上的代码移动

现在仔细考察沿一条路径向上移动一个操作意味着什么。假如想把一个操作沿控制流从块 *src* 向上移动到块 *dst*。假定这个移动没有违反任何数据相关,并且它使得通过 *dst* 到 *src* 的路径运行得较快。

如果 *dst* 和 *src* 等价,那么当被移动操作应该被执行时,它正好仅被执行一次。

如果 *src* 没有后支配 *dst*,那么存在通过 *dst* 但没有到达 *src* 的路径。在这种情况下,代码移动会导致在这样的路径上额外执行了被移动操作,因此这时代码移动是非法的,除非该被移动操作没有多余的副作用。如果该被移动操作可利用空闲资源"免费"执行,那么这个移动不增加代价,但是它仅在控制流到达 *src* 时获益。

如果 *dst* 不支配 *src*,那么存在没有首先经过 *dst* 而到达 *src* 的路径。在这种情况下,需要沿这样的路径插入被移动操作的副本。在 9.5 节讨论部分冗余删除时,已经知道怎样插入代码:把该被移动操作的副本放置到形成一个割集(分离入口和 *src*)的各基本块中。在每步插入被移动操作的位置,下面的约束必须得到满足:

(1) 该被移动操作的操作数必须和在原来位置时的值一样;

(2) 其结果不会覆盖还将使用的值;

(3) 在到达 *src* 以前,该结果不会被随后操作覆盖。

这些副本使得 *src* 中原来的那个操作完全冗余,因而可以把它删除。

被移动操作的额外副本被称为**补偿代码**。如 9.5 节讨论的那样,新基本块可以沿关键边插入以构成保存这些副本的地方。这些补偿代码可能会使得某些路径的执行变得慢一些。因此,只有在被优化路径比未优化路径执行得更频繁时,这种代码移动才改进程序的执行。

10.4.3 向下的代码移动

现在考虑把一个操作沿控制流从块 *src* 向下移动到块 *dst*。它可用类似于上一小节方式来推理代码移动。

如果 *src* 和 *dst* 等价,则情况和上一小节说的一样。

如果 *src* 不支配 *dst*,那么存在没有首先经过 *src* 而到达 *dst* 的路径。同样,在这样的路径上额外操作将被执行。不幸的是,向下移动的代码经常是存储操作,它们有覆盖旧值的副作用。这个问题的解决可以通过复制从 *src* 到 *dst* 路径上的各基本块,并且把被移动操作仅放置在 *dst* 的新副本中。如果有判定指令可用的话,另一种方式则是采用判定指令,通过用看守 *src* 块的判定条件来看守被移动操作。注意,判定指令只能被调度到由该判定条件的计算支配的基本块中,因为该判定条件在其他地方不可用。

如果 *dst* 没有后支配 *src*,那么如上面讨论的那样,必须插入补偿代码以保证被移动操作在所有不访问 *dst* 的路径上都执行。除了被移动代码的副本被放置在 *src* 下的一个割集(分离 *src* 和出口)中外,这个变换仍然类似于局部冗余删除。

可以从上面的讨论小结一下上下两个方向的代码移动。全局代码移动涉及的因素包括利益、代价和实现的复杂性等。表 10.1 给出各种代码移动的小结,表中四行分别对应到下面四种情况。

表 10.1 代码移动的小结

	向上:*src* 后支配 *dst*	向上:*dst* 支配 *src*	投机	补偿代码
	向下:*src* 支配 *dst*	向下:*dst* 后支配 *src*	代码复制	
1	是	是	否	否
2	否	是	是	否
3	是	否	否	是
4	否	否	是	是

(1) 在控制等价的块之间移动指令是最简单的,并且是最经济的。这时无额外的操作被执行,也不需要补偿代码。

(2) 在向上(向下)的代码移动中,如果 *src* 块没有后支配(支配)*dst* 块,则可能会执行额外的操作。如果该额外操作可以免费执行并且通过 *src* 块的路径被执行,则代码移动是有益处的。

(3) 在向上(向下)的代码移动中,如果 *dst* 块没有支配(后支配)*src* 块,则需要补偿代码。增加了补偿代码的路径的执行可能会变慢,所以重要的是被优化路径的执行频率应该较高。

(4) 这种情况集中了第(2)和第(3)种情况的缺点,也就是既可能执行额外操作,又需要补偿代码。

10.4.4 更新数据相关

下面的例 10.9 表明,代码移动会改变操作之间的数据相关关系。因此在每次代码移动后数据相关必须被更新。

例 10.9 在图 10.8 中的流图中,两个对 x 的赋值之一可以移动到最上面的那个基本块,因为该变换能维持原来程序中的所有相关性。但是,如果其中一个对 x 的赋值被上移,则另一个就不能移动了。具体来说,移动前变量 x 在最上面块的出口不是活跃的,移动后成为活跃的。如果一个变量在某个程序点活跃,则不能把对该变量的投机定值移到该程序点的上面。 □

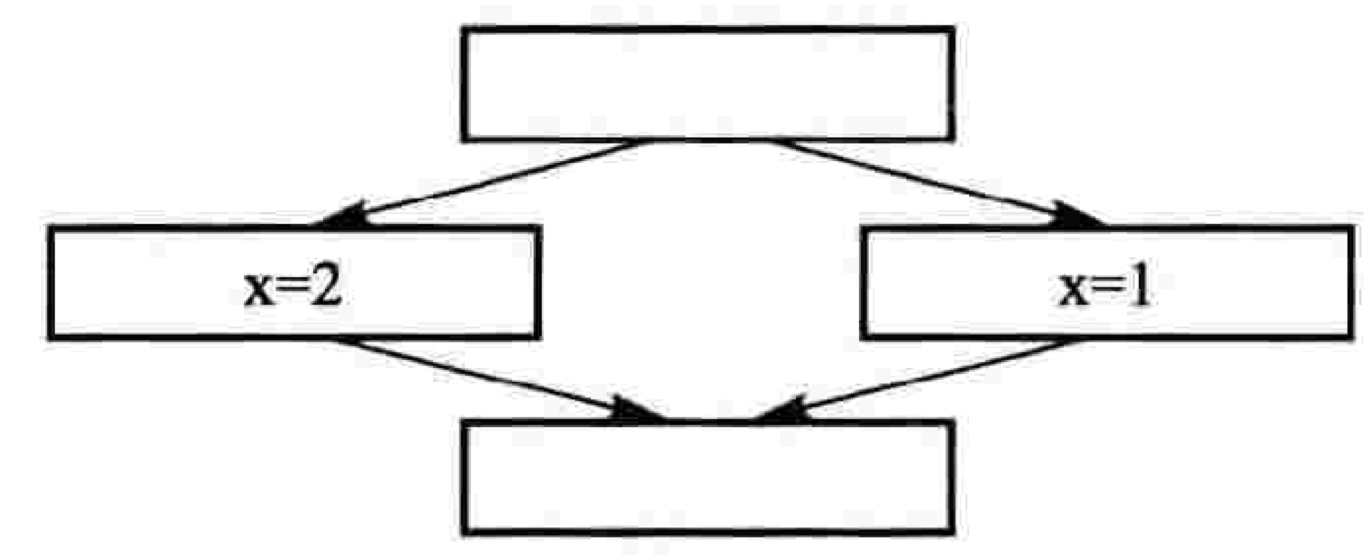


图 10.8 解释由于代码移动引起数据相关性改变的例子

10.4.5 全局调度的其他问题

从上面看到,代码移动使某些路径受益,但也可能损害了其他路径的性能。由于程序 90% 的执行时间消耗在低于 10% 的代码上,因此程序调度应该使经常执行的路径运行得快一些,而不经常执行的路径可能会因这种调度变得慢一些。

编译器可用来估计执行频率的技术有若干种。下面的一些假设是合理的:最内循环中的指令比较外循环中的指令执行得更频繁,分支指令往回跳转比不跳转要更经常些,看守程序出口或异常处理例程的分支语句很少被执行。最好的频率估计来自动态剖析。在这种技术中,程序被静态插桩(instrument)以用来记录运行时条件分支每次的走向。然后程序在有代表性的输入上执行,以推断它在一般情况下的行为。从这种技术得到的结果相当精确,这些信息反馈给编译器,用在它的代码优化中。

最简单的全局调度算法也相当复杂,并且涉及本书没有介绍的一些概念,因此本书不介绍全局调度算法。下面再说一个和全局调度有关的循环展开问题。

在一些全局调度算法中,循环迭代的边界是代码移动的一种屏障。一次迭代中的操作不可以和另一次迭代中的操作有任何重叠。打破这种屏障的一种简单而非常有效的技术是在代码调度前把循环展开几次迭代,使循环体中有更多的指令,从而全局调度算法有机会发现更多的并行性。图 10.9 给出了 for 循环展开的示例。

```
for(i=0;i<N;i ++) {
    S(i);
}
```

(a) 展开前的 for 循环

```
for(i=0;i+4<N;i += 4) {
    S(i);
    S(i+1);
    S(i+2);
    S(i+3);
}
for( ;i<N;i++) {
    S(i);
}
```

(b) 展开后的 for 循环

图 10.9 for 循环展开示例

10.4.6 静态调度器和动态调度器的相互影响

动态调度器的优点是可以根据运行时的情况建立新的调度表,而不需要事先编码所有可能的调度表。如果目标机器有动态调度器,那么静态调度器的基本功能是保证尽早地取长延迟的指令,使得动态调度器能够尽早发射它们。

缓存未命中是一类不可预测的事件,它们会给程序性能带来很大影响。如果数据预取指令可用,静态调度器通过尽早安排预取指令,使得数据到要用时已经在缓存,给动态调度器以很大帮助。如果没有预取指令可用,编译器可以预测哪些操作很可能不命中并尽早安排它们,这对动态调度器来说也是有用的。

如果目标机器上无动态调度可用,则静态调度器必须是稳妥的,它通过极小化延迟来分离每一对数据相关的操作。但是,如果动态调度可用,则编译器只需要给数据相关的操作安排正确的次序以保证程序的正确性。为了获得最好性能,编译器应该分配长延迟到很可能出现的数据相关,而分配短延迟到那些不大可能出现的数据相关。

分支预测错误是性能损失的一个重要原因。由于长预测错误的处罚(long misprediction penalty),在较少执行的路径上的指令仍然会对总执行时间有重要影响。因此应该给这样的指令较高优先级,以减少预测错误的代价。

10.5 软件流水

如本章开头所介绍,数值应用中有很多并行性,尤其是经常使用各次迭代之间相互独立的循环。这些循环称为 do-all 循环,从并行化的观点来看它们特别有吸引力,因为这种循环的各次迭代可以并行执行,以获得按该循环次数的线性加速。迭代次数很多的 do-all 循环有足够的并行

性使一个处理器上的资源饱和，调度器可以充分利用这种并行性。本节描述一种叫做**软件流水**的调度算法，它每次调度一个完整的循环，以充分利用穿越迭代的并行性。

10.5.1 引言

本节使用例10.10的do-all循环来解释软件流水。首先说明穿越迭代的调度非常重要，因为在单次迭代的操作中几乎没有什么并行性。然后通过重叠被展开循环的计算来说明循环展开可以改进性能。但是，被展开循环的边界仍然是代码移动的屏障，因而循环展开仍然有许多性能问题有待讨论。另一方面，软件流水技术不断地重叠一些相继迭代，直到所有迭代都填入流水线为止。这种技术使得软件流水产生高效和紧凑的代码。

例10.10 下面是一个典型的do-all循环：

```
for(i=0;i<n;i ++)
    D[i]=A[i]*B[i]+c;
```

在该循环中，各次迭代写不同的内存单元，并且它们同任何读单元都有区别。因此这些迭代之间没有任何内存相关，所有的迭代可以并行处理。

本节使用下面的模型作为目标机器。在该模型中：

(1) 该机器在1周期内可以同时发射一个读取、一个存储、一个算术运算和一个分支操作。

(2) 该机器有形式为

```
BL R,L
```

的循环回跳操作，它使得寄存器R的值自动减1，除非R的结果为0，否则跳转到单元L。

(3) 内存操作有一种自动增量的寻址模式，用寄存器后的++表示。该寄存器的值自动地增加，指向当前访问地址的后继地址。

```
     // R1,R2,R3=&A,&B,&D
     // R4=c
     // R10=n
L:LD  R5,0(R1++)
  LD  R6,0(R2++)
  MUL R7,R5,R6
  NOP
  ADD R8,R7,R4
  NOP
  ST  0(R3++),R8,     BL R10,L
```

图10.10 例10.10局部调度后的代码

(4) 算术操作是全流水的，它们可以在任意周期启动，但是它们的结果在2周期后才能用。所有其他指令只有1周期的延迟。

如果循环迭代每次一个地被调度,那么能得到该机器模型上最好调度见图 10.10,一些关于数据布局的假定也在该图中给出:寄存器 R1、R2 和 R3 分别保持数组 A、B 和 D 的开始地址,寄存器 R4 保存常量 c,并且寄存器 R10 保存值 n,它是在迭代外计算的。该计算大部分是串行的,它需要 7 周期;只有循环回跳指令和迭代中最后一条指令重叠。 □

一般来说,展开一个循环的几次迭代可以改进硬件利用率。但是,这样做也增加了代码的规模,即它对整体性能带来负面影响。因此必须取一个折中,展开一个循环若干次以获得最好的性能改进,而又不让代码膨胀得过大。下面是一个说明这种折中的例子。

例 10.11 在例 10.10 的例子中,虽然很难发现在一次迭代中的并行性,但是却有很多穿越迭代的并行性。循环展开把循环中几次迭代的代码放在一个大基本块中,然后一个简单的表调度算法就可用来让这些操作并行执行。如果把该例的循环展开为四次迭代,然后把算法 10.1 应用于展开后的代码,得到的调度表见图 10.11(为简单起见,忽略了寄存器分配的细节)。展开后每次迭代的执行用 13 周期,即原来的每次迭代仅需要 3.25 周期。

对例 10.10 来说,展开 k 次迭代的循环至少需要 $2k+5$ 周期,那么获得原先每次迭代仅需 $2+5/k$ 周期的吞吐能力。展开的迭代次数越多,该循环运行得越快。当 k 趋向于 ∞ 时,一个完全展开循环的执行可以使原先每次迭代平均只需 2 周期。然而,展开的迭代次数越多,代码的规模就越大,显然机器可能供不起展开所有的迭代。 □

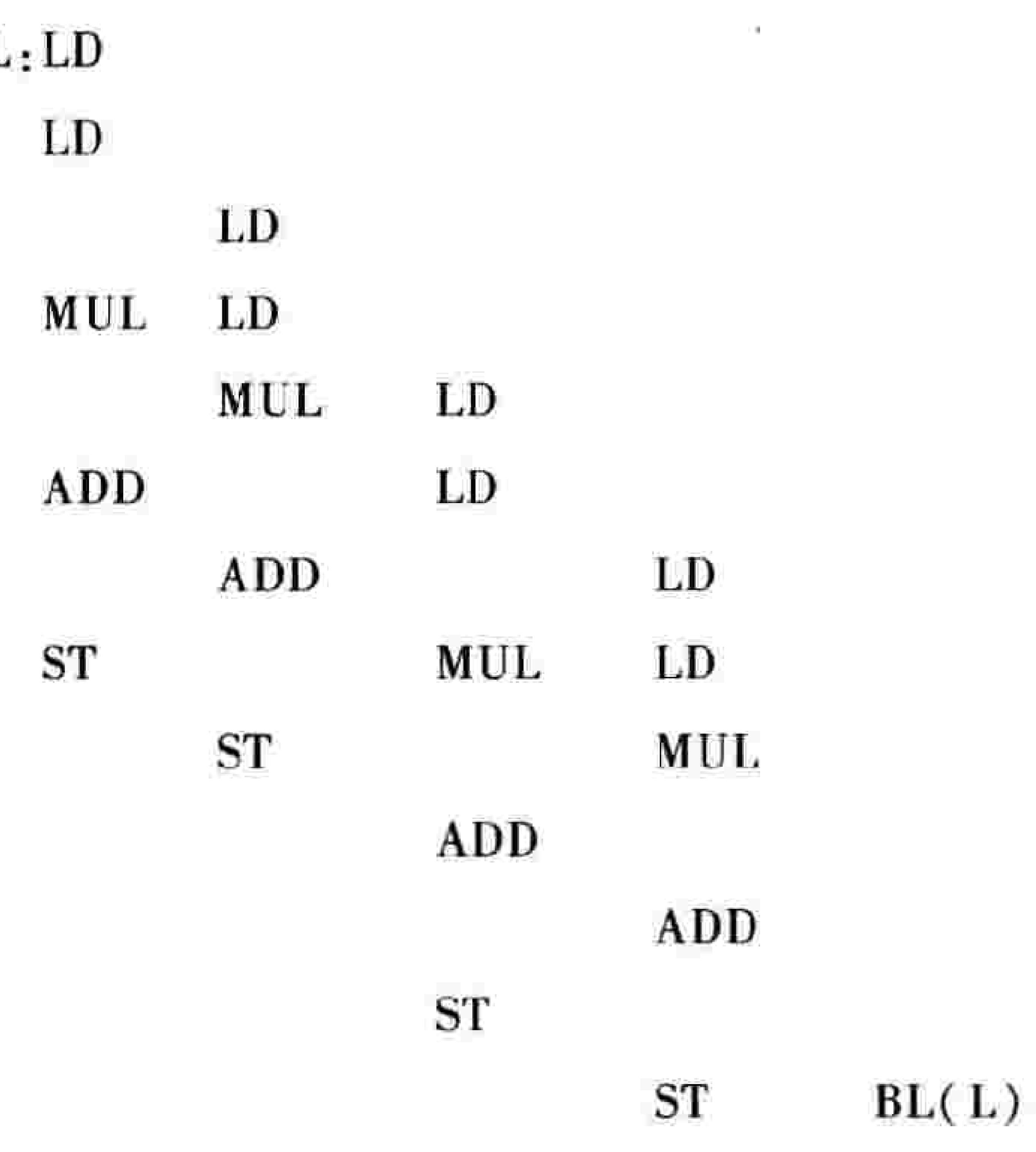


图 10.11 例 10.10 的展开代码

10.5.2 循环的软件流水

软件流水提供一种便利的方式来获得优化的资源利用并同时紧凑代码。仍用例子说明这一点。

例 10.12 在图 10.12 中,例 10.10 的代码展开 5 次(同样,在此没有考虑寄存器分配)。第 i

行是在第 i 周期发射的所有操作;第 j 列是第 j 次迭代的所有操作。注意,每次迭代相对它们的开始点有同样的调度,并且每次迭代的启动比前一次迭代滞后 2 周期。很容易看出该调度满足所有的资源和数据相关约束。

还可以看出,在第 7 和 8 周期执行的操作同第 9 和 10 周期执行的操作是一样的。第 7 和 8 周期执行的操作源于原来程序的前 4 次迭代,而第 9 和 10 周期执行的操作源于第 2 到 5 次迭代。事实上,可以持续执行同样的多操作指令对,以有效地撤出老的迭代并加入新的迭代,直到所有的迭代都填入为止。

如果该循环至少有 4 次迭代,那么这样的动态行为可以简洁地编码成图 10.13 的代码。该图中每行对应于一条机器指令。第 7 和第 8 两行构成一个 2 时钟的循环,重复执行 $n-3$ 次,其中 n 是原来循环的迭代次数。 □

上面描述的技术叫做软件流水,因为它是类似于调度硬件流水线技术的软件。在该例中,由每次迭代执行的调度可以看成 8 级流水线,每经过 2 周期,新一次迭代可以在该流水线上开始。最初只有第 1 次迭代在该流水线上,随着该迭代前进到第 3 级,第 2 次迭代开始执行它的第 1 级。

周期	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
1	LD				
2	LD				
3	MUL	LD			
4		LD			
5		MUL	LD		
6	ADD		LD		
7			MUL	LD	
8	ST	ADD		LD	
9				MUL	LD
10		ST	ADD		LD
11					MUL
12			ST	ADD	
13					
14				ST	ADD
15					
16					ST

图 10.12　例 10.10 代码经 5 次展开的迭代

在第 7 周期,该流水线由前 4 次迭代填满。进入稳定状态后,相继的 4 次迭代同时执行;当一个老的迭代在该流水线上撤出,会填入一个新的迭代。当所有迭代都填完,该流水线排空,那么所有迭代在该流水线上都完成运行。该例中,填充该流水线的第 1 到 6 行的指令序列叫做序曲,第 7 和 8 行叫做稳定状态,用来排空该流水线的第 9 到 14 行指令序列叫做尾声。

1		LD				
2		LD				
3		MUL	LD			
4			LD			
5			MUL	LD		
6		ADD		LD		
7	L:			MUL	LD	
8		ST	ADD		LD	BL(L)
9					MUL	
10			ST	ADD		
11						
12				ST	ADD	
13						
14					ST	

图 10.13 例 10.10 经软件流水化的代码

在该例中，循环的运行速度不可能快于每 2 周期一次迭代，因为机器在每周期只能发射一个读操作，而每次迭代有 2 个读操作。该例被软件流水化的循环需要执行 $2n+6$ 周期，其中 n 是原来循环的迭代次数。当 n 趋于 ∞ 时，该循环的吞吐能力逼近每 2 周期完成一次迭代的速度。因此，和循环展开不一样的是，软件调度能以非常紧凑的代码序列来编码最优调度。

注意，为单次迭代采用的调度可能不是最短的。比较图 10.10 的局部最优调度可以看出，那里在 ADD 操作前引入一个延迟，该延迟的安排是战略性的，以保证该调度能在每 2 周期启动内而没有资源冲突。如果在这里采用局部最优调度，启动间隔就不得不加长到 4 周期，以避免资源冲突，那么吞吐能力就降低一半。该例说明流水线调度的一个重要原则：必须仔细安排调度以最优化吞吐能力。局部紧凑的调度虽然极小化了单次迭代的时间，但把它流水化时可能得不到最理想的吞吐能力。

10.5.3 寄存器分配和代码生成

下面为例 10.12 中软件流水化的循环讨论寄存器分配。

例 10.13 在例 10.12 中，第一次迭代乘运算的结果在第 3 周期产生，在第 6 周期使用。在此期间，第二次迭代乘运算的结果在第 5 周期产生；该结果在第 8 周期使用。显然，这两次迭代乘运算的结果必须保存在不同的寄存器中，以防止它们相互干扰。由于相互干扰仅出现在相邻的一对迭代之间，因此这种干扰可以通过使用两个寄存器分别保存相邻迭代的乘运算结果来解决。这样，奇数次迭代和偶数次迭代的目标代码不完全一样，所以进入稳定状态后的循环由 2 周期加倍成 4 周期。

对迭代次数大于或等于 5 的任何奇数次迭代，很容易写出相应的代码。为了能够处理少于

5 次的迭代和偶数次迭代的循环,应该产生对应于图 10.14 源代码的目标代码,其中第一个循环处理迭代次数大于或等于 5 的任何奇数次迭代,该循环被流水化,它对应的目标代码见图 10.15。第 2 个循环不需要优化,因为它最多 4 次迭代。 □

```
if(N >= 5)
    N2 = 1+2 * ((N -1) / 2);
else
    N2 = 0;
for(i=0;i<N2;i ++)
    D[i] = A[i] * B[i]+c;
for(i=N2;i<N;i ++)
    D[i] = A[i] * B[i]+c;
```

图 10.14 例 10.10 循环的源级展开

```
(1)       LD R5,0(R1++)
(2)       LD R6,0(R2++)
(3)       LD R5,0(R1++)  MUL R7,R5,R6
(4)       LD R6,0(R2++)
(5)       LD R5,0(R1++)  MUL R9,R5,R6
(6)       LD R6,0(R2++)  ADD R8,R7,R4
(7)   L:  LD R5,0(R1++)  MUL R7,R5,R6
(8)       LD R6,0(R2++)  ADD R8,R9,R4  ST 0(R3++),R8
(9)       LD R5,0(R1++)  MUL R9,R5,R6
(10)      LD R6,0(R2++)  ADD R8,R7,R4  ST 0(R3++),R8  BL R10,L
(11)                     MUL R7,R5,R6
(12)                     ADD R8,R9,R4  ST 0(R3++),R8
(13)
(14)                     ADD R8,R7,R4  ST 0(R3++),R8
(15)
(16)                                   ST 0(R3++),R8
```

图 10.15 例 10.13 经软件流水和寄存器分配后的代码

10.5.4 Do-Across 循环

软件流水也可以用于迭代之间存在数据相关的循环,这样的循环叫做 do-across 循环。

例 10.14 代码

```
for(i=0;i<n;i ++) {
    sum = sum+A[i];
    B[i] = A[i] * b;
```

}

在连续的迭代之间存在数据相关,因为 A[i]加到 sum 先前的值,以建立 sum 新值。如果机器能提供足够的并行,那么执行求和可以在 $O(\log n)$ 的时间内完成,但是为便于讨论,假定所有的串行相关必须遵守,并且加法必须按原来的串行顺序完成。因为先前假设了完成 ADD 需要 2 周期,因此该循环的执行不可能快于每 2 周期一次迭代。该机器即使有更多的加法器或乘法器,也不可能使该循环更快一点。这个例子的 do-across 循环的吞吐能力受到穿越迭代的数据相关链的限制。

对每次迭代的最好紧凑调度见图 10.16(a),软件流水化的代码见图 10.16(b)。软件流水化的代码每 2 周期开始一次迭代,因而它是以最优的速度运算。 □

```
    // R1 = &A; R2 = &B
    // R3 = sum
    // R4 = b
    // R10 = n
L:LD  R5,0(R1++)
    MUL  R6,R5,R4
    ADD  R3,R3,R4
ST  R6,0(R2++)    BL  R10,L
```

(a) 最好的局部紧凑代码

```
   // R1 = &A; R2 = &B
   // R3 = sum
   // R4 = b
   // R10 = n
   LD  R5,0(R1++)
   MUL  R6,R5,R4
L:ADD  R3,R3,R4        LD  R5,0(R1++)
   ST  R6,0(R2++)      MUL  R6,R5,R4      BL  R10,L
                       ADD  R3,R3,R4
                       ST  R6,0(R2++)
```

(b) 软件流水化的版本

图 10.16 do-across 循环的软件流水

10.5.5 软件流水的目标和约束

软件流水的基本目标是极大化耗时较长的循环的吞吐能力,次要目标是保持所产生代码的规模较小。换句话说,软件流水化的循环应该有较小的流水线稳定状态。通过要求每次迭代的相对调度都相同并且这些迭代以同样的时间间隔逐步启动,那么可以获得较小的稳定状态。因

为一个循环的吞吐能力直接就是启动间隔的倒数，因此软件流水的目标是极小化这个间隔。

一个软件流水调度可以由下面两点来规范：

(1) 启动间隔 T。

(2) 相对调度表 S。S 描述一个迭代的每个操作相对于本迭代的执行时刻。

因此，第 i 次（从 0 开始计数）迭代的一个操作 n，在 $i\times T+S(n)$ 周期执行。像所有其他调度问题一样，软件流水有两类约束：资源和数据相关，下面分别讨论。

首先讨论资源约束。令机器资源由 $R=[r_1,r_2,\cdots]$ 表示，其中 r_i 是第 i 类资源可用的部件数目。如果某循环的一次迭代需要第 i 类资源 n_i 个部件，那么流水化循环的平均启动间隔至少是 $\max_i(n_i/r_i)$ 周期。软件流水要求任何一对迭代之间的启动间隔是常数，因此启动间隔至少是 $\max_i\lceil n_i/r_i\rceil$ 周期。如果 $\max_i(n_i/r_i)$ 小于 1，将源代码展开少数几次是有用的。

例 10.15 现在回到图 10.13 软件流水化循环。前面已经说过，该目标机器在每周期内可以同时发射一个读取、一个存储、一个算术运算和一个循环回跳操作。因为该循环有 2 个读取、2 个算术运算和 1 个存储操作，因此根据资源的约束，最小启动间隔是 2 周期。

从图 10.13 可以看出 4 次相继迭代越过迭代边界的资源需求。随着各迭代逐步启动，越来越多的资源被占用，资源占用的高峰在稳定状态。令 RT 代表单次迭代的资源预约表，并且 RT_s 代表稳定状态的资源预约表。RT_s 组合了启动间隔为 T 的 4 次相继迭代的资源预约，表 $RT_s[0]$ 对应于 $RT[0]$、$RT[2]$、$RT[4]$ 和 $RT[6]$ 的资源之和，$RT_s[1]$ 对应于 $RT[1]$、$RT[3]$、$RT[5]$ 和 $RT[7]$ 的资源之和。因此稳定状态第 i 行的资源预约由下式给出

$$RT_s[i]=\sum_{\{t\mid(t \bmod 2)=i\}} RT[t]$$

代表稳定状态的资源预约表称为该流水化循环的模资源预约表（modular resource-reservation table）。

为了检查软件流水调度是否会出现资源冲突，只需要检查模资源预约表就可以了。肯定地说，如果该表的需求能够满足，则流水线的序曲和尾声部分肯定也能满足。 □

一般而言，给定启动间隔 T 和单次迭代的资源预约表 RT，在资源向量为 R 的机器上，如果对所有的 $i=0,1,\cdots,T-1$，都有 $RT_s[i]\leqslant R$，那么流水化的调度不会出现资源冲突。

再讨论数据相关约束。软件流水中的数据相关不同于到目前为止碰到的数据相关。一个操作可能依赖于前一次迭代中同样操作的结果，这时仅用延迟来标记边已经不够用了，需要区别不同迭代中同一操作的实例。对于边 $n_1\to n_2$，如果在第 i 次迭代中的操作 n_2，至少等第 $i-\delta$ 次迭代中的操作 n_1 开始执行 d 周期后才能执行，则该边上的标记是 (δ,d)。令 S 是软件流水调度函数，它是从数据依赖图的结点到整数的函数，并且令 T 是启动间隔。那么，

$$(\delta\times T)+S(n_2)-S(n_1)\geqslant d$$

必须满足，其中迭代次数差 δ 必须非负。而且对于数据依赖图 G 上的一个环，至少有一条边上标记的迭代次数差是正的。

例 10.16 考虑下面的循环

```
for(i=0;i<n;i ++)
```

```
*(p++)= *(q++)+c;
```

并且假定不知道 p 和 q 的值。为稳妥起见,必须认为任何一对 *(p++)和 *(q++)访问可能引用同样的内存单元。因此所有的读写必须按它们原来的次序执行。假定目标机器的特征同例 10.10 所说的一样,则这段代码的数据依赖图 G 如图 10.17 所示的那样。注意,循环控制指令被忽略。 □

两个相关操作的迭代次数差可以大于 1,下面是一个这样的例子:

```
for(i=2;i<n;i++)
    A[i]=B[i]+A[i-2]
```

显然,存储 A[i]和读取 A[i-2]的依赖边上标记的迭代次数差是 2。

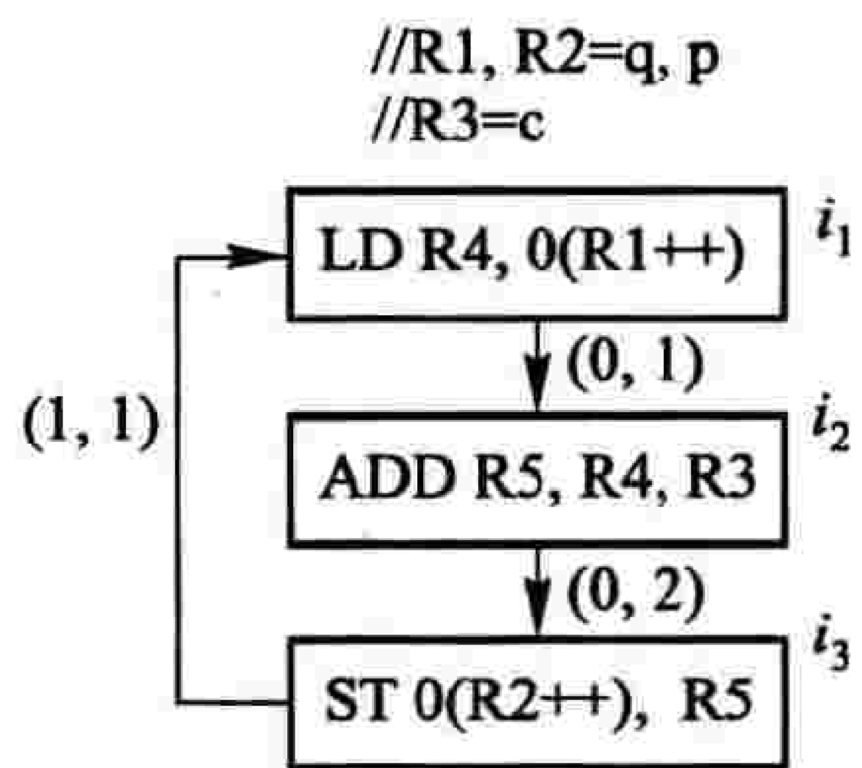


图 10.17 例 10.16 的数据依赖图

循环中数据依赖环的出现对程序执行的吞吐能力强加了另一个限制。例如图 10.17 的数据依赖环在相继迭代的两个读取操作之间强加了 4 周期的延迟。因此,该循环的执行速度不可能快于每 4 周期一次迭代。

流水化循环的启动间隔 T 不能小于

$$\max_{c \text{ is a cycle in } G} \left\lceil \frac{\sum_{e \text{ in } c} d_e}{\sum_{e \text{ in } c} \delta_e} \right\rceil$$

周期。

总之,软件流水化循环的启动间隔受到每次迭代中资源使用情况的限制:对任何一种资源,启动间隔不能小于所需部件数和该机器所能提供部件数的比值。还有,如果循环的数据依赖图中存在环,则启动间隔会进一步受到约束:启动间隔不能小于环中延迟的和除以迭代次数差的和。这些量中最大的一个定义了启动间隔的一个下界。

10.5.6 软件流水算法

软件流水的目标就是找到一种最小启动间隔的调度,该问题是 NP 完全的,并且能够被形式化为一个整数线性规划问题。从前面讨论可知,如果知道了最小启动间隔,则调度算法可以利用模资源预约表来安排每个操作,以避免资源冲突。但是,只有找到了一种调度,才能知道最小启动间隔。如何解决这两者之间的相互依赖?

从前面的讨论知道,启动间隔必须大于或等于从一个循环的资源需求和数据依赖环算出的下界。如果能够找到一种正好等于这个下界的调度,那么就是找到了最优调度。如果找不出这样的调度,则可以用较大一点的启动间隔再尝试,直至找到一种调度。注意,如果采用启发式,而不是穷尽搜索,则有可能找不到最优调度。

能否找到接近下界的调度取决于数据依赖图的性质和目标机器的体系结构。如果数据依赖图无环并且每条机器指令仅需要一种资源的一个部件,则很容易找到最优调度。如果有足够的硬件资源供带环的数据依赖图使用,则也很容易找到接近下界的调度。对这种情况,很明显可以从把下界作为初始启动间隔开始,然后每次加 1 周期再尝试,直至找到合适的启动间隔。另一种可能方式是用二分查找来寻找启动间隔。启动间隔的上界是表调度为一次迭代产生的调度的长度。

10.5.7 无环数据依赖图的调度

为了简单起见,假定被软件流水化的循环只有一个基本块。

算法 10.2 无环依赖图的软件流水。

输入 机器资源向量 $R=[r_1,r_2,\cdots]$,其中 r_i是机器第 i 种资源的可用部件数目;另外还有数据依赖图 $G=(N,E)$。N 中的每个操作 n 都带有它的资源预留表 RT_n,E 中每条边 $e=n_1\rightarrow n_2$标有(δ_e,d_e),表示 n_2必须在比本次早 δ_e次的迭代的 n_1开始 d_e周期后才能开始。

输出 一张软件流水化调度表 S 和一个启动间隔 T。

方法 执行图 10.18 的程序。 □

算法 10.2 首先依据图 G 中各操作的资源需求,计算出启动间隔的下界 T_0。然后尝试把 T_0 作为启动间隔来寻找软件流水化调度。该算法通过逐步增加启动间隔来重复,如果它还没有找出一种调度的话。

在每次尝试时,该算法使用一种表调度方式。它利用模资源预约表 RT 来确定稳定状态中的资源需求。操作按照区分优先级的拓扑次序来调度,因此数据相关总可以通过延迟操作而得到满足。调度每一个操作时,它首先根据数据相关的约束找出一个下界 s_0,然后调用 *NodeScheduled* 函数来检查在此稳定状况中是否存在资源冲突。如果存在冲突,它尝试把该操作调度到下一周期。如果该操作在 T 个连续周期的调度尝试都失败,那么由资源冲突检测的模本性知道,继续尝试是不会有结果的。因此在这一点,该算法认为当前尝试的启动间隔不可采用,转而尝试较大一点的启动间隔。

启发式的调度尽可能极小化一次迭代的调度表长度,然而,尽可能早地调度一条指令会导致某些变量的生存期变长。例如,变量的读取操作倾向于尽早调度,有时会出现远早于它们的使用的情况。一种简单的启发式方法是逆向调度数据依赖图,因为通常读取操作多于存储操作。

本书不再往下介绍有环数据依赖图的调度。

```
main() {
    T_0 = max_j ⌈ Σ_{n,i} RT_n(i,j) / r_j ⌉ ;
    for(T=T_0, T_0+1, …, 一直到 N 中的所有结点都被调度) {
        RT=一张 T 行的空资源预约表;
        for(N 中按照区分优先级的拓扑次序的每个结点 n) {
            s_0 = max_{e=p→n in E}(S(p)+d_e);
            for(s=s_0, s_0+1, …, s_0+ T -1)
                if(NodeScheduled(RT,T,n,s)) break;
            if(n 在 RT 中不能被调度) break;
        }
    }
}

NodeScheduled(RT,T,n,s) {
    RT' =RT;
    for(RT_n 中每行 i)
        RT'[(s+i) mod T]=RT'[(s+i) mod T]+RT_n[i];
    if(对所有的 i,RT'[i] ≤ R ) {
        RT=RT';
        S(n)=s;
        return true;
    }
    else return false;
}
```

图 10.18 无环依赖图的软件流水算法

10.6 并行性和数据局部性优化概述

本节介绍同并行性和数据局部性优化有关的基本概念。如果操作可以被并行地执行,则它们也可以因数据局部性或其他目的而被重排执行次序。反过来,如果一个程序中的数据相关限定指令必须串行执行,那么显然不存在并行性,也不可能有机会重排执行次序以改善数据局部性。因此并行化分析同时也是发现改进数据局部性的机会。

为了极小化并行代码中的通信,需要把所有相关的操作聚合在一起,并把它们指派到同一台处理器,这样的结果代码肯定有较好的数据局部性。在单处理器上获得良好数据局部性的一种

天然方式是让该处理器相继执行指派给每台处理器的代码。

本节首先介绍并行计算机系统结构的概况,然后给出并行化的基本概念、程序循环的变换和对并行化有用的概念,再讨论类似的考虑怎样用于优化数据局部性,最后给出矩阵乘算法的优化示例。

10.6.1 多处理器

最流行的并行机体系结构是对称多处理器。高性能个人计算机常常有两个处理器,许多服务器有4个、8个甚至10个处理器。而且,由于把多个高性能处理器集成在一块芯片上是可行的,所以多处理器已经被广泛使用。

对称多处理器上的各处理器共享同样的地址空间。为了通信,一个处理器直接写一个内存单元,该单元然后被其他某个处理器读取。对称多处理器之所以这样命名是因为所有处理器以统一的存取时间访问该系统中的所有内存。图10.19显示一种多处理器的高层体系结构。各处理器可以有它们自己的一级缓存和二级缓存,在某些情况下甚至有三级缓存。级数最高的缓存通过共享总线连接到内存。

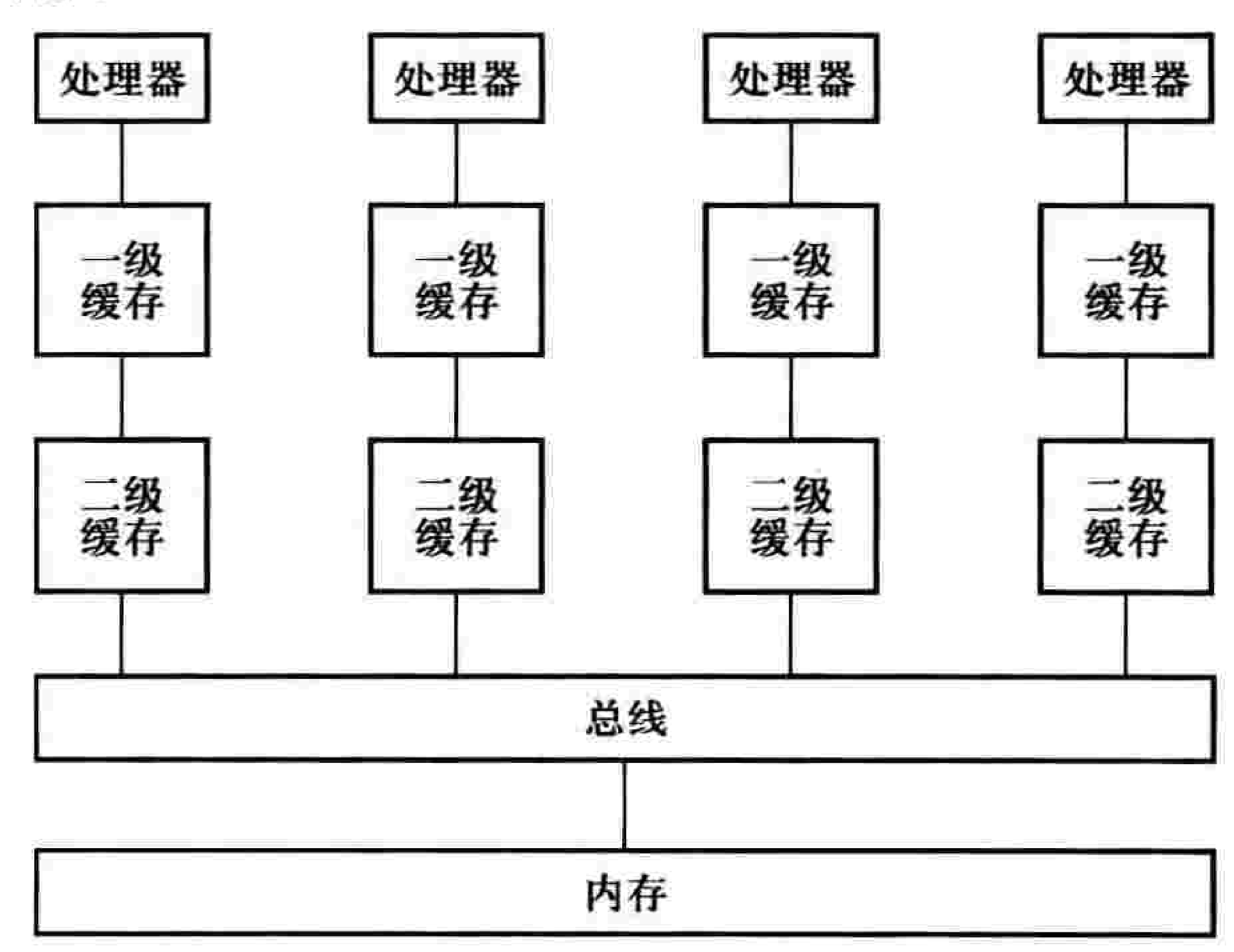


图10.19 对称多处理器的体系结构

对称多处理器使用**一致性缓存协议**来对程序员隐藏缓存的存在。在这样的协议下,允许多个处理器同时保持同样缓存行的副本,只要它们都只读取这些数据。当一个处理器打算写一个缓存行时,它在其他缓存上的副本被删除。当一个处理器请求的数据不在它的缓存上时,该请求到达共享总线,该数据从内存或从另一个处理器的缓存取出。

一个处理器和另一个处理器通信所需时间大约是一次内存访问代价的两倍。缓存行中的数据必须首先从第一个处理器的缓存写到内存,然后从内存取到第二个处理器的缓存。你可能认为处理器之间的通信是相对廉价的,因为它不过是一次内存访问时间的两倍。但是,要记住,和

缓存命中相比,内存访问的代价是非常大的,它可能慢一百倍。该分析使人认识到有效并行化和局部性分析之间的相似性。为了使一个处理器(在自身单处理器或多处理器环境上)的性能更好,它必须在它自己的缓存中找到它操作的大部分数据。

21 世纪初的那几年,对称多处理器的设计已经不再超过 10 个处理器的规模,因为共享总线或为此目的的任何其他方式互连,其操作不能像增加处理器个数那样增速。为了使处理器设计可扩展,体系结构师在内存分层中又引入一层。他们将内存分布到各处理器,使得每个处理器能迅速访问它自己的局部内存,如图 10.20 所示。较远的内存则构成内存分层中的下一个层次,它们容量较大,但需要较长的访问时间。类似于存取速度越快则容量越小的内存分层设计原则,支持处理器之间快速通信的机器只有少量处理器。

带分布内存的并行机有两种衍生:非均匀内存访问的机器和消息传递的机器。非均匀内存访问体系结构给软件提供了共享内存空间,允许处理器通过读写共享内存来通信。而在消息传递机器上,各处理器有不相交的地址空间,处理器之间通过发送消息进行通信。相比之下,为共享内存机器写代码是比较容易的;但是不管哪种机器,为获得良好的性能,软件都必须有很好的局部性。

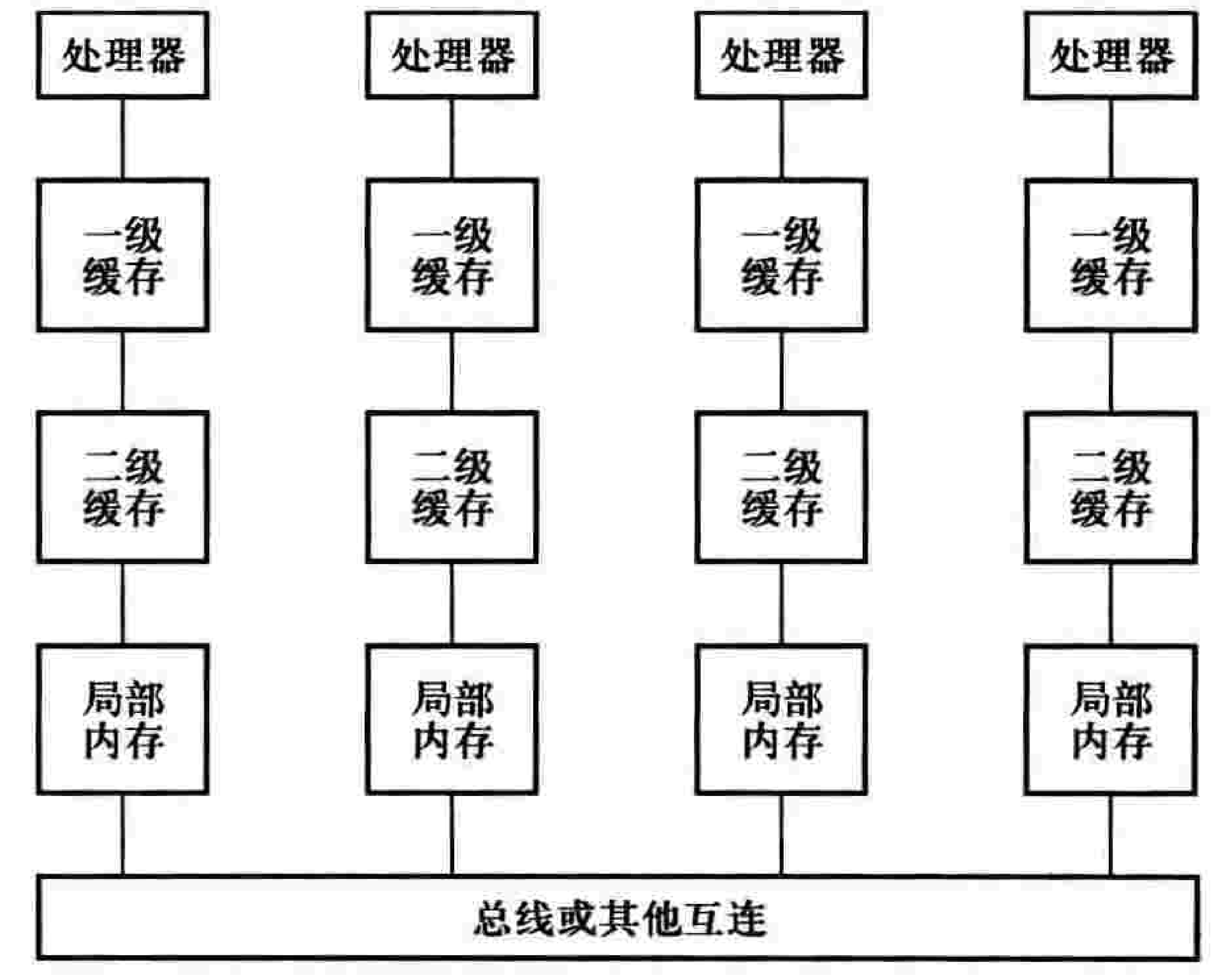


图 10.20 分布式内存机器

10.6.2 应用中的并行性

有两种衡量标准可用来评估并行应用的性能。一种称为**并行覆盖**,它是指整个计算中并行运行部分的百分比;另一种称为**并行粒度**,它是指每台处理器上无须和其他处理器同步或通信的计算量。循环对并行化来说是一个特别有吸引力的地方,一个循环可以有许多次迭代计算,如果这些计算相互独立,则它们是并行计算的主要来源。

并行覆盖的意义由 **Amdahl 定律**简洁地表达。Amdahl 定律说,如果 f 是被并行化的代码所

占的比例,并行化后的版本在一台有 p 个处理器的机器上运行且无通信或并行开销,那么加速比是

$$\frac{1}{(1-f)+(f/p)}$$

例如,若有一半计算仍然是串行的,则不管使用多少个处理器,计算速度最多只能是加倍。如果是 4 个处理器,则加速比是 1.6。即使并行覆盖部分占到 90%,在 4 个处理器的情况下,加速比最多也只能达到 3,处理器个数无限时加速比也只能达到 10。

对于并行粒度,理想的情况是一个应用的整个计算能够划分成许多独立的粗粒度任务,因为这时只需直接把不同的任务指派给不同的处理器即可。一个典型例子是地外文明探索(Search for Extra-Terrestrial Intelligence)项目,它使用连到因特网上各家庭计算机的闲置计算能力来分析世界上最大射电望远镜所获得数据的不同部分,以帮助科学家探索外星生命。参与该项目的每台计算机只需要少量输入,产生少量输出,并且以独立于其他所有参与该项目计算机的方式工作。这样的计算任务能很好地在连到因特网的这些计算机上运行,虽然因特网相对来说有较高的通信延迟和较低的带宽。

大多数应用需要处理器之间更多的通信和交互,但仍然允许粗粒度的并行。例如,考虑一台 Web 服务器,它为对某个公共数据库的大量访问请求提供服务,这些请求大都相互独立。这个应用可以运行在一个多处理器上,用一个线程实现数据库,用一组其他线程来服务于用户请求。另一些例子包括药物设计和机翼模拟,其中许多不同参数的结果可以独立地评估。有时,就是对一次模拟中一组参数的评估也需要很长时间,因而希望通过并行来加速。当一种应用中可用并行的粒度降低时,则需要较好的处理器间通信支持和更多的编程努力。

许多控制结构简单、数据量大并且计算时间长的科学和工程应用,很容易以较细粒度(和上面提到的应用相比)被并行化。下面给出这类程序的一个例子。

10.6.3 循环级并行

循环是并行化的主要对象,特别是那些使用数组的应用。长时间运行的应用一般都使用大数组,这些数组导致程序中出现有许多次迭代的循环,每次迭代用于计算这些数组的一个元素,而且这些迭代经常是相互独立的。可以把这类循环的大量迭代分派到各处理器上。如果在每个处理器上完成的工作大体上是相同的,则直接把这些迭代平分到各处理器上就可以获得最大的并行性。例 10.17 是一个展示怎样利用循环级并行的极其简单的例子。

例 10.17 循环

```
for(i=0;i<n;i++) {
    Z[i]=X[i]-Y[i];
    Z[i]=Z[i]*Z[i];
}
```

计算向量 X 和 Y 对应分量差的平方,并把结果作为向量 C 的对应分量。该循环可以并行,因为每次迭代访问一组不同的数据。在一台有 M 个处理器(它们的唯一标识分别为 p=0,1,…,M-1)的计算机上,该循环可以由每个处理器执行下面相同的代码来完成:

```
b = ceil(n/M);
for(i = b * p;i<min(n,b * (p+1));i++){
    Z[i] = X[i]-Y[i];
    Z[i] = Z[i] * Z[i];
}
```

注意,由于迭代的次数并不一定能被 M 整除,因此通过使用求最小值函数来保证最后一个处理器的执行不会超出原循环的上界。 □

例 10.17 的并行代码是一个单程序多数据的程序。所有的处理器执行同样的代码,但是它由对每个处理器来说是唯一的标识来参数化,所以不同的处理器完成的是不同的任务。典型的情况是,一个处理器作为主处理器,执行该计算的所有串行部分。当主处理器到达该代码的一个并行化部分,它唤醒所有的从处理器,这些从处理器执行该并行化的代码。当到达该并行化代码的末尾,所有这些处理器都参与一个**障碍同步**。在所有其他处理器被允许离开该同步障碍并执行障碍之后的操作之前,一个处理器进入该同步障碍之前的任何操作由系统确保完成。

如果仅并行化像例 10.17 这样的小循环,则结果代码很可能有较低的并行覆盖和相对细粒度的并行。因此更喜欢做的是去并行化一个程序的最外层循环,因为它产生最粗粒度的并行。一个典型的例子是 $n \times n$ 数据集上的二维快速傅里叶变换。也有很多应用没有可以并行并且上规模的最外层循环。这些应用的执行时间经常由耗时的**内核**(Kernel)支配,这样的内核可能是嵌套层次不同的循环组成的几百行代码。在有些情况下,重组内核的计算,通过聚焦在它的局部性上,把它划分成最大部分的、独立的若干计算单元是可能的。

寻找循环迭代以外的并行性也是可能的。例如,可以把两个不同的函数调用或两个不同的循环指派给两个不同的处理器。这种形式的并行被认为是**任务级并行**。对并行化来说,任务级不像循环级那样有吸引力。因为对一个程序而言,独立的任务数是一个常数,它不像典型的循环那样,独立的计算单元随迭代次数增加而增加。而且,任务通常不是等规模的,因此很难保证所有的处理器在所有时间都忙碌。

10.6.4 数据局部性

在并行化程序时,有两个不同的数据局部性概念需要考虑,它们在 6.5.3 节讨论程序局部性时已经被提到。如果某个单元的数据在很短的时间被多次使用,则出现**时间局部性**;如果毗邻单元的数据在很短的时间内都被访问,则出现**空间局部性**。同一个缓存行上的元素一起被使用的情况是空间局部性的一种重要形式。因为某缓存行上的一个元素被需要时,该行上所有元素就都被取到缓存,并且如果它们很快被使用的话,则很可能还在缓存中。由于这种空间局部性将缓

存未命中降到最低，因此使得程度获得明显的加速。

程序的内核经常可以写成语义等价而数据局部性和性能大不相同的多种形式。例 10.18 是表达例 10.17 计算的另一种方法。

例 10.18 下面的程序和例 10.17 一样，计算向量 X 和 Y 对应元素差的平方。

```
for(i=0;i<n;i++) {
    Z[i]=X[i]-Y[i];
}
for(i=0;i<n;i++) {
    Z[i]=Z[i] * Z[i];
}
```

第一个循环计算对应元素的差，第二个循环计算它们的平方。例 10.17 是把这里的两个循环融合成一个循环。实际程序中经常出现本例这样的代码，因为它对向量机来说是一种优化形式，向量机是一种一次能对整个向量完成简单算术运算的超级计算机。

例 10.17 被融合循环的性能较好，因为它有较好的数据局部性。每个差计算出来后就立即计算它的平方。事实上，差通常保存在寄存器中，然后计算它的平方，再把结果写到内存单元 Z[i]，而且就只写这一次。而本例需要取 Z[i]一次，写 Z[i]两次。更糟的是，如果 Z 数组大于缓存，则在第二次使用 Z[i]时需要从内存重新取。因此本例代码执行会慢得多。 □

例 10.19 假定给行为主（见 7.3.2 节）的数组 Z 的所有元素置零。图 10.21(a)和图 10.21(b)的程序分别逐列地和逐行地扫过该数组。这两个程序可以互相变换到对方。根据空间局部性，显然更愿意逐行地给该数组元素置零，因为在一个缓存行中的所有字相继置零。而对于逐列的方式，如果列的大小超过缓存大小的话，虽然每个缓存行在外循环的下次迭代时再次被使用，但是缓存行在再次使用前已经被抛弃。

为了获得最好的性能，应该像例 10.17 那样并行化 10.21(b)的外循环，见图 10.21(c)。 □

```
for(j=0;j<n;j++)
    for(i=0;i<n;i++)
        Z[i][j]=0;
```

(a) 逐列置零

```
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        Z[i][j]=0;
```

(b) 逐行置零

```
b=ceil(n/M);
for(i=b*p;i<min(n,b*(p+1));i++)
    for(j=0;j<n;j++)
        Z[i][j]=0;
```

(c) 逐行并行置零

图 10.21 数组元素置零的顺序和并行代码

上面两个例子说明对数组上的数值进行操作的几个重要特征：

(1) 数组代码经常有许多可以并行化的循环。

(2) 当循环有并行性时,它们的迭代可按任意次序执行,因而可以重新安排它们的计算次序以彻底改进数据局部性。

(3)在创建相互独立的并行计算大单元时,串行执行这些单元往往会产生较好的数据局部性。

10.6.5 矩阵乘法算法

本小节和下一小节以大家所熟悉的矩阵乘法算法为例,来说明即使是简单和易于并行化的程序,优化也是不容易的。通过这两小节的讨论将会明白,和直截了当的程序相比,重写代码会怎样有效地改进数据局部性,让缓存行保存连续数据元素也能够有效地缩短像矩阵乘法这样程序的运行时间。

```
for(i=0;i<n;i++)
    for(j=0;j<n;j++) {
        Z[i][j]=0.0;
        for(k=0;k<n;k++)
            Z[i][j]=Z[i][j]+X[i][k] * Y[k][j];
    }
```

图 10.22 基本的矩阵乘法算法

图 10.22 是一个典型的矩阵乘法程序。它取两个 $n \times n$ 的矩阵 X 和 Y,产生它们的积在第三个 $n \times n$ 的矩阵 Z 中。该图的代码产生 n^2 个结果,每个结果是矩阵 X 的一行和矩阵 Y 一列的内积。显然,Z 每个元素的计算是独立的,它们可以并行地执行。

n 的值越大,该算法计算 Z 每个元素需要的时间就越长。因为这三个矩阵虽然只有 $3n^2$ 个存储单元,但是该算法完成 n^3 次操作,每次操作完成 X 的一个元素和 Y 的一个元素相乘,再把积加到 Z 的一个元素上(后面简称乘加计算)。因此该算法是计算密集型的,原则上内存访问不应该构成瓶颈。

先考虑该程序在单处理器上顺序执行时的行为。最内循环读写 Z 的同一个元素,并且使用 X 的一行和 Y 的一列。Z[i][j]可以很容易保存在寄存器中,因而除了最后一次存储外,无须访问内存。不失一般性,假定矩阵的布局是行为主,并且为简单起见,假定正好 c 个数组元素能够放满一个缓存行。

图 10.23 表达执行图 10.22 最外循环一次迭代时的访问模式。图 10.23 具体表示的是 $i=0$ 时的第一次迭代。当从 X 第一行的一个元素移到下一个元素时,Y 某列的所有元素被访问。从图 10.23 可以看出所假设的矩阵到缓存行的组织,每个小长方形代表保存 4 个数组元素的一个缓存行(该图中取 $c=4, n=12$)。

访问 X 几乎不给缓存带来什么负担。X 的一行仅散布在 n/c 个缓存行上,假定缓存足以放下 X 所有的缓存行,那么对 X 的每一行来说,缓存未命中仅出现 n/c 次。对整个 X 来说一共出

现 n^2/c 次，不可能比它更小了。

但是，当使用 X 的完整一行时，该矩阵乘法算法需要逐列访问 Y 的所有元素。也就是，当 $j=0$ 时，内循环需要把 Y 第一列的所有元素都取到缓存，而这些元素分散在 n 个不同的缓存行中。如果缓存足够大（或 n 足够小）到能保存 n 个缓存行，并且缓存的其他应用也不会把它们从缓存中驱逐，那么当需要 Y 的第二列（$j=1$）时，这些数据已经在缓存，此时不会出现缓存未命中。一直到 $j=c$ 时才需要把 Y 中完全不同的另一组缓存行取到缓存。因此，对外循环的第一次（$i=0$）迭代，缓存未命中次数在 n^2/c 和 n^2 之间，取决于在缓存中的 Y 缓存行，能否从第二层循环的一次迭代驻留到下一次迭代需要它的时候。

在对 $i=1,2,\cdots,n-1$ 逐步完成最外循环的过程中，当需要 Y 时，可能还会出现或者根本不出现缓存未命中。如果缓存足够大，存放 Y 的所有 n^2/c 个缓存行都可以驻留在缓存上，那么就不会再出现缓存未命中。这时该算法的所有缓存不命中共 $2n^2/c$ 次，一半为 X，另一半为 Y。然而，如果缓存只能驻留 Y 的一列数据而不是 Y 的所有数据，那么对最外循环的一次迭代，需要重新将 Y 的所有数据取到缓存。这时缓存未命中的次数是 $n^2/c+n^3/c$，前一部分为 X，后一部分为 Y。最坏情况是，缓存连 Y 的一列数据也不能驻留，则对最外循环的一次迭代，出现 n^2 次缓存未命中，该算法总共有 $n^2/c+n^3$ 次缓存未命中。

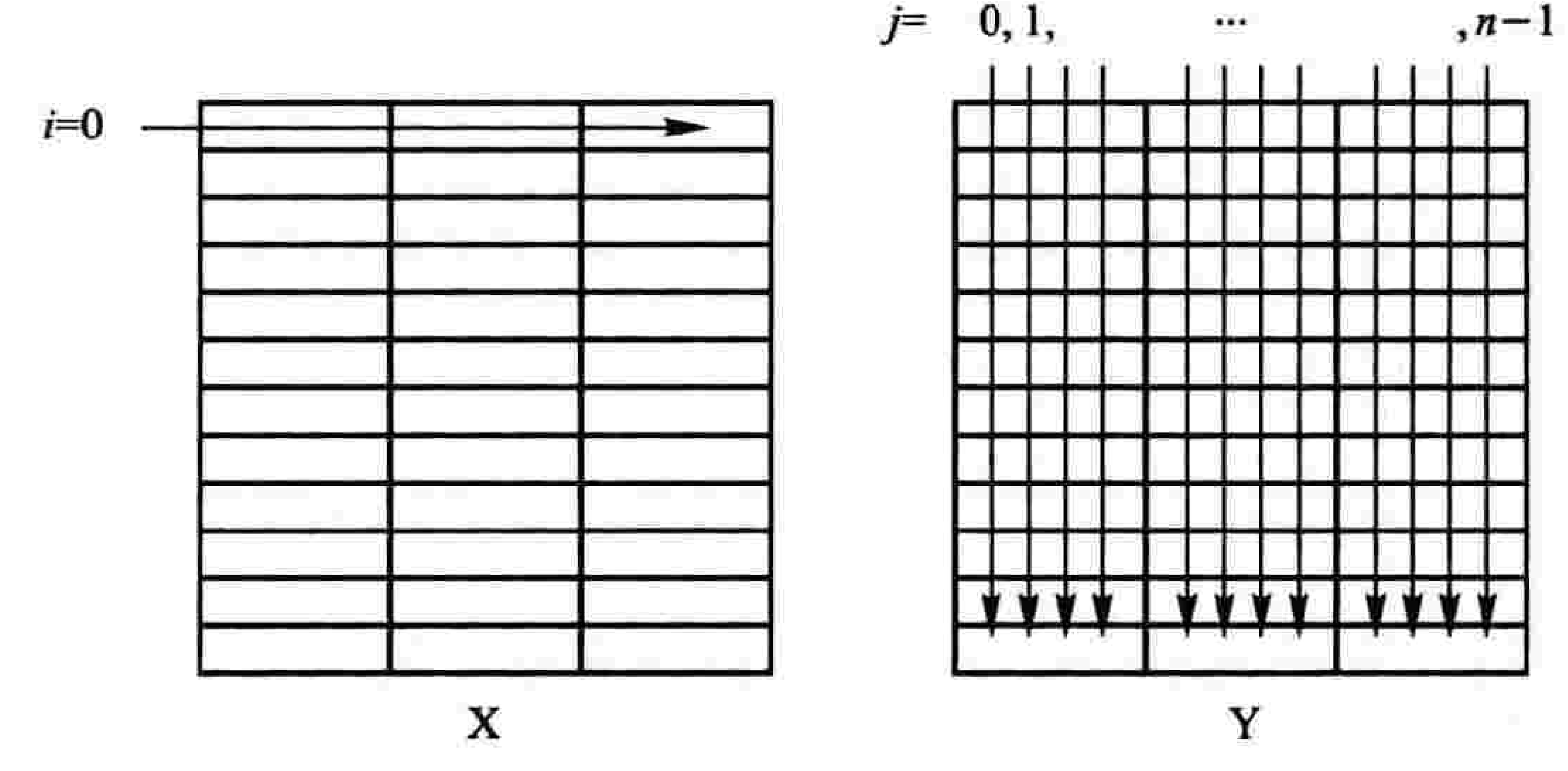


图 10.23 在矩阵乘中数据访问模式

现在考虑怎样使用 p 台处理器来加速图 10.22 矩阵乘法程序的执行。并行化该程序的一种明显方法是把 Z 不同行的计算指派到不同的处理器，每个处理器计算 Z 的连续 n/p 行（为简单起见，假定 n 可以由 p 整除）。这样，每个处理器需要访问矩阵 X 和 Z 的 n/p 行以及整个 Y；每个处理器将完成 n^3/p 次乘加运算来实现对 Z 的 n^2/p 个元素的计算。

虽然计算时间与 p 成比例减少，但通信代价却与 p 成比例增加。因为每个处理器需要读 X 的 n^2/p 个元素和 Y 的所有元素，那么交付给 p 个处理器之缓存的总缓存行是 $n^2/c+pn^2/c$，这两个项分别代表交付 X 和交付 Y 的 p 个副本。当 p 逼近 n 时，计算时间为 $O(n^2)$，而通信代价为 $O(n^3)$，也就是在内存和处理器之间传送数据的总线成为瓶颈。因此，按现在的数据布局，使用大量处理器来分享这些计算的话，不仅不能加速，反而会使计算速度降低。

10.6.6 矩阵乘法算法的优化

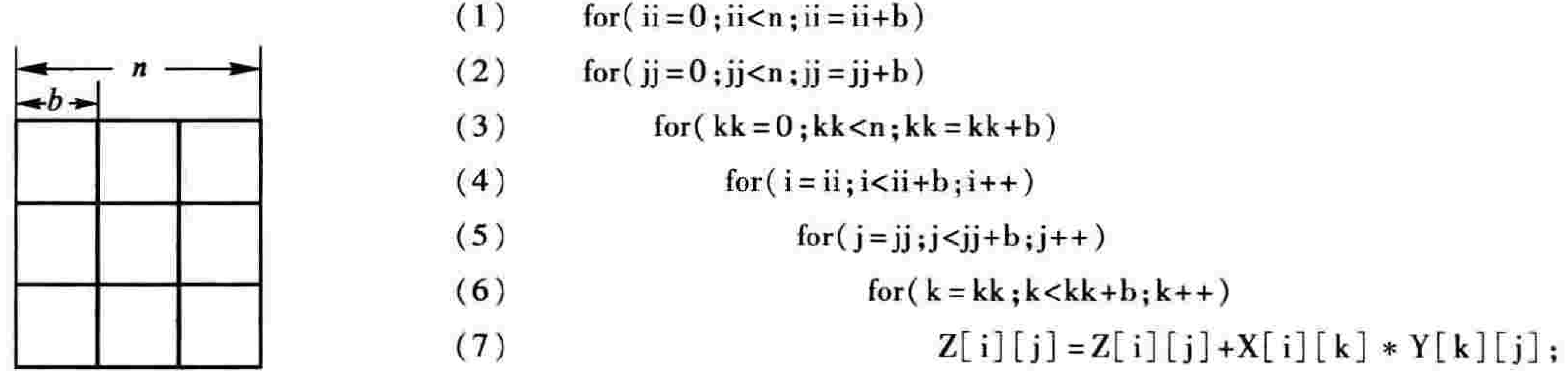
图 10.22 的矩阵乘法算法表明,一个算法复用同样的数据并不代表它的数据局部性就好,关键是看复用数据时能否做到缓存命中。而要做到缓存命中,则复用应该很快发生,而且在数据从缓存转移出去以前。在该算法中,n^2 个乘加操作隔开了矩阵 Y 中同一个数据的复用,n 个乘加操作隔开了 Y 中同一个缓存行的复用。另外,在一个处理器上,数据只有由本处理器复用时才可能出现缓存命中。在上一小节考虑并行实现时已经看到,每个处理器都需要使用 Y 所有的元素。这样的 Y 复用不会带来所期望的数据局部性。

改进程序数据局部性的一种方式是改变它数据结构的布局。例如,若把 Y 改成列为主次序存储的话,就为矩阵 Y 改进了缓存行的复用。但这种方式的可用性有限,因为同一个矩阵通常会用于不同的操作中。如果在另一个矩阵乘法中,Y 扮演 X 的角色,那么列为主的存放就给它带来困难,因为矩阵乘法中,第一个矩阵以行为主的次序为好。

有时,改变指令的执行次序可以改进数据局部性。10.6.4 节已经见过这样的例子。但是,互换内外循环的技术并不能改进矩阵乘法程序。如果互换图 10.22 矩阵乘法程序的最外层和第二层的循环,也就是每次产生 Z 矩阵的一列而不是一行,若矩阵仍然是行为主的次序,那么矩阵 Y 从这种互换中获得较好的空间和时间局部性,但是矩阵 X 的局部性下降了。

分块是重排循环中迭代次序的另一种方法,它能够极大地改进程序的局部性。分块就是把一个矩阵分成若干个子矩阵,或叫做块,如图 10.24 所示。分块是为了以块为单位来完成计算,即整个块在很短的时间内使用,而不是原来那样每次计算矩阵的一行或一列。典型情况下块为正方形,例如长度为 b。如果 b 能够整除 n,那么所有的块都是正方形;否则靠近底边和右边的块有一个边或相邻两边都小于 b。

图 10.25 给出了基本矩阵乘法算法的另一个版本,其中三个矩阵都正好能分成边长为 b 的若干个正方形。如果 b 不能整除 n,则第(4)行的循环上界需要修改成"min(ii+b,n)",并且第(5)和(6)行也要做相应修改。假定矩阵 Z 所有元素已经初始化为 0。




```
(1)  for(ii=0;ii<n;ii=ii+b)
(2)  for(jj=0;jj<n;jj=jj+b)
(3)      for(kk=0;kk<n;kk=kk+b)
(4)          for(i=ii;i<ii+b;i++)
(5)              for(j=jj;j<jj+b;j++)
(6)                  for(k=kk;k<kk+b;k++)
(7)                      Z[i][j]=Z[i][j]+X[i][k]*Y[k][j];
```

图 10.24　矩阵分块

图 10.25　分块的矩阵乘法

从第(1)到(3)行的三个较外循环分别使用 ii、jj 和 kk 作为循环控制变量,并且它们的值每次都增加 b,因此它们总是用来定位块的左上角。对一组确定的 ii、jj 和 kk,从第(4)到(7)行的程序用来计算左上角为 X[ii][kk]和 Y[kk][jj]的两块对左上角为 Z[ii][jj]的块的贡献。

如果 b 取得适当,该算法和图 10.22 的基本算法相比,并且是在缓存不足以装下整个 X、Y 和 Z 的情况下,可以大大减少缓存未命中的数量。现在适当选择 b,使得三个矩阵都各有一个块可以装入缓存。由于这些循环的次序,Z 的每一块实际只需要取到缓存中一次,因此和上一小节对基本算法的分析一样,在这里不考虑由 Z 引起的缓存未命中。

为了把 X 或 Y 的一个块取到缓存,会出现 b^2/c 次缓存未命中,c 仍然是一个缓存行的元素个数。对于 X 和 Y 的一对块,图 10.25 从第(4)到(7)的程序需要完成 b^3 次乘加计算。因为整个矩阵乘法需要 n^3 次乘加计算,所以取一对块到缓存的操作的总次数是 n^3/b^3。由于对于 X 和 Y 的一对块会有 $2b^2/c$ 次缓存未命中,因此缓存未命中的总次数是 $2n^3/bc$。

把 $2n^3/bc$ 和上一小节的估计进行比较是有意义的。在那里的估计是,如果三个完整的矩阵都能装到缓存,那么缓存未命中的数量是 $O(n^2/c)$。在这种情况下,可以取 $b=n$,即让整个矩阵是一个块,则得到的缓存未命中估计再次为 $O(n^2/c)$。另一方面,并非三个完整的矩阵都能装到缓存时,上一小节的估计是出现 $O(n^3/c)$ 次缓存未命中,甚至 $O(n^3)$ 次缓存未命中。在这种情况下,假定 b 能够取得比较大(例如 200,那么 8B 一个元素的三个块是可以装入一个 1 MB 的缓存中),则从 $2n^3/bc$ 可以看出使用分块方法是有很大好处的。

分块技术可以用到内存分层的每一层。例如,可以把一个 2×2 矩阵乘法的运算对象保存在寄存器中,以优化寄存器的使用。可以为不同层次的缓存和物理内存按序选择越来越大的块。

类似地,可以把块分布在各处理器上,以最小化数据通信量。经验已经表明,这样的优化可以使单处理器的性能提升到原来的 3 倍,在多处理器上的加速接近和所使用的处理器个数呈线性关系。

然而,缓存的实际情况不是本节所介绍的这么简单,通常一个缓存行不是能放置在缓存中任意地方的。因此上面所介绍的方法还要依据缓存的约束进行调整。

习 题 10

10.1 严格按照括号指定的次序,计算表达式((u+v)+(w+x))+(y+z),也就是不利用交换律或结合律来改变加法的计算次序。像例 10.3 那样给出提供最大可能并行的寄存器级的机器代码。

10.2 为下面的表达式完成习题 10.1 的要求。

(a) (u+(v+(w+x)))+(y+z)

(b) (u+(v+w))+(x+(y+z))

若用最小化寄存器数来取代最大化并行,将需要多少步计算?用最大化并行节省了几步?

10.3 习题 10.1 的表达式可以由下面的指令序列来执行。如果所需要的并行都能得到,那

么这些指令的执行需要几步？

```
LD  R1,u             // R1 = u
LD  R2,v             // R2 = v
ADD  R1,R1,R2        // R1 = R1+R2
LD  R2,w             // R2 = w
LD  R3,x             // R3 = x
ADD  R2,R2,R3        // R2 = R2+R3
ADD  R1,R1,R2        // R1 = R1+R2
LD  R2,y             // R2 = y
LD  R3,z             // R3 = z
ADD  R2,R2,R3        // R2 = R2+R3
ADD  R1,R1,R2        // R1 = R1+R2
```

10.4　把例 10.4 讨论的代码片段翻译成采用 CMOVZ 条件转移指令(10.2.6 节)的形式。在你的代码中数据相关出现在什么地方？

10.5　画出图 10.26 各代码片段的数据依赖图。

```
LD R1,a            LD R1,a            LD R1,a
LD R2,b            LD R2,b            LD R2,b
SUB R3,R1,R2       SUB R1,R1,R2       SUB R3,R1,R2
ADD R2,R1,R2       ADD R2,R1,R2       ADD R4,R1,R2
ST a,R3            ST a,R1            ST a,R3
ST b,R2            ST b,R2            ST b,R4
   (a)                (b)                (c)
```

图 10.26　习题 10.5 的机器代码

10.6　假定机器有一个算术运算部件 ALU(用于 ADD 和 SUB 操作)和一个内存资源 MEM(用于 LD 和 ST 操作)。再假定,除了 LD 操作需要 2 个周期外,所有操作都只需要 1 个周期。像例 10.6 那样,在 LD 操作一个内存单元启动 1 周期后,操作同一个内存单元的 ST 可以启动。请为图 10.26 的各代码片段寻找最短调度。

10.7　根据下面的资源情况,重做习题 10.6。

(a) 机器有一个 ALU 资源和两个 MEM 资源。

(b) 机器有两个 ALU 资源和一个 MEM 资源。

(c) 机器有两个 ALU 资源和两个 MEM 资源。

10.8　假定机器模型和习题 10.6 的一样。

(a) 为下面的代码片段画出数据依赖图。

```
LD  R1,a
ST  b,R1
```

```
LD R2,c
ST c,R1
LD R1,d
ST d,R2
ST a,R1
```

(b) 在(a)答案的图中,关键路径是哪几条?

10.9 考虑下面的代码片段

```
if(x==0)a=b;
else a=c;
d=a;
```

假定机器使用例10.6的延迟模型,还假定机器一次可以同时执行任意两条指令。找出该程序片段可能的最短执行。

10.10 下面是一个循环体的代码

```
L:LD R1,a(R9)
  ST b(R9),R1
  LD R2,c(R9)
  ADD R3,R1,R2
  ST c(R9),R3
  SUB R4,R1,R2
  ST b(R9),R4
  BL R9,L
```

a(R9)这样的地址表示内存单元,其中a是常量,R9是记录循环迭代次数的寄存器。假定每次迭代访问不同的内存单元,因为R9有不同的值。使用例10.10的机器模型,按下面方式调度该循环。

(a) 让每次迭代尽可能紧凑(即在每个算术操作后只引入一个NOP操作),循环展开两次。在不违反机器限制的前提下,尽早启动第二次迭代。

(b) 重复(a)的工作,但循环展开三次。同样,在不违反机器限制的前提下,尽早启动每次迭代。

*10.11 使用例10.10的机器模型,为循环

```
for(i=1;i<n;i++) {
    A[i]=B[i-1]+1;
    B[i]=A[i-1]+2;
}
```

找出最小启动间隔和对此间隔的一种调度。注意,迭代的计数通过寄存器的自动减1来完成,无须单独完成此计数的操作。

* 10.12 证明，在每个操作对每种资源仅需要一个部件的特殊情况下，算法 10.2 总能够找到满足下界的软件流水调度。

10.13 图 10.25 基于块的矩阵乘法算法没有把矩阵 Z 初始化为零，请加上把 Z 初始化为零的步骤。

第 11 章

编译系统和运行时系统

通常,除了编译器外,还需要一些其他工具的帮助,才能得到可执行的目标程序,这些工具包括预处理器、汇编器和连接器等。对于 FORTRAN 和 C 等语言来说,这些工具都较简单和明显。了解这些工具有助于掌握从源程序到可执行目标程序的实际处理过程,这些知识对于参与大型软件系统的开发是很有用的。本章介绍 C 语言编译系统。

此外,目标代码运行时,还需要一些工具的支撑,如动态连接程序、无用单元收集程序等,这些工具的集合称为运行时系统。本章还介绍 Java 语言的运行时系统及其无用单元收集程序。

11.1 C 语言的编译系统

除了编译器外,还需要一些其他的工具来建立一个可执行的目标程序。本节以 GNU C 编译系统(简称 GCC 系统)为例,说明编程语言编译系统的一般工作过程。

一个 C 源程序可以分成若干个模块,存储在不同的文件中。C 编译系统对这些源文件分别进行预处理、编译和汇编,形成可重定位的目标文件;然后再利用连接器将这些目标文件和必要的库文件连接成一个可执行的目标文件,即具有绝对地址的机器代码。这一过程可用图 11.1 描绘。

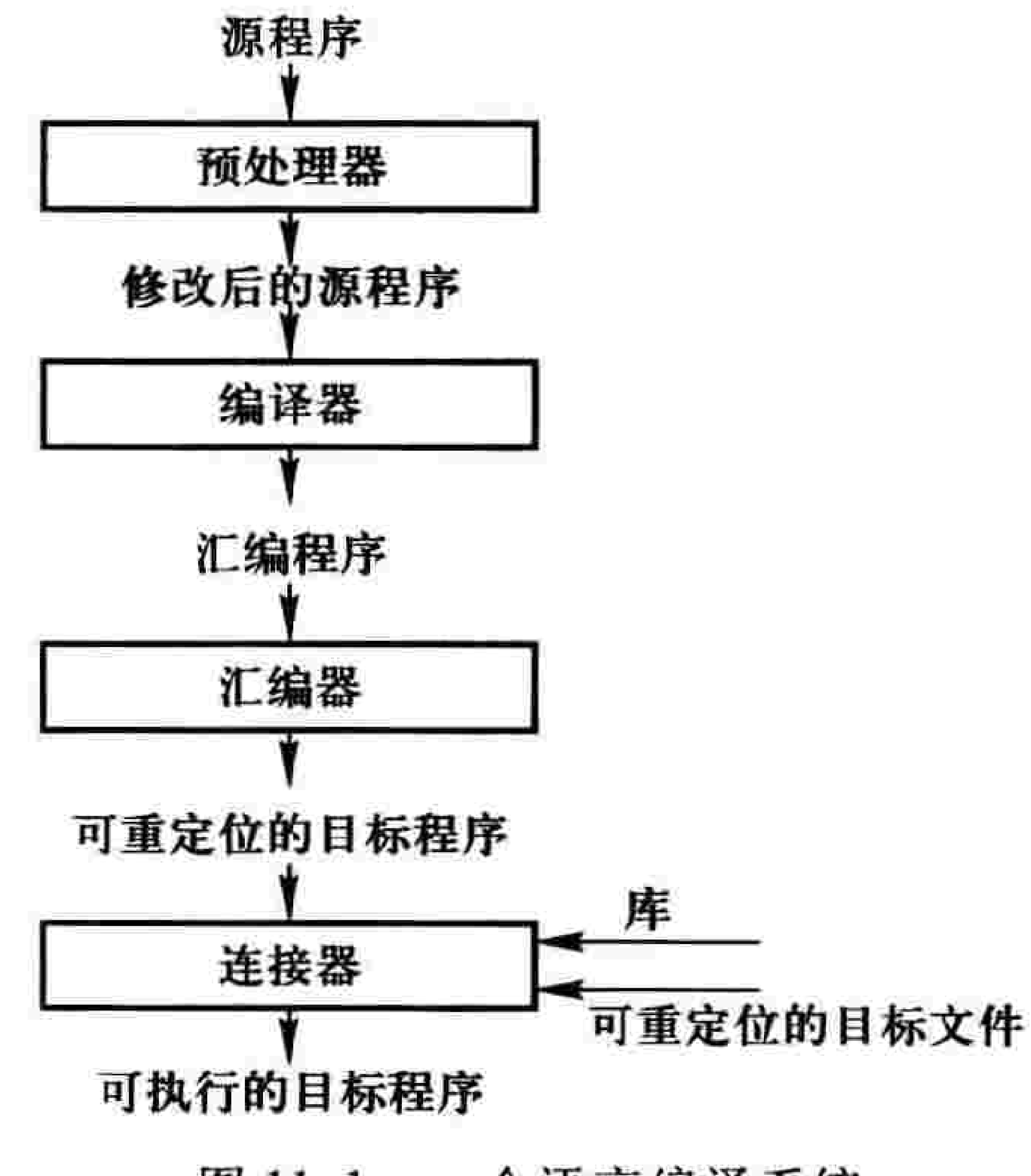


图 11.1　一个语言编译系统

大多数编译系统提供一个**驱动程序**来调用语言的预处理器、编译器、汇编器、连接器,以支持用户完成从源程序到可执行程序的翻译。在 GCC 系统中,驱动程序的名字是 gcc(或 cc)。

下面结合一个 C 语言的程序实例来讨论 GCC 系统的工作步骤。图 11.2 中的程序由两个文件 main. c 和 swap. c组成,为便于引用中间的语句,增加了行号。在 UNIX(还有 Linux)环境下,输入如下命令

可以得到该程序的可执行文件 swap：

gcc-v-o swap main. c swap. c

这里，使用选项-v 可以输出该编译系统各步骤执行的命令和执行结果，选项-o 后紧跟的字符串指示生成的可执行文件的名字。

main. c
```
(1) #if 1
(2) int buf[2];
(3) #else
(4) int buf[2] = {10,20};
(5) #endif
(6) void swap();
(7) #define A buf[0]
(8) int main()
(9) {
(10) scanf("%d, %d", buf, buf+1);
(11) swap();
(12) printf("%d, %d",A, buf[1]);
(13) return 0;
(14) }
```

swap. c
```
(1) extern int buf[2];
(2) int *bufp0 = buf;
(3) int *bufp1;
(4) void swap()
(5) {
(6) int temp;
(7) bufp1 = buf+1;
(8) temp = *bufp0;
(9) *bufp0 = *bufp1;
(10) *bufp1 = temp;
(11) }
```

图 11.2 main. c 和 swap. c 组成的程序

11.1.1 预处理器

GCC 首先调用**预处理器** cpp，将源程序文件翻译成一个 ASCII 中间文件，它是经修改后的源程序。图 11.3 是执行“cpp main. c main. i”后，main. c 被翻译为中间文件 main. i。注意，生成的中间文件也随 GCC 版本的不同而可能有所不同。

预处理器产生编译器的输入，它实现以下功能。

(1) 文件包含。预处理器可以把源程序文件中的包含声明(#include)展开为程序正文。例如，当源程序文件中含有语句

#include <stdio. h>

时，预处理器会在系统标准路径下搜索 stdio. h，再用文件 stdio. h 中的内容来代替这个语句。

```
(1) # 1 "main. c"
(2)
(3) int buf[2];
(4)
(5)
(6)
(7) void swap();
(8)
(9) int main()
(10) {
(11) scanf("%d, %d", buf, buf+1);
(12) swap();
(13) printf("%d, %d",buf[0], buf[1]);
(14) return 0;
(15) }
```

图 11.3 main. i 的内容

(2) 宏展开。C 程序可以使用#define 来定义**宏**,一个宏定义给出一段 C 代码的缩写。预处理器将源程序文件中出现的对宏的引用展开成相应的宏定义,这一过程称为**宏展开**。例如,main.c 的第(7)行为宏 A 的定义,第(12)行中的 A 是对该宏的引用。在预处理后产生的 main.i 中,宏 A 的定义转换成一个空行,对宏 A 的引用则展开成 buf[0]。

(3) 条件编译。预处理器根据#if 和#ifdef 等编译命令及其后的条件,将源程序中的某部分包含进来或排除在外。通常把排除在外的语句转换成空行。

显然,实现一个这样的预处理器并不困难。

有些语言的预处理器用于增强老的语言,使之包含现代的控制结构和数据类型。从增强语言到老语言的翻译由这样的预处理器完成。

11.1.2 汇编器

GCC 系统的编译器 cc1 负责把 C 程序文件编译成汇编代码,执行“cc1 main.i main.c -o main.s”后,main.i 被编译成 ASCII 汇编文件 main.s,见图 11.4。这些汇编代码由**汇编器**进一步处理。

最简单的汇编器对输入进行两遍扫描。在第一遍,汇编器扫描输入,将表示存储单元的所有标识符都存入符号表,并分配地址。在第二遍,汇编器再次扫描输入,把每个操作码翻译成机器语言中代表那个操作的位串,并把代表存储单元的每个标识符翻译成符号表中为这个标识符分配的地址。

倘若一个汇编代码文件中有对外部符号的引用,如果汇编器生成的是可重定位的目标文件,那么汇编器的工作将变得略微复杂。在 11.1.4 节,当知道了目标文件的格式后,要实现一个汇编器并不是一件困难的事情。另外,一遍扫描完成汇编代码到可重定位目标代码的翻译也是完全可能的。

在 GCC 编译系统中,要想得到源程序被编译成的汇编代码,只要加编译选项-S 就可以。例如,用

```
gcc -S main.c
```

可以得到汇编文件 main.s。使用汇编器 as

```
as -o main.o main.s
```

可以将 main.s 汇编成可重定位目标文件 main.o。

11.1.3 连接器

从第 8 章已经知道,汇编器或编译器输出的机器代码称为目标模块或目标文件,它有两种形式。

(1) 可重定位的目标文件。它包含二进制代码和数据,可以和其他可重定位目标文件组装成一个可执行的目标文件。

```
	.file"main.c"
	.version"01.01"
gcc2_compiled.:
.section.rodata
.LC0:
	.string"%d,%d"             // scanf 和 printf 中使用的格式串
.text
	.align 4                   // 按 4 字节对齐
.globl main
	.type main,@function       // 本模块定义的函数 main
main:
	pushl %ebp                 // 将老的基地址指针压栈
	movl %esp,%ebp             // 将当前栈顶指针作为基地址指针
	pushl $buf+4               // 实参 buf+1 入栈
	pushl $buf                 // 实参 buf 入栈
	pushl $.LC0                // 格式串指针入栈
	call scanf                 // 调用函数 scanf
	addl $12,%esp              // 栈顶指针恢复到参数压栈前的位置
	call swap                  // 调用函数 swap
	movl buf+4,%eax            // 取实参 buf[1]到寄存器
	pushl %eax                 // 实参 buf[1]入栈
	movl buf,%eax              // 取实参 buf[0]到寄存器
	pushl %eax                 // 实参 buf[0]入栈
	pushl $.LC0                // 格式串指针入栈
	call printf                // 调用函数 printf
	addl $12,%esp              // 栈顶指针恢复到参数压栈前的位置
	xorl %eax,%eax
	jmp .L1
	.p2align 4,,7
.L1:
	leave
	ret
.Lfe1:
	.size main,.Lfe1-main
	.comm buf,8,4              // 本模块定义的未初始化全局变量 buf
	                           // 占 8 字节,按 4 字节对齐
	.ident "GCC:(GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

图 11.4 汇编文件 main.s

(2) 可执行的目标文件。它包含二进制代码和数据,可以直接被复制到内存并被执行。

实际上另外还有一种形式,即共享目标文件。它是一种特殊的可重定位目标文件,可以在装入程序或运行程序时,动态地把共享目标文件装入内存并将它和程序连接。

从技术角度看,一个**目标模块**是一个字节序列,而一个**目标文件**则是一个以文件形式存储在外部存储器上的目标模块。不过,本书将不加区分地使用这两个术语。

连接是一个收集、组织程序所需的不同代码和数据的过程(它们可能在不同的目标模块中),以便程序能被装入内存并执行。连接可以在将源代码翻译成机器代码的编译阶段完成,也可以在程序装入内存并执行的装入时完成,甚至可以在程序运行时完成。

静态连接器负责将多个可重定位目标文件组织成一个可执行目标文件(也可以组织成一个可重定位目标文件);**动态连接器**则支持在内存中的可执行程序在执行时与共享目标文件进行动态的连接。有些系统将装入可执行程序时与共享目标文件进行的连接也称为动态连接。

如果这些目标文件是以有用的方式组织在一起的,那么它们之间就会出现一些外部引用,即一个文件中的代码引用另一文件中的存储单元。这种引用可以分为两种情况,一种是在一个文件中定义数据单元,而在另一个文件中使用它;另一种情况是函数入口点出现在一个文件,而调用点出现在另一个文件中。

在连接器 ld 的上下文中,一个重定位模块 M 定义和引用的符号通常分成以下三类。

(1) 全局符号。指那些在模块 M 中定义,可以被其他模块引用的符号。它包括模块 M 中定义的非 static 属性的函数和全局变量。

(2) 局部符号。指那些在模块 M 中定义,且只能在本模块中引用的符号。它包括模块 M 中定义的有 static 属性的函数和全局变量。

(3) 外部符号。指那些由模块 M 引用并由其他模块定义的符号。

这样,连接器主要完成以下两个任务。

(1) 符号解析(symbol resolution)。连接器识别各个目标模块中定义和引用的符号,为每一个符号引用确定它所关联的一个同名符号的定义。

(2) 重定位。编译器和汇编器产生的代码节和数据节分别都是从零地址开始。连接器按如下方式来重定位这两节:将每一个符号定义关联到一个内存位置,然后修改所有对这些符号的引用,以使它们指向相关联的内存位置。

在理解了 11.1.4 节中的目标文件的格式后,实现连接器并不是一件困难的事情。

11.1.4 目标文件的格式

目标文件的格式随系统不同而不同。来自 Bell 实验室的第一个 UNIX 系统使用 a.out 格式,至今 UNIX 下的可执行文件默认名仍为 a.out。System V UNIX 的早期版本使用 COFF(Common Object File Format)格式。Windows NT 使用 COFF 的一个变体,称为 PE(Portable Executable)格式。现代 UNIX 系统,如 Linux、System V UNIX 的后期版本、BSD UNIX 变体和 Sun Solaris,都使用

UNIX 的 ELF(Executable and Linkable Format)格式。这里仅讨论 UNIX 使用的 ELF 文件格式。

图 11.5 为典型的 ELF 可重定位目标文件格式。ELF **头**(header)开始于一个 16 字节的序列,它描述了字的大小和产生此文件的系统的字节次序。ELF 头的其余部分包含的信息用于连接器分析和解释目标文件,其中包括 ELF 头的大小、目标文件的类型(可重定位、可执行或共享等)、机器类型(如 IA32)、节头表(section header table)在本目标文件中的偏移、节头表中条目的大小和数量。

节头表描述目标文件中各节的位置和大小。在 ELF 头和节头表之间是节本身。典型的 ELF 可重定位目标文件包含以下各节。

图 11.5 ELF 可重定位文件格式

(1) . text:被编译程序的机器代码。

(2) . rodata:诸如 printf 语句中的格式串和 switch 语句的跳转表(见 7.4.4 节)等只读数据。

(3) . data:已初始化的全局变量,如 swap. c 中的 bufp0。

(4) . bss:未初始化的全局变量,如 main. c 中的 buf 及 swap. c 中的 bufp1。该节在目标文件中不占实际的空间,只是一个占位符。目标文件格式区分已初始化和未初始化变量是为了提高空间的利用率:在目标文件中,未初始化变量不必占用实际外部存储器的任何空间(历史上,bss 是 better save space 的缩写)。在有些汇编语言中,已经用. comm 代替了. bss。

(5) . symtab:记录在该模块中定义和引用的函数和全局变量信息的符号表。和编译器内部的符号表不同的是,. symtab 中不包含局部变量。

例 11.1 下面是用 readelf 工具显示的 main. o 的符号表的后 5 个条目。没有显示的前 9 个条目是供连接器内部使用的局部符号。注意,符号表的内容也随 GCC 版本的不同而略有不同。

Num:	Value	Size	Type	Bind	Ot	Ndx	Name
9:	0	66	FUNC	GLOBAL	0	1	main
10:	4	8	OBJECT	GLOBAL	0	COM	buf
11:	0	0	NOTYPE	GLOBAL	0	UND	scanf
12:	0	0	NOTYPE	GLOBAL	0	UND	swap
13:	0	0	NOTYPE	GLOBAL	0	UND	printf

在符号表中,Value 域表示符号的地址:对于可重定位目标文件,它是相对于定义该对象的节的起始处的偏移;对于可执行目标文件,它是绝对的运行地址。Size 域表示该对象的字节数。Type 域表示符号的类型,一般是数据对象(OBJECT)或函数(FUNC)。Bind 域表明符号是局部的(LOCAL)、全局的(GLOBAL)还是外部的(EXTERN)。Ndx 域指示与该符号关联的节,它的值是节头中的索引号。除了节头表中记录的节外,还有 ABS、UND 和 COM 三个特殊的**伪节**。其中,ABS 指不需要重定位的符号;UND 指被该模块引用但没有在该模块中定义的符号;COM 是未初始化的数据对象。对于 COM 符号,Value 域表示值的对齐方式,Size 域表示其最小的字节数。

在该例中,第9个符号main是位于.text节(Ndx=1)、偏移为0的66字节的函数;第10个符号buf是位于.bss节、按4字节对齐的8字节对象;随后是引用的3个外部符号scanf、swap和printf。 □

(6).rel.text:.text节中需要修改的单元的位置列表。当连接器将该目标文件和其他目标文件连接时需要这些信息,它包括任何调用外部函数或引用全局变量的指令。

(7).rel.data:用于被本模块引用或定义的全局变量的重定位信息。通常,任何要初始化的全局变量,若它的初值为某全局变量或外部函数的地址,则它的值需要修改。

(8).debug:用于调试程序的调试符号表,它包含在源程序文件中定义的局部变量和类型定义、在源程序文件中定义和引用的全局变量,以及最初的源文件等条目。只有用-g选项调用gcc才会出现此节。

(9).line:源程序文件和.text节中的机器指令之间的行号映射。该节仅在用-g选项调用gcc时才会出现。

(10).strtab:一组有空结束符的串构成的串表。它用于保存.symtab节和.debug节的符号表中的名字和节头表中节的名字。

11.1.5 符号解析

连接器需要将每个符号引用正确地与来自输入的可重定位模块的符号表中的一个符号定义相关联,从而确定各个符号引用的位置。在编译时,当遇到当前源文件没有定义的符号时,它假定该符号已在其他某个模块中定义,并为该符号产生一条符号表条目(如main.o符号表中的swap符号),把它留给连接器处理。如果连接器在所有输入模块中都找不到被引用符号的定义,则报告错误消息并结束连接。

解析全局符号的引用还是有点棘手,因为同一个符号可能被多个目标模块定义。此时,连接器必须报告一个错误,或者按某种方式选择其中一个定义而放弃其余的定义。UNIX系统中采用的方法涉及编译器、汇编器和连接器之间的合作,它会给粗心的程序员引入一些莫名其妙的错误。

在编译时,编译器向汇编器输出的全局符号区分为强和弱两种,汇编器隐式地将这些信息编码在可重定位目标文件的符号表中。函数和已初始化的全局变量为强符号,未初始化的全局变量为弱符号。对于图11.2的程序,main、bufp0和swap为强符号,buf和bufp1为弱符号。UNIX连接器使用如下规则处理多重定义的符号。

规则1 不允许有多重的强符号定义。

规则2 出现一个强符号定义和多个弱符号定义时,选择强符号的定义。

规则3 出现多个弱符号定义时,选择任意一个弱符号的定义。

比如,若将图11.2的swap.c的第1行“extern int buf[2];”改为“int buf[2]={11,21};”并存入swap1.c,再将main.c的第1行“#if 1”改为“#if 0”并存入main1.c,那么试图编译和连接

main1. c 和 swap1. c 时,连接器将报告错误信息,因为强符号 buf 被多次定义(规则 1)。

若试图将 swap1. c 和原先的 main. c 进行编译和连接生成可执行文件时,由于 main. c 中定义的 buf 为弱符号,这时连接器将隐含地选择在 swap1. c 中定义的强符号 buf,从而顺利地生成可执行文件(规则 2)。

再将 swap. c 的第 1 行中的 extern 去掉并存入 swap2. c 中,然后编译和连接 main. c 和 swap2. c,这时也能生成可执行文件(规则 3)。

不过,若将 swap2. c 中 buf 的类型由 int 改为 double 并存入 swap3. c,再编译和连接 main. c 和 swap3. c 时,虽能生成可执行文件(规则 3),但会因为多重定义的两个弱符号的类型不相容而给出警告信息。并且,由于这个类型不相容,生成的目标程序执行时,buf 数组元素的值和上一步得到的目标程序执行时的值不一样。

规则 2 和规则 3 的应用可能会产生一些难以理解的不会被捕获的错误,尤其是在多重定义的符号具有不同类型的时候。为避免这样的问题,程序员可以用适当的编译选项(如--warn-common 选项)调用连接器,以获得解析多重全局符号定义的警告信息。

11.1.6 静态库

到目前为止,一直假设连接器读入一组可重定位目标文件,将它们连接并输出一个可执行文件。事实上,所有的编译系统都提供一种机制,将相关的可重定位目标模块打包成一个文件,称为**静态库**。静态库可以作为连接器的输入被多次使用。当建立可执行文件时,连接器仅复制库中被应用程序引用的模块。

考虑 ANSI C,它定义了一个范围极广的标准 I/O、串操作和整型数学函数的集合,如 printf、strcpy、atoi 等。它们被编译打包在 libc. a 库中,可以供所有的 C 程序使用。ANSI C 还定义了浮点型的数学函数集,如 libm. a 库中的 sin 和 cos。现在思考编译器开发者不用静态库将这些函数提供给用户的几种不同方法。

一种方法是编译器认可对标准函数的调用并直接产生相关的代码。Pascal 语言使用这种方法,为程序员提供了少量的标准函数。对于 C 语言,这种方法是不可行的,因为 C 语言的标准定义了大量的标准函数,这样做会显著增加编译器的复杂性。当每次增加、删除或修改标准函数时,都需要发行新的编译器版本。不过对应用程序员来说,这种方法相当方便,因为这些标准函数总是可用的。

另一种方法是将 C 语言的所有标准函数放在一个可重定位目标文件中,如 libc. o,应用程序员可以将它连接并生成自己的可执行文件。这种方法的好处是标准函数的实现不再是编译器实现的一部分,而对程序员来说仍然相当方便。不过,一个严重的缺点是系统中每一个可执行文件都可能包含全部标准函数集的副本,这将极大地浪费外部存储器空间。更糟的是,每一个正在运行的程序在内存中都可能包含这些函数的副本,这将极大地消耗内存资源。另一个严重的缺点是,任何对标准函数的修改,不论是多么小的修改,都需要库开发者重新编译整个库的源文件,这

种耗时的操作将使标准函数的开发和维护变得复杂。

可以通过为每个标准函数创建一个可重定位文件,并按照大家熟知的目录存储它们来部分地解决这些问题。可是这种方法要求应用程序员显式地将有关的目标模块连接到他们的可执行文件中,这一过程很容易出错并且耗时。

静态库可以用来解决上述各种方法存在的问题。可以把相关的函数分成若干源文件,分别编译,然后把生成的目标模块打包成一个静态库文件。应用程序可以通过在命令行中指定该库文件名来使用库中定义的任何函数。

在连接时,连接器将只复制被程序引用的目标模块,这就减少了可执行程序在外部存储器和内存中的大小。另一方面,应用程序员的编译命令只需要把用到的库文件的名字包含进来就可以了。事实上,GCC 会自动地将 libc. a 等库传递给连接器,不需要应用程序员显式地指明。不过在直接用 ld 连接目标模块时,必须显式地给出被引用的所有库文件,否则连接器不能正确地完成连接。

在 UNIX 系统中,静态库是以一种特殊的文件格式,即档案文件(archive),存储在外部存储器中。一个档案文件是一个带档案头的被连接的目标文件的集合,档案头描述了每一成员目标文件的大小和位置。档案文件用. a 后缀表示。例如,可以用两个命令将 swap. c 编译并打包成一个自己的库 mylib. a:

```
gcc -c swap.c
ar rcs mylib.a swap.o
```

其中编译选项-c 表示生成目标文件,不进行连接。

这时可以编译 main. c 并将它和 mylib. a 连接,建立可执行程序 swap1:

```
gcc -static -o swap1 main.c /usr/lib/libc.a mylib.a
```

其中,-static 参数要求连接器建立一个完全连接的可执行目标文件,它可以直接被装入内存运行,不需要任何进一步的连接。

图 11.6 总结了带有-static 参数的连接器的活动。当连接器运行时,它确定由 swap. o 定义的 swap 符号被 main. o 引用,因此它将 swap. o 复制到可执行文件 swap1 中。而 mylib. a 中的其他成员模块(如果有的话),由于其定义的符号没有被引用,则不被连接器复制到可执行文件中。连

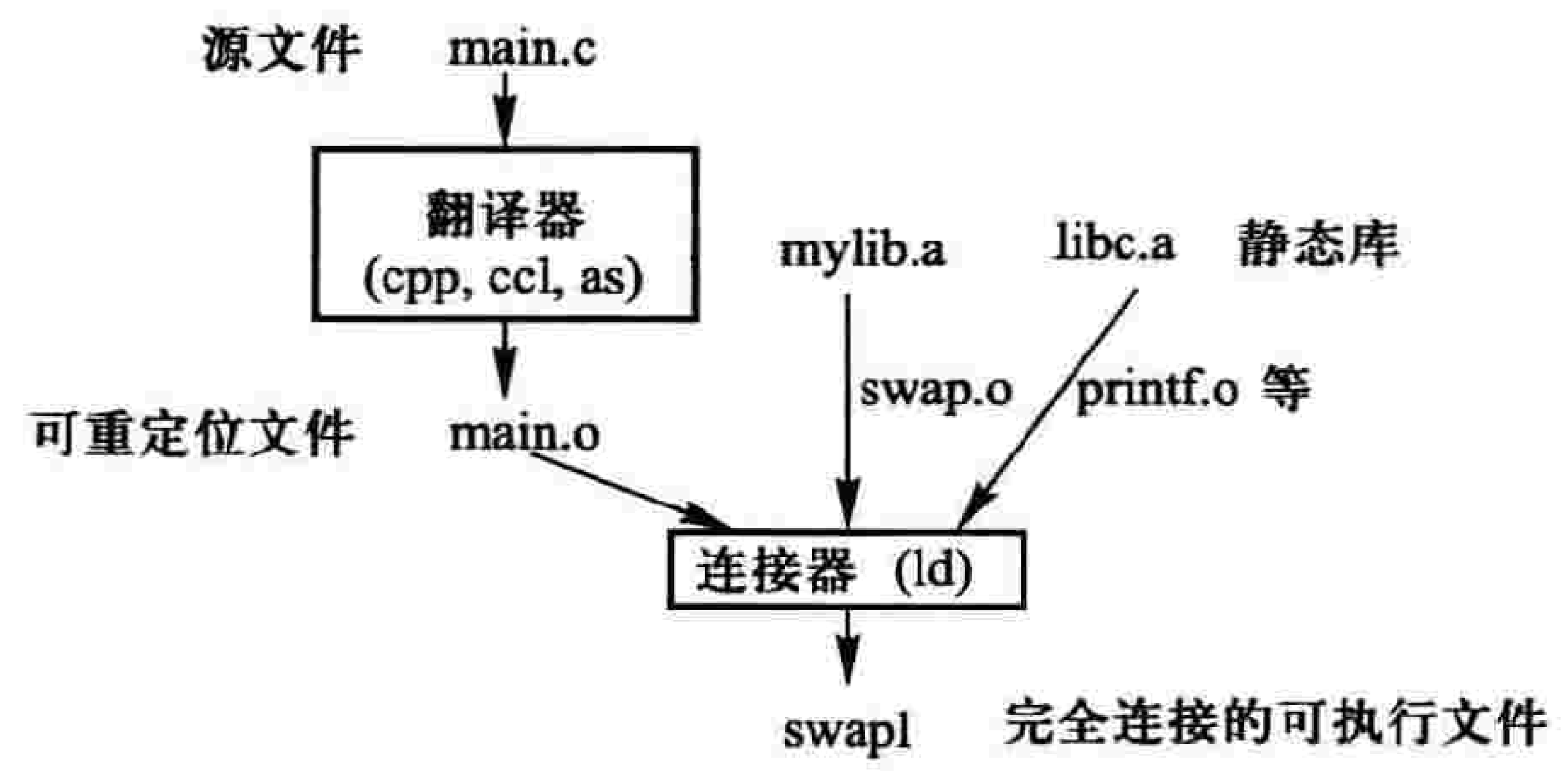


图 11.6 和静态库连接

接器同样复制来自 libc. a 的 printf. o 模块、scanf. o 模块和一些来自 C 运行时系统的其他模块。

静态库一方面是一种有用的基本工具,另一方面它们也让程序员感到混乱。因为,UNIX 连接器使用静态库来解析外部引用,在符号解析阶段,连接器按照可重定位的目标文件和档案文件在 gcc 或 ld 命令行上出现的次序从左到右扫描它们。命令行中档案文件和目标文件的次序相当重要。如果定义一个符号的档案文件在命令行中出现在引用该符号的目标文件之前,则这种引用可能不被解析,导致连接将失败。

使用档案文件的一般规则是将它们放在命令行的最后。如果不同档案文件的成员是相互独立的,则由于没有一个档案文件的成员引用其他档案文件的成员定义的符号,这些档案文件可以按任何次序放在命令行的最后。如果这些档案文件之间相互不独立,则它们必须按一定的次序出现在命令行中,使得对每个被一个档案文件中的成员引用的外部符号 s,至少有定义 s 的一个档案文件在命令行中出现在含有对 s 的引用的档案文件之后。

11.1.7 可执行目标文件及装入

图 11.7 概括了典型 ELF 可执行文件中的信息种类。它与可重定位目标文件格式类似。ELF 头描述文件的整体格式,它包含程序的入口点,即当程序运行时要执行的第一条指令的地址。对. text、. rodata 和. data 节,除了已经被重定位成最后运行时的内存地址以外,它们与可重定位目标文件中对应的节类似。. init 节定义一个称为_init 的小函数,它由程序的初始化代码调用。由于可执行程序已被完全连接,因此它不需要与可重定位文件的. rel. text、. rel. data 类似的可重定位节。

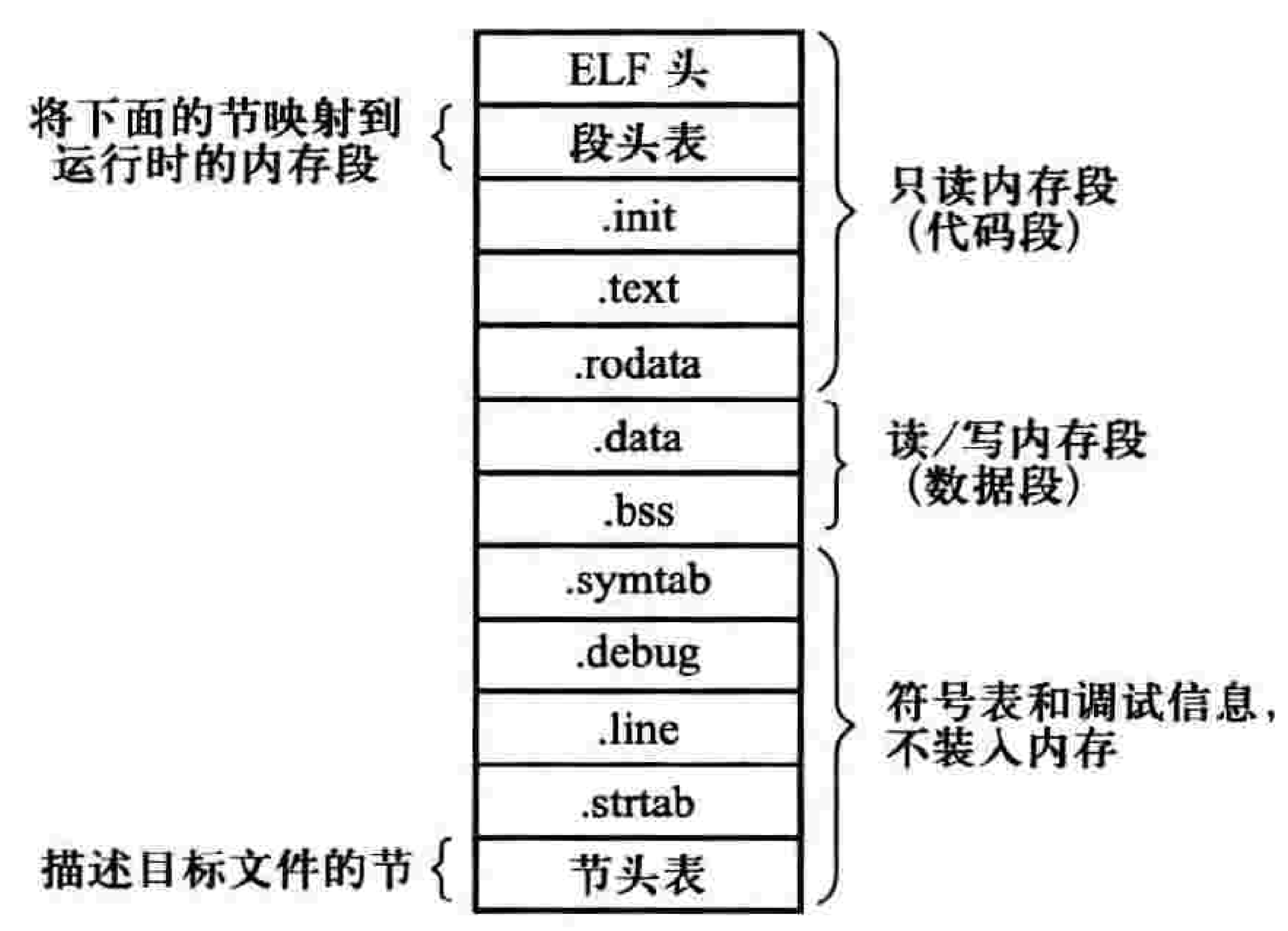


图 11.7 典型的 ELF 可执行目标文件

ELF 可执行文件被设计成易于装入内存,可执行文件中相邻的块被映射到相邻的内存段(segment)中。这种映射关系在段头表中描述。

为运行一个可执行程序 swap,可以在 UNIX shell 的命令行上输入它的名字:

./swap

由于 swap 不是内部的 shell 命令,shell 假定 swap 是一个可执行目标文件,它通过调用叫做**装载器**的驻留在内存中的操作系统代码来运行 swap。任何 UNIX 程序都可以通过调用 execve 函数来调用装载器。装载器将可执行目标文件从外部存储器复制到内存,然后通过跳转到程序的入口点来运行程序。复制程序到内存中并运行程序的过程被称为**装入**。

每一个 UNIX 程序都有一个和图 11.8 类似的运行时内存映像。在 Linux 系统中,代码段一般总是从 0x08048000 地址开始。数据段从下一个按 4 KB 对齐的地址开始。运行时的堆跟在可读写段后,从第一个按 4 KB 对齐的地址开始,并通过调用 malloc 库函数向上增长。从 0x40000000 地址处开始的段被保留用于共享库。用户栈总是从 0xbfffffff 地址开始并向较低的内存地址增长。位于栈之上,从 0xc0000000 地址开始的段被保留用于操作系统驻留内存部分的代码和数据,即操作系统内核。

当装载器运行时,它创建如图 11.8 的内存映像。在可执行目标文件的段头表的指导下,装载器将它的块复制到内存中的代码段和数据段。接着,装载器跳转到程序的入口点,它就是_start符号的地址。_start 地址处的启动代码定义在目标文件 crt1.o 中,它对于所有的 C 程序都一样。_start 完成用户程序运行前的准备,它调用.text 和.init 节中的初始化例程等后,调用应用程序的 main 函数,开始执行用户的 C 代码。

注意,这里描述的装入过程从概念上来说是正确的,只有在理解了进程、虚拟内存和内存分页等概念后,才能真正了解装入过程是如何工作的。

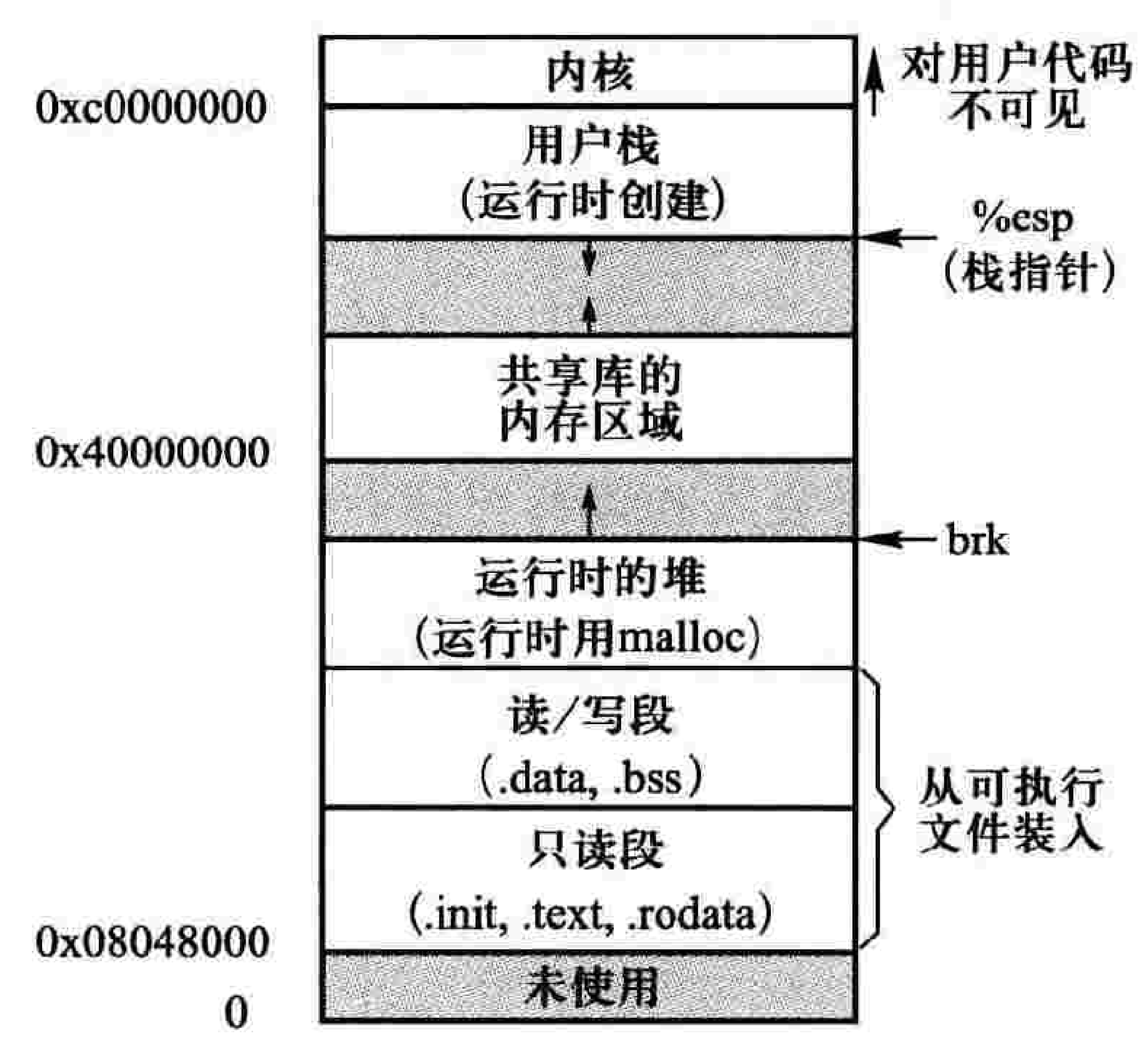


图 11.8 Linux 运行时的内存映像

11.1.8 动态连接

与所有软件一样,静态库需要周期性地维护和更新。如果应用程序员想要使用库的最新版本,他们必须设法知道库已经改变,然后显式地将他们的程序和被更新的库重新连接。另一个问题是,几乎每一个 C 程序都使用如 printf 和 scanf 这样的标准 I/O 函数。在运行时,这些函数的代码被复制在每一个正在运行的进程的 text 段中。在一个运行有 50 ~ 100 个进程的典型系统上,这会显著地浪费内存系统资源。

共享库是弥补静态库的缺点所进行的一场变革。一个共享库,也称共享目标文件,它在运行时可以装到任意的内存位置并和内存中的程序连接。这一过程称为**动态连接**,它由动态连接器执行。在 UNIX 系统上,共享库一般以.so 为后缀。Microsoft 操作系统称共享库为动态连接库,一般以.dll 为后缀。

共享库以两种不同的方式被共享。第一,在任何给定的文件系统中,对每个库,正好只存在一个.so文件,该.so文件中的代码和数据被所有引用该库的可执行目标文件所共享,这与静态库的内容被复制和嵌入到引用它的可执行目标文件中的方式相反。第二,运行时,共享库的.text节在内存中的一个副本可以被正在运行的不同进程共享。

图11.9总结了图11.2程序的动态连接过程,这里首先将源文件swap.c编译、连接成自己的共享库mylib.so:

```
gcc -shared -fPIC -o mylib.so swap.c
```

-fPIC选项指示编译器产生位置无关的代码(指无须连接器的修改而能装入到内存任意地址执行的代码),-shared选项指示连接器创建一个共享的目标文件。

一旦创建了mylib.so共享库,就可以将它和main.c连接成可执行程序swap2:

```
gcc -o swap2 main.c ./mylib.so
```

动态连接的基本思想是在可执行文件被创建时静态地做一部分连接准备工作,然后在程序被装入时动态地完成连接过程。需要注意的是,mylib.so中的代码节和数据节并没有被复制到可执行目标文件swap2中。连接器只是复制一些重定位信息和符号表信息,它们用以在运行时解析对mylib.so中代码和数据的引用。

当装载器装入和运行可执行程序swap2时,它使用11.1.7节讨论的技术,装入已被部分连接的可执行目标文件swap2。swap2包含有.interp节,该节包含动态连接器的路径名,如Linux系统下的ld-linux.so,它本身是个共享对象。装载器通常装入和运行动态连接器,而不是将控制直接传给应用程序。

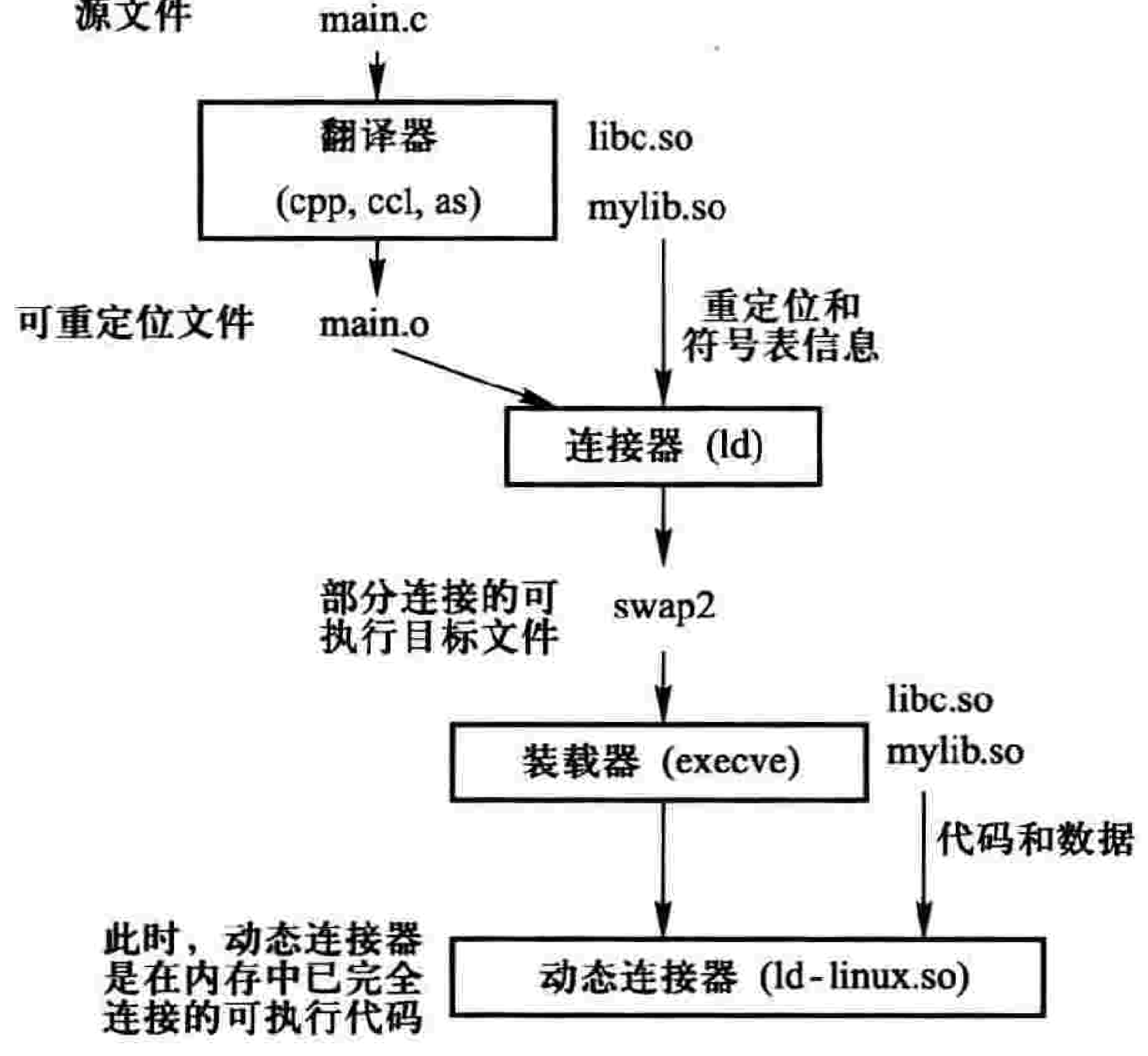


图11.9 和动态库连接

动态连接器接着完成连接任务：

(1) 把 libc.so 的文本和数据装入内存并进行重定位，在 IA32/Linux 系统上，共享库被装载在起始地址为 0x40000000 的地方（见图 11.8）。

(2) 把 mylib.so 的文本和数据装入内存并进行重定位。

(3) 重定位 swap2 中任何对 libc.so 或 mylib.so 定义的符号的引用。

最后，动态连接器将控制传递给应用程序。此后，共享库的位置被确定，并在该程序的执行期间不再改变。

上面介绍了共享库的一种装入方式，即在应用程序被装入时，动态连接器装入和连接共享库。还有另外一种方式，即应用程序在运行过程中通过显式的函数调用来请求动态连接器装入和连接共享库。两种方式的装入和连接过程本质上没有什么区别，这里不再介绍。

11.1.9 处理目标文件的一些工具

在 UNIX 系统中，有很多工具可以用来帮助你理解和处理目标文件。尤其是，GNU 的 binutils 包特别有用，可以运行在每一种 UNIX 平台上。下面是一些工具的名称和功能简介。

- ar：创建静态库，插入、删除、罗列和提取成员。
- strings：列出包含在目标文件中的所有可打印串。
- strip：从一个目标文件中删除符号表信息。
- nm：列出一个目标文件的符号表中定义的符号。
- size：列出目标文件中各段的名字和大小。
- readelf：显示目标文件的完整结构，包括编码在 ELF 头中的所有信息。它包括了 size 和 nm 的功能。
- objdump：所有的二进制工具之母，可以显示目标文件中的所有信息。其最有用的功能是反汇编.text 节中的二进制指令。

UNIX 系统还提供 ldd 程序用来处理共享库：

- ldd：列出可执行目标文件在运行时需要的共享库。

11.2 Java 语言的运行时系统

一般的高级语言程序如果要在不同的平台上运行，至少需要编译成不同的目标代码。随着互联网的流行，不同类型的计算机通过网络共享数据和其他计算资源，人们希望同一版本的程序能够在多种不同的平台上运行。Java 语言正是顺应这样的潮流而诞生的一种跨平台的编程语言。

Java 虚拟机技术则是实现 Java 平台无关性特点的关键。Java 源程序被编译成与任何计算机体系结构都无关的中间代码——Java 虚拟机语言（简称 JVML），任何机器只要安装了 Java 运行

时系统就能够运行这种中间代码。Java 虚拟机是一种抽象的机器,在实际的计算机上通过软件模拟来实现,Java 运行时系统就是 Java 虚拟机的一个实现。本书将不加区分地使用运行时系统和虚拟机这两个概念。Java 虚拟机有自己的指令系统,也有自己的存储器区域。它屏蔽了不同操作系统和机器设备的区别,在运行的 JVML 程序和底层的硬件与操作系统之间建立了一个缓冲区。JVML 程序只需要与虚拟机交互,不需要关心底层的硬件和操作系统。

用虚拟机来实现一种程序设计语言的思想并不是 Java 首创的,在 Java 出现之前,UCSD Pascal系统就已在一种商业产品 P-Code 机器中应用了这一思想。

11.2.1 Java 虚拟机语言简介

Java 虚拟机用以支持 JVML 这种中间语言,因此需要对 JVML 的一些概念和术语进行简要介绍,以更好地理解虚拟机。

Java 程序首先由 Java 编译器把它编译成字节码,也就是 JVML 程序,并被放置到后缀名为“.class”的文件中。每个 class 文件是一个 8 位字节流,它包含一个 Java 类或接口的信息。由一张符号表和各方法的字节码序列以及其他辅助信息组成。下面介绍 class 文件中最重要的几个部分。

1. 常量池(constant pool)

常量池包含了在该 class 文件结构及其子结构中引用的各种类型(字符串、浮点等)的常量,常量池的功能类似于传统编程语言中的符号表,但是它比通常的符号表包含更多的信息。

2. 类成员信息

一个 Java 类的成员信息放在两个长度可变的表中:域信息表和方法信息表。

3. JVML 指令序列

一条 JVML 指令由一个字节的操作码以及若干个供该操作使用的操作数构成。Java 虚拟机上同样也有运行数据区,每个线程都有一个运行栈,该栈保存局部变量和计算的中间结果,并参与方法的调用和返回。所有线程共享的堆用来动态分配对象。JVML 指令描述的就是在这样的抽象机上进行的操作。图 11.10 中展示了一个 JVML 程序的例子。

11.2.2 Java 虚拟机

Java 虚拟机一般由以下几个部分构成:类装载器(字节码验证器)、解释器或/和编译器,还有包括无用单元收集器(garbage collector)和线程控制模块在内的运行时支持系统,另外还有一些标准类和应用接口的 class 文件库。其组成情况如图 11.11 所示。

首先运行 Java 编译器把 Java 应用程序编译成字节码程序。字节码程序可以在网络上传输,在另一台机器的 Java 虚拟机上运行。Java 虚拟机执行字节码的过程可以分为三步:代码的装入、代码的验证和代码的执行。

```
Java 源程序中的方法:
int calculate (int i){
    int j=2;
    return ((i+j) * (j-1));
}
对应的字节码程序:
int calculate (int i)
iconst_2          // 在 Java 栈中压入常数 2
istore_2          // 将栈顶常数存放到局部变量 2(局部变量 j)中
iload_1           // 将局部变量 1(参数 i)压入栈顶
iload_2           // 将局部变量 2(局部变量 j)压入栈顶
iadd              // 从栈中弹出两整数并相加,结果压栈
iload_2           // 将局部变量 2(局部变量 j)压入栈顶
iconst_1          // 在 Java 栈中压入常数 1
isub              // 从栈中弹出两整数并相减,结果压栈
imul              // 从栈中弹出两整数并相乘,结果压栈
ireturn           // 将栈顶元素返回
```

图 11.10 JVML 程序示例

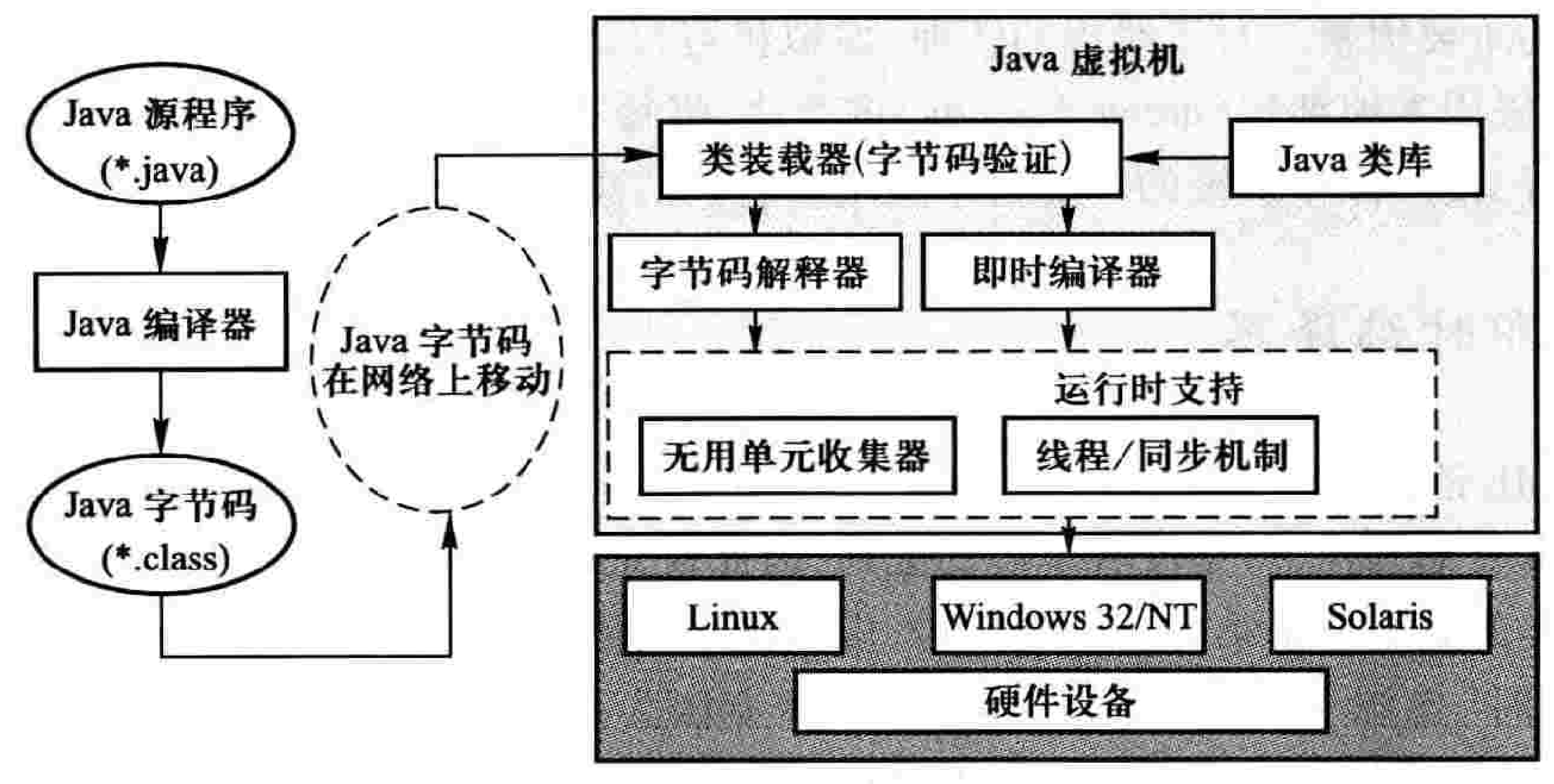


图 11.11 Java 的运行环境

代码的装入由类装载器完成,类装载器负责装入程序运行时需要的所有代码,其中包括程序代码中用到的所有类。随后,被装入的代码由字节码验证器进行安全性检查,以确保代码不违反 Java 的安全性规则,字节码验证器还可以发现操作数栈溢出、非法数据类型转换等多种错误,通过验证之后,代码就可以提交运行了。

Java 字节码的运行有两种方式:解释执行方式和即时编译(just-in-time compilation)方式。解

释执行的缺点是太过简单而且速度很慢,早期的SUN JDK解释器要比类似的C++代码慢5~30倍。即时编译方式是由即时编译器先将字节码编译成本地机器代码之后再执行。即时编译的方法能够产生质量较高、执行速度较快的代码,但需要花费额外的编译时间。需求驱动是即时编译的另一特点,一个方法直到被调用时才将字节码翻译成机器代码,当一个方法被调用两次以上时,机器代码的执行效率便足以补偿编译耗费的时间。

Java虚拟机还需要给字节码的执行提供其他运行时的支持。无用单元收集器用来管理所有线程共享的运行时的数据空间。在Java虚拟机的逻辑组件中,运行时数据空间包括Java栈、堆、方法区(method area)、常量池。其中Java栈是每个Java虚拟机线程私有的,与线程同时创建并同时结束。而堆、方法区、常量池则是所有线程共享的。堆是从中分配所有类实例和数组的运行时的数据区。Java对象从不被显式地回收,无用单元收集器自动地将程序中不再用到的单元回收。方法区类似于传统语言的代码存储区,如UNIX进程中的.text段,它存储每个类的公用数据和方法代码。方法区逻辑上是堆的一部分,也由无用单元收集器来管理,但是一些简单的实现可以选择不回收它。常量池是各个class文件中常量池的运行时的表示,其内容包括从编译时已知的数值和文字到必须在运行时解析的方法和域引用。常量池是方法区的一部分。11.3节将介绍无用单元收集技术。

Java语言支持多线程,对于多线程的应用而言,线程调度和同步支持是多线程协同工作正确性的重要保证。在典型的商务应用中,同步操作占了相当大的部分,因此高效地实现同步也是高性能虚拟机的重要因素。对于线程的管理,虚拟机可以选择由自己来对程序中的多个线程进行调度,也可以采用本地绑定(native-binding)的方法,将每一个Java线程都映射为实际运行的操作系统上的线程,使用操作系统的调度器来实现对它们的高效支持。

11.2.3 即时编译器

由于JVML语言平台无关的特点,JVML程序的装载、连接、编译过程与传统编程语言的编译过程相比更具有动态性,但是没有本质的区别,因此仅介绍即时编译器。

当一个类的某个方法第一次被调用时,虚拟机才激活即时编译器将它编译成机器代码,编译器被称为"即时"也源于此。即时编译器以一个方法为单位进行编译,能够生成较高质量的代码,它生成的代码的执行速度可以达到解释执行的10倍。

即时编译器的出现使得Java程序的执行效率得到了很大提高,但是执行过程不得不等待编译结束,因此使得执行时间变长。在传统的静态编译中,编译时间可以忽略不计,因为经过一次编译得到的可执行文件,可以被多次执行。而对于即时编译器来说却并非如此,即时编译器在运行时编译字节码,编译时间是运行时间的一部分。为了缓解执行效率和编译开销之间的矛盾,很多虚拟机都会使用快速解释器和优化编译器的组合或者是简单编译器和复杂编译器的组合。下面用一个具体的实现来举例说明即时编译器是如何动态地、按需地编译一个方法的。

Intel开发的开放式运行时平台(Open Runtime Platform,ORP)是一个研究动态编译和垃圾收

集技术的开放资源研究性平台。它的即时编译器的动态性不仅表现在直到方法第一次调用才进行编译,而且还体现在它能够动态评估不同的代码而采用不同的编译策略。

从图 11.12 可以看出,当一个类的方法表刚创建的时候,方法的代码指针并不是指向实际的方法代码,而是一段 compile-me 代码。顾名思义,compile-me 代码的作用就是编译自己。所以当方法第一次被调用时,实际执行的是调用即时编译器来编译该方法的字节码。编译器完成工作之后,compile-me 代码段将把方法表中的代码指针更新指向编译得到的本地代码,并且执行第一次调用。方法再次被调用时,执行的就是本地代码了。

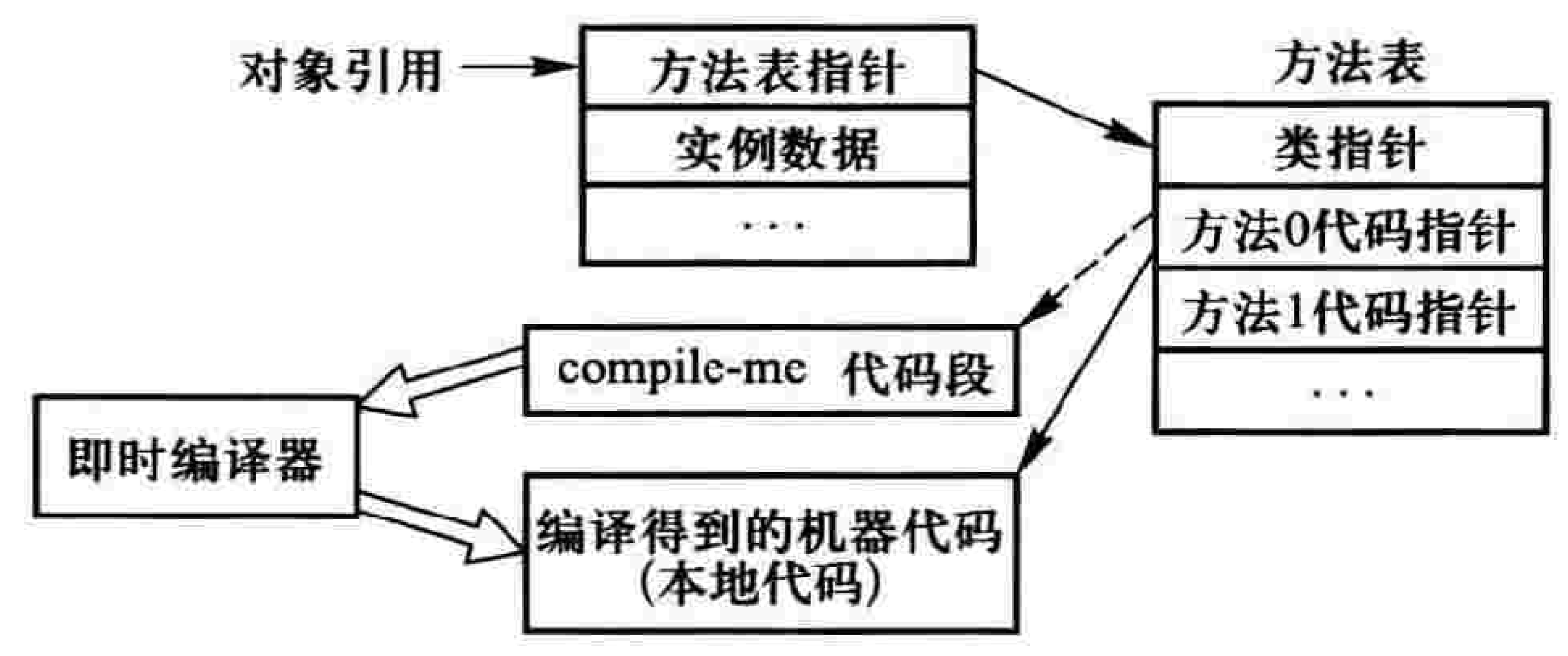


图 11.12 即时编译方法

ORP 的编译器还实现了一个动态的重编译机制。这个重编译机制的关键在于对不同的代码自适应、有选择地使用不同的编译方案:对于那些执行频率低的"冷"代码,采用快速而较粗糙的编译器;而对执行频率高的"热"代码,则花更多时间对其做细致的编译。这个重编译机制的自适应性表现在它能够收集运行时的信息,判断代码是否为热点,而及时地调整编译策略。

图 11.13 展示了 ORP 的编译结构。这个编译结构有三个重要的组成部分:快速生成代码的编译器、优化编译器、统计信息。

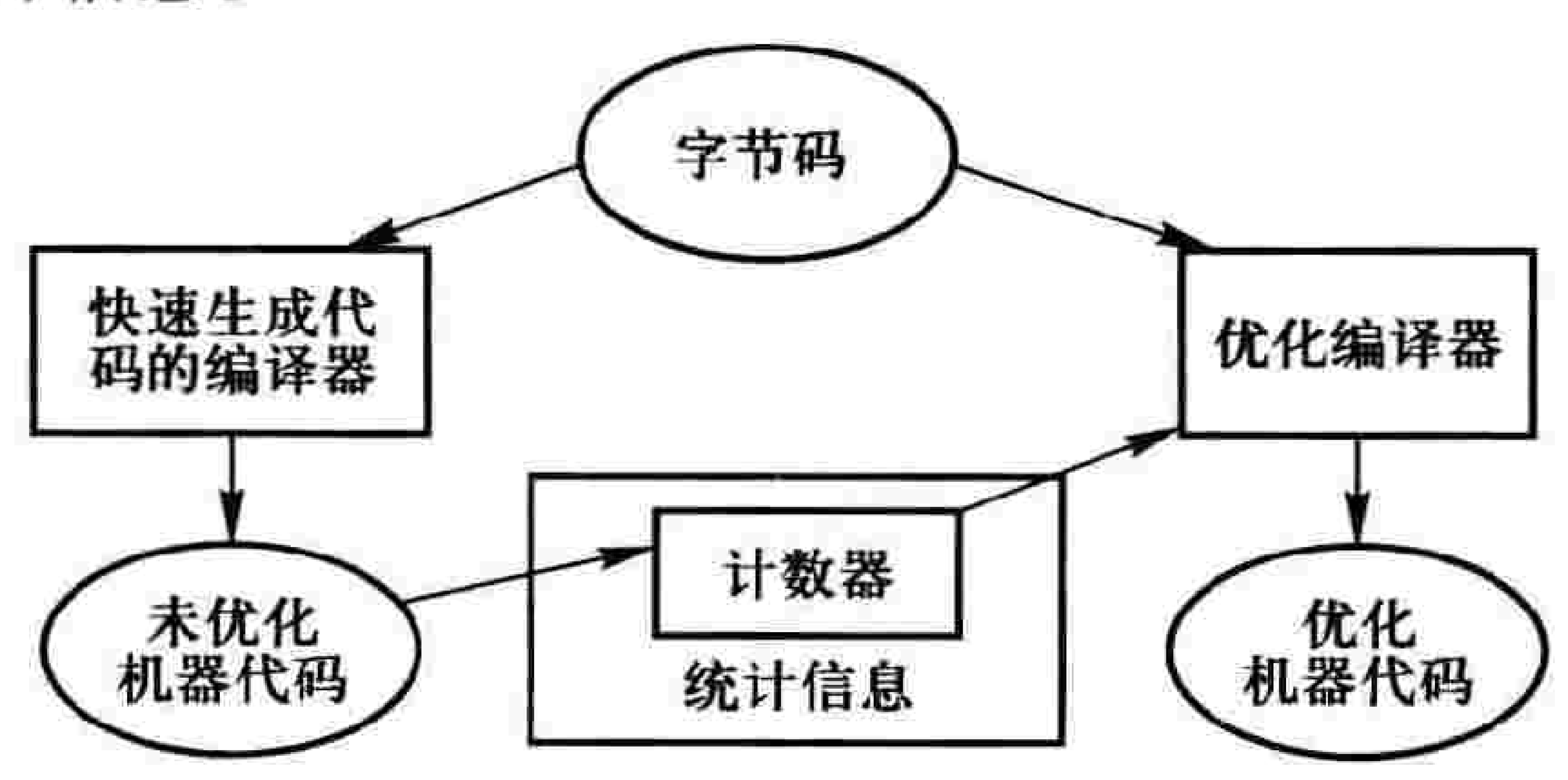


图 11.13 ORP 的重编译机制

快速生成代码的编译器,在 ORP 中通常也被称为 O1 编译器。所有的方法在第一次被调用时都用这个编译器编译成机器代码。ORP 快速编译器不但能达到比较理想的编译速度,而且能够对机器代码进行轻量级的优化,产生的机器代码的执行速度要比解释执行快得多。O1 编译器的目标是快速地产生机器代码,并且维持一个合理的代码质量,它能够在对字节码进行两次遍历后产生机器代码。

O1 编译器在机器代码中插入一些统计代码，用来收集统计信息，这些信息记录了方法或一段循环代码被调用的次数。这些统计信息包含进行重编译的触发条件。有了统计信息，就能够判断某段代码是否为“热”代码，当代码的调用次数达到某个阈值时，它将被优化编译器重新编译。优化编译器对方法重新编译时还将利用 O1 编译器收集的这些统计信息。

优化编译器也称 O3 编译器，它对代码进行更为细致的优化，产生质量较好的目标代码。之后，当方法再次被调用时，就会执行优化版本的机器代码。O3 编译器为了产生高质量的目标代码，需要花更多时间在优化上。它采用传统的编译方法，为字节码建立一种中间表示，基于中间表示进行全局优化。

ORP 就是这样采用简单编译器和复杂编译器的组合，构造了一种平衡编译时间和代码质量的动态编译机制。

* 11.3 无用单元收集

理论上讲，无用单元(garbage)就是那些在程序的继续运行过程中不会再被使用的数据单元。被无用单元占据的内存空间应该被回收，以便把它们分配给需要空间的变量使用，这样一个处理过程就称之为**无用单元收集**(garbage collection)，俗称**垃圾收集**。该过程不需要程序员的干预，它可以自动地执行对内存的这种管理。但它可能需要来自编译器、操作系统和硬件方面的支持，并且由运行时系统来决定何时和怎样执行无用单元收集。

无用单元收集器(以下简称收集器)需要根据数据的活跃性来判断哪些是无用单元。由于实际上并非总能判断出一个数据记录的值以后是否还需要，因此收集器所使用的活跃性分析采用稳妥的策略。该策略使得活跃性是通过**根集**(roots set)以及从根集开始的**可达性**来定义。按照这个策略所得到的可达记录并非都是真正活跃的，但是它们都被保留；另一方面，编译器被要求努力极小化可达记录中实际上并不活跃的记录的数目。这样，通常所指的无用单元是那些不可能从程序变量经指针链到达的堆分配记录。

函数式语言一般都用这种技术来回收内存空间。对于命令式语言 Java 来说，它与 C 和 C++ 语言不一样，对象和数组这样的数据都分配在堆上，在栈中有相应的指针指向它们，并且语言不向使用者提供释放空间的函数。这样，一旦这些指针在栈中被释放后，堆中相应的记录就很可能不可达，因此需要依靠收集器来回收它们。

本节简要介绍一些主要的无用单元收集方法，并且描述编译器和收集器之间的一些相互影响，包括编译器提供给收集器的支持和收集器提供给编译器的接口。

11.3.1 标记和清扫

标记和清扫收集方法首先标记堆上所有可达记录，然后回收未被标记的记录。

在进行无用单元收集时，全局可见的变量都被看作是活跃的，任何活跃着的过程的任何一个活动记录中的局部变量也都被看作是活跃的。这样，根集就包含了全局变量、活动记录栈中的局部变量和被活跃着的过程使用的寄存器。堆上活跃记录的集合也就是从根集开始的任何一条指针路径上的记录的集合，它们可以看成是一个有向图，其中记录是结点，指针是边，程序变量是根，因此任何图遍历算法，如深度优先算法，都可用于在这个图上标记所有的可达记录。这就是**标记**阶段。

任何未被标记的记录都是无用单元，应该被回收。回收过程称为**清扫**。从堆的首地址开始，逐个记录地考察整个堆，寻找未被标记的记录，把它们链接成一个空闲链表。并且，这一阶段同时清除被标记的记录的标记，以备下一轮收集。当空闲链表中的空间不足以支持程序的存储分配请求时，就启动下一轮收集。

如果用户程序需要很多不同大小的记录，一个简单的空闲链表对于分配函数来说可能显得效率不高。因为当分配一个大小为 n 字节的记录时，它可能需要沿着这个链表寻找，直至找到一个大小合适的空闲块。可以通过建立有若干个空闲链表的一个数组来解决这个问题，例如 freelist[i]就是所有大小为 2^i 的空闲块的链表。这样，如果程序要分配一个大小为 $n(2^{k-1} < n \leq 2^k-1)$ 的记录时，那就从 freelist[k]中取一块。如果该链表为空，那就从 freelist[$k+1$]中取一块，将其中一半分配给用户程序，剩余的一半链回到 freelist[k]链表中。如果空闲块大小小于 2^{k+i} 的所有链表均为空，而 freelist[$k+i$]链表不为空，则从 freelist[$k+i$]链表中取一块，将其中的 2^k 大小分配给用户程序，而将剩余部分分割成若干块，分别插入到 freelist[k] ~ freelist[$k+i-1$]链表中。如果这样的操作失败，那就调用收集器来补充空闲链表。

传统的标记和清扫方法通常有两大问题。

首先是**碎片**(fragmentation)问题，它可以分为外部碎片和内部碎片两个方面。外部碎片是指当要分配一个 n 字节大小的记录时，发现有很多小于 n 字节的空闲块存在，但就是没有合适的空闲块可分配给这个记录。内部碎片是指程序使用一个过大的记录而没有拆分它，导致没有被使用的内存处于该记录中。碎片的后果就是空闲块和活跃记录交织在一起，使得对大记录的分配非常困难。通过维护上面提到的一组空闲链表并合并相邻空闲块可以缓解这个问题，但问题并没有消失。

第二个问题涉及**引用局部性**(locality of reference)问题。既然记录都不会被移动，那么活跃记录在一次收集以后仍然在原位置，和空闲块相交织。然后，新记录使用这些空闲块，其结果是不同时期的记录交织在一起。这给引用局部性带来了消极的影响，因为这种交织有可能使得当前要使用的各个活跃记录被分散到很多的虚拟内存页中，这些页在内存中可能被频繁地换进换出。在有虚存或缓存的计算机系统中，良好的引用局部性是非常重要的。

引用局部性也是一类数据局部性，但是它的改进主要依赖于收集算法的改进。

11.3.2 引用计数

标记和清扫收集方法通过首先找出堆上所有可达记录来确定无用单元。也可以通过记住有多少指针指向每个记录来直接完成，这称为记录的**引用计数**，记录的引用计数存在该记录中。

在使用这项技术的系统中，编译器需要在每个出现指针存储的地方生成额外的指令，以调整一些引用计数器的值。比如记录 p 的引用存进 $x.f$ 中，那么 p 的引用计数值会加 1，而 $x.f$ 原来指向的记录的引用计数值会减 1。当一个记录的引用计数值为 0 的时候，就可以把该记录加入空闲链表，并且被回收记录本身的指针域都要一一检查，它们所指向的记录的引用计数值也都要减 1。

引用计数算法看起来非常简单，具有吸引力，但是除了碎片和引用局部性问题外，它还有两大问题：第一，它并不总是有效的；第二，很难提高效率。

(1) 引用计数的技术对于循环的数据结构会失效。如果一组记录中的指针形成了一个循环，那么，即使从根集已经不可能到达这些记录了，这些记录的引用计数也永远不可能减到零。而且事实上，这种循环在一般的程序中经常会产生。

(2) 引用计数的效率问题是它的代价。因为每当执行指针存储的时候，都需要执行额外的指令来调整一些引用计数器的值。

上述问题使得引用计数技术近年来已失去了吸引力。对于大部分高性能的通用系统来说，引用计数收集已经被跟踪型收集代替，这种收集器是在遍历可达记录图时，将活跃记录和无用单元区分开来。11.3.1 节的标记和清扫收集方法就是一种跟踪型收集方法。

但是引用计数方法本身还是有很多有意义的优点。例如，它回收迅速；另外，即使在大部分堆空间都被占据的时候，它的性能仍然不受影响，很多其他的收集器这时会需要更多的空间用于交换以提高效率；还有，它可以被直截了当地做成完全渐增的和实时的。将来也许还会发现这种技术的一些其他用途，也许在混合型的收集器中使用，也许通过特殊的硬件可以提高它的性能。但是，一般来说，引用计数方法不会作为传统的单处理器上一种主要的无用单元收集技术。

11.3.3 复制收集

这个算法跟 11.3.1 节中的标记和清除算法一样，它也遍历可达记录所组成的有向图，只不过它在遍历的同时进行清扫，并且这种清扫主要是**复制**活跃记录。

通常，这种方法将整个堆空间分成大小相等的两块，每块都是连续的区域，一块称 *from_space*，另一块称 *to_space*。在程序运行时，只有 *from_space* 是可用的。当运行程序请求内存时，就在 *from_space* 中向上线性地分配内存。当运行程序要求的内存分配超过了 *from_space* 中空闲区域的大小时，运行程序暂时被停下来，复制收集器被激活，用来回收空间。

复制收集器根据某种遍历算法将所有的活跃记录遍历一次，同时把它们从 *from_space* 复制

到 *to_space*，并且在 *to_space* 中，这些记录被紧缩到一边。一旦复制完成，*to_space* 和 *from_space* 的角色相互交换，然后程序继续运行。图 11.14 给出了收集前后的一个示例。

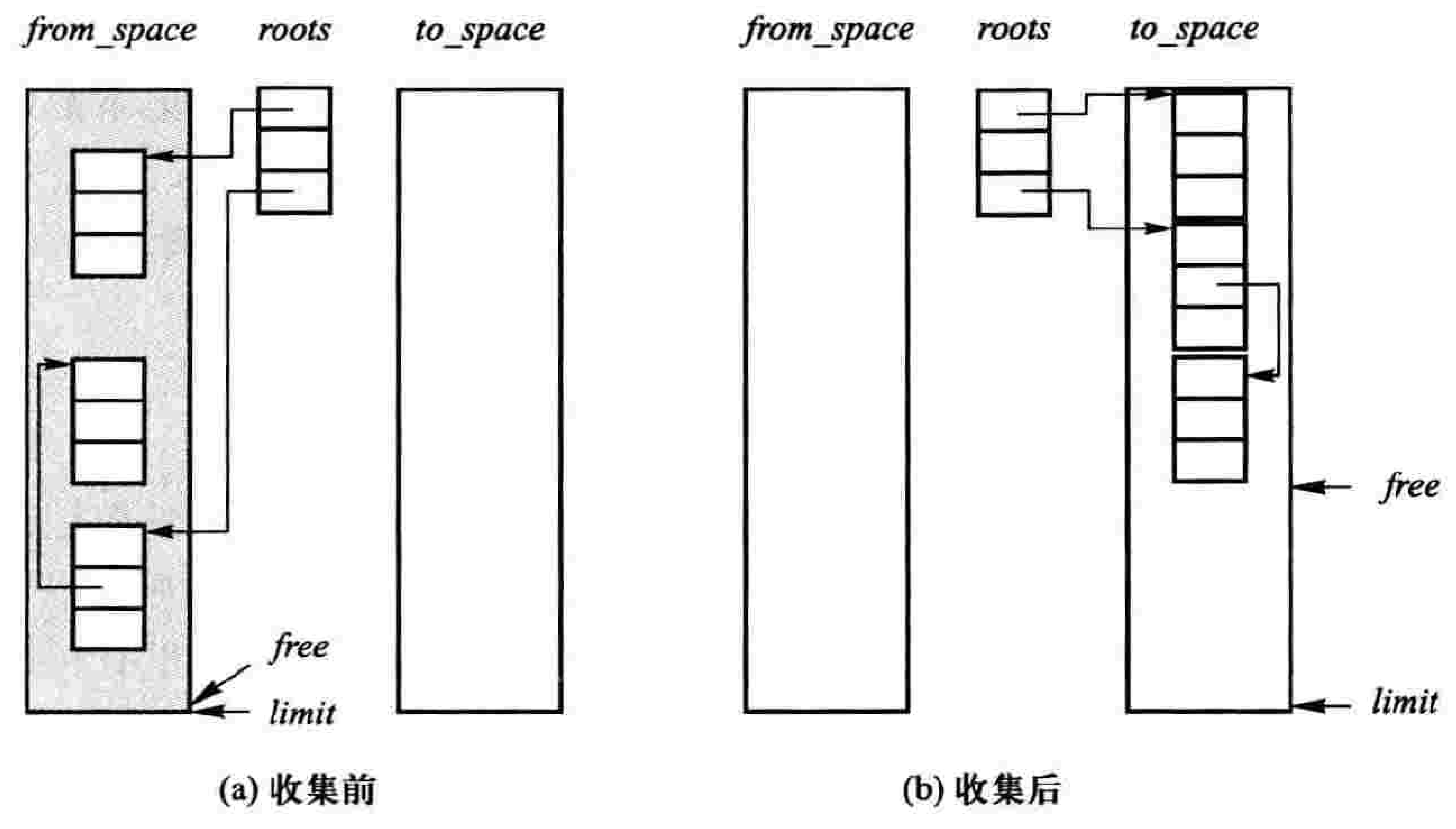


图 11.14 复制收集

对这个算法做稍微深入一点的讨论。该算法最基本的操作是指针的**转移**(forwarding)操作，就是把一个指向 *from_space* 的指针 *p*，修改成指向 *to_space* 中合适的位置，算法见图 11.15。有以下三种情况需要分别对待。

```
Forward(p);
    if (p 指向 from_space ) {
        if (p->f1 指向 to_space ) return p->f1;
        else {
            for(p 指向记录的每个域 fi) free->fi = p->fi;
            p->f1 = free;
            free = free+p 指向记录的大小;
            return p->f1;
        }
    } else return p;
```

图 11.15 转移指针的算法

(1) 如果指针 *p* 指向一个已经被复制到 *to_space* 的 *from_space* 记录，那么该 *from_space* 记录的第一个域，也就是 $p\text{->}f_1$ 是一个特殊的转移指针，它指示该 *from_space* 记录复制在什么地方。因此，现在只需要将它返回就可以了。

(2) 如果指针 *p* 指向的 *from_space* 记录还没有被复制，那么就将它复制到 *to_space* 中 *free* 所指位置，并把这个转移指针赋给 $p\text{->}f_1$，也就是让 $p\text{->}f_1$ 指向该记录在 *to_space* 中的位置。

(3) 如果 *p* 不是指针，或者如果 *p* 指向 *from_space* 之外的区域(如指向收集区域以外的地方，

或者指向 *to_space*)，那么对 p 的转移操作将什么也不做。

从理论上来说，只要有足够的内存，复制收集器可以有很高的效率。另外，它还可以将活跃数据紧缩在一起，碎片情况消失，引用局部性得到改善。当然，在实际使用中，这些好处会受到很多限制。首先这个算法需要的空间是实际需要空间的两倍。其次，复制大记录的代价太大。还有，如果遍历采用宽度优先搜索的话，引用局部性的改善不充分。但总的来说，复制算法的前景是很好的，它被广泛用作其他无用单元收集算法的基础，如分代算法、渐增式算法。

11.3.4 分代收集

分代收集的原理基于对这样一个现象的观察：在很多程序运行过程中，新建记录很可能很快死去，不会出现对它的复制；反过来，一个记录在几次收集后还可到达的话，那么很可能还会活跃到许多次收集后。因此收集器应该将精力集中到“年轻”的数据上，因为这里有相对高的无用单元比率。

堆将被分成**代**(generation)，最年轻的记录在 G_0 代，$G_i(i>0)$ 代中的记录比 G_{i-1} 代中的任何记录都要老。越年轻的代越被频繁地收集。

对 G_0 代进行收集时，就是从根集开始，使用复制方法或者使用标记和清扫方法。但是，这时的根集不仅仅是程序变量，它还包括 $G_1, G_2, \cdots$ 中指向 G_0 的指针。如果这样的指针太多的话，那么处理根的过程将比遍历 G_0 的可达记录的时间还要长。幸好，老记录指向年轻得多的记录的情况极少出现。通常是较新的记录指向老记录。

为了避免确定 G_0 的根集时在 $G_1, G_2, \cdots$ 中查找，需要编译器提供一些支持，让编译过的程序运行时能**记住**哪些地方有老记录到新记录的指针。这样的方法有以下几种。

(1) **记忆集合**。编译器产生指令，在每个执行形式为 $b\text{->}f=a$ 的修改存储单元后，将 b 放到被修改记录的一个向量(记忆集合)中。然后在收集时，收集器扫描记忆集合，寻找指进 G_0 的老指针 b。

(2) **卡标记**。该方法将存储区域划分为大小为 2^k 的逻辑“卡”，一个记录可以占据一块卡的一部分，或者从一块卡的中间开始，继续到下一块。每当地址 c 的内容被更新时，包含这个地址的卡就被标记。这里有一个字节数组用来做标记，字节索引可以通过将地址 c 右移 k 位获得。

(3) **页标记**。该方法类似于卡标记方法，如果 2^k 正好是页的大小，那么可以用操作系统的虚拟内存管理机制来标记页，而不需要编译器生成额外的指令。

记忆集合的优点在于精确，因为它包含的是被更新记录的地址，而卡标记方法的优点在于简单。对于每条更新指令的维护，记忆集合的方法可能需要 10 条额外的指令，而卡标记最少情况下只需要 2 条额外指令就够了，它所需要的开销远远小于记忆集合方法。如果将两种方法结合在一起使用，会带来更高的效率。

当收集开始时，记忆集合能够告知，老一代的哪些记录(或者卡、页)可能包含指向 G_0 的指针。这些指针是 G_0 根集的一部分。

通常，$G_i(i>0)$是按指数地大于G_{i-1}，比如G_0大小为0.5 MB，那么G_1大小应该是2 MB，G_2就应该是8 MB。假设G_0是当前要收集的代，它里面的活跃记录将复制到G_1中。经过几次对G_0的收集之后，G_1可能会聚集了相当数量的无用单元需要被收集。因为G_0可能包含了很多指向G_1的指针，所以最好将G_0和G_1一起收集。和前面一样，这时需要扫描记忆集合，以获得包含在G_2，G_3，… 中的那部分根。这样，各代都可能被收集，当然，越老的代收集频率越低。

11.3.5 渐增式收集

虽然收集时间的总和只占整个程序运行时间很小的比例，但是收集器仍然有可能偶尔将运行程序中断相对长的时间。例如分代算法提到G_0，G_1，…的大小呈指数变化，当对较老或最老的代进行收集时，可能就需要较多的时间。对于交互式程序和实时程序来说，这一点是难以接受的。而**渐增式收集器**或者**并发收集器**将程序运行和无用单元收集交错进行，避免了出现这种长时间的中断。

在渐增式算法中，只有当运行程序请求时，收集器才开始工作；在并行算法中，收集器可以在运行程序执行任何指令期间或之间进行工作。渐增式算法和并发算法面对的问题虽然类似，但是显然后者更加复杂和困难。

这些收集算法的困难在于，当收集器在做遍历以得到一个可达记录图时，运行程序并没有停止修改可达记录图。因此，收集算法必须有某种方法来跟踪可达记录图的变化。面对这些变化，也许需要重新计算可达记录图中的某些部分。已经有很多这方面的技术，在此不作介绍。这些技术都需要编译器生成额外的指令来提供支持，有些方法也可以利用虚拟内存的硬件来实现。

11.3.6 编译器与收集器之间的相互影响

在收集器的设计中，对收集器的底层有以下基本要求：

(1) 收集器必须能确定堆上分配的记录大小，这样它们才能被复制；

(2) 收集器必须能定位包含在堆记录里的指针，这样它们才能被跟踪和更新；

(3) 收集器必须能定位所有在全局变量中的指针；

(4) 收集器必须能在程序中任何一个可以进行收集的地方找到所有在活动记录栈中和寄存器中的指针；

(5) 收集器必须能找到所有由指针运算所产生的值指向的记录；

(6) 收集器必须能在记录被移动时，更新所有涉及的指针值。涉及的指针包括指向记录的指针，以及由指向记录的指针经计算而得到的指针。

这些要求都必须通过编译器的支持才能得到满足，因为很多所需信息只有在编译时能够获得。这是编译器必须对收集器提供支持的一些基本方面。上述6点中，对最后两点的理解需要阅读下面的导出指针部分。

概括地说,对于使用无用单元收集技术的语言,它的编译器与收集器之间的相互影响是:编译器生成(可能需要用收集器提供给它的接口函数)用来分配记录的代码;编译器为每个收集周期提供根集元素的存储位置描述;编译器向收集器提供堆上数据记录的布局描述等。

另外,对于渐增式收集和分代收集,编译器还必须生成一些额外的代码,如分代收集中所讲的建立记忆集合的指令。

1. 快速分配

某些编程语言和某些程序可以快速分配堆记录,同时也快速产生无用单元。对于函数式语言来说尤其如此,因为它不鼓励更新旧的数据。

可以想象,最快的分配记录和产生无用单元的速度是每遇到一条存储指令就分配一个字,这是因为一个堆分配记录的每个字通常都需要被初始化。经验数据表明,不管采用什么样的编程语言或者编写什么样的程序,每 7 条指令中就有一条是存储指令。因此,可以认为每条运行指令要进行 1/7 个字的分配。

假设收集的代价可以通过调整分代收集的代的数目变得很小,那么仍然需要可观的代价用于创建堆记录。若想最小化这样的代价,应该使用复制收集方法,因为它供分配的空间是连续空间,而不是空闲链表。若 *free* 指针指示下一个空闲位置,*limit* 指针指示空闲区域的末端(见图 11.14),要分配一个 *N* 字节大小的记录,一般是调用分配函数,下面列出其执行步骤:

(1) 分配函数的调用序列;

(2) 测试 $free+N < limit$?(如果失败,则调用收集器)

(3) $result=free$;

(4) 将 *free* 开始的 *N* 个字节都清空;

(5) $free=free+N$;

(6) 分配函数的返回序列:

　(a) 将 *result* 移到对后面的计算有用的某个地方;

　(b) 用一些有用的值将该记录初始化。

通过将分配函数在每个调用它的地方进行内联展开,步骤(1)和(6)可以删除。步骤(3)经常可以删除,将它合并到步骤(a)即可,步骤(4)也可以删除,用步骤(b)就可以了。步骤(a)和(b)之所以编号不同,是因为它们应该被看成是有用计算中的一部分,而不属于分配开销。

步骤(2)和(5)不能删除,但是如果在同一基本块中不止一个分配,则它们可以被多个分配操作共享。将 *free* 和 *limit* 放在寄存器中,步骤(2)和(5)总共用 3 条指令就可以完成。

通过这种合并技术,分配一个记录并且最终回收它的开销大约只需要 4 条指令就可以了。

2. 堆上数据布局的描述

收集器必须能够处理各种类型的记录,这些类型包括链表、树、程序声明的类型等。它必须能够知道每种类型的域的数目,以及其中哪些是指针域。

对于静态类型化的语言,如 Pascal 和 C,或者面向对象的语言,如 Java,确定堆记录最简单的方法就是让每个记录的第一个字指向一个特殊的类型或者类的描述符记录。该描述符记录会告

知该记录的大小以及每个指针域的位置。

这样,对于静态类型化语言,每个记录有一个字(描述符指针)的开销用于收集。但是对于面向对象的语言,每个对象本来就需要这样一个描述符指针以实现对方法的动态查找,因此每个对象不存在用于收集的额外开销。

类型或类的描述符必须由编译器的语义分析阶段所得的静态类型信息来生成。描述符指针将是运行时系统的 alloc 存储分配函数的参数。

除了描述堆上每个记录外,编译器还必须为收集器确定每个包含指针的临时变量和局部变量,不管它们是在一个寄存器中还是在一个活动记录中。因为每条指令都可能改变活跃临时变量的集合,因此**指针映像**(指活跃指针的集合)在程序的每个点都可能不同。于是,为简单起见,编译器一般只在收集工作可以启动的地方描述指针映像。这些地方就是 alloc 函数的调用点;还有,因为任何被函数调用可能会调用 alloc 的函数,因此在每个函数调用点必须描述指针映像。

在函数调用点的指针映像,检索它的最好键值是该调用点的返回地址,即一个处于地址 a 处的函数调用最好用紧跟在它后面的返回地址来索引,因为返回地址保存在活动记录中,它是收集器从活动记录栈中能看到的值。

在收集开始时,为了获得所有的根,收集器从栈顶开始,对所有的活动记录逐个地往下遍历栈。其中每个活动记录中保存的返回地址都是描述下一个活动记录的指针映像入口的关键字。在每个活动记录中,收集器从源于这个活动记录的指针去作标记(如果是标记和清扫方法的话)或者作转移(如果是复制收集的话)。

3. 导出指针

程序的指针有时可能指向一个堆记录的中间,或者指向这个记录的前面或后面。例如表达式 a[i-2000]经编译后被当作 M[a-2000+i]来计算(其中 M[a-2000+i]理解为取地址 a-2000+i 的内容):

$t_1 = a-2000$

$t_2 = t_1 + i$

$t_3 = M[t_2]$

如果表达式 a[i-2000]出现在一个循环内部,那么代码优化可能会将 $t_1 = a-2000$ 作为循环不变计算提升到该循环外面。如果该循环包括了一个 alloc 调用,并且一个收集发生时 t_1 还是活跃的,那么,收集器是否会感到困惑?因为指针 t_1 没有指向一个记录的开头,甚至出现更坏的情况,它指向一个不相关的记录。

t_1 被称为从基指针 a 导出的指针。指针映像必须能识别导出指针,并且知道每个导出指针的基指针是哪个指针。于是,当收集器将 a 重新分配到地址 a' 时,它必须把 t_1 调整到 $t_1 = t_1 + a' - a$。

当然,这意味着,只要 t_1 是活跃的,a 就必须保持活跃。考虑图 11.16 左边的一个函数体,其执行语句就是一个循环,该函数体的实现在图 11.16 的右边,假定数组分配在堆上。如果不存在 a 的其他使用,那么变量 a 在对 t_1 的赋值后不再活跃。但是和返回地址 L_2 相关的指针映像将不能适当地解释 t_1。于是,对于编译器的活跃变量分析来说,一个导出指针的活跃隐含地要求保持

它的基指针活跃。

int a[100] = {0};	r_1 = 100
int i;	r_2 = 0
for (i=2001; i <=2100; i++)	call alloc
f (a [i-2000]);	a = r_1
	t_1 = a-2000
	i = 2001
	L_1: r_1 = M[t_1+i]
	call f
	L_2: if i <= 2100 goto L_1

图 11.16 一个函数体及它的实现

习 题 11

11.1 如果 cfile 是一个 C 语言源程序(注意,该文件名没有后缀),在 x86/Linux 系统上,某版本的编译命令

```
cc cfile
```

的结果是错误信息

```
/usr/bin/ld: cfile: file format not recognized: treating as linker script
/usr/bin/ld: cfile: 1: parse error
collect2: ld returned 1 exit status
```

请解释为什么会有这样的错误信息。

11.2 C 语言程序(存储为一个文件)

```
long gcd(p,q) long p,q; {
    if (p%q==0)
        return q;
    else
        return gcd(q,p%q);
}
main() {
    printf("\n%ld\n",gcdx(4,12));
}
```

在 x86/Linux 系统上用某版本的 gcc 命令得到的编译结果如下:

```
In function 'main':
```

```
undefined reference to 'gcdx'
ld returned 1 exit status.
```

请问,这个 gcdx 没有定义,是在编译时发现的,还是在连接时发现的? 试说明理由。

11.3 一个 C 程序的三个文件的内容如下:

```
head.h:
short int a=10;

file1.c:
#include "head.h"
main() {
}

file2.c:
#include "head.h"
```

在 x86/Linux 系统上用某版本的命令:

```
cc file1.c file2.c
```

编译,其结果报错的主要信息如下:

```
multiple definition of 'a'
```

试分析为什么会报这样的错误。

11.4 一些 C 程序设计的教材上指出:"在需要使用标准 I/O 库中的函数时,应在程序前使用#include <stdio.h>预编译命令,但在用 printf 和 scanf 函数时,则可以不要。"(备注:有些编译器给出警告错误)事实上,并非仅限于这两个函数。例如下面的 C 程序编译后运行时输出字符 A 并换行,它并没有预编译命令#include <stdio.h>。试解释原因。

```
main() {
    putchar('A');
    putchar('\n');
}
```

11.5 把下面左边的文件 file1.c 提交给编译器,编译器没有报告任何错误。而把文件 file2.c 提交给编译器,错误报告如下:

```
file2.c: 2: error: conflicting types for 'func'
file2.c: 1: error: previous declaration of 'func'
```

试分析原因。(在这两个文件中,第 1 行都是函数 func 的原型,第 2 行都是函数 func 的定义,函数体为空)

file1.c	file2.c
int func(double);	int func(double);
int func(f) float f; {}	int func(float f) {}

11.6 C的一个源文件可以包含若干个函数,该源文件经编译可以生成一个目标文件;若干个目标文件可以构成一个函数库。如果一个用户程序引用库中的某个函数,那么,在连接时的做法是下面三种情况的哪一种,说明你的理由。

(a) 将该库函数的目标代码连到用户程序。

(b) 将该库函数的目标代码所在的目标文件连到用户程序。

(c) 将该函数库全部连到用户程序。

11.7 cc是UNIX系统上C语言编译命令,-l是连接库函数的选择项。某程序员自己编写了两个函数库libuser1.a和libuser2.a,当用命令

```
cc  test.c  -luser1.a  -luser2.a
```

编译时,报告有未定义的符号,而改用命令

```
cc  test.c  -luser2.a  -luser1.a
```

时,能得到可执行程序。试分析原因。(备注:库名中的lib在命令中省略。上面第一个命令和命令cc test.c libuser1.a libuser2.a的效果是一致的)

11.8 cc是UNIX系统上C语言编译命令,-l是连接库函数的选择项。两个程序员分别编写了函数库libuser1.a和libuser2.a。当用命令

```
cc  test.c  -luser1.a  -luser2.a
```

编译时,报告有重复定义的符号。而改用命令

```
cc  test.c  -luser2.a  -luser1.a
```

时,能得到可执行程序。试分析原因。

11.9 现将图11.2的swap.c编译成可重定位目标文件swap.o。对于在swap.o中定义或引用的符号,请说明它是否在swap.o的.symtab节中有相应的符号表条目?如果有,指出该符号被定义的模块(swap.o或main.o)、符号类型(local、global或extern),以及所在节(.text、.data或.bss)。

符号	在swap.o的.symtab节中有条目	符号类型	由哪模块定义	所在节
buf				
bufp0				
bufp1				
swap				
temp				

11.10 修改习题11.2中的程序，将main函数中的gcdx改为gcd，并将修改后的程序保存在gcd.c中。对该程序采用以下两种方式进行编译、连接：

```
gcc-o gcd1 gcd.c
gcc-static-o gcd2 gcd.c
```

所产生的可执行目标文件gcd1和gcd2的大小并不相同，前者约11 KB，后者却要接近1 MB。请分析产生这种不同的原因，它们在执行时存在什么样的差异。

11.11 a和b表示当前目录中的目标模块或静态库，a→b表示a依赖于b，即b定义了被a引用的符号。对于以下情况，给出最小的命令行（即包含最少数目的目标文件和库参数），从而使静态连接器能解析所有的符号引用。

（a）p.o→libx.a

（b）p.o→libx.a→liby.a

（c）p.o→libx.a→liby.a 并且 liby.a→libx.a→p.o

11.12 两个C语言文件link1.c和link2.c的内容分别如下：

```
int buf[1] = {100};
```

和

```
extern int * buf;
main() {
    printf("%d\n", * buf);
}
```

在x86/Linux系统上，经命令cc link1.c link2.c编译后，运行时产生如下的出错信息：

```
Segmentation fault
```

请说明原因。

11.13 两个C文件long.c和short.c的内容分别是

```
long i=32768 * 2;
```

和

```
extern short i;
main() {printf("%d\n",i);}
```

在x86/Linux系统上，用cc long.c short.c命令编译这两个文件，能否得到可执行目标程序？若能得到目标程序，运行时是否报错？若不报错，则运行结果输出的值是否为65536？若不等于65536，原因是什么？

11.14 下面左右两边分别是两个C程序文件file1.c和file2.c的内容，用命令cc file1.c file2.c对这两个文件进行编译和连接。请回答：

（a）编译器是否会报错？若你认为会，则说明理由。

（b）若编译器不报错，连接器是否会报错？若你认为会，则说明理由。

（c）若上面两步都不报错，则运行时是否会报错？若你认为会，则说明理由。

(d) 若上面三步都不报错,则运行输出的结果是什么?说明理由。

```
char k = 2;
char j = 1;
```

```
#include <stdio.h>
extern short k;
main() {
    printf("%d\n",k);
}
```

11.15 下面左右两边是一个C程序的两个源文件的内容:

```
#include <stdio.h>
char *p = "0123456789";
f() {
    printf("%s\n",p);
}
```

```
#include <stdio.h>
char p[10];
main() {
f(); p[1]='1';f();
}
```

运行目标程序所得到的结果如下(编译器是GCC:(GNU) 4.2.3 (Debian 4.2.3-5)):

```
0123456789
Segmentation fault
```

请描述该目标程序各数据区的内容,并回答为什么运行时会出现Segmentation fault。

11.16 下面三列是三个C程序(其中第三个程序由两个文件构成)。它们在编译、连接或运行时是否报错?若是,则指出编译器、连接器或运行时系统报告错误的原因。对于不报错的程序,除了告知输出结果外,还要对比有错的程序,说明为什么没有错误。

```
#include <stdio.h>
void f(m) long m[2]; {
  printf("%d,%d\n",
     m[0],m[1]);
}
typedef struct {
  long n[2];
} stype;
main() {
  stype k;
  k.n[0]=0; k.n[1]=1;
  f(k);
}
```

```
#include <stdio.h>
void f(m) long m[2]; {
printf("%d,%d\n",
  m[0],m[1]);
}
main() {
  long m[2];
  m[0]=0; m[1]=1;
  f(m);
}
```

文件link1.c

```
long k[2] = {0,1};
```

文件link2.c

```
#include <stdio.h>
typedef struct {
  long n[2];
} stype;
extern stype k;
main() {
  printf("%d,%d\n",
    k.n[0],k.n[1]);
}
```

*第 12 章

面向对象语言的编译

软件系统的规模越来越大，并且日趋复杂，以更有效和更透明的方法来开发这样的系统的呼声与日俱增。最终的目标是从已做好的标准构件去构造软件系统，就像现在构造硬件系统那样。模块化、模块的可重用性、模块的可扩充性和抽象性是朝向这个目标的一些尝试，而面向对象语言在这些方面提供了一种新的可能性。现在，面向对象已被看成管理复杂软件系统的一种重要风范。

本章回顾面向对象语言的一些重要概念，概述它们的实现技术。为突出一些重要概念的实现技术，本章以 C++语言为例，介绍如何将 C++程序翻译成 C 程序；实际的编译器大都把 C++程序直接翻译成低级语言程序，而不是把 C 语言作为中间语言并利用 C 语言编译器。

12.1 面向对象语言的概念

面向对象语言可以看成是命令式语言，除了变量、数组、结构体和函数等熟知的概念外，它还引入一些新概念。本节只回顾其中主要的概念。

12.1.1 对象和对象类

命令式语言的主要模块化单元是过程（包括函数）。在数据的复杂性与处理的复杂性相比显得微不足道时，这是适宜的抽象和模块化。但是，当任务的描述和有效的解决方案需要使用复杂的数据结构时，单用函数进行模块化是不够的。有效处理这些任务的合适抽象级别应该是允许把数据结构和操作这些数据结构的相关函数封装在一个单元中。

面向对象语言最基本的概念是**对象**。一个对象由一组**属性**和操作于这组属性的过程组成，属性到值的映射称为对象的**状态**，过程也叫做**方法**。属性和方法共同形成了对象的**特征**。因此，对象封装了数据及其上的操作。面向对象语言最重要的基本操作是激活对象 o 的方法 m，写成 o.m。在这儿，对象扮演着主要角色，而方法是对象的成分并且从属于对象。

面向对象语言拓广了 Pascal 和 C 等命令式语言的类型概念，增加了**对象类**（简称类）的概

念。一个对象类规范了该类中对象的属性和方法,包括它们的类型和原型(参数和返回值类型)。一个对象要想属于该类,它必须含有这些特征,当然还可以含有其他一些特征。某些面向对象的语言,例如 Eiffel,允许对方法作进一步规范,例如规定方法的前置条件和后置条件,这是提供一种约束方法含义(语义)的手段。

对象类形成了面向对象语言的模块单元。对于圆、椭圆、矩形、三角形、多边形、点、直线和折线等图形对象,可以为不同类型的图形对象定义不同的对象类。同一类中的不同对象,它们的属性值可能不一样,因此它们必须有自己存放属性的存储单元;但是,它们的方法是一样的,因此它们可以共享方法的代码。这一原则指导 12.2 节开始介绍的 C++程序到 C 程序的翻译。

下面将把术语“类”和“类型”看成是同义的。

12.1.2 继承

继承定义为类 A 的所有特征并入到新的类 B,B 可以进一步定义自己的一些其他特征,在一定条件下还可以重写或覆盖(overwrite)从 A 继承来的方法。某些语言允许重新命名继承来的特征,以避免名字冲突,或者允许在新的上下文中使用更有意义的名字。

如果类 B 继承类 A,那么类 B 叫做类 A 的**派生类**,而类 A 叫做类 B 的**基类**。

继承是面向对象语言最重要的概念之一。继承的层次性允许把类库结构化和引入不同级别的抽象。仍以图形对象的例子来说明这一点。图 12.1 给出了图形类库中继承层次的一部分。在该图中,用椭圆表示对象类,椭圆中包括类名和该类的部分方法;继承由箭头表示。例如,图形对象类 GraphicalObj 有方法 translate 和 scale,那么所有的图形对象都可以被平移(translate)和缩放(scale)。封闭图形类 ClosedGraphics 和折线类 PolyLine 从类 GraphicalObj 继承了这些方法。在类 PolyLine 中,这些被继承的方法被覆盖,并且引入计算周长的方法 length。类 ClosedGraphics 引入新方法 area,它计算封闭图形对象的面积。多边形类 PolyGon 同时继承类 ClosedGraphics 和

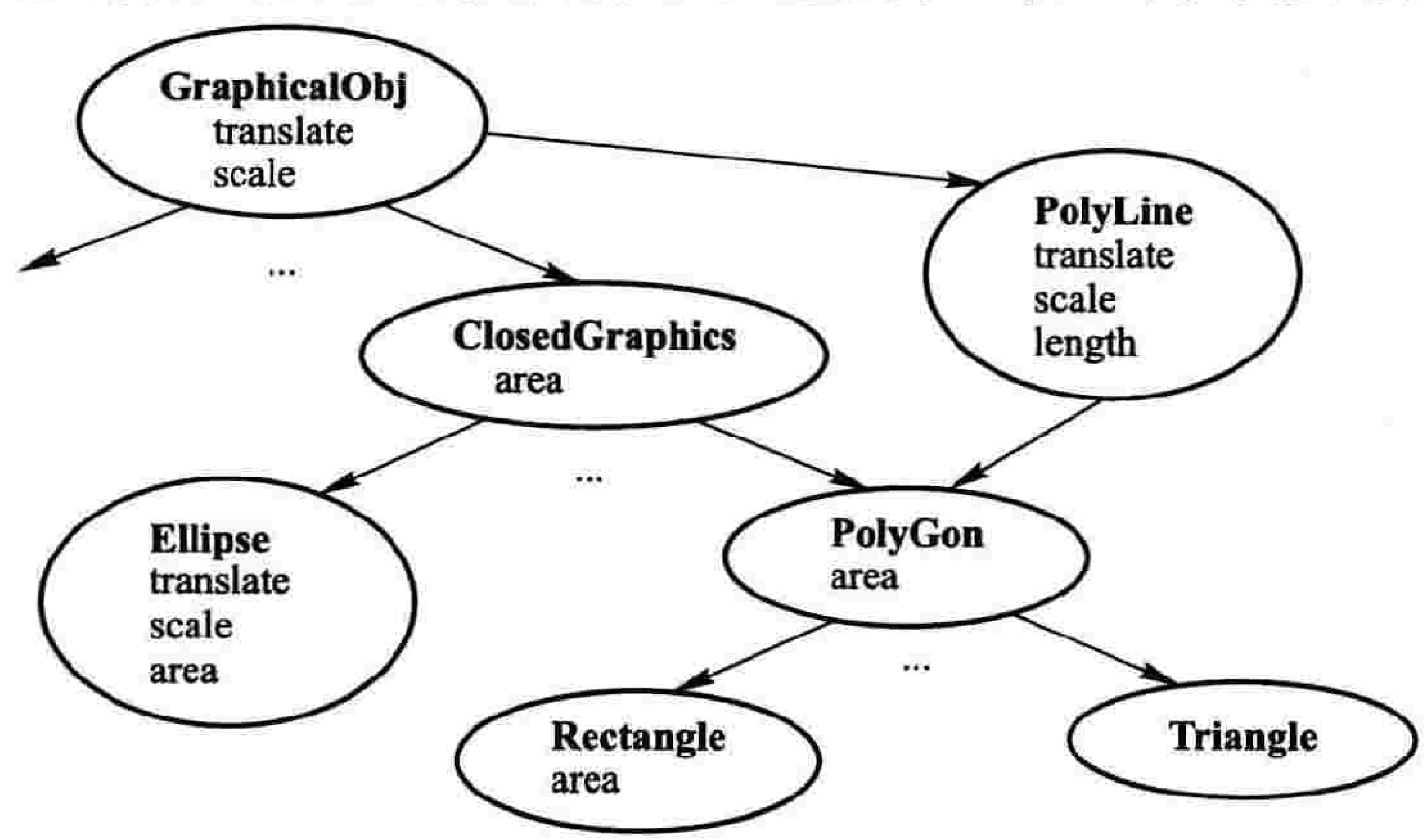


图 12.1 图形对象的继承层次结构

类 PolyLine，其中 area 被覆盖。最后，矩形类 Rectangle 继承类 PolyGon 并且覆盖 area。

虽然方法 translate 和 scale 在类 GraphicalObj 中引入，但是它们却不能在这儿定义。因为这些方法只能结合具体的图形对象来定义，例如椭圆类 Ellipse 和折线类 PolyLine。不过，这些方法在类 GraphicalObj 中引入，意味着每个图形对象必须有这些方法，虽然这儿并不提供它们的实现。包含未定义方法的类叫做**抽象类**，它没有自己的任何对象实例。

类似地，类 ClosedGraphics 引入方法 area，但该方法也不能在这个类中定义。area 首先在类 Ellipse 和类 PolyGon 中定义。但是，对一般的多边形而言，面积计算是复杂的，它取决于该多边形能划分成多少个三角形以及这些三角形的面积之和。另一方面，矩形的面积计算是简单的。如果矩形经常使用，最好在类 Rectangle 中以更有效的方法实现 area，如引入矩形的边长属性，利用它们直接计算面积。对于类 PolyLine 的所有方法，类 PolyGon 最好接管所有这些方法，不作任何覆盖。

由上所述，继承概念提供了这样的可能性：用简单的办法可以复用部分现有实现，或扩充它们，还可以重写个别方法以适应局部的特殊需求和环境。

此外，任何事物，如果它能用较高级别的抽象表述，那么它就有较广的应用可能性，因而有较大程度的可复用性。所以应尽可能使用高的抽象级别。另一方面，有时必须转移到更具体的级别，使得在解决某些特定问题时有更多的结构可用。假定图形对象的一种变换可以由一串平移、缩放和图形对象的一些其他操作描述，那么该变换可以用一个函数实现，该函数对任何类型的图形对象都可用。如果无抽象类并且编译器要进行类型检查，那么需要为每个类写一个这样的函数，即使这些函数的实现具有相同的内容也不能省略。

于是，类型化的面向对象语言考虑其类型系统中的继承层次结构。如果类 B 继承类 A，那么指派给 B 的类型是指派给 A 的类型的子类型。子类型的每个对象自动地是它超类型的一个元素。一个继承类是被它继承的类的**子类**。这样就有下列三条规则。

(1) **子类型规则**：当在某个输入位置（函数的输入参数、赋值的右部）需要某个类型的一个对象时，该类型的任何子类型的对象可以出现在这些位置。若函数的返回值要求是某个类型的一个对象，该类型的任何子类型的对象也可以作为返回值。

类 B 的一个对象，若它不同时是 B 的某个真子类的对象，那么称该对象是 B 的一个**真对象**，称 B 是该对象的**运行时类型**。这样，每个对象有唯一且确定的运行时类型，它是该对象所属的最小类型。此外，一个对象也是它运行时类型的每个超类型的元素。

凭借子类型规则，面向对象语言的方法可以接受运行时类型不同（因而结构也不同）的对象，例如在参数位置就可以这样。这是多态性的一种形式。

继承的子类型规则，以及继承类可以覆盖被继承的方法，引起了一个有意思的结果。它对编译器来说是重要的。例如有一个函数 f，它允许类 ClosedGraphics 的对象作为参数，并假定 f 调用该参数的 area 方法。因为类 ClosedGraphics 不能定义 area 方法，因此调用的实参肯定是类 ClosedGraphics 某子类的对象，而不是它本身的真对象。下面是一般性的规则。

(2) **方法选择规则**：如果类 B 继承类 A 并且重写了方法 m，那么对类 B 的对象 b 来说，即使

它作为类 A 的对象使用,也必须使用在类 B 中定义的方法 m。

该规则给编译器提出一个问题:编译器必须产生调用一个方法的代码,但在编译时它很可能确定不了究竟要调用哪一个方法。对上面的例子来说,在 f 的代码生成中,编译器不知道应该把名字 area 绑定到哪个类的 area 方法。一般来说,这个绑定只有在运行时当实参已经明确的情况才能完成,这样的绑定叫做**动态绑定**。因此,为体现动态特征,也可以把方法选择规则按下面方式表述。

(3) **动态绑定规则**:当对象 o 的一个方法可能被子类重新定义时,如果编译器不能确定 o 的运行时类型,那么必须对该方法进行动态绑定。

本章的主要部分就是讨论继承的有效实现。

12.1.3 信息封装

大多数面向对象语言提供了一种机制,它可用来将类的特征分成私有的(private)和公共的(public)两类。私有特征完全不可见,或者至少在某个上下文中不可见。某些面向对象语言用不同的上下文区分作用域,如“在一个类中”、“在派生类中”、“在友元类中”等。语言构造或一般性规则可以指明上下文的可见、可读、可写或可调用特征。

由编译器来实现这些作用域规则是简单而明显的。因此,虽然信息封装是非常重要的,但是下面不讨论它的实现。

现在,把到目前为止讨论的概念小结如下。

(1) 面向对象语言引入了新的模块化单元:对象类。对象类可以封装数据及在这些数据上的操作。

(2) 继承概念对构造现有模块(对象类)的派生模块是极其有用的。

(3) 面向对象语言的类型系统使用继承概念:派生类是其基类的子类型,派生类的对象可以用在其基类的对象所允许出现的任何地方。

(4) 继承层次结构把不同级别的抽象引入程序。即在程序或系统的不同点使用不同级别的抽象。

(5) 抽象类型可用作规范(规格说明),通过逐步继承而求精,直至最终的实现。也就是说,它提供从规范经过逐步设计到各种实现的无缝转变。

本章将以面向对象语言 C++和 Eiffel 为例。

12.2 方法的编译

本节用具体的例子来说明方法的编译。首先,用 C++描述前一节图形对象的部分类层次结构。

先定义一般的图形对象类 GraphicalObj 如下:

```
class GraphicalObj {
     virtual void translate (double x_offset,double y_offset);
     virtual void scale (double factor);
     …      // 可能还有一些其他方法
};
```

其中方法 translate 和 scale 声明为虚方法。在 C++中,这是要求它的派生类重写此方法。

类 GraphicalObj 的一个特别重要的子类是点类 Point。几乎每一种具体的图形对象类的实现都以某种方式使用点。类 Point 的定义如下:

```
class Point : public GraphicalObj {
     double xc,yc;
   public :
     void translate (double x_offset,double y_offset) {
          xc+=x_offset;
          yc+=y_offset;
     }
     void scale (double factor) {
          xc *=factor;
          yc *=factor;
     }
     Point(double x0=0,double y0=0) {xc=x0; yc=y0;}
     void set(double x0,double y0) {xc=x0; yc=y0;}
     double x(void) {return xc;}
     double y(void) {return yc;}
     double dist (Point &);
};
```

一个点有 x 轴和 y 轴坐标 xc 和 yc,它们表示点在二维空间中的位置。xc 和 yc 是点的私有数据,只能在该类的方法中访问(或者由友元函数访问)。在类 Point 中定义的方法是公共的,它可以在知道一个点的任何场合使用。例如,如果 p 是一个点,那么 p.x()激活 p 的方法 x 并且返回 p 的 x 轴坐标;p.translate(1,2)通过改变 p 的坐标,使它在 x 轴方向上移动一个单位,在 y 轴方向上移动两个单位。

一般而言,对一个对象 o,它带有参数 $arg_1,\cdots,arg_n$的方法 m 由 o.m($arg_1,\cdots,arg_n$)激活。

将一个 C++语言的类翻译成 C 语言的程序段,主要工作有如下几点(由继承引出的问题放在 12.3 节考虑)。

(1) 将 C++语言中一个类的所有非静态属性构成一个 C 语言的结构体类型,取类的名字作

为结构体类型的名字。

(2) 类的静态属性是该类的所有对象所共有的,应当翻译成C中的全局变量,但是需要改一个名字。根据下面(4)的(a),读者应该明白如何改名。

(3) C++语言中类的对象声明不加翻译就成了C语言中相应结构体类型的变量声明,不管对象声明出现在程序中的什么地方。

(4) 在解释类的非静态方法的翻译之前,先做个约定。在C++语言中,函数的参数传递有值调用和引用调用两种方式,当形式参数名前加字符 & 时,表示该参数是引用调用。但是C语言的参数传递只有值调用。为简洁起见,下面假定C也有引用调用方式,也用字符 & 表示这种方式,以避免在解释如何翻译引用调用上花笔墨。

将C++语言中类的非静态方法翻译成C语言的函数,对应的方法和函数的区别如下。

(a) 由于类声明在C++语言中形成一层作用域,类中方法声明的作用域就是该类;而在C语言中,函数声明的作用域至少是所在的文件。为了避免不同类的同名方法在C程序中变成同名函数,函数的名字必须在原来方法名的基础上修改,比较容易的做法是把类名字加上去。考虑到方法的重载,参数类型也编码到函数名中,才能保证不会有名字冲突。

(b) 和方法声明相比,函数声明增加一个形参,作为它的第一个形参,该形参的类型就是对应该类的结构体类型,该形参的名字通常取 this,传递方式是引用调用。

(c) 和方法调用相比,在函数体中出现的函数调用也要增加一个实参,作为它的第一个实参。若原来是调用本对象的方法,那么新增的实参就是 this;若是调用其他对象的方法,则新增实参是该对象对应的结构体变量。

(d) 在方法中对本对象的非静态属性的访问,改成对 this 相应域的访问。在方法中对其他对象的非静态属性的访问不必修改,直接就成了对对应结构体变量的相应域的访问。若是对静态属性的访问,则翻译成对C的全局变量的访问。

(5) 类的静态方法在定义和调用时,与该类的特定对象无关,因此在翻译时,无须增加表示当前操作对象的参数,只需要按(4)中的(a),将方法名改成函数名即可。

必须注意一点,对C++程序的语法和语义分析在这个翻译之前进行,因此生成C程序时,C++语言的可见性规则已经检查过;C的可见性规则虽然不一样,这时也没有什么影响了。

下面用一个例子来看方法是怎样用等价的函数实现的。

例 12.1 如果 m 是类C的一个非静态方法,它的原型是“返回值类型 m(形参表)”,那么等价于 m 的函数 fm 的原型是(下面给出的语法是非标准的):

返回值类型 fm(C &this,形参表)

C &this 表示第一个形参的类型和它的名字,传递方式是引用调用。若 m 中有对当前对象的非静态属性 k 的访问,有对 m 本身的递归调用,有对某个对象 o 的非静态方法 n 的调用 o.n,有对 o 的非静态属性 k 的访问 o.k,那么,它们的翻译结果见表12.1。

表 12.1 类 C 的方法 m 被翻译成函数 fm

	方法	函数
原型	返回类型 m(形参表)	返回类型 fm(C &this,形参表)
调用	m(实参表) o.n(实参表)	fm(this,实参表) fn(o,实参表)
属性访问	k o.k	this.k o.k

类 Point 的方法 x 翻译成等价的函数 x__5Point,其定义是:

```
doublex__5Point (Point &this) {return this.xc;}
```

方法调用 p.x()翻译成 x__5Point(p)。

类 Point 的方法 translate 翻译成函数 translate__5Pointdd:

```
void translate__5Pointdd(Point &this,double x_offset ,double y_offset) {
    this.xc+=x_offset;this.yc+=y _offset;
}
```

方法的名字本身没有作为实现函数的名字,而是扩展成还包含所属类的类名和参数类型的编码。把类名编码加到实现函数的名字中是必要的,类名的编码保证了生成的函数名是唯一的。考虑到方法的重载,在 C++ 的实现中,参数类型也编码到函数名中。例如,名字 translate__5Pointdd有下列 5 个部分:

(1) 方法名 translate;

(2) 分隔符__;

(3) 类名的编码 5Point,领头的数表示后面多少个字符属于类名;

(4) double 类型的编码 d(第一个参数的类型);

(5) double 类型的编码 d(第二个参数的类型)。 □

从本节可以看出把 C++这样的面向对象语言的程序翻译成 C 语言程序的可能性。

12.3 继承的编译方案

继承是面向对象语言引入的最重要概念。本节讨论它的实现。

如果类 B 直接或间接继承类 A,那么类 A 是类 B 的超类,类 B 的对象可以用在几乎所有类 A 的对象可用的地方。出于效率的考虑,编译器要求类的对象具有某种灵活的结构,因为为了使类 B 的对象可以作为类 A 的对象使用,编译器必须能以一种有效的方式产生类 B 对象的 A 视图。在这种视图中,编译器关于类 A 的对象结构的假设必须满足。

大家知道,类 A 的虚方法可以在类 B 中被重写。12.1.2 节给出的动态绑定规则要求,如果编译器不能直接确定类 A 的对象 o 的运行时类型,那么该方法应该动态绑定。例如,如果 o 的运行时类型是 B,那么应该使用 B 的方法,而不是 A 的方法。在这样的情况下,编译器必须做一些准备,使得在程序运行时,被激活方法所期望的视图(即 B 视图)能够有效地从 A 视图产生。

许多面向对象语言,尤其是一些老的面向对象语言,仅支持单一继承。12.3.1 节先讨论编译单一继承的一种合适方案。12.3.2 节转向编译多重继承这个更加复杂的问题。

12.3.1 单一继承的编译方案

对于只有单一继承的语言来说,每个类最多从一个类继承。这种语言的继承层次结构是树或森林。

先给出一个例子。图 12.2 的程序描述类 PolyLine 和它的派生类 Rectangle。

```
#include "graphicalobj.h"          /* imported GraphicalObj */
#include "list.h"                  /* imported lists */
#include "point.h"                 /* imported Point */
class PolyLine : public GraphicalObj {
    list <Point> points;
  public:
    void translate (double x_offset, double y_offset);
    virtual void scale (double factor);
    virtual double length (void);
};

#include "polyline.h"
class Rectangle : public PolyLine {
    double side1_length, double side2_length;
  public :
    Rectangle (double s1_len, double s2_len, double x_angle=0);
    void scale (double factor);
    double length (void);
};
```

图 12.2　类 PolyLine 和类 Rectangle

出于效率的原因,在矩形类 Rectangle 的定义中引入属性 side1_length 和 side2_length 存储矩形两条邻边的长度,它允许得到一个效率较高的 length 的重新定义。scale 也必须重新定义,因为缩放操作会改变边的长度。此外,translate 可以直接由 PolyLine 的相应定义接管,因为平移操作不改变边的长度。

仍必须解释编译器是怎样有效地实现动态绑定的。在上述例子中,PolyLine 的 scale 方法不能用于缩放矩形,因为它不知道新加的矩形边长属性也由缩放操作改变,因此必须使用 Rectangle 的 scale 方法。大家知道,类 Rectangle 的对象可以作为方法的参数传递,只要超类 PolyLine 的对象在这些地方被允许,例如,它可以用于函数 zoom:

```
void zoom (GraphicalObj &obj,double zoom_factor,Point &center) {
    obj.translate (-center.x,-center.y);              // 将"中心点"移至"点(0,0)"
    obj.scale (zoom_factor);                          // 缩放
}
```

该函数首先平移一个图形对象,使得中心点 center 落到原点,然后根据值 zoom_factor 缩放该对象。

如果函数 zoom 作用于矩形,那么 zoom 的体必须调用 Rectangle 的缩放函数,而不是 PolyLine 甚至 GraphicalObj 的缩放函数。然而,zoom 可能在类 Rectangle 被定义前已经被编译并存于库中。也就是说,当编译 zoom 时,编译器不知道运行时在 zoom 体应该激活哪个方法。而且,在 zoom 的不同调用点,应该激活的方法可能还是不一样的。因此,编译器不可能把 scale 绑定到一个具体的方法,而不得不由 zoom 在执行时将 scale 动态地绑定到一个方法。

编译器可以用下面的方案来有效地处理动态绑定。对每个类,编译器建立一张方法表,该表包含了定义在该类中并且必须动态绑定的方法。在 C++中,这样的方法表叫做**虚函数表**,它们包含一个类或它的超类中所有定义为 virtual 的方法的入口。12.2 节提到,每个对象在 C 程序中有对应的结构体,现在为这样的结构体增加一个域,作为第一个域,其内容是这样的方法表的指针。编译器把方法名绑定到方法表的索引。当运行时调用某方法时,存储在方法表中相应索引下的函数被激活。继承类的方法表按如下方式产生:首先复制其基类的方法表,其中,被重新定义的方法由新的定义覆盖;然后,新引入的方法被追加到这张表上。这就保证了基类中定义的方法名在新类中具有相同的方法表索引。

如果 B 是一个类,A 是 B 的超类,那么类 B 的对象 b 的 A 视图包含两部分:b 的前一部分域和 b 引用的方法表的前一部分。其中属于 A 视图的 b 的前一部分域,是由方法表的指针和从 A 继承的属性组成;属于 A 视图的方法表部分包括在 A 及其超类中引入的方法的索引。b 的其他各种视图由一个指向 b 的指针以同样的方式表示。

图 12.3 和图 12.4 的例子用来解释该编译方案。

图 12.3 给出了 GraphicalObj、PolyLine 和 Rectangle 的方法表。PolyLine 的方法表是从 GraphicalObj的方法表派生出来的。首先,由 PolyLine 重新定义的方法 translate_PL 和 scale_PL 取代 GraphicalObj 中的相应方法。然后新定义的虚方法 length_PL 被加入。依次地,Rectangle 的方法表从 PolyLine 中的方法表派生。在 Rectangle 中重新定义的方法取代 PolyLine 的相应方法。没有重新定义的方法 translate_PL 仍然保留。编译器把 translate、scale 和 length 分别绑定到方法表索引 0、1 和 2。

图 12.4 给出了 Rectangle 的对象表示。除了自身的状态外,每个这样的对象包含指向类

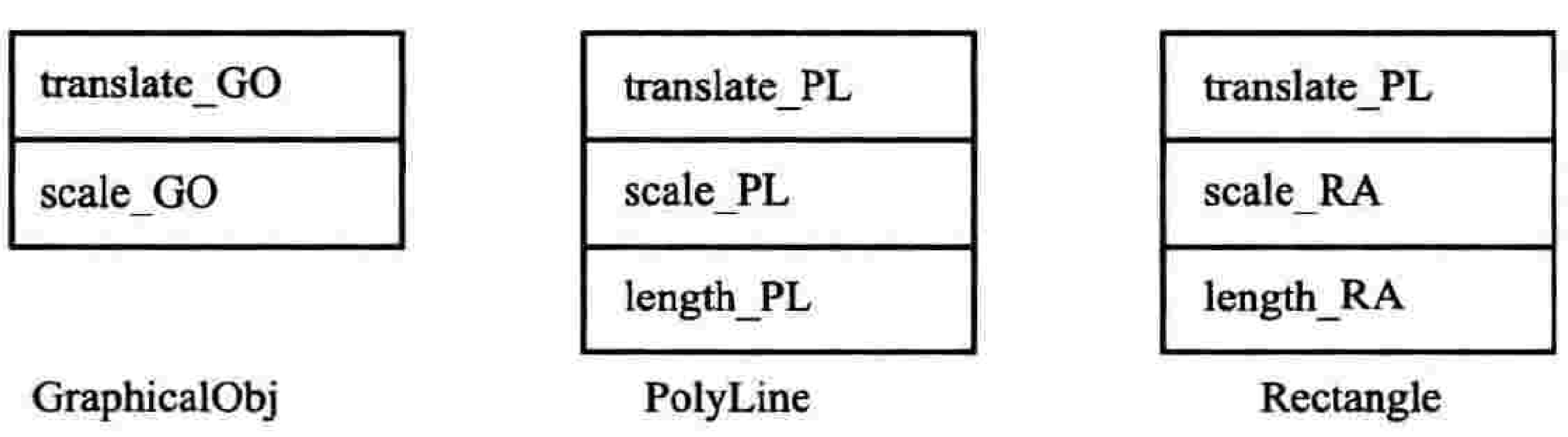


图 12.3 图形对象的不同子类的方法表

Rectangle 方法表的指针。

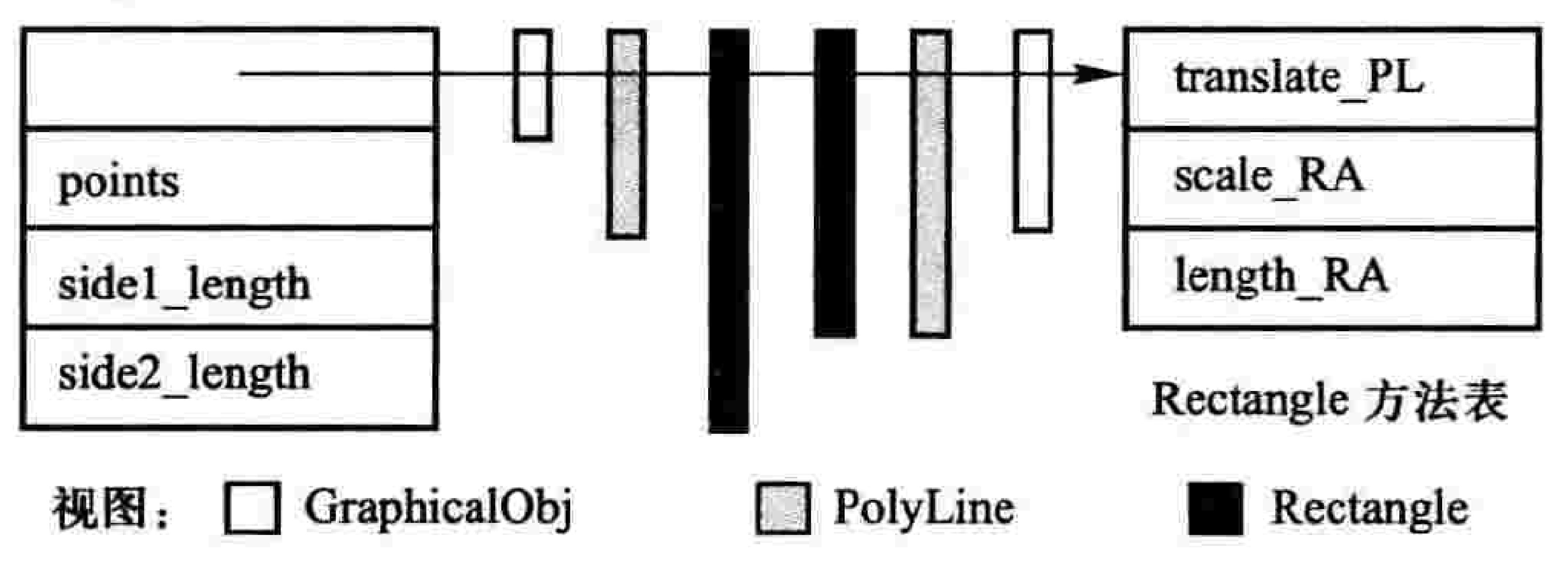


图 12.4 Rectangle 的对象表示

用动态绑定来实现单一继承，每个对象需要一个指针的额外存储空间开销。另外，和每个类关联的方法表需要存储空间。动态绑定引起了运行时方法调用的时间增加，因为通过指针找到方法表以及获得要被激活方法的入口地址需要消耗时间。

12.3.2 多重继承的编译方案

单一继承的编译方案比较容易，而多重继承对语言定义和编译器设计来说，都具有很大的挑战性。

在有多重继承的语言中，一个类可以从多个类继承，即一个类可以有多个直接的超类。因此继承层次结构不再是树，而是有向无环图。

多重继承的使用支持这样的编程风格：基本功能用一些很小的基类实现，通过继承用这些基类去构造更加复杂的类。多重继承也可用于抽象类，例如，可以把一个抽象类给出的规范和作为它的实现的一个具体类集成起来。比方说，对于一个预先定义了大小的栈，这样的类可以通过从抽象类 Stack 和作为其实现的具体类 Array 的继承来实现。

所有会碰到的问题和可能的解决办法都可以通过双重继承来阐明。本节假定类 C 同时从类 B1 和 B2 派生，见图 12.5。下面两点引出了语言定义中的问题，在某种程度上也引出了编译器设计的两个问题。

A
B1
B2
C

图 12.5 多重继承

(1) B1 和 B2 间的冲突与矛盾。例如，如果在两个基类中，方

法或属性使用相同的名字,那么继承将引起冲突。

(2) 多重继承。例如,若 B1 和 B2 都直接从 A 继承,则 C 将从 A 多重继承。

问题(1)基本上是语言定义问题。下面一些解决办法可以独立或组合地应用于语言设计中:

① 将 B1 定义为主要后代,冲突解决优先于 B1。这种办法主要用于解释执行的 LISP 的面向对象的扩充上。它通过预先定义的次序查找继承层次结构来动态地绑定名字。在图 12.5 中,先 C,然后 B1,最后 B2。以首先找到的为准。

这种办法并非没有危险,因为可能的冲突并非都是明显的。因此,当使用上述解决冲突的策略时,并非所有的冲突对程序开发者来说都总是清楚的。

② 语言允许重新命名被继承的特征,因而允许程序员通过显式的干预来解决可能的冲突。使用这种方式的有 Eiffel 语言。

③ 语言提供别的显式手段来解决冲突。例如,如果 B1 和 B2 中名字 n 的定义有矛盾,那么 B1::n 或 B2::n 将无二义地指明应该使用 B1 还是 B2 的 n 定义。使用这种方式的有 C++语言。

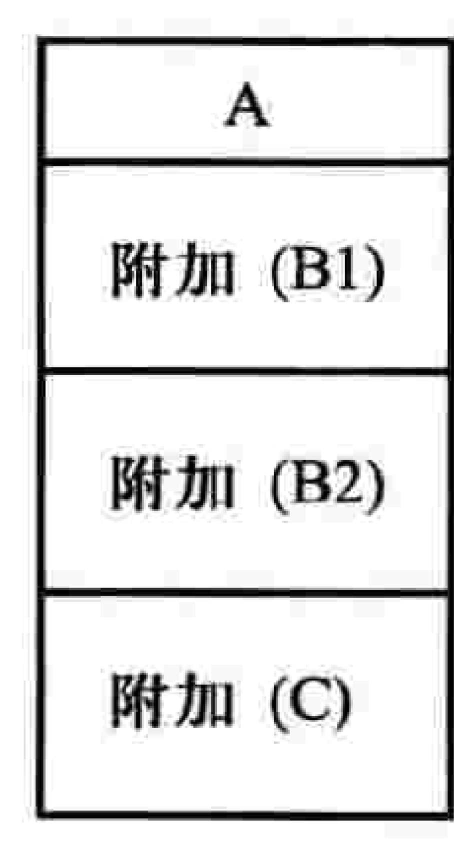
对于这些解决办法,实现起来并无什么困难,只涉及编译器符号表的组织和管理问题。这里不再继续讨论。

可是对前面的问题(2)而言,存在两种截然相反的解决方法:

① 被多重继承的类可以有多个实例(见图 12.6)。

② 被多重继承的类只能有一个实例(见图 12.7)。

这两种方法各有优缺点。在有些场合下需要被多重继承类有多个实例,而在另一些场合只需要它们的单个实例。甚至会有这样的需求:对被多重继承的某些特征需要单个实例,而对另一些特征则需要多个实例。Eiffel 提供这种灵活性。



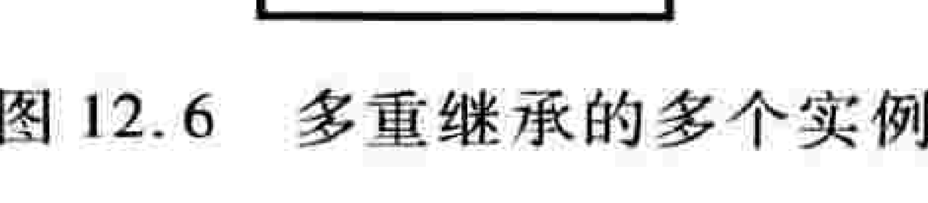


图 12.6 多重继承的多个实例

图 12.7 多重继承的单个实例

C++语言包含了对上述两种方法的支持。以图 12.5 所示的多重继承为例,在 C++中,当把这些类定义为:

```
class A {…};
class B1 : public A {…};
```

```
class B2 : public A {…};
class C : public B1,public B2 {…};
```

时，不论是类 B1 或类 B2 都内含一个类 A 的副本，这样在类 C 的对象布局中将包含两个独立的类 A 子对象。在 C++中，一般直接称这种继承为多重继承。

若将类 B1 和类 B2 设定为从类 A 虚拟继承，再让类 C 从 类 B1 和类 B2 继承，即

```
class A {…};
class B1 : public virtual A {…};
class B2 : public virtual A {…};
class C : public B1,public B2 {…};
```

这时，在 C 的对象布局中将只包含一个类 A 子对象。

下面考虑仅允许被多重继承的类有多个实例的编译方案。这些方案比同时还允许单个实例的编译方案要简单些，产生的代码效率也高些。在此讨论独立的多重继承的编译方案。

在独立的多重继承情况下，来自基类的继承是相互独立的。相应地，继承类 C 的对象包含基类 B1 和 B2 的完整副本，如图 12.8 所示。

B1
B2
附加 (C)

图 12.8 独立的多重继承时的对象结构(程序视图)

如图 12.6 所示，多重继承在下述情况导致冲突和二义：

(1) 当多实例的特征被用于访问、调用和覆盖的时候；

(2) 当类 C 的对象的 A 视图被建立时，因为类 C 的对象包含多个类 A 的子对象。

可见性规则在某些情况下可用于避免产生这些问题。例如，C++允许一个类对它的继承者隐藏它自己的继承性。比如，B1 可以私有地从 A 继承而 C 并不知道这一点，此时 B1 不是 A 的子类，并且 C 中不会由于 A 的多个实例而出现二义。在可见性规则不足以消除二义的地方，需要引入额外的语言构造：在 C++中，受限算符"::"可以用于 B1::f 的形式以表示特性 f 属于 B1。此外，可以使用类型转换，如显式地把 C 的对象转换成 B1 的对象。

现在把单一继承的编译方案扩充到这种独立的双重继承场合。在单一继承的情况下(见图 12.4)，为了有效地实现方法的动态绑定，在每个对象中加入了一个指针作为该类的第一个成分，该指针指向方法表。这里仍然保持这种方式。

注意，对于类 C 的每个超类 B，编译器必须能够产生类 C 对象的 B 视图。因为 B1 子对象处在 C 对象的开头，因而对 B1 仍可以使用单一继承的办法，即 C 对象的 B1 视图是 C 视图的开头部分(对象成分和方法表都这样)。但是不能用 C 视图的开头部分作为 B2 视图，因为一个对象的 B2 视图必须有一个方法表指针作为它的第一个成分，该方法表的内容是依据 B2 和 C 确定的；跟随该指针的是 B2 的属性值。这就导致了下面的方法：在 C 对象中，在 B2 属性值的前面加上指向另一方法表的指针；这个方法表由 B2 方法表复制后经 C 中重新定义的方法覆盖而产生。于是，B2 视图由一个 B2 引用表示，它指向 B2 子对象的开始。B2 子对象的第一个成分是指向 B2 子对象方法表的指针。对于超类 A 的每个实例，编译器知道 C 对象中对应子对象的偏移。编

译器通过把这个偏移加到 C 引用而产生相应的 A 引用。得到的结构显示在图 12.9 中。

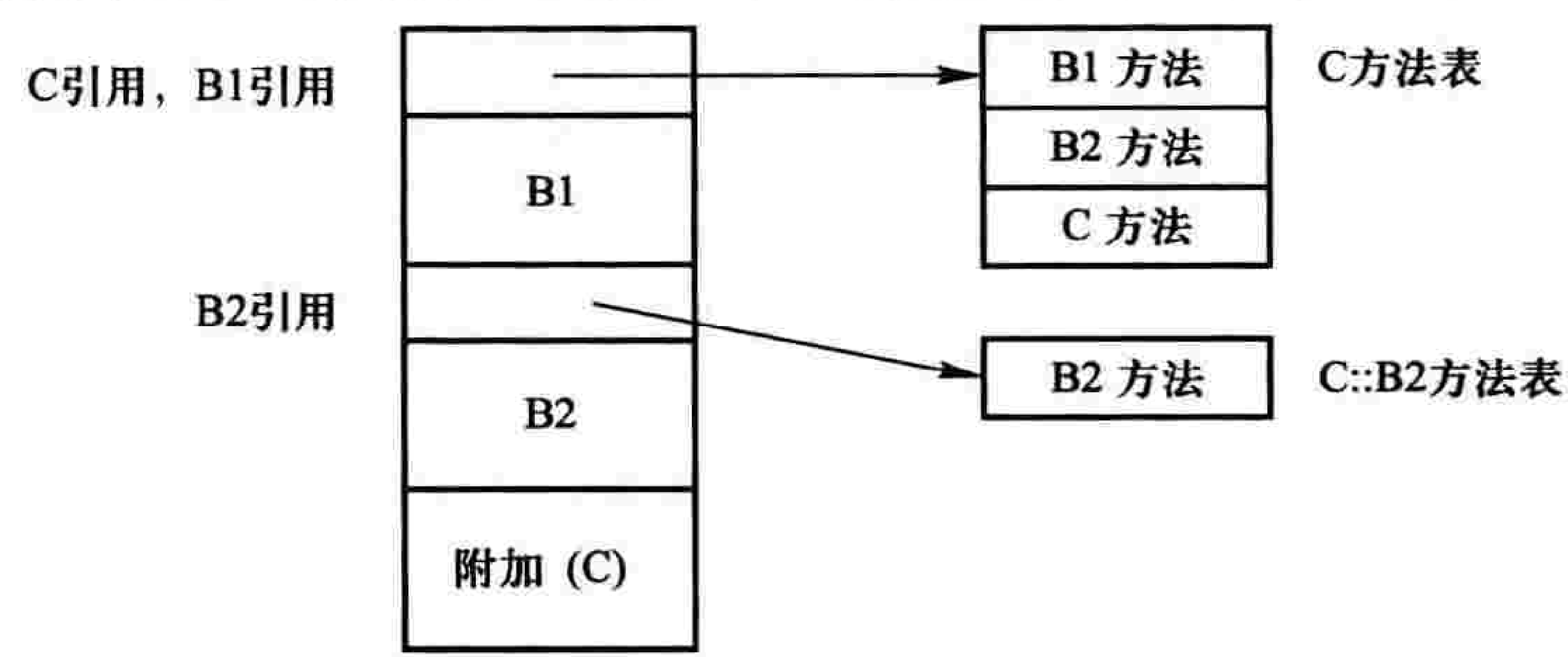


图 12.9 独立的多重继承的对象结构(实现视图)

下面通过例子来理解这种偏移在对象视图切换中的应用。比如,对于语句:

```
B2 * pb2=new C;
```

则新创建的 C 对象的地址在赋给 pb2 时必须调整,以指向其 B2 子对象,即提供 C 对象的 B2 视图。编译时将会产生以下代码:

```
C * temp=new C;
B2 * pb2=temp ? temp+sizeof(B1) : 0;
```

使得 pb2 是 B2 引用,以便通过它使用 B2 子对象的特征。

当程序员要删除 pb2 所指的对象时,如:

```
delete pb2;
```

如果指针 pb2 所指对象的运行时类型是 C,那么 pb2 的值必须从 B2 引用调整到 C 引用,即调整到 C 视图。然而,由于 pb2 所指对象的运行时类型一般不是静态可确定的,因此上述调整的偏移量一般不是静态可确定的。

更复杂的情况是,在 C 中定义的方法总是期望得到它为之激活的对象的 C 视图。如果一个这样的方法覆盖超类 A 的一个方法(更精确地说,超类 A 某实例的方法,因为 C 可以含 A 的几个实例),那么在该方法被激活时,可能只有相应对象的 A 视图可用。运行时,必须能够从它计算出所需的 C 视图。如果 A 和 C 都是已知的,那么这很容易做:因为视图可由相应的引用表示,并且对于每个 C 对象,A 引用和 C 引用之间的差 d 是一个常数,即在 C 对象中 A 子对象的偏移,因此 C 引用可以从 A 引用减去 d 计算出来。可惜,在该方法调用(它可以出现在 B1 或 B2 某方法的代码中)被编译和运行时,C 可能是未知的。

基于这些原因,编译器把用于确定所需视图的偏移存放在方法表中下邻该方法指针的地方。即方法表中的每个入口有两个成分:方法指针和偏移,使得该方法所期望的视图可以从该方法激活时的可用视图中生成。

于是第 i 个虚函数的调用操作,由

```
( * pb2->vptr[i])(pb2);
```

改为:

(* pb2->vptr[i]. faddr) (pb2+pb2->vptr[i]. offset) ;

其中 fadd 是虚函数的地址,offset 是调整视图的偏移。

这种做法的缺点是,不管是否需要用 offset 来调整,所有的虚函数调用都必须这么操作。

图 12.9 描述的办法有一个虽小但很重要的问题:类 C 的两个方法表包含 B2 方法表的两个(修改过的)副本。结果是,存放类 C 的方法表所需的存储空间可能会随 C 定义的复杂度增加而呈指数增长(类定义的复杂度指的是类的展开定义(unfolded definition)的大小,而类的展开定义是通过把类定义中的基类定义用该基类的展开定义代替而得到的)。

为了避免呈指数增长,必须只使用 B2 方法表的一个副本。为做到这一点,把 C 方法表以分布方式存储,如图 12.10 所示。该图隐蔽了一个有意义的细节:B1 和 B2 方法表的副本也可以按分布的方式存储。

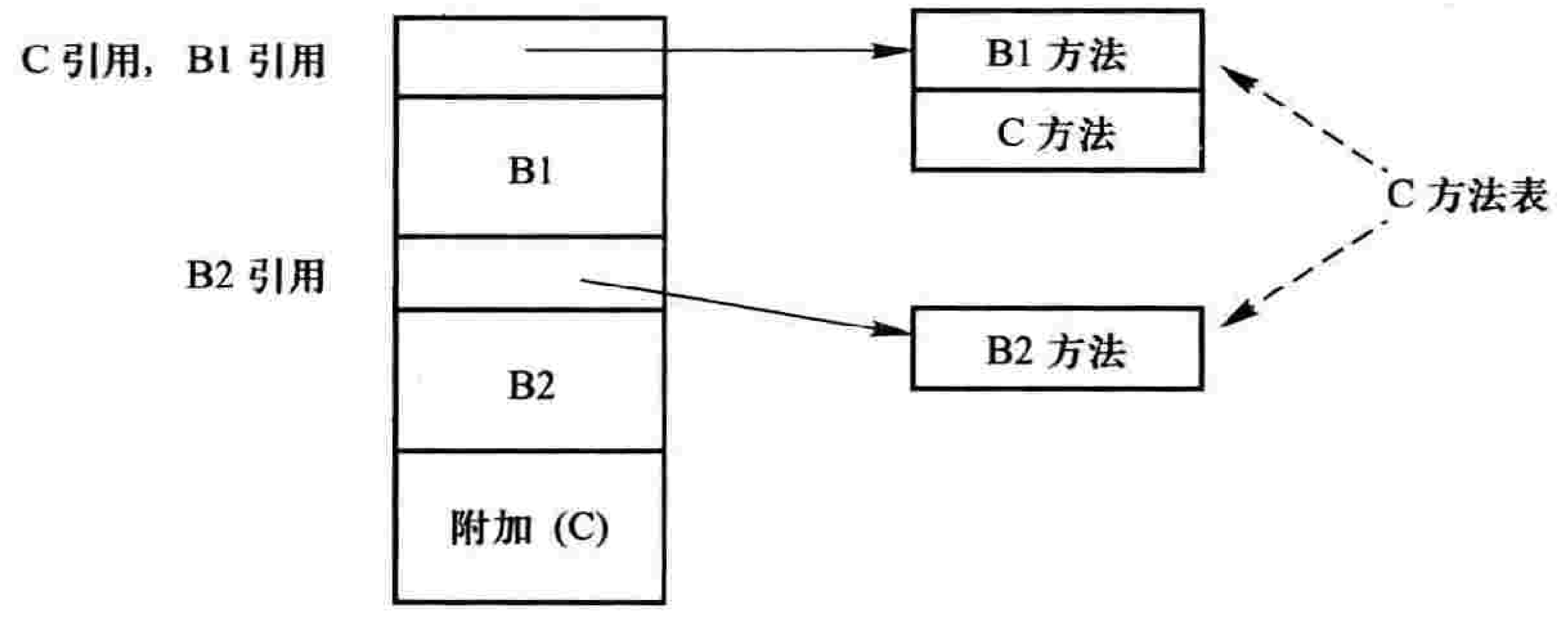


图 12.10 独立的多重继承的对象结构(实现视图)

单个实例的多重继承,以及其他一些形式的多重继承,它们的实现都比较复杂,在此不介绍。

习 题 12

12.1 图 12.11 给出图形对象类库的另一片断。根据 12.3.1 节的编译方案,确定该图中定义的类的方法表和方法索引。

12.2 根据独立的多重继承的编译方案,确定图 12.11 中 Circle 的方法表和方法索引,并且编译形式为 c. dist (p)的方法调用,其中 c 和 p 分别是类 Circle 和 Point 的对象。

```
class ClosedGraphics : public GraphicalObj {
  public:
    // Surface area enclosed by object
    virtual double area (void);
};

class Ellipse : public ClosedGraphics {
```

```
    Point _center;                              // Centre of the ellipse
    double _x_radius,_y_radius;                 // Radii of the ellipse
    double _angle;                              // Rotation from x-axis

  public:
    // Constructor
    Ellipse (Point &center,double x_radius,double y_radius,double angle=0) {
        _center=center;
        _x_radius=x_radius;
        _y_radius=y_radius;
        _angle=angle;
    }

    // Ellipse area--overwrites 'ClosedGraphics :: area'
    double area (void) {return PI * _x_radius * _y_radius;}

    // Distance to a point--expensive!
    virtual double dist (Point &);

    // Center
    const Point & center (void) {return _center;}

    // Translate--overwrites 'GraphicalObj :: translate'
    void translate (double x_offset,double y_offset) {
        _center.translate(x_offset,y_offset);
    }

    // Scale--overwrites 'GraphicalObj :: scale'
    void scale (double scale_factor) {
        _x_radius * =scale_factor;
        _y_radius * =scale_factor;
    }

    //...
};

class Circle : public Ellipse {
  public:
```

```
    // Constructor
    Circle ( Point &center,double radius) {
        Ellipse ( center,radius,radius);
    }

    // Distance to a point--overwrites 'Ellipse :: dist'
    virtual double dist (Point &p) {
        double center_dist=_center. dist (p);
        if (center_dist <=radius) return 0;
        else return center_dist-radius;
    }
    // ...
};
```

图 12.11 类 ClosedGraphics、Ellipse 和 Circle

12.3 C++中的对象声明语句应如何翻译成 C 语句，如图 12.11 程序中的

```
Point _center;
```

应翻译成什么？

12.4 表 12.1 给出了函数调用的翻译。但是这种翻译方式不能用于虚函数的情况。试说明 12.3.1 节中的虚函数调用 obj. scale (zoom_factor) 应该翻译成什么形式的 C 语句。

12.5 12.2 节在介绍类的翻译时有这么一句话："将 C++语言中一个类的所有非静态属性构成一个 C 语言的结构体类型，取类的名字作为结构体类型的名字"。在学完本章后，你认为这句话需要修改吗？

12.6 在面向对象语言中，编译器给每个对象分配空间的第一个域存放虚方法表的指针。是否可以把虚方法表指针作为最后一个域而不是第一个域？请简要说明理由。

12.7 在面向对象语言中，编译器给每个类建立虚方法表，如图 12.3 和图 12.4 那样。请简要说明，为什么编译器给每个类仅建立虚方法表，而不是建立所有方法的方法表。

12.8 12.2 节的结尾介绍，类 Point 的方法 translate 翻译成函数 translate__5Pointdd：

```
void translate__5Pointdd(Point &this,double x_offset ,double y_offset) {
    this. xc+=x_offset;this. yc+=y _offset;
}
```

其中假设 C 语言也有引用调用方式（形式参数前面加字符 &）。请按 C 语言实际只有值调用方式来考虑，写出方法 translate 翻译成的 C 函数，并说明调用点需要做的修改。

12.9 请按图 12.4 的方式给出 12.2 节中类 Point 的对象表示。

*第 13 章

函数式语言的编译

函数式程序设计语言起源于 LISP,因此可以追溯到 20 世纪 60 年代初期。到了 20 世纪 70 年代末期,当新的概念和实现方法出现时,这类语言才摆脱了 LISP 语言的统治,成为一类比较成熟的语言,其代表有 Miranda 和 ML 等。

在函数式语言中,函数是构造程序的基本成分,并且语言还提供一些机制用于构造更为复杂的函数。纯函数式语言禁止使用赋值语句,从而不会产生副作用,其优点是具有引用透明性,有助于程序的等式变换和推理。

函数式程序设计的主要任务在于定义(或构造)函数,以求解所提出的问题。所定义的作为主程序的函数可以包含一些辅助函数(可看作子程序)。计算机按照所定义函数的相应表达式,根据计算规则逐步计算,最后得到所需的结果。表达式中可能包含函数名,计算时可将其相应的定义作为归约规则。

本章介绍一种简单的函数式程序设计语言 SFP 及其实现。

13.1 函数式编程语言简介

本节介绍一种简单的函数式编程语言 SFP,用它来解释编译函数式编程语言的一些原理。本章的兴趣在于定义一个适当的抽象机和为该机器产生代码,而不在于像类型检查这样一些编译事务。例如,多态性概念是编译器的其他部分支持的,因而不把它放入 SFP。另外,也不考虑函数定义中包含分情形和模板的情况。

13.1.1 语言构造

现在描述 SFP 的语言构造。为此,假定 SFP 有一些基值类型和这些类型上的运算。这些基值类型是什么对下面的讨论并不重要,但它们必须包括布尔类型。另外,假定这些类型的每个值都只需占用抽象机的一个存储单元,以简化讨论。SFP 的表达式通过使用内部定义的算符和函数抽象及函数应用,可以直接从基值和变量归纳地构造。SFP 的语法论域总结在表 13.1 中,它

的语法总结的图 13.1 中。

表 13.1 SFP 的语法论域

元素名字	语法论域
b	B 基值集合,例如布尔值、整数、字符、…
op_{bin}	Op_{bin} 基值上的二元算符集合,例如+,-,=,≠,**and**,**or**,…
op_{un}	Op_{un} 基值上的一元算符集合,例如-,**not**,…
v	V 变量集合
e	E 表达式集合

对表 13.1 中的每个论域,给它一个"类型化"的名字,这样便于在图 13.1 中引用这些集合上的元素。图 13.1 中的 SFP 大部分构造是清楚的,但是函数抽象、函数应用和联立递归定义需作解释。

$e \to b \mid v \mid (op_{un}\ e) \mid (e_1\ op_{bin}\ e_2)$
$\quad \mid (\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3)$
$\quad \mid (e_1\ e_2)$ // 函数应用
$\quad \mid (\lambda v.\ e)$ // 函数抽象
$\quad \mid (\textbf{letrec}\ v_1 == e_1;$ // 联立递归定义
$\qquad v_2 == e_2;$
$\qquad \cdots$
$\qquad v_n == e_n$
$\quad \textbf{in}\ e_0)$

图 13.1 SFP 的语法

构造 $\lambda v.e$ 定义一个一元函数,其中 v 是形式参数,e 是定义该函数的表达式,即函数体。函数 $\lambda v.e$ 应用到表达式 e'写成$(\lambda v.e)e'$。根据该语言的语义和实现,它的效果是用表达式 e'(或它的值)代替形式参数 v 的所有自由出现,或者说把 v 约束到 e'(或 e'的值)。

为了少用括号,规定函数应用有最高优先级并且左结合;算术和逻辑算符有通常的优先级。在一个表达式中,λ 抽象选择最大可能的语法表达式作为 $\lambda v.e$ 的体 e,即 e 延伸到表达式的结尾或碰到第一个不能配对的右括号为止。

SFP 的语法允许函数定义和函数应用的嵌套。n 元函数可以用 $f == \lambda v_1.\ \lambda v_2.\ \cdots .\ \lambda v_n.\ e$ 来定义,一个函数可以用 $fe_1 \cdots e_m$的方式应用到若干个变元。为了书写的简便和执行的效率,通常把 n 元函数写成 $\lambda v_1 \cdots v_n.\ e$ 的形式,并把 $fe_1 \cdots e_m$实现为一次函数应用,而不是 m 次应用。

为了保证函数应用的语义十分清楚,有两点必须说明。第一点是参数传递机制,即对于e_1e_2,传给 e_1的是什么。第二点是在解释自由变量时,用静态约束还是动态约束,即第 6 章所讲的静态作用域还是动态作用域。

13.1.2 参数传递机制

在第6章介绍过值调用、换名调用和引用调用。在函数式语言中,后者没有意义,因为对变量只使用名字和值,不使用地址。在函数式语言中,对于表达式 e_1e_2,参数传递机制确定表达式 e_2本身还是它的值被传递。有以下三种可能性需要区分。

(1) **值调用**:先计算 e_2,然后把它的值传给 e_1。其优点是 e_2只计算一次。其缺点是即使 e_2的值不用,它也得计算;这时如果 e_2的计算不终止的话,它将是灾难性的。

(2) **换名调用**:从 e_1的计算开始,每当需要 e_2时,就计算它的值。于是 e_2以没有被计算的形式传到 e_1中相应形式参数出现的地方。其优点是 e_2只在真正被需要时才计算,因而它比值调用有较好的终止性。其缺点是 e_2可能被计算多次,而每次计算的值都一样。

(3) **按需调用**:又称**惰性计算**。从 e_1的计算开始,当第一次需要 e_2时,计算它的值,也就计算这一次。因此,第一次访问时引起 e_2的计算,其他的访问用第一次访问时计算的值。这种方式结合了前两种方式的优点。

在 SFP 中,对用户定义的函数用按需调用的方式传递参数。

虽然参数传递机制和约束规则并非完全独立,但是在讨论参数传递机制时,还是暂不考虑静态约束还是动态约束。

下面通过几个例子来熟悉 SFP 语言。

例 13.1 表达式

letrec $x==2$;
　　$f==\lambda y.\ x+y$;
　　$F==\lambda g\ x.\ g2$
in $F\ f\ 1$

如果是静态约束,f 体中的自由变量 x 引用定义 $x==2$,此时,$F\ f1$ 的值是4。若是动态约束,在 f 体中访问 x 的值之前把 x 约束到1,因此结果是3。 □

考虑静态约束的实现问题。在例13.1中,存在 x、f 和 F 的约束,特别是 x 约束到2。这样的一列约束通常叫做**环境**。为了使一个变元(相当于命令语言中的实参表达式概念)中的自由变量的正确环境在这些自由变量的每一个出现处都可用,该环境 u 和变元 e 需要一起传递,这样形成的二元组(e,u)叫做**闭包**。在闭包(e,u)中,环境 u 用来保证 e 中的自由变量会被正确地解释。(当然,在值调用场合,有时也需要形成闭包,主要是在有自由变量的函数作为参数传递的时候)

表达式 **letrec** $v_1==e_1$; … ; $v_n==e_n$ **in** e_0把 n 个新名字 $v_1,\cdots,v_n$填入环境,它们的作用域是 $G=e_0e_1\cdots e_n$。可以这样直观地解释,每当这些名字在这个作用域范围内碰到并且它们的值被需要时,定义这些名字的等式的右部就被使用。

SFP 虽然很简单,但仍可以被用来编程。

例 13.2 表达式

letrec $fac == \lambda n.$ **if** $n=0$ **then** 1 **else** $n * fac(n-1)$;

$fib == \lambda n.$ **if** $n=0$ **or** $n=1$ **then** 1 **else** $fib(n-1)+fib(n-2)$;

$one == \lambda n.\ 1$

in $fib((fac2)+one((fac2)-2))$ □

SFP 展示了函数式语言的一个重要特征,即高阶函数。函数可以作为函数的变元,也可以作为函数的结果。

例 13.3 表达式

letrec $F == \lambda x\ y.\ x\ y$; // 函数作为变元

$inc == \lambda x.\ x+1$

in $F\ inc\ 5$ // 值是 6

letrec $comp == \lambda f.\ \lambda g.\ \lambda x.\ f(gx)$; // 函数作为变元和结果

$F == \lambda x.\ \cdots$;

$G == \lambda z.\ \cdots$;

$h == comp\ F\ G$

in $h(\cdots)+F(\cdots)+G(\cdots)$ □

每一个 n 元函数可以作为一个高阶函数使用。当它作用于 m ($m< n$)个变元时,由于变元个数不足,其结果是一个($n-m$)元的函数。

13.1.3 变量的自由出现和约束出现

在命令语言中,变量和类型等的声明以及形式参数的声明引入新的名字。在 SFP 中,名字定义在 letrec 等式的左部和函数定义 $\lambda v_1 \cdots v_n.\ e$ 的 $\lambda v_1 \cdots v_n$ 中。把前者称为**等式定义的**名字,后者称为 **λ 定义的**名字。就函数式语言的语义和它的编译来考虑,在一个表达式中,一个(全局)变量的自由出现与它的定义性出现之间的联系是重要的。下面先归纳定义一个表达式 e 中自由变量的集合 $freevar(e)$,然后定义约束变量的集合。

定义 13.1 表达式中自由出现的变量集合。

$freevar(b) = \varnothing$ // 只由基值构成的表达式无自由变量

$freevar(v) = \{v\}$ // 只由一个变量构成的表达式含一个自由变量,即它本身

$freevar(op_{un}\ e) = freevar(e)$

$freevar(e_1\ op_{bin}\ e_2) = freevar(e_1) \cup freevar(e_2)$

$freevar($**if** e_1 **then** e_2 **else** $e_3) = freevar(e_1) \cup freevar(e_2) \cup freevar(e_3)$

$freevar(e_1\ e_2) = freevar(e_1) \cup freevar(e_2)$

$freevar(\lambda v_1 \cdots v_n.\ e) = freevar(e) - \{v_1, \cdots, v_n\}$

// e 中的自由变量 $v_1 \cdots v_n$ 在整个表达式中是受约束的

$freevar(\textbf{letrec}\ v_1 == e_1;\ \cdots\ ;\ v_n == e_n\ \textbf{in}\ e_0)$

$$= \bigcup_{i=0}^{n} freevar(e_i) - \{v_1, \cdots, v_n\} \qquad // \text{同上}$$

如果 $x \in freevar(e)$,就说 x 在 e 中有**自由出现**。 □

类似地,可以定义约束出现的变量集合。

定义 13.2 表达式中约束出现的变量集合。

$bdvar(b) = \varnothing$

$bdvar(v) = \varnothing$

$bdvar(op_{un} e) = bdvar(e)$

$bdvar(e_1\ op_{bin}\ e_2) = bdvar(e_1) \cup bdvar(e_2)$

$bdvar(\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) = bdvar(e_1) \cup bdvar(e_2) \cup bdvar(e_3)$

$bdvar(e_1\ e_2) = bdvar(e_1) \cup bdvar(e_2)$

$bdvar(\lambda v_1 \cdots v_n.\ e) = bdvar(e) \cup (\{v_1, \cdots, v_n\} \cap freevar(e))$

$bdvar(\textbf{letrec}\ v_1 == e_1;\ \cdots\ ;\ v_n == e_n\ \textbf{in}\ e_0)$

$$= \left(\bigcup_{i=0}^{n} bdvar(e_i)\right) \cup \left(\{v_1, \cdots, v_n\} \cap \bigcup_{i=0}^{n} freevar(e_i)\right)$$

如果 $x \in bdvar(e)$,就说 x 在 e 中有**约束出现**。 □

例 13.4 表达式

$e = (\lambda x\ y.\ (\lambda z.\ x+z)\ (y+z))\ x$

中的自由变量和约束变量分别为:

$freevar(e) = \{x, z\}$

$bdvar(e) = \{x, y, z\}$ □

可以看出,在一个表达式中,一个变量可以既有自由出现,又有约束出现。但是,一个变量的某个具体出现只能是自由的或约束的。

定义 13.3 自由出现和约束出现。

变量 x 的一个出现是**自由的**,如果这个出现既不是 $\lambda \cdots x \cdots.\ e$ 的项 e 的子项,也不是 **letrec** $v_1 == e_1;\ \cdots\ ;\ v_n == e_n$ **in** e_0 并且 $x = v_j, e = e_i (0 \le i \le n, 1 \le j \le n)$ 的项 e 的子项。否则 x 的这个出现称为是**约束的**。 □

在例 13.4 中,$x+z$ 中的 z 是约束的,而 $y+z$ 中的 z 是自由的。

在 SFP 中采用**静态约束**,换句话说,在一个表达式或子表达式中,变量的自由出现联系到该变量的第一个外围 **letrec** 定义或 λ 抽象(即命令式语言的静态作用域)。

13.2 函数式语言的编译简介

13.3 节将介绍一种抽象机 FAM,它的机器语言是 SFP 语言的目标语言。SFP 程序,更一般

地说,用按需调用语义和静态约束的函数式语言的程序可以编译成 FAM 的机器语言。该机器有一个栈,生存期符合栈式管理的所有变量都存在栈中;它还有一个堆,所有其他的变量都存在堆中。

本节首先用一连串的例子来启发后面从 SFP 程序到 FAM 程序的编译描述,然后讨论环境和约束。

13.2.1 几个受启发的例子

先从简单的例子开始,逐步引入到较复杂的例子。一个 SFP 程序最外的表达式叫做程序表达式,换句话说,该表达式的计算给出程序的结果。

例 13.5 程序表达式是

$e=1+2$

希望 e 的编译结果被执行时,它把 e 的值留在 FAM 上可访问的存储空间(栈或堆)中。基值类型的结果是可以保留在栈上的,但是,如果结果是一个函数,它将没有办法存放在栈上。当希望所有的程序表达式被执行时都以同样的方式表达结果,独立于它们的结果类型时,可以把程序表达式的结果统一存放在堆中,在栈顶用一个指针指向堆中的结果。 □

例 13.6 有表达式

letrec $x==1/y$; $y==0$; $z==x$ **in** $1+2$

该程序表达式必须这样编译,其结果 3 存放在堆中,并且栈顶的指针指向堆中这个位置。该表达式包含变量。在函数式语言中,变量代表值。就实现的效率而言,快速访问这些值是重要的。在编译(而不是解释)实现中,尽可能将可直接访问的存储单元分配给这些变量。在 SFP 实现中,由 **letrec** 或函数抽象引入的变量将在 FAM 的栈上分配单元。

x、y 和 z 的等式应该这样编译:产生的指令序列并不直接计算这些等式的右部(将来真正需要这些值的时候再计算)。另一方面,x、y 和 z 分别约束到这些右部的信息必须在将来需要 x、y 或 z 的时候是可用的。于是,生成的指令序列必须构造 x、y 和 z 的闭包,并分别将指向这三个闭包的三个指针存放在栈中。因为表达式 1+2 的计算不需要访问 x、y 或 z 的值,因此这三个闭包根本不计算。

再引入两点改进。y 的等式无须构造闭包,因为它的右部不含自由变量,只要让指针指向一个对象,该对象由值 0 和标明该对象是基值的标记组成。另一个改进是让 z 和 x 约束到同一个闭包。 □

可以看出,上下文清楚地决定了一个表达式应该怎样编译。例 13.6 中的表达式 1+2 必须这样编译:生成的代码产生它的值;而表达式 $1/y$ 的代码必须为它产生一个闭包。

例 13.7 表达式

if (**if** $1\neq2$ **then true else false**) **then** 3 **else** 4

如果仍希望结果在堆中,那么表达式 3 和 4 都必须按这种方式编译,即它们的值都应在堆

中。但是对于条件表达式(**if** 1≠2 **then true else false**),FAM 的假转指令 **jfalse** 希望在栈顶测试它的值,因此,该条件表达式是这样编译的,即生成的指令序列把它的值留在栈上。同样,表达式 **true** 和 **false** 也必须这样编译,即它们的值在栈顶。 □

例 13.8 表达式

letrec $f == \lambda y\ z.$ **if** $z=0$ **then** 1 **else** $1/y$;

$x == 5$

in $f\ 1\ (x+1)$

这个程序可用来研究两个问题:关于函数变元的闭包的构造和以函数为值的表达式的构造。先从后者开始。从前面的例子知道,为联立等式产生的代码必须为等式的右部形成闭包,并为左部存储一个指向该闭包的指针。像 $x==5$ 这类等式的优化已经碰到过了,现在来考虑右部是函数定义的情况。当为 $\lambda y\ z.$ **if** $z=0$ **then** 1 **else** $1/y$ 构造闭包时,把它进一步做成 FUNVAL 对象,因为 f 的函数应用代码的第一步必须保证 f 是可应用的函数。FUNVAL 对象和该闭包的区别仅在于前者还包含存放变元指针的存储空间。构造闭包并随后转变成 FUNVAL 对象这两步是由该等式的代码直接完成的。

怎样编译 f 的变元 1 和 $x+1$? FAM 的按需调用语义使得变元必须以闭包的形式传递,这样才可能保证它们的值仅在需要时计算。因此,必须为 f 的变元 $x+1$ 产生闭包。它包含一个指令序列和一个约束向量,该指令序列的执行给出该变元的值,该向量包含一些指针,它们指向出现在该变元中的自由变量的值(或值的表示)。现在的这个向量仅含指向 x 的闭包的指针。f 的等式和 x 的等式的指令序列是在构造 $x+1$ 闭包之前执行的,得到关于 f 的 FUNVAL 对象和对应 x 的基对象,这两个堆对象的指针在栈中,它们被赋给 f 和 x。当构造 $x+1$ 的闭包时,x 的闭包的指针可以从栈中复制到该闭包的约束向量中。对于 f 的变元 1,由前面知道,不必创建一个闭包,因为它不含自由变量。 □

例 13.9 表达式

letrec $x == 2+1$;

$f == \lambda a\ b.\ g\ a+h\ b$;

$g == \lambda x.\ \cdots$

$h == \lambda y.\ \cdots$

in $fx\ x$

该程序表明,SFP 程序的(以闭包或值形式的)表达式的指针可以复制任意多份,作为函数变元和全局变量的值等。因此在 SFP 的实现中,总是值的指针和闭包的指针而不是它们的本身在传递,并且将它们存于约束向量和栈帧(相当于第 6 章的活动记录)中。因为复制表达式的指针而不是表达式(更确切说是它的闭包)本身,因此每个表达式只有一个实例存在,在 letrec 定义中该表达式对应的变量的所有出现都有一个指针指向该实例。表达式对应变量的首次使用引起该表达式闭包的计算,该计算是面向该变量的所有出现的。以后的出现直接访问这个值而无须重新计算。 □

例 13.10 表达式

letrec f == **letrec** x == 2
　　　in $\lambda y.\ x+y$
in f 5

该程序可用来说明命令式语言和函数式语言在局部变量生存期上的区别。在 Pascal 语言中,除了那些由 new 过程在堆上创建的对象外,所有在过程激活时建立的对象都有相同的生存期,它们在过程终止时消失。

对于 SFP 函数,情况不再是这样。在上面的程序中,为了把 f 作用于 5,需要计算由最内的 letrec 构造出的函数。x 在这里局部于这个 letrec。若最内的这个 letrec 已经计算,栈式管理会忘掉属于这个 letrec 的一切东西,包括局部变量。这个例子说明高阶函数的出现需要延长局部变量的生存期,这意味着在该例中,局部变量必须组装到一个 FUNVAL 对象中,存活在堆上。 □

13.2.2 编译函数

13.2.1 节的例子已清楚地表明,同一个表达式在不同的上下文中会编译成不同的指令序列。在前面的例子中遇到了四种上下文,分别用不同的字母表示它们:P(program)、B(basic)、V(value)和 C(closure)。

(1) P:编译完整的程序表达式。结果在堆中,栈顶有一指针指向它。它总是最外上下文,编译是在这种上下文中开始的。

(2) B:结果必须是基值并且存在栈上。例如,处理条件表达式中的条件时,这种上下文会出现。

(3) V:结果在堆中,栈顶有一指针指向它。这是计算的正常情况。但是对于基值类型的表达式,作为结果的基值先放在栈上,然后将它做成一个堆对象。

(4) C:结果必须是被编译的表达式的闭包。函数的变元和递归等式的右部总是这种情况。

以上四种上下文对应四个编译函数:P_code、B_code、V_code 和 C_code,这些函数为相应上下文中的表达式产生指令序列。

13.2.3 环境与约束

SFP 有两类名字定义:由 λ 定义的名字,它出现在一个 λ 抽象的名字序列中;由等式定义的名字,它出现在 letrec 表达式的一个等式的左部。从例 13.5 到例 13.10 可知,一个名字的定义性出现总是关联到一个闭包。当一个等式被编译时,其左部的名字总是关联到其右部的闭包;而 λ 抽象中的约束名字是在函数应用时关联到该次应用的变元的闭包。

名字的引用性出现应该这样编译:它们获得相关联的定义性出现的值。SFP 按需调用语义是这样实现的:第一次碰到一个名字的引用性出现时,其对应的定义性出现的闭包被计算并且该

闭包被计算结果覆盖,以后再碰到该名字的引用性出现时就直接取这个值。FAM 的指令 **eval** 可处理这两种情况。

SFP 的编译器遵循通理,它使用编译时静态可用的信息去有效地管理运行时的动态对象。来看一下 SFP 程序的哪些信息是静态的,因而在编译时可用。表 13.2 列出从程序中可读到的关于函数的静态信息(称为原始静态信息)和从它们可以导出的信息。其中前两种情况用来确定在函数应用时创建的栈帧中的寻址,最后一种情况的信息允许在约束向量中相对寻址。

表 13.2 原始信息和导出信息

原始信息	导出信息
变元个数	形参地址(相对于第一个形参的地址)
在函数体中任何一点都可访问的由等式定义的局部名字的集合	这组等式定义的局部名字的地址(相对于第一个等式定义的局部名字的地址)
函数体中全局名字的集合	全局名字在约束向量中的下标

于是,对于 SFP 程序中变量的引用性出现,编译器知道它对于直接围绕它的 letrec 或函数定义是局部(约束)的还是全局(自由)的,并且知道哪个相对地址或约束向量中的哪个下标指派给它。这种静态信息包含在对应的**编译环境**中(见表 13.3)。

表 13.3 位置和编译环境

名字	论域	备注
p	$P=\{\mathrm{LOC},\mathrm{GLOB}\}\times integer$	位置(相对地址或下标)
β	$B=(V\rightarrow P)$	编译环境

在 Pascal 编译器中,地址环境中包含变量的相对地址和它的嵌套深度。在 SFP 编译器中,对应每个变量,编译环境包含它的位置和有关它是自由变量还是约束变量的信息。

当 SFP 程序被编译时,什么时候编译环境会改变?很显然,当函数抽象的体或 letrec 中的表达式开始编译时,新引入的局部变量必须加入编译环境。

当生成构造闭包或 FUNVAL 对象的代码时,必须把这时的全局变量构造成它们的**运行环境**。

表达式中的局部变量在 FAM 栈帧上分配存储单元,使用栈帧的相对地址。全局变量存储单元的指针分配在约束向量中,依据下标可以找到这些指针。

13.3 抽象机的体系结构

从本节开始,逐步描述 FAM 的体系结构和指令集合,以及把 SFP 编译到 FAM 的编译方案。

FAM 的存储器包含一个程序存储区 PS,每个单元含一条 FAM 指令。某些指令含一个运算对象。程序计数器 PC 总是保留当前指令的地址。在正常情况下,FAM 重复执行下列三步:

(1) 取当前指令;

(2) PC 增 1;

(3) 解释当前指令。

当碰到 **stop** 指令或发生错误时,FAM 停机。

另外,存储器中还包括一个栈 ST 和一个堆 HP,它们的空间不受限制。还假定 FAM 有一个堆管理器,它能在执行 new 过程时分配空间,并能自动回收不再使用的空间。

13.3.1 抽象机的栈

FAM 的栈和第 6 章介绍的活动记录栈类似,但 FAM 栈向下增长。SP 是栈顶指针寄存器,它指示已被占用的最高地址。下列四种对象都只占栈中一个单元:

(1) 基值,如整数、布尔值等;

(2) 栈地址;

(3) 堆地址;

(4) 程序指令的地址。

栈被分成若干帧,栈帧在函数应用和计算闭包时建立。前者与 Pascal 在过程激活时建立活动记录是一致的。后者在 Pascal 中无对应的概念,它是按需调用语义要求延迟变元计算引起的,稍后会讨论。图 13.2 显示出了这两种情况的栈帧结构。

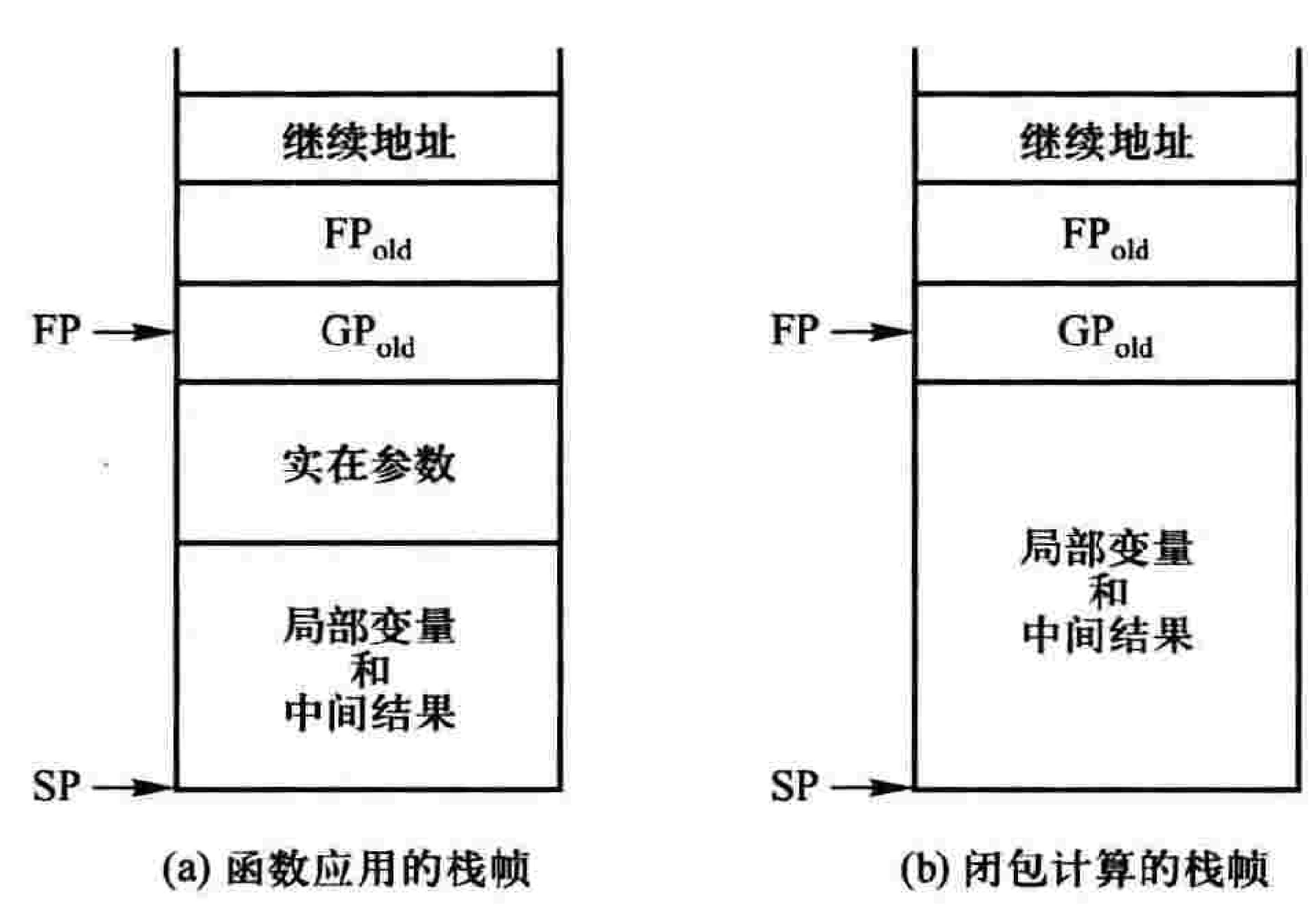


图 13.2 栈帧结构

闭包计算时没有实在参数单元(指向实在参数的指针单元),除此以外,两种情况下的栈帧有同样的结构。栈帧指针寄存器 FP 指示三个有序单元中地址最高的那一个。第一个单元是当

前活动(函数应用或闭包计算)结束后继续执行的指令地址,第二个单元是老 FP 的值,第三个单元是所有全局变量构成的向量的指针 GP。FP 和 GP 分别相当于第 6 章所介绍的动态链和静态链,但还是有区别的。在第 6 章中 GP 是链的开始,通过它可以找到所有的非局部变量,在这儿 GP 指向一个向量,该向量包含指向一个函数或表达式所有自由变量值的指针。

这种栈帧结构暗示了建立和释放栈帧可用一组固定的指令,见图 13.3,它们可以集成在 FAM 的 **mark** 和 **eval** 或 **return** 指令中。

ST[SP+1]=继续地址;	PC=ST[FP-2];
ST[SP+2]=FP;	GP=ST[FP];
ST[SP+3]=GP;	SP=FP-2;
SP=SP+3;	FP=ST[FP-1];
FP=SP;	
(a) 建立栈帧	**(b) 释放栈帧**

图 13.3 栈帧的建立与释放指令

13.3.2 抽象机的堆

FAM 的堆所存储的是其生存期和栈管理方式不相容的对象。堆对象有一个标记,用于指示对象的性质。四种标记 BASIC、FUNVAL、CLOSURE 和 VECTOR 分别表示下列对象。

(1) BASIC:存放基值的单元 b。

(2) FUNVAL:该对象表示一个函数值。它是一个三元组(cf,fap,fgp),其中指针 cf 指向程序区中函数体开始的地方,指针 fap 指向函数变元向量,指针 fgp 是函数各全局变量值的指针所组成的向量的指针。这两个向量也存在堆中。FUNVAL 对象是在函数定义的翻译结果被处理时构造的。在这里,变元向量当然是空的,因为现在还没有变元。当 n 元函数应用于 m($m<n$)个变元时,FUNVAL 对象的变元向量就变成非空。如前面所提到的,变元不足的应用结果仍然是函数,它仍由 FUNVAL 对象表示,该对象是在把这些变元装进先前的 FUNVAL 对象后得到的。

(3) CLOSURE:该对象是一个闭包。它表示一个延迟的计算,因为按需调用的参数传递是把变元的代码及其全局变量的值组成一个闭包,并且仅在需要时才计算闭包。该对象有两个成分,即代码的指针 cp 和全局变量值的指针向量的指针 gp,后面将会看到,变量的所有定义性出现都被赋了一个闭包。

(4) VECTOR:该对象是堆对象指针的向量。该向量存放函数变元的指针,或者存放 FUNVAL对象或 CLOSURE 对象的全局变量的指针。

标记选择子 tag 用来确定堆对象的标记。函数 size 用来确定 VECTOR 对象的向量长度。

某些 FAM 的指令根据标记来解释堆对象的内容,当它不能作用于堆对象(由于对象类型不对)时会报告错误。有些指令从最高栈单元的内容构造这样的对象,并且把新创建的这个堆对象的指针放在栈顶。这些指令列在表 13.4 中。

表 13.4 建立堆对象的指令

指令	含义	备注
mkbasic	ST[SP]=new(BASIC: ST[SP])	建立基本堆对象
mkfunval	ST[SP-2]=new(FUNVAL: ST[SP],ST[SP-1],ST[SP-2]); SP=SP-2	建立函数堆对象
mkclos	ST[SP-1]=new(CLOSURE: ST[SP],ST[SP-1]); SP=SP-1	建立闭包
mkvec *n*	ST[SP-*n*+1]=new(VECTOR: ST[SP-*n*+1], ST[SP-*n*+2],…,ST[SP]); SP=SP-*n*+1	建立有 *n* 个分量的向量
alloc	SP=SP+1; ST[SP]=new(CLOSURE: NIL,NIL)	建立空闭包

13.3.3 名字的寻址

对于 SFP 来说,栈管理和名字的寻址不同于 Pascal 等语言。函数定义的编译必须考虑函数应用允许变元不足和变元过剩的情况。当编译函数应用 $e\ e_1 \cdots e_m$时,编译器并非总是能知道被应用函数的变元个数。例如,e 可以是外围高阶函数的一个形参。

考虑编译函数应用 $e\ e_1 \cdots e_m$的可能方法。很清楚,为函数应用产生的指令序列必须产生 m 个闭包和一个 FUNVAL 对象。它们的指针必须放在这个函数应用的栈帧中,这些指针在栈帧中有两种安排方式,如图 13.4 所示。

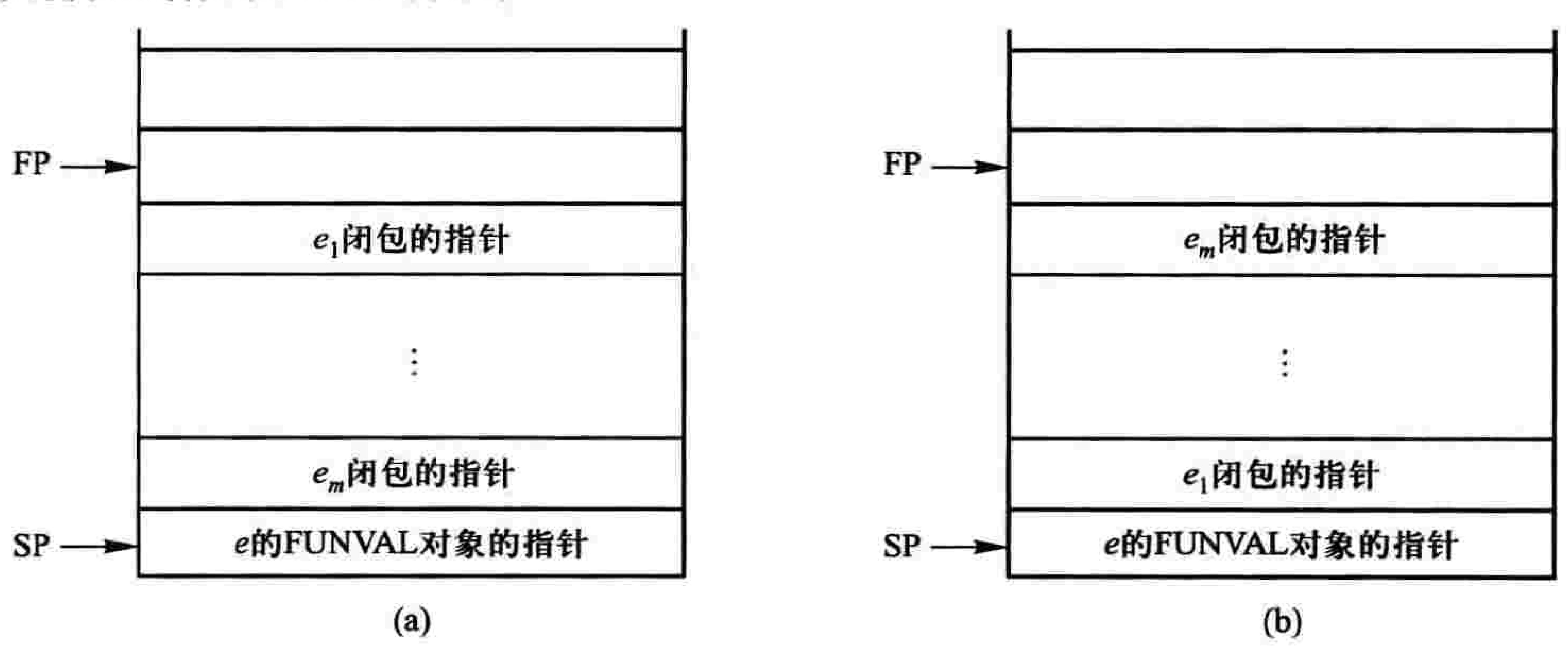


图 13.4 变元在栈帧中的可能安排

现在讨论这两种情况。图 13.4(a)的存储安排允许变元和形式参数的寻址相对于 FP 的内容。但是,因为编译函数定义时 m 的值是不知道的,而局部变量在栈帧中是邻近这些变元的,因此局部变量的寻址就有困难。此外,如果函数应用提供过多的变元时,也会遇到类似的问题。

图 13.4(b)的存储安排不允许相对于 FP 的内容作为形式参数寻址。但是形式参数和局部变量可以相对于 SP 的内容寻址。SP 在建立新的局部变量和中间结果时是变化的,因此选择某个动态地址作为形式参数和局部变量相对寻址的基地址。这个地址可选择为正好在 e_1 闭包的指针单元的下面,把它叫做 sp_0。这样,函数 $\lambda v_1 \cdots v_n. e$ 的形式参数 $v_1, \cdots, v_n$ 的相对地址必定是 $-1, -2, \cdots, -n$,而局部变量的地址必定是 $0, 1, 2, \cdots, m$。

下面的情况出现在一个函数体的处理的开始(由后面描述的 **apply** 指令产生)。PC 寄存器置到函数体指令区的开始位置,指针 GP 指向恰当的全局约束。栈包含了三个有序的单元,它们是继续地址、老的 FP 和老的 GP,还包含 $e_m, \cdots, e_1$ 的闭包的指针单元。SP 指向 e_1 的闭包的指针单元。e_1 的闭包的指针之上的地址是 sp_0,如图 13.5 所示。

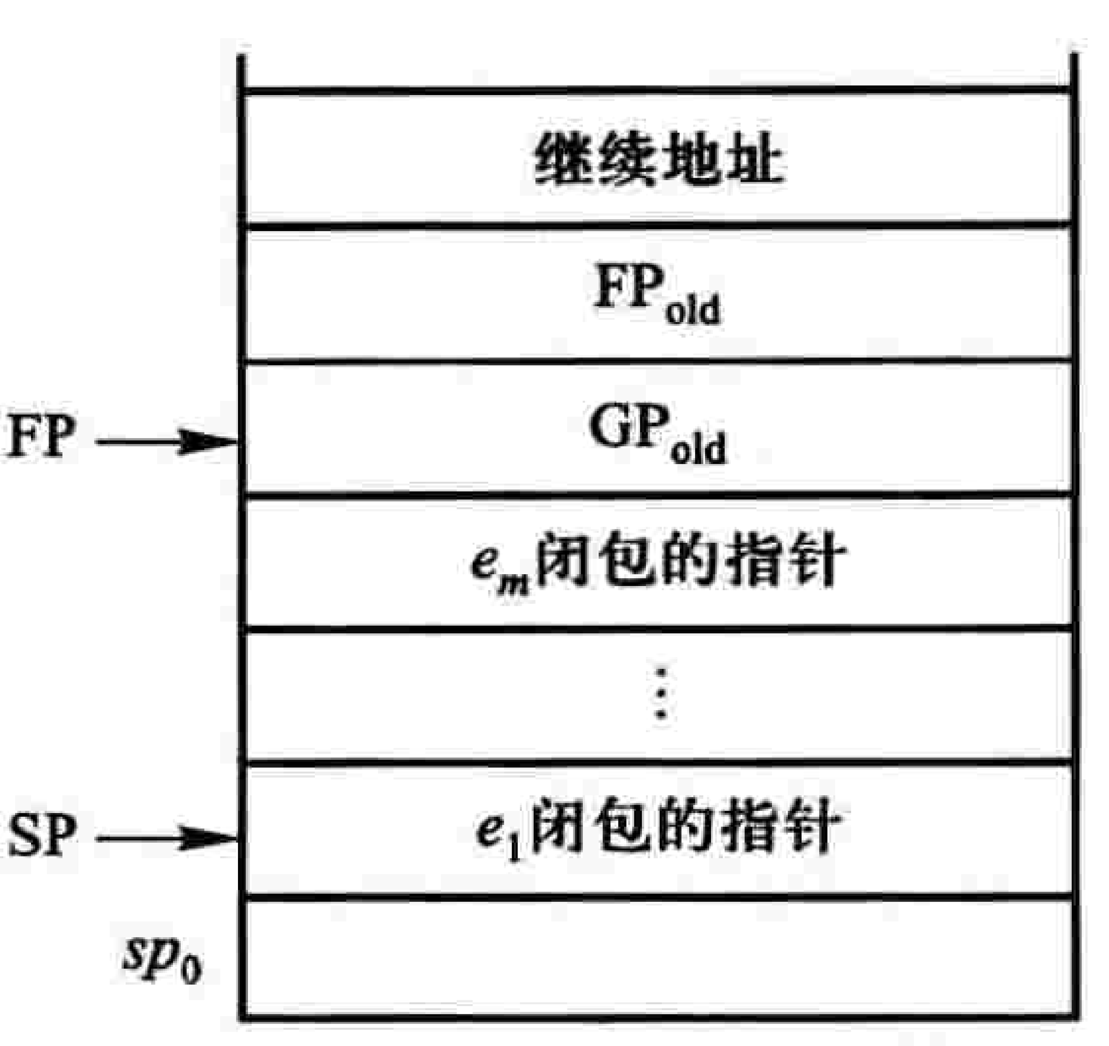


图 13.5 处理函数体之前的栈布局

如果处理函数体,由 letrec 表达式引入的局部变量必须在栈上分配空间,SP 的值相应地增加。还好,SP 的当前值和 sp_0 的值之间的差,对于函数体每一点来说是静态可确定的,因为已出现的新局部变量和中间结果的数目是已知的。这个差值在编译时保存在参数 sl 中,传给前面提到的四个编译函数。换句话说,如果编译到函数体中的一个点 a,sl 的当前值为 sl_a,运行时执行到这一点的 SP 值为 sp_a,那么下列地址关系式成立:

$$sp_a = sp_0 + sl_a - 1 \tag{13.1}$$

因此,生成的指令可以使用编译时确定的值 sl_a 和运行时 SP 的值 sp_a 来计算运行时的值 sp_0。形式参数可以用负的相对地址寻址,而局部变量可以用非负的相对地址寻址。

13.3.4 约束的建立

前面已讨论过,对于每个函数定义以及每个表达式,自由变量集合都是静态可知的。这个集合的元素可按任意次序排列,其位置便可作为自由变量的相对地址。这些相对地址在运行时用于访问自由(全局)变量的值。这些值的地址存放在一个向量中,该向量的指针存放在堆中作为 FUNVAL 或 CLOSURE 对象的一部分;在计算闭包或函数应用时,该指针复写到 GP,即运算时可以通过 GP 去寻找该向量的元素。到目前为止,还不清楚这些全局变量的值的指针是怎样进入这样的向量的。但有一点是清楚的,当整个程序表达式被编译时,还有被编译的程序开始执行时,自由变量集合和对应的值向量都为空。

假定函数对象或闭包被构造。在这两种情况下,全局变量的值的指针向量必须和它们一起组装。因为 SFP 已规定为静态约束,所以,刚处理的外围函数的形式参数、局部变量和全局变量

都应该看成该函数定义或表达式的全局变量。在 13.3.3 节已知形式参数和局部变量在运行时可正确寻址。如果归纳地假定能通过 GP 访问当前的全局变量,那么就可复制所有新的全局变量的值的指针到栈中,形成一个向量,并把它作为一个堆对象的一部分。

13.4 指令集和编译

通过对问题的分析和直观介绍,现在可以一步步地描述编译和所需要的 FAM 指令。前面已提到过,共有四个编译函数 P_code、B_code、V_code 和 C_code,它们与所期望的生成代码执行结果的性质相对应。这些 code 函数有三个参数:被编译的表达式 e,变量环境 β 和栈标高 sl。前面已经提到过,sl 定义了被生成的代码执行前 SP 寄存器的值和地址 sp_0 的差,形式参数和局部变量可相对于 sp_0 寻址。

13.4.1 表达式

SFP 程序表达式 e 的编译总是从 P_code 函数的一个应用开始:

P_code e = V_code e [ ] 0;
　　　　　　stop

因为 e 不能含任何自由变量,所以环境为空。因为栈未作任何填充,所以栈标高 sl 是 0。**stop** 指令停止机器的执行。

现在编译简单表达式,即仅由基值、算符和 **if** 构成的表达式。考虑需要一个基值作为执行结果的编译,编译函数 B_code 适用于这种情况。

由 B_code 生成的指令列在表 13.5。假定对 SFP 的每个一元运算符 op_{un} 和二元运算符 op_{bin},在 FAM 中都有对应的机器指令 $\boldsymbol{op}_{un}$ 和 $\boldsymbol{op}_{bin}$,于是

B_code b β sl = **ldb** b
B_code (e_1 op_{bin} e_2) β sl = B_code e_1 β sl;
　　　　　　　　　　　　　B_code e_2 β $sl+1$;
　　　　　　　　　　　　　$\mathbf{op}_{bin}$
B_code (op_{un} e) β sl = B_code e β sl;
　　　　　　　　　　　$\mathbf{op}_{un}$
B_code (**if** e_1 **then** e_2 **else** e_3) β sl = B_code e_1 β sl;
　　　　　　　　　　　　　　　　false l_1;
　　　　　　　　　　　　　　　　B_code e_2 β sl;
　　　　　　　　　　　　　　　　ujmp l_2;
　　　　　　　　　　　　　　　l_1: B_code e_3 β sl;

l_2:

B_codee β sl = V_code e β sl ;

getbasic

表 13.5 基值、运算和分支指令

指令	含义	备注
ldb b	SP = SP+1; ST[SP] = b	装入基值
getbasic	**if** HP[ST[SP]].$tag \neq$ BASIC **then** $error$ **fi** ; ST[SP] = HP[ST[SP]].b	从堆中往栈上 装入基值
$\mathbf{op}_{bin}$	ST[SP−1] = ST[SP−1] op_{bin} ST[SP]; SP = SP−1	二元运算
$\mathbf{op}_{un}$	ST[SP] = op_{un} ST[SP]	一元运算
false l	**if** ST[SP] = $false$ **then** PC = l fi; SP = SP−1	条件分支
ujmp l	PC = l	无条件分支
ldl l	SP = SP+1; ST[SP] = l	把标号放入栈中

函数 V_code 的对应情况与上面类似,增加了把作为结果的基值放入堆中并用栈顶指针指向它的指令:

V_code b β sl = B_code e β sl;

mkbasic

V_code(e_1 op_{bin} e_2) β sl = B_code (e_1 op_{bin} e_2) β sl;

mkbasic

V_code (op_{un} e) β sl = B_code (op_{un} e) β sl;

mkbasic

V_code (**if** e_1 **then** e_2 **else** e_3) β sl = B_code e_1 β sl;

false l_1;

V_code e_2 β sl;

ujmp l_2;

l_1: V_code e_3 β sl;

l_2:

注意上面条件表达式的编译,e_1是用 B_code 函数编译的,而 e_2和 e_3是用 V_code 函数编译的,因为 e_1的值需要放在栈上,而 e_2和 e_3的值需要放到堆中。

13.4.2 变量的引用性出现

当上下文为 V 时,变量的引用性出现必须编译成访问其值,而当上下文为 C 时,变量的引用性出现必须编译成访问其闭包。在前一种情况下,如果值还没有计算,必须首先从一个闭包计算这个值。

V_code v β sl = getvar v β sl;
　　　　eval
C_code v β sl = getvar v β sl

必须深入考虑代码生成函数 getvar:

getvar v β sl = **let** $(p, i) = \beta(v)$
　　in if p = LOC **then pushloc** $sl-i$
　　　　else pushglob i
　　fi

它为局部变量和形式参数产生 **pushloc** 指令,为全局变量产生 **pushglob** 指令。这些指令的定义在表 13.6 给出。

表 13.6　变量值的入栈指令

指令	含义	备注
pushloc j	SP = SP+1; ST[SP] = ST[SP−j]	把形式参数或局部变量的值的指针压入栈
pushglob j	SP = SP+1; ST[SP] = HP[GP]. $v[j]$	把全局变量的值的指针压入栈

现在考虑对于变量 v 的调用 getvar v β sl_a,其中 v 是形式参数或局部变量。环境 β 把它约束到二元组(LOC,i),其中 i 是非负数(局部变量)或负数(形式参数)。于是 getvar 生成一条指令 **pushloc** sl_a-i。

假定在指令 **pushloc** 执行前地址关系式(13.1)对于参数 sl_a 和 SP 的状态 sp_a 成立,即 $sp_a = sp_0+sl_a-1$。执行 **pushloc** sl_a-i 的效果是:

$$ST[sp] = ST[sp-(sl_a-i)]$$

其中 $sp = sp_a+1$,因为 sp 在存储访问前增 1。但是仍然有

$$sp-(sl_a-i) = sp_a+1-sl_a+i = (sp_0+sl_a-1)+1-sl_a+i = sp_0+i$$

于是形式参数和局部变量都能正确地装入。

就访问全局变量而言,假定编译时的环境 β 为一个向量中的所有全局变量定义一个下标,并

假定在运行时寄存器 GP 指向一个向量,该向量填充了根据这种下标定义安排的这些全局变量的指针。于是 **pushglob** 指令的效果就是从该向量中把一个全局变量的指针压入栈。

13.4.3 函数定义

函数定义可以在两种上下文(值上下文 V 和闭包上下文 C)中编译。前面已介绍过,生成的代码应该构造闭包,但是为了提高函数应用的效率,生成的代码立即建立 FUNVAL 对象。前面已经说过,这样的对象有三个成分:函数代码的起始地址,变元指针向量(初始时为空)的指针和约束向量的指针。这些指针必须在这个定义点赋值。给全局变量所置的相对地址,加上形式参数的相对地址,就构成了开始编译函数体时的环境:

$V_code\ (\lambda v_1 \cdots v_n . e)\ \beta\ sl = C_code\ (\lambda v_1 \cdots v_n . e)\ \beta\ sl$

$C_code\ (\lambda v_1 \cdots v_n . e)\ \beta\ sl =$

```
        pushfree fr β sl;        // 复制全局变量的值的指针
        mkvec g;
        mkvec 0;                 // 空的变元向量
        ldl l1;                  // 函数代码的地址
        mkfunval;
        ujmp l2;
    l1: targ n;                  // 测试变元个数
        V_code e ([vi↦(LOC,-i)]_{i=1}^n [v'j↦(GLOB,j)]_{j=1}^n) 0;
        return n;
    l2:
```

其中,

$fr = [v_1', \cdots, v_g'] = \text{list}(\text{freevar}\ (\lambda v_1 \cdots v_n . e))$

$\text{pushfree}\ [v_1, \cdots, v_g]\ \beta\ sl = \text{getvar}\ v_1\ \beta\ sl;$

$\text{getvar}\ v_2\ \beta\ (sl+1);$

$\cdots$

$\text{getvar}\ v_g\ \beta\ (sl+g-1)$

需要对上述代码作些解释。总的来说,由 C_code 函数生成的指令序列用于构造一个 FUNVAL对象。但与此同时,C_code 函数也必须编译函数定义,产生函数的代码。在上面的编译方案中,从标号 l_1到 **return** n 部分是函数的代码。其余部分用于生成 FUNVAL 对象,并把该函数代码的起始地址 l_1放到这个 FUNVAL 对象中。**ujmp** 指令用于跳过这段函数代码。

必须注意函数全局变量的处理。它们是静态可知的,这些变量的集合是用函数 freevar 构

造。函数 list 从这个集合构造它的(无重复的)成员表。全局变量的次序也决定了它们值的指针的次序。pushfree $[v_1, \cdots, v_g]\ \beta\ sl$ 生成建立该向量的指令序列。它用 getvar 把变量 $v_1, \cdots, v_g$ 的值的指针压入栈,getvar 用环境 β 中的信息进行寻址。函数体的编译开始时,参数 sl 的值为 0,在函数体的代码开始执行前,栈的布局如图 13.5 所示。于是,地址关系式(13.1)在函数体执行前保持,因为有

$$sp_a = sp_0 + 0 - 1$$

因为每个 getvar 产生一条指令,它的执行使 SP 增 1,因此 getvar 的参数 sl 也增 1,模拟运行时的 SP 增 1,这就保证了地址关系式(13 .1)在执行时成立,因此局部变量和形式参数可以正确地寻址。

13.4.4 函数应用

为函数应用所生成的指令序列必须保证:当进入函数执行时,栈的布局必须保持,该布局是编译函数定义时所假定的。在进入前的栈布局由图 13.5 说明。

V_code$(e\ e_1 \cdots e_m)\ \beta\ sl$ = **mark** l;
$e \neq e'e''$ C_code $e_m\ \beta\ (sl+3)$;
…
C_code $e_1\ \beta\ (sl+m+2)$;
V_code $e\ \beta\ (sl+m+3)$;
apply;
l :

mark l 为这个应用建立一个新的栈帧,继续地址 l、FP 与 GP 的当前值都被保留。然后,生成的指令序列为变元在堆上建立闭包,并把闭包的指针压入栈中。注意,**mark** 的执行使 SP 加 3。如果地址关系式(13.1)在编译和在执行的开始分别都保持,那么它在 e_m 编译前和相应的代码执行前也都保持。因为每个闭包的指针都需要一个存储单元,因此编译每个 $e_i(m \geq i \geq 1)$ 时都让参数 sl 加 1,保证了地址关系式(13.1)在该编译方案自始至终都保持。表 13.7 是 **mark** 指令的定义。**mark** 指令的效果由图 13.6 说明。

表 13.7 建立一个栈帧

指令	含义	备注
mark l	ST[SP+1] = l; ST[SP+2] = FP; ST[SP+3] = GP; SP = SP+3; FP = SP	建立栈帧,保存有关地址

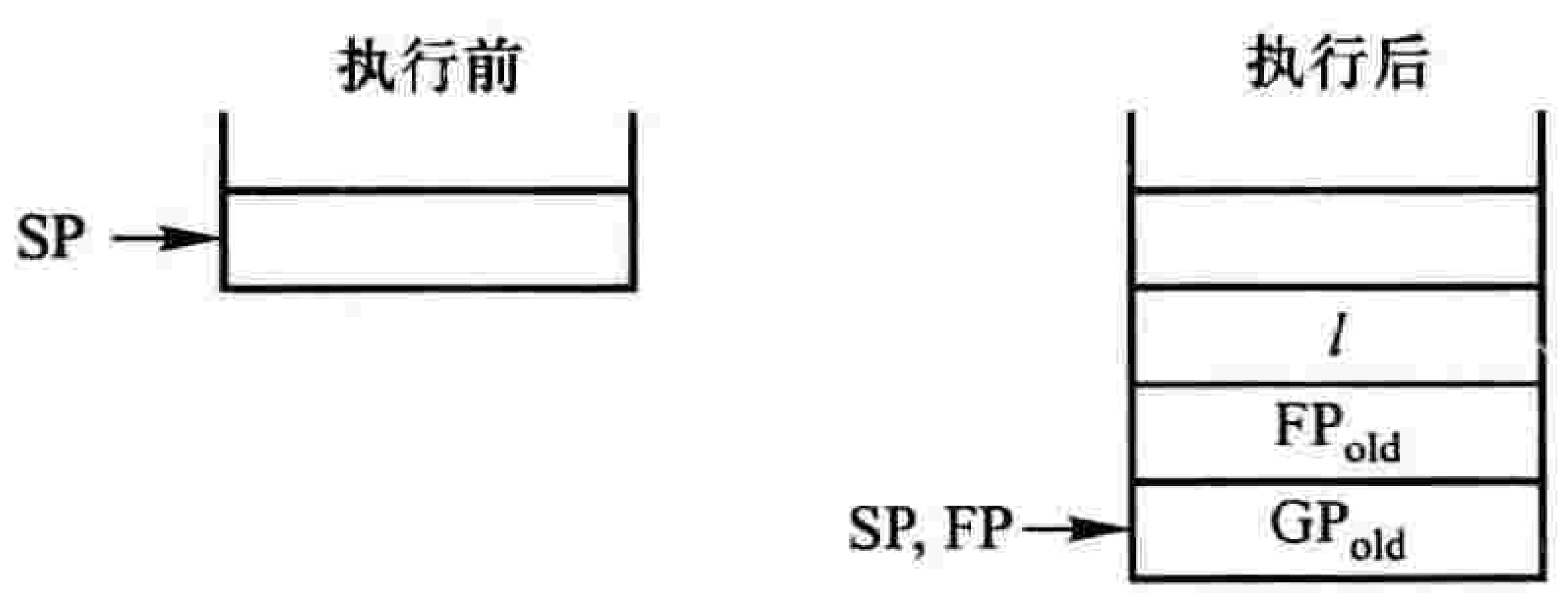


图 13.6 **mark** 指令执行前后情况

表 13.8 列出了 **apply** 指令的定义。在变元都从 FUNVAL 对象装入到栈中并且约束指针置好后,它跳到函数体指令的开始点。它执行后栈的布局如图 13.7 所示。

表 13.8 函 数 应 用

指令	含义	备注
apply	**if** HP[ST[SP]].*tag* ≠FUNVAL **then** error **fi**; **let**(FUNVAL: *cf*, *fap*, *fgp*) = HP[ST[SP]] **in** PC = *cf* ; GP = *fgp*; SP = SP- 1; **for** *i* =1 **to** size(HP[*fap*].*v*) **do** SP = SP+1; ST[SP] = HP[*fap*].*v*[*i*] **od** **tel**	函数应用 指向约束 从 FUNVAL 对象中把变元装入栈

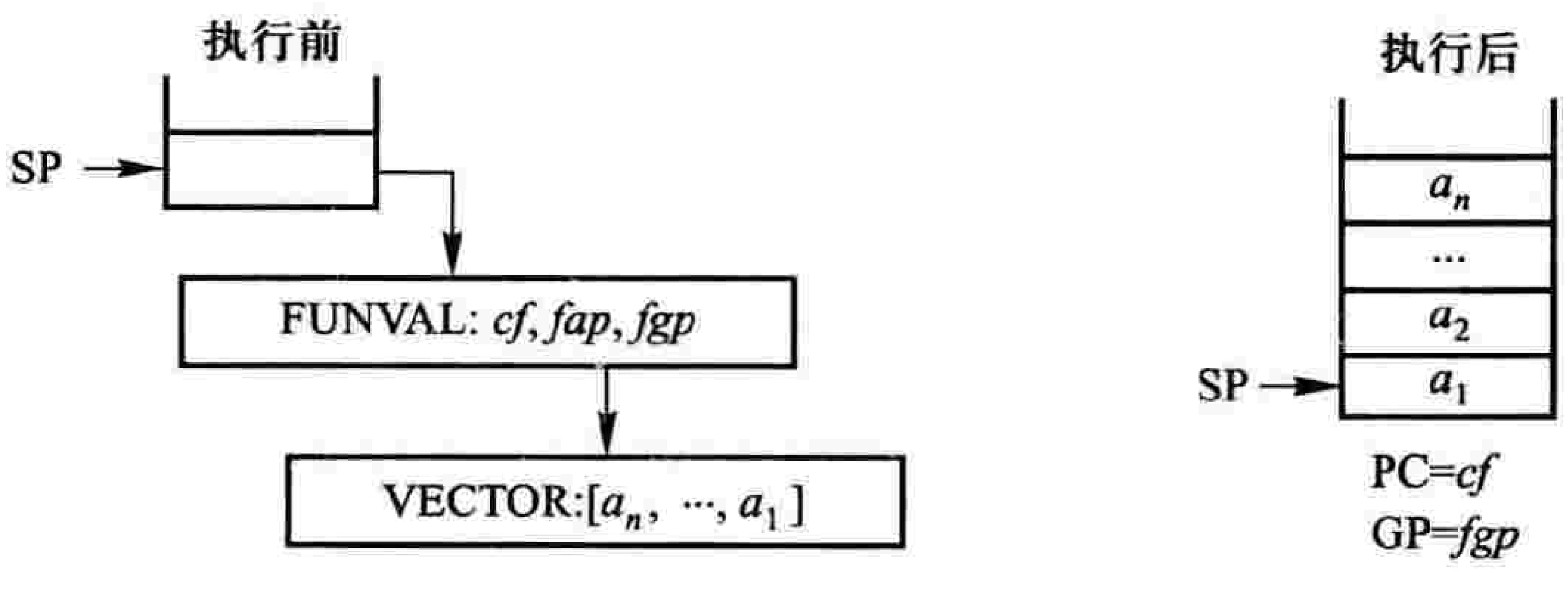


图 13.7 **apply** 指令执行前后情况

就实现函数的整个任务来说,已经讨论了下列子任务:

(1) 从函数定义生成 FUNVAL 对象,并且在正确的环境下编译函数体;

(2) 用 **mark** 指令为函数应用建立一个栈帧,随后的存储单元用于存放变元指针;

(3) 用 **apply** 指令,在建立了 FUNVAL 对象的正确约束后,函数代码开始执行。

此外,还有围绕着函数体的代码的两条指令需要介绍,即 **targ** 和 **return** 指令。**targ** 指令的

定义列在表 13.9 中,解释在图 13.8 中。**targ** 测试提供给函数的变量是否不足:如果不足,它把现存的变元组装到 FUNVAL 对象中,并且释放栈帧。

表 13.9 如果变元个数不足,形成 FUNVAL 对象

指令	含义	备注
targ n	**if** SP- FP < n **then** h=ST[FP-2]; ST[FP-2]=new(FUNVAL: PC-1, new(VECTOR: [ST[FP+1], ST[FP+2], …, ST[SP]]), GP); GP=ST[FP]; SP=FP-2; FP=ST[FP-1]; PC=h **fi**	变元个数不足

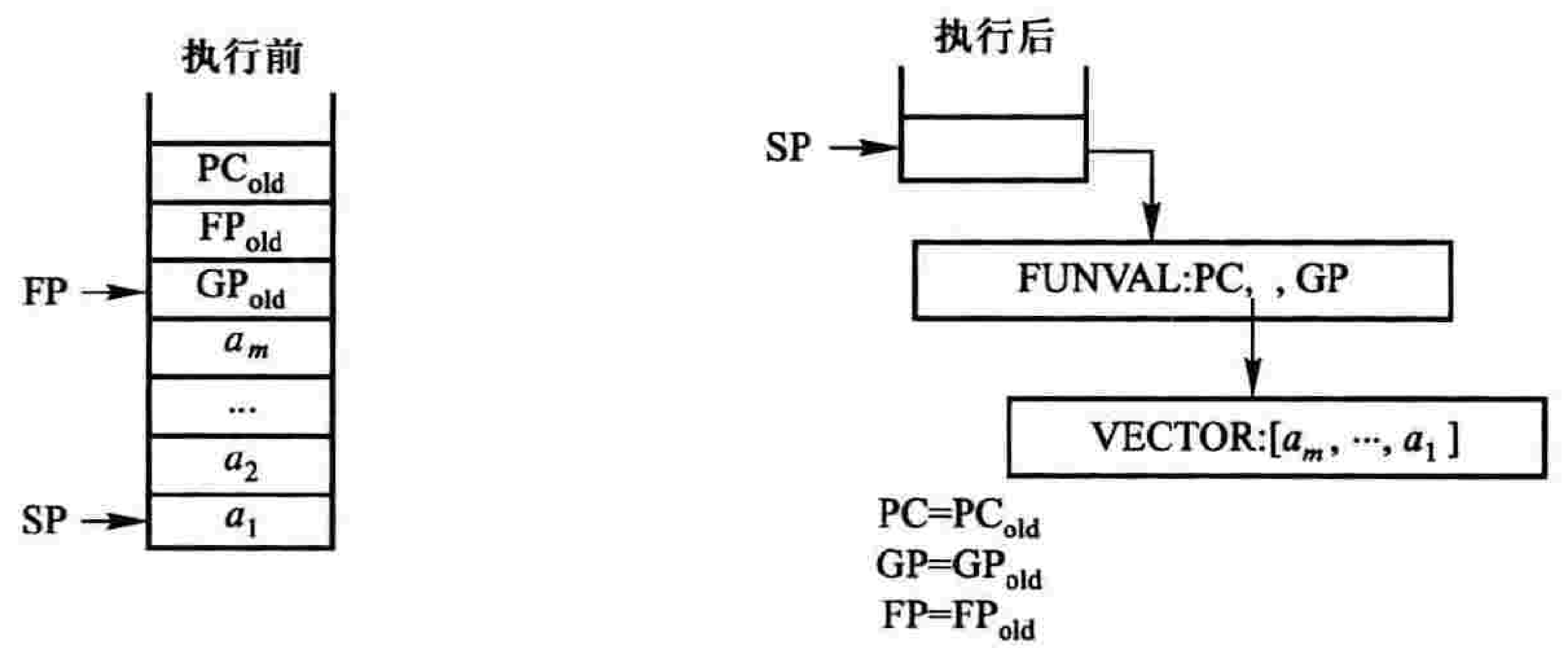


图 13.8 **targ** 指令发现变元不足($n>m$),建立 FUNVAL 对象

return 指令的责任是函数应用和闭包计算结尾的处理。它的参数定义了由函数应用或闭包计算消费的变元个数。它对应两种情况。在第一种情况中,栈帧包含的变元指针个数和被应用函数所需要的一样多,此时,将函数值复制到适当的地方,并释放当前栈帧。在第二种情况下,栈帧包含的变元指针个数多于被应用函数所需要的个数,此时的函数应用消费适当个数的变元,其结果是一个函数,再应用到剩余的变元,这些变元的指针仍在栈上。表达式($\lambda x.(\lambda yz.\ x+y+z)3)4\ 5$ 的执行会出现第二种情况,读者可以自行分析一下。

return 指令的定义在表 13.10,它的效果说明在图 13.9 中。

表 13.10 函数应用和闭包计算结尾的处理

指令	含义	备注

return *n*	**if** SP = FP +1+*n*
	then
	PC = ST[FP−2];
	GP = ST[FP];
	ST[FP−2] = ST[SP];
	SP = FP−2;
	FP = ST[FP−1]
	else
	if HP[ST[SP]].*tag* ≠FUNVAL **then** error **fi**;
	let(FUNVAL: *cf*, *fap*, *fgp*) = HP[ST[SP]]
	in PC = *cf*;
	GP = *fgp*;
	SP = SP− *n* −1;
	for *i* = 1 **to** size(HP[*fap*].*v*) **do**
	SP = SP +1;
	ST[SP] = HP[*fap*].*v*[*i*]
	od
	tel
	fi
		结束
		继续地址
		结果
		变元过多,函数的结果应用到剩余变元
		消费了 *n* 个变元

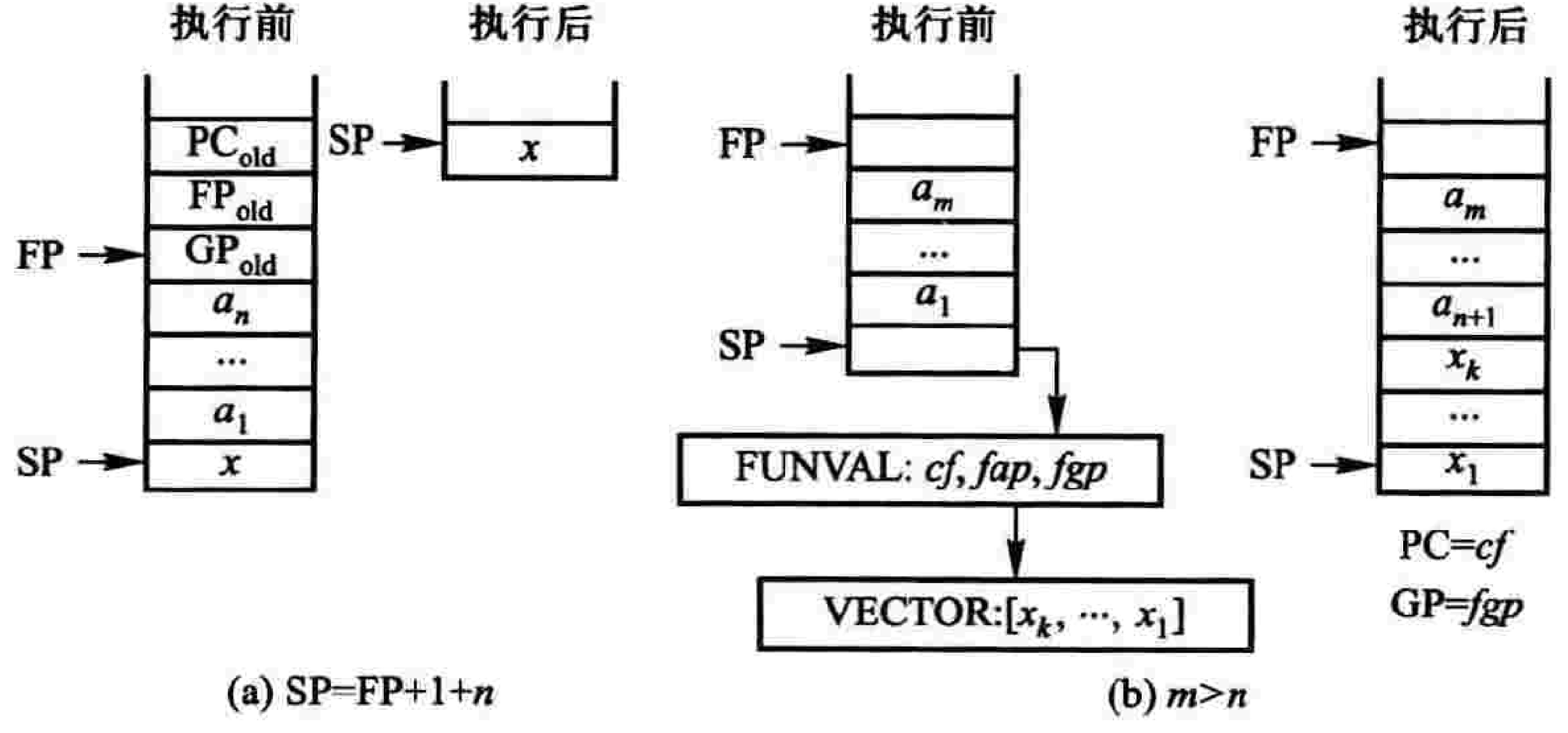


图 13.9 **return** 指令的两种情况

13.4.5 构造和计算闭包

用 C_code 函数编译表达式,其生成的代码执行时会为该表达式建立一个闭包。堆上的这个 CLOSURE 对象由两个指针组成,分别指向表达式代码和一个向量,该向量的每个元素指向一个全局变量的值。因为这些值都已给出,因此一个闭包是对它的外围没有其他需求的对象。和函数的处理类似,该表达式的真正编译是在 V 上下文和一个新环境下,该环境只知道全局变量和值为零的参数 *sl*。和函数应用一样,闭包的计算需要建立一个栈帧,在该栈帧中,局部变量可以按前面讨论过的方案寻址。

和编译函数定义一样,编译表达式得到的代码结构包含一个较外的块(建立闭包)和一个较内的块(表达式计算的代码)。

C_code $e\ \beta\ sl$ = pushfree $fr\ \beta\ sl$; // 将全局变量的值压栈
　　mkver g; // 把它们做成一个向量
　　ldl l_1;
　　mkclos;
　　ujmp l_2;
l_1: V_code $e\ [v_i \mapsto (\text{GLOB},\ i)]_{i=1}^{g}\ 0$;
　　update;
l_2:

其中,

$fr = [v_1,\ \cdots,\ v_g] = \text{list}((\text{freevar}(e))$

利用简单的优化,基本表达式可以处理得更有效。在这种情况下,不必显式地构造闭包。

C_code $b\ \beta\ sl$ = V_code $b\ \beta\ sl$

当生成的闭包的值被需要时,执行为 *e* 生成的代码。这由 **eval** 指令来完成,如果它在栈顶得到的是一个闭包的指针,那它就建立一个栈帧来计算该闭包。正如上下文 V 所刻画的,这个计算把结果留在堆中,并且在栈中该闭包的指针上面用一个指针指向堆中的这个结果。**update** 指令(表 13.11 和图 13.10)用这个结果去覆盖该闭包对象。这代表了 FAM 按需调用语义,因为以后再访问时无须重新计算,直接用第一次的计算结果即可。

图 13.11 说明 **eval** 指令在栈上的效果,它的定义列在表 13.12 中。

表 13.11 update 指令

指令	含义	备注
update	HP[ST[SP-4]] = HP[ST[SP]]; PC = ST[FP-2]; GP = ST[FP]; SP = FP-3; FP = ST[FP-1]	覆盖闭包

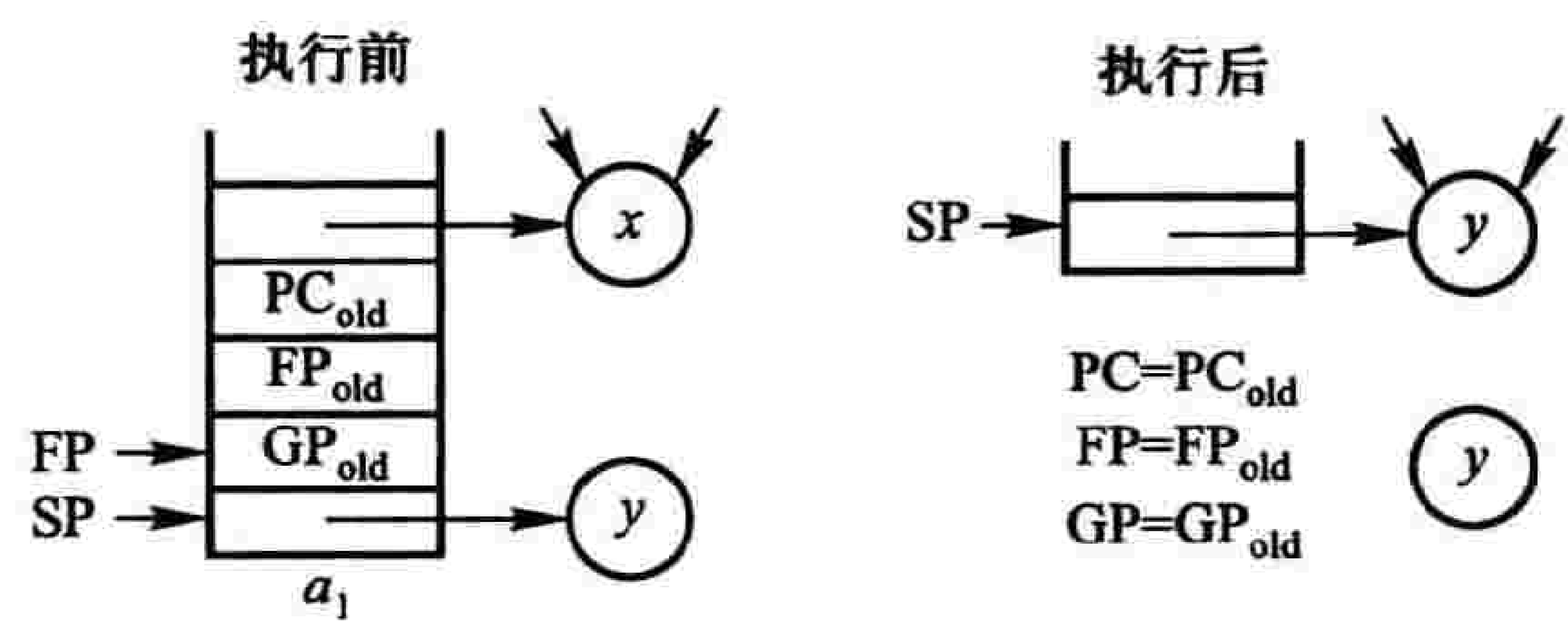


图 13.10 **update** 指令的效果

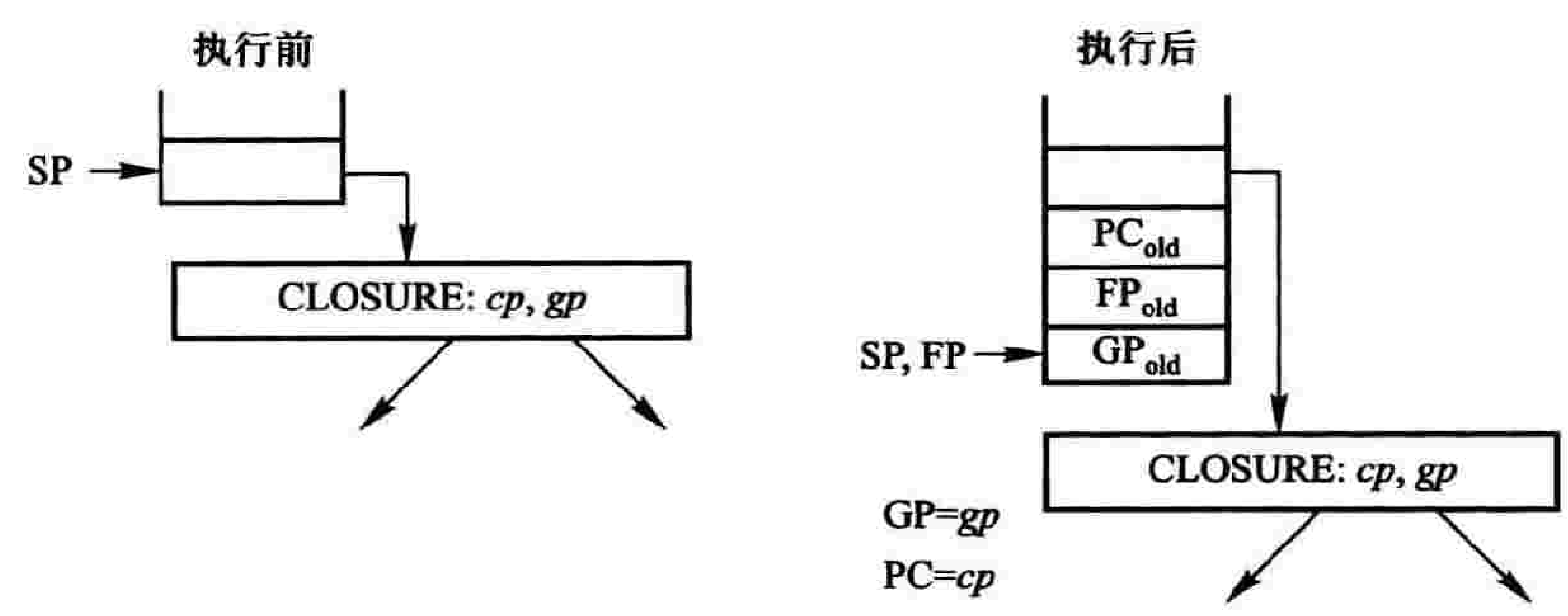


图 13.11 **eval** 指令的效果

表 13.12 在计算闭包时由 eval 完成栈帧的建立

指令	含义	备注
eval	**if** HP[ST[SP]].*tag* = CLOSURE	计算闭包
	then	
	ST[SP +1] = PC;	
	ST[SP +2] = FP;	
	ST[SP +3] = GP;	
	GP = HP[ST[SP]]. *gp*;	约束指针
	PC = HP[ST[SP]]. *cp*;	表达式代码的起始地址
	SP = SP +3;	
	FP = SP	
	fi	

13.4.6 letrec 表达式和局部变量

对于 letrec 表达式 **letrec** $v_1 == e_1$; …; $v_n == e_n$ **in** e_0，当在 V 上下文中编译它时，必须产生下列指令和环境：

(1) 为这 n 个表达式 $e_1,\cdots,e_n$建立闭包的指令序列必须产生;

(2) 计算 e_0的指令序列必须生成;

(3) 根据全局变量以及 v_1, …, v_n,为 e_0, e_1, …, e_n建立同样的环境。

它们由下列编译方案完成:

V_code(**letrec** $v_1 == e_1$; …; $v_n == e_n$ **in** e_0) β sl = repeat n **alloc**;
C_code e_1 β' sl';
rewrite n;
C_code e_2 β' sl';
rewrite $n-1$;
…
C_code e_n β' sl';
rewrite 1;
V_code e_0 β' sl';
slide n

其中,

$\beta' = \beta\ [v_i \mapsto (\mathrm{LOC},\ sl+i-1)]_{i=1}^{n}$;

$sl' = sl+n$;

repeat n c = **if** $n=0$ **then** nocode
else c;
repeat($n-1$) c
fi

逐步来考察这个方案。首先生成 n **alloc** 指令,它在堆上建立 n 个空对象,并把它们的指针压在栈上。然后为每一个表达式 e_j产生指令序列

C_code $e_j(\beta\ [v_i \mapsto (\mathrm{LOC},\ sl+i-1)]_{i=1}^{n})(sl+n)$;
rewrite($n-j+1$)

该序列为 e_j建立闭包,然后覆盖对应的空闭包对象。覆盖指令 **rewrite** m 的含义见表 13.13。因此,空闭包对象必须已经存在,建立闭包的代码才可以使用这些空对象的指针。但是它不访问这些空对象的成分。

在这儿,对于定义的次序必须有个假定。变量的 C_code 方案前面已经作了改进,用变量的约束作为返回而不是建立一个闭包。若不对定义排序,对于例 13.11,这将是灾难性的,因为这儿空闭包仍将用 **pushloc** 访问。为了避免这一点,对定义进行排序,使得像 $a==b$ 这样的变量命名总是处于 b 的定义的后面。循环重新命名是没有意义的,因此理所当然被编译器拒绝。

表 13.13　rewrite 指令和 slide 指令

指令	含义	备注
rewrite m	HP[ST[SP- m]] = HP[ST[SP]]; SP = SP-1	覆盖堆对象
slide m	ST[SP-m] = ST[SP]; SP = SP-m	复制结果

例 13.11　表达式

letrec $a == b$;

　　$b == 0$

in ⋯;　□

剩下还必须确定地址关系式(13.1)在这种情况下是否仍然保持,如果它在 letrec 编译前和生成的代码执行前分别保持的话。假设 sl 对应的开始值是 sl_0。n 次 **alloc** 使 SP 的值增加 n,结果是,e_1编译时的参数是 sl_0+n。**rewrite** 释放栈的最高存储单元,因此其他表达式 e_2, ⋯, e_n, e_0的编译仍然有正确的 sl 参数。

局部变量 v_i分配的地址是 sl_0+i-1,于是 v_1的地址是 sl_0, v_2的地址是 sl_0+1,等等。如果该letrec是函数体中第一个 letrec,那么 v_1的相对地址是 0, v_2的相对地址是 1,等等。因此,在函数getvar中用 **pushloc** 指令的局部变量寻址也是正确的。

习　题　13

13.1　为下列函数写出 SFP 表达式:

(a) 求两个数 a 和 b 的最大公约数的函数 *gcd*。

(b) 检查一个自然数是否为完全数的函数 *isperfect*。如果一个数等于它真因子的和,那么这个数就是完全数。例如,6 是完全数,因为 6 = 1+2+3。

13.2　确定下面 SFP 表达式中自由变量的集合和约束变量的集合:

$(\lambda x.\ xy)(\lambda y.\ y)$

$\lambda xy.\ z\ (\lambda z.\ z\ (\lambda x.\ y))$

$(\lambda xy.\ xz\ (yz\ ))(\lambda x.\ y(\lambda y.\ y))$

$\lambda x.\ x+$**letrec** $a == x$;

　　　$x == f\ y$;

　　　$y == z$

　　in $x+y\ +z$

13.3　把下列 SFP 表达式编译成 FAM 抽象机指令序列:

(a) $(\lambda x.\ x+1)\ 3$

(b) **letrec** $F == \lambda xy.\ xy$;

$inc == \lambda x.\ x+1$

in $F\ inc\ 5$

并说明在生成的 FAM 指令执行过程中,栈和堆是怎样变化的。

13.4 把下列 SFP 表达式编译成 FAM 代码:

letrec $fac == \lambda n.$ **if** $n = 0$ **then** 1 **else** $n * fac(dec\ n)$;

$dec == \lambda n.\ n - 1$

in fac 4

13.5 考虑如下形式的 SFP 程序(最外表达式是 **letrec** 表达式):

letrec $v_1 == e_1$;

…

$v_n == e_n$

in e

变量 v_i 可以按它们在栈开始点的绝对地址来寻址。引入一类称为 ABS 的变量,并且相应地修改编译方案。此时可以进行什么样的优化?

13.6 本习题优化下列形式的递归函数:

$f == \lambda v_1 \cdots v_n . \cdots (f\ e_1 \cdots e_n) \cdots$

in …

如果该递归调用 $(f\ e_1 \cdots e_n)$ 既不是一个算符的运算对象,又不是函数变元,那么称它为**尾递归**。在这种情况下,函数值的计算是通过对该函数的这个递归调用来完成的。通常,运行时碰到函数调用,必须建立一个新的栈帧。然而,对于尾递归,当前栈帧是空或者从递归调用返回后不再使用。于是,不必建立一个新的栈帧,而是在当前栈帧中计算递归调用。此时,老变元的指针直接由那些新变元的指针代替。

定义相应的指令,并且修改编译方案,使它能识别尾递归调用并且用较有效的方式处理它们。(提示:扩充那些代码函数,增加一个参数,该参数包含有关被编译表达式上下文的信息)

13.7 修改 FUNVAL 对象的结构以及函数定义和函数应用的代码方案,使得在函数变元不足的情况下,变元的解包和重新打包可以避免。

参 考 文 献

[1] Alfred V Aho, Ravi Sethi, Jeffrey D Ullman. Compilers: principles, techniques, and tools [M]. New York: Addison-Wesley, 1986.

[2] Reinhard Wilhelm, Dieter Maurer. Compiler design [M]. New York: Addison-Wesley, 1995.

[3] Andrew W Appel. Modern compiler implementation in Java [M]. Cambridge: Cambridge University Press, 1998.

[4] Andrew W Appel. Modern compiler implementation in C [M]. Cambridge: Cambridge University Press, 1998.

[5] Steven S Muchnick. Advanced compiler design and implementation [M]. Hong Kong: Morgan Kaufmann Publishers, 1997.

[6] John R Levine. Linkers and loaders [M]. Hong Kong: Morgan Kaufmann Publishers, 1999.

[7] Alfred V Aho, Monica S Lam, Ravi Sethi, et al. Compilers: principles, techniques, and tools [M]. 2nd ed. New York: Addison-Wesley, 2007.

[8] 陈意云,张昱. 编译原理习题精选与解析[M].2 版. 北京:高等教育出版社,2013.

[9] 张昱,陈意云. 编译原理与技术[M]. 北京:高等教育出版社,2010.

[10] 张昱,陈意云. 编译原理实验教程[M]. 北京:高等教育出版社,2009.

编译原理（第3版）

本书特色：

- 内容全面、强调主线。包括词法分析、语法分析、语法制导的翻译、静态语义分析、运行时存储空间的组织和管理、中间代码生成、目标代码生成、代码优化、编译系统与运行系统、面向对象语言编译技术和函数式语言编译技术等，并以编译的各个逻辑阶段为主线。

- 重视理论和形式方法。在围绕主线的同时，将相关理论和形式化技术的介绍穿插其中，有助于学生较快地领会和掌握；内容难易有别，难度较大的内容作为可选部分放在每章的最后，以拓宽教材的适用面。

- 习题联系实际。本教材的很多例题和习题是从实际碰到的问题中抽象或抽取出来的；它们联系编程、编译、运行的实际，能激发学生学习本课程的兴趣。