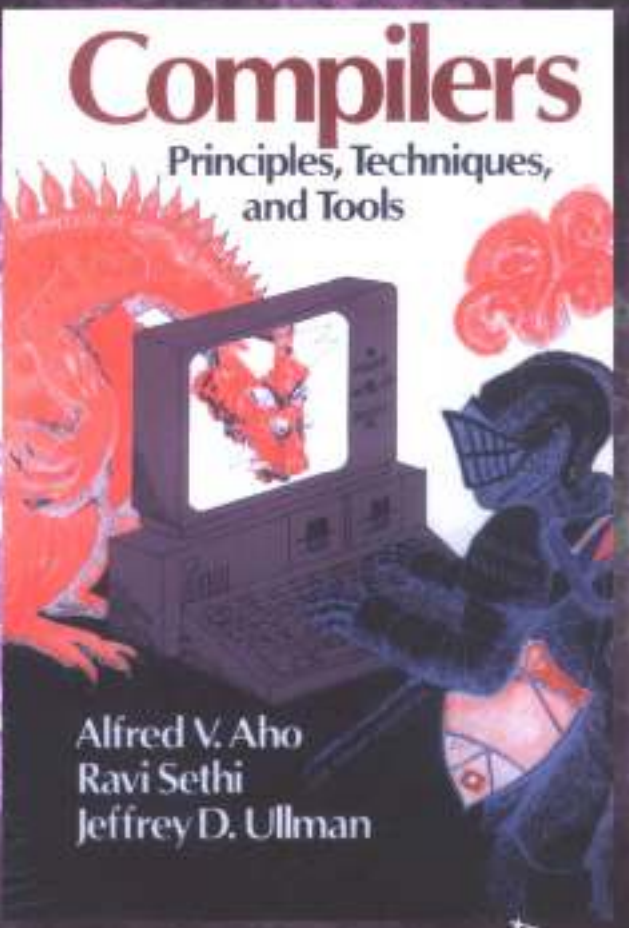
计 算 机 科 学 丛 书

编译原理

(美) Alfred V. Aho 贝尔实验室 Ravi Sethi Avaya实验室 Jeffrey D. Ullman 斯坦福大学 著 李建中 姜守旭 译



Compilers
Principles, Techniques, and Tools

机械工业出版社
China Machine Press

本书作者Alfred V. Aho、Ravi Sethi和Jeffrey D. Ullman是世界著名的计算机科学家，他们在计算机科学理论、数据库等很多领域都做出了杰出贡献。本书是编译领域无可替代的经典著作，被广大计算机专业人士誉为"龙书"。本书一直被世界各地的著名高等院校和科研机构（如贝尔实验室、哥伦比亚大学、普林斯顿大学和斯坦福大学等）广泛用作本科生和研究生编译原理与技术课程的教材，本书对我国计算机教育界也具有重大影响。

书中深入讨论了编译器设计的重要主题，包括词法分析、语法分析、语法制导分析、类型检查、运行环境、中间代码生成、代码生成、代码优化等，并在最后两章中讨论了实现编译器的一些编程问题和几个编译器实例，而且每章都提供了大量的练习和参考文献。

本书可以作为高等院校计算机专业本科生和研究生编译原理与技术课程的教材，也可以作为计算机技术人员必读的专业参考书之一。

作者简介

Alfred V. Aho 于普林斯顿大学获得博士学位，现任贝尔实验室基础科学研究院副院长、计算机科学研究中心主任，在贝尔实验室主要负责计算科学和软件研究工作，已经出版多本算法、数据结构、编译器、数据库系统及计算机科学基础等方面的经典著作。



Ravi Sethi 于普林斯顿大学获得博士学位，他1976年进入贝尔实验室，在贝尔实验室从事研究工作24年之久，并担任过贝尔实验室通信科学研究部高级副总裁。Sethi博士现任Avaya实验室总裁，他还是《程序设计语言：概念和结构》一书的作者。



Jeffrey D. Ullman 斯坦福大学计算机科学系教授，他的研究方向包括数据库理论、数据库集成、数据挖掘和利用信息基础设施教学等。他著有《数据库系统概念》等多本重要的数据库教材。



封面设计：刘永青

ISBN 7-111-12349-2

9 787111 123491

HZ BOOKS 华章图书

网上购书：www.china-pub.com

北京市西城区百万庄南街1号 100037

读者服务热线：(010)68995259，68995264

读者服务信箱：hzedu@hzbook.com

http://www.hzbook.com

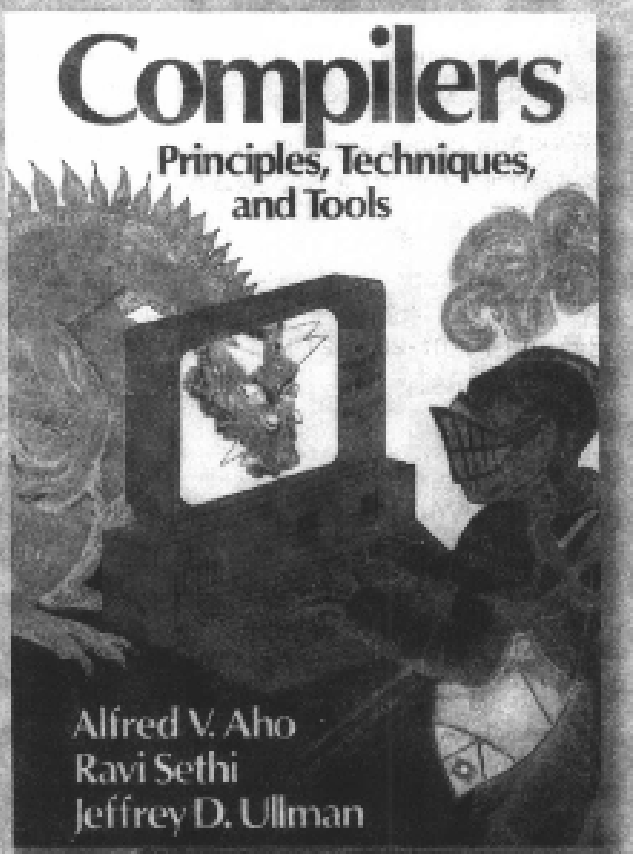
ISBN 7-111-12349-2/TP · 2742

定价：55.00 元

计 算 机 科 学 丛 书

编译原理

(美) Alfred V. Aho 贝尔实验室 Ravi Sethi Avaya实验室 Jeffrey D. Ullman 斯坦福大学 著 李建中 姜守旭 译



Compilers
Principles, Techniques, and Tools

TP314/68

机械工业出版社
China Machine Press

0767737

本书深入讨论了编译器设计的重要主题，包括词法分析、语法分析、语义指导分析、类型检查、运行环境、中间代码生成、代码生成、代码优化等，并在最后两章中讨论了实现编译器的一些编程问题和几个编译器实例，每章都提供了大量的练习和参考文献。本书从介绍编译的原理性概念开始，然后通过构建一个简单的一遍编译器来逐一解释这些概念。

本书是编译原理课程的经典教材，作者曾多次使用本书的内容在贝尔实验室、哥伦比亚大学、普林斯顿大学和斯坦福大学向本科生和研究生讲授初等及高等编译课程。

Authorized translation from the English language edition entitled Compilers: Principles, Techniques, and Tools by Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, published by Pearson Education, Inc, publishing as Addison-Wesley , Copyright © 1986 by Bell Telephone Laboratories, Incorporated.

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

Chinese simplified language edition published by China Machine Press.

Copyright © 2003 by China Machine Press.

本书中文简体字版由美国 Pearson Education 培生教育出版集团授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

本书版权登记号：图字：01-2001-2194

图书在版编目（CIP）数据

编译原理/（美）阿霍（Aho, A. V.）等著；李建中等译. -北京：机械工业出版社，2003.8
（计算机科学丛书）
书名原文：Compilers: Principles, Techniques, and Tools
ISBN 7-111-12349-2

Ⅰ.编… Ⅱ.①阿… ②李… Ⅲ.编译程序-程序设计-教材 Ⅳ.TP314

中国版本图书馆CIP数据核字（2003）第050118号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：杨海玲
北京中加印刷有限公司印刷·新华书店北京发行所发行
2003年8月第1版第1次印刷
787mm×1092mm 1/16·34印张
印数：0 001-5 000册
定价：55.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

出版者的话

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

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

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

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

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

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业

的教学度身订造的。其中许多教材均已为M. I. T.，Stanford，U.C. Berkeley，C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

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

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

专家指导委员会

（按姓氏笔画顺序）

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

译 者 序

编译器产生于20世纪60年代，在计算机科学技术的发展历史中发挥了巨大作用，是开发计算机应用系统不可缺少的重要工具。编译器的原理和技术具有十分普遍的意义。在每一个计算机科学技术工作者的职业生涯中，这些原理和技术都被反复用到。编译器的编写涉及到程序设计语言、计算机体系结构、语言理论、算法和软件工程等学科，是计算机科学技术的重要基础。作为计算机科学技术学科的专业基础课，编译器原理和技术是计算机科学技术专业学生的必修课程。本书是一部优秀的编译器原理和技术教材。

本书是 Alfred V. Aho 和 Jeffrey D. Ullman 所著的《*Principles of Compiler Design*》一书的后裔版。本书作者 Alfred V. Aho 是 AT&T 贝尔实验室计算机原理研究部负责人，Jeffrey D. Ullman 是斯坦福大学计算机科学系教授，Ravi Sethi 是 AT&T 贝尔实验室研究人员。Alfred V. Aho 和 Jeffrey D. Ullman 是世界著名的计算机科学家，他们在计算机科学理论、数据库等很多领域都做出了杰出贡献。他们的很多著作都被国际公认为是权威性著作，深受读者的喜爱。本书的英文版出版于20世纪80年代，是一部著名的编译器原理与技术方面的教材，一直被国际著名高等院校特别是美国著名大学作为编译器原理与技术的教科书。这部著作对我国计算机界也具有重大影响。机械出版社独具慧眼，决定将这部著作翻译成中文在国内出版，这必将对我国计算机科学技术的编译原理教学工作产生积极的推动作用。有幸承担该书的翻译工作，我们感到十分荣幸。

本书是编译器原理与技术的基本教程，旨在介绍编译器的一般原理，解决人们在编译器设计中遇到的普遍问题。本书在系统地介绍编译器的一般原理的同时，特别注重编译原理和技术的实际应用，给出了许多启示，并配置了大量的例题和习题。本书的内容适用于所有源语言和目标机器。本书介绍的概念和技术不仅适用于编译器的设计，也适用于一般的软件设计。显然，本书在目前只有少数人涉及编译器的构造和维护的情况下仍然具有重要的意义和价值。

本书共有十二章和一个附录。第1章介绍编译器的基本结构；第2章描述了一个变中缀表达式为后缀表达式的翻译器；第3章介绍了词法分析器、正规表达式、有穷自动机以及词法分析器的自动生成工具；第4章详述常用的语法分析技术；第5章阐述了语法制导翻译的基本概念；第6章介绍了实现静态语义检查的基本思想；第7章讨论了程序运行环境的存储组织问题；第8章首先介绍了中间语言的概念，然后讨论如何把一般的程序设计语言翻译成中间代码的问题；第9章介绍目标代码生成技术和代码生成器的自动生成方法；第10章全面介绍了代码优化方法；第11章讨论了实现编译器的一些编程问题；第12章提供了几个编译器实例；附录A描述了一种简单的语言，学生可以把它作为源语言，构造一个编译器。

本书可以作为高等院校计算机专业本科生和研究生编译原理与技术课程的教材，也可以作为计算机软件工作者的技术参考书。

限于译者水平，译文中疏漏和错误难免，欢迎批评指正。

李建中，姜守旭

2003年7月1日

译者简介



李建中，哈尔滨工业大学教授，博士生导师，国家杰出青年基金获得者，中国计算机学会理事，中国计算机学会数据库专业委员会副主任。从事计算机科学技术的教学、研究、开发工作二十余年。主要研究领域为数据库系统与并行计算，主持完成研究项目20余项，在统计与科学数据库、并行数据库、数据仓库、数据挖掘等方面取得了一系列研究成果，在IEEE Transactions on Knowledge and Data Engineering、VLDB、ACM SIGMOD等国内外重要学术刊物和学术会议发表学术论文180余篇，出版学术专著和教材4部，获得各类科学技术奖励多项。



姜守旭，哈尔滨工业大学副教授，硕士生导师。1990年毕业于哈尔滨工业大学计算机及应用专业并留校任教，1995年获得计算机软件硕士学位。多年来一直从事计算机科学技术的教学与科研工作，主要研究方向为操作系统与对等计算。曾获部科技进步三等奖一项，出版教材一部。

前　言

本书是 Alfred V.Aho和Jeffrey D.Ullman 所著的《*Principles of Compiler Design*》一书的后续版本。与后者类似，本书也是编译器设计基础课程的教材。本书的重点是解决人们在设计语言翻译器时遇到的普遍问题，而不论源和目标机器是什么。

本书介绍的概念和技术不仅适用于编译器的设计，也适用于一般的软件设计。例如，建立词法分析器的串匹配技术已用于文本编辑器、信息检索系统和模式识别器；上下文无关文法和语法制导定义等概念已用于设计许多诸如本书产生的排版、绘图系统这样的小语言；代码优化技术已用于程序验证器和从非结构化程序产生结构化程序的程序检验器之中。显然，本书在目前只有少数人涉及编译器的构造和维护的情况下仍然具有重要的意义和价值。

本书的使用方法

本书深入地讨论了编译器设计的重要主题。

第1章介绍编译器的基本结构，是阅读本书其余部分的基础。

第2章描述了一个变中缀表达式为后缀表达式的翻译器。这个翻译器使用本书介绍的一些基本技术构建。后面的一些章节逐渐地扩展了第2章介绍的内容。

第3章介绍了词法分析器、正规表达式、有穷状态机以及词法分析器的生成器工具。本章的内容已经被广泛地用于文本处理。

第4章深入介绍了常用的语法分析技术。本书讨论的语法分析技术比较广泛，从适用于手工实现的递归下降技术到用于语法分析器生成器的计算更密集的LR技术。

第5章介绍了语法制导翻译的主要概念。本章的内容将被用于本书中说明和实现翻译的其余各章。

第6章介绍了实现静态语义检查的主要思想，详尽讨论了类型检查与合一问题。

第7章讨论了用于支持程序运行环境的存储组织问题。

第8章首先介绍了中间语言的概念，然后讨论如何把一般的程序设计语言结构翻译成中间代码的问题。

第9章介绍目标代码生成技术，包括简单的代码生成方法以及产生表达式代码的优化方法。本章也讨论了窥孔优化方法和代码生成器的生成器。

第10章全面介绍了代码优化，详细讨论了各种数据流分析方法和几种主要的全局优化方法。

第11章讨论实现编译器的一些编程问题。软件工程和软件测试在构造编译器的过程中是非常重要的。

第12章提供几个编译器实例。这些编译器都使用了本书介绍的技术。

附录A描述了一种简单的语言。这种语言是Pascal语言的“子集”，它可以用做实现项目的基础。

本书作者曾使用本书的内容多次在贝尔实验室、哥伦比亚大学、普林斯顿大学和斯坦福

大学为本科生和研究生讲授初等和高等编译课程。

初等编译课程可以由本书以下章节构成：

简介	第1章、2.1~2.5节
词法分析	2.6节、3.1~3.4节
符号表	2.7节、7.6节
语法分析	2.4节、4.1~4.4节
语法制导翻译	2.5节、5.1~5.5节
类型检查	6.1~6.2节
运行环境的组织	7.1~7.3节
中间代码的生成	8.1~8.3节
代码生成	9.1~9.4节
代码优化	10.1~10.2节

第2章描述了编程项目所需的信息。

以编译器构造工具为核心的课程可以包括3.5节的词法分析器的生成器、4.8节和4.9节的语法分析器的生成器、9.12节的代码生成器的生成器以及第11章中有关编译器构造技术的内容。

高等编译课程可以由本书的以下内容组成：第3章和第4章介绍的词法分析器的生成器和语法分析器的生成器中使用的算法、第6章介绍的有关类型等价、重载、多态和合一的内容、第7章介绍的程序运行环境的存储组织、第9章介绍的模式制导的代码生成方法、第10章介绍的代码优化。

习题

我们用星号来标记习题的难度。没有星号的习题检验学生对基本定义的理解；具有一个星号的习题用于高等编译课程；具有两个星号的习题是值得深思熟虑的难题。

致谢

在本书的编写过程中，很多人都对手稿给予了有价值的建议。在此，我们对以下人员表示衷心的感谢，他们是：Bill Appelbe、Nelson Beebe、Jon Bentley、Lois Bogess、Rodney Farrow、Stu Feldman、Charles Fischer、Chris Fraser、Art Gittelman、Eric Grosse、Dave Hanson、Fritz Henglein、Robert Henry、Gerard Holzmann、Steve Johnson、Brian Kernighan、Ken Kubota、Daniel Lehmann、Dave MacQueen、Dianne Maki、Alan Martin、Doug McIlroy、Charles McLaughlin、John Mitchell、Elliott Organick、Robert Paige、Phil Pfeiffer、Rob Pike、Kari-Jouko Räihä、Dennis Ritchie、Sriram Sankar、Paul Stoecker、Bjarne Stroustrup、Tom Szymanski、Kim Track、Peter Weinberger、Jennifer Widom 和 Reinhard Wilhelm.

作者特别感谢 Patricia Solomon 对本书图片的加工制作。她的工作热情和专业水准是非常令人钦佩的。在本书的编写过程中，J.D.Ullman 得到了以色列艺术和科学学会的爱因斯坦奖学金的资助。最后，我们要特别感谢贝尔实验室在本书准备过程中对我们的支持。

A. V. A., R. S., J. D. U.

目　录

出版者的话
专家指导委员会
译者序
前言
第1章　编译简介 ……1
1.1　编译器 ……1
1.1.1　编译的分析-综合模型 ……1
1.1.2　编译器的前驱与后继 ……3
1.2　源程序分析 ……3
1.2.1　词法分析 ……3
1.2.2　语法分析 ……3
1.2.3　语义分析 ……5
1.2.4　文本格式器中的分析 ……5
1.3　编译器的各阶段 ……6
1.3.1　符号表管理 ……7
1.3.2　错误检测与报告 ……7
1.3.3　各分析阶段 ……7
1.3.4　中间代码生成 ……9
1.3.5　代码优化 ……9
1.3.6　代码生成 ……10
1.4　编译器的伙伴 ……10
1.4.1　预处理器 ……10
1.4.2　汇编器 ……11
1.4.3　两遍汇编 ……12
1.4.4　装配器和连接编辑器 ……12
1.5　编译器各阶段的分组 ……13
1.5.1　前端与后端 ……13
1.5.2　编译器的遍 ……13
1.5.3　减少编译的遍数 ……14
1.6　编译器的构造工具 ……14
参考文献注释 ……15
第2章　简单的一遍编译器 ……17
2.1　概述 ……17
2.2　语法定义 ……17
2.2.1　分析树 ……19
2.2.2　二义性 ……20
2.2.3　操作符的结合规则 ……20
2.2.4　操作符的优先级 ……21
2.3　语法制导翻译 ……22
2.3.1　后缀表示 ……22
2.3.2　语法制导定义 ……22
2.3.3　综合属性 ……23
2.3.4　深度优先遍历 ……24
2.3.5　翻译模式 ……25
2.3.6　翻译的输出 ……25
2.4　语法分析 ……26
2.4.1　自顶向下语法分析 ……27
2.4.2　预测分析法 ……29
2.4.3　何时使用ε产生式 ……30
2.4.4　设计一个预测语法分析器 ……30
2.4.5　左递归 ……31
2.5　简单表达式的翻译器 ……32
2.5.1　抽象语法和具体语法 ……32
2.5.2　调整翻译模式 ……33
2.5.3　非终结符*expr*、*term*和*rest*的过程 ……33
2.5.4　翻译器的优化 ……35
2.5.5　完整程序 ……35
2.6　词法分析 ……37
2.6.1　剔除空白符和注释 ……37
2.6.2　常数 ……37
2.6.3　识别标识符和关键字 ……37
2.6.4　词法分析器的接口 ……38
2.6.5　词法分析器 ……38
2.7　符号表 ……40
2.7.1　符号表接口 ……40
2.7.2　处理保留的关键字 ……41
2.7.3　符号表的实现方法 ……41
2.8　抽象堆栈机 ……42
2.8.1　算术指令 ……42
2.8.2　左值和右值 ……43

2.8.3 堆栈操作 ……43
2.8.4 表达式的翻译……43
2.8.5 控制流 ……44
2.8.6 语句的翻译 ……44
2.8.7 输出一个翻译……45
2.9 技术的综合 ……46
2.9.1 翻译器的描述……46
2.9.2 词法分析器模块lexer.c ……47
2.9.3 语法分析器模块parser.c……48
2.9.4 输出模块emitter.c ……48
2.9.5 符号表模块symbol.c和init.c……48
2.9.6 错误处理模块error.c ……48
2.9.7 编译器的建立……48
2.9.8 程序清单 ……49
练习 ……53
编程练习……54
参考文献注释 ……55
第3章 词法分析 ……57
3.1 词法分析器的作用 ……57
3.1.1 词法分析中的问题 ……58
3.1.2 记号、模式、词素 ……58
3.1.3 记号的属性 ……59
3.1.4 词法错误 ……60
3.2 输入缓冲 ……60
3.2.1 双缓冲区 ……61
3.2.2 标志 ……62
3.3 记号的描述 ……62
3.3.1 串和语言 ……62
3.3.2 语言上的运算……63
3.3.3 正规表达式 ……64
3.3.4 正规定义 ……65
3.3.5 缩写表示法 ……66
3.3.6 非正规集 ……66
3.4 记号的识别 ……67
3.4.1 状态转换图 ……68
3.4.2 状态转换图的实现 ……70
3.5 词法分析器描述语言 ……72
3.5.1 Lex说明 ……72
3.5.2 超前扫描操作……75
3.6 有穷自动机 ……76
3.6.1 不确定的有穷自动机 ……77
3.6.2 确定的有穷自动机 ……78
3.6.3 从NFA到DFA的变换 ……79
3.7 从正规表达式到NFA……81
3.7.1 从正规表达式构造NFA ……81
3.7.2 NFA的双堆栈模拟 ……84
3.7.3 时间空间的权衡 ……85
3.8 设计词法分析器的生成器 ……85
3.8.1 基于NFA的模式匹配 ……86
3.8.2 词法分析器的DFA ……88
3.8.3 实现超前扫描操作 ……88
3.9 基于DFA的模式匹配器的优化 ……89
3.9.1 NFA的重要状态 ……89
3.9.2 从正规表达式到DFA ……89
3.9.3 最小化DFA的状态数 ……93
3.9.4 词法分析器的状态最小化 ……95
3.9.5 表压缩方法 ……95
练习 ……97
编程练习 ……103
参考文献注释 ……103
第4章 语法分析 ……105
4.1 语法分析器的作用……105
4.1.1 语法错误的处理 ……106
4.1.2 错误恢复策略 ……108
4.2 上下文无关文法……109
4.2.1 符号的使用约定 ……110
4.2.2 推导 ……110
4.2.3 分析树和推导 ……112
4.2.4 二义性 ……113
4.3 文法的编写 ……113
4.3.1 正规表达式和上下文无关文法的比较 ……114
4.3.2 验证文法所产生的语言……114
4.3.3 消除二义性 ……115
4.3.4 消除左递归 ……116
4.3.5 提取左因子 ……117
4.3.6 非上下文无关语言的结构……118
4.4 自顶向下语法分析……120
4.4.1 递归下降语法分析法 ……120
4.4.2 预测语法分析器 ……121

4.4.3 预测语法分析器的状态转换图 ……121
4.4.4 非递归的预测分析 ……123
4.4.5 FIRST和FOLLOW ……124
4.4.6 预测分析表的构造 ……125
4.4.7 LL(1)文法 ……126
4.4.8 预测分析的错误恢复 ……127
4.5 自底向上语法分析 ……128
4.5.1 句柄 ……129
4.5.2 句柄裁剪 ……130
4.5.3 用栈实现移动归约分析 ……131
4.5.4 活前缀 ……133
4.5.5 移动归约分析过程中的冲突 ……133
4.6 算符优先分析法 ……134
4.6.1 使用算符优先关系 ……135
4.6.2 从结合律和优先级获得算符优先关系 ……136
4.6.3 处理一元操作符 ……137
4.6.4 优先函数 ……137
4.6.5 算符优先分析中的错误恢复 ……139
4.7 LR语法分析器 ……142
4.7.1 LR语法分析算法 ……142
4.7.2 LR文法 ……145
4.7.3 构造SLR语法分析表 ……146
4.7.4 构造规范LR语法分析表 ……151
4.7.5 构造LALR语法分析表 ……155
4.7.6 LALR语法分析表的有效构造方法 ……158
4.7.7 LR语法分析表的压缩 ……161
4.8 二义文法的应用 ……163
4.8.1 使用优先级和结合规则来解决分析动作的冲突 ……163
4.8.2 悬空else的二义性 ……164
4.8.3 特例产生式引起的二义性 ……165
4.8.4 LR语法分析中的错误恢复 ……167
4.9 语法分析器的生成器 ……168
4.9.1 语法分析器的生成器Yacc ……169
4.9.2 用Yacc处理二义文法 ……171
4.9.3 用Lex建立Yacc的词法分析器 ……173
4.9.4 Yacc的错误恢复 ……174
练习 ……174
参考文献注释 ……182
第5章 语法制导翻译 ……185
5.1 语法制导定义 ……185
5.1.1 语法制导定义的形式 ……186
5.1.2 综合属性 ……186
5.1.3 继承属性 ……187
5.1.4 依赖图 ……187
5.1.5 计算顺序 ……189
5.2 语法树的构造 ……189
5.2.1 语法树 ……190
5.2.2 构造表达式的语法树 ……190
5.2.3 构造语法树的语法制导定义 ……191
5.2.4 表达式的无环有向图 ……192
5.3 自底向上计算S属性定义 ……194
5.4 L属性定义 ……195
5.4.1 L属性定义 ……196
5.4.2 翻译模式 ……196
5.5 自顶向下翻译 ……198
5.5.1 从翻译模式中消除左递归 ……198
5.5.2 预测翻译器的设计 ……201
5.6 自底向上计算继承属性 ……202
5.6.1 删除嵌入在翻译模式中的动作 ……202
5.6.2 分析栈中的继承属性 ……203
5.6.3 模拟继承属性的计算 ……204
5.6.4 用综合属性代替继承属性 ……206
5.6.5 一个难计算的语法制导定义 ……207
5.7 递归计算 ……207
5.7.1 从左到右遍历 ……207
5.7.2 其他遍历方法 ……208
5.8 编译时属性值的空间分配 ……209
5.8.1 在编译时为属性分配空间 ……209
5.8.2 避免复制 ……211
5.9 编译器构造时的空间分配 ……211
5.9.1 从文法中预知生存期 ……212
5.9.2 不相重叠的生存期 ……214
5.10 语法制导定义的分析 ……215
5.10.1 属性的递归计算 ……216
5.10.2 强无环的语法制导定义 ……216
5.10.3 环形检测 ……217
练习 ……219

参考文献注释 ……221
第6章 类型检查 ……223
6.1 类型系统 ……224
6.1.1 类型表达式 ……224
6.1.2 类型系统 ……225
6.1.3 静态和动态类型检查 ……226
6.1.4 错误恢复 ……226
6.2 一个简单的类型检查器的说明 ……226
6.2.1 一种简单语言 ……226
6.2.2 表达式的类型检查 ……227
6.2.3 语句的类型检查 ……228
6.2.4 函数的类型检查 ……228
6.3 类型表达式的等价 ……229
6.3.1 类型表达式的结构等价 ……229
6.3.2 类型表达式的名字 ……231
6.3.3 类型表示中的环 ……232
6.4 类型转换 ……233
6.5 函数和运算符的重载 ……234
6.5.1 子表达式的可能类型的集合 ……235
6.5.2 缩小可能类型的集合 ……236
6.6 多态函数 ……237
6.6.1 为什么要使用多态函数 ……237
6.6.2 类型变量 ……238
6.6.3 包含多态函数的语言 ……239
6.6.4 代换、实例和合一 ……240
6.6.5 多态函数的检查 ……241
6.7 合一算法 ……244
练习 ……247
参考文献注释 ……251
第7章 运行时环境 ……253
7.1 源语言问题 ……253
7.1.1 过程 ……253
7.1.2 活动树 ……253
7.1.3 控制栈 ……255
7.1.4 声明的作用域 ……256
7.1.5 名字的绑定 ……256
7.1.6 一些问题 ……257
7.2 存储组织 ……257
7.2.1 运行时内存的划分 ……257
7.2.2 活动记录 ……258
7.2.3 编译时的局部数据布局 ……259
7.3 存储分配策略 ……260
7.3.1 静态存储分配 ……260
7.3.2 栈式存储分配 ……262
7.3.3 悬空引用 ……265
7.3.4 堆式存储分配 ……265
7.4 对非局部名字的访问 ……266
7.4.1 程序块 ……267
7.4.2 无嵌套过程的词法作用域 ……268
7.4.3 包含嵌套过程的词法作用域 ……269
7.4.4 动态作用域 ……274
7.5 参数传递 ……275
7.5.1 传值调用 ……275
7.5.2 引用调用 ……276
7.5.3 复制-恢复 ……277
7.5.4 传名调用 ……277
7.6 符号表 ……278
7.6.1 符号表表项 ……278
7.6.2 名字中的字符 ……279
7.6.3 存储分配信息 ……280
7.6.4 符号表的线性表数据结构 ……280
7.6.5 散列表 ……281
7.6.6 表示作用域的信息 ……283
7.7 支持动态存储分配的语言措施 ……285
7.7.1 垃圾单元 ……285
7.7.2 悬空引用 ……286
7.8 动态存储分配技术 ……287
7.8.1 固定块的显式分配 ……287
7.8.2 变长块的显式分配 ……287
7.8.3 隐式存储释放 ……288
7.9 Fortran语言的存储分配 ……288
7.9.1 COMMON区域中的数据 ……289
7.9.2 一个简单的等价算法 ……290
7.9.3 Fortran语言的等价算法 ……292
7.9.4 映射数据区 ……294
练习 ……294
参考文献注释 ……298
第8章 中间代码生成 ……299
8.1 中间语言 ……299
8.1.1 图表示 ……299

8.1.2 三地址码 ……300
8.1.3 三地址语句的类型 ……301
8.1.4 语法制导翻译生成三地址码 ……302
8.1.5 三地址语句的实现 ……303
8.1.6 表示方法比较：间址的使用 ……305
8.2 声明语句 ……305
8.2.1 过程中的声明语句 ……305
8.2.2 跟踪作用域信息 ……306
8.2.3 记录中的域名 ……308
8.3 赋值语句 ……309
8.3.1 符号表中的名字 ……309
8.3.2 临时名字的重用 ……310
8.3.3 寻址数组元素 ……311
8.3.4 数组元素寻址的翻译模式 ……312
8.3.5 赋值语句中的类型转换 ……314
8.3.6 记录域的访问 ……315
8.4 布尔表达式 ……315
8.4.1 翻译布尔表达式的方法 ……316
8.4.2 数值表示 ……316
8.4.3 短路代码 ……317
8.4.4 控制流语句 ……317
8.4.5 布尔表达式的控制流翻译 ……319
8.4.6 混合模式的布尔表达式 ……321
8.5 case语句 ……321
8.6 回填 ……323
8.6.1 布尔表达式 ……323
8.6.2 控制流语句 ……326
8.6.3 翻译的实现方案 ……326
8.6.4 标号和goto ……327
8.7 过程调用 ……328
8.7.1 调用序列 ……328
8.7.2 一个简单的例子 ……328
练习 ……329
参考文献注释 ……331
第9章 代码生成 ……333
9.1 代码生成器设计中的问题 ……333
9.1.1 代码生成器的输入 ……333
9.1.2 目标程序 ……334
9.1.3 存储管理 ……334
9.1.4 指令选择 ……334
9.1.5 寄存器分配 ……335
9.1.6 计算次序的选择 ……336
9.1.7 代码生成方法 ……336
9.2 目标机器 ……336
9.3 运行时存储管理 ……338
9.3.1 静态分配 ……339
9.3.2 栈式分配 ……340
9.3.3 名字的运行地址 ……342
9.4 基本块和流图 ……343
9.4.1 基本块 ……343
9.4.2 基本块的变换 ……344
9.4.3 保结构变换 ……344
9.4.4 代数变换 ……345
9.4.5 流图 ……345
9.4.6 基本块的表示 ……345
9.4.7 循环 ……346
9.5 下次引用信息 ……346
9.5.1 计算下次引用信息 ……346
9.5.2 临时名字的存储分配 ……347
9.6 一个简单的代码生成器 ……347
9.6.1 寄存器描述符和地址描述符 ……348
9.6.2 代码生成算法 ……348
9.6.3 函数*getreg* ……349
9.6.4 为其他类型的语句生成代码 ……350
9.6.5 条件语句 ……351
9.7 寄存器分配与指派 ……351
9.7.1 全局寄存器分配 ……352
9.7.2 引用计数 ……352
9.7.3 外层循环的寄存器指派 ……353
9.7.4 图染色法寄存器分配 ……354
9.8 基本块的dag表示法 ……354
9.8.1 dag的构造 ……355
9.8.2 dag的应用 ……357
9.8.3 数组、指针和过程调用 ……358
9.9 窥孔优化 ……359
9.9.1 冗余加载与保存 ……360
9.9.2 不可达代码 ……360
9.9.3 控制流优化 ……361
9.9.4 代数化简 ……361
9.9.5 强度削弱 ……361

9.9.6 机器语言的使用 ……362
9.10 从dag生成代码……362
9.10.1 重排序 ……362
9.10.2 对dag的启发式排序 ……362
9.10.3 树的最优排序……363
9.10.4 标记算法 ……364
9.10.5 从标记树中产生代码 ……364
9.10.6 多寄存器操作……367
9.10.7 代数性质 ……367
9.10.8 公共子表达式……368
9.11 动态规划代码生成算法 ……368
9.11.1 一种寄存器计算机 ……368
9.11.2 动态规划的原理 ……369
9.11.3 邻近计算 ……369
9.11.4 动态规划算法……369
9.12 代码生成器的生成器 ……371
9.12.1 采用重写树技术的代码生成 ……371
9.12.2 借助语法分析的模式匹配 ……375
9.12.3 用于语义检查的例程 ……376
练习 ……376
参考文献注释 ……378
第10章 代码优化 ……381
10.1 引言……381
10.1.1 代码改进变换的准则 ……381
10.1.2 性能的提高……382
10.1.3 优化编译器的组织 ……383
10.2 优化的主要种类 ……384
10.2.1 保持功能变换……385
10.2.2 公共子表达式……386
10.2.3 复制传播 ……387
10.2.4 无用代码删除……387
10.2.5 循环优化 ……388
10.2.6 代码外提 ……388
10.2.7 归纳变量和强度削弱 ……388
10.3 基本块的优化 ……390
10.4 流图中的循环 ……392
10.4.1 支配节点 ……392
10.4.2 自然循环 ……393
10.4.3 内循环 ……393
10.4.4 前置首节点……394
10.4.5 可约流图 ……394
10.5 全局数据流分析介绍 ……395
10.5.1 点和路径 ……396
10.5.2 到达定义 ……396
10.5.3 结构化程序的数据流分析 ……397
10.5.4 对数据流信息的保守估计 ……399
10.5.5 *in* 和 *out* 的计算 ……400
10.5.6 处理循环 ……401
10.5.7 集合的表示……402
10.5.8 局部到达定义……403
10.5.9 引用-定义链 ……404
10.5.10 计算顺序 ……404
10.5.11 一般控制流 ……404
10.6 数据流方程的迭代解 ……405
10.6.1 到达定义的迭代算法 ……406
10.6.2 可用表达式……408
10.6.3 活跃变量分析……410
10.6.4 定义-引用链 ……411
10.7 代码改进变换 ……412
10.7.1 全局公共子表达式删除 ……412
10.7.2 复制传播 ……413
10.7.3 循环不变计算的检测 ……415
10.7.4 代码外提 ……415
10.7.5 可选的代码外提方案 ……417
10.7.6 代码外提后对数据流信息的维护 ……418
10.7.7 归纳变量删除……418
10.7.8 带有循环不变表达式的归纳变量 ……421
10.8 处理别名 ……422
10.8.1 一种简单的指针语言 ……422
10.8.2 指针赋值的作用 ……422
10.8.3 利用指针信息……424
10.8.4 过程间的数据流分析 ……425
10.8.5 带有过程调用的代码模型 ……425
10.8.6 别名的计算……426
10.8.7 存在过程调用时的数据流分析 …427
10.8.8 change信息的用途 ……428
10.9 结构化流图的数据流分析……429

10.9.1 深度优先搜索 ……429
10.9.2 流图的深度优先表示中的边 ……431
10.9.3 流图的深度 ……431
10.9.4 区间 ……432
10.9.5 区间划分 ……432
10.9.6 区间图 ……433
10.9.7 节点分裂 ……433
10.9.8 T_1-T_2 分析 ……434
10.9.9 区域 ……434
10.9.10 寻找支配节点 ……435
10.10 高效数据流算法 ……436
10.10.1 迭代算法中的深度优先顺序 ……436
10.10.2 基于结构的数据流分析 ……437
10.10.3 对基于结构的算法的一些速度上的改进 ……440
10.10.4 处理不可约流图 ……441
10.11 一个数据流分析工具 ……441
10.11.1 数据流分析框架 ……442
10.11.2 数据流分析框架的公理 ……443
10.11.3 单调性和分配性 ……444
10.11.4 数据流问题的聚合路径解 ……447
10.11.5 流问题的保守解 ……447
10.11.6 通用框架的迭代算法 ……448
10.11.7 一个数据流分析工具 ……448
10.11.8 算法10.18的性质 ……449
10.11.9 算法10.18的收敛性 ……449
10.11.10 初始化的修正 ……450
10.12 类型估计 ……450
10.12.1 处理无穷类型集 ……451
10.12.2 一个简单的类型系统 ……452
10.12.3 前向方案 ……452
10.12.4 后向方案 ……453
10.13 优化代码的符号调试 ……455
10.13.1 基本块中变量值的推断 ……456
10.13.2 全局优化的影响 ……459
10.13.3 归纳变量删除 ……459
10.13.4 全局公共子表达式删除 ……459
10.13.5 代码外提 ……459
练习 ……460
参考文献注释 ……465
第11章 编写一个编译器 ……469
11.1 编译器设计 ……469
11.1.1 源语言问题 ……469
11.1.2 目标语言问题 ……469
11.1.3 性能标准 ……469
11.2 编译器开发方法 ……470
11.3 编译器开发环境 ……472
11.4 测试与维护 ……474
第12章 编译器实例 ……475
12.1 数学排版预处理器EQN ……475
12.2 Pascal编译器 ……475
12.3 C编译器 ……476
12.4 Fortran H编译器 ……477
12.4.1 Fortran H中的代码优化 ……478
12.4.2 代数优化 ……478
12.4.3 寄存器优化 ……478
12.5 BLISS/11编译器 ……479
12.6 Modula-2优化编译器 ……480
附录 一个程序设计项目 ……483
参考文献 ……489
索引 ……511

第1章 编译简介

编写编译器的原理和技术具有十分普遍的意义，以致于在每一个计算机科学家的研究生涯中，本书中的这些原理和技术都会反复用到。编译器的编写涉及到程序设计语言、计算机体系结构、语言理论、算法和软件工程等学科。幸运的是，有几种基本编译器编写技术已经被用于构建许多计算机的多种语言翻译器。本章通过描述编译器的组成、编译器的工作环境以及简化编译器建造过程的软件工具来介绍编译。

1.1 编译器

简单地说，编译器是一个程序，它读入用某种语言（源语言）编写的程序并将其翻译成一个与之等价的以另一种语言（目标语言）编写的程序（如图1-1所示）。作为这个翻译过程的一个重要组成部分，编译器能够向用户报告被编译的源程序中出现的错误。

目前，世界上存在着数千种源语言，既有Fortran和Pascal这样的传统程序设计语言，也有各计算机应用领域中出现的专用语言。目标语言也同样广泛，目标语言可以是另一种程序设计语言或者是从微处理机到超级计算机的任何计算机的机器语言。不同语言需要不同的编译器。根据编译器的构造方法或者它们要实现的功能，编译器被分为一遍编译器、多遍编译器、装入并执行编译器、调试编译器、优化编译器等多种类别。从表面来看，编译器的种类似乎千变万化，多种多样，实质上，任何编译器所要完成的基本任务都是相同的。通过理解这些任务，我们可以利用同样的基本技术为各种各样的源语言和目标机器构建编译器。

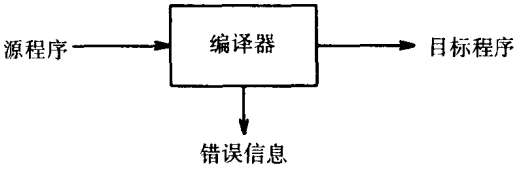


图1-1 编译器

从20世纪50年代早期第一个编译器出现至今，我们所掌握的有关编译器的知识已经得到了长足的发展。我们很难说出第一个编译器出现的准确时间，因为最初的很多实验和实现是由不同的工作小组独立完成的。编译器的早期工作主要集中在如何把算术表达式翻译成机器代码。

整个20世纪50年代，编译器的编写一直被认为是一个极难的问题。比如Fortran的第一个编译器花了18人年才得以实现（Backus et al. [1957]）。目前，我们已经系统地掌握了处理编译期间发生的许多重要任务的技术。良好的实现语言、程序设计环境和软件工具也已经被开发出来。借助于这些先进的技术、环境和工具，一个真正的编译器完全可以作为一个学期的编译器课程的学生实习项目来实现。

1.1.1 编译的分析-综合模型

编译由两部分组成：分析与综合。分析部分将源程序切分成一些基本块并形成源程序的中间表示，综合部分把源程序的中间表示转换为所需的目标程序。在这两部分中，综合部分需要大量的专门化技术。我们将在1.2节非正式地讲解分析部分，在1.3节粗略地讲解标准编译器中目标代码被综合的方法。

在分析期间，源程序所蕴含的操作将被确定下来并被表示成为一个称为语法树的分层结构。语法树的每个节点表示一个操作，该节点的子节点表示这个操作的参数。一个赋值语句的语法

树如图1-2所示。

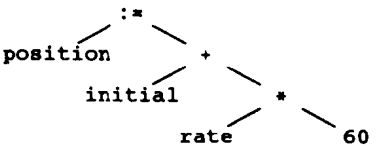


图1-2 position:=initial+rate*60的语法树

许多操纵源程序的软件工具都首先完成某种类型的分析。下边是这类软件工具的示例：

1. 结构编辑器。结构编辑器将一个命令序列作为输入来构造一个源程序。结构编辑器不仅实现普通的文本编辑器的文本创建和修改功能，而且还对程序文本进行分析，为源程序构造恰当的层次结构。结构编辑器能够完成程序准备过程中所需要的功能。例如，它可以检查输入的格式是否正确，能自动地提供关键字（例如，当用户敲入关键字 while 时，编辑器能够自动提供匹配的关键字 do 并提醒用户必须在二者之间插入一个条件体），能够从 begin 或者左括号跳转到与之匹配的 end 或者右括号。这类结构编辑器的输出常常类似于一个编译器的分析阶段的输出。

2. 智能打印机。智能打印机能够对程序进行分析，打印出结构清晰的程序。例如，注释以一种特殊的字体打印；根据各个语句在程序的层次结构中的嵌套深度来缩排这些语句。

3. 静态检查器。静态检查器读入一个程序，分析这个程序，并在不运行这个程序的条件下试图发现程序的潜在错误。静态检查器的分析部分与第10章中将要讨论的优化编译器的分析部分类似。比如，静态检查器可以查出源程序中永远不能被执行的语句，也可以查出变量在被定义以前被引用。另外，利用第6章要讨论的类型检查技术，静态检查器还可以捕获诸如将实型变量用作指针这样的逻辑错误。

4. 解释器。解释器不是通过翻译来产生目标程序，而是直接执行源程序中蕴含的操作。例如，对于一个赋值语句，解释器为之建立一个类似于图1-2的树，然后通过遍历这棵树来执行节点上的操作。在根节点，解释器会发现它有一个赋值操作要完成，因此它调用表达式计算例程去计算赋值操作右端的表达式，然后将结果存放到与标识符 position 相关联的地址。在根节点的右子节点处，表达式计算例程将发现它要计算两个表达式的和。表达式计算例程递归地调用它自身来计算表达式 rate*60 的值，然后将这个值加到变量 initial 的值上。

2 ~ 3

由于命令语言中执行的每个操作通常都是对编辑器或编译器一类复杂例程的调用，解释器经常用于执行命令语言。类似地，一些“非常高级”的语言，如APL,通常都是解释执行的，因为有许多关于数据的信息（如数组的大小和形状）不能在编译时得到。

按照传统的观念，编译器一般被看成是把使用 Fortran 等高级语言编写的源程序翻译成汇编语言或者某种计算机的机器语言的程序。然而，在很多与语言翻译毫不相关的场合，编译技术也常常被使用。下面举出的每一个例子中的分析部分都与传统观念中的编译器的分析部分相似。

1. 文本格式器（text formatter）。文本格式器的输入是一个字符流。输入字符流中的多数字符串是需要排版输出的字符串，同时字符流中也包含一些用来说明字符流中的段落、图表或者上标和下标等数学结构的命令。下一节将介绍一些由文本格式器完成的分析工作。

2. 硅编译器（silicon compiler）。硅编译器的输入是一个源程序，这个源程序的程序设计语言类似于传统的程序设计语言。但是，该语言中的变量不是内存中的地址，而是开关电路中的逻辑符号（0或1）或符号组。硅编译器的输出是一个以适当语言书写的电路设计。关于硅编译器的讨论参见 Johnson[1983]，Ullman[1984] 或者 Trickey[1985]。

3. 查询解释器（query interpreter）。查询解释器把含有关系和布尔运算的谓词翻译成数据库命令，在数据库中查询满足该谓词的记录（参见 Ullman[1982] 或 Date[1986]）。

1.1.2 编译器的前驱与后继

为了建立可执行的目标程序，除了编译器以外，我们还需要几个其他的程序。源程序可能被分成模块存储在不同的文件中。把存储在不同文件中的程序模块集成为一个完整的源程序这个任务由一个称为预处理器的程序完成。预处理器也能够把源程序中称为宏的缩写语句展开为原始语句加入到源程序中。

图1-3给出了一个典型的“编译”过程。由编译器创建的目标程序需要进一步处理才能运行。图1-3中的编译器产生汇编代码，汇编代码需要由汇编器翻译成机器代码，然后与一些库程序连接在一起形成可在计算机上运行的代码。

下两节讨论编译器的组成部件。图1-3中的其他程序将在1.4节讨论。

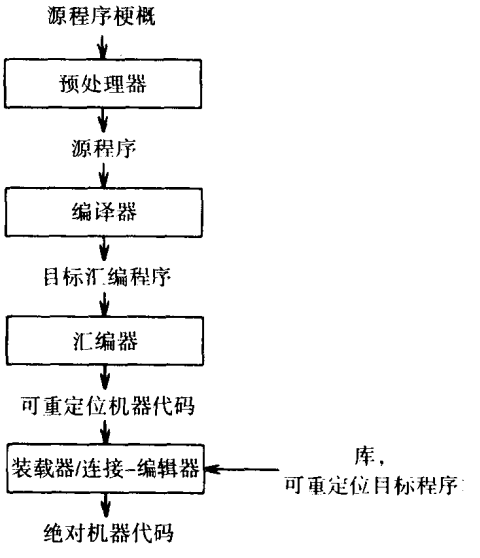


图1-3 一个语言处理系统

1.2 源程序分析

本节介绍编译器的分析过程，并说明它在一些文本格式化语言中的应用。这一论题将在第2、3、4、6章中更详细地讨论。在编译中，源程序的分析过程由如下三个阶段组成：

1. *线性分析*。在线性分析中，从左到右地读构成源程序的字符流，而且把字符流分组为多个*记号*（token），而记号是具有整体含义的字符序列。

2. *层次分析*。在层次分析中，字符串或记号在层次上划分为具有一定层次的多个嵌套组，每个嵌套组具有整体的含义。

3. *语义分析*。在语义分析中要进行某些检查，以确保程序各个组成部分确实是有意义地组合在一起的。

1.2.1 词法分析

在编译器中，线性分析被称为*词法分析*或者*扫描*。例如，在词法分析中，赋值语句 `position := initial + rate * 60` 中的字符将被分组为以下记号组：

1. 标识符 `position`。
2. 赋值符号 `:=`。
3. 标识符 `initial`。
4. 加号 `+`。
5. 标识符`rate`。
6. 乘号 `*`。
7. 数字 `60`。

在词法分析过程中，分隔这些记号的字符的空格将被删除。

1.2.2 语法分析

层次分析被称为*语法分析*（parsing 或者 syntax analysis）。它把源程序的记号进一步分组，产生被编译器用于生成代码的语法短语。通常，源程序的语法短语用图1-4所示的分析树来表示。

在表达式 `position := initial + rate * 60` 中，`rate*60`是一个逻辑单元，因

为通常的关于算术表达式的定律告诉我们乘法比加法先计算。由于表达式initial+rate后有一个*，这个表达式没有被划分成单个短语。

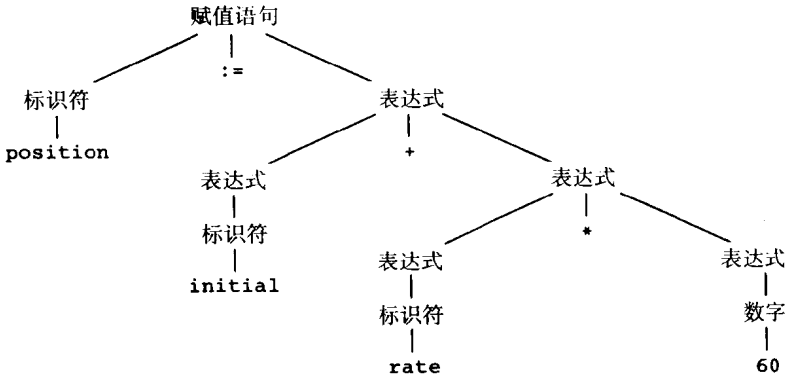


图1-4 position := initial + rate * 60 的分析树

程序的层次结构通常是通过递归规则来表达的。比如，我们可能把下述的规则作为表达式定义的一部分：

1. 任何一个标识符（identifier）都是表达式；
2. 任何一个数（number）都是表达式；
3. 如果$expression_1$和$expression_2$是表达式，则

$expression_1$ + $expression_2$
$expression_1$ * $expression_2$
($expression_1$)

也是表达式。

规则1和规则2是（非递归的）基本规则，而规则3通过将运算符用到其他表达式上递归地定义了表达式。于是，由规则1可知，initial和rate是表达式，由规则2可知，60是表达式，而由规则3我们首先可以知道rate*60是一个表达式，最后initial+rate*60是一个表达式。

6 类似地，许多语言用下列规则来递归地定义语句：

1. 如果 $identifier_1$ 是一个标识符，$expression_2$ 是一个表达式，则

$identifier_1$:= $expression_2$

是一个语句。

2. 如果 $expression_1$ 是一个表达式，$statement_2$ 是一个语句，则

while ($expression_1$) **do** $statement_2$
if ($expression_1$) **then** $statement_2$

也是语句。

词法分析和语法分析的界限在某种程度上是不确定的。我们通常采取能够使整个分析工作简化的方法来设定词法分析与语法分析的界限。决定词法分析和语法分析界限的因素是源语言是否具有递归结构。词法结构不要求递归，而语法结构常常需要递归。上下文无关文法是递归规则的一种形式化，可以用来指导语法分析。第2章和第4章将对上下文无关文法进行详细讨论。

例如，在识别源语言的标识符（由字母开头的字母和数字串）时，我们不需要递归，只要简单地扫描输入流就可以完成标识符的识别。一般地，直到遇见一个既不是字母也不是数字的

字符时为止，在这之前扫描到的字母和数字归结为一个标识符记号，被分组的字符存储到一个称为符号表的表中，并将这些字符从输入中删除以便开始扫描下一个记号。

另一方面，这种线性扫描不具有分析源语言的表达式或语句的能力。例如，不在输入上设置某种类型的层次结构或嵌套结构，我们就不能正确地匹配表达式中的括号或者语句中的begin和end。

图1-4中的分析树刻画了输入的语法结构。该语法结构的更常见的内部表示可以由图1-5a中的语法树给出。语法树是分析树的一种压缩表示，其内节点表示操作符，操作符的操作数是该操作符对应节点的子节点。5.2节将详细讨论图1-5a所示语法树的构造。我们将在第2章和第5章详尽地讨论语法制导翻译。在语法制导翻译中，编译器利用源程序的层次结构生成输出结果。

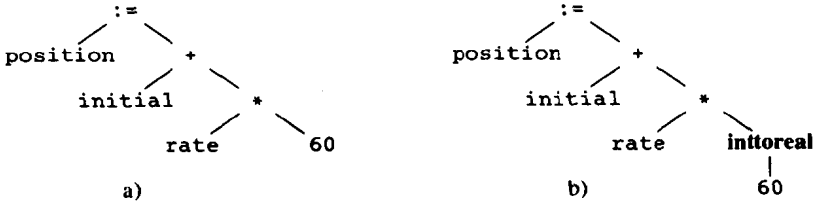


图1-5 语义分析插入了一个整数到实数的转换

1.2.3 语义分析

语义分析阶段检测源程序的语义错误，并收集代码生成阶段要用到的类型信息。语义分析利用语法分析阶段确定的层次结构来识别表达式和语句中的操作符和操作数。

语义分析的一个重要组成部分是类型检查。类型检查负责检验每个操作符的操作数是否满足源语言的说明。例如，很多程序设计语言都要求每当一个实数用于数组的索引时都要报错。程序设计语言可能允许一些操作数的强制类型转换。例如，一个二元算术操作符的操作数可以是一个整数和一个实数。在这种情况下，编译器将把整数强制转换成实数。类型检查和语义分析将在第6章中讨论。

例1.1 在机器内部，整数的二进制表示形式不同于实数的二进制表示形式。两个具有相同数值的整数与实数的机器内部表示也不相同。例如，假定图1-5中的所有标识符都被声明为实数，而60自己却被假定为整数。在对图1-5进行类型检查时编译器会发现*被应用到实数rate和整数60上。一般的解决方法是将整数转换成实数。图1-5b给出了整数转换为实数的方法，即创建一个额外节点 **inttoreal**，显式地将一个整数转换成一个实数。解决类型转换的另一种方法是用一个等值的实数常数来替代整数，因为 **inttoreal** 的操作数是常数。 □

1.2.4 文本格式器中的分析

将文本格式器的输入看成是由多个盒子构成的层次结构的说明是有益的。一个盒子是一个用某种位模式填充的矩形区域，填充的位模式表明该区域被输出设备打印成浅黑像素还是黑像素。

例如，TEX系统(Knuth[1984a])就是这样看待其输入的。非命令行的每个字符都表示一个盒子，其中包含了表示该字符的字体和尺寸的位模式。没有用“空白”（空格或者换行符）隔开的连续字符被识别为一个词，由一列水平排列的盒子构成，如图1-6所示。把字符串识别为词（或者命令）的过程就是文本格式器的线性分析或

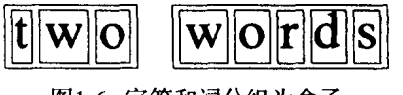


图1-6 字符和词分组为盒子

词法分析部分。

TEX中的一个盒子可以由多个较小盒子在水平和竖直方向上的任意组合来构成。例如，命令

```
\hbox{ <list of boxes> }
```

通过水平排列一列盒子来构成一个新盒子。类似地，命令\vbox通过垂直并置一列盒子来构成一个新盒子。这样，TEX的命令行

```
\hbox{\vbox{! 1} \vbox{@ 2}}
```

将产生图1-7所示的盒子结构。确定输入串中蕴含的盒子的层次结构是TEX语法分析的重要组成部分。

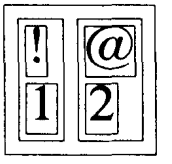


图1-7 TEX中盒子的层次结构

文本格式器的另一个例子是用于数学公式排版的预处理器 EQN（Kernighan and Cherry [1975]）或者 TEX 的数学公式排版处理器。它们使用操作符来构建数学表达式。例如，操作符 sub 和 sup 表示数学公式中的上标和下标。如果 EQN 遇到如下的输入文本

```
BOX sub box
```

它将把*box*的尺寸缩小并置于 *BOX* 的右下角，如图1-8所示。类似地，sup 操作符将 *box* 置于 *BOX* 的右上角。



图1-8 建立数学公式中的下标结构

这些操作符可以嵌套使用。例如，EQN文本

```
a sub {i sup 2}
```

将产生 a_{i^2}。将 EQN 文本中sub和sup操作符分组为记号的工作是 EQN 词法分析的一部分。EQN文本中盒子的大小和位置需要由 EQN 文本的语法结构来确定。

1.3 编译器的各阶段

从概念上讲，编译器是分阶段执行的。每个阶段将源程序从一种表示转换成另一种表示。编译器的一个典型的阶段划分如图1-9所示。我们在1.5节已经提到，实际上编译器的有些阶段可以合并到一起。如果几个阶段已经被合并到一起，则这些阶段的中间表示不需要明确地构造出来。

图1-9中的前三个阶段构成了上节中介绍的编译器的分析部分。图中的符号表管理和错误处理是编译器的六个阶段（词法分析、语法分析、语义分析、中间代码生成、代码优化和代码生成）都要涉及的两项活动。以后，我们也把符号表管理器和错误处理器非正规地称

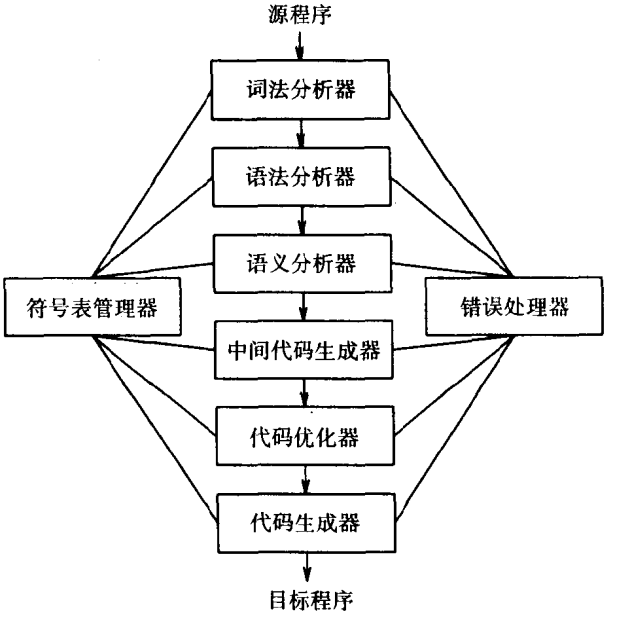


图1-9 编译器的各阶段

为编译器的“阶段”。 10

1.3.1 符号表管理

编译器的一个基本功能是记录源程序中使用的标识符并收集与每个标识符相关的各种属性信息。标识符的属性信息表明了该标识符的存储位置、类型、作用域（在哪段程序中有效）等信息。当一个标识符是过程名时，它的属性信息还包括诸如参数的个数与类型、每个参数的传递方法（如传地址方式）以及返回值的类型等信息。

*符号表*是一个数据结构。每个标识符在符号表中都有一条记录，记录的每个域对应于该标识符的一个属性。这种数据结构允许我们快速地找到每个标识符的记录，并在该记录中快速地存储和检索信息。符号表将在第2章和第7章中详细讨论。

当源程序中的一个标识符被词法分析器识别出来时，词法分析器将在符号表中为该标识符建立一条记录。但是，标识符的属性一般不能在词法分析中确定。例如，在如下的Pascal源程序的声明

```
var position, initial, rate : real ;
```

中，当 position、initial、rate 被词法分析器识别时，它们的数据类型（real）还是未知的。

标识符的属性信息将由词法分析以后的各阶段陆续写入符号表，并以各种方式被使用。例如，当进行语义分析和中间代码生成时，我们需要知道标识符是哪种类型，以便检查源程序是否正确地使用了这些标识符，并在它们之上生成正确的操作。代码生成器将赋予标识符的存储位置信息写入符号表，而且代码生成器还要使用符号表中标识符的存储位置信息。

1.3.2 错误检测与报告

每个阶段都可能遇到错误。各阶段检测到错误以后，必须以恰当的方式进行错误处理，使得编译器能继续运行，以检测出源程序中的更多错误。发现错误即停止运行的编译器不是一个好的编译器。

语法分析和语义分析阶段通常能够处理编译器所能检测到的大部分错误。词法分析阶段能够检测出输入中不能形成源语言任何记号的错误字符串。语法分析阶段可以确定记号流中违反源语言结构（语法）规则的错误。语义分析阶段试图检测出具有正确的语法结构但对操作无意义的部分。例如，我们试图将两个标识符相加，其中一个标识符是数组名，而另一个标识符却 11
是过程名。本书将在详细讨论编译器的各阶段时详细介绍各阶段的错误处理方法。

1.3.3 各分析阶段

随着编译器各个分析阶段的进展，源程序的内部表示不断地发生变化。我们现在通过语句

```
position := initial + rate * 60                    (1-1)
```

的翻译过程来描述源程序内部表示的变化过程。图1-10展示了该语句在每个阶段之后的表示。

词法分析阶段读入源程序中的字符，并将这些字符分组形成记号流，其中每个记号表示一个逻辑上相关的字符序列，如标识符、关键字（if、while等）、标点符号、多字符运算符:=等。形成一个记号的字符序列称为该记号的*词素*（lexeme）。

某些记号需要被赋予“词法值”。例如，当 rate 这样的记号被识别时，词法分析器不但要生成一个记号（比如 **id**），而且如果词素 rate 在符号表中不存在，还要在符号表中为 rate 建立一个记录，将单词 rate 写入到符号表中去。与 id 相关联的词法值指向符号表中 rate 的记录。

在本节，我们将使用 $\mathbf{id}_1$、$\mathbf{id}_2$、$\mathbf{id}_3$ 表示 position、initial、rate，以强调标识符的

内部表示不同于组成标识符的字符序列。显然，语句(1-1)的表示在词法分析后被表示为：

$$id_1 := id_2 + id_3 * 60 \tag{1-2}$$

我们还要为多字符运算符“:=”和数字60建立记号以反映它们的内部表示，但我们将这部分内容推迟到第2章。词法分析将在第3章中详细讨论。

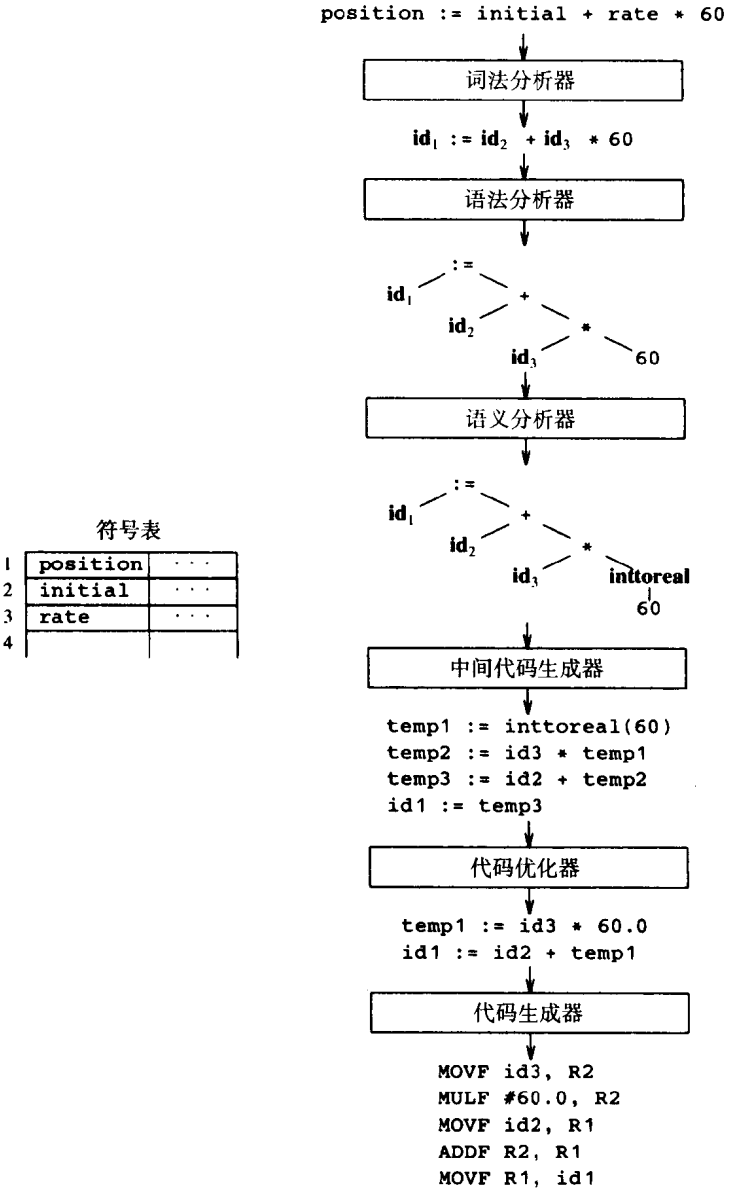
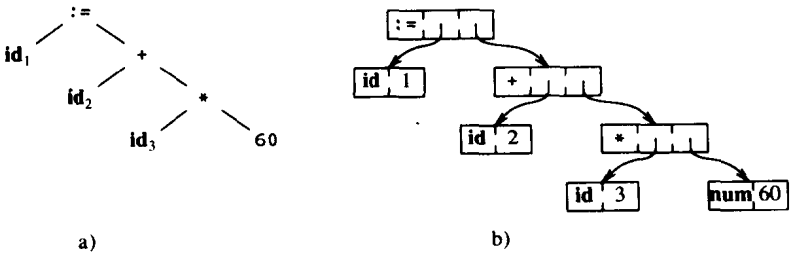


图1-10 一个语句的翻译

我们已经在1.2节介绍了第二阶段和第三阶段，即语法分析和语义分析。语法分析在记号流上建立一个层次结构，我们采用图1-11a中的语法树来表示记号流的层次结构。语法树的一种典型数据结构如图1-11b所示，其中每个内节点是一个记录，每个记录具有三个域，一个域存储操作符，另外两个域存储指向左、右子节点的指针。叶节点是一个具有两个或者更多域的记录，其中一个域用于标识该叶节点上的记号，其他域用于记录该记号的相关信息。我们可以

增加各节点记录的域，以便存储与语言结构相关的附加信息。我们将在第4章和第6章分别讨论语法分析和语义分析。



a) b)

图1-11 b)是a)中树的数据结构

1.3.4 中间代码生成

某些编译器在完成语法分析和语义分析以后，产生源程序的一个显式中间表示。我们可以将这种中间表示看成是某种抽象机的程序。源程序的中间表示应该具有两个重要性质。一是易于产生，二是易于翻译成目标程序。

源程序的中间表示有多种形式。在第8章，源程序的中间表示形式被称为“三地址码”，类似于某种机器的汇编语言，这种机器的每个存储单元的作用类似于寄存器。三地址码由指令序列组成，每个指令最多有三个操作数。式(1-1)中的源程序的三地址码如下：

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```
(1-3)

这种中间表示形式具有以下几个特点。首先，除赋值以外，每个三地址指令最多只有一个操作符。因此，在生成这些指令时，编译器必须确定完成操作的顺序。(1-1)中乘法优先于加法。其次，编译器必须为每条指令产生一个临时变量，用以保存这条指令的计算结果。第三，某些“三地址”指令的操作数少于三个，如(1-3)的第一条指令和第四条指令。

我们将在第8章介绍几种编译器中使用的主要中间表示形式。一般，中间表示不仅仅计算表达式，它们还必须处理控制流结构和过程调用等其他任务。第5章和第8章介绍一些典型程序设计语言结构的中间代码生成算法。

1.3.5 代码优化

代码优化阶段试图改进中间代码，以产生执行速度较快的机器代码。有些编译器几乎没有进行代码优化。例如，生成中间代码(1-3)的一个很自然的算法是对语义分析后的树的每个运算符产生一条指令。当然，还有更好的算法，如使用以下两条指令

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```
(1-4)

也可以完成同样的计算。因为问题可以在代码优化阶段被修正，所以这个简单算法没有什么错误。也就是说，在优化阶段，编译器会推断出，60从整型变为实型表示的转换可以在编译时一次完成多次使用，从而inttoreal操作可以删去；temp3只使用一次，即把它的值传给id1，可以用id1代替temp3，从而(1-3)中的最后一条语句不是必需的，这样就得到了(1-4)的结果。

不同编译器所产生的代码的优化程度差别很大。能够完成很大程度的代码优化的编译器称为“优化编译器”。优化编译器将相当多的时间都消耗在代码优化上。但是，有一些简单优化

方法，它们既可以使目标程序的执行时间得到很大的缩短，又不会使编译速度降低太大。很多这方面的内容将在第9章讨论。第10章将介绍最强大的优化编译器所使用的代码优化技术。

1.3.6 代码生成

编译的最后一个阶段是目标代码生成，生成可重定位的机器代码或者汇编代码。在这一阶段，编译器为源程序定义和使用的变量选择存储单元，并把中间指令翻译成完成相同任务的机器代码指令序列。这个阶段的一个关键问题是变量的寄存器分配。

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
```

每条指令的第一和第二个操作数分别代表源操作数和目的操作数。每条指令中的 F 告诉我们指令处理的是浮点数。第一和第二条指令把地址[⊖]id3 中的内容取出后存入寄存器 R2 中，然后把它乘上实数 60.0。#表示把60.0作为常数处理。第三条指令把 id2 中的内容取出后存入寄存器 R1。第四条指令把寄存器R2和R1中的值相加，并存入寄存器 R1。最后，第五条指令将寄存器 R1的值存入地址 id1。这样，这段代码实现了图1-10中的赋值语句。第9章将详细讨论代码生成。

15

1.4 编译器的伙伴

从图1-3我们可以看到，编译器的输入可能由一个或者多个预处理器产生，而编译器的输出也可能需要进一步的处理才能成为可执行的机器代码。本节讨论编译器的输入和输出进行预处理和后加工的编译器的伙伴。

1.4.1 预处理器

预处理器产生编译器的输入，一般具有以下功能：

1. *宏处理*。预处理器允许用户在源程序中定义宏。宏是被经常使用的较长结构的缩写。

2. *文件包含*。预编译器可以把头文件包含到程序正文中。例如，C语言的预处理器能够用<global.h>文件的内容替代源程序中的语句 #include <global.h>。

3. *“理性”预处理器*。这些处理器能把现代控制流和数据结构化机制添加到比较老式的语言中。例如，如果一种语言没有 while 语句和 if 语句这样的控制结构，理性预处理器可以用内部宏定义向用户提供这类控制结构。

4. *语言扩充*。这类处理器通过大量的内部宏定义来增强语言的能力。例如，Equel 语言（Stonebraker et al. [1976]）是一种嵌套在C语言中的数据库查询语言。以 ## 开始的语句被处理器识别为数据库存取语句（与C语言无关），并被翻译成对例程的过程调用，这些例程完成数据库的存取。

宏处理器处理两种类型的语句：宏定义和宏引用。宏定义由具有惟一性的字符或者关键字来标识，如 define 或 macro。宏定义包含被定义的宏的*名字*和构成其定义的*体*。通常，宏处理器允许宏定义中包含*形式参数*，即将被值替代的符号（这里，“值”是一个字符串）。宏的

⊖ 我们避开了一个重要的问题，即源程序中的标识符的存储分配。我们在第7章将会看到，运行时的存储组织依赖于被编译的语言。存储分配是在中间代码生成或者代码生成时决定的。

引用只需要提供宏名和实在参数。实在参数是形式参数的值。宏处理器用实在参数替代宏体中的形式参数，然后用变换后的宏体替代宏引用本身。

例1.2 1.2节中的TEX排版系统使用了一个一般的宏机制，宏定义具有如下形式

```
\define <宏名><模板>{<体>}
```

其中，宏名（macro name）是以反斜线打头的任意一串字符，模板（template）是任意字符串，具有形如 #1，#2，…，#9的字符串作为形式参数。这些符号也可以任意多次地出现在宏体中。例如，下面的宏定义了对*Journal of the ACM* 的引用： 16

```
\define\JACM #1;#2;#3.
    {{\sl J. ACM} {\bf #1}:#2, pp. #3.}
```

这里，宏名是\JACM，模板是“#1; #2; #3.”，分号将形式参数隔开，最后一个形式参数后紧跟一个点号。该宏的引用必须采用模板的形式，当然任何字符串都可用来代替形式参数[⊖]。于是，宏引用

```
\JACM 17;4;715-728.
```

的结果是

J. ACM **17**:4, pp. 715-728.

宏体的{\sl J. ACM}部分产生斜体“*J. ACM*”，表达式{\bf #1}指明第一个实在参数是黑体的，这个参数用于指明期刊卷号。

TEX 允许在宏 \JACM 定义中使用任意标点或者文本串来将卷号、期号和页码分隔开。我们也可以不用任何标点符号，在这种情况下，TEX将认为每个实在参数是一个单字符或者用{}括起来的串。 □

1.4.2 汇编器

某些编译器产生汇编代码，如前面的(1-5)。汇编代码需要交给汇编器做进一步的处理。也有些编译器能够完成汇编器的工作，产生可重定位的机器代码，交给装配器（loader）或者连接编辑器（link-editor）处理。我们假定读者已具备汇编语言的知识并知道汇编器所做的工作。我们需要回顾一下汇编代码与机器代码之间的关系。

汇编代码是机器代码的容易记忆的形式。汇编代码使用名称而不是二进制代码来表示操作，存储地址也用名称来表示。下边是一个典型的汇编指令序列：

```
MOV a, R1
ADD #2, R1                    (1-6)
MOV R1, b
```

这段代码将地址 a 中的内容移到寄存器 R1，然后与常数2相加（将寄存器 R1中的内容看成定点数加以处理），最后将结果存入地址 b。这段程序计算了 b:=a+2。 17

汇编语言也使用宏工具，汇编语言的宏工具与前面讨论过的那些宏预处理器类似。

⊖ 准确地说应为“几乎任何字符串”，因为我们只是简单地自左向右扫描宏引用，而且只要在模板中发现一个匹配#*i*符号的文本，我们就认为前面的字符串匹配#*i*。因此，如果我们要用#1来代替ab;cd，我们就会发现只有ab与#1匹配，而cd与#2匹配。

1.4.3 两遍汇编

最简单的汇编器对输入汇编源程序文件进行两遍扫描，每遍读输入文件一次。在第一遍扫描中，表示存储单元的所有标识符都被识别出来并存入符号表（汇编器的符号表与编译器的符号表是分开的）。一个标识符的存储单元是在它第一次被遇到时分配的。例如，读入(1-6)后，符号表中的内容如图1-12所示。在图1-12中，我们假定每个标识符占四个字节（构成一个字）的存储空间，并且地址从0开始分配。

标识符	地址
a	0
b	4

图1-12 汇编器的符号表

在第二遍扫描中，汇编器再一次从头扫描输入文件。这一次，它将每个操作符翻译成机器语言中代表相应操作的二进制位序列，将代表存储单元的每个标识符翻译成符号表中该标识符的地址。

第二遍扫描的输出是可重定位的机器代码。可重定位指的是装入的起始地址可以是任意的内存单元 L。也就是说，如果将L加到代码的所有地址上，整个程序对所有存储地址的引用都是正确的。为此，汇编器的输出必须说明哪些指令中引用了可重定位地址。

例1.3 假定下面是汇编器把(1-6)中的汇编指令翻译成的机器代码：

```
0001 01 00 00000000 *
0011 01 10 00000010                    (1-7)
0010 01 00 00000100 *
```

我们假想了一个很小的指令字：指令字的前四位是指令代码，0001、0010、0011分别代表操作装入、存储、加。装入表示把内存单元中的数据移到寄存器，存储表示把寄存器中的数据移到内存单元，加代表加运算。指令字的第5、6位用于指定寄存器。在上面的三条指令中，第
18 5、6位皆为01，均指寄存器1。指令字的第7、8位是“标志”位，00代表普通的地址模式，即最后8位是内存地址；10代表“立即”模式，即最后8位是操作数，这个模式出现在(1-7)的第二条指令中。

我们还可以看到(1-7)中的第一条和第三条指令中都含有*号。*号是重定位标志，与可重定位机器代码的各操作数有关。假定存放数据的地址空间的起始地址是 L，有*号的地方表示L必须加到那条指令的地址上。因此，如果 L = 00001111，即15，那么 a 和 b 的地址分别是15和19，(1-7)中的指令将变为

```
0001 01 00 00001111
0011 01 10 00000010                    (1-8)
0010 01 00 00010011
```

(1-8)中的机器代码是绝对（或不可重定位）机器代码。注意，(1-7)的第二条指令没有*号与之相连，所以 L 没有加到它的地址上。这是完全正确的，因为这8位代表的是常数2，而不是存储地址2。 □

1.4.4 装配器和连接编辑器

通常，装配器完成程序的装入和连接编辑两项功能。装入过程包括读入可重定位机器代码，修改可重定位地址（如例1.3中讨论的那样），并将修改后的指令和数据放到内存中适当的位置。

连接编辑器允许我们将多个可重定位机器代码的文件组装成一个程序。这些可重定位机器代码的文件可以是多次编译的结果，其中一个或多个可能是库程序文件。库程序文件是由系统提供的，可以被任何程序使用。

如果这些可重定位机器代码的文件以某种有用的方式组合在一起使用，则就可能出现外部引用，即一个文件中的代码引用另一个文件中的存储单元。这种引用可以是对定义在一个文件中而用于另一个文件的数据单元的引用，或者是对出现在一个文件的代码中而在另一个文件的代码中被调用的过程的入口点的引用。可重定位的机器代码文件必须把每个外部引用的数据单元或者指令标号的信息都保存在符号表中。由于事先并不知道哪些信息将被外部引用，所以，事实上必须将汇编器的整个符号表作为可重定位机器代码的一部分。

例如，(1-7)中的代码前面应该附有下面这张符号表

```
a    0
b    4
```

如果一个与(1-7)的代码文件一起装配的文件引用了b，则该引用被替代为4加上(1-7)的代码文件的数据单元被重定位时存储地址的偏移量。 19

1.5 编译器各阶段的分组

1.3节介绍的编译器的各个阶段是编译器的逻辑组织。在编译器的实际实现中，多个阶段的任务可能被组合在一起。

1.5.1 前端与后端

编译的多个阶段可以分为*前端*和*后端*两个大的阶段。前端包括依赖于源语言并在很大程度上独立于目标机器的某些阶段或者某些阶段的某些部分。前端一般包括词法分析、语法分析、符号表的建立、语义分析、中间代码生成以及相关的错误处理。相当一部分代码优化工作也在前端完成。

后端包括编译器中依赖于目标机器的阶段或某些阶段的某些部分。一般来说，后端完成的任务不依赖于源语言而只依赖于中间语言。后端主要包括代码优化、代码生成以及相关的错误处理和符号表操作。

为不同的机器编写相同源语言的编译器时，人们通常采取如下的方法：首先为所有的机器编写相同的编译器前端或者采用已有的编译器前端，然后为每个机器编写编译器的后端。如果编译器的后端是精心设计的，对于不同的机器不需要做很大的重新设计。这些问题将在第9章中讨论。我们可以将不同的源语言编译成同一中间语言，对不同的前端使用相同的后端，从而得到同一机器上的不同编译器。这是很迷人的工作。但是，由于不同语言的着眼点不同，这方面的研究只取得了有限的成果。

1.5.2 编译器的遍

编译的若干个阶段通常是以*一遍*来实现的，每遍读一次输入文件、产生一个输出文件。编译器的阶段组合为遍的方式千差万别，因此我们趋向于按阶段而不是按遍来讨论编译器。第12章要讨论一些有代表性的编译器并提到它们的阶段被组合为遍的方式。

如上所述，多个编译阶段可以被组合为编译的一遍，并且每一遍中的各编译阶段的工作是相互交错的。例如，词法分析、语法分析、语义分析以及中间代码的生成可以被组合为一遍。这样，词法分析形成的记号流可以被直接翻译成中间代码。更详细地说，我们可以认为该编译遍是在语法分析器的管理下进行的。语法分析器根据它读到的记号识别语法结构。当它需要下一个记号时，它通过调用词法分析器获得所需的记号。一旦语法结构找出来了，语法分析器就 20
调用中间代码生成器完成语义分析并生成中间代码的一部分。第2章给出了一个以这种方式组织的编译器。

1.5.3 减少编译的遍数

一方面，我们希望编译的遍数越少越好，以减少读写中间文件的时间开销。另一方面，如果我们将多个阶段组合为一遍，我们将不得不把这些阶段的整个程序保存在内存中，因为每个阶段需要的信息的顺序可能与前面各阶段产生这些信息的顺序不同。程序的内部形式可能远远大于源程序或者目标程序。所以，空间可能是一个很大的问题。

把某些阶段组合为一遍存在一些问题。例如，词法分析器和语法分析器间的接口通常被限制于单个记号。另一方面，在中间表示完全生成之前，要想完成代码生成是非常困难的。例如，PL/I 和 Algol 68 之类的语言允许变量在声明之前被使用。这样，如果我们不知道一个结构中的变量的类型，就不能生成这个结构的代码。类似地，多数的语言都允许向前跳转的goto语句。在一个goto语句转向到的源程序语句的目标代码被生成之前，我们无法确定这个goto语句的转移地址。

在某些情况下，可以为某些尚不知晓的信息留下空白位置，待获得这些信息后再填上这些空白位置。我们可以通过使用“回填”技术，把中间代码生成和目标代码生成划归到一遍中。但是在第8章讲述中间代码生成之前，我们不能清楚地解释所有的细节。我们可以使用汇编器来说明“回填”技术。我们在前一节曾讨论过一个两遍汇编器：第一遍识别出表示内存位置的标识符并为它们分配存储地址，第二遍用地址代替标识符。

我们可以按下面的方式把汇编器的两遍结合在一起。当遇到一个前向跳转的汇编语句

```
GOTO target
```

时，汇编器生成一个框架性指令，其中包含了 GOTO 的机器操作代码以及为 target 的地址保留的空白位置。所有为target的地址保留了空白位置的指令被保存在一个列表中，该列表与符号表中与target相关的记录相关联。当我们遇到指令

```
target: MOV foobar, R1
```

并确定了 target 的值（它是当前指令的地址）时，进行 target 地址的回填，即顺序扫描所
21 有需要 target 地址的指令的列表，用 target 的地址代替这些指令的地址域中的空白。如果具有空白地址的指令在内存中能够保存到所有目标地址都被确定下来，则这种方法很容易实现。

如果一个汇编器能够保证它的所有输出都保存在内存中，上述方法是合理的。由于对于汇编器来说代码的中间表示和最终表示大体一样，长度也基本相同，所以在整个汇编程序长度范围内的回填并非不可能。然而在编译器中，由于中间代码需要消耗大量空间，我们必须仔细考虑回填的范围。

1.6 编译器的构造工具

与任何程序员一样，编译器的编写者使用调试器、版本管理器、描述器（profiler）等软件工具是十分有益的。第11章将讨论这类软件工具是如何用于实现编译器的。除了这些软件开发工具以外，其他辅助实现编译器各个阶段的专用软件工具也已经被开发出来。本节简单地介绍这些专用软件工具，在相应的章节将详细讨论它们。

当第一批编译器被编写出来不久，用于辅助编译器编写过程的系统就出现了。这些系统通常被人们称为*编译器的编译器*、*编译器生成器*或者*翻译器编写系统*。多数这样的系统都以某种特定的语言模型为基础，适用于产生类似于该语言模型的语言的编译器。

例如，人们曾提出一个迷人的假定：所有语言的词法分析器除了对特殊的关键字和符号的

识别以外基本都是一样的。实际上，许多编译器的编译器确实生成很多用于被生成编译器的固定的词法分析器。这些程序的区别仅仅在于它们识别的关键字表不同。关键字表是需要用户提供的全部信息。这种方法是有效的，但是这种方法不能用于识别非标准的记号，比如含有非字母数字符号的标识符。

人们已经设计出了一些自动设计编译器特定构件的软件工具。这些工具使用了特殊的语言来说明和实现特定程序设计语言构件。很多工具使用了非常复杂的算法。成功的工具都把生成算法的细节隐藏起来而且所产生的构件很容易与编译器的其他构件集成在一起。下面是一些有用的编译器的构造工具。

1. *分析器生成器*。这类工具生成的语法分析器一般都以上下文无关文法为基础。在早期的编译器中，语法分析不仅占据了整个编译器运行时间的相当大部分，而且在编译器的编写工作中也占了相当大的比重。现在，使用分析器生成器工具，这个阶段已经非常容易实现。如果使用4.7节中描述的分析器生成器，本书中出现的许多“小型语言”，比如PIC(Kernighan[1982]), 22
可以在几天内实现。许多分析器生成器利用了功能强大的分析算法。这些算法非常复杂，单靠手工是无法完成的。

2. *扫描器生成器*。这类工具一般都根据以正规表达式为基础的说明自动生成词法分析器。正规表达式将在第3章讨论。这类工具产生的词法分析器的基础组织结构等效于有穷自动机。一种典型的扫描器生成器以及它的实现将在3.5节和3.8节讨论。

3. *语法制导翻译引擎*。这类工具产生一系列的翻译程序，这些翻译程序遍历图1-4那样的分析树，并在遍历分析树的同时产生中间代码。这类工具的基本思想是为分析树的每个节点关联一个或多个“翻译”，每个翻译都由树中该节点的邻节点上的翻译来定义。这样的引擎将在第5章中讨论。

4. *自动代码生成器*。这类工具以一个规则集合为输入，这些规则定义了中间语言的每个操作到目标机器的机器语言的翻译。这些规则必须包含足够详尽的信息以便我们能够处理数据的不同存取方法，比如变量可能存储在寄存器中，也可能在内存的固定存储单元中，还可能存储在堆栈的某个位置上。这类工具使用的基本技术是“模板匹配”。中间代码语句被表示机器指令序列的“模板”替代。由于变量的存放位置多种多样（比如可存放于一个或多个寄存器或存放于内存），对于给定的模板集存在很多可能的方式来替代中间代码。这样，我们有必要选择一种好的替代方式使得在编译器运行时不会产生组合爆炸。这类工具将在第9章中介绍。

5. *数据流引擎*。完成高质量代码优化所需要的很多信息都包含“数据流分析”。数据流分析收集有关值如何从程序的一个部分传递到程序的其他部分的信息。不同的任务均可由基本相同的程序来完成，只需用户提供中间代码语句与被收集信息之间关系的细节。这类工具将在10.11节中介绍。

参考文献注释

1962年 Knuth[1962] 在撰写编译器编写的历史时就说过，“在这一领域，很多技术是由很多独立工作的研究者同时发现的。”他还提到确实有多人独立地发现了“同一技术的各个方面，而且经过多年的精雕细刻已经变成了完美的算法。这一点还没有被创始人们意识到。”为编译技术论功是一项很难的任务。本书的参考文献注释只是为读者进一步研究编译技术提供帮助。 23

有关 Fortran 语言诞生之前的程序设计语言和编译器的历史可以参见Knuth and Trabb Pardo[1977]。Wexelblat[1981] 包括多种程序设计语言历史的回顾，这些回顾是由这些语言的

开发者参与撰写的。

一些有关编译的早期基础性论文已被收录在Rosen[1967]和Pollack[1972]中。1961年1月出版的《*Communications of the ACM*》给出了当时编译器编写情况的一个综述。Randell and Russell[1964]介绍了早期Algol 60编译器的细节。

由20世纪60年代初的语法研究开始，理论研究对编译器技术的开发产生了深远的影响，这种影响不亚于它在计算机科学的其他领域的影响。语法分析的魅力或许早已褪色，但编译技术一直是一个活跃的研究课题。当我们在以后各章中详细考察编译技术时，我们就会清楚地看到
24 编译技术的丰富研究成果。

第2章 简单的一遍编译器

这一章是本书第3章到第8章的内容简介。本章通过开发一个把中缀表达式转换成后缀表达式的C程序来展示一些基本编译技术。本章重点描述编译器的前端部分，即词法分析、语法分析和中间代码生成。第9章和第10章讲述代码生成和代码优化。

2.1 概述

程序设计语言可以通过描述以下两个方面来定义：第一方面是程序模式，即语言的语法；第二方面是程序含义，即语言的语义。为了说明语言的语法，我们介绍一种广为使用的表示法：上下文无关文法或者 BNF（Backus-Naur 范式）。使用现有的表示法描述语言的语义要比描述语言的语法难得多。因此，在定义语言的语义时，我们将使用非形式化方法和启发性实例。

上下文无关文法除了可以用于定义语言的语法之外，还可用于指导源程序的翻译。面向语法的编译技术，如语法制导翻译技术，对于组织编译器的前端十分有用，本章将广泛应用语法制导翻译技术。

在讨论语法制导翻译技术的过程中，我们将构造一个把中缀表达式转换成后缀表达式的编译器。在后缀表达式中，操作符出现在操作数的后面。例如，中缀表达式9-5+2的后缀表达形式为95-2+。我们可以使用一个堆栈把后缀表达式直接转换成计算该表达式的计算机代码。作为起步，我们先构造一个简单的程序，这个程序把由加号和减号分隔的数字所构成的表达式转换成后缀形式。在基本概念清晰以后，我们扩展该程序使之能够处理更一般化的程序设计语言结构。本书中每个功能较强的翻译器都是通过扩展功能较弱的翻译器而得到的。

在我们的编译器里，词法分析器首先把输入字符流转换成记号流。然后，记号流作为下一个阶段的输入，产生源程序的中间表示，这个过程如图2-1所示。图中的“语法制导翻译器”由一个语法分析器和一个中间代码生成器构成。我们之所以把由数字和操作符组成的表达式作为起点是为了使最初的词法分析简单，即每个输入字符就是一个记号。以后，我们将扩展该语言，使其包含数、标识符、关键字等复杂词法结构。对于这个扩展的语言，我们将构造一个词法分析器用于扫描连续的输入字符并将其转换成适当的记号。词法分析器的构造将在第3章详细讨论。

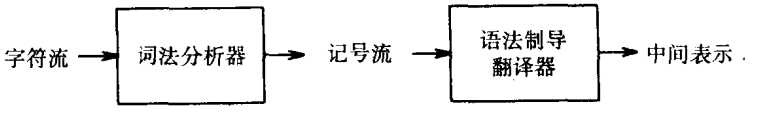


图2-1 我们的编译器前端的结构

2.2 语法定义

本节介绍一种定义语言语法的表示法，称为上下文无关文法（简称文法）。上下文无关文法将贯穿本书始末，作为编译器前端定义的一部分。

一个语法非常自然地描述了许多程序设计语言结构的层次结构。例如，C语言中的 if-else

语句具有如下形式：

if (表达式) 语句 **else** 语句

也就是说，整个语句是由关键字 **if**、一个左括号、一个表达式、一个右括号、一条语句、关键字 **else** 和另外一条语句组成的序列（在C语言中，没有关键字 **then**）。如果使用变量 *expr* 来标识表达式，使用变量 *stmt* 来标识一条语句，则 if-else 语句的构造规则可以表达为

stmt → **if** (*expr*) *stmt* **else** *stmt* (2-1)

这里，箭头可以读作“可以具有形式”。这样的规则称为产生式（production）。在一个产生式中，像关键字 **if** 和括号这样的词法元素称为记号（token），像 *expr* 和 *stmt* 这样的变量表示一个记号序列，并称之为非终结符（nonterminal）。

上下文无关文法包含如下四个部分：

26 1. 一个记号集合，称为终结符号。

2. 一个非终结符集合。

3. 一个产生式集合。每个产生式具有一个左部和一个右部，左部和右部由箭头连接，左部是一个非终结符。右部是记号和（或）非终结符序列。

4. 一个开始符号。开始符号是一个指定的非终结符。

我们约定，定义语法时只需列出文法的产生式，并把以开始符号为左部的产生式列在最前面。在以后的讨论中我们假设：数字、类似于<=的符号、类似于 **while** 的黑体字符串均为终结符，斜体名字表示非终结符，任何非斜体的名字或者符号都是记号[⊖]。为了表示上的方便，我们常把具有相同左部的产生式合并，写成一个产生式，其左部为所有产生式共有的那个非终结符，右部为所有产生式右部的组合，每个右部用“|”分隔。“|”读做“或”。

例2.1 本章的例子均使用由数字、加号和减号组成的表达式，如9-5+2、3-1、7等。因为加号和减号必须出现在两个数字之间，我们称这样的表达式为“用加号和减号分隔的数字序列”。下面的文法描述了这些表达式的语法。产生式为

list → *list* + *digit* (2-2)
list → *list* - *digit* (2-3)
list → *digit* (2-4)
digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** (2-5)

左部的非终结符皆为 *list*（列表）的三个产生式的右部可以合并成：

list → *list* + *digit* | *list* - *digit* | *digit*

按照我们前边的约定，文法的记号是下列符号：

+ - 0 1 2 3 4 5 6 7 8 9

斜体词 *list* 和 *digit* 是非终结符，*list* 是开始非终结符，因为它所对应的产生式列在最前面。□

如果一个非终结符出现在一个产生式的左部，该产生式称为该非终结符的产生式。记号串是零个或者多个记号的序列。一个包含零个记号的记号串称为空串，记为ϵ。

27 从开始符号出发，反复替代产生式中的非终结符（用该非终结符的产生式的右部），一个

⊖ 单个斜体字母在第4章详细研究文法时将用于另外的目的。例如，我们将用*X*，*Y*和*Z*来表示记号或非终结符。任何包含两个或多个字符的斜体名字仍然代表非终结符。

文法可产生一个串。由一个文法的开始符号产生的记号串形成了该文法定义的语言。

例2.2 由例2.1的文法定义的语言是由加号和减号分隔的数字序列。

非终结符 *digit* 的10个产生式表示 *digit* 可以代表0, 1, …, 9中的任何记号。产生式(2-4)表明单个数字本身是一个列表。产生式(2-2)和(2-3)表示：当我们碰到一个列表后面跟着一个加号或者减号，随后是另外一个数字，则我们得到了一个新的列表。

显然，产生式(2-2)到产生式(2-5)足以定义我们需要的语言。例如，我们可以按下面的方法推导出9-5+2是一个 *list*：

a) 由于9是一个 *digit*，根据产生式(2-4)，9是一个 *list*；

b) 由于9是一个 *list*，5是一个 *digit*，，根据产生式(2-3)，9-5是一个 *list*；

c) 由于9-5是一个 *list*，2是一个 *digit*，，根据产生式(2-2)，9-5+2是一个 *list*。

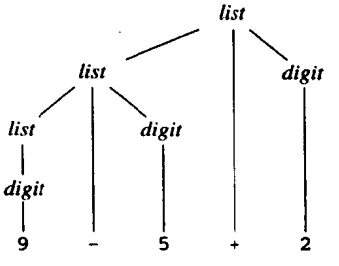


图2-2 与例2.1中文法对应的9-5+2的分析树

上述推理过程如图2-2中的树所示。树中的每个节点用一个文法符号标记。一个内节点和它的所有子节点对应一个产生式。内节点对应产生式的左部，子节点对应产生式的右部。这样的树称作分析树，后面将进一步详细讨论。 □

例2.3 Pascal 语言的 begin-end 语句块（block）是由分号分隔的语句序列。begin-end 语句块的文法结构类似于例2.1中的列表，差别仅在于 **begin** 和 **end** 之间允许有空语句。我们从以下的产生式开始，开发 begin-end 语句块的文法：

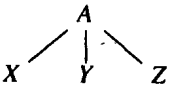
$$
\begin{aligned}
\mathit{block} &\rightarrow \textbf{begin}\ \mathit{opt_stmts}\ \textbf{end} \\
\mathit{opt_stmts} &\rightarrow \mathit{stmt_list} \mid \epsilon \\
\mathit{stmt_list} &\rightarrow \mathit{stmt_list}\ ;\ \mathit{stmt} \mid \mathit{stmt}
\end{aligned}
$$

28

请注意，*opt_stmts* 右部的第二个可选项ϵ表示空符号串，即 *opt_stmts* 可以用空串代替。于是 *block* 可以只由 **begin** 和 **end** 这两个记号构成。*stmt_list* 的产生式类似于例2.1中 *list* 的产生式，分号对应于算术操作符，*stmt* 对应于 *digit*。我们还没有写出 *stmt* 的产生式。稍后，我们将讨论不同语句对应的产生式，如 if 语句、赋值语句等等。 □

2.2.1 分析树

分析树描绘了如何从文法的开始符号开始推导出它的语言中的一个语句。如果非终结符*A*具有一个产生式 $A \rightarrow XYZ$，则*A*的一棵分析树如右图所示，内节点标记为*A*，*A*的三个子节点从左到右分别标记为 *X*、*Y* 和 *Z*。



形式地说，给定一个上下文无关文法，分析树是具有如下特性的树：

1. 树根标记为开始符号。
2. 每个叶节点由记号或者ϵ标记。
3. 每个内节点由一个非终结符标记。
4. 如果*A*是某个内节点的非终结符号标记，$X_1, X_2, \cdots, X_n$ 是该节点从左到右排列的所有子节点的标记，则 $A \rightarrow X_1X_2\cdots X_n$ 是一个产生式。这里，$X_1, X_2, \cdots, X_n$ 是一个终结符或非终结符。对于 $A \rightarrow \epsilon$，分析树中标记为*A*的节点只有一个标记为ϵ的子节点。

例2.4 在图2-2中，根节点的标记是例2.1中文法的开始符号 *list*。它的子节点从左到右分

别标记为 *list*、+ 和 *digit*。注意，

list → *list* + *digit*

是例2.1中文法的一个产生式。根节点的左子节点重复着相同的模式，只是+号变成-号。图中三个标记为 *digit* 的节点都有一个标记为数字的子节点。 □

一棵分析树从左到右的叶节点是这棵分析树生成的结果。分析树生成的结果是由根节点的
29 非终结符生成或导出的串。图2-2的分析树生成的串是 9-5+2。在图2-2中，所有的叶节点都在最底层。今后，我们不必如此排列叶节点。任何树的叶节点都满足从左到右排列的自然顺序，即如果 *a* 和 *b* 具有相同的父节点，且 *a* 在 *b* 的左部，则 *a* 和 *a* 的所有后代都在 *b* 和 *b* 的所有后代的左部。

使用分析树的概念，我们可以定义：一个文法生成的语言是它的某个分析树生成的串的集合。为给定的记号串找到一个分析树的过程称为这个串的语法分析（parsing）。

2.2.2 二义性

我们在谈及根据文法建立的串的某个结构时必须谨慎。一棵分析树读完它的叶节点只能生成惟一的一个串，但是，一个文法可能有多棵分析树生成相同的记号串。这样的文法称为具有二义性的文法。判断一个文法是否具有二义性，我们只需检查是否存在一个具有多棵分析树的记号串。由于具有多棵分析树的记号串通常具有多种含义，所以在构造程序设计语言及其编译器时，我们需要设计无二义性文法，或者使用增加了额外的规则解决二义性问题的二义性文法。

例2.5 如果不区分例2.1中的*digit*和*list*，例2.1中的文法可以改写成如下形式：

string → *string* + *string* | *string* - *string* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

把*digit*和*list*的概念合成一个非终结符串*string*是有意义的，因为一个 *digit* 是 *list* 的特例。

从图2-3中可以看到，表达式9-5+2现在有了不止一棵分析树，分别对应着不同的带括号表达式(9-5)+2和9-(5+2)。这两个表达式的值是不同的，分别是6和2。例2.1的文法不允许这样的解释。 □

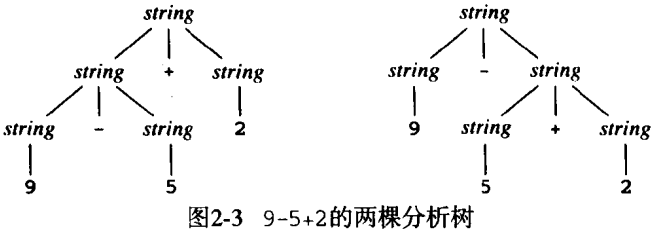


图2-3 9-5+2的两棵分析树

2.2.3 操作符的结合规则

依照惯例，9+5+2和(9+5)+2相等，9-5-2和(9-5)-2相等。在上述表达式中，操作数5左右两侧都有操作符，需要决定哪个操作符使用该操作数。我们说操作符“+”是左结合的，因为当一个操作数左右两侧都有“+”号时，它将被其左部操作符使用。在大多数的程序设计语言中，加、减、乘、除四种算术操作符都是左结合的。

某些常用操作符是右结合的，如指数操作。C语言中的赋值运算操作符“=”号也是右结合的。例如，在C语言中，表示式a=b=c等价于a=(b=c)。

30 由右结合的操作符构成的串，如a=b=c，可以由如下文法产生：

right → *letter* = *right* | *letter*

letter → a | b | · · · | z

图2-4给出了左结合操作符“-”的分析树和右结合操作符“=”的分析树的比较。可以看到，9-5-2分析树向左下端延伸，而a=b=c的分析树向右下端延伸。

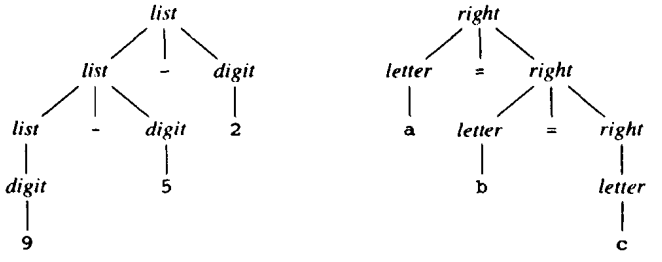


图2-4 左结合操作符和右结合操作符的分析树

2.2.4 操作符的优先级

考虑表达式9+5*2。该表达式有两种可能的解释，即(9+5)*2或者9+(5*2)。+号和*号的结合性无法解决这种二义性。为此，当不止一种操作符出现的时候，我们需要确定操作符之间的优先关系。

如果*在+之前计算，我们说*比+具有更高的优先级。在普通的算术运算中，乘法和除法比加法和减法具有较高的优先级。因此，在表达式9+5*2和9*5+2中，操作数5都首先参与*运算，分别等价于表达式9+(5*2)和(9*5)+2。

表达式的语法。算术表达式的文法可以根据操作符的结合性和优先级表来构建。我们从四个常用的算术操作符和一个优先级表开始说起。在优先级表中，操作符按照优先级递增的次序排列，相同优先级的操作符出现在同一行上： 31

左结合： + -
左结合： * /

我们使用两个非终结符 *expr* 和 *term* 分别表示两个不同的优先级层次，使用另一个非终结符 *factor* 产生表达式中的基本单元。表达式中的基本单元是数字和带括号的表达式。*factor* 的产生式如下：

factor → **digit** | (*expr*)

现在我们考虑具有最高优先级的二元操作符*和/。由于这些操作符是左结合的，其产生式和左结合的*list*的产生式类似：

term → *term* * *factor*
| *term* / *factor*
| *factor*

类似地，*expr* 生成由其他操作符分隔的 *term* 表：

expr → *expr* + *term*
| *expr* - *term*
| *term*

算术表达式的最终文法为

expr → *expr* + *term* | *expr* - *term* | *term*

term → *term* * *factor* | *term* / *factor* | *factor*
factor → **digit** | (*expr*)

该文法把一个表达式看作是由+号和-号分隔的 *term* 表，把 *term* 看成是是由*号或/号分隔的 *factor* 表。任何带括号的表达式都是一个 *factor*。使用括号，我们可以构造具有任意嵌套深度的表达式（及具有任意深度的分析树）。

语句的语法。在多数语言中，我们可以使用关键字识别语句。除了赋值语句和过程调用语句以外，所有的 Pascal 语句都由一个关键字开始。一些 Pascal 语句可以用下面的二义性文法来定义，其中 **id** 表示一个标识符：

stmt → **id** := *expr*
| **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| **while** *expr* **do** *stmt*
| **begin** *opt_stmts* **end**

使用例2.3中的产生式，非终结符 *opt_stmts* 可以产生一个由分号隔开的（可以是空的）语
32 句表。

2.3 语法制导翻译

为了翻译程序设计语言的某个结构，除了为该结构生成的代码以外，编译器还需要保存许多信息。例如，编译器可能需要知道这个结构的类型、目标代码中第 1 条指令的位置、生成的指令个数等等。我们抽象地称这些信息为与该结构相关的*属性*。属性可以表示任意的信息，如类型、串、内存位置等。

本节给出一种称为*语法制导定义*的形式化方法，用以说明程序设计语言中各种结构的翻译。一个语法制导定义根据与其语义部分相关联的属性说明了程序设计语言的一个结构的翻译。在以后的章节里，语法制导定义将用于说明很多发生在编译器前端的翻译。

我们还要介绍一个更加过程化的概念，叫做*翻译模式*，用来描述翻译过程。本章我们将翻译模式用于把中缀表达式翻译成后缀表达式。第5章将详细地讨论语法制导定义及其实现方法。

2.3.1 后缀表示

一个表达式 E 的*后缀表示*可以归纳地定义如下：

1. 如果 E 是一个变量或者常量，则 E 的后缀表示是 E 本身。

2. 如果 E 是形如 $E_1\ op\ E_2$ 的表达式，其中 op 是一个二元操作符，则 E 的后缀表示是 $E_1'\ E_2'\ op$，这里 E_1' 和 E_2' 分别是 E_1 和 E_2 的后缀表示。

3. 如果 E 是形如 (E_1) 的表达式，则 E_1 的后缀表示是 E 的后缀表示。

因为一个表达式的操作符的位置和每个操作符的操作数的个数（参数数量）只允许后缀表达式的一种解码方式，所以在后缀表示中不需要括号。例如，(9-5)+2 的后缀表示是 95-2+，9-(5+2)的后缀表示是 952+-。

2.3.2 语法制导定义

*语法制导定义*使用上下文无关文法来说明输入的语法结构。它通过每个文法符号和一个属性集合相关联，通过每一个产生式和一个*语义规则*集合相关联。语义规则用来计算与产生式中出现的符号相关联的属性的值。文法和语义规则集合构成了语法制导定义。

33 翻译是一个输入到输出的映射。每个输入 x 的输出用下面的方式来说明。首先，构建x的

分析树。假定分析树的节点 n 用文法符号 X 标识。我们用 $X.a$ 表示节点 n 上 X 的属性 a 的值。节点 n 上的 $X.a$ 的值是使用与 X 产生式相关联的属性 a 的语义规则来计算的。每个节点都具有属性值的分析树称为*注释分析树*。

2.3.3 综合属性

如果分析树的某个节点的属性值是由其子节点的属性值确定的，则我们称该属性为综合属性。一棵分析树的所有综合属性值的计算只需要分析树的一次自底向上遍历。本章只使用综合属性，继承属性将在第5章考虑。

例2.6 把一个由加号和减号分隔的数字序列组成的表达式翻译成后缀表示的语法制导定义如图2-5所示。图中与每个非终结符相关联的是一个具有字符串值的属性 t，属性 t 表示该非终结符产生的表达式的后缀表示。

产生式	语义规则
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.t \parallel term.t \parallel '+'$
$expr \rightarrow expr_1 - term$	$expr.t := expr_1.t \parallel term.t \parallel '-'$
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow 0$	$term.t := '0'$
$term \rightarrow 1$	$term.t := '1'$
...	...
$term \rightarrow 9$	$term.t := '9'$

图2-5 中缀到前缀翻译的语法制导定义

一个数字的后缀形式是该数字本身。例如，与产生式 $term \rightarrow 9$ 相关联的语义规则定义：当该产生式在分析树的节点上被使用时，$term.t$ 的值是9。当产生式 $expr \rightarrow term$ 被应用时，$term.t$ 的值成为 $expr.t$ 的值。

产生式 $expr \rightarrow expr_1 + term$ 导出一个包含加号的表达式。加操作符的左操作数由 $expr_1$ 给出，右操作数由 $term$ 给出。产生式中 $expr_1$ 的下标是为了区别产生式左右两侧的 $expr$。与这个产生式关联的语义规则

$$expr.t := expr_1.t \parallel term.t \parallel '+'$$

定义了属性 $expr.t$ 值的计算方式，即首先连接左右操作数的后缀形式 $expr_1.t$ 和 $term.t$，然后连接上加号。语义规则中的操作符 $\parallel$ 表示字符串的连接。 34

图2-6给出了对应于图2-2中分析树的注释分析树。每个节点上的t属性的值用与该节点的产生式相关联的语义规则计算。根节点的属性值是该分析树生成的串的后缀表示。 □

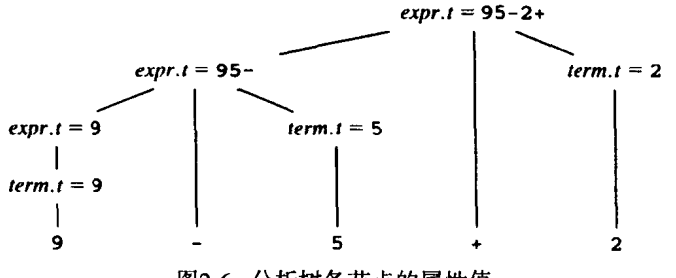


图2-6 分析树各节点的属性值

例2.7 假定一个机器人可以被指示从当前位置向东、向北、向西或者向南移动一步。这样的指令序列可以由下面的文法产生：

$seq \rightarrow seq\ instr \mid$ **begin**
$instr \rightarrow$ **east** | **north** | **west** | **south**

根据输入

begin west south east east east north north

机器人产生的位置改变如图2-7所示。

在图2-7中，每个位置用（x, y）对标志，其中 x 和 y 分别表示从起点开始，向东和向北的步数。如果 x 是负数，则机器人在起始位置向西移动。如果 y 是负数，则机器人在起始位置向南移动。

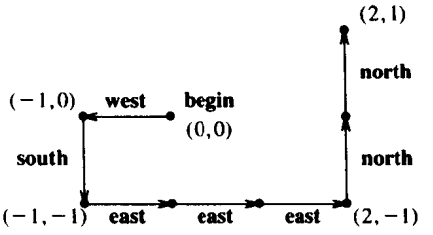


图2-7 机器人的位置的跟踪

35

我们来构建一个语法制导定义，把指令序列翻译成机器人的位置。我们要使用两个属性 *seq.x* 和 *seq.y* 来记录非终结符 *seq* 生成的指令序列所产生的位置。最初，*seq* 生成 **begin**，*seq.x* 和 *seq.y* 均被初始化为0，如图2-8中 **begin west south** 的分析树的最左内节点所示。

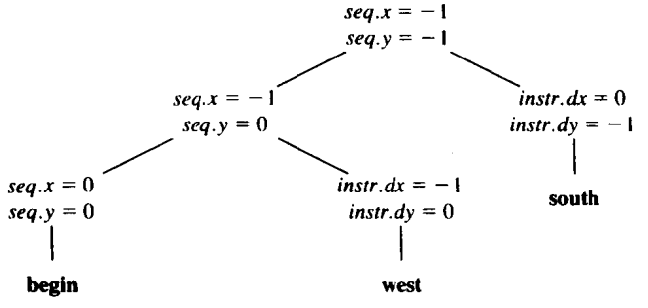


图2-8 **begin west south** 的注释分析树

用 *instr.dx* 和 *instr.dy* 来给定 *instr* 生成的单个指令所产生的位置变化。例如，如果 *instr* 生成 **west**，则 *instr.dx* = −1，*instr.dy* = 0。假定 *seq* 是由一个序列 seq_1 后面跟随一个新指令所形成的，则机器人的新位置由下面的规则给出：

$$seq.x := seq_1.x + instr.dx$$
$$seq.y := seq_1.y + instr.dy$$

用于将指令序列翻译成机器人位置的语法制导定义如图2-9所示。

产生式	语义规则
seq → **begin**	*seq.x* := 0 *seq.y* := 0
seq → seq_1 *instr*	*seq.x* := seq_1.*x* + *instr.dx* *seq.y* := seq_1.*y* + *instr.dy*
instr → **east**	*instr.dx* := 1 *instr.dy* := 0
instr → **north**	*instr.dx* := 0 *instr.dy* := 1
instr → **west**	*instr.dx* := −1 *instr.dy* := 0
instr → **south**	*instr.dx* := 0 *instr.dy* := −1

图2-9 机器人位置的语法制导定义

2.3.4 深度优先遍历

语法制导定义没有规定分析树中属性的计算顺序，只是在计算属性 a 时，a 所依赖的所有其他属性必须已经进行了计算。满足这个要求的任何计算顺序都是可以接受的。通常，在遍历一个分析树的时候，有的属性节点一经访问就必须进行计算，有的属性需要其所有的子节点都被访问以后才能计算，有的属性计算发生在访问其所有子节点的过程中。属性值的计算顺序将在第5章详细讨论。

本章所有的翻译都是通过按照一种预定的顺序对分析树属性的语义规则进行计算来实现的。树的*遍历*是指从根开始，以某种顺序访问树的每一个节点。本章使用图2-10定义的深度优先遍历计算语义规则。它从根开始，从左到右递归访问每个节点的子节点，如图2-11所示。一旦给定节点的所有后代都被访问，则该节点的语义规则将被计算。之所以称之为“深度优

```
procedure visit(n: nodc);
begin
    for n 的每个子节点 m，从左到右 do
        visit(m);
    计算节点 n 处的语义规则
end
```

图2-10 树的深度优先遍历

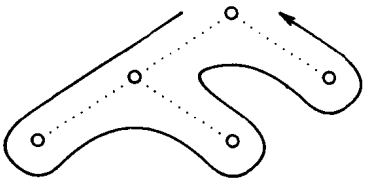


图2-11 树的深度优先遍历示例

先”遍历，是因为它尽可能地访问一个节点的未访问的子节点，于是它尽可能快地访问离根最远的节点。

2.3.5 翻译模式

本章的其余部分用一种过程说明来定义翻译。一个翻译模式是一个上下文无关文法，其中被称为语义动作的程序段被嵌入到产生式右部。一个翻译模式类似于语法翻译制导定义，只是语义规则的计算顺序是显式给出的。一个语义动作的执行位置通过用括号把语义动作括起来并将其放在产生式右部来表示，如

$$rest \rightarrow + \; term \; \{ print('+') \} \; rest_1$$

翻译模式对于由基本语法产生的每个语句 x 都产生一个输出，方法是：按照 x 的分析树的深度优先遍历顺序执行语义动作。我们来考虑具有一个用 *rest* 标记的节点的分析树（这里 *rest* 表示 *rest* 的产生式）。语义动作{ *print* ('+')}在 *term* 子树之后、$rest_1$ 子树之前执行。

当我们给一个翻译模式画一棵分析树的时候，我们通过为语义动作构造一个特殊的子节点来指出语义动作，并使用虚线连接到其产生式的节点。例如，表示 *rest* 产生式的分析树和语义动作如图2-12所示。语义动作节点没有子节点，使得该节点在第一次被访问到的时候便被执行。

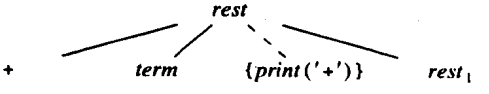


图2-12 为语义动作创建的特殊的叶节点

2.3.6 翻译的输出

翻译模式的语义动作把翻译的输出以一次一个字符或一个字符串的形式写入一个文件。例如，我们通过每次写9-5+2中一个字符的方式把表达式9-5+2翻译成95-2+，而不需要额外的空间存储子表达式的翻译。当输出按照这种方式递增地被创建时，写字符的顺序变得很重要。

语义制导定义具有如下特性：每个产生式左部的非终结符的翻译是将该产生式右部的非终结符的翻译按照它们在右部出现的次序连接起来得到的，在连接过程中可能还需要附加（也可能不需要）一些额外的串。具有这样特性的语义制导定义称之为简单的语义制导定义。例如，考虑图2-5中的第一个产生式和语法制导定义的语义规则：

36 ~ 38

产生式	语义规则	
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.t \parallel term.t \parallel '+'$	(2-6)

其中，*expr.t* 的翻译是 $expr_1$ 的翻译、*term* 的翻译、+的连接。请注意，在产生式的右部，$expr_1$ 出现在 *term* 的前面。

在下面的例子中，*term.t* 和 $rest_1.t$ 之间出现了额外的串：

产生式	语义规则	
$rest \rightarrow + \; term \; rest_1$	$rest.t := term.t \parallel '+' \parallel rest_1.t$	(2-7)

在产生式的右部，非终结符 *term* 出现在 $rest_1$ 的前面。

简单语法制导定义可以用翻译模式来实现。在翻译模式中，语义动作按照定义中出现的顺序输出额外串。下面产生式中的语义动作分别输出了(2-6)和(2-7)中的额外串：

$$expr \rightarrow expr_1 + term \ \{ print('+') \}$$
$$rest \rightarrow + term \ \{ print('+') \} \ rest_1$$

例2.8 图2-5是把表达式翻译成后缀形式的一个简单的语法制导定义。由该定义得到的翻译模式如图2-13所示，带语义动作的9-5+2的分析树如图2-14所示。注意，尽管图2-6和图2-14描述的是相同的输入-输出映射，但两者构造翻译的过程是不相同的。图2-6是把输出附加在分析树的根，而图2-14把输出递增地显示出来。

expr	→ *expr* + *term*	{ *print*('+') }
expr	→ *expr* - *term*	{ *print*('-') }
expr	→ *term*	
term	→ 0	{ *print*('0') }
term	→ 1	{ *print*('1') }
	...	
term	→ 9	{ *print*('9') }

图2-13 把表达式翻译成后缀形式的语义动作

图2-14的根表示图2-13中的第一个产生式。在深度优先遍历中，当我们遍历根节点的最左子树时，先执行左操作数 *expr* 的子树中
39 的所有语义动作。然后，我们访问没有语义动作的叶节点+。接下来，我们执行右操作数 *term* 的子树中的所有语义动作。最后，执行特殊节点上的语义动作 { *print*('+')}。

由于 *term* 产生式右部只有一个数字，语义动作只显示该数字。产生式 *expr* →*term* 不产生输出，头两个产生式的语义动作只须显示操作符。在深度优先遍历分析树的过程中执行图2-14中的语义动作显示95-2+。

作为一般规则，多数分析方法都以一种“贪心”的方式从左到右地处理输入，即在读入下一个记号之前构造尽可能多的分析树的组成部分。在简单翻译模式（由简单语义制导定义得到的）中，语义动作也是按照从左到右的顺序执行的。因此，实现简单翻译模式时，我们可以在语法分析的时候执行语义动作，完全没有必要构造分析树。

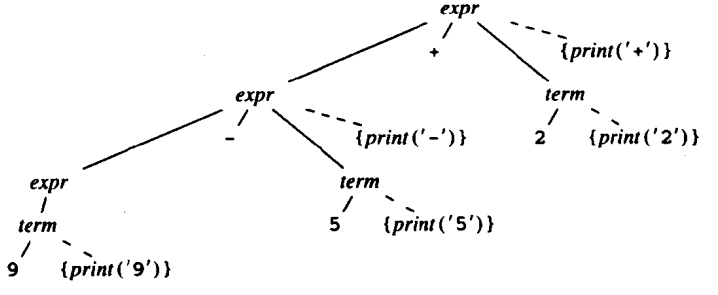


图2-14 把9-5+2翻译成95-2+的语义动作

2.4 语法分析

语法分析是决定一个记号串是否能够由一个文法产生的过程。在讨论这个问题时，尽管编译器可能没有真正构造这样一棵分析树，我们认为构造分析树是有益的。语法分析器应该具有构造分析树的能力，否则，不能保证翻译的正确性。

本节介绍一种语法分析方法，这种方法可以用来构造语法制导翻译器。下节给出一个实现图2-13的翻译模式的完整C程序。使用软件工具直接从翻译模式生成翻译器是构造翻译器的理想方法。4.9节将详细介绍这样一种软件工具。这个软件工具无需修改就可以实现图2-13的翻译模式。

我们可以为任何文法构造语法分析器。实际中使用的文法一般都具有特定的形式。对于任

意上下文无关文法，我们可以构造一个时间复杂性为 $O(n^3)$ 的语法分析器，即在 $O(n^3)$ 时间内完成对具有 n 个记号的串的语法分析。但是，立方阶的时间代价太昂贵了。给定一种程序设计 40
语言，通常可以构造一个可快速分析的文法。线性时间复杂性算法足以分析实际中出现的所有程序设计语言。程序设计语言语法分析器总是从左到右扫描输入，每次超前扫描一个记号。

语法分析方法可以分为两类：自顶向下方法和自底向上方法。这些术语是指构造分析树节点的顺序。前者按照从根节点到叶节点的顺序构造分析树，后者按照从叶节点到根节点的顺序构造分析树。自顶向下分析器是常用的语法分析器，其原因在于这种语法分析器可以很容易地通过自顶向下的方法手工构造出来。然而，自底向上分析方法可以处理大量文法和翻译模式，所以直接从文法产生语法分析器的软件工具通常使用自底向上的方法。

2.4.1 自顶向下语法分析

我们通过一种适于自顶向下语法分析的文法来介绍自顶向下的分析方法。我们将在本节的后面考虑构造自顶向下语法分析器的一般方法。下面的文法生成一个Pascal的类型子集。我们用记号 **dotdot** 表示 ".."，**dotdot** 强调该字符序列是一个基本符号单元。

$$
\begin{aligned}
\textit{type} &\rightarrow \textit{simple} \\
&\quad \mid \uparrow \textbf{id} \\
&\quad \mid \textbf{array}\ [\ \textit{simple}\ ]\ \textbf{of}\ \textit{type} \\
\textit{simple} &\rightarrow \textbf{integer} \\
&\quad \mid \textbf{char} \\
&\quad \mid \textbf{num dotdot num}
\end{aligned}
\tag{2-8}
$$

为了自顶向下地构造一个分析树，我们从标有开始非终结符的根节点开始，反复执行下面两步（见图2-15中的例子）：

1. 在标有非终结符 A 的节点 n，选择 A 的一个产生式，用该产生式右部的符号构造节点 n 的子节点。

2. 寻找下一个要构造子树的节点。

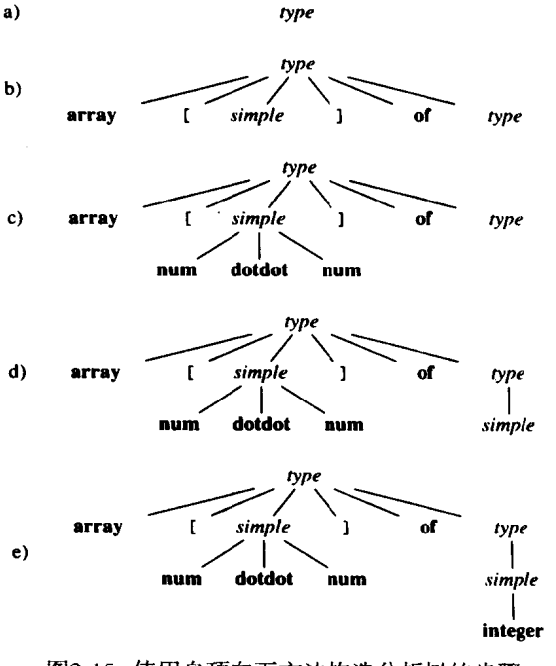


图2-15 使用自顶向下方法构造分析树的步骤

对于某些文法，上面的步骤可以通过一次从左到右扫描输入串来实现。输入中当前被扫描的记号通常是被称为*超前扫描符号*（lookahead symbol）。以下我们也称超前扫描符号为当前符号。最初，超前扫描符号是输入串的第一个记号，即最左端的记号。图2-16说明了对如下输入串的语法分析过程：

array [num dotdot num] of integer

最初，记号 **array** 是超前扫描符号，分析树的已知部分只有标记为开始非终结符 *type* 的根节点，如图2-16a所示。我们的目标是：以构造出来的分析树所产生的字符串与输入字符串匹配的方法构造分析树的其余部分。

为了与输入串匹配，图2-16a中的非终结符 *type* 必须导出一个以超前扫描符号 **array** 开始的串。在文法(2-8)中，只有一个以 *type* 为左部的产生式可以导出这样的串，所以我们使用这个产生式来构造根节点 *type* 的子节点，并在子节点上标上产生式右部的符号。

在图2-16所示的三个步骤中，都有两个箭头。一个指向输入串中超前扫描符号，另一个箭头指向分析树当前被考虑的节点。当一个节点的子节点都已经构造完毕，我们从该节点的最左端子节点开始继续构造分析树的其余部分。在图2-16b中，根节点的子节点都已经构造完毕，下一个要考虑的节点是标记为 **array** 的最左端的子节点。

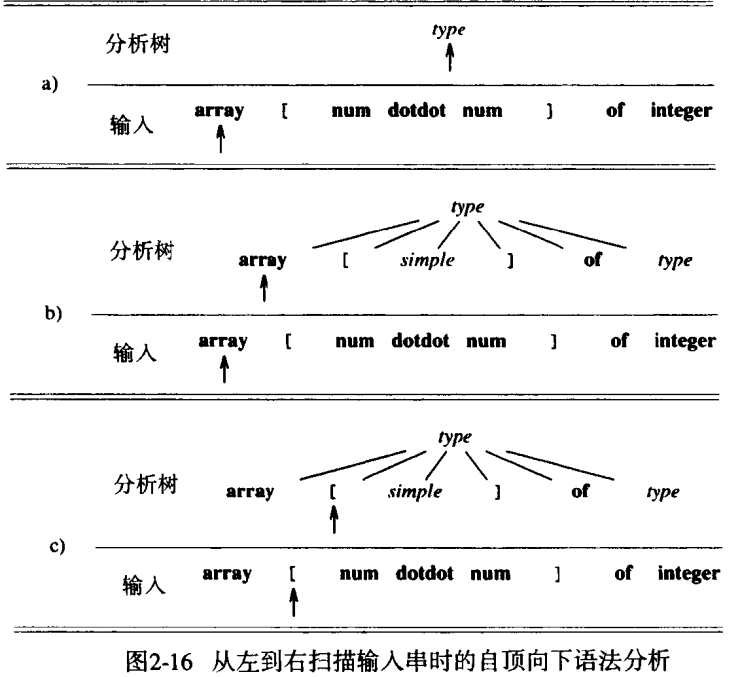


图2-16 从左到右扫描输入串时的自顶向下语法分析

如果当前被考查的分析树的节点是一个终结符，而且该终结符与超前符号匹配，则分析树的箭头和输入的箭头都前进一步。输入的下一个记号成为新的超前扫描符号，分析树的下一个子节点将被考查。在图2-16c中，分析树中的箭头指向了根的下一个子节点，输入的箭头指向了下一个记号“[”。接下来，分析树的箭头指向标记非终结符 *simple* 的子节点。在考虑一个标有非终结符的节点时，我们将重复为这个非终结符选择一个适当的产生式的过程。

通常，为非终结符选择产生式可能会涉及“试验和错误”的问题，即在选择产生式的时候，如果一个产生式不合适，我们将不得不回溯，测试另外一个产生式。一个产生式“不适合”是

指使用了该产生式之后无法产生与输入串匹配的分析树。下面我们介绍预测分析方法，在这种分析方法中不会发生回溯问题。

41
~
43

2.4.2 预测分析法

*递归下降分析法*是一种自顶向下的语法分析方法，在这种方法中，我们执行一组递归过程来处理输入串。每一个过程都惟一地与文法的一个非终结符相关联。这里，我们考虑一种特殊的递归下降分析法，称为*预测分析法*。在预测分析法中，超前扫描符号无二义地确定了为每个非终结符选择的过程。处理输入时调用的过程序列隐式地定义了输入串的分析树。

图2-17的预测语法分析器由非终结符 *type* 的过程、非终结符 *simple* 的过程和一个称为 *match* 的过程组成。我们用 *match* 过程来简化 *type* 过程和 *simple* 过程的代码。如果变量 *t* 和超前扫描符号匹配，输入符号串的箭头将前进一步，指向下一个输入记号。因此，*match* 过程改变了当前被扫描的输入记号 *lookahead* 变量的值。

在我们的文法中，分析过程从调用开始非终结符 *type* 的过程开始。输入串与图2-16中的串相同，*lookahead* 初始化为第一个记号 **array**。过程 *type* 对应着产生式 *type*→**array**[*simple*] **of** *type*的右部执行如下代码：

match(**array**); *match*('['); *simple*; *match*(']'); *match*(**of**); *type* (2-9)

注意，产生式 *type* → **array**[*simple*] **of** *type* 右部的每个终结符都和超前扫描符号匹配，而每个非终结符都产生一个过程调用。

```
procedure match(t: token);
begin
    if lookahead = t then
        lookahead := nexttoken
    else error
end;

procedure type;
begin
    if lookahead is in { integer, char, num } then
        simple
    else if lookahead = '↑' then begin
        match('↑'); match(id)
    end
    else if lookahead = array then begin
        match(array); match('['); simple; match(']'); match(of); type
    end
    else error
end;

procedure simple;
begin
    if lookahead = integer then
        match(integer)
    else if lookahead = char then
        match(char)
    else if lookahead = num then begin
        match(num); match(dotdot); match(num)
    end
    else error
end;
```

图2-17 预测语法分析器的伪代码

以图2-16中的串为输入，在 **array** 和 [匹配以后，超前扫描符号是 **num**。这时，过程 *simple* 被调用，并在其过程体中执行如下代码：

match(**num**); *match*(**dotdot**); *match*(**num**)

超前扫描符号指导产生式的选择。如果产生式的右部由一个记号开始，则当该记号与超前扫描符号匹配的时候这个产生式被选用。现在考虑一个由非终结符开始的产生式的右部：

type → *simple* (2-10)

如果超前扫描符号可以由 *simple* 产生，则该产生式被选用。例如，在执行(2-9)代码段时，假设当控制到达过程 *type* 的调用时超前扫描符号是 **integer**。没有第一个记号是 **integer** 的 *type* 产生式。然而，第一个记号为 **integer** 的 *simple* 产生式是存在的。所以，产生式(2-10)通过调用过程 *simple* 来使用。

预测分析依赖于产生式右部产生的第一个符号是什么。更精确地说，令 α 是非终结符 *A* 的某产生式的右部。定义 FIRST(α) 是作为由 α 产生的一个或多个串的第一个符号出现的集合。如果 α 是 ϵ 或者可以产生 ϵ，则 ϵ 也属于 FIRST(α) [⊖]。例如：

FIRST(*simple*) = { **integer**, **char**, **num** }
FIRST(↑ **id**) = { ↑ }
FIRST(**array** [*simple*] **of** *type*) = { **array** }

44
~
45

实际上，许多产生式的右部都由记号开始，从而简化了FIRST集合的构造。计算FIRST的算法在4.4节中给出。

如果有两个产生式 $A \rightarrow \alpha$ 和 $A \rightarrow \beta$ 可供选用，则必须考虑相应的 FIRST 集合。无回溯的递归下降分析方法要求 FIRST(α) 和 FIRST(β) 不相交。这样超前扫描符号就可以选择正确的过程去执行。如果超前扫描符号在 FIRST(α) 集合中，则使用 α，否则，如果超前扫描符号在 FIRST(β) 中，则使用 β。

2.4.3 何时使用 ϵ 产生式

右部是 ϵ 的产生式称为 ϵ 产生式，需要特殊处理。当没有其他产生式可用的时候，递归下降语法分析器把 ϵ 产生式作为默认产生式使用。例如，考虑下面的产生式：

stmt → **begin** *opt_stmts* **end**
opt_stmts → *stmt_list* | ϵ

当分析到 *opt_stmts* 时，如果超前扫描符号没有在 FIRST(*stmt_list*) 集合中，则使用 ϵ 产生式。如果超前扫描符号是**end**，这种选择是正确的，除了**end** 之外的任何超前扫描符号都将导致一个错误，可以在 *stmt* 的语法分析中检测到。

2.4.4 设计一个预测语法分析器

预测语法分析器是一个由多个过程组成的程序，每个过程对应一个非终结符。每个过程完成如下两项任务：

1. 检查超前扫描符号，决定使用哪个产生式。如果超前扫描符号在 FIRST(α) 中，则选择使用右部为 α 的产生式。对于任何超前扫描符号，如果产生式右部存在冲突，那么我们不能

⊖ 右部带有 ϵ 的产生式将使确定一个非终结符所产生的第一个符号的工作复杂化。例如，如果非终结符 *B* 能导出空串，而且存在产生式 $A \rightarrow BC$，那么由 *C* 所产生的第一个符号还可以是 *A* 所产生的第一个符号。如果*C*也可以产生 ϵ，则 FIRST(*A*) 和 FIRST(*BC*) 均包含 ϵ。

在这种文法上使用这种分析方式。如果超前扫描符号不在任何其他右部的 FIRST 集合中，右部具有 ϵ 的产生式将被使用。

2. 过程通过模仿其右部来使用一个产生式。一个非终结符导致该非终结符对应的过程被调用。一个与超前扫描符号匹配的记号导致下一个输入记号被读入。如果在某个点上，产生式的记号与超前扫描符号不匹配，则报告出错。图2-17是对文法(2-8)应用这些规则的结果。

类似于通过扩展文法来形成一个翻译模式，我们也可以通过扩展预测语法分析器来形成一个语法制导翻译器。实现此目的的算法在5.5节中给出。因为本章实现的翻译模式不涉及非终结符的属性，下面仅给出满足当前要求的构造方法：

1. 构造一个预测语法分析器，忽略产生式中的语义动作。 46

2. 把翻译模式中的语义动作拷贝到语法分析器。如果一个语义动作出现在产生式p的文法符号X的后面，则该语义动作被拷贝到实现X的代码后面。否则，如果语义动作出现在一个产生式的开始，则该语义动作被拷贝在该产生式的实现代码的最前面。

我们将在下一节构造这样一个翻译器。

2.4.5 左递归

递归下降语法分析器很可能造成无限循环。当出现下面这样一个左递归产生式时，无限循环就会出现：

$$expr \rightarrow expr + term$$

在这里，产生式右部的最左符号和产生式左部的非终结符是相同的。假定 *expr* 对应的过程要使用这个产生式。因为右部是由 *expr* 开始的，所以 *expr* 过程被递归调用，出现了无限循环。注意，只有右部终结符与超前扫描符号匹配时，超前扫描符号才会发生改变。因为产生式是以非终结符 *expr* 开始的，输入符号在递归调用期间没有机会改变，所以导致无限循环。

通过重写与递归相关的产生式，我们可以消除左递归产生式。考虑下面非终结符 A 的两个产生式：

$$A \rightarrow A\alpha \mid \beta$$

这里 α 和 β 是不以 A 开始的终结符和非终结符序列。例如，在产生式

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

中，$A = expr, \alpha = + term, \beta = term$。

因为产生式 $A \rightarrow A\alpha$ 的右部最左面的符号是 A 自身，因此 A 是*左递归产生式*。重复应用这个产生式，在 A 的右部产生一个 α 的序列，如图2-18a。当 A 最终由 β 替换时，β 后面跟着0个或者多个α的序列。

同样的结果可通过以如下方式改写产生式得到（如图2-18b所示）：

$$\begin{aligned} A &\rightarrow \beta R \\ R &\rightarrow \alpha R \mid \epsilon \end{aligned} \tag{2-11}$$

这里，R是一个新的非终结符。产生式 $R \rightarrow \alpha R$ 以 R 自身作为产生式右部最后一个符号，因而是*右递归*的。右递归产生式导致向右下侧延伸的分析树，如图2-18b所示。向右下侧延伸的分析树使得包含左结合操作符表达式的翻译变得十分困难。不过，在下一节我们可以看到，通过基于右递归文法的翻译模式的仔细设计，可以将表达式正确翻译成后缀形式。

在第4章，我们将考虑更一般的左递归形式，而且讨论如何从一个文法中消除所有左递归。

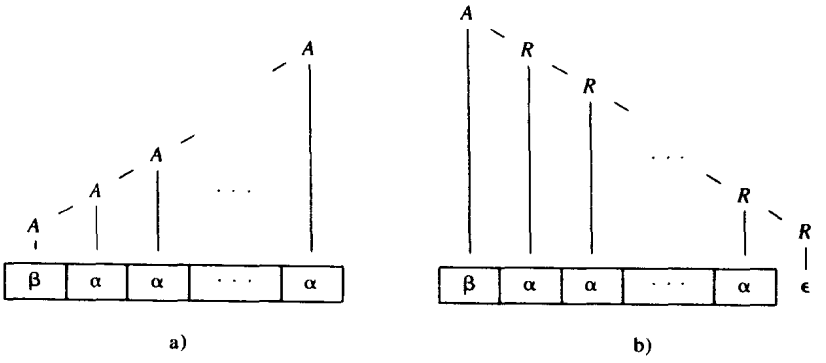


图2-18 产生一个串的左递归和右递归方式

2.5 简单表达式的翻译器

使用前面三节介绍的技术，我们可以用C语言编写一个语法制导翻译器，这个翻译器可以把算术表达式翻译成后缀形式。为了使最初的程序比较小而且易于理解，我们从最简单的表达式开始，即由加号和减号分隔的由数字构成的表达式。在随后的两节中，我们将使其扩展为包括数字、标识符和其他操作符。由于表达式出现在许多程序设计语言中，详细地研究一下它们的翻译问题是很有价值的。

一个语法制导翻译模式可以作为一个翻译器的规范。我们用图2-19中的模式作为要实现的翻译的定义。通常，一个给定翻译模式的文法在能够被预测语法分析器分析之前需要加以修改。图2-19给出的模式的文法是左递归的，如上节所述。预测语法分析器不能处理左递归文法。我们可以通过消除左递归得到一个适用于预测递归下降编译器的文法。

47
~
48

```
expr → expr + term    { print('+') }
expr → expr - term    { print('-') }
expr → term
term → 0              { print('0') }
term → 1              { print('1') }
  · · ·
term → 9              { print('9') }
```

图2-19 中缀到后缀翻译器的初始说明

2.5.1 抽象语法和具体语法

为了便于理解，我们从*抽象语法树*开始讨论一个输入串的翻译。在一个抽象语法树中，每个节点表示一个操作符，该节点的子节点表示操作数。与此相对应，分析树称为*具体语法树*，相应的文法称作*具体语法*。抽象语法树（或简称*语法树*）与分析树是不同的。因为形式上的差别（不影响翻译）没有出现在语法树中，所以抽象语法树与分析树有所不同。

例如，9-5+2的语法树如图2-20所示。+和-具有相同的优先级，相同优先级的操作符从左到右进行计算，因此9-5被组成一个子表达式。将图2-2的分析树与图2-20相比，我们可以看到，在语法树中，操作符都是内节点；在分析树中，操作符皆为叶节点。

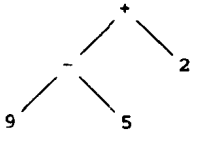


图2-20 9-5+2的语法树

我们希望翻译模式所基于的文法的分析树尽可能与语法树相同。图2-19中文法的子表达式分组与语法树中的分组是相似的。然而，图2-19中的文法是左递归的，不适于预测分析。我们似乎遇到了一个矛盾：一方面，我们需要一个便于分析的文法；另一方面，为了便于翻译，我们又需要一个不同的文法。最明显的解决方法是消除左递归。不过，我们必须小心行事，如下面的例子所示。

例2.9 下面的文法不适合将表达式翻译成后缀形式，尽管它和图2-19产生相同的语言并

且能够用于递归下降分析：

```
expr → term rest
rest → + expr | - expr | ϵ
term → 0 | 1 | ··· | 9
```

这个文法存在如下问题：*rest* → + *expr* 和 *rest* → -*expr* 产生的操作符的操作数是不明显的。用下面两种模式从 *expr.t* 翻译 *rest.t* 都是无法接受的：

```
rest → - expr  { rest.t := '-' || expr.t }        (2-12)
rest → - expr  { rest.t := expr.t || '-' }        (2-13)
```

我们只列出了减号操作符的产生式及其语义动作。9-5的正确翻译是95-。然而，如果我们使用(2-12)的语义动作，则减号出现在 *expr.t* 的前面，9-5的翻译还是9-5，这显然是不对的。

另一方面，如果我们使用(2-13)以及相似的加法规则，操作符将一直移到右部末尾。这样，9-5+2被错误地翻译成952+-，而正确的翻译应该是95-2+。 □

2.5.2 调整翻译模式

图2-18中的左递归消除技术同样可以应用到包含语义动作的产生式。我们将在5.5节扩展这种技术，以处理综合属性。这种技术将产生式$A \rightarrow A\alpha \mid A\beta \mid \gamma$转换成

```
A → γR
R → αR | βR | ϵ
```

当语义动作嵌入在产生式中时，我们将在转换过程中一直携带这些语义动作。如果令A = *expr*，α = +*term*{*print* ('+')}，β = -*term*{*print* ('-')}，γ = *term*，则上面的转换将产生(2-14)中的翻译模式。图2-19中的 *expr* 产生式已经转换成(2-14)中 *expr* 和 *rest* 的产生式，其中 *rest* 是新引入的非终结符。*term* 的产生式是图2-19中 *term* 产生式的重复。注意，如此得到的文法与例2.9中的文法是不同的，正是这种不同才可能得到我们预期的翻译。

```
expr → term rest
rest → + term { print('+') } rest | - term { print('-') } rest | ϵ
term → 0 { print('0') }
term → 1 { print('1') }                           (2-14)
  ···
term → 9 { print('9') }
```

图2-21说明了如何使用上面的文法把9-5+2翻译成为95-2+。

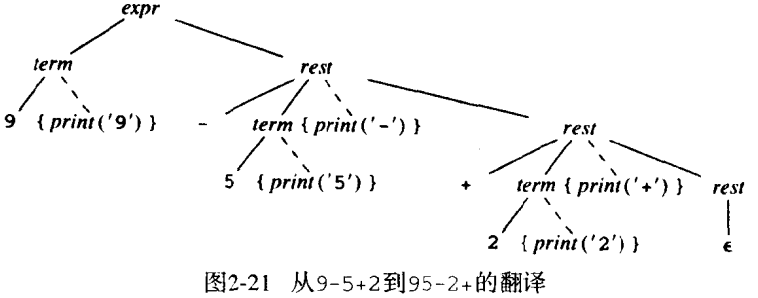


图2-21 从9-5+2到95-2+的翻译

2.5.3 非终结符 *expr*、*term* 和 *rest* 的过程

现在我们用C语言利用语法制导翻译模式(2-14)实现翻译器。翻译器的重要部分是图2-22中

的 expr、term 和 rest 函数的 C 代码。这些函数实现了(2-14)中相应的非终结符。

match 函数将在后面给出，它是图2-17中 match 过程的C代码。match 函数把给定的记号与超前扫描符号进行匹配，然后读取输入串中的下一个符号，并将其作为新的超前扫描符号。由于在我们的语言里每个记号都是单个字符，所以 match 函数是通过比较字符和读字符操作来实现的。

```
expr()
{
    term(); rest();
}

rest()
{
    if (lookahead == '+') {
        match('+'); term(); putchar('+'); rest();
    }
    else if (lookahead == '-') {
        match('-'); term(); putchar('-'); rest();
    }
    else ;
}

term()
{
    if (isdigit(lookahead)) {
        putchar(lookahead); match(lookahead);
    }
    else error();
}
```

图2-22 非终结符*expr*、*rest* 和 *term* 的函数

有些人不熟悉C语言，因而在这里我们介绍一下C语言和其他 Algol 派生语言（如 Pascal 语言）之间的显著区别。C语言编写的程序由一系列函数定义组成，程序从一个特殊的函数 main 开始执行。函数定义不可以嵌套。函数的输入参数由函数名后面用括号括起来的参数表来表示。即使函数没有参数，括起参数表的括号也是必须的，因此我们写 expr()、term() 和rest()。函数间的通信通过传递参数或者访问全局变量来实现。例如，函数term()和rest() 用全局标识符 lookahead 来传递超前扫描符号。

C 和 Pascal 语言用右面的符号进行赋值和相等测试。

操作	C语言	Pascal语言
赋值	=	:=
相等测试	==	=
不等测试	!=	<>

非终结符函数模拟产生式的右部。例如，产生式 *expr* → *term rest*是通过在 expr() 函数中调用 term() 和 rest()来实现的。

作为另外一个例子，如果超前扫描符号是加号，函数rest()使用(2-14)中*rest*的第一个产生式；如果超前扫描符号是减号，则使用 *rest* 的第二个产生式；默认情况使用产生式 *rest* → ϵ。*rest* 的第一个产生式是用图2-22中的 if 语句实现的。如果超前扫描符号是+，调用match('+')来匹配加号。在调用 term()以后，调用C语言标准库例程 putchar('+')输出加符号来实现语义动作。因为 *rest* 的第三个产生式的右部是ϵ，所以在 rest()中最后一个 else 的后面是一个空语句。

term 的10个产生式用来生成10个数字。在图2-22中，例程 isdigit 测试超前扫描符号是不是一个数字。如果测试成功，这个数字首先被输出，然后调用 match()。否则，出错。(注意，函数 match()改变超前扫描符号，所以输出一定要在调用 match()之前进行。) 在给出完整的程序之前，我们先来优化图2-22中的代码。

2.5.4 翻译器的优化

某些递归调用可以用循环替换。如果一个过程中执行的最后一条语句是对该过程的递归调用，则该调用称为是*尾递归*的。例如，函数rest()的第4行和第7行的rest()调用都是尾递归的，因为执行完这些调用之后控制到达函数体的末尾。 51~52

我们可以通过用循环代替尾递归来加速程序。对于没有参数的过程，我们可以用一个转移到过程开始位置的跳转语句来替换尾递归。rest过程的代码可以重写为：

```
rest()
{
L:   if (lookahead == '+') {
          match('+'); term(); putchar('+'); goto L;
     }
     else if (lookahead == '-') {
          match('-'); term(); putchar('-'); goto L;
     }
     else ;
}
```

只要超前扫描符号是一个加号或者减号，rest 过程匹配该符号，并调用 term 去匹配一个数字，然后重复这一过程。注意，因为 match 过程每次调用的时候都删除加减号，所以这个循环只有在加减号和数字交替出现的时候才发生。图2-22中的代码经过这些改变之后，rest 函数仅由 expr 函数调用 (见第3行)。因此，这两个函数可以合二为一，如图2-23所示。在C语言中，语句 *stmt* 的重复执行可以通过下面的语句来实现：

```
while(1) stmt
```

因为条件1总为真。我们可以通过执行一条 break 语句退出循环。图2-23中的代码风格允许我们方便地添加其他操作符。

```
expr()
{
     term();
     while(1)
          if (lookahead == '+') {
               match('+'); term(); putchar('+');
          }
          else if (lookahead == '-') {
               match('-'); term(); putchar('-');
          }
          else break;
}
```

图2-23 图2-22中expr和rest函数的替换 53

2.5.5 完整程序

上述翻译器的完整C语言程序如图2-24所示。第1行用#include语句开始，用来加载

<ctype.h>文件。<ctype.h>是一个标准例程文件，其中包括判定函数isdigit的代码。

由单个字符构成的记号由标准库例程 getchar 提供，getchar 读入输入文件中的下一个字符。不过，在图2-24中的第2行把 lookahead 声明为整型，使之能够存储后续几节中将会使用的多字符记号。由于 lookahead 在所有的函数之外说明，它是一个可以由图2-24第2行之后定义的所有函数存取的全局变量。

```
#include <ctype.h> /* 加载带有判定函数isdigit的文件 */
int lookahead;

main()
{
     lookahead = getchar();
     expr();
     putchar('\n'); /* 增加尾部换行符 */
}

expr()
{
     term();
     while(1)
          if (lookahead == '+') {
               match('+'); term(); putchar('+');
          }
          else if (lookahead == '-') {
               match('-'); term(); putchar('-');
          }
          else break;
}

term()
{
     if (isdigit(lookahead)) {
          putchar(lookahead);
          match(lookahead);
     }
     else error();
}

match(t)
     int t;
{
     if (lookahead == t)
          lookahead = getchar();
     else error();
}

error()
{
     printf("syntax error\n"); /* 打印错误信息 */
     exit(1);      /* 停止 */
}
```

图2-24 把中缀表达式翻译成后缀表达式的C程序

函数 match 检查记号是否匹配。如果超前扫描符号匹配，match 读入下一个输入，否则调用出错例程，报告错误信息。

函数 error 使用标准库函数 printf 语句输出信息“syntax error”，然后调用 exit(1)终止程序的执行。

2.6 词法分析

现在我们为上节的翻译器增加一个词法分析器。词法分析器读入输入串，将其转换成将被语法分析器分析的记号流。回想一下2.2节的文法定义，一个语言的语句是由记号串构成的。构成一个记号的一个输入字符序列称为词素。词法分析器把语法分析器和记号的词素表示分隔开来。作为本节的开始，我们先来讨论我们希望词法分析器完成的一些功能。

2.6.1 剔除空白符和注释

上一节介绍的表达式翻译器读取并处理输入中的每一个字符，所以像空格这样的附加字符将导致失败。许多语言允许“空白符”（空格、制表符或者换行符）出现在记号之间。源程序中的注释一般都被语法分析器和翻译器忽略，所以它们也可以看成空白符。

如果词法分析器消除了空白符，语法分析器就不必再考虑空白符。修改文法使得语法中包含空白符的做法实现起来很难。

2.6.2 常数

在一个表达式中，任何一个允许单个数字出现的位置都应该允许任何整型常数出现。因为整型常数是一个数字序列，我们可以通过在文法中添加产生式或者创建常数的记号使整型常数成为合法的。由于翻译期间把数作为一个单元来处理，收集数字形成整数这一任务一般由词法分析器来完成。

令 **num** 是表示整数的记号。当一个数字序列出现在输入流中时，词法分析器将把 **num** 传递给语法分析器。整数的值作为记号 **num** 的属性值传递给语法分析器。从逻辑上说，词法分析器传递记号及其属性值给语法分析器。如果我们把记号和它的属性值用< >括起来作为一个元组，那么输入31+28+59就可以写成 **<num, 31>** < +, > **<num, 28>** <+, > **<num, 59>**。‘+’ 没有相应的属性，所以元组的第2个分量（属性）在语法分析中没有任何作用，但是在翻译时还是需要的。

2.6.3 识别标识符和关键字

程序设计语言使用标识符作为变量名、数组名、函数名和一些其他的语言对象名。程序设计语言的文法常把标识符作为记号处理。基于这类文法的语法分析器在输入中每遇到一个标识符都赋予它们相同的记号 **id**。例如，词法分析器将输入

```
count = count + increment;                    (2-15)
```

转换成记号流

```
id = id + id ;                                (2-16)
```

这个记号流将用于语法分析。

在谈及输入行(2-15)的词法分析时，区分记号 **id** 和与这个记号的实例相关的词素count和 increment 之间的不同是很有用的。翻译器需要知道 count 词素形成了(2-16)的前两个 **id** 的实例，increment 词素形成了第3个 **id** 的实例。

当输入流中出现形成标识符的词素时，我们需要某种机制来决定该词素以前是否出现过。如第1章中所述，符号表就是这样一种机制。词素存储在符号表的一个表项中，而指向该表项的指针则成为记号 **id** 的一个属性。

许多程序设计语言使用固定的字符串（如begin、end、if等等）作为标点符号标志或者某种结构的标识。这些字符串称作关键字，通常也满足形成标识符的规则。于是，我们还需要一种机制来决定一个词素何时形成关键字何时形成标识符。如果将关键字保留，也就是说，如果它们不能作为标识符，这个问题就很容易解决。于是，只有字符串不是关键字时它才形成标识符。

54
~
56

如果相同的字符出现在多个记号的词素中，我们又会遇到记号分割的问题。例如，Pascal中的<、<=和<>中都包含<。有效地识别这种记号的技术将在第3章中讨论。

2.6.4 词法分析器的接口

词法分析器介于语法分析器和输入流之间，并与这两者交互（如图2-25所示）。词法分析器从输入串读字符并形成词素，然后将词素生成的记号及其属性值传递给编译器的下一个阶段。在某些情况下，词法分析器在把记号传给语法分析器之前，需要从输入串超前地读入一些字符，以确定需要传递给语法分析器的正确记号。例如，Pascal的词法分析器在读到>字符时需要读入下一个字符。如果下一个字符是=，则词法分析器把>=组合在一起作为形成“大于等于”操作符记号的词素；否则把 > 作为形成“大于”操作符记号的词素，并且词法分析器已经多读了一个字符。多读入的字符必须退回给输入流，因为它可能是下一个词素的开始符号。

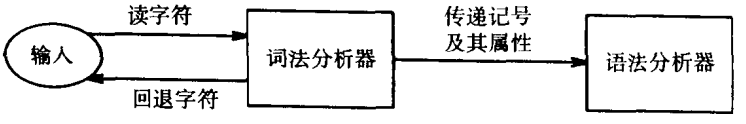


图2-25 在输入和语法分析器之间插入词法分析器

词法分析器和语法分析器形成“生产者-消费者”对。词法分析器产生记号，语法分析器消费记号。产生的记号在被消费之前保存在记号缓冲区中。两者的交互仅受缓冲区大小的限制，原因是：当缓冲区满时，词法分析器不能继续产生记号；当缓冲区空时，语法分析器不能继续分析。通常，缓冲区只能存储一个记号。在这种情况下，二者之间的交互可以通过下面的方式简单地实现：使词法分析器成为被语法分析器调用并为语法分析器返回所需的记号的过程。

读入字符和退回字符操作一般都通过建立一个输入缓冲区来实现。编译器每次把一组字符读入缓冲区，用一个指针指向当前已经被分析的输入部分。如果需要退回字符，只需将指针向回移动。为了能够给出详细的错误信息报告（例如，必须给出错误出现在输入串的位置），我们需要保存输入字符。输入字符的缓冲可以提高编译器的效率，每次读一组字符比每次读一个

57

字符的效率高。输入缓冲技术在3.2节讨论。

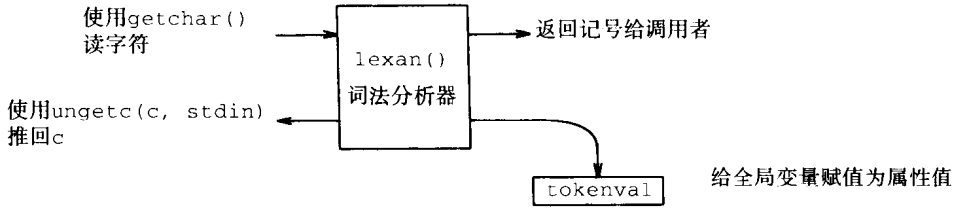
2.6.5 词法分析器

现在我们为2.5节的表达式翻译器构造一个简单的词法分析器。词法分析器的目的是使表达式中只允许出现空白符和由多个数字组成的数。我们在下一节将扩展这个词法分析器，以允许表达式中出现标识符。

图2-26说明了词法分析器（用C语言书写的函数lexan）如何实现图2-25中的交互。getchar 和 ungetc 是包含文件 <stdio.h> 中的标准例程，实现了输入串的缓冲处理。lexan 分别调用 getchar 和 ungetc 来实现读入字符和推回字符。设c是字符变量，语句 c=getchar() 和 ungetc(c, stdin) 保持输入串的次序不乱。函数 getchar 的调用把下一个输入字符赋值给c，ungetc 推回c中的字符到标准输入 stdin。

如果实现编译器的程序设计语言不允许从函数返回数据结构，则记号和它的属性必须分别

传递。函数lexan返回记号的整数编码。一个字符的记号是那个字符的传统整数编码。对**num**这样的记号，其编码可以用大于任何单个字符的整数编码的整数进行编码，即256。为了使编码修改方便，我们使用符号常量NUM作为 **num** 的整数编码。在 Pascal 语言中，NUM 与其编码的关联可以通过 **const** 声明来实现。在C语言中，使用如下的 define 语句令其表示256：#define NUM 256。如果在输入串中遇到一个数字序列，lexan 函数返回 NUM，同时把全局变量 tokenval 设置成这串数字的值。例如，若输入串中7后面跟着6，则 tokenval 变量的值是76。



58

图2-26 实现图2-25中的交互

为了允许数出现在表达式中，需要对图2-19的文法做一点修改。我们用非终结符 *factor* 代替单个的数字，并引入下面的产生式和语义动作：

factor → (*expr*)
| **num** { *print*(**num**.*value*) }

图2-27中 *factor* 的C代码是上述产生式的直接实现。如果 lookahead 等于 NUM，属性 **num**. value 值由全局变量 tokenval 给出，语义动作由标准库函数 printf 完成。printf 的第一个变量是用双引号括起来的一个字符串，用来定义后面要输出的变量的格式。在这个串中，%d 表示输出下一个参数的十进制表示。图2-27中的 printf 语句输出一个空格，接着输出 tokenval 的十进制表示，再输出一个空格。

```
factor()
{
    if (lookahead == '(') {
        match('('); expr(); match(')');
    }
    else if (lookahead == NUM) {
        printf(" %d ", tokenval); match(NUM);
    }
    else error();
}
```

图2-27 操作数可以是数时的 *factor* 的C代码

图2-28给出了 lexan 函数的实现。每当第(8)~(26)行的 while 语句体被执行时，第(9)条语句读入一个字符到变量t。如果字符是空格或者制表符（即'\t'），则没有记号返回给语法分析器，只是再进行一次 while 循环。如果字符是一个换行符（'\n'），仍然没有记号返回给语法分析器，只是将全局变量 lineno 加1。lineno 用来记录输入的行数，在报错时用来指示出错行号以帮助程序调试者定位错误。

第(14)~(23)行的代码用于读入一列数字。文件 <ctype.h> 中的函数 isdigit(t)用在

第(14)行和第(17)行来判定一个输入字符t是否是数字。如果是数字，其整型值由表达式t-'0'给出（不论ASCII还是EBCDIC）。对于其他字符集，该转换可能有些不同。在2.9节，该词法分
59 析器将被加入到表达式翻译器。

```
(1)  #include <stdio.h>
(2)  #include <ctype.h>
(3)  int  lineno = 1;
(4)  int  tokenval = NONE;

(5)  int  lexan()
(6)  {
(7)       int t;
(8)       while(1) {
(9)            t = getchar();
(10)           if (t == ' ' || t == '\t')
(11)                ;    /* 去除空格和制表符 */
(12)           else if (t == '\n')
(13)                lineno = lineno + 1;
(14)           else if (isdigit(t)) {
(15)                tokenval = t - '0';
(16)                t = getchar();
(17)                while (isdigit(t)) {
(18)                     tokenval = tokenval*10 + t-'0';
(19)                     t = getchar();
(20)                }
(21)                ungetc(t, stdin);
(22)                return NUM;
(23)           }
(24)           else {
(25)                tokenval = NONE;
(26)                return t;
(27)           }
(28)      }
(29) }
```

图2-28 消去空白符并识别数的词法分析器的C代码

2.7 符号表

符号表是一种数据结构，通常用于保存源语言结构的各种信息。编译器在分析阶段收集信息放入符号表，在综合阶段使用符号表中的信息生成目标代码。例如，在词法分析阶段，形成标识符的字符串或词素被存储在符号表的一个表项中。编译器的以后各阶段会在这个表项上逐步添加其他信息，如标识符的类型、用处（如用作过程名、变量名或标号）以及存储位置。在
60 代码生成阶段，编译器使用这些信息生成存取这些变量的正确代码。在7.6节，我们将详细讨论符号表的实现和使用，这里简单地说明上一节讨论的词法分析器如何使用符号表。

2.7.1 符号表接口

与符号表有关的例程的功能主要是存取词素。当一个词素被保存时，我们也保存与该词素相关的记号。下边是在符号表上执行的操作：

- `insert(s,t)`：将字符串s和记号t的插入符号表，返回相应表项的索引。
- `lookup(s)`：到符号表中查找字符串s，如果找到则返回相应表项的索引，否则返回0。

词法分析器使用lookup操作确定某个词素的项在符号表中是否已经存在。如果不存在，它使用insert操作在符号表中建立一个新表项存储该词素及其相关信息。我们将讨论一种实现方法，让词法分析器和语法分析器都知道符号表表项的格式。

2.7.2 处理保留的关键字

上述符号表子程序能够处理任何保留的关键字的集合。例如，考虑具有 div 和 mod 词素的两个记号 **div** 和 **mod**。我们用下面的调用来初始化符号表：

```
insert("div", div);
insert("mod", mod);
```

符号表如此初始化以后，调用 lookup("div")将返回记号 **div**，于是，div 不能再被用作标识符。

任何保留关键字的集合都可以通过适当地初始化符号表而得到正确的处理。

2.7.3 符号表的实现方法

图2-29给出了一种符号表实现方法的数据结构。我们不希望预留固定大小的空间来保存形成标识符的词素，因为固定空间大小可能不足以保存长标识符，而对于短标识符（如i）又会造成空间浪费。在图2-29中，我们使用了单独的数组 lexemes 存储形成标识符的字符串。每一个字符串用一个字符串终结符EOS结束。EOS 不会出现在任何标识符中。符号表数组 symtable 中的每个表项都是一个包含两个域的记录：一个域是指向词素开始位置的指针域 lexptr，另一个域是存储记号的 token 域。符号表可以有更多的域以存储属性值，这里我们不详细讨论。

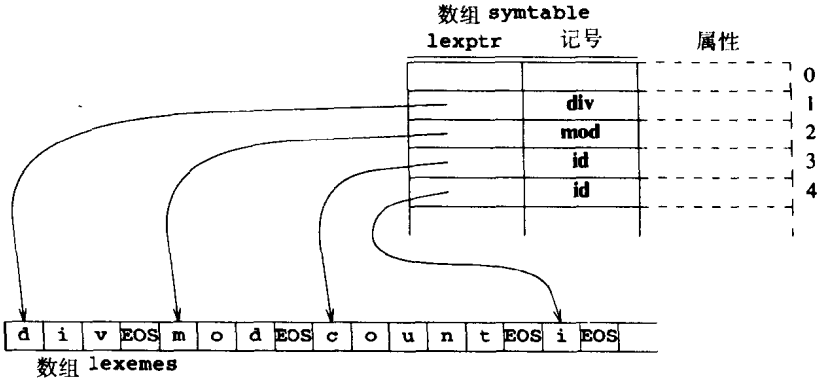


图2-29 符号表和存储字符串的数组

在图2-29中，第0项是空的，其原因是 lookup 在没有找到字符串对应的项时返回0。第1项和第2项分别对应关键字 div 和 mod。第3项和第4项分别对应标识符 count 和i。

图2-30给出了处理标识符的词法分析器的伪代码，其C语言实现见2.9节。词法分析器处理空白符和整型常数的方法和图2-28中的方法相同。

当读到一个字母时，词法分析器开始把接下来的字母和数字保存在一个叫做 lexbuf 的缓冲区中。然后，使用 lookup 操作在符号表中查找缓冲区中的字符串。因为符号表已经用关键字 div 和 mod 进行了初始化（如图2-29所示），所以，如果 lexbuf 中包含 div 或者 mod，lookup 操作将找到对应的表项。如果符号表中不存在 lexbuf 中的字符串，即 lookup 操作返回0，则 lexbuf 中包含的是一个新标识符的词素。我们用 insert 操作在符号表中创建

新标识符的表项，插入 lexbuf 中的字符串。插入完成以后，p 是符号表中存储 lexbuf 中字符串的表项的索引。通过设置全局变量 tokenval 为 p，将 p 值传递给语法分析器，并返回表项 token 域中的记号。

默认的动作是返回读入字符的整型编码作为记号。因为单字符记号没有任何属性值，所以 tokenval 设置为 NONE。

```
function lexan: integer;
var   lexbuf:   array [0..100] of char;
      c:        char;
begin
      loop begin
          读一个字符到c;
          if c是空格或制表符then
              什么也不做
          else if c是换行符then
              lineno := lineno + 1
          else if c 是一个数字 then begin
              该数字和其后数字的所表示的数的值存入tokenval;
              return NUM
          end
          else if c是一个字母then begin
              将c和其后的连续字母和数字存入lexbuf;
              p := lookup(lexbuf);
              if p = 0 then
                  p := insert(lexbuf, ID);
              tokenval := p;
              return表项p的token域
          end
          else begin  /* 记号是单个字符 */
              将tokenval置为NONE;   /* 没有属性 */
              return 字符c的整数编码
          end
      end
end
```

图2-30 词法分析器的伪代码

2.8 抽象堆栈机

如第1章所述，编译器可以划分为前端和后端两部分。前端构造源程序的中间表示，后端从中间表示生成目标代码。一种流行的中间表示是抽象堆栈机代码。编译器划分为前端和后端可以使之经简单修改就可以运行在一台新机器上。

本节描述一种抽象堆栈机，并讨论其代码是如何生成的。抽象堆栈机把指令存储器和数据存储器分开，并且所有的算术操作都在堆栈上执行。指令个数非常有限，可以分成三类：整型算术、堆栈操作和控制流。图2-31给出了一个抽象堆栈机。指针 *pc* 指明要执行的指令。下面简单讨论一下指令的含义。

2.8.1 算术指令

抽象机必须用中间语言实现每个操作符。抽象机直接支持象加法和减法这样的简单操作。更复杂的操作需要由一个抽象机指令系列来实现。为简化堆栈机的描述，我们假定每个算术操作对应一条指令。

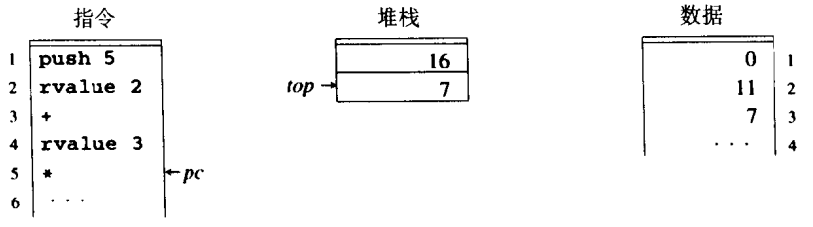


图2-31 前四条指令执行后的堆栈机快照

一个算术表达式的抽象机代码用堆栈模拟该后缀表达式的计算。这个计算过程从左到右处理后缀表达式，遇见操作数，就将其压入堆栈。当遇到一个k元操作符时，它的最左面的参数在栈顶下面$k-1$的位置，最右面的参数在栈顶。在栈顶的k个元素上应用这个k元操作符：弹出操作数，并将结果压入堆栈。例如，对后缀表达式13+5*进行计算时，需要执行下面动作：

1. 1入栈。
2. 3入栈。
3. 将栈顶的两个元素相加，从栈中弹出这两个元素，并将结果4压入堆栈。
4. 5入栈。
5. 将栈顶的两个元素相乘，从栈中弹出这两个元素，将结果20压入堆栈。

最后栈顶元素20是整个表达式的最终结果。

在中间语言中，所有的值都是整数，0可以对应于布尔值 false，非0值可以对应于布尔值 ture。布尔型操作符 and 和 or 要求其两个参数都已计算完毕。

2.8.2 左值和右值

赋值表达式左部和右部的标识符的含义是不一样的。在赋值语句

```
i := 5;
i := i + 1;
```

中，表达式的右部是一个整型值，左部是值要存放的位置。与此相似，如果p和q是指向字符的指针，表达式

61 ~ 64

```
p↑ := q↑;
```

中，右部q↑表示一个字符，左部p↑表示这个字符应该存储的位置。术语*左值*和*右值*分别指赋值表达式左部和右部对应的值。也就是说，右值是我们平常意义上的值，而左值是一个位置。

2.8.3 堆栈操作

除了明显的将常整数压入堆栈的指令和将栈顶元素弹出的指令，还有几个访问数据内存的指令：

- push v 将v压入栈顶。
- rvalue l 将存储器位置l上的数据内容压入栈。
- lvalue l 将存储器位置l的地址压入栈。
- pop 弹出栈顶元素。
- := 栈顶元素的右值被存放到栈顶的下一个元素的左值中，且二者均被弹出。
- copy 把栈顶元素的副本压入栈顶。

2.8.4 表达式的翻译

使用堆栈机计算表达式的代码与表达式的后缀表示密切相关。根据定义，表达式$E+F$的后

缀形式是 E 的后缀形式、F 的后缀形式和加号的连接。类似地，计算 $E + F$ 的堆栈机代码是计算E 的代码、计算 F 的代码以及将它们的值相加的指令的连接。因此，将表达式翻译成堆栈机代码可以通过修改2.6节和2.7节的翻译器得到。

本节生成的表达式堆栈机代码中，数据位置是用符号地址表示的。表达式a+b翻译成

```
rvalue a
rvalue b
+
```

即，把a和b位置上的数据压入栈顶，然后将栈顶的两个数据弹出，将其相加，把结果压入栈顶。

赋值表达式翻译成堆栈机代码的过程是：被赋值的标识符的左值压入栈顶，计算表达式，将结果的右值赋给标识符。例如，赋值语句

```
day := (1461*y) div 4 + (153*m + 2) div 5 + d
```
(2-17)

65 被翻译成图2-32中的代码。

赋值语句可以形式化地表示如下：

stmt → **id** := *expr*
{ *stmt.t* := 'lvalue' ‖ **id**.*lexeme* ‖ *expr.t* ‖ ':=' }

每个非终结符具有属性 t，t 给出这个非终结符的翻译。标识符 **id** 的属性 *lexeme* 给出了标识符的字符串表示。

```
lvalue day          push 2
push 1461           +
rvalue y            push 5
*                   div
push 4              +
div                 rvalue d
push 153            +
rvalue m            :=
*
```

图2-32 day := (1461*y) div 4 + (153*m + 2) div 5 + d 的翻译

2.8.5 控制流

堆栈机是顺序执行指令的，除非碰到条件指令或者无条件转移语句。说明转移目标地址的方法有如下几种：

1. 转移指令的操作数给出转移的目标地址。
2. 转移指令操作数给出转移的相对位移（正数或者负数）。
3. 用符号表示转移的目标地址，即机器所支持的标号。

在前两种方法中，操作数有可能从栈顶获得。

我们为抽象机选择第三种方法表示转移目标地址，这是因为这种方法比较容易产生控制转移，而且在产生抽象机代码以后无需改变符号地址，只需在代码上进行一些指令的插入和删除。

堆栈机的控制流指令如下：

- `label` l 说明转移的目标l，没有其他效果。
- `goto` l 从标有l的指令开始执行下一条指令。
- `gofalse` l 弹出栈顶值，如果是0，则转移到l。
- `gotrue` l 弹出栈顶值，如果非0，则转移到l。

66 - `halt` 停止执行程序。

2.8.6 语句的翻译

图2-33给出了条件语句和 while 语句的抽象机代码的框架。下面的讨论集中在标号的建立上。

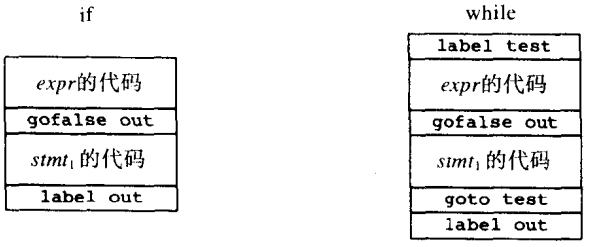


图2-33 条件语句和 while 语句的代码框架

考虑图2-33中 if 语句的代码框架。在源程序的翻译中，只允许有一个 `label out` 指令，否则，执行到 `goto out` 语句时将产生冲突而不知道将控制转到何处。因此，当翻译 if 语句时，我们需要采取某些机制，用惟一的标号替换代码框架中的 `out`。

假定 *newlabel* 是一个过程，每次调用它时，返回一个新标号。在下面的语义动作中，调用 *newlabel* 过程返回的标号保存在局部变量 *out* 中。

$$
\begin{aligned}
stmt \rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt_1 \quad \{ \quad & out := newlabel; \\
& stmt.t := expr.t \parallel \\
& \quad \texttt{'gofalse'}\ out \parallel \\
& \quad stmt_1.t \parallel \\
& \quad \texttt{'label'}\ out \}
\end{aligned}
\tag{2-18}
$$

67

2.8.7 输出一个翻译

2.5节的表达式翻译器使用 print 语句逐渐生成一个表达式的翻译。类似的 print 语句也可以用于产生一个语句的翻译。此处我们不再使用 print 语句而使用 *emit* 过程来隐藏输出细节。例如，*emit* 可能关心每个抽象机指令是否需要输出到单独一行。使用 *emit* 过程，我们可以用下面的产生式代替(2-18)：

```
stmt → if
       expr      { out := newlabel; emit('gofalse', out); }
       then
       stmt1     { emit('label', out); }
```

当产生式中出现语义动作时，我们按照从左到右的顺序考虑产生式右部的每个元素。在上面的产生式中，语义动作的顺序如下：在分析 *expr* 指令时 *out* 设置成 *newlabel* 返回的标号，然后输出 `gofalse` 指令，在分析 $stmt_1$ 语句时执行语义动作，最后 `label` 指令被输出。假设在分析 *expr* 和 $stmt_1$ 的过程中，语义动作输出了这些非终结符的代码，上面产生式实现了图2-33所示的代码框架。

翻译赋值语句和条件语句的伪代码如图2-34所示。因为 *out* 是过程 *stmt* 的局部变量，所以它的值不会受过程 *expr* 和 *stmt* 调用的影响。标号的产生需要一些考虑。假设翻译中的标号是这样的形式：`L1`，`L2`，…。伪代码把L后面跟随的整数作为的实际的转移标号。因此，*out* 被说明成一个整型量，*newlabel* 返回的整型值作为 *out* 的值，*emit* 输出编号必须是L跟随一个给定的整数。

图2-33中 while 语句的代码框架可以按类似的方式转换成代码。语句序列的翻译是简单地将各个语句连接起来，这留给读者完成。

多数单入口单出口的语法结构的翻译都和 while 语句的翻译相似。这一点我们将通过考虑表达式中的控制流来说明。

```
procedure stmt;
var test, out: integer;   /* 标号 */
begin
    if lookahead = id then begin
        emit('lvalue', tokenval); match(id); match(':='); expr
    end
    else if lookahead = 'if' then begin
        match('if');
        expr;
        out := newlabel;
        emit('gofalse', out);
        match('then');
        stmt;
        emit('label', out)
    end
        /* 此处放剩余语句的代码 */
    else error;
end
```

图2-34 翻译语句的伪代码

68

例2.10 2.7节的词法分析器包含如下形式的条件语句：

if t = **blank or** t = **tab then** · · ·

如果 t 是空格，则没有必要测试 t 是否为制表符，因为第一个等式 t = **blank** 隐含了这个条件（t = **blank or** t = **tab**）为真。因此，表达式

$expr_1$ **or** $expr_2$

可以实现为

if $expr_1$ **then true else** $expr_2$

读者可以验证下面的代码可以实现 **or** 操作：

```
expr1的代码
copy          /* 备份expr1的值 */
gotrue  out
pop           /* 弹出expr1的值 */
expr2的代码
label   out
```

gotrue 和 gofalse 指令弹出栈顶数值来简化条件语句和 while 语句的代码生成。通过备份 $expr_1$ 的值，我们可以保证：如果 gotrue 指令产生转移，则栈顶值为真。

2.9 技术的综合

本章讨论了许多用于构建一个编译器前端的语法制导技术。为了总结这些技术，本节给出一个C语言编写的翻译器，它把用分号分隔的中缀表达式序列翻译为相应的后缀表达式序列。表达式由数字、标识符、操作符（+、-、*、/、div、mod）构成。该翻译器是2.5节至2.7节中程序的扩展。本节末尾给出了完整的C语言程序。

2.9.1 翻译器的描述

该翻译器是用图2-35中的语法制导翻译模式设计的。记号 **id** 用来表示一个由字母开始的

非空字母数字序列，**num** 是一个数字序列，**eof** 是一个表示文件结束的字符。记号由空格、制表符和换行符（“空白符”）分隔。记号 **id** 的属性 *lexeme* 给出了形成该记号的字符串。**num** 的属性 *value* 给出了由 **num** 表示的整型数。 69

翻译器的代码由7个模块构成，分别存储在各自的文件里。程序的执行从 `main.c` 模块开始，该模块调用 `init()`进行初始化，然后调用 `parse()`进行翻译。其余的6个模块如图2-36所示。这7个模块中有一个全局头文件 `global.h`，它包含各模块公用的定义，其余各模块的第一条语句

```
#include "global.h"
```

使得每个模块都包含这个头文件。在给出翻译器代码之前，我们简单描述一下各个模块以及它们是如何构造的。

```
start → list eof
 list → expr ; list
      | ε
 expr → expr + term        { print('+') }
      | expr - term        { print('-') }
      | term
 term → term * factor      { print('*') }
      | term / factor      { print('/') }
      | term div factor    { print('DIV') }
      | term mod factor    { print('MOD') }
      | factor
factor → ( expr )
       | id                { print(id.lexeme) }
       | num               { print(num.value) }
```

图2-35 中缀到后缀翻译器的规格说明

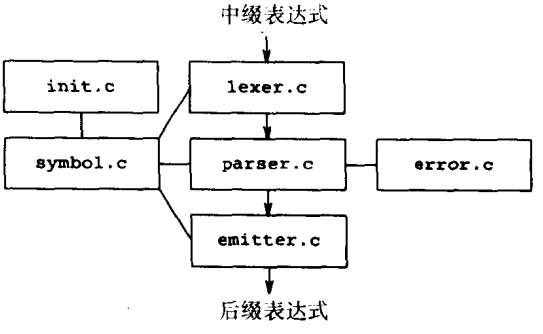


图2-36 中缀到后缀翻译器的模块

2.9.2 词法分析器模块lexer.c

词法分析器是一个称为`lexan()`的程序。语法分析器调用 `lexan()`获取记号。`lexan()`由图2-30的伪代码实现。`lexan()`每次读入一个字符，并将它发现的记号返回给语法分析器。与记号关联的属性的值被赋给全局变量 `tokenval`.

下列记号是语法分析器所需要的：

```
+ - * / DIV MOD ( ) ID NUM DONE
```

这里，`ID` 表示一个标识符，`NUM` 是一个数字，`DONE` 是文件末尾字符。空白符已被词法分析器去除。图2-37中的表给出了每个源程序语言的词素对应的记号和属性值。

词素	记号	属性值
空格 ……		
数字序列 ……	**NUM**	序列的数值
div ……	**DIV**	
mod ……	**MOD**	
其他字母打头的字母数字的字母序列 ……	**ID**	符号表索引
文件结束符 ……	**DONE**	
任何其他字符 ……	该字符	**NONE**

图2-37 记号的描述

词法分析器使用符号表程序lookup判定一个标识符词素是否曾经出现过。insert程序将新词素存储到符号表中。每当读到一个换行符，全局变量lineno加1。

2.9.3 语法分析器模块 parser.c

该语法分析器是用2.5节的技术构建的。我们首先从图2-35的翻译模式中消除左递归，以便该文法可以由递归下降语法分析器进行语法分析。转换后的翻译模式如图2-38所示。

```
      start → list eof
       list → expr ; list
            | ϵ
       expr → term moreterms
 moreterms → + term { print('+') } moreterms
            | - term { print('-') } moreterms
            | ϵ
       term → factor morefactors
morefactors → * factor { print('*') } morefactors
            | / factor { print('/') } morefactors
            | div factor { print('DIV') } morefactors
            | mod factor { print('MOD') } morefactors
            | ϵ
     factor → ( expr )
            | id { print(id.lexeme) }
            | num { print(num.value) }
```

图2-38 消除左递归后的语法制导翻译模式

我们来构建非终结符*expr*，*term*和*factor*的函数，构造方法类似于图2-24所示的方法。函数parse()实现文法的起始符号，在它需要一个新的记号时调用lexan函数。语法分析器使用emit函数产生输出并用error函数报告语法错误。

2.9.4 输出模块emitter.c

输出模块由单个函数emit(t, tval)组成，它为具有属性值tval的记号t产生输出。

2.9.5 符号表模块symbol.c和init.c

符号表模块symbol.c实现2.7节图2-29中的数据结构。数组symtable的每一项由一个指向lexemes数组的指针和一个表示记号的整数编码组成。insert(s,t)操作返回词素s（词素s构成记号t）在symtable中的索引。lookup(s)函数返回词素s在symtable中项的索引，如果s不存在，返回0。

init.c 模块用于为符号表 symtable 预加载关键字。所有关键字的词素和记号表示都保存在 keywords 数组中，keywords 数组与 symtable 数组有相同的类型。init() 函数顺序地扫描 keywords 数组，利用 insert() 操作将关键字插入符号表。这种组织方式使得关键字的记号表示容易改变。

2.9.6 错误处理模块error.c

70～72

错误处理模块负责错误的报告，这是极为基本的。一旦语法错误被发现，编译器将显示一条消息说明当前输入行出现错误，并停止分析。一种较好的错误恢复技术是使编译器跳过出错的语句，继续进行语法分析；我们鼓励读者为翻译器做这种修改。更复杂的错误恢复技术将在第4章讨论。

2.9.7 编译器的建立

各个模块的代码保存在7个文件里：lexer.c、parser.c、emitter.c、symbol.c、init.c、error.c 和 main.c。在这个C程序中，文件 main.c 包含主程序，它调用 init()，然后调用 parse()，分析成功后调用 exit(0) 返回。

在UNIX操作系统中，编译器可以通过执行命令

```
cc lexer.c parser.c emitter.c symbol.c init.c error.c main.c
```

来建立，或者使用命令

```
cc -c filename.c
```

分别编译各个文件，然后连接前面得到的*filename*.o文件：

```
cc lexer.o parser.o emitter.o symbol.o init.o error.o main.o
```

这条cc命令产生一个包含该翻译器的a.out文件。使用该翻译器的时候，键入a.out以及紧随的欲翻译表达式，如

```
2+3*5;
12 div 5 mod 2;
```

或者其他任何你喜欢的表达式。试一试吧！

2.9.8 程序清单

下面是实现该翻译器的C语言程序清单。第1个文件是全局头文件global.h，后面是7个源程序文件。为了清楚起见，该程序是用基础C语言的风格来书写的。

```
/****   global.h   ********************************************/

#include <stdio.h>      /* 输入/输出 */
#include <ctype.h>      /* 加载字符测试程序 */

#define BSIZE  128      /* 缓冲区大小 */
#define NONE   -1
#define EOS    '\0'

#define NUM    256
#define DIV    257
#define MOD    258
#define ID     259
#define DONE   260

int tokenval;      /*  记号的属性值  */
int lineno;

struct entry {     /* 符号表的表项格式 */
     char *lexptr;
     int  token;
};

struct entry symtable[];   /*  符号表  */

/****   lexer.c   *********************************************/

#include "global.h"

char lexbuf[BSIZE];
int  lineno = 1;
int  tokenval = NONE;

int  lexan()      /* 词法分析器 */
{
     int t;
     while(1) {
          t = getchar();
          if (t == ' ' || t == '\t')
               ;  /*  去除空白符  */
          else if (t == '\n')
               lineno = lineno + 1;
          else if (isdigit(t)) {       /* t是数字 */
               ungetc(t, stdin);
               scanf("%d", &tokenval);
               return NUM;
          }
```

```
        else if (isalpha(t)) {        /* t是字母 */
             int p, b = 0;
             while (isalnum(t)) {     /* t是字母或数字 */
                  lexbuf[b] = t;
                  t = getchar();
                  b = b + 1;
                  if (b >= BSIZE)
                       error("compiler error");
             }
             lexbuf[b] = EOS;
             if (t != EOF)
                  ungetc(t, stdin);
             p = lookup(lexbuf);
             if (p == 0)
                  p = insert(lexbuf, ID);
             tokenval = p;
             return symtable[p].token;
        }
74      else if (t == EOF)
             return DONE;
        else {
             tokenval = NONE;
             return t;
        }
    }
}

/****   parser.c   ******************************************/

#include "global.h"

int lookahead;

parse()     /* 分析并翻译表达式列表 */
{
    lookahead = lexan();
    while (lookahead != DONE ) {
        expr(); match(';');
    }
}

expr()
{
    int t;
    term();
    while(1)
        switch (lookahead) {
        case '+': case '-':
            t = lookahead;
            match(lookahead); term(); emit(t, NONE);
            continue;
        default:
            return;
        }
}

term()
{
    int t;
    factor();
    while(1)
        switch (lookahead) {
        case '*': case '/': case DIV: case MOD:
```

```
            t = lookahead;
            match(lookahead); factor(); emit(t, NONE);
            continue;
        default:
            return;
        }
}                                                          75
factor()
{
    switch(lookahead) {
        case '(':
            match('('); expr(); match(')'); break;
        case NUM:
            emit(NUM, tokenval); match(NUM); break;
        case ID:
            emit(ID, tokenval); match(ID); break;
        default:
            error("syntax error");
    }
}
match(t)
    int  t;
{
    if (lookahead == t)
        lookahead = lexan();
    else error("syntax error");
}

/****   emitter.c   ************************************************/

#include "global.h"

emit(t, tval)     /* 生成输出  */
    int  t, tval;
{
    switch(t) {
        case '+': case '-': case '*': case '/':
            printf("%c\n", t); break;
        case DIV:
            printf("DIV\n"); break;
        case MOD:
            printf("MOD\n"); break;
        case NUM:
            printf("%d\n", tval); break;
        case ID:
            printf("%s\n", symtable[tval].lexptr); break;
        default:
            printf("token %d, tokenval %d\n", t, tval);
    }
}

/****   symbol.c   *************************************************/

#include "global.h"

#define STRMAX 999    /* lexemes数组的大小 */                  76
#define SYMMAX 100    /* symtable的大小 */
char lexemes[STRMAX];
int  lastchar = - 1;  /* lexemes中最后引用的位置 */
struct entry symtable[SYMMAX];
int  lastentry = 0;   /* symtable中最后引用的位置 */
```

```
int  lookup(s)          /* 返回s的表项的位置 */
     char s[];
{
     int p;
     for (p = lastentry; p > 0; p = p - 1)
          if (strcmp(symtable[p].lexptr, s) == 0)
               return p;
     return 0;
}

int  insert(s, tok)    /* 返回s的表项的位置 */
     char s[];
     int  tok;
{
     int len;
     len = strlen(s); /* strlen计算s的长度 */
     if (lastentry + 1 >= SYMMAX)
          error("symbol table full");
     if (lastchar + len + 1 >= STRMAX)
          error("lexemes array full");
     lastentry = lastentry + 1;
     symtable[lastentry].token = tok;
     symtable[lastentry].lexptr = &lexemes[lastchar + 1];
     lastchar = lastchar + len + 1;
     strcpy(symtable[lastentry].lexptr, s);
     return lastentry;
}

/****   init.c   ************************************************/

#include "global.h"

struct entry keywords[] = {
     "div", DIV,
     "mod", MOD,
     0,     0
};

init()     /* 将关键字填入符号表 */
{
     struct entry *p;
     for (p = keywords; p->token; p++)
          insert(p->lexptr, p->token);
}

/****   error.c   ************************************************/

#include "global.h"

error(m)     /* 生成所有的出错信息 */
     char *m;
{
     fprintf(stderr, "line %d: %s\n", lineno, m);
     exit(1);     /* 非正常终止 */
}

/****   main.c   ************************************************/

#include "global.h"

main()
{
     init();
```

```
    parse();
    exit(0);      /* 正常终止 */
}

/*****************************************************************/
```

练习

2.1 考虑下面的上下文无关文法：

$$S \rightarrow SS+ \mid SS* \mid a$$

a) 试说明如何使用该文法生成串 aa+a*。

b) 试为 aa+a* 构建一个分析树。

c) 该文法产生的语言是什么？证明你的答案。

2.2 下面的文法产生什么语言？证明你的答案。

a) $S \rightarrow 0S1 \mid 01$
b) $S \rightarrow +SS \mid -SS \mid a$
c) $S \rightarrow S(S)S \mid \epsilon$
d) $S \rightarrow aSbS \mid bSaS \mid \epsilon$
e) $S \rightarrow a \mid S+S \mid SS \mid S* \mid (S)$

2.3 练习2.2中哪些文法具有二义性？

2.4 为下面的每种语言构建无二义性的上下文无关文法。证明你的文法是正确的。 78

a) 后缀表示的算术表达式。

b) 逗号分隔的左结合的标识符列表。

c) 逗号分隔的右结合的标识符列表。

d) 由整数、标识符、四个二元运算符（+、-、*、/）构成的算术表达式。

e) 在d中增加一元运算符 + 和 - 之后的算术表达式。

* 2.5 a) 证明：用下面文法产生的所有二进制串的值都能被3整除。（提示：对分析树节点数目使用数学归纳法。）

$$num \rightarrow 11 \mid 1001 \mid num\ 0 \mid num\ num$$

b) 上面的文法是否产生所有能被3整除的二进制串？

2.6 为罗马数字构建一个上下文无关文法。

2.7 构建一个语法制导翻译模式，将算术表达式从中缀表示翻译成前缀表示。在前缀表示中，操作符出现在操作数的前面，例如，$-xy$是$x-y$的前缀表示。给出输入9-5 + 2和9-5 *2的注释分析树。

2.8 构建一个语法制导翻译模式，将算术表达式从后缀表示翻译成中缀表示。给出输入95-2*和952*-的注释分析树。

2.9 构建一个语法制导翻译模式，将整数翻译成罗马数字。

2.10 构建一个语法制导翻译模式，将罗马数字翻译成整数。

2.11 构建练习2.2中文法 a、b、c 的递归下降语法分析器。

2.12 构建一个语法制导翻译器，验证一个输入串中的括号是成对出现的。

2.13 下面的规则定义了一个英文单词到倒读隐语（pig Latin）的翻译：

a) 如果一个单词以非空的辅音串开始，则将最前面的辅音串移到单词的末尾，并增

加一个后缀 AY；例如，pig 变成 igpay。

b) 如果单词以元音字母开始，增加一个后缀 YAY；例如，owl 变成 owlyay。

c) 跟在Q后面的U是辅音。

79 d) 如果Y在一个单词的开头，而且后面没有跟着元音，则Y是元音。

e) 单字母构成的单词不变。

为倒读隐语构造一个语法制导翻译模式。

2.14 在C程序设计语言中，for 语句具有如下形式：

for ($expr_1$; $expr_2$; $expr_3$) *stmt*

$expr_1$在循环之前执行，用来初始化循环变量。$expr_2$ 在每次循环之前测试，当表达式为0时，循环退出。循环本身由语句 {*stmt* $expr_3$; }构成。$expr_3$ 在每次循环的末尾执行，用于给循环变量增值。for 语句和下面的语句含义相似：

$expr_1$; **while** ($expr_2$) { *stmt* $expr_3$; }

构造一个语法制导翻译模式将C语言的 for 语句翻译成堆栈机代码。

* 2.15 考虑下面的 for 语句：

for i := 1 **step** 10 − j **until** 10 * j **do** j := j +1

对这条语句可以给出三种语义定义。一种可能的语义是，条件10 * j 和增量10−j 在循环之前一次性计算，如在PL/I中。例如，如果循环之前 j = 5，循环运行10次后退出。第二种可能的语义完全不同，每次循环都要重新计算条件和增量。例如，如果进入循环之前 j = 5，则循环永远不会终止。第三种语义由Algol这样的语言给出。当增量是负值时，循环终止条件的判断是 i < 10 * j ，而不是 i > 10 * j。为上面的每一种语义定义构建一个语法制导模式，将for循环语句翻译成堆栈机代码。

2.16 考虑下面的 if-then 语句和 if-then-else 语句的部分文法：

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

其中，**other** 代表语言中的其他语句。

a) 证明该文法是具有二义性的。

80 b) 构造一个等价的无二义性文法，使得 **else** 与前面最近的没有匹配的 **then** 匹配。

c) 基于该文法构造一个语法制导翻译模式，将条件语句翻译成堆栈机代码。

* 2.17 构造一个语法制导翻译模式，将算术表达式从中缀表示翻译成没有多余括号的中缀表示。给出输入((1+2) * (3 * 4)) + 5)的注释分析树。

编程练习

P2.1 基于练习2.9中开发的语法制导翻译模式，实现把整数翻译成罗马数字的翻译器。

P2.2 修改2.9节中的翻译器，使其产生2.8节的抽象堆栈机的代码作为输出。

P2.3 修改2.9节中的错误恢复模块，使得遇到错误时，跳到下一个输入表达式。

P2.4 扩展2.9节中的翻译器，使其能够处理所有Pascal表达式。

P2.5 扩展2.9节中的编译器，将下面文法生成的语句翻译成堆栈机代码：

stmt → **id** := *expr*

| **if** *expr* **then** *stmt*
| **while** *expr* **do** *stmt*
| **begin** *opt_stmts* **end**

opt_stmts → *stmt_list* | ϵ

stmt_list → *stmt_list* ; *stmt* | *stmt*

* P2.6 构造2.9节编译器的一组测试表达式，使得在导出某个测试表达式时，每个产生式至少使用一次。构造一个能作为通用编译器测试工具的测试程序。使用你的测试程序在你的测试表达式上运行你的编译器。

P2.7 构造练习P2.5中编译器的一组测试语句，使得在产生某个测试语句时，每个产生式至少使用一次。使用练习P2.6中的测试程序在你的测试表达式上运行你的编译器。

参考文献注释

本章是介绍性的一章，涉及了许多随后各章要详细介绍的主题。这些章节的参考文献将包含更丰富的素材。

Chomsky[1956] 将上下文无关文法作为研究自然语言的一部分提出。上下文无关文法在定义程序设计语言语法方面的应用是独立出现的。在起草Algol 60的工作中，John Backus 使用了 Emil Post 产生式（Wexelblat[1981,p.162]）。这种表示法是上下文无关文法的一个变种。学者 Panini 提出了一种等价的语法表示，用来说明公元前400年到公元前200年之间出现的 Sanskrit 文法的规则（Ingerman [1967]）。 81

BNF 最初是 Backus Normal Form 的缩写，读作 Backus-Naur 范式，其目的是纪念 Algol 60报告的作者 Naur 的贡献（Naur [1963]）。BNF 最早出现在 Knuth[1964]中。

语法制导定义是一种语义结构上的归纳定义，长期以来一直非正式地用于数学中。随着 Algol 60 报告中结构文法的使用，才将其应用到程序设计语言中。不久，Irons[1961]构造了一个语法制导编译器。

递归下降语法分析从20世纪60年代初就已经被使用了。Bauer[1976] 将这种方法归功于 Lucas[1961]。Hoare[1962b, p.128] 描述了一个Algol编译器，这个编译器是“一组过程，每个过程能够处理Algol 60报告中的一个语义单元”。Foster[1968] 讨论了在不影响属性值的前提下，消除包含语义动作的产生式中的左递归。

McCarthy[1963] 建议语言的翻译可以基于抽象语法。在同一篇文章McCarthy[1963, p.24] 里试图让读者相信，一个尾递归的阶乘函数的公式和一个循环程序是等价的。

将一个编译器分成前端和后端的好处是由 Strong 等人（Strong et al. [1958]）在委员会报告中首次提出的。该报告给通用的中间语言提出了一个名字UNCOL（universal computer oriented language）。这个概念一直是一种理想。

学习实现编译器技术的一个好的方法是阅读现有的编译器的程序代码。不幸的是，代码通常是不公开的。Randell and Russell[1964] 给出了一个复杂的早期 Algol 编译器的代码。编译器的代码也可以从 McKeeman, Horning and Wortman[1970]中找到。Barron[1981] 是关于 Pascal 语言实现的论文集，包括 Pascal P 编译器（Nori et al. [1981]）的实现注释、代码生成细节（Ammann[1977]）、Pascal S的实现代码、Wirth[1981] 为学生使用设计的 Pascal 子集。Knuth[1985] 极其清晰详细地描述了 TEX 翻译器。

Kernighan and Pike[1984] 详细地描述了如何使用 UNIX 上可用的编译器构建工具围绕语法制导翻译模式来建立一个桌面计算器程序。等式（2-17）来自于 Tantzen[1963]。 82

第3章 词法分析

本章主要讨论词法分析器的说明和实现技术。实现词法分析器的简单方法包括两步：首先建立一张描述源语言记号的结构的图。然后，手工地把这张图翻译成能够识别源语言记号的程序。用这种方法可以产生有效的词法分析器。

这种实现词法分析器的技术也经常用于其他领域，如查询语言与信息检索系统。在每个应用中，最基本的问题是如何设计与说明一种特殊的程序，它能够完成由字符串中的模式触发的动作。因为模式制导程序设计方法的应用非常广泛，我们介绍一种用于说明词法分析器的“模式-动作”语言Lex。在这种语言中，模式用正规表达式说明，而且Lex语言的编译器能够产生一个识别正规表达式的有穷自动机识别器。

除了Lex以外，还有一些语言也是用正规表达式来描述模式的。例如，模式扫描语言AWK利用正规表达式来选择输入行进行处理，UNIX系统的shell允许用户通过正规表达式指定一组文件名，如UNIX命令`rm *.o`用来删除所有文件名以“.o”结尾的文件。[⊖]

词法分析器的自动生成工具可以使具有不同背景的人员在他们各自的应用领域中使用匹配的模式。例如，Jarvis[1976]用词法分析器的生成器生成一个程序，用于识别印制电路板中的缺陷。电路是数字扫描的并转换成不同角度的线段的“串”。“词法分析器”在线段的“串”中寻找对应于这些缺陷的模式。词法分析器的生成器的最大优点是它能利用最著名的模式匹配算法为那些不精通模式匹配技巧的人产生有效的词法分析器。

3.1 词法分析器的作用

词法分析是编译的第一阶段。词法分析器的主要任务是读入输入字符，产生记号序列，提交给语法分析使用。这种交互（示意性地总结在图3-1中）通常可以通过使词法分析器作为语法分析器的子程序或协作程序来实现。当词法分析器收到语法分析器发出的“取下一个记号”的命令时，词法分析器读入输入字符，直到识别出下一个记号。

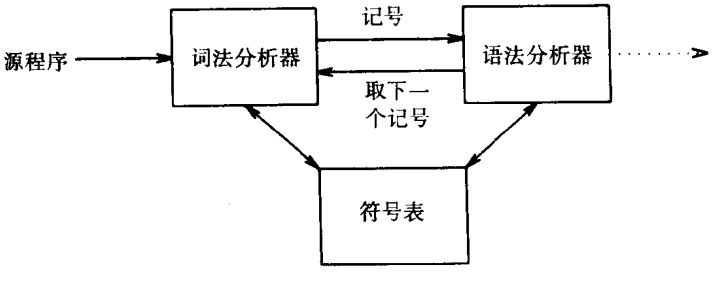


图3-1 词法分析器与语法分析器之间的交互

词法分析器是编译器中读入源程序的部分，因此它还可以完成一些相关的辅助任务。一个任务是滤掉源程序中的注释、空格、制表符、换行符；另一个任务是使编译器能将发现的错误

⊖ 表达式*.o是常用的正规表示法中的一个变种。练习3.10和练习3.14中给出了某些正规表达式的常用表示法。

信息与源程序的出错位置联系起来。例如，词法分析器负责记录遇到的换行符的个数，以便将行号与出错信息联系起来。在某些编译器中，词法分析器负责拷贝一份源程序，并将出错信息加入其中。如果源语言支持宏预处理功能，可以在词法分析阶段完成这些预处理功能。

有时，词法分析器可以分两阶段：第一个阶段是扫描阶段，第二个阶段是词法分析阶段。扫描程序负责完成一些简单的任务，词法分析器要完成比较复杂的任务。例如，Fortran编译器可以使用扫描程序从输入中清除空格。

3.1.1 词法分析中的问题

把编译过程的分析阶段划分为词法分析和语法分析的原因如下：

1. 简化编译器的设计可能是最重要的考虑。词法分析和语法分析分离可以简化两者的设计。
84 例如，如果把空白符和注释的处理包含在语法分析时一并考虑，而不是由词法分析器来完成，将会使语法分析器的结构变得十分复杂。可见，把词法分析从语法分析中分离出来会有利于我们简化这一阶段的设计。

2. 提高编译器的效率。把词法分析独立出来使我们能构造更有效的专门的处理器。编译的大部分时间消耗在读源程序并将其切分为记号方面。采用专门的缓存技术来进行输入字符串的读取和记号的处理可以显著地提高编译器的性能。

3. 增强编译器的可移植性。与设备有关的特征以及语言的字符集的特殊性的处理可以被限制在词法分析器中。类似于Pascal语言中“↑”的特殊的或非标准的符号的表示可以在词法分析器中解决，不会影响编译器其他部分的设计。

把词法分析和语法分析分开后，可以借助很多工具来自动地构造它们。本书将介绍这类工具的几个例子。

3.1.2 记号、模式、词素

在词法分析的讨论中，我们使用术语“记号”、“模式”、“词素”表示特定的含义。图3-2是使用它们的例子。通常，在输入中有一组字符串会产生相同的记号（作为输出），这个字符串构成的集合由一个与该记号相关联的称为模式的规则来描述。这个模式被说成匹配该集合中的每个字符串。词素是源程序的字符序列，由一个记号的模式来匹配。例如，Pascal语句

```
const pi = 3.1416;
```

中的子串pi是记号“标识符”的词素。

记号	词素示例	模式的非形式描述
const	`const`	`const`
if	`if`	`if`
relation	`<, <=, =, <>, >, >=`	<或<=或=或<>或>=或>
id	`pi, count, D2`	字母打头的字母数字串
num	`3.1416, 0, 6.02E23`	任何数字常数
literal	`"core dumped"`	在"与"之间除"以外的任何字符

85 图3-2 记号的例子

我们把记号作为源语言文法的终结符，用黑体名字表示记号。由记号的模式所匹配的词素表示源程序的字符串，它们是词法单位。

在多数程序设计语言中，下列结构被处理为记号：关键字、操作符、标识符、常量、文字串和标点符号（如括号、逗号和分号）。在上面的例子中，当源程序中出现字符串pi时，一个

表示标识符的记号将返回给语法分析器。记号的返回通常是通过传递代表这个记号的整数来实现的。在图3-2中这个整数被指定为黑体的 **id**。

模式是描述源程序中表示特定记号的词素集合的规则。图3-2中的记号 **const** 的模式是一个字符串const，它是一个关键字。记号 **relation** 的模式是6个Pascal关系操作符的集合。为了能精确地描述 **id**（表示标识符）和 **num**（表示数）这样更复杂的记号，我们将使用3.3节介绍的正规表达式。

某些语言的约定给词法分析带来了困难。例如，Fortran语言要求某些结构出现在输入行的固定位置，于是词素对准（alignment）对确定源程序的正确性非常重要。现代语言设计的倾向是自由格式输入，允许各种结构出现在输入行的任何地方。因此这个问题在词法分析中不再是重要的了。

不同的语言在空格的处理上有较大的差别。在一些语言（如Fortran和Algol 68）中，空格无意义（文字串中除外）。在程序中可以随意加入空格来改善其可读性。对空格的约定增加了识别记号的复杂性。

Fortran语言的 DO 语句可以作为典型例子来说明它给识别记号带来的困难，在语句

```
DO 5 I = 1.25
```

中，一直到看见小数点后，才能确定 DO 不是关键字，而是标识符 DO5I 的一部分。另一方面，在语句

```
DO 5 I = 1,25
```

中，有7个记号：关键字 DO、语句标号5、标识符I、操作符=、常数1、逗号和常数25。在该语句中，没看见逗号之前，我们也不敢确定 DO 是关键字。为了减轻这种不确定性，Fortran 77 允许标号和 DO 语句循环变量之间有一个可选的逗号。鼓励使用这个逗号，因为使用这个逗号可以使 DO 语句更清晰可读。

很多语言规定某些字符串是保留的，即它们的含义是预定义的，不能由用户改变。如果关 86
键字不是保留的，那么词法分析器必须能区分出关键字和用户自定义的标识符。在PL/I语言中关键字不是保留的，因而把关键字从标识符中区别出来的规则相当复杂，从下面的PL/I语句我们就可以清楚地看出这一点。

```
IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;
```

3.1.3 记号的属性

如果不止一个记号的模式能匹配一个词素，词法分析器必须为这个记号提供附加的关于匹配的特殊词素的信息。例如，模式**num**既能匹配字符串0，也能匹配字符串1，此时代码生成器需要知道**num**到底匹配了哪一个字符串。

词法分析器把与记号有关的信息收集到记号的属性中。记号影响语法分析，而属性影响记号的翻译。在实际实现时，记号通常只有一个属性，即指向符号表中一个表项的指针，与记号有关的信息保存在这个对应的表项中。为了诊断错误，我们不仅需要知道匹配标识符的词素，而且还需要知道这个词素第一次出现的行号。这些信息都可以存储在符号表中该标识符对应的表项内。

例3.1 Fortran语句

```
E = M * C ** 2
```

中的记号和它们的属性值可用二元组序列表示如下：

<**id**，指向符号表中与E相关的表项的指针>

<**assign_op**, >

<**id**，指向符号表中与M相关的表项的指针>

<**mult_op**, >

<**id**, 指向符号表中与C相关的表项的指针>

<**exp_op**，>

<**num**，整数值2〉

注意，某些二元组不需要属性值，它的第一个分量足以标识词素。在上述例子中，记号**num**的属性是一个整数值。当然，编译器也可以把形成数的字符串存入符号表中，并让记号**num**的属
87 性是指向符号表中相应表项的指针。 □

3.1.4 词法错误

因为词法分析器不能从全局的角度考察源程序，所以能在词法层发现的错误是有限的。如果词法分析器在如下的C程序中第一次遇到`fi`

```
fi ( a == f(x) ) ···
```

它无法区别`fi`究竟是关键字`if`的错误拼写还是一个未声明的函数标识符。由于`fi`是合法的标识符，词法分析器必须返回该标识符的记号，而让编译器的其他阶段去处理这种错误。

有时会出现由于剩余输入的前缀不能和任何记号的模式匹配而使词法分析器无法处理的情况。此时，最简单的错误恢复策略也许是“紧急方式”恢复，即反复删掉剩余输入最前面的字符，直到词法分析器能发现一个正确的记号为止。这种恢复技术可能会给语法分析带来一些麻烦，但在交互计算环境中是非常有效的。

其他错误恢复动作包括：

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。

这样的错误变换可以用于对输入错误的修补。最简单的策略是看一下剩余输入的前缀能否通过上面的一个变换变成一个合法的词素。这种策略假设大多数词法错误是多、漏或错了一个字符或者相邻的两个字符错位的结果。事实上，这种假设通常（但不总是）是正确的。

在程序中发现错误的一种方法是计算把一个错误程序转换成一个语法上正确的程序所需要的错误变换个数的最小值。当把一个错误程序转换成一个正确程序所需的最短的错误变换序列长度为k时，我们说这个程序有k个错误。最小距离错误校正是一种理论上的标准，但因其实现起来代价太高，实际上并不常用。然而，一些试验性的编译器在进行局部校正时确实用到了最小距离标准。

3.2 输入缓冲

本节讨论与输入缓冲有关的效率问题。我们首先介绍一种双缓冲输入方案，这种方案在为识别记号而需要进行超前扫描的情况下非常有用。然后，我们介绍一些其他提高词法分析器效
88 率的技术，如使用标志（sentinel）标记缓冲区边界。

实现词法分析器最常用的三种方案如下：

1. 使用词法分析器生成器（如将要在3.5节介绍的Lex编译器），从基于正规表达式的说明自动产生一个词法分析器。在这种情况下，由生成器提供子程序实现输入流的读取和缓冲。

2. 使用传统的程序设计语言编写词法分析器，并使用该语言提供的I/O功能对输入流进行读取。

3. 使用汇编语言编写词法分析器，并显式地控制输入流的读取。

这三种方案是按照实现难度递增的次序来排列的。不幸的是，构造的词法分析器效率越高，构造的难度就越大。尽管编译器的后面各阶段在概念上更复杂，但由于词法分析器是编译器中惟一的逐个字符读取源程序的阶段，所以它可能会耗费大量的时间。因此，在设计编译器时，词法分析器的速度是一个关键因素。本章将用很大篇幅介绍实现词法分析器的第一种方案，即利用自动生成器自动生成词法分析器的方案。此外还将介绍一些对手工编写词法分析器有用的技术。3.4节将讨论状态转换图。在手工设计词法分析器时，状态转换图是非常有用的概念。

3.2.1 双缓冲区

对很多源语言来说，在一个词素被一个模式匹配上之前，词法分析器往往需要超前扫描该词素后面的若干字符。第2章的词法分析器使用ungetc函数将超前扫描的字符退回到输入流中。然而，这种方法需要大量的移动字符时间。因此人们开发出一些特殊的缓冲技术以减少这种时间开销，但是，这些缓冲技术在某种程度上依赖于系统的参数，因而，本节只能简单地介绍一下这类技术的原理。

我们把一个缓冲区分成两个部分，每部分能容纳 N 个字符，如图3-3所示。一般来说，N 是一个磁盘块中字符的个数，如1024或4096。

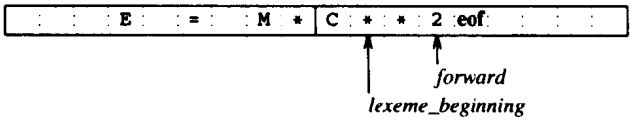


图3-3 分成两部分的输入缓冲区

89

我们每次用一个系统读命令向缓冲区的每半部分读入 N 个字符，而不是每读入一个字符调用一次读命令。如果剩余的输入数据不足 N 个字符，则在缓冲区中最后一个输入字符后面会读进来一个特殊字符 **eof**。**eof** 不同于任何其他的输入字符，它用于标识源文件的结尾，如图3-3所示。

输入缓冲区包括两个指针，在两个指针之间的字符串就是当前的词素。一开始，两个指针都指向下一个要识别的词素的第一个字符上。然后，其中一个指针（即向前（*forward*）指针）向前扫描，直至发现一个与某个模式匹配的词素为止。一旦一个词素被确定，向前指针将指向它的最右字符。在处理完这个词素后，两个指针同时定位到这个词素的一个字符。在这种策略中，注释和空白符可以由不生成记号的模式来匹配。

如果向前指针将要移过缓冲区的中间标记，则往缓冲区的右半部读入 N 个新字符。如果向前指针将要移过缓冲区的右端，则往其左半部读入 N 个新字符，且将向前指针绕回到缓冲区的头部继续处理。

这种缓冲机制在多数情况下都非常有效，但限制了超前扫描的数量。在超前扫描时，若向前指针需要移动的距离超过了缓冲区的长度，词法分析器就无法识别出记号。例如，如果在对PL/I 程序中的我们看到

```
DECLARE ( ARG1, ARG2, . . . , ARGn )
```

在扫描到右括号后面的字符之前，我们无法确定DECLARE是关键字还是数组名称。在这两种情况下，词素都在第二个E字符处结束。需要超前扫描的字符数量与参数个数成正比，而参数个数在原则上是不限制的。

3.2.2 标志

如果我们采用图3-3的模式，在每次移动向前指针时都必须检查是否到了缓冲区某半部分的末尾，若是，则需重装缓冲区的另半部分，即需要按照图3-4的算法来移动向前指针。

```
if forward在缓冲区第一部分末尾 then begin
    重装缓冲区第二部分;
    forward := forward + 1
end
else if forward在缓冲区第二部分末尾 then begin
    重装缓冲区第一部分;
    将forward移到缓冲区第一部分开始
end
else forward := forward + 1
```

图3-4 移动向前指针的算法

如果先前指针不在缓冲区某半部分的末尾，图3-4的代码每次移动向前指针时都需要做两次测试。如果我们在缓冲区两部分的结尾处各设置一个标志字符，则可以减少一次测试。这个“标志”必须是源语言词素集合的特殊字符。一个比较自然的选择就是 **eof**。图3-5的缓冲区分配与图3-3一致，只是在其基础上加入了“标志”字符。

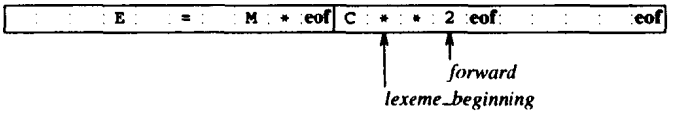


图3-5 在两半部分的末尾加入“标志”的缓冲区

对图3-5所示的缓冲区分配，我们可采用图3-6中的算法来移动向前指针并测试源文件是否结束。在大部分情况下，算法只需测试一次，以判断 *forward* 指针是否已经指向 **eof**。只有当到达缓冲区半部分的末尾或源文件尾时，算法才需要进行更多的测试。由于在两个 **eof** 之间有 N 个字符，每个输入字符的平均检查次数近似于1。

```
forward := forward + 1
if ↑ forward := eof then begin
    if forward在第一部分末尾 then begin
        重装第二部分;
        forward := forward + 1
    end
    else if forward 在第二部分末尾 then begin
        重装第一部分;
        将 forward 移到第一部分开始
    end
    else /* eof 在表示输入结束的缓冲区中 */
        终止词法分析
end
```

图3-6 包含标志的缓冲区中移动向前指针的算法

我们还需要确定如何处理向前指针当前所指的字符。该字符可能是一个记号的结尾标志，也可能是正被识别的关键字的一部分，还可能是其他各种情况。如果所选的实现语言支持 case语句，则可以用它来完成这些测试。例如，我们可用 case 语句的一个分支来实现测试：

```
if forward↑ = eof
```

3.3 记号的描述

正规表达式是表示模式的一种重要方法。每个模式匹配一个字符串集。因此正规表达式将作为字符串集的名字。3.5节将把这种表示法扩展为词法分析的模式制导语言。

3.3.1 串和语言

术语字母表或字符类表示有限符号的集合。符号的典型例子是字母和字符。集合{0,1}是

二进制字母表。ASCII 和 EBCDIC 是两个计算机字母表实例。

字母表上的字符串是该字母表中符号的有穷序列。在语言理论中，术语句子和字常作为“符号串”的同义词。符号串 s 的长度是出现在 s 中的符号的个数，通常记作$|s|$。例如，banana 是长度为6的符号串。空字符串是长度为0的特殊符号串，用ϵ表示。图3-7概括了用于表示字符串各部分的常用术语。

术　语	定　义
s 的前缀	去掉串 s 尾部的0个或多个符号后得到的字符串。例如，ban是banana的前缀
s 的后缀	去掉串 s 头部的0个或多个符号后得到的字符串。例如，nana是banana的后缀
s 的子字符串	去掉 s 的一个前缀和一个后缀后得到的字符串。例如，nan是banana的一个子串。s 的每个前缀和后缀都是 s 的一个子串，但子串并不总是 s 的前缀或后缀。对于每个字符串s，s和ϵ是 s 的前缀、后缀和子串
s 的真前缀（真后缀、真子串）	如果非空串 x 是串 s 的前缀（后缀、子串），而且 $s \neq x$，则称 x 是 s 的真前缀（真后缀、真子串）
s 的子序列	从串 s 中删除0个或多个符号后得到的串（这些被删除的符号可以不相邻）。例如，baaa是banana的子序列

图3-7 字符串的各部分的术语

语言是给定字母表上的任意一个字符串集合。这个定义是广义的。像空集和仅包含空符号串的集合$\{\epsilon\}$这样的抽象语言也符合这个定义。所有语法正确的Pascal程序的集合和所有语法正确的英语句子的集合也都符合此定义，当然，要描述后两个集合要难得多。注意，这个定义并没有把任何意义赋予语言中的符号串，这个问题将在第5章讨论。

如果 x 和 y 是符号串，那么 x 和 y 的连接（记作 xy）是把 y 连接到 x 后面所形成的符号串。例如，如果 x = dog 且 y = house，那么 xy = doghouse。对连接运算而言，空符号串是一个单位元，也就是说，$s\epsilon = \epsilon s = s$。

如果把两个符号串的连接看成是这两个串的“乘积”，我们可以定义符号串的“指数”如下：定义 s^0 为ϵ，对于 $i>0$，s^i 为 $s^{i-1}s$，因为ϵs就是 s 本身，所以 $s^1 = s$，$s^2 = ss$，$s^3 = sss$，依此类推。

3.3.2 语言上的运算

有一些重要的运算可以应用到语言中。对词法分析而言，我们感兴趣的是并、连接和闭包运算，图3-8中给出了这些运算的定义。我们还可以将“指数”运算符推广到语言上，即把L^0定义为$\{\epsilon\}$，L^i定义为$L^{i-1}L$，即L连接自己i-1次。

例3.2 令 L 表示集合{A, B, …, Z, a, b, …, z}，D 表示集合{0, 1, …, 9}。我们可以将L看成是由大、小写字母组成的字母表，将D看成是10个数字组成的字母表。同时，由于单个符号也可以看成长度为1的符号串，我们可以把 L 和 D 分别看成是有穷的语言集。下面是将图3-8中定义的运算作用于 L 和 D 所得到新语言：

1. $L \cup D$ 是字母和数字的集合。
2. LD 是一个字母后随一个数字的符号串的集合。
3. L^4 是由四个字母构成的符号串的集合。
4. L^* 是所有字母构成的串（包括ϵ）的集合。

90
~
93

5. $L(L \cup D)^*$ 是所有以字母开头的字母数字串的集合。

6. D^+ 是由一个或多个数字构成的数字串的集合。 □

运　算	定　义
L 和 M 的并(记作 $L \cup M$)	$L \cup M = \{s \mid s$属于L或s属于$M\}$
L 和 M 的连接(记作 LM)	$LM = \{st \mid s$属于L且t属于$M\}$
L 的克林（Kleene）闭包(记作 L^*)	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^*表示0个或多个L的连接
L 的正闭包 (记作 L^+)	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+表示1个或多个L的连接

图3-8 语言上的运算的定义

3.3.3 正规表达式

在Pascal语言里，标识符是一个字母后跟随零个或多个字母或数字组成的符号串，即一个标识符是例3.2中5所定义的集合中的元素。本节将介绍一种叫正规表达式的表示法，它使得我们能够精确地定义这样的集合。使用这种表示法，Pascal的标识符集可以定义为

letter (letter | digit) *

其中竖线的含义是“或”，括号用于把子表达式组在一起，星号的含义是“零个或多个”括号中的表达式，**letter** 和 **(letter | digit)*** 的并列表示两者的连接。

建立正规表达式时，可以先定义简单的正规表达式，然后用它们构造出更复杂的正规表达式。每个正规表达式 r 表示一个语言 $L(r)$。这些定义规则说明 $L(r)$ 是怎样由 r 的子表达式所表示的语言以不同的方式组合形成的。

下面是定义字母表 Σ 上的正规表达式的规则，每一条规则后带有所定义的正规表达式所表示的语言的一个说明：

1. ϵ 是正规表达式，它表示 $\{\epsilon\}$，即包含空串的集合。

2. 如果 a 是 Σ 上的符号，那么 a 是正规表达式，表示 $\{a\}$，也就是包含符号串 a 的集合。虽然，我们使用相同的表示法，但正规表达式 a、符号串 a 和符号 a 这三者的含义是不同的，
94 我们可以从上下文中清楚地区分出所谈到的 a 的具体含义。

3. 假定 r 和 s 都是正规表达式，分别表示语言 $L(r)$和 $L(s)$，则：

a) $(r) \mid (s)$ 是正规表达式，表示 $L(r) \cup L(s)$。

b) $(r)(s)$ 是正规表达式，表示 $L(r)L(s)$。

c) $(r)^*$ 是正规表达式，表示 $(L(r))^*$。

d) (r) 是正规表达式，表示 $L(r)$。[⊖]

正规表达式表示的语言叫做正规集。

正规表达式的说明是一种递归定义。规则1、2是定义的基础。我们把 ϵ 和出现在正规表达式中的 Σ 中的符号称为基本符号。规则3提供了归纳的步骤。

如果采用如下约定：

⊖ 这条规则说明如果需要，我们可以用括号把正规表达式括起来。

1. 一元运算符 * 具有最高的优先级，并且是左结合的。

2. 连接的优先级次之，也是左结合的。

3. | 的优先级最低，同样是左结合的。

那么，在正规表达式中可以避免一些不必要的括号。在此约定下，$(a)|((b)^*(c))$ 等价于$a|b^*c$。这两个表达式都表示由单个 a 构成的符号串或者由0个或多个 b 后面跟着一个 c 组成的符号串集合。

例3.3 令$\Sigma = \{a, b\}$。

1. 正规表达式$a|b$表示集合$\{a, b\}$。

2. 正规表达式 $(a|b)(a|b)$ 表示 $\{aa, ab, ba, bb\}$，即由 a 和 b 组成的长度为2的符号串集合。表示同样集合的另一正规表达式是 $aa|ab|ba|bb$。

3. 正规表达式 a^* 表示由零个或多个a组成的所有串的集合$\{\epsilon, a, aa, aaa, \cdots\}$。

4. 正规表达式 $(a|b)^*$ 表示由零个或多个 a 或 b 构成的符号串集合，即由 a 和 b 构成的所有符号串的集合。这个集合也可用另一个正规表达式 $(a^*b^*)^*$ 来表示。

5. 正规表达式 $a|a^*b$ 表示包含串 a 和零个或多个 a 后跟随一个 b 构成的符号串集合。 □

如果两个正规表达式 r 和 s 表示同样的语言，则称 r 和 s 等价，记作 $r = s$。例如，$(a|b) = (b|a)$。

正规表达式遵循一些代数定律，它们可以用于正规表达式的等价变换，图3-9是正规表达式 r、s 和 t 遵循的代数定律。 95

公　理	描　述
$r\|s = s\|r$	| 是可交换的
$r\|(s\|t) = (r\|s)\|t$	| 是可结合的
$(rs)t = r(st)$	连接是可结合的
$r(s\|t) = rs\|rt$ $(s\|t)r = sr\|tr$	连接对 | 是可分配的
$\epsilon r = r$ $r\epsilon = r$	ϵ 是连接的单位元
$r^* = (r\|\epsilon)^*$	*和 ϵ 间的关系
$r^{**} = r^*$	*是幂等的

图3-9 正规表达式的代数性质

3.3.4 正规定义

为表示方便，我们可能希望为正规表达式命名，并用这些名字来定义正规表达式，就如同它们也是符号一样。如果Σ是基本符号的字母表，那么正规定义是如下形式的定义序列：

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\cdots$$
$$d_n \rightarrow r_n$$

其中，每个 d_i 都是一个名字，并且它们各不相同，每个 r_i 是$\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$（即基本符号和前面定义的名字）中符号上的正规表达式。由于限制了每个 r_i 中只含有 Σ 中的符号和在它之前定义的名字，所以我们可以通过反复地用名字所代表的正规表达式替代该名字的方法为任何一个 r_i 构造 Σ 上的正规表达式。如果 r_i 用到了 d_j，并且 $j \geq i$，则 r_i 是递归定义的，而且这

个替换过程不会中止。

为了区别名字和符号，用黑体字表示正规定义中的名字。

例3.4 如前面所述，Pascal语言的标识符集合是以字母开头的字母数字串的集合，这个集合的正规定义是：

```
letter → A | B | ··· | Z | a | b | ··· | z
 digit → 0 | 1 | ··· | 9
    id → letter ( letter | digit )*
```

□

96

例3.5 Pascal语言中的无符号数是形如5280、39.37、6.336E4或1.894E-4这样的符号串。下面的正规定义给出了这类符号串的精确说明：

```
            digit → 0 | 1 | ··· | 9
           digits → digit digit*
 optional_fraction → . digits | ϵ
optional_exponent → ( E ( + | - | ϵ ) digits ) | ϵ
              num → digits optional_fraction optional_exponent
```

在这个定义中，**optional_fraction** 是空串（空缺）或小数点后再跟上一个或多个数字。如果 **optional_exponent** 不是空串，则是 E 后随一个可选的 + 号或 - 号，再跟上一个或多个数字。注意，小数点后至少要有一个数字，所以 **num** 不能匹配 1.，但能匹配 1.0。 □

3.3.5 缩写表示法

在正规表达式中，某些结构出现频繁，为方便起见，我们可以用缩写形式表示它们。

1. 一个或多个实例。一元后缀操作符 + 的意思是“一个或多个实例”。如果 r 是表示语言 $L(r)$的正规表达式，那么 $(r)^+$ 是表示语言 $(L(r))^+$ 的正规表达式。正规表达式 a^+表示由一个或多个a构成的所有串的集合。操作符 + 和操作符 * 具有同样的优先级和结合性。代数恒等式$r^* = r^+ | \epsilon$ 与 $r^+ = rr^*$ 表达了克林闭包 * 和正闭包 + 这两个操作符间的关系。

2. 零个或一个实例。一元后缀操作符 ? 的意思是“零个或一个实例”。$r?$ 是$r | \epsilon$ 的缩写形式。如果r是正规表达式，则 $(r)?$ 是表示语言 $L(r) \cup \{\epsilon\}$的正规表达式。例如，使用 + 和 ? 操作符，可以重写例3.5中 **num** 的正规定义如下：

```
            digit → 0 | 1 | ··· | 9
           digits → digit+
 optional_fraction → ( . digits )?
optional_exponent → ( E ( + | - )? digits )?
              num → digits optional_fraction optional_exponent
```

3. 字符类。[abc] (其中a、b和c是字母表中的符号) 表示正规表达式 a|b|c。缩写的字符类 [a-z] 表示正规表达式 a|b|···|z。使用字符类，我们可以用下述正规表达式描述标识符：

97

```
[A-Za-z][A-Za-z0-9]*
```

3.3.6 非正规集

某些语言不能用正规表达式描述。为了说明正规表达式的描述能力有限，我们给出一些不能用正规表达式表示的程序设计语言结构的例子。对这些结论的证明，见参考文献。

正规表达式不能用于描述均衡或嵌套结构。例如，具有配对括号的符号串集合不能用正规

表达式描述，但它们可以用上下文无关文法来描述。

重复符号串不能用正规表达式表示。集合 {*wcw*| *w* 是 *a* 和 *b* 组成的串} 不能用正规表达式描述，也不能用上下文无关文法来说明。

正规表达式只能表示固定次数的重复或给定结构的没有指定次数的重复。由于正规表达式不能比较任意两个数是否相等，因此我们不能用正规表达式描述早期 Fortran 语言中形如 $n\mathrm{H}a_1a_2\cdots a_n$ 的 Hollerith 字符串，因为H后面的字符数目要等于H前面的十进制数。

3.4 记号的识别

上一节我们关心的是如何描述记号，这一节将讨论怎样识别记号。本节将以下述文法定义的语言作为例子来讨论记号的识别。

例3.6 考虑下述文法片断：

```
stmt → if expr then stmt
     | if expr then stmt else stmt
     | ε
expr → term relop term
     | term
term → id
     | num
```

其中，终结符 **if**、**then**、**else**、**relop**、**id** 和 **num** 产生由以下正规定义给出的串的集合：

```
   if → if
 then → then
 else → else
relop → < | <= | = | <> | > | >=
   id → letter ( letter | digit )*
  num → digit+ ( . digit+ )? ( E( +|- )? digit+ )?
```

98

其中，**letter** 和 **digit** 的定义与前面相同。

对这个给定的语言，词法分析器将识别关键字 if、then、else和由 **relop**（关系操作符）、**id**（标识符）和 **num**（数）表示的词素。为简单起见，我们假定关键字是保留的，也就是说，它们不能作为标识符使用。类似于例3.5，这里的 **num** 表示 Pascal 中的无符号整数和实数。

此外，我们还假定词素由空白符分隔。空白符是空格、制表符、换行符组成的非空序列。词法分析器还要完成去掉空白符的任务。这个任务通过把输入串与如下的 **ws** 正规定义相比较来完成：

```
delim → blank | tab | newline
   ws → delim+
```

如果发现了与 **ws** 匹配的字符串，则词法分析器不返回记号给语法分析器，继续识别空白符后面的记号，然后把它返回给语法分析器。

我们的目标是构造一个词法分析器，这个词法分析器能利用图3-10给出的翻译表在输入缓冲区中识别出下一个记号的词素，产生该词素

正规表达式	记 号	属 性 值
ws	-	-
if	if	-
then	then	-
else	else	-
id	id	指向符号表表项的指针
num	num	指向符号表表项的指针
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

图3-10 记号的正规表达式模式

相应的记号和属性值的二元组。关系操作符的属性值由符号常量 LT、LE、EQ、NE、GT 和 GE 给出。 □

3.4.1 状态转换图

99 作为构造词法分析器的中间步骤，我们先来构造*状态转换图*（transition diagram）。状态转换图描绘语法分析器为得到下一个记号而调用词法分析器时词法分析器要做的动作，如图3-1所示。假设输入缓冲区如图3-3所示，并且词素开始（*lexeme-beginning*）指针指向上次发现的词素后面的字符。当向前指针扫描输入流字符时，我们用状态转换图来记录所读信息的轨迹，方法是在读字符的过程中我们不断地在状态转换图的各位置之间移动。

状态转换图的位置用圆圈表示，叫做*状态*。状态间由箭头连接，称为*边*。由状态 s 到状态 r 的边上标记的字符表示使状态 s 转换到状态 r 的输入字符。标记 **other** 表示任意一个未被离开状态s的边所标定字符。

本节假定状态转换图是*确定的*，即没有一个符号可以同时与离开一个状态的两条以上的边的标记匹配。3.5节将放宽这个条件，使词法分析器的设计更加简单。如果使用恰当的工具，词法分析器的实现会更容易。

状态转换图中具有一个状态标记为 start 状态，这个状态称为*初始状态*。识别记号时，我们将从这个状态开始。有些状态可以具有动作，当控制流到达一个具有动作的状态时，我们将执行这些动作。当进入一个状态时，我们需要读下一个输入字符。若存在一个离开当前状态的边，其标记和读入字符匹配，控制就转到由这条边指向的状态，否则表示失败。

图3-11是模式 > = 和 > 的状态转换图。它的开始状态是状态0。在状态0读下一个字符，如果该字符是 >，则转向状态6，否则便告识别 > 或 > = 失败。

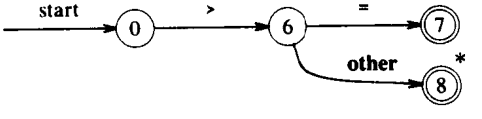


图3-11 > = 和 > 的状态转换图

到达状态6时，读下一个字符，如果它是 = ，则转向状态7，否则标有 **other** 的边表明已经转向状态8。在状态7上有双圈，表示它是接受状态。当进入这个状态时，状态转换图识别记号了 > =。

请注意，如果按照从 start 状态到达接受状态8的边的顺序，则意味着 > 和一个与之无关的
100 字符已经被读过。由于这个无关字符不是关系操作符 > 的一部分，而是下一个词素的一部分，所以向前指针必须回退一个字符。状态上的 * 表示向前指针必须回退一个字符。

通常可能有多个状态转换图，每个图说明一组记号。如果沿着一个状态转换图识别输入字符串时失败，我们需要把向前指针回退到进入该图开始状态时该指针所指向的输入字符串位置，并启动下一个状态转换图。因为在状态转换图的开始状态，词法分析器的词素*开始指针*和*向前指针*都指向同一个位置，所以向前指针被回退到词素开始指针所指向的位置。如果我们在所有状态转换图上都失败了，则意味着输入字符串有词法错误。这时，我们需要调用错误恢复程序进行错误处理。

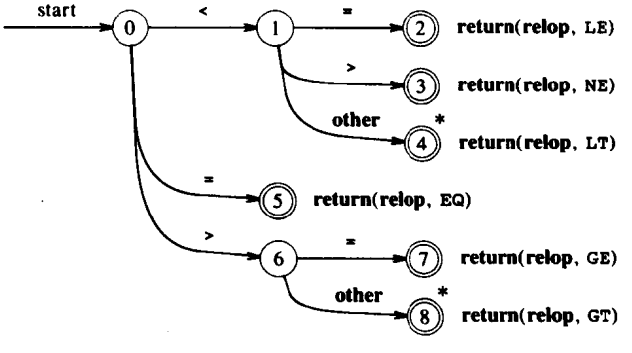


图3-12 关系操作符的状态转换图

例3.7 图3-12给出了记号 **relop** 的

状态转换图。注意，图3-11中的状态转换图只是这个更复杂的状态转换图的一部分。 □

例3.8 因为关键字是字母序列，所以它们也符合标识符的规则，即由字母开头的字母和数字的序列。一般来说，我们不为关键字单独构造状态转换图，而是把关键字看成特殊的标识符如在2.7节所述。当到达图3-13的接受状态时，执行一段代码，以确定这次识别的词素是关键字还是标识符。

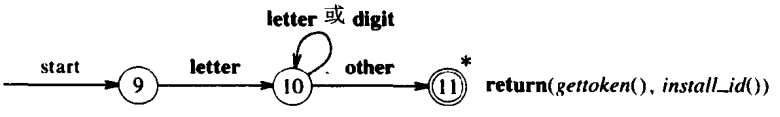


图3-13 标识符和关键字的状态转换图 101

把关键字从标识符中分离出来的一种简单技术是适当地初始化符号表（符号表中保存与标识符有关的信息）。对于图3-10中的记号，我们需要在开始扫描输入字符之前把字符串if、then和else填入符号表。这些符号的记号也将被记录在符号表中，以便它们在输入字符串中被识别出来时，返回它们的记号。图3-13中接受状态旁边的return语句分别使用*gettoken* ( )和 *install_id* ( )来获得要返回的记号和属性值。过程 *install_id* ( )访问缓冲区，标识符词素被定位在其中，并用该词素查符号表，如果在符号表中找到了该词素，当它被标记为关键字时，*install_id* ( )返回0，当它是程序变量时，*install_id* ( )返回指向相应符号表表项的指针。如果在符号表中没有找到该词素，则把该词素作为变量填入符号表中，并返回指向新建表项的指针。

过程 *gettoken* ( )也以类似的方式在符号表中查找词素。如果该词素是个关键字，则返回相应的记号，否则返回记号 **id**。

如果有要增加的关键字，无需修改状态转换图，只需将新增关键字对应的字符串和记号填入符号表即可。 □

如果手工编写词法分析器，把关键字先放进符号表的技术是很重要的。如果不这样做，一个典型的程序设计语言的词法分析器的状态数会达到几百个。如果使用这种技术，需要的状态数可能不到一百个。

例3.9 当我们为如下正规定义所给定的无符号数构造识别器时可能会发现很多问题：

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

注意，这个定义是 **digits fraction? exponent?** 的形式，后两部分是可选的。

一个给定记号的词素必须是最长的。例如，当输入串是12.3E4时，词法分析器不应该在发现12或12.3后就停止。在图3-14中，起始状态25、20、12在读入12、12.3、12.3E4后分别到达接受状态，这里假设12.3E4后面是一个非数字的字符。以25、20、12为开始状态的状态转换图分别用来识别 **digits**、**digits fraction** 和 **digits fraction? exponent**。因此，我们选择起始状态的顺序应该是12、20和25。

当到达接受状态19、24或27后，调用过程 *install_num*，将词素插入到存放数的表中，并返回指向新建表项的指针。词法分析器返回这个指针（作为词素的值）以及记号 **num**。 □

有关不符合记号正规定义语言的信息可以用来指出输入的错误。例如，当输入为1.<x时，我们将在状态14和22（图3-14）处遇到输入字符< 时失败。我们希望能够指出这个错误并继续执行，就好像输入是1.0<x一样，而不是返回1。这些想法也可以用来简化状态转换图，因为

错误处理可以使某些导致失败的错误得到恢复。

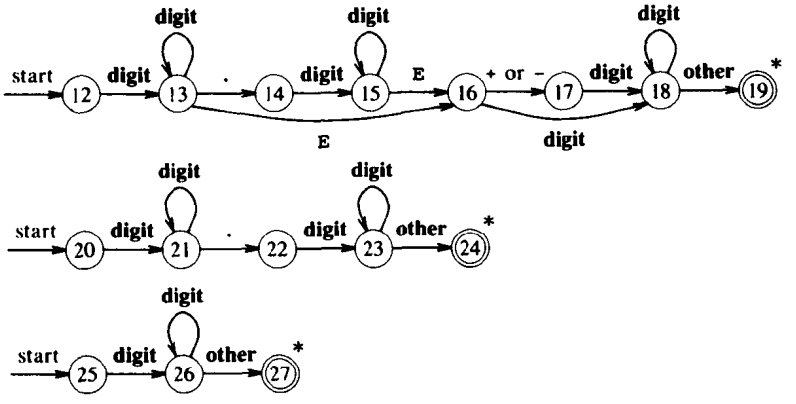
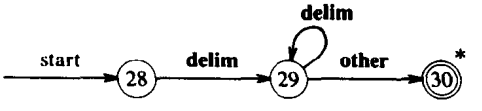


图3-14 Pascal中无符号数的状态转换图

我们可以使用多种方法避免图3-14中的多余匹配。一种方法是将这些状态转换图合并成一张图，一般来说这个任务比较艰巨。另一种方法是改变对失败的响应策略。本章后面将介绍一种方法，这种方法使我们能越过几个接受状态，当失败时，再恢复到越过的最后一个接受状态。

例3.10 把图3-12、图3-13和图3-14的状态转换图放在一起，我们便得到了例3.6中所有记号的转换图序列。进行匹配时，首先尝试编号小的开始状态，然后尝试编号大的开始状态。

接下来是与空白符处理问题有关的问题。对代表空白符的 **ws** 的处理方式与前面讨论的其他模式的处理方式有所不同，因为当在输入串中发现空白符时，没有任何信息返回给语法分析器。识别 **ws** 的状态转换图如下图所示。在到达接受状态时，什么也不返回，只是回到第一个状态转换图的开始状态，寻找其他模式。



102
~
103

只要有可能，先寻找出现频率较高的记号比较有利，因为只有前面所有的状态转换图都匹配失败以后，才会到达下一个状态转换图。空白符肯定是经常出现的，所以把识别空白符的状态转换图放在较前面将有利于提高效率。 □

3.4.2 状态转换图的实现

状态转换图序列可以变换成程序，用来识别该序列所定义的记号。我们将采用对所有状态转换图都适用的系统化方法来构造程序，该程序的大小与图中状态数和边数成正比。

每个状态对应一个代码段。如果一个状态具有出边，该状态的代码便读一个字符并选择应跟随的边。函数`nextchar()`用来从输入缓冲区中读入下一个字符，每次调用都向前移动向前指针，并返回读入的字符。[⊖]如果存在标记为该字符的边，或标记为包含该字符的字符类的边，则控制转给这条边指向的状态所对应的代码。如果不存在这样的边，而且当前状态不是接受状态，调用`fail()`程序，把向前指针撤回到开始指针指向的位置，启动下一个状态转换图对应的代码继续匹配。如果不存在下一个状态转换图，`fail()`调用错误恢复程序，进行错误处理。

我们用全局指针变量`lexical_value`返回记号。当识别出一个标识符或一个数时，`lexical_value`被赋值为`install_id()`和 `install_num()`过程返回的指针。记号类由

⊖ 一种更有效的实现方法是用嵌入式的宏来代替函数`nextchar()`。

词法分析器的主过程nexttoken()返回。

我们使用case语句来查找下一个状态转换图的开始状态。在图3-15的C语言实现中，两个变量state和start分别用来保存当前状态转换图的当前状态和起始状态。这段代码中的状态号就是图3-12至图3-14中的状态号。

如图3-16所示，在状态转换图中一条边一条边地往下匹配的过程是通过不断地选择一个状态对应的代码段来执行，以确定出下一个状态，并将控制转到该状态对应的代码段去执行。图3-16给出了状态0对应的代码（在例3.10进行修改以处理空白符）以及图3-13和图3-14中的状态转换图对

```
int state = 0, start = 0;
int lexical_value;
    /* 返回记号的第二个分量 */
int fail()
{
    forward = token_beginning;
    switch (start) {
       case 0:   start = 9; break;
       case 9:   start = 12; break;
       case 12:  start = 20; break;
       case 20:  start = 25; break;
       case 25:  recover(); break;
       default:   /* 编译错误 */
    }
    return start;
}
```

图3-15 找出下一个开始状态的C代码

```
token nexttoken()
{   while(1) {
        switch (state) {
        case 0:   c = nextchar();
            /* c是超前扫描字符 */
            if (c==blank || c==tab || c==newline) {
                state = 0;
                lexeme_beginning++;
                    /* 词素开始指针的前移 */
            }
            else if (c == '<') state = 1;
            else if (c == '=') state = 5;
            else if (c == '>') state = 6;
            else state = fail();
            break;
            .../* 这里放cases 1-8 */
        case 9:   c = nextchar();
            if (isletter(c)) state = 10;
            else state = fail();
            break;
        case 10:  c = nextchar();
            if (isletter(c)) state = 10;
            else if (isdigit(c)) state = 10;
            else state = 11;
            break;
        case 11:  retract(1); install_id();
            return ( gettoken() );
            .../* 这里放cases 12-24 */
        case 25:  c = nextchar();
            if  (isdigit(c)) state = 26;
            else state = fail();
            break;
        case 26:  c = nextchar();
            if (isdigit(c)) state = 26;
```

图3-16 词法分析器的C代码

```
            else state = 27;
            break;
        case 27:  retract(1); install_num();
            return ( NUM );
        }
    }
}
```

图3-16 （续）

应的代码。请注意，C代码 `while(1)` *stmt* 会不断地重复执行 *stmt*，直到遇见一个`return`为止。

由于C语言不允许同时返回记号和属性值，所以 `install_id()` 和 `install_num()` 用全局变量来存放对应于 **id** 和 **num** 表项的属性值。

如果实现状态转换图的语言没有 case 语句，可以为每个状态创建一个数组，用字符作下标。如果 *state*1 是这样的数组，则当超前扫描的字符是 *c* 时，*state*1[*c*] 是指向需要执行的程序段的指针。这些代码段一般以转到下一个状态的代码段的goto语句结束。状态 *s* 的数组可以看成是 *s* 的间接状态转换表。

3.5 词法分析器描述语言

目前有很多基于正规表达式从特定表示法构建词法分析器的工具。前面我们已经看到怎样使用正规表达式来描述记号模式。在考虑把正规表达式转换成模式匹配程序的算法之前，我们先给出一个使用这类算法的工具示例。

本节将介绍一个叫做 Lex 的工具。Lex 已经广泛地应用于各种语言的词法分析器的描述。我们称这种工具为 Lex 编译器，而且 Lex 编译器的输入称为 Lex 语言。讨论现有的工具的目的在于说明如何把正规表达式描述的模式与行为（如在符号表中创建新表项，这是词法分析器需要做的动作）结合起来。我们将使用类 Lex 语言来说明词法分析器，虽然这种词法分析器说明不能用现有的 Lex 编译器编译；我们可以使用上节介绍的状态转换图技术将这个说明转成可以运行的程序。

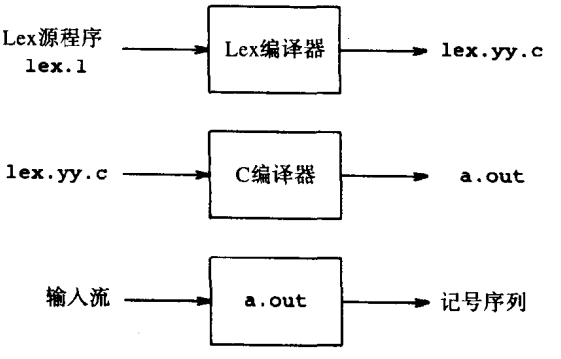


图3-17 用 Lex 建立一个词法分析器

Lex 的使用方法通常如图3-17所示。首先，使用 Lex 语言写一个定义词法分析器的源程序 `lex.l`。然后，利用 Lex 编译器将 `lex.l` 转换成 C 语言程序 `lex.yy.c`。它包括从`lex.l` 的正规表达式构造的状态转换图的表格形式以及使用该表格识别词素的标准子程序。与 `lex.l` 中正规表达式相关联的动作是 C 代码段，这些动作可以直接加到 `lex.yy.c` 中。最后，`lex.yy.c` 通过 C 编译器生成目标程序 `a.out`，`a.out` 就是把输入流转换成记号序列的词法分析器。

3.5.1 Lex 说明

一个Lex程序由如下三部分组成：

声明部分

%%

```
转换规则
%%
辅助过程
```

声明部分包括变量声明、符号常量声明和正规定义。(符号常量是被声明来表示常数的标识符。) 104~107
正规定义类似3.3节的正规定义，作为转换规则中正规表达式的部分。

Lex 程序的转换规则是如下形式的语句：

$$\begin{array}{ll} p_1 & \{ action_1 \} \\ p_2 & \{ action_2 \} \\ \cdots & \cdots \\ p_n & \{ action_n \} \end{array}$$

其中每个 p_i 是一个正规表达式，每个 $action_i$ 表示当模式 p_i 匹配上一个词素后词法分析器所要执行的程序段。在 Lex 中，这些 *action* 是用 C 语言编写的，当然也可以用其他语言来实现。

Lex 程序的第三部分包含 *action* 所需要的辅助过程。这些过程可以单独编译，并与词法分析器一起装载。

由Lex创建的词法分析器与语法分析器协同工作的方式如下：词法分析器被语法分析器调用后，从尚未扫描的输入字符串中读字符，每次读入一个字符，直到发现能与某个正规表达式 p_i 匹配的最长前缀。然后，词法分析器执行 $action_i$。通常 $action_i$ 会将控制返回给语法分析器。然而，如果不将控制交给语法分析器，词法分析器可以继续发现更多的词素，直到某个操作将控制返回给语法分析器。词法分析器的这种不断查找词素，直到以显式的 return 调用结束工作的方式，使其可以方便地处理空白符和注释。

词法分析器只返回记号给语法分析器，带有与词素相关信息的属性值是通过全局变量 `yylval`传递的。

例3.11 图3-18是识别图3-10中记号的 Lex 程序，这个程序返回识别的记号。我们通过观察这段代码可以发现很多 Lex 的特点。

在声明部分，我们可以看到转换规则所使用的符号常量的声明[⊖]。这些声明被一对特殊括号 **%{**和**%}**括在一起。所有出现在括号内的内容都直接复制到词法分析器 `lex.yy.c` 中。它们不作为正规定义或转换规则的一部分。对第三部分的辅助过程也进行同样的处理。在图3-18中有两个过程 `install_id` 和 `install_num`，它们被照原样复制到 `lex.yy.c` 中，这两个过程将由转换规则调用。

在声明部分还包含一些正规定义。每个定义由一个名字和这个名字所代表的正规表达式组成。例如，第一个名字定义为 `delim`，它代表字符类 `[\t\n]`，即空格、制表符（由`\t`表示）、换行符（由`\n`表示）三者之一。第二个是关于空白符定义，由名字ws表示。空白符是一个或多个分隔符组成的序列。注意，在 Lex 中词 `delim` 必须由大括号括起来以便与包含 `delim` 这五个字符的模式区别开。 108

在 `letter` 的正规定义中使用了字符类。`[A-Za-z]`表示大写字母A到Z或小写字母a到z中的任何一个。在 `id` 的定义（第五个定义）中使用了圆括号，圆括号是 Lex 语言的元符号，与通常情况下的含义相同，表示包括。类似地，竖线也是 Lex 语言的元符号，表示并。

⊖ 通常 `lex.yy.c` 作为由 Yacc 生成的语法分析器的子程序，Yacc 是将在第4章中讨论的语法分析生成器，这里，符号常量可以由语法分析器定义，在 `lex.yy.c` 中一同编译。

在 number 的正规定义中可以看到更多的细节。?是元符号，表示出现过0次或一次。反斜杠被当成转义字符，使得 Lex 的元符号能表示它的本来意义。在 number 的定义中，小数点表示成 "\."，因为在 Lex 和很多 UNIX 系统的处理正规表达式的程序中，单独的一个点表示除了换行符以外的所有字符的字符类。在字符类 [+\-] 中，减号前面的反斜杠是为了避免与减号表示范围的用法混淆（如 [A-Z] ）。[⊖]

```
%{
    /* 符号常量定义
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* 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}        {/* 没有动作和返回值 */ }
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = install_id(); return(ID);}
{number}    {yylval = install_num(); 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);}

%%

install_id() {
    /* 往符号表中填入词素的过程。yytext 指向词素的第一个字符，yyleng表示词素的长度。将词
    素填入符号表，返回指向该词素所在表项的指针 */
}

install_num() {
    /* 与填词素的过程类似，只不过词素是一个数 */
}
```

图3-18 关于图3-10中记号的 Lex 程序

还有一种方法能使字符保持本来的意义，即使它们是 Lex 的元符号。这种方法就是用引号把字符括起来。在转换规则部分中，我们使用了这种方法来表示六个关系操作符。[⊖]

现在，让我们考虑跟在第一个 %% 后面的转换规则。第一条规则表示如果发现 ws（任何由

⊖ 事实上，Lex不用反斜杠就可以正确地处理字符集[+-]，因为出现在最后的减号不可能表示范围。

⊖ 这样做是因为 < 和 > 是 Lex 的元符号；用来将状态名括起来，使得Lex在遇到特定记号如注释或引用的字符串时改变状态，保证它们与正常的文本的处理方式不同。没有必要将等号也放入引号中，但也不反对这么做。

空格、制表符和换行符组成的最长序列）则不做任何动作，控制也不返回给语法分析器，词法分析器继续识别记号，直到与某一个记号关联的动作调用了return语句。

第二条规则表示如果识别出 if，则返回记号 IF，它是表示某个整数的符号常量，语法分析器将这个整数理解为记号 **if**。类似地，接下来的两条规则用来识别关键字 then 和 else。

在 **id** 的规则中，关联的动作有两条语句。第一条语句将过程 install_id 的返回值赋给变量 yylval，该过程的定义在第三部分给出。变量 yylval 是在 Lex 的输出 lex.yy.c中定义的。语法分析器也可以访问这个变量。使用 yylval 的目的是保存词素的属性值，因为 return（ID）语句（即第二个语句）只能返回记号类。 109~110

我们在这里没有给出 install_id 的详细代码，但我们可以设想它的工作方式是在符号表中查找与模式 **id** 匹配的词素。Lex 使用两个变量 yytext 和 yyleng 来保证第三部分的程序能够访问匹配的词素。变量 yytext 就是前面介绍的词素 *lexeme_beginning*（开始指针），即指向词素第一个字符位置的指针。变量 yyleng 存放词素的长度。如果 install_id 在符号表中没有找到这个词素，则为它创建一个新的表项，输入流中从 yytext 开始的 yyleng 个字符被复制到一个字符数组中，并以一个字符串结尾符（2.7节）做结束标记，在符号表的新表项中添入一个指向这个字符串起始位置的指针。

接下来的一条规则以类似的方式处理数。在最后六条规则中，yylval 用来返回识别出的关系操作符对应的代码，而实际上对这六个关系操作符返回的都是记号 **relop** 的代码。

假设由图3-18中的程序生成的词法分析器被给定一个由两个制表符、一个 if 和一个空格组成的输入串。两个制表符是能与模式ws匹配的初始最长前缀。与 ws 相关联的动作不做任何事，因此词法分析器移动词素的开始指针（yytext）使其指向 i，并开始查找下一个记号。

下一个匹配的词素是 if。请注意，模式 if 和{id}均匹配这个词素，并且没有能匹配更长串的模式。由于在图3-18中匹配关键字if的模式先于匹配标识符的模式执行，所以 if 被匹配为关键字。通常，采用将匹配关键字的模式置于匹配标识符的模式之前的策略，可以简单有效地保留关键字。

再举一个例子，假设读入的头两个字符是 <=。模式<匹配上第一个字符，但它不是能匹配输入字符串的最长前缀的模式。Lex 采用“选择最长匹配前缀的策略”方便地解决了 < 和 <= 之间的冲突。这里，当然 <= 被选择作为下一个记号。 □

3.5.2 超前扫描操作

如3.1节所述，对于某些程序设计语言结构，词法分析器需要超前扫描词素后面的若干字符来确定一个记号。回忆我们前面提到的 Fortran 语句例子：

```
DO 5 I = 1.25
DO 5 I = 1,25
```

在 Fortran 中，除了在注释和 Hollerith 串中之外，空格不代表任何意义。我们可以认为在词法分析器开始工作时，所有的空格都已经去掉。这样，上面的两个语句就变为如下形式： 111

```
DO5I=1.25
DO5I=1,25
```

在第一个语句中，直到词法分析器发现了小数点才可以断定 DO 是标识符 DO5I 的一部分。在第二个语句中，DO 是关键字。

在 Lex 中我们可以把模式写成 r_1/r_2 的形式，其中，r_1 和 r_2 都是正规表达式。它的意思

是当一个字符串与 r_1 匹配时，还需其后的字符串与 r_2 匹配，这样才算该字符串与 r_1 匹配成功。在超前扫描操作符 / 后面的正规表达式 r_2 表示需要进一步匹配的内容，这里它只是匹配模式的一个限制，而不是匹配的一部分。例如，将上述语句中的 DO 识别为关键字的Lex说明如下：

```
DO/({letter} | {digit})* = ({letter} | {digit})*,
```

根据这个说明，词法分析器在输入缓冲区超前地扫描一串字母或数字，接着扫描等号以及后面的一串字母或数字，最后扫描到逗号才能够判断出这不是一个赋值语句。但只有超前扫描符前面的 D 和 O 才是与模式匹配的词素的部分。经过成功的匹配，yytext 指针指向字符 D 并且 yyleng = 2。注意，这个简单的超前扫描模式使得当 DO 后面跟着一些无意义的符号（如Z4 = 6Q）时也会识别出 DO，但它决不会把做为标识符一部分的 DO 识别为一个词素。

例3.12 超前扫描操作符还可以用来解决 Fortran 词法分析中的另一个难题：区别关键字和标识符。例如，

```
IF(I, J) = 3
```

是一个正确的赋值语句，而不是一个逻辑判断 if 语句。使用 Lex 描述关键字 IF 的一种方法是使用超前扫描操作符定义 IF 右边的正文。逻辑 if 语句的一种简单形式是

```
IF ( 条件 ) 语句
```

Fortran 77中介绍了if语句的另一种表示形式：

```
IF ( 条件 ) THEN
     then块
ELSE
     else块
END IF
```

每个无标号Fortran语句都以一个字母开始，并且每个用来于下标或操作数组的右括号都跟着操
112 作符，如 = 、+、逗号、另外一个右括号或者语句结尾。这样的右括号后面不能跟随字母。基于这种情况，为了确定IF是关键字而不是数组名，我们还需要向前扫描，寻找在换行符之前出现的由一个字母跟随的右括号（我们假定连续的卡片“删除”了换行符）。识别关键字IF的模式可以写为

```
IF / \( .* \) {letter}
```

其中的圆点表示除了换行符以外的任何字符，而括号前面的反斜杠表示括号取其本来的意思，而不是正规表达式中的元符号（见练习3.10）。 □

处理 Fortran 的 if 语句问题的另外一种方法是：当看到字符串 IF（后，先确定 IF 是否被声明为数组。如果是，我们才去匹配上面给出的整个模式。这样的检查使得由 Lex 说明自动实现一个词法分析器变得很难，而且它们在运行时可能耗费更多的时间，因为要由模拟状态转化图的程序频繁地判断是否要进行这样的检查。应该说明的是，对 Fortran 程序进行词法分析是很不规则的任务。使用特殊的程序设计语言直接编写 Fortran 词法分析器比使用自动生成器来生成词法分析器更容易。

3.6 有穷自动机

语言的识别器是一个程序，它以字符串 x 作为输入，当x是语言的句子时，回答“是”，否

则回答“不是”。我们可以通过构造有穷自动机把正规表达式编译成识别器。有穷自动机是更一般化的状态转换图，它可以是确定的或不确定的，其中“不确定”的含义是：对于某个输入符号，在同一个状态上存在不止一种转换。

确定和不确定的有穷自动机都能而且仅能识别正规集，即它们能够识别正规表达式所表示的语言。但是，它们之间有着时空的权衡。确定的有穷自动机导出的识别器比不确定的有穷自动机导出的识别器快得多，但确定的有穷自动机可能比与之等价的不确定的有穷自动机大得多。下节将给出把正规表达式变成两种有穷自动机的方法。由于变成不确定的自动机更直接一些，我们首先讨论这一类自动机。

本节和下节的基本例子都是由正规表达式 $(a|b)^*abb$ 表示的语言，即包括所有以 abb 结尾的a和b的符号串。类似的语言在实际中也会出现。例如，表示所有以 .o结尾的文件名的正规表达式是(.|o|c)*.o，其中c代表除.和o以外的任何字符。又如，C语言的注释是由开括号/* 之后以*/结尾的任意字符序列组成的，其任何真前缀都不以*/结尾。 113

3.6.1 不确定的有穷自动机

不确定的有穷自动机（简写为NFA）是一个由以下几部分组成的数学模型：

1. 一个状态的有穷集合 S。
2. 一个输入符号集合 Σ，即*输入符号字母表*。
3. 一个转换函数 *move*，它把由状态和符号组成的二元组映射到状态集合。
4. 状态s_0是惟一的*开始*或*初始*状态。
5. 状态集合 F 是*接受*（或*终止*）状态集合。

NFA可以用带标记的有向图表示，称为*转换图*（transition graph），其节点是状态，有标记的边表示转换函数。这种转换图和前面所讲的状态转换图（transition diagram）很类似，但略有区别：同一个字符可以标记始于同一个状态的两个或多个转换，边可以由输入字符符号，也可以由特殊符号 ϵ 标记。

图3-19给出了识别语言（$a|b)^*abb$的NFA的转换图。这个 NFA 的状态集合是{0, 1, 2, 3}，输入符号表是{a, b}，状态0是开始状态，状态3是接受状态，用双圈表示。

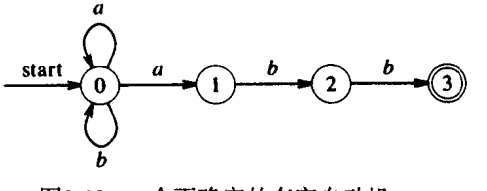


图3-19 一个不确定的有穷自动机

描述NFA时，我们常用这种转换图表示。正如我们将要看到的那样，我们可以在计算机上使用不同的方法实现 NFA 的转换函数。最简单的办法是使用*转换表*。转换表的每个状态占一行，每个输入符号占一列。如果必要，符号 ϵ 也占一列。表中第 i 行 a 列对应的表项是当输入为 a 时从状态 i 所能到达的状态的集合（实际上它很可能是指向状态集合的指针）。图3-20是与图3-19的 NFA 对应的转换表。

状 态	输入符号	
	a	b
0	{0, 1}	{0}
1	–	{2}
2	–	{3}

图3-20 与图3-19的NFA对应的转换表

转换表表示的优点是能够快速地确定给定状态在给定字符上的转换。它的缺点是：当输入字母表较大而且大多数转换是空集时，需要耗费大量空间。转换函数的邻接表表示法能提供较紧凑的实现，但在确定一个给定的转换时速度较慢。显然，我们很容易就能把有穷自动机的一种实现转变成另一种。 114

当且仅当对应的转换图中存在从开始状态到某个接受状态的路径，使得该路径的边上的标记恰好连成字符串 x 时，一个 NFA 接受输入字符串 x。图3-19的NFA可以接受输入串*abb*，*aabb*，*babb*，*aaabb*，…。例如，从状态0开始，沿着标记为 a 的边再回到状态0，然后沿着标记分别为 a、b、b 的边进入状态1、2、3。因为3是接受状态，所以*aabb* 被接受。

一条路径可以用状态转换序列表示，其中的状态转换叫做移动。下图是接收输入字符串*aabb*过程中的所有移动：

$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

一般说来，可能有多个移动序列可以到达接受状态。注意，对于输入串*aabb*，也许还可以沿着一些其他的移动序列走下去，但它们都不会终止在接受状态。例如，对于输入串*aabb*，下述移动序列会停留在非接受状态0上：

$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$

由NFA定义的语言是它接受的输入字符串的集合。不难看出图3-19的NFA接受的语言是 $(a|b)^*abb$。

例3.13 图3-21是接受 $aa^* \mid bb^*$ 的NFA。字符串 *aaa* 经过状态0、1、2、2、2的路径被接受。这些边上的标记分别为ϵ，*a*，*a* 和 *a*，连接成 *aaa*。ϵ 在连接中“消失”。 □

3.6.2 确定的有穷自动机

确定的有穷自动机（简称 DFA）是不确定的有穷自动机的特例，其中：

1. 没有一个状态具有 ϵ 转换，即在输入 ϵ 上的转换。

2. 对每个状态*s*和输入符号 *a*，最多只有一条标记为 *a* 的边离开 *s*。

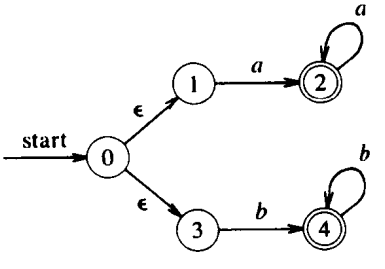


图3-21 接受$aa^*|bb^*$的NFA

确定的有穷自动机在任何状态下，对任一输入符号，最多只有一个转换。如果用转换表表示 DFA 的转换函数，那么表中的每个表项最多只有一个状态。因而，很容易确定 DFA 是否接受某输入字符串，因为从开始状态起，最多只有一条到达接受状态的路径可由这个符号串标记。下边的算法说明怎样在一个输入串上模拟 DFA 的行为。

算法3.1 模拟DFA。

输入：输入以文件结束符 **eof** 结尾的串 x；一个DFA D，其开始状态为 s_0，接受状态集合为 F。

输出：如果 D 接受 x，则回答“yes”，否则回答“no”。

方法：把图3-22的算法应用于输入字符串 x。函数 *move* (s, c) 给出在状态 s 上遇到输入字符 c 时应该转换到的下一个状态。函数 *nextchar* 返回输入串 x 的下一个字符。

```
s := s0;
c := nextchar;
while c ≠ eof do
        s := move(s, c);
        c := nextchar
end;
if s is in F then
        return "yes"
else return "no";
```
图3-22 模拟DFA

□

115 ~ 116

例3.14 图3-23是与图3-19的NFA接受同一语言 $(a|b)^*abb$ 的 DFA 的转换图。对这个DFA和输入串 $ababb$，算法3.1沿着状态序列0、1、2、1、2、3移动，并返回“yes”。 □

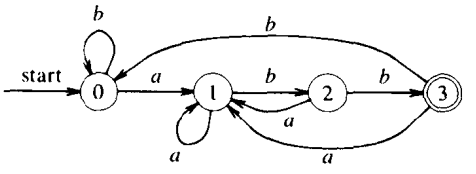


图3-23 接受$(a|b)^*abb$的DFA

3.6.3 从NFA到DFA的变换

图3-19的NFA在状态0对输入a有两个转换，即可能进入状态0或状态1。类似地，图3-21的NFA在状态0对 ϵ 也有两个转换。虽然未给出例子，但是我们不难想像当在某一个状态上既可以根据 ϵ 也可以根据一个实际的输入符号进行转换时，会引起歧义。这种多值转换函数使得我们很难用计算机程序模拟 NFA。“接受”的定义仅仅是说必然存在一条从开始状态到某个接受状态的路径，该路径的标记是输入字符串。如果有很多路径其边上的标记都可以连成同样的输入字符串，则在找到一条接受路径或发现没有路径可到达接受状态前，我们可能不得不考虑所有这些路径。

现在我们给出从 NFA 构造识别同样语言的 DFA 的算法。这个算法通常被称为*子集构造算法*，它有利于使用计算机程序模拟 NFA。一个和它紧密相关的算法在下一章构造 LR 语法分析器时将起到重要作用。

在 NFA 的转换表中，每个表项是一个状态集；而在 DFA 的转换表中，每个表项只有一个状态。从 NFA 变换到 DFA 的基本思想是让 DFA 的每个状态对应 NFA 的一个状态集。这个 DFA 用它的状态去记住 NFA 在读输入符号后到达的所有状态。也就是说，在读了输入 $a_1a_2\cdots a_n$ 后，DFA 到达一个代表 NFA 的状态子集 T 的状态。这个子集 T 是从 NFA 的开始状态沿着那些标有 $a_1a_2\cdots a_n$ 的路径能到达的所有状态的集合。DFA 的状态数有可能是 NFA 状态数的指数。但实际上，这种最坏的情况很少发生。 117

算法3.2 （*子集构造算法*）从 NFA 构造 DFA。

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 $Dtran$，DFA 的每个状态是 NFA 的状态集，D 将“并行”地模拟 N 对输入串的所有可能的移动。

我们用图3-24的操作来记录 NFA 的状态集的轨迹（s 代表 NFA 的状态，T 代表 NFA 的状态集）。

在读第一个输入符号前，N 可以处于集合 ϵ-$closure(s_0)$ 中的任何状态上，其中 s_0 是 N 的开始状态。假定从 s_0 出发经过输入字符串上的一系列移动，N 到达集合 T 中的状态。令 a 是下一个输入符号。遇到 a 时，N 可以移动到集合 $move(T, a)$ 中的任何状态。由于允许 ϵ 转换，遇到 a 以后，N 可以处于 ϵ-$closure(move(T, a))$ 中的任何状态。

操 作	描 述
ϵ-$closure(s)$	从NFA状态s只经过ϵ转换可以到达的NFA状态集
ϵ-$closure(T)$	从T中的状态只经过ϵ转换可以到达的NFA状态集
$move(T, a)$	从 T 中的状态 s 经过输入符号 a 上的转换可以到达的NFA状态集

图3-24 NFA状态上的操作

我们按下面的方法构造 D 的状态集合 *Dstates* 和 D 的转换表 *Dtran*。D 的每个状态对应于NFA的一个状态集，它是 N 读了某个输入符号序列后所能到达的全部状态，包括所有的 ϵ 转换。D的开始状态是 ϵ -*closure*(s_0)。使用图3-25的算法构造 D 的状态和转换。如果 D 的某个状态是至少包含一个 N 的接受状态的 NFA 状态集，那么它是 D 的一个接受状态。

ϵ -*closure* (T) 是一个典型的从给定节点集合出发在转换图上搜索可达节点集的过程。这里，T 的状态是给定的节点集合，转换图中只包含NFA中由 ϵ 标记的边。计算 ϵ -*closure* (T) 的简单算法是用栈来保存其边还没有完成 ϵ 转换检查的状态。图3-26给出了这样的过程。 □

```
初始时，ε-closure(s0)是Dstates中惟一的状态且未被标记;
while Dstates中存在一个未标记的状态T do begin
    标记T;
    for 每个输入符号a do begin
        U := ε-closure(move(T, a));
        if U 没在Dstates中 then
            将U 作为一个未标记的状态添加到 Dstates 中;
        Dtran [T, a] := U
    end
end
```

图3-25 子集构造法

```
将T中所有的状态压入栈 stack 中;
将ε-closure(T)初始化为 T;
while 栈 stack 不空 do begin
    将栈顶元素 t 弹出栈;
    for 每个这样的状态 u：从 t 到 u 有一条标记为ε的边 do
        if u 不在ε-closure(T)中 do begin
            将u添加到ε-closure(T);
            将u压入栈stack中
        end
end
```

图3-26 ϵ -*closure* 的计算

例3.15 图3-27给出了接受语言 (*a*|*b*)**abb* 的另一个 NFA N（它是下一节中从正规表达式开始一步一步地构造出来的NFA）。我们现在把算法3.2运用到 N。等价的 DFA 的开始状态是 ϵ -*closure*(0)，即 $A = \{0, 1, 2, 4, 7\}$，其中的每个状态都是从状态0出发经过每条边都由 ϵ 标记的路径能到达的状态。注意，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。

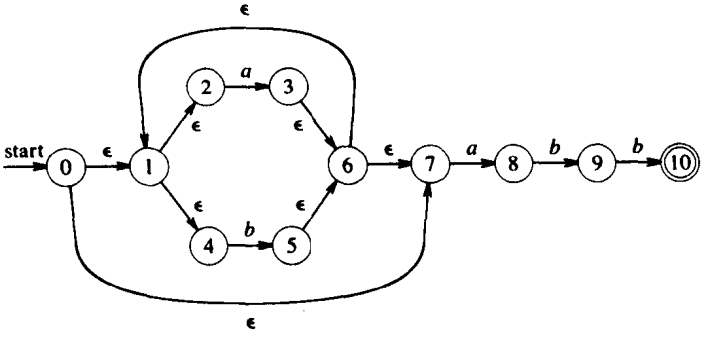


图3-27 (*a*|*b*)* *abb* 的 NFA N

这里的输入符号表是{*a*, *b*}。图3-25中给出的算法告诉我们要先标记A，然后计算 ϵ -*closure* (*move* (A, *a*))。让我们首先计算 *move* (A, *a*)，即对输入 *a* 从 A 状态可以转换到的 N 的状态集。在状态0，1，2，4和7中只有2和7有 *a* 上的转换，分别到达状态3和8，所以

$$\epsilon\text{-}closure(move(\{0, 1, 2, 4, 7\}, a)) = \epsilon\text{-}closure(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$$

让我们称这个集合为 B。于是，*Dtran* [A, *a*] = B。

在A中只有状态 4 对输入 *b* 有一个转换（转换到状态5），所以 DFA 对输入 *b* 有一个从 A

到C的转换，其中 $C = \epsilon\text{-}closure(\{5\}) = \{1, 2, 4, 5, 6, 7\}$。因此 $Dtran[A, b] = C$。

对新的没标记过的集合 B 和 C 继续这个过程，最终会使得所有的集合 (即 DFA 的状态)都已标记过。因为包含11个状态的集合其不同子集“只有” 2^{11} 个，而且一个集合一旦被标记就永远是标记的，所以这个过程肯定能终止。最终，实际构造出的5个不同的状态集合是：

$A = \{0, 1, 2, 4, 7\}$　　$D = \{1, 2, 4, 5, 6, 7, 9\}$
$B = \{1, 2, 3, 4, 6, 7, 8\}$　　$E = \{1, 2, 4, 5, 6, 7, 10\}$
$C = \{1, 2, 4, 5, 6, 7\}$

状态 A 是开始状态，状态 E 是惟一的接受状态，完整的转换表 $Dtran$ 如图3-28所示。

所得 DFA 的转换图如图3-29所示。注意，图3-23的 DFA 也接受 $(a|b)^*abb$，并且少一个状态。DFA 的状态数的最小化问题将在3.9节中讨论。 □

状 态	输入符号	
	a	*b*
A	*B*	*C*
B	*B*	*D*
C	*B*	*C*
D	*B*	*E*
E	*B*	*C*

图3-28 DFA的转换表*Dtran*

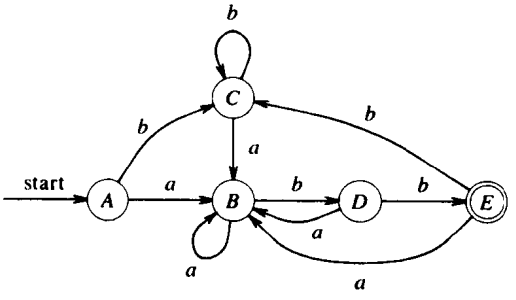


图3-29 对图3-27应用子集构造法得到的结果

3.7 从正规表达式到 NFA

有很多从正规表达式建立其识别器的策略，各有优劣。其中有一个策略常用于文本编辑程序，该策略先使用本节将要介绍的算法3.3从正规表达式构造 NFA，然后利用算法3.4模拟 NFA 在输入串上的行为。若想提高运行速度，我们可以利用上一节介绍的子集构造法把 NFA 变成 DFA。在3.9节，我们可以看到另一种直接由正规表达式构造 DFA 而无需建立过渡的 NFA 的方法。本节还将讨论基于 NFA 和 DFA 的识别器的实现在时间与空间复杂性的权衡问题。

3.7.1 从正规表达式构造 NFA

现在我们给出从正规表达式构造 NFA 的算法。这个算法有很多变形，本节给出一个易于实现的简单版本。这个算法是语法制导算法，该算法使用正规表达式的语法结构来制导构造过程。算法的分支遵循正规表达式定义的分支。我们首先构造自动机使其能够识别 ϵ 和字母表中任何符号，然后由此构造自动机来识别包含一个交换、一个连接或一个克林闭包运算符的正规表达式。例如，对于正规表达式 $r|s$，可从 r 和 s 的 NFA 构造出它的 NFA。

在构造过程中，每步最多引入两个新的状态，于是，为一个正规表达式构造的最终 NFA 的状态数最多两倍于该正规表达式中符号和操作符数。

118 ~ 121

算法3.3 （Thompson构造法）从正规表达式构造 NFA。

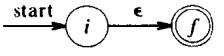
输入：字母表 Σ 上的一个正规表达式 r。

输出：接受 $L(r)$ 的 NFA N。

方法：首先，分析r并将其分解成最基本的子表达式，然后使用下面的规则1和规则2为 r 中的每个基本符号（ϵ 或字母表中的符号）构造 NFA。基本符号对应正规表达式定义的1和2两部分。请注意，如果符号 a 在 r 中出现多次，则要为它的每次出现构造一个 NFA。

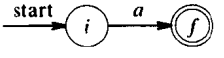
然后，由正规表达式 r 的语法结构制导，用下面的规则3逐步地组合前面构造的 NFA，直到获得整个正规表达式的 NFA 为止。在构造过程中所产生的中间 NFA（与 r 的子表达式对应）有几个重要的性质：只有一个终态；开始状态无入边，终态无出边。

1. 对 ϵ，构造NFA



其中，i 是新的开始状态，f 是新的接受状态。很明显这个 NFA 识别{ ϵ }。

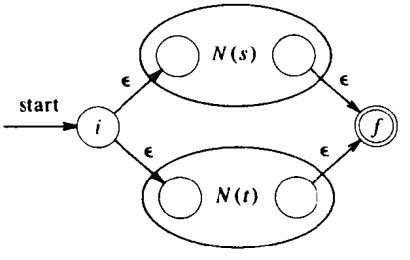
2. 对于 Σ 中的每个符号 a，构造 NFA



同样，i 是新的开始状态，f 是新的接受状态。这个 NFA 识别{ a }。

3. 如果 $N(s)$ 和 $N(t)$ 是正规表达式 s 和 t 的 NFA，则：

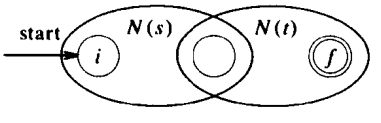
a) 对于正规表达式 $s \mid t$，可构造复合的 NFA $N(s \mid t)$如下：



122

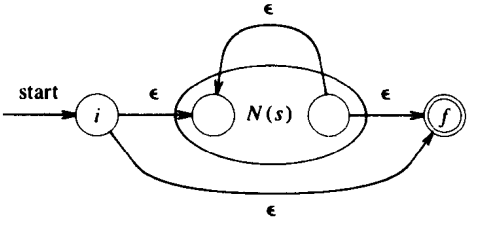
这里 i 是新的开始状态，f 是新的接受状态。从 i 到 $N(s)$ 和 $N(t)$ 的开始状态有 ϵ 转换，从 $N(s)$和 $N(t)$ 的接受状态到 f 也有 ϵ 转换。$N(s)$ 和 $N(t)$ 的开始和接受状态不是 $N(s \mid t)$ 的开始和接受状态。这样，从 i 到 f 的任何路径必须独立完整地通过 $N(s)$或 $N(t)$，因此这个复合的 NFA 识别 $L(s) \cup L(t)$。

b) 对于正规表达式 st，可构造复合的 NFA $N(st)$ 如下：



这里，$N(s)$ 的开始状态成为复合后的 NFA 的开始状态，$N(t)$ 的接受状态成为复合后的NFA的接受状态。$N(s)$ 的接受状态和 $N(t)$ 的开始状态合并，也就是说 $N(t)$ 的开始状态上的所有转换现在变成了 $N(s)$ 的接受状态上的转换。合并后的状态不作为复合后的NFA的接受或开始状态。从 i 到 f 的路径必须首先经过 $N(s)$，然后经过 $N(t)$，所以路径上的标记是 $L(s)L(t)$ 中的串。因为没有边进入 $N(t)$ 的开始状态或离开 $N(s)$ 的接受状态，所以从 i 到 f 的路径不能从 $N(t)$ 回到 $N(s)$，因此复合的NFA识别 $L(s)L(t)$。

c) 对于正规表达式 s^*，可构造复合的 NFA $N(s^*)$ 如下：



在此，i 和 f 分别是新的开始状态和接受状态。在这个复合的 NFA 中，可以沿着一条标记了 ϵ 边直接从 i 到达 f，这表示 ϵ 属于$(L(s))^*$。我们还可以从 i 经过一次或多次 $N(s)$到达 f。显然，这个复合的 NFA 识别 $(L(s))^*$。

d) 对于括起来的正规表达式 (s)，使用 $N(s)$ 本身作为它的 NFA。 123

在上述构造过程中，每次构造的新状态都要赋予不同的名字。这样，任何NFA的两个状态都具有不同名字。即使同一符号在r中出现多次，我们也要为该符号的每个实例创建一个独立的带有自己状态的 NFA。 □

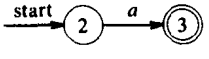
可以验证，算法3.3构造的每一步都产生识别对应语言的 NFA。此外，产生的 NFA 具有下列性质：

1. $N(r)$ 的状态数最多是 r 中符号和运算符个数的两倍。因为构造的每步最多引入两个新状态。

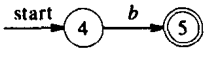
2. $N(r)$ 只有一个开始状态和一个接受状态，接受状态没有出边。作为构成成份的每个自动机也具有这一性质。

3. $N(r)$ 的每个状态或者有一个用 Σ 中的符号标记的出边，或者至多有两个标记为 ϵ 的出边。

例3.16 我们用算法3.3构造正规表达式 $r = (a|b)^*abb$ 的 $N(r)$。图3-30是 r 的分析树。这个分析树类似于2.2节中为算术表达式构造的分析树。对成份 r_1（即第一个 a），构造它的 NFA 如下：



对 r_2，构造它的NFA如下：



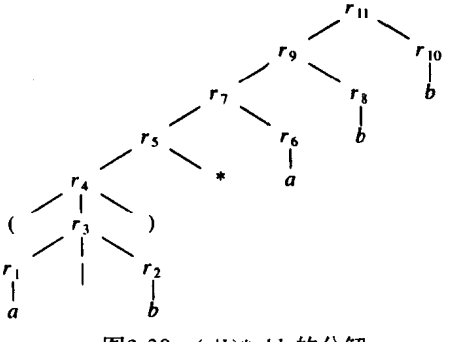
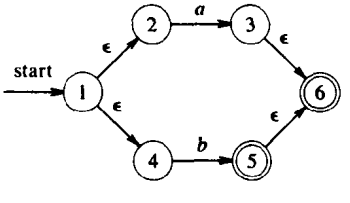
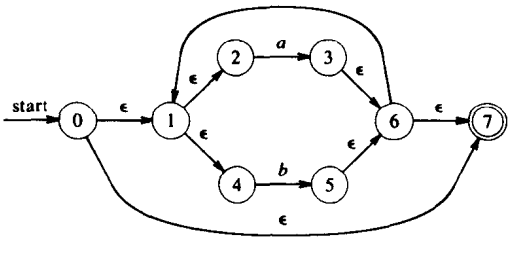


图3-30 $(a|b)^*abb$ 的分解 124

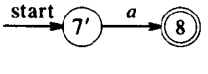
再用并规则组合 $N(r_1)$ 和 $N(r_2)$，得到 $r_3 = r_1|r_2$的NFA 如下：



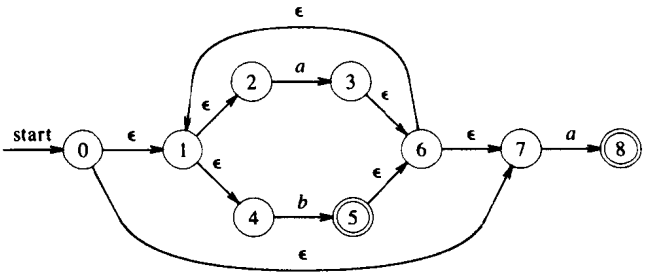
(r_3)的 NFA 和 r_3的一样。$(r_3)^*$ 的 NFA 构造如下：



$r_6 = a$的NFA如下：



我们合并状态7和7′（称其为状态7）得到 $r_5 r_6$ 的自动机，该自动机如下：



125 这样依次做下去，最后得到 $r_{11} = (a|b)^*abb$ 的 NFA，如图3-27所示。 □

3.7.2 NFA的双堆栈模拟

现在我们给出一个算法，对于给定的由算法3.3生成的 NFA N 和输入串 x，判断自动机 N 是否能够接受字符串 x。算法每次从输入字符串读取一个字符，然后计算自动机 N 在读入输入字符串的每个前缀后可能进入的所有状态的集合。这个算法利用由算法3.3生成的 NFA 的一些特殊性质，有效地计算非确定的状态集合。这个算法的运行时间与 $|N| \times |x|$ 成正比，其中，$|N|$ 表示N 的状态数，$|x|$ 表示串 x 的长度。

算法3.4 模拟 NFA。

输入：由算法3.3生成的 NFA N 和输入串 x。假定输入串 x 由字符 **eof** 做结束标记，N 以状态s_0 为开始状态，F 是接受状态集。

输出：如果 N 接收 x，则返回 “yes”，否则返回 “no”。

方法：把图3-31给出的算法应用到输入串x。这个算法在运行时执行了子集构造算法。它分两步计算从当前状态集到下一个状态集的转换。第一步，它先求 $move(S, a)$，即状态 S 在输入 a（当前输入字符）上经过一个转换能到达的所有状态的集合。第二步求出 $move(S, a)$ 在经过0个或多个 ϵ 转换后能到达的状态集。算法每次使用函数 *nextchar* 从输入字符串x读下一个字符。当 x 中的所有字符都读完后，如果接受状态在当前集合 S 中，则返回 “yes”，否则返回 “no”。 □

```
S := ϵ-closure({s0});
a := nextchar;
while a ≠ eof do begin
        S := ϵ-closure(move(S, a));
        a := nextchar
end
if S ∩ F ≠ ∅ then
        return "yes";
else return "no";
```

图3-31 模拟由算法3.1构造的NFA

算法3.4可以使用两个堆栈和由 NFA 状态做索引的位向量来有效地实现。一个堆栈用于跟踪非确定状态的当前集合的轨迹，另一个堆栈用于计算下一个非确定状态集。我们使用图3-26
126 所示的算法来计算 ϵ-*closure*。用位向量可以在在常数时间内判断一个非确定状态是否已在堆栈中，以防重复加入。一旦我们已经在第二个栈求出了下一个状态，则两个栈的角色互换。由于每一个非确定状态至多有两个输出边，因此每个状态在一个转换中至多会增加两个新状态。我们用 $|N|$ 表示 N 的状态数。因为一个栈中至多有 $|N|$ 个状态，所以计算当前状态集的下一个状态集的时间与 $|N|$ 成正比。因此，模拟 N 在输入字符串 x 上的行为需要的时间正比于 $|N| \times |x|$。

例3.17 N 是图3-27所示的NFA，x 是只有一个字符 a 的输入字符串。开始状态是 ϵ-*closure* ({0}) = {0, 1, 2, 4, 7}。当输入为 a 时，状态2转换到状态3，状态7转换到状态8，因此 T = {3, 8}。取 T 的 ϵ-*closure* 得到下一个状态{1, 2, 3, 4, 6, 7, 8}，其中的状态都不是接受状态，因此，算法返回 “no”。

注意，算法3.4是在运行时执行子集构造法的。例如，比较上述的转换和图3-29中由图3-27的 NFA 构造的 DFA 状态图。开始状态集和读 a 后可达到的状态集对应着 DFA 的 A 和 B 状态。 □

3.7.3 时间空间的权衡

给定一个正规表达式 r 和输入字符串 x，我们已经介绍了两种方法来确定 x 是否在 $L(r)$ 中。第一种方法是利用算法3.3为 r 构造一个 NFA N。这种构造法的时间复杂性是 $O(|r|)$，其中 $|r|$ 是 r 的长度。N 至多具有两倍于 $|r|$ 的状态，每一个状态至多有两个转换，因此 N 的转换表的空间复杂性是 $O(|r|)$。我们使用算法3.4判断 N 是否接受字符串 x，其时间复杂性是 $O(|r| \times |x|)$。我们使用这种方法判断 x 是否在 $L(r)$ 中的时间代价正比于 r 的长度和 x 长度的乘积。在很多文本编辑器中，当目标字符串 x 不是很长时，可以用这种方法寻找正规表达式模式。

第二种方法是先用 Thompson 构造法（算法3.3）从正规表达式 r 构造其 NFA，然后再用子集构造法（算法3.2）构造DFA（3.9节将介绍一种避免显式生成中间 NFA 的构造方法）。我们利用转换表实现状态转换函数，并使用算法3.1模拟 DFA 在输入串 x 上的动作。这个算法时间代价与 x 的长度成正比，但与 DFA 的状态数无关。这种方法经常用于在文本文件中寻找正规表达式模式的模式匹配程序。一旦有穷自动机创建成功，查找的速度将非常快。当目标串 x 非常长时，这种方法是很有利的。

然而，存在一些正规表达式，它们的最小DFA也有很多的状态，其状态数是正规表达式大小的指数。例如，若正规表达式 $(a|b)^*a(a|b)(a|b)\cdots(a|b)$ 包含 $n-1$ 个 $(a|b)$，则识别该正规表达式的任意 DFA 的状态数不可能少于 2^n。这个正规表达式表示 a 和 b 的字符串，这个字符串的倒数第 n 个字符是 a。不难证明这个正规表达式的任何一个 DFA 都必须记忆输入字符串的最后 n 个字符的轨迹，否则，它会给出错误的答案。显然，至少需要 2^n 个状态来记忆任何由 a 和 b 构成的长度为n的字符串的轨迹。虽然在一些应用中会出现类似的正规表达式，但幸运的是，在词法分析应用中，这种情况并不经常出现。 127

另一种方法是使用 DFA，但通过利用“惰性转换计算”技术来避免创建整个状态转换表。转换是在运行时计算的，只有在真正需要的时候才去计算给定状态在给定输入上的转换。计算的转换存储在 cache 中。每次要进行状态转换时，先检查 cache。如果需要的转换不在 cache 中，我们才去计算它，并将其存入 cache。如果 cache 满了，我们可以清除一些旧的转换，为新的转换腾出空间。

图3-32总结了用从 NFA 和 DFA 构造的识别器判断输入字符串x是否在由正规表达式 r 表示的语言中所需的最坏的时间、空间复杂性。“惰性”技术结合了 NFA 的空间需求小和 DFA 的时间需求小的特点。它的空间需求是正规表达式的大小加上 cache 的大小。它的运行时间几乎与 DFA 识别器相同。在某些应用中，“惰性”技术比 DFA 方法还要快，因为它没有计算不必要的状态转换。

自动机	空间	时间
NFA	$O(\|r\|)$	$O(\|r\| \times \|x\|)$
DFA	$O(2^{\|r\|})$	$O(\|x\|)$

图3-32 识别正规表达式所需的空间和时间

3.8 设计词法分析器的生成器

本节讨论从 Lex 语言程序自动生成词法分析器的软件工具的设计。尽管我们将给出多种方法，并且没有哪一种方法与 UNIX 系统的 Lex 命令所用方法完全相同，但我们仍称这些生成词法分析器的程序为 Lex 编译器。

假设词法分析器的说明形式如下： 128

p_1 $\{action_1\}$
p_2 $\{action_2\}$

. . . . . .

p_n $\{ action_n \}$

与3.5节相同，其中每个 p_i 都是正规表达式，每个 $action_i$ 是当输入字符串中的一个词素匹配 p_i 后要执行的程序段。

我们的问题是怎样构造一个在输入缓冲区中查找词素的识别器。如果有多个模式匹配成功，识别器将选择与最长词素匹配的模式。如果有多个模式与最长词素匹配，则选择第一个与最长词素匹配的模式。

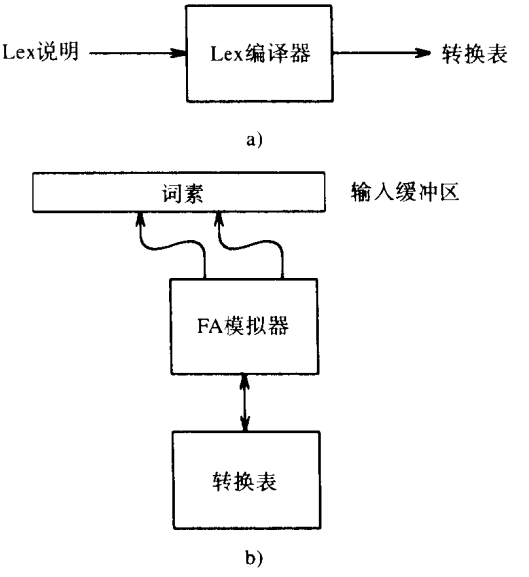


图3-33 Lex编译器模型

a) Lex编译器 b) 词法分析器示意图

有穷自动机是一种创建词法分析器的自然模型。由 Lex 编译器构造出的词法分析器具有图3-33b所示的结构。与3.2节介绍的一样，有一个输入缓冲区，缓冲区有两个指针，其中一个是词素的*开始指针*，另一个是词素的*向前指针*。Lex 编译器根据使用 Lex 说明书写的正规表达式模式为有穷自动机构造转换表。词法分析器本身包括一个有穷自动机模拟器，这个模拟器使用转换表在输入缓冲区中查找正规表达式模式。

129 本节的剩余部分将介绍基于 NFA 或 DFA 的 Lex 编译器的实现。在上一节的最后我们已经看到，由正规表达式生成的 NFA 的转换表远小于 DFA 的转换表，但进行模式匹配时DFA比NFA要快得多。

3.8.1 基于NFA的模式匹配

一种基于 NFA 的模式匹配方法是为模式 $p_1 \mid p_2 \mid \cdots \mid p_n$ 的 NFA N 构造状态转换表。为此，我们可以首先使用算法3.3为每个模式 p_i 构造 NFA $N(p_i)$，然后加入一个新的开始状态 s_0，并用 ϵ 转换将 s_0 和每个 $N(p_i)$ 的开始状态相连，如图3-34所示。

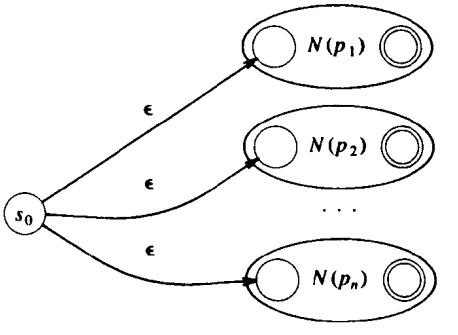


图3-34 根据Lex说明构造的NFA

我们可以把算法3.4稍做修改来模拟这个 NFA。修改必须保证该 NFA 能识别输入字符串中被匹配的最长前缀。在这个 NFA 中，每一个模式 p_i 有一个接受状态。当使用算法3.4模拟 NFA 时，我们构造一个状态集序列，其中每个状态集都是 NFA 看到每个输入字符后能够进入的状态集。即使在一个状态集中包括了一个接受状态，但为了实现最长的匹配，我们仍需继续模拟 NFA，穷尽当前输入符上的所有转换，即到达一个终止。

假设我们设计的 Lex 说明对任何一个有效的源程序来说，在 NFA 没有到达终止之前，这个源程序都不会完整地进入输入缓冲区。例如，每一种编译器都对标识符的长度有限制。当输入缓冲区溢出时，这种限制将被检测。

130 为了找到正确的匹配，我们对算法3.4做两点修改。第一，每当给当前状态集加入一个接受状态时，我们要记录当前输入的位置和与这个接受状态对应的模式 p_i。如果当前状态集已经包括了一个接受状态，那么只记录在 Lex 说明中先出现的那个模式。第二，不断地进行状态转

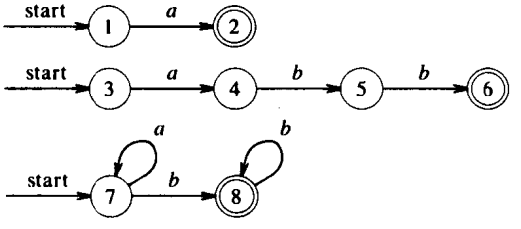
换直到遇到 NFA 进入终止。当进入终止时，我们把向前指针回退到最后一次匹配的位置。做出这个匹配的模式可以识别发现的记号，而且被匹配的词素是开始指针和向前指针之间的串。

通常，我们所写的 Lex 说明总能使某个模式（也可以是错误模式）匹配成功。如果没有一个模式能匹配成功，则意味着我们没把错误情况考虑周全，此时词法分析器需要把控制转给某个默认的错误恢复程序。

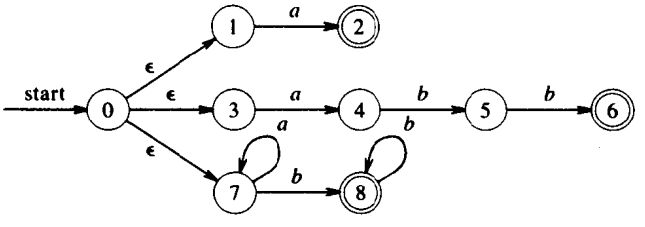
例3.18 一个简单的例子可以说明上述想法。假设有一个如下所示的 Lex 程序，它由三个正规表达式组成，且没有正规定义：

```
  a     { } /* 这里忽略了动作 */
 abb    { }
a*b+    { }
```

上述三类记号可以由图3-35a所示的自动机识别。第三个自动机是算法3.3所生成的自动机的化简形式。如前所述，我们可以将图3-35a所示的三个自动机合并成图3-35b所示的 NFA N。



a)



b)

图3-35 识别三个不同模式的 NFA

a) a, abb 和 a^*b^+ 的 NFA b) 组合后的 NFA

现在让我们用修改后的算法3.4模拟 N 在输入串 $aaba$ 上的行为。图3-36给出了读入输入串 $aaba$ 中的每一个字符时自动机可能到达的状态集和匹配上的模式。初始的状态集合是 {0, 1, 3, 7}。当输入字符 a 时，状态1、3、7分别有一个到状态2、4和7的转换。因为状态2是第一个模式的接受状态，我们记录了如下事实：在读入第一个符号 a 时第一个模式匹配成功。

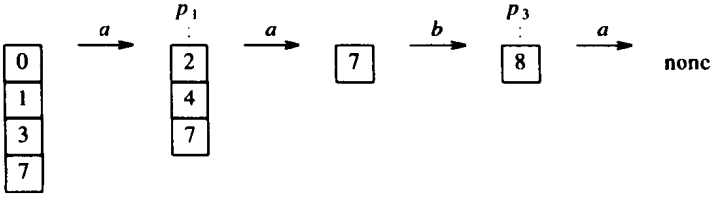


图3-36 在处理输入 $aaba$ 的过程中进入的状态集的序列

然而，状态7在第二个输入字符上有一个到自身的转换，因此我们还得继续进行下去。对输入字符 b，状态7有一个到状态8的转换。状态8是第三个模式的接受状态。到达状态8后，读入下一个字符 a 时没有可以实现的转换，所以到达了终止状态集。因为最后一次匹配是在读入第三个字符时发生的，所以报告的第三个模式匹配上的词素是 aab。 □

在 Lex 说明中与模式 p_i 相关的动作 $action_i$ 的作用如下：当识别出 p_i 的一个实例时，词法分析器就执行相关联的程序 $action_i$。注意，当 NFA 进入到一个包含接受状态的状态集时，与 p_i 对应的 $action_i$ 并不一定马上执行，只有 p_i 是产生最长匹配的模式时，$action_i$ 才被执行。

3.8.2 词法分析器的DFA

从Lex说明构造词法分析器的另一种方法是使用 DFA 完成模式匹配。这种方法与上面描述
131 的对 NFA 的模拟完全类似，只是在确定正确的模式匹配时有一些细微的差别。当我们使用子
~ 集构造算法3.2进行 NFA 到 DFA 的转换时，在一个非确定状态的子集中可能会有多个接受状
132 态。在这种情况下，在 Lex 说明中位置靠前的模式对应的接受状态具有优先权。类似于模拟 NFA，在穷尽对当前的输入符号的转换之前，我们还需要继续进行状态转换。为了找到匹配的词素，我们需要将缓冲区的向前指针返回到 DFA 最后一次进入到接受状态时的位置上。

例3.19 如果将图3-35所示的 NFA 转换成 DFA，我们将得到图3-37所示的状态转换表，其中，DFA 的状态是用 NFA 的状态序列命名的。图3-37的最后一列给出了当进入 DFA 的每个状态时它能识别的模式。例如，在NFA的状态2、4、7中，只有2是接受状态。它是正规表达式 a 对应的自动机（见图3-35a）的接受状态。因此，DFA 的状态247识别模式 a。

状 态	输入符号		识别的模式
	a	b	
0137	247	8	none
247	7	58	a
8	-	8	$a*b^+$
7	7	8	none
58	-	68	$a*b^+$
68	-	8	abb

图3-37 DFA 的转换表

注意，字符串 abb 能被两个模式 abb 和 $a*b^+$ 匹配，其接受状态分别为 NFA 的6和8。DFA 的状态68（转换表的最后一行）包括了 NFA 的两个接受状态。由于在 Lex 声明的转换规则中 abb 出现得比 $a*b^+$ 早，因此当 DFA 处于状态68时，模式 abb 匹配成功。

当输入串为 $aaba$ 时，DFA 进入的状态是通过模拟 NFA 得到的，如图3-36所示。我们来考虑另外一个例子，假定输入串是 aba。图3-37的 DFA 开始于状态0137。当读入 a 后，它进入状态247。读入 b 后进入状态58，但是再输入 a 时它没有下一个转换状态，即 DFA 依次经过状态0137、247、58后终止，其中最后一个状态包括了 NFA 的接受状态8（图3-35a）。因此在状
133 态58，DFA识别了模式 $a*b^+$，并选择输入字符串的前缀 ab 作为识别出来的词素。 □

3.8.3 实现超前扫描操作

我们在3.4节曾经介绍过，由于在某些情况下表示特定记号的模式可能需要描述实际词素后面的一部分正文，所以需要使用超前扫描操作符 / 。将一个含有 / 的模式转化成 NFA 时，可以将 / 看成 ϵ，使得我们不用真正地在输入字符串中查找 / 。然而，当由这样的正规表达式表示的字符串在输入缓冲区中被识别出来时，词素的末尾并不在 NFA 的接受状态的位置上，而是在最后一个在（假想的）/ 上具有转换的状态上。

例3.20 图3-38是识别例3.12中给出的 `IF` 模式对应的 NFA。状态6表明了关键字 `IF` 的出现。但是我们需要返回到状态2去寻找记号 `IF`。 □

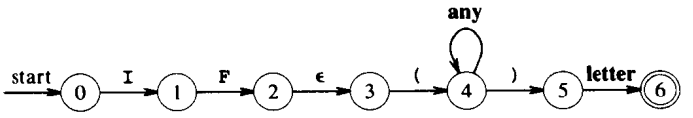


图3-38 识别 Fortran 关键字 IF 的 NFA

3.9 基于DFA的模式匹配器的优化

本节给出三个算法，这些算法已经应用于由正规表达式构造模式匹配器的实现和优化。第一个算法适于集成到 Lex 编译器中，因为它直接由正规表达式创建 DFA，而不创建中间的 NFA。

第二个算法能最小化 DFA 的状态数，所以可用于减小基于 DFA 的模式匹配器的大小。这个算法是高效的，其运行时间是 $O(n\log n)$，n 是 DFA 中的状态数。第三个算法用来生成一个比通常的二维表查询效率更快、更紧凑的 DFA 的状态转换表的表示形式。

3.9.1 NFA 的重要状态

如果一个NFA的状态有一个标记为非 ϵ 的出边，则称这个状态是重要状态。图3-25中的子集构造法在确定 ϵ -*closure*（*move*（*T*, *a*））时只使用了子集合 *T* 中的重要状态。仅当状态 *s* 是重要的，集合 *move*（*s*, *a*）才是非空的。在构造过程中，两个子集可以被认为是等同的，如果它们的重要状态相同并且两者同时包含或不包含 NFA 的接受状态。 134

当子集构造法被应用到由正规表达式经算法3.3生成的 NFA 时，我们可以利用NFA的特殊性质来将两种构造方法合二为一。合并的构造法把 NFA 的重要状态与出现在正规表达式中的符号相关联。只有字母表上一个符号出现在正规表达式中时，Thompson 构造法才创建一个重要状态。例如，对于 (*a*|*b*)***abb*，Thompson 构造法为每个 *a* 和 *b* 创建重要状态。

此外，结果 NFA 有且只有一个接受状态，但该接受状态不是重要的，因为它没有出边。通过在正规表达式 *r* 右端连接一个惟一的结束符 #，我们给 *r* 的接受状态增加一个在#上的转换，使它成为 NFA 的重要状态。换句话说，通过使用扩展的正规表达式 (*r*)#，在子集构造过程中可以忽略接受状态。当构造结束时，任何在#上有转换的 DFA 的状态都是接受状态。

我们用语法树表示扩展的正规表达式。语法树的叶节点表示基本符号，内节点表示操作符。如果一个内节点被标记为连接 · 或 * 操作符，则分别称其为*cat*-节点、or-节点或star-节点。图3-39a是一个表示正规表达式的语法树，其中*cat*-节点用圆点来表示。正规表达式语法树的构造方法与第2章中构造算术表达式语法树的方法相同。

正规表达式语法树的叶节点由符号表中的符号或 ϵ 标记。对于每一个非 ϵ 标记的叶节点，我们分配一个惟一的整数，这个整数表示叶节点的位置，同时也表示对应符号的位置。一个重复出现的符号会有多个位置。在图3-39a中，位置标记在符号的下边。在图3-39c的 NFA 中被编号的状态对应着图3-39a的叶节点的位置。这些状态是 NFA 的重要状态。在图3-39c中，非重要状态用大写字母来表示。

如果使用子集构造法并将具有相同重要状态的状态集视为等同的，则可由图3-39c的 NFA 获得图3-39b中的 DFA。把含有相同重要状态的状态集视为等同的状态集使得生成的 DF A具有更少的状态。图3-39b中的 DFA 比图3-29所示的 DFA 少一个状态。

3.9.2 从正规表达式到DFA

本节介绍如何从一个扩展的正规表达式 (*r*)# 构造 DFA。首先为正规表达式 (*r*)# 构造一个语法树*T*，然后通过遍历这棵语法树计算 *nullable*、*firstpos*、*lastpos* 和 *followpos* 四个函数。最

后我们由 *followpos* 构造DFA。函数 *nullable*、*firstpos* 和 *lastpos* 定义在语法树的节点上，用于
135 计算函数 *followpos*，而函数 *followpos* 定义在位置集合上。

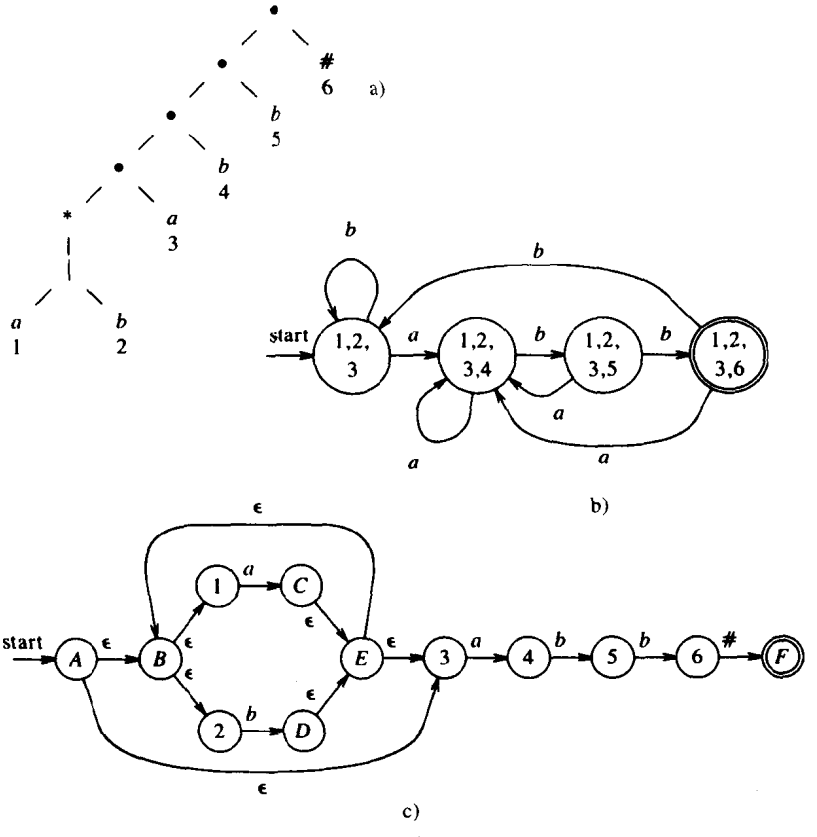


图3-39 从(*a*|*b*)**abb#* 构造的 DFA 和 NFA

a) (*a*|*b*) * *abb#* 的语法树 b) 结果 DFA c) 基础 NFA

利用 NFA 的重要状态和正规表达式语法树的叶节点的等价性，我们可以绕过 NFA 的构造而直接构造一个其状态对应于语法树的位置集合的 DFA。NFA的ϵ转换表示相当复杂的位置结构。特别地，它包含了这样的信息：什么时候一个位置可以跟随另一个位置，即输入到 DFA 的字符串中的每一个符号都可以被一个位置匹配。一个输入符号 *c* 只能由具有符号 *c* 的位置匹配，但不是每一个具有符号 *c* 的位置都一定匹配输入字符串中 *c* 的一次出现。

136 一个位置匹配一个输入符号的概念将由函数 *followpos* 定义。如果 *i* 是一个位置，那么 *followpos* (*i*) 是满足如下条件的位置 *j* 的集合：对于某个输入字符串 $\cdots cd\cdots$，位置 *i* 对应着 *c* 的出现，位置 *j* 对应着 *d* 的出现。

例3.21 在图3-39a中，*followpos* (1) = {1, 2, 3}，原因是：如果我们看到一个对应位置1的 *a*，则我们发现了闭包 (*a*|*b*)* 中 *a*|*b* 的一个出现。我们可能接下来还会看到 *a*|*b* 的下一次出现的开始位置，这就解释了为什么1和2在 *followpos* (1)中。同样，我们也可能接着看到跟在 (*a*|*b*)* 后面出现的符号的开始位置，即位置3。 □

为了计算函数 *followpos*，我们需要知道什么位置能够匹配由一个给定的正规表达式的子表达式生成的字符串的第一个或最后一个符号（这样的信息在例3.21中也非正式地用过）。如

果 r^* 是这样的子表达式，r 的每一个开始位置都跟随 r 的每个结束位置。类似地，如果 rs 是子表达式，则s的每个开始位置都跟随r的每个结束位置。

在正规表达式的语法树的每个节点n上，我们定义一个函数 *firstpos*(n)，它是与以节点 n 为根的子表达式生成的字符串中第一个符号相匹配的位置集合。同样地，我们也可以定义函数*lastpos* (n)，它是与这样的字符串的最后一个符号相匹配的位置集合。例如，如果 n 是图3-39a中语法树的根节点，那么 *fisrtpos* (n) = {1, 2, 3}, *lastpos* (n) = {6}。随后我们将给出计算这些函数的算法。

为了计算 *firstpos* 和 *lastpos*，我们需要知道产生包含空串语言的子表达式的根节点。这样的节点称为可空的。此外，如果节点 n 是可空的，则定义 *nullable*(n) 为真，否则为假。

下面我们给出计算函数 *nullable*、*firstpos*、*lastpos* 和 *followpos* 的规则。对于前三个函数，我们用一条基本规则给出基本符号的表达式，然后使用三条归纳性规则沿着语法树自底向上地确定函数的值。这三条归纳性规则分别对应于语法树中的并、连接和闭包操作符。计算 *nullabe* 和 *firstpos* 的规则在图3-40中给出。计算 *lastpos* (n) 的规则与计算 *firstpos*(n) 的规则相同，但是要调换c_1和c_2的位置，在此不再一一列举。

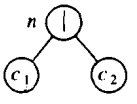
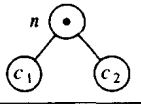
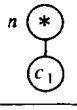
节点n	*nullable*(n)	*firstpos*(n)
n 是一个标记为 ϵ 的叶节点	**true**	$\varnothing$
n 是一个标记为位置 i 的叶节点	**false**	$\{i\}$
	*nullable*(c_1) **or** *nullable*(c_2)	*firstpos*(c_1) ∪ *firstpos*(c_2)
	*nullable*(c_1) **and** *nullable*(c_2)	**if** *nullable*(c_1) **then** *firstpos*(c_1) ∪ *firstpos*(c_2) **else** *firstpos*(c_1)
	**true**	*firstpos*(c_1)

图3-40 计算 *nullable* 和*firstpos* 的规则

nullable 的第一条规则是：如果n是标记为 ϵ 的叶节点，则 *nullable*(n) 为真。第二条规则是：如果n是由一个字符表中的符号标记的叶节点，则*nullable*(n)为假。在这种情况下，每一个叶节点对应一个输入符号，因此不能产生 ϵ 。*nullable* 的最后一条规则是：如果n是具有子节点c_1的 *star*-节点，则 *nullable*(n) 为真，因为一个表达式的闭包产生包括 ϵ 的语言。

作为另一个例子，*firstpos*的第四条规则是：如果n是具有左子节点c_1和右子节点c_2的*cat*-节点而且 *nullable*(c_1) 为真，则

$$firstpos(n) = firstpos(c_1) \cup firstpos(c_2)$$

否则，*firstpos*(n) = *firstpos*(c_1)。这个规则说明，如果正规表达式r产生一个 ϵ ，则s的第一个位置就会“透过” r，成为rs的第一个位置。否则，只有r的第一个位置是rs的第一个位置。*nullable*和*firstpos*的其余规则类似。

函数 *followpos*(i) 表明在语法树上哪些位置紧跟在 i 的后面出现。下边的两条规则定义了

一个位置可以跟随另一个位置的各种情况：

1. 如果 n 是 *cat* 节点，它具有左子节点 c_1 和右子节点 c_2，并且 i 是 $lastpos(c_1)$ 中的一个位置，则 $firstpos(c_2)$ 中的所有位置都在 $followpos(i)$ 中；

2. 如果 n 是 *star* 节点，并且 i 是 $lastpos(n)$ 中的一个位置，则所有 $firstpos(n)$ 中的位置都在 $followpos(i)$ 中。

如果每个节点的 *firstpos* 和 *lastpos* 都已经计算了，那么通过对语法树进行一次深度优先遍历就可以计算每一个位置的 *followpos*。

例3.22 图3-41显示了图3-39a中语法树的所有节点的 *firstpos* 和 *lastpos* 的值。$firstpos(n)$ 在节点的左边，$lastpos(n)$ 在节点的右边。例如，标记为 *a* 的最左叶节点的 *firstpos* 值是{1}，这是因为这个叶节点被标记为位置1。类似地，第二个叶节点的 *firstpos* 是{2}，因为它由位置2标记着。由图3-40中的第三条规则可知它们的父节点的 *firstpos* 是{1, 2}。

137
~
138

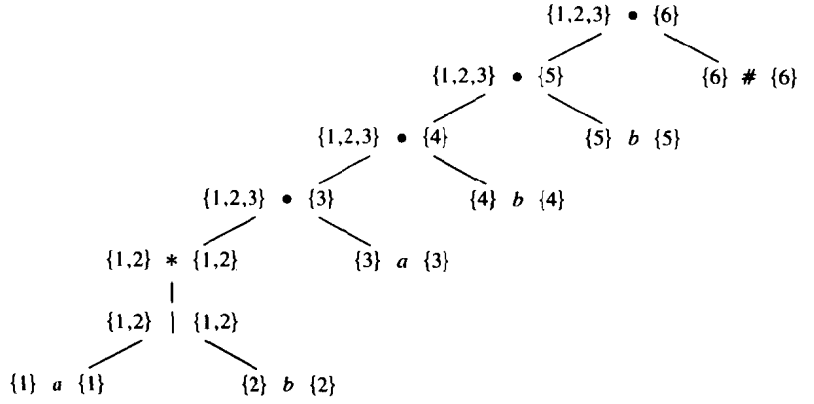


图3-41 $(a|b)^*abb\#$ 的语法树中节点的 *firstpos* 和 *lastpos*

标记为 * 的节点是惟一可空的节点。根据第四条规则中的 if 条件，该节点的父节点（表示 $(a|b)^*a$ 的节点）的 *firstpos* 的值是{1,2}与{3}的并，{1, 2}和{3}分别是其左右子节点的 *firstpos* 的值。另一方面，*else* 条件适用于该节点的 *lastpos* 函数，因为在位置3上的叶节点是不可空的。因此 *star*-节点的父节点的 *lastpos* 只包含位置3。

现在让我们自底向上地计算图3-41语法树中每个节点的 *followpos*。在*star*-节点，我们根据规则 2 把 1 和 2 加入到 $followpos(1)$ 和 *followpos* (2) 中。在 star-节点的父节点，我们根据规则1将位置3加入到 $followpos(1)$ 和 *followpos* (2) 中。在下一个 *cat*- 节点，根据规则1将4加入到 *followpos* (3) 中，在接下来的两个 *cat*-节点我们根据同一规则将 5 加入到 *followpos* (4) 中并将6加入到 *followpos* (5) 中。至此，结束了 *followpos* 的构造。图3-42总结了 *followpos* 的内容。

节点	*followpos*
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	-

图3-42 *followpos* 函数

我们可以通过创建一个有向图来表示函数 *followpos*，其节点表示位置，从 i 到 j 的有向

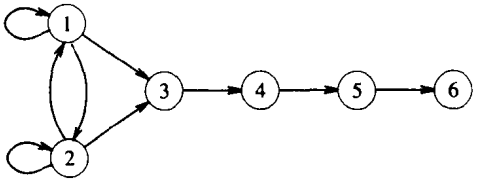


图3-43 函数*followpos*的有向图

边表示 j 在 *followpos*(i) 中。图3-43表示了图3-42所示的 *followpos* 的有向图。

这个有向图可以看成是我们正在讨论的正规表达式的无ϵ转换的 NFA，如果满足以下三条：

1. 根节点的 *firstpos* 中的所有位置作为开始状态。
2. 每一个有向边（i，j）用位置 j 上的符号标记。
3. 把与#相关的位置看成是惟一的接受状态。

毫无疑问，我们可以使用子集构造法把 *followpos* 图转化成 DFA。使用下面的算法，DFA 的构造可以基于该位置实现。 □

算法 3.5 从正规表达式 r 构造 DFA。

输入：正规表达式 r。

输出：识别 $L(r)$ 的 DFA D。

方法：

1. 构造扩展的正规表达式 (r)# 的语法树 T，其中#是附加在 (r) 后面的惟一结束标志。
2. 通过对 T 进行深度优先遍历计算函数 *nullable*、*firstpos*、*lastpos* 和 *followpos* 的值。
3. 利用图3-44所示的过程构造 *Dstates*（*D*的状态集）、生成*D*的状态转换表 *Dtran*。*Dstates* 中的状态是位置集，初始情况下，每一个状态都是“未标记的”，只有我们开始考虑其出边时，这个状态才变成“标记的”。*D* 的开始状态是 *firstpos*(*root*)，接受状态是包含与结束符#相关的位置的状态。 □

例3.23 构造正规表达式 $(a|b)^*abb$ 的 DFA。$((a|b)^*abb)$# 的语法树如图3-39a所示。只有标记为 * 的节点是可空的。函数 *firstpos* 和 *lastpos* 如图3-41所示，*followpos* 如图3-42所示。 139 ~ 140

在图3-41中，根节点的 *firstpos* 是{1, 2, 3}，我们令这个集合为 A，并考虑输入符号 a。因为位置1和3与 a 相关，所以我们令 $B = followpos(1) \cup followpos(3) = \{1,2,3,4\}$。因为这个集合没有出现过，所以令 $Dtran[A, a] := B$。

```
初始时，Dstates中惟一未标记的节点是firstpos(root)，root 是 (r)# 语法树的根节点;
while Dstates 中存在一个未标记的状态 T do begin
    标记T;
    for 每个输入符号a do begin
        令U是followpos(p)中的位置的集合，p是T中的某个位置，位置p的符号为a;
        if U非空而且不在Dstates中 then
            将U作为一个未标记的状态加入到 Dstates 中;
        Dtran [T, a] := U
    end
end
```

图3-44 DFA的构造

当考虑输入符号 b 时，在 A 中只有2与 b 相关，所以我们必须考虑集合 *followpos* (2) = {1, 2, 3}。由于这个集合已经出现过（即集合A），所以它不再加入到 *Dstates* 中，但需增加转换 *Dtran* [A，b] := A。

然后继续处理 B = {1, 2, 3, 4}。我们最后得到的状态和转换与图3-39b所示的一致。 □

3.9.3 最小化DFA的状态数

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 是惟一的（状态名不同的同构情况除外）。本节介绍如何把一个 DFA 的状态数化简到最

少，而不影响所接受的语言。假定有一个 DFA M，其状态集合是S，输入符号表是 Σ，每个状态对每个输入符号都有转换。如果不是这样，可以引入一个“死状态”d，d 在所有输入符号上都转换到 d。如果 s 在符号 a 上没有转换，加上一个在 a 上从 s 到 d 的转换。

我们说字符串 w 区别状态 s 和 t，如果：DFA M 从状态 s 出发，对输入串 w 进行状态转换，最后停在某个接受状态；从 t 出发，对输入串 w 进行状态转换，停在一个非接受状态；反之亦然。例如，空串 ϵ 区别任何接受状态和非接受状态。在图3-29的 DFA 中，输入 bb 区别状态 A 和 B，因为对输入 bb，A 到达非接受状态 C，而 B 对同样的输入到达接受状态 E。

141

极小化 DFA 的状态数的算法是把 DFA 的状态分成一些不相交的组，同一个组中的状态都是不可区别的，而不同组的状态则可以由某个输入串区别。我们把每个状态组合并成一个状态。该算法先把所有状态划分为两个组，然后逐步将这个划分精细化。在一个划分中，如果两个状态处于同一分组则说明它们还没有被任何串区别出来，反之如果两个状态处于不同分组则说明它们已经被某个串区别出来。

初始划分包含两个组：接受状态组和非接受状态组。算法的基本步骤是从当前划分中取一个状态组，比如 $A=\{s_1, s_2, \cdots, s_k\}$，选定一个输入符号 a，检查 $s_1, \cdots, s_k$ 在 a 上的转换。如果这些转换所到达的状态落入当前划分的两个或更多不同的状态组，那么 A 必须进一步划分，使得划分后的每个子集在a上的转换能落入当前划分的同一个状态组中。例如，若 s_1 和 s_2 在 a 上的转换分别到达 t_1 和 t_2，并且 t_1 和 t_2 在该划分的不同的组中，那么 A 至少分成两个子集，一个含 s_1，另一个含 s_2。注意，因为 t_1 和 t_2 可以由某个串 w 区别，所以 s_1 和 s_2 可以由串aw 区别。

在当前划分中重复上述划分组的过程，直到没有任何一组需要再分裂为止。当我们在说明为什么分在不同组中的状态是可区别的时，我们并没有说明为什么最终留在一组中的状态是不能由任何输入串区别的，这个证明留给对这一理论有兴趣的读者（见Hopcroft and Ullman [1979]）。通过从最终划分的每组取一个状态、去掉死状态和从开始状态不可到达的状态的方法构造的DFA 是接受同样语言的状态数最少的 DFA。我们把这个结论的证明也留给有兴趣的读者去完成。

算法3.6 最小化DFA的状态数。

输入：DFA M（其状态集合为S），输入符号集为Σ，转换函数为 f：$S\times\Sigma\to S$，开始状态为 s_0，接受状态集为 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. 构造具有两个组的状态集合的初始划分Π：接受状态组 F，非接受状态组 $S-F$。
2. 对Π采用图3-45所述的过程来构造新的划分Π_{new}。
3. 如果$\Pi_{new}=\Pi$，令$\Pi_{final}=\Pi$，再执行步骤4；否则，令$\Pi := \Pi_{new}$，重复步骤2。
4. 在划分Π_{final}的每个状态组中选一个状态作为该组的代表。这些代表构成了简化后的DFA M' 的状态。令 s 是一个代表状态，而且假设：在 DFA M 中，在输入 a 上有从 s 到 t 的转换。令 t 所

```
for Π中的每个组G do begin
    当且仅当对任意输入符号a，状态s和t在a上的转换
        到达Π的同一组中的状态时，才把G划分成小
        组，以便G的两个状态s和t在同一小组中；
    /* 最坏情况下，一个状态就可能成为一个组 */
    用所有新形成的小组集代替Π_new中的G;
end
```

图3-45 Π_{new}的构造

在组的代表是 r (r 可能就是 t)，那么在 M' 中有一个从 s 到 r 的 a 上的转换。令包含 s_0 的状态组的代表是 M' 的开始状态，并令 M' 的接受状态是那些属于 F 集的状态所在组的代表。注意，Π_{final} 的每个组或者仅含 F 中的状态，或者不含 F 中的状态。

5. 如果 M' 含有死状态（即一个对所有输入符号都有到自身的转换的非接受状态 d），则从 M' 中去掉它；删除从开始状态不可到达的状态；取消从任何其他状态到死状态的转换定义。 □

例3.24 让我们重新考虑图3-29中给出的 DFA。初始划分 Π 包括两个组：接受状态组 (E) 和非接受状态组 ($ABCD$)。构造 Π_{new} 时，图3-45中的算法首先考虑 (E)。因为这个组只包含一个状态，它不能再划分，所以把 (E) 仍放回 Π_{new} 中。然后，算法考虑 ($ABCD$)。对于输入 a，这些状态都转换到 B，因此分组 ($ABCD$) 不变；但对于输入 b，A、B 和 C 都转换到状态组 ($ABCD$) 的一个成员，而 D 转换到另一组的成员 E。于是，在 Π_{new} 中，状态组 ($ABCD$) 必须分裂成两个新组 (ABC) 和 (D)。因而 Π_{new} 成了 (ABC)(D)(E)。

再执行一遍图3-45中的算法时，在输入 a 上仍然没有分裂，但对输入 b，(ABC) 还要划分，因为 A 和 C 都转到 C，但 B 转到 D，而 C 和 D 不在一个组中，于是 Π_{new} 的下一个值是 (AC)(B)(D)(E)。 142~143

再执行一遍图3-45中的算法时，只有 (AC) 有划分的可能。但是对于输入 a，A 和 C 都转换到 B，对输入 b，它们都转换到状态 C，因而不必再划分。这次循环结束时，$\Pi_{new} = \Pi$。于是，Π_{final} 是 (AC)(B)(D)(E)。

如果我们选择 A 作为 (AC) 的代表，选择 B、D 和 E 作为其他单状态组的代表，最后可以得到简化的自动机。它的转换表如图3-46所示，状态 A 是开始状态，状态 E 是惟一的接受状态。

状 态	输入符号	
	a	b
A	B	A
B	B	D
D	B	E
E	B	A

图3-46 简化的DFA的转换表

例如，在这个简化的自动机中，状态 E 有一个在 a 上到状态 A 转换，因为 A 是 C 所在组的代表，并且在原来的自动机中 E 有一个在 b 上到 C 的转换。类似的变化也发生在状态 A 和输入 b 相关的表项上。其余的都是从图3-29中复制过来的。在图3-46中没有死状态，并且所有的状态都是从开始状态 A 可到达的。 □

3.9.4 词法分析器的状态最小化

为了对在3.7节生成的 DFA 应用状态最小化算法，在开始运行算法3.6时，我们必须对识别不同记号的状态进行分组，以得到一个初始划分。

例3.25 在图3-37的 DFA 中，初始划分可以将0137和7分为一组，因为它们都没有可以识别的记号；8和58可以分为一组，它们识别 a^*b^+，其他的状态自己分为一组。我们立即可以发现0137和7应该属于不同的组，因为当输入为 a 时它们进入了不同的组。同样，8和58也不属于一个组，因为当输入 b 时，它们将转换到不同的组。因此图3-37所示的 DFA 是具有相同功能的所有 DFA 中状态数最少的。 □

3.9.5 表压缩方法

正如前面指出的，可以有多种方法来实现有穷自动机的状态转换函数。因为词法分析是编译器中惟一的逐个处理输入字符流的过程，所以它占用了编译器的可观的时间。因此词法分析器需要最小化它对每一个输入字符所执行的操作个数。如果用DFA来实现词法分析器，则需要对状态转换函数进行有效的表示。由状态和字符索引的二维数组可以提供最快的访问速度，但 144

它占用了大量的空间（比如，由128个字符构成的几百个状态）。一个高度压缩但速度相对较慢的方案是使用一个链表来存储每个状态的出边所表示的转换。在链表的最后存放一个默认的转换。这个默认的转换应该是最经常用到的转换。

这里介绍一个更精妙的实现，它既有数组表示法访问速度快的优点，又有链表结构占用空间小的优点。我们使用图3-47所示的由4个数组组成的数据结构，这些数组由状态做索引。[⊖] *base* 数组用于决定存储在 *next* 和 *check* 数组中的与每个状态相关的表项的基位置。*default* 数组用于确定当前基位置无效情况下可选的基位置。

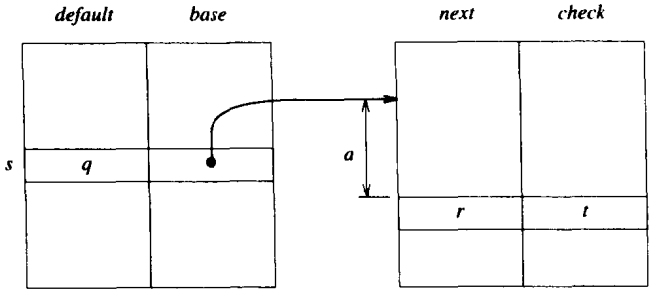


图3-47 表示状态表的数据结构

为了计算状态 *s* 在遇到字符 *a* 后要进入的状态 *nextstate* (*s*, *a*)，我们首先查看 *next* 和 *check* 这对数组。特别地，要从这两个数组中找到对应于状态 *s* 的表项所在位置 *l* = *base*[*s*]+*a*，其中 *a*可以看成整数。如果 *check* [*l*] = *s*，我们取 *next* [*l*]为 *s* 在输入 *a* 上的下一个状态 。如果 *check*[*l*]≠*s*，则令*q* = *default*[*s*]，然后用 *q* 代替 *s* 并递归地重复整个过程。该过程描述如下：

```
procedure nextstate(s, a);
    if check[base[s]+a] = s then
        return next[base[s]+a]
    else
        return nextstate(default[s], a)
```

图3-47所示的结构利用状态的相似性来缩短 *next* 和 *check* 数组的长度。例如，*s* 状态的默
145 认状态 *q* 说明我们处于“正在处理某标识符”的状态中，如图3-13中的状态10。*s* 是遇到关键字then的前缀th以及某个标识符的前缀时进入的状态。当输入字符e后，我们必须进入一个特别的状态来记录我们看到了the，否则*s*的动作和状态 *q* 的动作相同。因此，我们将 *check* [*base*[*s*]+e] 的值置为 *s*，将 *next*[*base*[*s*]+e] 的值置为标识 the 的状态。

我们可能无法选择 *base* 值使得 *next-check* 中不存在未被使用的项。经验表明，采用下述简单的策略就可以得到相当好的效果：将 *base* 设置为使得新插入的特殊表项不会与存在的表项发生冲突的最小数。与采用最小可能值相比，上述策略占用的空间只多了一点点。

如果 DFA 有每个状态 *t* 的所有入边都有相同的标号 *a* 这样的性质，我们可以把 *check* 缩短到一个由状态索引的数组中。为了实现这个方案，我们令 *check*[*t*] = *a* 并用

```
if check[next[base[s]+a]] = a then
```

来替换过程 *nextstate* 中第二行的测试语句。

⊖ 实际上当状态 *s* 加入时，应该有由*s*索引的另外一个数组，给出匹配上的模式，如果有的话。这个信息是在由 NFA 构造 DFA 状态 *s* 时获得的。

练习

3.1 下列各语言的输入字母表是什么?

a) Pascal

b) C

c) Fortran 77

d) Ada

e) Lisp

3.2 练习3.1中的各种语言对空格的使用有什么约定?

3.3 识别下面的各段程序中构成记号的词素，并给出每个记号的合理的属性值:

a) Pascal程序:

```
function max ( i, j : integer ) : integer ;
{返回整数i和j的最大者}
begin
    if i > j then max := i
    else max := j
end;
```

b) C程序:

```
int max ( i, j ) int i, j;
/* 返回整数i和j的最大者 */
{
    return i>j?i:j;
}
```

c) Fortran 77程序:

```
      FUNCTION MAX ( I, J )
C     返回整数I和J的最大者
        IF (I .GT. J) THEN
          MAX = I
        ELSE
          MAX = J
        END IF
      RETURN
```

146

3.4 使用3.2节中描述的带有“标志”的输入缓冲模式，编写3.4节中的`nextchar()`函数的程序。

3.5 在一个长度为n的字符串中，分别有多少个前缀、后缀、子串、真前缀和子序列?

* 3.6 试描述下列正规表达式所表示的语言:

a) 0(0|1)*0

b) ((ϵ|0)1*)*

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

d) 0*10*10*10*

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

* 3.7 试写出下列语言的正规定义:

a) 包含5个元音的所有字母串，其中元音按顺序出现。

b) 按词典递增序排列的所有字母串。

c) 处于/*和*/之间的串构成的注解，注解中间没有*/除非它们出现在双引号中。

* d) 所有没有重复数字的数字串。

e) 所有最多只有一个重复数字的数字串。

f) 由偶数个0和奇数个1构成的所有0和1组成的串。

g) 国际象棋的步法集合，如 $p-k4$ 或 $kbp \times qn$。

h) 所有不含子串011的0和1组成的串。

i) 所有不含子序列011的0和1组成的串。

3.8 说明练习3.1中的各种语言的数值常数的词法形式。

3.9 说明练习3.1中的各种语言的标识符和关键字的词法形式。

147

3.10 图3-48按照优先级降序的方式列出了 Lex 允许的正规表达式。在这个表中，c 表示任意的字符，r 表示一个正规表达式，s 表示一个字符串。

表达式	匹配	例子
c	任何非运算符字符 c	a
\c	字符 c 本身	*
"s"	串 s 本身	"**"
.	除换行以外的任何字符	a.*b
^	行的开始	^abc
$	行的结束	abc$
[s]	s 中的任何字符	[abc]
[^s]	不在 s 中的任何字符	[^abc]
r*	零个或多个 r	a*
r+	一个或多个 r	a+
r?	零个或一个 r	a?
$r\{m,n\}$	r 重复出现（出现次数介于 m 和 n 之间）	a{1,5}
r_1r_2	r_1 后跟有 r_2	ab
$r_1 \mid r_2$	r_1 或 r_2	a|b
(r)	r	(a|b)
r_1/r_2	后面跟有 r_2 时的 r_1	abc/123

图3-48 Lex正规表达式

a) 如果下述操作符用做要匹配的字符，它们的特殊意义必须被屏蔽：

\ " . ^ $ [ ] * + ? { } | /

使用两种引号方式就可以实现这一点。表达式 "s" 匹配字符串 s 本身，假设 s 中没有 " 出现。例如，"**" 匹配字符串 **。我们同样可以用表达式 ** 匹配这个字符串。注意没有用引号引起来的 * 表示克林闭包操作符。写出匹配字符串 "\ 的 Lex 正规表达式。

b) 在Lex中，补字符类是以符号 ^ 开始的字符类。某字符类的补字符类匹配所有不在该字符类中的字符。因此，[^a] 匹配不是a的所有字符，[^A-Za-z] 匹配所有不是大小写字母的字符等等。试证明：对于每个具有补字符类的正规表达式，都存在一个等价的不含补字符类定义的正规表达式。

148

c) 正规表达式 $r\{m, n\}$ 匹配模式 r 的 m 到 n 次出现。例如，a{1,5}匹配 a 的一到五次出现。试证明：对于每一个包含重复操作符的正规表达式，都存在一个等价的不包含重复操作符的正规表达式。

d) 操作符 ^ 匹配一行的最左端。^ 和表示补操作的操作符是同一个符号，但是 ^ 的上下文能够确定它的惟一含义。操作符 $ 匹配一行的最右端。例如，^[^aeiou]*$ 匹配任何不包含小写元音字符的行。对于每个包括 ^ 和 $ 操作符的正规表达式，是否存在一个等价的不包含这些操作符的正规表达式?

3.11 编写一个 Lex 程序，该程序复制一个文件，并将每一个非空的空白符序列用一个空格来代替。

3.12 编写一个 Lex 程序，该程序复制一个 Fortran 程序，并用 REAL 替换 DOUBLE PRECISION。

3.13 使用在练习3.9中你对Fortran 77关键字和标识符的描述来识别下列语句的记号：

```
IF(I) = TOKEN
IF(I) ASSIGN5TOKEN
IF(I) 10,20,30
IF(I) GOTO15
IF(I) THEN
```

可以用 Lex 写出你对关键字和标识符的描述吗?

3.14 在UNIX系统中，shell 命令sh在文件名表达式中使用图3-49所示的操作符来表达文件名的集合。例如，文件名表达式 `*.o` 匹配所有以.o结束的文件名，`sort.?` 匹配所有形如 `sort.c`的文件，其中*c*可以是任何字符。字符类可以缩写为 `[a-z]`。试问如何使用正规表达式来表示 shell 文件名表达式。

表达式	匹配	例子
's'	串*s*本身	'\'
\c	字符*c*本身	\'
*	任何串	*.o
?	任何字符	sort1.?
[*s*]	*s*中的任何字符	sort.[cso]

图3-49 程序sh中的文件名表达式

3.15 修改算法3.1，使之能查找输入串中被DFA接受的最大前缀。

3.16 用算法3.3为下列正规表达式构造非确定的有穷自动机，并给出它们处理输入串 *ababbab*的状态转换序列：

a) $(a|b)^*$

b) $(a^*|b^*)^*$

c) $((\epsilon|a)b^*)^*$

d) $(a|b)^*abb(a|b)^*$

149

3.17 用算法3.2把练习3.16中的 NFA 转换成 DFA。给出它们处理输入串 *ababbab* 的状态转换序列。

3.18 使用算法3.5为练习3.16中的正规表达式构造 DFA，并与练习3.17生成的 DFA 的大小进行比较。

3.19 由图3-10给出的记号的状态转换图构造 DFA。

3.20 扩展图3-40中的表使其包括正规表达式操作符 ? 和 + 。

3.21 使用算法3.6最小化练习3.18生成的 DFA 的状态。

3.22 我们可以证明：如果两个正规表达式的最少状态 DFA 除状态名以外完全相同，则这两个正规表达式等价性。使用这种技术，证明下面的正规表达式是等价的。

a) $(a|b)^*$

b) $(a^*|b^*)^*$

c) $((\epsilon|a)b^*)^*$

3.23 为下列的正规表达式构造最少状态 DFA

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

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

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

** d) 证明正规表达式 $(a|b)^*a\ (a|b)\ (a|b)\ \cdots\ (a|b)$ (共有$n-1$个 $(a|b)$) 对应的任何一个 DFA 至少有 2^n 个状态。

150

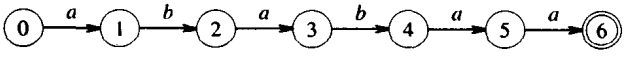
3.24 为练习3.19的状态转换表构造类似于图3-47的表示。选出默认状态并使用下面两种方法构造 *next* 数组，并比较使用的空间量。

a) 以最紧密的状态集（与它们的默认状态集有着最多不同表项的状态集）开始，将这些状态的表项放到 *next* 数组中。

b) 以随机的次序将状态的表项放到 *next* 数组中。

3.25 3.9节介绍的表压缩策略的一个变形可以通过为每个状态分配一个固定默认位置来避免递归过程 *nextstate*。采用这种非递归技术为练习3.19的状态转换表构造类似于图3-47的表示，并与练习3.24的方法所用的空间大小进行比较。

3.26 令$b_1b_2\cdots b_m$是一个模式字符串，称为关键字。一个关键字的 trie 是指具有 $m+1$ 个状态的状态转换图，该状态转换图中每一个状态对应关键字的一个前缀。对于$1\leqslant s\leqslant m$，在输入符号b_s上存在一个从状态 $s-1$ 到 s 的转换。开始和结束状态分别对应于空串和完整的关键字。关键字 *ababaa* 的trie是：



现在我们为该状态转换图的每一个状态（除开始状态以外）定义一个失败函数 f。设状态 s 和 t 分别代表关键字的前缀 u 和 v。当且仅当 v 是 u 的最长真后缀时定义 $f(s)=t$。上面 trie 的失败函数 f 是：

s	1	2	3	4	5	6
$f(s)$	0	0	1	2	3	1

例如，状态3和状态1代表关键字 *ababaa* 的前缀 *aba* 和 *a*。$f(3)=1$，因为 *a* 是 *aba* 的真后缀。

a) 为关键字 *abababaab* 构造一个失败函数。

* b) 如果 trie 的状态是0，1，…，m，0是开始状态。证明：图3-50中的算法可以正确地计算失败函数。

* c) 证明：在图3-50所示的算法的整个执行中，内层循环的赋值语句 $t:=f(t)$ 至多执行 m 次。

* d) 证明：该算法的时间复杂性是 $O(m)$。

```
/* 计算b1…bm的失败函数 */
t := 0; f(1) := 0;
for s := 1 to m-1 do begin
    while t > 0 and bs+1 ≠ bt+1 do t := f(t);
    if bs+1 = bt+1 then begin t := t+1; f(s+1) := t end;
    else f(s+1) := 0
end
```

图3-50 计算练习3.26中失败函数的算法

```
/* a1…an包含子串b1…bm吗? */
s := 0;
for i := 1 to n do begin
    while s > 0 and ai ≠ bs+1 do s := f(s);
    if ai = bs+1 then s := s+1
    if s = m then return "yes"
end;
return "no"
```

图3-51 算法KMP

3.27 图3-51中的算法 KMP 使用练习3.26中构造的失败函数 f 来确定关键字 $b_1\cdots b_m$ 是否是目标字符串 $a_1\cdots a_n$ 的子串。$b_1\cdots b_m$ 的 trie 中状态的编号为0到 m，如练习3.26(b) 所述。

a) 应用算法 KMP 来判断 *ababaa* 是否是 *abababaab* 的子串。

* b) 证明：当且仅当 $b_1\cdots b_m$ 是 $a_1\cdots a_n$ 子串时算法 KMP 返回“yes”；

* c) 证明：算法 KMP 的时间复杂性是 $O(m+n)$；

* d) 给定关键字 y，证明：利用失败函数可以在$O(|y|)$时间内创建一个具有$|y|+1$个状态的DFA，这个DFA识别正规表达式 .*y.*，其中“.”表示任意的输入字符。

** 3.28 定义字符串 s 的周期为一个整数 p 使得对于某个 $k \geq 0$，s 可以表示成 $(uv)^k u$，其中 $|uv| = p$，且 v 不是空串。例如，2和4是字符串 *abababa* 的周期。

a) 证明：当且仅当对某些长度为 p 的字符串 t 和 u 有 $st = us$ 时，p 是字符串 s 的周期。

b) 证明：如果 p 和 q 是字符串 s 的周期而且 $p + q \leq |s| + \gcd(p, q)$，则 $\gcd(p, q)$ 是 s 的周期，其中 $\gcd(p, q)$ 是 p 和 q 的最大公约数。 151 ~ 152

c) 设 $sp(s_i)$ 是字符串 s 的长度为 i 的前缀的最小周期。证明：失败函数 f 具有如下性质：$f(j) = j - sp(s_{j-1})$。

* 3.29 令字符串 s 的最短重复前缀是满足下列条件的 s 的最短前缀 u：对某个 $k \geq 1$，$s = u^k$。例如，*ab* 是 *abababab* 的最短重复前缀，而 *aba* 是 *aba* 的最短重复前缀。试构造一个算法，在 $O(|s|)$ 时间内找出字符串 s 的最短重复前缀。提示：使用练习3.26中的失败函数。

3.30 Fibonacci字符串定义如下：

$$s_1 = b$$
$$s_2 = a$$
$$s_k = s_{k-1}s_{k-2},\ k > 2。$$

例如，$s_3 = ab$，$s_4 = aba$，$s_5 = abaab$

a) s_n 的长度是多少？

** b) s_n 的最小周期是多少？

c) 试构造 s_6 的失败函数。

* d) 使用归纳法证明，s_n 的失败函数可以用 $f(j) = j - |s_{k-1}|$ 表示，其中 k 满足：对于 $1 \leq j \leq |s_n|$，有 $|s_k| \leq j + 1 < |s_{k+1}|$。

e) 应用算法 KMP 判断 s_6 是否是目标串 s_7 的子串。

f) 构造正规表达式 $.*s_6.*$ 的 DFA。

** g) 在算法KMP中，判断 s_k 是目标串 s_{k+1} 的子串时失败函数被执行的最大次数是多少？

3.31 按照下面的方法，我们可以将练习3.26中 trie 和失败函数的概念从一个关键字扩展到一个关键字集合上。trie 中的每个状态对应一个或多个关键字的前缀。开始状态对应一个空字符串，与一个完整的关键字对应的状态是终止状态。在计算失败函数时，可能会添加附加的终止状态。关键字集合{he, she, his, hers}的状态转换图如图3-52所示。

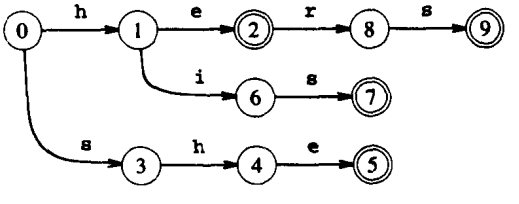


图3-52 关键字{he , she, his, hers }的trie

对于trie，我们定义转换函数 g 是一个满足下列条件的从二元组（状态，符号）到状态的映射：如果状态 s 对应某个关键字的前缀 $b_1 \cdots b_j$，s' 对应前缀 $b_1 \cdots b_j b_{j+1}$，则 $g(s, b_{j+1}) = s'$。如果 s_0 是开始状态，我们定义 $g(s_0, a) = s_0$，a 是任意输入符号但不是任何关键字的初始符号。对于任何没有定义的转换，令 $g(s, a) = fail$。注意初始状态没有 *fail* 转换。

s	1	2	3	4	5	6	7	8	9
$f(s)$	0	0	0	1	2	0	3	0	3

假设状态 s 和 t 表示某个关键字的前缀 u 和 v。当且仅当 v 是 u 的最长真后缀同时还是某个关键字的真前缀时，我们定义 $f(s)=t$。上面状态转换图的失败函数是：

例如，状态4和状态1表示前缀sh和h。$f(4)=1$，因为h是sh的最长真后缀且h是某个关键字的前缀。使用图3-53所示的算法可以按深度递增的方式计算状态的失败函数 f。状态的深度是它到开始状态的距离。

```
for 深度为1的每个状态 s do
    f(s) := s0;
for 每个深度d≥1 do
    for 每个满足g(sd, a) = s'的深度为d的状
        态 sd和字符a do begin
            s := f(sd);
            while g(s, a) = fail do s := f(s);
            f(s') := g(s, a);
    end
```

图3-53 计算关键字的 trie 的失败函数的算法

注意，由于对于任何字符 c，$g(s_0, c)\neq fail$，所以图3-53中的 while 循环一定会结束。在设置 $f(s')$为 $g(t, a)$后，如果 $g(t, a)$ 是终止状态，我们还要令 s' 为终止状态，如果它还不是的话。

153
~
154

a) 构造关键字集合{*aaa*，*abaaa*，*ababaaa*}的失败函数。

* b) 证明图3-53的算法能够正确计算失败函数。

* c) 证明计算失败函数的时间正比于关键字长度之和。

3.32 给定关键字集合 $K=\{y_1, y_2, \cdots, y_k\}$，令 g 是状态转换函数，f 是练习3.31中的失败函数。图3-54所示的算法 AC 使用 g 和 f 来判断目标串 $a_1\cdots a_n$ 是否包含是关键字的子串。状态 s_0 是K 的状态转换图的开始状态，F 是终止状态集合。

```
/* a1…an包含一个为关键字的子串吗? */
s := s0;
for i := 1 to n do begin
    while g(s, ai) = fail do s = f(s);
    s := g(s, ai);
    if s is in F then return "yes"
end;
return "no"
```

图3-54 AC 算法

a) 使用练习3.31中的转换函数和失败函数，对输入串ushers应用算法 AC。

* b) 证明：当且仅当某个关键字 y_i 是 $a_1\cdots a_n$ 的子串时，算法AC返回 “yes”。

* c) 证明：算法AC处理一个长度为 n 的字符串时至多做出了 $2n$ 次状态转换。

* d) 证明：利用关键字集合 $\{y_1, y_2, \cdots, y_k\}$ 的状态转换图和失败函数，可以在线性时间内构造出正规表达式 $.*\{y_1|y_2|\cdots|y_k\}.*$ 的 DFA，且该 DFA 至多有 $\sum_{i=1}^{k}|y_i|+1$个状态。

e) 修改算法 AC，使其打印出在目标串中发现的每个关键字。

3.33 使用练习3.32中的算法为Pascal中的关键字构造一个词法分析器。

3.34 两个字符串 x 和 y 的最长公共子序列 $lcs(x, y)$ 定义为既是 x 的子序列又是y的子序列并且是其中最长的子序列。例如，tie是striped和tiger的最长公共子序列。x 和 y 的距离 $d(x, y)$ 定义为把 x 变换成 y 所需插入和删除的最少次数。例如，d(striped, tiger) = 6.

155

a) 证明：对于任意两个串 x 和 y，它们的距离和它们的最长公共子序列的长度之间

的关系是 $d(x, y) = |x|+|y|-(2*|lcs(x, y)|)$。

* b) 编写一个算法，其输入为两个串 x 和 y，输出为 x 和 y 的一个最长公共子序列。

3.35 两个串x和y的编辑距离 $e(x, y)$ 定义为把 x 变换成 y 所需的字符插入、删除和替换的最少次数。令 $x = a_1 \cdots a_m$，$y = b_1 \cdots b_n$，$e(x, y)$ 可以使用距离数组 $d[0 \cdots m, 0 \cdots n]$ 通过动态规划算法来计算，其中 $d[i, j]$ 是 $a_1 \cdots a_i$ 和 $b_1 \cdots b_j$ 之间的编辑距离。图3-55中的算法用来计算 d 矩阵，函数 *repl* 是字符替换的代价：若 $a_i = b_j$，则 $repl(a_i, b_j) = 0$，否则为1。

```
for i := 0 to m do d[i, 0] := i;
for j := 1 to n do d[0, j] := j;
for i := 1 to m do
    for j := 1 to n do
        D[i, j] := min(d[i-1, j-1] + repl(a_i, b_j),
                       d[i-1, j] + 1,
                       d[i, j-1] + 1)
```

图3-55 计算两个串之间编辑距离的算法

a) 编辑距离和练习3.34中的距离定义有什么联系？

b) 使用图3-55中的算法计算 *ababb* 和 *babaaa* 之间的编辑距离。

c) 构造一个算法，显示出把 x 变换成 y 所需的最小编辑变换序列。

3.36 试给出一个算法，其输入为串 x 和正规表达式 r，输出为 $L(r)$ 中的串 y，使得 $d(x, y)$ 尽可能小，其中 d 是练习3.34中定义的距离函数。

编程练习

P3.1 用C语言或Pascal语言编写一个图3-10中记号的词法分析器。

P3.2 写出 Pascal 记号的说明，并根据该说明构造状态转换图。再利用该状态转换图，使用 Pascal 或C语言实现一个 Pascal 的词法分析器。 156

P3.3 完成图3-18中的Lex程序，比较由 Lex 程序生成的词法分析器和练习P3.1生成的词法分析器的大小和速度。

P3.4 写出 Pascal 记号的 Lex 说明，使用 Lex 编译器构造一个 Pascal 的词法分析器。

P3.5 编写一个程序，输入是一个正规表达式和一个文件名，输出是文件中所有包含由正规表达式表示的子串的行。

P3.6 为图3-18中的 Lex 程序添加错误恢复机制，使其在出现一个错误后还可以继续查找记号。

P3.7 从练习3.18中构造的 DFA 编写一个词法分析器程序，并和练习P3.1、P3.3中构造的词法分析器进行比较。

P3.8 构造一个能够从说明记号集合的正规表达式生成词法分析器的工具。

参考文献注释

一个语言在词法方面的限制通常是由创建该语言的环境决定的。当Fortran在1954年诞生时穿孔卡片是主要的输入介质。Fortran语言忽略空格的部分原因是因为当时打孔员需要从手写的笔记来准备卡片，经常数错空格（Backus[1981]）。Algol 58语言把硬件表示和参考语言分离是在一个设计委员的坚持下获得的："不，我永远不会用句号表示小数点"（Wegstein[1981]）。

Knuth[1973a]提出了一些输入缓冲技术。Feldman[1979b]讨论了Fortran 77中记号识别的实际困难。

正规表达式首先由Kleene[1956]开始研究，Kleene对于可以由McCulloch and Pitts[1943]提

出的描述神经活动的有穷自动机模型所能表示的事件感兴趣。Huffman[1954]和Moore[1956]最早研究有穷自动机的最小化问题。确定自动机和不确定自动机在识别语言能力上的等价性是由Rabin and Scott[1959]证明的。McNaughton and Yamada[1960]描述了一个直接从正规表达式构造DFA的算法。关于正规表达式的更多理论可以在Hopcroft and Ullman[1979]中找到。

在实现编译器时，由正规表达式说明生成词法分析器的工具非常有用，这个观点被广泛地接受了。Johnson et al. [1968]讨论了早期的这种系统。本章讨论的语言Lex是由Lesk[1975]设计
157 的，已经应用到UNIX系统中的很多编译器上。3.9节中的转换表的压缩实现方法是由S. C. Johnson提出的，他首先在Yacc语法生成器的实现中使用了这种方法（Johnson[1975]）。Dencker, Dürre and Heuft[1984]中对另一种表压缩方法进行了讨论和评价。

Tarjan and Yao[1979]以及Fredman, Komlós and Szemerédi[1984]在理论上系统地研究了转换表压缩的问题。基于这些工作，Cormack, Horspool and Kaiserswerth[1985]提出了一个非常好的散列算法。

正规表达式和自动机已经被应用于很多编译以外的领域。很多文本编辑器应用正规表达式进行上下文搜索。例如，Thompson[1968]在文本编辑器QED的上下文中描述了由正规表达式构造NFA的算法（算法3.3）。在UNIX系统中有三个应用正规表达式的查找程序：grep、egrep和fgrep。grep不允许在正规表达式中使用“并”或括号进行分组，但它允许受限的向后引用，类似于Snobol。grep使用算法3.3和算法3.4来搜索它的正规表达式模式。egrep中的正规表达式与Lex中的正规表达式相似，只是不包括循环和超前扫描。egrep使用3.7节提到的“惰性”状态构造法生成DFA，用来搜索正规表达式模式。fgrep使用Aho and Corasick[1975]中提出的算法匹配由关键字集合组成的模式，该算法在练习3.31、练习3.32中进行了讨论。Aho[1980]分析了这些程序的相关性能。

正规表达式在文本检索系统、数据库查询语言和文件处理语言（如AWK（Aho, Kernighan, and Weinberger[1979]））中都有着广泛的应用。Jarvis[1976]在描述印制电路的缺陷时也使用了正规表达式。Cherry[1982]使用练习3.32中的关键字匹配算法查找手稿中的错误用语。

练习3.26、练习3.27中的字符串模式匹配算法来自Knuth, Morris and Pratt[1977]。该论文还对字符串的周期进行了讨论。另一个有效的字符串匹配算法是由Boyer和Moore[1977]提出的，他们证明了不需要检查目标串中的所有字符就可以确定子串匹配成功。在字符串匹配中散列法也是一种有效的技术(Harrison[1971])。

练习3.34提到的最长公共子序列的概念已经用在了UNIX系统的文件比较程序diff中(Hunt and McIlroy[1976])。Hunt and Szymanski[1977]中描述了一个有效的计算最长公共子序列的实用算法。Wagner and Fischer[1974]中提出了计算练习3.35中的最短编辑距离的算法。Wagner[1974]给出了练习3.36的解答。Sankoff and Kruskal[1983]包含了最小距离识别方法的广
158 泛应用，从遗传序列的研究到语音处理中的问题。

第4章 语法分析

每一种程序设计语言都具有描述程序语法结构的规则。例如，Pascal程序由程序块组成，程序块由语句组成，语句由表达式组成，表达式由记号组成，等等。程序设计语言结构的语法可以用2.2节介绍的上下文无关文法或BNF范式（Backus-Naur范式）表示法来描述。文法为程序语言设计者和编译器编写者提供了很大的便利：

- 文法为程序设计语言提供了精确、易懂的语法说明。
- 从某类文法可以自动构造一个有效的语法分析器，用来判断一个源程序在语法上是否正确。语法分析器的构造过程能揭示出在语言和编译器的最初设计阶段未被发现的语法二义性和其他难以进行语法分析的结构。
- 设计合理的文法使程序设计语言具有良好的结构。这将有利于把源程序翻译成正确的目标代码并有利于错误检测。现在已经有很多把基于文法的描述转换成工作程序的工具。
- 语言经过一段时期的发展，可能需要增加新的结构及新的功能。如果存在基于该语言的文法描述的实现，这些新结构就能更容易地加入到语言中。

本章将介绍编译器中使用的典型语法分析方法。我们首先介绍基本概念，然后讨论适合于手工实现的技术，最后介绍自动生成工具中所使用的算法。由于程序经常会出现一些语法错误，所以我们扩展了这些语法分析方法，使之能从常见的错误中得以恢复。 159

4.1 语法分析器的作用

在本书的编译器模型中，语法分析器接收词法分析器提供的记号串，检查它们是否能由源程序语言的文法产生，如图4-1所示。我们希望语法分析器能用易于理解的方式提示语法错误信息，并能从常见的错误中恢复过来，以便后面的输入能继续处理下去。

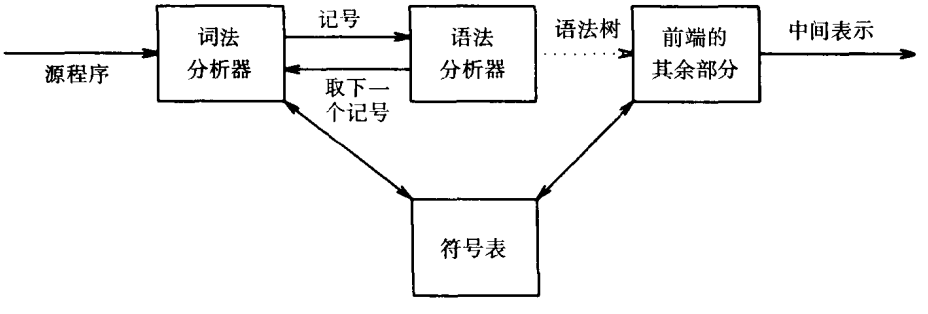


图4-1 语法分析器在编译器中的位置

典型的文法的语法分析器有三类。一类是通用的语法分析方法，如Cocke-Younger-Kasami算法和Earley算法，这些方法能分析任何文法（参见参考文献注释）。然而这些方法在生成编译器时效率太低。编译器常用的是自顶向下和自底向上的方法。正如它们的名字所示，自顶向下语法分析器沿着从顶（根）向底（叶）的方向建立分析树，而自底向上语法分析器则沿着从叶向根的方向建立分析树。不论是哪一种方法，语法分析器都是自左向右地扫描输入字符串，

每次读一个符号。

最有效的自顶向下和自底向上分析方法只能处理文法的一些子类。然而，这些子类中的某些文法，如LL文法和LR文法，足以描述程序设计语言的大部分语法结构。LL文法的语法分析器常由手工实现，如2.4节中介绍的构造LL文法的语法分析器的方法。大多数LR文法的语法分析器则常利用自动生成工具来构造。

本章假设语法分析器的输出是对词法分析器产生的记号流的分析树的某种表示。事实上，在分析过程中编译器还可以完成许多其他任务。例如，把与各种记号有关的信息收集到符号表中、进行类型检查和其他一些语义分析检查、产生如第8章所示的中间代码，等等。我们把所
160 有这些任务都包括在图4-1的“前端的其余部分”框中，并在下面的三章中详细讨论它们。

本节的剩余部分将考虑语法错误的性质和错误恢复的一般策略。我们在讨论各种语法分析方法时将详细介绍两种错误恢复策略：一种称为*紧急方式恢复策略*，另一种称为*短语级恢复策略*。编译器的编写者可以自行实现每一种策略，我们只是给出一些有关方法的提示。

4.1.1 语法错误的处理

如果编译器只处理正确程序，它的设计和实现就可以大大简化。但是程序设计人员经常会编写出错误的程序，因此好的编译器应该能帮助程序员识别和定位错误。虽然错误经常发生，但是绝大多数语言在设计时都没有考虑到出错处理。如果我们的口语也和计算机语言一样要求语法的准确性，我们的文明会大不相同。大多数程序设计语言的说明都没有描述编译器应该怎样响应语法错误，而是把它留给了编译器的设计者来处理。因此，在设计编译器的开始阶段我们就应该规划出错处理策略，这样既可以简化编译器的结构也能改进它对错误的响应能力。

我们知道程序可能包含不同级别的错误。下边是一些程序错误的示例：

- 词法错误，如标识符、关键字或操作符的拼写错误。
- 语法错误，如算术表达式的括号不配对。
- 语义错误，如操作符作用于不相容的操作数。
- 逻辑错误，如无限的递归调用。

源程序的多数错误诊断和恢复都集中在语法分析阶段。原因之一是多数错误都是语法错误，或者当来自词法分析器的记号流违背定义程序设计语言的文法规则时才暴露出来。另一个原因是现代语法分析方法的准确性使得它们能非常有效地检查出程序中的语法错误。在编译阶段准确地诊断语义和逻辑错误是非常困难的。本节将介绍几种基本的语法错误恢复技术，它们的实现技术将与本章的各种语法分析方法一起讨论。

语法分析器中出错处理程序的基本目标是：

- 清楚而准确地报告错误的出现。
- 迅速地从每个错误中恢复过来，以便能继续检查后面的错误。
- 不能过分降低正确程序的处理速度。

有效地实现这些目标并不是一件容易的事。

161 幸好，常见的错误都是简单错误，用相对简单的错误处理机制就足以处理这些错误。但是，在有些情况下，错误的发生远远先于发现它的位置，而且很难诊断出这种错误的准确性质。更困难的是，出错处理程序可能需要猜测程序员编程时的意图。

有些分析方法，如LL方法和LR方法，可以尽快地检查出语法错误。更准确地说，它们具有*萌发前缀性*（viable-prefix property），即当它们一旦发现一个输入字符串的前缀不是该语言任何字符串的前缀时就能检查出错误。

例4.1 为了鉴别实践中出现的错误种类，我们来看一下Ripley and Druseikis [1978]在抽样检查学生的Pascal程序时发现的错误。

他们发现，错误并不经常出现，60%的被编译程序的语法和语义都是正确的。即使有错误出现，错误的数量也相当少。80%的出错语句只含一个错误，只有13%的出错语句含有两个错误。多数错误都是微不足道的，90%的错误是单个记号错。

多数错误可以简单地分为以下几类：60%是标点符号错，20%是操作符或运算对象错，15%是关键字错，剩下的5%是其他类型的错误。多数标点符号错都属于分号的不正确使用。

让我们看一个具体的例子。下面是一个Pascal程序：

```
(1)     program prmax(input, output);
(2)     var
(3)         x, y: integer;
(4)     function max(i:integer; j:integer) : integer;
(5)     { return maximum of integers i and j }
(6)     begin
(7)         if i > j then max := i
(8)         else max := j
(9)     end;
(10)    begin
(11)        readln (x,y);
(12)        writeln (max(x,y))
(13)    end.
```

一种常见的标点错误是在函数说明的参数表中出现的，这种错误是：在应该使用分号的位置错误地使用了逗号（如在第(4)行第一个分号的位置用了逗号）。另一种常见的错误是在一行的末尾忽略了必须写的分号（如第(4)行末尾的分号）。还有一种常见的错误是在`else`出现的前一行的末尾多加了分号（如在第(7)行的末尾加了分号）。

分号错误普遍出现的原因是不同语言在分号的用法上有很大的差别。在Pascal语言中，分 162
号是语句的分隔符；在PL/1和C语言中，分号则是语句的结束符。研究表明，后一种用法较少出错（Gannon and Horning [1975]）。

操作符错误的典型例子是漏写了 := 中的冒号。关键字拼写错误通常比较少见，其典型的例子是`writeln`漏写了`i`。

多数Pascal编译器都可以毫无困难地处理普通的插入、删除和修改错误。事实上，有些Pascal编译器可以正确地编译带有一个普通的标点或操作符错误的程序；它们只是给出一个警告信息，并正确地指出错误的结构。

然而，另一类常见的错误是很难正确修复的，比如漏写了`begin`或`end`（如在第(9)行中漏写）。多数编译器都不能修复这类错误。 □

出错处理程序应该怎样报告错误呢？至少，它们应该报告源程序的错误被检测到的位置，因为实际错误可能就在它前面几个记号中。很多编译器普遍采用的办法是：显示出错的程序行，用指针指出检测到错误的位置。如果能够知道实际错误可能是什么，编译器还会显示附带的诊断信息，如“此处遗漏了分号”等。

一旦检查出错误，语法分析器将如何恢复这个错误呢？正如我们将看到的，有很多一般性的策略，但没有哪种策略占绝对优势。多数情况下，语法分析器在检测到一个错误后就放弃继

续分析的做法是不妥的，因为继续处理输入字符串可以发现更多的错误。通常，一些语法分析器试图将自己恢复到某一状态，以便能够继续分析输入字符串或正确地处理出现的错误。

不充分地恢复程序可能会引起大量令人烦恼的“伪”错误的出现，这些错误不是程序设计人员造成的，而是由于错误恢复时改变了语法分析器的状态而引起的。同样，语法错误的恢复也可能引起语义伪错误，这些错误会在语义分析或代码生成阶段被检查出来。例如，错误恢复时，语法分析器可能跳过某个变量的声明，如变量zap。以后在表达式中碰到变量zap时，虽然语法上没有错，但由于符号表中没有变量zap的表项，会产生“zap没有定义”的出错信息。

编译器一个保守的策略是：如果在输入流中查到某个错误，并且它离前一个错误非常近，则抑制这个错误信息的出现，即在发现一个语法错误后，编译器应该在成功地分析几个记号之后才报告下一个错误信息。然而，在某些情况下，可能会有太多的错误，以至于编译器无法继续进行合理的处理。试想一下，Pascal编译器对输入的Fortran程序会如何反应呢？考虑到各种错误情况的出现以及合理处理，错误恢复的策略是一个需要认真考虑的折衷。

163

正如前面所提到的，有些编译器试图猜测程序员编程时的意图并修复错误，如PL/C编译器（Conway and Wilcox[1973]）。除非处理初学者写的短程序，否则恢复大量错误的过程不可能是有效的。事实上，随着交互式计算以及良好程序设计环境日趋重要，错误恢复机制将趋于简单化。

4.1.2 错误恢复策略

语法分析器可以采用的语法错误恢复策略有很多种。虽然没有一种策略被普遍接受，但有几种策略已经得到广泛应用。我们将主要介绍下面几种策略：

- 紧急方式恢复策略。
- 短语级恢复策略。
- 出错产生式策略。
- 全局纠正策略。

紧急方式恢复策略。紧急方式恢复是最容易实现的方法，适用于多数语法分析方法。当发现错误时，语法分析器开始抛弃输入记号，每次抛弃一个记号，直到发现某个指定的同步记号为止。同步记号通常是定界符，如分号或**end**，它们在源程序中的作用是明显的。当然，编译器的设计者必须恰当地选择同步记号。这种方法常常跳过大量的输入记号，而不检查其中是否有其他错误。和下面的几种方法相比，这种方法比较简单，不会陷入死循环。所以，当一个语句中出现的错误数较少时，这种方法比较合适。

短语级恢复策略。发现错误时，语法分析器对剩余的输入字符串做局部纠正，即用一个能使语法分析器继续工作的字符串来代替剩余输入的前缀。典型的局部纠正包括用分号代替逗号、删除多余的分号或插入遗漏的分号。局部纠正的选择由编译器的设计者来完成。当然，编译器的设计者必须仔细选择替换字符串，以免引起死循环。例如，若总是在当前输入符号的前面插入一些字符，就有可能引起死循环。这种类型的替换可以纠正任何输入字符串，并且已经被用在多个错误修复编译器中。该方法首先被用于自顶向下的语法分析中。它的主要缺点是难以应付实际错误出现在诊断点之前的情况。

164

出错产生式策略。如果对经常遇到的错误有很清楚的了解，我们可以扩充语言的文法，增加产生错误结构的产生式。然后用由这些错误产生式扩充的文法构造语法分析器。如果语法分析器使用了出错产生式，就可以产生适当的错误诊断信息，指出我们在输入字符串中识别出的

错误结构。

全局纠正策略。我们总是希望一个理想的编译器在处理不正确的输入字符串时做尽可能少的改动。有一些算法可以选择最小的修改序列，以获得全局代价最小的错误纠正。如果给定错误输入串x和文法G，这些算法会发现y的一棵分析树，以便使用最少的符号插入、删除和修改操作把x变换成正确的输入字符串y。不幸的是，实现这些算法的时间和空间开销太大，目前人们只是进行了一些理论上的探讨。

必须指出，最近似正确的纠错程序并不一定是程序员所关心的。然而，“最小代价纠正”的概念已经成为评价错误恢复技术的一种标准，并且已经被用于短语级恢复方法中最优替换字符串的选择。

4.2 上下文无关文法

程序设计语言的许多结构都包含固有的递归结构，这种递归结构可以用上下文无关文法定义。例如，可能有用如下规则定义的条件语句：

如果 S_1 和 S_2 是语句，E是表达式，则“**if** E **then** S_1 **else** S_2”是语句。 (4-1)

这种形式的条件语句不能用正规表达式说明。在第3章中，我们已经看到正规表达式能够说明记号的词法结构。如果使用语法变量*stmt*表示语句类，*expr*表示表达式类，我们可以使用下面的文法产生式很容易地表示出(4-1)：

stmt → **if** ***expr*** **then** ***stmt*** **else** ***stmt*** (4-2)

本节将回顾一下上下文无关文法的定义，并介绍一些和语法分析有关的术语。由2.2节中可知，上下文无关文法由终结符、非终结符、开始符号和产生式组成。

1. 终结符是组成字符串的基本符号。在讨论程序设计语言的文法时，“记号”和“终结符”是同义词。在 (4-2) 中，关键字 **if**、**then** 和 **else** 都是终结符。 165

2. 非终结符是表示字符串集合的语法变量。在 (4-2) 中，*stmt* 和 *expr* 是非终结符。非终结符所定义的字符串集合有助于定义该文法所产生的语言。非终结符强加给语言一种层次结构，这种层次结构对语法分析和翻译都非常有用。

3. 在文法中，有一个非终结符被指定为开始符号。开始符号表示的字符串集合就是文法所定义的语言。

4. 文法的产生式说明了终结符和非终结符组合成串的方式。每个产生式由非终结符开始，跟随一个箭头（有时用 ::= 代替箭头），然后是非终结符和终结符组成的串。

例4.2 具有下述产生式的文法定义了简单的算术表达式。

expr → ***expr op expr***
expr → (***expr***)
expr → − ***expr***
expr → **id**
op → +
op → −
op → *
op → /
op → ↑

在该文法中，终结符包括 **id**、+、−、*、/、↑、(和)，非终结符包括 *expr* 和 *op*，*expr* 是

开始符号。 □

4.2.1 符号的使用约定

为了避免反复说明“这些是终结符”、“那些是终结符”等等，本书以后将采用下列与文法有关的约定：

1. 下列符号是终结符：

1) 字母表中比较靠前的小写字母，如a、b、c等。

2) 操作符，如+、-等。

3) 标点符号，如括号、逗号等。

4) 数字0，1，…，9。

5) 黑体串，如 **id**、**if** 等。

2. 下列符号是非终结符：

166 1) 字母表中比较靠前的大写字母，如 A、B、C 等。

2) 字母S，它常常代表开始符号。

3) 小写斜体名字，如*expr*、*stmt*等。

3. 字母表中比较靠后的大写字母，如 X、Y、Z 等，表示文法符号，也就是说，可以是非终结符也可以是终结符。

4. 字母表中比较靠后的小写字母，如 u，v，…，z 等，表示终结符号的串。

5. 小写希腊字母，如 α、β、γ 等，表示文法符号的串。因此，一个通用产生式可以写作 $A\rightarrow\alpha$，箭头左边（产生式的*左部*）是一个非终结符 A，箭头右边是文法符号串（产生式*右部*）。

6. 如果 $A\rightarrow\alpha_1$、$A\rightarrow\alpha_2$、…、$A\rightarrow\alpha_k$ 是所有以 A 为左部的产生式（称为 A *产生式*），则可以把它们写成 $A\rightarrow\alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$，我们将 α_1、α_2、…、α_k 称为 A 的*候选式*。

7. 除非另有说明，否则第一个产生式左部的符号是开始符号。

例4.3 使用上述简写约定，例4.2的文法可以简写为：

$$E \rightarrow E\ A\ E \mid (\ E\ ) \mid -\ E \mid \mathbf{id}$$
$$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$$

根据上述约定，E和A是非终结符，E是开始符号，其他符号都是终结符。 □

4.2.2 推导

我们可以使用多种方法观察文法定义语言的过程。在2.2节中，我们将该过程看成是分析树的建立过程，但推导也是描述文法定义语言过程的有用方法，其核心思想是把产生式看成重写规则，即用产生式右部的串来代替左部的非终结符。事实上，推导给出了自顶向下构造分析树过程的精确描述。

例如，考虑下面的算术表达式文法：

$$E \rightarrow E + E \mid E * E \mid (\ E\ ) \mid -\ E \mid \mathbf{id} \tag{4-3}$$

其中，非终结符E表示一个表达式。产生式 $E\rightarrow -E$ 意味着前面带有减号的表达式仍然是表达式。这个产生式允许用$-E$ 代替出现的任何 E，以便从简单的表达式产生更复杂的表达式。如果用
167 $-E$代替单个 E，这个动作可以描述为

$$E \Rightarrow -\ E$$

读为“E 推导出$-E$”。产生式 $E\rightarrow(E)$ 表示可用 (E) 代替在文法符号串中出现的任何 E。如

$E*E \Rightarrow (E)*E$ 或 $E*E \Rightarrow E*(E)$。

我们可以从 E 开始，不断地（以任何顺序）应用产生式，得到一个替换序列。例如，

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$$

我们把这个替换序列称为从 E 到 $-(\mathbf{id})$ 的推导。上边的推导证明，字符串 $-(\mathbf{id})$ 是表达式的一个特殊实例。

更抽象地，我们说 $\alpha A\beta \Rightarrow \alpha\gamma\beta$，如果 $A\rightarrow\gamma$ 是产生式，而且 α 和 β 是任意的文法符号的串。如果 $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$，则说 α_1 推导出 α_n。符号$\Rightarrow$表示“一步推导”。通常我们用 $\overset{*}{\Rightarrow}$ 表示“零步或多步推导”，因此，

1. 对任何串α，$\alpha \overset{*}{\Rightarrow} \alpha$。

2. 如果$\alpha \overset{*}{\Rightarrow} \beta$，而且$\beta \Rightarrow \gamma$，则$\alpha \overset{*}{\Rightarrow} \gamma$。

类似地，我们用 $\overset{+}{\Rightarrow}$ 表示“一步或多步推导”。对于开始符号为 S 的文法 G，我们可以用 $\overset{+}{\Rightarrow}$ 关系来定义 G 所产生的语言 $L(G)$。$L(G)$ 中的字符串只包含 G 的终结符。当且仅当$S \overset{+}{\Rightarrow} w$时，我们说终结符串 w 在 $L(G)$中。终结符串 w 称为 G 的句子。由上下文无关文法产生的语言称为上下文无关语言。如果两个文法产生同样的语言，则称这两个文法等价。

对于开始符号为 S 的文法 G，如果$S \overset{*}{\Rightarrow} \alpha$，则称 α 为 G 的句型，其中 α 可能含有非终结符。句子是不含非终结符的句型。

例4.4 字符串 $-(\mathbf{id}+\mathbf{id})$ 是文法(4-3)的句子，因为存在如下推导：

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id}) \tag{4-4}$$

出现在这个推导中的字符串E、$-E$、$-(E)$、…、$-(\mathbf{id}+\mathbf{id})$ 都是该文法的句型。我们用 $E \overset{*}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$表示 $-(\mathbf{id}+\mathbf{id})$ 可以由E推导出来。

按推导长度进行归纳，我们可以证明文法(4-3)产生的语言中的每个句子都是由二元操作符 + 和 *、一元操作符-、括号以及运算对象 **id** 组成的算术表达式。同样，按算术表达式的长度进行归纳，我们也可以证明这样的算术表达式都可以由文法(4-3)产生。因此，文法(4-3)正好产生所有包括二元操作符 + 和 *、一元操作符 - 、括号以及操作数 **id** 的算术表达式的集合。 □

在推导的每一步都有两个选择，首先我们需要选择被替换的非终结符，然后再选择用于替 168
换该非终结符的候选式。例如，例4.4中的推导(4-4)也可以从 $-(E+E)$ 开始如下进行：

$$-(E+E) \Rightarrow -(E+\mathbf{id}) \Rightarrow -(\mathbf{id}+\mathbf{id}) \tag{4-5}$$

(4-5)中的每个非终结符都使用与例4.4中相同的右部来代替，但代替的顺序不一样。

为了理解语法分析器是怎样工作的，我们需要考虑每一步都替代最左非终结符的推导。这样的推导叫做*最左推导*。如果 $\alpha \Rightarrow \beta$ 是最左推导，可以写成$\alpha \underset{lm}{\Rightarrow} \beta$。因为推导(4-4)是最左推导，它可以写成

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

使用前面的约定，每一步最左推导可以写成$wA\gamma \underset{lm}{\Rightarrow} w\delta\gamma$，其中 w 只含终结符，$A\rightarrow\delta$ 是推导所用的产生式，γ 是文法符号的串。为了强调 α 通过最左推导推导出 β 这一事实，我们写 $\alpha \overset{*}{\underset{lm}{\Rightarrow}} \beta$。如果$S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$，则称 α 是该文法的*左句型*。

我们可以类似地定义*最右推导*，即每步推导都替代最右非终结符的推导。最右推导有时也

称为规范推导。

4.2.3 分析树和推导

分析树可以看成是推导的图形表示，但它不能显示出替代顺序的选择。回顾2.2节，分析树的每个内节点都标以某个非终结符A。A的子节点从左到右分别被用来替换A所使用的产生式右部的各符号标记。分析树的叶节点用非终结符或终结符来标记，它们从左到右构成一个句型，称为树的边界或果实。例如，**-(id+id)** 的推导过程如(4-4)所示，其分析树如图4-2所示。

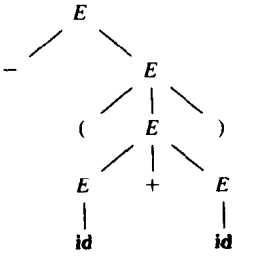


图4-2 **-(id+id)** 的分析树

为了说明推导和分析树之间的关系，我们考虑任意的推导 $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$，其中 α_1 是单个非终结符A。对推导中的每个句型α_i，构造产生 α_i 的分析树。该过程是对 i 的归纳。$\alpha_1 = A$ 对应的分析树是标有 A 的单个节点，是归纳的基础。为了完成这种归纳，假设我们已经构造了产生 $\alpha_{i-1} = X_1X_2\cdots X_k$（在我们的约定中，$X_i$ 代表一个非终结符或终结符）的分析树，假设 α_i 是用 $\beta = Y_1Y_2\cdots Y_r$ 代替α_{i-1}中的非终结符X_j所产生的，即在推导的第i步中，对 α_{i-1} 应用产生式 $X_j \rightarrow \beta$，推导出 $\alpha_i = X_1X_2\cdots X_{j-1}\beta X_{j+1}\cdots X_k$。

为了模拟推导的这一步，我们在当前的分析树上找到左边第 j 个叶子，即标记为 X_j 的叶子。我们为这个叶子建立 r 个子节点，并从左到右标记为 Y_1，Y_2，…，Y_r。对于 $r = 0$这种特例，即 $\beta = \epsilon$，我们为第 j 个叶子建立一个子节点，其标记为 ϵ。

例4.5 考虑推导(4-4)。从该推导所构造出的分析树序列如图4-3所示。推导的第一步是 $E \Rightarrow -E$。为了模拟这一步，我们为最初的分析树的根节点 E增加两个子节点，分别标记为-和 E。

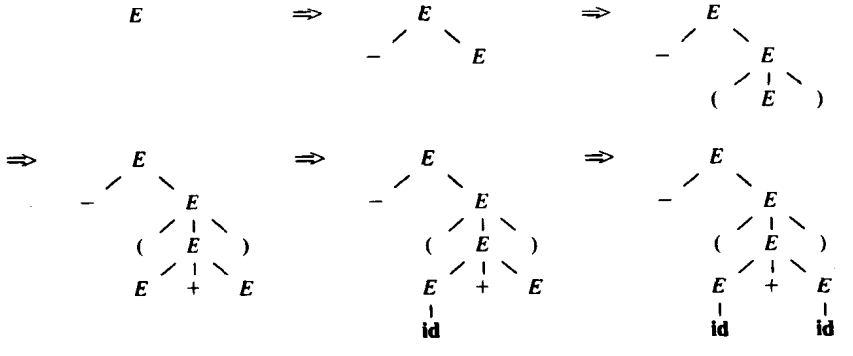


图4-3 从推导(4-4)所构造的分析树

推导的第二步是$-E \Rightarrow -(E)$，所以为第二个分析树的标记为E的叶节点增加三个子节点，分别标记为（、E和），从而获得带有结果$-(E)$的第三棵树。如此继续，我们将得到如第六棵树所示的整个分析树。 □

如前所述，分析树忽略了句型中符号被替代的顺序。例如，若构造推导(4-5)的分析树，其最终分析树和推导(4-4)的最终分析树(如图4-3所示)相同。如果只考虑最左推导（或最右推导），则可以消除推导过程中产生式应用顺序的不一致。不难看出，每棵分析树都有一个与之对应的惟一的最左推导和惟一的最右推导。不难理解，我们可以用产生分析树方法来代替推导。以后

169 ~ 170

我们将经常使用最左推导和最右推导来进行语法分析。然而，每一个句子不一定只有一个分析树，或者说不一定只有一个最左推导或最右推导。

例4.6 让我们再次考虑算术表达式文法(4-3)。句子 **id+id*id** 有两个不同的最左推导：

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

相应的分析树如图4-4所示。 □

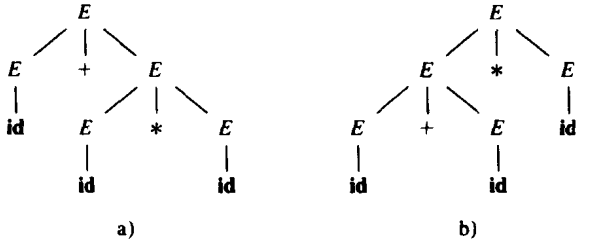


图4-4 **id+id*id**的两棵分析树

注意，图4-4a的分析树反映了通常所假定的 + 和 * 的优先级，而图4-4b的分析树则没有。也就是说，习惯上 * 比 + 具有更高的优先级，因而表达式 $a+b*c$ 被看成 $a+(b*c)$，而不是 $(a+b)*c$。

4.2.4 二义性

给定一个文法 G，如果 $L(G)$ 中存在一个具有两棵或两棵以上分析树的句子，则称 G 是二义性的。我们也可以如下定义二义性文法：如果 $L(G)$ 中存在一个具有两个或两个以上最左（或最右）推导的句子，则 G 是二义性文法。很多语法分析器要求所处理的文法是无二义的，否则对具有二义性的句子无法确定应该选择哪棵分析树。某些应用可能要求我们可以构造适应于二义性文法的语法分析器，不过，这种文法要具有消除二义性的规则，以便语法分析器能够“抛弃”不需要的分析树而为每个句子保留惟一一棵分析树。 171

4.3 文法的编写

文法能够描述程序设计语言的大部分语法成分，但不能描述程序设计语言的全部语法成分。当词法分析器从输入字符串产生记号序列时，将完成一定量的语法分析工作。对输入字符串的某些限制（如标识符的声明必须先于它们的使用）不能用上下文无关文法来描述。因此，语法分析器接受的记号序列形成了程序设计语言的超集。语法分析以后的各编译阶段必须分析语法分析器的输出，以保证输入字符串符合语法分析器无法检查的那些规则（参见第6章）。

本节首先考虑词法分析器和语法分析器的分工。每种语法分析方法只能处理一种形式的文法。为了适应所选择的分析方法，我们常常不得不改写初始文法。适于表达式的文法常常用结合律和优先级信息来构造，如2.2节所讨论的那样。本节将考虑一些用于改写文法的变换规则，以便产生适于自顶向下分析的文法。本节最后将讨论一些不能用上下文无关文法描述的语言结构。

4.3.1 正规表达式和上下文无关文法的比较

正规表达式所描述的每一种结构都可以用上下文无关文法来描述。例如，正规表达式 $(a|b)^*abb$ 和下述文法描述的语言皆为由 a 和 b 组成的以 abb 结尾的字母串：

$$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$$
$$A_1 \rightarrow bA_2$$
$$A_2 \rightarrow bA_3$$
$$A_3 \rightarrow \epsilon$$

我们可以机械地把一个不确定的有穷自动机（NFA）转换成一个等价的上下文无关文法，该文法是由图3-23的 NFA 按照下列规则构造的：

1. 对NFA的每个状态 i，创建一个非终结符 A_i。
2. 如果状态 i 遇见输入符号 a 转换到状态 j，则引入产生式 $A_i \rightarrow aA_j$。
3. 如果状态 i 遇见输入符号 ϵ 转换到状态 j，则引入产生式 $A_i \rightarrow A_j$。
4. 如果状态 i 是接受状态，则引入产生式 $A_i \rightarrow \epsilon$。
5. 如果状态 i 是开始状态，则 A_i 是文法的开始符号。

既然正规集都是上下文无关语言，我们可能要问：为什么要用正规表达式而不用上下文无关文法来定义语言的词法？理由如下：

172

1. 语言的词法规则通常非常简单，不必动用强大的文法来描述。
2. 对于记号，正规表达式比上下文无关文法提供了更简洁且易于理解的定义。
3. 从正规表达式可以自动地构造出有效的词法分析器，从任何文法都很难构造词法分析器。
4. 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。

对于任何一种语言来说，哪些结构应作为词法规则，哪些结构应作为语法规则，并没有严格的界限。正规表达式对描述标识符、常数和关键字等词法结构最有用。另一方面，文法在描述括号配对、begin-end 配对、if-then-else 对应等嵌套结构时最有用。正规表达式不能描述这些嵌套结构。

4.3.2 验证文法所产生的语言

尽管编译器设计者很少关心一个完整的程序设计语言的文法是否正确地产生该语言，验证给定的文法或产生式集合是否产生指定的语言是非常重要的。通过精确而且简要的文法并研究文法所产生的语言可以研究棘手的语言结构。后面我们会构造一个这样的文法。

对“文法 G 产生语言 L”的证明包括两部分：我们必须证明由 G 产生的每个字符串都在 L 中；反之，L 中的每个字符串都能由 G 产生。

例4.7 下边的文法 G 能而且仅能产生所有配对的括号串：

$$S \rightarrow (S)S \mid \epsilon \tag{4-6}$$

为了证明 $L(G) = \{s \mid s$是一个配对的括号串$\}$，我们首先证明从 S 推导出的所有句子都是配对的括号串，然后证明每个配对括号串都可以从 S 推导出来。我们对推导的步数使用数学归纳法，证明从 S 推导出的所有句子都是配对的括号串。从 S 经过一步推导得出的终结符串只有空串。空串可以视为配对的括号串。显然，当推导步数为1时命题正确。

现在假设所有少于 n 步的推导所产生的句子都是配对的括号串，我们来考察一个 n 步最左推导。这个推导一定具有如下形式：

$$S \Rightarrow (S)S \overset{*}{\Rightarrow} (x)S \overset{*}{\Rightarrow} (x)y$$

由于从 S 推导出 x 和 y 的步数少于 n 步，根据归纳假设，x 和 y 都是配对括号串。因此，$(x)y$ 也一定是配对括号串。

因此我们已经证明了任何从 S 中推导出来的串都是配对的括号串。下边，我们对括号串的 173
长度使用数学归纳法，证明每个配对括号串都可以从 S 推导出来。空串是配对的括号串，并且可以使用产生式 $S \rightarrow \epsilon$ 由 S 推导出来。于是，长度为0的括号串可以从 S 推导出来。

假设每个长度小于 $2n$ 的配对括号串都可以从 S 推导出来，我们来考虑长度为 $2n$ $(n \geq 1)$ 的配对括号串 w。可以肯定，w 是由左括号开始的，令 (x) 是 w 的具有相同个数的左右括号的最短前缀。那么 w 可以写作 $(x)y$，其中 x 和 y 都是配对括号串。既然 x 和 y 的长度小于 $2n$，由归纳假设，它们可以从 S 推导出来。于是，我们可以找出如下形式的推导：

$$S \Rightarrow (S)S \overset{*}{\Rightarrow} (x)S \overset{*}{\Rightarrow} (x)y$$

从而证明 $w = (x)y$ 也能从 S 推导出来。 □

4.3.3 消除二义性

有些二义性文法可以通过改写来消除二义性。作为一个例子，我们来消除下面的“不匹配 else”文法的二义性：

$$\begin{aligned} stmt \rightarrow\ & \textbf{if}\ expr\ \textbf{then}\ stmt \\ |\ & \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\ |\ & \textbf{other} \end{aligned} \tag{4-7}$$

这里，**other** 代表任何其他语句。按照这个文法，复合条件语句

if E_1 **then** S_1 **else if** E_2 **then** S_2 **else** S_3

的分析树如图4-5所示。文法(4-7)是具有二义性的，因为串

if E_1 **then if** E_2 **then** S_1 **else** S_2 (4-8)

有两棵分析树，如图4-6所示。

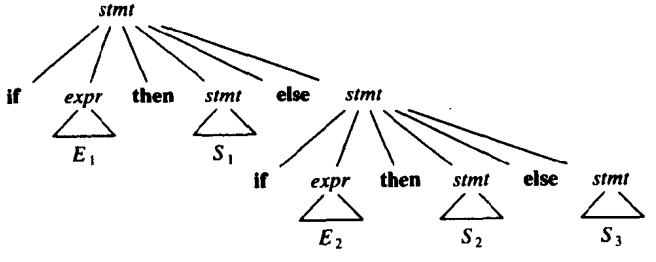


图4-5 条件语句的分析树 174

所有包含这种条件语句的程序设计语言都使用第一种分析树。一般规则是，“每个 **else** 和前面最近的没有配对的 **then** 配对”。这条避免二义性的规则可以直接并入文法中。例如，可以把文法(4-7)改写成下面的无二义性文法，其基本思想是：出现在 **then** 和 **else** 之间的语句必须是“配对”的，即它不能以一个未配对的 **then** 后面跟随任意的非 **else** 语句结束，于是 **else** 会被迫与这个未配对的 **then** 匹配。配对的语句是一个不包含不配对语句的 **if-then-else** 语句或者任何非条件语句。因此，按照上述思想改写后的文法如下：

$$\begin{aligned} stmt \rightarrow\ & matched_stmt \\ |\ & unmatched_stmt \end{aligned}$$

$$
\begin{aligned}
\textit{matched_stmt} &\rightarrow \textbf{if } \textit{expr} \textbf{ then } \textit{matched_stmt} \textbf{ else } \textit{matched_stmt} \\
&\quad | \ \textbf{other} \\
\textit{unmatched_stmt} &\rightarrow \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\
&\quad | \ \textbf{if } \textit{expr} \textbf{ then } \textit{matched_stmt} \textbf{ else } \textit{unmatched_stmt}
\end{aligned} \tag{4-9}
$$

该文法和文法(4-7)产生同样的语言，但对串(4-8)只有一棵分析树，即每个 **else** 与前面最近的没有配对的 **then** 配对的分析树。

4.3.4 消除左递归

如果文法G具有一个非终结符A使得对某个字符串α存在推导 $A \overset{+}{\Rightarrow} A\alpha$，则称 G 是*左递归*的。自顶向下语法分析法不能处理左递归文法，因此我们需要一种消除左递归的变换。在2.4节我们讨论了简单的左递归（其中存在形如 $A \rightarrow A\alpha$ 的产生式）。这里我们将研究一般情况。在2.4节我们说明了左递归产生式 $A \rightarrow A\alpha|\beta$ 可以由下面的非左递归产生式来代替：

$$
\begin{aligned}
A &\rightarrow \beta A' \\
A' &\rightarrow \alpha A' \mid \epsilon
\end{aligned}
$$

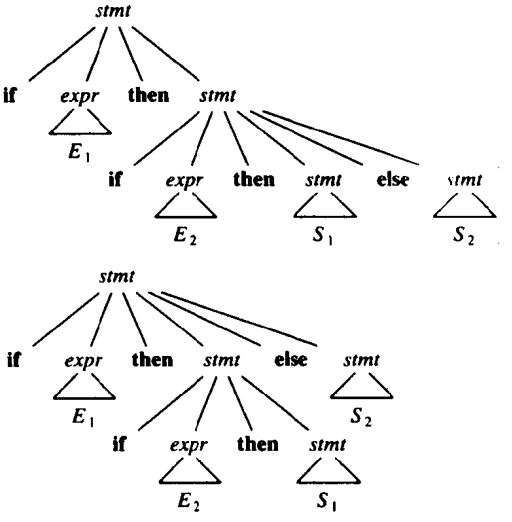


图4-6 一个具有二义性的句子的两棵分析树

这种变换没有改变从 A 推导出的字符串集合。这条规则适用于很多文法。

例4.8 考虑下面的算术表达式文法：

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \textbf{id}
\end{aligned} \tag{4-10}
$$

消除 E 和 T 的直接左递归（形如 $A \rightarrow A\alpha$ 的产生式），可以得到

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE' \mid \epsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT' \mid \epsilon \\
F &\rightarrow (E) \mid \textbf{id}
\end{aligned} \tag{4-11}
$$

□

无论有多少 A 产生式，我们都可以用下面的技术来消除直接左递归。首先，把 A 产生式放在一起：

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

其中，每个 β_i 都不以 A 开头。然后用下面的产生式代替 A 产生式：

$$
\begin{aligned}
A &\rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\
A' &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon
\end{aligned}
$$

变换后的非终结符 A 与变换前的非终结符 A 产生同样的字符串集合，但已经没有左递归了。这种方法可以从 A 产生式和 A' 产生式（假定 α_i 都不等于 ϵ）消除直接左递归，但不能消除包括两步或多步推导的左递归。例如，考虑文法

$$
\begin{aligned}
S &\rightarrow Aa \mid b \\
A &\rightarrow Ac \mid Sd \mid \epsilon
\end{aligned} \tag{4-12}
$$

非终结符 S 是左递归的，因为 $S \Rightarrow Aa \Rightarrow Sda$，但不是直接左递归。

下面的算法4.1能够系统地消除文法中的左递归。该算法对于所有无循环推导（形如$A \overset{+}{\Rightarrow} A$的推导）和$\epsilon$产生式（形如 $A \rightarrow \epsilon$ 的产生式）的文法都有效。循环推导和ϵ产生式都可以系统地从文法中消除掉（参见练习4.20和练习4.22）。

算法4.1 消除左递归。

输入：无循环推导和 ϵ 产生式的文法 G。

输出：与 G 等价的无左递归文法。

方法：对文法 G 应用图4-7的算法。注意，得到的非左递归文法可能含有ϵ产生式。 □

```
1. 以某种顺序排列非终结符A1, A2, …, An;
2. for i := 1 to n do begin
       for j := 1 to i - 1 do begin
           用产生式 Ai → δ1γ | δ2γ | … | δkγ 代替每个形如 Ai → Ajγ 的产生式,
           其中, Aj → δ1 | δ2 | … | δk 是所有的当前 Aj 产生式;
       end
       消除Ai产生式中的直接左递归
   end
```

图4-7 从文法中消除左递归的算法

现在我们来说明图4-7中算法正确的原因。算法第2步中的外层 **for** 循环被循环执行 i-1次以后，对于任何形如 $A_k \rightarrow A_l \alpha$ 的产生式，其中 $k < i$，必有 $l > k$。结果，在下一次循环中，循环变量为 j 的内层循环不断地增大形如 $A_i \rightarrow A_m \alpha$ 的产生式中 m 的下限，直到 $m \geqslant i$。然后，算法消除 A_i 产生式的直接左递归，迫使 m 大于 i。

例4.9 让我们把消除左递归算法应用到文法(4-12)上。从技术上讲，因为有ϵ产生式，算法4.1不一定有效，但在这种情形下产生式 $A \rightarrow \epsilon$ 是无害的。

令非终结符的次序是 S、A。在 S 产生式中没有直接左递归，所以在算法第2步，对于$i = 1$，什么也没做。$i = 2$时，用 S 产生式替换 $A \rightarrow Sd$ 中的 S，得到下面的 A 产生式：

$$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$$

消除 A 产生式中的直接左递归，产生下面的文法：

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow bdA' \mid A' \\ A' &\rightarrow cA' \mid adA' \mid \epsilon \end{aligned}$$

□

4.3.5 提取左因子

提取左因子是一种对产生适合预测分析的文法非常有用的文法变换。提取左因子的基本思想是：当不清楚应该用两个选择中的哪一个来替换非终结符 A 时，可改写 A 产生式来推迟这种决定，直到看见足够多的输入能做出正确选择为止。

例如，我们有如下两个产生式：

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

看到输入记号 **if** 时，我们不能立刻决定选择哪个产生式来扩展 *stmt*。一般地，如果 $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ 是 A 的两个产生式，输入字符串由从 α 导出的非空串开始，我们不知道是用 $\alpha\beta_1$ 来

扩展 A 还是用 $\alpha\beta_2$。然而，我们可以通过先将 A 扩展到 $\alpha A'$ 来推迟这个决定。然后，扫描完由 α 导出的输入字符串后，再把 A' 扩展成 β_1 或 β_2，亦即提取左因子。经过这样的变换以后，原来的产生式变为

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

算法4.2 提取左因子。

输入：文法 G。

输出：一个等价的提取了左因子的文法。

方法：对每个非终结符 A，找出它的两个或更多候选式的最长公共前缀 α。如果 $\alpha \neq \epsilon$，即有一个非平凡的公共前缀，则用下面的产生式代替所有 A 产生式 $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$，其中 γ 表示所有不以 α 开头的候选式：

$$A \rightarrow \alpha A' \mid \gamma$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

其中 A' 是一个新的非终结符。反复应用这种变换，直到任一非终结符都没有两个候选式具有公共前缀为止。 □

例4.10 下面是从文法(4-7)中抽象出来的文法：

$$S \rightarrow iEtS \mid iEtSeS \mid a$$
$$E \rightarrow b \tag{4-13}$$

这里的 i、t 和 e 分别代表 **if**、**then** 和 **else**，E 和 S 表示表达式和语句。提取左因子后，该文法
178 变为

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow eS \mid \epsilon \tag{4-14}$$
$$E \rightarrow b$$

于是，如果输入是 i，我们可以将 S 扩展到 $iEtSS'$，等到 $iEtS$ 出现后，再决定是将 S' 扩展到 eS 还是 ϵ。当然，由于文法(4-13)和(4-14)都是具有二义性的，所以对于输入 e，我们不清楚应该选择S'的哪个候选式。例4.19中将讨论解决这种问题的方法。 □

4.3.6 非上下文无关语言的结构

有些语言不能用任何文法产生，这并不奇怪。事实上，在很多程序设计语言中，仅用文法难以完成某些语法结构的说明。本小节将给出一些这样的结构，并用简单的抽象语言来说明其难度。

例4.11 考虑抽象语言 $L_1 = \{wcw \mid w \text{ 属于}(a \mid b)^*\}$。$L_1$ 是所有由 c 隔开的两个相同 a、b 串组成的字母串集合，例如 $aabcaab$。这个语言是检查程序中标识符的声明应先于其引用的抽象，即 wcw 中的第一个 w 表示标识符 w 的声明，第二个 w 表示它的引用。可以证明该语言不是上下文无关语言，但该证明超出了本书的范围。这个例子意味着 Algol 和 Pascal 等程序设计语言都不是上下文无关语言，因为它们要求标识符的声明先于引用，并且允许标识符任意长。

由于上述原因，描述 Algol 和 Pascal 语法的文法并不定义标识符中的字符，而只是用文法中 **id** 这样的记号代表所有的标识符。在这类语言的编译器中，语义分析阶段检查标识符的声

明是否先于引用。 □

例4.12 语言 $L_2 = \{a^n b^m c^n d^m \mid n \geq 1 且 m \geq 1\}$不是上下文无关语言。$L_2$是由正规表达式 $a^*b^*c^*d^*$ 所表示的语言的子集合，在它的每个句子中，a 和 c 的个数相等，b 和 d 的个数相等。(a^n意味着 a 被写 n 次。)它是“过程声明的形参个数和过程引用的实参个数应该一致”的抽象，a^n 和 b^m 表示两个过程说明的形参表中分别有 n 和 m 个参数，c^n 和 d^m 分别表示调用这两个过程时的实参表。

注意，过程定义和引用的语法并不涉及到参数的个数。例如，一个类似于 Fortran 语言的中的CALL语句的文法如下： 179

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

其中带有 *expr* 的产生式。通常在语义分析阶段检查 call 中的实参个数是否正确。 □

例4.13 语言 $L_3 = \{a^n b^n c^n \mid n \geq 0\}$ 是$L(a^*b^*c^*)$ 子集合，每个串包括相等个数的 a、b 和 c。L_3 不是上下文无关语言。下面是一个与 L_3 相关的问题。设打字时用下划线标记的正文，排版输出时改用斜体。为了把在行式打印机打印的文本文件转换成适于在照相排版机上输出的文本，我们需要用斜体代替下划线。在打字机上打印一个用下划线标记的单词时，首先在键盘上敲打一串与这个单词对应的字母键，然后敲打相等数量的退格键，最后敲打相等数量的底线键。如用 a 表示任意的字母键，b 表示退格键，c 表示底线键，则 L_3 可以表示所有用下划线标记的单词。结论是我们不能用文法描述用下划线标记的单词集合。但是，如果用 < 字母,退格,底线 >三元组表示用下划线标记的单词，我们可以用正规表达式 $(abc)^*$ 表示用下划线标记的单词集合。 □

有趣的是，有些语言类似于 L_1、L_2、L_3，但却是上下文无关语言。例如，$L_1' = \{wcw^R \mid w$ 属于 $(a \mid b)^*\}$ 是上下文无关语言，其中 w^R 表示 w 的逆序。它可以由下面的文法产生：

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

$L_2' = \{a^n b^m c^m d^n \mid n \geq 1 且 m \geq 1\}$是上下文无关语言，其文法为：

$$\begin{aligned} S &\rightarrow aSd \mid aAd \\ A &\rightarrow bAc \mid bc \end{aligned}$$

$L_2'' = \{a^n b^n c^m d^m \mid n \geq 1 且 m \geq 1\}$也是上下文无关语言，其文法为：

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow aAb \mid ab \\ B &\rightarrow cBd \mid cd \end{aligned}$$

最后，$L_3' = \{a^n b^n \mid n \geq 1\}$是上下文无关语言，其文法为：

$$S \rightarrow aSb \mid ab$$

值得注意的是，L_3'是不能用正规表达式表示的典型例子。为证明这一点，假设 L_3' 是由某个正规表达式表示的语言，亦即我们可以构造一个 DFA D 接收 L_3'。D 的状态数必定有限，设为 k。令 D 读入 ϵ，a，aa，…，a^k 到达的状态分别为 s_0，s_1，s_2，…，s_k，即s_i是 D读 入 i 个 a 后进入的状态。 180

因为 D 只有 k 个不同的状态，在序列 s_0，s_1，…，s_k 中至少有两个状态相同。假设 s_i 和 s_j

相同。因为 a^ib^i 属于 L_3'，从状态 s_i，D 可接受 i 个 b，并到达一个接收状态 f。D 中还存在一条从开始状态 s_0 到 s_i 再到 f 的路径，该路径的标记为 a^jb^i，如图4-8所示。于是，D 也接受 a^jb^i，与D接受 L_3' 的假设矛盾。

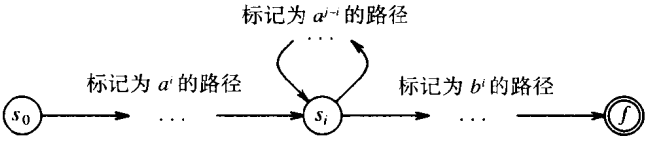


图4-8 接受 a^ib^i 和 a^jb^i 的 DFA D

通俗地讲，我们说“有穷自动机不能计数”是指有穷自动机不能接受像 L_3' 这样的语言（它要求在看到 b 之前记住 a 的个数）。类似地，我们说“上下文无关文法能计2项的数，但不能计3项的数”是指上下文无关文法可以定义 L_3'，但不能定义 L_3。

4.4 自顶向下语法分析

本节介绍自顶向下语法分析的基本概念以及构造无回溯自顶向下语法分析器的方法。无回溯自顶向下语法分析器也称为预测语法分析器。本节还将定义LL(1)文法，这种文法的预测语法分析器可以自动生成。本节除了形式化2.4节中关于预测语法分析器的讨论之外，我们只考虑非递归的预测语法分析器。本节最后讨论错误恢复问题。自底向上的语法分析器将在4.5节至4.7节讨论。

4.4.1 递归下降语法分析法

自顶向下语法分析的目的是为输入字符串寻找最左推导，或者说，从根节点（文法开始符号）开始，自上而下、从左到右地为输入字符串建立一棵分析树，并以预先确定的顺序创建分析树的节点。在2.4节，我们已经讨论了一种不需要回溯的特殊递归下降分析法，称为预测分析法。现在我们考虑自顶向下分析的一般形式，称为*递归下降分析法*。它可能需要回溯，即需要重复地扫描输入。然而，需要回溯的语法分析器是不常见的，其原因是在分析程序设计语言的结构时很少需要回溯。即使在分析自然语言的情况下，回溯也不是非常有效的，并且列表的方法（如练习4.63的动态规划算法或Earley [1970] 的方法）更可取。关于一般分析方法的描
181 述请参见 Aho and Ullman[1972b]。

下面是一个需要回溯的例子。当需要回溯时，建议使用一种记录输入轨迹的方法。

例4.14 考虑下述文法和输入字符串 $w = cad$：

$$\begin{aligned} S &\rightarrow cAd \\ A &\rightarrow ab \mid a \end{aligned} \tag{4-15}$$

为了自顶向下地为 w 建立分析树，我们首先建立只具有标记为 S 的单个节点的树。输入指针指向 w 的第一个符号 c。然后，我们用 S 的第一个产生式来扩展该树，图4-9a所示的树。

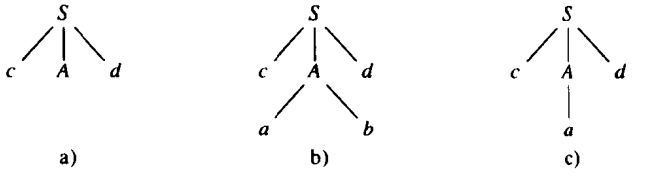


图4-9 自顶向下分析的步骤

最左边的叶子标记为 c，匹配 w 的第一个符号。现在，我们将输入指针移到 w 的第二个符号 a。考虑下一个标记为 A 的叶子。用 A 的第一个候选式扩展 A，得到图4-9b所示的树。现在我们已经匹配了第二个输入符号 a，再将输入指针移到第三个输入符号 d，把它和下一个标记为 b的叶节点进行比较。因为 b 和 d 不匹配，报告失败，回到 A，看是否还有别的候选式可试。

回到 A 时，我们必须将输入指针重置到第二个符号，即第一次进入 A 时的位置。这意味着A的程序（类似于图2-17中非终结符的过程）必须将输入指针保存在一个局部变量中。现在尝试 A 的第二种候选式，得到图4-9c所示的分析树。叶子 a 匹配 w 的第二个符号，叶子 d 匹配 w 的第三个符号。因为已经产生了 w 的分析树，我们停止分析并宣告分析成功。 □

虽然递归下降语法分析器带有回溯能力，左递归文法也会使其进入无限循环，也就是说，当试图扩展 A 时，我们可能最终会发现：我们在不断地试图扩展 A，但输入指针并没有前移。

4.4.2 预测语法分析器

在许多情况下，通过仔细地编写文法，消除左递归，提取左因子，我们可以获得一个有效 182
的文法，这个文法可以用不带回溯的递归下降语法分析器（即2.4节讨论的预测语法分析器）来分析。为了构造预测语法分析器，对给定的当前输入符号 a 和将要扩展的非终结符 A，我们必须知道，在 A 的所有可选产生式 $A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$ 中，哪个候选式是惟一能推导出以 a 开头的串。预测语法分析器能够通过观察候选式所推导出的第一个符号，确定正确的候选式。这种方法可以检测出多数程序设计语言中具有不同关键字的控制流结构。例如，假设我们有如下产生式：

stmt → **if** *expr* **then** *stmt* **else** *stmt*
| **while** *expr* **do** *stmt*
| **begin** *stmt_list* **end**

那么关键字 **if**、**while**、**begin** 告诉我们如果我们想找到一条语句哪个候选式是惟一可能成功的选择。

4.4.3 预测语法分析器的状态转换图

在2.4节，我们已经讨论了怎样用递归程序实现预测语法分析器，如图2-17所示。在3.4节，我们看到对于词法分析器的设计者来说，状态转换图（transition diagram）是非常有用的设计工具。我们也可以为预测语法分析器创建状态转换图。

词法分析器的状态转换图和预测语法分析器的状态转换图具有明显的区别。对于预测语法分析器，每个非终结符都对应一个状态转换图，边上的标记是记号和非终结符。记号（终结符）上的转换意味着如果该记号是下一个输入符号，就应进行该转换。非终结符 A 上的转换是对与 A 对应的过程的调用。

为了由文法构造预测语法分析器的状态转换图，首先需要消除文法中的左递归，然后提取左因子，并对每个非终结符 A 执行如下操作：

1. 创建一个开始状态和一个终态（返回状态）。

2. 对每个产生式 $A \rightarrow X_1X_2\cdots X_n$，创建一条从开始状态到终止状态的路径，边上的标记分别为X_1，X_2，…，X_n。

预测语法分析器以状态转换图为基础完成分析工作，其工作方式如下：开始，语法分析器进入状态图的开始状态，输入指针指向输入符号串的第一个符号。如果经过一些动作后，语法分析器进入状态 s，且在状态图上从状态 s 到状态 t 的边上标记终结符 a，而下一个输入符又正

好是a，则语法分析器将输入指针向右移动一位指向下一个符号，语法分析器进入状态 t。另一方面，如果边上标记的是非终结符A，则语法分析器进入 A 的初始状态，但不移动输入指针。一旦语法分析器到达 A 的终止状态，则立刻进入状态 t。事实上，语法分析器从状态 s转移到状态t时，它已经从输入符号串“读”了 A。最后，如果从s到t有一条标记为ϵ的边，那么语法
183 分析器从状态s直接进入状态 t 而不移动输入指针。

基于状态转换图的预测分析程序试图进行终结符和输入的匹配，并且，当它经过标记为非终结符的边时，进行潜在的递归过程调用。一种非递归的实现方法是，当在状态 s 上有一个标记为非终结符的指向其他状态的转换时，则将状态 s 压入栈中，当到达该非终结符的终止状态时，将状态 s 弹出栈。下面我们将更详细地讨论状态转换图的实现。

如果给定的状态转换图是确定的，即一个状态对于一个输入仅有一个转换，则上述方法是有效的。如果出现二义性，可以用下边例4.15中的方法解决。如果不能消除不确定性，我们就不能构造预测语法分析器。但是我们可以构造递归下降语法分析器，用回溯的方法尝试所有可能的情况（如果这是我们能找到的最好的分析策略）。

例4.15 图4-10给出了文法（4-11）所对应的状态转换图。惟一的二义性在于确定是否经过ϵ边。如果我们把E'的开始状态的出边解释为：对 E' 的开始状态，如果下一个输入是+，则选择+上的转换，否则选择ϵ上的转换。对 T' 也作同样的假定，我们就消除了二义性，进而就可以为文法(4-11)编写预测语法分析器程序。 □

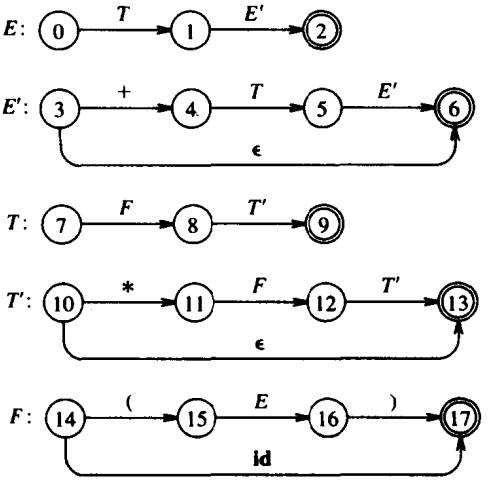


图4-10 文法(4-11)的状态转换图

我们可以通过图的变换化简状态转换图。这些变换与2.5节中文法的变换类似。例如，在 E' 的状态转换图中，对 E' 自身的调用可以被跳
184 转到开始状态的转换所代替，如图4-11a所示。

图4-11b是E'的等价状态转换图。用图4-11b所示的状态转换图代替图4-10中 E 的状态转换图中 E' 上的转换，可以得到图4-11c中的状态转

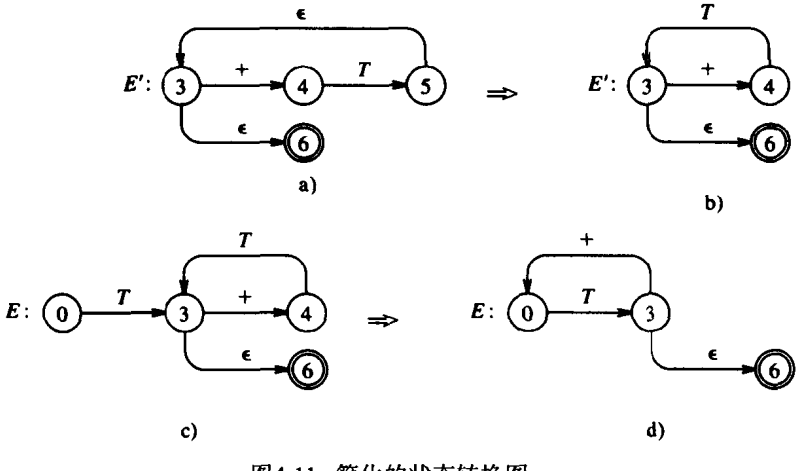


图4-11 简化的状态转换图

换图。最后，我们观察到图4-11c的第一个和第三个节点是等价的，所以我们将这两个节点合并，结果如图4-11d所示。图4-12的第一个图重复了图4-11d。把同样的技术应用到 T 和 T' 的状态转换图中，结果如图4-12所示。用 C 实现的与图4-12对应的预测语法分析器要比图4-10所对应的预测语法分析器快20%~25%。

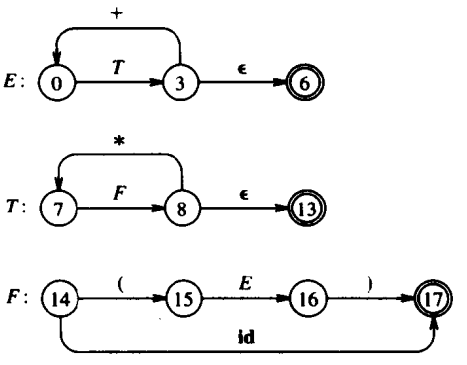


图4-12 算术表达式的简化状态转换图

4.4.4 非递归的预测分析

通过显式地维护一个状态栈，而不是通过隐式的递归调用，我们可以构造非递归的预测语法分析器。预测分析的关键问题就是确定用于扩展非终结符的产生式，图4-13中的非递归语法分析器通过查分析表来选取产生式。下面我们将看到怎样从文法直接构造分析表。

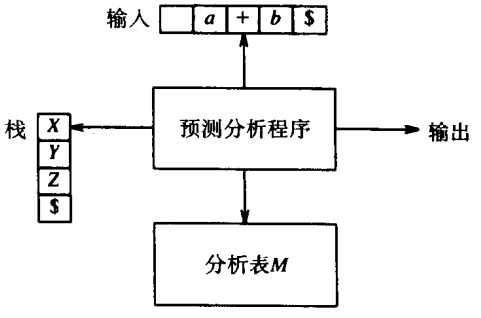


图4-13 非递归的预测语法分析器模型

表驱动的预测语法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。输入缓冲区包含要分析的串，后面跟一个符号$，$是输入串的结束标记。栈用来存放文法符号序列。栈底符号是 $。初始时，栈中含有文法的开始符号及其下边的$。分析表是一个二维数组$M[A,a]$，A 是非终结符，a 是终结符或 $。

语法分析器由一个按如下方式工作的程序控制：程序根据栈顶当前的符号 X 和当前输入符号 a 决定语法分析器的动作：

1. 如果 $X = a = \$$，则语法分析器宣告分析成功并停止。

2. 如果 $X = a \neq \$$，则语法分析器弹出栈顶符号 X，并将输入指针移到下一个输入符号上。

3. 如果 X 是非终结符，则程序访问分析表 M 的 $M[X,a]$ 项。$M[X,a]$ 项是文法的一个 X 产生式或者是出错信息。例如，如果 $M[X,a] = \{X \rightarrow UVW\}$，则语法分析器用 WVU（U 在栈顶）代替栈顶符号 X。至于输出，我们假设语法分析器只是打印出所用的产生式，当然也可以执行其他代码。如果 $M[X,a]$ = **error**，则语法分析器调用错误恢复程序。

185
~
186

语法分析器的行为可以用它的*格局*来描述，格局中给出了栈的内容和剩余的输入。

算法4.3 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：开始时，语法分析器的格局是 S 在栈里（其中 S 是 G 的开始符号且在栈顶），$w$$在输入缓冲区。图4-14是用预测分

```
令ip 指向w$的第一个符号;
repeat
    令X是栈顶符号，a是ip 指向的符号;
    if X 是终结符或者是$ then
        if X = a then
            从栈中弹出X，ip 指向下一个符号
        else error()
    else    /* X是非终结符 */
        if M[X, a] = X→Y1 Y2…Yk then begin
            从栈中弹出X;
            将Yk，Yk-1，…，Y1压入栈，Y1在栈顶;
            输出产生式X→Y1Y2…Yk
        end
        else error()
until X = $    /* 栈空 */
```

图4-14 预测分析程序

析表 M 分析输入串的程序。 □

例4.16 考虑例4.8中的文法(4-11)。该文法的预测分析表如图4-15所示。表中空白表项表示出错，非空白表项表示一个产生式，用来替换栈顶的非终结符。注意，稍后我们再说明怎样选择这些表项。

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

图4-15 文法(4-11)的分析表 M

如果输入是 **id+id*id**，预测语法分析器所做的移动如图4-16所示。输入指针指向输入栏中符号串最左边的符号。如果仔细观察语法分析器的动作，我们将会发现语法分析器跟踪的是输入的最左推导，即产生式输出的正好是最左推导中使用的那些产生式。已经扫描过的输入符号加上栈中的文法符号（从顶到底）构成该推导的左句型。 □

4.4.5 FIRST和FOLLOW

构造文法 G 的分析表需要两个与 G 有关的函数 FIRST 和 FOLLOW。我们可以用这两个函数来填写 G 的分析表的表项。由 FOLLOW 函数产生的记号集合还可用做紧急方式错误恢复期间的同步记号。

如果 α 是任意的文法符号串，则我们定义 FIRST(α)是从 α 推导出的串的开始符号的结终符集合，即 FIRST(α) = $\{a \mid \alpha \overset{*}{\Rightarrow} a\cdots, a$ 是终结符$\}$。如果$\alpha \overset{*}{\Rightarrow} \epsilon$，则$\epsilon$也属于FIRST($\alpha$)。

设 A 是一个非终结符，我们定义 FOLLOW(A)是包含所有在句型中紧跟在 A 后面的终结符 a 的集合，即 FOLLOW(A) = $\{a \mid S \overset{*}{\Rightarrow} \alpha Aa\beta$，$a$是终结符 $\}$。注意，在推导的某一时刻，在 A和 a 之间可能有符号，但如果是这样，它们将推导出ϵ并消失。如果 A 是某个句型的最右符号，那么$属于 FOLLOW($A$)。

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

图4-16 预测语法分析器在输入**id+id*id**上所做的移动

为了计算文法符号X的FIRST(X)，我们可以应用下列规则，直到没有终结符或ϵ可加到某个FIRST集合为止：

1. 如果 X 是终结符，则 FIRST(X) 是 $\{X\}$。
2. 如果 $X \to \epsilon$ 是一个产生式，则将ϵ加到 FIRST(X)中。
3. 如果 X 是非终结符，且 $X \to Y_1Y_2\cdots Y_k$ 是一个产生式，则

a) FIRST(Y_i) 中的所有符号在 FIRST(X)中。⊖

⊖ 此小点内容原书中未体现，为译者补充。——编者注。

b) 若对于某个 i，a 属于 FIRST(Y_i) 且 ϵ 属于 FIRST(Y_1)，…，FIRST(Y_{i-1})，即 $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$，则将 a 加入 FIRST(X) 中。

c) 若对于所有的 $j=1, 2, \cdots, k$，ϵ 在 FIRST(Y_j)中，则将 ϵ 加到 FIRST(X)中。

例如，FIRST(Y_1)中的每个元素确实都在FIRST(X)中。如果 Y_1 不能导出 ϵ，则不再往 FIRST(X)中增加新符号，如果 $Y_1 \stackrel{*}{\Rightarrow} \epsilon$，则将 FIRST($Y_2$)加到 FIRST($X$)中，依此类推。

现在，我们可以按如下方式计算任何串 $X_1X_2\cdots X_n$ 的开始符号集合 FIRST($X_1X_2\cdots X_n$)。将 FIRST(X_1) 中的所有非 ϵ 符号加到 FIRST ($X_1X_2\cdots X_n$)中。如果 ϵ 属于 FIRST(X_1)，还要加入 FIRST(X_2) 中的所有非 ϵ 符号，如果 ϵ 属于 FIRST (X_1)和 FIRST (X_2)，则加入 FIRST (X_3) 的所有非 ϵ 符号，依此类推。最后，如果 ϵ 属于所有 FIRST(X_i)，把 ϵ 加入 FIRST ($X_1X_2\cdots X_n$)。

为计算所有非终结符A的后继符号集合 FOLLOW(A)，我们可以应用如下规则，直到每个 FOLLOW 集合都不能再加入任何符号或 \$ 为止：

1. 将\$放入 FOLLOW(S) 中，其中 S 是开始符号，\$ 是输入串的结束符。
2. 如果存在产生式 $A \rightarrow \alpha B\beta$，则将FIRST($\beta$)中除 ϵ 以外的符号都放入 FOLLOW(B)中。
3. 如果存在产生式 $A \rightarrow \alpha B$，或 $A \rightarrow \alpha B\beta$，其中 FIRST($\beta$) 中包含 ϵ (即 $\beta \stackrel{*}{\Rightarrow} \epsilon$)，则将 FOLLOW($A$) 中的所有符号都放入 FOLLOW($B$) 中。

例4.17 我们把文法(4-11)重写如下：

$$
\begin{aligned}
&E \rightarrow TE' \\
&E' \rightarrow +TE' \mid \epsilon \\
&T \rightarrow FT' \\
&T' \rightarrow *FT' \mid \epsilon \\
&F \rightarrow (\,E\,) \mid \mathbf{id}
\end{aligned}
$$

187
≀
189

那么，

FIRST(E) = FIRST(T) = FIRST(F) = {(, **id**}.

FIRST(E') = {+, ϵ}

FIRST(T') = {*, ϵ}

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

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

FOLLOW(F) = {+, *,), \$}

例如，根据计算 FIRST 的规则1，FIRST(**id**) = {**id**}且 FIRST('(') = { (}。根据规则3与 $i=1$，**id** 和左括号被加入 FIRST(F)中。再根据规则3与 $i=1$，产生式 $T \rightarrow FT'$ 意味着 **id** 和左括号也在 FIRST(T) 中。另外，根据规则2，ϵ 属于 FIRST(E')。

为计算 FOLLOW 集，我们根据计算 FOLLOW 的规则1将\$放入 FOLLOW(E) 中。把规则2应用到产生式 $F \rightarrow (E)$，右括号也在 FOLLOW(E)中。把规则3应用到产生式 $E \rightarrow TE'$，\$和右括号在 FOLLOW(E') 中。因为$E' \stackrel{*}{\Rightarrow} \epsilon$，所以它们也在 FOLLOW($T$)中。作为 FOLLOW 计算规则应用的最后一个例子，根据规则2，产生式 $E \rightarrow TE'$ 意味着 FIRST(E') 中除 ϵ 以外的所有字符都应该放入 FOLLOW(T) 中。我们已经看到 \$ 在 FOLLOW(T) 中。 □

4.4.6 预测分析表的构造

下面的算法可以用于构造文法G的预测分析表。该算法的基本思想是：如果 $A \rightarrow \alpha$ 是产生

式且 a 在 FIRST(α)中，那么当前输入符号为 a 时，语法分析器将用 α 展开 A。惟一复杂的情况发生在 $\alpha = \epsilon$ 或者 $\alpha \stackrel{*}{\Rightarrow} \epsilon$ 时。在这种情况下，如果当前输入符号属于 FOLLOW(A)，或者如果输入已经到达 \$ 而 \$ 在 FOLLOW(A) 中，语法分析器仍将用 α 展开 A。

算法4.4 构造预测分析表。

输入：文法 G。

输出：分析表 M。

方法：

1. 对于文法中的每个产生式 $A \rightarrow \alpha$，执行第2和第3步。

2. 对 FIRST(α) 中的每个终结符 a，将 $A \rightarrow \alpha$ 加入到 $M[A, a]$中。

3. 若 ϵ 在 FIRST(α) 中，则对 FOLLOW(A) 的每个终结符 b，将 $A \rightarrow \alpha$ 加入到 $M[A, b]$ 中；若 ϵ 在 FIRST(α)中，且 \$ 在 FOLLOW(A)中，则将 $A \rightarrow \alpha$ 加入到 $M[A, \$]$ 中。

190 4. 将M中每个没定义的表项均置为 **error**。 □

例4.18 让我们把算法4.4用于文法(4-11)。因为 FIRST(TE') = FIRST(T) = {(, **id**}，产生式 $E \rightarrow TE'$ 使得 M[E, (] 和 M[E, **id**] 表项含有产生式 $E \rightarrow TE'$。

产生式 $E' \rightarrow +TE'$ 使得 M[E',+]表项含有 $E' \rightarrow +TE'$。因为 FOLLOW(E') = {), \$}，产生式 $E' \rightarrow \epsilon$ 使得$M[E',)]$和$M[E', \$]$表项含有$E' \rightarrow \epsilon$ 。

把算法4.4 应用于文法(4-11)后产生的分析表如图4-15所示。 □

4.4.7 LL(1)文法

算法4.4应用于任何文法 G 可以产生分析表 M。然而，对某些文法，M 可能含有多重定义的表项。例如，若 G 是左递归的或是二义性的，则 M 至少包含一个多重定义的表项。

例4.19 让我们再一次考虑例4.10中的文法(4-13)。为方便起见，将文法重写如下：

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow eS \mid \epsilon$$
$$E \rightarrow b$$

该文法的分析表如图4-17所示。

非终结符	输入符号					
	a	*b*	*e*	*i*	*t*	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

图4-17 文法(4-13)的分析表 M

因为 FOLLOW(S') = {e, \$}，所以 $M[S', e]$ 同时包含 $S' \rightarrow eS$ 和 $S' \rightarrow \epsilon$ 。该文法是具有二义性的。这种二义性非常明显，即遇见e (**else**) 时不知该选用哪个产生式。我们可以只选择 $S' \rightarrow eS$ 来解决这种二义性，这个选择刚好与 **else** 和最接近的 **then** 配对的原则相对应。注意，选择 $S' \rightarrow \epsilon$ 会使 e 不能压入栈或从输入中移掉，因此肯定是错误的。 □

分析表中没有多重定义表项的文法叫做LL(1)文法。LL(1)中的第一个L代表从左向右扫描
191 输入，第二个L代表产生最左推导，1代表在决定语法分析器的每步动作时向前扫描一个输入符。

可以证明，算法4.4可以为任何LL(1)文法 G 产生分析表。这个分析表能分析 G 的所有句子，而且只能分析 G 的句子。

LL(1)文法有一些特殊的性质。它不是二义性的，也不含左递归。可以证明，G 是LL(1)文法，当且仅当 G 的任何两个不同的产生式 $A\rightarrow\alpha \mid \beta$ 满足下面的条件：

1. 不存在这样的终结符 a，使得 α 和 β 导出的串都以 a 开始。

2. α 和 β 中至多有一个能导出空串。

3. 如果 $\beta \stackrel{*}{\Rightarrow} \epsilon$，那么 α 不能导出以 FOLLOW(A) 中的终结符开始的任何串。

很明显，(4-11) 的算术表达式文法是LL(1)文法。(4-13)中模拟 if-then-else 语句的文法不是 LL(1) 文法。

剩下的问题是，当分析表中含有多重定义的表项时应该怎么办？ 一种办法是进行文法变换，消除左递归和提取所有可能的左因子，以期产生的新文法的分析表中没有多重定义的表项。不幸的是，有些文法不论怎么变化也不能产生 LL(1) 文法。文法(4-13)就是这样的一个例子，根本没有 LL(1) 文法能产生它的语言。正如我们所看到的，让 $M[S', e] = \{S'\rightarrow eS\}$，那么仍可以用预测语法分析器对(4-13)进行分析。一般来说，没有一个普遍适用的规则可以用来删除多重定义的表项，使其成为单值而不影响语法分析器所识别的语言。

使用预测分析的主要困难在于为源语言编写一个能构造出预测语法分析器的文法。虽然消除左递归和提取左因子都非常简单，但它们使得结果文法很难阅读而且不易于翻译。为降低上述难度，编译器中语法分析器常常同时使用两种方法，即使用预测分析方法分析控制结构，使用算符优先分析方法分析表达式（将在4.6节中讨论）。然而，如果LR语法分析器（如4.9节中讨论的）的生成器可用，则可以自动获得预测分析和算符优先分析的一切优点。

4.4.8 预测分析的错误恢复

非递归预测语法分析器的栈使得语法分析器希望同剩余输入串进行匹配的终结符和非终结符变得十分清楚。因此，在下面的讨论中，我们将引用栈中符号。在预测分析过程中，如果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，a 是下一个输入符号，而 $M[A,a]$ 是空白表项，则检测出一个错误。

紧急方式错误恢复策略主要基于以下思想：跳过一些输入符号，直到期望的同步记号中的 192
一种出现为止。它的效果依赖于同步记号集合的选择。这个集合的选择应该使得语法分析器能迅速地从实际可能发生的错误中恢复过来。下面是一些启发式的方法：

1. 开始，我们可以把 FOLLOW(A) 中的所有符号放入非终结符 A 的同步记号集合中。如果出现错误时栈顶元素是 A，我们可以跳过一些记号，直到看见 FOLLOW(A) 中的元素，再把 A 弹出栈。这样我们就可以继续进行语法分析。

2. 仅使用 FOLLOW(A) 作为 A 的同步集合是不够的。例如，在C语言中分号用于表示语句结束，语句开始的关键字很可能不出现在产生表达式的非终结符号的 FOLLOW 集合中。因此赋值语句分号的遗漏会导致下一语句的开始关键字被跳过。语言的结构往往具有层次结构，如表达式出现在语句中，语句出现在程序块中，等等。我们可以把高层结构的开始符号加到低层结构的同步集合中。例如，可以把表示语句开始的关键字加入产生表达式的非终结符的同步集合中。

3. 如果把 FIRST(A) 的符号加入非终结符 A 的同步集合，那么我们可以恢复关于 A 的分析，只要 FIRST(A) 的符号出现在输入中。

4. 如果非终结符能产生空串，则可以将产生空串的产生式作为默认选择。这样做会延迟某

些错误的发现，但不会漏掉，而且这种方法可以减少错误恢复时要考虑的非终结符数。

5. 如果栈顶的终结符不能被匹配，那么简单的办法就是弹出该终结符，并给出提示信息，说明输入中插入了该符号，然后继续进行分析。实际上，这种方式等于把所有其他的记号作为该记号的同步集合。

例4.20 根据文法(4-11)分析表达式时，使用 FOLLOW 集和 FIRST 集中的符号作为同步记号是合情合理的。该文法的分析表（图4-15）重写为图4-18，并用 synch 来指示从非终结符的FOLLOW 集合中获得的同步记号。非终结符的 FOLLOW 集可以由例4.17获得。

图4-18所示表的使用过程如下：如果语法分析器发现表项 $M[A,a]$ 为空，则跳过输入符号 a；如果表项是 synch，则弹出栈顶的非终结符并试图恢复分析；如果栈顶的记号与输入符号不匹配，则从栈顶弹出该记号。

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

图4-18 加到图4-15的分析表中的同步记号

图4-18对应的语法分析器和错误恢复机制在面临错误的输入 **)id * + id** 时的行为如图4-19所示。 □

栈	输入	备注
$$E	) **id** * + **id** $	错误，跳过)
$$E	**id** * + **id** $	**id**在FIRST(E)中
$$E'T	**id** * + **id** $	
$$E'T'F	**id** * + **id** $	
$$E'T'**id**	**id** * + **id** $	
$$E'T'	* + **id** $	
$$E'T'F*	* + **id** $	
$$E'T'F	+ **id** $	错误，$M[F,+]$ = synch
$$E'T'	+ **id** $	F已经被弹出
$$E'	+ **id** $	
$$E'T+	+ **id** $	
$$E'T	**id** $	
$$E'T'F	**id** $	
$$E'T'**id**	**id** $	
$$E'T'	$	
$$E'	$	
$	$	

图4-19 预测语法分析器在语法分析和错误恢复过程中所做的移动

上面讨论的紧急方式恢复策略没有涉及出误信息这个重要问题。一般来说，出误信息必须由编译器的设计者提供。

通过在预测分析表的空白表项填上出错处理程序指针即可实现短语级恢复。这些程序可以修改、插入或删除输入符号并给出适当的出错信息。它们也可能弹出栈中的符号。如果允许替换栈顶符号或者将新的符号压入栈顶可能会出问题，因为这样会使得语法分析器执行的步骤跟语言中任何词的推导都不对应。不管怎样，我们必须确保不会产生无限循环。有一种很好的方法可以防止这种循环，即确认所有的恢复动作最终都将会使剩余输入串缩短（或者，如果到了输入的末端，使栈缩短）。

193 ~ 194

4.5 自底向上语法分析

本节介绍一种比较常用的自底向上分析法，称为移动归约分析法。4.6节将讨论一种最易于实现的移动归约分析法，称为算符优先分析法。更一般的移动归约分析方法叫做LR分析法，

将在4.7节讨论。LR分析法可用于许多自动的语法分析器的生成器。

移动归约分析法为输入串构造分析树时从叶节点（底端）开始，向根节点（顶端）前进。我们可以把该过程看成是把输入串 w “归约”成文法开始符号的过程。在每一步归约（reduction）中，如果一个子串和某个产生式的右部匹配，则用该产生式的左部符号代替该子串。如果每一步都能恰当地选择子串，我们就可以得到最右推导的逆过程。

例4.21 考虑文法

$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

句子 $abbcde$ 可按下述步骤归约到S：

$abbcde$
$aAbcde$
$aAde$
$aABe$
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 都可以匹配某产生式右部。选择最左边的 b，用 A 代替 b（因为A是产生式 $A \rightarrow b$ 的左部），得到 $aAbcd$。现在子串 Abc、b 和 d 都能匹配某产生式的右部。虽然 b 是匹配某产生式右部的最左子串，但我们用 A 代替子串 Abc（因为 A 是产生式 $A \rightarrow Abc$ 的左部），得到 $aAde$。然后，因为 B 是产生式 $B \rightarrow d$ 的左部，所以用 B 代替 d，得到 $aABe$，再用 S 代替此串。于是，这四步归约将 $abbcde$ 归约到 S。事实上，这些归约与如下最右推导的逆过程相对应：

$$S \underset{rm}{\Rightarrow} aABe \underset{rm}{\Rightarrow} aAde \underset{rm}{\Rightarrow} aAbcde \underset{rm}{\Rightarrow} abbcde$$

□ 195

4.5.1 句柄

非形式地，一个符号串的“句柄”是和一个产生式右部匹配的子串，而且把它归约到该产生式左部的非终结符代表了最右推导逆过程的一步。在很多情况下，匹配某个产生式 $A \rightarrow \beta$ 右部的最左输入子串 β 不是句柄，因为用这个产生式归约产生的串不能归约成开始符号。在例4.21中，如果在第二个串 $aAbcde$ 中用 A 代替 b，则得到 $aAAcde$，而 $aAAcde$ 不能归约到 S。因此，我们必须更精确地定义句柄。

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 $A \rightarrow \beta$ 以及 γ 的一个位置，在该位置可以找到串 β，而且用 A 代替 β 可以得到 γ 的最右推导的前一个右句型，即如果$S \underset{rm}{\overset{*}{\Rightarrow}} \alpha Aw \underset{rm}{\Rightarrow} \alpha\beta w$，那么紧跟在 α 后面的 $A \rightarrow \beta$ 是 $\alpha\beta w$ 的句柄。句柄右侧的串 w 只包含终结符。注意，如果文法是具有二义性的，则句柄不一定惟一，因为可能有不止一个 $\alpha\beta w$ 的最右推导。只有文法没有二义性时，它的每个右句型才有一个句柄。

在上面的例子中，$abbcde$ 是右句型，句柄是在位置2的 $A \rightarrow b$。同样，$aAbcde$ 也是右句型，它的句柄是在位置2的 $A \rightarrow Abc$。如果我们能清楚地知道 β 的位置和产生式 $A \rightarrow \beta$，有时也可以直接说“子串 β 是 $\alpha\beta w$ 的句柄”。

图4-20描述了右句型 $\alpha\beta w$ 的分析树中的句柄 $A \rightarrow \beta$。该句柄

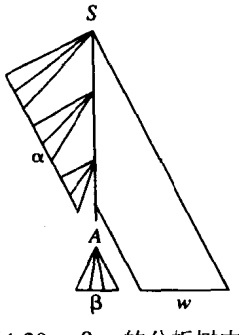


图4-20 $\alpha\beta w$ 的分析树中的句柄 $A \rightarrow \beta$

代表了由一个内节点和其所有子节点组成的最左完全子树。在图4-20中，A 是最底层的最左内节点，它的所有子节点都在该树中。在 $\alpha\beta w$ 中，把 β 归约到 A 可想像成“裁剪句柄”，即把 A 的子节点从分析树中删除。

例4.22 考虑文法

$$\begin{array}{ll}(1) & E \to E + E \\ (2) & E \to E * E \\ (3) & E \to (E) \\ (4) & E \to \mathbf{id}\end{array} \tag{4-16}$$

和最右推导

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

196

为方便起见，我们给 **id** 加以下标，并给每个右句型的句柄加上下划线。例如，$\mathbf{id}_1$ 是右句型 $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$ 的句炳，因为 **id** 是产生式 $E \to \mathbf{id}$ 的右部，且用 E 代替 $\mathbf{id}_1$ 产生前一个右句型 $E + \mathbf{id}_2 * \mathbf{id}_3$。注意，句柄右边的串中仅含终结符。

因为文法(4-16)是具有二义性的，存在 $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$ 的另一个最右推导：

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

考虑右句型 $E + E * \mathbf{id}_3$，在该推导中 $E + E$ 是 $E + E * \mathbf{id}_3$ 的句柄，而在上一个推导中 $\mathbf{id}_3$ 是该右句型的句柄。

本例中的两个最右推导类似于例4.6中的两个最左推导。第一个推导中 * 的优先级高于 +，第二个推导正好相反。 □

4.5.2 句柄裁剪

通过“裁剪句柄”可以得到最右推导的逆过程。我们从被分析的终结符串 w 开始。如果 w 是文法的一个句子，那么 $w = \gamma_n$，其中 γ_n 是下面的未知最右推导的第 n 步右句型：

196 ~ 197

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w.$$

为构造这个推导的逆过程，需要在 γ_n 中找到句柄 β_n，并用产生式 $A_n \to \beta_n$ 的左部代替 β_n，得到第 $n-1$步的右句型 γ_{n-1}。注意，目前我们还不知道如何找到句柄，但很快就会看到寻找句柄的方法。

重复此过程，即在 γ_{n-1} 中找到句柄 β_{n-1}，并对该句柄进行归约得到右句型 γ_{n-2}。如果继续此过程我们将得到只包含开始符号 S 的右句型，那么就宣告分析成功并停止。在归约过程中所用产生式序列的逆序就是输入串的最右推导。

例4.23 考虑例4.22的文法(4-16)和输入串 $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$。它的归约序列如图4-21所示。容

易看出，该右句型序列正好是例4.22中第一个最右推导序列的逆序。 □

右句型	句柄	归约产生式
$\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$	$\mathbf{id}_1$	$E \to \mathbf{id}$
$E + \mathbf{id}_2 * \mathbf{id}_3$	$\mathbf{id}_2$	$E \to \mathbf{id}$
$E + E * \mathbf{id}_3$	$\mathbf{id}_3$	$E \to \mathbf{id}$
$E + E * E$	$E * E$	$E \to E * E$
$E + E$	$E + E$	$E \to E + E$
E		

图4-21 由移动归约语法分析器产生的归约

4.5.3 用栈实现移动归约分析

如果使用上述裁剪句柄的分析方法，有两个问题必须解决。第一个问题是定位右句型中将要归约的子串；第二个问题是，如果这个子串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，首先看一下移动归约语法分析器所使用的数据结构。

实现移动归约分析的一种简单方法是用栈来保存文法符号，用输入缓冲区来保存要分析的串 w，用 \$ 来标记栈底，也用它标记输入串的右端。初始，栈是空的，串 w 在输入缓冲区中，如下所示：

栈	输入
\$	w \$

语法分析器将零个或多个输入符号压入栈，直到句柄 β 在栈顶出现为止，语法分析器再把 β 归约成某个恰当的产生式的左部。语法分析器重复此过程，直到它发现错误或者栈中只含有开始符号并且输入串为空： 198

栈	输入
\$$S$	\$

进入这个格局以后，语法分析器停止并宣告分析成功。

例4.24 让我们逐步看一下移动归约语法分析器在分析输入串 $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$ 时的动作，文法是(4-16)，使用例4.22的第一种推导。动作序列如图4-22所示。注意，由于文法(4-16)对该输入有两种最右推导，所以语法分析器还可能采取另一个动作序列。 □

	栈	输入	动作
(1)	\$	$\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$\$	移动
(2)	\$$\mathbf{id}_1$	$+ \mathbf{id}_2 * \mathbf{id}_3$\$	用$E \to \mathbf{id}$归约
(3)	\$$E$	$+ \mathbf{id}_2 * \mathbf{id}_3$\$	移动
(4)	\$$E +$	$\mathbf{id}_2 * \mathbf{id}_3$\$	移动
(5)	\$$E + \mathbf{id}_2$	$* \mathbf{id}_3$\$	用$E \to \mathbf{id}$归约
(6)	\$$E + E$	$* \mathbf{id}_3$\$	移动
(7)	\$$E + E *$	$\mathbf{id}_3$\$	移动
(8)	\$$E + E * \mathbf{id}_3$	\$	用$E \to \mathbf{id}$归约
(9)	\$$E + E * E$	\$	用$E \to E * E$归约
(10)	\$$E + E$	\$	用$E \to E + E$归约
(11)	\$$E$	\$	接受

图4-22 移动归约语法分析器在输入 $\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$ 上的格局

移动归约语法分析器的基本动作是移动和归约，但实际上有四种可能的动作：移动、归约、

接受、出错。

1. 移动：把下一个输入符号移动到栈顶。

2. 归约：语法分析器知道句柄的右端已在栈顶。它必须在栈中确定句柄的左端，并选择正确的非终结符替代句柄。

3. 接受：语法分析器宣告分析成功。

199 4. 出错：语法分析器发现了一个语法错误，并调用错误恢复程序进行错误处理。

有一个重要的事实说明在移动归约分析中使用栈是有道理的：句柄最终总会出现在栈顶，而不是在栈的里面。如果我们考察任意最右推导中连续两步的可能形式，这个事实将是显而易见的。这两步的可能形式是：

(1) $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha Az \overset{*}{\underset{rm}{\Rightarrow}} \alpha\beta Byz \overset{*}{\underset{rm}{\Rightarrow}} \alpha\beta\gamma yz$

(2) $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha BxAz \overset{*}{\underset{rm}{\Rightarrow}} \alpha Bxyz \overset{*}{\underset{rm}{\Rightarrow}} \alpha\gamma xyz$

在(1)中，A 被 βBy 代替，然后最右边的非终结符 B 由 γ 代替。在(2)中，A 仍然先被替换，但这次只是由终结符串 y 来代替。下一个最右非终结符 B 将会在 y 左边的某个地方出现。

让我们反过来看一下(1)，移动归约语法分析器已经到达下面的格局：

栈	输入
$\$\alpha\beta\gamma$	$yz\$$

语法分析器把句柄 γ 归约成 B，到达下面的格局：

栈	输入
$\$\alpha\beta B$	$yz\$$

因为 B 是 $\alpha\beta Byz$ 的最右非终结符，$\alpha\beta Byz$ 的句柄的右端不可能出现在栈的里面。因此语法分析器把 y 移入到栈中，进入下面的格局：

栈	输入
$\$\alpha\beta By$	$z\$$

其中，βBy 是句柄，它被归约成 A。

让我们再反过来看一下(2)，移动归约语法分析器到达下面的格局：

栈	输入
$\$\alpha\gamma$	$xyz\$$

句柄 γ 在栈顶。语法分析器把 γ 归约成 B 以后，可以移入 xy 以得到在栈顶的下一个句柄 y：

栈	输入
$\$\alpha Bxy$	$z\$$

现在语法分析器可以把 y 归约成 A。

在这两种情况下，在执行了一步归约之后，语法分析器都必须移动零个或多个符号以便使下一个句柄进栈。语法分析器不需要深入到栈中去寻找句柄。由此可知，使用栈来实现移动归约语法分析器是非常有效的。我们还必须说明怎样选取动作才能使移动归约语法分析器正常工
200 作。下边讨论的算符优先语法分析器和LR语法分析器就是两种这样的技术。

4.5.4 活前缀

出现在移动归约语法分析器栈中的右句型的前缀集合称为*活前缀*。等价的定义为：活前缀是右句型的前缀，而且其右端不会超过该句型的最右边句柄的末端。根据该定义，我们可以把终结符加到活前缀的末尾得到右句型。因此，在给定时刻只要输入串中已分析过的那部分能归约成活前缀，就没有错误。

4.5.5 移动归约分析过程中的冲突

有些上下文无关文法不能使用移动归约分析方法进行分析。这种文法的每一个移动归约语法分析器会形成这样的格局：根据栈中的内容和下一个输入符号不能决定是移动还是归约（*移动-归约冲突*），或不能决定按哪一个产生式进行归约（*归约-归约冲突*）。下面我们会给出属于这类文法的语法结构的例子。从技术上讲，这些文法不属于4.7节定义的LR(k)类文法。我们称它们为非LR文法。LR(k)中的 k 代表超前搜索 k 个输入符号。编译中使用的文法通常都属于LR(1)类，即超前搜索一个输入符号。

例4.25 二义性文法一定不是LR文法。例如，考虑4.3节的文法(4-7)：

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

如果移动归约语法分析器处于格局

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

无论栈中 **if** *expr* **then** *stmt* 下面是什么符号，我们都不能判断 **if** *expr* **then** *stmt* 是否为句柄，这里存在移动-归约冲突。根据输入中 **else** 后的内容，也许将 **if** *expr* **then** *stmt* 归约成 *stmt* 是正确的，或者也许移动 **else** 进栈，然后寻找另一个 *stmt* 来完成 **if** *expr* **then** *stmt* **else** *stmt* 的替换是正确的。因此，在这种情况下我们不能判断应该移动还是归约，所以这个文法不是LR(1)文法。一般地说，任何二义性文法都不是LR(k)文法（对任何k）。

但是，我们必须指出，对移动归约分析方法进行简单的改造即可分析某些二义性文法，如上面的 if-then-else 文法。当我们为包含上面产生式的文法构造这样的语法分析器时，存在移动-归约冲突：对于 **else** 或者移动，或者用 *stmt*→**if** *expr* **then** *stmt* 归约。如果我们通过优先使 201
用移动来解决这个冲突，语法分析器就能正常工作了。4.8节将讨论这种二义性文法的语法分析器。 □

非 LR 出现的另一种常见的情况是：知道了句柄，但根据栈里的内容和下一个输入符号不足以判断在归约中使用哪个产生式。下面的例子说明了这种情况。

例4.26 假设词法分析器对任何标识符都返回记号 **id**，而不管它是如何使用的。假设语言通过给出过程的名字及用括号括起来的参数表来调用过程，而且数组元素的引用也采用同样的语法。因为数组引用中的下标和过程调用中的参数的翻译是不一样的，所以，我们用不同的产生式来产生实参表和下标表。我们的文法应该具有下面一些产生式：

(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*

以A(I,J)开始的语句可能以记号流 **id(id, id)** 的形式呈现给语法分析器。把前三个记号移进栈后，移动归约语法分析器的格局是：

栈	输入
· · · **id (id**	**, id)** · · ·

显然，必须对栈顶的 **id** 进行归约，但是按哪个产生式归约呢？正确的选择是：如果 *A* 是过程，应按产生式(5)归约；如果 *A* 是数组，应按产生式(7)归约。但栈中的信息不能告诉我们应按哪个产生式归约，必须使用符号表中有关 *A* 的信息。

解决上述问题的一种办法是把产生式(1)的记号 **id** 改为 **procid**，并且使用更复杂一些的词法分析器，当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要访问符号表。

这样修改后，处理 A(i, j) 时，语法分析器可能处于格局

栈	输入
· · · **procid (id**	**, id)** · · ·

202

或处于前面的格局。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的产生式，虽然它本身并不包含在这个归约中。移动归约分析可以利用栈深处的信息来指导分析。 □

4.6 算符优先分析法

LR文法是一大类适合移动归约语法分析器的文法。LR文法将在4.7节中讨论详细讨论。然而，对于一小部分非常重要的文法，我们可以很容易地手工构造有效的移动归约语法分析器。这些文法具有下面的性质：所有产生式的右部都不是 ϵ 或两个相邻的非终结符。具有第二个性质的文法称为*算符文法*。

例4.27 下面是表达式的文法：

$$E \rightarrow EAE \mid (E) \mid -E \mid \mathbf{id}$$
$$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$$

它不是算符文法，因为右部 *EAE* 有2个（实际上是3个）连续的非终结符。然而，如果我们用 *A* 的每个候选式替代第一个产生式中的 *A*，将得到下面的算符文法：

$$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid \mathbf{id} \tag{4-17}$$

现在我们给出一种易于实现的语法分析技术，即算符优先分析。历史上，该技术最初是一种处理记号的技术，而这些记号不是建立在文法的基础上。实际上，一旦我们根据一个文法构造了一种算符优先语法分析器，我们就可以忽略该文法，栈中的非终结符仅仅作为与这些非终结符相关的属性的占位符。

作为通用的语法分析技术，算符优先分析法具有许多缺点。例如，它很难处理像减号这样的记号，因为减号有两个不同的优先级（取决于它是一元操作符的还是二元操作符的）。更糟

糕的是，因为要分析的语言的文法和算符优先语法分析器本身的关系不是十分紧密的，所以不能肯定语法分析器接受的就是所期望的语言。

然而，由于算符优先分析技术比较简单，很多编译器的语法分析器经常采用算符优先分析技术对表达式进行分析，对于语句和高级结构的分析则采用4.4节中描述的递归下降分析法。有些编译器的语法分析器甚至对整个语言都采用算符优先技术进行语法分析。

在算符优先分析中，我们在某些终结符对之间定义如下三种优先关系：<·、≐和·>。这些优先关系可用于指导句柄的选取，它们的含义如右表所示。

关 系	含 义
$a <\cdot b$	a的优先级低于b
$a \doteq b$	a的优先级等于b
$a \cdot> b$	a的优先级高于b

注意，这些关系类似于算术关系“小于”、“等于”和“大于”，但优先关系具有不同的性质。例如，对于同一语言，$a <\cdot b$ 和 $a \cdot> b$ 可能同时成立，或者对于某两个终结符 a 和 b，$a <\cdot b$ 和 $a \doteq b$ 以及 $a \cdot> b$ 均不成立。

有两种常用的方法确定终结符之间的优先关系。第一种方法是直觉主义方法，即基于习惯的操作符间的结合规则和运算优先关系。例如，如果 * 比 + 具有更高的优先级，我们可以令 + <·* 和 *·>+ 成立。在解决文法(4-17)的二义性时将看到这种方法，并且我们可以使用这种方法写出文法(4-17) 的算符优先语法分析器（虽然一元减会有问题）。

确定算符优先关系的第二种方法是首先为语言构造无二义性文法。该文法在分析树上反映了正确的结合规则和优先级。对于表达式来说，这项工作不难，2.2节中表达式的语法提供了这种实例。对于二义性的其他来源，如 **else** 的不匹配，文法(4-9)是一个有用的模型。如果已经获得了无二义性文法，则存在一种从该文法构造算符优先关系的机械方法。这些关系可能是相交的，而且它们可能分析非该文法产生的语言，但是对于标准算术表达式来说，实际上很少遇到问题。这里将不讨论优先级的构造方法，请参见 Aho and Ullman[1972b]。

4.6.1 使用算符优先关系

优先关系主要用于界定右句型的句柄。<· 标记句柄的左端，≐出现在句柄的内部，·> 标记句柄的右端。假设我们有一个算符文法的右句型，那么“没有相邻的非终结符出现在产生式右部”意味着“右句型不会有两个相邻的非终结符”。于是，我们可以用 $\beta_0 a_1 \beta_1 \cdots a_n \beta_n$ 表示右句型，这里β_i是ϵ（空串）或单个非终结符，a_i是单个终结符。

假设 a_i 和 a_{i+1}之间只满足 <·、≐或 ·> 中的某一种优先关系，用 $ 标记符号串的两端，对所有的终结符 b，定义 $ <·b 和 b ·> $。现在假设我们从符号串移走非终结符，并在每两个终结符之间以及两端的终结符与 $ 之间放上正确的优先关系 <·、≐ 或 ·>，其中 $ 标记符号串的两端。例如，假设开始时有右句型 **id** + *id* * **id**和图4-23所给出的优先关系，这些关系是根据文法(4-17)进行语法分析所确定的关系集合的子集。然后，插入优先关系后的符号串为：

203 ~ 204

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

图4-23 算符优先关系

$$\$ <\cdot \textbf{id} \cdot> + <\cdot \textbf{id} \cdot> * <\cdot \textbf{id} \cdot> \$ \tag{4-18}$$

例如，由于 <· 是优先关系表中行为$列为 **id** 的表项，所以<· 被插入在左端的 $ 和 **id** 之间。应用下面的过程即可发现句柄：

1. 从左端开始扫描串，直到遇到第一个 ·> 为止。在上面的(4-18)中，它出现在第一个**id**和+之间。

2. 向左扫描，跳过所有的 $\doteq$，直到遇到一个 $<\cdot$ 为止。在(4-18)中，我们向左扫描到$。

3. 句柄包括从第2步遇到的 $<\cdot$ 的右部到第一个 $\cdot>$ 的左部之间的所有符号，包括介于其间或者两边的非终结符。包括两边的非终结符是必要的，可以使两个相邻的非终结符不会出现在右句型中。在(4-18)中，句柄是第一个 **id**。

如果我们正在处理文法(4-17)，就把 **id** 归约为 E。这时，我们可以得到右句型 E + **id** * **id**。按照同样的步骤将剩余的两个 **id** 归约为 E 之后，可以得到右句型 $E + E*E$。现在考虑删除非终结符后的符号串 $ + * $。插入优先关系后，我们得到

$$\$ <\cdot + <\cdot * \cdot> \$$$

可以看出，句柄的左端位于 + 和 * 之间，右端位于 * 和 $ 之间。这些优先关系表明，在右句型 $E + E*E$ 中，句柄是 $E*E$。注意 * 两边的 E 也在句柄中。

205 因为非终结符不会影响语法分析，所以我们不需要考虑区分它们。我们可以把一个标记“非终结符”保存在移动归约栈中，作为属性值的占位符。

从上面的讨论可以看出，为了找到句柄，每步都必须扫描整个右句型。如果我们用栈存储已看到的输入符号，并且用优先关系指导移动归约语法分析器的动作，情况就不同了。如果栈顶的终结符和下一个输入符之间的优先关系是 $<\cdot$ 或 $\doteq$，则语法分析器移动，表示还没有发现句柄的右端；如果是 $\cdot>$ 关系，就调用归约。这时语法分析器已找到句柄的右端，而且优先关系还能用来在栈中找到句柄的左端。

如果一对终结符之间没有优先关系（由图4-23中的空白表项指出）成立，则表示检测出了语法错误，须调用错误恢复例程。下面的算法可以形式化描述上述思想。

算法4.5 算符优先分析算法。

输入：输入字符串 w 和优先关系表。

输出：如果 w 是一个句子，则输出一个分析树架子，它的所有内节点均由占位非终结符 E来标注；否则，指出错误。

方法：初始时，栈中放入 $，输入缓冲区放入 $w$$。为了进行语

206 法分析，执行图4-24的程序。 □

```
(1)  令 ip 指向 w$ 的第一个符号;
(2)  repeat forever
(3)      if $在栈顶而且 ip 指向$ then
(4)          return
         else begin
(5)          令a 是栈中最上面的终结符，而b 是 ip 所指向的符号;
(6)          if a <· b 或 a ≐ b then begin
(7)              将b 压入栈中;
(8)              ip 指向下一个输入符号;
             end;
(9)          else if a ·> b then          /* 归约 */
(10)             repeat
(11)                 从栈中弹出符号
(12)             until 栈顶终结符与最近弹出的终结符之间满足 <·
(13)         else error()
         end
```

图4-24 算符优先分析算法

4.6.2 从结合律和优先级获得算符优先关系

我们可以以合适的方式创建算符优先关系，并希望由它们指导的算符优先分析算法能运行正常。类似于文法(4-17)产生的算术表达式语言，我们可以用下面的启发式方法产生正确的优先关系。注意，文法(4-17)是二义性的，右句型可能有多个句柄。我们的设计规则是选择恰当的句柄来反映二元操作符的结合律和优先级。

1. 如果操作符 θ_1 比 θ_2 具有更高的优先级，则 $\theta_1 \cdot> \theta_2$ 和 $\theta_2 <\cdot \theta_1$ 都成立。例如，如果 * 比 + 具有更高的优先级，那么，$*\cdot>+$ 和 $+<\cdot*$ 都成立。这些关系可以确保在形如 $E + E * E + E$ 的表达式中，中间的 $E * E$ 是先被归约的句柄。

2. 如果 θ_1 和 θ_2 是同优先级的操作符（可以是同一个操作符），那么当它们满足左结律时，$\theta_1 \cdot> \theta_2$ 和 $\theta_2 \cdot> \theta_1$ 成立；当它们满足右结合律时，$\theta_1 <\cdot \theta_2$ 和 $\theta_2 <\cdot \theta_1$ 成立。例如，+ 和 − 满足左结合律，+·>+、+·>−、−·>−、−·>+ 同时成立。↑ 满足右结合律，↑<·↑ 成立。这些关系可以确保在 E–E+E 中，E–E 会被选为句柄；而在 E↑E↑E中，最后的E↑E会被选为句柄。

3. 对于所有的操作符θ来说，θ <· **id**、**id** ·> θ、θ <· (、(<·θ、) ·> θ、θ·>)，θ ·> $、$<·θ均成立。(≐)、$<·(、$<·**id**、(<·(、**id**·>$、)·>$、(<·**id**、**id**·>)、)·>) 也都成立。

这些规则可以确保 **id** 和 (*E*) 都会被归约为 *E*，而且只要 $ 作为左右两端的标记，就可能会促使在 $ 之间找到句柄。

例4.28 基于以下假设，图4-25给出了文法(4-17)的算符优先关系（空白表示错误项）：

1. ↑ 具有最高优先级，满足右结合律。
2. * 和 / 具有次最高优先级，满足左结合律。
3. + 和 − 的优先级最低，满足左结合律。

读者应当试着查看一下该表是否能够正常工作，此处忽略了一元减号的问题。例如，当输入为 **id** * (**id** ↑ **id**) − **id** / **id** 时，试查阅一下图4-25所示的这张表。 □ 207

	+	−	*	/	↑	**id**	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
−	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	≐	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

图4-25 算符优先关系

4.6.3 处理一元操作符

如果我们有一个类似于 ¬（逻辑非）的操作符，并且不是二元操作符，那么我们可以把它合并到上面构造算符优先关系的方案中。假设 ¬ 是一元前缀操作符，对于任何操作符 θ，无论是一元的还是二元的，均使 θ <· ¬ 成立。如果 ¬ 比 θ 的优先级高，则使 ¬·>θ 成立，否则 ¬<·θ 成立。例如，设 ¬ 比 & 优先级高，且&满足左结合律，通过这些规则可以看出，E&¬E&E 等价于 (E&(¬E))&E。一元后缀操作符的规则类似。

对于类似于减号这样的操作符（它既是一元前缀操作符又是二元中缀操作符）情况就不同了。即使给一元和二元的减号赋予同样的优先级，图4-25中的表也不能正确分析类似 **id***−**id** 这样的输入。在这种情况下，最好的办法是用词法分析器来区分一元和二元减号，当它看到一元的减号时返回不同的记号。遗憾的是，词法分析器不能通过超前扫描来区分这两者，它必须记住前一个记号。例如，在 Fortran 中，如果下一个记号是操作符、左括号、逗号或赋值符号，则减号是一元的。

4.6.4 优先函数

使用算符优先语法分析器的编译器不需要存储优先关系表。在大多数情况下，该表可以用两个*优先函数* f 和 g 来表示。f 和 g 把终结符映射为整数。对于符号 a 和 b，选择 f 和 g 以满足：

1. $f(a) < g(b)$，如果$a <\cdot b$。

2. $f(a) = g$ (b)，如果$a \doteq b$。

3. $f(a) > g(b)$，如果$a \cdot\!> b$。

208 这样 a 和 b 之间的优先关系就可以通过 $f(a)$ 和 $g(b)$ 之间的数值比较来确定。但是，优先矩阵中的空白项是模糊的，因为无论 $f(a)$和 $g(b)$ 取何值，上面的1、2或3总有一项成立。错误检测能力的损失还没有严重到阻止使用优先函数的程度。如果在调用归约时没有发现句柄，仍然会发现错误。

注意，并不是所有的优先关系表都能用优先函数来表示；但是，在实际应用中，优先函数通常是存在的。

例4.29 图4-25的优先关系表对应下面的优先函数：

	+	−	*	/	↑	(	)	id	$
f	2	2	4	4	4	0	6	6	0
g	1	1	3	3	5	5	0	5	0

例如，$* \lessdot$ **id**，所以 $f(*) < g$ (**id**)。注意，虽然 f (**id**) $> g$ (**id**)意味着 **id** $\cdot\!>$ **id**，事实上，**id** 和 **id** 之间并没有优先关系。类似地，图4-25中的其他空白项都被某种优先关系所代替。 □

如果优先函数存在的话，下面将给出一种构造某个算法优先表的优先函数的简单算法。

算法4.6 优先函数的构造。

输入：算符优先表。

输出：表示输入矩阵的优先函数，或指出它不存在。

方法：

1. 为每个终结符 a 或 $ 创建符号 f_a 和 g_a。

2. 把生成的符号按下面的方式分成尽可能多的组：如果 $a \doteq b$，则 f_a 和 g_b 在同一组。注意，即使某些符号之间没有 $\doteq$ 关系，我们也可能不得不把它们放在同一组。例如，如果$a \doteq b$且$c \doteq b$，那么 f_a 和 f_c 必在同一组，因为它们都和 g_b 在同一组。此外，如果 $c \doteq d$，那么即使 $a \doteq d$ 不成立，f_a 和 g_d 也可能在同一组。

3. 构造一个以(2)中的符号组为节点的有向图。对任意的 a 和 b，如果 $a \lessdot b$，则从 g_b 所在的组引出一条边指向 f_a 所在的组。如果 $a \cdot\!> b$，则从 f_a 所在的组引一条边指向 g_b 所在组。注意从 f_a 到 g_b 的边或路径意味着 $f(a)$必超过 $g(b)$，从 g_b 到 f_a 的路意味着 $g(b)$必超过 $f(a)$。

209 4. 如果(3)中构造的图中有环路，则不存在优先函数。如果没有环路，则将 $f(a)$ 设为从 f_a 所在的组开始的最长路径的长度，$g(a)$ 设为从 g_a 所在的组开始的最长路径的长度。 □

例4.30 考虑图4-23所示的优先关系矩阵。由于没有 $\doteq$ 关系，所以每个符号所在的组中只有它自身。图4-26是用算法4.6构造的图。

由于图中没有环路，所以存在优先函数。由于没有从 $f_\$$ 和 $g_\$$ 出发的边，所以 $f(\$) = g(\$) = 0$。由于从 g_+ 出发的最长的路径长度是1，所以 $g(+) = 1$；由于从 g_{id} 出发的最长路径是从 g_{id} 到f_* 到 g_* 到 f_+ 到 g_+ 再到 $f_\$$，所以 g (**id**) = 5。结果，得到下面的优先函数：

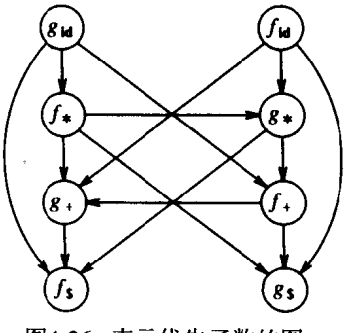


图4-26 表示优先函数的图

	+	*	id	$
f	2	4	4	0
g	1	3	5	0

□

4.6.5 算符优先分析中的错误恢复

算符优先分析器在语法分析的过程中，能发现以下两种情况下的语法错误：

1. 如果栈顶的终结符和当前输入之间没有优先关系。[⊖]
2. 如果发现句柄，但句柄不是任何产生式的右部。

回想一下，算符优先分析算法（算法4.5）似乎只归约由终结符组成的句柄。然而，虽然 210
非终结符是匿名处理的，但它们仍在分析栈中占据位置。因此，当在我们上面的(2)中讨论匹配产生式右部的句柄时，意味着如果终结符是这样，被非终结符占据的位置也是这样。

容易看出，除了上面的(1)和(2)，其他情况下都不能检测到错误。为找到句柄的左端，按图4-24的(10)~(12)步向下扫描栈时，一定能找到$<\cdot$，因为标记栈底的$和栈中任何紧挨着它的符号之间都是$<\cdot$关系。注意，如果图4-24中的符号之间没有$<\cdot$或$\doteq$关系，则不允许它们在栈中相邻。这样，(10)~(12)步一定能成功地完成归约。

然而，不能因为我们在栈中找到了一个符号序列 $a<\cdot b_1 \doteq b_2 \doteq \cdots \doteq b_k$，就认为$b_1b_2\cdots b_k$是某个产生式右部的终结符串。我们在图4-24中没有检查该条件，但是显然可以这样做。事实上，如果希望将语义规则和归约相关联，就必须这么做。修改 (10)~(12)步以确定归约时哪个产生式是句柄，我们就有机会用图4-24的程序检测到错误。

4.6.5.1 归约时的错误处理

我们可以把错误检测和恢复程序分成几部分。一部分解决类型(2)的错误。例如，该程序可以像图4-24的步骤 (10)~(12) 那样，把符号弹出栈。然而，因为没有用于归约的产生式，所以没有语义动作，而只是打印诊断信息。为了确定打印什么诊断信息，处理情况(2)的程序必须确定被弹出的字符串与哪个产生式的右部相像。例如，假设 *abc* 被弹出，并且没有哪个产生式的右部由 *a*、*b*、*c* 和零个或多个非终结符组成，那么也许我们可以考虑是否删除 *a*、*b*和*c* 中的一个就可以产生一个合法的产生式右部（忽略非终结符）。例如，如果有一个右部为 *aEcE*，则可以给出下面的诊断信息：

illegal *b* **on line** （包含*b*的行）

我们也可以考虑改变或插入一个终结符。如果 *abEdc* 是一个产生式的右部，则可以给出下面的诊断信息：

missing *d* **on line** （包含*c*的行）

我们也可能发现对于某个右部，终结符的顺序正确，但非终结符的模式不对。例如，*abc* 被弹出栈并且在它的中间和两边均没有非终结符，而且 *abc* 不是某产生式的右部，而 *aEbc* 是，则可以给出下面的诊断信息： 211

missing *E* **on line** （包含*b*的行）

这里的 *E* 代表由非终结符*E*所表示的一个适当的语法类型。例如，如果 *a*、*b* 或 *c* 是操作符，则可能代表“表达式”；如果 *a* 是类似 **if** 这样的关键字，则可能代表“条件”。

⊖ 在使用优先函数表示优先关系表的编译器中，这种错误源的检测可能不可用。

一般地，当没有发现的句柄不是任何产生式的右部时，要给出合适的诊断信息，其困难在于图4-24的(10)~(12)步是否弹出有限或无限数目的字符串。图4-24的 (10)~(12) 弹出的任何字符串 $b_1b_2\cdots b_k$ 在相邻的符号间必须满足 $\doteq$ 关系，即 $b_1 \doteq b_2 \doteq \cdots \doteq b_k$。如果算符优先表告诉我们仅由有限数目的终结符序列满足 $\doteq$ 关系，则可以用枚举法处理这些串。对于每个这样的串 x，我们可以事先确定一个具有最小距离的字符串 y，y 是某个产生式的右部，并给出诊断信息来暗示希望找到 y 时却找到了 x。

确定图4-24的(10)~(12)步能够弹出的所有字符串是很容易的，其方法是构造一个有向图，图的节点表示终结符，当且仅当$a \doteq b$时，从 a 到 b 有一条边。沿着图上各路径的节点标记所形成的字符串就是图4-24的(10)~(12)步能够弹出的字符串。路径可能包含单个节点。然而，对某个输入，为了能够弹出 $b_1b_2\cdots b_k$，必须有一个符号a (可能是\$) 使得 $a \lessdot b_1$，b_1 称为起点；同时也必须有一个符号 c（可能是\$）使得 $b_k \gtrdot c$，b_k称为终点。只有这时才能调用归约，使 $b_1b_2\cdots b_k$ 成为被弹出的符号序列。如果从起点到终点的路径中有环路存在，则有可能弹出无限个串，否则只能弹出有限个串。

例4.31 让我们重新考虑文法(4-17)：

$$E \to E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid \mathbf{id}$$

该文法的优先矩阵如图4-25所示，其有向图如图4-27所示。图中只有一条边，因为只有左右括号间才存在 $\doteq$ 关系。除了右括号以外所有的节点都是起点，除了左括号以外所有的节点都是终点。于是，从起点到终点长度为1的路径是 +，-，*，/，**id** 以及 ↑，长度为2的路径是从（到）的路径。路径的数目有限，而且每个都与文法中某个产生式右部的终结符串相对应。于是，归约的错误检查程序只需检查出现在被归约的终结符串间的非终结符标记的真子集。检查程序进行如下检查：

1. 如果 +、-、*、/ 或 ↑ 被归约，则检查两边是否有非终结符出现。如果没有，输出诊断信息“missing operand”。

2. 如果 **id** 被归约，检查两边是否有非终结符出现。如果有，输出警告“missing operator”。

3. 如果 () 被归约，检查两个括号中间是否有非终结符出现。如果没有，输出警告“no expression between parentheses”。

除此之外，还要检查是否有无非终结符出现在括号对的任何一边，如果是，给出和(2)一样的诊断信息。 □

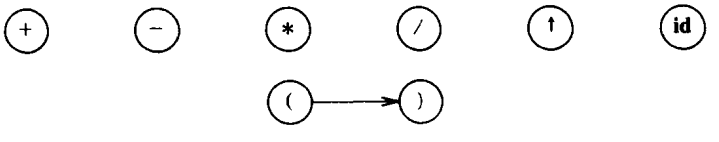


图4-27 图4-25中的优先矩阵所对应的图

如果图4-24的(10)~(12)能够弹出无限个字符串，则错误信息不能用枚举法列出。我们可以使用通用程序来确定是否有某个产生式的右部与弹出的字符串接近（即距离是1或2，此处的距离根据插入、删除或修改的记号数而不是字符数来计算的）。如果有，则在“这个产生式是欲寻找的产生式”的假设下，给出一个特定诊断信息。如果没有产生式的右部与弹出的字符串接近，则给出一条一般的诊断信息“当前行有错误”。

4.6.5.2 处理移动归约错误

下面我们讨论算符优先语法分析器检测错误的其他方法。如果根据优先矩阵决定符号移动还是归约（图4-24的第6行和第9行），有时会发现栈顶符号和第一个输入符号间没有优先级关系。例如，假设 a 和 b 是两个相临栈顶符号，b 在栈顶，c 和 d 是接下来的两个输入符号，且 b 和 c 之间没有优先关系。为恢复错误，必须修改栈、输入或者两者都修改。我们可以改变符号、在输入或栈中插入符号或者从输入或栈中删除符号。如果改变或插入符号，必须小心，以免陷入死循环。例如，在输入的前端不断地插入符号而不能归约或移动任何插入的符号，就可能会造成死循环。

一种不会陷入死循环的方法是确保在恢复后能够把当前输入符号移动进栈（如果当前符号是$，确保不会往输入中插入符号，且栈最终会被缩短）。例如，假设 ab 在栈里，cd 是输入，如果 $a \lessdot \cdot c^2$（$\lessdot \cdot$表示“$<\cdot$或$\doteq$”），我们把b从栈里弹出。另一种选择方案是：如果 $b \lessdot \cdot d$，从输入中删除 c。第三种选择方案是找到某个符号 e，使得 $b \lessdot \cdot e \lessdot \cdot c$，在输入字符串中把 e 插到 c 的前面。更一般地，如果找不到单个符号插入，我们可以插入符合如下条件的一串符号：

212
~
213

$$b \lessdot \cdot e_1 \lessdot \cdot e_2 \lessdot \cdots \lessdot \cdot e_n \lessdot \cdot c$$

总之，所选择的方案应该能够反映出编译器设计者面对每种可能出现的错误的直觉。

对于优先矩阵中的每个空白项，我们必须指定一个错误恢复程序，而且同一程序可用在多个地方。这样，当语法分析器在图4-24的步骤(6)发现 a 和 b 之间没有优先关系时，就可以找到指向相应错误恢复程序的指针。

例4.32 再次考虑图4-25的优先矩阵。在图4-28中，我们给出了具有一个或多个空白项的行和列，并将错误处理程序的名字填在空白处。

	id	(	)	$
id	e3	e3	$\cdot>$	$\cdot>$
(	$<\cdot$	$<\cdot$	$\doteq$	e4
)	e3	e3	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	e2	e1

图4-28 带有错误项的算符优先矩阵

这些出错处理程序的主旨如下：

- e1：/* 缺少整个表达式时调用 */
  把 **id** 插入到输入字符串；
  输出诊断信息“missing operand”。
- e2：/* 表达式以右括号开始时调用 */
  从输入中删掉）；
  输出诊断信息“unbalanced right parenthesis”。
- e3：/* **id** 或)后面跟随 **id** 或(时调用 */
  把 + 插入到输入字符串；
  输出诊断信息“missing operator”。
- e4：/* 表达式以左括号结束时调用 */
  从栈中弹出(；
  输出诊断信息“missing right parenthesis”。

下面让我们看一看错误处理机制如何处理错误输入字符串 **id** +)。语法分析器的第一个动作是把 **id** 移动进栈，并将其归约为 E（我们还是用 E 作为栈中的匿名非终结符），然后移进 +，现在的格局是：

214

栈	输入
$E+	)$

因为 $+\cdot>)$，调用归约，句柄是+。错误检查程序将进行归约时的错误检测，检查+的左右是否有

E，发现右边缺少E，给出诊断信息“missing operand”，并进行归约。现在的格局为：

栈	输入
$\$E$	)$

$和)之间没有优先关系，而且图4-28中与这对符号对应的项是 e2。程序 e2 被调用并给出诊断信息“unbalanced right parenthesis”，然后从输入中删掉右括号，语法分析器最后的格局为：

栈	输入
$\$E$	$

□

4.7 LR语法分析器

本节介绍一种有效的自底向上语法分析技术。它适用于一大类上下文无关文法的语法分析。这种技术叫做 LR(k) 分析法，L 指的是从左向右扫描输入字符串，R 指的是构造最右推导的逆过程，k 指的是在决定语法分析动作时需要向前看的符号个数。(k)省略时，假设 k 是1。LR分析富有吸引力的原因有以下几点：

- LR语法分析器能识别几乎所有能用上下文无关文法描述的程序设计语言的结构。
- LR分析法是已知的最一般的无回溯移动归约语法分析法，而且可以和其他移动归约分析法一样被有效地实现。
- LR分析法分析的文法类是预测分析法能分析的文法类的真超集。
- 在自左向右扫描输入符号串时，LR 语法分析器能及时发现语法错误。

215

这种分析方法的主要缺点是，对典型的程序设计语言文法，手工构造LR语法分析器的工作量太大，因而需要专门的工具，即 LR 语法分析器的生成器。幸好有许多这样的生成器是可用的。我们将在4.9节讨论其中的一个，即 Yacc 的设计和使用。有了这种生成器，只要写出上下文无关文法，就可以用它自动生成该文法的语法分析器。如果文法有二义性或有其他难以自左向右分析的结构，这种生成器能够识别这些结构，并向编译器设计者报告。

在讨论完 LR 语法分析器的操作后，我们将给出三种构造 LR 分析表的方法。第一种方法称为简单 LR 方法（简称 SLR），它最容易实现，但功能也最弱。对某些文法，另外两种方法能成功地产生语法分析表，但用它却会失败。第二种方法称为规范的 LR 方法，它的功能最强，代价也最高。第三种方法叫做向前看的LR方法（简称LALR），其功能和代价介于前两者之间。LALR方法可用于大多数程序设计语言的文法，并且可以高效地实现。在本节的后面将讨论LR语法分析表的压缩技术。

4.7.1 LR语法分析算法

LR语法分析器的模型如图4-29所示，它是由输入、输出、栈、驱动程序以及包含动作（*action*）和转移（*goto*）两部分的语法分析表构成的。驱动程序对所有的 LR 语法分析器都是一样的，不同的语法分析器只是语法分析表有所不同。分析程序每次从输入缓冲区读入一个符号，并使用栈来存储形如 $s_0X_1s_1X_2s_2\cdots X_ms_m$ 的串，其中 s_m 在栈顶，X_i 是文法符号，s_i 是称为状态的符号，每个状态符号概括了栈中位于它下面的信息。栈顶的状态符号和当前的输入符号用

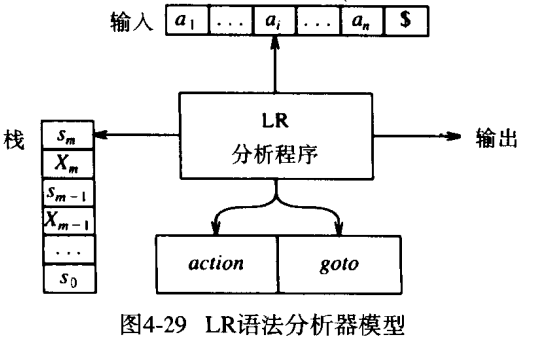


图4-29 LR语法分析器模型

来检索语法分析表，以决定移动归约分析的动作。在实际实现中，文法符号不必出现在栈里，不过我们在讨论中总是包含它们，以帮助解释LR语法分析器的行为。

语法分析表由动作函数 *action* 和转移函数 *goto* 两部分组成。驱动 LR 语法分析器的程序工作如下：它根据栈顶状态 s_m 和当前输入符号 a_i 访问 $action[s_m, a_i]$，即 s_m 和 a_i 所对应的语法分析表项的动作部分，可能的动作如下：

- 移动状态 s 进栈。
- 按文法产生式 $A\rightarrow\beta$ 归约。
- 接受。
- 出错。

函数 *goto* 以状态和文法符号为参数，产生一个状态。我们将会看到，用 SLR、规范的 LR 或 LALR 方法从文法 G 构造的语法分析表的 *goto* 函数是识别 G 的活前缀的确定有穷自动机的转换函数。回想一下，文法 G 的活前缀是右句型的前缀，并且它不会超过句柄的最右端，所以它可以出现在移动归约语法分析器的栈顶。这个确定有穷自动机的初始状态是初始时置于 LR 语法分析器栈中的状态。

LR 语法分析器的格局是一个如下形式的二元组，它的第一个分量是栈的内容，第二个分量是尚未处理的输入：

$$(s_0 X_1 s_1 X_2 s_2 \cdots X_m s_m,\ a_i a_{i+1} \cdots a_n\$)$$

这个格局代表下面的右句型

$$X_1 X_2 \cdots X_m a_i a_{i+1} \cdots a_n$$

由此可以看出，它本质上和一般的移动-归约分析器相同，只是栈中新加了一些状态。

语法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 查询语法分析表的表项 $action[s_m, a_i]$。四种不同的移动所引起的格局变化如下：

1. 如果 $action[s_m, a_i]$=“移动状态 s 进栈”，语法分析器执行移动操作，进入下面的格局：

$$(s_0 X_1 s_1 X_2 s_2 \cdots X_m s_m a_i s,\ a_{i+1} \cdots a_n\$)$$

这里，语法分析器把当前输入符号 a_i 和下一个状态 s（它在 $action[s_m, a_i]$中给出）都移进栈；a_{i+1} 成为当前输入符号。 216 ~ 217

2. 如果 $action[s_m, a_i]$ =“按文法产生式 $A\rightarrow\beta$ 归约”，语法分析器执行归约操作，进入下面的格局：

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

其中，$s = goto[s_{m-r}, A]$，r 是产生式右部 β 的长度，在此，语法分析器首先从栈中弹出 $2r$ 个符号，分别为 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的右部 β），栈顶为状态 s_{m-r}。然后，语法分析器把产生式左边的符号 A 和 $goto[s_{m-r}, A] = s$ 压入栈。执行归约操作时，当前输入符号没有变化。对于我们要构造的 LR 语法分析器，从栈中弹出的文法符号序列 $X_{m-r+1}\cdots X_m$ 总是能匹配归约式的右部 β。LR 语法分析器的输出由归约时执行的与归约产生式有关的语义动作产生，现在，我们暂且认为输出就是打印归约中使用的产生式。

3. 如果 $action[s_m, a_i]$ =“接受”，分析完成。

4. 如果 $action[s_m, a_i]$ =“出错”，语法分析器发现错误，调用错误恢复程序。

下边是 LR 分析算法的小结。所有的 LR 语法分析器都以这种方式工作，只是语法分析表的 *action* 和 *goto* 域的信息有所不同。

算法4.7 LR 分析算法。

输入：文法 G 的 LR 语法分析表和输入串 w。

输出：如果 w 属于 $L(G)$，则输出 w 的自底向上分析，否则报错。

方法：首先，把初始状态 s_0 放在语法分析器栈顶，把 $w\$$ 放在输入缓冲区；然后，语法分析器执行图4-30的程序，直到遇见接受或出错动作为止。 □

```
令 ip 指向w$ 的第一个符号;
repeat forever begin
    令s 是栈顶的状态，a 是ip 所指向的符号;
    if action[s, a] = "移动状态s' 进栈" then begin
        把a和s' 依次压入栈顶;
        让 ip 指向下一个输入符号
    end
    else if action[s, a] = "按文法产生式A→β归约" then begin
        从栈顶弹出 2*|β| 个符号;
        令s'是现在的栈顶状态;
        把A和goto[s', A]依次压入栈;
        输出产生式A→β
    end
    end if action [s, a] = "接受" then
        return
    else error()
end
```

图4-30 LR 分析程序

例4.33 下面是含有二元操作符 + 和 * 的算术表达式的文法：

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to (E)$
(6) $F \to \mathbf{id}$

状态	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

图4-31 表达式文法的语法分析表

图4-31给出了 G 的LR语法分析表，包括动作函数和转移函数。

各个动作的编码的含义是：

1. s_i 表示把当前输入符号和状态 i 压进栈。

2. r_j 表示按第 j 个产生式归约。

3. acc 表示接受。

4. 空白表示出错。

218
~
219

注意，对于终结符 a，状态转移动作 *goto* $[s, a]$可以在表的动作表项 *action*$[s, a]$中找到，所以转移域中仅给出非终结符 A 的 *goto*$[s, A]$。另外，我们还没有解释图4-31中的表项都是怎么确定的，这个问题稍后讨论。

给定输入字符串 **id*id+id**，栈和输入内容的变化序列如图4-32所示。例如，在第一行，

LR语法分析器处于状态 0，**id** 是第一个输入符号。图4-31中第0行和第 **id** 列对应的动作域是 s5，其含义是 **id** 进栈，并把状态 5 压入栈顶。图4-32的第二行给出了执行动作 s5 后的格局：第一输入记号 **id** 和状态 5 进栈，把 **id** 从输入字符串删除。

然后，* 成为当前输入符号，状态 5 遇见输入 * 的动作是按 $F \rightarrow$ **id** 归约：弹出栈顶的两个符号（一个状态符号和一个文法符号），栈顶为状态 0。因为 *goto*[0, *F*] 是 3，语法分析器把 *F*和3压入栈顶，到达第3行所示的格局。我们可以类似地确定剩余的动作。 □

	栈	输入	动作
(1)	0	**id** * **id** + **id** $	移动进栈
(2)	0 **id** 5	* **id** + **id** $	用$F \rightarrow$ **id**归约
(3)	0 *F* 3	* **id** + **id** $	用 $T \rightarrow F$ 归约
(4)	0 *T* 2	* **id** + **id** $	移动进栈
(5)	0 *T* 2 * 7	**id** + **id** $	移动进栈
(6)	0 *T* 2 * 7 **id** 5	+ **id** $	用 $F \rightarrow$ **id**归约
(7)	0 *T* 2 * 7 *F* 10	+ **id** $	用 $T \rightarrow T * F$ 归约
(8)	0 *T* 2	+ **id** $	用 $E \rightarrow T$ 归约
(9)	0 *E* 1	+ **id** $	移动进栈
(10)	0 *E* 1 + 6	**id** $	移动进栈
(11)	0 *E* 1 + 6 **id** 5	$	用 $F \rightarrow$ **id** 归约
(12)	0 *E* 1 + 6 *F* 3	$	用 $T \rightarrow F$ 归约
(13)	0 *E* 1 + 6 *T* 9	$	用 $E \rightarrow E + T$ 归约
(14)	0 *E* 1	$	接受

图4-32 LR语法分析器在输入 **id*id+id** 上所做的移动

4.7.2 LR文法

现在我们讨论怎样为一个给定的文法构造 LR 语法分析表。给定文法 *G*，如果我们能为 *G* 构造出 LR 语法分析表，则称 *G* 是 LR 文法。很多上下文无关文法不是 LR 文法。但是，典型的程序设计语言的结构一般都可以避免非 LR 文法。直观地，为了使一个文法是 LR 文法，只要保证在句柄出现在栈顶时，自左向右扫描的移动归约分析器能够及时地识别它。 220

LR 语法分析器不需要扫描整个栈就可以知道什么时候句柄出现在栈顶。栈顶的状态符号包含了所需要的一切信息。一个非常重要的事实是：如果仅知道栈中的文法符号就可以识别句柄，则存在一个有穷自动机，通过自顶向下读栈中的文法符号自动机就能确定栈顶的句柄（如果有的话）。LR 语法分析表的 goto 函数实质上就是这样的有穷自动机。不过，这个有穷自动机不需要每次移动都读栈。如果识别句柄的有穷自动机自底向上读栈中的文法符号，栈顶所存的状态符号正好是这个自动机所进入的状态。于是，LR语法分析器从栈顶的状态即可确定它需要从栈中了解的一切。

能够用来帮助LR语法分析器作出移动归约决定的另一个信息源是下 *k* 个输入符号。我们实际上感兴趣的是 $k = 0$ 或 $k = 1$ 时的情况。这里我们仅讨论 $k \leqslant 1$ 的情况。例如，图4-31中的动作表只需要向前看一个符号。每步需要向前看 *k* 个符号的LR语法分析器所分析的文法叫做 LR(*k*)文法。

LL 文法和 LR 文法之间有明显的区别。对于 LR(*k*) 文法，我们必须通过向前看 *k* 个输入符号就能够知道一个产生式的右部所能推导出的所有字符串，进而识别出这个产生式右部的出现。这个要求远不如 LL(*k*) 那么严格。LL(*k*) 文法要求只要看到了产生式右部推出的前 *k* 个符号后就能识别出用于归约的产生式。所以 LR 文法比 LL 文法描述的语言更多。

4.7.3 构造SLR语法分析表

现在我们来讨论怎样由文法构造LR分析表。我们将给出三种方法，它们的能力和实现的难易程度各不相同。第一种方法被称为简单LR，即SLR。SLR的功能最弱，但最易实现。根据这种方法构造的语法分析表叫做SLR语法分析表。使用SLR语法分析表的LR语法分析器叫做SLR语法分析器。能够为之构造SLR语法分析器的文法叫做SLR文法。另外两种方法通过用向前看信息增强了SLR文法。所以SLR方法是研究LR语法分析的理想起点。

文法 G 的LR(0)项目（简称项目）是在 G 的产生式右部的某处加点的产生式。例如，产生式 $A \rightarrow XYZ$ 可以生成如下四个项目：

$$
\begin{aligned}
&A \rightarrow \cdot XYZ\\
&A \rightarrow X \cdot YZ\\
&A \rightarrow XY \cdot Z\\
&A \rightarrow XYZ \cdot
\end{aligned}
$$

221

产生式 $A \rightarrow \epsilon$ 只生成一个项目 $A \rightarrow \cdot$。项目可以用一对整数来表示：第一个数表示产生式的号码，第二个数表示点的位置。直观地，项目表示：在语法分析过程中的某一时刻，我们已经看见了一个产生式所能推出的字符串的多大部分。例如，上面第一个项目表示我们希望下一步从输入中看见由 XYZ 推出的字符串，第二个项目表示我们刚从输入中看见了由 X 推出的字符串，下面希望看见由 YZ 推出的字符串。

SLR方法的主要思想是首先从文法构造出识别活前缀的确定有穷自动机。我们把项目划分成一组集合，这些集合对应SLR语法分析器的状态（也是DFA的状态。——译者注）。项目可以看成是识别活前缀的NFA的状态，而“项目分组”就是3.6节中讨论的子集构造法。

被称为规范LR(0)项目集族的LR(0)项目集合族是构造SLR语法分析表的基础。为了构造文法的规范LR(0)项目集族，我们需要定义拓广文法的概念，并引入闭包（*closure*）运算和转移函数（*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 的基础上增加一个新的开始符号 S' 和产生式 $S' \rightarrow S$。新产生式的目的是用来指示语法分析器什么时候应该停止分析并宣布接受输入，即当且仅当语法分析器执行归约 $S' \rightarrow S$ 时，分析成功。

4.7.3.1 闭包运算 *closure*

如果 I 是文法 G 的项目集，那么 *closure*(I) 是从 I 出发由下面两条规则构造的项目集：

1. 初始时，把 I 的每个项目都加入到 *closure*(I)中；

2. 如果 $A \rightarrow \alpha \cdot B\beta$ 在 *closure*(I)中，且存在产生式 $B \rightarrow \gamma$，若 $B \rightarrow \cdot\gamma$ 不在 *closure*(I) 中，则将其加入 *closure*(I)。反复运用这条规则，直到没有更多的项目可加入 *closure*(I) 为止。

直观上，$A \rightarrow \alpha \cdot B\beta$ 在 *closure*(I) 中表示在语法分析过程的某一时刻，下一步应该从输入中看见的是由 $B\beta$ 推出的串。如果 $B \rightarrow \gamma$ 是一个产生式，我们希望在这个时刻，也应该从输入串中看见从 γ 推出的子串。基于这个原因，我们把 $B \rightarrow \cdot\gamma$ 加入到 *closure*(I)。

例4.34 考虑拓广的表达式文法：

$$
\begin{aligned}
&E' \rightarrow E\\
&E \rightarrow E + T \mid T\\
&T \rightarrow T * F \mid F\\
&F \rightarrow (E) \mid \mathbf{id}
\end{aligned}
\tag{4-19}
$$

222 如果 I 是一个项目集合 $\{[E' \rightarrow \cdot E]\}$，那么*closure*($I$)包含下列项目：

$E' \rightarrow \cdot E$
$E \rightarrow \cdot E+T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T*F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot(E)$
$F \rightarrow \cdot$**id**

这里，按规则(1)，把 $E'\rightarrow \cdot E$ 加入到 *closure*(*I*)。因为 E 紧挨在点的右边，由规则 (2)，把点在左端的 E 产生式，即 $E\rightarrow \cdot E+T$ 和 $E\rightarrow \cdot T$ 都加入到 *closure*(*I*)。同样，把 $T\rightarrow \cdot T*F$ 和 $T\rightarrow \cdot F$ 也加入进去。最后，$T\rightarrow \cdot F$ 导致了$F\rightarrow \cdot(E)$ 和 $F\rightarrow \cdot$**id** 的加入。再没有其他项目可以应用规则 (2)了。 □

计算函数 *closure* 的算法如图4-33所示。一种简单的实现方法是使用一个由 G 的非终结符作为索引的布尔数组 *added*。当为每个 $B\rightarrow\gamma$ 产生式增加项目 $B\rightarrow \cdot\gamma$ 时，将 *added* [*B*] 置为 **true**。

```
function closure ( I )
begin
    j := I;
    repeat
        for J 的每个项目A → α·Bβ 和 G的每个产生式
            B → γ, 若 B → ·γ 不在 J 中 do
                把B → ·γ 加入 J;
     until 没有新项目可加入 J;
    return J
end
```

图4-33 *closure* 的计算

注意，如果一个点在左端的 B 产生式被加入 *closure*(*I*)，则所有 B 产生式都将类似地加入*closure*(*I*)。事实上，我们并不需要列出如此加入项目 $B\rightarrow\cdot\gamma$，而只要列出这样的非终结符 B 就足够了。因此，我们可以把感兴趣的项目集中的项目分成两类：

1. 核心项目：初始项 S′→·S 和所有点不在左端的项目。
2. 非核心项目：点在左端的非初始项目。

我们感兴趣的每个项目集都可以通过求核心项目集的闭包得到。当然，加入闭包的项目决 223
不会是核心项目。这样，如果扔掉所有非核心项目，可以用较少的存储空间来表示我们需要的项目集，因为非核心项目可以通过求闭包来重新生成。

4.7.3.2 *goto* 函数

第二个非常有用的函数是 *goto*(*I*, *X*)，其中 I 是项目集，X 是文法符号。*goto*(*I*, *X*) 定义为所有项目集 [$A\rightarrow\alpha X\cdot\beta$]（[$A\rightarrow\alpha\cdot X\beta$]在 I 中）的闭包。直观上讲，如果 I 是对某个活前缀 γ 的有效项目集，那么 *goto*(*I*, *X*)是对活前缀 γX 有效的项目集。

例4.35 若 I 是两个项目的集合｛[$E'\rightarrow E\cdot$]，[$E\rightarrow E\cdot+T$]｝，则 *goto*(*I*, +)包含以下项目：

$E \rightarrow E + \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot(E)$
$F \rightarrow \cdot$**id**

我们通过考察 I 的项目，找到 + 紧挨在点右侧的项目来计算*goto*(*I*, +)。$E'\rightarrow E\cdot$ 不是这样的项目，但 $E\rightarrow E\cdot+T$ 是这样的项目。把 $E\rightarrow E\cdot+T$ 中的点移过 + 得到｛$E\rightarrow E+\cdot T$｝，然后取它的闭包。 □

4.7.3.3 项目集的构造

现在我们给出拓广文法 G' 的规范 LR(0) 项目集族（下面用 C 表示）的构造算法，如图

4-34所示。

例4.36 例4.34中文法(4-19)的规范LR(0)项目集族如图4-35所示。这个项目集的 *goto* 函数被示为图4-36中有穷自动机 *D* 的状态转换图。 □

```
procedure items (G')
begin
    C := {closure ({[S'→ ·S]})};
    repeat
        for C 的每个项目集 I 和每个文法符号X,
            若goto(I, X)非空且不在C中 do
                把goto(I, X)加入C中
    until 没有更多的项目可以加入C
end
```

图4-34 项目集的构造

在图4-36中，如果 *D* 的每个状态都是终态且I_0是初始状态，那么 *D* 识别的刚好是文法(4-19)的活前缀。这不是偶然的。对每个文法 *G*，规范项目集族的 *goto* 函数定义了一个确定的有穷自动机，识别 *G* 的活前缀。事实上，我们也可以想像一个识别活前缀的不确定的有穷自动机 *N*，其状态就是项目本身。从 $A\to\alpha\cdot X\beta$ 到 $A\to\alpha X\cdot\beta$有一个标记为 *X* 的转换，从 $A\to\alpha\cdot B\beta$ 到 $B\to\cdot\gamma$ 有一个标记为ϵ 的转换。于是，由 *N* 的状态集组成的项目集 *I* 的*closure*(*I*) 恰好是3.6节定义的 NFA 的一个状态集的ϵ 闭包。在由 *N* 用子集构造法构造的 DFA 中，*goto*(*I*, X)定义了状态 *I* 遇见符号 *X* 时的转换。按照这种观点，图4-34中的过程 *items*(*G'*) 正好是应用在由 *G'* 构造的NFA *N* 上的子集构造法。

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

I_5: $F\to$ **id**$\cdot$

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

I_7: $T\to T*\cdot F$
$F\to\cdot(E)$
$F\to\cdot$**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4-35 文法(4-19)的规范LR(0)项目集族

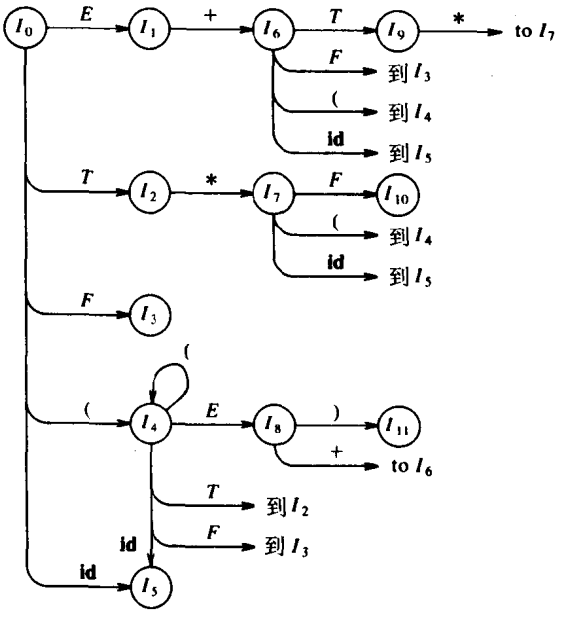


图4-36 活前缀的DFA *D*的状态转换图

有效项目。如果存在一个推导 $S\overset{*}{\underset{rm}{\Rightarrow}}\alpha Aw\overset{*}{\underset{rm}{\Rightarrow}}\alpha\beta_1\beta_2 w$，则说项目 $A\to\beta_1\cdot\beta_2$ 对活前缀 $\alpha\beta_1$ 是有效的。一般而言，同一项目可能对多个活前缀有效。$A\to\beta_1\cdot\beta_2$ 对活前缀 $\alpha\beta_1$ 有效这个事实告诉我们，在发现 $\alpha\beta_1$ 在分析栈时是移进还是归约，特别是，如果 $\beta_2\neq\epsilon$，它暗示句柄还没有完全进栈，动作应该是移进。如果 $\beta_2=\epsilon$，那么 $A\to\beta_1$ 是句柄，应该用这个产生式归约。当然，同一个活前缀的两个有效项目可能告诉我们做不同的事情，有些这样的冲突可以通过向前看下一个输入符号来解决，其他一些冲突可以用下一节的方法解决。当LR方法用于构造任意文法的

语法分析表时，不能保证所有语法分析动作的冲突都能被解决。

我们可以很容易地计算出能够出现在LR语法分析器栈中的每个活前缀的有效项目集。实际上，如果 A 是一个由规范项目集族构造、以 *goto* 函数为转换函数的 DFA，则一个活前缀 γ 的有效项目集正好从 A 的初态出发，沿着标记为 γ 的路径所能到达的那些项目的集合。这是LR分析理论的一条基本定理。本质上，有效项目集蕴含了所有可以从栈中收集到的有用信息。我们不打算证明这个定理，而是用例子来说明。

224 ~ 226

例4.37 让我们再次考虑文法(4-19)。它的项目集和*goto*函数如图4-35和图4-36所示。很明显，串 $E+T*$ 是(4-19)的活前缀。图4-36的自动机读完 $E+T*$ 之后，进入状态 I_7。状态 I_7 包含下列项目：

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

它们都对 $E+T*$ 有效。为了说明这一点，考虑下面三个最右推导：

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

第一个推导说明了 $T \to T * \cdot F$ 对 $E+T*$ 有效性，第二个推导说明了 $F \to \cdot(E)$ 对 $E+T*$ 有效性，第三个推导说明了 $F \to \cdot\mathbf{id}$ 对 $E+T*$ 有效性。可以证明，不存在对 $E+T*$ 有效的其他项目。这个证明留给感兴趣的读者。 □

4.7.3.4 SLR语法分析表

现在我们来说明怎样从识别活前缀的确定有穷自动机构造SLR语法分析表的动作（*action*）函数和转移（*goto*）函数。我们的算法不能为所有的文法都产生惟一确定的分析动作表，但是它可以成功地应用在许多程序设计语言的文法上。给定文法 G，我们首先把 G 拓广到 G'，然后构造 G' 的规范项目集族 C，最后使用下面的算法从 C 构造语法分析表的 *action* 函数和 *goto* 函数。这个算法要求我们知道每个非终结符 A 后面跟随的符号集 FOLLOW(A) (参见4.4节)。

算法4.8 SLR语法分析表的构造。

输入：拓广文法 G'。

输出：G' 的SLR语法分析表函数 *action* 和 *goto*。

方法：

1. 构造 $C = \{I_0, I_1, \cdots, I_n\}$，即$G'$的规范LR(0)项目集族。

2. 从 I_i 构造状态 i，它的分析动作确定如下：

a) 如果 $[A \to \alpha \cdot a\beta]$ 在 I_i 中，并且 $goto(I_i, a) = I_j$，则置 *action*$[i, a]$为“移动 j 进栈”，这里的 a 必须是终结符。

b) 如果$[A \to \alpha \cdot]$在 I_i 中，则对 FOLLOW(A)中的所有 a，置 *action*$[i, a]$为“归约 $A \to \alpha$”。这里的 A 不能是 S'。 227

c) 如果$[S' \to S \cdot]$在 I_i 中，则置 *action*$[i, \$]$为“接受”。

如果由上面的规则产生的动作有冲突，那么文法 G 就不是SLR(1)文法。在这种情况下，该算法构造不出 G 的语法分析器。

3. 对所有的非终结符 A，使用下面的规则构造状态 i 的 *goto* 函数：如果 $goto(I_i, A) = I_j$，则 $goto[i, A] = j$。

4. 不能由规则(2)和规则(3)定义的表项都置为“出错”。

5. 语法分析器的初始状态是从包含 $[S' \rightarrow \cdot S]$ 的项目集构造出的状态。

由算法4.8所确定的由 *action* 和 *goto* 函数组成的语法分析表叫做文法G的SLR(1)表。使用G的SLR(1)表的LR语法分析器叫做 G 的SLR(1)语法分析器。存在SLR(1)语法分析表的文法叫做SLR(1)文法。通常，我们省略SLR后面的(1)，因为我们不讨论向前看多个符号的分析器。

例4.38 让我们为文法(4-19)构造SLR表。它的规范LR(0)项目集族已在图4-35中给出了。首先考虑项目集 I_0：

$$
\begin{aligned}
&E' \rightarrow \cdot E\\
&E \rightarrow \cdot E + T\\
&E \rightarrow \cdot T\\
&T \rightarrow \cdot T * F\\
&T \rightarrow \cdot F\\
&F \rightarrow \cdot (E)\\
&F \rightarrow \cdot \mathbf{id}
\end{aligned}
$$

项目 $F \rightarrow \cdot(E)$ 使得 $action[0, (]$= “移动4进栈”，项目 $F \rightarrow \cdot \mathbf{id}$ 使得 $action[0, \mathbf{id}]$ = “移动5进栈”。I_0中的其他项目不产生动作。现在考虑 I_1：

$$
\begin{aligned}
&E' \rightarrow E\cdot\\
&E \rightarrow E\cdot + T
\end{aligned}
$$

第一项导致 $action[1, \$]$= “接受”，第二项使得 $action[1, +]$ = “移动6进栈”。接下来考虑 I_2：

$$
\begin{aligned}
&E \rightarrow T\cdot\\
&T \rightarrow T\cdot * F
\end{aligned}
$$

因为 FOLLOW(E) = {\$, +,)}，第一项使得 $action[2, \$] = action[2, +] = action[2,)]$ = “归约 $E \rightarrow T$”，第二项使得 $action[2, *]$ = “移动7进栈”。以这种方法继续下去即可得到图4-31中的动作
228 (*action*) 表和转移 (*goto*) 表。在图4-31中，归约动作中的产生式序号和它在原文法(4-18)中出现的顺序是相同的，即 $E \rightarrow E + T$ 的序号是1，$E \rightarrow T$ 的序号是2，等等。 □

例4.39 每个SLR(1)文法都不是二义的，但有很多非二义的文法不是SLR(1)文法。考虑具有下列产生式的文法。

$$
\begin{aligned}
&S \rightarrow L = R\\
&S \rightarrow R\\
&L \rightarrow * R\\
&L \rightarrow \mathbf{id}\\
&R \rightarrow L
\end{aligned}
\qquad (4\text{-}20)
$$

可以把 L 和 R 想像成分别代表左值和右值，而 * 代表一个“取单元内容”的操作符[⊖]。文法(4-20)的规范LR(0)项目集族如图4-37所示。

考虑项目集 I_2。该集合的第一项使得 $action[2, =]$为“移动6进栈”。因为 FOLLOW(R)包含 =（这是因为 $S \Rightarrow L = R \Rightarrow *R = R$），第二项使得 $action[2, =]$为“归约 $R \rightarrow L$”。于是，

⊖ 正如2.8节所介绍的，左值表示一个位置，而右值是可以保存在位置中的值。

action[2，=]有多重定义。因为其中既有移动进栈又有归约，所以状态2在输入符号为 = 时存在移动-归约冲突。

文法 (4-20) 不是二义性的。移动-归约冲突的出现，说明 SLR 语法分析器的构造方法没有强到记住足够多的上下文，以决定在已经看见了可归约到 L 又面临输入 = 时，语法分析器应该采取什么动作。下面要讨论的规范 LR 方法和 LALR 方法能够成功地应用到更大的文法类上，包括文法 (4-20)。但是应该指出，存在一些非二义文法，每种 LR 语法分析器构造方法都会为其产生包含冲突分析动作的语法分析动作表。幸运的是，程序设计语言中通常可以避免使用这样的文法。□

I_0: $S' \to \cdot S$
$S \to \cdot L = R$
$S \to \cdot R$
$L \to \cdot *R$
$L \to \cdot \mathbf{id}$
$R \to \cdot L$

I_1: $S' \to S \cdot$

I_2: $S \to L \cdot = R$
$R \to L \cdot$

I_3: $S \to R \cdot$

I_4: $L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot \mathbf{id}$

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

I_6: $S \to L = \cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot \mathbf{id}$

I_7: $L \to *R \cdot$

I_8: $R \to L \cdot$

I_9: $S \to L = R \cdot$

图4-37 文法(4-20)的规范LR(0)项目集族

4.7.4 构造规范LR语法分析表

下面讨论构造LR语法分析表的最一般技术。让我们回顾一下，在SLR方法中，如果项目集 I_i 包含项目[$A \to \alpha \cdot$]并且 a 在FOLLOW(A)中，则状态 i 调用 $A \to \alpha$ 归约。然而，在某些情况下，当状态 i 出现在栈顶时，栈内的活前缀 $\beta\alpha$ 使得在任何右句型中，βA 不能跟随 a。因此，在这种情况下，用 $A \to \alpha$ 进行归约是无效的。

例4.40 让我们重新考虑例4.39。状态2中的项目 $R \to L \cdot$ 对应于上面的 $A \to \alpha$，a 对应于符号=，=在FOLLOW(R)中。于是，当SLR语法分析器处于状态2而且下一个输入符号为 = 时，SLR语法分析器调用 $R \to L$ 归约（因为项目 $S \to L \cdot = R$ 也在状态2中，移动进栈动作也要被调用）。然而，例4.39的文法没有以 R=… 开始的右句型。因此，仅与活前缀 L 相对应的状态2不应该调用把 L 归约成 R 的归约动作。□

我们可以令该状态蕴涵更多的信息，使之能够剔除上述那些用 $A \to \alpha$ 进行的无效归约。必要时，我们可以通过分裂状态，使LR语法分析器的每个状态能确切地指出句柄 α 后紧跟哪些终结符时才可能把 α 归约为 A。

通过重新定义项目，使之包含一个终结符作为第二个分量，可以把更多的信息并入状态中。项目的一般形式也就变成了[$A \to \alpha \cdot \beta$，a]，其中 $A \to \alpha\beta$ 是产生式，a 是终结符或 \$。这样的对象叫做LR(1)项目。1是第二个分量的长度，这个分量叫做项目的搜索符。⊖搜索符对于 β 为非 ϵ 的项目[$A \to \alpha \cdot \beta$，a]没有影响。对于形如[$A \to \alpha \cdot$，a]的项目，搜索符则表示只有下一个输入符号是 a 时该项目才能调用 $A \to \alpha$ 归约。于是，只有[$A \to \alpha \cdot$，a]是栈顶状态的 LR(1)项目而且输入符号为 a 时，我们才能使用 $A \to \alpha$ 进行归约。这样的 a 的集合总是 FOLIOW(A)的一个子集，可能是真子集，如例4.40那样。

229 ~ 230

形式地，我们说 LR(1)项目[$A \to \alpha \cdot \beta$，a]对活前缀 γ 有效，如果存在推导 $S \overset{*}{\underset{rm}{\Rightarrow}} \delta A w \underset{rm}{\Rightarrow} \delta\alpha\beta w$，其中：

⊖ 当然，搜索符可能是长度大于1的串，但我们在此不考虑这种串。

1. $\gamma = \delta\alpha$。

2. a 是 w 的第一个符号，或者 w 是 ϵ 且 a 是 \$。

例4.41 让我们考虑如下文法：

$$S \rightarrow BB$$
$$B \rightarrow aB \mid b$$

它有一个最右推导 $S \overset{*}{\underset{rm}{\Rightarrow}} aaBab \underset{rm}{\Rightarrow} aaaBab$。在上面的定义中，令 $\delta = aa$，$A = B$，$w = ab$，$\alpha = a$，$\beta = B$，我们可以看到，项目 $[B \rightarrow a \cdot B, a]$ 对活前缀 $\gamma = aaa$ 是有效的。这个文法的另一个最右推导是 $S \overset{*}{\underset{rm}{\Rightarrow}} BaB \underset{rm}{\Rightarrow} BaaB$。从这个推导可以看出，项目$[B \rightarrow a \cdot B, \$]$对于活前缀 Baa 是有效的。 □

规范构造有效的LR(1)项目集族的方法本质上和构造规范LR(0)项目集族的方法是一样的，只需要修改 *closure* 和 *goto* 这两个过程。

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 $[A \rightarrow \alpha \cdot B\beta, a]$，必定存在一个最右推导 $S \overset{*}{\underset{rm}{\Rightarrow}} \delta Aax \underset{rm}{\Rightarrow} \delta\alpha B\beta ax$，其中 $\gamma = \delta\alpha$。假设 βax 能推出终结符串 by，那么对每个形如 $B \rightarrow \eta$ 的产生式，存在推导 $S \overset{*}{\underset{rm}{\Rightarrow}} \gamma Bby \underset{rm}{\Rightarrow} \gamma\eta By$，于是 $[B \rightarrow \cdot\eta, b]$ 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 $\beta ax \overset{*}{\Rightarrow} by$ 中，β 推出 ϵ，b 就成了 a。总结这两种可能性，可以说 b 是 FIRST(βax) 中的任何终结符，其中 FIRST 的定义见4.4节。注意，x 不可能包含 by 的第一个终结符，所以 FIRST(βax) = FIRST (βa)。现在我们给出 LR(1) 项目集的构造算法。

算法4.9 LR(1)项目集的构造。

输入：拓广文法 G'。

输出：LR(1)项目集，它们是对 G' 的一个或多个活前缀有效的项目集。

方法：构造项目集的过程 *closure* 和 *goto* 及主例程 *items* 如图4-38所示。 □

例4.42 考虑下面的拓广文法：

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow CC \\ C &\rightarrow cC \mid d \end{aligned} \tag{4-21}$$

231

首先计算{[$S' \rightarrow S \cdot$, \$]}的闭包。为计算闭包，我们用项目 $[S' \rightarrow S \cdot, \$]$ 来匹配过程 *closure*中的项目 $[A \rightarrow \alpha B \cdot \beta, a]$，即令$A = S'$，$\alpha = \epsilon$，$B = S$，$\beta = \epsilon$，$a = \$$。在函数 *closure* 中，要求对每个产生式 $B \rightarrow \gamma$ 和 FIRST(βa)的每个终结符b，把 $[B \rightarrow \cdot\gamma, b]$ 加到闭包中。根据当前文法，$B \rightarrow \gamma$ 只能是 $S \rightarrow CC$，而且因为$\beta = \epsilon$，$a = \$$，b 也只能是 \$，因此将 $[S \rightarrow \cdot CC, \$]$ 加入到闭包中。

接下来，对于 FIRST($C\$$)中的 b，将项目$[C \rightarrow \cdot\gamma, b]$加入闭包。也就是说，用项目$[S \rightarrow \cdot CC, \$]$匹配项目 $[A \rightarrow \alpha B \cdot \beta, a]$，其中$A=S$，$\alpha = \epsilon$，$B = C$，$\beta = C$，$a = \$$。因为 C 不会推导出空串，所以FIRST($C\$$) = FIRST(C)。又因为FIRST(C)包含终结符 c 和 d，所以将以下项目加入闭包：$[C \rightarrow \cdot cC, c]$，$[C \rightarrow \cdot cC, d]$，$[C \rightarrow \cdot d, c]$，$[C \rightarrow \cdot d, d]$。因为再没有紧跟在点右面的非终结符，所以我们就完成了对第一个LR(1)项目集的计算。这个初始项目集为：

I_0: $S' \rightarrow \cdot S, \$$
$S \rightarrow \cdot CC, \$$
$C \rightarrow \cdot cC, c/d$
$C \rightarrow \cdot d, c/d$

为方便起见，我们把表示LR(1)项目的方括号省略了，而且用 [$C \rightarrow \cdot cC$, c/d] 表示 [$C \rightarrow \cdot cC$, c]和 [$C \rightarrow \cdot cC$, d] 的缩写。

```
function closure(I);
begin
    repeat
        for I中的每个项目[A→α·Bβ, a]，G'中的每个产生式B→γ和FIRST(βa )的每个终结符b，如果
        [B→·γ, b]不在 I 中 do
                把[B→·γ, b]加到 I 中；
    until再没有项目可加到 I 中；
    return I
end;

function goto(I, X);
begin
    对于 I 中的项目[A→α·Xβ, a]，令 J 是项目[A→αX·β, a]的集合；
    return closure(J)
end;

procedure items(G');
begin
    C := {closure({[S' → ·S, $]})};
    repeat
        for C 的每个项目I和每个文法符号X，若goto(I, X)
            非空且不在 C 中 do
                把goto(I, X)加入 C 中
    until 再没有项目集可以加入 C 中
end
```

图4-38 文法G'的LR(1)项目集的构造

现在对不同的 X 值计算 $goto(I_0, X)$。对$X = S$，因为点在右端，所以项目集中只有项目 [$S' \rightarrow S\cdot$, \$]，因此第二个项目集为：

I_1: $S' \rightarrow S\cdot, \$$

对 $X = C$，求[$S \rightarrow C\cdot C$, \$]的闭包，加入第二个分量是 \$ 的 C-产生式（左部为 C 的所有产生式）得到如下项目集：

I_2: $S \rightarrow C\cdot C, \$$
$C \rightarrow \cdot cC, \$$
$C \rightarrow \cdot d, \$$

接下来，令 $X = c$，求{[$C \rightarrow c\cdot C$, c/d]} 的闭包，加入第二个分量是 c/d 的 C-产生式得到如下项目集：

I_3: $C \rightarrow c\cdot C, c/d$
$C \rightarrow \cdot cC, c/d$
$C \rightarrow \cdot d, c/d$

最后，令 $X = d$，用以下项目集结束：

I_4: $C \to d\cdot, c/d$

现在已经求完 I_0 上的转移函数，I_1 上没有转移，但下一个输入是 C、c 和 d 时 I_2 上有转移。对 C 可得如下项目集：

I_5: $S \to CC\cdot, \$$

不需要再求闭包。对 c，求{[$C \to c\cdot C$, \$]}的闭包可得：

I_6: $C \to c\cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$

注意，I_6 和 I_3 只是第二个分量不相同。我们将会看到，文法的许多LR(1)项目集的第一个分量相同而第二个分量不相同。当我们构造LR(0)项目集时，每个LR(0)项目集会与一个或多个LR(1)项目集的第一个分量一致。我们将在讨论LALR分析法时再详细解释这种现象。

232
~
233

下面继续构造 I_2 的 *goto* 函数，*goto* (I_2, d)是：

I_7: $C \to d\cdot, \$$

现在计算 I_3 的 *goto* 函数，I_3 在 c 和 d 上的转移分别是 I_3 和 I_4，而*goto* (I_3, C) 是：

I_8: $C \to cC\cdot, c/d$

I_4 和 I_5 上没有转移。I_6 在 c 和 d 上的转移分别是 I_6 和 I_7，*goto* (I_6, C) 是：

I_9: $C \to cC\cdot, \$$

剩下的项目集上都没有转移，因此最终的十个项目集及其上的转移如图4-39所示。 □

现在我们给出从LR(1)项目集构造LR(1)语法分析表动作函数和转移函数的规则。动作函数和转移函数也采用前面的表的表示方式，惟一的区别是表项的值不同。

算法4.10 规范LR语法分析表的构造。

输入：拓广文法 G'。

输出：文法 G' 的规范 LR 语法分析表函数 *action* 和 *goto*。

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 构造语法分析器的状态 i，状态 i 的分析动作确定如下：

a) 如果 [$A \to \alpha\cdot a\beta$, b]在I_i 中且 *goto* (I_i, a) = I_j，则置 *action*[i, a]为s_j，即“移动 j 进栈”，这里要求 a 必须是终结符。

b) 如果[$A \to \alpha\cdot$, a]在I_i 中且 $A \neq S'$，则置*action*[i, a]为 r_j，即按 r_j 归约，其中 j 是产生式 $A \to \alpha$ 的序号。

c) 如果[$S' \to S\cdot$, \$]在 I_i 中，则置 *action*[i, \$] = acc，表示接受。

如果用上面的规则构造语法分析表时出现冲突，那么该文法就不是LR(1)的，该算法对此文法失败。

3. 状态i的转移按下面的方法确定：如果 *goto*(I_i, A) = I_j，那么 *goto*[i, A] = j 。

4. 用规则(2)和规则(3)中未能定义的所有表项都置为“出错”。

5. 语法分析器的初始状态是由包含 [$S' \to \cdot S$, \$] 的项目集构造出的状态。 □

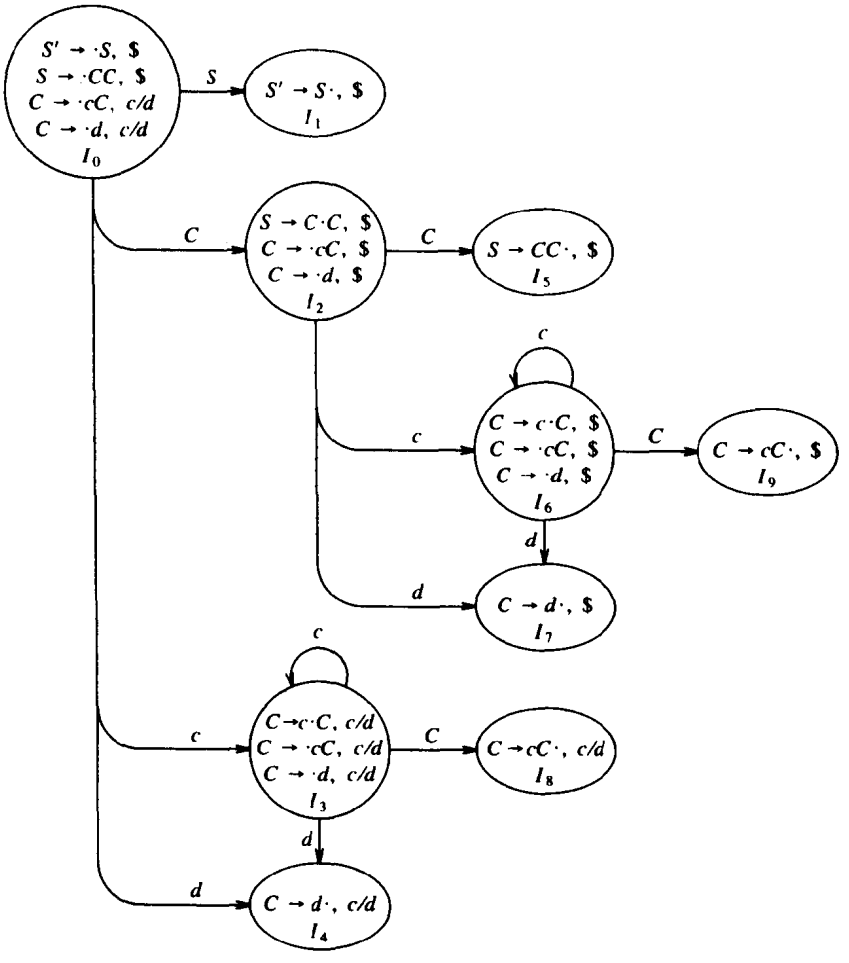


图4-39 文法(4-21)的转移图

由算法4.10产生的动作函数和转移函数所构成的表叫做*规范LR(1)语法分析表*，使用这种表的LR语法分析器叫做规范LR(1)分析器。如果动作函数没有多重定义的表项，那么这个文法叫做LR(1)文法。和前面一样，如果不会引起误解的话，可以省略“(1)”。

例4.43 文法(4-21)的规范LR语法分析表如图4-40所示，产生式1、2和3分别是 $S\rightarrow CC$、$C\rightarrow cC$ 和 $C\rightarrow d$。 □

每个SLR(1)文法都是LR(1)文法，但对于SLR(1)文法，规范LR语法分析器可能比同一文法的SLR语法分析器具有更多的状态。上面例子中的文法是SLR文法，它的SLR语法分析器只有7个状态，而图4-40却有10个状态。

状态	action			goto	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

图4-40 文法(4-21)的规范LR语法分析表

4.7.5 构造LALR语法分析表

现在我们介绍最后一种分析器的构造方法，即 LALR(lookahead-LR) 技术。在实践中经常

使用这种方法，因为由它产生的语法分析表比规范LR语法分析表要小得多，而大多数普通的程序设计语言的结构又都可以方便地用LALR文法来表示。同样的结论对SLR文法几乎也是对的，但是有少数结构不能方便地用SLR技术进行处理（例4.39便是一个例子）。

就语法分析器的大小而言，SLR表和LALR表对同一个文法具有同样多的状态，对Pascal这样的语言有几百个状态，而同样大小的语言的规范LR表则有几千个状态。所以，构造SLR表和LALR表比构造规范LR表要经济得多。

作为入门，让我们再次考虑文法(4-21)，它的LR(1)项目集如图4-39所示。取一对看起来类似的状态，如 I_4 和 I_7，它们都只有一个项目，并且第一分量都是 $C\rightarrow d\cdot$，I_4 中搜索符是 c 或 d，I_7 中搜索符是 \$。

我们来看一下语法分析器中 I_4 和 I_7 的不同作用。注意，文法(4-21)产生的是正规集 c^*dc^*d。
234
~
236
当读入输入串 $cc\cdots cdcc\cdots cd$ 时，语法分析器把第一组 c 及其后面的 d 移进栈中，进入状态4。如果下一个输入符号是 c 或 d，语法分析器将按产生式 $C\rightarrow d$ 归约。要求 c 或 d 跟随是合乎情理的，因为它们属于串 c^*d 的开始符号。如果跟在第一个 d 后的下一个输入符号是 \$，例如，输入符号串 ccd，因为该符号串不在此正规集内，因此分析器将报错（由状态4正确的指出错误）。

语法分析器在读入第二个 d 之后进入状态7。下一个输入符号必须是\$，否则输入串就不是 c^*dc^*d 的形式。所以合理的做法是，面临\$时状态7应按 $C\rightarrow d$ 归约，面临 c 或 d 时报告错误。

现在，让我们把状态 I_4 和 I_7 合并为 I_{47}，把它们的搜索符合起来，成为[$C\rightarrow d\cdot$，$c/d/\$$]。从 I_0，I_2，I_3 和 I_6 到达 I_4 或 I_7 的在 d 上的转移现在都进入 I_{47}，状态 I_{47} 的动作是对任何输入符号都进行归约。修改后的语法分析器的行为本质上和原来的一样，但它会把某些情况下的 d 归约成 C，而原来的语法分析器对这些情况是报错的，如输入为 ccd 或 $cdcdc$ 时。值得庆幸的是，这些错误最终会被捕获，而且是在移进下一个输入符号前被捕获。

更一般地讲，我们可以寻找同心的（即第一分量相同）LR(1)项目集，并把这些同心的项目集合并成一个项目集。例如，在图4-39中，I_4 和 I_7 形成这样的一对同心集，心为 $\{C\rightarrow d\cdot\}$。类似地，I_3 和 I_6 形成另一对，它们的心是 $\{C\rightarrow c\cdot C,\ C\rightarrow\cdot cC,\ C\rightarrow\cdot d\}$。还有一对是 I_8 和 I_9，它们的心是$\{C\rightarrow cC\cdot\}$。注意，一般而言，心是相应文法的一个LR(0)项目集；另外，LR(1)文法可能产生多个同心的项目集合。

因为 $goto(I,\ X)$的心只依赖于I的心，所以LR(1)项目集合并后的转移函数可以通过 $goto(I,\ X)$自身的合并而得到，这样，在合并项目集时不必同时考虑修改转移函数的问题。动作函数应作相应修改，使得它能反映各个被合并集合的既定动作。

假设有一个LR(1)文法，即它的LR(1)项目集中不存在分析动作冲突。如果我们把同心的项目集合并为一，就可能导致冲突，但是这种冲突不会是移动-归约冲突。因为如果存在这种冲突，则意味着对当前输入符号 a，有一个项目[$A\rightarrow\alpha\cdot$，a]要求以 $A\rightarrow\alpha$ 进行归约，同时又有另一个项目[$B\rightarrow\beta\cdot a\gamma$，$b$]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[$A\rightarrow\alpha\cdot$，a]和[$B\rightarrow\beta\cdot\alpha\gamma$，$c$]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动-归约冲突。从而文法不是LR(1)的，这与假设不符。事实上移动-归约冲突不依赖于搜索符号而只依赖于其心，因此，同心集的合并不会引起新的移动-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，如下面的例子所示。

237
例4.44 考虑如下文法：

$S' \rightarrow S$

$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$
$A \rightarrow c$
$B \rightarrow c$

它只产生四个串：*acd*，*ace*，*bcd* 和 *bce*。通过构造该文法的LR(1)项目集，可以看出没有冲突，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 {[$A \rightarrow c\cdot$, *d*], [$B \rightarrow c\cdot$, *e*]}，对 *bc* 有效的项目集为{[$A \rightarrow c\cdot$, *e*], [$B \rightarrow c\cdot$, *d*]}，这两个集合都没有产生冲突且是同心的，然而，它们合并后

$A \rightarrow c\cdot, d/e$
$B \rightarrow c\cdot, d/e$

产生归约-归约冲突。因为当面临输入符号 *d* 或 *e* 时，不知道应该用哪个产生式进行归约。 □

下面我们将给出构造LALR语法分析表的第一个算法，其基本思想是：首先构造LR(1)项目集族，如果它不存在冲突，就把同心集合并在一起，再按这个项目集族构造语法分析表。该方法可以作为描述LALR(1)文法的基本定义。在实际应用中，由于构造完整的LR(1)项目集族需要很多的空间和时间，因而需要另找算法。

算法4.11 一个简易但耗空间的LALR表构造法。

输入：拓广文法 G'。

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

方法：

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

2. 对出现在LR(1)项目集中的每个心，找出所有与之同心的项目集，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。按照算法4.10中的方式从 J_i 来构造状态*i*的动作。如果分析动作出现冲突，算法无法产生分析表，说明该文法不是LALR(1)文法。

4. *goto* 表的构造如下：如果 J 是一个或多个LR(1)项目集的并，即 $J = I_1 \cup I_2 \cup \cdots \cup I_k$，那么 $goto(I_1, X)$, $goto(I_2, X)$, $\cdots$, $goto(I_k, X)$也同心，因为 I_1, I_2, $\cdots$, I_k 都同心。记 K 为所有与 $goto(I_1, X)$同心的项目集的并，则 $goto(J, X) = K$。 □

238

由算法4.11产生的表称为 G 的LALR分析表。如果分析动作没有冲突，则该文法称为LALR(1)文法。在第(3)步中构造的项目集族叫做LALR(1)项目集族。

例4.45 再次考虑文法(4-21)，它的转移图如图4-39所示。正如我们前面所提到的，有3对项目集可以被合并，I_3和I_6合并成：

I_{36}: $C \rightarrow c\cdot C, c/d/\$$
$C \rightarrow \cdot cC, c/d/\$$
$C \rightarrow \cdot d, c/d/\$$

I_4和I_7合并成：

I_{47}: $C \rightarrow d\cdot, c/d/\$$

I_8和I_9合并成：

I_{89}: $C \rightarrow cC\cdot, c/d/\$$

状态	*action*			*goto*	
	c	*d*	$	*S*	*C*
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

图4-41 文法(4-21)的LALR语法分析表

将项目集压缩后的 LALR 动作函数和移转函数如图4-41所示。

现在来看看转移函数是怎样计算的。考虑 $goto(I_{36}, C)$，在原来的LR(1)项目集中，$goto(I_3, C) = I_8$，而 I_8 现在是 I_{89} 的一部分。因此，置 $goto(I_{36}, C) = I_{89}$。如果考虑 I_{36} 的另一部分 I_6，可以得到同样的结论，即 $goto(I_6, C) = I_9$，而 I_9 现在是 I_{89} 的一部分。又例如，考虑 $goto(I_2, c)$，它指出了在面对输入符号 c 时执行了 I_2 的移进动作之后的转移方向。在原来的LR(1)项目集中，$goto(I_2, c) = I_6$，因为 I_6 现在是 I_{36} 的一部分，所以 $goto(I_2, c) = I_{36}$。于是，语法分析表中状态2面临输入符号 c 的表项是 s36，其含意是移进 c，再把状态36置于栈顶。 □

当输入串为 $c*dc*d$ 时，无论是图4-40的LR语法分析器还是图4-41的LALR语法分析器，都给出了同样的移动归约序列。其差别只是状态名不同而已，即如果LR语法分析器把 I_3 或 I_6 压
239 进栈，LALR语法分析器就把 I_{36} 压进栈。对于正确的输入串，LR语法分析器和LALR语法分析器将非常相像。

但是，当输入串存在错误时，LALR语法分析器可能比LR语法分析器多做了一些不必要的归约，而LR语法分析器则能立即报错。但LALR语法分析器在LR语法分析器报错之后决不会移进更多的符号[⊖]。例如，若输入串是 ccd 并跟随以 $ 时，图4-34的LR语法分析器将把

0 *c* 3 *c* 3 *d* 4

压入栈，并在状态4发现错误，因为状态4面临 $ 的动作是“出错”。对于同一输入串，LALR语法分析器将产生相应的动作，即把

0 *c* 36 *c* 36 *d* 47

压入栈。但状态47面临 $ 的动作是归约 $C \to d$，栈的内容成为

0 *c* 36 *c* 36 *C* 89

现在状态89面临 $ 的动作是归约 $C \to cC$，这时栈的内容改为

0 *c* 36 *C* 89

再经一次类似的归约，获得的栈为

0 *C* 2

最后，状态2面临 $ 的动作是“出错”，语法分析器这时发现错误。

4.7.6 LALR语法分析表的有效构造方法

我们可以对算法4.11进行一些修改，以避免在创建LALR(1)语法分析表时包含全部的LR(1)项目集族，首先，我们注意到可以用项目集I的核来表示I，也就是说，可以用初态项目[$S' \to \cdot S$, $]或那些圆点不在右部最左端的项目来表示$I$。

其次，我们只需利用项目集I的核就能计算I所产生的动作。除非 $\alpha = \epsilon$，否则任何调用 $A \to \alpha$ 进行归约的项目都在该核中。当且仅当存在核项目 $[B \to \gamma \cdot C\delta, b]$，使得对某些 η 及 FIRST($\eta\delta b$)中的a有$C \overset{*}{\underset{rm}{\Rightarrow}} A\eta$时，对输入 a 才可以用 $A \to \epsilon$ 进行归约。对每个非终结符 C，满足 $C \overset{*}{\underset{rm}{\Rightarrow}} A\eta$的所有非终结符$A$都可以预先计算出来。

由 I 产生的移进动作可以通过下列方法由 I 的核来确定。如果存在核项目 $[B \to \gamma \cdot C\delta, b]$，

⊖ 也就是说，就准确地指出输入串的出错位置这一点而言，LALR语法分析器和LR语法分析器是等效的。——译者注

其中$C \overset{*}{\underset{rm}{\Rightarrow}} ax$，且这个推导的最后一步不使用$\epsilon$产生式，则可以将输入符号$a$移进。对于每个$C$，满足$C \overset{*}{\underset{rm}{\Rightarrow}} ax$的所有终结符$a$也是可以预先计算出来的。

下面我们来看一看如何通过核来计算 *goto* 转换。如果 $[B \to \gamma \cdot X\delta, b]$ 在 I 的核中，那么 $[B \to \gamma X \cdot \delta, b]$ 也在 $goto(I, X)$ 的核中。如果 $[B \to \gamma \cdot C\delta, b]$ 在 I 的核中，并且对于某个 η 有 $C \overset{*}{\underset{rm}{\Rightarrow}} A\eta$，那么 $[A \to X \cdot \beta, a]$ 也在 $goto(I, X)$ 的核中。如果我们对每对非终结符 C 和 A 都能预先确定对某个 η 而言是否有关系 $C \overset{*}{\underset{rm}{\Rightarrow}} A\eta$，那么，从核计算项目集这一工作仅比从闭包计算项目集的工作在效率上稍差一点而已。 240

为了计算拓广文法 G' 的LALR(1)项目集，我们先从初态项目集 I_0 的核 $S' \to \cdot S$ 开始，然后，按上述方法计算出从 I_0 出发的转移转换的核。我们继续计算出每一个新生成的核的转移转换直到我们拥有了全部 LR(0) 项目集的核为止。

例4.46 让我们再次考虑下面的拓广文法：

$$
\begin{aligned}
&S' \to S\\
&S \to L = R \mid R\\
&L \to * R \mid \mathbf{id}\\
&R \to L
\end{aligned}
$$

I_0:	$S' \to \cdot S$	I_5:	$L \to \mathbf{id}\cdot$
I_1:	$S' \to S\cdot$	I_6:	$S \to L = \cdot R$
I_2:	$S \to L\cdot = R$	I_7:	$L \to *R\cdot$
	$R \to L\cdot$	I_8:	$R \to L\cdot$
I_3:	$S \to R\cdot$	I_9:	$S \to L = R\cdot$
I_4:	$L \to *\cdot R$		

图4-42 文法(4-20)的LR(0)项目集的核

该文法的LR(0)项目集的核如图4-42所示。 □

现在我们着手为 LR(0) 项目集的核的每一个项目都配上适当的搜索符（第二个分量）。为了了解搜索符是如何从一个项目集I传播到 $goto(I, X)$ 的，让我们考虑 I 的核中的一个 LR(0)项目$B \to \gamma \cdot C\delta$。假设对某个 η，$C \overset{*}{\underset{rm}{\Rightarrow}} A\eta$成立(可能 $C = A$，$\eta = \epsilon$)，并且 $A \to X\beta$ 是一个产生式，则LR(0) 项目 $A \to X \cdot \beta$ 在 $goto(I, X)$中。

假设我们计算的是 LR(1) 项目而不是 LR(0) 项目，并且 $[B \to \gamma \cdot C\delta, b]$属于 I，那么对于哪些 a的值，$[A \to X \cdot \beta, a]$ 将在 $goto(I, X)$中呢？当然，如果 a 属于 FIRST($\eta\delta$)，那么由推导 $C \overset{*}{\underset{rm}{\Rightarrow}} A\eta$ 可知$[A \to X \cdot \beta, a]$一定在 $goto(I, X)$之中。在这种情况下，b 的值是无关的，我们称 a（作为 $A \to X \cdot \beta$的一个搜索符）是自生的。由此定义可知，作为初态项目集中的项目 $S' \to \cdot S$ 的一个搜索符，\$ 是自生的。

但是项目 $A \to X \cdot \beta$ 的搜索符的产生还有另一种途径。如果 $\eta\delta \Rightarrow \epsilon$，那么 $[A \to X \cdot \beta, b]$ 也将在 $goto(I, X)$ 中，这时我们称搜索符从 $B \to \gamma \cdot C\delta$ 传播到 $A \to X \cdot \beta$。下面的算法用来确定上述两种方式所产生的搜索符。 241

算法4.12 确定搜索符。

输入：一个LR(0)项目集 I 的核 K 和一个文法符号 X。

输出：对于 $goto(I, X)$ 中的核项目，由 I 中的项目自生的搜索符，以及对于 I 中的项目，搜索符从这些项目将会传播到 $goto(I, X)$ 中的核项目中。

方法：算法如图4-43所示，它使用一个哑搜索符 # 来检测出现超前扫描符号传播的情形。 □

现在我们来考虑如何找到与LR(0)项目集的核中的每个项目相关的搜索符。首先，我们知道，\$是LR(0)初态项目集的项目 $S' \to \cdot S$ 的搜索符，用算法4.12可以为每个核的所有项目

列出其全部自生搜索符。然后，让这些自生搜索符进行传播，直到不能再传播为止。有许多种不同的传播方法，这些方法在某种意义上记录已经传播到一个项目的“新”搜索符，但这些“新”的搜索符没有再继续传播出去，下面的算法描述了一种将搜索符传播到全部项目的技术。

```
for K中每一项B→γ·δ do begin
    J' := closure ({[B→γ·δ, #]});
    if [A→α·Xβ, a] ∈ J' 其中a不是# then
        对于goto(I, X)中的核项目A→αX·β，搜索符a是自生的;
    if [A→α·Xβ, #] ∈ J' then
        搜索符从 I 中的 B→γ·δ 传播到goto(I, X)中的A→αX·β
end
```

图4-43 搜索符的自生与传播的发现

算法4.13 计算LALR(1)项目集族的核。

输入：拓广文法 G'。

输出：文法 G' 的LALR(1) 项目集族的核。

方法：

242

1. 利用上述方法，构造 G 的LR(0)项目集的核。

2. 对每个LR(0)项目集的核和文法符号X应用算法4.12，确定出哪些搜索符对于 $goto(I, X)$ 中的核项目是自生的，并确定出从哪些 I 中的项目出发搜索符可以传播到在 $goto(I, X)$中的核项目中去。

3. 对于每个项目集的每个核项目，初始化一个表，使之给出与之相联系的各个搜索符。开始时，与每一个项目相联系的搜索符仅是那些在(2)中确定的自生搜索符。

4. 反复遍历所有集合中的核项目。当我们访问一个项目i时，利用(2)中所建立的信息来查看i能将它的搜索符传播到哪些核项目。把i的当前搜索符集合附加到已经与i相关联的各个项目中去，即附加到i已将其搜索符传播到达的那些项目中去。继续遍历核项目，直到没有新的搜索符被传播为止。 □

例4.47 下面为上例中的文法构造LALR(1)项目集的核，该文法的LR(0)项目集的核如图4-42所示。当我们把算法4.12应用到 I_0 项目集的核时，计算 {[S'→·S, #]}的闭包，结果是：

$S' \rightarrow \cdot S$, #
$S \rightarrow \cdot L=R$, #
$S \rightarrow \cdot R$, #
$L \rightarrow \cdot *R$, #/=
$L \rightarrow \cdot$ **id**, #/=
$R \rightarrow \cdot L$, #

在该闭包中有两项可导致搜索符的自生，项目[L→ ·$*R$, =]使得“=”是 I_4 中的核项目 L→*·R 的自生搜索符，而项目 [L→·**id**, =] 使得“=”是 I_5 中的核项目 L→ **id**· 的自生

从		到	
I_0:	$S' \rightarrow \cdot S$	I_1:	$S' \rightarrow S\cdot$
		I_2:	$S \rightarrow L\cdot=R$
		I_2:	$R \rightarrow L\cdot$
		I_3:	$S \rightarrow R\cdot$
		I_4:	$L \rightarrow *\cdot R$
		I_5:	$L \rightarrow$ **id**·
I_2:	$S \rightarrow L\cdot=R$	I_6:	$S \rightarrow L=\cdot R$
I_4:	$L \rightarrow *\cdot R$	I_4:	$L \rightarrow *\cdot R$
		I_5:	$L \rightarrow$ **id**·
		I_7:	$L \rightarrow *R\cdot$
		I_8:	$R \rightarrow L\cdot$
I_6:	$S \rightarrow L=\cdot R$	I_4:	$L \rightarrow *\cdot R$
		I_5:	$L \rightarrow *\cdot R$
		I_8:	$R \rightarrow L\cdot$
		I_9:	$S \rightarrow L=R\cdot$

图4-44 搜索符的传播

搜索符。

算法4.13的第2步所确定的核项目之中的搜索符的传播方式如图4-44所示。例如，I_0 在符号 S，L，R，$*$ 和 **id** 之上的转移分别是 I_1，I_2，I_3，I_4 和 I_5。对于 I_0，我们只得到含有一个核项目 $[S'\rightarrow\cdot S, \#]$ 的闭包。于是，$S'\rightarrow\cdot S$ 把它的搜索符传播到 I_1 至 I_5 的每个核项目。

在图4-45中，我们描述了算法4.13的第3和第4步，标有“初始”的一列给出了每个核项目的自生搜索符。在第1遍，搜索符 $ 由 I_0 的 $S'\rightarrow\cdot S$ 传播到图4-44中列出的6个项目。搜索符= 由 I_4 中的 $L\rightarrow *\cdot R$ 传播到 I_7 中的 $L\rightarrow *R\cdot$ 和 I_8 中的 $R\rightarrow L\cdot$。它还传播给它自己以及 I_5 中的 $L\rightarrow$ **id**·，但这些搜索符已经出现过了。在第2遍和第3遍，仅有的可被传播的新搜索符是$，这是在第 2 遍的 I_2 和 I_4 的后继和在第3遍的 I_6 的后继中发现的。在第4遍就没有新的搜索符可传播了，所以搜索符的最后集合如图4-45的最右一列所示。

注意，例4.39中利用SLR方法所出现的移动-归约冲突在这里运用LALR技术时已经不存在了，原因是只有搜索符$与 I_2 中的 $S\rightarrow L\cdot$ 相关，所以，对I_2中的项目 $S\rightarrow L\cdot=R$ 产生的=执行移进时不会产生冲突。 □

集合	项目	搜索符			
		初始	第1遍	第2遍	第3遍
I_0:	$S'\rightarrow\cdot S$	$	$	$	$
I_1:	$S'\rightarrow S\cdot$		$	$	$
I_2:	$S\rightarrow L\cdot=R$		$	$	$
I_2:	$R\rightarrow L\cdot$		$	$	$
I_3:	$S\rightarrow R\cdot$		$	$	$
I_4:	$L\rightarrow *\cdot R$	=	=/$	=/$	=/$
I_5:	$L\rightarrow$ **id**·	=	=/$	=/$	=/$
I_6:	$S\rightarrow L=\cdot R$			$	$
I_7:	$L\rightarrow *R\cdot$		=	=/$	=/$
I_8:	$R\rightarrow L\cdot$		=	=/$	=/$
I_9:	$S\rightarrow L=R\cdot$				$

图4-45 搜索符的计算

4.7.7 LR语法分析表的压缩

一个典型程序设计语言的文法有50～100个终结符和100个产生式，它的LALR语法分析表可能有几百个状态，它的 *action* 函数或许有20 000个输入项，每项至少需要8位进行编码。显然，寻找一种比二维数组更有效的编码非常重要，我们简要介绍几种用于压缩LR语法分析表的 *action* 域和 *goto* 域的技术。

压缩 *action* 域的一种有用技术是基于这样一点，即 *action* 表中经常有许多行是相同的。如图4-40，状态0和状态3就具有相同的 *action* 表项，状态2和6也是，于是我们就能节省很大的空间，时间开销也很少。如果我们对每个状态建立一个指向一维数组的指针，则相同表项的指针指向同一个位置。为存取这个数组的信息，我们给每个终结符分配一个从0到终结符总数减1的整数，并把这个数作为对于每个状态的指针值的偏移量。给定一个状态，第 i 个终结符的分析动作保存在那个状态指针值下的第 i 个位置上。

为每个状态的动作创建一个列表可进一步提高空间的利用效率，但要以轻微降低语法分析器的速度为代价（一般认为这是一种合理的折衷，因为LR式的语法分析器占用整个编译时间的很少一部分）。列表由（终结符，动作）对组成。频繁发生的动作放在列表的尾部，并可用符号“any”代替一个终结符，这意味着如果当前输入符号没有在列表中发现，则不管输入是什么都将执行这个动作。此外，对一行上的错误表项可用归约动作安全取代。以后在移进之前，错误将会被检测到。

例4.48 考虑图4-31的语法分析表，我们首先注意到状态0，4，6和7的动作相同，因此可

以用如下列表描述它们：

符号	动作
id	s5
(	s4
any	error

状态1也有一个类似的列表：

符号	动作
+	s6
$	acc
any	error

243 ~ 245

在状态2，我们可以用 r2 替代 error（错误）表项，故对任一除 * 以外的输入，都将用产生式2进行归约，所以状态2的列表是：

符号	动作
*****	s7
any	r2

状态3只有error表项和r4，可以用后者代替前者，所以状态3的列表仅由（any，r4）组成，状态5,10,11可以类似地进行处理。状态8的列表是：

符号	动作
+	s6
)	s11
any	error

而状态9的列表为：

符号	动作
*****	s7
any	r1

□

通过列表也可以对 *goto* 表进行编码，但在此为每个非终结符 *A* 构造序对列表看起来效率更高。*A* 的列表中序对的格式为（*current_state*，*next_state*），含义为：

$$goto[current_state, A] = next_state$$

因为 *goto* 表的任一列只有很少的状态，所以这个方法很有用。原因是非终结符 *A* 的 *goto* 项只能是从这样的项目集导出的状态，在这些项目集的项中，*A* 在点的左边。如果 $X \neq Y$，则点左边为 *X* 和 *Y* 的项不会出现在同一个项目集中。因此每个状态最多出现在一个 *goto* 列中。

要进一步节省空间，我们注意到在 *goto* 表中的错误表项从来没有被考虑过，因而，我们可以用最常用的非错误表项替换这些错误表项，使该常用表项成为默认值，并把当前的状态替换成“any”。

例4.49 再次考虑图4-31。*F* 列对应状态7的表项为10，其他表项或者是3，或者是 error。我们可以用3替换 error 表项从而为 *F* 列创建如下列表：

current_state	*next_state*
7	10
any	3

同样，*T* 列相应的列表为：

current_state	*next_state*
6	9
any	2

对 E 列可以选择1或8作为默认值，不管哪种情况都需要两个表项。例如，可以为 E 列创建下述列表：

246

current_state	*next_state*
4	8
any	1

□

如果读者合计一下本例和前例所创建列表的表项，再加上从状态到动作列表以及从非终结符到 *next_state* 列表的指针，就不会对实现图4-31矩阵所节省的空间有什么印象。然而，我们不应被这个小例子所误导，对实际的文法，列表表示所需要的空间要比矩阵表示所需的空间少10%。

还应指出，3.9节所讨论的对有穷自动机的表压缩方法也可用于LR语法分析表，这些方法的应用将在练习中进行讨论。

4.8 二义文法的应用

任何二义文法都不是LR文法，因而不属于前一节所讨论的任何一类文法，这是一条定理。但是，正如在本节将要看到的，某些二义文法对语言的说明和实现非常有用。对于像表达式这样的语言结构，二义文法比任何等价的非二义文法提供的说明都要更短、更自然。另外，为了便于对一些特例语法结构进行优化，需要将它们从一般语法结构中分离出来。通过使用二义文法，往文法中增加新的产生式，我们就能标识这些特例结构，这是二义文法的另一种应用。

必须强调，虽然我们使用的文法是二义性的，但在所有情况下都说明了消除二义的规则，以保证对每个句子只有一棵分析树。这样，整个语言的说明仍然是无二义的。另外，我们还强调应该慎用二义结构，并在严格控制的方式下使用。否则不能保证语法分析器将识别什么样的语言。

4.8.1 使用优先级和结合规则来解决分析动作的冲突

考虑程序设计语言中的表达式。下面包含操作符+和*的算术表达式文法是二义的，因为它没有指出操作符+和*的结合规则和优先级：

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id} \tag{4-22}$$

下面的无二义文法产生同样的语言，但赋予了+一个比*低的优先级，并且两个操作符都是左结合的。

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned} \tag{4-23}$$

247

有两个理由可以说明为什么我们愿意用文法(4-22)而不是文法(4-23)。首先，正如我们将要看到的那样，我们可以方便地改变操作符+和*的结合规则及优先级而无需修改文法(4-22)中的产生式及结果语法分析器中的状态数。其次，文法(4-23)的语法分析器要花一部分时间来完成产生式 $E \rightarrow T$ 和 $T \rightarrow F$ 的归约，它们的作用只是突出结合规则和优先级。文法(4-22)的语法分析器不会将时间消耗在归约这样的单产生式(右部只有一个非终结符的产生式。——译者注)上。

用 $E'\rightarrow E$ 拓广后的文法(4-22)的LR(0)项目集如图4-46所示。因为文法(4-22)是二义的，如果我们试图从这些项目集生成LR语法分析表，语法分析动作会有冲突。与项目集 I_7 和 I_8 对应的状态会产生这些冲突。假设我们用SLR方法来构造分析动作表，I_7 产生的归约 $E\rightarrow E+E$ 与面临+和*进行移进之间的冲突不能解决，因为+和*都在FOLLOW(E)中；另一个由 I_8 产生的冲突在归约 $E\rightarrow E*E$ 与面临+和*进行移进之间。事实上，用任何一种LR语法分析表的构造方法都会产生这些冲突。

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: $E\rightarrow E*\cdot E$
$E\rightarrow\cdot E+E$
$E\rightarrow\cdot E*E$
$F\rightarrow\cdot(E)$
$E\rightarrow\cdot$ **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4-46 文法(4-22)拓广后的LR(0)项目集

但是，这些问题可以用+和*的优先级和结合信息来解决。考虑输入 **id + id * id**，它使得基于图4-46的语法分析器在处理 **id + id** 后进入状态7，形成如下格局：

栈	输入
0 E 1 + 4 E 7	* **id** $

如果*的优先级高于+，语法分析器应把*移进栈，准备将*和它两边的 **id** 归约成一个表达式。这正是识别相同语言的图4-31的SLR语法分析器所要做的。另一方面，如果+的优先级高于*，语法分析器应该将 $E+E$ 归约成 E。这样，根据+和*的相关优先级就可以解决状态7中用 $E\rightarrow E+E$ 归约和面临*进行移进之间的冲突。

如果输入是 **id + id + id** 的话，语法分析器处理完 **id + id** 后仍将到达栈内容为 0E1 + 4E7 的格局。在状态7面临+时仍有移动-归约冲突，现在是由操作符 + 的结合规则来决定应如何解决冲突。如果+是左结合的，正确的动作是用 $E\rightarrow E+E$ 归约，即第一个+前后的 **id** 应看成一组。这个选择和例4.34中文法的SLR语法分析器的动作是一致的。

总之，假如+是左结合的，在状态7面临+时应该用 $E\rightarrow E+E$ 归约；如果*的优先级高于+，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*是左结合的且优先级高于+，那么不论面临+还是面临*，语法分析器在状态8的动作都是用 $E\rightarrow E*E$ 归约。原因是面临+时，*的优先级高于+，而面临*时，*是左结合的。

状态	*action*						*goto*
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			8
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

图4-47 文法(4-22)的分析表

按这种方式处理，得到如图4-47所示的LR分析表，产生式1～4分别是 $E\rightarrow E+E$，$E\rightarrow E*E$，$E\rightarrow(E)$ 和 $E\rightarrow$ **id**。有趣的是，从图4-31基于文法(4-23)的SLR语法分析表中删去用单产生式 $E\rightarrow T$ 和 $E\rightarrow F$ 所进行的归约，可以得到类似的动作表。像(4-22)那样的二义文法在LALR分析和规范LR分析中可以用类似的方法处理。

4.8.2 悬空else的二义性

再次考虑下面的条件语句文法：

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

在4.3节已提到过，这个文法是二义的，因为它没有解决悬空 else 的二义性。为了简化讨论，让我们考虑上面文法的抽象，用 i 代表 **if** *expr* **then**，用 e 代表 **else**，并用 a 代表所有其他产生式，然后加上拓广的产生式 $S'\rightarrow S$，重写文法如下：

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow iSeS \mid iS \mid a \end{aligned} \tag{4-24}$$

文法(4-24)的LR(0)项目集如图4-48所示。(4-24)的二义性表现在 I_4 的移动-归约冲突上。$S \rightarrow iS\cdot eS$ 要求移进 e，而 FOLLOW(S) = {e, \$}，所以项目$S\rightarrow iS\cdot$ 要求面临输入 e 时进行归约。

I_0: $S' \rightarrow \cdot S$
$S \rightarrow \cdot iSeS$
$S \rightarrow \cdot iS$
$S \rightarrow \cdot a$

I_1: $S' \rightarrow S\cdot$

I_2: $S \rightarrow i\cdot SeS$
$S \rightarrow i\cdot S$
$S \rightarrow \cdot iSeS$
$S \rightarrow \cdot iS$
$S \rightarrow \cdot a$

I_3: $S \rightarrow a\cdot$

I_4: $S \rightarrow iS\cdot eS$
$S \rightarrow iS\cdot$

I_5: $S \rightarrow iSe\cdot S$
$S \rightarrow \cdot iSeS$
$S \rightarrow \cdot iS$
$S \rightarrow \cdot a$

I_6: $S \rightarrow iSeS\cdot$

图4-48 拓广文法(4-24)的LR(0)状态

现在回到 **if** ⋯ **then** ⋯ **else** 术语，即 **if** *expr* **then** *stmt* 在栈顶并且**else**是第一个输入符号时，究竟是移进 **else**(移进e) 还是将 **if** *expr* **then** *stmt* 归约成 *stmt*（按 $S\rightarrow iS$ 归约）呢？答案是应该移进**else**。因为它要和前面一个 **then** 配对。用文法(4-24)的术语，输入为e（代表 **else**）时，只能让它成为以栈顶的 iS 开始的右部的一部分。如果 e 后面的输入不能分析成S而形成右部 $iSeS$，那么，分析将无法继续。

我们得出的结论是，要解决 I_4 中的移动-归约冲突，应该首先移进 e。按照这个解决办法，由图4-48的项目集构造的SLR语法分析表如图4-49所示。产生式1 ~ 3分别是 $S\rightarrow iSeS$，$S\rightarrow iS$ 和 $S\rightarrow a$。

例如，若输入是 $iiaea$，语法分析器的动作如图4-50所示，它正确地解决了悬空 else 问题。在第(5)行，面临 e 时，状态4选择移进；而在第(9)行，面临 \$ 时，状态4按 $S\rightarrow iS$ 进行归约。

状态	*action*				*goto*
	i	*e*	*a*	$	*S*
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4		s5		r2	
5	s2		s3		6
6		r1		r1	

图4-49 抽象的悬空else文法的LR语法分析表

状态	栈	输入
(1)	0	*iiaea*$
(2)	0*i*2	*iaea*$
(3)	0*i*2*i*2	*aea*$
(4)	0*i*2*i*2*a*3	*ea*$
(5)	0*i*2*i*2*S*4	*ea*$
(6)	0*i*2*i*2*S*4*e*5	*a*$
(7)	0*i*2*i*2*S*4*e*5*a*3	$
(8)	0*i*2*i*2*S*4*e*5*S*6	$
(9)	0*i*2*S*4	$
(10)	0*S*1	$

图4-50 输入为*iiaea* 时的分析动作

作为比较，如果不允许用二义文法说明条件语句，那么我们将不得不使用(4-9)那样的繁琐文法。

4.8.3 特例产生式引起的二义性

用来说明二义文法用途的最后一个例子是下面这种情况，即引入一个额外的产生式来标识

语法结构的特例，这比用其余的产生式来标识更自然一些，但如果我们加入这个额外的产生式就会引起分析动作的冲突。这种冲突的解决需要消除二义性规则，这个规则指出，如果出现归约-归约冲突，按特例产生式归约。这样，和特例产生式相联的语义动作允许用更专门的措施来处理这些特例。

Kernighan 和 Cherry 在他们的公式编排预处理器 EQN 中使用了特例产生式，这是一个有趣的应用。在 EQN 中，数学表达式的语法用带下角标操作符 **sub** 和上角标操作符 **sup** 的文法来描述，如文法片段(4-25)所示，预处理器用花括号表示复合表达式，c 是表示任意正文串的记号。

$$
\begin{array}{ll}
(1) & E \rightarrow E \ \textbf{sub}\ E \ \textbf{sup}\ E \\
(2) & E \rightarrow E \ \textbf{sub}\ E \\
(3) & E \rightarrow E \ \textbf{sup}\ E \\
(4) & E \rightarrow \{\ E\ \} \\
(5) & E \rightarrow c
\end{array}
\tag{4-25}
$$

文法(4-25)是二义的有以下几个原因。该文法没有说明操作符 **sub** 和 **sup** 的结合规则和优先级。即使由 **sub** 和 **sup** 的结合规则和优先级引起的二义性解决了，比如规定这两个操作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，即形如 E **sub** E **sup** E 的表达式。把这种形式的表达式处理为一种特例的理由是，像 a **sub** i **sup** 2 这样的表达式应该排成 a_i^2，而不是 a_i^2 的形式。只有加上特例产生式后，Kernighan 和 Cherry 才使得 EQN 能够产生这种特殊的输出。

为了理解 LR 语法分析器怎样处理这种二义性，我们来构造文法(4-25)的SLR语法分析器。这个文法的LR(0)项目集如图4-51所示。在这个族中，有3个项目集产生分析动作冲突，I_7，I_8和I_{11}在面临 **sub** 或 **sup** 时产生移动-归约冲突，因为文法没有说明这两个操作符的优先级和结合规则。如果规定它们具有相同的优先级而且是右结合的，这些语法分析动作的冲突就都可以解决。因此，每种情况都是移进优先。

I_0:
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E$ **sub** E **sup** E
$E \rightarrow \cdot E$ **sub** E
$E \rightarrow \cdot E$ **sup** E
$E \rightarrow \cdot\{\ E\ \}$
$E \rightarrow \cdot c$

I_1:
$E' \rightarrow E\cdot$
$E \rightarrow E\cdot$ **sub** E **sup** E
$E \rightarrow E\cdot$ **sub** E
$E \rightarrow E\cdot$ **sup** E

I_2:
$E \rightarrow \{\cdot E\ \}$
$E \rightarrow \cdot E$ **sub** E **sup** E
$E \rightarrow \cdot E$ **sub** E
$E \rightarrow \cdot E$ **sup** E
$E \rightarrow \cdot\{\ E\ \}$
$E \rightarrow \cdot c$

I_3:
$E \rightarrow c\cdot$

I_4:
$E \rightarrow E$ **sub** $\cdot E$ **sup** E
$E \rightarrow E$ **sub** $\cdot E$
$E \rightarrow \cdot E$ **sub** E **sup** E
$E \rightarrow \cdot E$ **sub** E
$E \rightarrow \cdot E$ **sup** E
$E \rightarrow \cdot\{\ E\ \}$
$E \rightarrow \cdot c$

I_5:
$E \rightarrow E$ **sup** $\cdot E$
$E \rightarrow \cdot E$ **sub** E **sup** E
$E \rightarrow \cdot E$ **sub** E
$E \rightarrow \cdot E$ **sup** E
$E \rightarrow \cdot\{\ E\ \}$
$E \rightarrow \cdot c$

I_6:
$E \rightarrow E\cdot$ **sub** E **sup** E
$E \rightarrow E\cdot$ **sub** E
$E \rightarrow E\cdot$ **sup** E
$E \rightarrow \{\ E\cdot\}$

I_7:
$E \rightarrow E\cdot$ **sub** E **sup** E
$E \rightarrow E$ **sub** $E\cdot$ **sup** E
$E \rightarrow E\cdot$ **sub** E
$E \rightarrow E$ **sub** $E\cdot$
$E \rightarrow E\cdot$ **sup** E

I_8:
$E \rightarrow E\cdot$ **sub** E **sup** E
$E \rightarrow E\cdot$ **sub** E
$E \rightarrow E\cdot$ **sup** E
$E \rightarrow E$ **sup** $E\cdot$

I_9:
$E \rightarrow \{\ E\ \}\cdot$

I_{10}:
$E \rightarrow E$ **sub** E **sup** $\cdot E$
$E \rightarrow E$ **sup** $\cdot E$
$E \rightarrow \cdot E$ **sub** E **sup** E
$E \rightarrow \cdot E$ **sub** E
$E \rightarrow \cdot E$ **sup** E
$E \rightarrow \cdot\{\ E\ \}$
$E \rightarrow \cdot c$

I_{11}:
$E \rightarrow E\cdot$ **sub** E **sup** E
$E \rightarrow E$ **sub** E **sup** $E\cdot$
$E \rightarrow E\cdot$ **sub** E
$E \rightarrow E\cdot$ **sup** E
$E \rightarrow E$ **sup** $E\cdot$

图4-51 文法(4-25)的LR(0)项目集

I_{11}在面临}和$时，还产生归约-归约冲突，该冲突发生在产生式

$E \rightarrow E$ **sub** E **sup** E
$E \rightarrow E$ **sup** E

之间。当我们看到已经在栈顶归约出 E **sub** E **sup** E 的输入后，I_{11}将出现在栈顶。如果我们用产生式(1)来解决归约-归约冲突，我们将把形如 E **sub** E **sup** E 的式子当成特例。使用这种消除二义性的规则，可以得到如图4-52所示的SLR语法分析表。

状态	action						goto
	sub	sup	{	}	c	$	E
0			s2		s3		1
1	s4	s5				acc	
2			s2		s3		6
3	r5	r5		r5		r5	
4			s2		s3		7
5			s2		s3		8
6	s4	s5		s9			
7	s4	s10		r2		r2	
8	s4	s5		r3		r3	
9	r4	r4		r4		r4	
10			s2		s3		11
11	s4	s5		r1		r1	

图4-52 文法(4-25)的语法分析表

构造一个分离特例语法结构的无二义文法是非常困难的。为了体会其困难程度，请读者为文法(4-25)构造一个等价的无二义文法，该文法将把形如 E **sub** E **sup** E 的表达式分离出来。

4.8.4 LR语法分析中的错误恢复

LR语法分析器在访问动作表时若遇到出错表项，就检测出一个错误，但它在访问转移表时决不会检测出错误。与算符优先语法分析器不同的是，LR语法分析器只要发现已扫描的输入出现一个不正确的后继就会立即报告错误。规范LR语法分析器在报告错误之前不会进行任何无效归约。SLR语法分析器和LALR语法分析器在报告错误前可能执行几步归约，但它们决不会把出错点的输入符号移进栈。

在LR分析中，可以采用下面的方法实现紧急方式的错误恢复：从栈顶开始退栈，直至发现在特定非终结符 A 上具有转移的状态 s 为止；然后丢弃零个或多个输入符号，直至找到符号 a为止，它是 A 的合法后随符号；接着，语法分析器把状态 $goto[s, A]$ 压进栈，并恢复正常分析。A 的选择可能不惟一，一般 A 应是代表主要程序结构的非终结符，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 **end**。 254

这种恢复方法实质是试图将含有语法错误的短语分离出来。语法分析器认为由A推导出的串含有一个错误，该串的一部分已经处理过，处理的结果是处于栈顶的状态序列，该串的剩余部分仍在输入中。语法分析器试图跳过该串的剩余部分，在输入中找到一个符号，它是 A 的合法后随符号。通过从栈中移出一些状态，跳过若干输入符号，并把 $goto[s, A]$ 推进栈，语法分析器假装已发现了 A 的一个实例，并恢复正常分析。

短语级恢复的实现是通过检查LR分析表的每个出错表项，并根据语言的使用情况确定最可能引起该错误的程序员最容易犯的错误，然后为该表项编一个适当的错误恢复例程。该例程大概会采用一种适合于相应出错表项的方式来修改栈顶符号和（或）第一个输入符号。

与算符优先语法分析器相比，设计LR语法分析器的专门错误处理例程要容易一些。尤其是，不必担心不正确的归约，LR语法分析器所执行的归约保证是正确的。于是，我们可以在语法分析表动作域的每个空白项填上一个指针，它指向编译器设计者为之设计的出错处理例程。该例程的动作可能包括从栈顶和(或)输入中插入或删除符号，改变或变换输入符号，就像算符优先语法分析器那样，所做的选择不应使LR语法分析器陷入死循环。保证至少有一个输入符号被移走或最终被移进，或者保证在到达输入的末尾时保证栈最终会缩短，这样的策略足以防止上述问题的发生。要避免从栈中弹出覆盖一个非终结符的状态，因为这种修改从栈中删掉了已经成功分析的一个结构。

例4.50 再次考虑下面的表达式文法：

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

图4-53给出了该文法的LR语法分析表，它是由图4-47修改而得，加上了错误诊断和恢复。我们还把某些出错表项改成了归约。这些修改会推迟错误的发现，多执行了一步或几步归约，但错误仍将在移进下一个符号前被捕获。图4-47中的其余空白表项已经改换成对出错例程的调用。

状态	action						goto
	id	+	*	(	)	$	E
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	acc	
2	s3	e1	e1	s2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	s3	e1	e1	s2	e2	e1	7
5	s3	e1	e1	s2	e2	e1	8
6	e3	s4	s5	e3	s9	e4	
7	r1	r1	s5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

图4-53 带有出错处理例程的LR语法分析表

出错处理例程如下。应该注意，这些动作及其所代表的错误同例4.32中的类似。但是，LR语法分析器中的e1通常由算符优先语法分析器的归约处理程序来处理。

e1：/*处于状态0，2，4，5时，要求输入符号为运算对象的首终结符，即 **id** 或左括号；但遇到的是+、*或$时，则调用此例程。*/
把一个假想的 **id** 压进栈，上面盖以状态3（状态0，2，4和5面临 **id** 的转移）。[⊖]
给出诊断信息："缺少运算对象"。

e2: /*处于状态0，1，2，4和5时，若遇到右括号，则调用此例程。*/
从输入中删除右括号。
给出诊断信息："右括号不配对"。

e3：/*处于状态1或6时，期望一个操作符，而遇到的是 **id** 或右括号，则调用此例程。*/
把+压入栈，盖以状态4。
给出诊断信息："缺少操作符"。

255
~
256

e4: /*处于状态6时期望操作符或右括号，但如果遇上$，则调用此例程。*/
把右括号压入栈，盖以状态9。
给出诊断信息："缺少右括号"。

以例4.32中的错误输入 **id** +) 为例，语法分析器进入的格局序列如图4-54所示。 □

状态	输入	错误信息和动作
0	**id**+)$	
0**id**3	+)$	
0*E*1	+)$	
0*E*1+4	)$	
0*E*1+4	$	"unbalanced right parenthesis"
		e1删掉右括号
0*E*1+4**id**3	$	"missing operand"
		e1把**id**3压入栈
0*E*1+4*E*7	$	
0*E*1	$	

图4-54 LR语法分析器的语法分析和错误恢复

4.9 语法分析器的生成器

本节说明怎样用语法分析器的生成器来辅助构造编译器的前端。我们将使用LALR语法分

[⊖] 注意，文法符号实际上没有放在栈中。把它们想像成在那里有助于我们想起状态所表示的符号。

析器的生成器Yacc作为讨论的基础，因为它实现了前两节讨论的许多概念，而且应用广泛。Yacc是“yet another compiler-compiler”的缩写，它是20世纪70年代初期语法分析器的生成器盛行时的产物，它的第一板本是S.C.Johnson设计的。Yacc在UNIX系统下可作为命令使用，并已帮助实现了几百个编译器。

4.9.1 语法分析器的生成器Yacc

一个翻译器可用Yacc按图4-55说明的方式构造出来。首先，准备一个包含翻译器的Yacc说明的文件，如`translate.y`。UNIX系统的命令

```
yacc translate.y
```

把文件`translate.y` 翻译成名为`y.tab.c`的C程序，使用的是算法4.13中描述的LALR方法。程序`y.tab.c`中包含用C语言编写的LALR语法分析器和其他用户准备的C语言例程。为节省空间，用4.7节描述的方法对LALR语法分析表进行了压缩。用命令 257

```
cc y.tab.c -ly
```

对`y.tab.c`进行编译，其中，1y表示使用LR分析程序的库。编译后得到目标程序`a.out`，它完成Yacc源程序指定的翻译。[⊖] 如果需要其他过程，它们可以和`y.tab.c`一起编译或装入，就像使用普通的C程序一样。

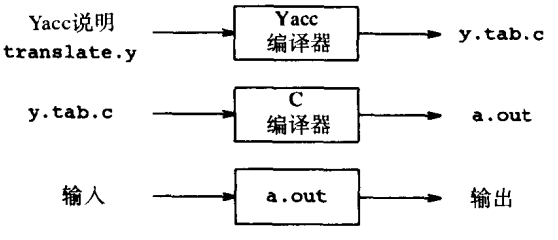


图4-55 用Yacc建立翻译器

Yacc源程序由3部分组成：

```
声明
%%
翻译规则
%%
用C语言编写的支持例程
```

例4.51 为说明怎样准备Yacc源程序，让我们构造一个简单的台式计算器，它读入一个算术表达式，计算并打印它的值。我们将从下面的算术表达式文法出发，建立一个台式计算器：

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \textbf{digit}$$

记号 **digit** 是0～9的单个数字。由该文法得到的Yacc台式计算器程序如图4-56所示。 □

(1) *声明部分*。Yacc程序的声明部分由可任选的两节组成。第一节处于分界符`%{`和`%}`之间，它是一些普通的C语言声明。第二部分和第三部分的翻译规则或过程所使用的所有临时声明都放在这里。在图4-56中，这一节只有一个包含语句 258

```
#include <ctype.h>
```

它使得C的预处理程序包含标准头文件`<ctype.h>`，该文件含有谓词`isdigit`。

声明部分的第二节是文法记号的声明，在图4-56中，语句

```
%token DIGIT
```

⊖ 名字ly依赖于具体的系统。

将DIGIT声明为记号。这一节说明的记号可用于 Yacc 说明的第二部分和第三部分。

```
%{
#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4-56 简单台式计算器的 Yacc 说明

259 (2) 翻译规则部分。这一部分位于第一个%%后面，用于放置翻译规则，每条规则由一个文法产生式和有关的语义动作组成。产生式集合

<left side> → <alt 1> | <alt 2> | · · · | <alt *n*>

在Yacc中写成：

```
<left side>    :   <alt 1>   {语义动作1}
               |   <alt 2>   {语义动作2}
          . . .
               |   <alt n>   {语义动作n}
               ;
```

在Yacc产生式中，单引号括起来的字符'c'是由终结符号c组成的记号；没有引号的字母数字串若没有声明为记号，则是非终结符。右部的各个选择之间用竖线隔开，最后一个右部的后面用分号，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 $$ 表示左部非终结符的属性值，而$i表示右部第*i*个文法符号（终结符或非终结符）的值。每当归约一个产生式时，就执行与之相关联的语义动作，所以语义动作一般是根据 $i的值计算 $$ 的值。在这个 Yacc 说明中，两个*E*产生式

$$E \rightarrow E + T \mid T$$

及与它们有关的语义动作写成

```
expr    :   expr '+' term    { $$ = $1 + $3; }
        |   term
        ;
```

注意，第一个产生式的非终结符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把右部的expr和term的值相加，把结果赋给左部的非终结符expr，作为它的值。我们省略了第二个产生式的语义动作，因为右部只有一个文法符号时，值的复写是默认的语义动作，即它的语义动作是{$$ = $1; }。

注意，我们将一个新的开始产生式

```
line    :   expr '\n'        { printf("%d\n", $1); }
```

加到了这个Yacc说明中。该产生式的意思是，这个台式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。 260

(3) 支持例程部分。Yacc说明的第三部分是一些用C语言编写的支持例程。必须提供名字为yylex()的词法分析器。其他的过程，如错误恢复例程，如果需要的话，也可以加上。

词法分析器yylex()返回二元组(记号，属性值)。返回的记号，如DIGIT，必须在Yacc说明的第一部分声明。属性值必须通过Yacc定义的变量yylval传给语法分析器。

图4-56的词法分析器是非常粗糙的。它用C函数getchar()每次读入一个输入字符，如果是数字字符，则将它的值存入变量yylval中，返回记号DIGIT，否则将字符本身作为记号返回。

4.9.2 用Yacc处理二义文法

现在我们修改这个Yacc说明，使台式计算器更加有用。首先，我们将允许台式计算器计算表达式序列，每行一个，还允许表达式之间有空白行。为做到这一点，将第一条规则改为:

```
lines   :   lines expr '\n'  { printf("%g\n", $2); }
        |   lines '\n'
        |
        ;
```

在Yacc中，第三行的空选择表示ϵ。

其次，我们将扩大表达式的类，使之包含由多个数字组成的数，包括操作符+，-(一元和二元)，*和/。说明这类表达式的最简单的方法是使用下面的二义文法:

$$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid (E) \mid -E \mid \textbf{number}$$

最终的Yacc说明如图4-57所示。

因为图4-57的 Yacc 说明中的文法是二义的，LALR算法将产生语法分析动作冲突。Yacc 会报告产生的语法分析动作冲突的数目。项目集和语法分析动作冲突的描述可以通过在调用 Yacc 时加-v选项来获得。该选项产生一个附加的文件 y.output，它包含语法分析时发现的项目集的核、由LALR算法产生的语法分析动作冲突的描述以及LR语法分析表的可读表示，该可读表示显示出语法分析动作冲突是怎样解决的。当Yacc报告它发现语法分析动作冲突时，明智的做法是建立和查阅文件y.output，以了解为什么会出现分析动作冲突以及它们是否已经被正确解决。 261

除非另有说明，否则 Yacc 将按下面两条规则解决所有语法分析动作的冲突：

1. 归约-归约冲突的解决是从冲突产生式中选择在 Yacc 说明中最先出现的那个产生式。因此，为了解决排版文法(4-25)的冲突，只要把产生式(1)放在产生式(3)前面就足够了。

2. 移动-归约冲突的解决是移进优先。这条规则正确地解决了悬空 else 二义所带来的移

动-归约冲突。

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* double type for Yacc stack */
%}

%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
lines       :  lines expr '\n'   { printf("%g\n", $2); }
            |  lines '\n'
            |  /* ε */
            ;
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4-57 更高级的台式计算器的Yacc说明

因为这些默认的规则并不总是编译器编写者所需要的，因而 Yacc 提供了解决移动-归约冲突的一般方法。在声明部分，我们可以为终结符指定优先级和结合规则。声明

```
%left '+' '-'
```

使得 + 和 - 具有同样的优先级和左结合规则。声明

```
%right '^'
```

使得操作符^为右结合的。还可以用声明限制二元操作符为不可结合的（即该操作符的两个相邻出现根本不能组合），如

```
%nonassoc '<'
```

记号的优先级按它们在声明部分出现的次序来确定，先出现的记号的优先级低，同一声明中的记号有相同的优先级。于是，图4-57中的声明

```
%right UMINUS
```

使得UMINUS的优先级高于前面5个终结符。

Yacc 通过为与冲突有关的每个产生式赋予优先级和结合规则来解决移动-归约冲突。如果Yacc 必须在移进输入符号 a 和按产生式 $A \rightarrow \alpha$ 归约这两个动作之间进行选择，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是左结合的时，执行归约动作，否则选择移进。

通常，产生式的优先级和它最右边的终结符号的优先级一致。大多数情况下，这种决定是合理的。例如，给定产生式

$$E \rightarrow E + E \mid E * E$$

若搜索符是+，归约产生式是 $E \rightarrow E+E$，那么归约优先，因为右部的+和搜索符具有同样的优先级，而+是左结合的；但如果搜索符是*，那么选择移进，因为搜索符的优先级高于这个产生式中+的优先级。

如果最右终结符不能给产生式以适当的优先级，我们可以通过给产生式附加标记

```
%prec <terminal>
```

来强制它的优先级，它的优先级和结合规则同这个标记的终结符一样。该终结符大概需在声明部分定义。Yacc 不会报告用优先级和结合规则解决了的移动-归约冲突。

262 ~ 263

该终结符可以是一个占位符，如图4-57中的UMINUS那样，它不由词法分析器返回，只是用来决定一个产生式的优先级。图4-57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，标记

```
%prec UMINUS
```

出现在产生式

```
expr    :   '-' expr
```

的后面，这使得该产生式中的一元减操作符的优先级高于其他任何操作符。

4.9.3 用Lex建立Yacc的词法分析器

Lex 产生的词法分析器可以用于 Yacc。Lex 库ll将提供名为 yylex()的驱动程序，这个名字就是 Yacc 所需要的词法分析器的名字。如果用 Lex 产生词法分析器，那么 Yacc 说明中第三部分的例程yylex( )应由语句

```
#include "lex.yy.c"
```

来代替。使用这条语句，程序 yylex()可以访问 Yacc中记号的名字，因为 Lex 的输出文件是作为 Yacc 输出文件的一部分被编译的，所以每个 Lex 动作都返回 Yacc 知道的终结符。

在 UNIX 系统中，如果 Lex 说明在文件 first.l中，Yacc 说明在 second.y 中，我们可以用命令

```
lex first.l
yacc second.y
cc y.tab.c -ly -ll
```

来获得所需的翻译器。

图4-58的 Lex 说明可用来取代图4-57中的词法分析器。最后一个模式是 \n|，因为.在 Lex 中匹配除换行以外的任何字符。

```
number      [0-9]+\.?|[0-9]*\.[0-9]+
%%
[ ]         { /* 跳过空格 */ }
{number}    { sscanf(yytext, "%lf", &yylval);
              return NUMBER; }
\n|.        { return yytext[0]; }
```

图4-58 图4-57中的 yylex() 的 Lex 说明

4.9.4 Yacc的错误恢复

在 Yacc 中，可以通过使用出错产生式的形式进行错误恢复。首先，由用户决定哪些“主要的”非终结符会与错误恢复有关，典型的选择是用于产生表达式、语句、程序块和过程的那些非终结符。然后用户把形如 $A \rightarrow$ **error** α 的出错产生式加到文法中，其中 A 是主要非终结符，α是文法符号串，也可能是空串，**error** 是 Yacc 保留字。Yacc 将从这样的说明产生语法分析器，并把出错产生式当作普通产生式来处理。

但是，当 Yacc 产生的语法分析器遇到错误时，会用一种特定的方式来处理项目集中含有出错产生式的状态。遇到错误时，Yacc从栈中弹出状态，直到发现栈顶状态的项目集含有形如 $A \rightarrow \cdot$**error** α的项目为止。然后语法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一样。

当α为ϵ时，立即归约为A并执行产生式 $A \rightarrow$ **error** 的语义动作（它可能是用户说明的错误恢复例程），然后语法分析器丢弃若干输入符号，直到发现一个能恢复正常处理的输入符号为止。

如果 α 非空，Yacc 在输入串上向前寻找能够归约为 α 的子串。如果α包含的都是终结符，那么它在输入上寻找这样的终结符串，并把它们移进栈，这时，语法分析器的栈顶为 **error** α，语法分析器把 **error** α 归约成 A，并恢复正常语法分析。

例如，出错产生式

stmt → **error**;

要求语法分析器看见错误时跳过下一个分号，好像该语句已经被看完一样。这个出错产生式的语义例程不需要处理输入，只需产生诊断信息并设置禁止生成目标代码的标记。

例4.52 图4-59给出了图4-57中带有下列出错产生式的 Yacc 台式计算器：

264 ~ 265

```
lines :  error '\n'
```

当输入行有语法错误时，语法分析器从栈中弹出符号，直至碰到一个含有移进记号 **error** 动作的状态为止。该例中，状态0是惟一的这种状态，因为它的项目包含

lines → · **error** '\n'

而且状态0总是在栈底。语法分析器把记号 **error** 移进栈，并跳过输入符号，直至发现换行符为止。这时语法分析器把换行符移进栈，把 **error** '\n' 归约成 *lines*，输出诊断信息“reenter last line:”。专门的 Yacc 例程 yyerrok用于将语法分析器恢复正常操作模式。 □

练习

4.1 考虑文法

$$S \rightarrow (\ L\ ) \mid a$$
$$L \rightarrow L\ ,\ S \mid S$$

a) 哪些是终结符、非终结符和开始符号?

b) 试建立下列句子的分析树。

i) (*a*, *a*)

ii) (*a*, (*a*, *a*))

iii) (*a*, ((*a*, *a*), (*a*, *a*)))

c) 试为(b)中的每个句子构造一个最左推导。

d) 试为(b)个的每个句子构造一个最右推导。

* e) 该文法产生的语言是什么?

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* 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 ("重新输入上一行 ; ")
                       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
        ;
%%
#include "lex.yy.c"
```

图4-59 包含错误恢复的台式计算器

4.2 考虑文法

$S \rightarrow aSbS \mid bSaS \mid \epsilon$

a) 为句子*abab*构造两个不同的最左推导，以证明该文法是二义的。

b) 试为*abab*构造相应的最右推导。

c) 试为*abab*构造相应的分析树。

* d) 该文法产生的语言是什么?

4.3 考虑文法

bexpr → *bexpr* **or** *bterm* | *bterm*
bterm → *bterm* **and** *bfactor* | *bfactor*
bfactor → **not** *bfactor* | (*bexpr*) | **true** | **false**

a) 试为句子 **not (true or false)**构造分析树。

b) 试证明该文法产生所有的布尔表达式。

* c) 该文法是二义的吗? 为什么?

4.4 考虑文法

$R \rightarrow R\ '|'\ R \mid RR \mid R^* \mid (R) \mid a \mid b$

注意，第一个竖线是符号“或”，不是候选式之间的分隔符。

a) 试证明该文法产生字母表{a，b}上的所有正规表达式。

b) 试证明该文法是二义的。

* c) 构造一个等价的无二义文法，它赋予操作符*、并置和 | 的优先级和结合规则与3.3节中的定义相同。

266
~
267

d) 按上面两个文法构造句子 $a \mid b^*c$ 的分析树。

4.5 下面的if-then-else语句文法试图消除悬空 else 的二义性，试证明该文法仍然是二义的。

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

* 4.6 试为下面每种语言设计一个文法，并说明哪些语言是正规的?

a) 每个0的后面至少跟有一个1的所有0和1的串。

b) 0和1的个数相等的0和1的串。

c) 0和1的个数不相等的0和1的串。

d) 不含011子串的0和1的串。

e) 形如 xy 且 $x \neq y$ 的0和1的串。

f) 形如 xx 的0和1的串。

4.7 为下面每种语言的表达式构造一个文法。

a) Pascal

b) C

c) Fortran 77

d) Ada

e) Lisp

4.8 为练习4.7中每种语言的语句构造一个无二义性文法。

4.9 我们可以在文法产生式的右部使用类似正规表达式的操作符。方括号可以用来表示产生式的可选部分。例如，我们可以用下面的产生式来表示可选的 **else** 子句。

$\textit{stmt} \rightarrow \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}\ [\ \textbf{else}\ \textit{stmt}\ ]$

一般地，$A \rightarrow \alpha[\beta]\gamma$ 等价于两个产生式 $A \rightarrow \alpha\beta\gamma$ 和 $A \rightarrow \alpha\gamma$。

花括号可以用来表示一个出现零次或多次的短语，例如，

$\textit{stmt} \rightarrow \textbf{begin}\ \textit{stmt}\ \{\ ;\ \textit{stmt}\ \}\ \textbf{end}$

表示处于 **begin** 和 **end** 之间的由分号分隔的语句表。一般地，$A \rightarrow \alpha\{\beta\}\gamma$ 等价于 $A \rightarrow \alpha B\gamma$ 和 $B \rightarrow \beta B \mid \epsilon$。

268

在某种意义上，[β] 代表正规表达式 $\beta \mid \epsilon$，而{β}代表 β^*。我们可以推广这些表示，

以允许文法符号的任何正规表达式都可以出现在产生式的右部：

a) 修改上面的 *stmt* 产生式，使得以分号终止的语句表可以出现在产生式右部。

b) 试给出一组上下文无关的产生式，它们和 $A \rightarrow B^*a\ (C|D)$ 产生同样的串集。

c) 说明如何用一组有穷的上下文无关产生式来代替任何产生式 $A \rightarrow r$，其中 r 是正规表达式。

4.10 下列文法为单个标识符产生声明：

stmt → **declare id** *option_list*
option_list → *option_list option* | ϵ
option → *mode* | *scale* | *precision* | *base*
mode → **real** | **complex**
scale → **fixed** | **floating**
precision → **single** | **double**
base → **binary** | **decimal**

a) 说明怎样将该文法推广到允许 n 个选项 A_i，其中$1 \leqslant i \leqslant n$，每个选项或者为 a_i，或者为b_i。

b) 上面的文法容许冗余或者矛盾的声明，如

```
declare zap real fixed real floating
```

我们可以坚持认为该语言的语法禁止这样的声明，但在此我们只考虑语法是否正确。于是，就有有穷数目的语法正确的记号序列。很明显，这些声明是上下文无关的，是正规集。试为带有 n 个选项的声明构造一个文法，使得每个选项至多出现一次。

** c) 试证明(b)部分的文法至少有 2^n 个符号。

d) (c)是否说明通过语言的语法定义来强制声明选项中的非冗余和无矛盾是可行的？

4.11 a) 消除练习4.1中文法的左递归。

b) 为(a)的文法构造预测语法分析器。给出该语法分析器在分析练习4.1(b)中的句子时的行为。

4.12 试为练习4.2中的文法构造一个带回溯的递归下降语法分析器。能否为这个文法构造预测语法分析器？

4.13 下面的文法产生除空串以外的所有长度为偶数的a的串。

$S \rightarrow aSa \mid aa$

a) 试为该文法构造一个带回溯的递归下降语法分析器，要求在S产生式的两个候选式aa和aSa中首先考虑aSa。证明：S所对应的过程可以成功地分析2，4，8个a的串，但6个a的串却不行。 269

* b) 你的语法分析器能识别什么样语言？

4.14 为练习4.3的文法构造一个预测语法分析器。

4.15 为练习4.4中正规表达式的无二义文法构造一个预测语法分析器。

* 4.16 证明：左递归文法一定不是LL(1)文法。

* 4.17 证明：LL(1)文法一定不是二义的。

4.18 证明：一个没有 ϵ 产生式的文法，只要它的每个非终结符的各个候选式以不同的终结符开始，那么它就是LL(1)文法。

4.19 如果不存在形如 $S \overset{*}{\Rightarrow} wXy \overset{*}{\Rightarrow} wxy$ 的推导，则文法符号 X 是无用的，即 X 不会出现

在某个句子的推导中。

a) 试编写一个算法，从文法中删除包含无用符号的产生式。

b) 将你的算法应用到下面的文法上：

$S \rightarrow 0 \mid A$
$A \rightarrow AB$
$B \rightarrow 1$

4.20 我们说一个文法是无 ϵ 的，如果它没有 ϵ 产生式，或者只有一个 ϵ 产生式 $S \rightarrow \epsilon$，而且开始符号 S 不出现在任何产生式的右部。

a) 试编写一个算法，把给定的文法转换成等价的无 ϵ 的文法。（提示：首先确定所有能够产生空串的非终结符。）

b) 把你的算法应用于练习4.2中的文法。

4.21 单产生式是右部只有一个非终结符的产生式。

a) 试编写一个算法，把文法转换成等价的不含单产生式的文法。

b) 把你的算法应用于表达式文法(4-10)。

4.22 对任何非终结符 A，不存在形如 $A \stackrel{+}{\Rightarrow} A$ 推导的文法是无环的文法。

a) 试编写一个算法，把文法转换成等价的无环文法。

b) 把你的算法应用于文法 $S \rightarrow SS \mid (S) \mid \epsilon$。

4.23 a) 用练习4.1的文法构造(*a*, (*a*, *a*))的一个最右推导，并给出每个右句型的句柄。

270 b) 试给出与(a)的最右推导对应的移动-归约分析器的步骤。

c) 试给出在(b)的移动-归约分析过程中，自底向上构造分析树的步骤。

4.24 图4-60给出了练习4.1中文法的算符优先关系，利用这些优先关系分析练习4.1(b)的句子。

4.25 试给出与图4-60中的表对应的算符优先函数。

	a	(	)	,	$
a			$\cdot>$	$\cdot>$	$\cdot>$
(	$<\cdot$	$<\cdot$	$\doteq$	$<\cdot$	
)			$\cdot>$	$\cdot>$	$\cdot>$
,	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$	
$	$<\cdot$	$<\cdot$			

图4-60 练习4.1中文法的算符优先关系

4.26 给定一个算符文法，包括那些包含许多不同非终结符的文法，有一种方法可以自动地产生它的算符优先关系。对非终结符 A，我们将 *leading*(*A*)定义为从 A 导出的某个串的最左终结符 a 的集合，将 *trailing*(*A*) 定义为从A导出的某个串的最右终结符的集合。对终结符 a 和 b，如果存在一个形如 $\alpha a \beta b \gamma$ 的右部，其中 β 或者为空，或者是单个非终结符，α和γ是任意的文法符号串，则我们说 $a \doteq b$；如果存在一个形如 $\alpha a A \beta$ 的右部，而且 b 在 *leading*(*A*)中，则我们说 $a <\cdot b$；如果存在一个形如 $\alpha A b \beta$ 的右部，而且 a 在 *trailing*(*A*) 中，则我们说 $a \cdot> b$。这两种情况下α和β都是任意的串。而且，如果 b 在 *leading*(*S*)中，则 $\$ <\cdot b$；如果 a 在 *trailing*(*S*)中，则 $a \cdot> \$$，其中，S 是开始符号。

a) 对练习4.1中的文法，计算 S 和 T 的 *leading* 和 *trailing*。

b) 试验证图4-60中的优先关系是从该文法导出的。

4.27 试给下列文法构造算符优先关系。

a) 练习4.2的文法。

b) 练习4.3的文法。

c) 表达式文法(4-10)。

4.28 试给正规表达式构造一个算符优先语法分析器。 271

4.29 一个文法称为（惟一可逆的）*算符优先文法*，如果它是一个算符文法，而且任意两个右部的终结符样式都不相同，在每对终结符之间用练习4.26的方法至多产生一种优先关系。练习4.27的文法中哪一个是算符优先文法。

4.30 一个文法称为 Greibach 范式(GNF)文法，如果它是无ϵ的，而且每个产生式（$S\rightarrow\epsilon$除外）均形如$A\rightarrow a\alpha$，其中，a是终结符，α是非终结符串，也可能为空。

** a) 试编写一个算法，将给定文法转换成 Greibach 范式文法。

b) 将你的算法应用到表达式文法(4-10)上。

* 4.31 试证明：每个文法均可以被转换成等价的算符文法。提示：首先将其转换成Greibach范式。

* 4.32 试证明：每个文法均可以被转换成一个算符文法，它的每个产生式都是下列格式之一：

$A \rightarrow aBcC \quad A \rightarrow aBb \quad A \rightarrow aB \quad A \rightarrow a$

如果ϵ在该语言中，那么还有产生式$S\rightarrow\epsilon$。

4.33 考虑下面的二义文法

$$S \rightarrow AS \mid b$$
$$A \rightarrow SA \mid a$$

a) 试构造该文法的 LR(0) 项目集规范族。

b) 试构造一个 NFA，它的状态是(a)的 LR(0)项目。证明：从该 NFA 用子集构造法构造的 DFA 和该文法的 LR(0)项目集规范族的转移图是一致的。

c) 试构造该文法的 SLR 语法分析表。

d) 试给出 SLR 语法分析器在输入 *abab* 上的动作。

e) 试构造规范的 LR 语法分析表。

f) 试用LALR算法4.11构造语法分析表。

g) 试用LALR算法4.13构造语法分析表。

4.34 试为练习4.3的文法构造SLR语法分析表。

4.35 考虑下面的文法

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T\ F \mid F$$
$$F \rightarrow F^* \mid a \mid b$$

272

a) 试为该文法构造SLR语法分析表。

b) 试构造LALR语法分析表。

4.36 根据4.7节的方法压缩练习4.33、4.34和4.35中构造的语法分析表。

4.37 a) 证明下面的文法是 LL(1)文法，但不是 SLR(1)文法。

$$S \rightarrow AaAb \mid BbBa$$
$$A \rightarrow \epsilon$$
$$B \rightarrow \epsilon$$

** b) 证明所有 LL(1)文法都是 LR(1)文法。

* 4.38 证明 LR(1)文法都是无二义的。

4.39 证明下面的文法是 LALR(1)文法，但不是 SLR(1)文法。

$S \rightarrow Aa \mid bAc \mid dc \mid bda$
$A \rightarrow d$

4.40 证明下面的文法是LR(1)文法，但不是LALR(1)文法。

$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$
$A \rightarrow d$
$B \rightarrow d$

*4.41 考虑下式定义的文法族 G_n：

$S \rightarrow A_i b_i \qquad 1 \leqslant i \leqslant n$
$A_i \rightarrow a_j A_i \mid a_j \quad 1 \leqslant i, j \leqslant n$ 且 $j \neq i$

a) 试证明：G_n具有 $2n^2-n$ 个产生式和 2^n+n^2+n 个 LR(0)项目集。该结果说明 LR 语法分析器的大小同文法的大小相比有多大？

b) G_n是 SLR(1)文法吗？

c) G_n是 LALR(1)文法吗？

4.42 试编写一个算法，为文法中的每一个非终结符A计算集合$\{B \mid A \overset{*}{\Rightarrow} B\alpha$，其中 B 是非终结符，α 是文法符号串$\}$。

4.43 试编写一个算法，为文法中的每一个非终结符A计算集合$\{a \mid A \overset{*}{\Rightarrow} aw$，其中 a 是终结符，w 是终结符号串，而且在最后一步推导中不能用ϵ 产生式$\}$。

4.44 试为练习4.4的文法构造SLR语法分析表，语法分析动作冲突的解决要保证正规表达式能以正常的方式进行分析。

273

4.45 试为悬空 else 文法(4-7)构造一个 SLR 语法分析器，把 *expr* 看作一个终结符，用通常的方法解决分析动作冲突。

4.46 a) 为下面的文法构造SLR语法分析表，解决分析动作冲突，使得能以与图4-52的LR语法分析器同样的方式分析表达式。

$E \rightarrow E$ **sub** $R \mid E$ **sup** $E \mid \{ E \} \mid c$
$R \rightarrow E$ **sup** $E \mid E$

b) 在该 SLR 语法分析表的构造过程中产生的每个归约-归约冲突，是否都能通过变换文法而改成移动-归约冲突？

*4.47 试为排版文法(4-25)构造一个等价的 LR 文法，它能把形如 E **sub** E **sup** E 的表达式处理成一种特例。

*4.48 考虑下面 n 个二元中缀操作符的二义文法：

$E \rightarrow E\ \theta_1\ E \mid E\ \theta_2\ E \mid \cdots \mid E\ \theta_n\ E \mid (E) \mid$ **id**

假设所有的操作符都是左结合的，而且如果 $i > j$，则 θ_i 的优先级高于 θ_j。

a) 试为该文法构造 SLR 项目集。共有多少个项目集？是 n 的函数吗？

b) 试为该文法构造 SLR 语法分析表，并用4.7节的列表表示对其进行压缩。该表示中使用的列表的总长度是多少？是 n 的函数吗？

c) 分析 **id** θ_i **id** θ_j **id** 需用多少步？

*4.49 对下面的无二义文法重做练习4.48。

$E_1 \rightarrow E_1\ \theta_1\ E_2 \mid E_2$
$E_2 \rightarrow E_2\ \theta_2\ E_3 \mid E_3$

$$
\begin{aligned}
E_n &\rightarrow E_n\ \theta_n\ E_{n+1} \mid E_{n+1} \\
E_{n+1} &\rightarrow (E_1) \mid \mathbf{id}
\end{aligned}
$$

从练习4.48和4.49的答案来看，等价的二义文法和无二义文法的语法分析器的效率有什么差别？构造语法分析器的效率又有什么差别？

4.50 试编写一个Yacc程序，其输入为算术表达式，输出为相应的后缀表达式。 274

4.51 试编写一个计算布尔表达式的Yacc“台式计算器”程序。

4.52 试编写一个Yacc程序，其输入为正规表达式，输出为正规表达式的分析树。

4.53 用例4.20、例4.32和4.50中的预测语法分析器，算符优先语法分析器和LR语法分析器，试描绘出它们在下面的错误输入上的分析动作：

a) (**id** + (***id**)

b) * + **id**) + (**id** *

* 4.54 试为下面的文法构造具有纠错功能的算符优先语法分析器和LR语法分析器：

$$
\begin{aligned}
\mathit{stmt} &\rightarrow \mathbf{if}\ e\ \mathbf{then}\ \mathit{stmt} \\
&\mid \mathbf{if}\ e\ \mathbf{then}\ \mathit{stmt}\ \mathbf{else}\ \mathit{stmt} \\
&\mid \mathbf{while}\ e\ \mathbf{do}\ \mathit{stmt} \\
&\mid \mathbf{begin}\ \mathit{list}\ \mathbf{end} \\
&\mid s \\
\mathit{list} &\rightarrow \mathit{list}\ ;\ \mathit{stmt} \\
&\mid \mathit{stmt}
\end{aligned}
$$

4.55 用下面的产生式代替练习4.54中文法的 *list* 产生式可以将其改造成LL文法。

$$
\begin{aligned}
\mathit{list} &\rightarrow \mathit{stmt}\ \mathit{list}' \\
\mathit{list}' &\rightarrow ;\ \mathit{stmt} \mid \epsilon
\end{aligned}
$$

试为修正后的文法构造一个具有纠错功能的预测语法分析器。

4.56 试给出你为练习4.54和4.55构造的语法分析器在下列错误输入上的分析动作。

a) **if** *e* **then** *s* ; **if** *e* **then** *s* **end**

b) **while** *e* **do begin** *s* ; **if** *e* **then** *s* ; **end**

4.57 用分号和**end**作同步记号，试为练习4.54和4.55的文法构造带有紧急方式错误恢复的预测语法分析器和LR语法分析器。给出你设计的语法分析器在练习4.56的错误输入上的分析动作。

4.58 在4.6节中，我们提出了一种面向图的方法，它能在算符优先语法分析器的归约动作中，确定从栈中弹出的串的集合。

* a) 试给出一个算法，找出表示所有这种串的正规表达式。

b) 试给出一个算法，确定这种串的集合是有穷的还是无穷的，如果有穷则列出它们。 275

c) 将你的算法从(a)到(b)应用到练习4.54的文法上。

** 4.59 对图4-18、图4-28和图4-53中的纠错语法分析器，我们要求每一次纠错最终至少从输入中移走一个符号或者在到达输入的末尾时使栈缩短。但所选的纠正动作不一定都会立即从输入中移走一个符号，你能证明图4-18、图4-28和图4-53中的语法分析器不可能进入死循环吗？提示：对算符优先语法分析器，即使存在错误，栈中相邻的终结符之间的关系也是$\lessdot$·。对 LR 语法分析器，即使存在错误，栈中仍将包含活前缀。

** 4.60 试给出一个算法，检测预测语法分析表、算符优先语法分析表和LR语法分析表中的不可达表项。

4.61 当栈顶状态是4或5（+和*分别在栈顶时）以及下一个输入是+或*时，图4-53的 LR 语法分析器以完全相同的方式处理这四种情况：调用例程 e1，将一个 **id** 插在它们之间。我们可以很容易地想像出一个LR语法分析器，它按同样的方式处理包含所有算术操作符的表达式：将 **id** 插入到两个相连的操作符之间。在某些语言（如PL/1或C，但不是 Fortran 或 Pascal）中，最好以一种特殊方式处理/在栈顶且*是下一输入的情况。为什么？纠错程序将采取什么样的合理动作？

4.62 一个文法称为Chomsky范式(CNF)文法，如果它是无ϵ的，而且每个非ϵ产生式均形如$A \rightarrow BC$或$A \rightarrow a$。

** a) 试编写一个算法，将给定文法转换成一个等价的 Chomsky 范式文法。

b) 将你的算法应用到表达式文法(4-10)上。

4.63 给定一个Chomsky范式文法G和一个输入串$w = a_1 a_2 \cdots a_n$，试编写一个算法，确定w是否在$L(G)$中。提示：用动态规划方法填一张$n \times n$的表T，其中$T[i,j] = \{A \mid A \stackrel{*}{\Rightarrow} a_i a_{i+1} \cdots a_j\}$。输入串 w 在 $L(G)$ 中，当且仅当 S 在 $T[1,n]$中。

* 4.64 a) 给定一个 Chomsky 范式文法 G，说明怎样将单个插入、删除和修改错误的产生式
276 填加到该文法中，使得扩大后的文法能产生所有可能的记号串。

b) 试修改练习4.63的算法，使得对于任何串 w，该算法使用最少数目的出错产生式即可找到 w 的一个分析。

4.65 试采用例4.50中的错误恢复机制为算术表达式编写一个 Yacc 语法分析器。

参考文献注释

有重大影响的Algol 60报告(Naur[1963])使用Backus-Naur范式(BNF)来定义了一个较大的程序设计语言的语法。BNF范式和上下文无关文法的等价很快引起了人们的注意，而且形式语言理论在20世纪60年代得到了极大的关注。Hopcroft and Ullman[1979]介绍了这一领域的基础。

随着上下文无关文法的发展，语法分析方法变得更加系统化。出现了一些可以分析任何上下文无关文法的通用技术。练习4.63中提到的动态规划技术是最早出现的分析技术之一，这种技术是由J.Cocke、Younger[1967]和Kasami[1965]分别独立发现的。Earley[1970]在他的博士论文中也提出了一种分析所有上下文无关文法的通用算法。Aho and Ullman[1972b，1973a]详细讨论了这些方法以及其他一些分析方法。

已经有很多不同的语法分析技术被应用在编译器中。Sheridan[1959]描述了Fortran编译器的原始版本所使用的一种分析方法，为了能够分析表达式，这种方法在操作数的两边引入了额外的括号。算符优先的思想和优先函数的使用来自Floyd[1963]。在20世纪60年代，人们提出了很多自底向上分析策略，主要包括简单优先法(Wirth and Weber[1966])、有界上下文法(Floyd[1964]，Graham[1964])、混合策略优先法(McKeeman, Horning and Wortman[1970])以及弱优先法(Ichbiah and Morse[1970])。

递归下降法和预测分析法在实践中得到了广泛应用。由于递归下降分析法的灵活性，它用在了很多早期的编译器编写(compiler-writing)系统中，譬如META(Schorre[1964])和TMG(McClure[1965])。练习4.13的解答和这种分析技术的一些理论可以在Birman and

Ullman[1973]中找到。Pratt[1973]提出了一种自顶向下的算符优先语法分析法。

Lewis and Stearns[1968]研究了LL文法，Rosenkrantz and Stearns[1970]则研究了LL文法的性质。Knuth[1971a]对预测语法分析器进行了深入研究。Lewis，Rosenkrantz and Stearns[1976]描述了预测语法分析器在编译器中的应用。Foster[1968]、Wood[1969]、Stearns[1971]和Soisalon-Soininen and Ukkonen[1979]提出了一种将文法转换成LL(1)形式的算法。

LR文法和语法分析器首先是由Knuth[1965]提出的，并且他还描述了规范LR语法分析表的构造。直到Korenjak[1969]证明使用LR方法可以构造大小合理的语法分析器；人们才认为这种 277
方法是实用的。当DeRemer[1969，1971]设计了比Korenjak的方法更加简单的SLR和LALR方法时，LR技术就成为语法分析器自动生成器的一种选择。如今，LR语法分析器的生成器在编译器构造领域非常流行。

有很多研究工作深入到了LR语法分析器的工程中。二义文法在LR语法分析过程中的应用要归功于Aho，Johnson, and Ullman[1975]和Earley[1975a]。Anderson，Eve, and Horning[1973]、Aho and Ullman[1973b]、Demers[1975]、Backhouse[1976]、Joliat[1976]、Pager[1977b]、Soisalon-Soininen[1980]和Tokuda[1981]中讨论了怎样消除用单产生式进行的归约。

LaLonde[1971]、Anderson, Eve and Horning[1973]、Pager[1977a]、Kristensen and Madsen[1981]、DeRemer and Pennello[1982]、Park,Choe, and Chang[1985]提出了计算LALR(1)搜索符集的技术，Park,Choe, and Chang[1985]还提供了一些试验比较。

Aho and Johnson[1974]给出了LR分析法的综述，并讨论了Yacc语法分析器生成器用到的一些算法，包括用出错产生式来进行错误恢复。Aho and Ullman [1972b，1973a]扩展了LR语法分析法和它的理论基础。

人们已经提出了很多语法分析器的错误恢复技术。Ciesinger[1979]和Sippu[1981]对错误恢复技术进行了综述。Irons[1963]提出了一种基于文法的语法错误恢复技术。Wirth[1968]用出错产生式来处理PL360编译器中的错误。Leinius[1970]提出了短语级恢复策略。Aho and Peterson[1972]表明通过将出错产生式和上下文无关文法的一般分析算法相结合可以获得全局最小开销的错误恢复。Mauney and Fischer[1982]通过使用Graham, Harrison, and Ruzzo[1980]的语法分析技术将这种思想扩展到LL和LR语法分析器的局部最小开销的修复中。Graham and Rhodes[1975]讨论了优先分析中的错误恢复技术。

Horning[1976]讨论了好的出错信息应当具有的性质。Sippu and Soisalon-Soininen[1983]对Helsinki语言处理器(Räihä et al. [1983])中的错误恢复技术与Pennello and DeRemer [1978]的“前向移动”(forward move)恢复技术、Graham，Haley, and Joy[1979]的LR错误恢复技术以及Pai and Kieburtz[1980]的“全局上下文”(global context)恢复技术的性能进行了比较。

Conway and Maxwell[1963]、Moulton and Muller[1967]、Conway and Wilcox[1973]、Levy[1975]、Tai[1978]和Röhrich[1980]讨论了语法分析过程中的错误校正。Aho and Peterson[1972]中含有对练习4.63的解答。 278

第5章　语法制导翻译

本章扩展了2.3节的内容，主要研究上下文无关文法所产生的语言的翻译。通过把属性附加到代表语法结构的文法符号上，我们可以将语义信息和程序设计语言的结构联系起来。属性的值是用与文法产生式相关联的“语义规则”来计算的。

把语义规则同产生式联系起来要涉及两个概念，即语法制导定义和翻译模式(translation scheme)。语法制导定义是关于语言翻译的高层次规格说明，它隐蔽了许多具体实现细节，使用户不必显式地说明翻译发生的顺序。翻译模式则指明了语义规则的计算顺序，以便说明某些实现细节。在第6章的语义检查（尤其是类型的确定）以及第8章的中间代码生成中，我们都将使用这两个概念。

从概念上讲，不论是语法制导定义还是翻译模式，都要首先对输入符号串进行语法分析，建立分析树，然后根据需要遍历分析树，并在分析树的节点处计算语义规则（见图5-1）。语义规则的计算可以生成代码、在符号表中保存信息、发出错误信息或完成其他活动。这样，对输入符号串的翻译过程就是对语义规则求值的过程。

输入符号串 ⟶ 分析树 ⟶ 依赖图 ⟶ 语义规则的计算顺序

图5-1　从概念的角度看语法制导翻译过程

实际实现时，并不一定完全按照图5-1中的步骤进行。在某些情况下，语法制导定义可以在单遍扫描中实现，即在语法分析期间计算语义规则，而不用显式地构造分析树或属性间的依赖图。因为单遍扫描的实现方式对提高编译的效率非常重要，所以本章主要致力于研究这种情况。有一个重要的子类称为“L-属性”定义，实际上它包含了所有不必显式构造分析树即可完成的翻译。 279

5.1　语法制导定义

语法制导定义是对上下文无关文法的推广，其中每个文法符号都有一个相关的属性集。属性分为两个子集，分别称为该文法符号的综合属性和继承属性。如果把分析树中对应该文法符号的节点看成是一条记录，其中包含若干存储信息的域，那么属性就相当于一个域的名字。

属性可以代表任何对象：字符串、数字、类型、内存单元或其他对象。分析树节点上属性的值由该节点所用产生式的语义规则来定义。节点的综合属性值是通过分析树中其子节点的属性值计算出来的；而继承属性值则是由该节点的兄弟节点及父节点的属性值计算出来的。

语义规则建立了属性间的依赖关系，它们可以用图来表示。从依赖图中我们可以得到语义规则的计算顺序。语义规则的计算定义了输入符号串在分析树节点上的属性值。语义规则还可能具有副作用，如打印一个值或修改全局变量等。当然，具体实现时可能不必显式地构造分析树或依赖图，只需对每个输入串产生相同的输出即可。

我们将每个节点都带有属性值的分析树称为注释分析树，而计算节点属性值的过程则称为注释或装饰分析树。

5.1.1 语法制导定义的形式

在语法制导定义中，每个产生式 $A \rightarrow \alpha$ 都有一个形如 $b := f(c_1, c_2, \cdots, c_k)$ 的语义规则集合与之相关联，其中 f 是函数，并且满足下面两种情况之一：

1. b 是 A 的一个综合属性，且 $c_1, c_2, \cdots, c_k$ 是该产生式文法符号的属性；

2. b 是产生式右部某个文法符号的一个继承属性，且 $c_1, c_2, \cdots, c_k$ 也是该产生式文法符号的属性。

对这两种情况都称为属性 b 依赖于属性 $c_1, c_2, \cdots, c_k$。属性文法是一个语法制导定义，其中
280 语义规则中的函数不能有副作用。

语义规则中的函数通常被写成表达式的形式。有时，语法制导定义中某个规则的目的就是为了产生副作用。这种语义规则一般被写成过程调用或程序段。在这种情况下，可以把它看作是定义产生式左部非终结符的虚综合属性值，但这种语义规则中的虚属性和符号 := 都不给出来。

例5.1 图5-2是一个台式计算器程序的语法制导定义。该定义将一个整数值综合属性 *val* 与每个非终结符 E，T 和 F 联系起来。对每个以 E，T 和 F 为左部的产生式，语义规则从产生式右部非终结符的 *val* 值计算出产生式左部非终结符的 *val* 值。

产生式	语义规则
$L \rightarrow E$ **n**	*print*(*E.val*)
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val \times F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow$ **digit**	$F.val :=$ **digit**.*lexval*

图5-2 一个简单台式计算器的语法制导定义

记号 **digit** 具有综合属性 *lexval*，其值由词法分析器提供，而产生式 $L \rightarrow E$**n**（其中 L 是文法开始符号）所对应的语义规则只是一个打印 E 所产生的算术表达式的值的过程，我们可以认为该规则为非终结符 L 定义了一个虚属性。图4-56中给出了该台式计算器的 Yacc 规格说明，在LR 语法分析过程中使用规格说明进行翻译。 □

在语法制导定义中，我们假设终结符只具有综合属性，因为定义没有为终结符提供任何语义规则。正如在3.1节所讨论的那样，终结符的属性值通常由词法分析器提供。此外，假设开始符号不具有任何继承属性，除非另有说明。

5.1.2 综合属性

综合属性在实践中具有广泛的应用。我们将仅仅使用综合属性的语法制导定义称为 *S* 属性定义。在 *S* 属性定义的分析树中，我们可以自底向上地在每个节点用语义规则来计算综合属性
281 值，即从叶节点到根节点进行计算。5.3节将描述如何修改LR语法分析器生成器使其能够实现基于 LR 文法的 *S* 属性定义。

例5.2 例5.1的 *S* 属性定义详细说明了一个台式计算器，它读入包含数字、括号、运算符 + 和 * 的算术表达式（表达式后面跟有换行符 **n**），并打印表达式的值。例如，输入表达式 3*5+4，后跟一个换行符 **n**，该程序将打印出值19。图5-3是输入 3*5+4**n** 的注释分析树，树根节点输出的是其第一个子节点的 *E.val* 值。

为弄清属性值是如何计算的，先考虑最左边最底层的内部节点，它所对应的产生式是 $F \rightarrow$ **digit**。相应的语义规则为 *F.val* := **digit**.*lexval*。因为其子节点的 **digit**.*lexval* 值是3，所以该节点的属性 *F.val* 的值也为3。同样地，F 节点的父节点的属性 *T.val* 的值也为 3。

现在考虑产生式 $T \rightarrow T * F$ 所对应的节点，其属性 *T.val* 的值由产生式 $T \rightarrow T_1 * F$ 的语义规则 $T.val := T_1.val \times F.val$ 来定义。对此节点应用语义规则：从左子节点可以得到 $T_1.val$ 的值为3，

从右子节点可以得到 *F.val* 的值为5，故求得在这一节点 *T.val* 的值为15。

与产生式*L*→*E*n 相关联的语义规则打印由 *E* 生成的表达式的值。 □

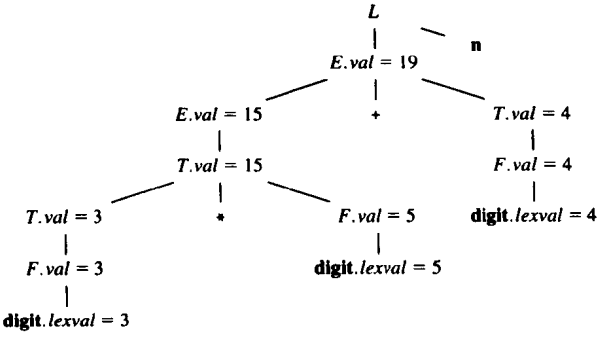


图5-3 3*5+4n的注释分析树

282

5.1.3 继承属性

在分析树中，一个节点的继承属性值是由该节点的父节点和（或）兄弟节点的属性决定的。用继承属性来表示程序设计语言上下文结构的依赖性是很方便的。例如，我们可以使用继承属性来跟踪一个标识符，看它是出现在赋值号的左边还是右边来确定是需要它的地址还是它的值。虽然我们总是可以只用综合属性来改写语法制导定义，但是使用带有继承属性的语法制导定义会更为自然。

在下面的例子中，继承属性将类型信息提供给声明中的各个标识符。

例5.3 在图5-4的语法制导定义中，非终结符 *D* 所产生的声明是由关键字 **int** 或 **real** 后跟一个标识符表所组成的。非终结符 *T* 有一个综合属性 *type*，其值由声明中的关键字来确定。与产生式 *D*→*TL* 相关联的语义规则 *L.in* := *T.type* 用来将继承属性 *L.in* 置为所声明的类型。语义规则使用继承属性 *L.in* 把该类型信息沿分析树向下传递。与 *L* 产生式相关联的语义规则调用过程 *addtype*，该过程将各标识符的类型填加到符号表的相应表项中（由属性 *entry* 指向）。

图5-5给出了语句 **real** $\mathbf{id}_1$，$\mathbf{id}_2$，$\mathbf{id}_3$ 的注释分析树。3个 *L* 节点的 *L.in* 值分别给出了标识符 $\mathbf{id}_1$，$\mathbf{id}_2$ 和 $\mathbf{id}_3$ 的类型。这些值是按下面的方法确定的：首先计算根的左子节点的属性值 *T.type*，然后在根的右子树中自上而下计算3个 *L* 节点的 *L.in* 值。在每个 *L* 节点处，我们还调用过程 *addtype*，它在符号表中记下相应的标识符类型（即该节点的右子节点上的标识符类型为 real）。 □

产生式	语义规则
$D \rightarrow TL$	*L.in* := *T.type*
$T \rightarrow$ **int**	*T.type* := *integer*
$T \rightarrow$ **real**	*T.type* := *real*
$L \rightarrow L_1$, **id**	L_1*.in* := *L.in* *addtype*(**id**.*entry*, *L.in*)
$L \rightarrow$ **id**	*addtype*(**id**.*entry*, *L.in*)

图5-4 带有继承属性 *L.in* 的语法制导翻译

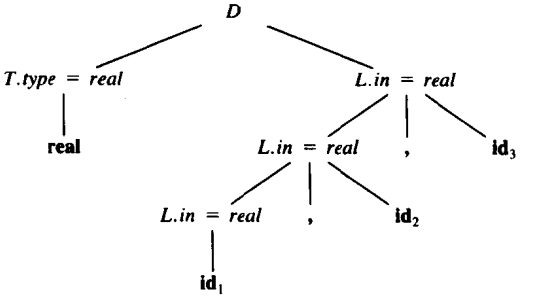


图5-5 在每个 *L* 节点处都带有继承属性 *in* 的分析树

5.1.4 依赖图

如果分析树中某个节点的属性 *b* 依赖于属性 *c*，那么在该节点处 *b* 的语义规则必须在 *c* 的

语义规则之后计算。分析树中各节点的继承属性和综合属性间的依赖关系可以用有向图来描述，这种有向图称为依赖图。

在构造分析树的依赖图之前，我们为每个包含过程调用的语义规则引入一个虚综合属性 b，以便把每条语义规则都变成 $b := f(c_1, c_2, \cdots, c_k)$ 的形式。依赖图的每个节点表示一个属性，边表示属性间的依赖关系，如果属性 b 依赖于属性 c，那么从 c 到 b 就有一条有向边。更详细地说，给定一棵分析树，其依赖图是按照下面的步骤构造出来的。

for 分析树的每个节点 n **do**
  for 与节点 n 对应的文法符号的每个属性 a **do**
    在依赖图中为 a 构造一个节点；
for 分析树的每个节点 n **do**
  for 节点 n 所用产生式对应的每条语义规则 $b := f(c_1, c_2, \cdots, c_k)$ **do**
    for $i := 1$ **to** k **do**
    从节点 c_i 到节点 b 构造一条有向边；

例如，假设 $A.a := f(X.x, Y.y)$ 是产生式 $A \rightarrow XY$ 的一条语义规则，该规则定义了一个依赖于属性 $X.x$ 和 $Y.y$ 的综合属性 $A.a$。如果分析树中含有该产生式，那么在依赖图中将有3个节点 $A.a$，$X.x$ 和 $Y.y$，而且由于 $A.a$ 依赖于 $X.x$，所以有一条从 $X.x$ 到 $A.a$ 的有向边，由于 $A.a$ 也依赖于 $Y.y$，所以还有一条从 $Y.y$ 到 $A.a$ 的边。

283
284

如果产生式 $A \rightarrow XY$ 所关联的语义规则还有 $X.i := g(A.a, Y.y)$，那么图中还应有一条从 $A.a$ 到 $X.i$ 的边，以及一条从 $Y.y$ 到 $X.i$ 的边，因为 $X.i$ 依赖于 $A.a$ 和 $Y.y$。

例5.4 只要下列产生式出现在分析树中，我们就把图5-6中的边加到依赖图中：

产生式	语义规则
$E \rightarrow E_1 + E_2$	$E.val := E_1.val + E_2.val$

依赖图中标记为•的3个节点代表综合属性 $E.val$，$E_1.val$ 和 $E_2.val$，从 $E_1.val$ 到 $E.val$ 的边表示 $E.val$ 依赖于 $E_1.val$，同样从 $E_2.val$ 到 $E.val$ 的边表示 $E.val$ 依赖于 $E_2.val$。图中虚线代表分析树，它们不是依赖图的一部分。 □

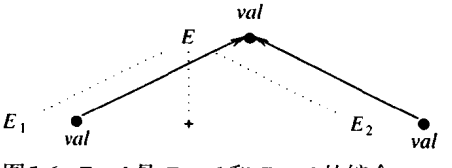


图5-6 $E.val$ 是 $E_1.val$ 和 $E_2.val$ 的综合

例5.5 图5-7给出了图5-5中分析树的依赖图。依赖图中的节点用数字标记，这些数字将在后面用到。从节点4的 $T.type$ 到节点5的 $L.in$ 有一条边，这是因为根据产生式 $D \rightarrow TL$ 的语义规则 $L.in := T.type$，继承属性 $L.in$ 依赖于综合属性 $T.type$。因为产生式 $L \rightarrow L_1$，**id** 具有语义规则 $L_1.in := L.in$，从而导致 $L_1.in$ 依赖于 $L.in$，所以有两条向下的边进入节点7和9。每个与 L 产生式相关联的语义规则 $addtype$ (**id**.$entry$, $L.in$) 都产生一个虚属性，节点6，8和10都是为这些虚属性建立的。 □

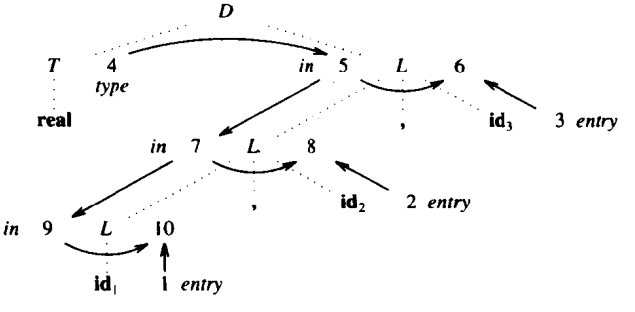


图5-7 图5-5中分析树的依赖图

5.1.5 计算顺序

无环有向图的*拓扑排序*是图中节点 $m_1, m_2, \cdots, m_k$ 的这样一种排序：若 $m_i \rightarrow m_j$ 是从 m_i 到 m_j 的边，那么在此排序中 m_i 先于 m_j。[⊖]

依赖图的任何拓扑排序都给出了一个分析树中各节点语义规则计算的正确顺序，即在计算 f 之前，语义规则 $b := f(c_1, c_2, \cdots, c_k)$ 中的依赖属性 $c_1, c_2, \cdots, c_k$ 都是已知的。

语法制导定义所说明的翻译可以像下面这样精确给出。最基本的文法用于构造输入串的分析树；依赖图像上面所讨论的那样建立；从依赖图的拓扑排序可以得到语义规则的计算顺序；按该顺序计算语义规则即可得到输入串的翻译。

例5.6 在图5-7的依赖图中，每条边都是从编号小的节点指向编号大的节点，所以将这些节点按编号从小到大写出即可得到该依赖图的一种拓扑排序。从该拓扑排序可以得到下面的翻译程序（其中 a_n 表示依赖图中编号为 n 的节点的属性）：

$a_4 := real;$
$a_5 := a_4;$
$addtype(\mathbf{id}_3.entry, a_5);$
$a_7 := a_5;$
$addtype(\mathbf{id}_2.entry, a_7);$
$a_9 := a_7;$
$addtype(\mathbf{id}_1.entry, a_9);$

对这些语义规则的计算将把类型 *real* 存放到符号表中每个标识符的相应表项中。 □

人们已经提出了一些计算语义规则的方法：

1. *分析树法*。在编译时，这种方法从分析树所构成的依赖图的拓扑排序中得到语义规则的计算顺序。若所考虑的特定的分析树的依赖图中含有环路，则这种方法将会失败。 285 ~ 286

2. *基于规则的方法*。在编译器构造时，与产生式相关联的语义规则是用手工或专门的工具来分析的。对于每一个产生式，计算该产生式所关联的属性的顺序在编译器构造时已经预先确定好了。

3. *忽略规则的方法*。这种方法选择计算顺序时不考虑语义规则。例如，如果翻译是在语法分析过程中进行的，那么计算顺序的选择就由语法分析方法来确定，而跟语义规则无关。这种方法限制了能被实现的语法制导定义的种类。

基于规则的方法和忽略规则的方法在编译时都不必显式地构造依赖图，因此它们使编译的时空效率更高。

如果某个文法所产生的某些分析树的依赖图中存在*环路*，则其语法制导定义是*循环的*。5.10节讨论如何检测语法制导定义的循环性。

5.2 语法树的构造

在本节中，我们将说明如何应用语法制导定义来完成语法树的构造以及语言结构的其他图形表示。

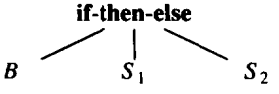
用语法树作为中间表示，可以把翻译从语法分析中分离出来。在语法分析期间完成翻译固然有很多优点，但也存在一些问题。如在分析期间调用翻译子程序受到两个限制。首先，适于

⊖ 当然，拓扑排序不止一种结果。——译者注

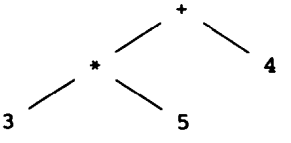
语法分析的文法可能并不反映语言成分的自然层次结构。例如，Fortran的文法可能把子程序看成只是简单地由语句序列组成的，没有反映出DO循环的嵌套。如果我们采用能够反映DO循环嵌套的树形表示，那么子程序的语法分析将会更容易一些。其次，语法分析方法限制了分析树中各节点的考察顺序，该顺序可能和各语法成分的信息变为可用的顺序不匹配。基于上述原因，C编译器通常要构造语法树。

5.2.1 语法树

（抽象）语法树是分析树的压缩形式，它对表示语言的结构很有用。如产生式 $S \rightarrow$ **if** B **then** S_1 **else** S_2 在语法树中可能出现为：



在语法树中，运算符和关键字不再是叶节点，而是作为内部节点成为分析树中叶节点的父
287 节点。语法树的另一个简化是单产生式（形如 $A \rightarrow B$）链可能消失，图5-3的分析树将变成下面的语法树：



语法制导翻译可以基于语法树，也可以基于分析树，其方法都是一样的。就像在分析树中那样，我们也可以给语法树的节点附加属性。

5.2.2 构造表达式的语法树

构造表达式的语法树类似于把表达式翻译成后缀表示。我们通过为每个运算符和运算对象建立节点来为子表达式构造子树。运算符节点的子节点分别是表示该运算符各运算对象的子表达式组成的子树的根。

语法树的每个节点都可以用带有几个域的记录来表示。对于运算符节点，记录中的一个域用来标识该运算符，其余的域包含指向运算对象的指针。我们常常将运算符称为该节点的标记（label）。在翻译时，语法树的节点可能还有一些其他域用来保存附加在该节点上的属性值(或指向属性值的指针)。在本节中，我们用下列函数建立带有二元运算符的表达式的语法树节点，每个函数都返回一个指向新建节点的指针：

1. *mknode*(*op*, *left*, *right*)。它建立一个标记为 *op* 的运算符节点，其两个域 *left* 和 *right* 是指向其左右运算对象的指针。

2. *mkleaf* (**id**, *entry*)。它建立标记为 **id** 的标识符节点，其域 *entry* 是指向该标识符在符号表中的相应表项的指针。

3. *mkleaf* (**num**, *val*)。它建立标记为 **num** 的数节点，域 *val* 保存该数的值。

例5.7 下面的函数调用序列用来建立图5-8中表达式 a-4+c 的语法树。在该序列中 p_1, p_2,…, p_5 是指向节点的指针，*entrya* 和 *entryc* 分别指向符号表中标识符a和c对应的表项。

(1) p_1 := *mkleaf* (**id**, *entrya*);

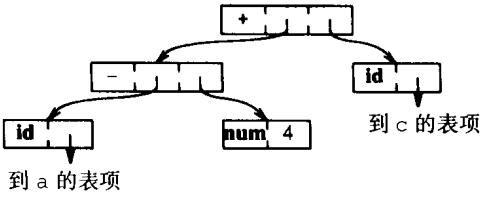


图5-8 a-4+c 的语法树

(2) p_2 := *mkleaf* (**num**, 4);

(3) p_3 := *mknode* ('-', p_1, p_2);

(4) p_4 := *mkleaf* (**id**, *entryc*);

(5) p_5 := *mknode* ('+', p_3, p_4);

该树是自底向上构造的，函数调用 *mkleaf* (**id**, *entrya*) 和 *mkleaf* (**num**, 4)分别为 a 和4建立叶节点，指向这两个节点的指针保存在 p_1 和 p_2 中；然后，函数调用 *mknode*('-', p_1 , p_2)构造一个内部节点，其子节点为叶节点 a 和4；再经过两步即可建立起语法树，p_5 指向根节点。 □

5.2.3 构造语法树的语法制导定义

图5-9是为包含+和-的表达式的构造语法树的 S 属性定义。它通过为文法的基本产生式安排函数 *mknode* 和 *mkleaf* 的调用来构造语法树。*E* 和 *T* 的综合属性 *nptr* 用来记住函数调用返回的指针。

产生式	语义规则
$E \rightarrow E_1 + T$	*E.nptr* := *mknode* ('+', E_1.*nptr*, *T.nptr*)
$E \rightarrow E_1 - T$	*E.nptr* := *mknode* ('-', E_1.*nptr*, *T.nptr*)
$E \rightarrow T$	*E.nptr* := *T.nptr*
$T \rightarrow (\ E\ )$	*T.nptr* := *E.nptr*
$T \rightarrow$ **id**	*T.nptr* := *mkleaf* (**id**, **id**.*entry*)
$T \rightarrow$ **num**	*T.nptr* := *mkleaf* (**num**, **num**.*val*)

图5-9 构造表达式的语法树的语法制导定义

例5.8 图5-10是一棵注释分析树，用于描述表达式 a-4+c 的语法树的构造。分析树用虚线表示。非终结符 *E* 和 *T* 标记的分析树节点用综合属性 *nptr* 来保存指向语法树中该非终结符所代表的表达式节点的指针。

产生式 *T*→**id** 和 *T*→**num** 所对应的语义规则将属性 *T.nptr* 定义为指向标识符和数的新建叶节点的指针。属性 **id**.*entry* 和 **num**.*val* 是词法值，假设词法分析器将这些值连同记号 **id** 和 **num** 一起返回。

288 ~ 289

在图5-10中，当表达式 *E* 是一个单个项时，即使用产生式 *E*→*T* 时，属性 *E.nptr* 得到 *T.nptr* 的值。当使用产生式 *E* → E_1–*T* 所对应的语义规则 *E.nptr* := *mknode* ('–', E_1.*nptr*, *T.nptr*) 时，先前的规则已经把 E_1.*nptr* 和 *T.nptr* 分别置成指向叶节点 a 和4的指针。

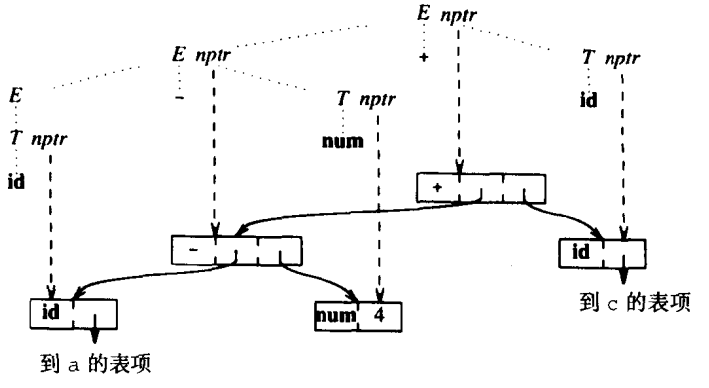


图5-10 a-4+c 的语法树的构造

对图5-10的解释关键要认识到，由记录形成的低层树才是构成输出的“真正的”语法树，而上层虚线形成的树是分析树，在此处只具有象征意义。在下一节中，我们将说明如何用自底向上语法分析器的栈来保存属性值以实现 S 属性定义。事实上，在这种实现中，节点建立函数的调用顺序和例5.7中的顺序相同。 □

5.2.4 表达式的无环有向图

表达式的无环有向图（directed acyclic graph，后面简称dag）可以识别表达式中的公共子表达式。和语法树一样，在dag中，表达式中的每一个子表达式都有一个节点与之对应。内部节点代表运算符，子节点代表其运算对象。不同的是，在dag中，代表公共子表达式的节点具有多个“父节点”，而在语法树中，公共子表达式由重复的子树表示，所以它只有一个父节点。

图5-11是表达式 a+a*(b-c)+(b-c)*d的dag。其叶节点 a具有两个父节点，因为a是两个子表达式 a和 a*(b-c)的公共子表达式。类似地，公共子表达式 b-c 的两次出现也由同一个节点表示，相应地亦拥有两个父节点。

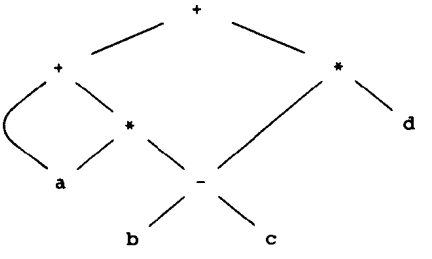


图5-11 表达式a+a*(b-c)+(b-c)*d的dag

如果我们修改构造节点的函数，图5-9的语法制导定义即可用于构造 dag 而不是语法树：只要在构造节点之前先检查是否存在相同节点，就可以得到 dag。例如，在以标记 *op* 和指向*left* 和 *right* 的指针的域构造新节点之前，*mknode* (*op*, *left*, *right*) 先检查是否已经存在这样的节点。如果存在的话，*mknode* (*op*, *left*, *right*) 可以返回一个指向先前已构造好了的节点的指针，叶节点构造函数 *mkleaf* 可采用类似的修改。

例5.9 如果 *mknode* 和 *mkleaf* 仅在必要时才建立新节点，而且尽量返回指向具有正确标记和子节点的现存节点的指针，那么图5-12的指令序列将构造出图5-11的 dag。在图5-12中，*a*, *b*, *c* 和 *d* 分别指向符号表中标识符 a, b, c 和 d 的表项。

```
(1)  p1 := mkleaf(id, a);
(2)  p2 := mkleaf(id, a);
(3)  p3 := mkleaf(id, b);
(4)  p4 := mkleaf(id, c);
(5)  p5 := mknode('-', p3, p4);
(6)  p6 := mknode('*', p2, p5);
(7)  p7 := mknode('+', p1, p6);
(8)  p8 := mkleaf(id, b);
(9)  p9 := mkleaf(id, c);
(10) p10 := mknode('-', p8, p9);
(11) p11 := mkleaf(id, d);
(12) p12 := mknode('*', p10, p11);
(13) p13 := mknode('+', p7, p12);
```

图5-12 构造图5-11的 dag 的指令

在第(2)行重复调用 *mkleaf* (**id**, *a*)时，前一个 *mkleaf* (**id**, *a*) 调用所构造的节点将被返回，所以 $p_1 = p_2$。类似地，第(8)行和第(9)行返回的节点分别与第(3)行和第(4)行返回的节点相同。因此，第(10)行返回的节点和第(5)行调用 *mknode* 所构造的节点是相同的。 □

290
~
291

如图5-13所示，在许多应用中节点都被实现为记录保存在数组中。图中每个记录都有一个

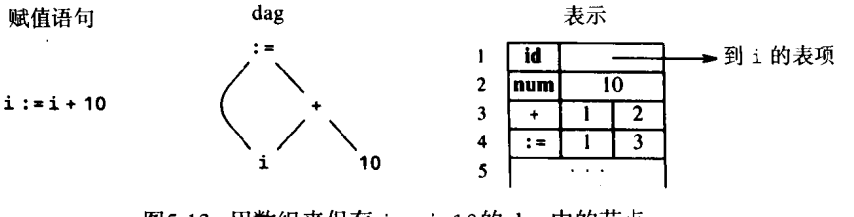


图5-13 用数组来保存 i := i+10的 dag 中的节点

标记域，它决定了节点的性质。可以通过节点在数组中的下标或位置来引用节点。由于历史原因，节点的整数索引通常被称为值编号（value number）。例如，使用值编号，我们可以说节点3的标记为+，其左子节点为节点1，右子节点为节点2。下面的算法可用来为表达式的dag表示创建节点。

算法5.1 构造dag节点的值编号法。

假设节点保存在一个数组中（就像图5-13中那样），而且每个节点通过其值编号来引用。令运算符节点的签名（signature）为一个三元组 $<op, l, r>$，其中 op 为标记，l 为左子节点，r 为右子节点。

输入：标记 op、节点 l 及节点 r。

输出：带有签名 $<op, l, r>$ 的节点。

方法：搜索数组，寻找具有标志 op，且左子节点为 l，右子节点为 r 的节点 m。如果存在，则返回 m；否则，创建一个新节点 n，将其标记域置为 op，左子节点置为 l，右子节点置为 r，并返回 n。

确定节点 m 是否已经存在于数组中的一个简单方法是将前面建立的所有节点保存在一个列表中，然后依次查看列表中的每个节点，看其是否具有所要求的签名。使用 k 个称为桶的列表可以提高查找节点 m 的效率，而散列函数 h 用来确定要搜索的桶。[⊖]

散列函数 h 根据 op，l 和 r 的值来计算散列桶的序号。给定相同的参数，它总是返回相同的桶号。如果节点 m 不在散列桶 $h(op, l, r)$ 中，就创建一个新节点 n，并将其加入该散列桶中，所以接下来的搜索将会发现它已经在该桶中了。不同的签名可能会被散列到同一个散列桶中[⊖]，但实际上我们希望每个桶只包含少数节点。

可以像图5-14那样用链表来实现每个散列桶。链表中的每个单元代表一个节点。散列桶的 292
首部是指向链表中第一个单元的指针，我们将其保存在数组中。由散列函数 $h(op, l, r)$ 返回的桶序号是该数组的一个下标。

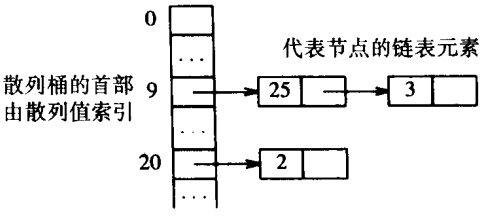


图5-14 搜索散列桶的数据结构

对于非顺序存储的节点，我们可以修改此算法。在许多编译器中，为避免预分配一个大部分时间保存太多节点而有时又没有保存足够节点的数组，节点是按照需要进行分配的。在这种情况下，我们就不能假定节点是顺序存放的了，此时必须用指针来访问节点。如果可以设计一个能根据标记值和指向子节点的指针来计算散列桶号的散列函数，那么我们就可以使用指向节点的指针而不是值编号。否则，我们可以以任何方式对节点进行编号并用这些编号作为节点的值编号。 □

由于dag可以有多个根节点，因此dag可以用来表示表达式的集合。在第9章和第10章中，赋值语句序列所执行的计算将被表示为一个dag。

⊖ 实现Aho, Hopcroft, and Ullman[1983]中提出的字典的任何数据结构都可以满足这里的需要。该数据结构的重要性质是：给定一个关键字，即标记 op 和两个节点 l 和 r，我们可以很快地获得带有签名 $<op, l, r>$ 的节点，或确定不存在这样的节点。

⊖ 我们可以通过外散列法来解决这种冲突。——译者注

5.3 自底向上计算S属性定义

既然我们已经知道了怎样使用语法制导定义来说明翻译，现在我们就可以研究如何实现这种翻译器了。针对任意一个语法制导定义的翻译器可能很难建立，但是有很多种语法制导定义的翻译器却是很容易建立的。本节中，我们将主要考虑这样一类语法制导定义：S属性定义。它是一种只含有综合属性的语法制导定义。下面几节我们还将考虑具有继承属性的语法制导定义的实现。

综合属性可以在分析输入串的同时由自底向上的语法分析器来计算。这种语法分析器可以将文法符号的综合属性值保存在栈中，每当进行归约时，新的综合属性值就由栈中正在归约的
293 产生式右部符号的属性值来计算。本节将说明如何扩充语法分析栈，使之能够保存这些综合属性的值。在5.6节，我们将看到这种实现方式也可用于某些继承属性。

在图5-9中，构造表达式语法树的语法制导定义中只有综合属性出现。因此可以用本节的方法在自底向上的语法分析过程中构造语法树。在5.5节中我们将看到，在自顶向下的语法分析过程中表达式的翻译经常会用到继承属性。因此我们将在下一节考查了“自左向右”的依赖关系以后再讨论自顶向下语法分析过程的翻译。

语法分析栈中的综合属性

S属性定义的翻译器可以借助LR语法分析器生成器来实现，如4.9节所讨论的 Yacc。从一个S属性定义，语法分析器生成器可以在分析输入时构造一个计算属性的翻译器。

自底向上语法分析器用栈来保存已分析子树的信息，我们可以在分析栈中使用一个附加的域来保存综合属性值。图5-15给出了一个带有一个属性值空间的分析栈的例子。我们假设图中的栈是由一对数组 *state* 和 *val* 来实现的。每个 *state* 表项都是一个指向LR(1)语法分析表的指针（或索引）(注意，文法符号隐含在 *state* 中而无需存放在栈中)。但是，为方便起见，像4.7节那样将文法符号放入栈中时，我们用单个文法符号来代替状态。如果第 *i* 个 *state* 符号是 *A*，那么 *val*[*i*] 保存的就是 *A* 节点的属性值。

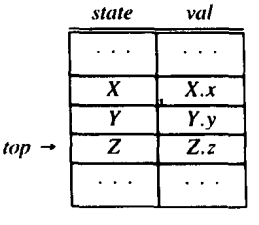


图5-15 带有综合属性域的分析栈

当前栈顶由指针 *top* 指示，我们假设综合属性正好是在每次归约之前计算的。假设与产生式 $A \rightarrow XYZ$ 相关联的语义规则是 $A.a := f(X.x, Y.y, Z.z)$。在将 *XYZ* 归约为 *A* 之前，属性 *Z.z* 的值在 *val*[*top*]中，*Y.y* 的值在 *val* [*top*−1]中，*X.x* 的值在
294 *val* [*top*−2]中。（如果符号没有属性，那么 *val* 数组中相应的元素就没有定义。）归约后，*top* 减2，*A* 的状态放在 *state* [*top*] 中(即 *X* 原来的位置)，综合属性 *A.a* 的值放在 *val* [*top*]中。

例5.10 再次考虑图5-2中台式计算器的语法制导定义。图5-3注释分析树上的综合属性值可以由LR语法分析器在分析输入串 3*5+4**n** 时计算出来。同前，假设属性 **digit**.*lexval* 的值由词法分析器提供，它是代表数字的每个记号的数值。当语法分析器把 **digit** 移进栈时，记号 **digit** 放在 *state*[*top*]中，其属性值放在 *val* [*top*] 中。

我们可以用4.7节的技术来构造该文法的LR语法分析器。为了计算属性，我们修改语法分析器，使其在归约前执行图5-16中的代码段。注意，由于每次归约前都要确定应该选择哪个产生式，因此我们可以将属性的计算和归约动作联系起来。图中的代码段是从图5-2的语义规则得来的，只是将其中的属性值替换为 *val* 数组中的位置而已。

代码段中没有说明变量 *top* 和 *ntop* 是如何管理的。如果归约的产生式右部有 *r* 个符号，则

置 *ntop* 为*top* − *r* + 1。每个代码段执行完之后，则置 *top* 为 *ntop*。

图5-17给出了语法分析器在输入串 3*5+4**n** 上的移动序列，图中还给出了每次移动后分析栈中 *state* 和 *val* 域的内容。我们仍然用相应的文法符号来代替栈中的状态。同时为直观起见，用实际的输入数字来代替记号 **digit**。

考虑遇到输入符号3时的事件序列。第一步，语法分析器把记号 **digit**（其属性值为3）的状态移进栈中（状态用3来表示，而且值 3 放在 *val* 域中）。第二步，语法分析器用产生式 *F*→**digit** 归约，并执行语义规则 *F.val* := **digit**.*lexval*。第三步，语法分析器用产生式*T*→*F*归约。因为没有代码段与该产生式相关联，所以 *val* 数组没有改变。注意，每次归约后，*val* 栈顶存放的是与归约时所用产生式左部相关联的属性值。 □

产生式	代码段
$L \rightarrow E$ **n**	*print*(*val*[*top*])
$E \rightarrow E_1 + T$	*val*[*ntop*] := *val*[*top* − 2]+*val*[*top*]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	*val*[*ntop*] := *val*[*top* − 2]×*val*[*top*]
$T \rightarrow F$	
$F \rightarrow (E)$	*val*[*ntop*] := *val*[*top* − 1]
$F \rightarrow$ **digit**	

图5-16 用LR语法分析器实现台式计算器

输入	*state*	*val*	使用的产生式
3*5+4**n**	–	–	
*5+4**n**	3	3	
*5+4**n**	*F*	3	*F* → **digit**
*5+4**n**	*T*	3	*T* → *F*
5+4**n**	*T* *	3 _	
+4**n**	*T* * 5	3 _ 5	
+4**n**	*T* * *F*	3 _ 5	*F* → **digit**
+4**n**	*T*	15	*T* → *T* * *F*
+4**n**	*E*	15	*E* → *T*
4**n**	*E* +	15 _	
n	*E* + 4	15 _ 4	
n	*E* + *F*	15 _ 4	*F* → **digit**
n	*E* + *T*	15 _ 4	*T* → *F*
n	*E*	19	*E* → *E* + *T*
	E **n**	19 _	
	L	19	*L* → *E* **n**

图5-17 翻译器在输入3* 5+4**n** 上所做的移动

在上面描述的实现中，代码段正好在归约之前执行。归约提供了一个“挂钩”，任何代码段所组成的动作都可以挂在“挂钩”上。也就是说，我们可以允许用户把一个动作和一个产生式联系起来，该动作是利用该产生式进行归约时要被执行的。下一节讨论的翻译模式提供了一种将语义动作与语法分析交叉进行的表示法。在5.6节中，我们将看到更大的一类语法制导定义可以在自底向上的语法分析过程中实现。

5.4 L属性定义

在语法分析过程中进行翻译时，属性的计算顺序将与分析方法建立分析树节点的顺序相关。有一种能够描述许多种自顶向下和自底向上翻译方法的自然顺序，那就是*深度优先顺序*，以分析树的根为参数调用图5-18的过程 *dfvisit* 即可得到该顺序。即使没有实际构造分析树，分析树节点属性的深度优先计算对研究语法分析期间的翻译也非常有用。

```
procedure dfvisit (n: node);
begin
    for n的每个子节点m（从左到右）do begin
        计算m的继承属性;
        dfvisit (m)
    end;
    计算n的综合属性
end
```

图5-18 分析树属性的深度优先计算顺序

现在我们介绍一种语法制导定义，它称为L属性定义，其属性总可以用深度优先顺序来计算（其中 L 表示左，因为属性信息是从左向右传播的）。本章下面的3节将讨论L属性定义的实现。L属性定义包括所有基于 LL(1)文法的语法制导定义，5.5节将给出一种使用预测分析法并用一遍扫描即可实现这类L属性定义的方法。

通过对5.3节翻译方法的扩充，5.6节将在自底向上语法分析期间实现更大的一类L属性定义。5.7节将给出一种实现所有L属性定义的通用方法。

5.4.1 L属性定义

一个语法制导定义是L属性定义，如果对每个产生式 $A \rightarrow X_1X_2\cdots X_n$，其右部符号 X_j（$1 \leqslant j \leqslant n$）的每个继承属性仅依赖于下列属性：

1. 产生式中X_j左边的符号 X_1，X_2，…，X_{j-1}的属性。

2. A 的继承属性。

注意，每个 S 属性定义都是L属性定义，因为1和2只限制了继承属性。

产生式	语义规则
$A \rightarrow L\ M$	$L.i := l(A.i)$
	$M.i := m(L.s)$
	$A.s := f(M.s)$
$A \rightarrow Q\ R$	$R.i := r(A.i)$
	$Q.i := q(R.s)$
	$A.s := f(Q.s)$

图5-19 非 L 属性的语法制导定义

例5.11 图5-19的语法制导定义不是 L 属性定义，因为文法符号 Q 的继承属性 $Q.i$ 依赖于属性 $R.s$，而 R 在 Q 的右边。5.8节和5.9节还有一些其他非 L 属性的语法制导定义的例子。 □

5.4.2 翻译模式

295～297

翻译模式就是将文法符号同属性相关联的上下文无关文法，而且包含在{ }中的语义动作可以插入产生式右部的某个位置（如2.3节）。在本章中，说明语法分析过程中所进行的翻译时，我们将翻译模式作为一种有用的表示法。

本章所考虑的翻译模式可以同时具有综合属性和继承属性。在第2章所考虑的简单翻译模式中，属性的类型为字符串，每个符号具有一个属性，而且对每个产生式 $A \rightarrow X_1 \cdots X_n$，语义规则把 $X_1, \cdots, X_n$ 的属性字符串依次连接起来形成 A 的属性字符串，在此拼接过程中还可以加入其他可选串。显然，按照语义规则中字符串出现的顺序简单地打印出这些字符串，就可以完成翻译。

例5.12 下面就是一个简单翻译模式，它把带有加号和减号的中缀表达式翻译成后缀表达式。对第2章的翻译模式 (2-14) 稍加修改即可得到该翻译模式。

$$
\begin{aligned}
E &\rightarrow T\ R \\
R &\rightarrow \textbf{addop}\ T\ \{print(\textbf{addop}.lexeme)\}\ R_1 \mid \epsilon \\
T &\rightarrow \textbf{num}\ \{print(\textbf{num}.val)\}
\end{aligned}
\tag{5-1}
$$

图5-20是输入串 9-5+2的分析树，每个语义动作都作为产生式左部所对应节点的子节点。实际上，我们将语义动作看成是终结符，对确定动作何时执行来说，它是一种方便的助记符号。图中用具体的数字和加（减）运算符代替了记号 **num** 和 **addop**。当以深度优先顺序执行图5-20中的动作时，其输出为95-2+。 □

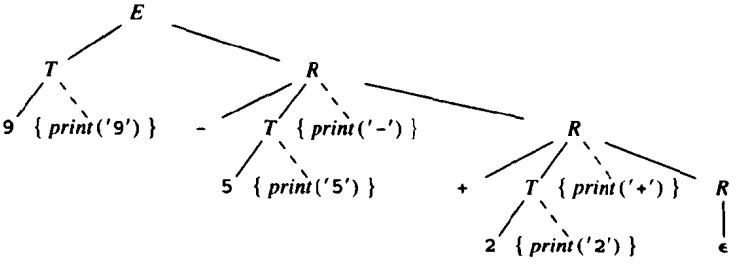


图5-20 9-5+2的带有动作的分析树

设计翻译模式时，我们必须遵守某些限制以保证当某个动作引用一个属性时它是可用的。

L属性定义所规定的这些限制确保一个动作不会引用一个还没有计算出来的属性。

只需要综合属性的情况最为简单。在这种情况下，我们这样来建立翻译模式：为每个语义规则建立一个赋值动作，并把该动作放在产生式右部的末尾。例如，产生式和语义规则

产生式	语义规则
$T \rightarrow T_1 * F$	$T.val := T_1.val \times F.val$

产生的产生式和语义动作为：

$T \rightarrow T_1 * F \ \{ T.val := T_1.val \times F.val \}$

如果同时存在继承属性和综合属性，则需要注意：

1. 产生式右部符号的继承属性必须在这个符号以前的动作中计算出来。
2. 一个动作不能引用该动作右边符号的综合属性。
3. 产生式左部非终结符的综合属性只有在其引用的所有属性都计算出来以后才能计算。计算该属性的动作通常放在产生式右部的末尾。

下面两节将说明满足上述 3 点要求的翻译模式如何由一般的自顶向下和自底向上语法分析器来实现。

下面的翻译模式不满足上述 3 点要求中的第1点：

$S \rightarrow A_1 A_2 \ \{ A_1.in := 1; A_2.in := 2 \}$
$A \rightarrow a \quad \{ print(A.in) \}$

可以看出，对输入串 *aa* 的分析树进行深度优先遍历时，当试图打印第二个产生式的 *A.in* 值时，继承属性 *A.in* 尚未定义。也就是说，从 *S* 开始进行深度优先遍历，在访问 A_1 和 A_2 的子树之前，$A_1.in$ 和 $A_2.in$ 未被置值。如果定义 $A_1.in$ 和 $A_2.in$ 的值的动作嵌在 $S \rightarrow A_1A_2$ 右部的 A_1 之前，那么每次执行 *print* (*A.in*) 时，*A.in* 将被定义。 298 ~ 299

从一个 L 属性语法制导定义，我们总能构造出满足上述 3 个条件的翻译模式。下面的例子说明了这种构造过程，它基于数学排版语言EQN（在1.2节曾简单描述过）。给定输入

```
E sub 1 .val
```

EQN 将 *E*，1和 *.val* 按不同大小放在相应的位置上，如图5-21所示。注意，其中下标1将以较小的字体印，并且其位置也低于 *E* 和 *.val*。

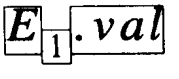


图5-21 盒子的语法制导布局

例5.13 从图5-22的L属性定义我们可以构造出图5-23的翻译模式。图中非终结符 *B*（表示盒子）代表一个公式。产生式 $B \rightarrow B\ B$ 代表两个盒子的并列，$B \rightarrow B$ **sub** B 代表一种布局，其中第二个盒子是第一个盒子的下标，它的尺寸较第一个盒子小，并被放在合适的位置上。

继承属性 *ps*（以点计的尺寸）将影响公式的高度。产生式 $B \rightarrow$ **text** 的语义规则用来计算正文的实际高度，即正文的标准高度乘以 *B.ps*。给定记号 **text** 所代表的字符，通过查表即可获得**text** 的属性 *h*。应用产生式 $B \rightarrow B_1B_2$ 时，B_1 和B_2 通过复制规则继承了 *B* 的 *ps* 属性。*B* 的高度由综合属性 *ht* 表示，*B* 是 B_1 和 B_2 高度的最大值。

使用产生式 $B \rightarrow B_1$ **sub** B_2 时，函数 *shrink* 将 $B_2.ps$ 缩小30%，函数 *disp* 在计算 *B* 的高度时把盒子 B_2 向下放置。此处没有给出产生实际排版命令的语义规则。

图5-22中的定义是 L 属性定义。惟一的继承属性是 *B.ps*，定义 *ps* 的每个语义规则仅依赖于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。

产生式	语义规则
$S \rightarrow B$	$B.ps := 10$ $S.ht := B.ht$
$B \rightarrow B_1\ B_2$	$B_1.ps := B.ps$ $B_2.ps := B.ps$ $B.ht := max(B_1.ht, B_2.ht)$
$B \rightarrow B_1$ **sub** B_2	$B_1.ps := B.ps$ $B_2.ps := shrink(B.ps)$ $B.ht := disp(B_1.ht, B_2.ht)$
$B \rightarrow$ **text**	$B.ht :=$ **text**$.h \times B.ps$

图5-22 定义盒子大小和高度的语法制导定义

$S \rightarrow$ { $B.ps := 10$ }
  B { $S.ht := B.ht$ }
$B \rightarrow$ { $B_1.ps := B.ps$ }
  B_1 { $B_2.ps := B.ps$ }
  B_2 { $B.ht := max(B_1.ht, B_2.ht)$ }
$B \rightarrow$ { $B_1.ps := B.ps$ }
  B_1
  sub { $B_2.ps := shrink(B.ps)$ }
  B_2 { $B.ht := disp(B_1.ht, B_2.ht)$ }
$B \rightarrow$ **text** { $B.ht :=$ **text**$.h \times B.ps$ }

图5-23 从图5-22构造的翻译模式

图5-23的翻译模式是根据上述 3 点要求，将图5-22中的语义规则插入产生式而得到的。为增加可读性，产生式的每个文法符号写在不同的行上，而动作在其右边给出。所以，把

$S \rightarrow \{ B.ps := 10 \}\ B\ \{ S.ht := B.ht \}$

写成

$S \rightarrow$ { $B.ps := 10$ }
  B { $S.ht := B.ht$ }

300
~
301

注意，置继承属性 $B_1.ps$ 和 $B_2.ps$ 的动作刚好在产生式右部 B_1 和 B_2 的前面。 □

5.5 自顶向下翻译

在本节中，我们将在预测分析的过程中实现 L 属性定义。为了明显地看出动作和属性计算的发生顺序，我们利用翻译模式而不是语法制导定义来进行说明。另外，我们还将消除左递归的算法扩展到带有综合属性的翻译模式中。

5.5.1 从翻译模式中消除左递归

因为大多数算术运算符是左结合的，所以人们很自然地用左递归文法来表示算术表达式。这里，我们将扩充2.4节和4.3节消除左递归的转换算法，使之适用于带有综合属性的翻译模式。它使5.1节和5.2节的许多语法制导定义都可以用预测分析法来实现。下面是一个转换的例子。

例5.14 下面将图5-24的翻译模式转换成图5-25的翻译模式。新的翻译模式产生图5-26中表达式9-5+2的注释分析树，图中的箭头表明了确定表达式的值的方法。

$E \rightarrow E_1 + T$ { $E.val := E_1.val + T.val$ }
$E \rightarrow E_1 - T$ { $E.val := E_1.val - T.val$ }
$E \rightarrow T$ { $E.val := T.val$ }
$T \rightarrow (\ E\ )$ { $T.val := E.val$ }
$T \rightarrow$ **num** { $T.val :=$ **num**$.val$ }

图5-24 左递归文法的翻译模式

在图5-26中，每个数都是由 T 产生的，$T.val$ 的值就是该数的词法值 **num**$.val$。子表达式9-5中的9是由最左边的 T 产生的，减号和 5 是由根的右子节点 R 产生的。继承属性 $R.i$ 的值 9 得自$T.val$，然后计算9-5并向下将结果 4 传递到中间的 R 节点，减法操作是由嵌在产生式 $R \rightarrow -TR_1$中 T、R_1之间的如下动作来完成的：

$\{ R_1.i := R.i - T.val \}$

用相似的动作可以把 2 加到9-5上，并在底层的 R 节点上输出结果 $R.i = 6$。最终的结果是根

节点的 $E.val$ 值，用 R 的综合属性 s 会把最终结果向上复制到根（这一部分没有在图5-26中给出）。□

对自顶向下语法分析，我们可以假设动作是在与之处于同一位置的符号被展开时执行的。例如，图5-25中的第二个产生式中，第一个动作{ $R_1.i := R.i + T.val$}（赋值给$R_1.i$）是在 T 被完全展开成终结符后执行的，第二个动作{$R.s := R_1.s$}是在 R_1被完全展开后执行的。正如5.4节对L属性定义的讨论，符号的继承属性必须由该符号前面的动作来计算，并且产生式左部非终结符的综合属性必须在它所依赖的所有属性都计算出来之后才能计算。

$E \rightarrow T$ { $R.i := T.val$ }
R { $E.val := R.s$ }

$R \rightarrow +$
T { $R_1.i := R.i + T.val$ }
R_1 { $R.s := R_1.s$ }

$R \rightarrow -$
T { $R_1.i := R.i - T.val$ }
R_1 { $R.s := R_1.s$ }

$R \rightarrow \epsilon$ { $R.s := R.i$ }

$T \rightarrow ($
E
$)$ { $T.val := E.val$ }

$T \rightarrow$ **num** { $T.val :=$ **num**.val }

图5-25 转换后的右递归文法的翻译模式

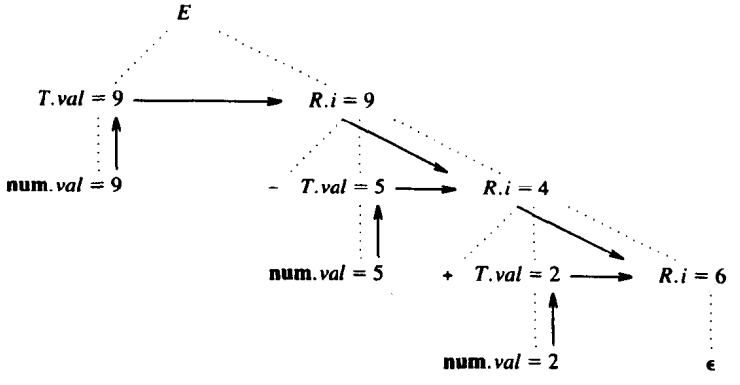


图5-26 表达式9-5+2的计算

为了使其他左递归翻译模式适于预测语法分析，我们将更抽象地表示图5-25中属性 $R.i$ 和 $R.s$ 的使用。假如我们有如下的翻译模式：

302
~
303

$$\begin{array}{l} A \rightarrow A_1\ Y \quad \{\ A.a := g(A_1.a,\ Y.y)\ \} \\ A \rightarrow X \quad\ \ \{\ A.a := f(X.x)\ \} \end{array} \tag{5-2}$$

每个文法符号都有一个综合属性，用相应的小写字母表示，而且 f 和 g 是任意函数。对于更一般的情况，如用串来代替符号 X 和 Y 等，我们会在例5.15中描述。

2.4节中消除左递归的算法可以从(5-2)构造出下面的文法：

$$\begin{array}{l} A \rightarrow X\ R \\ R \rightarrow Y\ R \mid \epsilon \end{array} \tag{5-3}$$

考虑到相应的语义动作，翻译模式变为：

$$\begin{array}{ll} A \rightarrow X & \{\ R.i := f(X.x)\ \} \\ \quad R & \{\ A.a := R.s\ \} \\ R \rightarrow Y & \{\ R_1.i := g(R.i,\ Y.y)\ \} \\ \quad R_1 & \{\ R.s := R_1.s\ \} \\ R \rightarrow \epsilon & \{\ R.s := R.i\ \} \end{array} \tag{5-4}$$

如图5-25所示，转换后的翻译模式使用 R 的属性 i 和 s。为弄清为什么 (5-2)和(5-4) 的结果是一样的，我们考虑图5-27中两棵注释分析树。图5-27a中$A.a$的值是根据(5-2)计算的。图5-27b中包含了按照(5-4)从上到下对 $R.i$ 的计算。底部的 $R.i$ 值未加改变地被作为 $R.s$ 的值传递到上面，并作为根节点 $A.a$ 的正确值（图5-27b中没有给出 $R.s$ 的值）。

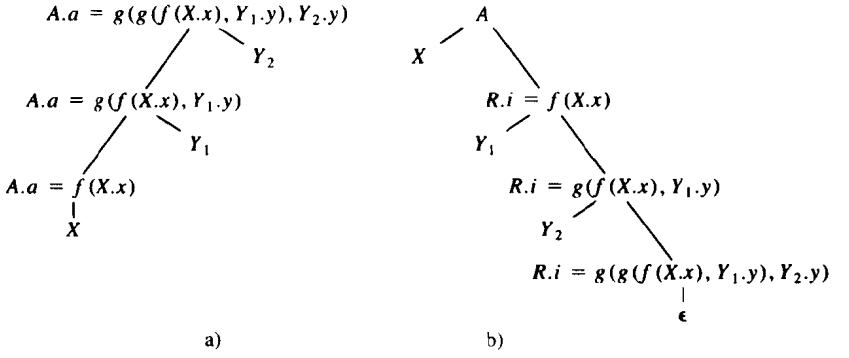


图5-27 计算一个属性值的两种方法

304 **例5.15** 如果将图5-9中的构造语法树的语法制导定义转换成翻译模式，那么 E 的产生式和语义规则将变成：

$$
\begin{aligned}
&E \rightarrow E_1 + T \quad \{\ E.nptr := mknode('+', E_1.nptr, T.nptr)\ \}\\
&E \rightarrow E_1 - T \quad \{\ E.nptr := mknode('-', E_1.nptr, T.nptr)\ \}\\
&E \rightarrow T \qquad\ \{\ E.nptr := T.nptr\ \}
\end{aligned}
$$

当从该翻译模式中消除左递归时，非终结符 E 对应于(5-2)中的 A，前面两个产生式中的串 $+\ T$ 和$-T$ 对应于 Y，第三个产生式的 T 对应于 X。转换后的翻译模式如图5-28所示。其中 T 的产生式和语义动作类似于图5-9中的原始定义。

图5-29说明怎样用图5-28的动作为表达式 a-4+c 构造语法树。综合属性标在文法符号节点的右边，继承属性标在文法符号节点的左边。像例5.8那样，语法树的叶节点由产生式 $T\rightarrow$**id** 和 $T\rightarrow$**num** 的翻译动作来构造。在最左边的节点 T，属性 $T.nptr$ 指向叶节点 a，该指针由 $E\rightarrow TR$ 的右部 R 的属性 $R.i$ 来继承。

当产生式 $R\rightarrow -TR_1$ 作用在根的右子节点 R 上时，$R.i$ 指向节点 a，$T.nptr$ 指向节点4，将 *mknode* 作用在减号和这些指针上即可构造出对应于a-4的节点。

$E \rightarrow T$	{ $R.i := T.nptr$ }
R	{ $E.nptr := R.s$ }
$R \rightarrow +$	
T	{ $R_1.i := mknode('+', R.i, T.nptr)$ }
R_1	{ $R.s := R_1.s$ }
$R \rightarrow -$	
T	{ $R_1.i := mknode('-', R.i, T.nptr)$ }
R_1	{ $R.s := R_1.s$ }
$R \rightarrow \epsilon$	{ $R.s := R.i$ }
$T \rightarrow$ (	
E	
)	{ $T.nptr := E.nptr$ }
$T \rightarrow$ **id**	{ $T.nptr := mkleaf($**id**, **id**$.entry)$ }
$T \rightarrow$ **num**	{ $T.nptr := mkleaf($**num**, **num**$.val)$ }

图5-28 转换后的构造语法树的翻译模式

最后，应用产生式$R\rightarrow\epsilon$，$R.i$ 指向语法树的根。接着，这棵树通过 R 节点的 s 属性逐层返回（图5-29中没有给出），并最终用 305 $E.nptr$ 值返回结果。 □

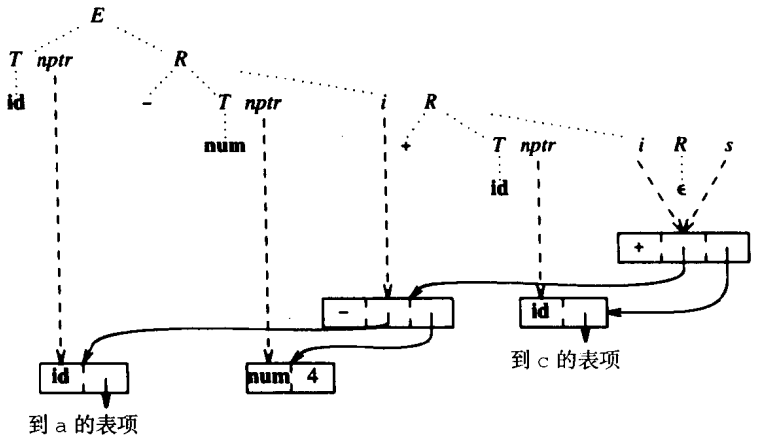


图5-29 使用继承属性构造语法树

5.5.2 预测翻译器的设计

下面的算法将预测语法分析器的构造进行推广，使之可以实现基于适合自顶向下分析的文法的翻译模式。

算法5.2 构造语法制导的预测翻译器。

输入：一个带有适于预测分析的基础文法的语法制导翻译模式。

输出：语法制导翻译器的代码。

方法：修改2.4节中的预测语法分析器构造技术。

1. 为每个非终结符 A 构造一个函数，A 的每个继承属性均对应该函数的一个形式参数，其返回值为 A 的综合属性的值（可能是一个记录、一个指向记录的指针或使用传地址参数传递机制）。为简单起见，我们假设每个非终结符只有一个综合属性。同时，A 的函数对出现在 A 产生式中的每个文法符号的每个属性都有一个局部变量。

2. 正如2.4节所讨论的，非终结符 A 的代码会根据当前的输入决定使用哪个产生式。

3. 与每个产生式有关的代码执行如下动作：从左到右考虑产生式右部的记号、非终结符及语义动作。 306

i) 对于带有综合属性 x 的记号 X，把 x 的值保存在 $X.x$ 中；然后产生一个匹配 X 的调用，并继续输入。

ii) 对于非终结符 B，产生一个右部带有函数调用的赋值语句 $c := B(b_1, b_2, \cdots, b_k)$，其中 $b_1, b_2, \cdots, b_k$ 是代表 B 的继承属性的变量，c 是代表 B 的综合属性的变量。

iii) 对于每个动作，将其代码复制到语法分析器，并把对属性的引用改为对相应变量的引用。 □

我们将在5.7节对算法5.2进行扩展，以便能实现任何L属性定义，假设事先已经构造好了分析树。在5.8节中，我们将考虑几种改进算法5.2所构造的翻译器的方法。例如，可以消除形如 $x := y$ 的复制语句，或者用一个变量保存若干属性值。另外，还可以用第10章的方法来自动完成某些这类改进。

例5.16 图5-28中的文法是LL(1)文法，因而适合于自顶向下分析。依据文法中非终结符的属性，我们可以得到函数 E，R 和 T 的参数和结果的如下类型。由于 E 和 T 没有继承属性，所以它们没有参数。

```
function E : ↑ syntax_tree_node;
function R(i: ↑ syntax_tree_node): ↑ syntax_tree_node;
function T : ↑ syntax_tree_node;
```

我们将图5-28中的两个 R 产生式组合在一起，以便使翻译器变得更小一些，新的产生式用 **addop** 代表 + 和 -：

$$
\begin{array}{lll}
R \rightarrow \textbf{addop} & & \\
\quad T & \{\ R_1.i := mknode(\textbf{addop}.lexeme, R.i, T.nptr)\ \} & \\
\quad R_1 & \{\ R.s := R_1.s\ \} & \\
R \rightarrow \epsilon & \{\ R.s := R.i\ \} &
\end{array}
\tag{5-5}
$$

R 的代码基于图5-30的分析过程：如果超前扫描符号是 **addop**，则使用产生式 $R \rightarrow$ **addop** TR，通过过程 *match* 读入 **addop** 之后的下一个输入记号，然后再调用 T 和 R 的过程；否则，这一过程将什么也不做以模仿产生式 $R \rightarrow \epsilon$。

```
procedure R;
begin
    if lookahead = addop then begin
        match(addop); T; R
    end
    else begin /*什么都不做*/
    end
end;
```

图5-30 产生式 $R \rightarrow$ **addop** $TR \mid \epsilon$ 的语法分析过程

图5-31中 R 的函数包含计算属性的代码。记号 **addop** 的值 *lexval* 存放在*addoplexeme*中，如果匹配 **addop**，则调用函数 T，并用 *nptr* 保存其返回结果。变量 *i1* 对应于继承属性 $R_1.i$，变量 *s1* 对应于综合属性 $R_1.s$。**return** 语句正好在控制离开函数以前返回 s 的值。类似地可以构造 E 和 T 的函数。 □

```
function R(i: ↑ syntax_tree_node): ↑ syntax_tree_node;
    var nptr, i1, s1, s: ↑ syntax_tree_node;
        addoplexeme : char;
begin
    if lookahead = addop then begin
        /* 产生式 R → addop T R */
        addoplexeme := lexval;
        match(addop);
        nptr := T;
        i1 := mknode(addoplexeme, i, nptr);
        s1 := R(i1);
        s := s1
    end
    else s := i;   /* 产生式 R → ε */
    return s
end;
```

图5-31 语法树的递归下降构造

5.6 自底向上计算继承属性

在本节中，我们给出了一种在自底向上分析框架中实现L属性定义的方法。该方法可以实现上一节所考虑的所有基于LL(1)文法的L属性定义，它还能实现许多(但不是所有的)基于LR(1)文法的L属性定义。该方法对于5.3节介绍的自底向上翻译技术来说，更具有一般性。

5.6.1 删除嵌入在翻译模式中的动作

307～308

在5.3节的自底向上翻译方法中，我们需要把所有的翻译动作都放在产生式右端的末尾，而在5.5节中的预测分析方法中我们可以在产生式右部的任何地方嵌入动作。在开始讨论如何自底向上处理继承属性之前，我们先介绍一种变换方法，它可以使翻译模式中的所有嵌入动作都出现在产生式右端的末尾。

这种变换在基础文法中加入新的形如 $M \rightarrow \epsilon$ 的产生式，其中 M 称为标记（marker）非终结符。我们将每个嵌入动作用不同的标记非终结符 M 来代替，并把该动作放在此空产生式的末端。例如，使用标记非终结符 M 和 N，可以将翻译模式

$$
\begin{array}{l}
E \rightarrow T\ R \\
R \rightarrow +\ T\ \{print('+')\}\ R \mid -\ T\ \{print('-')\}\ R \mid \epsilon
\end{array}
$$

$T \rightarrow$ **num** $\{print(\textbf{num}.val)\}$

转换成

$E \rightarrow T\ R$
$R \rightarrow +\ T\ M\ R \mid -\ T\ N\ R \mid \epsilon$
$T \rightarrow$ **num** $\{print(\textbf{num}.val)\}$
$M \rightarrow \epsilon\ \{print('+')\}$
$N \rightarrow \epsilon\ \{print('-')\}$

两个翻译模式中的文法接受相同的语言，而且，在分析树中若用额外的节点表示出动作，则可以看出动作的执行顺序也是一样的。在转换后的翻译模式中，动作都在产生式右端的末尾，因此它们可以在自底向上分析过程中刚好在产生式右部被归约之前执行。

5.6.2 分析栈中的继承属性

自底向上语法分析器对产生式 $A \rightarrow XY$ 的归约就是从分析栈顶移走 X 和 Y 并用 A 来代替它们。假设 X 有一个综合属性 $X.s$，在5.3节中我们已经将该属性与 X 一起放在分析栈中了。

因为在对 Y 的任何子树进行归约前，$X.s$ 的值已经放在分析栈中，所以该值可以被 Y 继承。也就是说，如果继承属性 $Y.i$ 是由复制规则 $Y.i := X.s$ 定义的，那么我们可以在需要 $Y.i$ 的地方使用 $X.s$ 的值。正如我们将要看到的，在自底向上的语法分析中，复制规则在继承属性的计算过程中起着重要的作用。

例5.17 如图5-32（改编自图5-7）所示，使用继承属性，标识符的类型属性可以通过复制规则来传递。我们首先分析自底向上语法分析器在输入

```
real p, q, r
```

上所做的移动，然后说明应用 L 产生式时如何获取属性 $T.type$ 的值。我们想要实现的翻译模式是：

$D \rightarrow T\ L$	$\{ L.in := T.type \}$
$T \rightarrow$ **int**	$\{ T.type := integer \}$
$T \rightarrow$ **real**	$\{ T.type := real \}$
$L \rightarrow L_1$, **id**	$\{ addtype(\textbf{id}.entry, L.in) \}$
$L \rightarrow$ **id**	$\{ addtype(\textbf{id}.entry, L.in) \}$

如果我们忽略上述翻译模式中的语义动作，语法分析器分析图5-32的输入时产生的移动顺序如图5-33所示。为清楚起见，我们用相应的文法符号代替栈状态，用实际的标识符代替记号 **id**。

我们假设像5.3节中那样，分析栈是由一对数组 *state* 和 *val* 来实现的。如果 *state*[*i*] 代表文法符号 X，那么 *val*[*i*] 就保存综合属性 $X.s$。图5-33中给出了 *state* 数组中的内容。注意，图5-33中每次归约 L 产生式的右部时，T 在栈中刚好在右部的下面。我们可以利用这一事实来

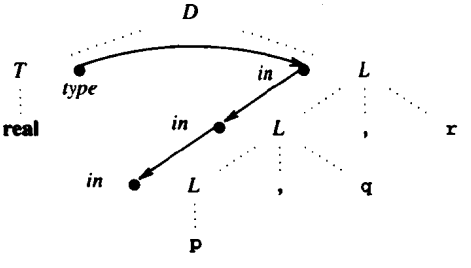


图5-32 在每个 L 节点上，$L.in = T.type$

输入	*state*	所用产生式
real p,q,r	–	
p,q,r	**real**	
p,q,r	T	$T \rightarrow$ **real**
,q,r	T p	
,q,r	$T\ L$	$L \rightarrow$ **id**
q,r	$T\ L$,	
,r	$T\ L$, q	
,r	$T\ L$	$L \rightarrow L$, **id**
r	$T\ L$,	
	$T\ L$, r	
	$T\ L$	$L \rightarrow L$, **id**
	D	$D \rightarrow T\ L$

图5-33 每次归约 L 的右部时，T 正好在右部的下面

获得属性*T.type* 的值。

图5-34中的实现利用了这样的事实：属性 *T.type* 在 *val* 栈中的位置相对于栈顶是已知的。令 *top* 和 *ntop* 分别代表归约前和归约后栈顶项的下标。根据定义 *L.in* 的复制规则，我们知道可以用 *T.type* 来代替 *L.in*。

产生式	代码段
$D \rightarrow T\ L$;	
$T \rightarrow$ **int**	*val*[*ntop*] := *integer*
$T \rightarrow$ **real**	*val*[*ntop*] := *real*
$L \rightarrow L$, **id**	*addtype*(*val*[*top*], *val*[*top* − 3])
$L \rightarrow$ **id**	*addtype*(*val*[*top*], *val*[*top* − 1])

图5-34 在*L.in*处使用*T.type*的值

当我们使用产生式 $L \rightarrow$ **id** 时，**id**.*entry* 在 *val* 栈顶，*T.type* 刚好在其下面。所以 *addtype* (*val* [*top*], *val* [*top*−1]) 等价于 *addtype* (**id**.*entry*, *T.type*)。类似地，由于产生式 $L \rightarrow L$, **id** 的右部有3个符号，所以归约时 *T.type* 出现在 *val* [*top*−3]中。因为我们用栈中的属性值 *T.type* 代替了 *L.in*，所以和 *L.in* 有关的复制规则就可以省略掉。 □

5.6.3 模拟继承属性的计算

由上面从分析栈中获取属性值的方法中可以看出，必须可以预测属性值在栈中的位置，这一方法才行得通。

例5.18 下面的语法制导定义就是一个不能预测位置的例子：

产生式	语义规则
$S \rightarrow aAC$	$C.i := A.s$
$S \rightarrow bABC$	$C.i := A.s$
$C \rightarrow c$	$C.s := g(C.i)$

(5-6)

C 通过复制规则继承了综合属性 *A.s*。注意，在栈中 *A* 和 *C* 之间可能有一个 *B* 也可能没有 *B*，因此用产生式 $C \rightarrow c$ 归约时，*C.i* 的值就可能在 *val*[*top*−1]中，也可能在 *val* [*top*−2] 中，但我们
310 ~ 311 不能确定究竟是哪种情况。

在图5-35中，引入一个新的标记非终结符 *M*，它正好插在(5-6)中第二个产生式右部 *C* 的前面。如果使用产生式 $S \rightarrow bABMC$ 进行分析，那么 *C.i* 可通过 *M.i* 和 *M.s* 间接地继承 *A.s* 的值。当使用产生式 $M \rightarrow \epsilon$ 归约时，复制规则 *M.s* := *M.i*可以保证 *M.s* = *M.i* = *A.s* 正好出现在栈中 *C* 的子树所对应的符号串的前面。所以，使用 $C \rightarrow c$ 时，*C.i* 的值就可以在 *val*[*top*−1]中找到，而不管使用的是下面的修改后的(5-6)中的第一个产生式还是第二个产生式。

产生式	语义规则
$S \rightarrow aAC$	$C.i := A.s$
$S \rightarrow bABMC$	$M.i := A.s$; $C.i := M.s$
$C \rightarrow c$	$C.s := g(C.i)$
$M \rightarrow \epsilon$	$M.s := M.i$

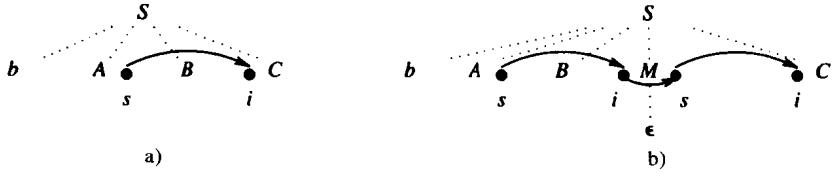


图5-35 通过标记 *M* 复制属性的值

a) 原来的产生式 b) 修改后的依赖关系

标记非终结符也可以用来模拟不是复制规则的语义规则。例如，考虑

产生式	语义规则
$S \rightarrow aAC$	$C.i := f(A.s)$

(5-7)

此时，定义$C.i$的规则不是复制规则，所以 $C.i$ 的值还没有在 val 栈中。我们仍然可以用标记来解决该问题，比如可以将文法改造为：

产生式	语义规则
$S \rightarrow aANC$	$N.i := A.s$; $C.i := N.s$
$N \rightarrow \epsilon$	$N.s := f(N.i)$

(5-8)

标记非终结符 N 用复制规则继承属性 $A.s$ 的值，其综合属性值 $N.s$ 被置为 $f(A.s)$，然后 $C.i$ 用一个复制规则来继承该值。当用 $N \rightarrow \epsilon$ 归约时，我们在放置 $A.s$ 的地方（即$val[top-1]$）找到 $N.i$的值。当我们用 $S \rightarrow aANC$ 进行归约时，$C.i$ 的值仍然可以在 $val[top-1]$中找到，因为它就是 $N.s$。实际上，此时我们并不需要 $C.i$，我们在把终结符号串归约为 C 时才需要它，那时 $C.i$ 的值已经和 N 一起安全地存放在栈中了。 □

例5.19 图5-36中使用了3个标记非终结符L，M和N，以保证在 B 的子树被归约时，继承属性 $B.ps$的值出现在分析栈中的已知位置。原来的属性文法是在图5-22给出的，它在正文编排中的应用已在例5.13中解释过。

产生式	语义规则
$S \rightarrow L\ B$	$B.ps := L.s$ $S.ht := B.ht$
$L \rightarrow \epsilon$	$L.s := 10$
$B \rightarrow B_1\ M\ B_2$	$B_1.ps := B.ps$ $M.i := B.ps$ $B_2.ps := M.s$ $B.ht := max(B_1.ht, B_2.ht)$
$B \rightarrow B_1$ **sub** $N\ B_2$	$B_1.ps := B.ps$ $N.i := B.ps$ $B_2.ps := N.s$ $B.ht := disp(B_1.ht, B_2.ht)$
$B \rightarrow$ **text**	$B.ht := \mathbf{text}.h \times B.ps$
$M \rightarrow \epsilon$	$M.s := M.i$
$N \rightarrow \epsilon$	$N.s := shrink(N.i)$

图5-36 由复制规则设置所有的继承属性

首先，用 L 进行初始化。由于图5-36中 S 的产生式是 $S \rightarrow LB$，所以 B 下面的子树被归约时，L 将留在栈中。继承属性 $B.ps = L.s$ 的值10通过与 $L \rightarrow \epsilon$ 关联的规则 $L.s := 10$ 压入分析栈中。

$B \rightarrow B_1\ M\ B_2$ 中标记 M 的作用类似于图5-35中 M 的作用，它用来保证在分析栈中 $B.ps$ 的值刚好在 B_2 的下面。在产生式 $B \rightarrow B_1$ **sub** NB_2 中，标记 N 的用法和(5-8)中的用法相同。通过复制规则 $N.i := B.ps$，N继承了 $B_2.ps$ 所依赖的属性值 $B.ps$，并通过规则 $N.s := shrink\ (N.i)$ 综合出$B_2.ps$ 的值。结果当我们归约出 B 时，$B.ps$ 的值总是在产生式右部的下面（其证明留作练习）。

实现图5-36中语法制导定义的代码段如图5-37所示。所有的继承属性都由图5-36中的复制规则来设置，所以该实现是通过跟踪继承属性在 val 栈中的位置来获得其属性值的。top 和 $ntop$ 的意义同前，分别是归约前和归约后栈顶的下标。 □

产生式	代码段
$S \rightarrow L\ B$	$val[ntop] := val[top]$
$L \rightarrow \epsilon$	$val[ntop] := 10$
$B \rightarrow B_1\ M\ B_2$	$val[ntop] := max(val[top-2], val[top])$
$B \rightarrow B_1$ **sub** $N\ B_2$	$val[ntop] := disp(val[top-3], val[top])$
$B \rightarrow$ **text**	$val[ntop] := val[top] \times val[top-1]$
$M \rightarrow \epsilon$	$val[ntop] := val[top-1]$
$N \rightarrow \epsilon$	$val[ntop] := shrink(val[top-2])$

图5-37 图5-36中语法制导定义的实现

如同在(5-6)的修改版本和(5-7)中那样，引入标记使得在LR分析期间计算L属性定义成为可

能。因为每个标记只有一个产生式，所以加入标记后的文法仍然是LL(1)文法。而任何LL(1)文法也是LR(1)文法，所以将标记加入LL(1)文法中不会引起语法分析的冲突。但是，对LR(1)文法却并非如此，因为将标记加入某些LR(1)文法中会引起语法分析的冲突。

下面的算法是上例思想的形式化描述。

算法5.3 带有继承属性的自底向上语法分析和翻译。

输入：基础文法是LL(1)文法的L属性定义。

输出：计算分析栈中所有属性值的语法分析器。

方法：为简单起见，我们假设每个非终结符 A 都有一个继承属性 $A.i$，而且每个文法符号 X 都有一个综合属性 $X.s$。如果 X 是终结符，则其综合属性值就是词法分析器所返回的词法值。和前面的例子一样，词法值也出现在栈数组 *val* 中。

对每个产生式 $A \rightarrow X_1 \cdots X_n$，引入 n 个新的标记非终结符 M_1，…，M_n，并用 $A \rightarrow M_1 X_1 \cdots M_n X_n$ 代替该产生式。[⊖]其中，综合属性 $X_j.s$ 将放在分析栈 *val* 数组内与 X_j 对应的条目中，如果有继承属性 $X_j.i$，它也出现在同一个数组中与 M_j 对应的条目中。

在语法分析时，一直保持这样的一个重要特征：如果继承属性 $A.i$ 存在，则其值可以在 *val* 数组紧贴 M_1 下面的地方找到。由于我们假设文法开始符号没有继承属性，所以此处即使文法开始符号是 A 时也满足这一特征（但是，即使开始符号有这样的继承属性，我们也可以把它放在栈底的下面）。我们可以通过对自底向上的分析步数进行归纳来证明这种特征。并依靠这样的事实来完成证明：继承属性将和标记非终结符 M_j 联系在一起，属性值 $X_j.i$ 在 M_j 处完成计算，而且该计算是在归约 X_j 之前进行的。

312
|
314

接下来，考虑如何自底向上计算属性，分为两种情况。第一种情况，当归约标记非终结符 M_j 时，它出现在产生式 $A \rightarrow M_1 X_1 \cdots M_n X_n$ 中，从而可以确定为计算继承属性 $X_j.i$ 所需要的那些属性的位置：$A.i$ 在 *val*[*top*−2*j* +2] 中，$X_1.i$ 在 *val*[*top*−2*j* +3] 中，$X_1.s$ 在 *val*[*top*−2*j* +4] 中，$X_2.i$ 在 *val*[*top*−2*j* +5] 中，依次类推。于是，我们就可以计算出 $X_j.i$，并将其存放在 *val*[*top*+1] 中，归约后它成为新的栈顶。注意此时要求是LL(1)文法，否则就不能断定 ϵ 应归约为哪一个非终结符，因而也就无从知道该执行什么语义动作以及判断所需属性的位置。关于LL(1)文法加上标记后仍然是LR(1)文法的证明留作练习。

第二种情况是归约非标记符号，譬如用产生式 $A \rightarrow M_1 X_1 \cdots M_n X_n$ 进行归约。在这种情况下，我们只需要计算综合属性 $A.s$，因为 $A.i$ 已经被计算出来了，而且放在栈中 A 要插入的位置之下。很明显，进行归约时，计算 $A.s$ 所需要的其他属性也都在栈中的已知位置，即 $X_j\ (1 \leqslant j \leqslant n)$ 所在的位置。

下面的简化可以减少标记的个数，其中第二条还可以避免左递归文法中的语法分析冲突：

1. 如果 X_j 没有继承属性，我们就不需使用标记 M_j。当然，如果省略了 M_j，属性在栈中的预期位置就会改变，但是语法分析器可以很容易地适应这种变化。

2. 如果 $X_1.i$ 存在，但它是由复制规则 $X_1.i := A.i$ 计算的，那么我们可以省略 M_1。因为我们知道 $A.i$ 存放在栈中紧挨 X_1 下面的地方，因此该值也可同时作为 $X_1.i$ 的值。 □

5.6.4 用综合属性代替继承属性

有时我们可以通过改变基础文法来避免使用继承属性。例如，在 Pascal 中，声明语句由标

⊖ 虽然在 X_1 之前插入 M_1 可以简化标记非终结符的讨论，但不幸的是，它具有副作用，即可能将语法分析冲突引入左递归文法。参见练习5.21。正如下面要说明的，M_1 可以被删除。

识符表加上类型组成，如 m, n: integer。该声明可以用包含下列产生式的文法产生：

$D \rightarrow L : T$
$T \rightarrow$ **integer** | **char**
$L \rightarrow L$, **id** | **id**

315

因为标识符是由 L 产生的，但类型却不在 L 的子树中，因此仅仅使用综合属性不能把类型和标识符联系在一起。事实上，如果非终结符 L 在第一个产生式中从其右边的 T 中继承类型的话，我们将得到一个不是L属性的语法制导定义，从而基于它的翻译也就不能在语法分析期间进行。

这个问题可以通过重新构造文法来解决（将类型作为标识符表的最后一个元素）：

$D \rightarrow$ **id** L
$L \rightarrow$, **id** L | : T
$T \rightarrow$ **integer** | **char**

现在，类型信息可以作为 L 的综合属性 $L.type$，当标识符由 L 产生时，其类型就能被填进符号表中。

5.6.5 一个难计算的语法制导定义

在自底向上语法分析过程中计算继承属性的算法5.3可以扩展到某些（但不是全部）LR文法上。图5-38中的L属性定义基于一个简单的LR(1)文法，但它不能在LR语法分析期间用算法5.3进行计算。$L \rightarrow \epsilon$ 中的非终结符 L 继承由 S 产生的 1 的个数。但由于自底向上语法分析器首先使用产生式 $L \rightarrow \epsilon$ 进行归约，所以此时翻译器不可能知道输入中 1 的个数。

产生式	语义规则
$S \rightarrow L$	$L.count := 0$
$L \rightarrow L_1$ 1	$L_1.count := L.count + 1$
$L \rightarrow \epsilon$	$print(L.count)$

图5-38 一个难于计算的语法制导定义

5.7 递归计算

可以用5.5节中的预测翻译技术在语法制导定义的基础上构造递归函数，使其可以在遍历分析树时计算出属性值。这种递归函数使我们可以实现一些不能与分析同时实现的语法制导定义。在本节中，我们将每个非终结符都与单个翻译函数联系起来。此函数以某种顺序访问非终结符节点的各子节点，这种顺序是由该非终结符节点对应的产生式决定的，而且不一定是从左到右的。5.10节将描述如何通过将非终结符和多个过程联系起来，以达到多于一遍扫描所得到的翻译效果。

316

5.7.1 从左到右遍历

在算法5.2中，通过为每个非终结符构造一个递归函数，可以分析和翻译任何基于LL(1)文法的L属性定义。同样，在已构造好的分析树上，在非终结符节点上调用类似的递归函数，也可以实现所有的L属性语法制导定义。通过节点的产生式，这种函数可以确定其子节点是什么，进而知道在这些子节点上应调用什么函数。非终结符 A 对应的函数以分析树上的 A 节点及其继承属性值作为参数，并返回其综合属性值。

除第2步以外，构造的细节与算法5.2相同，只是算法5.2中非终结符的函数是根据当前的输入符号来决定使用哪个产生式的，而此处函数是用case语句来确定节点所用的产生式的。我们用一个例子来说明这种方法。

例5.20 考虑图5-22中确定公式大小和高度的语法制导定义。非终结符 B 有一个继承属性 ps 和一个综合属性 ht。用上述修改后的算法5.2为 B 构造出的函数如图5-39所示。

函数 B 以节点 n 和节点 n 处 $B.ps$ 的值作为参数，并返回节点 n 处 $B.ht$ 的值。该函数用case语句来区分各种 B 产生式。与每个产生式相应的代码用来模拟与该产生式相关的语义规则，而且这些规则的使用顺序必须满足：非终结符的继承属性必须在调用该非终结符的函数之前计算出来。

```
function B(n, ps);
    var ps1, ps2, ht1, ht2;
begin
    case production at node n of
    'B → B1 B2':
        ps1 := ps;
        ht1 := B(child(n, 1), ps1);
        ps2 := ps;
        ht2 := B(child(n, 2), ps2);
        return max(ht1, ht2);
    'B → B1 sub B2':
        ps1 := ps;
        ht1 := B(child(n, 1), ps1);
        ps2 := shrink(ps);
        ht2 := B(child(n, 3), ps2);
        return disp(ht1, ht2);
    'B → text':
        return ps × text.h;
    default:
        error
    end
end;
```

图5-39 图5-22中非终结符B的函数

在对应于产生式 $B \rightarrow B$ **sub** B 的代码中，变量 ps，ps_1 和 ps_2 用来保存继承属性 $B.ps$，$B_1.ps$ 和 $B_2.ps$ 的值。同样地，ht，ht_1 和 ht_2 用来保存综合属性 $B.ht$，$B_1.ht$ 和 $B_2.ht$ 的值。我们用函数 $child(m, i)$来表示节点 m 的第 i 个子节点。因为 B_2 是节点 n 的第3个子节点的标记，所以 $B_2.ht$ 的值由函数调用 $B(child(n, 3), ps2)$来确定。 □

5.7.2 其他遍历方法

一旦获得一棵清楚的分析树，我们就可以按任意顺序访问一个节点的各子节点。考虑例5.21中非L属性定义的例子，在该定义所表示的翻译中，一个产生式对应节点的各子节点要求按从左到右的顺序访问各子节点，另一个产生式对应节点的子节点则要求从右到左地访问。

这个抽象的例子将说明在计算分析树节点的属性值时，使用相互递归函数可以获得强有力的效果。此时，函数不必依赖于分析树节点的建立顺序，而所要考虑的主要问题是：节点继承属性的值应该在第一次访问该节点之前计算出来；而节点综合属性的值应该在最后一次离开该节点之前计算出来。

例5.21 图5-40中的每一个非终结符都有一个继承属性 i 和一个综合属性 s。图5-40还给出了两个产生式的依赖图。与产生式 $A \rightarrow LM$ 对应的规则建立了从左到右的依赖，而与产生式 $A \rightarrow QR$ 对应的规则建立了从右到左的依赖。

产生式	语义规则
$A \rightarrow L\ M$	$L.i := l(A.i)$ $M.i := m(L.s)$ $A.s := f(M.s)$
$A \rightarrow Q\ R$	$R.i := r(A.i)$ $Q.i := q(R.s)$ $A.s := f(Q.s)$

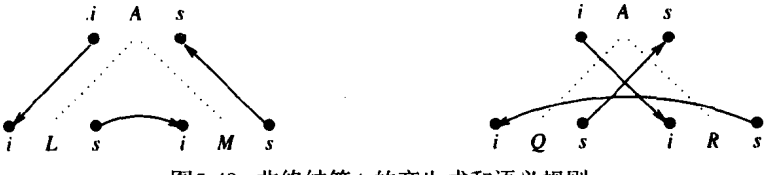


图5-40 非终结符A的产生式和语义规则

非终结符 A 的函数如图5-41所示：我们假设 L，M，Q 和 R 的函数也可被建立。图5-41中

的变量是由非终结符及其属性来命名的，如 *li* 和 *ls* 分别是与 *L.i* 和 *L.s* 对应的变量。

与产生式 $A \rightarrow LM$ 对应的代码的构造方法同例5.20，即首先计算 *L* 的继承属性，然后调用 *L* 的函数计算其综合属性，再对 *M* 进行类似的处理。而与 $A \rightarrow QR$ 对应的代码则先访问 *R* 的子树，然后再访问 *Q* 的子树。除此以外，两个产生式的其他代码几乎相同。 □

5.8 编译时属性值的空间分配

本节将讨论编译时属性值的空间分配。因为我们要用到分析树依赖图的信息，所以本节的方法适合于用依赖图来确定计算顺序的情况。下一节我们将考虑一种能预知计算顺序的情况，那样的话，在构造编译器的时候，我们可以一次性地确定所有属性的空间。

我们给属性定义一个*生存期*的概念，它可以以属性的深度优先计算顺序为基础。属性的生存期在它第一次被计算时开始，在所有依赖它的属性都被计算完之后结束。为节省空间，我们可以只在属性的生存期内才为其分配空间。

为强调本节的技术可以用于任何计算顺序中，我们考虑下面的非L属性语法制导定义，它的作用是在声明语句中将类型信息传递给标识符。

```
function A(n, ai);
begin
    case 节点n 处的产生式 of
    'A → L M':      /* 从左到右的顺序 */
        li := l(ai);
        ls := L(child(n, 1), li);
        mi := m(ls);
        ms := M(child(n, 2), mi)
        return f(ms);
    'A → Q R':      /* 从右到左的顺序 */
        ri := r(ai);
        rs := R(child(n, 2), ri);
        qi := q(rs);
        qs := Q(child(n, 1), qi);
        return f(qs);
    default:
        error
    end
end;
```

图5-41 图5-40的依赖关系决定了子节点被访问的顺序

例5.22 图5-42的语法制导定义是图5-4的扩展，它可以包含如下形式的声明语句：

```
real c[12][31];                    (5-9)
int  x[3], y[5];                   (5-10)
```

(5-10) 的分析树如图5-43a中的虚线所示。节点上的数字将在下个例子中讨论。同例5.3，*L* 将继承从 *T* 获得的类型信息，并将该信息向下传递给声明语句中的标识符。从 *T.type* 到 *L.in* 的边表明属性 *L.in* 依赖于属性 *T.type*。图5-42中的语法制导定义不是 L 属性定义，这是因为 $I_1.in$ 依赖于 **num**.*val*，而在产生式 $I \rightarrow I_1$[**num**] 中，**num** 在 I_1 的右边。 □

产生式	规　则
$D \rightarrow T\ L$	*L.in* := *T.type*
$T \rightarrow$ **int**	*T.type* := *integer*
$T \rightarrow$ **real**	*T.type* := *real*
$L \rightarrow L_1$, I	$L_1.in$:= *L.in* *I.in* := *L.in*
$L \rightarrow I$	*I.in* := *L.in*
$I \rightarrow I_1$ [**num**]	$I_1.in$:= *array*(**num**.*val*, *I.in*)
$I \rightarrow$ **id**	*addtype*(**id**.*entry*, *I.in*)

图5-42 在声明语句中将类型信息传递给标识符

5.8.1 在编译时为属性分配空间

假设我们用一系列寄存器来保存属性的值。为方便起见，我们假设每个寄存器都可以保存任何属性值。如果属性代表不同的类型，我们就将占用相同存储空间的属性分为一组，然后分开考虑每个组。我们将依据属性生存期的信息来确定寄存器的使用。

317 ~ 320

例5.23 假设我们按依赖图5-43[⊖]中各节点上的数字顺序来计算属性，各节点的生存期从第一次计算其属性开始，到最后一次使用其属性结束。例如，在计算节点2时，节点1的生存期将会结束，这是因为节点2是依赖于节点1的惟一节点。同样，节点2的生存期将在计算节点6时结束。 □

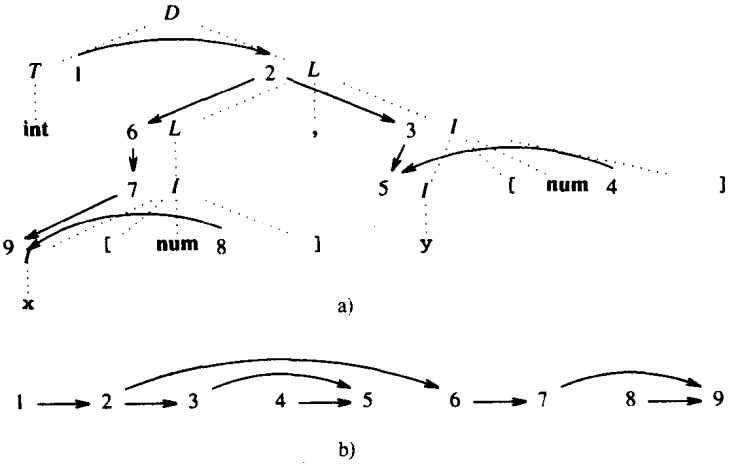


图5-43 确定属性值的生存期

a) 一个语法树的依赖图 b) a)的节点计算顺序

图5-44给出了一种使用尽可能少的寄存器来计算属性值的方法。我们按节点的计算顺序依次考虑分析树依赖图 D 中的各节点。初始时，我们有一个寄存器池 r_1，r_2，…。如果属性 b 是由语义规则 $b := f(c_1, c_2, \cdots, c_k)$ 定义的，那么 c_1，c_2，…，c_k 中某些属性的生存期可能会随着对 b 的计算而结束，于是，计算完 b 以后，保存这些属性的寄存器就可以被释放。尽可能把属性值 b 保存在某个被释放的寄存器中。

```
for m1, m2, …, mN 中的每个节点m do begin
    for 每个生存期在计算 m 时结束的节点 n do
        标记 n 的寄存器;
    if 某个寄存器r被标记了 then begin
        清除 r 的标记;
        计算 m，并将其保存到寄存器 r 中;
        将标记寄存器返回到寄存器池
    end
    else /* 没有寄存器打上标记 */
        计算 m，并将其保存到寄存器池中的某个寄存器中;
    /* 使用属性值 m 的动作可以插在这里 */
    if m 的生存期已经结束 then
        将 m 的寄存器返回到寄存器池
end
```

图5-44 为属性值分配寄存器

在计算依赖图5-43中各属性值时，寄存器的使用情况如图5-45所示。一开始先计算节点1，并将结果保存到寄存器 r_1 中。节

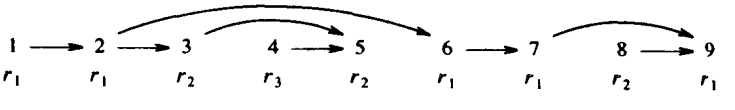


图5-45 用于保存图5-43中属性值的寄存器

⊖ 图5-43中的依赖图没有给出与语义动作 *addtype*(**id**.*entry*, *I.in*) 相对应的节点，这是因为没有给虚属性分配空间。但请注意，直到 *I.in* 的值可用时才能计算该语义规则。确定该事实的算法必须工作在包含该语义规则所对应的节点的依赖图上。

点1的生存期在计算节点2时结束，所以节点2的计算结果将保存在寄存器 r_1 中。由于节点6需要节点2的值，所以需从寄存器池中为节点3分配一个新的寄存器，即 r_2。

5.8.2 避免复制

通过将复制规则看成一种特殊情况，我们可以改进图5-44的算法。复制规则是形如*b*: = *c*的规则，所以如果 *c* 的值在寄存器 *r* 中，那么 *b* 的值也就出现在寄存器 *r* 中。复制规则所定义的属性编号可能非常多，所以我们想要避免显式复制。 321 ~ 322

一组具有相同属性值的节点形成一个等价类。我们可以向下面这样来修改图5-44的算法，以便将一个等价类的属性值保存在一个寄存器中。考虑节点 *m* 时，我们首先检查它是否是由复制规则定义的，如果是，其属性值就必定已经保存在寄存器中了，则将 *m* 加入到相应的等价类中。此外，对于寄存器的释放，需等到相应等价类中所有节点的生存期都结束之后，该寄存器才能返回寄存器池。

例5.24 图5-43的依赖图可以重画为图5-46，其中加等号的节点表示其属性值是由复制规则定义的。根据图5-42的语法制导定义，我们发现由节点1所确定的类型信息被复制到标识符列表中的每一个元素上，这就导致在图5-43中节点2, 3, 6和7的属性值都是节点1的复制。

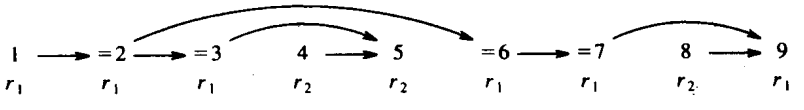


图5-46 考虑复制规则的寄存器分配图

在图5-46中，因为节点2和节点3都是节点1的复制，所以它们的值都取自于寄存器 r_1。节点3的生存期在计算节点5时结束，但是由于等价类中节点2的生存期还没有结束，所以保存节点3的值的寄存器 r_1 就不能返回到寄存器池。

下面的代码说明编译器如何对例5.22中的声明(5-10)进行处理：

```
r1 := integer;          /* 计算节点1, 2, 3, 6, 7 */
r2 := 5;                /* 计算节点4 */
r2 := array(r2, r1);    /* y的类型 */
addtype(y, r2);
r2 := 3;                /* 计算节点8 */
r2 := array(r2, r1);    /* x的类型 */
addtype(x, r2);
```

在上面的代码中，x 和 y 是指向符号表中 x 和 y 表项的指针。适当的时候必须调用过程 *addtype* 以便将x和y的类型信息添加到相应的符号表表项中。 □

5.9 编译器构造时的空间分配

尽管在一次遍历中有可能将全部属性值保存在一个栈中，但有的时候用多个栈可以避免复制。通常，如果属性间的依赖关系使得属性值不便于放在栈中，那么可以将属性值直接保存在语法树节点上。 323

我们已经在5.3节和5.6节学习了如何在自底向上语法分析中用栈来保存属性值。同时，在递归下降语法分析器中栈还可以隐式地用来保存过程调用的断点，这个问题将在第7章讨论。

为节省空间，栈可以和其他技术相结合。第2章翻译模式中广泛使用的打印动作尽可能将字符串类型的属性值写到一个输出文件中。构造5.2节的语法树时，我们只传递指向节点的指

针，而不是整棵子树。通常，传递指向对象的指针而不是直接传递对象可以节省空间。例5.27和例5.28就应用了这些技术。

5.9.1 从文法中预知生存期

如果属性的计算顺序是通过对分析树的特殊遍历获得的，那么我们就可以在编译器构造时预知属性的生存期。例如，像5.4节中那样，假设我们在深度优先遍历过程中是从左到右访问子节点的。在产生式 $A \rightarrow BC$ 对应的节点开始遍历时，B 子树先被遍历，然后遍历 C 子树，最后遍历根节点 A。因为 A 的父节点不能引用 B、C 节点的属性，所以当遍历到 A 时，B 和 C 的生存期就结束了。注意，上述结论基于产生式 $A \rightarrow BC$ 以及非终结符节点的访问顺序。在 B 和 C 对应的节点处我们不需要知道它们的子树。

对于任何计算顺序，如果属性 c 的生存期包含在属性 b 的生存期中，那么栈中 c 的值可以在b 的上面，这里的 b 和 c 不必都是同一个非终结符的属性。对于产生式 $A \rightarrow BC$，我们可以在深度优先遍历中按照如下方法来使用栈。

首先从其继承属性已在栈中的节点 A 开始，然后计算 B 的继承属性值并将其压入栈中。当遍历 B 的子树时，这些值仍留在栈中。从 B 的子树返回时，B 的综合属性值压在这些值的上面。对 C 重复上述过程，亦即压入其继承属性，遍历其子树，然后将返回的综合属性压入栈顶。如果将 X 的继承属性和综合属性分别写成 $\mathbf{I}(X)$和 $\mathbf{S}(X)$，那么此时栈中包含：

$$\mathbf{I}(A), \mathbf{I}(B), \mathbf{S}(B), \mathbf{I}(C), \mathbf{S}(C) \tag{5-11}$$

计算A的综合属性所需要的所有属性值现在都已经在栈中了，所以我们可以返回到A，相应的栈包含：

$$\mathbf{I}(A), \mathbf{S}(A)$$

注意，文法符号的继承属性和综合属性的个数是固定的（假设尺寸也是固定的），所以，
324 上述过程中的每一步我们都知道某个属性离栈顶有多远。

例5.25 假设图5-22的排版翻译将属性值保存在上面所讨论的栈中。对于产生式 $B \rightarrow B_1 B_2$，从B节点开始遍历，栈顶为 $B.ps$。访问每个节点之前和之后栈的内容如图5-47所示。同往常一样，栈是向下增长的。

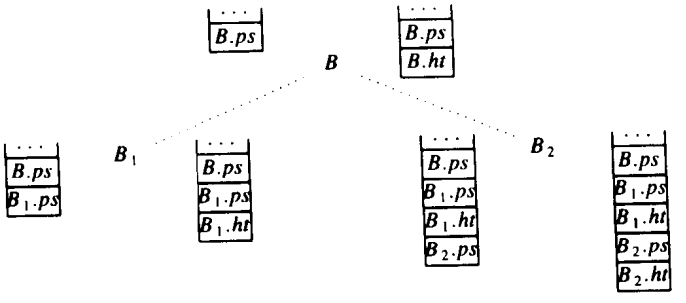


图5-47 遍历节点之前和之后栈的内容

注意，刚好在 B 节点第一次被访问之前，其 ps 属性位于栈顶。刚好在最后一次访问之后，即遍历从 B 节点向上移时，其 ht 属性位于栈顶，ps 属性位于次栈顶。 □

如果属性b是由复制规则 $b := c$ 定义的，而且 c 的值位于属性值栈的栈顶，那么可能没有必要将 c 的复制压入栈中。另外，如果使用多个栈来保存属性的值，我们就有更多的机会消除

复制规则。在下面的例子中，我们将综合属性和继承属性分成两个栈存放，与例5.25相比，这种方法可以消除更多的复制规则。

例5.26 对图5-22的语法制导定义，假设我们用分开的两个栈分别保存继承属性 *ps* 和综合属性 *ht*。我们将对这两个栈进行维护，以保证在 *B* 第一次被访问之前以及最后一次被访问之后，*B.ps* 都处于 *ps* 栈的栈顶，同时保证在 *B* 刚好被访问之后 *B.ht* 位于 *ht* 栈的栈顶。

把栈分开，我们就可以利用和产生式 $B \rightarrow B_1 B_2$ 相联系的两个复制规则 $B_1.ps := B.ps$ 和 $B_2.ps := B.ps$。如图5-48所示，因为 *B.ps* 的值已处于栈顶，所以没必要将 $B_1.ps$ 压入栈。

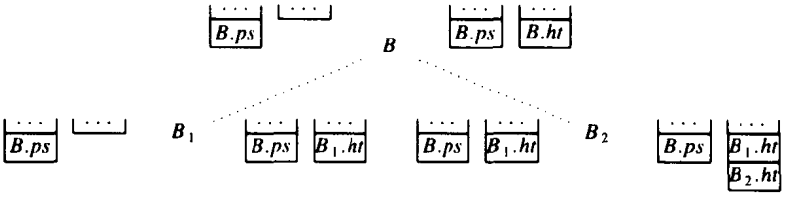


图5-48 用不同的栈存放属性 *ps* 和 *ht*

基于图5-22语法制导定义的一个翻译模式如图5-49所示，其中 *push*(*v*, *s*) 操作把值 *v* 压入栈 *s*，*pop*(*s*) 操作从栈 *s* 中弹出一个值，*top*(*s*) 表示栈 *s* 的栈顶元素。 □

```
S →      { push(10, ps) }
   B
B → B1
   B2    { h2 := top(ht); pop(ht);
           h1 := top(ht); pop(ht);
           push(max(h1, h2), ht) }
B → B1
   sub   { push(shrink(top(ps)), ps) }
   B2    { pop(ps);
           h2 := top(ht); pop(ht);
           h1 := top(ht); pop(ht);
           push(disp(h1, h2), ht) }
B → text { push(text.h × top(ps), ht) }
```

图5-49 维护*ps* 栈和 *ht* 栈的翻译模式

下面的例子中，我们将属性值栈的使用同输出代码的动作结合起来。

例5.27 这里我们考虑一种描述中间代码生成的语法制导定义实现技术。如果 *E* 为假，那么布尔表达式 "*E* **and** *F*" 的值就为假。在 C 语言中，如果 *E* 为假则不必计算子表达式 *F* 的值。8.4节将讨论这种布尔表达式的计算方法。

图5-50是布尔表达式的语法制导定义，其中布尔表达式是用标识符和 **and** 运算符构造的。每个表达式 *E* 有两个继承属性 *E.true* 和 *E.false*，它们是两个标号，分别表示 *E* 为 *true* 或 *false* 时程序控制要跳转到的地方。 325 ~ 326

产生式	语义规则
$E \rightarrow E_1$ **and** E_2	$E_1.true := newlabel$ $E_1.false := E.false$ $E_2.true := E.true$ $E_2.false := E.false$ $E.code := E_1.code \parallel gen(\text{'label'}\ E_1.true) \parallel E_2.code$
$E \rightarrow$ **id**	$E.code := gen(\text{'if'}\ \textbf{id}.place\ \text{'goto'}\ E.true) \parallel$ $gen(\text{'goto'}\ E.false)$

图5-50 布尔表达式的短路计算法

对于 $E \rightarrow E_1$ **and** E_2，如果 E_1 为 *false*，那么控制流就转移到继承标号 *E.false*；若 E_1 为 *true*，则将控制流转移到计算 E_2 的代码。函数 *newlabel* 产生的标号标记了 E_2 代码的开始，*gen* 用来

产生单条指令。图5-50和中间代码生成的相关性的进一步讨论请参见8.4节。

图5-50中的语法制导定义是L属性定义，因此我们可以为它构造一个翻译模式。图5-51的翻译模式用过程 *emit* 来产生并输出指令代码，指令编号逐次增加。正如5.4节所讨论的，图中还给出了设置继承属性值的动作，并被插在合适的文法符号之前。

图5-52是更进一步的翻译模式，它用两个不同的栈分别保存继承属性 *E.true* 和 *E.false* 的值。像例5.26中那样，复制规则对栈没有影响。为实现规则 $E_1.true := newlable$，在访问 E_1 之前将一个新的标号压入 *true* 栈。该标号的生存期在动作 *emit*('**label**'*top*(*true*))（与 *emit* ('**label**'E_1.*true*) 对应的动作）之后结束，因此在该动作之后要对*true*栈执行一次*pop*操作。在本例中，*false*栈没有变化，但是当布尔表达式中除了 **and** 运算符之外还允许 **or** 运算符时，它就会发生变化。 □

```
E → E1 and E2     { E1.true := newlabel;
                    E1.false := E.false }
                  { emit('label' E1.true) };
                    E2.true := E.true;
                    E2.false := E.false }
E → id            { emit('if' id.place 'goto' E.true);
                    emit('goto' E.false) }
```

图5-51 布尔表达式的代码生成

```
E → E1 and E2     { push(newlabel, true) }
                  { emit('label' top(true));
                    pop(true) }
E → id            { emit('if' id.place 'goto' top(true));
                    emit('goto' top(false)) }
```

图5-52 布尔表达式的代码生成

327

5.9.2 不相重叠的生存期

单个寄存器是栈的特例。如果每个*push*操作之后都跟随一个*pop*操作的话，那么任一时刻栈中至多只有一个元素。在这种情况下，我们可以用一个寄存器来代替栈。根据生存期的定义，如果两个属性的生存期不相重叠，那么它们的值就可以保存在同一个寄存器中。

例5.28 图5-53的语法制导定义用来为类似于列表的表达式构造语法树，这种表达式中的各运算符优先级相同。该语法制导定义来自图5-28的翻译模式。

产生式	语义规则
$E \rightarrow T\ R$	$R.i := T.nptr$ $E.nptr := R.s$
$R \rightarrow$ **addop** $T\ R_1$	$R_1.i := mknode($**addop**$.lexeme, R.i, T.nptr)$ $R.s := R_1.s$
$R \rightarrow \epsilon$	$R.s := R.i$
$T \rightarrow$ **num**	$T.nptr := mkleaf($**num**, **num**$.val)$

图5-53 从图5-28修改得来的语法制导定义

R 的每个属性的生存期在依赖于该属性的各属性都被计算完以后结束。可以证明，对任何分析树，*R* 的属性可以在同一个寄存器 *r* 中完成计算。图5-54的推理过程是分析文法的典型方法，其基本思想是对 *R* 子树的大小进行归纳。

应用 $R \rightarrow \epsilon$ 时，得到的是最小的 *R* 子树，此时，*R.s* 是 *R.i* 的复制，所以其值都保存在寄存器 *r* 中。

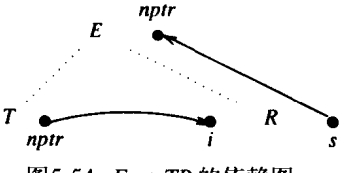


图5-54 $E \rightarrow TR$ 的依赖图

对于较大的 R 子树，子树根节点对应的产生式必定是 $R \rightarrow$ **addop** TR_1。$R.i$ 的生存期在计算 $R_1.i$ 时结束，因此 $R_1.i$ 就可以保存在寄存器 r 中。由归纳假设，R_1 子树可以在同一个寄存器中完成计算，所以 R 子树的计算也可以在同一个寄存器中完成。最后，由于 $R.s$ 是从 $R_1.s$ 复制而来的，因此其值已经保存在寄存器 r 中了。

图5-55的翻译模式用来计算图5-53中属性文法的属性，它只用一个寄存器 r 来保存所有非终结符 R 实例的 $R.i$ 和 $R.s$ 属性值。

为完整起见，我们在图5-56中给出实现上述翻译模式的代码，它是根据算法5.2构造出来的。因为非终结符 R 不再具有属性，所以 R 就成了一个过程而不是函数。函数 E 中，r 是局部变量，因此我们可以递归调用函数 E，虽然在图5-55的模式中我们不需要这样做。我们可以像在2.5节中那样将代码进一步改进，方法是：消除尾递归，并用结果过程体替换 R 的其他调用。 □

```
E → T       { r := T.nptr  /* r 现在保存R.i */ }
    R       { E.nptr := r  /* r 返回值 R.s */}
R → addop
    T       { r := mknode(addop.lexeme, r, T.nptr) }
    R
R → ϵ
T → num     { T.nptr := mkleaf(num, num.val) }
```

图5-55 转换后的构造语法树的翻译模式

```
function E: ↑ syntax_tree_node;
    var r: ↑ syntax_tree_node;
        addoplexeme : char;

    procedure R;
    begin
        if lookahead = addop then begin
            addoplexeme := lexval;
            match(addop);
            r := mknode(addoplexeme, r, T);
            R
        end
    end;

begin
    r := T; R
    return r
end;
```

图5-56 将过程 R 和图5-31中的代码进行比较

5.10 语法制导定义的分析

在5.7节中，属性是在遍历分析树时用一组相互递归调用的函数来计算的。每一个非终结符对应的函数将节点的继承属性值映射成节点的综合属性值。 328 ~ 329

5.7节中的方法可以扩展到不能在一次深度优先遍历中完成的翻译中。此时，我们将为每个非终结符的每个综合属性分别定义一个函数。虽然有些综合属性可以形成一组，并可用一个函数来计算，5.7节处理的就是将所有综合属性当成一组一起计算的特殊情况。属性如何分组由语法制导中语义规则所建立的依赖关系来确定。下面的例子用于说明如何构造递归计算程序。

例5.29 图5-57中的语法制导定义是针对一个实际问题的，该问题我们要在第6章进行讨论。简单地说，该问题的基本意思是："重载"的标识符可以有一组可能的类型，因此表达式也可以有一组可能的类型。需要用上下文信息为每个子表达式选择一个可能的类型。通过自底向上扫描，综合出可能类型的集合，再自顶向下扫描，将该集合缩小成单个类型即可解决该问题。

图5-57中的语义规则是该问题的抽

产生式	语义规则
$S \rightarrow E$	$E.i := g(E.s)$ $S.r := E.t$
$E \rightarrow E_1\ E_2$	$E.s := fs(E_1.s, E_2.s)$ $E_1.i := fi1(E.i)$ $E_2.i := fi2(E.i)$ $E.t := ft(E_1.t, E_2.t)$
$E \rightarrow$ **id**	$E.s :=$ **id**.s $E.t := h(E.i)$

图5-57 综合属性 s 和 t 不能一起计算

象描述。综合属性 s 代表可能类型的集合，继承属性 i 代表上下文信息。另外一个综合属性 t 代表生成的代码或子表达式的类型，它不能在扫描计算 s 的同一遍中完成计算。图5-57中产生式的依赖图如图5-58所示。 □

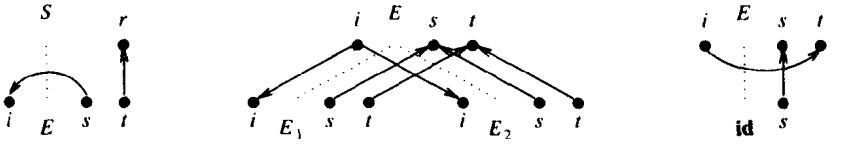


图5-58 图5-57中产生式的依赖图

5.10.1 属性的递归计算

每一个产生式的语义规则都形成一个依赖子图，分析树的依赖图就是在此基础上组装而成的。产生式 p 的依赖图 D_p 只是基于该产生式的语义规则，即基于该产生式左部综合属性的语义规则以及产生式右部文法符号继承属性的语义规则。也就是说，D_p 只给出了局部依赖关系。例如，图5-58中 $E \rightarrow E_1\ E_2$ 的依赖图中，所有边都在同类属性（如 s 属性）的不同实例之间。从该依赖图我们还看不出 s 属性必须在其他属性之前计算。

图5-59是分析树的整体依赖图，对其详细分析可以发现，非终结符 E 的每个实例的属性必须按照 $E.s$，$E.i$ 和 $E.t$ 的顺序计算。注意，图5-59中的所有属性都可以在 3 遍扫描中完成计算：自底向上扫描以计算 s 属性，自顶向下扫描以计算 i 属性，最后再自底向上扫描以计算 t 属性。

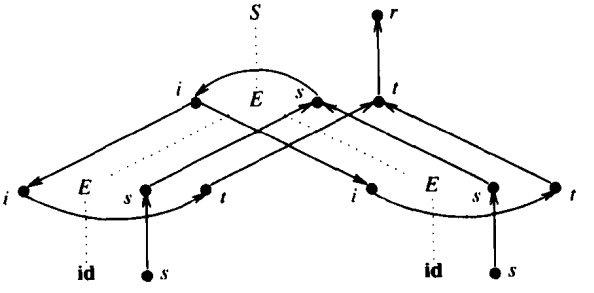


图5-59 分析树的依赖图

递归计算程序中，计算综合属性的函数以某些继承属性的值为参数。一般地，如果综合属 $A.a$ 依赖于继承属性 $A.b$，那么计算 $A.a$ 的函数就以 $A.b$ 为参数。在分析依赖关系之前，我们用下例来说明其应用。

330
~
331

例5.30 图5-60中的函数 Es 和 Et 返回标号为 E 的节点 n 的综合属性 s 和 t 的值。正如5.7节所讨论的，非终结符对应的函数根据所用产生式来分情况考虑，每种情况下执行的代码分别计算图5-57中与该产生式相关的语义规则。

由上述对图5-59中依赖图的讨论，我们知道分析树中节点 E 的属性 $E.t$ 可能依赖于属性 $E.i$，因此继承属性 i 是函数 Et 的参数。因为属性 $E.s$ 不依赖于任何继承属性，所以函数 Es 没有属性值参数。 □

5.10.2 强无环的语法制导定义

我们可以为一类称为“强无环”定义的语法制导定义构造递归计算程序。对这类定义，一个非终结符所有实例节点的属性都可按同样的顺序（偏序）进行计算。当我们构造计算该非终结符综合属性的函数时，该顺序可以用来选择作为函数参数的继承属性。

我们将形式化地定义这类语法制导定义，并证明图5-57的语法制导定义就属于此类。接下来，我们给出一个是否有环及是否强无环的检测算法，并说明如何将例5.30的实现方法扩展到所有的强无环语法制导定义上。

考虑在分析树节点 n 处的非终结符 A。在分析树的依赖图中通常可能有这样的路径：从节点 n 的某一属性开始，经过分析树其他节点的属性，到 n 的另一个属性结束。对我们而言，只需观察分析树中 A 以下的路径。显然，这样的路径是从 A 的某个继承属性到 A 的某个综合属性的，我们将通过考察 A 的属性的偏序关系来对这种路径的集合进行估计。

令产生式 p 右部的非终结符为 $A_1, A_2, \cdots, A_n$，令 RA_j 是 $A_j(1 \leqslant j \leqslant n)$ 的属性上的偏序关系。在 D_p 中，如果在RA_j中属性 $A_j.b$先于 $A_j.c$，则加上从 $A_j.b$ 到 $A_j.c$ 的边，依次考察各非终结符的各属性即可得到一个依赖图：记为 $D_p[RA_1, RA_2, \cdots, RA_n]$。

一个语法制导定义是强无环的，如果对每个非终结符 A，我们都可以找到 A 的属性上的一个偏序关系 RA，使得对每个以 A 为左部，$A_1, A_2, \cdots, A_n$ 出现在右部的产生式p都有：

1. $D_p[RA_1, RA_2, \cdots, RA_n]$是无环的；并且

2. 如果 $D_p[RA_1, RA_2, \cdots, RA_n]$中存在从属性 $A.b$ 到 $A.c$ 的边，那么 RA 中 $A.b$ 先于 $A.c$。

例5.31 令 p 是图5-57中的产生式 $E \rightarrow E_1 E_2$，其依赖图 D_p 如图5-58中间所示。令 RE 是 E 的属性上的偏序关系（此时是全序关系）$s \rightarrow i \rightarrow t$。非终结符 E 在 p 的右部出现两次，和通常一样，写成 E_1 和 E_2。于是，RE_1, RE_2 和 RE 是相等的，图 $D_p[RE_1, RE_2]$如图5-61所示。

图5-61中和根 E 相关的属性中，只有一条路径，即从 i 到 t 的路径。因为 RE 中 i 先于 t，所以不违反第2个条件。 □

```
function Es(n);
begin
    case 节点n处的产生式 of
    'E → E1 E2':
        s1 := Es(child(n, 1));
        s2 := Es(child(n, 2));
        return fs(s1, s2);
    'E → id':
        return id.s;
    default:
        error
    end
end;

function Et(n, i);
begin
    case 节点n处的产生式 of
    'E → E1 E2':
        i1 := fi1(i);
        t1 := Et(child(n, 1), i1);
        i2 := fi2(i);
        t2 := Et(child(n, 2), i2);
        return ft(t1, t2);
    'E → id':
        return h(i);
    default:
        error
    end
end;

function Sr(n);
begin
    s := Es(child(n, 1));
    i := g(s);
    t := Et(child(n, 1), i);
    return t
end;
```

图5-60 计算图5-57中综合属性的函数

给定强无环定义和每个非终结符A的属性上的偏序关系 RA，计算 A 综合属性 s 的函数的参数为：如果 RA 中继承属性 i 先于 s，那么 i 就是该函数的参数；否则不是。

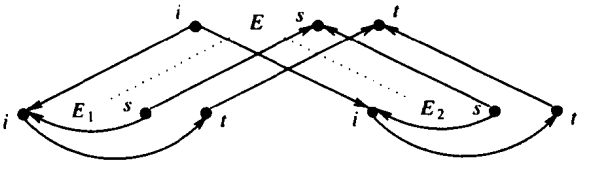


图5-61 产生式的扩充依赖图

5.10.3 环形检测

如果某个分析树的依赖图中存在环路，则称该语法制导定义是环形的。环形语法制导定义是不规则且无意义的。环上的任何属性值都无法计算。计算属性上的偏序关系（它可以保证一个制导定义是强无环的）与检测一个制导定义是否是环形的密切相关。因此，我们首先讨论环形检测问题。

例5.32 在下面的语法制导定义中，A 的属性间的路径依赖于所使用的产生式。如果使用

$A \to 1$，$A.s$ 就依赖于 $A.i$；否则，就没有该依赖关系。因此，为完整地分析可能的依赖信息，我们必须考虑非终结符属性上的偏序关系集合。

产生式	语义规则
$S \to A$	$A.i := c$
$A \to 1$	$A.s := f(A.i)$
$A \to 2$	$A.s := d$

图5-62中算法的基本思想为：用无环有向图表示偏序关系。给定一个产生式右部符号属性的 dag，我们可以用下述方法确定其左部符号属性的 dag。

```
for 文法符号X do
    𝒫(X) 中有一个带有X的属性的图，图中没有边;
repeat
    change := false;
    for 产生式 p given by A → X1X2 ··· Xk do begin
        for dags G1∈𝒫(X1), ..., Gk∈𝒫(Xk) do begin
            D := Dp;
            for Gj 中的边 b → c, 1≤j≤k do
                在D中添加一条Xj的属性b和c之间的边;
            if D有环路 then
                环形测试失败
            else begin
                G := 一个新的图，其节点为A的属性，没有边;
                for A的每对属性b和c do
                    if G中有一条从b到c的路径then
                        将边 b → c 添加到G中;
                if G还没有在𝒫(A)中then begin
                    将G加入到𝒫(A)中;
                    change := true
                end
            end
        end
    end
until change = false
```

图5-62 环形检测算法

335 令产生式 p 为 $A \to X_1X_2 \cdots X_k$，其依赖图为 D_p，且 D_j 为 X_j 的 dag，$1 \leqslant j \leqslant k$。$D_j$ 中的每条边 $b \to a$ 都被暂时加入依赖图 D_p 中。如果扩充后的结果图中有环路，则该语法制导定义就是环形的。否则，从结果图中的路径信息可以得出该产生式左部属性上的 dag，将此dag加入$\mathcal{F}(A)$中。

对任何文法符号X，图5-62的环形检测算法的时间复杂性随集合$\mathcal{F}(X)$中图的个数呈指数增长。存在多项式时间内不能完成环形检测的语法制导定义。

如果一个语法制导定义是强无环的，我们就可以将图5-62的算法转换成一个更有效的算法，方法为：对每一个 X，我们不是维护图$\mathcal{F}(X)$的集族，而是将此集族中的信息汇总为一个图 $F(X)$ 来保存。注意，对 X 的属性，$\mathcal{F}(X)$中的每一个图都有相同的节点，只是边有可能不同。如果$\mathcal{F}(X)$的任一图中存在一条从 $X.b$ 到 $X.c$ 的边，则 $F(X)$ 中也存在同样的边。$F(X)$ 表示 X 属性间依赖关系的“最坏估计”。所以，如果 $F(X)$是无环的，就能保证语法制导定义是无环的。但反过来不一定成立，即如果 $F(X)$ 中存在环，语法制导定义却不一定是环形的。

如果成功的话，修改后的环形检测算法可以为每个 X 构造无环图 $F(X)$。用这些图可以为语法制导定义构造一个属性计算程序，其方法可以从例5.30直接衍生出来。计算综合属性 $X.s$ 的函数将且只将在 $F(X)$ 中先于 s 的所有继承属性作为参数。节点 n 处调用的函数调用其子节点处其他计算所需综合属性的函数。计算这些属性的子程序是为它们所需要的继承属性所传递的值。事实上，强无环检测的成功保证了这些继承属性是可被计算的。

练习

5.1 根据图5-2中的语法制导定义，为输入表达式(4*7+1)*2构造注释分析树。

5.2 根据

a) 图5-9的语法制导定义。

b) 图5-28的翻译模式。

为表达式((a)+(b))构造分析树和语法树。

5.3 构造下述表达式的无环有向图（dag），并用值编号标识各个子表达式，假设+是左结合的：

```
a+a+(a+a+a+(a+a+a+a))
```

* 5.4 试给出将中缀表达式翻译成无多余括号的中缀表达式的语法制导定义。例如，因为+和*都是左结合的，所以((a*(b+c))*(d)可以写成a*(b+c)*d。 336

5.5 对于将算术运算符+、*应用到变量x和常数上所形成的求导表达式，如x*(3*x+x*x)，试给出其语法制导定义。假设对结果不需化简，那么3*x将翻译成3*1+0*x。

5.6 下列文法产生的表达式是对整常数和实常数应用运算符+所形成的。只有两个整数相加时，结果才是整型，否则就是实型。

$E \rightarrow E + T \mid T$
$T \rightarrow \textbf{num} \ . \ \textbf{num} \mid \textbf{num}$

a) 试给出确定每个子表达式类型的语法制导定义。

b) 扩充(a)的语法制导定义使之既可以确定类型，又能把表达式翻译为后缀表示。使用一元运算符 **inttoreal** 将整型值转换成实型值，以使得后缀式中+的两个运算对象具有相同的类型。

5.7 扩充图5-22的语法制导定义，使之除了可以记住盒子的高度外，还能记住其宽度。假设终结符 **text** 具有综合属性 w，它给出正文的标准宽度。

5.8 令综合属性*val*给出下列文法中 S 产生的二进制数的值。例如，输入101.101时，$S.val = 5.625$

$S \rightarrow L \ . \ L \mid L$
$L \rightarrow L \ B \mid B$
$B \rightarrow 0 \mid 1$

a) 试用综合属性确定 $S.val$。

b) 试用语法制导定义确定 $S.val$。在该定义中，B 只有综合属性 c，它给出由 B 产生的位对最终值的贡献。例如，101.101的第一位和最后一位对值5.625的贡献分别为4和0.125。

5.9 重写例5.3中语法制导定义的基础文法，使得类型信息只需用综合属性进行传播。

* 5.10 当下列文法产生的语句被翻译成抽象机器代码时，break 语句翻译为一个跳转指令，它跳转到最近的 while 循环外。为简单起见，我们用终结符 **expr** 表示表达式，用终结符 **other** 表示其他类型的语句。这些终结符都具有综合属性 *code*，表示其翻译后的代码。 337

$S \rightarrow \textbf{while expr do begin} \ S \ \textbf{end}$
$\mid \ S \ ; \ S$
$\mid \ \textbf{break}$
$\mid \ \textbf{other}$

试给出一个语法制导定义，将语句翻译成2.8节的栈式机器代码。要确保嵌套while语句中 break 的正确翻译。

5.11 试从练习5.6(a)和练习5.6(b)中的语法制导定义中消除左递归。

5.12 由下列文法产生的表达式中可能包括赋值号：

$S \rightarrow E$
$E \rightarrow E$:= $E \mid E$ + $E \mid$ (E) $\mid$ **id**

表达式的语义和C语言一样，即表达式 b := c 将 c 的值赋给 b，而 a := (b := c) 把 c 的值赋给 b 然后再赋给 a。

a) 试构造一个语法制导定义，检查赋值表达式的左部是否为左值。用非终结符E 的继承属性 *side* 来指出 E 所产生的表达式是出现在赋值号的左部还是右部。

b) 扩充(a)中的语法制导定义使其在检查输入时生成2.8节栈式机器的中间代码。

5.13 重写练习5.12中的基础文法，使之能对子表达式进行分组，把:=的子表达式分到左边，把+的子表达式分到右边。

a) 试构造一个模拟练习5.12(b)中语法制导定义的翻译模式。

b) 修改(a)中的翻译模式，使之能够按指令编号递增地将代码输出到文件中。

5.14 试给出一个翻译模式，以检查相同的标识符是否不会在标识符表中出现两次。

5.15 假设声明语句是由下列文法产生的：

$D \rightarrow$ **id** L
$L \rightarrow$, **id** $L \mid$: T
$T \rightarrow$ **integer** | **real**

338

a) 试构造一个翻译模式，像例5.3那样，将每个标识符的类型填入符号表中。

b) 根据(a)的翻译模式构造一个预测翻译器。

5.16 下面的文法是图5-22中基础文法的无二义性版本，其中括号{}只是用来为盒子分组，在翻译过程中将被删掉。

$S \rightarrow L$
$L \rightarrow L\ B \mid B$
$B \rightarrow B$ **sub** $F \mid F$
$F \rightarrow$ { L } $\mid$ **text**

a) 试改写图5-22中的语法制导定义使之适用于上面的文法。

b) 试将(a)中的语法制导定义转换成翻译模式。

* 5.17 扩充5.5节中消除左递归的变换方法，使之允许(5-2)中的非终结符A具有：

a) 由复制规则定义的继承属性。

b) 继承属性。

5.18 从练习5.16(b)的翻译模式中消除左递归。

* 5.19 设有L属性定义，其基础文法或者是LL(1)的，或者是一个能解决其二义性并能构造预测语法分析器的文法。试证明可以将继承属性和综合属性放在由预测语法分析表所驱动的自顶向下语法分析器的分析栈中。

* 5.20 试证明在LL(1)文法的任何位置加上惟一的标记非终结符后，结果文法都将是LR(1)文法。

5.21 考虑对LR(1)文法$L \rightarrow Lb \mid a$的如下改造：

$L \rightarrow MLb \mid a$
$M \rightarrow \epsilon$

a) 在输入串 *abbb* 的分析树中，自底向上语法分析器应用产生式的顺序是什么？

* b) 证明修改后的文法不是LR(1)文法。

* 5.22 证明在基于图5-36的翻译模式中，无论何时归约 B 的右部，分析栈中继承属性$B.ps$的值总是紧挨在该产生式右部的下边。

5.23 算法5.3用于完成带有继承属性的自底向上语法分析和翻译，它使用标记非终结符在分析栈中预知的位置上保存继承属性的值。如果该值存放在与分析栈不同的栈中，那么所需标记非终结符的数量将会减少。 339

a) 试将图5-36中的语法制导定义转换成翻译模式。

b) 修改(a)中构造的翻译模式，使得继承属性 *ps* 的值出现在另一个栈中。消除标记非终结符 M。

* 5.24 像练习5.23中那样，考虑语法分析过程中的翻译。S. C. Johnson 提出如下方法来模拟一个分离的继承属性栈，它需要使用标记和一个与每个继承属性对应的全局变量。在下面的产生式中，第一个动作将值 v 压入栈 i 中，而第二个动作将它从栈中弹出：

$A \rightarrow \alpha \ \{ push(v, i) \} \ \beta \ \{ pop(i) \}$

我们可以通过下面的产生式来模拟栈 i，它使用了一个全局变量 g 和一个带有综合属性 s 的标记非终结符 M：

$A \rightarrow \alpha \ M \ \beta \ \{ g := M.s \}$
$M \rightarrow \epsilon \ \{ M.s := g; \ g := v \}$

a) 试将该变换应用到练习5.23(b)中的翻译模式中，用对全局变量的引用来替换对另一个栈栈顶的引用。

b) 试证明用(a)中构造的翻译模式来计算文法开始符号的综合属性可得到和练习5.23(b)相同的值。

5.25 应用5.8节中的方法，用一个布尔变量实现练习5.12(b)的翻译模式中所有的 *E.side* 属性。

5.26 修改例5.26深度优先遍历中的栈的使用方法，使得该栈中的值与保存在例5.19中分析栈中的值一一对应。

参考文献注释

Irons[1961]中阐述了如何用综合属性来表示语言的翻译。Samelson and Bauer[1960]和Brooker and Morris[1962]中讨论了调用语义动作的语法分析器的设计思想。Knuth[1968]中叙述了继承属性、依赖图以及强无环检测等内容，同时在该文的修订版中提出了环形检测的概念和方法。该文中的扩充例子使用了全局属性训练过的副作用，此全局属性和分析树树根相关联。如果属性可以是函数，那么继承属性就可以被删除。像指称语义学中所做的那样，我们可以将一个非终结符与一个从继承属性到综合属性的函数联系起来。这些结论来自Mayoh[1981]。 340

在语法制导编辑中，语义规则中的副作用是不受欢迎的。像Reps[1984]中所讨论的那样，假设有一个从属性文法产生的源语言编辑器，考虑对源程序的一次编辑修改，它将导致程序分析树中的部分被删掉。只要不存在副作用，修改后的程序的属性值就可以被重新计算。

Ershov[1958]使用散列法来保存公共子表达式。

Lewis, Rosenkrantz, and Stearns[1974]中提出了在语法分析过程中进行翻译的L属性文法定义。Bochmann[1976]中把类似的对属性依赖关系的限制应用到每一个从左到右的深度优先遍历中。Koster[1971]中介绍了和L属性文法相关的词缀文法。Koskimies and Räihä[1983]中提出了对L属性文法的限制来控制对全局属性的访问。

Bochmann and Ward[1978]中描述了一种机械式的预测翻译器构造方法，其思想与算法5.2类似。自顶向下的语法分析可以给翻译带来更大的灵活性这种说法在Brosgol[1974]中已被证明是错误的，在该文献中提出基于LL(1)文法的翻译模式可以在LR(1)分析中被模拟。Watt[1977]中使用标记非终结符来保证在自底向上语法分析过程中继承属性的值能出现在栈中。Purdom and Brown[1980]中考虑了在不失去LR(1)特性的情况下把标记非终结符安全地插入到产生式右部相应的位置上（参见练习5.21）。仅需要由复制规则定义的继承属性不足以保证在自底向上语法分析过程中可以完成对属性的计算，因此在Tarhio[1982]中给出了充分的语义规则条件。Jones and Madsen[1980]中给出了一个能在LR(1)分析过程中进行计算的属性的特征（根据语法分析器的状态）。作为一个不能在语法分析过程进行翻译的例子，Giegerich and Wilhelm[1978]中考虑了布尔表达式的代码生成。我们将会看到8.6节中的回填不能用在这种问题上，因此一遍完整的第二遍扫描是没有必要的。

从Fang[1972]中的FOLDS开始，人们已经开发了许多种用于实现语法制导定义的工具，但很少有得到广泛应用的。Lorho[1977]中提出的DELTA在编译时构造依赖图，它通过保存属性的生存期和消除复制规则来节省空间。基于属性计算方法的分析树在Kennedy and Ramanathan[1979]和Cohen and Harry[1979]中进行了讨论。

Engelfriet[1984]对属性计算方法进行了综述，它的姊妹篇Courcelle[1984]则综述了其理论根据。Räihä et al. [1983]中描述的HLP实现了交互式的深度优先遍历，正如Jazayeri and Walter[1975]中所讨论的那样。Farrow[1984]中的LINGUIST也实现了交互式的遍历。Ganzinger et al. [1982]中指出了MUG可以允许由节点产生式确定其子节点被访问的顺序。Kastens, Hutt,
341 and Zimmerman[1982]中的GAG允许重复访问一个节点的子节点。GAG实现了Kastens[1980]中定义的一类有序属性文法。重复访问的思想最早出现在Kennedy and Warren[1976]中，它构造了可以计算更大的一类强无环文法的计算程序。如果在随后的一次访问过程中不再需要属性的值，Kennedy 和 Warren提出的方法可以通过把这些属性值保存在栈中来节省空间，Saarinen[1978]中对这一方法进行了改进。Jourdan[1984]中描述的实现方法可以为这类文法构造递归计算程序。Katayama[1984]中也构造了一个递归计算程序。Madsen[1980]在NEATS中考虑了一种不同的方法，它为表示属性值的表达式构造了一个dag。

在编译器构造时对依赖关系的分析可以节省编译时的时间和空间。环形检测是一个典型的分析问题。Jazayeri, Ogden, and Rounds[1975]中证明了环形检测需要的时间随文法的大小呈指数增长。在Lorho and Pair [1975]、Räihä and Saarinen[1982]和Deransart Jourdan, and Lorho [1984]中分别给出了环形检测的改进算法。

朴素的计算对空间的需求带动了保存空间技术的发展。Marill[1962]中以独特的内容描述了5.8节中提到的把属性值分配到寄存器中的算法。寻找依赖图的拓扑顺序使得寄存器被使用的个数最少的问题在Sethi[1975]中被证明是一个NP完全问题。Räihä[1981]和Jazayeri and Pozefsky [1981]中给出了多遍计算中对生存期的编译时分析。Branquart et al. [1976]中提出在遍历时使用分离的栈来保存综合属性和继承属性。GAG执行生存期分析并把属性值放在了所需要的全局变量、栈和分析树节点中。在Farrow and Yellin [1984]中对GAG和LINGUIST使用到
342 的两种节省空间技术进行了比较。

第6章 类型检查

编译器必须检查源程序是否符合源语言规定的语法和语义要求。这种检查称为静态检查（以区别于在目标程序运行时进行的动态检查），它检测并报告程序中某些类型的错误。静态检查的例子包括：

1. 类型检查。如果操作符作用于不相容的操作数，编译器应该报错；例如，将数组变量和函数变量相加。

2. 控制流检查。引起控制流从某个结构中跳转出来的语句必须能够决定控制流转向的目标地址。例如，C语言中的 break 语句可以使控制从 while、for 或 switch 等语句所确定的最小“范围”中跳出，如果不存在这样的“范围”语句，就会发生错误。

3. 惟一性检查。有时，某个对象只能被定义一次。比如，Pascal 语言中的标识符就只能被声明一次；而case语句中的标号要求互不相同，标量类型中的元素也不能重复。

4. 与名字相关的检查。有时，要求同一名字在特定位置出现两次或多次。例如，Ada 语言中循环结构（或程序块）的名字可能出现在该结构的开始位置及结束位置。编译器必须检查这两个地方是否使用了同一名字。

本章将集中讨论类型检查。正如上面的例子所指出的，大部分静态检查都是一些简单的工作，可以使用上一章的技术来实现。其中一些工作可以并入其他活动中。例如，当我们将名字信息填入符号表时，即可对该名字的惟一性进行检查。许多 Pascal 编译器将静态检查和中间代码生成同语法分析结合在一起。对于更复杂的结构，如 Ada 中的一些结构，将类型检查作为语法分析和中间代码生成之间单独的一遍会更方便一些，如图6-1所示。

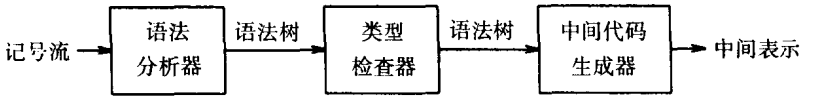


图6-1 类型检查器的位置

类型检查器将验证结构的类型是否与上下文所期望的类型相匹配。例如，Pascal 内置的算术运算符 mod 要求整型操作数，所以类型检查器必须检查 mod 的操作数是否为整型。同样地，类型检查器必须能够验证指针地址访问只作用于指针，下标只作用于数组，用户自定义的函数只能用于有正确参数个数及参数类型等情况。6.2节给出了一个简单的类型检查器，6.3节讨论类型的表示以及两个类型匹配条件的问题。

代码生成可能需要类型检查器所收集的信息。例如，像“+”这样的算术运算符通常作用于整型或实型，也可用于其他类型，所以必须检查“+”的上下文以确定其具体含义。一个符号在不同的上下文中可能代表不同的操作，我们称其为“重载”。重载可能伴随有强制类型转换，以便编译器把操作数转化为上下文所期望的类型。

与重载不同的另一个概念是“多态”。多态函数的函数体可以用不同类型的参数来执行。本章最后将给出一个推断多态函数类型的统一的算法。

6.1 类型系统

类型检查器的设计基于语言中语法结构的信息、类型的概念以及语言结构的类型指派规则。以下内容分别摘录自 Pascal 报告和 C 参考手册，编译器编写者可能必须以此为起点。

- “如果算术运算符加、减和乘的两个操作数都是整型，则结果类型亦是整型。”
- “一元运算符&返回指向操作数所引用对象的指针。如果操作数类型是‘…’，结果的类型就是‘指向…的指针’。”

343 ~ 344

上面的摘录暗含了这样的思想：任何表达式都有一个与之相关的类型，而且类型是有结构的，如类型“指向…的指针”就是从“…”所指的类型那里构造得来的。

Pascal 和 C 的类型都分为基本类型和构造类型两种。基本类型是原子类型，对程序员而言没有内部结构。在 Pascal 中，基本类型包括布尔型、字符型、整型以及实型。另外，子界类型（如`1..10`）和枚举类型（如 `{violet, indigo, blue, green, yellow, orange, red}`）也可以看成是基本类型。Pascal 允许程序员利用基本类型和其他构造类型来构造新类型，例如数组、记录和集合等。此外，指针和函数也可以看作构造类型。

6.1.1 类型表达式

一个语言结构的类型可以用“类型表达式”来表示。在这里，基本类型是类型表达式，由称为*类型构造符*的操作符作用于类型表达式而形成的表达式也是类型表达式。基本类型集和构造符依赖于被检查的语言。

本章采用下述*类型表达式*的定义：

1. 基本类型是类型表达式。基本类型包括 *boolean*、*char*、*integer* 以及 *real*。另外，有一个特殊的基本类型 *type_error*，它用来标志类型检查中的错误。最后，基本类型 *void* 表示“没有值”，它允许对语句进行检查。

2. 因为可以为类型表达式命名，所以类型名也是类型表达式。应用类型名的例子见下面的 3 中的 c)。包含名字的类型表达式将在6.3节讨论。

3. 作用于类型表达式的类型构造器仍是类型表达式。构造器包括：

a) *数组*。如果 T 是类型表达式，那么 *array*（I, T）就是表示元素类型为 T、下标集为 I 的数组类型的类型表达式。其中 I 通常是一个整数区间。例如，Pascal 中的声明

```
var A: array[1..10] of integer;
```

将类型表达式 *array*(1..10, *integer*)和 A 联系起来。

b) *乘积*。如果 T_1 和 T_2 是类型表达式，则其笛卡儿积 $T_1 \times T_2$ 也是类型表达式。假定 × 是左结合的。

c) *记录*。记录和乘积的区别在于记录中的域有名字。类型构造符 *record* 将作用于由域名和域类型组成的元组。（从技术上来说，域名应是类型构造符的一部分，但将域名和其相关类型联系在一起会更方便一些。在第8章，类型构造符 *record* 作用在指向以域名为表项的符号表的指针之上。）例如，Pascal 程序段

345

```
type row = record
               address: integer;
               lexeme: array [1..15] of char
           end;
var  table: array [1..101] of row;
```

声明了表示下列类型表达式的类型名 row：

record((**address** × *integer*) × (**lexeme** × *array*(1..15, *char*)))

变量 table 是该类型的记录数组。

d) 指针。如果 T 是类型表达式，那么 *pointer*（T）也是类型表达式，表示“指向类型为 T 的对象的指针”。例如，Pascal 中的声明

```
var p: ↑ row
```

声明变量 p 具有 *pointer*（row）类型。

e) 函数。从数学的角度看，函数将一个集合（定义域）中的元素映射到另一集合（值域）上。可以将程序设计语言中的函数看成是从定义域类型 D 到值域类型 R 的映射。这种函数的类型可以用类型表达式 $D \rightarrow R$ 来表示。例如，Pascal 内部函数 mod，其定义域类型为 *int* × *int*，即一对儿整数，值域类型为 *int*。所以说 mod 的类型为：[⊖]

int × *int* → *int*

另一个例子是如下的 Pascal 声明：

```
function f(a, b: char) : ↑ integer; ···
```

f 定义域类型为 *char* × *char*，值域类型为 *pointer*（*integer*）。所以函数 f 的类型可以用下面的类型表达式表示：

char × *char* → *pointer*(*integer*)

通常，出于具体实现上的原因（下一章讨论），可能对函数的返回类型有一些限制，比如，函数的返回类型不能为数组类型或函数类型。但是，也有些语言允许函数返回任意的类型，Lisp 就是最好的例子。例如，我们可以定义一个如下类型的函数 g： 346

(*integer* → *integer*) → (*integer* → *integer*)

即，函数 g 以将整数映射到整数的函数作为参数并输出同种类型的另一个函数作为结果。

4. 类型表达式可以包含其值为类型表达式的变量。类型变量将在6.6节介绍。

可以用图方便地表示类型表达式。使用5.2节中的语法制导方法，可以为类型表达式构造一棵树或dag，其内节点表示类型构造符，叶节点表示基本类型、类型名以及类型变量（见图6-2）。6.3节给出了一个类型表达式表示的例子，该例子已经被用在了编译器中。



图6-2 表示*char* × *char*→*pointer* (*integer*) 的树和dag

6.1.2 类型系统

*类型系统*是将类型表达式指派到程序各部分的一组规则。类型检查器用来实现类型系统。本章的类型系统采用语法制导方式来说明，所以使用上一章的技术可以很容易地实现。

同一语言的不同编译器或处理器可能使用不同的类型系统。例如，在 Pascal 语言中，数组的类型包括数组的下标集合，所以以数组为参数的函数只能用于具有同样下标集合的数组。但

⊖ 假设 × 的优先级高于→，所以 *int* × *int* → *int* 与（*int* × *int*）→*int* 是相同的。→还是右结合的。

是，许多 Pascal 编译器在传递数组参数时，允许不说明其下标集合。这些编译器使用的类型系统就不同于 Pascal 语言定义中的类型系统。类似地，在 UNIX 系统中，命令 `lint` 检查 C 程序中的某些错误，它使用的类型系统比 C 编译器使用的类型系统更加细致。

6.1.3 静态和动态类型检查

由编译器完成的检查称为静态检查，而在目标程序运行时完成的检查则称为动态检查。原
347 则上，如果目标代码将每个元素的类型和其值保存在一起，则任何检查都可以动态完成。

*健全的*类型系统能静态地确定在目标程序运行时不会发生错误，因此不需动态检查。也就是说，如果一个健全的类型系统将一个不是 *type_error* 的类型指派给程序的一部分，那么这部分程序的目标代码在运行时就不会发生类型错误。如果某一语言的编译器能保证它所接受的程序不会在运行时发生类型错误，则称此语言是*强类型语言*。

实际上，有些检查只能动态完成。例如，若首先声明下列变量：

```
table: array[0..255] of char;
i: integer
```

然后再计算 `table[i]`，通常编译器不能保证执行期间 `i` 的值总是在0到255之间。[⊖]

6.1.4 错误恢复

由于类型检查具有捕捉程序中错误的潜力，所以，对类型检查器来说，在发现错误时执行一些有意义的操作就非常重要。编译器至少必须能报告错误的性质和位置，而且希望类型检查器能从错误中恢复，以便检查剩下的输入。因为错误处理会影响类型检查的规则，所以一开始就必须将其正确地设计进类型系统中，这些规则必须准备处理各种错误。

包含出错处理的类型系统可能比说明正确程序所需的类型系统大得多。例如，一旦出现错误，我们可能不知道这一错误程序片段的类型。为处理这一缺少信息的情况，我们需要使用类似于不必先声明即可使用标识符的技术。6.6节讨论的类型变量可用于保证未声明或明显误声明标识符的使用一致性。

6.2 一个简单的类型检查器的说明

本节说明了一个简单语言的类型检查器，在该语言中，每个标识符使用前必须先声明。该类型检查器是一个翻译模式，可以从子表达式的类型合成表达式的类型，能够处理数组、指针、
348 语句以及函数。

6.2.1 一种简单语言

图6-3的文法产生由非终结符 P 表示的程序，该程序由一系列声明 D 和随后的一个表达式 E 组成。

用图6-3中的文法可以产生如下程序段：

```
key: integer;
key mod 1999
```

$P \rightarrow D\ ;\ E$
$D \rightarrow D\ ;\ D \mid \mathbf{id}\ :\ T$
$T \rightarrow \mathbf{char} \mid \mathbf{integer} \mid \mathbf{array}\ [\ \mathbf{num}\ ]\ \mathbf{of}\ T \mid \uparrow T$
$E \rightarrow \mathbf{literal} \mid \mathbf{num} \mid \mathbf{id} \mid E\ \mathbf{mod}\ E \mid E\ [\ E\ ] \mid E\uparrow$

图6-3 源语言文法

在讨论表达式以前，让我们先考虑语言中的类型。语言本身有两种基本类型 *char* 和 *integer*，第三种基本类型 *type_error* 用于报错。为简单起见，我们假定所有数组的下标都从1

⊖ 类似于第10章讨论的数据流分析技术可以用来推断 `i` 是否越界。但是没有哪种技术能在任何情况下都做出正确的判断。

开始。例如，

```
array [256] of char
```

导致类型表达式 *array*(1..256, *char*)，它是由作用于子界1..256和类型 *char* 上的构造符 *array* 组成的。同在 Pascal 中一样，声明中的前缀操作符↑建立了一种指针类型，所以

```
↑ integer
```

导致类型表达式 *pointer*(*integer*)，它是由作用于 *interger* 类型上的构造符 *pointer* 组成的。

在图6-4的翻译模式中，和产生式 $D \rightarrow$ **id**: T 相关联的动作将类型存入标识符的符号表项中。动作 *addtype* (**id**.*entry*, *T.type*)的参数为综合属性 *entry*（它指向符号表中的 **id** 表项）和由非终结符 T 的综合属性 *type* 表示的类型表达式。

如果 T 产生 **char** 或 **integer** 类型，那么 *T.type* 就被定义为 *char* 或 *integer*。如果 T 产生数组，则其上界就从记号 **num** 的属性 *val* 中得到，它给出 **num** 所代表的整数。假定数组的下界为1，此时类型构造符 *array* 被应用在子界1..**num**.*val* 和相应元素类型上。

在$P \rightarrow D$; E 的右部，D出现在 E 之前，这就保证了所有已声明标识符的类型在 E 所产生的表达式被检查之前都已被保存起来（见第5章）。事实上，通过适当地修改图6-3中的文法，可以在自顶向下或自底向上的语法分析过程中实现本节的翻译模式。 349

$P \rightarrow D$; E	
$D \rightarrow D$; D	
$D \rightarrow$ **id** : T	{ *addtype*(**id**.*entry*, *T.type*) }
$T \rightarrow$ **char**	{ *T.type* := *char* }
$T \rightarrow$ **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*(1..**num**.*val*, T_1.*type*) }

图6-4 翻译模式中保存标识符类型的部分

6.2.2 表达式的类型检查

在下面的规则中，E 的综合属性 *type* 定义了由 E 所产生的表达式的类型表达式（由类型系统分配）。下面的语义规则指出由记号 **literal** 和 **num** 表示的常量分别具有类型 *char* 和 *integer*：

$E \rightarrow$ **literal**	{ *E.type* := *char* }
$E \rightarrow$ **num**	{ *E.type* := *integer* }

我们使用函数 *lookup*(*e*) 取出保存在由 *e* 所指向的符号表项中的类型。当标识符出现在表达式中时，从符号表中取出其类型并赋给属性 *type*：

$E \rightarrow$ **id**	{ *E.type* := *lookup*(**id**.*entry*) }

通过将 mod 操作符应用于两个类型为 *integer* 的子表达式上而形成的表达式的类型也是 *integer*，否则其类型就是 *type_error*。规则如下：

$E \rightarrow E_1$ **mod** E_2	{ *E.type* := **if** E_1.*type* = *integer* **and** E_2.*type* = *integer* **then** *integer* **else** *type_error* }

对于数组引用 $E_1[E_2]$，下标表达式 E_2 必须为 *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*。这里没有用到数组的下标集合 *s*。

$E \rightarrow E_1 [E_2]$ { $E.type$:= **if** $E_2.type$ = *integer* **and** $E_1.type$ = *array*(s, t) **then** t **else** *type_error* }

在表达式中，后缀操作符↑产生由其操作数所指向的对象。E↑的类型是指针 E 所指向的对象的类型 t：

$E \rightarrow E_1 \uparrow$ { $E.type$:= **if** $E_1.type$ = *pointer*(t) **then** t **else** *type_error* }

350 至于通过增加产生式及语义规则来允许表达式具有其他类型和操作的问题，我们留给读者去完成。例如，为允许标识符具有 *boolean* 类型，我们可在图6-3的文法中引入产生式 $T \rightarrow$ **boolean**。把<这样的比较操作符及 **and** 这样的逻辑操作符引进产生式 E，即可构造 *boolean* 类型的表达式。

6.2.3 语句的类型检查

因为像语句这样的语言结构通常都没有值，因而可以为其分配特殊的基本类型*void*。如果在语句中发现错误，则为其分配类型 *type_error*。

我们考虑的语句有赋值语句、条件语句和 while 语句，各语句之间用分号隔开。如果我们定义完整程序的产生式为 $P \rightarrow D; S$，并把图6-5中的产生式并入图6-3的产生式中，则程序就由声明及其后的语句组成。因为语句中可以含有表达式，所以仍需要上述检查表达式类型的规则。

$S \rightarrow$ **id** := E	{ $S.type$:= **if** **id**.$type$ = $E.type$ **then** *void* **else** *type_error* }
$S \rightarrow$ **if** E **then** S_1	{ $S.type$:= **if** $E.type$ = *boolean* **then** $S_1.type$ **else** *type_error* }
$S \rightarrow$ **while** E **do** S_1	{ $S.type$:= **if** $E.type$ = *boolean* **then** $S_1.type$ **else** *type_error* }
$S \rightarrow S_1 ; S_2$	{ $S.type$:= **if** $S_1.type$ = *void* **and** $S_2.type$ = *void* **then** *void* **else** *type_error* }

图6-5 检查语句类型的翻译模式

检查语句的规则如图6-5所示。第一条规则检查赋值语句的左边和右边是否具有相同的类型。[⊖]第二条和第三条规则指明条件语句和 while 语句中的表达式必须具有 *boolean* 类型。错误由图6-5中的最后一条规则来传播，因为只有当每个子语句都具有 *void* 类型时，语句序列的类型才是 *void*。在上述规则中，类型不匹配会产生 *type_error* 类型。当然，友好的类型检查器
351 还应报告类型不匹配的性质及其位置。

6.2.4 函数的类型检查

可用下面的产生式来产生一个函数：

$E \rightarrow E (E)$

其中，表达式是由一个表达式作用于另一个表达式而形成的。通过下面的产生式和动作可以将类型表达式和非终结符 T 关联起来，进而扩充了规则，使得声明中可以定义函数类型。

$T \rightarrow T_1 \text{ '}\rightarrow\text{' } T_2$ { $T.type$:= $T_1.type \rightarrow T_2.type$ }

⊖ 如果表达式允许出现在赋值语句的左部，则我们还必须区分左值和右值。例如，1:=2是不正确的，因为常数1不能被赋值。

其中，箭头两边加上引号是为了将它和用作产生式元符号的箭头区分开来，它表示函数构造符。

函数的类型检查规则为

$$E \rightarrow E_1\ (\ E_2\ ) \qquad \{\ E.type := \textbf{if}\ E_2.type = s\ \textbf{and}\ E_1.type = s \rightarrow t\ \textbf{then}\ t\ \textbf{else}\ type_error\ \}$$

这一规则指出，在将 E_1 作用于 E_2 所形成的表达式中，E_1 的类型必须是函数类型 $s \rightarrow t$，其中 s 为 E_2 的类型，t 为某一值域类型，$E_1(E_2)$ 的类型是 t。

许多和函数的类型检查有关的问题可以应用上述简单语法进行讨论。推广到具有多个参数的函数，可以通过构造参数的积类型来完成。注意，类型为 $T_1, \cdots, T_n$ 的 n 个参数可以看作类型为 $T_1 \times \cdots \times T_n$ 的单个参数。例如，可以用

```
root : (real → real) × real  →  real                    (6-1)
```

来声明函数 root。root 的参数是一个返回实数的函数和另一个实数，其返回值也是实数。这个声明的类 Pascal 语法是

```
function root (function f (real): real; x: real): real
```

(6-1)中的语法将函数类型的声明同其参数名分离开来。

6.3 类型表达式的等价

上一节的检查规则具有如下形式："**if**两个类型表达式相等**then**返回一个特定类型**else**返回 *type_error*。"因而精确地定义何时两个类型表达式是等价的非常重要。如果为类型表达式命名并在随后的类型表达式中使用这些名字，就会出现潜在的二义性。关键问题是类型表达式中的名字是代表它自己还是代表另一个类型表达式的缩写。

由于类型等价的概念和类型表示互相影响，所以我们将二者放在一起讨论。为了提高效率， 352
编译器采用了能快速确定类型等价的表示方法。由特定编译器实现的类型等价概念通常被解释成本节将要讨论的结构等价和名字等价。我们将根据类型表达式的图形表示来展开讨论。如图6-2所示，图中叶节点表示基本类型和类型名，内节点表示类型构造符。正如我们将要看到的，如果名字被看成是类型表达式的缩写，则递归地定义的类型将导致类型图中具有环路。

6.3.1 类型表达式的结构等价

只要类型表达式是由基本类型和类型构造符构造出来的，则两个类型表达式等价的直观想法就是*结构的等价*，即两个表达式或者是相同的基本类型，或是将同样的类型构造符作用于结构等价的类型而构成的。也就是说，两个类型表达式是结构等价的充分必要条件就是它们完全相同。例如，类型表达式 *integer* 只与类型 *integer* 等价，因为它们是相同的基本类型。类似地，*pointer*(*integer*)只与*pointer*(*integer*)等价，因为二者是将相同的构造符 *pointer* 作用于等价的类型的结果。如果我们使用算法5.1中的值编号方法来构造类型表达式的 dag 表示，则相同的类型表达式将表示为同一节点。

实际上，为了反映源语言的实际类型检查规则，常常需要修改结构等价的概念。例如，当传递数组变元时，我们可能不希望将数组的边界也作为其类型的一部分。

为测试修改后的等价概念，我们可以修改图6-6中测试结构等价的算法。假定类型构造符只有数组、乘积、指针和函数。该算法递归地比较类型表达式的结构而不检查是否有环，因此，可以将其用于类型表达式的树型表示或dag 表示。相同的类型表达式不必用dag的同一节点来表示。有环类型图中节点的结构等价可以用6.7节的算法来测试。

如果将图6-6中第4行、第5行的数组等价测试代码重写为

else if $s = array(s_1, s_2)$ **and** $t = array(t_1, t_2)$ **then**
    return $sequiv(s_2, t_2)$

则

$s = array(s_1, s_2)$
$t = array(t_1, t_2)$

中的边界 s_1 和 t_1 可以被忽略。

```
(1)  function sequiv(s, t): boolean;
     begin
(2)      if s和t是同样的基本类型 then
(3)          return true
(4)      else if s = array(s1, s2) and t = array(t1, t2) then
(5)          return sequiv(s1, t1) and sequiv(s2, t2)
(6)      else if s = s1×s2 and t = t1×t2 then
(7)          return sequiv(s1, t1) and sequiv(s2, t2)
(8)      else if s = pointer(s1) and t = pointer(t1) then
(9)          return sequiv(s1, t1)
(10)     else if s = s1 → s2 and t = t1 → t2 then
(11)         return sequiv(s1, t1) and sequiv(s2, t2)
         else
(12)         return false
     end
```

图6-6 测试两个类型表达式 s 和 t 的结构等价

某些情况下，我们可以为类型表达式找到一种比类型图紧凑得多的表示方法。在下例中，类型表达式中的某些信息被编码成一个二进制位序列，然后可以将其解释为一个整数。这种编码用不同的整数表示结构上不等价的类型表达式。可以先通过比较这些整数来测出结构不等价的类型表达式，只有当这些整数相同时才应用图6-6的算法，这样能加速结构等价测试过程。

例6.1 本例中的类型表达式的编码来自 D. M. Ritchie所编写的 C 编译器。Johnson[1979]描述的 C 编译器也使用这种编码。

考虑具有如下指针、函数和数组类型构造符的类型表达式：*pointer*(*t*)表示指向类型 *t* 的指针，*freturns*(*t*)表示返回类型为 *t* 的对象的函数，*array*(*t*)表示元素类型为 *t* 的数组（不定长）。注意，此处我们简化了数组和函数的类型构造符。虽然我们将跟踪数组元素的个数，但这一计数值不成为类型构造符 *array* 的一部分。类似地，构造符 *freturns* 的惟一操作数是函数的结果的类型，函数参数的类型将保存在其他地方。所以，该类型系统中结构等价的表达式在将图6-6中的测试应用于更详尽的类型系统时可能是结构不等价的。

353 ~ 354

由于这些构造符都是一元操作符，将这些构造符应用于基本类型上所形成的类型表达式就具有统一的结构。这种类型表达式的例子如下：

char
freturns(*char*)
pointer(*freturns*(*char*))
array(*pointer*(*freturns*(*char*)))

使用一种简单的编码方案，上面的每个表达式就都可以用一个二进制位序列来表示。因为只有三个类型构造符，所以使用两位即可为一个构造符编码，如下所示：

类型构造符	编码
pointer	01
array	10
freturns	11

在Johnson[1979]中，C 的基本类型采用4位编码。我们的4个基本类型编码如下：

基本类型	编码
boolean	0000
char	0001
integer	0010
real	0011

受限的类型表达式现在可以编码成二进制位序列。最右边4位是基本类型的编码。从右向左移，接下来的两位是应用于基本类型上的构造符编码，再下两位是再次应用的构造符编码，依此类推。举例如下：

类型表达式	编码
char	000000 0001
freturns(*char*)	000011 0001
pointer(*freturns*(*char*))	000111 0001
array(*pointer*(*freturns*(*char*)))	100111 0001

更详细的描述参见练习6.12。

除节省空间之外，这种表示方法还能反映出现在任何类型表达式中的构造符。两个不同的二进制序列表示不同的类型，因为这两个类型表达式要么是基本类型不同，要么是构造符不同。当然，由于数组的大小和函数的参数没有被表示出来，所以不同的类型也可能具有相同的二进制序列。

可以扩展本例的编码以包含记录类型。其思想为，在编码中将每个记录当作一个基本类型，而记录的各个域类型都用一个独立的二进制序列来编码。C 中的类型等价将在例6.4中做进一步研究。 □ 355

6.3.2 类型表达式的名字

在某些语言中可以给类型命名。如在 Pascal 程序段

```
type link = ↑ cell;
var  next : link;
     last : link;                    (6-2)
     p    : ↑ cell;
     q, r : ↑ cell;
```

中，标识符link被声明为类型 ↑cell的名字。此时会出现这样的问题：变量 next、last、p、q 和 r 的类型相同吗？令人吃惊的是，答案依赖于具体实现。这一问题的出现是因为当初的 Pascal 报告中没有“相同类型”这一概念。

为描述这种情况，我们允许对类型表达式进行命名，并允许这些名字出现在类型表达式中，而在此之前类型表达式中只有基本类型。例如，若 cell 是一个类型表达式的名字，那么 *pointer*(cell)也是类型表达式。暂时假定没有循环的类型表达式定义，即不会将 cell 定义为包含 cell 的类型表达式的名字。

当允许类型表达式中出现名字时，将伴随出现类型表达式等价的两个概念，这取决于如何看待名字。*名字等价*将每个类型名看作是可区分的类型，所以两个类型表达式当且仅当名字完全相同时才是名字等价的。而在*结构等价*的情况下，类型名将被它们所定义的类型表达式所代替，所以，如果替换所有的名字后，两个类型表达式结构上等价，那么它们就是结构等价的。

例6.2 下面给出了与声明(6-2)中各个变量相关联的类型表达式。

变量	类型表达式
next	link
last	link
p	*pointer*(cell)
q	*pointer*(cell)
r	*pointer*(cell)

在名字等价的情况下，变量 next 和 last 具有相同的类型，因为它们具有相同的关联类型表达式。变量 p、q 和 r 也具有相同的类型，但是 p 和 next 类型不同，因为它们的关联类型表达式不同。在结构等价的情况下，上面5个变量具有同样的类型，因为 link 是类型表达式 *pointer*(cell)的名字。 □

各种不同的语言在通过声明来联系标识符和类型时都使用一些规则，在解释这些规则时，
356 结构等价和名字等价是两个非常有用的概念。

例6.3 由于在 Pascal 的许多实现中把隐式类型名和每个声明的标识符联系起来，所以出现了混淆。如果声明中包含一个不是名字的类型表达式，那么就建立一个隐式类型名。每当变量声明中出现类型表达式时，也建立一个新的隐式类型名。

于是，在(6-2)中包含 p、q 和 r 的两个声明中，将为类型表达式建立隐式类型名。也就是说，这些声明处理起来就像下面这样：

```
type link = ↑ cell;
     np   = ↑ cell;
     nqr  = ↑ cell;
var  next : link;
     last : link;
     p    : np;
     q    : nqr;
     r    : nqr;
```

在此，引入了新的类型名 np 和 nqr。在名字等价的情况下，由于 next 和 last 是用相同的类型名声明的，所以我们认为它们具有等价的类型。同样，我们认为 q 和 r 也具有等价的类型，因为与它们相关联的隐式类型名相同。但是 p、q 和 next 不具有等价的类型，因为它们的类型名不相同。

典型的实现方法是构造一个类型图来表示类型。每当遇到类型构造符或基本类型时，就建立一个新的节点；每当遇到新的类型名时，就建立一个叶节点，并同时记下该名字所引用的那个类型表达式。用这种表示方法，如果两个类型表达式在类型图中用同一节点表示，则它们等价。图6-7给出了声明(6-2)的类型图，其中虚线表示变量和类型图中节点的联系。注意，类型名 cell 有3个父节点，它们
357 均被标记为 *pointer*。等号出现在类型名 link 和它所引用的类型图节点之间。 □

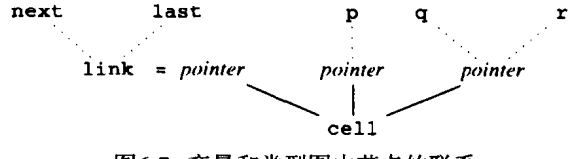


图6-7 变量和类型图中节点的联系

6.3.3 类型表示中的环

链表和树这样的基本数据结构通常都是递归定义的，如链表或者为空或者是由含有指向链表指针的单元组成。这样的数据结构通常用记录来实现，记录中含有指向同类型记录的指针。定义这样的记录类型时，类型名起着重要作用。

考虑某链表，它的每个单元包含某些整型信息和一个指向表中下一单元的指针。链和单元对应的类型名的 Pascal 声明如下：

```
type  link = ↑ cell;
      cell = record
                 info : integer;
                 next : link
             end;
```

注意，类型名 link 是根据 cell 定义的，而 cell 又是根据 link 定义的，所以它们是递归定义的。

如果在类型图中引入环，则递归定义中的类型名就可以被替换掉。如果用 *pointer*(cell) 替换 link，则将得到图6-8a中的 cell 类型表达式。再使用图6-8b中的环，就可以删除该类型图中 *record* 节点下出现的 cell。

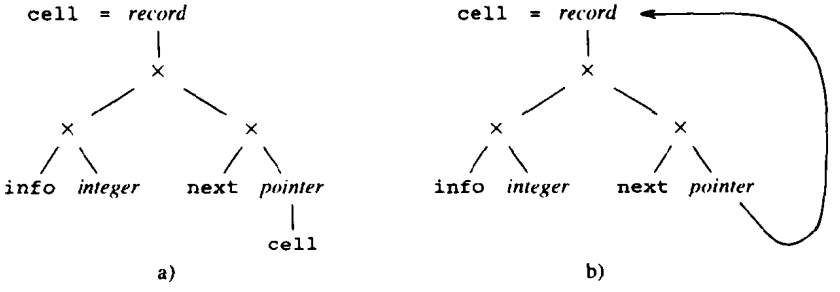


图6-8 递归定义的类型名cell

例6.4 C 语言通过对记录以外的其他类型使用结构等价来避免在类型图中出现环。C 中 cell 的声明为：

```
struct cell {
    int info;
    struct cell *next;
};
```

C 使用关键字 struct 而不是 record，而且名字 cell 成为该记录类型的一部分。实际上， 358
C 使用的就是图6-8a中的无环表示。

C 语言要求类型名在使用前要先声明，但在定义记录中的指针域时可使用未声明的记录类型。因此，所有潜在的环都源于指向记录的指针。由于记录的名字是其类型的一部分，此时进行结构等价的测试时，若遇到记录构造符，那么被比较的类型或者因为它们具有同样的命名记录类型而等价，或者不等价。 □

6.4 类型转换

考虑表达式 x+i，其中 x 是实型，i 是整型。因为整数和实数在计算机内的表示不同，且相应操作的机器指令也不同，因此对于上述加法，编译器需先对加法中的一个操作数进行类型转换，使得两个操作数具有相同的类型。

语言的定义会指出哪些转换是必须的。当把整数赋给实型变量（或相反）时，应将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，通常的转换是将整数变为实数，然后在两个实型对象上进行实数运算。可以使用类型检查器将这些转换操作插入源程序的中间表示中。例如，x+i 的后缀表示可能如下：

```
x i inttoreal real+
```

此时，**inttoreal** 操作符先将i从整数转换成实数，然后 **real+** 在两个操作数上执行实数加。

类型转换还常常出现在另一种情况下：一个符号在不同的上下文中具有不同的含义，我们将其称为重载，下一节将对重载进行详细讨论，但此处提到重载是因为类型转化常常伴随着重载。

强制类型转换

如果类型转换可以由编译器自动完成，则称其为*隐式转换*。隐式类型转换也叫做*强制类型转换*。许多语言要求隐式转换不丢失信息。例如，整数可以转换成实数，反之却不能。事实上，即使当实数转换成相同位数的整数时，也可能发生信息的丢失。

相反，如果转换必须由程序员写出，则称为*显式转换*。实际上，Ada 中的所有转换都是显式的。对类型检查器而言，显式转换就像是函数应用，因而不会导致新问题。

359 例如，在 Pascal 中，内置函数 `ord` 把字符映射为整数，函数 `chr` 则把整数映射为字符，因此这些转换都是显式的。而在 C 语言的算术表达式中，ASCII 字符会被强制转换（即隐式转换）为0到127之间的整数。

例6.5 考虑把算术运算符 **op** 作用在常数和标识符上所形成的表达式，如图6-9所示的文法。假设有两种类型：整型和实型，必要时可以将整数转换为实数。非终结符 E 的属性 *type* 可以是 *integer* 也可以是 *real*，类型检查规则如图6-9所示。如同6.2节那样，函数 *lookup*(*e*)返回保存在 *e* 所指向的符号表项中的类型。 □

产生式	语义规则
$E \rightarrow$ **num**	*E.type* := *integer*
$E \rightarrow$ **num . num**	*E.type* := *real*
$E \rightarrow$ **id**	*E.type* := *lookup*(**id**.*entry*)
$E \rightarrow E_1$ **op** E_2	*E.type* := **if** E_1.*type* = *integer* **and** E_2.*type* = *integer* **then** *integer* **else if** E_1.*type* = *integer* **and** E_2.*type* = *real* **then** *real* **else if** E_1.*type* = *real* **and** E_2.*type* = *integer* **then** *real* **else if** E_1.*type* = *real* **and** E_2.*type* = *real* **then** *real* **else** *type_error*

图6-9 从整型到实型强制类型转换的类型检查规则

常数的隐式转换通常在编译时即可完成，从而可以大大缩短目标程序的运行时间。在下面的代码段中，`X` 是一个实型数组，而且所有的元素均被初始化为1。用某种 Pascal 编译器编译后，Bentley[1982] 发现执行代码段

```
for I := 1 to N do X[I] := 1
```

需要48.4N微秒，而执行代码段

```
for I := 1 to N do X[I] := 1.0
```

需要5.4N微秒。两个程序段都是把1赋给实型数组的元素。但第一个代码段的目标代码（由编译器产生）包含将整数1转化为实数表示的运行时例程调用，所以更耗时。由于在编译时
360 就知道 `X` 是一个实型数组，所以更彻底的编译器应在编译时就将1转化成1.0。

6.5 函数和运算符的重载

在不同的上下文中具有不同含义的符号称为*重载符号*。在数学中，加法运算符 + 就是重载符号，因为当 A 和 B 分别是整数、实数、复数或矩阵时，$A+B$ 中的+具有不同的含义。在Ada语言中，括号 `()` 是重载符号，表达式 `A(I)` 可能是数组 `A` 的第 `I` 个元素，也可能是用参数 `I`

来调用函数 A，或者是表达式 I 到类型 A 的显式转换。

如果重载符号每次出现都能确定其惟一的含义，则称该重载是可解的。例如，如果 + 既可以表示整数加也可以表示实数加，则在表达式 x+(i+j) 中两次出现的+就可以表示不同形式的加，这将取决于 x、i 和 j 的类型。重载的解有时也称为操作符的识别，因为它是要确定操作符究竟表示哪一种操作。

在大多数语言中，算术运算符都是重载符号。但是，像+这样的算术操作符的重载只要察看其操作数即可解决。如何确定是使用整数加还是使用实数加有点类似于图6-9中 $E \rightarrow E_1$ **op** E_2 的语义规则，即 E 的类型是通过察看 E_1 和 E_2 的可能类型来确定的。

6.5.1 子表达式的可能类型的集合

正如下例所示，仅仅通过查看函数的参数并不总能解决重载。一个子表达式对应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来缩小这一集合，最终成为单个类型。

例6.6 在 Ada 中，操作符*的一个标准（即内部定义）解释是一个从一对整数到一个整数的函数。通过加入下面的声明，该操作符即可重载：

```
function "*" ( i, j : integer ) return complex;
function "*" ( x, y : complex ) return complex;
```

此时*可能的类型包括：

integer × *integer* → *integer*
integer × *integer* → *complex*
complex × *complex* → *complex*

如果2、3和5可能的类型只能是整型，那么对于上述声明，子表达式3*5可能是整型或复型，这取决于具体的上下文。如果完整的表达式是 2*(3*5)，那么3*5就是整型，因为*的两个操作数要么都是整型，要么都是复型。另一方面，如果完整的表达式是(3*5)*z，而且 z 被声明为复型，那么3*5就是复型。 □ 361

在6.2节中，我们假定每个表达式只有惟一的类型，所以函数的类型检查规则就是：

$E \rightarrow E_1 (E_2)$ { *E.type* := **if** E_2.*type* = s **and** E_1.*type* = $s \rightarrow t$ **then** t **else** *type_error* }

图6-10中将这一规则推广到类型集合。图中只有函数运算，表达式中其他操作符的检查规则是类似的。重载标识符可能有多个声明，所以我们假定符号表表项中可能包含可能类型的集合，该集合由 *lookup* 函数返回。文法开始非终结符 E' 产生完整的表达式，它的作用在下面说明。

产生式	语义规则
$E' \rightarrow E$	E'.*types* := *E.types*
$E \rightarrow$ **id**	*E.types* := *lookup*(**id**.*entry*)
$E \rightarrow E_1 (E_2)$	*E.types* := { t | { t | E_2. *types* 中存在一个s，使得 $s \rightarrow t$ 属于 E_1. *types* }

图6-10 确定表达式的可能类型的集合

如果用自然语言叙述的话，图6-10的第三条规则可以叙述为：如果 s 是 E_2 的一个类型，并且 E_1 的某个类型能把 s 映射到 t，那么 t 就是 $E_1(E_2)$ 的一个类型。函数的类型不匹配会导致集

合 *E.types* 为空，我们暂且将其作为通知类型错误的条件。

例6.7 本例除了解释图6-10以外，还说明了如何将该方法推广到其他结构中。特别是，考虑表达式3*5，令操作符*的声明如例6.6，即根据上下文*可以把一对整数映射到一个整数或一个复数。子表达式3*5的可能类型的集合如图6-11所示，其中 *i* 和 *c* 分别是 *integer* 和 *complex* 的缩写。

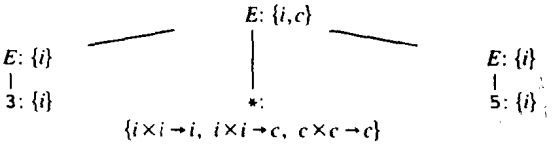


图6-11 表达式3*5的可能类型的集合

再次假设3和5的惟一可能类型为 *integer*。因此运算符*应用于一对整数。如果我们将这对整数看作一个单元，那么其类型为 *integer* × *integer*。在*的类型的集合中有两个作用于一对整数的函数，其中一个返回整数，另一个返回复数；因此根有两种类型：*integer* 和 *complex*。 □

362

6.5.2 缩小可能类型的集合

Ada 要求一个完整的表达式只有惟一的类型。我们可以根据上下文来确定惟一的类型，从而缩小每个子表达式的类型选择范围。如果该过程不能使每个子表达式的类型惟一，就宣布该表达式中存在类型错误。

在自顶向下地分析表达式之前，先仔细察看图6-10的规则所构造的 *E.types* 集合。可以证明 *E.types* 中的每个类型 *t* 都是一个可行的类型，即可以适当地选择 *E* 中标识符的重载类型以使 *E* 获得类型 *t*。由于 **id**.*types* 的每个元素都是可行的，所以标识符的声明满足这一性质。应用归纳法，考虑 *E* 为 $E_1(E_2)$时 *E.types* 中的类型 *t*。根据图6-10中的函数规则，对某个类型 *s*，*s* 必须在 E_2.*types* 中，并且类型 $s \to t$ 必须在 E_1.*types* 中，根据归纳可知，*s* 和 $s \to t$ 分别是 E_2 和 E_1 的可行类型，从而 *t* 就是 *E* 的可行类型。

可能存在多种达到同一可行类型的方法。例如，考虑表达式 f(x)，其中 f 可以具有类型 $a \to c$ 和 $b \to c$，x 可以具有类型 *a* 和 *b*。那么，f(x)具有类型 *c*，而 x 却有两种可能的类型 *a* 或 *b*。

将图6-10的语法制导定义加上确定 *E* 继承属性 *unique* 的语义规则可以得到图6-12的语法制导定义。*E* 的综合属性 *code* 将在以后讨论。

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' | {E_2. *types*中存在一个s, 使得$s \to s'$在E_1.*types*中} } $t := E.unique$ $S :=$ { s | $s \in E_2.types$ **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**' ':' $E.unique$)

图6-12 缩小表达式的类型的集合

因为 *E'* 产生了整个表达式，所以我们希望 *E'.types* 是一个包含单个类型 *t* 的集合。这个惟一的类型将由 *E.unique* 继承。基本类型 *type_error* 仍然用来报错。

如果函数 $E_1(E_2)$返回类型 t，我们就可以找到类型 s，它对变元 E_2 是可行的，同时，$s \rightarrow t$ 对该函数是可行的。图6-12中的语义规则中出现的集合 S 就是用来检查是否存在惟一的类型 s 满足这一性质的。

通过对表达式语法树的两次深度优先遍历即可实现图6-12中的语法制导定义。第一次遍历时，属性 *types* 自底向上综合。第二次遍历时，属性 *unique* 自顶向下传播，而且当我们从节点返回时，可以综合出 *code* 属性。在实际应用中，类型检查器可以简单地将一个惟一的类型附加到语法树的每个节点上。在图6-12中，我们生成后缀表示以说明中间代码是如何生成的。在后缀表示中，函数 *gen* 将一个类型附加到每个标识符以及 **apply** 操作符的实例上。

6.6 多态函数

普通函数执行时,其函数体中各语句的变元必须是固定类型；而对于多态函数，每次执行时，函数体中语句的变元可以是不同的类型。“多态”这一概念也可用在使用不同类型变元执行的代码段上，因此既存在多态函数，也存在多态运算符。

索引数组、应用函数和操纵指针等内置的运算符通常都是多态的，因为它们并不仅仅限于特定的类型。例如，C 语言参考手册中关于指针运算 & 的说明是：“如果操作数的类型是‘…’，那么结果的类型为指向‘…’的指针。”因为“…”可以代表任何类型，所以 C 语言的 & 运算符是多态的。

在 Ada 中，“类属”函数也是多态的，但 Ada 对多态性进行了限制。因为“类属”还被用来表示重载函数以及对函数变元的强制类型转换，所以我们将避免使用该术语。

本节旨在解决为带有多态函数的语言设计类型检查器时存在的问题。为处理多态性，我们扩展了类型表达式的集合，使其包括带有类型变量的表达式。类型变量的引入产生了一些与类型表达式等价有关的算法问题。 363 ~ 364

6.6.1 为什么要使用多态函数

多态函数便于实现那些操作数据结构但不必考虑其成员类型的算法。例如，使用多态函数，不必了解列表中元素的具体类型即可很容易地写出求列表长度的程序。

像 Pascal 这样的语言要求对函数变元的类型给出完整的说明，因此计算整数链表长度的函数不能用于实数链表。图6-13是求整数列表长度的 Pascal 代码。函数 `length` 顺着链表的 `next` 链前进，直至遇到 `nil` 链为止。尽管函数代码不以任何方式依赖于表中单元信息的类型，但 Pascal 要求在编写函数 `length` 时，必须声明 `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6-13 求列表长度的 Pascal 程序

在带有多态函数的语言中，如 ML(Milner[1984])，我们可以重写函数 `length`,使之适用于任何类型的列表，如图6-14所示。关键字 `fun` 表示 `length` 是一个递归函数。函数`null` 和 `tl` 是预定义的，`null` 测试表是否为空，而 `tl` 返回删除第一个元素后剩下的列表。按照图6-14所示的定义，

以下两次对函数 length 的调用都输出3：

```
length(["sun","mon","tue"]);
length([10,9,8]);
```

```
fun length(lptr) =
        if null(lptr) then 0
        else length(tl(lptr)) + 1;
```

图6-14 求列表长度的ML程序

第一次调用中，length 作用于字符串列表，第二次调用中则作用于整数列表。

6.6.2 类型变量

表示类型表达式的变量允许我们对未知类型进行讨论。在本节剩下的部分中，我们采用希腊字母 α，β，… 来表示类型表达式中的类型变量。

在不要求标识符先声明后引用的语言中，类型变量的一个重要应用就是检查标识符引用的一致性。类型变量代表未声明的标识符的类型。我们可以通过查看程序来获得未声明的标识符的引用情况。如果在某条语句中它被用作整型，而在另一语句中又被用作数组，就可以报告发生了引用不一致的错误。相反，如果变量总被用作整型，则在保证引用一致性的同时，也能推断出其类型必定是整型。

通过引用方式来确定一个语言结构的类型的问题称为*类型推断*。这一术语常被用来描述从函数体推断函数类型的问题。

例6.8 类型推断技术可用于 C 语言及 Pascal 语言等，在编译时填充程序中缺少的类型信息。图6-15中的代码段给出了一个过程 mlist，其参数 p 也是一个过程。通过观察过程 mlist 的第一行，我们只知道 p 是一个过程，而不能确定 p的参数个数和类型。C 和 Pascal 参考手册都允许这种不完整的类型声明。

```
type link ↑cell;

procedure mlist ( lptr : link; procedure p ) ;
begin
    while lptr <> nil do begin
        p(lptr);
        lptr := lptr↑.next
    end
end;
```

图6-15 带过程参数 p 的过程 mlist

过程 mlist 将参数 p 作用于链表的每个单元。例如，p 可用于给链表单元中的整型变量置初值或打印这些整型值。尽管没有指明参数 p 的具体类型，但是从表达式 p(lptr) 中 p 的使用方式我们可以推断出 p 的类型必定是：

$$link \rightarrow void$$

对 mlist 的任何调用，如果过程参数 p 不是这种类型，就报错。我们可以将过程看成是无返回值的函数，所以其结果类型就是 *void*。 □

类型推断技术和类型检查技术有很多共同点。两种情况都要处理包含类型变量的类型表达式。类型检查器可以用下例的推断原理来推断变量所代表的类型，这一应用将在本节的后续部分描述。

例6.9 我们可以推断出下面的伪码程序中多态函数 deref 的类型。函数 deref 与 Pascal 中的解除指针操作符 ↑ 具有相同的作用。

```
function deref(p);
begin
    return p↑
end;
```

遇到第一行代码

```
function deref(p);
```

时，我们对 p 的类型一无所知，所以让我们用类型变量 β 来代表其类型。根据定义，后缀操作符↑作用在一个指向对象的指针上，并返回该对象。因为↑操作符在表达式 p↑ 中作用于 p 上，所以 p 必定是指向某未知类型 α 的指针，于是我们得到

$$\beta = pointer(\alpha)$$

其中，α 是另一个类型变量。此外，表达式 p↑ 具有类型 α，因此函数 deref 的类型的类型表达式可以写成

对任何类型 α，$pointer(\alpha) \rightarrow \alpha$ (6-3)□

6.6.3 包含多态函数的语言

到目前为止，我们所说的多态函数是指该函数在多次执行时，其参数可具有“不同的类型”。关于多态函数可作用的类型的集合，我们可以用符号 ∀ 来表达其精确定义，符号 ∀ 意即“对任何类型”。所以，例6.9中函数 deref 的类型表达式可以写成：

365 ~ 367

$$\forall\alpha.\ pointer(\alpha) \rightarrow \alpha \tag{6-4}$$

图6-14中的多态函数 length 作用于元素为任何类型的列表，并返回一个整数，所以其类型表达式可以写成

$$\forall\alpha.\ list(\alpha) \rightarrow integer \tag{6-5}$$

这里 *list* 是一个类型构造符。如果不用符号 ∀，就只能给出 length 定义域类型的可能取值以及值域类型的可能取值，如：

$$list(integer) \rightarrow integer$$
$$list(list(char)) \rightarrow integer$$

像(6-5)这样的类型表达式是我们能够对多态函数类型做出的最一般描述。

符号 ∀ 称为*全称量词*，它所作用的类型变量称为由它*约束*的。约束变量可以任意换名，只要所有出现该变量的地方都换成相应的名称即可。所以，类型表达式

$$\forall\gamma.\ pointer(\gamma) \rightarrow \gamma$$

等价于(6-4)。通常含有符号 ∀ 的类型表达式可以非正式地称为“多态类型”。

我们用于检查多态函数的语言是由图6-16中的文法产生的。

该文法产生的程序是由一系列声明和要被检查的表达式 *E* 组成的。例如，

```
deref : ∀α. pointer(α) → α;
    q : pointer(pointer(integer));
deref(deref(q))
```
(6-6)

```
P → D ; E
D → D ; D | id : Q
Q → ∀ type_variable . Q | T
T → T '→' T
  | T × T
  | unary_constructor ( T )
  | basic_type
  | type_variable
  | ( T )
E → E ( E ) | E , E | id
```

图6-16 带有多态函数的语言的文法

为简化问题，我们用非终结符 *T* 来直接产生类型表达式。构造符→形成函数类型，×形成乘积类型。**unary_constructor** 所表示的一元构造符允许写出形如 *pointer* (*integer*) 和 *list* (*integer*) 的类型。括号只是用来组合类型。用于产生被检查表达式的语法很简单：它们可以是标识符、

368

组成元组的表达式序列或者作用在变元上的函数。

多态函数的类型检查规则与6.2节中普通函数的类型检查规则存在3个方面的区别。我们首先用程序(6-6)中的表达式 deref(deref(q))来说明这些区别，然后给出类型检查规则。该表达式的语法树如图6-17所示。其中每个节点附带两个标号，第一个标号指出该节点所代表的子表达式，第二个标号是分配给该子表达式的类型表达式。下标 i 和 o 分别用来区分括号内出现的 deref 和括号外出现的 deref。

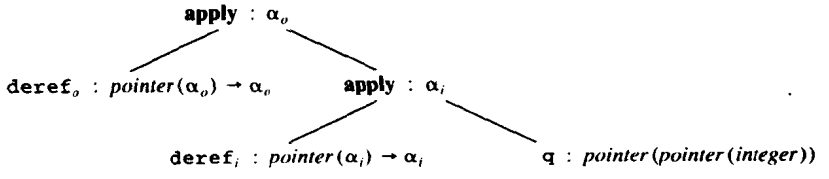


图6-17 deref(deref(q))的带标号的语法树

多态函数的类型检查规则和普通函数的类型检查规则存在如下区别：

1. 多态函数在同一表达式中的不同地方出现，其变元类型不必相同。在表达式 $\mathtt{deref}_o(\mathtt{deref}_i(\mathtt{q}))$ 中，$\mathtt{deref}_i$ 删除了对指针的一级间接访问，所以 $\mathtt{deref}_o$ 和 $\mathtt{deref}_i$ 作用在不同类型的变元上。这一性质的实现基于对 $\forall\alpha$ 的解释，即“对任意类型 α”。deref 的每次出现对(6-4)中的约束变量 α 代表什么类型都可以有不同的解释。所以我们为 deref 的每次出现分配一个类型表达式，该类型表达式是用新的类型变量代替(6-4)中的 α，并删除全称量词 $\forall$ 而形成的。在图6-17中，新类型变量 α_o 和 α_i 分别用在分配给 deref 的内部出现和外部出现的类型表达式中。

2. 因为变量可以出现在类型表达式中，所以我们必须重新检查类型等价的概念。假如类型为 $s \rightarrow s'$ 的 E_1 作用于类型为 t 的 E_2 上，我们必须将它们“合一”，而不是简单地确定 s 和 t 的等价性。合一的非形式化定义如下：通过将 s 和 t 中的类型变量用类型表达式来代替，我们才能确定 s 和 t 是否结构等价。如图6-17中标记为 **apply** 的内节点，如果 α_i 被 *pointer* (*integer*) 代替，则等式

369 $$pointer(\alpha_i) = pointer(pointer(integer))$$

成立。

3. 我们需要定义一种方法来记录两个表达式合一的结果。通常，一个类型变量可以出现在多个类型表达式中，如果 s 和 s' 的合一导致变量 α 代表类型 t，那么在类型检查过程中，α 必须继续代表类型 t。例如，在图6-17中，α_i 是 $\mathtt{deref}_i$ 的值域类型，因此我们可以用它来代表 $\mathtt{deref}_i(\mathtt{q})$ 的类型。于是，将 $\mathtt{deref}_i$ 的定义域类型和 q 的类型合一会影响标记为 **apply** 的内节点的类型表达式。图6-17中的另一个类型变量 α_o 代表 *integer*。

6.6.4 代换、实例和合一

通过定义一个称为代换的从类型变量到类型表达式的映射，我们可以将变量所代表的类型信息形式化。下面的递归函数 *subst*(*t*) 可以精确地应用代换 S 来替换表达式 t 中的所有类型变量。通常，我们用函数类型构造符作为“典型”构造符。

```
function subst(t : type_expression) : type_expression;
begin
    if t是基本类型 then return t
    else if t是变量 then return S(t)
    else if t是 t1→t2 then return subst(t1)→subst(t2)
end
```

为方便起见，我们用 $S(t)$ 来表示将 *subst* 作用于 t 后所得到的结果类型表达式，$S(t)$ 称为 t 的实例。如果代换 S 没有为变量 α 指定相应的表达式，那么我们假定 $S(\alpha)$ 就是 α，即 S 是这种变量上的恒等映射。

例6.10 在下面的示例中，我们用 $s < t$ 表示 s 是 t 的实例：

$$
\begin{aligned}
pointer(integer) &< pointer(\alpha)\\
pointer(real) &< pointer(\alpha)\\
integer \rightarrow integer &< \alpha \rightarrow \alpha\\
pointer(\alpha) &< \beta\\
\alpha &< \beta
\end{aligned}
$$

但是，对于下面的示例，左边的类型表达式不是右边类型表达式的实例（原因列在旁边）：

integer	*real*	代换不能用于基本类型
integer → *real*	$\alpha \rightarrow \alpha$	α 的代换不一致
integer → α	$\alpha \rightarrow \alpha$	α 的所有出现都应该被替换

□

如果存在某个代换 S，使得 $S(t_1) = S(t_2)$，那么 t_1 和 t_2 就能合一。实际上，我们感兴趣的是 370
最一般的合一代换，它是对表达式中的变量限制最少的代换。更精确地说，表达式 t_1 和 t_2 最一般的合一代换是具有如下性质的代换 S：

1. $S(t_1) = S(t_2)$，并且

2. 对任何其他满足 $S'(t_1) = S'(t_2)$ 的代换 S'，代换 S' 是 S 的实例（即对任何 t，$S'(t)$ 是 $S(t)$ 的实例）。

以后我们所说的合一都是指最一般的合一代换。

6.6.5 多态函数的检查

检查由图6-16中的文法产生的表达式的规则将根据下面的操作编写，这些操作作用在表示类型的图上：

1. *fresh*(t) 用新的变量代替类型表达式 t 中的约束变量，返回指向代表结果类型表达式的节点的指针。在该过程中，删除 t 中所有的符号 $\forall$。

2. *unify* (m, n)。将 m 和 n 所指向的节点所代表的类型表达式合一。该操作具有一定的副作用，即记录使表达式等价的代换。如果表达式不能合一，则整个类型检查失败。[⊖]

类型图中的各个叶节点和内节点使用类似于5.2节中的 *mkleaf* 和 *mknode* 操作来构造。必须为每个类型变量构造惟一的叶节点，而其他结构上等价的类型表达式则不必用惟一的节点来表示。

unify 操作基于下列关于合一和代换的图论公式。假定图中节点 m 和 n 分别代表类型表达式 e 和 f，如果 $S(e) = S(f)$，则称节点 m 和 n 在代换 S 下等价。寻找最一般的合一代换 S 的问题可以归结为分组问题，即把节点划分为在 S 下一定等价的节点集合。在 S 下等价的表达式其根也一定等价。两个节点 m 和 n 等价的充分必要条件是，其根代表同样的操作符，而且它们对应的子节点等价。

将两个表达式合一的算法将在下一节给出。该算法能够记住在已经出现的代换下等价的节点集。

表达式的类型检查规则如图6-18所示。此处，我们没有说明如何处理声明。当检查由非终 371

⊖ 中止类型检查过程的原因是某些合一操作的副作用可能在检测出错之前被记录下来。如果合一操作的副作用被推迟到表达式完成成功合一，则可以实现错误恢复。

结符 T 和 Q 产生的类型表达式时，根据5.2节中 dag 的构造算法，*mkleaf* 和 *mknode* 将节点加到类型图中。声明一个标识符时，声明中的类型信息保存在符号表中，其形式为指向代表该类型的节点的指针。在图6-18中，该指针被称为综合属性 **id**.*type*。同上，*fresh* 操作用新变量代替约束变量，并去掉 ∀ 符号。和产生式 $E \rightarrow E_1, E_2$ 相关联的动作将 *E.type* 设置为 E_1 和 E_2 类型的积。

$E \rightarrow E_1$ (E_2)	{ p := *mkleaf* (*newtypevar*); *unify* (E_1.*type*, *mknode* ('→', E_2.*type*, p)); *E.type* := p }
$E \rightarrow E_1$, E_2	{ *E.type* := *mknode* ('×', E_1.*type*, E_2.*type*) }
$E \rightarrow$ **id**	{ *E.type* := *fresh* (**id**.*type*) }

图6-18 检查多态函数的翻译模式

函数 $E \rightarrow E_1(E_2)$ 的类型检查规则是从考虑 E_1.*type* 和 E_2.*type* 都是类型变量（即 E_1.*type* = α 且 E_2.*type* = β）的情况引申出来的。此时 E_1.*type* 一定是函数，并使得对于某一未知类型 γ，有 α = β→γ 成立。在图6-18中，创建一个与 γ 对应的新类型变量，并将 E_1.*type* 和 E_2.*type*→γ 进行合一。每次调用 *newtypevar* 都将返回一个新的类型变量，相应的叶节点由 *mkleaf* 构造，要与 E_1.*type* 合一的、代表函数 E_2.*type*→γ 的节点由 *mknode* 来构造。合一后，新的叶节点代表结果类型。

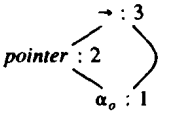
我们用一个简单的例子来详细说明图6-18中的规则。在图6-19中，通过写出为每个子表达式分配的类型表达式，我们总结了该算法的工作过程。每次调用函数时，*unify* 操作都可能有副作用，即记录某些类型变量的类型表达式。这样的副作用如图6-19的代换栏所示。

表达式：类型	代换
q : *pointer* (*pointer* (*integer*))	
$\texttt{deref}_i$: *pointer* (α_i) → α_i	
$\texttt{deref}_i$(q) : *pointer* (*integer*)	α_i = *pointer* (*integer*)
$\texttt{deref}_o$: *pointer* (α_o) → α_o	
$\texttt{deref}_o(\texttt{deref}_i(\texttt{q}))$: *integer*	α_o = *integer*

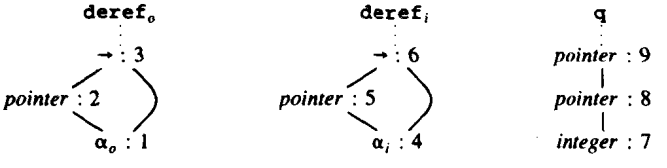
373

图6-19 自底向上确定类型的总结

例6.11 对程序(6-6)中表达式 $\texttt{deref}_o(\texttt{deref}_i(\texttt{q}))$ 的类型检查从叶节点开始自底向上进行处理。下标 *o* 和 *i* 仍然用来区别不同位置出现的 deref。考虑子表达式 $\texttt{deref}_o$ 时，*fresh* 用新类型变量 α_o 构造下列节点：

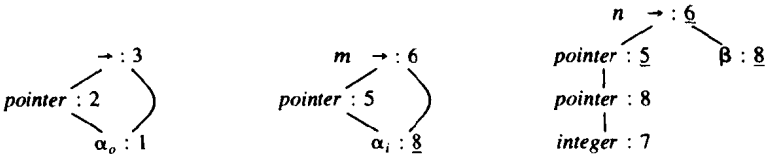


节点上的编号表示节点所属的等价类。类型图中有关这3个标识符的部分如下所示。虚线表示节点3、6和9分别代表 $\texttt{deref}_o$、$\texttt{deref}_i$ 和 q。



函数 $\texttt{deref}_i$(q) 是通过构造从 q 的类型到新类型变量 β 的函数节点 *n* 来检查的。该函数成

功地与节点 m 所代表的类型 $\mathtt{deref}_i$ 合一。在节点 m 和 n 合一之前，每个节点都有不同的编号。合一以后，等价节点就是下面那些具有相同编号的节点，编号改变的节点下面加了下划线。



注意，α_i 和 *pointer*(*integer*) 的节点编号都是8，即 α_i 和该类型表达式合一，如图6-19所示。接下来，α_o 和 *integer* 合一。 □

下面的例子将ML中的多态函数的类型推断和图6-18的类型检查规则联系在一起。ML中函数定义的语法由下式给出：

fun $\mathbf{id}_0$ ($\mathbf{id}_1$, . . . , $\mathbf{id}_k$) = E ;

其中，$\mathbf{id}_0$ 代表函数名，$\mathbf{id}_1$，…，$\mathbf{id}_k$ 代表其参数。为简单起见，假定表达式 E 的语法如图6-16所示，而且 E 中的标识符只有函数名、函数参数和内置函数。 373

该方法是例6.9中方法的形式化，在例6.9中，我们为 deref 推断出一个多态类型。在这里，我们为函数名及其参数构造新的类型变量。内置函数通常具有多态类型，出现在这些类型中的任何类型变量都受全称量词 ∀ 的约束。然后，我们检查表达式 $\mathbf{id}_0(\mathbf{id}_1, \cdots, \mathbf{id}_k)$ 的类型和 E 的类型是否匹配。如果匹配成功，我们就可以推断出函数名的类型。最后，为给出该函数的多态类型，在推断出的类型中，任何变量都受全称量词 ∀ 的约束。

例6.12 回想一下图6-14中确定列表长度的ML函数：

```
fun length(lptr) =
        if null(lptr) then 0
        else length(tl(lptr)) + 1;
```

类型变量 β 和 γ 是分别为类型 length 和 lptr 引入的。我们发现，length(lptr)的类型与形成函数体的表达式的类型相匹配，而且 length 一定具有类型

对任何类型α，*list*(α) → *integer*

所以 length 的类型为

∀α. *list*(α) → *integer*

更详细地说，我们构造如图6-20所示的程序，并在该程序上应用图6-18中的类型检查规则。该程序中的声明将 length 和 lptr 与新类型变量 β 和 γ 联系起来，并显式地给出内置操作的类型。我们将按图6-16的格式给出条件语句，即将多态操作符if作用于3个操作对象上，这3个操作对象分别代表测试条件、then 部分以及 else 部分。声明语句表明，then 部分和 else部分可与任何类型匹配，同时也是结果的类型。

```
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6-20 类型声明及要检查的表达式

显然，length(lptr)和函数体必须具有同样的类型，该检查用操作符 match 来表示。match 的使用使得所有的检查都可用图6-16那种风格的程序来完成，它提供了一种技术上的便利。

将图6-18的类型检查规则用于图6-20中的程序，其结果如图6-21所示。由操作 *fresh* 引入的新变量用于描述内置操作的多态类型，这些新变量根据 α 的下标来区分。由第(3)行得知，length一定是从 γ 到某个未知类型 δ 的函数。然后，当检查子表达式 null(lptr)时，在第(6)行我们发现 γ 和 *list* (α_n)合一，其中 α_n 是一个未知类型。此时，我们知道 length 的类型一定是：

对任何类型 α_n，　$list(\alpha_n) \rightarrow \delta$

行	表达式：类型	代换
(1)	lptr : γ	
(2)	length : β	
(3)	length(lptr) : δ	$\beta = \gamma \rightarrow \delta$
(4)	lptr : γ	
(5)	null : $list(\alpha_n) \rightarrow boolean$	
(6)	null(lptr) : *boolean*	$\gamma = list(\alpha_n)$
(7)	0 : *integer*	
(8)	lptr : $list(\alpha_n)$	
(9)	tl : $list(\alpha_t) \rightarrow list(\alpha_t)$	
(10)	tl(lptr) : $list(\alpha_n)$	$\alpha_t = \alpha_n$
(11)	length : $list(\alpha_n) \rightarrow \delta$	
(12)	length(tl(lptr)) : δ	
(13)	1 : *integer*	
(14)	+ : $integer \times integer \rightarrow integer$	
(15)	length(tl(lptr))+1 : *integer*	$\delta = integer$
(16)	if : $boolean \times \alpha_i \times \alpha_i \rightarrow \alpha_i$	
(17)	if(···) : *integer*	$\alpha_i = integer$
(18)	match : $\alpha_m \times \alpha_m \rightarrow \alpha_m$	
(19)	match(···) : *integer*	$\alpha_m = integer$

图6-21　推断 length 的类型 $list(\alpha_n) \rightarrow integer$

374 ~ 375

最后，在第(15)行对加法进行检查时，将 δ 与 *integer* 合一。此处，为清晰起见，将“+”号写在两变元之间。

检查完毕时，类型变量 α_n 仍留在 length 的类型中。因为对 α_n 的类型没有做任何假设，所以使用该函数时，可以用任何类型代替它。因此我们把它作为约束变量，并将 length 的类型写为

$\forall \alpha_n.\ list(\alpha_n) \rightarrow integer$　□

6.7 合一算法

非形式化地讲，合一就是确定将两个表达式 *e* 和 *f* 中的变量用相应的表达式替换后能否变成等价表达式的问题。测试表达式相等是合一的特殊情况。如果 *e* 和 *f* 中只包含常量而没有变量，则当且仅当 *e* 和 *f* 完全相同时 *e* 和 *f* 合一。本节的合一算法可以用于带环图，所以该算法可用于测试存在环类型的结构等价。[⊖]

⊖ 在某些应用中，将一个变量和包含该变量的表达式合一是错误的。算法6.1允许这种代换。

在上一节，合一是基于称为代换的函数 S（从变量到表达式的映射）来定义的。如果将e中的每个变量 α 均用 $S(\alpha)$ 代换，则得到的表达式记为 $S(e)$。如果 $S(e) = S(f)$，则称 S 为 e 和 f 的合一代换。本节的算法旨在确定这样一种代换：它是一对表达式上的最一般的合一代换。

例6.13 为了对最一般的合一代换有一个整体概念，考虑下面两个类型表达式：

$$((\alpha_1 \to \alpha_2) \times list(\alpha_3)) \to list(\alpha_2)$$
$$((\alpha_3 \to \alpha_4) \times list(\alpha_3)) \to \alpha_5$$

这两个表达式的合一代换 S 和 S' 见右表。

x	$S(x)$	$S'(x)$
α_1	α_3	α_1
α_2	α_2	α_1
α_3	α_3	α_1
α_4	α_2	α_1
α_5	$list(\alpha_2)$	$list(\alpha_1)$

这两个代换将 e 和 f 映射成：

$$S(e) = S(f) = ((\alpha_3 \to \alpha_2) \times list(\alpha_3)) \to list(\alpha_2)$$
$$S'(e) = S'(f) = ((\alpha_1 \to \alpha_1) \times list(\alpha_1)) \to list(\alpha_1)$$

代换 S 是 e 和 f 的最一般的合一代换。注意，$S'(e)$ 是 $S(e)$ 的一个实例，因为将 $S(e)$ 中的两个变量都用 α_1 替代即可得到 $S'(e)$。但是，反之则不成立，因为 $S'(e)$ 中的 α_1 的每一次出现都必 376
须用同一个表达式来代替，所以通过代换 $S'(e)$ 中的变量 α_1 不可能获得 $S(e)$。 □

当用树表示要合一的表达式时，尽管 S 是最一般的合一代换，但代换后的表达式 $S(e)$ 对应的树节点个数却是 e 和 f 对应的树节点个数的指数倍。然而，如果用图而不是用树来表示表达式和代换，那么节点数目爆炸式增大的情况就不会发生。

要将合一的基于图论的形式化描述实现为算法，关键是将最一般的合一代换下等价的两个表达式所对应的节点进行分组。例6.13中的两个表达式由图6-22中标记为→:1的两个节点表示。在节点1被合一之后，节点上的整数表示节点所属的等价类编号。等价类具有如下性质，即同一等价类中的所有内节点表示相同的操作符。在一个等价类中，内部节点对应的子节点也都是等价的。

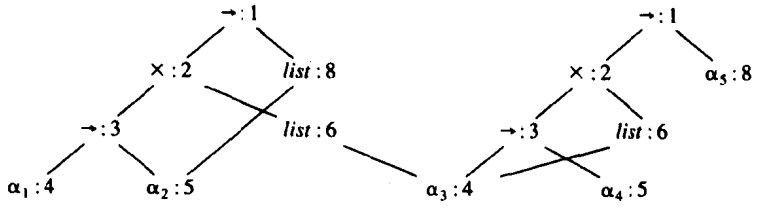


图6-22 合一后的等价类

算法6.1 图中一对节点的合一。

输入：一个图和一对要合一的节点 m 和 n。

输出：如果节点 m 和 n 所表示的表达式合一，则输出布尔值 true，否则输出 false。如果算法中的函数不是返回 false，而是改为导致失败，那么该算法就是图6-18中类型检查规则所需的 *unify* 的另一个版本。

方法：节点用图6-23中的记录来表示，该记录包含一个二元操作符域和指向左右儿子的指针。

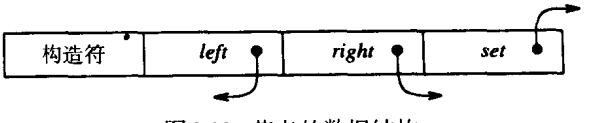


图6-23 节点的数据结构

等价节点的集合用 *set* 域来保存。每个等价类都选择一个节点作为该等价类的惟一代表，该节点的 *set* 域置为空指针，等价类中其他节点的 *set* 域均指向（可能间接地通过集合中的其他节点）这个代表节点。初始时，每个节点 *n* 的等价类就是自身，而且 *n* 也是其自身的代表节点。

```
function unify (m, n: node): boolean
begin
    s := find (m);
    t := find (n);
    if s = t then
        return true
    else if s 和 t 是代表同一基本类型的节点 then
        return true
    else if s是一个带有子节点 s1 和 s2 的操作符节点and
            t 是一个带有子节点 t1 和 t2 的操作符节点 then begin
        union (s, t);
        return unify (s1, t1) and unify (s2, t2)
    end
    else if s 或 t 代表一个变量 then begin
        union (s, t);
        return true
    end
    end return false
        /* 带有不同操作符的内节点不能被合一 */
end
```

图6-24 合一算法

图6-24中的合一算法使用了下面两个作用在节点上的操作：

1. *find* (*n*) 返回节点 *n* 当前所在等价类的代表节点。

2. *union*(*m*, *n*) 合并节点 *m* 和 *n* 所在的两个等价类。如果两个等价类中的某一代表节点是非变量节点，那么 *union* 将该节点作为合并后等价类的代表节点。否则 *union* 任选其中的一个作为新的代表节点。在 *union* 的说明中，这种非对称性很重要，因为变量不能当作一个包含类型构造符或基本类型的表达式的等价类代表。否则的话，两个不等价的表达式就可能通过该变量合一。

377 ~ 378

集合上的 *union* 操作可通过修改其中一个等价类的代表上的 *set* 域，使其指向另一等价类的代表来实现。为寻找某节点所属的等价类，我们逐级查找节点的 *set* 指针，直到到达代表节点（该节点的 *set* 域为空指针）为止。

注意，图6-24中的算法分别使用 *s* = *find* (*m*)和*t* = *find* (*n*)，而不是 *m* 和 *n*。如果 *m* 和 *n* 在同一等价类中，则代表节点 *s* 和 *t* 相等。如果 *s* 和 *t* 代表同一基本类型，则 *unify* (*m*, *n*)返回 true。如果 *s* 和 *t* 都是代表二元类型构造符的内部节点，我们先碰碰运气将其合并，然后递归地检查它们各自的子节点是否等价。通过先进行合并，在递归检查子节点之前，等价类的数目就会减少，所以算法最终会停止。

用表达式代换变量是通过将代表变量的叶节点加到包含该表达式对应节点的等价类中来实现的。如果 *m* 或 *n* 是代表变量的叶节点，而且已经被加到某节点所属的等价类中，而该节点是一个包含类型构造符或基本类型的表达式所对应的节点，那么其上的 *find* 将返回该类型构造符或基本类型的代表，所以此时变量不可能与两个不同的表达式进行合一。 □

例6.14 在图6-25中，我们给出了例6.13中两个表达式所对应的图的初始形式。此时，图中各节点都已被编号并且各自处于它自己的等价类中。现在要计算 *unify*(1, 9)，算法注意到节

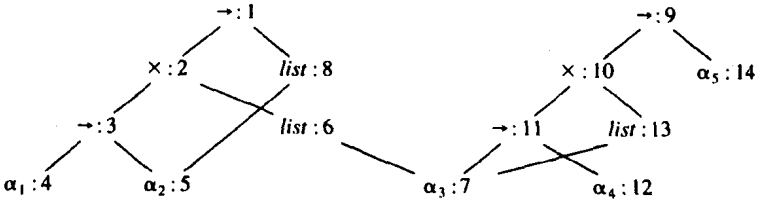


图6-25 每个节点都属于它自己的等价类的初始 dag

点1和节点9代表相同的操作符，所以将1和9合并到同一个等价类中。之后，递归调用 *unify*(2, 10)和 *unify*(8, 14)。计算 *unify*(1, 9)的结果是图6-22中所示的图。 □

如果算法6.1返回 true，我们可以如下构造代换 S（可充当某个合一代换）。令结果图中的每个节点 n 代表与 *find* (n) 相关联的表达式。这样，对每一个变量 α，*find* (α) 给出节点 n，它是 α 所在等价类的代表。n 所代表的表达式就是 $S(\alpha)$。例如，在图6-22中，α_3 的代表节点为4， 379
同时它也代表α_1，α_5的代表节点为8，同时它也代表 *list*(α_2)。

例6.15 算法6.1可以用于测试下面两个类型表达式是否结构等价：

```
e : real → e
f : real → (real → f)
```

它们的类型图如图6-26所示。为方便起见，每个节点均已编号。

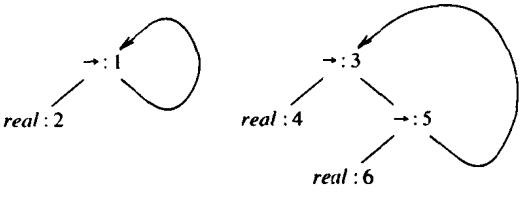


图6-26 两个带环类型的图

我们调用 *unify* (1, 3) 来测试这两个表达式是否结构等价。算法将节点1和3并入某等价类中，并递归调用 *unify* (2, 4) 和 *unify* (1, 5)。由于 2 和 4 代表同样的基本类型，所以调用 *unify* (2, 4) 返回true。调用 *unify* (1, 5) 时将5加入1和3的等价类中，接下来，再递归调用 *unify* (2, 6) 和 *unify* (1, 3)。

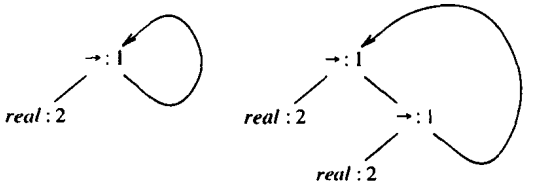


图6-27 说明节点等价类的类型图

调用 *unify* (2, 6)返回 true，因为2和6也代表同样的基本类型。第二次调用 *unify* (1, 3)时，因为我们已经将节点1和3并入同一等价类中了，所以算法终止。返回值为 true，说明这两个类型表达式确实等价。最后节点的等价类如图6-27所示，标记相同整数的节点在同一个等价类中。 □ 380

练习

6.1 请写出下列类型的类型表达式：

a) 指向实数的指针数组，数组的下标从1到100。

b) 二维整型数组（即数组的数组），其行下标从0到9，列下标从-10到10。

c) 函数，其定义域是从整数到整型指针的函数，值域是整数和字符组成的记录。

6.2 设有如下C语言声明：

```
typedef struct {
    int a, b;
    } CELL, *PCELL;
CELL foo[100];
PCELL bar(x, y) int x; CELL y { ··· }
```

请写出foo和bar的类型表达式。

6.3 下面的文法定义了建立在文字（literal）列表上的列表。各个符号的含义和图6-3中文法的符号相同，只是增加了 **list** 类型，它表示元素类型为 T 的列表：

$$P \rightarrow D\ ;\ E$$
$$D \rightarrow D\ ;\ D \mid \mathbf{id}\ :\ T$$

$$
\begin{array}{lll}
T & \rightarrow & \textbf{list of } T \mid \textbf{char} \mid \textbf{integer} \\
E & \rightarrow & (\ L\ ) \mid \textbf{literal} \mid \textbf{num} \mid \textbf{id} \\
L & \rightarrow & E\ ,\ L \mid E
\end{array}
$$

请写出类似于6.2节的翻译模式以确定表达式（E）和表（L）的类型。

6.4 在练习6.3的文法中加入产生式 $E \rightarrow$ **nil**，意即表达式可以为空。修改练习6.2的答案以考虑这样的事实，即 **nil** 可以代表元素为任意类型的空表。

6.5 使用6.2节的翻译模式，计算下面程序段中表达式的类型。给出分析树中每个节点的类型。

a)
```
c: char; i: integer;
c mod i mod 3
```

381

b)
```
p: ↑integer; a: array [10] of integer;
a[p↑]
```

c)
```
f: integer → boolean;
i: integer; j: integer; k: integer;
while f(i) do
    k := i;
    i := j mod i;
    j := k
```

6.6 修改6.2节中检查表达式类型的翻译模式，使其在发现错误时打印错误说明信息，并继续检查，就好像已经遇到了期望的类型。

6.7 重写6.2节中表达式的类型检查规则，使其访问表示类型表达式的图节点。重写后的规则应该使用类 Pascal 语言所支持的数据结构和操作。在下列情况下使用类型表达式的结构等价：

a) 类型表达式用树表示，如图6-2所示，且

b) 类型图是一个dag，每个类型表达式对应惟一的节点。

6.8 修改图6-5的翻译模式，使之能处理如下情况：

a) 具有值的语句。赋值语句的值是赋值号“:=”右边的表达式的值；条件语句或 while 语句的值是语句体的值；而语句列表的值是列表中最后一个语句的值。

b) 布尔表达式。为逻辑操作符 **and**、**or**、**not** 和比较操作符（“<”等）增加产生式，然后，增加给出这些表达式的类型的适当的翻译规则。

6.9 推广6.2节最后给出的函数的类型检查规则，使之能处理n元函数。

6.10 假定类型名 `link` 和 `cell` 像在6.3节中那样定义。下面的表达式中哪些是结构等价的？哪些是名字等价的？

i) **link**.
ii) *pointer*(**cell**).
iii) *pointer*(**link**).
iv) *pointer*(*record*((**info** × *integer*) × (**next** × *pointer*(**cell**)))

382

6.11 重新描述图6-6中测试结构等价的算法，使得*sequiv*的参数为指向dag中节点的指针。

6.12 考虑例6.1中将受限的类型表达式编码为二进制序列的方法。在Johnson[1979]中，构造符的两位域编码以相反的顺序出现，而最外层的构造符编码紧挨着基本类型的四位域编码。例如，

类型表达式	编码
char	000000 0001

freturns (*char*)	000011 0001
pointer (*freturns* (*char*))	001101 0001
array (*pointer* (*freturns* (*char*)))	110110 0001

使用 C 的操作符编写代码以从 *t* 的表示构造 *array* (*t*)的表示，以及从 *array* (*t*)的表示构造 *t* 的表示。假定按下述方式编码：

a) Johnson[1979]。

b) 例6.1。

6.13 假设每个标识符的类型都是一个整型子界。对含有操作符 +、-、*、div 和 mod 的表达式（如 Pascal 中的表达式），请写出为每个子表达式分配子界的类型检查规则，要求子表达式的值必须落在所分配的子界中。

6.14 给出测试 C 语言中类型等价的算法（参见例6.4）。

6.15 有些语言，如 PL/I，会强制地将布尔值转换成整数，即用1表示 true，用0表示 false。例如，3<4<5会被组合成 (3<4)<5，而且其值为 true（或1），因为3<4的值为1且1<5为 true。为布尔表达式构造翻译规则以完成这种强制转换。需要时，可在中间语言中使用条件语句将整数值赋给代表布尔表达式值的临时变量。

6.16 推广图6-9和图6-12中的算法，使之能够处理含有类型构造符 *array*、*pointer* 以及笛卡儿积的表达式。

6.17 下面哪些递归的类型表达式是等价的？

e1 = *integer* → e1
e2 = *integer* → (*integer* → e2)
e3 = *integer* → (*integer* → e1)

6.18 使用例6.6中的规则确定下面的哪些表达式具有惟一的类型。假定z是一个复数。

a) **1 * 2 * 3**
b) **1 * (z * 2)** 383
c) **(1 * z) * z**

6.19 假设我们允许例6.6的类型转换。当 *a*、*b* 和 *c* 的类型（整型或复型）满足什么条件时，表达式 (*a***b*)**c* 将具有惟一的类型？

6.20 使用类型变量表示下列函数的类型：

a) 函数 *ref*，它以任意类型的对象为参数，并返回指向该对象的指针。

b) 一个函数，它以下标为整数的数组为参数，数组元素为任意类型，而且返回一个数组，其元素是参数数组中元素所指向的对象。

6.21 找出下列类型表达式的最一般的合一代换：

i) (*pointer* (α)) × (β → γ)

ii) β × (γ → δ)

如果(ii)中的 δ 是 α，结果又是什么？

6.22 从下面的表达式列表中找出每对表达式的最一般的合一代换，或证实不存在这种代换：

a) $\alpha_1 \rightarrow (\alpha_2 \rightarrow \alpha_1)$

b) *array* (β_1) → (*pointer* (β_1) → β_3)

c) $\gamma_1 \rightarrow \gamma_2$

d) $\delta_1 \rightarrow (\delta_1 \rightarrow \delta_2)$

6.23 扩展例6.6中的类型检查规则使之适用于记录类型。记录的类型表达式和表达式使用下述附加产生式来产生：

```
    T → record fields end
    E → E . id
fields → fields ; field | field
 field → id : T
```

缺少类型名对能被定义的类型增加了什么限制？

* 6.24 6.5节中重载的解决分两阶段进行：首先，确定每个子表达式的可能类型的集合，第二阶段，在整个表达式的惟一类型确定之后，将每一个子表达式的可能类型的集合缩小到惟一类型。在自底向上的一遍扫描中，你将使用什么样的数据结构来解决重载问题？

** 6.25 如果标识符声明是可选的，那么解决重载问题将变得更加困难。更确切地说，假设声明可以用于重载代表函数符号的标识符，但未声明标识符的所有出现都具有相同类型。试证明：确定该语言中的表达式是否具有有效类型的问题是一个 NP 完全问题。该问题是在对实验语言 Hope（Burstall, MacQueen, and Sannella[1980]）进行类型检查时出现的。

384

6.26 仿照例6.12，对下面的 map 推断其多态类型：

map : ∀α. ∀β. ((α→β)×*list*(α)) → *list*(β)

map的 ML 定义为

```
fun map(f, l) =
                if null(l) then nil
                else cons( f(hd(l)), map(f, tl(l)))
```

在该函数体中，内置标识符的类型为

```
null :   ∀α. list(α) → boolean;
 nil :   ∀α. list(α);
cons :   ∀α. (α×list(α)) → list(α);
  hd :   ∀α. list(α) → α;
  tl :   ∀α. list(α) → list(α);
```

** 6.27 证明6.7节的合一算法能确定出最一般的合一代换。

* 6.28 修改6.7节的合一算法，使之不会将变量和包含该变量的表达式合一。

** 6.29 假设用树来表示表达式。找出表达式 e 和 f，使得对任何合一代换 S，$S(e)$中的节点数是 e 和 f 中节点数的指数倍。

6.30 如果两个节点代表等价的表达式，那么它们称为全等的。即使在初始类型图中任何两个节点都不是全等的，但合一以后，不同的节点有可能是全等的。

a) 给出一个算法，将一类互相全等的节点合并为一个节点。

** b) 扩展(a)中的算法，合并全等的节点直到没有两个不同的节点全等为止。

* 6.31 在图6-28中的完整 C 程序中，第(9)行的表达式 g(g) 是一个调用自身的函数。第(3)行的声明指出 g 的值域类型为 *integer*，而 g 的参数类型没有被指定。当试着运行该程序时，编译器会发出警告，因为第(3)行将 g 声明为函数类型，而不是指向函数的指针。

a) 关于 g 的类型你能说些什么？ 385

b) 使用图6-18中多态函数的类型检查规则推断下面程序中 g 的类型。

```
    m : integer;
times : integer × integer → integer;
    g : α;

times( m, g(g) )
```

```
(1)  int n;

(2)  int f(g)
(3)  int g();
(4)  {
(5)      int m;
(6)      m = n;
(7)      if( m == 0 ) return 1;
(8)      else {
(9)          n = n - 1; return m * g(g);
(10)     }
(11) }

(12) main()
(13) {
(14)     n = 5; printf("%d factorial is %d\n", n, f(f) );
(15) }
```

图6-28 一个包含调用自身的函数的 C 程序

参考文献注释

在Fortran和Algol 60这样的早期语言中，对基本类型和类型构造符的限制比较严，所以类型检查不是严重的问题。因此，在它们的编译器中，对类型检查的描述就隐藏在对表达式代码生成的讨论中。Sheridan[1959]描述了最初的Fortran编译器对表达式的翻译。该编译器能知道表达式的类型是整型还是实型，但该语言不允许强制类型转换。Backus[1981, p.54]回顾道："我认为只是因为我们不喜欢混合模式表达式的规则，所以我们决定：'让我们抛掉它，这样会更简单。'" Naur[1965]是早期关于 Algol 编译器中的类型检查的文章。该编译器所使用的技术类似于6.2节所讨论的技术。

Zuse在他的Plankalkül中抢先使用了数据的结构化措施，如数组和记录，但Plankalkül没有什么直接影响（Bauer and Wössner[1972]）。Algol 68是最早允许系统地构造类型表达式的语言之一。它可以递归地定义类型表达式，并采用结构等价。名字等价和结构等价的明显区别是在 386
EL1中发现的，选择哪一种由程序员决定（Wegbreit[1974]）。Welsh, Sneeringer, and Hoare [1977] 对 Pascal 的批评引起了对该区别的注意。

强制类型转换和重载的结合可能导致二义性：强制转换变元的类型可能导致重载被解析为另一种算法。因此要对强制类型转换或者重载加以限制。PL/I采用了一种对强制类型转换不加约束的方法，其中，第一个设计准则是："什么都行。如果符号的特定结合具有合适的切合实际的含意，就将其作为正式的含意（Radin and Rogoway[1965]）。" 基本类型集合常常是排序的（例如，Hext[1967]中描述了一种施加在 CPL 基本类型上的栅格结构）并且低层类型可能被强制转换成高层类型。

在APL（Iverson[1962]）和SETL（Schwartz[1973]）这样的语言中，重载的编译时解可以潜在地降低程序的运行时间（Bauer and Saal[1974]）。Tennenbaum[1974]将下面两种解加以区别：一种是从操作数确定操作符可能类型集的“前向”解，另一种是基于上下文期望类型的“后向”解。使用类型栅格，Jones and Muchnick[1976]以及Kaplan and Ullman[1980]解决了对前向和后向分析所获得的类型的约束。Ada中的重载可以通过一遍前向分析跟以一遍后向分析来解决，如6.5节所述。该结论出现在许多论文中，如Ganzinger and Ripken[1980]、Pennello, DeRemer and Meyers[1980]、Janas[1980]、Persch et al. [1980]。Cormack[1981]中提供了一种递归实现，而Baker[1982]则通过携带可能类型的dag而避免了一遍后向分析。

Curry研究了与组合逻辑和λ演算（Church[1941]）有关的类型推断（Curry and Feys [1958]）。很早即发现λ演算是函数语言的核心。在本章中，我们已经反复使用对参数的函数调用来讨论类型检查的概念。在λ演算中，定义和应用函数可以不必考虑类型。Curry对它们的“函数特征”感兴趣，并想确定现在我们应怎样称呼最一般的多态类型，它是由类型表达式和全称量词组成的，如6.6节所述。受Curry的启发，Hindley[1969]发现合一可用于推断类型。Morris在他的论文中（Morris[1968a]）也独立地提出：通过建立一组方程并对其求解来将类型分配给λ表达式可以确定与变量相关的类型。在对Hindley的工作一无所知的情况下，Milner[1978]也发现合一可用于求解这组方程，并将该思想用于推断ML程序设计语言中的类型。

Cardelli[1984]中讨论了ML中类型检查的语用（pragmatics）。该方法已经被用于Meertens
387 [1983]所开发的一种语言中。Suzuki[1981]研究了它在Smalltalk 1976（Ingalls [1978]）上的应用。Mitchell[1984]中说明了如何将强制类型转换加进去。

Morris[1968a]发现递归或循环类型允许我们推断包含调用自身的函数的表达式类型。图6-28中的C语言程序包含一个调用自身的函数，它是受Ledgard[1971]中一个Algol程序的启发而得来的。练习6.31来自MacQueen, Plotkin, and Sethi[1984]，其中给出了一个递归多态类型的语义模型。McCracken[1979]和Cartwright[1985]中还给出了其他方法。Reynolds[1985]中综述了ML类型系统、避免与强制类型转换和重载有关的异常的理论方针以及高阶多态函数。

Robinson[1965]最先对合一进行了研究。从测试(1)有穷自动机和(2)带环路的链表（Knuth[1973a]，2.3.5节，练习11）的算法可以很容易地改造出6.7节的合一算法。Hopcroft and Karp[1971]给出的测试有穷自动机等价的几乎线性的算法可以看作是Knuth[1973a]594页概述的实现。通过数据结构的巧妙使用，Paterson and Wegman[1978]以及Martelli and Montanari[1982]中给出了无环情况下的线性算法。Downey，Sethi, and Tarjan[1980]中描述了寻找全等节点（见练习6.30）的算法。

Despeyroux[1984]中描述了一个类型检查器生成器，它使用模式匹配来从基于推理规则的
388 操作语义描述生成一个类型检查器。

第7章 运行时环境

在考虑代码生成以前，我们需要把静态的源程序正文和实现该程序的运行时必须发生的活动联系起来。程序执行时，源程序正文中同样的名字可以表示目标机器中不同的数据对象。本章就是要考察名字和数据对象之间的关系。

数据对象的分配和释放由*运行时支撑程序包*管理。它由一些例行子程序组成，这些子程序和所产生的目标代码一起装配。运行时支撑程序包的设计受过程的语义的影响。使用本章讨论的技术可以构造像 Fortran、Pascal 和 Lisp 等语言的支撑程序包。

过程的每次执行称为该程序的一个*活动*。如果过程是递归的，则在某一时刻可能有它的几个活动是活跃的。在 Pascal 中，过程的每次调用引起一个活动，这个活动可以操纵分配给它使用的数据对象。

数据对象运行时的表示由它的类型决定。通常，像字符、整数和实数这样的基本数据类型可以由目标机器中等价的数据对象表示。但是，像数组、字符串和结构这样的集合数据类型通常由一组基本对象表示，它们的空间安排将在第8章中讨论。

7.1 源语言问题

为便于说明，假定程序像 Pascal 那样由过程组成。读者应该注意区别过程的源正文和它运行时的活动。

7.1.1 过程

*过程定义*是一个声明，它的最简单形式是把一个标识符和一个语句联系起来。该标识符是*过程名*，而这个语句是*过程体*。例如，图7-1中的 Pascal 代码包括过程名为 `readarray` 的过程（第(3)～(7)行）；它的过程体为第(5)～(7)行。有返回值的过程在许多语言中叫做*函数*，不过，把它们称为过程也是可以的。完整的程序也可以看成一个过程。 389

当过程名出现在可执行语句中时，则称过程在该点*被调用*。过程调用就是执行被调用过程的过程体。图7-1中的主程序（第(21)～(25)行）在第(23)行调用过程 `readarray`，在第(24)行调用过程 `quicksort`。注意，过程调用也可以出现在表达式中（如在第(16)行中那样）。

出现在过程定义中的某些标识符是特殊的，称为该过程的*形式参数*（或*形参*）（C中称它们为形式变元，Fortran 称它们为哑变元）。第(12)行出现的标识符 `m` 和 `n` 就是过程 `quicksort` 的形参。称为*实在参数*（或*实参*）的变元被传递给被调用过程，它们取代过程体中的形式参数。建立实参和形参之间对应关系的方法将在7.5节中介绍。图7-1的程序中第(18)行用实参 `i+1` 和 `n` 来调用 `quicksort` 过程。

7.1.2 活动树

我们对程序执行时过程间的控制流作出如下假设：

1. 控制流是连续的，即程序的执行由一些连续的步骤组成，在任何一步，控制处于程序中的某点。

2. 过程的每次执行都从过程体的起点开始，最后控制返回到直接跟随在本次调用点之后的位置。这意味着过程间的控制流可以用树来描绘，稍后就会看到这一点。

```
(1) program sort(input, output);
(2)     var a : array [0..10] of integer;

(3)     procedure readarray;
(4)         var i : integer;
(5)         begin
(6)             for i := 1 to 9 do read(a[i])
(7)         end;

(8)     function partition(y, z: integer) : integer;
(9)         var i, j, x, v: integer;
(10)        begin ...
(11)        end;

(12)    procedure quicksort(m, n: integer);
(13)        var i : integer;
(14)        begin
(15)            if ( n > m ) then begin
(16)                i := partition(m,n);
(17)                quicksort(m,i-1);
(18)                quicksort(i+1,n)
(19)            end
(20)        end;

(21)    begin
(22)        a[0] := -9999; a[10] := 9999;
(23)        readarray;
(24)        quicksort(1,9)
(25)    end.
```

图7-1 读入整数并排序的Pascal程序

过程体的每次执行叫做该过程的一个活动。过程 p 的一个活动的生存期是从过程体执行的第一步到最后一步的步序列，包括执行被 p 调用的过程的时间，以及再由这样的过程调用过程所花的时间，等等。一般而言，术语“生存期”是指程序执行期间的连续的步序列。

在像Pascal这样的语言中，每次控制从过程 p 进入过程 q，最后都将返回到 p（除非出现致命错误）。更准确地说，每次控制流从过程 p 的活动记录进入过程 q 的某个活动记录时，它都会返回到 p 的同一活动记录中。

如果 *a* 和 *b* 是过程的活动，那么它们的生存期或者不交迭，或者嵌套。也就是说，如果在离开 *a* 之前进入 *b*，那么控制在离开 *b* 之后才能离开 *a*。

活动生存期的嵌套特性可以通过在每个过程中插入两个打印语句来说明，一个在过程体的第一条语句之前，另一个在最后一条语句之后。第一条打印语句打印输出 enter，然后是调用的过程名和实参的值；第二条打印语句打印输出leave，接着仍然是过程名和实参值。图7-1中的程序加入这两个打印语句之后的结果在图7-2中给出。活动 quicksort(1,9) 的生存期是在打印输出 enter quicksort(1,9)和leave quicksort(1,9) 之间执行的步序列。图7-2中假设 partition(1,9) 的返回值是4。

如果同一个过程的一次新的活动可以在前一次活动

```
Execution begins...
enter readarray
leave readarray
enter quicksort(1,9)
enter partition(1,9)
leave partition(1,9)
enter quicksort(1,3)
    ...
leave quicksort(1,3)
enter quicksort(5,9)
    ...
leave quicksort(5,9)
leave quicksort(1,9)
Execution terminated.
```

图7-2 图7-1中过程活动的输出

结束前开始，则称这样的过程是*递归的*。图7-2中，控制从第(24)行进入活动quicksort(1,9)，是整个程序执行过程中比较早的阶段，却几乎在整个程序执行过程的最后结束。同时还有其他几个quicksort活动，因此称quicksort为递归的。

一个递归的过程 p 不必直接调用自己；p 可以调用另一个过程 q，然后 q 通过一系列过程调用之后再调用 p。我们可以用一棵树（称之为*活动树*）来描绘控制进入和离开活动的方式。在活动树中：

1. 每个节点代表过程的一个活动。
2. 根节点代表主程序的活动。
3. 当且仅当控制流从活动 *a* 进入活动*b*时，节点 *a* 是节点 *b* 的父节点。
4. 当且仅当 *a* 的生存期先于 *b* 的生存期发生时，*a* 节点处于 *b* 节点的左边。

因为节点和活动一一对应，所以当控制在某节点所代表的活动中时，我们就直接说控制在这个节点。

例7.1 图7-3的活动树是从图7-2中的输出构造出来的[⊖]。为节省空间，每个过程仅用第一个字母代表。活动树的根是整个程序 sort。在 sort 执行期间，有一个 readarray 活动，由根的第一个子节点代表，标记为r。下一个活动由根的第二个子节点表示，是实参为1和9的 quicksort 的活动。在这个活动期间，在图7-1的第(16)~(18)行调用 partition 和 quicksort 而引起活动 p(1,9)，q(1,3)和 q(5,9)。注意，q(1,3)和q(5,9)的活动是递归的，它们在q(1,9)结束之前开始和结束。 □ 390 ~ 392

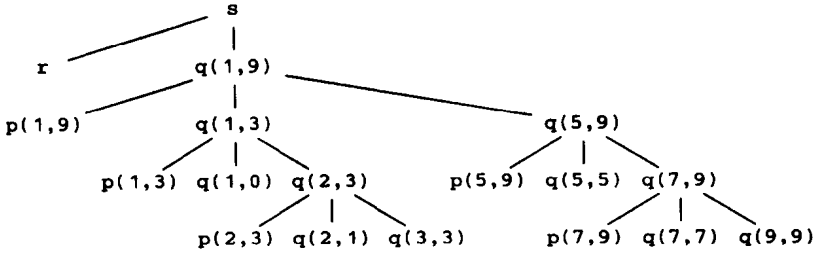


图7-3 图7-2中程序输出所对应的活动树

7.1.3 控制栈

程序的控制流对应于从活动树根节点开始的深度优先遍历，也就是在访问一个节点的子节点之前先访问该节点，然后再以从左到右的顺序递归地访问每个节点的子节点。图7-2中的输出结果就可以通过遍历图7-3中的活动树得到。第一次经过一个节点的时候打印 enter，在一个节点的整棵子树被访问之后打印 leave。

我们可以用*控制栈*来保存活跃着的过程活动。基本思想是：当活动开始时，把这个活动的节点压入控制栈；当这个活动结束时，弹出这个节点。控制栈的内容与到活动树的根节点的一条路径相关。当节点 *n* 在控制栈的栈顶时，栈内包含的是从节点 *n* 到根的路径上的节点。

例7.2 图7-4给出了控制进入 q(2,3)代表的活动时图7-3的活动树中那些曾经到达过的节点。标记为 r，p(1,9)，p(1,3)和 q(1,0)的活动已经执行完毕，所以图中用虚线连到这些节

⊖ quicksort所做的实际调用依赖于partition的返回值（参见Aho, Hopcroft, and Ullman[1983]中的算法细节）。图7-3描述了一棵可能的调用树。它与图7-2是一致的，虽然树中底层的一些调用在图7-2中没有给出。

点。实线标记从节点 q(2,3) 到根的路径。

此时，控制栈包含如下从该点到根的路径上的节点（栈顶的节点在右边）：

```
s, q(1,9), q(1,3), q(2,3)
```

并且除此之外没有其他节点。 □

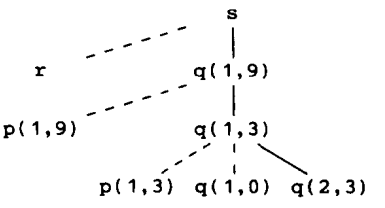


图7-4 控制栈包含到根的路径上的节点

我们可以把控制栈拓广到栈式存储分配技术，它可以用来实现像Pascal和C这样的语言。这种技术将在7.3节和7.4节中详细讨论。

7.1.4 声明的作用域

语言中的声明是把信息与名字联系起来的语法结构。声明可以是显式的，例如在 Pascal 程序段

```
var i : integer;
```

中。声明也可以是隐式的，例如在 Fortran 程序中，若无其他声明，以I开始的变量名代表整型变量。

在程序的不同部分可能有同一名字的互相独立的声明。当程序正文中出现一个名字时，语言的作用域规则确定应使用该名字的那一个声明。在图7-1的 Pascal 语言程序中，i分别在第(4)、(9)和(13)行声明了共3次，它在过程 readarray，partition 和 quicksort 中的使用是相互独立的。第(4)行的声明适用于第(6)行中对i的使用，即第(6)行中i的两次出现是在第(4)行的声明的作用域之内。第(16)~(18)行中i的3次出现是在第(13)行的声明的作用域之内。

一个声明起作用的那部分程序称为该声明的作用域。过程中一个名字如果出现在该过程的一个声明的作用域内，则称这个出现局部于该过程；否则，称为非局部的。局部和非局部名字的区分适用于任何包含声明的语法结构。

作用域是名字声明的一个性质。为简单起见，用简称“名字 x 的作用域”来代替“对名字 x 的这次出现起作用的 x 的声明的作用域”。这样，图7-1中第(17)行的 i 的作用域就是 quicksort 的过程体。[⊖]

编译时，符号表可用来寻找对一个名字的出现起作用的声明。看见一个声明时，就为它建立一个符号表表项。只要处于这个声明的作用域中，在查找到这个声明中的名字时，返回的总是这个表项。符号表将在7.6节讨论。

393 ~ 394

7.1.5 名字的绑定

即使每个名字在程序中只声明一次，同一个名字在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。

在程序设计语言的语义中，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的值的函数，如图7-5表示。使用第2章中左值和右值的术语，可以说环境把名字映射到左值，状态把左值映射到右值。



图7-5 从名字到值的两步映射

大部分时间，在不引起混淆的情况下，术语名字、标识符、变量和单词可以交替使用。

环境和状态是有区别的，赋值改变状态，但不改变环境。例如，如果变量 pi 的存储单元

⊖ 大部分时间，在不引起混淆的情况下，术语名字、标识符、变量和词素可以交替使用。

地址是100，其值是0，那么在赋值语句 pi := 3.14执行之后，pi 的存储单元地址仍然是100，但它保存的值变为3.14。

如果环境把存储单元 s 与名字 x 联系起来，我们说 x 绑定到 s，这个联系本身称为 x 的绑定。术语“存储单元”是象征性的，如果 x 不是基本类型，x 的存储单元 s 可能是一组存储字。

如图7-6所示，绑定是声明的动态对应物。我们已经看见，在某一时刻，递归过程可以有不止一个活动活跃着。在Pascal中，过程中的局部变量的名字在过程的不同活动中绑定到不同的存储单元。局部变量名字的绑定技术将在7.3节中考虑。

静态概念	动态对应物
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	绑定的生命期

图7-6 静态概念和动态概念的对应

395

7.1.6 一些问题

语言的编译器组织其存储区和绑定名字的方法，在很大程度上取决于对下面这些问题的回答：

1. 过程能递归吗?
2. 控制从过程的活动返回时，局部名字的值发生了怎样的变化?
3. 过程能引用非局部的名字吗?
4. 过程被调用时是怎样传递参数的?
5. 过程是否可以作为参数被传递?
6. 过程能否作为结果被返回?
7. 存储区能否在程序控制下动态地分配?
8. 存储区是否必须显式地释放?

这些问题对程序设计语言所需的运行时支持的影响将在本章的其余部分讨论。

7.2 存储组织

本节讨论的运行时存储组织的方式可用于 Fortran、Pascal 和 C 等语言。

7.2.1 运行时内存的划分

假定编译器从操作系统得到一块存储区，用于被编译过的程序的运行。根据上节的讨论，运行时该存储区可以划分成块，用以保存：

1. 产生的目标代码。
2. 数据对象。
3. 记录过程活动的控制栈的对应物。

产生的目标代码的长度在编译时即可确定，所以编译器可以把目标代码放在静态确定的区域中，可能是内存的低地址区。同样，某些数据对象的长度也可以在编译时知道，因此它们也可以放在静态确定的区域中，如图7-7所示。尽可能对数据对象进行静态分配的一个理由是，这些对象的地址可以编译到目标代码中。Fortran 语言的所有数据对象都可以静态分配。

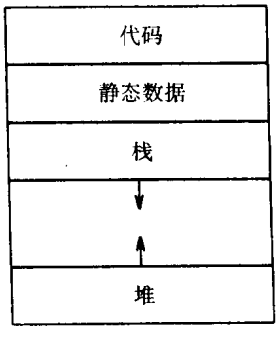


图7-7 将运行时内存划分成代码区和数据区的典型划分

Pascal 和 C 这样的语言的实现使用拓广的控制栈来管理过程的活动。当调用出现时，一个活动的执行被中断，有关机器状态的信息，如程序计数器和机器寄存器的值，就保存在这个栈

中。当控制从调用返回时，在恢复了有关寄存器的值和把程序计数器置到紧接该调用的下一个点后，该活动能够继续。生存期被包含在这个活动中的数据对象可以分配在这个栈中，与这个活动的其他有关信息放在一起。这种策略将在下节中讨论。

运行时内存的另一个单独区域叫做堆，它保存所有其他信息。如7.7节讨论的那样，Pascal允许数据在程序控制下分配，这种数据的存储空间可以分配在堆区。活动的生存期不能用活动树表示的语言的实现时可以用堆来保存活动的信息。数据放在栈上比放在堆上开销要小些，这是由它们对数据的分配和释放方式决定的。

程序执行时，栈的长度和堆的长度都会改变，所以在图7-7中把它们分放在内存的两端，需要时，它们向对方增长。Pascal 和 C 语言既需要运行时栈也需要堆，但不是所有的语言都这样。

通常，栈向下增长。也就是说，“栈顶”向页面的底部增长。因为内存地址随页面向下而增长，“向下增长”就意味着地址值增加。如果用 *top* 标记栈顶，距栈顶的偏移可以用 *top* 减去偏移来计算。在许多机器中，通过将 *top* 的值保存在寄存器中可以有效地完成该计算。因此，栈地址可以用距 *top* 的偏移来表示。[⊖]

396 ~ 397

7.2.2 活动记录

过程一次执行所需的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*，它由图7-8中所示的各个域组成。不是所有的语言，也不是所有的编译器都使用所有这些域，它们中的一个或多个域往往可以用寄存器取代。像Pascal和C这样的语言的习惯做法是，在过程被调用时把它的活动记录压入运行栈，在控制返回调用者时把这个活动记录从栈中弹出。

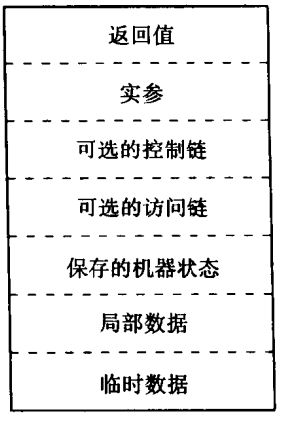


图7-8 一般的活动记录

活动记录的各个域的用途如下（从临时数据域开始）：

1. *临时数据域*。如计算表达式时出现的那些值，存放在临时数据域中。

2. *局部数据域*。保存局部于过程执行的数据，这个域的布局将在后面讨论。

3. *机器状态域*。保存过程调用前的机器状态信息，包括程序计数器的值和控制从这个过程返回时必须恢复的机器寄存器的值。

4. *可选的访问链*。7.4节中用它来引用存于其他活动记录中的非局部数据，Fortran这样的语言不需要访问链，因为非局部数据保存在固定的地方。Pascal语言需要访问链或者display表机制。

398

5. *可选的控制链*。用来指向调用者的活动记录。

6. *实在参数域*。用于存放调用过程提供给被调用过程的参数。我们把参数的空间放在活动记录中，但实际上常常用机器寄存器传递参数，以提高效率。

7. *返回值域*。用于存放被调用过程返回给调用过程的值。实际上，为提高效率，这个值也常用寄存器返回。

每个域的长度都可以在过程调用时确定。事实上，差不多所有域的长度都可以在编译时确

⊖ 图7-7中的存储组织假定运行时内存是由开始执行时获得的单个连续的存储块组成的。该假定使得组成栈和堆的存储空间的大小具有固定的限制。如果该限制足够大使之很少被超过，则对大部分程序来说都是一种浪费。如果将栈和堆中的目标链接起来，则维护栈顶的开销将变大。此外，不同的目标机器对存储区域的安排会有所不同。例如，某些机器允许将地址的正偏移放在寄存器中。

定。一个例外是，如果过程中有大小由实参值决定的局部数组，则只有运行到这个过程被调用时才能确定局部数据区域的大小。活动记录中可变长数据的分配将在7.3节讨论。

7.2.3 编译时的局部数据布局

假定运行时的存储空间是连续字节的存储块，其中字节是内存可编址的最小单位。在许多机器上，一个字节为8位，几个字节形成一个机器字。多字节对象存于连续的字节中，并以第一个字节的地址作为该对象的地址。

名字所需的存储空间的数量是由它的类型确定的。基本数据类型，如字符、整数或实数，通常可以用整数个字节存储。对于数组或记录这样的集合体，它的存储区必须大到足以存放它所有的成分。为便于访问它的成分，这种集合体的存储空间的典型分配是使用一块连续的字节区。我们将在8.2节和8.3节详细讨论。

局部数据域在编译过程中的声明时分配，长度可变的数据保存在这个域之外。我们记住为前面声明的变量分配的内存单元地址值，通过这个地址值就能够确定一个局部变量的*相对地址*，如相对于活动记录的开始点的相对地址。相对地址（或*偏移*）是数据对象地址和某个位置的地址差。

数据对象的存储布局深受目标机器的寻址约束的影响。例如，整数加法的指令可能需要整数*对齐*，即整数必须放在内存中特定的位置，如被4整除的地址。虽然10个字符的数组只需要足够存放10个字符的字节，但编译器很可能分配12个字节，其中两个字节不用。由于考虑到对齐而产生的无用空间叫做*填充空白区*。如果空间很宝贵，编译器可以*压缩*数据，使得没有任何填充空白区存在，但是运行时可能需要执行一些额外的指令来定位压缩数据，使得这些数据就像是对齐的一样以便正常操作。 399

例7.3 图7-9给出了由M1、M2两台机器上的C编译器使用的数据布局。C提供三种不同大小的整型，即短整型、整型和长整型，分别用关键字`short`、`int`、`long`声明。在M1中这些类型将分别分配16、32和32位存储空间，而在M2中将分配24、48和64位存储空间。图7-9中给出了两台机器数据类型分配位数的对比情况，尽管两台机器都不允许直接按位进行地址访问。

类型	大小（位）		对齐（位）	
	M1	M2	M1	M2
`char`	8	8	8	64①
`short`	16	24	16	64
`int`	32	48	32	64
`long`	32	64	32	64
`float`	32	64	32	64
`double`	64	128	32	64
字符指针	32	30	32	64
其他指针	32	24	32	64
结构	≥8	≥64	32	64

① 数组中的字符每8位对齐。

图7-9 用于两个C编译器的数据布局

M1的内存按每8位组成一个字节。虽然每个字节都有一个地址，但指令通常将短整型分配在地址为偶数的字节里，整型被分配在地址能被4整除的字节里。为了把短整型分配在地址为偶数的字节里，编译器甚至必须跳过一个字节作为填充空白区。但是这样一来，很可能把包含四个字节（32位）的一段区域分配给跟随在短整型后面的字符。

在M2中，每个字包含64位，用24位寻址。因为在一个字中有64个不同的位，所以需要用6

位附加位来区分一个字中的每个位。这样，一个指向字符的指针在M2中占用30位，24位用来寻找字，6位用来指出一个字符在一个字中的位置。

尽管一些类型不需要一个字那么大的空间，但M2指令集强烈的字倾向导致编译器每次分配一个完整的字，例如给只需要8位的字符类型分配一个字。因此在对齐规则下，图7-9中每一种类型都将占用64位。包含128位的双字可能会分配给一个字符和跟随其后的一个短整型，其
400 中，字符只占第一个字中的8位，而短整型只占第二个字的24位。 □

7.3 存储分配策略

下面三种存储分配策略可分别用于图7-7中的三种数据区域：

1. 静态分配策略。在编译时为所有数据对象分配存储单元。
2. 栈式分配策略。按栈方式管理运行时的存储空间。
3. 堆式分配策略。在运行时根据需要从堆数据区域分配和释放存储空间。

本节我们将这些分配策略用于活动记录，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。

7.3.1 静态存储分配

在静态分配中，名字在程序编译时与存储单元绑定，所以不需要运行时支撑程序包。因为运行时不改变绑定，所以每次过程活动时，它的名字都绑定到同样的存储单元。这种性质允许局部名字的值在过程停止活动后仍然保持，即当控制再次进入过程时，局部名字的值同控制上一次离开时一样。

根据名字的类型，编译器可以确定该名字所需的存储空间，如7.2节所讨论的那样。这个存储空间的地址由相对于该过程活动记录一端的偏移量表示。编译器最后必须确定活动记录在目标程序中的位置，如相对于目标代码的位置。一旦这一点确定下来，每个活动记录的位置以及活动记录中每个名字的存储位置就都固定了，所以编译时在目标代码中能填上所要操作的数据对象的地址。同样，过程调用时保存信息的地址在编译时也是已知的。

然而，仅仅使用静态分配会带来局限性。

1. 数据对象的长度和它在内存中位置的约束在编译时必须知道。

2. 不允许递归过程，因为一个过程的所有活动使用同样的局部名字绑定。

3. 数据结构不能动态建立，因为没有运行时的存储分配机制。

Fortran语言允许静态存储分配。Fortran程序由主程序、子例程和函数（它们都称为过程）组成，如图7-10的Fortran77

```
(1)      PROGRAM CNSUME
(2)         CHARACTER * 50 BUF
(3)         INTEGER NEXT
(4)         CHARACTER C, PRDUCE
(5)         DATA NEXT /1/, BUF /' '/
(6)  6         C = PRDUCE()
(7)            BUF(NEXT:NEXT) = C
(8)            NEXT = NEXT + 1
(9)            IF ( C .NE. ' ' ) GOTO 6
(10)           WRITE (*,'(A)') BUF
(11)           END

(12)     CHARACTER FUNCTION PRDUCE()
(13)        CHARACTER * 80 BUFFER
(14)        INTEGER NEXT
(15)        SAVE BUFFER, NEXT
(16)        DATA NEXT /81/
(17)           IF ( NEXT .GT. 80 ) THEN
(18)              READ  (*,'(A)') BUFFER
(19)              NEXT = 1
(20)           END IF
(21)           PRDUCE = BUFFER(NEXT:NEXT)
(22)           NEXT = NEXT+1
(23)           END
```

图7-10 一个Fortran 77程序

程序所示。使用图7-7中的内存组织，这个程序的代码和活动记录的布局如图7-11所示。在CNSUME的活动记录中，有局部变量BUF、NEXT和C的空间。名字BUF与保存了50个字符的存储单元绑定，接着是存放NEXT对应的整数值和C对应的字符值的空间。PRDUCE中也有NEXT的声明，但这不会引起问题，因为分别局部于这两个过程的NEXT在各自过程的活动记录中得到空间。

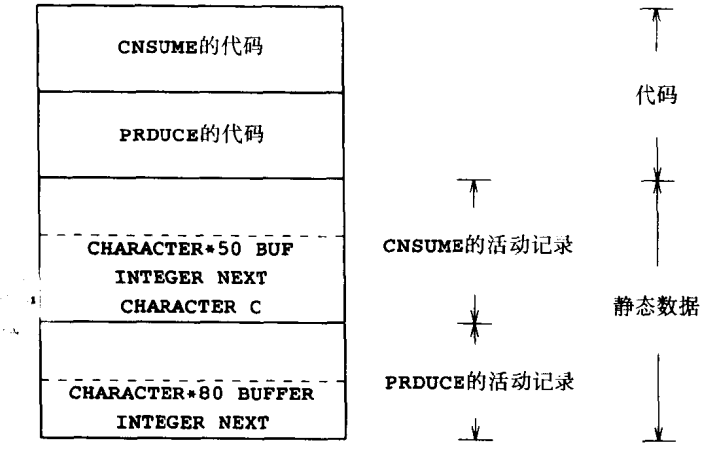


图7-11 一个Fortran 77程序局部标识符的静态存储

因为可执行代码的长度和活动记录的长度在编译时已知，除了图7-11的方式外，也可以使用其他的存储组织方式。Fortran编译器可以把每个过程的活动记录和该过程的代码放在一起。在某些计算机系统中，还可以不指定活动记录的相对位置，而让链接编辑程序去连接活动记录和可执行代码。

例7.4 图7-10中程序的结果取决于过程活动后被保留的局部量的值。Fortran77的SAVE语句指出，一个活动开始时的局部量的值必须与上一个活动结束时的值一样，这些局部量的初值可以用DATA语句指定。

过程PRDUCE中第(18)行的语句每次读一行正文到缓冲区，该过程的每次活动递交后继字符。主程序CNSUME也有缓冲区，它累积字符直至看见空格为止。当输入是

```
hello world
```

时，由PRDUCE的活动返回的字符描绘在图7-12中。程序的输出是

```
hello
```

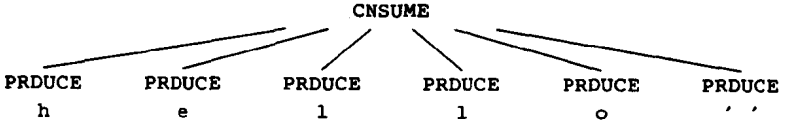


图7-12 PRDUCE的活动所返回的字符

PRDUCE存放一行正文的缓冲区在该过程的各个活动之间必须维持它的值。第(15)行的SAVE语句保证当控制回到PRDUCE时，局部变量BUFFER和NEXT的值和控制上一次离开该过程时的值一样。控制第一次到达PRDUCE时，局部变量NEXT的值取自第(16)行的DATA语句，即NEXT的初值是81。 □

401 ~ 403

7.3.2 栈式存储分配

栈式存储分配是基于控制栈的思想；把存储空间组织为栈，而且随着过程活动的开始和结束将活动记录进栈和出栈。过程每次调用时，局部量的存储空间包含在该次调用的活动记录中，由于每次调用都引起新的活动记录进栈，所以每次活动时局部量都绑定到新的存储单元。而且，因为活动记录弹出栈时局部量的存储空间被释放，所以，活动结束时局部量的值被删除。

我们首先描述所有活动记录的长度在编译时都可知的栈式存储分配形式，编译时只能获得不完整的长度信息的情况在后面讨论。

假定寄存器 *top* 标记栈顶，运行时活动记录的分配和释放分别由增大和减少 *top* 来完成。如果过程 q 的活动记录长度是 *a*，那么在 q 的代码执行前把 *top* 增加 *a*，控制从过程 q 返回时将 *top* 减少 *a*。

例7.5 图7-13表示当控制流通过如图7-3所示的活动树时，活动记录压入运行栈和从栈中弹出的情况，树中的虚线引向已经结束的活动。执行开始时有过程s的活动，当控制到达s中的第一个调用时，过程r被激活，它的活动记录压入栈；当控制从这个活动返回时，这个活动记录弹出，栈中仅留下s的活动记录。在s的活动中，控制到达以1和9为实参的过程q的调用，

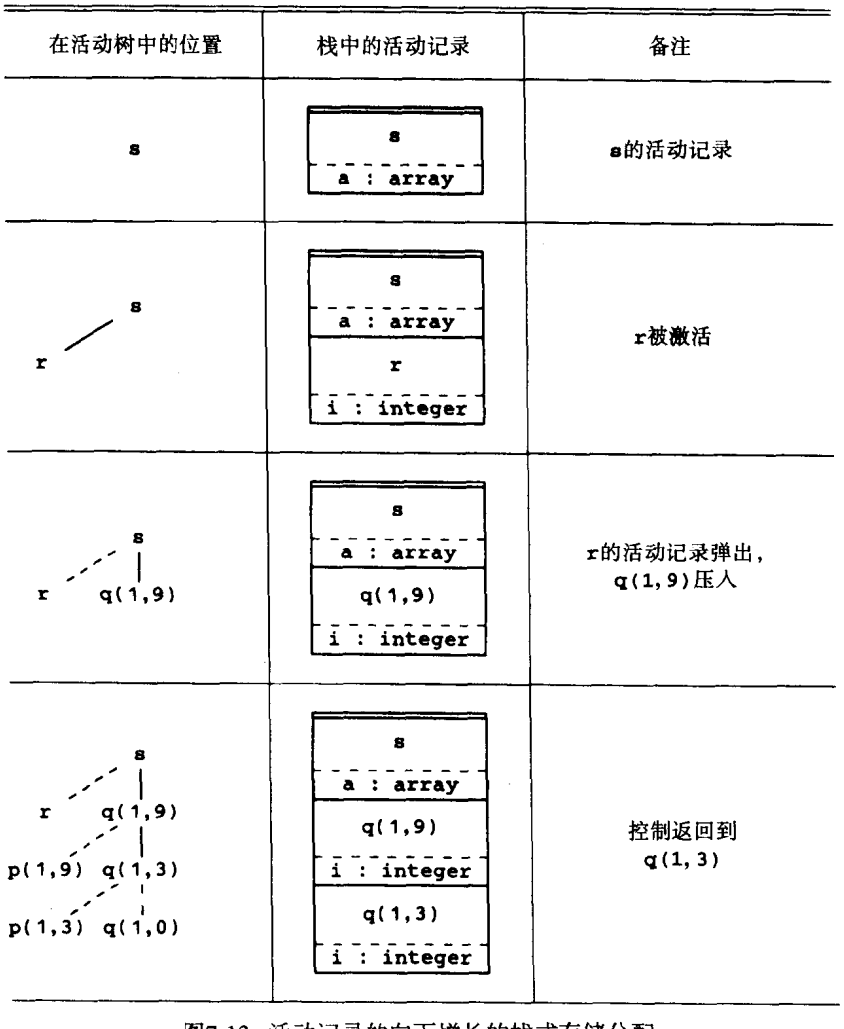


图7-13 活动记录的向下增长的栈式存储分配

q的这个活动的活动记录分配在栈顶。只要控制在这个活动中，它的活动记录就在栈顶。

图7-13中最后两幅图之间有几个活动出现。在最后一幅图中，活动 p(1,3)和 q(1,0)都在q(1,3)的生存期里开始并已经结束，所以它们的活动记录已经压入和弹出堆栈，留下q(1,3)的活动记录在栈顶。 □

在Pascal的过程中，像7.2节所讨论的那样，我们能够确定局部数据在活动记录中的相对地址。运行时，假定 *top* 指示活动记录的末端，该过程目标代码中局部名字 x 的地址可以写成 *dx*(*top*)，它表示绑定到 x 的数据可以在 *top* + *dx* 的位置找到。当然，也可以用指向活动记录中一个固定点的任意其他寄存器的值加（减）某个偏移来计算地址。

7.3.2.1 调用序列

过程调用是通过在目标代码中生成*调用序列*来实现的。*调用序列*分配活动记录，并把信息填入它的域中；*返回序列*恢复机器状态，使调用过程能够继续执行。

即使是实现同一种语言，调用序列和活动记录也可能不同。调用序列的代码常常分成两部分，分别处于调用过程和被调用过程中。过程调用序列在调用过程和被调用过程间无准确的划分，源语言、目标机器和操作系统强加的限制可能使一种办法比另一种办法更合适。[⊖] 404 ~ 405

有助于设计调用序列和活动记录的一个原则是，长度能较早确定的域放在活动记录的中间。在图7-8的一般活动记录中，控制链、访问链和机器状态域出现在中间。是否使用控制链和访问链取决于编译器的设计，因此这些域可以在构造编译器时固定。如果对于每个活动，要保存的机器状态信息量完全相同，那么可以用同样的代码来执行各个活动的保存和恢复。而且，当出现错误时，调试器将很容易辨认栈的内容。

即使临时数据域的长度可以在编译时最终确定，但就前端而言，这个域的大小也可能是未知的，因为代码生成或优化可能缩减过程所需的临时数据区。在活动记录中，一般把临时数据域放在局部数据域的后面，它的长度的改变不会影响数据对象相对于中间那些域的位置。

因为每个调用都有它自己的实参，调用者通常计算实参，并把它们传到被调用者的活动记录。参数传递方式将在7.5节中讨论。在运行栈中，调用者的活动记录刚好处于被调用者的下面，如图7-14所示，因此把参数域和可能有的返回值域放在紧靠调用者活动记录的地方是有好处的。调用者可以根据它自己活动记录的末端的偏移来访问这些域，无须知道被调用者的活动记录的整个布局。尤其是，对调用者来说，根本没有必要知道被调用者的局部数据和临时数据。这种信息隐蔽的另一好处是能够处理参数个数可变的过程，如C语言的标准库函数printf，这个问题将在后面讨论。

Pascal语言要求在编译时就能够确定过程的局部数组的长度。但是，局部数组的长度常常依赖于传递给该过程的参数值。在这种情况下，过程的所有局部数据的大小只有在过程调用时才能确定。处理变长数据的技术本节稍后讨论。

下面给出的调用序列受到上面讨论的启发。如图7-14所示，寄存器*top_sp*指向活动记录中机器状态域的末端，该位置对调用者是已知的，因此在控制转移到被调用过程之前它用来置*top_sp*的值。被调用过程的代码可以使用相对于*top_sp*的偏移量来访问它的临时变量和局部数据。其调用序列是： 406

1. 调用者计算实参。

⊖ 如果一个过程被调用*n*次，则不同调用者中调用序列的一部分可能被生成*n*次。但是，被调用者中的这部分可以被所有的调用所共享，所以它只被生成一次。因此，要求将调用序列中尽可能多的内容放到被调用者中。

2. 调用者把返回地址和 *top_sp* 的旧值存入被调用者的活动记录中。调用者然后将 *top_sp* 的值增加到图7-14所示的位置，即 *top_sp* 移过调用者的局部数据域和临时变量域以及被调用者的参数域和状态域。

3. 被调用者保存寄存器值和其他机器状态信息。

4. 被调用者初始化其局部数据，并开始执行。

一个可能的返回序列如下：

1. 被调用者将返回值放入邻近调用者的活动记录的地方。

2. 利用状态域中的信息，被调用者恢复 *top_sp* 和其他寄存器，并且按调用者代码中的返回地址返回。

3. 尽管 *top_sp* 的值减小了，但调用者可以将返回值复制到自己的活动记录中，并用它来计算一个表达式。

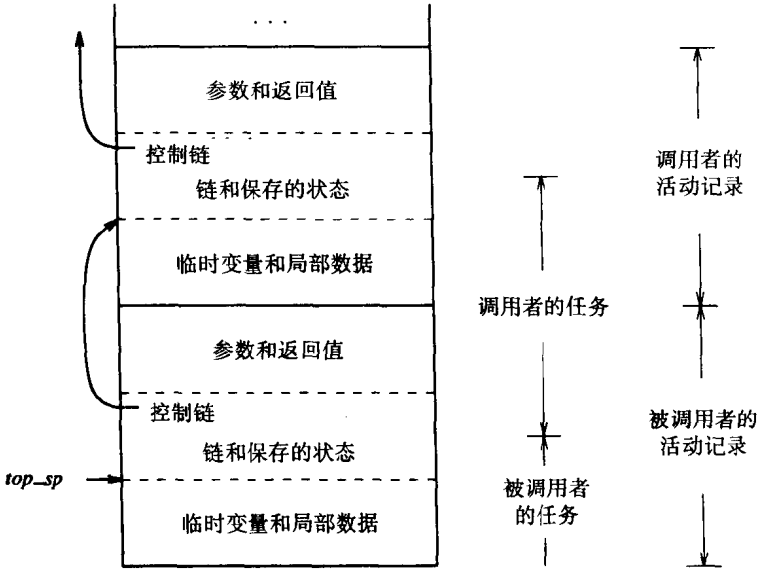


图7-14 调用者和被调用者之间的任务划分

407 上面的调用序列使被调用的过程的参数个数依赖于调用。注意，编译时，调用者的目标代码知道它提供给被调用者的参数个数，所以调用者知道参数域的大小。但是，被调用者目标代码必须准备应付参数个数不同的各种调用，所以直到它被调用时才检查参数域。使用图7-14所示的存储组织形式，描述参数的信息放在与状态域相邻的地方，这样，被调用者才能够找到它。例如，考虑C语言的标准库函数 `printf`，它的第一个参数指出了其余参数的性质，所以一旦 `printf` 能够确定第一个参数的位置，就能够找到其余的参数。

7.3.2.2 变长数据

图7-15给出了一种处理变长数据的常用策略，其中过程 `p` 有三个局部数组。`p` 的活动记录中并不存储这些数组，只保存指向每个数组起始位置的指针。相关指针的地址在编译时是已知的，这样目标代码可以通过指针访问数组元素。

图7-15中还显示了 `p` 所调用的过程 `q`。`q` 的活动记录在 `p` 的数组后面，然后是 `q` 的变长数组。

通过两个指针 *top* 和 *top_sp* 访问栈中的数据。第一个指针指出实际的栈顶，它指向要存放下一个活动记录的开始位置。第二个指针用于找出局部数据。为了与图7-14的组织形式一致，

假定 *top_sp* 指向机器状态域的末端。在图7-15中，*top_sp* 指向 q 的活动记录的机器状态域的末端。在这个域内，有一个指向当控制在 p 的活动中时的 *top_sp* 值的控制链。

使用活动记录中这两个域的大小可以在编译时生成保存 *top* 和 *top_sp* 的代码。当 q 返回时，*top* 的值就是 *top_sp* 减去 q 的活动记录中机器状态域和参数域的长度。至少对调用者来说，该长度在编译时是已知的。调整 *top* 以后，*top_sp* 的新值可以从 q 的控制链中复制过来。

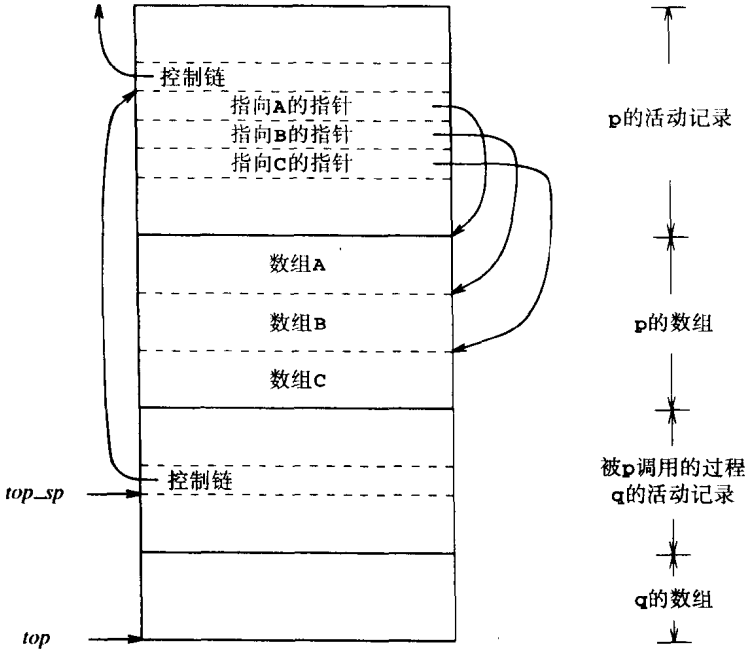


图7-15 访问动态分配的数组

7.3.3 悬空引用

只要存储空间可以释放，悬空引用问题就会出现。引用某个已经释放的存储单元就叫做悬空引用。使用悬空引用是逻辑错误，因为按大多数语言的语义，被释放的存储单元的值是没有定义的。更糟糕的是，因为这种存储单元后来可能又分配给其他数据，使得在有悬空引用的程序中会出现不可预期的错误。

例7.6 图7-16中C语言程序的过程 dangle 返回指向绑定到局部名 i 的存储单元的指针。这个指针由算符&作用于 i 来产生。当控制从 dangle 返回到 main 时，局部变量的存储空间已经释放，并可以另作他用。因为 main 中的 p 引用这个存储单元，所以p的使用是悬空引用。 □

```
main()
{
    int *p;
    p = dangle();
}
int *dangle()
{
    int i = 23;
    return &i;
}
```

图7-16 指针 p 指向已释放空间的 C 程序

7.7节的一个例子包含了程序控制下的空间释放。

7.3.4 堆式存储分配

上面讨论的栈式存储分配策略在下列情况下不能使用：

1. 活动结束时必须保持局部名字的值。
2. 被调用者的活动比调用者的活动的生存期长。这种现象在使用活动树来正确描绘过程间

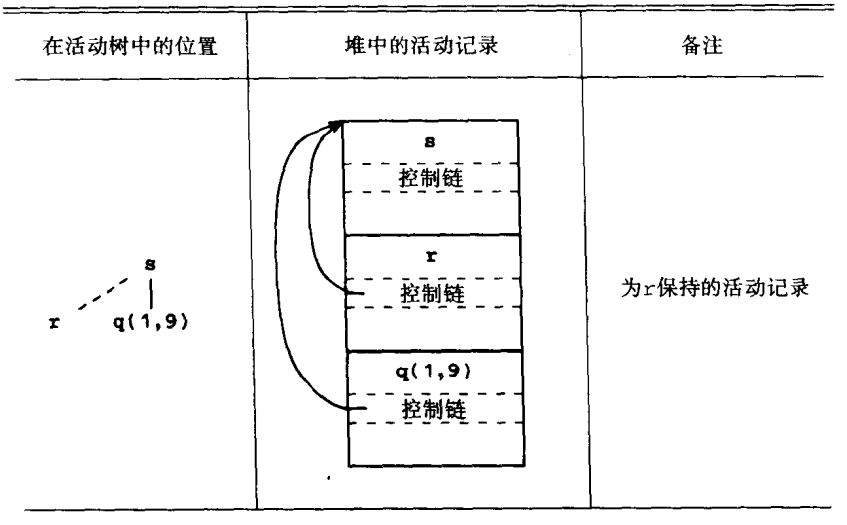
408
~
409

的控制流的语言中是不能发生的。

在上面任何一种情况下，活动记录的释放都不能使用后进先出的形式，所以存储空间不能组织成栈。

堆式存储分配是把连续存储区域分成块，当活动记录或其他对象需要时就分配。块的释放可以按任意次序进行，所以经过一段时间后，堆可能包含交错的正在使用的和已经释放的区域。

活动记录的堆式存储分配和栈式存储分配的区别可以从图7-17和图7-13中看出。在图7-17中，过程 r 的活动记录在该活动结束后仍然保持，新的活动 q(1, 9)的活动记录不能像图7-13中那样在物理上跟随 s 的活动记录。现在，如果r的活动记录被释放，那么在此堆中，s 和q(1, 9)活动记录间有一块空闲区，由堆管理器去重新使用这个空间。



410

图7-17 堆中活动记录不必相邻

堆管理的效率问题是数据结构理论中一个比较特殊的问题，7.8节将再次介绍。通常使用堆管理器会有一些时间和空间开销。当活动记录较小或其大小可预知时，为提高效率可按如下的特殊方式处理：

1. 对每一个感兴趣的活动记录大小，保存一个相应大小的空闲块的链表。

2. 可能的话，为大小为s的请求分配一个大小为s'的块，其中s'是大于等于s的最小块。当该块最终被释放后，将其链回原来的空闲块链表。

3. 对于大块存储空间，使用堆管理器管理。

这种方法将导致对较小存储空间的快速分配与释放，因为从指针列表中获取和返回块的操作是高效的。对于多数存储空间来说，需要相当的计算时间才能耗尽，因此与执行计算所花的时间相比，存储分配所花的时间是可以忽略的。

7.4 对非局部名字的访问

上一节的存储分配策略适用于本节讨论的非局部名字的访问，虽然讨论是基于活动记录的栈式分配的，但同样的思想亦可用于堆式分配。

语言的作用域规则确定了如何处理非局部名字的访问。一种常用的规则称为*词法作用域规则*或*静态作用域规则*。它仅仅根据程序正文即可确定用于名字的声明。许多语言，如Pascal、C和Ada，都使用词法作用域规则，并加上下面所讨论的“最近嵌套”规则。另一种规则叫做

动态作用域规则，它在运行时根据当前的活动来决定用于名字的声明。Lisp、APL 和 Snobol 是使用动态作用域的语言。

我们首先从程序块和“最近嵌套”规则开始讨论，然后考虑在像C语言这样的语言中用到的非局部名字。C 语言使用的是词法作用域规则，所有的非局部名字绑定到静态分配的存储单元中，不允许嵌套的过程说明。

在像 Pascal 这样的语言中，它们有嵌套过程和词法作用域，属于不同过程的名字可能是某一个时刻的环境的一部分。寻找包含绑定到非局部名字的存储单元的活动记录有两种办法：访问链和 display。我们将分别讨论。

最后一节讨论动态作用域的实现。

7.4.1 程序块

程序块本身是含有局部数据声明的语句。程序块的概念起源于 Algol 语言，在C语言中，程序块的语法是：{声明语句}。 411

程序块的一个特点是它的嵌套结构。分界符标记程序块的开始和结束。C 语言用括号“{”和“}”来作为分界符，而Algol语言的传统是用 begin 和 end 作为分界符。分界符保证程序块不是互相独立便是一个嵌在另一个里面，不可能出现程序块 B_1 和程序块 B_2 交叠，即 B_1 先于 B_2 开始又先于 B_2 结束的情形。这种嵌套性有时又称作*程序块结构*。

程序块语言中声明的作用域由下面的*最近嵌套规则*给出：

1. 程序块 B 中声明的作用域包括 B。

2. 如果名字 x 没有在 B 中声明，那么 B 中 x 的出现是在外围程序块 B' 对 x 的声明的作用域中，且满足：i）B' 有 x 的声明；并且 ii）B' 比其他任何包含 x 的声明的程序块更接近被嵌套的 B。

图7-18中的每个声明给被声明名字置的初值是它所在的程序块编号。B_0 中 b 声明的作用域不包括 B_1，而是图中的 $B_0 \sim B_1$，因为在 B_1 中 b 被重新声明了。这样的间隙在声明的作用域中称为*洞*（hole）。

图7-18中程序的输出反映了最近嵌套作用域规则。在源程序正文中，控制流从恰好在块之前的点进入程序块，从恰好在块之后的点离开程序块。因此，打印语句按照 B_2，B_3，B_1 和 B_0 的次序执行，这正是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
```

程序块结构可以用栈式分配来实现。因为声明的作用域决不会超出它所在的程序块。所以被声明名字的存储空间可以在控制进入程序块时分配，在控制离开程序块时释放。这种观点把程序块看成无参过程，它仅在程序块前面的点被调用，并且仅返回到该程序块后面的点。程序块的非局部环境的维护可以用本节后面讨论的用于过程的技术，不过程序块比过程简单，因为没有参数传递，并且控制只能沿静态程序正文进入和退出程序块。[⊖] 412

另一种实现是每次为一个完整的过程体分配存储空间。如果在过程中有程序块，则要为这

⊖ 从一个程序块到外围程序块的转移可以通过将外围块的活动记录弹出栈来实现。某些语言允许跳转到某个程序块的转移。在控制按这种方式转移之前，需要为源程序块建立活动记录。至于怎样初始化这些活动记录中的局部数据，要由语言的语义来确定。

些程序块中的声明留出所需的存储空间。对于图7-18中程序块 B_0，可以如图7-19所示分配存储空间。局部量 a 和 b 的下标用来标识声明它们的程序块。注意，可以为 a_2 和 b_3 分配同一个存储单元，因为它们所在的程序块不会同时都是活跃的。

```
main()
{
    int a = 0;
    int b = 0;
    {
        int b = 1;
        {
            int a = 2;
            printf("%d %d\n", a, b);
        }
        {
            int b = 3;
            printf("%d %d\n", a, b);
        }
        printf("%d %d\n", a, b);
    }
    printf("%d %d\n", a, b);
}
```

B_0 B_1 B_2 B_3

声明	作用域
int a = 0;	$B_0 \sim B_2$
int b = 0;	$B_0 \sim B_1$
int b = 1;	$B_1 \sim B_3$
int a = 2;	B_2
int b = 3;	B_3

图7-18 一个 C 语言程序中的程序块

当没有变长数据时，一个程序块的执行所需要的最大存储空间可以在编译时确定。（变长数据可以用像7.3节那样的指针来处理。）在确定其存储空间时，我们保守地假定程序中所有的控制路径都会被真正地执行到。也就是说，我们假定条件语句的 then 和 else 部分都将被执行到，而且 while 循环中的所有语句都将被执行到。

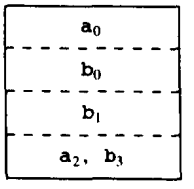


图7-19 图7-18中所声明名字的存储

413

7.4.2 无嵌套过程的词法作用域

由于C语言中过程的定义不能嵌套，所以我们下面讨论的C语言的词法作用域规则比Pascal语言的词法作用域规则简单，也就是说C语言中过程的定义不能出现在另一个过程中。如图7-20所示，一段C程序由一系列的变量声明和过程（C语言中称为函数）组成。如果某个函数中出现了对名字 a 的非局部引用，则a必须在所有函数之外声明。在函数之外的声明的作用域包含声明后面的函数体，除非它在某个函数中重新被声明。在图7-20中，在 readarray，partition 和 main 中出现的非局部名字 a 指的是第一行中声明的数组。

```
(1) int a[11];
(2) readarray() { ... a ... }
(3) int partition(y,z) int y, z; { ... a ... }
(4) quicksort(m,n) int m, n; { ... }
(5) main() { ... a ... }
```

图7-20 含有对a的非局部引用的C语言程序

在没有嵌套过程的情况下，7.3节中对局部名字的栈式存储分配策略可以直接用于C这样的词法作用域语言。在任何过程之外声明的所有名字的存储空间可以静态地分配。这些存储空间的位置在编译时是已知的，所以如果一个名字在过程体内是非局部的，则只需使用静态分配的地址空间。任何其他的名字都必须是栈顶活动的局部名字，可通过*top*指针访问。但是该方法不能用于嵌套过程，因为一个非局部名字可能指向一个处于堆栈深处的数据，下面我们将对此进行讨论。

非局部名字的静态分配的一个重要的优点在于声明的过程可以自由地作为参数进行传递和作为结果返回（C语言中函数的传递是通过指向函数的指针进行的）。因为是词法作用域，而且不包含嵌套过程，因此一个过程的非局部名字对任何过程也是非局部的。它的静态地址可以被所有过程使用，无论它是否是活动的过程。类似地，如果过程作为结果返回，该过程中的非局部变量直接指向为它们分配的静态存储空间。

例如，考虑图7-21中的 Pascal 程序。名字 m 的所有出现，在图7-21中用圆圈标识，都在第(2)行中声明的作用域内。由于 m 对程序中所有的过程都是非局部的，所以它的存储空间可以静态分配。过程 f 和 g 无论何时运行，都可以通过静态的地址来访问 m。f 和 g 作为参数传递这一事实只影响它们处于活跃状态的时间，但不会影响它们对 m 值的访问方式。 414

更详细地说，第(11)行的调用 b(f)将函数 f 和形参 h 联系起来。当第(8)行 write(h(2))中调用形参 h 时，函数 f 被激活。f 的活动将返回2，因为非局部变量 m 的值是0而形参 n 的值为2。接着向下执行，调用 b(g) 将 g 与 h 联系起来，这时，对 h 的调用将激活 g。程序的输出为

```
2          0
```

7.4.3 包含嵌套过程的词法作用域

在Pascal语言中，如果一个过程对名字 a 进行非局部引用，则 a 应属于静态程序代码中离a最近的嵌套声明的作用域。

```
(1)  program pass(input, output);
(2)      var (m): integer;
(3)      function f(n : integer) : integer;
(4)          begin f := (m)+ n end { f };
(5)      function g(n : integer) : integer;
(6)          begin g := (m)* n end { g };
(7)      procedure b(function h(n : integer) : integer);
(8)          begin write(h(2)) end { b };
(9)      begin
(10)         (m):= 0;
(11)         b(f); b(g); writeln
(12)     end.
```

图7-21 带有 m 的非局部出现的 Pascal 程序

图7-22中 Pascal 程序的嵌套过程定义表示为如下的缩进形式：

```
sort
    readarray
    exchange
    quicksort
        partition
```

图7-22中第(15)行出现的 a 处于嵌套在过程 quicksort 中的 partition 函数。对 a 的最近嵌套声明在第(2)行，其作用域包括整个程序。最近嵌套规则对过程名同样适用。在第(17)行中被 partition 调用的过程 exchange 对于 partition 而言是非局部的。使用该规则，我们首先检查 exchange 是否在 quicksort 中定义；因为没定义，所以我们在主程序sort中寻找它的定义。 415

```
(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7-22 一个带有嵌套过程的 Pascal 程序

7.4.3.1 嵌套深度

下面用过程的嵌套深度这一概念来实现词法作用域。令主程序名的嵌套深度为1，从一个包围过程进入一个被包围过程时嵌套深度加1。在图7-22中，第(11)行的过程quicksort的嵌套深度为2，而第(13)行的partition过程的嵌套深度为3。对于每个出现的名字，我们将它和它所在的过程的嵌套深度联系在一起。在第(15)~(17)行partition过程中的a、v和i的出现分别具有嵌套深度1，2和3。

7.4.3.2 访问链

嵌套过程的词法作用域的一种直接实现方式是为每个活动记录增加一个称为访问链的指
416 针。如果在源代码中过程 p 直接嵌套在过程 q 中，则 p 的活动记录的访问链指向 q 的最近一次活动的活动记录的访问链。

图7-22中的程序执行时，运行时刻栈的情况如图7-23所示。再强调一次，为了节省空间，图中只显示了每个过程名字的第一个字母。因为没有包含 sort 的过程，所以 sort 的活动记录中访问链为空。quicksort 的每个活动记录的访问链都指向 sort。注意，在图7-23c中，partition(1,3)的活动记录的访问链指向最近的 quicksort 的活动记录，即quicksort (1,3)。

假设过程 p 的嵌套深度为 n_p，p 中引用了非局部的 a，其中 a 的嵌套深度为 n_a，$n_a \leqslant n_p$。可以按以下的步骤找到 a 的存储位置。

1. 当控制在 p 中时，p 的活动记录在栈顶，下面是 $n_p - n_a$ 个访问链。$n_p - n_a$ 的值可以在编译时预先计算得到。如果一个活动记录中的访问链指向另一个活动记录中的访问链，则可以通过一个间接操作来跟踪该访问链。

2. 经过 $n_p - n_a$ 个访问链，到达包含 a 这一局部名的过程的活动记录。如上一节所讨论的，a的位置用活动记录中相对于某个位置的一个固定偏移量表示。特别地，偏移量也可以是相对

于访问链的。

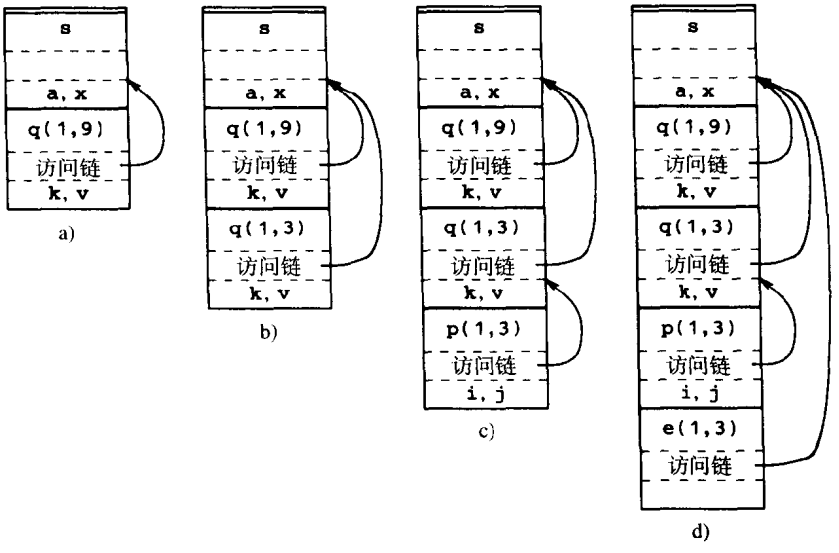


图7-23 寻找非局部名字存储位置的访问链

因此，过程 p 中非局部名 a 的地址可以由以下序对给出，它的值是在编译时计算的，并存放在符号表中：

（n_p-n_a，包含 a 的活动记录中的偏移量）

上式的第一部分给出了需要遍历的访问链的数量。

例如，图7-22中第(15)和(16)行嵌套深度为3的过程 partition 引用了嵌套深度分别为1和2的非局部的 a 和 v。从 partition 的活动记录开始，分别沿着 3-1 = 2和3-2 = 1个访问链就可以找到包含这些非局部名字的活动记录。

用于建立访问链的代码属于调用序列的一部分。假设嵌套深度为 n_p 的过程 p 调用嵌套深度为 n_x 的过程 x，用于建立被调用过程访问链的代码依赖于被调用过程是否嵌套在调用过程中。

1. $n_p < n_x$ 的情况。由于被调用过程 x 比 p 处于更深层的嵌套，它必须在 p 中定义，否则它对于p来说将是不可访问的。在图7-23a中过程 sort 调用过程 quicksort 以及图7-23c中过程 quicksort 调用过程 partition 都会发生这种情况。此时，被调用过程的访问链必须指向栈中刚好在下方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，调用和被调用过程的嵌套深度为1, 2, …, n_x -1 的外围过程必须相同。例如，图7-23b中 quicksort 调用它自身和图7-23d中 partition 调用 exchange。从调用者开始，沿着 $n_p - n_x + 1$ 个访问链，就可以到达包围调用和被调用过程并离它们最近的过程的当前活动记录。这时到达的访问链就是被调用过程中的访问链所指向的访问链。同样，$n_p - n_x + 1$ 可以在编译时计算。

7.4.3.3 过程的参数

即使嵌套过程作为参数传递，词法作用域规则仍然适用。图7-24中的 Pascal 程序的第(6)和(7)行的函数 f中包含一个非局部的 m；程序中所有 m 的出现都用圆圈标示出来。在第(8)行，过程c将0赋给 m，然后将f作为参数传递给 b。注意，第(5)行中声明的 m 的作用域并不包括第(2)行和第(3)行的 b 的过程体。

```
(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 ⓜ: integer;
(6)        function f(n : integer) : integer;
(7)           begin f := ⓜ+ n end { f };
(8)        begin ⓜ:= 0; b(f) end { c };
(9)     begin
(10)     c
(11)    end.
```

图7-24 一个必须用实参 f 传递的访问链

在 b 的过程体内，语句 writeln(h(2))将激活 f，因为形参 h 引用了 f。也就是说，writeln 将打印调用f(2)的结果。

如何建立 f 的活动记录的访问链呢？答案如图7-25所示，一个作为参数传递的嵌套过程必须和它的访问链一起被传递。当过程 c 传递参数 f 时，就像调用 f 一样为 f 建立一个访问链，这个访问链同f一起传给 b。然后，当 f 在 b 中被激活时，这个访问链被用来建立 f 的活动记录的访问链。

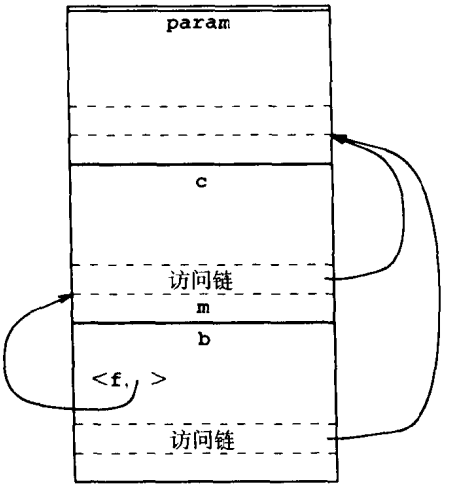


图7-25 实参f携带其访问链

7.4.3.4 display表

通过使用称之为 display 表的一个指向活动记录的指针数组 *d*，可以实现比使用访问链要快的对非局部名字的访问。我们维护 display 表使得嵌套深度为 *i* 的非局部名字 a 所在的活动记录就是 display 表中 *d*[*i*] 元素所指向的活动记录。

假设当前控制处于嵌套深度为 *j* 的过程 p 的活动记录中。那么，display 表中的前 *j*-1个元素指向按词法作用域规则包围过程p的那些过程的最新的活动记录，而 *d*[*j*] 则指向 p 的活动记录。一般而言，使用 display 表要比沿着访问链搜索快，因为包含非局部名字的活动记录是通过访问 *d* 的一个元素找到的，仅仅经过了一个指针。

维护 display 表的一种简单方法是在使用 display 表的同时使用访问链。作为调用序列和返回序列的一部分，通过沿着访问链访问来更新 display 表。当一个嵌套深度为 *n* 的活动记录被跟踪，display 表的元素 *d*[*n*]被设置为指向该活动记录的指针。实际上，display 表复制了访问链中的信息。

以上的简单方法可以改进如下。如果过程没有作为参数传递，一般情况下图7-26中的方法在过程进入和退出时需要较少的工作。在图7-26中，display 表包含一个独立于栈的全局数组。它给出了图7-22中程序执行时的情况，仍然用过程名字中的第一个字母代替过程名。

图7-26a为活动q(1,3)开始前的情况。由于quicksort的嵌套深度为2，当一个新的活动 quicksort开始时，display 表的元素 *d*[2]被改变，这种改变如图7-26b所示。*d*[2] 现在指向新

的活动记录，$d[2]$ 原有的值在新的活动记录中被保存[⊖]。以后当控制交回到 q(1,9)时，被保存的 $d[2]$ 的值将被用来恢复图7-26a中 display 表的状态。

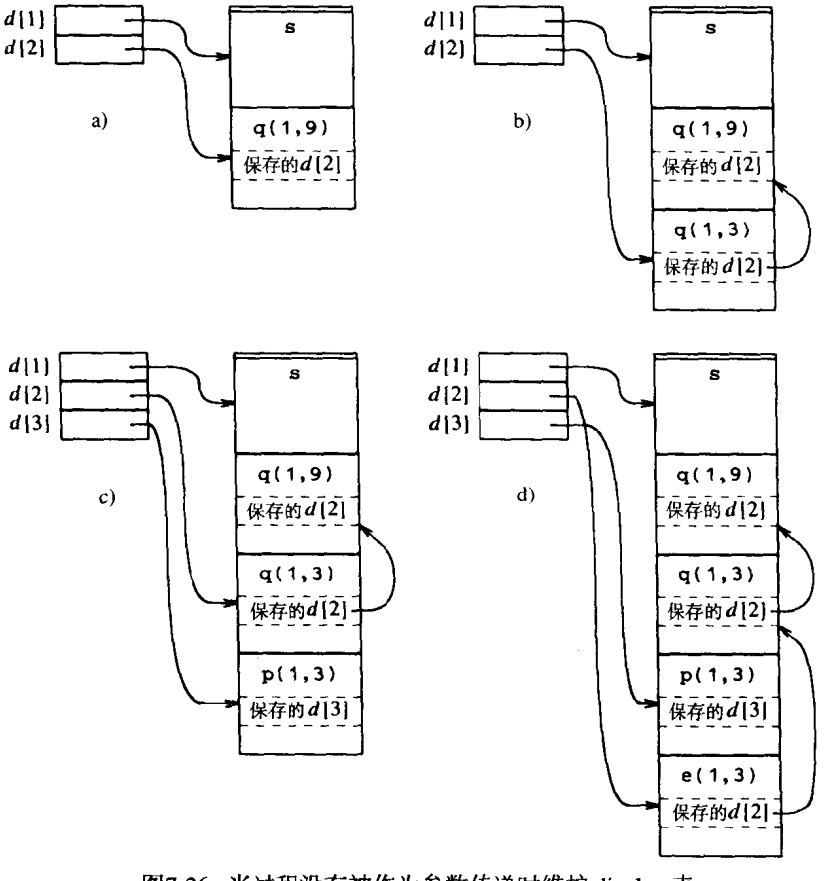


图7-26 当过程没有被作为参数传递时维护 display 表

当一个新的活动开始时，display 表被改变，同时，当控制权从新的活动中返回时，display 表的状态必须被恢复。Pascal 等词法作用域语言的作用域规则允许 display 表通过以下步骤维护。我们只讨论过程不作为参数传递时的简单情况（见练习7.8）。当一个嵌套深度为i的过程的新的活动开始时，

1. 在新的活动记录中保存 $d[i]$的当前值；同时

2. 使 $d[i]$指向新的活动记录。

在活动结束前，$d[i]$ 被恢复为已保存的值。

上述步骤的验证过程如下。假设一个嵌套深度为 j 的过程调用一个嵌套深度为 i 的过程。同讨论访问链时一样，根据源代码中被调用过程是否嵌套在调用过程中，可以分为两种情况进行讨论：

1. $j < i$ 的情况。$i = j + 1$并且被调用过程嵌套在调用过程中。display 表的前 j 个元素不需要进行改变，我们将 $d[i]$设为指向新的活动记录。这种情况如在图7-26a中sort调用 quicksort 和图7-26c中 quicksort 调用 partition 时所示。

417 ~ 421

⊖ 注意，q(1,9)还保存了$d[2]$，虽然碰巧第二个display元素以前从来没有被使用，而且不需要被恢复。对于 q 的所有调用，保存$d[2]$比运行时决定此存储是否必要要容易一些。

2. $j \geq i$ 的情况。嵌套深度为1, 2, …, i-1的调用和被调用的外围过程必须相同。这里，我们在新的活动记录中保存 $d[i]$的旧值，然后将 $d[i]$ 指向新的活动记录。display 表将被正确地维护，因为它的前 i-1 个元素没有改变。

在图7-26b中，当 quicksort 被递归调用时就是情况2的一个例子，这里 $i = j = 2$。一个更好的例子如图7-26d所示，当嵌套深度为3的活动记录 p(1,3) 调用嵌套深度为2的e(1,3)时，它们的外层过程是嵌套深度为1的 s（程序在图7-22中）。注意，当e(1,3)被调用时，属于 p(1,3)的 d [3] 的值仍然在 display 表中存在，虽然当控制权在 e 中时它不能被访问。如果 e 调用另一个嵌套深度为3的过程，该过程会存储 d [3] 的值并在返回 e 时恢复该值。可见，每个过程都可以看到低于自己嵌套深度的所有嵌套深度的 display 表。

display 表可以在几个地方存放。如果有足够数量的寄存器，数组形式的 display 表可以是一组寄存器的集合。注意，编译器可以决定这个数组的长度，也就是程序中的最大嵌套深度。另外，display 表可以保存在静态分配的内存中，并且可以通过适当的 display 表指针使用间接地址访问活动记录。在支持间接寻址的机器中使用这种方法比较合适，虽然每次间接寻址将耗费一次内存循环。另一种可能是将 display 表存放在运行时堆栈中，在每个过程的入口建立它的一个新的复本。

7.4.4 动态作用域

在动态作用域中，一个新的活动记录将继承已经存在的非局部名字与存储空间的绑定。被调用的活动记录中的一个非局部名字 a 同调用过程中的该名字指向相同的存储空间。对于被调用过程中的非局部名字将建立新的绑定，该名字指向新的活动记录中的存储空间。

图7-27中的程序说明了动态作用域。第(3)和(4)行的过程 show 将显示非局部名 r 的值。根据 Pascal 的词法作用域规则，非局部名 r 是在第(2)行的声明的作用域之内，因此该程序的输出是

```
0.250 0.250
0.250 0.250
```

如果使用动态作用域规则，输出是

```
0.250 0.125
0.250 0.125
```

在第(10)和(11)行，当主程序调用 show 过程时，显示0.250，因为使用的是主程序中的局部变量r。然而，当在第(7)行show被过程 small 调用时，将显示0.125，因为使用的是 small 的局部变量 r。

```
(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7-27 输出依赖于使用的是词法作用域还是动态作用域

下面是两种实现动态作用域的方法，它们分别与在词法作用域的实现中使用的访问链和 display 表有相似之处。

1. *深访问*。从概念上讲，如果访问链与控制链指向相同的活动记录则得到动态作用域。一种简单的实现方法是分配访问链并使用控制链在栈中查找，直到找到包含非局部名字的存储单元的第一个活动记录。*深搜索*一词就来源于这种搜索可能会深入到栈中这一事实。搜索的深度取决于程序的输入，并且不能在编译时确定。

2. *浅访问*。这种方法的思想是将每个名字的当前值存储在静态分配的存储空间中。一旦过程 p 开始一个新的活动记录，p 中一个局部的名字 n 将取代为 n 静态分配的内存空间。n 的前

一个值必须保存在 p 的活动记录中，当该活动记录结束时必须恢复 n 的值。

可以权衡选择这两种方法：深访问在访问一个非局部名字时需要较长的时间，但是在一个活动记录开始和结束时没有多余的开销。浅访问可以直接查找非局部名，但是在一个活动记录开始和结束时需要额外的时间开销来维护这些值。如果函数作为参数被传递和作为结果被返回，通过深访问方法能够获得更直接的实现。

422 ~ 423

7.5 参数传递

当一个过程调用另一个过程时，它们之间交换信息的方法通常是通过非局部名字和被调用过程的参数来实现的。图7-28的过程使用了非局部名字和参数来交换 a[i]和 a[j]的值。这里，数组 a 对过程 exchange 是非局部的，i 和 j 是参数。

```
(1)  procedure exchange(i, j: integer);
(2)      var x : integer;
(3)      begin
(4)          x := a[i]; a[i] := a[j]; a[j] := x
(5)      end
```

图7-28 带有非局部变量和参数的 Pascal 过程

本节将讨论形参和实参相关联的几种方法。包括传值调用、引用调用、复制-恢复、传名调用和宏扩展。了解语言（或编译器）所使用的参数传递方法是很重要的，因为程序的结果可能依赖于所使用的方法。

为什么会有这么多种方法呢？不同的方法来源于对表达式代表什么的不同解释。像 a[i] := a[j] 这样的赋值语句中，表达式 a[j] 代表值，而 a[i] 代表 a[j] 的值要被置入的存储单元。使用存储单元还是用表达式所代表的值，取决于表达式出现在赋值号的左边还是右边。同第2章一样，术语“左值”指的是表达式所表示的存储单元，“右值”指的是这个存储单元中所存的值。所谓的“左”和“右”来源于在表达式的左边还是右边。

参数传递方法之间的区别主要是基于实参代表的是左值、右值还是实参的正文本身。

7.5.1 传值调用

在某种意义上，传值调用是最简单的参数传递方法。它计算实参，并把它的右值传给被调用过程。C 语言使用传值调用，Pascal 的参数常常也按此方式传递，本章到目前为止的所有程序都是按这种方法传递参数的。传值调用可以按下面的方式实现：

1. 把形参当作局部名字看待，形参的存储单元在被调用过程的活动记录中。

2. 调用者计算实参，并把其右值放入形参的存储单元中。

传值调用的显著特征是对形参的运算不影响调用者活动记录中的值。如果图7-29的第(3)行没有关键字

```
(1)  program reference(input,output);
(2)  var a, b: integer;
(3)  procedure swap(var x, y: integer);
(4)      var temp : integer;
(5)      begin
(6)          temp := x;
(7)              x := y;
(8)              y := temp
(9)      end;
(10) begin
(11)     a := 1; b := 2;
(12)     swap(a,b);
(13)     writeln('a =', a); writeln('b =', b)
(14) end.
```

图7-29 带有过程 swap 的 Pascal 程序

var，Pascal 将把 x 和 y 的值传递给过程 swap。第(12)行的调用 swap(a,b) 不会改变 a 和 b 的值。传值调用时，调用 swap(a,b) 的效果等价于下面的步骤：

```
   x := a
   y := b
temp := x
   x := y
   y := temp
```

其中 x、y 和 temp 是 swap 的局部变量。虽然这些赋值改变了局部变量 x、y 和 temp 的值，但当控制从这个调用返回且 swap 的活动记录被释放时，这个改变将消失。因而这次调用没有影响到调用者的活动记录。

传值调用的过程只能通过非局部名字（见图7-28中的 exchange）或作为值传递的指针来影响其调用者。在图7-30的C程序中，第(2)行的 x 和 y 声明为指向整数的指针，第(8)行的调用swap(&a,&b)中的&操作符将导致指向a和b的指针被传给swap。这个程序的输出是

424
~
425

```
a is now 2, b is now 1
```

在这个例子中，指针的使用指出了编译器怎样用引用调用来交换值。

```
(1) swap(x,y)
(2) int *x, *y;
(3) {   int temp;
(4)     temp = *x; *x = *y; *y = temp;
(5) }
(6) main()
(7) {   int a = 1, b = 2;
(8)     swap( &a, &b );
(9)     printf("a is now %d, b is now %d\n",a,b);
(10) }
```

图7-30 在传值过程调用中使用指针的C程序

7.5.2 引用调用

当参数通过引用（也叫做传址调用或传位置调用）传递时，调用过程把实参存储单元的地址传递给被调用过程：

1.如果实参是有左值的名字或表达式，则传递这个左值本身。

2.如果实参是 a+b 或2这样的表达式，它没有左值，则计算该表达式的值并存入新的存储单元，然后传递这个单元的地址。

在被调用过程的目标代码中，形式参数的引用是通过传给该被调用过程的指针来间接引用的。

例7.7 考虑图7-29中的过程 swap。用 i 和 a[i] 作为调用 swap 的实参（即 swap(i,a[i])）和下列步骤有相同的效果：

1. 把 i 和 a[i] 的地址（左值）复制到被调用过程的活动记录，假设对应于 x 和 y 的位置分别为 arg1 和 arg2。

2. 把 arg1 指向的单元的内容赋给temp(如设置 temp 等于 I_0，其中 I_0 是 i 的初值)。这一步对应 swap 的定义中第(6)行的temp := x。

3. 把 arg2 指向的单元的值赋给 arg1 指向的单元，即 i := a[I_0]。该步对应 swap 的定

义中第(7)行的 x := y。 426

4. 让 arg2 指向的单元的内容等于 temp 的值，即 a[I_0] := i。该步对应 y := temp。 □

有一些语言中使用引用调用，Pascal 的 var 参数就是按这种方式传递的。数组通常都通过引用传递。

7.5.3 复制-恢复

传值调用和引用调用的混合叫做*复制-恢复连接*（也称为*复制进复制出*，或*传值结果*）。

1. 在控制流进入被调用过程之前计算实参，实参的右值像传值调用那样传递给被调用过程。此外，如果实参有左值的话，在调用之前确定它的左值。

2. 当控制返回时，将形参的当前右值复制回实参的左值，该左值是上述调用前计算的左值。当然，只是有左值的实参被复制。

第一步是把实参的值“复制进”被调用过程的活动记录（进入形参的存储单元）；第二步是把形参的值“复制出”到调用过程的活动记录中（即调用前计算的实参的左值）。

使用复制-恢复的方式，swap(i,a[i])仍然能正确工作，因为 a[i] 的位置在调用前由调用过程计算并保存。所以，形参 y 的终值，即 i 的初值，能复制到正确的位置，即使 a[i]的单元因调用而被改变了（因为 i 的值已变）。

某些 Fortran 的实现使用了复制-恢复方式，其他实现使用引用调用。如果被调用过程有不止一种方式访问调用者的活动记录，则可以看出两种参数传递方式的区别。图7-31中第(6)行的调用 unsafe(a) 建立的活动记录可以把 a 作为非局部名字来访问，或者通过 x 访问 a。在引用调用方式下，对 x 和 a 的赋值都立即影响 a，所以 a 的最终值是0。但是，在复制-恢复方式下，实参a的值1复制到形参 x，而 x 的终值2在控制刚好返回前复制出到 a 的左值，所以 a 的终值是2。 427

```
(1) program copyout(input,output);
(2)     var a : integer;
(3)     procedure unsafe(var x : integer);
(4)         begin x := 2; a := 0 end;
(5)     begin
(6)         a := 1; unsafe(a); writeln(a)
(7)     end.
```

图7-31 引用调用换成复制-恢复后的输出变化

7.5.4 传名调用

传名调用由 Algol 中的*复制规则*定义为：

1. 过程被看作宏；也就是说，在调用过程中将调用替换为被调过程的过程体，但要把任何一个出现的形式参数都文字地替换为相应的实在参数。这种文字替换被称为*宏扩展*或*内嵌扩展*。

2. 被调用过程中的局部名字要保持与调用过程中的名字不同，因此可以考虑在进行宏扩展以前，将被调用过程中的每一个名字都系统地给以重新命名。

3. 如果需要保持实参的完整性，可以把实参用圆括号括起来。

例7.8 例7.7中的调用 swap(i, a[i])将被实现为：

```
temp := i
   i := a[i]
a[i] := temp
```

这样，在传名调用下，正如期望的那样，swap 将 i 设置成 a[i]，但这也导致了意想不到的结果，那就是 a[a[I_0]] 而不是 a[I_0] 被换成了 I_0，其中 I_0 是 i 的初值。这种现象发生的原因是在赋值 x := temp 中的 x 直到需要时才求值，但这时 i 的值已经改变了。很明显，如果采用传名调用法，就不可能给出一个 swap 的正确版本（见 Fleck[1976]）。 □

尽管传名调用有较高的理论价值，但是建议使用概念上与之相关的内嵌扩展技术来缩短程序运行时间。创建过程的活动记录需要一定的开销——要给活动记录分配空间、要保存机器的状态、要建立连接、要完成控制的转移。当过程主体很小时，调用序列的代码会超过过程主体的代码。因此在调用时用该主体的内嵌扩展来替代它会更有效，尽管程序会变大一点。在下面的例子中，内嵌扩展被用于传值调用的过程中。

例7.9 假定函数f在赋值语句 x := f(*A*) + f(*B*) 中采用传值调用。这里实参 *A* 和 *B* 是表达式。将 f 函数体中的形参的每次出现替换为表达式 *A* 和 *B* 会导致传名调用，回想一下上
428 一个例子中的 a[i]。

引进新的临时变量，在过程体被执行前先求实参表达式的值并存入该变量。

```
t1 := A;
t2 := B;
t3 := f(t1);
t4 := f(t2);
 x := t3 + t4;
```

这样，当第一个调用和第二个调用分别被扩展时，内嵌扩展将用 t_1 和 t_2 来替代形参的每一次出现[⊖]。 □

通常传名调用是通过给被调过程传递无参子程序实现的，这个子程序被称为*形实转换程序*（thunk），它能求出实参的左值或右值。像使用词法作用域的语言中作为参数传递的过程一样，一个形实转换程序带一个访问链，它指向调用过程的当前活动记录。

7.6 符号表

编译器使用符号表来记录名字的作用域以及绑定信息。每当在源文件中遇到一个名字时，就要搜索符号表。如果发现新的名字或者已有名字的新信息，符号表就要发生变化。

符号表机制必须允许我们能够添加新表项和快速查找现有表项。本节将给出两种符号表机制：线性表和散列表。我们是基于增加n个表项和查找 e 个表项所需的时间来评价每种机制的。线性表实现起来很容易，但当n和e值很大时，它的性能较差，而散列表却能在编程工作量和空间开销较大情况下提供更好的性能。这两种机制均适于处理最近嵌套作用域规则。

如果有必要，编译时能动态地增加符号表对编译器来说是很有用的。如果编译器写好后，符号表的大小就固定的话，那么符号表必须设计得够大，以便能处理可能遇到的任何源程序。这种固定大小的符号表对大部分程序来说都太大，但对有些程序来说又不够用。

7.6.1 符号表表项

符号表中的每个表项都是为一个名字的声明建立的。表项的格式不必一致，因为存储的有
429 关名字的信息与该名字的用法有关。每个表项可以作为一条记录来实现，该记录是由连续的存储字序列组成的。为保持符号表记录的一致，可以把一个名字的某些信息存放在符号表表项以外，而在记录中存放一个指向这些信息指针。

信息会在不同的时间放入符号表中。关键字在初始时全部放入表中。在3.4节中讲到的词法分析器通过查找表中的字母和数字序列来判定某个保留的关键字或名字是否已经被收集在符号表中了。使用这种方法，必须在词法分析之前将关键字放入符号表中。或者，如果词法分析

⊖ 存在与临时变量有关的隐含开销。它们可能引起在活动记录中分配额外的空间。如果活动记录中的局部变量已被初始化，则额外的临时变量还将导致时间的浪费。

器截获了这些关键字，那么它们就不用出现在符号表中了。如果这种语言不保留关键字，那么有必要在将关键字放入符号表时，提示一下它们可能被用作关键字。

当信息可用时随着属性值的填入，一旦知道名字的作用就可以建立符号表表项本身。某些情况下，只要在输入中看到一个名字，即可从词法分析器来初始化该表项。更多的时候，即使在同一程序块或者过程中，一个名字也可能会表示几种不同的对象。例如，C语言声明

```
int    x;
struct x { float y, z; };
```
(7-1)

既将 x 作为一个整型变量又作为一个有两个域的结构类型。在这种情况下，词法分析器只能返回名字本身（或指向这个名字串的指针）给语法分析器，而不会返回指向符号表的表项的指针。当该名字构成句子的作用一旦被发现，符号表中的记录就会被创建。对于(7-1)中的声明，符号表中产生两个关于 x 的表项，一个是整型，一个是结构类型。

对应于相应的声明，名字的属性被输入符号表，这也许是隐式的。标号通常是后跟冒号的标识符，这样，与识别这种标识符相关的动作将是把这些情况输入到符号表中。相似地，过程声明中的语法将指出这些标识符是形参。

7.6.2 名字中的字符

正如在第3章中讲到的，标识符或名字的记号**id**、构成名字的字符串所组成的词素与名字的属性三者是有区别的。字符串是不实用的方法，所以编译器通常采用名字的定长表示而不是词素。当符号表的表项第一次建立时，以及查找输入的一个词素来判断该名字是否已经出现过时，需要用到词素。名字的一般表示是一个指向符号表中相关表项的指针。 430

如果一个名字的长度有适中的上限，那么该名字的字符就可以存储在符号表的表项中，如图7-32a所示。如果名字的长度没有限制，或者这种限制很难做到，则可采用如图7-32b中的间

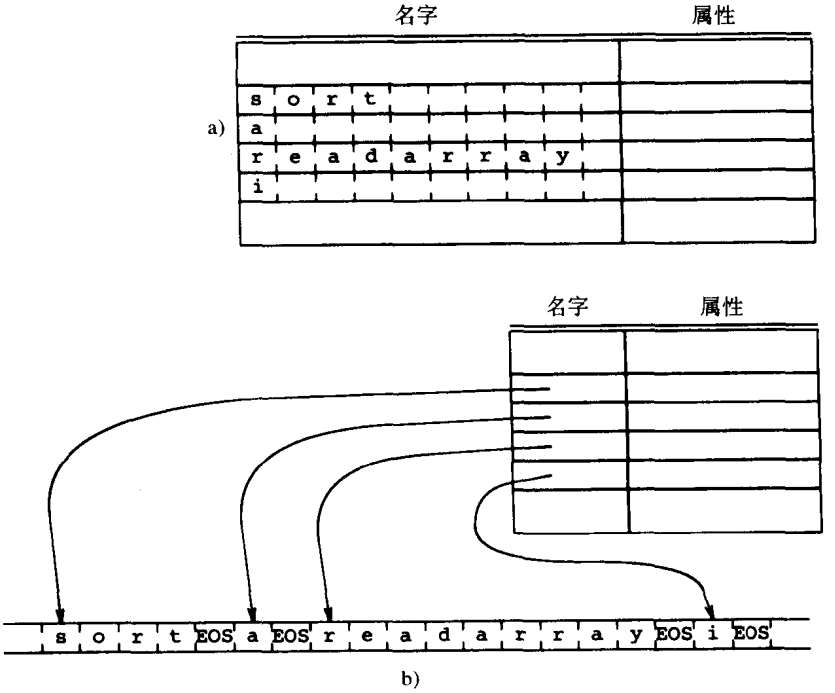


图7-32 名字字符的存储

a) 记录中的定长空间 b) 存放在分离的数组中

接方法。不给符号表的表项分配最大可能的空间来存储词素，而是只给它分配一个可以存储指针的空间，这样我们就能更有效地利用存储空间了。我们把名字的字符存储在一个单独的数组（字符串表）中，这样在符号表中名字的记录中，我们只存储了一个指针，该指针指向这个名字的第一个字符。图7-32b中的这种存储方法就允许符号表中名字域的长度为定值了。

构成名字的整个词素都必须被保存，这样才能保证该名字的所有使用都与符号表中的同一个记录相关联。不过，我们必须区分在不同声明的作用域中的相同词素的不同出现。 431

7.6.3 存储分配信息

在运行时与名字绑定的存储单元信息保存在符号表中。先考虑静态分配情况。如果目标代码是汇编语言，我们让汇编程序去处理不同名字的存储分配问题。我们所要做的是在程序的汇编代码产生后扫描符号表，并为每个名字生成汇编语言的数据定义，该定义需添加到汇编语言程序中。

如果由编译器产生的是机器代码，那么每个数据对象的位置就与固定的起始点相关，例如必须确定活动记录的起始点。相同的说明也用在数据模块上，这些模块是与程序相独立的。例如，Fortran语言中的COMMON块是独立装载的，所以应该知道名字相对于所在的COMMON块的起始点的位置。7.3节中的方法对Fortran来说应该有所修改，原因将在7.9节中说明，在那里当过程的所有声明已经处理后必须为名字分配偏移量并且需要处理EQUIVALENCE声明。

如果名字的存储区域在栈或者堆中，编译器就不会分配存储空间，编译器会策划每个过程的活动记录，如7.3节讨论的那样。

7.6.4 符号表的线性表数据结构

符号表最简单、最容易实现的数据结构是记录的线性表，如图7-33所示。我们用一个或几个等价的数组来存储名字和它们的相关信息。新名字将按遇到的顺序加入数组。数组的最后一个位置由 *available* 指针标识，它指向下一个符号表表项的位置。名字的查找从数组的尾端向起始端进行。一旦名字被查到，它的相关信息就能在紧接着的位置查到。如果从头到尾都没有找到这个名字，就会出错——期望的名字不在表中。

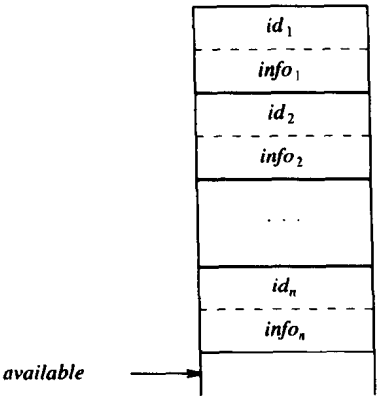


图7-33 记录的线性表

注意，为名字建立表项和在符号表中查找名字是独立的操作——我们希望能够独立地执行每个操作。在块结构语言中，名字的出现属于该名字的最近嵌套声明的作用域。我们可以使用线性表数据结构来实现该作用域规则，方法是每次声明一个名字就为它建立一个新表项。新表项放在紧接着 *available* 指针的地方；该指针也就会按照符号表记录的大小向下增长。既然表项是从数组的起始处按序插入的，那么它们出现的顺序就和产生的顺序一样了。通过从 *available* 到数组的起始处进行查找，我们一定可以找到最近产生的表项。

如果符号表中包含n个名字，而且插入一个新名字前不必查找它是否已存在，那么插入一个名字所花费的时间是个常数。如果不允许多个表项对应一个名字，则需查找整个符号表以确认名字是否已经在表中，它花费的时间与 n 成正比。找到这个名字平均需要查找 $n/2$个名字，那么查找时间仍与 n 成正比。既然查找和插入时间都与 n 成正比，那么插入 n 个名字和查找e个名字的总时间最多是 $cn(n+e)$，这里 c 是一个常数，它代表几个机器操作的时间。在一个

中等大小的程序中，我们设 $n = 100$而 $e = 1000$，这样所有的操作需要上百万个机器操作时间。这是可以忍受的，因为整个处理时间不到一秒。但是，当 n 和 e 都扩大十倍时，处理时间将扩大100倍，这样的处理时间就不能忍受了。这同时也告诉我们编译器在哪里花费了时间，以及判定是否在线性表的查找上花费了太多的时间。

7.6.5 散列表

各种各样的散列技术已经在许多编译器上实现。这里我们考虑一种比较简单的称为外散列表的散列技术，外散列表对表中表项的数量没有限制。这种表在 n 个名字中做 e 条查找时，查找时间与 $n(n+e)/m$ 成正比，其中 m 为任意常数。既然 m 可以任意取值，就可以把它取得很大，直到 n。这种方法一般要比线性表的效率高，所以它是经常采用的方法。但是，这种方法花费的空间随 m 的增大而增大，所以时间和空间的权衡是不可避免的。 432 ~ 433

基本的散列表如图7-34所示。它的数据结构包括两部分：

1. 由一个固定的数组组成的散列表，数组的 m 个元素是 m 个指向表项的指针。

2. 表项被组织到 m 个独立的链表中，这些链表称为桶（有些桶可能是空的）。符号表中的每条记录刚好出现在某一个链表中。记录的存储可能来源于记录数组，下一节将会讨论此内容。另外，动态分配策略在获得空间的同时会损失效率。

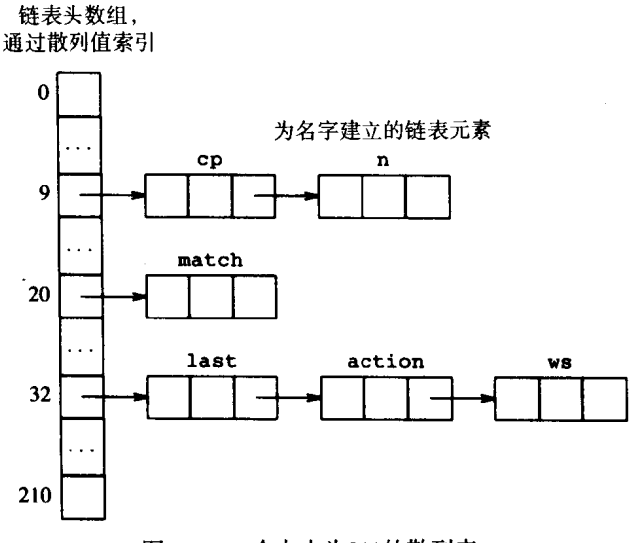


图7-34 一个大小为211的散列表

为了判断字符串 s 的表项是否在符号表中，我们对 s 采用散列函数 h，$h(s)$ 返回 0 到 $m-1$之间的一个整数。如果 s 在符号表中，那么它位于链表的 $h(s)$ 位置上。如果s不在符号表中，那么为它创建一条记录，并把它插到 $h(s)$ 对应的链表前。

根据经验可知，如果大小为 m 的散列表保存了 n 个名字，那么平均每个链表有 n/m 个记录。通过选择 m 的值，n/m 会达到一个小的常数，比如2，那么查找表中表项的时间也就成为一个常数了。

符号表所占的空间包括散列表的 m 个字和 n 个表项的 cn 个字，其中 c 为每个表项占用的字数。这样散列表的空间只依赖于 m，表项占用的空间只依赖于表项的个数。 434

m 的选择依赖于对符号表的应用。如果把 m 选为几百，那么即使对于中等大小的程序来说，查表所用的时间相对于编译程序的总开销而言也是可以忽略的。但是当程序的规模很大时，

m 的值定的大点比较好。

人们把很大的注意力放在散列函数的设计上，使得该函数可以很容易地计算出字符串的散列值并把所有的字符串均匀地分配到 m 个链表中去。

下面是一个适合计算散列函数方法：

1. 从字符串 s 的字符 c_1，c_2，…，c_k 中确定正整数 h。一般的语言都支持把字符转换为整数。Pascal语言中就有这样的函数 *ord*；如果对字符进行数学运算，C语言能自动把它转化为整型。

2. 把上面确定的 h 转化为链表的编号，即 0 到 m-1之间的一个整数。最简单的方法是用 h 除以 m，把余数作为链表的编号。采用余数法在 m 是素数时效果更好，这就是图7-34中取211而不是200的原因。

散列函数察看字符串的所有字符肯定比只察看字符串的头几个或中间几个字符麻烦。要记住，字符串在输入编译器之前已经被程序使用了，所以程序已做了避免它们发生冲突的工作，比如具有一定风格的格式。人们常用“一簇”名字，比如baz、newbaz、baz1等。

一个简单的计算 h 值的方法是将字符串中所有字符对应的整数值加在一起。更好的方法是在加入下一个字符的值以前将老的 h 值乘以一个常数 α，即令 $h_0=0$，$h_i=\alpha h_{i-1}+c_i$，$1\leqslant i\leqslant k$，且令 $h=h_k$，其中 k 是字符串的长度（记住散列值是用 h **mod** m）。简单地将每个字符对应的整数值相加就是 α 等于1的情况。相似的策略是用 αh_{i-1} 与 c_i 进行异或运算，而不是加。

```
(1) #define PRIME 211
(2) #define EOS '\0'
(3) int hashpjw(s)
(4) char   *s;
(5) {
(6)     char *p;
(7)     unsigned h = 0, g;
(8)     for ( p = s; *p != EOS; p = p+1 ) {
(9)         h = (h << 4) + (*p);
(10)        if (g = h&0xf0000000) {
(11)            h = h ^ (g >> 24);
(12)            h = h ^ g;
(13)        }
(14)    }
(15)    return h % PRIME;
(16) }
```

图7-35 用C语言编写的散列函数 *hashpjw*

对于32位的整数来说，如果我们取 $\alpha=65599$，一个接近2^{16}的素数，那么在计算 αh_{i-1} 时很快就会发生溢出。既然 α 是素数，我们可以忽略溢出而只取它的低32位。

一组试验是采用图7-35中所示的散列函数 *hashpjw*，该散列函数来源于 P.J.Weinberger 的C语言编译器中。对各种大小的表，试验结果都很好（如图7-36所示）。表的大小包括大于100，200，…，1500的第一个素数。类似的另一组试验是将旧的 h 值乘以65599，忽略溢出，再与下一字符相加得到新的 h 值。函数 *hashpjw* 中 h 的初值是0。对每个字符c，将 h 左移四位，并加上 c，得到新的 h。如果 h 的高4位上有一位是1，就把这4位右移24位，再与 h 进行异或运算，最后把 h 的高4位中曾经是1的位再置成0。

例7.10 为了得到最好的结果，当散列函数设计完以后，散列表的长度和经常出现的输入就成了我们关心的问题了。例如，我们希望最常出现的名字的散列值是惟一的。如果把关键字也输入符号表中，那么关键字是最常出现的，尽管在某些C语言程序中 i 的使用次数是 while 的三倍。

一种测试散列函数的方法是看有多少个字符串落在同一个链表中。假设文件F是由n个字符串组成的，再假设有 b_j 个字符串落入了链表 j 中，$0\leqslant j\leqslant m-1$。字符串在链表中分布是否均匀的度量方法是计算

$$\sum_{j=0}^{m-1} b_j(b_j+1)/2 \tag{7-2}$$

这个式子给我们的直观感觉是：针对要查找的一个字符串，在链表 j 中要比较第一项看是否相等，不相等则比较第二项，如此进行下去直到第 b_j 项。平均比较次数用 1, 2, …, b_j 的和是 $b_j(b_j+1)/2$。

在练习7.14中，将字符串随机分布在各个散列桶中的散列函数(7-2)的值是

$$(n/2m)(n + 2m - 1) \tag{7-3}$$

435 ~ 436

图7-36中给出在9个文件上应用几种散列函数时，式(7-2)和(7-3)的比率。这9个文件分别是：

1. C语言的程序中，50个出现最频繁的名字和关键字。
2. 同1，但是考虑100个出现最频繁的名字和关键字。
3. 同1，但是考虑500个出现最频繁的名字和关键字。
4. 952个UNIX操作系统内核中使用的外部名字。
5. 627个C++（Stroustrup[1986]）生成的C程序中的名字。
6. 915个随机产生的字符串。
7. 614个取自本书3.1节的字。
8. 1201个英语单词，并将xxx作为其前缀或后缀。
9. 从v100到v399的300个名字。

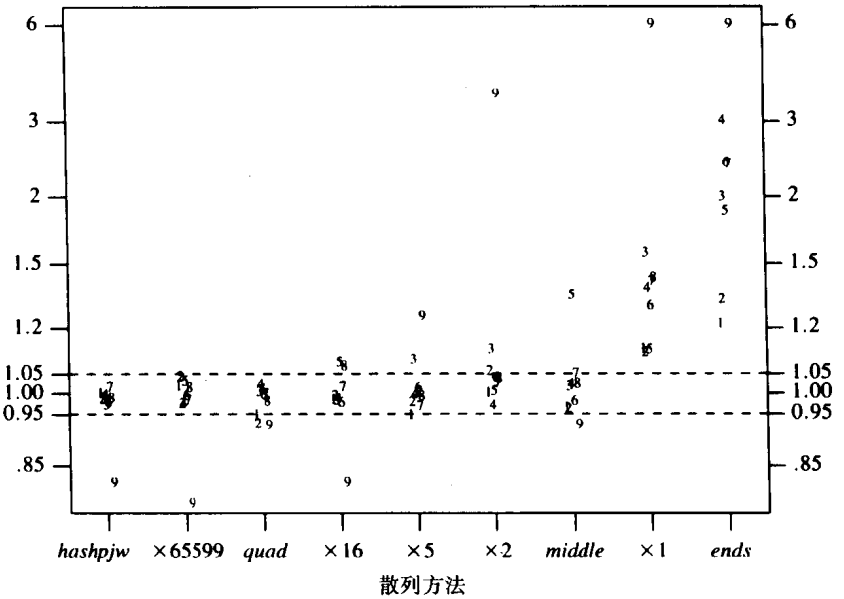


图7-36 散列函数在大小为211的表上的相对性能

函数 *hashpjw* 如图7-35所示。函数 $\times\alpha$ 计算 h **mod** m，其中 α 是一个整常数，h 是递归得到的，其初值为0，每次在旧值上乘以α再加入下一个字符的值得到新值。函数 *middle* 的 h 值取自字符串的中间四个字符，而函数 *ends* 的 h 值是将字符串的前三个和后三个字符相加得到的。最后，函数 *quad* 把每四个连续的字符聚合成一个整数，再累加这些整数。 □

437

7.6.6 表示作用域的信息

符号表中的表项对应于名字的声明。当在符号表中查找源文件中出现的一个名字时，声明这个名字的符号表表项必须被返回。源语言的作用域规则确定了哪个声明是正确的。

一种简单的方法就是为每个作用域建立一个独立的符号表。实际上，这种为每个过程或作用域建立的符号表与编译时的活动记录是等价的。一个过程的非局部名字的信息可以通过扫描外围过程的符号表而得到，这里的外围过程是指遵循语言作用域规则的那些外围过程。同样，我们可以把过程的局部信息放在程序语法树中相应过程的节点上。通过这种方法，符号表就被集成到输入程序的中间表示中了。

最近嵌套作用域规则可以通过改写本节前面所提到的数据结构来实现。我们通过为每个过程赋予一个惟一的编号来跟踪过程中局部变量的名字。如果语言是块结构的，程序块也必须编号。过程编号在语法制导方式下可以从标识过程开始与结束的语义规则中计算出来。过程编号是过程中声明的局部变量的一部分；这样局部变量在符号表中的表示由两部分组成，即名字和过程编号。(在某些安排中，正如下面将要描述的，过程的编号并不实际出现，因为可以从它在符号表中的位置推导出来。)

当我们查找一个新的被扫描到的名字时，只有当名字的每一个字符都与表项中的名字的字符相匹配，而且相应的编号与正在处理的过程的编号相同，这样才算匹配。最近嵌套作用域规则可以通过以下对名字的操作来实现：

lookup：查找最近建立的表项
insert：建一个新表项
delete：删除最近建立的表项

这里删除的表项必须被保留；它们只是从活动符号表中删掉了。在单遍编译器中，当过程体处理完以后，符号表中有关它的作用域信息在编译时就不需要了。但在运行时可能是需要的，尤
438 其是当实现运行时诊断系统时。这种情况下，符号表中的这些信息必须加到为链接程序或运行时诊断系统产生的代码中。可以参照8.2节和8.3节中对记录中域名的处理。

为支持上面的操作，必须维护本节讨论的各种数据结构，如线性表和散列表。

我们在本节前面描述记录数组构成的线性表时，曾提到如何实现*lookup*。如果从尾端插入新表项，那么表项在线性表中的顺序就和插入表项的顺序相同了。从尾到头进行扫描即可发现最近建立的名字的表项。链表的情况与此类似，如图7-37所示。指针*front*指向链表中最近建立的那个表项。实现*insert*操作花费常数时间，因为新表项放在表的前面。通过从*front*指针处开始扫描，直到目标找到或者到达了线性表的最尾端来完成*lookup*的实现。在图7-37中，块B_2嵌套在块B_0中，所以B_2中声明的a的表项比B_0中声明的a的表项更靠近线性表的前端。

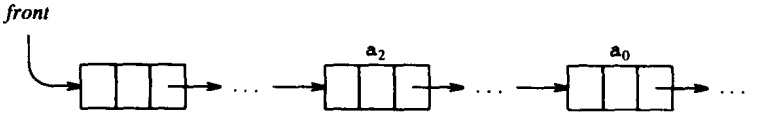


图7-37 a的最近出现的表项更靠近指针*front*

对于*delete*操作，注意到在嵌套最深的过程中声明的表项离表的前面最近，所以不必在每一个表项中保存过程编号——如果跟踪每个过程的第一个表项，那么，在处理完一个过程的作用域以后，一直到第一个表项的所有表项就都可以从活动符号表中删除了。

散列表由m条链组成，这m条链通过数组访问。既然一个名字总是散列到同一个链中，就可以像图7-37那样维护单独的链表。但是执行删除操作时，我们希望不要查找整个散列表。可以使用下面的方法，假设每个表项有两条链：

1. 一条散列链，它将所有散列值相同的表项链在一起。

2. 一条作用域链，它将同一作用域中的所有表项链在一起。

如果从散列表中删除表项时对作用域链不做处理，那么这个作用域链将构成一个单独的（没有激活的）符号表。 439

从散列表中删除表项必须非常谨慎，因为删除一个表项对散列表中前面的表项会产生影响。回想一下我们通过将第 i-1个表项指向第 i+1个表项，来删除第 i 个表项的情况。因此仅仅通过作用域链来寻找第 i 个表项是不够的。在循环链表中，第 i-1个元素可以被找到，因为最后一个元素的指针指向第一个元素。另外，我们可以使用栈来跟踪含有要被删除的表项的列表。当一个新过程被扫描时，在栈内放置一个标记。标记上面是包含该过程中声明的名字表项的列表编号，当这个过程执行完毕时，标记上面的列表编号都从栈中弹出。练习7.11中将讨论另一种机制。

7.7 支持动态存储分配的语言措施

在本节中，我们将简要描述一下某些语言提供的支持动态存储分配的措施。这种数据的存储一般采用堆的方式。分配的数据一般需要显式释放，而分配操作本身可以是显式的或者隐式的。例如，在Pascal中，显式分配采用标准过程new，new(p)操作将内存分配给指向对象的指针p。在大多数的Pascal实现中释放内存使用dispose操作。

隐式内存分配一般发生在表达式的执行结果需要存储单元来存放的时候。比如Lisp中，当使用cons的时候，一个新单元就被插入到列表中去，不再被使用的单元会被自动回收。Snobol允许在运行时改变字符串的长度，并使用堆来管理字符串所使用的存储空间。

例7.11 图7-38所示的Pascal程序将产生图7-39所示的链表，并显示单元里的整数，其输出如下：

```
76          3
 4          2
 7          1
```

当程序从第(15)行开始执行的时候，为指针head分配的存储单元在整个程序的活动记录中。每次执行到下列语句时：

```
(11)        new(p); p↑.key := k; p↑.info := i;
```

调用new(p)导致在堆里分配一个单元，在第(11)行的赋值中p↑指向这个单元。

需要注意的是，从程序的输出来看，当控制从insert返回主程序后，分配的单元仍可以被访问。换句话说，当控制从insert返回主程序后，在insert的活动记录中仍然保留用new分配的存储单元。 □ 440

7.7.1 垃圾单元

动态分配的存储单元可能变得不可访问。这种在程序中分配了但是不能被引用的存储单元称为*垃圾单元*。在图7-38中，假设在第(16)和(17)行之间将nil被赋值给head↑.next：

```
(16)        insert(7,1); insert(4,2); insert(76,3);
            head↑.next := nil;
(17)        writeln(head↑.key, head↑.info);
```

图7-39中最左边的单元将包含一个 nil 指针而不是指向中间单元的指针。当指向中间单元的指针丢失后，中间以及最右边的存储单元将变为垃圾单元。

Lisp语言具有收集垃圾操作，该过程将在下一节中讨论。Pascal和C语言中没有垃圾收集操作，而是在程序中显式地释放不需要的存储单元。在这些语言中，释放的存储单元可以被重新使用，但垃圾单元在程序结束之前不能被使用。

```
(1)  program table(input, output);
(2)  type link = ↑ cell;
(3)       cell = record
(4)                   key, info : integer;
(5)                   next : link
(6)       end;
(7)  var  head : link;
(8)  procedure insert(k, i : integer);
(9)       var  p : link;
(10)      begin
(11)         new(p); p↑.key := k; p↑.info := i;
(12)         p↑.next := head; head := p
(13)      end;
(14) begin
(15)      head := nil;
(16)      insert(7,1); insert(4,2); insert(76,3);
(17)      writeln(head↑.key, head↑.info);
(18)      writeln(head↑.next↑.key, head↑.next↑.info);
(19)      writeln(head↑.next↑.next↑.key,
             head↑.next↑.next↑.info)
(20) end.
```

图7-38 Pascal 中用 new 进行的动态存储分配

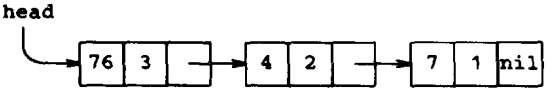


图7-39 图7-38中程序建立的链表

7.7.2 悬空引用

只要存储空间可以显式释放，悬空引用问题就会出现。如7.3节提到的，引用某个已释放的存储单元就会引起悬空引用。例如，考虑在图7-38中第(16)与(17)行之间执行语句 dispose (head↑.next)的效果：

```
(16)      insert(7,1); insert(4,2); insert(76,3);
          dispose(head↑.next);
(17)      writeln(head↑.key, head↑.info);
```

对dispose的调用释放了图7-40中head所指向的单元，但是head↑.next没有改变，所以变成了一个悬空的指针指向了一个已释放的单元。

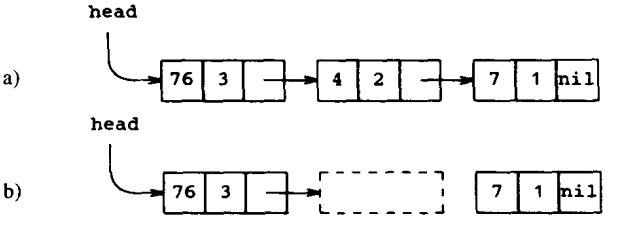


图7-40 悬空引用与垃圾单元的产生

a) 操作前 b) 操作后

悬空引用和垃圾单元是两个相互关联的概念：若空间释放发生在最后一个引用之前就会出现悬空引用，然而，若最后一次引用发生在空间释放之前就会存在垃圾单元。

7.8 动态存储分配技术

实现的动态存储分配技术依赖于存储的释放方式。如果存储是隐式释放的，则由运行时支持程序包负责确定存储块何时不再需要。如果由程序员显式释放内存，编译器就不需要做什么工作了。我们首先考虑显式释放的情况。

441 ~ 442

7.8.1 固定块的显式分配

动态存储分配的最简单形式是使用固定大小的块。如图7-41所示，通过将这些块连接成一个列表，存储分配与释放可以很快地执行，而且几乎不需要额外的存储空间。

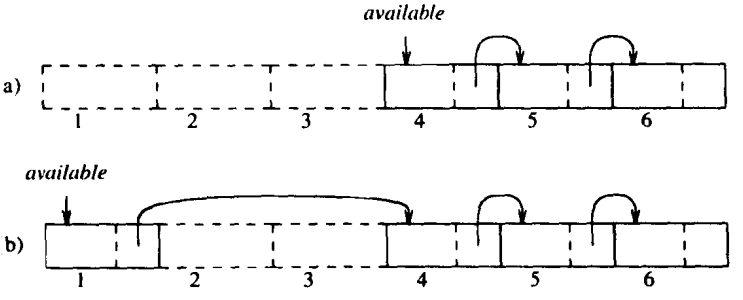


图7-41 一个释放的块被加到可用块列表中

假设存储块可以从一个连续的存储区域得到。该存储区域的初始化是使用每个存储块的一部分存放指向下一块的指针来实现的，指针 *available* 指向第一个存储块。存储分配指的是从列表中取下一个存储块，而存储释放就是将存储块放回列表。

管理存储块的编译程序不需要知道存储在各个存储块中的对象的类型。我们可以将各个存储块当作一个可变记录，编译程序将其看作一个存储块的链表，而用户程序将其看作是其他的数据类型。这样，由于用户可以使用整个块，所以不需要额外的存储空间。当存储块被返回时，编译程序可以使用存储块本身的一部分空间将其连接到可用块列表中去，如图7-41所示。

7.8.2 变长块的显式分配

当存储块被分配和释放后，存储中会出现*碎片*；也就是说堆中交替存在空闲块和已用块，如图7-42所示。

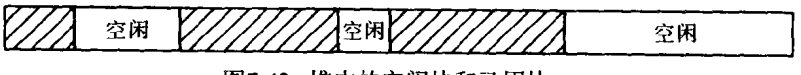


图7-42 堆中的空闲块和已用块

443

当程序分配了5个块后，继而又释放了第2个和第4个存储块，便会出现图7-42所示的情况。如果分配的存储块均是固定大小的话，则不存在碎片问题。但是如果分配的是变长存储块的话，就会出现图7-42所示的碎片问题。即使存在足够的可用空间，也无法为一个大于任何空闲块的存储块分配空间。

有一种分配变长存储块的方法称作*最先符合法*。需要分配大小为s的存储块时，查找第一个其长度f满足$f \geqslant s$的空闲块，然后将该块分为大小为s的已用块和大小为$f-s$的空闲块。注意存储分配将引入时间开销，因为我们必须查找足够大的空闲块。

当一个块被释放后，我们检查它是否与某个空闲块相邻。如果可能，则与之合并成为一个

更大的空闲块。将相邻空闲块合并为更大的空闲块可以防止出现更多的内存碎片。至于如何在可用块列表中分配、释放和维护空闲块还有很多具体的细节，在时间、空间以及大存储块的可用性等方面也要进行折衷考虑。读者可以参考 Knuth[1973a]或者 Aho, Hopcroft, and Ullman [1983]中对该问题的讨论。

7.8.3 隐式存储释放

隐式存储释放需要用户程序与运行包的协调，因为后者需要知道存储块何时不再被用户程序所使用。这种协调通过固定存储块格式来实现。我们目前的讨论假设存储块的格式如图7-43所示。

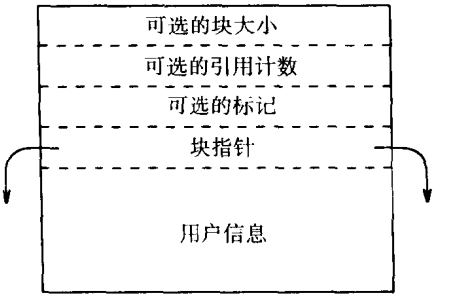


图7-43 块的格式

首先是块边界的识别问题。如果一个块的大小是固定的，则可以使用位置信息。例如，如果每个块占用20个字，则每20个字开始一个新块儿。否则，我们需要在一个不能被访问到的存储区域
444 内保存块的大小信息，以便确定块的开始地址。

第二个问题是识别块是否被占用。我们假设如果用户程序可能引用块内的信息，则该块是被占用的。这种引用可能通过一个指针或一系列指针指向该块，因此，编译程序需要知道存储区中所有指针的位置。使用图7-43的存储块格式，指针保存在块中的一个固定位置。另外，我们还可以假设存储块的用户信息区域不包含任何指针。

有两种方法可以用来进行隐式存储释放。下文将简要介绍一下，更多细节请参见 Aho, Hopcroft, and Ullman [1983]。

1. 引用计数。我们跟踪直接指向当前块的存储块的数目。如果该计数值降为0，则释放该块，因为它已不再被引用了。换句话说，该块变成了可被回收的垃圾单元。维护引用计数的时间开销很大；指针赋值 p := q 导致 p 和 q 指向的存储块引用计数都发生变化。p 指向的块数减1，而 q 指向的块数加1。引用计数的方法在不存在引用循环的情况下效果较好。例如，在图7-44中，每个块都不能被任何其他程序访问，因此它们都是垃圾单元，但是每一个的引用计数均为1。

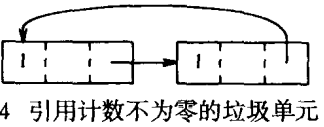


图7-44 引用计数不为零的垃圾单元

2. 标记技术。另一种方法是暂时挂起目前执行的用户程序，并利用静态指针来判断哪些块被占用。这种方法需要知道所有指向堆的指针。概念上，我们通过这些指针来对堆内的存储空间做标记，任何标记过的存储块表示被占用，其他的可以被释放。更具体地说，我们先将堆内所有存储块标记为*未使用*，然后根据指针标记在这一过程中到达过的任何块为*已使用*的存储块，最后一遍对堆的遍历将允许仍标记为未使用的存储块被收回。

445 对于变长块，我们有可能需要将已用存储块从当前位置移走[⊖]。该过程称为*压缩*，它将所有已用块移至堆的一端，以便将所有的空闲块收集为一个大的空闲块。压缩过程也需要关于指针的信息，因为移动一个已用块时，所有的指针都需要进行相应的调整。其优点是可以消除可用存储的碎片问题。

7.9 Fortran语言的存储分配

如7.3节所述，Fortran 语言可以进行静态存储分配。但是，也存在一些问题，如对COMMON

⊖ 对固定大小的块我们也可以这么做，但得不到什么好处。

和 EQUIVALENCE 声明的处理等，它们在 Fortran 语言中是比较特殊的。Fortran 编译器可以产生大量的数据区，即用来保存对象值的存储块。在 Fortran 中，每一个过程使用一个数据区，每一个已命名 COMMON 块和空白 COMMON，如果使用的话，也使用一个数据区。符号表中必须记录每一个名字所在的数据区以及在此数据区内的偏移（即相对于该数据区起点的位置）。编译器必须最终决定数据区与可执行代码的对应关系以及数据区之间的对应关系，但是这个选择是比较武断的，因为各个数据区相互独立。

编译器必须计算各个数据区的大小。对于过程的数据区来说，一个计数器就足够了，因为每个过程处理完后就可以知道它们的大小。对于 COMMON 块来说，在所有过程的处理期间必须为每一个块保存一条记录，因为每个使用块的过程可能需要不同的大小，而实际的大小是各过程所需要的最大值。如果分别编译各过程，则在进行连接的程序间必须使用链接编辑程序将 COMMON 块的大小定为同名块的最大值。

编译器为每个数据区建立一个内存映像，用来描述这一数据区的内容。这个内存映像可能仅仅包括名字在符号表中的表项或在数据区的偏移。我们不需要回答“该数据区内都有哪些名字？”但是，在 Fortran 中，对所有过程的数据区我们知道上述问题的答案，因为在过程中声明的所有名字（如果不是 COMMON 中的名字或等价于 COMMON 中的名字）都在该过程的数据区中。COMMON 名字的符号表表项按照它们在块中出现的顺序被链接起来。实际上，直到整个过程处理完才能确定名字在数据区中的偏移（Fortran 中，数组可以在声明维数之前进行声明），所以建立 COMMON 名字链是很必要的。

一个 Fortran 程序由主程序、子程序和函数（我们将其统称为过程）组成。名字的每一次出现都有一个作用域，该作用域只包含一个过程。到达过程的结尾时我们就可以为该过程生成 446
目标代码。如果这样的话，符号表中的大部分信息就都可以被删掉。我们只需要保存那些外部名字，即其他过程或 COMMON 块中的名字。这些名字对于被编译的整个程序来说可能不是外部的，但必须将它们保留到所有的过程都处理完为止。

7.9.1 COMMON区域中的数据

我们为每个 COMMON 块建立一条记录，记录中给出该块中声明的属于当前过程的第一个和最后一个名字。当处理如下声明的时候：

```
COMMON /BLOCK1/ NAME1, NAME2
```

编译器必须做如下工作：

1. 在 COMMON 块名字的表中，如果不存在 BLOCK1 的记录，为 BLOCK1 建立一条记录。

2. 在符号表的 NAME1 和 NAME2 的表项中，建立一个指针使之指向符号表中 BLOCK1 的表项，表示它们在 COMMON 块中，而且是 BLOCK1 的成员。

3. a) 如果刚刚为 BLOCK1 建立了记录，则在记录中设置一个指向符号表中 NAME1 表项的指针，用以指示该 COMMON 块中的第一个名字。然后使用为连接相同的 COMMON 块的成员而保留的符号表域将符号表中 NAME1 的表项与 NAME2 的表项连接起来。最后，在 BLOCK1 的记录中设置一个指向符号表中 NAME2 表项的指针，用以指示该块中的最后一个成员。

b) 如果这不是 BLOCK1 的第一次声明，则只需简单地将 NAME1 和 NAME2 连接到BLOCK1 的成员列表末端即可。当然，在为 BLOCK1 建立的记录中指向 BLOCK1 列表末端的指针也需要随之更新。

处理完一个过程后，我们将应用稍后提到的等价算法。我们也许会发现 COMMON 中有一些额外的名字，这是因为它们与 COMMON 中的某些名字等价。实际上没有必要将这样的名字

XYZ 连接至 COMMON 块的列表中。符号表中 XYZ 的表项中有一位，通过对这一位进行设置来表示 XYZ 与某一个名字等价。后面将要提到一个数据结构，用来给出这样的 XYZ 相对于在 COMMON 中声明的某个名字的位置。

执行了这样的等价操作以后，我们可以通过扫描块中名字的列表来为每一个 COMMON 块建立一个内存映像。将计数器初始化为零，并且使得列表中的每一个名字的偏移等于计数器的当
447 前值。然后，将名字所代表的数据对象所占用的内存单元数加到计数器中。这样 COMMON 块记录可以被删除，其空间可以被下一过程使用。

如果 COMMON 中的名字 XYZ 与另一个不在 COMMON 中的名字等价，我们必须确定从 XYZ 至其他等价于 XYZ 的名字的偏移的最大值。例如，如果 XYZ 是一个实数，等价于 A(5,5)，此处A是一个10 × 10的实数数组。A(1,1)在XYZ之前44个字的位置，而 A(10,10)在 XYZ 后55个字出现，如图7-45所示。A的存在不影响 COMMON 块的计数器，计数器仅当XYZ被考虑时才递增1个字，与 XYZ 与谁等价无关。但是，COMMON 块的数据区末端必须与起始端距离足够远，以便用来分配数组A。因此，我们记录从 COMMON 块的开始至任何等价于该块成员名字的最大偏移。在图7-45中，数量至少为 XYZ 的偏移加上55。我们同时需要检查数组 A 没有超出数据区的起始位置。也就是说，XYZ 的偏移至少为44，否则，将产生错误诊断信息。

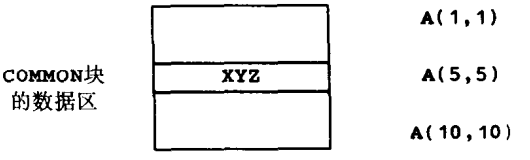


图7-45 COMMON与EQUIVALENCE语句的关系

7.9.2 一个简单的等价算法

第一个处理等价语句的算法出现在汇编器而不是编译器中。因为这些算法比较复杂，尤其是考虑到 COMMON 和 EQUIVALENCE 声明之间的相互作用，所以先考虑汇编语言中的一种典型情况。在这种情况下，EQUIVALENCE 语句的格式为：

```
EQUIVALENCE A,B+offset
```

其中，A 和 B 是存储单元的名字。该语句使得 A 表示位置B以后偏移量为 *offset* 的内存单元所在的位置。

一系列的 EQUIVALENCE 语句将一组名字归为一个等价集合，它们彼此之间的相对位置都通过 EQUIVALENCE 语句来定义。例如，声明序列

```
EQUIVALENCE A, B+100
EQUIVALENCE C, D-40
EQUIVALENCE A, C+30
448 EQUIVALENCE E, F
```

将名字划分为集合{A, B, C, D}和{E, F}，其中 E 和 F 代表相同的位置。C 在 B 后70个单元处，A 在 C 后30个单元处，而D在A后10个单元处。

为了计算等价集合，我们为每一个集合建立一棵树。树中每一个节点表示一个名字，同时包含相对于父节点的偏移量。在树根节点的名字称为*入口语句*（leader），任何名字相对于入口语句的位置可以根据从根到该节点的路径上所有偏移量之和来确定。

例7.12 上述等价集合{A, B, C, D}可以用图7-46中的树来表示。D 是入口语句，我们可以发现 A 在 D 前10个单元处，这是因为从 A 到 D 的路径上的偏移量之和为100 + (-110) = -10。 □

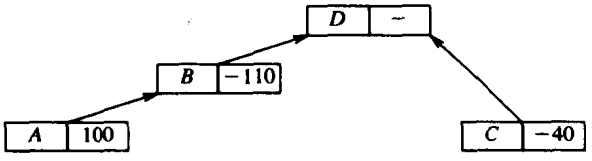


图7-46 表示等价集合的树

下面我们给出构造等价集合树的算法。符号表项中的相关域为：

1. *parent*，指向符号表中父节点的表项，如果该名字为根则其值为空（或不等价于其他任何名字）。

2. *offset*，给出一个名字相对于父节点名字的偏移量。

在该算法中我们假设任何名字均可以作为等价集合中的入口语句。实际上，在汇编语言中，集合中有且只能有一个名字利用伪操作定义实际位置，而该名字将作为入口语句。可以很容易地改变该算法，使之只采用一个特定的入口语句。

算法7.1 等价树的构造。

输入：如下形式的等价定义语句序列：

EQUIVALENCE A, B+*dist*

输出：等价树集合，满足对于输入的等价序列中出现的任何名字，我们都可以通过计算从名字到根的路径中偏移量的和来确定该名字相对于入口语句的距离。 449

方法：对于每一个等价语句EQUIVALENCE A, B+*dist*，依次重复图7-47中的步骤。第(12)行的公式计算A的入口语句与B的入口语句的相对偏移。A的位置（记为l_A）等于c加上A的入口语句的位置（记为m_A）。B的位置（记为l_B）等于 d 加上 B 的入口语句的位置（记为 m_B）。因为 $l_A = l_B + dist$，所以$c + m_A = d + m_B + dist$，即 $m_A - m_B = d - c + dist$。 □

```
      begin
(1)     令p和q分别指向节点A和B;
(2)     c := 0; d := 0;  /* c 和d 用来计算A和B相对于它们各自集合的入口语句的偏移 */
(3)     while parent(p) ≠ null do begin
(4)         c := c + offset(p);
(5)         p := parent(p)
        end;     /* 使p指向a的入口语句, 并累加偏移 */
(6)     while parent(q) ≠ null do begin
(7)         d := d + offset(q);
(8)         q := parent(q)
        end;     /* 对B执行同样的工作 */
(9)     if p = q then /* A 和B 等价 */
(10)        if c - d ≠ dist then error;
                /* A 和B 已经赋成了不同的相对位置 */
        else begin    /* 合并集合A 和 B */
(11)        parent(p) := q    /* 令 A 的入口语句成为 B 的入口语句的儿子 */
                a child of B's leader */
(12)        offset(p) := d - c + dist
        end
    end
```

图7-47 等价算法

例7.13 如果我们处理如下语句：

```
EQUIVALENCE     A, B+100
EQUIVALENCE     C, D-40
```

我们将得到图7-46所示的格局，只是在节点 B 中没有偏移 -110，而且没有 B 到 D 的链接。当
450 我们处理如下语句时：

```
EQUIVALENCE     A, C+30
```

我们发现执行完第(3)行的 while 后，p 指向 B，而执行完第(6)行的 while 后，q 指向 D。我们也得到 $c = 100$、$d = -40$。在第(11)行处我们使得 D 成为 B 的父节点，B 节点的 *offset* 域值为 -110，也就是（-40）-（100）+30。 □

算法7.1的时间复杂性为n^2，其中n为要处理的等价语句的条数，因为最坏情况下，第(3)行和第(6)行的循环对于各自树中的每一个节点都需要执行一遍。等价操作在编译时只需要很少的一个时间片断，因此n^2步的代价并不大，而且比图7-47算法更加复杂的算法也许并不适用。但是我们可以通过做很简单的两个工作便可以使得算法7.1在处理一些等价语句时的复杂性降为线性。一般地，等价集合可能大不到需要使用这些优化，值得注意的是等价操作可以用在许多像集合合并这样的处理中。例如，很大一部分高效的数据流分析算法均建立在快速等价算法上，有兴趣的读者可以参考第10章的参考文献注释。

第一种优化方法是我们可以为每一个入口语句保存其树中的节点数。然后，在第(11)和(12)行，不是简单地将 A 的入口语句链接至 B 的入口语句，而是将计数值少的链接到另外一个上。这样使得树可以横向增长，以减少树的高度。这种方式完成的 n 个等价不会产生长于 $\log_2 n$ 个节点的路径，其证明留作练习。

```
begin
    h := offset(n_{k-1});
    for i := k-2 downto 1 do begin
        parent(n_i) := n_k;
        h := h + offset(n_i);
        offset(n_i) := h
    end
end
```

图7-48 偏移的调整

第二种方法是路径压缩。当在第(3)行和第(6)行的循环中沿着路径到达树根时，使得所有遇到的节点变成入口语句的子节点。也就是说，沿着路径记录遇到的所有节点 $n_1, n_2, \cdots, n_k$，这里 n_1 是A 或者 B 的节点，n_k 是入口语句。然后按照图7-48
451 中的步骤调整偏移量，使得$n_1, n_2, \cdots, n_{k-2}$成为 n_k 的子节点。

7.9.3 Fortran语言的等价算法

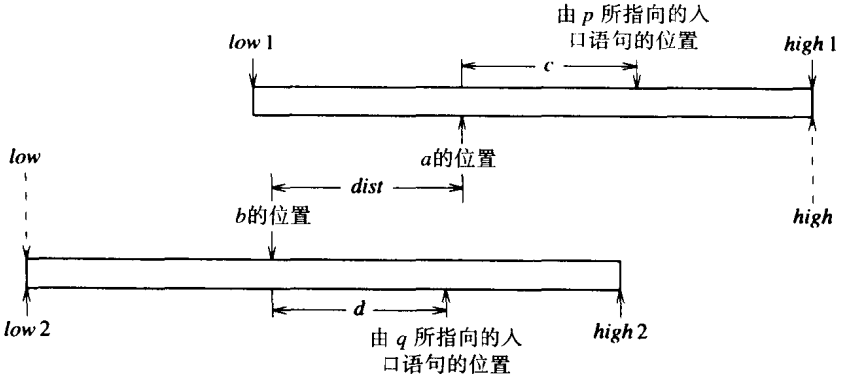
要使算法7.1适用于 Fortran 语言，则必须加入一些新特性。首先，我们必须确定等价集合是否在 COMMON 中。这可以通过如下方式实现：对于每一个入口语句，记录其集合中的每一个名字是否在 COMMON 中，如果是的话，在哪一个块中。

第二，在汇编语言中，等价集合的一个成员会成为一个语句的标号，使得集合中所有名字所表示的地址都可以相对于这个位置进行计算。但是，在 Fortran 语言中是由编译器决定存储位置的，所以一个不在COMMON中的等价集合可以被看作是“浮动”的，直到编译器在合适的数据区确定整个集合的位置。为了正确地执行，编译器需要知道等价集合的外延，即集合中名字所占存储单元的个数。为了处理这个问题，我们给入口语句增加 *low* 和 *high* 两个域，分别代表等价集合中相对于入口语句的最小和最大偏移量。第三，名字可以是数组，而且数组中间的位置可以等价于其他数组中的某个位置，这些事实也带来了一些小问题。

由于与每一个入口语句相关的有三个域（*low*、*high* 以及一个指向 COMMON 块的指针），我们不希望在所有的符号表表项中为这三个域分配额外的空间。一种方法是使用算法7.1中入口

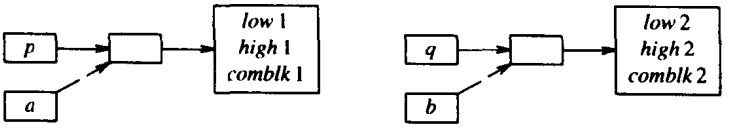
语句的 *parent* 域指向一个新表中的记录，该记录包括三个域：*low*、*high* 和 *comblk*。由于该表与符号表占据不相交的两个区域，所以我们可以知道指针指的是哪一个表。另一种方法是，符号表可以包含一个位用以指示某个名字是否为入口语句。如果空间的确很宝贵的话，另外一个算法可以避免消耗这额外的一位，进一步的编程技巧将在练习中加以讨论。

下面我们考虑替换图7-47中第(11)、(12)行的计算表达式。如图7-49a所示，分别由指针 *p* 和 *q*指向的入口语句对应的两个等价集合必须合并。表示这两个等价集合的数据结构如图7-49b所示。首先，我们必须检查确认两个等价集合中没有两个成员出现在 COMMON 中。即使它们都属于同一个块，Fortran 语言也禁止它们等价。如果任何一个 COMMON 块包含一个属于某一个等价集合的成员，则合并后的集合包含一个指向 *comblk* 中相应块记录的指针。完成这一检查工作的代码（假设 *q* 指向的入口语句成为合并后等价集合的入口语句），如图7-50所示。在图7-47中的第(11)、(12)行，我们必须计算合并后等价集合的外延。图7-49a给出了计算与 *q* 指向的入口语句相关的 *low* 和 *high* 的计算公式。 452



$$low = \min(low2,\ low1 - c + dist + d)$$
$$high = \max(high2,\ high1 - c + dist + d)$$

a)



b)

图7-49 合并等价集

a) 等价集合的相对位置 b) 数据结构

于是我们必须进行如下计算：

```
begin
    low(parent(q)) := min(low(parent(q)),
    low(parent(p))-c+dist+d);
    high(parent(q)) := max(high(parent(q)),
    high(parent(p))-c+dist+d)
end
```

这些语句置于图7-47中第(11)、(12)行语句后，用来反映两个等价集合的合并。

```
begin
    comblk1 := comblk(parent(p));
    comblk2 := comblk(parent(q));
    if comblk1 ≠ null and comblk2 ≠ null then
        error;    /* COMMON中的两个名字等价 */
    else if comblk2 = null then
        comblk(parent(q)) := comblk1
end
```

图7-50 计算 COMMON 块

为了使得算法7.1适用于 Fortran，还必须做最后两件事情。在 Fortran 中，我们可以令一个

数组的中间位置与其他数组中的某个位置或者简单名字等价。数组 A 的相对于它的入口语句的偏移指的是 A 的第一个位置相对于入口语句的第一个位置的偏移。如果位置 A(5,7) 等价于 B(20)，我们必须计算 A(5,7) 相对于A(1,1)的位置，图7-47中第(2)行将 *c* 初始化为这段距离的负值。同样，*d* 必须被初始化为 B(20) 相对于 B(1) 的距离的负值。8.3节的公式以及数组A和 B 中元素的大小足以计算 *c* 和 *d* 的初始值。

最后一点是 Fortran 允许涉及许多位置的 EQUIVALENCE 语句，比如

```
EQUIVALENCE (A(5,7), B(20), C, D(4,5,6))
```

它可以被分解为：

```
EQUIVALENCE (B(20), A(5,7))
EQUIVALENCE (C, A(5,7))
EQUIVALENCE (D(4,5,6), A(5,7))
```

注意，如果我们依照该顺序执行等价操作，只有 A 成为具有多于一个元素的集合的入口语句。一个带有 *low*、*high* 和 *comblk* 的记录可以被单个名字的等价集合使用多次。

7.9.4 映射数据区

现在我们描述为每个例程的名字分配各种数据区中的空间时所遵循的一些规则。

1. 对于每一个 COMMON 块，按照名字在块中的声明顺序进行访问（利用符号表中产生的 COMMON 名字列表）。依次为各个名字分配所需的字数的空间，并保持一个分配字数计数器，以便可以计算每一个名字的偏移。如果一个名字 A 被等价，则不论该等价集合的外延多大，我们必须检查 A 的入口语句的 *low* 值，使其不超出 COMMON 块的起始界。参考入口语句的 *high* 值为块中最后一个字确定一个底限。这些计算的精确公式留给读者思考。

453 ~ 454

2. 按任意顺序访问例程中的所有名字。

a) 如果一个名字出现在 COMMON 中，则什么也不做，空间已经在（1）中被分配。

b) 如果一个名字没有出现在 COMMON 中而且没有等价名字，在数据区中为该例程分配相应字数的空间。

c) 如果一个名字有等价名字，找到它的入口语句，设为 L。如果 L 已经在例程的数据区中被分配了位置，通过求A至入口语句的路径上所有偏移量的和来计算 A 的位置。如果 L 没有被分配位置，将数据区中接下来的 *high-low* 个字分配给该等价集合。L 在这些字中的位置距离起始处-*low* 个字，A 的位置可以像前面那样通过求偏移之和来确定。

练习

7.1 利用Pascal的作用域规则，确定图7-51中对应于名字 a、b 的每次出现的声明。该程序的输出由整数1到4组成。

7.2 考虑一个块结构的语言，其中一个名字可以被声明为整型或者实型。假设表达式用终结符 **expr** 表示，而且语句只包括赋值、条件、while 和

```
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.
```

图7-51 带有a、b 的多个声明的 Pascal 程序

顺序语句。又假设整型占用一个字，实型占用两个字。给出一个语法制导算法（基于声明和块的合理文法）以确定从名字到可以被块的活动记录所使用的字的绑定。你的存储分配使用的是可以满足块的任何执行的最少的字数吗？ 455

* 7.3 在7.4节中我们说明了如果每个深度为 i 的过程都在活动的开始处保存 $d[i]$，并在活动结束时恢复 $d[i]$，display 表将正确保持。试对调用次数进行归纳，证明每个过程都能得到正确的 display 表项。

7.4 宏是过程的一种形式，是通过将过程的每一次调用用宏按字面进行替换来实现的。图7-52给出了一个 Pic 程序及其输出。前两行定义了宏 show 和 small。宏的过程体的包含在两个%之间。图中的四个圆都是用宏 show 画出的，圆的半径通过非局部变量 r来给出。Pic 中的程序块用[和]括了起来。分配给这个程序块的每个变量都在块中隐式地声明。根据此输出，你能指出 r 的每次出现的作用域吗？

```
define show  % { circle radius r at Here } %
define small % [ r = 1/12; show ] %
[
    r = 1/6;
    show; small;
    move;
    show; small;
]
```



图7-52 Pic 程序画出的圆

7.5 写一个过程，通过传递指向链表头的指针来向链表中插入一项。哪种参数传递机制能够正确实现该过程？

7.6 假设采用以下参数传递机制：(a) 传值调用，(b) 引用调用，(c) 复制-恢复，(d) 传名调用，图7-53中程序显示的分别都是什么？

7.7 在词法作用域语言中，如果过程被作为参数传递，它的非局部环境可以用一个访问链来传递。试给出一个确定该访问链的算法。

```
program main(input, output);
    procedure p(x, y, z);
        begin
            y := y + 1;
            z := z + x;
        end;
    begin
        a := 2;
        b := 3;
        p(a + b, a, a);
        print a
    end.
```

图7-53 说明参数传递的伪码程序

7.8 图7-54中的 Pascal 程序阐述了与作为参数传递的过程相关的三种环境——*词法环境*、*传递环境*和*活动环境*。过程的这三个环境分别由定义过程、作为参数传递过程和活动过程中标识符的绑定构成的。注意函数f在第(11)行被作为参数传递。

使用 f 的词法环境、传递环境和活动环境，第(8)行的非局部变量m分别出现在第(6)、(10)和(3)行中 m 的声明的作用域中。

a) 画出该程序的活动树。

b) 使用 f 的词法环境、传递环境和活动环境，该程序的输出分别是什么？

* c) 当激活一个作为参数被传递的过程时，修改词法作用域语言中 display 表的实现方式，以正确建立词法环境。

```
(1) program param(input, output);
(2)    procedure b(function h(n: integer): integer);
(3)       var m : integer;
(4)       begin m := 3; writeln(h(2)) end { b };
(5)    procedure c;
(6)       var m : integer;
(7)          function f(n : integer) : integer;
(8)             begin f := m + n end { f };
(9)          procedure r;
(10)            var m : integer;
(11)            begin m := 7; b(f) end { r };
(12)      begin m := 0; r end { c };
(13)   begin
(14)      c
(15)   end.
```

图7-54 词法环境、传递环境和活动环境的例子

* 7.9 图7-55的伪码程序中，第(11)行的语句 $f := a$ 调用函数 a，它将函数 *addm* 作为结果返回。

a) 画出该程序执行的活动树。

b) 假设非局部变量用于词法作用域。为什么当采用栈式存储分配时会导致该程序执行失败？

c) 采用堆式分配时，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.
```

图7-55 函数 *addm* 作为结果返回的伪码程序

* 7.10 某些语言，如 Lisp，可以在运行时返回新产生的过程。在图7-56中，所有函数，无论是在源程序正文中定义的还是运行时产生的，都至多接受一个参数，并返回一个值：函数或者实数。操作符“∘”代表函数的合成，即 $(f \circ g)(x) = f(g(x))$。

a) 主程序 *main* 的显示值是什么？

456 ~ 458

* b) 假设无论何时过程 p 被创建或返回，它的活动记录都成为返回 p 的函数的活动记录的子记录。p 的传递环境可以通过保存一个活动记录树而不是使用栈来维护。当a由图7-56中的 *main* 计算时活动记录树是什么？

* c) 假设激活 p 时建立 p 的活动记录，并令其作为调用 p 的过程的活动记录的子记录。这种方法可用来维持 p 的活动环境。画出执行 *main* 中语句时的活动记录及它们之间的父子关系。使用这种方法时，栈是否足以用来保存活动记录？

7.11 另一种从散列表中删除名字（其作用域已经被穿过，如7.6节所述）的方法是将过期的名字保存在列表中，直到列表被再次搜索。假设表项中包含声明所在的过程的名字，我们就可以知道一个名字是否过期，如果过期就删除它。试给出一个过程的索引方式，使之能在 $O(1)$ 时间内得出一个过程是否过期，即已经经过其作用域。

7.12 许多散列函数可以用一个整数序列 $\alpha_0, \alpha_1, \cdots$ 来刻画。如果 c_i（$1 \leqslant i \leqslant n$）是字符串 s

中的第 i 个字符的整数值，则该字符串被散列到如下位置：

$$hash(s) = (\alpha_0 + \sum_{i=1}^{n} \alpha_i c_i) \bmod m$$

其中m是散列表的大小。对于下列情况，确定常数序列$\alpha_0, \alpha_1, \cdots$，或者证明其不存在。每一种情况确定一个整数，散列值即由该整数模 m 获得。

a) 取所有字符的和。

b) 取第一个与最后一个字符的和。

c) 取h_n，其中$h_0 = 0$, $h_i = 2h_{i-1} + c_i$。

d) 将中间四个符号作为一个32位整数。

e) 一个32位整数可以看作是由4个字节组成，每个字节代表一个数字，该数字有256种可能的值。从0000开始，对$1 \leqslant i \leqslant n$，把 c_i 加到第 i **mod** 4个字节，并且允许进位。例如，c_1 和c_5 被加给第一个字节，c_2 和c_6 加给第二个字节，依此类推。返回最终值。

```
function f (x : function);
var   y : function;
          y := x ∘ h;    /* 执行时创建y   */
          return y
end { f };

function h ();
      return sin
end { h };

function g (z : function);
var   w : function;
          w := arctan ∘ z;    /* 执行时创建w   */
          return w
end { g };

function main ();
var   a : real;
          u, v : function;
          v := f (g);
          u := v ();
          a := u (π/2);
          print a
end { main }.
```

图7-56 运行时生成函数的伪码程序

* 7.13 如果输入由连续的字符串（如v000, v001, …）组成，为什么练习7.12中由一系列整数刻画的散列算法执行效果会不好？症状是在某处它们的行为偏离随机数并且是可预测的。

** 7.14 如果 n 个字符串被散列到 m 个列表中，则无论字符串的分布多么不均匀，每个列表的平均串数都将是 n/m。假设 d 是一种分布，即一个随机的字符串被放在第 i 个列表中的概率为 $d(i)$。假设分布为 d 的散列函数将随机选择的 b_j 个串放在列表 j 中，$0 \leqslant j \leqslant m-1$。证明期望值 $W = \sum_{j=0}^{m-1} (b_j)(b_j+1)/2$ 与分布 d 的变化成线性关系。在均匀分布情况下，证明期望值 W 为 $(n/2m)(n+2m-1)$。

7.15 假设一个 Fortran 程序中有以下声明序列。

```
SUBROUTINE SUB(X,Y)
INTEGER A, B(20), C(10,15), D, E
COMPLEX F, G
COMMON /CBLK/ D, E
EQUIVALENCE (G, B(2))
EQUIVALENCE (D, F, B(1))
```

指出 SUB 和CBLK的数据区的内容（至少从 SUB 可以访问 CBLK 的部分数据）。其中为什么没有 X 和 Y?

459 ~ 460

* 7.16 用于等价计算的一种非常有用的数据结构是环状结构。在每一个符号表表项中使用一个指针和一个偏移域来链接等价集合中的成员。该结构如图7-57所示，其中 A、B、C、D 等价，E、F 等价，B 的位置在 A 的位置之后20个字处，以此类推。

a) 给出一个计算 X 相对于 Y 的偏移的算法，假设 X 和 Y 在同一个等价集合内。

b) 给出一个算法计算相对于某个名字 Z 的 *low* 和 *high* 值，其定义参见7.9节。

c) 给出一个算法处理如下语句：

```
EQUIVALENCE U, V
```

U，V 不一定在不同的等价集合内。

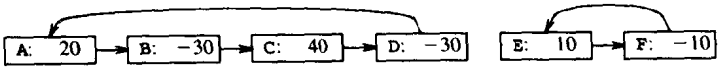


图7-57 环状结构

* 7.17 7.9节给出的映射数据区算法需要我们检验 A 的等价集的入口语句的 *low* 值没有超出COMMON 块的起始边界，而且如果需要的话，需要重新计算 A 的入口语句的 *high* 值来增大 COMMON 块的上限。根据 *next*（COMMON 块中 A 的偏移）和 *last*（块中最后的字），给出一个公式来测试和更新 *last*（如果需要的话）。

参考文献注释

在递归函数的实现中，栈扮演了举足轻重的角色。McCarthy[1981, p.178]中回顾了从1958年开始的实现Lisp的项目中，栈被用来保存递归程序中变量的值，以及子程序的返回地址。Algol 60中对块和递归过程的引入（详细过程参见Naur[1981,Section 2.10]）也促进了栈式分配技术的发展。使用display表描述词法作用域语言中访问非局部变量的思想参见Dijkstra [1960, 1963]。尽管Lisp使用动态作用域，它也可能使用由函数和一个访问链组成的“funargs”来达
461 到词法作用域效果。McCarthy[1981]描述了这种机制的发展。Lisp的后续版本，如公用Lisp（Steele[1984]）已经从动态作用域中脱离了出来。

对于名字绑定的解释可以从程序设计语言教科书中找到，参见Abelson and Sussman[1985]、Pratt[1984]或者Tennent[1981]。第2章中提出的另外一种方法，是读取编译器的描述。在Kernighan and Pike[1984]中，从算术表达式的计算器到包含递归过程的简单语言的解释器都有详细叙述。关于栈式分配、display 表的使用、数组的动态分配的详细叙述，请参见Randell and Russell[1964]。

Johnson and Ritchie[1981]中讨论了参数数目可变的调用序列的设计。设置全局display表的一般方法是沿着访问链，在处理过程中设置 display 中的元素。7.4节中的方法仅涉及到一个元素，在一段时间内已经很“有名”了，参见 Rohl[1975]。Moses[1970]中讨论了将函数作为参数传递的环境的差异，而且指出如果这种情况采用浅访问和深访问会产生的问题。栈式分配不能用于有协同例程或多进程的语言。Lampson[1982] 考虑了使用堆式分配的快速实现方法。

在数理逻辑中，Frege[1879] 的 Begriffsschrift中提出了限制作用域和代换的量词。代换与参数传递很长时间以来成为数理逻辑和程序设计语言领域比较热门的主题。Church[1956, p.288]中指出“函数型变量的代换规则的正确表述是特别困难的问题”，而且将这种规则的研究与命题演算联系起来。Church 的 λ 演算[1941]已经被应用到了程序设计语言环境中，例如Landin[1964]。由函数和访问链组成的序对通常被称作闭包，参见Landin[1964]。

符号表的数据结构与搜索符号表的算法在Knuth[1973b]和Aho, Hopcroft, and Ullman[1974, 1983]中有详细的讨论。散列的核心技术参见Knuth[1973b]和Morris[1968b]。最初讨论散列的文献为Peterson[1957]。关于符号表组织技术的更多信息参见McKeeman[1976]。例7.10摘自Bentley, Cleveland, and Sethi[1985]。Reiss[1983]中描述了一个符号表生成器。

等价算法在Arden, Galler, and Graham[1961]和Galler and Fischer[1964]描述；我们已经采用了后一种方法。关于等价算法效率问题的讨论参见Fischer[1972]、Hopcroft and Ullman[1973]
463 和Tarjan[1975]。

第8章 中间代码生成

在编译器的分析综合模型中，前端将源程序翻译成一种中间表示，后端根据这个中间表示生成目标代码。目标语言的细节要尽可能限制在后端。尽管源程序可以直接翻译成目标语言，但使用与机器无关的中间形式具有如下优点：

1. 再目标比较容易：不同机器上的编译器可以在已有前端的基础上附加一个适合这台新机器的后端来生成。

2. 可以在中间表示上应用与机器无关的代码优化器。第10章会详细讨论这种优化器。

本章说明如何使用第2章及第5章讨论的语法制导方法将源程序翻译成中间形式的编程语言结构，如声明、赋值及控制流语句。为简单起见，我们假定源程序已经进行过语法分析及静态检查，如图8-1中所示的组织结构。本章中大部分的语法制导定义都可由第5章介绍的自底向上或自顶向下语法分析技术实现，如果需要的话，可以将中间代码生成阶段并入语法分析阶段中。

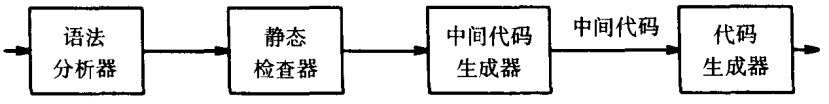


图8-1 中间代码生成器的位置

8.1 中间语言

5.2节与2.3节介绍的语法树和后缀表示就是两种中间表示。本章将使用第三种中间表示，称为三地址码。从通用的编程语言结构生成三地址码的语义规则与构建语法树及生成后缀表示相似。

8.1.1 图表示

语法树描述了源程序的自然层次结构。dag以更紧凑的形式给出了相同的信息，因为dag中公共子表达式已被识别出来。赋值语句`a := b*-c + b*-c` 的语法树及dag如图8-2所示。

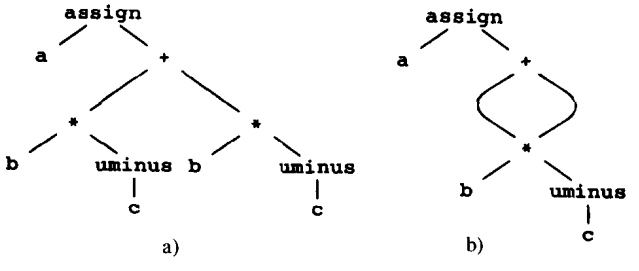


图8-2 `a := b*-c + b*-c`的图表示

a) 语法树 b) Dag

后缀表示是语法树的线性化表示；它是树中节点的列表，每个节点紧跟在其子节点后面出现。图8-2a中的语法树的后缀表示为

```
a b c uminus * b c uminus * + assign
```

(8-1)

语法树中的边不会显式地出现在后缀表示中。它可以根据节点出现的顺序及节点上的操作符所要求的操作数的个数来恢复。边的恢复类似于使用栈对后缀表示形式的表达式求值。详见2.8节及后缀表示和栈式机器代码之间的关系。

赋值语句的语法树由图8-3中的语法制导定义生成；它是对5.2节中相应部分的扩展。非终结符 S 生成赋值语句。两个二元操作符 + 和 * 是典型语言全部操作符集中的两个例子。操作符的结合性及优先级和通常的一样，即使文法中没有包括也如此。该定义由输入 `a := b*-c + b*-c` 构建了图8-2a的语法树。

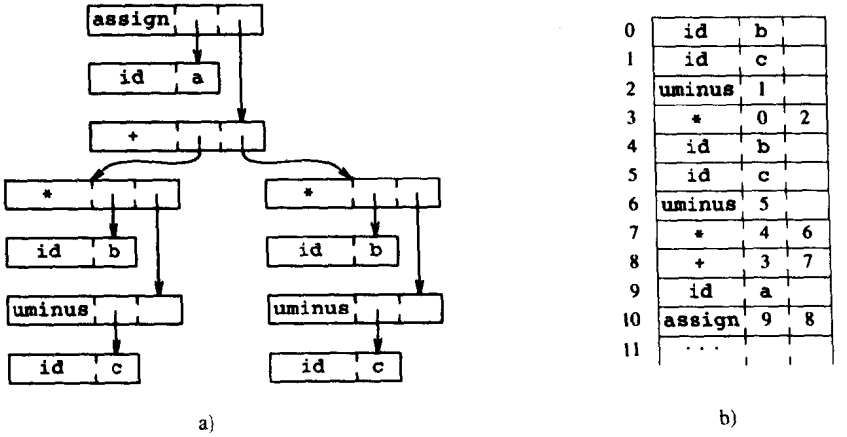
464

产生式	语义规则
$S \rightarrow$ **id** := E	$S.nptr$:= *mknode*('assign', *mkleaf*(**id**, **id**.*place*), $E.nptr$)
$E \rightarrow E_1$ + E_2	$E.nptr$:= *mknode*('+', $E_1.nptr$, $E_2.nptr$)
$E \rightarrow E_1$ * E_2	$E.nptr$:= *mknode*('*', $E_1.nptr$, $E_2.nptr$)
$E \rightarrow$ - E_1	$E.nptr$:= *mkunode*('uminus', $E_1.nptr$)
$E \rightarrow$ (E_1)	$E.nptr$:= $E_1.nptr$
$E \rightarrow$ **id**	$E.nptr$:= *mkleaf*(**id**, **id**.*place*)

图8-3 生成赋值语句语法树的语法制导定义

如果函数 *mkunode*(*op*, *child*) 和 *mknode* (*op*, *left*, *right*) 尽可能返回指向现存节点的指针，而不是构建一个新的节点的话，那么同样的语法制导定义将生成如图8-2b所示的 dag。符号 **id** 有一个属性*place*指向标识符在符号表的表项。在8.3节中，我们说明如何通过属性 **id**.*name* 找到符号表表项，它表示与那个 **id** 的出现相关联的词素 (lexeme)。如果词法分析器用单个字符数组中保存了所有的词素，则属性 *name* 可能就是词素的第一个字符的索引。

图8-2a中语法树的两种表示如图8-4所示。每个节点表示为一个记录，记录中有一个域表示操作符，一个附加的域表示指向其子节点的指针。在图8-4b中，节点从记录数组中分配，并且该节点的位置或索引作为指向该节点的指针。从位置为10的根节点开始，沿着指针可以访问到语法树中所有的节点。



a) b)

465

图8-4 图8-2a中语法树的两种表示

8.1.2 三地址码

三地址码是下列一般形式的语句序列：

x := y *op* z

其中，x，y及z是名字、常量或编译器生成的临时变量；*op* 代表任何操作符，例如定点或者浮

点算术操作符，或者是布尔型变量的逻辑操作符。注意这里不允许组合的算术表达式，因为语句右边只有一个操作符。这样，像 x + y * z 这样的源语言表达式应该被翻译成如下序列：

```
t1 := y * z
t2 := x + t1
```

t_1 与 t_2 是编译器生成的临时变量。这种复杂算术表达式及嵌套控制流语句的拆解使得三地址码适合于目标代码生成及优化（见第10章及第12章）。由程序计算出来的中间值的名字的使用使得三地址码容易被重排列——而不像后缀表示那样。

三地址码是语法树或dag的线性表示，在三地址码中，显式名字对应于图的一个内节点。图8-2中的语法树和dag由图8-5中的三地址码序列表示。变量名字可以直接出现在三地址语句中，所以图8-5a中没有与图8-4中的叶子相对应的语句。

```
t1 := - c                    t1 := - c
t2 := b * t1                 t2 := b * t1
t3 := - c                    t5 := t2 + t2
t4 := b * t3                 a  := t5
t5 := t2 + t4
a  := t5
```

a) b)

图8-5 与图8-2中的树和 dag 相对应的三地址码

a) 语法树的代码 b) dag 的代码

三地址码的得名原因是每条语句通常包含三个地址，两个是操作数地址，一个是结果地址。本节后边给出的三地址码的实现中，由程序员定义的名字被一个指向该名字的符号表表项的指针所代替。 466

8.1.3 三地址语句的类型

三地址语句与汇编代码类似。语句可以有符号标号，而且还有控制流语句。符号标号表示三地址语句在保存中间代码的数组中的索引。通过一遍单独的扫描，或使用8.6节中讨论的“回填”（backpatching），可用实际的索引替换符号标号。

下面是本书后面使用的通用三地址语句：

1. 形如 x :=y *op* z 的赋值语句，其中，*op* 是二元算术操作符或逻辑操作符。

2. 形如 x := *op* y 的赋值指令，其中 *op* 是一元操作符。基本的一元操作符包括一元减、逻辑非、移位操作符及转换操作符，例如，将定点数转换为浮点数。

3. 形如 x := y 的复制语句，将 y 的值赋给 x。

4. 无条件跳转语句 goto L。接下来将执行标号为L的三地址语句。

5. 形如 if x *relop* y goto L 的条件跳转语句。这条指令对 x 和 y 应用逻辑操作符（<，=，>=等），如果 x 与 y 满足 *relop* 关系则执行带有标号L的语句。如果不满足，紧接着执行 if x *relop* y goto L 后面的语句，与通常的顺序一样。

6. 过程调用 param x 和 call p,*n* [⊖] 及 return y，其中 y 代表一个返回值，是可选的。它们的典型应用如下面的三地址语句序列：

```
param x1
param x2
```

⊖ call p,*n* 是一个过程调用，“,”分割调用参数。——译者注

```
...
param x_n
call p,n
```

作为过程 $p(x_1, x_2, ..., x_n)$的一部分生成。整数n表示调用“call p, n”中的实际参数的个数，它不是多余的因为调用可以被嵌套。过程调用的实现将在8.7进行略述。

7. 形如 x := y[i] 及 x[i] := y 的被变址的赋值语句。第一个表达式将 x 的值设置为地址y 之后第 i 个内存单元的值。表达式 x[i] := y 将地址 x 之后的第 i 个单元的值设置为y的值。在这两条指令中，x，y 及 i 表示数据对象。

8. 形如 x := &y，x := *y 及*x := y的地址及指针赋值语句。第一个表达式将 x 的值设
467 置成 y 的地址。假定 y 是一个名字或临时变量，表示一个具有左值的表达式，比如 A[i,j]，x是一个指针名或是一个临时变量。也就是说，x 的右值是某个对象的左值(地址)。在语句 x := *y中，假定 y 是一个指针或是一个右值是地址的临时变量。x 的右值被置成与 y 的内容相同。最后，*x := y，将 x 指向的对象的右值置成 y 的右值。

选择允许的操作符是设计中间形式的重要问题。显然操作符集合必须足够用来实现源语言的操作。小操作符集合在新的目标机上容易实现。然而，受限的指令集或许会强制前端为某些源语言的操作生成长的语句序列。要生成优质的代码，优化器及代码生成器的负担会更大。

8.1.4 语法制导翻译生成三地址码

生成三地址码以后，临时名字组成了语法树的内节点。$E \rightarrow E_1 + E_2$ 的左部的非终结符 E 的值将计算到一个新的临时变量 t 中。一般地，**id** := E 的三地址码由将 E 值存入临时变量 t 的代码组成，后边紧接着赋值 **id**.*place* := t。如果表达式是一个标识符，比如说 y，则 y 本身保存该表达式的值。这时，每次需要一个临时变量时我们就生成一个新的名字。临时变量的重用技术将在8.3节给出。

图8-6中的 S 属性定义生成赋值语句的三地址码。给定输入 a := b * - c + b * - c，它生成图8-5a的代码。综合属性 *S.code* 代表赋值S的三地址码。非终结符 E 具有两个属性：

1. *E.place*，保存 E 的值的名字。
2. *E.code*，计算 E 的三地址语句序列。

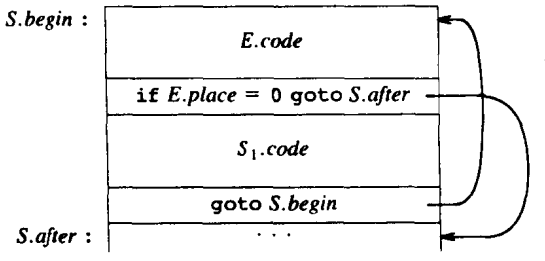
连续调用函数 *newtemp* 返回不重复的名字序列 $t_1, t_2, \cdots$。

产生式	语义规则
$S \rightarrow$ **id** := E	*S.code* := *E.code* ‖ *gen*(**id**.*place* ':=' *E.place*)
$E \rightarrow E_1 + E_2$	*E.place* := *newtemp*; *E.code* := E_1.*code* ‖ E_2.*code* ‖ *gen*(*E.place* ':=' E_1.*place* '+' E_2.*place*)
$E \rightarrow E_1 * E_2$	*E.place* := *newtemp*; *E.code* := E_1.*code* ‖ E_2.*code* ‖ *gen*(*E.place* ':=' E_1.*place* '*' E_2.*place*)
$E \rightarrow - E_1$	*E.place* := *newtemp*; *E.code* := E_1.*code* ‖ *gen*(*E.place* ':=' 'uminus' E_1.*place*)
$E \rightarrow (E_1)$	*E.place* := E_1.*place*; *E.code* := E_1.*code*
$E \rightarrow$ **id**	*E.place* := **id**.*place*; *E.code* := ''

图8-6 为赋值语句产生三地址码的语法制导定义

为方便起见，我们在图8-6中使用符号 *gen*(*x* ':=' *y* '+' *z*) 来表示三地值语句 *x* := *y* + *z*。当被传到 *gen* 中时，计算的是表达式，而不是像 *x*，*y*，*z* 那样的变量，而像'+'这样被引号括起来的操作符或操作数照字面含义接受。实际上，三地值码可能被送到输出文件中，而不是生成 *code* 属性。

通过使用像图8-7中 while 语句一样的产生式及语义规则，可以将控制流语句加入到图8-6中的赋值语言中。在图8-7中，生成*S*→**while** *E* **do** S_1 的代码使用了属性 *S.begin* 和 *S.after* 来分别标记 *E*的代码中的第一条语句及紧跟在 *S* 的代码后面的语句。这些属性代表由函数 *newlabel* 生成的标号，每次调用 *newlabel* 都返回一个新标号。注意，*S.after* 成为 while 语句代码后边的代码的标号。我们假定非零表达式表示真；也就是说，当表达式 *E* 的值变成零时，控制将离开 while 语句。



产生式	语义规则
$S \rightarrow$ **while** E **do** S_1	*S.begin* := *newlabel*; *S.after* := *newlabel*; *S.code* := *gen*(*S.begin* ':') ‖ *E.code* ‖ *gen*('if' *E.place* '=' '0' 'goto' *S.after*) ‖ S_1.*code* ‖ *gen*('goto' *S.begin*) ‖ *gen*(*S.after* ':')

图8-7 为 while 语句生成代码的语义规则

支配控制流的表达式通常是包含关系及逻辑操作符的布尔表达式。8.6节中 while 语句的语义规则与图8-7中的有所不同，它允许布尔表达式内具有控制流。

后缀表示可以通过应用图8-6（或参见图2-5）中的语义规则获得。标识符的后缀表示就是标识符本身。其他产生式的规则只是连接在操作数代码后边的操作符。例如，产生式 $E \rightarrow -E_1$ 的语义规则为：

E.code := E_1.*code* ‖ '**uminus** '

一般来说，通过对语义规则做相似的修改，本章中语法制导翻译产生的中间形式可以被修改。

8.1.5 三地址语句的实现

三地址语句是中间代码的一种抽象形式。在编译器中，这些语句可以以带有操作符和操作数域的记录来实现。四元式、三元式及间接三元式是三种这样的表示。

8.1.5.1 四元式

四元式是带有四个域的记录结构，即 *op*，*arg*1，*arg*2 及 *result*。*op* 域包含操作符的内码。三地址语句 `x := y op z` 通过将 `y` 放入 *arg*1，`z` 放入 *arg*2，并且将`x`放入 *result* 而表示为四元式。像 `x := -y` 或 `x := y` 这样的一元操作符语句不使用 *arg*2。像 `param` 这样的操作符不使用 *arg*2 及 *result*。条件及非条件跳转语句将目标标号存入 *result* 域。图8-8a是赋值语句 `a := b*-c + b*-c`的四元式。它们是从图8-5a中的三地址码获得的。

	op	arg 1	arg 2	result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

a)

	op	arg 1	arg 2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

b)

图8-8 三地址语句的四元式及三元式表示

a) 四元式 b) 三元式

*arg*1，*arg*2 及 *result* 域的内容正常情况下是指向这些域所代表的名字在符号表表项的指针。这样的话，临时名字在生成时一定要被填入符号表。

468
~
470

8.1.5.2 三元式

为了避免将临时名字填入符号表中，我们可以通过计算临时值的语句的位置来引用它。如果这样做，三地址语句就可以用只包含三个域的记录表示：*op*，*arg*1及*arg*2，如图8-8b所示。操作符*op*的操作数，即域*arg*1和*arg*2，或者是指向符号表（对于程序员定义的名字或常量）的指针或者是指向三元式结构（对于临时变量）的指针。由于使用了三个域，这种中间代码形式被称为三元式[⊖]。除了对程序员定义的名字的处理之外，三元式与以节点数组表示的语法树或dag相对应，如图8-4所示。

带括弧的数字表示指向三元式结构的指针，而符号表指针由名字本身代表。实际上，解释 *arg*1 及 *arg*2 中不同种类条目所需要的信息可以被编码到 *op* 域或一些附加域中。图8-8b的三元式与图8-8a的四元式相对应。注意，通过将 a 放入 *arg*1 域并使用操作符 assign 可以将复制语句a := t_5 编码为三元式表示。

形如x[i] := y这样的三元算符在三元式结构中需要两个表项，如图8-9a所示，而x := y[i]自然地表示为图8-9b中的两个操作。

	op	arg 1	arg 2
(0)	[]=	x	i
(1)	assign	(0)	y

a)

	op	arg 1	arg 2
(0)	=[]	y	i
(1)	assign	x	(0)

b)

图8-9 更多三元式表示

a) x[i] := y b) x := y[i]

8.1.5.3 间接三元式

已经讨论过的三地址码的另外一个实现是列出指向三元式的指针，而不是列出三元式本身。这种实现方式很自然的被称为间接三元式。

譬如，让我们用数组 *statement* 按要求的顺序列出指向三元式的指针，那么图8-8b的三元式可以表示为图8-10。

	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	arg 1	arg 2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

图8-10 三地址语句的间接三元式表示

⊖ 有些人将三元式称为“二地址码”，而将“四元式”称为“三地址码”。但我们认为“三地址码”是一个抽象概念，可以有不同的实现，主要是三元式和四元式。

8.1.6 表示方法比较：间址的使用

三元式与四元式的差异可以看作在表示中引入了多少间址。当我们最终生成目标代码时，每个名字，不管是临时变量还是程序员定义的变量，都将被分配一些运行时内存地址。该地址将被保存到数据所在的符号表表项中。使用四元式表示，定义或使用临时变量的三地址语句可通过符号表直接访问该临时变量的地址。

使用四元式的一个更重要的好处体现在优化编译器中，此时语句经常被移来移去。使用四元式表示，符号表在值的计算及使用之间提供了一次额外间址。如果我们移动计算x的语句，使用x的语句不需要改变。然而，在三地址符号中，移动一条定义临时值的语句需要我们改变在*arg*1及*arg*2数组中所有对该语句的引用。这个问题使得三元式很难用在编译优化中。

间接三元式没有这样的问题。可以通过对 *statement* 列表的重新排序来移动语句。由于指向临时值的指针查阅 *op-arg*1-*arg*2 数组，该数组不变，这些指针也不需要改变。因此，就其效用而言，间接三元式看上去与四元式非常相似。两种表示需要大约相同的存储空间，并且对代码重新排序的效率相同。对于普通的三元式，必须将对那些临时变量的存储分配推迟到代码生成阶段。然而，如果相同临时值的使用超过一次，则间接三元式与四元式相比可以节省空间，原因是在语句数组中的两条或多条可以指向 *op-arg*1-*arg*2 结构的同一行。例如，可以将图8-10中的第(14)及(16)行结合，然后可以再把第(15)与(17)行结合。

471 ~ 472

8.2 声明语句

当过程或程序块内的声明序列被考查之后，我们可以为局部于该过程的名字分配存储空间。对每个局部名字，我们都将在符号表中创建一个表项，并填写类型及名字的相对存储地址等相关信息。相对地址是指相对于静态数据区基址或活动记录中局部数据域基址的偏移量。

当前端生成地址时，或许已经知道了目标机。假定在字节寻址的机器中，连续整数的地址相差4，则由前段生成的地址计算或许就要乘以4。目标机的指令集或许会偏爱数据对象的某种布局，这也导致地址的相同布局。我们在此忽略数据对象的对齐；例7.3说明了两个编译器如何对数据对象进行对齐。

8.2.1 过程中的声明语句

像C、Pascal和Fortran这样的语言的文法允许将单个过程中所有的声明语句作为一个组来处理。在这种情况下，一个全局变量，譬如 *offset*，可以跟踪下一个可用的相对地址。

在图8-11的翻译模式中，非终结符 *P* 产生一系列形如 **id**:*T* 的声明语句。在考虑第一个声明之前，将 *offset* 置为零。以后每次遇到一个新的名字，就将该名字填入符号表，并将其偏移置为当前的 *offset* 值，然后将该名字所代表的数据对象的宽度加到 *offset* 上。

过程 *enter* (*name* , *type* , *offset*) 为名字建立符号表表项，并给出该名字的类型 *type* 及其在过程数据区中的相对地址 *offset*。我们使用综合属性 *type* 及 *width* 说明非终结符 *T* 的类型及宽度，或该类型的对象所占用的内存单元数。属性 *type* 表示通过应用类型构造器 *pointer* 或 *array* 从基本类型 *integer* 及 *real* 构造出来的类型表达式，正如6.1节所述。如果用图来表示类型表达式，则属性 *type* 可能是指向代表一个类型表达式的节点的指针。

在图8-11中，整数的宽度为4而实数为8。数组的宽度可以通过将数组中每个元素的宽度乘以数组中元素的个数获得[⊖]。每个指针的宽度假定为4。在Pascal及C中，指针可能在我们知道

473

⊖ 对于那些下界不是0的数组，如果添入符号表的偏址按8.3节中讨论的调整的话，数组元素的地址计算就很简单了。

其所指对象的类型之前遇到（见6.3节对递归类型的讨论）。如果所有的指针具有相同的宽度，对该类型的存储分配将变得更简单。

$P \rightarrow D$	{ *offset* := 0 }
$D \rightarrow D\ ;\ D$	
$D \rightarrow$ **id** : T	{ *enter*(**id**.*name*, T.*type*, *offset*); *offset* := *offset* + T.*width* }
$T \rightarrow$ **integer**	{ T.*type* := *integer*; T.*width* := 4 }
$T \rightarrow$ **real**	{ T.*type* := *real*; T.*width* := 8 }
$T \rightarrow$ **array** [**num**] **of** T_1	{ T.*type* := *array*(**num**.*val*, T_1.*type*); T.*width* := **num**.*val* × T_1.*width* }
$T \rightarrow \uparrow T_1$	{ T.*type* := *pointer*(T_1.*type*); T.*width* := 4 }

图8-11 计算声明语句中名字的类型和相对地址

在图8-11的翻译模式中，如果某一行中第一个产生式形式如下，则 *offset* 的初始化将变得更明显，

$$P \rightarrow \{\ offset := 0\ \}\ D \tag{8-2}$$

非终结符产生的ϵ，5.6节称为的标记非终结符，可以被用来重写上述产生式以便所有的动作都出现在产生式右部的末 端。使用标记非终结符 M，式(8-2)可被重写为

$$P \rightarrow M\ D$$
$$M \rightarrow \epsilon \qquad \{\ offset := 0\ \}$$

8.2.2 跟踪作用域信息

在允许嵌套过程的语言中，局部于每个过程的名字可以使用图8-11中的方法分配相对地址。当看到嵌套的过程时，应暂时挂起对外围过程声明语句的处理。这种方法可以通过为如下语言增加语义规则来说明。

474

$$P \rightarrow D$$
$$D \rightarrow D\ ;\ D \mid \textbf{id} : T \mid \textbf{proc id}\ ;\ D\ ;\ S \tag{8-3}$$

产生语句的非终结符S及产生类型的非终结符 T 的产生式没有说明，因为我们考虑的是声明。非终结符 T 具有综合属性 *type* 及 *width*，同图8-11中的翻译模式相同。

为简单起见，假定语言(8-3)中每个过程有一张单独的符号表。符号表的一个可能的实现是名字表项的一个链表。如果需要可用更聪明点的实现方法。

当遇到过程声明 $D \rightarrow$ **proc id** D_1; S 时便创建一张新的符号表，并在此符号表中为 D_1 中的声明创建相应的表项。新表有一个指针指回外围过程的符号表；由 **id** 代表的名字本身是该外围过程的局部名字。与图8-11的变量声明处理方式惟一不同的是要告诉过程 *enter* 在哪个符号表中添加表项。

例如，图8-12给出了五个过程的符号表。过程的嵌套结构可以从符号表之间的连接推导出来；程序见图7-22。过程 `readarray`，`exchange` 及 `quicksort` 的符号表指向其包含过程

sort 的符号表，sort 由整个程序组成。由于 partition 在 quicksort 中声明，它的符号表指向 quicksort 的符号表。

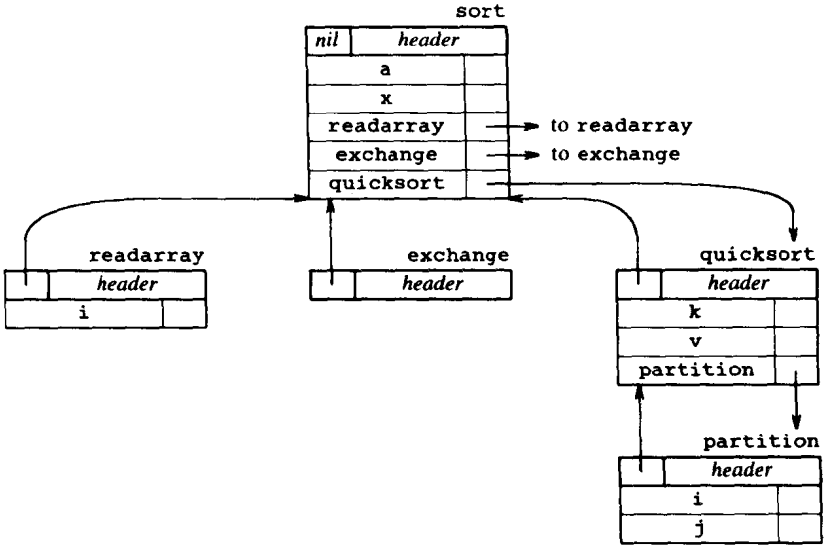


图8-12 嵌套过程的符号表

475

语义规则是利用以下操作定义的：

1. *mktable* (*previous*) 创建一张新的符号表，并返回指向新表的指针。参数 *previous* 指向先前创建的符号表，可以推测，是外围进程的符号表。指针 *previous* 放在新符号表的表头，表头中还可以放一些其他信息，如过程的嵌套深度。我们也可以按过程声明的顺序为其编号，并将编号保存在表头中。

2. *enter* (*table*, *name*, *type*, *offset*) 在 *table* 指向的符号表中为名字 *name* 建立新表项。同时，*enter* 将类型 *type* 及相对地址 *offset* 放入该表项的域中。

3. *addwidth* (*table*, *width*)将 *table* 指向的符号表中所有表项的宽度和记录在与符号表关联的表头中。

4. *enterproc* (*table*, *name*, *newtable*) 在 *table* 指向的符号表中为过程 *name* 建立一个新表项。参数 *newtable* 指向过程 *name* 的符号表。

通过使用栈 *tblptr* 保存指向外围过程符号表的指针，图8-13的翻译模式说明了一遍扫描如何布局数据。对于图8-12的符号表，当考虑 partition 中的声明语句时，*tblptr* 将包含指向 sort, quicksort 及 partition 的符号表的指针。指向当前符号表的指针在栈顶。另一个栈 *offset* 是对图8-11中属性 *offset* 的嵌套过程的自然的一般推广。*offset* 的栈顶元素是下一个当前过程中局部对象可用的相对地址。

产生式中最后的动作 $action_A$ 开始之前，下面产生式的子树 *B* 和 *C* 中的所有语义动作都已经结束：

$A \rightarrow BC\ \{action_A\}$

因此，在图8-13中与标志 *M* 关联的动作首先被完成。

非终结符 *M* 的动作用最外层作用域的符号表初始化栈 *tblptr*，该符号表由 *mktable*(*nil*) 操作创建。该动作还将相对地址0压入栈 *offset*。当出现一个过程声明时，非终结符 *N* 扮演相同

的角色。其动作使用 *mktable*(*top*(*tblptr*)) 操作来创建一个新的符号表。此时，参数 *top*(*tblptr*) 给出了该新表的外围作用域。指向新表的指针被压入到栈中指向该外围作用域的指针的上面。同样，0被压入栈 *offset*。

$P \rightarrow M\ D$	{ *addwidth*(*top*(*tblptr*), *top*(*offset*)); *pop*(*tblptr*); *pop*(*offset*) }
$M \rightarrow \epsilon$	{ *t* := *mktable*(*nil*); *push*(*t*, *tblptr*); *push*(0, *offset*) }
$D \rightarrow D_1\ ;\ D_2$	
$D \rightarrow$ **proc id** ; $N\ D_1\ ;\ S$	{ *t* := *top*(*tblptr*); *addwidth*(*t*, *top*(*offset*)); *pop*(*tblptr*); *pop*(*offset*); *enterproc*(*top*(*tblptr*), **id**.*name*, *t*) }
$D \rightarrow$ **id** : T	{ *enter*(*top*(*tblptr*), **id**.*name*, *T.type*, *top*(*offset*)); *top*(*offset*) := *top*(*offset*) + *T.width* }
$N \rightarrow \epsilon$	{ *t* := *mktable*(*top*(*tblptr*)); *push*(*t*, *tblptr*); *push*(0, *offset*) }

图8-13 处理嵌套过程中的声明语句

对于每个变量声明 **id**:*T*，在当前符号表中建立该 **id** 的表项。这个声明不改变栈 *tblptr*，但栈*offset* 的栈顶指针增加 *T.width*。当产生式 $D \rightarrow$**proc id**; $N\ D_1$; S 右边的动作执行时，由 D_1 产生的所有声明出现在 *offset* 的栈顶；它使用过程 *addwidth* 来记录。然后，弹出栈 *tblptr* 及 *offset* 的栈顶元素，下面我们转回到外围过程中继续处理其中的声明。此时将被包含过程的名字添入外围过程的符号表中。

8.2.3 记录中的域名

下面的产生式允许非终结符 *T* 除产生基本类型、指针和数组之外，还生成记录：

$T \rightarrow$ **record** D **end**

图8-14中翻译模式的动作中强调作为语言结构的记录的布局与活动记录之间的相似性。由于在图8-13中过程定义并不影响宽度计算，上面的产生式也允许过程定义出现在记录中，不过我们忽略这种情况。

$T \rightarrow$ **record** $L\ D$ **end**	{ *T.type* := *record*(*top*(*tblptr*)); *T.width* := *top*(*offset*); *pop*(*tblptr*); *pop*(*offset*) }
$L \rightarrow \epsilon$	{ *t* := *mktable*(*nil*); *push*(*t*, *tblptr*); *push*(0, *offset*) }

图8-14 为记录中的域名建立符号表

476
~
477

当看到关键字 **record** 之后，与标记*L*关联的动作为域名创建一个新的符号表。指向该符号表的指针被压入栈 *tblptr* 且相对地址0被压入栈 *offset*。因此图8-13中产生式 $D \rightarrow$**id**:*T* 的动作是将域名 **id** 的信息填入到该记录的符号表中。进而，当记录的所有域都被检查过之后，*offset* 的栈顶将保存记录中所有数据对象的宽度。在图8-14中 **end** 之后的动作将该宽度作为综合属性 *T.width* 返回。类型 *T.type* 通过将构造器 *record* 应用于指向该记录符号表的指针获得。该指针将在下一节被用来恢复记录中的域名、类型及域宽等。

8.3 赋值语句

在本节中，表达式的类型可以是整型、实型、数组及记录。作为将赋值语句转换为三地址码的一部分，我们将说明如何在符号表中查找名字以及如何存取数组及记录中的元素。

8.3.1 符号表中的名字

在8.1节，我们使用名字本身形成三地址语句，条件是名字代表指向其符号表表项的指针。图8-15的翻译模式说明了如何找到这些符号表的表项。由 **id** 表示的名字的词素由属性 **id**.*name* 给出。操作 *lookup*(**id**.*name*) 检查符号表中是否出现该名字的表项。如果有，返回指向该表项的指针；否则，*lookup* 返回 *nil*，表示没有找到这样的表项。

图8-15中的语义动作使用过程 *emit* 来将三地址语句输出到输出文件中，而不是如图8-6那样为非终结符建立 *code* 属性。从2.3节起，翻译可以通过向输出文件发送三地址语句来实现，只要产生式左部非终结符的 *code* 属性是由产生式右部非终结符的 *code* 属性按其出现顺序连接而成，当然非终结符之间也许附加一些字符串。

通过重新解释图8-15中的 *lookup* 操作，即使像 Pascal 语言那样将最接近嵌套作用域规则应用于非局部名字，亦可采用该翻译模式。更具体地讲，假设赋值语句出现的上下文由下列文法给定：

$P \rightarrow M\ D$
$M \rightarrow \epsilon$
$D \rightarrow D\ ;\ D \mid \textbf{id}\ :\ T \mid \textbf{proc id}\ ;\ N\ D\ ;\ S$
$N \rightarrow \epsilon$

将这些产生式加到图8-15中的那些产生式上，非终结符 P 就成为新的开始符号。 478

$S \rightarrow \textbf{id} := E$	{ *p* := *lookup*(**id**.*name*); **if** *p* ≠ *nil* **then** *emit*(*p* ':=' *E*.*place*) **else** *error* }
$E \rightarrow E_1 + E_2$	{ *E*.*place* := *newtemp*; *emit*(*E*.*place* ':=' E_1.*place* '+' E_2.*place*) }
$E \rightarrow E_1 * E_2$	{ *E*.*place* := *newtemp*; *emit*(*E*.*place* ':=' E_1.*place* '*' E_2.*place*) }
$E \rightarrow -E_1$	{ *E*.*place* := *newtemp*; *emit*(*E*.*place* ':=' 'uminus' E_1.*place*) }
$E \rightarrow (E_1)$	{ *E*.*place* := E_1.*place* }
$E \rightarrow \textbf{id}$	{ *p* := *lookup*(**id**.*name*); **if** *p* ≠ *nil* **then** *E*.*place* := *p* **else** *error* }

图8-15 为赋值语句产生三地址码的翻译模式

对于该文法产生的每个过程，图8-13的翻译模式为它们建立单独的符号表。每个这样的符号表都有一个表头，其中包含一个指向外围过程符号表的指针（见图8-12的例子）。当检查形成过程体的语句时，一个指向该过程的符号表的指针出现在栈 *tblptr* 的顶部。这个指针是由与 $D \rightarrow \textbf{proc id}\ ;\ N\ D_1\ ;\ S$ 右部的标记非终结符 N 关联的动作压入栈中的。

令非终结符 S 的产生式是图8-15中的那些产生式。由 S 产生的赋值语句中的名字必须在 S 所在的过程中已经被声明或者在某个外围过程中已被声明。当作用于 *name* 时，修改后的 *lookup* 操作首先检查 *name* 是否出现在当前符号表中，通过 *top*(*tblptr*) 即可访问。如果不是，*lookup* 使用符号表表头中的指针来查找外围过程的符号表，看名字是否在那里。如果在所有这样的作用域中都找不到该名字，那么 *lookup* 返回 *nil*。

例如，假定符号表如图8-12所示，并且过程 partition 中的一个赋值语句正被检查。操作 *lookup*(i) 会在 partition 的符号表中找到一个表项。由于 v 不在该符号表中，*lookup*(v)
479 将用该符号表表头中的指针继续查找外围过程 quicksort 的符号表。

8.3.2 临时名字的重用

我们一直沿用这样的假定：每当需要时，*newtemp* 就产生一个新的临时名字。每次调用 *newtemp* 产生一个单独的名字是有用的，在优化编译器中尤其如此，第10章将会确认这样做合理。但是，表达式计算中保存临时值的临时名字趋向于使符号表散乱，并且为保存它们的值不得不分配空间。

修改 *newtemp* 过程可以重用临时名字。另外一种方法是在代码生成阶段将独立的临时变量封装到同一个地址，这将在下一章进行讨论。

在表达式的语法制导翻译期间，像图8-15那样的规则会产生大量的临时名字来表示数据。由 $E \rightarrow E_1 + E_2$ 的规则产生的代码的一般形式为：

```
计算E1，将结果暂存到 t1
计算E2，将结果暂存到 t2
t := t1 + t2
```

从综合属性 *E.place* 的规则可以看出，不是在程序的任何地方都用到 t_1 和 t_2。这些临时变量的生存期可以像配对括号那样嵌套。事实上，计算 E_2 时用的所有临时变量的生存期都包含在 t_1 的生存期里。因此有可能修改 *newtemp*，使得它在过程的数据区域中使用一个小数组来保存临时名字，好像它是栈一样。

为简单起见，假定我们只处理整数。使用一个计数器 c，它的初值为0。每当临时名字作为运算对象使用时，c 减去1；每当要求产生新的临时名字时，使用 $\$c$ 并把 c 增加1。注意，该临时名字的“栈”在运行时并没有入栈和出栈操作，虽然编译器可能碰巧在“栈顶”存取临时变量的值。

例8.1 考虑赋值语句

```
x:= a*b + c*d - e*f
```

图8-16给出了如果 *newtemp* 修改后的三地址语句序列，它们原由图8-15的语义规则产生。该表还包含每个语句生成后 c 的“当前”值的指示。例如，计算 $0-$1 时，c 的值减少到0，所以 $0 又可用来保存结果。 □

语句	c 的值
	0
$0 := a * b	1
$1 := c * d	2
$0 := $0 + $1	1
$1 := e * f	2
$0 := $0 - $1	1
x := $0	0

图8-16 带有栈式临时变量的三地址码

对临时名字进行赋值和（或）使用可能不止一次，例如条件赋值语句中，临时名字不能以上述的后进先出方式指派名字。由于它们很少出现，所以可为所有这样的临时值单独指派自己的名字。当执行代码优化时，例如合并公共子表达式或将计算移出循环时（见第10章），临时

变量定义或使用多次的问题也会出现。一种合理的策略是每当创建一个额外的定义或使用一个临时名字或移动其计算时，均创建一个新的名字。

8.3.3 寻址数组元素

如果元素存放在连续的存储块中，数组元素就可以迅速地被访问。如果每个数组元素的宽度是 w，那么数组 A 的第 i 个元素的开始地址为：

$$base + (i - low) \times w \tag{8-4}$$

其中，low 是下标的下界，$base$ 是分配给数组的存储空间的相对地址，即 $base$ 是 A[low] 的相对地址。

如果把表达式 (8-4) 重写成

$$i \times w + (base - low \times w)$$

那么在编译时可以完成它的部分计算。子表达式 $c = base - low \times w$ 可以在看到数组声明时计算。假定 c 保存在 A 的符号表表项中，则 A[i] 的相对地址可以通过简单地将 $i \times w$ 加入 c 来获得。

编译时的预计算还可以应用在多维数组元素的地址计算上。二维数组形式上可以用两种形式之一来保存，或者*行优先*（一行接一行），或者*列优先*（一列接一列）。图8-17给出了一个 2×3数组的(a)行优先形式和(b)列优先形式。Fortran 使用列优先形式，Pascal 使用行优先形式，因为 A[i,j]等价于 A[i][j]，而且每个数组 A[i] 的元素都是连续存放的。

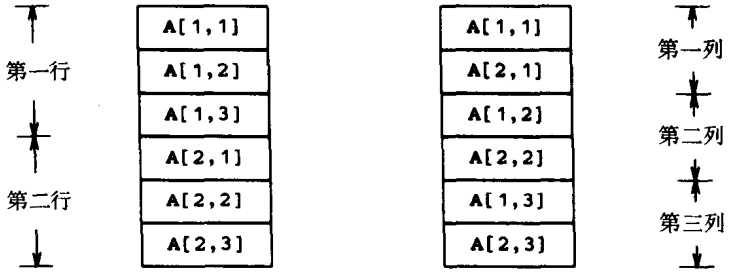


图8-17 二维数据的布局

a) 行优先 b) 列优先

如果二维数组采用行优先形式存储，可用如下公式计算A[i_1, i_2]的相对地址：

$$base + ((i_1 - low_1) \times n_2 + i_2 - low_2) \times w$$

其中，low_1 和 low_2 分别是 i_1，i_2 的下界，n_2 是 i_2 可取值的个数。也就是说，如果 $high_2$ 是 i_2 的上界，那么 $n_2 = high_2 - low_2 + 1$。假定 i_1，i_2 是编译时惟一尚未知道的值，我们可以把上述表达式重写成：

$$((i_1 \times n_2) + i_2) \times w + (base - ((low_1 \times n_2) + low_2) \times w) \tag{8-5}$$

该表达式的后一项可以在编译时确定。

我们可以将行优先或列优先的形式推广到多维数组。行优先形式的推广以如下方式存储元素：当向下扫描存储块时，最右边的下标变化最快，就像计程仪显示数字那样。表达式(8-5)可推广成如下表达式以计算 A[i_1, i_2, ⋯, i_k] 的相对地址：

$$((\cdots((i_1n_2+i_2)n_3+i_3)\cdots)n_k+i_k)\times w + base - ((\cdots((low_1n_2+low_2)n_3+low_3)\cdots)n_k+low_k)\times w \tag{8-6}$$

因为对任何 j，$n_j=high_j-low_j+1$ 是固定的，(8-6)中的第二行的项可以由编译器计算出来，并存放到符号表中 A 的表项里[⊖]。列优先形式则推广到相反的布局，最左边的下标变化最快。

进行过程调用时，某些语言允许数组的大小在运行时刻动态确定。这种在运行时栈上的数组分配已在7.3节讨论过了。其元素的访问公式和固定大小的数组相同，但上、下界在编译时是未知的。

480 ~ 482

为数组引用产生代码的主要问题是将式(8-6)的计算与数组引用的文法联系起来。如果非终结符 L 允许带有如下产生式，则赋值语句中允许有数组引用，其中 **id** 出现在图8-15中。

$$L \rightarrow \textbf{id}\ [\ Elist\ ] \mid \textbf{id}$$
$$Elist \rightarrow Elist\ ,\ E \mid E$$

为了使数组各维的长度 n_j 在我们将下标表达式合并到 *Elist* 时是可用的，需将产生式重写为：

$$L \rightarrow Elist\ ] \mid \textbf{id}$$
$$Elist \rightarrow Elist\ ,\ E \mid \textbf{id}\ [\ E$$

即把数组名与最左边的下标表达式连在一起，而不是在形成 L 时同 *Elist* 并在一起。这些产生式允许将符号表中指向数组名表项的指针作为 *Elist* 的综合属性 *array* 进行传递。[⊖]

我们还使用 *Elist.ndim* 来记录 *Elist* 中的维数（下标表达式）。函数 *limit*(*array*, j)返回 n_j，即由 *array* 指向其符号表表项的数组的第 j 维的元素个数。最后，*Elist.place* 表示临时名字，该名字保存根据 *Elist* 的下标表达式计算出来的值。

产生 k 维数组引用 A[i_1, i_2, ⋯, i_k]的前 m 个下标的 *Elist* 生成三地址码来计算下式，

$$(\cdots((i_1n_2+i_2)n_3+i_3)\cdots)n_m+i_m \tag{8-7}$$

它使用的是下列递归公式：

$$e_1 = i_1$$
$$e_m = e_{m-1}\times n_m + i_m \tag{8-8}$$

于是，当 $m=k$ 时，计算出式(8-6)第一行的子项只需要乘以宽度 w 就可以了。这里的 i_j 可能真是表达式的值，用来计算这些表达式的代码将散布在计算式(8-7)的代码之中。

左值 L 有两个属性，*L.place* 和 *L.offset*。当 L 是简单名字时，*L.place* 是指向符号表中该名字表项的指针，而 *L.offset* 是 **null**，表示该左值是一个简单名字，而不是数组引用。非终结符 E对 *E.place* 的翻译与此相同，与在图8-15中的含义相同。

483

8.3.4 数组元素寻址的翻译模式

我们将把语义动作加到下面的文法上：

(1) $S \rightarrow L := E$
(2) $E \rightarrow E + E$

⊖ 在C语言中，多维数组是通过定义其元素也是数组的数组来模拟的。例如，假设x是一个整数数组的数组。那么，该语言既允许x[i]又允许x[i][j]这样的表达式，而且它们的宽度不同。但是，所有数组的下界均为0，所以(8-6)的第二行的项在每一种情况下都简化为*base*。

⊖ 该变换同5.6节末尾提到的消除继承属性的变换类似。在此，我们也可以解决继承属性的问题。

(3) $E \rightarrow (E)$
(4) $E \rightarrow L$
(5) $L \rightarrow Elist\ ]$
(6) $L \rightarrow \mathbf{id}$
(7) $Elist \rightarrow Elist\ ,\ E$
(8) $Elist \rightarrow \mathbf{id}\ [\ E$

与在没有数组引用的表达式中一样，三地址码由在语义动作中调用的过程 *emit* 来产生。

如果L是简单名字，则生成一般的赋值；否则生成对 L 所指示位置的索引赋值：

```
(1)  S → L := E     { if L.offset = null then /* L是简单id */
                        emit(L.place ':=' E.place);
                    else
                        emit(L.place'['L.offset']' ':=' E.place) }
```

算术表达式的代码和图8-15中的完全一样：

```
(2)  E → E1 + E2    { E.place := newtemp;
                      emit(E.place ':=' E1.place '+' E2.place) }
(3)  E → ( E1 )     { E.place := E1.place }
```

如果数组引用 L 归约为 E，则我们需要 L 的右值。因此我们用索引获得位置 *L.place*[*L.offset*]的内容：

```
(4)  E → L          { if L.offset = null then /* L是简单id */
                        E.place := L.place
                    else begin
                        E.place := newtemp;
                        emit(E.place ':=' L.place'['L.offset']')
                    end }
```

下面，*L.offset* 是一个新的临时变量，表示式(8-6)中的第一项；函数 *width*(*Elist.array*) 返回式(8-6)中的w。*L.place* 表示式(8-6)中的第二项，由函数 *c*(*Elist.array*)返回。

```
(5)  L → Elist ]    { L.place := newtemp;
                      L.offset := newtemp;
                      emit(L.place ':=' c(Elist.array));
                      emit(L.offset ':=' Elist.place '*' width(Elist.array)) }
```

484

空的 *offset* 表示简单名字。

```
(6)  L → id         { L.place := id.place;
                      L.offset := null }
```

当看到下一个下标表达式时，应用递归公式(8-8)。在下面的动作中，$Elist_1.place$ 对应式(8-8)中的 e_{m-1}，*Elist.place* 对应 e_m。注意，如果 $Elist_1$ 有 $m-1$个分量，则产生式左部的 *Elist* 有 m 个分量：

```
(7)  Elist → Elist1 , E   { t := newtemp;
                            m := Elist1.ndim + 1;
                            emit(t ':=' Elist1.place '*' limit(Elist1.array, m ));
                            emit(t ':=' t '+' E.place);
                            Elist.array := Elist1.array;
                            Elist.place := t;
                            Elist.ndim := m }
```

E.place 不仅保存表达式 E 的值，还保存当 $m = 1$时式(8-7)的值。

(8) *Elist* → **id** [*E* { *Elist.array* := **id**.*place*;
Elist.place := *E.place*;
Elist.ndim := 1 }

例8.2 设A是一个10×20的数组，$low_1 = low_2 = 1$，因此$n_1 = 10$，$n_2 = 20$。取$w = 4$。赋值语句x := A[y,z]的注释分析树如图8-18所示。该赋值语句被翻译成下列三地址语句序列：

```
t1 := y * 20
t1 := t1 + z
t2 := c          /* 常量 c = baseA - 84 */
t3 := 4 * t1
t4 := t2[t3]
x := t4
```

对于每个变量，我们已用它的名字代替了**id**.*place*。 □

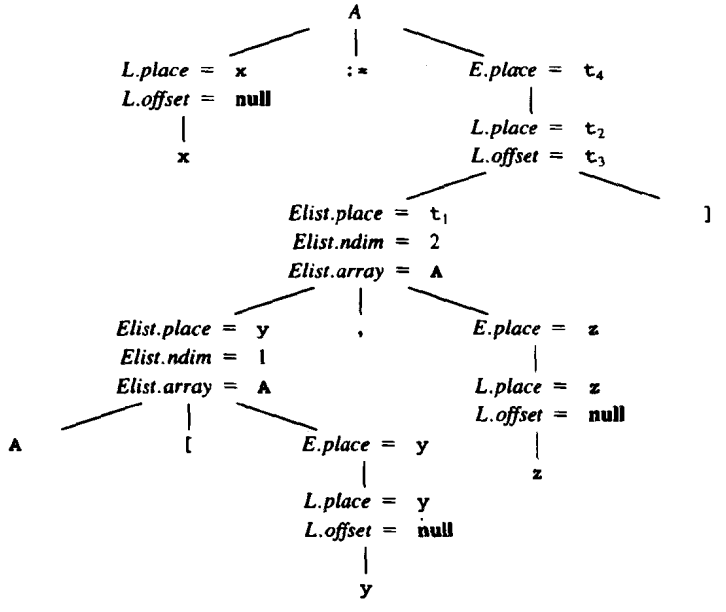


图8-18 x := A[y,z]的注释分析树

8.3.5 赋值语句中的类型转换

实际上，变量和常量有许多种不同的类型，因而编译器或者拒绝某种混合类型的操作，或者生成适当的强制（类型转换）指令。

考虑上述赋值语句的文法，假定有两种类型——整型和实型，必要时整型可以转换为实型。我们引入另一个属性*E.type*，它的值是*real*或*integer*。与产生式$E \rightarrow E + E$相关联的*E.type*的语义规则为：

E → *E* + *E* { *E.type* :=
if E_1.*type* = *integer* **and**
E_2.*type* = *integer* **then** *integer*
else *real* }

该规则符合6.4节提出的精神。这里以及本章的其余部分，我们将忽略类型错误的检查，这个问题已在第6章讨论过了。

必须修改 $E \rightarrow E + E$ 和大多数其他产生式的语义规则以便必要时生成x := inttorealy格式的三地址语句，它的作用是把整数 y 转换成等值的实数，称为x。我们还必须在操作符的代码中包含一个标记，指示这是定点算术运算还是浮点算术运算。产生式 $E \rightarrow E_1 + E_2$ 的完整语义动作列在图8-19中。

```
E.place := newtemp;
if E1.type = integer and E2.type = integer then begin
      emit(E.place ':=' E1.place 'int+' E2.place);
      E.type := integer
end
else if E1.type = real and E2.type = real then begin
      emit(E.place ':=' E1.place 'real+' E2.place);
      E.type := real
end
else if E1.type = integer and E2.type = real then begin
      u := newtemp;
      emit(u ':=' 'inttoreal' E1.place);
      emit(E.place ':=' u 'real+' E2.place);
      E.type := real
end
else if E1.type = real and E2.type = integer then 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8-19 $E \rightarrow E_1 + E_2$ 的语义动作

例如，对于输入 x := y + i*j，假定x和y的类型是 *real*，i 和 j 的类型是 *integer*，输出将是：

```
t1 := i int* j
t3 := inttoreal t1
t2 := y real+ t3
x := t2
```

图8-19的语义动作使用了非终结符 *E* 的两个属性 *E.place* 和 *E.type*。由于需要转换的类型数增加，随之而来的情况将成二次方增加（如果有多于两个变元的操作符，则情况更糟）。所以，有较多类型时，仔细地组织语义动作是非常重要的。

485 ~ 487

8.3.6 记录域的访问

编译器必须明了记录域的类型和相对地址，把这些信息记在域名字的符号表表项中的好处是，在符号表中查找名字的例程亦可用于查找域名。知道了这点，可以通过上一节中图8-14的语义动作为每个记录类型单独构造一张符号表。如果 *t* 是指向记录类型符号表的指针，那么将构造器 *record* 作用于该指针形成的类型 *record*(*t*) 作为 *T.type* 被返回。

我们用表达式

```
p↑.info + 1
```

来说明怎样从属性 *E.type* 中抽取出指向符号表的指针。从该表达式中的操作可以知道，p 是指向带有域名 info 的记录的指针，且 info 的类型必须是算术类型。如果类型如图8-13和图8-14那样构造，则 p 的类型必须由类型表达式

pointer(*record*(*t*))

给出。那么 p↑ 的类型是 *record* (*t*)，从中可以抽取 *t*。可以在 *t* 指向的符号表中查找域名 info。

8.4 布尔表达式

在程序设计语言中，布尔表达式有两个基本的作用。一个是用于计算逻辑值，但更多地是用作如 if-then、if-then-else 或 while-do 等语句中改变控制流的条件表达式。

布尔表达式是由布尔运算符（**and**，**or** 和 **not**）作用于布尔变量或关系表达式而构成的。关系表达式的形式是 E_1 **relop** E_2，其中 E_1 和 E_2 是算术表达式。一些语言，譬如说 PL/I，允许更一般的表达式，即布尔、算术及关系运算符可以作用于任何类型的表达式，布尔值和算术值

之间没有区别，必要时可以进行强制转换。本节考虑由如下文法生成的布尔表达式：

$E \rightarrow E$ **or** $E \mid E$ **and** $E \mid$ **not** $E \mid (\ E\ ) \mid$ **id relop id** $\mid$ **true** $\mid$ **false**

我们用属性 *op* 来确定 **relop** 究竟代表的是比较算符 <,≤,=,≠,>,≥ 中的哪一个。按习惯，我们
488 假定 **or** 和 **and** 是左结合的，**or** 的优先级最低，其次是 **and**，最后是 **not**。

8.4.1 翻译布尔表达式的方法

有两种主要方法表示布尔表达式的值。第一种方法是将真和假按数值编码，从而对布尔表达式的计算类似于对算术表达式的计算。常常用1表示真，而用0表示假，虽然许多其他的编码也是可能的。例如，可以用非0表示真，0表示假；或者非负数表示真，负数表示假。

实现布尔表达式的第二种方法是通过控制流，即用程序到达的位置来表示布尔表达式的值。用这种方法实现控制流语句中的布尔表达式尤其方便。例如，对于表达式 E_1 **or** E_2，如果确定了 E_1 是真，那么可以肯定整个表达式为真，而不必再去计算 E_2。

程序设计语言的语义决定了是否需要计算布尔表达式的所有部分。如果语言定义允许（或要求）部分布尔表达式不用计算，那么，编译器可以优化布尔表达式的计算，使之只计算足够的部分便可确定它的值。因此，在 E_1 **or** E_2 这样的表达式中，不必完全计算出 E_1 及 E_2。如果 E_1或 E_2 是有副作用的表达式（即包含修改全局变量的函数），那么可能会得到跟预期不一致的结果。

上述两种方法中，不能绝对地说某一种优于另一种。例如，BLISS/11优化编译器为每个表达式单独选择适当的方法。本节将讨论把布尔表达式翻译成三地址码的这两种方法。

8.4.2 数值表示

首先考虑用1表示真用0表示假的布尔表达式的实现。表达式将从左到右按与算术表达式类似的方式完全计算。例如，

```
a or b and not c
```

将被翻译成如下三地址序列：

```
t1 := not c
t2 := b and t1
t3 := a or t2
```

a<b 这样的关系表达式等价于条件语句 `if a<b then 1 else 0`，它可以被翻译成三
489 地址码序列为（假定语句标号从100开始）：

```
100:  if a < b goto 103
101:  t := 0
102:  goto 104
103:  t := 1
104:
```

为布尔表达式产生三地址码的一个翻译模式见图8-20。在该模式中，假定 *emit* 以正确的格式把三地址语句发送到输出文件中，*nextstat* 给出下一个三地址语句在输出序列中的索引，产生每一个三地址语句之后，*emit* 便将 *nextstat* 加1。

例8.3 图8-20中的翻译模式为表达式a<b or c<d and e<f生成的三地址码如图8-21所示。 □

$E \rightarrow E_1$ **or** E_2	{ *E.place* := *newtemp*; *emit*(*E.place* ':=' E_1.*place* 'or' E_2.*place*) }
$E \rightarrow E_1$ **and** E_2	{ *E.place* := *newtemp*; *emit*(*E.place* ':=' E_1.*place* 'and' E_2.*place*) }
$E \rightarrow$ **not** E_1	{ *E.place* := *newtemp*; *emit*(*E.place* ':=' 'not' E_1.*place*) }
$E \rightarrow$ (E_1)	{ *E.place* := E_1.*place* }
$E \rightarrow$ **id**$_1$ **relop** **id**$_2$	{ *E.place* := *newtemp*; *emit*('if' **id**$_1$.*place* **relop**.*op* **id**$_2$.*place* 'goto' *nextstat* +3); *emit*(*E.place* ':=' '0'); *emit*('goto' *nextstat* +2); *emit*(*E.place* ':=' '1') }
$E \rightarrow$ **true**	{ *E.place* := *newtemp*; *emit*(*E.place* ':=' '1') }
$E \rightarrow$ **false**	{ *E.place* := *newtemp*; *emit*(*E.place* ':=' '0') }

图8-20 用数值表示布尔值的翻译模式

```
100:  if a < b goto 103        107:  t2 := 1
101:  t1 := 0                  108:  if e < f goto 111
102:  goto 104                 109:  t3 := 0
103:  t1 := 1                  110:  goto 112
104:  if c < d goto 107        111:  t3 := 1
105:  t2 := 0                  112:  t4 := t2 and t3
106:  goto 108                 113:  t5 := t1 or t4
```

图8-21 表达式 a<b or c<d and e<f 的翻译

8.4.3 短路代码

即使不对布尔运算符生成代码并且不必生成整个表达式的代码，我们也可以把布尔表达式翻译成的三地址码。这种风格的计算有时叫做“短路”或“跳转”代码。如果用代码序列中的位置来表示表达式的值，那么可能不用为布尔运算符 **and**，**or** 和 **not** 生成代码就可以计算表达式的值。例如，图8-21中，可以根据到达语句101还是语句103来确定 t_1 的值，因此 t_1 的值是冗余的。对于许多布尔表达式，确定表达式的值而不用完全计算它是可能的。

8.4.4 控制流语句

现在考虑在由下面文法生成的 if-then、if-then-else 和 while-do 语句的上下文中的将布尔表达式翻译成三地址语句：

$S \rightarrow$ **if** E **then** S_1
| **if** E **then** S_1 **else** S_2
| **while** E **do** S_1

在这些产生式中，E 是要翻译的布尔表达式。在翻译中，假定可以用符号标号来标识一条三地址语句，每次调用函数 *newlabel* 将返回一个新的符号标号。

对于一个布尔表达式 E 关联两个标号：*E.true* 和 *E.false*，它们分别表示 E 为真和假时控制流转向的标号。翻译控制流语句 S 的语义规则允许控制从 *S.code* 中跳转到紧接在 *S.code* 之后

的那条三地址指令。但在某些情况下，紧跟 *S.code* 的指令是跳转到某个标号 L 的跳转指令，用继承属性 *S.next* 可以避免从 *S.code* 中跳转到 L 的跳转。*S.next* 的值是标号，它是 *S* 的代码后应执行的第一个三地址指令的标号[⊖]。我们没有给出 *S.next* 的初始化。

在翻译 if-then 语句 *S*→**if** *E* **then** S_1 时，建立一个新标号 *E.true*，并把它作为语句 S_1 生成的第一条三地址指令的标号，如图8-22a所示。图8-23给出了一个语法制导定义。*E* 的代码是：*E* 为真则跳转到 *E.true*；*E* 为假则跳转到 *S.next*。因此，我们置 *E.false* 为 *S.next*。

491
~
492

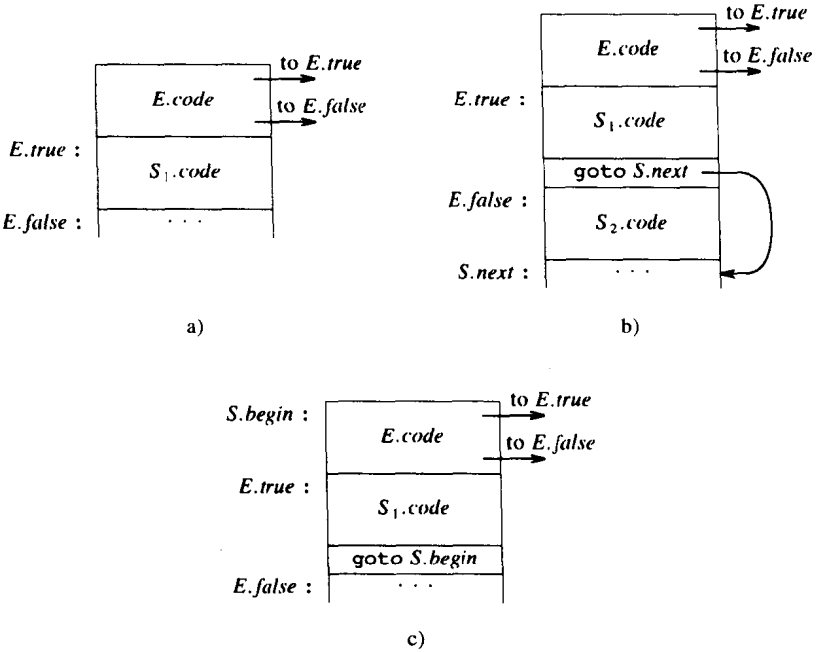


图8-22 if-then，if-then-else 和 while-do 语句的代码

a) if-then b) if-then-else c) while-do

在翻译 if-then-else 语句 *S*→**if** *E* **then** S_1 **else** S_2 时，如果 *E* 为真，布尔表达式的代码跳转到 S_1 代码的第一条指令；如果*E*为假，跳转到 S_2 代码的第一条指令，如图8-22b所示。和 if-then 语句一样，继承属性 *S.next* 给出了执行 *S* 的代码后要执行的第一个三地址语句的标记。在 S_1 的代码之后有一条明显的 goto *S.next* 指令，但 S_2 之后没有。使用这些语义规则，如果 *S.next* 不是紧跟在 S_2.*code* 之后的指令的标号，那么外围语句将在 S_2 的代码之后提供跳转到 *S.next* 的指令，其证明留给读者。

语句 *S*→**while** *E* **do** S_1 的代码如图8-22c所示。首先建立一个新的标号 *S.begin*，并将其作为*E* 的第一条指令的标号。另一个新标号 *E.true* 作为 S_1 的第一条指令的标号。*E* 的代码在 *E* 为真时跳转到 *E.true*，而在 *E* 为假时跳转到 *S.next*，因此 *E.false* 还是置为 *S.next*。在 S_1 的代码之后，我们放上指令 goto *S.begin*，它引起控制跳转到布尔表达式代码的开始位置。注意，S_1.*next*置为标号 *S.begin*，这样在从 S_1.*code* 中的跳转指令就可以直接跳转到 *S.begin*。

我们将在8.6节更详细地讨论控制流语句的翻译，8.6节使用一种称为“回填”的方法，经过一遍扫描即可生成这些语句的代码。

⊖ 如果按字面来实现，继承标号 *S.next* 的方法可能导致标号的增生。8.6节的回填方法只在需要时才生成标号。

产生式	语义规则
$S \rightarrow$ **if** E **then** S_1	$E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.next;$ $S.code := E.code \parallel gen(E.true\ ':') \parallel S_1.code$
$S \rightarrow$ **if** E **then** S_1 **else** S_2	$E.true := newlabel;$ $E.false := newlabel;$ $S_1.next := S.next;$ $S_2.next := S.next;$ $S.code := E.code \parallel gen(E.true\ ':') \parallel S_1.code \parallel gen('\texttt{goto}'\ S.next) \parallel gen(E.false\ ':') \parallel S_2.code$
$S \rightarrow$ **while** E **do** S_1	$S.begin := newlabel;$ $E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.begin;$ $S.code := gen(S.begin\ ':') \parallel E.code \parallel gen(E.true\ ':') \parallel S_1.code \parallel gen('\texttt{goto}'\ S.begin)$

图8-23 控制流语句的语法制导定义

8.4.5 布尔表达式的控制流翻译

我们现在讨论 *E.code*，图8-23中为布尔表达式 E 生成的代码。正如前面所说的，E 被翻译成三地址语句序列，将 E 作为跳到下面两个位置之一的条件跳转和无条件跳转序列来计算：*E.true* 表示 E 为真时跳转到的位置；*E.false* 表示 E 为假时跳转到的位置。

这种翻译的基本思想如下所述：假定 E 形如 a<b，则生成的代码形式为：

```
if a < b goto E.true
goto E.false
```

设 E 形如 E_1 **or** E_2。如果 E_1 为真，则立即可知 E 为真，即 $E_1.true$ 和 $E.true$ 相同。如果 E_1 为假，则必须计算 E_2 的值，因此我们令 $E_1.false$ 为 E_2 代码的第一条语句的标号。E_2 的真、假出口分别与 E 的真、假出口相同。 493

翻译 E_1 **and** E_2 也采用类似地考虑。**not** E_1 形式的表达式 E 无需生成代码，只要交换 E_1 的真值出口和假值出口就可以得到 E 的真、假值出口。按这种方法为布尔表达式生成三地址码的语法制导定义见图8-24，注意，*true* 和 *false* 都是继承属性。

例8.4 再次考虑如下表达式：

```
a < b or c < d and e < f
```

假定整个表达式的真、假出口已分别置为 Ltrue 和 Lfalse，那么用图8-24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
```

注意，这里生成的代码不是优化的，因为删掉第二条语句不会改变代码的值。这种形式的冗余指令用窥孔（peephole）优化器（见第9章）可以很容易地删除。避免产生这些冗余跳转的另一个办法是把形如 $\mathbf{id}_1 < \mathbf{id}_2$ 的关系表达式翻译成 `if` $\mathbf{id}_1 \geqslant \mathbf{id}_2$ `goto` *E.false* 语句，但前提是条件为真时执行紧跟在它之后的代码。 □

产生式	语义规则
$E \rightarrow E_1$ **or** E_2	$E_1.true := E.true$; $E_1.false := newlabel$; $E_2.true := E.true$; $E_2.false := E.false$; $E.code := E_1.code \parallel gen(E_1.false\text{':'}) \parallel E_2.code$
$E \rightarrow E_1$ **and** E_2	$E_1.true := newlabel$; $E_1.false := E.false$; $E_2.true := E.true$; $E_2.false := E.false$; $E.code := E_1.code \parallel gen(E_1.true\text{':'}) \parallel E_2.code$
$E \rightarrow$ **not** E_1	$E_1.true := E.false$; $E_1.false := E.true$; $E.code := E_1.code$
$E \rightarrow$ (E_1)	$E_1.true := E.true$; $E_1.false := E.false$; $E.code := E_1.code$
$E \rightarrow \mathbf{id}_1$ **relop** $\mathbf{id}_2$	$E.code := gen(\text{'if'}\ \mathbf{id}_1.place\ \mathbf{relop}.op\ \mathbf{id}_2.place\ \text{'goto'}\ E.true) \parallel$ $gen(\text{'goto'}\ E.false)$
$E \rightarrow$ **true**	$E.code := gen(\text{'goto'}\ E.true)$
$E \rightarrow$ **false**	$E.code := gen(\text{'goto'}\ E.false)$

图8-24 为布尔表达式生成三地址码的语法制导定义

例8.5 考虑如下语句：

```
while a < b do
    if c < d then
        x := y + z
    else
        x := y - z
```

上面的语法制导定义，以及赋值语句和布尔表达式的翻译模式，将生成下列代码：

```
L1: if a < b goto L2
    goto Lnext
L2: if c < d goto L3
    goto L4
L3: t1 := y + z
    x := t1
    goto L1
L4: t2 := y - z
    x := t2
    goto L1
Lnext:
```

注意，通过改变测试的方向，可以删除前两个goto语句。这种类型的局部翻译可以利用第9章讨论的窥孔优化器来完成。 □

8.4.6 混合模式的布尔表达式

需要注意的是我们已经简化了布尔表达式的语法。实际上，布尔表达式通常含有算术子表达式，譬如(a+b)<c。在假值为算术值0、真值为数值1的语言中，(a<b)+(b<a)甚至可以被看作是算术表达式。如果a与b的值相等，则等于0，否则等于1。 494 ~ 495

即使用计算其值的代码表示算术表达式，利用跳转代码表示布尔表达式的方法仍然可以使用。例如，考虑下面的代表语法：

$$E \rightarrow E + E \mid E \textbf{ and } E \mid E \textbf{ relop } E \mid \textbf{id}$$

我们假定 $E+E$ 产生一个整型算术结果（本例中，包含实型或其他的算术类型将使问题变得更复杂，却没有什么价值），而 E **and** E 及 E **relop** E 产生由控制流语句代表的布尔值。表达式 E **and** E 要求两个参数都必须是布尔型的，但是操作+及 **relop** 使用两种参数类型都可以，包括混合类型。$E \rightarrow$ **id** 也被认为是算术操作数，尽管可以使用布尔标识符扩展该例子。

```
E.type := arith;
if E1.type = arith and E2.type = arith then begin
    /* 正常算术加法 */
    E.place := newtemp;
    E.code := E1.code || E2.code ||
        gen(E.place ':=' E1.place '+' E2.place)
end
else if E1.type = arith and E2.type = bool then begin
    E.place := newtemp;
    E2.true := newlabel;
    E2.false := newlabel;
    E.code := E1.code || E2.code ||
        gen(E2.true ':' E.place ':=' E1.place + 1) ||
        gen('goto' nextstat + 1) ||
        gen(E2.false ':' E.place ':=' E1.place)
else if ...
```

图8-25 产生式 $E \rightarrow E_1 + E_2$ 的语义规则

在这种情况下生成代码，我们可以使用综合属性 *E.type*，它根据 E 的类型决定是 *arith* 还是*bool* 类型。对布尔表达式，E 具有继承属性 *E.true* 及 *E.false*，而对算术表达式则具有综合属性*E.place*。$E \rightarrow E_1 + E_2$ 的部分语义规则如图8-25所示。

在混合模式情况下，我们先为 E_1 生成代码，然后是 E_2，紧跟着是如下三个语句：

```
E2.true:  E.place := E1.place + 1
          goto nextstat + 1
E2.false: E.place := E1.place
```

496

当 E_2 为真时，第一条语句为 E 计算 $E_1 + 1$的值。当 E_2 为假时，第三条语句为 E 计算E_1的值。第二条语句跳过第三条语句。剩余情况与其他产生式的语义规则与此十分相似，我们将其留作练习。

8.5 case语句

许多语言中都提供 switch 语句或 case 语句；甚至 Fortran 中的计算 goto 和赋值 goto 都可以看作是 switch 语句的变种。我们的 switch 语句的语法如图8-26所示。

这里有一个需要计算的选择表达式，后面有 n 个常量值，它们是表达式可能的取值。也许还包含一个默认值，它总是匹配那些没有其他值匹配的选择表达式。

```
switch 表达式
    begin
        case 值: 语句
        case 值: 语句
            ...
        case 值: 语句
        default: 语句
    end
```

图8-26 switch 语句的语法

switch 语句的翻译代码是：

1. 计算表达式的值。

2. 在 case 列表中查找与表达式值相同的值。如果没有这样显式的值，则默认值与表达式值匹配。

3. 执行与找到的值相关联的语句。

步骤2是 n 路的分支，它可以用几种方法实现。如果情况数不是很多，譬如说最多10个，则用条件 goto 序列是合理的，每个 goto 测试一个单独的值，为真时就转移到相应语句的代码。

实现该条件 goto 序列的一种更为紧凑的办法是建立一个序对表，每个序对由值和相应语句代码的标号组成。还要生成把表达式的值放在该表末尾的代码，和它配对的是默认语句的标号。编译器可以生成一个简单循环，将表达式的值和表中每个值进行比较，如果确信没有其他的值可匹配，最后的默认条目肯定匹配。

497 如果值的数目大约超过10，为这些值构造散列表（见7.6节）会更有效，将这些语句的标号作为散列表的表项。如果找不到与 switch 语句包含的表达式值相应的表项，则生成到默认语句的跳转。

有一种通常的特殊情况，可以用更有效的办法来实现 n 路分支。如果所有的值落在一个较小的区间内，比方说 i_{min} 和 i_{max} 之间，不同值的个数占 $i_{max}-i_{min}$ 的相当一部分，那么可以构造一个标号数组，值 j 的语句标号放在偏移为 $j-i_{min}$ 的表项中，空白的表项全添上默认语句的标号。执行 switch 时，先计算表达式以获得它的值 j，然后检查它的值是否在 i_{min} 和 i_{max} 之间，再间接转移到偏移为 $j-i_{min}$ 的表项。例如，如果表达式为字符类型，则可以建立一个具有128个表项（根据字符集的大小）的表，这时范围检查可以省去。

case语句的语法制导翻译

考虑下面的 switch 语句：

```
switch E
    begin
        case V1:    S1
        case V2:    S2
            ...
        case Vn-1:  Sn-1
        default:    Sn
    end
```

用语法制导翻译模式，可以很容易地把该 case 语句翻译成形如图8-27所示的中间代码。

```
            计算 E 并将其存入t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 = Vn-1 goto Ln-1
            goto Ln
next:
```

图8-27 case 语句的一种翻译

所有的测试都放在末端，以便简单的代码生成器可以识别这种多路分支，并用本节开头建议的最恰当的实现方法为其生成高效的代码。如果要生成图8-28中更直截的代码序列，则代码生成器还要做更深入的分析，才能找出效率最高的实现方法。注意，把分支语句放在开始处是不方便的，因为编译器不能在遇到每个 S_i 时就为其生成代码。

为了翻译成图8-27的形式，当看到关键字 **switch** 时，首先生成两个新的标号 test 和 next 以及一个新的临时名字 t。然后在分析表达式 E 时，生成将 E 的计算值存到 t 的代码。处理完 E 之后，生成跳转 goto test 语句。

然后，当看到关键字 **case** 时，建立一个新的标号 L_i，并把它填进符号表中，把指向该符

号表表项的指针和 case 常量的值 V_i 压入专用于存储 case 的栈。(如果该 switch 语句嵌在另一个switch语句的内部，还需在栈上放一个标记，以区别内外两个switch语句的case。)

我们通过输出新创建的标号L_i处理每个语句 **case** V_i : S_i，L_i 后面跟以S_i的代码，再后面是跳转 goto next。当遇到终止switch语句体的关键字 **end**时，我们已准备好为n路分支生成代码了。自底向上扫描case栈，读出指针-值序对，即可生成如下形式的三地址语句序列：

```
case V1 L1
case V2 L2
    ...
case Vn-1 Ln-1
case t Ln
label next
```

```
        计算E并将其存入t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:
```

图8-28 case语句的另一种翻译

其中，t 是保存选择表达式 E 的值的名字，L_n 是默认语句的标号。三地址语句 case V_i L_i 和图8-27 中if t = V_i goto L_i一样，但是对最终的代码生成器来说，case 比较容易被检测为需特殊处理候选式。在代码生成阶段，case 语句序列能否被翻译成效率更高的 n 路分支，取决于有多少个值，以及这些值是否落在一个小区间内。

8.6 回填

8.4节中语法制导定义的最简单的实现方法是使用两遍扫描。首先，为输入构建一棵语法树，然后按深度优先顺序遍历语法树，计算定义中给出的翻译。通过单遍扫描为布尔表达式和控制流语句生成代码的主要问题是在单遍扫描中，生成跳转语句时我们可能不知道控制要转向的语句标号。为此，我们可以先生成暂时没有指定目标标号的一系列跳转指令来克服这个问题。每个这样的语句都将放在一个 goto 语句列表中，goto 语句列表的标号在目标标号确定下来后再添入。我们称这种之后添入标号的方式为回填。

本节将说明如何利用回填技术通过单遍扫描生成布尔表达式和控制流语句的代码。除了生成标号的方式以外，我们生成的翻译将与8.4节的形式相同。为具体起见，我们将四元式放在一个四元式数组中，标号将是这个数组的下标。为操纵标号列表我们要用到以下三个函数：

1. *makelist* (i)，创建一个只包含i的新列表，i 是四元式数组的下标，*makelist* 返回指向它所创建列表的指针。

2. *merge* (p_1, p_2)，连接由 p_1 和 p_2 指向的两个列表，并返回指向连接后的列表的指针。

3. *backpatch* (p, i)，将 i 插入到由 p 指向的列表的每一条语句中作为该语句的目标标号。

8.6.1 布尔表达式

我们现在构造一种翻译模式，它适合于在自底向上语法分析过程中为布尔表达式生成四元式。我们将把标记非终结符 M 插入到文法中，以便引入一个语义动作在适当的时候获得下一四元式的索引。我们将使用如下文法：

(1) $E \rightarrow E_1$ **or** $M\ E_2$
(2) | E_1 **and** $M\ E_2$
(3) | **not** E_1
(4) | (E_1)
(5) | **id**$_1$ **relop** **id**$_2$
(6) | **true**
(7) | **false**
(8) $M \rightarrow \epsilon$

非终结符 E 的综合属性 *truelist* 和 *falselist* 用于生成布尔表达式的跳转代码。在生成 E 的代码时，还不能完全完成跳转到真、假出口的代码，因为指令中的目标标号域尚不能填写。这些未完成的跳转指令放在由 *E.truelist* 和*E.falselist* 所指向的列表中。

语义动作应反映上述考虑。考虑产生式 $E \rightarrow E_1$ **and** $M\ E_2$。如果 E_1 为假，则 E 也为假，所以 E_1.*falselist* 列表中的语句应是 *E.falselist* 列表的一部分。如果 E_1 为真，则必须进一步检查 E_2，因此 E_1.*truelist* 列表中转移语句的目标标号一定是 E_2 的第一条语句的标号。该目标标号是通过标记非终结符 M 获得的。属性 *M.quad* 记录 E_2.*code* 的第一条语句的编号。对于产生式 $M \rightarrow \epsilon$，我们关联其语义动作为：

{ *M.quad* := *nextquad* }

变量 *nextquad* 用于保存紧跟着的下一四元式的索引。在看到产生式 $E \rightarrow E_1$ **and** $M\ E_2$ 的剩余部分后，该值回填 E_1.*truelist* 列表中。翻译模式如下：

(1) $E \rightarrow E_1$ **or** $M\ E_2$ { *backpatch* (E_1.*falselist*, *M.quad*);
E.truelist := *merge*(E_1.*truelist*, E_2.*truelist*);
E.falselist := E_2.*falselist* }

(2) $E \rightarrow E_1$ **and** $M\ E_2$ { *backpatch* (E_1.*truelist*, *M.quad*);
E.truelist := E_2.*truelist*;
501 *E.falselist* := *merge*(E_1.*falselist*, E_2.*falselist*) }

(3) $E \rightarrow$ **not** E_1 { *E.truelist* := E_1.*falselist*;
E.falselist := E_1.*truelist* }

(4) $E \rightarrow$ (E_1) { *E.truelist* := E_1.*truelist*;
E.falselist := E_1.*falselist* }

(5) $E \rightarrow$ **id**$_1$ **relop** **id**$_2$ { *E.truelist* := *makelist*(*nextquad*);
E.falselist := *makelist*(*nextquad* + 1);
emit('`if` **id**$_1$.*place* **relop**.*op* **id**$_2$.*place* '`goto _`')
emit('`goto _`') }

(6) $E \rightarrow$ **true** { *E.truelist* := *makelist*(*nextquad*);
emit('`goto _`') }

(7) $E \rightarrow$ **false** { *E.falselist* := *makelist*(*nextquad*);
emit('`goto _`') }

(8) $M \rightarrow \epsilon$ { *M.quad* := *nextquad* }

简单地说，产生式(5)的语义动作将生成两条语句，一条是条件 goto，另一条是无条件 goto，它们的目标标号均未填写。第一条生成语句的下标生成列表，由 *E.truelist* 中指针指向这个它，第二条生成语句 goto_也生成列表，加入到 *E.falselist* 中。

例8.6 再次考虑表达式 `a<b or c<d and e<f`。它的一棵注释分析树如图8-29所示。在对树的深度优先遍历过程中执行动作。因为所有的动作均出现在产生式右部的末端，所以可

以在自底向上语法分析中伴随着对产生式的归约来执行它们。在用产生式(5)将 a<b 归约为 E 时，生成如下两个四元式：

```
100: if a < b goto _
101: goto _
```

（在此我们再次假定语句编号从100开始。）产生式 $E \rightarrow E_1$ **or** M E_2 中的标记非终结符 M 记录 *nextquad* 的值，此时为102。在用产生式(5)将 c<d 归约为 E 时，生成如下两个四元式：

```
102:  if c < d goto _
103:  goto _
```

现在我们已经在产生式 $E \rightarrow E_1$ **and** M E_2 中看到 E_1，在此产生式中的标记非终结符 M 记录下 *nextquad* 的值，现在是104。用产生式(5)将 e<f 归约为 E 时，生成如下两个四元式：

```
104:  if e < f goto _
105:  goto _
```

502

现在我们用产生式 $E \rightarrow E_1$ **and** M E_2进行归约。相应的语义动作调用 *backpatch*({102},104)，其中参数{102}表示一个指针，指向仅包含102的列表，而此列表就是那个由 *E.truelist* 指向的列表。该 *backpatch* 调用将104填到语句102中。至此生成的六条语句为：

```
100:  if a < b goto _
101:  goto _
102:  if c < d goto 104
103:  goto _
104:  if e < f goto _
105:  goto _
```

最后用产生式 $E \rightarrow E_1$ **or** M E_2 进行归约，相应的语义动作调用 *backpatch* ({101},102)将上述语句变为：

```
100:  if a < b goto _
101:  goto 102
102:  if c < d goto 104
103:  goto _
104:  if e < f goto _
105:  goto _
```

当且仅当到达编号为100或104的 goto 时，整个表达式为真，而当且仅当到达编号为103或

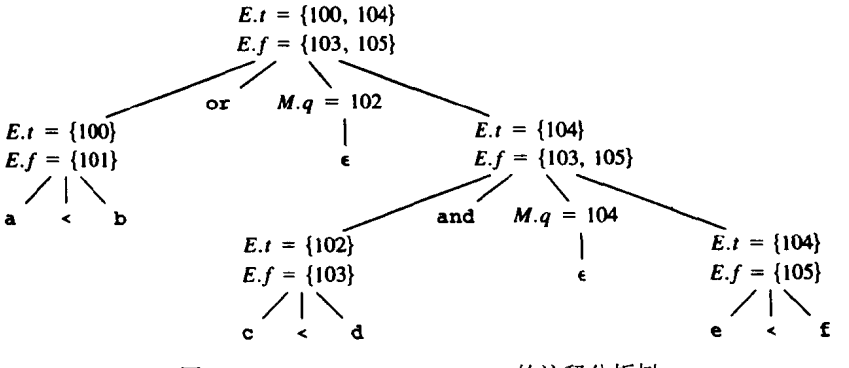


图8-29 a<b or c<d and e<f 的注释分析树

105的 goto 时，整个表达式为假。这些指令的目标标号将在后面的编译中填写，那时根据表达
503 式的真假状态就知道了要做什么。 □

8.6.2 控制流语句

现在我们来介绍如何将回填技术用在控制流语句的一遍翻译中。如上所述，我们将注意力集中在四元式的生成上，而且继续沿用上节中有关域名翻译的符号和列表处理过程。作为一个更大点的例子，我们将给出下述文法所生成语句的一个翻译模式：

(1) $S \rightarrow$ **if** E **then** S
(2) | **if** E **then** S **else** S
(3) | **while** E **do** S
(4) | **begin** L **end**
(5) | A
(6) $L \rightarrow L$; S
(7) | S

在此，S 表示语句，L 表示语句列表，A 为赋值语句，而 E 为布尔表达式。注意还必须有一些其他产生式，如生成赋值语句的产生式。不过这里给出的产生式已足以说明控制流语句的翻译技术。

我们使用与8.4节相同的 if-then、if-then-else 和 while-do 语句的代码结构。我们还假定执行时紧跟在某给定语句后面的代码在四元式数组中也紧跟其后。否则，必须给出一条显式的跳转语句。

我们的一般方法是：当发现其目标时再填写跳转语句的转向。不仅布尔表达式需要两个跳转列表来存放表达式为真、假时的转向，而且语句亦需要跳转列表（由属性*nextlist*给出）指向执行序列中紧随其后的代码。

8.6.3 翻译的实现方案

我们现在来描述生成上述控制流结构翻译的语法制导翻译模式。如前所述，非终结符 E 有两个属性 *E.truelist* 和 *E.falselist*。L 和 S 也分别需要一个未填写目标标号的四元式列表，其目标标号最终用回填技术回填。这些列表分别由属性 *L.nextlist* 和 *S.nextlist* 指向。*S.nextlist* 是一个指向跳转列表的指针，列表中存放转向紧跟语句S之后要执行的四元式的条件跳转和无条件跳转语句。*L.nextlist* 的定义与此类似。

图8-22c中的语句 $S \rightarrow$ **while** E **do** S_1 的代码布局中，有两个标记 *S.begin* 和 *E.true* 分别标记完整语句 S 和语句体 S_1 的代码的开始位置。下列产生式中标记非终结符M的两次出现记录这些位置的四元式编号：

504 $S \rightarrow$ **while** M_1 E **do** M_2 S_1

M 的惟一一条产生式是 $M \rightarrow \epsilon$，它的动作是将属性 *M.quad* 设置为下一四元式的编号。当while语句的循环体 S_1 执行完以后，控制流转向S语句的开始处。因此，当我们把 **while** M_1 E **do** M_2 S_1 归约为 S 时，我们回填 S_1.*nextlist*，使得其中所有目标标号为 M_1.*quad*。当然，我们需要在S_1的代码后面追加一条显式跳转到E的代码的开始位置的指令，因为控制流可能会从底部离开循环。通过将 *E.truelist* 中的跳转指向 M_2.*quad*，*E.truelist* 被回填以跳转到S_1的起始位置。

为条件语句 **if** E **then** S_1 **else** S_2 生成代码时，更能看出使用 *S.nextlist* 和 *L.nextlist* 的理由。与S_1是赋值的情形一样，如果控制离开 S_1 的底部，我们必须在S_1的代码后面增加一条跳转指令

以跳过 S_2 的代码。我们用另一个标记非终结符在 S_1 之后引入这条跳转指令，设非终结符 N 就是这个标记，它的产生式为 $N \rightarrow \epsilon$。N 具有属性 $N.nextlist$，它是由 N 的语义规则生成的 goto_语句的四元式编号构成的列表。下面给出修改后文法的语义规则：

(1) $S \rightarrow$ **if** E **then** $M_1\ S_1\ N$ **else** $M_2\ S_2$

{ *backpatch*($E.truelist$, $M_1.quad$);
backpatch($E.falselist$, $M_2.quad$);
$S.nextlist$:= *merge*($S_1.nextlist$, *merge*($N.nextlist$, $S_2.nextlist$)) }

当 E 为真时我们用 $M_1.quad$ 回填 $E.truelist$，它是 S_1 代码的起始位置。类似地，E 为假时我们回填跳转语句使它转到 S_2 的代码的起始位置。列表 $S.nextlist$ 中包含所有跳出 S_1、S_2 的跳转指令，也包含由 N 生成的跳转指令。

(2) $N \rightarrow \epsilon$ { $N.nextlist$:= *makelist*(*nextquad*);
emit('goto _') }

(3) $M \rightarrow \epsilon$ { $M.quad$:= *nextquad* }

(4) $S \rightarrow$ **if** E **then** $M\ S_1$ { *backpatch*($E.truelist$, $M.quad$);
$S.nextlist$:= *merge*($E.falselist$, $S_1.nextlist$) }

(5) $S \rightarrow$ **while** $M_1\ E$ **do** $M_2\ S_1$ { *backpatch*($S_1.nextlist$, $M_1.quad$);
backpatch($E.truelist$, $M_2.quad$);
$S.nextlist$:= $E.falselist$
emit('goto' $M_1.quad$) }

(6) $S \rightarrow$ **begin** L **end** { $S.nextlist$:= $L.nextlist$ }

(7) $S \rightarrow A$ { $S.nextlist$:= **nil** }

赋值语句 $S.nextlist$:= **nil** 将 $S.nextlist$ 初始化为空列表。

(8) $L \rightarrow L_1\ ;\ M\ S$ { *backpatch*($L_1.nextlist$, $M.quad$);
$L.nextlist$:= $S.nextlist$ } 505

执行顺序中 L_1 后面的语句应是 S 的起始位置。这样 $L_1.nextlist$ 列表由 $M.quad$ 回填转到 S 的代码的起始位置。

(9) $L \rightarrow S$ { $L.nextlist$:= $S.nextlist$ }

注意，在上述语义规则中除了规则(2)和规则(5)以外，均未生成新的四元式。所有其他代码都是由与赋值语句和表达式相关联的语义动作生成的。控制流所做的是进行适当的回填，以便赋值语句与布尔表达式的计算正确地连接起来。

8.6.4 标号和goto

最基本的改变控制流的程序语言结构是标号和 goto。当编译器遇到像 goto L这样的语句时，它必须检查在 goto 语句的作用域中恰好有一条标号为 L 的语句。如果此标号已经出现，无论它是在标号声明语句中还是作为某条源语句的标号出现，符号表中都一定有一个表项，记录编译器生成的标记，它标识标号为 L 语句的第一条三地址指令。翻译时我们生成 goto 三地址语句，目标是这个编译器生成的标号。

在源程序中第一次遇到标号L时，无论是在声明语句中还是作为前向 goto 的目标标号，我们都将把 L 插入到符号表中并为 L 生成一个符号标号。

8.7 过程调用

过程[⊖]是一个如此重要而又频繁使用的程序设计结构，以至于编译器必须为过程调用和返回生成优良的代码。处理过程的参数传递、调用和返回的运行时例程是运行时支撑程序包的一部分。我们在第7章讨论了实现运行时支撑程序包的各种不同机制，本节我们讨论典型情况下为过程调用和返回生成的代码。

考虑下面一个简单的过程调用语句的文法：

(1) $S \rightarrow$ **call id** (*Elist*)
(2) *Elist* → *Elist* , *E*
506 (3) *Elist* → *E*

8.7.1 调用序列

正如第7章所讨论的那样，过程调用的翻译包括一个调用序列，它是进入和离开每一个过程所执行的一序列动作。即使对相同语言的实现，调用序列也可以是不同的，下面是典型情况下发生的动作。

当出现一个过程调用时，必须为被调用过程的活动记录分配空间，而且必须计算被调过程的参数并将其放到某已知位置，使其对被调过程可用。另外还必须建立环境指针以使被调过程可以访问外围程序块中的数据。还应保存调用过程的运行状态以便调用完成后恢复原来的执行。返回地址也要保存在已知的位置，返回地址是被调过程执行完后的转移地址，它经常是调用过程中紧跟调用指令之后的指令地址。最后，还要生成一条转移到被调过程代码起始位置的跳转语句。

当从过程返回时，也必须执行一些动作。如果被调过程是一个函数，则必须将返回结果存到指定的位置。调用过程的活动记录需要被恢复。还必须生成一条转移到调用过程返回地址的跳转语句。

调用过程和被调过程的运行时任务没有严格的区分。但源语言、目标机器和操作系统经常会强加一些偏向某种方案的要求。

8.7.2 一个简单的例子

下面考虑一个简单的例子，其中参数传递采用传址方式，存储分配为静态分配。这种情况下，我们可以使用 param 语句本身作为参数的占位符。用寄存器将指向第一个 param 语句的指针传给被调过程，然后就可以用相对于该指针的适当的偏移获得指向每个参数的指针。为这种类型的调用生成三地址码时，只要为表达式参数生成计算参数的三地址语句即可，然后跟以 param 三地址语句列表，每个参数对应一条 param 语句。如果我们不想将参数计算语句同 param 语句混在一起，则对 **id**(*E*, *E*, ⋯, *E*) 中的每个 *E*，我们都需要保存 *E.place* 的值。[⊖]

保存这些值的方便的数据结构是队列，一个先进先出列表。产生式*Elist*→*Elist*, *E* 的语义例
507 程将包含一条将 *E.place* 存放到队列 *queue* 中的指令。然后产生式 *S*→**call id**(*Elist*) 的语义例程将为 *queue* 中的每一项生成一条 param 语句，并将这些语句放在计算参数表达式的语句之后。计算参数表达式的语句是在将参数归约为 *E* 时生成的。下面的语法制导翻译采用了这些思想。

⊖ 我们使用的术语“过程”包含了函数。函数是带有返回值的过程。

⊖ 如果通过将参数放入栈中来将其传递给被调用过程，就像动态数据分配那样，则没有理由不把参数计算同 param 语句合在一起。在代码生成时 param 语句将被把参数压入栈的代码取代。

(1) $S \rightarrow$ **call id** (*Elist*)
{ **for** *queue* 中的每个 *p* **do** }
emit('**param**' *p*);
emit('**call**' **id**.*place*) }

S 的代码就是 *Elist* 的代码，它计算各参数的值，后面跟着与每个参数对应的 `param` *p* 语句，接着是 **call** 语句。**call** 语句中没有生成计算参数个数的指令，但我们可以用上一节计算 *Elist.ndim* 的方式计算它。

(2) *Elist* → *Elist* , *E*
{ 将*E.place* 添加到*queue*的队尾 }
(3) *Elist* → *E*
{ 将*queue*初始化为只包含*E.place* }

这里，首先将 *queue* 置空，然后获得一个指向符号表中表示 *E* 值的名字的位置的指针。

练习

8.1 把算术表达式 `a*-(b+c)` 翻译成如下形式：
a) 语法树。
b) 后缀表示。
c) 三地址码。

8.2 把表达式 `-(a+b)*(c+d)+(a+b+c)` 翻译成如下形式：
a) 四元式。
b) 三元式。
c) 间接三元式。

8.3 设有如下 C 程序：

```
main()
{
    int i;
    int a[10];
    i = 1;
    while (i <= 10) {
       a[i] = 0; i = i + 1;
    }
}
```

将其中的可执行语句翻译成如下形式： 508
a) 语法树。
b) 后缀表示。
c) 三地址码。

* 8.4 证明：如果所有的运算符都是二元的，那么当且仅当下列两个条件成立时，运算符和运算对象的串是后缀表达式：
(1) 运算符的个数正好比运算对象的个数少一个。
(2) 该表达式的每个非空前缀的运算符数少于运算对象数。

8.5 修改图8-11中计算说明语句中名字类型和相对地址的翻译模式，以允许在形如 $D \rightarrow$ **id**:*T* 的声明中可以出现一串名字表而不只是一个名字。

8.6 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) 给出 a*-(b+c) 的前缀形式。

** b) 证明：所有的语义动作都是打印动作并且所有的动作都出现在产生式右部末端的翻译模式不能把中缀表达式翻译成前缀表达式。

c) 给出把中缀表达式翻译成前缀表达式的语法制导定义。你能用第5章的哪种方法？

8.7 试编写一个程序，用来实现图8-24中将布尔表达式翻译成三地址码的语法制导定义。

8.8 修改图8-24中的语法制导定义，以生成2.8节中的栈式机器代码。

8.9 图8-24中的语法制导定义把 $E \rightarrow \mathbf{id}_1 < \mathbf{id}_2$ 翻译成下面一对语句：

if $\mathbf{id}_1 < \mathbf{id}_2$ **goto** ···
goto ···

我们可以用一条语句

if $\mathbf{id}_1 \geqslant \mathbf{id}_2$ **goto** _

来代替它，当 E 为真时执行后继代码。修改图8-24中的定义，使之生成这种性质的代码。

8.10 试编写一个程序，用来实现图8-23中给出的控制流语句的语法制导定义。

8.11 编程实现8.6节中给出的回填算法。

8.12 用8.3节的翻译模式，把下列赋值语句翻译成三地址码：

509

```
A[i,j] := B[i,j] + C[A[k,l]] + D[i+j]
```

* 8.13 在某些像 PL/I这样的语言中，允许给一串名字赋予一串属性，而且允许声明互相嵌套。下面是这一问题的文法：

$$
\begin{aligned}
D &\rightarrow \mathit{namelist}\ \mathit{attrlist} \\
&\quad \mid (\ D\ )\ \mathit{attrlist} \\
\mathit{namelist} &\rightarrow \mathbf{id}\ ,\ \mathit{namelist} \\
&\quad \mid \mathbf{id} \\
\mathit{attrlist} &\rightarrow A\ \mathit{attrlist} \\
&\quad \mid A \\
A &\rightarrow \mathbf{decimal} \mid \mathbf{fixed} \mid \mathbf{float} \mid \mathbf{real}
\end{aligned}
$$

$D \rightarrow (D)\mathit{attrlist}$ 的含义是出现在括号中声明中提到的所有名字都具有属性 *attrlist*，无论有多少层嵌套。注意 n 个名字 m 个属性的声明将导致符号表中 nm 条信息。给出这个文法定义的声明的语法制导定义。

8.14 C语言中，for 语句的格式为：

```
for ( e1 ; e2 ; e3 ) stmt
```

它与下列语句含义相同：

```
e1;
while ( e2 ) {
    stmt;
    e3;
}
```

试构造将 C 格式的 for 语句翻译成三地址码的语法制导定义。

8.15 标准 Pascal 语言中，语句

for *v* := *initial* **to** *final* **do** *stmt*

和下面的代码序列含义相同：

```
begin
    t1 := initial; t2 := final;
    if t1 ≤ t2 then begin
        v := t1;
        stmt
        while v ≠ t2 do begin
            v := succ(v);
            stmt
        end
    end
end
```

510

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，而 *final* = MAXINT 时，该程序将做什么动作？其中，MAXINT 是目标机器允许的最大整数。

* b) 试构造为 Pascal 的 for 语句生成三地址码的语法制导定义。

参考文献注释

UNCOL(Universal Compiler Oriented Language)是20世纪50年代中期探索的一种通用中间语言。给定一个UNCOL程序，Strong et al. [1958]的报告中说明了怎样利用下面的方式来构造编译器，即将相应源语言的前端程序和相应目标机的后端程序连在一起。报告中给出的自举技术可以用于重置编译器的目标机（见11.2节）。Steel[1961]中含有UNCOL的原始提议。

可重置目标的编译器含有一个前端程序，它可以和几个用于实现相应目标机语言的后端程序放在一起。Neliac是早期带有可重置目标编译器（Huskey, Halstead, and McArthur[1960]）的一种语言，其编译器就是用该语言本身书写的。其他可重置目标机编译器可以参见：Richards [1971]中的BCPL编译器、Nori et al. [1981]中的Pascal编译器及Johnson[1979]中的C编译器。Newey, Poole, and Waite[1972]将后端可变的思想应用在宏处理器、文本编辑器和Basic编译器中。

已经有许多种方法可以接近UNCOL在m种机器上实现n种语言的理想（通过写出m个前端程序和n个后端程序，对应有$n \times m$个不同的编译器）。一种方法是在一个现存的编译器上为一种新语言增加一个前端程序。Feldman[1979b]描述了为Johnson[1979]和Ritchie[1979]的C编译器增加了Fortran 77前端处理程序。Davidson and Fraser[1984b]、Leverett et al. [1980]以及Tanenbaum et al. [1983]中描述了容纳多个前端和后端程序的编译器组织的设计。

Davidson and Fraser[1984b]中使用的“并”和“交”抽象机突出了中间表示中允许的运算符集的作用。交机器的指令集和寻址方式是有限的，因此生成中间代码时前端程序不必做许多 511

选择。并机器提供了实现源级结构的可选方法。因为所有这些可选方法可能不必由所有的目标机器直接实现，所以机器的更丰富的指令集可能允许对目标机器的某些依赖。对其他中间代码如语法树和三地址码有相似的评论。Fraser and Hanson[1982]讨论了用与机器无关的操作访问运行时栈的表达方式。

Randell and Russell[1964]以及Grau，Hill, and Langmaack[1967]对Algol 60的实现进行了详细的讨论。Freiburghouse[1969]讨论了PL/I，Wirth[1971]对Pascal，Branquart et al. [1976]对Algol 68也分别进行了讨论。

Minker and Minker[1980]以及Giegerich and Wilhelm[1978]中讨论了怎样为布尔表达式生成
512 最优代码。练习8.15选自Newey and Waite[1985]。

第9章 代码生成

我们的编译器模型的最后一个阶段是代码生成器，它将源程序的中间表示作为输入，并产生等价的目标程序作为输出，如图9-1所示。不管代码生成的前面是否有代码优化阶段，本章给出的代码生成技术都是适用的。代码优化阶段试图将中间表示转换成一种可以生成更有效率的目标代码的形式。代码优化将放在下一章中详细讨论。

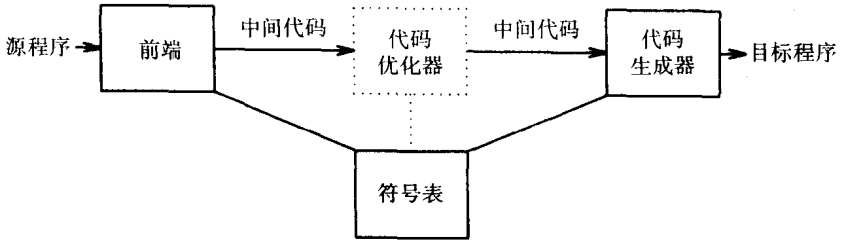


图9-1 代码生成器在整个编译中的位置

对代码生成器的要求是严格的，它输出的代码必须正确而且质量高。质量高的含义是它应该有效地利用目标机器的资源，此外，代码生成器本身也应该高效地运行。

从理论上讲，产生最优代码的问题是不可判定的。在实践中，能够产生好的（而不必是最优的）代码的启发式技术就很令人满意了。之所以选择启发式技术是因为仔细设计的代码生成算法比快速设计的算法产生的代码快好几倍。

9.1 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语言和操作系统，但很多问题（如存储管理、指令选择、寄存器分配和计算次序等）几乎是所有的代码生成器所固有的问题。本节考察设计各种代码生成器的公共问题。

9.1.1 代码生成器的输入

代码生成器的输入包括前端产生的源程序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时地址。

正如我们在前一章中指出的，中间语言有多种选择，包括线性表示法（如后缀式），三地址表示法（如四元式），虚拟机器表示法（如栈式机器代码），图形表示法（如语法树和无环有向图）。虽然本章的算法是用三地址码、树和无环有向图来表述的，但其中许多技术亦可用于其他中间表示。

假定在代码生成前，编译器的前端已经将源程序扫描、分析和翻译成为足够详细的中间表示，这样，中间语言中名字的值可以表示为目标机器能够直接操作的量(位、整数、实数、指针等)。还假定已经完成了必要的类型检查，因此类型转换算符已插在需要的地方，而且明显的语义错误(如试图把浮点数作为数组下标)等都已经被检测出来了。这样，代码生成阶段可以认为其输入中没有错误。在有些编译器中，这种语义检查和代码生成一起完成。

9.1.2 目标程序

代码生成器的输出是目标程序。像中间代码那样，输出的形式也是多种多样的：绝对机器语言、可重定位的机器语言或汇编语言。

生成绝对机器语言程序作为输出的好处是，它可以被放在内存中的固定地方并且立即被执行，这样，小的程序可以被迅速地编译和执行。一些面向学生的编译器，如WATFIV和PL/C等，就产生绝对机器代码。

生成可重定位的机器语言程序(目标模块)作为输出允许分别编译子程序，一组可重定位模块可以由连接装配器连接在一起并装入执行。虽然产生可重定位模块必须增加额外的开销来连
514 接和装配，但带来的好处是灵活性：可以分别编译子程序，而且可以从目标模块中调用其他事先编译好的程序模块。如果目标机器不能自动处理重定位，编译器必须给装配器提供显式的重定位信息，以连接分别编译的程序段。

生成汇编语言程序作为输出可以使代码生成的过程变得容易一些，我们可以产生符号指令并利用汇编器的宏功能来帮助生成代码，付出的代价是代码生成后的汇编步骤。由于产生汇编代码并不重复汇编器的整个工作，因此这也是一种合理的选择，尤其对于内存小、编译必须分成几遍的情况更是这样。为了增加可读性，我们在本章中用汇编代码作为目标语言。但是必须强调的是，只要可以从符号表中的偏移或其他信息计算出地址，代码生成器就可以像产生符号地址一样容易地为名字产生可重定位的或绝对的地址。

9.1.3 存储管理

把源程序中的名字映射成运行时数据对象的地址是由前端和代码生成器共同完成的。在上一章中，我们假定三地址语句中的名字是指符号表中该名字的表项。在8.2节，符号表表项是在检查过程中的声明时建立的，声明的类型决定了被声明名字的宽度，即所需存储空间的大小。从符号表信息可以得到名字在过程数据区的相对地址。在9.3节，我们将描述数据区的静态和栈式存储分配的实现，并说明如何将中间表示中的名字转换为目标代码中的地址。

如果要生成机器代码，必须将三地址语句的标号转变成指令的地址。该过程与8.6节的回填技术类似。假定标号是指四元式数组的四元式编号，当依次扫描四元式时，可以推断出为该四元式生成的第一条指令地址，这只要维护一个计数器，记住到目前为止产生的指令已用了多少个字就可以了。该计数可以用一个额外的域保存在四元式数组中，因而如果碰到 j: goto i 这样的引用，并且 i 小于 j，j 是当前四元式的编号，就可以直接产生跳转指令，目标地址等于第i 个四元式的第一条指令的机器地址。但是如果是向前跳，即 i 大于 j，则必须把为四元式 j 生成的第一条机器指令的地址记在与四元式 i 相关联的列表中，然后，当处理到四元式 i 时，
515 为所有向前跳转到 i 的指令回填上适当的机器地址。

9.1.4 指令选择

目标机器指令集的性质决定了指令选择的难易程度，指令集的一致性和完整性是重要的因素。例如，如果目标机器不能以一致的方式支持各种数据类型，那么每种例外都需要专门的处理。

指令速度和机器的特点也是重要因素。如果不考虑目标程序的效率，则指令的选择是直截了当的。对每一类三地址语句，可以设计它的代码骨架，它将给出为这种语句生成的目标代码的轮廓。例如，若x、y和z都采用静态存储分配，则可以将每个形如x := y+z的三地址语句翻译成如下的代码序列：

```
MOV y,R0   /* 将y装入寄存器R0中 */
ADD z,R0   /* 将z加到R0上 */
MOV R0,x   /* 将R0存入x中 */
```

不幸的是，这种逐条语句的代码生成方法常常产生质量低劣的代码。例如，语句序列

```
a := b + c
d := a + e
```

将被翻译成

```
MOV b,R0
ADD c,R0
MOV R0,a
MOV a,R0
ADD e,R0
MOV R0,d
```

其中，第四条指令是多余的，如果以后不再使用a的话，那么第三条指令也是多余的。

所生成代码的质量取决于它的执行速度和长度。指令集丰富的目标机器可能提供几种办法实现某一操作，由于不同实现方法的开销可能大不一样，因此中间代码的简单的翻译会产生正确但效率可能无法接受的目标代码。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
```

虽然指令速度对设计好的代码序列是需要的，但往往很难获得精确的时间信息。要确定哪个指令序列对给定的三地址结构是最优的，可能还要用到该结构出现的上下文知识。9.12节将讨论构造指令选择器的工具。 516

9.1.5 寄存器分配

操作数在寄存器中的指令通常要比操作数在内存中的指令短一些，执行也要快一些。因此，充分利用寄存器对生成好的代码尤其重要。寄存器的使用可以分成两个子问题：

1. 在*寄存器分配*期间，在程序的某一点选择要驻留在寄存器中的变量集。
2. 在随后的*寄存器指派*阶段，挑出变量将要驻留的具体寄存器。

选择最优的寄存器指派方案非常困难，这个问题是NP完全的。该问题还会进一步复杂化，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。

某些机器要求对某些操作数和结果使用*寄存器对*（偶寄存器和下一个奇寄存器）。例如，在IBM System/370机器上，整数乘和整数除就要使用寄存器对。乘法指令的形式是：

```
M    x, y
```

其中，x是被乘数，是偶/奇寄存器对的偶寄存器，被乘数的值从该对的奇寄存器中取；乘数y是单个寄存器。积占据整个偶/奇寄存器对。

除法指令的形式是：

```
D    x, y
```

其中，64位的被除数占据一个偶/奇寄存器对，它的偶寄存器是x，y代表除数。除完以后，偶寄存器保存余数，奇寄存器保存商。

现在考虑图9-2中的两个三地址码序列，它们仅有的区别是第二条语句的操作符不同，其最短汇编代码序列在图9-3中给出。

```
t := a + b          t := a + b
t := t * c          t := t + c
t := t / d          t := t / d
   a)                  b)
```

图9-2 两个三地址码序列

Ri 代表寄存器 i。L, ST 和 A 分别代表装入、存储和加。SRDA[⊖] R0, 32 把除数移入 R1，并清 R0，使得所有位都等于它的符号位。注意，装入 a 的寄存器的最佳选择依赖于 t 以后还有什么用。9.7节将讨论寄存器的分配策略。

```
L    R1, a          L    R0, a
A    R1, b          A    R0, b
M    R0, c          A    R0, c
D    R0, d          SRDA R0, 32
ST   R1, t          D    R0, d
                    ST   R1, t
     a)                  b)
```

图9-3 最优的机器代码序列

9.1.6 计算次序的选择

计算执行的次序会影响目标代码的效率。我们将看到，某些计算次序比其他次序需要较少的寄存器来保存中间结果。选择最佳次序也是一个NP完全问题。开始，为避免该问题，我们只讨论按照中间代码生成器产生的三地址语句的次序来产生目标代码。

9.1.7 代码生成方法

毋庸置疑，评判代码生成器的最重要准则是产生正确的代码。代码生成器可能面临的特殊情况的数量使得正确性特别重要。除此以外，易于实现、测试和维护也是重要的设计目标。

9.6节中包括一个简单的代码生成算法，它使用操作数后续使用情况的信息为寄存器机器产生代码。该算法依次考虑每个语句，并将操作数尽可能长时间地保持在寄存器中。这种算法的输出可以使用9.9节讨论的窥孔优化技术进行改进。

517 ~ 518

9.7节通过考虑中间代码的控制流，讨论了寄存器的较优使用技术，它强调把寄存器分配给内循环中频繁使用的变量。

9.10节和9.11节给出了一些树制导代码选择技术，它们适用于构造可重置目标的代码生成器。可移植的C编译器 PCC 就使用这种代码生成器，并已移植到了多种机器上。UNIX操作系统可以运行在很多种机器上就归功于 PCC 的可移植性。9.12节将说明怎样将代码生成器看作是重写树（tree-rewriting）的过程。

9.2 目标机器

熟悉目标机器及其指令集是设计一个好的代码生成器的先决条件。但是，在代码生成的一般性讨论中，不可能对目标机器的细节描述到足够详细的程度，因而难以对该目标机器的一个完整的语言生成好的代码。本章中，我们将采用可作为几种微型机代表的寄存器机器作为目标机器，不过，本章中提出的一些代码生成技术亦可用于许多其他类型的机器。

我们的目标机器是一个字节可寻址机器，4字节组成一个字，并有 n 个通用寄存器 R0, R1, …, Rn-1。它有如下两地址指令形式：

op source, destination

其中 *op* 是操作码，*source* 和 *destination* 称为源和目的，是数据域。该机有如下的操作码：

MOV （将源移到目的中）

⊖ Shift Right Double Arithmetic。

```
ADD   (将源加到目的中)
SUB   (在目的中减去源)
```

其他指令将在需要时再介绍。

由于源和目的域不足以保存内存地址，所以这些域的某些位用来指明一条指令之后的包含操作数和(或)地址的那些字。一条指令的源和目的是通过将寄存器和带有地址模式的内存单元结合起来确定的。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相关的开销如下所示：

地址模式	汇编形式	地址	增加的开销
绝对地址	M	M	1
寄存器	R	R	0
索引	*c*(R)	*c* + *contents*(R)	1
间接寄存器	*R	*contents*(R)	0
间接索引	**c*(R)	*contents*(*c* + *contents*(R))	1

519

当用内存地址M或寄存器R作为源或目的时，它们代表自身。例如，指令

```
MOV R0,M
```

把寄存器R0的内容存入内存单元M中。

从寄存器R中的值的地址偏移*c*可写成*c*(R)。那么，指令

```
MOV 4(R0),M
```

把值

contents(4 + *contents*(R0))

存入内存单元M中。

后面的两个间接模式由前缀*表示。这样，指令

```
MOV *4(R0),M
```

将值

contents(*contents*(4 + *contents*(R0)))

存入内存单元M中。

最后一种地址模式允许源是一个常数：

地址模式	汇编形式	常数	增加的开销
字面常数	#*c*	*c*	1

因此，指令

```
MOV #1,R0
```

把常数1保存在寄存器R0中。

指令开销

我们把与源地址模式和目的地址模式相关的开销(在上面的地址模式表中表示为增加的开销)再加上1作为一条指令的开销，这个开销与指令的长度(以字计算)相对应。寄存器地址模式的开销是0，而内存单元模式与常数模式的开销是1，因为这些操作数必须和指令存在一起。

如果空间是至关重要的，我们应该使指令的长度极小化。这样做有一个额外的重要好处，对于大多数机器和大多数指令而言，从内存取指令的时间要超过执行指令的时间，这样，极小

520 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将寄存器R0的内容复制到寄存器R1中。这条指令的开销是1，因为它仅占内存中的一个字。

2. 存储指令MOV R5，M将寄存器R5的内容复制到内存单元M中。这条指令的开销是2，因为内存单元M的地址存放在该指令之后的一个字中。

3. 指令ADD #1，R3将寄存器R3的内容增加1。这条指令的开销是2，因为常数1必须出现在指令后面的下一个字中。

4. 指令SUB 4(R0)，*12(R1)将值

contents (*contents* (12 + *contents* (R1))) − *contents* (4 + *contents* (R0))

存入目的地址 *12(R1)中。这条指令的开销是3，因为常数4和12要存放在指令之后的下两个字中。

考虑为形如a := b + c的三地址语句会产生什么样的代码，可以看出为上述机器产生代码的一些困难之处，其中名字 b 和 c 是在由这两个名字表示的不同存储单元中的简单变量。这条语句可以通过许多不同的指令序列来实现，下面是几个例子：

```
1.  MOV   b, R0
    ADD   c, R0          开销 = 6
    MOV   R0, a

2.  MOV   b, a
    ADD   c, a           开销 = 6
```

假定R0、R1、R2中分别存放了a、b和c的地址，我们可以使用如下指令序列：

```
3.  MOV   *R1, *R0
    ADD   *R2, *R0       开销 = 2
```

假定R1和R2中分别包含b和c的值，并且b的值在这个赋值以后不再需要，则可以使用如下指令序列：

```
4.  ADD   R2, R1
    MOV   R1, a          开销 = 3
```

可以看出，要为这种机器生成好的代码，必须有效地使用它的寻址能力。另外，如果一个
521 名字在不久后将被引用，则应尽可能地将该名字的左值或右值保存在寄存器中。

9.3 运行时存储管理

正如在第7章中所看到的，语言中过程的语义决定了执行期间名字如何与存储空间相联系。过程的一次执行所需要的信息存放在一个称为活动记录的存储块中，过程中的局部名字的存储空间也在活动记录中。

在本节中，我们讨论运行时管理活动记录应生成哪些代码。7.3节给出了两个标准的存储分配策略，即静态分配和栈式分配。对于静态分配，内存中活动记录的位置在编译时便已确定。对于栈式分配，过程的每次执行都将在栈顶压入一个新的活动记录，活动结束时将该记录弹出

⊖ 开销准则必须是有意义的而不是实际的。为一条指令分配整个字可以简化确定开销的准则。对时间的更准确估计将考虑一条指令是否需要将操作数的值及其地址（用该指令找到的）从内存装入。

栈。在本章后面，我们将考虑一个过程的目标代码如何引用活动记录中的数据对象。

正如我们在7.2节所看到的，一个过程的活动记录包含存放下列信息的域：参数、结果、机器状态信息、局部数据、临时变量等等。本节，我们将利用保存返回地址的机器状态域和局部数据域来阐述分配策略。假定其他域的处理如第7章所述。

由于运行时活动记录的分配和释放是作为过程调用和返回序列的一部分，我们集中讨论下面的三地址语句:

1. call（调用）。
2. return（返回）。
3. halt（中断）。
4. action（动作），其他语句的占位符。

例如，图9-4中过程c和p的三地址码只包含这几种语句。活动记录的大小和布局是通过符号表中的名字信息传递给代码生成器的。为清晰起见，我们在图9-4中给出活动记录的布局而不是符号表的表项。

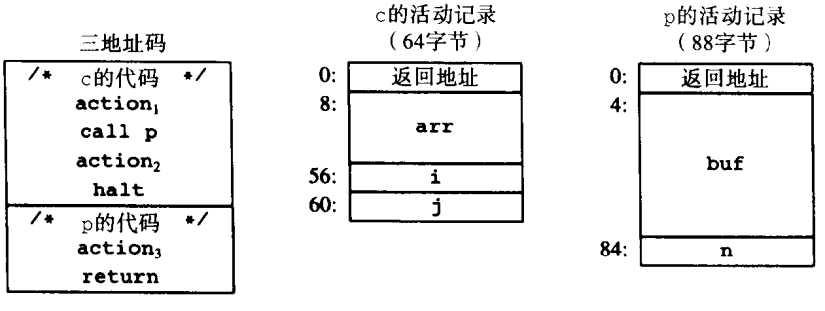


图9-4 代码生成器的输入

如7.2节所述，我们假定运行时内存被划分成代码区、静态数据区和一个栈区(这里不用额外的堆区)。

9.3.1 静态分配

考虑实现静态分配所需的代码。一条中间代码的call语句由两条目标机器指令来实现。一条MOV指令保存返回地址，而另一条GOTO语句将控制转移到被调用过程的目标代码:

```
MOV    #here +20, callee.static_area
GOTO   callee.code_area
```

属性 *callee.static_area* 和 *callee.code_area*是常数，分别指向被调用过程活动记录的地址及其第一条指令。在 MOV 指令中源 *#here* + 20 是一个字面常量，作为返回地址；它是紧跟在 GOTO 指令之后的那条指令的地址。(从9.2节的讨论可知，三个常数加上两条调用指令的开销为5个字即20个字节。)

过程的代码以一条返回到调用过程的指令结束。另外，因为第一个过程没有调用者，因此最后的指令是 HALT，它将控制返回到操作系统。从过程 *callee* 的返回由

```
GOTO  *callee.static_area
```

来实现。它将控制转移到存放在活动记录开始位置的返回地址处。

例9.1 图9-5的代码是由图9-4的过程 c 和过程 p 创建的。我们用伪指令 ACTION 来实现

语句 action。它代表与本次讨论无关的三地址码。另外，我们假定过程的代码分别从地址100和200开始，而且每条 ACTION 指令占用20个字节。过程的活动记录分别从地址300和地址364开始静态分配。

从地址100处开始的指令实现第一个过程c的语句：

```
action1; call p; action2; halt
```

因此，程序是从地址100处的 $ACTION_1$ 指令开始执行的，地址120处的 MOV 指令将返回地址
522 ~ 523 140存入机器状态域中，即存入过程 p 的活动记录的第一个字中。地址132处的 GOTO 指令将控制转移到被调用过程的目标代码。

```
                        /* c的代码 */
100:  ACTION1
120:  MOV #140, 364     /* 保存返回地址140 */
132:  GOTO 200          /* 调用p */
140:  ACTION2
160:  HALT
      ...

                        /* p的代码 */
200:  ACTION3
220:  GOTO *364         /* 返回到保存在364中的地址 */
      ...

                        /* 300~363保存c的活动记录 */
300:                    /* 返回地址 */
304:                    /* c的局部数据 */
      ...

                        /* 364~451保存p的活动记录 */
364:                    /* 返回地址 */
368:                    /* p的局部数据 */
```

图9-5 图9-4中输入的目标代码

由于140通过以上调用序列已存放到地址364中，从而当执行地址220的 GOTO 语句时，*364表示140。因此控制将返回到地址140，恢复调用过程 c 的执行。 □

9.3.2 栈式分配

通过对活动记录的存储单元使用相对地址，可以将静态分配改造成栈式分配。一个过程的活动记录位置直到运行时才能知道。在栈式分配中，该位置通常存放在寄存器中，因此活动记录中的字可以通过相对于该寄存器中值的偏移来访问。目标机器的索引地址模式可以很方便地实现该目的。

活动记录中的相对地址可以看成该活动记录中对任意已知位置的偏移，如我们在7.3节中所见到的那样。为方便起见，我们将通过在寄存器SP中保存一个指向栈顶活动记录开始位置的指针而使用正的偏移。当发生过程调用时，调用过程给SP一个增量，并将控制转移到被调用过程。当控制返回到调用过程时，再将 SP 减去原来的增量，从而释放了被调用过程的活动
524 记录。[⊖]

第一个过程的代码通过将 SP 置为内存中栈区的开始位置来初始化栈：

⊖ 对于负的偏移，我们可以让SP指向栈的末尾并让被调用过程增加SP。

```
MOV #stackstart, SP          /* 初始化栈 */
第一个过程的代码
HALT                         /* 终止过程的执行 */
```

过程调用序列给SP一个增量，并保存返回地址，将控制转移到被调用过程。

```
ADD #caller.recordsize, SP
MOV #here+16, *SP            /* 保存返回地址 */
GOTO callee.code_area
```

属性*caller.recordsize*表示活动记录的大小，因此ADD指令使SP指向下一个活动记录的开始位置。MOV指令的源*#here*+16是跟在GOTO之后的指令的地址，它被存放在SP所指向的地址中。

返回序列包含两部分。被调用过程采用如下指令将控制返回到调用过程：

```
GOTO *0(SP)   /* 返回到调用过程 */
```

在GOTO语句中使用 *0(SP)的原因是我们需要二级间接地址：0(SP)是活动记录第一个字的地址，而 *0(SP) 是保存在 0(SP) 中的返回地址。

返回序列的第二部分在调用过程中，它减去 SP，因而将 SP 恢复到以前的值。也就是说，减去以后SP将指向调用者活动记录的开始：

```
SUB #caller.recordsize, SP
```

调用序列的更深入的讨论以及调用过程和被调用过程的折衷参见7.3节。

例9.2 图9-6中的程序是7.1节所讨论的 Pascal 程序的三地址码，过程 q 是递归的，因而在同一时刻可能有q的多个活动记录存活。

假设过程 s、p 和 q 的活动记录的大小已在编译时分别确定为 *ssize*、*psize* 和 *qsize*。每个活动记录的第一个字存放返回地址。三个过程的代码分别从地址100、200、300开始，并且栈从地址600开始。图9-6中程序的目标代码如下：

```
                    /* s的代码 */
100:  MOV #600, SP  /* 初始化栈 */
108:  ACTION1
128:  ADD #ssize, SP  /* 调用序列开始 */
136:  MOV #152, *SP   /* 压入返回地址 */
144:  GOTO 300        /* 调用q */
152:  SUB #ssize, SP  /* 恢复SP */
160:  ACTION2
180:  HALT
      ...
                    /* p的代码 */
200:  ACTION3
220:  GOTO *0(SP)     /* 返回 */
      ...
                    /* q的代码 */
300:  ACTION4         /* 条件转移到456 */
320:  ADD #qsize, SP
328:  MOV #344, *SP   /* 压入返回地址 */
336:  GOTO 200        /* 调用p */
344:  SUB #qsize, SP
```

三地址码

```
/* s的代码 */
  action1
  call q
  action2
  halt
/* p的代码 */
  action3
  return
/* q的代码 */
  action4
  call p
  action5
  call q
  action6
  call q
  return
```

图9-6 说明栈式分配的三地址码

```
352:  ACTION5
372:  ADD #qsize, SP
380:  MOV #396, *SP    /* 压入返回地址 */
388:  GOTO 300         /* 调用q */
396:  SUB #qsize, SP
404:  ACTION6
424:  ADD #qsize, SP
432:  MOV #448, *SP    /* 压入返回地址 */
440:  GOTO 300         /* 调用q */
448:  SUB #qsize, SP
456:  GOTO *0(SP)      /* 返回 */
      ...

600:                   /* 栈从此处开始 */
```

525 ~ 526

我们假设 $ACTION_4$ 中包含一个条件转移指令转移到q的返回地址456；否则递归过程 q 将永远递归调用下去。在下面的例子中，我们将考虑该程序的执行，q 的第一次调用没有立即返回，但接下来的所有调用则不然。

如果 *ssize*、*psize* 和 *qsize* 分别为20、40和60，则 SP 通过地址100的第一条指令初始化为600，即栈的开始地址。由于 *ssize* 是20，因而在控制从 s 转移到 q 时，SP 的值为620。接下来，当 q 调用 p 时，地址320处的指令将 SP 增加到680，即 p 的活动记录的开始；当控制返回到 q 时，SP 又变成620。如果下两次对 q 的递归调用是立即返回的，则这次执行期间 SP 的最大值是680。但要注意，最后一个被使用的栈地址是739，因为 q 的活动记录从680开始并占用了60个字节。 □

9.3.3 名字的运行地址

过程的存储分配策略和活动记录中局部数据的安排决定了如何访问名字的存储单元。在第8章，我们曾假定三地址语句中的名字是一个指向符号表中对应该名字表项的指针，这种方法有个重要的好处：使编译器的可移植性更强，因为即使编译器被转移到另一个不同的机器上，而这种机器需要不同的运行组织（如 display 表可放在寄存器中而不是内存中），前端也无须改变。另一方面，在生成中间代码时产生明确的访问序列在优化编译器时也大有好处，因为它使得优化器可以利用在简单的三地址语句中甚至根本看不到的细节。

无论哪种情况，名字最终必须由访问存储地址的代码来取代。我们考虑简单的三地址语句 x := 0的一些细节。处理完过程的声明以后，假设符号表中x的表项包含一个x的相对地址12，
527 首先，假定x是在一个从 *static* 开始的静态分配的区域中，那么 x 的实际运行地址是 *static* + 12。尽管编译器可在编译时确定 *static* + 12 的值，但是当存取名字的中间代码生成的时候，可能还不知道静态区域的位置。在这种情况下，它必须生成三地址码以“计算” *static* + 12，该计算将在代码生成阶段进行，或者在程序运行前由装配器执行。因此，赋值语句x := 0将被翻译成

```
static[12] := 0
```

如果静态区域从地址100开始，则该语句的目标代码为

```
MOV #0, 112
```

另一方面，假定我们采用类 Pascal 语言，并且如7.4节所讨论的，用 display 表存取非局部名字，又假定 display 表存放在寄存器中，而且 x 是一个活动过程的局部变量，该活动过程的 display 表指针放在寄存器 R3 中。那么，我们可以将语句 x := 0翻译成如下的三地址语句：

```
t1 := 12 + R3
```

```
*t1 := 0
```

其中，t_1 中存放的是 x 的地址。该序列可以用下面的一条机器指令实现：

```
MOV #0, 12(R3)
```

注意寄存器R3中的值不能在编译时确定。

9.4 基本块和流图

三地址语句的一种图形表示叫做流图，即使该图并未由代码生成算法明显地构造出来，它对理解代码生成算法也非常有用。流图中的节点表示计算，边表示控制流。在第10章，我们将充分利用流图作为从中间代码收集信息的工具。某些寄存器分配算法使用流图来寻找消耗程序大部分运行时间的内循环。

9.4.1 基本块

基本块是一个连续的语句序列，控制流从它的开始进入，并从它的末尾离开，不可能有中断或分支（末尾除外）。下面的三地址语句序列就形成一个基本块：

528

```
t1 := a * a
t2 := a * b
t3 := 2 * t2
t4 := t1 + t3
t5 := b * b
t6 := t4 + t5
```

(9-1)

三地址语句 x := y+z 称为定义 x 并使用（或引用）y 和 z。对于基本块中的一个名字，如果它的值在某一点以后还要被引用的话，当然，也可能是在另一个基本块中引用它，我们称它在程序中的该点是*活跃*的。

下面的算法可用于把三地址语句序列划分成基本块。

算法9.1 划分成基本块。

输入：一个三地址语句序列。

输出：一个基本块列表，其中每个三地址语句仅在一个块中。

方法：

1. 首先确定*入口语句*（即基本块的第一个语句）的集合。所用规则如下：
   i) 第一个语句是入口语句。
   ii) 任何能由条件转移语句或无条件转移语句转移到的语句都是入口语句。
   iii) 紧跟在转移语句或条件转移后面的语句是入口语句。
2. 对于每个入口语句，其基本块由它和直到下一个入口语句（但不含该入口语句）或程序结束的所有语句组成。 □

例9.3 考虑图9-7中的源代码片段，它计算两个长度为20的向量a和b的点积。在我们的目标机器上完成该计算的三地址语句序列如图9-8所示。

```
begin
    prod := 0;
    i := 1;
    do begin
        prod := prod + a[i] * b[i];
        i := i + 1
    end
    while i <= 20
end
```

529

图9-7 计算点积的程序

让我们用算法9.1来生成图9-8的三地

址码的基本块。由规则(i)，语句(1)是一个入口语句。由规则(ii)，语句(3)也是一个入口语句，因为最后一条语句可以跳转到它。由规则(iii)，跟在语句(12)之后的语句是一个入口语句（注意，图9-7仅是程序片断）。这样，语句(1)和(2)构成一个基本块，从语句(3)开始的余下的语句形成第二个基本块。 □

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a [ t1 ]        /* 计算a[i] */
(5)   t3 := 4 * i
(6)   t4 := b [ t3 ]        /* 计算b[i] */
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12)  if i <= 20 goto (3)
```

图9-8 计算点积的三地址码

9.4.2 基本块的变换

基本块计算一组表达式，这些表达式是在基本块出口活跃的名字的值。如果两个基本块计算一组同样的表达式，则称它们是等价的。

可以对基本块应用很多变换而不改变它计算的表达式集合，许多这样的变换对改进最终由某基本块生成的代码的质量很有用。下一章将阐述全局代码优化器怎样使用这样的变换来重新安排程序的计算次序，以缩减最终目标程序的运行时间或空间需求。有两类重要的局部等价变换可用于基本块，它们是保结构变换和代数变换。

9.4.3 保结构变换

基本块主要的保结构变换是:

1. 公共子表达式删除。
2. 无用代码删除。
3. 重新命名临时变量。
4. 交换两个独立的相邻语句的次序。

530

让我们稍微详细一些来考察这些变换。目前，我们先假定基本块没有数组、指针和过程调用。

1. *公共子表达式删除。*考虑基本块

```
a := b + c
b := a - d
c := b + c
d := a - d
```
(9-2)

第二条语句和第四条语句计算同样的表达式，即b+c-d，因此，可以将该基本块转换成如下的等价基本块：

```
a := b + c
b := a - d
c := b + c
d := b
```
(9-3)

虽然(9-2)和(9-3)的第一条语句和第三条语句右部的表达式相同，但由于第二条语句重新定义了 b，使得第一条语句和第三条语句中的 b 具有不同的值，所以它们计算的不是相同的表达式。

2. *无用代码删除。*基本块的语句 x := y + z 给 x 定值，但如果以后不再引用 x，则称 x 为无用变量。这样的语句可以删去，而不会改变基本块的值。

3. *重新命名临时变量。*假如有语句 t := b + c，其中 t 是临时变量。如果我们将该语句改

成 u := b + c，其中 u 是新的临时变量，并且把这个 t 的所有引用都改成u，那么基本块的值不改变。事实上，我们总可以把一个基本块变换成这样的等价基本块，其中每条定义临时变量的语句都定义一个新的临时变量。这样的基本块叫做*范式基本块*。

4. *语句的交换*。如果基本块有两条相邻的语句

```
t1 := b + c
t2 := x + y
```

当且仅当 x 和 y 都不是 t_1，而且 b 和 c 都不是 t_2 时，可以交换这两个语句而不影响基本块的值。注意，范式基本块允许所有可能的语句交换。 531

9.4.4 代数变换

有许多代数变换可用于把基本块计算的表达式集合变换成代数等价的集合，但有用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中删除而不改变它计算的表达式集合。语句

```
x := y ** 2
```

中的指数算符通常需要用函数调用来实现。使用代数变换，这个语句可以由快速、等价的语句

```
x := y * y
```

来代替。

在9.9节的窥孔优化和10.3节的基本块优化中会更详细地讨论代数变换。

9.4.5 流图

通过构造称为*流图*的有向图，我们可以把控制流信息加到组成程序的基本块集合中。流图的节点是基本块。有一个特殊的节点称为*首节点*，这个基本块的入口语句是程序的第一个语句。如果在某个执行序列中 B_2 跟随在 B_1 之后，则从 B_1 到 B_2 有一条有向边，即如果

1. 从B_1的最后一条语句有条件或无条件转移到 B_2 的第一个语句；或者
2. 按程序的次序，B_2 紧跟在 B_1 之后，并且 B_1 不是结束于无条件转移

则我们说 B_1 是 B_2 的*前驱*，而 B_2 是 B_1 的*后继*。

例9.4 图9-7中程序的流图如图9-9所示。B_1 是首节点，注意，最后一条语句中，转移到语句(3)的语句已由等价的转移到 B_2 块开始的语句所代替。 □

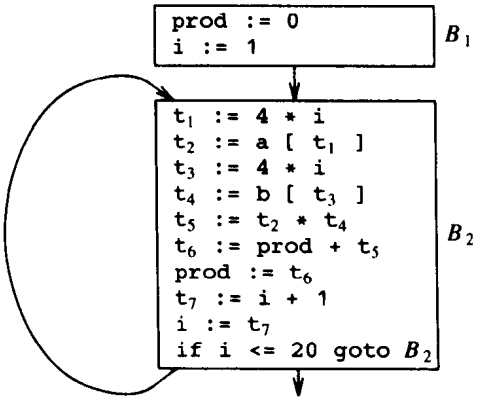


图9-9 程序的流图

9.4.6 基本块的表示

基本块可以用多种数据结构来表示。例如，用算法9.1将三地址语句划分完毕后，每个基本块可以用一个记录来表示，该记录包括块中四元式的个数、指向入口语句（第一个四元式）

的指针、前驱基本块表和后继基本块表。另一种办法是为每个基本块建立一个四元式链表。如果代码优化时要移动四元式的话，那么在块结尾的跳转语句中直接引用四元式的序号会引起麻烦。例如，图9-9中的中间代码中，如果从语句(3)到语句(12)的基本块 B_2在四元式数组中被移动到别的地方或者被缩短的话，那么 `if i <= 20 goto(3)`中的(3)就必须被改变。所以，我们宁可让跳转语句指向块而不是指向四元式，正如图9-9中所做的那样。

值得注意的是，流图中从块 B 到块 B' 的边没有指出在什么条件下控制从 B 转移到B'，即边没有告诉我们当条件满足或不满足的时候，B 末尾的条件转移语句是否转移到 B' 的入口语句。这些信息在需要时可以从 B 中的跳转语句恢复。

9.4.7 循环

在流图中，什么是循环？怎样找出所有的循环？大多数情况下，这些问题很容易回答。例如，图9-9中存在一个循环，它由块B_2组成。然而，对这些问题的全面回答却有点难以捉摸，我们将在下一章详细讨论。目前，只要知道循环是流图中满足下列条件的一簇节点就行了：

532
↓
533

1. 簇中所有节点是强连通的，即从循环中任一节点到另一节点都有一条长度大于等于1的路径，路径上的所有节点都在这簇节点中。

2. 这种节点簇有惟一的入口。从循环外的节点到达循环中任一节点的惟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。

9.5 下次引用信息

在本节中，我们将收集基本块中名字的下次引用信息。如果存放在寄存器中的名字以后不再需要，则可以将该寄存器分配给其他名字。只有当某个名字在后面还会用到时才将其保存在内存中的思想可以应用到许多种语境中。在5.8节中我们用该思想为属性值分配空间，下一节的简单代码生成器将其用于寄存器分配。作为最后一种应用，我们将考虑临时名字的存储分配。

9.5.1 计算下次引用信息

三地址语句中名字的引用（use）定义如下：假定三地址语句 i 将一个值赋给 x，如果语句 j 中的 x 是一个操作数，并且控制沿着一条没有对 i 赋值的路径从 i 到达 j，那么，我们称 j 引用在 i 中计算出的 x 的值。

我们希望为每个三地址语句 `x := y` *op* `z` 确定 x、y 和 z 的下次引用信息是什么。目前，我们不考虑在包含该语句的基本块以外的引用情况，但若需要这种信息，可使用第10章中的活跃变量分析技术来确定是否有这样的引用。

确定下次引用信息的算法对每个基本块反向扫描。扫描三地址语句流可以很容易地找到基本块的末尾，如算法9.1那样。因为过程可能有副作用，为方便起见，我们假定每个过程调用都开始一个新的基本块。

找到基本块的结尾后，我们由后向前扫描直到开始，并为每个名字x在符号表中登记它在本块中是否有下次引用；如果没有，则登记它在本块出口处是否活跃。如果已经做完了第10章讨论的数据流分析，则我们会知道哪些名字在每个块的出口处是活跃的。如果还没有做活跃变量分析，则可以采用一种保守的做法，即假定所有的非临时变量在出口处都是活跃的。如果产生中间代码或最优代码的算法允许某些临时变量在基本块间交叉引用，则把它们也当作是活跃的。一种好的办法是标记这样的临时变量，免得将所有的临时变量都看作是活跃的。

假如反向扫描时我们到达了三地址语句 i：`x := y` *op* `z`，然后我们执行如下操作：

1. 从符号表中找到 x、y 和 z 的下次引用信息和活跃情况，并把它们附加到语句 i 上。[534] ⊖

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，将 y 和 z 置成活跃，而且将它们的下次引用信息置为 i。

注意，步骤2和步骤3的次序不能颠倒，因为 x 可能是 y 或 z。

如果三地址语句 i 形如 x := y 或 x := *op* y，步骤同上，但忽略 z。

9.5.2 临时名字的存储分配

虽然在优化编译器中，每当需要临时变量时都建立一个不同的名字可能是有用的（理由在第10章中），但也必须分配空间以保存这些临时变量的值。7.2节中活动记录临时变量域的大小随临时变量个数的增加而增长。

一般情况下，如果两个临时变量不同时活跃的话，可以把它们压缩在同一个单元中。因为几乎所有的临时变量都是在基本块中定义和引用的，因此可以用下次引用信息来紧缩临时变量。对于那些在基本块间交叉引用的临时变量，放在第10章的数据流分析中进行讨论。

临时变量存储单元的分配可以这样进行：依次检查临时变量域的单元，找到第一个不含活跃临时变量的单元，把它指派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域中增加一个单元。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9.6节的讨论。

例如，基本块(9-1)中有6个临时变量可压缩在2个单元中，它们是 t_1 和 t_2：

```
t1 := a * a
t2 := a * b
t2 := 2 * t2
t1 := t1 + t2
t2 := b * b
t1 := t1 + t2
```

9.6 一个简单的代码生成器

本节中的代码生成策略为三地址语句序列生成目标代码，它依次考虑每条语句，记住该语句当前是否有操作数在寄存器中，如果有的话，尽量利用这一点。为简单起见，我们假设三地址语句的每种算符都有对应的目标机器算符。还假定将计算结果尽可能长时间地保留在寄存器中，只有在下面两种情况下才将其存入内存：[535]

(a) 如果此寄存器要用于其他计算。

(b) 正好在过程调用、转移或标号语句之前。⊜

条件(b)暗示在基本块的结尾，必须将所有东西都保存起来。⊛必须这样做的原因是，离开一个基本块后，可能进入几个不同的基本块中的一个，或者进入一个还可以从其他块进入的基本块。在这两种情况下，没有额外的工作，就认为不管控制怎样到达某基本块，该块引用的数据总是处于相同的寄存器中是不妥的。因此，为避免可能的错误，这个简单的代码生成算法在穿越基本块或调用过程时，存储所有的东西。稍后我们将考虑穿越基本块边界时将数据保存在

⊖ 如果 x 不活跃，则这个语句可以删掉。9.8节将考虑这种变换。

⊜ 但是，为产生符号转储（这将使得内存单元和寄存器的值会根据源程序的名字而变得可用），如果一个程序错误突然引起一个突然中断和退出的话，那么将程序员定义的变量（不必是编译器生成的临时变量）在计算之后立即保存起来将更方便。

⊛ 注意，我们没有假设四元式被编译器划分为基本块，基本块在任何情况下都是一个有用的概念。

寄存器中的方法。

如果我们生成单条开销为1的指令 ADD Rj，Ri，而将结果 a 存在 Ri 中，那么我们就可以为三地址语句 a := b + c 产生合理的代码。该序列只有在以下情况下才是可能的，即寄存器 Ri 中存放了 b，Rj 中存放了 c，且 b 在此语句之后不再活跃，即 b 在此语句之后不再被引用。

如果 Ri 中存放了 b，但 c 在内存单元中（为方便起见，仍叫做 c），b 仍然不再活跃，那么可以产生如下代码序列：

```
ADD   c, Ri          开销 = 2
```

或

```
MOV   c, Rj          开销 = 3
ADD   Rj, Ri
```

如果 c 的值以后还要被引用的话，则第二个指令序列比较有吸引力，因为可以从寄存器 Rj 中取它的值。基于 b 和 c 当前在什么地方以及 b 的值以后是否还会被引用，还有很多情况需要考虑。还必须考虑 b 和 c 中的一个或两个都是常数的情况。如果+是可交换的话，要考虑的情况还会增加。所以，我们可以看出，代码生成要考察大量情况，采用哪种情况依赖于三地址语
536 句出现的上下文。

9.6.1 寄存器描述符和地址描述符

代码生成算法使用寄存器描述符和地址描述符来记录寄存器的内容和名字的地址。

1. 寄存器描述符记录每个寄存器的当前内容。每当需要一个新的寄存器时，就要察看此描述符。假定初始时寄存器描述符指示所有的寄存器均为空（如果寄存器分配穿越块边界，就不是这种情况了）。当对基本块进行代码生成时，在任一给定时刻，每个寄存器将保存零个或多个名字的值。

2. 地址描述符记录运行时该名字的当前值存放的一个或多个位置。该位置可能是一个寄存器、一个栈单元、一个内存地址或这些地址的某个集合，因为复制时值仍然保留在原来的地方。这些信息可以存放在符号表中，用来确定对名字的存取方式。

9.6.2 代码生成算法

代码生成算法将构成一个基本块的三地址语句序列作为输入，对每个形如x := y *op* z的三地址语句执行如下动作：

1. 调用函数 *getreg* 以确定存放 y *op* z 的计算结果的位置 L。L 通常是寄存器，也可能是内存单元，我们将在下面简要描述函数 *getreg*。

2. 查看 y 的地址描述符以确定 y'，y' 是 y 的值存放的当前位置（或当前位置之一）。如果 y的值当前既在内存单元中又在寄存器中，则首选 y' 为寄存器。如果 y 的值还不在L中，则产生指令 MOV y', L，把 y 的值复制到L中。

3. 产生指令 OP z', L，其中 z' 是 z 的当前位置之一。同(2)一样，如果 z 值既在寄存器中又在内存单元中，则首选 z' 为寄存器。更新 x 的地址描述符以记录 x 在 L 中。如果 L 是寄存器，则更新这个寄存器描述符以记录该寄存器存有 x 的值。

4. 如果 y 和（或）z 的当前值没有下次引用，在基本块的出口也不活跃，并且在寄存器中，则修改寄存器描述符以表示在执行了 x := y *op* z 之后，这些寄存器分别不再包含 y 和（或）z的值。

如果当前的三地址语句中有一元算符，处理步骤与上面类似，我们略去这些细节。一个重要的特例是三地址语句 x := y。如果 y 在寄存器中，只要改变寄存器描述符和地址描述符以记录x的值现在只能在保存 y 值的寄存器中找到即可。如果 y 没有下次引用且在基本块的出口不 537
是活跃的，则该寄存器不再保存 y 的值。

如果 y 的值仅在内存中，原则上记住 x 的值在 y 的内存单元中即可，但这样会使我们的算法变得复杂，因为以后若要改变 y 的值，必须先保存 x 的值。所以如果 y 在内存，则用 *getreg* 来找到一个存放 y 的寄存器，并将此寄存器作为 x 的位置。

另一种办法是产生指令 MOV y, x。如果 x 的值在基本块中没有下次引用，这样做会比较好一些。值得注意是，如果使用第10章中的各种优化算法，尤其是复制传播算法，大多数（如果不是所有的）复制指令可以删掉。

一旦处理完基本块的所有三地址语句，我们就用 MOV 指令存储那些在基本块的出口是活跃的并且还不在它的内存单元中的名字的值。为完成这一工作，我们用寄存器描述符来确定哪些名字仍保留在寄存器中，用地址描述符来确定这些名字中哪些还不在它们的内存单元中，用活跃变量信息来确定是否要存储这些名字。如果基本块间的数据流分析还没有计算出活跃变量信息，则必须假定所有用户定义的名字在基本块的末尾都是活跃的。

9.6.3 函数*getreg*

函数 *getreg* 返回位置 L，用来保存语句 x := y *op* z 的 x 值。要为L产生非常好的选择须在这个函数的实现上付出很大的努力，本节将讨论一种基于前面的下次引用信息的简单易行的办法。

1. 如果名字 y 在寄存器中，且该寄存器不含其他名字的值(注意，别忘了 x := y 这样的复制指令会使得寄存器同时保存两个或更多变量的值)，并且 y 不活跃，且在执行 x := y *op* z 以后没有下次引用，那么就返回 y 的这个寄存器作为 L，并更新 y 的地址描述符以表示 y 已不在 L中。

2. (1)失败时，如果有空寄存器的话，就返回一个这样的寄存器作为L。

3. (2)失败时，如果x在该基本块中有下次引用，或者 *op* 是一个需要使用寄存器的算符，如索引，则找一个已被占用的寄存器 R。如果 R 的值还没有出现在适当的存储单元 M 中，则用指令 MOV R, M 将 R 的值存入内存单元 M，更新M的地址描述符，并返回 R。如果 R 中保存了几个变量的值，则对于每个需要存储的变量都要生成一条MOV指令。一个合适的被占用寄存器可能是其数据在最远的将来被引用或者其值同时在内存中的寄存器。我们没有给出精确的选择，因为没有人证明哪种选择方法是最佳的。

4. 如果 x 在该基本块中不再被引用，或者找不到合适的被占用寄存器，则选择 x 的内存单元作为L。 538

更复杂的 *getreg* 函数在确定存放x值的寄存器时还要考虑 x 的后续引用情况和算符 *op* 的交换性。

例9.5 赋值语句 d := (a-b)+(a-c)+(a-c) 可以翻译成如下的三地址码序列：

```
t := a - b
u := a - c
v := t + u
d := v + u
```

其中 d 在基本块的出口是活跃的。上述代码生成算法为该三地址语句序列产生的代码序列如图

9-10所示。图中给出了在代码生成过程中寄存器描述符和地址描述符的值，但没有给出这样的事实：a、b 和 c一直在内存中。同时，我们还假定作为临时变量的 t、u 和 v 不在内存中，除非用 MOV 指令显式地将它们的值存到内存中。

语句	生成的代码	寄存器描述符	地址描述符
		寄存器空	
t := a - b	MOV a, R0 SUB b, R0	R0 包含 t	t 在 R0 中
u := a - c	MOV a, R1 SUB c, R1	R0 包含 t R1 包含 u	t 在 R0 中 u 在 R1 中
v := t + u	ADD R1, R0	R0 包含 v R1 包含 u	u 在 R1 中 v 在 RC 中
d := v + u	ADD R1, R0	R0 包含 d	d 在 RC 中
	MOV R0, d		d 在 RC 和内存中

图9-10 代码序列

函数 *getreg* 的第一次调用返回 R0 作为计算 t 的寄存器。因为 a 不在 R0 中，生成指令 MOV a, R0 和指令 SUB b, R0，然后更新寄存器描述符以记录 R0 包含 t。

代码生成以这种方式继续进行，直到最后一个三地址语句 d := v+u 处理完为止。注意，因为 u 没有下次引用，R1 将变为空。最后，在基本块的结尾生成指令 MOV R0, d，存储活跃变量 d。

539 图9-10中生成的代码的开销为12。可以将它缩减到11，只要在第一条指令后立即生成指令 MOV R0, R1，而删除指令 MOV a, R1 即可，但这需要更复杂的代码生成算法。开销能减少的原因是从 R0 装入 R1 比从内存装入要廉价一些。 □

9.6.4 为其他类型的语句生成代码

三地址语句中的索引与指针运算和二元运算的处理方式相同。图9-11给出了为索引赋值语句 a := b[i] 和 a[i] := b 生成的代码序列，假定 b 是静态分配的。

语句	i在寄存器Ri中		i在内存Mi中		i在栈中	
	代码	开销	代码	开销	代码	开销
a:=b[i]	MOV b(Ri),R	2	MOV Mi,R MOV b(R),R	4	MOV Si(A),R MOV b(R),R	4
a[i]:=b	MOV b,a(Ri)	3	MOV Mi,R MOV b,a(R)	5	MOV Si(A),R MOV b,a(R)	5

图9-11 索引赋值的代码序列

i的当前位置决定了生成的代码序列。图中给出了3种情况，分别是i在寄存器 Ri 中、i 在内存单元 Mi 中以及 i 在栈中，其偏移为 Si 且指向 i 所在活动记录的指针在寄存器 A 中。寄存器R是调用函数 *getreg* 之后返回的寄存器。对于第一个赋值，如果 a 在该基本块中有下次引用，并且寄存器R是可用的，则将 a 保留在寄存器 R 中。在第二个赋值语句中，假定 a 是静态分配的。

图9-12给出了为指针赋值语句 a := *p 和 *p := a 产生的代码序列。这里，p 的当前位置决

定了生成的代码序列。

语句	p在寄存器Rp中		p在内存Mp中		p在栈中	
	代码	开销	代码	开销	代码	开销
a:=*p	MOV *Rp,a	2	MOV Mp,R MOV *R,R	3	MOV Sp(A),R MOV *R,R	3
*p:=a	MOV a,*Rp	2	MOV Mp,R MOV a,*R	4	MOV a,R MOV R,*Sp(A)	4

图9-12 指针赋值的代码序列

同上面一样，这里也给出了3种情况：p 开始在 Rp 中，在内存单元 Mp 中，在栈中，其偏移地址为 Sp 且指向 p 所在活动记录的指针在寄存器 A 中。寄存器 R 也是调用函数 *getreg* 之后返回的寄存器，第二个赋值语句中也假定 a 是静态分配的。 540

9.6.5 条件语句

机器实现条件转移的方式有两种。一种方式是根据寄存器的值是否为下面6个条件之一进行分支：负、零、正、非负、非零和非正。在这样的机器上，像 `if x<y goto z` 这样的三地址语句可以通过下面的方法实现：把 x 减 y 的值存入寄存器 R，如果 R 的值为负，则跳到 z。

第二种方式是用一组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。这种方法适于大多数机器。通常，比较指令（在我们的机器上是CMP）有这样的性质：它设置条件码而不真正计算值。也就是说，若 x>y，那么 CMP x，y 置条件码为正，等等。条件转移机器指令根据指定的条件 <、=、>、≤、≠或≥是否满足来决定是否转移。我们用指令 CJ<= z 表示如果条件码为负或者零则转移到z。例如，`if x<y goto z` 可以用如下指令来实现：

```
CMP   x, y
CJ<   z
```

为带条件码的机器生成代码时，维护一个条件码描述符将非常有用，该描述符将告诉我们上次设置条件码的名字或被比较的名字对，于是下面的语句

```
x := y + z
if x < 0 goto z
```

可以实现为

```
MOV   y, R0
ADD   z, R0
MOV   R0, x
CJ<   z
```

前提是我们知道条件码是在指令 ADD z，R0 之后由 x 确定的。

9.7 寄存器分配与指派

只涉及寄存器操作数的指令要比那些涉及内存操作数的指令短且快。因此有效地利用寄存器对于产生好的代码非常重要。本节将给出判断程序中哪些值应该驻留在寄存器中（寄存器分配）以及每个值应驻留在哪个寄存器中（寄存器指派）的种种策略。 541

一种寄存器分配和指派的方法是给目标程序中的具体值分配某些寄存器。比如，可以给基地址分配一组寄存器，算术运算则分配另一组，运行时栈顶分配一个固定的寄存器等等。

这种方法的优点是它简化了编译器的设计，其缺点是应用范围限制太严，且寄存器的使用效率较低，某些寄存器可能在大部分代码中闲置未用，而且可能产生不必要的装载和存储。但是在多数计算环境中为基址寄存器、栈指针等保留几个寄存器而其余的寄存器由编译器灵活分配还是比较合理的。

9.7.1 全局寄存器分配

9.6节中的代码生成算法使用寄存器来保存单个基本块的值。但是在每个块的结尾要存放所有的活跃变量。为节省其中的某些存储和相应的负载，我们可以将寄存器分配给那些频繁使用的变量，并且在基本块之间（全局地）保持这些寄存器的一致性。因为程序将大多数时间花在内循环上，一种比较自然的全局寄存器分配方法是在整个循环中将经常使用的值保存在固定的寄存器中。现在假定，我们已经知道流图的循环结构，而且知道基本块中计算出的哪些值要在该块外使用。下一章将涉及计算这些信息的技术。

一种全局寄存器分配策略是分配固定数目的寄存器来保存每个内部循环中大部分活跃的值。不同循环中选中的值可能不同。没被分配掉的寄存器可以用来存放像9.6节中的一个块的局部值。这种方法有一个缺点：固定的寄存器数目对全局寄存器分配来说不总是够用。但是这种方法实现简单，而且已经被用在 Fortran H，即 IBM-360 系列机上的优化 Fortran 编译器中（Lowry and Medlock[1969]）。

在类C语言和Bliss语言中，通过使用寄存器声明在过程运行期间将某些值保存在寄存器中，程序员可以直接执行一些寄存器分配操作。寄存器声明的正确运用可以加速程序，但是程序员在没有弄清整个程序的轮廓时最好不要进行寄存器分配。

9.7.2 引用计数

要确定在执行循环 L 时通过将变元 x 保存在寄存器中所节省的空间，有一种简单的办法就
542 是要认识到，在我们的机器模型中，如果 x 在寄存器中，对 x 的每次引用都将节省一个单元的开销。如果我们使用上一节所用的为基本块产生代码的方法，那么在基本块中计算完x以后，如果在基本块中接下来还要引用 x，则 x 将被保留在寄存器中。这样，如果在循环 L 中存在对 x 的赋值，则对循环 L 中对 x 赋值之前的 x 的每一次引用也能节省一个存储单元。如果我们能在块结束时避免存储 x，我们还可以节省两个存储单元。这样，如果为 x 分配了一个寄存器，则对 L 的每个块，如果 x 在出口是活跃的，而且 x 在基本块中被赋予一个值，则计算两个单元的空间节省。

如果 x 在循环头部是活跃的，则我们必须在正好要进入循环 L 之前将 x 装载到寄存器中。这个装载需要耗费两个单元。类似地，如果 x 在循环 L 的入口是活跃的，则对 L 中的每个这样的块 B，其出口将转移到 L 外的它的某个后继，我们必须耗费两个单元来存放x。但是假如循环要迭代许多次，我们可以忽略这些开销。因此计算循环 L 中为 x 分配寄存器所带来的好处的近似公式为

$$\sum_{L\text{中的块}B} \left(use(x, B) + 2 * live(x, B) \right) \tag{9-4}$$

这里 $use(x, B)$ 是定义 x 之前，在 B 中对 x 的引用次数。如果 x 从 B 中退出时是活跃的，而且在 B 中被赋予一个值，则 $live(x, B)$ 为1，否则为0。注意，(9-4)是个近似公式，这是由于并非所有的块都是以相同的频率计算的，而且(9-4)基于这样一个假设：循环会迭代许多次。在其他机器上需要开发一个类似于(9-4)但可能大不相同的公式。

例9.6 考虑图9-13所述的内循环中的基本块，其中的转移和条件转移语句已经被忽略了。

假设我们用寄存器 R0、R1 和 R2 来保存循环中的值。为方便起见，在每个块的入口和出口活跃的变量如图9-13所示，它们分别紧跟在每个块的上面和下面。第10章我们会讨论到活跃变量的一些微妙的问题。例如，注意 e 和 f 在 B_1 的末尾都是活跃的，但是它们当中只有e在B_2的入口是活跃的，只有 f 在 B_3 的入口是活跃的。一般情况下，在块末尾活跃的变量是在其每个后继块的开始活跃的变量的并集。

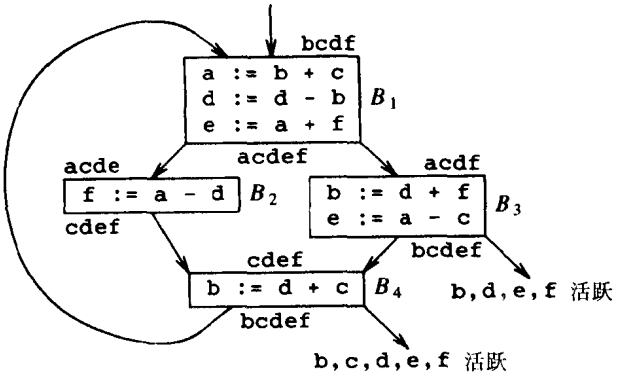


图9-13 一个内循环的流图

为计算 x = a 时式(9-4)的值，我们发现，a 在 B_1 的出口是活跃的，而且在那里被赋予一个值，但在 B_2、B_3 和 B_4 的出口不是活跃的。因此 $\sum_{B\text{中的}} 2live(a,B)=2$。$use(a,B_1)=0$，因为a是在任何引用之前于 B_1 中定义的。同样，$use(a,B_2)=use(a,B_3)=1$，而且，$use(a,B_4)=0$。所以，$\sum_{L\text{中的}B} use(a,B)=2$。因此，$x$ = a 时式 (9-4)的值为4，即通过将 a 存入一个全局寄存器可以节省4个单元的开销。当 x = b，c，d，e 和 f 时，式(9-4)的值分别为6、3、6、4和4。所以我们可以分别选择 a、b 和 d 放入寄存器 R0、R1 和 R2。使用 R0 存放 e 或 f 而不是 a 可以节省相同的开销。假设采用9.6节的技术为各个基本块生成代码，则从图9-13生成的汇编代码如图9-14所示。对于省略了的结束图9-13中每个基本块的条件和无条件转移语句，我们没有给出它们生成后的代码，因此我们也没有以一个单独的流给出生成后的代码。值得注意的是，如果我们不严格坚持保留寄存器 R0、R1 和 R2 的技术，则对 B_2 我们可以使用如下指令：

```
SUB  R2, R0
MOV  R0, f
```

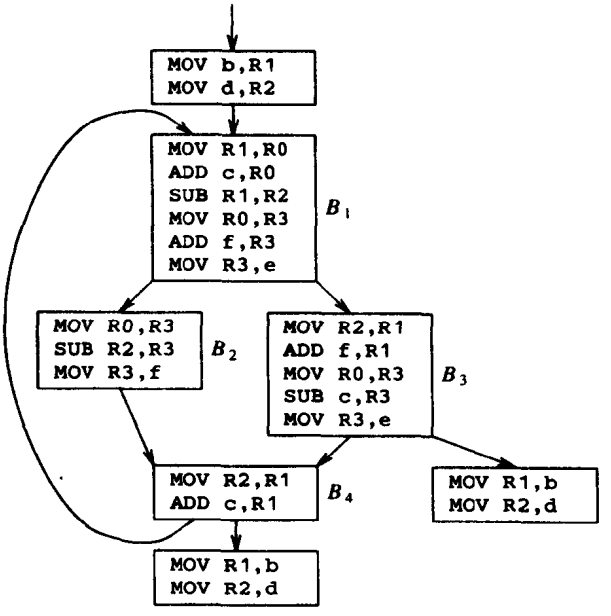


图9-14 使用全局寄存器指派的代码序列

因为 a 在 B_2 的出口不是活跃的，所以可以节省一个单元。类似的节省可以应用在 B_3 中。 □

9.7.3 外层循环的寄存器指派

为内循环指派了寄存器并产生代码以后，我们可以将同样的方法应用到外层循环中。如果外层循环 L_1 包含内循环 L_2，则在 L_2 中分配了寄存器的名字不必在 L_1-L_2 中再分配寄存器。但是，如果名字 x 是在循环 L_1 中而不是在 L_2 中分配了寄存器，我们必须在 L_2 的入口处保存 x 并在离开 L_2 进入块 L_1-L_2 之前装载 x。类似地，如果在 L_2 而不是 L_1 中为 x 分配寄存器，则我们

必须在 L_2 的入口装载 x，并在 L_2 的出口保存 x。假设已经对 L 中所有嵌套循环中的名字分配了寄存器，请读者给出为外层循环 L 中的名字分配寄存器的方法。

9.7.4 图染色法寄存器分配

如果计算时需要寄存器但所有可用的寄存器均被占用，则必须将某个正被使用的寄存器中的内容存放（溢出）到内存单元中以释放一个寄存器。图染色法是一种简单的用于寄存器分配和寄存器溢出管理的系统技术。

用这种方法，需要两遍处理。在第一遍中，选中目标机器指令，仿佛有无穷个符号寄存器。实际上，中间代码中使用的名字成了寄存器名，而三地址语句成了机器语言语句。如果访问变量需要使用栈指针、显示指针、基址寄存器或其他辅助访问的量的指令，那么我们假定这些量保存在寄存器中。一般情况下，它们的使用能够直接翻译到对机器指令所提地址的访问模式。
543 ~ 545 如果访问更加复杂，访问过程必须分解成若干机器指令，而且可能需要一个（或几个）临时的符号寄存器。

一旦选中指令，第二遍将把物理寄存器指派给符号寄存器。目标是找到溢出开销最小的指派。

在第二遍中，为每个过程创建一个寄存器冲突图。该图中节点是符号寄存器，如果一个节点在另一个节点的定义点是活跃的，则在这两个节点间有一条边。例如，图9-13的寄存器冲突图中含有名为 a 和 d 的节点。在定义 d 的块 B_1 中，a 在第二个语句是活跃的，因此，图中节点 a 和 d 之间有一条边。

我们试图用 k 种颜色给该寄存器冲突图图染色，k 是可指派寄存器的数目。（如果图的每个节点都被指派了一种颜色，且任何两个相邻节点的颜色不相同，则称该图为*染色图*。）一种颜色代表一个寄存器，染色可以保证没有两个互相冲突的符号寄存器被指派以相同的物理寄存器。

尽管确定一个 k 色图着色问题是 NP 完全的，但是下面的启发式技术还是可以在实际中快速实现着色。假设图 G 中的节点 n 具有少于 k 个邻居（通过边连接到 n 的节点）。从 G 中删除 n 和与它相连的边，得到图 G'。通过为 n 指派一种与其所有邻居不同的颜色，可以将对 G' 的 k 色着色扩展到对 G 的 k 色着色。

通过反复删掉少于 k 条边的节点，我们要么得到一个空图，要么得到一个每个节点有 k 个或多于 k 个相邻节点的图。对前一种情况来说，按节点被移走的相反顺序为其着色即可得到一个 k 色染色图；对后一种情况，k 色着色是不可能的。在此一个节点将通过引入存储和重装寄存器的代码而产生溢出。然后适当地修改一下该寄存器冲突图并继续进行着色。Chaitin[1982] 和 Chaitin et al. [1981] 中描述了几种选择溢出寄存器的启发式算法。原则上应避免将溢出代码引入内循环。

9.8 基本块的dag表示法

无环有向图（简称dag）是实现基本块变换的一种非常有用的数据结构。dag图示说明了基本块中每个语句计算的值是如何被本块中的后继语句引用的。对下列问题，从三地址语句构造dag是一种好办法：确定公共子表达式（多次计算的表达式）；确定哪些名字需要在块内使用，而在块外计算；确定块中哪些语句计算的值可以在块外使用。

*基本块的dag*是一种其节点带有如下标号的无环有向图：

546 1. 叶节点由惟一的标识符所标记，即变量名或常数。根据作用到名字上的操作符可以确定需要的是名字的左值还是右值；大多数叶节点代表右值。叶节点代表名字的初始值，为了避免

与指示名字当前值的标号相混淆，我们给这些叶节点加上下标0（如在下面的(3)中那样）。

2. 内部节点用操作符符号标记。

3. 节点还可能为标号附以一系列标识符，目的是用内部节点表示已经计算的值，而且标记节点的标识符也具有该值。

值得注意的是，不要把流图和 dag 弄混，流图的每个节点可以用一个 dag 表示，因为流图中的每个节点代表一个基本块。

```
(1)   t1 := 4 * i
(2)   t2 := a [ t1 ]
(3)   t3 := 4 * i
(4)   t4 := b [ t3 ]
(5)   t5 := t2 * t4
(6)   t6 := prod + t5
(7)   prod := t6
(8)   t7 := i + 1
(9)   i := t7
(10)  if i <= 20 goto (1)
```

图9-15 块B_2的三地址码

例9.7 图9-15给出了与图9-9中的块 B_2 相对应的三地址码。为方便起见，语句序号从(1)开始。与其对应的 dag 如图9-16所示，给出构造dag的算法后我们再讨论其意义。现在可以看到，dag 的每个节点都可代表一个由叶节点（即在基本块的入口变量所具有的值和常数）形成的公式。例如，图9-16中标有t_4的节点代表 b [4*i]，即从地址 b 偏移 4*i个字节后所得到的字的值，即 t_4 的值。 □

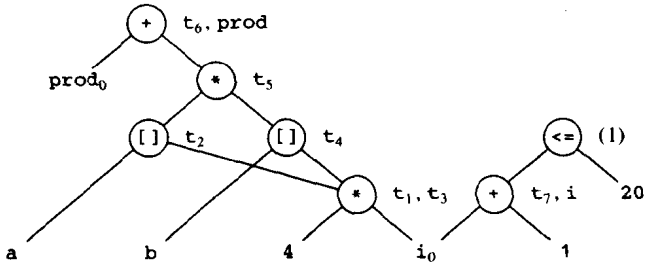


图9-16 图9-15的块的 dag

9.8.1 dag的构造

为了构造一个基本块的 dag，我们将依次处理块中的每条语句。当我们遇到形如x := y+z 的语句时，我们要寻找代表 y 和 z 当前值的节点。这些节点可能是叶节点，或者是dag的内部节点（如果 y 和 / 或 z 已在该块中先前的语句中计算过）。然后我们创建一个标以 + 的节点，并给它两个子节点；左子节点代表 y，右子节点代表 z。然后将标以 + 的节点再标以 x。但是，如果已经有一个节点表示同样的值 y + z，我们就不往 dag 中增加新的节点，而是给已经存在的节点附加标号 x。

另外还有两个细节应该提到。首先，如果 x（不是x_0）已经标在某个节点上了，我们要删除该标号，因为 x 的当前值是刚被创建的节点。其次，对于像 x := y 这样的赋值语句我们不创建新节点，而只是将标号 x 标在 y 的当前值所在节点的名字列表上。

现在我们给出计算基本块 dag 的算法。除了我们在此为每个节点附加上的标识符附加列表以外，该算法和算法5.1几乎一样。应该提醒读者注意的是，如果存在对数组的赋值，或者存在通过指针的间接赋值，或者由于 EQUIVALENCE 语句或实参与形参的对应而导致两个或更多的名字指向一个内存单元，该算法可能运行不正确。本章末尾我们将讨论处理这些情况所需要的修正。

算法9.2 构造一个 dag。

输入：一个基本块。

输出：包含下列信息的该基本块的dag。

1. 每个节点都有一个标号。叶节点的标号是一个标识符（允许是常数），内部节点的标号是一个操作符符号。

547 ~ 548

2. 每个节点有一个附加的标识符表（可能为空），这里的标识符不允许是常数。

方法：假定有合适的数据结构用于创建带有1个或2个子节点的节点，对于后者用左、右儿子区分其子节点。另外该数据结构中还有存放每个节点标号的位置以及创建每个节点附加标识符列表的机制。

除了这些内容以外，我们还需要维护一个与一个节点相关联的所有标识符（包括常数）的集合。这个节点或者是标以该标识符的叶节点，或者是该标识符出现在其附加标识符列表中的内部节点。我们假定存在一个函数 *node*(*identifier*)，当我们建立dag时，它将返回最新建立的与 *identifier* 相关联的节点。直观地看，*node*(*identifier*)是一个dag节点，该节点表示在dag创建过程的当前点标识符 *identifier* 所具有的值。实际上符号表中的 *identifier* 表项将指出*node*(*identifier*) 的值。

dag构造过程是依次对基本块中的每条语句执行下述步骤1至3。最初我们假设dag中没有节点，而且函数 *node* 对任何参数都是未定义的。假设当前三地址语句是如下三种形式之一：

(i) x := y *op* z。

(ii) x := *op* y。

(iii) x := y。 ⊖

分别将其称为(i)、(ii)和(iii)型。对于像`if i <= 20 goto`（x未定义）这样的关系操作符我们将其看成(i)型。

1. 如果 *node*(y)没有定义，则创建一个标号为 y 的叶节点，并令 *node*(y)为该节点。如果是(i)型，而且 *node*(z)没有定义，则创建一个标号为 z 的叶节点并令 *node*(z)为该叶节点。

2. 如果三地址语句为(i)型，确定是否有一个标号为 *op*，其左儿子为 *node*(y)，且右儿子为 *node*(z)的节点。（这种检查是为了发现公共子表达式。）如果没有，就创建一个这样的节点。无论是哪种情况，令 *n* 是所找到的或创建的节点。如果是(ii)型，确定是否有一个标号为 *op* 且只有一个子节点 *node*(y)的节点。如果没有，则创建一个这样的节点，并令 *n* 是这个找到的或创建的节点。如果是(iii)型，则令 *n* 为 *node*(y)。

3. 从 *node*(x)的附加标识符表中删除 x。在第2步找到的节点 *n* 的附加标识符表中增加标识符x，并且置 *node*(x)为*n*。 □

例9.8 让我们回到图9-15的基本块，看一看如何构造其dag（如图9-16所示）。第一条语句是 t_1 := 4*i。在第1步中，我们必须创建标以4和i_0的叶节点（同前，我们使用下标0区分标号和附加标识符，但下标不是标号的一部分）。在第2步中，我们创建一个标以*的节点，在第3步中，我们将标识符 t_1 加入到标号为*的节点的附加标识符表中。图9-17a给出了这一步的dag图。

对于第二条语句 t_2 := a[t_1]，我们创建一个标以 a 的新的叶节点并找到先前创建的节点 *node*(t_1)。我们还要

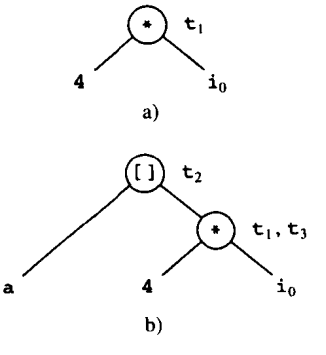


图9-17 dag创建过程中的一步

⊖ 假设操作符最多有两个操作数。推广到三个或多个操作数非常简单。

创建一个标以[]的新节点并将其左、右子节点设为 a 和 t_1。

对于语句(3)，t_3 := 4*i，我们可以确定 *node*(4)和 *node*(i)已经存在。因为操作符是*，所以我们不用为语句(3)创建新节点，而只是将 t_3 添加到以*为标号的节点的附加标识符表中。产生的dag如图9-17b所示。用5.2节的值-编号方法可以很快发现已经存在4*i的节点。

请读者自己完成dag图的构造。我们只提一下对语句(9)i := t_7 要采取的步骤。在处理语句(9)以前，*node*(i)是一个标以 i_0 的叶节点。语句(9)是一个(iii)型的实例，因此我们找到节点 *node*(t_7)，并将i附加到其标识符列表上，将 *node*(i)置为 *node*(t_7)。这是仅有的两条语句中的一个（另一个是语句(7)），其中 *node* 的值发生了变化，正是这种变化确保了 i 的新节点成为操作符<=节点的左儿子，该操作符节点是为语句(10)创建的。 □

9.8.2 dag的应用

当我们执行算法9.2时，可以获得一些非常有用的信息。首先请注意，我们可以自动检测出公共子表达式。其次，我们可以确定哪些标识符的值在该基本块中被引用过；它们就是那些某一时刻在步骤1中创建的叶节点所对应的标识符。第三，我们可以确定基本块中哪些语句计算的值可以在块外被引用；它们正好是这样的语句 *S*，即对于步骤2中为其创建或发现的节点 *n*，在dag构造结束时仍有 *node*(x) = *n*，其中x是语句 *S* 所指派的标识符。（等价地，x 仍然是 *n* 的附加标识符。）

549 ~ 550

例9.9 在例9.8中，所有的语句均满足上述约束条件，因为只有两次 *node* 被重新定义，即对 prod 和 i，而 *node* 的先前值是叶节点。因此，所有内部节点的值都可以在基本块外被引用。现在假设我们在语句(9)之前插入一个新语句*s*，它将一个值赋给 i。在语句 *s* 中我们将创建一个节点 *m* 并置 *node*(i) = *m*。但是在语句(9)中我们要重新定义 *node*(i)。于是，在语句 *s* 中计算的值不能在基本块外被引用。 □

dag 的另一个重要应用是利用公共子表达式并不允许使用形如 x := y 的赋值操作（除非绝对必要）来重建一个简化的四元式列表。也就是说，只要有一个节点的附加标识符列表含有一个以上的标识符，我们就检查该节点，看哪些标识符（如果有的话）需要在块外被引用。正如我们曾提到的，要找到块结尾处的活跃变量需要用到第10章讨论的称为“活跃变量分析”的数据流分析。然而，在许多情况下我们可以假定图9-15中基本块外不需要像 t_1，t_2，…，t_7 这样的临时名字。（但是需要弄清逻辑表达式是如何翻译的，一个表达式可能贯穿好几个基本块。）

我们一般可以用任意的拓扑分类顺序来计算dag的内部节点。按照dag的拓扑分类，直到所有的子节点（内部节点）都计算完毕才能计算它们的父节点。当我们计算一个节点时，我们将其值赋给它的一个附加标识符 x，当然要首先选择那些其值在块外需要被引用的标识符，但是，如果还有另一个节点 *m*，其值也由 x 保留，我们就不能选择x，以便计算完 *m* 以后 *m* 仍是活跃的。在这里我们定义 *m* 是活跃的，如果 *m* 的值需要在块外被引用，或者如果 *m* 有一个父节点还没被计算过。

如果节点 *n* 有另外的附加标识符 y_1，y_2，…，y_k，它们的值也要在块外被引用，则我们使用赋值语句 y_1 := x，y_2 := x，…，y_k := x 为其赋值。如果 *n* 根本没有附加标识符（这是可能的，如果 *n* 由对 x 的赋值语句所创建，但接下来x又被重新赋值），我们可以创建一个新的临时名字来保存 *n* 的值。读者应该知道，如果赋值语句中含有指针或数组，则并不是dag的每一种拓扑分类都是允许的，稍后我们将讨论该问题。

例9.10 按照图9-16的dag节点的建立顺序对节点进行排序：t_1，t_2，t_4，t_5，t_6，t_7，(1)，让我们从图9-16的dag出发重新构造一个基本块。请注意原始块中的语句(3)和(7)没有创建新节

点，而只是将标号 t_3 和 prod 分别添加到节点 t_1 和 t_6 的附加标识符列表中。我们假设任何临时变量 t_i 在块外都不需要。

我们从表示 4*i 的节点开始。该节点有两个附加标识符 t_1 和 t_3。我们用 t_1 保存 4*i 的
551 值，那么重新构造的第一条语句是

```
t1 := 4 * i
```

这和原始基本块中的语句是一样的。第二个考虑的节点标有 t_2，从该节点构造的语句是

```
t2 := a [ t1 ]
```

也和从前一样，下一个考虑的节点标有 t_4，从该节点构造的语句是

```
t4 := b [ t1 ]
```

后一个语句使用 t_1 而不是像在原始块中那样使用 t_3 作为参数，因为 t_1 是被选中用来保存 4*i 的值的名字。

接下来我们考虑的节点标有 t_5，并生成如下语句：

```
t5 := t2 * t4
```

对于标以 t_6，prod 的节点，我们选取 prod 保存该值，因为我们认为标识符 prod 而不是 t_6 在块外需要被引用。同 t_3 一样，t_6 不再出现。产生的下一条语句是：

```
prod := prod + t5
```

类似地，我们选择 i 而不是 t_7 来保存值 i+1，产生的最后两条语句是：

```
i := i + 1
if i <= 20 goto (1)
```

注意，通过利用dag创建过程中发现的公共子表达式，并删去不必要的赋值语句，图9-15中的10条语句已减少到7条。 □

9.8.3 数组、指针和过程调用

考虑如下基本块：

```
x := a[i]
a[j] := y                    (9-5)
z := a[i]
```

如果我们使用算法9.2构造(9-5)的dag，则a[i]将成为公共子表达式，而且“优化后”的基本块将为：

```
x := a[i]
z := x                       (9-6)
a[j] := y
```

但是，在 i=j 并且 y≠a[i]时，(9-5)和(9-6)式所计算出来的 z 值是不相同的。原因是当我们为数组 a 赋值时，我们可能正在改变表达式 a[i]的右值，尽管 a 和 i 并没有变。因此当我们
552 处理对数组 a 的赋值时有必要删除标号为[]、左参数是 a 加上或减去一个常数（也可能是0）的节点[⊖]。也就是说，我们认为对这些节点添加附加标识符是非法的，这样可以避免它们被错

⊖ 注意，[]的参数说明数组名可能是a本身，或者是一个 a-4 这样的表达式。在后一种情况下，节点a将是节点[]的子节点的子节点，而不是其子节点。

误地识别为公共子表达式。因此我们需要为每个节点设置一个标志位以标记它是否已被删除。此外，对于基本块中提到的每个数组 a，为方便我们可以保存一个节点表，表中含有所有当前未被删除但若有对 a 的一个元素的赋值就必须被删除的节点。

对赋值 *p := w 会产生同样的问题，其中 p 是一个指针。如果我们不知道 p 指向什么，则在上述意义下当前正在建立中的dag中的每个节点都必须被删掉。如果标有 a 的节点 *n* 被删掉了，而且接下来有一个对 a 的赋值，那么我们必须为 a 创建一个新的叶节点，然后使用该叶节点而不是 *n*。后面我们会考虑由于删除节点而导致的对计算顺序的限制。

在第10章，我们将讨论可以发现p只能指向某些标识符子集的方法。如果 p 只能指向 r 或 s，那么只有 *node*(r) 和 *node*(s) 必须被删掉。另外也可以相信我们能够发现在基本块(9-5)中 i=j 是不可能的，在这种情况中节点 a[i]不必因为 a[j] := y 而被删掉。但是后一种发现经常是不值得的。

在基本块中的过程调用将删除所有的节点，因为对被调用过程缺少了解，我们必须假定任何变量都可能发生变化而产生副作用。第10章将讨论我们怎样才可能确定某些标识符没有被过程调用所改变，因而这些标识符所对应的节点不必被删除。

如果我们打算把dag重新组装成基本块，而且不想按照创建dag节点的顺序，那么我们必须在dag中指明某些独立的节点必须以某种顺序进行计算。例如在(9-5)中，语句 z := a[i] 必须跟在 a[j] := y 之后，而 a[j] := y 又必须跟在 x := a[i]之后。让我们在dag中引进边 $n \rightarrow m$，它并不表示*m*是*n*的一个参数，而只是表明计算dag时 *n* 的计算必须跟在 *m* 的计算之后。下面是需要遵循的规则：

1. 对数组 a 的一个元素的任何赋值或计算都必须跟在其前对该数组的一个元素的赋值（如果有的话）之后。

2. 对数组 a 的一个元素的任何赋值都必须跟在其前对 a 的任何计算之后。

3. 对任何标识符的任何引用都必须跟在其前的过程调用或通过指针的间接赋值（如果有的话）之后。

4. 任何过程调用或通过指针的间接赋值都必须跟在其前对任何标识符的任何计算之后。 553

也就是说，当对代码重新排序时，不可以出现对数组 a 的交叉引用，而且任何语句都不得穿越过程调用或通过指针的间接赋值。

9.9 窥孔优化

逐条语句进行的代码生成策略经常产生含有大量冗余指令和次最优结构的目标代码。通过对这些目标程序进行“优化”转换可以提高这种目标代码的质量。“优化”这个术语有些误导，因为没有任何数学工具可以保证结果代码是最优的。但是，许多简单的转换可以显著地改善目标程序的运行时间和空间需求，所以知道哪种转换在实际中有用是非常重要的。

*窥孔优化*是一种简单有效的局部优化方法，它通过检查目标指令的短序列（称为*窥孔*），并尽可能用更小更短的指令序列代替这些指令以提高目标程序的性能。尽管我们将窥孔优化作为改善目标代码质量的技术，它也可以直接用在中间代码生成之后以提高中间代码的质量。

窥孔是放在目标程序上的一个移动的小窗口。孔中的代码不需要是连续的，尽管有些实现要求这样。窥孔优化的特征是每次改进可能又为进一步的改进带来机会。总而言之，为获得最大收益，有必要对目标代码重复进行多遍优化。本节中我们将给出下面一些程序转换的例子，它们是窥孔优化的典型特征。

- 冗余指令消除。
- 控制流优化。
- 代数化简。
- 机器语言的使用。

9.9.1 冗余加载与保存

如果我们遇到如下指令序列：

```
(1) MOV   R0, a
(2) MOV   a, R0
```
(9-7)

我们可以删去指令(2)，因为当执行(2)时，(1)可以保证 a 的值已经存入 R0。注意如果(2)有标
554 号[⊖]，我们就不能保证(1)总是刚好在(2)之前执行，那就不能删去(2)。当两条语句都在一个基本块中时这种转换是安全的。

然而，如果使用9.6节提出的算法，就不会产生像(9-7)这样的代码，如果使用了9.1节开始时提出的那种较简单的算法倒有可能产生这样的代码。

9.9.2 不可达代码

窥孔优化的另一个措施是删去不可达指令。紧跟在无条件转移指令后的无标号指令可以删除。可以重复执行这种操作以删去一系列指令。例如，为了调试，一个大程序中的某些程序段只有在变量 debug 为1的情况下才会被执行。在C中，源代码可能如下所示：

```
#define debug 0
...
if ( debug ) {
    显示调试信息
}
```

在中间表示中，if 语句可能被翻译为

```
    if debug = 1 goto L1
    goto L2
L1: 显示调试信息
L2:
```
(9-8)

一种明显的窥孔优化是删除转移之后的转移指令。这样，无论 debug 的值是多少都可以将(9-8)替换为

```
    if debug ≠ 1 goto L2
    显示调试信息
L2:
```
(9-9)

现在，既然 debug 在程序的开始被置成0[⊜]，常量传播会将(9-9)替换成

```
    if 0 ≠ 1 goto L2
    显示调试信息
L2:
```
(9-10)

由于(9-10)的第一条语句的参数值是常数 **true**，因此可以将它替换为 goto L2。这样所有打印

⊖ 生成汇编代码的一个好处是可以提供标号，这将简化这种窥孔优化。如果要生成机器代码，而且要求进行窥孔优化，那么我们可以使用一个位来标记具有标号的指令。

⊜ 要说明debug具有值0我们需要执行全局“到达定义”数据流分析，正如第10章所讨论的。

调试信息的语句显然都是不可达的，可以一次一条地被删除。 555

9.9.3 控制流优化

第8章的中间代码生成算法经常会产生无条件转移到无条件转移指令、无条件转移到条件转移指令或条件转移到无条件转移指令的转移语句。通过下面这种窥孔优化可以在中间代码或者目标代码中将这些不必要的转移语句删除。我们可以将转移序列

```
    goto L1
    ...
L1: goto L2
```

替换为

```
    goto L2
    ...
L1: goto L2
```

现在如果没有转移到 L1的转移语句[⊖]，而且在它之前有一个无条件转移语句，那么就可以删除语句 L1： goto L2。类似地，语句序列

```
    if a < b goto L1
    ...
L1: goto L2
```

可以替换为

```
    if a < b goto L2
    ...
L1: goto L2
```

最后，假设只有一个转移到L1的转移语句，而且在L1之前有一个无条件转移指令，那么序列

```
    goto L1
    ...
L1: if a < b goto L2                    (9-11)
L3:
```

可以被替换为

```
    if a < b goto L2
    goto L3
    ...                                 (9-12)
L3:
```

虽然(9-11)和(9-12)的指令数相同，但有时我们可以跳过(9-12)中的无条件转移指令，而在(9-11)中则不行。因此，(9-12)在执行时间上要优于(9-11)。 556

9.9.4 代数化简

利用窥孔优化可以进行无数种代数化简。然而，只有几个代数恒等式经常出现以至于值得考虑其实现。例如，像 x := x+0 或 x := x*1 这样的语句经常用直接的中间代码生成算法生成，通过窥孔优化可以很容易地删除它们。

9.9.5 强度削弱

强度削弱是指在目标机器上用时间开销小的等价操作代替时间开销大的操作。某些机器指

⊖ 如果使用这种窥孔优化，我们可以在符号表中该标号对应的表项中计算跳转到该标号的转移语句的数目；该代码的搜索是不需要的。

令比别的指令快，费时运算的某些特殊情况可以用它们来代替。例如，用 x*x 实现 x^2 要比调用一个指数例程快很多。用移位操作实现乘以2或除以2的定点运算要更快一些。浮点数除以常数可以（近似）实现为乘以常数，那样可能会更快一些。

9.9.6 机器语言的使用

目标机器可以使用一些硬件指令来有效地实现某些特定的操作。检测出那些可以使用这些机器指令的代码可以大大减少执行时间。例如，有一些机器具有自动加1和自动减1的寻址模式。这些模式的运用可以大大提高参数传递过程中入栈和出栈的代码质量。这些模式还可以用在形如 i := i+1 的语句的代码中。

9.10 从dag生成代码

本节我们将讨论如何利用dag表示来生成基本块的代码。利用dag而不是三地址语句或四元式的线性序列，我们可以更容易地看出如何重新组织最终的计算顺序。我们将重点讨论dag是
557 树的情况。在这种情况下可以产生优化的代码，我们可以证明其程序短或使用的临时变量数最少。中间代码是分析树时也可以使用该算法。

9.10.1 重排序

我们先简单地考虑一下什么样的计算顺序会影响目标代码的开销。考虑下面的基本块，其dag如图9-18所示（该dag正好是一棵树）。

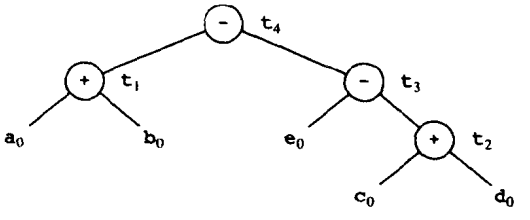


图9-18 基本块的dag

```
t1 := a + b
t2 := c + d
t3 := e - t2
t4 := t1 - t3
```

注意，使用8.3节的算法对表达式 (a+b)-(e-(c+d)) 进行语法制导翻译，我们可以自然地获得该顺序。

如果使用9.6节的算法为这些三地址语句生成代码，我们将得到图9-19中的代码序列。（假设有两个寄存器 R0 和 R1 是可用的，而且在基本块的出口只有 t_4 是活跃的）。

另一方面，假设我们重新组织语句序列如下，以便计算 t_1 的语句刚好在计算 t_4 的语句之前：

```
t2 := c + d
t3 := e - t2
t1 := a + b
t4 := t1 - t3
```

那么，使用9.6节的代码生成算法，我们将得到图9-20的代码序列（仍然假定只有 R0 和 R1 是可用的）。按这种顺序进行计算，我们能够节省两条指令：MOV R0, t_1（将 R0 中的值保存到内存单元 t_1）和 MOV t_1, R1（将 t_1 中的值重新加载到寄存器R1中）。

```
MOV a, R0
ADD b, R0
MOV c, R1
ADD d, R1
MOV R0, t1
MOV e, R0
SUB R1, R0
MOV t1, R1
SUB R0, R1
MOV R1, t4
```

图9-19 代码序列

```
MOV c, R0
ADD d, R0
MOV e, R1
SUB R0, R1
MOV a, R0
ADD b, R0
SUB R1, R0
MOV R0, t4
```

图9-20 修正后的代码序列

9.10.2 对dag的启发式排序

上述重排序改进代码的原因在于计算 t_4 的语句正好紧跟在计算 t_1（树中 t_4 的左操作数）的语句之后，这种排序的好处是显而易见

的。为了有效地计算 t_4，其左操作数应放在寄存器中，而计算 t_1 的语句正好在 t_4 之前就可以保证这一点。

在选择dag的节点顺序时，惟一的约束是必须保证该顺序保留了dag中的边的关系。回顾9.8节，这些边或者表示操作符-操作数关系，或者表示由于过程调用、数组赋值或指针赋值所带来的隐含约束。我们提出下面的启发式排序算法，该算法尽量推迟计算这样的节点，该节点紧跟在其最左参数的计算之后。图9-21的算法产生的是反向顺序。

例9.11 将图9-21的算法应用于图9-18中的树所产生的顺序可以用来生成图9-20的代码，作为一个更完整的例子，考虑图9-22的dag。 558 ~ 559

```
(1)  while 还有未列入表中的内部节点 do begin
(2)      选取一个未列入表的但其全部父节点均已列入表的节点 n;
(3)      将 n 列入表中;
(4)      while n 的最左子节点 m 不是叶节点并且其所有父节点均已列入表中 do
             /* 因为 n 刚被列入，m 还没被列入*/
             begin
(5)              将m列入表中;
(6)              n := m
             end
     end
```

图9-21 节点列表算法

最初没有未列入表的父节点的惟一节点是1，因此我们在第(2)行置 $n = 1$并在第(3)行将1列入表中。现在1的左参数2的父节点已经全部列入表中，因此我们将2列入表中并在第(6)行置 $n = 2$。现在，在第(4)行我们找到2的最左子节点6，它有一个未列入表的父节点5。所以我们在第(2)行选择一个新的 n，节点3是惟一的候选节点。我们将3列入表中并且沿着其左链将4、5和6列入表中。这样内部节点中只剩下节点8，我们将它也列入表中。结果表是1234568。所以建议的计算顺序是8654321。这个顺序对应下面的三地址语句序列：

```
t8 := d + e
t6 := a + b
t5 := t6 - c
t4 := t5 * t8
t3 := t4 - e
t2 := t6 + t4
t1 := t2 * t3
```

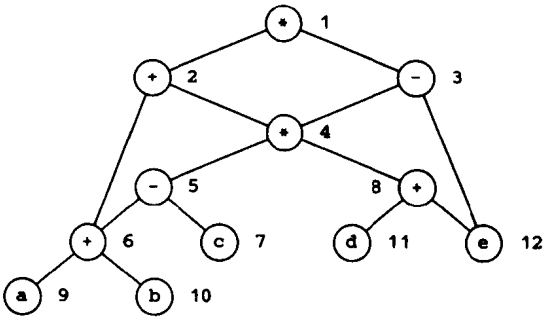


图9-22 一个dag

556

无论机器的寄存器个数是多少，如果我们使用9.6节的代码生成算法，将产生该 dag 的最优代码。要注意的是，在这个例子中，我们的排序启发在第(2)步没有任何选择可做，但一般情况下它可以有许多选择。 □

9.10.3 树的最优排序

在块的dag表示是一棵树的情况下，对于9.2节的机器模型，我们可以给出一个简单的、检测基本块中语句的最优计算顺序的算法。这里的最优顺序是指产生计算该树的最短指令序列的顺序。该算法已经被用在 Algol、Bliss 和 C 编译器中。

该算法包括两部分。第一部分用一个整数自底向上地标记树的每个节点，整数表示不用保

存中间结果要计算该树所需要的最少的寄存器个数，算法的第二部分按照计算出的节点标号的顺序遍历树。结果代码是在树的遍历过程中生成的。

直觉地讲，给定二元运算的两个操作数，通过首先计算需要更多寄存器的操作数，该算法可以发挥效用。如果两个操作数所需要的寄存器数目是相同的，则先计算哪一个都可以。

9.10.4 标记算法

我们用“左叶子”来表示一个叶节点，而且该叶节点是其父节点的最左后代。所有其他的叶节点都称为“右叶子”。

标记过程是按自底向上的顺序进行的，以便只有当一个节点的所有子节点都被标记过以后才能访问该节点。如果用分析树作中间代码，则分析树的节点生成顺序是合适的。在这种情况下，标号可用作语法制导翻译。图9-23给出了计算节点n的标号的算法。有一种重要的特殊情况，n是二元节点，且其子节点标号为l_1和l_2，则第(6)行的公式可以化简为

$$label(n)=\begin{cases}\max(l_1, l_2) & l_1 \neq l_2\\ l_1+1 & l_1=l_2\end{cases}$$

```
(1)  if n是叶节点 then
(2)      if n是其父节点的最左儿子 then
(3)          label(n) := 1
(4)      else label(n) := 0
     else begin /* n是内部节点 */
(5)      令n1, n2, …, nk是n的按 label 排序的子节点,
           因此 label(n1) ≥ label(n2) ≥ … ≥ label(nk);
(6)      label(n) := max_{1≤i≤k} (label(ni)+i-1)
     end
```

图9-23 标号计算

例9.12 考虑图9-18中的树。对节点的后序遍历[⊖]顺序为：a，b，t_1，e，c，d，t_2，t_3，t_4。后序遍历总是标记计算的一种合适的顺序。节点 a 被标记为1，因为它是一个左叶子。节点b 被标记为0，因为它是右叶子。节点t_1被标记为1，因为它的子节点的标号不相等，而且子节点的最大标号就是1。图9-24给出了结果标记树。它表明要计算 t_4 需要两个寄存器，而且事实上，要计算 t_3 就需要两个寄存器。 □

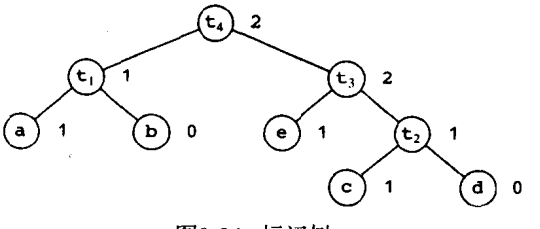


图9-24 标记树

9.10.5 从标记树中产生代码

现在我们给出一个算法，该算法的输入是一棵标记树 T，其输出为计算 T 并将结果存入R0中的机器代码序列。（然后可以将 R0 保存到内存中的合适位置。）我们假定树 T 中只有二元操作符。将操作符推广到具有任意数目的操作数并不难，我们将它留作一个练习。

该算法使用递归过程 *gencode*(n) 来产生计算以 n 为根的 T 的子树的机器代码，并将结果保存到寄存器中。过程 *gencode* 使用栈 *rstack* 来分配寄存器。初始时，*rstack* 中包含所有可以使

⊖ 后序遍历递归地访问节点n的子节点n_1，n_2，…，n_k，然后访问节点n。该顺序是自底向上语法分析过程中分析树节点的建立顺序。

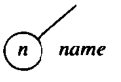
用的寄存器，假定按顺序依次为 R0，R1，…，R(*r*−1)。调用过程 *gencode* 可能会找到 *rstack* 中寄存器的一个子集，可能顺序会有所不同。当该过程返回时，它将按照找到它们的顺序将寄存器留在栈 *rstack* 中。结果代码计算树 *T* 的值，并将结果保存到 *rstack* 栈顶的寄存器中。 561 ~ 562

函数 *swap* (*rstack*) 用来交换 *rstack* 栈顶的两个寄存器。使用 *swap* 是为了保证左儿子和其双亲的计算结果放在相同的寄存器中。

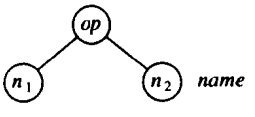
过程 *gencode* 使用栈 *tstack* 来分配临时内存单元。我们假定 *tstack* 最初包含 T0，T1，T2，…。实际上，如果我们只是记录 *i* 以便 T*i* 当前在栈顶，则 *tstack* 不必用列表来实现。*tstack* 的内容一直都是 T0，T1，…的后缀。

语句 *X* := *pop*(*stack*) 表示将栈顶元素弹出并将弹出的值赋给 *X*，相反我们用 *push*(*stack*, *X*)表示将 *X* 压入 *stack* 中。*top*(*stack*) 代表 *stack* 的栈顶值。

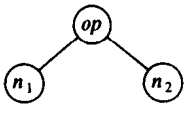
代码生成算法在*T*的根节点调用 *gencode*，过程 *gencode* 如图9-25所示。通过考查下面的五种情况可以解释该过程。第0种情况，我们有一棵形如下图的子树：



也就是说，*n* 是一个叶节点，并且是其父节点的最左儿子。因而我们只是生成一条装载指令。第1种情况，我们有一个形如下图的子树：



我们首先生成计算 n_1 的代码，并将结果存放到寄存器 *R* = *top*(*rstack*)中，然后生成指令 *op name*，*R*。第2种情况，子树形状如下：



在此，n_1 不用存储即可计算，但 n_2 要比 n_1 难算（即需要更多的寄存器）。对于这种情况，我们交换 *rstack* 栈顶的两个寄存器，然后计算n_2，并将结果存入 *R* = *top*(*rstack*)中。我们从栈 *rstack* 中移走 *R* 再计算 n_1，并将结果放入 *S* = *top* (*rstack*)中。注意，*S* 是第2种情况开始时 *rstack* 的栈顶寄存器。然后我们生成指令 *op R*，*S*，它将产生 *n* 的值并放在寄存器*S*中。再次调用过程 *swap* 将栈 *rstack* 恢复为本次 *gencode* 调用开始时的状态。 563

第3种情况与第2种情况类似，只是现在左子树难算，要先算左子树。而且也不需要交换寄存器。

第4种情况是指不存储时计算两棵子树都需要*r*个或更多个寄存器。因为我们必须使用临时内存单元，我们首先计算右子树，并将结果保存到临时内存单元 *T* 中，然后再计算左子树，最后是根。

例9.13 让我们为图9-24中的标记树生成代码，首先初始化 *rstack* = R0，R1。对过程 *gencode* 的调用序列以及打印代码的步骤如图9-26所示。图中括号中给出了每次调用 *gencode* 时栈 *rstack* 中的内容（栈顶在右端）。此处的代码序列是图9-20中代码序列的置换。 □

在不考虑操作符代数性质且假设没有公共子表达式的前提下，我们可以证明*gencode*可以

```
procedure gencode(n);
begin
/* 情况0 */
if n表示操作数name的左叶子，而且n是其父节点的最左儿子then
    print 'MOV' ‖ name ‖ ',' ‖ top(rstack)
else if n 是操作符为op，左儿子为n1，
        右儿子为n2的内部节点 then
/* 情况1 */
    if label(n2) = 0 then begin
        令name为n2所表示的操作数;
        gencode(n1);
        print op ‖ name ‖ ',' ‖ top(rstack)
    end
/* 情况2 */
    else if 1 ≤ label(n1) < label(n2) and label(n1) < r then begin
        swap(rstack);
        gencode(n2);
        R := pop(rstack); /* n2被计算进寄存器 R 中 */
        gencode(n1);
        print op ‖ R ‖ ',' ‖ top(rstack);
        push(rstack, R);
        swap(rstack)
    end
/* 情况3 */
    else if 1 ≤ label(n2) ≤ label(n1) and label(n2) < r then begin
        gencode(n1);
        R := pop(rstack); /* n1被计算进寄存器R中 */
        gencode(n2);
        print op ‖ top(rstack) ‖ ',' ‖ R;
        push(rstack, R)
    end
/* 情况 4，两个标号均≥r，r为寄存器的总数 */
    else begin
        gencode(n2);
        T := pop(tstack);
        print 'MOV' ‖ top(rstack) ‖ ',' ‖ T;
        gencode(n1);
        push(tstack, T);
        print op ‖ T ‖ ',' ‖ top(rstack)
    end
end
```

图9-25 函数*gencode*

```
gencode(t4)      [R1R0]                    /* 情况2 */
    gencode(t3)      [R0R1]                /* 情况3 */
        gencode(e)       [R0R1]            /* 情况0 */
            print MOV e,R1
        gencode(t2)      [R0]              /* 情况1 */
            gencode(c)       [R0]          /* 情况0 */
                print MOV c,R0
            print ADD d,R0
        print SUB R0,R1
    gencode(t1)      [R0]                  /* 情况1 */
        gencode(a)       [R0]              /* 情况0 */
            print MOV a,R0
        print ADD b,R0
    print SUB R1,R0
```

图9-26 *gencode* 程序跟踪

产生最优代码。要证明此结论，首先要证明所有代码序列都必须执行下述的操作，具体证明留给读者当作练习。

1. 对应每个内部节点的操作。
2. 每个为其父节点的最左儿子的叶节点上的装入。
3. 其子节点的标号均大于等于r的每个节点上的存储。

由于 *gencode* 正好产生这些步骤，因此它将产生最优代码。

9.10.6 多寄存器操作

我们可以修改标记算法以处理诸如乘、除或函数调用等通常需要多个寄存器的操作。只是简单地修改图9-23中标记算法的第6步，就可使 *label*(*n*) 返回操作所需寄存器的最小数目。例如，假设一个函数需要所有 r 个寄存器，就将(6)行换成 $label(n) = r$。如果乘法需要两个寄存器，在二元的情况下：

564 ~ 565

$$label(n) = \begin{cases} \max(2, l_1, l_2) & l_1 \neq l_2 \\ l_1 + 1 & l_1 = l_2 \end{cases}$$

其中 l_1 和 l_2 是 n 的子节点的标记。

但是，上述修改方法并不能保证对乘法、除法或多精操作而言有一个寄存器对是可用的。有一些机器使用一种有用的窍门，即假装乘法和除法需要三个寄存器。如果在 *gencode* 中总不使用 *swap* 的话，*rstack* 将总包含连续的高序号寄存器，对某个i而言，就是 i，$i+1$，…，$r-1$。因此，其中的头3个一定含有寄存器对。由于许多操作都是可交换的，所以我们经常可以避免使用 *gencode* 的第二种情况，即 *swap*。而且，即使 *rstack* 的栈顶中不包含三个连续的寄存器，我们也有很大的可能在 *rstack* 中的某处发现寄存器对。

9.10.7 代数性质

如果我们可以对不同操作符假定代数法则，那么我们就可能用带有更小标号（这样可以避免 *gencode* 中第4种情况的存储开销）以及/或者更少左叶子（可以避免第0种情况的装入开销）的树来代替给定的树 T。例如，因为+通常是可交换的，我们就可以用图9-27b中的树来代替图9-27a中的树，此举可以减少一个左叶子，还有可能降低某些标号。

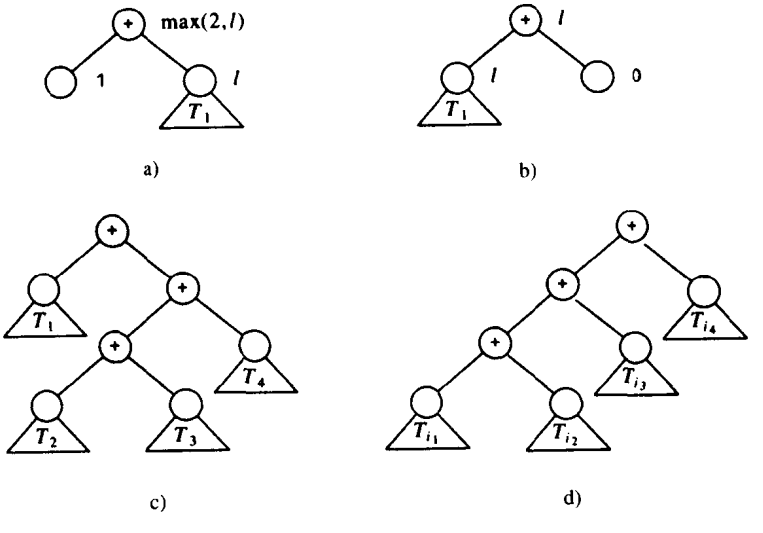


图9-27 可交换及可结合的变换

又因为+通常在满足交换律的同时也满足结合律，我们可以将图9-27c中的标号为+的那些节点用图9-27d中的左链来代替。为了使根的标号最小，我们只需使 T_{i_1} 为 T_1、T_2、T_3 和 T_4 中标号最大的一个，同时 T_{i_1} 不应是叶节点，除非 T_1、T_2、T_3 和 T_4 都是叶节点。

9.10.8 公共子表达式

当基本块中含有公共子表达式时，相应的dag将不再是树。公共子表达式对应的节点有多个双亲，称为*共享节点*。此时，我们不能再直接应用标记算法或 *gencode*。事实上，从数学的观点看，公共子表达式大大提高了代码生成的难度。Bruno and Sethi[1976]中证明了在单寄存器机器上为dag生成最优代码是NP完全的。Aho，Johnson，and Ullman[1977a]中证明了即使在具有无穷数目寄存器的机器上该问题仍然是NP完全的。难点在于如何用最小的开销来确定计
566 算dag的最优顺序。

实际上，如果我们通过为每个根和/或共享节点 n 找到以 n 为根的最大子树（该子树不包含其他共享节点，除非它是叶节点）将dag划分成树的集合的话，就能得到一个合理的解决方案。例如，图9-22中的dag就可以划分成图9-28中的树。每个带有 p 个双亲的共享节点作为叶节点至多出现在 p 棵树中。那些在同一棵树上有多个双亲的节点应该被划分成尽可能多的叶节点，这样就不存在含有多个双亲的叶节点。

一旦我们以上述方式将dag划分成了树，我们就可以给树的计算排序，而且可以用前面的任一算法为每棵树生成代码。树的顺序必须遵循以下原则：对树计值时，对应树叶的共享值必须是可用的。可以对那些共享的量进行计算，并将其存入内存（如有足够多的可用寄存器则可以存在寄存器中）。虽然这一过程在产生最优代码时并非是必需的，但执行这一过程的效果通常是令人满意的。

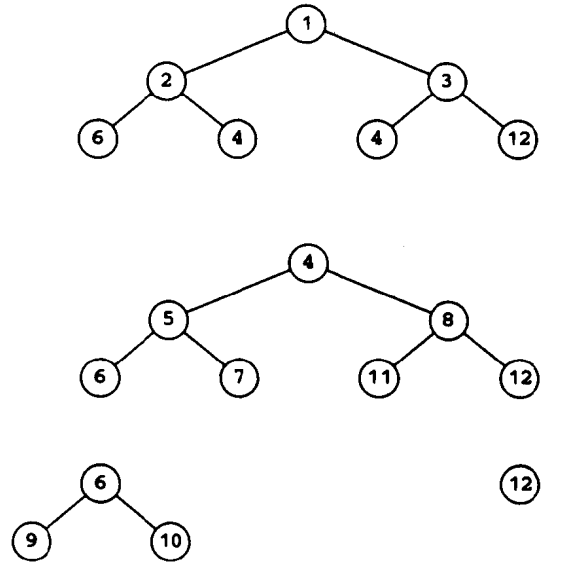


图9-28 从dag到树的划分

9.11 动态规划代码生成算法

在前一节中，*gencode* 可以从一棵表达式树生成最优代码，所用时间与树的大小成线性关系。*gencode* 适用于那些所有计算都在寄存器中完成并且指令都是由作用于两个寄存器或作用于寄存器与内存单元的操作构成的计算机。

使用基于动态规划原理的算法可以扩展可最优化机器的种类，使得我们可以从表达式树以线性时间为其生成最优代码。动态规划算法适用于广泛的带有复杂指令集的寄存器计算机。

9.11.1 一种寄存器计算机

动态规划算法可以适用于一切带有可互换寄存器（将这些寄存器记为 R0，R1，…，Rr-1）且指令格式为 Ri := E 的计算机，其中 E 指所有含操作符、寄存器和内存单元的表达式。如果 E中含有一个或多个寄存器，那么 Ri 必是其中之一。9.2节中提到的机器就是这种模式。

例如，ADD R0，R1 代表 R1 := R1+R0。指令ADD *R0，R1 代表 R1 := R1+ind R0

(ind代表间接寻址符)。

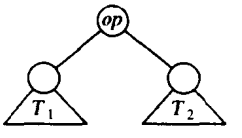
假设机器带有装入指令 R*i* := M、存储指令 M := R*i* 以及寄存器到寄存器的复制指令 R*i* := R*j*。虽然动态规划算法经过简单修改即可处理每条指令的开销各不相同的问题，但为简单起见，我们假设每一条指令的开销都是一样的。

567
~
568

9.11.2 动态规划的原理

动态规划算法把为表达式生成最优代码的问题分解成为该表达式的子表达式生成最优代码的子问题。举一个简单的例子，考虑形如 $E_1 + E_2$的表达式。E 的最优程序由 E_1 和 E_2 的最优程序（以一种或另一种顺序——E_1、E_2 或 E_2、E_1）和紧跟它们的操作符+的计算代码组成。可以类似地解决为 E_1 和 E_2 生成最优代码的子问题。

由动态规划算法生成的最优程序有一个重要的特点：对表达式 $E = E_1\ op\ E_2$ 是“邻近”计算的。我们可以通过观察 E 的语法树 T 而体会其中的含义：



在此，T_1 和 T_2 分别是 E_1 和 E_2 的语法树。

9.11.3 邻近计算

如果程序 P 先计算树T 的那些需要将计算结果存入内存的子树，我们就说程序 P 邻近计算树 T。此后 P 再以 T_1、T_2 然后根或者 T_2、T_1 然后根的顺序来计算 T 的剩余部分，不管以何种顺序，只要需要即可使用前一步中存入内存的结果。下面举一个非邻近计算的例子，P 可能先计算 T_1 并将值放到寄存器中（而非内存），接着计算 T_2，之后再转而计算 T_1 的剩余部分。

对上面定义的寄存器机器，我们可以证明，给定任意的用于计算表达式树 T 的机器语言程序 P，都可找到一个与之等价的程序 P'，P' 满足下列条件：

1. P' 的开销不比 P 的高。
2. P' 不比 P 使用更多的寄存器。
3. P' 以邻近的方式计算树。

这一结论暗示我们：每一个表达式树都可由一个邻近程序来进行最优化的计算。

作为对比，像IBM System/370这样的带有奇偶寄存器对的机器，不总是有最优的邻近计算。对这些机器，我们能够给出一些表达式树的例子，在这些例子中，最优的机器语言程序必须先计算根的左子树的一部分，并将结果存入寄存器，此后再计算右子树的一部分，接着再计算左子树的另一部分，然后是右子树的另一部分，依此类推。对于使用普通寄存器机器对任意表达式树的最优计算来说，这种震荡是不必要的。

569

上面定义的邻近计算的性质告诉我们，对于任何表达式树 T，总存在一个最优程序，该程序由计算根的子树的最优程序，跟以计算根的指令组成。这个性质允许我们用动态规划算法为树T生成一个最优程序。

9.11.4 动态规划算法

动态规划算法分三个阶段进行。假设目标机器有 r 个寄存器。在第一个阶段，自底向上为表达式树 T 的每个节点 n 确定一个开销数组 C，其中第 i 个元素 $C[i]$是计算以 n 为根的子树S（其值存入寄存器）的最佳开销，这里假设有 i 个寄存器可用于计算，$1 \leqslant i \leqslant r$。寄存器数量一定时，该开销包括任何计算 S 所必需的装载和存储，还包括计算 S 的根部操作符的开销。开销

向量的第0个元素是计算子树 S（其值存入内存）的最佳开销。通过只考虑合并计算 S 的子树的最优程序，该邻近计算性质可以保证为 S 生成一个最优程序。这个限制减少了必须考虑的情况的总数。

为了计算节点 n 处的 $C[i]$，考虑每条机器指令 R := E，其中表达式 E 与以节点 n 为根的子表达式相匹配。通过检查 n 的相应后代的开销向量来确定计算 E 的操作数的开销。对于 E 的那些寄存器操作数，考虑能够将 T 的相应子树计算到寄存器中的所有可能的顺序。在每一种顺序中，使用 i 个可用的寄存器即可计算与寄存器操作数相对应的第一棵子树，使用$i-1$个寄存器即可计算第二棵子树，依此类推。为了确定节点 n 的开销，把用于匹配节点 n 的指令 R := E 的开销加入到节点 n 的开销中。那么 $C[i]$ 的值是所有顺序中最小的开销。

整个树 T 的开销向量可以通过自底向上的方法来计算，而所用时间与 T 的节点总数成线性比例关系。对每个 i，在每个节点保存用于达到 $C[i]$ 的最少开销的指令是很方便的。在 T 的树根的开销向量中最小的开销就是计算 T 的最小开销。

在该算法的第二个阶段，遍历 T，用开销向量来确定 T 的哪个子树必须被计算到内存中。在第三个阶段，遍历每棵树，并用开销向量和相关的指令来产生最终的目标代码。要先产生计算到内存单元中的子树的代码。这两个阶段的运行时间也可以达到与表达式树的大小成线性比例关系。

570

例9.14 考虑带有两个寄存器 R0 和 R1 以及如下指令的机器，每条指令的开销为1：

```
Ri := Mj
Ri := Ri op Rj
Ri := Ri op Mj
Ri := Rj
Mi := Ri
```

这些指令中，Ri 为 R0 或 R1，Mj 是内存单元。

让我们用动态规划算法为图9-29的语法树生成最优代码。在第一个阶段，我们先计算图中每个节点所示的开销向量。为了说明开销的计算过程，考虑在叶节点 a 的开销向量。由于 a 已经在内存中，所以计算 a 到内存中的开销 $C[0]$ 是0。又因为我们可以用指令 R0 := a把a装载到寄存器，因此计算 a 到寄存器的开销 $C[1]$ 是1。当只有两个寄存器可用时，$C[2]$ 是把 a 装到寄存器的开销，这个开销与只有一个寄存器可用时相同。因此在叶节点 a 的开销向量为（0, 1, 1）。

考虑根节点的开销向量。我们首先确定只有一个和两个寄存器可用时计算根节点的最小开销。因为根节点标有+操作符，所以机器指令 R0 := R0+M 与根节点相匹配。使用该指令，在一个寄存器可用时计算根节点的最小开销等于将其右子树计算到内存中的最小开销与将其左子树计算到寄存器的最小开销之和，再加上该指令的开销1。不存在其他方法。根节点的右儿子和左儿子处的开销向量说明在一个寄存器可用时计算根节点的最小开销是5+2+1=8。

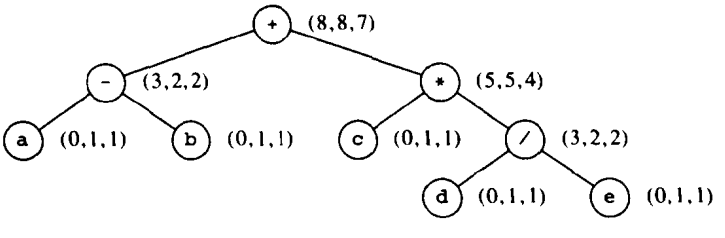


图9-29 (a-b)+c*(d/e)的带开销向量的语法树

现在考虑有两个寄存器可用时计算根节点的最小开销。根据计算根节点的指令和计算其左右子树的顺序，将导致三种情况出现。

1. 两个寄存器可用时将左子树计算到寄存器R0，一个寄存器可用时将右子树计算到R1，再使用指令 R0 := R0+R1 计算根节点。这个序列开销为 2 + 5 + 1 = 8。

2. 两个寄存器可用时将右子树计算到R1，一个寄存器可用时将左子树计算到寄存器R0，再使用指令 R0 := R0 + R1。该序列开销为 4 + 2 + 1 = 7。 571

3. 计算右子树，并存入内存单元M，两个寄存器可用时计算左子树，并存入寄存器 R0。再使用指令 R0 := R0 + M。该序列开销为 5 + 2 + 1 = 8。

第二种选择开销最小，为7。

将根计算到内存的最小开销等于所有寄存器可用时计算根的最小开销加上1。也就是说，将根计算到寄存器，再保存结果。因此，计算根的开销向量是(8, 8, 7)。

从上述开销向量，我们可以很容易地通过遍历树来构造代码序列。从图9-29中的树开始，假设有两个寄存器可用，最优代码序列为：

```
R0 := c
R1 := d
R1 := R1 / e
R0 := R0 * R1
R1 := a
R1 := R1 - b
R1 := R1 + R0
```

□

这种技术最初是在Aho and Johnson[1976]中提出的，该技术已被用到了大量的编译器上，其中包括S.C.Johnson的可移植 C编译器的第二版（PCC2）。这种技术还有利于重置目标的实现，这是因为动态规划技术可以应用于很多机型。

9.12 代码生成器的生成器

代码生成包括选择操作的计算顺序、分配寄存器以及选择合适的目标语言指令来实现中间表示中的操作符等。即使我们假定计算顺序已经给定，而且寄存器的分配是由单独的机制处理的，解决怎样选择指令的问题也是一项艰巨的任务，在带有多种寻址方式的机器上更是如此。在本节中，我们介绍了重写树技术，重写树技术可用于从目标机器的高级描述出发，自动构造代码生成器的指令选择阶段。

9.12.1 采用重写树技术的代码生成

本节中，代码生成过程的输入是一系列目标机器上的语义树。这些树可以通过将运行时地址插入中间表示来获得，正如9.3节所述。 572

例9.15 图9-30是赋值语句 a[i] := b+1的树，其中 a 和 i 是局部变量，它们的运行时地址由相对于 SP 的偏移量 $const_a$ 和 $const_i$ 给出，此时寄存器中包含的是指向当前活动记录开始位置的指针。数组 a 存在运行时栈中。对 a[i] 的赋值是间接赋值，a[i]的位置的右值被设为表达式 b + 1的右值。数组 a 的地

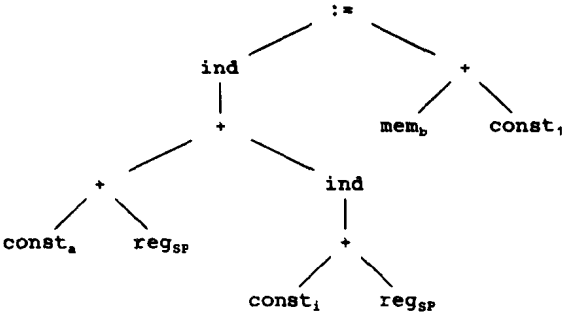


图9-30 a[i] := b+1的中间代码树

址等于常数 $const_a$ 的值加上寄存器 SP 中的值，i的值所在的位置等于常数 $const_i$ 的值加上寄存器 SP 中的值。变量 b 是内存单元 mem_b 处的全局变量。为简单起见，我们假设所有的变量都是字符类型的。

在该树中，ind 操作符将其参数视为内存地址。作为赋值操作符的左儿子，ind 节点给出了赋值操作符右部的右值被存入的内存单元。如果参数 a+或者 ind 操作符是一个内存单元或是一个寄存器，那么该内存单元或寄存器的内容会被当作该值。该树的叶节点是带有下标的类型属性，下标指出了属性的值。 □

通过将一系列重写树规则应用到这棵树上，可以将此输入树归约为单个节点，在此过程中即可生成目标代码。每个重写树规则是如下形式的语句：

replacement ← *template* {*action*}

其中，

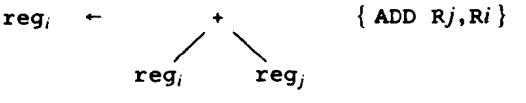
1. *replacement* 是单个节点。
2. *template* 是一棵树。

573 3. *action* 是一个代码片段，就像在语法制导翻译模式中那样。

重写树规则的集合叫做树翻译模式。

每个树模板代表由相关动作发出的一系列机器指令所执行的计算。通常，一个模板仅对应一条机器指令。模板的叶节点是带有下标的属性，就像输入树里那样。通常对模板中下标的值会施加某些限制，这些限制用语义谓词来描述，只有满足这些语义谓词才能进行模板匹配。例如，一个谓词可能规定一个常量只能在某一特定区间内取值。

树翻译模式可以很方便地表示代码生成器的指令选择阶段。作为重写树规则的例子，考虑寄存器到寄存器加法指令的规则：



可以像下面这样来使用该规则。如果输入树中包含一棵匹配该树模板的子树，也就是说，该子树的根标有操作符+，而且其左右儿子是寄存器*i*和寄存器*j*中的量，那么我们可以用一个标有 reg_i的单个节点来替换该子树，并输出指令ADD R_j, R_i。在给定时间可能有多个模板匹配一棵子树，我们应该简短地描述一些机制，以决定在冲突的情况下应用哪条规则。假设寄存器分配是在代码选择之前完成的。

例9.16 图9-31含有用于目标机器少数指令的重写树规则。这些规则将用在一个贯穿本节的运行示例上。头两条规则对应装载指令，接下来的两条规则对应存储指令，剩下的规则对应索引装载和加法指令。注意规则(8)要求常量的值为1。该条件将用语义谓词表示。 □

树翻译模式按下面的方式工作。给定一棵输入树，将重写树规则中的模板应用到它的子树上。如果一个模板匹配了一棵子树，则用规则中的替换节点替换输入树中的匹配子树，并执行与该规则相关联的动作。如果该动作包含一系列机器指令，则输出这些指令。重复该过程直到这棵树被归约为单个节点，或者再没有模板匹配为止。当输入树归约为单个节点时，生成的机器指令序列就构成了在给定输入树上的树翻译模式的输出。

详细说明代码生成器的过程与使用语法制导翻译模式说明翻译器的过程类似。我们编写了

574 一个树翻译模式来描述目标机器的指令集。实际上，我们更愿意寻找一种模式，对于每个输入

树，它能生成一个最小开销的指令序列。有几个工具可以帮助我们从树翻译模式自动建立代码生成器。

(1)	$reg_i \leftarrow const_c$	{ MOV #c, Ri }
(2)	$reg_i \leftarrow mem_a$	{ MOV a, Ri }
(3)	$mem \leftarrow :=(mem_a, reg_i)$	{ MOV Ri, a }
(4)	$mem \leftarrow :=(ind(reg_i), reg_j)$	{ MOV Rj, *Ri }
(5)	$reg_i \leftarrow ind(+(const_c, reg_j))$	{ MOV c(Rj), Ri }
(6)	$reg_i \leftarrow +(reg_i, ind(+(const_c, reg_j)))$	{ ADD c(Rj), Ri }
(7)	$reg_i \leftarrow +(reg_i, reg_j)$	{ ADD Rj, Ri }
(8)	$reg_i \leftarrow +(reg_i, const_1)$	{ INC Ri }

图9-31 用于某些目标机器指令的重写树规则

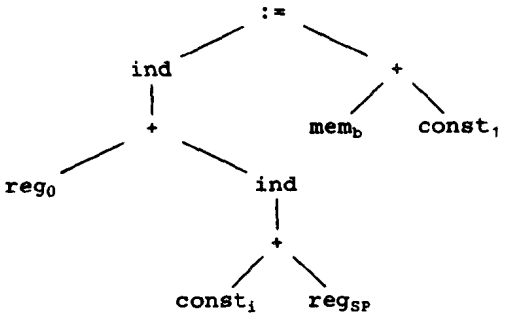
例9.17 让我们用图9-31的树翻译模式为图9-30的输入树生成代码。假设用第一个规则

(1) $reg_0 \leftarrow const_a$ { MOV #a,R0 }

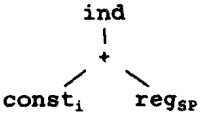
把常数 a 装入寄存器 R0。从而最左叶节点的标号从 $const_a$ 变为 reg0，并生成指令 MOV #a, R0。现在第7条规则

(7) $reg_0 \leftarrow +(reg_0, reg_{SP})$ { ADD SP,R0 }

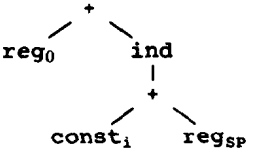
与以标号为+的节点为根的最左子树相匹配。使用这条规则，我们将该子树重写为标以 reg_0 的单个节点，并生成指令 ADD SP，R0。现在这棵树如下所示：



现在，我们应该使用规则(5)来把子树

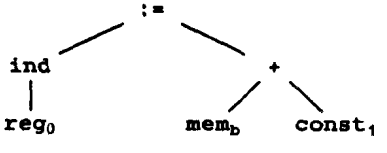


归约为标以 reg_1 的单个节点。然而，我们还可以使用规则(6)继续将稍大的子树

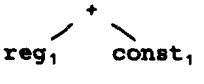


575
l
576

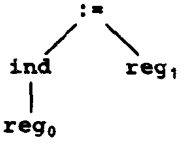
归约为标以 reg_0 的单个节点，并生成指令 ADD i(SP)，R0。假设使用单条指令计算大子树比计算小子树的效率高，我们选择后面的归约得到



在右子树中，将规则(2)应用于叶节点 mem_b。该规则生成一条将b装入寄存器1的指令。现在，使用规则(8)可以匹配下面的子树，并生成加1指令 INC R1。



现在，输入树已经被归约为



剩下的树被规则(4)匹配，它将这棵树归约为单个节点，并生成指令 MOV R1，*R0。

在该树归约为单个节点的过程中，我们生成了下面的代码序列：

```
MOV #a,R0
ADD SP,R0
```

```
ADD i(SP),R0
MOV b,R1
INC R1
MOV R1,*R0
```
□

上述树的归约过程还有几个方面需要进一步说明。我们没有详细说明树的模式匹配是怎样进行的。我们也没有详细说明模板匹配的顺序，或者当有多个模板能匹配时应该怎么办。另外，请注意，如果没有模板匹配，则该代码生成过程将被阻塞。在另一种极端的情况下，可能无休止地重写一个单个节点，从而生成一个寄存器移动指令的无穷序列或者装入和存储指令的无穷序列。

高效率地执行树模式匹配的一种方法是将练习3.32中的多关键字模式匹配算法扩展为自顶 577
向下的树模式匹配算法。每个模板可以用一个串集合来表示，即用根到叶子的路径集合来表示。从这些串的集合，我们可以构造一个像练习3.32那样的树模式匹配器。

使用树模式匹配并与前一节的动态规划算法相结合可以解决排序和多重匹配问题。通过将每个重写树规则与应用该规则所生成的机器指令序列的开销联系在一起，可以为树翻译模式增加开销信息。

实际上，在对输入树的深度优先遍历过程中运行树模式匹配器，并在节点最后一次被访问时执行归约即可实现重写树过程。如果我们并发地运行动态规划算法，使用与每个规则相关联的开销信息就能选择一个最优的匹配序列。我们可能需要推迟决定一个匹配直到所有可选规则的开销都已知道为止。使用这种方法，从树翻译模式可以快速地建立一个高效率的小型代码生成器。此外，动态规划算法使代码生成器的设计者不必去解决冲突匹配或确定计算顺序。

9.12.2 借助语法分析的模式匹配

另一种方法是使用LR语法分析器进行模式匹配。通过使用其前缀表示可以将输入树看作一个串。例如，图9-30中的树的前缀表示为：

$:= \ ind \ + \ + \ const_a \ reg_{SP} \ ind \ + \ const_i \ reg_{SP} \ + \ mem_b \ const_1$

通过把重写树规则替换为上下文无关文法的产生式，可以将树翻译模式转化成语法制导翻译模式，其中，产生式的右部是指令模板的前缀表示。

例9.18 图9-32中的语法制导翻译模式是基于图9-31的树翻译模式转换而来的。 □

使用第4章的一种LR语法分析器构造技术，我们可以从该翻译模式的产生式建立一个LR语法分析器。通过输出与每个产生式对应的机器指令即可生成目标代码。

(1)	$reg_i \rightarrow const_c$	{ MOV #c,Ri }
(2)	$reg_i \rightarrow mem_a$	{ MOV a,Ri }
(3)	$mem \rightarrow := mem_a \ reg_i$	{ MOV Ri,a }
(4)	$mem \rightarrow := ind \ reg_i \ reg_j$	{ MOV Rj,*Ri }
(5)	$reg_i \rightarrow ind + const_c \ reg_j$	{ MOV c(Rj),Ri }
(6)	$reg_i \rightarrow + reg_i \ ind + const_c \ reg_j$	{ ADD c(Rj),Ri }
(7)	$reg_i \rightarrow + reg_i \ reg_j$	{ ADD Rj,Ri }
(8)	$reg_i \rightarrow + reg_i \ const_1$	{ INC Ri }

图9-32 从图9-31构造出的语法制导翻译模式

代码生成文法通常具有很高的二义性，而且在建立语法分析器时，要更多地将注意力放到怎样解决语法分析动作的冲突上面。在缺乏开销信息的情况下，一般的规则是最大归约优先。这意味着在归约-归约冲突的情况下，采用长归约；而在移进-归约冲突时，则选择移进。这种

"最大贪吃"方法使得大量的操作用单机器指令执行。

代码生成时使用LR语法分析器有如下几个优点。首先，这种语法分析方法是高效率的、很好理解的，因此我们可以使用第4章描述的算法来产生可靠的和高效率的代码生成器。其次，可以相对容易地为代码生成器重置新的目标机器；通过写一个文法来描述新机器的指令，就可以为新机器构造一个代码选择器。第三，为了充分利用机器的习惯用法，我们可以增加特殊情况产生式，以使生成的代码效率更高。

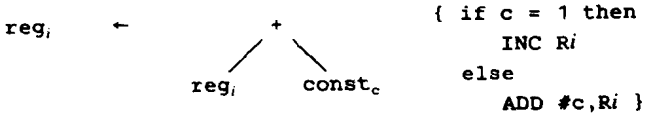
当然，使用这种方法也存在一些困难。语法分析方法决定了计算顺序只能是自左而右的顺序。而且，对于一些带有大量寻址方式的机器来说，机器的描述文法和产生的语法分析器可能变得非常大。因而，我们需要特别的技术来编码和处理机器描述文法。我们还必须小心谨慎以防产生的语法分析器在分析表达式树时阻塞（没有下一移动），原因是文法没有处理某些操作模式，或者语法分析器在语法分析动作冲突时做出了错误的决定。我们还要确保语法分析器不会因为使用单产生式（产生式右部只有一个符号）归约而陷入无限循环。生成语法分析表时使用状态分离技术可以解决这种循环问题（见Glanville[1977]）。

9.12.3 用于语义检查的例程

输入树的叶节点是带有下标的类型属性，其中，下标将一个值与属性关联起来。在代码生成翻译模式中，会出现相同的属性，但是通常对下标的取值做了限制。例如，一个机器指令可能要求属性在某个特定区间内取值或者要求两个属性的值有联系。

这些对属性值的限制可以用谓词来说明，并在归约之前调用这些谓词。实际上，与纯粹的代码生成器语法说明相比，语义动作和谓词的使用可以提供更大的灵活性并且易于描述。普通的模板用于表示指令类，而语义动作即可用于为特殊情况选择指令。例如，加法指令的两种形式能够用一个模板表示：

578
~
579



利用消除二义性谓词可以解决语法分析动作冲突，即在不同的上下文环境下允许使用不同的选择策略。用更小的描述描写目标机器是有可能的，因为机器体系结构的某些方面（例如寻址模式）能被表示为属性。这种方法的复杂之处在于，作为对目标机器的忠实描述，验证属性文法的准确性将变得非常困难，尽管这个问题在某种程度上存在于所有的代码生成器中。

练习

9.1 假定所有变量都是静态变量，对于9.2节的目标机器，试为下面的C语言语句生成代码。假定有三个寄存器是可用的。

a) `x = 1`
b) `x = y`
c) `x = x + 1`
d) `x = a + b * c`
e) `x = a/(b+c) - d*(e+f)`

9.2 假定所有的变量都是自动的（在栈中分配的），重做练习9.1。

9.3 假定所有变量都是静态变量，对于9.2节的目标机器，试为下面的C语言语句生成代码。假定有三个寄存器是可用的。

a) `x = a[i] + 1`
b) `a[i] = b[c[i]]`
c) `a[i][j] = b[i][k] * c[k][j]`
d) `a[i] = a[i] + b[j]`
e) `a[i] += b[j]`

9.4 使用下面的方法做练习9.1：

a) 使用9.6节的算法。

b) 使用过程 *gencode*。

c) 使用9.11节的动态规划算法。 580

9.5 试为下列C语言语句生成代码：

a) `x = f(a) + f(a) + f(a)`
b) `x = f(a)/g(b,c)`
c) `x = f(f(a))`
d) `x = ++f(a)`
e) `*p++ = *q++`

9.6 试为下面的C语言程序生成代码：

```
main()
{
    int i;
    int a[10];
    while (i <= 10)
        a[i] = 0;
}
```

9.7 假定对于图9-13中的循环我们分配三个寄存器来保存 a、b 和 c。试为该循环的基本块生成代码，并比较一下你的代码与图9-14中的代码的开销。

9.8 试为图9-13中的程序构造寄存器冲突图。

9.9 假定为简单起见，在每次程序调用之前我们自动将所有的寄存器保存到栈中（或者内存中，如果没有使用栈的话），并在返回后恢复它们。这对公式(9-4)（该公式用于计算将寄存器分配给循环中给定变量的效用）会产生怎样的影响？

9.10 修改9.6节的函数 *getreg* 使其在需要时返回寄存器对。

9.11 试给出一个dag的例子，使得使用图9-21中节点的启发式排序不会给出该dag的最佳顺序。

* 9.12 试为下面的赋值语句生成最优代码：

a) `x := a + b * c`
b) `x := (a * -b) + (c - (d + e))`
c) `x := (a/b - c)/d`
d) `x := a + (b + c/d*e)/(f*g - h*i)`
e) `a[i,j] := b[i,j] - c[a[k,l]] * d[i+j]`

9.13 试为下面的 Pascal 程序生成代码：

```
program forloop(input, output);
    var i, initial, final: integer;
    begin
        read(initial, final);
        for i:= initial to final do
            writeln(i)
    end.
```

581

9.14 试为下面的基本块构造 dag:

```
d := b * c
e := a + b
b := b * c
a := e - d
```

9.15 在下列条件下，对于练习9.14中的 dag，合法的计算顺序和节点的值的名字是什么?
a) 假定 a、b 和 c 在基本块的末尾是活跃的。
b) 假定只有 a 在基本块的末尾是活跃的。

9.16 在练习9.15(b)中，如果我们要为只有一个寄存器的机器生成代码，哪个计算顺序是最好的?为什么?

9.17 我们可以修改 dag 构造算法以考虑给数组赋值及通过指针的赋值。当一个数组的任何元素被赋值时，我们都假定为该数组生成了一个新值。这一新值由一个节点来表示，节点的子节点是数组的旧值、数组的下标值以及被赋的新值。通过指针的赋值时，我们假定我们已经为指针可能指向的每一个变量都生成了一个新值；对应每个新值的节点的子节点是指针的值以及变量的旧值。基于这些假设，试为下面的基本块构造 dag:

```
a[i] := b
*p := c
d := a[j]
e := *p
*p := a[i]
```

假定 (a)p 可以指向任何地方，(b)p 只指向 b 或 d。不要忘了给出隐含的顺序限制。

9.18 如果 a[i]或 *p这样的指针或数组表达式被赋值，然后被引用，而且其值在赋值和引用之间不可能改变，我们可以识别并利用这一状况以简化 dag。例如，在练习9.17的代码中，由于 p 在第2条和第4条语句之间没有被赋值，因此语句 e := *p 可以用 e := c 来替代（尽管我们不知道 p 指向哪儿，但我们确信无论 p 指向什么变量其值都会与 c 值相同）。利用该推论修改 dag 构造算法。应用你的算法为练习9.17的代码构造 dag。

582

** 9.19 在例9.14的 n 寄存器机器上，设计一个算法为形如 a := b+c 的三地址语句序列生成最优代码。必须按给定的顺序执行各语句。你的算法的时间复杂性是多少?

参考文献注释

对代码生成的研究综述感兴趣的读者应参考Waite[1976a,b]、Aho and Sethi[1977]、Graham[1980,1984]、Ganapathi, Fischer, and Hennessy[1982]、Lunell[1983]和Henry[1984]。Wulf et al. [1975]讨论了Bliss的代码生成，Ammann[1977]中讨论了Pascal的代码生成，Auslander and Hopkins[1982]则讨论了PL.8的代码生成。

程序引用统计对编译器的设计非常有用。Knuth[1971b]对Fortran 程序进行了经验统计研究。Elshoff[1976]中提供了一些PL/I的引用统计，Shimasaki et al. [1980]和Carter[1982]则分析了Pascal程序。Lunde [1977]、Shustek[1978]和Ditzel and McLellan[1982]中讨论了几种编译器在不同的计算机指令集上的性能。

本章建议的几种启发式的代码生成技术已经用在了各种各样的编译器上。Freiburghouse [1974]中讨论了将引用计数作为为基本块生成好的代码的辅助手段。Belady [1966]中说明了

*getreg*使用的一种获得空闲寄存器的策略在交换页上下文中是最优的，它通过释放一个其值最长时间没有被使用的变量的寄存器来获得空闲寄存器。Marill[1962]中提到了我们所用的分配固定数目的寄存器来保存循环期间变量的策略，该策略还用在Lowry and Medlock[1969]对Fortran H的实现中。

Horwitz et al. [1966]中给出了一个在Fortran中优化索引寄存器的使用的算法。J. Cocke、Ershov[1971]和Schwartz[1973]中提出了一种图染色法寄存器分配技术。9.7节的图染色法来自于Chaitin et al. [1981]和Chaitin[1982]。Chow and Hennessy[1984]描述了一种用于寄存器分配的基于优先级的图染色算法。Kennedy[1972]、Harrison[1975]、Johnsson[1975]、Beatty[1974]和Leverett[1982]中分别讨论了一些其他的寄存器分配算法。

9.10节的标记算法是对河流命名算法的模仿：一个主河流和一个小支流汇合后继续使用主河流的名字；两个相等河流汇合后被赋以一个新名字。标记算法最初出现在Ershov[1958]中。Anderson[1964]、Nievergelt [1965]、Nakata[1967]、Redziejowski [1969]和Beatty[1972]中分别提出了使用这种方法的代码生成算法。Sethi and Ullman[1970]在一个算法中使用了标记方法，他们还能证明该算法在许多种情况下可以为表达式树生成最优代码。9.10节中的过程*gencode*就是 583
Stockhausen[1973]对Sethi和Ullman算法的修正。如果目标机器具有必须按栈使用的寄存器，则Bruno and Lassagne[1975]以及Coffman and Sethi[1983]中给出了一种表达式树的最优代码生成算法。

Aho and Johnson[1976]设计了9.11节描述的动态规划算法。该算法被用作S. C. Johnson的可移植C编译器PCC2的代码生成器的基础，还被Ripken[1977]用在IBM 370机器的编译器中。Knuth[1977]将该动态规划算法推广到了带有非对称寄存器的机器如IBM 7090和CDC 6600之上。在推广过程中，Knuth将代码生成看作是对上下文无关文法的语法分析问题。

Floyd[1961]中给出了一个处理算术表达式中公共子表达式的算法。将dag划分成树，并且在树上使用*gencode*等过程的思想来自Waite[1976a]。Sethi[1975] 和 Bruno and Sethi[1976]中说明了对dag的最优代码生成问题是NP完全的。Aho，Johnson, and Ullman[1977a]中说明了即使对单寄存器和无穷寄存器机器该问题仍是NP完全的。Aho, Hopcroft, and Ullman[1974] 以及Garey and Johnson[1979]中讨论了一个问题是NP完全问题的重要性。

Aho and Ullman[1972a]以及Downey and Sethi[1978]中已经研究了基本块的变换问题。McKeeman[1965]、Fraser[1979]、Davidson and Fraser[1980，1984a, b]、Lamb[1981]和Giegerich[1983]中讨论了窥孔优化问题。Tanenbaum，van Staveren, and Stevenson[1982]中提倡在中间代码中也使用窥孔优化。

Wasilew[1971]、Weingart[1973]、Johnson[1978]和Cattell[1980]中将代码生成视为重写树的过程。9.12节重写树的例子来自Henry[1984]。Aho and Ganapathi[1985]中提议将有效的树模式匹配与动态规划相结合。基于9.12节的树翻译模式，Tjiang[1986]中实现了一种称为Twig的代码生成语言。 Kron[1975]、Huet and Levy[1979]和Hoffman and O'Donnell[1982]中描述了树模式匹配的普通算法。

通过利用LR语法分析器进行指令选择的Graham-Glanville代码生成方法在Glanville[1977]、Glanville and Graham[1978]、Graham[1980, 1984]、Henry[1984]和Aigrain et al. [1984]中进行了讨论和评价。Ganapathi[1980]以及Ganapathi and Fischer[1982]中使用属性文法说明并实现代码生成器。

Fraser[1977]、Cattell[1980]和Leverett et al. [1980]中提出了其他一些代码生成器自动构造技术。Richards[1971, 1977]中还讨论了编译器的可移植性。Szymanksi[1978]以及Leverett and Szymanski[1980]中描述了用于链接跨度相关转移指令的技术。 对于练习9.19，Yannakakis [1985]中有一个多项式时间的算法。 584

第10章 代码优化

理想情况下，编译器产生的目标代码应和手写的一样好。但现实是该目标只能在有限的情况下才能达到，而且相当困难。不过，可以对简单编译算法产生的代码进行改进，使其运行得更快一些，占用更少的空间，或者两者兼顾。这种改进是通过程序变换达到的，这种变换称为优化。虽然术语“优化”有些不当，因为很难保证结果代码是最好的。应用代码改进变换的编译器叫做优化编译器。

本章强调的是与机器无关的代码优化，即改进目标代码而不考虑基于目标机器任何特性的程序变换。依赖于机器的优化，例如寄存器分配和特殊机器指令序列的利用，已在第9章中讨论过了。

如果我们能识别出程序中经常执行的部分，并使得这些部分尽可能地达到高效率，那么便能以最小的代价获得最大的收益。流行的说法是大部分程序将90%的执行时间消耗在10%的代码上。虽然实际比例可能有变化，但通常程序运行的大部分时间都消耗在一小部分程序上。根据有代表性的输入数据精确地勾画出程序运行时的轮廓，可以识别出程序中频繁执行的区域。不幸的是，编译器不能受益于这样的典型输入数据，所以它必须尽一切努力猜测出哪里是程序的热点。

实际上，程序的内层循环是重点要改进的地方。在强调像while和for语句这样的控制结构语言中，循环是显式的出现在程序的语法中的。通常，程序流程图中的循环由控制流分析过程识别。

本章提供了许多有用的优化变换和实现它们的技术。确定编译器使用什么变换的最好技术是收集关于源程序的统计信息，并在真正的源程序的典型例子上评价一组给定优化算法的受益。 585
第12章描述的变换已经证明在好几种语言的优化编译器中都非常有用。

本章讨论的重点之一是数据流分析，它是收集程序中变量使用方式信息的过程。在程序的不同点收集的这种信息可以用简单的集合方程式联系起来。我们将提出用数据流分析收集信息和在优化中有效地使用这些信息的一些算法。我们还要考虑像过程和指针这样的语言结构对优化所产生的影响。

本章的最后四节处理更高级的细节。包括一些与控制流相关的图论的思想，并将这些思想应用到数据流分析中。本章最后将讨论用于数据流分析的通用工具和用于调试优化代码的技术。总之，贯穿本章的重点是应用在语言上的优化技术。第12章中将介绍使用这些思想的一些编译器。

10.1 引言

要建立高效的目标语言程序，程序员所需要的不仅仅是一个优化编译器。本节中，我们将考察为建立高效的目标程序，程序员和编译器可做的选择。我们会提到程序员和编译器的编写者可能期望用于改进程序性能的改进代码的变换种类，还要考虑可用于变换的程序表示。

10.1.1 代码改进变换的准则

简而言之，最好的程序变换是代价最小、收益最大的变换。由优化编译器提供的变换应该具有下列几种性质。

首先，代码变换必须保持程序的含义。也就是说，对于给定的输入，“优化”不能改变程

序产生的输出，也不能引起在源程序的原版本中不会出现的错误，如除数为零等。我们会始终采取比较安全稳妥的方案，即宁可失去某些优化的机会也不冒改变程序行为的风险。

其次，按平均数计算，变换必须将程序的速度加快一个可测量的值。虽然代码的规模不像以往那么重要，但有时我们还是对减少目标代码所需空间感兴趣。当然，并不是每种变换都能成功地改进每一个程序，而且有时优化甚至可能稍稍降低某个程序的运行速度，但只要按平均
586 数来说能够有所改进即可。

第三，一种变换必须值得我们付出努力。编译器的编写者为实现一种改进代码的变换所花费的时间和精力，以及编译器编译源程序时的额外开销，如果不能从目标程序的运行中得到补偿，那么这种改进变换是没有意义的。某些局部变换或者9.9节讨论的窥孔优化都是简单有益的，任何编译器都应包含它们。

有些变换，只有在对源程序进行详尽的、往往是费时的分析后才能使用，因此很少把它们用于只运行几次的程序。例如，快速的、非优化的编译器可能对调试或者对运行几次就要扔掉的“学生作业”更有帮助。只有当程序的执行要占用相当可观的机器时间时，为改进代码质量而消耗在运行优化编译器上的时间才是值得的。

10.1.2 性能的提高

程序运行时间的显著改进，例如从几个小时减少到几秒钟，通常要在所有级别上都进行改进才能达到，如图10-1所示，即从源程序级一直到目标代码级都要进行改进。在每一级中，可提供的选择都落在两个极端内，即寻找更好的算法和用更少的指令实现给定的算法之间。

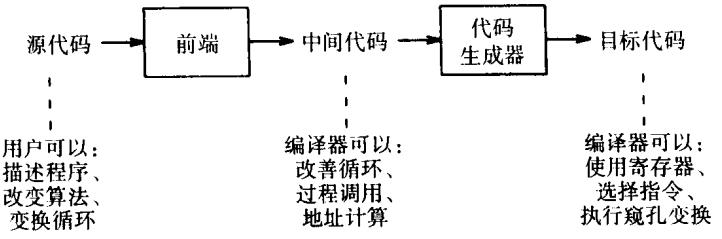


图10-1 用户和编译器可以改进代码的潜在位置

算法的变换对程序的运行时间有时会产生惊人的改进。例如，Bentley[1982]中叙述了当用快速分类代替插入分类时，对 N 个元素进行分类的程序的运行时间从 $2.02N^2\mu s$ 降到 $12N\log_2 N\,\mu s$。[⊖] 当 $N=100$ 时，快速分类使程序的运行速度提高2.5倍。当 $N=100\,000$ 时，这种改进更显著，速
587 度提高了1000多倍。

不幸的是，没有一个编译器能为给定程序找到最好的算法。不过，有时编译器能够用代数上等价的操作序列来替换一个操作序列，使得程序的运行时间有可观的缩短。当代数变换用于高级语言程序，例如数据库的查询语言（见Ullman[1982]）时，这样的节省更是常见。

本节和下一节中，我们将用一个快速分类程序 `quicksort` 来说明各种代码改进变换的作用。图10-2的 C 语言程序来自Sedgewick[1978]，该文讨论了对于这种程序的手工优化。在此我们不讨论该程序的算法。事实上，为了使这个程序能正常工作，`a[0]` 和 `a[max]` 应分别是被分类的最小元素和最大元素。

有些代码改进变换要想在源语言级完成是不大可能的。例如，在 Pascal 和 Fortran 这样的语言中，程序员只能按常规的方式引用数组元素，如 `b[i, j]`。然而，到了中间代码一级，代

⊖ 关于这些分类算法和其速度的讨论请参阅Aho, Hopcroft, and Ullman [1983]。

码改进的机会就会出现。例如，三地址码提供了许多改进地址计算的机会，尤其是在循环中。假定每个数组元素占4个字节，考虑决定a[i]值的三地址码:

```
t1 := 4*i;  t2 := a[t1]
```

这样的中间代码将对a[i]的每次出现重新计算4*i。因为它们是隐含在语言的中间代码里，而不是显式地出现在用户所写的代码中，因此程序员无法控制这种冗余的地址计算。在这种情况下，编译器应该删除它们。然而，在像C这样的语言中，这种变换可以由程序员在源程序级完成，因为访问数组元素可以使用指针系统地重写，以提高它们的效率。这种重写类似于Fortran的优化器传统上所应用的变换。

```
void quicksort(m,n)
int m,n;
{
     int i,j;
     int v,x;
     if ( n <= m ) return;
     /* 程序段由此开始 */
     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;
     }
     x = a[i]; a[i] = a[n]; a[n] = x;
     /* 程序段在此结束 */
     quicksort(m,j); quicksort(i+1,n);
}
```

图10-2 快速分类的C代码

在目标机器级，机器资源的合理有效使用则是编译器的责任。例如，把最频繁使用的变量保存在寄存器中，常常可使运行时间减少一半。还是C语言，它允许程序员要求编译器把某些变量保持在寄存器中，但是大多数语言却不允许。类似地，编译器可以利用机器的寻址方式，选择一条指令来完成通常需要两条或三条指令完成的动作，正如第9章所讨论的那样。

即使程序员有可能改进代码，但让编译器来完成某些改进可能更方便些。如果可以依赖编译器产生高效率的代码，那么用户可以将精力集中在书写清晰的程序上。

10.1.3 优化编译器的组织

正如我们已经提到的，我们常常可以在几个级别上改进程序。因为分析和变换程序所需要的技术不会随程序级别的不同而有较大变化，所以本章使用图10-3中所示的组织，主要集中在对中间代码的变换上。代码改进阶段由控制流分析、数据流分析和变换三部分组成。第9章中讨论的代码生成器从变换后的中间代码生成目标程序。

为方便表示，我们假定中间代码由三地址语句组成。用第8章的技术为图10-2的部分程序产生的中间代码如图10-4所示。若使用其他的中间表示，则图10-4中的临时变量t_1, t_2, …, t_{15}不必显式出现，如第8章所述。

图10-3中的组织有下列优点:

1. 实现高级结构所需的操作在中间代码中是显式的，这就有可能优化它们。例如，a[i]

588 ~ 589 的地址计算在图10-4中是显式进行的，这样，像表达式4*i这样的重复计算就可以被删除（如下一节将要讨论的那样）。

2. 中间代码可以（相对地）独立于目标机器，所以，由一种机器的代码生成器改为另一种机器的代码生成器时，优化器不必做太多修改。图10-4的中间代码认为数组的每个元素占4个字节，某些中间代码，如Pascal的P代码，把这件事情留给代码生成器，由它填入数组元素的大小，因此中间代码独立于机器字的大小。如用符号常量代替4，得到的中间代码也能完成同样的事情。

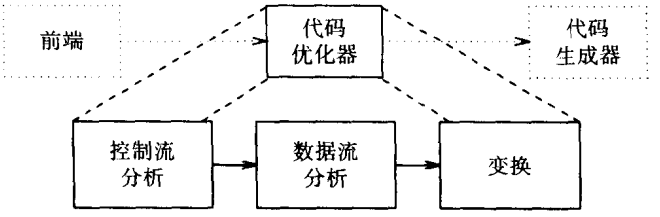


图10-3 代码优化器的组织

```
(1)  i := m-1                (16)   t7 := 4*i
(2)  j := n                  (17)   t8 := 4*j
(3)  t1 := 4*n               (18)   t9 := a[t8]
(4)  v := a[t1]              (19)   a[t7] := t9
(5)  i := i+1                (20)   t10 := 4*j
(6)  t2 := 4*i               (21)   a[t10] := x
(7)  t3 := a[t2]             (22)   goto (5)
(8)  if t3 < v goto (5)      (23)   t11 := 4*i
(9)  j := j-1                (24)   x := a[t11]
(10) t4 := 4*j               (25)   t12 := 4*i
(11) t5 := a[t4]             (26)   t13 := 4*n
(12) if t5 > v goto (9)      (27)   t14 := a[t13]
(13) if i >= j goto (23)     (28)   a[t12] := t14
(14) t6 := 4*i               (29)   t15 := 4*n
(15) x := a[t6]              (30)   a[t15] := x
```

590 图10-4 图10-2中程序段的三地址码

正如9.4节所讨论的那样，在代码优化器中，程序用流图表示，其中，边表示控制流，节点表示基本块。除非有其他说明，否则这里所说的程序都意味着单个过程。我们会在10.8节讨论过程间的优化。

例10.1 图10-5是图10-4中程序的流图。B_1是初始节点。图10-4中语句之间的所有条件转移和无条件转移在图10-5中都改成了基本块之间的转移，原来所转向的语句则是基本块的入口语句。

图10-5中有3个循环，B_2 和 B_3 都单独构成循环；以 B_2 为入口，块 B_2，B_3，B_4和B_5一起组成一个循环。 □

10.2 优化的主要种类

本节中，我们将介绍一些最有用的代码改进变换，实现这些变换的技术将在以下各节给出。只考查一个基本块中的语句就可以完成的变换叫做*局部变换*，否则叫做*全局变换*。许多变换既

可以在局部级也可以在全局级完成。通常首先执行局部变换。

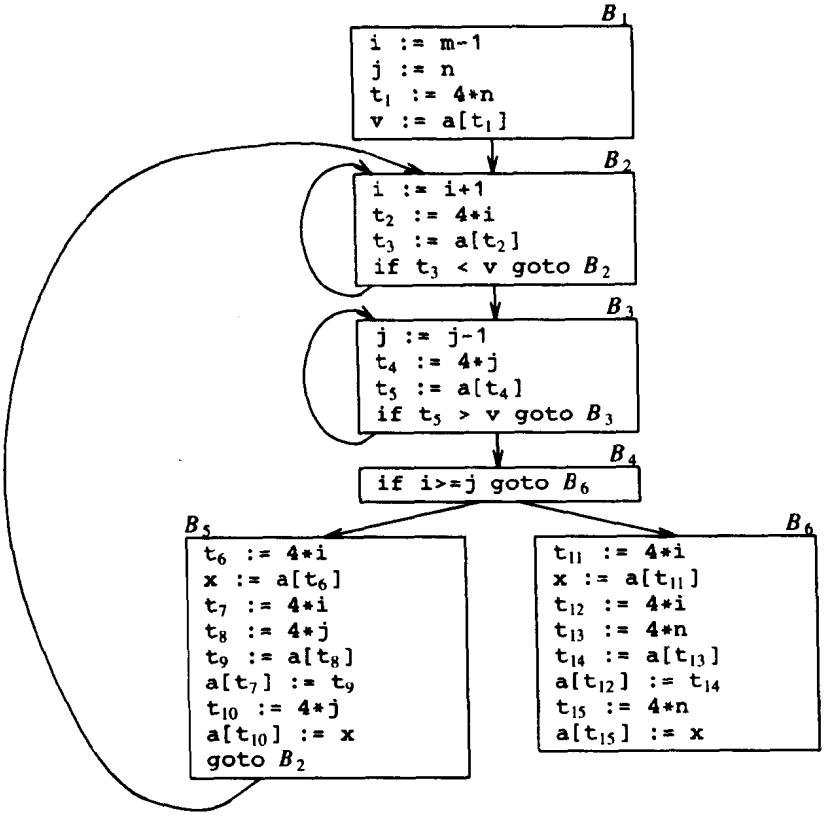
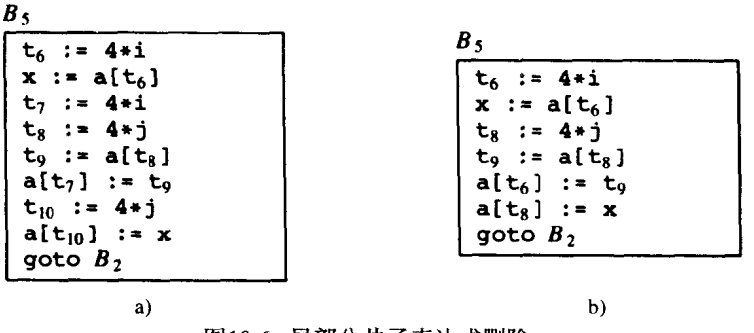


图10-5 流图

10.2.1 保持功能变换

有许多种方法可以使编译器改进一个程序而不改变其功能。例如，公共子表达式删除、复制传播、无用代码删除和常量合并等等。在9.8节中我们已经介绍了在构建基本块的dag表示时，是如何删除局部公共子表达式的。其他变换主要在进行全局优化时才执行，下面我们将逐一介绍。

一个程序经常包含对同一个值的多次计算，例如求一个数组地址的偏移量。正如在10.1节中提到的，程序员不能完全避免这些重复的计算，因为这些计算隐藏在源语言的细节层。比如，图10-6a中的块 B_5 重新计算了 `4*i`和`4*j`。



a) b)

图10-6 局部公共子表达式删除

a) 删除前 b) 删除后

10.2.2 公共子表达式

591 ~ 592 如果表达式E先前已被计算过，并且从先前的计算到现在，E 中变量的值没有改变，那么E的这次出现就称为公共子表达式。如果我们能够利用先前的计算结果，就可以避免表达式的重复计算。例如，在图10-6a中，对 t_7 和 t_{10} 的赋值语句分别有公共子表达式 4*i 和 4*j 在它们的右部。在图10-6b中，用 t_6 代替 t_7，用 t_8 代替 t_{10} 就已删除了这些公共子表达式。

例10.2 图10-7给出了从图10-5的流图的 B_5 和 B_6 块中删除全局和局部公共子表达式后的结果。我们首先讨论 B_5 的变换，然后解释涉及数组的一些微妙的地方。

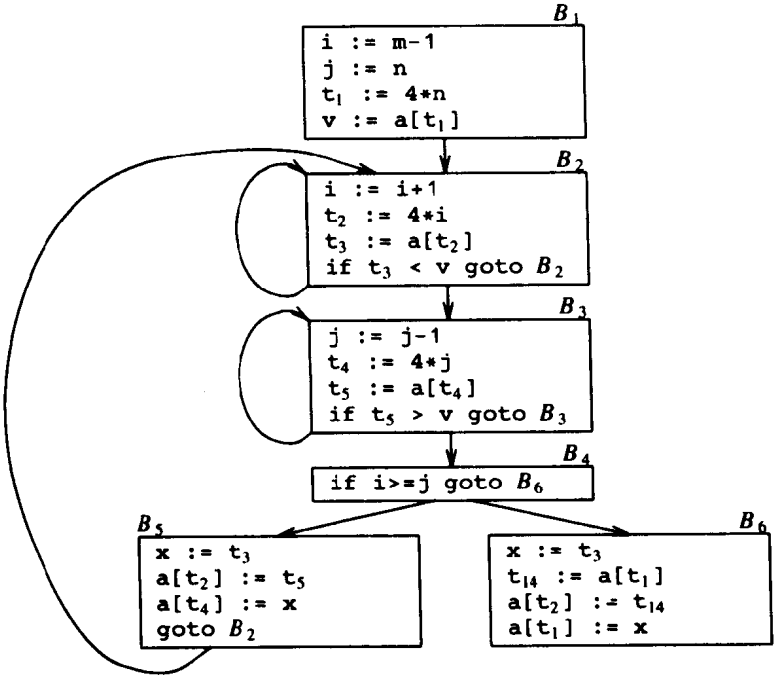


图10-7 删除公共子表达式后的B_5和B_6

如图10-6b所示，删除了局部公共子表达式后，B_5 仍然计算 4*i 和 4*j。它们都是公共子表达式。特别是，如果使用 B_3 中计算的 t_4，则 B_5 中的3个语句

```
t8 := 4*j;  t9 := a[t8];  a[t8] := x
```

可以用下列语句代替:

593
```
t9 := a[t4];  a[t4] := x
```

在图10-7中可以看到，当控制从 B_3 中对 4*j 的计算传到 B_5 时，中间没有改变 j，所以需要4*j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就变得清楚了。新的表达式 a[t4]对应于源代码中a[j] 的值。当控制离开 B_3 进入 B_5 时不仅j的值没有变，而且a[j]的值也没有变（a[j]的值计算后存在临时变量 t_5 中），因为在这期间没有对 a 的元素进行赋值，从而 B_5 中的语句

```
t9 := a[t4];  a[t6] := t9
```

可以由语句

```
a[t6] := t5
```

代替。

类似地，图10-6b的 B_5 中赋给 x 的值和 B_2 中赋给 t_3 的值是一样的。图10-7中的 B_5 是从图10-6b的 B_5 中删掉了与源代码表达式 a[i]和 a[j]的值相对应的公共子表达式后的结果。图10-7中的 B_6 也完成了一系列类似的变换。

图10-7中的 B_1 和 B_6 中的表达式 a[t_1]不能被看作为公共子表达式，虽然在这两个地方都使用了 t_1。因为控制离开 B_1 进入 B_6 之前，它可以通过 B_5，而 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，所以把 a[t_1]作为公共子表达式是不安全的。 □

10.2.3 复制传播

图10-7中的块 B_5 可以通过使用两种新的变换删除 x 而得到进一步的改进。一种和形如 f := g的赋值语句有关，这种赋值语句叫做*复制语句*，或简称为*复制*。如果我们在例10.2中进行了更深入的讨论的话，复制概念会更早一些提出，因为删除公共子表达式的算法已经引入了复制。当然，其他的一些算法中也引入复制。例如，当删除图10-8中的公共子表达式 c := d+e 时，该算法使用新的变量t来保存 d + e的值。因为控制要么在对 a 的赋值之后，要么在对b的赋值之后到达 c := d+e，所以用 c := a 或 c := b 来代替 c := d+e 是不正确的。

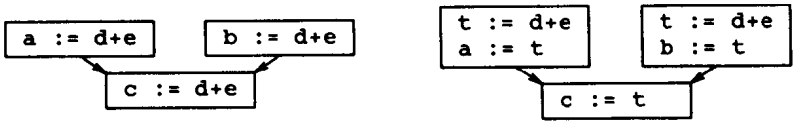


图10-8 删除公共子表达式期间引进的复制

复制传播变换的思想是，在复制语句 f := g 之后尽可能用 g 代替 f。例如，图10-7的块 B_5 中的赋值语句 x := t_3 是个复制。复制传播应用于 B_5 将产生如下代码:

```
x := t3
a[t2] := t5
a[t4] := t3                    (10-1)
goto B2
```

看起来这似乎没有改进，但我们将会看到，它增加了删除对x的赋值的机会。

10.2.4 无用代码删除

如果变量的值以后还要被引用，则称它在程序的该点是活跃的，否则称它在该点是无用的。其相关概念是无用代码，即它所计算的值决不会被引用的语句。虽然程序员不会故意引入无用代码，但前面的变换却可能引入无用代码。例如，在9.9节，我们讨论了debug的使用，在程序的不同地方可以将其置为真或假，而且将其用在类似于下面的语句中:

```
if (debug) print ...           (10-2)
```

从数据流分析有可能推断出：程序每次到达这个语句时debug的值总是假。通常这是因为有一个特定的语句:

```
debug := false
```

而且我们可以断定，不论程序实际上取什么分支序列，对debug的最后一次赋值总是先于测试语句式(10-2)。如果复制传播用假代替 debug，那么该打印语句是无用代码，因为它是不可到达的，于是可以从目标代码中删掉测试和打印。更一般地，在编译时推断出一个表达式的值是常量，并且用该常量代替它，这种变换叫做*常量合并*。

复制传播的优点之一是它常常将复制语句变成无用代码。例如，通过复制传播后再删除无用代码，删除了(10-1)中对 x 的赋值，并把它变成为:

```
a[t2] := t5
a[t4] := t3
goto B2
```

594 ~ 595

这段代码是图10-7中块 B_5 的进一步改进。

10.2.5 循环优化

现在我们简要介绍一个非常重要的可优化的地方，即循环，尤其是程序要消耗大部分时间的内循环。如果我们减少内循环的指令数，即使增加了该循环外的指令数，程序的运行时间也可减少。循环优化的3种重要技术是:

1. 代码外提。它把代码移出循环。
2. 归纳变量删除。用它我们可以从图10-7的内循环 B_2 和 B_3 中删掉 i 和 j。
3. 强度削弱。它用较快的操作代替较慢的操作，如用加代替乘。

10.2.6 代码外提

减少循环中代码总数的一个重要办法是代码外提。这种变换把循环不变计算，即所产生的结果独立于循环执行次数的表达式，放到循环的前面。注意，这里假定循环存在一个入口。例如，下面的 while 语句中，limit-2 是循环不变计算:

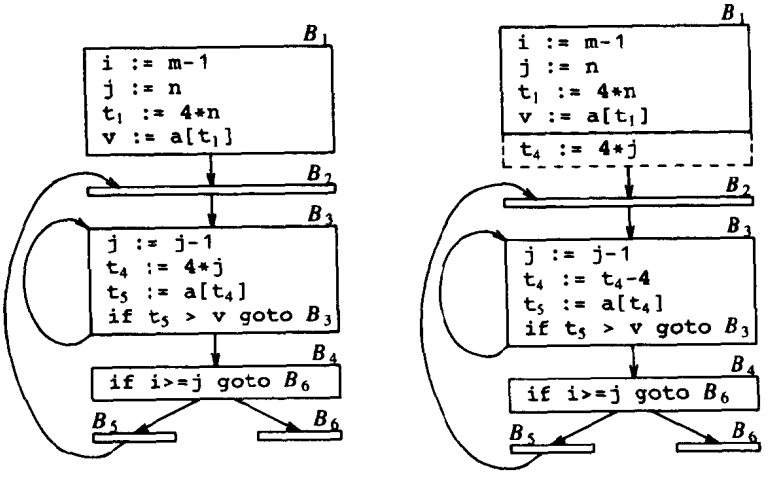
```
while ( i <= limit-2 ) /* 循环体中的语句不改变limit的值 */
```

代码外提将产生以下等价的语句:

```
t = limit-2;
while ( i <= t ) /* 循环体中的语句不改变limit或t */
```

10.2.7 归纳变量和强度削弱

虽然代码外提不能用于快速分类的例子，但另外的两种循环优化方法却可以。例如，考虑围绕块 B_3 的循环，图10-9只给出了和 B_3 的变换有关的部分流图。



a) b)

图10-9 将强度削弱应用到块 B_3 中的4*j

a) 变换前 b) 变换后

注意 j 和 t_4 的值总是同步变化，每次 j 的值减1，t_4 的值就减4，因为4*j赋给了 t_4，我们将这样的变量称为归纳变量。

如果在循环中有两个或更多的归纳变量，也许可以只保留一个，而去掉其余的，这将由归纳变量删除过程来完成。对于图10-9a中围绕块 B_3 的内循环，我们不能去掉j或t_4；因为t_4在B_3中被引用，而j在 B_4 中被引用。然而可以用它们来说明强度削弱和归纳变量删除的部分过程，最后，当考虑外循环 $B_2 \sim B_5$ 时，j将被删除。

例10.3 在图10-9a中，对内循环 B_3，若不考虑第一次进入，关系$t_4 = 4*j$在 B_3 的入口一定成立，在 j := j-1之后，关系 $t_4 = 4*j-4$也将成立。因此 t_4 := 4*j可以用 t_4 := t_4-4来代替。现在妨碍进行该变换的惟一问题是第一次进入 B_3 时 t_4 没有初值。因为我们必须在块 B_3 的入口维持关系 $t_4 = 4*j$，所以就在j本身被置初值的那一个块的末尾给 t_4 置一个恰当的初值 4*j。在图10-9b中，该语句放在块 B_1 的最后，并用虚线表示。

如果乘运算需要的时间远大于加或减运算的话（许多机器都是这样），那么这种以加减运算代替乘运算的变换就会加快目标代码的速度。 □

10.7节将讨论怎样寻找归纳变量以及可以施加什么变换。下面我们再举一个归纳变量删除的例子作为本节的结束，该例处理外循环 B_2, B_3, B_4 和 B_5 的上下文中的 i 和 j。

例10.4 把强度削弱应用于围绕 B_2 和 B_3 的内循环之后，i和j的作用仅在于决定 B_4 中测试语句的结果。我们已知道 i 和 t_2 满足关系 $t_2 = 4*i$，j和t_4也满足关系$t_4 = 4*j$，那么测试 $t_2 >= t_4$ 就等价于 i >=j。一旦作出这种替换，块 B_2 中的 i 和块 B_3 中的 j 就成了无用变量，在这些块中对它们的赋值也就成了可以删除的无用代码，结果流图如图10-10所示。 □ 596 ~ 597

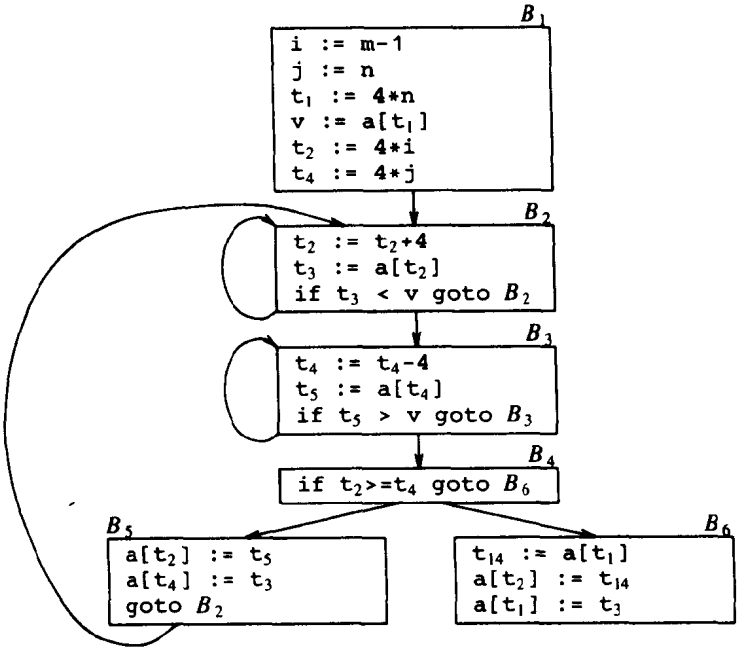


图10-10 归纳变量删除后的流图

这种代码改进变换是有效的。在图10-10中，块 B_2 和块 B_3 的指令数都从图10-5最初流图中的4条减为3条。B_5 从9条减到3条，B_6 从8条减成3条。虽然 B_1 从4条增加到6条，但是块 B_1 在

这段程序中仅执行一次，所以总的运行时间几乎不受块 B_i 大小的影响。

10.3 基本块的优化

在第9章中，我们已经看到许多对基本块的代码改进变换，包括诸如公共子表达式删除和无用代码删除的保结构变换，以及像强度削弱那样的代数变换。

许多保结构变换可以通过为基本块构建dag来实现。每一个出现在基本块中的变量其初始值在dag中都有一个节点，而且块中每一条语句 s 都有一个节点 n 与之相关联。n 的子节点是
598 那些在 s 之前、最后一次对 s 中用到的操作数进行定义的语句所对应的节点。节点 n 由 s 中用到的运算符来标记，而且将块中最后一次定义的变量列表附加到节点 n。如果有的话，我们还要记下那些其值在块的出口是活跃的节点，它们是输出节点。

当有一个新节点 m 将要加入时，如果存在节点 n，并且 n 和 m 有同样顺序、同样运算符以及同样的子节点，则可以知道有公共子表达式被检测到了。如果这样的话，n 和 m 计算相同的值，而且可以用在 m 的位置上。

例10.5 基本块(10-3)的dag如图10-11所示。

```
a := b + c
b := a - d
c := b + c
d := a - d
```
(10-3)

当我们为第3条语句 c := b + c构造节点时，我们知道在 b + c 中对 b 的引用指向图10-11中标以-的节点，因为那是对b最近的定义。这样，我们就不会混淆语句1和3所计算的值。

但是，与第4条语句 d := a-d 对应的节点具有运算符-，以及标号为 a 和 d_0 的子节点。因为运算符和子节点与语句2所对应节点的运算符和子节点相同，所以我们没有创建这个节点，而是将d添加到标号为-的节点的定义列表中。 □

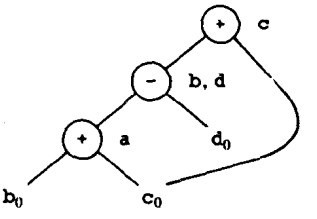


图10-11 基本块(10-3)的dag

由于图10-11中的dag中只有三个节点，基本块(10-3)可以被只有三条语句的块取代。事实上，如果 b（或d）在基本块的出口处不是活跃的，我们就不需要计算 b（或d），而可以利用d（或b）来接收由图10-11中标以-的节点所表示的值。譬如，如果 b 在出口处不是活跃的，我们就可以使用如下基本块来代替块(10-3)。

```
a := b + c
d := a - d
599 c := d + c
```

但是，如果b和d在出口处都是活跃的，则必须使用第4条语句来将一个节点的值复制到另一个。[⊖]

注意，当我们寻找公共子表达式时，我们其实是在寻找保证是计算相同值的表达式，而不

⊖ 一般地，当从dag重构代码时我们必须小心选择与节点对应的变量名。如果变量x被定义了两次，或者如果它被赋值一次而其初值x_0还被引用了，那么我们必须保证不改变x的值，除非我们已经使用了所有那些以前保存过x值的节点。

管这个值是怎样计算出来的。因此，dag方法将丢失这样的事实：在序列(10-4)中由第1条和第4条语句计算的表达式是相同的，即 b+c。

```
a := b + c
b := b - d
c := c + d
e := b + c
```
(10-4)

可是，如同接下来要讨论的，应用于dag的代数恒等式可以揭示这种等价关系。该序列的dag如图10-12所示。

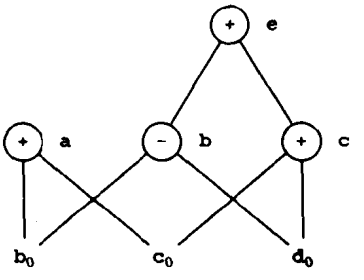


图10-12 基本块(10-4)的dag

在dag上非常容易实现与无用代码删除相对应的操作。我们从dag中删除任何没有活跃变量的根节点（没有祖先的节点）。重复应用该变换会从dag中删除所有和无用代码相对应的节点。

代数恒等式的应用

代数恒等式代表基本块上另一类重要的优化。在9.9节中，我们介绍了一些可以在优化中应用的简单代数变换，例如，我们可以应用像下面这样的算术恒等式: 600

```
x + 0 = 0 + x = x
x - 0 = x
x * 1 = 1 * x = x
x / 1 = x
```

另外一类代数优化包括强度削弱，即用较快的运算符取代较慢的运算符，例如:

```
x ** 2 = x * x
2.0 * x = x + x
x / 2 = x * 0.5
```

第三类相关的优化技术是常量合并。这里我们在编译时对常量表达式进行计算，并利用它们的值取代常量表达式[⊖]。因而，表达式2*3.14将被6.28取代。许多常量表达式是通过使用符号常量出现的。

dag构造过程可以帮助我们应用上述变换和更多其他的通用代数变换，比如交换律和结合律。例如，*是满足交换律的，即 x*y=y*x。在我们创建一个标号为 *，左儿子为 m，右儿子为 n 的新节点之前，我们要检查这样的节点是否已经存在了，然后我们还要检查是否存在具有操作符*、左儿子 n 及右儿子 m 的节点。

关系运算符 <=，>=，<，>，= 和≠有时会生成意外的公共子表达式。例如，条件x>y也可以通过减操作并在减操作所设置的条件代码上执行测试来实现。(减操作可能导致上溢出和下溢出，但是比较操作不会)。因而，只要在dag中为x-y和x>y生成一个节点即可。

结合律亦可以用于寻找公共子表达式。例如，如果源代码有赋值语句

```
a := b + c
e := c + d + b
```

⊖ 编译时和运行时对算术表达式的计算方式应该是一样的。K. Thompson已经提出了一种非常好的常量合并解决办法：编译该常量表达式，在该点执行其目标代码，并用结果代替该表达式。因而编译器不需要包含一个解释器。

则可能生成下面的中间代码:

```
a := b + c
t := c + d
e := t + b
```

如果在该块以外不需要 t，我们可以利用+的交换律和结合律将该序列变换为:

601

```
a := b + c
e := a + d
```

因为计算机算术并不总是符合数学的代数恒等式，编译器的编写者应当仔细检查语言的说明以决定什么样的计算重组是允许的。譬如，如果括号的完整性没有破坏，Fortran 77的标准指出编译器可以计算任何数学上等价的表达式。也就是说，编译器可以把 x*y-x*z 看成 x*(y-z)，但是它不会把 a+(b-c)看成 (a+b)-c。因此，如果要依照语言的定义来优化程序，Fortran编译器必须明了源语言中括号出现的位置。

10.4 流图中的循环

在考虑循环优化之前，我们需要定义流图中的循环是由什么构成的。我们将使用一个节点“支配”另一个节点的概念来定义自然循环和重要的可约流图。寻找支配节点和检查流图可约性的算法将在10.9节中给出。

10.4.1 支配节点

如果从流图的初始节点到节点n的每条路径都要经过节点d，则说节点d支配节点n，记作d *dom* n。根据这个定义，每个节点支配它自身，而循环的入口则支配循环中所有的节点。

例10.6 考虑图10-13的流图，其初始节点为1。初始节点支配所有的节点。节点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节点。除节点1和节点2以外，节点3支配其他所有的节点。因为从节点1出发的所有路径都必须由1→2→3→4或1→3→4开始，除了节点1，2和3以外，节点4支配其他所有的节点。节点5和节点6只支配它们自身，因为控制流可以经过其中一个节点而跳过另一个节点。最后，节点7支配节点7, 8, 9, 10，节点8支配节点8, 9, 10，而节点9和节点10只支配它们自身。 □

表示支配节点信息的一种有用形式是树，叫做支配树。其中，初始节点是树根，树中每个节点仅支配其后代节点。例如，图10-14给出了图10-13中流图的支配树。

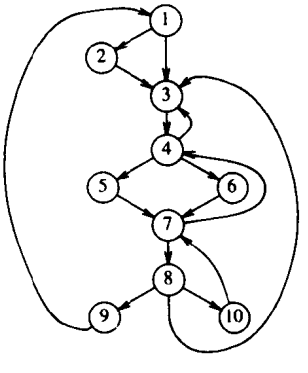


图10-13 流图

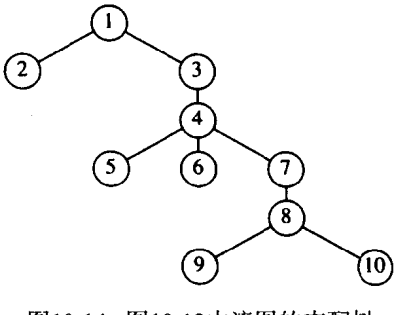


图10-14 图10-13中流图的支配树

支配树的存在依赖于支配节点的性质。每个节点 n 有一个惟一的直接支配节点 m，它在初始节点到 n 的任何路径上都是 n 的最后一个支配节点。根据 *dom* 关系，直接支配节点 m 有下列性质：如果 $d \neq n$，而且 d *dom* n，那么 d *dom* m。

10.4.2 自然循环

支配节点信息的一个重要应用是确定流图中适合于改进的循环。这样的循环应具有两个基本性质：

1. 循环必须有惟一的入口点，称为首节点（header）。这个入口点支配循环中的所有节点，否则它不是该循环的惟一入口。

2. 循环至少迭代一次，也就是至少有一条返回首节点的路径。

602 ~ 603

寻找流图中所有循环的一种好办法是找出流图的回边[⊖]，即头支配尾的边。（如果 $a \rightarrow b$ 是一条边，则 b 是头，a 是尾。）

例10.7 在图10-13中，4 *dom* 7，7→4是回边。类似地，7 *dom* 10，10→7是回边。其他的回边有4→3，8→3及9→1。注意，这些边正好是流图中有可能形成循环的那些边。 □

给定一条回边 $n \rightarrow d$，我们定义该边的自然循环为 d 加上所有不经过 d 而能到达 n 的结点集合。d 是该循环的首节点。

例10.8 边10→7的自然循环由节点7，8和10组成，因为8和10是所有不经过7而能到达10的节点。回边9→1的自然循环是整个流图。（不要忘记路径10→7→8→9!） □

算法10.1 构造回边的自然循环。

输入：流图 G 和回边 $n \rightarrow d$。

输出：由 $n \rightarrow d$ 的自然循环中所有节点构成的集合*loop*。

方法：由节点 n 开始，考虑已放入 *loop* 中的每个节点 m，$m \neq d$，确定将 m 的前驱也放入*loop*中。算法如图10-15所示。除 d 以外，*loop* 中的每个节点一旦加入 *stack*，就要检查它的前驱。注意，因为 d 是初始时放入循环的，我们决不会考察它的前驱，因此找出的只是那些不经过 d 而能到达 n 的节点。 □

```
procedure insert(m);
if m不在loop中 then begin
      loop := loop ∪ {m};
      将m压入栈stack
end;

/* 下面是主程序 */
stack := 空;
loop := { d };
insert(n);
while stack非空 do begin
      从stack 弹出第一个元素m;
      for m的每个前驱p do insert (p)
end
```

604

图10-15 自然循环的构造算法

10.4.3 内循环

如果把自然循环作为“循环”，我们将得到循环的一个有用的性质：除非两个循环的首节点相同，否则它们或者不相交，或者一个完全包含(嵌入)在另一个里面。于是，暂时忽略首节点相同的循环，则内循环的概念是：不包含任何其他循环的循环。

当两个循环具有相同的首节点时，如图10-16那样，则很难说哪个是内循环。例如，若B_1结尾的测试为：

```
if a = 10 goto B2
```

则可能循环 $\{B_0, B_1, B_3\}$ 是内循环。但是，如果不详细检查代码，我们不能保证这一点。可能

⊖ 如果 b *dom* a，则边 $a \rightarrow b$ 称为回边。——译者注

a几乎总是10，那么在进入B_3之前会环绕循环$\{B_0, B_1, B_2\}$很多次。所以我们认为，当两个自然循环具有相同的首节点，并且不是一个嵌在另一个里面时，则将其合并，看作一个循环。

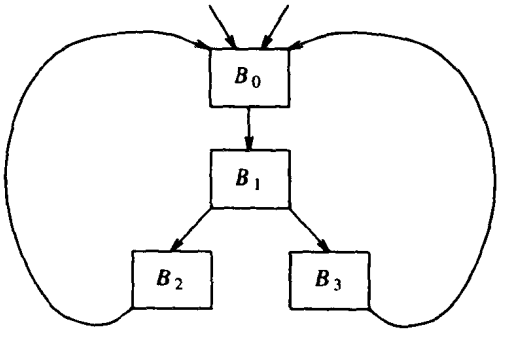


图10-16 具有相同首节点的两个循环

10.4.4 前置首节点

某些变换要求我们将某些语句移到首节点的前面。于是在开始处理循环 L 之前，我们先创建一个称为*前置首节点*的新块。前置首节点的惟一后继是L的首节点，并且原来从L外到达L首节点的边都改成进入前置首节点。从循环L里面到达首节点的边不改变。这种整理如图10-17所示。起初，前置首节点为空，但对L的变换可能会将一些语句放到该节点中。

605

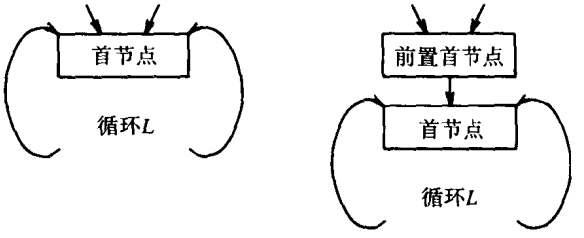


图10-17 前置首节点的引入
a) 引入前 b) 引入后

10.4.5 可约流图

实际出现的流图常常属于下面定义的这类可约流图。单使用像 if-then-else，while-do，continue 和 break 这样的结构化控制流语句所产生的程序，其流图总是可约的。甚至事先不知道结构化程序设计的程序员用 goto 语句编写的程序，也几乎都是可约的。

对于可约流图，已经提出了好几种定义。我们采用的是能显示出其最重要性质之一的定义，即不存在从循环外到循环内的转移，要进入循环只能通过其首节点。练习和文献注释中包含该概念的主要历史。

一个流图 G *是可约的*，当且仅当可以把它的边分成两个不相交的组，其中的边分别叫做*前向边*和*回边*，并且具有下列性质:

1. 所有前向边形成一个无环有向图，在该图中，每个节点都可以从 G 的初始节点到达。
2. 回边组仅由前面所定义的回边组成。

例10.9 图10-13中的流图是可约的。通常，如果知道了流图的 *dom* 关系，就可以找出并去掉所有的回边。如果流图是可约的，那么剩下的边必定是前向边。所以要检查流图是否可约，只要检查所有前向边是否构成无环有向图便可以了。对于图10-13，如果删掉5条回边4→3，7→4，8→3，9→1和10→7，则很容易看出剩下的图是无环的。 □

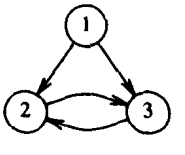


图10-18 一个不可约流图

例10.10 考虑图10-18的流图，其初始节点为1。该流图没有回边，因为没有头支配尾的边。因此，如果整个图是无环的，

那么它就是可约的。但因为它不是无环的，所以该流图不可约。直观地讲，该流图不可约的原因是可以从节点2和节点3两处进入环2-3。 □

用于循环分析的可约流图有一个关键性质，就是当流图中的一些节点被看作是循环的节点集合时，这些节点之间一定包含一条回边。事实上，为了找出流图可约程序中的所有循环，只要检查回边的自然循环即可。相反地，图10-18的流图似乎有一个由节点2和节点3构成的“循环”，但是不存在这样的回边，使这个“循环”是自然循环。实际上，该“循环”有两个首节点，即节点2和节点3，它使得许多代码优化技术，如10.2节介绍的代码外提和归纳变量删除，都不能直接应用。

幸好，像图10-18这样的不可约控制流结构在大多数程序里面很少出现，所以研究具有多个首节点的循环没有多大价值。甚至有些语言，如Bliss和Modula 2，只允许程序有可约流图；其他许多语言，只要不使用 goto 语句，亦只产生可约流图。

例10.11 再次回到图10-13，可以看出仅有的“内循环”是{7, 8, 10}，它是回边10→7的自然循环。集合{4, 5, 6, 7, 8, 10}是7→4的自然循环（注意，8和10可经边10→7到达7）。直观看上去，{4, 5, 6, 7}是一个循环，但这是错的，因为4和7都是入口点，违反了我们单入口的要求。从另一个角度说，没有理由认为控制会环绕着节点集合 {4, 5, 6, 7} 消耗较多的时间，控制从7到8的次数与从7到4的次数相比很难知道谁多谁少。把8和10包含在这个循环里，我们更确信已分离出了程序频繁执行的一个区域。

但应该认识到，作出分支频率的假设是危险的。例如，若把循环{7, 8, 10}中的不变语句移出8或10，而事实上控制经过7→4比经过7→8的次数更频繁，因此，实际上我们将增加被移动语句的执行次数。我们将在10.7节讨论避免这个问题的方法。

下一个较大的循环是{3, 4, 5, 6, 7, 8, 10}，它是回边4→3和8→3的自然循环。同前面一样，如果把{3, 4}看成循环将违反单入口点的要求。最后一个循环（回边9→1的自然循环）是整个流图。 □ 606 ~ 607

可约流图还有一些有用的性质，我们将在10.9节讨论深度优先搜索和区间分析的时候，再对其进行介绍。

10.5 全局数据流分析介绍

为了优化代码，编译器需要把程序作为一个整体来收集信息，并把这些信息分配给流图的各个基本块。例如，在9.7节我们看到，了解每个基本块的出口处哪些变量是活跃的可以改进寄存器的利用率。10.2节提出了怎样用全局公共子表达式的知识去删除冗余计算，同样地，9.9节和10.3节讨论了为执行像常量合并和无用代码删除这样的变换，编译器怎样利用“到达定义”。例如，在到达给定块之前，要知道像debug这样的变量是在哪里最后被定义的。这些信息只是优化编译器通过*数据流分析*所收集到的*数据流信息*的少数例子。

通过在程序的各个点建立和求解与信息有关的方程系统即可收集数据流信息。典型的方程形式如下：

$$out\,[S] = gen\,[S] \cup (\,in\,[S] - kill\,[S]\,) \tag{10-5}$$

这个方程的意思是，当控制流通过一个语句时，在语句末尾得到的信息或者是在该语句中产生的信息，或者是进入语句开始点时携带的并且没有被这个语句注销的那些信息。这样的方程叫

做数据流方程。

建立和求解数据流方程依赖于3个因素：

1. 产生和注销的概念依赖于所需要的信息，即依赖于所要解决的数据流分析问题。而且，对于某些问题，不是沿着控制流前进，由 *in* [*S*] 来定义 *out* [*S*]，而是反向前进，由 *out* [*S*]来定义*in*[*S*]。

2. 因为数据沿控制路径流动，所以数据流分析受程序控制结构的影响。事实上，我们写*out*[*S*]时隐含地认为语句有一个惟一的结束点；一般地，方程是在基本块级而不是在语句级建立的，因为基本块确实有惟一的结束点。

3. 有些难以捉摸的问题会伴随着像过程调用、通过指针变量的赋值甚至对数组变量的赋值等语句而产生。

608 在本节中，我们将考虑如何确定到达程序中某一点的定义的集合以及它们在寻找常量合并的机会中的用途。在本章的后面，代码外提和归纳变量删除算法也将使用这些信息。

开始我们先考虑用 **if** 和 **do-while** 语句构造的程序。这些语句中可预测的控制流允许我们将注意力集中在需要建立和求解数据流方程的地方。本节中的赋值语句是复制语句或者是形如 `a := b+c` 的语句。本章中我们将经常用“+”作为一种典型的运算符。我们所讲的所有内容均可直接应用于其他运算符，包括单运算对象和多于两个运算对象的运算符。

10.5.1 点和路径

在基本块中，我们将经常谈到相邻语句间的点，以及第一个语句前和最后一个语句后的点。例如，图10-19中的块 B_1 有4个点：第1个点在所有赋值语句前，每个语句后还各有1个点。

现在，我们以全局观点来考虑所有块中的所有点。从 p_1 到 p_n 的路径是点的序列 p_1，p_2，…，p_n，而且对1和 $n-1$ 间的每个 i 满足下面条件1、2中的一个：

1. p_i 是紧接在一个语句前面的点，p_{i+1} 是同一块中紧跟在该语句后面的点。

2. p_i 是某基本块的结束点，而 p_{i+1} 是后继块的开始点。

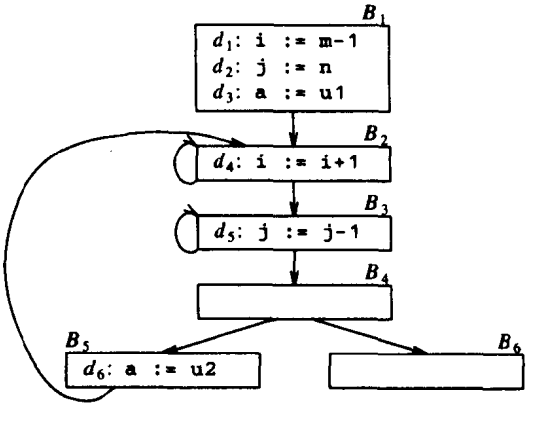


图10-19 一个流图

609 **例10.12** 在图10-19中，有一条从块 B_5 的开始到块 B_6 的开始的路径。它经过块 B_5 的最后一点，然后依次通过块 B_2，B_3 和 B_4 的所有点，最后到达块 B_6 的开始。 □

10.5.2 到达定义

变量 x 的定义是一条赋值或可能赋值给x的语句。最普通的定义是对 x 的赋值或从 I/O 设备读一个值并存储在 x 中的语句，这些语句的确为 x 定义了一个值，称为 x 的*明确定义*。也还有一些语句，它们可能为 x 定义一个值，叫做*含糊定义*。x 的含糊定义的最常见形式为：

1. 把 x 作为参数的过程调用（传值参数除外）或者可能访问 x 的过程，因为 x 在过程的作用域之内。还要考虑“别名”的可能性，x 虽然不在该过程的作用域内，但 x 等同于另一个变量，这个变量被作为参数传递或在此作用域内。

2. 通过可能指向 x 的指针赋值。例如，如果 q 可能指向 x 的话，则赋值语句 `*q := y` 是 x 的定义。确定指针可能指向什么变量的方法将在10.8节进行讨论。在这里由于不清楚指针具体指向哪个变量，我们必须假定通过指针的赋值是对每个变量的定义。

如果存在一条从紧跟 d 的点到 p 的路径，并且在这条路径上 d 没有被“注销”，则说定义 d到达点 p。直观地，如果某个变量 a 的定义 d 到达点 p，那么 p 引用的 a 的最新定义可能在位置d。如果沿着这条路径的某两点间存在 a 的定义，那么我们将注销（kill）变量 a 的那个定义。注意，只有 a 的明确定义才能注销 a 的其他定义。这样，一个点可以由一条路径上的明确定义和出现在后面的同一变量的含糊定义到达。

例如，图10-19的块 B_1 中的定义 i := m-1和 j := n 到达块 B_2 的开始点，如果在块 B_4 和块 B_5中不对j赋值或读j，块 B_3 的定义 j := j-1后面的部分也是如此，那么定义 j := j-1 也可到达块 B_2。不过，块 B_3 中对 j 的赋值注销了定义 j := n，因此这个定义不能到达块 B_4，B_5 或 B_6。

通过定义前面的到达定义，有时我们允许一定的不精确，但是，它们都是在“安全”和“稳妥”的范围内。例如，我们假设流图的所有边都能被遍历到，但实际上可能不是这样。例如，不管 a 和 b 是什么值，控制也不会到达下面程序段中的赋值语句 a := 4：

610

```
if a = b then a := 2
else if a = b then a := 4
```

一般地，确定流图中是否每条路径都会被经过是一个不可判定问题，我们不打算解决这个问题。

再次提醒一下，在设计代码改进变换时，面临任何怀疑，我们都必须采取保守的决策，虽然保守决策会使我们失去某些实际上可安全进行的变换。如果一个决策永远不会导致程序计算结果的改变，则该决策就是保守的。在到达定义的应用中，假定定义能够到达一个可能不会到达的点是保守的。从而，在程序的任何执行中，我们允许存在可能总也不会被遍历的路径，而且允许定义穿越同一变量的含糊定义。

10.5.3 结构化程序的数据流分析

像do-while语句这样的控制流结构的流图具有一种有用的性质：控制只能从一个开始点进入，而且当语句结束时只能从一个结束点离开。对下面语法所表示的语句，当我们讨论到达语句开始和结束的定义时可以发现这条性质。

$S \rightarrow$ **id** := E | S ; S | **if** E **then** S **else** S | **do** S **while** E
$E \rightarrow$ **id** + **id** | **id**

该语言中的表达式和中间代码中的相似，但是语句的流图限制在图10-20中的形式之内。这一节的首要目的是研究图10-21中的数据流方程。

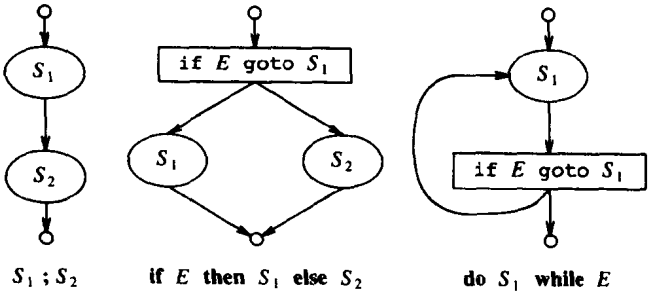


图10-20 一些结构化控制结构

我们将流图中的一部分（称为区域）定义为包含首节点的一组节点 N，其中首节点支配区域中的所有其他节点。除了进入首节点的那些边以外，N 中所有节点之间的边都在区域中[⊖]。

⊖ 循环是区域的特例，它是强连通的，而且包含所有到首节点的回边。

与语句 S 对应的那部分流图是一个区域，它遵守更严格的限制：当控制离开该区域时，只可以流到一个外面的块中。

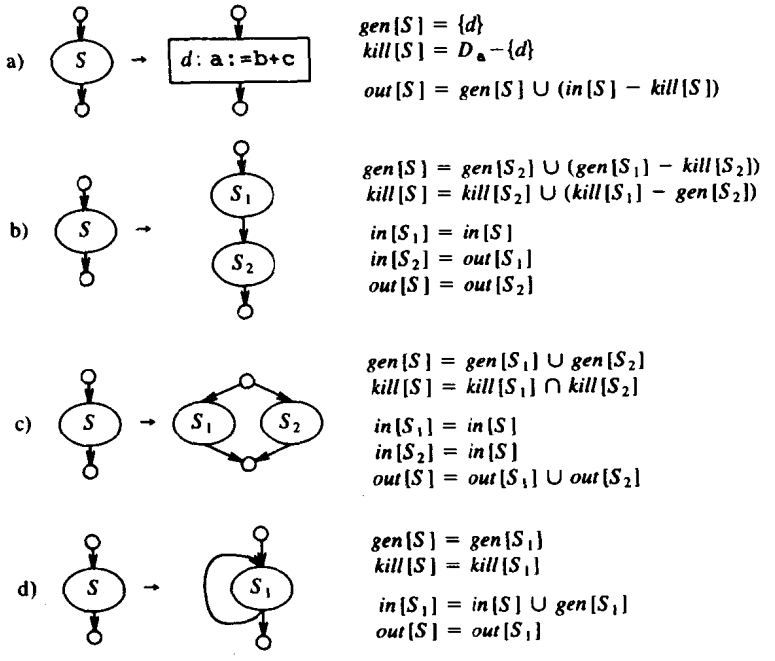


图10-21 到达定义的数据流方程

为了技术上的方便，我们假设有一个没有语句的哑块（图10-20中由空心圆表示），控制刚好在进入区域之前或者刚好在离开区域之后才通过这种块。哑块在一个语句区域的入口和出口处的开始点分别是该语句的开始点和结束点。

611～612

图10-21中的方程是关于所有语句 S 的 $in[S]$，$out[S]$，$gen[S]$ 和 $kill[S]$ 集合的归纳定义或者说是语法制导定义。集合 $gen[S]$ 和 $kill[S]$ 是综合属性，它们是自底向上、从最小的语句集到最大的语句集进行计算的。我们的要求是如果 d 到达 S 的末尾，则定义 d 在 $gen[S]$中，而与它是否到达 S 的开始无关。从另一个角度来看，d 必须出现在S中并且通过不会转到 S 外面的路径到达 S 的末尾。这就是称 $gen[S]$ 是“由 S 产生的”定义之集的理由。

类似地，我们想让 $kill[S]$ 作为从未到达 S 末尾的定义的集合（即使它们能到达 S 的开始）。这样，将这些定义看成“被S注销”是有意义的。为了使定义 d 在 $kill[S]$ 中，从 S 的开始到末尾的每一条路径上都必须有一个 d 所定义的同一变量的明确定义，而且如果 d 出现在 S 中，则其后沿任意路径的 d 的每次出现都必须是同一变量的另一个定义。[⊖]

因为是综合翻译，gen 和 $kill$ 的规则比较容易理解。首先，观察图10-21a中对变量 a 的一个赋值的规则。该赋值语句确实是 a 的定义，假定为定义 d。那么不管 d 是否到达语句的开始，它是惟一的确实能到达语句末尾的定义。因此，

$$gen[S] = \{d\}$$

另一方面，d 注销了 a 的所有其他定义，所以我们有

$$kill[S] = D_a - \{d\}$$

⊖ 本节是介绍性的，我们将假定所有的定义都是明确的。10.8节将处理含糊定义所需要的修正。

其中，D_a 是程序中变量 a 的所有定义的集合。

如图10-21b所示，串联语句的规则更微妙一些。由 $S = S_1 ; S_2$ 所产生的定义 d 所处的环境是什么呢？首先，如果它是由 S_2 产生的，那么它当然就是由 S 产生的。如果 d 是由 S_1 产生的，假设它没有被 S_2 注销它将到达 S 的末尾。于是，我们有

$$gen[S] = gen[S_2] \cup (gen[S_1] - kill[S_2])$$

将同样的推理应用于定义的注销，我们将得到

$$kill[S] = kill[S_2] \cup (kill[S_1] - gen[S_2])$$

如图10-21c所示，对于 if 语句，我们注意到：如果 if 语句的任一分支产生一个定义，那么该定义将到达语句 S 的末尾。从而，

$$gen[S] = gen[S_1] \cup gen[S_2]$$

但是，为了注销定义 d，在从 S 的开始到 S 的结束的所有路径上 d 所定义的变量都必须被注销。特别地，它必须在任何分支上都被注销，所以

$$kill[S] = kill[S_1] \cap kill[S_2]$$

613

最后，考虑图10-21d中循环的规则。简而言之，循环对 *gen* 和 *kill* 不会产生影响。如果定义d是在 S_1 中产生的，那么它将到达 S_1 和 S 的末尾。相反，如果 d 是在 S 中产生的，它只可能是在 S_1 中产生的。如果 d 被 S_1 注销，那么执行循环将不起作用，d 的变量在每一次循环中都将在 S_1 中被重新定义。相反，如果 d 被 S 注销，那么它一定是被 S_1 注销的。我们的结论是：

$$gen[S] = gen[S_1]$$
$$kill[S] = kill[S_1]$$

10.5.4 对数据流信息的保守估计

在图10-21中给出的 *gen* 和 *kill* 的规则中，有一处细微的误算。我们已经假设 if 和 do 语句中的条件表达式 E 是“不可解释”的，也就是说，存在使程序的控制经过某一分支的输入。另一方面，我们假设流图中的任何图论路径也是执行路径，即当程序以至少一个可能的输入运行时被执行的路径。

这种情况并不是总能发生的，事实上通常我们不能确定哪一个分支将被选中。例如，假设 if 语句中的表达式 E 总为真，那么在图10-21c中通过 S_2 的路径将永远不能被选中。这将带来两个结果。第一，由 S_2 产生的定义并不真的是由 S 产生的，因为没有从 S 的开始到达语句 S_2 的途径。第二，*kill* $[S_1]$中没有定义能到达 S 的末尾。因此，每一个这样的定义即使不在 *kill*$[S_2]$中，逻辑上也应该在 *kill* $[S]$中。

当我们将计算出来的 *gen* 和真实的 *gen* 进行比较时，我们发现真实的 *gen* 总是计算出来的 *gen* 的子集。另一方面，真实的 *kill* 总是计算出来的 *kill* 的超集。这些包含关系即使在我们考虑图10-21的其他规则时仍然成立。譬如，如果 do-S-while-E 语句中的表达式E永远不能为假，那么我们就不能从循环中跳出。这样，真实的 *gen* 就是 ∅，而且每一个定义都被循环注销了。在图10-21b中语句串联的情况下，必须考虑由于无穷循环而不能从 S_1 或 S_2 跳出的情况，我们将其留作练习。

很自然地会考虑到，在真实的 *gen* 和 *kill* 与计算出的 *gen* 和 *kill* 之间存在差异是否会对数据流分析产生严重的影响。答案依赖于这些数据的用途。在到达定义这种特定情况下，我们通常利用这些信息来推断在某一点上变量 x 的值是否被限制在一个可能性很少的范围内。例如，

如果我们发现到达该点的x的惟一定义是形如 x := 1的语句,我们可以推断在该点上 x 的值为1。于是，我们可以决定用对1的引用取代对x的引用。

614 因而，过高地估计到达一个点的定义集合看起来并不严重，它只不过会阻止我们进行一些合理的优化。另一方面，过低地估计定义集合却是致命的错误，它会导致改变程序的计算结果。例如，我们可能认为 x 的所有到达定义都给x赋值1，因而用1取代x，但是可能存在另一个未检测到的到达定义给 x 赋值2。于是，对于到达定义这种情况，如果估计集是真实到达定义集合的超集（不需要是真超集），则称定义集合是安全的或是保守的。如果它不是真实集合的超集，则称估计是不安全的。

对于每一个数据流问题，我们必须检查不准确估计对它们可能导致的各种程序变化所产生的影响。通常我们会接受安全的差异，因为它们至多只能阻止合法优化的进行，但我们不能接受不安全的差异，因为它们可能会导致改变程序行为的优化。在每一个数据流问题中，真实答案的子集或者超集通常是安全的（但二者不能同时满足）。

现在回到对到达定义的 *gen* 和 *kill* 进行安全估计的问题上来，请注意其中的差异，*gen* 的超集和 *kill* 的子集都在安全的范围内。直观地，增加 *gen* 将会增加到达一个点的定义的集合，但不能阻止一个定义到达它确实能到达的地方。同样地，减少 *kill* 只能增加到达任何给定点的定义的集合。

10.5.5 *in* 和 *out* 的计算

许多数据流问题可以像计算 *gen* 和 *kill* 那样用综合翻译来解决。例如，我们可能希望为每一个语句 *S* 确定由 *S* 定义的变量集合。这样的信息可以通过类似于 *gen* 的方程来计算，它甚至不需要类似于 *kill* 的集合。譬如，这个方法可以用来确定循环不变的计算。

当然，还存在许多其他类型的数据流信息，如我们曾经当作例子的到达定义问题，我们还需要计算某些继承属性。可以证明 *in* 是一个继承属性，*out* 是依赖于 *in* 的综合属性。考虑到整个程序的控制流，我们希望 *in*[*S*] 是到达 *S* 开始的定义的集合，包括在 *S* 之外的语句或者包围 *S* 的语句。类似地可以定义 *out*[*S*]为到达 *S* 结束的定义的集合。但一定要注意 *out*[*S*]和 *gen*[*S*]的区别。后者是没有经过 *S* 外面的路径而到达 *S* 结尾的定义的集合。

615 作为二者区别的一个简单例子，考虑图10-21b中的串联语句。语句 *d* 可能是在 S_1 中产生的，所以会到达 S_2 的开始。如果 *d* 不在 *kill*[S_2]中，*d* 将到达 S_2 的末尾，因此会在 *out*[S_2]中。但是，*d* 不在 *gen*[S_2]中。

如果 S_0 是整个程序，那么 *in*[S_0] = ∅。基于此，在自底向上地为所有的语句*S*计算完*gen*[*S*]和 *kill*[*S*]之后，我们就可以从表示整个程序的语句开始计算 *in* 和 *out*。也就是说，没有定义会到达整个程序的开始。对于图10-21中的每一种语句，我们可以假设 *in*[*S*]是已知的。我们必须利用它来计算 *S* 的每一个子语句的 *in*（第2~4种情况无足轻重，而第1种情况是不相关的）。然后，我们递归地（自顶向下）计算每一个子语句 S_1 或 S_2 的 *out*，并用这些集合计算*out*[*S*]。

最简单的情况是图10-21a，其中只有一个赋值语句。假设*in*[*S*]是已知的，我们用方程(10-5)来计算 *out*，亦即：

$$out[S] = gen[S] \cup (in[S] - kill[S])$$

如果一个定义是由 *S* 产生的（即定义 *d* 是一个语句），或者它到达语句 *S* 的开始而且没有被该语句注销，那么该定义会到达 *S* 的末尾。

假设我们已经算出了 *in*[*S*]，而且 *S* 是两个语句 S_1；S_2 的串联，如图10-21的第二种情况那样。我们从 *in*[S_1] = *in*[*S*] 开始。然后，我们递归地计算 *out*[S_1]，从它我们可以得到 *in*[S_2]，这

是因为当且仅当一个定义到达 S_1 的末尾时该定义才到达 S_2 的开始。现在我们可以递归地计算 $out[S_2]$，该集合与 $out[S]$相等。

接下来，考虑图10-21c中的if语句。因为我们已经保守地假设控制会经过每个分支，当一个定义到达 S 的开始时，它也恰好到达 S_1 或者 S_2 的开始，亦即：

$$in[S_1] = in[S_2] = in[S]$$

从图10-21c中的图还可以看出，当且仅当一个定义到达了一个或者所有子语句的末尾时该定义才到达 S 的末尾，即

$$out[S] = out[S_1] \cup out[S_2]$$

于是，我们可以使用这些方程通过 $in[S]$来计算 $in[S_1]$和 $in[S_2]$，再递归地计算 $out[S_1]$和 $out[S_2]$，然后用它们来计算 $out[S]$。

10.5.6 处理循环

最后一种情况，图10-21d有一些特殊问题。我们再次假设 $gen[S_1]$ 和 $kill[S_1]$ 已经被自底向上地算出，还假定 $in[S]$ 已经给定，就如同在对分析树执行深度优先遍历的过程中那样。和情况(b)与(c)不同的是，我们不能简单地将 $in[S]$ 作为 $in[S_1]$ 使用，因为 S_1 中到达 S_1 末尾的定义能够沿着弧线返回到 S_1 的开始，因此这些定义也在 $in[S_1]$ 中。更合适地，我们有：

$$in[S_1] = in[S] \cup out[S_1] \tag{10-6}$$

616

对 $out[S]$ 显然有如下方程：

$$out[S] = out[S_1]$$

一旦我们计算出 $out[S_1]$，就可以使用该方程。但是，看来似乎只有在计算出 $out[S_1]$ 之后我们才能用(10-6)计算 $in[S_1]$，而我们本来是要先计算语句的 in，再用它来计算 out 的。

幸运的是，有一个根据 in 计算 out 的直接方法，(10-5)中已经给出，在本例中其形式如下：

$$out[S_1] = gen[S_1] \cup (in[S_1] - kill[S_1]) \tag{10-7}$$

理解这里发生了什么是非常重要的。我们并没有真正明白为什么(10-7)对于任意的语句S_1都是正确的，我们只是猜测它应该是正确的，因为感觉上，当且仅当一个定义是在该语句中产生的或者它到达语句的开始而且没有被该语句注销时，该定义才应该到达语句的末尾。但是，我们知道，为一条语句计算 out 的惟一方法是利用图10-21a至图10-21c中给出的公式。首先，我们要假设(10-7)成立并推导出图10-21d中 in 和 out 的方程。然后，我们能够用图10-21a至图10-21d中的方程证明(10-7)对任意的 S_1 都成立。通过对语句 S 的大小进行归纳，我们可以将这些证明合在一起构成一个有效的证明：方程(10-7)和所有图10-21中的方程对于S和它的所有子语句均成立。我们不准备证明而将其留作练习，但是下面进行的推理有一定的指导作用。

即使假设(10-6)和(10-7)均成立，我们仍然没有脱离困境。这两个方程同时对 $in[S_1]$ 和 $out[S_1]$ 进行循环定义。我们可以将方程简写为

$$\begin{aligned} I &= J \cup O \\ O &= G \cup (I - K) \end{aligned} \tag{10-8}$$

其中 I，O，J，G 和 K 分别对应着 $in[S_1]$，$out\,[S_1]$，$in\,[S]$，$gen\,[S_1]$ 和 $kill\,[S_1]$。前两个是变量，其他三个是常量。

为求解方程 (10-8)，开始我们假设 $O = \varnothing$。然后，我们可以用(10-8)的第一个方程得到 I 的

一个估计，即

$$I^1 = J$$

接下来，我们可以利用第二个方程得到对 O 的更好的估计：

$$O^1 = G \cup (I^1 - K) = G \cup (J - K)$$

将第一个方程用于 O 的这个新的估计上，我们得到：

$$I^2 = J \cup O^1 = J \cup G \cup (J - K) = J \cup G$$

如果我们再应用第二个方程，则 O 的下一估计是：

617 $$O^2 = G \cup (I^2 - K) = G \cup (J \cup G - K) = G \cup (J - K)$$

注意，$O^2 = O^1$。这样，如果我们计算 I 的下一个估计，它将等于 I^1，I^1 使得我们对 O 的下一次估计等于 O^1，如此等等。于是，I 和 O 的限制值是上面给出的 I^1 和 O^1 的值。从而，我们可以导出图10-21d中的方程，即

$$in[S_1] = in[S] \cup gen[S_1]$$
$$out[S] = out[S_1]$$

第一个方程是从上面的计算得来的，第二个是对图10-21d中的图进行分析得来的。

剩下的细节是为什么我们有资格从估计 $O = \varnothing$开始。回忆一下我们对于保守估计的讨论，我们建议 O 所代表的像 $out[S_1]$这样的集合应当多估计一些而不是少估计。事实上，如果我们从 $O = \{d\}$开始，其中 d 是没有出现在 J，G 和 K 中的定义，那么 d 应当在对 I 和 O 的限制取值中结束。

在此，我们必须援引 *in* 和 *out* 的预期含义，如果这样的 d 确实在 $in[S_1]$中，就必须有一条从 d 到 S_1 的开始的路径，它说明了 d 是怎样到达那个点的。如果 d 在 S 之外，那么 d 必须在 $in[S]$中，而如果 d 在 S 中（从而在 S_1 中），它必须在 $gen[S_1]$中。在第一种情况下，d 在 J 中，故利用(10-8)可以放到 I 中。在第二种情况下，d 在 G 中，利用(10-8)中的 O 又可传到 I 中。结论是，从非常小的估计开始，并通过将更多的定义加入 I 和 O 来向上构建是估计 $in[S_1]$的一种安全的方法。

10.5.7 集合的表示

使用位向量可以简洁地表示像 $gen[S]$ 和 $kill[S]$ 这样的定义的集合。我们为流图中每个感兴趣的定义分配一个号码，那么表示定义集合的位向量在位置 i 的值为1的充分必要条件是，编号为 i 的定义在该集合中。

定义语句的编号可以被看作是一个数组中语句的索引，该数组中存有指向语句的指针。但是，在全局数据流分析中并不是所有的定义都是我们感兴趣的。例如，只用在单个块中的临时变量的定义就不需要赋予编号，计算表达式时产生的大部分临时变量就是这样的。因此，感兴趣的定义的编号将被存放在一个单独的表中。

集合的位向量表示还可以高效地实现集合操作。两个集合的并和交操作可分别用逻辑或（**or**）和逻辑与（**and**）来实现，这是大多数面向系统的程序设计语言所提供的基本操作。集合 A 和 B 的差 $A - B$ 可以通过求 B 的补集，然后利用逻辑与（**and**）来计算 $A \wedge \neg B$ 来实现。

618 **例10.13** 图10-22给出了一段含有7个定义的程序，并在语句左面的注释中将这7个定义标记为 d_1 到 d_7。图10-23的语法树中，在每个节点的左面给出了表示图10-22中各语句的 *gen* 和

kill 集合的位向量。这些集合本身是通过对语法树节点所表示的语句使用图10-21的数据流方程计算出来的。

考虑图10-23右下角的节点 d_7，*gen* 集合{d_7}用000 0001来表示，*kill* 集合{d_1, d_4}用100 1000表示。也就是说，d_7 注销了其变量 *i* 的所有其他定义。

if 节点的第二个和第三个子节点分别表示条件语句的 then 和 else 部分。注意，在 **if** 节点上的 *gen* 集合000 0011是在第二个和第三个子节点上的集合000 0010和000 0001的并。*kill* 集合是空集，因为被 then 和 else 注销的定义是不相交的。

```
/* d1 */    i := m-1;
/* d2 */    j := n;
/* d3 */    a := u1;
            do
/* d4 */        i := i+1;
/* d5 */        j := j-1;
            if e1 then
/* d6 */            a := u2
            else
/* d7 */            i := u3
            while e2
```

图10-22 说明到达定义的程序

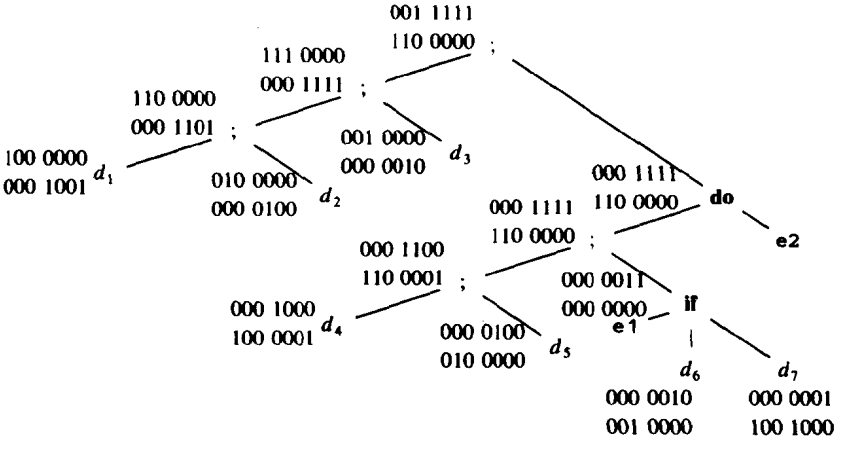


图10-23 语法树节点上的 *gen* 和 *kill* 集合

将串联语句的数据流方程应用到 **if** 节点的父节点上，即可得到该节点的 *kill* 集合：

$$000\,0000 \cup (110\,0001 - 000\,0011) = 110\,0000$$

简而言之，条件语句什么也没有注销，被语句 d_4 注销的 d_7 是由条件语句产生的，所以只有d_1和 d_2 在 **if** 节点的父节点的 *kill* 集合中。

现在，我们可以从分析树的顶端开始计算 *in* 和 *out*。我们假设语法树根节点的 *in* 集合是空的。从而根节点的左儿子的 *out* 是那个节点的 *gen* 集合，即111 0000。这也是 **do** 节点上 *in* 集合的值。从图10-21中与 **do** 产生式相关联的数据流方程可以看出，**do** 循环内语句的 *in* 集合是通过对 **do** 节点上的 *in* 集合111 0000和该语句上的 *gen* 集合000 1111取并获得的。该并集为 111 1111，因此所有定义都可以到达循环体的开始。但是，刚好在定义 d_5 前面的点 *in* 集合是 011 1110，这是因为定义 d_4 注销了 d_1 和 d_7。余下的 *in* 和 *out* 计算留作练习。 □

10.5.8 局部到达定义

通过只保存特定点的信息而在需要时重新计算中间点的信息，我们可以用时间来换取存储数据流信息的空间。全局流分析通常以基本块为单位，这样可以将注意力集中在基本块的开始点。由于通常存在比基本块更多的点，所以将精力集中在基本块上将是很大的节约。需要时，从基本块开始点的到达定义可以计算出基本块中每个点的到达定义。

更详细地，考虑基本块 B 中的赋值语句序列 S_1；S_2；…；S_n。我们将 B 的开始记做点 p_0，语句 S_i 和 S_{i+1} 之间的点记做 p_i，基本块的末尾记做点 p_n。从 S_1 开始考虑语句 S_1；S_2；…；S_j，并应用图10-21中串联语句的数据流方程，我们可以从 $in[B]$ 获得到达点 p_j 的定义。初始时，令 $D = in[B]$。当考虑 S_i 的时候，我们从 D 中删除被 S_i 注销的定义，而加入由 S_i 产生的定义。最后，D 由到达 p_j 的定义组成。

619
~
620

10.5.9 引用-定义链

将到达定义信息存为“引用-定义链”或“ud链”是非常合适的，它是一个列表，对于变量的每一次引用，到达该引用的所有定义都在该列表中。如果块 B 中变量 a 的引用之前没有任何a的明确定义，那么 a 的这次引用的 ud 链为 $in[B]$ 中 a 的定义的集合。如果 B 中在 a 的这次引用之前存在 a 的明确引用，那么只有 a 的最后一次定义会在 ud 链中，而 $in[B]$ 不能放在 ud 链中。另外，如果存在 a 的含糊定义，那么所有那些在该定义和 a 的这次引用之间没有 a 的明确定义的定义都将被放在 a 的这次引用的 ud 链中。

10.5.10 计算顺序

第5章我们讨论了在属性计算期间预留空间的技术，使用图10-21中那样的规范说明，这些技术也可以应用到数据流信息的计算中。特别地，对 *gen*，*kill*，*in* 和 *out* 集合的计算顺序只有一种约束，那就是这些集合之间的依赖关系。选定了一种计算顺序以后，如果某个集合的所有引用都已完成，我们就可以释放该集合的空间。

本节的数据流方程和第5章中属性的语义规则在一个方面有所不同，属性间的循环依赖在第5章是不允许的，但是我们已经看到数据流方程中可能存在循环依赖。例如，式(10-8)中的 $in[S_1]$ 和 $out[S_1]$ 就互相依赖。对于到达定义问题，我们可以重写数据流方程以删除循环（试比较一下式(10-8)与图10-21中没有循环的方程）。一旦得到一个没有循环的规范说明，第5章的技术即可用来求数据流方程的高效解。

10.5.11 一般控制流

数据流分析必须要考虑所有的控制路径。如果从语法上可以直接看出控制路径是什么，则可按语法制导的方式建立和求解数据流方程，正如本节所讨论的一样。如果程序可以包含 goto 语句或更严格的 break 和 continue 语句，则必须修改我们所采用的方法以处理实际的控制路径。

有好几种方法可以被采用。下一节介绍的迭代法适用于任意的流图。因为存在 break 和 continue 语句时的流图是可约的，所以使用10.10节将要讨论的基于区间的方法可以系统地处理这样的结构。

但是，允许程序含有 break 和 continue 语句时不必放弃语法制导法。在结束这一节之前，我们只考虑一个介绍如何支持 break 语句的实例，而把这些思想的详细介绍留到10.10节。

例10.14 图10-24中 **do-while** 循环中的 break 语句和跳转到循环末尾的转移语句等价。那么我们将如何定义如下语句的 *gen* 集合呢？

621

```
if e3 then a := u2
else begin i := u3; break end
```

因为 d_6 是沿这条语句开始到结束的控制路径所

```
/* d1 */   i := m-1;
/* d2 */   j := n;
/* d3 */   a := u1;
           do
/* d4 */       i := i+1;
/* d5 */       j := j-1;
               if e3 then
/* d6 */           a := u2
               else begin
/* d7 */               i := u3;
                       break
               end
           while e2
```

图10-24 含有一个 break 语句的程序

产生的惟一定义，所以我们将 *gen* 定义为$\{d_6\}$，其中 d_6 是定义 a := u2。处理整个 **do-while** 循环时，会考虑定义 d_7，即 i := u3。

当我们处理循环体中的语句时，有一种方法允许我们避免 break 语句所引起的跳转：如图10-25所示，我们取 break 语句的 *gen* 和 *kill* 集合分别为空集和 U（所有定义的全集）。图10-25中剩下的 *gen* 和 *kill* 集合是用图10-21的数据流方程所确定的，图中，*gen* 集合显示在 *kill* 集合之上。语句 S_1 和 S_2 代表赋值语句序列。do 节点上的 *gen* 和 *kill* 集合留待确定。

以 break 结尾的任何语句序列的结束点都是不可到达的，所以取语句序列的 *gen* 集为 ∅，*kill* 集为 U 是没有害处的；结果仍然是 *in* 和 *out* 的保守估计。类似地，if语句的结束点只能通过then部分到达，而且图10-25中 **if** 节点上的 *gen* 和 *kill* 集合也确实和其第二个子节点的相应集合相同。 622

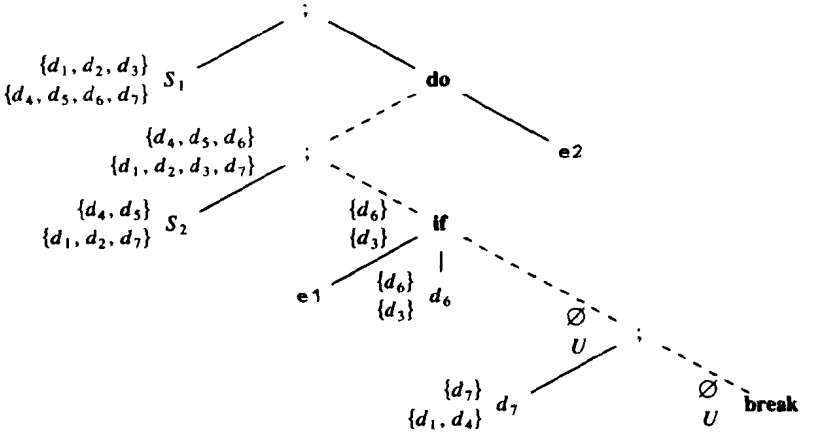


图10-25 break 语句对 *gen* 和 *kill* 集合的影响

do 节点上的 *gen* 和 *kill* 集合必须考虑 do 语句从开始到结尾的所有路径，所以它们受到 break 语句的影响。现在让我们来计算两个集合 G 和 K，开始当我们遍历从 do 节点到 break 节点的虚线路径时，它们都是空集。在直觉上，G 和 K 表示从循环体开始到 break 语句的控制流所产生和注销的定义。do-while 语句的 *gen* 集合可以通过对 G 和循环体的 *gen* 集求并集来确定，因为控制要到达 **do** 的末尾，不是从 break 语句就是通过了循环体。出于同样的原因，**do** 的 *kill* 集合是通过对 K 和循环体的 *kill* 集求交集来确定的。

就在我们到达 **if** 节点之前，我们有 $G = gen[S_2] = \{d_4, d_5\}$，$K = kill[S_2] = \{d_1, d_2, d_7\}$。在 **if** 节点上，我们对如下情况感兴趣：当控制流到了 break 语句时，条件语句的 then 部分对 G 和 K 不产生影响。沿虚线路径的下一个节点是语句序列节点，所以我们计算 G 和 K 的新值。将语句序列节点（标记为 d_7 的节点）的左儿子表示的语句记做 S_3，我们使用下式计算 G 和 K：

$$G := gen[S_3] \cup (G - kill[S_3])$$

$$K := kill[S_3] \cup (K - gen[S_3])$$

于是，到达 break 语句时 G 和 K 的值分别是$\{d_5, d_7\}$和$\{d_1, d_2, d_4\}$。 □ 623

10.6 数据流方程的迭代解

上一节的方法既简单又有效，但是对于像 Fortran 或 Pascal 等允许任意流图的语言来说，它就不能满足需要了。10.10节将讨论“区间分析”法，在严重损害概念复杂性的前提下，这是一种有利于用语法制导方法对一般流图进行数据流分析的方法。

在此，我们要讨论另外一种解决数据流问题的重要方法。我们首先建立流图，然后同时计算每个节点的 *in* 和 *out* 集合，而不是用分析树来驱动 *in* 和 *out* 集合的计算。讨论这种新方法时，我们还要利用该机会向读者介绍一些不同的数据流分析问题，给出它们的一些应用，并指出这些问题的不同之处。

有许多数据流问题的方程，它们“产生”和“注销”信息的形式是类似的。但是，有两种主要的方法对应的方程在细节上却有所不同。

1. 上一节到达定义的方程是前向方程，意即 *out* 集合是根据 *in* 集合计算出来的。我们还会看到数据流问题是后向的，即 *in* 集合是从 *out* 集合计算出来的。

2. 当有多条边进入块 *B* 时，到达 *B* 的开始点的定义是沿每一条边到达的定义的并，因此我们称该并操作是聚合操作（confluence operator）。相反，我们会考虑像全局可用表达式这样的问题，其中交操作是聚合操作，因为只有当表达式在 *B* 的每一个前继的末尾都是可用的，它在 *B* 的开始才是可用的。在10.11节，我们将看到聚合操作的其他例子。

在本节中，前向和后向方程的例子我们都会看到，它们依次将并和交作为聚合操作符。

10.6.1 到达定义的迭代算法

同上一节一样，我们可以为每个基本块 *B* 定义 *out*[*B*]，*gen*[*B*]，*kill*[*B*] 和 *in*[*B*]，注意，每个块 *B* 可以看作是一个或多个赋值语句的串联。假设已经计算出了每一个块的 *gen* 和 *kill*，如式(10-9)所示，我们可以创建两组和 *in* 与 *out* 有关的方程。第一组方程是基于这样的观察得来的，即 *in*[*B*] 是从 *B* 的所有前驱到达的定义的并。第二组方程是我们声称对所有语句都成立的
624 通用规则(10-5)的特例。这两组方程为：

$$\begin{aligned} in[B] &= \bigcup_{B\text{的前驱}P} out[P] \\ out[B] &= gen[B] \cup (in[B] - kill[B]) \end{aligned} \tag{10-9}$$

如果流图有 *n* 个基本块，我们将从(10-9)得到2*n* 个方程。通过循环计算 *in* 和 *out* 集合即可对这2*n* 个方程进行求解，就像上一节求解 **do-while** 语句的数据流方程(10-6)和(10-5)那样。上一节中，开始时我们把空集作为所有 *out* 集合的起始估计。从(10-9)可以看出，作为 *out* 集合的并，*in* 集合在 *out* 集合为空时亦为空，因此，这里我们也将以空的 *in* 集合开始。虽然我们可以认同方程(10-6)和(10-7)只需要一次迭代，但如果它们是更复杂的方程，我们就不能预先限定迭代的次数。

算法10.2 到达定义。

输入：已经算出每个块 *B* 的 *kill*[*B*] 和 *gen*[*B*] 的流图。

输出：每个块 *B* 的 *in*[*B*] 和 *out*[*B*]。

方法：我们使用迭代法，对所有的 *B*，从“估计” *in*[*B*] = ∅ 开始，然后逐步收敛到 *in* 和 *out* 的期望值。因为我们必须重复迭代一直到 *in*(*out*)收敛（converge），我们利用一个布尔变量 *change* 来记录在对块的每一遍扫描中 *in* 是否发生了变化。算法框架如图10-26所示。 □

直观地，只要定义没有被注销，算法10.2就会将它们传播得尽可能远，从某种意义上说，就是模拟程序所有可能的执行。文献注释中包括一些相应的参考书，在这些参考书中可以找到该问题和其他一些数据流分析问题正确性的形式化证明。

我们可以看到，算法最终会停止，因为对任何 *B*，*out*[*B*]都不再变小，一旦有定义增加，它将永远停留在那里（该事实的证明留为练习）。因为所有定义的集合是有穷的，所以最终一定会有一次 while 循环，使得在第(9)行中对每个 *B* 都有 *oldout* = *out*[*B*]，于是，*change* 将保持

为 **false**，而算法也会停止。我们可以安全地终止算法，因为如果 *out* 没有发生变化，在下一次循环中 *in* 也不会发生变化。并且如果 *in* 不发生变化，*out* 也不会发生变化，所以在随后的每次循环中，不可能再发生变化。

```
    /* 初始化out，假设对所有的B，in[B] = ∅ */
(1) for 每个块B do out[B] := gen[B];
(2) change := true;          /* 使得while循环继续下去 */
(3) while change do begin
(4)     change := false;
(5)     for 每个块B do begin
(6)         in[B] := ∪ out[P];
                     B的前驱P
(7)         oldout := out[B];
(8)         out[B] := gen[B] ∪ (in[B] − kill[B]);
(9)         if out[B] ≠ oldout then change := true
        end
    end
```

图10-26 计算*in* 和*out* 的算法

可以证明 while 循环次数的上界是流图中节点的个数。直观地，其原因是如果一个定义到达了某一点，它可以沿着一条无环路的路径这样做，而流图中节点的数量是无环路路径上节点数量的上界。每次执行 while 循环，该定义沿着这条路径至少前进一个节点。

事实上，如果我们在第(5)行的 for 循环中适当地安排块的顺序，有充分的证据表明在实际程序上迭代的平均数将小于5（见10.10节）。因为这些集合可以用位向量表示，而且这些集合上的操作可以用位向量上的逻辑操作来实现，所以，算法10.2的效率实际上非常高。

例10.15 图10-27中的流图来自上一节图10-22中的程序，我们将把算法10.2应用于该流图以便对这两节的方法进行比较。

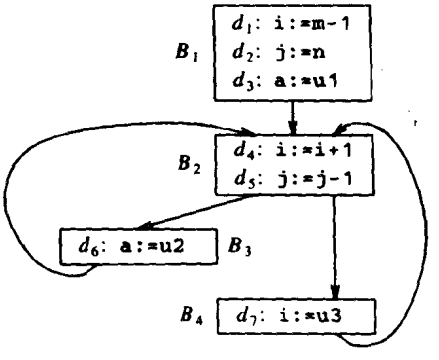


图10-27 用于说明到达定义的流图

我们只对图10-27中定义 i，j 的定义 d_1，d_2，…，d_7 以及 a 感兴趣。同上一节一样，我们将用位向量来表示定义的集合，其中，从左边数起位*i*表示定义d_i。

625 ~ 626

图10-26中第(1)行的循环对每个块*B*进行初始化，即 *out*[*B*] = *gen*[*B*]，图10-28的表中给出了这些 *out*[*B*] 的初始值。虽然没有计算或使用各个 *in*[*B*] 的初始值（∅），但为完整起见也在表中给了出来。假设第(5)行的 for 循环按照 $B = B_1, B_2, B_3, B_4$ 的顺序执行。当 $B = B_1$时，因为初始节点没有前驱节点，所以 $in[B_1]$ 仍然为空集，用000 0000表示；结果，$out[B_1]$ 还是等于

$gen[B_1]$。该值和第(7)行计算出来的 *oldout* 没有区别，所以我们还是没有将 *change* 置为 **true**。

块B	初始时		第1遍		第2遍	
	$in[B]$	$out[B]$	$in[B]$	$out[B]$	$in[B]$	$out[B]$
B_1	000 0000	111 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	000 1100	111 0011	001 1110	111 1111	001 1110
B_3	000 0000	000 0010	001 1110	000 1110	001 1110	000 1110
B_4	000 0000	000 0001	001 1110	001 0111	001 1110	001 0111

图10-28 *in* 和*out* 的计算

然后我们考虑 $B = B_2$，并计算下式：

$$in[B_2] = out[B_1] \cup out[B_3] \cup out[B_4]$$
$$= 1110000 + 0000010 + 0000001 = 1110011$$
$$out[B_2] = gen[B_2] \cup (in[B_2] - kill[B_2])$$
$$= 0001100 + (1110011 - 1100001) = 0011110$$

计算结果如图10-28所示。在第1遍的末尾，$out[B_4]$ = 001 0111，反映了这样的事实，即 d_7 是由 B_4 产生的，而 d_3，d_5 和 d_6 到达 B_4，并且在 B_4 中没有被注销。从第2遍开始，任何 *out* 集合都不再有变化，所以算法终止。 □

10.6.2 可用表达式

如果从初始节点到 p 的每一条路径（不必是无环路的）均计算 x + y，并且在到达 p 之前的最后一次这样的计算之后，再没有对 x 或 y 的赋值，则表达式 x + y 在 p 点是可用的。对于可用表达式，如果一个块为 x 或 y 赋值（或可能赋值），并且后来没有重新计算 x + y，则我们称该块注销了表达式 x + y。如果一个块明确地计算了 x + y，并且后来没有重新定义 x 或 y，则该块产生表达式 x + y。

注意，可用表达式的“注销”或者“产生”同到达定义中的“注销”或者“产生”的概念并不完全相同。不过，这些“注销”和“产生”的概念和到达定义中的遵守相同的法则。倘若
627 我们修改一下图10-21a中针对简单赋值语句的规则，我们就可以完全按10.5节那样计算它们。

可用表达式信息的主要用途是检测公共子表达式。例如，在图10-29中，如果4*i在 B_3 的入口处是可用的，则块 B_3 中的表达式4*i就是公共子表达式。如果 i 在 B_2 中没有被赋予新值或者像图10-29b中那样，在 B_2 中，对i赋值后又重新计算了 4*i，则 4*i将是可用的。

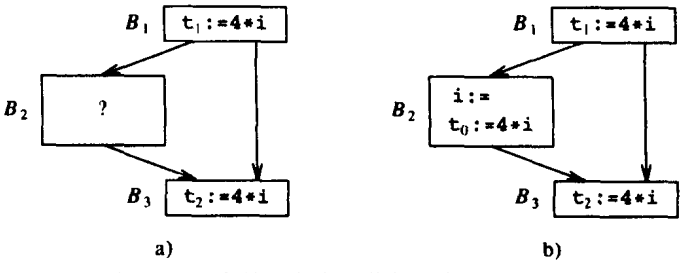


图10-29 穿越基本块的潜在公共子表达式

从一个块的开始到结尾，我们可以很容易地计算出块中每个点所产生的表达式集合。假设在该块前面的点没有可用表达式。如果在点 p 表达式集 A 是可用的，而且 q 是 p 后面的点，

它们之间还有语句 x := y+z，那么我们可以用下面的两个步骤来形成 q 点的可用表达式集：

1. 将表达式 y+z 添加到 A 中。

2. 从 A 中删除任何含有 x 的表达式。

注意，这些步骤必须按照正确的顺序执行，因为 x 可能与 y 或 z 相同。到达块的末尾之后，A 将是该块所产生的表达式集。该块所注销的表达式集是所有这样的表达式 y+z，其中 y 或 z 是在该块中定义的，但 y+z 不是该块所产生的。

例10.16 考虑图10-30中的4条语句。在第一条语句之后，b+c 是可用的。第二条之后 a-d 成为可用的，但是 b + c 不再可用，因为 b已经被重新定义。第三条语句还是没有将 b + c变为可用表达式，因为 c 的值立刻又被改变了。在最后一个语句之后，a - d 不再是可用的，因为 d 已经发生了变化。因此，没有语句被产生，而且所有包含 a，b，c 或 d 的语句都被注销了。 □

语句	可用表达式
……	没有
`a := b+c`	
……	只有 `b+c`
`b := a-d`	
……	只有 `a-d`
`c := b+c`	
……	只有 `a-d`
`d := a-d`	
……	没有

图10-30 可用表达式的计算

我们可以用到达定义的计算方法找出可用表达式。假设U是程序中出现在一个或多个语句右面的所有表达式的全集。对每个块 B，令 $in[B]$是 U 中这样的表达式之集，它们恰好在 B的开始点之前是可用的，令$out[B]$是 B 的结束点之后的这样的集合。将 $e_gen[B]$定义为 B 所产生的表达式，且 $e_kill[B]$ 为 U 中被 B 注销的表达式集合。注意，in，out，e_gen 和 e_kill 均可用位向量表示。下面的方程将未知的 in 和 out，以及已知的 e_gen 和 e_kill 联系在一起。

$$
\begin{aligned}
&out[B] = e_gen[B] \cup (in[B] - e_kill[B]) \\
&in[B] = \bigcap_{B\text{的前驱}p} out[P]\text{，}B\text{不是初始块} \\
&in[B_1] = \varnothing\text{，其中 }B_1\text{是初始块}
\end{aligned}
\tag{10-10}
$$

方程(10-10)和到达定义的方程(10-9)看起来几乎是相同的。第一个不同点是初始节点的 in 是作为特殊情况处理的。可以证明这一点是正确的，因为如果程序恰好从初始节点开始的话，则没有表达式是可用的，即使某些表达式可能在沿程序中其他地方到初始节点的所有路径上都是可用的。如果我们不强制使 $in[B_1]$ 为空的话，我们可能错误地推导出某些表达式在程序开始之前就是可用的。

第二个不同点，也是更重要的不同点，在于聚合操作符不是并而是交。该操作是正确的，因为只有当表达式在一个块的所有前驱的末尾都是可用的，它在该块的开始才是可用的。相反，一个定义只要到达某个块的一个或多个前驱的末尾，它就到达该块的开始。

使用∩而不是∪使得方程(10-10)和(10-9)有所不同。因为哪一组方程都没有惟一解，所以对于(10-9)它是对应“到达”定义的最小解，开始我们假设没有任何定义可以到达任何地方，然后逐步增大解，最后获得该最小解。这种方法中，除非可以找到一条将 d 传播到 p 的实际路径，否则我们不假设定义 d 能够到达点 p。相反，对于方程(10-10)，我们希望得到一个最大可能解，所以我们从一个非常大的近似解开始并逐步减小该近似解。

628
↓
629

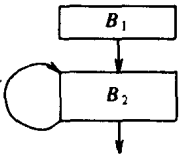
开始我们假设“任何表达式，即集合U在任何地方都是可用的”，然后我们删除这样的表

达式，即我们能够找到一条路经使得沿这条路径该表达式是不可用的，这样，我们可以得到一个确实可用的表达式集合，但这一点并不是显然的。在可用表达式的情况下，产生一个精确的可用表达式集合的子集是比较稳妥的，我们也是这样做的。子集之所以稳妥的依据是，这些信息的用途是用前面算出的值来替换可用表达式的计算（见下一节的算法10.5），而不知道一个表达式是可用的只是限制了我们对代码进行变换。

例10.17 我们将集中讨论图10-31中的单个块 B_2，来说明 $in[B_2]$ 的初始近似值对 $out[B_2]$ 的影响。令 G 和 K 分别为 $gen[B_2]$ 和 $kill[B_2]$ 的缩写。块 B_2 的数据流方程为：

$$in[B_2] = out[B_1] \cap out[B_2]$$
$$out[B_2] = G \cup (in[B_2] - K)$$

令 I^j 和 O^j 分别表示 $in[B_2]$ 和 $out[B_2]$ 的第 j 次近似值，则这些方程可以重写为图10-31中的递归形式。该图还说明，从 $I^0 = \varnothing$ 开始，我们将得到 $O^1 = O^2 = G$，而从 $I^0 = U$ 开始，我们将得到一个更大的集合 O^2。在每一种情况下，恰好都有 $out[B_2]$等于 O^2，因为两次迭代都收敛在显示的点上。



$O^{j+1} = G \cup (I^j - K)$
$I^{j+1} = out[B_1] \cap O^{j+1}$

$I^0 = \varnothing$	$I^0 = U$
$O^1 = G$	$O^1 = U - K$
$I^1 = out[B_1] \cap G$	$I^1 = out[B_1] - K$
$O^2 = G$	$O^2 = G \cup (out[B_1] - K)$

图10-31 将*in*初始化为空集限制太严

直观地，从 $I^0 = U$ 开始，利用

$$out[B_2] = G \cup (out[B_1] - K)$$

所得到的解是令人满意的，因为它正确地反映出了 $out[B_1]$ 中没有被 B_2 注销的表达式在 B_2 的末尾是可用的，就如同 B_2 所产生的表达式一样。 □

630

算法10.3 可用表达式。

输入：流图 G，其中每个块 B 的 $e_kill[B]$ 和 $e_gen[B]$ 都已算出。初始块是 B_1。

输出：每个块 B 的 $in[B]$ 集合。

方法：执行图10-32中的算法，各个步骤的解释同图10-26中的类似。 □

```
in[B1] := ∅;
out[B1] := e_gen[B1];  /* 初始节点 B1 的 in 和 out 从不改变 */
for B ≠ B1 do out[B] := U - e_kill[B];  /* 初始估计过大 */
change := true;
while change do begin
    change := false;
    for B ≠ B1 do begin
        in[B] := ∩ out[P];
                 P∈B的前驱
        oldout := out[B];
        out[B] := e_gen[B] ∪ (in[B] - e_kill[B]);
        if out[B] ≠ oldout then change := true
    end
end
```

图10-32 可用表达式计算

10.6.3 活跃变量分析

许多代码改进变换依赖于按照与程序中控制流相反的方向计算出来的信息。现在我们将考

虑其中的某些变换。在*活跃变量分析*中，我们希望知道对于变量 x 和点 p，在流图中沿从 p 开始的某条路径是否可以引用 x 在 p 点的值。如果可以，则我们说 x 在 p 点是*活跃的*，否则，x 在 p 点就是*无用的*。

正如我们在9.7节所看到的，活跃变量信息在目标代码生成时具有重要的作用。当我们在寄存器中计算一个值之后，并且假设在某个块中还要引用它，如果它在该块的末尾是无用的，则不需要存储该值。如果所有的寄存器都是满的，而且我们需要另外的寄存器，则我们应当使用含有无用值的寄存器，因为这种寄存器中的值不再需要了。

我们将 *in*[*B*] 定义为在紧靠 *B* 之前的点活跃的变量之集，而将 *out*[*B*] 定义为在紧跟 *B* 之后的点活跃的变量之集。令 *def* [*B*]为 *B* 中这样的变量之集：在引用该变量之前已明确地对该变量进行了赋值；而令 *use* [*B*]为 *B* 中这样的变量之集：在该变量的任何定义之前可能引用该变量。那么将 *def* 和 *use* 与未知的 *in* 和 *out* 联系在一起的方程就是： 631

$$
\begin{aligned}
in[B] &= use[B] \cup (out[B] - def[B]) \\
out[B] &= \bigcup_{B\text{的后继}S} in[S]
\end{aligned}
\tag{10-11}
$$

第一组等式表示一个变量在进入某个块时是活跃的，如果它在该块中于定义前被引用，或者它在离开该块时是活跃的而且在该块中没有被重新定义。第二组等式表示一个变量在离开某个块时是活跃的，当且仅当它在进入该块的某个后继时是活跃的。

应当注意(10-11)和到达定义方程(10-9)之间的关系。在此，*in* 和 *out* 将它们的作用进行了交换，而 *use* 和 *def* 则分别取代了 *gen* 和 *kill* 。同(10-9)一样，(10-11)的解也不必惟一，我们只需要最小解。用来求最小解的算法本质上是算法10.2的回退。因为检测 *in* 中任何变化的机制同算法10.2与算法10.3中检测 *out* 变化的方法类似，所以我们省略了检查终止条件的细节。

算法10.4 活跃变量分析。

输入：已经计算了每个块 *def* 和 *use* 的流图。

输出：*out*[*B*]，流图中在每个块 *B* 的出口活跃的变量之集。

方法：执行图10-33中的程序。 □

```
for每个块B do in [B] := ∅;
while 集合in发生变化 do
    for 每个块B do begin
        out[B] =  ∪  in[S]
                 B的后继S
        in[B] := use[B] ∪ (out[B] - def[B])
    end
```

图10-33 活跃变量计算

10.6.4 定义-引用链

有一种事实上和活跃变量分析方式相同的计算，即*定义-引用链接*（du链接）。如果在语句 s 中需要一个变量的右值，则称该变量在语句 *s* 中*被引用*。例如，b 和 c（但没有 a）在语句 `a := b+c` 和 `a[b] := c` 中都被引用过。定义-引用链接问题是对某个点 p 计算变量 x 的引用 *s* 的集合，使得从 p 到 *s* 有一条没有重新定义 x 的路径。 632

就像计算活跃变量那样，如果我们能够计算 *out* [*B*]，即从块 *B* 的末尾能够到达的引用之集，我们就可以通过扫描块 *B* 中 p 后面的那部分来计算从 *B* 中任意点 p 到达的定义。特别地，如果块中有一个变量 x 的定义，我们就可以确定该定义的 du链，即该定义所有可能引用的列表。该方法类似于10.5节讨论的计算ud链的方法，我们把它留给读者。

计算du链接信息的方程看起来完全像式(10-11)，只是替换了 *def* 和 *use*。用来代替 *use* [*B*] 的是 *B* 中向上暴露的引用的集合，即序对(*s*，x)的集合，其中，*s* 是 *B* 中引用变量 x 的语句，而且在 *B* 中之前没有出现 x 的定义。取代 *def* [*B*]的也是序对(*s*，x)的集合，其中，*s* 是引用 x

的语句，s 不在 B 中，但 B 含有 x 的定义。这些方程显然可以利用类似于算法10.4的方法求解，我们就不进行进一步的讨论了。

10.7 代码改进变换

10.2节介绍的完成代码改进变换的算法依赖于数据流信息。在上两节中，我们已经明白了怎样收集这些信息。在此，我们将考虑公共子表达式删除、复制传播、循环不变计算外提和归纳变量删除。对于许多语言，改进循环的代码可以使运行时间得到很大改进。在编译器中，有些变换可以一起完成，但是我们这里提出的想法是基于单个变换的。

本节强调的是将程序信息作为整体使用的全局变换。正如在上两节中所见到的，全局数据流分析通常不关心基本块中的点，因此全局变换代替不了局部变换，两者都必须实施。例如，当执行全局公共子表达式删除时，我们只关心一个表达式是否是由基本块产生的，而不关心它是否在块中被重新计算了几次。

10.7.1 全局公共子表达式删除

上节讨论的可用表达式数据流问题允许我们判定位于流图中 p 点的表达式是否为公共子表
633 达式。下面的算法将10.2节提出的删除公共子表达式的直观想法加以形式化。

算法10.5 全局公共子表达式删除。

输入：带有可用表达式信息的流图。

输出：修正后的流图。

方法：对每个形如x := y+z[⊖]的语句 s，如果 y+z 在 s 所在块的开始点可用，且该块中在语句 s 之前没有对y或z的定义，则执行下面的步骤：

1. 为了寻找到达 s 所在块的 y+z 的计算，我们顺着流图的边，从该块开始反向搜索，但不穿过任何计算 y+z 的块。在遇到的每个块中，对 y+z 的最后一次计算是到达 s 的 y+z 的计算。

2. 建立新变量 u。

3. 把步骤1中找到的每个语句 w := y+z 用如下语句代替：

```
u := y+z
w := u
```

4. 用 x := u 代替语句 s。 □

下面是关于该算法的一些说明：

1. 步骤1中寻找到达 s 的 y+z 的计算也可以形式化为一个数据流分析问题。但是，为所有的表达式 y+z 和所有的语句或基本块求解这个问题是没有意义的，因为这样会收集到太多无关信息。所以我们宁可在流图上搜索相关的语句和表达式。

2. 由算法10.5完成的修改并非都是改进。我们可能需要限制在步骤1中发现的到达s的不同计算的个数，很可能限制到1。不过，下面将要讨论的复制传播在有多个 y+z 的计算到达s时也能获得益处。

3. 算法10.5将会漏掉这样的事实：a*z和c*z在下面的语句中具有相同的值：

```
a := x+y        c := x+y
b := a*z   和   d := c*z
```

⊖ 我们仍用+代表一般的运算符。

因为处理公共子表达式的这种简单方法只考虑字面上的表达式，而不考虑表达式所计算的值。Kildall[1973]提出了一种方法，它可以在一遍扫描中找到这样的等价表达式，我们将在10.11节讨论它。但是，用算法10.5进行多遍扫描也能找到它们，可以考虑重复算法直到没有新的变化为止。如果 a 和 c 是临时变量，它们在块外没有被引用，那么，对临时变量做特殊处理，也可以找到公共子表达式(x+y)*z，见下面的例子。 634

例10.18 假定在图10-34a的流图中没有对数组 a 的赋值，我们可以安全地认为a[t_2]和a[t_6]是公共子表达式。问题是怎样删除这个公共子表达式。

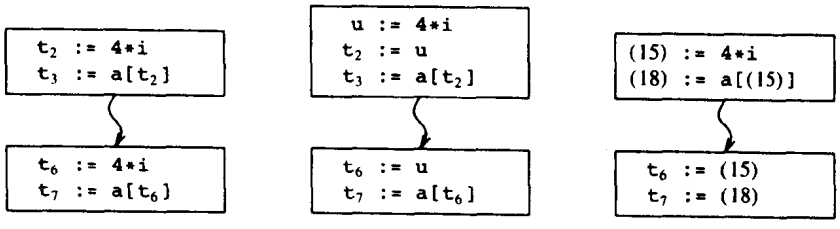


图10-34 公式子表达式4*i的删除

图10-34a中的公共子表达式 4*i在图10-34b中已经被删除。确定 a[t_2]和 a[t_6]也是公共子表达式的一种方法是用复制传播（下面将会讨论）把 t_2 和 t_6 换成 u，这样两个表达式都变成了a[u]，重新使用算法10.5即可删除它。注意，图10-34b的两个块中都插入了相同的新变量 u，所以局部的复制传播足以把 a[t_2]和 a[t_6]均变成 a[u]。

另一种方法是考虑到这样的事实：临时变量是由编译器插入的，而且只能在它们出现的块中被引用。通过仔细考察可用表达式计算过程中表达式的表示方式，我们将获得这样的事实：不同的临时变量可能表示相同的表达式。表示表达式集的推荐技术是为每个表达式编号，然后用位向量来表示，其中第i位表示编号为i的表达式。可以用5.2节的值-编号技术来对表达式进行编号，以便以一种特殊的方式来处理临时变量。

更详细地，假设4*i的值为15。如果我们使用值号码15而不是临时变量名 t_2 和 t_6，则表达式a[t_2]和a[t_6]将得到相同的值号码。又假设结果值号码为18。这样在数据流分析中位18将同时表示a[t_2]和a[t_6]，而且我们可以确定a[t_6]是可用的，因而可以将其删除。结果代码如图10-34c所示，我们使用(15)和(18)来表示与带有相应值号码的表达式相对应的临时变量。实际上，t_6是没有用的，而且将在局部活跃变量分析过程中被删除。临时变量t_7也将不会被计算，而是将对 t_7 的引用换成对(18)的引用。 635 □

10.7.2 复制传播

前面刚讨论的算法10.5和其他一些算法，如本节将要讨论的归纳变量删除算法，都会引入形如 x := y 的复制语句。复制也可能是由中间代码生成器直接产生的，虽然它们大多都只包括局部于基本块的临时变量，而且可以用9.8节讨论的dag构造算法删除掉。如果能找出复制语句 s: x := y 中定义 x 的所有引用点，并用 y 代替 x，那么就可以删除该复制语句。前提是 x的每个引用 u 必须满足下列条件：

1. 语句 s 必须是到达 u 的 x 的惟一定义（即引用 u 的ud链只包含 s）。

2. 在从 s 到 u 的每条路径，包括穿过 u 若干次（但没有第二次穿过 s）的路径上，没有对 y 的赋值。

条件1可以用ud链信息来检查，但如何检查条件2呢？我们将建立一个新的数据流分析问题，其中 *in*[*B*]是复制语句 *s*：x := y的集合，就是从初始节点到块 *B*（包含语句 *s*）开始点的每条路径上，以及语句 *s* 的最后一次出现之后，都没有对y的赋值。相应地可以定义集合 *out*[*B*]，只不过它是关于块 *B* 的结束点的。如果复制语句 *s*：x := y出现在块 *B* 中，且块 *B* 中后来没有对 y 进行赋值，则称复制语句 *s* 是在块 *B* 中产生的。如果 x 或 y 在块 *B* 中后来被赋值，并且 *s* 不在块 *B* 中，则称 *s*：x := y 在块 *B* 中被注销。对x的赋值注销 x := y 的概念类似于到达定义，但是对 y 赋值也会注销 x := y 是该问题所特有的。请注意不同的赋值 x := y 会互相注销这个事实的重要结果，*in*[*B*] 中只能含有一个 x 在左边的复制语句。

令 *U* 为程序中所有复制语句的全集，注意，不同位置的语句 x := y 在 *U* 中是不相同的。将 *c_gen*[*B*] 定义为块 *B* 所产生的所有复制语句的集合，*c_kill*[*B*]为*U*中所有被 *B* 注销的复制语句的集合。那么下列方程将这些定义联系在一起：

$$
\begin{aligned}
&out[B] = c_gen[B] \cup (in[B] - c_kill[B]) \\
&in[B] = \bigcap_{B\text{的前驱}P} out[P]，B\text{不是初始块} \\
&in[B_1] = \varnothing，\text{其中}B_1\text{是初始块}
\end{aligned}
\tag{10-12}
$$

636

如果 *c_kill* 和 *c_gen* 分别由 *e_kill* 和 *e_gen* 代替的话，那么方程(10-12)和方程(10-10)就是一样的。所以，式(10-12)可用算法10.3求解，因而在此不再讨论。不过，我们将给出一个例子来揭示复制传播的某些细微差别。

例10.19 考虑图10-35的流图。这里，*c_gen* [B_1] = {x := y}，*c_gen* [B_3] = {x := z}；*c_kill* [B_2] = {x := y}（因为y在B_2中被赋值），*c_kill* [B_1] = {x := z}（因为x在 B_1 中被赋值），*c_kill* [B_3] = {x := y}（同样的原因）。

其他的 *c_gen* 和 *c_kill* 为 ∅，由方程(10-12)，*in* [B_1] = ∅。算法10.3的一遍扫描可以确定下式成立：

$$in[B_2] = in[B_3] = out[B_1] = \{x := y\}$$

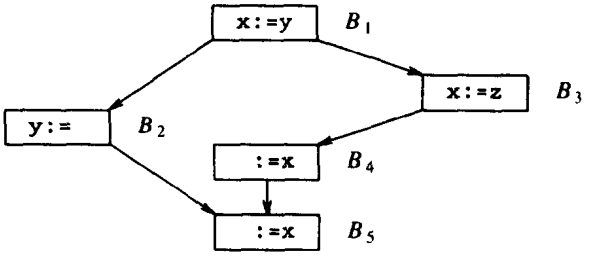


图10-35 流图的例子

类似地，*out*[B_2] = ∅，而且

$$out[B_3] = in[B_4] = out[B_4] = \{x := z\}$$

最后，*in*[B_5] = *out*[B_2] ∩ *out*[B_4] = ∅。

我们注意到，按算法10.5的原则，复制 x := y 和 x := z 都不能到达块 B_5 中 x 的引用，虽然按到达定义的含义，这些 x 的定义都能到达块 B_5。这两个复制几乎都不能传播，因为不能用 y（或z）代替定义 x := y（或x := z）所能到达的 x 的所有引用，仅能做的是将块 B_4 中的 x 用 z 代替，但这样做并没有改进代码。 □

现在我们详细说明删除复制语句的算法。

算法10.6 复制传播。

输入：流图 *G*，带有给出到达块 *B* 的定义的ud链，以及表示方程(10-12)的解的 *c_in*[*B*]，

637 即沿着每条路径到达块 *B* 的复制语句 x := y 的集合，在这些路径上 x := y 的最后一次出现之后没有对 x 或 y 的赋值。我们还需要能给出每个定义的引用的du链。

输出：修正后的流图。

方法：对每个复制语句 s：x := y 执行下列步骤：

1. 确定出由该 x 的定义，即 s：x := y，所能到达的那些 x 的引用。

2. 对步骤1中找到的每个 x 的引用，确定 s 是否在 $c_in[B]$中，其中块 B 是含有 x 的这次引用的基本块，而且块 B 中该引用的前面没有 x 或 y 的定义。回想一下，如果 s 在 $c_in[B]$ 中，那么s 是惟一的到达块 B 的 x 的定义。

3. 如果 s 满足步骤2的条件，则删掉 s，且把步骤1中找出的所有 x 的引用用 y 来代替。 □

10.7.3 循环不变计算的检测

我们将利用ud链来检测那些循环不变计算，即只要控制不离开循环，其值就不改变的那些计算。10.4节已讨论过，循环是由一组基本块组成的区域，其首节点支配所有其他块，所以进入循环只能通过首节点。我们还要求从循环中的任何块至少有一条路径回到首节点。

如果循环中含有赋值 x := y+z，而 y 和 z 所有可能的定义都在循环外面（包括 y、z 是常数的特殊情况），那么 y+z 就是循环不变计算。因为只要控制不离开循环，每次碰到 x := y+z 时 y+z 的值都是一样的。所有这样的赋值都可以从ud链中找到，ud链就是到达赋值语句 x := y+z 的 y 和 z 的所有定义点列表。

识别出 x := y+z 计算的 x 值在循环中不变后，如果循环中还有另一条语句 v := x+w，其中 w 也只能在循环外定义，那么 x+w 也是循环不变计算。

根据上面的想法，可以对循环进行重复扫描，找出越来越多的循环不变计算。如果我们同时具有ud链和du链，则甚至不需要重复扫描代码。定义 x := y+z 的du链告诉我们 x 的值在哪儿被引用，我们只要检查循环中的这些x的引用，看它们是否引用 x 的其他定义（通过ud链）即可。如果除 x 以外，其他运算对象也是循环不变量，则这些循环不变赋值可以移到前置首节点中，正如下面的算法所讨论的那样。

算法10.7 循环不变计算检测。

输入：由一组基本块构成的循环 L，每个基本块包括一系列的三地址语句。对于每个三地址语句，10.5节计算的ud链都是可用的。 638

输出：从控制进入循环 L 一直到离开 L，每次都计算同样值的三地址语句的集合。

方法：我们只给出算法的非形式化说明，相信能说清它的基本原理：

1. 将下面这样的语句标记为“不变”：它们的运算对象或者是常数，或者它们的所有到达定义都在循环 L 的外面。

2. 重复步骤3，直到某次重复没有新的语句可标记为“不变”为止。

3. 将下面这样的语句标记为“不变”：它们先前没有被标记，而且它们的所有运算对象或者是常数，其到达定义都在循环 L 之外，或者只有一个到达定义，这个定义是循环 L 中已标记为“不变”的语句。 □

10.7.4 代码外提

找出循环不变语句后，可以对其中的一些语句实施称为代码外提的优化，即把这些语句移到循环的前置首节点。下面3个条件保证代码外提不会改变程序计算的内容。没有一个条件是非要不可的，之所以用这些条件，是因为它们易于检查，并且可用到实际程序中。后面还将讨论放宽这些条件的可能性。

将语句 s：x := y+z 外提的条件是：

1. 含有语句 s 的块是循环中所有出口节点的支配节点，出口节点指的是有后继节点不在循环中的节点。

2. 循环中没有其他语句对 x 赋值。如果x是只赋值一次的临时变量，这个条件肯定满足，不必检查。

3. 循环中 x 的引用仅由 s 到达，如果 x 是临时变量，这个条件一般也满足。

下面3个例子分别针对上面这些条件。

例10.20 把不需要在循环中执行的语句移到循环外，可能会改变程序计算的内容，如图10-36所示。该例触发条件1，因为只要不陷入无限循环，支配所有出口的语句一定会执行。

考察图10-36a中的流图，块 B_2，B_3 和 B_4 形成一个循环，块 B_2 是首节点，块 B_3 中的语句 i := 2 显然是循环不变的，但是块 B_3 并不支配惟一的出口块 B_4。如果把 i := 2 外提到新的前
639 置首节点 B_6 中，如图10-36b所示，如果块 B_3 不执行，B_5 中赋给 j 的值会有所不同。例如，如果第一次进入 B_2 时 u = 30且 v = 25，因为从没进入过 B_3，所以图10-36a在 B_5 将 j 置为1，而图10-36b却将 j 置为2。 □

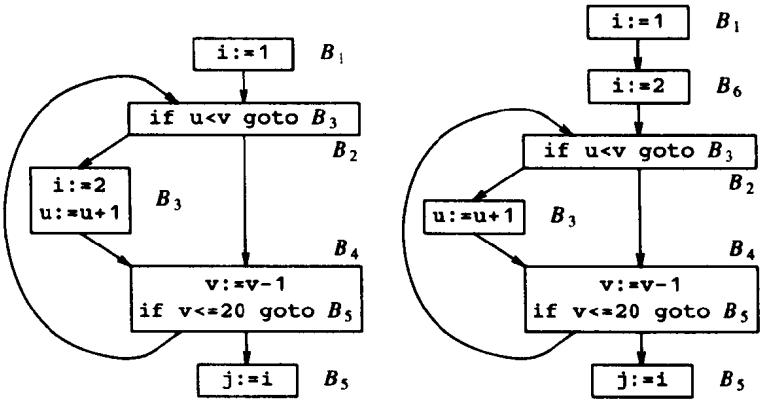


图10-36 非法代码外提的例子

a) 变换前 b) 变换后

例10.21 如果循环中对x不止一次进行赋值，则条件2是必需的。例如，图10-37的流图结构同图10-36a的一样，我们也像图10-36b那样建立前置首节点 B_6。

因为图10-37中的块 B_2 支配循环的惟一出口块 B_4，条件1不能阻止我们把 i := 3外提到前置首节点 B_6。但是，如果我们这样做，只要块 B_3 执行，就会将i的值置为2，于是到达块 B_5 时i的值将为2，即便我们是沿 $B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2 \rightarrow B_4 \rightarrow B_5$ 这样的路径前进的。例如，考虑首次到达B_2 时 v=22且u=21的情形，如果 i := 3在 B_2 中，则在 B_5 中将把j置为3，然而，如果将 i := 3外提到前置首节点，j 将被置为2。 □

例10.22 现在考虑条件3。图10-38中块 B_4 中的 i引用可以由块 B_1 的 i := 1到达，也可以由块B_3 的 i := 2到达，所以不能将 i := 2外提到前置首节点。因为若 u >= v，到达块 B_5 的 k 值会改变。例如，如果 u = v = 0，则图10-38的流图将把 k 置为1，但如果将 i := 2外提到前
640 置首节点，k 将被置为2。 □

算法10.8 代码外提。

输入：带有ud链和支配节点信息的循环 L。

输出：循环的修正版本，增加了前置首节点，而且（可能）有一些语句外提到前置首节点。

方法：

1. 用算法10.7寻找循环不变语句。

2. 对步骤1中找到的每个定义 x 的语句 s，检查以下几项：

i) s 所在的块支配 L 的所有出口。

ii) x 在 L 的其他地方没有被定义。

iii) L 中所有 x 的引用只能由 s 中 x 的定义到达。

3. 按算法10.7找出的次序，把步骤1中找出的满足步骤2中3个条件的每条语句 s 移到新创建的前置首节点。但前提是：在 L 中被定义的 s 的任何运算对象（由算法10.7的步骤3找出这种 s）已经将其定义语句移到前置首节点。 □

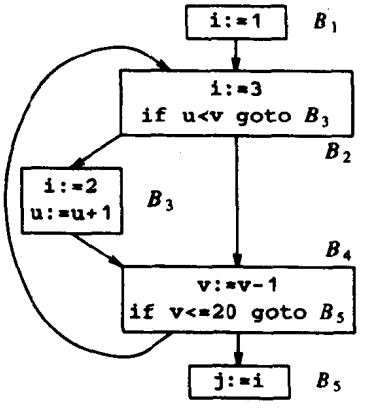


图10-37 说明条件2的例子

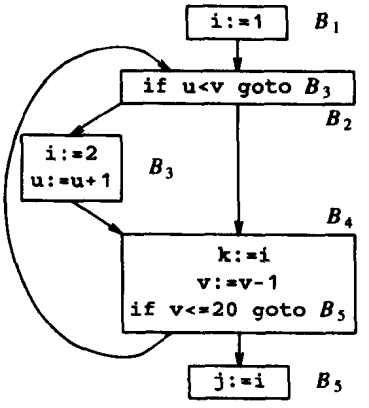


图10-38 说明条件3的例子

为了理解为什么没有改变程序所计算的内容，算法10.8的条件2i和2ii保证在 s 中计算的 x 值必定是 L 任何出口之后的 x 值，当把 s 外提到前置首节点时，s 仍然是到达 L 任何出口的 x 的定义。条件2iii保证 L 中任何 x 的引用在外提前后都将引用 s 计算的 x 值。 641

为了明白为什么变换不会增加程序的运行时间，我们只需注意条件2i即可，它保证控制每次进入循环 L 时，s 至少执行一次。代码外提后，它在前置首节点中仅执行一次，而且控制进入 L 时，它根本不再执行。

10.7.5 可选的代码外提方案

如果我们愿冒风险，即代码外提实际上可能增加一点程序的运行时间，那么我们可以稍微放宽条件1；当然，我们决不能改变程序所计算的内容。代码外提条件1（即算法10.8中的2i项）的宽松版本是仅当下列条件成立时，我们可以将定义x的语句 s 外提：

1′. 包含 s 的块支配循环的所有出口，或者x在循环之外没有被引用。例如，如果x是一个临时变量，我们可以确定（在许多编译器中）它的值只能在它自己的块中被引用。通常，需要通过活跃变量分析来判断x在循环的任何出口是否是活跃的。

如果使用条件1′修改算法10.8，有时候运行时间会有少量的增加，但是我们可以期望得到比较好的平均时间。修改后的算法可以将某些循环中不被执行的计算移到前置首节点中。但是这种冒险不仅会显著减慢程序的运行速度，还可能会在特定的环境下引发错误。譬如，循环中除法x/y的计算可能跟在 y=0 是否成立的测试之后，所以如果我们将x/y移到前置首节点，可能就会出现被0除的错误。为此，除非优化受到程序员的约束或者我们在除语句中使用条件1，

否则使用条件1′是不明智的。

对于算法10.8中的条件2i、2ii和2iii，即使赋值语句 x := y+z 均不满足它们，我们仍然可以将 y+z 移到循环之外。创建一个新的临时变量t，并在前置首节点中置 t := y+z，然后在循环中用 x := t 替换 x := y+z。在很多情况下，我们能够传播循环中的复制语句 x := t，如同本节前面所讨论的。注意，如果满足算法10.8的条件2iii，也就是说，如果循环 L 中所有 x 的引用都是在 x := y+z（现在是 x := t）中定义的，那么，通过用 t 的引用替换 L 中 x 的引用，并将 x := t 放在循环的每一个出口之后，我们就可以删除语句 x := t。

10.7.6 代码外提后对数据流信息的维护

算法10.8的变换没有改变ud链接信息，因为根据条件2i、2ii和2iii，由外提语句 s 赋值的变量的所有引用，原来能由 s 到达的，现在仍能由新位置的 s 到达。s 引用的变量的定义或者在循环 L 外（这种情况下它们到达前置首节点）或者在循环L内（这种情况下，由步骤3，它们已
642 移到 s 之前的前置首节点中）。

如果ud链由指向语句的指针的指针表（而不是指向语句的指针表）来表示，则ud链的维护将非常简单。在外提语句 s 时，只要修改指向 s 的指针即可。也就是说，为每个语句 s 建一个指针 p_s，它总是指向 s，把 p_s 放在每个包含 s 的ud链中，然后，不管 s 移到哪里，我们只要改变 p_s 即可，而不管现在有多少个含 s 的ud链。当然，多一层间接寻址会增加编译器的时间和空间开销。

如果我们用语句地址（指向语句的指针）表来表示ud链，在外提语句时仍能维护ud链。但是，若为提高效率，我们需要用du链。在外提s时，我们可顺着它的du链，改变所有指向 s 的引用的ud链。

代码外提时，支配节点信息略有改变。前置首节点现在是首节点的直接支配节点，而前置首节点的直接支配节点是原先首节点的直接支配节点。也就是说，前置首节点作为首节点的父节点被插入到支配树中。

10.7.7 归纳变量删除

在循环 L 中，若变量 x 值的每次改变都是增加或减少某个固定的常数，那么x称为循环 L 的*归纳变量*。通常，每次环绕循环，归纳变量增加相同的常数值，例如以 **for** i := 1 **to** 10开头的循环中的 i。我们的方法将处理当我们环绕循环时会增加或减少0次、1次、2次或多次的变量。归纳变量改变的次数甚至会随不同的迭代而有所不同。

归纳变量的一种常见情况是作为数组的下标，例如 i；某些其他的归纳变量（如 t，它的值是 i 的线性函数）是用于访问数组的实际偏移。i 经常仅用于测试循环的终止，因此用对某个 t 的测试代替它，我们就可以去掉 i。

为便于表示，下面的算法处理受限制的一类归纳变量。通过增加归纳变量的种类可以扩展该算法，但需要证明那些关于含有通用算术运算符表达式的定理。

如果循环中的变量 i只有惟一形如 $i := i \pm c$ 的赋值，其中 c 是常量，那么 i 就是我们要找的循环的*基本归纳变量*[⊖]。然后再寻找其他的归纳变量j，它仅在 L 中定义一次，而且其值
643 是某个基本归纳变量i的线性函数。

算法10.9 归纳变量检测。

输入：带有到达定义信息和循环不变计算信息（由算法10.7得到）的循环 L。

⊖ 在我们关于归纳变量的讨论中，“+”只代表加法运算符而不是一般运算符，其他标准的算术运算符也是如此。

输出：一组归纳变量。与每个归纳变量j相关联的是三元组 (i, *c*, *d*)，其中i是基本归纳变量，*c* 和 *d* 是常量，在 j 的定义点，j 的值由 *c**i+*d* 给出。我们称 j 属于 i 族，基本归纳变量 i 也属于它自己的族。

方法：

1. 扫描 *L* 的语句，找出所有的基本归纳变量。在此，我们要用到循环不变计算的信息。与每个基本归纳变量 i 相关联的三元组是(i, 1, 0)。

2. 寻找 *L* 中只有一次赋值的变量k，它具有下面的形式之一：

k:=j*b, **k:=b*j**, **k:=j/b**, **k:=j±b**, **k:=b±j**

其中，*b* 是常数，j 是基本的或非基本的归纳变量。

如果 j 是基本归纳变量，则 k 在 j 族中。k 的三元组依赖于定义它的指令。例如，如果 k 由 k := j**b* 定义，那么 k 的三元组是 (j, *b*, 0)。其余情况的三元组可以类似地定义。

如果j不是基本归纳变量，令 j 属于 i 族，那么我们附加的要求是：

(a) 在循环 *L* 中对 j 的惟一赋值和对 k 的赋值之间没有对 i 的赋值。

(b) 循环 *L* 外没有j的定义可到达 k。

常见的情况是 k 和 j 属于同一块中的临时变量，它们比较容易检查。通常，如果我们分析 *L* 的流图以确定哪些块（哪些定义）位于对j和对k赋值之间的路径上，那么到达定义信息将提供我们需要的这种检查。

我们从j的三元组 (i, *c*, *d*) 和定义k的指令来计算k的三元组。例如，定义 k := *b**j 导致 k 的三元组为 (i, *b***c*, *b***d*)。注意，*b***c* 和 *b***d* 可以在分析过程中完成计算，因为 *b*，*c* 和 *d* 都是常量。 □

一旦找出归纳变量族，我们就可以修改计算归纳变量的指令，改用加或减而不是乘。这种用较快指令代替较慢指令的变换称为**强度削弱**。 644

例10.23 图10-39a中由基本块B_2构成的循环中含有基本归纳变量 i，因为循环中对 i 惟一的赋值就是将 i 加1。i 族含有 t_2，因为对t_2只有一次赋值，其右部为 4*i，于是 t_2 的三元组是 (i, 4, 0)。同样地，在由块 B_3 构成的循环中，j 是仅有的基本归纳变量，t_4 属于 j 族，其三元组为 (j, 4, 0)。

我们还可以寻找以块 B_2 为首节点包括 B_2, B_3, B_4 和 B_5 的外循环中的归纳变量，i和j都是这个较大循环中的基本归纳变量；t_2 和 t_4 也是归纳变量，其三元组分别是 (i, 4, 0)和 (j, 4, 0)。

图10-39b的流图是将下面的算法应用到图10-39a后得到的。下面我们将讨论该变换。 □

算法10.10 用于归纳变量的强度削弱。

输入：循环 *L*，附带有到达定义信息和由算法10.9算出的归纳变量族。

输出：修正后的循环。

方法：依次考虑每个基本归纳变量i。对每个三元组为 (i, *c*, *d*) 的i族中的归纳变量j，执行如下步骤：

1. 建立新变量s，但如果变量j_1和j_2具有同样的三元组，则只为它们建立一个新变量。

2. 用j := s代替对j的赋值。

3. 在 *L* 中紧跟在每个赋值语句 i := i+*n* 之后（*n* 是常量），添加上如下语句：

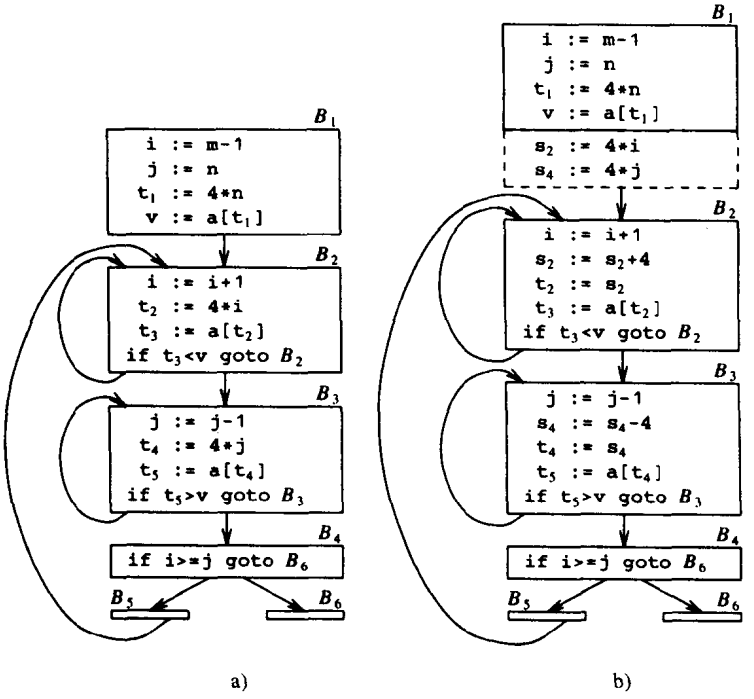
s := s + c*n

其中；表达式 $c*n$ 的计算结果为一个常数，因为 c 和 n 是常数。将 s 放入 i 族，其三元组为 (i, c, d)。

4. 还必须保证在循环的入口处 s 的初值为 $c*i + d$，该初始化可以放在前置首节点的末尾，初始化由下面两个语句组成：

```
s := c*i     /* 如果c为1，则为 s := i */
s := s+d     /* 如果d为0则省略 */
```

注意，s 是 i 族的归纳变量。 □



a) b)

图10-39 强度削弱

a) 变换前 b) 变换后

例10.24 假设我们从里向外考虑图10-39a中的循环。因为内循环 B_2 和 B_3 的处理非常类似，所以我们仅考虑环绕 B_3 的循环。在例10.23中，我们发现在围绕 B_3 的循环中，j 是其基本归纳变量，t_4 是另一个归纳变量，其三元组为 (j, 4, 0)。在算法10.10的步骤1中，构造了一个新变量 s_4。在步骤2中，赋值语句 t_4 := 4*j被 t_4 := s_4 代替。步骤4将赋值语句 s_4 := s_4-4 插在赋值语句j := j-1的后面，其中，-4是由-1乘4得到的，-1是对j 的赋值中的-1，而4是 t_4 的三元组 (j, 4, 0) 中的4。

因为块 B_1 充当循环的前置首节点，所以我们可以将对 s_4 的初始化放在块 B_1 的末尾，而块 B_1 含有j的定义。增加的指令放在块 B_1 中用虚线扩展的部分。

当考虑外循环时，流图如图10-39b所示。变量i，s_2，j和 s_4 都可以看作是归纳变量，但算法10.10的步骤3已把新建变量分别加入了i族和j族。为了完成i和j的删除，我们需要使用下一个算法。 □

645
~
646

强度削弱后，我们发现有些归纳变量只是用于测试，可以用对其他归纳变量的测试代替对

这种归纳变量的测试。例如，若i和t是归纳变量，t的值总是i的4倍，那么测试i>=j就等价于t>=4*j，替换后有可能删除i。但如果t=-4*i，则需同时改变关系运算符，因为i>=j等价于t<=-4*j。在下面的算法中，我们将只考虑乘数为正的情况，而将算法推广到适用于负数的情况留作练习。

算法10.11 归纳变量删除。

输入：带有到达定义信息、循环不变计算信息（从算法10.7得到）和活跃变量信息的循环 L。

输出：修正后的循环。

方法：

1. 考虑每个仅用于计算同族中和条件分支中其他归纳变量的基本归纳变量i。取i族的某个j，优先取其三元组 (i, c, d) 中的 c 和 d 尽可能简单的j（即优先考虑 $c=1$ 和 $d=0$ 的情况），把每个含i的测试改成用j代替i。我们假定下面的c是正的。用下面的语句来代替形如if i **relop** x goto B 的测试，其中x不是归纳变量：

```
r := c*x          /* 如果c等于1，则为 r := x */
r := r+d          /* 如果d为0则省略 */
if j relop r goto B
```

其中r是新的临时变量。对if x **relop** i goto B 的处理与此类似。如果测试if i_1 **relop** i_2 goto B 中的 i_1 和 i_2 都是归纳变量，则检查 i_1 和 i_2 是否都能被代替。最简单的情况是我们有三元组为 (i_1, c_1, d_1) 的 j_1 和三元组为 (i_2, c_2, d_2) 的 j_2，并且 $c_1=c_2$，$d_1=d_2$，那么，i_1 **relop** i_2 等价于 j_1 **relop** j_2。在更复杂的情况下，测试的替换可能是没有价值的，因为我们可能要引入两步乘和一步加，而删除 i_1 和 i_2 只能节省两步。

最后，因为被删除的归纳变量已经没有什么用处，所以从循环L中删除所有对它们的赋值。

2. 现在考虑由算法10.10为其引入的语句 j := s 的每个归纳变量 j。首先检查在引入的j := s 和任何j的引用之间有没有对s的赋值，应该是没有。一般情况下，j 在定义它的块中被引用，这可以简化该检查；否则，需要用到达定义信息，并加上一些对图的分析来实现这种检查。然后用对 s 的引用代替所有对j的引用，并删除语句 j := s。 □ 647

例10.25 考虑图10-39b的流图。环绕 B_2 的内循环包含两个归纳变量i和 s_2，但是一个也不能被删除，因为 s_2 是数组 a 的下标，而i被用于循环外的测试。同样，环绕 B_3 的循环包含归纳变量 j 和 s_4，但是它们也都不能被删除。

再让我们把算法10.11用于外循环。当算法10.10建立新变量 s_2 和 s_4 时，s_2 被置于i族，s_4 被置于j族，如例10.24所讨论的那样。考虑i族，i 的惟一用途是在块 B_4 中测试循环是否终止，所以i是在算法10.11的步骤1中要被删除的候选。块 B_4 的测试包含两个归纳变量 i 和 j，幸好，分别包含在 i 族和 j 族中的归纳变量 s_2 和 s_4 的三元组具有同样的常数，因为这两个三元组分别是 $(i, 4, 0)$ 和 $(j, 4, 0)$，于是，测试 i>=j 可由 s_2>=s_4 代替，使得 i 和 j 都可以被删除。

算法10.11的步骤2将复制传播用于新建的变量，用 s_2 和 s_4 分别代替了 t_2 和 t_4。 □

10.7.8 带有循环不变表达式的归纳变量

在算法10.9和算法10.10中，我们允许用循环不变表达式代替常量，但是归纳变量j的三元组 (i, c, d) 中可能包含循环不变表达式而不是常量。这些表达式的计算应该在循环L外的前置首节点中完成。而且，因为中间代码要求每个语句至多含有一个运算符，因此我们必须准备为这种表达式的计算生成中间代码语句。算法10.11中测试的替换需要知道乘法常量 c 的正负

号，为此，我们应该把注意力集中在 c 是已知常量的那些情况。

10.8 处理别名

如果两个或多个表达式表示相同的内存地址，我们称这些表达式互为*别名*。指针和过程都有可能引入别名，本节讨论包含指针和过程的数据流分析。

因为指针会引起定义和引用的不确定性，使得数据流分析变得更加复杂。如果我们不知道指针p指向何处，一种安全的做法是假定指针的间接赋值会潜在地改变（即定义）任何变量。我们还必须假设，对指针所指向数据的引用，比如 x := *p，潜在地可能引用任何变量。这些
[648] 假设产生了更多的活跃变量和到达定义，以及更少的可用表达式。幸运的是，可以利用数据流分析约束指针的指向，从而在其他数据流分析中得到更多的正确信息。

和指针变量赋值相似，在进行过程调用时，如果能计算出过程中可能发生改变的变量的集合，就不必假设所有变量都可能发生改变。在所有的代码优化中，我们仍会犯保守性错误。也就是说，可能被改变或引用的变量的集合完全包含了在程序执行中实际被改变或引用的变量。通常，我们只需努力接近真正被改变或引用的变量的集合，而不必过分费力或错误地改变程序行为。

10.8.1 一种简单的指针语言

特殊地，让我们考虑一种只具有基本数据类型（如整型和实型）和数组的语言，每种基本数据类型占用一个字。此外，该语言包含指向基本数据类型和数组的指针，但不包含指向其他指针的指针。对于指向数组的指针，只需知道指针p指向数组a的某处即可，而不必关心p指向a的哪个特定元素。考虑使用指针的目的，将数组的所有元素聚集在一起是合理的。典型地，指针在整个数组中可以当作游标使用，如果我们能实现一个更详细的数据流分析，应该在程序的特定点上指出p可能指向a的哪一个元素。

我们还必须假设指针上的算术操作在语义上是有意义的。首先，如果指针p指向基本数据元素，那么p上的任何算术操作将产生一个非指针类型的值，这个值可以是整型。如果p指向数组，那么p加减一个整数将使其指向同一数组的其他元素，而指针上的其他算术操作将产生一个非指针值。尽管不是所有的语言都禁止通过增加指针的值将指针从一个数组a转移到另一个数组 b 上，但这样的操作依赖于数组b在存储器中必须紧跟在a的后面。在我们的观点中，在决定执行何种优化时，一个优化编译器只需关注语言定义。但是，每个编译器实现者都必须判定编译器应该执行何种特定的代码优化。

10.8.2 指针赋值的作用

在这些假设下，只有被声明为指针的变量，以及通过指针加/减一个常数得到的临时变量，
[649] 才能被用作指针变量。将所有这样的变量称为指针。确定指针p指向何处的规则如下：

1. 如果存在赋值语句 s：p := &a，那么紧跟 s 之后，p 只能指向 a。如果 a 是一个数组，那么在对 p 进行形如 p := &a ± c （c为常量）的赋值之后，p 只能指向 a。[⊖]通常，我们认为&a指向数组a的第一个元素的位置。

2. 如果存在赋值语句 s：p := q ± c，其中 c 是非零整数，p 和 q 是指针，那么紧跟 s 之后，p可以指向 s 之前 q 指向的任何数组，但不能指向其他变量。

3. 如果存在赋值语句 s：p := q，那么紧跟s之后，p 可以指向 s 之前 q 指向的任何变量。

⊖ 在本节中，+代表它本身而不是一般运算符。

4. 如果对 p 做其他赋值，p 不能指向任何对象，这样的赋值可能没有意义（依赖于语言的语义）。

5. 对 p 以外的变量进行赋值后，p 仍然指向该赋值语句前所指向的变量。注意，该规则假设指针不能指向指针。放宽该假设不会使事情变得特别复杂，我们把对它的推广留给读者。

对于块 B，定义 $in[B]$函数：对每个指针 p，该函数给出在块 B 开始点 p 指向的变量的集合。形式上，$in[B]$ 是序对 (p, a) 的集合，其中 p 是一个指针，a 是一个变量，序对 (p, a)表示 p 指向a。实际上，$in[B]$ 可以表示为指针的列表，指针 p 的列表给出使（p, a）在 $in[B]$中的 a 的集合。类似地，在块 B 结束点定义了 $out[B]$。

定义转换函数 $trans_B$ 来说明块 B 对变量的影响。函数 $trans_B$ 把序对集合S作为参数并产生另一个集合 T。每个序对形如 (p, a)，其中 p 是一个指针，a 是一个非指针变量。可以推测，$trans_B$ 是应用到集合 $in[B]$上，并产生结果集合 $out[B]$。我们只需给出计算单条语句的 $trans$ 结果集合的方法，然后将块 B 的每条语句 s 的 $trans_s$结果集合组合起来就形成了块 B 的 $trans_B$。

计算 $trans$ 的规则如下：

1. 如果a是一个数组，语句 s 是p := &a或者p := &a ± c，那么

$trans_s(S) = (S - \{(p, b) \mid b\text{是任意变量}\}) \cup \{(p, a)\}$

2. 如果s是p := q ± c，其中q是指针，c是非零整数，那么

$trans_s(S) = (S - \{(p, b) \mid b\text{是任意变量}\}) \cup \{(p, b) \mid (q, b)\text{在}S\text{中，}b\text{是数组变量}\}$

这条规则在 p=q 时也是有意义的。

3. 如果 s 是p := q，那么 650

$trans_s(S) = (S - \{(p, b) \mid b\text{是任意变量}\}) \cup \{(p, b) \mid (q, b)\text{在}S\text{中}\}$

4. 如果 s 将任何其他表达式的值赋给指针 p，那么

$trans_s(S) = S - \{(p, b) \mid b\text{是任意变量}\}$

5. 如果 s 不是给指针赋值，那么 $trans_s(S) = S$。

现在，我们写出 in，out 和 $trans$ 的方程式如下：

$$out[B] = trans_B(in[B])$$
$$in[B] = \bigcup_{B\text{的前驱}P} out[P] \qquad (10\text{-}13)$$

其中，如果 B 由语句 s_1，s_2，…，s_k组成，则

$$trans_B(S) = trans_{s_k}(trans_{s_{k-1}}(\cdots(trans_{s_2}(trans_{s_1}(S)))\cdots))$$

方程（10-13）可以按照类似于算法10.2中的到达定义进行求解。因此，下面我们不介绍算法的细节，而是用一个实例解释我们的方法。

例10.26 考虑图10-40中的流图。假设a是一个数组，c是一个整数，p和q是指针。开始，我们置 $in[B_1]$为 ∅。然后，$trans_{B_1}$删除所有第一分量为 q 的序对，将序对 (q, c) 增加到结果集中。也就是说，q被声明为指向c的指

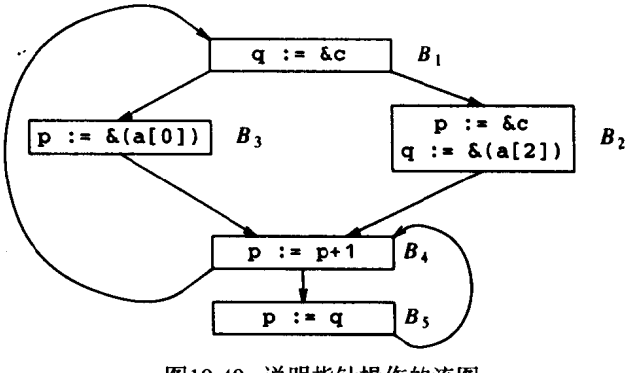


图10-40 说明指针操作的流图

针。于是有：

$$out[B_1] = trans_{B_1}(\varnothing) = \{(q, c)\}$$

然后令 $in[B_2] = out[B_1]$。p := &c的作用是用序对 (p, c) 取代所有第一分量为p的序对。q := &(a[2])的作用是用 (q, a) 取代所有第一分量为q的序对。注意，q := &(a[2])实际上是形如q := &a+c的赋值，c为常量。现在我们可以计算得到：

$$out[B_2] = trans_{B_2}(\{(q, c)\}) = \{(p, c), (q, a)\}$$

同样地，令 $in[B_3] = \{(q, c)\}$，计算得到 $out[B_3] = \{(p, a), (q, c)\}$。

接下来，我们发现 $in[B_4] = out[B_2] \cup out[B_3] \cup out[B_5]$。可以推测，$out[B_5]$开始为 $\varnothing$，并且在这一遍中没有被改变。但是 $out[B_2] = \{(p, c), (q, a)\}$，$out[B_3] = \{(p, a), (q, c)\}$，于是：

$$in[B_4] = \{(p,a), (p,c), (q,a), (q,c)\}$$

在 B_4 中的 p := p+1 使得 p 必须指向一个数组，即：

$$out[B_4] = trans_{B_4}(in[B_4]) = \{(p, a), (q, a),(q, c)\}$$

注意，无论何时执行 B_2 时，如果在 B_4 中 p := p+1之后间接引用了 p，则 B_2 中将 p 指向 c 的语句会成为一个语义上没有意义的动作。因此，该流图是不“实际的”，但它的确阐明了关于
651 指针的一些推论。

接下来，令 $in[B_5] = out[B_4]$，$trans_{B_5}$复制 q 的目标变量并赋给 p。因为 q 在 $in[B_5]$中可以指向a或者c，所以

$$out[B_5] = \{(p, a), (p, c), (q, a), (q, c)\}$$

在下一遍循环中，我们发现 $in[B_1] = out[B_4]$，所以 $out[B_1] = \{(p, a), (q, c)\}$。这个值也是 $in[B_2]$ 和 $in[B_3]$ 的新值，但这些新值不再改变 $out[B_2]$、$out[B_3]$ 和 $in[B_4]$。于是，我们逼近了想要的答案。 □

10.8.3 利用指针信息

假定 $in[B]$ 是位于块 B 开始点的所有指针指向的变量的集合，并假定在块 B 中引用了指针p。从 $in[B]$开始，对块 B 中引用 p 之前的每条语句 s 应用 $trans_s$。这些计算将指出，在包含重要信息的特定语句上指针 p 指向了何处。

假设已经可以确定每一个指针在用于间接引用（可能出现在赋值号左边，也可能出现在赋值号右边）中时指向哪些变量，那么，对通常的数据流问题，如何利用这些信息以获得更精确的解呢？在每种情况下，我们都应该利用指针信息来保证只犯保守性的错误。为了解怎样做出上述选择，让我们考虑两个例子：到达定义和活跃变量分析。

可以用算法10.2计算到达定义，但还需要知道一个块的 *kill* 和 *gen* 值。通常只对没有通过指针间接赋值的语句才计算 *gen* 值。因为 p 有可能指向任意变量 b，所以间接赋值 *p := a 会
652 产生对任意变量b的定义。这个假设是保守的，因为正如同10.5节所讨论的：假设定义到达一个点而实际上它并没有到达，这种假设通常是保守的。

当计算 *kill* 时，我们假设仅当 b 不是数组而且是p指向的惟一变量时，*p := a 才会注销 b 的定义。如果p指向两个或者多个变量，那么我们假设任何一个变量的定义都没有被注销。我们再次变得保守起来，因为除非能够证明 *p := a 重新定义了 b，否则我们将允许 b 的定义穿越*p = a，到达它们能够到达的任何地方。换句话说，出现疑问时，我们假设定义会到达那里。

可以利用算法10.4处理活跃变量，但必须重新考虑如何定义形如 *p := a 和 a := *p 的语

句的 *def* 和 *use* 。语句 *p := a 只引用了 a 和 p。仅当 b 是 p 可能指向的惟一变量时，我们说它定义了 b。除非 b 的引用确实被 *p := a所阻塞，否则允许b的引用穿越赋值 *p := a。这样，我们永远也不会在 b 活跃的点上声明它是无用的。语句 a := *p 表示 a 的定义，并表示 p 的引用及 p 指向的任何变量的引用。通过最大化可能发生的引用，我们将活跃变量的估计最大化。通过最大化活跃变量，我们通常变得保守。例如，我们会生成一个保存了无用变量的代码，但我们决不会忘记保存任何活跃变量。

10.8.4 过程间的数据流分析

至此，我们所说的程序只是单个过程，因而只有一个流图。现在我们讨论如何从许多相互作用的过程中收集信息。基本思想是，首先确定一个过程如何影响其他过程的 *gen*，*kill*，*use* 和 *def* 信息，然后独立地计算每个过程的数据流信息。

在数据流分析期间，我们必须处理过程调用时由参数所建立的别名。因为两个全局变量不可能表示相同的内存地址，所以在一对别名中至少有一个是形式参数。形式参数可以被传递给过程，所以两个形式参数有可能互为别名。

例10.27 设有过程p，带有两个形式参数x和y，参数传递方式为传地址方式。在图10-41中，我们看到 b+x在 B_1 和 B_3 中都进行了计算。假设从 B_1 到 B_3 的所有路径都经过 B_2，而且在所有的路径上都不存在对b或x进行赋值的语句，那么 b+x 在 B_3 是否可用呢？答案依赖于x和y是否表示相同的内存地址。例如，可能存在过程调用 p(z, z)，或者是 p(u, v)，其中 u 和 v 是另一个过程 q(u, v)的形式参数，而且可能调用 q(z, z)。

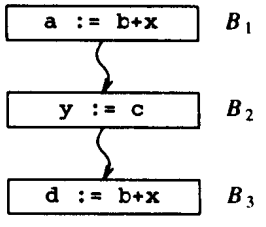


图10-41 别名问题的示例

同样地，如果 x 是 p(x, w)的形式参数，变量 y 的作用域是某个调用了 p（即 p(y, t)）的过程 q，则 x 和 y 可能是别名。更加复杂的情况可能使得 x 和 y 互为别名，我们将要制定一些通用的规则以确定所有这样的别名。 □

可以证明，在某些情况下，不将变量看成互为别名是一种保守的做法。例如，在到达定义中，如果我们希望声称 a 的定义被 b 的定义注销了，我们最好确定只要 b 的定义被执行，a和 b 就一定是别名。其他情况下出现疑问时，将变量看作互为别名的做法也是保守的。例10.27就是这样的情况。如果可用表达式 b+x 没有被 y 的定义注销，我们最好认为 b 和 x 都不能是y的别名。

10.8.5 带有过程调用的代码模型

为了说明如何处理别名，让我们考虑一种允许递归过程的语言，它允许同时引用局部变量和全局变量。一个过程中可用的数据只包括全局变量和它自己的局部变量，也就是说，语言中不存在块结构。参数传递方式为传地址方式。我们要求所有过程的流图都只有一个入口（初始节点）和一个返回节点（它使得控制返回到调用例程）。为方便起见，我们假设过程中的每一个节点都位于从入口到返回节点的某条路径上。

现在，假设在过程 p中，遇到过程调用 q(u, v)。要计算到达定义、可用表达式或任何其他数据流分析，必须知道 q(u, v)是否可能改变某些变量的值。注意，我们说的是“可能”改变，而不是“将要”改变。和所有数据流问题一样，确切地知道一个变量的值是否发生变化是不可能的，只能找出一个变量集合，其中的某些变量的确发生了变化，有些变量则没有。通过小心地削减后一类变量，可以获得和事实最相近的近似变量集合，而且尽量只犯保守性错误。

653 ~ 654

过程调用 q(u, v)可以定义的变量只有全局变量和变量 u 和 v。变量 u和v 可能是 p 的局

部变量。q 的局部变量定义在过程返回之后不保存任何结果。即使 p=q，也只是 q 的局部变量的拷贝发生了变化，而这些拷贝在调用返回之后将不再出现。要确定 q 所定义的全局变量非常简单，只需查看 q 中定义了哪些变量，或者在 q 引起的过程调用中定义了哪些变量。此外，如果 q 定义了第一个和/或第二个形式参数，或者这些形式参数作为实参被q传递给定义 u 和 v 的过程，u 和/或 v（它们可能是全局变量）将会发生变化。因为变量可能有别名，并不是每个被 q 改变的变量都需要被 q 或被 q 调用的某个过程明确定义。

10.8.6 别名的计算

在给定的过程中哪些变量会发生变化？在回答该问题之前，我们必须设计一个寻找别名的算法。这里采用的是一种简单的方法。我们将概念“是……的别名”形式化为变量上的关系≡，并且求出该关系。在这样做的过程中，尽管区分具有相同标识符但属于不同过程的局部变量，但是，对同一过程的不同调用中出现的同名变量不加区分。

为简单起见，对程序不同点上的别名集合不加区分。如果两个变量可能互为别名，则假设它们总是如此。最后，我们要做一个保守的假设，即≡是传递的，所以变量被分成若干等价类，两个变量互为别名当且仅当它们在同一等价类中。

算法10.12 简单别名计算。

输入：过程和全局变量的集合。

输出：等价关系≡，≡具有如下性质：只要在程序中的某个位置上，x 和 y 互为别名，则 x≡y；反过来不一定成立。

方法：

1. 如果需要，将变量重新命名，使得没有两个过程使用相同的形式参数或者局部变量标识符。也不允许局部变量、形式参数或者全局变量共用同一个标识符。

2. 如果存在过程 $p(x_1, x_2, \cdots, x_n)$ 和对该过程的调用 $p(y_1, y_2, \cdots, y_n)$，则对所有的 i，令 $x_i \equiv y_i$。也就是说，每个形式参数都可以成为相应实参的别名。

655 3. 通过增加以下关系式求出≡在形-实参数对应别名上的自反传递闭包。

a) 如果y≡x，则x≡y。

b) 对任意y，如果x≡y且y≡z，则x≡z。 □

例10.28 考虑图10-42给出的三个过程框架。其中，参数传递采用传地址方式。包含两个全局变量 g 和 h，两个局部变量 i 和 k。i属于过程 main，k 属于过程 two。过程one包含形式参数w和x，过程 two 包含形式参数 y 和 z，过程 main 没有任何形式参数。在本例中，不需要为变量重命名。我们首先计算形-实参数对应的别名。

在过程 main 中，对过程 one 的调用使得 h≡w，i≡x。对过程 two 的第一次调用使得 w≡y，w≡z，第二次调用使得 g≡y，x≡z。

过程 two 对过程 one 的调用使得 k≡w，y≡x。当我们求以≡表示的别名关系的传递闭包时，我们发现在本例中所有的变量都可能是另一个变量的别名。 □

```
global g, h;
    procedure main( );
        local i;
        g := ... ;
        one(h, i)
    end
    procedure one(w, x);
        x := ... ;
        two(w, w);
        two(g, x)
    end;
    procedure two(y, z);
        local k;
        h := ... ;
        one(k, y)
    end
```

图10-42 过程样例

算法10.12中的别名计算通常不会发现例10.28中的这种扩展别名。直觉上，可以预期两个具有不同类型的变量不能互为别名。此外，程序员无疑对其使用的变量存在概念类型。比如，如果过程 p 的第一个形式参数表示速度，可以预期在程序员的眼中，对 p 的任何调用，第一 656
个参数都应该是表示速度的。于是，直观地我们可以期望大多数程序只生成很少的几组别名。

10.8.7 存在过程调用时的数据流分析

作为一个例子，让我们考虑存在过程调用时怎样计算可用表达式，其中参数传递采用传地址方式。正如10.6节所讨论的，我们必须确定什么时候可以定义一个变量，从而注销一个表达式，并且必须确定表达式是何时产生(计算)的。

对每一个过程 p，可以定义集合 *change*[p]，集合中的元素是执行p时可能被改变的全局变量及 p 的形式参数。在这一点上，如果一个变量的别名等价类中的某个成员被改变，不能认为该变量发生了改变。

令 *def* [p]是形式参数和p中显式定义的全局变量（不包括那些 p 所调用的过程中定义的变量）的集合。在 p 调用过程时，用全局变量或 p 的形式参数作为实参进行调用，为了写出 *change* [p] 的方程式，我们只需将 p 调用过程时所用的实参同被调用过程的相应形式参数联系起来。于是可以写出下列方程：

$$change\ [p] = def\ [p] \cup A \cup G \tag{10-14}$$

其中，

1. A = {a | a是全局变量或 p 的形式参数，使得，对于某个过程 q 和整数 i，p 调用 q 时将 a 作为第 i 个实参，并且 q 的第 i 个形式参数在 *change* [q]中}。

2. G = {g | g是 *change*[q]中的全局变量，且p调用了q}。

方程(10-14)可以用迭代法求解。虽然解不惟一，但我们只需求得最小解。通过从一个很小的近似值开始并逐步迭代，可以收敛于该解。最初的很小的近似值显然是 *change*[p] = *def* [p]。迭代的细节留给读者作为练习。

迭代中过程的访问顺序需要考虑。例如，如果过程不是相互递归的，那么可以先访问不调用任何其他过程的过程（至少有一个）。对于这些过程，*change* = *def*。接下来，可以计算只调用了没调用任何其他过程的过程的 *change* 值。既然对所有的q，*change*[q]都可以用(10-14)得到，对下一组过程就可以直接应用(10-14)。

可以通过如下方法使该思想更精确。画出一个调用图，其节点表示过程。如果p调用了q， 657
就有一条从p到q的边[⊖]。无递归过程集合的调用图中没有环路。在这种情况下，每个节点仅被访问一次。

现在给出计算 *change* 的算法。

算法10.13 变化变量的过程间分析。

输入：过程 p_1，p_2，…，p_n。如果调用图是一个无环图，假设仅当 $j<i$ 时 p_i 才调用 p_j。否则，对过程间的调用顺序不做任何假设。

输出：对于每一个过程p，给出 *change*[p]，即那些并非通过别名改变而是被 p显式改变的全局变量和p的形式参数的集合。

方法：

⊖ 在此我们假设没有过程定值变量。它们将使调用图的构造复杂化，因为我们在构造调用图的边时必须确定与过程定值的形参相对应的可能实参。

1. 通过检查，对每个过程p，计算 *def* [p]。
2. 执行图10-43中的程序计算 *change*。 □

```
(1) for 每个过程p do change[p] := def[p];        /* 初始化 */
(2) while change [p]发生变化 do
(3)     for i := 1 to n do
(4)         for p_i 调用的每个过程q do begin
(5)             将change [q]中的所有全局变量添加到change [p_i]中;
(6)             for q的每个形式参数x （第j 个）do
(7)                 if x在change[q]中 then
(8)                     for p_i对q的每次调用do
(9)                         if 本次调用的第j 个实参a是全局变量
                               或p_i的形式参数
(10)                            then 将a添加到change [p_i]中
        end
```

图10-43 计算change的迭代算法

例10.29 再次考虑图10-42。通过检查，*def* [main] = {g}，*def* [one] = {x}，*def* [two] = {h}。这些是 *change* 的初始值。该过程的调用图如图10-44所示。按访问过程的顺序依次处理two，one和main。

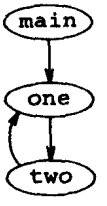


图10-44 调用图

对图10-42中的程序考虑 p_i =two时的情况，在第(4)行q只能是过程one。因为初始时*change* [one] = {x}，在第(5)行没有往 *change* [two]添加任何变量。既然第一个实参是two的局部变量，在第(6)行和第(7)行，我们只需要考虑过程one的第二个形式参数。在惟一的一次two对one的调用中，第二个实参是y，而且相应的形式参数x被改变了，所以在第(10)行，我们置*change* [two] = {h, y}。

现在我们考虑p_i =one时的情况。在第(4)行q只能是过程two。在第(5)行，h是 *change* [two]中的全局变量，所以我们置 *change* [one] = {h, x}。在第(6)行和第(7)行，只有two的第一个形式参数在 *change* [two]中，所以我们必须在第(10)行将g和w添加到 *change* [one]中，它们是调用过程two时的前两个实参。于是，*change* [one] = {g, h, w, x}。

现在考虑 main，过程 one 改变了它所有的形参，所以 h 和 i 在 main 调用 one 的过程中都被改变了。但 i 是局部变量，不需要考虑。因而我们置 *change* [main] = {g, h}。最后，我们重复执行第(2)行的 while 循环，重新考虑 two，我们发现 one 改变了全局变量g。于是，调用 one (k, y)使得g被改变了，所以 *change* [two] = {g, h, y}。迭代时没有再发生变化，故算法终止。 □

10.8.8 change信息的用途

作为使用 *change* 的一个例子，让我们考虑全局公共子表达式的计算。假设我们正在计算过程 p 的可用表达式并且想要计算块 *B* 的*a_kill* [*B*]。我们认为变量 a 的定义会注销含有 a 或含有可能成为 a 的别名的表达式x。但是，除非a是 *change* [q]中某个变量的别名（记住，a 是它自己的别名），否则在 *B* 中对过程 q 的调用不能注销含有 a 的表达式。因而，算法10.12和算法10.13计算出来的信息就可以被用来构造被注销表达式集合的一个安全的近似。

为了计算含有过程调用的程序的可用表达式，我们必须用一种保守的方法来估计过程调用所产生的表达式的集合。为稳妥起见，假定 a+b 由对 q 的调用产生，当且仅当在每一条从 q 的入口

到其返回点的路径上，在a+b之后没有对a或b的重新定义。当寻找a+b表达式的出现时，除非在q的所有调用中，x是a的别名、y是b的别名，否则不能接受x+y作为表达式a+b的出现。

我们之所以提出这样的要求，是因为当一个表达式可用时，却被假定为不可用，是犯了保守性错误。基于同样的考虑，必须假设如果变量z有可能成为a或b的别名，则a+b会被z的定义注销。因而，为所有过程的所有节点计算可用表达式的最简单的方法，就是假设过程调用不会产生任何表达式，而且所有块 B 的 *a_kill* [B]可以按照上面的方法进行计算。我们预期，典型的过程不会产生很多表达式，所以在大多数情况下，该方法是可用的。 658 ~ 659

计算可用表达式的更复杂、更准确的一种方法，是对每一个过程p迭代计算 *gen* [p]。利用上面的方法，可以将 *gen* [p]初始化为在p返回节点的可用表达式的集合。也就是说，产生的表达式不允许有别名，即使有其他变量可能是a或b的别名，a+b也只代表它自己。

现在再次为所有过程的所有节点计算可用表达式。但是，对q(a, b)的调用产生了新的表达式，即 *gen* [q]中将a和b用q的相应形式参数代替后产生的表达式。*a_kill* 的计算和前面相同。对于每个过程p，*gen* [p]的新值可以通过查看在p的返回点上的可用表达式来得到。该迭代一直进行到任何节点中的可用表达式都不再变化为止。

10.9 结构化流图的数据流分析

不含goto的程序具有可约流图，很多程序设计方法学鼓励程序具有可约流图。对多种程序的研究显示，事实上人们编写的所有程序都具有可约流图[⊖]。该观察结果同程序优化有关，因为在可约流图上我们可以找到执行速度非常快的优化算法。本节我们将讨论一些流图的概念，例如与结构化流图有关的“区间分析”。本质上，我们要把10.5节中的语法制导技术应用到更加一般的情况中，即语法不必提供结构但流图可以提供。

10.9.1 深度优先搜索

一种很有用的流图节点排序方法，称为*深度优先排序*，它是2.3节介绍的对树的深度优先遍历的推广。深度优先排序可以用来检测流图中的循环，还有助于提高10.6节中所讨论的迭代数据流算法的速度。深度优先排序从初始节点开始，然后搜索整个图，并尽可能快地访问离初始节点远的节点（*深度优先*）。搜索的路线形成一棵树。在给出算法之前，让我们考虑一个例子。 660

例10.30 对图10-45中流图的深度优先搜索如图10-46所示。实线边形成一棵树，虚线边表

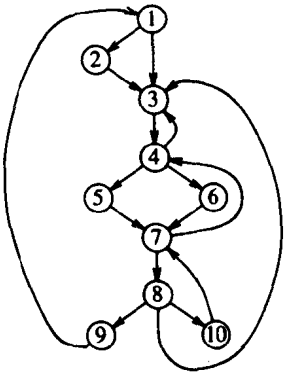


图10-45 流图

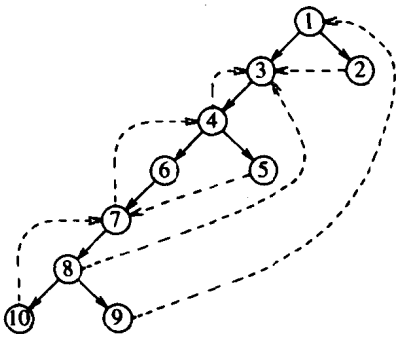


图10-46 深度优先表示

⊖ “人们所编写的”不是多余的，因为我们知道许多程序生成带有goto的“鼠窝图”的代码，但这没有什么关系，这种结构包含在这些程序的输入中。

示流图的其他边。对流图的深度优先搜索相当于对树的前序遍历，1→3→4→6→7→8→10，然后返回8，再到9。再一次返回8，回退到7，6，4，然后到5。再从5退回到4，再返回到3和1。从1到2，再从2返回1，这样我们就对整个树进行了前序遍历。注意，我们并没有解释这棵树是怎样从流图中选出来的。 □

节点的深度优先顺序和我们在前序遍历中各节点最后一次被访问的顺序相反。

例10.31 在例10.30中，遍历树时访问节点的完整序列是：

1, 3, 4, 6, 7, 8, 10, 8, 9, 8, 7, 6, 4, 5, 4, 3, 1, 2, 1

在该列表中，标注出每个数字的最后一次出现，可以得到

1, 3, 4, 6, 7, 8, 10, 8, 9, 8, 7, 6, 4, 5, 4, 3, 1, 2, 1

深度优先顺序是标注的序列的相反顺序。在此，这个序列恰好是1，2，…，10。也就是说，一开始节点是按深度优先顺序编号的。 □

通过构造并遍历一棵以初始节点为根的树，并设法让树中的路径尽可能长，我们可以给出一个计算流图深度优先顺序的算法。这样的树称为深度优先生成树（*dfst*）。下面就是用来从图
661 10-45构造图10-46的算法。

算法10.14 深度优先生成树和深度优先排序。

输入：流图G。

输出：G的*dfst* T和G的节点的深度优先顺序。

方法：使用图10-47中的递归过程 *search* (n)，该算法将 G 的所有节点初始化为“未访问”，然后调用 *search* (n_0)，其中 n_0 是初始节点。当我们调用 *search* (n)时，我们首先将 n 标记为“已访问”，以免将 n 添加到树中两次。用变量 i 记录 G 的节点数，并递减到1，搜索时将深度优先编号 *dfn* [n]赋给节点 n。边 T 的集合形成 G 的深度优先生成树，它们被称为树边。 □

```
    procedure search (n);
    begin
(1)     将n标记为“已访问”;
(2)     for n 的每个后继s do
(3)         if s 标记为“未访问” then begin
(4)             将边n → s添加到T;
(5)             search(s)
            end;
(6)     dfn[n] := i;
(7)     i := i - 1
    end;

    /* 下面是主程序 */
(8) T := 空;      /* 边集 */
(9) for G 的每个节点 do 将n 标记为“未访问”;
(10) i := G 的节点数;
(11) search(n0)
```

图10-47 深度优先搜索算法

例10.32 考虑图10-47。将 i 置为10并调用 *search* (1)。在 *search* 的第(2)行，我们必须考虑节点1的每个后继。假设我们首先考虑 s = 3。然后我们将边1→3添加到树中并调用 *search* (3)。在 *search* (3)中，我们将边3→4添加到 T 中并调用 *search* (4)。

假设在 *search* (4)中，我们首先选择 s = 6。然后将边4→6添加到T 中并调用 *search* (6)。这又使我们将6→7添加到T 中并调用 *search* (7)。节点7有两个后继，4和8。但是4已经被 *serach* (4)标记为“已访问”，所以s = 4时什么也不做。s = 8时我们将7→8添加到 T 中并调用 *search* (8)。假设接下来我们选择 s = 10，将边8→10添加到树中并调用 *search* (10)。

现在10有一个后继7，但是7已经被标记为“已访问”，所以在 *search* (10)中，我们向下跳

到图10-47中的步骤(6)，置 *dfn* [10] = 10，*i* = 9。这将结束对 *search* (10)的调用，所以我们返回 662
到 *search*(8)。现在我们在 *search* (8)中置*s* = 9，将边8→9添加到 *T* 中并调用 *search* (9)。9惟一的后继节点1已经被标记为“已访问”，所以我们置 *dfn* [9] = 9，*i* = 8。然后我们返回到 *search* (8)。8的最后一个后继3已经标记为“已访问”，所以 *s* = 3时什么也不做，在这一点上，我们已经考虑了8的所有后继节点，所以我们置 *dfn* [8] = 8，*i* = 7，并返回到 *search* (7)。

7的所有后继都已经被考虑过，所以我们置*dfn* [7] = 7，*i* = 6，并返回到 *search* (6)。同样地，6的后继都已经被考虑过，所以我们置 *dfn* [6] = 6，*i* = 5，并返回到 *search* (4)中。4的后继3已经被访问过，但是5还没有被访问，所以我们将4→5添加到*T* 中并调用 *search* (5)，这个过程没有任何调用，因为5的后继7已经被标记为“已访问”。从而，*dfn* [5] = 5，将 *i* 置为4并返回到 *search* (4)。我们已经考虑了4的全部后继，因此置 *dfn* [4] = 4，*i* = 3并返回到 *search* (3)，然后我们置 *dfn* [3] = 3，*i* = 2并返回到 *search* (1)。

最后的步骤是在 *search* (1)中调用 *search* (2)，置 *dfn* [2] = 2，*i* = 1并返回到 *search* (1)，置 *dfn* [1] = 1，*i* = 0。注意，我们选择的节点编号正好使 *dfn* [*i*] = *i*，但是对于任意的图，不必满足该关系，甚至图10-45中图的其他深度优先顺序也不必满足该关系。 □

10.9.2 流图的深度优先表示中的边

当我们构造流图的 *dfst* 时，流图的边可以分为以下3类：

1. 树中有从节点 *m* 到 *m* 的祖先（可能到 *m* 自身）的边。我们称这些边为后退（retreating）边。如7→4和9→1就是图10-46中的后退边。有趣且有用的是，如果流图是可约的，则后退边正好是流图中的回边[⊖]，与图10-47中步骤2访问后继的顺序无关。对于任意的流图，每个回边都是后退边，尽管如果这个图是不可约的，将会存在一些不是回边的后退边。

2. 树中有从节点 *m* 到 *m* 的后代的边，称为前进边。*dfst* 中所有的边都是前进边。图10-46中没有其他的前进边，但是如果4→8是一条边，它将属于这类边。

3. 在 *dfst* 中 *m* 和 *n* 互相不是祖先，我们称边 *m*→*n* 为交叉边。图10-46中，边2→3和5→7 663
就是这样的边。交叉边具有一个重要特性：如果我们画 *dfst* 时，节点的儿子是从左到右按照它们加入树的顺序画出的，那么所有的交叉边都是从右到左进行遍历的。

应该注意的是，*m*→*n* 是一条后退边当且仅当 *dfn* [*m*]≥*dfn* [*n*]。在 *dfst* 中，如果 *m* 是 *n* 的后代，那么 *search* (*m*)将在 *search* (*n*)之前终止，所以 *dfn* [*m*]≥*dfn*[*n*]。相反，如果 *dfn*[*m*]≥*dfn*[*n*]，那么 *search* (*m*)将在 *search* (*n*)之前终止，或者 *m* = *n*。但是如果存在边 *m*→*n*，或者在 *dfst*中 *n* 是 *m* 的后继这个事实使得 *n* 成为 *m* 的后代，那么 *search* (*n*)一定在 *search* (*m*)之前开始。因此 *search* (*m*)活跃的时间是 *search* (*n*)活跃的时间的子区间，因此在 *dfst* 中 *n* 是 *m* 的祖先。

10.9.3 流图的深度

流图有一个重要的参数称为深度。给定一个图的深度优先生成树，深度是无环路径上后退边的最大数目。

例10.33 在图10-46中，深度是3，因为路径10→7→4→3带有3条回退边，而且没有无环路径带有4条或者更多的回退边。碰巧的时，这里“最深的”路径上只有回退边，通常最深路径可能同时带有后退边、前进边和交叉边。 □

我们可以证明深度永远不会大于流图中循环嵌套的深度。如果流图是可约的，我们在定义

⊖ 回想一下，回边是那些头支配尾的边。

深度的时候可以用回边代替后退边，因为在 *dfst* 中，后退边正好就是回边。深度的概念和实际选择的 *dfst* 无关。

10.9.4 区间

将流图分解为区间（interval）可以在流图上添加层次结构，而该结构允许我们应用10.5节提出的用于语法制导数据流分析的规则。

直观地，流图中的“区间”是自然循环加上在该循环节点上的无环路结构。区间的一个重要性质是具有支配区间中所有节点的首节点，也就是说，每个区间是一个区域。形式地，给定一个带有初始节点 n_0 的流图 G 和 G 的一个节点 n，首节点为 n 的区间（记为 I(n)）定义如下：

1. n 在 I(n)中。
2. 如果某个节点 m ($m \neq n_0$)的所有前驱在 I(n)中，则 m 在 I(n)中。

664

3. I(n)中没有其他节点。

因此从 n 开始，根据规则2添加节点 m 我们就可以建立 I(n)。按什么顺序添加两个候选的 m无关紧要，因为一旦某个节点的前驱都在 I(n)中，它们就会在 I(n)中，而且根据规则2，每个候选变量最终都会被添加到 I(n)中。最后，没有更多的节点可以被添加到 I(n)中，节点的结果集合是首节点为 n 的区间。

10.9.5 区间划分

给定任意的流图 G，我们可以用下述方法将 G 划分为不相交的区间。

算法10.15 流图的区间分析。

输入：带有初始节点 n_0 的流图 G。

输出：流图 G 的不相交区间的一个划分。

方法：对任意的节点 n，通过上面概述的算法计算 I(n)：

```
I(n) := {n};
while 存在所有前驱都在 I(n)中的
      节点, m≠n0 do
          I(n) := I(n) ∪ {m}
```

划分中作为区间首节点的特定节点按照下面的方法进行选择。初始时，没有节点是“被选中的”。

```
构造 I(n0)并“选中”该区间中的所有节点;
while 存在一个还没有“被选中的”的节点m但有一个前驱节点被选中的节点m do
          构造 I(m)并“选中”该区间中所有的节点
```

□

一旦候选 m 具有一个被选中的前驱 p，m 就永远不能被添加到某个不包含 p 的区间中。于是，候选的 m 一直都是候选直到它们被选作它们自己区间的首节点为止。因而算法10.15中选择区间首节点 m 的顺序不会影响最终的区间划分。而且，如果所有的节点从 n_0 都是可达的，对从 n_0 到 n 的路径的长度进行归纳可以证明：节点 n 最后不是被放到其他节点的区间中，就是成为它自己区间的首节点，但两者不能同时成立。因此，由算法10.15构造的区间集合确实是 G 的划分。

例10.34 让我们为图10-45寻找区间划分。我们从构造 I(1)开始，因为节点1是初始节点。我们可以将2加到I(1)，因为2的惟一前驱是1。但是，我们不能将3添加到I(1)中，因为它具有不在I(1)中的前驱4和8，同样地，除1和2之外的每个其他节点都具有不在 I(1)中的前驱，所以，I(1) = {1，2}。

现在我们可以计算 I(3)，因为3有一些“被选中的”的前驱，即1和2，但是3本身不在区间

中。可是，没有节点可以被添加到I(3)中，所以 I(3) = {3}。现在，4是一个首节点，因为它有 665
一个前驱3在区间中。我们可以将5和6添加到I(4)中，因为它们只有一个前驱4，但是没有其他节点可以被添加进去，比如，7有前驱10。

接下来，7成为首节点，我们可以将8添加到 I(7)中。然后，我们可以将9和10添加进去，因为它们只有前驱8。于是，图10-45划分的区间为：

I(1) = {1, 2}　　I(4) = {4, 5, 6}

I(3) = {3}　　I(7) = {7, 8, 9, 10}　　□

10.9.6 区间图

从流程图 G 的区间，根据下列规则可以构造一个新的流图I(G)：

1. I(G) 的节点与 G 的区间划分中的区间相对应。

2. I(G) 的初始节点是包含 G 的初始节点的 G 的区间。

3. 有一条从区间 I 到不同的区间 J 的边，当且仅当在 G 中存在一条从 I 中某个节点到 J 的首节点的边。注意，从 J 的外部不可能有边进入 J 的某个不是首节点的节点 n ，因为在算法10.15中无法将 n 添加到 J 中。

交替使用算法10.15和上述区间图构造方法，就可以产生图 G 的序列，I(G)，I(I(G))，…。最后，我们会得到一个图，该图的每个节点是一个完全孤立的区间。该图被称作 G 的限制流图。有趣的是，流图是可约的，当且仅当它的限制流图是单个节点。[⊖]

例10.35 对图10-45反复应用区间构造方法所得到的结果如图10-48所示。该图的区间已在例10.34中给出，而且从其构造的区间图如图10-48a所示。注意，图10-45中的边10→7在图10-48a中不会导致一条从表示{7, 8, 9, 10}的节点到它自身的边，因为区间图构造方法明确地排除了这样的循环。还要注意的是，图10-45中的流图是可约的，因为它的限制流图是单个节点。

□

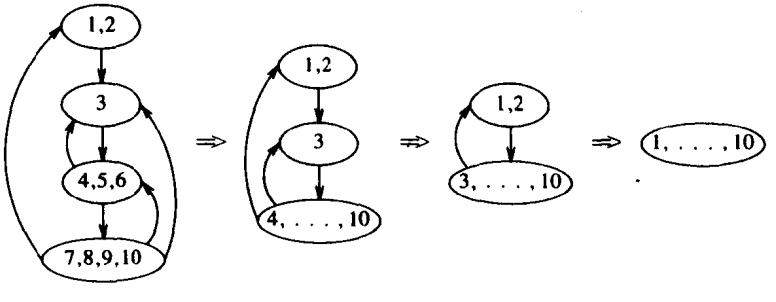


图10-48 区间图序列

10.9.7 节点分裂

如果我们获得一个不是单个节点的限制流图，只要将一个或多个节点分裂我们就可以继续构造下去。如果节点 n 有 k 个前驱，我们可以用 k 个节点 n_1，n_2，…，n_k 取代 n。n 的第 i 个前驱只能成为 n_i 的前驱，而 n 的所有后继都成为所有 n_i 的后继。

如果我们将算法10.15应用到结果图上，则每个 n_i 具有一个惟一的前驱，所以它一定会成为该前驱的区间的一部分。从而，一个节点分裂加上一轮区间划分将导致一个带有更少节点的图。

⊖ 事实上，该定义是历史上原来的定义。

因而，区间图的构造（需要时可能有散布的节点分裂）最终必然会获得一个只有单个节点的图。该结论的重要性在下一节会更明显，届时我们将利用这两个图上的操作来设计数据流分析算法。

例10.36 考虑图10-49a的流图，它是其自身的限制流图。我们可以将节点2分裂成2*a*和2*b*，分别带有前驱1和3，如图10-49b所示。如果我们应用两次区间划分，将会得到图10-49c和图10-49d所示的图序列，导致了单个节点。 □

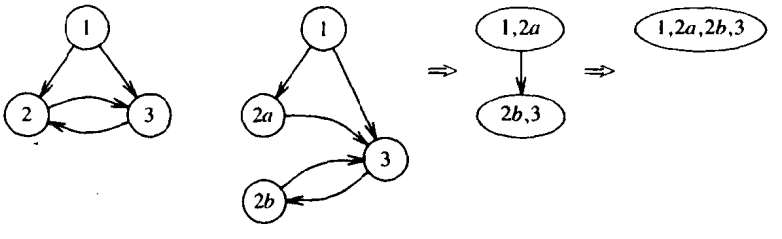


图10-49 区间划分后的节点分裂

10.9.8 T_1-T_2 分析

有一种方便的办法可以达到同区间分析同样的效果，即对流图应用两次简单的变换：

T_1：如果 n 是一个带有循环的节点，即存在边 $n \rightarrow n$，则删除这条边。

T_2：如果节点 n（不是初始节点）具有惟一的前驱 m，那么 m 通过删除 n 并将 n 的所有后继（可能包括 m）变成 m 的后继即可消掉 n。

下面是关于 T_1 和 T_2 变换的一些有趣的事实：

1. 如果我们按任意顺序将 T_1 和 T_2 应用到流图 G 上，直到产生一个无法再应用 T_1 或 T_2 的流图为止，那么结果将产生一个惟一的流图。原因是即使先应用了某些 T_1 或者 T_2，由 T_1 删除的或者由 T_2 消掉的候选者仍然是候选者。

2. 对 G 彻底应用 T_1 和 T_2 所产生的流图是 G 的限制流图。其证明有些琐碎，留作练习。结果，“可约流图”的另一个定义就是可以被 T_1 和 T_2 转化为单个节点的流图。

例10.37 图10-50给出了一系列 T_1 和 T_2 变换，这些变换是从一个对图10-49b重命名的流图开始的。在图10-50b中，*c* 消掉了 *d*。注意，图10-50b中 *cd* 上的循环是由图10-50a中的边 $d \rightarrow c$ 引入的。在图10-50c中该循环被 T_1 删除了。还要注意，在图10-50d中 *a* 消掉 *b* 以后，从 *a* 和 *b* 到节点 *cd* 的边就成了一个边。 □

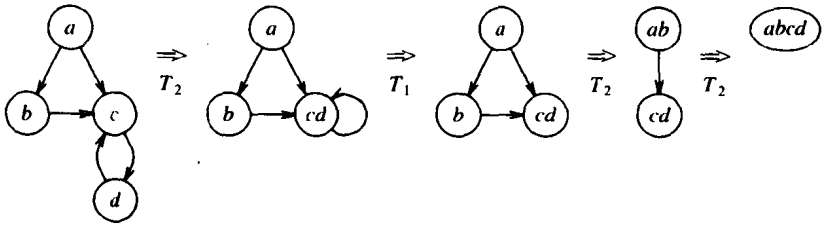


图10-50 利用 T_1 和 T_2 对流图的化简

10.9.9 区域

回想一下，在10.5节中，流图中的区域（region）是含有一个首节点的一组节点 N，首节

666
~
668

点支配区域中所有其他节点。除了（有可能）进入首节点的边以外，N 中节点间的所有边都在区域中。例如，每个区间是一个区域，但存在不是区间的区域，例如，因为它们可能删除了某些区间要包含的节点或者某些回到首节点的边。正如我们将要看到的，还有一些区域比任何区间都要大。

用T_1和T_2化简流图 G 时，下列条件一直为真：

1. 一个节点代表 G 的一个区域。

2. 一条从 a 到 b 的边代表一组边，每条这样的边都是从 a 所表示的区域中的某个节点指向 b 所表示的区域的首节点。

3. G 中每个节点和每条边正好由当前图中的一个节点或一条边代表。

要理解为什么这些结论是正确的，首先请注意，对 G 本身它们一般是成立的。每个节点本身是一个区域，每条边只代表它自己。假设我们对代表区域 R 的某节点 n 应用 T_1，而循环 $n\rightarrow n$ 代表某组边E，E 中每条边一定进入 R 的首节点。如果我们将边 E 加入区域 R 中，R 仍然是一个区域，所以删除边 $n\rightarrow n$ 之后，节点 n 代表 R 和 E 中的边，从而上述条件1至条件3成立。

如果我们使用 T_2，用节点 a 消掉节点b，令 a 和 b 分别代表区域 R 和 S。同样，令 E 为边 $a\rightarrow b$ 所代表的那组边。我们断言 R，S 和 E 合起来形成一个区域，其首节点是 R 的首节点。为了证明这一点，我们必须验证 R 的首节点支配 S 中的每一个节点，否则，必须存在一条通向 S 的首节点的路径，而且该路径不以 E 中的边为结束。那么该路径的最后一条边在当前流图中只能由进入 b 的某条其他边代表。但不存在这样的边，或者说不能用 T_2 消掉 b。

例10.38 图10-50b中标记为 cd 的节点代表图10-51a所示的区域，它是通过由 c 消掉d 所形成的。注意，边$d\rightarrow c$不是该区域的一部分；图10-50b中，cd 上的循环代表这条边。但是，图10-50c中，边 $cd\rightarrow cd$ 已经被删除，现在节点 cd 代表图10-51b所示的区域。

图10-50d中，节点 cd 仍然代表图10-51b的区域，而节点 ab 代表图10-51c的区域。图10-50d中的边 $ab\rightarrow cd$ 代表图10-50a中原始流图的边 $a\rightarrow c$ 和 $b\rightarrow c$。当我们应用 T_2 得到图10-50e时，剩下的节点代表整个流图，即图10-50a。 □

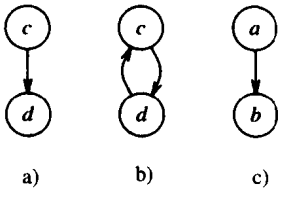


图10-51 一些区域

我们应该注意，上面提到的T_1和 T_2 变换的性质对区间分析仍然成立。下面的事实留作练习：当我们建立 I(G)，I(I(G))，…时，这些图中的每个节点代表一个区域，每条边代表一组满足上述性质2的边。

10.9.10 寻找支配节点

我们用一个高效的算法来结束这一节，该算法涉及一个我们经常使用的概念，在研究流图理论和数据流分析时我们还要继续使用它。基于下面的原理，我们将给出一个简单的算法来计算流图中每个节点 n 的支配节点：如果 p_1，p_2，…，p_k 是n的所有前驱并且 $d\neq n$，那么 d *dom* n 当且仅当对每个 i，d *dom* p_i。该方法和把交运算作为聚合操作符的前向数据流分析（如可用表达式）相似，那时我们取 n 的支配节点集的近似值，并通过反复地依次访问所有这些节点来对其求精。

现在，我们选择的初始近似值含有只被初始节点支配的初始节点，除初始节点以外，所有的节点都互相支配。直观地，该方法之所以正确，是因为只有当我们找到一条可以证明 m *dom* n 为假的路径时，才能将那个候选支配节点排除。如果我们不能找到这样一条从初始节点

到 n 并排除 m 的路径，那么 m 就确实是 n 的支配节点。

算法10.16 寻找支配节点。

输入：流图 G，带有节点集合 N，边集合 E 和初始节点 n_0。

输出：关系 *dom*。

方法：重复使用图10-52中的过程来计算 $D(n)$，即 n 的支配节点集。过程结束时，d 在 $D(n)$中当且仅当 d *dom* n。至于如何检测 $D(n)$发生了变化，读者可以参照算法10.2给出其详细描述。 □

```
(1)  D(n0) := {n0};
(2)  for N - {n0}中的n do D(n) := N;
     /* 初始化完毕 */
(3)  while D(n)发生变化 do
(4)      for N - {n0}中的n do
(5)          D(n) := {n} ∪ ∩ D(p);
                        n的前驱P
```

图10-52 支配节点计算算法

可以证明图10-52中第(5)行计算出的 $D(n)$ 始终是当前 $D(n)$ 的子集。因为 $D(n)$ 不能无限地变小，最后我们一定能终止 while 循环。收敛之后，$D(n)$ 是 n 的支配节点集的证明留给感兴趣的读者。图10-52中的算法效率非常高，因为 $D(n)$ 可以用一个位向量表示，而且第(5)行中的集合操作可以用逻辑运算 **and** 和 **or** 来实现。 □

例10.39 让我们回到图10-45的流图中，并假设在第(4)行的 for 循环中节点是按数字顺序访问的。节点2只有前驱1，所以 $D(2):=\{2\}\cup D(1)$。因为1是初始节点，$D(1)$在第(1)行被赋值为 $\{1\}$。从而，$D(2)$在第(5)行被置为 $\{1,2\}$。

然后考虑节点3，它带有前驱1，2，4和8。第(5)行给出 $D(3)=\{3\}\cup(\{1\}\cap\{1,2\}\cap\{1,2,\cdots,10\})=\{1,3\}$。剩下的计算是：

$$D(4)=\{4\}\cup(D(3)\cap D(7))=\{4\}\cup(\{1,3\}\cap\{1,2,\ldots,10\})=\{1,3,4\}$$

$$D(5)=\{5\}\cup D(4)=\{5\}\cup\{1,3,4\}=\{1,3,4,5\}$$

$$D(6)=\{6\}\cup D(4)=\{6\}\cup\{1,3,4\}=\{1,3,4,6\}$$

$$D(7)=\{7\}\cup(D(5)\cap D(6)\cap D(10))$$
$$=\{7\}\cup(\{1,3,4,5\}\cap\{1,3,4,6\}\cap\{1,2,\ldots,10\})=\{1,3,4,7\}$$

$$D(8)=\{8\}\cup D(7)=\{8\}\cup\{1,3,4,7\}=\{1,3,4,7,8\}$$

$$D(9)=\{9\}\cup D(8)=\{9\}\cup\{1,3,4,7,8\}=\{1,3,4,7,8,9\}$$

$$D(10)=\{10\}\cup D(8)=\{10\}\cup\{1,3,4,7,8\}=\{1,3,4,7,8,10\}$$

可以看出：while 循环的第二遍没有改变 D，于是上面的值输出关系 *dom*。 □

10.10 高效数据流算法

669 ~ 671

本节考虑两种使用流图理论加快数据流分析的方法。第一种方法使用深度优先遍历减少10.6节中算法的迭代次数，第二种方法使用区间或者 T_1、T_2 变换实现10.5节的语法制导方法。

10.10.1 迭代算法中的深度优先顺序

至此，在所有已研究问题中，某个节点的到达定义、可用表达式或者活跃变量等信息都是沿一条无环路径传播到该节点的。例如，定义 d 在 *in*[B]中，则存在从包含 d 的块到达 B 的无环路径，并且 d 存在于该路径上的所有节点的 *in* 和 *out* 中。同样地，如果表达式 x+y 在块 B 的入口点是不可用的，则存在一条这样的无环路径：要么该路径从初始节点开始且不包含任何注销或产生 x+y 的语句，要么该路径从一个注销x+y的块开始，而且该路径上不再产生x+y。最后，对于活跃变量，如果x在块 B 的出口是活跃的，则存在一条从 B 到 x 的引用的无环路径，

该路径上没有对x的定义。

读者可能发现，在各种情况下，路径上带有环路不会产生什么影响。例如，如果从块 *B* 的末尾沿着一条有环路径到达x的引用，可以删除该环以找出一条更短的路径，沿该路径从 *B* 仍然可以到达 x 的引用。

如果所有的有用信息都是沿无环路径传播，就可以修改迭代数据流算法中访问节点的顺序，以便在较少的几次遍历之后，就可以确定信息已经经过所有的无环路径。特别地，Knuth [1971b]搜集的统计信息指出：典型的流图都具有非常低的区间深度，其中，区间深度是指为得到限定流图需要应用区间划分的次数；结果发现平均次数是2.75。定义“深度”是后退边最多的无环路径上的后退边的最大数目（如果流图是不可约的，深度依赖于所选的深度优先生成树）。可以证明流图的区间深度永远不会比“深度”小。

回想上一节关于深度优先生成树的讨论，我们注意到如果 $a \rightarrow b$ 是边，那么只有当该边是后退边时，*b* 的深度优先编号才比 *a* 的编号小。因而用下面的语句代替图10-26中的第(5)行，该行访问流图中的每个块 *B* 来计算到达定义：

for 深度优先顺序中的每个块 *B* **do**

假设存在一条传播定义 *d* 的路径：

3→5→19→35→16→23→45→4→10→17

其中，整数表示该路径上块的深度优先编号。第一次通过图10-26中第(5)至(9)行的循环时，*d* 会从 *out*[3]传播到 *in*[5]，再传播到 *out*[5]，依此类推，一直到达 *out*[35]。因为深度优先编号16在35之前，所以将*d* 放进*out*[35]时已经计算了 *in*[16]，因此本次循环中，*d* 不会到达 *in*[16]。然而，下一次在执行第(5)至(9)行的循环中计算 *in*[16]时，*d* 因为在 *out*[35]中而被包含进来。定义*d* 还会传播到 *out*[16]、*in*[23]等，一直到 *out*[45]。因为 *in*[4]已经被计算，本次循环结束。在 672
第三遍中，*d* 传播到 *in*[4]，*out*[4]，*in*[10]，*out*[10]和 *in*[17]。经过三遍循环之后，我们确定定义 *d* 可以到达块17。[⊖]

从该例可以很容易地得到一个普遍原理。如果在图10-26中应用深度优先顺序，那么沿无环路径传播到达定义所需要的遍数不会超过后退边个数加1。后退边是指沿该路径从编号高的块到编号低的块所经过的边。所以所需的遍历次数是深度加1。当然，算法10.2没有发觉这样的事实，即所有的定义已经到达了它们所能到达的地方。使用深度优先顺序的算法遍数的上界实际上是深度加2。如果我们相信Knuth[1971b]的结果，则遍历次数是深度加5。

深度优先顺序对可用表达式（算法10.3）计算或通过前向传播解决数据流问题的方法都很有利。对于活跃变量问题（通过向后传播解决），如果选择深度优先顺序的逆序，遍历次数的平均值可以达到5遍。从而，只需一遍即可将块17中的变量引用沿如下路径向后传播到 *in*[4]：

3→5→19→35→16→23→45→4→10→17

至此，为到达 *out*[45]我们必须等待下一遍。在第二遍中，块17的变量引用将到达 *in*[16]。第三遍从 *out*[35]到达 *out*[3]。一般地，如果我们选择按深度优先顺序的逆序访问节点，使得在一遍中沿着编号递减的顺序进行传播，则深度加1遍足以将变量的引用沿任意无环路径向后传播。

10.10.2 基于结构的数据流分析

再加一点劲，我们就可以实现下面的数据流算法：它访问节点（并且应用数据流方程）的次数不会大于流图的区间深度，且访问节点的平均次数往往比区间深度小得多。做更多的努力

⊖ 定义*d* 还到达*out*[17]，但这与正被讨论的路径无关。

能否节省更多的次数？就此还没有得到严格的证实，但是像这种基于区间分析的技术已经被用在若干编译器中。此外，这里所提出的思想不仅仅可以应用在10.5节讨论的 **if…then** 和 **do…while** 语句上，也可以应用在对各种结构化控制语句的语法制导数据流算法中，这些已经出现在许多编译器中。

我们的算法是基于在流图上应用 T_1 和 T_2 变换导出的结构上。和10.5节一样，我们关心的是控制流通过一个区域时产生或注销的定义。和通过if或 while 语句定义的区域不同，普通的
673 区域可以有多个出口，所以对于区域 R 中的每个块 B，我们要计算从区域首节点到块B末尾的路径上所产生和注销的定义集合 $gen_{R,B}$ 和 $kill_{R,B}$。这些集合将用于定义转移函数 $trans_{R,B}(S)$。对任意的定义集合 S，给定 S 中到达 R 首节点的所有定义，$trans_{R,B}(S)$将告诉我们什么样的定义集合会沿着 R 中的路径到达块 B 的末尾。

正如我们在10.5节和10.6节所看到的，到达块B末尾的定义分为以下两类：

1. 在 R 中产生并传播到块 B 末尾的与 S 无关的定义。

2. 不是在 R 中产生的定义，但是也没有在 R 的首节点到块 B 末尾的路径中被注销，因此它们在 $trans_{R,B}(S)$中当且仅当它们在 S 中。

从而我们可以写出下面形式的 $trans$：

$$trans_{R,B}(S) = gen_{R,B} \cup (S - kill_{R,B})$$

算法的核心是为流图的某个 (T_1, T_2) 分解所定义的逐渐变大的区域计算 $trans_{R,B}$。目前，假设流图是可约的，对算法施加简单的改变即可应用到不可约流图。

起点是由单个块 B 构成的区域。此时，区域的转移函数就是块本身的转移函数，因为一个定义到达块的末尾当且仅当它或者是该块产生的，或者在集合 S 中没有被注销。也就是说：

$$gen_{B,B} = gen[B]$$
$$kill_{B,B} = kill[B]$$

现在，考虑用 T_2 构造区域 R。也就是说，R_1 消掉 R_2 形成 R，如图10-53所示。首先注意到，因为任何从 R_2 到 R_1 首节点的边都不属于 R，所以在区域 R 中没有 R_2 到 R_1 的后退边。任何在 R 中的路径都首先（可选择地）穿过 R_1，然后（可选择地）穿过 R_2，但不能再返回 R_1。还要注意，R 的首节点是 R_1 的首节点。我们可以断定：在 R 中，R_2 不会影响 R_1 中节点的转移函数。也就是说，对 R_1 中所有的 B，下列式子成立：

$$gen_{R,B} = gen_{R_1,B}$$
$$kill_{R,B} = kill_{R_1,B}$$

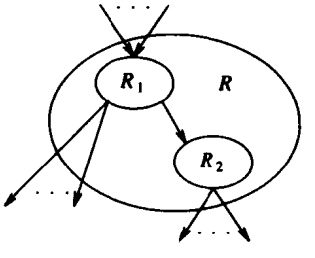


图10-53 使用T_2构造的区域

对 R_2 中的 B，满足下面任何一个条件的定义都能到达 B 的末尾：

1. 定义是在 R_2 中产生的。

2. 定义是在 R_1 中产生并到达 R_2 首节点的某个前驱节点的末尾，而且在 R_2 的首节点到B的路径上没有注销。

3. 定义属于 R_1 首节点的定义集合 S，在到达 R_2 首节点的某个前驱时没有被注销，而且从 R_2 的首节点到 B 的路径上也没有被注销。

所以，能够到达 R_2 首节点的前驱块（位于R_1 中）末尾的定义起着特殊的作用。本质上，假设 S 是进入 R_1 首节点的定义集合，当 S 中的定义试图通过 R_2 首节点的一个前驱到达 R_2 首节

点时，我们关心定义集合 S 所发生的变化。到达 R_2 首节点某个前驱的定义集合成为 R_2 的输入集合，我们对该集合应用 R_2 的转移函数。

于是，令 G 为 R_2 首节点的所有前驱 P 的 $gen_{R_1,P}$ 的并集，并令 K 为所有前驱 P 的 $kill_{R_1,P}$ 的交集。那么如果 S 是到达 R_1 首节点的定义集合，沿着 R 中的路径到达 R_2 首节点的定义集合就是 $G \cup (S-K)$。因此，对于 R_2 的块 B 在 R 中的转移函数可以通过下式计算：

$$gen_{R,B} = gen_{R_2,B} \cup (G - kill_{R_2,B})$$

$$kill_{R,B} = kill_{R_2,B} \cup (K - gen_{R_2,B})$$

接下来，考虑通过在区域 R_1 应用变换 T_1 建立区域 R 的情况。一般情况如图10-54所示，注意，通过对 R_1 加上一些指向 R_1 首节点（当然也是 R 的首节点）的回边形成区域 R。正如我们在本节前面所讨论的，穿过首节点两次的路径会成为环路，但在这里我们无需考虑环路。

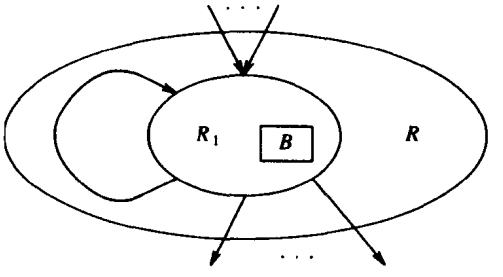


图10-54 用 T_1 建立的区域

于是，所有在块 B 末尾产生的定义都通过下面两种方式之一产生：

1. 定义在 R_1 中产生，并且为了到达块 B 的末尾不需要合并到 R 中的回边。

2. 定义在 R_1 中的某处产生，经过一条回边到达首节点的一个前驱，并且在首节点到 B 的传播中没有被注销。

如果我们令 G 为 R 中首节点所有前驱 P 的 $gen_{R_1,P}$ 的并集，那么

$$gen_{R,B} = gen_{R_1,B} \cup (G - kill_{R_1,B})$$

一个定义从首节点传播到 B 时被注销当且仅当它在所有的无环路径上都被注销，所以合并到R中的回边不会导致更多的定义被注销，即：

$$kill_{R,B} = kill_{R_1,B}$$

例10.40 再考虑一下图10-50中的流图，它的 (T_1, T_2) 分解如图10-55所示，图中是带有命名的分解的区域。在图10-56中还给出了假定的位向量，用来表示三个定义以及它们在图10-55的各个块中是否被产生或注销。

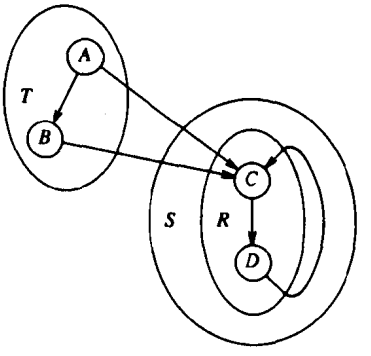


图10-55 流图的分解

从里向外考察图10-55。开始注意到，单节点区域（分别称为A，B，C 和D）的 gen 和 $kill$如图10-56所示。然后继续考察区域 R，R 通过在 C 上应用 T_2 消掉 D 形成。根据 T_2 的规则，我们注意到对于 C，gen 和 $kill$ 没有发生改变，即：

$$gen_{R,C} = gen_{C,C} = 000$$
$$kill_{R,C} = kill_{C,C} = 010$$

对于节点D，必须在区域 C 中找出区域 D 首节点所有前驱的 gen 的并集。当然，区域 D 的首节点就是节点 D，而且区域 C 中只有该首节点的一个前驱，即节点 C。因此，

$$gen_{R,D} = gen_{D,D} \cup (gen_{C,C} - kill_{D,D}) = 001 + (000 - 000) = 001$$
$$kill_{R,D} = kill_{D,D} \cup (kill_{C,C} - gen_{D,D}) = 000 + (010 - 001) = 010$$

现在，利用 T_1 从区域 R 构造区域 S。$kill$ 没有发生改变，

块	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

图10-56 图10-55中块的 gen 和 $kill$ 信息

所以有

$$kill_{S,C} = kill_{R,C} = 010$$
$$kill_{S,D} = kill_{R,D} = 010$$

为计算 S 的 gen，我们注意到，指向 S 首节点并从 R 合并到 S 中的惟一回边就是边 $D \rightarrow C$。因此，

$$gen_{S,C} = gen_{R,C} \cup (gen_{R,D} - kill_{R,C}) = 000 + (001 - 010) = 001$$
$$gen_{S,D} = gen_{R,D} \cup (gen_{R,D} - kill_{R,D}) = 001 + (001 - 010) = 001$$

674 ~ 677

对区域 T 的计算类似于对区域 R 的计算，于是得到

$$gen_{T,A} = 100$$
$$kill_{T,A} = 010$$
$$gen_{T,B} = 010$$
$$kill_{T,B} = 101$$

最后，计算区域 U，即整个流图的 gen 和 $kill$。因为U 是在 T 通过变换 T_2 消掉 S 时形成的，所以节点 A 和 B 的 gen 和 $kill$ 值与上面给出的值相同，并未改变。对于 C 和 D，注意到 S 的首节点，即节点C，在区域 T 中有两个前驱A 和B 。所以，可以计算

$$G = gen_{T,A} \cup gen_{T,B} = 110$$
$$K = kill_{T,A} \cap kill_{T,B} = 000$$

然后计算

$$gen_{U,C} = gen_{S,C} \cup (G - kill_{S,C}) = 101$$
$$kill_{U,C} = kill_{S,C} \cup (K - gen_{S,C}) = 010$$
$$gen_{U,D} = gen_{S,D} \cup (G - kill_{S,D}) = 101$$
$$kill_{U,D} = kill_{S,D} \cup (K - gen_{S,D}) = 010$$

□

我们已经为每个块 B 计算出了$gen_{U,B}$和 $kill_{U,B}$，其中U 是整个流图构成的区域，实质上我们已经为每个块 B 计算出了$out[B]$。也就是说，如果考察$trans_{U,B}(S) = gen_{U,B} \cup (S - kill_{U,B})$ 的定义，会看到 $trans_{U,B}(\varnothing)$正好是 $out[B]$。但$trans_{U,B}(\varnothing) = gen_{U,B}$，于是，基于结构的到达定义算法的实现就是将 gen 用作 out，并且通过求前驱的 out 的并集来计算 in。这些步骤可以归纳为以下算法。

算法10.17 基于结构的到达定义。

输入：可约流图 G 以及 G 中每个块 B 的定义集合 $gen[B]$ 和 $kill[B]$。

输出：每个块 B 的 $in[B]$。

方法：

1. 找出 G 的 (T_1, T_2) 分解。

2. 对分解中的每个区域 R，从内向外计算 R 中每个块 B 的$gen_{R,B}$和 $kill_{R,B}$。

3. 如果 U 是整个图构成的区域的名字，那么对每个块 B，置 $in[B]$为块 B 所有前驱 P 的 $gen_{U,P}$的并集。

678

□

10.10.3 对基于结构的算法的一些速度上的改进

首先，如果存在一个转移函数 $G \cup (S - K)$，那么从 K 中删除 G 的某些成员不会改变该函数。因此，当我们应用 T_2时，不是用如下公式：

$$gen_{R,B} = gen_{R_2,B} \cup (G - kill_{R_2,B})$$
$$kill_{R,B} = kill_{R_2,B} \cup (K - gen_{R_2,B})$$

而是用下面的公式代替上面的第二个公式：

$$kill_{R,B} = kill_{R_2,B} \cup K$$

这样，区域 R_2 中的每个块都可以节省一个操作。

另一个有用的想法是，因为惟一的一次应用 T_1 的操作，发生在先用 R_1 消掉了某个区域 R_2 之后，并且存在一些从 R_2 到 R_1 首节点的回边，所以，不必先用 T_2 操作改变 R_2 而后再用 T_1 操作改变 R_1 和 R_2，而是通过执行如下操作，合并这两组改变：

1. 使用 T_2 规则，计算 R_2 中 R_1 首节点的所有前驱节点的新的转移函数。

2. 使用 T_1 规则，计算 R_1 中所有节点的新的转移函数。

3. 使用 T_2 规则，计算 R_2 中所有节点的新的转移函数。注意，应用 T_1 的反馈已经到达 R_2 的前驱并利用 T_2 规则传到了 R_2 的所有节点，所以不必再对 R_2 应用 T_1 规则。

10.10.4 处理不可约流图

如果对流图的 (T_1, T_2) 化简终止在一个非单个节点的限制流图上，那么必须执行节点分裂。分裂限制流图的一个节点对应着复制该节点表示的整个区域。例如，图10-57的流图中，原有的9个节点被 T_1 和 T_2 分解成由边连接的三个区域，图10-57中给出了对其执行节点分裂的结果。

正如上一节所提到的，依次进行节点分裂和化简，流图一定可以简约成单个节点。分裂的结果是，原始流图的某些节点在单个节点表示的区域中存在多个副本。稍加改变即可将算法10.17应用在该区域上。惟一的不同是：分裂一个节点时，在分裂节点表示的区域中原图中节点的 *gen* 和 *kill* 必须被复制。例如，在图10-57中，左图两节点区域中的节点的 *gen* 和 *kill* 被复制到右图的所有两节点区域的相应节点上。最后，当计算所有节点的 *in* 时，如果在分裂的区域图中，同一个原图节点存在多个代表，则通过求其所有代表的 *in* 的并集来计算它们的 *in* 值。 679

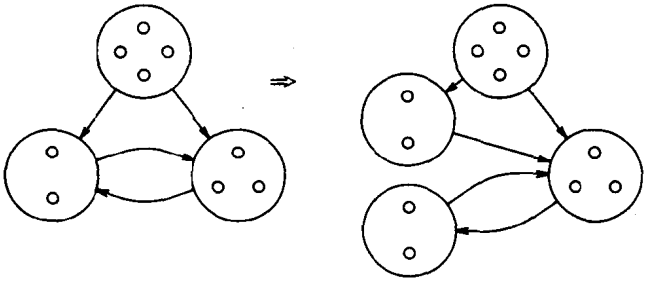


图10-57 不可约流图的分裂

在最坏的情况下，节点分裂可能导致所有区域的节点总数呈指数增长。因而，如果预计存在许多不可约流图，则不应该使用基于结构的方法。幸运的是，不可约流图非常少，通常可以忽略节点分裂的开销。

10.11 一个数据流分析工具

如前所述，许多数据流问题都有很大的相似性。10.6节的数据流方程的区别为：

1. 所用的转移函数，其形式均为 $f(X) = A \cup (X - B)$。例如，对到达定义来说，$A = kill$，$B = gen$。

2. 迄今为止，所有的聚合操作符都是并或交。

3. 信息的传播方向：前向或后向。

因为区别不大，使得所有问题可以按照统一的方法进行处理。在Kildall[1973]中给出了一

种这样的方法。Kildall 还实现了一个简化数据流问题的工具，并应用到一些编译器项目中。但是，也许因为该系统节省的劳动量没有语法分析器的生成器所节省的劳动量多，该系统并没有
680 得到广泛应用。但是，它提出了一种简化优化编译器实现的方法，并且它有助于统一本章的各种思想，因此我们应当知道它能够做些什么。此外，本节介绍如何开发更强大的数据流分析策略，它们比目前为此提到的算法能提供更加精确的信息。

10.11.1 数据流分析框架

下面描述前向传播问题的建模框架。如果只考虑求解数据流问题的迭代形式，则流的方向没有区别。将边的方向反转过来，并对初始节点的估计做一些较小的调整，则可将后向问题转换成前向问题。因为可约流图的反转不必是可约的，所以基于结构的算法不能按照同样的方法解决前向和后向问题。我们将后向问题的处理留作练习，主要集中处理前向问题。

一个*数据流分析框架*由以下内容组成：

1. 一个需要传播的*值集合* V。*in* 和 *out* 的值都是V 的成员。
2. 从 V 到 V 的*转移函数集合* F。
3. 集合 V 上的二进制与（meet）操作 $\wedge$，用来代表聚合操作符。

例10.41 对于到达定义，V由该程序中的定义集合的所有子集组成。集合 F 是形如$f(X) = A \cup (X-B)$ 的所有函数的集合，其中，A、B 是定义集合，即 V 的成员。A 和 B 分别是代表 *gen* 和 *kill*。最后，操作 $\wedge$ 代表并操作。

对于可用表达式，V 由该程序计算出的表达式集合的所有子集组成，F 是形如 $f(X) = A \cup (X-B)$ 的表达式的集合，此时，A 和 B 是表达式集合。操作 $\wedge$ 为交操作。 □

例10.42 尽管处理复杂问题时，时间复杂性和难度都会增加，Kildall的方法并不局限于处理简单的问题。该练习将给出强有力的说明。计算得到的数据流信息指出，在某一点上所有的表达式对都具有相同的值。我们从下面的例子中可以得到一些启发。通过确定哪些变量为常数，可以获得比到达定义更多的信息。例如，新的框架会知道：如果 x 是由 d: x := x+1定义的，而且x在赋值之前的值为常量，则通过赋值后x仍然是常量。

相反，用到达定义传播常量，如果假设x的值不是常量，则语句d可能是x的一个定义。一
681 遍之后 d: x := x+1的右部可能被一个常量取代。然后在另外一次常量传播可以检测出在 d 中定义的x的引用实际上是常量引用。

在新的框架中，V是程序变量到特定值集合的所有映射的集合。值集合包含：

1. 所有常量。
2. *nonconst*，即已经确定为没有常量值的变量。如果在数据流分析期间，两条路径上存在两个不同的值（假设是2和3）分别赋给了x，或者一条路径上对x的定义是读语句，则将值*nonconst*赋给变量x。
3. *undef*，指无法断言的变量值。可能在数据流分析的前期执行中，没有发现变量定义到达被讨论的点。

注意，*nonconst* 和 *undef* 是不一样的，实质上它们是相反的。前者指的是已经看到某个变量可以用许多方法进行定义，以致于我们知道它不是常数。后者指的是关于该变量的信息很少，以致于我们根本不知道它的任何信息。

与操作用下面的表来定义。令 μ 和 ν 是 V 的两个成员，也就是说，μ和ν都将每个变量映射到一个常量、*nonconst* 或 *undef* 上。然后在图10-58中定义了函数 $\rho = \mu \wedge \nu$，其中，对每个变

量x，ρ(x)的值都根据 μ(x) 和 ν(x) 的值给出。在该表中，c 是任意的常量，d 是不等于 c 的任何常量。例如，如果 μ(x) = c，而且 ν(x) = d，那么显然x沿两条不同的路径分别获得两个不相同的值c和d。在这两条路径会合的地方，x的值一定不是常量，因此 ρ(x) = *nonconst*。又例如，如果沿某条路径对x一无所知（μ(x) = *undef*）而沿另外的一条路径可以确定x的值为 c，那么在这两条路径的汇合处，我们只能断言x具有值c。当然，如果后来又发现另一条到达该汇合点的路径，沿该路径x还具有一个不同于 c 的值，则在会合后将 x 赋值成 *nonconst*。

ν(x)	μ(x)			
	nonconst	*c*	*d*(≠*c*)	*undef*
nonconst	*nonconst*	*nonconst*	*nonconst*	*nonconst*
c	*nonconst*	*c*	*nonconst*	*c*
undef	*nonconst*	*c*	*d*	*undef*

图10-58 根据 μ(x) 和 ν(x)定义的 ρ(x)

最后，需要设计函数集合 F，F 中的函数反映从块开始到块结束的信息转移。尽管思想很直接，但对函数集合的描述非常复杂。因此，首先通过描述单个定义语句的函数给出一个函数的基础集合，然后通过合成函数基础集合构造整个函数集合以反映具有多条定义语句的块。

1. F 中包含恒等函数，该函数反映没有定义语句的块。如果 I 是恒等函数，μ 是从变量到值的映射，那么 I(μ) = μ。注意，μ本身不必是恒等映射，它是任意的。

2. 对每个变量x和常数c，F 中存在一个函数 f，使得对 V 中的每个映射μ，我们有f(μ) = ν，其中，对x之外的所有w，ν(w) = μ(w)，并且 ν(x) = c。该函数反映赋值语句x := c的动作。

3. 对任意三个变量x、y和z（允许相同），F 中存在一个函数 f，使得对V中任意的一个映射μ，我们有 f(μ) = ν。映射ν定义如下：对 x 之外的每个 w，我们有ν(w) = μ(w)，并且 ν(x) = μ(y) + μ(z)。如果 μ(y)或μ(z)为 *nonconst*，那么该和也是 *nonconst*；如果μ(y)或μ(z)为*undef*，但不是 *nonconst*，那么结果是 *undef*。该函数说明了赋值语句x := y+z的作用。在本章将+看作通用操作符，在此，如果操作是一元的、三元的或者更多元，则需要对操作符进行修正。此外，在考虑复制语句 x := y的作用时，需要修正操作符。

4. 对每个变量x，F 中存在一个函数 f，使得对每个 μ，f(μ) = ν，其中，对 x 之外的所有 w，ν(w) = μ(w)，且ν(x) = *noncanst*。该函数反映的是通过读 x 对 x 进行定义。在读语句之后，必须假定x不具有任何常量值。 □

10.11.2 数据流分析框架的公理

为了让我们讨论的各种数据流算法运行于任意框架，需要给出了对集合 V、集合 F 和与操作符∧的一些假设。虽然对于某些数据流算法，还需要额外的假设，但基本的假设如下： 682 ~ 683

1. F 含有恒等函数 I，使得对 V 中所有的 μ，I(μ) = μ。

2. F 在合成运算下封闭，即对 F 中的任意两个函数 f 和 g，由h (μ) = g (f(μ)) 定义的函数 h还在 F 中。

3. ∧操作满足交换律、结合律和幂等律。对 V 中所有的 μ，ν 和 ρ，这三个性质的代数表示如下：

$$\mu \wedge (\nu \wedge \rho) = (\mu \wedge \nu) \wedge \rho$$
$$\mu \wedge \nu = \nu \wedge \mu$$
$$\mu \wedge \mu = \mu$$

4. V 中存在一个栈顶元素 ⊤，满足规则：对V中所有的 μ，⊤ ∧ μ = μ。

例10.43 考虑到达定义。F 含有恒等函数，即 *gen* 和 *kill* 都是空集的函数。为证明 F 在

合成运算下封闭，设有如下两个函数：

$$f_1(X) = G_1 \cup (X - K_1)$$

$$f_2(X) = G_2 \cup (X - K_2)$$

那么

$$f_2(f_1(X)) = G_2 \cup ((G_1 \cup (X - K_1)) - K_2)$$

可以验证，上面等式的右部代数上等于

$$(G_2 \cup (G_1 - K_2)) \cup (X - (K_1 \cup K_2))$$

如果我们令 $K = K_1 \cup K_2$，$G = (G_2 \cup (G_1 - K_2))$，则我们已经证明：$f_1$ 和 f_2 的合成，即 $f(X) = G \cup (X - K)$，具有 F 成员的形式。

在这里，与操作符为并，容易证明并操作满足交换律、结合律和幂等律。因为对任意的集合 X，$\varnothing \cup X = X$，所以“栈顶”元素为空集。

考虑可用表达式时，使用和到达定义同样的参数，同样能够证明 F 含有恒等函数，并在合成运算下封闭。在这里，与操作符是交，容易证明交操作满足交换律、结合律和幂等律。因为对任意的表达式集合 X，$E \cap X = X$，则栈顶元素是程序中所有表达式的集合 E。 □

例10.44 考虑例10.42中介绍的常量计算框架。函数集合 F 中含有恒等函数，而且在合成
684 运算下封闭。检验 $\wedge$ 是否满足交换律、结合律和幂等律，只需证明它们对每个变量 x 都适合。作为一个实例，我们检验其幂等性。令 $\nu = \mu \wedge \mu$，即对所有的 x，$\nu(x) = \mu(x) \wedge \mu(x)$。容易验证，$\nu(x) = \mu(x)$。例如，图10-58中 *nonconst* 和它自身配对的结果还是 *nonconst*，则 $\mu(x) =$ *nonconst* 时，有 $\nu(x) =$ *nonconst*。

最后，栈顶元素是映射 τ，对所有的变量x，$\tau(x) =$ *undef*。由图10-58可以验证，对任意映射 μ 和任意变量 x，如果 $\nu = \tau \wedge \mu$，则 $\nu(x) = \mu(x)$。图10-58中 *undef* 和任何变量配对的结果都是另外的值。 □

10.11.3 单调性和分配性

为使数据流分析的迭代算法运行起来，我们还需要单调性的条件，即如果从集合 F 中取出任意的一个函数 f，并将 f 应用到 V 中的两个成员上，其中一个成员“较大”于另一个，那么将 f 应用到较大成员所得到的结果不小于将 f 应用到较小成员所得到的结果。

为精确定义“较大”这个概念，我们定义 V 上的关系 $\leqslant$：

$\mu \leqslant \nu$ 当且仅当 $\mu \wedge \nu = \mu$

例10.45 在到达定义框架中，与操作符是并，V 的成员是定义集合，$X \leqslant Y$ 意味着 $X \cup Y = X$，也就是说，X 是 Y 的超集。因而，$\leqslant$ 看起来是“后向的”：V 的较小元素是较大元素的超集。

对于可用表达式，与操作符是交，V 的成员是表达式集合，$X \leqslant Y$ 意味着 $X \cap Y = X$，即 X 是 Y 的子集。 □

从例10.45中可以看出，$\leqslant$ 不需要具有 $\leqslant$ 在整数上的所有性质。$\leqslant$ 是传递的，作为练习，利用 $\wedge$ 的公理，读者可以证明如果 $\mu \leqslant \nu$ 且 $\nu \leqslant \rho$，则 $\mu \leqslant \rho$。但我们感觉 $\leqslant$ 不是全序的。例如，在可用表达式框架中，存在两个集合 X 和 Y 互不为子集，这种情况下，$X \leqslant Y$ 和 $Y \leqslant X$ 都不成立。

用*栅格图*画出集合 V 通常很有帮助。栅格图中，节点为 V 中元素，边的方向都是向下的。如果 $Y \leqslant X$，则有一条从 X 到 Y 的边。例如，图10-59给出了到达定义数据流问题中的集合 V，

共有三个定义 d_1，d_2 和 d_3。因为≤意指“是……的超集”，于是一条边就从这三个定义的任意子集指向其超集。因为≤是传递的，如果在图中还存在另一条从 X 到 Y 的路径，则我们照惯例删除从 X 到 Y 的边。所以，尽管$\{d_1, d_2, d_3\} \leqslant \{d_1\}$，但它可以由穿过$\{d_1, d_2\}$的路径来表示，所以我们没有画出这条边。

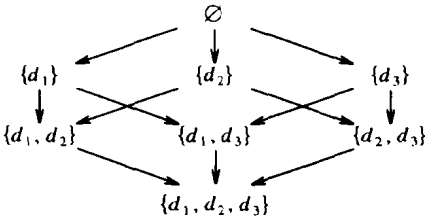


图10-59 定义子集的栅格

我们注意到，因为 Z 总是 $X \wedge Y$ 的超集，从 X 和 Y 都有路径指向 Z，所以可以从图上直接读出与操作的结果。例如，因为与操作符是并，如果 X 是$\{d_1\}$，Y 是$\{d_2\}$，则图10-59中的Z 是$\{d_1, d_2\}$。栈顶元素将出现在栅格图的最顶部，即从 $\top$ 到每一个元素都存在一条路径。

如果对 V 中所有的 μ 和 ν，以及 F 中的 f，

$$\text{当 } \mu \leqslant \nu \text{ 成立时，} f(\mu) \leqslant f(\nu) \text{也成立} \tag{10-15}$$

则定义框架 $(F, V, \wedge)$为单调的。

一种等价的定义单调性的方法是，对 V 中所有的 μ 和 ν，以及 F 中的 f，

$$f(\mu \wedge \nu) \leqslant f(\mu) \wedge f(\nu) \tag{10-16}$$

有时交换使用两个等价定义会非常有用。利用≤的定义以及∧的结合律、交换律和幂等律，我们简单给出两个定义的等价性证明，而将一些简单的结论留给读者去验证。

假定式(10-15)成立，证(10-16)成立。首先注意到，对任意的 μ和 ν，$\mu \wedge \nu \leqslant \mu$ 和 $\mu \wedge \nu \leqslant \nu$ 都成立。有一个简单的证明留给读者，即证明对任意的 x 和 y，$(x \wedge y) \wedge y = x \wedge y$。从而，由(10-15)，$f(\mu \wedge \nu) \leqslant f(\mu)$且$f(\mu \wedge \nu \leqslant f(\nu)$。存在法则：如果 $x \leqslant y$ 且$x \leqslant z$ 成立，则 $x \leqslant y \wedge z$ 也成立。它的证明留给读者去完成。令$x = f(\mu \wedge \nu)$，$y = f(\mu)$，$z = f(\nu)$，就会得到(10-16)。

反之，假设(10-16)成立，证(10-15)成立。假设 $\mu \leqslant \nu$，利用(10-16)证明 $f(\mu) \leqslant f(\nu)$ 即可证明(10-15)。方程(10-16)告诉我们 $f(\mu \wedge \nu) \leqslant f(\mu) \wedge f(\nu)$。但因为已经假设 $\mu \leqslant \nu$，由定义，$\mu \wedge \nu = \mu$。于是，根据(10-16)，$f(\mu) \leqslant f(\mu) \wedge f(\nu)$ 成立。作为通用规则，读者可以证明：

如果 $x \leqslant y \wedge z$ 则 $x \leqslant z$

从而，由(10-16)可以导出 $f(\mu) \leqslant f(\nu)$，于是我们证明了(10-15)成立。

通常，一个框架要遵守比(10-16)更强的条件，我们称之为分配性条件，对 V 中所有的 μ 和 ν，以及 F 中的 f，

$$f(\mu \wedge \nu) = f(\mu) \wedge f(\nu)$$

685
~
686

当然，如果 $x = y$，那么根据幂等律有 $x \wedge y = x$，所以 $x \leqslant y$。因此，由分配性可以导出单调性。

例10.46 考虑到达定义框架。令 X 和 Y 为定义集合，f 是由$f(Z) = G \cup (Z - K)$ 定义的函数，G 和 K 是定义集合。通过检验下式，可以验证到达定义框架满足分配性条件：

$$G \cup ((X \cup Y) - K) = (G \cup (X - K)) \cup (G \cup (Y - K))$$

尽管该关系看起来很复杂，但画出文氏图可以使其证明很显然。 □

例10.47 让我们证明常量计算框架是单调的，但不具有分配性。首先，将∧操作符和≤关系应用到出现在图10-58表中的元素上。也就是说，定义：

$nonconst \wedge c = nonconst$ 对任意常量 c

$c \wedge d = nonconst$ 对常量 $c \neq d$

$c \wedge undef = c$ 对任意常量 c

$nonconst \wedge undef = nonconst$

$x \wedge x = x$ 对任何值 x

那么图10-58可以解释为 $\rho(a) = \mu(a) \wedge \nu(a)$。

从操作符 $\wedge$ 的定义可以确定值上的关系 $\leqslant$。我们发现：

$nonconst \leqslant c$ 对任意常量 c

$c \leqslant undef$ 对任意常量 c

$nonconst \leqslant undef$

该关系如图10-60中的栅格图所示，其中 c_i 表示所有可能的常量。图中表示的是单独变量a的μ(a)值的集合上的关系，而不是 V 的元素上的≤。V 的元素可以被看成是这种值的向量，一个分量对应着一个变量。V 的栅格图可以从图10-60中分离出来，只要记住，$\mu \leqslant \nu$ 成立当且仅当对所有的 a，$\mu(a) \leqslant \nu(a)$成立，即表示 μ 和 ν 的向量的每个分量通过≤联系在一起，并且每个分量中上关系的方向是一致的。

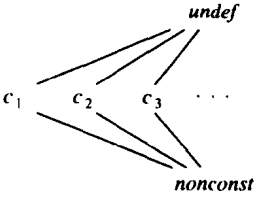


图10-60 变量值的栅格图

因此，如果 $\mu \leqslant \nu$，则只要 μ(a) 是常量c，ν(a) 或者是常量或者是 *undef*；只要 μ(a)是*undef*，ν(a)也是 *undef*。通过检查与不同类型的定义语句相关联的函数，可以验证：如果 $\mu \leqslant \nu$，则 $f(\mu) \leqslant f(\nu)$。从而证明了(10-15)，也证明了单调性。例如，如果 f 和语句a := b+c相关联，只有μ(a)和ν(a)发生变化，我们必须检查如果 $\mu \leqslant \nu$，即对所有的x，$\mu(x) \leqslant \nu(x)$，那么$[f(\mu)](a) \leqslant [f(\nu)](a)$[⊖]。我们必须考虑 μ(b)、μ(c)、ν(b)和ν(c) 所有可能的值，使之服从约束 $\mu(b) \leqslant \nu(b)$和$\mu(c) \leqslant \nu(c)$。例如，如果

$\mu(b) = nonconst$
$\nu(b) = 2$
$\mu(c) = 3$
$\nu(c) = undef$

那么$[f(\mu)](a) = nonconst$ 且$[f(\nu)](a) = undef$。因为$nonconst \leqslant undef$，已经验证了第一种情况，其他情况作为练习留给读者。

现在，必须检验常量计算框架不具有分配性。为此，令函数 f 与赋值语句 a := b+c 相关联，且令 $\mu(b) = 2$，$\mu(c) = 3$，$\nu(b) = 3$，$\nu(c) = 2$。令$\rho = \mu \wedge \nu$。于是 $\mu(b) \wedge \nu(b) = 2 \wedge 3 = nonconst$。类似地，$\mu(c) \wedge \nu(c) = nonconst$。等价地，$\rho(b) = \rho(c) = nonconst$。因为我们认为两个不是常量的值其和也不是常量，可以断定$[f(\rho)](a) = nonconst$。

另一方面，因为假设 b = 2且c = 3，则赋值语句a := b+c将a置为5，所以有$[f(\mu)](a) = 5$。类似地，$[f(\nu)](a) = 5$。所以$[f(\mu) \wedge f(\nu)](a) = 5$。现在我们看到 $\rho(a) = [f(\mu \wedge \nu)](a) \neq [f(\mu) \wedge f(\nu)](a)$，从而违反了分配性条件。

直观地，违反分配性的原因是常量计算框架没有记住所有的不变量。特别地，尽管 b 和 c 本身都不是常量，沿 μ 或 ν 描述的路径，方程 b+c=5 仍然成立。我们可以设计更复杂的框架来避免该问题，尽管这样做的收效还不清楚。幸运的是，正如下面将会看到的，单调性足以让迭代数据流算法正常运转。 □

687 ~ 688

⊖ 读$[f(\mu)](a)$这样的表达式时必须小心。它的意思是：将 f 应用到μ得到某个映射$f(\mu)$，我们称其为μ′。然后再将μ′应用到a，结果是图10-60所示的图中的一个值。

10.11.4 数据流问题的聚合路径解

设想流图中的每一个节点都和集合 F 中的一个转移函数相关联。对每个块 B，令 f_B 为 B 的转移函数。

考虑从初始节点 B_0 到某个块 B_k 的任意路径 $P = B_0 \rightarrow B_1 \rightarrow \cdots \rightarrow B_k$。可以将 P 的转移函数定义为 f_{B_0}，f_{B_1}，…，$f_{B_{k-1}}$ 的合成。注意，因为该路径到达块 B_k 的开始而不是其末尾，所以 f_{B_k} 不是该合成的一部分。

已经假设V中的值表示程序中引用的数据的信息，聚合操作符 $\wedge$ 则说明当路径汇聚时信息的结合方式。因为路径汇合之后，带有栈顶元素的路径至多向其他路径输出它所携带的信息，所以可以认为栈顶元素表示“没有信息”。因而，如果 B 是流图中的块，通过考虑初始节点到B的每一条可能的路径，从“没有信息”开始，查看路径上发生的事情，则进入 B 的信息应该是可计算的。也就是说，对 B_0 到 B 的每条路径 P 计算 $f_P(\top)$，并取所有的结果元素的“与”。

原则上，因为存在无穷条不同的路径，结果“与”可以有无穷的不同值。实际上，只需考虑无环路径，即使关于上面讨论的常量计算框架不是无环路径，通常也可以找到其他的依据，使得特定的流图的结果的“与”是有穷的。

流图的聚合路径解（meet-over-paths solution）的形式化定义为：

$$mop(B) = \bigwedge_{B_0 \text{到} B \text{的路径} P} f_P(\top)$$

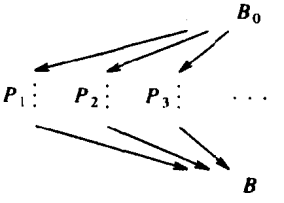


图10-61 说明到B的所有可能路径的图

对到达块 B 的信息的计算，聚合路径解是有意义的。流图和图10-61相同，其中，每条路径（数量可能无穷）P_1，P_2，…相关的转移函数，在原始流图中被赋予一条到B的完全独立的路径。在图10-61中，到达B的信息由所有聚合路径给出。

10.11.5 流问题的保守解

当求解来自于任意框架的数据流方程时，或许 *mop* 解不易得到。幸运的是，当处理10.5节和10.6节中的数据流框架的具体示例时，错误出现在一个安全的范围内，并且数据流迭代算法总能提供一个安全解。如果对所有的块 B，$in[B] \leqslant mop[B]$，则称 $in[B]$是安全解。

不管读者怎么想，到目前为止，我们还没有把定义完全说清楚。回想在任意流图中，到一个节点的表观路径（流图中的路径）集合可以是真实路径（流图对应程序的某次执行所采用的路径）的真子集。因为通常我们不能判别真实路径和表观路径，所以为了使数据流分析的结果有用，在保证数据安全的前提下，需要通过删除某些路径修改流图。

假设在图10-61中的无穷路径集合中，x是某次执行的真实路径 P 的 $f_P(\top)$ 的“与”。同样地，令y是所有其他路径 P 的 $f_P(\top)$的“与”。因而，$mop\ (B) = x \wedge y$。于是在节点 B 我们的数据流问题的真实答案是x，但 *mop* 解却是 $x \wedge y$。因为$(x \wedge y) \wedge y = x \wedge y$，所以有 $x \wedge y \leqslant y$。因此，*mop* 解$\leqslant$真实解。

尽管我们更喜欢数据流问题的真实解，但我们几乎没有有效的办法来准确区分哪些路径是真实的，哪些不是，所以我们不得不接受将 *mop* 解作为最接近的可行解。对于任何数据流信息都必须满足“得到的解$\leqslant$真实解”。一旦我们接受了这一点，我们也应该能够接受一个$\leqslant$ *mop*（从而$\leqslant$真实解）的解。这样的解比单调非分配框架的 *mop* 更容易得到。对于10.6节中的分配框架，用简单的迭代算法计算 *mop* 解。

10.11.6 通用框架的迭代算法

对算法10.2进行推广，可以使其运行在大多数的框架上。该迭代算法要求框架是单调的，
689 ~ 690 而且要求图10-61中路径的无穷集合的“与”等价于有穷子集的“与”。下面给出该算法，并讨论如何确保其有穷性。沿无环路径传播可以保障有穷性。

算法10.18 通用数据流框架的迭代解。

输入：流图，值集合 V，函数集 F，“与”操作 $\wedge$，F 的成员对流图中每个节点的赋值。

输出：对流图的每个节点 B，给出 V 中的 $in[B]$值。

方法：算法如图10-62所示。与常见的数据流迭代算法类似，用逐次近似计算法计算每个节点的 in 和 out。假设 f_B 是 F 中与块 B 相关联的函数，该函数的作用与10.6节中 *gen* 和 *kill* 的作用相同。 □

```
(1)  for每个节点B do   /* 初始化，假设 in[B] = ⊤ */
(2)      out[B] := f_B[⊤];
(3)  while out发生变化 do
(4)      for 每个块B（按深度优先顺序）do begin
(5)          in[B] := ∧_{B的前驱P} out[P];
             /* 在上面，空集的与是 ⊤ */
(6)          out[B] := f_B(in[B])
         end
```

图10-62 通用框架的迭代算法

10.11.7 一个数据流分析工具

现在，让我们来看看如何将本节的思想应用到一个数据流分析工具中。算法10.18的运转依赖于下面的子程序：

1. 将 F 中一个给定的 f_B 应用到 V 中一个给定的值的程序，图10-62的第(2)行和第(6)行引用了该程序。

2. 将“与”操作符应用到 V 中两个值上的程序，这个程序在算法第(5)行中需要调用0次或多次。

3. 判断两个值是否相等的程序。这种测试在图10-62中没有明确给出，隐含在对 *out* 值的变化测试中。

为了将参数传递给上面提到的子程序，需要为 F 和 V 声明特定的数据类型。图10-62中 *in* 和 *out* 值也属于为 V 声明的类型。最后，还需要一个以基本块内容的一般表示（即语句列表）
691 为输入，以 F 的一个元素（即转移函数）为输出的程序。

例10.48 对到达定义框架，可以先建立一个表，用1到某个最大值 m 间的惟一整数值标识流图中的每条语句。V 的类型是长度为 m 的位向量，F 用同样大小的位向量对（即 *kill* 和 *gen* 集合）表示。给定某个块的语句以及位向量位置与定义语句的关联表，*gen* 和 *kill* 位向量的构造程序、计算“与”（位向量的逻辑或）的程序、位向量相等比较以及将 *gen-kill* 对应用到位向量的应用函数等都变得非常直接。 □

数据流分析工具比图10-62的实现要复杂一些。在图10-62中需要调用“与”操作、函数应用以及比较程序。数据流分析工具支持流图的固定表示，因此能够执行以下任务：找出某个节

点的所有前驱、找出流图的深度优先顺序或者对每个块应用F中与该块相关的函数。使用这样一个工具的优势在于，对于各种数据流分析，算法10.18的图操作和收敛检验不必重写。

10.11.8 算法10.18的性质

我们应当清楚，在何种假设下算法10.18能实际运转，以及算法收敛时会收敛在何处。首先，如果框架是单调的且收敛，则声明该算法的结果是：对所有的块B，$in[B] \leqslant mop(B)$。其原因是：在图10-62中，沿着初始节点到$B = B_k$的任何路径$P = B_0, B_1, \cdots, B_k$，通过对$i$归纳证明，最多$i$次 while 语句迭代之后，即可获知$B_0$到$B_i$的路径作用。也就是说，如果$P_i$的路径是$B_0, B_1, \cdots, B_i$，那么$i$次迭代之后，$in[B_i] \leqslant f_{P_i}(\top)$。因此，如果算法收敛，在$B_0$到$B$的每条路径上都有$in[B] \leqslant f_P(\top)$。利用规则“如果$x \leqslant y$且$x \leqslant z$，那么$x \leqslant y \wedge z$”[⊖]，我们可以证明$in[B] \leqslant mop(B)$。

如果框架是分配的，可以证明算法10.18会收敛到 *mop* 解。基本思想是证明：在算法运行期间，证明$in[B]$和$out[B]$分别等于到达B的起始点和到达B的结束点的路径集合P的$f_P(\top)$的“与”。但是，当框架是单调的但不具有分配性时，情况未必如此，见下例。

例10.49 研究例10.47中讨论的常量计算框架，该框架不具有分配性，相关的流图如图10-63所示。映射 μ 和 ν 来自于例10.47的B_2和B_4。进入B_5的映射 ρ 等于 μ∧ν。σ 是从B_5出来的映射。尽管每一条真实路径（以及每一条表观路径）在B_5之后都计算出 a=5，我们还是置 a 为 *nonconst*。 692

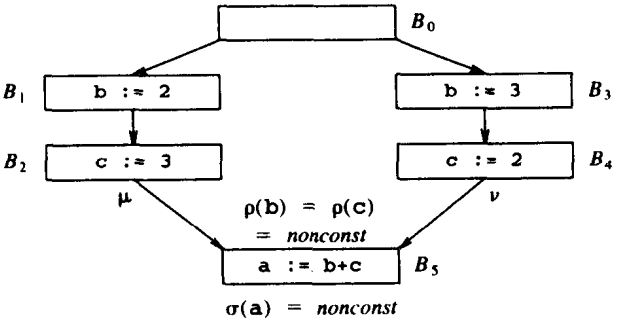


图10-63 比 *mop* 解少的解的例子

直观地，存在如下问题：算法10.8在处理不具有分配性的框架时，会将一些甚至不是表观路径（流图中的路径）的节点序列认为是真实路径。因而，在图10-63中，算法执行时将$B_0 \rightarrow B_1 \rightarrow B_4 \rightarrow B_5$或$B_0 \rightarrow B_3 \rightarrow B_2 \rightarrow B_5$这样的路径认为是真实路径，从而将 b+c 置为 b 值和 c 值的组合，而不是置为5。 □

10.11.9 算法10.18的收敛性

很多方法可以证明算法10.18对特定的框架收敛。最常见的情况可能是只需要无环路径的情况，即可以证明无环路径的“与”和所有路径上的 *mop* 解相同。如果是这种情况，该算法不仅收敛，而且通常收敛得非常快。正如10.10节所讨论的，使用流图的深度加2遍就可以得到结果。

另一方面，在常量计算框架中，不能只考虑无环路径。例如，图10-64给出了一个简单流图，其中，必须考虑路径$B_1 \rightarrow B_2 \rightarrow B_2 \rightarrow B_3$，从而认识到x在进入$B_3$时不是常量。

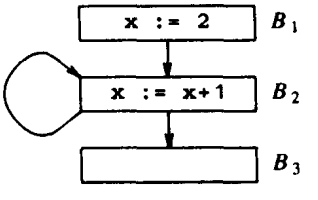


图10-64 需要将环路包含在 *mop* 中的流图

⊖ 这里有一个技术上的细节：必须证明该规则不只适合两个值 y 和 z（即如果$x \leqslant y_i$，则对y_i的任何有穷集合，$x \leqslant \wedge_i y_i$），而且对无穷数目的y_i，该规则同样成立。实际上，当算法10.18收敛时，我们将发现有穷条路径使得所有路径的“与”等于这有穷条路径的“与”。

然而，对于常量计算，可按照如下方法推出算法10.18收敛。首先，易证对任意的单调框架和任意的块 B，$in[B]$ 和 $out[B]$ 形成一个不增序列，即这些变量的新值≤旧值。回想图10-60中映射值的栅格图，我们认识到对任意的变量，$in[B]$ 或 $out[B]$ 的值只能下降两次，一次是从 *undef* 到一个常量，一次是从该常量到 *nonconst*。

设有 n 个节点和 v 个变量，在图10-62的 while 循环的每次迭代中，$out[B]$ 中至少有一个变量的值发生变化，否则，算法已经收敛，$in[B]$ 或 $out[B]$ 的值将不再发生改变。因此，迭代次数限制为 $2nv$。发生$2nv$ 次改变之后，流图中的每个块的变量必定都已到达 *nonconst*。

10.11.10 初始化的修正

在某些数据流问题中，算法10.18给出的解同我们直觉上想要的解之间存在偏差。回想一下在处理可用表达式时，∧是交操作，所以 T 必须是所有表达式的集合。因为算法初始时假设对每个块 B（包含初始节点），$in[B]$ = T，所以，假设初始节点是可用的，算法10.18产生的 *mop* 解实际上是块 B 的入口处可用表达式的集合，但实际上初始节点表达式是不可用。

差别是，可能存在一条初始节点到B的路径，使得表达式 `x+y` 在该路径上既不会被产生，也不会被注销。算法10.18认为`x+y`是可用的，但事实上，因为沿该路径找不到任何变量保存它的值，表达式是不可用的。修正操作非常简单。我们可以修改算法10.18使得在可用表达式框架中，将 $in[B_0]$ 置为空集并保持为空集，或者在流图中引入一个注销所有表达式的虚初始节点，该虚初始节点是真正初始节点的前驱。

10.12 类型估计

现在我们来考虑另外一种数据流问题，它比前一节的框架更具挑战性。在APL、SETL和
693 ~ 394 许多Lisp语言中，不需要声明变量的类型，甚至允许同一个变量在不同时间保存不同类型的值。因为将两个整数相加的代码比调用一个将两个不同类型（比如整型、实型、向量类型）的对象相加的通用程序效率要高，想要真正地将这些语言编译成高效的代码，需要使用数据流分析去推断变量的类型。

最初，我们猜测变量类型估计与到达定义计算有些相像。在特定点上，可以将一组类型与变量联系起来。聚合操作符是对数据类型集合的并。如果变量x在一条路径上可能的类型的集合为 S_1，在另一条路径可能的类型的集合为 S_2，那么x在路径汇合之后就可能是集合 $S_1 \cup S_2$ 中的任何一种类型。当控制穿过一条语句时，基于语句中的操作符、操作对象可能的类型，以及结果的类型，能够对变量类型做出推断。例6.6就是这种推断的一个例子，它处理一个将整型数和复型数相乘的操作符。

但是，这种方法至少还存在两个问题：

1. 一个变量的可能类型集合也许是无穷的。

2. 为获得可能类型的准确估计，类型判定通常既需要前向信息传播，又需要后向信息传播。因此，为充分地评判该问题，即使10.11节的框架也不能够完全满足要求。

在考虑第1点之前，让我们先考察一下常见的语言中对类型做出的一些推断。

例10.50 考虑下面的语句：

```
i := a[j]
k := a[i]
```

假设起初我们对变量a，i，j，k的类型一无所知，但是让我们假设数组访问操作符[]要

求参数为整型。通过检查第一条语句，我们可以推断出j在该点是一个整数，而且a是某种类型元素的数组。然后，第二条语句告诉我们i是整数。

现在，我们可以向后传播这些推断。如果i在第一条语句中被计算为一个整数，那么表达式a[i]的类型一定是整型，这意味着a必定是一个整数数组。然后再向前推导我们可以发现由第二条语句赋给k的值一定也是个整数。注意，如果只是向前推导或只是向后推导，都不能发现a的元素为整型。 □

10.12.1 处理无穷类型集

存在大量超出常规的情况，这时一个变量的可能类型集合是无穷的。例如，SETL语言允许在循环内执行 x := {x}这样的语句。如果起初只知道x可能是个整数，那么在循环的一次迭代之后，我们意识到x可能是个整数，也可能是个整数的集合。在第二次迭代之后，又发现 x 可能是个整数的集合的集合，依此类推。 695

传统的C语言没有类型声明，也可能发生类似的问题。像 x=&x 这样的语句（起初x可能是个整数）将导致我们发现x可能是如下形式的任何一种类型：

指向整数的指针的指针…的指针

处理这类问题的传统做法是，将变量的可能类型集合减少为有穷的数目。一般地，将无穷的可能类型划分成只保留较简单类型的有穷类，将特别复杂的类型（希望也是最少的）划分到大的类别中。如果这么做，必须知道如何对类型和操作符之间的相互作用做出推断。下面的例子将告诉我们怎么做。

例10.51 继续考虑第6章的例子，当时用操作符→作为函数的类型构造符。在这里，类型集合包括基本类型*int*，以及形如 $\tau \rightarrow \sigma$ 的所有类型。$\tau \rightarrow \sigma$ 代表定义域类型为 τ、值域类型为 σ 的函数类型，其中 τ 和 σ 都是类型集合中的类型。因此，本例的类型集合是无穷的，它可能包括下面这样的类型：

$$(int \rightarrow int) \rightarrow ((int \rightarrow int) \rightarrow int)$$

为了将类型集合的种类减少到有穷数目，通过用名字 *func* 取代至少包含一个→的表达式中的子表达式，我们限制一个类型表达式只有一个函数类型构造符→。因而只有5种不同的类型：

int
int → *int*
int → *func*
func → *int*
func → *func*

用长度为5的位向量表示类型集合，各位分别对应上面列出的5种类型。因此，01111代表除整型以外的类型。因为 *func* 不可能是整型，在某种意义上说，01111代表了 *func* 类型。

在该模型中基本的赋值语句为：

```
x := f(y)
```

696

知道了f和y的可能类型，可以通过查找图10-65的表确定x的类型。如果f是集合 S_1 中的任一类型，y是集合 S_2 中的任一类型，在 S_1 和 S_2 中分别取 τ 和 σ 进行组合，然后查找以τ为行，以 σ 为列的表项（我们将其称为 $\tau(\sigma)$），最后将所有查找结果并起来可得到x的可能类型集合。

例如，如果 $\tau = int \rightarrow func$ 且 $\sigma = int$，则 $\tau(\sigma) = 01111$。也就是说，$int \rightarrow func$ 类型到 *int* 类型的映射结果是 *func*，*func* 是除 *int* 之外的四种类型之一。因为将无穷种类型划分为5类的模糊不

清的做法妨碍了我们的认识，我们不能区分 *func* 到底是四种类型中的哪一个。

τ	σ				
	int	*int→int*	*int→func*	*func→int*	*func→func*
int	00000	00000	00000	00000	00000
int→int	10000	00000	00000	00000	00000
int→func	01111	00000	00000	00000	00000
func→int	00000	10000	10000	10000	10000
func→func	00000	01111	01111	01111	01111

图10-65 τ(σ)的值

再例如，令 τ 取先前的值而 σ = *int→int*，此时 τ(σ) = 00000，因为 τ 的定义域类型与 σ 的类型一定不相同，因此不能应用映射。 □

10.12.2 一个简单的类型系统

为阐明类型推断算法的思想，我们介绍一个简单的类型系统和基于例10.51的语言。类型是例10.51中说明的那5种，语言中的语句有如下3种：

1. `read x`。x的值通过输入设备读入，且假定对它的类型一无所知。

2. `x := f(y)`。x的值通过在值y上应用函数f得到，图10-65中总结了赋值后x可能的类型。

3. `use x as τ`。当我们穿越这样的语句时，可以假设程序是正确的。因而，在该语句之前和之后，x的类型都只能是τ。x的值和类型不受该语句的影响。

697 对于由这三种类型的语句构成的程序，通过在其流图上执行数据流分析来推断变量类型。为简单起见，假设所有的块都只包含一条语句。块的 *in* 和 *out* 值是从变量到例10.51中的5种类型的集合的映射。

起初，每个 *in* 和 *out* 都将变量映射到5种类型的集合。随着信息的传播，在某些点上将减少与某些变量相关联的类型集合，直到不能再减少为止。最后的集合将指出在每个点上每个变量的可能类型。因为只有能证明（假设程序正确）某个类型是不可能的时，才删除它，因此这种假设是保守的。一般地，我们希望利用“某些类型是不可能的”的事实，而不是利用“某些类型是可能的”事实去进行推断。相对于错误而言，“太大”是一种安全的方向。

采用“前向”方案和“后向”方案来修改 *in* 和 *out*。前向方案使用块 *B* 中的语句和 *in*[*B*] 的值约束 *out*[*B*][⊖]，后向方案正好相反。在每种方案中，聚合操作符都是 “合理变量的并”，意即，对所有的变量x，两个映射 α 和 β 的聚合映射 γ 满足：

$$\gamma[x] = \alpha[x] \cup \beta[x]$$

10.12.3 前向方案

假设有一个块 *B*，*in*[*B*]中的映射 μ，以及 *out*[*B*]中的映射 ν。前向方案要求约束 ν。约束 ν 的规则依赖于块 *B* 中的指令。

1. 如果语句是 `read x`，此时各种类型都可能被读入。如果在读之后已知 x 的类型，那么在这次前向传播中记住 x 的类型，无须改变 ν(x)。对所有的其他变量 y，置：

⊖ 值得注意的是，在传统的前向数据流方案中，我们没有约束*out*，每次从*in*重新计算*out*。之所以可以这样做是因为*in*和*out*总是沿相同的方向改变，即要么总是增长，要么总是收缩。但是，在类型推断问题中，我们或者执行前向传播，或者执行后向传播，可能存在这种情况：后向传播得到的*out*比对*in*应用前向规则证明得到的*out*要小。于是，我们在前向传播时不必提高*out*，只是在后向传播时再次降低即可。类似地，后向传播时我们必须约束*in*而不是重新计算它。

$\nu(y) := \nu(y) \cap \mu(y)$

2. 假设语句是use x as τ。经过该语句之后，τ是x惟一可能的类型。如果已经知道x不可能是类型τ，那么经过该语句之后，x就没有可能的类型。将这些结论概括为： 698

$\nu(x) := \nu(x) \cap \{\tau\}$
$\nu(y) := \nu(y) \cap \mu(y)$ 如果 $y \neq x$

3. 考虑语句 x := f(y)。经过该语句之后x惟一可能的类型是：

i) 可能依赖 ν 的当前值，并且

ii) 是将某个类型 τ 的映射应用到类型 σ 上的结果，τ 和 σ 分别是该语句执行之前f和y可能具有的类型。

形式化为：

$\nu(x) := \nu(x) \cap \{\rho \mid \rho = \tau(\sigma),\ \tau在\mu(f)中,\ \sigma在\mu(y)中\}$

还可以对f和y的类型做出推断，因为假设程序是正确的，如果f不能应用到某种数据类型上，则不可能具有该类型，当y不能作为f的参数类型，则y不能具有该类型。也就是说，如果 $f \neq x$，则

$\nu(f) := \nu(f) \cap \{\tau \mid 在\mu(f)中，而且对\mu(y)中的某个\sigma，\tau(\sigma) \neq \varnothing\}$

如果 $y \neq x$，则

$\nu(y) := \nu(y) \cap \{\sigma \mid \sigma在\mu(y)中，而且对\mu(f)中的某个\tau，\tau(\sigma) \neq \varnothing\}$

对所有的其他z，

$\nu(z) := \nu(z) \cap \mu(z)$

10.12.4 后向方案

现在考虑在后向方案中，如何根据 ν 和语句的传播信息来约束 μ。

1. 如果语句是 read x，则容易看出在该语句执行之前，不能推断出新的不可能类型，所以μ(x)不会改变。但是对所有的 $y \neq x$，可以通过置 $\mu(y) := \mu(y) \cap \nu(y)$ 来向后传播信息。

2. 如果语句是 use x as τ，则可以做出与前向方案相同的推断；在该语句前，x只能有类型 τ，在该语句之前和之后都可能出现的是其他变量类型。也就是说，

$\mu(x) := \mu(x) \cap \{\tau\}$
$\mu(y) := \mu(y) \cap \nu(y)$ ，对于 $y \neq x$

3. 和前面一样，最复杂的情况就是形如x := f(y)的语句。开始，在语句之前，除非x碰巧是某个f或y，否则不能推断出x的新信息。因此，除非使用下面涉及f和y的规则，否则μ(x)不会改变。接下来，与前向规则中一样，可以根据以下事实做出推断：在该语句之前f和y的类型必须兼容。但是如果 $f \neq x$，则可将 μ(f)限制为ν(f)中的类型，关于y有一个类似的句子成立。另一方面，如果f=x，那么在该语句之后f的类型就和该语句之前f的类型没有联系，故不允许 699
这样的约束。如果y=x，则有一个类似的句子成立。为f和y定义一个特殊的映射将有助于反映该结果，于是我们定义：

如果 f = x，则 $\mu_1(f) := \mu(f)$，否则 $\mu_1(f) := \mu(f) \cap \nu(f)$
如果 y = x，则 $\mu_1(y) := \mu(y)$，否则 $\mu_1(y) := \mu(y) \cap \nu(y)$

现在可以将f和y限制成与其他类型集合兼容的类型。同时可以基于下面事实来限制f和y的类型，即它们不仅是兼容的，而且必须输出 ν 所确定的x的可能类型。因此，我们定义：

$\mu(f) := \{\tau \mid \tau在\mu_1(f)中，而且对\mu_1(y)中的某个\sigma，\tau(\sigma) \cap \nu(x) \neq \varnothing\}$

$\mu(y) := \{\sigma | \sigma$在$\mu_1(y)$中，而且对$\mu_1(f)$中的某个$\tau$，$\tau(\sigma) \cap \nu(x) \neq \varnothing\}$

$\mu(z) := \mu(z) \cap \nu(z)$，对不等于x、y或f的z

继续类型确定算法之前，回想一下10.5节关于到达定义的讨论。如果开始时错误地假定某个定义d在一个循环中的某处是可用的，那么将把这个事实沿着循环错误地传播，最终导致到达定义集合比必需集合大。类似的问题也存在于类型确定中，在一个循环中，关于某个变量可能具有某种类型的假设将被它自身“证明”。所以除了例10.51的32个类型集合之外，我们要引入第33个值，即*undef*。映射μ可以将值*undef*赋给一个变量。*undef*的作用与其常量传播框架中的作用相似。

在聚合过程中，值*undef*服从于其他任何值，就像类型00000一样。另一方面，当对类型集合执行交操作时，如计算$\mu(x) \cap \nu(x)$，值*undef*仍服从于其他类型集合，其功能就像11111类型。举例来讲，当我们读一个变量x的值时，读之后x的“类型”被看作是*undef*的事实被否决，而且x的类型变成11111。

算法10.19

输入：一个流图，它的块由三种类型（读、赋值和use-as）的单个语句组成。

输出：在每个点的每个变量的类型集合。这个集合是保守的，即任何一种真实的运算必将导致该集合中的一个类型。

方法：为每个块B计算映射$in[B]$和$out[B]$，每个映射都将程序的变量发送到例10.51中引入的类型系统中的类型集合中。初始时，所有的映射都将每个变量发送到*undef*。

700

接着我们交替采用前向传播和后向传播穿越流图，直到连续的前向和后向传播均不再产生任何变化为止。前向传播执行：

```
for 深度优先顺序中的每个块B do begin
    in[B] := ∪ out[P];   (B的前驱P)
    out[B] := 前面定义的in[B]和out[B]的函数
end
```

后向传播执行：

```
for 逆深度优先顺序中的每个块B do begin
    out[B] := ∪ in[S];   (B的后继S)
    in[B] := 前面定义的in[B]和out[B]的函数
end
```

□

例10.52 考察图10-66中所示的一个简单的直线式（无转移）程序，我们对四个映射感兴趣，分别命名为μ_1到μ_4。每个μ_i既是$out[B_i]$又是$in[B_{i+1}]$。因为本节我们假设每个块都是单语句，B_1不能包含两个语句。因为所有变量在B_1结束前可以具有任意类型，所以不必关心在B_1结束前所发生的事情。

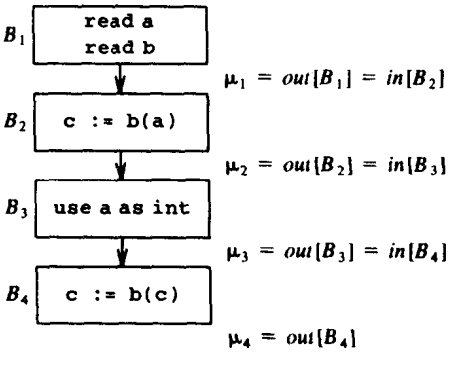


图10-66 程序示例

这表明在收剑发生前我们需要5遍，还需要两遍去检测收敛的发生，如图10-67a至图10-67e所示。第一遍是前向的。当考虑B_2时，我们发现b不能是整数，因为它已经被用作映射，还发现a在B_3中被用作整数，因此在μ_3和μ_4中只

能被映射为 *int*，这些结论被总结在图10-67a中。 701

如图10-67b所示，第二遍是后向的。在这一遍中，当考虑 B_2 时，我们知道将b应用到a时，a一定是整数。从而，b的类型只能是 *int→int* 或 *int→func*。在第三遍（前向传播）中，对b的类型的约束沿流图的所有路径向下传播，如图10-67c所示。

	a	b	c
μ_1	11111	11111	*undef*
μ_2	11111	01111	11111
μ_3	10000	01111	11111
μ_4	10000	01111	11111

a)

	a	b	c
μ_1	10000	01100	*undef*
μ_2	10000	01111	11111
μ_3	10000	01111	11111
μ_4	10000	01111	11111

b)

	a	b	c
μ_1	10000	01100	*undef*
μ_2	10000	01100	11111
μ_3	10000	01100	11111
μ_4	10000	01100	11111

c)

	a	b	c
μ_1	10000	01000	*undef*
μ_2	10000	01100	10000
μ_3	10000	01100	10000
μ_4	10000	01100	11111

d)

	a	b	c
μ_1	10000	01000	*undef*
μ_2	10000	01000	10000
μ_3	10000	01000	10000
μ_4	10000	01000	10000

e)

图10-67 算法10.19在图10-66的流图上的模拟执行

a) 前向 b) 后向 c) 前向 d) 后向 e) 前向

第四遍是后向的，如图10-67d所示。此时，在 B_4 中 c 是 b 的参数。这个事实告诉我们 c 只能是整数。考虑 B_2 时我们还发现 b（a）的结果只能是 c 的类型，即 *int*。这个事实排除了 b 是 *int→func* 类型的可能性。最后在图10-67e中，我们看到在第5遍（前向传播）中，关于 b 和 c 的这些事实是如何传播的。在下一遍中，不能做出新的推断，在这种情况下，我们已经将每一点的每个变量可能类型的集合减少到单个类型：a 和 c 是整型，b 是从整型到整型的映射。一般，我们也许会得到某一点的某个变量的几种可能类型。 □ 702

10.13 优化代码的符号调试

*符号调试器*是这样一个系统，它允许在程序运行时观察程序的数据。当程序运行出错，如发生溢出，或到达程序员在源代码中设定的某条语句时，通常会调用符号调试器。符号调试器一旦被调用，它将允许程序员检查变量，或者改变当前正被运行程序访问的变量。

为了使调试器能理解诸如 “给我显示a的当前值” 这样的用户命令，它必须具有某些可用信息：

1. 必须有某种方式将a标识符和它代表的地址联系起来。因此，编译器必须用符号表记录每个变量的分配位置，如全局数据区或某个过程活动记录中的一个位置。保存符号表并供调试器使用。例如，这些信息可被编码成程序的可加载模块。

2. 必须有作用域的信息，才能明确引用多次声明的标识符，并能分辨出在程序处于某一过程p时，其他过程中的哪些数据可以访问，以及如何在栈或其他运行时结构中找出这样的数据。这些信息必须从编译器的符号表中获取，并被保存以供调试器将来使用。

3. 调用调试器时我们必须知道程序正运行到什么位置。当编译器处理一个用户声明的调试器调用时，运行位置信息被编译器嵌入到对调试器的调用中。当运行时错误引起调试器被调用时，可以从异常处理程序中获得该信息。

4. 为了使用户能够获得3中的程序运行位置信息，必须有一个表，将机器语言语句和它的源代码语句联系起来。这个表可以在编译器产生代码时建立。

当考虑符号调试器设计的自身正确性时，我们只考虑为优化编译器编写符号调试器时出现的困难。乍一看，好像根本不需要调试一个优化后的程序。在正常的开发过程中，当用户调试程序时，通常使用快速的、非优化编译器，直到源程序正确为止，然后才使用优化编译器进行优化。

但是，对相同的输入数据，用非优化编译器编译后能正确运行的程序，用优化编译器编译后却不能正确运行。例如，优化编译器中可能有错误，或者在执行重排序操作时，优化编译器
703 可能引入上溢或下溢错误。而且，即使“非优化”编译器也可做一些简单的变换，如在基本块中清除局部公共子表达式或重排代码，而这些变换对符号调试器的设计难度非常重要。因此，对于可以随意变换基本块的优化编译器，设计符号调试器时需要考虑使用何种算法和数据结构。

10.13.1 基本块中变量值的推断

为简单起见，假定源代码和目标代码都是中间语句序列。因为中间代码比源代码更通用，所以把源代码看成中间代码不会出现问题。例如，用户可能只在源代码的语句之间设置断点（调用调试器），在这里，允许在任何中间语句之后设置断点。当优化器把单个中间语句断成几个单独的机器语句时，把目标代码看作是中间代码会有问题。例如，由于某种原因，把两条中间语句

```
u := v + w
x := y + z
```

编译成代码时，两个加法运算在不同的寄存器中执行并被交叉存取。如果发生这种情况，我们把寄存器看作是中间代码的临时变量。例如：

```
r1 := v
r2 := y
r1 := r1 + w
r2 := r2 + z
 u := r1
 x := r2
```

用户以为在执行源程序块，但实际上，正在执行的是块的优化版本。因此，当用户使用这样的块时，会产生若干问题：

1. 假定我们正在执行一个程序，该源程序的某个基本块已经被优化。当执行语句 a := b+c 时发生了溢出，调试器必须告诉用户在某条源语句中发生了一个错误。因为b+c可能是出现在两条或更多条源语句中的一个公共子表达式上，那么，该错误应归咎于哪条语句呢？

2. 如果调试器的用户想要察看某个变量d的当前值，则会出现更难的问题。在优化程序中，d可能最后在某个语句s中被赋值，但在源程序中，出现在调用调试器的语句之后的s，可能对d
704 赋值。因此，对调试器可用的d值并不是用户源代码列表中d的当前值。同样地，如果s出现在调用调试器的语句之前，但在源代码中，两者之间有另外一个对d的赋值，所以对调试器可用

的d值已经过时。能不能把d的正确值提供给用户呢？例如，d可否为优化代码中的其他变量的值，或是否可以从其他变量的值计算出来？

3. 最后，如果用户在源代码中的某条语句之后设置了断点，在执行优化代码时，应该在什么时候将程序的控制权交给调试器？

有一种解决方法是：同时运行基本块的未优化版本和优化版本，以使每个变量的正确值一直都是可用的。但我们拒绝这种解决方法，因为当引起问题的指令在时间上和空间上被相互隔离时，细微的程序缺陷，尤其是编译器引入的缺陷，可能会消失。

我们采用的解决方法是，为调试器提供足够的基本块信息，使其至少可以回答下面的问题：能不能提供变量a的正确值？如果能，怎样提供？用来表示该信息的数据结构是基本块的dag，用来注释在源程序和优化后的程序中的什么时间，哪个变量保存着与dag中某个节点相对应的值。节点附带的注释 a: *i*-*j* 表示从语句 *i* 开始到出现赋值语句前的语句 *j* 中，该节点所表示的值一直被保存在变量a中。如果 $j = \infty$，则一直到基本块的末尾a都将保存该节点的值。

例10.53 图10-68a是一个源代码的基本块，图10-68b是该代码的一种优化版本。图10-69是两者的dag，图中标明了源程序和优化代码中每个变量的取值范围，′号用来指示该语句范围是在优化代码中。例如，标以+的节点是源代码中从语句(2)开始到赋值语句(3)赋值之前c的值。它还是源代码中从语句(3)开始到基本块末尾的d的值。另外，这个节点还是优化代码中从语句(2′)开始到块末尾的d的值。 □

```
(1)  c := a+b        (1′)  d := a+b
(2)  d := c          (2′)  t := b*e
(3)  c := c-e        (3′)  a := d-e
(4)  a := d-e        (4′)  b := d/t
(5)  b := b*e        (5′)  c := a
(6)  b := d/b
        a)                    b)
```

图10-68 源代码和优化代码

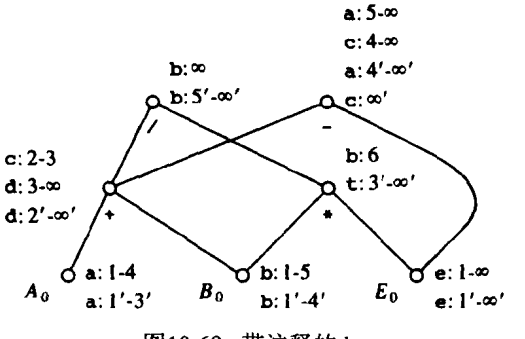


图10-69 带注释的dag

现在，我们可以回答上面提出的第1个问题了。假定执行优化代码的语句 *j*′ 时发生了溢出错误，因为与语句 *j*′ 相同的 dag 节点的任何源语句都将计算相同的值，所以有必要告诉用户，错误发生在计算该节点的第一条源语句处。因此，在例10.53中，如果错误发生在语句(1′)，(2′)，(3′)或(4′)，则可以报告用户错误相应地发生在语句(1)，(5)，(3)或(6)处。因为语句(5)没有计算任何值，所以不会发生错误。在例10.54中，我们会详细介绍对应语句的计算方法。

我们还可以回答第2个问题。假设在优化代码的语句 *j*′ 发生了一个错误，而用户被告知控制正处在源代码的语句 *i*。如果用户想查看变量x的值，我们必须找到一个变量 y（在多数情况下，y就是x），使得源程序语句 *i* 中变量x的值和优化代码语句 *j*′ 中变量y的值是同一个dag节点。检查dag找出哪个节点代表语句 *i* 中变量 x 的值，我们可以从该节点中读出目标程序曾经具有该值的所有变量，以查看是否有哪个变量具有语句 *j*′ 中的这个值。

假如是这样，问题已经得到解决；如果不是，我们可能还要通过语句 *j*′ 中的其他变量来计算语句 *i* 中的x值。令 *n* 是代表 *i* 时刻变量 x 的节点，则我们可以考虑节点 *n* 的子节点 *m* 和 *p*，来确认这两个节点是否代表 *j* 时刻某一变量的值。如果 *m* 节点有一变量而 *p* 节点没有，那么我们可以递归地考虑 *p* 的子节点。最后，要么可以找到一种计算 *i* 时刻 x 的值的方法，要么可以

705 ~ 706 断定x的值无法计算。如果找到了计算 m 和 p 的值的方法，我们就可以计算它们，并在节点 n 运用该算子来计算i时刻x的值。[⊖]

例10.54 假设执行图10-68b中的代码时，在语句(2′)处发生了一个错误。语句(2′)正在计算图10-69中标记为*的节点，而且计算该值的第一条源代码语句是语句(5)，因此我们报告在语句5发生了一个错误。

在图10-70中，我们列表给出在源代码语句(5)的开始和优化代码语句 (2′) 的开始，与源代码和优化代码的每个变量相对应的 dag 节点。节点用标号来表示，标号是一个运算符号或者是像 A_0 这样的初始值符号。我们将说明怎样从时刻 2′ 的变量值来计算时刻5的值。例如，如果用户想知道a的值，标记为-的节点的值就会被给出。在时刻 2′ 没有变量具有该值，但幸运的是，变量 d 和 e 在时刻 2′ 具有节点-的各个子节点的值，于是可以通过计算 d-e 的值来求 a 的值。 □

变量	在下列时刻的值		获自
	时刻 2′	时刻 5	
a	A_0	-	d-e
b	B_0	B_0	b
c	未定义	-	d-e
d	+	+	d
e	E_0	E_0	e
t	未定义		

图10-70 在时刻2′和时刻5的变量值

现在，让我们来回答第3个问题：怎样处理用户对调试器的调用。在某种意义上，答案并不重要。如果用户在源程序的语句i后设置断点，我们可以在该块的开始暂停程序的执行。如果用户想在语句 i 之后知道某个变量x的值，可以从带注释的dag中找到代表x值的节点，然后从该块的变量初始值中计算出该值。

707 另一方面，如果尽可能迟地调用调试器，可以减少调试器的工作，还可以避免计算某个值导致的错误。很容易找到优化代码中最后一条语句 j'，使得在语句 j' 之后调用调试器。这样做，对用户来说，感觉是在源程序语句i之后调用调试器。为了找到这样的语句 j'，令 S 是dag中的节点集合，该集合中的节点对应于源程序中紧跟语句 i 之后的某个变量的值。用户可能要计算 S 中任一节点的值，因此，只要对 S 中每个节点 n，存在某一个 $k' > j'$，使得在优化代码的时刻 k' 某个变量同节点 n 相关联，我们就可以在优化代码中语句 j' 之后设置断点。于是知道，n 的值紧跟在语句 j' 之后是可用的，或者在语句 j' 之后的某一时刻求出来。在前一种情况下，如果在语句 j' 之后设置断点，计算 n 的值意义不大；而在后一种情况下，语句 j' 之后可用的值足以计算出 n 值。

例10.55 再次考虑图10-68中的源代码和优化代码，假设用户在源代码语句(3)之后设置了一个断点。为了找到集合 S，考察图10-69的dag，并找出在时刻4，有哪些节点附带有源程序变量，即图中标记为 A_0，B_0，E_0，+和-的节点。

现在，再次考察dag，找出最大的 j'，使得集合 S 中的每个节点在绝对大于 j' 的时刻都附带有优化代码的某个变量。标记为+，-和 E_0 的节点没有问题，因为它们的值在时刻∞′分别由变量d，a和e携带。节点 A_0 和 B_0 限制了 j' 的值，A_0 和 B_0 中最早改变其值的是节点 A_0，语句 3′ 改变了它的值。因此，$j' = 2'$ 是 j' 的最大可能值。也就是说，如果用户想在源程序语句3之后设置断点，我们可以在语句2′之后为其设置相应的断点。 □

⊖ 如果节点n的值的计算引起另一个错误，则会有细微的差别。我们必须报告用户该错误实际上早就出现了，即在计算n的值的第一条源语句处。

因为不存在真正的好解，读者应该知道例10.55的微妙之处。如果在调用调试器之前，运行优化代码并穿过语句2′，在语句2′中计算b*e的值时可能出错（例如下溢），这个错误可能导致调试器提前被调用。然而，因为和语句2′对应的计算在源程序中直到语句5才会出现，所以要告诉用户该错误是在语句5出现。在语句3处没有调用调试器，我们是怎么到达语句5的呢？用户对此可能感到有些神秘。也许最好的解决方法是，不允许 j' 如此大，以致存在一个这样的优化代码语句 k'，$k' \leqslant j'$，使得直到设置了断点的语句 i 之后，源代码才计算 k' 所计算的值。

10.13.2 全局优化的影响

当编译器进行全局优化时，对符号调试器来说，还有更难的问题需要解决，并且在某一点 708
常常无法找到变量的正确值。归纳变量删除和全局公共子表达式删除这两个重要的变换不会产生重大的问题；两种情况下都可以将问题局限于少数几个块内，并按前面讨论的方式进行处理。

10.13.3 归纳变量删除

如果删除源程序中的归纳变量i，以利于i族的某个成员t，那么i和t之间就存在某个线性函数。如果我们采用10.7节的方法，则i在某个块中被改变，优化代码也将在相应的块中改变t的值，所以i和t之间的线性关系总是成立。因此，在考虑记录某个块中给t赋值的语句（在源程序中，是给i赋值的语句）之后，就可利用t的线性变换为用户提供i的“当前”值。

如果i在循环之前没有被定义，就要非常小心，因为t在进入循环之前肯定要被赋值，并且可能就在程序中，用户认为i是未定义的地方，为i提供了一个值。幸运的是，通常情况下，源程序中的归纳变量在进入循环之前都会被初始化，只有编译器产生的变量（用户不会想要它们的值）在循环的入口才会不被定义。如果某个归纳变量i不属于这种情况，那么我们将会遇到一个类似于代码外提的问题，这个问题将在下面进行讨论。

10.13.4 全局公共子表达式删除

如果我们对表达式a+b执行全局公共子表达式删除，同样影响了有限数目的基本块。如果t是用来保存 a+b的值的变量，那么在某些计算 a+b的块中我们可将 c := a + b 替换成：

```
t := a+b
c := t
```

采用已经讨论过的处理基本块的方法即可处理这种变化。

在其他块中，像 d := a+b 这样的引用可以用 d := t 来替换。为了用前面的方法来处理这种情况，只需要在该块的dag中注意t的值一直代表 a+b 节点的值（代表a+b的节点只会在源代码的dag中出现，而不会在优化代码的dag中出现）。 709

10.13.5 代码外提

其他变换处理起来比较难。例如，假设把循环不变的语句s：a := b+c移到循环之外。如果在循环内调用调试器，我们将不知道在源程序中语句s是否被执行过，因此也就不知道a的当前值是不是用户在源程序中看到的值。

一种可能的办法是在优化代码的循环中插入一个新变量，该变量指示a是否在循环中被赋值（这个变量只能放在语句s原来的位置上）。然而，这种策略也并不总是适用，所以要想绝对可靠，只能使用真正的代码，而不使用为了调试构造的代码。

然而有一种特殊的情况我们可以做得很好。假设源程序包含语句 s 的块 B 把循环分成两个节点集合：支配 B 的节点和 B 所支配的节点。此外，假设B支配首节点的所有前驱，如图10-71所示。于是程序第一次经过支配 B 的块时，我们可以假设a在循环中没有被赋值，而程序第一次经过 B 所支配的块时，a在语句 s 处被赋值。第二次和以后各次经过循环时，a都会

在s处被赋值。

如果对调试器的调用是由运行时错误引起的，则第一次循环时极有可能发现错误。如果是这样，我们只需知道错误位于图10-71中B的上面还是下面。然后我们会知道a的值是否在 s 处被定义，或者a的值是否是源程序中a在循环入口处的值。对前一种情况，我们只需打印优化代码产生的a值，对后一种情况，我们什么都不能做，除非：

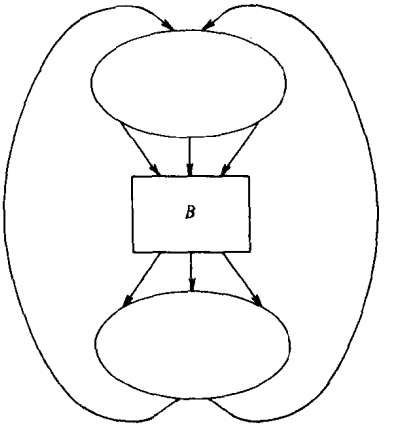


图10-71 将循环分成两部分的块

1. 对源程序和优化程序，调试器都有可用的到达定义信息。

2. 在源程序中存在一个到达首节点的 a 的惟一定义。

3. 该定义还是到达调试器被调用处的某个变量 x 的惟一定义。

如果这些条件都满足，那么我们可以打印出x的值，它也就是a的值。

如果通过用户设置的断点调用调试器，则读者应该知道这一行的推理不成立，我们没有任何理由怀疑我们正第一次经过该循环。然而，如果利用用户设置的断点，在优化程序中加入一些代码，帮助调试器分辨程序是否第一次经过循环的做法也许是合理的。然而这种方法需要修改优化代码，所以可能不适用。

练习

10.1 考虑图10-72中的矩阵乘法程序。

```
begin
    for i := 1 to n do
        for j := 1 to n do
            c[i,j] := 0;
    for i := 1 to n do
        for j := 1 to n do
            for k := 1 to n do
                c[i,j] := c[i,j] + a[i,k] * b[k,j]
end
```

图10-72 矩阵乘法程序

710
~
711

a) 假设 a，b，c 采用静态存储分配，在字节寻址内存中每个字由4个字节构成，请写出图10-72中程序的三地址语句。

b) 从三地址语句生成目标机器代码。

c) 从三地址语句构造流图。

d) 从每个基本块中删除公共子表达式。

e) 找出流图中的循环。

f) 将循环不变计算移出循环。

g) 找出每个循环的归纳变量并尽量删除它们。

h) 从(g)的流图生成目标机器代码，并与(b)中产生的代码做比较。

10.2 计算练习10.1(c)的流图和练习10.1(g)的流图的到达定义和ud链。

10.3 图10-73中的程序用筛选法在一个规模适当的数组中统计从2到n的素数。

```
begin
    read n;
    for i := 2 to n do
        a[i] := true;    /* 初始化 */
    count := 0;
    for i := 2 to n ** .5 do
        if a[i] then    /* i 是素数 */
            begin
                count := count + 1;
                for j := 2 * i to n by i do
                    a[j] := false
                    /* j能被i除 */
            end;
    print count
end
```

图10-73 计算素数的程序

a) 假设a采用静态存储分配，把图10-73中的程序翻译成三地址语句。

b) 从三地址语句生成目标机器代码。

c) 从三地址语句构造流图。

d) 给出(c)中流图的支配树。

e) 对(c)中流图，指出其回边和它们的自然循环。

f) 用算法10.7将循环不变计算移出循环。

g) 尽可能删除归纳变量。 712

h) 尽量将复制语句传播出去。

i) 该循环能收缩吗？如果能则执行它。

j) 如果假定n一直是偶数，每次展开内循环一次。现在可以执行何种新优化？

10.4 假设a采用动态存储分配，用指向a的第一个字的指针 ptr，重做练习10.3。

10.5 针对图10-74中的流图，计算如下信息：

a) ud链和du链。

b) 每个块末尾的活跃变量。

c) 可用表达式。

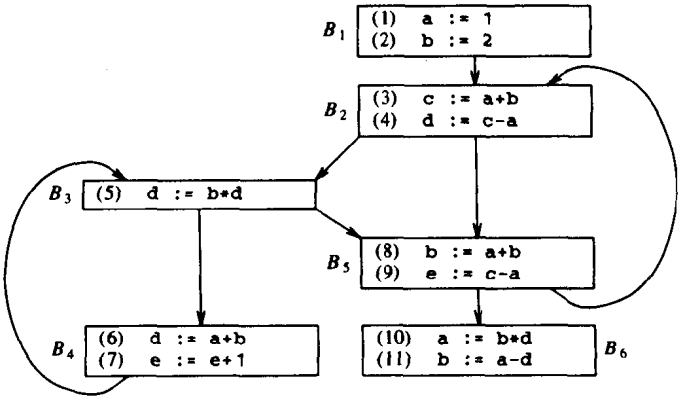


图10-74 流图

10.6 图10-74中能否进行常量合并？如果能则执行它们。

10.7 图10-74中有公共子表达式吗？如果有则删除它们。

10.8 如果从 p 点出发的任何路径上，在计算任何运算对象之前，都要先计算表达式 e，则称表达式 e 在点 p 非常忙。按照10.6节的模式给出一个数据流算法找出所有的非常忙表达式。你使用什么样的聚合操作符？信息传播是前向的还是后向的？将你的
713 算法应用到图10-74中的流图上。

* 10.9 如果表达式 e 在点 p 非常忙，通过在点 p 计算 e，并保存其值以备后用，这样可以提升 e。（注意：这种优化通常只能节省时间，而不能节省空间。）给出一个提升非常忙表达式的算法。

10.10 图10-74中有哪些表达式可以提升吗？如果有则提升它们。

10.11 如果可能，将练习10.6、练习10.7和练习10.10的改进中引入的复制步骤传播出去。

10.12 块序列 B_1，…，B_k 中，对 $1 \leqslant i < k$，B_i 是 B_{i+1} 的惟一前驱，而 B_1 没有惟一的前驱，则称该块序列是一个扩展基本块。从下列图中，找出在下图中每个节点结束的扩展基本块：

a) 图10-39。

b) 练习10.1(c)中构造的流图。

c) 图10-74。

* 10.13 给出一个基于 n 节点流图且运行时间为 $O(n)$的算法，找出在每个节点结束的扩展基本块。

10.14 把每个扩展基本块当作基本块处理，不用做任何数据流分析就可以完成一些块间的代码优化。给出在扩展基本块内执行如下优化的算法，并指出每种情况中一个扩展基本块中的变化对其他扩展基本块的影响。

a) 公共子表达式删除。

b) 常量合并。

c) 复制传播。

10.15 对练习10.1(c)中的流图，

a) 找出该图的 T_1 和 T_2 化简序列。

b) 找出区间图序列。

c) 什么是限制流图？该流图是可约的吗？

10.16 对图10-74中的流图重做练习10.15。

** 10.17 证明下列条件是等价的（它们是“可约流图”的可选定义）。

a) T_1-T_2化简的极限是单个节点。

b) 区间分析的极限是单个节点。

c) 流图的边可分为两类：一类边形成无环图，另一类边是由头支配尾的边组成的回边。

d) 该流图没有图10-75所示形式的子图。其中，n_0 是初始节点，而且 n_0，a，b 和 c都互不相同（除了有可能 $a = n_0$）。箭头表示不相交节点路径（端点除外）。

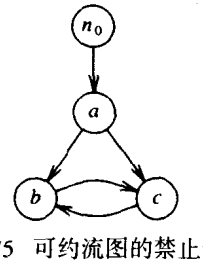


图10-75 可约流图的禁止子图

10.18 对10.8节讨论的带有指针的语言，给出计算(a)可用表达式和(b)活跃变量的算法。确

保对(b)中的 *gen*，*kill*，*use* 和 *def* 做保守假设。

10.19 用10.8节的模型给出计算过程间到达定义的算法，确保对真实解做保守的近似假设。

10.20 假设参数传递采用传值方式而不是传地址方式，那么两个名字可以互为别名吗？采用复制-恢复方式传值又如何？

10.21 练习10.1(c)中的流图的深度是多少？

** 10.22 证明可约流图的深度不小于产生单个节点所必须执行的区间分析次数。

* 10.23 将10.8节的基于结构的数据流分析算法推广到10.11节的通用数据流框架上。为保证算法正常运转，应对F和$\wedge$做何种假设？

* 10.24 设想被传播的值是表达式所有可能的划分（只有当两个表达式具有相同的值时，它们才属于同一个等价类），我们可以得到一个有趣的且功能强大的数据流分析框架。为了避免列出所有的无限个可能的表达式，我们可以通过只列出与其他表达式等价的最小的表达式来表示这些值。例如，如果我们执行下面的语句：

```
A := B
C := A + D
```

714 ~ 715

那么我们将得到如下最小的等价关系：A≡B，C≡A+D。由此可得其他等价关系，如 C≡B+D 和 A+E≡B+E，但没有必要一一列出它们。

a) 对这种框架而言，合适的与（或聚合）操作符是什么？

b) 给出表示值的数据结构和实现与操作符的算法。

c) 同语句相关联的合适的函数是什么？同 A := B+C 这样的赋值语句相关联的函数对一个划分的作用是什么？

d) 该框架具有分配性吗？是单调的吗？

10.25 如何使用由练习10.24的框架搜集的数据执行下列优化：

a) 公共子表达式删除。

b) 复制传播。

c) 常量合并。

* 10.26 给出下列栅格图上的关系≤的形式化证明：

a) 如果 $a \leq b$ 且 $a \leq c$，则 $a \leq (b \wedge c)$。

b) 如果 $a \leq (b \wedge c)$，则 $a \leq b$。

c) 如果 $a \leq b$ 且 $b \leq c$，则 $a \leq c$。

d) 如果 $a \leq b$ 且 $b \leq a$，则 $a = b$。

** 10.27 证明下列条件是深度优先顺序的迭代数据流算法在深度加2遍之内收敛的充分必要条件：对所有的函数 f 和 g 及值 a，

$$f(g(a)) \geq f(a) \wedge g(a) \wedge a$$

10.28 证明到达定义和可用表达式框架满足练习10.27中的条件。注意：事实上，这些框架在深度加1遍之内收敛。

** 10.29 单调性意味着练习10.27中的条件成立吗？分配性呢？反之成立吗？

10.30 在图10-76中我们看到两个基本块，第一个是原始代码，第二个是优化代码。

a) 为图10-76a和图10-76b中的块构造dag。假设在出口只有J是活跃的，验证这两个

块是等价的。

b) 在每个节点上，将变量值已知的变量标注到带时间的dag上。

c) 如果一个错误发生在语句(1′)到(4′)的各个语句上，请分别指出错误发生在10-76a中的哪条语句上。

716

d) 对(c)中的每个错误，指出图10-76a中哪个变量是可求值的？我们应该怎么求值？

e) 假设允许使用形如“如果 $a + b = c$ 则 $a = c - b$”的有效代数法则，(d)的答案会有变化吗？

(1)	E := A+B	(1′)	E := A+B
(2)	F := E-C	(2′)	E := E-C
(3)	G := F*D	(3′)	F := E*D
(4)	H := A+B	(4′)	J := E+F
(5)	I := I-C		
(6)	J := I+G		
	a)		b)

图10-76 原始代码和优化代码

a) 原始代码 b) 优化代码

10.31 推广例10.14，考虑带有break语句的任意集合，并将其推广到包含 continue 语句。contuinue语句不中断内循环，而是直接进入循环的下一次迭代。提示：使用10.10节提出的用于可约流图的方法。

10.32 证明：在算法10.3中，定义的 *in* 集和 *out* 集不会减小。同样地，证明：在算法10.4中，表达式的in集和*out*集不会增大。

10.33 将归纳变量删除算法10.9推广到乘法常数允许为负数的情况。

10.34 将10.8节中确定指针指向的算法推广到允许指针指向其他指针的情况。

* 10.35 当估计下面的每个集合时，指出太大估计是保守的，还是太小估计是保守的？按照信息的预定用途解释你的答案。

a) 可用表达式。

b) 被过程改变的变量。

c) 没有被过程改变的变量。

d) 属于给定族的归纳变量。

e) 到达给定点的复制语句。

* 10.36 对算法10.12进行求精，计算在给定点的给定变量的别名。

717

* 10.37 针对下面的参数传递方式，试修改算法10.12：

a) 传值。

b) 复制-恢复传递。

* 10.38 证明算法10.13收敛到一个确实被改变的变量的超集（不必是真超集）。

* 10.39 推广算法10.13，在允许过程定值变量的情况下，确定被改变的变量。

* 10.40 证明在每个区间图中，每个节点代表原流图的一个区域。

10.41 证明算法10.16正确地计算出了每个节点的支配节点的集合。

* 10.42 修改算法10.17（基于结构的到达定义），只计算指定的小区域的到达定义，而不要求整个流图一次都装在内存中。确保你的结果是保守的。修改你的算法以适合于可用表达式。哪一个更有可能提供有用的信息？

* 10.43 在10.10节，基于 T_1 和 T_2 化简的合并，我们提出了对算法10.17的一种加速。试证明这种修正的正确性。

10.44 将10.11节的迭代方法推广到后向流 (backward-flowing) 问题。

** 10.45 通过证明对每条长度为 i 的路径 P，经过 i 次迭代以后，$in[B_i] \leq f_P[\top]$，证明当算法10.18收敛时，结果解 $\leq mop$ 解。

10.46 图10-77是10.12节介绍的假设语言程序的流图，使用算法10.19找出每个变量的类型的最佳估计。

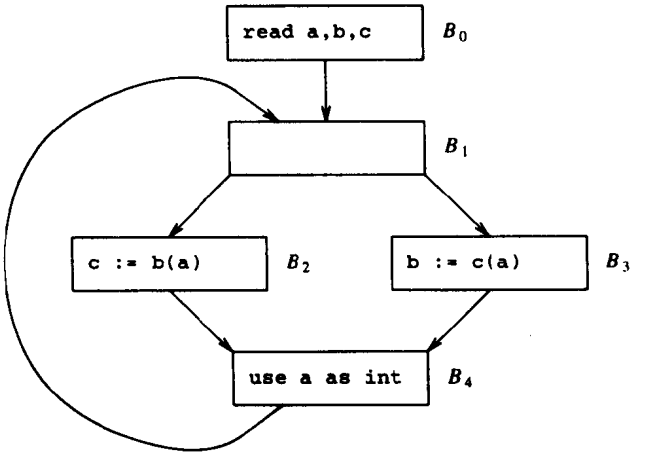


图10-77 类型推断的程序例子

参考文献注释

从Cocke and Schwartz[1970]、Abel and Bell[1972]、Schaefer[1973]、Hecht[1977]以及Muchnick and Jones[1981]中可以获得更多关于代码优化的信息。Allen[1975]给出了一份关于程序优化的参考文献列表。

许多优化编译器在文献中都有记载。Ershov [1966]讨论了一个早期的使用复杂优化技术构造的编译器。Lowry and Medlock[1969]以及Scarborough and Kolsky[1980]中详细描述了Fortran优化编译器的构造方法。Busam and Englund[1969]以及Metcalf[1982]则给出了更多的Fortran优化技术。Wulf et al. [1975]讨论了一个颇有影响的Bliss优化编译器的设计。

Allen et al. [1980]描述了一个程序优化的试验性系统。Cocke and Markstein[1980] 阐述了一种类PL/I语言各种优化的有效性。Anklam，Cutler，Heinen, and MacLaren[1982]描述了用于PL/I和C语言编译器的优化变换的实现。Auslander and Hopkins[1982]发表了一个PL/I变体的编译器，它使用一种简单的算法来产生低层中间代码，这种代码进而又被全局优化变换所改进。Freudenberger，Schwartz, and Sharir[1983]传授了设计SETL优化器的经验。Chow[1983] 发表了一个可移植的、与机器无关的Pascal优化器上的实验。Powell[1984]则描述了一个可移植的、与机器无关的Modula-2优化编译器。

虽然在Allen[1970]和Cocke[1970]前，各种不同的数据流分析方法已经在使用，因为他们两人联合发表了关于该技术的Allen and Cocke[1976]，我们仍然将Allen[1970]和Cocke[1970]视为数据流分析技术系统研究的开端。

10.5节介绍的语法制导数据流分析已经被用在Bliss（Wulf et al. [1975]，Geschke[1972]）、SIMPL（Zelkowitz and Bail[1974]）和Modula-2（Powell[1984]）中。Hecht and Schaffer[1975]、Hecht[1977]以及Rosen[1977]对该类算法进行了更多的讨论。

10.6节讨论的迭代数据流分析方法可以追溯到Vyssotsky(参见Vyssotsky and Wegner[1963])，他在1962年的一个Fortran编译器上使用了该方法。采用深度优先顺序改善编译器效率的方法来自Hecht and Ullman[1975]。

718 ~ 719

Cocke[1970]开创了区间数据流分析的先河。Kennedy[1971]创造性地将该方法用于解决像活跃变量这样的后向数据流问题。如果被优化的语言很少产生不可约流图，那么就有理由相信按照Kennedy[1976]的方法，基于区间的方法在某种程度上比迭代法更有效。基于T_1和T_2的方法的变体来自Ullman[1973]。Graham and Wegman[1976]给出了一个某种程度上更快的版本，该版本利用了这样的事实：大部分代码区域只有一个出口。

Allen[1970]中给出了可约流图的最初定义，即可约流图在迭代区间分析下将变成单个节点。等价的描述可以在Hecht and Ullman[1972，1974]、Kasyanov[1973]以及Tarjan[1974b]中找到。不可约流图的节点分裂来自Cocke and Miller[1969]。

Kosaraju[1974]、Kasami，Peterson, and Tokura[1973]以及Cherniavsky，Henderson, and Keohane[1976]中描述了用可约流图作为结构化控制流模型的思想。Baker[1977]描述了它们在程序结构化算法中的用途。

用于迭代数据流分析的栅格理论方法始于Kildall[1973]。Tennenbaum[1974]和Wegbreit[1975]给出相似的方程式。Kam and Ullman[1976]提出了使用深度优先顺序的Kildall算法的高效版本。

在Kildall假定分配性条件（例10.42描述了他的常量计算框架，这类框架实际上并不满足该条件）成立的同时，可以从许多给出数据流算法的论文中发觉单调性的充分性，这些论文包括Tennenbaum[1974]、Schwartz[1975a，b]、Graham and Wegman[1976]、Jones and Muchnick[1976]、Kam and Ullman[1977]以及Cousot and Cousot[1977]等。

由于不同的算法需要对数据做不同的假设，Kam and Ullman[1977]、Rosen[1980]和Tarjan[1981]提出了关于特定算法需要何种特性的理论。

从Kildall的论文可以得到另外的指导，就是对处理他所介绍的特定数据流（例10.42）算法进行改进。一个主要的思想是不必把栅格元素看作是原子的，但是我们可以利用这样的事实：它们确实是从变量到值的映射。参见Reif and Lewis[1977]以及Wegman and Zadeck[1985]。Kou[1977]还将这些思想用在更传统的问题上。

Kennedy[1981]是数据流分析技术的综述，而Cousot[1981]则概述了栅格理论的思想。

Gear[1965]介绍了代码外提和归纳变量删除的约束形式等基本循环优化。Allen[1969]是一
720 篇关于循环优化的重要论文；Allen and Cocke[1972]以及Waite[1976b]在该领域进行了更广泛的研究。Morel and Renvoise[1979]描述了同时删除循环中的冗余和不变计算的算法。

10.7节讨论的归纳变量删除技术是在Lowry and Medlock[1969]的基础上提出的。想获知更好的算法，请参阅Allen，Cocke，and Kennedy[1981]。

关于循环问题的一些算法，诸如判定是否存在一条从a到b但不经过c的路径，这里并没有详细论述。这些问题可以采用Wegman[1983]的有效算法来解决。

虽然Lowry和Medlock将支配节点的一般思想归功于Prosser[1959]，但是我们仍然认为用于循环发现和执行代码外提的支配节点是由Lowry and Medlock[1969]开创的。查找支配节点的算法10.16是由Purdom and Moore[1972]以及Aho and Ullman[1973a]各自独立发现的。采用深度优先顺序加速算法的方法来自Hecht and Ullman[1975]，而完成该工作的最有效的渐近方法则来自于Tarjan[1974a]。Lengauer and Tarjan[1979]描述了一种适合实际应用的寻找支配节点的高效算法。

别名和过程间分析的研究始于Spillman[1971]和Allen[1974]。现在已经有一些比10.8节所介绍的方法更有效的方法。一般地，这些方法处理程序的各个点的别名关系，以防止我们的简单

算法“发现”的不可能的别名对。有关著作包括Barth[1978]、Banning[1979]以及Weihl[1980]。还可以参阅Ryder[1979]关于调用图的构造。

Hennessy[1981]讨论了一种与过程间分析类似的问题，即异常对程序数据流分析的影响。

Tennenbaum[1974]是一篇通过数据流分析确定类型的重要论文，10.12节的讨论就是以该论文为基础的。Kaplan and Ullman[1980]给出了一种更好的类型检测算法。

10.13节关于优化代码的符号调试的讨论来自Hennessy[1982]。

许多论文试图评价各种优化所带来的改进。优化的价值似乎高度依赖于被编译的语言。读者可能希望参考Knuth[1971b]中对Fortran优化的经典研究，或参考以下文献：Gajewska[1975]、Palm[1975]、Cocke and Kennedy[1976]、Cocke and Markstein[1980]、Chow[1983]、Chow and Hennessy[1984]以及Powell[1984]。

我们在此没有包括的另一个优化问题是集合理论语言SETL这样的“超高级”语言的优化，要讨论这个话题，我们需要改变底层算法和数据结构。该领域的主要优化是推广的归纳变量删除，可以在Earley[1975b]、Fong and Ullman[1976]、Paige and Schwartz[1977]以及Fong[1979]中找到相关论述。

另一个超高级语言优化的主要步骤是数据结构的选择，可以从Schwartz[1975a，b]、Low and Rovner[1976]以及Schonberg，Schwartz, and Sharir[1981]中找到相关话题的论述。 721

我们也没有谈及增量式代码优化。如果采用增量式代码优化，对程序进行小的修改不需要对程序进行完全的重新优化。Ryder[1983]讨论了增量式数据流分析，而Pollock and Soffa[1985]则试图对基本块进行增量式代码优化。

最后应该提及的是许多已经应用到数据流分析中的一些技术。Backhouse[1984]描述基于与语法分析器有关的转换图，使用其中一种技术进行错误恢复。Harrison[1977]以及Suzuki and Ishihata[1977]讨论了它在编译时数组边界检查中的应用。

除了代码优化以外，编译时程序错误的静态检查是数据流分析最重要的用途。Fosdick and Osterweil[1976]是一篇很重要的论文，而Osterweil[1981]、Adrion，Bronstad, and Cherniavsky [1982]以及Freudenberger[1984]则给出了一些更新的研究成果。 722

第11章 编写一个编译器

了解了编译器设计的原理、技术和工具以后，假设现在我们想要编写一个编译器。预先制订一些计划可以使实现工作更加快捷顺利地进行。本章将简要介绍编译器构建中出现的一些实现问题，大部分内容都围绕着利用 UNIX 操作系统及其工具来编写编译器这一主题。

11.1 编译器设计

为了编译新的源语言，或为了产生新的目标语言，或二者均需要时，可能需要编写一个新的编译器。利用本书中介绍的框架，我们最终会设计出一个由一组模块构成的编译器。有几个不同的因素会影响这些模块的设计和实现。

11.1.1 源语言问题

语言的“规模”会影响这些模块的规模和数量。尽管语言的规模还没有一个明确的定义，但与Ratfor（一个Fortran“有理数”预处理器，Kernighan[1975]）或者 EQN（一种数学排版语言）这样的小语言的编译器相比，显然 Ada 或者 PL/I 的编译器就比较大而且较难实现。

另一个重要的因素是构建编译器的过程中源语言变化的程度。尽管源语言规范可能看起来是不可变的，但在构造编译器的整个生命周期中几乎没有保持不变的语言。甚至成熟的语言也是如此，尽管变化很慢。例如，现在的 Fortran 版本与1957年的版本相比已经有了显著的变化。Fortran 77中的循环语句、霍尔瑞斯（Hollerith）语句及条件语句与最初版本中的这些语句已经有了很大的不同。Rosler[1984]记录了 C 的演变过程。

另一方面，新的试验性语言在其实现过程中可能会发生突变。将语言的编译器工作原型演化成另一个编译器，使其目标语言能满足某些特定用户群的需要，这是一种开发新语言的途径。许多像 AWK 和 EQN 这样的最初依赖于 UNIX 系统的“小”语言都是这样开发出来的。

因此，一个编译器编写者在编译器构造的生命期内可能需要参与一定量源语言定义的修订工作。模块化的设计以及工具的使用有助于应付这些变化。例如，直接编写词法分析器和语法分析器的代码，与用生成器来实现词法分析器和语法分析器比较起来，后者能使编写者更游刃有余地应付语言定义中的语法变化。

11.1.2 目标语言问题

目标语言的性质、约束以及运行环境对于编译器的设计以及代码生成策略都有很大的影响，因此需要仔细地加以考虑。如果目标语言是新语言，编译器的编写者最好能确定它的正确性并对其时序有充分的理解。新机器或新汇编器可能有一些编译器可以发现的 bug，但目标语言的 bug 可能会加重编译器本身的调试任务。

一个成功的源语言可能会在多个目标机器上实现。如果某语言不断地被沿用下来，该语言的编译器就需要为几代目标机器生成新的目标代码。机器硬件的进一步发展似乎是必然的，因此可重置目标的编译器的研究是值得探索的。于是中间语言的设计变得重要起来，因为这可以将与硬件相关的细节限制在少数几个模块中。

11.1.3 性能标准

有几个考虑编译器性能的指标：编译器速度、代码质量、错误诊断、可移植性和可维护性。这些标准的轻重不能明显地区分开来，而且编译器规范对其中的许多参数也未做说明。例如，

编译的速度是否比目标代码的速度重要？有用的错误信息和错误恢复到底有多重要？

要提高编译器的速度，我们可以尽可能地减少模块的数量和编译的遍数，或许可以一遍直接生成机器代码。但是采用这种做法，我们就不可能得到能生成高质量代码的编译器，更不用说得到易维护的编译器了。

对可移植性来说要考虑两方面：可重置目标能力（retargetability）和可变换宿主机能力（rehostability）。具有可重置目标能力的编译器通过简单的修改就可以为新的目标语言生成代码。具有可变换宿主机能力的编译器是很容易移植到另一台机器上的编译器。因为用于特定机器的编译器可以放心地对目标机器做一些假设，而可移植的编译器则不能，因此可移植的编译器可能不如专门为特定机器设计的编译器高效。

724

11.2 编译器开发方法

实现编译器有几个常用的一般方法。最简单的办法是将现成的编译器重新确定目标机器或者重新确定宿主机器。如果找不到合适的现成编译器，可以使用已知的类似语言编译器的结构，使用组件生成工具或手工方式实现相应的组件。需要全新的编译器结构的情况是很少见的。

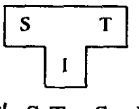
不管采用什么方法，编译器的编写是软件工程中的一种训练。软件开发中的一些经验教训（可参见Brook[1975]）可以用来提高最终产品的可靠性和可维护性。一个能很好地适应变化的设计方案将使编译器能与语言同步发展。在这点上使用编译器构建工具将有很大的帮助。

自举

编译器是一个很复杂的程序，所以我们往往愿意用一种比汇编语言更友好的语言来编写它。在UNIX的编程环境下，编译器通常用C语言编写，甚至连C的编译器也用C来编写。使用语言提供的功能来编译该语言自身，这就是自举（bootstrapping）的实质。下面我们将讨论如何用自举来构建编译器，以及如何通过修改后端将它们从一个机器移植到另一个机器上。在20世纪50年代中期人们就已经知道了自举的基本思想（Strong et al. [1958]）。

自举可能会引出这样的问题，“第一个编译器是怎样被编译的？”。这个问题听起来就像问“到底是先有鸡呢还是先有蛋？”。但这个问题本身却很容易回答。为了获得答案我们先看看Lisp是怎样成为一种编程语言的。McCarthy[1981]中谈到：在1958年后期Lisp是一种编程符号，用于编写函数，然后用手工将它们翻译成汇编语言并运行。一个Lisp解释器的实现却意外地诞生了。McCarthy原想说明Lisp是描述函数的符号，他称Lisp“比图灵机或者递归函数论中使用的一般的递归定义还要规整”，于是他用Lisp写了一个函数 *eval*[*e*,*a*]，该函数将一个Lisp表达式 *e* 作为一个参数。S. R. Russell注意到函数 *eval* 可以作为Lisp的解释器，于是对它进行手工编码，这样就使Lisp成为了一种带有解释器的编程语言。像1.1节中提到的那样，解释器实际上执行源程序中的操作，而不是生成目标代码。

为了自举，一个编译器可以用三种语言来刻画：编译器要编译的源语言S，编译器要产生的目标语言T，编写编译器所用的实现语言I。我们用下面的T型图来表示这三种语言。T型图
725 这个名称源于图的形状（Bratman[1961]）。



在文章中，我们将上面的T型图简写为 $S_I T$。S、I和T可能是三种完全不同的语言。例如，在某种机器上运行的编译器可能为另外一种机器生成目标代码，这样的编译器通常被称为**交叉编译器**。

假定现在要利用已实现的语言S为新语言L编写一个交叉编译器，并生成机器N的目标

代码，即创建 L_SN。如果语言 S 的编译器在机器 M 上运行并生成 M 的代码，我们用 S_MM 来刻画。如果 L_SN 通过 S_MM 来运行，我们可以获得编译器 L_MN，即一个运行于机器 M 上的将语言 L 编译成语言 N 的编译器。将这些编译器的T型图放在一起得到下面的图11-1，该图很好地说明了上述过程。

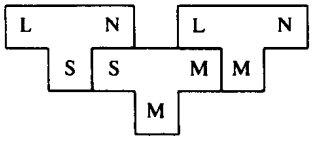
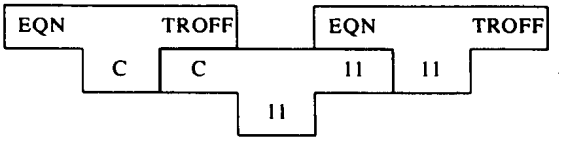


图11-1 编译一个编译器

像图11-1那样将 T 型图组合在一起的时候，编译器 L_SN 的实现语言 S 必须和已有编译器 S_MM 的源语言相同，而 S_MM 的目标语言 M 必须和翻译后的 L_MN 的实现语言相同。图11-1可以看成如下的一个等式：

$$L_SN + S_MM = L_MN$$

例11.1 EQN 编译器（参见12.1节）最初的版本以 C 为实现语言并为文本格式化工具 TROFF生成代码。如下图所示，在运行于 PDP-11 上的 C 编译器 $C_{11}11$ 的基础上，我们使用 EQN_CTROFF 就可以得到运行于 PDP-11 上的 EQN 的交叉编译器。

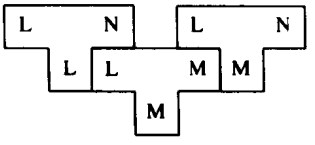


□ 726

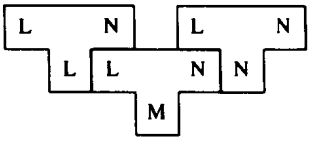
自举的一种形式是为不断扩充的语言子集构建编译器。假设新语言 L 要在机器 M 上实现。我们可以先编写一个小编译器将 L 的子集 S 编译成 M 的目标代码，即建立编译器 S_MM。然后我们用子集 S 为 L 编写编译器 L_SM。在 S_MM 的基础上运行 L_SM，我们就得到语言 L 的一个实现，即 L_MM。Neliac 是最早通过自身来实现的语言之一（Huskey, Halstead, and McArthur[1960]）。

Wirth[1971] 谈到 Pascal 最早是用 Pascal 自身编写的编译器来实现的。在此基础上，再“手工”将该编译器翻译成可用的未经优化的低级语言。这个编译器是为 Pascal 语言的一个子集（多于60%）编写的；经过几次自举后就得到整个 Pascal 的编译器。Lecarme and Peyrolle-Thomas[1978] 中总结了自举 Pascal 编译器所用到的方法。

为了完全认清自举的优点，编译器必须以要被编译的语言来编写。假设我们用语言 L 编写了一个 L 的编译器 L_LN，并生成机器 N 的目标代码。开发工作在机器 M 上进行，M 上运行着编译器 L_MM，该编译器为 L 生成机器 M 的目标代码。首先用 L_MM 来编译 L_LN，我们即可获得一个在 M 上运行却能为 N 生成代码的交叉编译器：



用上面生成的交叉编译器再一次编译编译器 L_LN：



我们便获得了一个运行于 N 上并为 N 生成目标代码的编译器 L_NN。这种两步编译的过程有许多应用，因此我们用图11-2来表述它。

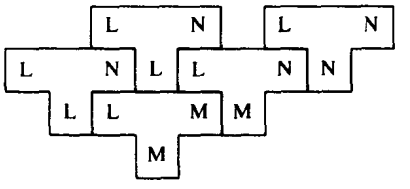
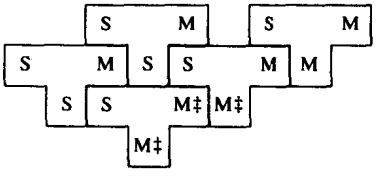


图11-2 自举一个编译器

例11.2 本例是受 Fortran 语言的 H 编译器（见12.4节）开发的启发而提出来的。“编译器本身是用 Fortran 语言编写的，并进行了三次自举。第一次是将其运行的机器从 IBM 7094 转换到 System/360——这是一个费劲的过程。第二次是它自身的优化，这一过程将编译器的大小从约 550K 减少到约 400K 字节”（Lowry and Medlock[1969]）。

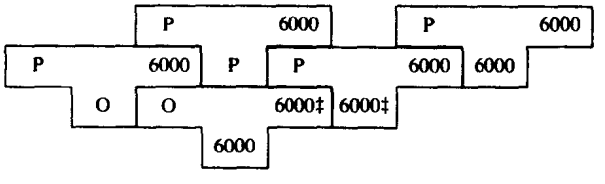
利用自举技术，优化编译器能对自身进行优化。假定所有的开发都是在机器 M 上进行的，而且我们有一个用 S 编写的语言 S 的优质优化编译器 S_SM，现在我们要构建一个用 M 编写的 S 的优质优化编译器 S_MM。

我们可以在 M 上为 S 创建一个暂且应急的编译器 $S_{M‡}M‡$，它不仅生成劣质的代码，而且需要花很长的时间来完成该过程。（M‡ 表示 M 的一个粗劣的实现。$S_{M‡}M‡$ 表示编译器的一个粗劣的实现，它生成糟糕的代码。）然而经过两步，利用这个平凡的编译器 $S_{M‡}M‡$ 我们却可以得到优秀的 S 编译器：



首先，优化编译器 S_SM 被应急编译器翻译成 $S_{M‡}M$，这个新产生的编译器是优化编译器的一个粗劣的实现，但是它生成的代码是优质的。然后用 $S_{M‡}M$ 来编译 S_SM 就得到优质的优化编译器 S_MM。 □

例11.3 Ammann[1981]描述了怎样利用类似于例11.2的过程得到 Pascal 的一个清晰实现。对 Pascal 的修正导致了1972年编写的运行于 CDC 6000 系列机上的全新编译器的产生。下图中，O 代表“老” Pascal，P 代表修正后的 Pascal。



727
⌇
728

修正版 Pascal 的编译器是用老版本的 Pascal 编写的，然后被翻译成 $P_{6000‡}6000$。与例11.2中一样，符号‡表示粗劣的语言或编译器。老编译器不能产生高效代码。“所以，$[P_{6000‡}6000]$ 编译器的速度虽然还可以，但其存储要求非常高（Ammann[1981]）。”修正版的 Pascal 非常小，这使得无须费很大的功夫就可以手工将编译器 P_O6000 翻译成 P_P 6000，然后通过并不高效的编译器 $P_{6000‡}6000$ 就可以得到 Pascal 的一个清晰的实现。 □

11.3 编译器开发环境

事实上，编译器只是一个程序。该程序的开发环境可能影响实现编译器的速度和可靠性。

编译器的开发语言同样重要。尽管很多编译器已经用 Fortran 这样的语言实现，但是使用 C 这样的面向系统的语言是更明智的选择。

如果源语言本身是一个新的面向系统的语言，那么最好用其本身来编写编译器。使用前几节讨论的自举技术将有助于编译器的调试。

编程环境中的软件构建工具为创建高性能的编译器提供了极大的方便。在编写编译器时，习惯上将整个程序划分为一些模块，对每个模块可以用不同的方式进行处理。对于编译器的编写者来说，一个管理各模块的程序是不可缺少的辅助工具。UNIX 系统中有一条称为 *make*（Feldman[1979a]）的命令，它管理和维护计算机程序的各个组成模块。*make* 跟踪程序模块间的关系，并在程序被修改后发出指令，重新编译被修改的模块，通过这种方式可以保持程序的一致性。

例11.4 命令 *make* 从文件 `makefile` 中读取要完成的任务的说明。在2.9节中，我们构建了一个翻译器，该翻译器是用 C 编译器将7个均依赖于全局头文件 `global.h` 的源程序编译后得到的。为了说明 *make* 是怎样将编译任务放在一起完成的，我们可以看一下下面的例子。假定我们将最终的编译器称为 `trans`，文件 `makefile` 中的说明可能如下：

```
OBJS = lexer.o parser.o emitter.o symbol.o\
       init.o error.o main.o

trans: $(OBJS)
       cc $(OBJS) -o trans

lexer.o parser.o emitter.o symbol.o\
       init.o error.o main.o:  global.h
```

729

第一行中的等号使得左部的 `OBJS` 代表右部的7个目标文件。（长行可以分成短行，只要在要分行的地方加\即可。）第二行的冒号表明 `trans` 依赖于 `OBJS` 所代表的所有文件。这样一个有依赖性的行的后面可以紧跟一个命令来创建冒号左边的文件。第三行则告诉我们目标文件 `trans` 是通过将目标文件 `lexer.o`，`parser.o`，…，`main.o` 连接在一起而创建的。但是，*make* 知道它必须先创建这些目标文件，而且它能像下面这样自动地完成这些工作：它先找到相应的源文件 `lexer.c`，`parser.c`，…，`main.c`，然后用 C 编译器编译它们得到相应的目标文件。最后一行说的是所有的7个目标文件都依赖于全局头文件 `global.h`。

要建立翻译器，只要键入 *make* 命令即可，它将执行如下命令：

```
cc -c lexer.c
cc -c parser.c
cc -c emitter.c
cc -c symbol.c
cc -c init.c
cc -c error.c
cc -c main.c
cc lexer.o parser.o emitter.o symbol.o\
       init.o error.o main.o -o trans
```

完成之后，如果上一遍编译之后依赖的源文件被改动了，编译过程将会重新执行一次。Kernighan and Pike[1984]中包括一些利用 *make* 帮助构建编译器的例子。 □

描述器（profiler）是另一个有用的编译器编写工具。一旦编译器完成后，可以用描述器来确定编译器在编译源文件时主要把时间用在哪些文件上，从而能确定编译器的热点部分。对热

点部分进行修改可以使编译器的速度提高2到3倍。

除了软件开发工具以外，有很多专门针对编译器开发过程的工具也已经被开发出来了。在3.5节中，我们曾描述过生成器 Lex，它能根据词法分析器的正规表达式的说明自动地生成词法分析器；在4.9节，我们描述了生成器 Yacc，它能根据语言的语法描述自动地生成 LR 语法分析器。上面描述的 *make* 命令在需要的时候将自动调用 Lex 和 Yacc。除了词法分析器和语法分析器的生成器外，已经开发出来的构建编译器部件的辅助工具还有属性文法生成器和代码生成器的生成器。许多这样的编译器构建工具都有一个有用的性质，即能够捕获编译器规范中的bug。

730 在编译器构建问题上，有一些关于程序生成器的效率和方便程度的争论（Waite and Carter[1985]）。人们已经认同的事实是，好的程序生成器对生成可靠的编译器组件有很大的帮助。与用手工实现语法分析器相比，利用语言的语法描述和一个语法分析器生成器来生成一个正确的语法分析器要容易得多。然而，一个重要的问题是这些生成器之间以及它们和程序之间的接口是怎样的。在设计生成器时最普遍的错误是将它假定为设计的核心。一种较好的设计可以让生成器产生具有简明接口的子程序，其他程序可以通过这些接口调用相应的子程序（Johnson and Lesk[1978]）。

11.4 测试与维护

编译器必须生成正确的代码。最理想的情况是，我们有一台计算机，它能机械地验证编译器是否完全按照其规范实现。有几篇论文确实讨论了各种编译算法的正确性，但不幸的是，编译器基本上并不是按规范来实现的，因此也就无法检查编译器的任意一个实现是否违反了其正式规范。由于编译器通常都是相当复杂的程序，因此还存在验证编译器规范本身的正确性的问题。

实际上，我们必须借助一些系统方法来测试编译器，以此增强我们对编译器使用的信心，相信它们在该领域中能令人满意地工作。“回归”测试是一种比较成功的方法，很多编译器编写者都采用了该方法。在这种方法中，我们维护了一套测试程序，只要编译器进行了改动，这些测试程序就分别用新、老版本的编译器编译。两个编译器编译所得的目标程序的任何差别都会向编译器编写者报告。还可以用 UNIX 系统命令 *make* 来将该测试自动化。

选择将什么程序放入测试包中是一个很难的问题。我们的目标是测试程序能使编译器中的每条语句至少被使用一次。找这样一个测试包需要花极大的心智。人们已经建立了 Fortran、TEX、C 等语言的穷举性测试包。许多编译器编写者向回归测试包中增加在老版本编译器中发现过 bug 的程序。令人失望的是由于对编译器做了新的改正，以前出现过的 bug 还会重新出现。

性能测试也很重要。有些编译器编写者通过在回归测试中做一些计时研究，发现新版编译器生成的代码的质量与老版的基本一样。

编译器维护是另一个重要问题，特别是如果编译器要在不同的环境中运行，或者参加编译器项目的工作人员随时有离开或加入的可能时，这个问题更值得重视。编译器维护的一个关键因素是具有良好的编程风格和完整的文档。作者听说过一个编译器只用了7条注释，其中一条是“该代码很糟糕”。不用说，除了最初的编写者以外，对任何人来说这样的程序都是很难

731 维护的。

Knuth[1984b] 开发了一个称为 WEB 的系统，该系统用于解决为大型 Pascal 程序编写文档的问题。WEB 使得可读性编程变得容易了；在代码写出来的同时文档也被开发出来了，不需

732 要以后添加。WEB 中的许多思想同样可以很好地用于其他语言。

第12章　编译器实例

本章将讨论一些现有编译器的结构，这些编译器对应的语言有文本格式化语言、Pascal、C、Fortran、Bliss 和 Modula 2。在此列出这些编译器并不是提倡采用它们的设计方式而否定其他设计方式，我们只是想给大家展示编译器实现中可能存在的不同之处。

选择 Pascal 编译器是因为它们影响了语言本身的设计，选择 C 编译器是因为 C 是 UNIX 系统上最主要的编程语言，选择 Fortran H 编译器是因为它在优化技术的发展方面有着巨大的影响，BLISS/11 则被用来展示以优化空间为目标的编译器的设计，选择 DEC Modula 2 编译器则是因为它采用相对简单的技术却能产生优质的代码，并且它是仅由一个人在几个月的时间内完成的。

12.1　数学排版预处理器EQN

许多计算机程序可能的输入所构成的集合可以视为一个小语言。该集合的结构可以用一个文法来描述，而且语法制导翻译可以用来精确地指定该程序的行为。这样的话，我们就可以利用编译器技术来实现该程序。

UNIX 编程环境下的第一个小语言编译器是由 Kernighan 和 Cherry[1975] 编写的 EQN。正如1.2节所简要描述的那样，EQN 以“`E sub 1`”这样的形式作为输入并产生命令，以此作为文本格式化工具 TROFF 的输入，最终产生“E_1”这种形式的输出。

EQN 的实现如图12-1所示。宏预处理（见1.4节）和词法分析器是一起实现的。词法分析后的记号流会被语法分析翻译成文本格式化命令。翻译器是使用4.9节描述的语法分析器生成器 Yacc 构建的。

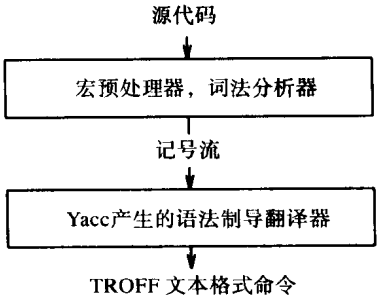


图12-1　EQN的实现

笔者注意到，将 EQN 的输入看成一个语言并使用编译器技术来构造翻译器的方法有几个好处。

1. 易于实现。“构建一个能对大量例子进行试验的工作系统在过去也许要一个人-月。”

2. 语言的演化。语法制导定义促进了输入语言的改变。这些年为适应用户需求 EQN 在不断地完善发展。

作者最后得出的结论是“定义一个语言，并使用编译器/编译器来为其构建编译器看起来似乎是惟一明智的办法。”

12.2　Pascal编译器

正如 Wirth[1971] 中所谈到的那样，Pascal 的设计与其第一个编译器的开发是“相互依赖的”。所以，查看一下由 Wirth 和他的合作者们为该语言编写的编译器的结构就具有指导意义。第一个（Wirth[1971]）和第二个（Ammann[1981,1977]）Pascal 编译器为 CDC 6000 系列机生成绝对的机器代码。对第二个编译器做的可移植性试验导致了为抽象栈式机器生成代码的

Pascal-P 编译器（Nori et al. [1981]）的出现，它生成的代码称为 P 代码。

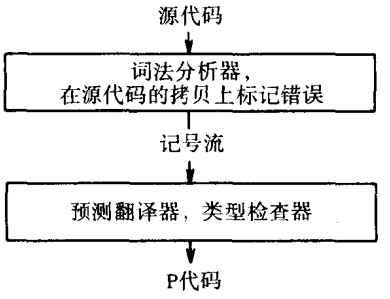


图12-2 Pascal-P编译器

上述的每一个编译器都是围绕递归下降语法分析器组织起来的单遍编译器，它类似于第2章中的“小型”前端。Wirth[1971]注意到，“实践证明，根据语法分析方法的限制来定制语言相对比较容易。”Pascal-P 编译器的组织如图12-2所示。

Pascal-P 编译器所使用的抽象栈式机器中的基本操作反映了 Pascal 的需求。机器的存储区域分为四部分：

1. 过程代码。
2. 常数。
3. 活动记录栈。
4. 使用 **new** 操作符进行分配的数据的堆[⊖]。

由于 Pascal 的过程可以嵌套，过程的活动记录包含访问链和控制链。过程调用被翻译成抽象机器的“标记栈”指令，其参数为访问链和控制链。过程代码通过使用到活动记录结尾处的偏移来实现对局部名字存储位置的引用。对非局部量存储位置的引用则通过一个序对，该序对由要被遍历的访问链的数量和一个偏移组成，如7.4节所述。第一个编译器使用 display 表来实现对非局部量的高效访问。

Ammann[1981]根据编写第二个编译器的经验得出下面的结论。一方面，单遍编译器易于实现并产生适度的输入/输出活动（过程体的代码在内存中编译并作为一个单位写到二级存储器上）。另一方面，单遍组织结构“对生成的代码质量的限制太大且有相对较高的存储要求。”

12.3 C编译器

C 是 D. M. Ritchie 设计的一种通用编程语言，而且在UNIX操作系统（Ritchie and Thompson [1974]）上已经成为一种主要的编程语言。UNIX 本身是用 C 实现的并被移植到从微处理器到大型机的许多机器上，移植过程只要求先移植一个 C 编译器。本节我们将简要描述一下 Ritchie [1979] 为 PDP-11 编写的编译器的总体结构和 Johnson[1979] 编写的可移植的 C 编译器的 PCC 系列。PCC 代码的四分之三是独立于目标机器的。所有的这些编译器本质上都是两遍的；PDP-11 编译器有一个可供选择的第三遍，这一遍将对汇编语言输出做一些优化，如图12-3所示。这一窥孔优化阶段将消除冗余

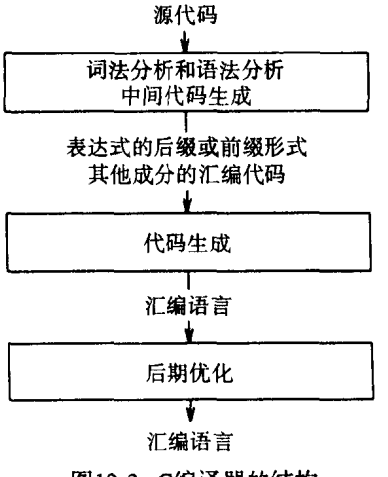


图12-3 C编译器的结构

⊖ 用它所编译的子集编写的编译器使用像栈那样的堆可以很容易地实现自举，所以初始时可以使用一个简单的堆管理器。

的或不可达的语句。

各种编译器的第一遍都完成词法分析、语法分析和中间代码生成。PDP-11 编译器采用递归下降法分析除表达式以外的一切语法成分，对表达式的分析采用的是算符优先法。中间代码由表达式的后缀表示和控制流语句的汇编代码组成。PCC 使用 Yacc 生成的 LALR(1) 语法分析器，它的中间代码由表达式的前缀表示和其他结构的汇编代码组成。在每种情况下，局部名字的存储分配都是在第一遍中完成的，因此使用活动记录的偏移就可以引用这些局部名字。

在后端，表达式用语法树表示。PDP-11 编译器通过对树的遍历来生成代码，遍历算法使用类似9.10节中描述的标记算法。对该算法已经做了一些修改以确保寄存器对对需要它们的操作是可用的，并且可以利用常量操作数。

Johnson[1978] 总结了 PCC 的理论的影响。PCC 以及其后续版本的编译器 PCC2 都通过重写树来生成表达式的代码。PCC 的代码生成器每次检查一条源语言语句，重复地找出只用可用寄存器而不必使用存储空间就能完成计算的最大子树。在9.10节计算出的标号识别出要计算的子表达式并将它们存到临时变量中。计算并存储这些子树所表示的值的代码是在这些子树被选定时由编译器生成的。PCC2 中的重写更明显，其代码生成器基于9.11节的动态规划算法。

733 ~ 736

Johnson and Ritchie[1981] 中描述了目标机器对活动记录和过程调用/返回序列设计的影响。标准库函数 `printf` 可以有可变数量的参数，因此某些机器上的调用序列的设计取决于是否需要允许变长参数列表。

12.4 Fortran H 编译器

最初的 Fortran H 编译器是Lowry和Medlock [1969] 编写的，它是一个使用广泛、功能颇为强大的优化编译器，该编译器在构建过程中就已经使用了本书中描述的方法。到目前为止已经进行了几次改进其性能的尝试；“扩展”版的编译器已经在 IBM/370上开发成功；“增强”版的编译器也已经由 Scarborough 和 Kolsky[1980] 开发成功。Fortran H 为用户提供了不做优化、只做寄存器优化和做完全优化等选项。执行完全优化的编译器的梗概如图12-4所示。

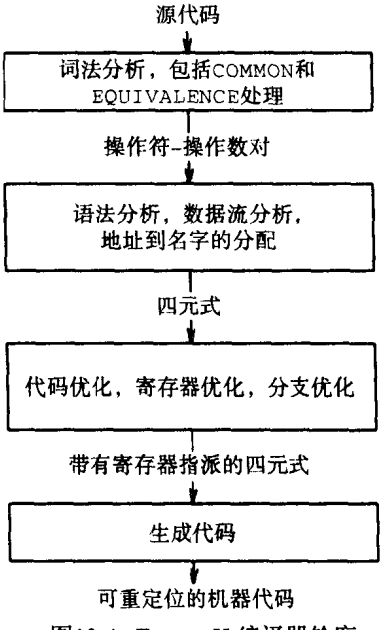


图12-4 Fortran H 编译器轮廓

源程序文本要经过四遍处理。头两遍执行词法分析和语法分析，并产生四元式。接下来将代码优化和寄存器优化一并完成，最后一遍根据四元式和寄存器指派来生成目标代码。

词法分析阶段有点特殊，因为它的输出不是记号流而是“操作符-操作数对”流，操作符-操作数对大致可以理解为一个以非操作数记号为前导的操作数记号。需要注意的是，与其他很多语言一样，Fortran 里不允许连续的两个标识符或者常数这样的操作数记号，这样连续的两个记号至少要用一个标点记号将其分开。

例如，赋值语句

```
A = B(I) + C
```

将被翻译成下面的序对序列：

"赋值语句"	A
=	B
(s	I
)	—
+	C

词法分析阶段需要将引入参数列表或者下标的左括号与把操作数分组的左括号区分开来。因此，符号"(s"表示该左括号是用作下标操作符的。右括号后面不跟任何操作数，而且 Fortran H 并不像区分左括号那样区分右括号的这两种作用。

对 COMMON 和 EQUIVALENCE 语句的处理是同词法分析连在一起的。可以在这一阶段对 COMMON 块做出存储规划，还可以对子程序存储块进行规划，并在某个静态存储区为程序中提到的每个变量确定存储位置。

由于 Fortran 没有像 while 语句这样的结构化控制语句，因此除了对表达式的语法分析外，其他的语法分析都非常简单；而且 Fortran H 也只是简单地运用算符优先分析器来分析表达式。在生成四元式时只是执行一些非常简单的局部优化，例如，用左移操作代替乘以2的幂运算。

12.4.1 Fortran H 中的代码优化

737
~
738

在 Fortran H 中，每个子程序均被划分为基本块。正如10.4节所述，通过在流图中找出那些头支配尾的边即可发现循环结构。编译器执行以下优化：

1. *公共子表达式删除*。编译器会设法找出局部公共子表达式，以及某个块 *B* 和块 *B* 所支配的一个或者多个块的公共子表达式。对其他的公共子表达式则不进行检测。此外，公共子表达式的检测是一次一个表达式，而不是使用10.6节中描述的位向量法。有趣的是，在开发该编译器的"增强"版的过程中，作者发现用位向量法可以大幅提高编译器的速度。

2. *代码外提*。如10.7节所述，将循环不变语句从循环中移出。

3. *复制传播*。这也是按一次一条复制语句的方式来完成的。

4. *归纳变量删除*。这一优化过程只是针对在循环中只被赋值一次的变量。这里消除归纳变量使用的方法不是用10.7节中描述的"族"法，而是通过多遍扫描来检测属于其他归纳变量族的归纳变量。

尽管数据流分析是按一次一个的方式进行的，但是我们称为 *in* 和 *out* 的相应的值是以位向量的方式来存储的。然而，在最初的编译器中这种向量的长度被限制为127位，因此大程序中的优化仅涉及最常引用的变量。增强版的编译器没有消除这一限制而是提高了限制值。

12.4.2 代数优化

由于 Fortran 常用于数值计算，因此代数优化是很危险的，因为在计算机运算中，表达式的变换可能引起溢出或者精度的丢失，而且如果我们将代数化简看得过于理想的话，我们将忽视这些后果。但是，与整数有关的代数变换一般来说还是安全的，而且增强版的编译器只在存在数组引用的情况下才执行这种优化。

一般地，像 A(I, J, K)这样的数组引用会涉及到偏移量的计算，该偏移量用形如 aI+bJ+cK+d 的表达式来计算；式中常数的精确值依赖于 A 的位置和数组的维数。如果 I 和 K 也是常数，不管是数值常数还是循环不变变量，则编译器利用交换律和结合律可以得到表达式 bJ+e，其中 $e = a$I+cK+d。

12.4.3 寄存器优化

Fortran H 将寄存器分为三类。这些寄存器集分别用于局部寄存器优化、全局寄存器优化

和“分支优化”。编译器可以对每一类寄存器中寄存器的确切数目适当地进行调整。 739

全局寄存器逐个循环地被分配给该循环中引用最频繁的变量。如果一个变量在某个循环 L 中适合放到寄存器中，但在包含 L 的直接外层循环中却不适合为其分配寄存器，则该变量在 L 的前置首节点会被装入寄存器，而在 L 的出口则会被保存到内存中。

局部寄存器在基本块中被用于保存某个语句的结果，直到它被用于新的语句。仅当没有足够的局部寄存器时，临时变量才会被保存到主存中。如果某操作数后来变成了无用操作数，编译器将会用保存该操作数的寄存器来计算新的值。增强版的 Fortran H 编译器试图找出全局寄存器可以和其他寄存器交换的情况，以增加在寄存器（保存了其操作数）中执行运算的次数。

分支优化是在 IBM/370 指令集中使用的一种技巧，这种技术为了保险，规定跳转的地址只限于寄存器可以表达的数值加上一个0到4095的常数。因此，Fortran H 在代码空间中以4096个字节的间隔分配一些寄存器来保存地址，以便在庞大的程序中实现高效的跳转。

12.5 BLISS/11编译器

该编译器在PDP-11上实现了系统程序设计语言 Bliss（Wulf et al. [1975]）。在某种意义上，该编译器是一个优化编译器，但是由于该编译器只是进行存储优化以减少存储空间，而不是时间，因此在今天看来，它已经没有实际应用价值了。然而，该编译器大多数优化操作同样也节省了时间开销，而且该编译器的后续版本目前仍在使用。

有几个原因使该编译器值得引起我们的注意。首先，其优化能力强，而且它实现了许多几乎在别的地方找不到的变换。其次，它率先将语法制导方法用于优化，正如10.5节所讨论的那样，也就是说，Bliss 语言被设计成只产生可约流图（它没有 goto 语句）。因此就有可能在分析树而不是在流图上直接执行数据流分析。

该编译器在一遍中完成编译，其中每一个过程都在下一个输入之前被完成。设计者们认为这个编译器由5部分组成，如图12-5所示。

LEXSYNFLO 执行词法分析和语法分析。这里使用的是一个递归下降语法分析器。由于BLISS不允许有 goto 语句，因此所有 BLISS 过程的流图都是可约的。事实上，当我们进行语法分析时，该语言的语法使我们能够建立流图，并确定其中的循环和循环入口。LEXSYNFLO 就是这样工作的，而且利用可约流图的结构它可以确定公共子表达式以及 ud- 链和 du- 链变量。LEXSYNFLO 的另一个重要工作是检测相似表达式组。这些表达式组有可能被单个子程序代替。注意，这种替换会使程序运行得更慢，但是可以节约空间。 740

DELAY 模块检查语法树，以确定诸如不变代码外提、公共子表达式删除这样的常见优化中，哪一种在实际中可以带来好处。表达式的计算顺序也被同时确定，确定的依据是9.10节中描述的标记策略，该策略经过修改，可以用于标识因保存公共子表达式的值而不可用的寄存器。代数法则被用来确定是否要进行运算重组。正如8.4节所述，条件表达式的计算既可以是数值型，也可以由控制流决定，而且由 DELAY 确定哪种模式代价更低。

TNBIND 模块考虑哪个临时变量名应与寄存器绑定，寄存器和内存单元都要进行分配。分配所使用的策略是，首先将语法树中被赋予同一寄存器的节点分成一组。正如9.6节所讨论的，在其父节点所使用的同一个寄存器中计算一个节点是有好处的。接下来，将临时变量放入寄存器，如果该变量在小范围内被多次使用，那么这种做法是有好处的。根据给最有利的节点安排寄存器的顺序，寄存器会不断地被指派出去，直到被指派完毕。CODE 模块将该树，包括其顺序和寄存器分配信息，转换成可重定位的机器代码。

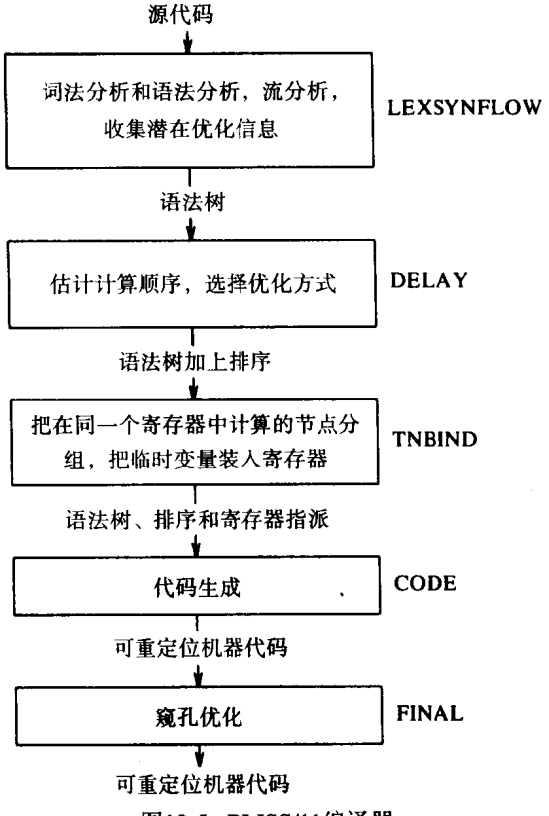


图12-5 BLISS/11编译器

接下来，代码会被执行窥孔优化的 FINAL 模块重复检查，直到不能对其进行进一步的改进为止。这些改进包括消除多重跳转（不管是条件的还是无条件的）以及条件式求反，正如9.9节所讨论的那样。

冗余代码或者不可达指令也会被消除（这可能是由其他 FINAL 优化完成的）。我们试图合并一个分支的两条路径上的相似代码序列，这个过程就像常量的局部传播。我们还试图进行许多别的局部优化，其中某些优化的硬件相关性很强。一个重要的局部优化是尽可能用 PDP-11 “分支”指令替换跳转指令，“分支”指令至少含一个字但被限制在128个字以内。

12.6 Modula-2优化编译器

该编译器在 Powell[1984] 中有所描述，其开发目的为使用代价小、回报高的优化来产生优质代码。作者把他的策略描述为寻找“最好的简易”优化方法。这种想法实现起来很困难。不经过试验和测试，很难预先知道哪个是“最好的简易”优化，而且 Modula-2 编译器中的一些做法可能不适合要进行最大程度优化的编译器。尽管如此，这种策略确实达到了作者产生优秀代码的预期目标，而使用的只是一个一个人在几个月的时间内编写出的编译器。该编译器前端的5遍如图12-6所示。

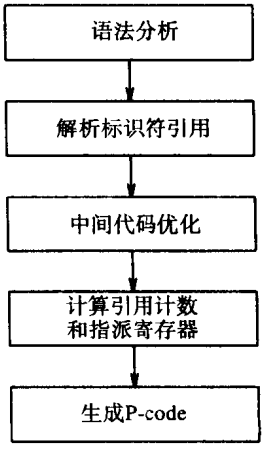


图12-6 Modula-2编译器的遍

该编译器的语法分析器是用 Yacc 生成的，而且因为 Modula-2 的变量不需要在使用前进行声明，因此它在两遍内即可产生语法树。有人尝试使该编译器与已有的工具兼容。为了与众多的 Pascal 编译器相容，其中间代码是 P 代码。由于该编译器的过程调用格式与运行于 Berkeley UNIX 下的 Pascal 编译器和 C 编译器的过程调用格式一致，因此用这三种语言编写的过程可以很容易地集成在一起。

该编译器不进行数据流分析。Modula-2 是一种只产生可约流图的语言，这与 Bliss 十分相似，因此10.5节中的方法同样可以用到这里来。实际上，Modula 编译器在语法的使用上优于 Bliss 编译器。其中循环是通过语法（即编译器去寻找 while 和 for 结构）来识别的。根据表达式中没有变量是在循环中定义的事实可以检测不变表达式，这些不变表达式将会被提到循环头部。惟一被检测的归纳变量是 for 循环索引族中的变量。当一个公共子表达式位于某个块中，而该块支配所有其他块时，我们就可以检测出全局公共子表达式，但这种分析只能是一次一个表达式，而不是采用位向量的方式。

寄存器分配策略设计得比较简单，它只是完成一些合理的工作，而不是进行彻底的分配。实际上，它只将下列变量看作分配给寄存器的候选变量：

1. 表达式（指那些获得第一优先权的表达式）计算期间所用到的临时变量。
2. 公共子表达式的值。
3. for 循环的索引和上限值。
4. 在形如 `with` *E* `do` 的表达式中的 *E* 的地址。
5. 当前过程的局部简单变量（字符、整数等）。

741 ~ 743

我们还试图估计将上述2~5类变量保存在寄存器中的好处到底有多大。假定一个嵌套在 d 层循环中的语句被执行 10^d 次。引用不超过两次的变量我们不予考虑，其他变量按照估算的使用次数来制定等级，如果为表达式的临时变量和具有高等级的变量指派完寄存器后还有寄存器可用，就按照等级也为它们指派寄存器。

744

附录 一个程序设计项目

A.1 引言

本附录为基于本书的编译器设计课程提供了一个用于编程实验的编程练习。该练习是由Pascal 子集的编译器基本组件的实现组成的。这个子集是最小的，但能将7.1节的递归排序过程这样的程序表达清楚。作为现存语言的一个子集它具有一定的实用性。用该子集编写的程序的意义是由 Pascal 语言的语义确定的（Jensen and Wirth[1975]）。如果你手边有 Pascal 编译器，可以用它来检验作为练习编写的该编译器的工作是否符合要求。子集中的各种结构在许多编程语言中都可以找到，因此如果你没有 Pascal 编译器，那么使用别的语言也可以检验相应的练习。

A.2 程序结构

一个程序由一系列全局数据声明、一系列过程和函数声明以及一个复合语句即“主程序”组成。全局数据采用静态存储分配，过程和函数的局部数据采用栈式存储分配，程序允许递归，参数传递方式为传地址。我们假定 `read` 和 `write` 这两个过程由编译器提供。

图A-1给出了一个程序的例子。程序名为 `example`，而 `input` 和 `output` 分别是 `read` 和 `write` 使用的文件名。

```
program example(input, output);
var x, y: integer;
function gcd(a, b: integer): integer;
begin
    if b = 0 then gcd := a
    else gcd := gcd(b, a mod b)
end;

begin
    read(x, y);
    write(gcd(x, y))
end.
```

图A-1 一个程序例子

A.3 Pascal子集的语法

下面列出的是Pascal子集的 LALR(1)文法。利用2.4节和4.3节所给出的技术，通过消除左递归，可以对该文法进行改造以适用于递归下降语法分析。通过替换掉 **relop**、**addop** 和 **mulop** 并消除 ϵ 产生式，可以为表达式构造一个算符优先语法分析器。

加入产生式

statement → **if** ***expression*** **then** ***statement***

会引入悬空 else 二义性，但可以按照4.3节的方法将其消除（如果采用预测语法分析法也可以

参照例4.19)。

简单变量与不带参数的函数调用之间没有语法区别。二者都用下面的产生式来生成：

factor → **id**

因此，如果 b 已被声明为一个函数，则赋值语句 a := b 将把 a 置为函数 b 返回的值。

program →
    program id (*identifier_list*) ;
    declarations
    subprogram_declarations
    compound_statement
    .

identifier_list →
    id
    | *identifier_list* , **id**

declarations →
    declarations **var** *identifier_list* : *type* ;
    | ϵ

745
≀
746

type →
    standard_type
    | **array** [**num** .. **num**] **of** *standard_type*

standard_type →
    integer
    | **real**

subprogram_declarations →
    subprogram_declarations *subprogram_declaration* ;
    | ϵ

subprogram_declaration →
    subprogram_head *declarations* *compound_statement*

subprogram_head →
    function id *arguments* : *standard_type* ;
    | **procedure id** *arguments* ;

arguments →
    (*parameter_list*)
    | ϵ

parameter_list →
    identifier_list : *type*
    | *parameter_list* ; *identifier_list* : *type*

compound_statement →
    begin
    optional_statements
    end

optional_statements →
    statement_list
    | ϵ

statement_list →
    statement
    | *statement_list* ; *statement*

statement →

```
        variable assignop expression
      | procedure_statement
      | compound_statement
      | if expression then statement else statement
      | while expression do statement

variable →
        id
      | id [ expression ]

procedure_statement →
        id
      | id ( expression_list )                                    747

expression_list →
        expression
      | expression_list , expression

expression →
        simple_expression
      | simple_expression relop simple_expression

simple_expression →
        term
      | sign term
      | simple_expression addop term

term →
        factor
      | term mulop factor

factor →
        id
      | id ( expression_list )
      | num
      | ( expression )
      | not factor

sign →
        + | -
```

A.4 词法规则

记号的表示来自3.3节。

1. 注释用{ 和 }括起来。注释体中不能含有{ 。注释可以出现在任何记号的后面。

2. 记号间的空格可有可无，但关键字前后必须有空格、换行、程序的开头或者结尾的圆点。

3. 标识符的记号 **id** 与以字母开头的字母数字串相匹配：

```
letter → [a-zA-Z]
digit → [0-9]
id → letter ( letter | digit )*
```

实现者可能希望对标识符的长度加以限制。

4. 记号**num**与无符号整数相匹配（参见例3.5）：

```
            digits → digit digit*
optional_fraction → . digits | ε
optional_exponent → ( E ( + | - | ε ) digits ) | ε
               num → digits optional_fraction optional_exponent     748
```

5. 关键字要被保留而且在文法中以黑体字出现。

6. 关系运算符（**relop**）是指：=、<>、<、<=、>= 和 >。注意<>表示≠。

7. **addop** 指的是 +、- 和 or。

8. **mulop** 指的是 *、/、div、mod 和 and。

9. 记号 **assignop** 的词素是 :=。

A.5 推荐练习

适合一学期课程的编程练习是为上面定义的语言或者别的高级语言的相似子集编写一个解释器。该项目主要包括将源程序翻译成四元式或者栈式机器代码这样的中间表示，然后再对中间表示进行解释。对于模块的构造我们应规划一个顺序。该顺序不同于编译器中模块被执行的顺序，因为用一个能运转的解释器来调试其他编译器组件是比较方便的。

1. *设计符号表*。决定符号表的组织方式。要考虑收集有关名字的信息，但此时还要保证符号表记录结构的灵活性。要编写程序来完成以下工作：

i) 在符号表中查找给定的名字，如果该名字在符号表中不存在，就在符号表中为其创建新的表项，否则返回指向该名字的指针。

ii) 从符号表中删除给定过程的所有局部名字。

2. *为四元式编写解释器*。此时四元式集仍可保持开放而不需精确地确定下来，但应包含对应语言中操作符集合的算术和条件跳转语句。如果条件是用算术方式确定的而不是用程序中的位置确定的，则还应包含逻辑运算。另外，整数到实数的转换、过程开头和结尾的标记以及参数传递和过程调用，预计都会用到四元式。

此时还需要为被解释程序设计调用序列和运行时的组织方式。7.3节讨论的简单栈式组织适用于本示例语言，因为在该语言中不允许过程声明的嵌套；也就是说，变量或者是全局的（在整个程序的级别上声明的）或者对简单过程来说是局部的。

为简单起见，可以用另一个高级语言来代替该解释器。每个四元式可以是 C，甚至 Pascal 这样的高级语言的一条语句。那么编译器的输出就是一系列 C 语言语句，于是就可以用已有
749 的 C 语言编译器来编译。该方法使得实现者能够将精力集中在运行时的组织方式上。

3. *编写词法分析器*。为记号选择内码，决定常数在编译器中的表示方式，统计行数以供后面的错误处理程序使用，需要的话输出源程序的清单，编写一个将关键字填入符号表的程序，将词法分析器设计成一个可以由语法分析器调用并返回（记号，属性值）对的子程序。至此，词法分析器所检测到的错误可以通过调用错误打印子程序并终止执行来进行处理。

4. *编写语义动作*。编写生成四元式的语义子程序；为使翻译更容易，在某些地方需要修改文法（关于修改文法的有效方法请参见5.5节和5.6节的例子）；此时即可进行语义分析，必要时将整数转换成实数。

5. *编写语法分析器*。如果手边有 LALR 语法分析器的生成器可用，则将在很大程度上简化语法分析器的编写。如果手边有 Yacc 这样的处理二义性文法的语法分析器生成器，则可以将表示表达式的非终结符合并。而且如果发生移进-归约冲突则执行移进即可消除悬空 else 二义性。

6. *编写错误处理例程*。准备恢复词法错误和语法错误，为词法、语法和语义错误打印错误诊断信息。

7. *评价*。可以用图7-1中的 Pascal 程序，该程序的 `partition` 函数的代码对应于图10-2中 C 程序的标记代码段。如果有描述器（profiler），用它确定一下编译器的热点。考虑一下，

为提高编译器的速度，哪些模块需要修改？

A.6 解释器的改进

为该语言构建翻译器的另一种办法是从实现一个桌面计算器（即表达式的翻译器）开始。然后逐步地向语言中添加结构直到得到整个语言的解释器。Kernighan and Pike[1984]中用过一个类似的方法。建议按如下顺序添加结构：

1. *将表达式翻译成后缀表示*。可以使用第2章中的递归下降语法分析法，也可以使用语法分析器生成器。为熟悉编程环境，你可以试着编写一个将简单算术表达式翻译成后缀表示的翻译器。

2. *添加词法分析器*。在上面构建的翻译器中允许出现关键字、标识符和数字。为翻译器重新确定目标机器以输出栈式机器代码或者四元式。

3. *为中间表示编写解释器*。如A.5节中讨论的那样，高级语言可以用来代替解释器。目前， 750
解释器只需要支持算术操作、赋值和输入/输出。通过允许全局变量声明、赋值语句和对过程 `read` 和 `write` 的调用来扩展语言。这些结构使得我们可以对解释器进行测试。

4. *添加语句*。现在，用该语言编写的程序由一个不含子程序声明的主程序构成。对翻译器和解释器都要进行测试。

5. *添加过程和函数*。现在，符号表必须允许标识符的作用域被限制在过程体中。设计一个调用序列。在此，使用7.3节的简单栈式组织就足够了。对解释器进行扩展以支持该调用序列。

A.7 扩展

不用大幅增加编译的复杂性即可将许多特征添加到该语言中。这些特征包括：

1. 多维数组。
2. for 语句和 case 语句。
3. 块结构。
4. 记录结构。

如果时间允许，将这些扩展中的一个或多个添加到你的编译器中。 751

参考文献

ABEL, N. E. AND J. R. BELL [1972]. "Global optimization in compilers," *Proc. First USA-Japan Computer Conf.*, AFIPS Press, Montvale, N. J.

ABELSON, H. AND G. J. SUSSMAN [1985]. *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, Mass.

ADRION, W. R., M. A. BRANSTAD, AND J. C. CHERNIAVSKY [1982]. "Validation, verification, and testing of computer software," *Computing Surveys* **14:2**, 159-192.

AHO, A. V. [1980]. "Pattern matching in strings," in Book [1980], pp. 325-347.

AHO, A. V. AND M. J. CORASICK [1975]. "Efficient string matching: an aid to bibliographic search," *Comm. ACM* **18:6**, 333-340.

AHO, A. V. AND M. GANAPATHI [1985]. "Efficient tree pattern matching: an aid to code generation," *Twelfth Annual ACM Symposium on Principles of Programming Languages*, 334-340.

AHO, A. V., J. E. HOPCROFT, AND J. D. ULLMAN [1974]. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass.

AHO, A. V., J. E. HOPCROFT, AND J. D. ULLMAN [1983]. *Data Structures and Algorithms*, Addison-Wesley, Reading, Mass.

AHO, A. V. AND S. C. JOHNSON [1974]. "LR parsing," *Computing Surveys* **6:2**, 99-124.

AHO, A. V. AND S. C. JOHNSON [1976]. "Optimal code generation for expression trees," *J. ACM* **23:3**, 488-501.

AHO, A. V., S. C. JOHNSON, AND J. D. ULLMAN [1975]. "Deterministic parsing of ambiguous grammars," *Comm. ACM* **18:8**, 441-452.

AHO, A. V., S. C. JOHNSON, AND J. D. ULLMAN [1977a]. "Code generation for expressions with common subexpressions," *J. ACM* **24:1**, 146-160.

AHO, A. V., S. C. JOHNSON, AND J. D. ULLMAN [1977b]. "Code generation for machines with multiregister operations," *Fourth ACM Symposium on Principles of Programming Languages*, 21-28.

AHO, A. V., B. W. KERNIGHAN, AND P. J. WEINBERGER [1979]. "Awk – a pattern scanning and processing language," *Software—Practice and Experience* **9:4**, 267-280.

AHO, A. V. AND T. G. PETERSON [1972]. "A minimum distance error-correcting parser for context-free languages," *SIAM J. Computing* **1:4**, 305-312.

AHO, A. V. AND R. SETHI [1977]. "How hard is compiler code generation?" Lecture Notes in Computer Science **52**, Springer-Verlag, Berlin, 1-15.

AHO, A. V. AND J. D. ULLMAN [1972a]. "Optimization of straight line code," *SIAM J. Computing* **1:1**, 1-19.

AHO, A. V. AND J. D. ULLMAN [1972b]. *The Theory of Parsing, Translation and Compiling*, Vol. I: *Parsing*, Prentice-Hall, Englewood Cliffs, N. J.

AHO, A. V. AND J. D. ULLMAN [1973a]. *The Theory of Parsing, Translation*

and Compiling, Vol. II: *Compiling*, Prentice-Hall, Englewood Cliffs, N. J.

AHO, A. V. AND J. D. ULLMAN [1973b]. "A technique for speeding up LR(k) parsers," *SIAM J. Computing* **2:2**, 106-127.

AHO, A. V. AND J. D. ULLMAN [1977]. *Principles of Compiler Design*, Addison-Wesley, Reading, Mass.

AIGRAIN, P., S. L. GRAHAM, R. R. HENRY, M. K. MCKUSICK, AND E. PELEGRI-LLOPART [1984]. "Experience with a Graham-Glanville style code generator," *ACM SIGPLAN Notices* **19:6**, 13-24.

ALLEN, F. E. [1969]. "Program optimization," *Annual Review in Automatic Programming* **5**, 239-307.

ALLEN, F. E. [1970]. "Control flow analysis," *ACM SIGPLAN Notices* **5:7**, 1-19.

ALLEN, F. E. [1974]. "Interprocedural data flow analysis," *Information Processing 74*, North-Holland, Amsterdam, 398-402.

ALLEN, F. E. [1975]. "Bibliography on program optimization," RC-5767, IBM T. J. Watson Research Center, Yorktown Heights, N. Y.

ALLEN, F. E., J. L. CARTER, J. FABRI, J. FERRANTE, W. H. HARRISON, P. G. LOEWNER, AND L. H. TREVILLYAN [1980]. "The experimental compiling system," *IBM. J. Research and Development* **24:6**, 695-715.

ALLEN, F. E. AND J. COCKE [1972]. "A catalogue of optimizing transformations," in Rustin [1972], pp. 1-30.

ALLEN, F. E. AND J. COCKE [1976]. "A program data flow analysis procedure," *Comm. ACM* **19:3**, 137-147.

ALLEN, F. E., J. COCKE, AND K. KENNEDY [1981]. "Reduction of operator strength," in Muchnick and Jones [1981], pp. 79-101.

AMMANN, U. [1977]. "On code generation in a Pascal compiler," *Software—Practice and Experience* **7:3**, 391-423.

AMMANN, U. [1981]. "The Zurich implementation," in Barron [1981], pp. 63-82.

ANDERSON, J. P. [1964]. "A note on some compiling algorithms," *Comm. ACM* **7:3**, 149-150.

ANDERSON, T., J. EVE, AND J. J. HORNING [1973]. "Efficient LR(1) parsers," *Acta Informatica* **2:1**, 12-39.

ANKLAM, P., D. CUTLER, R. HEINEN, JR., AND M. D. MACLAREN [1982]. *Engineering a Compiler*, Digital Press, Bedford, Mass.

ARDEN, B. W., B. A. GALLER, AND R. M. GRAHAM [1961]. "An algorithm for equivalence declarations," *Comm. ACM* **4:7**, 310-314.

AUSLANDER, M. A. AND M. E. HOPKINS [1982]. "An overview of the PL.8 compiler," *ACM SIGPLAN Notices* **17:6**, 22-31.

BACKHOUSE, R. C. [1976]. "An alternative approach to the improvement of LR parsers," *Acta Informatica* **6:3**, 277-296.

BACKHOUSE, R. C. [1984]. "Global data flow analysis problems arising in locally least-cost error recovery," *TOPLAS* **6:2**, 192-214.

BACKUS, J. W. [1981]. "Transcript of presentation on the history of Fortran I, II, and III," in Wexelblat [1981], pp. 45-66.

BACKUS, J. W., R. J. BEEBER, S. BEST, R. GOLDBERG, L. M. HAIBT, H. L. HER-

RICK, R. A. NELSON, D. SAYRE, P. B. SHERIDAN, H. STERN, I. ZILLER, R. A. HUGHES, AND R. NUTT [1957]. "The Fortran automatic coding system," *Western Joint Computer Conference*, 188-198. Reprinted in Rosen [1967], pp. 29-47.

BAKER, B. S. [1977]. "An algorithm for structuring programs," *J. ACM* **24:1**, 98-120.

BAKER, T. P. [1982]. "A one-pass algorithm for overload resolution in Ada," *TOPLAS* **4:4**, 601-614.

BANNING, J. P. [1979]. "An efficient way to find the side effects of procedure calls and aliases of variables," *Sixth Annual ACM Symposium on Principles of Programming Languages*, 29-41.

BARRON, D. W. [1981]. *Pascal – The Language and its Implementation*, Wiley, Chichester.

BARTH, J. M. [1978]. "A practical interprocedural data flow analysis algorithm," *Comm. ACM* **21:9**, 724-736.

BATSON, A. [1965]. "The organization of symbol tables," *Comm. ACM* **8:2**, 111-112.

BAUER, A. M. AND H. J. SAAL [1974]. "Does APL really need run-time checking?" *Software—Practice and Experience* **4:2**, 129-138.

BAUER, F. L. [1976]. "Historical remarks on compiler construction," in Bauer and Eickel [1976], pp. 603-621. Addendum by A. P. Ershov, pp. 622-626.

BAUER, F. L. AND J. EICKEL [1976]. *Compiler Construction: An Advanced Course*, 2nd Ed., Lecture Notes in Computer Science **21**, Springer-Verlag, Berlin.

BAUER, F. L. AND H. WOSSNER [1972]. "The 'Plankankül' of Konrad Zuse: A forerunner of today's programming languages," *Comm. ACM* **15:7**, 678-685.

BEATTY, J. C. [1972]. "An axiomatic approach to code optimization for expressions," *J. ACM* **19:4**, 714-724. Errata **20** (1973), p. 180 and 538.

BEATTY, J. C. [1974]. "Register assignment algorithm for generation of highly optimized object code," *IBM J. Research and Development* **5:2**, 20-39.

BELADY, L. A. [1966]. "A study of replacement algorithms for a virtual storage computer," *IBM Systems J.* **5:2**, 78-101.

BENTLEY, J. L. [1982]. *Writing Efficient Programs*, Prentice-Hall, Englewood Cliffs, N. J.

BENTLEY, J. L., W. S. CLEVELAND, AND R. SETHI [1985]. "Empirical analysis of hash functions," manuscript, AT&T Bell Laboratories, Murray Hill, N. J.

BIRMAN, A. AND J. D. ULLMAN [1973]. "Parsing algorithms with backtrack," *Information and Control* **23:1**, 1-34.

BOCHMANN, G. V. [1976]. "Semantic evaluation from left to right," *Comm. ACM* **19:2**, 55-62.

BOCHMANN, G. V. AND P. WARD [1978]. "Compiler writing system for attribute grammars," *Computer J.* **21:2**, 144-148.

BOOK, R. V. [1980]. *Formal Language Theory*, Academic Press, New York.

BOYER, R. S. AND J S. MOORE [1977]. "A fast string searching algorithm,"

Comm. ACM **20:10**, 262-272.

BRANQUART, P., J.-P. CARDINAEL, J. LEWI, J.-P. DELESCAILLE, AND M. VANBEGIN [1976]. *An Optimized Translation Process and its Application to Algol 68*, Lecture Notes in Computer Science, **38**, Springer-Verlag, Berlin.

BRATMAN, H. [1961]. "An alternate form of the 'Uncol diagram'," *Comm. ACM* **4:3**, 142.

BROOKER, R. A. AND D. MORRIS [1962]. "A general translation program for phrase structure languages," *J. ACM* **9:1**, 1-10.

BROOKS, F. P., JR. [1975]. *The Mythical Man-Month*, Addison-Wesley, Reading, Mass.

BROSGOL, B. M. [1974]. *Deterministic Translation Grammars*, Ph. D. Thesis, TR 3-74, Harvard Univ., Cambridge, Mass.

BRUNO, J. AND T. LASSAGNE [1975]. "The generation of optimal code for stack machines," *J. ACM* **22:3**, 382-396.

BRUNO, J. AND R. SETHI [1976]. "Code generation for a one-register machine," *J. ACM* **23:3**, 502-510.

BURSTALL, R. M., D. B. MACQUEEN, AND D. T. SANNELLA [1980]. "Hope: an experimental applicative language," *Lisp Conference*, P.O. Box 487, Redwood Estates, Calif. 95044, 136-143.

BUSAM, V. A. AND D. E. ENGLUND [1969]. "Optimization of expressions in Fortran," *Comm. ACM* **12:12**, 666-674.

CARDELLI, L. [1984]. "Basic polymorphic typechecking," Computing Science Technical Report 112, AT&T Bell Laboratories, Murray Hill, N. J.

CARTER, L. R. [1982]. *An Analysis of Pascal Programs*, UMI Research Press, Ann Arbor, Michigan.

CARTWRIGHT, R. [1985]. "Types as intervals," *Twelfth Annual ACM Symposium on Principles of Programming Languages*, 22-36.

CATTELL, R. G. G. [1980]. "Automatic derivation of code generators from machine descriptions," *TOPLAS* **2:2**, 173-190.

CHAITIN, G. J. [1982]. "Register allocation and spilling via graph coloring," *ACM SIGPLAN Notices* **17:6**, 201-207.

CHAITIN, G. J., M. A. AUSLANDER, A. K. CHANDRA, J. COCKE, M. E. HOPKINS, AND P. W. MARKSTEIN [1981]. "Register allocation via coloring," *Computer Languages* **6**, 47-57.

CHERNIAVSKY, J. C., P. B. HENDERSON, AND J. KEOHANE [1976]. "On the equivalence of URE flow graphs and reducible flow graphs," *Proc. 1976 Conference on Information Sciences and Systems*, Johns Hopkins Univ., 423-429.

CHERRY, L. L. [1982]. "Writing tools," *IEEE Trans. on Communications* **COM-30:1**, 100-104.

CHOMSKY, N. [1956]. "Three models for the description of language," *IRE Trans. on Information Theory* **IT-2:3**, 113-124.

CHOW, F. [1983]. *A Portable Machine-Independent Global Optimizer*, Ph. D. Thesis, Computer System Lab., Stanford Univ., Stanford, Calif.

CHOW, F. AND J. L. HENNESSY [1984]. "Register allocation by priority-based coloring," *ACM SIGPLAN Notices* **19:6**, 222-232.

CHURCH, A. [1941]. *The Calculi of Lambda Conversion*, Annals of Math. Stu-

dies, No. 6, Princeton University Press, Princeton, N. J.

CHURCH, A. [1956]. *Introduction to Mathematical Logic*, Vol. I, Princeton University Press, Princeton, N. J.

CIESINGER, J. [1979]. "A bibliography of error handling," *ACM SIGPLAN Notices* **14:1**, 16-26.

COCKE, J. [1970]. "Global common subexpression elimination," *ACM SIGPLAN Notices* **5:7**, 20-24.

COCKE, J. AND K. KENNEDY [1976]. "Profitability computations on program flow graphs," *Computers and Mathematics with Applications* **2:2**, 145-159.

COCKE, J. AND K. KENNEDY [1977]. "An algorithm for reduction of operator strength," *Comm. ACM* **20:11**, 850-856.

COCKE, J. AND J. MARKSTEIN [1980]. "Measurement of code improvement algorithms," *Information Processing 80*, 221-228.

COCKE, J. AND J. MILLER [1969]. "Some analysis techniques for optimizing computer programs," *Proc. 2nd Hawaii Intl. Conf. on Systems Sciences*, 143-146.

COCKE, J. AND J. T. SCHWARTZ [1970]. *Programming Languages and Their Compilers: Preliminary Notes, Second Revised Version*, Courant Institute of Mathematical Sciences, New York.

COFFMAN, E. G., JR. AND R. SETHI [1983]. "Instruction sets for evaluating arithmetic expressions," *J. ACM* **30:3**, 457-478.

COHEN, R. AND E. HARRY [1979]. "Automatic generation of near-optimal linear-time translators for non-circular attribute grammars," *Sixth ACM Symposium on Principles of Programming Languages*, 121-134.

CONWAY, M. E. [1963]. "Design of a separable transition diagram compiler," *Comm. ACM* **6:7**, 396-408.

CONWAY, R. W. AND W. L. MAXWELL [1963]. "CORC – the Cornell computing language," *Comm. ACM* **6:6**, 317-321.

CONWAY, R. W. AND T. R. WILCOX [1973]. "Design and implementation of a diagnostic compiler for PL/I," *Comm. ACM* **16:3**, 169-179.

CORMACK, G. V. [1981]. "An algorithm for the selection of overloaded functions in Ada," *ACM SIGPLAN Notices* **16:2** (February) 48-52.

CORMACK, G. V., R. N. S. HORSPOOL, AND M. KAISERSWERTH [1985]. "Practical perfect hashing," *Computer J.* **28:1**, 54-58.

COURCELLE, B. [1984]. "Attribute grammars: definitions, analysis of dependencies, proof methods," in Lorho [1984], pp. 81-102.

COUSOT, P. [1981]. "Semantic foundations of program analysis," in Muchnick and Jones [1981], pp. 303-342.

COUSOT, P. AND R. COUSOT [1977]. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," *Fourth ACM Symposium on Principles of Programming Languages*, 238-252.

CURRY, H. B. AND R. FEYS [1958]. *Combinatory Logic*, Vol. I, North-Holland, Amsterdam.

DATE, C. J. [1986]. *An Introduction to Database Systems*, 4th Ed., Addison-Wesley, Reading, Mass.

DAVIDSON, J. W. AND C. W. FRASER [1980]. "The design and application of a

retargetable peephole optimizer," *TOPLAS* **2:2**, 191-202. Errata **3:1** (1981) 110.

DAVIDSON, J. W. AND C. W. FRASER [1984a]. "Automatic generation of peephole optimizations," *ACM SIGPLAN Notices* **19:6**, 111-116.

DAVIDSON, J. W. AND C. W. FRASER [1984b]. "Code selection through object code optimization," *TOPLAS* **6:4**, 505-526.

DEREMER, F. [1969]. *Practical Translators for LR(k) Languages*, Ph. D. Thesis, M.I.T., Cambridge, Mass.

DEREMER, F. [1971]. "Simple LR(*k*) grammars," *Comm. ACM* **14:7**, 453-460.

DEREMER, F. AND T. PENNELLO [1982]. "Efficient computation of LALR(1) look-ahead sets," *TOPLAS* **4:4**, 615-649.

DEMERS, A. J. [1975]. "Elimination of single productions and merging of nonterminal symbols in LR(1) grammars," *J. Computer Languages* **1:2**, 105-119.

DENCKER, P., K. DURRE, AND J. HEUFT [1984]. "Optimization of parser tables for portable compilers," *TOPLAS* **6:4**, 546-572.

DERANSART, P., M. JOURDAN, AND B. LORHO [1984]. "Speeding up circularity tests for attribute grammars," *Acta Informatica* **21**, 375-391.

DESPEYROUX, T. [1984]. "Executable specifications of static semantics," in Kahn, MacQueen, and Plotkin [1984], pp. 215-233.

DIJKSTRA, E. W. [1960]. "Recursive programming," *Numerische Math.* **2**, 312-318. Reprinted in Rosen [1967], pp. 221-228.

DIJKSTRA, E. W. [1963]. "An Algol 60 translator for the X1," *Annual Review in Automatic Programming* **3**, Pergamon Press, New York, 329-345.

DITZEL, D. AND H. R. MCLELLAN [1982]. "Register allocation for free: the C machine stack cache," *Proc. ACM Symp. on Architectural Support for Programming Languages and Operating Systems*, 48-56.

DOWNEY, P. J. AND R. SETHI [1978]. "Assignment commands with array references," *J. ACM* **25:4**, 652-666.

DOWNEY, P. J., R. SETHI, AND R. E. TARJAN [1980]. "Variations on the common subexpression problem," *J. ACM* **27:4**, 758-771.

EARLEY, J. [1970]. "An efficient context-free parsing algorithm," *Comm. ACM* **13:2**, 94-102.

EARLEY, J. [1975a]. "Ambiguity and precedence in syntax description," *Acta Informatica* **4:2**, 183-192.

EARLEY, J. [1975b]. "High level iterators and a method of data structure choice," *J. Computer Languages* **1:4**, 321-342.

ELSHOFF, J. L. [1976]. "An analysis of some commercial PL/I programs," *IEEE Trans. Software Engineering* **SE2:2**, 113-120.

ENGELFRIET, J. [1984]. "Attribute evaluation methods," in Lorho [1984], pp. 103-138.

ERSHOV, A. P. [1958]. "On programming of arithmetic operations," *Comm. ACM* **1:8** (August) 3-6. Figures 1-3 appear in **1:9** (September 1958), p. 16.

ERSHOV, A. P. [1966]. "Alpha – an automatic programming system of high efficiency," *J. ACM* **13:1**, 17-24.

ERSHOV, A. P. [1971]. *The Alpha Automatic Programming System*, Academic

Press, New York.

ERSHOV, A. P. AND C. H. A. KOSTER [1977]. *Methods of Algorithmic Language Implementation*, Lecture Notes in Computer Science **47**, Springer-Verlag, Berlin.

FANG, I. [1972]. "FOLDS, a declarative formal language definition system," STAN-CS-72-329, Stanford Univ.

FARROW, R. [1984]. "Generating a production compiler from an attribute grammar," *IEEE Software* **1** (October) 77-93.

FARROW, R. AND D. YELLIN [1985]. "A comparison of storage optimizations in automatically-generated compilers," manuscript, Columbia Univ.

FELDMAN, S. I. [1979a]. "Make – a program for maintaining computer programs," *Software—Practice and Experience* **9:4**, 255-265.

FELDMAN, S. I. [1979b]. "Implementation of a portable Fortran 77 compiler using modern tools," *ACM SIGPLAN Notices* **14:8**, 98-106.

FISCHER, M. J. [1972]. "Efficiency of equivalence algorithms," in Miller and Thatcher [1972], pp. 153-168.

FLECK, A. C. [1976]. "The impossibility of content exchange through the by-name parameter transmission technique," *ACM SIGPLAN Notices* **11:11** (November) 38-41.

FLOYD, R. W. [1961]. "An algorithm for coding efficient arithmetic expressions," *Comm. ACM* **4:1**, 42-51.

FLOYD, R. W. [1963]. "Syntactic analysis and operator precedence," *J. ACM* **10:3**, 316-333.

FLOYD, R. W. [1964]. "Bounded context syntactic analysis," *Comm. ACM* **7:2**, 62-67.

FONG, A. C. [1979]. "Automatic improvement of programs in very high level languages," *Sixth Annual ACM Symposium on Principles of Programming Languages*, 21-28.

FONG, A. C. AND J. D. ULLMAN [1976]. "Induction variables in very high-level languages," *Third Annual ACM Symposium on Principles of Programming Languages*, 104-112.

FOSDICK, L. D. AND L. J. OSTERWEIL [1976]. "Data flow analysis in software reliability," *Computing Surveys* **8:3**, 305-330.

FOSTER, J. M. [1968]. "A syntax improving program," *Computer J.* **11:1**, 31-34.

FRASER, C. W. [1977]. *Automatic Generation of Code Generators*, Ph. D. Thesis, Yale Univ., New Haven, Conn.

FRASER, C. W. [1979]. "A compact, machine-independent peephole optimizer," *Sixth Annual ACM Symposium on Principles of Programming Languages*, 1-6.

FRASER, C. W. AND D. R. HANSON [1982]. "A machine-independent linker," *Software—Practice and Experience* **12**, 351-366.

FREDMAN, M. L., J. KOMLOS, AND E. SZEMEREDI [1984]. "Storing a sparse table with $O(1)$ worst case access time," *J. ACM* **31:3**, 538-544.

FREGE, G. [1879]. "Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought," in Heijenoort [1967], 1-82.

FREIBURGHOUSE, R. A. [1969]. "The Multics PL/I compiler," *AFIPS Fall Joint*

Computer Conference **35**, 187-208.

FREIBURGHOUSE, R. A. [1974]. "Register allocation via usage counts," *Comm. ACM* **17:11**, 638-642.

FREUDENBERGER, S. M. [1984]. "On the use of global optimization algorithms for the detection of semantic programming errors," NSO-24, New York Univ.

FREUDENBERGER, S. M., J. T. SCHWARTZ, AND M. SHARIR [1983]. "Experience with the SETL optimizer," *TOPLAS* **5:1**, 26-45.

GAJEWSKA, H. [1975]. "Some statistics on the usage of the C language," AT&T Bell Laboratories, Murray Hill, N. J.

GALLER, B. A. AND M. J. FISCHER [1964]. "An improved equivalence algorithm," *Comm. ACM* **7:5**, 301-303.

GANAPATHI, M. [1980]. *Retargetable Code Generation and Optimization using Attribute Grammars*, Ph. D. Thesis, Univ. of Wisconsin, Madison, Wis.

GANAPATHI, M. AND C. N. FISCHER [1982]. "Description-driven code generation using attribute grammars," *Ninth ACM Symposium on Principles of Programming Languages*, 108-119.

GANAPATHI, M., C. N. FISCHER, AND J. L. HENNESSY [1982]. "Retargetable compiler code generation," *Computing Surveys* **14:4**, 573-592.

GANNON, J. D. AND J. J. HORNING [1975]. "Language design for programming reliability," *IEEE Trans. Software Engineering* **SE-1:2**, 179-191.

GANZINGER, H., R. GIEGERICH, U. MONCKE, AND R. WILHELM [1982]. "A truly generative semantics-directed compiler generator," *ACM SIGPLAN Notices* **17:6** (June) 172-184.

GANZINGER, H. AND K. RIPKEN [1980]. "Operator identification in Ada," *ACM SIGPLAN Notices* **15:2** (February) 30-42.

GAREY, M. R. AND D. S. JOHNSON [1979]. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco.

GEAR, C. W. [1965]. "High speed compilation of efficient object code," *Comm. ACM* **8:8**, 483-488.

GESCHKE, C. M. [1972]. *Global Program Optimizations*, Ph. D. Thesis, Dept. of Computer Science, Carnegie-Mellon Univ.

GIEGERICH, R. [1983]. "A formal framework for the derivation of machine-specific optimizers," *TOPLAS* **5:3**, 422-448.

GIEGERICH, R. AND R. WILHELM [1978]. "Counter-one-pass features in one-pass compilation: a formalization using attribute grammars," *Information Processing Letters* **7:6**, 279-284.

GLANVILLE, R. S. [1977]. *A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers*, Ph. D. Thesis, Univ. of California, Berkeley.

GLANVILLE, R. S. AND S. L. GRAHAM [1978]. "A new method for compiler code generation," *Fifth ACM Symposium on Principles of Programming Languages*, 231-240.

GRAHAM, R. M. [1964]. "Bounded context translation," *AFIPS Spring Joint Computer Conference* **40**, 205-217. Reprinted in Rosen [1967], pp. 184-205.

GRAHAM, S. L. [1980]. "Table-driven code generation," *Computer* **13:8**, 25-34.

GRAHAM, S. L. [1984]. "Code generation and optimization," in Lorho [1984], pp. 251-288.

GRAHAM, S. L., C. B. HALEY, AND W. N. JOY [1979]. "Practical LR error recovery," *ACM SIGPLAN Notices* **14:8**, 168-175.

GRAHAM, S. L., M. A. HARRISON, AND W. L. RUZZO [1980]. "An improved context-free recognizer," *TOPLAS* **2:3**, 415-462.

GRAHAM, S. L. AND S. P. RHODES [1975]. "Practical syntactic error recovery," *Comm. ACM* **18:11**, 639-650.

GRAHAM, S. L. AND M. WEGMAN [1976]. "A fast and usually linear algorithm for global data flow analysis," *J. ACM* **23:1**, 172-202.

GRAU, A. A., U. HILL, AND H. LANGMAACK [1967]. *Translation of Algol 60*, Springer-Verlag, New York.

HANSON, D. R. [1981]. "Is block structure necessary?" *Software—Practice and Experience* **11**, 853-866.

HARRISON, M. C. [1971]. "Implementation of the substring test by hashing," *Comm. ACM* **14:12**, 777-779.

HARRISON, W. [1975]. "A class of register allocation algorithms," RC-5342, IBM T. J. Watson Research Center, Yorktown Heights, N. Y.

HARRISON, W. [1977]. "Compiler analysis of the value ranges for variables," *IEEE Trans. Software Engineering* **3:3**.

HECHT, M. S. [1977]. *Flow Analysis of Computer Programs*, North-Holland, New York.

HECHT, M. S. AND J. B. SHAFFER [1975]. "Ideas on the design of a 'quad improver' for SIMPL-T, part I: overview and intersegment analysis," Dept. of Computer Science, Univ. of Maryland, College Park, Md.

HECHT, M. S. AND J. D. ULLMAN [1972]. "Flow graph reducibility," *SIAM J. Computing* **1**, 188-202.

HECHT, M. S. AND J. D. ULLMAN [1974]. "Characterizations of reducible flow graphs," *J. ACM* **21**, 367-375.

HECHT, M. S. AND J. D. ULLMAN [1975]. "A simple algorithm for global data flow analysis programs," *SIAM J. Computing* **4**, 519-532.

HEIJENOORT, J. VAN [1967]. *From Frege to Gödel*, Harvard Univ. Press, Cambridge, Mass.

HENNESSY, J. [1981]. "Program optimization and exception handling," *Eighth Annual ACM Symposium on Principles of Programming Languages*, 200-206.

HENNESSY, J. [1982]. "Symbolic debugging of optimized code," *TOPLAS* **4:3**, 323-344.

HENRY, R. R. [1984]. *Graham-Glanville Code Generators*, Ph. D. Thesis, Univ. of California, Berkeley.

HEXT, J. B. [1967]. "Compile time type-matching," *Computer J.* **9**, 365-369.

HINDLEY, R. [1969]. "The principal type-scheme of an object in combinatory logic," *Trans. AMS* **146**, 29-60.

HOARE, C. A. R. [1962a]. "Quicksort," *Computer J.* **5:1**, 10-15.

HOARE, C. A. R. [1962b]. "Report on the Elliott Algol translator," *Computer J.* **5:2**, 127-129.

HOFFMAN, C. M. AND M. J. O'DONNELL [1982]. "Pattern matching in trees," *J. ACM* **29:1**, 68-95.

HOPCROFT, J. E. AND R. M. KARP [1971]. "An algorithm for testing the equivalence of finite automata," TR-71-114, Dept. of Computer Science, Cornell Univ. See Aho, Hopcroft, and Ullman [1974], pp. 143-145.

HOPCROFT, J. E. AND J. D. ULLMAN [1969]. *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, Mass.

HOPCROFT, J. E. AND J. D. ULLMAN [1973]. "Set merging algorithms," *SIAM J. Computing* **2:3**, 294-303.

HOPCROFT, J. E. AND J. D. ULLMAN [1979]. *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Mass.

HORNING, J. J. [1976]. "What the compiler should tell the user," in Bauer and Eickel [1976].

HORWITZ, L. P., R. M. KARP, R. E. MILLER, AND S. WINOGRAD [1966]. "Index register allocation," *J. ACM* **13:1**, 43-61.

HUET, G. AND G. KAHN (EDS.) [1975]. *Proving and Improving Programs*, Colloque IRIA, Arc-et-Senans, France.

HUET, G. AND J.-J. LEVY [1979]. "Call-by-need computations in nonambiguous linear term rewriting systems," Rapport de Recherche 359, INRIA Laboria, Rocquencourt.

HUFFMAN, D. A. [1954]. "The synthesis of sequential machines," *J. Franklin Inst.* **257**, 3-4, 161, 190, 275-303.

HUNT, J. W. AND M. D. MCILROY [1976]. "An algorithm for differential file comparison," Computing Science Technical Report 41, AT&T Bell Laboratories, Murray Hill, N. J.

HUNT, J. W. AND T. G. SZYMANSKI [1977]. "A fast algorithm for computing longest common subsequences," *Comm. ACM* **20:5**, 350-353.

HUSKEY, H. D., M. H. HALSTEAD, AND R. MCARTHUR [1960]. "Neliac — a dialect of Algol," *Comm. ACM* **3:8**, 463-468.

ICHBIAH, J. D. AND S. P. MORSE [1970]. "A technique for generating almost optimal Floyd-Evans productions for precedence grammars," *Comm. ACM* **13:8**, 501-508.

INGALLS, D. H. H. [1978]. "The Smalltalk-76 programming system design and implementation," *Fifth Annual ACM Symposium on Principles of Programming Languages*, 9-16.

INGERMAN, P. Z. [1967]. "Panini-Backus form suggested," *Comm. ACM* **10:3**, 137.

IRONS, E. T. [1961]. "A syntax directed compiler for Algol 60," *Comm. ACM* **4:1**, 51-55.

IRONS, E. T. [1963]. "An error correcting parse algorithm," *Comm. ACM* **6:11**, 669-673.

IVERSON, K. [1962]. *A Programming Language*, Wiley, New York.

JANAS, J. M. [1980]. "A comment on "Operator identification in Ada" by Ganzinger and Ripken," *ACM SIGPLAN Notices* **15:9** (September) 39-43.

JARVIS, J. F. [1976]. "Feature recognition in line drawings using regular expressions," *Proc. 3rd Intl. Joint Conf. on Pattern Recognition*, 189-192.

JAZAYERI, M., W. F. OGDEN, AND W. C. ROUNDS [1975]. "The intrinsic exponential complexity of the circularity problem for attribute grammars," *Comm. ACM* **18:12**, 697-706.

JAZAYERI, M. AND D. POZEFSKY [1981]. "Space-efficient storage management in an attribute grammar evaluator," *TOPLAS* **3:4**, 388-404.

JAZAYERI, M. AND K. G. WALTER [1975]. "Alternating semantic evaluator," *Proc. ACM Annual Conference*, 230-234.

JENSEN, K. AND N. WIRTH [1975]. *Pascal User Manual and Report*, Springer-Verlag, New York.

JOHNSON, S. C. [1975]. "Yacc – yet another compiler compiler," Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N. J.

JOHNSON, S. C. [1978]. "A portable compiler: theory and practice," *Fifth Annual ACM Symposium on Principles of Programming Languages*, 97-104.

JOHNSON, S. C. [1979]. "A tour through the portable C compiler," AT&T Bell Laboratories, Murray Hill, N. J.

JOHNSON, S. C. [1983]. "Code generation for silicon," *Tenth Annual ACM Symposium on Principles of Programming Languages*, 14-19.

JOHNSON, S. C. AND M. E. LESK [1978]. "Language development tools," *Bell System Technical J.* **57:6**, 2155-2175.

JOHNSON, S. C. AND D. M. RITCHIE [1981]. "The C language calling sequence," Computing Science Technical Report 102, AT&T Bell Laboratories, Murray Hill, N. J.

JOHNSON, W. L., J. H. PORTER, S. I. ACKLEY, AND D. T. ROSS [1968]. "Automatic generation of efficient lexical processors using finite state techniques," *Comm. ACM* **11:12**, 805-813.

JOHNSSON, R. K. [1975]. *An Approach to Global Register Allocation*, Ph. D. Thesis, Carnegie-Mellon Univ., Pittsburgh, Pa.

JOLIAT, M. L. [1976]. "A simple technique for partial elimination of unit productions from LR(k) parser tables," *IEEE Trans. on Computers* **C-25:7**, 763-764.

JONES, N. D. [1980]. *Semantics Directed Compiler Generation*, Lecture Notes in Computer Science **94**, Springer-Verlag, Berlin.

JONES, N. D. AND C. M. MADSEN [1980]. "Attribute-influenced LR parsing," in Jones [1980], pp. 393-407.

JONES, N. D. AND S. S. MUCHNICK [1976]. "Binding time optimization in programming languages," *Third ACM Symposium on Principles of Programming Languages*, 77-94.

JOURDAN, M. [1984]. "Strongly noncircular attribute grammars and their recursive evaluation," *ACM SIGPLAN Notices* **19:6**, 81-93.

KAHN, G., D. B. MACQUEEN, AND G. PLOTKIN [1984]. *Semantics of Data Types*, Lecture Notes in Computer Science **173**, Springer-Verlag, Berlin.

KAM, J. B. AND J. D. ULLMAN [1976]. "Global data flow analysis and iterative algorithms," *J. ACM* **23:1**, 158-171.

KAM, J. B. AND J. D. ULLMAN [1977]. "Monotone data flow analysis frameworks," *Acta Informatica* **7:3**, 305-318.

KAPLAN, M. AND J. D. ULLMAN [1980]. "A general scheme for the automatic inference of variable types," *J. ACM* **27:1**, 128-145.

KASAMI, T. [1965]. "An efficient recognition and syntax analysis algorithm for context-free languages," AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Mass.

KASAMI, T., W. W. PETERSON, AND N. TOKURA [1973]. "On the capabilities of while, repeat, and exit statements," *Comm. ACM* **16:8**, 503-512.

KASTENS, U. [1980]. "Ordered attribute grammars," *Acta Informatica* **13:3**, 229-256.

KASTENS, U., B. HUTT, AND E. ZIMMERMANN [1982]. *GAG: A Practical Compiler Generator*, Lecture Notes in Computer Science **141**, Springer-Verlag, Berlin.

KASYANOV, V. N. [1973]. "Some properties of fully reducible graphs," *Information Processing Letters* **2:4**, 113-117.

KATAYAMA, T. [1984]. "Translation of attribute grammars into procedures," *TOPLAS* **6:3**, 345-369.

KENNEDY, K. [1971]. "A global flow analysis algorithm," *Intern. J. Computer Math. Section A* **3**, 5-15.

KENNEDY, K. [1972]. "Index register allocation in straight line code and simple loops," in Rustin [1972], pp. 51-64.

KENNEDY, K. [1976]. "A comparison of two algorithms for global flow analysis," *SIAM J. Computing* **5:1**, 158-180.

KENNEDY, K. [1981]. "A survey of data flow analysis techniques," in Muchnick and Jones [1981], pp. 5-54.

KENNEDY, K. AND J. RAMANATHAN [1979]. "A deterministic attribute grammar evaluator based on dynamic sequencing," *TOPLAS* **1:1**, 142-160.

KENNEDY, K. AND S. K. WARREN [1976]. "Automatic generation of efficient evaluators for attribute grammars," *Third ACM Symposium on Principles of Programming Languages*, 32-49.

KERNIGHAN, B. W. [1975]. "Ratfor – a preprocessor for a rational Fortran," *Software—Practice and Experience* **5:4**, 395-406.

KERNIGHAN, B. W. [1982]. "PIC – a language for typesetting graphics," *Software—Practice and Experience* **12:1**, 1-21.

KERNIGHAN, B. W. AND L. L. CHERRY [1975]. "A system for typesetting mathematics," *Comm. ACM* **18:3**, 151-157.

KERNIGHAN, B. W. AND R. PIKE [1984]. *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, N. J.

KERNIGHAN, B. W. AND D. M. RITCHIE [1978]. *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N. J.

KILDALL, G. [1973]. "A unified approach to global program optimization," *ACM Symposium on Principles of Programming Languages*, 194-206.

KLEENE, S. C. [1956]. "Representation of events in nerve nets," in Shannon and McCarthy [1956], pp. 3-40.

KNUTH, D. E. [1962]. "A history of writing compilers," *Computers and Automation* (December) 8-18. Reprinted in Pollack [1972], pp. 38-56.

KNUTH, D. E. [1964]. "Backus Normal Form vs. Backus Naur Form," *Comm. ACM* **7:12**, 735-736.

KNUTH, D. E. [1965]. "On the translation of languages from left to right,"

Information and Control **8:6**, 607-639.

KNUTH, D. E. [1968]. "Semantics of context-free languages," *Mathematical Systems Theory* **2:2**, 127-145. Errata **5:1** (1971) 95-96.

KNUTH, D. E. [1971a]. "Top-down syntax analysis," *Acta Informatica* **1:2**, 79-110.

KNUTH, D. E. [1971b]. "An empirical study of FORTRAN programs," *Software—Practice and Experience* **1:2**, 105-133.

KNUTH, D. E. [1973a]. *The Art of Computer Programming:* Vol. 1, 2nd. Ed., *Fundamental Algorithms*, Addison-Wesley, Reading, Mass.

KNUTH, D. E. [1973b]. *The Art of Computer Programming:* Vol. 3, *Sorting and Searching*, Addison-Wesley, Reading, Mass.

KNUTH, D. E. [1977]. "A generalization of Dijkstra's algorithm," *Information Processing Letters* **6**, 1-5.

KNUTH, D. E. [1984a]. *The TEXbook*, Addison-Wesley, Reading, Mass.

KNUTH, D. E. [1984b]. "Literate programming," *Computer J.* **28:2**, 97-111.

KNUTH, D. E. [1985,1986]. *Computers and Typesetting*, Vol. 1: *TEX*, Addison-Wesley, Reading, Mass. A preliminary version has been published under the title, "*TEX: The Program.*"

KNUTH, D. E., J. H. MORRIS, AND V. R. PRATT [1977]. "Fast pattern matching in strings," *SIAM J. Computing* **6:2**, 323-350.

KNUTH, D. E. AND L. TRABB PARDO [1977]. "Early development of programming languages," *Encyclopedia of Computer Science and Technology* **7**, Marcel Dekker, New York, 419-493.

KORENJAK, A. J. [1969]. "A practical method for constructing LR(k) processors," *Comm. ACM* **12:11**, 613-623.

KOSARAJU, S. R. [1974]. "Analysis of structured programs," *J. Computer and System Sciences* **9:3**, 232-255.

KOSKIMIES, K. AND K.-J. RAIHA [1983]. "Modelling of space-efficient one-pass translation using attribute grammars," *Software—Practice and Experience* **13**, 119-129.

KOSTER, C. H. A. [1971]. "Affix grammars," in Peck [1971], pp. 95-109.

KOU, L. [1977]. "On live-dead analysis for global data flow problems," *J. ACM* **24:3**, 473-483.

KRISTENSEN, B. B. AND O. L. MADSEN [1981]. "Methods for computing LALR(k) lookahead," *TOPLAS* **3:1**, 60-82.

KRON, H. [1975]. *Tree Templates and Subtree Transformational Grammars*, Ph. D. Thesis, Univ. of California, Santa Cruz.

LALONDE, W. R. [1971]. "An efficient LALR parser generator," Tech. Rep. 2, Computer Systems Research Group, Univ. of Toronto.

LALONDE, W. R. [1976]. "On directly constructing LR(k) parsers without chain reductions," *Third ACM Symposium on Principles of Programming Languages*, 127-133.

LALONDE, W. R., E. S. LEE, AND J. J. HORNING [1971]. "An LALR(k) parser generator," *Proc. IFIP Congress 71* **TA-3**, North-Holland, Amsterdam, 153-157.

LAMB, D. A. [1981]. "Construction of a peephole optimizer," *Software—*

Practice and Experience **11**, 638-647.

LAMPSON, B. W. [1982]. "Fast procedure calls," *ACM SIGPLAN Notices* **17:4** (April) 66-76.

LANDIN, P. J. [1964]. "The mechanical evaluation of expressions," *Computer J.* **6:4**, 308-320.

LECARME, O. AND M.-C. PEYROLLE-THOMAS [1978]. "Self-compiling compilers: an appraisal of their implementation and portability," *Software—Practice and Experience* **8**, 149-170.

LEDGARD, H. F. [1971]. "Ten mini-languages: a study of topical issues in programming languages," *Computing Surveys* **3:3**, 115-146.

LEINIUS, R. P. [1970]. *Error Detection and Recovery for Syntax Directed Compiler Systems*, Ph. D. Thesis, University of Wisconsin, Madison.

LENGAUER, T. AND R. E. TARJAN [1979]. "A fast algorithm for finding dominators in a flowgraph," *TOPLAS* **1**, 121-141.

LESK, M. E. [1975]. "Lex – a lexical analyzer generator," Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N. J.

LEVERETT, B. W. [1982]. "Topics in code generation and register allocation," CMU CS-82-130, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, Pennsylvania.

LEVERETT, B. W., R. G. G. CATTELL, S. O. HOBBS, J. M. NEWCOMER, A. H. REINER, B. R. SCHATZ, AND W. A. WULF [1980]. "An overview of the production-quality compiler-compiler project," *Computer* **13:8**, 38-40.

LEVERETT, B. W. AND T. G. SZYMANSKI [1980]. "Chaining span-dependent jump instructions," *TOPLAS* **2:3**, 274-289.

LEVY, J. P. [1975]. "Automatic correction of syntax errors in programming languages," *Acta Informatica* **4**, 271-292.

LEWIS, P. M., II, D. J. ROSENKRANTZ, AND R. E. STEARNS [1974]. "Attributed translations," *J. Computer and System Sciences* **9:3**, 279-307.

LEWIS, P. M., II, D. J. ROSENKRANTZ, AND R. E. STEARNS [1976]. *Compiler Design Theory*, Addison-Wesley, Reading, Mass.

LEWIS, P. M., II AND R. E. STEARNS [1968]. "Syntax-directed transduction," *J. ACM* **15:3**, 465-488.

LORHO, B. [1977]. "Semantic attribute processing in the system Delta," in Ershov and Koster [1977], pp. 21-40.

LORHO, B. [1984]. *Methods and Tools for Compiler Construction*, Cambridge Univ. Press.

LORHO, B. AND C. PAIR [1975]. "Algorithms for checking consistency of attribute grammars," in Huet and Kahn [1975], pp. 29-54.

LOW, J. AND P. ROVNER [1976]. "Techniques for the automatic selection of data structures," *Third ACM Symposium on Principles of Programming Languages*, 58-67.

LOWRY, E. S. AND C. W. MEDLOCK [1969]. "Object code optimization," *Comm. ACM* **12**, 13-22.

LUCAS, P. [1961]. "The structure of formula translators," *Elektronische Rechenanlagen* **3**, 159-166.

LUNDE, A. [1977]. "Empirical evaluation of some features of instruction set processor architectures," *Comm. ACM* **20:3**, 143-153.

LUNELL, H. [1983]. *Code Generator Writing Systems*, Ph. D. Thesis, Linköping University, Linköping, Sweden.

MACQUEEN, D. B., G. P. PLOTKIN, AND R. SETHI [1984]. "An ideal model of recursive polymorphic types," *Eleventh Annual ACM Symposium on Principles of Programming Languages*, 165-174.

MADSEN, O. L. [1980]. "On defining semantics by means of extended attribute grammars," in Jones [1980], pp. 259-299.

MARILL, T. [1962]. "Computational chains and the simplification of computer programs," *IRE Trans. Electronic Computers* **EC-11:2**, 173-180.

MARTELLI, A. AND U. MONTANARI [1982]. "An efficient unification algorithm," *TOPLAS* **4:2**, 258-282.

MAUNEY, J. AND C. N. FISCHER [1982]. "A forward move algorithm for LL and LR parsers," *ACM SIGPLAN Notices* **17:4**, 79-87.

MAYOH, B. H. [1981]. "Attribute grammars and mathematical semantics," *SIAM J. Computing* **10:3**, 503-518.

MCCARTHY, J. [1963]. "Towards a mathematical science of computation," *Information Processing 1962*, North-Holland, Amsterdam, 21-28.

MCCARTHY, J. [1981]. "History of Lisp," in Wexelblat [1981], pp. 173-185.

MCCLURE, R. M. [1965]. "TMG – a syntax-directed compiler," *Proc. 20th ACM National Conf.*, 262-274.

MCCRACKEN, N. J. [1979]. *An Investigation of a Programming Language with a Polymorphic Type Structure*, Ph. D. Thesis, Syracuse University, Syracuse, N. Y.

MCCULLOUGH, W. S. AND W. PITTS [1943]. "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Math. Biophysics* **5**, 115-133.

MCKEEMAN, W. M. [1965]. "Peephole optimization," *Comm. ACM* **8:7**, 443-444.

MCKEEMAN, W. M. [1976]. "Symbol table access," in Bauer and Eickel [1976], pp. 253-301.

MCKEEMAN, W. M., J. J. HORNING, AND D. B. WORTMAN [1970]. *A Compiler Generator*, Prentice-Hall, Englewood Cliffs, N. J.

MCNAUGHTON, R. AND H. YAMADA [1960]. "Regular expressions and state graphs for automata," *IRE Trans. on Electronic Computers* **EC-9:1**, 38-47.

MEERTENS, L. [1983]. "Incremental polymorphic type checking in B," *Tenth ACM Symposium on Principles of Programming Languages*, 265-275.

METCALF, M. [1982]. *Fortran Optimization*, Academic Press, New York.

MILLER, R. E. AND J. W. THATCHER (EDS.) [1972]. *Complexity of Computer Computations*, Academic Press, New York.

MILNER, R. [1978]. "A theory of type polymorphism in programming," *J. Computer and System Sciences* **17:3**, 348-375.

MILNER, R. [1984]. "A proposal for standard ML," *ACM Symposium on Lisp and Functional Programming*, 184-197.

MINKER, J. AND R. G. MINKER [1980]. "Optimization of boolean expressions – historical developments," *A. of the History of Computing* **2:3**, 227-238.

MITCHELL, J. C. [1984]. "Coercion and type inference," *Eleventh ACM Sympo-*

sium on Principles of Programming Languages, 175-185.

MOORE, E. F. [1956]. "Gedanken experiments in sequential machines," in Shannon and McCarthy [1956], pp. 129-153.

MOREL, E. AND C. RENVOISE [1979]. "Global optimization by suppression of partial redundancies," *Comm. ACM* **22**, 96-103.

MORRIS, J. H. [1968a]. *Lambda-Calculus Models of Programming Languages*, Ph. D. Thesis, MIT, Cambridge, Mass.

MORRIS, R. [1968b]. "Scatter storage techniques," *Comm. ACM* **11:1**, 38-43.

MOSES, J. [1970]. "The function of FUNCTION in Lisp," *SIGSAM Bulletin* **15** (July) 13-27.

MOULTON, P. G. AND M. E. MULLER [1967]. "DITRAN – a compiler emphasizing diagnostics," *Comm. ACM* **10:1**, 52-54.

MUCHNICK, S. S. AND N. D. JONES [1981]. *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Englewood Cliffs, N. J.

NAKATA, I. [1967]. "On compiling algorithms for arithmetic expressions," *Comm. ACM* **10:8**, 492-494.

NAUR, P. (ED.) [1963]. "Revised report on the algorithmic language Algol 60," *Comm. ACM* **6:1**, 1-17.

NAUR, P. [1965]. "Checking of operand types in Algol compilers," *BIT* **5**, 151-163.

NAUR, P. [1981]. "The European side of the last phase of the development of Algol 60," in Wexelblat [1981], pp. 92-139, 147-161.

NEWEY, M. C., P. C. POOLE, AND W. M. WAITE [1972]. "Abstract machine modelling to produce portable software – a review and evaluation," *Software—Practice and Experience* **2:2**, 107-136.

NEWEY, M. C. AND W. M. WAITE [1985]. "The robust implementation of sequence-controlled iteration," *Software—Practice and Experience* **15:7**, 655-668.

NICHOLLS, J. E. [1975]. *The Structure and Design of Programming Languages*, Addison-Wesley, Reading, Mass.

NIEVERGELT, J. [1965]. "On the automatic simplification of computer code," *Comm. ACM* **8:6**, 366-370.

NORI, K. V., U. AMMANN, K. JENSEN, H. H. NAGELI, AND CH. JACOBI [1981]. "Pascal P implementation notes," in Barron [1981], pp. 125-170.

OSTERWEIL, L. J. [1981]. "Using data flow tools in software engineering," in Muchnick and Jones [1981], pp. 237-263.

PAGER, D. [1977a]. "A practical general method for constructing LR(k) parsers," *Acta Informatica* **7**, 249-268.

PAGER, D. [1977b]. "Eliminating unit productions from LR(k) parsers," *Acta Informatica* **9**, 31-59.

PAI, A. B. AND R. B. KIEBURTZ [1980]. "Global context recovery: a new strategy for syntactic error recovery by table-driven parsers," *TOPLAS* **2:1**, 18-41.

PAIGE, R. AND J. T. SCHWARTZ [1977]. "Expression continuity and the formal differentiation of algorithms," *Fourth ACM Symposium on Principles of Programming Languages*, 58-71.

PALM, R. C., JR. [1975]. "A portable optimizer for the language C," M. Sc. Thesis, MIT, Cambridge, Mass.

PARK, J. C. H., K. M. CHOE, AND C. H. CHANG [1985]. "A new analysis of LALR formalisms," *TOPLAS* **7:1**, 159-175.

PATERSON, M. S. AND M. WEGMAN [1978]. "Linear unification," *J. Computer and System Sciences* **16:2**, 158-167.

PECK, J. E. L. [1971]. *Algol 68 Implementation*, North-Holland, Amsterdam.

PENNELLO, T. AND F. DEREMER [1978]. "A forward move algorithm for LR error recovery," *Fifth Annual ACM Symposium on Principles of Programming Languages*, 241-254.

PENNELLO, T., F. DEREMER, AND R. MEYERS [1980]. "A simplified operator identification scheme for Ada," *ACM SIGPLAN Notices* **15:7** (July-August) 82-87.

PERSCH, G., G. WINTERSTEIN, M. DAUSSMANN, AND S. DROSSOPOULOU [1980]. "Overloading in preliminary Ada," *ACM SIGPLAN Notices* **15:11** (November) 47-56.

PETERSON, W. W. [1957]. "Addressing for random access storage," *IBM J. Research and Development* **1:2**, 130-146.

POLLACK, B. W. [1972]. *Compiler Techniques*, Auerbach Publishers, Princeton, N. J.

POLLOCK, L. L. AND M. L. SOFFA [1985]. "Incremental compilation of locally optimized code," *Twelfth Annual ACM Symposium on Principles of Programming Languages*, 152-164.

POWELL, M. L. [1984]. "A portable optimizing compiler for Modula-2," *ACM SIGPLAN Notices* **19:6**, 310-318.

PRATT, T. W. [1984]. *Programming Languages: Design and Implementation*, 2nd Ed., Prentice-Hall, Englewood Cliffs, N. J.

PRATT, V. R. [1973]. "Top down operator precedence," *ACM Symposium on Principles of Programming Languages*, 41-51.

PRICE, C. E. [1971]. "Table lookup techniques," *Computing Surveys* **3:2**, 49-65.

PROSSER, R. T. [1959]. "Applications of boolean matrices to the analysis of flow diagrams," *AFIPS Eastern Joint Computer Conf.*, Spartan Books, Baltimore, Md., 133-138.

PURDOM, P. AND C. A. BROWN [1980]. "Semantic routines and LR(k) parsers," *Acta Informatica* **14:4**, 299-315.

PURDOM, P. W. AND E. F. MOORE [1972]. "Immediate predominators in a directed graph," *Comm. ACM* **15:8**, 777-778.

RABIN, M. O. AND D. SCOTT. [1959]. "Finite automata and their decision problems," *IBM J. Research and Development* **3:2**, 114-125.

RADIN, G. AND H. P. ROGOWAY [1965]. "NPL: Highlights of a new programming language," *Comm. ACM* **8:1**, 9-17.

RAIHA, K.-J. [1981]. *A Space Management Technique for Multi-Pass Attribute Evaluators*, Ph. D. Thesis, Report A-1981-4, Dept. of Computer Science, University of Helsinki.

RAIHA, K.-J. AND M. SAARINEN [1982]. "Testing attribute grammars for circularity," *Acta Informatica* **17**, 185-192.

RAIHA, K.-J., M. SAARINEN, M. SARJAKOSKI, S. SIPPU, E. SOISALON-SOININEN, AND M. TIENARI [1983]. "Revised report on the compiler writing system HLP78," Report A-1983-1, Dept. of Computer Science, University of Helsinki.

RANDELL, B. AND L. J. RUSSELL [1964]. *Algol 60 Implementation*, Academic Press, New York.

REDZIEJOWSKI, R. R. [1969]. "On arithmetic expressions and trees," *Comm. ACM* **12:2**, 81-84.

REIF, J. H. AND H. R. LEWIS [1977]. "Symbolic evaluation and the global value graph," *Fourth ACM Symposium on Principles of Programming Languages*, 104-118.

REISS, S. P. [1983]. "Generation of compiler symbol processing mechanisms from specifications," *TOPLAS* **5:2**, 127-163.

REPS, T. W. [1984]. *Generating Language-Based Environments*, MIT Press, Cambridge, Mass.

REYNOLDS, J. C. [1985]. "Three approaches to type structure," *Mathematical Foundations of Software Development*, Lecture Notes in Computer Science **185**, Springer-Verlag, Berlin, 97-138.

RICHARDS, M. [1971]. "The portability of the BCPL compiler," *Software—Practice and Experience* **1:2**, 135-146.

RICHARDS, M. [1977]. "The implementation of the BCPL compiler," in P. J. Brown (ed.), *Software Portability: An Advanced Course*, Cambridge University Press.

RIPKEN, K. [1977]. "Formale beschreibun von maschinen, implementierungen und optimierender maschinen-codeerzeugung aus attributierten programmgraphe," TUM-INFO-7731, Institut für Informatik, Universität München, Munich.

RIPLEY, G. D. AND F. C. DRUSEIKIS [1978]. "A statistical analysis of syntax errors," *Computer Languages* **3**, 227-240.

RITCHIE, D. M. [1979]. "A tour through the UNIX C compiler," AT&T Bell Laboratories, Murray Hill, N. J.

RITCHIE, D. M. AND K. THOMPSON [1974]. "The UNIX time-sharing system," *Comm. ACM* **17:7**, 365-375.

ROBERTSON, E. L. [1979]. "Code generation and storage allocation for machines with span-dependent instructions," *TOPLAS* **1:1**, 71-83.

ROBINSON, J. A. [1965]. "A machine-oriented logic based on the resolution principle," *J. ACM* **12:1**, 23-41.

ROHL, J. S. [1975]. *An Introduction to Compiler Writing*, American Elsevier, New York.

ROHRICH, J. [1980]. "Methods for the automatic construction of error correcting parsers," *Acta Informatica* **13:2**, 115-139.

ROSEN, B. K. [1977]. "High-level data flow analysis," *Comm. ACM* **20**, 712-724.

ROSEN, B. K. [1980]. "Monoids for rapid data flow analysis," *SIAM J. Computing* **9:1**, 159-196.

ROSEN, S. [1967]. *Programming Systems and Languages*, McGraw-Hill, New York.

ROSENKRANTZ, D. J. AND R. E. STEARNS [1970]. "Properties of deterministic top-down grammars," *Information and Control* **17:3**, 226-256.

ROSLER, L. [1984]. "The evolution of C — past and future," *AT&T Bell Labs Technical Journal* **63:8**, 1685-1699.

RUSTIN, R. [1972]. *Design and Optimization of Compilers*, Prentice-Hall, Englewood Cliffs, N.J.

RYDER, B. G. [1979]. "Constructing the call graph of a program," *IEEE Trans. Software Engineering* **SE-5:3**, 216-226.

RYDER, B. G. [1983]. "Incremental data flow analysis," *Tenth ACM Symposium on Principles of Programming Languages*, 167-176.

SAARINEN, M. [1978]. "On constructing efficient evaluators for attribute grammars," *Automata, Languages and Programming, Fifth Colloquium*, Lecture Notes in Computer Science **62**, Springer-Verlag, Berlin, 382-397.

SAMELSON, K. AND F. L. BAUER [1960]. "Sequential formula translation," *Comm. ACM* **3:2**, 76-83.

SANKOFF, D. AND J. B. KRUSKAL (EDS.) [1983]. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, Mass.

SCARBOROUGH, R. G. AND H. G. KOLSKY [1980]. "Improved optimization of Fortran object programs," *IBM J. Research and Development* **24:6**, 660-676.

SCHAEFER, M. [1973]. *A Mathematical Theory of Global Program Optimization*, Prentice Hall, Englewood Cliffs, N. J.

SCHONBERG, E., J. T. SCHWARTZ, AND M. SHARIR [1981]. "An automatic technique for selection of data representations in SETL Programs," *TOPLAS* **3:2**, 126-143.

SCHORRE, D. V. [1964]. "Meta-II: a syntax-oriented compiler writing language," *Proc. 19th ACM National Conf.*, D1.3-1 - D1.3-11.

SCHWARTZ, J. T. [1973]. *On Programming: An Interim Report on the SETL Project*, Courant Inst., New York.

SCHWARTZ, J. T. [1975a]. "Automatic data structure choice in a language of very high level," *Comm. ACM* **18:12**, 722-728.

SCHWARTZ, J. T. [1975b]. "Optimization of very high level languages," *Computer Languages*. Part I: "Value transmission and its corollaries," **1:2**, 161-194; part II: "Deducing relationships of inclusion and membership," **1:3**, 197-218.

SEDGEWICK, R. [1978]. "Implementing Quicksort programs," *Comm. ACM* **21**, 847-857.

SETHI, R. [1975]. "Complete register allocation problems," *SIAM J. Computing* **4:3**, 226-248.

SETHI, R. AND J. D. ULLMAN [1970]. "The generation of optimal code for arithmetic expressions," *J. ACM* **17:4**, 715-728.

SHANNON, C. AND J. MCCARTHY [1956]. *Automata Studies*, Princeton University Press.

SHERIDAN, P. B. [1959]. "The arithmetic translator-compiler of the IBM Fortran automatic coding system," *Comm. ACM* **2:2**, 9-21.

SHIMASAKI, M., S. FUKAYA, K. IKEDA, AND T. KIYONO [1980]. "An analysis of Pascal programs in compiler writing," *Software—Practice and Experience* **10:2**, 149-157.

SHUSTEK, L. J. [1978]. "Analysis and performance of computer instruction sets," SLAC Report 205, Stanford Linear Accelerator Center, Stanford University, Stanford, California.

SIPPU, S. [1981]. "Syntax error handling in compilers," Rep. A-1981-1, Dept. of Computer Science, Univ. of Helsinki, Helsinki, Finland.

SIPPU, S. AND E. SOISALON-SOININEN [1983]. "A syntax-error-handling technique and its experimental analysis," *TOPLAS* **5:4**, 656-679.

SOISALON-SOININEN, E. [1980]. "On the space optimizing effect of eliminating single productions from LR parsers," *Acta Informatica* **14**, 157-174.

SOISALON-SOININEN, E. AND E. UKKONEN [1979]. "A method for transforming grammars into LL(k) form," *Acta Informatica* **12**, 339-369.

SPILLMAN, T. C. [1971]. "Exposing side effects in a PL/I optimizing compiler," *Information Processing 71*, North-Holland, Amsterdam, 376-381.

STEARNS, R. E. [1971]. "Deterministic top-down parsing," *Proc. 5th Annual Princeton Conf. on Information Sciences and Systems*, 182-188.

STEEL, T. B., JR. [1961]. "A first version of Uncol," *Western Joint Computer Conference*, 371-378.

STEELE, G. L., JR. [1984]. *Common LISP*, Digital Press, Burlington, Mass.

STOCKHAUSEN, P. F. [1973]. "Adapting optimal code generation for arithmetic expressions to the instruction sets available on present-day computers," *Comm. ACM* **16:6**, 353-354. Errata: **17:10** (1974) 591.

STONEBRAKER, M., E. WONG, P. KREPS, AND G. HELD [1976]. "The design and implementation of INGRES," *ACM Trans. Database Systems* **1:3**, 189-222.

STRONG, J., J. WEGSTEIN, A. TRITTER, J. OLSZTYN, O. MOCK, AND T. STEEL [1958]. "The problem of programming communication with changing machines: a proposed solution," *Comm. ACM* **1:8** (August) 12-18. Part 2: **1:9** (September) 9-15. Report of the Share Ad-Hoc committee on Universal Languages.

STROUSTRUP, B. [1986]. *The C++ Programming Language*, Addison-Wesley, Reading, Mass.

SUZUKI, N. [1981]. "Inferring types in Smalltalk," *Eighth ACM Symposium on Principles of Programming Languages*, 187-199.

SUZUKI, N. AND K. ISHIHATA [1977]. "Implementation of array bound checker," *Fourth ACM Symposium on Principles of Programming Languages*, 132-143.

SZYMANSKI, T. G. [1978]. "Assembling code for machines with span-dependent instructions," *Comm. ACM* **21:4**, 300-308.

TAI, K. C. [1978]. "Syntactic error correction in programming languages," *IEEE Trans. Software Engineering* **SE-4:5**, 414-425.

TANENBAUM, A. S., H. VAN STAVEREN, E. G. KEIZER, AND J. W. STEVENSON [1983]. "A practical tool kit for making portable compilers," *Comm. ACM* **26:9**, 654-660.

TANENBAUM, A. S., H. VAN STAVEREN, AND J. W. STEVENSON [1982]. "Using peephole optimization on intermediate code," *TOPLAS* **4:1**, 21-36.

TANTZEN, R. G. [1963]. "Algorithm 199: Conversions between calendar date and Julian day number," *Comm. ACM* **6:8**, 443.

TARHIO, J. [1982]. "Attribute evaluation during LR parsing," Report A-1982-4, Dept. of Computer Science, University of Helsinki.

TARJAN, R. E. [1974a]. "Finding dominators in directed graphs," *SIAM J. Computing* **3:1**, 62-89.

TARJAN, R. E. [1974b]. "Testing flow graph reducibility," *J. Computer and System Sciences* **9:3**, 355-365.

TARJAN, R. E. [1975]. "Efficiency of a good but not linear set union algorithm," *JACM* **22:2**, 215-225.

TARJAN, R. E. [1981]. "A unified approach to path problems," *J. ACM* **28:3**, 577-593. And "Fast algorithms for solving path problems," *J. ACM* **28:3** 594-614.

TARJAN, R. E. AND A. C. YAO [1979]. "Storing a sparse table," *Comm. ACM* **22:11**, 606-611.

TENNENBAUM, A. M. [1974]. "Type determination in very high level languages," NSO-3, Courant Institute of Math. Sciences, New York Univ.

TENNENT, R. D. [1981]. *Principles of Programming Languages*, Prentice-Hall International, Englewood Cliffs, N. J.

THOMPSON, K. [1968]. "Regular expression search algorithm," *Comm. ACM* **11:6**, 419-422.

TJIANG, S. W. K. [1986]. "Twig language manual," Computing Science Technical Report 120, AT&T Bell Laboratories, Murray Hill, N. J.

TOKUDA, T. [1981]. "Eliminating unit reductions from LR(k) parsers using minimum contexts," *Acta Informatica* **15**, 447-470.

TRICKEY, H. W. [1985]. *Compiling Pascal Programs into Silicon*, Ph. D. Thesis, Stanford Univ.

ULLMAN, J. D. [1973]. "Fast algorithms for the elimination of common subexpressions," *Acta Informatica* **2**, 191-213.

ULLMAN, J. D. [1982]. *Principles of Database Systems*, 2nd Ed., Computer Science Press, Rockville, Md.

ULLMAN, J. D. [1984]. *Computational Aspects of VLSI*, Computer Science Press, Rockville, Md.

VYSSOTSKY, V. AND P. WEGNER [1963]. "A graph theoretical Fortran source language analyzer," manuscript, AT&T Bell Laboratories, Murray Hill, N. J.

WAGNER, R. A. [1974]. "Order-n correction for regular languages," *Comm. ACM* **16:5**, 265-268.

WAGNER, R. A. AND M. J. FISCHER [1974]. "The string-to-string correction problem," *J. ACM* **21:1**, 168-174.

WAITE, W. M. [1976a]. "Code generation," in Bauer and Eickel [1976], 302-332.

WAITE, W. M. [1976b]. "Optimization," in Bauer and Eickel [1976], 549-602.

WAITE, W. M. AND L. R. CARTER [1985]. "The cost of a generated parser," *Software—Practice and Experience* **15:3**, 221-237.

WASILEW, S. G. [1971]. *A Compiler Writing System with Optimization Capabilities for Complex Order Structures*, Ph. D. Thesis, Northwestern Univ., Evanston, Ill.

WATT, D. A. [1977]. "The parsing problem for affix grammars," *Acta Informatica* **8**, 1-20.

WEGBREIT, B. [1974]. "The treatment of data types in EL1," *Comm. ACM* **17:5**, 251-264.

WEGBREIT, B. [1975]. "Property extraction in well-founded property sets," *IEEE Trans. on Software Engineering* **1:3**, 270-285.

WEGMAN, M. N. [1983]. "Summarizing graphs by regular expressions," *Tenth Annual ACM Symposium on Principles of Programming Languages*, 203-216.

WEGMAN, M. N. AND F. K. ZADECK [1985]. "Constant propagation with conditional branches," *Twelfth Annual ACM Symposium on Principles of Programming Languages*, 291-299.

WEGSTEIN, J. H. [1981]. "Notes on Algol 60," in Wexelblat [1981], pp. 126-127.

WEIHL, W. E. [1980]. "Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables," *Seventh Annual ACM Symposium on Principles of Programming Languages*, 83-94.

WEINGART, S. W. [1973]. *An Efficient and Systematic Method of Code Generation*, Ph. D. Thesis, Yale University, New Haven, Connecticut.

WELSH, J., W. J. SNEERINGER, AND C. A. R. HOARE [1977]. "Ambiguities and insecurities in Pascal," *Software—Practice and Experience* **7:6**, 685-696.

WEXELBLAT, R. L. [1981]. *History of Programming Languages*, Academic Press, New York.

WIRTH, N. [1968]. "PL 360 – a programming language for the 360 computers," *J. ACM* **15:1**, 37-74.

WIRTH, N. [1971]. "The design of a Pascal compiler," *Software—Practice and Experience* **1:4**, 309-333.

WIRTH, N. [1981]. "Pascal-S: A subset and its implementation," in Barron [1981], pp. 199-259.

WIRTH, N. AND H. WEBER [1966]. "Euler: a generalization of Algol and its formal definition: Part I," *Comm. ACM* **9:1**, 13-23.

WOOD, D. [1969]. "The theory of left factored languages," *Computer J.* **12:4**, 349-356.

WULF, W. A., R. K. JOHNSSON, C. B. WEINSTOCK, S. O. HOBBS, AND C. M. GESCHKE [1975]. *The Design of an Optimizing Compiler*, American Elsevier, New York.

YANNAKAKIS, M. [1985]. private communication.

YOUNGER, D. H. [1967]. "Recognition and parsing of context-free languages in time n^3," *Information and Control* **10:2**, 189-208.

ZELKOWITZ, M. V. AND W. G. BAIL [1974]. "Optimization of structured programs," *Software—Practice and Experience* **4:1**, 51-57.

索　引

索引中的页码为英文原书页码，与书中页边标注的页码一致。

A

Abel, N. E., 718
Abelson, H., 462
Absolute machine code（绝对机器代码）, 5, 19, 54~515
Abstract machine（抽象机）, 62~63。另见Stack machine
Abstract syntax tree（抽象语法树）, 49。另见Syntax tree
Acceptance（接受）, 115~116, 199
Accepting state（接受状态）, 114
Access link（访问链）, 398, 416~420, 423
Ackley, S. I., 157
Action table（动作表）, 216
Activation（活动）, 389
Activation environment（活动环境）, 457~458
Activation record（活动记录）, 398~410, 522~527
Activation tree（活动树）, 391~393
Actual parameter（实参）, 390, 399
Acyclic graph（无环图）, 另见Directed acyclic graph
Ada, 343, 361, 363~364, 366~367, 411
Address descriptor（地址描述符）, 537
Address mode（地址模式）, 18~19, 519~521, 579~580
Adjacency list（邻接表）, 114~115
Adrion, W. R., 722
Advancing edge（前进边）, 663
Affix grammar（词缀文法）, 341
Aho, A. V., 158, 181, 204, 277~278, 292, 392, 444~445, 462, 566, 572, 583~584, 587, 721
Aigrain, P., 584
Algebraic transformation（代数变换）, 532, 557, 566, 600~602, 739
Algol 24, 80, 82, 157, 277, 428, 461, 512, 561
Algol 68, 86
Algol-68, 21, 386, 512
Alias（别名）, 648~660, 721
Alignment, of data（数据对齐）, 399~400, 473
Allen, F. E., 718~721
Alphabet（字母表）, 92
Alternative（候选式）, 167
Ambiguity（二义性）, 30, 171, 174~175, 184, 191~192, 201~202, 229~230, 247~254, 261~264, 578
Ambiguous definition（含糊定义）, 610
Ammann, U., 82, 511, 583, 728~729, 734~735
Analysis（分析）, 2~10。另见Lexical analysis, Parsing
Anderson, J. P., 583
Anderson, T., 278
Anklam, P., 719
Annotated parse tree（注释分析树）, 34, 280
APL, 3, 387, 411, 694
Arden, B. W., 462
Arithmetic operator（算术运算符）, 361
Array（数组）, 345, 349, 427
Array reference（数组引用）, 202, 467, 481~485, 552~553, 582, 588, 649
ASCII, 59
Assembly code（汇编代码）, 4~5, 15, 17~19, 89, 515, 519
Assignment statement（赋值语句）, 65, 467, 478~488
Associativity（结合规则）, 30, 32, 95~96, 207, 247~249, 263~264, 684
Attribute（属性）, 11, 33, 87, 260, 280。另见Inherited attribute, Lexical Value, Syntax-directed definition, Synthesized attribute
Attribute grammar（属性文法）, 280, 580
Augmented dependency graph（扩充依赖图）, 334
Augmented grammar（拓广文法）, 222
Auslander, M. A., 546, 583, 719
Automatic code generator（自动代码生成器）, 23
Available expression（可用表达式）, 627~631, 659~660, 684, 694
AWK 83, 158, 724

B

Back edge（回边）, 604, 606, 664
Back end（后端）, 20, 62
Backhouse, R.C., 278, 722
Backpatching（回填）, 21, 500~506, 515
Backtracking（回溯）, 181~182
Backus, J. W., 2, 82, 157, 386
Backus-Naur form（Backus-Naur范式）, 见BNF
Backward data-flow equations（后向数据流方程）, 624, 699~702
Bail, W. G., 719
Baker, B. S., 720
Baker, T. P., 387

Banning, J., 721
Barron, D. W., 82
Barth, J. M., 721
Basic block (基本块), 528~533, 591, 598~602, 609, 704。另见Extended basic block
Basic induction variable (基本归纳变量), 643
Basic symbol (基本符号), 95, 122
Basic type (基本类型), 345
Bauer, A. M., 387
Bauer, F. L., 82, 340, 386
BCPL, 511, 583
Beatty, J. C., 583
Beeber, R. J., 2
Begriffsschrift, 462
Belady, L. A., 583
Bell, J. R., 718
Bentley, J. L., 360, 462, 587
Best, S., 2
Binary alphabet (二进制字母表), 92
Binding, of names (名字的绑定), 395~396
Birman, A., 277
Bliss, 489, 542, 561, 583, 607, 718~719, 740~742
Block (块), 见Basic block, Block structure, Common block
Block structure (块结构), 412, 438~440
BNF, 25, 82, 159, 277
Bochmann, G. V., 341
Body (体), 见Procedure body
Boolean expression (布尔表达式), 326~328, 488~497, 501~503
Bootstrapping (自举), 725~729
Bottom-up parsing (自底向上语法分析), 41, 195, 290, 293~296, 308~316, 463。另见LR parsing, Operator precedence parsing, Shift-reduce parsing
Bounded context parsing (有界上下文语法分析), 277
Boyer, R. S., 158
Branch optimization (分支优化), 740, 742
Branquart, P., 342, 512
Branstad, M. A., 722
Bratman, H., 725
Break statment (break语句), 621~623
Brooker, R. A., 340
Brooks, F. p., 725
Brosgol, B. M., 341
Brown, C. A., 341
Bruno, J. L., 566, 584
Bucket (桶), 434。另见Hashing
Buffer (缓冲区), 57, 62, 88~92, 129
Burstall, R. M., 385
Busam, V. A., 718
Byte (字节), 399

C

C, 52, 104~105, 162~163, 326, 358~359, 364, 366, 396~398, 411, 414, 424, 442, 473, 482, 510, 542, 561, 589, 696, 725, 735~737
Call (调用), 见Procedure call
Call-by-address (传址方式), 见Call-by-reference
Call-by-location (传位置方式), 见Call-by-reference
Call-by-name (传名方式), 428~429
Call-by-reference (引用方式), 426
Call-by-value (传值方式), 424~426, 428~429
Calling sequence (调用序列), 404~408, 507~508
Canonical collection of sets of items (项目集规范族), 222, 224, 230~232
Canonical derivation (规范推导), 169
Canonical LR parsing (规范LR语法分析法), 230~236, 254
Canonical LR parsing table (规范LR语法分析表), 230~236
Cardelli, L., 387
Cardinael, J.-P., 342, 512
Carter, J. L., 718, 731
Carter, L. R., 583
Cartesian product (笛卡儿乘积), 345~346
Cartwright, R., 388
Case statement (case语句), 497~500
Cattell, R. G. G., 511, 584
CDC, 6600 584
CFG, 见Context-free grammar
Chaitin, G. J., 546, 583
Chandra, A. K., 546, 583
Chang, C. H., 278
Changed variable (变化变量), 657~660
Character class (字符类), 92, 97, 148。另见Alphabet
Cherniavsky, J. C., 720, 722
Cherry, L. L., 9, 158, 252, 733
Child (子节点), 29
Choe, K. M., 278
Chomsky, N., 81
Chomsky normal form (Chomsky范式), 276
Chow, F., 583, 719, 721
Church, A., 387, 462
Ciesinger, J., 278
Circular syntax-directed definition (环形语法制导定义), 287, 334~336, 340, 342
Cleveland, W. S., 462
Closure (闭包), 93~96, 123
Closure, of set of items (项目集的闭包), 222~223, 225, 231~232
CNF。见Chomsky normal form
Cobol, 731
Cocke, J., 160, 277, 546, 583, 718~721

Cocke-Younger-Kasami algorithm（Cocke-Younger-Kasami算法），160，277
Code generation（代码生成），15，513~584，736~737
Code hoisting（代码提升），714
Code motion（代码外提），596，638~643，710~711，721，742~743
Code optimization（代码优化），14~15，463，472，480，513，530，554~557，585~722，738~740
Coercion（强制类型转换），344，359~360，387
Coffman，E. G.，584
Cohen，R.，341
Coloring（染色），见Graph coloring
Column-major form（列优先形式），481~482
Comment（注释），84~85
Common block（公共块），432，446~448，454~455
Common Lisp 462
Common subexpression（公共子表达式），290，531，546，551，566，592~595，599~600，633~636，709，739~741，743。另见Available expression
Commutativity（交换性），96，684
Compaction，of storage（存储压缩），446
Compiler-compiler（编译器的编译器），22
Composition（合成），684
Compression（压缩），见Encoding of types
Concatenation（连接），34~35，92~96，123
Concrete syntax tree（具体语法树），49。另见Syntax tree
Condition code（条件码），541
Conditional expression（条件表达式），见Boolean expression
Configuration（格局），217
Conflict（冲突），见Disambiguation rule，Reduce/reduce conflict，Shift/reduce conflict
Confluence operator（聚合运算符），624，680，695
Congruence closure（全等闭包），见Congruent nodes
Congruent nodes（全等节点），385，388
Conservative approximation（保守近似），611，614~615，630，652~654，659~660，689~690
Constant folding（常量合并），592，595，601，681~685，687~688
Context-free grammar（上下文无关文法），25~30，40~41，81~82，165~181，280。另见LL grammar，LR grammar，Operator grammar
Context-free language（上下文无关语言），168，172~173，179~181
Contiguous evaluation（邻近计算），569~570
Control flow（控制流），66~67，468~470，488~506，556~557，606，611，621，689~690，720
Control link（控制链），398，406~409，423
Control stack（控制栈），393~394，396
Conway，M. E.，277
Conway，R. W.，164，278
Copy propagation（复制传播），592，594~595，636~638
Copy rule（复制规则），322~323，325，428~429
Copy statement（复制语句），467，551，594
Copy-restore-linkage（复制-恢复连接），427
Corasick，M. J.，158
Core，of set of items（项目集的核心），237
Cormack，G. V.，158，387
Courcelle，B.，341
Cousot，P.，720
Cousot，R.，720
CPL，387
Cross compiler（交叉编译器），726
Cross edge（交叉边），663
Curry，H. B.，387
Cutler，D.，719
Cycle（循环），177
Cycle，in type graphs（类型图中的环），358~359
Cycle-free grammar（无环的文法），270
CYK algorithm（CYK算法），见 Cocke-Younger-Kasami algorithm

D

DAG，见Directed acyclic graph
Dangling else（悬空else），174~175，191，201~202，249~251，263，268
Dangling reference（悬空引用），409，442
Data area（数据区），446，454~455
Data layout（数据布局），399，473~474
Data object（数据对象），见Object
DATA statement（DATA语句），402~403
Data-flow analysis（数据流分析），586，608~722
Data-flow analysis framework（数据流分析框架），681~694
Data-flow engine（数据流引擎），23，690~694
Data-flow equation（数据流方程），608，624，680
Date，C. J.，4
Daussmann，M.，387
Davidson，J. W，511，584
Dead code（无用代码），531，555，592，595
Debugger（调试器），406
Debugging（调试），555。另见Symbolic debugging
Declaration（声明），269，394~395，473~478，510
Decoration（装饰），见Annotated parse tree
Deep access（深访问），423
Default value（默认值），497~498
Definition（定义），529，610，632
Definition-use chain（定义-引用链），见Du-chain
Delescaille，J.-P. 342，512
Deletion，of locals（局部变量删除），404
DELTA，341
Demers，A. J.，278

Dencker, P., 158
Denotational semantics (指称语义学), 340
Dependency graph (依赖图), 279~280, 284~287, 331~334
Depth, of a flow graph (流图的深度), 664, 672~673, 716
Depth-first ordering (深度优先排序), 296, 661, 671~673
Depth-first search (深度优先搜索), 660~664
Depth-first spanning tree (深度优先生成树), 662
Depth-first traversal (深度优先遍历), 36~37, 324~325, 393
Deransart, P., 342
DeRemer, F., 278, 387
Derivation (推导), 29, 167~171
Despeyroux, T., 388
Deterministic finite automaton (确定的有穷自动机), 113, 115~121, 127~128, 132~136, 141~146, 150, 180~181, 217, 222, 224~226
Deterministic transition diagram (确定的状态转换图), 100
DFA, 见Deterministic finite automaton
Diagnostic (诊断的), 见Error message
Directed acyclic graph (无环有向图), 290~293, 347, 464~466, 471, 546~554, 558~561, 582, 584, 598~602, 606, 705~708
Disambiguating rule (消除二义性规则), 171, 175, 247~254, 261~264
Disambiguation (消除二义性), 387。另见Overloading
Display (display表), 420~422
Distance, between strings (字符串间的距离), 155
Distributive framework (分配性框架), 686~688, 692
Distributivity (分配性), 720
Ditzel, D., 583
Do statement (do语句), 86, 111~112
Dominator (支配节点), 602, 639, 670~671, 721
Dominator tree (支配树), 602
Downey, P. J., 388, 584
Drossopoulou, S., 387
Druseikis, F. C., 162
Du-chain (du链), 632~633, 643
Dummy argument (哑变元), 见Formal parameter
Durre, K., 158
Dynamic allocation (动态存储分配), 401, 440~446
Dynamic checking (动态检查), 343, 347
Dynamic programming (动态程序设计), 277, 567~572, 584
Dynamic scope (动态作用域), 411, 422~423

E

Earley, J., 181, 277~278, 721
Earley's algorithm (Earley算法), 160, 277
EBCDIC, 59
Edit distance (编辑距离), 156
Efficiency (效率), 85, 89, 126~128, 144~146, 152, 240~244, 279, 360, 388, 433, 435~438, 451, 516, 618~620, 724
另见Code optimization
Egrep, 158
EL1, 387
Elshoff, J. L., 583
Emitter (输出程序), 67~68, 72
Empty set (空集), 92
Empty string (空字符串), 27, 46, 92, 94, 96
Encoding of types (类型编码), 354~355
Engelfriet, J., 341
Englund, D. E., 718
Entry, to a loop (循环入口), 534
Environment (环境), 395, 457~458
ϵ-closure (ϵ闭包), 118~119, 225
ϵ-free grammar (无ϵ文法), 270
ϵ-production (ϵ产生式), 177, 189, 270
ϵ-transition (ϵ转换), 114~115, 134
EQN, 9~10, 252~254, 300, 723~724, 726, 733~734
Equel, 16
Equivalence, of basic blocks (基本块的等价), 530
Equivalence, of finite automata (有穷自动机的等价), 388
Equivalence, of grammars (文法的等价), 168
Equivalence, of regular expressions (正规表达式的等价), 95, 150
Equivalence, of syntax-directed definitions (语法制导定义的等价), 302~305
Equivalence, of type expressions (类型表达式的等价), 352~359。另见Unification
Equivalence statement (等价语句), 432, 448~455
Equivalence, under a substitution (代换下等价), 371, 377~379
Error handling (出错处理), 11, 72~73, 88, 161~162
另见Lexical error, Logical error, Semantic error, Syntax error
Error message (错误信息), 194, 211~215, 256~257
Error productions (出错产生式), 164~165, 264~266
Ershov, A. P., 341, 583, 718
Escape character (Escape字符), 110
Evaluation order, for basic blocks (基本块的计算顺序), 518, 558~561
Evaluation order, for syntax-directed definitions (语法制导定义的计算顺序), 285~287, 298~299, 316~336
Evaluation order, for trees (树的计算顺序), 561~580
Eve, J., 278
Explicit allocation (显式存储分配), 440, 443~444

Explicit type conversion（显式类型转换），359
Expression（表达式），6~7，31~32，166，290~293，350~351。另见Postfix expression
Extended basic block（扩展基本块），714
External reference（外部引用），19

F

Fabri，J.，718
Failure function（失败函数），151，154
Family，of an induction variable（归纳变量族），644
Fang，I.，341
Farrow，R.，341~342
Feasible type（可行的类型），363
Feldman，S. I.，157，511，729
Ferrante，J.，718
Feys，R.，387
Fgrep，158
Fibonacci string（Fibonacci串），153
Field，of a record（记录的域），477~478，488
Final state（终态），见Accepting state
Find（find操作），378~379
Finite automaton（有穷自动机），113~144。另见Transition diagram
FIRST，45~46，188~190，193
Firstpos，135，137~140
Fischer，C. N.，278，583~584
Fischer，M. J.，158，462
Fleck，A. C.，428
Flow graph（流图），528，532~534，547，591，602。另见Reducible flow graph
Flow of control（控制流），529。另见Control flow
Flow-of-control check（控制流检查），343
Floyd，R. W.，277，584
FOLDS，341
FOLLOW，188~190，193，230~231
Followpos，135，137~140
Fong，A. C.，721
Formal parameter（形参），390
Fortran 2，86，111~113，157，208，386，396，401~403，427，432，446~455，481，602，718，723
Fortran H，542，583，727~728，737~740
Forward data-flow equations（前向数据流方程），624，698~702
Forward edge（前向边），606
Fosdick，L. D.，722
Foster，J. M.，82，277
Fragmentation（碎片），443~444
Frame（帧），见Activation record
Fraser，C. W.，511~512，584
Fredman，M.，158
Frege，G.，462
Freiburghouse，R. A.，512，583
Freudenberger，S. M.，719，722
Front end（前端），20，62
Fukuya，S.，583
Function（函数），见Procedure
Function type（函数类型），346~347，352，361~364。另见Polymorphic function

G

GAG，341~342
Gajewska，H.，721
Galler，B. A.，462
Ganapathi，M.，583~584
Gannon，J. D.，163
Ganzinger，H.，341，387
Garbage collection（垃圾收集），441~442
Garey，M. R.，584
Gear，C. W.，720
Gen，608，612~614，627，636
Generation，of a string（串的生成），29
Generic function（类属函数），364。另见Polymorphic function
Geschke，C. M.，489，543，583，718~719，740
Giegerich，R.，341，512，584
Glanville，R. S.，579，584
Global error correction（全局错误纠正），164~165
Global name（全局名），653。另见Nonlocal name
Global optimization（全局优化），592，633
Global register allocation（全局寄存器分配），542~546
GNF，见Greibach normal form
Goldberg，R.，2
Goto，of set of items（项目集的转移），222，224~225，231~232，239
Goto statement（goto语句），506
Goto table（goto表），216
Graham，R. M.，277，462
Graham，S. L.，278，583~584，720
Graph（图），见Dependency graph，Directed acyclic graph，Flow graph，Interval graph，Reducible flow graph，Register-interference graph，Search，Transition graph，Tree，Type graph
Graph coloring（图染色），545~546
Grau，A. A.，512
Greibach normal form（Greibach范式），272
Grep，158

H

Haibt，L. M.，2
Haley，C. B.，278

Halstead, M. H., 511, 727
Handle (句柄), 196~198, 200, 205~206, 210, 225~226
Handle pruning (句柄裁剪), 197~198
Hanson, D. R., 512
Harrison, M. A., 278
Harrison, M. C., 158
Harrison, W. H., 583, 718, 722
Harry, E., 341
Hash function (散列函数), 434~438
Hash table (散列表), 498
Hashing (散列), 292~293, 433~440, 459~460
Hashpjw, 435~437
Head (头), 604
Header (首节点), 603, 611, 664
Heap (堆), 397, 735
Heap allocation (堆式存储分配), 401~402, 410~411, 440~446
Hecht, M. S., 718~721
Heinen, R., 719
Held, G., 16
Helsinki Language Processor (Helsinki语言处理器), 见HLP
Henderson, P. B., 720
Hennessy, J. L., 583, 721
Henry, R. R., 583~584
Herrick, H. L., 2
Heuft, J., 158
Hext, J. B., 387
Hierarchical analysis (层次分析), 5。另见Parsing
Hill, U., 512
Hindley, R., 387
HLP, 278, 341
Hoare, C. A. R., 82, 387
Hobbs, S. O., 489, 511, 543, 583~584, 718~719, 740
Hoffman, C. M., 584
Hole in scope (作用域中的洞), 412
Hollerith string (Hollerith字符串), 98
Hopcroft, J. E., 142, 157, 277, 292, 388, 392, 444~445, 462, 584, 587
Hope, 385
Hopkins, M. E., 546, 583, 719
Horning, J. J., 82, 163, 277~278
Horspool, R. N. S., 158
Horwitz, L. P., 583
Huet, G., 584
Huffman, D. A., 157
Hughes, R. A., 2
Hunt, J. W., 158
Huskey, H. D., 511, 727
Hutt, B., 341

I

IBM-7090, 584
IBM-370, 517, 569, 584, 737, 740
Ichbiah, J. D., 277
Idempotence (幂等性), 96, 684
Identifier (标识符), 56, 86~87, 179
Identity function (恒等函数), 683~684
If statement (if语句), 112~113, 491~493, 504~505
Ikeda, K., 583
Immediate dominator (直接支配节点), 602
Immediate left recursion (直接左递归), 176
Implicit allocation (隐式存储分配), 440, 444~446
Implicit type conversion (隐式类型转换), 359
Important state (重要状态), 134
Indexed addressing (变址寻址), 519, 540
Indirect addressing (间接寻址), 519~520
Indirect triples (间接三元式), 472~473
Indirection (间址), 472
Induction variable (归纳变量), 596~598, 643~648, 709, 721, 739
Infix expression (中缀表达式), 33
Ingalls, D. H. H., 387
Ingerman, P. Z., 82
Inherited attribute (继承属性), 34, 280, 283, 299, 308~316, 324~325, 340。另见Attribute
Initial node (初始节点), 532
Initial state (初始状态), 见State
In-line expansion (内嵌扩展), 428~429。另见Macro
Inner loop (内循环), 534, 605
Input symbol (输入符号), 114
Instance, of a polymorphic type (多态类型的实例), 370
Instruction selection (指令选择), 516~517
Intermediate code (中间代码), 12~14, 463~512, 514, 589, 704。另见Abstract machine, Postfix expression, Quadruple, Three-address code, Tree
Interpreter (解释器), 3~4
Interprocedural data flow analysis (过程间数据流分析), 653~660
Interval (区间), 664~667
Interval analysis (区间分析), 624, 660, 667, 720。另见 T_1-T_2 analysis
Interval depth (区间深度), 672
Interval graph (区间图), 666
Interval partition (区间划分), 665~666
Irons, E. T., 82, 278, 340
Ishihata, K., 722
Item (项目), 见Kernel item, LR(1) item, LR(0) item
Iterative data-flow analysis (迭代数据流分析), 624~633, 672~673, 690~694
Iverson, K., 387

J

Jacobi, Ch., 82, 511, 734
Janas, J. M., 387
Jarvis, J. F., 83, 158
Jazayeri, M., 341~342
Jensen K., 82, 511, 734, 745
Johnson, D. S., 584
Johnson, S. C., 4, 157~158, 257, 278, 340, 354~355, 383, 462, 511, 566, 572, 584, 731, 735~737
Johnson, W. L., 157
Johnsson, R. K., 489, 543, 583, 718~719, 740
Joliat, M., 278
Jones, N. D., 341, 387, 718, 720
Jourdan, M., 342
Joy, W. N., 278

K

Kaiserwerth, M., 158
Kam, J. B., 720
Kaplan, M. A., 387, 721
Karp, R. M., 388, 583
Kasami, T., 160, 277, 720
Kastens, U., 341~342
Kasyanov, V. N., 720
Katayama, T., 342
Keizer, E. G., 511
Kennedy, K., 341~342, 583, 720~721
Keohane, J., 720
Kernel item (核心项目), 223, 242
Kernighan, B. W., 9, 23, 82, 158, 252, 462, 723, 730, 733, 750
Keyword (关键字), 56, 86~87, 430
Kieburtz, R. B., 278
Kildall, G. A., 634, 680~681, 720
Kill (注销), 608, 612~614, 627, 636
Kiyono, T., 583
Kleene closure (克林闭包), 见Closure
Kleene, S. C., 157
KMP algorithm (KMP算法), 152
Knuth, D. E., 8, 23~24, 82, 157~158, 277, 340, 388, 444, 462, 583~584, 672~673, 721, 732
Knuth-Morris-Pratt algorithm (Knuth-Morris-Pratt算法), 158。另见KMP algorithm
Kolsky, H. G., 718, 737
Komlos, J., 158
Korenjak, A. J., 277~278
Kosaraju, S. R., 720
Koskimies, K., 341
Koster, C. H. A., 341
Kou, L., 720
Kreps, P., 16
Kristensen, B. B., 278
Kron, H., 584
Kruskal, J. B., 158

L

Label (标号), 66~67, 467, 506, 515
LaLonde, W. R., 278
LALR collection of sets of items (LALR项目集族), 238
LALR grammar (LALR文法), 239
LALR parsing (LALR语法分析), 见Lookahead LR parsing
LALR parsing table (LALR语法分析表), 236~244
Lamb, D. A., 584
Lambda calculus (λ演算), 387
Lampson, B. W., 462
Landin, P. J., 462
Langmaack, H., 512
Language (语言), 28, 92, 115, 168, 203
Lassagne, T., 584
Lastpos, 135, 137~140
Lattice (栅格), 387
L-attributed definition (L属性定义), 280, 296~318, 341
Lazy state construction (惰性状态构造), 128, 158
Leader (人口语句), 529
Leaf (叶子), 29
Lecarme, O., 727
Ledgard, H. F., 388
Left associativity (左结合), 30~31, 207, 263
Left factoring (提取左因子), 178~179
Left leaf (左叶子), 561
Left recursion (左递归), 47~50, 71~72, 176~178, 182, 191~192, 302~305
Leftmost derivation (最左推导), 169
Left-sentential form (左句型), 169
Leinius, R. P., 278
Lengauer, T., 721
Lesk, M. E., 157, 731
Leverett, B. W., 511, 583~584
Levy, J. P., 278
Levy, J.-J., 584
Lewi, J., 342, 512
Lewis, H. R., 720
Lewis, P. M., 277, 341
Lex, 83, 105~113, 128~129, 148~149, 158, 730
Lexeme (词素), 12, 54, 56, 61, 85, 430~431
Lexical analysis (词法分析), 5, 12, 26, 54~60, 71, 83~158, 160, 172, 261, 264, 738
Lexical environment (词法环境), 457~458
Lexical error (词法错误), 88, 161

Lexical scope（词法作用域），411~422
Lexical value（词法值），12，111，281
Library（库），4~5，54
Lifetime，of a temporary（临时变量的生存期），480
Lifetime，of an activation（活动记录的生存期），391，410
Lifetime，of an attribute（属性的生存期），320~322，324~329
Limit flow graph（限制流图），666，668
Linear analysis（线性分析），4。另见Lexical analysis
LINGUIST，341~342
Link editor（连接编辑器），19，402
Linked list（链表），432~433，439
Lint（lint命令），347
Lisp，411，440，442，461，694，725
Literal string（文字串），86
Live variable（活跃变量），534~535，543，551，595，599~600，631~632，642，653
LL grammar（LL文法），160，162，191~192，221，270，273，277，307~308
Loader（装配器），19
Local name（局部名字），394~395，398~400，411
Local optimization（局部优化），592，633
Loewner，P. G.，718
Logical error（逻辑错误），161
Longest common subsequence（最长公共子序列），155
Lookahead（向前看，超前扫描），90，111~113，134，215，230
Lookahead LR parsing（向前看LR语法分析），216，236~244，254，278。另见Yacc
Lookahead symbol（超前扫描符），41
Loop（循环），533~534，544，602~608，616~618，660
Loop header（循环头，循环首节点），见Header
Loop optimization（循环优化），596。另见Code motion，Induction variable，Loop unrolling，Reduction in strength
Loop-invariant computation（循环不变计算），见Code motion
Lorho，B.，341~342
Low，J.，721
Lowry，E. S.，542，583，718，721，727，737
LR grammar（LR文法），160，162，201~202，220~221，247，273，278，308
LR parsing（LR语法分析），215~266，341，578~579
LR(1) grammar（LR(1)文法），235
LR(1) item（LR(1)项目），230~231
LR(0) item（LR(0)项目），221
Lucas，P.，82，277
Lunde，A.，583
Lunnel，H.，583
L-value（左值），64~65，229；395，424~429，547

M

Machine code（机器代码），4~5，18，557，569
Machine status（机器状态），398，406~408
MacLaren，M. D.，719
MacQueen，D. B.，385，388
Macro（宏），16~17，84，429，456
Madsen，C. M.，341
Madsen，O. L.，278，341~342
Make，729~731
Manifest constant（符号常量），107
Marill，T.，342，583
Marker nonterminal（标记非终结符），309，311~315，341
Markstein，J.，719，721
Markstein，P. W.，546，583
Martelli，A.，388
Mauney，J.，278
Maximal munch（最大贪吃），578
Maxwell，W. L.，278
Mayoh，B. H.，340
McArthur，R.，511，727
McCarthy，J.，82，461，725
McClure，R. M.，277
McCracken，N. J.，388
McCulloch，W. S.，157
McIlroy，M. D.，158
McKeeman，W. M.，82，277，462，584
McKusick，M. K.，584
McLellan，H. R.，583
McNaughton，R.，157
Medlock，C. W.，542，583，718，721，727，737
Meertens，L.，387
Meet operator（与操作符），681
Meet-over-paths solution（聚合路径解），689~690，692，694
Memory map（内存映像），446
Memory organization（内存组织），见Storage organization
META，277
Metcalf，M.，718
Meyers，R.，387
Miller，R. E.，583，720
Milner，R.，365，387
Minimum-distance error correction（最小距离错误校正），88
Minker，J.，512
Minker，R. G.，512
Mitchell，J. C.，387
Mixed strategy precedence（混合策略优先法），277
Mixed-mode expression（混合模式表达式），495~497

ML, 365, 373~374, 387
Mock, O., 82, 511, 725
Modula, 607, 719, 742~743
Moncke, U., 341
Monotone framework (单调框架), 686~688, 692
Monotonicity (单调性), 720
Montanari, U., 388
Moore, E. F., 157, 721
Moore, J. S., 158
Morel, E., 720
Morris, D., 340
Morris, J. H., 158, 387~388
Morris, R., 462
Morse, S. P., 277
Moses, J., 462
Most closely nested rule (最近嵌套规则), 412, 415
Most general unifier (最一般的合一代换), 370~371, 376~377
Moulton, P. G., 278
Muchnick, S. S., 387, 718, 720
MUG, 341
Muller, M. E., 278

N

Nageli, H. H., 82, 511, 734
Nakata, I., 583
Name (名字), 389
Name equivalence, of type expressions (类型表达式的名字等价), 356~357
Name-related check (与名字有关的检查), 343~344
Natural loop (自然循环), 603~604
Naur, P., 82, 277, 386, 461
NEATS, 342
Neliac, 511, 727
Nelson, R. A., 2
Nested procedures (嵌套过程), 415~422, 474~477
Nesting depth (嵌套深度), 416
Nesting, of activations (活动的嵌套), 391。另见 Block structure
Newcomer, J. M., 511, 584
Newey, M. C., 511~512
NFA, 见Nondeterministic finite automaton
Nievergelt, J., 583
Node splitting (节点分裂), 666~668, 679~680
Nondeterministic finite automaton(不确定的有穷自动机), 113~114, 117~128, 130~132, 136
Nondeterministic transition diagram (不确定的状态转换图), 184
Nonlocal name (非局部名字), 395, 411~424, 528
Nonreducible flow graph (不可约流图), 607, 679~680
Nonregular set (非正规集), 180~181。另见Regular set
Nonterminal (非终结符), 26, 166~167, 204~205。另见Marker nonterminal
Nori, K. V., 82, 511, 734
Nullable expression (可空表达式), 135, 137~140
Nutt, R., 2

O

Object (对象), 389, 395
Object code (目标代码), 704
Object language (目标语言), 见Target language
O'Donnell, M. J., 584
Offset (偏移), 397, 400, 450, 473, 524
Ogden, W. F., 342
Olsztyn, J., 82, 511, 725
One-pass compiler (一遍编译器), 见Single-pass translation
Operator grammar (算符文法), 203~204
Operator identification (操作符标识), 361。另见Overloading
Operator precedence (算符优先), 31
Operator precedence grammar (算符优先文法), 271~272
Operator precedence parsing (算符优先语法分析), 203~215, 277, 736
Operator precedence relations (算符优先关系), 203~204
Optimizing compiler (优化编译器), 见Code optimization
Osterweil, L. J., 722
Overloading (重载), 330, 344, 361~364, 384~385, 387

P

Packed data (压缩数据), 399
Padding (填充空白区), 399
Pager, D., 278
Pai, A. B., 278
Paige, R., 721
Pair, C., 342
Palm, R. C., 721
Panic mode (紧急方式), 88, 164, 192~193, 254
Panini, 82
Parameter passing (参数传递), 414~415, 424~429, 653~654
Parentheses (圆括号), 95, 98, 173~174
Park, J. C. H., 278
Parse tree (分析树), 6~7, 28~30, 40~43, 49, 160, 169~171, 196, 279, 296。另见Syntax tree
Parser generator (语法分析器生成器), 22~23, 730~731。另见Yacc
Parsing (语法分析), 6~7, 12、30, 40~48, 57, 71~72, 84~85, 159~278。另见Bottom-up parsing, Bounded context parsing, Top-down parsing
Partial order (偏序), 333~335

Partition（划分），142
Pascal，52，85，94，96~97，162~163，347，349，356~357，365~366，396~398，411，424~425，427，440~442，473，481，510~512，583，719，727~729，734~735
Pass（遍），20~22
Passing environment（传递环境），457~458
Paterson，M. S.，388
Pattern matching（模式匹配），85 129~131，577~578，584
PCC，519，572，584，735~737
P-code（P代码），734，742
Peephole optimization（窥孔优化），554~558，584，587
Pelegri-Llopart，E.，584
Pennello，T.，278，387
Period，of a string（字符串的周期），152
Persch，G.，387
Peterson，T. G.，278
Peterson，W. W.，462，720
Peyrolle-Thomas，M. -C.，727
Phase（阶段），10。另见Code generation，Code optimization，Error handling，Intermediate code，Lexical analysis，Parsing，Semantic analysis，Symbol table
Phrase-level error recovery（短语级错误恢复），164~165，194~195，255
Pic，456
Pig Latin（倒读隐语），79~80
Pike，R.，82，462，730，750
Pitts，W.，157
Plankalkul，386
PL/C，164，514
PL/I，21，80，87，162~163，383，387，488，510，512，719
Plotkin，G.，388
Point（点），609
Pointer（指针），349，409，467~468，540~541，553，582，648~653
Pointer type（指针类型），346
Pollack，B. W.，24
Pollock，L. L.，722
Polymorphic function（多态函数），344，364~376
Polymorphic type（多态类型），368
Poole，P. C.，511
Pop（弹出），65
Portability（可移植性），85，724
Portable C compiler（可移植的C编译器），见PCC
Porter，J. H.，157
Positive closure（正闭包），97。另见Closure
Post，E.，82
Postfix expression（后缀表达式），25，33~34，464，466，470，509
Postorder traversal（后根遍历），561~562
Powell，M. L.，719，721，742
Pozefsky，D.，342
Pratt，T. W.，462
Pratt，V. R.，158，277
Precedence（优先），31~32，95，207，247~249，263~264。另见Operator precedence grammar
Precedence function（优先函数），208~210
Precedence relations（优先关系），见Operator precedence relations
Predecessor（前驱），532
Predictive parsing（预测语法分析法），44~48，182~188，192~195，215，302~308
Predictive translator（预测翻译器），306~308
Prefix（前缀），93
Prefix expression（前缀表达式），509
Preheader（前置首节点），605
Preprocessor（预处理器），4~5，16
Pretty printer（智能打印机），3
Procedure（过程），389
Procedure body（过程体），390
Procedure call（过程调用），202，396，398，404~411，467，506~508，522~527，553，649。另见Interprocedural data flow analysis
Procedure definition（过程定义），390
Procedure parameter（过程参数），414~415，418~419
Product（乘积），见Cartesian product
Production（产生式），26，166
Profiler（描述器），587
Programming language（程序设计语言），见Ada，Algol，APL，BCPL，Bliss，C，Cobol，CPL，EL1，Fortran，Lisp，ML，Modula，Neliac，Pascal，PL/I，SETL，SIMPL，Smalltalk，Snobol
Programming project（程序设计项目），745~751
Project，programming（项目，程序设计），745~751
Propagation，of lookahead（搜索符的传播），242
Prosser，R. T.，721
Purdom，P. W.，341，721
Push（压入），65

Q

Quadruples（四元式），470，472~473
Query interpreter（查询解释器），4
Queue（队列），507~508
Quicksort（快速分类），390，588

R

Rabin，M. O.，157
Radin，G.，387
Raiha，K. -J.，278，341~342

Ramanathan, J., 341
Randell, B., 24, 82, 462, 512
Ratfor, 723
Reaching definition (到达定义), 610~621, 624~627, 652~653, 674~680, 684
Recognizer (识别器), 113
Record type (记录类型), 346, 359, 477~478
Recursion (递归), 6~7, 165, 316~318, 329~332, 391, 401。另见Left recursion, Right recursion, Tail recursion
Recursive-descent parsing (递归下降语法分析法), 44, 82, 181~182, 736, 740
Reduce/reduce conflict (归约-归约冲突), 201, 237, 262, 578
Reducible flow graph (可约流图), 606~608, 664, 666, 668, 714~715, 740
Reduction (归约), 195, 199, 211~213, 216, 255
Reduction in strength (强度削弱), 557, 596~598, 601, 644, 646
Redundant code (冗余代码), 554
Redziejowski, R. R., 583
Reference (引用), 见Use
Reference count (引用计数), 445
Region (区域), 611~612, 669~670, 673~679,
Register (寄存器), 519~521
Register allocation (寄存器分配), 517, 542~546, 562~565, 739~743
Register assignment (寄存器指派), 15, 517, 537~540, 544~545
Register descriptor (寄存器描述符), 537
Register pair (寄存器对), 517, 566
Register-interference graph (寄存器冲突图), 546
Regression test (回归测试), 731
Regular definition (正规定义), 96, 107
Regular expression (正规表达式), 83, 94~98, 107, 113, 121~125, 129, 135~141, 148, 172~173, 268~269
Regular set (正规集), 98
Rehostability (可变换宿主机能力), 724
Reif, J. H., 720
Reiner, A. H., 511, 584
Reiss, S. P., 462
Relative address (相对地址), 见Offset
Relocatable machine code (可重定位机器代码), 5, 18, 515
Relocation bit (重定位标志位), 19
Renaming (重命名), 531
Renvoise, C., 720
Reps, T. W., 341
Reserved word (保留字), 56, 87
Retargetability (可重置目标能力), 724
Retargeting (重置目标), 463
Retention, of locals (局部名字的保持), 401~403, 410
Retreating edge (后退边), 663
Return address (返回地址), 407, 522~527
Return node (返回节点), 654
Return sequence (返回序列), 405~408
Return value (返回值), 399, 406~407
Reynolds, J. C., 388
Rhodes, S. P., 278
Richards, M., 511, 584
Right associativity (右结合), 30~31, 207, 263
Right leaf (右叶子), 561
Right recursion (右递归), 48
Rightmost derivation (最右推导), 169, 195~197
Right-sentential form (右句型), 169, 196
Ripken, K., 387, 584
Ripley, G. D., 162
Ritchie, D. M., 354, 462, 511, 735~737
Robinson, J. A., 388
Rogoway, H. P., 387
Rohl, J. S., 462
Rohrich, J., 278
Root (根), 29
Rosen, B. K., 719~720
Rosen, S., 24
Rosenkrantz, D. J., 277, 341
Rosler, L., 723
Ross, D. T., 157
Rounds, W. C., 342
Rovner, P., 721
Row-major form (行优先形式), 481~482
Run-time support (运行时支撑), 389。另见Heap allocation, Stack allocation
Russell, L. J., 24, 82, 462, 512
Russell, S. R., 725
Ruzzo, W. L., 278
R-value (右值), 64~65, 229, 395, 424~429, 547
Ryder, B. G., 721~722

S

Saal, H. J., 387
Saarinen, M., 278, 341~342
Safe approximation (安全近似), 见Conservative approximation
Samelson, K., 340
Sankoff, D., 158
Sannella, D. T., 385
Sarjakoski, M., 278, 341
S-attributed definition (S属性定义), 281, 293~296
Save statement (SAVE语句), 402~403
Sayre, D., 2

Scanner（扫描器），84
Scanner generator（扫描器生成器），23。另见Lex
Scanning（扫描），见Lexical analysis
Scarborough，R. G.，718，737
Schaefer，M.，718
Schaffer，J. B.，719
Schatz，B. R.，511，584
Schonberg，E.，721
Schorre，D. V.，277
Schwartz，J. T.，387，583，718~721
Scope（作用域），394，411，438~440，459，474~479
Scott，D.，157
Search，of a graph（图的搜索），119。另见Depth-first search
Sedgewick，R.，588
Semantic action（语义动作），37~38，260
Semantic analysis（语义分析），5，8
Semantic error（语义错误），161，348
Semantic rule（语义规则），33，279~287
Semantics（语义学），25
Sentence（句子），92，168
Sentential form（句型），168
Sentinel（哨兵），91
Sethi，R.，342，388，462，566，583~584
SETL，387，694~695，719
Shallow access（浅访问），423
Shared node（共享节点），566~568
Sharir，M.，719，721
Shell，149
Sheridan，P. B.，2，277，386
Shift（移进），199，216
Shift/reduce conflict（移进-归约冲突），201，213~215，237，263~264，578
Shift-reduce parsing（移进-归约分析），198~203，206。另见LR parsing，Operator precedence parsing
Shimasaki，M.，583
Short-circuit code（短路代码），490~491
Shustek，L. J.，583
Side effect（副作用），280
Signature，of a DAG node（DAG节点的签名），292
Silicon compiler（硅编译器），4
SIMPL，719
Simple LR parsing（简单LR语法分析），216，221~230，254，278
Simple precedence（简单优先），277
Simple syntax-directed translation（简单语法制导翻译），39~40，298
Single production（单产生式），248，270
Single-pass translation（单遍翻译），279，735
Sippu，S.，278，341
Skeletal parse tree（分析树架子），206
SLR grammar（SLR文法），228
SLR parsing（SLR语法分析），见Simple LR parsing
SLR parsing table（SLR语法分析表），227~230
Sneeringer，W. J.，387
Snobol，411
Soffa，M. L.，722
Soisalon-Soininen，E.，277~278，341
Sound type system（完备的类型系统），348
Source language（源语言），1
Spillman，T. C.，721
Spontaneous generation，of lookahead（超前搜索符的自生），241
Stack（堆栈），126，186，198，217，255，275~276，290，294~296，310~315，324~328，393~394，397，476~479，562，735。另见Control stack
Stack allocation（栈式存储分配），401，404~412，522，524~528
Stack machine（堆栈机），62~69，464，584
Start state（开始状态），100
Start symbol（初始符号），27，29，166，281
State（状态），100，114，153，216，294
State minimization（状态最小化），141~144
State (of storage)（（存储单元的）状态），395
Statement（语句），26，28，32，67，352。另见Assignment statement，Case statement，Copy statement，Do statement，Equivalence statement，Goto statement，If statement，While statement
Static allocation（静态存储分配），401~403，522~524，527~528
Static checking（静态检查），3，343，347，722
Static scope（静态作用域），见Lexical scope
Staveren，H. van，511，584
Stdio.h，58
Stearns，R. E.，277，341
Steel，T. B.，82，511，725
Steele，G. L.，462
Stern，H.，2
Stevenson，J. W.，511，584
Stockhausen，P. F.，584
Stonebraker，M.，16
Storage（存储），395
Storage allocation（存储分配），401~411，432，440~446
Storage organization（存储组织），396~400
String（字符串），92，167
String table（字符串表），431
Strong，J.，82，511，725
Strongly noncircular syntax-directed definition（强无环语法制导定义），332~336，340
Strongly typed language（强类型语言），348
Stroustrup，B.，437
Structural equivalence，of type expressions（类型表达式

的结构等价), 353~355, 376, 380
Structure editor (结构编辑器), 3
Subsequence (子序列), 93。另见 Longest common subsequence
Subset construction (子集构造), 117~121, 134
Substitution (代换), 370~371, 376~379
Substring (子串), 93
Successor (后继), 532
Suffix (后缀), 93
Sussman, G. J., 462
Suzuki, N., 387, 722
Switch statement (switch语句), 见Case statement
Symbol table (符号表), 7, 11, 60~62, 84, 160, 429~440, 470, 473, 475~480, 703
Symbolic debugging (符号调试), 703~711
Symbolic dump (符号转储), 536
Symbolic register (符号寄存器), 545
Synchronizing token (同步记号), 192~194
Syntax (语法), 25。另见Context-free grammar
Syntax analysis (语法分析), 见Parsing
Syntax error (语法错误), 161~165, 192~195, 199, 206, 210~215, 218, 254~257, 264~266, 275, 278
Syntax tree (语法树), 2, 7, 49, 287~290, 464~466, 471。另见Abstract syntax tree, Concrete syntax tree, Parse tree
Syntax-directed definition (语法制导定义), 33, 279~287。另见Annotated parse tree, Syntax-directed translation
Syntax-directed translation (语法制导翻译), 8, 25, 33~40, 46~54, 279~342, 464~465, 468~470
Syntax-directed translation engine (语法制导翻译引擎), 23。另见GAG, HLP, LINGUIST, MUG, NEATS
Synthesized attribute (综合属性), 34, 280~282, 298~299, 316, 325。另见Attribute
Szemeredi, E., 158
Szymanski, T. G., 158, 584

T

Table compression (表压缩), 144~146, 151, 244~247
Table-driven parsing (表驱动分析), 186, 190~192, 216~220。另见Canonical LR parsing, LALR parsing, Operator precedence parsing, SLR parsing
Tai, K. C., 278
Tail (尾), 604
Tail recursion (尾递归), 52~53
Tanenbaum, A. S., 511, 584
Tantzen, R. G., 82
Target language (目标语言), 1
Target machine (目标机器), 724
Tarhio, J., 341
Tarjan, R. E., 158, 388, 462, 720~721
T-diagram (T型图), 725~728
Temporary (临时变量), 398, 470, 480~481, 535, 635, 639
Tennenbaum, A. M., 387, 720~721
Tennent, R. D., 462
Terminal (终结符), 26, 165~167, 281
Testing (测试), 731~732
T_EX, 8~9, 16~17, 82, 731
Text editor (文本编辑器), 158
Text formatter (文本格式器), 4, 8~10
Thompson, K., 122, 158, 601, 735
Three-address code (三地址码), 13~14, 466~472
Thunk (形实转换程序), 429
Tienari, M., 278, 341
Tjiang, S., 584
TMG, 277
Token (记号), 4~5, 12, 26~27, 56, 84~86, 98, 165, 179
Tokuda, T., 278
Tokura, N., 720
T_1-T_2 analysis (T_1-T_2分析), 667~668, 673~680
Tools (工具), 724。另见Automatic code generator, Compiler-compiler, Data-flow engine, Parser generator, Scanner generator, Syntax-directed translation engine
Top element (栈顶元素), 684
Top-down parsing (自顶向下语法分析), 41~48, 176, 181~195, 302, 341, 463。另见Predictive parsing, Recursive-descent parsing
Topological sort (拓扑排序), 285, 551
Trabb Pardo, L., 24
Transfer function (转移函数), 674, 681, 689
Transition diagram (状态转换图), 99~105, 114, 183~185, 226。另见Finite automaton
Transition function (转换函数), 114, 153~154
Transition graph (转换图), 114
Transition table (转换表), 114~115
Translation rule (翻译规则), 108
Translation scheme (翻译模式), 37~40, 297~301
Translator-writing system (翻译器编写系统), 见Compiler-compiler
Traversal (遍历), 36, 316~319。另见Depth-first traversal
Tree (树), 2, 347, 449。另见 Activation tree, Depth-first spanning tree, Dominator tree, Syntax tree
Tree rewriting (树重写), 572~580, 584
Tree-translation scheme (树翻译模式), 574~576
Trevillyan, L. H., 718
Trickey, H. W., 4
Trie, 151, 153~154
Triples (三元式), 470~472

Tritter, A., 82, 511, 725
TROFF, 726, 733~734
Two-pass assembler (两遍汇编程序), 18
Type (类型), 343~388
Type checking (类型检查), 8, 343~344, 347, 514
Type constructor (类型构造符), 345
Type conversion (类型转换), 359~360, 485~487。另见 Coercion
Type estimation (类型估计), 694~702
Type expression (类型表达式), 345~347
Type graph (类型图), 347, 353, 357~359
Type inference (类型推断), 366~367, 373~376, 694
Type name (类型名), 345~346, 356
Type system (类型系统), 347~348, 697~698
Type variable (类型变量), 366

U

Ud-chain (ud链), 621, 642~643
Ukkonen, E., 277
Ullman, J.D., 4, 142, 157, 181, 204, 277~278, 292, 387, 392, 444~445, 462, 566, 583~584, 587~588, 720~721
Unambiguous definition (明确定义), 610
Unary operator (一元运算符), 208
UNCOL, 82, 511
Unification (合一), 370~372, 376~380, 388
Union (并), 93~96, 122~123, 378~379
Uniqueness check (惟一性检查), 343
Unit production (单位产生式), 见Single production
Universal quantifier (全称量词), 367~368
UNIX, 149, 158, 257, 584, 725, 735
Unreachable code (不可到达代码), 见Dead code
Upwards exposed use (向上暴露的引用), 633
Usage count (引用计数), 542~544, 583
Use (引用), 529, 534~535, 632
Use-definition chain (引用-定义链), 见Ud-chain
Useless symbol (无用符号), 270

V

Valid item (有效项目), 225~226, 231
Value number (值编号), 292~293, 635
Value-result linkage (传值结果连接), 见Copy-restore-linkage
Van Staveren, 见Staveren, H. van
Vanbegin, M., 342, 512
Variable (变量), 见Identifier, Type variable
Variable-length data (变长数据), 406, 408~409, 413
Very busy expression (非常忙表达式), 713~714
Viable prefix (活前缀), 201, 217, 224~225, 230~231
Void type (空类型), 345, 352
Vyssotsky, V., 719

W

Wagner, R. A., 158
Waite, W. M., 511~512, 583~584, 720, 731
Walter, K. G., 341
Ward, P., 341
Warren, S. K., 342
Wasilew, S. G., 584
WATFIV, 514
Watt, D. A., 341
Weak precedence (弱优先), 277
WEB, 732
Weber, H., 277
Wegbreit, B., 387, 720
Wegman, M. N., 388, 720~721
Wegner, P., 719
Wegstein, J. H., 82, 157, 511, 725
Weihl, W. E., 721
Weinberger, P. J., 158, 435
Weingart, S., 584
Weinstock, C. B., 489, 543, 583, 718~719, 740
Welsh, J., 387
Wexelblat, R. L., 24, 82
While statement (while语句), 491~493, 504~505
White space (空白符), 54, 84~85, 99
Wilcox, T. R., 164, 278
Wilhelm, R., 341, 512
Winograd, S., 583
Winterstein, G., 387
Wirth, N., 82, 277~278, 462, 512, 727, 734, 745
Wong, E., 16
Wood, D., 277
Word (字), 92
Wortman, D. B., 82, 277
Wossner, H., 386
Wulf, W. A., 489, 511, 543, 583~584, 718~719, 740

Y

Yacc, 257~266, 730, 736, 742
Yamada, H., 157
Yannakakis, M., 584
Yao, A. C., 158
Yellin, D., 342
Yield (结果), 29
Younger, D. H., 160, 277

Z

Zadeck, F. K., 720
Zelkowitz, M. V., 719
Ziller, I., 2
Zimmermann, E., 341
Zuse, K., 386