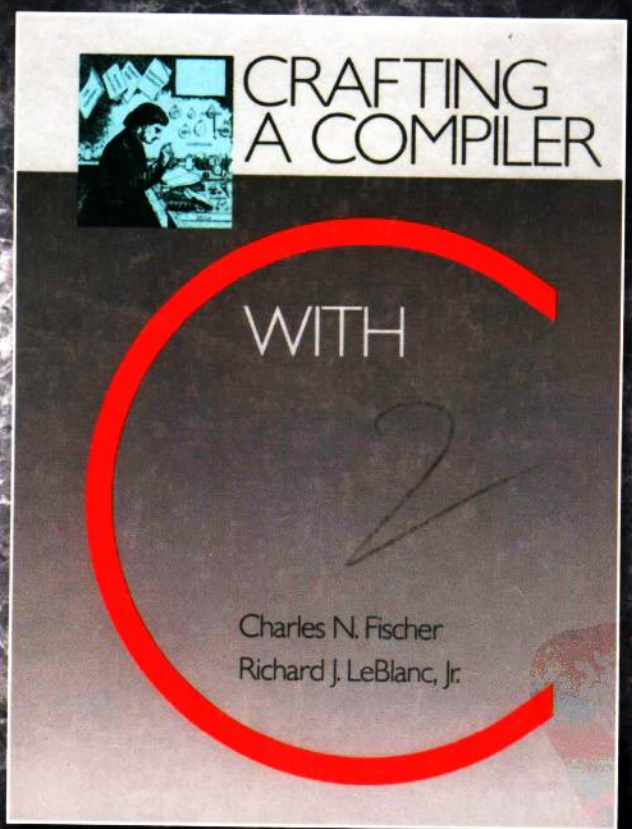
计 算 机 科 学 丛 书

编译器构造

C语言描述

(美) Charles N. Fischer 威斯康星大学麦迪逊分校 Richard J. LeBlanc, Jr. 佐治亚理工学院 著 郑启龙 姚震 译



Crafting A Compiler with C

机械工业出版社
China Machine Press

本书是一本优秀的编译器构造方面的教材，适合高等院校计算机专业的学生及专业程序员使用，已经被国际上多所大学或学院选作教材。本书提供了创新的编译器构造方法，通过大量的示例和练习，描述如何从头至尾设计一个可用的编译器。书中均衡覆盖了编译器设计中的理论与实现两大部分，详细讨论了标准编译器设计的相关主题（如自顶向下和自底向上的语法分析、语义分析、中间表示和代码生成）。本书中所有的程序均采用易读的基于C语言的代码来表示。

本书特色：

- 均衡讨论了编译器设计的理论与实现两大部分，既很好地介绍了编译器理论，又提供了大量的编译器设计示例和练习。
- 强调使用可以生成语法分析器和词法分析器的编译器工具。
- 彻底讨论LR语法分析和归约技术。
- 介绍了FLex和ScanGen。
- 在每章末尾包含可选的高级主题。

作者简介

Charles N. Fischer 是威斯康星大学麦迪逊分校的计算机教授，他的研究兴趣主要包括编译器设计和实现等。



Richard J. LeBlanc, Jr. 是佐治亚理工学院计算学院的教授和副主任，ACM和IEEE计算机协会的会员，他的研究兴趣主要包括软件工程、编程语言设计和实现、编程环境等。



www.PearsonEd.com

ISBN 7-111-16474-1

9 787111 164746

封面设计：陈子平

HZ BOOKS

华章图书

上架指导：计算机/编译原理

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

投稿热线：(010) 88379604

购书热线：(010) 68995259, 68995264

读者信箱：hzjsj@hzbook.com

ISBN 7-111-16474-1/TP · 4281

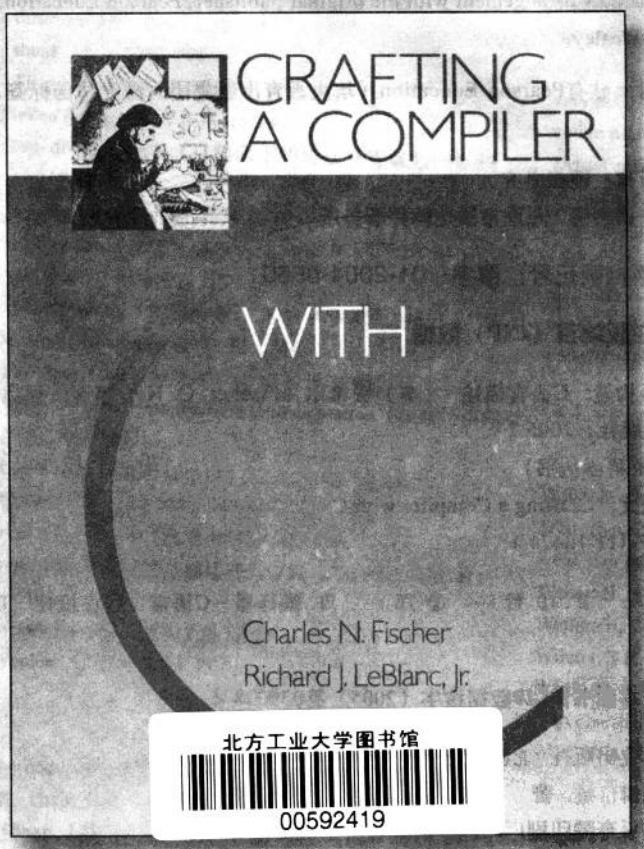
定价：65.00 元

TP312
1643

计 算 机 科 学 丛

编译器构造
C语言描述

(美) Charles N. Fischer 威斯康星大学麦迪逊分校 Richard J. LeBlanc, Jr. 佐治亚理工学院 著 郑启龙 姚震 译



北方工业大学图书馆
00592419

Crafting A Compiler with C

机械工业出版社
China Machine Press

SJS47 106

本书均衡讲述了编译器设计中的理论与实现两大部分，详细讨论了标准编译器设计的相关主题（如自顶向下和自底向上的语法分析、语义分析、中间表示和代码生成），提供了创新的编译器构造方法，使读者可以从头至尾地学习如何设计一个可用的编译器。

本书是一本优秀的编译器构造方面的教材，适合于高等院校计算机专业的学生和使用C语言的专业程序员。

Simplified Chinese edition copyright © 2005 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *Crafting a Compiler with C* (ISBN 0-8053-2166-7) by Charles N. Fischer and Richard J. LeBlanc, Jr., Copyright © 1991.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Addison-Wesley.

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

版权所有，侵权必究。

本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2004-0560

图书在版编目（CIP）数据

编译器构造：C语言描述 /（美）费希尔（Fischer, C. N.）等著；郑启龙等译. -北京：机械工业出版社，2005.7

（计算机科学丛书）

书名原文：Crafting a Compiler with C

ISBN 7-111-16474-1

Ⅰ. 编 … Ⅱ. ① 费 … ② 郑 … Ⅲ. 编译器－C语言－程序设计 Ⅳ. ① TN762 ② TP312

中国版本图书馆CIP数据核字（2005）第039038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：姚 蕾

北京昌平奔腾印刷厂印刷 · 新华书店北京发行所发行

2005年7月第1版第1次印刷

787mm × 1092mm 1/16 · 34.25印张

印数：0 001-4 000册

定价：65.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅illegible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
周立柱	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

当今计算机体系结构正处于从32位向64位跨越的时代，除了硬件本身的革命性设计之外，要想充分发挥64位硬件计算平台的效率并以此提高应用软件的性能，编译器作为现代计算机中最重要的系统软件之一，其作用是无可替代的。因此，编译器的原理与设计应该是计算机专业学生所必须学习和掌握的一门重要的专业基础课。尽管最终从事编译器构造这种浩繁而艰巨的工程的人只是少数，但通过对编译器原理的学习、对其设计技巧的掌握，可以使学生对诸如程序设计语言、数据结构和算法设计与分析等诸多计算机专业基础课程有更加深入的理解与体会。通过对编译器这个沟通软硬件的“翻译”桥梁角色的认识，还可以让学生对计算机软、硬件的相互依存、相互支持的关系有更进一步的领悟与洞察。而这一切的获得往往开始于选择一本好的教材、一本有益的参考书。可以这么说，好的选择是成功道路上坚实的第一步。

由Charles N. Fischer和Richard J. LeBlanc, Jr两位教授合著的本书就是一本这样的书。本书中文版的第一译者最早在11年前就接触了本书的原版，从中受益并采撷了部分知识用于当时的科研工作，之后在中国科学技术大学多次讲授“编译原理与技术”课程时也就一直把它作为重要的不可或缺的教学参考书。

正如本书的书名首先让人想到的，它是一本告诉人们如何用C语言去实际地构造编译器的书，这正是本书的最大特色——完美而均衡地覆盖了编译器的理论和实现的相关议题。本书使读者在阅读深奥而枯燥的理论的同时，又能享受精妙而细致的实现，虽精巧但又不落于旁支细节，既可信手拈来又给人以启迪与思考的空间，在精彩或重要之处作者更是不惜笔墨。另外，本书还是一本编译器构造理论方面的有价值的专著，书中不乏围绕构造一个编译器而进行的系统全面、内容详实的理论介绍。全书正文共分17章，其中包括了从最基本的词法分析到语法分析，再到语义处理、运行时存储环境、代码生成和代码优化，从符号表的组织到程序错误的处理等编译器各个典型的组成部分的周到的分析与讨论；所涉及的程序设计语言从较早期的Algol 60、Modula-2到美国国防部专用的、功能丰富但也十分复杂的Ada语言，从简单的Pascal语言到灵活的C语言，再到时下流行的专业程序员首选的C++语言等。

尽管本书反映的是截止到20世纪90年代初期的研究成果，但它精辟的理论分析和细致的实现描述却有力地影响着后来的编译器的设计与实现。本书译者曾在某国际著名软件公司的商用编译器所生成的目标代码中看到了本书所介绍的优化编译技术的踪影。对于这本内容丰富、理论与实践交融的专著，我们有理由相信它会成为对编译器感兴趣并有着不同需求的读者们的良师益友。

本书中文版的翻译工作由郑启龙和姚震完成。其中：姚震负责完成前言、第1章～第7章、附录A～附录F的翻译工作，郑启龙负责第8章～第17章的翻译工作并校对了全部译稿。

本书是两位译者第一次翻译国外计算机专业著作，由于水平有限，加之时间仓促，其中难免有这样或那样的错误与不当之处，恳请各位读者批评指正。

郑启龙　姚震

2005年4月于

中国科学技术大学

前　言

本书以作者实现编译器和开发编译器构造课程的经验为基础，介绍了编译器构造的实际方法。其宗旨是使读者不仅能够对编译器的所有组件有深入的理解，而且能够对编译器的各组件如何实际组合、构成一个可以工作的编译器有感性认识。我们相信这种理念是本书一个与众不同的特色。因为我们专注于对现代编译器构造技术的介绍，所以强调尽可能地使用编译器工具生成编译器的组件。（附录B～F中所述工具的源代码，可以从出版社网站下载。）

本书和*Crafting a Compiler*一书基本相同，只是其中的算法和伪代码示例使用C而不是Ada语法。由于C语言在编译器课程中广泛使用，因此许多教师认为这样的版本会很有价值。为使所有伪代码即使对于那些不是C语言专家的读者也尽可能易读，我们使用了一些标准C语法的扩展（在下面描述），而且并不总是使用通用的C语言惯用法。由于本书作者均不是熟练的C程序员，因此我们感谢Arnold Robbins对本书做了从Ada语言到C语言的转换工作，并给出很多编辑上的建议。

教科书和参考书

作为教学用书，本书主要面向近15年来我们开发的一种课程组织结构。但是，本书的使用可以非常灵活，已经用于从为期10周的3学分高年级本科生课程到为期3个月的6学分研究生课程教学中。本书也可作为一本有价值的专业参考书，因为它完整覆盖了对编译器编写者和设计者有着重要实际意义的技术。

课程设计方法

本书全面地覆盖了与构造编译器相关的理论主题。而且，一个密切相关的课程设计实现也是我们课程组织的重要组成部分。因此，本书也是面向课程设计的。附录A包含一个称为 Ada/CS的语言定义，它主要是Ada程序设计语言的一个重要子集。但出于教学目的，这里介绍的Ada/CS并非Ada程序设计语言的一个严格子集。针对使用本书的课程，我们建议的课程设计可以涉及部分或完整的Ada/CS实现，具体情况要根据课程的长度、层次以及授课教师的要求来定。

本书第2章介绍并讨论一个针对非常简单的Ada/CS子集的递归下降编译器。将上述课程设计实施为该编译器的扩充，可以让学生即使在短到一个学期的课程中也能完成一个较大的课程设计。在时间充裕的情况下，这种扩充方法同样有价值。要求学生阅读并扩充一个实际的程序，可使他们获得在许多计算机科学课程中难以得到的重要经验。这也教会他们如何将编译器的各部分组合起来，而这种知识是难以用其他任何方式来讲授的。

C程序设计语言

本书中的示例伪代码采用基于ANSI C的语法编写。从PC机到大型机直到超级计算机，C语言在许多计算环境中都是流行的语言，它小巧且富于表达力、高效，同时通常具有很好的可移植性。C语言最近已成为美国国家标准（ANSI 1989）。支持标准C语法的编译器也已广泛可用，而且将会更加普及。因此，我们选择为示例程序使用ANSI语法而不是在Kernighan and Ritchie（1978）中描述的更广为人知的语法。而Kernighan和Ritchie的新书（1988）则是ANSI C的优秀参考书。

为了将算法尽可能以最易读的方式描述（而不关注语法细节），我们以几种方式扩充了C语言。首先，在正文的几个地方使用了*匿名联合*（anonymous union），该特性借用自C++（Stroustrup 1986）。一个匿名联合看起来像这样：

```
struct somestruct {
    int elem1;
    union {
        float  f2;
        int    i2;
        double d2;
        long       l2;
    };
    long   elem3;
} s;
. . .
s.elem1 = 10;
s.f2 = 10.0
. . .
```

注意：该结构的**union**成员没有名字，联合中的元素作为结构的元素被直接引用。在传统的C语言中，同样的行为通常通过宏预处理器获得：

```
struct somestruct {
    int elem1;
    union {
        float  u_f2;
        int    u_i2;
        double u_d2;
        long       u_l2;
    } u;
    long   elem3;
} s;
#define f2 u.u_f2
#define i2 u.u_i2
#define d2 u.u_d2
#define l2 u.u_l2
. . .
s.elem1 = 10;
s.f2 = 10.0
. . .
```

其次，从第10章开始，使用*匿名结构*（anonymous structure）作为匿名联合的成员。实际编程中，这些结构需要通过上面描述的预处理器方案实现。

最后，在许多高层伪代码示例中，使用下列*构造函数*（constructor）机制来创建结构表达式：

```
struct something {
    int elem1;
    char *elem2;
} v;
. . .
v = (something) {
        .elem1 = 1;
        .elem2 = "string";
    };
```

尽管这种结构不能使用宏来代替，但其含义显而易见。

一般情况下，在使用高层伪代码而不是正规C语言时，我们将这些图标记为“算法”而不是“程序”。

与*Crafting a Compiler*一样，在使用本书的课程中不必使用任何特定的语言来实现课程设计。不论选择哪种实现语言，伪代码都是极佳的设计描述手段。此外，我们提供的词法分析器和语法分析器生成工具生成的是表格而不是程序，因此它们可用于任何语言环境中。为了那些使用C语言实现课程设计的读者，我们也讨论Lex和Yacc的使用。

Ada的角色

我们所建议的课程设计和对语言特性的讨论都基于Ada语言，因为它事实上包含了在语义分析部分

我们希望讨论的所有语言特性。假如选择任何其他语言作为基础（例如Modular-2），就必须描述很多扩充以讨论编译这些东西（比如exit语句、异常处理和操作符重载）的技术。

使用本书的学生当然不必熟悉Ada语言。附录A中的Ada/CS介绍可以作为在语义分析部分所讨论的Ada语言特性的教程。在适当的情况下，我们也考虑从其他语言（包括Modula-2、Pascal、C、ALGOL-60、ALGOL-68以及Simula）中抽取的语言特性的翻译。

各章描述

本书作为入门课程，可以讲授第1～4章、第5章或第6章、第7章以及第8～13章和第15章的部分内容。

本书作为高级课程，可以包含讲述语法分析的各章（第5～6章）的内容，以及第8～13章和第15章加上第14、16和17章的高级主题部分。

第1章 绪论

在本书的一开始概述编译过程，强调从一组组件构造编译器的概念，介绍使用工具生成其中一些组件的思想。

第2章 一个简单编译器

介绍一个非常简单的语言Micro，讨论编译Micro的编译器的每个组件。本章包含Micro编译器的部分文本（用Ada语言编写）。而对更全面的Ada子集的语言特性的编译则引出了以下各章所介绍的技术。

第3章 词法分析——理论与实践

介绍构造编译器词法分析组件的基本概念和技术。本章中的讨论既包括开发手写的词法分析器，又包括使用词法分析器生成工具实现表驱动的词法分析器。

第4章 文法和语法分析

介绍形式语言概念和文法的基本知识，包括上下文无关文法、BNF表示法、推导和语法分析树。由于在自顶向下和自底向上的语法分析技术的定义中都用到First和Follow集，本章也对它们进行了定义。本章还包括对语言和文法关系的讨论。

第5章 LL(1)文法及分析器

介绍语法分析的第一种方法——自顶向下的语法分析。讨论递归下降和LL(1)，重点放在后者。语法分析器生成器的使用是本章的重点。

第6章 LR分析

介绍另一种语法分析方法——自底向上的语法分析。介绍LR、SLR和LALR语法分析概念及其与LL技术的比较。语法分析器生成器的使用也是本章的重点。

第7章 语义处理

本章结合自顶向下和自底向上语法分析器来介绍语义处理的基本原理。主题包括：不同编译器组织方式的比较，（在自顶向下的语法分析中）向文法增加动作符号，（在自底向上的语法分析中）为“语义钩子”（semantic hook）重写文法，定义语义记录和使用语义栈，检查语义正确性，产生中间代码。

第8章 符号表

本章强调符号表的使用，它作为一个抽象组件由编译器的其他组件通过精确定义的接口使用。本章介绍其可能的实现，随后讨论用符号表处理嵌套的作用域和其他用来定义可从外围作用域（例如记录和

Ada中的包）访问的名字的语言特性。

第9章 运行时存储组织

介绍运行时存储管理的基本技术，包括静态分配、基于栈的分配和一般动态（堆）分配的讨论。

第10章 处理声明

讨论处理类型、变量和常量声明的基本技术。这些内容的组织基于处理特定语言特性的语义例程。

第11章 处理表达式和数据结构引用

概述处理变量引用和算术、布尔表达式的语义例程。在对变量引用的讨论中，包括针对数组元素和记录域的地址计算方法。在本章及随后两章中，强调检查语义正确性和产生被目标代码生成器使用的中间代码的技术。

第12章 翻译控制结构

本章的重点是针对像**if**语句、**case**语句和各种循环结构等特性的编译技术。强调使用语义栈或语法树简化这些结构的处理。这些结构可以嵌套并扩展到任意规模的程序文本中。学生应通过特定的方法来了解这种通用技术的优点。

第13章 翻译过程和函数

介绍处理子程序声明和调用的技术。由于这个主题的很多复杂性涉及参数，本章提供了很多材料来讲述创建参数描述、检查子程序调用中实际参数的正确性以及不同参数模式所需的代码生成技术。这里讨论运行时活动栈的概念，给出实现它所必需的支持例程。

第14章 属性文法和多遍翻译

多遍翻译由遍历中间代码形式模拟。本章强调信息流的属性模型。

第15章 代码生成和局部代码优化

介绍代码生成器，它作为一个独立组件，将语义例程生成的中间代码翻译为编译器最终的目标代码。介绍像指令选择、寄存器管理和寻址模式的使用等主题。本章还包括对基本块优化的讨论。

第16章 全局优化

本章的焦点是那些通过适度的努力可以生成有用代码改进的实用技术。因此，本章的主要部分包括全局数据流分析、优化子程序调用和优化循环。

第17章 现实世界中的语法分析

本章包括实现一个实际编译器必需的两个主要课题：语法错误处理和表压缩。错误处理部分介绍用于递归下降、LL和LR分析器的错误恢复和错误修复技术。表压缩技术用于LL和LR分析器，以及词法分析器表和任何其他需要用快速访问稀疏表中的元素来高效存储的场合。

致谢

很多人为*Crafting a Compiler*和本书作出贡献。首先，感谢威斯康星大学麦迪逊分校CS 701/2和乔治亚理工学院ICS 4410中的许多学生，他们使用了本书最初版本和最终成为本书部分章节的讲义。此外，还要感谢两所学校里使用我们的讲义授课的许多教师。其中包括Raphael Finkel、Marvin Solomon和K. N. King，他们为我们的讲义贡献了部分材料；还有Nancy Lynch、Martin McKendry、Nancy Griffeth和David Pitts，他们也使用了我们的讲义。乔治亚理工学院的Arnold Robbins提供了正文中出现的所有C语言代码。他还提供了一些章节的习题、封面设计的想法以及许多对我们写作风格很有用的建议。G A

Venkatesh、Will Winsborough和Felix Wu提供了大部分习题的参考答案。Jon Mauney、Gary Sevitsky、Robert Gray和Felix Wu开发了附录中描述的编译器工具。Kathy Schultz出色地完成了手稿的最终订正、Sheryl Pomraning出色地完成了美工工作。

我们感谢C. Wrandel Barth、Jean Gallier、James Harp、Harry Lewis、Eric Roberts和Henry Shapiro，他们对我们最初的提议和样章提供了有价值的反馈。我们非常感激Steve Allan、Henry Bauer、Roger Eggen、Norman Hutchinson、Sathis Menon、Jim Bitner、Charles Shipley、Donald K. Friesen、Donald Cooley、Susan Graham、Steve Zeigler，尤其是Paul Hilfinger和Alan Wendt——他们作为审阅者提供了大量意见，使我们在很长时间内忙于完成本书及其早期版本。

特别感谢Alan Apt,我们杰出的编辑，感谢他的耐心和鼓励。同时感谢Benjamin/Cummings的所有员工，他们如此热情地支持我们的工作；感谢Todd Proebsting和Chris Sabooni，他们细心地检查我们的手稿；感谢Alyssa Weiner以及工作组的其他成员的细致工作；感谢Jane Rundell和Impressions的所有员工。

最后，感谢Miriam Robbins，她在Arnold努力将大量的Ada算法转换为清晰而精确的C算法时表现出了极大的耐心。

目　录

出版者的话
专家指导委员会
译者序
前言

第1章　绪论 ……1
1.1　概述和历史 ……1
1.2　编译器可以做什么 ……2
1.3　编译器结构 ……5
1.4　程序设计语言的语法和语义 ……8
1.5　编译器设计与程序设计语言设计 ……10
1.6　编译器分类 ……11
1.7　影响编译器设计的因素 ……12
练习 ……13
第2章　一个简单编译器 ……15
2.1　Micro编译器结构 ……15
2.2　Micro词法分析器 ……15
2.3　Micro 语法 ……19
2.4　递归下降语法分析 ……21
2.5　翻译 Micro ……25
2.5.1　目标语言 ……25
2.5.2　临时变量 ……25
2.5.3　动作符号 ……25
2.5.4　语义信息 ……25
2.5.5　Micro动作符号 ……26
练习 ……31
第3章　词法分析——理论和实践 ……33
3.1　概述 ……33
3.2　正则表达式 ……34
3.3　有限自动机和词法分析器 ……35
3.4　使用词法分析器生成器 ……38
3.4.1　ScanGen ……38
3.4.2　Lex ……43
3.5　实现时考虑的问题 ……45
3.5.1　保留字 ……45
3.5.2　编译器指示与源程序行列表 ……47
3.5.3　符号表中的标识符条目 ……47
3.5.4　词法分析器的终止 ……48
3.5.5　多字符的超前搜索 ……48
3.5.6　词法错误恢复 ……49
3.6　将正则表达式转换为有限自动机 ……51
3.6.1　构造确定的有限自动机 ……52
3.6.2　优化有限自动机 ……54
练习 ……55
第4章　文法和语法分析 ……59
4.1　上下文无关文法：概念与记号 ……59
4.2　上下文无关文法中的错误 ……61
4.3　转换扩展 BNF文法 ……62
4.4　语法分析器与识别器 ……63
4.5　文法分析算法 ……64
练习 ……69
第5章　LL(1)文法及分析器 ……71
5.1　LL(1) Predict函数 ……71
5.2　LL(1)分析表 ……73
5.3　从LL(1)分析表构造递归下降分析器 ……74
5.4　LL(1)分析器驱动程序 ……77
5.5　LL(1)动作符号 ……77
5.6　文法的LL(1)化 ……78
5.7　LL(1)分析中的If-Then-Else问题 ……81
5.8　LLGen—LL(1)语法分析器生成器 ……82
5.9　LL(1)分析器的性质 ……85
5.10　LL(k)分析 ……85
练习 ……87
第6章　LR分析 ……89
6.1　移进-归约分析器 ……89
6.2　LR分析器 ……91
6.2.1　LR(0)分析 ……92

6.2.2 如何判定LR(0)分析程序工作的正确性 ……96
6.3 LR(1)分析 ……98
6.3.1 LR(1)分析的正确性 ……101
6.4 SLR(1)分析 ……102
6.4.1 SLR(1)分析的正确性 ……103
6.4.2 SLR(1)技术的局限性 ……104
6.5 LALR(1)分析 ……105
6.5.1 构造LALR(1)分析器 ……108
6.5.2 LALR(1)分析的正确性 ……112
6.6 在移进-归约分析器中调用语义例程 …113
6.7 使用语法分析器生成器 ……114
6.7.1 LALRGen语法分析器生成器 ……114
6.7.2 Yacc ……116
6.7.3 可控二义性的使用和误用 ……117
6.8 优化分析表 ……120
6.9 实用的LR(1)分析器 ……123
6.10 LR分析的性质 ……125
6.11 LL(1)和LALR(1)分析方法的比较 ……125
6.12 其他的移进-归约技术 ……128
6.12.1 扩展的超前搜索技术 ……128
6.12.2 优先级技术 ……128
6.12.3 一般的上下文无关分析器 ……130
练习 ……132
第7章 语义处理 ……137
7.1 语法制导翻译 ……137
7.1.1 使用分析的语法树表示 ……137
7.1.2 编译器组织的候选形式 ……138
7.1.3 一遍编译中的分析、检查和翻译 …142
7.2 语义处理技术 ……143
7.2.1 LL分析器和动作符号 ……143
7.2.2 LR分析器和动作符号 ……143
7.2.3 语义记录表示 ……144
7.2.4 实现动作控制的语义栈 ……146
7.2.5 分析器控制的语义栈 ……149
7.3 中间表示和代码生成 ……155
7.3.1 比较中间表示和直接代码生成 ……155
7.3.2 中间表示的形式 ……155
7.3.3 一个元组语言 ……157
练习 ……157
第8章 符号表 ……161
8.1 符号表接口 ……161
8.2 基本实现技术 ……162
8.2.1 二叉搜索树 ……162
8.2.2 哈希表 ……163
8.2.3 串空间数组 ……164
8.3 块结构符号表 ……165
8.4 块结构符号表的扩展 ……169
8.4.1 域和记录 ……169
8.4.2 导出规则 ……170
8.4.3 导入规则 ……174
8.4.4 可更改的搜索规则 ……175
8.5 隐式声明 ……177
8.6 重载 ……177
8.7 前向引用 ……178
8.8 小结 ……180
练习 ……180
第9章 运行时存储组织 ……183
9.1 静态分配 ……183
9.2 栈分配 ……183
9.2.1 显示表 ……185
9.2.2 块级与过程级活动记录 ……187
9.3 堆分配 ……188
9.3.1 无空间释放 ……188
9.3.2 显式释放 ……188
9.3.3 隐式释放 ……189
9.3.4 管理堆空间 ……190
9.4 内存中的程序布局 ……191
9.5 静态链簇和动态链簇 ……193
9.6 形式过程 ……195
9.6.1 静态链簇 ……196
9.6.2 显示表 ……197
9.6.3 一些看法 ……198
练习 ……199
第10章 处理声明 ……203
10.1 声明处理的基本原则 ……203
10.1.1 符号表中的属性 ……203
10.1.2 类型描述符结构 ……204
10.1.3 语义栈中的列表结构 ……205

10.2 简单声明的动作例程 ……207
10.2.1 变量声明 ……207
10.2.2 类型定义、声明和引用 ……209
10.2.3 记录类型 ……212
10.2.4 静态数组 ……214
10.3 高级特性的动作例程 ……216
10.3.1 变量和常量声明 ……216
10.3.2 枚举类型 ……218
10.3.3 子类型 ……220
10.3.4 数组类型 ……221
10.3.5 变体记录 ……227
10.3.6 访问类型 ……232
10.3.7 包 ……233
10.3.8 attributes和semantics_record结构 ……236
练习 ……239
第11章 处理表达式和数据结构引用 ……241
11.1 概述 ……241
11.2 简单名字、表达式和数据结构的动作例程 ……242
11.2.1 处理简单标识符和文字常量 ……242
11.2.2 处理表达式 ……243
11.2.3 简单的记录和数组引用 ……246
11.2.4 记录和数组示例 ……249
11.2.5 串 ……250
11.3 高级特性的动作例程 ……250
11.3.1 多维数组的组织和引用 ……250
11.3.2 含动态对象的记录 ……258
11.3.3 变体记录 ……261
11.3.4 访问类型的引用 ……262
11.3.5 Ada中其他名字的使用 ……263
11.3.6 记录和数组聚合 ……265
11.3.7 重载解析 ……266
练习 ……269
第12章 翻译控制结构 ……271
12.1 if语句 ……271
12.2 循环 ……274
12.2.1 while循环 ……274
12.2.2 for循环 ……275
12.3 编译exit语句 ……280
12.4 case语句 ……282
12.5 编译goto语句 ……287
12.6 异常处理 ……290
12.7 短路计算布尔表达式 ……294
12.7.1 单地址短路计算 ……299
练习 ……305
第13章 翻译过程和函数 ……309
13.1 简单子程序 ……309
13.1.1 声明无参子程序 ……309
13.1.2 调用无参过程 ……311
13.2 向子程序传递参数 ……312
13.2.1 值、结果和值-结果参数 ……313
13.2.2 引用和只读参数 ……314
13.2.3 处理参数声明的语义例程 ……314
13.3 处理子程序调用和参数表 ……316
13.4 子程序调用 ……317
13.4.1 保存和恢复寄存器 ……317
13.4.2 子程序的入口和出口 ……318
13.5 标号参数 ……321
13.6 名字参数 ……323
练习 ……324
第14章 属性文法和多遍翻译 ……327
14.1 属性文法 ……327
14.1.1 简单赋值形式和动作符号 ……329
14.1.2 树遍历的属性计算程序 ……331
14.1.3 直接属性计算程序 ……335
14.1.4 属性文法示例 ……340
14.2 树结构的中间表示 ……342
14.2.1 抽象语法树接口 ……343
14.2.2 语法树抽象接口 ……344
14.2.3 实现树 ……348
练习 ……349
第15章 代码生成和局部代码优化 ……351
15.1 概述 ……351
15.2 寄存器和临时变量管理 ……352
15.2.1 临时变量的分类 ……353
15.2.2 分配和释放临时变量 ……353
15.3 简单的代码生成器 ……354
15.4 解释性代码生成 ……356

15.4.1 优化地址计算 ……357
15.4.2 避免冗余计算 ……359
15.4.3 寄存器追踪 ……361
15.5 窥孔优化 ……367
15.6 从树结构生成代码 ……368
15.7 从dag生成代码 ……371
15.7.1 别名 ……376
15.8 代码生成器的生成器 ……377
15.8.1 基于文法的代码生成器 ……380
15.8.2 在代码生成器中使用语义属性 …382
15.8.3 生成窥孔优化器 ……385
15.8.4 基于树重写的代码生成器的生成器 ……387
练习 ……387
第16章 全局优化 ……393
16.1 概述——目标与限制 ……393
16.1.1 理想的优化编译器结构 ……394
16.1.2 优化展望 ……397
16.2 优化子程序调用 ……397
16.2.1 子程序调用的内联展开 ……397
16.2.2 优化对封闭子例程的调用 ……399
16.2.3 过程间数据流分析 ……402
16.3 循环优化 ……406
16.3.1 外提循环不变式 ……406
16.3.2 循环中强度削弱 ……409
16.4 全局数据流分析 ……411
16.4.1 单路径流分析 ……411
16.4.2 全路径流分析 ……415
16.4.3 数据流问题的分类 ……416
16.4.4 其他重要的数据流问题 ……416
16.4.5 使用数据流信息的全局优化 ……418
16.4.6 求解数据流方程 ……422
16.5 集成优化技术 ……430
练习 ……431
第17章 现实世界中的语法分析 ……437
17.1 压缩表 ……437
17.1.1 压缩LL(1)分析表 ……439
17.2 语法错误的恢复与修复 ……440
17.2.1 即时错误检测 ……442
17.2.2 递归下降分析器中的错误恢复 …443
17.2.3 LL(1)分析器中的错误恢复 ……445
17.2.4 FMQ LL(1)错误修复算法 ……446
17.2.5 在FMQ修复算法中添加删除操作…449
17.2.6 FMQ算法的扩展 ……450
17.2.7 利用LLGen进行错误修复 ……454
17.2.8 LR错误恢复 ……455
17.2.9 Yacc中的错误恢复 ……455
17.2.10 自动生成的LR修复技术 ……456
17.2.11 利用LALRGen进行错误修复 ……462
17.2.12 其他LR错误修复技术 ……462
练习 ……463
附录A Ada/CS语言定义 ……467
附录B ScanGen ……487
附录C LLGen用户手册 ……493
附录D LALRGen用户手册 ……499
附录E LLGen和LALRGen错误修复特性 …507
附录F 编译器开发实用工具 ……511
参考文献 ……517
索引 ……523

第1章 绪 论

1.1 概述和历史

编译器是现代计算技术的基本组成部分。它们担任*翻译器*（translator）的角色，将面向人的*程序设计语言*（programming language）转换成面向计算机的*机器语言*（machine language）。对于大多数用户来说，编译器可以被看作是一个“黑箱”，执行图1-1所示的转换。

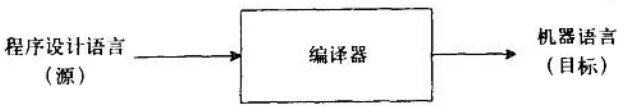


图1-1 用户眼中的编译器

编译器事实上允许所有计算机用户忽略与机器相关的机器语言细节。因此，编译器允许程序和程序设计的专业技术成为*与机器无关的*（machine-independent）。在计算机的数量和种类持续爆炸性增长的时代，这是一项特别有价值的优点。

术语*编译器*（compiler）是在20世纪50年代早期由Grace Murray Hopper创造。那时翻译被看作是“编译”一系列从程序库中选出的子程序。实际上，编译（正如我们现在所知道的）那时就被称为“自动程序设计”，而对它究竟成功与否却存在着普遍的怀疑。今天，程序设计语言的自动翻译已经是一个既成事实，但程序设计语言翻译器仍被称为编译器。

现代意义上的最早的真正编译器是20世纪50年代晚期的FORTRAN编译器。它们为用户提供了一个面向问题的、与机器密切相关的源语言，并执行一些相当有挑战性的“优化”以产生高效的机器代码。而这种优化对于成功地和当时占统治地位的汇编语言进行竞争来说是必要的。这些FORTRAN系统证明了经过编译的高级（即较少依赖机器的）语言的生命力，并为随后语言和编译器的大量涌现铺平了道路。

第一个FORTRAN编译器花费了18个人年来构造。早期的编译器都是专用结构；组件和技术通常随着编译器的构造而被发明出来。构造编译器是一项复杂和代价高昂的工作。今天，编译技术已被很好地理解，一个简单的编译器可以在几个月中构造出来。尽管如此，构造高效可靠的编译器仍然是一项富有挑战性的任务。我们的方法是掌握编译的基本原理，随后研究高级主题以及许多新近发明的重要的技术革新。

通常认为编译器是将程序设计语言翻译为机器语言指令。然而，编译器技术是广泛适用的并已经用于许多过去未曾预料到的领域。例如，文本格式化语言已经变得日益复杂。格式化程序（formatter），像UNIX®的*nroff* 和*troff*，实际上就是编译器，它们将文本和格式化命令翻译为非常复杂的排版命令。像所有成功的语言一样，*nroff* 和*troff* 已被推广，用来提供新的功能。例如，像*eqn*（用来处理公式）、*tbl*（用来格式化表格）和*pic*（用来绘制图形）这样的预处理器包非常类似于通常用来扩展现有程序设计语言的预处理器（例如，Ratfor FORTRAN预处理器）。程序设计语言和编译器设计的持续挑战之一就是发明允许用简单且自然的方式扩展现有语言的机制。

创建VLSI电路是另一项可以被建模为从高级源语言到低级目标语言的翻译任务。在此情形中，*硅*

编译器（silicon compiler）指定VLSI电路掩膜的布局和组成。就像普通的编译器必须理解并强制遵守特殊的机器语言规则一样，硅编译器必须理解并强制遵守给定电路的可行性设计规则。

编译器技术几乎在任何提供非平凡命令集的程序（包括操作系统的命令语言和数据库系统的查询语言）中都是有价值的。尽管我们的讨论将集中于传统的编译任务，但有创新精神的读者无疑会为这里所
2 介绍的技术找到新的未曾预料到的应用。

1.2 编译器可以做什么

图1-1显示一个编译器，它作为一个翻译器将被编译的程序设计语言翻译为某种机器语言。该描述暗示所有编译器做着相当多的同样事情，惟一的区别是它们对源语言和目标语言的选择。然而，实际情况比这更复杂。接受源语言的问题事实上比较简单，但编译器的输出却可用许多其他的方法加以描述，这些方法远非像命名在其上产生输出的特定计算机那样简单。

编译器可以根据它们试图生成的目标代码类型来分类：

(1) 纯机器代码（Pure Machine Code）

首先，编译器可以为特定机器的指令集生成代码，而不假定存在任何操作系统或库例程。这种机器代码通常被称为纯代码，因为它仅包含指令集中的指令。这种方式较少采用；它大多用于系统实现语言的编译器中，这些语言用来实现操作系统和其他像这样的低级软件。这种形式的目标代码可以不依赖于任何其他软件而直接在硬件上执行。

(2) 拓广的机器代码（Augmented Machine Code）

其次，更通常的情况是，编译器为由操作系统例程和语言支撑例程所拓广的机器体系结构生成机器代码。为执行一个由这样的编译器生成的程序，在目标机器上必须有特定的操作系统，而且必须与程序一起加载一组语言特定的支撑例程（输入/输出、存储分配、数学函数等）。目标机器的指令集与这些操作系统和语言支撑例程的组合被认为是定义了一个虚拟机（virtual machine）——即，另一台仅作为硬件/软件组合而存在的机器。

虚拟机与实际的机器相匹配的程度可能变化很大。某些一般的编译器将源代码几乎完全翻译为硬件指令（例如，大多数FORTRAN编译器仅对输入/输出和数学函数使用软件支持）。其他编译器则采用大量的虚拟指令。这些虚拟指令可能包括：数据传输指令（例如，用来移动位域）、过程调用指令（用来传递参数、保存寄存器、分配栈空间等）以及动态存储分配指令。

(3) 虚拟机器代码（Virtual Machine Code）

最后，虚拟机定义的极端情况代表最终的目标机器选择。生成的代码完全由虚拟指令组成。该方法作为产生能够容易地运行在多种计算机上的可移植编译器的技术，尤其具有吸引力。这种方法能够增强
3 可移植性，因为移植编译器（如果它能够自举（bootstrap）——编译它自身——或者以一种可用的语言写成）仅需要为该编译器所使用的虚拟机编写模拟器。如果该虚拟机保持简洁，则解释器将非常容易编写。该方法的一个著名例子是Pascal编译器，最初由Nicklaus Wirth所领导的小组编写。该编译器生成的代码称为P-代码（P-code），适合于一个虚拟栈机器。一个合乎标准的P-机器（P-machine）可由一个程序员在几周内完成。P-代码的执行速度约比经过编译的代码慢3倍。另一种方法是，通过更多的工作，在最终编译阶段可将P-代码翻译为等价的机器代码，或者修改编译器以直接生成机器代码。该方法使得Pascal很容易在几乎任何机器上都可用。因此，该编译器是促使Pascal高度普及的一个重要因素。

从前面的讨论可见，虚拟指令可以达到很多目的。它们通过利用适合于所翻译的特殊语言的原语（例如过程调用、字符串操纵等）来简化编译器的工作。它们为编译器提供了可移植性，而且可以允许所生成代码量的显著减少成为现实。也就是说，指令可被设计用来很好地满足特殊程序设计语言的需要

(例如，Pascal的P-代码)。据报告，使用这种方法可以将生成的程序大小缩减2/3 (Tanenbaum 1974)。

当一个完整的虚拟指令集被用于目标机器语言时，必须用软件来解释（模拟）该指令集。与由硬件执行的目标指令相比，使用一个良好实现的模拟器，可能有3:1到10:1的速度下降。生成需要这样执行的代码时，编译器（像Berkeley Pascal *pi*处理程序）有时也被称为*解释器* (interpreter)，尽管我们更喜欢使用这个术语来指那些不明确地分离翻译和执行的语言处理程序。解释器将在本节中稍后讨论。

某些计算机的指令集可以通过*微程序设计* (microprogramming) 来修改。微程序设计是解释虚拟指令集的理想机制，因为它使高速执行成为可能。当然，一旦使用了微程序设计，虚拟指令集也就成为实际机器的指令集，而不再是虚拟的!

总之，几乎所有编译器都或多或少地为虚拟机生成代码，其中一些虚拟机的操作必须由软件或固件 (firmware) 来解释。我们把它们也看作是编译器，因为它们在执行之前使用了独立的翻译阶段。

编译器之间相互区别的另一个方面是它们生成的目标机器代码的格式。目标机器格式可分为汇编语言、可重定位的二进制代码或者内存映像。

(1) *汇编语言（符号）格式* (Assembly Language (Symbolic) Format)

编译器编写者可能选择生成汇编代码的主要原因是简化（及模块化）翻译。即，许多代码生成决定(跳转目标、长/短地址格式，等等）可以留给汇编器。这种方法在UNIX编译器中很常见。在小的计算机中——UNIX最初即为这些计算机设计——这种模块化可能是强制的，因为完整的编译器通常无法放 4
在可用的内存中。生成汇编代码对于*交叉编译* (cross-compilation)（在一台计算机上运行编译器，而其目标语言是另一台计算机的机器语言）通常也是有用的；产生的符号格式易于在不同计算机之间进行传输。该方法也使得检查编译器的正确性变得更容易，因为我们可以观察它的输出。

尽管有这些优点，但是生成符号代码在现代编译器中并不常见且一般不被推荐。它的主要缺点是输出的代码必须在编译器之后由一个汇编器处理。汇编器通常很慢，且程序员不得不使用一个额外的步骤来准备程序的执行。应当注意的是，UNIX汇编器被专门设计用来处理编译器的输出；因此它们缺少大多数编译器所共有的许多特性，因而较快。UNIX编译器驱动程序也自动调用汇编器，因此它的使用对于程序员是透明的。然而，如果一个编译器不生成汇编语言，编译器编写者仍旧需要一种方法来检查所生成代码的正确性。在这种情况下，一个好办法是这样来设计：编译器可选地生成*伪汇编语言* (pseudo-assembly language)，即一个列表。如果汇编语言被生成的话，它看起来就像该列表这样。

(2) *可重定位的二进制代码格式* (Relocatable Binary Format)

作为第二种可选的方法，目标代码可以用一种二进制格式生成，其中外部引用和局部指令以及数据地址尚未绑定。相反地，地址赋值为相对于模块开始位置或相对于某些以符号命名的位置的偏移。(后一种方法易于集中代码序列或数据区。）这种形式是汇编器的典型输出，因此这种方法简单地消除了准备程序执行过程中的一个步骤。需要一个链接步骤以便添加任何支撑库以及从所编译的程序中引用的其他预编译例程，并产生可执行的绝对二进制程序格式。

可重定位的二进制代码和汇编语言格式两者都允许模块化编译（将一个大程序分成可单独编译的片段)、交叉语言引用（调用汇编语言例程或用其他高级语言编写的子程序）以及支持例程库（例如，输入/输出、存储分配以及数学例程)。

(3) *内存映像（加载和运行）格式* (Memory-Image (Load-and-Go) Format)

还有一种方法是经过编译的输出可以被加载到编译器的地址空间中，并立即执行（而不是像前两种方法那样放在一个文件中)。该过程通常比经过相对较慢且较昂贵的链接/编辑中间步骤更快。它也允许程序在一个单独的步骤中进行准备和执行。然而，与外部引用、库以及预编译例程的接口非常有限，而且对于每次执行，程序必须被重新编译，除非提供某些手段来存储内存映像。加载和运行式的编译器对于学生和调试用途非常有用，在这些用途中，频繁的更改是家常便饭，而编译代价远远超过执行代价。 5

当我们不希望创建并保存程序时（例如，为节省文件空间或保证仅使用最近的库和系统例程），该方法也很有用。加载和运行式编译器的一个有趣应用是ASAP信息检索系统（Conway 1972）。在该系统中，数据以加密格式保存，因此只能通过查询语句进行访问。编译器对查询语句进行翻译，并在其代码中添加解密例程。编译器静态地检查用户的访问权限，拒绝编译访问被禁止数据的查询。如果经过编译的程序能够被保存，则不可能更改或撤消访问权限。

以上讨论的这些代码格式选择和目标机器选择说明：编译器互相之间可以极其不同，而仍然执行相同类型的翻译工作。另一类语言处理器，称为*解释器*（interpreter），它与编译器不同，因为它执行程序而不明确进行翻译。图1-2示意性地说明了解释器如何工作。

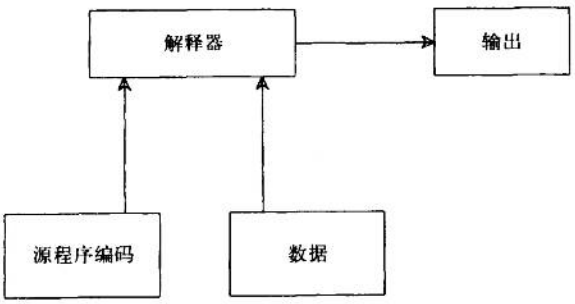


图1-2 理想化的解释器

和编译器形成对比，可以考虑解释器的下列特性。对于解释器，程序只是可被任意操纵的数据，就像任何其他的数据一样。执行过程中的控制点在解释器中而不在用户程序中（即，用户程序是被动而不是主动的）。

解释器允许：

- 在执行中可以对用户程序进行修改或向用户程序添加代码。该机制提供了直接的交互式调试能力。这种修改在像APL或BASIC这样的非块结构的语言中最为容易，因为可以修改个别语句而不需要重新分析整个程序。
- 对象所代表的类型可以动态改变的语言。用户程序在执行过程中不断地被重新检查，而符号不必有固定的意义（即，某个符号可以在某一点代表一个整数，而在后面的另一点代表一个布尔型数组）。这样的*可变绑定*（fluid binding）对于编译器显然更麻烦，因为一个符号意义的动态改变使得不可能直接把程序转换为机器代码。
- 更好的错误诊断信息。因为源文本分析（通常在编译时完成）与程序的执行混合在一起，所以比起编译器来，解释器更容易产生特别好的诊断信息（出错源代码行的重建、错误信息中变量名的使用，等等）。
- 极大程度的机器无关性，因为不生成机器代码。所有操作都在解释器中执行。因此，为移植一个解释器，只需要在一台新机器上重新编译它。

然而，解释也会包含很大的开销：

- 在执行过程中，程序文本必须不断地被重新检查，标识符绑定、类型和操作都在每次引用时潜在地被重新考虑。对于非常动态化的语言，这将导致在执行速度上100:1（或者更坏）的因子。对于更静态化的语言（例如BASIC），速度下降接近于10:1。
- 可能牵涉巨大的空间开销。解释器和所有支撑例程必须经常保持可用。这种程序表示法通常不像经过编译的机器代码那样紧凑（例如，存在符号表，且程序文本的存储格式被设计为易于访问和修改，而不是为了空间的最小化）。这种空间损失可能导致对程序大小、变量或过程数量等的限制。

超出这些内建限制的程序不能被解释器处理。

最后，某些语言（例如，BASIC、LISP和Pascal）同时拥有解释器（用于调试和程序开发）和编译器（用于生产工作）。事实上，在编译器和解释器之间不总是存在清晰的分界线。例如，许多BASIC解释器把源程序“翻译”成内部形式，其中像**let**和**goto**这样的关键字被表示为单字节*操作码*（operation code），而标识符则类似地由内部表的引用替换。究竟是否把做这种压缩的程序称为编译器并把生成的压缩形式称为目标程序，只是一个喜好问题。我们选择不这样做，因为这种情况下所完成的“翻译”完全是语法上的。更进一步说，这种解释器对源程序进行转换所形成的内部形式对使用该解释器的程序员是不可见的，这明显不同于典型编译器输出的可见性。

总之，所有语言处理都在某种程度上涉及解释。我们称为解释器的处理程序直接解释它们所处理的源程序或是这些源程序语法上的转换版本。它们可能利用源程序表示的可用性以允许在执行和调试时改变程序文本。编译器拥有独立的翻译和执行阶段，但其中也涉及“解释”。翻译阶段可能生成由软件解释的虚拟机语言或者由特定的计算机用固件或硬件来解释的真正的机器语言。 7

1.3 编译器结构

任何编译器必须执行两个主要任务：*分析*（analysis）要编译的源程序，以及*综合*（synthesis）机器语言程序。机器语言程序在运行时将正确执行由源程序所描述的动作。几乎所有现代编译器都是*语法制导的*（syntax-directed）。即，编译过程由*语法分析器*（parser）所识别的源程序的语法结构来驱动。语法分析器由*词法记号*（token）创建结构，而词法记号是定义程序设计语言语法的最低级符号。语法结构的识别是分析任务的主要部分。*语义例程*（semantic routine）基于语法结构，实际提供程序的意义（语义）。这些例程扮演着双重角色，因为除了着手开始或完成所有的综合任务外，它们通过执行某些正确性检查（例如，类型和作用域规则）来完成分析任务。语义例程可以生成程序的某些*中间表示*（intermediate representation，IR）或直接生成目标代码。如果生成IR，则它将作为*代码生成器*（code generator）组件的输入，实际产生所要的机器语言程序。IR可以任选地由*优化器*（optimizer）进行转换，以便生成更高效的机器语言程序。图1-3中示意性地描述了编译器中这些组件的组织方式。

本节剩余部分更详细地描述了每个组件的任务。在第2章中将介绍一个简单的编译器，用来为本概述中所介绍的许多概念提供具体的示例。

(1) *词法分析器*（Scanner）

词法分析器通过逐字符地读取输入，并将字符分组为独立的单词和符号来开始对源程序的分析。这些单词和符号被称为*词法记号*（token），代表基本的程序实体，如标识符、整数、保留字、分隔符等。词法分析是接连为输入产生高级解释的若干步骤中的第一步。词法记号被编码（例如，编码为整数）并随后传递给语法分析器进行语法分析。有时组成词法记号的实际字符串（或者指向存储所有标识符文本的*字符串空间*的指针）也被一同传递以便由语义例程随后使用。

词法分析器做以下工作：

- 把程序翻译成紧凑和一致的形式（词法记号）。
- 消除不需要的信息（比如注释）。
- 处理编译器控制指令（打开或关闭列表，从一个文件中包含源程序映像，等等）。这些指令通常通过*伪注释*（pseudocomment）完成。伪注释拥有注释的语法形式，但包含打算由编译器处理的特殊信息。另一种方法用于Ada中，其中一个特殊的语法结构*pragma*，用于此目的。 8
- 把初步的信息放在符号和属性表中（例如，用来产生随后的交叉引用列表）。
- 格式化并列出源程序。

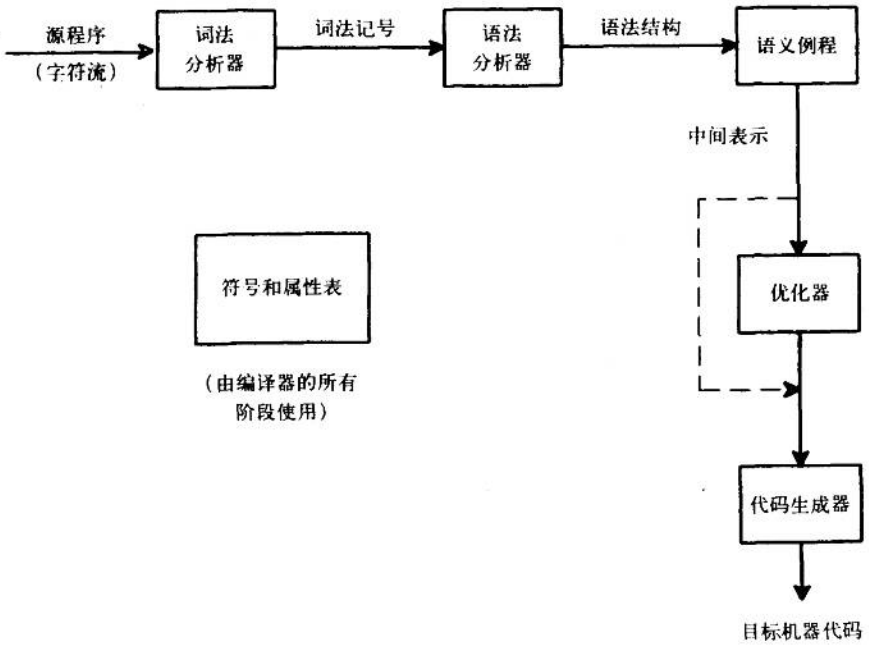


图1-3 语法制导的编译器结构

创建词法记号的主要动作通常由词法记号描述驱动。*正则表达式*（regular expression）记号（在第3章中讨论）是描述词法记号的一种有效文法。正则表达式是描述现代程序设计语言所需的各种词法记号的足够强大的正式记号。此外，它们可用作识别*正则集*（regular set）——即正则表达式所定义的集合——的*有限自动机*（finite automata）的自动生成规范。正则表达式的这个解释是*词法分析器生成器*（scanner generator）的基础。只需给出所要识别的词法记号的规范，词法分析器生成器就可以实际产生可工作的词法分析器。显然，这样一个程序是有价值的编译器构造工具。

9

为简单起见，词法分析器还可以由*手工编写*，即，显式地构造识别特殊语言词法记号的程序。使用这种方法是合理的，因为程序中可能完全包含词法分析器生成器所需的一组词法记号定义，而学习使用这样一个生成器可能比直接编写词法分析器需要更多的工作。手工编写的词法分析器直到近来才成为普遍习惯，因为传统的常识认为由生成器所产生的表驱动的词法分析器只是比手工编码的词法分析器稍慢一些。但任何额外开销都可能是至关重要的，因为词法分析可能代表编译过程的很大一部分（因为有很多的字符级处理）。例如，有报告称：编译器仅为了跳过空格就需要花费20%的时间。最近的发展已经证明表驱动的词法分析器总是比手工编写的词法分析器更快。此外，表驱动的词法分析器的优点是同样的驱动程序可以用于不同的词法分析器——仅有表格必须进行修改。因此，一旦知道如何使用词法分析器生成器工具，词法分析器的构造会非常简单。

(2) *语法分析器*（Parser）

给定一个形式语法规范（典型情况下以*上下文无关文法*（context-free grammar, CFG）的形式提供），语法分析器读入词法记号并将它们按照所使用的CFG产生式的规定分组为单元。（文法在第4章中讨论；语法分析在第5章和第6章中讨论。）语法分析器在典型情况下由表进行驱动。该表由*语法分析器生成器*（parser generator）根据CFG创建。

进行语法分析时，语法分析器验证程序语法是否正确，如果发现语法错误，则显示适当的诊断信息。它也能够修复错误（以形成语法上有效的程序）。在许多情况下，语法错误恢复或修复能够通过查阅由语法分析器生成器或修复生成器所创建的*错误修复表*（error-repair table）来自动完成。

在识别语法结构的同时，语法分析器可以直接调用相应的语义例程，或者构造*语法树*（syntax tree），即该结构的表示，用来在这棵树被完整构造后驱动语义处理。

(3) *语义例程*（Semantic Routine）

语义例程执行两种功能。首先，它们检查每个结构的*静态语义*（static semantics）。即，它们验证结构是合法的和有意义的（其中涉及的变量是已定义的，类型是正确的，等等）。如果结构是语义上正确的，语义例程也进行实际的*翻译*。即，生成正确实现该结构的IR代码。语义例程通常是手工编写的，且与单独的CFG产生式或语法树的子树相关联。

当识别了一个产生式应用时，相关的语义例程被调用以检查静态语义并执行翻译操作。因此，对于产生式E → E+T，可以编写一个语义例程来首先检查类型相容性并随后生成 IR代码来执行加法。 10

编译器的核心部分，也是它的大多数文本，都在语义例程中。语义例程定义了每个结构如何进行检查和翻译的细节。编译的这个方面可以通过*属性文法*（attribute grammar）进行形式化。属性文法通过代表像类型、值或正确性的语义属性的值（或称*属性*）来对普通CFG进行扩展。关于属性文法和其他定义语义规范的方法，会在本章稍后进行更详细的讨论。

保持语义例程的检查和翻译组件的独立通常是有用的。语义检查首先完成，它由源语言的语义规则单独控制。IR生成则受源语言语义（生成的IR代码必须正确地实现语言结构）和目标机器（IR代码的选择可能反映所期望的目标机器的能力）两者的共同影响。如果这两种组件是清晰分离的，则让一个编译器为不同的目标机器生成代码的工作就会被简化，因为仅有IR生成（的一小部分）和目标代码生成（的大部分）是与机器相关的。

(4) *优化器*（Optimizer）

由语义例程生成的IR代码被优化器分析并转换为功能上等价但经过改进的IR代码。该阶段可以*非常*复杂和缓慢；它通常包括大量的子阶段，其中一些子阶段可能应用不止一次。大多数编译器允许关闭优化以加速翻译。其他一些编译器则没有优化器，而是由语义例程生成对代码生成器的直接调用以产生目标代码。

一类不太复杂的（而且并不昂贵的）优化称为“*窥孔优化*”（peephole optimization）。这类优化通常应用于目标代码。它每次仅考虑少量指令（事实上，通过一个“窥孔”）并试图进行简单的、通常是机器相关的代码改进。例如，普通的窥孔优化包括消除与1的乘法或与0的加法，当先前的指令已经把某个值从寄存器保存到某个内存位置时，删除将该值放入寄存器的装载指令，以及把一个指令序列替换为一条有同样效果的单独的指令（例如，一个把某内存单元中的值加1的指令）。窥孔优化器没有一个全面的优化器那样的潜在收效，但它可以在非常局部的级别上极大地改进代码，并通常对于“整理”（cleanning up）由应用更复杂的优化所得到的最终代码是非常有用的。

(5) *代码生成器*（Code Generator）

IR代码由代码生成器映射为目标机器代码。这需要关于目标机器的详细信息，并通常涉及特定于机器的优化（例如寄存器分配、指令格式选择、寻址模式等）。一般的代码生成器是手工编写的，并且十分复杂，因为生成好的目标机器代码需要考虑许多特殊情况。 11

如果不需要优化，那么简化编译器结构的一个方法是将语义例程和代码生成部分合并，并消除对IR的使用。其结果是一个*一遍*（one-pass）编译器，这种结构常常用于Pascal编译器（包括UW-Pascal [Fischer and LeBlanc 1980]）。

在最近几年中，代码生成器的自动生成的思想被广泛地研究。这里的思想是自动地把一个低级IR与目标指令模板相匹配。这使编译器对目标机器的依赖局部化，而且，至少在原则上使得将一个编译器再目标（retarget）到一台新的目标机器成为可能。这是一个特别理想的目标，因为将一个编译器移植到一台新机器上通常需要做大量的工作。因此，简单地改变目标机器模板的设定并（从模块）生成新的代码生成器的可能性非常有吸引力。它激励着我们进行大量研究工作。

使用一些基于这些观点的技术的实用编译器之一是GCC，GNU C编译器（Stallman 1989）[⊖]。GCC是一个做大量优化的编译器，它含有针对10余种流行的计算机体系结构的机器描述文件，并且至少含有两种语言的前端（C和C++）。

(6) *符号和属性表*（Symbol and Attribute Table）

符号表（symbol table）是允许将信息（属性）与标识符相关联的机制。每次使用一个标识符时，符号表提供对处理标识符声明时所收集到的标识符相关信息的访问。因此，这些表可以被任何编译器组件使用，用于添加、共享以及之后检索关于变量、过程、标号等的信息。

最后，注意在讨论编译器设计和构造时，通常会谈到*编译器编写工具*（compiler writing tool）。这些工具通常被打包为*编译器生成器*（compiler generator）或*编译器的编译器*（compiler-compiler）。这样的包通常包括词法分析器和语法分析器生成器，某些包还包含符号表例程和代码生成能力。更高级的包可能支持错误修复的生成。

这些种类的生成器对于构造编译器模块有很大的帮助，但在构造编译器过程中大量的工作花费在编写和调试语义例程。这些例程数量众多，通常有上百个，而且几乎完成检查静态语义和生成IR代码（或者在一遍编译器中生成目标代码）等所有的工作。

1.4 程序设计语言的语法和语义

程序设计语言的完整定义必须包括它的*语法*（结构）和*语义*（意义）的规范。假定CFG作为对程序设计语言几乎通用的语法规范机制，语法通常可以分为*上下文无关*（context-free）和*上下文相关*
12 （context-sensitive）组件。上下文无关语法定义合法的符号序列，而不依赖于任何关于符号意义的概念。例如，上下文无关语法可能宣称A:= B + C; 是语法上合法的但A:= B +; 则不是。然而，并非所有的程序结构都能用上下文无关语法来描述。类型相容性和作用域规则是上下文相关的问题。例如，如果A:= B + C中的任何变量是未声明的或者如果B或C是布尔型变量，则A:= B + C是非法的。这种规则就不能用CFG来指定。它是一种上下文相关的约束。

因为CFG的局限性，上下文相关的约束被作为语义问题处理。因此，程序设计语言的语义组件被典型地划分为两类：*静态语义*（static semantics）和*运行时语义*（run-time semantics）。语言的静态语义定义了上下文相关的约束，在一个程序能够被认为是有效的之前，必须强制这些约束。典型的静态语义规则需要所有标识符都被声明，操作符和操作数是类型相容的，以及用适当数量的参数调用过程。贯穿所有这些约束的共同线索是：它们都不能用CFG来表达。静态语义以此来补充上下文无关规范，并完成对有效的程序看起来应该是什么样子这一问题的定义。静态语义可以被形式化地或非形式化地指定。在大多数程序设计语言手册中的文字性特征描述是非形式化的规范。它们倾向于相对简洁易读但通常是不精确的。形式化的定义可以用多种记号中的任意一种来表达。例如，属性文法是形式化地规定静态语义的普遍方法。它们对编译器中常见的语义检查进行形式化。作为属性文法的示例，对于产生式$E_1 \rightarrow E_2 + T$，可以通过E和T的类型属性以及一个需要类型相容性的谓词，如

⊖ 随着GCC的发展，它现在可以处理除C语言外的其他许多程序设计语言。GCC现在代表*GNU编译器集*（the GNU Compiler Collection）。——译者注

$(E_2.type = numeric)$ **and** $(T.type = numeric)$

来对其进行扩充。属性文法适当地兼顾了形式和可读性，但它们可能仍然很冗长。许多编译器编写系统利用属性文法并提供属性值的自动计算功能。

运行时语义或执行语义用于规定程序做什么，即计算什么。在语言手册或报告中，这些语义通常被极其非形式化地规定。作为选择，可以使用更为正式的操作或解释器模型。在这样的模型中，定义程序"状态"，而程序执行则用状态的改变来描述。例如，语句A:= 1的语义是相应于A的状态组件变为1。

维也纳定义语言（Vienna definition language，VDL）（Bjorner and Jones 1978）包含一个操作模型，其中抽象程序树（IR的一种变形）被遍历并用值来修饰以便对程序执行建模。VDL已被用于定义PL/I的语义，尽管所得到的定义非常巨大而冗长。

公理定义（Axiomatic definition）（一份极好的参考文献是Gries [1981]）可以用于在比操作模型更为抽象的层次上执行建模。这些语义基于形式化定义的、与程序变量相关的关系或谓词。语句以它们如何改变这些关系来定义。 13

作为公理定义的示例，定义var := exp的公理通常陈述在语句执行之后一个涉及var的谓词为真，当且仅当预先把该谓词中所有出现的var替换为exp所得到的谓词为真。例如，为使y>3在语句y := x+1执行之后为真，谓词x+1>3在语句执行前应当为真。类似地，y=21在执行x:=1之后为真，如果y=21在它执行前为真，这是宣称x的改变不影响y的迂回方式。然而，如果x是y的一个别名，则该公理是无效的。例如，如果x是Pascal的一个绑定到y的var参数，则x是y的别名。事实上，别名使得公理定义更加复杂。这也是在现代语言设计中普遍试图限制或禁止别名的原因。Ada简单地声明使用别名的程序是非法的；其他语言则包含消除别名的特性和静态语义约束。

公理方法对程序正确性的推导证明是很好的，因为它避免了实现细节，并专注于语句的执行如何改变变量之间的关系。因此，在我们的赋值公理中没有被更新的内存中位置的概念；相反，赋值语句改变了变量间的关系。尽管公理可以形式化地描述程序设计语言语义的重要性质，但是用它们来完整地定义大多数程序设计语言是非常困难的。例如，它们无法很好地为像内存耗尽这样在实现上需考虑的因素建模。

指称模型（denotational model）比操作模型在形式上更加数学化，而且提供了作为冯·诺依曼（von Neumann）语言中心内容的内存访问和更新概念。因为它们依赖于数学中的概念和术语，指称定义通常相当简洁，尤其是与操作定义相比。

指称定义可被看作是语法制导的定义，因为一个结构的意义依照它的直接组成部分的意义来定义。例如，为定义加法，可以使用下列规则：

$$E[T1 + T2] = E[T1] \text{ is integer and } E[T2] \text{ is integer} \Rightarrow \text{range}(E[T1] + E[T2]) \text{ else error}$$

该定义宣称如果两个操作数（T1和T2）都是整数（E操作符计算表达式的意义），则T1 + T2的意义是操作数的值之和，测试保证它们在可表示的整数范围之内。否则，该表达式的意义是一个错误值。

指称技术已经变得相当普遍，而且已经写出了Ada的大部分定义（不包括并发性）。该定义已经成为许多早期Ada编译器的基础。这些编译器通过实现一个给定程序的指称表示来运行。第一个采用这种方法的Ada系统是NYU Ada-Ed系统，但是它执行程序的速度非常慢。它的编写者声称缓慢主要是由关键指称功能的低效实现造成的。编译器研究中的大量努力都指向找出将指称表示自动转换为等价的可直接执行的表示的方法（Sethi 1983, Wand 1982, Appel 1985）。如果这种努力获得成功，指称定义（以及词法和语法定义）可能足够自动产生一个可工作的解释器或编译器。 14

再者，关注精确的语义规范的动机基于这样一个事实：为一个程序设计语言编写完整的和精确的编译器需要该语言被良好地定义。该断言看起来是不言而喻的，但许多语言都是由不精确的语言参考手册

所定义。这样一个手册在典型情况下包含正式的语法规范，但是是以非正式的文字风格编写。结果产生的定义不可避免地是二义的或在某些方面是不完整的。例如，考虑下面摘自Pascal的表达式：

```
(I <> 0) and (K div I > 10)
```

该表达式总是被良好定义的吗（假定I和K是已经适当初始化的整数）？它依赖于是否必须对该表达式的两个操作数都进行求值（如果不需要对两个操作数都进行求值的话，它还依赖于操作数的计算次序）。如果**and**被视为普通二元运算符，在应用**and**之前，它的两个操作数都必须被求值。这意味着如果I=0，该表达式将会失败（被零除错误）。然而，**and**是特殊的，因为在确定它的值时它的两个操作数不必都进行计算。特别地，如果**and**的左边操作数为假，则整个表达式必定为假。这就是*短路计算*（short-circuit evaluation），经常用于布尔运算符。如果使用短路计算，上述表达式总是被良好定义的。

原始的Pascal定义中没有提到短路计算。它简单地说计算次序是未指定的，而类似上面那样依赖于计算次序的表达式，则应当避免。结果，某些Pascal编译器使用普通的完全计算，而其他的编译器则对布尔运算符使用短路计算。这种不兼容性导致在编译器间移植程序时存在严重问题。所有语言都必须允许某些细节（如整数和实数的范围）依赖于实现，但是留给实现者决定的每个选择显然都是创造编译器间不兼容性的机会。

编译器通常作为事实上的语言定义。也就是说，程序设计语言事实上是由编译器选择接受的内容以及选择如何翻译语言结构的方式来定义。事实上，上面介绍的形式化语义定义的操作方法就采用了这种观点。为一种语言定义一个标准解释器，那么程序的意义就恰好是解释器所宣称的任何东西。一个早期的（而且非常优雅的）例子是由McCarthy（1965）所提供的LISP解释器的一个操作定义。假定仅仅采用7个基本函数以及参数绑定和函数调用的概念，就能够根据LISP解释器的动作定义出整个LISP。

15

1.5 编译器设计与程序设计语言设计

我们的主要研究兴趣是现代程序设计语言编译器的设计和实现。该研究的一个有趣的方面是程序设计语言设计和编译器设计之间如何相互影响。显然，程序设计语言的设计影响并常常支配编译器的设计。许多聪明且有时非常精巧的编译器技术都由应付某些程序设计语言结构的需要而产生的。一个很好的例子是*thunk*机制，它为处理Algol 60的*换名调用*（call-by-name）参数而被发明出来。（thunk是特殊类型的函数，由编译器创建，用来获得某个参数的值。）编译器设计的技术发展水平也强有力地影响了程序设计语言的设计，因为不能被有效编译的程序设计语言通常是不被采用的！许多成功的程序设计语言设计者（如Pascal和Modula-2的设计者Niklaus Wirth）都拥有广泛的编译器设计背景。易于编译的程序设计语言有许多优点：

- 它们通常易于学习、阅读和理解。如果一个特性难以编译，它很可能也难以理解。
- 它们将会在许多种机器上都拥有高质量的编译器。这对于一种语言的成功通常是至关重要的。例如，C、Pascal和FORTRAN到处可用且非常流行；PL/I和Algol 68则仅有有限的可用性，因而远不够流行。
- 它们通常会生成更好的代码。质量低劣的代码在关键性的应用程序中是致命的。
- 它们会有较少的编译器错误。如果编译器编写者没有完全理解一种语言，他如何能够制造出一个可靠的编译器？
- 编译器会更小、更廉价、更快、更可靠、更普及。
- 编译器错误诊断和程序开发特性通常会更好。

贯穿编译器设计学习的全过程，我们将会从许多语言中得到想法、解决方案以及明白它们的缺点。我们主要关注Ada，但也会考虑Pascal、Algol 60、C和FORTRAN。专注于Ada并非是因为它易于编译，

而是因为它向编译器设计者提出了许多挑战，并因此为我们提供了一个介绍各种编译技术的统一框架。我们也将有限地引用Simula 67、PL/I、Algol 68和Euclid。这些语言中的每一种都被认为是*可编译的*（尽管某些语言比其他语言更具有挑战性）。每一种语言都可以被适当地翻译为典型计算机的机器语言。根据先前的讨论，每种这样的语言都有一个明确的翻译阶段。 16

相反，像Snobol和APL这样的语言通常被认为是*不可编译的*，因为在最一般的情况下，在这些语言中所指定的操作，仅使用执行前可用的信息，将不能完全地被翻译为机器代码。（这些语言的编译器确实存在；它们试图尽可能多地翻译一个程序，将其余的部分留给运行时例程来解释。）

一个允许编译的程序设计语言必须具备什么属性呢？有许多需要考虑的问题，但问题主要是什么可以在执行开始之前被确定和限制，以及什么必须推迟到执行开始以后才能确定。也就是说，哪些方面是动态的，而哪些方面又是静态的？尤其是要考虑下列的问题：

- 能否在执行开始之前确定每个标识符引用的作用域和绑定？也就是说，能否在编译时确定由标识符所表示的数据对象或者指向数据对象的指针？如果不能，可能必须在每次用到时，在一张运行时符号表中对标识符进行有效的查找。
- 能否在运行开始之前确定一个对象的类型？如果不能，每个运算符的意义可能需要不断地重新确定。例如，如果A和B是数组而不是整数的话，A*B可能表示非常不同的意义。

  像Snobol和APL这样没有类型声明且允许变量的类型动态改变的语言，称为*非类型的*（untyped）或*动态类型的*（dynamic-typed）语言。注意：这些语言与*无类型的*（typeless）语言（如Bliss或BCPL）是不同的。在无类型的语言中只有一种类型的数据，单元（cell）或字（word）。许多系统实现语言（System Implementation Language, SIL）是无类型的或几乎是无类型的。例如，早期版本的C更多地使用类型来定义对象大小而不是指定允许的操作类别，这与在像Pascal和Ada这样的语言中完成的对操作数的强类型检查形成鲜明对比。C现在比最初时变得更加强类型化。
- 现存的程序文本在执行过程中能否被改变或添加？如果能，程序文本则可能需要用介于源语句和机器代码之间的某种内部表示进行保存。LISP是允许在运行时创建程序文本的语言中最著名的例子，但Snobol和APL也拥有此特性。

一般地，可编译的语言是那些拥有结构、标识符作用域以及类型绑定在编译时（即在执行之前）固定的语言。实际上，对于一个编译器来说，如果它要将程序操作完全翻译成具体的机器指令，则它必须知道这些程序组件是静态的。

Pascal程序的结构是编译所考虑的问题如何影响语言设计的另一个例子（不真正涉及语言特性）。Pascal被设计为可由一遍编译器进行翻译。所有全局声明必须在程序开头出现，以保证它们对所有需要引用它们的子程序可用。同样地，主程序出现在程序文本的最后，所以它所使用的任何变量和它所调用的任何子程序都已经被定义。这种程序结构并不显著地增强程序的可读性，但它确实为编译器的效率做出积极的贡献。 17

1.6 编译器分类

有许多种类的编译器变体。*错误诊断编译器*（diagnostic compiler）（例如PL/C [Conway and Wilcox 1973]、WATFIV [WATFIV 1981]、UW-Pascal等）被特别设计以辅助程序的开发和调试。它们提供对程序的详细检查并详细地说明程序员的错误。它们通常能够自动*修复*较小的错误（例如，缺少逗号或括号）。某些程序错误仅在运行时（即在程序执行时）才能被检测到。这样的错误包括无效下标、指针的误用以及非法文件操作。错误诊断编译器有能力包含能够检测运行时错误并中止程序执行的检查代码。尽管错误诊断编译器主要用于教学环境中，但是错误诊断技术在所有编译器中都是有价值的。过

去，错误诊断编译器仅用于程序开发的开始阶段。当程序接近完成时，使用“产品编译器”（production compiler）。产品编译器通过忽略错误诊断问题来提高程序速度。这种策略被Tony Hoare比喻为在旱地上航海课时穿着救生衣，但在海里时却把它扔掉！实际上，有一点变得日益清楚：对于几乎所有的程序，正确性而非速度才是最重要的问题。

优化编译器（optimizing compiler）被特别设计用来以编译器复杂性和编译时间的增加为代价产生高效的机器代码。实践中，所有的产品级质量编译器——它们的输出将被用于日常工作中——都做出某种努力以生成好的机器代码。例如，对表达式I+0通常不会生成加法指令。

术语“优化编译器”实际上是一个误称，因为没有任何的编译器，无论它多么复杂，能够为所有程序产生最优代码。其原因有两点。首先，计算科学在理论上已经证明即使像两个程序是否等价这么简单的一个问题都是不可判定的；也就是说，它不可能由任何计算机程序来解决。因此，找出一个程序最简单的（而且最高效的）翻译并不总是能够完成的。其次，许多程序优化需要与被编译的程序大小的指数函数成正比的时间。最优代码，即使在理论上是可能的，在实践中通常也是办不到的。

优化编译器实际上使用各种混合的“转换”来改善程序的性能。优化编译器的复杂度源于对多种转换的使用需要，其中一些转换互相冲突。例如，将频繁使用的变量放在寄存器中减少了它们的访问时间，
18 但使得过程和函数调用更加昂贵，因为需要在调用期间保存寄存器。许多优化编译器提供很多优化级别，各个级别以逐渐增大的代价提供逐渐增加的代码改进。对于哪种改进最有效（并且最廉价）的选择是判断和经验的问题。在后面各章中，将建议可能的优化，并强调那些既简单又有效的优化。全面的优化编译器超出了本书的范围，但以合理的代价产生高质量代码的编译器是可以达到的目标。

编译器是为特定的程序设计语言（源语言）和特定的目标计算机（编译器将为其生成代码）所设计。给定现存的多种程序设计语言和计算机，必须编写大量相似、但不相同的编译器。这种形势决定了从事编译器编写行业的人们的利益，但它也导致许多重复努力以及编译器质量的极大差异。结果，一种新的编译器类型，可再目标编译器（retargetable compiler），变得尤为重要。

可再目标编译器的目标机器可以改变，而不需要重写它的与机器无关的组件。可再目标编译器比普通编译器更难以编写，因为必须仔细地局部化对目标机器的依赖。类似地，可再目标编译器通常难以像普通编译器那样生成同样高效的代码，因为难以利用特殊情况和机器特性。尽管如此，因为可再目标编译器允许分担开发成本并提供跨计算机的一致性，所以它不失为一项重要的新技术。在学习编译的基本原理时，将专注于目标为单一机器的编译器。在后面的各章中，也将考虑为提供再目标能力所需的技术。

实际中的编译器仅仅是在编辑-编译-测试循环中使用的一个工具。用户首先编辑一个程序，然后编译它，最后测试它的性能。因为程序错误必须被发现并纠正，所以该循环会重复很多次。一种新类型的程序设计工具，程序开发环境（Program Development Environment，PDE），被设计用来集成该循环（例如，康奈尔程序综合器[Teitelbaum and Reps 1981]、Gandalf [Notkin 1985]、Cope [Archer and Conway 1981]、Poe [Fischer et al. 1984]）。PDE允许增量式地构造程序，其中完整集成了程序检查和测试。PDE可被视为编译器演化过程中的下一个阶段。本书中讨论的编译技术是创建PDE所需要的基本要素。

1.7 影响编译器设计的因素

以前，计算机的指令集更多地面向汇编级程序设计而不是高级程序设计。结果，生成高质量的代码变得非常难以实现（Wulf 1981）。存在的问题有很多：

- 众所周知，指令集是不一致的。例如，某些操作必须在寄存器中完成，而其他的操作可以在内存
19 中完成。通常存在许多寄存器类，每一类适于某些特殊的操作类。
- 高级操作没有被很好地支持。虽然大多数现在的语言支持块结构和动态存储分配，但栈和堆存储

通常还是难以高效地实现。尽管如此，Wulf警告说，不要使通用的体系结构过分紧密地面向特定的语言。

- 数据和程序集成的价值被低估，而速度则被过分强调。结果，程序设计错误可以不被发现，只因为害怕额外的检查会造成不可接受的程序执行的减缓。

因为可进行微程序设计的机器广泛可用，以及在工程师和计算机科学家之间更好的沟通，所以情况正在改善。通过把指令集调整为最普遍需要的操作，可以实现程序大小的惊人缩减（已经报告了三倍的缩减）。单独的指令可以支持普通的高级操作，例如，VAX拥有单独的指令，可以在过程调用中保存寄存器、传递参数以及压入栈空间。那些被认为昂贵得惊人的操作（如动态存储管理）可以作为机器设计的一个必要组成部分。它的一个好的例子是MIT LISP机器（Sussman 1981），其中包含集成的垃圾收集器（garbage collector）。彻底的经过重新组织的机器设计也允许更仔细地控制程序和数据对象的使用（以及潜在的误用）。例如，Intel iAPX 432对多种访问控制机制提供了直接硬件支持。

另一学派的思想则主张指令集体系结构应当更简单，而不是更复杂。依据这种哲学设计的机器称为*精减指令集计算机*（reduced instruction set computer, RISC）（Patterson 1985）。奇怪的是，这些设计也被认为是面向编译器的，因为它们通常假定它们的程序将以高级语言编写并由优化编译器处理。它们的指令集的简单性意在通过使代码生成器可能的选择最小化来减轻编译器的任务。然而，有趣的是，要注意：即使是一些RISC机器也拥有对子程序调用的相对复杂的支持。

总之，高效和可靠地支持现代程序设计语言结构的需要已经开始对指令集设计产生巨大的影响。这里还不十分清楚我们描述的这两种方法中的哪一种是最合适的。重要的是两者都慎重考虑用高级语言编写的程序的执行，因此在最佳利用已有硬件资源方面，它们都极大减轻了编译器编写者的负担。

在后续各章中，假设了一个虚拟机器，它的指令集对于现代计算机是有代表性的。选择一个特殊的指令集是为了使我们的讨论更具体。在实践中，读者也会希望把它替换为一个更熟悉的体系结构。熟悉目标机器是至关重要的；代码生成的本质是确定哪种指令序列可以最好地实现给定的结构。像编译器设计中的其他方面一样，经验是最好的指导，因此我们从翻译一个非常简单的语言开始，并努力完成今后更具挑战性的翻译任务。 20

练习

1. 我们介绍的编译模型基本上是面向批处理的。特别地，它假定一个完整的源程序已经编写出来，而且在程序员能够执行或对程序进行任何改动之前该程序将被完全编译。一个有趣的选择是“交互式编译器”（interactive compiler）。交互式编译器通常作为程序开发环境的一部分，允许程序员交互式地响应程序错误，在错误被检测到时即对其进行改正。它也允许程序在被完全编写出来以前进行测试，提供逐步的实现和测试。

   重新设计图1-3的编译器结构，以允许增量式编译。（关键思想是允许对个别的程序结构进行修改和编译而不必重新编译所有东西。）

2. 在1.7节中我们注意到RISC体系结构通过减少代码生成器所必须做出的选择数量对其进行简化。RISC体系结构在很大程度上从所谓的CISC（Complex Instruction Set Computer，复杂指令集计算机）体系结构的负面反映中得到启发。在CISC中包含大量的操作代码和寻址模式。一个著名的CISC体系结构是流行的VAX系列计算机。

   假定要你编写一个以CISC机器（像VAX）为目标的编译器。你有两种选择。可以设计编译器使其利用所有的指令和寻址模式，或者审慎地选择一个可用的子集并仅使用该子集。分析这两种方法的权衡之处。在什么情况下你会分别推荐每一种方法？

3. 为认识语法（syntax）、静态语义（static semantics）和运行时语义（run-time semantics）的区别，在你喜欢的教科书中查找Pascal的with语句定义。对定义with的每条规则，确定它属于哪一种定义种类。
4. 编译器通常以它们所编译的语言来编写。当有人考虑最初的编译器是怎么来的时，这会造成“鸡和蛋”的问题。如果你需要在系统Y上为语言X创建第一个编译器，一种方法是创建一个交叉编译器（cross-compiler）。交叉编译器在一台机器上运行但为某个不同的机器生成代码。解释你会如何使用交叉编译来创建在系统Y上运行的语言X的编译器，并为系统Y生成代码。如果系统Y是裸系统——即没有操作系统或任何语言的编译器——会引起什么额外的问题？
5. 交叉编译假定：语言X的编译器存在于某台机器上。当为一种新语言创建第一个编译器时，该假定不成立。在这种情况下，可以采用一种自举（bootstrapping）方法。首先，选择语言X中足够实现一个简单编译器的子集。其次，X子集的简单编译器由任何可用的语言编写。该编译器必须是正确的，但它不应比所需的更复杂，因为它将随后被丢弃。下一步是用X的子集重写X的子集编译器并随后使用
21 先前创建的子集编译器重新编译它。最后，可以扩大X的子集以及它的编译器，直到以X编写的X的一个完整编译器可用为止。

   假定你要自举Pascal（或C）。简要描述一个合适的子集语言。该语言必须具备哪些特性？其他的哪些特性也是值得具备的？
6. 为了允许建立可打印的文档，创建了像*troff* 和T_EX这样的工具。它们可被视为程序设计语言的变体，其输出控制激光打印机或照排机。源语言命令控制像间距、字体选择、磅值以及特殊符号这样的细节。如果要翻译*troff* 或T_EX的输入，使用图1-3的语法制导编译器结构，提出在每个编译器阶段应当发生的处理的种类。

   为文档进行“程序设计”的另一种方法是使用复杂的编辑器来交互式地输入和编辑文档。（编辑操作允许选择字体、选择磅值、包含特殊符号等。）这种文档准备的方法被称为“所见即所得”，因为文档的确切形式总是可见的。

   这两种方法的相对优点和缺点各是什么？对于程序设计语言是否存在类似的方法？
7. 尽管编译器被设计用于翻译特定的语言，但它们通常也允许调用以其他语言（通常为FORTRAN、C或汇编语言）编写的子程序。为什么要允许这样的“对外调用”（foreign call）？它们以何种方式使
22 编译复杂化？

第2章　一个简单编译器

在这一章，为了向读者提供对编译过程组织的总体认识，我们将较为详细地考虑如何为一个非常小的程序设计语言构造编译器。我们的语言称为*Micro*，它是一个极为简单的语言，甚至缺少足够的特性用来编写一个有用的程序。设计Micro只是为讨论简单的示例编译器提供一个具体的语言。

我们首先非形式化地定义Micro:

- 仅有的数据类型是整型。
- 所有的标识符采用隐式声明，且其长度不超过32个字符。标识符必须以字母开头并由字母、数字和下划线组成。
- 文字常量由一串数字组成。
- 注释由“--”开始，并在当前行尾结束。
- 语句类型为

赋值语句:

```
ID := Expression ;
```

Expression是由标识符、文字常量以及+和-运算符组成的中缀表达式结构，其中允许含有括号。

输入/输出语句:

```
read (List of IDs) ;
write (List of Expressions) ;
```

- **begin**、**end**、**read**和**write**为保留字。
- 每条语句以分号（;）结束。程序体由**begin**和**end**界定。
- 源代码中每行结尾添加一个空格符；因此，词法记号不能跨行。

2.1　Micro编译器结构

本章的主题是一个简单的Micro编译器。该编译器结构基于图1-3的描述。为简单起见，该编译器采用单遍编译技术，其最重要的特征是不使用显式的中间表示。各组件间的接口如下:

- 词法分析器从文本文件中读取源程序并产生记号表示流（在第3章中严格定义）。事实上，在任意时刻都不必有实际的流存在，因为词法分析器实际上是一个由语法分析器调用的函数，每调用一次，就产生一个记号表示。
- 语法分析器处理记号，直到遇到需要语义处理的语法结构。它随后直接调用语义例程。其中有些语义例程在其处理过程中使用记号表示。
- 语义例程为一个简单的三地址虚拟机产生汇编语言代码输出。因而，在Micro编译器结构中没有优化器，而代码生成也是通过从语义例程直接调用适当的支持例程来完成的。
- 符号表仅由语义例程使用。其接口在2.5.5节描述。

2.2　Micro词法分析器

Micro词法记号可被形式化地定义，参见第3章。依照现在的目的，仅需一个非形式化的定义。创建

Micro编译器的第一步是构造Micro词法分析器。定义枚举类型token表示Micro的词法记号集。词法分析器将是一个不带参数，返回token类型值的函数。

```
typedef enum token_types {
    BEGIN, END, READ, WRITE, ID, INTLITERAL,
    LPAREN, RPAREN, SEMICOLON, COMMA, ASSIGNOP,
    PLUSOP, MINUSOP, SCANEOF
} token;

extern token scanner(void);
```

词法分析器读入字符并将它们组成词法记号。注意：有时不能读过多的字符。特别地，需要查看下一个记号的第一个字符以判断当前记号是否结束。对于Micro，仅需超前搜索一个字符。额外的那个字符可使用标准的ungetc()函数方便地压回输入流。

为简单起见，假定输入来自stdin。实际上应该打开一个源文件并使用一个显式的FILE指针。

词法分析器在被调用时必须找到某个记号的开头。为此，它检查下一个输入字符。如果该字符不是任何记号的开头，则产生一个词法或记号错误，显示错误信息，并试图从错误中恢复。简单的恢复方法是忽略该字符并重新开始扫描。该过程持续到找到某个记号的开头字符为止。随后匹配可组成一个合法记号的最长可能字符序列。

图2-1给出识别Micro标识符和文字常量（整数常量）的词法分析器的主循环。它忽略空白字符（空格符、制表符和行结束标记）。为简单起见，假定存在“行结束”字符。在C语言中，通常称之为“换行符”，并以转义序列'\n'表示。即使字符集中缺少特定的“换行符”，C语言的输入/输出库也会在遇到某种行结束标记时返回'\n'。

```
#include <stdio.h>
#include <ctype.h>

int in_char, c;

while ((in_char = getchar()) != EOF) {
    if (isspace(in_char))
        continue;    /* do nothing */
    else if (isalpha(in_char)) {
        /*
         * ID ::= LETTER | ID LETTER
         *               | ID DIGIT
         *               | ID UNDERSCORE
         */
        for (c = getchar(); isalnum(c) || c == '_';
                    c = getchar())
            ;
        ungetc(c, stdin);
        return ID;
    } else if (isdigit(in_char)) {
        /*
         * INTLITERAL ::= DIGIT |
         *                INTLITERAL DIGIT
         */
        while (isdigit((c = getchar())))
            ;
        ungetc(c, stdin);
        return INTLITERAL;
    } else
        lexical_error(in_char);
}
```

图2-1 识别标识符和整数常量的词法分析器循环

在词法分析器中增加操作符、注释和分隔符等词法记号的识别也很容易。图2-2在图2-1的循环的基

础上添加了识别上述记号的代码。

```
#include <stdio.h>
#include <ctype.h>

int in_char, c;

while ((in_char = getchar()) != EOF) {
    if (isspace(in_char))
        /* do nothing */
        continue;
    else if (isalpha(in_char))
        /* code to recognize identifiers goes here */
    else if (isdigit(in_char))
        /* code to recognize int literals goes here */
    else if (in_char == '(')
        return LPAREN;
    else if (in_char == ')')
        return RPAREN;
    else if (in_char == ';')
        return SEMICOLON;
    else if (in_char == ',')
        return COMMA;
    else if (in_char == '+')
        return PLUSOP;
    else if (in_char == ':') {
        /* looking for ":=" */
        c = getchar();
        if (c == '=')
            return ASSIGNOP;
        else {
            ungetc(c, stdin);
            lexical_error(in_char);
        }
    } else if (in_char == '-') {
        /* looking for --, comment start */
        c = getchar();
        if (c == '-') {
            while ((in_char = getchar()) != '\n');
        } else {
            ungetc(c, stdin);
            return MINUSOP;
        }
    } else
        lexical_error(in_char);
}
```

图2-2 加入用来识别操作符、注释和分隔符的新代码的词法分析器循环

在Micro词法分析器中尚未包含对保留字的识别。问题当然在于保留字看起来和标识符没有什么不同。保留字也许需要以某种特殊方式来标记（例如，将它们放在引号中或者以大写字母表示），但这种 25
方法对程序员来说很麻烦，因此我们宁愿不用这种方法。有两种普遍使用的方法可用于区分标识符和保留字。在第一种方法中，词法分析器拥有一张保留字表。每当一个标识符被识别时，词法分析器都会检查保留字表，如果一个记号在此表中，则它总会被解释成保留字而不是标识符。而在另一种方法中，保留字作为编译器符号表的初始部分，含有特殊属性*reserved*。词法分析器在识别一个标识符后，在符号表中查找该标识符。如果找到，并且含有该特殊属性，就将它识别为保留字。

对于Micro，两种方法都可行。假定词法分析器有一个例程`check_reserved()`，该例程返回当前 26
被识别的标识符的正确记号类（ID或者某个保留字）。然而，图2-1中识别标识符的词法分析器代码不适 ~
合使用这种方式发现保留字。在进行词法分析时，我们未提供任何措施来保存已扫描过的记号中的字符。 27
对非常简单的记号，像操作符或分隔符，知道记号类别就足够了。对于其他的记号，例如标识符和文字常量，需要知道该记号的实际文本。为此，调用例程`buffer_char()`，将其参数加入一个称为

token_buffer的字符缓冲区中。clear_buffer()把字符缓冲区重置为空字符串。该缓冲区对编译器的任何部分都可见，并总是包含最近扫描过的记号文本。在本章的示例编译器中，我们特别感兴趣的是由语义例程使用token_buffer。check_reserved()也使用该缓冲区中的字符来确定一个看起来像标识符的记号是否是保留字。

我们还必须决定如何处理文件的结尾。语法分析器必须知道何时输入被用完，以便证实已经完成对一个完整程序的分析，因此创建一个称为SCANEOF的文件结束记号。在语法分析算法的形式描述以及语法分析器生成器中，该记号通常用“$”表示。但是，“$”在任何典型的程序设计语言中并不是有效的枚举文字常量，因此我们使用SCANEOF来代替它。一旦调用词法分析器时feof(stdin)为真，则SCANEOF被返回。

图2-3包含词法分析器主例程的完整代码。其中未包含该例程使用的辅助例程（如buffer_char()等）。

```
#include <stdio.h>
/* character classification macros */
#include <ctype.h>

extern char token_buffer[];

token scanner(void)
{
    int in_char, c;

    clear_buffer();
    if (feof(stdin))
        return SCANEOF;

    while ((in_char = getchar()) != EOF) {
        if (isspace(in_char))
            continue;     /* do nothing */
        else if (isalpha(in_char)) {
            /*
             * ID ::= LETTER | ID LETTER
             *              | ID DIGIT
             *              | ID UNDERSCORE
             */
            buffer_char(in_char);
            for (c = getchar(); isalnum(c) || c == '_';
                    c = getchar())
                buffer_char(c);
            ungetc(c, stdin);
            return check_reserved();
        } else if (isdigit(in_char)) {
            /*
             * INTLITERAL ::= DIGIT |
             *                INTLITERAL DIGIT
             */
            buffer_char(in_char);
            for (c = getchar(); isdigit(c);
                    c = getchar())
                buffer_char(c);
            ungetc(c, stdin);
            return INTLITERAL;
        } else if (in_char == '(')
            return LPAREN;
        else if (in_char == ')')
            return RPAREN;
        else if (in_char == ';')
            return SEMICOLON;
        else if (in_char == ',')
            return COMMA;
        else if (in_char == '+')
            return PLUSOP;
        else if (in_char == ':') {
            /* looking for ":=" */
```

图2-3 完整的Micro词法分析器函数

```
                c = getchar();
                if (c == '=')
                    return ASSIGNOP;
                else {
                    ungetc(c, stdin);
                    lexical_error(in_char);
                }
            } else if (in_char == '-') {
                /* is it --, comment start */
                c = getchar();
                if (c == '-') {
                    do
                        in_char = getchar();
                    while (in_char != '\n');
                } else {
                    ungetc(c, stdin);
                    return MINUSOP;
                }
            } else
                lexical_error(in_char);
        }
    }
```

图2-3 （续）

28 29

2.3 Micro语法

在本节中，不会非形式化地定义Micro语法，而是将使用上下文无关文法（Context-Free Grammar，CFG）给出Micro的严格定义。CFG有时也被称为BNF（Backus-Naur form，巴科斯-诺尔范式）文法。

通俗地说，CFG就是一组重写规则或*产生式*（production）。产生式的形式为

$A \rightarrow B\,C\,D \cdots Z$

A是产生式的*左部*（Left-Hand Side, LHS）；B C D…Z组成产生式的*右部*（Right-Hand Side, RHS）。每个产生式的左部仅有一个符号。其右部可以有任意数量的符号（零个或者多个）。产生式代表了一个规则，即其左部文法符号可由其右部文法符号代替。因此产生式

<program> → **begin** <statement list> **end**

规定一个程序必须是由**begin**和**end**所界定的语句列表。

在CFG中可以出现两种文法符号：*非终结符*（nonterminal）和*终结符*（terminal）。本书中，非终结符通常由“<”和“>”限定以易于识别。不过，非终结符也可由它们出现在产生式左部来识别。非终结符实际上是占位符。所有非终结符必须根据以它作为左部的产生式来替换或重写。相反，终结符从不被改变或重写。它们代表语言的*词法记号*（token）。因此，一组产生式（一个CFG）的全部目的就是指定哪些终结符（词法记号）序列是合法的。CFG用一种非常优雅的方式完成这个目标：由一个称为*开始符号*（start）或*目标符号*（goal）的非终结符号开始，随后应用产生式重写非终结符，直到仅剩下终结符。任何可由此产生的终结符序列都被认为是合法的。类似地，如果一个终结符序列不能由任何非终结符替换序列产生，则该序列被认为是非法的。为了明白这个过程如何起作用，让我们看一下Micro的一个CFG。λ代表空或空字符串。因此，产生式A→λ规定A可被空字符串替换，实际上就是被消去。

程序设计语言结构通常含有可选项或者项列表。为了清晰地表示这样的特性，通常利用*扩展*BNF记号（extended BNF notation）。可选项序列由方括号“[”和“]”括起来，例如，在产生式

<program> → [ID :] **begin** <statement list> **end**

中，程序有一个可选的标号。可选的项列表由大括号“{”和“}”括起来。因此在产生式

30 <statement list> → <statement> {<statement>}

中，语句列表定义为一条语句接着零条或多条可选的语句。

扩展BNF与普通BNF有着相同的定义能力。特别地，可使用下列转换将扩展BNF映射为标准形式。可选序列由产生λ或序列项的新非终结符替换。类似地，可选列表由一个新非终结符替换，该非终结符产生λ或列表项接着该非终结符。因此语句列表可转换为

<statement list> → <statement> <statement tail>
<statement tail> → λ
<statement tail> → <statement> <statement tail>

扩展BNF的优点是更紧凑易读。可以想像一个预处理器取扩展BNF作为输入，使用上面这些转换方法产生标准BNF。

图2-4给出Micro的扩展CFG。一个拓广产生式：

<system goal> → <program> SCANEOF

出现在文法中，以确保由<system goal>匹配的字符串包含了所有输入。它指定文件结束标记SCANEOF必须紧随程序的最后一个有效词法记号。

1.	<program>	→ **begin** <statement list> **end**
2.	<statement list>	→ <statement> {<statement>}
3.	<statement>	→ ID := <expression> ;
4.	<statement>	→ **read** (<id list>) ;
5.	<statement>	→ **write** (<expr list>) ;
6.	<id list>	→ ID {, ID}
7.	<expr list>	→ <expression> {, <expression>}
8.	<expression>	→ <primary> {<add op> <primary>}
9.	<primary>	→ (<expression>)
10.	<primary>	→ ID
11.	<primary>	→ INTLITERAL
12.	<add op>	→ PLUSOP
13.	<add op>	→ MINUSOP
14.	<system goal>	→ <program> SCANEOF

图2-4 定义Micro的扩展CFG

为了明白该文法如何定义合法的Micro程序，让我们从非终结符<program>开始，推导这样一个程

31 序，**begin** ID := ID + (INTLITERAL – ID); **end**。

<program>	
begin <statement list> **end**	(Apply rule 1)
begin <statement> {<statement>} **end**	(Apply rule 2)
begin <statement> **end**	(Choose 0 repetitions)
begin ID := <expression> ; **end**	(Apply rule 3)
begin ID := <primary> {<add op> <primary>} ; **end**	(Apply rule 8)
begin ID := <primary> <add op> <primary> ; **end**	(Choose 1 repetition)
begin ID := <primary> + <primary> ; **end**	(Apply rule 12)
begin ID := ID + <primary> ; **end**	(Apply rule 10)
begin ID := ID + (<expression>) ; **end**	(Apply rule 9)
begin ID := ID + (<primary> {<add op> <primary>}) ; **end**	(Apply rule 8)
begin ID := ID + (<primary> <add op> <primary>) ; **end**	(Choose 1 repetition)
begin ID := ID + (<primary> – <primary>) ; **end**	(Apply rule 13)
begin ID := ID + (INTLITERAL – <primary>) ; **end**	(Apply rule 11)
begin ID := ID + (INTLITERAL – ID) ; **end**	(Apply rule 10)

现在，没有剩下非终结符，因此上面Micro程序的推导完成。

CFG定义一个语言，即词法记号序列的集合。能用文法推导出的任意词法记号序列都是有效的；反之则是无效的。事实上，严格地说，从CFG推导出的任意词法记号序列都被认为是语法上有效的。当语义例程检查静态语义时，可发现语法上有效的程序所包含的语义错误。例如，Pascal语言的语句

A := 'X' + True;

没有语法错误，但是含有一个语义错误："+"操作符不是用来将字符与布尔值相加的。

在CFG中既可以定义语法，又可以定义结构。对于表达式，这包括结合性（associativity）和运算符优先级（operator precedence）。结合性关系到应用连续操作符实例的顺序（如在A-B-C中）。运算符优先级是指操作符的相对优先级。例如，我们期望A+B*C表示A+(B*C)，是因为通常认为"*"比"+"有更高的优先级。Micro仅有一个优先级层次，因为它不包含乘法。如果包含乘法，则它应该比加法的优先级更高。下列文法片段定义了这样的优先级关系。

```
<expression> → <factor> {<add op> <factor>}
<factor>     → <primary> {<mult op> <primary>}
<primary>    → ( <expression> )
<primary>    → ID
<primary>    → INTLITERAL
```

分析由该文法片段推导出的表达式A+B*C的推导树（derivation tree），可以说明该定义如何工作。 32
推导树以非终结符的扩展作为其子树。图2-5给出上面表达式的推导树。该树显示"*"比"+"有更高的优先级，因为第二个和第三个ID（由"*"）组合在一棵子树中，而该子树又和第一个ID（由"+"）组合在主推导树中。

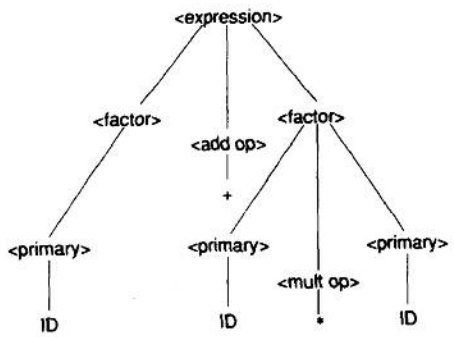


图2-5 A+B*C的推导树

在该文法中可以形成一棵有错误优先级的推导树吗？答案是否定的，因为产生式规则不允许。试着构造ID+ID*ID，其中先应用"+"。由于"*"仅可由<factor>生成，因此ID+ID必须出现在以<primary>为根的子树中。然而，<primary>不能生成ID+ID，除非将它放在括号中。通过括号可以强制结合，如图2-6所示。

2.4 递归下降语法分析

Micro语法分析器使用一种著名的语法分析技术，称为递归下降（recursive descent）。该名字取自递归语法分析例程，实际上，在处理一个程序时，它下降遍历所识别的分析树。递归下降是用于实际编译器中的最简单的语法分析技术之一。递归下降语法分析的基本思想是：每个非终结符都有一个相关的语法分析过程（parsing procedure）用以识别由该非终结符生成的任意词法记号序列。在语法分析过程中，非终结符和终结符都能被匹配。为匹配非终结符A，调用A所对应的语法分析过程。按照惯例，该分析过程也命名为A。这些调用可以是递归的，因此称为递归下降方法。为匹配终结符号t，调用过程match(t)。match()调用词法分析器获得下一个词法记号。如果是t，则一切正常，且该记号被存放在名为current_token的全局变量中。如果该记号不是t，则发现了一个语法错误，产生错误信息，并进 33
行一些错误修正或补救以重新开始语法分析并继续进行编译。

为了明白该过程如何工作，考虑为对应Micro文法产生式所编写的语法分析例程。通过调用

<system goal>所对应的过程启动语法分析器:

```
void system_goal(void)
{
    /* <system goal> ::= <program> SCANEOF */
    program();
    match(SCANEOF);
}
```

也就是说，为了正确分析一个Micro程序，必须匹配由<program>生成的词法记号序列，其后紧随一个SCANEOF。类似地，对于<program>，有以下过程:

34

```
void program(void)
{
    /* <program> ::= BEGIN <statement list> END */
    match(BEGIN);
    statement_list();
    match(END);
}
```

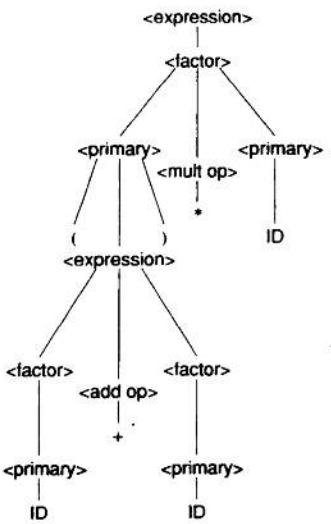


图2-6 (A+B)*C的推导树

对于<statement list>，必须决定怎样处理可选的语句序列。令next_token()是返回下一个要匹配的词法记号的函数。如果next_token()可作为从非终结符<statement>生成的第一个（即最左边的）词法记号，则试图识别可选的语句。否则，我们断定已经匹配了一个完整的语句列表。这种方法并不对所有的CFG有效，但CFG的子集——LL(1)文法却非常适合于递归下降语法分析。LL(1)文法及其分析将在第5章详细讨论。

如何确定哪些词法记号可由一个非终结符作为首记号推导出呢？我们将在第5章说明如何自动计算这些词法记号。但对于像Micro这样很小的CFG，这项工作可由手工检查完成。假定想得到非终结符A的所有首记号。将此记号集表示为First(A)。选择由A作为左部的所有产生式。对于每个这样的产生式，简单地检查它右部的最左符号。如果该符号是终结符，将其加入First(A)。如果该符号是非终结符B，计算First(B)，将结果加入First(A)。对于<statement>，情况比较简单，因为<statement>的所有产生式以终结符开始。相应的语法分析过程如下:

```
void statement_list(void)
{
```

```
    /*
     * <statement list> ::= <statement>
     *                      { <statement> }
     */
    statement();
    while (TRUE)  {
        switch (next_token()) {
        case ID:
        case READ:
        case WRITE:
            statement();
            break;
        default:
            return;
        }
    }
}
```

35

在<statement>所对应的语法分析过程定义中，出现了<statement>位于多条产生式左部的情况。因此，必须决定匹配哪一个产生式。为此，必须仔细分析每个产生式可能产生什么。考虑相应于每个<statement>产生式的First集。如果这些集合对于每个产生式均惟一，则可做出惟一选择——选择在其First集中包含next_token()的那条产生式。如果某产生式右部为λ，则当没有其他产生式被选中时，将其作为默认情况进行匹配（显然λ产生空的First集）。如果next_token()未出现在任何一个First集中，并且没有λ产生式，则产生一个语法错误，因为没有产生式可以匹配下一个词法记号。

并非在所有CFG中，共享相同左部符号的产生式都可基于它们的首记号集被区分。然而，LL(1)文法确实有此性质，这也是它们适于递归下降语法分析的原因。

<statement>以及其余的Micro语法分析过程在图2-7中一并给出，它们都使用本节介绍的技术来构造。

```
void statement(void)
{
    token tok = next_token();

    switch (tok) {
    case ID:
        /* <statement> ::= ID := <expression> ; */
        match(ID); match(ASSIGNOP);
        expression(); match(SEMICOLON);
        break;

    case READ:
        /* <statement> ::= READ ( <id list> ) ; */
        match(READ); match(LPAREN);
        id_list(); match(RPAREN);
        match(SEMICOLON);
        break;

    case WRITE:
        /* <statement> ::= WRITE ( <expr list> ) ; */
        match(WRITE); match(LPAREN);
        expr_list(); match(RPAREN);
        match(SEMICOLON);
        break;

    default:
        syntax_error(tok);
        break;
    }
}
void id_list(void)
{
    /* <id list> ::= ID { , ID } */
    match(ID);
```

图2-7 Micro的其余语法分析过程

```
    while (next_token() == COMMA) {
        match(COMMA);
        match(ID);
    }
}

void expression(void)
{
    token t;

    /*
     * <expression> ::= <primary>
     *                  { <add op> <primary> }
     */
    primary();
    for (t = next_token(); t == PLUSOP || t == MINUSOP;
                t = next_token()) {
        add_op();
        primary();
    }
}

void expr_list(void)
{
 /* <expr list> ::= <expression> { , <expression> } */
    expression();

    while (next_token() == COMMA) {
        match(COMMA);
        expression();
    }
}

void add_op(void)
{
    token tok = next_token();

    /* <addop> ::= PLUSOP | MINUSOP */
    if (tok == PLUSOP || tok == MINUSOP)
        match(tok);
    else
        syntax_error(tok);
}
void primary(void)
{
    token tok = next_token();

    switch (tok) {
    case LPAREN:
        /* <primary> ::= ( <expression> ) */
        match(LPAREN); expression();
        match(RPAREN);
        break;

    case ID:
        /* <primary> ::= ID */
        match(ID);
        break;

    case INTLITERAL:
        /* <primary> ::= INTLITERAL */
        match(INTLITERAL);
        break;

    default:
        syntax_error(tok);
        break;
    }
}
```

图2-7 （续）

2.5 翻译Micro

2.5.1 目标语言

现在准备好开始Micro的实际翻译工作。首先，必须确定为何种机器以及以何种形式（汇编代码、目标模块或诸如此类的形式）生成代码。为简单起见，使用三地址机器的汇编代码。这种机器的指令形式为

```
OP A,B,C
```

其中OP是操作码（或伪操作码），A和B标明特定操作的操作数，C指定操作结果的存放位置。操作数可以是变量名或整数文字常量。对于某些OP，可能不使用A或B或C（例如，停机指令就是简单的Halt）。输出的汇编码格式是字符串。对于Micro，假定所有算术运算操作都使用整数运算。

36 ~ 38

该目标代码适合于简单的虚拟机；它可以用来驱动一个解释器，或者每条指令可由一个更复杂的代码生成器扩展成真正机器的代码。事实上，我们的目标代码与常用的中间表示——四元式（quadruple）非常相似。这一点说明了虚拟指令集的一个有趣性质：视其用途，它们既可被看作中间表示，又可被看作编译器的输出。

2.5.2 临时变量

在编译过程中，常常需要使用临时分配的存储位置，称为*临时变量*（temporary），来存放计算的中间结果。对于Micro编译器，临时变量是在需要时隐式声明的*内部变量*。因为Micro中的普通变量也是隐式声明的，所以这种技术工作得很好。更实际的语言编译器常将寄存器用作临时变量，但也可能为特殊目的使用*存储器临时变量*（即内存位置，如在Micro编译器中所使用的那些临时变量），例如，当没有寄存器可用或在过程调用之前需要保存当前寄存器值的时候。我们约定作为临时变量使用的内部变量以Temp&N的形式表示，其中N是临时变量的索引，从1开始。由于“&”不能出现在普通Micro变量名中，因此普通变量和临时变量不会产生命名冲突。

2.5.3 动作符号

如第1章所述，大部分翻译工作由语法分析器调用的语义例程来完成。何时调用给定的语义例程由编译器作者决定。可通过在文法中添加*动作符号*（action symbol）来指定何时进行语义处理。动作符号在下面的示例中由#name表示，它们可被放在产生式右部的任意位置。每个动作符号都对应一个语义例程。例如，动作符号#add对应名为add()的语义例程。当创建语法分析过程时，将在原动作符号所在的位置上插入语义例程的调用或是内联的代码段，以完成语义处理。如果包含动作符号的文法作为语法分析器生成器的输入，生成器必须在生成的表格中包含适当的信息，以便在语法分析过程的相应时刻触发语义例程的调用。

动作符号对由CFG驱动的语法分析器所识别的语言没有影响。因此，它们实际上不是由CFG所指定的语法的组成部分。在此语境中，动作符号用来“注释”CFG，指示何时需要执行语义动作。当CFG用作语法分析器生成器的输入时，动作符号就不仅仅是注释。它们告诉语法分析器生成器何时调用相应的语义例程。

39

2.5.4 语义信息

在语义例程的设计中，一个重要的问题是它们操作的数据和生成信息的规范。我们的方法是将一个*语义记录*（semantic record）与每种文法符号（ID、INTLITERAL、<expression>等）相关联。每个不同的符号都会有一个独特的包含该符号适当信息的记录。同类符号的每次出现在其语义记录中都会有相

同类型的数据。因此，ID能以区别于INTLITERAL的不同种类的数据表示，但所有ID的语义记录将会有相同的格式。如果某个符号不需要语义数据，它可以有空的语义记录。例如，分号就不需要语义记录。

终结符的语义记录包含该记号的token_buffer及其导出值。例如，INTLITERAL的值被表示为整型对象，由词法记号文本导出。语法分析器成功地将一个词法记号与所期望的终结符号匹配后，调用语义例程产生这样一个记录。

语义例程通过访问某个产生式右部任意符号的信息来创建该产生式左部非终结符的语义记录。如果产生式右部的某些符号为非终结符，则这些符号的语义记录来自它们各自产生式中特定的语义例程。为了明白这种机制是如何工作的，考虑

<expression> → <primary> + <primary> #add

产生式右部的每个<primary>都将产生一个语义记录。这些语义记录记录每个操作数相关的数据（例如，它存储在哪里，值是多少）。当调用add()时，必须给出这些记录作为参数。这些记录用来产生适当的代码，随后产生相应于<expression>的新的语义记录，其中包含了刚被处理过的表达式的必要信息。

使用递归下降语法分析器时，这些语义记录可以作为语法分析例程的局部变量存储。包含非终结符号语义信息的记录可作为结果参数由相应的语法分析例程返回。使用表驱动的语法分析器时，需要一个显式的语义栈（semantic stack），用以在语义例程调用之间存储语义记录，如第8章所述。

为了定义语义记录，检查CFG中的每个符号并根据情况决定一个符号究竟需要什么样的语义信息。图2-8给出翻译Micro所需的语义记录op_rec和expr_rec。（在expr_rec中首次使用匿名联合，它不属于标准C语言。）

```
#define MAXIDLEN        33
typedef char string[MAXIDLEN];

typedef struct operator {     /* for operators */
    enum op { PLUS, MINUS } operator;
} op_rec;

/* expression types */
enum expr { IDEXPR, LITERALEXPR, TEMPEXPR };

/* for <primary> and <expression> */
typedef struct expression {
    enum expr kind;
    union {
        string name;      /* for IDEXPR, TEMPEXPR */
        int val;          /* for LITERALEXPR */
    };
} expr_rec;
```

图2-8 Micro文法符号的语义记录

所有其他的符号不需要相关语义信息，因而它们的语义记录为空。考虑到效率，空记录不在任何地方显式定义和存储。然而，如果确定需要额外的数据，也可以向语义记录中添加新的域。在确定语义记录时，我们其实是在决定语义例程将含有什么参数。

40

2.5.5 Micro动作符号

图2-9列出一个包含动作符号的Micro的CFG。相比先前的文法，其中添加了一个产生式：

<ident> → ID #process_id

该产生式非常有用，因为在先前的文法里，ID出现在许多不同的上下文中，而我们需要在语法分析器匹配ID的任意一次出现之后立即调用process_id()（以访问token_buffer中的字符并创建适当的语义记录）。将文法中出现的所有ID替换为<ident>，就可以保证总会调用process_id()。

```
<program>          → #start begin <statement list> end
<statement list>   → <statement> {<statement>}
<statement>        → <ident> := <expression> #assign ;
<statement>        → read ( <id list> ) ;
<statement>        → write ( <expr list> ) ;
<id list>          → <ident> #read_id {, <ident> #read_id }
<expr list>        → <expression> #write_expr
                       {, <expression> #write_expr}
<expression>       → <primary>
                       {<add op> <primary> #gen_infix}
<primary>          → ( <expression> )
<primary>          → <ident>
<primary>          → INTLITERAL #process_literal
<add op>           → PLUSOP #process_op
<add op>           → MINUSOP #process_op
<ident>            → ID #process_id
<system goal>      → <program> SCANEOF #finish
```

图2-9 带有动作符号的Micro文法

我们的编译器中将利用一些辅助例程：generate()取四个字符串作为参数，分别对应操作码、两个操作数和结果域。它将在输出文件中产生经过正确格式化的指令。extract()取语义记录作为参数并返回其中包含的语义信息所对应的字符串。该字符串可以是一个标识符、操作码或文字常量等。提取出的信息传给generate()以创建一条完整的指令。

因为Micro很简单，所以符号表例程也很简单。例如，不需要存储任何类型信息作为标识符的属性，因为所有标识符都代表整型变量。由于产生的汇编语言指令可指示汇编程序为变量分配存储空间，因此也不需要记录任何地址信息作为标识符的属性。事实上，这里没有使用任何显式属性。对于标识符，我们感兴趣的惟一信息是它是否已经在符号表中，由此编译器将知道是否需要生成进行空间分配的指令。 41
符号表例程的规范为

```
/* Is s in the symbol table? */
extern int lookup(string s);

/* Put s unconditionally into symbol table. */
extern void enter(string s);
```

由许多语义例程所使用的辅助例程check_id()为：

```
void check_id(string s)
{
    if (! lookup(s)) {
        enter(s);
        generate("Declare", s, "Integer", "");
    }
}
```

lookup()将检查名为s的条目是否已在符号表中。除了符号的名字外，不需要在符号表中存储任何其他信息。enter()无条件地将字符串s加入符号表中。因此，如果需要，check_id()将通过把一个变量加入符号表并生成一条预留存储空间的汇编命令语句来声明变量。在我们的汇编语言中，Declare是向汇编程序声明名字并定义其类型的伪操作码。它针对简单的、非结构化的全局变量。汇编程序决定该变量需要多少空间以及具体存放在哪里。 42

我们需要一个例程来分配临时变量。如前所述，临时变量的分配类似于普通变量。惟一的不同是临时变量名将由编译器生成，并且对Micro程序并无特别的含义。它们的名字会是Temp&1、Temp&2等。在更实际的编译器中，临时变量通常被看作虚拟寄存器，代码生成器负责将它们尽可能地映射到实际的寄存器。函数get_temp()负责分配临时变量。

```
char *get_temp(void)
{
    /* max temporary allocated so far */
    static int max_temp = 0;
    static char tempname[MAXIDLEN];

    max_temp++;
    sprintf(tempname, "Temp&%d", max_temp);
    check_id(tempname);
    return tempname;
}
```

现在已经有了为定义相应于Micro动作符号的语义例程所必需的辅助例程。Micro的动作例程见图2-10。

```
void start(void)
{
    /* Semantic initializations, none needed. */
}

void finish(void)
{
    /* Generate code to finish program. */
    generate("Halt", "", "", "");
}

void assign(expr_rec target, expr_rec source)
{
    /* Generate code for assignment. */
    generate("Store", extract(source),
             target.name, "");
}
op_rec process_op(void)
{
    /* Produce operator descriptor. */
    op_rec o;

    if (current_token == PLUSOP)
        o.operator = PLUS;
    else
        o.operator = MINUS;
    return o;
}

expr_rec gen_infix(expr_rec e1, op_rec op,
                   expr_rec e2)
{
    expr_rec e_rec;
    /* An expr_rec with temp variant set. */
    e_rec.kind = TEMPEXPR;

    /*
     * Generate code for infix operation.
     * Get result temp and set up semantic record
     * for result.
     */
    strcpy(erec.name, get_temp());
    generate(extract(op), extract(e1),
             extract(e2), erec.name);
    return erec;
}

void read_id(expr_rec in_var)
{
    /* Generate code for read. */
    generate("Read", in_var.name,
             "Integer", "");
}
```

图2-10 Micro的动作例程

```
expr_rec process_id(void)
{
    expr_rec t;

    /*
     * Declare ID and build a
     * corresponding semantic record.
     */
    check_id(token_buffer);
    t.kind = IDEXPR;
    strcpy(t.name, token_buffer);
    return t;
}
expr_rec process_literal(void)
{
    expr_rec t;

    /*
     * Convert literal to a numeric representation
     * and build semantic record.
     */
    t.kind = LITERALEXPR;
    (void) sscanf(token_buffer, "%d", & t.val);
    return t;
}

void write_expr(expr_rec out_expr)
{
    generate("Write", extract(out_expr),
             "Integer", "");
}
```

图2-10 （续）

给定这些语义例程，现在再来看一下如何修改一个语法分析例程以包含语义处理。在图2-11中给出的过程expression()已被修改用来产生expr_rec作为输出参数。返回时，这个expr_rec中包含由expression()所识别的表达式的语义信息。过程体中包含内部变量，用来存储调用其他语法分析过程所生成的语义记录，并用来调用代码生成例程gen_infix()。（在C语言中使用指向受影响的语义记录的指针来实现输出参数。）

43 ~ 45

```
void expression(expr_rec *result)
{
    expr_rec left_operand, right_operand;
    op_rec op;

    primary(& left_operand);
    while (next_token() == PLUSOP ||
           next_token() == MINUSOP) {
        add_op(& op);
        primary(& right_operand);
        left_operand = gen_infix(left_operand, op,
                                 right_operand);
    }
    *result = left_operand;
}
```

图2-11 包含语义处理的语法分析过程

递归下降语法分析和翻译示例

作为示例，考虑下列简单Micro程序的编译：

```
begin A := BB - 314 + A ; end SCANEOF
```

以下是在处理该程序过程中语法分析器的分析步骤记录，连同在每一步剩余的输入和处理过程中可能生成的代码。相应的语法分析器动作可以是调用语法分析例程找出输入中的一个字符串来匹配文法中的非终结符，或者是调用match()来匹配文法终结符和一个输入词法记号，亦或是调用语义动作例程。

Step	Parser Action	Remaining Input	Generated Code
(1)	Call system_goal()	begin A:=BB–314+A ; end SCANEOF	
(2)	Call program()	begin A:=BB–314+A ; end SCANEOF	
(3)	Semantic Action: start()	begin A:=BB–314+A ; end SCANEOF	
(4)	match(BEGIN)	begin A:=BB–314+A ; end SCANEOF	
(5)	Call statement_list()	A:=BB–314+A ; end SCANEOF	
(6)	Call statement()	A:=BB–314+A ; end SCANEOF	
(7)	Call ident()	A:=BB–314+A ; end SCANEOF	
(8)	match(ID)	A:=BB–314+A ; end SCANEOF	
(9)	Semantic Action: process_id()	:=BB–314+A ; end SCANEOF	Declare A,Integer
(10)	match(ASSIGNOP)	:=BB–314+A ; end SCANEOF	
(11)	Call expression()	BB–314+A ; end SCANEOF	
(12)	Call primary()	BB–314+A ; end SCANEOF	
(13)	Call ident()	BB–314+A ; end SCANEOF	
(14)	match(ID)	BB–314+A ; end SCANEOF	
(15)	Semantic Action: process_id()	–314+A ; end SCANEOF	Declare BB,Integer
(16)	Call add_op()	–314+A ; end SCANEOF	
(17)	match(MINUSOP)	–314+A ; end SCANEOF	
(18)	Semantic Action: process_op()	314+A ; end SCANEOF	
(19)	Call primary()	314+A ; end SCANEOF	
(20)	match(INTLITERAL)	314+A ; end SCANEOF	
(21)	Semantic Action: process_literal()	+A ; end SCANEOF	
(22)	Semantic Action: gen_infix()	+A ; end SCANEOF	Declare Temp&1,Integer Sub BB,314,Temp&1
46 (23)	Call add_op()	+A ; end SCANEOF	
(24)	match(PLUSOP)	+A ; end SCANEOF	
(25)	Semantic Action: process_op()	A ; end SCANEOF	
(26)	Call primary()	A ; end SCANEOF	
(27)	Call ident()	A ; end SCANEOF	
(28)	match(ID)	A ; end SCANEOF	
(29)	Semantic Action: process_id()	; end SCANEOF	
(30)	Semantic Action: gen_infix()	; end SCANEOF	Declare Temp&2,Integer Add Temp&1,A,Temp&2
(31)	Semantic Action: assign()	; end SCANEOF	Store Temp&2,A
(32)	match(SEMICOLON)	; end SCANEOF	
(33)	match(END)	end SCANEOF	
(34)	match(SCANEOF)	SCANEOF	
(35)	Semantic Action: finish()		Halt

为程序

```
begin A := BB – 314 + A ; end SCANEOF
```

生成的代码总结如下。很容易验证它是正确的。

```
Declare A,Integer
Declare BB,Integer
Declare Temp&1,Integer
Sub BB,314,Temp&1
Declare Temp&2,Integer
Add Temp&1,A,Temp&2
Store Temp&2,A
Halt
```

练习

1. 使用C语言的switch语句代替if和else序列重写图2-3中的C代码。你喜欢哪一个版本？为什么？
2. 为什么不能使用EOF作为枚举文字常量代替SCANEOF？
3. 在一个没有相应工具[⊖]支持的语言（例如Pascal、Modula-2）中，用超前搜索（lookahead）或者压回（pushback）方式实现单字符输入。紧记需要行结束标记和文件结束标记。
4. 实现保存词法记号字符串所需的例程buffer_char()和clear_buffer()，以及识别保留字的例程check_reserved()。实现时紧记C语言字符串的特性。 47
5. 如图2-9所示，在Micro文法中加入非终结符<ident>将需要修改一些递归下降语法分析例程。重写所有为实现这个改动而需修改的例程。
6. 以图2-11中的expression()过程为范例，为所有语法分析例程添加所需的语义处理代码。
7. 实际的语法分析器必须以某种方式对语法分析中遇到的语法错误做出反应。Micro语法分析器采用递归下降语法分析，需要为文法中的每个非终结符都编写一个单独的语法分析过程。这种语法分析技术潜在地要求将错误处理集成在每个语法分析过程中。从对Micro的语法分析过程的检查来看，显然可以有两种方式发现语法错误：match()可能无法找到源程序中的正确词法记号，或者一个语法分析过程在switch或if语句中检查next_token()时可能无法找到可接受的记号。在后一种情况下，Micro语法分析例程调用syntax_error()。match()和syntax_error()必须被设计用来执行某些动作以允许语法分析继续进行。

   match()可以被放在一个布尔函数中以指示是否在输入中找到了所需的词法记号，但这样的改动会极大地增加每个调用它的语法分析过程的复杂性。简单的变通办法是让match()遇到错误时假装它看到了正确的词法记号。在这种情况下，必须决定match()是否应当像成功匹配时那样，消耗它在词法记号流中找到的不正确的词法记号。这两种方法有什么含义？其间又有什么权衡？

   如果遇到第二种语法错误，则不可能有这样简单的处理机制，因为任意词法记号集都可以被接受以继续分析。处理这样的错误需要显式地重新编写一些语法分析过程。请提出一个通用的方法修改像statement()这样的过程以处理语法错误。
8. 即使在一个像Micro编译器一样简单的编译器中，我们也可以实现一种称为*常量合并*（constant folding）的优化，即在编译时刻计算常量表达式的值。如果一个表达式的两个操作数都是常量，则不需要生成计算该表达式的代码，因为它的值可由编译器确定。找出并适当地修改与实现常量合并相关的语义动作例程。
9. 假定你在编写一个Micro的解释器而不是编译器。解释器将在进行语法分析时执行Micro代码而不是生成汇编语言供以后执行。必须怎样修改语义动作例程和语义记录以支持解释执行？
10. 对Micro的一个有用的扩展是包含一种新型的表达式，*条件表达式*（conditional expression），其语法为：$(E_1 | E_2 | E_3)$。当计算该表达式时，如果E_1非零则返回E_2作为表达式的值；否则返回E_3作为表达式的值。 48

    (a) 写出适当的产生式，以便为Micro添加条件表达式，其中包括动作符号以指定对所需动作例程的调用。

    (b) 假定有一条新的汇编语言指令Skip A，如果操作数A取非零值则导致忽略下一条指令。写出实现条件表达式所需的动作例程。 49

    (c) 别忘了同时扩充词法分析器！

⊖ 如C语言中的ungetc()函数。——译者注

第3章　词法分析——理论和实践

3.1 概述

词法扫描器（scanner）的主要功能是将输入字符分组为词法记号。词法扫描器有时也称为*词法分析器*（lexical analyzer），这两个术语可以互换使用[⊖]。第2章中的Micro词法分析器十分简单，很容易由任何称职的程序员编码实现。现在，我们深入研究的问题是为更全面的程序设计语言创建词法分析器。 50

本章将介绍用来指定词法记号精确结构的*形式表示*（formal notation）。乍看，这样做似乎没有必要，因为大多数程序设计语言中的词法记号结构都很简单。这里的问题是词法记号可能比我们所期望的更复杂。例如，大家都熟悉Pascal中由引号引用的简单字符串。字符串中可包含除引号外的任意字符序列（其中若出现引号，则必须用两个引号表示）。但这个简单定义真的正确么？换行符可以出现在字符串中么？也许不能，因为跨行的字符串难以阅读。如果允许包含换行符，结尾缺少引号的“失控字符串”将更加难以检测。C语言中包含换行符，并允许被转义的换行符出现在字符串中，而Pascal语言禁止换行符出现在字符串中。Ada则进一步禁止字符串中出现任何不可打印字符（正因为它们通常不可读）。类似地，允许出现（长度为零的）空字符串吗？Pascal不允许，因为在Pascal中字符串是一个*压缩型字符数组*（packed array of characters），而零长度的数组是不允许出现的。相反，Ada和C则允许空字符串。

为保证实施词法规则，词法记号的严格定义显然是必需的。形式定义也使得语言设计者能够预见设计缺陷。例如，实际上所有语言都允许定点十进制数，像0.1或10.01。但允许.1或10.吗？在FORTRAN和C中允许，但有趣的是，在Pascal和Ada中不允许。词法分析器通常试图使一个词法记号尽可能长，例如，ABC被扫描分析为一个标识符而不是三个标识符。现在考虑字符序列1..10。在Pascal和Ada中希望它被解释为区间说明符（1到10）。如果在词法定义中不小心，会把1..10扫描分析为两个实数文字常量，1.和.10，并将导致直接的（而且是意想不到的）语法错误。（在CFG中反映出两个实数不能相邻这个事实，是由语法分析器而不是词法分析器要求的。）

给定了词法记号和程序结构的形式规范，就有可能检查一个语言是否有设计缺陷。例如，考虑所有可以相邻的词法记号对，确定这两个词法记号是否可能被错误地扫描。如果是，则需要一个分隔符（如针对相邻的标识符和保留字的情形），否则词法或程序语法就需要重新设计。关键是语言设计比我们所预期的更为棘手，而形式规范允许在设计完成前发现缺陷。

所有词法分析器，不论它识别什么样的词法记号，都执行相同的功能。因此，从零开始编写词法分析器意味着重新实现所有词法分析器的公共组件，是大量的重复劳动。*词法分析器生成器*（scanner generator）的目标是通过指定词法分析器识别哪些词法记号来减少构造词法分析器的努力。利用形式表示，告诉词法分析器生成器我们想识别哪些词法记号；产生符合我们规范的词法分析器则是生成器的责任。一些生成器不产生完整的词法分析器，而是产生可与标准驱动程序一起使用的表格。将生成的表格和标准驱动程序结合起来可以产生想要的定制词法分析器。 51

使用词法分析器生成器编程是*一种非过程式程序设计*（nonprocedural programming）。也就是说，它不同于普通的称为*过程式的程序设计*（procedural programming），我们不告诉词法分析器生成器怎样扫描

⊖ 在本书中，统一将“scanner”译为“词法分析器”。——译者注

而是简单地告诉它我们想要扫描什么。在各种方法中，这是一种更自然的高级方法。近来，有相当一部分计算机科学研究都指向非过程式程序设计风格。（数据库查询语言和Prolog（一种“逻辑”程序设计语言）都是非过程式的。）非过程式程序设计最成功地用于有限领域，例如词法分析，其中必须自动做出的实现决定的范围是有限的。尽管如此，计算机科学家的长期（且尚未实现的）目标是从源语言特性和目标计算机的规范出发产生完整的编译器。

在后面各节中，首先介绍*正则表达式*（regular expression）的表示，它非常适合词法记号的形式化定义。其次，研究正则表达式和*有限自动机*（finite automata）之间的对应关系。有限自动机特别有用，因为它们能够被“执行”以读入字符并将其分组为词法记号。本章还将详细讨论两个词法分析器生成器，*ScanGen*和*Lex*。它们将（以正则表达式形式给出的）词法记号定义作为输入。ScanGen产生可由小型词法分析器驱动程序使用的表格。Lex则产生完整的词法分析子程序，以备编译和使用。接下来，要讨论的话题是在创建词法分析器以及将其和编译器的其他部分集成时所要考虑的一些实际问题。这些问题包括：预见可能使词法分析复杂化的词法记号和上下文，以及从词法错误中恢复。本章的最后一节解释词法分析器生成器如何将正则表达式翻译成有限自动机。读者若希望简单地将词法分析器生成器视为一个黑箱，则可以跳过该节。尽管如此，这些材料可用来补充前面介绍的正则表达式和有限自动机的概念。该节还说明如何构造、合并、简化甚至优化有限自动机。

3.2 正则表达式

正则表达式是描述某些简单（尽管可能是无限的）字符串集合的便利手段。正则表达式有着实际的重要性，因为它们可以用来描述程序设计语言中所使用的词法记号的结构。特别是，正则表达式可用来为词法分析器生成器编制程序。

由*正则表达式*（regular expression）所定义的字符串集称为*正则集*（regular set）。我们从有限字符集或*词汇表*（由V表示）开始。词汇表通常是用来形成词法记号的字符集。允许出现空字符串（由λ表示，读作“lambda”）。字符串由V中的字符通过*连接操作*（catenation）构造（例如，:=, begin, 123）。空字符串与任何字符串s连接，生成s。即，$s\lambda = \lambda s = s$。

52 可按以下方式将连接操作扩展到字符串集上：令P和Q是字符串集。则当且仅当s可被拆分成两部分：$s = s_1 s_2$，其中$s_1 \in P$且$s_2 \in Q$时，$s \in (P\ Q)$。小的有限集可通过列出它们的元素方便地表示。这些元素可以是单个字符或字符串。括号用来分隔表达式，选择操作符“|”用来分隔不同的选项。

字符“(”、“)”、“'”、“*”、“+”和“|”是*元字符*（meta-character），作为标点符号和正则表达式操作符使用。引号中的元字符表示普通字符，以避免二义性。（任何字符或字符串都可以由引号引用，但通常避免使用不必要的引号，以增强可读性。）例如：

```
Delim = ('('|')'|':='|';'|'+'|'-'|'*'|'/'|'='|$$)
```

可将选择操作扩展到字符串集上。令P和Q是字符串集。则当且仅当$s \in P$或$s \in Q$时，$s \in (P|Q)$。大的（或无限的）集合可通过有限字符和字符串集的连接操作和选择操作方便地表示。此外，还允许使用第三种操作，*Kleene闭包*（Kleene closure）。操作符*用来表示Kleene闭包操作符。令P为一个字符串集。当且仅当字符串s可被拆分为零个或多个部分：$s=s_1\ s_2\ s_3 \cdots s_n$，使得每个$s_i \in P$时，$s \in P^*$。$P^*$中的字符串是来自P的（可能重复的）零个或多个选项的连接。（明确允许$n = 0$，因此λ总在P^*中。）

*正则表达式*定义如下。每个正则表达式由一个字符串集（*正则集*）表示。

- Ø是正则表达式，它表示空集（该集合中不包含字符串）。
- λ是正则表达式，它仅包含空字符串。注意：该集合与空集不同，因为它包含一个元素。
- 字符串s是表示仅包含s的正则表达式。如果s包含元字符，可以将s放在引号中以避免二义性。

- 如果A和B是正则表达式，则A|B、A B和A*也是正则表达式，分别表示相应正则集的选择、连接和Kleene闭包。

任意有限字符串集可由形如$(s_1|s_2|\cdots|s_k)$的正则表达式表示。

通常使用下列操作作为简便记法。它们不是必需的，因为（尽管稍有点笨拙）也可以通过三种标准正则操作符（选择、连接、Kleene闭包）达到这样的效果：

- P^+表示由P中的一个或多个字符串连接而成的所有字符串：$P^* = (P^+|\lambda)$且$P^+ = P\ P^*$。
- 如果A是一个字符集，Not(A)表示(V − A)；即，所有在V但不在A中的字符。由于Not(A)是有限的，因此它是平凡正则的。可以将Not扩展到字符串而不仅限于V。即，如果S是字符串集，可以定义Not(S)为$(V^* - S)$。尽管该集合可能是无限的，但它仍然是正则的（见练习20）。 53
- 如果k是常数，集合A^k表示由A中的（不必是不同的）字符串连接k次所形成的所有字符串。即，$A^k = (A\ A\ A\ \cdots)$（k个A）。

现在说明如何用正则表达式指定词法记号。令

D = (0 | ··· | 9) L = (A | ··· | Z)

则

- 以“--”开始，以Eol（行结束符）结束的注释可定义为：

  Comment = -- Not(Eol)* Eol

- 定点十进制文字常量可定义为：

  Lit = D+ . D+

- 标识符由字母、数字和下划线组成，以字母开始，以字母或下划线结尾，且不包含连续的下划线。标识符可定义如下：

  ID = L (L | D)* (_ (L | D)+)*

- 更复杂的例子是由“##”标记限定的注释，允许在注释体中出现单个的“#”：

  Comment2 = ## ((# | λ) Not(#))* ##

所有有限集和许多无限集，例如刚刚列出的那些无限集，都是正则的。但并非所有无限集都是正则的。例如，考虑$\{[^i]^i \mid i > 1\}$，该集合是形如[[[[[···]]]]]的配对方括号集。这是一个著名的非正则集。问题是任何正则集要么不包含所有的配对嵌套，要么包含多余的不必要的字符串。（练习16对此进行证明。）

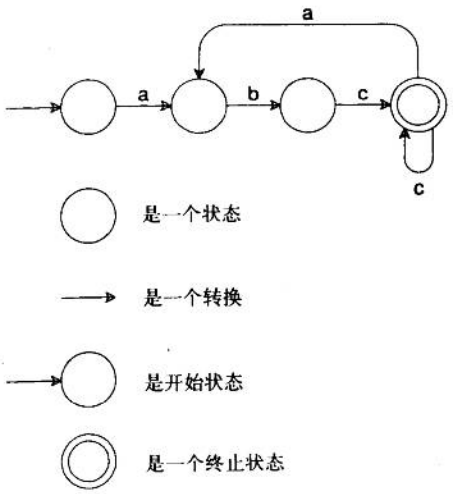
很容易写出一个CFG，它精确定义了配对方括号。所有正则集都可由CFG定义。因此，方括号的例子表明CFG是比正则表达式更为强大的描述机制。正则表达式完全足够用来描述词法记号级语法。 54

3.3 有限自动机和词法分析器

有限自动机（Finite Automaton, FA）用来识别由正则表达式定义的词法记号。FA是一个简单的理想化的计算机，可用来识别属于正则集的字符串。它包含：

- 一个有限状态集。
- 从一个状态到另一个状态的转换（或移动）集，标记V中的字符。
- 一个特殊的开始状态。
- 一组终止或接受状态。

可以用转换图来图形化地表示有限自动机：



在这些图中，从开始状态开始进行分析。如果下一个输入字符匹配从当前状态出发的转换标签，则转移到该转换所指向的状态。如果没有可能的转换，则停止。如果在一个终止状态结束，则所读入的字符序列是一个有效的词法记号；否则就不是一个有效的记号。如图所示，有效的词法记号是由正则表达式(a b (c)*)+所描述的字符串。

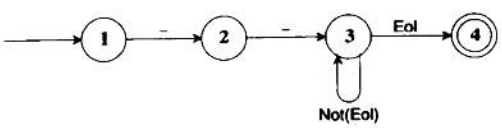
一个转换可标以多个字符（例如，Not(c)）作为简写。如果当前输入字符匹配标记转换的任何字符，就可以发生转换。

如果一个FA（对于给定状态和字符）总有惟一转换，则该FA是*确定的*（即，一个确定的FA或DFA）。确定的有限状态机通常用来驱动词法分析器。在计算机中通常用*转换表*（transition table）来表示DFA。

55 一个转换表T由DFA状态和词汇表符号索引。表项是一个DFA状态或错误标志。如果在状态s，读入字符c，则T[s][c]将是下一个要访问的状态，或者是一个错误标志，指示c不能作为当前词法记号的一部分。例如，正则表达式

```
-- Not(Eol)* Eol
```

定义一个Ada语言的注释，可被转换为



相应的转换表为

状态	字符				
	-	Eol	a	b	···
1	2				
2	3				
3	3	4	3	3	3
4					

在该表中，错误项为空。完整的转换表中针对每个字符将包含相应的一列。为节省空间，常常使用某些形式的表压缩；这将在3.4.1节深入讨论。

任意正则表达式都可被转换为一个DFA，这个DFA接受由该正则表达式表示的字符串集（作为有效的词法记号）。该转换可由以下方式完成：

- 由词法分析器生成器自动完成。
- 由程序员手工完成。

DFA可被实现为由一个驱动程序“解释”的转换表，或“直接”实现为程序的控制逻辑（每条语句对应一个DFA状态）。例如，假定current_char为当前输入字符，使用上面介绍的识别Ada注释的DFA，这两种方法可产生图3-1和图3-2中列出的程序。

```
/*
 * Note: current_char is already set to
 * the current input character.
 */
state = initial_state;
while (TRUE) {
    next_state = T[state][current_char];
    if (nextstate == ERROR)
        break;
    state = next_state;
    if (current_char == EOF)
        break;
    current_char = getchar();
}
if (is_final_state(state))
    /* Return or process valid token. */
else
    lexical_error(current_char);
```

图3-1 解释转换表的词法分析器驱动器

```
if (current_char == '-') {
    current_char = getchar();
    if (current_char == '-') {
        do
            current_char = getchar();
        while (current_char != '\n');
    } else {
        ungetc(current_char, stdin);
        lexical_error(current_char);
    }
}
else
    lexical_error(current_char);
/* Return or process valid token. */
```

图3-2 含固定词法记号定义的词法分析器

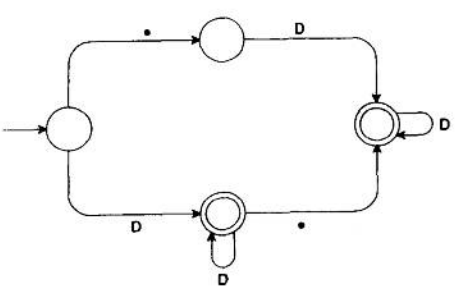
第一种形式通常和词法分析器生成器一起使用，它是与语言无关的。如果已正确设置了转换表（存放在T中），则该形式是一个能够扫描任何词法记号的简单驱动器。后一种形式通常手工产生。在这里，要扫描的词法记号被“硬编码”到程序中。 56

下面是正则表达式和它们所对应的DFA的另外两个例子：

- FORTRAN式的实数文字常量（在小数点一边或两边都需要数字，或仅是一个数字串）可定义为 57

$$RealLit = (D^+ (\lambda \mid .)) \mid (D^* . D^+)$$

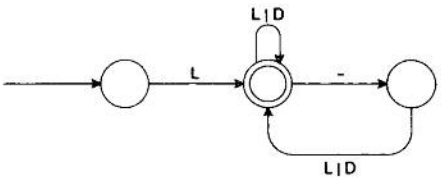
它对应DFA



• 由字母、数字和下划线组成的标识符，其中不含相邻下划线或后置下划线，可定义为

ID = L (L | D)* (_ (L | D)+)*

相应的DFA为



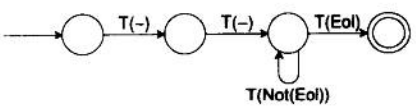
可为FA添加输出机制，使FA成为**转换器**（transducer）。在读入字符时，它们可被转换并连接到输出字符串中。对我们的目的来说，应当将转换操作限制为保存或删除输出字符。一个词法记号被识别后，被转换过的输入可被传递给其他编译阶段进行进一步处理。使用这个词法记号：

a ——→ 表示把a保存到词法记号缓存中

58

T(a) ——→ 表示不保存a（将它抛弃）

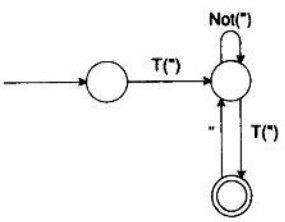
例如，对于注释，可以写



一个更有趣的例子是根据正则表达式，由引号中的字符串给出的

(" (Not(") | "")* ")

相应的转换器可以是



输入"""Hi"""会产生输出"Hi"。

3.4 使用词法分析器生成器

在本节中，将说明两个流行的词法分析器生成器ScanGen和Lex的使用。ScanGen和Lex的完整描述可在附录B和Lesk and Schmidt（1975）中找到。这里，我们的目的是说明如何将正则表达式及其相关信息呈现给生成器。学习使用词法分析器生成器的一个好办法是：从这里介绍的简单例子开始并逐步将它们推广以解决手头面临的问题。对于没有经验的读者，词法分析器规范会显得晦涩难懂。最好紧记问题的关键总是作为正则表达式的词法记号规范，其余的内容则仅是为了提高效率并处理各种细节。

3.4.1 ScanGen

ScanGen是一个易于移植的、高度独立于语言和机器的词法分析器生成器。ScanGen产生由驱动程

序使用的表格，继而创建一个完整的词法分析器。ScanGen驱动程序的一般结构将在下一节讨论。ScanGen被用于世界上许多大学和研究实验室中。

59

我们将学习在ScanGen中如何通过检查图3-3中所示的定义来定义词法记号。该定义扩展了Micro的词法记号集，其中包含实数和字符串文字常量以及一个用来处理失控字符串的特殊的“错误词法记号”。

```
Options
    List, tables, optimize

Class
    E           = 'E', 'e';
    OtherLetter = 'A'..'D','F'..'E','a'..'d','f'..'z';
    Digit       = '0'..'9';
    Blank       = ' ';
    Dot         = '.';
    Plus        = '+';
    Minus       = '-';
    Equal       = '=';
    Colon       = ':';
    Comma       = ',';
    Semicolon   = ';';
    Lparen      = '(';
    Rparen      = ')';
    Quote       = '"';
    Underscore  = '_';
    Tab         = 9;
    Linefeed    = 10;

Definition
    Token EmptySpace {0} = (Blank, Linefeed, Tab)+;
    Token Comment {0} =
        Minus . Minus . (Not(Linefeed))* . Linefeed;
    Letter = E, OtherLetter;
    Token Identifier {1} = Letter.(Letter,Digit,Underscore)*
        Except
            'Begin' {4},
            'End' {5},
            'Read' {6},
            'Write' {7};
    Token IntLit {2,1} = Digit+;
    Token RealLit {2,2} = IntLit.Dot.IntLit.
          (Epsilon, E.(Epsilon, Plus, Minus).IntLit);
    Token StrLit {2,3} = Quote{Toss}.
         (Not(Quote,Linefeed),Quote{Toss}.Quote)* . Quote{Toss};
    Token RunOnStringLit {3} = Quote{Toss}.
         (Not(Quote, Linefeed), Quote{Toss}.Quote)*
              . Linefeed{Toss};
    Token LparenToken {8} = Lparen;
    Token RparenToken {9} = Rparen;
    Token SemicolonToken {10} = Semicolon;
    Token CommaToken {11} = Comma;
    Token AssignOp {12} = Colon . Equal;
    Token PlusOp {13} = Plus;
    Token MinusOp {14} = Minus;
```

图3-3 扩展Micro的ScanGen定义

ScanGen的输入分为三节，每节由一个保留字打头。第一节指定由ScanGen使用的选项。在我们的示例中，需要列出输入列表，产生输出表格，并包含一个状态优化阶段。

第二节定义字符类（character class）。在对有限自动机的讨论中，假定状态转换是由有限词汇表中的字符进行标记。在实践中，该词汇表是某个计算机字符集，可能是ASCII或EBCDIC。由于用于定义FA的转换表的大小是状态数和字符数的乘积，因此字符集大小是一个重要因素。

很容易看出自然落入字符类中的字符，其中一个类中的所有字符在正则表达式或FA中都被同等对待。在字符类一节中根据字符序列或字符范围定义类名。对应单个字符的类当然是被允许的，而与字符等价的十进制数用于不可打印字符。在任何字符类定义中都未提及的字符将被忽略。如果愿意，可定义一个“非法”类，强迫产生词法分析器错误（和错误信息）。所有正则表达式和有限自动机都使用字符类来定义。ScanGen产生一个向量，将字符映射到字符类。使用字符类使得正则表达式更易于阅读，并

（极大地）减小了转换表的大小。

对于一个典型的程序设计语言，ScanGen产生约50个状态和30个字符类。假定每个条目占两个字节（一个用于action标志，一个用于next_state值），这种表示方法意味着使用普通数组结构存放的一个转换表的大小约为3000字节。因为对每个输入字符都要查询转换表，所以对表项的快速访问非常重要。因此，如果可以满足普通二维数组的空间需求，则该格式将是词法分析器转换表的最佳表示方法。

尽管如此，压缩转换表仍然可能是必要的。在第17章将讨论许多表压缩技术。这些技术可用于缩小转换表的尺寸，但需要谨慎行事。特别地，访问压缩表的额外开销可能超过空间节省所带来的益处。在采用任何压缩方案前先估算速度和空间开销是一种明智的做法。

ScanGen的最后一节使用正则表达式定义词法记号，其中使用中缀“.”（连接操作）和“,”（选择操作）、后缀“*”（Kleene闭包）和“+”（正闭包）以及一元Not。连接操作作为显式操作用来增强可读性并简化正则表达式的翻译。Epsilon表示匹配λ的正则表达式。（在一些教材中以ε表示空串，而我们在本书中使用λ。）不直接定义词法记号的正则表达式（例如Letter）可以用来在定义其他正则表达式时使用。

60
~
61

词法记号定义的形式为

```
Token name{major,minor} = regular expression ;
```

语法分析器认为一个词法记号的所有实例都是等价的，但有时同样的词法记号的子类由不同的正则表达式识别。在图3-3中，使用三个不同的正则表达式来识别整数、实数和字符串文字常量。词法记号子类可以有不同的语义解释。例如，不同的文字常量类需要以不同的方式转换到内部形式。因此，词法记号描述的一个有用特性是同时以*主词法记号代码*（major token code）和*次词法记号代码*（minor token code）标记词法记号的能力。主词法记号代码识别语法分析器所需的词法记号，次词法记号代码则可由语义处理例程使用。在词法记号定义中，主代码和次代码都是整型值。次代码是可选的（它默认为零）。

相同的字符序列可以被多个正则表达式匹配。在这种情况下，在规范中较早列出的正则表达式优先。它的主、次代码不变。

一些词法记号可能是“噪音记号”（noise token）（像注释），它们（一旦被识别就）必须被删除而不是返回给语法分析器。根据ScanGen中的约定，为零的主代码指示相应的词法记号将会被删除而不是被传给语法分析器。定义一个*空白词法记号*（由空白符、制表符和换行符组成）以消耗感兴趣的词法记号之间的空白通常是有用的。在图3-3中，词法记号EmptySpace即用于此目的。当然，它是标记为需要被删除的。

词法记号定义可以有一个*异常子句*（exception clause，以Except开始）命名一列异常。每个异常（由含主、次代码的文字常量定义）必须匹配相关的正则表达式。ScanGen产生经过排序的一列异常（适于二分查找）以及一个标志指示该词法记号定义含有异常。当找到匹配该表达式的词法记号时，词法分析器驱动程序必须搜索异常列表以确定识别了什么词法记号。

异常列表适于许多不同的词法记号有相似词法结构的情况。该问题的一个例子是保留字通常和标识符在词法上重叠。将保留字作为标识符的异常来处理简化了规范，同时也能减小词法分析器所需的FA的尺寸。该问题会在3.5节中进一步讨论。

在正则表达式中，字符类的名字或Not表达式可以由{Toss}作为后缀。当在某个正则表达式中匹配以{Toss}标记的字符类（或其互补类）的名字时，匹配的字符被丢弃而不是被保存。所有未以{Toss}标记的字符类的名字和Not表达式被隐含地假定保存它们所匹配的任意字符。

ScanGen通常针对ASCII或EBCDIC字符集进行配置。然而，也可以通过增加字符集尺寸参数来编译它使之可接受伪字符。这样就能以简洁的方式处理文件结束条件。特别地，当文件结束时，读字符例程可以返回一个由Eof词法记号所定义的Eof伪字符。（在第2章中，Eof词法记号由SCANEOF表示。C语言

62

标准I/O库称之为EOF。)

除此之外，还可以修改词法分析器驱动程序，以便在查阅词法分析器转换表前测试文件结尾。如果在调用词法分析器时文件结尾标志为真，则立即返回Eof记号。如果在进行词法分析的过程中文件结束标志为真，则当前词法记号完成并返回。下次调用词法分析器时将返回Eof记号。

ScanGen驱动程序

在本节简要介绍驱动程序例程，它们可以和ScanGen产生的表格一起使用以实现词法分析器。

通常，在进行词法分析时FA必须超前搜索。也就是说，它必须检查一个有可能不属于当前词法记号一部分的字符，以证实已经看到了一个完整的词法记号。例如，在扫描一个标识符时，必须保持读入，直到看到一个不属于当前标识符的字符（比如一个空白符或“;”）。我们必须当心不要丢掉超前搜索的字符，因为可能需要它作为下一个要扫描的词法记号的开头。为控制对输入字符的使用，我们使用来自C语言标准I/O库的两个例程:

```
#include <stdio.h>
/* getc() is usually a macro */
extern int getc(FILE *);
extern int ungetc(int, FILE *);
```

getc()返回当前正在处理的输入字符并将文件位置指针移到下一个字符，从而“消耗”掉当前字符。ungetc()将一个字符放回当前正在读取的文件，因此下一次调用getc()将返回该字符，而文件位置指针并未改变。C语言标准I/O库可保证完成一个字符的“压回”操作。

有时，超前搜索一个字符和压回一个字符是不够的。在这种情况下，词法分析器必须实现它自己的缓冲机制。不过，这通常是很简单的。

在Pascal语言中，词法分析器驱动程序以不同的方式操纵它的输入。通过input^查看当前字符并随后当字符有效时通过get(input)消耗它。

ScanGen表格的确切格式在附录B中详述。控制词法分析的表格称为action表。action[state][ch]可指示一个*移动*（move）动作（继续扫描）或*停止*（halt）动作（已经识别了一个词法记号）。因为可能需要查阅超前搜索的字符，而且扫描过的字符可能需要被丢弃或者保留，所以动作表包含6种不同的值:

(1) ERROR

词法记号错误。没有可识别的有效词法记号。

(2) MOVEAPPEND

移动到下一个状态。消耗当前字符并将其添加到正在构造的词法记号串末尾。

(3) MOVENOAPPEND

移动到下一个状态。消耗当前字符但不将其添加到正在构造的词法记号串末尾。

(4) HALTAPPEND

消耗当前字符并将其添加到正在构造的词法记号字符串末尾。已经找到了一个有效的词法记号。

(5) HALTNOAPPEND

消耗当前字符但不将其添加到正在构造的词法记号字符串末尾。已经找到了一个有效的词法记号。

(6) HALTREUSE

不消耗当前字符。已经找到了一个有效的词法记号。

对于移动动作，下一个要访问的状态被存放在表格的next_state[state][ch]位置。零指示没有下一个状态。

对于停止动作，通过调用下面的过程获得主、次代码:

```
extern void lookup_codes(state current_state,
        char cur_char, codes *major, codes *minor);
```

在查找了主、次代码后，通过调用下面的过程来处理异常：

```
extern void check_exceptions(codes *major,
        codes *minor, char *token_text);
```

如果major和minor指示一个拥有异常的词法记号类，则检查token_text以确定它是否真的是一个异常。如果token_text不是异常，则返回原始的代码。

在图3-4中列出了一个ScanGen驱动程序。

```
#define reset() { ind = 0; \
    token_text[ind] = '\0'; state = STARTSTATE; }
extern enum scan_state next_state[NUMSTATES][NUMCHARS];
extern FILE *srcfile;
void scanner(codes *major, codes *minor,
             char *token_text)
{
    /*
     * major will always be set. minor and
     * token_text may not be, depending on
     * whether a minor code is used, and whether
     * token characters are saved or tossed.
     */
    enum scan_state state;
    int ind;
    int c;
    reset();
    while (TRUE) {
        c = getc(srcfile);
        switch (action[state][c]) {
        case ERROR:
            /*
             * Do lexical error recovery.
             * ungetc(c, srcfile) may or may
             * not be necessary.
             */
            break;
        case MOVEAPPEND:
            state = next_state[state][c];
            token_text[ind++] = c;
            break;
        case MOVENOAPPEND:
            state = next_state[state][c];
            break;
        case HALTAPPEND:
            lookup_codes(state, c, major, minor);
            token_text[ind++] = c;
            token_text[ind] = '\0';
            check_exceptions(major, minor, token_text);
            if (*major == 0) {
                /* Do not return this token. */
                reset();
                continue;
            }
            return;
    case HALTNOAPPEND:
        lookup_codes(state, c, major, minor);
        token_text[ind] = '\0';
        check_exceptions(major, minor, token_text);
        if (*major == 0) {
            /* Do not return this token. */
            reset();
            continue;
```

图3-4 一个ScanGen驱动程序

```
            }
            return;

        case HALTREUSE:
            lookup_codes(state, c, major, minor);
            token_text[ind] = '\0';
            check_exceptions(major, minor, token_text);
            ungetc(c, srcfile);
            if (*major == 0) {
                /* Do not return this token. */
                reset();
                continue;
            }
            return;
        } /* end switch */
    } /* end while */
}
```

图3-4 （续）

3.4.2 Lex

Lex是一个由AT&T贝尔实验室的M.E Lesk和E. Schmidt开发的词法分析器生成器。它运行于UNIX操作系统下，主要和用C语言写的程序一起使用。Lex的原始版本也可运行在GCOS和OS/370下，而且能够产生用Ratfor语言和C语言编码的词法分析器。Lex的当代版本仅在UNIX操作系统下可用，且仅产生用C语言编码的词法分析器。（生成的词法分析器不仅限于UNIX环境。）

Lex产生一个完整的词法分析器模块，它可被编译并与其他编译器模块连接。Lex所使用的方法比ScanGen更广泛。特别地，Lex将正则表达式和任意的代码段相关联。当一个表达式被匹配时，将执行相应代码段。Lex不提供像主/次代码或toss标志这样的特殊功能，因为所有这些（以及更多的功能）可以在用户编写的代码段中实现。图3-5说明一个词法分析器的Lex定义，它与图3-3中的定义等价。

64

```
E               [Ee]
OtherLetter     [A-DF-Za-df-z]
Digit           [0-9]
Letter          {E} | {OtherLetter}
IntLit          {Digit}+
%%
[ \t\n]+                                        { /* delete */          }
[Bb][Ee][Gg][Ii][Nn]                            { minor=0; return(4);  }
[Ee][Nn][Dd]                                    { minor=0; return(5);  }
[Rr][Ee][Aa][Dd]                                { minor=0; return(6);  }
[Ww][Rr][Ii][Tt][Ee]                            { minor=0; return(7);  }
{Letter}({Letter} | {Digit} | _)*                { minor=0; return(1);  }
{IntLit}                                        { minor=1; return(2);  }
({IntLit}[.]{IntLit})({E}[+-]?{IntLit})?        { minor=2; return(2);  }
\"([^\"\n] | \"\")*\"                           { stripquotes();
                                                  minor=3; return(2);  }
\"([^\"\n] | \"\")*\n                           { stripquotes();
                                                  minor=0; return(3);  }
"("                                             { minor=0; return(8);  }
")"                                             { minor=0; return(9);  }
";"                                             { minor=0; return(10); }
","                                             { minor=0; return(11); }
":="                                            { minor=0; return(12); }
"+"                                             { minor=0; return(13); }
"-"                                             { minor=0; return(14); }
%%

/* Strip unwanted quotes from string in yytext; adjust yyleng. */
void stripquotes(void)
{
    int frompos, topos = 0, numquotes = 2;
```

图3-5 扩展Micro的Lex定义

```
        for (frompos = 1; frompos < yyleng; frompos++) {
            yytext[topos++] = yytext[frompos];
            if (yytext[frompos] == '"' && yytext[frompos+1] == '"') {
                frompos++;
                numquotes++;
            }
        }
        yyleng -= numquotes;
        yytext[yyleng] = '\0';
    }
```

图3-5 （续）

像在ScanGen中一样，首先定义字符类和辅助正则表达式。这些定义在第一节完成。（节由“%%”分隔符分隔。）字符类由“[”和“]”界定。除“\”、“^”和“-”外的单个字符不放在引号中且不借助任何分隔符进行连接。因此，[xyz]表示可以匹配一个x、y或z的类。字符范围以“-”分隔。[x-z]与[xyz]相同。“\”是转义字符，用来表示不可打印字符和特殊符号。遵照C语言的习惯，“\n”是换行符（即行结束符），“\t”是制表符，“\\”是反斜线符号本身，而“\10”是对应八进制10的字符。“^”符号取一个字符类的补（类似Not）。[^xy]是匹配除x和y之外所有字符的字符类。

65
66

Lex提供了标准正则表达式操作符以及一些扩展。连接操作由两个表达式的并置指定，不使用显式操作符。因此，[ab][cd]将匹配ad、ac、bc和bd中的任意一个。在字符类的方括号外出现的单个字母和数字匹配它们自身。其他字符需放在引号中（以避免误译为正则表达式操作符）。例如，begin可由表达式begin、"begin"或[b][e][g][i][n]匹配。

67

大小写是有区别的。选择操作符是“|”。括号通常用来控制子表达式分组。因此，在上面的定义中，为匹配允许大小写混合的保留字end，可使用：

```
(E|e) (N|n) (D|d)
```

Lex中同时提供了后缀运算符“*”（Kleene闭包）和“+”（正闭包），以及“?”（可选包含）。Expr?匹配Expr零次或一次。它和(expr)|λ等价，同时避免了对显式λ符号的需要。

符号“{”和“}”触发第一节中定义的符号的宏展开。例如，因为Digit被定义为[0-9]，所以{Digit}+展开为[0-9]+。

第二节定义正则表达式和相应命令的列表。当一个表达式被匹配时，将执行它所对应的命令。如果输入序列不匹配任何表达式，则该序列将被简单地逐字复制到标准输出文件。匹配的输入存储在字符串变量yytext中（其长度为yyleng）。命令能以任何方式改变yytext并随后将改动过的文本写到输出文件中。

Lex创建可从语法分析器（如在语法制导的编译中的标准语法分析器）中调用的整型函数yylex()，其返回值通常是由Lex扫描的词法记号的代码。像空白这样的词法记号可以通过在它们相应的命令中不返回任何值而被简单地删除。词法分析将继续进行，直到执行到有返回值的命令。

Lex不像ScanGen那样特别提供异常列表机制。在识别标识符时，可以调用像3.4.1节中的check_exceptions()例程那样的子程序来识别异常并返回正确的词法记号代码。

除此之外，Lex还允许正则表达式重叠（即匹配公共的输入序列）。在重叠的情况下，应用两条规则。首先，执行最长可能匹配。Lex在需要时自动进行缓存。其次，如果两个表达式匹配相同的字符串，则选择较早声明的表达式（以在Lex规范中的定义为序）。因此，可通过将（仅匹配特定字符串的）特殊表达式放在一般模式（通常是一个标识符）之前来处理异常。

Lex中没有特别提供主、次词法记号代码机制。通常，主词法记号代码会作为调用Lex生成的词法分析器yylex()的返回结果。如果需要，次词法记号代码可以放在一个共享变量中返回。

Lex中没有丢弃机制。因此，必须在返回前处理词法记号文本（存放在yytext中）。这容易通过调

用一个子例程（紧随表达式和命令列表，在第三节中定义）重新处理词法记号文本来完成。例程stripquotes()是这类例程的一个示例。

在Lex中，文件结束不由正则表达式来处理，而是通过yylex()返回整数值零来表示EOF词法记号。由语法分析器来将零返回值识别为表示EOF词法记号。

如果要扫描多个源文件，可通过隐藏在词法分析器内部的机制来完成。yylex()使用三个用户自定义函数来处理字符输入和输出。它们是：

input()	获得一个字符，遇到文件结束时返回0。
output(c)	将一个字符写到输出。
unput(c)	将一个字符放回输入以便重新读入。

当yylex()遇到文件结束时，它调用一个用户自定义的名为yywrap()的整型函数。该例程的目的是“包装”输入处理。如果没有更多的输入，它返回值1。否则，它返回零并准备由input()提供更多的字符。

编译器编写者可以提供input()、output()、unput()和yywrap()函数（通常作为C语言的宏）。Lex提供默认版本，来从标准输入读取字符并将它们写到标准输出。yywrap()的默认版本简单地返回1，表示没有更多的输入。（output()的使用允许将Lex用作一个工具，来产生独立的用来转换数据流的数据“过滤器”。）

总之，Lex是一个非常实用的生成器，它可以转换输入（例如，作为预处理器），同时也可以划分或扫描输入。除了这里所讨论的内容，它还提供许多高级特性。它需要代码片段用C语言编写，这使得它不像仅产生整数表格的ScanGen那么通用。Lex是供编译器编写者使用的词法分析器生成器工具类的代表。

Lex广泛使用于UNIX社区内，而在其外界则较少使用。对于生成编译器来说，它并不足够高效，但可以为其编写更高效的实现版本。事实上，Jacobsen（1987）近来的工作表明可以改进Lex以使得它总是比手写的词法分析器运行得更快。Lex也已经被重新实现。Flex（Fast Lex[Paxson 1990]）是一个可免费发行的Lex克隆。它产生的词法分析器比Lex产生的词法分析器快得多。Flex也提供允许调整词法分析器大小和速度的选项，以及一些Lex所没有的特性（比如支持8位的字符）。Flex容易获得，可免费发布，与Lex兼容，同时比Lex运行得更好。因此，我们推荐使用Flex。

另一种有趣的选择是GLA（Generate Lexical Analyzer，生成词法分析器[Gray 1998]）。GLA使用基于正则表达式的词法分析器描述和公共词法惯用语库（比如“Pascal的注释”），产生用C语言编写的可直接执行的（即，非DFA驱动的）词法分析器。GLA在设计时就考虑到要使生成的词法分析器既易于使用又很高效。

3.5 实现时考虑的问题

本节讨论在为真正的程序设计语言构造真正的词法分析器时需要考虑的实际问题。有人可能会想，我们描述的有限自动机模型有时是不足的，必须予以补充。而且，一些错误处理机制必须结合到现实世界的词法分析器中。

我们将按顺序讨论许多潜在的问题领域。在每种情况下，将评价各种解决方案，尤其是结合我们已经研究过的词法分析器生成器。

3.5.1 保留字

实际上，所有的程序设计语言都有匹配普通标识符词法结构的符号（如**if**和**begin**）。这些符号被称为关键字（key word）。如果语言规定关键字不能用作程序员定义的标识符，则它们又称为保留字（reserved word，即它们被保留用作特殊用途）。

大多数程序设计语言选择将关键字保留。这样做简化了语法分析，而语法分析驱动着编译过程并使程序更加易读。例如，假定在Pascal语言中begin和end不是保留的，而一些步入歧途的程序员已经声明名为begin和end的过程。下列程序能够以许多方式进行语法分析，因此它的含义并没有被良好定义：

```
begin
  begin;
  end;
  end;
  begin;
end
```

通过精心设计，可以避免明显的二义性。例如，在PL/I语言中关键字不被保留，但使用显式的call关键字来调用过程（如call P）。尽管如此，由于关键字可用作变量名，仍会有很多费解的用法：

```
if if then else = then;
```

保留字的问题是：如果它们太多，就会使那些没有经验的程序员感到困惑，他们可能无意中选择了和保留字相冲突的变量名。这通常会在一些程序中导致语法错误。那些程序“看起来正确”，但事实上只有假设那些可疑符号不是保留字时程序才会正确。COBOL语言在这个问题上声名狼藉。它有数百个
70 保留字。例如，在COBOL中，zero是个保留字，zeros也是。而zeroes还是！

保留字看起来像标识符，这使得通过正则表达式定义的词法分析器规范变得极其复杂。在练习20中证明使用**Not**的任意正则表达式等价于一个不使用**Not**的正则表达式。因此，可以通过除去表达式中的**Not**得到一个非保留Id的正则表达式：

Not (**Not**(Id) | begin | end | $\cdots$)

我们也可以直接写下正则表达式。然而，它将会非常长而且很复杂。仅仅设想一下，假设END是惟一的保留字，而词汇表中仅含字母，可以写：

$$\text{Nonreserved} = L \mid (L\,L) \mid ((L\,L\,L)\,L^*) \mid ((L - 'E')\,L^*) \mid (L\,(L - 'N')\,L^*) \mid (L\,L\,(L - 'D')\,L^*)$$

（即，任意多于或少于三个字母，或不以E开头，或N不在其第二个位置，等等。）

一个更简单的解决方案是将保留字看作普通标识符（就正则表达式而言）并使用一张单独的表格进行查找以发现保留字。也就是说，将保留字视为普通标识符的异常情况。在扫描到一个像标识字的记号后，查阅异常表来判断是否识别了一个保留字。如果保留字区别大小写，则异常查找需要精确匹配；否则，在进行查找前，词法记号被转换为标准形式（全大写或全小写）。

可以用多种方式组织异常表。最普通的组织方式是一个适于进行二分查找的有序异常表。也可以使用哈希表。例如，词法记号的长度可以用来作为异常表中相同长度异常的索引。如果异常长度良好地分布，则需要很少的几次比较就可以确定一个词法记号是标识符还是保留字。Cichelli（1980）已经证明，有可能构造完美的哈希函数。也就是说，每个保留字被映射到异常表中的惟一位置，且表中没有未使用的位置。一个词法记号要么是由哈希函数选择的保留字，要么是普通标识符。

识别保留字的第三种方法是为每个保留字构造单独的正则表达式。如果所使用的词法分析器生成器允许多个正则表达式匹配同一个字符序列（大多数词法分析器生成器确实允许这样），则这种方法是可行的。这种方法的真正问题是底层有限自动机以及它的转换表将非常大。这有两个原因。首先，由于并非所有字母都映射到单独的类，因此字符类的数量将大大增加。根据保留字的数量和拼写，仅对字母就将有26个（甚至52个）类。第二个问题是自动机状态数将大大地增加，新的状态代表部分匹配的保留字。

71 即使在像Micro这样简单的语言中，为（全部的四个！）保留字构造正则表达式也会导致表的尺寸大大增加。使用3.4.1节中图3-3的定义（其中将保留字视为异常），将创建12个状态和17个字符类。当4个保留字由单独的正则表达式表示时，则需要30个状态和26个字符类。假定用简单数组表示转换表，则产生四倍的增量。而在像Ada和Pascal这样的语言中，保留字数量巨大，因此将会产生多得多的状态和

字符类。

3.5.2 编译器指示与源程序行列表

编译器指示和pragma用来控制编译器选项（列表、优化、性能剖析，等等）。它们可能由词法分析器或语义例程处理。如果指示是简单的标志，则可由词法记号中提取（可能使用丢弃/保存机制）。然后执行相应命令，最终该记号被删除。更多复杂的指示，像Ada语言中的pragma，拥有非平凡结构，并可以像其他任何语句一样进行语法分析和转换。

词法分析器也可能不得不处理*源代码包含指示*（source inclusion directive），这将导致它读取并扫描一个文件的内容。像C这样的语言拥有复杂的宏定义和展开机制，通常在词法分析和语法分析之前的预处理阶段进行处理。

一些语言（像C和PL/I）包含*条件编译指示*（conditional compilation directive），来控制语句是被编译还是被忽略。这样的指示可用来从公共的源代码创建程序的不同版本。通常这些指示拥有**if**语句的一般形式，因此要分析并计算一个条件表达式。紧随表达式的词法记号将被传递给语法分析器或被忽略，直到到达一个**end if**分隔符为止。如果条件编译结构可以嵌套，将需要一个针对指示的*框架式语法分析器*（skeletal parser）。

词法分析器的另一个可能的功能是列出源代码行。尽管这看起来是个无关紧要的功能，但有必要稍加关注。产生源代码列表的最明显方式是在读入字符时进行回显，使用行结束条件来终止一行，增加行计数器，等等。然而这种方法有很多问题：

- 可能需要打印错误信息，而这些（如果可能的话）应当写在源代码行后面，其中含有指向出错符号的指针。
- 源代码行可能需要在输出前进行编辑。这可能涉及插入或删除符号（例如对于错误修复）、替换符号（因为宏预处理）以及重新格式化符号（对程序进行优美打印）。
- 读入的源代码行和输出的源代码行列表通常不是一一对应的。例如，在UNIX中，源程序可以合法地压缩到单独一行中（UNIX对于行的长度没有预先的限制）。试图缓存整个源代码行的词法分析器肯定会超出缓冲区长度。

72

基于这些考虑，在扫描词法记号时，最好*递增地构造*（通常由设备限制所限定的）输出行。放在输出缓冲区中的词法记号映像可能并非恰好是被扫描的词法记号映像，这取决于错误修复、优美打印、大小写转换或任何其他需要的处理。如果一个词法记号不适合放在某一输出行，该行将被写到输出中，而缓冲区被清除。（为简化编辑，行号应该对应源程序行。）在极少的情况下需要拆散一个词法记号；例如，如果一个字符串太长以至于超过了输出行长度限制。

词法分析器返回每个词法记号时，也应当包含该词法记号在输出行缓冲区中的位置。如果发现涉及该词法记号的错误，则使用位置标记来指向该词法记号。错误信息自身被缓存并通常在相应输出缓冲区内容输出时立即进行打印。在某些情况下，一个错误可能要到包含该错误的行已被处理之后很久才被检测到。这种情况的一个例子是**goto**语句跳转到未定义的标号。如果这样的延迟错误很少见（像在Ada和Pascal中那样），则可以产生一条引用行号的信息——例如，“未定义的标号，在语句101中。”在允许自由地前置引用的语言中，可能有很多延迟错误。例如，PL/I允许对象引用后再声明。在这种情况下，可以写一个以行号为关键字的错误信息文件，并随后将它与处理过的源代码行合并产生完整的源代码列表。

UNIX的方法是编译器应当专注于生成代码，而将列表和优美打印等功能交给其他工具。当然，这样做极大地简化了词法分析器。

3.5.3 符号表中的标识符条目

在仅有全局变量和声明的简单语言中，若某标识符还不在符号表中，通常由词法分析器立即将其加

入符号表。不论标记符应当加入符号表还是已经在表中，词法分析器都返回指向符号表条目的指针。

在块结构的语言中，因为标识符可以用于许多上下文中（作为变量，或在声明中作为记录域、标号，等等），所以通常不是由词法分析器将标识符添加到符号表中或在表中查找标识符。词法分析器一般不可能知道何时应当将一个标识符添加到当前作用域的符号表中，以及何时应当返回指向先前作用域中实例的指针。因此，最好返回字符串自身，并允许单独的语义例程解析标识符的预期用法。通常，用*字符串空间*（string space）来表示标识符（见第8章），因此词法分析器可以将标识符加入字符串空间并返回字符串空间指针而不是实际的文本。

73 在某些语言（例如C语言）中，大小写是有区别的。但在其他语言（像Ada和Pascal）中，大小写是没有区别的。如果大小写有区别，标识符文本必须在扫描时被精确地返回。保留字查找必须区分仅大小写不同的标识符和保留字。另一方面，如果大小写没有区别，则需要保证标识符或保留字拼写中的大小写差别不会导致错误。一个容易的办法是对于所有被扫描的词法记号，在将其返回或在保留字表中进行查找之前，作为标识符被转换成一致的大小写形式。

3.5.4 词法分析器的终止

词法分析器被设计用来读取输入字符并将它们划分为词法记号。当到达输入结尾时会发生什么情况？在发生这种情况时，创建一个Eof伪字符是方便的。这样做将允许定义一个可被回传给语法分析器的Eof*词法记号*。Eof词法记号在CFG中很有用，因为它允许语法分析器验证相应于一个程序物理结尾的逻辑结尾。

如果在到达文件结尾后调用词法分析器会发生什么？显然，一个致命错误会被记录下来。但这将破坏词法分析器总是返回一个词法记号的简单模型。更好的方法是继续向语法分析器返回Eof词法记号。这样做允许语法分析器简洁地处理终止问题，尤其当Eof词法记号仅在完整的程序之后在词法上有效时。如果Eof词法记号出现得太早或太晚，那么语法分析器可以执行错误修复或产生适当的错误信息。

3.5.5 多字符的超前搜索

我们可以推广FA以越过下一个输入字符进行超前搜索。该特性对于实现FORTRAN语言的词法分析很重要。例如，语句DO 10 I = 1,100是一个循环的开始，其索引I从1变化到100。语句DO 10 I = 1.100是对变量DO10I的赋值（空白在FORTRAN中没有意义）。

FORTRAN词法分析器一直要读到逗号（或句号）后才确定O是否为DO记号的最后一个字符。（事实上，在FORTRAN的DO循环中将“.”错误地替换为“,”曾导致一艘宇宙飞船发射失败！因为替换的结果仍是一条有效语句，该错误直到运行时才被发现，不过这已经是在火箭发射之后。火箭偏离了轨道并不得不被摧毁。）

该问题的一个较为轻微的形式发生在Pascal和Ada中：为扫描10..100，需要在10之后超前搜索两个字符。假定使用图3-6中的FA。给定10..100，我们会扫描三个字符并停在非终结状态。如果在非终结状态停止读入，可以回退已接受的字符直到找到一个终结状态。回退的字符将被重新扫描以形成后来的词法记号。如果在回退中未到达终结状态，则产生词法错误并进行词法错误恢复。

在Pascal或Ada中，我们知道从不需要多于两个字符的超前搜索，它简化了对要重新扫描字符的缓
74 存。除此之外，还可以在上面的自动机中添加一个新的终结状态，对应形如(D+.)的伪词法记号。如果识别出了该词法记号，则从词法记号文本中删除结尾的“.”并将它缓存供以后重用。随后返回整数文字常量的词法记号代码。

在Ada中，撇号字符“'”既用作属性符号（arrayname'length）又用来界定字符文字常量（'x'）。Ada词法分析器必须区分这两种用法。考虑TypeName' ('a', 'b')，第一个撇号是属性符号，用来限定该

聚合表达式的类型。剩下的撇号界定聚合中出现的字符文字常量。多数Ada词法分析器使用的解决方案是将撇号字符视为特殊情况。当看到撇号时，检查（由语法分析器设置的）一个标志以确定下一步可以读到属性符号还是字符文字常量。根据这个标志，扫描一个单独的撇号或一个引号中的字符。

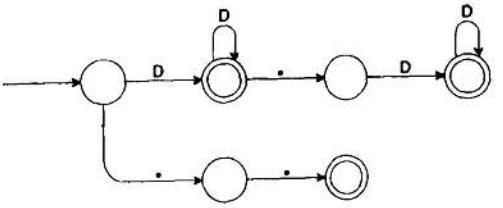


图3-6 一个扫描整数和实数文字常量和子界算符的FA

在扫描无效（invalid）程序时也可以考虑多字符超前搜索。例如，12.3e+q是一个无效词法记号。在这种情况下，词法分析器可以回退以产生四个词法记号。由于该词法记号序列(12.3, e, +, q)是无效的，因此语法分析器在处理该序列时将检测到语法错误。认为该错误是词法错误还是语法错误（或两者都是）都无关紧要，总之编译器必须在某个阶段检测到此类错误。

很容易构造一个能执行一般回退操作的词法分析器。在扫描每个字符时，都将它缓存，并设置一个标志指示到目前为止扫描的字符序列是否是有效的词法记号（该标志可以是适当的词法记号代码）。如果此时我们不在终结状态且不能扫描更多的字符，将执行回退操作。我们从缓冲区右端提取字符并将它们加入队列以重新扫描。该过程将持续到我们到达被标志为有效词法记号的已扫描字符的前缀为止。该记号将由词法分析器返回。如果没有前缀被标志为有效记号，则产生一个词法错误。（词法错误在3.5.6节讨论。）

75

作为含有回退的词法分析示例，考虑先前12.3e+q的例子。下面的表格说明怎样构造缓冲区及设置标志：

Buffered Token	Token Flag
1	Integer Literal
12	Integer Literal
12.	Invalid
12.3	Real Literal
12.3e	Invalid
12.3e+	Invalid

当扫描q时，执行回退。最长有效词法记号前缀是12.3，因此将返回一个实数文字常量，而e+被重新加入队列以便随后重新扫描。

3.5.6 词法错误恢复

词法分析器有时会检测到词法错误。为了这样一个很小的错误而停止编译是不切实际的，因此必须尝试某种类型的词法错误恢复（lexical error recovery）。可以考虑两种方法：

- 删除到目前为止读入的字符并在下一个未读字符处重新开始扫描。
- 删除词法分析器读取的首字符并从其后的字符恢复扫描。

两种方法都是合理的。前者很容易做到。我们仅需重置词法分析器并重新开始扫描。后者则稍有点困难，但更加安全（由于仅删除较少的字符）。它可以通过前面讨论词法分析器回退的章节中所描述的缓存机制来实现。

在大多数情况下，词法错误是由通常出现在词法记号开头的某些非法字符引起的。在这种情况下，

上述两种方法工作得同样好。词法错误恢复的结果可能会产生语法错误，而语法错误可由语法分析器检测并恢复。例如，… beg#in …或许被修复为… beg in …，而这几乎肯定会导致语法错误。这样的情况几乎不可避免，而一个好的语法错误修复算法将会进行一些合理的修复（尽管很可能不是正确的修复！）。

如果语法分析器拥有语法错误修复机制（见第17章），则当词法错误发生时返回特殊的警告词法记号是有用的。警告词法记号的语义值是为重新开始词法分析而删除的字符串。当语法分析器看到警告词法记号时，它被警告下一个词法记号不可靠并可能需要错误修复。此外，被删除的文本也可能有助于选择最合适的修复。

76 某些词法错误需要特别留心。尤其是，失控字符串和注释应当接收特殊错误信息。首先考虑失控字符串。由于通常不允许字符串跨越行边界，因此当遇到行结束时才会检测到失控字符串。普通的恢复方法可能不适于这种错误。特别是，由于不恰当地将字符串文本作为普通输入来分析，删除首字符（引号）并重新开始词法分析几乎一定会导致更多的级联错误。

捕获失控字符串的一种方法是引入*错误词法记号*（error token），表示字符串由行结束符而不是引号终止。（这在ScanGen和Lex示例中做过说明。）因此，对于正确引用的字符串，可以有：

```
" (Not(" | Eol) | "")* "
```

而对于失控字符串，应当使用：

```
" (Not(" | Eol) | "")* Eol
```

其中，Eol是行结束字符。当识别出失控字符串词法记号时，应当产生特殊的错误信息。此外，字符串可以通过返回其中除去了开头的引号和结尾的Eol的普通字符串记号（就像除去开头和结尾引号的普通字符串）而被修复为正确的字符串。注意，尽管如此，这种修复可能是也可能不是“正确的”。如果真是丢掉了结尾的引号，则这个修复是好的；如果结尾的引号出现在后续行中，则会接着产生不适当的级联词法和语法错误。

在像Pascal这样的语言中允许多行注释。失控注释会造成类似的问题。失控注释直到词法分析器发现（可能属于另一个注释的）注释结束符号或直到遇到文件结束时才会被检测到。显然，需要特殊的错误信息。为处理以Eof终止的注释，再次使用错误词法记号方法：

```
{ Not(})* }  和  { Not(})* Eof
```

为了处理由属于另一个封闭注释的结束符结尾的注释（例如，{… missing close comment … { normal comment }}），我们会发出警告（但不是错误信息，因为这种形式的注释在词法上是合法的）。特别是，内含一个注释开始符号的注释最有可能是上述类型疏忽的征兆。因此，我们将合法的注释定义拆分成两个词法记号。其中一个在其内部接受一个注释开始符号，将导致打印一个警告信息（“Possible unclosed comment”（可能未封闭的注释））。现在我们有

```
{ Not({ | })* } 和 { (Not({ | })* { Not({ | })*)* } 以及 { Not(})* Eof
```

77 第一个定义匹配其中不包含注释开始符号的正确注释。第二个定义匹配其中至少包含一个注释开始符号的注释。这些注释是正确的，但将导致产生一条警告信息。第三个定义匹配以Eof结束的失控注释。这些词法记号导致产生一条错误信息。

当然，Ada的注释总是以Eol结尾，因此不会受到失控注释问题的困扰。不过，它们要求多行注释中的每一行都包含一个注释开始符号。注意，正如在前面所见到的，通常*嵌套注释*（nested comment）无法被正确识别，因为FA和正则表达式无法正确识别平衡的注释开始/注释结束记号序列。当我们想要注释嵌套，尤其是当“注释掉”一段代码（这段代码自身也可能含有注释）时，无法识别这样的序列将会导致问题。该问题的通常解决方案是拥有两种或是多种类型的注释分隔符。例如，UW-Pascal识别三类注释：1）由“{”和“}”匹配的注释，2)由“(*”和“*)”匹配的注释，以及3)由“""”和“""”匹配的注释。如果某

类注释用于一般文档，另一类则可以专门用来注释掉代码段。但是，新的Pascal标准规定注释可以由某类注释分隔符开始而以另一类注释分隔符结束（例如，(* comment *)）。该规定使得注释嵌套不可能存在。

正如先前所注意到的，对于词法分析器来说，假定一个包含像Eol和Eof这样的伪字符的扩展字符集通常是方便的。这些伪字符可以用来定义正则表达式。它们也可以用来控制格式（使用缩进、反缩进和换行伪字符）。伪字符可以由词法分析器的读例程创建（例如，如果行结束条件为真，则可以返回Eol），或者作为处理编译器指示的结果。这样的编译器指示可以通知对输入程序进行优美打印。

幸运的是，C语言和标准I/O库通过为不可打印字符提供转义序列（例如'\n'代表换行，而EOF代表文件结束）替我们完成了部分工作。

3.6 将正则表达式转换为有限自动机

正则表达式等价于FA。事实上，词法分析器生成器程序的主要工作是将正则表达式定义转换为等价的FA。它首先将正则表达式转换为非确定有限自动机（Nondeterministic Finite Automaton, NFA）。在读取特定输入时，NFA不需要为访问哪一个状态做出惟一（确定的）选择。例如，如图3-7所示，NFA被允许含有一个状态，该状态拥有两个以同样符号标记的出转换（箭头）。如图3-8所示，NFA中也可以含有标记为λ的转换。

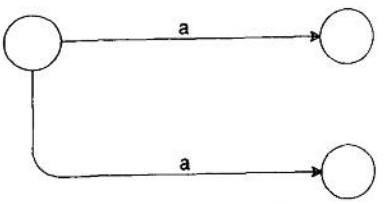


图3-7 含有两个a转换的NFA

转换通常以V中的单个字符（字母）标记，尽管λ是一个字符串（该字符串中不含任何字符），但它绝对不是一个字符。在后一个例子中，当自动机处于左边状态，而下一个输入字符是a时，它可以选择使用标以a的转换或首先跟随λ转换（无论何时你总能找到λ转换）然后跟随一个a转换。不包含λ转换且对于任意符号总是拥有惟一后继状态的FA是确定的（deterministic）。 78

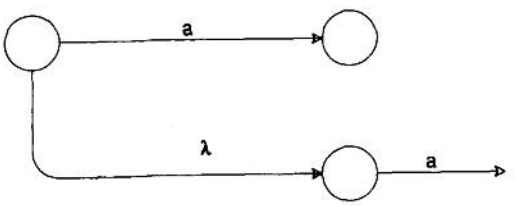


图3-8 含有一个λ转换的NFA

由正则表达式构造FA的算法分为两步：首先，它将正则表达式转换为NFA，然后再将NFA转换为确定的FA。第一步非常简单。事实上，可以将任意正则表达式转换为具有下列性质的NFA：

- 有惟一的终结状态。
- 终结状态没有后继。
- 每个其他状态拥有一个或两个后继。

正则表达式都是由原子正则表达式a（其中a是V中的一个字符）和λ通过三种操作A B和A | B以及

A*构造出来。其他操作（像A+）只是这些操作组合的缩写。如图3-9所示，a和λ的NFA很简单。

79

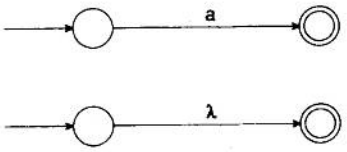


图3-9 a和λ的NFA

现在假定已经有了A和B的NFA，想要A | B的NFA。构造图3-10所示的NFA。标记为A和B的状态分别是A和B的自动机的终结状态。我们为该组合自动机创建一个新的终结状态。

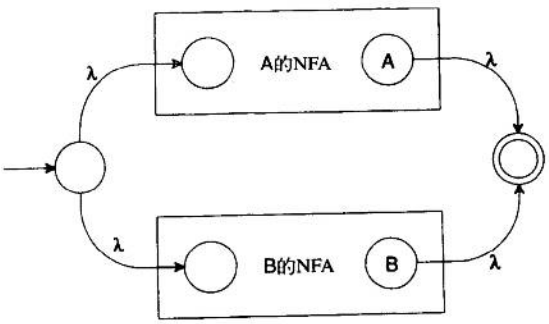


图3-10 A | B的NFA

如图3-11所示，为A B构造自动机更容易。该组合自动机的终结状态与B的终结状态为同一个状态。也可以将A的终结状态和B的初始状态合并。我们没有选择这样做，仅仅因为这样的话，图更不容易画出来。最后，A*的NFA如图3-12所示。

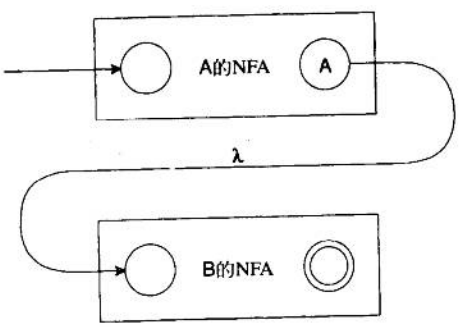


图3-11 A B的NFA

3.6.1 构造确定的有限自动机

NFA N到等价的DFA M的转换通过有时被称为子集构造（subset construction）的过程进行。M的每

80

个状态对应N中状态的一个集合。其想法是，在读入给定输入字符串后M将处于状态{x, y, z}，当且仅当N能够处于状态x、y或z中的任意一个（依赖于它所选择的转换）。因此，M记录N可能选择的所有路径，并且平行地管理它们。M的接受状态将会是包含N的接受状态的任意集合，它反映了这样一个约定：如果通过选择“正确的”转换，存在任何一种可以使之到达终结状态的方式，则N处于接受状态。

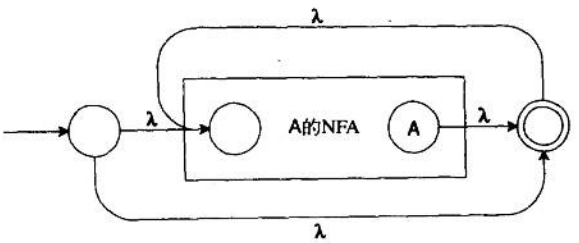


图3-12 A*的NFA

M的初始状态是N在不读入任何输入字符就能处于的状态——即，由N的初始状态仅跟随λ箭头就可以到达的状态集。算法close()计算那些仅通过λ转换就能到达的状态。一旦构造了M的开始状态，我们就可以开始创建后继状态。为此，取M的任意状态S，以及任意字符c，计算在面临输入c时S的后继。S由N的某个状态集{n_1, n_2, …}确定。我们找到在面临输入c时{n_1, n_2, … }所有可能的后继状态，获得集合{m_1, m_2, … }。最后，计算T = close({m_1, m_2, … })。将T作为一个状态加入M中，并将一个标以c的从S到T的转换添加到M中。继续向M中添加状态和转换，直到已有状态的所有可能的后继都已被添加。因为每个状态对应于N中状态的一个（有限）子集，所以向M中添加新状态的过程必定会终止。 81

下面是λ闭包和DFA构造的完整算法。（C语言中没有集合操作，我们通过类似宏的记号对它们进行简要描述。）

```
/*
 * Add to S all states reachable from it
 * using only λ transitions of N
 */
void close(set_of_fa_states *S)
{
    while (there is a state x in S
        and a state y not in S such that
        x→y using a λ transition)
            add y to S
}
```

使用该过程，可以定义M的构造：

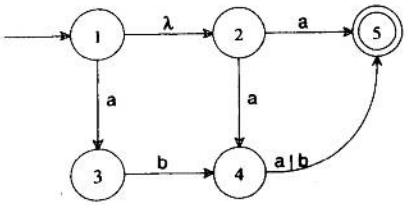
```
void make_deterministic(nondeterministic_fa N,
                        deterministic_fa *M)
{
    set_of_fa_states T;

    M->initial_state = SET_OF(N.initial_state) ;
    close(& M->initial_state);
    Add M->initial_state to M->states;
    while (states or transitions can be added)
    {
        choose S in M->states and c in Alphabet;

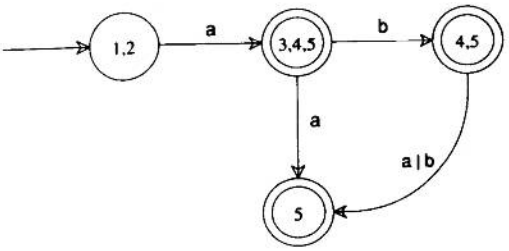
        T = SET_OF(y in N.states
            SUCH_THAT x -c-> y for some x in S) ;
        close(& T);
        if (T not in M->states)
            add T to M->states;

        Add the transition to M->transitions: S -c-> T ;
    }
    M->final_states =
       SET_OF(S in M->states SUCH_THAT
              N.final_state in S);
}
```

为了解子集构造如何进行，考虑下面的NFA: 82



从状态1（N的开始状态）开始，添加状态2（它的λ-后继）。因此，M的开始状态是{1, 2}。面临输入b时状态1和状态2都没有后继。面临输入a时，{1, 2}的后继是{3, 4, 5}。面临输入a和b时{3, 4, 5}的后继分别是{5}和{4, 5}。面临输入b时{4, 5}的后继是{5}。M的终结状态是那些包含N的终结状态（5）的状态集。最终的DFA为:



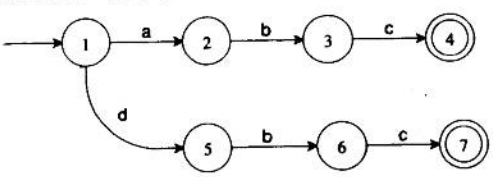
不难证明：由make_deterministic()所构造的DFA与原来的NFA等价（见练习22）。一个不太明显的事实是我们所构造的DFA有时会比原来的NFA大得多。DFA的状态由NFA的状态集确定。如果NFA有n个状态，则有2^n个不同的NFA状态集，因此DFA可能有2^n个状态。练习18讨论了一个NFA，它实际展示了在被确定化时其大小的指数爆炸。幸运的是，由那些指定程序设计语言词法记号的正则表达式所构造的NFA，在其确定化时通常不会发生这种爆炸问题。通常，用于词法分析的DFA既简单又紧凑。

83

3.6.2 优化有限自动机

我们不必停步于由make_deterministic()所创建的DFA上。有时，该DFA拥有一些多余的状态。此外，已知对于每个DFA都有（就状态数而言）惟一最小的等价DFA。换句话说，假定一个DFA（M）有75个状态，而拥有50个状态的DFA（M'）接受同一个字符串集。进一步假定没有少于50个状态的DFA等价于M。则M'是惟一拥有50个状态且等价于M的DFA。使用下面讨论的技术，有可能通过使用M'替换M对其进行优化。事实上，这正是ScanGen的optimize选项所做的。

我们从尝试最乐观的状态合并开始。按照定义，终结和非终结状态是截然不同的，因此我们开始仅试着创建两个状态：一个代表所有终结状态的合并，另一个则代表所有非终结状态的合并。将所有状态仅合并为两个状态可能过于乐观。特别是，对于一个给定字符，如果一个合并状态中所有成员未能对转换达成一致，则该状态必须被分裂。作为示例，假定由下面的自动机开始:



初始时有非终结状态{1, 2, 3, 5, 6}和终结状态{4, 7}。当且仅当所有成员状态就相同后继状态达成一致时，一个合并才是合法的。例如，给定字符c，状态3和6都将到达终结状态；而状态1、2和5则不会，因此必须发生分裂。我们在原始的DFA中添加一个错误状态s_E作为面临任意非法字符时的后继状态。

（因此，到达s_E等价于检测到非法词法记号。）s_E不是真正的状态，它只是允许我们假定每个状态在面临每个字符时都有一个后继。s_E从不和任意真正的状态合并。

图3-13中给出的例程将分裂那些面临任意给定字符、其成员没有公共后继的合并状态。当split()终止时，我们知道仍旧合并在一起的状态是等价的，因为它们总是有公共后继。

```
void split(set_of_fa_states *ss)
{
    do {
        Let S be any merged state corresponding to
            {s1 ,..., sn} and
            let c be any character;
        Let t1 ,..., tn be the successor states to
            {s1 ,..., sn} under c;
        if (t1 ,..., tn do not all belong to the
            same merged state)
        {
            Split S into new states so that si and
            sj remain in the same merged state if
            and only if ti and tj are in
            the same merged state;
        }
    } while (more splits are possible);
}
```

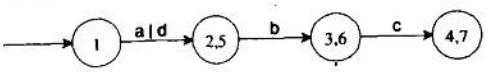
图3-13 分裂FA状态的算法

回到我们的例子中，初始时有状态{1, 2, 3, 5, 6}和{4, 7}。调用split()，首先观察到状态3和6在面临c时有公共后继，而状态1、2和5在面临c时没有后继（或者，等价于有错误状态s_E）。这将导致一次分裂，产生{1, 2, 5}、{3, 6}、{4, 7}。现在，对于字符b，状态2和5会到达合并状态{3, 6}，但状态1不会，因此发生另一次分裂。现在有：{1}、{2, 5}、{3, 6}、{4, 7}。在此已经完成了分裂，因为合并状态的所有成员对于每个输入符号都有相同的后继。

84

一旦执行了split()，就基本上完成了优化。合并状态间的转换与原始DFA中状态间的转换相同。也就是说，如果面临字符c，状态s_i和s_j之间存在转换，则现在面临c，存在从包含s_i的合并状态到包含s_j的合并状态的转换。开始状态是包含原来的开始状态的合并状态；终结状态是包含原来的终结状态的那些合并状态（回想终结和非终结状态从不合并）。

回到刚才的例子，我们所获得的最小状态自动机为：



对于该最小化算法的正确性和最优性的证明可以在Hopcroft and Ullman（1979）中找到。

FA可被看作最简单的CPU，它被设计用于匹配字符而不是执行更一般的计算。正则表达式则是FA的程序设计语言，它们被编译和优化为可执行的形式。在大多数情况下，我们可以通过使用（像先前介绍的）简单的驱动程序来模拟FA。然而，可以设想直接实现FA的专用处理器（可能以VLSI芯片实现）。在此方向上至少报道过一个实验。在以词法分析为一个限制因素的任意应用程序中，专用“词法分析引擎”被证明是有优势的。

85

练习

1. 假定为Pascal词法分析器提交下列文本：

```
program m(output);
const
    pay=284.00;
```

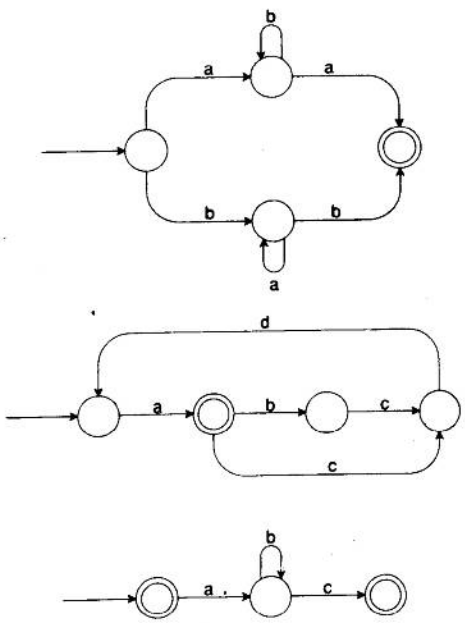
```
var
    bal:real;
    month:0..60;
begin
    month:=0;
    bal:=11163.05;
    while bal>0 do begin
        writeln('Month: ', month:2,
                ' Balance: ', bal:10:2);
        bal:=bal-(pay-0.015*bal);
        month:=month+1;
    end;
end.
```

生成什么词法记号序列？对于哪些词法记号，词法记号文本必须与词法记号代码一起被返回？

2. 在下面的Pascal程序段中会出现多少词法错误（如果有的话）？词法分析器如何处理每个错误？

```
If a = 1. Then b :=1.0else c := 1.0E+N;
Writeln('''',"Hi there!",'''');
```

3. 写出正则表达式来定义下列FA所识别的字符串：



86

4. 写出三个DFA来识别由下列正则表达式所定义的词法记号：

(a | (bc)* d)+

((0 | 1)* (2 | 3)+) | 0011

(a Not(a))* aaa

5. 写出正则表达式来定义Pascal式定点十进制文字常量，其中不含多余的开头或结尾的零。即，0.0、123.01和123005.0是合法的，但00.0、001.000和002345.1000是非法的。

6. 写出正则表达式来定义由“(*”和“*)”界定的Pascal式注释。单独的“*”和“)”可以出现在注释体中，但字符对“*)”不可以。

7. 取扩展Micro的ScanGen或者Lex定义并将其进一步扩展，使其包含练习5和练习6中的定点十进制文字常量和Pascal式注释的定义。运行ScanGen或Lex以验证你的定义是合法的。

8. 在3.5.1节中，我们比较了识别保留字的两种方法：将它们作为标识符词法记号的异常以及为它们创建单独的正则表达式定义。在Micro中，我们发现当使用单独的词法记号定义时，转换表的大小大约增长为原来的4倍。

   （使用ScanGen或者Lex）重复该比较，这次使用（在附录A中定义的）Ada/CS的保留字。比较词法分析器的大小和词法分析器的速度。你推荐使用哪一种方法？

9. 将词法记号类AlmostReserved定义为那些不是保留字但如果改变其中一个字符就会变成保留字的标识符。为什么知道一个标识符“几乎”是一个保留字是有用的？你怎样使词法分析器既能识别AlmostReserved词法记号又能识别普通保留字和标识符？ 87

10. 当首次设计和实现一个编译器时，明智的做法是专注于设计的正确性和简单性。在编译器已经被完全实现和测试之后，则有必要加快编译速度。你怎样确定编译器的词法分析器组件是否是主要的性能瓶颈？如果是，你应该通过什么方式（在不影响编译器正确性的情况下）改善性能？

11. 通常编译器可以产生被编译的程序的源代码列表。该列表通常仅是源文件的一个复本，可能添加了行号和分页符等修饰。假如我们希望产生优美打印的列表（即，列表中的文本正确缩进、**begin-end**对对齐，等等），你将如何修改ScanGen驱动程序以产生优美打印的列表？在Lex中你又将怎么做（其中完整的词法分析器模块由Lex处理器生成）？

    当产生优美打印的列表时，编译器诊断和行编号如何变得复杂？

12. 对于大多数现代程序设计语言来说，词法分析器只需要很少的上下文信息。也就是说，可以通过检查其文本，或许再加上一个或两个字符的超前搜索就可以识别出一个词法记号。如3.5.5节中所讨论的那样，在Ada中，需要额外的信息来区分一个单独的撇号（构成一个属性符号）和一个撇号、字符、撇号的序列（构成一个加引号的字符）。

    假定当一个加引号的字符可以被分析时，（由语法分析器）设定标志can_parse_char。如果下一个输入字符是撇号，can_parse_char可以用来控制如何对撇号进行词法分析。

    说明怎样将can_parse_char标志简洁地集成到由ScanGen或Lex创建的词法分析器中。你所建议的改动应当不会无故地使普通词法记号的词法分析复杂化或变慢。

13. 不像Pascal或Ada，FORTRAN一般忽略空白，因此可能需要大量的超前搜索以确定怎样扫描一个输入行。先前我们曾注意到一个很好的例子：DO 10 I = 1，10产生7个词法记号，而DO 10 I = 1.10产生3个词法记号。你会怎样设计词法分析器来处理FORTRAN所需的扩展的超前搜索？

    Lex包含一个完成这种超前搜索的机制。在此例中，你怎样匹配标识符？

14. 因为FORTRAN通常忽略空白，包含n个空白的字符序列可以有2^n种不同的词法分析方式。这些选择中的每一种都同样是可能的么？如果不是，你怎样改变练习13中你所建议的设计以首先检查最可能的选择？

15. 假设我们正在设计最终的程序设计语言，“Utopia 94”。我们已经指定了该语言的词法记号（使用正则表达式）和该语言的语法（使用CFG）。现在希望确定那些需要由空格来分隔的词法记号序列（像**begin** A）以及那些在词法分析中需要额外的超前搜索的词法记号序列（像1..10）。说明怎样才能使用正则表达式和上下文无关文法自动地找出需要特殊处理的所有词法记号对。 88

16. 证明集合$\{[]^i \mid i \geq 1\}$不是正则的。*提示*：证明没有固定数目的FA状态有能力精确匹配左右方括号。

17. 给出使用3.6节的技术为下面的表达式构造的NFA：

    $(ab^*c) \mid (abc^*)$

    使用make_deterministic()，将该NFA转换为DFA。使用3.6.2节的技术，将你所创建的DFA优化为一个最小状态等价。

18. 考虑下面的正则表达式：

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

画出对应该表达式的NFA。

证明：等价的DFA比你给出的NFA大指数倍。

19. 将正则表达式转换为NFA是快速而简单的。创建等价DFA则较慢，而且会导致更大的自动机。一个有趣的变通办法是使用NFA进行词法分析，由此彻底消除构造DFA的需要。其想法是在进行词法分析时模拟`close()`操作和`make_deterministic()`例程（如3.6.1节中的定义）。不维护单一一个当前状态，而是维护一组可能状态。读入字符时，跟随从当前集合中的每个状态出发的转换，创建新的状态集。如果当前集合中的任意状态是终结状态，则已读入的字符组成一个有效的词法记号。为NFA定义一种合适的编码（可能是用于DFA的转换表的推广）并遵循上面所描述的状态集方法编写能够使用该编码的词法分析器驱动程序。该词法分析方法无疑会比使用DFA的标准方法要慢。在什么情况下使用NFA进行词法分析比较有吸引力？

20. 假设R是任意正则表达式。**Not**(R)表示所有不在R所定义的正则集中的字符串的集合。证明**Not**(R)是正则集。

提示：如果R是正则表达式，则存在一个用来识别由R所定义的正则集的FA。将该FA转换为将识别 89
Not(R)的一个FA。

21. 令Rev为将字符串逆转的操作符。例如，Rev(abc) = cba。令R为任意正则表达式。Rev(R)是R所代表的字符串集，其中的每个字符串都是被翻转的。Rev(R)是正则集么？为什么？

22. 证明：在3.6.1节中由`make_deterministic()`所构造的DFA与原来的NFA是等价的。为此，你必须证明一个输入字符串能够导向NFA中的终结状态，当且仅当同样的字符串会导向相应DFA的终结 90
状态。

第4章 文法和语法分析

4.1 上下文无关文法：概念与记号

在第2章中，我们学习了上下文无关文法的基本知识。我们现在来形式化我们的定义并引入一些有用的记号。上下文无关文法（CFG）由下列四个组件定义：

(1) 一个有限的*终结符词汇表*（terminal vocabulary）V_t；这是由词法分析器产生的词法记号集。

(2) 不同中间符号的一个有限集，称为*非终结符词汇表*（nonterminal vocabulary）V_n。

(3) 一个开始所有推导的*开始符号*（start symbol）$S \in V_n$。开始符号有时也称为*目标符号*（goal symbol）。

(4) P，产生式（有时称为重写规则）的一个有限集。产生式的形式为$A \rightarrow X_1 \cdots X_m$，其中

$$A \in V_n,\ X_i \in V_n \cup V_t,\ 1 \leqslant i \leqslant m,\ m \geqslant 0$$

注意：$A \rightarrow \lambda$ 是一个有效产生式。

这些组件通常被归为一个“四元组”(V_t, V_n, S, P)，这就是CFG的形式化定义。CFG的*词汇表*V是终结符和非终结符的集合（即$V_t \cup V_n$）。

由S开始，使用产生式重写非终结符直到只剩下终结符（在此推导已经完成）。可从S推导出的字符串集合组成文法G的*上下文无关语言*（context-free language），由L(G)表示。

在描述CFG和它们的语法分析器时，区分是需要一个单独的符号还是需要一个符号串有时显得很重要。类似地，有时仅有终结符或者非终结符是合适的，而在其他时候任意词汇表符号都可以出现。为精确地阐明期望什么类别的符号和符号串，使用下列记号约定：

a, b, c, …	表示V_t中的符号
A, B, C, …	表示V_n中的符号
U, V, W, …	表示V中的符号
$\alpha, \beta, \gamma, \cdots$	表示V^*中的符号串
u, v, w, …	表示V_t^*中的符号串

使用这种记号，产生式可以写成$A \rightarrow \alpha$或$A \rightarrow X_1 \cdots X_m$。这种格式强调在产生式的左边必须是一个单独的非终结符，但右边是零个或多个词汇表符号组成的串。

通常多个产生式可以共享相同的左边。一般使用“或记号”而不是重复左边：

$$A \rightarrow \alpha \mid \beta \mid \cdots \mid \zeta$$

这是下列产生式序列的缩写：

$$A \rightarrow \alpha$$
$$A \rightarrow \beta$$
$$\cdots$$
$$A \rightarrow \zeta$$

如果$A \rightarrow \gamma$是一个产生式，则$\alpha A \beta \Rightarrow \alpha\gamma\beta$，其中$\Rightarrow$表示*一步推导*（one-step derivation）（使用产生式$A \rightarrow \gamma$）。将$\Rightarrow$扩展到$\Rightarrow^+$，表示一步或多步推导；而将$\Rightarrow$扩展到$\Rightarrow^*$，表示零步或多步推导。如果$S \Rightarrow^* \beta$，则β被称为该CFG的一个*句型*（sentential form）。SF(G)是文法G的句型集合。类似地，$L(G) = \{x \in V_t^* \mid S \Rightarrow^* x\}$。注意：$L(G) = SF(G) \cap V_t^*$；即，G的语言就是那些仅是终结符串的G的句型。

92 当推导一个词法记号序列时，如果出现多个非终结符，则需要选择下一次扩展哪一个。为描述推导序列的特征，需要在每一步指定扩展哪一个非终结符以及应用什么产生式。如果采用一个约定，指定在每一步必须扩展哪一个非终结符，则可以简化这种特征描述。一个明显的约定是在每一步都选择*最左*(leftmost)可能的非终结符。遵守此约定的推导被称为*最左推导*(leftmost derivation)。如果知道一个推导是最左推导，则仅需指定使用了什么产生式，而非终结符的选择总是固定的。为表示一个推导是最左推导，使用$\Rightarrow_{lm}$、$\Rightarrow^{*}_{lm}$和$\Rightarrow^{+}_{lm}$。通过最左推导序列产生的句型被称为*左句型*(left sentential form)。由一大类语法分析器(自顶向下的语法分析器)所发现的产生式序列是最左推导；因此，我们称这些语法分析器产生*最左语法分析*(leftmost parse)。

作为示例，考虑文法G_0，它生成简单表达式(V表示变量，F表示函数)：

```
E      → Prefix ( E )
E      → V Tail
Prefix → F
Prefix → λ
Tail   → + E
Tail   → λ
```

F(V+V)的一个最左推导是：

$$
\begin{aligned}
E &\Rightarrow_{lm} \text{Prefix (E)} \\
&\Rightarrow_{lm} \text{F(E)} \\
&\Rightarrow_{lm} \text{F(V Tail)} \\
&\Rightarrow_{lm} \text{F(V+E)} \\
&\Rightarrow_{lm} \text{F(V+V Tail)} \\
&\Rightarrow_{lm} \text{F(V+V)}
\end{aligned}
$$

相对于最左推导的另一种推导是*最右推导*(rightmost derivation)，有时也称为*规范推导*(canonical derivation)，其中总是扩展最右边的非终结符。由于我们通常从左到右的偏好，该推导序列可能看起来不太直观，但它很好地对应于一类重要的语法分析器(自底向上的语法分析器)。特别是，当一个自底向上的语法分析器发现那些用来推导一个词法记号序列的产生式时，它发现一个最右推导，不过是*逆序*(reverse order)的最右推导。也就是说，在最右推导中最后应用的产生式首先被发现，而首先使用的产生式(包括开始符号)最后才被发现。由自底向上的语法分析器所识别的产生式序列称为*最右*或*规范*语法分析；注意，它是表示最右推导的产生式序列的严格逆序。对于最右推导，使用记号$\Rightarrow_{rm}$、$\Rightarrow^{+}_{rm}$
93 和$\Rightarrow^{*}_{rm}$。*右句型*(right sentential form)是由最右推导产生的句型。在文法G_0中F(V+V)的最右推导是：

$$
\begin{aligned}
E &\Rightarrow_{rm} \text{Prefix (E)} \\
&\Rightarrow_{rm} \text{Prefix (V Tail)} \\
&\Rightarrow_{rm} \text{Prefix (V+E)} \\
&\Rightarrow_{rm} \text{Prefix (V+V Tail)} \\
&\Rightarrow_{rm} \text{Prefix (V+V)} \\
&\Rightarrow_{rm} \text{F(V+V)}
\end{aligned}
$$

推导通常由*分析树*(parse tree)表示。分析树以开始符号为根，以文法符号或λ为叶结点。分析树的内部结点是非终结符。分析树中非终结符的子结点代表应用一个产生式。即，结点A可以有子结点$X_1 \cdots X_m$，当且仅当存在产生式$A \rightarrow X_1 \cdots X_m$。当推导完成时，相应分析树的叶结点是终结符或λ。

作为示例，在文法G_0中相应于F(V+V)的分析树如图4-1所示。最左和最右推导都是分析树的线性表示。

如果我们有一个句型，则我们知道它可以从开始符号推出。因此，必定存在一个分析树。给定一个句型和其分析树，该句型的*短语*(phrase)是分析树中单独的非终结符下面的符号序列。*简单短语*
94 (simple phrase)或*素短语*(prime phrase)是不含更小短语的短语。也就是说，简单短语是由一个非终结符直接推出的符号序列。句型的*句柄*(handle)是其最左简单短语。(简单短语不可能重叠，因此“最左”是没有歧义的。)考虑图4-1中的分析树以及句型F(V Tail)。F和V Tail是简单短语，而F是句柄。句柄非常重要，因为它们代表可被不同语法分析技术所识别的特别推导步骤。

仅限于形如A → a B和C → λ的产生式的CFG形成正则文法（regular grammar）类。正如其名字所暗示，正则文法（精确）定义了正则集类（见练习6）。在第3章我们注意到语言$\{[^i]^i \mid i \geq 1\}$不是正则的。该语言可以由一个非常简单的CFG生成：

S → [T]
T → [T] | λ

该文法确定可由正则文法（正则集）定义的语言是上下文无关语言的真子集。

尽管CFG广泛用于定义程序设计语言的语法，并非所有的语法规则都能用CFG表达。例如，变量必须先定义再使用这一规则就无法在CFG中表达——没有办法可以传递程序体中定义的变量的精确集合。在实践中，不能在CFG中表示的语法细节被认为是静态语义的一部分，由语义例程（以及作用域和类型规则）来检查。

可以推广CFG以建立一个功能更为丰富的定义机制。上下文相关文法（context-sensitive grammar）要求仅当非终结符出现在特定上下文中（例如αAβ → αδβ）时才能被重写。0-型文法（type-0 grammar）则更为通用，它允许重写任意模式（例如α→ β）。尽管上下文相关文法和0-型文法比CFG更强大，但它们远不如CFG有用。问题是对于这些扩展的文法类不存在高效的语法分析器，而没有语法分析器就无法用文法定义来驱动编译器。然而，对于大多数种类的CFG，确实存在高效的语法分析器；因此，CFG代表了在通用性和实用性之间的一个极好的平衡。在本书中，我们将集中讨论CFG。无论何时我们提到一个文法（而没有说它是什么类型），都将假定该文法是上下文无关的。

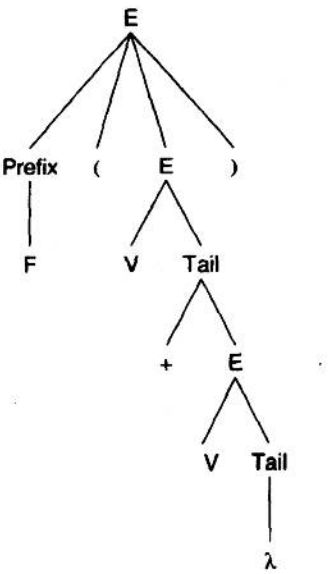


图4-1 相应于F(V+V)的分析树

4.2 上下文无关文法中的错误

CFG是一种定义机制。但是，它们可能含有错误，就像程序那样。某些错误容易检测和修正，而另一些则较难处理。

CFG的基本概念是，由开始符号开始，应用产生式直到产生终结符串。而某些CFG是有缺陷的，由于它们含有“无用的”非终结符。考虑下面的文法（G_1）：

S → A | B
A → a
B → B b
C → c

在G_1中，非终结符C不能从S（开始符号）到达，而非终结符B无法推导出终结符串。无法到达的以及不能推导出终结符串的非终结符被称为无用非终结符串。无用非终结符（以及含有它们的产生式）可从文法中安全地删除而不改变文法所定义的语言。包含无用非终结符的文法被称为非简化的（nonreduced）。在删除无用终结符后，该文法是简化的（reduced）。G_1是非简化的。在删除了B和C之后，得到等价文法G_2，它是简化的：

S → A
A → a

用于检测无用非终结符的算法是容易写出的（见练习7）。许多语法分析器生成器检查文法是不是简化的。如果不是，则文法中可能含有错误（通常由文法规范中的笔误导致）。

更为严重的文法缺陷是，有时文法允许一个程序有两棵或更多棵不同的分析树（因此成为一个非惟一结构）。例如，考虑下面的文法，它生成仅使用中缀“-”的表达式：

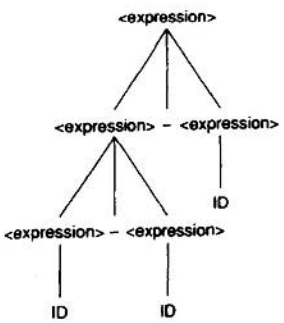
```
<expression> → <expression> - <expression>
<expression> → ID
```

该文法对于ID-ID-ID允许有两棵不同的分析树，如图4-2和图4-3所示。

若文法对于同一个终结符串允许有不同的分析树，那么这样的文法被称为是二义的（ambiguous）。它们很少使用，因为不能保证对所有输入都有惟一结构（即分析树），因此可能无法获得由分析树结构所指导的惟一转换。我们通常都限制只使用非二义（unambiguous）文法以保证惟一结构。

自然地，我们希望有一个算法能检查文法是不是二义的。然而，不可能确定一个给定的CFG是不是二义的（Hopcroft and Ullman 1969），因此不可能构造出这样一个算法。幸运的是，对于某些文法类（包括那些可以生成语法分析器的文法类），可以证明它们是非二义的。

96 文法中可能含有的最严重的潜在缺陷是它产生“错误的语言”。这一点非常难以察觉，因为文法通常被当作语言的定义。通常，通过比较那些我们期望是有效的输入和该文法的一个语法分析器所实际接受的输入，来非形式化地检测文法的正确性。尽管很少那样做，我们还是可以试着比较由一对文法定义的语言的等价性（将其中一个作为标准）。对于某些文法类，这种比较可以完成；对其他类，则没有已知的比较算法。我们已经认识到不可能找到适用于所有CFG的通用比较算法。



97 图4-2 ID-ID-ID的一棵分析树

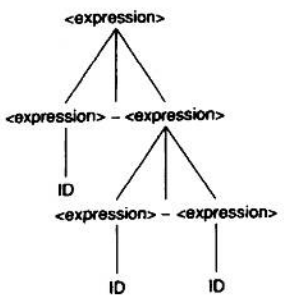


图4-3 ID-ID-ID的另一棵分析树

4.3 转换扩展BNF文法

正如我们在第2章所注意到的，扩展BNF文法在表示许多程序设计语言结构时非常有用，因为它们允许使用方括号指定的可选项，而且允许使用大括号指定的可选项列表结构。在分析文法和构造语法分析器时，假想一个更简单的“标准形式”文法是有益的。我们可以使用图4-4的算法来将扩展BNF转换成标准形式。

```
for (each production P = A → α [X1 ... Xn] β) {
    Create a new nonterminal, N.
    Replace production P with P' = A → α N β
    Add the productions: N → X1 ... Xn and N → λ
}

for (each production Q = B → γ {Y1 ... Ym} δ) {
    Create a new nonterminal, M.
    Replace production Q with Q' = B → γ M δ
    Add the productions: M → Y1 ... Ym M and
                         M → λ
}
```

图4-4 用于将扩展BNF文法转换成标准形式的算法

4.4 语法分析器与识别器

假定我们以某种方式知道一个文法是无二义的。给定一个输入串作为词法记号序列，可以询问：该输入在语法上是有效的吗？（即：它能由该文法产生吗？）完成这项工作的布尔值测试算法被称为识别器（recognizer）。

我们可能需要该算法完成更多的工作并询问：这个输入有效吗？如果该输入有效，它的结构（分析树）是什么？解答这个更一般问题的算法被称为语法分析器（parser）。由于我们计划使用语言结构来驱动编译器，因此我们对于语法分析器尤其感兴趣。

有两种通用的语法分析方法。第一种方法，包括在第2章中研究的递归下降技术，称为自顶向下的（top-down）。如果一个分析器从树的顶端（开始符号）开始"发现"词法记号序列所对应的分析树，并随后以深度优先的方式（通过预测）对其进行扩展，则它被认为是自顶向下的。自顶向下的语法分析器对应分析树的前序遍历。自顶向下的语法分析技术本质上是预测性的（predictive），因为它们总是在实际匹配开始之前预测将要被匹配的产生式。 98

很多的语法分析技术采用了另一种方法。它们属于自底向上的（bottom-up）语法分析器类。正如其名字所暗示的，自底向上的语法分析器从分析树的底部（树的叶结点，它们都是终结符号）开始发现其结构，并确定用来生成叶结点的产生式。随后发现用来生成叶结点的直接父结点的产生式。这种发现将持续到语法分析器到达用于扩展开始符号的产生式为止。在这时，已经确定了全部的分析树。自底向上的语法分析器对应分析树的后序遍历。

为对比自顶向下和自底向上语法分析的不同之处，考虑下面的简单文法，它生成一个程序设计语言的块结构框架：

<Program> → **begin** <Stmts> **end $**
<Stmts> → <Stmt> ; <Stmts>
<Stmts> → λ
<Stmt> → SimpleStmt
<Stmt> → **begin** <Stmts> **end**

文法 G_3

begin SimpleStmt; SimpleStmt; **end** $的自顶向下的语法分析如图4-5所示。

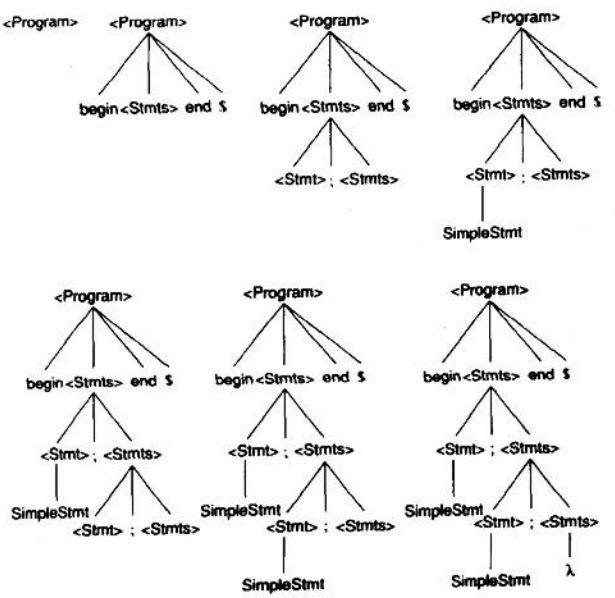


图4-5 一个自顶向下的语法分析过程

自底向上的语法分析通过发现子树并将它们链接成逐渐增大的树来进行。如图4-6所示。

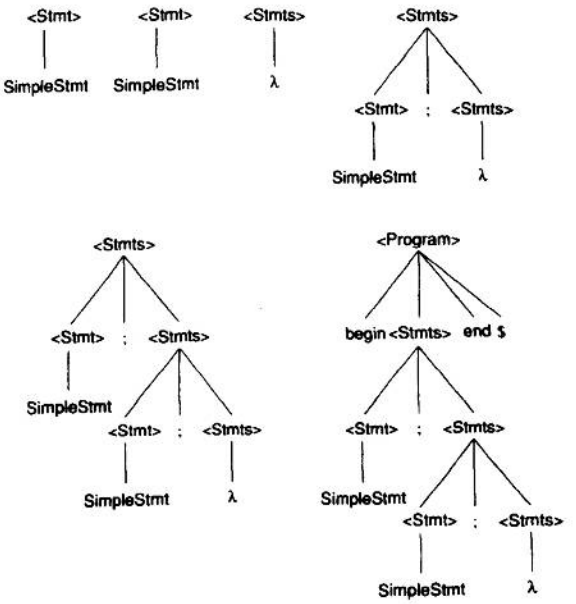


图4-6 一个自底向上的语法分析过程

由自顶向下的语法分析器所预测的产生式代表最左推导；因此，如前面所注意到的那样，这样的语法分析器产生最左分析。由自底向上的语法分析器所识别的产生式序列叫做最右分析；而又如前面所注意到的，它是代表最右推导的产生式序列的精确逆序。

当指定一种语法分析技术时，必须声明它产生最左分析还是最右分析。已知最好的且广泛使用的自顶向下和自底向上的语法分析策略分别称为LL和LR。这些名字看起来很神秘，其实它们只是简单地对如何读取输入以及由此所产生的语法分析的类型进行编码。因而，在这两个名字中，第一个L声明词法记号序列将从左到右地进行分析。第二个字母（L或R）则声明会产生最左还是最右分析。可以进一步通过包含语法分析器用来做出语法分析选择的超前搜索符号（即超出当前词法记号之外的符号）的数目来描述语法分析技术的特征。单个符号的超前搜索是最常见的，因此我们经常遇到LL(1)或LR(1)语法分
99 析器或它们的近似变体。

4.5 文法分析算法

通常必须分析文法的属性来确定一个文法是否为语法分析做好准备；如果是，则构造可用来驱动语法分析算法的表格。本节讨论许多重要的分析算法。我们的讨论将用来强化文法和推导的基本概念。而且，在构造语法分析器时实际需要这里所讨论的许多技术，并且它们将作为实际的语法分析器生成器的组件。

为使我们（在本章和随后的两章中）的介绍更为具体，我们将使用下列简化的数据结构来描述文法。特别是，我们将忽略在产品级程序中必须认真管理的动态内存分配的细节，并简单地对待每个数组值元
100 素，就像它是静态分配的。

文法G由一个记录（C语言的`struct`）来表示。它有名为`terminals`、`nonterminals`、`productions`和`start_symbol`的域。`productions`域是由`struct`组成的数组，其中每个`struct`都

有三个域lhs、rhs和rhs_length。另外，有时引用整个vocabulary也是有用的，它是terminals和nonterminals的组合。（在实践中不会实际复制vocabulary信息。）

```
typedef int symbol;   /* a symbol in the grammar */
/*
 * The symbolic constants used below, NUM_TERMINALS,
 * NUM_NONTERMINALS, and NUM_PRODUCTIONS are
 * determined by the grammar.  MAX_RHS_LENGTH should
 * simply be "big enough."
 */
#define VOCABULARY  (NUM_NONTERMINALS + NUM_TERMINALS)

typedef struct gram {
   symbol terminals[NUM_TERMINALS];
   symbol nonterminals[NUM_NONTERMINALS];
   symbol start_symbol;
   int num_productions;
   struct prod {
      symbol lhs;
      int rhs_length;
      symbol rhs[MAX_RHS_LENGTH];
   } productions[NUM_PRODUCTIONS];
   symbol vocabulary[VOCABULARY];
} grammar;

typedef struct prod production;

typedef symbol terminal;
typedef symbol nonterminal;
```

101

最普通的文法计算之一是确定什么非终结符能够推导出λ。这个信息非常重要，因为可以推导出λ的非终结符可能在分析过程中消失，因此必须谨慎地加以处理。确定一个非终结符是否能够推导出λ并非微不足道，因为推导可能需要多步。例如，可能有序列

$$A \Rightarrow BCD \Rightarrow BC \Rightarrow B \Rightarrow \lambda$$

为了进行这个计算，我们使用一个迭代标记算法。首先，要标记出能够直接（一步）推导出λ的非终结符。然后，要找到需要分析树高为2的非终结符。我们继续该过程，找出需要高度不断增长的分析树的非终结符，直到没有更多的非终结符可以被标记为能够推导出λ。完整的算法如图4-7所示。

当构造语法分析器时，通常通过分析文法来计算一个集合Follow(A)，其中A是任意非终结符。非正式地，Follow(A)是可以在某个句型中跟随A的终结符的集合。如果A能够作为最右符号在一个句型中出现，则λ包含在Follow(A)中（表示可能没有任何符号跟随A）。更精确地，对于$A \in V_n$，

$$Follow(A) = \{a \in V_t \mid S \Rightarrow^+ \cdots Aa \cdots\} \cup (\text{if } S \Rightarrow^+ \alpha A \text{ then } \{\lambda\} \text{ else } \varnothing)$$

Follow集很有用，因为它们定义了与一个给定非终结符一致的正确的上下文。也就是说，Follow(A)提供的超前搜索字符能够通告已识别出以A为左边的产生式。

用于构造语法分析器的另一个常用集合是First(α)。First(α)是所有能够开始一个可由α推导出的句型的终结符号的集合。如果$\alpha \Rightarrow^* \lambda$，则集合中也包含λ。正式地，

$$First(\alpha) = \{a \in V_t \mid \alpha \Rightarrow^* a\beta\} \cup (\text{if } \alpha \Rightarrow^* \lambda \text{ then } \{\lambda\} \text{ else } \varnothing)$$

如果α是一个产生式的右边，则First(α)包含的终结符号能够作为可由α推导出的字符串的起始符号。

我们将用一个数组`first_set[X]`来代表文法的First集，其中X是任意单独的词汇表符号。`first_set`中的元素是终结符号和λ。类似地，我们的Follow集表示将是一个数组`follow_set[A]`，其中A是非终结符号。而`follow_set`中的元素也是终结符号和λ。

对任意（由非终结符和终结符混合组成的）字符串α，无法预先计算其First和Follow集，因此使用算法`compute_first(α)`，它返回由First(α)所定义的终结符集合。如果α恰好仅含有一个符号，`compute_first(α)`将简单地返回`first_set[α]`。`compute_first()`和相关定义如图4-8所示。

```
typedef short boolean;
typedef boolean marked_vocabulary[VOCABULARY];

/*
 * Mark those vocabulary symbols found to
 * derive λ (directly or indirectly).
 */
marked_vocabulary mark_lambda(const grammar g)
{
   static marked_vocabulary derives_lambda;
   boolean changes;
                  /* any changes during last iteration? */
   boolean rhs_derives_lambda;
                  /* does the RHS derive λ? */
   symbol v;      /* a word in the vocabulary */
   production p;  /* a production in the grammar */
   int i, j;      /* loop variables */

   for (v = 0; v < VOCABULARY; v++)
      derives_lambda[v] = FALSE;
      /* initially, nothing is marked */

   do {
      changes = FALSE;
      for (i = 0; i < g.num_productions; i++) {
         p = g.productions[i];
         if (! derives_lambda[p.lhs]) {
            if (p.rhs_length == 0) {
               /* derives λ directly */
               changes = derives_lambda[p.lhs] = TRUE;
               continue;
            }
            /* does each part of RHS derive λ? */
            rhs_derives_lambda = derives_lambda[p.rhs[0]];
            for (j = 1; j < p.rhs_length; j++)
               rhs_derives_lambda = rhs_derives_lambda
                           && derives_lambda[p.rhs[j]];

            if (rhs_derives_lambda)
               changes = TRUE;
               derives_lambda[p.lhs] = TRUE;
         }
      }
   } while (changes);
   return derives_lambda;
}
```

102
~
103

图4-7 用于确定一个非终结符是否可推导出λ的算法

```
typedef set_of_terminal_or_lambda termset;
termset follow_set[NUM_NONTERMINAL];
termset first_set[SYMBOL];
marked_vocabulary derives_lambda = mark_lambda(g);
/* mark_lambda(g) as defined above */

termset compute_first(string_of_symbols alpha)
{
 int i, k;
 termset result;

 k = length(alpha);
 if (k == 0)
    result = SET_OF( λ );
 else {
   result = first_set[alpha[0]];
   for (i = 1; i < k && λ ∈ first_set[alpha[i-1]]; i++)
     result = result ∪ (first_set[alpha[i]] - SET_OF( λ ));

   if (i == k && λ ∈ first_set[alpha[k - 1]])
     result = result ∪ SET_OF( λ );
 }
 return result;
}
```

104

图4-8 用于计算First(alpha)的算法

fill_first_set()的定义见图4-9。它初始化first_set。该算法迭代执行，首先考虑单独的产生式，然后考虑产生式链。

```
extern grammar g;

void fill_first_set(void)
{
   nonterminal A;
   terminal    a;
   production  p;
   boolean     changes;
   int         i, j;

   for (i = 0; i < NUM_NONTERMINAL; i++) {
      A = g.nonterminals[i];
      if (derives_lambda[A])
         first_set[A] = SET_OF( λ );
      else
         first_set[A] = ∅;
   }
   for (i = 0; i < NUM_TERMINAL; i++) {
      a = g.terminals[i];
      first_set[a] = SET_OF( a );
      for (j = 0; j < NUM_NONTERMINAL; j++) {
         A = g.nonterminals[j];
         if (there exists a production A → aβ)
            first_set[A] = first_set[A] ∪ SET_OF( a );
      }
   }
   do {
      changes = FALSE;
      for (i = 0; i < g.num_productions; i++) {
         p = g.productions[i];
         first_set[p.lhs] = first_set[p.lhs] ∪
            compute_first(p.rhs);
         if ( first_set changed )
            changes = TRUE;
      }
   } while (changes);
}
```

图4-9 用于计算V的First集的算法

105

fill_first_set()的执行如下所示（其中使用了4.1节中的文法G_0）：

Step	first_set							
	E	Prefix	Tail	(	)	V	F	+
(1) First loop	∅	{λ}	{λ}					
(2) Second (nested) loop	{V}	{F,λ}	{+,λ}	{(}	{)}	{V}	{F}	{+}
(3) Third loop, production 1	{V,F,(}	{F,λ}	{+,λ}	{(}	{)}	{V}	{F}	{+}

fill_follow_set()的定义见图4-10。它初始化follow_set。该算法定位在产生式中出现的非终结符，然后对跟随该非终结符的产生式后缀计算其First值。如果λ在First集中，则将产生式左边符号的Follow集也加入follow_set中。

```
void fill_follow_set(void)
{
   nonterminal A, B;
   int i;
   boolean     changes;

   for (i = 0; i < NUM_NONTERMINAL; i++) {
      A = g.nonterminals[i];
```

图4-10 用于计算所有非终结符的Follow集的算法

```
        follow_set[A] = ∅;
    }
    follow_set[g.start_symbol] = SET_OF( λ );

    do {
        changes = FALSE;
        for (each production A → αBβ) {
            /*
             * I.e. for each production and each occurrence
             * of a nonterminal in its right-hand side.
             */
            follow_set[B] = follow_set[B] ∪
                    (compute_first(β) - SET_OF( λ ));
            if ( λ ∈ compute_first(β) )
                follow_set[B] = follow_set[B] ∪ follow_set[A];
            if ( follow_set[B] changed )
                changes = TRUE;
        }
    } while (changes);
}
```

106

图4-10 （续）

作为示例，我们使用4.1节中的文法G_0来说明fill_follow_set()的执行：

Step	follow_set		
	E	Prefix	Tail
(1) Initialization	{λ}	∅	∅
(2) Process Prefix in production 1	{λ}	{(}	∅
(3) Process E in production 1	{λ,)}	{(}	∅
(4) Process Tail in production 2	{λ,)}	{(}	{λ,)}

可以推广First和Follow集，使其包括长度为k而不是长度为1的字符串。$First_k(\alpha)$是可以由α推导出的长度为k个符号的终结符前缀集合。类似地，$Follow_k(A)$是可以在某些句型中跟随A的k个符号的终结符串集合。$First_k$和$Follow_k$用于定义使用k个符号的超前搜索的语法分析技术（例如LL(k)和LR(k)）。可以推广用于计算$First_1$和$Follow_1$的算法，来计算$First_k$和$Follow_k$。

计算First和Follow集的更多示例

为了进一步说明用于计算First和Follow集的程序的运行，这里给出两个额外的示例。对于下面的文法：

S → a S e
S → B
B → b B e
B → C
C → c C e
C → d

fill_first_set的执行过程如下：

Step	first_set							
	S	B	C	a	b	c	d	e
(1) First loop	∅	∅	∅					
(2) Second (nested) loop	{a}	{b}	{c,d}	{a}	{b}	{c}	{d}	{e}
(3) Third loop, production 2	{a,b}	{b}	{c,d}	{a}	{b}	{c}	{d}	{e}
(4) Third loop, production 4	{a,b}	{b,c,d}	{c,d}	{a}	{b}	{c}	{d}	{e}
(5) Third loop, production 2	{a,b,c,d}	{b,c,d}	{c,d}	{a}	{b}	{c}	{d}	{e}

107

fill_follow_set的执行说明如下：

Step	follow_set		
	S	B	C
(1) Initialization	{λ}	∅	∅
(2) Process S in production 1	{e,λ}	∅	∅
(3) Process B in production 2	{e,λ}	{e,λ}	∅
(4) Process B in production 3	no changes		
(5) Process C in production 4	{e,λ}	{e,λ}	{e,λ}
(6) Process C in production 5	no changes		

对于第二个示例文法

S → A B c
A → a
A → λ
B → b
B → λ

fill_first_set的执行过程如下:

Step	first_set					
	S	A	B	a	b	c
(1) First loop	∅	{λ}	{λ}			
(2) Second (nested) loop	∅	{a,λ}	{b,λ}	{a}	{b}	{c}
(3) Third loop, production 1	{a,b,c}	{a,λ}	{b,λ}	{a}	{b}	{c}

fill_follow_set的执行说明如下:

Step	follow_set		
	S	A	B
(1) Initialization	{λ}	∅	∅
(2) Process A in production 1	{λ}	{b,c}	∅
(3) Process B in production 1	{λ}	{b,c}	{c}

练习

1. 使用4.3节的算法，将Micro的扩展CFG定义（第2章的图2-4）转换为标准形式的CFG。 108
2. 对定义Micro的CFG（扩展形式或者标准形式）进行扩展，使之包含相等运算符“=”以及幂运算符“**”。相等运算应当比加和减的优先级低，而幂运算应当比加和减的优先级高。即，A+B**2 = C+D应当等价于(A+(B**2)) = (C+D)。而且，幂运算应当从右边起进行分组（因此A**B**C等价于A**(B**C)），而相等运算则根本不应当分组（A = B = C是非法的）。确保你的文法是无二义的。
3. 使用fill_first_set()和fill_follow_set()来计算定义Micro的文法的first_set和follow_set。
4. 形如A → Aα的产生式被称为*左递归的*（left-recursive）。类似地，形如B → βB的产生式被称为*右递归的*（right-recursive）。证明：任意同时包含相同左边符号的左递归和右递归产生式的文法必定是二义的。
5. 假定希望从n个选项的集合{O_1, … , O_n}中生成可选项列表。该列表能够以任意次序包含可选项的任意子集，但可选项不能重复。写出一个CFG，生成所期望形式的列表。
   你的文法大小和可能的选择数量n之间有什么关系?
   如果进一步要求选项以特殊的次序出现（例如，仅当i<j时O_i可以出现在O_j之前），你的文法会更加简化还是会更加复杂?
6. 通过写出将正则文法转换为有限自动机以及进行相反转换的算法，证明正则文法和有限自动机有同等的定义能力。
7. 可通过删除无用非终结符和含有无用非终结符的产生式来化简CFG。我们可以通过首先删除从开始符号不可达的非终结符，再删除不推导出任何终结符串的非终结符，来化简一个文法。另外，也可以首先删除不推导出任何终结符串的非终结符，再删除不可达非终结符。这两种方法等价么？如果不，哪一种方法的顺序比较好?
8. 简要证明对于所有的词汇表符号，fill_first_set()都能正确地计算first_set的值。
9. 令G为任意CFG，并假定λ ∉ L(G)。证明：G能够被转换为不使用λ产生式的等价CFG。
10. *单位产生式*（unit production）是形如A → B的产生式。其中B是单个非终结符。证明：包含单位产生式的任意CFG都能被转换为不使用单位产生式的等价CFG。

11. 某些CFG生成无限大的语言；其他的CFG则产生有限大的语言。写出一个算法，测试一个给定的CFG是否产生无限的语言。

109 提示：使用练习9和练习10的结果来简化分析。

12. 令G为不含λ产生式的无二义CFG。如果$x \in L(G)$，证明：推导x所需步数和x的长度呈线性关系。

110 13. 编写一个程序，使用4.3节的算法，将扩展CFG转换为等价的标准形式的CFG。

第5章 LL(1)文法及分析器

在第2章中，我们学习了递归下降分析的基本知识。现在来研究LL(1)文法（LL(1)grammer），这是一类适合递归下降分析的CFG。我们还同时定义LL(1)分析器（LL(1) parser），它使用LL(1)分析表（LL(1)parse table）而不是递归过程来控制分析。

正如我们已经看到的，CFG是一种非常有用的定义机制（definitional mechanism）。它们也经常用来自动生成语法分析器。这里的思想是使用一个程序——语法分析器生成器（parser generator），以任意文法类作为输入，产生一个语法分析器作为输出，这个语法分析器将正确分析由该文法所定义的语言。这个概念与编译器的概念类似——高级定义（源程序或文法）被翻译成可执行的形式（目标程序或语法分析器）。这种方法使得构造一个语法分析器成为构造编译器过程中最容易的一部分工作——写出一个文法，然后将它输入到自动的语法分析器生成器中。修改语法分析器（例如，向语言中添加新的结构）也同样很简单。新的语法分析器由更新过的文法创建。

5.1 LL(1)Predict函数

正如我们在第2章中所学习的，递归下降分析器使用分析过程来匹配由非终结符生成的词法记号。在构造分析过程中的主要问题是确定匹配哪一个产生式。这个决定可以通过定义一个Predict函数进行形式化。该函数检查超前搜索符号，来推断必须使用哪个产生式来扩展每个非终结符。

考虑产生式$A \rightarrow X_1 \cdots X_m$, $m > 0$。需要计算可能指示匹配该产生式的可能的超前搜索记号集。该集合无疑是那些可以由$X_1 \cdots X_m$产生的终结符。由于仅超前搜索一个词法记号，因此我们需要可能产生的首记号集（即最左记号集）。如我们在第4章中所学到的，该记号集是$First(X_1 \cdots X_m)$。

先考虑最左符号X_1。如果X_1是终结符，则$First(X_1 \cdots X_m) = X_1$。如果$X_1$是非终结符，则计算每个对应于$X_1$的产生式右部的First集。当$X_1$能生成λ时怎么办？那样的话，$X_1$实际上能够被抹去，而$First(X_1 \cdots X_m)$依赖于$X_2$。特别地，如果$X_2$是终结符，则它将被包含于$First(X_1 \cdots X_m)$中。如果$X_2$是非终结符，则计算每个对应于$X_2$的产生式右部的First集。类似地，如果$X_1$和$X_2$都能产生λ，则考虑$X_3$，依次类推。如果产生式右部的所有符号都能产生λ该怎么办？那样的话，超前搜索符号需要由跟随左部符号（在我们的例子中是A）的那些终结符确定。为此可以使用Follow(A)，它是在某些合法推导中可以跟随A的词法记号集。

现在定义导致预测产生式$A \rightarrow X_1 \cdots X_m$的超前搜索记号集。该集合称为Predict:

$$
\begin{aligned}
&Predict(A \rightarrow X_1 \cdots X_m) = \\
&\quad \text{if } \lambda \in First(X_1 \cdots X_m) \\
&\qquad (First(X_1 \cdots X_m) - \lambda) \cup Follow(A) \\
&\quad \text{else} \\
&\qquad First(X_1 \cdots X_m)
\end{aligned}
$$

即，任意词法记号（可以是由产生式右部产生的首符号）将预测该产生式。进一步，如果产生式右部整个能够产生λ[$\lambda \in First(X_1 \cdots X_m)$]，则能够紧跟在产生式左部非终结符后的词法记号也将预测该产生式。因为λ不是终结符号，所以它不能作为超前搜索符号，且因此不包含于任何Predict集合中。

还必须处理最后一个问题。如果对于两个产生式，$A \rightarrow X_1 \cdots X_m$和$A \rightarrow Y_1 \cdots Y_p$，存在某个词法记号t，其中$t \in Predict(A \rightarrow X_1 \cdots X_m)$且$t \in Predict(A \rightarrow Y_1 \cdots Y_p)$，该如何处理？即，如果同样的词法记

112 号预测多个产生式该怎么办？该冲突将会把这些文法排除在LL(1)文法类之外。LL(1)包含那些对共享公共左部的产生式拥有不相交的预测集合的文法。从经验得知，通常有可能为一个程序设计语言构造LL(1)文法。然而，并非所有的文法都是LL(1)文法。许多非LL(1)的文法属于其他（更复杂的）文法类，它们的语法分析器也能够被自动地构造。

为了确认我们已经理解了刚刚介绍的概念，看一下怎样为Micro的产生式计算Predict函数。

图5-1给出使用4.3节的算法（见图4-4）将第2章的Micro文法转换成的标准形式。“$”用来表示文件结束记号。

```
 1   <program>       → begin <statement list> end
 2   <statement list> → <statement> <statement tail>
 3   <statement tail> → <statement> <statement tail>
 4   <statement tail> → λ
 5   <statement>     → ID := <expression> ;
 6   <statement>     → read ( <id list> ) ;
 7   <statement>     → write ( <expr list> ) ;
 8   <id list>       → ID <id tail>
 9   <id tail>       → , ID <id tail>
10   <id tail>       → λ
11   <expr list>     → <expression> <expr tail>
12   <expr tail>     → , <expression> <expr tail>
13   <expr tail>     → λ
14   <expression>    → <primary> <primary tail>
15   <primary tail>  → <add op> <primary> <primary tail>
16   <primary tail>  → λ
17   <primary>       → ( <expression> )
18   <primary>       → ID
19   <primary>       → INTLIT
20   <add op>        → +
21   <add op>        → −
22   <system goal>   → <program> $
```

图5-1 标准形式的Micro文法

图5-2和图5-3给出Micro中非终结符的First和Follow集，它使用4.5节的技术（见图4-8～图4-10）计算得到。对于像Micro一样小的文法，这些集合可以简单地通过手算得到。对于大的文法，使用自动工具计算会较为安全。（5.8节中的LLGen工具有一个选项可令其列出First和Follow集。）

Nonterminal	First Set
<program>	{**begin**}
<statement list>	{ID, **read**, **write**}
<statement>	{ID, **read**, **write**}
<statement tail>	{ID, **read**, **write**, λ}
<expression>	{ID, INTLIT, (}
<id list>	{ID}
<expr list>	{ID, INTLIT, (}
<id tail>	{COMMA ,λ}
<expr tail>	{COMMA ,λ}
<primary>	{ID, INTLIT, (}
<primary tail>	{+, −, λ}
<add op>	{+, −}
<system goal>	{**begin**}

图5-2 Micro的First集

113 下一步，通过替换First和Follow集并简化表达式来计算Predict集，如图5-4所示。

NonTerminal	Follow Set
<program>	{$}
<statement list>	{**end**}
<statement>	{ID, **read**, **write**, **end**}
<statement tail>	{**end**}
<expression>	{COMMA, SEMICOLON,)}
<id list>	{)}
<expr list>	{)}
<id tail>	{)}
<expr tail>	{)}
<primary>	{COMMA, SEMICOLON, +, −,)}
<primary tail>	{COMMA, SEMICOLON,)}
<add op>	{ID, INTLIT, (}
<system goal>	{λ}

图5-3 Micro的Follow集

Prod	Predict Set		
1	First(**begin** <statement list> **end**) =	First(**begin**) =	{**begin**}
2	First(<statement> <statement tail>) =	First(<statement>) =	{ID, **read**, **write**}
3	First(<statement> <statement tail>) =	First(<statement>) =	{ID, **read**, **write**}
4	(First(λ)−λ) ∪ Follow(<statement tail>) =	Follow(<statement tail>) =	{**end**}
5	First(ID := <expression> ;) =	First(ID) =	{ID}
6	First(**read** (<id list>) ;) =	First(**read**) =	{**read**}
7	First(**write** (<expr list>) ;) =	First(**write**) =	{**write**}
8	First(ID <id tail>) =	First(ID) =	{ID}
9	First(, ID <id tail>) =	First(,) =	{,}
10	(First(λ)−λ) ∪ Follow(<id tail>) =	Follow(<id tail>) =	{)}
11	First(<expression> <expr tail>) =	First(<expression>) =	{ID, INTLIT, (}
12	First(, <expression> <expr tail>) =	First(,) =	{,}
13	(First(λ)−λ) ∪ Follow(<expr tail>) =	Follow(<expr tail>) =	{)}
14	First(<primary> <primary tail>) =	First(<primary>) =	{ID, INTLIT, (}
15	First(<add op> <primary> <primary tail>) =	First(<add op>) =	{+, −}
16	(First(λ)−λ) ∪ Follow(<primary tail>) =	Follow(<primary tail>) =	{COMMA, ;,)}
17	First((<expression>)) =	First(() =	{(}
18	First(ID) =		{ID}
19	First(INTLIT) =		{INTLIT}
20	First(+) =		{+}
21	First(−) =		{−}
22	First(<program> $) =	First(<program>) =	{**begin**}

图5-4 为Micro计算Predict集

5.2 LL(1)分析表

包含在Predict函数中的语法分析信息可以被方便地用LL(1)分析表来表示。该表记为T，其形式为:

$$T : V_n \times V_t \rightarrow P \cup \{Error\}$$

其中，P是所有产生式的集合。如果A是待匹配的非终结符，而t是超前搜索记号，则T[A][t]指示预测哪个产生式。如果没有合适的产生式，则T[A][t]产生一个Error值，指示语法错误。T定义如下:

$T[A][t] = A \rightarrow X_1 \cdots X_m$ if $t \in$ Predict$(A \rightarrow X_1 \cdots X_m)$;
$T[A][t]$ = Error otherwise

114
115

文法G是LL(1)的，当且仅当T中的所有条目包含惟一的产生式或错误标志。换一种说法，如果G是LL(1)的，则对任意非终结符A和任意超前搜索符号t，不允许存在$t \in$ Predict$(A \rightarrow \alpha)$ 和$t \in$ Predict$(A \rightarrow \beta)$，其中 $A \rightarrow \alpha$ 和 $A \rightarrow \beta$ 是不同的产生式。

现在容易明白为什么LL(1)文法这么适于自顶向下的语法分析。如果一个文法是LL(1)的，则保证对任何可能的非终结符和超前搜索符号的组合，都有一个惟一的预测，或一个错误标志。所有预测都是惟一的，因此容易把非终结符替换为正确的产生式右部。这就允许我们从开始符号开始，向下处理作为树的叶结点的输入符号，来为任意合法输入字符串构造分析树。

作为示例，将前面计算的Micro的Predict集转换为图5-5中的LL(1)分析表。该表可被用来构造组成Micro的递归下降分析器的语法分析过程，或者，该表可被直接用来驱动一个LL(1)分析器。在图5-5中，整数条目是产生式编号；空白是错误条目。

	ID	INTLIT	:=	.	;	+	−	(	)	begin	end	read	write	$
<program>										1				
<statement list>	2											2	2	
<statement>	5											6	7	
<statement tail>	3										4	3	3	
<expression>	14	14						14						
<id list>	8													
<expr list>	11	11						11						
<id tail>				9					10					
<expr tail>				12					13					
<primary>	18	19						17						
<primary tail>				16	16	15	15		16					
<add op>						20	21							
<system goal>										22				

图5-5 Micro的LL(1)分析表

5.3 从LL(1)分析表构造递归下降分析器

当构造语法分析器时，由文法来计算LL(1)分析表。在LL(1)分析表中记录的语法分析决策可以被硬编码到由递归下降分析器所使用的语法分析过程中。回忆第2章中语法分析过程的形式为

116

```
void non_term(void)
{
   token tok = next_token();
   switch (tok) {
   case TERMINAL_LIST:
      parsing_actions();
      break;
    . . .
   default:
      syntax_error(tok);
      break;
   }
}
```

non_term()是该语法分析过程所处理的非终结符的名字。next_token()返回超前搜索符号。TERMINAL_LIST表示终结符号列表。parsing_actions()表示语法分析动作序列：调用语法分析过程来匹配非终结符，调用match()来匹配终结符。如果next_token()没有预测到合法的产生式，则调用syntax_error()。

更实际一些，图5-6的语法分析过程匹配Micro中的<statement>:

```
void statement(void)
{
   token tok;
   tok = next_token();
   switch (tok) {
```

图5-6 <statement>的语法分析过程

```
        case ID:
            match(ID); match(ASSIGNOP); expression();
            match(SEMICOLON);
            break;
        case READ:
            match(READ); match(LPAREN); id_list();
            match(RPAREN); match(SEMICOLON);
            break;
        case WRITE:
            match(WRITE); match(LPAREN); expr_list();
            match(RPAREN); match(SEMICOLON);
            break;
        default:
            syntax_error(tok);
            break;
        }
    }
```

图5-6 （续）

117

我们将考虑一个算法，它从文法的LL(1)分析表中自动地构造类似图5-6的语法分析过程。所有过程都将是这种形式。TERMINAL_LIST和parsing_actions()的值将会从LL(1)分析表进行实例化。

现在用文法中符号的名字（names）来扩充用以描述第4章中文法的数据结构。一个文法（grammar）现在是：

```
typedef int symbol;   /* a symbol in the grammar */

#define VOCABULARY  (NUM_NONTERMINALS + NUM_TERMINALS)

typedef struct gram {
   symbol terminals[NUM_TERMINALS];
   symbol nonterminals[NUM_NONTERMINALS];
   symbol start_symbol;
   int num_productions;
   struct prod {
      symbol lhs;
      int rhs_length;
      symbol rhs[MAX_RHS_LENGTH];
   } productions[NUM_PRODUCTIONS];
   symbol vocabulary[VOCABULARY];
   char *names[VOCABULARY];
} grammar;

typedef struct prod production;

typedef symbol terminal;
typedef symbol nonterminal;
```

假定我们拥有希望在一个语法分析过程中进行匹配的文法符号数组。例程gen_actions()（见图5-7）取文法符号作为参数，并生成在递归下降分析器中匹配它们所必需的动作（调用语法分析过程以及match()函数）。gen_actions()假定函数make_id()取文法符号名字作为参数，并将它转换为有效的程序标识符。这可能包括剔除像“<”和“>”这样的非法字符，删除内嵌的空白，对字符进行重命名，等等。例如，make_id("<statement list>")返回"statementlist"，而make_id(":=")返回"COLONEQUAL"。

make_parsing_proc()在图5-8中定义。该算法取一个非终结符和LL(1)分析表作为参数，为该终结符生成完整的语法分析过程。该例程假定有两个函数：prods()和rhs()。prods(A)返回以A作为左部的产生式集合。rhs(P)则返回产生式P右部的符号串。

118

一个很好的练习是：给定<statement>和图5-5所示的LL(1)分析表作为参数，来验证make_parsing_proc()将生成图5-6所示的语法分析过程，其中一些标识符可能被重新命名。

```
extern char *make_id(char *);

void gen_actions(symbol x[], int x_length);
{
   int i;
   char *id;

   /*
    * Generate recursive descent
    * actions needed to match x.
    */
   if (x_length == 0)
      printf ("; /* null */\n");
   else {
      for (i = 0; i < x_length; i++) {
         id = make_id(g.names[x[i]]);
         if (is_terminal(x[i]))
            printf("\t\tmatch(%s);\n", id);
         else
            printf("\t\t%s();\n", id);
      }
   }
}
```

图5-7 生成递归下降动作的算法

```
void make_parsing_proc(const nonterminal A,
                       const lltable T)
{
   /*
    * Generate recursive descent
    * parsing procedure for A.
    */
   extern grammar g;
   production p;
   terminal x;
   int i, j;

   printf("void %s(void)\n{\n", make_id(g.names[A]));
   printf("\ttoken tok = next_token()\n");
   printf("\tswitch (tok) {\n");

   /* for each production where A is the LHS */
   for (i = 0; i < g.num_productions; i++) {
      if (g.productions[i].lhs != A)
        continue;
      p = g.productions[i];
     /* for each terminal in the grammar */
     for (j = 0; j < NUM_TERMINALS; j++) {
       x = g.terminals[j];
       if (T[A][x] == i)  /* this production */
         printf("\tcase %s:\n", make_id(g.names[x]));
     }
     gen_actions(p.rhs, p.rhs_length);
     printf("\t\tbreak;\n");
   }
   printf("\tdefault:\n");
   printf("\t\tsyntax_error(tok);\n");
   printf("\tbreak;\n\t}\n}\n");
}
```

图5-8 生成语法分析过程的算法

很容易扩展make_parsing_proc()，使其对某些特殊的情形，生成更高效的语法分析过程。最重要的是，如果一个非终结符仅能以一种方式被扩展，则不必生成任何条件逻辑，语法分析过程的过程体就是简单的gen_actions(rhs(p))，其中p是要被匹配的产生式。该优化大量用在第2章的语法分析过程中。

5.4 LL(1)分析器驱动程序

通常用执行语义分析和代码生成的代码来扩充递归下降分析过程。因此，语法分析过程一旦被建立并集成到编译器中，就不容易更改。常常要更新表示程序设计语言语法的文法以适应新的或改进的结构。所以，希望有办法更新语法分析器而无需不必要地影响其他的编译器组件。

除了使用LL(1)分析表来构造语法分析过程外，还可能使用该表与驱动程序一起组成LL(1)分析器(LL(1) parser)。LL(1)分析表只在构造分析器时被计算一次。利用这些表控制语法分析的LL(1)驱动程序认为这些表是只读的。因为同样的驱动程序可以和所有LL(1)分析表一起使用，所以改变文法并构造新的语法分析器将很容易——计算新的LL(1)分析表并以之替换旧表。进一步，由于LL(1)驱动程序使用栈而不是递归过程调用来存储尚未被匹配的符号，因此，可以期望它所生成的语法分析器比相应的递归下降分析器更小且更快。 119 ~ 120

图5-9所示的LL(1)驱动程序非常简单。它把待匹配或待扩展的符号压入栈中。栈中的终结符号必须匹配输入符号；非终结符号则通过使用LL(1)分析表被扩展。

```
void lldriver(void)
{
   /* Push the Start Symbol onto an empty stack. */
   push(s);

   while (! stack_empty() ) {
      /* Let X be the top stack symbol; */
      /* let a be the current input token. */

      if (is_nonterminal(X)
            && T[X][a] == X → Y1 · · · Ym) {
         /* Expand non-terminal */
         pop(1);
         Push Ym, Ym-1, · · · Y1 onto the stack;
      } else if (X == a) {    /* X in terminals */
         pop(1);              /* Match of X worked */
         scanner(& a);        /* Get next token */
      } else
         /* Process syntax error. */
   }
}
```

图5-9 LL(1)分析器驱动程序

在此，重复2.5.5节的递归下降分析示例，这一次使用LL(1)分析表和驱动程序。图5-10显示出在给定输入**begin** A := BB-314 + A; **end** $时由LL(1)分析器所执行的步骤。

5.5 LL(1)动作符号

回忆第2章，形如#Name的动作符号被添加到文法中以标记需要语义动作的地方。动作符号并非文法的实际组成部分，并在计算LL(1)分析表时被忽略。在分析中，产生式中所出现的动作符号用来执行相应的语义动作。

在递归下降分析器的情形中，我们将扩展5.3节（图5-7）的gen_actions()例程来包含动作符号而不仅仅是文法符号。当处理动作符号时，它将被转换为对相应语义例程的调用。例如，调用 121
gen_actions("ID := <expression> #gen_assign ;")将会生成：

```
match(ID);
match(COLONEQUAL);
expression();
gen_assign();
match(SEMICOLON);
```

语义例程调用不传递显式参数。必要的参数通过语义栈（semantic stack）来传送。语义栈的使用将在第7章中详细讲述。需要强调，语义栈和分析栈是完全不同的数据结构。在递归下降分析器中，分析栈“隐藏”于运行中的编译器的过程调用栈中。通过让每个分析例程返回一个语义记录，语义栈也可以这
122 样隐藏。

Step	Parser Action	Remaining Input	Parse Stack
(1)	Predict 22	**begin** A:=BB-314+A ; **end** $	<system goal>
(2)	Predict 1	**begin** A:=BB-314+A ; **end** $	<program> $
(3)	Match	**begin** A:=BB-314+A ; **end** $	**begin** <statement list> **end** $
(4)	Predict 2	A:=BB-314+A ; **end** $	<statement list> **end** $
(5)	Predict 5	A:=BB-314+A ; **end** $	<statement> <statement tail> **end** $
(6)	Match	A:=BB-314+A ; **end** $	ID := <expression> ; <statement tail> **end** $
(7)	Match	:=BB-314+A ; **end** $	:= <expression> ; <statement tail> **end** $
(8)	Predict 14	BB-314+A ; **end** $	<expression> ; <statement tail> **end** $
(9)	Predict 18	BB-314+A ; **end** $	<primary> <primary tail> ; <statement tail> **end** $
(10)	Match	BB-314+A ; **end** $	ID <primary tail> ; <statement tail> **end** $
(11)	Predict 15	-314+A ; **end** $	<primary tail> ; <statement tail> **end** $
(12)	Predict 21	-314+A ; **end** $	<add op> <primary> <primary tail> ; <statement tail> **end** $
(13)	Match	-314+A ; **end** $	- <primary> <primary tail> ; <statement tail> **end** $
(14)	Predict 19	314+A ; **end** $	<primary> <primary tail> ; <statement tail> **end** $
(15)	Match	314+A ; **end** $	IntLiteral <primary tail> ; <statement tail> **end** $
(16)	Predict 15	+A ; **end** $	<primary tail> ; <statement tail> **end** $
(17)	Predict 20	+A ; **end** $	<add op> <primary> <primary tail> ; <statement tail> **end** $
(18)	Match	+A ; **end** $	+ <primary> <primary tail> ; <statement tail> **end** $
(19)	Predict 18	A ; **end** $	<primary> <primary tail> ; <statement tail> **end** $
(20)	Match	A ; **end** $	ID <primary tail> ; <statement tail> **end** $
(21)	Predict 16	; **end** $	<primary tail> ; <statement tail> **end** $
(22)	Match	; **end** $	; <statement tail> **end** $
(23)	Predict 4	**end** $	<statement tail> **end** $
(24)	Match	**end** $	**end** $
(25)	Match	$	$

图5-10 **begin** A : = BB - 314 + A; **end** $的LL(1)分析

动作符号使得在递归下降分析器中包含语义动作非常容易。进一步，由于语义动作是从分析过程提取出来的，因此，为一个已更新的文法生成新的分析过程就是直截了当的。

对于LL(1)分析器，当预测到一个特定的产生式时，出现在其中的动作符号被压入分析栈中。当动作符号出现在分析栈的顶部时，它被弹出，且相应的语义例程被调用。因此，图5-9的LL(1)驱动程序被扩展，如图5-11所示。

5.6 文法的LL(1)化

对于缺乏经验的编译器作者来说，构造LL(1)文法并不总是那么容易。问题是LL(1)对于每个非终结符和超前搜索符号的组合都要求惟一的预测，而写出违反惟一预测要求的产生式并不困难。

幸运的是，大多数LL(1)预测冲突都能够被归为两类：*公共前缀*（common prefix）和*左递归*（left recursion）。消除公共前缀和左递归的简单文法变换已为人知，而这些变换允许我们将大多数文法改写
123 成有效的LL(1)形式。

在第一类冲突中，两个拥有相同左部的产生式的右部常常共享公共前缀。例如，可能有

<stmt> → **if** <expr> **then** <stmt list> **end if** ;
<stmt> → **if** <expr> **then** <stmt list> **else** <stmt list> **end if** ;

共享公共前缀的产生式导致预测冲突，因为每个右部的First集不是无交集的（除非公共前缀仅生成λ）。由公共前缀导致的预测冲突的解决办法是简单的提取变换，如图5-12所示。

```
void lldriver(void)
{
   /* Push the Start Symbol onto an empty stack */
   push(s);

   while (! stack_empty() ) {
      /* Let X be the top stack symbol; */
      /* let a be the current input token */

      if (is_nonterminal(X)
            && T[X][a] == X → Y1 · · · Ym) {
         /* Expand nonterminal */
         Replace X with Y1 · · · Ym on the stack;
      } else if (is_terminal(X) && X == a) {
         pop(1);          /* Match of X worked */
         scanner(& a);    /* Get next token */
      } else if (is_action_symbol(X)) {
         pop(1);
         Call Semantic Routine corresponding to X;
      } else
         /* Process syntax error */
   }
}
```

图5-11 处理动作符号的LL(1)驱动程序

```
void factor(grammar *G)
{
   while (G has 2 or more productions with the same
          LHS and a common prefix) {
      Let S = {A → αβ , . . . , A → αζ}
          be the set of productions with the same
          left-hand side, A, and a common prefix, α

      Create a new nonterminal, N;

      Replace S with the production set
          SET_OF( A → α N, N → β , . . . , N → ζ )
   }
}
```

图5-12 提取公共前缀的算法

使用**If-then-else**的例子，factor()产生

```
<stmt>      → if <expr> then <stmt list> <if suffix>
<if suffix> → end if ;
<if suffix> → else <stmt list> end if ;
```

在第二类冲突中，如果一个产生式的左部符号也是它右部的第一个符号，则该产生式是*左递归的*。例如，产生式E → E+T是左递归产生式。如果一个非终结符是某个左递归产生式的左部符号，则该非终结符是左递归的。包含左递归产生式的文法不可能是LL(1)的。为明白其原因，假定某个超前搜索符号t导致预测某个左递归产生式A → Aβ。在该预测之后，A再次成为栈顶符号，因而将会永远预测相同的产生式。

在图5-13中给出算法remove_left_recursion()，它从一个已经提取公共前缀的文法中删除左递归。 124

为理解remove_left_recursion()如何操作，观察在提取公共前缀后，共享左部符号的产生式集合中最多只能有一个产生式是左递归的。假定是A → Aα。可以应用左递归任意多次，但最终必须使用其他产生式中的一个，否则非终结符A将永远不会被消去。产生式A → NT生成N后面跟着零个或多个α。

```
void remove_left_recursion(grammar *G)
{
   while (G contains a left-recursive nonterminal) {
      Let S = {A → Aα, A → β, . . . , A → ζ}
          be the set of productions with the same
          left-hand side, A, where A is
          left-recursive.

      Create two new nonterminals, T and N;

      Replace S with the production set
          SET_OF( A → N T, N → β, . . . . , N → ζ,
              T → αT, T → λ )
   }
}
```

图5-13 删除左递归的算法

N生成β, … ζ 中的任意一个。例如，考虑下面的左递归表达式文法:

E → E + T
E → T
T → T * P
T → P
P → ID

该文法包含左递归，因此不是LL(1)的。它的形式适合自底向上的语法分析技术，因此可能出现在程序设计语言的文法中。remove_left_recursion()能够用来重写其中的两个左递归产生式，得到

E → E1 Etail
E1 → T
Etail → + T Etail
Etail → λ
T → T1 Ttail
T1 → P
Ttail → * P Ttail
Ttail → λ
125 P → ID

非终结符E1和T1各自都仅是一个产生式的左部，因此可以用相应的右部来替换它们，得到

E → T Etail
Etail → + T Etail
Etail → λ
T → P Ttail
Ttail → * P Ttail
Ttail → λ
P → ID

该文法等价于原来的文法，且是LL(1)的。事实上，它非常类似于通常用于程序设计语言的LL(1)文法。

提取公共前缀和删除左递归是用于将文法LL(1)化的主要变换方法。然而，在极少情况下，可能需要其他的转换。例如，考虑下面的文法片段，它可能出现在允许把标识符作为标号的语言中:

<stmt> → <label> <unlabeled stmt>
<label> → ID :
<label> → λ
<unlabeled stmt> → ID := <expr> ;

在该片段中没有公共前缀，且没有左递归，但该文法不是LL(1)的。问题是ID同时预测两个<label>产生式。解决方法是从第二个和第四个产生式中提取ID，得到下列等价文法，它是LL(1)的:

<stmt> → ID <id suffix>
<id suffix> → : <unlabeled stmt>
<id suffix> → := <expr> ;
<unlabeled stmt> → ID := <expr> ;

在某些情况下甚至需要更复杂的提取公共前缀的方法。例如，在Ada的数组声明中，数组范围可被声明为显式区间对或者是朴素类型（discreet type）或子类型（subtype）的名字。也就是说，可以有A：**array**(I..J, Boolean)。在定义数组范围时可以写

```
<array bound> → <expr> .. <expr>
<array bound> → ID
```

因为ID可以从<expr>中生成，所以这里有一个预测冲突。由于Ada中可能存在的表达式的多样性，从<expr>中提取ID会非常冗长乏味。因此，一个变通的方法是写：

126

```
<array bound>   → <expr> <bound tail>
<bound tail>    → .. <expr>
<bound tail>    → λ
```

如果仅出现单独的<expr>，它必须生成一个ID。这可以在语义处理时检查，因为只有ID能够命名一个类型或子类型。

所有包含结束标记的文法都能被重写成所有产生式右部都以终结符号开始的形式；这种形式称为*Greibach范式*（Greibach normal form）（见练习9）。一旦一个文法是Greibach范式形式，提取公共前缀就很容易。可是，令人惊讶的是，即使这种形式也不保证文法是LL(1)的（见练习10）。事实上，正如下一节所讨论的，确实存在没有LL(1)文法的语言结构。幸运的是，这样的结构在实践中很罕见，而且可以通过对LL(1)分析技术的适度扩展来进行处理。

5.7 LL(1)分析中的If-Then-Else问题

几乎所有普通的程序设计语言结构都能由LL(1)文法描述。然而，一个著名的例外是Algol 60、Pascal和C的**if-then-else**结构。该结构受到被称为“悬空else”（dangling else）的问题的影响，因为**else**子句是可选的。问题是**then**部分可能比**else**部分更多，这意味着**then**和**else**的匹配不是惟一的。实际上，可以认为**If** <expr> **then** <stmt>部分是一个*开括号*而else <stmt>部分是*可选的闭括号*。因此我们不得不分析在结构上等价于

$BL = \{[^i]^j \mid i \geq j > 0\}$

的东西。不幸的是，该语言不是LL(1)的，而且事实上对于任意的k，它也不是LL(k)的。可以通过考虑一些可用来描述BL（Bracket Language，括号语言）的文法来了解该问题。

显而易见的第一次尝试是G_1：

```
S   → [S CL
S   → λ
CL  → ]
CL  → λ
```

这里CL生成可选的闭括号。然而，G_1有一个主要问题——它是二义的。例如，CL CL能够以两种不同的方式生成“]”，这要取决于哪一个CL生成“]”而哪一个生成λ。

127

为消除二义性问题，可以构造遵守Algol 60、Pascal和C规则的文法，它以*最近未匹配的*“[”来匹配每个“]”。这将导致G_2：

```
S   → [ S
S   → S1
S1  → [ S1 ]
S1  → λ
```

G_2生成零个或多个未匹配的“[”，后面跟随零个或多个匹配括号对。事实上，G_2可以通过使用自底向上技术进行分析（例如SLR(1)，这在第6章中讨论）。然而，它不是LL(1)的，对任意k，它也不是LL(k)的。问题是[∈First([S)且[∈First(S1)。类似地，[[∈$First_2$([S)且[[∈$First_2$(S1)，等等。特别地，仅看到开括号，

LL分析器无法确定预测它是配对的还是不配对的开括号。这说明自底向上的语法分析器有一个优点——它们能延迟宣告一个产生式，直到匹配了完整的右部。自顶向下的方法却无法延迟——它们必须仅根据所看到的由右部推导出的第一个（或前k个）符号预测一个产生式。在这种情形下，延迟的能力是至关重要的。

常常用来处理LL(1)分析器中悬空else问题的技术是使用二义文法加上一些特殊情况的规则来解析分析中出现的任意非惟一产生式。该技术将在6.7.3节中详细讨论。

考虑G_3:

G → S;
S → **if** S E
S → Other
E → **else** S
E → λ

G_3是二义的，并导致下面LL(1)分析表:

	if	else	Other	;
S	Predict 2		Predict 3	
E		Predict 4 Predict 5		Predict 5
G	Predict 1		Predict 1	

当然，因为G_3是二义的，所以它的LL(1)分析表也就不是单值的。要强加的辅助规则是**else**和与其最接近的**if**相关联。即，在预测E时，如果看到**else**为超前搜索符号，将会立刻进行匹配。因此，令
128 T[E][**else**] = Predict 4。这项工作可以手工完成，或者，最好通过声明对产生式4的预测优先于产生式5。如5.8节所述，LLGen语法分析器生成器允许通过使用产生式列出的顺序定义它们的优先级来解决二义的预测选择。

最后，再回到悬空else问题。在某种非常现实的意义上，这不是文法或语法分析问题，而是语言设计问题。如果所有**if**语句都以**end if**或某些等价符号结束，就不会出现这种问题。因而，可以使用下列形式的文法:

S → **if** S E
S → Other
E → **else** S **end if**
E → **end if**

该文法无疑是LL(1)的。谨慎的语言设计通常能够扩展可能的语法分析选择的范围。语言设计必须始终记住可由一个结构引起的编译和语法分析问题。

5.8 LLGen—LL(1)语法分析器生成器

LLGen是一个LL(1)语法分析器生成器，它接受CFG规范并产生用来分析指定语言的分析表。LLGen由威斯康辛大学麦迪逊分校的Jon Mauney编写。

LLGen的输入

LLGen的输入主要有三节：运行所需的选项，文法的终结符号，文法的产生式规则。输入的一般形式为

```
注释
* fmq
选项
* define
常量定义
* terminals
```

终结符规范
* Productions
产生式规范
* end
注释

Micro的规范示例在图5-14中给出:

129

```
This is an LL(1) grammar for the
world famous Micro language
*fmq
bnf vocab
statistics noerrortables parsetables
*terminals
ID
INTLIT
:=
,
;
+
-
(
)
begin
end
read
write
*productions
<program>          ::= begin <statement list> end
<statement list>   ::= <statement> <statement tail>
<statement tail>   ::= <statement> <statement tail>
<statement tail>   ::=
<statement>        ::= ID := <expression> ;
<statement>        ::= read ( <id list> ) ;
<statement>        ::= write ( <expr list> ) ;
<id list>          ::= ID <id tail>
<id tail>          ::= , ID <id tail>
<id tail>          ::=
<expr list>        ::= <expression> <expr tail>
<expr tail>        ::= , <expression> <expr tail>
<expr tail>        ::=
<expression>       ::= <primary> <primary tail>
<primary tail>     ::= <add op> <primary> <primary tail>
<primary tail>     ::=
<primary>          ::= ( <expression> )
<primary>          ::= ID
<primary>          ::= INTLIT
<add op>           ::= +
<add op>           ::= -
*end
```

图5-14 Micro的LLGen规范

130

符号

符号由任意可打印字符序列组成；它们由空白、制表符或行结束符分隔。符号中不能包含空白或制表符，除非该符号由尖括号“<”和“>”包围。如果符号以“<”开头，则必须以“>”结尾。

注释

在*fmq之前或*end之后的任何东西都被认为是注释而会被忽略。但是，注释不能包含上述两个保留符号中的任何一个。

选项

跟在*fmq之后是零个或多个选项的列表，以空白、制表符或行结束符分隔。选项控制所生成的表的种类和打印的信息类别。可用选项的完整描述出现在附录C的LLGen用户手册中。

常量定义

常量定义节是可选的。如果存在，它以保留符号*define开始，由一列定义组成，每个定义都占单独一行。每个定义的形式为

<const name> <integer value>

其中，<const name>是上面所描述的符号，而<integer value>是无符号整数（即仅包含数字的符号）。随后，该常量可以被用于任何需要整数常量的地方：在随后的常量定义中以及对于语义例程编号。注意，该特性不像开始看起来那么有益，因为LLGen的输出列表将使用数量值，而不是常量名。

终结符

保留符号*terminals开始一列终结符号。终结符规范节由这样的一列规范组成，每条规范都占单独一行。所有终结符必须出现在该列表中。应当对终结符进行排序，以使所分配的序号与词法分析器中
131 所使用的任意整数代码一致。也就是说，如果**end**的主词法记号代码是4，则**end**应当在终结符列表中第四个出现。

产生式

符号*productions将终结符和产生式分隔开。产生式由一组规则指定，每条规则都占单独一行。产生式规范的形式为：

<lhs> ::= <rhs>

符号“::=”是“→”的同义词，而“→”在大多数字符集中不可用。<lhs>或<rhs>可以不存在。<lhs>是一个表示非终结符的符号。如果它不存在，则使用前面产生式的<lhs>。<rhs>是一个符号串，包含产生式的文法符号，以及指示当到达产生式的适当点时所调用语义例程的动作符号。动作符号由“#”后面跟随一个无符号整数或已定义的常量组成，其间没有空白。如果<rhs>不存在或其中仅包含动作符号，则<lhs>推导出空串λ。<rhs>可以通过使随后的行以保留符号“...”开始来续行（仅有产生式可以这样续行）。

结束符

产生式列表以*end结束。在处理完所有产生式之后，添加拓广产生式。两个符号，<goal>和$$$，以及一个产生式：

<goal> ::= <s> $$$

被添加到文法中，其中，<s>是列表中第一个产生式的左部，<goal>是开始符号，而$$$是结束标记。LLGen将结束标记表示为$$$以允许$和$$在所定义的语言中可以自由使用。

LLGen的输出

分析表格式的详细描述见LLGen用户手册（参见附录C）。LLGen的输出所提供的信息包括

- 尺寸参数（终结符的数量、产生式的数量，等等）。
- 所有产生式的右部。
- LL(1)分析表。
- 能导出空串的符号列表。
- 文法中所有符号的符号化表示。

如果指定的文法不是LL(1)的，则将报告所有的冲突。如果操作resolve处于激活状态，则产生式将根据它们出现的次序被赋予优先级。第一个列出的产生式有最高优先级。因此Pascal以及其他语言中的悬空else可以用以下产生式进行分析：

```
<if stmt>     ::= if <expr> then <stmt> <else part>
<else part>   ::= else <stmt>
              ::=
```

132

冲突将通过优先使用语句的第一种形式（即将**else**与最近出现的**if**匹配）来解决。

应当小心使用该解析机制。冲突必须被仔细地检查以保证所采取的分析动作是所期望的动作。例如，颠倒上面<else part>产生式的次序对于LLGen来说完全可以接受，但是会产生灾难性的后果。当**else**出现在超前搜索符号中时，所采取的分析动作将总是预测<else part>推导出空串；**else**将永远不被接受。

图5-14中所示的Micro的规范说明了LLGen的使用。

5.9 LL(1)分析器的性质

我们可以证明LL(1)分析器的下列一些有用的性质：

- 确保存在正确的最左语法分析。

  这一点根据LL(1)分析器“模拟”最左推导这样一个事实得出。对于共享公共左部的产生式，Predict集总是惟一的。因此，在任意给定点仅有一个可能的预测符合剩余的输入。这就是LL(1)分析器所选择的预测。

- LL(1)类中所有文法都是非二义的。

  如果一个文法是二义的，则某些字符串拥有两个或多个不同的最左分析。这意味着在某些点存在多个可能的正确预测，而对于某些左部，这将导致非惟一的Predict集。

- 所有LL(1)分析器在线性时间内运行，而且最多占用线性大小的空间（相对于被分析的输入的长度）。

  LL(1)分析器的每次迭代都由一个输入符号负责。此外，任意给定的输入符号最多负责常数次迭代。词法记号可能导致一系列预测，随后是一个匹配或错误标志。直接或间接推导出λ的A的预测由导致A被压入栈中的输入符号负责。所有其他的预测由当前输入符号（超前搜索符号）负责。如果当前记号导致不推导出λ的B的预测，则在匹配当前记号（或者发现一个错误）之前B不能再次出现。如果B确实又一次出现，将导致无限循环，而这会违反LL(1)的正确性性质（第1点）。因 133
  此，每一个输入符号导致有限次数的迭代，并遵循线性规律。

  （在分析栈中）使用多于线性的空间将意味要花费多于线性的时间将条目压入栈中。

5.10 LL(k)分析

在LL(1)中使用的单符号超前搜索可以扩展到k个符号。这将产生*强LL(k)*（Strong LL(k)）文法类。根据定义，G是强LL(k)的，当且仅当对产生式$A \to \beta$和$A \to \gamma$ $(\beta \neq \gamma)$，

$$First_k(\beta\ Follow_k(A)) \cap First_k(\gamma\ Follow_k(A)) = \varnothing$$

回忆$First_k$和$Follow_k$是将First和Follow推广到k个符号的一般形式。

强LL(k)使用*全局超前搜索*（global lookahead）（通过Follow集）来做出分析决策。巧合的是，LL(1) = 强LL(1)，但一般而言，对于$k > 1$，强LL(k)是LL(k)的真子集。

直观地说，可以把LL(k) 定义为最强大的自顶向下方法，它使用所有左部上下文（left-context）、要扩展的非终结符以及k个超前搜索符号来做出分析决策。这种方法可以形式化如下：G是LL(k)的，当且仅当三个条件

(1) $S \Rightarrow_{lm}^{*} wA\alpha \Rightarrow_{lm} w\beta\alpha \Rightarrow^{*} wx$

(2) $S \Rightarrow_{lm}^{*} wA\alpha \Rightarrow_{lm} w\gamma\alpha \Rightarrow^{*} wy$

(3) $First_k(x)=First_k(y)$

蕴涵$\beta = \gamma$。

该定义简单地声称G是LL(k)的，当且仅当已知所有左部上下文w、要扩展的符号A，以及下面的k个输入符号时，$First_k(x) = First_k(y)$总是足够惟一地确定下一个预测。

考虑

G → S$
S → aAa
S → bAba
A → b
A → λ

该文法不是LL(1)的，因为b同时预测两个左部为A的产生式。然而，它是LL(2)的。这是因为在上下文aAa中，超前搜索符号ba预测A → b，而a$则预测A → λ。类似地，在上下文bAba中，超前搜索符号bb
134 预测A → b，而超前搜索符号ba则预测A → λ。但对任何k ≥ 1，该文法不是强LL(k)的，因为

$$First_k(ba\$) \in First_k(bFollow_k(A)) \cap First_k(\lambda\ Follow_k(A))$$

问题是使用Follow集的全局超前搜索有时是不精确的。即，如果一个符号b可以在某些上下文中跟随A，则b ∈ Follow(A)。然而，这并不意味着b可以在所有上下文中跟随A。因此，在上述例子中，ba$可以在一个上下文中跟随A，但不能在另一个上下文中跟随A。强LL(k)文法不能处理这种微妙的区别。

解决方法是构造针对LL(k)分析机制的精确超前搜索。为构造LL(k)分析器，创建形式为[A, L]的新的非终结符条目，其中$A \in V_n$且$L \subseteq V_t^{*k}$。V_t^{*k}是不长于k的终结符串的集合。L表示在某些上下文中适于A的精确超前搜索符号集合。

从[S, {λ}]开始。现在如果预测条目[A, L]，要求对于$x \in L$, $A \to \alpha$, $A \to \beta$ $(\alpha \neq \beta)$，必须总是满足$First_k(\alpha x) \cap First_k(\beta x) = \varnothing$。也就是说，使用存储在L中的局部超前搜索符号而不是$First_k(A)$来确定怎样扩展A。假定对[A, L]，如上所述，已经决定使用$A \to \alpha$扩展A，其中$\alpha = x_0\ B_1\ x_1\ B_2 \cdots B_m\ x_m$, $m > 0$且$x_i \in V_t^*$, $B_i \in V_n$, $1 \leq i \leq m$。注意，任意产生式右部都可以被写成这种形式，它分离了产生式右部的非终结符。

然后通过将

$$x_0\ [B_1, L_1]\ x_1\ [B_2, L_2] \cdots [B_m, L_m]\ x_m$$

压入栈中来扩展A，其中对$1 \leq i \leq m$，$L_i = \{ x \mid x \in First_k(x_i\ B_{i+1} \cdots B_m\ x_m\ y), y \in L\}$。自然地，我们计算这些L集合一次，然后构造分析表，并使用[A, L]对，就好像它们是添加到扩展CFG中的新的非终结符。

重新考虑我们的示例文法：

G → S$
S → aAa
S → bAba
A → b
A → λ

并且构造一个LL(2)分析器。从[G, {λ}]开始：

现在[G, {λ}]遇到{aa, ab, bb}中的超前搜索符号时预测[S, {$}]$。

[S, {$}]遇到{aa, ab}中的超前搜索符号时预测a [A, {a$}] a。

[S, {$}]遇到{bb}中的超前搜索符号时预测b [A, {ba}] ba。

[A, {a$}]遇到{ba}中的超前搜索符号时预测b，遇到{a$}中的超前搜索符号时预测λ。

135 类似地，[A, {ba}]遇到{bb}中的超前搜索符号时预测b，遇到{ba}中的超前搜索符号时预测λ。

关键是A被分裂为两个非终结符。这将允许[A, {a$}]遇到超前搜索符号ba时预测b，而[A, {ba}]遇到ba时预测λ。实际上，我们创建了一个等价、但是更大的CFG：

[G,{λ}] → [S,{$}] $
[S,{$}] → a [A,{a$}] a
[S,{$}] → b [A,{ba}] ba
[A,{a$}] → b
[A,{a$}] → λ
[A,{ba}] → b
[A,{ba}] → λ

强LL(k)和LL(k)主要是有理论上的价值，因为只有LL(1)分析器被用于实践。然而，正如所期望的

那样，可以证明有趣的文法包含关系。例如：

- $LL(k) \subset LL(k+1)$
- 强$LL(k) \subset$ 强$LL(k+1)$
- 强$LL(k) \subset LL(k), k > 1$

有趣的是，能够由LL(k)和强LL(k)分析器分析的语言类随着k的增加而增加。下列语言能够由至少使用k个超前搜索符号的LL或强LL分析器进行分析。

$$L_k = \{a^n(b,c,b^kd)^n \mid n \geq 1\}$$

这里的思想是对每个a，必须匹配一个b、一个c，或者一个b^kd串。使用k个符号的超前搜索，可以这样做：接受一个b，如果下面的k个符号是$b^{k-1}d$，则也接受它们以形成b^kd串。使用k-1个符号的超前搜索则不行，因为在第一个b之后看到b^{k-1}而无法判断它是$b^{k-1}d$的一部分还是k个独立的b的一部分。

先前我们曾注意到LL(1) = 强LL(1)，尽管如此，对于$k \geq 2$，$LL(k) \neq$ 强LL(k)。这个结果在练习12中得出。尽管强LL(1)和LL(1)精确地表示相同的文法类，强LL(1)和LL(1)所需的分析表大小并不相同，有时差别很大。如上所见，这是因为LL(1)结构添加了新的非终结符和新的产生式。因此，在实践中使用的LL(1)分析器几乎总是需要较小分析表的强LL(1)分析器。然而，当发现语法错误时，强LL(1)和LL(1)分析器的动作不同。尽管这在分析中没有区别，但在执行语法错误修复时仍然会成为一个问题。在第17章中，讨论在使用强LL(1)分析表时执行LL(1)错误修复的方法。 136

练习

1. 下列文法中哪些是LL(1)的？解释为什么。

a.
$S \rightarrow ABc$
$A \rightarrow a$
$A \rightarrow \lambda$
$B \rightarrow b$
$B \rightarrow \lambda$

b.
$S \rightarrow Ab$
$A \rightarrow a$
$A \rightarrow B$
$A \rightarrow \lambda$
$B \rightarrow b$
$B \rightarrow \lambda$

c.
$S \rightarrow ABBA$
$A \rightarrow a$
$A \rightarrow \lambda$
$B \rightarrow b$
$B \rightarrow \lambda$

d.
$S \rightarrow aSe$
$S \rightarrow B$
$B \rightarrow bBe$
$B \rightarrow C$
$C \rightarrow cCe$
$C \rightarrow d$

2. 为下列文法构造LL(1)分析表：

Expr → - Expr
Expr → (Expr)
Expr → Var ExprTail
ExprTail → - Expr
ExprTail → λ
Var → ID VarTail
VarTail → (Expr)
VarTail → λ

137

3. 跟踪练习2中文法的LL(1)分析器在输入为ID--ID((ID))时的操作。
4. 使用5.6节的技术，将下面的文法变换为LL(1)的形式：

DeclList → DeclList ; Decl
DeclList → Decl
Decl → IdList : Type
IdList → IdList , ID
IdList → ID
Type → ScalarType
Type → **array** (ScalarTypeList) **of** Type
ScalarType → ID

```
ScalarType      → Bound .. Bound
Bound           → Sign INTLIT
Bound           → ID
Sign            → +
Sign            → -
Sign            → λ
ScalarTypeList  → ScalarTypeList , ScalarType
ScalarTypeList  → ScalarType
```

5. 通过LLGen或者任何其他的LL(1)语法分析器生成器来运行练习4的解决方案，以验证它的确是LL(1)的。你怎样知道你的解决方案与原始文法生成相同的语言？
6. 证明每个正则集都能够通过LL(1)文法来定义。
7. 如果有$A \Rightarrow^{+} A$这样的情况，称一个文法拥有环（cycle）。证明有环的文法不可能是LL(1)的。
8. 在5.9节中证明了LL(1)分析器在线性时间内运行。即，当分析输入时，语法分析器处理每个输入符号平均仅需要常数范围的时间。

   是否会出现这样的情况：一个LL(1)分析器需要多于常数范围的时间来接收某些特定符号？换句话说，我们能否限定连续调用词法分析器以得到下一个输入记号的时间间隔为常数？
9. 文法是*Greibach*范式（Greibach Normal Form，GNF），如果所有产生式都形如$A \rightarrow a\,\alpha$，其中a是终结符而α是任意符号串。令G为不生成λ的任意文法。给出一个算法，将G转换为GNF。
10. 如果使用练习9中开发的算法将一个文法转换为GNF，我们知道文法中不会有左递归。转换后的文法仍然可以含有公共前缀，因此可能不是LL(1)的。假定使用5.6节（见图5-12）的`factor()`函数来提取公共前缀。产生的文法将既没有左递归又没有公共前缀，并且将在形式上“接近”LL(1)。证明：在非二义文法中不含公共前缀和左递归不保证该文法是LL(1)的。

138
11. 使用5.10节的技术为下列文法构造LL(2)分析器：
12. 证明每个LL(1)文法也是强LL(1)的。

```
Stmt    → ID ;
Stmt    → ID ( IdList ) ;
Stmt    → ID : Stmt
IdList  → ID
IdList  → ID , IdList
```

提示：证明任意不满足强LL(1)定义的的文法也一定不满足LL(1)定义。

13. 证明对于每个LL(k)文法，存在生成相同语言的强LL(k)文法。

提示：考虑通过构造形式为[A, L]的非终结符对文法进行扩展。

139
14. 使用5.3节的技术，编写程序，读入由LLGen生成的分析表并产生相应的递归下降分析过程。

第6章 LR 分 析

6.1 移进-归约分析器

自顶向下的语法分析器所关心的基本问题是决定使用哪个产生式来扩展特定的非终结符。类似地，自底向上的语法分析器所关心的基本问题是决定何时将看似产生式右部的符号替换为它的左部。这决不是无足轻重的。多个产生式可能拥有相同的右部。而且，可能看起来像产生式右部的符号实际上并不是。如果文法包含λ产生式，则λ可以在任意分析上下文中被匹配的事实使得识别产生式右部变得复杂化。

一个*移进-归约分析器*（shift-reduce parser）按如下方式工作。一个分析栈，初始为空，包含已经分析过的符号。分析栈和剩余输入连接起来总表示一个右句型。词法记号被移进分析栈中，直到栈顶包含句型的句柄。回忆：句柄是匹配某些产生式右部的符号序列，它可被正确地替换为产生式左部。句柄的归约通过在分析栈中将它替换为分析树中作为其父结点的非终结符进行。当所有输入都已经被消耗且栈中仅包含目标符号时报告分析成功。

问题是要知道何时已经到达句柄的末尾，然后确定句柄的长度，最后要知道当存在两个拥有相同右部的产生式时，把句柄归约成哪个非终结符。

考虑一个非常简单的移进-归约分析器驱动程序，如图6-1所示。该驱动程序利用了一个包含分析状态（parse state）的分析栈（parse stack），分析状态通常被编码为整数。分析状态表示分析的当前状态。非正式地说，分析状态对已经移进的符号和当前正在匹配的句柄进行编码。驱动程序使用两张表，action和go_to。action告诉分析器是移进、归约、成功终止，还是通知一个语法错误。go_to表定义在一个词法符号或产生式左部被匹配并移进时的后继状态。

```
void shift_reduce_driver(void)
{
   /*
    * Push the Start State, S0,
    * onto an empty parse stack.
    */
   push(S0);

   while (TRUE) {    /* forever */
      /*
       * Let S be the top parse stack state;
       * let T be the current input token.
       */

      switch (action[S][T]) {
      case ERROR:
         announce_syntax_error();
         break;

      case ACCEPT:
         /* The input has been correctly parsed. */
         clean_up_and_finish();
         return;

      case SHIFT:
         push(go_to[S][T]);
         scanner(& T);    /* Get next token. */
```

图6-1 简单的移进-归约驱动程序

```
                    break;
                case Reduce_i:
                    /*
                     * Assume i-th production is X → Y_1 · · · Y_m.
                     * Remove states corresponding to
                     * the RHS of the production.
                     */
                    pop(m);
                    /* S' is the new stack top. */
                    push(go_to[S'][X]);
                    break;
                }
            }
        }
```

图6-1 （续）

各种移进-归约分析器从上下文无关文法中计算出分析状态、action表以及go_to表的方法各不相同。后面会描述解决该问题的许多方法，其复杂性和使用范围各不相同。

下列文法G_0生成Pascal式语言的块结构：

1. <program> → **begin** <stmts> **end** $
2. <stmts> → SimpleStmt ; <stmts>
3. <stmts> → **begin** <stmts> **end** ; <stmts>
4. <stmts> → λ

140 ~ 141

出现在图6-2和图6-3中的action表和go_to表对应于G_0。随后将详细介绍它们的结构；目前，假定它们由适当的移进-归约分析器生成器创建。在action表中，S代表移进，A代表接受，整数代表归约，而空白是错误条目。在go_to表中，条目是状态编号。开始符号<program>所在的行，在两张表中都是空的。这是因为一旦到达开始符号，分析随即终止。作为优化，可以删除这些行。

142

符号	状态											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	S	S			S		S			S		
end		R4	S		R4		R4	S		R4	R2	R3
;						S			S			
SimpleStmt		S			S		S			S		
$				A								
<program>												
<stmts>		S			S		S			S		

图6-2 G_0的移进-归约action表

符号	状态											
	0	1	2	3	4	5	6	7	8	9	10	11
begin	1	4			4		4			4		
end			3					8				
;						6			9			
SimpleStmt		5			5		5			6		
$												
<program>												
<stmts>		2			7		10			11		

图6-3 G_0的移进-归约go_to表

现在可以跟踪移进-归约分析器在输入为**begin** SimpleStmt; SimpleStmt; **end** $时的分析步骤，如图6-4所示。产生下列归约序列：产生式4，产生式2，产生式2。该序列是最右分析；即，它是相应最右推导的逆序。

143

步骤	分析栈	剩余输入	分析动作
(1)	0	**begin** SimpleStmt ; SimpleStmt ; **end $**	Shift
(2)	0,1	SimpleStmt ; SimpleStmt ; **end $**	Shift
(3)	0,1,5	; SimpleStmt ; **end $**	Shift
(4)	0,1,5,6	SimpleStmt ; **end $**	Shift
(5)	0,1,5,6,5	; **end $**	Shift
(6)	0,1,5,6,5,6	**end $**	Reduce 4
(7)	0,1,5,6,5,6,10	**end $**	Reduce 2
(8)	0,1,5,6,10	**end $**	Reduce 2
(9)	0,1,2	**end $**	Shift
(10)	0,1,2,3	**$**	Accept

图6-4 移进-归约分析示例

6.2 LR分析器

LR分析的概念由Knuth(1965)提出。像所有的语法分析技术一样，LR分析器由超前搜索符号的个数来刻画。LR分析器检查这些超前搜索符号以决定分析动作。我们可以明确表示超前搜索参数并讨论LR(k)分析器，其中k是超前搜索符号的个数。

理论上，关注LR(k)分析器，是因为它是最多使用k个超前搜索符号的最强大的确定性自底向上的分析器类。确定性语法分析器必须在每一步惟一确定正确的分析动作；它们不能回退或重试分析动作。该特性意味着如果一个文法G能够由任何使用k个超前搜索符号的确定性自底向上分析器分析，则能够为G构造一个LR(k)分析器；反之，则并不一定，这并不奇怪——某些可由LR(k)技术分析的文法超出了其他自底向上技术的能力。

本章的大部分内容将致力于众多LR(k)变种的分析器构造问题。在我们深入实现细节之前，根据LR(k)所必有的特性来形式化LR(k)的定义将是有效。

所有移进-归约分析器都移进符号并检查超前搜索符号，直到找到句柄的结尾。随后句柄被归约为在栈中替换它的非终结符。为了移进-归约分析器能够正确运行，分析器必须在仅知道已经移进的符号以及下面的k个超前搜索符号时，决定是移进还是归约。

144

假定在某些LR(k)文法中存在两个句型$\alpha\beta w$和$\alpha\beta y$，它们是如此相似，以至它们共享一个公共前缀$\alpha\beta$以及一个共同的k个超前搜索符号$First_k(y) = First_k(w)$。假定$\alpha\beta w$可被归约为αAw，而$\alpha\beta y$可被归约为γBx。因为这两个句型依据已经移进的前缀和超前搜索符号串来说是完全相同的，所以同样的归约必须能够同时应用于两者。也就是说，可以把$\alpha\beta y$归约为γBx或者αAy，而且必须得到相同的结果，这意味着$\alpha Ay = \gamma Bx$。

按照定义，LR(k)分析器在已知直到句柄结尾处的所有左部上下文以及k个超前搜索符号时，总能够确定正确的归约。用正式的术语，一个文法G是LR(k)的，当且仅当三个条件：

(1) $S \Rightarrow_{rm}^{*} \alpha Aw \Rightarrow_{rm} \alpha\beta w$, and
(2) $S \Rightarrow_{rm}^{*} \gamma Bx \Rightarrow_{rm} \alpha\beta y$, and
(3) $First_k(w) = First_k(y)$

蕴含$\alpha Ay = \gamma Bx$。

该定义是有益的，因为它定义了可由LR(k)技术分析的文法所必须拥有的最少特性。它没有告诉我们怎样实际构造一个LR(k)分析器，而事实上Knuth的开创性工作的主要贡献是LR(k)的一个构造算法。我们开始于可能是最简单的LR分析器：LR(0)，它不使用任何超前搜索符号。尽管LR(0)分析器过于简单而无法用于分析真正的程序设计语言，但它们的确说明了一般LR(k)分析的许多原则。讨论过LR(0)分析器之后，我们将把讨论推广到LR(1)分析器和它们的变体。

6.2.1 LR(0)分析

LR(0)和所有其他的LR式分析方法都基于如下形式的项目（configuration或item）的概念：

$A \rightarrow X_1 \cdots X_i \bullet X_{i+1} \cdots X_j$

更精确地说，具有该形式的项目是LR(0)项目（LR(0) configuration），因为其中不含超前搜索信息。

点符号“•”可以出现在产生式右部的任何地方。它标记相应产生式已被匹配的多少。一般而言，将考虑许多可能应用的产生式，因此将使用项目集（configuration set）。项目集包含在分析的给定点所
145 应用的所有项目。

例如，项目集

```
<stmt> → ID • := <expr>
<stmt> → ID • : <stmt>
<stmt> → ID •
```

表示一个标识符可以和三个不同产生式中的任意一个的一部分匹配的情形。因为不知道哪个产生式可用，所以需要维护表示所有三种可能的项目。

我们从项目集{S → • α$}开始分析，它预测拓广产生式。回忆在第2章中假定所有文法都进行了拓广，因此它们有惟一以开始符号S作为左部的产生式。该产生式总是生成结束符号$，作为推导中的最后一个符号。

一般而言，我们称点号在产生式右部最左端的项目预测（predict）该产生式。类似地，我们称点号在产生式右部最右端的项目识别（recognize）该产生式。

在S → α$中，α可由非终结符开始，在此情况下将必须添加更多的预测和项目。这由LR(0)闭包（closure）运算完成，如图6-5中所定义。

```
configuration_set closure0(configuration_set s)
{
    configuration_set s' = s;

    do {
        if (B →δ • Aρ ∈ s' for A∈V_n) {
            /*
             * Predict productions with A
             * as the left-hand side.
             */
            Add all configurations of the form
                A → • γ to s'
        }
    } while (more new configurations can be added)

    return s';
}
```

146 图6-5 求LR(0)项目集闭包的算法

作为示例，考虑G_1：

```
S → E$
E → E + T | T
T → ID | (E)

closure0({S → • E$}) = {   S → • E$,
                           E → • E + T,
                           E → • T,
                           T → • ID,
                           T → • (E) }
```

该闭包集说明了一个事实：为匹配非终结符E，必须匹配以E作为左部的一个产生式。因为可能包括E → T，所以可能需要匹配以T作为左部的产生式。闭包运算确保包含为匹配所有合法推导序列所必需的项目。

为构造初始项目集s_0，预测拓广产生式并求其闭包：

s_0 = closure0({S → •α$})

给定项目集s，可以计算面临符号X时它的后继（successor）s'，以go_to0(s, X) = s'表示，如图6-6所示。 147

```
configuration_set go_to0(configuration_set s, symbol X)
{
   s_b = ∅;
   for (each configuration c ∈ s)
      if (c is of the form A →β • X γ)
         Add A →βX • γ to s_b;

   /*
    * That is, we advance the • past the symbol X,
    * if possible.  Configurations not having a
    * dot preceding an X are not included in s_b.
    */

   /* Add new predictions to s_b via closure0. */
   return closure0(s_b);
}
```

图6-6 计算LR(0) go_to函数的算法

有可能出现这种情况：go_to0(s, X) = ∅，空集。这意味着面临X时s中的项目没有后继，因此结果是空项目集。在分析时遇到空项目集，则表示出现了语法错误。

一个上下文无关文法中的产生式数量是有限的，因此不同项目和项目集的数量也是有限的。所以，可以通过将项目集和后继运算分别视为状态和转换来构造称为特征化有限状态机（Characteristic Finite State Machine, CFSM）的有限自动机。构造CFSM的算法在图6-7中给出。

```
void build_CFSM(void)
{
   Create the Start State of the CFSM; Label it with s_0
   Create an Error State in the CFSM; Label it with ∅

   S = SET_OF( s_0 );

   while(S is nonempty) {
      Remove a configuration set s from S;
      /* Consider both terminals and nonterminals */
      for (X in Symbols) {
         if (go_to0(s,X) does not label a CFSM state) {
            Create a new CFSM state and label it
               with go_to0(s,X);
            Put go_to0(s,X) into S;
         }
         Create a transition under X from the state s
            labels to the state go_to0(s,X) labels;
      }
   }
}
```

图6-7 为文法计算CFSM的算法

例如，给定文法G_2:

S′ → S$
S → ID | λ

build_CFSM()会构造如图6-8所示的CFSM。为清晰起见，我们忽略了相应于空项目集的错误状态以及到它的转换。

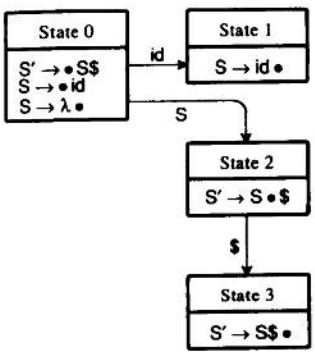


图6-8 G_2的CFSM

因为可能的项目集数量是有限的，且build_CFSM()仅处理每个项目集一次，所以我们可知该算法总会终止。

148

在第3章中我们学习了有限状态机能够用转换表来表示。CFSM也能够以表格的形式表示——以移进-归约分析器中使用的go_to表的形式。图6-9给出由CFSM构造go_to表的算法。

```
int ** build_go_to_table(finite_automaton CFSM)
{
   const int N = num_states(CFSM);
   int **tab;

   Dynamically allocate a table of dimension
      N × num_symbols(CFSM) to represent
      the go_to table and assign it to tab;

   Number the states of CFSM from 0 to N-1,
      with the Start State labeled 0;

   for (S = 0; S <= N - 1; S++) {
      /* Consider both terminals and nonterminals. */
      for (X in Symbols) {
         if (State S has a transition under X
              to some state T)
            tab[S][X] = T;
         else
            tab[S][X] = EMPTY;
      }
   }
   return tab;
}
```

图6-9 构造LR(0) go_to表的算法

对图6-8的CFSM应用build_go_to_table()，得到图6-10所示的go_to表。注意：图6-8中省略的状态4是错误状态。

一般而言，我们在示例中给出CFSM而不是go_to表。实际的LR(0)分析器利用对应于CFSM的go_to表以及一个action表来做出分析决策。

右句型的*活前缀*（viable prefix）是不超出句柄的任意前缀。CFSM从结构上说，接受从所分析的文

法中可推导出的活前缀。

回忆移进-归约分析器，它们包括LR式分析器，是由go_to表和action表来驱动的。前面已经定义了LR(0) go_to表；现在必须确定怎样计算action表。action表的角色很简单——它用于决定CFSM是否到达句柄的末尾。如果已经到达，则指示归约动作或接受动作；否则，应当执行移进动作。

状态	符号		
	ID	$	S
0	1	4	2
1	4	4	4
2	4	3	4
3	4	4	4
4			

图6-10 文法G_2的go_to表

因为LR(0)不使用超前搜索，所以必须从CFSM的项目集中直接提取action函数。令Q = {Shift, Reduce$_1$, Reduce$_2$, …}为可能的移进和归约动作集。定义一个*投影*（projection）函数P，将项目集s映射为表示s中可能的移进和归约动作的Q的子集。 149

令S_0为CFSM的状态集，每个状态由一个特定的状态集标识。则$P : S_0 \rightarrow 2^Q$。2^Q是Q的幂集，即Q的所有子集的集合。P把每个CFSM集映射到Q的适当子集。P(s)被定义为：

{Reduce$_i$ | B→ρ•∈ s and production i is B → ρ} ∪

(If A → α•aβ ∈ s for a∈V_t Then {Shift} Else ∅) 150

G是LR(0)的，当且仅当$\forall s \in S_0$ |P(s)| = 1。（|P(s)|代表集合P(s)的大小。因此|P(s)| = 1意味着P必须是单值的。）

如果G是LR(0)的，则action表可以被简单地从P中提取：

- P(s) = {Shift} ⇒ action[s] = Shift
- P(s) = {Reduce$_j$}，其中产生式j是拓广产生式，⇒ action[s] = Accept
- P(s) = {Reduce$_j$}，i ≠ j ⇒ action[s] = Reduce$_i$
- P(s) = ∅ ⇒ action[s] = Error; action[∅] = Error

对|P(s)|>1的任意状态$s \in S_0$被称为*不足的*（inadequate），有两类语法分析器冲突会在项目集中创建不足的状态：

- *移进-归约冲突*（shift-reduce conflict）。在相同的项目集中同时可能存在移进和归约两种动作。
- *归约-归约冲突*（reduce-reduce conflict）。在相同的项目集中可能存在两个或多个不同的归约动作。

通常，CFSM状态中的不足性通过使用action函数中的超前搜索符号来解决。

由于很容易在CFSM状态中引入不足状态，因此，很少有真正的文法（那些用于生成真正的程序设计语言的文法）是LR(0)的。例如，考虑λ产生式。由于λ产生式的右部为空，它已准备好一旦被预测就进行归约。因此，涉及λ产生式的惟一可能的项目形式为A → λ •（有时写成A → •）。然而，如果A能够生成任意不同于λ的终结符串，则一个移进动作也一定是可能的（为接受First(A)中的符号）。因此移进-归约冲突对于λ产生式是不可避免的，除非该λ产生式的左部符号只能生成λ。

类似地，大多数程序设计语言都拥有运算符优先级别。在缺少显式的括号时，某些运算符比其他运算符优先。例如，在大多数程序设计语言中，A := B+C*D;的含义是A := B+(C*D);而不是A := (B+C)*D;。

使用LR(0)分析器的编译器对于操作符优先级的正确处理可能会有问题。如果编译器在翻译A := B+C+D;，则它应当在移进C之后归约B+C，因为加法是左结合的。然而，如果编译器在翻译A := B+C*D;，则它在移进C后不应当归约B+C，因为乘法比加法优先。没有超前搜索符号（LR(0)不能使用它）时，不容易区别这两种情况。

前面的讨论阐明了很难为“真正的”程序设计语言编写LR(0)文法。不过，一个更基本的问题是，究竟是否有可能为感兴趣的程序设计语言编写LR(0)文法。令人惊讶的是，答案是可以。如本章稍后所述，如果一种程序设计语言有一个结束标记——实际情况总是这样——而且能够由任何一种使用任意数量超前搜索符号的LR式语法分析技术进行分析，则等价的LR(0)文法必定存在。当然，该LR(0)文法可

151 能非常大，非常难以阅读，而且不适于语法制导的翻译，但它必定存在。

作为LR(0)分析器构造的例子，考虑G_1:

S → E$
E → E + T | T
T → ID | (E)

首先，构造CFSM，如图6-11所示。CFSM状态以它们所代表的项目集标记，并且从0开始编号以便
152 在action表中引用。

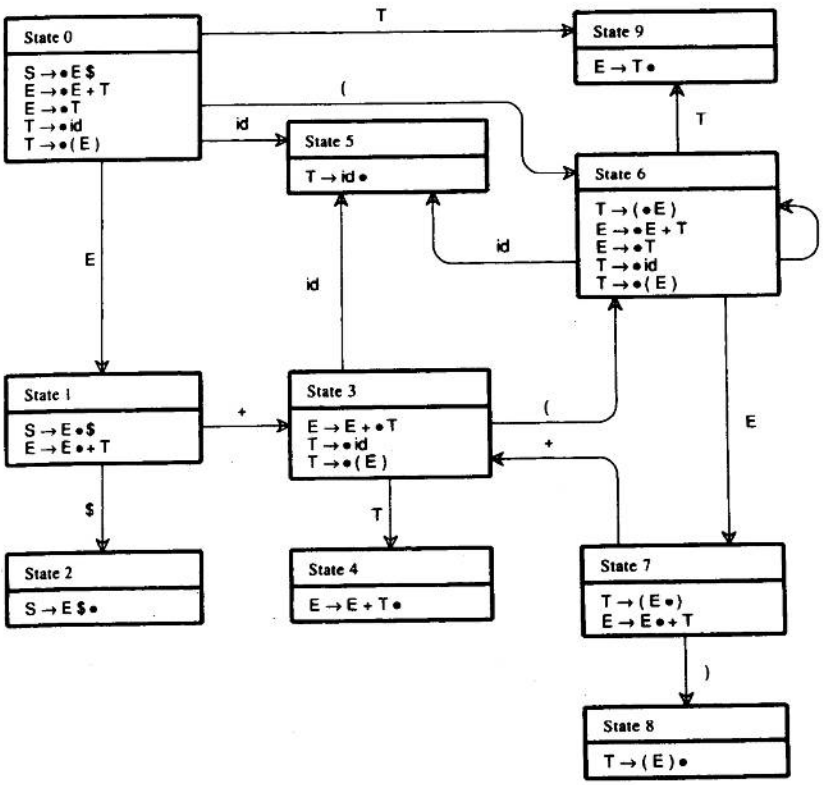


图6-11 G_1的CFSM

如果检查图6-11的CFSM，将看到每个状态要么执行移进（因为•在终结符左边）要么执行惟一的归约（因为•处于一个单独的产生式的右端）。这意味着没有移进-归约和归约-归约冲突；因此，G_1是LR(0)的。go_to表可以使用build_go_to_table()来创建。在go_to表中，在CFSM图中被省略的状态10是错误状态。action表如图6-12所示。S表示Shift；A表示Accept；空白表示Error；Ri表示Reduce$_i$。

状态	0	1	2	3	4	5	6	7	8	9	10
动作	S	S	A	S	R2	R4	S	S	R5	R3	

图6-12 G_1的action表

6.2.2 如何判定LR(0)分析程序工作的正确性

在实际使用LR(0)分析器之前，我们需要确信它们可以正确运行。这通常通过陈述并证明一个“正确性定理”来完成。对于我们的目的而言，重要的是包含在这样一个证明中的洞察力。一旦理解了基本

思想，也就容易明白证明的细节。

LR(0)分析器的正确性依赖于两点观察。首先，CFSM仅能读入的符号串是所分析的文法的活前缀。这一点观察非常重要，因为它告诉我们如果看到非法符号，那么该符号将不会被语法分析器所接受，理由是它不可能是活前缀的一部分。

第二点观察是语法分析器的action表正确地指示了适当的动作。即，action表指示移进直到看到句柄的结尾。随后它指示必要的归约，用相应的产生式的左部替换句柄。

回忆右句型的活前缀是不超出其句柄的任意前缀。句柄是可被正确地归约为非终结符的最左短语。直观地说，LR(0)分析器或任何其他的移进-归约分析器会移进符号以形成一个活前缀，直到整个句柄都被移进为止。随后该句柄被归约为单独的非终结符。LR(0)分析器的CFSM必须能够读入所有活前缀以保证没有漏掉任何合法归约。CFSM不需要读入右句型的所有可能的前缀，因为句柄一被移进并识别，就立即被归约。类似地，如果一个符号序列不可能是任何右句型的前缀，则它无法被分析并且不应被CFSM接受。

假定有活前缀v，怎样从其中创建新的活前缀呢？v的任意前缀都是活前缀，而且如果v仅包含终结符，则这些前缀是可由它创建的仅有的活前缀。如果v至少包含一个非终结符B，则可以把v写成$\alpha B\gamma$。可以使用某个产生式$B \to \beta$来重写B得到$\alpha\beta\gamma$。$\alpha\beta$的任意前缀都是活前缀，由于β是句柄，因此没有活前缀可以扩展超过β。作为一般的规则，我们通过取以非终结符B结尾的活前缀并把B替换为β的任意前缀，来创建新的活前缀，其中$B \to \beta$。 153

为证明活前缀和由CFSM所读入的字符串的对应关系，从CFSM的开始状态s_0开始。s_0可通过读入λ到达，而λ是所有右句型的活前缀。根据其构造，s_0包含一个基本项：$S \to \cdot\ \alpha\$$。s_0面临$\alpha\$$时有后继。这是正确的，正如我们所知$\alpha\$$的所有前缀都是活前缀，因为$\alpha\$$可被归约为S。令βC是$\alpha\$$的以非终结符结尾的任意前缀。在读入β后，将处于CFSM的一个状态s，其中包含形如$D \to \gamma \cdot C\delta$的项目。这种形式的一个项目必定出现在s中，因为我们知道从状态s开始可以读入C。因为C是非终结符，对于左部为C的每条产生式，LR(0)闭包操作将包含形如$C \to \cdot\ \rho$的项目。根据其构造，状态s面临输入ρ时有后继状态，这意味着$\beta\rho$的任意前缀可以从s_0读入。重复这样的论证，容易看出从$\beta\rho$创建的任意活前缀也能够由CFSM读入。因此，以状态s_0开始的CFSM能够读入所有可能的活前缀。

为明白CFSM仅读入活前缀，假定由CFSM所读入的所有长度为n的字符串都是活前缀。这对$n = 0$的情况肯定是正确的。令βX是可由CFSM读入的长度为n+1的字符串。在读入β后，必定处于状态s中，从它开始可以读入X。这意味着s包含形如$B \to \gamma \cdot X\delta$的项目。该项目是基本项或者闭包项。如果它是基本项，γ必定至少一个字符长，否则该项目一定是开始项目$S \to \cdot\ \alpha\$$。已知$\alpha\$$的所有前缀都是活跃的，因此假定$|\gamma| = m \geqslant 1$。βX的最后m个符号必为γ，因为s中包含$B \to \gamma \cdot X\delta$。在读入βX开头的$n - m$个符号ω之后，将处于包含$B \to \cdot\ \gamma X\delta$的状态s'。s'必须也包含B的前面直接为•的项目，因为$B \to \cdot\ \gamma X\delta$仅能通过预测被创建。因此，B能够从s'读入，且因此ωB能够从s_0读入。由于$|\omega B| \leqslant n$，ωB是活前缀，且因此$\omega\gamma X = \beta X$也是活前缀。

如果$B \to \gamma \cdot X\delta$是闭包项，则γ必定为空。$B \to \cdot\ X\delta$通过求某个基本项的闭包被直接或间接地加入。即，s必定包含基础项$D \to \theta \cdot C_1\pi$，其中$C_1 \to C_2\sigma_2$，$C_2 \to C_3\sigma_3$，…，$C_p \to B\sigma_{p+1}$。利用上一段的论证，可以断定βB是活前缀，且βX也是活前缀。

以上已经证明了CFSM精确地接受所分析的文法的活前缀集。为确信LR(0)分析器能够正确工作，余下的一切就是要证明相应于CFSM状态的语法分析器action表总是执行正确的分析动作。

假定正在分析某个有效输入字符串z，并且已经完成了零次或多次正确的归约得到右句型γy，其中y是剩余的输入，而γ已经被移进分析栈中。我们通过从y移进零个或多个终结符号到达下一个句柄。假定$S \Rightarrow^*_{rm} \alpha Aw \Rightarrow_{rm} \alpha\beta w$，其中$\alpha\beta = \gamma x$，且$xw = y$。即，正确的语法分析器动作是读入x，随后执行$A \to \beta$ 154

的归约。可以确信这是LR(0)编译器将要做的吗?

假定在移进γ之后处于状态s。x能够从s读入，因为γx = αβ是活前缀。进一步，语法分析器直到移进所有x之前不执行归约。这个结果是根据这样一个事实得出：在移进x时访问的每个状态必须执行移进动作，因此任意归约动作将导致移进-归约冲突，而这对LR(0)文法来说是不允许的。

因此语法分析器在读入x时不执行假归约。已知αA是活前缀。读入α之后所到达的状态一定包含项目$B \rightarrow v \bullet A\delta$，并因此也包含项目$A \rightarrow \bullet \beta$。该状态面临β的后继必须包含项目$A \rightarrow \beta \bullet$，此状态在移进x之后到达，而且将会执行所期望的归约动作。

以上论述证明，在已经将输入字符串z归约为γy之后，语法分析器将正确地执行下面的步骤，并把αβw归约为αAw。通过对推导z所需步数的归纳，可以确定它将最终被归约为目标符号S，由此成功地完成分析。

最后一点，如果交给LR(0)分析器一个不正确的输入字符串将会怎样？ CFSM仅能够读入活前缀，因此输入若在某些点不能被归约为活前缀，语法错误将会被正确地检测出来。

6.3 LR(1)分析

正如先前所注意到的，因为 LR(0)分析器不使用超前搜索符号，它们不能分析编译器编写者所感兴趣的大多数文法。然而，LR(1)通过在项目中包含超前搜索部分推广了LR(0)。LR(1)项目的形式为

$$A \rightarrow X_1 \cdots X_i \bullet X_{i+1} \cdots X_j, l \quad \text{其中} \quad l \in V_t \cup \{\lambda\}$$

像从前一样，加点的产生式表示已经匹配了多少右部符号。超前搜索部分l表示在整个产生式被匹配后可能的超前搜索符号。超前搜索部分通常是一个终结符号。λ仅在拓广产生式中作为超前搜索符号
155 出现，因为在结束标记之后不存在超前搜索符号。

通常，会有许多LR(1)项目仅有超前搜索部分不同。使用下列记号来表示共享相同加点产生式的LR(1)项目集:

$$A \rightarrow X_1 \cdots X_i \bullet X_{i+1} \cdots X_j, \{l_1 \cdots l_m\}$$

这种形式是含有相同加点产生式和超前搜索符号$l_1, \cdots, l_m$的m个项目的集合的简便表示。

在LR(1)项目中添加超前搜索部分允许我们做出超出LR(0)分析器能力之外的分析决策。然而，需要付出一定代价。现在，比起LR(0)项目来有更多不同的LR(1)项目（与词法记号集的大小$|V_t|$成比例），因此可能有更多的LR(1)项目集。这会大大增加go_to表和action表的大小，而表的大小正比于所创建的项目集的数量。事实上，LR(1)分析器的主要困难不是它们的分析能力（回忆LR(1)分析器是可能存在的最强大的仅使用一个超前搜索符号的确定性自底向上语法分析器），而是找出最经济的存储方式来表示他们。

为创建LR(1)分析器，重复用来创建LR(0)分析器的步骤，但这一次包含超前搜索部分。语法分析开始于项目$S \rightarrow \bullet \alpha\$, \{\lambda\}$；它预测拓广产生式，并将空串作为惟一的超前搜索符号。α可能以非终结符开始，因此需要项目的闭包。LR(1)闭包操作在图6-13中定义。

作为示例，重新考虑G_1:

$$S \rightarrow E\$$$
$$E \rightarrow E + T \mid T$$
$$T \rightarrow ID \mid (E)$$

从$S \rightarrow \bullet E\$, \{\lambda\}$开始，并预测以E作为左部符号并以$作为搜索符的产生式。这将添加
156 $E \rightarrow \bullet E+T, \{\$\}$和$E \rightarrow \bullet T, \{\$\}$。再次预测以E作为左部符号并以+作为搜索号的产生式。这将添加$E \rightarrow \bullet E+T, \{+\}$和$E \rightarrow \bullet T, \{+\}$。现在将预测以T作为左部符号并以$和+作为搜索符的产生式：$T \rightarrow \bullet ID, \{+\$\}$和$T \rightarrow \bullet (E), \{+\$\}$。因此有:

closure1(S → •E$,{λ}) = { S → •E$, {λ};
E → •E + T, {$+};
E → •T, {$+};
T → •ID, {$+};
T → •(E), {$+} }

```
configuration_set closure1(configuration_set s)
{
   configuration_set s' = s;

   do {
      if (B →δ • Aρ, l ∈s' for A∈Vn) {
         /*
          * Predict productions with A as the
          * left-hand side. Possible lookaheads
          * are First(ρl)
          */
         Add all configurations of the form A → • γ, u,
            where  u ∈First(ρl), to s'
      }
   } while (more new configurations can be added)
   return s';
}
```

图6-13 求LR(1)项目集闭包的算法

为创建初始LR(1)项目s_0，预测拓广产生式并求其闭包：

s_0 = closure1({S → •α$}, {λ})

给定一个LR(1)项目集s，使用图6-14中的算法，计算其面临符号X时的后继s'，以go_to1(s, X)表示。

```
configuration_set go_to1(configuration_set s, symbol X)
{
   sb = ∅;
   for (each configuration c ∈ s)
      if (c is of the form A →β • X γ, l)
         Add A →βX • γ, l to sb;

   /*
    * That is, we advance the • past the symbol X,
    * if possible.  Configurations not having a
    * dot preceding an X are not included in sb.
    */
   /* Add new predictions to sb via closure1. */
   return closure1(sb);
}
```

图6-14 计算LR(1) go_to函数的算法

LR(1) action函数仅当存在非空后继状态时指示移进动作。这意味着空项目集是不可达的(unreachable)并且可以被忽略。不同的LR(1)项目和项目集的数量是有限的。可以构造类似于LR(0) CFSM的有限自动机，称其为*LR(1) FSM*或更简单地称为*LR(1)*机器。构造LR(1) FSM的算法在图6-15中给出。

LR(1)机器和CFSM是密切相关的。特别地，一个文法的CFSM能够从LR(1)机器通过“合并”除超前搜索部分外完全相同的项目集来获得。相似地，如果通过添加超前搜索信息对CFSM项目集进行“分裂”，可以创建LR(1)机器。

157

考虑G_3，一个包含+和*的表达式文法：

S → E$
E → E + T | T
T → T * P | P
P → ID | (E)

```
void build_LR1(void)
{
 Create the Start State of the FSM; Label it with s0
 Put s0 into an initially empty set, S.
 while (S is nonempty) {
    Remove a configuration set s from S;
    /* Consider both terminals and nonterminals */
    for (X in Symbols) {
       if (go_to1(s,X) != ∅) {
          if (go_to1(s,X) does not label a FSM state) {
             Create a new FSM state and label it
                with go_to1(s,X);
             Put go_to1(s,X) into S;
          }
          Create a transition under X from the
             state s labels to the state
             go_to1(s,X) labels;
       }
    }
 }
}
```

图6-15 构造LR(1) FSM的算法

给定文法G_3，build_LR1会构造LR(1)机器，如图6-16所示。

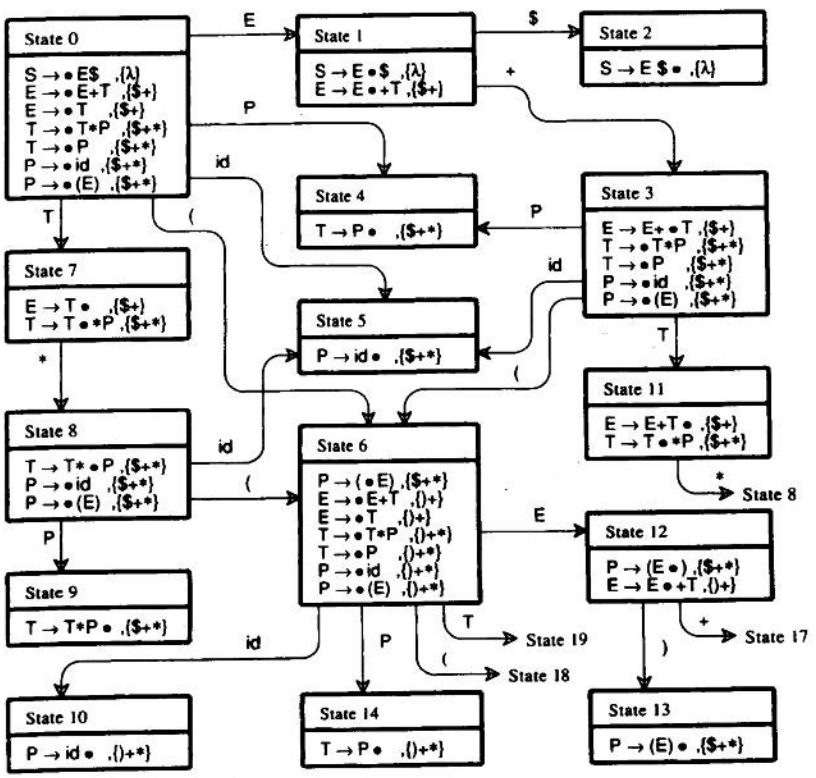


图6-16 G_3的LR(1)机器

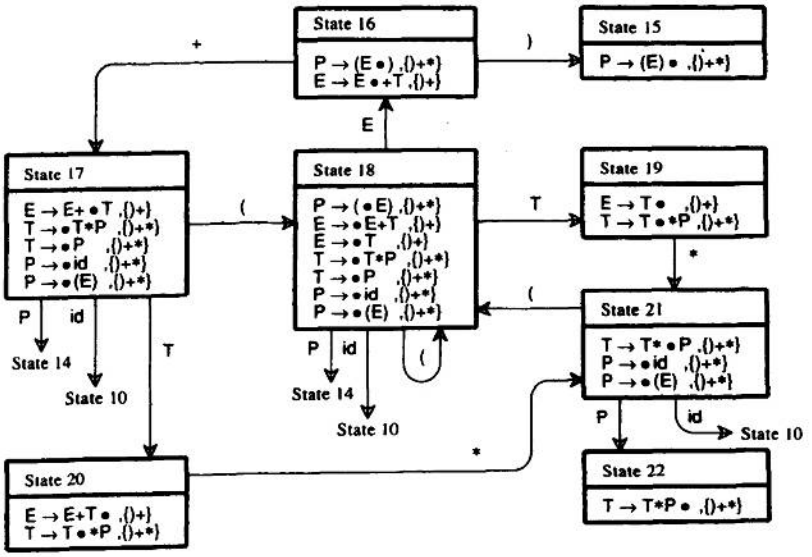


图6-16 （续）

G_3的LR(1)机器有23个状态，而相同文法的CFSM只有13个状态。对于程序设计语言的文法，LR(1)机器的大小通常比相应的CFSM更大。例如，比Ada的文法更小也更简单的Algol 60文法的早期经验表明：需要数千个LR(1)状态。

用于驱动LR(1)分析器的go_to表使用图6-9的build_go_to_table()例程直接从LR(1)机器中提取。由于项目中包含超前搜索信息，action表也可以直接从LR(1)机器的项目集中提取。定义*投影函数*（projection function）P，将项目集和超前搜索符号映射到相应可能的移进或归约动作集。 158

令S_1为LR(1)机器的状态集，每个状态由一个特定的LR(1)项目集标识。则$P : S_1 \times V_t \rightarrow 2^Q$，其中Q是可能的移进或归约动作集。P(s, a)定义为

$$\{\text{Reduce}_i \mid B \rightarrow \rho \bullet, a \in s \text{ and production } i \text{ is } B \rightarrow \rho\} \cup$$

$$(\text{If } A \rightarrow \alpha \bullet a\beta, b \in s \text{ Then } \{\text{Shift}\} \text{ Else } \varnothing)$$

G是LR(1)的当且仅当$\forall s \in S_1\ \forall a \in V_t\ |P(s, a)| < 1$。即，对于所有状态和搜索符对，P至多只能包含一个非错误动作。

如果G是LR(1)的，则容易从P中提取action函数：

- P(s,$) = {Shift} ⇒ action[s][$] = Accept
- P(s,a) = {Shift}, a≠$ ⇒ action[s][a] = Shift
- P(s,a) = {Reduce$_i$} ⇒ action[s][a] = Reduce$_i$
- P(s,a) = ∅ ⇒ action[s][a] = Error

对图6-16的项目集的检查表明每个可能的超前搜索符号导致惟一的动作。即，搜索符l可能导致移进（如果一个•出现在它的左边），或者导致归约（给定形如A → α •, l的惟一项目），或者导致一个错误（如果它既不能被移进，也不是归约搜索符）。因为没有移进-归约和归约-归约冲突发生，G_3是LR(1)的。go_to表可以使用build_go_to_table()直接从LR(1)机器中提取。G_3的LR(1) action表在图6-17中给出。

6.3.1 LR(1)分析的正确性

为证明LR(1)分析的正确性，我们处理与6.2.2节相同的问题。在6.2.2节中曾证明了LR(0)分析的正

确性。像CFSM一样，LR(1)机器仅读入活前缀。LR(1)机器中的项目包含超前搜索部分，且证明这些超前搜索部分是正确的非常简单。即，状态s包含LR(1)项目$A \rightarrow \alpha \bullet, a$当且仅当存在最右推导$S \Rightarrow^*_{rm} \beta Aaw \Rightarrow_{rm} \beta\alpha aw$，其中在移进βα之后到达状态s。

状态	超前搜索符号					
	+	*	ID	(	)	$
0			S	S		
1	S					A
2						
3			S	S		
4	R5	R5				R5
5	R6	R6				R6
6			S	S		
7	R3	S				R3
8			S	S		
9	R4	R4				R4
10	R6	R6			R6	
11	R2	S				R2
12	S				S	
13	R7	R7				R7
14	R5	R5			R5	
15	R7	R7			R7	
16	S				S	
17			S	S		
18			S	S		
19	R3	S			R3	
20	R2	S			R2	
21			S	S		
22	R4	R4			R4	

图6-17 G_3的LR(1) action函数

该证明非常类似于证明CFSM精确地读入活前缀集。我们再一次从s_0开始并证明如果超前搜索的正确性性质对于某个状态成立（它对于s_0当然成立），则它对于该状态的直接后继也成立。

一旦已经证明LR(1)超前搜索部分是精确的，分析器的正确性自然立即成立。假定我们正在分析某个有效输入串z并且已经执行了零个或多个正确的归约以获得右句型γy，其中y是剩余输入，而且γ已经被移进分析栈中。通过从y中移进零个或多个终结符号到达下一个句柄。假定$S \Rightarrow^*_{rm} \alpha Aw \Rightarrow_{rm} \alpha\beta w$，其中αβ = γx且xw = y。即，分析器的正确动作是读
159 入x，然后执行$A \rightarrow \beta$的归约。

进一步假定在移进γ后处于状态s，已经从s能够移进x，因为γx = αβ是活前缀。进一步，分析器在直到所有的x被移进之前将不执行归约，否则会存在移进-归约冲突。

令First(w) = a。已知状态中的超前搜索符号是正确的，在读入α之后，到达包含项目$B \rightarrow v \bullet A\sigma$, b的状态s'，其中$a \in First(\sigma b)$。在求闭包的过程中，加入一项$A \rightarrow \bullet \beta, a$。s'面临β时的后缀是必须包含项目$A \rightarrow \beta \bullet, a$，且由分析器在移进x之后所到达的该状态将指示所期望的归约动作。

该论证表明在已经把输入串z归约为γy之后，分析器将正确地执行下一个步骤，并把αβw归约为
160 αAw。通过对推导z所需步数的归纳，可以确定它最终会被归约为目标符号S，并成功地结束分析。

6.4 SLR(1)分析

LR(0)分析器产生紧凑的go_to表和action表，但它们缺乏分析用于定义真正的程序设计语言的文法的能力。而LR(1)分析器则是使用一个超前搜索符号的最强大的移进-归约分析器类。并不意外的是，实际上对于所有程序设计语言都存在LR(1)文法。事实上，语言设计者通常需要通过LR(1)文法来规定新的程序设计语言的语法。LR(1)的问题在于LR(1)机器包含了如此多的状态以至于go_to表和action表变得惊人地庞大。

为了应对LR(1)分析表空间使用的低效性，计算机科学家发明了几乎与LR(1)同样强大却只需要小得多的分析表的语法分析技术。可以看到两种通用的方法。一种是从通常小得可以接受的CFSM开始，并在构造出CFSM之后向其中添加超前搜索符号。这种方法中最著名的例子就是将在本节中讨论的SLR(1)技术。

另一种降低LR(1)的空间低效性的方法是合并不必要的LR(1)状态。如果简单地合并与CFSM的状态
161 对应的所有状态，则得到LALR(1)技术，它将在6.5节中讨论。如果对于如何合并状态更谨慎一些，有可能创建与普通的LR(1)分析器同样充分强大的语法分析器，而其空间需求与SLR(1)和LALR(1)相当。

LR(1)优化技术在6.9节中讨论。

SLR(1)代表简单的LR(1)。其思想由DeRemer(1969, 1971)引入。SLR(1)使用了LR(0) CFSM并结合以一个符号的超前搜索。即，超前搜索符号不是直接增加到项目中，而是在构造了LR(0)项目集之后加入。结果是一个与LR(1)几乎同样强大但使用少得多的空间的语法分析器。事实上，直到SLR(1)和LALR(1)被广泛认可（在20世纪70年代早期）之前，LR的概念一直被视为仅有理论上而不是实践上的重要性。其后不久它们就取代了当时使用的基于优先级的技术（见6.12.2节），并且今天仍在被广泛使用。 162

回忆在LR(1)分析器中，项目中的超前搜索部分用于决定何时适合执行归约动作。SLR(1)并不从项目中提取超前搜索符号，而是采用一种更简单的方法代替。如果有LR(0)项目B → ρ •，则对要求执行归约动作的超前搜索符号的最低要求是它和该产生式左部兼容。也就是说，如果超前搜索符号*l*要求产生式B → ρ应当被归约，则*l*必须能够合法地跟随B，因为在归约之后它们将会相邻。使用这个逻辑，如果超前搜索符号在集合Follow(B)中，SLR(1)分析器将为项目B → ρ •执行归约动作。这导致下面的SLR(1) action表定义。SLR(1) go_to表是从CFSM中提取，并且与LR(0) go_to表是完全相同的。

回忆S_0是CFSM状态集。SLR(1)投影函数将CFSM状态和超前搜索符号映射到可能的分析器动作：

$P : S_0 \times V_t \rightarrow 2^Q$ 其中Q是可能的移进和归约动作集

P(s, a)被定义为：

$\{Reduce_i \mid B \rightarrow \rho \bullet \in s, a \in Follow(B), \text{and production } i \text{ is } B \rightarrow \rho\} \cup$

$(\text{If } A \rightarrow \alpha \bullet a\beta \in s \text{ for } a \in V_t \text{ Then } \{Shift\} \text{ Else } \varnothing)$

G是SLR(1)的当且仅当$\forall s \in S_0\ \forall a \in V_t \mid P(s, a) \mid < 1$。即，对于所有状态和超前搜索符号对，P至多只能包含一个非错误动作。

如果G是SLR(1)的，容易从P中提取action函数：

- $P(s,\$) = \{Shift\} \Rightarrow action[s][\$] = Accept$
- $P(s,a) = \{Shift\},\ a \neq \$ \Rightarrow action[s][a] = Shift$
- $P(s,a) = \{Reduce_i\} \Rightarrow action[s][a] = Reduce_i$
- $P(s,a) = \varnothing \Rightarrow action[s][a] = Error$

很明显，SLR(1)是LR(0)的真超集。

作为示例，重新考虑G_3，已知它是LR(1)但不是LR(0)的。它是SLR(1)的吗？首先，利用build_CFSM()构造图6-18所示的CFSM。

状态7和状态11是不足的，因为它们各自包含一个移进-归约冲突。在这两种情况下，如果看到在Follow(E) = {$, +,)}中的超前搜索符号将会归约，而如果看到一个*则会移进。由于超前搜索在这两种情况都解决了移进-归约冲突，所以G_3是SLR(1)的。完整的SLR(1)action表如图6-19所示。注意：即使在那些充足状态中也使用了超前搜索符号。通过利用这些超前搜索符号，我们可以比LR(0)分析器稍早检测到语法错误。然而，在充足状态中超前搜索符号可以被忽略。这样做使我们可以缩减分析表的大小，这将在6.8节中详细讨论。 163

6.4.1 SLR(1)分析的正确性

SLR(1)分析器像LR(0)分析器一样都使用CFSM。事实上，这两种技术间惟一真正的差别是在SLR(1)中的产生式p仅当超前搜索符号在p左部符号的Follow集中时才会被归约。我们只需要证明使用超前搜索符号导致正确的归约。而证明这一点非常容易。

当分析某个合法输入串z时，假定已经进行了零次或多次归约以获得右句型αβw，其中$S \Rightarrow^*_{rm} \alpha Aw \Rightarrow_{rm} \alpha\beta w$，而且进一步First(w) = a。CFSM将移进αβ，到达包含项目A → β •的状态。从推导中，我们知道a ∈ Follow(A)，而因此会发生正确的归约。 164

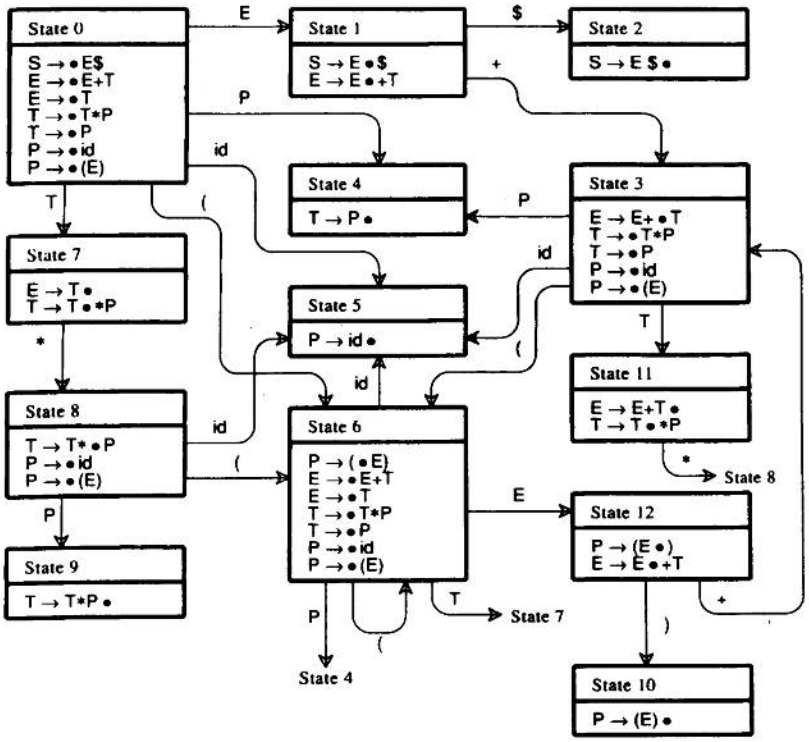


图6-18 G_3的CFSM

状态	超前搜索符号					
	+	*	ID	(	)	$
0			S	S		
1	S					A
2						
3			S	S		
4	R5	R5			R5	R5
5	R6	R6			R6	R6
6			S	S		
7	R3	S			R3	R3
8			S	S		
9	R4	R4			R4	R4
10	R7	R7			R7	R7
11	R2	S			R2	R2
12	S				S	

图6-19 G_3的SLR(1) action函数

6.4.2 SLR(1)技术的局限性

G_3同时是SLR(1)和LR(1)的。实际上，许多LR(1)文法也是SLR(1)的或者能够通过适度的努力被改写为SLR(1)的。结果是，编译器编写者对于SLR(1)分析器有实际的兴趣。

使用Follow集来推测能够预测归约动作的超前搜索符号不如使用合并到LR(1)项目中的精确的超前搜索符号那么准确。容易找出是LR(1)但不是SLR(1)的文法，而且在实践中常常出现这样的文法。例如，

考虑生成两个或更多元素的列表的文法。元素可以是ID，或者括号中的ID，或者列表。该文法不允许单个元素的列表，以避免二义性。

相应的文法G_4是

```
Elem   → (List, Elem)
Elem   → Scalar
List   → List, Elem
List   → Elem
Scalar → ID
Scalar → (Scalar)
```

相应CFSM的一部分如图6-20所示。 165

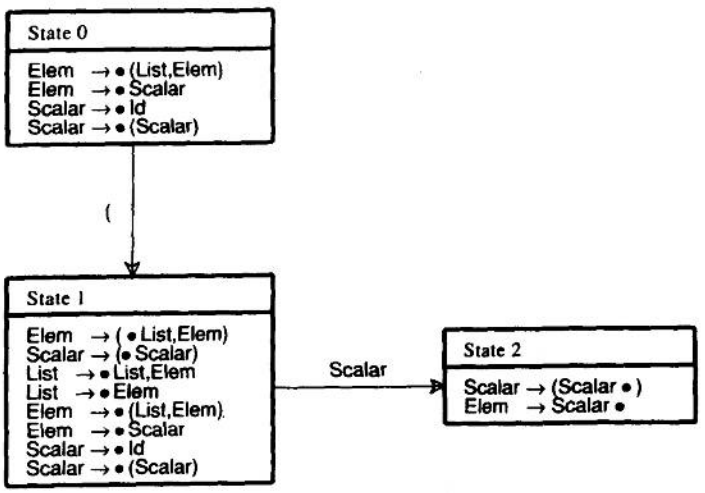


图6-20 G_4的CFSM的一部分

状态2是不足的。由于) ∈ Follow(Elem)，SLR(1)超前搜索符号不能解决二义性；因此，该文法不可能是SLR(1)的。事实上，任意数量的)都能够同时跟随Scalar和Elem，因此可以证明对任意的k，该文法不是SLR(k)的。

当然，可能是该文法仅仅是难以分析，但仔细的研究表明并非如此。注意到如果Elem → Scalar被归约，则下一步必定识别出List → Elem，而且随后从状态1到达一个仅能够读入“,”的状态。因此LR(1)超前搜索符号是“,”，而且该文法片段是LR(1)的。

该例子说明了通过Follow集对超前搜索符号不精确的计算会导致不必要的分析冲突。当发生这样的冲突时，编译器编写者要么重写文法的一部分以符合SLR(1)规则，要么使用一种更强大的分析技术。除了所需分析表的大小，LR(1)文法是显而易见的选择。在随后的一节中，我们讨论LALR(1)分析。LALR(1)拥有SLR(1)的空间效率但能够处理更广泛的文法类（尽管不是所有的LR(1)文法）。结果是，许多编译器编写者使用LALR(1)分析器而不是SLR(1)分析器，而且事实上LALR(1)是最普遍使用的自底向上的语法分析方法。 166

6.5 LALR(1)分析

SLR(1)分析器通过首先构造CFSM，随后通过计算Follow集确定超前搜索符号来创建。相反，LALR(1)分析器可以通过首先构造一个LR(1)分析器并随后合并状态来创建。特别地，LALR(1)分析器就是一个LR(1)分析器，其中仅有项目的超前搜索部分不同的所有状态都已被合并。LALR是超前搜索的

LR（Look Ahead LR）的缩写。这有点用词不当，因为除LR(0)之外所有的移进-归约分析器都使用超前搜索符号。关键是LALR(1)可被视为向底层的CFSM添加了超前搜索符号。LALR分析器在DeRemer(1969)中首先提出。

考虑LR(1)机器中的任何状态s。该状态可以通过简单地删除其中所有项目的超前搜索部分而被惟一地映射到相应CFSM的状态 $\bar{s}$ 。因此如果s是

$A \rightarrow a\bullet, \{b,c\}$
$B \rightarrow a\bullet, \{d\}$

$\bar{s}$ 将会是

$A \rightarrow a\bullet$
$B \rightarrow a\bullet$

通常该映射是多对一的（many to one）。称 $\bar{s}$ 是s的核心（core），并使用记号 $\bar{s}$ = Core(s)。可以通过结合LR(1)机器状态中所有共享相同核心的项目创建同心（cognate）LR(1)项目集。其定义为

$$Cognate(\bar{s}) = \{c | c \in s, Core(s) = \bar{s}\}$$

利用Cognate函数，创建称为LALR(1)机器的有限自动机，它在结构上与CFSM是完全相同的。即，LALR(1)机器和CFSM将拥有恰好相同的项目集和转换。惟一不同的是CFSM状态是LR(0)项目集，而LALR(1)机器状态是LR(1)状态集。因为LR(1)项目包含超前搜索符号，容易将LALR(1)状态投影到可能的分析器动作。LALR(1)投影函数取一个CFSM状态和一个超前搜索符号作为参数。可将CFSM状态转换为它的LALR(1)同心状态，并提取可能的动作。

我们有 $P : S_0 \times V_t \rightarrow 2^Q$，其中Q是可能的移进和归约动作集。P(s, a)被定义为：

$$\{Reduce_i \mid B \rightarrow \rho\bullet, a \in Cognate(s), \text{and production i is } B \rightarrow \rho\} \cup$$
$$(\text{If } A \rightarrow \alpha\bullet a\beta \in s \text{ Then } \{Shift\} \text{ Else } \varnothing)$$

167

G是LALR(1)的当且仅当 $\forall s \in S_0\ \forall a \in V_t | P(s, a) | \leq 1$。

如果G是LALR(1)的，则可以从P中容易地提取action表：

- $P(s,\$) = \{Shift\} \Rightarrow action[s][\$] = Accept$
- $P(s,a) = \{Shift\}, a \neq \$ \Rightarrow action[s][a] = Shift$
- $P(s,a) = \{Reduce_i\} \Rightarrow action[s][a] = Reduce_i$
- $P(s,a) = \varnothing \Rightarrow action[s][a] = Error$

作为示例，让我们回到G_3。考虑LR(1)机器（见图6-16）的每个状态，并合并同心状态。LR(1)状态和它们的LALR(1)同心状态如图6-21所示；在合并同心状态后得到的LALR(1)机器如图6-22所示。

LALR(1) Cognate State	LR(1) States with Common Core
State 0	State 0
State 1	State 1
State 2	State 2
State 3	State 3, State 17
State 4	State 4, State 14
State 5	State 5, State 10
State 6	State 6, State 18
State 7	State 7, State 19
State 8	State 8, State 21
State 9	State 9, State 22
State 10	State 13, State 15
State 11	State 11, State 20
State 12	State 12, State 16

图6-21 G_3的同心状态

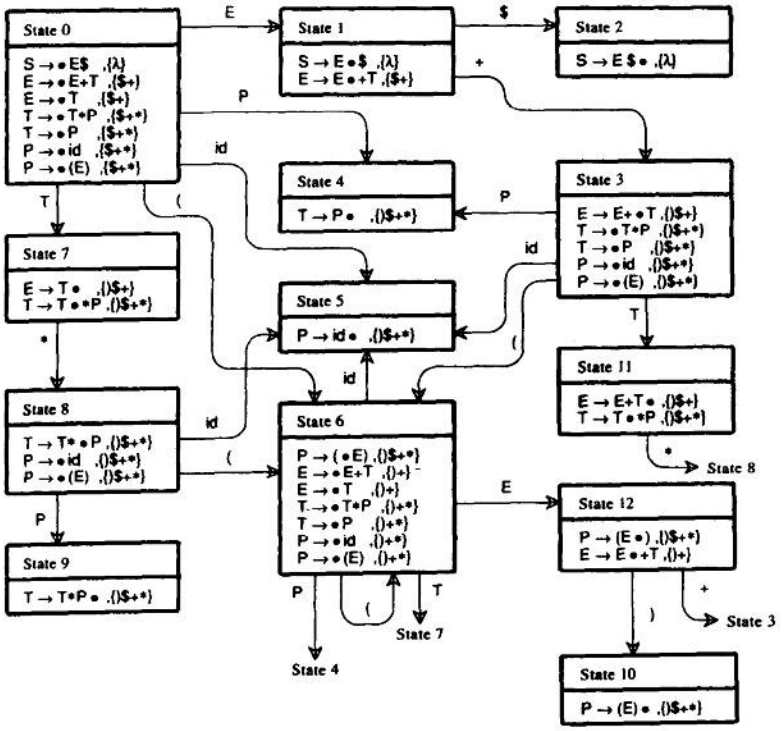


图6-22 G_3的LALR(1)机器

G_3的LALR(1) action表与SLR(1) action表相同。这并不奇怪，因为该文法非常简单。

我们已经看到一个例子（G_4），其中需要更仔细的LALR(1)超前搜索符号计算。需要 LALR(1)的另一种普遍情况由文法G_5说明：

```
<stmt> → ID
<stmt> → <var> := <expr>
<var>  → ID
<var>  → ID [ <expr> ]
<expr> → <var>
```

当利用<prog> → <stmt>预测一条语句时，得到如图6-23所示的CFSM状态。 168

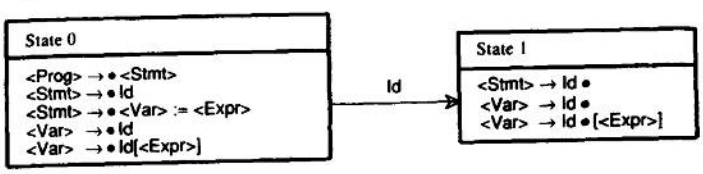


图6-23 G_5的CFSM的一部分

假定按照标准，语句由;分隔，有

; ∈ Follow(<stmt>) 且

; ∈ Follow(<var>)，由于<expr> → <var>

因为从Follow集推导出的超前搜索符号没有解决状态1中的归约-归约冲突，所以G_5不是SLR(1)的。 169

然而，LALR(1)再次满足了要求。如果<var> → ID被归约，则下一步必须移进:=。:=和[都不能跟随

<stmt>，由此解决了冲突。

将LALR(1)文法改写为SLR(1)形式的一种普遍的技术是引入一个新的非终结符，其全局（即，SLR）超前搜索符号更紧密地对应于LALR的精确超前搜索符号。例如，在上述文法中使用该技术，可以将第二个产生式修改为：

<stmt> → <lhs> := <expr>

并随后添加两个新的产生式

<lhs> → ID
<lhs> → ID [<expr>]

Follow(<lhs>) = {:=}。以添加两个新产生式为代价，该文法就被改写成了SLR(1)文法。这些例子显示在实践中构造LALR(1)分析器的额外代价和复杂性是值得的。

SLR(1)和LALR(1)分析器都通过使用CFSM构造。究竟是否会发生这样的情况：无论怎么仔细地计算，无论使用多少超前搜索符号，没有action表能够分析直觉上“容易”分析的文法？答案是肯定的——这有时是由CFSM自身的缺陷所造成。考虑G_6:

S → (Exp1)
S → [Exp1]
S → (Exp2]
S → [Exp2)
Exp1 → ID
Exp2 → ID

G_6表示表达式可以由圆括号或广括号分隔的一种语言。它允许不匹配的分隔符。不同的表达式非终
170 结符用来允许对错误或警告的诊断。相应CFSM的一部分如图6-24所示。

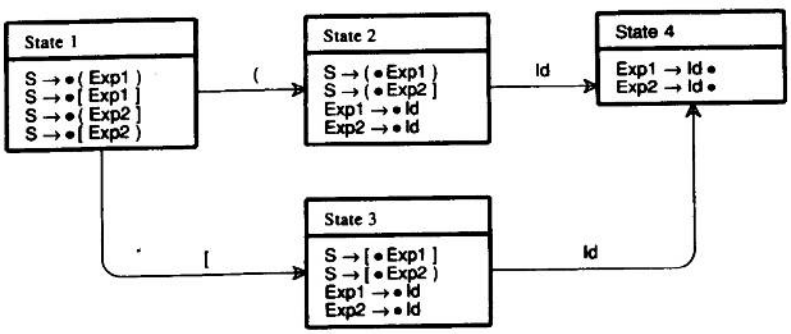


图6-24 文法G_6的CFSM的一部分

状态4是不足的。但没有action函数能够解决它。输入为(ID)时必须识别Exp1 → ID，而输入为[ID)时必须识别Expr2 → ID。然而，这两者（ID和[ID都导致状态4，且在这两种情况下）都是超前搜索符号。额外的超前搜索符号也于事无补——在任意有效输入中根本没有多于一个符号可以跟在ID之后。

如果构造一个完整的LR(1)分析器，状态4“分裂”为两个同心状态，而单个符号的超前搜索足够满足要求。这说明LR(1)能够处理LALR(1)所不能处理的文法。从另一个角度看，LALR(1)对LR(1)状态的合并有时是二义的。我们将在6.9节中讨论其他的LR(1)状态归约技术。

6.5.1 构造LALR(1)分析器

我们的LALR(1)定义暗示LR(1)机器首先被构造，然后它的状态被合并以形成与CFSM在结构上一致的自动机。在实践中这会非常低效，因为我们知道LR(1)机器对于用于定义普通程序设计语言的文法可以有数万个状态。另一种方法是首先构造CFSM。然后LALR(1)超前搜索符号从一个项目到另一个项目

"传播"。特别地，给每个项目一个初始为空的超前搜索符号集（lookahead set）。依据构造，当完成传播 171
时集合中将包含该项目的正确超前搜索符号。

假定每个项目集同时包含基础项目和闭包项目。（严格地说，仅有基础项目是必需的——闭包项目从基础项目中计算出来）随后将那些一个项目的超前搜索符号可以影响另一个项目超前搜索符号的所有项目都链接在一起。称这些链为传播链（propagatie link）。

一个明显需要传播链的情况是当从一个先前的状态经过移进操作创建一个项目的时候。因此，如果有项目$A \rightarrow \alpha \cdot X\gamma$, L_1，其中L_1是超前搜索符号集，则创建一个到项目$A \rightarrow \alpha X \cdot \gamma$, L_2的传播链。这说明L_1中包含的每个符号必须被传送到L_2。

另一种有时需要传播链的情况是当一个项目被作为另一个项目上的闭包操作或预测操作的结果被创建的时候。这里出现两种子情况。假定$A \rightarrow \cdot \alpha$, L_2作为求$B \rightarrow \beta \cdot A\gamma$, L_1闭包的结果被添加。L_2可通过如下计算获得

$$L_2 = \{x \mid x \in \text{First}(\gamma t) \text{ and } t \in L_1\}$$

该集合可以被简写为$\text{First}(\gamma L_1)$。

有时L_2中的符号独立于L_1的值。如果$\text{First}(\gamma)$中不包含λ，就会发生这种情况。这样的超前搜索符号被称为是自发的（spontaneous），因为它们由γ单独确定。它们易于计算，因为不需要知道L_1的值。

在其他时候，$\text{First}(\gamma L_1)$中的符号依赖于L_1的值（当$\gamma \Rightarrow^* \lambda$时）。这样的符号被称为传播的超前搜索符号（propagate lookahead），因为它们必须从L_1中传播出来。链接$B \rightarrow \beta \cdot A\gamma$, L_1和$A \rightarrow \cdot \alpha$, L_2，当且仅当L_2能够从L_1接收传播的超前搜索符号。当且仅当$\gamma \Rightarrow^* \lambda$时，会发生这种情况。我们很容易确定是否$\gamma \Rightarrow^* \lambda$。

在构造了CFSM之后，可以创建所有必要的传播链来将超前搜索符号从一个项目传送到另一个。随后自发的超前搜索符号被确定。在一个项目集中，可以通过将其包含在L_2中来完成。对于项目$A \rightarrow \cdot \alpha$, L_2，其所有自发的超前搜索符号都从形如$B \rightarrow \beta \cdot A\gamma$, L_1的项目导出。它们就是$\text{First}(\gamma)$的非λ值。如果多个项目都能够预测$A \rightarrow \cdot \alpha$, L_2，则L_2可以从每个可能的预测值中接收自发的超前搜索符号。

自发的超前搜索符号用作超前搜索符号计算中的初始值。我们也将初始项目的超前搜索符号集初始化为空集。

随后通过传播链传播超前搜索符号。为此将形如（状态，项目，超前搜索符号）的条目压入栈或队列中。依次考虑每个这样的三元组，而超前搜索符号从指示的项目和状态中传播出来。 172

这个栈初始化为含有超前搜索符号的三元组。随后执行图6-25中所示的算法。

```
while (stack is not empty)
{
   pop top item, assign its components to (s,c,L)

   if (configuration c in state s
         has any propagate links) {
      Try, in turn, to add L to the lookahead set of
        each configuration so linked.
      for (each configuration c̄ in state s̄
           to which L is added)
         Push (s̄,c̄,L) onto the stack.
   }
}
```

图6-25 LALR(1)超前搜索符号传播算法

作为超前搜索符号如何传播的例子，考虑G_7:

S → Opts $
Opts → Opt Opt
Opt → ID

为G_7构造CFSM，建立传播链，并用自发的超前搜索符号来初始化超前搜索符号集。这导致图6-26
173 所示的情形，其中只画出传播链。

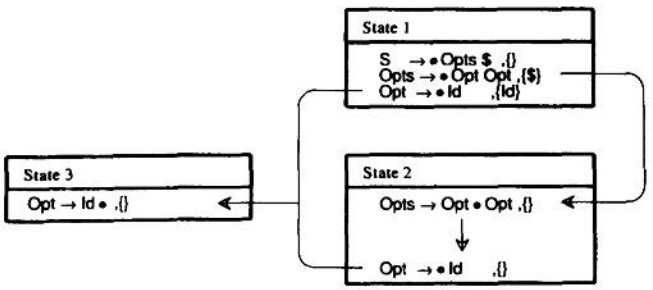


图6-26 含传播链的G_7的CFSM的一部分

因为存在两个自发的超前搜索符号，所以我们将两个三元组压入栈中以开始计算。图6-27给出在图6-26的CFSM上传播超前搜索符号时所执行的步骤。

步骤	栈	动作
(1)	(s1,c2,$), (s1,c3,ID)	Pop (s1,c2,$) Add $ to c1 in s2 Push (s2,c1,$)
(2)	(s2,c1,$), (s1,c3,ID)	Pop (s2,c1,$) Add $ to c2 in s2 Push (s2,c2,$)
(3)	(s2,c2,$), (s1,c3,ID)	Pop (s2,c2,$) Add $ to c1 in s3 Push (s3,c1,$)
(4)	(s3,c1,$), (s1,c3,ID)	Pop (s3,c1,$) Nothing is added (no links)
(5)	(s1,c3,ID)	Pop (s1,c3,ID) Add ID to c1 in s3 Push (s3,c1,ID)
(6)	(s3,c1,ID)	Pop (s3,c1,ID) Nothing is added (no links)
(7)	Empty	Terminate algorithm

图6-27 超前搜索符号传播的示例

最终的CFSM中所有的超前搜索符号都已经被传播，如图6-28所示。

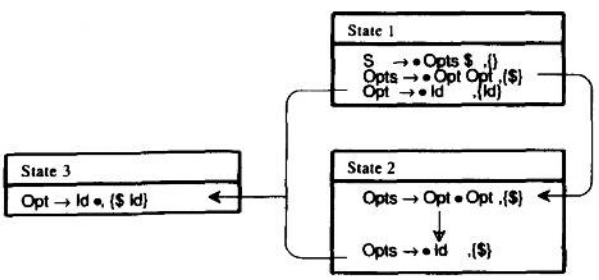


图6-28 超前搜索符号已被传播的G_7的CFSM

为证明超前搜索符号传播算法是正确的，必须证明：

- 它会终止。
- 它恰好传播正确的超前搜索符号集。即，超前搜索符号x被添加到CFSM状态s的项目A → α • β中，当且仅当在LR(1)状态 $\bar{s}$ 存在项目A → α • β, x，且Core($\bar{s}$) = s。

容易证明该算法的终止性。仅当发现新的超前搜索符号时，才压入新的栈条目。状态、项目以及可能的超前搜索符号的数目都是有限的，因此只能出现有限数目的栈条目。

为证明其正确性，首先注意到自发的超前搜索符号必定是正确的，因为它们由加点的项目右部惟一确定，且不受项目的超前搜索符号集影响。类似地，由闭包操作所添加的项目的超前搜索符号由基础项目惟一确定。也就是说，如果基础项目的超前搜索符号是正确的，则所有项目集的超前搜索符号都将是 174
正确的。

假定在某LR(1)状态 $\bar{s}$ 中存在A → α • β, x。超前搜索符号x初始时作为某状态s_1中自发的超前搜索符号创建，且随后通过一个状态序列$s_1, s_2, \ldots, s_n$来传播，其中$n \geq 1$且$s_n = \bar{s}$。在x被作为s_1中自发的超前搜索符号创建时，它也被创建于CFSM状态Core(s_1)中。现在注意到传播链类似于x的原始传输，这次是从Core(s_1)到Core(s_2)并最终到Core(s_n)。这保证存在一条到达CFSM状态Core(s_n)中项目A → α • β的x的传送路径。因此，传播算法包含所有LR(1)状态中的超前搜索符号。

为明白传播算法仅包含LR(1)状态中的超前搜索符号，再次注意初始时所有超前搜索符号都是自发的。这些超前搜索符号一定出现在所有相应的LR(1)状态中。因此开始时，如果(s, c, x)在栈中，则项目(c, x)存在于LR(1)状态 $\bar{s}$ 中，其中Core($\bar{s}$) = s。传播表示LR(1)超前搜索符号被传送的方式。如果跟随一条从(s, c, x)到(s', c')的链，并传播超前搜索符号x，则可以确定状态s'中的一个LR(1)项目(c', x)，其中Core($\bar{s}'$) = s'。因此，在每次迭代中，传播算法传递与一个有效LR(1)项目相对应的一个超前搜索符号。

我们可以期望超前搜索符号传播算法运行得很快，因为它仅传播已知为新的超前搜索符号。然而，它可能需要过量的空间，因为有大量的(状态, 项目, 超前搜索符号)三元组。

另一种变通的方法是使用(状态, 项目)偶对，而不是三元组。向每个项目中添加一个标志以指示处于特定状态的该项目是否处于栈中等候处理。如果一个符号被添加到项目的超前搜索符号中，且其标志为假，则压入一个(状态, 项目)偶对且将该标志设置为真。

当弹出一个(s, c)偶对时，将状态s中c的标志设置为假，并试图传播c的所有超前搜索符号。这节省 175
了空间，因为（在最坏情况下）有较少的偶对需要被压入栈中，但需要更多时间，因为可能需要试图多次传播一个超前搜索符号。

许多LALR(1)分析器生成器使用超前搜索符号传播来计算分析器的action表。在6.7.1节中描述的LALRGen生成器使用在（状态，项目，超前搜索符号）三元组栈上运行的传播算法。Yacc不使用栈(Aho and Ullman 1977, p. 241)，而是依次访问每个项目集，并且所有超前搜索符号都从集合中传播出去。此操作将持续到所有项目集都被访问而没有任何新的超前搜索符号被传播为止。

传播LALR超前搜索符号的另一个颇具吸引力的方法是通过对CFSM的反向搜索按需计算它们。即，不是以正向将超前搜索符号传播到所有状态，而是等到找出一个需要超前搜索符号信息的状态为止。

假定状态s中的项目A → α •需要超前搜索符号。按照定义，超前搜索符号是在执行了正被讨论的归约A → α之后可能被移进的符号。为确定可能的超前搜索符号，可以通过从s回退到预测A → • α的状态并随后检查它们面临A的后继来“模拟”该归约。例如，考虑图6-29。在状态3中，想要指示A → ID归约的超前搜索符号。假如执行了该归约，将会从栈中弹出状态3。状态1或状态4将会是新的栈顶，而且在移进A之后，将处于状态2或状态5。由于在这些状态中仅有)和]可以被移进，因此这两个符号是对于状态3中的归约A → ID仅有的正确的超前搜索符号。

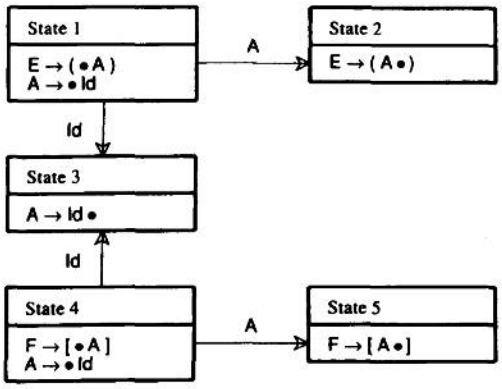


图6-29 使用反向搜索分析的CFSM

把s面临产生式A → α的归约后继（reduction successor）定义为如果在状态s中进行归约A → α可以达到的CFSM状态集，以succ(s, A → α)表示。如果一个终结符号可以从succ(s, A → α)的某个状态被移进，则它指示状态s中A → α的归约。

可能出现这样的情形：在succ(s, A → α)的一个状态s'中，另一个归约也是可能的。例如，s'可能包含项目B → βA •。在这种情况下，在同时出现于succ(s, A → α)和succ(s', B → βA)的状态中检查可被移进的终结符。通常，无论何时当后继集中的某个状态包含可能的归约时，该状态的后继也会被考虑。这表示在移进超前搜索符号前进行一系列归约的情况。

利用后继状态来确定超前搜索符号的思想看起来直截了当，但存在缺陷，尤其是涉及λ产生式时。例如，考虑图6-30所示的CFSM状态s。

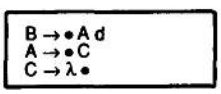


图6-30 一个可能导致反向搜索分析失败的CFSM状态

如果想要C → λ •的超前搜索符号，需要计算succ(s, C → λ)，也就是go_to[s][C] = s'。因为s'包含A → C •，所以考虑succ(s', A → C •)。这将产生问题，因为succ(s', A → C •)可能包含除go_to[s][A]之外的状态。难点是s'可能含有除s之外的前驱，而且这些前驱在计算succ时会被考虑。然而，我们开始于s，因此不应当包含s'的其他前驱。

在使用反向搜索方法的LALR实现中通常不考虑该问题。结果，使用反向搜索的实现通常不能正确
176 地处理所有LALR方法，而被称为*NQLALR*（not quite LALR，不完全LALR）。

正确地实现反向搜索方法的办法在文献（DeRemer and Pennello，1982）中讨论。尽管使反向搜索方法变得正确需要一些技巧，但它确实有要求较少的总工作量的优点，这会导致语法分析器生成器速度的极大提高。

6.5.2 LALR(1)分析的正确性

证明LALR(1)分析的正确性非常直截了当。采取LALR(1)就是LR(1)经过状态合并的版本这样一种观点，这两种技术怎么会不同？在某些情况下合并LR(1)状态形成LALR(1)状态会引入分析器动作冲突。
177 这意味着并非所有的LR(1)方法都是LALR(1)的。如果状态合并不引入冲突，则对于正确的输入，LALR(1)将表现得与LR(1)完全相同，而我们已经知道如何证明LR(1)是正确的。

对于不正确的输入，LALR(1)可能做出错误的归约。这并不是真正的问题，因为底层CFSM仅接受

活前缀。也就是说，用作超前搜索符号的不正确的输入符号，可能导致不适当的归约，但它自身不可能被移进。错误检测的轻微延迟仅在进行错误修复时才会成为问题（不正确的归约不应当被完成）。该困难的解决办法在第17章中讨论。

6.6 在移进-归约分析器中调用语义例程

在LL(1)分析器中，动作符号对应语义例程。当在产生式中遇到动作符号时，将暂停分析，并调用适当的语义例程。在移进-归约分析器中，情况更为复杂。难点是移进-归约分析器不是预测性的，因此直到产生式完整的右部都被匹配之前，不总是能够确定正在识别哪个产生式。然而，对于分析目的来说，这是一个优点，因为它允许移进-归约分析器在做出选择前检查更多的输入。事实上，移进-归约分析器通常能够比LL(1)分析器处理更大的文法类，这也是它们普及的一个主要原因。

如以前所注意到的那样，对于移进-归约分析器所提供的通用性必须付出的代价是仅在产生式被识别并归约之后才能调用语义例程。这相当于仅允许动作符号处在产生式右部的最右端。然而，该限制并不像最初看起来那么严重。事实上，已知两种普通的技巧可以允许语义例程调用位置有更多灵活性。例如，考虑一个生成条件语句的产生式：

```
<stmt> → if <expr> then <stmts> else <stmts> end if
```

需要在条件表达式**else**和**end if**被匹配后调用语义例程。如果使用移进-归约分析器的话，我们无法把动作符号放在必要的位置。然而，为扮演动作符号的角色，可以创建新的生成λ的非终结符。例如，可以创建下列非终结符： 178

```
<stmt>              → if <expr> <test cond>
                        then <stmts> <process then part>
                        else <stmts> end if ;
<test cond>         → λ
<process then part> → λ
```

当识别<test cond>或<process then part>时，将调用适当的语义例程，以模拟动作符号的效果。

用生成λ的非终结符来替换动作符号并不像看起来那么简单。在某些分析上下文中，可能有多个右部需要考虑。如果产生式右部所要调用的语义例程不同，则语法分析器将无法正确确定调用哪个例程。如果生成λ的非终结符被用于调用语义例程，则这种二义性会表现为分析冲突。例如，

```
<stmt>              → if <expr> <test cond1>
                        then <stmts> <process then part>
                        else <stmts> end if ;
<stmt>              → if <expr> <test cond2>
                        then <stmts> <process then part>
                        end if ;
<test cond1>        → λ
<test cond2>        → λ
<process then part> → λ
```

在if <expr>被分析之后，语法分析器无法确定是识别<test cond1>还是<test cond2>。这反映了相互竞争的产生式调用两个冲突的语义例程这一事实。

除使用生成λ的非终结符外，另一种方法是把一个产生式拆分为许多部分，其中拆分点位于需要语义例程的地方。例如，先前的例子可被改写为：

```
<stmt>      → <if head> <then part> <else part>
<if head>   → if <expr>
<then part> → then <stmts>
<else part> → else <stmts> end if ;
```

该方法会使产生式难以阅读，但有不需要λ产生式的优点。（早期的移进-归约分析器无法处理λ产生式，现代技术则能够处理。） 179

6.7 使用语法分析器生成器

在本节中讨论两种流行的LALR(1)分析器生成器LALRGen和Yacc。LALRGen和Yacc的完整总结分别见附录D和Johnson（1975）。我们的讨论举例说明上下文无关产生式、动作符号及相关信息是如何被呈现给分析器的生成器并被其利用。学习分析器生成器使用的最好方法是从这里给出的简单示例开始并随后逐渐推广它们来解决手头上的问题。

6.7.1 LALRGen语法分析器生成器

LALRGen是一个LALR(1)分析器生成器，它接受上下文无关文法规范并产生分析指定语言的分析表。LALRGen由威斯康辛大学麦迪逊分校的Jon Mauney编写。

LALRGen的输入

LALRGen的输入主要有三节：运行所需的选项、文法的终结符号以及文法的产生式规则。输入的一般形式为：

```
注释
*ecp
选项
*define
常量定义
*terminals
终结符号定义
*productions
产生式定义
*end
注释
```

180 Micro的一个示例规范在图6-31中给出。

```
*ecp
bnf vocab
statistics noerrortables parsetables
*define
start          1
finish         2
push_id        3
assign         4
read_id        5
write_expr     6
gen_infix      7
copy_expr      8
push_lit       9
push_op        10
*terminals
ID
INTLITERAL
:=
,
;
+
-
(
)
begin
end
read
write
*productions
```

图6-31 Micro的LALRGen规范

```
<program>          ::= begin <start> <statement list> end ## finish
<start>            ::= ## start
<statement list>   ::= <statement list> <statement>
<statement list>   ::= <statement>
<statement>        ::= ID <push id> := <expression> ; ## assign
<statement>        ::= read ( <id list> ) ;
<statement>        ::= write ( <expr list> ) ;
<push id>          ::= ## push_id
<id list>          ::= ID ## read_id
<id list>          ::= <id list> , ID ## read_id
<expr list>        ::= <expression> ## write_expr
<expr list>        ::= <expr list> , <expression> ## write_expr
<expression>       ::= <expression>  <add op> <primary> ## gen_infix
<expression>       ::= <primary> ## copy_expr
<primary>          ::= ( <expression> ) ## copy_expr
<primary>          ::= ID ## push_id
<primary>          ::= INTLITERAL ## push_lit
<add op>           ::= +  ## push_op
<add op>           ::= -  ## push_op
*end
```

图6-31 （续）

符号

符号由任意可打印字符的序列组成；它们以空白、制表符或行结束符分隔。符号中不能含有空白或制表符，除非该符号以尖括号<和>包围。如果一个符号以<开头，则它必须以>结尾。 181

注释

在*ecp之前或*end之后的任何东西都被认为是注释并将被忽略。然而，注释中不能包含上述两个保留记号。注释也可以被置于任意一行的结尾；在记号--和行结束符之间的所有文本将会被忽略。

选项

跟随在*ecp之后是一列零个或多个选项的列表，以空白、制表符或行结束符分隔。选项控制产生的表的种类以及打印的信息类别。可用选项的完整描述见附录D中的LALRGen用户手册。

常量定义

常量定义节是可选的。如果存在，它以保留记号*define开始，由一列定义组成，每个定义都占单独一行。每个定义的形式为

```
<const name> <integer value>
```

其中<const name>是上面所描述的记号，而<integer value>是无符号整数（仅包含数字的记号）。随后，该整数可以被用于任何需要整数常量的地方：在随后的常量定义中以及对于语义例程编号。

终结符

保留记号*terminals开始一列终结符号。终结符号规范节由这样的一列规范组成，每条规范都占单独一行。所有终结符都必须出现在该列表中。应当对终结符进行排序，以使得所分配的序号与词法分析器所使用的整数代码一致。也就是说，如果end的词法记号代码是4，则end应当在终结符列表中第4个出现。一个好习惯是使用选项vocab列出分配给终结符的整数代码。应当把这些代码与词法分析器产生的代码比较，以确定编号是一致的。

产生式

记号*productions分隔终结符与产生式。产生式由一组规则指定，每条规则占单独一行。产生式规范的形式为

```
<lhs> ::= <rhs> <action symbol>
```

符号“::=”是“→”的同义词，而“→”在大多数字符集中不可用。<lhs>、<rhs>和<action symbol>中任何一个都可以不存在。<lhs>是一个表示非终结符的符号。如果它不存在，则使用前面产 182

生式左部的文法符号。<rhs>是一个包含该产生式文法符号的词法记号串。如果<rhs>不存在，则<lhs>推导出空串λ。<rhs>可以通过使随后的行以保留记号“...”开始来续行（仅有产生式可以这样续行）。<action symbol>的形式为## <number>，它指定当识别产生式时要调用的语义例程。<number>是一个无符号整数或已经定义的常量。如果它不存在，则使用零。如6.6节中所讨论的，不在产生式最右端的动作符号的效果可能通过创建新的非终结符号和包含所要的动作符号的λ产生式来获得。

结束符

产生式列表以*end结束。在所有产生式都被处理之后，添加拓广产生式。两个符号<goal>和$$$，以及一个产生式：

```
<goal> ::= <s> $$$
```

被添加到文法中，其中<s>是列表中第一个产生式的左部，<goal>是开始符号，而$$$是结束标记。拓广产生式被分配以一个代码为-1的动作符号。

LALRGen的输出

分析表格式的详细描述见LALRGen用户手册。LALRGen的输出所提供的信息包括：

- 尺寸参数。包括CFSM状态的数量，文法符号的数量以及产生式的数量。
- 组合的go_to/action表。单个的归约状态已经被删除（见6.8节），因此有三类非错误条目：go_to状态、Reduce动作以及SingleReduce动作。对于拓广产生式，Accept动作被表示为一个SingleReduce动作。
- 每个产生式右部的长度。
- 每个产生式的左部符号。
- 每个产生式动作符号的数量。
- 文法中所有符号的符号化表示。
- 每个CFSM状态的条目符号（即，为到达该状态所移进的符号）。语法分析并不实际需要这些，但它在错误报告和修复中非常有用。

183

6.7.2 Yacc

Yacc是一个LALR(1)分析器生成器，它由AT&T贝尔实验室的S.C.Johnson等人开发。Yacc是“Yet another compiler-compiler”（另一个编译器的编译器）的同义语。严格地说，Yacc并不是一个编译器的编译器，因为它生成可集成的语法分析器，而不是完整的编译器。然而，它确实为语义栈操纵和语义例程规范做了准备，这也就提供了一个典型编译器的大部分结构。Yacc也生成C语言例程文件；因此，它大多用于运行UNIX操作系统的机器上。Yacc可以使用由Lex生成或用C语言手工编写的词法分析器。

Yacc的输入形式为

```
声明
%%
产生式
%%
子例程
```

其中%%分隔规范中的各节。

第一节包含多种声明，其中最重要的是词法记号（即，终结符号）列表。词法记号可以是符号名（以字母打头，可能包含数字、点和下划线）或由引号引用的字符文字常量（例如，‘+’或‘-’）。命名的词法记号必须被声明以区别于非终结符号；由引号引用的文字常量可以可选地声明。声明所有的词法记号是一个好主意，既可用于文档整理目的，又可以辅助与词法分析器的同步。

词法记号声明的形式为

```
%token token1 integer1 token2 integer2 . . .
```

跟在每个词法记号后面的整数定义了由词法分析器所使用的该记号的代码。结束标记必须有一个零值或负值；所有其他词法记号必须有正的代码。

词法记号代码的分配是可选的；没有被分配显式代码的词法记号可以接受它们的字符代码（如果它是一个单引号中的字符），也可以接受一个从257开始的值，这个值对每个未赋值的词法记号是递增的。文件y.tab.h中包含词法代码分配，可以由Yacc可选地创建。一个好的做法是比较由Yacc采用的词法记号分配和由词法分析器产生的那些分配，以确定它们是一致的。这可以通过让词法分析器包含y.tab.h文件并使用其中所包含的定义来自动完成。

其他声明包括C语言的类型、变量和子例程原型，还有开始符号的名字以及运算符优先级和结合性声明（用于二义文法，见6.7.3节）。 184

产生式节定义要被分析的文法。产生式的形式为

```
A : B1 · · · BN ;
```

其中A是产生式的右部符号，而B1…BN是零个或多个终结符号或非终结符号。产生式可以跨越多行；它以一个分号终止。共享相同左部符号的一系列产生式可以写成

```
A   : B1 · · · BN
    | C1 · · · CM
      · · ·
    ;
```

第一个产生式的左部被作为开始符号，除非在声明节出现一个形如%start StartSym的指示命令。在Yacc中，以C代码编写并由{和}界定的语义例程与产生式规则混合在一起。因为底层语法分析器是LALR(1)的，语义例程代码通常出现在产生式的末尾。例如：

```
stmt : STOP ';'     { gen("halt", "", "", ""); } ;
```

Yacc也维护一个语法分析器控制的语义栈并提供方便的形式来访问与产生式相关联的值。符号$$表示产生左部的语义栈值；$1,$2,...表示产生式右部的语义栈值，从左向右编号。语义栈值默认为整型，但Yacc根据需要创建联合（union）类型以保证类型相容性。例如，可以使用下列代码为中缀加法生成代码：

```
Exp : Exp '+' Term
                    { get_temp(n);
                      gen("plus", $1, $3, n);
                      $$ = n; };
```

语义例程代码也可以插入到产生式右部中间，只要知道它不会与其他产生式发生混淆。这些语义例程被视为动作符号并可以访问和更新语义栈。

Yacc规范的最后一节包含语法分析器所需的子程序；在该节中必须提供一个名为yylex()的分析器，也可以把它作为外部子例程提供。如果一个语义动作需要多条语句，最好把它封装为一个子例程，而在产生式一节中仅包含对该子例程的调用。 185

作为示例，简单Micro编译器的Yacc定义示于图6-32中。该定义直接摘自第2章。第2章中的文法是LL(1)形式的；在这时已经被改写为LALR(1)形式。这并非绝对必要，因为实际上所有LL(1)文法也都是LALR(1)的。然而，修改过的文法更好地说明了在自底向上分析中通常所使用的结构，尤其是左递归产生式的频繁使用，而这在自顶向下的语法分析器中是禁止使用的。

6.7.3 可控二义性的使用和误用

所有类型的语法分析器生成器都拒绝二义文法，因为它们会导致不确定的分析决策。然而，研究表

明：如果受控的话，二义性在为真正的程序设计语言产生高效的语法分析器方面是有价值的（Aho, Johnson, and Ullman 1975）。

```
%token BEGIN 10   END  11  ID  1  ASG 3   '+'   6
%token ';'   5    READ 12  '(' 8  ')'  9  '-' 7
%token WRITE 13   ','  4   INTLITERAL 2
%%
program : BEGIN   { start(); }
          statement_list END   { finish(); }
        ;
statement_list : statement_list statement
               | statement
               ;
statement : ID   { $$ = push_id(); }
     ASG expression ';'   { assign($2, $4); }
   | READ '(' id_list ')' ';'
   | WRITE '(' expr_list ')' ';'
   ;
id_list : ID   { read_id(); }
        | id_list ',' ID   { read_id(); }
        ;
expr_list : expression   { write_expr($1); }
          | expr_list ',' expression
            { write_expr($3); }
          ;
expression : expression add_op primary
             { $$ = gen_infix($1, $2, $3); }
           | primary
           ;
primary : '(' expression ')'   { $$ = $2; }
        | ID   { $$ = push_id(); }
        | INTLITERAL   { $$ = push_lit(); }
        ;
add_op : '+'   { $$ = 1; }
       | '-'   { $$ = 2; }
       ;
%%
/*
 * This section has been elided for brevity. In a
 * complete definition it will contain the definition
 * of action and support routines, from Chapter 2.
 */
```

图6-32 Micro的Yacc规范

186

在极少的情况下由于不存在非二义文法而使得二义性对于分析一个程序结构是绝对必需的。该问题最著名的例子是Algol 60和Pascal的悬空else问题。已知LL(1)文法无法确定性地生成**if-then**和**if-then-else**语句。按照语义规则的需要，其中的**else**子句与最近未匹配的**then**子句匹配。

确实存在能够正确处理悬空else问题的LALR(1)文法，但它们不容易产生。显而易见，产生式

Stmt → **if** Expr **then** Stmt
Stmt → **if** Expr **then** Stmt **else** Stmt

会出问题，因为例如在**if** A **then if** B **then** C **else** C中，**else**可以和两个**then**中的任意一个匹配。这其中的窍门是把第二个产生式修改成下列形式

Stmt → **if** Expr **then** RestrictedStmt **else** Stmt

其中RestrictedStmt能够生成除**if-then**结构外的任意语句。

如果能够以某种方式控制其二义性的话，这里给出的简单但有二义性的产生式可以被使用。在LALRGen中，如果选项resolve是激活的，则产生式按其出现的顺序被赋予优先级，第一个指定的产生式被赋予最高优先级。因此，如果把上面**if-then-else**的产生式重新排序为

Stmt → **if** Expr **then** Stmt **else** Stmt
Stmt → **if** Expr **then** Stmt

则移进-归约冲突将通过优先支持第一个产生式，即把**else**与最近出现的**if**匹配而解决。两个含有相同底层产生式的项目间的冲突将通过优先进行归约来解决。这意味着二义文法

E → E + E
E → ID

将正确地分析涉及+和ID并强制左结合的表达式，因为将总是优先选择归约而不是移进。 187

Yacc也允许二义产生式（在警告了用户之后），并提供下列二义分析选择的解决方案：

- 遇到移进-归约冲突时，进行移进。
- 遇到归约-归约冲突时，归约在文法规范中先列出的产生式。

在**if-then**和**if-then-else**结构的简单文法中，第一条规则正是进行适当的匹配所需要的。特别地，我们想要尽快地匹配一个**else**，移进而不是归约可以完成这项任务。

在这种情况下，允许受控的二义性提供了比无二义的LALR(1)分析器所能够接受的更简单的文法。然而，在允许二义性时需要付出代价。作为语法分析器的用户，通常不必关心它如何工作——它所找出的任意分析都是*正确的*分析。但当允许二义性时，必须理解消除二义性的机制如何工作，而这需要关于语法分析器和所使用的语法分析器生成器的知识。我们不再纯粹地通过其所分析的文法来指定语法分析器，而是显式地包含辅助规则来消除文法中的冲突。

受控二义性的使用在规定程序设计语言表达式中有极大的优势。非终结符号和产生式的层次被用于为运算符优先级和结合性编码。该编码极大地增加了文法和分析表的大小。进一步，如6.8节中所讨论的，一系列单位归约会降低分析速度。

Yacc允许从文法中删除操作符优先级和结合性规则并提供显式的指示来代替。表达式可以通过高度二义性的产生式推导出来：

Expr → Expr BinaryOp Expr
Expr → UnaryOp Expr

结合性在声明节中由`%left`、`%right`和`%nonassoc`指定。大多数二元运算符（像+、-、*和/）都是左结合的（即，它们从左向右分组）。少数运算符（如指数运算符）是右结合的，还有少数运算符（典型的如关系运算符）根本不可结合。

运算符被赋予结合性的次序表明了它们的优先级，首先出现的运算符优先级最低，最后出现的运算符优先级最高。在很少的情况下，一个运算符以不同的优先级同时被当作一元和二元运算符。在这样的情况下，将使用`%prec`指示命令强制特定用法运算符的优先级。

为举例说明包括`%prec`在内的各种指示命令的用法，我们对表达式利用可控二义性把先前Micro的例子重写为另一种形式。Micro仅包含两种二元运算符+和-；它们有相同的优先级和结合性，因此我们将定义拥有更丰富运算符集合的扩展Micro，*MicroPlus*。MicroPlus的运算符集合的定义如图6-33所示； 188
相应的Yacc定义如图6-34所示。一个很好的练习是将图6-34的定义重写为纯上下文无关文法的形式而不使用显式的优先级和结合性定义。指定表达式所需的子文法将会非常大而且笨重。

运算符	优先级	结合性
一元 -	最高优先级	
*, /	第二最高优先级	左结合
+, 二元 -	第三最高优先级	左结合
=	最低优先级	无

图6-33 MicroPlus运算符

```
%token  ID      1     INTLITERAL  2     ASG   3     ','     4
%token  ';'     5     '+'         6     '-'   7     '('     8
%token  ')'     9     BEGIN       10    END   11    READ    12
%token  WRITE 13      '*'         14    '/'   15    '='     16
%nonassoc   '='
%left   '+' '-'
%left   '*' '/'
%left unary_minus  /* Placeholder for unary '-'; see below */
%%
program : BEGIN   { start(); }
          statement_list END   { finish(); }
        ;
statement_list : statement_list statement
               | statement
               ;
statement : ID   { $$ = push_id(); }
            ASG expression ';'    { assign($1, $3); }
          | READ '(' id_list ')' ';'
          | WRITE '(' expr_list ')' ';'
          ;
id_list : ID   { read_id(); }
        | id_list ',' ID   { read_id(); }
        ;
expr_list : expression   { write_expr($1); }
          | expr_list ',' expression   { write_expr($3); }
          ;
expression :
    expression '=' expression   { $$ = gen_infix($1, 0, $3); }
  | expression '+' expression   { $$ = gen_infix($1, 1, $3); }
  | expression '-' expression   { $$ = gen_infix($1, 2, $3); }
  | expression '*' expression   { $$ = gen_infix($1, 3, $3); }
  | expression '/' expression   { $$ = gen_infix($1, 4, $3); }
  | '-' expression  %prec  unary_minus   { $$ = negate($2); }
  | '(' expression ')'   { $$ = $2; }
  | ID   { $$ = push_id(); }
  | INTLITERAL   { $$ = push_lit(); }
  ;
%%
/*
 * The same action and support routines as used in
 * the previous Yacc example, with negate() added.
 */
```

图6-34 使用受控二义性的MicroPlus的Yacc规范

分配给运算符的优先级解决了大多数如下形式的移进-归约冲突。

$Expr \rightarrow Expr\ Op_1\ Expr \bullet$
$Expr \rightarrow Expr \bullet Op_2\ Expr$

如果Op_1拥有比Op_2更高的优先级，则归约。如果Op_2的优先级更高，则移进。当Op_1和Op_2拥有相同的优先级时，使用结合性定义。如果这两个运算符都是左结合的，则归约；如果这两个运算符都是右结合的，则移进；如果这两个运算符是非结合的，则产生一个错误。

利用运算符优先级和结合性来指导分析的思想源于6.12.2节中所讨论的运算符优先级分析技术。尽管运算符优先级的能力过于有限而无法处理所有现代程序设计语言的语法，但它非常适合于大多数语言的表达式结构。

189

6.8 优化分析表

在实践中可以进行许多改进来降低LR分析器的空间需求。可以使用单独一张合并的分析表，而不是拥有独立的go_to表和action表。在action表中的移进条目被扩展以包含go_to表中的相应状态条目。非终结符条目也包含在合并的表中，这张表被简单地称为分析表。

回到G_3，我们将编码了CFSM信息的go_to表和SLR(1)或LALR(1) action表组合在一起获得图6-35所示的分析表。

分析表条目通常编码为整数。一种方便的编码方案是用零来表示错误条目，用正整数表示归约动作，用负整数表示移进动作，并用拓广产生式（通常是R1，编码为1）的归约动作表示接受动作。

许多语法分析器状态在给定正确的超前搜索符号时只识别单一的归约。因此图6-35中的状态4总是识别产生式5或产生错误。我们称这些状态为*单一归约状态*（single reduce state）。作为优化，可以消除所有单一归约状态。典型地，每个产生式都有一个相应的单一归约状态，因此该优化能够极大地缩减必须在分析表中表示的状态的数量。 190

状态	符 号								
	+	*	ID	(	)	$	E	T	P
0			S5	S6			S1	S7	S4
1	S3					A			
2									
3			S5	S6				S11	S4
4	R5	R5			R5	R5			
5	R6	R6			R6	R6			
6			S5	S6			S12	S7	S4
7	R3	S8			R3	R3			
8			S5	S6					S9
9	R4	R4			R4	R4			
10	R7	R7			R7	R7			
11	R2	S8			R2	R2			
12	S3				S10				

图6-35 G_3的SLR(1)分析表

引用单一归约状态的分析表条目将被替换为一个特殊标记，在这里使用一个L前缀和在该状态中所识别的产生式来表示。例如，移进到状态4会被替换为条目L5。这里的思想是如果在某个状态只能够做一种可能的归约，就不需要实际到达这个状态，而是立即做归约。我们也修改语法分析器驱动程序，使得识别L类型归约时，从分析栈中少弹出一个状态。这是必要的，因为对于L类型归约条目，我们立即进行归约而不是移进到一个单一归约状态并随后归约。

但如果超前搜索符号不正确该怎么办？因为我们不检查超前搜索符号，所以无论如何都要进行归约。对错误的检测被推迟到试图移进该超前搜索记号时。在这里错误必定被检测到，因为如果一个超前搜索符号不正确的话就无法移进它。除LR(k)（没有相容状态合并）之外的所有LR分析技术都只使用近似超前搜索符号来决定归约动作；因此消除单一归约状态不会引入任何新的复杂性。

在大多数情况下，在看到错误的记号和将其识别为错误这段时间之间的少许延迟没有什么影响。如果正在进行错误修复，可以通过*缓存*分析器活动来预见不正确的超前搜索符号出现的可能性。所有归约都被保存在缓存中，直到超前搜索符号被成功地移进并生效。如果不能移进超前搜索符号，那么在执行错误修复前将用缓冲区恢复先前的归约。这将在第17章更完整地讨论，其中涉及LR分析器的错误修复。 191

在实践中，删除单一归约状态将极大地缩减分析表的大小（基本上，对于每个产生式都会有一个状态被删除）。对于G_3，一个优化的SLR(1)分析表（其中的状态没有被重新编号）如图6-36所示。

在删除单一归约状态后，对剩余的状态重新编号很简单。再一次将分析表条目编码为整数。错误条目用零表示，归约动作用正整数表示，移进动作用负整数表示，而接受动作则用拓广产生式的归约动作来表示。L条目被编码为正整数加上某个大于产生式数的整数。因此L7可以被编码为1000 + 7 = 1007。

单一归约状态的删除是一种简单而又有效的优化。通过注意到某些状态“几乎”是单一归约状态还可以进一步推广这里的思想。也就是说，几乎所有非错误条目都是一个特定的归约动作。图6-36中的状态11说明了这一点。有三个R2动作和一个S8动作。可以通过让R2成为该状态的*默认动作*来删除R2动作。也就是说，创建一个以状态作为索引的辅助向量，其中包含每个状态的默认归约动作。某些没有默认动作的状态，其默认动作就是报告错误。当一个默认归约动作被选择时，它被加入辅助表并从分析表中删

除。这样做的效果是从分析表的一行中删除许多相同动作并只保留该动作的一个复本。分析表将含有更少的非错误条目，因此它也更易于压缩。

状态	符号								
	+	*	ID	(	)	$	E	T	P
0			L6	S6			S1	S7	L5
1	S3					A			
3			L6	S6				S11	L5
6			L6	S6			S12	S7	L5
7	R3	S8			R3	R3			
8			L6	S6					L4
11	R2	S8			R2	R2			
12	S3				L7				

图6-36 G_3的优化SLR(1)分析表

当分析表指示一个错误动作时，将查询辅助表。如果有默认归约，则执行它；否则，识别出一个错误。像单一动作状态一样，这项优化可能延迟错误的识别。然而，移进动作决不会被作为默认动作，因此错误记号不会被不正确地接受。因为LR分析表相当稀疏，所以在第17章中所讨论的压缩技术常常能够极大地减少存储需求。

刚刚讨论的优化是空间优化，设计用于减小LR分析表的大小和密度。然而，速度优化也是可能的，
192 其中受关注最多的是消除单位归约（unit reduction）。单位归约是用把产生式的长度为1的右部用一个非终结符（当然，长度也是1）来替换。常常发生一系列的单位归约，并且有可能把这样的归约链折叠为单一归约。

右部长度为1的单位产生式用于表达式中以强制运算符优先级。例如，在G_3中，有E → T、T → P和P → ID。一般情况下，如果一个程序设计语言有n个运算符优化级，它将有从标识符到表达式的n+1个单位产生式组成的产生式链。在真正的语言中n可以是10或更大，而分析一个由单个标识符或文字常量组成的平凡（而且非常普通的）表达式会需要大量步骤。

如果在ID和表达式中间的单位产生式都不包含动作符号，则它们可以被安全地跳过。这里的思想是检查每个合法的超前搜索符号并确定在消耗该超前搜索符号之前将应用多少个单位产生式。作为示例，重新考虑G_3的LALR(1)机器的状态0，如图6-37所示。

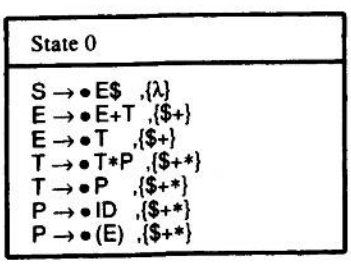


图6-37 文法G_3的一个CFSM状态

如果移进ID，可能的超前搜索符号为$、+和*。如果*是超前搜索符号，则ID将被归约为P，随后P被归约为T。如果+或$是超前搜索符号，则ID将被归约为P，随后P被归约为T，最后T被归约为E。我们通过修改状态s = go_to[0][ID]的分析表条目来优化单位归约链。对于一个超前搜索符号*，识别伪产生式（pseudoproduction）T → ID并执行必要的语义例程，随后将go_to[0][T]压入分析栈中，略过产生式T → P。对于超前搜索符号+或$，识别E → ID。

单位归约优化会提高语法分析器的速度，但可能会以新的分析器状态或分析表条目为代价。研究人员已经研究了不过度增加分析表大小的单位归约优化方法（Soisalon-Soininen 1982，Pager 1977）。这些技术相当复杂，而且并不总是需要它们的完全通用性。

193 一种有效的直观推断是先选择预测表达式的那些状态（例如G_3的CFSM中状态0和状态6，见图6-18），并且对这些状态仅修改其面临开始一个归约链的符号（ID或INTLITERAL）时的后继状态。这里的直觉是最常见的表达式都是由单个标识符或文字常量组成的平凡表达式。因此，折叠从标识符或文字常量到

表达式的单位产生式链非常有益。

我们修改的后继状态通常会被删除，因此会有适度的空间损失。只要超前搜索符号对相同的移进和归约动作达成一致（归约动作可以覆盖错误动作），新创建的状态就可以共享相同的分析表行。因此，在图6-36中G_3的优化SLR(1)分析表中，消除了错误条目的状态0对于ID的后继状态变成

[$: Reduce E → ID; +: Reduce E → ID; *: Reduce T → ID]

类似地，状态6对于ID的后继状态变成

[): Reduce E → ID; +: Reduce E → ID; *: Reduce T → ID]

这里不存在冲突，因此这两行可以进行覆盖。

在某些状态中，一系列单位归约必须独立于超前搜索符号发生。例如，在G_3的CFSM的状态3中，在ID被归约为P之后，P必须总是被归约为T。将L类型归约P → ID替换为伪产生式T → ID，可以在没有空间损失的情况下优化归约链。

6.9 实用的LR(1)分析器

编译器编写者的传统观点是LALR(1)是最强大的“实用的”移进-归约分析器，而且事实上大多数自底自上的语法分析器生成器都是LALR(1)的。完整的LR(1)分析器常被忽视，因为它们通常包含过多的状态而无法用于真正的程序设计语言。

然而，该观点并不完全合理。如我们所看到的，LALR(1)可以被视为对LR(1)分析器的优化，其中合并了所有同心状态。LALR(1)是一种要么全有、要么全无的方法；我们合并所有可能的相应LR(1)状 194
态并随后或者接受或者拒绝这个“优化的”语法分析器。

更恰当的方法是当可能时合并状态，但在会引入分析冲突时不合并状态。这保证了全部的LR(1)文法类都会被接受，而且仍然会极大地节约空间。如果一个文法是LALR(1)的，则所试图进行的所有合并必定成功，而且获得传统的LALR(1)机器。如果一个文法“几乎是LALR(1)的”，则得到仅比相应CFSM多出少量状态的语法分析器。

在可能时合并状态的思想是很吸引人的，但它并不像看起来那么简单。主要困难是一对状态可能看起来可以合并而事实上并非如此。当一对状态被合并时，面临同样符号的后继状态必须也被合并，而分析冲突完全有可能仅在执行后继状态合并时出现。考虑图6-38所示的LR(1)状态对。合并这两个状态是错误的，因为它们的后继在合并时会产生不可解决的归约-归约冲突。

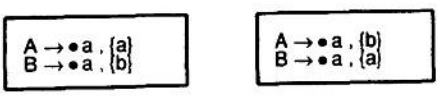


图6-38 要被合并的LR(1)状态对

另一个困难是状态合并的顺序可能造成最终“优化的”语法分析器大小上的差异。这意味着不能在所有情况下保证得到最少状态分析器，除非尝试所有合并状态的方法。仅考虑一种合并次序的实际效果是，所得到的最少的状态相对于最优情况来说差别可能非常地小。

假定有两个LR(1)状态s_1和s_2同心。需要某种方法确定合并这两个状态是否会导致分析冲突。一种可能的办法是分两步来进行。首先，确定$s_1 \cup s_2$是否有任何不能通过超前搜索符号来解决的冲突。这很容易做到。如果立即合并是安全的，则研究合并其后继状态的结果——例如，go_to[s_1][X]和go_to[s_2][X]。如果对任何一对后继状态的合并会导致不可解决的冲突，则回溯并拒绝合并s_1和s_2。

这种回溯方法的一个真正问题是它假定整个LR(1)机器已经被构造出来，因此能够根据需要对状态和它们的后继进行合并。更好的方法是在创建状态时对它们进行合并，以极大地减少需要操纵的中间状

态的数量。该方法已经由Pager（1977）进行了研究。他没有去合并状态并随后研究那些甚至还没有被创建的后继状态的合并效果，Pager只是定义了保证合并安全的标准。

最简单的标准被称为“弱相容性”（weak compatibility）。考虑两个LR(1)状态s和 $\overline{s}$，它们同心并因此可能被合并。考虑s中的两个项目$A \rightarrow \alpha \bullet \beta, L_1$和$B \rightarrow \delta \bullet \gamma, L_2$，其中$L_1$和$L_2$表示可以应用于相应加点产生式的超前搜索符号集。因为 $\overline{s}$ 和s同心，所以 $\overline{s}$ 必定包含形如$A \rightarrow \alpha \bullet \beta$、$\overline{L_1}$和$B \rightarrow \delta \bullet \gamma$、$\overline{L_2}$的项目。我们称s和 $\overline{s}$ 是弱相容的，当且仅当对于所有项目对满足下列三个条件之一：

195

(1) $L_1 \cap \overline{L_2} = \emptyset$且 $\overline{L_1} \cap L_2 = \emptyset$

(2) $L_1 \cap L_2 \neq \emptyset$

(3) $\overline{L_1} \cap \overline{L_2} \neq \emptyset$

不难证明两个状态是弱相容的，当且仅当它们的基础集是弱相容的。这个事实很有用，因为它使检查弱相容性更为容易——仅有基础项目需要针对条件1、2和3进行（逐对地）检查。

弱相容性的重要性在于：如果两个状态是弱相容的，则它们可以被安全地合并。为理解为什么是这样，首先观察合并两个状态可能引入归约-归约冲突，但不会引入移进-归约冲突。也就是说，如果出现移进-归约冲突，则它必定在合并任何状态之前就已经存在。特别地，冲突必定已经存在于这样的状态之中：它们中含有的需移进的符号可作为归约动作的超前搜索符号。

归约-归约冲突可能出现在合并后的状态中吗？弱相容性的条件1声明在合并之后添加的超前搜索符号不会产生冲突，条件2和条件3仅当在合并之前存在归约-归约冲突时成立，而在这种情况下无论如何都不会试图进行合并。

归约-归约冲突可能出现在合并状态的后继中吗？考虑s和 $\overline{s}$ 面临某个符号x时的直接后继。称他们为s_x和 $\overline{s_x}$。Basis(s_x)和Basis($\overline{s_x}$)是弱相容的，因为它们直接从已知是相容的s和 $\overline{s}$ 获得。如前面所提到的，已知如果基础项目集是弱相容的，则整个项目集也是。这直接暗示了如果合并s_x和 $\overline{s_x}$ 不会引入任何归约-归约冲突。重复该论证，所有后继状态都是弱相容的，且因此可以安全地进行合并。

弱相容性的概念导致了一个直截了当的LR(1)生成算法。当一个LR(1)状态s被创建时，我们检查已经创建的与s同心的状态。如果存在与s弱相容的这样一个状态 $\overline{s}$，则合并s和 $\overline{s}$。如果已经创建了 $\overline{s}$ 的后继，则试图用$s \cup \overline{s}$的后继来替换它们。但并不总能这样做，因为 $\overline{s}$ 的后继可能已经与其他状态合并。弱相容性能够用于验证替换是否可能。如果不可能，则创建一个新的后继状态。如果 $\overline{s}$ 的后继还未创建，合并s和 $\overline{s}$ 将导致稍后生成$s \cup \overline{s}$的后继状态。

如果底层文法实际上是LR(1)的，则该算法总是创建正确的LR(1)分析器。然而，它不一定产生最少状态LR(1)分析器。原因是弱相容性规则可能排除对实际上可以安全合并的状态的合并。例如，假定有图6-39所示的LR(1)状态。

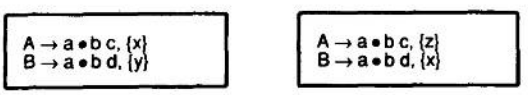


图6-39 能够被合并的LR(1)状态对

这两个状态同心但不是弱相容的。然而，它们可以被安全地合并，因为在此情况下不可能有归约-归约冲突，而且甚至不需要超前搜索符号。问题是超前搜索符号x暗示未来的归约-归约冲突，而事实上该冲突不会被触及。

196

实际上，这看起来没有问题，因为在“真正”的文法中弱相容性极少排除可行的合并。尽管如此，确实存在保证执行所有可行合并的方法。Pager（1977）定义了强相容性（strong compatibility）标准。这些标准通过下列要求改善了弱相容性标准：如果一个合并看起来会导致最终的归约-归约冲突，则该

冲突必须实际上可以被触及。在前面的例子中，项目A → a • bc, x和B → a • bd, x不会导致最终的归约-归约冲突，因此强相容性允许对其进行合并。另一种方法是，一旦已经利用弱相容性构造了LR(1)机器，类似于早先描述的一个回溯算法能够被用于探测尚未匹配的状态。如果不产生冲突，无法由弱相容性处理的状态将被合并。否则，进行回溯并考虑其他的可能性。

一个称为*LR*的完整LR(1)分析器生成器由Wetherell和Shannon（1981）所创建。LR系统利用Pager的弱相容性标准在创建状态时对其进行合并。LR已被很好地用于若干商用开发中，并已毫不费力地为Ada和PL/I生成了语法分析器。LR以标准FORTRAN语言编写并且很容易移植。

6.10 LR分析的性质

前面已经讨论的所有LR分析的变体都共享一些我们需要了解的重要的公共性质。我们已经讨论了最重要的性质：正确性。每个LR式的语法分析器都保证能正确地分析有效输入并在试图分析无效输入时检测出语法错误。此外，因为所有LR式语法分析器都仅接受活前缀，一旦分析器试图移进不能作为活前缀一部分的词法记号，就将检测到语法错误。这允许即时和方便的错误报告。

另一个重要的性质是*无二义性*（unambiguity）。即，如果已知一个文法适合任意LR式分析类，则该文法将不允许二义的推导。这个结果其实就是LR式语法分析器都是确定的这样一个事实的副作用。特别地，在每个状态中，以及对于每个超前搜索符号，都要求惟一的分析器动作。二义文法允许选择性的推导，这将导致在某个状态中的移进-归约冲突或归约-归约冲突。 197

所有LR式语法分析器所共有的另一个重要性质是它们都是高效的。如果我们正在分析一个含有n个词法记号的程序，则

- 分析栈决不会包含多于$c_1 \times n$个状态，其中c_1是由正在分析的文法所确定的一个常量。
- 语法分析器决不会进行多于$c_2 \times n$次移动，其中c_2也是由文法确定的一个常量。

也就是说，LR分析器的运行是*线性的*。该结论实际上是至关重要的，因为产品级编译器常常会分析含有数万个词法记号的程序。假如不能保证线性分析，大程序的编译会惊人地昂贵。

为证明线性性质，首先注意到，已知在无二义文法中任意串s的分析树在尺寸上线性正比于s。该结果是第4章练习12的推广。假如该事实不是真的，则利用任何技术的线性语法分析都将是不可能的，因为语法分析器在运行时都“发现”相应的分析树。

为证明LR分析器仅需要线性空间，我们注意到：在语法分析中出现的每个符号要么是终结符要么是非终结符，都恰好移进一次。符号总数正比于输入的大小，因此分析栈最大深度也同样正比于输入的大小。

为证明线性时间，我们首先注意到：语法分析器驱动程序的每次循环都需要有限的时间。因为每次循环消耗一个符号，而符号总数正比于输入长度，所以总分析时间也同样正比于输入长度。

如果包括动作符号，识别它们并调用语义例程所需的总时间也是线性的。这是因为每个产生式中仅有固定数量的动作符号，通常是一个。执行一个语义例程的时间不需要有限制。因此，尽管分析必定是线性的，语义处理和代码生成不必也是线性的（尽管实际上它们都是线性的，除非执行大量的优化）。

6.11 LL(1)和LALR(1)分析方法的比较

在真正的编译器中占统治地位的两种语法分析技术是LL(1)和LALR(1)。实际上所有现代编译器都使用这两种技术中的一种或是相近似的变体（例如，递归下降或SLR(1)）。编译器编写者意识到每种技术的优点和弱点并因此能够做出明智的选择是非常重要的。

在下列小节中，将依据许多标准分别对LL(1)和LALR(1)进行比较。正如所期望的，每种分析技术

都有它自己特殊的优点。在为特定的编译器设计选择分析技术之前，研究这些标准并选择该设计所需的最合适技术或许是个好主意。然而，在许多情况下，抽象因素也会产生影响。就像在家乡的舞蹈俱乐部中一样，首先学会的技术通常会成为我们最喜欢的。抛开这样的偏见不谈，没有哪种技术是普遍受欢迎的，而一个公平的比较将会很有价值。

198

简单性

LL(1)和LALR(1)都有非常简单的驱动程序。LL(1)的基本概念非常易于可视化及理解。LALR(1)则相对较为复杂，它含有项目、项目集、状态合并以及超前搜索符号传播等概念。因为使用了语法分析器生成器，语法分析器构造的内部细节通常是隐藏的。然而有时当调试一个文法（即对文法进行修正以使其符合一个语法分析类的要求）时，会涉及内部细节。在这种情况下，简单性无疑是有利因素，因此在这里更强大的文法是LL(1)。

通用性

在其他所有方面都相同时，语法分析技术能够处理愈广泛的文法类愈好。LL(1)和LALR(1)都不能处理所有非二义文法。然而，所有LL(1)文法都是LR(1)的，而且实际上也都是LALR(1)的。LL(1)分析器对于文法形式的要求相当严格，它禁止左递归和共享公共前缀的产生式。

很容易将一个文法变为LALR(1)的形式，而且实际上所有现代语言的设计都适于构造LALR(1)分析器。事实上，谨慎的语言设计努力常常包括一个已经是LALR(1)形式的“参考文法”。

相反，大多数参考文法必须重写为LL(1)形式。在极少的情况下，像Pascal的悬空else这样的结构不存在非二义的LL(1)表述。非LL(1)语言结构很少见。一个有效的经验是对于任意合理的程序设计语言都能构造LL(1)和LALR(1)文法。然而，LALR(1)文法几乎肯定更容易书写并更容易阅读。总之，无论如何，LALR(1)在通用性方面有着明显的优势。

动作符号

动作符号是语法分析器和语义例程之间的接口。LL(1)允许将动作符号放置在产生式右部的任何地方。LALR(1)允许将动作符号放置在产生式的最右端但不允许放在任何其他地方。然而，通常LALR(1)文法可以被重写以允许必要的语义例程调用。事实上，Yacc允许将代码片段放置在产生式右部的任何地方，并以6.6节结尾所描述的方式自动引入新的匿名非终结符号。通常不会由此引入分析冲突，这在语义动作的放置方面给予Yacc与LL(1)分析器几乎相同的灵活性。

199

总之，LL(1)在动作符号的放置方面允许最佳的灵活性。某些LALR(1)生成器几乎同样灵活，但其他一些LALR(1)生成器则在语义动作的放置方面显然更具限制性。

错误修复

错误修复在第17章中详细讨论。简单地说，LL(1)分析栈中包含已经预测但尚未匹配的符号。该信息对于确定可能的修复无疑是很有价值的。LALR(1)分析栈中包含关于已经看到的符号的信息。而决定可能用作修复的后续动作却不容易。因此，LL(1)错误修复倾向于更为简单。

分析表大小

LL(1)和LALR(1)都需要可能相当大的分析表。如果编译器的大小是需要考虑的问题，则比较这两种分析技术所需分析表的相对大小是有益的。

对于LL(1)分析器而言，未压缩的分析表大小为$|V_n| \times |V_t|$。此外，还需要一张产生式右部及其大小的表，其大小为$|G|$，其中$|G|$是所有产生式长度的和。

相反，LALR(1)分析器需要一张最大为$|States| \times (|V_n|+|V_t|)$的未压缩分析表，以及一张大小为$2 \times |P|$的产生式长度和产生式左部的表。在最坏情况下（该情况是可能出现的，见练习35），状态数相对于文法的大小是指数阶的。即，LALR(1)状态对应于项目集。不同项目的数量等于$|G|$，因此不同项目集的数量是$2^{|G|}$。

因为LALR(1)分析表大小可能呈指数爆炸，所以LL(1) 看似有较安全的最坏情况行为。然而，更合理的比较是在期望情况而不是最坏情况下获得的。当然，普通文法不会导致指数阶的分析表大小。就比较目的而言，下列经验规则通常用于典型的程序设计语言文法：

- $|V_t| \approx 0.5 \times |V_n|$
- $|P| \approx 2 \times |V_n|$
- $|G| \approx 7 \times |V_n|$（即，$3.5 \times |P|$）
- $|States| \approx |P| \approx 2 \times |V_n|$

利用这些规则，可以计算出分析表条目的粗略估计值。因为分析表通常以压缩形式使用，所以我们仅考虑非错误条目。这里，对LL(1)采用10%的非错误条目估计值，而对LALR(1)采用5%的估计值。

因此，LL(1)条目的数量为

$$\begin{aligned}|LL(1)| &\approx 0.1 \times |V_n| \times |V_t| + |G| \\ &= 0.1 \times |V_n| \times 0.5 \times |V_n| + 7 \times |V_n| \\ &= 0.05 \times |V_n|^2 + 7 \times |V_n|\end{aligned}$$

200

类似地，LALR(1)条目的数量为

$$\begin{aligned}|LALR(1)| &\approx 0.05 \times |P| \times (|V_n| + |V_t|) + 2 \times |P| \\ &= 0.1 \times |V_n| \times (|V_n| + 0.5 \times |V_n|) + 2 \times 2 \times |V_n| \\ &= 0.1 \times |V_n| \times (1.5 \times |V_n|) + 4 \times |V_n| \\ &= 0.15 \times |V_n|^2 + 4 \times |V_n|\end{aligned}$$

通常，|LALR(1)|会比较大，极限比为

$$\lim_{|V_n| \to \infty} \frac{|LALR(1)|}{|LL(1)|} =$$

$$\lim_{|V_n| \to \infty} \frac{0.15 \times |V_n|^2 + 4 \times |V_n|}{0.05 \times |V_n|^2 + 7 \times |V_n|} = 3$$

对于典型的程序设计语言，$|V_n| \approx 100$。由此给出比值：

$$\frac{0.15 \times 100^2 + 4 \times 100}{0.05 \times 100^2 + 7 \times 100} = \frac{1500 + 4 \times 100}{500 + 7 \times 100} \approx 1.58$$

在该范围内，LL(1)表中的产生式右部约需要与LL(1)分析表自身同样大的空间。为评价这些估算的合理性，让我们考虑两个实际的Pascal文法，一个是LL(1)文法，另一个是LALR(1)文法。

在LL(1) Pascal文法中，$|V_n| = 125$，$|V_t| = 60$，$|P| = 234$。有591个非错误分析表条目和640个产生式右部信息条目。与预测值$0.05 \times 125^2 + 7 \times 125 \approx 1656$相比，这里给出了1231个条目。

类似地，LALR(1) Pascal文法有$|V_t| = 60$，$|V_n| = 127$，$|P| = 280$。它有2798个非错误分析表条目和560个产生式长度和产生式左部信息条目，与预测值$0.15 \times 127^2 + 4 \times 127 \approx 2927$相比，总共有3358个条目。

比值是2927 / 1656 ≈ 1.77。因此，LALR(1)与LL(1)分析表大小的比值呈现出约为2比1的良好工作值。LL(1)在空间需求方面有明显的优势，而在空间紧张的情况下，这种差异会成为选择LL(1)的决定性因素。

因为LALR(1)和LL(1)的驱动程序都检查分析树中的每个终结符和非终结符，所以可以预期它们的分析速度是相当的。

对比较的总结

LL(1)在除通用性之外的所有领域都有优势，而LALR(1)则在通用性方面有明显的优势。尽管如此，

在所有领域中这两种分析都完全可行。如果一个LL(1)文法已经可用，则LL(1)对于第一个编译器可能是更好的选择。知识渊博的编译器编写者应当同时精通这两种技术。随后，经验以及适当语法分析器生成
201 器的可用性将会指导个人的选择。

6.12 其他的移进-归约技术

本章所讨论的基于LR的语法分析技术目前在移进-归约分析器中占主导地位。为完整起见，本节中将讨论许多其他的移进-归约技术。这些技术主要有历史和理论上的意义。尽管如此，简短的概览将说明近年来开发的扩展方法和其他可选的方法。

6.12.1 扩展的超前搜索技术

SLR(1)、LALR(1)和LR(1)语法分析技术可以被推广以利用k个超前搜索符号。其结果为SLR(k)、LALR(k)和LR(k)技术，它们利用k个符号的First和Follow集，由$First_k$和$Follow_k$表示。

推广是直截了当的：

(1) **LR(k)**：当求一个包含$A \to \alpha \bullet B\beta, x$的项目集的闭包时，预测$B \to \bullet\gamma, y$，其中$y \in First_k(\beta x)$。

如果当前状态包含$A \to \alpha \bullet, x$，则在面临超前搜索符号x时按照产生式$A \to \alpha$归约。

如果当前状态包含$A \to \alpha \bullet a\beta, y$，其中$a \in V_t$且$x \in First_k(a\beta y)$，则在面临超前搜索符号x时移进。

(2) **SLR(k)**：如果当前状态包含$A \to \alpha \bullet$，其中$x \in Follow_k(A)$，则在面临超前搜索符号x时按照产生式$A \to \alpha$归约。

如果当前状态包含$A \to \alpha \bullet a\beta$，其中$a \in V_t$且$x \in First_k(a\beta Follow_k(A))$，则在面临超前搜索符号x时移进。

(3) **LALR(k)**：合并LR(k)机器中同心的那些状态和语法分析器动作。

正如所期望的，这些推广扩展了可被分析的文法类。例如，如果LR(k)表示可利用k个符号超前搜索的LR技术进行分析的文法集，则容易证明$LR(0) \subset LR(1) \subset \cdots LR(k) \subset LR(k+1) \cdots$。对于SLR(k)和LALR(k)适用类似的包含结果。可以提出任意数量的包含问题，例如：是否存在是LR(1)和LALR(2)、但不是LALR(1)的文法？以及是否存在是SLR(3)但不是SLR(2)的文法？（答案是存在。）

在实际意义上，扩展的超前搜索技术几乎没有价值。对于扩展的超前搜索技术，其分析表大小迅速增加。比传统分析表大一个或两个数量级的分析表是否能够被认真考虑是个问题。更重要的是，看起来并不真正需要扩展的超前搜索技术。也就是说，用于定义程序设计语言的文法仅需要一个超前搜索符号。在看起来需要额外超前搜索符号的情况下，简单的文法变换能够将超前搜索符号的需求缩减为一个单独的符号。理论通过以下观察结果支持实践：已知所有可被确定分析的语言都有SLR(1)文法（如果使用结束标记，它们甚至都有LR(0)文法）。这暗示未来的程序设计语言将能够继续利用传统的单个符号超前搜
202 索技术进行分析。

6.12.2 优先级技术

到目前为止我们已经研究的LR式语法分析技术都相当复杂。因此，它们看来起不可能是第一种被开发和研究的移进-归约技术。而且，事实上，在LR技术占据支配地位之前，一大类*优先级技术*（precedence technique）曾被透彻研究并广泛使用。

优先级技术处理移进-归约分析的基本问题——怎样分离并归约右句型的句柄。我们将简要讨论两种最著名的优先级技术：*简单优先*（simple precedence）和*算符优先*（operator precedence）。

我们将要讨论的简单优先版本由Wirth和Weber（1966）形式化。该技术被开发用于分析Algol W语言。简单优先相当简单。在文法符号对上定义三种*优先关系*，$\lessdot$、$\doteq$和$\gtrdot$。如果一个文法是简单优先

文法，则一对符号至多能够出现这三种关系中的一种。非正式地说，$\lessdot$ 标记句柄的开头（左端），$\doteq$ 界定句柄的内部，而 $\gtrdot$ 则界定句柄的结尾（右端）。

这些关系可以很好地适应移进-归约分析器。在符号被读入的同时，只要符合 $\lessdot$ 或 $\doteq$ 关系，它们就被压入分析栈中。当栈顶符号和超前搜索符号符合 $\gtrdot$ 关系时，句柄结尾已被定位。随后符号被弹出，直到栈顶符号和最后弹出的符号符合 $\lessdot$ 关系为止。被弹出的符号形成将要归约的句柄。在归约之后，产生式左部的符号通常取代了栈中的句柄。

一旦句柄被分离，对于哪个产生式将被归约必须没有歧义。这通过要求*惟一可逆性质*（unique invertibility property）来保证，该性质是指两个产生式不能有相同的右部。给定惟一可逆性，分离句柄等价于确定正确的归约。

作为示例，考虑G_8:

S → E
E → F
F → F+T | T
T → ID | (E)

此文法是G_1的变体，其中添加了非终结符以消除优先关系冲突。文法中添加了左结束标记以（通过 $\lessdot$ 关系）分离句柄的开头。可以为G_8计算图6-40所示的*优先表*。（关于如何计算优先关系的细节，见 Aho and Ullman 1972，卷1，6.2.2节。）

	E	F	T	ID	+	(	)	$
E							$\doteq$	$\doteq$
F					$\doteq$		$\gtrdot$	$\gtrdot$
T					$\gtrdot$		$\gtrdot$	$\gtrdot$
ID					$\gtrdot$		$\gtrdot$	$\gtrdot$
+			$\doteq$	$\lessdot$		$\lessdot$		
(	$\doteq$	$\lessdot$	$\lessdot$	$\lessdot$		$\lessdot$		
)					$\gtrdot$		$\gtrdot$	$\gtrdot$
$	$\doteq$	$\lessdot$	$\lessdot$	$\lessdot$		$\lessdot$		

图6-40 G_8的简单优先级分析表

图6-41举例说明了$ID+(ID+ID)$的简单优先级分析。当且仅当输入被归约为E时，分析器接受输入。优先关系仅出于说明的目的而给出；它们并未被实际压入栈中。 203

步骤	分析栈	剩余输入
1		$ID+(ID+ID)$
2	$ $\lessdot$	ID+(ID+ID)$
3	$ $\lessdot$ ID $\gtrdot$	+(ID+ID)$
4	$ $\lessdot$ T $\gtrdot$	+(ID+ID)$
5	$ $\lessdot$ F $\doteq$	+(ID+ID)$
6	$ $\lessdot$ F $\doteq$ + $\lessdot$	(ID+ID)$
7	$ $\lessdot$ F $\doteq$ + $\lessdot$ ($\lessdot$	ID+ID)$
8	$ $\lessdot$ F $\doteq$ + $\lessdot$ ($\lessdot$ ID $\gtrdot$	+ID)$
9	$ $\lessdot$ F $\doteq$ + $\lessdot$ ($\lessdot$ T $\gtrdot$	+ID)$
10	$ $\lessdot$ F $\doteq$ + $\lessdot$ ($\lessdot$ F $\doteq$	+id)$
11	$ $\lessdot$ F $\doteq$ + $\lessdot$ ($\lessdot$ F $\doteq$ + $\lessdot$	ID)$
12	$ $\lessdot$ F $\doteq$ + $\lessdot$ ($\lessdot$ F $\doteq$ + $\lessdot$ ID $\gtrdot$	)$
13	$ $\lessdot$ F $\doteq$ + $\lessdot$ ($\lessdot$ F $\doteq$ + $\doteq$ T $\gtrdot$	)$
14	$ $\lessdot$ F $\doteq$ + $\lessdot$ ($\lessdot$ F $\gtrdot$	)$
15	$ $\lessdot$ F $\doteq$ + $\lessdot$ ($\doteq$ E $\doteq$	)$
16	$ $\lessdot$ F $\doteq$ + $\lessdot$ ($\doteq$ E $\doteq$) $\gtrdot$	$
17	$ $\lessdot$ F $\doteq$ + $\doteq$ T $\gtrdot$	$
18	$ $\lessdot$ F $\gtrdot$	$
19	$ $\doteq$ E $\doteq$	$
20	$ $\doteq$ E $\doteq$ $	

图6-41 简单优先级分析示例

尽管简单优先级在概念上比LR技术简单，但它有许多缺点，主要是它所能分析的文法类有限。所
204 有简单优先文法都是SLR(1)的，但反之并不成立。SLR(1)文法相当容易书写，但简单优先文法却非常难以构造。注意：在简单优先文法中不允许出现λ产生式，因为无法利用优先关系正确地将λ分离为句柄。进一步，产生式是惟一可逆的以及优先关系无交集这两条要求使得优先文法比相应的LR文法更大且更脆弱。它们更大是因为新的非终结符对于消除优先冲突通常是必需的。它们脆弱是因为即使很小的修改或增补都会引起微妙的优先冲突。事实上，存在没有简单优先文法的确定性语言（它们能够由所有LR技术分析）。

在给出了简单优先的缺点之后，我们对于该技术被SLR(1)和LALR(1)取代就不会感到惊讶。简单优先现在通常只有历史意义，尽管它偶尔会在老式的编译器中出现。

*算符优先分析*技术可追溯到20世纪60年代早期，当时语法分析仅涉及比将表达式从中缀翻译为前缀或后缀形式稍多的东西。实际上，算符优先主要是程序设计语言中出现的运算符优先级的形式化。当分析一个表达式时，算符优先分析器试图分离可粗略地等价于句柄的简单子表达式。仅定义在终结符上的算符优先关系完成这项工作。基本上，Op1 ⋖ Op2，如果运算符Op2比Op1有更高的优先级。因此，通常+ ⋖ *且* ⋗ +。如果Op1和Op2处于相同的优先级并且都是左结合的，Op1 ⋗ Op2。括号强制进行分组，因此对于所有运算符和标识符，Op ⋖ (，(⋖ Op，Op ⋗)，且) ⋗ Op。例如，分析$ID+(ID+ID)$，有

$ ⋖ ID + ⋖ (⋖ ID + ID ⋗) ⋗ $

其中⋖和⋗分离了表达式。类似地，对于$ID*ID+ID*ID$，有

$ ⋖ ID*ID ⋗ + ⋖ ID*ID ⋗ $

其中再次强制正确的分组。

因为算符优先关注终结符，所以包含两个连续非终结符的产生式是被禁止的。这比简单优先更具限制性，而简单优先更通用且内容更丰富。从历史上看，简单优先取代了算符优先，并且随后又让位于今天使用的LR技术。

6.12.3 一般的上下文无关分析器

到目前为止我们已经描述的所有分析技术都局限于上下文无关文法的某个子集。通常，仅允许非二义文法。如果允许二义文法，则正确性的保证不再有效。进一步，由于超前搜索符号是固定的，因此仅
205 有确定性语言和文法能够被处理。

偶尔，我们也会需要能够处理任意文法的通用上下文无关分析器。已知最佳的通用上下文无关分析器是*Earley算法*（Earley's algorithm），由Jay Earley（1970）所设计。Earley算法可以被视为LR(0)分析器的*解释性*版本。它比我们已经研究的LR式技术更通用，当然它的代价昂贵也就不足为奇。普通的语法分析技术（包括LL、LR和优先级方法）通常在线性时间和线性空间内工作；Earley算法有时需要立方阶的时间和四次方阶的空间，尽管这种非线性的性能通常发生在分析超出普通分析器能力的文法时。也存在能够处理任意上下文无关文法的自顶向下语法分析技术（Graham, Harrison, and Ruzzo 1980）。

不像LR(k)分析，Earley算法从不需要或使用超前搜索符号。无论什么时候，只要对何时进行归约有疑问，将平行地采取所有可能的动作。进一步，我们不预先计算任何状态或自动机，而是在分析进行中计算并操纵项目集，这就是认为该算法是解释性的原因。

在Earley算法中，项目（加点产生式）通过第二个部分——一个整型值的*预测指针*（prediction pointer）来拓广。项目的预测指针表示其所在的项目集在哪个项目集中被预测。该指针是必需的，因为二义文法可能多次预测相同的产生式以表示不同的分析。项目通过与普通LR式分析器几乎相同的方式被操纵：

- 初始项目是S → •α, 0。(初始时预测拓广产生式，其预测指针为0，其中0是状态0的索引。)
- 如果状态i中包含项目A → α•Bβ, j，则预测B → •γ, i。预测指针为i是因为项目在状态i中被预测。
- 如果状态i中包含项目A → α•Xβ, j，且下一个输入符号是X，则将A → αX•β, j添加到状态i + 1中。
- 如果状态i中包含项目A → γ•, j，则我们知道该产生式在状态j中被预测。到状态j并查找形如B → α•Aβ, k的项目。将形如B → αA•β, k的相应项目添加到状态i中。该操作与归约动作性质相同，因为已匹配的产生式右部被它的左部替换，而其左部随后被移进。
- 在分析了包含n个记号的输入后，将创建n + 1个项目集。如果项目n含有S → α•, 0，则输入是有效的，因为输入使得目标符号得以匹配。

作为简单的示例，考虑二义文法: 206

S → E
E → E + E | ID

图6-42通过分析ID + ID + ID举例说明了Earley算法。该输入有两种不同的分析方法；该算法可以同时找出两者。

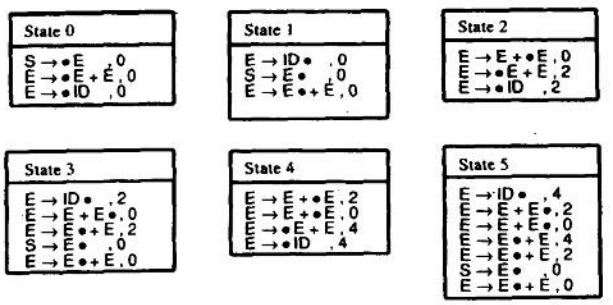


图6-42 Earley算法示例

因为状态5中包含S → E •, 0，所以我们知道输入是有效的。事实上，因为该输入有两种不同的分析方法，所以该项目被添加了两次。项目E → E + E •, 0的完成导致拓广产生式被完成。这对应于将输入分析为(ID + ID) + ID。项目E → E + E •, 2的完成导致状态2中的E → ID •, 2被完成，而它又依次导致拓广产生式被完成。这对应于将输入分析为ID + (ID + ID)。

通常，为提取一个分析，一种实现方法是添加“线索”，将一个项目连接到添加它的那个已完成的项目。如果一个项目被添加了多次，则不同的线索指向不同的分析。

从我们的简单示例中容易看出随着分析越来越多的输入，项目集可能变得更大。这是因为相同的加点产生式可以出现多次，其中带有不同的预测指针。状态i的大小可能正比于i，因此在n个状态中的项目数可能正比于n^2。类似地，因为二义性，所以一个项目可以被添加多次。要小心，可能获得正比于n^3的分析时间。更快($n^{\log_2 7}$)但是也更复杂的通用上下文无关分析器由Valiant (1975)创建。

在LR(0)文法的情形中，Earley算法仅需要线性的时间和空间，但会有非常多的额外开销因素，因为在Earley算法的每一步，必须操纵并存储项目，而这些项目对普通的LR(0)分析器来说是被预先计算出来的。

Earley算法曾被实际使用过吗？实际上，它的空间和时间需求使其过于昂贵，也只在实验性系统中会有价值。事实上，需要二义文法或非限制性文法的应用通常用像Earley算法这样的通用语法分析器做实验。有此类需求的应用领域包括自然语言处理、自动代码生成（Christopher et al. 1984）以及可由用户扩展的语言开发。已知有许多可以改进Earley算法的方法（Graham, Harrison, and Ruzzo 1980, Aretz 1989)。 207

练习

1. 对于输入串

   begin begin SimpleStmt ; **end** ; SimpleStmt ; **end $**

   使用图6-2和图6-3的action和go_to表跟踪图6-1的移进-归约驱动程序的执行。

2. 下列产生式已经用动作符号进行了扩充，以规定一个简单**while**循环的翻译：

   ```
   <stmt> → while #loop_head <expr> #test_expr
       loop <stmts> end loop ; #loop_end
   ```

   将该产生式改写为适合于移进-归约分析的形式。

3. 为下列文法构造CFSM：

   ```
   <prog>      → <block> $
   <block>     → begin <stmt list> end
   <stmt list> → <stmt list> ; <stmt>
   <stmt list> → <stmt>
   <stmt>      → <block>
   <stmt>      → <var> := <expr>
   <var>       → ID
   <var>       → ID [ <expr> ]
   <expr>      → <expr> + <term>
   <expr>      → <term>
   <term>      → <var>
   <term>      → ( <expr> )
   ```

   给出相应的go_to表。

4. 下列文法中有哪些不是LR(0)的？解释为什么。

   a.
   ```
   S     → StList $
   StList → StList ; Stmt
   StList → Stmt
   Stmt  → null
   ```

   b.
   ```
   S     → StList $
   StList → Stmt ; StList
   StList → Stmt
   Stmt  → null
   ```

208

   c.
   ```
   S     → StList $
   StList → StList ; StList
   StList → Stmt
   Stmt  → null
   ```

   d.
   ```
   S     → StList $
   StList → null StTail
   StTail → λ
   StTail → ; StList
   ```

5. 证明相应于LL(1)文法的CFSM有如下性质：如果所有非终结符都推导出不同于λ的串，则每个项目集都恰好有一个基础项目。

6. 构造相应于练习3的文法的LR(1)机器。

7. 下列文法中有哪些是LR(1)的？哪些是LALR(1)的？哪些是SLR(1)的？在每种情形下都证明你的分类是恰当的。

   a.
   ```
   S → ID := E ;
   E → E + P
   E → P
   P → ID
   P → ( E )
   P → ID := E
   ```

   b.
   ```
   S → ID := A ;
   A → ID := A
   A → E
   E → E + P
   E → P
   P → ID
   P → ( A )
   ```

209

   c.
   ```
   S → ID := A ;
   A → ID := A
   A → E
   E → E + P
   E → P
   E → P +
   P → ID
   P → ( A )
   ```

   d.
   ```
   S → ID := A ;
   A → Pre E
   Pre→ Pre ID :=
   Pre→ λ
   E → E + P
   E → P
   P → ID
   P → ( A )
   ```

e.
```
S → ID := A ;
A → Pre E
Pre→ ID := Pre
Pre→ λ
E → E + P
E → P
P → ID
P → ( A )
```

f.
```
S → ID := A ;
A → ID := A
A → E
E → E + P
E → P
P → ID
P → ( A ; A )
P → ( V , V )
P → { A , A }
P → { V ; V }
V → ID
```

210

g.
```
S → ID := A ;
A → ID := A
A → E
E → E + P
E → P
P → ID
P → ( ID ; ID )
P → ( A )
```

8. 证明LR(1)项目的超前搜索部分是精确的。即，
   (a) 如果状态s包含一个LR(1)项目A → α •, a，则存在最右推导$S \Rightarrow^*_{rm} \beta Aaw \Rightarrow_{rm} \beta\alpha aw$，其中在移进βα后到达状态s。
   (b) 如果存在最右推导$S \Rightarrow^*_{rm} \beta Aaw \Rightarrow_{rm} \beta\alpha aw$，则存在状态s，它在移进βα之后到达，其中包含项目A → α •, a。
9. 构造相应于练习3的文法的SLR(1) action表。
10. 取为练习6构造的LR(1)机器并列出同心状态。合并同心状态以构造LALR(1)机器。
11. 从练习3中所构造的CFSM开始，使用图6-25的超前搜索符号传播算法来确定LALR(1)超前搜索符号。比较用这种方法计算出的超前搜索符号和练习10中通过状态合并计算出的那些超前搜索符号。
12. 修改图6-25的超前搜索符号传播算法，把（状态，项目）偶对而不是（状态，项目，超前搜索符号）三元组压入栈中。解释为什么你修改过的算法与原始算法传播相同的超前搜索符号集。
13. 在6.5.1节中描述反向搜索超前搜索符号传播算法时，曾注意到一个归约序列可能在移进超前搜索符号之前发生。编写一个能够正确处理归约序列的反向搜索传播算法。利用练习3的CFSM举例说明你的算法。
14. 图6-30举例说明了一个可以导致反向搜索传播算法失败的CFSM状态。修改你在练习13中创建的算法，以正确地处理图6-30中说明的问题。你修改过的算法对于所有CFSM和文法都能够计算正确的超前搜索符号吗？

211

15. 下面的文法不是LALR(1)的：

```
<prog>      → <block>
<block>     → begin <decl list> <stmt list> end
<decl list> → <decl> <decl list>
<decl list> → λ
<stmt list> → <stmt list> <stmt>
<stmt list> → <stmt>
<decl>      → ID : <type> ;
<type>      → ID
<type>      → array ( <bound> ) of <type>
<bound>     → ID
<bound>     → <expr>
<stmt>      → ID := <expr> ;
<stmt>      → <block> ;
<stmt>      → ID : <stmt>
<expr>      → <expr> + <pri>
<expr>      → <pri>
<pri>       → ( <expr> )
<pri>       → ID
<pri>       → ID +
```

将该文法改写为可被LALRGen或者Yacc读取的形式。使用你所选择的语法分析器生成器确定出故

障的地方。然后将这些产生式重写为有效的LALR(1)形式。可以根据需要改变产生式，但由你修改的文法所定义的语言必须和原始文法所定义的语言相同。

16. 为图6-33所定义的MicroPlus的表达式结构编写上下文无关文法。你写的文法必须遵守图6-33中所示的运算符优先级和结合性。

17. 利用Yacc风格的优先级和结合性定义，定义附录A中所规定的Ada/CS的表达式结构。注意除关系运算符外的所有二元运算符都是左结合的。如果你能够使用运行Yacc的计算机，就请测试你的定义。

18. 在Yacc中不可能给出把左结合和右结合运算符放在同一优先级的优先级和结合性定义。这是一个疏漏，还是有某种原因不允许这样的定义？

19. 假定有一个Pascal式的语言，拥有如下列结构：

```
if <expr> then <stmt>
if <expr> then <stmt> else <stmt>
while <expr> do <stmt>
```

212

给出能够定义这些结构并正确处理悬空else问题的非二义LALR(1)文法。

20. 扩展图6-1的移进-归约驱动程序，以正确处理利用6.8节的技术优化分析表时所创建的L类型条目和默认条目。

21. 通过LALRGen，利用parsetable选项运行Ada/CS定义。LALRGen使用L类型条目来消除单独的归约状态。它不删除默认动作。从parsetable选项所产生的列表中估算通过消除单独的归约状态节省了多少空间。如果从分析表中删除默认状态，则又能够多节省多少空间？

22. 修改图6-19的SLR(1)分析表以包含对单位归约的优化。即，修改后的分析表不应当执行一系列的单位归约，而是执行折叠此归约链的一次单独的归约。

23. 在6.9节中概括叙述了一个LR(1)状态合并算法。只要不引入分析动作冲突，它就会合并同心的LR(1)状态。如果两个状态的合并导致其后继状态的合并冲突，则该算法回溯并取消导致该非法合并的前驱状态的合并。

详细说明实现上述方法的算法。举例说明你的算法在下列文法上的操作：

```
S    → ( Exp1 )
S    → [ Exp1 ]
S    → ( Exp2 ]
S    → [ Exp2 )
S    → { Exp1 }
S    → < Exp1 >
S    → { Exp2 >
S    → < Exp2 }
Exp1 → # ID
Exp2 → # ID
```

24. 用于合并LR(1)状态的最普通的标准是弱相容性，如6.9节中的定义。为练习23的文法创建LR(1)机器，并证明可以利用弱相容性对状态进行合并。

25. 证明当优化LR(1)分析器时合并LR(1)状态的次序可能会导致很大的差别。也就是说，如果以某种次序合并状态，则优化过的LR(1)机器的大小可能比选择某个其他次序时更大。

26. 证明如果一个LR(1)状态的基础项目是弱相容的，则该状态中的所有项目必定是弱相容的。

213

27. 证明任意没有λ产生式的LL(1)文法是LR(0)的。

28. 证明存在不是LL(1)的LR(0)文法、SLR(1)文法以及LALR(1)文法。

29. 通常，LALR(1)分析器产生规范分析或最右分析。如何利用LALR(1)分析器产生最左分析（像LL(1)分析器那样）？如果已知所分析的文法是LL(1)的，问题会被简化吗？

30. 假定有一个使用LL(1)分析表的可以工作的编译器。根据经验，几乎所有的LL(1)文法同时也是LALR(1)的。因此，将LL(1)分析器替换为LALR(1)分析器应当是可行的。必须做什么才能保证该替

换对编译器的其余部分是透明的?

31. 证明所有LL(1)文法也是LR(1)的。
32. 证明存在是SLR(k+1)和LALR(k+1)以及LR(k+1)的、但不是SLR(k)或LALR(k)或LR(k)的文法。
33. 构造拥有下列所有性质的文法:
    - 它是SLR(3)的、但不是SLR(2)的。
    - 它是LALR(2)的、但不是LALR(1)的。
    - 它是LR(1)的。
34. 证明每个SLR(1)文法也是LALR(1)的，并且每个LALR(1)文法也是LR(1)的。如果使用k个符号而不是一个符号的超前搜索，这些包含关系还成立吗?
35. 考虑下列拥有$O(n^2)$个产生式的文法:

$$S \rightarrow X_i z_i \qquad 1 \le i \le n$$
$$X_i \rightarrow y_j X_i \mid y_j \qquad 1 \le i,j \le n,\ i \ne j$$

    证明该文法的CFSM有$O(2^n)$个状态。该文法是SLR(1)的吗?
36. 利用图6-40的分析表，当输入为$((ID+ID)+(ID+ID))$时，跟踪一个简单优先分析器的执行。
37. 证明所有简单优先文法都是SLR(1)的。这可以通过证明以下命题来完成：利用SLR(1)式的超前搜索符号不可解决的移进-归约或归约-归约冲突必定导致三种简单优先关系间的冲突，或者必定违反简单优先文法的惟一可逆性质。
38. 当使用下列文法分析aaa时，给出Earley算法所创建的状态:

$$A \rightarrow A\ B \mid B$$
$$B \rightarrow B\ A \mid A$$
$$A \rightarrow a$$
$$B \rightarrow b$$

214

39. 当分析某些文法时，Earley算法将总是产生大小限制在某常数值之内的状态，而不依赖于所分析的输入的大小。这样的文法被称为受限状态文法（bounded state grammar）。我们不难证明Earley算法能够在线性时间内分析受限状态文法。又已知Earley算法能够在线性时间内分析所有LR(0)文法。是否所有的LR(0)文法都必须是受限状态文法?

215

第7章 语义处理

几乎所有的现代编译器都是*语法制导的*（syntax-directed）。即，随着语法分析器识别源程序的语法结构，由这些语法结构来驱动编译过程。语义例程——编译器解释程序含义（语义）的部分——基于程序的语法结构执行解释工作。

在由编译器所完成的处理中，这些例程实际上扮演双重角色，它们完成编译的分析任务并随后开始综合任务。分析，即语义例程将从声明获得的语义信息与标识符的所有使用相关联，并检查程序满足语言中的所有静态语义约束。静态语义检查的例子包括确定程序中使用的所有变量都被声明，以及表达式中的操作数类型与相应运算符相容。综合是以生成程序的*中间表示*（Intermediate Representation，IR）或 216
实际的目标代码开始的。语义例程由综合步骤得名，因为这些例程的输出必须反映由词法分析器和语法分析器所识别的语法结构。由于翻译过程的语法制导特性，语义例程通常与上下文无关文法的单独产生式或语法树的子树相关联。

7.1 语法制导翻译

7.1.1 使用分析的语法树表示

我们首先看看最一般的方法来开始对语义处理的讨论，随后由此考虑特殊的实现选择。因为我们的翻译过程归类为语法驱动的，概念性的方法首先考虑我们对于在语法分析器检查表示一个程序的词法记号时所识别的语法结构做些什么。在先前讨论自顶向下和自底向上的语法分析的章节中，我们看到由语法分析器所生成的动作序列可以解释为一系列构造分析树的指令。因此在语义处理的一般性方法中第一步是取得语法分析动作序列并用它们来构造输入程序的*语法树*（syntax tree）表示。典型地，该分析树并非由语法分析器所识别的字面上的分析树。例如，它不需要在表达式子树中包含代表非终结符的中间结点，这些非终结符通常被引入程序中用于描述运算符优先级和结合性。它确实包含了足够的结构去驱动语义处理，甚至可以重新生成产生它的输入。这样的树通常被称为*抽象语法树*（abstract syntax tree），以表明它和实际（*具体*，concrete）语法和分析树的关系。图7-1给出一条赋值语句的抽象语法树表示，在该语句的右边含有算术表达式。

给定这种表示，语义处理可以通过对树的一次或多次遍历来完成。两种语义处理任务——静态语义检查和IR或代码生成——可以利用依附于语法树结点的语义*属性*来完成。当语义处理开始时，仅有的可用属性是那些依附于语法树上与拥有语义值的词法记号（例如，标记符和常量）对应的叶结点的属性。这样的初始状态在图7-2中说明。

对于大多数语言，以后序遍历来访问结点的树遍历方法可以将语义属性传遍整棵树并同时进行静态语义检查。语义属性的*传播*（propagation）包括如下动作：处理声明以创建符号表，在符号表中查找标识符以将相关的属性信息依附于语法树的适当结点上，以及检查运算符的参数类型以确定其结果类型。来自声明（符号表）的信息自顶向下地传播并穿越语句和表达式子树。语义检查和代码生成所用到的表达式类型信息通常自底向上传播。所有属性传播完成之后，树被称为是*经过修饰的*（decorated），而且它包含足够的信息驱动代码生成。 217

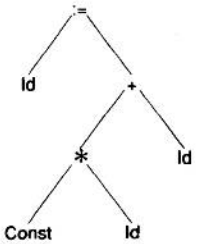


图7-1 Y := 3 * X + I的抽象语法树

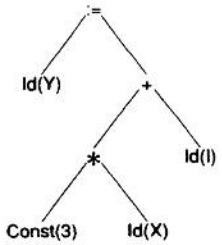


图7-2 带有初始属性的Y := 3 * X + I的抽象语法树

在以上示例中，所有标识符都必须在符号表中查找以确定它们的类型和存储地址等信息。这些信息项目（或符号表条目的引用）随后成为标识符结点的属性。一旦这些属性可用，我们就能够确定操作3 * X的合法性，并且，如果合法，还可以确定它的结果类型。此后，对第二个表达式也可以做同样
218 的事情，而且最终得以检查整个赋值操作的合法性。

有两类静态语义检查（与属性传播相互作用）。某些检查完全依赖于被传播的语义属性。对于赋值运算两边的类型相容性检查就是一个例子。其他的检查则将来自语法树的结构信息和语义信息相结合。将包含在过程调用中的实参数目（结构信息）与其声明中的形参数目（根据过程名传播而来的语义信息）相比较就是后者的一个例子。

语义处理的翻译任务基于语法树的另一次遍历。与每个结点相关联的语义属性是在翻译过程中使用的数据，而翻译则实质上由语法树的结构所驱动。翻译的输出可以是多种形式中的任意一种。可以由这种方式直接输出机器代码；可以生成某些线性化的中间表示（如第二章中见到的元组）；也可以将树本身以及*代码生成属性*（code generation attribute）用于优化器或代码生成器的输入。

使用*属性文法*（attribute grammar）符号可以形式化地描述这种以语法树结构和树的多遍遍历为中心来组织编译器的方法。属性文法是上下文无关文法的扩展，其中属性与产生式中的每个文法符号相关联，而且附属于每个产生式的规则将用来计算属性值。属性信息可以从树的叶结点向上流动（*综合属性*，synthesized attribute）或者从高层的非终结符向下面的叶结点流动（*继承属性*，inherited attribute）。属性的计算规则可以包含以上讨论的所有动作，如静态语义检查，甚至代码生成。我们将在第14章里更加详尽地定义和讨论属性文法以及它们在构造多遍编译器中的用途。

从第10章到第13章，我们将考查实现各种程序设计语言特性的技术。我们的组织框架将根据我们假定的直接由语法分析器调用的语义例程来定义这些技术。在大多数场合中，这些技术通常是有用的，无论语义动作是直接由语法分析器调用还是由树遍历例程调用。

7.1.2 编译器组织的候选形式

作为讨论编译器组织的基础，考虑图7-3，那个在第1章中首先出现的编译器结构图。图中所示的编译器组件能够以许多不同的方式连接起来，形成本节中所要讨论的编译器组织的选项集。在讨论中，分析（analysis）指编译器通过词法分析器、语法分析器和语义例程的静态语义检查组件所执行的分析任务。类似地，综合（synthesis）指由生成某种程序表示的语义例程组件、优化器和代码生成器所执行的
219 综合任务。我们将考虑六种组织选项。

一遍完成分析和综合

在普通的*一遍编译器*（one-pass compiler）中，分析和综合将在单遍处理中进行。词法分析、语法分析、检查以及到目标机器代码的翻译是交织在一起的。不产生源代码的显式中间表示。这种方法基本上是第2章中Micro编译器所使用的方法，并且它极其类似于在关于语义例程的各章中所使用的模型，主要区别

在于它生成目标机器代码而不是元组。这种改变可以相当透明地通过由语义例程所调用的generate()例程产生相应的目标代码而不是元组来完成。因此generate()例程定义了代码生成器的接口。

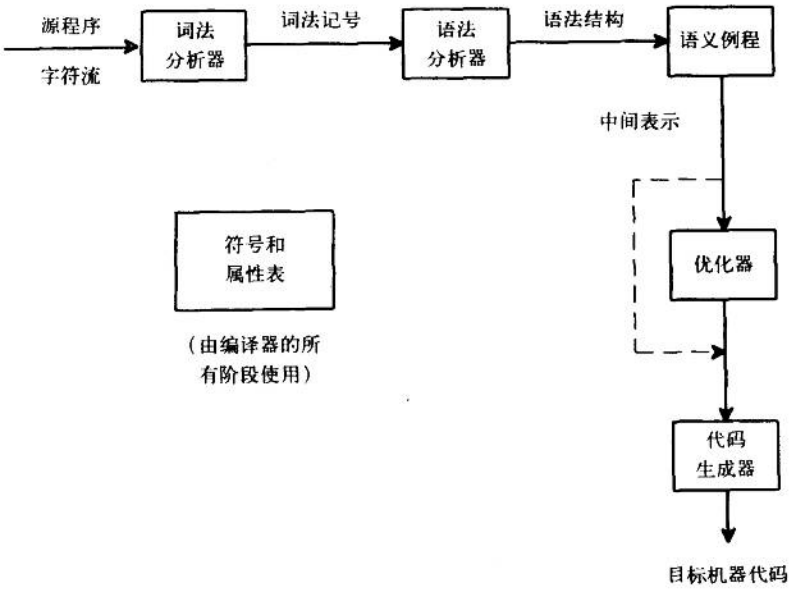


图7-3 语法制导的编译器结构

在代码生成器中，临时变量必须被恰当地映射到寄存器，而且必须选择与元组运算符对应的代码序列。因为代码生成器实际上被限制为一次仅看一条元组，几乎没有重要的优化能够被执行。某些在线寄存器管理有可能消除冗余负担。 220

利用generate()例程作为代码生成器接口使得编译器的分析部分和目标机器保持相对独立。然而，给定的语义例程可能不仅牵涉一个generate()调用，而且它对将要生成的代码的视野比代码生成器更加宽广。例如，分配临时变量的语义例程通常知道一些该临时变量最终是如何使用的。如果目标机器的寄存器集合不是一致的，那么这样的信息对于优化临时变量和寄存器之间的映射可能至关重要。至少，这意味着应当有相应于目标机器寄存器类的临时变量类，以使得代码生成器能够利用原本仅对语义例程可用的信息。这种设计决策使得编译器的分析阶段较少依赖于目标机器，而又使简单的编译器能够生成高质量的代码。

没有独立代码生成器的一遍编译器也十分普遍。这些编译器在语义例程中直接包含了做出所有代码生成决策的代码。某些分配寄存器及处理目标机器寻址模式和指令格式等细节的支撑例程也将被使用，但像指令选择这样的基本任务将直接在语义例程中执行。这些编译器或许能生成稍好一些的代码，或者它们执行效率更高一些，但缺乏目标机器独立性使它们非常难以移植或再目标（retarget）。

一遍编译器加窥孔优化

改善简单的一遍编译器所生成的代码质量的一条有效途径是添加一遍窥孔优化。窥孔优化在第15章中有更为详细的描述。为了此处的讨论，有必要了解这样的优化器在生成的机器代码上执行一遍的情况：它每次仅查看少数几条指令（窥孔优化由此得名），试图通过选择更好的指令或更有效利用寄存器和寻址模式来改进输出的代码。

窥孔优化器在消除由代码生成器的不同调用所生成的连续代码段之间的“毛边”上特别有效。窥孔

优化器的使用允许代码生成器在某种程度上更简单一些，因为它可以更少地关注正在生成的代码所处的上下文。

一遍分析和IR综合加一遍代码生成

一遍分析和IR综合加一遍代码生成是第10~13章中概述的语义例程中所假定的组织方式。显式的中间代码表示需用来作为与代码生成器的接口。IR通常是线性的，就像我们使用的元组。与单遍方法相比，
221 这种组织方式的主要优点是灵活性。代码生成器可以简单地每次查看一个元组，就像在刚刚讨论的带窥孔优化的单遍组织方式中一样。此外，由于存在显式的中间表示，代码生成器也可以检查任意数量的元组以便做出代码生成决策。这些代码生成选项的更多细节在第15章中给出。

灵活性所带来的一个优点是添加IR优化过程的能力以及目标机器与前端间的更大独立性。当然，添加优化器会将编译器变为另一种组织方式——多遍综合。重要的是，显式中间表示的存在允许做出这样的改变而同时对现存的前端和代码生成器的影响较少或者没有影响。

使前端与目标机器保持相对独立是有利的，因为它使得再目标一个编译器尽可能地简单。前端的目标机器无关性是它与代码生成器的接口的目标机器无关性的函数。分离的代码生成过程将接口严格地限制为中间表示所提供的接口。因此，IR的机器无关性决定了一个使用这种组织方式构造的编译器再目标的难度。

多遍分析

下面将要讨论的两种供选择的组织方式是多遍分析和多遍综合。顾名思义，这些并非组织完整编译器的方式，而是用来组织分析或综合组件的方式。它们可以与更简单的综合和分析组件一起使用。

有多种理由来选择多遍分析这种组织方式。历史上，多遍分析用于那些需要适应极度有限地址空间的编译器中。在这种编译器中，词法分析器自身作为一遍，产生表示词法记号流的文件作为对源程序分析的结果。标识符和常量由词法分析器放入符号表中，因此在词法记号文件中仅需要它们最简单的表示。语法分析器也是单独的一遍，产生即将调用的语义动作流或分析树的某种隐含相同信息的线性化表示。如同词法分析器，语法分析器的输出被存储在辅助存储器中。由语法分析器驱动的声明处理和静态语义检查可以在一遍或两遍分析中实现，这依赖于空间限制以及被编译的语言的符号表处理需求。代码或某种IR通常与语义检查在同一遍分析中进行综合。如果使用IR，像元组和后缀式等简单线性形式是最有可能的选择。

除空间限制外，使用某种形式的多遍分析还有其他原因。不需要标识符在使用前声明的语言强制使用这样的组织方式，至少也为了静态语义检查的目的。以上描述的多遍分析的极端形式，及其相关的读写中间文件的开销并非必需的。然而，语义处理必须被划分开来，以使得第一遍分析通过处理所有声明创建符号表，而随后的第二遍进行检查并生成代码或IR。词法分析和语法分析可以和第一遍语义处理相
222 结合，其风格与单遍分析选择极其相似。这一遍分析和第二遍语义处理之间的接口十分简单，它包括由第一遍所创建的符号表，以及在第二遍中即将以它们被语法分析器生成的顺序来执行的语义动作流。词法记号也必须为那些从词法记号值生成语义记录的动作可用。因此，如果第二遍直接生成代码，则以这种方式组织的两遍编译器可能比一遍编译器要略复杂一些。

采取多遍分析的最后一种可能的动因是要利用树结构中间表示的一般性。到目前为止已描述的所有分析组织都仅限于利用在线性化的语法分析树的一遍分析中可用的信息。分析树显式表示的可用性允许利用限制较少的分析技术。例如，第11章中介绍的处理运算符重载的专用技术就是一种为避开没有完整表达式树可用的情况所必须的方法。

除了简单地使更多信息可用外，树结构IR的使用也促进了基于属性文法的更为形式化的语义处理技术的使用。利用这种组织方式，词法分析和语法分析通常组合在一遍分析中，并生成语法树作为其输出。语义信息，即属性，可以利用与语法树的每个结点相关联的规则来计算。这些规则不仅能指定要计算的

语义值，而且还可规定由静态语义检查所强加的限制。对树的一遍或多遍分析可以用来计算属性的值并检查语义的正确性。

显式语法树的可用性还有允许分析和综合完全分离的优点。和先前讨论的将静态语义检查与某些IR或代码综合相混合的技术相比，这里所描述的处理都是纯分析的。这种分离是可能的，因为作为语义分析阶段输入IR的语法树及其属性也扮演着分析和综合之间的IR接口。

多遍综合

正如先前所建议的，多遍综合可以与刚刚讨论的任意一遍或多遍分析方案一同使用。惟一的要求是某些IR必须对驱动综合阶段可用。IR可以是线性的或树结构的。

最简单的多遍综合组织方式是代码生成器和窥孔优化器的组合。回想一下，窥孔优化是试图改进由代码生成器产生的代码的与机器相关的最后一遍处理。该组织方式说明这样一个事实——用于执行多遍综合的组件比多遍分析组件有更少的相互依赖。特别地，代码生成器可以完全不知道窥孔优化器的存在。

通过在代码生成器处理IR之前允许一遍或多遍的优化对IR进行变换，我们可以创建更为复杂的后端。 223
典型地，机器无关的优化首先在IR上执行。如果包含完整的全局优化，则仅该阶段就通常需要多遍。分离的与机器相关的优化阶段可随后执行，或者这种优化可以包含在代码生成阶段里。无论使用什么样的组合，优化都被称作是*变换*（transform）IR，因为它们的输出格式通常与作为其输入的IR相同，而不像其他的编译器阶段那样通常将一种表示*翻译*（translate）成另外的一种表示。这使得后端各组件间没有典型的相互依赖：代码生成器不需要知道优化器是否存在。

如第15章中所讨论的，代码生成器自身可能涉及一遍或多遍处理。简单的代码生成算法在一遍处理中运行得十分充分。然而，如果能多遍处理寄存器的可用性和代码选择之间的相互依赖，则常常能更有效地分配寄存器。进一步，如果一个高级的、与机器无关的IR作为代码生成阶段的输入，可能需要一遍处理将IR翻译为低级的、更加依赖于机器的形式。如果使用了某个基于机器描述的代码生成技术（在第15章中介绍），则这个步骤很可能是必需的。

多语言和多目标编译器

标准化组织方式（通常是多遍处理）与适当的中间表示的结合允许构造*编译器族*（a family of compilers）。一组针对特定语言、使用相同的分析和与机器无关的优化组件而同时却拥有针对不同机器的不同的代码生成器的编译器，是编译器族的一个例子。这类编译器族需要使用与机器相对无关的IR。一些商用的Ada编译器就是以这种方式构造的。它们使用的Diana中间表示是Ada事实上的标准IR，而且是高度语言相关但与机器无关的。使用基于机器描述的代码生成器尤其有利于构造多目标编译器族。

另一类编译器族是多语言族。这类编译器族涉及使用基于语言无关IR（与Dianna相反）的编译器组织方式。针对不同语言的前端都产生相同的IR。公共的综合组件用于生成针对特定目标机器的代码。与语言相关的IR相比，这类编译器族所用的IR通常是更低级的（即，更加面向机器的）。我们说的是“面向机器”而不是“机器相关的”，因为IR可能基于某种虚拟机器——例如简单的栈机器，而不是基于任何实际的硬件。在多语言编译器族中使用更高级的IR也是有可能的；然而，这样的IR可能难以定义，因为它必须足够强大以便充分表示所涉及的所有源语言中的所有结构。就像代码生成器的生成工具协助多目标编译器族的开发那样，前端生成工具支持多语言编译器族的开发。这样的工具包括我们先前讨论的词法分析器和语法分析器的生成器，以及更多的、通常基于属性文法的用于语义分析生成 224
的实用工具。

GNU C编译器GCC（Stallman 1989）使用两种中间形式。第一种是高级的、面向树结构的中间形式。第二种称为RTL，寄存器转移语言（Register Transfer Language），它更多地面向机器。GNU编译器前端将源语言转换为树结构的中间形式，与语言无关的例程再将树转换为RTL，然后，在RTL上执行若干遍与机器无关的优化。最终，RTL被转换为汇编语言，并可选地进行窥孔优化。

7.1.3 一遍编译中的分析、检查和翻译

第2章的Micro编译器被构造为词法分析、语法分析和语义处理（语义检查和代码生成）等阶段交错进行。这些阶段间的这种关系在任何一遍编译器中都是必需的。即使在包含单独的优化和代码生成阶段的编译器中，也仍然希望能一遍生成被后面的阶段使用的中间表示。这种方法有两个主要优点：（1）编译器前端更加简单，因为不需要构造或遍历树的代码；（2）如果不显式地创建整个语法树，则处理一个程序所需的存储空间会更少。自然，它也有一些相应的缺点。没有完整的树表示限制了对每个语义例程直接可用的信息总量。因此，需要某些特殊技术来克服这个限制。

对于语义处理章节（见第10章~第13章）中的例子，我们假定语法分析和语义处理交错于一遍分析中。该假定源于保持我们的方法尽可能简单的愿望。我们所使用的技术可以很容易地被推广以便和程序的显式语法树表示一同工作。

一遍翻译器中的词法分析器和语法分析器之间的关系非常简单。语法分析器需要来自词法分析器的记号流。词法记号可以在需要时由词法分析器利用其内部数据结构和源程序产生。因此语法分析器简单地将词法分析器作为一个子程序来调用并接收词法记号作为调用的结果。就像Micro编译器所展示的，语法分析器/语义关系要复杂得多。出现在驱动语法分析器的产生式中的动作符号导致语法分析器在分析源程序时调用相应的语义例程。考虑下列描述**if**语句的产生式：

```
<statement> → if <expression> #start_if
              then <statement list> end if #finish_if
```

225 该产生式指出对语义例程`start_if()`和`finish_if()`的两次调用将在语法分析器处理**if**语句的适当时刻发生。在编译器所处理的语言中，对每个结构调用哪些语义例程以及何时调用它们都要显式地指定。

由语义例程的调用所产生的语义记录将表示与语法树相应结点关联的属性。每个不同的文法符号，包括终结符和非终结符，都有不同的含有那个符号适当信息的记录。然而，同类符号的每次出现——例如每个ID或<expression>——在其语义记录中存储着完全相同的数据集。如果没有可存储的语义数据，符号也有可能拥有空的语义记录。例如，像“；”这样的符号就不需要语义记录。

尽管语义例程从不显式调用其他语义例程，但它们通过接收语义记录作为参数并生成语义记录作为结果来隐式地相互通信。那些确实拥有关联语义信息的终结符的语义记录可由在第2章Micro编译器中使用的`process_op()`和`process_id()`那样的例程来构造。它们使用由词法分析器所提供的有关词法记号的信息来构造适当的记录。非终结符所对应的语义记录通常由那个在该非终结符的产生式的整个右部被匹配之后调用的语义例程产生。该语义记录由处理产生式右边符号相应记录的语义例程创建。产生式右边符号相应的语义记录被作为参数传递给该语义例程。处理这些记录可以包括生成代码或在符号表中登记信息，以及构造新的语义记录。根据使用的是递归下降还是表驱动的语法分析，语义例程将由语法分析例程或语法分析器驱动程序来调用。在任何一种情况下，都必须提供某种措施在各语义例程的调用之间存储语义记录，因为这些例程不直接调用其他例程。在第2章的Micro编译器中，我们看到语义记录被存储在语法分析例程的局部变量中，直到它们用作语义例程调用的参数。作为选择，表驱动的语法分析器必须依赖于一个显式数据结构——栈，我们称之为**语义栈**（semantic stack）——来存储由语义例程所产生的语义记录，直到它们在后续的调用中被作为参数使用。图7-4举例说明了在语法分析器分析A := B + 1时所调用的语义例程对语义栈的影响。

在使用自顶向下或自底向上的语法分析器交错进行语义处理与语法分析时，可能的语义属性计算被限制为那些在分析树的一次后序遍历中可能进行的属性计算。由两种语法分析器中任意一种所生成的语法分析动作序列都提供了足够的信息来创建分析树，而且它可以被认为是树的线性化。类似地，在语法分析中生成的语义动作调用序列对应在树的后序遍历中结点访问和属性计算的顺序。这些观察准确指出了交错进行语法分析和语义处理的缺点。在进行语法分析时构造显式的语法树允许使用希望支持语义处

理的任意信息流。把这两个阶段交错在一起要求我们的语义例程被设计为仅利用在一次后序遍历中可用的信息即可完成其工作。这样的设计是否可能取决于被编译的语言的定义。 226

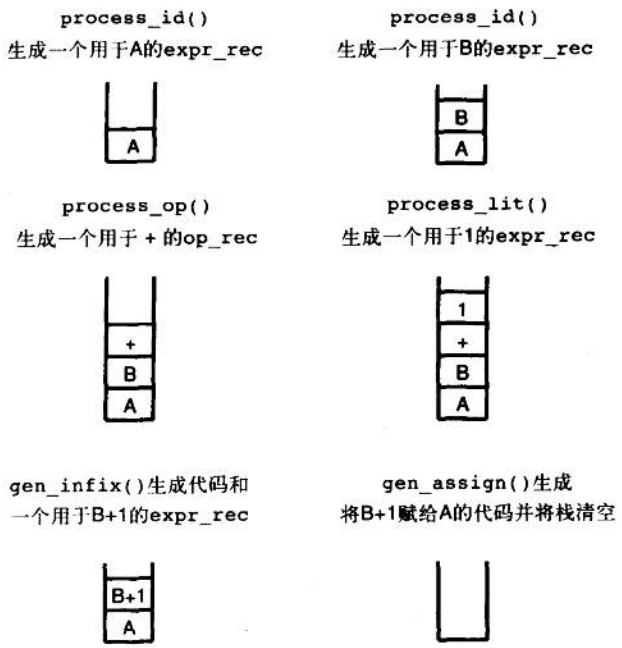


图7-4 语义栈示例（处理A := B + 1）

7.2 语义处理技术

7.2.1 LL分析器和动作符号

在前面一节中介绍的**if**语句产生式说明了在自顶向下的语法分析器中如何使用动作符号来指定何时调用特定的语义例程。动作符号不必包含于每个产生式中（某些非终结符没有与之相关的语义信息），因此，相应的产生式对语义属性的计算没有影响。<statement list>的产生式是这种情况的一个很好的例子。然而，某些产生式，像**if**语句产生式，可能包含多个动作符号。多个动作符号通常出现在控制结构的产生式中，因为像**case**、**if**和**loop**语句这样的结构要求在源代码的多个地方生成代码。 227

通过动作符号来指定语义例程调用能够和LL分析器很好地配合工作，如图5-11中的语法分析例程所说明的那样。动作符号和其他文法符号一样被同等对待，并在预测一个产生式时被压入分析栈。仅需对LL分析算法做一点小的扩展就能处理这种新的语法符号类型。当动作符号到达栈顶时，语法分析器简单地调用相应语义例程而不是试图执行某种语法分析动作。

动作符号与LL分析的良好结合归因于自顶向下分析的预测特性。一个产生式在与其右部对应的符号被分析器处理之前将被选择用来扩展分析树，因为 LL(k)分析器为选择产生式将超前搜索、但并不消耗产生式右部的前k个符号。

7.2.2 LR分析器和动作符号

与LL(k)分析方法不同，LR(k)分析器直到处理了产生式整个右部并超前搜索k个符号之后才决定应

用哪个产生式。由于这种不确定性，在产生式右部符号被语法分析器接受时，动作符号不能用于触发语义例程调用。两个除动作符号不同外，右部符号均相同的产生式可以被同时考虑。只有在语法分析器使用超前搜索符号选择适当的产生式后才可能调用正确的语义例程。

再次考虑先前介绍的**if**语句产生式：

```
<statement> → if <expression> #start_if
              then <statement list> end if #finish_if
```

我们认识到在处理了产生式的整个右部之后才调用`start_if()`太迟了。`start_if()`必须在为<expression>和<statement list>产生的代码之间生成条件跳转指令。推广这种认识，显然动作符号只能被放在用于LR分析器的产生式的最右端。

if语句产生式必须重写为两个独立的产生式，以在语法分析的正确时刻产生语义例程调用。因此，必须引入新的非终结符，它对应着原产生式右部直到并包含第一个动作符号的部分。我们使用下列产生
228 式为LR分析器指定**if**语句：

```
<statement>→ <if head> then <statement list> end if #finish_if
<if head>→ if <expression> #start_if
```

非终结符<if head>有时被称为语义钩子（semantic hook），因为它在文法中的存在对于语法分析器所接受的语言没有影响。它仅用于使语法分析器能够在正确的时机，即在处理**then**后面跟随的<statement list>之前，调用`start_if()`。

即使在产生式右部仅有一个动作符号，如果它不在最右端，则类似的转换也是必须的。例如，考虑Micro中<program>的产生式，它需要应用同样的转换：

```
<program>→ #start begin <statement list> end
```

必须变为

```
<program>          → <program head> begin <statement list> end
<program head>     → #start
```

以使得`start()`在处理<statement list>之前调用。

某些LR分析器生成器，其中最著名的是Yacc，允许在产生式右部的任意位置插入语义动作。刚刚描述的转换由语法分析器生成器在必要时自动执行。

这种文法转换的一个重要的性质是它对语义例程没有任何影响。它们与所使用的语法分析技术完全无关。这种转换及其逆转换使得从LL或LR分析器有可能生成相同的语义例程调用序列。因此，针对两种语法分析器，那些将在第10章到第13章中研究的语义处理技术均可以很好地工作。用于表示它们的文法片段可以被转换以适应任意一种分析技术。

像Yacc那样产生完整的语法分析器而不仅是分析表的语法分析器生成器可以允许在我们给出语义动作符号的位置上插入任意的程序代码。该特性对于小的翻译工作尤其有用，因为它允许语法和语义都在一个地方被指定。对于较大的翻译器，像编译器，这种组合变得十分难以阅读，除非所插入的语义动作被限定为子程序调用。然而，两种生成器的能力几乎相同，其中表生成器更为灵活，因为它们不强迫使
229 用特定的语言书写语义动作。

7.2.3 语义记录表示

如第2章的Micro编译器所说明的那样，语义记录是传给语义例程的参数。由语义例程所访问的记录包含与符号相关联的语义信息，这些符号已经在调用语义例程之前由语法分析器处理。每个语义例程接收一定数量的记录作为参数并可选地产生一个或多个记录表示其结果。

Micro编译器是语义例程调用出现在语法分析例程中的一个例子。在7.2.1节和7.2.2节，我们讨论了

语义例程调用如何由表驱动的语法分析器自动触发。在第2章的示例编译器中，使用了两种不同类型的语义记录：op_rec和expr_rec。无论何时当某个记录需要被暂时存储用于随后的语义例程调用时，在语法分析例程中要声明相应的适当类型的局部变量。当使用表驱动的语法分析器时，需要一个语义栈来暂存语义记录。语义栈的实现通常使用单独的一种语义记录类型。然而，各种文法符号的语义栈条目必须包含多种信息。

当使用支持强类型检查的实现语言时，将不同的信息放入语义栈条目中最好能通过使用变体记录或类型联合来处理。这些特性可将不同类型组合于单独的类型中，这正是构造包含各种文法符号不同信息的语义栈所需要的。使用这些特性的一个特别的优点是各种语义记录版本在声明中显式地明确给出，因此增强了可读性，并允许编译器对使用语义栈记录的代码执行类型检查。

图7-5中的类型声明举例说明了Micro的语义记录定义所使用的方法。首先，重复来自第2章定义不同类型语义记录的类型声明。随后这些声明与枚举变量组成称为semantic_record的结构，它定义语义栈中所有可能的可选条目。（与第2章中一样，expr_rec是一个匿名联合，该结构并非标准C语言的一部分，但我们将在这里及后续章节中大量使用它。）

```
#define MAXIDLEN        33
typedef char string[MAXIDLEN];

typedef struct operator {    /* for operators */
   enum op { PLUS, MINUS } operator;
} op_rec;

/* expression types */
enum expr { IDEXPR, LITERALEXPR, TEMPEXPR };

/* for <primary> and <expression> */
typedef struct expression {
   enum expr kind;
   union {
      string name;    /* for IDEXPR and TEMPEXPR */
      int val;        /* for LITERALEXPR */
   };
} expr_rec;

enum semantic_record_kind { OPREC, EXPRREC };

typedef struct sem_rec {
   enum semantic_record_kind record_kind;
   union {
      op_rec   op_record;      /* OPREC */
      expr_rec expr_record;    /* EXPRREC */
   };
} semantic_record;
```

Figure 7.5 semantic_record Declaration for Micro

图7-5 Micro的semantic_record声明

错误处理

图7-5中的语义记录声明假定任何被设计来返回语义记录的语义例程将把某些有意义的信息放在那些记录当中。当语义错误被检测出来时，该假定不成立。在要求声明的语言的编译器中，例如，对于一个在符号表中查找标识符的语义例程，如果标识符没有找到，或者该标识符的定义不符合其用法，则该例程产生一条出错信息。语义例程通常会产生含有某些从符号表条目中所提取信息的语义记录。然而，当发现错误时，没有这样可用的信息。

试图校正程序错误的编译器可能产生用于后续例程的语义记录信息，但这种方式会有导致报告一长串无意义的、容易使人混淆的出错信息的风险。然而，存在一种可用的简单技术能保证对于语义例程检

测到的每个错误，只会出现一条出错信息。该技术要求定义一种额外的语义记录类型，我们称之为ERROR。它不需要有相应的变体，因为ERROR记录仅警告其他语义例程发生了错误并且不存在有用的信息。因此semantic_record的定义将被改变，如图7-6所示。

```
enum semantic_record_kind { OPREC, EXPRREC, ERROR };

typedef struct sem_rec {
    enum semantic_record_kind record_kind;
    union {
        op_rec   op_record;     /* OPREC */
        expr_rec expr_record;   /* EXPRREC */
        /* empty variant */     /* ERROR */
    };
} semantic_record;
```

图7-6 带有出错选择的Micro的semantic_record声明

任何检测到错误的语义例程必须产生ERROR记录而不是它通常所产生的那些语义记录。对于作为参数收到的记录，每个例程必须在试图使用它们之前对其进行检查，以确定其中是否有ERROR记录。如果它的任何一个参数是ERROR记录，它就可以忽略所有正常处理并简单地返回一个ERROR记录。（如果仅使用除ERROR记录之外的其他参数，某些静态语义检查也许仍然可以进行，但寻求这种可能性将使得语义
231 例程变得极为复杂化。）通过该机制，在检测到一个错误之后，所有语义例程消耗并产生同样数量的语义记录，并且一个出错指示被传播到某个语义例程，通常是在<statement>或<declaration>级的例程，它们不产生语义记录。程序其余部分的编译可以随后正常继续，并且通常仅有单独一条适当的出错信息被生成。（通常仅做进一步的分析，因为为一个含有严重错误的程序生成代码很可能是不值得的。）

7.2.4 实现动作控制的语义栈

在为编译器设计基于栈的语义处理阶段时，一个重要的问题是如何控制入栈、出栈以及将栈中的条目作为参数传递给语义例程。一种方法是使栈成为语义动作例程可直接访问的。利用这种方法，动作例程从栈顶取它们的参数而不是在被调用时显式接收参数。类似地，在去除了参数以后，动作例程所产生的任何语义记录将被直接压入语义栈中。以此方式管理的栈被称为*动作控制的*（action-controlled）语义栈。

语义栈可以实现为像7.2.3节中所定义的semantic_record类型记录的数组，或者动态分配的记录组成的链表。数组方式要求为数组中的每个元素分配语义记录的任何变体所需的最大存储空间。它也必须对栈可能增长的最大深度给出固定限制（除非实现语言支持动态改变数组的大小）。然而，入栈和出
232 栈非常迅速，所需的仅是改变定义栈顶的下标变量。相反，链表方式允许栈几乎无限制地增长，而对于每条单独的条目仅需分配恰好满足其需求的足够空间。另一方面，分配和释放需求使得入栈和出栈操作比利用数组实现时要慢许多。（如果所有记录都像数组实现中那样以最大尺寸分配，并且记录在出栈后被保存并重用，而不是归还给动态存储管理器，则这些操作的效率可以被改善。）如果不使用数组的话，访问除栈顶元素外的其他元素的代价可能更为昂贵。

语义栈实现示例

尽管语义例程总是从语义栈顶取得其参数并将结果放在栈顶，在特定的例程中语义栈不必被作为抽象栈对待。语义例程可以直接访问栈中作为其参数的所有条目，因为它知道它们相对栈顶的位置。这种方法避免了为给变量赋值而将它们弹出栈的复制代价。这些条目仅在例程的末尾被弹出并丢弃。类似地，在那些将被弹出的条目下面的条目也可以被访问，甚至被改变，（在它上面的条目被弹出后）它将留在栈中供以后使用。作为这些考虑的结果，动作控制的语义栈不需要被定义为抽象数据结构。相反，它的实现知识可被普遍用于语义例程中。

图7-7包含Micro语义栈实现的定义，它允许像上面所描述的那样使用语义栈。它使用图7-6中给出的semantic_record类型声明加上来自图7-5的辅助声明。跟在这些类型声明后面的是实际用于实现语义栈的变量——一个数组和一个下标变量的声明。最后是可用于操纵栈的两个宏，push()和pop()。它们并非真正必需的，因为栈的数据结构完全可见，但它们便于改变该数据结构。宏的内容没有给出，因为push()和pop()的实现很简单。 233

```
#define MAXIDLEN      33
typedef char string[MAXIDLEN];

typedef struct operator {   /* for operators */
   enum op { PLUS, MINUS } operator;
} op_rec;

/* expression types */
enum expr { IDEXPR, LITERALEXPR, TEMPEXPR };

/* for <primary> and <expression> */
typedef struct expression {
   enum expr kind;
   union {
      string name;    /* for IDEXPR and TEMPEXPR */
      int val;        /* for LITERALEXPR */
      };
   } expr_rec;

   enum semantic_record_kind { OPREC, EXPRREC, ERROR };

   typedef struct sem_rec {
      enum semantic_record_kind record_kind;
      union {
         op_rec   op_record;   /* OPREC */
         expr_rec expr_record; /* EXPRREC */
         /* empty variant */   /* ERROR */
      };
   } semantic_record;

   #define STACKLIMIT   100
   int top = -1;
   semantic_record sem_stack[STACKLIMIT];

   /*
    * Following are two macros that are not strictly
    * necessary since the semantic stack can be freely
    * manipulated by any routine that has access to this
    * header.  However, they do encapsulate some stack
    * manipulation details.
    */

   /*
    * Pushes entry onto the stack; if there is no room
    * generates a "stack full" fatal error.
    */
   #define push(entry)    { . . . }

   /*
    * Removes number entries from the stack by
    * decrementing top. If stack has less than number
    * entries, generates a "stack empty" fatal error.
    */
   #define pop(number)    { . . . }
```

图7-7 动作控制的语义栈

抽象语义栈

动作控制的语义栈有两个缺点：(1)它们的实现对于动作例程完全可见，因此对实现的改变很困难，

以及(2)它们要求动作例程中包含进行语义栈管理的代码。其中后一点使得动作例程比我们在第2章中所见到的更为复杂（那些例程通过其参数，与编译器的其余部分有非常简单的接口）。这个问题可以用两
234 种方式来解决。如果有适当的语法分析器生成器可用，可以使用分析器控制的语义栈（parser-controlled semantic stack）。该技术在7.2.5节描述。作为选择，还可以使用参数驱动的动作例程，如果它们不是由语法分析器直接调用而是由处理栈操作的中间例程调用。这些中间例程由语法分析器调用。

不论用哪种方法，使用像图7-8a所示的抽象语义栈接口都是合适的。（图7-8b含有相应的实现。）该例程与图7-7的例程不同，其中语义栈的实现被隐藏在实际的过程中，而这些过程在其他地方定义。因此我们获得语义例程和语义栈之间更简洁的接口，但放弃了访问尚在栈中的条目的能力（反正这种能力已经不再需要）。为访问一个条目，必须调用pop()，它将导致栈中条目被复制到由调用例程所指定的某个变量中。这种复制使得抽象栈比起先前给出的语义栈更低效。然而，抽象方法使得从数组实现切换
235 到链表实现很简单，而使用第一个版本的语义栈将会十分复杂。

```
/* Header file to be included by the action routines.
 */
     .
     .
     .
  /* semantic_record and related type
   * declarations go here.
   */
     .
     .
     .
/*
 * Pushes entry onto the stack; if there is
 * no room, generates a "stack full" fatal error
 */
extern void push(const semantic_record entry);
/*
 * Removes number entries from the stack by
 * decrementing top. If stack has less than number
 * entries, generates a "stack empty" fatal error.
 */
extern semantic_record pop(void);
```

a) 接口

```
/* In a separate source file. */
#define STACKLIMIT   100
static int top = -1;
static semantic_record sem_stack[STACKLIMIT];
void push(const semantic_record entry)
{
   if (top >= STACKLIMIT-1)
      fatal_error("semantic stack overflow");
   else
      sem_stack[++top] = entry;
}
semantic_record pop(void)
{
   if (top < 0)
      fatal_error("semantic stack underflow");
   else
      return sem_stack[top--];
}
```

b) 实现

图7-8 抽象语义栈的接口和实现

7.2.5 分析器控制的语义栈

动作控制的语义栈必须忍受以下缺点：即除了它们的其他工作之外，每个语义例程都必须压入和弹出适当的信息。语义栈增长和缩小依赖于动作例程如何编写，这意味着它的改变与分析栈的改变仅有松散的关联。语义例程的设计者必须假定或验证在每个语义例程开始时适当的记录已处于语义栈顶，以及每个例程都使语义栈保持正确的格局。可以通过让语法分析器控制语义栈将这种额外的复杂性从语义例程中删除。

LR分析器控制的语义栈

在匹配产生式右部过程中的任意一点，LR分析器的分析栈包含了与到目前为止匹配的每个符号——终结符和非终结符——对应的条目。为让语法分析器控制语义栈，仅需要在分析栈条目中为语义记录增加空间，或者为语义记录提供一个平行的栈。使用该技术，语义栈中为每个匹配的文法符号均包含了一个相应的条目，当然也包括那些带有空语义记录的符号。当产生式右部相应的文法符号被弹出分析栈时，它们也将被弹出语义栈。必须为每个产生式都指定一个语义动作；该动作必须定义当产生式左部的非终结符入栈时要存储的相应语义记录值。

236

第6章中所描述的Yacc分析器生成器是一个提供这种分析器控制的语义记录栈机制的著名例子。在程序员所编写的作为Yacc规则一部分的动作中，相应于产生式右部符号的语义记录可以通过使用伪变量名$1、$2等来引用。产生式左部非终结符的语义值由分配给它的伪变量$$来定义。如果对一个产生式没有指定动作，Yacc自动生成从$1到$$的赋值。

LL分析器控制的语义栈

LL分析栈包含预测的符号而不是已经分析的符号，因此不可能使用像LR分析器所使用的那种平行的栈机制来管理语义栈。语法分析器在预测非终结符或匹配终结符时在分析栈上执行入栈和出栈操作，但语义栈必须以不同的方式工作，因为与符号有关的语义信息通常在符号匹配之后才被维护。

无论何时一个产生式被预测时，不仅相关的符号（终结符、非终结符和动作符号）被压入分析栈，而且与产生式右部的每个终结符和非终结符有关的新的条目也被压入语义栈。无论何时一个产生式完成时，其右部的语义记录被弹出栈。左部的语义记录留在栈中；刚刚完成的产生式左部是预测它的那条产生式右部的一个成员。事实上，聪明的语义栈管理允许当前产生式的左部和前一个产生式右部的成员（同一个非终结符）使用完全相同的条目。因此，在产生式终止时不必复制任何语义记录。

语法分析器总是维护若干指向语义栈的索引。其中之一，left_index，指向存放当前产生式左部文法符号语义记录的条目。类似地，right_index指向右部第一个元素的条目。产生式右部其他元素的语义记录可以在语义栈的位置right_index+1，right_index+2等处找到。current_index指向右部当前正被展开的元素。最后，top_index指向语义栈顶的第一个自由条目。

为使语法分析器分辨何时一个产生式已经结束，我们使用一种新的分析栈条目。到目前为止，分析栈条目是终结符、非终结符或动作符号。第5章中的lldriver()例程就做了这种假设。我们的新条目被称为*产生式结束符*（end of production），EOP。left_index、right_index、current_index以及top_index的旧值存储在EOP条目中。实际的实现可以为这些值使用单独的栈，但假定它们存储在EOP条目中是很方便的。图7-9给出lldriver()的一个版本，它合并了这些管理语义栈的扩展。

237

当语法分析器控制着语义栈时，语义例程可以通过它们用于输入和输出的语义记录来显式地参数化。除了参数不是一般的semantic_record类型以外，在第2章中的语义例程采用的就是这种形式。我们需要一种记号告诉语法分析器哪些语义栈条目应当作为参数传递给语义例程。为提供这个信息，我们增强了动作符号的表示——使用与Yacc中相同的语法为动作符号添加一个语义记录引用表的后缀。因此$$将指定在left_index位置的语义栈条目，而$1指在right_index处的条目，$2指在right_index+1处的条目，以此类推。

```
void lldriver(void)
{
   int left_index = -1, right_index = -1;
   int current_index, top_index;
   /*
    * Push the Start Symbol onto
    * an empty parse stack.
    */
   push(s);
   /* Initialize the semantic stack. */
   current_index = 0;
   top_index = 1;
   while (! stack_empty() ) {
      /* Let a be the current input token. */
      X = pop();
      if (is_nonterminal(X)
          && T[X][a] = X → Y1 · · · Ym) {
         /* Expand nonterminal */
         Push EOP(left_index, right_index,
            current_index, top_index) on the parse stack;
         Push Ym · · · Y1 on the parse stack;
         left_index = current_index;
         right_index = top_index;
         top_index += m;
         /* m is the number of non-action symbols */
         current_index = right_index;
      } else if (is_terminal(X) && X == a) {
         Place token information from scanner
            in sem_stack[current_index];
         current_index++;
         scanner(& a);    /* Get next token */
      } else if (X == EOP) {
         Restore left_index, right_index, current_index,
            top_index from the EOP symbol;
         /* Move to next symbol in RHS */
         /* of previous production */
         current_index++;
      } else if (is_action_symbol(X))
         Call Semantic Routine corresponding to X;
      else
         /* Process syntax error */
   }
}
```

图7-9 包含语义栈管理的lldriver()

我们也需要一个在分析产生式时能知道语义信息存储在哪里的约定。初始信息被存储在产生式左部的语义记录中。在分析非终结符之前我们将所有必要的语义信息转移到该非终结符的语义记录中，因此在该非终结符被作为左部符号时其语义记录已经准备好。在产生式的整个右部已经完成时，我们将所有作为结果的语义信息存储在左部的语义记录中，因为右部的语义栈即将消失。

238 ~ 239 图7-10再次给出Micro文法，其中带有传给动作例程的栈条目参数规范，它们将用于分析器控制的语义栈。上述文法以标准BNF而不是扩展BNF书写，以便指定作为参数传给动作例程的语义栈条目的位置。

图7-11举例说明了当产生式被LL(1)分析器预测时语义栈的创建过程。语义栈中的条目被标记以它们所表示的文法符号而不是它们所含有的语义记录版本。图7-11a显示在开始符号<system goal>被压入分析栈中之后分析栈和语义栈的状态，而产生式<system goal> → <program> $ #finish和<program> → #start **begin** <statement list> **end**已被预测。图7-11b显示在start()被调用、**begin**被匹配以及<statement list>被展开之后的状态。图7-11 c显示将<statement>展开为赋值语句的状态，而图7-11d显示<ident>和:= 被匹配而且<expression>被展开之后的栈状态。图7-11d中语义栈上<ident>的条目尤其有趣。它在语义上是重要的，其中包含有关赋值目标标识符的信息。其相应的条目在它先前被匹配时就

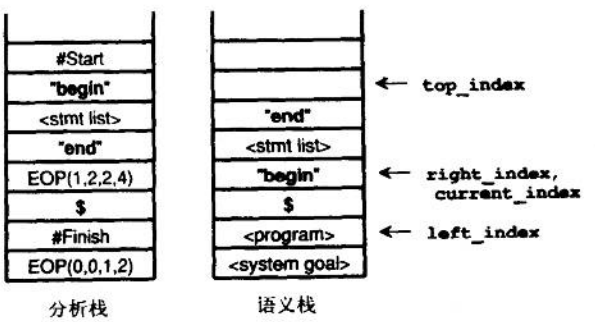
240

241

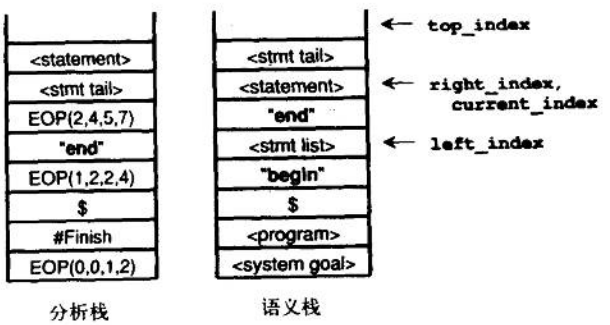
从分析栈中弹出，但语义条目留在栈中供assign()动作例程使用。

```
<program>        → #start begin <statement list> end
<statement list> → <statement> <statement tail>
<statement tail> → <statement> <statement tail>
<statement tail> → λ
<statement>      → <ident> := <expression> ; #assign($1,$3)
<statement>      → read ( <id list> ) ;
<statement>      → write ( <expr list> ) ;
<id list>        → <ident> #read_id($1) <id tail>
<id tail>        → , <ident> #read_id($2) <id tail>
<id tail>        → λ
<expr list>      → <expression> #write_expr($1) <expr tail>
<expr tail>      → , <expression> #write_expr($2) <expr tail>
<expr tail>      → λ
<expression>     → <primary> #copy($1,$2) <primary tail> #copy($2,$$)
<primary tail>   → <add op> <primary> #gen_infix($$,$1,$2,$3)
                     <primary tail> #copy($3,$$)
<primary tail>   → λ
<primary>        → ( <expression> ) #copy($2,$$)
<primary>        → <ident> #copy($1,$$)
<primary>        → INTLITERAL #process_literal($$)
<add op>         → PLUSOP #process_op($$)
<add op>         → MINUSOP #process_op($$)
<ident>          → ID #process_id($$)
<system goal>    → <program> $ #finish
```

图7-10 带参数化动作符号的Micro文法

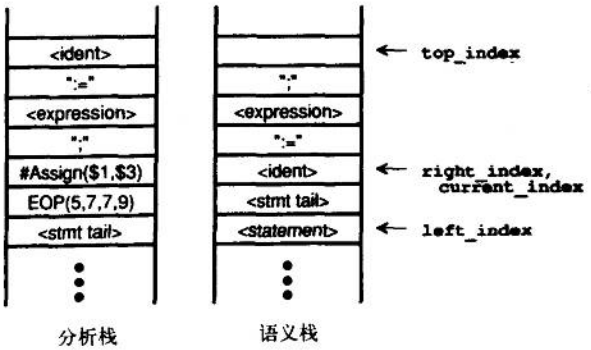


a) 刚刚预测<program> → #start **begin** <statement list> **end**之后的栈

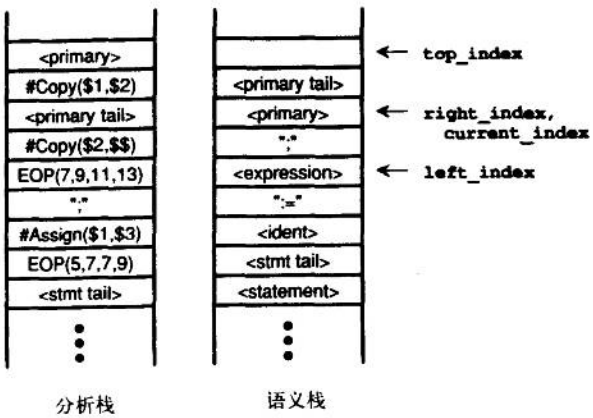


b) 刚刚预测<statement list> → <statement> <statement tail>之后的栈

图 7-11



c) 刚刚预测<statement> → <ident> : = <expression> ; #assign($1, $3)之后的栈



d) 刚刚预测<expression> →<primary> #copy($1, $2) <primary tail> #copy($2, $$)之后的栈

图7-11 （续）

辅助例程lookup()、enter()、check_id()和get_temp()，以及动作例程start()和finish()的代码可以与第2章中的完全相同。在第2章中产生语义记录的语义例程被写成函数；现在它们必须是带有指针参数的过程。所有动作例程必须变为以一般的类型semantic_record为参数。这些改变在图7-12中举例说明。(assert()宏保证被调用的例程带有正确的参数。如果断言失败，编译器应
242 当在发出一条“内部错误”诊断信息后退出。)

```
#include <assert.h>

void process_id(semantic_record *id_record)
{
 /* Declare ID & build corresponding semantic record */
 check_id(token_buffer);
 id_record->record_kind = EXPRREC;
 id_record->expr_record.kind = IDEXPR;
 strcpy(id_record->expr_record.name, token_buffer);
}

void process_literal(semantic_record *id_record)
{
 /*
  * Convert literal to a numeric representation
  * and build semantic record.
```

图7-12 修改后的Micro动作例程

```
    */
    id_record->record_kind = EXPRREC;
    id_record->expr_record.kind = LITERALEXPR;
    sscanf(token_buffer, "%d", &id_record->expr_record.val);
}

void process_op(semantic_record *op)
{
    /* Produce operator descriptor. */
    op->record_kind = OPREC;
    if (current_token == PLUSOP)
        op->op_record.operator = PLUS;
    else
        op->op_record.operator = MINUS;
}

void gen_infix(const semantic_record e1,
               const semantic_record op,
               const semantic_record e2,
               semantic_record *result)
{
    assert(e1.record_kind == EXPRREC);
    assert(op.record_kind = OPREC);
    assert(e2.record_kind == EXPRREC);

    /* Result is an expr_rec with temp variant set. */
    result->record_kind = EXPRREC;
    result->expr_record.kind = TEMPEXPR;

    /*
     * Generate code for infix operation.
     * Get result temp and semantic record for result.
     */
    strcpy(result->expr_record.name, get_temp());
    generate(extract(op), extract(e1), extract(e2),
             result->expr_record.name);
}

void assign(const semantic_record target,
            const semantic_record source)
{
    assert(target.record_kind == EXPRREC);
    assert(target.expr_record.kind = IDEXPR);
    assert(source.record_kind == EXPRREC);

    /* Generate code for assignment. */
    generate("Store", extract(source),
             target.expr_record.name, "");
}

void read_id(const semantic_record in_var)
{
    assert(in_var.record_kind == EXPRREC);
    assert(in_var.expr_record.kind = IDEXPR);

    /* Generate code for read. */
    generate("Read", in_var.expr_record.name,
             "Integer", "");
}

void write_expr(const semantic_record out_expr)
{
    assert(out_expr.record_kind == EXPRREC);

    generate("Write", extract(out_expr),
             "Integer", "");
}

/*
 * Copy information from one part of
 * the Semantic Stack to another
 */
void copy(semantic_record *source,
          semantic_record *dest)
{
    *dest = *source;
}
```

图7-12 （续）

这些语义例程对语义栈的内容不做任何隐含的假定，它们也不以任何方式操纵栈。嵌入在文法中的动作看起来比我们在第2章的Micro文法中所指定的更为复杂，后者可以利用动作控制的语义栈。我们看起来拥有很多的动作，而且这些动作通常只是将语义记录从栈的一个位置复制到另一个位置。实际上，如果我们适当地实现语义栈，所有这些复制都可以通过指针操作来实现，因此除了额外的过程调用外，分析器控制的机制不一定效率低下。它易于正确编程的事实可以胜过它所引入的较少的低效性。

使用LL分析器控制的语义栈的一个明显缺陷是栈可能变得非常大。例如，如果一个程序有100条语句，并且我们使用图7-10中的Micro文法，则由于构造<statement tail>的递归产生式，语义栈将需要至少200个条目。栈的增长是我们所不希望的，因为语义处理不可能使用这么多条目来完成处理。如果语法分析器生成器识别出其语义记录从不使用的非终结符，则这个问题可以被克服。这样的非终结符很容易识别：在它们的任意产生式中都没有动作符号，或者其产生式中的动作符号从不使用$$作参数。在图7-10的文法中，<statement list>和<id list>就是这种非终结符的例子。一旦这些非终结符被鉴别出来，语法分析器生成器将会在它们的每条产生式的最后一个非终结符之前插入一种新的动作符号，前提是那个非终结符后面没有动作符号跟随。这个我们称之为reuse的新符号告诉语法分析器的驱动程序，那些与当前产生式左部一同存储的语义信息是不需要的，因此，随后的非终结符的展开可以复用那个语义栈的相同部分。例如，Micro文法有产生式：

243
~
244

<statement list> → <statement> <statement tail>
<statement tail> → <statement> <statement tail>
<statement tail> → λ

没有和<statement tail>或<statement list>相关联的语义信息。当语法分析器匹配<statement list>时，它可以重用为<statement list>产生式的右部符号所保留的语义栈空间以展开<statement tail>。对先前保留的语义栈空间所做的同样类型的重用对于<statement tail>的递归展开也是合适的。因此reuse符号的插入如下所示：

<statement list> → <statement> #reuse <statement tail>
<statement tail> → <statement> #reuse <statement tail>
<statement tail> → λ

lldriver()主循环中额外处理reuse的情况由下列代码说明：

```
else if (X == reuse) {
   /* Let X be the new top stack symbol. */
   if (T[X][a] == X → Y1 · · · Ym) {
      /* Expand nonterminal */
      Push EOP(left_index, right_index,
         current_index, top_index) on the
         parse stack;
      Push Ym · · · Y1 on the parse stack;
      top_index = right_index + m;
      /* m is the number of nonaction symbols */
      current_index = right_index;
   } else
      /* Process syntax error */
}
```

评价

分析器控制栈与LR分析结合得非常好，很难反对将它们与这样的语法分析器一同使用。这些概念在Yacc中的成功集成说明了该组合的简单性和实用性。而对于LL分析器，情况并不这么清楚。对于LL分析器来说，分析器控制的语义栈和分析栈之间的关系比在LR分析器中更为复杂。因此支持LL分析器使用的分析器控制的栈的语法分析器生成器较为罕见。

在以上两种情况下，分析器控制的栈的优点是语义例程不需要对语义栈的内容作出任何隐含的假定，
245 它们也不以任何方式操纵栈。然而，比起动作控制的语义栈，分析器控制的栈要求更多的动作必须被嵌

入到文法中。额外的动作简单地将语义记录从栈的一个位置复制到另一个位置。如果语义栈被适当地实现，则所有这样的复制都可以通过指针操作执行，但这可能有其他性能上的损失。分析器控制栈使得更易于正确地编写语义例程这一事实可以胜过它所引入的较少的低效性。

最后，使用分析器控制的语义栈限制了栈中的信息如何表示。例如，与单个非终结符（如<identifier list>）对应的列表项，可以利用动作控制的栈以多个栈条目来表示。相反，在分析器控制的栈中每个非终结符都被限制为单个栈条目，需要使用栈外的存储空间来存放与这类非终结符相关的信息。

7.3 中间表示和代码生成

7.3.1 比较中间表示和直接代码生成

在设计语义例程和代码生成器时，必须决定是生成某种中间表示（例如四元组、三元组或树），还是直接生成目标代码。两种选择都有某些优点。利用中间表示的优点包括：

- 目标机器被抽象为某种虚拟机。面向程序设计语言的原语，如Open Block和Call Procedure通常是虚拟机接口的一部分。这种抽象可帮助将高级操作与可能的低级的与机器相关的实现分离开。
- 代码生成和通常的临时变量到寄存器的指派从语义例程中清晰地分离出来，而语义例程仅处理由中间表示给出的抽象。对目标机器的依赖更为谨慎地与代码生成例程隔离。
- 优化可以在中间表示一级完成。这种组织方式使优化在很大程度上与目标机器无关，并使得复杂的优化例程更具可移植性。因为中间表示有意更为抽象和一致，优化例程也就可以更加简单。

直接生成目标代码的优点包括：

- 可以避免将内部表示翻译为目标代码的那个可能的额外分析阶段的开销。
- 对于适当的程序设计语言允许概念上简单的一遍编译模型。

246

如果优化或可移植性是重要的因素，则中间表示拥有真正的价值。如果这些因素并不重要，则简单的直接代码生成更为可取。

隔离和参数化目标机器细节需要极度小心。寻址模式和数据大小、是否存在寄存器、操作的效率等等都是与目标机器相关的。此外，如果最初的设计不易于移植，则为移植编译器所进行的后续修改将会极度困难和昂贵，因为我们必须找出并去除所有的机器相关性。

7.3.2 中间表示的形式

在编译器的历史上，由于多种原因使用了多种中间表示。最简单的可能是后缀表示（postfix notation），它作为算术表达式的无括号表示，在用于编译器之前就在数学上广为人知。正如其名字所暗示的，后缀表示中运算符出现在其所作用的操作数之后。图7-13中的示例说明简单程序设计语言中表达式和语句与其后缀表示的对应关系。

中缀	后缀
a + b	a b +
a + b * c	a b c * +
(a + b) * c	a b + c *
a := b * c + b * d	a b c * b d * + :=

图7-13 后缀表示示例

后缀表示的主要吸引力在于翻译过程简单以及表示简洁。这些因素使得它作为驱动解释器的中间表示尤其有用。事实上，后缀作为优化器或代码生成器的输入并不是特别有效，除非最终的目标机器拥有

栈体系结构。

我们考虑的下一类IR有时被称为三地址码（three-address code）。这类IR是虚拟三地址机器汇编码的有效推广。即，每条“指令”由一个运算符和三个地址组成，其中两个作为操作数，一个作为结果位置。这类IR包括许多稍有不同的表示，其中最重要的是三元组（triple）和四元组（quadruple）。这些表示法和后缀表示的主要区别在于它们包含对结果或中间结果的显式引用，而在后缀表示中这些结果是从栈中隐式引用的。三元组和四元组的区别是，中间值在三元组中由创建它们的元组编号引用，而四元组要求它们给出显式的名字。利用图7-13赋值语句的例子，可得到图7-14所示的三元组和四元组表示。（破折号——用于四元组和三元组中表示未使用的操作数。）

a := b * c + b * d

三元组	四元组
(1) (*, b, c)	(1) (*, b, c, t1)
(2) (*, b, d)	(2) (*, b, d, t2)
(3) (+, (1), (2))	(3) (+, t1, t2, t3)
(4) (:=, (3), a)	(4) (:=, t3, a, —)

图7-14 三地址表示——示例1

三元组有更为简洁的明显优点，但它们的位置依赖性使得涉及移动或删除代码的优化变得相当复杂。这两种形式与后缀表示都编码有相同的信息，但三元组和四元组对于接下来的翻译步骤、目标机器代码
247 的生成来说相当方便。在最简单的情况下，代码生成可以被想像为比宏展开稍复杂一点的过程，变量和临时变量的位置充当每个可能的运算符的宏的参数。

图7-14中的三元组和四元组实际上并不包含通过宏展开完成代码生成的足够信息。作为操作数出现的名字可能代表符号表条目，为发现操作数的类型以及它们的地址，必须引用这些条目。操作数类型必须随后用于确定实现符号运算符（+和*）所需的实际指令。如果代码生成器的简单性是主要的考虑因素，则可以使用稍详细一些的表示。在图7-15的示例中，假定a和d是实型变量，而b和c是整型变量。还假定这是一条Pascal语句，以允许这种类型的混合。

a := b * c + b * d

三元组	四元组
(1) (MULTI, Addr(b), Addr(c))	(1) (MULTI, Addr(b), Addr(c), t1)
(2) (FLOAT, Addr(b), —)	(2) (FLOAT, Addr(b), t2, —)
(3) (MULTF, (2), Addr(d))	(3) (MULTF, t2, Addr(d), t3)
(4) (FLOAT, (1) —)	(4) (FLOAT, t1, t4, —)
(5) (ADDF, (4), (3))	(5) (ADDF, t4, t3, t5)
(6) (:=, (5), Addr(a))	(6) (:=, t5, Addr(a), —)

图7-15 三地址表示——示例2

这两个三地址示例的主要区别在于，第一个示例，和后缀表示一样，实际上是输入在语法上的变换，而第二种表示则更多地是一种基于程序设计语言语义的翻译。无论在哪种情况下，都假定对输入已经完成了静态语义检查，因此代码生成器不需要担心处理像变量未定义和类型相容性错误这样的问题。如果
248 实际已经完成了这样的检查，则不需要额外的工作来识别IR中的特殊运算符，如图7-15所示。

三元组和四元组，像后缀表示一样，基本上是面向表达式的。它们所允许的操作数数量不必兼顾其他用途。例如，在四元组示例中的赋值运算符有一个未使用的操作数。无论使用三元组还是四元组，无条件分支指令都只需要一个操作数。因此，将四元组的概念推广到根据运算符不同而操作数数目可变的

元组（tuple）是有用的。再次回到赋值语句的例子，图7-16说明相应的元组表示。

```
a := b * c + b * d

(1) (MULTI, Addr(b), Addr(c), t1)
(2) (FLOAT, Addr(b), t2)
(3) (MULTF, t2, Addr(d), t3)
(4) (FLOAT, t1, t4)
(5) (ADDF, t4, t3, t5)
(6) (:=, t5, Addr(a))
```

图7-16 元组表示示例

正如本章开头所建议的，基于分析树的结构是最一般的中间表示。出于代码生成的目的，树最早是用来表示单独的语句中的表达式。对于某些目标机器类，有基于树或有向无环图（DAG）表示的表达式最优代码生成算法。（这种算法在第15章中讨论。）近来，表达式树这个概念已经被推广到整个程序的抽象语法树表示。

使用抽象语法树能够有效地统一若干需要某种形式中间表示的编译的不同方面。例如，某些语言要 249
求多遍处理来完成静态语义检查，因为标识符的使用在代码的文本中可以出现在相应声明之前。在这种情况下，第一遍处理程序，包括词法分析和语法分析，能够产生树和符号表。第二遍只是简单的树遍历，将传播属性并完成静态语义检查。作为树变换实现的机器相关优化可以由另一次遍历完成。最终，又一次的树遍历可以直接生成代码或产生一种不同的（更简单的）更适合于特定的与机器相关的代码生成器或优化器的表示。

因此，我们看到抽象语法树作为一种多用途表示，可以用于语义分析、优化和代码生成。事实上，它们的用途更为丰富。许多Ada的实现基于称为Diana的特定的抽象语法表示。Diana不仅用于所描述的编译器中，它还作为独立编译单元的程序库（包和过程）表示，以及作为和其他工具间的公共接口。我们将Diana称为抽象语法表示而不是抽象树表示，因为它实际上是DAG而不是真正的树。抽象语法树在第14章中进一步讨论。

7.3.3 一个元组语言

第10章~第13章中概述的语义例程将生成元组作为中间表示。在此我们定义用于这些章节的元组语言。图7-17中所列运算符的元组包括操作数和由

```
RESULT := ARG1 OP ARG2
```

所定义的标准解释。关系和逻辑运算符返回1表示真，0表示假。

```
ADDI  ADDF  SUBI  SUBF  MULTI  MULTF  DIVI
DIVF  MOD   REM   EXPI  EXPF   AND    OR
XOR   EQ    NE    GT    GE     LT     LE
```

图7-17 标准三地址元组运算符

在图7-18中列出的运算符的元组有如图所定义的特殊的解释以及不同数目的操作数。 250

作为如何使用这些元组的示例，考虑图7-19中的程序和相应的元组。为简单起见，该示例中假定有读写整数的元组运算符。

练习

1. 在7.1节中对语法制导翻译的讨论基于这样的假定：抽象语法树可以从语法分析器在分析一个程序时

所生成的分析动作序列构造出来。抽象语法树不是字面分析树，它包含相应分析树的“语义上有用的”细节。

(a) 给定一个从LL或LR分析器生成的动作序列构造分析树的算法。

(b) 说明如何修改该算法以产生抽象语法树。

UMINUS	ARG2 : = −ARG1
NOTA	ARG2 : = not ARG1
ASSIGN	ARG3 : = ARG1, 大小是ARG2
FLOAT	ARG2 : = FLOAT(ARG1) [ARG1为整数形式]
ADDRESS	ARG2 : = ARG1的地址
RANGETEST	如果ARG3 < ARG1或者ARG3 > ARG2，则中止执行
LABEL	ARG1用于为下一条元组加标签
JUMP	跳转到标记为ARG1的元组
JUMP0	如果ARG1 = 0则跳转到ARG2
JUMP1	如果ARG1 = 1则跳转到ARG2
CASEJUMP	ARG1是case选择器表达式
CASELABEL	ARG1是case语句标签
CASERANGE	ARG1是标签区间的下界
	ARG2是标签区间的上界
CASEEND	不带参数，case语句结束符
PROCENTRY	进入处于嵌套级ARG1的子程序
PROCEXIT	退出处于嵌套级ARG1的子程序
STARTCALL	ARG1是引用活动记录的临时变量
REFPARAM	ARG1是实参
	ARG2是参数偏移
	ARG3是活动记录的引用
COPYIN	ARG1是实参
	ARG2是参数偏移
	ARG3是活动记录的引用
COPYOUT	ARG1是实参
	ARG2是参数偏移
	ARG3是活动记录的引用
COPYINOUT	ARG1是实参
	ARG2是参数偏移
	ARG3是活动记录的引用
PROCJUMP	ARG1是子程序起始地址（标签）
	ARG2是活动记录的引用

图7-18 元组运算符的特别解释

```
begin                          (READI, A)
  read(A,B);                   (READI, B)
  if A > B then                (GT, A, B, t1)
    C := A + 5;                (JUMP0, t1, L1)
  else                         (ADDI, A, 5, C)
    C := B + 5;                (JUMP, L2)
  end if;                      (LABEL, L1)
  write(2 * (C - 1));          (ADDI, B, 5, C)
end                            (LABEL, L2)
                               (SUBI, C, 1, t2)
                               (MULTI, 2, t2, t3)
                               (WRITEI, t3)
```

图7-19 元组示例

251

2. 根据7.1.2节中对编译器组织方式的讨论，检查你所熟悉的一个程序设计语言。解释该语言的特定性质将如何使得这些组织方式非常适合或不适合编译该语言。

3. 7.2.3节中的语义错误处理的讨论描述了每个语义例程处理它的一个或多个输入可能是ERROR记录时

所执行的标准动作。概述一个代码预处理器的算法，它取一个不含错误处理代码的语义例程作为参数，并为之添加适当的代码以处理ERROR记录。

4. 将语义栈实现为数组的一个明显缺点是存在因数组的固定大小而引起栈溢出的可能性。尽管有这样的缺点，数组还是比链表更常用于语义栈的实现，这是因为它们简单，以及push()和pop()操作比使用动态分配的语义记录表更为高效。设计另一种语义栈的实现，它可处理任意数量元素的栈，但只要栈中的记录数小于某个固定的值，其效率就接近于数组。以分析或经验来比较你的实现与数组实现的性能。
5. 在图7-10 Micro的分析器控制的栈文法中添加一条7.1.3节中的**if**语句产生式。下列信息应当作为参数化动作符号的基础：
   (a) start_if()动作例程要求与<expression>关联的语义记录作为输入并将信息留在与**then**相关联的语义记录中。
   (b) finish_if()使用start_if()的输出作为输入，并且不产生语义记录。
   再添加一条包含**else**部分的**if**语句产生式。此时，我们将不得不引入一个新的动作例程，它使用 252
   start_if()的输出作为输入，并把输出留给finish_if()。
6. 设计一个算法重写带有内部动作符号的产生式，以使它们可用于LR分析器。（参见7.2.2节。）
7. 利用图7-10中的产生式，在下列Micro程序被编译时跟踪由LL或LR分析器所驱动的分析器控制的语义栈：

```
begin
  A := 5;
  B := A - 2;
  C := 1 - (A + B);
end
```

8. 将练习7的程序翻译为7.3.2节中的后缀式、三元组和元组。 253

第8章 符 号 表

符号表（symbol table）是一种将值或属性与名字相关联的机制，当我们使用这些值时，可借助它们的名字来访问。因为这些属性代表着与之关联的名字的含义（或语义），所以符号表有时也称为字典（dictionary）。符号表是编译器必不可少的组成部分，因为每个名字的定义仅能在程序中某个惟一的地方出现——即它的声明点，而该名字的使用却可出现在程序文本中的许多地方。每次使用一个名字时，符号表提供到在处理该名字声明时所收集信息的访问。即使在像FORTRAN那样不要求显式声明的语言里，名字的第一次出现也充当着声明的角色而在符号表中为该名字建立相应条目。该名字的首次出现同样是如此构造的符号条目的立即使用者。无论是在单遍还是在多遍组织的编译器中，符号表总是语义处理的集成部分。

8.1 符号表接口

对于符号表，我们感兴趣的有两个地方：其一是那些与符号表相关且在编译器其他组件中均可见的操作；其二是那些操作的实现方式。这一章里主要讨论实现问题，但在描述操作的实现之前，我们首先要考虑一个符号表的抽象定义。我们的符号表接口可以通过图8-1中的定义与声明来定义。习惯上，这些定义与声明将被组织并存放到一个头文件中以定义符号表“包”。

```
typedef char string[MAXSTRING];

typedef struct symtab {
    . . .
} *symbol_table;   /* a pointer */

typedef struct id_entry {
    . . .
} id_entry;

/* Create a new (empty) symbol table. */
extern symbol_table create(void);

/* Remove all entries in table and destroy it. */
extern void destroy(symbol_table table);

/*
 * Enter name in table; return a reference to the
 * entry corresponding to name and a flag to
 * indicate whether the name was already present.
 */
extern void enter(symbol_table table,
                  const string name,
                  id_entry *entry,
                  boolean *present);

/*
 * Search for name in table; return a reference to
 * the entry corresponding to name (if there is one)
 * and a flag to indicate whether the name was present.
 */
extern void find(const symbol_table table,
                 const string name,
```

图8-1 符号表接口

```
                    id_entry *entry,
                    boolean *present);

/* Associate the attrs record with entry. */
extern void set_attributes(id_entry *entry,
                           const attributes *attrs);
/* Get the attributes record associated with entry. */
extern void get_attributes(const id_entry entry,
                           attributes *attrs);
```

255

图8-1 （续）

该接口提供了符号表的抽象视图，即它未明确指出符号表是如何实现的。甚至也未指明将名字和属性相关联的方法。此外，该接口支持存放任意数目的符号表。类型attributes的定义见第10章。

8.2 基本实现技术

在符号表的实现中，我们首先考虑的是例程enter()和find()将如何存放和查找名字。根据我们所希望容纳名字数量（的多少）以及所期望获得的性能（的不同），符号表有多种可能的实现方式：

• 无序表

无序表可能是最简单的存储方式。所需的数据结构仅仅是数组，其插入操作可通过将新名字放入下一个可用的位置来完成。我们也可以使用链表避免因数组大小固定而带来的种种限制。查找操作可以简单地使用迭代算法来进行，但可能会很慢，除非符号表中包含不超过20个左右的条目。

• 有序表

如果数组中的名字保持有序，那么可使用二分搜索法进行查找，n个条目所需的查找时间为O(log(n))。然而，这要求每个新条目都必须在合适的位置上插入。通常，在一个有序数组中进行插入相对较昂贵。于是，我们常常在事先已知整个名字表的情况下才会使用有序表。有序表非常适于存放保留字表或汇编操作码表。

• 二叉搜索树

二叉搜索树的设计兼顾了链式型数据结构的大小灵活及插入效率，其搜索速度可由二分搜索法保证。平均而言，在由随机输入组成的二叉搜索树中添加或搜索名字所需时间为O(log(n))。然而，对符号表来说，此平均情况性能却不能保证，因为程序中使用的标识符肯定不是随机出现的。二叉搜索树的优势之一是由于它们简单且广为人知的实现。它们的这种简单性以及人们对其较好的平均情况性能的认可，使得二叉搜索树成为一种受欢迎的符号表实现技术。二叉搜索树将在8.2.1节中讨论。

256

• 哈希表

在产品级编译器和其他系统软件中，哈希表可能是最常见的符号表实现方法。在表足够大，并有好的哈希函数和适当的冲突处理技术时，无论表中有多少条目，查找工作可在常数时间（constant time）内完成。哈希表将在8.2.2节中详细讨论。

8.2.1 二叉搜索树

我们可在任何有关数据结构的书中找到简单二叉搜索树的实现算法，因此这里不再赘述。我们所关注的是采用二叉搜索树的符号表实现能确保何种可接受的性能。如果二叉树是完美平衡的，那么所期望的时间是O(log(n))。如前所述，由随机输入构建的树也同样有与树中项目数的对数成正比的期望时间，尽管它的平均搜索时间大约会比一棵平衡树的要多38%（Knuth 1973）。虽然如此，最差情况下的性能却是O(n)，然而这种最差情况却不大可能实际发生。例如，以字母序添加名字(A, B, C, D, E)导致实际上是线性表的“树”，甚至看似随机的名字序列也能产生类似的结果（例如，A, E, B, D, C）。

这个问题可以通过使用保持树近似平衡的插入算法而得到解决（Knuth 1973, p. 451）。此举对插入的效率影响不大，但它却使实现显著地复杂化。树平衡算法的基本思想是使以某个结点为根的每一棵子树的高度保持在它兄弟子树的高度内。而全部子树在插入操作使结点失去平衡时将移至不同的根结点处。这种通过移动子树而非个别结点完成的重新平衡方法将使插入代价保持在O(log(n))。

利用二叉树实现符号表的一个明显优势在于它的空间开销（为存储定义树的指针）直接和树中结点数成正比。相比而言，无论已添加了多少名字，哈希表都有着固定的空间开销（哈希表自身的存储）。8.3节中讨论的一种实现技术将使用多个符号表而非单个的全局符号表来表示各种程序组件。树作为这种实现方式的基础有着明显的优势。

8.2.2 哈希表

哈希函数

哈希表的中心思想是把在一个较大的名字空间中的每一个可能添加到符号表的名字映射到哈希表中的一个固定的位置上。此映射由*哈希函数*（hash function）来完成。 257

通常，我们假定哈希函数有如下特征：

- h(n)只依赖于n。
- h可被快速计算。
- h将均匀和随机地将名字映射为哈希地址。也就是说，所有的哈希地址被映射的概率均等，且相似的名字不会聚集在相同的哈希地址中。

某些哈希函数将名字看作字的序列，每个字含若干个字符。比一个字要长的名字被折叠为一个字，其做法通常是采用异或操作，或者将两个n位字长的值相乘再取乘积的中间n位。然后哈希值可由模m的余数获得，其中m是哈希表所含条目数。注意，如果m等于2^b，那么该除法将分离出最右边的b位。因此，应当避免这样大小的表。

计算哈希值的另外一种办法是逐个字符地处理，就像词法记号扫描那样。这类简单的哈希函数包括$(c_1 + c_2 + \cdots + c_n)$ **mod** m或$(c_1 * c_2 * \cdots * c_n)$ **mod** m，其中$c_1, c_2, \cdots, c_n$是组成单词的记号。此时还需采取措施以避免或处理计算中出现的溢出。UW-Pascal编译器使用一个更简单的哈希函数：$h(c_1, c_2, \cdots, c_n) = (c_1 * c_n)$ **mod** m，即只使用第一个和最后一个字符。尽管那些使用单词记号中所有字符的函数能做得更好且代价也不那么昂贵，但这个非常简单的哈希函数看起来工作得也不错。

解决冲突

因为可能添加到符号表中的名字的数量通常远比哈希地址多，所以*冲突*（collision）在所难免。即，对名字n_1和n_2（$n_1 \neq n_2$），有$h(n_1) = h(n_2)$。当这样的冲突发生时，我们可采用以下多种冲突处理技术：

- 线性解决方法。

如果位置h(n)被占用，那么将尝试(h(n)+1) **mod** m，(h(n)+2) **mod** m，等等。如果表中还有空闲位置，那它们最终将被找到。该方法的主要问题在于在填表时容易形成较长的搜索链。

- 加哈希的再哈希法（Add-the-Hash Rehash）。

如果h(n)被占用，那么尝试(2*h(n)) **mod** m，(3*h(n)) **mod** m，等等。这有助于防止较长的搜索链，但如果要尝试所有的哈希位置，则m必须是素数。

- 二次再哈希法（Quadratic Rehash）。

如果h(n)被占用，那么尝试(h(n)+1**2) **mod** m，(h(n)+2**2) **mod** m，等等。

- 链式冲突解决法。

名字根本不在哈希表中存放，相反，表中存放的是具有相同哈希值的名字的记录链表。哈希表自身只存放该链表的表头。即，我们可以有如图8-2所示的哈希表组织形式。 258

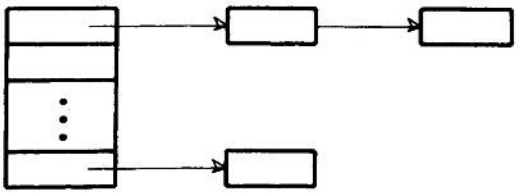


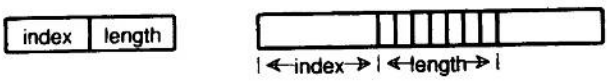
图8-2 带有链式冲突解决法的哈希表

链式冲突解决方法有许多地方吸引我们。首先，它减少了哈希表自身的空间开销，每个条目仅需存放一个指针的空间。其次，它不会像其他冲突解决方法那样在所有哈希表条目填满时导致灾难性的失败，这里假定用于名字记录的空间是动态分配的。实际上，给定一个均匀的哈希函数，如果我们有n个名字和一张大小为m的表，那么查找一个名字的平均（或期望）时间正比于1 + n/2m，添加一个名字的时间则正比于2 + n/m。如果使用的哈希表大小在50至100之间，那么对于除大型程序之外的其他所有程序而言，这些搜索和添加时间基本上都是常量。即使名字的数量大大超出表的大小，我们仍可以获得有n/m个项目的线性表的效率。甚至还可以通过将每条链组织为二叉搜索树而不是线性表的形式来进一步改善（平均）性能。最后，链式冲突解决方法使我们很容易删除个别名字，而其他的哈希技术却不行。由于以上这些优点，采用链式的冲突解决方法成为最受欢迎的哈希表组织形式。

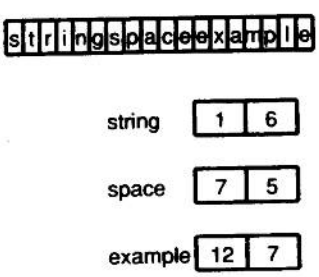
8.2.3 串空间数组

由于添加到符号表里的串的长度变化很大，因而导致串的存储效率相当得低。特别地，如果每个符号表条目包含一个名字域用来表示该条目所对应的实际名字，那么我们将不得不给这个域分配足够的空间以容纳最长可能的串。如果允许名字长度超过8个或10个字符，那么将浪费大量的空间，因为许多名字相对较短。

减少这种浪费的一种方法是使用通常称为串空间（string space）的字符数组来存放所有的名字。使用这种技术，我们将在串空间里存放串的长度和索引以取代把名字本身存放于符号表条目中。这样，如果我们存放长度k和索引i，那么名字将实际存放在串空间的位置i，i+1，…，i+k-1，如图8-3所示。



a) 串空间的图解表示



b) 串和描述符示例

图8-3 一级串空间

任何给定的串仅需在串空间中出现一次。在添加某个串之前，我们首先计算它的哈希值并在合适的哈希链上所有现存条目中检查是否已有该串。检查条目时仅需对那些含有相同串长的条目进行字符比较即可。如果在哈希链上未发现该串，我们才将它添加到串空间的最左空闲位置上。 259

除非首次确定某串不在串空间里，否则不应该将此串添加到串空间里，通常是由符号表例程而不是词法分析器来负责维护串空间的。这种职责分配在具有块结构的语言里十分必要，其中一个名字可在多个作用域中使用（或定义）。单独的作用域可以表示在不同的符号表中以及不同的串空间里，所有这些均由符号表例程来管理。如果词法扫描、语法分析和语义分析依次进行，那么通常从词法分析器识别名字时，即可将该名字“浮动”至串空间顶部直到采取合适的动作为止。如果该名字即将被添加到串空间里，那么它可以呆在它原来所在的地方；否则它将被删除，其占用的空间可被再次使用。这种删除很简单，只要该名字仍在串空间顶部即可，因此相应语义例程的设计必须在向串空间添加任何其他串之前解析该名字的使用情况。

如果串空间被实现为固定大小的数组，那么它的长度选择将是件困难的事情。如果选择得太小，则在存储名字时很容易造成空间耗尽。如果过大的话，又容易浪费空间，而这一点正是我们在使用串空间时应尽量避免的。无论如何，采用空间的动态分配仍不失为一种有效的解决方案。我们可将串空间分配为容纳500或1000个字符的空间段并将原串空间索引看作两级索引。特别地，如果每个段为s个字符长，索引i则表示段（i **div** s）中的位置（i **mod** s）。（参见图8-4的说明。）一般地，我们仅在当前段填满后才为新的段分配空间。 260

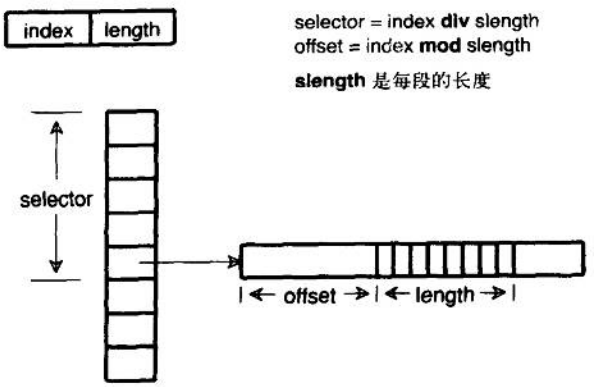


图8-4 分段的串空间

分段的串空间受限于段指针数组的大小。一般讲来，这种限制并不是很严重的问题；例如，能容纳50个指针的数组，每个指针指向含1000字符的段，总计将提供50K字符的串空间。我们甚至可以通过使用（段指针，偏移，长度）三元组确定串而省去指针数组来避免这种限制。但此三元组在某种程度上比（索引，长度）的表示要大一些，因此就适中的串空间需求的平均情况而言前者的代价要略高一些。像C语言那样允许字符指针的语言可以使用（指针，长度）的表示，或使用一个指针指向动态分配的串拷贝，如果其中有特殊的终止字符被用来界定串的话。当然，一种能有效实现动态串的实现语言将会是更好的选择。

8.3 块结构符号表

大多数程序设计语言沿用Algol 60引入的概念允许有名字作用域（name scope）的嵌套。名字作用域通常被定义为由诸如子程序、包或最初的基本块等程序单元所包围的程序文本。自Algol 60以后 261

设计的大多数语言中，这些程序单元可以相互在其中定义，这就使得名字作用域得以嵌套。允许名字作用域嵌套的语言有时也称为*块结构语言*（block-structured language）。

任意一行程序可被一个或多个定义名字作用域的程序单元所包含。这其中由*最内层*（innermost）单元定义的名字作用域称为*当前作用域*（current scope）。由当前作用域和任何外层包围程序单元所定义的名字作用域称为*开放式作用域*（open scope）。而不包含该正文行的程序单元所定义的名字作用域称为*封闭式作用域*（closed scope）。根据这些定义，*当前*、*开放式*及*封闭式*作用域都不是作用域的固有属性；它们均是相对于程序中某个特定的点而定义的。

常用的*可见性规则*（visibility rule）集合给出了在众多作用域中出现的名字的解释：

- 在程序正文中的任何一点，只能访问在当前作用域以及在包含它的开放式作用域中声明的名字。
- 如果某个名字在不止一个开放式作用域中声明，那么离名字引用最近的*最内层*的声明将用来解释该名字。
- 只能在当前作用域中进行新的声明。

这些规则的一个明显的含义是，当一个作用域封闭时，其中的声明就成为不可访问的。例如，考虑图8-5中的程序片段。

```
declare
   H,A,L : Integer;
begin
      declare
         X,Y : Real;
      begin
         .
         .
         .
      end;

      declare
         A,C,M : Character;
      begin
         .
         .
         -- Current position in program <=
         .
         .
      end;

end;
```

图8-5 嵌套作用域示例

在图8-5中所指示的位置上，声明A（为字符型）、C、H、L和M均可见。而X和Y不可见，因为它们在其中定义的作用域已关闭。

与子程序关联的参数名字局部于该子程序体。然而，子程序自身的名字却定义在包含该子程序声明的作用域中。（如果认为子程序的名字局部于它的程序体，那么该子程序将永不被调用！）

有两种常见的方法可以实现块结构符号表：针对每个作用域有一个符号表；或者有一个单一的全局符号表。

每作用域一个符号表

如果为每个作用域创建单独的符号表，则必须采取某些措施确保搜索过程能找到由嵌套作用域定义的名字。因为名字作用域是按后进先出的方式打开和关闭的，因此，栈是组织这种搜索的合适手段。这样，将维护一个符号表*作用域栈*（scope stack），栈上每个条目对应一个开放式名字作用域。最内层的作用域位于栈顶，它的最近包含作用域位于次栈顶，等等。当打开一个新的作用域时，将创建一个新的符

号表并压入栈中；在关闭一个作用域时，将弹出栈顶的符号表。由作用域栈弹出的作用域在一遍编译器中可以被销毁，但在多遍编译器中该作用域必须作为定义它的程序单元的属性而被保存下来以便在随后的编译阶段用来查找名字。于是，对于图8-5中的程序代码，我们可以得到如图8-6所示的符号表布局。 262

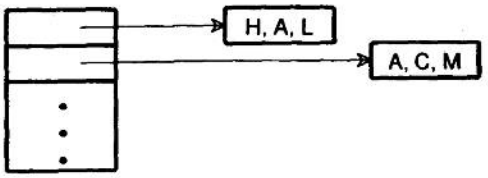


图8-6 嵌套作用域的单独符号表实现

为查找名字，我们首先在栈顶符号表中搜索，然后再从次栈顶查找，等等，搜索过程将持续到名字被找到或栈被耗尽为止。定义在图8-7中的例程可配合图8-1中的符号表接口用来维护和使用作用域栈。 263 sts_push()和sts_pop()是维护栈的例程。sts_pop()返回从栈顶取出的符号表，这样我们就可以根据编译器的组织形式决定是保存还是销毁那个符号表。sts_current_scope()是用来获取最内层作用域的引用以便它在例程enter()被调用时可用。sts_find()使用find()沿作用域栈向下搜索name。

```
extern void sts_push(const symbol_table table);
extern symbol_table sts_pop(void);
extern symbol_table sts_current_scope(void);

/*
 * Search stack of tables for name; return a
 * reference to the entry corresponding to name
 * (if there is one) and a flag to indicate
 * whether the name was present.
 */
extern void sts_find(const string name,
                     id_entry *entry,
                     boolean *present);
```

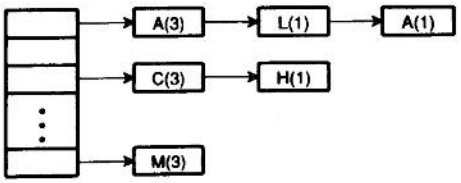
图8-7 作用域栈例程

使用这种作用域栈方法的一个不足之处是，在找到一个名字之前我们可能需要在多个不同的符号表中进行搜索。例如，一个全局定义的名字要求搜索栈中所有的符号表。如此搜索的代价包括引用栈中的每个符号表，加上引用每个表中的每一个条目，再加上搜索每一个相关的链。这种栈搜索的代价因程序而异，主要取决于程序中非局部名字引用的数量和开放式作用域的嵌套深度。

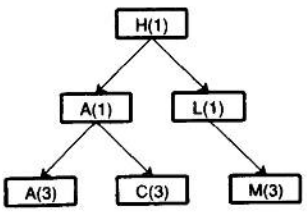
在采用哈希表的实现时出现的另一个问题要归于为每个作用域分配存放哈希表的存储块的大小。如果每张表太大，则会因为多数作用域仅包含了少量名字的定义而造成大量的空间浪费。如果表太小，则在像最外层作用域那样包含较多名字定义的作用域中搜索标识符时，速度会因为链太长而减慢。在内层作用域中维护一张较小的表是可能的（尽管会更为复杂），因为那里往往不大可能出现大量的声明。当然，在采用二叉搜索树的实现时不存在这种问题，因为每棵二叉树没有固定的开销。

单一符号表

使用单一符号表时，所有嵌套作用域中的所有名字均出现在单独一张表中。每个名字作用域被赋予一个惟一的作用域号（scope number）。名字可以在这张符号表中出现多次，只要每次重复的出现有不同的作用域号即可。图8-8a显示了为图8-5中的程序所构造的采用单个哈希表的一种可能的全局符号表结构；图8-8b显示了相应的二叉搜索树。 264



a) 哈希表实现



b) 二叉树实现

图8-8 嵌套作用域的全局实现

使用图8-8a中的哈希表实现，新的名字将被添加到链的前端。此时名字搜索很容易，在合适的哈希链上目标名字的首次出现即为所需条目。在作用域被关闭时，所有和这个正被关闭的作用域有相同作用域号的条目将被从每一条链中删除。此时，在每条链上不需要检查在第一个作用域号不匹配的条目之后的那些条目。

由图8-8b可看出，由单个二叉树实现的多个作用域其运作不如哈希表那样方便。因为二叉树上的插入操作要在叶结点处完成，一个名字的搜索在第一次匹配后必须继续朝叶结点方向进行并返回最后一次所发现的匹配条目。类似地，当作用域被关闭时，要对整棵树进行遍历并删除那些和正被关闭的作用域有相同作用域号的叶结点和子树。

265

如果要实现单一的全局符号表，我们可以通过把类型symbol_table实现为作用域号而将作用域概念合并到符号表的接口中，而此举几乎不需要更改该接口。然后，create()操作只是简单地返回一个作用域号，destroy()则删除标记有那个作用域号的所有条目。惟一要做修改的是find()的接口，它的参数table将被删除。

作为选择，我们可以向我们的“包”中添加open_scope()和close_scope()操作；其中每一个操作都将以一个符号表为参数并内部处理作用域号。这后一种方法的优点是，我们的符号表接口可以保持足够的一般性以支持多个作用域有多个符号表的情况。

使用哈希表的全局符号表的方法在某种程度上比每个作用域一张表的方法更难于实现，但由于它搜索的仅是一张表，因此也就提供了较快的搜索速度。此外，它还能更有效地利用空间，因为它仅有一张表头而非每个作用域一个。尽管如此，此方法中为每个条目添加作用域号所需的额外空间将部分抵消这些节省的空间。

采用二叉树实现的全局符号表方法并不吸引人。因为在内层作用域中搜索名字将花费更长的时间。在到达内层作用域的条目时必须先行遍历与外层作用域对应的树。关闭一个作用域的代价也很昂贵，因为需要遍历整棵树以寻找要删除的条目。此外，也没有空间节省来补偿存储作用域号所需的额外空间。简言之，我们不推荐使用二叉搜索树来实现单一的全局符号表。

在早期的编译器中（在动态存储管理普及之前），使用了这种单一符号表的一种变形。在此变形中，由于所有新的定义必须在最内层作用域中做出，因此栈被用来容纳符号表的定义。在一个作用域中做出

的所有定义（在栈中）均相连，且最内层中的定义位于此栈的顶部。对每个作用域都将保持一个“高水位”标记，这样在关闭一个嵌套作用域时，栈就可以清退到那个标记位置。

在此变形中可使用多种搜索技术，但由于动态数据结构的广泛使用（链式冲突解决法或二叉搜索树），它在现代编译器中已不常用。然而有一点却保留了下来。给定作用域中所有添入串空间的名字均是相连的，因此可以使用类似的方法在作用域关闭时回收被作用域占用的串空间。（无论将符号表实现为多个单独的表还是一个全局表，都可使用此项技术。）

单一符号表方法真正最适用的地方是一遍编译器，在那里所有关于作用域的信息在编译器到达那个作用域末端时都将被抛弃。典型地，多遍编译器在某一遍中建立符号表，而在后续的某遍或多遍处理中使用该符号表。如果当作用域关闭时不删除条目，那么单一全局符号表的实现就变得非常复杂且潜在地效率低下。因此，多遍编译器通常为每一个作用域均采用单独的符号表。 266

8.4 块结构符号表的扩展

由Algol 60引入并于前面章节描述的块结构名字作用域规则构成了大多数现代程序设计语言的基本作用域规则。然而，多数语言以各种方式扩展了这些标准的作用域规则，且这些扩展对符号表的设计有着显著的影响。

在本章的剩余部分，我们要考虑诸如控制名字的可见性、改变搜索规则和允许一个作用域中名字的多重使用等语言特性将如何影响符号表的结构。我们也将考虑隐式声明和在定义前引用名字等语言特性所带来的影响。

对标准名字作用域的扩展可分为两类：其中一类扩展改变个别名字或名字集合的可见性，而另一类则改变搜索规则。我们将考察代表这两类扩展的语言特性的处理技术。标准的名字可见性规则指出与名字引用关联的定义可能是包含该引用的最内层作用域中的那个定义。然而，并非所有的名字都遵守这个规则。例如，记录域的名字通常仅当它们由记录名限定（qualified）时才可见。更重要的是，现代程序设计语言常常允许显式地控制非局部名字的可见性。这样的控制又可以划分为*导入规则*（import rule）和*导出规则*（export rule）。最后，像Pascal语言中的**with**语句构造和Ada语言中的**use**子句均实际带来新的搜索规则。

对于大多数扩展，均可采取以下两种方法中的任何一种来实现。一种方法是复制那些通常在最内层名字作用域中可以访问但可能在常规搜索过程中不被发现的符号表条目。这种方法的实现通常相当简单且能加快搜索速度。然而，它却容易带来显著的空间开销，尤其是在必须拷贝大量条目的时候。

另一种方法则试图避免条目的拷贝并调整有关标志和符号表链接以便符号表结点在使用常规搜索方法时可见或不可见。这种方法通常更复杂且有时更慢，但它却可能减少总体空间开销。Graham等（1979）给出有关这些方法很好的全面讨论。

8.4.1 域和记录

在C、Pascal和Ada语言中，域名仅需在其所声明的记录中保持惟一即可。尽管它们在包含记录声明的作用域中可见，它们也无须和声明在相同作用域中的其他记录的域名或其他局部和非局部名字不同。因此，我们必须能处理像以下Pascal程序示例中的声明： 267

```
A,R : record
      A : Integer;
      X : record
            A : Real;
            C : Boolean;
          end;
    end;
```

这些关于域名的规则是我们想要的，因为它们使程序员不必担心在不同的记录中的域名重复，如此也就简化了程序员的工作而且增强了程序的可读性（与那些要求在同一作用域中域名必须惟一的语言相比）。每个域可以根据它的用途取合适的名字，即使另一个记录有相同名字的域也没有关系。例如，C语言的最初定义就要求这样的惟一性，强迫程序员采纳有关的命名约定，诸如：

```
struct rec1 { int r1_contents; struct rec1 *r1_next; };
struct rec2 { char *r2_contents; struct rec2 *r2_next; };
```

此例中的前缀r1_和r2_是用来确定惟一的域名，因为两个记录不允许有相同的域名contents和next。这倒使得实现相对简单，一个符号表用于结构域，另一个符号表则用于别的名字。

在Pascal和Ada语言里，拥有相同名字的多个记录域绝不会导致二义性，因为对记录成员的引用必须总是完整地指定——例如，R.A或R.X.A。在PL/I和COBOL语言里，（域名）引用的中间部分可以删除。在这两个语言中，例如，R.C（表示R.X.C）是合法的引用，但这要求更加细致的算法以检测二义性并解析非二义的缩写形式。

域名的处理有两种选择。第一种方法是为每个记录类型分配一张符号表。那个符号表不出现在作用域栈上而是作为记录类型的属性。Pascal P-编译器就是这么做的，其中每个记录类型均有包含域名的二叉树。当编译器处理像R.A那样的引用时，它首先根据常规搜索规则发现R并返回存放R类型信息的符号表引用。然后它以那个符号表和A为参数来调用find()例程。这种方法因为不对符号表接口做任何改变所以很容易实现，但如果采用哈希表实现，它的空间需求将是很昂贵的。这种方法一般很适合于二叉树实现，因为在记录中域的数量通常不是很多。

另一种方法将域名看作普通的标识符并把它们和当前作用域中其他名字一起存放在符号表中。每个
268 记录于是被分配一个惟一的记录号（record number）。此记录号可由一个计数器产生，或是记录可能使用的符号表条目的地址。将记录号与所有的域名相关联，那么即使相同的域名可以在多个记录中使用，编译器也能够判定记录域归属哪个记录。非记录域的名字可认为有一个为零的记录号。为解释像R.A那样的引用，编译器同以前一样，首先在符号表中找到R并查证它是否为记录类型。为了说明，设它的记录号为k。然后，编译器在符号表中查找是记录域且记录号为k的A的声明。此过程对于像R.X.A那样复杂的表达式可能要重复多次。当编译器在符号表中查找的是普通变量，如A而非R.A时，它使用的记录号为零且忽略所有是域名的条目。

要使用这种方法来处理记录域，必须修改符号表接口，在enter()和find()等例程的参数表中添加记录号参数：

```
        . . .
extern void enter(symbol_table table,
                  const string name,
                  const int record_num,
                  id_entry *entry,
                  boolean *present);

extern void find(const symbol_table table,
                 const string name,
                 const int record_num,
                 id_entry *entry,
                 boolean *present);
        . . .
```

8.4.2 导出规则

导出规则允许程序员指定局部于某个作用域的某些名字在那个作用域之外仍可见。这种有选择的可见性可与通常的块结构作用域相比较，后者规定局部于某个作用域的名字在那个作用域之外不可见；与前面讨论的记录域规则相比较，后者通过合适的限定修饰使定义在记录中的所有名字可见。输出规则通常和语言中将相关定义封装在一起的模块化特性相联系，诸如Ada的“包”（package）、Modula-2的“模

块”（module）和C++的“类”（class）。模块或包常常既定义了一种数据结构又定义了在那个数据结构上执行操作的过程和函数。例如，考虑如下的Modula-2模块，它定义了数据结构——栈：

269

```
MODULE IntegerStack;
 EXPORT Push, Pop;
 CONST StackMax = 100;
 VAR Stack : ARRAY [1..StackMax] OF Integer;
     Top : 1..StackMax;

     PROCEDURE Push(I:Integer);
      BEGIN ··· END;
     PROCEDURE Pop : Integer;
      BEGIN ··· END;

    BEGIN
     Top := 1
    END IntegerStack;
```

在模块IntegerStack外，过程Push和Pop可见，而其他所有局部于该模块的定义均被隐藏起来。特别地，这种栈的实现不能从模块外访问。导出规则的目的不是简化编译处理，而是将程序单元中相关的定义很容易地组装在一起并有选择地访问那些定义。

为正确处理导出规则，我们必须在退出作用域时确保所导出的名字可见，就好像它们是在外层作用域中声明的一样。我们可以标识导出的名字并在作用域关闭时将它们移至临近的外层作用域，在移出后要确保删除那些导出标志。在Modula-2程序中，导出符号必须在新作用域的一开始的地方就列出，因此，在关闭作用域时我们很容易定位它们。使用带有链式冲突解决的哈希表来实现符号表时，根据使用的是单个表还是多张表，可将所有的外部符号连续地存放在链的尾部或直接放在属于临近外层作用域的符号的前面。如果采用带有每作用域一棵树的二叉树实现，可在临近树根的地方发现所有的导出符号。

为处理导出符号列表，对符号表接口所做的最基本修改是，必须在例程enter()的参数表中添加额外的参数exported。该参数是一个布尔值，用来标记正被添加到符号表的名字name是否在包围当前作用域的外层作用域中可见。我们更关注的是作用域被关闭时，如何实际导出符号表条目。如果符号表接口包括open_scope()和close_scope()操作，那么我们将内部处理作用域，且可像前面描述的那样导出条目，而无需修改符号表接口。如果作用域表示为单独的符号表且作用域栈是在符号表例程之外处理的，那么我们必须为符号表接口添加新的操作：

```
extern void export(const symbol_table from,
                   symbol_table to);
```

export()必须在from所指示symbol_table中搜寻所有的导出条目并将它们移至由to所指示的符号表中。

Ada语言的“程序包”语法面向大型程序支持。包的定义分为两部分，首先是规范部分（specification part），它定义了由包导出的名字；其次是包主体（package body），其中隐藏了所有声明并给出在规范部分声明的过程的程序体。前面Modula-2的栈的Ada语言版本定义如下：

270

```
package IntegerStack is
 procedure Push(I:Integer);
 function Pop return Integer;
end IntegerStack;

package body IntegerStack is
 StackMax : constant Integer := 100;
 Stack : array (1..StackMax) of Integer;
 Top : Integer range 1..StackMax;

 procedure Push(I:Integer) is
  begin ··· end;
 function Pop return Integer is
  begin ··· end;

begin
 Top := 1;
end IntegerStack;
```

这里没有任何显式的导出列表；在包规范部分的声明在包外部可见。因此，Push和Pop由Ada的程序包IntegerStack导出。在Ada语言中，导出的对象不能自动地导入到其他程序单元。相反地，它们可以通过在其前面加上程序包标识符的限定（例如，IntegerStack.Push）来访问，或借助**use**子句导入它们来进行访问。

如果使用的是每作用域一张表的方法，我们将把程序包中所有的可见声明放在该程序包的符号表里并借助程序包名来访问它们。对于单独编译的程序包，将有一个库条目包含这张表的内容。如果使用的是单个表的方法，我们将在每条哈希链上添加一个特殊的搜索终止标记。该标记终止沿哈希链的普通搜索。当处理程序包的规范时，它的局部声明将放置在该标记的后面，这使它们不可见。当编译程序包主体时，它们被移到标记的前面并在包体处理后返回。正如我们稍后所讨论的，选择性的访问和**use**子句方法将超越这个搜索终止标记进行搜索以发现程序包的声明。

Modula-2语言也允许将模块划分成类似的部分，如下例所示：

```
DEFINITION MODULE IntegerStack;
 EXPORT QUALIFIED Push, Pop;
 PROCEDURE Push(I:Integer);
 PROCEDURE Pop : Integer;
END IntegerStack.
IMPLEMENTATION MODULE IntegerStack;
 CONST StackMax = 100;
 VAR Stack : ARRAY [1..StackMax] OF Integer;
   Top : 1..StackMax;

 PROCEDURE Push(I:Integer);
  BEGIN ··· END;
 PROCEDURE Pop : Integer;
  BEGIN ··· END;

BEGIN
 Top := 1
END IntegerStack.
```

271

我们注意到，在Modula-2定义模块中的导出列表包括了关键字**QUALIFIED**。当模块被分成两部分时修饰符是必需的；否则它是可选的。受限定的导出必须以模块名为前缀来访问。在此例中，我们必须使用IntegerStack.Push和IntegerStack.Pop来引用导出的过程名。Ada语言所有的导出标识符可以选择使用程序包名或使用**use**子句来导入。我们可以使用前面我们所开发的用来引用记录域的技术来处理受限定的导出名字。

单独编译

当程序包或模块声明用作单独编译的程序单元时，整个程序可以从这些单独编译的部分构造得到。当程序包规范或定义模块被编译时，编译器将有关导出声明的信息保存在一个库（library）中。（这里所说的库，简单地讲就是一个存放单独编译单元的信息的场所。）保存在库中的信息可使编译器在编译相应的程序包体或实现模块时以及在编译从此单元导入声明的其他单元时能够为这个单独编译的声明建立符号表。这样，即便使用单独编译，这种库也使得进行完整的编译时静态检查成为可能。

每作用域一张表的符号表组织形式最适合于实现包含单独编译特性的语言，如Ada和Modula-2。信息库可以包含单独编译单元符号表的符号化表示（如，正文）或内存映象表示。如果这样的单元在编译其他单元时被引用，此库中条目可用来重新构造它的符号表，使它包含的所有信息均成为可访问的，就好像它正和其他单元一起被编译一样。

272

像C那样不依赖编译器已知的信息库的语言，将使用别的方法进行单独编译。在其他单元中已编译对象的描述可以通过使用特殊的编译器指令（如，C语言中的#include）以源语言形式被包含进某个编译单元。处理这些声明将建立合适的符号表条目，但这远不如装入一个先前已建好的符号表高效。此外，还无法保证包含进来的声明与这些被导入对象在定义它们的模块中的声明相匹配。

隐藏类型表示

Ada和Modula-2语言均提供使用程序包和模块来定义类型的语言特性，而类型的实际实现却被隐藏

起来。这样，我们可能导出一个栈类型，接着再创建任意数量的栈——相比之下，此前的例子中仅允许创建单个栈。然后可以通过所提供的操作来访问并操控这些栈的实例。这样的栈类型被称为*抽象数据类型*（abstract data type）。Ada和Modula-2语言中提供此能力的语言特性可参见图8-9a和图8-9b的示例。Modula-2只允许指针类型被隐藏，因此，编译器将自动获悉此种类型的大小。在Ada语言里，程序包规范的*私有部分*（private part）中私有类型的定义提供了必要的由程序员定义的任意类型的大小信息，而这些信息对程序的其他部分不可见。

这些示例中有趣的是，在Ada语言中使用了**type Stack is private**，而在Modula-2语言中则使用的是**TYPE** Stack。Stack在Ada中称为私有类型，而在Modula-2中称作不透明的类型。在这两种情况下，导出的只有名字Stack，而有关它的实现细节则不可用。如此一来，在包体外面将不能访问此Ada实现中的记录域名。在两个语言中，不需要看见包主体或模块体，编译器就已获得足够信息从而知晓这些隐藏类型大小。这些信息足以用来编译使用这些类型的模块。这些语言特性对符号表惟一影响是它们对导出的信息所施加的约束。而这些情况并不需要什么新的技术来处理。

```
package IntegerStack is
 type Stack is private;
 procedure Initialize (S : in out Stack);
 procedure Push(I:Integer; S : in out Stack);
 procedure Pop(I: out Integer; S : in out Stack);
private
 StackMax : constant Integer := 100;
 type Stack is record
                Stack : array (1..StackMax) of Integer;
                Top : Integer range 1..StackMax;
              end record;
end IntegerStack;

package body IntegerStack is

 procedure Initialize (S : in out Stack) is
  begin ··· end;
 procedure Push(I:Integer; S : in out Stack) is
  begin ··· end;
 procedure Pop(I: out Integer; S : in out Stack) is
  begin ··· end;

end IntegerStack;
```

a) 在Ada语言中

```
DEFINITION MODULE IntegerStack;
 EXPORT QUALIFIED Stack, Initialize, Push, Pop;
 TYPE Stack;
 PROCEDURE Initialize (VAR S:Stack);
 PROCEDURE Push(I:Integer; VAR S:Stack);
 PROCEDURE Pop(VAR S:Stack):Integer;
END IntegerStack.

IMPLEMENTATION MODULE IntegerStack;
 TYPE
  StackEntry = RECORD
                 Value : Integer;
                 Next : Stack
               END;
  Stack = POINTER TO StackEntry;
 PROCEDURE Initialize (VAR S:Stack);
               BEGIN ··· END;
 PROCEDURE Push(I:Integer; VAR S:Stack);
               BEGIN ··· END;
 PROCEDURE Pop(VAR S:Stack):Integer;
               BEGIN ··· END;
END IntegerStack.
```

b) 在Modula-2语言中

图8-9 Integer Stack作为一种抽象数据类型

8.4.3 导入规则

在强制使用导入规则的语言里，作用域可分类为*导入型*或*非导入型*两种。（术语*开放*和*封闭*有时用来取代*导入型*和*非导入型*；我们选择后面这一对以避免和本章前面介绍的开放式与封闭式作用域定义有任何的混淆。）导入型作用域自动地接收对外层包围作用域的定义的访问。在Algol 60和Pascal语言里，所有的作用域均为导入型作用域。

274 在非导入型作用域里，对某些或全部非局部名字的访问要求必须有显式的导入声明。Modula-2中的模块是非导入型作用域。只有那些标准的预定义的标识符能够被自动地导入且无需在导入列表中列出。它们被称为*有渗透力的*（pervasive）。

在某些语言里，非局部对象在导入时可带有约束条件，如*只读型*对象就意味着它在导入作用域中不能被修改。对象必须在非导入型作用域间逐层导入，且在导入时绝不会比在原先的作用域中拥有更多的特权。也就是说，一旦变量作为只读变量被导入，它也只能作为只读变量被导入到内层作用域。

C++语言有令人感兴趣的作用域概念，它提供了三种类成员的可见性规则：`private`、`protected`和`public`。任何人都可以访问类的`public`成员，这里通常包括类的大多数或全部例程。子类可以访问父类的`public`和`protected`成员。类自身的实例可以访问所有三种类成员。另有`friend`函数，它是一个非类成员的函数但却可以访问类的所有成员，即使是`private`的成员。

作为导入规则应用的示例，可以考虑以下的模块。它是另一种栈的实例，但这一次栈的元素类型Thing是从外层包围作用域中导入的。因为Modula-2模块是非导入型作用域，因此需要使用**IMPORT**使Thing的使用合法化。

```
MODULE ThingStack;
 EXPORT Push, Pop;
 IMPORT Thing;
 CONST StackMax = 100;
 VAR Stack : ARRAY [1..StackMax] OF Thing;
   Top : 1..StackMax;

 PROCEDURE Push(I:Thing);
  BEGIN ··· END;
 PROCEDURE Pop : Thing;
  BEGIN ··· END;

BEGIN
 Top := 1
END ThingStack;
```

显式的导入规则的目的是更准确地控制诸如子程序和模块等主要程序单元的接口。它们意在增强这些程序单元的可读性和可靠性。

为实现导入规则，我们必须首先改变原先标准的搜索机制以便所有名字或某些种类对象的名字（通常是变量）在穿越非导入作用域边界时不能被自动地引用。这很容易做到。我们已经记录了一个定义所属的作用域，因此当遇到那些非导入型作用域时，我们可以使该定义成为不可访问的。根据作用域实现
275 方式，我们通过为`create()`和`open_scope()`等符号表例程添加有关参数而将每个作用域标记为导入型的或非导入型的。在搜索完成时，我们将查看所发现的对象是否具有渗透力（或自动被导入）。如果它不是，且它是在最内的非导入型作用域的外围作用域中被发现的，则搜索的结果就如同此标识符不曾出现在符号表中一样。

为实现导入语句，我们简单地在当前作用域中创建与导入名字对应的条目。通过添加如下单个过程我们可以扩展到符号表的接口以支持导入操作：

```
/*
 * Import name into table; return a reference to
 * the entry corresponding to name and a flag to
```

```
 * indicate whether the name was successfully found.
 */
extern void import(symbol_table table,
                   const string name,
                   id_entry *entry,
                   boolean *found);
```

import()将用来替代enter()处理被导入的名字。它创建的局部条目包含到导入名字先前条目的间接链接。这样，对搜索机制而言，这些导入名字看似局部定义并能正确地被找到。此外，因为任何一个导入名字存在相应的局部条目，如果我们查看到某个局部声明的名字和导入的名字冲突，那我们便发现了一个“多重定义”的错误。

这种方法仅在非局部变量必须导入时才可以工作得很好。特别地，大多数例程一般不会访问很多的非局部变量，因为这样被认为是不好的程序设计风格，因此，那些额外条目的数量应该在一个合理的范围内。此外，实现只读型导入仅需要做简单的扩展。在那个副本条目中，我们要么设置标志表明此约束，要么拷贝此非局部变量条目的内容，并添加标识以表示修改此变量是非法的。（这样一种标识也可用于Ada中的**in**参数。）

额外拷贝方法的主要困难出现在那些要求程序员导入所有非局部名字而不仅仅是变量名字的语言中。这种情况下，额外的空间和组织时间可能成为明显的负担。如果我们有许多嵌套的非导入型作用域，情况将更加严重。因为此时我们通常要逐级地进行导入。这里我们有另外一种方法，它采取特殊措施以便当非局部定义可见时进行标记。为达此目的，一个简单方法是为每个符号提供可以看见它的作用域的列表。不幸的是，这样的列表自身可能非常长且使用代价不菲。根据一个重要的观察，我们排除了对那些列表的需求。非局部名字是逐级导入的；因此，能够看见某个定义的那些嵌套的非导入型作用域必定是一个连续序列。也就是说，一旦一个作用域不能导入某个符号，那么嵌套在它内部其他作用域也不能 276
导入那个符号。这样，我们仅需给每个符号标记一个maximum_depth域。这个域的意思是说，那个符号在直到但不超过这个深度的作用域中可见。这个域初始设置为符号定义处的嵌套深度，而有渗透力的符号其嵌套深度为无穷。

如果符号没有导入到当前作用域中，那么当前嵌套深度将大于该符号所允许的最大深度，并拒绝对该符号的访问。如果符号被导入，它的最大深度域的值将递增，且允许对该符号的访问。注意，当我们编译完导入在其中完成的作用域时，最大深度域必须要重新设置为它们的原来的值。这项工作可以通过寻找所有最大深度域值大于它们原来嵌套层次的符号来完成，因为最大深度值只有随着导入才会增大。

如果允许只读型导入，我们可以使用另外的域来标识那些执行只读引用的作用域数量。如果这个值为零，将放弃这个只读约束。这个计数同样必须在处理只读（作用域）列表时更新且在作用域结束时递减。

这种方法的困难在于，它必须搜索整个符号表来寻找那些由于导入和只读列表而递增值的条目。许多产品级编译器实际上也是这么做的。我们可以通过把导入的条目移至（带有链式冲突解决法的）哈希链的首部来避免该方法的开销。现在，当作用域结束时，那些导入的条目可以很快地被发现并恢复到原来的位置上。

8.4.4 可更改的搜索规则

with语句

Pascal语言的**with**语句是一个为了解释标识符含义而改变作用域的检查次序的很好的语言特性的例子。我们有如下语法形式：

with R **do** <statement>

其中<statement>可以是任意大小的复合语句，它所包含的标识符，如有可能，将解释为记录R的域引用。也就是说，作用域检查的次序为：（1）**with**语句中出现的记录（此例中为R），（2）最内层的过程

(或函数)，(3) 外层包围过程（或函数)，(4) 全局名字作用域。**with**语句可以嵌套，那时首先按从最内层到最外层的顺序检查这些语句所指示的记录，然后再检查正常的嵌套名字作用域。

当进入**with**语句时，那些通常因为没有加记录名限制而不可见的记录域现在必须对搜索过程可见。如果每个记录和作用域均有它们自己的符号表，这一点将很容易做到。记录的符号表被压入作用域栈并在语句结束时从栈顶弹出。

277 如果记录域和符号表中其他的标识符混在一起，则必须借助8.4.1节中的记录号技术，并采取别的方法来让这些域名可见。一种方法是打开一个新的作用域并把所有属于指定记录的域的相关条目拷贝到此作用域中。此技术虽然避免了对名字搜索过程做任何改动，但所有拷贝的开销是不能接受的。(此开销或许可通过仅拷贝指针以取代拷贝条目本身来减轻一些。) 另外一种方法是用一个栈包含由所有开放式**with**语句所指定记录的记录号。例程`find()`首先使用此栈顶部的记录号来搜索名字，然后是次栈顶的记录号，等等。如果使用栈中的任何记录号均未发现有匹配的名字，该例程将尝试记录号为零的搜索，而这将导致进行通常的搜索。使用这种方法，搜索速度可能会很慢，因为在发现名字条目不是记录域之前，我们要额外搜索栈中的每一个记录号所对应的作用域。把记录域添加到一个新作用域中虽然避免了搜索上的时间惩罚，但在语句的开始和结束处还是要花费额外的时间。

名字选择

在Ada语言里，可以通过在对象前面冠以程序包、子程序、程序块或循环的名字来选择它们。对程序包而言，此方法是需要的，因为可见对象不能通过普通搜索规则来访问，除非使用**use**子句。这种选择还可用来让那些可能被对象名字的局部重定义所隐藏的对象重新可见。此外，这种方法在显式地选择重载名字的某个特别定义时也很有用（参见第8.6节)。

如果某个标识符命名了一个作用域，此标识符的属性之一是指向那个作用域的符号表的指针或者是与那个作用域关联的作用域号。此时选择性访问和记录域的访问类似。即，我们使用作用域的名字来确定一个符号子集，然后仅在这个子集中搜索标识符。

作用域名字自身也必须遵循特殊的作用域规则。在Ada/CS语言里，程序包不能嵌套，因此，编译过程中已经处理的包名和正在处理的当前程序包的名字均被看成全局名字。在Ada语言里，程序包可以嵌套而且实际上它们可以在程序块、子程序和其他程序包中声明。包名也因此像其他对象的名字一样处理。单独编译的程序包可通过**with**子句来访问（它和Pascal语言的**with**语句有很大区别)。

子程序名也遵守通常的作用域规则，但对子程序内部定义的选择性访问也仅限于该子程序内部。块
278 和循环的名字均局部于相应语言构造的作用域，因而不可能从其外部进行选择性的访问。

Ada的use子句

Ada语言的情况更加复杂。一个**use**子句其实是一个声明而不是一条语句，它指出那些在一个或多个包中可见的定义可以被直接访问。当编译器处理以下子句时:

use $p_1, p_2, \cdots, p_n$;

(其中$p_1, p_2, \cdots, p_n$是程序包的名字）那些程序包中的所有可见定义均是可访问的，就好像它们是局部定义的一样。然而，此时需要特殊规则来控制名字的冲突:

- 如果名字可以通过使用通常的作用域规则找到（不包括由程序包提供的名字)，那么就应用那个定义。也就是说，包里的名字能够直接访问仅当它们不与任何外围作用域中声明的名字冲突；换句话说，可认为它们在包围正在编译中的程序单元的作用域中声明。
- 如果在**use**列表上的多个包均提供相同的名字，那么此时这些包中的那个相同名字的定义没有一个能直接可见，除非它们命名的实体允许重载（参见第8.6节)。此规则保证了程序包在**use**列表中命名次序的无关性。

此外，可以使用与处理**with**语句类似的技术来处理Ada语言的**use**子句。我们可以试着为所有从包

中导出且没有冲突的定义添加局部符号表条目。这可能代价不菲，因为某些程序包可能很大，有数百个可访问的定义（例如，提供I/O或数值子程序定义的程序包）。

作为选择，我们可以采用标准的搜索过程，它带有新的选项以应对寻找名字定义的失败。如果找到一个定义，则不需要检查任何程序包。反之，如果没有发现任何定义，我们必须要检查**use**列表上所有的程序包以查验的实际定义所在。如果由程序包提供的名字访问频繁，这种方法可能会很慢。

一种有效的折中方法是，只对名字的首次引用进行上述那种彻底的搜索过程。如果一个定义在某个包中被找到，我们将创建它的局部拷贝。我们这样做是因为如果名字被引用一次，它很可能会再度被引用。后续的搜索可能会因此而加快，而对那些没被引用的包中的名字，我们从不为它们建立条目。

8.5 隐式声明

有时仅仅是符号的出现即可充当对象的隐式声明。这样，在Algol 60中，在语句边上出现的标号即 279
可用来声明（或定义）此标号，而且可以应用平常的作用域规则。一个有趣的例子是Ada语言里的**for loop**索引：一个循环索引被隐式地声明为和循环的范围指示同一个类型，且一个新的作用域被打开以便循环索引不会和已存在的变量产生冲突。

处理隐式声明时，所考虑的关键问题是，被隐式声明的名字是遵循正常的作用域规则还是打开新的作用域以避免和已有名字产生冲突。如果名字确实遵循普通的作用域规则且不需要它自己的作用域，那么和其他声明一样，我们把它添加到符号表中。如果符号不遵守通常的作用域规则（像Pascal的标号和Ada的**for**循环索引），那么它就需要特殊的处理。

在Pascal语言里，标号可以存放在符号表中，但需要小心处理，一旦那些直接包含它的子程序或结构化语句退出，必须将标号标记为不可访问的。这个完成以后，我们就不可能从结构化语句的外面跳到它的内部。Pascal标号的问题在于它的声明和访问规则不能很好地配合。和其他种类的标识符一样，一个给定标号在一个名字作用域（子程序或主程序）中仅能有一个声明，但从那个声明所在作用域到那个标号的所有跳转并不都是合法的。

对于像Ada中的**for**循环那样的能打开一个新的名字作用域的构造，我们在处理它的时候可以实际为它打开一个仅包含单个声明的全新作用域。如果允许内层声明隐藏名字的选择性访问，如在Ada中那样，那么就有必要创建新的作用域。如果不允许选择性的访问，那么作为小小的优化，我们可以试着将索引变量包括到当前作用域中。如果先前没有该名字，这点将很容易做到。如果已有相同名字，那么该名字的当前条目将被“抽出”，而取代它的是循环索引的条目。在循环结束时，循环索引条目将从当前作用域中删除，且原先的条目将得到恢复（如果有的话）。

8.6 重载

大多数块结构的语言允许某些程度上的重载。即，相同的符号根据它们的上下文的不同可以拥有不同的含义。在Pascal语言里，一个简单的重载例子是，相同的标识符可以指示一个函数的名字和它的返回值。此重载可导致一些不易察觉的错误。例如，如果语句f := f + 1在无参函数f中出现，那么在左部出现的f将指示返回变量，而右边的f则表示该函数的递归调用。

Ada语言中支持重载的地方很多。过程与函数的名字、运算符、或枚举文字值均能被重载。也就是说，在同一时间可以访问多个不同的定义。例如，我们可以有：

```
function "+" (X,Y : Complex) return Complex is . . .
```

和 280

```
function "+" (U,V : Polar) return Polar is . . .
```

它们重载了+运算符来求解复数和极座标值，而+运算符也可用于整数和实数求和。所有这些可选择的定义同时存在。类似地，我们有：

```
type month is (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
```

和

```
type base is (Bin, Oct, Dec);
```

它们重载了Oct和Dec。

Ada语言中的重载规则是，过程、函数、运算符或枚举文字值等的新定义将重载而不是隐藏现有的定义，如果上下文可用来区分这些新旧选择的话。这样，如果我们有：

```
A,B : polar;  那么  ··· A + B ···
```

就是非二义的，因为在此上下文中只能有一个运算符+的重载定义是合法的。Ada语言有一个精巧的算法（将在第11章里讨论）可以判断给定的上下文所暗示的重载符号可能的解释。

C++语言也允许重载多数内建的运算符，这其中包括数组下标和函数调用的操作。

符号表必须提供名字所有可能的含义以便它们为重载解释算法所用。处理符号表中重载名字的关键在于将重载名字的所有可能的定义链接在一起。这就确保了在查找名字时，可以很快获得所有可能的定义。然后，使用重载名字的语义例程检查每个可能的定义并通过上下文选择其中的一个。做出这样选择的算法很复杂。我们将在考虑表达式翻译的语义例程中详细讨论这些算法。为使这个方法正常工作，无论名字在何时被重载定义，我们都必须首先查看该名字是否已经可见。如果它可见且不在当前作用域中，则把它填入当前作用域中并将此新条目与任何已有的定义相链接，那些已有的定义将成为不可见的。如果在当前作用域中发现该名字，我们就将这个新定义加入到重载链中，如果它是一个合法重载的话。无论是哪一种情况，我们都必须执行相关的检查以确保重载名字的新定义通过重载解析算法能够实际区别于它先前的所有定义。

在关闭作用域时，可以删除某些重载（名字）。这点很容易做到，只要重载链从内层中的定义向前
281 延伸至更远处作用域中的定义。关闭作用域将从重载链首的地方删除若干定义。例如，Pascal的函数定义创建了包括返回变量和函数名的适度形式的重载。返回变量是在函数体作用域内，而函数名则是在函数体外的作用域中（否则函数将不能在它的函数体外被调用）。在函数体内，这两个重载名字将链在一起以使对返回变量的引用和函数的递归调用成为可能。在关闭函数体时，返回变量的定义将消失，而函数名继续保留。这样就保证了在函数外部对函数的调用，但那里返回变量是不可访问的。

8.7 前向引用

在允许前向引用的语言里，若将名字引用解析为已有的非局部定义的引用而不是即将处理的局部定义的引用，那么可能存在着一定的危害。Pascal程序中有这种情况发生，那里指针声明中的类型在相同作用域的后面部分才开始声明。例如：

```
type T = Integer;
    ···
procedure PP;
  type
    P = ↑T;
      ···
    T = Real;
      ···
end
```

P应当是指向Real的指针，但是它却很容易（不正确地）被解释为指向Integer的指针。注意，前向引用是提供用来允许那些互相引用并链接的类型。

类似的问题也出现在像Algol 60那样允许到非局部标号的**goto**的语言里。因为标号遵守作用域规则，引用标号L的**goto**语句若最终绑定到非局部标号L，仅当所有中间作用域已全部处理完毕。这是因为在中间作用域中出现的L将会取代L的已有出现，而且检验无此定义存在的惟一方法是彻底地处理所有这样的中间作用域。在Pascal中，事情要简单一些，因为标号必须在定义前声明。这起码告诉我们期望有一个标号的定义，即使我们还没有发现它。

前向引用中更具挑战性的问题出现在我们试图严格执行与前向引用相反的规则的时候。例如，对常量的前向引用在Pascal和Ada语言中是非法的。但如果在Pascal程序出现如下语句，情况会如何呢？ 282

```
const C = 10;
  .
  .
  .
procedure PP;
 const D = C;
     .
     .
     .
   C = 20;
end
```

D应该等于10还是20？事实上，它应该哪个都不是，因为D的定义包含一个非法的前向引用。特别地，过程PP中所有C的引用必须是对最内层中C的定义的引用，但在D的定义点，它却是一个前向引用。

这个错误很难发现，因为我们知道，如果D是正确定义的，它一定会等于10。事实上，几乎所有Pascal编译器会将D置为10且不发出任何错误信息。仅在遇到局部C的定义的时候，我们才有机会注意到这个错误，而那时D的处理已经全部完成了。注意，尽管如此，一个正确的Pascal编译器还是会发现这个错误的。Ada语言指出，一个声明的作用域仅从其定义点延伸至包含它的作用域末尾，因此，在Ada中不会出现这个问题。

处理前向引用

为正确处理前向引用，在看到符号所有可能的定义之前，我们必须要尽量避免立即对它进行解析。如果允许无约束的前向引用，那么处理此类问题惟一的方法是对程序正文进行多遍处理。第一遍寻找所有的定义并建立符号表。紧接着，在后续遍中处理声明和语句，此时使用已经建好的符号表解析所有的名字引用。它所依据的是，如果一遍编译器确实可行，那么前向引用必定是严格受限的。

在Pascal语言里，到类型名的前向引用可以用在指针类型中。在处理时，我们可以将所有指向类型T的指针引用链接在一起直到类型声明部分结束为止。那时，能找到并解析所有的类型名。这种处理方法是可行的，因为类型声明并不产生代码，惟一需要更新的是表示类型信息的内部数据结构。

goto语句中标号的前向引用问题比较棘手，因为我们要为**goto**语句生成代码，但即使在一遍编译器中问题也还是有可能解决的，因为我们几乎知道我们将产生什么样的代码——所缺少的就是待填入的地址。很多一般的前向引用通常不可能在一遍处理中得到解决。例如，在PL/I语言里，如果我们允许语句A：= B + C出现在A、B或C的定义前，那么简单地填写地址是不会满足要求的，因为代码的种类（和它的大小）将依赖于A、B或C的声明。于是，可能需要先进行一遍处理以便编译器在试图生成合适的代码之前能够确定标识符的类型。 283

非法前向引用

为发现一遍编译器中非法的前向引用，我们采取了那些用于导入表的技术。这里的想法是，如果我们知道某个符号的引用可以做非局部的解析，而且它的局部定义依然没有的话，我们将视那个符号已被导入。在实际情况中也确实如此，因为该引用隐含地指出将不会有那个符号的局部声明出现。如果有那么一个的话，它也必须因为前向引用的约束而先于它的首次使用。我们可以特殊标记这个隐式的导入，

这样，如果找到局部定义，那么将会出现与那个导入符号的冲突，而且我们也将做出正确的诊断。

这种方法是可行的。但惟一的问题是，我们必须在所有的情况中都做出大量的工作而其结果只是在碰运气式地希望在后面会出现一个非法的局部定义。如果我们想彻底地进行检查，这些就不可避免。这也让我们很容易明白为什么许多语言的定义只是简单地指出，在新定义找到前，非局部定义可以被引用。这就是Ada语言的作用域规则；到目前为止，它还是比较容易执行的，而且在大多数情况下它也是相当合理的。

8.8 小结

本章的大部分都在讨论块结构符号表以及各种对基本作用域思想的扩展给符号表带来的影响。根据这些讨论，图8-1中提出的符号表接口在某种程度上明显地被理想化了。为在任何特定编译器中使用那个接口而必须做出的修改取决于正被编译的语言的特性集合以及所选择的作用域表示。作用域表示的选择——单个的全局表或者每作用域一张表，其实可以由语言来决定，或者它可能受到来自编译器设计、特别是多遍组织的编译器的设计的强烈影响。

练习

1. 在产品编译器中用来实现符号表的最常见的两种数据结构是二叉搜索树和哈希表。这两种数据结构各自都有什么优缺点？

284

2. 如果在某个没有动态存储分配的场合中使用哈希表，可以通过从固定数组中分配哈希链上的条目来实现外部冲突解决法。试将这种方法和内部解决技术以及采用动态分配的外部链式技术进行比较以确定它的优缺点。
3. 描述本章里所介绍的处理符号表中多重作用域的两种技术，并列出每种方法中打开和关闭作用域所需的动作。追踪每种方法在编译图8-5中的示例程序时所执行的动作序列。
4. 分别给出采用二叉搜索树和哈希表方法在实现每种多重作用域（单一全局符号表和每作用域一张表）时所需的四种名字搜索算法。
5. 什么样的语言使用标识符的串空间表示是不合适的？为什么？
6. 比较本章中介绍的处理记录域名字的两种方法。每种方法是如何与每种多重作用域选择协调工作的？
7. 描述在每种多重作用域的实现中，在外层包围作用域中处理Modula-2模块导出名字的技术（见8.4.2节的介绍）。
8. 比较处理导入名字的两种方法。它们是如何与两种作用域表示方法协调工作的？
9. 分析你喜欢的程序设计语言，确定它的哪些特性对设计适合那个语言编译的符号表有影响。具体描述每一个相关特性的影响。

   根据上面的分析，给出编译你所分析的语言所需符号表的接口和内部设计。
10. 从你或其他程序员编写的程序中收集一些标识符。编写程序将这些标识符存储在二叉搜索树中并报告结果树是如何保持完美平衡的。
11. 完美哈希的概念曾在3.5节作为词法扫描器快速识别关键字的方法介绍过。许多已发表的关于此话题的文章均强调创建使用空间尽可能少的最小完美哈希表的算法。该方法的一个缺点是，大多数或所有的非关键字标识符串产生的哈希值与关键字的哈希值相冲突，这样，为了最终确定单词需要进行串比较的操作。因为串比较在许多机器上较慢，因此我们将尽量避免它们。较大的哈希表可能使得某些哈希值没有与之关联的关键字。如果标识符映射到这样的值，则无需串比较来做出标识符/

关键字的选择。采用实验或分析的方法研究完美哈希函数所用哈希表大小和关键字识别效率之间的关系。

12. 在8.4.1节讨论记录域名字的时候，我们提到某些语言允许名字域表达式的缩写形式。使用那一节里的示例，R.X.C和R.C将是等价的引用，如同R.X.A和R.A一样。描述正确处理这样的域引用所必需的数据结构和算法。 285

13. 另一种命名记录域的方法和Pascal、Ada或类似语言中所使用的顺序相反。也就是说，它使用C.X.R来代替R.X.C。描述在从左到右的单遍处理中正确处理这样的域引用所必需的数据结构和算法。

14. 练习12或练习13中描述的可更改的搜索规则将如何影响Pascal语言的**with**语句的实现？ 286

第9章 运行时存储组织

程序设计语言的发展带来了日渐多变的运行时存储管理，起初，所有的存储分配均为静态的（static）——即，在整个程序运行期间保持不变。Algol 60及其后继语言引入了要求栈式分配的特性，存储空间将在程序运行期间的特定时刻被压入或弹出运行时栈，例如，在过程被调用或返回时。Lisp及其后来的语言，包括Pascal，引入了堆分配（heap allocation），允许在程序运行期间的任一时刻进行空间 287
的分配和释放。

作为一种考察语言存储分配需求特性的方法，我们首先考虑数据区（data area）的概念。数据区是一个存储块，该区域中变量和其他编译器所已知的对象均采用一致的存储分配需求。即，数据区中任一对象必须要分配空间时，由它包含的所有的对象也必须同时分配空间。Micro程序的变量就是一个简单数据区示例。它们在程序开始运行时即被分配，并且保持这种分配直至程序运行结束。由Pascal或Ada中的调用new动态分配的记录也可以作为一种数据区。

9.1 静态分配

在许多早期语言，尤其是汇编语言和FORTRAN中，所有存储分配都是静态的。在程序生命期中，数据对象的存储空间被分配到固定的位置。仅当待分配对象的数量和大小在编译时刻是已知的时候，静态分配的使用才成为可能。这种分配方法，当然，使得存储分配变得极简单，但造成很大的空间浪费。于是，程序员有时必须覆盖（overlay）变量。例如，在FORTRAN中equivalence语句常用来减少存储需求。覆盖能导致很微妙的程序设计错误，这是因为对一个变量的赋值将隐式地改变另一个变量的值。覆盖还降低了程序可读性。

在现代语言中，静态分配用于那些大小固定、在程序整个执行期间均可访问的全局变量，以及那些需要在程序执行期间保持不变的文字常量。静态分配还可用在Modula-2的模块或Ada的高层次包结构中的局部变量，以及C中static和extern变量。

从概念上讲，我们可以将静态对象绑定到绝对地址。人们往往比较喜欢用一个(DataArea, Offset)对来表示静态数据对象。Offset在编译时刻是固定的，而DataArea的地址可以延迟到链接或运行时刻确定。例如，在FORTRAN中，数据区可以开始于多个公共块中某一个，或者开始于某个子例程中局部变量的存储块。典型地，这些地址是在链接时刻确定的。地址绑定必须推迟到链接时刻，因为，FORTRAN的子例程可以是单独编译的，这将使编译器无法了解程序中所有的数据区信息。

作为选择，可将DataArea的地址装入到一个寄存器中，这就允许静态数据项可以表示为(Register, Offset)对。这种寻址方式几乎在每种机器上都有。这样表示一块静态数据的好处在于能使我们把静态对象置于何处的决定的做出一直延迟到执行之前。这项技术在加载-运行(load-and-go)的这样一个无显式链接步骤的环境下是很有用的，因为程序中各种构件的大小在整个程序被翻译前是无法确定的。这正是用来实现语言Micro的技术。例如，在知道生成的程序代码到底多大之前，我们将无法决定在哪里放置 288
文字常量池。因此，在完成语句编译前，不可能产生到该常量池的绝对地址的引用或访问。

9.2 栈分配

几乎所有的现代程序设计语言都包含了递归过程这一需要动态分配的语言特性。每一次递归调用需

要为过程局部变量的新的拷贝分配空间，因此，在程序执行期间所需的数据对象的数量在编译时刻是未知的。为实现递归，一个过程或函数所需所有数据空间将被看成是一个数据区，由于这个数据区处理方式特殊，我们称之为*活动记录*（Activation Record，AR）。当子程序返回时，AR从栈顶弹出，释放例程的局部数据。为明白栈分配如何工作，请考虑图9-1中所示的程序：

```
procedure p(a : integer) is
  b : real;
  c : array(1..10) of real;
begin
  b := c(a) * 2.51;
end;
```

图9-1 一个简单的子程序

图9-1中的例程要求为它的参数a和局部变量b、c分配空间。它也需要存放控制信息如返回地址的空间。在过程被编译时，它的空间需求被记录下来。特别地，每个数据项相对于AR开始位置的*偏移*将存放在符号表中。总的空间需求，也即AR的大小也被记录下来。在我们的例子中，假设p的控制信息需要5个字的空间（通常，所有过程的这个需求都是一样的）。参数a需要一个字的空间，变量b需要2个字的空间，数组c需要20个字的空间。图9-2显示了p的AR。

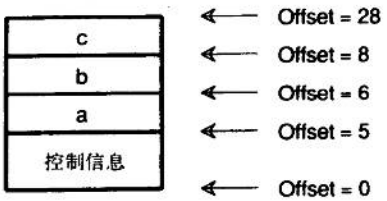


图9-2 过程p的活动记录

在p中，每个局部数据对象通过相对于AR开始处的偏移来寻址，这个偏移是一个固定的可在编译时确定的常量。因为我们通常把AR的开始位置存放在寄存器中，因此，每块数据可以用(Register, Offset)对来寻址，而这几乎是所有的计算机体系结构中的标准寻址模式。例如，如果寄存器R指向p的AR的开始位置，变量b可以寻址为(R,6)，那么该变量的存储地址在运行时刻将计算为R的值加6。

通常情况下，文字常量2.51不存放在活动记录中，因为存储在AR中的局部数据的值在调用结束时将
289 消失。如果2.51被存放在AR中，那么它的值在每次调用前将不得不初始化。而将这些文字常量分配到一个称为*文字常量池*（literal pool）的静态区中是一种既简单又高效的做法。

很多语言（包括Ada和Ada/CS）允许*动态数组*（dynamic array）。动态数组的边界是在运行时而非编译时确定的，因此，这些数组不能在活动记录中分配。一旦相关的声明清晰后（即，完全计算出来），动态数组即被分配空间。大多数语言中声明优先于语句，因此为动态数组分配空间往往是子程序在被调用后首先做的工作之一。

在第11章里我们将详细讨论数组声明的翻译。对于动态数组，我们在包含它的类型声明的子程序的活动记录中将存放一个称为*内情向量*（dope vector）的大小固定的描述符。内情向量是包含数组的大小和边界。这些信息是在详细描述数组声明时获得的。一旦数组的内情向量被初始化，我们就可以为数组的每一次出现分配空间。我们很方便在运行时栈上、当前AR的上方分配此空间。事实上，可以扩展AR以容纳动态数组。

在编译时刻，在当前AR中给每个动态数组指派一个的位置。在运行时刻，该位置将指向在AR末端以外的某个地方为数组所分配的空间。动态数组是间接访问的：首先从AR中获得数组的运行时地址，然后根据这一地址可以访问数组或它的元素。

为说明这点，可以在图9-1过程p中添加以下声明：

```
d, e : array(1..N) of integer;
```

在过程p被调用时，如图9-3所示的活动记录被压入到运行时栈里。数组声明在p被调用后即被详细说明。这意味要计算N，然后初始化内情向量。内情向量含有数组的大小信息，因此可在运行时栈上分配数组
290 d和e的空间，并且指向这些空间的指针保存在AR中。这一刻，我们得到如图9-4所示的AR结构。



图9-3 新版本p的活动记录

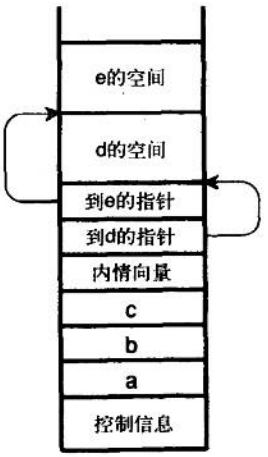


图9-4 p在声明清晰后的活动记录

一旦在其声明清晰以后，动态数组的大小也随之固定下来。此外，动态数组的生成期和其他任何局部变量相同，因此，通过扩展活动记录来包含动态数组既简单又有效。 291

一些语言提供过程内静态分配的变量（C中的`static`，Algol 60中的own）。这些变量的值在穿越调用时必须加以保留。同文字常量的情况一样，静态变量被分配到全局位置中而不是在活动记录里。Algol 60 甚至允许声明为own的数组在编译时大小不用确定，这就给实现上带来相当大的难度。正如稍后所讨论的，对于这种对象，我们不难在堆中给它找到空间，但是在声明它的过程被调用期间，如果它需要更大的空间，那么它将不得不拷贝到一个新的能容纳下它的地方。那些两个边界都是动态的二维数组更是难以处理。

在一些语言（包括Ada、Modula-2和Simula）中，单个的运行时栈已不能满足要求。这些语言没有沿用子程序退出时后进先出的Algol模式。Ada和Modula-2允许控制在进程间来回切换，这里的进程就像过程一样有自己的包含局部和非局部变量的引用环境。事实上，每一进程都必须有它自己的栈，类似地，Simula中允许过程作为协同例程来调用，因此，过程的所有局部变量都必须保留到过程确认终止为止。这些语言采用堆分配方式来分配整个栈。

在Ada、Simula和某些LISP实现版本中所分析的另外一个语言特性允许两个进程共享一个非局部变量的访问。而将访问环境的共享部分拷贝到两个进程的运行时栈中是不够的，因为在一个进程修改非局部值的时候，另一个进程应当总看见这个新值。替代的方法是，将栈建成一段一段的以便共享。这种方法有时称为cactus栈（cactus stack），因为这种组织形式让人想起既能从主干上也能从其他枝干上长出新分枝的树形仙人掌。重要的是，语言的设计必须保证共享的栈段在所有共享它们的进程终止以后方可释放。

9.2.1 显示表

活动记录常常通过指向它开始位置的某个寄存器来寻址。然而在运行期间，活动记录的数目可能超过可用寄存器的数量。例如，考虑图9-5中所示的程序。 292

我们可能有如图9-6所示的调用序列：

我们如何处理这样的调用序列呢？幸运地是，因为有作用域规则，不是所有的活动记录都需要在同一时刻成为可访问的。特别地，我们一直需要主程序的AR，但它却可以采用静态分配。所有的过程和函数都有一个由程序结构所确定的*静态嵌套层次*（static nesting level）（p和s在层次1，q和r在层次2）。在任一点上，依据块结构规则，任一给定的层次上只有一个过程能使它的变量可访问。我们为每个可能

的静态嵌套层次分配一个寄存器。(最大嵌套层次常常由编译器来限制决定)。这个寄存器集合称为显示表(display),该表中的每一个寄存器称为显示表寄存器(display register)。在任一给定点,第i个显示表寄存器(表示为D[i])将指向当前可访问的、静态嵌套层次为i的AR(若有的活)。在我们的示例中,如果当前我们在过程q中,那么我们可能有如图9-7所描述的状况。

```
1  program main is
2
3      procedure p is
4
5        procedure q is
6                ...
7        end q;
8
9        procedure r is
10               ...
11       end r;
12     end p;
13
14     procedure s is
15       ...
16     end s;
17
18 begin -- main
19   ...
20 end main;
```

图9-5 含有嵌套过程的程序

```
Call main program
Call p
Call s
Call p
Call s
Call p
  ...
Call p
Call q
Call r
Call q
Call r
  ...
```

图9-6 程序main中的过程调用序列

在任一点上,许多AR不能由任何显示表寄存器来寻址。它们中的局部变量,在那一点,也不是直接可寻址的。当过程返回时,显示表寄存器被更新,先前不能访问的AR此时可通过某些显示表寄存器
293 来寻址。因此,每次调用必须为被调过程建立合适的显示表,而每次返回必须将显示表恢复到调用前的情况。

一种极端做法是将整个显示表保存在每个过程的AR中,以便当过程返回时所保存的显示表可以用来恢复所有的显示表寄存器。AR中的"显示表区域"的长度可由当前过程的词法层次确定。我们总是留出足够长的最大区域以容纳最大允许的显示表。

幸运的是,我们不需要保存完整的显示表。在任何调用中仅保存一个显示表就足够了。调用可以是到比当前词法层次深一层的过程的调用(我们称之为"较高",属于父过程调用它的直接子过程的情况),也可以是到与当前层次相同的过程的调用(属于兄弟过程之间的调用),或是到任何较浅层次的过程的调用(我们称之为"较低",属于内层过程调用其任何外层过程的情况)。在每种情况下,在被调用层次上的显示表寄存器的值必须保存到被调用者的AR的显示表区域中。这个区域可以占一个字长。当调用返回时,这个值必须恢复。把显示寄存器保存在调用者的AR中也可以工作得很好,但那样的话,调用者在完成各种不同层次的过程的调用后,必须使用不同代码(来恢复不同层次的显示表寄存器)。因此,让被调过程来恢复寄存器要更容易一些;它只需恢复相同的寄存器,而不管调用过程的层次是多少。

这种机制在每次调用后重建整个显示表将其恢复到调用前的状态,即使其中部分显示表未被调用者使用。为什么维护那些高层显示表寄存器如此重要呢?考虑图9-8所示程序。

假定有以下过程调用序列:

A B C A' B' C'

294 我们用'来标记被调过程的不同运行实例。每次调用中使用的以及保存的显示表值如图9-9所示。

当A'返回到C时,上述方法仅将display[1]从A'更改为A;而其他显示表寄存器已正确设置为B和C。尽管A'自身不需要那些寄存器,但由A'开始的调用序列必须保存它们的值,以便在A'返回时,仅需恢复一个寄存器便能建立过程C的引用环境。

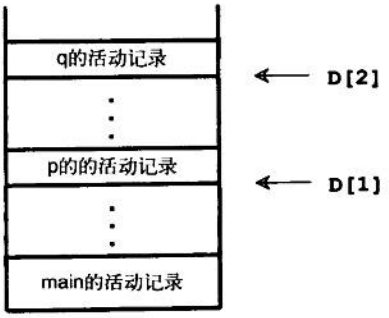


图9-7 运行时栈上的活动记录序列

```
procedure A is
   procedure B is
      procedure C is
         ··· A; ···
      end C;

      ··· C; ···
   end B;

   ··· B; ···
end A;
```

图9-8 嵌套过程调用的程序

calls	A	B	C	A′	B′	C′
display[3]	??	??	C	C	C	C′
display[2]	?	B	B	B	B′	B′
display[1]	A	A	A	A′	A′	A′
saved	–	?	??	A	B	C

图9-9 显示表的保存与恢复示例

9.2.2 块级与过程级活动记录

像Algol 60、C、Ada和Ada/CS这样的语言，允许块和过程一样，拥有局部声明的变量。拥有局部变量的块可被看成是无参的内联过程，因此，我们可以为每一个拥有局部声明的块创建一个新的活动记录。这种方法需要更多的显示表寄存器，因为包括块以后，静态嵌套层次可以加深许多。此外，它使得块的执行开销很大，因为AR要被压入栈，显示表寄存器要更新，等等。为避免这些开销，有可能仅有真正的过程才使用AR，即使在过程中的块也可以拥有局部声明。这种技术称为*过程级*（procedure-level）AR分配，与之相对的是称为*块级*（block-level）AR分配，即为每个拥有局部声明的块分配相应的AR。

```
procedure A (X,Y : in real) is
   QQ : integer;
begin
   B1: declare
      D,E: real;
   begin
      ···
   end B1;

   B2: declare
      G,H,I: integer;
   begin
      B3: declare
         J : real;
      begin
         ···
      end B3;
   end B2;
end A;
```

图9-10 有块级声明的程序

过程级AR分配的中心思想是，过程里的各个块中的变量的相对位置可以在编译时被计算并固定下来。这是可以做到的，因为块是严格按照词法顺序进入和退出的。例如，考虑图9-10所示的过程。

存放参数X、Y以及过程级变量QQ的空间必须在A中始终是可访问的。可能需要为块B1或B2中的变量分配空间，但不必同时分配。类似地，块B3的空间分配也只有当我们位于B2中时才能进行。因此，我们可以在编译时为过程A创建一个包含在A中声明的所有变量的AR。特别地，X、Y以及QQ的空间位置靠前，D、E的空间可以和G、H和I空间相互覆盖，而J可以刚好放置在G、H和I之上。

图9-11给出上述过程级活动记录中内容的安排。

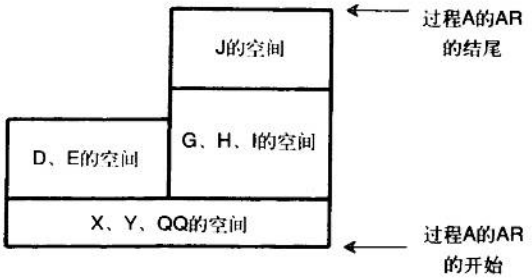


图9-11 过程A的过程级AR

9.3 堆分配

最灵活且最昂贵的存储分配是堆分配（heap allocation）。数据对象可以在任一时刻，以任意次序分配和释放。通常，我们需要一个称为存储池的堆。（请不要将我们这个作为内存分配池的堆的定义与常用于从集合中检索最小元素的堆数据结构混淆。不幸的是，这两个定义均被广泛地使用。）

在某些语言里有显式分配堆空间的命令（如PL/I的allocate，Pascal和Ada中的new），但其他一些语
296 言，如Snobol 或LISP中以用户语句的执行结果来隐式地分配堆空间（如，Str = Str 'XYZ'或（CONS A B））。堆空间的分配不是特别困难——我们只是简单地一直从堆中分配空间直至堆空间耗尽为止。堆分配复杂的原因是由于空间可以按任意的次序返回到（即释放回）堆。这种情况导致了许多不同的堆释放策略，它们将在下面详细讨论。

设计语言时的第三种策略是，根本不包含堆分配机制，无论显式的还是隐式的。C语言就是这样做的，它将存储分配移到标准程序库中（如例程`malloc()`和`free()`）。这种方法允许程序库开发商提供针对特定应用而调整的运行库来替代一般的内存分配机制。然而，它却阻碍了编译器和运行时系统做出任何有效检查以确保堆空间的正确管理。

C++语言引入一个折中方案，它提供了显式的`new` 和`delete`操作符用于存储的分配与释放；但这些操作符可以被任何特定的类重载。这意味着程序员既可以选择由C++编译器和运行系统提供默认的内存管理机制，又可以自行提供应用定制的存储管理机制。

9.3.1 无空间释放

我们可以选择忽略空间释放。当空间耗尽时，程序就停止运行。一些Pascal实现（Berkeley的Pascal解释器）采用这一策略。如果大多数堆对象一旦分配便始终在使用的话，此方法也不算太差。它也可以应用于大规模虚拟内存的实现中；不再使用的数据对象不会扰乱主存储区。想节省空间的程序员仍可以通过为每一个对象类型建立一个将该类型的自由对象链接成表的DisposeObject过程来管理空间的释放。
297 而AllocateObject过程试着从自由表的表头返回空间；如表为空，则它要求助于一般的堆分配。

9.3.2 显式释放

若我们有显式的分配命令，那我们也可以有显式的释放命令（如Ada中unchecked_deallocation，C中`free()`和Pascal中的dispose）。由用户负责使用释放命令来释放不再需要的空间。堆管理程序仅是记住已释放的空间，并在分配命令发出后能使之再次可用。这种方法将最困难的决策——空间应何时释放——转移到了用户手中并可能带来灾难性的悬空指针错误（dangling pointer error）。例如，考虑图9-12中Pascal程序片段。

在指针p赋给q后，两者均指向同一对象。而在p被释放后，q成为悬空指针。通过q进行的赋值是非法的，若未检测出来，则会带来不可预知的后果。在少数的实现中，包括UW-Pascal，通过检查指针有效性来发现悬空指针的使用（参见第11章），但在其他大多数系统中不对这种错误进行检测。

```
var p,q : ↑ real;
    ...
    new(p);
    q := p;
    dispose(p);
    q↑ := 1.0;
```

图9-12 Pascal中出现的悬空指针

9.3.3 隐式释放

无论采用的是显式分配还是隐式分配，我们都可以选择*隐式释放*（implicit deallocation）——即，自动恢复不使用的堆空间。这一过程通常被称为*垃圾收集*（garbage collection）。现在有许多不同的隐式释放的方法。

单一引用

隐式释放的方法之一是要求对任何堆对象的引用（即，指向该对象的指针）决不多于一个。当这个引用改变时（例如，有关赋值完成或作用域结束），我们可以释放该对象，它的惟一引用也被撤销。

这种方法相当容易实现，但它要求不能存在多重引用。因此，它最适合用于简单数据类型，如字符串，但不适合复杂的链式数据结构（例如图、网络等等）。尽管如此，Ada/CS中的字符串实现可以利用这种方法，我们将稍后加以讨论。 298

引用计数

我们也可以通过允许多重引用一个堆对象且在每个堆对象中存放*引用计数*（reference count）的方法来进行隐式释放，如图9-13所示。引用计数表明指向该对象的指针的多少；当计数归零时，该对象可以被释放。当一个引用被建立、复制或撤销时，我们必须更新这个引用计数。在某些情况下（如循环链表），引用计数可能从不归零，某些对象也因此从不被释放。



图9-13 带有引用计数的堆对象

垃圾收集

另一种隐式释放的方法是，追踪所有现存指针，并递归标记所有可访问的堆对象。这一过程常被称为*标记-清除式垃圾收集*（mark-and-sweep garbage collection）。我们从全局指针变量和那些出现在AR中（局部于子程序）的指针变量开始。我们记录下这些指针的出处（可能要存放于辅助数据结构中）。对于可能包含指针的数据对象，尤其是记录，我们必须追踪其中的指针。

在标记阶段之后，我们知道任何未标记的对象是不可访问的并且可以被释放。随后，我们在堆中进行清除，收集未标记对象并送回自由空间供以后使用。在清除阶段，我们也去除那些尚在使用的堆对象的标记。

标记-清除式垃圾收集功能非常强大，但也十分复杂。它吸引人的地方在于，它仅在堆空间耗尽时、而非每次创建或撤销引用的时候进行工作。也就是说，除非我们用完了堆空间而去执行垃圾收集，否则我们不需付出任何代价。

在任何标记-清除模式中，至关重要的是我们必须要标记所有堆对象。如果漏掉了一个指针，我们将无法标记它为可访问堆对象，也不能在以后正确地释放它。对于像LISP这样有非常一致的数据结构的语言，发现其中所有指针并不困难，但在像Simula和Ada这样的语言中，事情却相当难办，因为那里的指针与数据结构中的其他对象混在一起，另外还存在着指向临时变量的隐式的指针，等等。为达此目的，我们必须在运行时获得大量的有关数据结构和活动记录的信息。

一个形式更简单的垃圾收集——其实仅包含堆的清除——是可能的，如果我们知道指向每个堆对象
299 仅有一个指针。该技术非常适合于长度可以动态改变且因此必须从堆中分配的Ada/CS的字符串。

与其采用非常复杂的标记过程，我们倒不如采用以下的握手协议（handshaking convention）。指针依然指向堆对象。每个堆对象都有一个回指引用它的单个指针的标题（header）域。该过程在图9-14中说明。

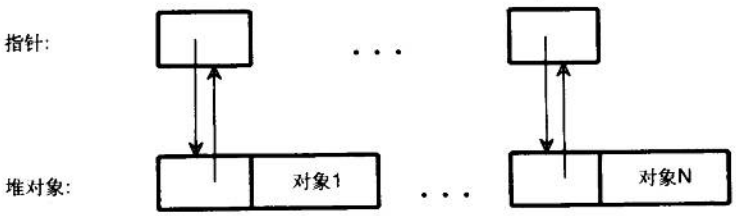


图9-14 堆对象的握手协议

在清扫堆时，我们跟随标题域中的指针并检查它引用的位置是否指回到该标题域。对所有可访问的堆对象，这种握手方式都将成功。如果一个指针p已被改变，先前由p引用的堆对象将仍然指向p，但p并没有回指，这指示该堆对象可以被释放。出现在活动记录中的指针在该AR被弹出运行时栈之后被认为是不可访问的。对任何堆对象，如果其标题域指针指向当前栈顶以上的地方，则它被认为是不可访问并可以被复用。有可能出现这样的情况，一个AR从栈中弹出，接着另一个AR被压入栈并占据与前者相同的空间。这种情况下，先前被一个指针所占据的空间现在可以为其他数据所覆盖，先前的握手关系几乎肯定遭到破坏，而这将导致堆对象能被正确地释放。还有一种极小的可能性，即，覆盖先前一个指针所占据空间的新的数据恰好与指针拥有相同的位模式。在这种不大可能发生的情况下，不可访问的堆对象将不被收集除非这个新的数据改变它的值。这其实并不是一个问题——我们只是没有收集所有不可访问的堆对象。

在收集不可访问的堆对象时，许多垃圾收集器会执行紧缩（compaction）阶段。所有仍在使用的堆对象被集中放在一起。这就允许那些不可访问的堆对象可以合并成一个单一的自由存储区，有利于分配较大的数据对象。如果不执行紧缩的话，可能出现堆空间分配器在堆中拥有许多自由存储区，但没有一个足够大的能够满足需求的区域。由于分配和释放操作，堆可能会变成碎片状的（fragmented）；紧缩操作将小的碎片连接成更大且更有用的存储块。

在标记-清除式垃圾收集过程中进行紧缩时，需要对所有可访问的指针进行第二遍扫描以便重置指
300 针，使其指向它们所引用的对象的新地址。而在采用握手协议时，事情依然很简单，因为每个堆对象中回指的那个指针的标题指针也必须重新设置。然而存在着一种危险。如前所述，在运行时栈中看似一个指针的东西实际上却是一个非指针对象，只是因为这个数据对象覆盖了曾经被一个有效指针占据的位置。此时进行紧缩，我们可能会错误地对这个不是指针的位置进行了修改。因此，较为明智的做法是紧缩那些由全局指针所引用的堆对象，而全局分配的指针是不会被其他对象所覆盖的。

9.3.4 管理堆空间

在所有存储恢复模式以及所有提供堆分配的语言中，我们必须小心应对未初始化指针（uninitialized pointer）。我们要么必须在创建每个指针时将其初始化为一个空值（Simula采用该方法），要么必须在使用指针前对其进行有效性检查（UW-Pascal采用该方法）。如果不这样做，则我们可以通过无效指针来引用（修改、或释放）几乎任何一块存储区（如程序正文段、栈空间、文字常量区、等等）。

如果同时允许分配与释放，无论它们是隐式的还是显式的，堆管理程序都应当按需分配大小可变的

空间块并接收返回的空间块。就分配而言，总有几个可以满足分配需求的空间块。*最佳适配*（best-fit）方法从中挑选那个可以使空间浪费减至最小的块，如果我们使用它的一部分用于分配。然而，这种方法通常效率不高；且因此增加了很小的，可能也是无用的自由存储块。*首次适配*（first-fit）方法挑选第一个满足需求的自由块而不管浪费多少空间。这种方法虽好，但如果每次分配总是从堆开始的地方搜索，那么较小块将趋于在堆前面的部位聚集。搜索合适的自由空间块的平均时间为自由块数量的一半。我们推荐使用的方法是*循环首次适配*（circular first fit）方法，该方法也像首次适配法那样搜索、分配自由块，只不过每次的搜索不是从堆开始的地方而是从上次搜索结束时的位置开始的。通常，仅需少许几次搜索即可完成分配。

记录堆状态的合理的数据结构是双链表，可以把指向自由块的指针组放入链表中。每个块，无论是自由块还是在使用的块，其第一个字和最后一个字被保留分别用于指示它的状态：自由或忙，和它的长度。在需要自由块时，我们从上次搜索结束的地方开始搜索双链表直至发现合适的块。如果需要，这个块也可以再分成一个新的状态为忙的块和一个新的自由块。若大小正合适，则整个块将从双链自由表中摘除。当块返回自由空间时，它试图同在它前后的块合并如果它们都是自由空间的话，然后该块加入到自由表中。这种方法被称为*边界标签*（boundary tag）。

循环首次适配是比较好而且通用的方法，但程序员们经常仅需要分配少量的不同大小的块。尽管编译器可能不知道全部块的大小（如字符串），但它还是知道不少块的大小（例如，为记录分配的空间）。因此，从不同的堆中分配大小不同的块比从惟一的全局堆中进行分配可能效率更高。于是，*位图*（bit map）被用来指示堆中哪块区域是自由的。即，在固定大小的对象堆中的每个对象均被映射为位图中的一位以表示它是已分配的还是自由的。因为块较小，所以没有浪费空间的危险。 301

9.4 内存中的程序布局

既然我们已经讨论了编译器使用的运行时存储组织，现在可以考虑一个已编译的程序在内存中是如何布局的。有许多布局模式，图9-15给出了一个适合Pascal和Ada/CS的布局图。

保留区是由目标计算机系统留作特殊用途的内存区，在一些机器上，寄存器和操作数栈会映射到保留区。编译器生成的程序代码采用静态分配且通常是只读的。文字常量池包含出现在程序中所有文字常量运行时的表示。此内存段也采用静态分配且是只读的。全局数据段包含所有的全局和静态变量。它采用静态分配且是可写的。接下来是运行时支持模块的库模块（输入/输出、内存管理、剖析和调试例程、等等）。分块编译的子程序和包也可以放在这里。这些模块中的每一个通常和主程序一样也拥有程序代码段、文字常量池和静态数据段。 302

库模块和分块编译的模块作为在主程序中出现的外部引用的结果而被链接器或加载器包含进来。链接器或加载器必须确保正确重定位在包括主程序在内的代码段中出现的所有地址引用。交叉模块（外部）引用，也必须被解析到正确的地址。链接器或加载器也负责适当地将程序段编组并有序地放入内存中。当数据段和程序代码段被生成后，它们被赋予相对某个特定重定位单元的偏移值。这些偏移定义了重定位单元的内部结构。

在静态分配的程序段被分配内存位置后，剩余的内存被划分为两个动态结构——栈和堆。程序开始运行时，栈和堆被初始化。通常，栈刚好开始于静态分配的程序段之上，向高地址方向增长。而堆则从地址最高端开始，向较低地址方向增长。栈和堆一旦相遇，会产生内存使用冲突，因此，每一个延伸栈或堆的操作都必须检查是否存在这种冲突。一旦发生冲突，则可能要调用垃圾收集程序，包括内存紧缩，来分离这两种结构。

某些计算机体系结构，如VAX和Intel 8086，采用段式存储。即，在这些机器中可用内存将分成许多不同的段，一些用于程序代码，另一些用于数据。代码段是只读的且可用于存放程序代码和文字常量池。不同的数据段可用作栈和堆；这样就消除了冲突的可能性。

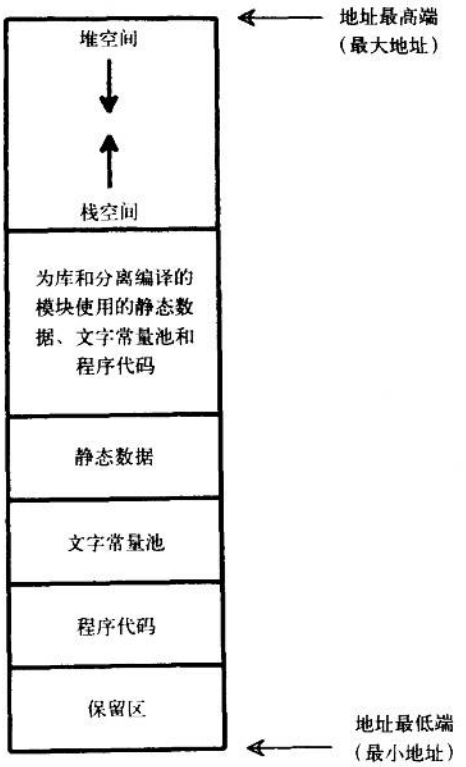


图9-15 典型的程序内存布局

某些计算机上使用操作数栈而不是寄存器来执行算术运算。可以借助运行时AR栈的栈顶来存放操作数，但这样做不是很方便，因为当计算操作数和表达式时可能必须在栈中弹出或压入活动记录。因此，更简单的做法是，如果需要，可以给单独的程序段分配一个操作数栈。如果可以确定操作数栈的最大深度，则这种分配将很容易做到。操作数通常在跨越语句时不会保留在操作数栈上，除非表达式中出现函数调用。在每条语句的代码生成后，该语句所需的栈的深度可以通过检查入栈和出栈完成
303 的顺序来获得。可以计算出函数中任意一条语句所需的栈的深度。在表达式中出现函数调用时可以使用该值。除非函数调用是递归的，否则我们可以计算出操作数栈的最大边界并用它来为操作数栈分配空间。如果发生递归函数调用，可能有必要使用活动记录栈存放操作数或为操作数栈指定一个任意大小的边界。

如果编译器被设计用于加载-运行操作，如大多数Ada/CS的工程编译器那样，内存中的程序布局将被显著改变。程序和数据将被分配地址，且它们可以在生成的时候就被加载到内存中。这里没有明显的链接和加载阶段，因此，有利于减少重定位地址和解析交叉模块引用的需求。分块编译很少被加载-运行式编译器支持。因此我们只关注以何种方式把库模块包括进来。有两种方式较有吸引力。我们可以把所有的库例程做成*自重定位*（self-relocating）的，这意味着这些例程的编写方式使得它们无论被加载到哪里都可以正确运行。（这样的代码通常被称为*位置无关的*（position-independent）代码。）这一点可以通过使所有地址引用都相对于某个基址寄存器（在运行时加载该例程的起始地址）或相对于程序计数寄存器（如果该机器体系结构使用这种寄存器的话）来做到。

交叉模块引用通过使用*转移向量*（transfer vector）来处理。从主程序中直接或间接引用的库模块中

每一个入口点都被赋予分配在文字常量池里的转移向量中的一个位置。在运行时转移向量将包含赋予相应入口点的地址。对库模块入口点的所有引用都成为通过转移向量的间接引用。我们并不生成直接转移到OpenFile例程的代码，而是将一个固定位置p赋予转移向量中的OpenFile条目并生成一个通过位置p进行的间接转移。当包含OpenFile的库模块被加载时，我们用正确的地址对p进行初始化。

如果库例程集较小，像在Pascal和Ada/CS中那样，则有可能使用一种更为简单的方式——我们简单地把它们全部加载到固定位置上。我们假定整个运行时支持库总是处于固定位置上，或许它恰好挨着该体系结构所保留的位置。因为库例程总是在相同的地方，我们可以事先对其进行重定位，这样加载到内存中的便是这些例程的可执行的二进制映像。交叉模块的引用也很简单。库例程总是被加载到相同的位置，我们可以在编译时维护一张入口点地址表。不需要间接跳转。如果我们想调用OpenFile，我们可以查找赋予它的地址并在为该调用所生成的代码中使用这个地址。

最后，如果我们在生成指令和数据时将它们加载到内存中，则无法将文字常量和全局变量放到紧临程序段之上的内存段中。问题在于，在生成文字常量和全局变量时，我们还不知道程序段将会有多大，因为整个程序还没有被编译完。作为选择，可以把文字常量和全局变量分配到同一个程序段中并从最高内存地址开始向低地址方向填充该段。和往常一样，我们从低内存地址开始填充程序段。在程序和数据段都被填充之后，剩作的空间被分给栈和堆。这将导致图9-16中的程序布局。 304

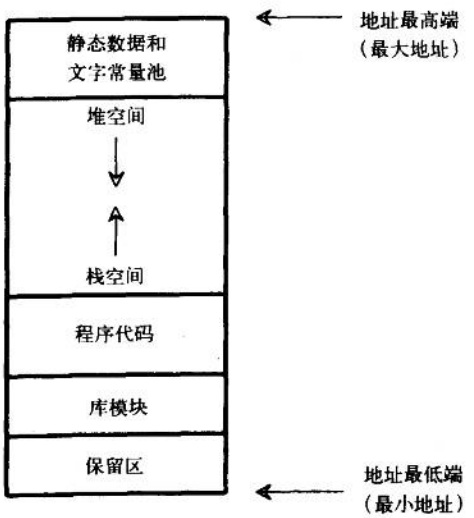


图9-16 加载-运行式编译器的程序布局

9.5 静态链簇和动态链簇

在本节中我们考虑一种可用来替代显示表访问活动记录的方法。有时我们无法或不愿意分配那么多寄存器用作显示表寄存器，作为替代，可以使用通常称为*活动寄存器*（activation register）或*活动指针*（activation pointer）的单一的寄存器，它指向运行时栈中最上面的活动记录。所有AR的设计（通常在偏移0处）包含一个指向直接包围过程所对应的AR的指针。这个指针称为**静态链**（static link），因为它反映了过程和函数的静态嵌套结构。另一个指针（也在AR中的固定偏移处）称为**动态链**（dynamic link），指向运行时栈中恰好位于当前AR下面的那个AR。

由任意AR中开始的静态链簇所包含的信息与显示表中的信息相同。事实上，一个静态链簇可被认

为是显示表的另一种实现。如果我们当前在层次为i的过程里，活动指针引用与D[i]相同的AR。进一步，此AR中的静态链指向D[i-1]所指的位置。通过从最顶端的AR追踪j次静态链，我们可以到达由D[i-j]所引用的AR。在必须访问非局部变量时，编译器生成追踪静态链适当次数的代码，而那个次数在编译
305 时可知。

动态链将例程的AR与它的调用例程的AR相链接。当从调用返回时，这些链可用来恢复活动指针。静态链序列被称为静态链簇（static chain）。类似地，动态链序列被称为动态链簇（dynamic chain）。

例如，考虑图9-17所示的程序框架。

```
1   program Main is
2       procedure P is
3           procedure R is
4               -- Body of R
5           end R;
6       begin -- P
7           R;
8       end P;
9
10      procedure Q is
11      begin
12          P;
13      end Q;
14
15  begin -- Main
16      Q;
17  end Main;
```

图9-17 一个程序框架

从Main开始，Q被调用，Q调用P，而P又调用R。当我们处于第4行的R中时，得到如图9-18所示的由静态链簇和动态链簇链接在一起的AR序列。静态链总是指示当前过程静态地嵌套其中的那个过程所拥有的AR。动态链总是指示动态地调用当前过程的过程所拥有的AR。

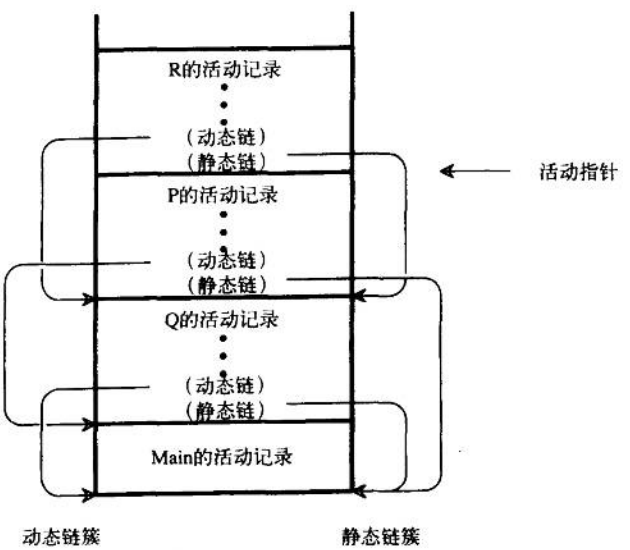


图9-18 静态链簇和动态链簇的示例

使用静态链簇可能会比显示表引用数据更慢，因为必须追踪静态链簇。幸运的是，通常这不像它最

初看起来那么昂贵。在许多程序中，多数的数据引用要么是指向局部数据，要么指向全局数据（即，指向最内层或最外层活动记录），而且这种引用特别有效。例如，对Simula 67程序所做的一个广泛的引用模式的研究（Magnusson 1982）发现全部引用中至少80%是对处于最外层的变量或局部变量的引用。另外17%的引用是对直接外围块中的局部变量的引用。局部数据由活动指针所指向；全局数据是静态的，实际上不需要通过静态链簇引用。因此，只有很少的引用实际上需要追踪静态链簇，并且其中的大多数引用将仅需要一层这样的间接访问。一些设计用来支持类Algol语言的机器提供对静态链簇的硬件支持。特别地，在这种机器上，数据通过(ChainLength, Offset)对来寻址访问。Offset，和以前一样，是相对于AR开始位置的偏移。ChainLength是为得到正确的AR的所必需的追踪静态链簇的次数 306
(ChainLength = 0意味着使用由活动指针所指的最顶层的AR)。该寻址模式非常便于使用；给定上述引用模式以及硬件支持，它几乎可以和显示表机制同样高效。然而，这种或其他要求通过内存引用而不是寄存器引用来获取AR指针的机制在某种程度上总是效率不高的。

9.6 形式过程

形式过程就是将子程序作为参数传递给另一个子程序。子程序调用的实现细节将在第13章里讨论。在本节中，我们将关注形式过程对通过显示表或静态链簇和动态链簇进行的AR访问的影响。

形式过程参数实现时有一定的技巧，因为每一个形式过程必须携带相关的环境（environment）。这个环境就是形式过程执行所必需的可访问的AR集合。有时我们称这个包含环境的AR集合为闭包（closure）。一个环境在子程序首次创建时即被绑定，即，在它的声明被详细描述的时候。声明的详细描述发生在程序执行、控制流进入包含它的块或过程的时候。这个环境可能与子程序最终通过它的形式过 307
程名字来调用时的环境有很大的差别。使用子程序创建时定义的引用环境，我们在术语上称为静态绑定（static binding）；使用最后调用点的环境在术语上则称为动态绑定（dynamic binding）。Algol 60和它后继语言都采用静态绑定。早期的LISP实现采用动态绑定，而后来的一些实现，包括Common LISP和Scheme，采用静态绑定。

```
1   B0: begin
2           procedure A(F); procedure F;
3           begin
4                   F(1)
5           end;
6           procedure B;
7           begin
8                   procedure G(X); integer X;
9                   begin
10                          ...
11                          goto L;
12                          ...
13                  end;
14                  A(G);
15          L:      ;
16          end;
17          B;
18  end;
```

图9-19 形式过程参数的示例

考虑图9-19中的Algol 60程序片段。当实参过程G在第14行绑定到形式过程F的时候，它的环境，绑定于G在B的入口处创建的时候，是B0和B。当它作为形式过程F在第4行被调用时，此时使用的环境却是B0和A。但是，当G执行时它必须访问B的局部变量，即使它们在它的调用点（第4行）是不可访问的。因此，我们必须能改变在形式过程被调用时的环境并在返回时恢复它们。

我们甚至还必须处理跳出形式过程的**goto**语句，如第11行所示。**goto**语句要想成为合法的，它的跳转目标就必须在执行环境中可见，但是运行时栈上可能有目标AR的多个实例。这些实例中只有一个在当前环境，且这个是应当继续执行的那个。所有其他中间的AR必须被抛弃就像它们的过程已经返回一样。（我们将会看到，在执行非局部**goto**后，有关寄存器与显示表的重建可能非常复杂。）

308 巧合的是，静态链簇使形式过程的实现相当简单；显示表则反之。我们将讨论这两种方式。

9.6.1 静态链簇

我们已经了解了如何建立静态链簇。当一个常规的（非形式的）过程被调用时，编译器知道调用者和被调者之间的词法层次差。这个差值表明被调者的静态链簇应指向哪里。如果差值d为-1（被调者比调用者在层次上要深1层），则被调者的静态链簇应指向调用者。否则，被调者的静态链簇是调用者的静态链簇在去除d个链之后的一个拷贝。

对于形式过程而言，它的静态链簇不能在编译时确定。相反，我们要求将静态链簇作为形式过程的一部分来传递。于是，当形式过程最终被调用时，它的静态链簇就已经可用了，且存放在调用者活动记录中作为调用者所知晓的此参数信息的一部分。

当一个过程首次作为形式过程传递时，如A将B传递给C，这里的实参B是常规的过程。和以前一样，编译器计算调用者A和B之间的词法层次差，且生成那些只有在直接调用B时才使用的可建立B的静态链的代码。调用者A将B的代码指针和那个B的静态链一起传递给C。而C把这两个有关B的重要信息保留在自己的AR中。如果C用相应的形参名字调用B，则它会把它保留的静态链交给B；如果C再次将B传递另外的例程，那么它也必须一并传递B的静态链。

再次考虑图9-19中的示例。开始的时候，B0执行；接着调用B，进入程序的第14行。此时运行时栈的情况见图9-20。

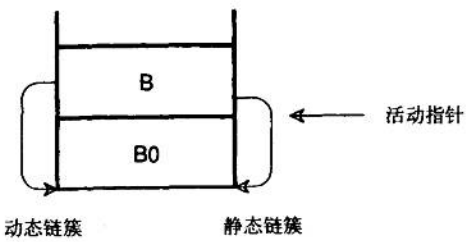


图9-20 调用B后的运行时栈

G的环境由指向B的AR的指针表示。然后，A被调用，进入程序的第4行。现在运行时栈的情况由图9-21所示。

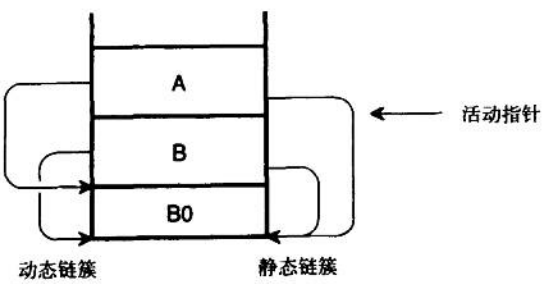


图9-21 调用A后的运行时栈

这里，通过对应的形参名字F来调用的G即将被调用。G的指向B的AR的静态指针可用来建立静态链，如图9-22所示。在正常返回时，我们依然使用动态链来恢复调用者的环境。

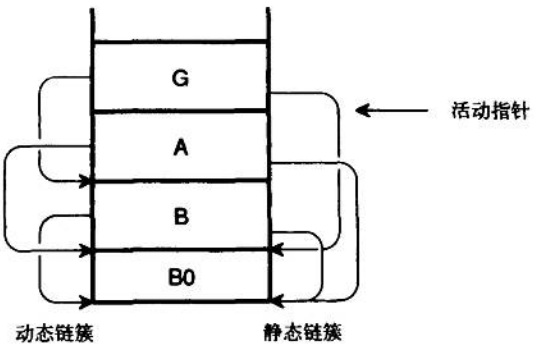


图9-22 调用G后的运行时栈

从G中跳回到B里的语句**goto** L使用静态链来寻找正确的B的实例（这里，只能有一个，但一般情况下，可能有多个）。**goto**执行后，运行时栈的情况如图9-23所示。 309

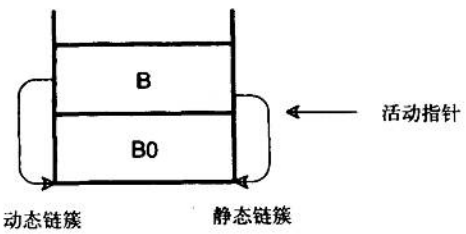


图9-23 **goto** L执行后的运行时栈 310

这种实现形式过程的方法简单、有效。然而，它却要忍受静态链簇所特有的低效率访问。此时，过程调用不会比以前更复杂；导出和传递静态指针也不会比保存一个显示表寄存器需要做更多的工作。

9.6.2 显示表

我们可以修改显示表机制以支持形式过程。当一个过程被绑定到形式参数时，我们将所有显示表寄存器的内容连同该过程的地址一起传递，如图9-24所示。所传递的数据块大小由词法嵌套的最大深度决定，而这受限于显示表寄存器数量。另外，还有一个名为restore_AR的全局变量或寄存器用来从形式过程返回时恢复调用者的环境。现在，就在形式过程被调用前，我们可以：

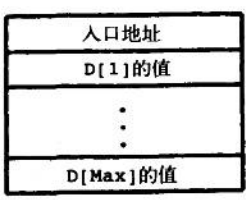


图9-24 使用显示表的形式过程描述符

- 把restore_AR的当前值保存到调用者的AR中。
- 把显示表寄存器（D[0],…,D[Max]）的当前值保存到调用者的AR中。
- 让restore_AR指向当前AR。
- 将当前显示表的值替换为先前传入的形式过程描述符中的显示表的值。
- 通过入口地址调用形式过程。

从形式过程返回时，我们可以：

- 利用restore_AR重新装入调用者的显示表。
- 恢复先前的restore_AR的值。

形式过程的开销（交换整个显示表）也仅当在我们使用它的时候才会有。普通的过程调用其表现就好像形式过程不存在一样。

311 只要每个形式过程能正常返回，上述方法就是可行的。为此，我们要么不允许**goto**语句跳出子程序，那样子程序在它被调用时就不必知道（或关心）它是作为形式过程还是普通过程被调用的；要么我们必须以某种代价来扩展相应的技术以处理跳出子程序的**goto**语句。这种扩展可通过采用静态链簇来保存和恢复显示表的混合式方法来实现。但是，这会增加普通过程调用的开销。

另外一种允许**goto**语句跳出形式过程的方法是去模拟一系列的过程返回。**goto**语句执行后的局部环境的AR所对应的运行时栈中的位置总是很容易确定。如果标号L在词法层次i处声明，那么语句**goto** L执行时必须将其执行后的局部环境的AR恢复为由D[i]指向的AR。因此，**goto**语句的代码将反复执行过程返回所需的动作直到活动指针指向正确的AR。其中一些返回动作较简单，仅涉及恢复一个被保存的显示表寄存器。然而，那些从形式过程的返回将涉及恢复整个显示表。这种技术会使跳出过程的**goto**语句执行得很慢，但是非局部的**goto**总得承担恢复寄存器和显示表值的开销，且这笔开销经常不会太小。总之，非局部**goto**一般仅被程序员用于异常情况中，在那里执行速度已不是那么至关紧要的了（例如，在出错事件中放弃某个操作）。

9.6.3 一些看法

形式过程中复杂的环境交换暗示着这样的过程一般都是很难理解的。例如，图9-25中的Pascal程序会打印什么样的答案呢？

```
program Main(Output);
    function f(Level : Integer; function Arg : Integer) : Integer;
        function Local : Integer;
        begin {Local}
            Local := Level
        end; {Local}
    begin {f}
        if Level > 10 then
            f := f(Level - 1, Local)
        else if Level > 1 then
            f := f(Level - 1, Arg)
        else
            f := Arg { actually call Arg() }

    end; {f}

    function Dummy : integer;
    begin {Just a placeholder} end;

begin {Main}
    writeln('The answer is:',f(13,Dummy));
end.
```

图9-25 使用形式过程的一个Pascal程序

F由Main调用，然后递归调用自己12次。在第12次调用时，它返回调用Arg时得到的值，这里的Arg绑定到局部过程Local的某次出现。Local访问非局部变量Level。根据类-Algol语言常用的静态作用域规则，这个非局部引用必须找到声明在过程f中的实例。但在Level的13个实例中，哪一个是可用的呢？不要被静态和动态作用域规则之间的区别弄糊涂了！静态作用域规则提到当一个同名变量有多处声明时，我们所要找的是最接近（closest）而非近期（most recent）声明的变量。这里，只能有一个Level声明符合要求。若采用近期的Level出现将导致一个“近期”错误。Local实例中使用的Level，必须等到包含

该它的Local声明清晰后——即控制流进入该Local的时候，才能有效绑定。当这个Local出现被选择用来作为参数传递时，它同时携带了相关环境以及所绑定的Level。最终得以调用的那个Local版本，它所获得的Level值为11。因此，这个程序打印11。

C语言不允许子程序嵌套在其他子程序中。但这种限制同时赋予了一些子程序在其他语言中所没有的更大的可见性，也确实有助于了解和实现C语言中的形式过程。作为参数传递的子程序不需要携带相关环境，因为它们仅可以访问所有子程序共享的全局变量或建立于子程序活动时的局部变量。（事实上，在C语言中没有上述那样的形式过程参数。相反地，那些参数是具有“函数指针”类型的简单变量。）Ada语言甚至比C语言更严格，它不允许有形式过程。限制作为形式过程的子程序不能嵌套在其他子程序中将会允许更多有用的形式过程的应用而且也避免了维护正确执行环境时的内在的复杂性。 312

练习

1. 程序设计语言提供了各种数据对象的构造器。为下面描述的各类数据对象设计最合适的运行时的存储组织。

   list of T;

   T为任意类型名。表可以使用append操作进行连接；使用head和tail操作进行拆分。 313

   set of Lower .. Upper;

   Lower和Upper是常量值。这其实是Pascal提供的集合构造器。

   set of Lower .. Upper;

   Lower和Upper是在运行时明确集合声明的时候计算的表达式。

   set of T;

   T是任意类型名。

   ExtendedInt;

   ExtendedInt是精度扩展的整数。它没有MaxInt或MinInt限制；相反，其精度表示可根据需要进行扩展以容纳任意值。

   file of T;

   T是除文件（或包含文件的结构）类型之外的任意类型。这是Pascal提供的文件构造器。

   String(N);

   N是常量值。串的长度可以介于0和N之间。

   String(N);

   N是在运行时明确串的声明的时候计算的表达式。串的实际长度介于0和N之间。这其实是PL/I提供的串构造器。

2. 给出下面例程的过程级和块级AR的布局图。

```
procedure q(a : integer; b: real) is
  c : array(1..N) of real;
  d : string;
begin
  declare
    e : array(1..10) of integer;
  begin
    ...
  end;
  declare
```

```
    f : array(1..M) of integer;
    g : integer;
314 begin
      ...
    end;
  end q;
```

3. 跟踪以下程序执行时的AR序列以及所需的显示表操作。

```
procedure main is

  function m(i : integer) return integer is

    function p(i : integer) return integer is
    begin
      return m(i * 2 + 1);
    end p;

    function q(i : integer) return integer is
    begin
      return m(i + 111);
    end q;

    function r(i : integer) return integer is
    begin
      return i * 3 ;
    end r;

    begin
      case i mod 3 is
        when 0 => return p(i / 3);
        when 1 => return q(i / 3);
        when 2 => return r(i / 3);
      end case;
    end m;

begin
  write(m(157));
end main;
```

4. 用静态链和动态链方式给出在练习3中在函数r活跃时运行时栈上的AR序列。
5. 在一些计算机系统上有可能扩展最大可访问的内存地址，从而增加有效内存的大小。当程序因动态存储结构生长而用尽自由空间时，这种内存扩展操作就显得格外有意义了。为了获得该扩展，你将
315 如何在内存中放置程序呢？
6. 在9.3.3节中识别可访问堆对象的握手方案不能安全地进行堆紧缩。提出一个允许堆紧缩的一般化的握手方案。提示：将两方握手推广到三方握手。
7. 许多编译器分配固定的寄存器集合用于显示表。有时过多地分配会造成浪费，而太少的分配又会限制能编译成功的程序种类。

   描述一个算法，编译器可用它来确定一个特定程序恰好需要的显示表寄存器数。这个算法若用在一遍编译器中会出现什么问题？
8. 假设我们组织的堆中的每个对象只允许由一个指针指向。当指向一个对象的指针改变时，应做哪些操作呢？在打开或关闭作用域时应做什么样的操作？能允许指针或堆对象的赋值吗？
9. 假设我们采用引用计数来组织堆。当分派指向堆对象的指针时，应做哪些操作？在打开或关闭作用域时应做什么操作呢？
10. 包括C语言在内的一些语言有创建指向数据对象指针的操作。如，

    ```
    p = &x;
    ```

    取类型为t的对象x的地址赋给指针p，而p的类型则为t*。

    如果可以创建指向AR里任意数据对象的指针，那么运行时栈的管理会复杂到什么程度？

为确保运行时栈的完整性，需要对指向数据对象的指针的创建和拷贝做哪些约束？

11. 假设我们不是维护单个的堆，而是为每种类型T都维护一个不同的可动态分配的子堆。这种维护子堆的做法有什么优缺点？子堆将如何简化不可访问对象的回收？
12. 考虑一种称为*最差适配*（worst-fit）的堆分配策略。不像最佳适配方法是从自由空间块中分配与需求大小最接近的块，最差适配方法选择最大的可用自由块进行分配。与最佳适配、首次适配和循环首次适配等分配策略相比，最差适配有什么优缺点？
13. 复杂算法的性能评测往往通过模拟它们的行为来进行。编写程序模拟一系列随机的堆分配和释放。比较最佳适配、首次适配和循环首次适配等技术分配堆对象的平均搜索次数。
14. 一些程序设计语言提供以下描述程序步并发执行的结构：

```
cobegin <stmt 1> | <stmt 2> | ··· | <stmt n> coend
```

316

语句<stmt 1>, ···, <stmt n>可以以任意次序并发执行。在**cobegin**结构中的每条语句里，可以使用普通的**begin-end**语句块来强制顺序执行。

普通的运行时栈能否满足含有**cobegin**结构的语言？如果不能，应该如何扩充运行时栈来支持**cobegin**中的并发或者交错执行次序？

15. 假设使用显示表交换的方法实现形式过程。跟踪下面Pascal程序执行时的AR序列以及所需的显示表操作。

```
program prog(output);

procedure q(procedure c(var i:integer));
var z : integer;
begin
   z := 17;
   c(z);
end;

procedure p(procedure a(procedure x(var j:integer));
      procedure b(var k:integer));
begin
   a(b);
end;

procedure r;
var i : integer;
   procedure s(var j:integer);
   begin
      writeln(j + i)
   end;
begin
    i := 10;
    p(q,s);
end;

begin
    r;
end.
```

现在假设使用静态/动态链簇来实现形式过程。再次跟踪上面程序执行时AR序列以及所需的静态/动态链簇操作。

317

16. 说明用来实现形式过程的显示表交换方法不能用于下面的Pascal程序：

```
program prog(output);

procedure q;
label 1;
   procedure exec(procedure z);
   begin
```

```
        z;
    end;

    procedure r;
    begin
        goto 1;
    end;
begin   exec(r);
1:
end;

procedure p(procedure a);
var v : integer;
begin
    v := 10;
    a;
    writeln(v);
end;

begin
    p(q);
end.
```

318 静态/动态链簇方法是否可以正确处理上面的程序?

第10章 处理声明

本章中主要有三节内容。第一节给出处理声明的基本技术，包括表示声明的各种结构以及在处理声明时语义栈的特殊使用方式。第二节讲述处理Ada/CS语言中各种声明的一个简单子集所需的语义例程。这个子集包括变量和类型名的声明，同时也包括记录和静态数组的类型定义。这些语义例程的概述是采用基于ANSI C的伪码编写的，因此对于熟悉ANSI C的人来说，它们应该是非常易读的。（在前言中已简 319
述了对ANSI C的扩展。）第三节包括一些类似的用于Ada/CS语言其他部分的语义例程的概述。在后续的三章里，将继续使用语义例程概述这种手段来讨论语言特性的编译。尽管这种表示是基于某个特定语言的特性，但所介绍的技术将尽可能地通用以期编译范围更广的语言。

注意以下术语的称谓。术语“记录”（record）在从Algol派生出来的语言中使用，而在C语言中则称之为“结构”（structure）。“变体记录”（variant record）在C语言中称为“联合”（union），而一个的变体对应于一个C语言的联合中的一个成员。特别地，C语言的union等价于Pascal的无标签变体。C语言中没有与Pascal或Ada语言中的有标签变体等价的类型。程序员应该知道在任何特定时刻一个联合中有什么。

10.1 声明处理的基本原则

10.1.1 符号表中的属性

在第8章，符号表是作为将名字和一些*属性信息*（attribute information）相关联的一种手段。那时，我们不考虑与名字相关联的属性中包含什么样的信息或它们的表示方式。现在，这些问题将在本节里予以讨论。

名字的属性通常包括编译器所知道的关于该名字的任何东西。因为编译器获取名字相关信息的主要来源是名字的声明，所以属性可被认为是声明的内部表示形式。编译器在内部确实生成某些属性信息，通常，在名字声明出现的上下文或者在一个隐式声明的名字的使用处生成属性信息。在现代程序设计语言中，名字有许多不同的使用方式，这其中包括变量、常量、类型和过程。因此，每个名字均拥有不同的与之关联的属性集合。当然，这些集合中的属性分别对应着名字的声明和使用。

图10-1中的记录定义揭示了可移植Pascal编译器——Pascal P-4中使用的名字和属性表示的样式，那曾是许多Pascal实现的基础。根据名字的不同使用方式，它采用了变体记录来处理必须与名字相关联的各类属性。

Name、Llink和Rlink域用来实现符号表。如注释所言，这个可移植的Pascal编译器采用二叉树形式的符号表。这样，IdPtr被定义为指向标识符记录的指针。其他的各个域用来记录符号表中标识符的属性。第一个属性域IdType被声明为TypePtr，是指向TypeDescriptor记录的指针，这个记录是另一个重要的结构，我们将简要地讨论它。跟在IdType之后是标识符记录的变体部分，那里存放着各种不同标识符的 320
适当的信息。由于研究这个可移植Pascal编译器不是我们的目的，因此我们也就不必考虑这个记录的所有细节。尽管如此，读者在阅读完本章及后续章节所展示的Ada/CS语言的编译技术后，应该会很容易地明白它们。过程和函数名字的变体值得关注，因为它包含一个嵌套的变体以区别标准例程和那些在程序中声明的例程。根据所需存储的属性信息的复杂度，我们可以采用任意复杂的结构来表示它们。

```
type Identifier =
  record
      Name : Alpha;              (* the identifier represented here *)
      Llink, Rlink : IdPtr;      (* to implement a binary tree symbol table *)
    (* The previous fields implement the symbol table search mechanism; *)
    (* those that follow describe the attributes of Name  *)
      IdType : TypePtr;
      Next : IdPtr;              (* used to construct lists of identifiers *)
      case Class : IdClass of
        Constant: (Value : ValueType);
        Variable : (Vkind : IdKind;
                    Vlevel : LevelRange;
                    Vaddr : AddressRange);
        Field: (FieldOffset : AddressRange);
        Proc, Func: (
          case PFDeclKind : DeclKind of
              Standard: ( ··· );
              Declared: ( ··· ))
end ;
```

图10-1 Pascal P-4编译器中的符号表记录

在概述处理Ada/CS声明的语义例程时，我们给这个正在编译的语言定义一个类似的属性记录。我们的记录和前面的Pascal记录的重大区别是，我们将符号表的实现和属性分离开来，并把它们放置在不同的结构中。如果语言允许一个名字在单个作用域中有多重定义，如Ada语言通过重载特性所做的，那么该语言的编译器就必须将单个名字与多个属性集合关联起来。通过将每个属性集合放在单独的结构中并且建立这些结构的链表来表示名字的重载使用，可以很容易地满足这种需求。

10.1.2 类型描述符结构

在图10-1中的记录所示的名字的属性中，有一个是名字的类型信息，它包括在一个通过指针指向的TypeDescriptor记录里。表示一个类型给编译器的设计者提出了和表示一个属性同样的问题：有许多不
321 同的类型，它们的描述也需要不同的信息。再一次地，我们使用一个如图10-2所示样式的变体记录（C语言中，是一个union），它也是基于Pascal P-4结构的。

```
type TypeDescriptor =
  record
      Size : AddressRange;
      PackedFlag : boolean;
      case Form : TypeForm of
        Scalar:      (case ScalarKind : DeclKind of
                      Declared: (FirstConst : IdPtr);
                      Standard: () );
        Subrange:    (RangeBaseType : TypePtr;
                      Min, Max : Value);
        Pointer:     (PtrBaseType : TypePtr);
        SetType:     (SetBaseType : TypePtr);
        ArrayType:   (IndexType, ElementType : TypePtr);
        RecordType:  (FirstField : IdPtr);
        FileType:    (FileBaseType : TypePtr)
  end;
```

322 图10-2 Pascal P-4编译器的类型描述符

TypeDescriptor记录开始于给出类型大小的域以及指示是否压缩的标志域。这些信息是所有类型都有的。此记录的其他部分则是一个变体部分，如何变化取决于所描述的类型的种类。该记录重要的一点就是它的变体部分中几乎每个域都是一个指针。例如，枚举（标量）类型被包含该类型常量值的Identifier记录列表所描述。记录（record）被可用于查找记录域的二叉树所描述，就像为其他标识符所

做的那样。数组通过指向其下标和元素类型的TypeDescriptor的指针来表示。这样，类型通常不是采用单个的记录而是使用由指针构造的可扩展的数据结构来描述。这种表示在处理像C、Modula-2、Pascal和Ada等允许使用强有力的组合规则来构造类型的语言时是至关重要的。采用这种技术而非某些固定的表格式表示，还可以让编译器在处理程序员的声明方面更加灵活。例如，有了这种技术，也就不需要限制数组中所允许的维数，或者记录中所允许的域的个数。在某些语言（如FORTRAN）中有这样的限制纯粹是出于实现上的考虑。我们通常乐于使用这样一些技术，即这些技术能够使编译器避免因为一个程序的大小或它的某个部分而拒绝这个合法的程序。由TypeDescriptor记录所暗示的动态链接结构的使用是这种技术的重要基础。

10.1.3 语义栈中的列表结构

在Ada/CS和其他一些程序设计语言中的很多构造均涉及到标识符列表、表达式列表等。其中有些列表（如声明列表）没有语义上的含义。各种声明均是单独处理的。而其他一些列表（如Pascal过程调用中的实参表达式列表）就有语义上的意义，当语法分析器识别列表项时，它们可以被逐项地处理，这是因为从列表所出现的上下文中编译器已获得足够可用的信息。在上述任何一种情况下，都不需要显式地从语义栈上收集列表项。

有时，在列表项上的大多数语义处理工作必须等到收集整个列表后才能完成。下面的Ada/CS的产生式描述了这样的问题。

<object declaration> → <id list> : <object tail>

在处理<object tail>过程中收集到的语义信息是用来给<id list>中的标识符构造属性记录。收集这些在语义栈上的标识符也只有等到整个声明分析完后才有意义。如果我们定义一个名为MARK的语义栈记录类型，这一点将很容易做到（通过动作控制的语义栈）。一个MARK记录就像ERROR记录一样不包含任何信息。它在语义栈中特定位置上的出现传送着所有需要的信息。我们也需要供标识符选择的一个语义记录。在第7章所讨论的Micro编译器的语义记录中，标识符串存放在表达式记录里。在更复杂的语言中，所有的标识符并不解释为表达式，因此，就需要一个单独的语义记录的变体。

带有动作符号的<id list>的产生式如下：

```
<id list> → #push_mark <id> { , <id> }
<id>      → IDENTIFIER #process_id
```

上述语义例程仅需要简单的动作：

```
push_mark(void)
{
   Push a MARK record onto the semantic stack.
}

process_id(void)
{
   Push an ID record onto the semantic stack, containing
      the token string of the identifier returned by the
      scanner.
}
```

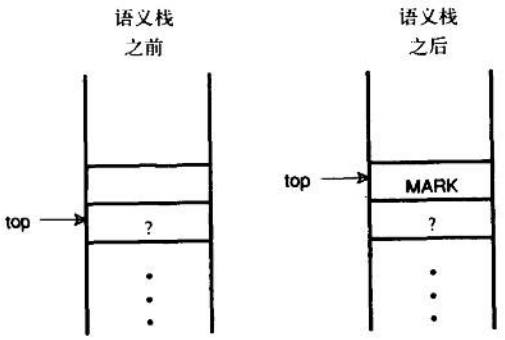
323

图10-3a显示了push_mark()在语义栈上的执行效果。图10-3b则给出了处理<id list>的一部分或全部之后的语义栈格局。

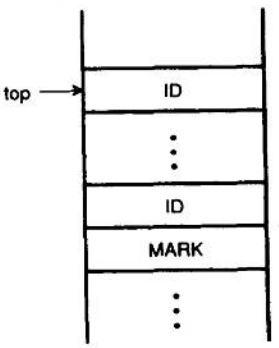
MARK记录用作列表的分界符。那些使用<id list>的语义例程可以深入栈中处理标识符直至遇到MARK记录。这种技术的主要思想就是：任意长的列表，即使它只对应着文法中一个非终结符，它仍可以通过栈中一系列的条目来表示。

此项技术的主要优点在于它的简单性。其主要缺点是：如果用数组实现语义栈，则非常长的列表会

引起栈的溢出。尽管这个问题在处理较长的标识符声明列表时不大可能发生，但是在处理普遍存在的有大量选择分支的情况语句时还是有可能发生的。因此，为应对这种情况，我们必须考虑另外的处理列表的办法。一种有效的办法是由语义栈中一个记录来表示列表，并建立显式的链接结构以包括列表的元素。以<id list>为例，在处理部分或全部列表后，我们可以得到如图10-4所示的结构。



a) push_mark()在语义栈上的执行效果



b) 调用process_id()之后的语义栈格局

324

图 10-3

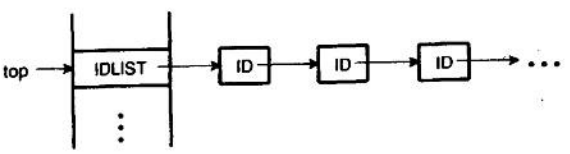


图10-4 另一种<id list>表示

除了处理溢出问题以外，这项技术还与由分析器控制的语义栈的使用相兼容，而将列表放在栈中的那些技术则不行。在下面<id list>的产生式中，我们将动作符号摆放在略微不同的位置；非终结符<id>没有出现在这个产生式中，因为由它的产生式触发的动作例程直接把一个ID记录放在了栈上：

<id list> → IDENTIFIER #start_id_list { , IDENTIFIER #next_id }

相应的动作例程显示在下面。在这个例子中，我们引入了新的符号，因此动作例程不需要显式地引用语义栈。文法符号（终结符和非终结符）就像参数一样出现在例程名后的括号内，与这些文法符号相关联的语义记录用作动作例程的参数。如果语义例程输出的结果是语义记录，则与之关联的文法符号跟

在符号=>之后。动作例程中的赋值由←表示。这个符号使得动作例程独立于所使用的语义栈，无论这个语义栈是由动作控制的还是由分析器控制的。在许多场合中，尤其是在考虑表达式和语句时，如果使用了抽象语法树的表现形式，这些例程甚至可以作为树属性计算的例程。

```
/*
 * Since identifier lists can be either in the semantic
 * stack or in a list, these two routines are purposely
 * vague about adding new identifiers to the list.
 */

start_id_list(IDENTIFIER) => <id list>
{
   Create an ID record for the token string
      corresponding to IDENTIFIER
   <id list> ← an IDLIST record containing a
      pointer to that ID record
}

next_id(<id list>, IDENTIFIER) => <id list>
{
   Create an ID record for the token string
      corresponding to IDENTIFIER
   Add this record to the list of such records in the
      IDLIST record
   <id list> ← the updated IDLIST record
}
```

325

start_id_list()产生如图10-4所示的列表的首记录并把第一个标识符放在列表中。next_id()的调用则把后续的标识符添加到列表中。

10.2 简单声明的动作例程

现在我们开始研究编写Ada/CS编译器所需的技术。这一节首先讨论Ada/CS声明的一个简单子集的处理技术。

10.2.1 变量声明

在开始研究声明的处理之前，我们先考虑诸如在Pascal和Modula-2等语言中出现的变量声明。和Ada和Ada/CS等语言不同，这些语言中变量和常量声明的语法形式均不相同，而且变量声明可以不包括初始化规范。因此，这种变量声明的语法很简单：

<variable declaration> → <id list> : <type> #var_decl

326

下面所示的记录用在语义栈上表示类型。这种记录的实例将由分析<type>时所调用的语义例程放在语义栈上。

```
struct type_ref {
   type_descriptor *object_type;
};
```

现在我们给出能满足处理声明子集特性的属性和类型描述符记录。这些记录很像前面讨论过的Pascal记录，但它们更适于处理Ada/CS的特性。稍后会讨论使用它们处理各类声明时的细节。数组变体中使用的struct range将在处理数组声明的那一节里定义。（这是第一次将匿名结构（anonymous structure）用作匿名联合的成员。）

图10-5、图10-6分别给出了处理同一个Ada/CS子集的合适的属性定义和semantic_ record结构声明。

在阅读本章和第11~13章的语义例程时，记住其中使用的终结符或非终结符名（像<id list>）表示struct semantic_record。

```
#include "symtab.h"   /* see Chapter 8 */
typedef short boolean;
enum id_class { VARIABLE, FIELD, TYPENAME };
enum type_form { INTEGERTYPE, FLOATTYPE, STRINGTYPE,
   ARRAYTYPE, RECORDTYPE, ERRORTYPE };
typedef unsigned long address_range;
struct address {
   short var_level;
   address_range var_offset;
   boolean indirect;
};
typedef struct attributes {
   id_entry id;
   struct type_des *id_type;
   enum id_class class;
   union {
      /* class == VARIABLE */
      struct address var_address;
      /* class == FIELD */
      struct {
         address_range field_offset;
         struct attributes *next_field;
      };
      /* class == TYPENAME --- empty variant */
   };
} attributes;
typedef struct type_des {
   address_range size;
   enum type_form form;
   union {
      /* form == INTEGERTYPE, FLOATTYPE, STRINGTYPE,
                 ERRORTYPE -- empty variant */
      /* form == ARRAYTYPE */
      struct {
         struct range bounds;
         struct type_des *element_type;
      };
      /* form == RECORDTYPE */
      struct {
         symbol_table fields;
         attributes *field_list;
      };
   };
} type_descriptor;
```

图10-5 属性定义

```
struct id { string id; };
struct type_ref { type_descriptor *object_type; };
struct record_def {
   type_descriptor *this_type;
   address_range next_offset;
};
struct range { long lower, upper; };
enum semantic_record_kind { ID, TYPEREF, RANGE,
     CONSTOPTION, RECORDDEF, MARK, ERROR };
struct semantic_record {
   enum semantic_record_kind record_kind;
   union {
     struct id id;                  /* ID */
     struct type_ref type_ref;      /* TYPEREF */
     struct range range;            /* RANGE */
     struct const_option const_option; /* CONSTOPTION */
     struct record_def record_def;     /* RECORDDEF */
     /* empty variant */            /* MARK */
     /* empty variant */            /* ERROR */
   };
};
        ...
```

图10-6 语义记录类型声明

仅有一个语义动作符号出现在变量声明的产生式中，但相应的语义例程却依赖许多其他语义动作的结果。该语义动作的参数包括一个在分析<type>时调用语义动作所产生的TYPEREF语义记录以及<id list>，后者可由在本章前面所讨论的两种方法之一来表示。var_decl()把列表中所有的标识符填入符号表并为它们建立相应的属性记录。在一遍编译器中，此时将分配变量在活动记录中的位置。我们可以通过产生类型为struct address的属性来达到此目的，该属性包括当前过程的嵌套层次、每个特定变量的偏移以及间接标志（对于简单变量而言，此标志总为FALSE）：

```
var_decl(<id list>, <type>)
{
   for (each identifier in <id list>) {
      Call enter() to put the identifier in the
         current scope of the symbol table
      if (it is already there) {
         generate an appropriate error message
         continue;
      }
      Allocate storage for the variable, recording its
         offset in the local variable offset
      (The size of the block of storage to allocate is
      obtained from <type>.type_ref.object_type.size)
      The following expression describes the attribute
      record to be created for the variable:
      (attributes) {
         .class = VARIABLE;
         .id_type = <type>.type_ref.object_type;
         .id = the id_entry returned by enter();
         .var_address = (struct address) {
            .var_level = current_nesting_level;
            .var_offset = offset;
            .indirect = FALSE;
         }
      }
   }
}
```

327
329

10.2.2 类型定义、声明和引用

在语义例程var_decl()中，我们假设<type>的产生式包含对能够产生TYPEREF语义记录的语义例程的调用。我们现在可以检查那些语义例程以了解TYPEREF记录的产生过程。相关的语法是：

```
<type>              → <type name>
<type>              → <type definition>
<type name>         → <id> #type_reference
<type definition>   → <record type definition>
<type definition>   → <array type definition>
```

类型定义指出采用Ada/CS类型构造器机制之一来构造新类型的过程。在类型定义产生式中的语义例程实际创建类型描述符并建立引用以上描述符的TYPEREF语义记录。10.2.3节和10.2.4节将研究记录和数组的这种处理。

当<type name>候选产生式被用于<type>时，其中的标识符要么必须是预定义的类型名，要么必须是在类型声明中出现在前面的名字。无论是哪一种情况，都必须由语义例程type_reference()在符号表中找到相应的type_descriptor引用。编译器必须在其初始化阶段为预定义类型建立合适的类型描述符和符号表条目。在Ada/CS中，Integer、Float和String是预定义类型名。我们不久将看到它们的类型描述符的样子。

类型声明就是给名字赋予类型的语法机制。类型声明的语法是：

```
<type declaration> → type <id> is <type definition> #type_decl
```

处理类型声明不需要新的语义记录类型。检查图10-5中的属性记录，我们可以发现其中类型标识符所对应的是空变体。这里所需要的也就是一个空变体，因为需要与类型名关联的属性信息只是指向相关类型描述符的指针。（与Ada语言不同，C语言中没有显式的**null**变体；因此这里只是简单地使用注释以表明它们的存在。）

正如检查<type declaration>产生式所期待的那样，语义例程type_decl()在ID记录和TYPEREF记录上进行操作。因为它是一个声明产生式，所以不产生语义记录：

```
type_decl(<id>, <type definition>)
{
   The identifier is entered into the symbol table
   with the following associated attributes record:
   (attributes) {
      .class = TYPENAME;
      .id_type = <type definition>.type_ref.object_type;
      .id = the id_entry returned by enter(); };
}
```

330

和变量声明一样，类型声明不产生存放在语义栈上的语义记录。相反，从声明处获得的信息将保存在符号表中。因为对已声明的标识符的引用可以出现在程序的任何地方，所以显而易见的是，栈不是用来保持和访问这些信息的有用场所。在大多数时间里，语义栈是用来保存和某个声明、定义或语句（尽管它们可能嵌套）的处理相关的信息。声明和语句全部处理后的结果会出现在符号表和生成的代码中，而不是出现在语义栈中。

现在，我们已经知道了如何在符号表中表示类型名，接下来就可以定义type_reference()语义例程。就如我们已经看到的，它产生包含适当类型引用的一个语义记录。它从与<id>关联的attributes记录获取类型描述符引用并放在那个记录中。

```
type_reference(<id>) => <type name>
{
   Find <id>.id.id in the symbol table
   if (it is there && its attrs.class == TYPENAME)
      <type name> ← (struct type_ref) {
         .object_type = <id>.attrs.id_type };
   else
      <type name>.class = ERROR;
}
```

到目前为止，我们所考察的例程已经说明了类型描述符技术的一般性。在描述变量和类型声明时，除了用来分配变量空间的类型大小信息外，我们并没有考虑类型描述符的更多细节。无论是标量类型还是结构化类型，都不影响上述例程。惟一重要的是，所有不同的类型都采用统一的类型描述符来描述并且与类型相对应的语义记录包含着指向这些描述符的指针。使用这种方法可以很容易地添加新的类型而无需改变编译器中有关类型处理的那部分结构。

处理类型声明错误

在处理类型定义的语义例程描述中，要确定各种符合静态语义约束的检查。多数情况下，语义例程的描述假设这些检查是成功的，而在假设不成立时则使用某些标准的错误处理技术。在第7章里介绍的技术将某个预期的语义记录替换为ERROR记录以表示错误已经发生并且错误信息也已发出。那种技术一般可用于本章里描述的所有例程，从而寄希望于每个例程将检查它使用的语义记录是否有错。

331

在引入ERROR记录时，我们曾提及这些错误标记将一直传播下去直至它们遇到像var_decl()和type_decl()那样的例程。由于这些例程不产生用于进一步传播的语义记录，因此也就阻止了错误标记的进一步传播。此方法的不足在于：那些即将由声明例程填入符号表的标识符将被忽略。而随后程序中对这些标识符的使用将导致更多错误信息的产生，即报告这些标识符是未定义的。我们的目标是为每个错误产生最少的错误信息（最好是一个），为此，必须改进我们的技术。在声明例程之前出现的错误主

要来源于不恰当的类型定义，因此我们在TYPEREF记录中包括一个特殊的选项，称为ERRORTYPE。当其中一个声明例程发现它接收了ERROR记录而非TYPEREF记录时，它继续处理并用ERRORTYPE类型描述符来声明它正在处理的一个或多个标识符。在符号表中检索任意标识符的属性时，我们可以查看它关联的类型是否为ERRORTYPE。如果是，则产生ERROR记录以取代处理该标识符的语义例程所期待的语义记录。遵照通常使用ERROR记录的协议，后来的例程不会再为此产生任何错误信息。

type_descriptor记录

以下两节将考察类型描述符的结构，它必须被构建成能处理最普通的结构化类型（如记录和数组）的类型定义。为了表示这些类型以及预定义的数值类型，类型描述符记录必须具有这样的组成，最初是在图10-5中描述：

```
enum type_form { INTEGERTYPE, FLOATTYPE, STRINGTYPE,
   ARRAYTYPE, RECORDTYPE, ERRORTYPE };

typedef struct type_des {
   address_range size;
   enum type_form form;
   union {
      /* form == INTEGERTYPE, FLOATTYPE, STRINGTYPE,
                 ERRORTYPE -- empty variant */

      /* form == ARRAYTYPE */
      struct {
         struct range bounds;
         struct type_des *element_type;
      };
      /* form == RECORDTYPE */
      struct {
         symbol_table fields;
         attributes *field_list;
      };
   };
} type_descriptor;
```

332

给定上述这个C语言的类型声明，我们很容易构造出Integer、Float和String等预定义类型的类型描述符，它们将被符号表中这些名字的属性条目所引用。注意，这些名字必须放在符号表中的特殊区域内，该区域包含了正被编译的程序的最外围作用域中的名字。我们必须有这么一个特别的区域，因为这些名字均不是保留字。它们可以在任何程序中被重新定义，但那样做是以牺牲程序的可读性为代价的。Integer类型描述符如下：

```
(type_descriptor) { .form = INTEGERTYPE;
                    .size = INTEGERSIZE; }
```

尽管Float和String使用不同的常量FLOATSIZE和STRINGSIZE来指定size域的值，但它们的类型描述符还是很相似。这三个常量在编译器前端代表着机器相关性。它们明显依赖于目标机器的数值表示细节和所选择的串的实现方式。INTEGERSIZE和FLOATSIZE的典型值（字节数）分别是2和4。STRINGSIZE将在第11章介绍串的实现时讨论。（此时，C语言的sizeof构造未必有用，因为这些常量代表着目标机器上的“数值”，而这些目标机器不需要和正在运行编译器的机器一样。）

类型相容性

剩下的问题是：“类型相同或者有约束的类型相容”究竟是什么意思？Ada、Pascal和Modula-2有严格的类型等价定义，即每个类型定义都定义了一个和其他所有的类型不相容的新的、不同的类型。这种定义意味着以下声明：

```
A, B : array (1..10) of Integer;
C, D : array (1..10) of Integer;
```

等价于

```
type Type1 is array (1..10) of Integer;
A, B : Type1;
type Type2 is array (1..10) of Integer;
C, D : Type2;
```

333

A和B的类型是相同的，C和D的类型也是相同的；但这两个类型是由不同的类型定义来定义的，因而它们彼此不相容，以至于将C的值赋给A是非法的。此规则很容易由编译器来强制执行。因为每个类型定义生成不同的类型描述符，而类型等价的测试仅需比较相应的描述符指针即可。

其他语言（如最为知名的Algol 68）使用别的规则来定义类型等价。最常见的办法是使用结构化类型等价。正如这个名字所暗示的，使用这个规则时，如果两个类型有相同的定义结构，则它们是类型等价的。这样，前面提到的Type1和Type2可看作等价的类型。乍看起来，这种规则似乎是一种很恰当的选择，对于使用这种语言的程序员很方便。然而，近期一些语言的设计者已经认识到结构等价规则不可能让程序员从类型检查的概念中获取全部好处。也就是说，即使程序员想让编译器通过类型检查来区分Type1和Type2，编译器也做不到，因为这些类型明显地在结构上等价。

结构等价很难实现。这种类型等价不能由简单的指针比较来决定。相反地，需要对两个类型描述符结构进行平行的遍历。为此需要为类型描述符记录中的每个变体设计特殊的分支代码。作为选择，当语义动作例程处理类型定义时，所定义的类型可以和前面已定义类型进行比较，以便用同样的数据结构表示等价的类型，即使它们是分开定义的也是如此。此项技术允许用指针比较的方法来实现类型等价测试，但需要一种索引机制，以便在声明处理时能够判别一个新定义的类型是否和前面任何一个已定义类型相等价。

此外，指针类型中可能出现的递归会给类型的结构等价测试的实现带来难以捉摸的麻烦。考虑能够确定下面两个Ada类型是否结构等价的例程的编写问题：

```
type A is access B;

type B is access A;
```

尽管这样的定义在语义上没有任何意义，但它却是合乎语法的（假设在A的定义前有引入名字B的不完整的声明）。这样，采用结构等价规则的语言的编译器必须能做出适当的决定——即A和B是等价的。如果采用平行的遍历来实现等价测试，遍历例程必须在比较过程中“记住”它们已访问的描述符以避免陷入死循环（见练习1）。可见，还是比较指向类型描述符的指针来得容易一些。

334

10.2.3 记录类型

记录定义从一系列域的声明中构造一个新的类型。域的声明在语法上和处理方式上同变量声明类似。记录可以由一个类型描述符来表示，该描述符包括一个新的包含这些域的符号表。

记录定义的语法及所需的语义动作如下：

```
<record type definition>  → record #start_record <component list>
                              #end_record end record
<component list>          → <component declaration>
                              { <component declaration> }
<component declaration>   → <id list> : <type name> #field_decl ;
```

再次地，我们需要一个新的语义记录类型：

```
struct record_def {
   type_decriptor *this_type;
   address_range next_offset;
};
```

在TYPEREF记录中，我们曾看到描述记录的下列变体：

```
struct {   /* form == RECORDTYPE */
   symbol_table fields;
   attributes *field_list;
};
```

上述结构采用两种方法来表示记录域：1) 作为单独的符号表，它可提供最快速的域名搜索；2)作为有序列表，利于处理记录聚合。记录通常不是很大，因此有理由仅保持这个列表并使用它作为特殊的符号表来定位记录域名。但我们还是包括了这两种形式以便提供最全面的处理能力。

attributes记录包括以下处理记录域的变体：

```
struct {   /* class == FIELD */
   address_range field_offset;
   struct attributes *next_field;
};
```

field_offset记录了该域在包含它的记录中的偏移；next_field用来为这个包含记录建立它的域列表。

在处理记录的过程中调用的第一个动作例程是start_record()。它建立能被处理域声明的动作例程使用的RECORDDEF语义记录。下面的动作例程描述中的开头部分指出由start_record()例程产生的 335
语义记录与符号**record**相关联。这个关联很有用，因为该语义记录实质上描述了正在处理的记录声明，并且我们将在处理每个域的声明时引用它：

```
start_record(void) => record
{
   Create a type descriptor, T, as follows:
        (type_descriptor) { .form = RECORDTYPE;
                            .size = 0;
                            .field_list = NULL;
                            .fields = create(); }

   record ← (struct record_def) {
                  .this_type = & T;
                  .next_offset = 0; } ;
}
```

在采用动作控制的语义栈的编译器中，由start_record()产生的语义记录将被放在栈上且在调用field_decl()时可用，此时栈的格局如图10-7所示。(假设标识符列表全部存放在栈里。)

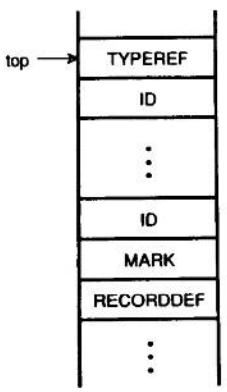


图10-7 调用field_decl()时的语义栈格局 336

为了阐述在我们的语义例程描述符号中的这种用法，我们使用了如下的上下文有关（context-sensitive）的产生式来描述<component declaration>：

record <component declaration> → **record** <id list> : <type name> #field_decl ;

除了所做的仅与当前记录相关而非与当前过程作用域相关以外，由例程field_decl()完成的处理和var_decl()所做的处理非常类似。

```
field_decl(record, <id list>, <type name>) => record
{
  Let RD name the semantic record associated with record

  for (each identifier in <id list>) {
    Enter the identifier in the symbol table
       referenced by RD.fields
    if (it is already there) {
       generate an appropriate error message
       continue;    /* go on to the next identifier */
    }

    The following expression describes the attribute
    record to be created for the field:

    (attributes) {
       .class = FIELD;
       .id_type = <type name>.type_ref.object_type;
       .id = the id_entry returned by enter();
       .field_offset = RD.next_offset;
       .next_field = NULL; }
    /* Allocate space for the field with the record. */
    RD.next_offset += <type name>.type_ref.object_type.size

    Add this attribute record to the end of the list
       referenced by RD.field_list
  }

   record ← RD ;
}
```

由field_decl()使用的空间分配技术把每个域的大小均加到RD.next_offset上，此项技术是基于不需要地址对齐的简单假设。它不能用于某些体系结构的机器上，因为在这些机器上，如果记录的域比对齐因子小，则要求相应的数据对象开始于被2或4整除的地址。在这样的机器上，域的大小必须加以调整来维持适当的地址对齐。我们继续在其他涉及空间分配的例程中使用这种相同的简化假设。而对齐
337 考虑所必需的扩展也是很简单的。

最后，在end_record()例程中，我们看到所构造的TYPEREF语义记录用来表示刚被处理的记录类型定义：

```
end_record(record) => <record type definition>
{
   type_descriptor *T;

   T ← record.record_def.this_type;
   T.size ← record.record_def.next_offset;
   <record type definition> ← (struct type_ref) {
                .object_type = T; } ;
}
```

再一次，新的类型由TYPEREF语义记录来表示。此类型的描述符由一个指向按声明次序排列的记录域的attributes列表的指针（利于处理记录聚合）和一个包含每一个记录域相应条目的（任意结构的）符号表组成。这些符号表的条目当然也包含用于每个记录域的attributes记录的引用。

10.2.4 静态数组

程序中带有静态下标范围的数组声明是非常普遍的，因此它们值得单独考虑。这也是很多著名的语言（如C、Pascal和Modula-2）中所允许的惟一的数组类型定义形式。因为这个特殊情况的识别可以使编译器有效地减少数组实现的开销，所以我们就从数据结构的定义和实现静态数组的动作例程来开始有

关数组的讨论。

下面是用于约束数组定义的Ada/CS语法的一个子集。（约束数组（constrained array）是Ada的一种数组类型，它的范围由特定的离散类型来确定。）特别地，这个子集被限制为仅允许整型文字常量用作（下标）范围描述，这样就确保了数组的所有（下标）范围均是静态的。

```
<array type definition> → array ( <static range> ) of <type> #array_def
<static range>          → INTLITERAL #lower_bound .. INTLITERAL #upper_bound
```

这些产生式仅定义了一维数组。多维数组必须被定义为数组的数组。我们需要一个新的语义记录类型来表示<static range>。它由下列声明来定义：

```
struct range {
   long lower, upper;
};
```

338

如前所述，TYPEREF记录中数组的候选为：

```
struct {   /* form == ARRAYTYPE */
   struct range bounds;
   struct type_des *element_type;
};
```

有关的语义例程定义如下。lower_bound()和upper_bound()建立RANGE记录，例程array_def()使用该记录和数组元素类型的TYPEREF来产生数组的TYPEREF。

```
lower_bound(INTLITERAL) => <static range>
{
   <static range> ← a RANGE record with lower
      set to the value of INTLITERAL
}

upper_bound(<static range>, INTLITERAL) => <static range>
{
   long u, l;
   <static range>.range.upper ← the value of INTLITERAL
   u = <static range>.range.upper;
   l = <static range>.range.lower;
   if (u >= l)
      <static range> ← the updated RANGE record
   else {
      Produce an appropriate error message
      <static range> ← an ERROR record or a corrected
         version of the RANGE record
   }
}

array_def(<static range>, <type>) => <array type definition>
{
   long u, l;
   u = <static range>.range.upper;
   l = <static range>.range.lower;
   Create a new type descriptor for an array type:
      T ← (type_descriptor) {
            .form = ARRAYTYPE;
            .element_type = <type>.type_ref.object_type;
            .bounds = <static range>.range;
            .size = .element_type->size * (u - l + 1); }
   <array type definition> ← (struct type_ref) {
            .object_type = T; }
}
```

339

因此，对以下的这个数组定义

```
array (1..10) of array (1..20) of Integer
```

将建立如图10-8所示的type_descriptor结构。

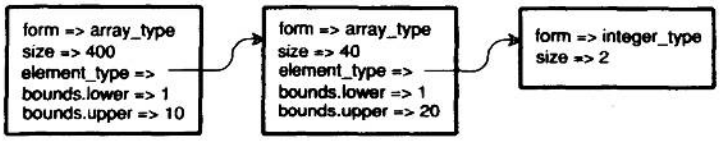


图10-8 type_descriptor示例

10.3 高级特性的动作例程

10.3.1 变量和常量声明

现在考虑Ada/CS中全部的对象声明语法。它包括以下的产生式和语义动作记号:

```
<object declaration>    → <id list> : <object tail>
<object tail>           → <constant option> <type or subtype>
                            <initialization option> #object_decl
<constant option>       → #not_constant
<constant option>       → constant #is_constant
<type or subtype>       → <type>
<type or subtype>       → <subtype definition>
<initialization option> → #no_initialization
<initialization option> → := <expression>
```

而且还需要一个额外的语义记录类型:

```
struct const_option { Boolean is_constant; };
```

这里引入处理可选关键字**constant**的两个例程。它们中的每一个都产生CONSTOPTION语义记录来
340 标识该关键字是否出现。

```
is_constant(void) => <constant option>
{
    <constant option> ← (struct const_option) {
                    .is_constant = TRUE; } ;
}

not_constant(void) => <constant option>
{
    <constant option> ← (struct const_option) {
                    .is_constant = FALSE; } ;
}
```

Ada和Ada/CS中的常量可以是其值在编译时就能确定的文字常量，或者其值由运行时方能计算的表达式来决定。若是文字常量，则在attributes记录中使用由CONST标记的新的变体:

```
/* class == CONST */
struct value_type value;
```

其中value_type在讨论表达式的第11章里定义。

若是运行时的值，则常量将用作只读变量，因为我们必须像处理其他变量那样给它分配空间，但它也可以仅由一个初始化表达式来定值。因此，这个特性需要扩展在10.2.1节中定义的struct address类型:

```
struct address {
    short var_level;
    address_range var_offset;
    boolean indirect, read_only;
};
```

非终结符<type or subtype>从语法上定义了正在声明的对象类型。和10.2节里简单子集中的类型一样，这里也由TYPEREF记录来表示类型。因此，不需要新的语义例程或者新的语义记录类型来处理这个

非终结符。

对象的初始化可以选择性地出现在其声明中。如果它出现，它将由描述表达式的语义记录来表示（参见第11章）；反之，例程no_initialization()必须产生某种可作为占位符的空记录。MARK记录可以胜任此项工作：

```
no_initialization(void) => <initialization option>
{
   <initialization option> ← a MARK semantic record ;
}
```

341

object_decl()显然比var_decl()更复杂，这是因为它不但要处理变量而且还要处理常量和初始化表达式。像var_decl()一样，它的基本工作也是把所有已声明的标识符放入符号表中并且将它们与合适的属性记录相关联。然而，由object_decl()建立的与标识符关联的attributes记录有设置为VARIABLE或者CONST的class域。图10-9描述了例程object_decl()所做的工作，在其中突出的位置给出了初始化选项的描述。

```
enum init_kind { INITCONST, INITVARIABLE, INITNONE };

object_decl(<id list>, <constant option>, <type or subtype>,
            <initialization option>)
{
  enum init_kind initialization;

  if (<initialization option>.kind == DATAOBJECT) {
     Verify the assignability of
        <initialization option>.data_object.object_type
        to <type or subtype>.type_ref.object_type.
     if (the expression has a compile-time value)
        initialization = INITCONST;
     else
        initialization = INITVARIABLE;

  } else {
     /*
      * An initialization expression must be present if
      * so indicated by <const option>.
      */
     if (<constant option>.const_option.is_constant)
        Produce an "Initialization required" error message
     initialization = INITNONE;
  }

  for (each identifier in <id list>) {
     Call enter() to put the identifier in the current
        scope of the symbol table
     if (it is already there)
        generate an appropriate error message
     else if (! <constant option>.const_option.is_constant
         || initialization == INITVARIABLE) {
        Allocate storage for the variable, recording its
        offset in the local variable offset. (The size of
        the block of storage to allocate is obtained from
        <type or subtype>.type_ref.object_type.size)
        The following expression describes the attribute
        record to be created for the variable:
       (attributes) {
          .class = VARIABLE;
          .id_type = <type or subtype>.type_ref.object_type;
          .id = the id_entry returned by enter();
          .var_address = (struct address) {
             .var_level = current_nesting_level;
             .var_offset = offset;
```

342

图10-9 object_decl()动作例程

```
                .indirect = FALSE;
                .read_only =
                  <constant option>.const_option.is_constant; };
            }

            if (initialization != INITNONE)
               Generate assignment code to initialize
                the object
         } else  /* the identifier names a literal constant */
            The following expression describes the attribute
            record for the identifier:

            (attributes) {
               .class = CONST,
               .id_type = <type or subtype>.type_ref.object_type;
               .id = the id_entry returned by enter();
               .value = <initialization option>.data_object.value; }
      }
   }
```

图10-9 （续）

另一种语义记录DATAOBJECT在第11章里定义。

10.3.2 枚举类型

枚举类型由一个不同标识符的列表来定义。其中每个标识符是该枚举类型的一个常量。这些常量按它们在该类型定义中的位置排序并由整数值来表示。通常，代表第一个标识符的值是0，所有其他标识符的值比列表中它的前驱的值多1。（C语言中enum类型允许程序员可以选择性地指定枚举常量的值。此外，这些值是有符号的整数，而非无符号的整数。）

<enumeration type definition>的语法以及相应的语义动作记号如下：

```
<enumeration type definition>  → ( <enumeration id list> #finish_enum_type )
<enumeration id list>          → IDENTIFIER #first_enum_id
                                   { , IDENTIFIER #enum_id }
```

343

用于枚举类型声明的语义记录如下：

```
struct enum_def {
   type_descriptor *this_type;
   attributes *last_const;
};
```

type_descriptor记录必须有新的变体来处理枚举类型：

```
/* form == ENUMTYPE */
attributes *first_const;
```

attributes记录也必须加以扩展来处理枚举类型常量。一个新的id_class，ENUMCLASS必须被加入以处理枚举类型。在attributes记录中相应的变体是：

```
struct {    /* class == ENUMCONST */
   unsigned long enum_value;
   struct attributes *next_const;
};
```

其中，第一个域记录用来表示枚举常量的值，第二个域创建所有枚举类型常量值的列表。

借助枚举类型定义，所有的标识符在处理前都不必放在栈上。它的语法告诉我们，一旦分析器看见左括号就确定后面会紧跟着出现枚举类型定义。这样，一旦枚举常量被分析器所接受，跟在后面的语义例程就开始处理它们。first_enum_id()为新类型分配type_descriptor，并用那个作为参数而接收的IDENTIFIER作为枚举常量列表的开始。该例程同时将此参数作为ENUMCONST填入符号表。

enum_id()用它的IDENTIFIER参数完成同样的工作，另外还要将其添加到枚举类型的常量列表中。finish_enum_id()在参数类型定义结束处完成相关的处理，和类型定义的通常处理办法一样，这里也产生TYPEREF记录:

```
first_enum_id(IDENTIFIER) => <enumeration id list>
{
    Allocate a type_descriptor for an ENUMTYPE and let
       T point to it.
    Enter the IDENTIFIER into the symbol table.
       There may be more than one use of an identifier
       as enumeration constants of different types
       within a single scope (overloading), but an
       identifier used as an enumeration constant
       cannot have any other uses within a scope.
    Assuming an error is not reported, the identifier
       just entered into the symbol table has the
       following attributes record, A, associated
       with it:
   (attributes) {
       .class = ENUMCONST;
       .id_type = T;
       .id = the id_entry returned by enter();
       .enum_value = 0;
       .next_const = NULL; }
   Set T->size to INTEGERSIZE and T->first_const to point
      to the attribute record just created
   <enumeration id list> ← (struct enum_def) {
                                          .this_type = T;
                                          .last_const = A; };
   /* assuming A references the attributes record */
}

enum_id(<enumeration id list>, IDENTIFIER) => <enumeration id list>
{
   Let ED rename <enumeration id list>.enum_def
   Enter the IDENTIFIER into the symbol table.
      The same error checking must be done as in
      first_enum_id().  In addition, the identifier
      must not already be in the symbol table as a
      constant of the type currently being processed.
   Assuming an error is not found, create an ENUMCONST
      attribute record for the identifier with
      id_type = ED.this_type
      enum_value = ED.last_const.enum_value + 1
      next_const = NULL
   Set ED.last_const.next_const to point to the
      attribute record just created
   Set ED.last_const to ED.last_const.next_const, thus
      adding the new attribute record to the end
      of the list
   <enumeration id list> ← the updated struct enum_def
}

finish_enum_type(<enumeration id list>) => <enumeration type definition>
{
   <enumeration type definition> ← (struct type_ref) {
     .object_type = <enumeration id list>.enum_def.this_type; }
}
```

Ultimately, the enumeration type is represented by a **TYPEREF** record, as all types are. The type descriptor consists of a **type_descriptor** record that points to a list of **attributes**, one for each enumeration constant.

最后，和所有其他类型一样，枚举类型也由一个TYPEREF记录来表示。这个类型描述符由一个type_descriptor记录组成，那个记录指向一个由每个枚举类型常量对应的attributes所构成的列表。

10.3.3 子类型

子类型（subtype）声明命名类型的约束型应用。（C语言没有约束类型。）由于子类型和类型在多数时候可以互换使用，因此子类型也可以由type_descriptor记录来表示。为描述子类型，需要在type_descriptor中增加新的变体，以及一个在这个新变体中使用的新类型struct constraint_des。对type_descriptor的扩展包括：

```
struct {   /* form == SUBTYPE */
   struct type_des *base_type;
   struct constraint_des constraint;
};
```

其中的“约束”如下：

```
enum constraint_form { DYNAMICRANGE, STATICRANGE,
      UNCONSTRAINEDINDEX, ARRAYBOUNDS
   };
```

struct constraint_des定义大致如下：

```
struct constraint_des {
   enum constraint_form form;
   union {
      /* form == DYNAMICRANGE */
      address_range address;
      /* form == STATICRANGE */
      struct { long lowerbound, upperbound; };
      /* form == UNCONSTRAINEDINDEX - empty */
      /* form == ARRAYBOUNDS */
      struct {
         struct index_list *bounds;
         struct address dope_vector_addr;
      };
   };
};
```

下面的产生式给出了子类型定义和声明的语法。它包括两种基本形式：范围约束用于离散类型，下标约束用于非约束数组类型。非终结符<range>在语言的其他特性中也有大量的应用（例如，在**for loop**循环里）。

346

```
<subtype>              → <type name>
<subtype>              → <subtype definition>
<subtype declaration>  → subtype <id> is
                         <subtype definition> #subtype_decl
<subtype definition>   → <type name> <range constraint> #range_subtype
<subtype definition>   → <type name> <index constraint> #array_subtype
<range constraint>     → range <range>
<range>                → <simple expression> .. <simple expression>
                         #range_pair
<index constraint>     → ( <discrete range> #start_index_list
                         {, <discrete range> #append_index} )
<discrete range>       → <subtype>
<discrete range>       → <range>
```

处理子类型不需要额外的语义记录类型。处理范围约束的语义例程也是如此。辅助语义例程start_index_list()、append_index()和array_subtype()在10.3.4节随数组一起讨论。在这个小节里，我们感兴趣的是那些定义范围约束的子类型。范围约束可用于任何离散类型，对Ada/CS而言，它包括整型和任何用户定义的枚举类型。正如它的名字所暗示的，子类型没有定义新的类型。相反地，它描述了基类型的在一个受约束范围内的值。当变量被声明为特定的子类型时，编译器必须确保任何赋给该变量的值在所允许的范围内。如果此条件不能被编译时的分析所识别，则需要添加运行时检查。

动作例程range_pair()从作为范围边界而给出的两个表达式中建立struct constraint_des。

range_subtype()为子类型建立type_descriptor。这些步骤是分开的，因为非终结符<range>既可以出现在子类型定义中又可以出现在文法的其他上下文中。

```
range_pair(<simple expression>1, <simple expression>2) => <range>
{
 Check that the two expression entries are of the same
    discrete type.
 If they both denote compile-time constants, then this
    is a static range; otherwise, it is dynamic.

 if (the constraint is static) {
    Create the following compile-time constraint
      descriptor, C:
    (struct constraint_des) {
     .form = STATICRANGE;
     .upperbound = (long) (<simple expression>2.expr.value);
     .lowerbound = (long) (<simple expression>1.expr.value); }
     } else {   /* the constraint is dynamic */
        Allocate space in the data area of the current
         activation record for a run-time descriptor
         (2*INTEGERSIZE words, one for each of
         the bounds).
        Generate code to store the bounds into the
         descriptor.
        Create the following compile-time constraint
         descriptor, C:
                 (struct constraint_des) {
                      .form = DYNAMICRANGE;
                      .address = . . . ; };
                      /* the space just allocated */
     }

     The type descriptor, T, for the range will be:
        (type_descriptor) {
          .form = SUBTYPE;
          .size = <simple expression>2.expr.expr_type.size;
          .base_type = <simple expression>2.expr.expr_type;
          .constraint = C; }

     <range> ← (struct type_ref) { .object_type = T; } ;
    }


    range_subtype(<type name>,
                  <range constraint>) => <subtype definition>
    {
      <range constraint>.type_ref.object_type.base_type
          must refer to the same type as
          <type name>.type_ref.object_type.
      if (<type name>.type_ref.object_type is constrained)
         The new constraint must not be less restrictive
            than the old one.
      if (no errors)
         <subtype definition> ← <range constraint>
      else
         <subtype definition> ← an ERROR record
    }


    subtype_decl(<id>, <type definition>)
    {
       /* same as type_decl()... */
       type_decl(<id>, <type definition>);
    }
```

347

348

10.3.4 数组类型

Ada和Ada/CS包括两种形式的数组定义：约束数组与非约束数组。约束数组类型有固定数目的元素，

尽管元素个数可以由仅在运行时计算的表达式来指定。非约束数组类型仅通过定义数组的下标类型来指定，它的元素个数留待以后说明且随下标类型实例的不同而变化。非约束数组类型的任何实例的大小都可以在编译器时确定，也可以像约束数组类型那样，由运行时表达式的值来决定。

实现动态数组

约束数组类型或子类型的变量，若其边界是在运行时而非编译时计算的，则它们常常被称为动态数组（dynamic array）。在程序的不同执行需要大小不同的数组时，这种数组大小的迟后绑定就常常显得非常方便了。然而，它带来的问题是：用于存放动态数组的空间不能被分配在活动记录中；这种数组的大小在编译时刻当AR偏移均固定时仍是未知的。我们所采取的办法是：在AR中放一个用于边界约束的大小固定的描述符（通常称之为内情向量（dope vector））。这个内情向量中包含了用于存放计算后的边界的空间。这种类型的变量可以用包含其元素地址的字以及（隐式地）用内情向量来表示。当一个过程被调用时，它将计算动态数组类型的边界，并在紧随当前AR之后的栈上分配每个动态数组的空间。此空间的地址放在与每个动态数组相关的指定位置上，所有对数组元素的引用将通过这个指针来进行。例如，

```
procedure P (N : Integer) is
 A, B : array(1..N) of Integer;
   ...
end P;
```

数组A和B的大小在调用P时确定。首先，P的（大小固定的）AR被压入栈上，然后N值将被填入内情向量。最后，在栈上分配A和B的空间。这些操作导致如图10-10所示的栈结构。

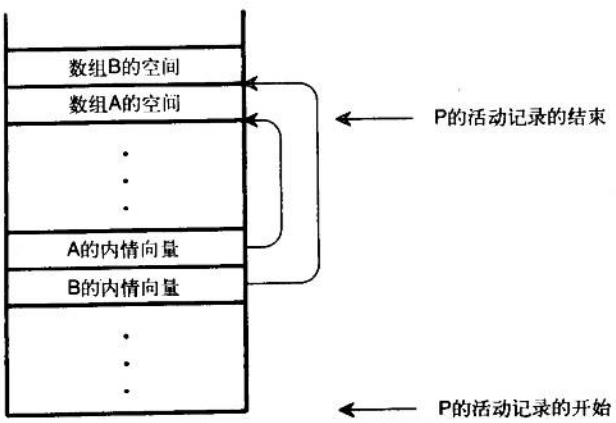


图10-10 带有动态数组的活动记录

在块中声明的动态数组可以和在过程中声明的动态数组一样处理。如果使用块级的分配策略，操作将很简单，即先压入块的AR，然后再压入它包含的所有动态数组的空间。

但如果使用过程级的分配策略，那情况又会如何呢？同以前一样，包含所有局部变量（动态数组空间除外）的AR首先被压入栈中。动态数组的空间在进入包含它们的块时分配。（和过程一样）我们为每个块维持一个stack_top值。stack_top指向块中动态数组空间的末端。当进入块时，它的
349 stack_top值是从包围它的块（若没有，则从包围它的过程）中的值继承的。该值指示了在哪里可以分配动态数组并随着局部动态数组空间被压入栈中而增加。当退出块时，仅需恢复使用包围它的块的stack_top值而无需额外的工作。（这种方式容易实现跳出块的**exit**和**goto**语句。）作为示例，重新考虑前面那个现已添加块的例子：

```
procedure P (N : Integer) is
 A : array(1..N) of Integer;
begin
    ...
    declare
        B : array(1..N) of Integer;
    begin
        ...
    end;
end P;
```

图10-11说明了使用块级stack_top值创建的栈结构。

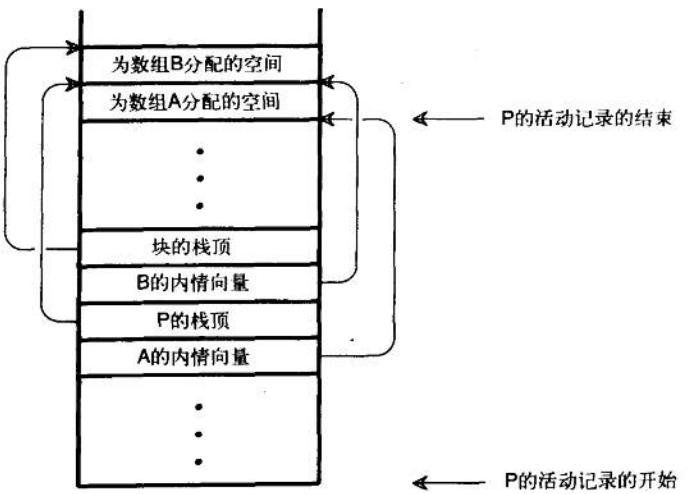


图10-11 带有块级stack_top值的活动记录

有了动态数组，编译器不再知晓每个子程序的调用在运行时栈上所需的确切的空间数量。此决定必须等到运行时才能做出，所付出的相关代价是，在活动记录中需要存放内情向量及stack_top值的空间。在每个数组引用上需要花费时间，以便访问内情向量并从中获取边界信息以计算下标和进行边界检查（在第11章讨论）。另外，在块入口也需要时间用于分配数组。 350

语义例程array_def()处理动态数组定义的方式与静态数组略有不同，主要区别在于用内情向量占用的空间（而不是数组元素占用的空间）来填写记录type_descriptor()的size域。在声明动态数组类型及动态数组变量时，例程array_def()和object_decl()必须分别为它们生成用于填入内情向量并分配动态数组空间的代码。

Ada数组的语义例程

可以采用两种稍有不同的语法形式来指定Ada/CS中的两种数组（约束数组和非约束数组）。它们被表示在下面包括了动作符号的产生式中： 351

```
<array type definition>                  → <unconstrained array definition>
<array type definition>                  → <constrained array definition>

<unconstrained array definition>         →
    array <unconstrained index list> of <element type> #array_def
<unconstrained index list>               → ( <index subtype def>
    #start_index_list {, <index subtype def> #append_index} )
<index subtype def>                      →
    <type name> range <> #unconstrained_index

<constrained array definition>           →
    array <constrained index list> of <element type> #array_def
```

```
<constrained index list>                    →
    ( <discrete range> #start_index_list {, <discrete range> #append_index} )
<discrete range>                            → <subtype>
<discrete range>                            → <range>

<element type>                              → <type or subtype>
```

因为数组可以用下标类型的列表来定义，所以处理数组需要定义一个新的类型以构造这些列表：

```
struct index_list {
    type_descriptor *index_type;
    struct index_list *next;
};
```

type_descriptor记录需要以下变体替代10.2.1节中的相应部分来处理Ada/CS数组类型：

```
struct {    /* form == ARRAYTYPE */
    struct index_list *index_types;
    struct type_des *element_type;
};
```

为处理下标列表也需要定义新的语义记录类型：

```
struct index_list_type {
    struct index_list *list;
    boolean constrained;
};
```

通过对前面数组语法中的语义动作标记进行检查，我们可以清楚地看到，处理约束数组定义所做的工作和相应处理非约束数组所需做的工作相比，并没有什么太大的区别。为使用相同的例程来处理这两种情况，我们必须保证非约束的<index subtype def>和<discrete range>的语义记录表示是一样的。为此，我们需要将它用一个TYPEDEF记录来表示。unconstrainted_index()为非约束子类型构造合适
352 的描述符：

```
unconstrained_index(<type name>) => <index subtype def>
{
    <type name>.type_ref must describe a discrete type.
    <index subtype def> ← (type_descriptor) {
        .form = SUBTYPE;
        .size = <type name>.type_ref.object_type.size;
        .base_type = <type name>.type_ref.object_type;
        .constraint = (struct constraint_des) {
              .form = UNCONSTRAINEDINDEX; } ;
        };
}
```

使用start_index_list()和append_index()来处理非约束的和约束的下标列表。在它们的参数列表和返回值描述中，<index subtype>要么表示<index subtype def>要么表示<discrete range>，而<index list>要么表示<unconstrained index list>要么表示<constrained index list>。start_index_list()处理下标列表中的第一个下标类型。它构造INDEXLIST语义记录，这个语义记录包含该列表中第一个<index subtype>的条目。

```
start_index_list(<index subtype>) => <index list>
{
    type_descriptor *T;

    T ← <index subtype>.type_ref.object_type
    <index list> ← (struct index_list_type) {
        .constrained =
              (T.constraint.form != UNCONSTRAINEDINDEX);
        .list = (struct index_list_type) {
            .list->index_type = T,
            .list->next = NULL; } ;
        };
}
```

append_index()给<index list>相关的列表添加下一个下标子类型:

```
append_index(<index list>, <index subtype>) => <index list>
{
   Append the index type referenced in
     <index subtype>.type_ref to the end of the list
     referenced in <index list>.index_list
   <index list> ← the new struct index_list
}
```

array_def()给每个类型定义创建一个类型描述符。其中为非约束数组建立的是ARRAYTYPE类型描述符，而为约束数组建立的是SUBTYPE类型描述符。SUBTYPE类型描述符可以用作变量的类型，而ARRAYTYPE类型描述符却不行。后者必须添加下标约束以创建子类型。动作例程array_def()被描述在图10-12中。图10-13a、b给出了为非约束数组和约束数组创建的类型描述符的结构示例。 353

```
array_def(<index list>, <element type>) => <array type definition>
{
  Create a new type descriptor, T, for the array type:
  if (<index list>.index_list.constrained == FALSE) {
     T ← (type_descriptor) {
       .form = ARRAYTYPE;
       .size = ADDRESSSIZE; /* space for the array adr */
       .element_type = <element type>.type_ref.object_type;
       .index_types = <index list>.index_list.list; };
  } else { /* <index list>.index_list.constrained == TRUE */
     Allocate space in the current activation record
     for a dope vector, 2 integers for each entry on
     <index list>.  Create a struct address describing
     the location of the dope vector and call it
     DV_addr.

     Generate code to copy the static and dynamic
     bounds descriptors on <index list> into the
     appropriate locations in the dope vector.

     T ← (type_descriptor) {
       .form = SUBTYPE;
       .size = ADDRESSSIZE; /* space for the array adr */
       .base_type = (type_descriptor) {
          .form = ARRAYTYPE;
          .size = ARRAYDESCRIPTORSIZE;
          .element_type =
             <element type>.type_ref.object_type,
          .index_types = NULL; };
       .constraint = (struct constraint_des) {
          .form = ARRAYBOUNDS;
          .dope_vector_addr = DV_addr;
          .bounds = <index list>.index_list.list; } ;
     }
  }
       <array type definition> ← (struct type_ref) {
              .object_type = T; } ;
}
```

图10-12 动作例程array_def()

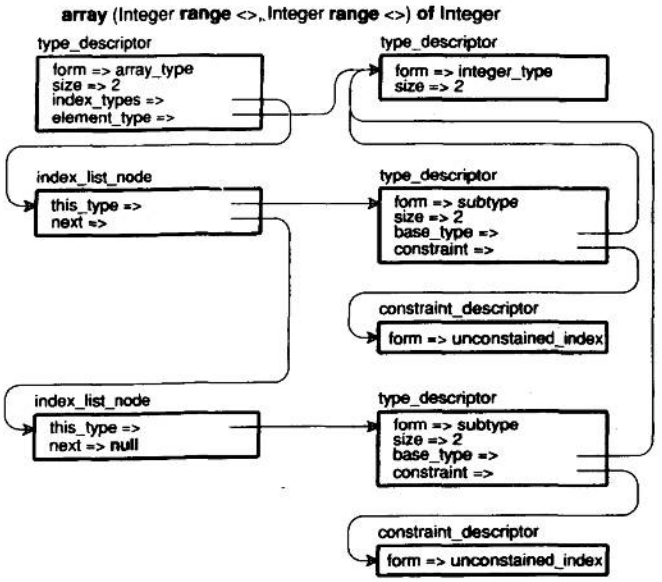
对于所有下标范围均为编译时常量的约束数组子类型，不需要采用运行时描述符。作为一种优化，可以在编译时在当前数据区中分配这种数组，其大小为:

```
element_type.size * (product of all index lengths)
```

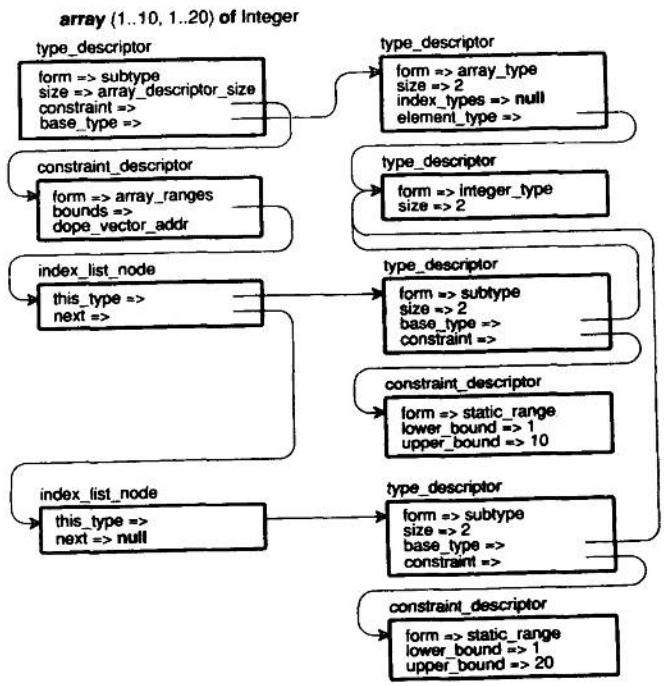
354

使用前面的例程，约束数组类型的定义将导致两个类型描述符的创建：一个是用于所定义数组的实际边界的子类型描述符，另一个是底层数组类型本身的描述符。然而，后者的记录域index_types包含一个NULL指针而不是一个指向非约束离散类型列表的指针。这样的一个列表可以从子类型描述符的bounds

355 列表中的所有类型的base_types来构造，但由于编译器从来不使用该列表，因此我们没有必要建立它。



a) 非约束数组类型的type_descriptor



b) 约束数组类型的type_descriptor

图10-13 约束数组和非约束数组的type_descriptor

指定下标约束的子类型

数组约束子类型的相关语法如下:

```
<subtype definition> → <type name> <index constraint> #array_subtype
<index constraint>   → ( <discrete range> #start_index_list
                         {, <discrete range> #append_index} )
```

356

下标约束仅能应用于非约束数组类型。列表中的每个<discrete range>给由<type name>所指示的数组中相应位置上的下标提供一个约束。同前面描述的约束数组定义一样，该子类型也由一个带有指向约束列表的SUBTYPE变体的类型描述符记录来指定。

因为我们之前已经描述了语义例程start_index_list()和append_index()，所以现在仅需要考虑一个新的例程array_subtype()，它建立SUBTYPE类型描述符:

```
array_subtype(<type name>, <index constraint>) => <subtype definition>
{
 <index constraint>.index_list references a list of
     subtype descriptors for discrete ranges.
 <type name>.type_ref represents the unconstrained
     array type.
 Check that this is an unconstrained array type.
 Check that the discrete ranges on the struct index_list
   are valid constraints for the corresponding index
   types of the array.  The number of constraints must
   match the number of index types in the array.
 Allocate space in the current activation record for a
   dope vector, with an entry corresponding to
   each index on <index list>.
 Create a struct address describing the location of
   the dope vector and call it DV_addr.
 Generate code to copy the static and dynamic
   bounds descriptors on the list referenced by
   <index constraint> into the appropriate
   locations in the dope vector.

 Construct a new type descriptor, T:
 (type_descriptor) {
   .form = SUBTYPE;
   .size = ARRAYDESCRIPTORSIZE;
   .base_type = <type name>.type_ref.object_type;
   .constraint = (struct constraint_des) {
         .form = ARRAYBOUNDS;
         .dope_vector_addr = DV_addr;
         .bounds = <index constraint>.index_list.list;
      } ;
   )

 <subtype definition> ← (struct type_ref) {
             .object_type = T; };
}
```

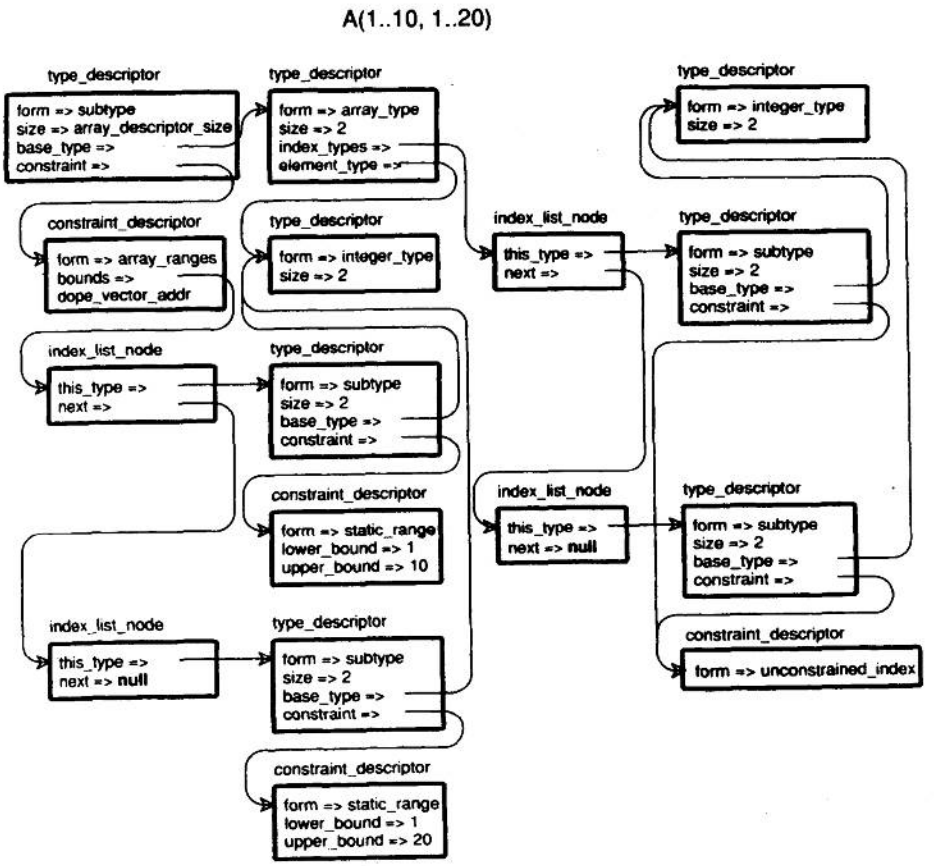
通过将下标约束应用于先前定义的非约束数组类型而定义的约束数组子类型，可以由一个子类型描述符来表示，这个子类型描述符就像我们为约束数组定义所创建的描述符一样。因此，这两种定义性方法产生的描述符有效等价。而差别在于后者，如先前所讨论的，会导致底层数组类型描述符的index_list域为NULL。在前面描述的语义例程中，这个下标列表仅用于检查约束列表，而它在一个无名数组类型中没有任何用处。

357

如果在图10-13a中定义的非约束数组类型被命名为A，则通过指定A的下标约束所创建的类型描述符结构如图10-14所示。

10.3.5 变体记录

变体记录允许在单个记录类型中描述可供选择的成员列表。每个变体描述了与特定的（Ada）判别式的值或（Pascal和Ada/CS）标签域对应的记录成员。



358 图10-14 在非约束数组类型中应用下标约束的type_descriptor

为简化变体记录的编译技术介绍，Ada/CS所定义的变体记录特性在某种程度上比Ada所定义的变体记录特性要简单一些。包含变体的记录定义的语法如下：

```
<record type definition>  → record #start_record <component list>
                            #end_record end record
<component list>          → <component declaration>
                            { <component declaration> }
<component list>          → { <component declaration> } <variant part>
<component list>          → null ;
<component declaration>   → <id list> : <type name> ; #field_decl

<variant part>            → case <id> : <type name> #tag_field is
                            <variant> { <variant> } #end_variant_part
                            end case ;
<variant>                 → when <choice> #new_variant
                            => <component list>
<choice>                  → <simple expression>
```

为描述变体列表，我们需要一个新结构：

```
struct variant_des {
   long choice_value;
   attributes *tag_field, *field_list;
   struct variant_des *inner_variants;
   struct variant_des *next_variant;
};
```

对记录定义的语义记录类型必须加以扩展以包括一个变体描述符的指针，这个指针指向当前正在处理的最内层变体：

```
struct record_def {
   type_descriptor *this_type;
   address_range next_offset;
   struct variant_des *current_variant;
};
```

因为变体可以嵌套，所以在进入嵌套的变体部分时，需要一个语义记录类型来保存current_variant的值：

359

```
struct variant_part {
   struct variant_des *outer_variant;
   attributes *tag_field;
};
```

像RECORDDEF语义记录一样，在type_descriptor记录中描述记录的变体也必须加以扩展：

```
struct {   /* form == RECORDTYPE */
   symbol_table fields;
   attributes *field_list;
   struct variant_des *variant_list;
};
```

被field_list所引用的域列表仅包含那些在记录定义中的变体部分前面声明的成员。variant_list域指向struct variant_des记录的列表，其中每个记录包含定义单个变体的域列表。因此，对变体记录来说，重要的是将所有的域保存到符号表中而不仅仅存放在上述列表中。否则，域名的搜索将会是很复杂且相对低效的。

在考虑记录类型中的变体域时，需要在attributes记录的field变体中添加两个额外的域以描述记录域：

```
struct {   /* class == FIELD */
   address_range field_offset;
   attributes *next_field;
   struct variant_des *enclosing_variant;
   boolean is_tag;
};
```

enclosing_variant引用的struct variant_des记录是内层变体的外层包围变体，它的值为NULL则表示记录域不是某变体的一部分。is_tag标志的意义很明确，并且由于标签域使用上的限制，该标志也是必需的。

例程start_record()中仅需改动的是：为添加到type_descriptor中的域以及用于处理变体的struct record_def记录指定合适的初值。

```
start_record(void) => record
{
   Create a type descriptor, T, as follows:
   (type_descriptor) {
      .form = RECORDTYPE;
      .size = 0;
      .field_list = NULL;
      .variant_list = NULL;
      .fields = create(); };
   record ← (struct record_def) { .this_type = T;
            .next_offset = 0;
            .current_variant = NULL; };
}
```

360

field_decl()也需做类似的改动，以便能指定is_tag的值为FALSE以及在为每个域创建attributes记录时将enclosing_variant赋值为RD.current_variant。

需要三个新的语义例程来处理变体：tag_field()、new_variant()和end_variant_part()。这些动作例程的调用出现在以下的语法上下文中：

```
record { <component declaration> } <variant part> →
   record { <component declaration> } case <id> : <type name>
      #tag_field is <variant> { <variant> } #end_variant_part end case ;
record { <component declaration> } case <id> : <type name> is
   { <variant> } <variant> →
      record { <component declaration> } case <id> : <type name> is
         { <variant> } when <choice> #new_variant => <component list>
```

tag_field()根据变体记录中的标签处理其中不同域的声明，该标签在运行时的值决定如何解释记录的变体部分。像field_decl()处理普通的记录域时所做的那样，tag_field()也修改了与**record**关联的语义记录。另外，tag_field()还创建了与符号**case**相关联的VARIANTPART语义记录。特殊情况的嵌套变体可以通过将**record**.record_def. current_variant的值保存在variant_part语义记录中而得到处理：

```
tag_field(record, <id>, <type name>) => (record, case)
{
   Let RD name the semantic record associated with record
   Enter <id> in the symbol table referenced by RD.fields
   if (it is already there)
      generate an appropriate error message
   else {
      The following expression describes the attribute
      record, A, to be created for the tag field.  It
      is identical to that for other field except for
      the value of is_tag.
      (attributes) {
         .class = FIELD;
         .id_type = <type name>.type_ref.object_type;
         .id = the id_entry returned by enter();
         .field_offset = RD.next_offset;
         .next_field = NULL;
         .enclosing_variant = RD.current_variant;
361      .is_tag = TRUE; }
      Allocate space for the field within the record by
          adding <type name>.type_ref.object_type.size
          to RD.next_offset
      if (RD.current_variant == NULL)
         Add this attribute record to the end of the
             list referenced by RD.field_list
      else
         Add this attribute record to the end of the
             list referenced by RD.current_variant
   }
        case ← (struct variant_part) {
         .outer_variant = RD.current_variant;
         .tag_field = A; };
   RD.current_variant = NULL;
   record ← the updated RD
}
```

分析器在识别出变体标号后就立即调用new_variant()。该例程为新的变体初始化struct variant_des记录并将它链接到当前记录的变体描述符列表中。图10-15显示了相应的RECORDDEF语义记录以及在调用new_variant()后它所引用的结构：

```
new_variant(record, case, <choice>) => record
{
  Let RD name the semantic record associated with record
  <choice> must describe a constant value, distinct from
    all other choice_values on the list of
    variant_descriptors referenced by RD.current_variant
  Create a struct variant_des, V, as described by the
```

```
    following expression:
  (struct variant_des) {
    .choice_value = (long) <choice>;
    .tag_field = case.variant_part.tag_field;
    .field_list = NULL;
    .inner_variants = NULL;
    .next_variant = RD.current_variant; }
  RD.current_variant ← V
  record ← the updated RD
}
```

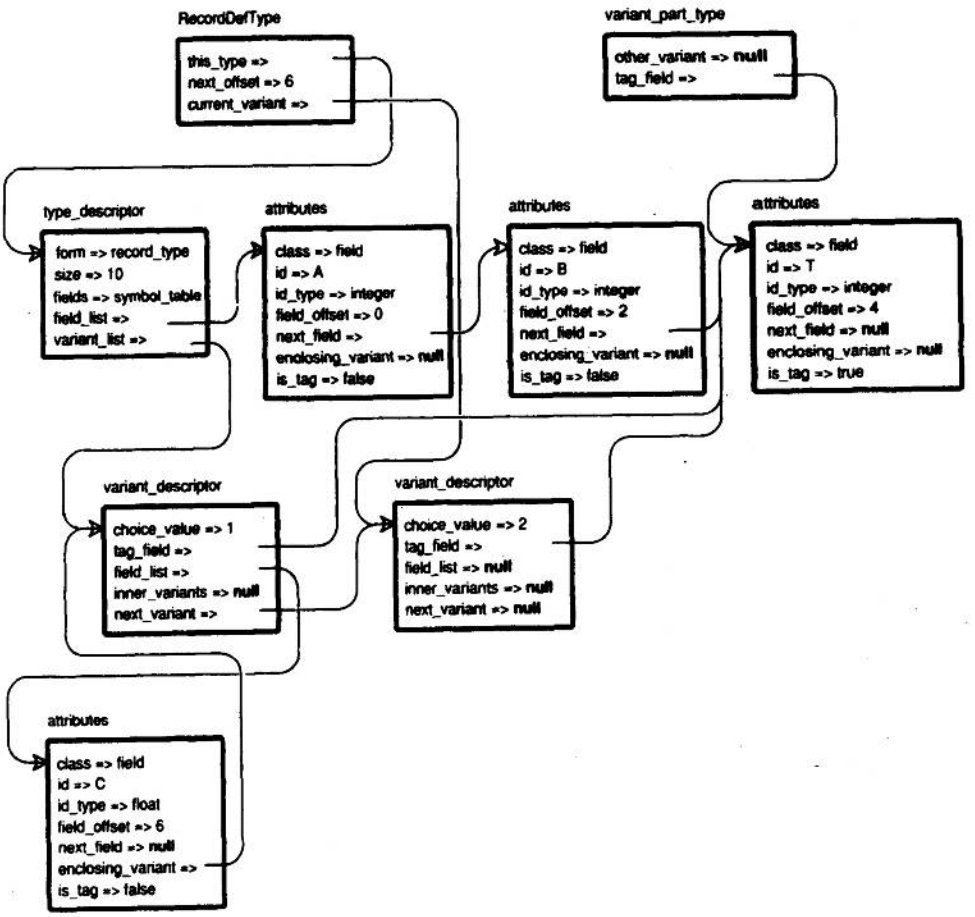


图10-15 变体处理时的RECORDDEF语义记录

就像它的名字所提示的，end_variant_part()将在成员列表中变体部分的末尾处被调用。可通过域引用**record**.record_def.current_variant来访问变体部分的struct variant_des记录列表。在把这个列表与适当的包含结构（整个记录或者是外层的成员列表）的描述符联系后，end_variant_part()恢复由tag_field()保存的**record**.record_def.current_variant的值：

362
363

```
end_variant_part(record, case, <choice>) => record
{
  Let RD name the semantic record associated with record
```

```
if (case.variant_record_part.outer_variant == NULL)
   RD.this_type.variant_list ← RD.current_variant;
else
   case.variant_record_part.outer_variant.inner_variants
      ← RD.current_variant;
RD.current_variant ←
      case.variant_record_part.outer_variant;
record ← the updated RD;
}
```

为了少许简化前面的讨论和数据结构，我们强行限制了每个变体仅有单一的标号。而大多数允许变体记录的语言同时允许变体有多重标号。可以在struct variant_des中添加两项扩展来处理多重标号，其一是必须将域choice_value从类型long改为到值列表的指针，其二是必须添加一个sequence_number域。列表中每个变体都有惟一的序列号。这个值在运行时用来替代标签域的值以检查变体的合法性。

为描述变体记录所构造的数据结构如图10-16所示。

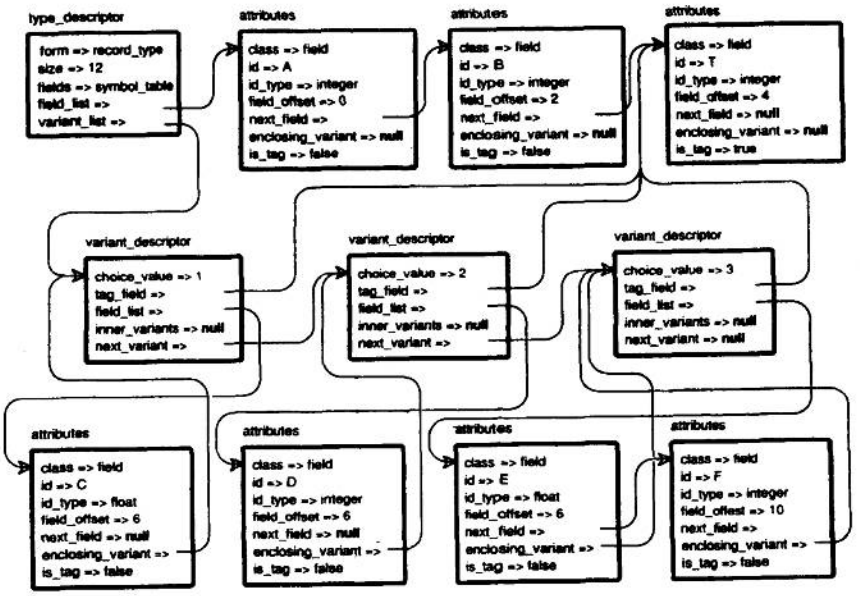


图10-16 变体记录的type_descriptor

10.3.6 访问类型

Ada中动态分配的对象可以通过访问（access）类型来引用，访问类型等价于其他语言中的指针或引用类型。编译访问类型的声明相当简单；而提供它们所需的运行时支持却非常复杂。这里所需的支持包括在第9章里讨论过的堆分配机制。

364

我们感兴趣的语法是：

```
<type definition> → access <subtype> #access_type
<type decl>       → type IDENTIFIER #incomplete_type
```

为处理访问类型，需要在type_descriptor中添加一个新的变体：

```
/* form == ACCESSTYPE */
struct type_des *reference_type;
```

而为处理不完整的类型，需要添加另一个新的变体：

```
/* form == INCOMPLETETYPE */
struct type_des *dependent_type;
```

在Ada和Ada/CS中引入不完整的类型声明，用来处理访问类型声明中的前向引用——例如，用来定义自引用类型。下面取自Ada语言参考手册的例子说明了不完整类型的使用：

```
type Cell;  — incomplete type declaration
type Link is access Cell;  — Cell must already be declared

type Cell is
 record
  Value : Integer;
  Succ : Link;
  Pred : Link;
 end record;
```

所需的语义例程描述如下。其中，access_type()和incomplete_type()均建立type_descriptor记录。此外，incomplete_type()还建立一个符号表条目，access_type()检查所引用的类型是否为不完整的类型。如果是，则将正在声明的类型保存为依赖这个不完整类型的类型。

```
access_type(<subtype>) => <type definition>
{
  Create a new type descriptor, T, for an access type:
  (type_descriptor) {
     .form = ACCESSTYPE;
     .referenced_type = <subtype>.type_ref.object_type;
     .size = ACCESSSIZE; }
     /* ACCESSSIZE is target machine dependent */
  if (T.referenced_type.form == INCOMPLETETYPE)
     T.referenced_type.dependent_type = T;

  <type definition> ← (struct type_ref) { .object_type = T; };
}


incomplete_type(IDENTIFIER)
{
  Create a new type descriptor, T, for an
     incomplete type:
  (type_descriptor) {
     .form = INCOMPLETETYPE;
     .dependent_type = NULL;
     .size = 0; }    /* a meaningless value */

  The identifier is entered into the symbol table with
  the following associated attributes record:

  (attributes) { .class = TYPENAME;
     .id_type = T;
     .id = the id_entry returned by enter(); }
}
```

不完整类型出现的可能性意味着对前面概述过的例程type_decl()必须加以扩展以检查不完整类型所试图声明的类型名是否已在符号表中。如果是的话，不完全类型的type_descriptor中的dependent_type域将用来更新使用这个不完整类型所定义的访问类型的类型描述符。然后，INCOMPLETETYPE type_descriptor将被放回到与类型名相关联的属性记录中。为妥善起见，我们应当关注针对不完整类型的多重依赖类型的处理以及检查所有的不完全类型是否在它们出现的作用域结束前有完整的定义。练习5和6提出了处理这些问题的技术。

10.3.7 包

包允许将任意的声明集合组装在一起并用一个名字来命名它们。Ada中的包声明包括两部分：规范

365

部分（specification part）和包主体（body）。规范部分中的声明不同于那些出现在私有部分（private part）中的声明，它们在包外部是可见的。而出现在私有部分或包体中的声明只有在包的内部是可访问的。Ada/CS允许另外一种形式，即把这两部分合并形成单个的声明（像Modula-2中的模块）。这种形式
366 的编译涉及较为简单的符号表处理。描述包规范部分、包主体以及单个的声明形式的产生式如下：

```
<package declaration>      → package <package spec or body> ;

<package spec or body>     → <id> #start_package is { <declaration> }
                             #end_visible_part [ <private part> ] <body option>
                             end <id option> ; #end_package
<package spec or body>     → body <id> #start_package_body is
                             { <body declaration> } [ <statement> } end
                             <id option> ; #end_package

<body option>              → [ body { <body declaration> } [ <statement> }
                             #body_present ]
<id option>                → [ <id> #check_package_id ]
```

包编译所涉及的主要问题是处理在包中各个部分引入的名字的可见性。出现在规范部分中的名字（不包括出现在私有部分中的名字）和过程的参数一样扮演着双重角色。如果采用包名来修饰限制，它们可以在包的外围作用域中可见，这就像记录域可以在包含记录声明的作用域中可见一样。它们也可以直接在包主体中可见。此外，如果包名在use子句中出现，则它们可以在外围作用域中直接可见。

这里需要一个新的语义记录类型：

```
struct package {
   boolean body_seen;
   attributes *old_current_package;
};
```

attributes记录必须包括用于处理包的一个变体：

```
/* class == PACKAGENAME */
symbol_table scope;
```

相关的语义例程描述如下。start_package()将包名放在符号表中并设置一个名为current_package的全局变量来引用相关联的attributes记录。它也为包创建一个新的作用域并使之成为新的当前作用域：

```
start_package(<id>) => <package spec or body>
{
   Enter <id> in the symbol table of the current scope.
   Build a PACKAGENAME attribute record for it, creating
     a new symbol table to provide a value for scope.
        <package spec or body> ← (struct package) {
             .body_seen = FALSE;
367          .old_current_package = current_package; }
   Assign a pointer to the new attribute record to
      current_package.
   Call open_scope() to make current_package->scope
      the current scope.
}
```

那些在调用end_visible_part()之前在包中声明的标识符组成包的导出名字集。

```
end_visible_part(void)
{
   Mark all identifiers declared so far in the current
   scope as exported.  (The scope is referenced by
   current_package->scope)
}
```

例程end_package()在规范部分及包主体结束的地方将分别被调用。它使用由start_package()所创建的语义记录中的信息并参考由current_package所引用的attributes记录。最后，它将current_package重置为调用start_package()之前的值：

```
end_package(<package spec or body>)
{
   if (<package spec or body>.package.body_seen) {
      Check that all procedures specified in the
         visible part have had a corresponding
         body declaration.
      Delete all but the exported names from the
         current_package->scope
   }
   Exit current symbol table scope, retaining the scope
      as an attribute of current_package;
   current_package ←
      <package spec or body>.package.old_current_package
}
```

在包主体开始的地方将调用start_package_body()。它希望找到与它的<id>参数相关联的PACKAGENAME attributes记录:

```
start_package_body(<id>) => <package spec or body>
{
   Find <id> in the symbol table and check that it names
      a package
   <package spec or body> ← (struct package) {
            .body_seen = TRUE;
            .old_current_package = current_package; };
   Set current_package to the attribute record found
      in the symbol table
   Make the scope found in the attribute record the
      current scope
}
```

368

check_package_id()是一个简单的动作例程，用来确保出现在包规范或主体结尾处的标识符同包的名字匹配:

```
check_package_id(<id>)
{
   Check that <id> names current_package
}
```

单模块的包形式中使用的body_present指示包体已被处理，以此来区别单模块包与包的规范:

```
body_present(<package spec or body>) => <package spec or body>
{
   <package spec or body>.package.body_seen = TRUE;
}
```

过程规范能出现在包规范的可见部分，它们相应的过程体则出现在包主体中。其语法如下:

```
<declaration> → <subprogram specification> ; #end_proc_spec
```

end_proc_spec()将在上述规范结尾处被调用。由于它所做的处理与其他处理过程的动作例程的工作相关，因此我们将在第13章而不是在本节里讨论它。它所引入的主要扩展是，要求处理过程声明的动作例程适应在符号表中先前已存在的过程名条目，其属性记录仅描述了过程的规范。这其中最为复杂的工作是检查参数描述符列表的匹配问题。

私有类型和私有部分

以下声明形式只有在包的可见部分才是合法的:

```
<private type declaration> → type <id> is private #private_type_decl
```

这就像一个不完整类型声明，因为它把类型名填入符号表时却没有指定它所表示的类型的结构。私有类型的完整类型声明必须出现在同一个包规范的私有部分。只有类型名在包外可见，而相应的类型声明的细节也只能在包主体中是可访问的。

369

因此，private_type_decl()将<id>填入符号表并使用type_descriptor的以下扩展为它建立

合适的类型描述符:

```
struct {   /* form == PRIVATETYPE */
   struct type_des *the_type;
   attributes *containing_package;
};


private_type_decl(<id>)
{
   The identifier is entered into the symbol table with
   the following associated attributes record:

   (attributes) {
      .class = TYPENAME;
      .id_type = (type_descriptor) {
            .form = PRIVATETYPE;
            .the_type = NULL;
            .containing_package = current_package; } ;
      .id = the id_entry returned by enter() ; }
}
```

包规范的私有部分仅可以包含子类型与类型声明:

```
<private part>   → private <private item> { <private item> }
<private item>   → subtype <id> is <subtype definition> ;
                 → type <id> is <non-private type definition> ;
```

在私有部分，我们可以调用type_decl()来寻找已声明为PRIVATETYPE的标识符。这种情况下，作为参数传递给type_decl()的类型描述符可用来填写已存在的类型描述符的the_type域。任何时候在使用类型描述符时，我们都必须考虑私有类型这个特殊的情况。仅当类型描述符的containing_package域等于current_package时，我们可以使用the_type域来访问类型表示的细节。

10.3.8 attributes和semantics_record结构

图10-17和图10-18显示了最终版本的attributes和semantics_record结构，它们包括了为处理
370 在本章讨论的各种特性而建议添加的所有扩展。

```
#include "symtab.h"   /* see Chapter 8 */

typedef short boolean;

enum id_class { CONST, VARIABLE, ENUMCONST, FIELD,
   TYPENAME, PACKAGENAME };

enum type_form { INTEGERTYPE, FLOATTYPE, STRINGTYPE,
   ENUMTYPE, ARRAYTYPE, RECORDTYPE, SUBTYPE,
   PRIVATETYPE, ACCESSTYPE, INCOMPLETETYPE,
   ERRORTYPE };

typedef unsigned long address_range;

struct address {
   short var_level;
   address_range var_offset;
   boolean indirect, read_only;
};

typedef struct attributes {
   id_entry id;
   struct type_des *id_type;
   enum id_class class;
   union {
      /* class == CONST */
```

图10-17 最终版本的attributes结构

```
        struct value_type value;

        /* class == VARIABLE */
        struct address var_address;

        /* class == FIELD */
        struct {
           address_range field_offset;
           struct attributes *next_field;
           struct variant_des *enclosing_variant;
           boolean is_tag;
        };

        /* class == TYPENAME --- empty variant */

        /* class == ENUMCONST */
        struct {
           unsigned long enum_value;
           struct attributes *next_const;
        };

        /* class == PACKAGENAME */
        symbol_table scope;
     };
  } attributes;

  typedef struct type_des {
     address_range size;
     enum type_form form;
     union {
        /* form == INTEGERTYPE, FLOATTYPE, STRINGTYPE,
              ERRORTYPE -- empty variant */

        /* form == ENUMTYPE */
        attributes *first_const;

        /* form == ARRAYTYPE */
        struct {
           struct index_list *indextypes;
           struct type_des *element_type;
        };

        /* form == RECORDTYPE */
        struct {
           symbol_table fields;
           attributes *field_list;
           struct variant_des *variant_list;
        };

        /* form == SUBTYPE */
        struct {
           struct type_des *base_type;
           struct constraint_des constraint;
        };

        /* form == ACCESSTYPE */
        struct type_des *referenced_type;

        /* form == INCOMPLETETYPE */
        struct type_des *dependent_type;

        /* form == PRIVATETYPE */
        struct {
           struct type_des *the_type;
           attributes *containing_package;
        };
     };
  } type_descriptor;

  enum constraint_form { DYNAMICRANGE, STATICRANGE,
        UNCONSTRAINEDINDEX, ARRAYBOUNDS };
```

图10-17 （续）

```
struct constraint_des {
   enum constraint_form form;
   union {
      /* form == DYNAMICRANGE */
      address_range address;

      /* form == STATICRANGE */
      struct {
         long lowerbound;
         long upperbound;
      };

      /* form == UNCONSTRAINEDINDEX --- empty variant */

      /* form == ARRAYBOUNDS */
      struct {
         struct index_list *bounds;
         struct address dope_vector_addr;
      };
   };
};

struct index_list {
   type_descriptor *index_type;
   struct index_list *next;
};

struct variant_des {
   long choice_value;
   attributes *tag_field, *field_list;
   struct variant_des *inner_variants;
   struct variant_des *next_variant;
};

extern attributes *current_package;
```

图10-17 （续）

```
struct id { string id; };

struct type_ref { type_descriptor *object_type; };

struct record_def {
   type_descriptor *this_type;
   address_range next_offset;
   struct variant_des *current_variant;
};

struct range { long lower, upper; };

struct const_option { boolean is_constant };

struct enum_def {
   type_descriptor *this_type;
   attributes *last_const;
};

struct index_list_type {
   struct index_list *list;
   boolean constrained;
};

struct variant_part {
   struct variant_des *outer_variant;
   attributes *tag_field;
};

struct package {
   boolean body_seen;
   attributes *old_current_package;
};
```

图10-18 最终版本的semantic_record结构

```
enum semantic_record_kind { ID, TYPEREF, RANGE,
      CONSTOPTION, RECORDDEF, ENUMDEF, INDEXLIST,
      VARIANTPART, PACKAGE, MARK, ERROR };

struct semantic_record {
  enum semantic_record_kind record_kind;
  /* initialize to ERROR */
  union {
    struct id id;                  /* ID */
    struct type_ref type_ref;      /* TYPEREF */
    struct range range;            /* RANGE */
    struct const_option const_option;  /* CONSTOPTION */
    struct record_def record_def;      /* RECORDDEF */
    struct enum_def enum_def;          /* ENUMDEF */
    struct index_list_type index_list; /* INDEXLIST */
    struct variant_part variant_part;  /* VARIANTPART */
    struct package package;        /* PACKAGE */
    /* empty variant */           /* MARK */
    /* empty variant */           /* ERROR */
  };
};
```

图10-18 （续）

371
~
374

练习

1. 概述一个算法，用于测试采用图10-17中定义的形式的type_descriptor记录所表示的两个类型是否结构等价。
2. 使用图10-15中定义的type_descriptor记录，给出为描述下列类型定义而创建的结构：

a.
```
record
    I, J : Integer;
    A : array (1..10) of Float;
    X, Y, Z : Float;
end record;
```

b.
```
array (1..10) of
    record
        A, B : array (2..20) of Integer;
        F : Float;
    end record;
```

375

c.
```
record
    K : Integer;
    R : record
        S, T : Float;
        end record;
    V : Float;
end record;
```

3. 为以下声明构造相应的基于图10-17中声明的attributes记录。

```
A : constant Integer := 10;
B : Integer;
C : Integer := B+10;
```

4. 使用图10-17中定义的type_descriptor记录，建立用来描述以下类型和子类型定义的结构：

a.
```
(Monday, Tuesday, Wednesday, Thursday, Friday);
```

b.
```
Integer range 1..10;
```

c.
```
record
    I, J : Integer;
    A : array (1..10) of Float;
    X, Y, Z : Float;
end record;
```

d.
```
record
    K : Integer;
    case T : Integer is
       when 1 => A : Integer;
       when 2 => B : Float;
       when 3 => C : Integer;
                 D : Float;
    end case;
end record;
```

5. 描述应如何改变用于处理不完整类型（见10.3.6节）的type_descriptor和动作例程以应对针对一个不完整类型声明的多重依赖类型。
6. 描述为检查所有在包的可见部分声明的私有类型在相应的包私有部分是否有完整的定义而需对数据结构和动作例程所做扩展。
7. 在以下的类型声明中，假定T是未定义的类型。试解释此时在语义例程中所必须采取的错误处理方法。

```
type A is array (1..10) of array (1..10) of T;
```

8. 描述为把布尔类型添加到10.2.2节描述的预定义类型的列表中，需要对数据结构做何种改变以及添加何种预定义的标识符。
9. 给出练习4a中枚举类型的type_descriptor的构造步骤。确定该结构每一步的改变所需的语义例程调用。
10. 举出类型声明的示例来说明图10-17中定义的struct constraint_des记录的四种变体的使用。
11. 给出以下类型的type_descriptor结构的图示：

```
type FloatArray is array (1..N) of Float;
```

其中N是整型变量。

12. 给出以下类型的type_descriptor结构的图示：

```
type ArrayRef is access FloatArray;
```

376 其中FloatArray在练习11中定义。

13. 扩展图10-15中的变体记录以包含一个嵌套在变体中的变体部分。给出在内层变体部分中调用new_variant()之后的RECORDDEF语义记录和相关联的结构（像图10-15中那样）。
14. 对于你在练习13中定义的变体记录，（像图10-16中那样）给出完整的type_descriptor结构的图示。
15. 详细描述在10.3.5节中为了允许变体有多个标号而对语法、数据结构和动作例程所做的必要修改。
16. 包有两个需要借助符号表的实现来特殊处理的性质：a) 在包中声明的一个标识符子集是导出的；b) 包的作用域，也许包含了某些隐藏的标识符，必须在规范部分和包主体的编译时间的间隙内加以保存。根据这些需求，你将如何设计符号表的实现来适应包呢？
17. 在10.3.7节提出的私有类型的实现由于Ada/CS中存在的包不可以嵌套的事实而在某种程度上得以简
377 化。这种限制允许做什么样的简化？描述应如何扩展已有的技术以支持包的嵌套。

第11章 处理表达式和数据结构引用

11.1 概述

处理表达式和数据结构引用涉及到各种各样的任务，这些任务有一个主要的共同点：它们均产生数 378
据对象。本章就从定义用于描述这些数据对象的新的语义记录开始。这个记录比较复杂，因为它将用于描述文字常量、变量、复合结构的成员以及表达式。而这些情况之所以可以用一种记录类型来表示，是因为它们在许多不同的语法上下文中可以互换使用。

```
enum object_form { OBJECTVALUE, OBJECTADDRESS };

typedef struct data_object {
   type_descriptor *object_type;
   enum object_form form;
   union {
      /* form == OBJECTVALUE */
      struct value_type value;

      /* form == OBJECTADDRESS */
      struct address addr;
   };
} data_object;


enum value_kind { INTKIND, FLOATKIND,
                  STRINGKIND, COMPOSITEKIND };

struct value {
   enum value_kind kind;
   union {
      /* kind == INTKIND */
      long int_value;

      /* kind == FLOATKIND */
      double float_value;

      /* kind == STRINGKIND */
      struct string *string_value;

      /* kind == COMPOSITEKIND */
      struct composite *composite_value;
   };
};
```

记录struct value中的最后两个变体使用了尚未定义的类型。我们将在本章合适的小节内讨论这部分内容。

我们选择的数据对象的表示假设我们想让编译器前端为变量分配地址。因此，编译器前端在某种程度上是与机器相关的，它依赖于有关信息，如应该为任何基本数据类型的数据项预留多少空间等。它还 379
依赖于实现数组和过程等语言特性的运行时存储模型。这些依赖性是一遍编译器或者是含有简单的代码生成阶段的编译器所特有的。

为使编译器更具可移植性，可以在前端和代码生成器之间采用更加抽象的接口。为此，可以使用符号表属性的引用作为数据对象的表示。对象的值和地址将做同样的处理，这就意味着文字常量将和标识

符一样在符号表中表示。符号表中的属性记录将不含任何偏移地址直到在代码生成器中的地址分配阶段产生它们。因此在前面一段中所列出的机器依赖性将从前端中去除，但整个符号表还必须为代码生成器所用。

11.2 简单名字、表达式和数据结构的动作例程

11.2.1 处理简单标识符和文字常量

作为表达式操作数的标识符在语法上首次出现在非终结符<name>的产生式中。那个产生式的右部还包括了非终结符<name suffix>可选的出现。在本节里考虑不使用<name suffix>的情况。

```
<name>        → <simple name> [ <name suffix> ]
<simple name> → <id> #new_name
```

回忆在第10章中我们所介绍的，非终结符<id>包括了对产生ID语义记录的例程process_id()的调用。这个语义记录包含了标识符记号的串的表示。new_name()在符号表中搜索该标识符并基于它的属性产生DATAOBJECT语义记录：

```
new_name(<id>) => <simple name>
{
   Find <id>.id.id in the symbol table and acquire its
      attributes.  (For now we will assume that it is
      a variable; other possibilities will be considered
      later.)

   if (there is an entry for <id> in the symbol table)
     <simple name> ← a DATAOBJECT record with
        appropriate information extracted from
        the symbol table attributes.
   else
     <simple name> ← an ERROR record
}
```

380

11.2.2节中表达式的产生式序列可用来编码运算符优先级和结合性，它开始于非终结符<primary>的两个产生式。其中一个产生式引入了另外一个非终结符<literal>，以包括在表达式中出现的文字常量。名字常量可通过<name>获得。

```
<primary> → <name> #check_data_object
<primary> → <literal>

<literal>  → INTLITERAL #process_literal
<literal>  → FLOATLITERAL #process_literal
<literal>  → STRINGLITERAL #process_literal
```

check_data_object()是必不可少的，这是因为（稍后在本章讨论的）<name>的某些实例可以由一个记录而不是一个DATAOBJECT来表示：

```
check_data_object(<name>) => <primary>
{
   if (<name>.record_kind == DATAOBJECT)
      <primary> ← <name>
   else {
      Generate an appropriate error message.
      <primary> ← an ERROR record
   }
}
```

process_literal()很像process_id()，它得到由词法分析器返回的记号值并创建与之相应的语义记录。在这种情况下，该记录是一个DATAOBJECT记录，其form == OBJECTVALUE并且它的值也被适当地记录下来。在变体object_value中使用的记录struct value_type包含文字常量的编译时

表示。最后，还必须有可用的运行时表示以允许这些常量用于表达式或其他上下文中。在代码生成的某个时刻，我们要么决定把文字常量用作指令中的立即操作数，要么在常量数据区中为它们分配存储空间。（我们可以用文字常量的值来初始化这样的数据区并静态分配它们。）

到这种运行时表示的转换时刻越迟，编译器前端也就越独立于目标机器。此外，保持文字值的源语言形式或在宿主机上的表示可用，可以增加优化的可能性，例如在编译时计算常量表达式。利用面向宿主机的文字常量表示在某种程度上是一种折中的办法。例如，整型或浮点型文字常量到宿主机表示的转 381
换就假设知道目标体系结构的某些特性，尤其是所允许的数值范围的某些信息。正如在讨论DATAOBJECT语义记录时所提到的，最抽象的方法是将文字常量以串的形式存放在符号表中并将它们作为符号表引用的结果传递给代码生成器。

11.2.2 处理表达式

以下是描述Ada/CS表达式的产生式集合，其中不包括能够定义运算符非终结符的产生式：

```
<expression>        → <relation> { <logical op> <relation> }
<expression>        → <relation> { and then <relation> }
<expression>        → <relation> { or else <relation> }
<relation>          → <simple expression>
                        [ <relational op> <simple expression> ]
<simple expression> → [ <unary adding op> ] <term> { <adding op> <term> }
<term>              → <factor> { <multiplying op> <factor> }
<factor>            → <primary> [ ** <primary> ]
                    → not <primary>
                    → abs <primary>
<primary>           → <literal>
                    → <name>
                    → ( <expression> )
```

上述若干产生式有着相似的结构只是成员有所不同。这些结构相似的产生式的语义处理也相当类似，但有一个例外。包含**and then**和**or else**的<expression>产生式描述了将在第12章里讨论的*短路计算*（short-circuit evaluation）。除去这些产生式并添加语义动作符号后，我们得到如图11-1所示的表达式文法。

```
<expression>        → <relation> { <logical op> <relation> #eval_binary }
<relation>          → <simple expression>
                        [ <relational op> <simple expression> #eval_binary ]
<simple expression> → <unary term> { <adding op> <term> #eval_binary }
<unary term>        → <unary adding op> <term> #eval_unary
<unary term>        → <term>
<term>              → <factor> { <multiplying op> <factor> #eval_binary }
<factor>            → <primary> [ ** #process_op <primary> #eval_binary ]
                    → not #process_op <primary> #eval_unary
                    → abs #process_op <primary> #eval_unary
<primary>           → <literal>
                    → <name> #check_data_object
                    → ( <expression> )

<logical op>        → and #process_op
                    → or #process_op
<relational op>     → = #process_op
                    → /= #process_op
                    → < #process_op
                    → <= #process_op
                    → > #process_op
                    → >= #process_op
<adding op>         → + #process_op
                    → - #process_op
                    → & #process_op
<unary adding op>   → + #process_op
                    → - #process_op
<multiplying op>    → * #process_op
                    → / #process_op
                    → mod #process_op
```

图11-1 含有动作符号的表达式文法

我们重写了<simple expression>的产生式——通过引入新的非终结符<unary term>来简化可选的一元运算符的处理。

这些产生式仅引入了三个新的语义例程：process_op()、eval_unary()和eval_binary()。process_op()是其中最简单的，它产生的语义记录包含了刚刚由词法分析器返回的运算符记号。该运算符记号被其他两个例程之一用来为操作数的记号和类型选择合适的元组运算符。记号条目所需的新的语义记录类型是：

382

```
struct token { token operator; };
```

其他两个例程接收TOKEN和DATAOBJECT记录作为参数并生成适当的元组。每个例程总是产生DATAOBJECT记录来描述由操作数和运算符语义所确定的表达式计算结果。这种对DATAOBJECT记录的一致性使用使得语义例程可以处理任意复杂的表达式。动作例程eval_unary()和eval_binary()的描述见图11-2和图11-3。符号<result>将根据调用eval_unary()和eval_binary()的特定产生式，表示<expression>、<relation>、<simple expression>、<unary term>或者<factor>。

```
eval_unary(<operator>, <operand>) => <result>
{
   tuple_operator tuple_op;
   type_descriptor *result_type;
   struct address T;

   select_unary_operator(<operator>.token,
                  <operand>.data_object,
                  & tuple_op, & result_type);
   if (tuple_op == NONE)
      <result> ← an ERROR record
   else if (un_no_code_needed(tuple_op, & <operand>)) {
      /*
       * un_no_code_needed() checks whether the
       * unary expression can be evaluated at
       * compile-time.  If it can, it does so
       * and updates <operand> to reflect
       * the result.
       */
      <result> ← <operand>
   } else {
      T = get_temporary();
      generate(tuple_op, <operand>.data_object, T, "");
      <result> ← (data_object) {
                  .form = OBJECTADDRESS;
                  .object_type = /* result type */;
                  .addr = T; }
   }
}
```

图11-2 动作例程eval_unary()

这些语义例程调用select_binary_operator()、get_temporary()、bi_no_code_needed()、

383 un_no_code_needed()以及select_unary_operator()。除了选择元组运算符外，这两个运算符选择例程必须查看任一操作数是否为ERROR记录并确保操作数类型和运算符记号兼容——即，在语言中是否存在着此类操作。在出现类型错误时，例程必须生成合适的错误信息。如果其中任何一项检查失败，则选择例程应当返回一个特殊的运算符以标志错误情况。（此时上述代码返回运算符NONE。）如果语言允许隐式类型转换作为表达式的一部分（例如，在Pascal中将一个整型数和一个实型数相加），那么选择例程将生成元组来实现这样的转换。

generate()例程是我们在语义例程描述中将使用的代码生成器的简化的接口。此例程将用作一个重载的名字，表示多种由其参数个数及参数类型所区分的代码生成例程，而不仅仅代表单个的例程。这

384 些参数中的第一个总是元组运算符（如7.3.3节中所定义的）。其他的参数通常是一些串、地址和

data_object结构的组合。

```
eval_binary(<operand1>, <operator>, <operand2>) => <result>
{
   tuple_operator tuple_op;
   type_descriptor *result_type;
   struct address T;

   select_binary_operator(<operand1>.data_object
                          <operator>.token,
                          <operand2>.data_object,
                          & tuple_op, & result_type);
   if (tuple_op == NONE)
      <result> ← an ERROR record
   else if (bi_no_code_needed(& <operand1>,
                      tuple_op, <operand2>)) {
      /*
       * bi_no_code_needed() checks whether the binary
       * expression can be evaluated at compile-time.
       * If it can, it does so and updates <operand1>
       * to reflect the result.
       */
      <result> ← <operand1>
   } else {
      T = get_temporary();
      generate(tuple_op, <operand1>.data_object,
               <operand2>.data_object, T);
      <result> ← (data_object) {
               .form = OBJECTADDRESS;
               .object_type = /* result type */;
               .addr = T; }
   }
}
```

图11-3 动作例程eval_binary()

运算符选择例程的描述将假设不存在通过分析操作数类型还无法解析的重载运算符。大多数程序设计语言为符号+和*定义了多种含义，这些符号被称为可重载的（overloaded）。通过分析相应操作数的类型可以确定这些符号每次出现时的确切含义（例如，确定+代表整数加法还是浮点数加法）。Ada和C++允许用户使用自定义的函数来重载标准的运算符，即：表达式的上下文可能需要用来选择运算符的合适的含义。在后面的章节中，我们将考虑处理此问题的算法。

例程un_no_code_needed()和bi_no_code_needed()以及它们在if语句分支上的调用不是严格必需的。我们可以用它们实现某种简单的、与机器无关的优化，如在操作数均为常量时进行编译时计算 385
或者针对一元加法操作不生成代码。因此，根据需要，这些例程可以很简单也可以很复杂，这取决于此时在源语言级还是稍后在中间表示上进行某种优化。

临时变量管理

例程get_temporary()在例程eval_binary()和eval_unary()中的调用表明，表达式的语义例程将涉及临时变量分配例程。临时变量管理的细节将在涉及代码生成的第15章里讨论。在大多数情况下，寄存器将用作临时变量，而寄存器管理是代码生成中必不可少的部分。

例程get_temporary()为操作结果提供名字或存放位置。在这一级抽象中，临时变量基本上被认为是可以无限制提供的虚拟寄存器。代码生成器将临时变量映射到真实的寄存器或内存位置。get_temporary()，如其在eval_unary()和eval_binary()中的调用所表明的，将在每次被调用时返回一个struct address。我们必须从struct address中选择var_offset和var_level域的值的某种编码来区别临时变量和一般变量。例如，允许var_level为负值来表示临时变量，而此时var_offset域将用来保存惟一的临时变量编号。另外一种描述则是为临时变量采用不同的变体，但稍后我们会看到，同其他地址一样，也需要将indirect标志应用于临时变量上。

请求临时变量实质上就是匿名声明。正如所预计的那样，我们必须要提供描述临时变量使用情况的信息。至少需要提供临时变量的大小。由于许多机器提供了不同的寄存器类，因此机器级的类型信息（整数、浮点数和地址）可用来简化和改善了寄存器的分配。

通常，优先选择或者某种提示（诸如分配一个寄存器或存储位置）允许在临时变量指派过程中使用语义例程来辅助代码生成器。*寄存器优先*（register preference）表明临时变量在其分配后可能经常或不久即被引用，为此将它指派到寄存器是有益的。类似地，*存储器优先*（storage preference）则表明将临时变量分配到主存中是必须的（例如，需要创建指向临时变量的指针）。那些语义例程可直接使用的信息常常会在临时变量指派阶段被丢弃，因此，这些优先规则无疑有助于产生高效的目标代码。

一旦分配后，临时变量就必须保留到它的值不再需要时为止。*非活跃变量*（dead variable）（和非活跃临时变量）是那些它们的值不再需要的变量；我们可以仔细分析程序流来发现它们。然而，很少有非优化编译器去执行这些必要的流分析，因此，它们都做好了最坏情况下的假设。对变量而言，仅当它们
386 的声明作用域退出时，其存储空间才被释放。而对于临时变量，由于没有显式的作用域规则可供使用，临时变量管理器要么必须假设临时变量一旦被引用就不再需要，要么必须依靠来自知道如何使用临时变量的语义例程的更为准确的建议。这种建议通常是例程`free_temporary()`的调用，该例程将指出那些不再需要的特定的临时变量。调用`free_temporary()`并不是绝对必要的，但没有它们则可能产生不必要的代码（去保存被认为是仍然活跃的临时变量）。

如果分配临时变量的语义例程不需要将它传递给另一个的语义例程，那么临时变量的释放将很容易。然而，临时变量却经常必须在一个例程中分配而在另一个例程中释放。例如，在计算较大的表达式时，常常使用临时变量来保存子表达式的值。一旦语义例程使用了子表达式的值，保存该值的临时变量也就可以被释放。使用该值的语义例程必须显式地执行释放操作。某些临时变量的生存期较长，这就需要不同的技术。例如，在**for loop**中的循环变量由于经常被访问，因此按照寄存器优先原则将它指派到某个临时变量中。该临时变量的名字可以存放在与循环首部对应的语义记录中以便在循环结束时调用的语义例程能够释放该临时变量。

临时变量可以保存一个计算的地址而非一个值。例如，在编译下标化变量A(I)的引用时，必须分配一个临时变量来计算并存放A(I)的地址（该临时变量类型为`struct address`，如果例程`get_temporary()`的接口允许这么做的话）。该临时变量的`struct address`记录用来表示A(I)，但该数组元素并没有存放在这个临时变量里，所存放的是它的地址。因此表示该临时变量的`struct address`记录中的`indirect`域必须设置为`TRUE`。A(I)的地址可被代码生成器和临时变量管理器识别并确实被当作临时变量对待。然而，通过这个地址间接访问而得到的A(I)的值却不是临时变量，且必须予以保留，除非有显式的赋值改变它。这样，我们就识别出两种不同的临时变量的用法，这对应着程序设计语言中名字的两种使用（*右值*（r-value）和*左值*（l-value））：一个对象可以存放在临时变量中，或它的地址可以保存到一个临时变量中。认识这种区别很重要。

11.2.3 简单的记录和数组引用

这一节将讨论简单的记录和数组引用的翻译。首先考虑的是记录域大小固定的记录。随后将研究带有常量边界的一维数组。最后将讨论那些更加复杂的数据结构，如包含动态大小的域的记录、多维数组以及非常量边界的数组。

记录域引用

在第10章里，记录定义中的语义例程连续分配所有记录域的地址。地址的连续性简化了记录赋值和
387 域引用。每个记录域在声明时即被赋予相对于记录开始位置的偏移。通常，该偏移是一个常量（在Ada/CS中该偏移始终为常量）。域的偏移（如果可能的话，在编译时）加上记录的开始地址即可得到记

录域的地址。我们必须记住的是，机器上经常有对齐限制（alignment constraint）。这意味着某些类型的记录域必须开始于某种偶地址边界，例如，双精度浮点数可能必须对齐到8字节的地址边界。

考虑下面的声明：

```
A : record
      B : Integer;
      C : Float;
    end record;
```

当处理域引用，如A.C时，

(1) 首先在符号表查找A。可得到它的地址（如，[Level = LL, Offset = Loc]）和类型（必须是记录）。

(2) 随后在该记录的类型描述符所包含的符号表中查找C。C的属性之一是它相对记录开始位置的偏移。这里，偏移值为2（假设整数大小为2个字节）。

(3) 在编译时得到A.C的位置，即A.C = (LL, Loc+2)。

下面的语法定义了域的引用和用于翻译它们的动作符号：

```
<name>           → <simple name> { <name suffix> }
<simple name>    → <id> #new_name
<name suffix>    → . <selected suffix> #field_name
<selected suffix> → <id>
```

在处理A.C时，将调用new_name()处理A并产生一个DATAOBJECT记录来描述它。例程field_name()将被调用来处理C。该例程试图将先前构造的DATAOBJECT记录解释为记录对象的描述符，并在这个包含记录所定义的符号表中查找C。最后，该例程产生新的描述该记录域的DATAOBJECT记录：

```
field_name(<name>, <selected suffix>) => <name>
{
  struct id suff_id;

  if (<name>.data_object.object_type.form != RECORDTYPE) {
     <name> ← an ERROR record
     return;
  }
  suff_id = <selected suffix>.id.id from the symbol table
     referenced by <name>.data_object.object_type.fields
  if (suff_id is not present in that symbol table) {
     <name> ← an ERROR record
     return;
  }
  Get the attributes record for suff_id from the
  record's symbol table.  Add the field_offset
  value from the attributes to the address
  described in <name>.data_object.  This can
  be done at compile-time unless the address is
  indirect, in which case a tuple to do so must be
  generated.  Adjust the address in data_object to
  describe the result.
  Set <name>.data_object.object_type to id_type
     from the field name's attributes

  <name> ← the updated DATAOBJECT record
}
```

388

引用有固定边界的数组中的元素

一维数组连续分配其元素。当数组边界由文字常量指定时，数组所需的空间可在编译时在活动记录或静态数据区中分配。考虑数组：

```
A : array(1..10) of Integer;
```

在遇到A(I)的引用时，我们计算数组第I个元素相对于数组开始地址的偏移。此例中，该偏移为I-1；而通常此偏移为 (index-lower_bound)*element_size。（C语言中的数组情况较为简单；数组下界总是零。

因此，偏移就是index*element_size。）

然后将第I个元素的偏移加上数组的开始地址即可形成所期望的地址。如果数组下标不是常量，则数组元素的地址被存放在临时变量中，如果有可能，也可存放在寄存器中。可以通过将DATAOBJECT记录中的标志indirect设置为真来标记此临时变量中存放的是地址。在一遍编译器中由于临时变量驻留在寄存器中，因而有可能做一些简单的优化。如果寄存器含有用于间接访问的地址，则通过将该寄存器用作变址或基址寄存器并且伴随的偏移值为0，就可以形成等价的直接地址。这种变换利用了在大多数计算机上可用的*变址寻址模式*（indexed addressing mode）以产生高效的数组元素访问代码。

回到A(I)的例子。我们首先分配临时变量T并将I值装入。然后减去下界1。如果数组的元素大小大于1，则必须将T值乘上元素的大小。若启用下标检查，则我们还要检查I是否为合法的下标值。接着，T要加上A的开始地址。最后，产生描述A(I)的DATAOBJECT记录。其中，object_type域是数组元素的类型，其地址为T，标志indirect为真。

389

由于下标化变量被广泛使用，因此，高效地产生数组引用代码就显得愈发重要了。如前所述，要形成A(I)的地址，我们必须先装入下标的值，再减去下界值，然后再将此偏移放大元素大小的倍数，最后再加上开始地址。对固定大小的数组而言，其下界一定是常量；为了剔除上述的减法步骤，我们可以计算数组的*虚拟起始地址*（virtual origin），即start_address- lower_bound × element_size。这也就是A(0)的地址，尽管0是非法的下标。利用虚拟起始地址可以避免显式地减去下界的操作。这样，我们仅需装入下标值，然后加上该虚拟起始地址而非实际数组地址即可。

下列产生式说明一维数组如何被包含于<name>的语法中：

```
<name>          → <simple name> [ <name suffix> ]
<name suffix>   → ( <expression> ) #process_index
```

例程process_index()同field_name()非常相似，它将新的名字后缀即下标表达式应用于先前处理过的名字前缀。不管此名字前缀多么复杂，它都由单个的DATAOBJECT记录来描述。例程process_index()的描述见图11-4。

```
process_index(<name>, <expression>) => <name>
{
  if (<name>.data_object.object_type.form != ARRAYTYPE) {
     <name> ← an ERROR record
     return;
  }
  if (<expression>.data_object.object_type does not match
      the index type of the array) {
     <name> ← an ERROR record
     return;
  }
  Allocate a temporary, T.  This temporary should be a
     register if possible.
  Let Index denote the value described by
     <expression>.data_object

  If subscript checking is enabled, generate code to check
    that L <= Index <= U where:
    L == <name>.data_object.object_type.bounds.lower and
    U == <name>.data_object.object_type.bounds.upper

  Let E_type = <name>.data_object.object_type.element_type
  Let E_size = E_type.size
  Generate code to load T with Index × E_size

  Let A_level = <name>.data_object.addr.var_level
  Let A_offset = <name>.data_object.addr.var_offset

  Generate code to add the contents of the A_level
     register to T.
  Generate code to add A_offset - E_size × L
```

图11-4 动作例程process_index()

```
      (a constant value) to T.
   <name> ← (data_object) {
      .object_form = OBJECTADDRESS;
      .object_type = E_type;
      .addr = (struct address) {
         .var_level = T.var_level;
         .var_offset = T.var_offset;
         .indirect = TRUE;
         .read_only = FALSE; } ; };
}
```

图11-4 （续）

处理同时包含数组和记录引用的复杂名字不需要额外的动作例程。例如给定A(3).BB，首先处理数组引用并产生描述A(3)的DATAOBJECT记录。随后例程field_name()将DATAOBJECT记录作为一个记录类型描述符并以此来解释BB的ID记录。BB的偏移加上A(3)的地址，即形成A(3).BB的地址。

对于像Pascal这样的语言来说，其多维数组可以看作数组的数组，因此一个像A(I, J)这样的名字可以利用本节的技术将其处理为A(I)(J)。11.3节描述了更为复杂的多维数组的处理方法。

11.2.4 记录和数组示例

图11-5a 给出一些Ada/CS的记录和数组的声明示例，其中在变量和域名的右边给出了由编译器赋予的偏移。图11-5b 则显示了翻译使用那些变量的语句而生成的元组。

Declarations	ID Offsets
I,	0
J: Integer;	2
type R **is**	
record	
X,	0
Y: Integer	2
end;	
ARecord: R;	4
A1: **array** (1..10) **of** R;	8
A2: **array** (4..6) **of array** (8..12) **of** Integer;	48

a) 记录和数组声明

```
A1(I) := ARecord;
  (SUBI, Addr(A1), 4, t1)          -- 4 = lower(A1)*element_size(A1)
  (RANGETEST, 1, 10, I)
  (MULTI, I, 4, t2)                -- element size is 4
  (ADDI, t1, t2, t3)               -- t3 contains address of A1(I)
  (ASSIGN, ARecord, 4, @t3)        -- @ indicates indirection

ARecord.Y := I;
  (ASSIGN, I, 2, @(Addr(ARecord)+2)) -- offset of Y = 2

A2(I,J) := A1(I).Y;
  (SUBI, Addr(A2), 40, t4)         -- 40 = lower(A2)*element_size(A2)
  (RANGETEST, 4, 6, I)
  (MULTI, I, 10, t5)               -- element size is 10
  (ADDI, t4, t5, t6)               -- t6 contains address of A2(I)
  (SUBI, t6, 16, t7)               -- subtract 8*2
  (RANGETEST, 8, 12, J)            -- no multiplication since
                                   -- element size is 1
  (ADDI, t7, J, t8)                -- t8 contains address of A2(I,J)
  (SUBI, Addr(A1), 4, t9)          -- 4 = lower(A1)*element_size(A1)
  (RANGETEST, 1, 10, I)
  (MULTI, I, 4, t10)               -- element size is 4
  (ADDI, t9, t10, t11)             -- t11 contains address of A1(I)
  (ADDI, t11, 2, t12)              -- add offset of Y;
                                   -- t12 has address of A1(I).Y
  (ASSIGN, @t12, 2, @t8)
```

b) 记录和数组语句及其相应的元组

图 11-5

11.2.5 串

390 ~ 391

在很多语言（如Pascal和Ada）中，串的大小是固定的，因此它们可以被简单地实现为字符数组。这种实现方式既容易又有效，但它不支持一些最有用的串操作。动态创建新串的操作，例如，从给定的串中提取子串或者连接两个或多个串，均产生新的串对象而其大小通常仅在运行时才能确定。

为了支持这些更通用的串操作，Ada/CS包含了动态串。在10.2.2节中曾讨论过，String是Ada/CS中的预定义类型。在9.3.3节中给出的基于堆的动态对象管理模式适合于实现Ada/CS的串。利用这种方法，每个串变量或常量将表示为一个指向动态创建的串对象的指针。因此常量STRING-SIZE的值，即描述在活动记录中表示一个串所需空间大小的值为4，这也就是说允许使用4个字节来存放串指针。

如果定义了适当的元组运算符，则可以将串上的操作翻译成元组。不像算术运算的元组运算符，我们不期望代码生成器将串操作元组翻译成1～2条机器指令。典型地，我们将调用库例程来执行串操作，特别是像子串或串连接那样创建新串的操作。而在某些机器的指令集中确实包含着可用来实现串操作的串操纵指令。

Ada/CS中定义的串操作包括所有基于通常比较运算符的串的词典序比较操作、由运算符&表示的串连接操作和一个预定义的子串函数——Substr(StringObject, StartPos, Length)。此外，串赋值还必须加以特别地实现，因为此时要拷贝的是串本身而非表示串的指针。需用来表示上述特性的元组运算符是

SEQ, SNE, SGT, SGE, SLT, SLE, CATENATE, SUBSTR, SCOPY

除了子串和串拷贝运算符外，其他运算符均使用三个操作数的标准元组形式。SUBSTR使用四个操作数，前三个作为函数的输入参数，第四个是存放结果的串变量或临时变量。对SCOPY而言，ARG1是拷贝的源操作数，而ARG2是目的操作数。

图11-6举例说明了串操作的翻译，其中S1、S2和S3均为串。

```
if S1 = S2 then                    (SEQ, S1, S2, t1)
   S2 := S3;                       (JUMP0, t1, L1)
else                               (SCOPY, S3, S2)
   S1 := S2 & Substr(S3,4,3);      (JUMP, L2)
end if;                            (LABEL, L1)
                                   (SUBSTR, S3, 4, 3, t2)
                                   (CATENATE, S2, t2, t3)
                                   (ASSIGN, t3, S1)
                                   -- SCOPY not necessary because
                                   -- source is a temporary
                                   (LABEL, L2)
```

392 ~ 393

图11-6 由串运算符生成的元组示例

11.3 高级特性的动作例程

11.3.1 多维数组的组织和引用

在描述数组引用的语义例程之前，必须考虑如何组织数组的存储。根据它们的存储分配需求，存在着三种形式的数组：

（1）如果数组所有的边界均为常量，那么它可以分配在静态存储、栈或者堆中。这里不需要运行时的描述符，但如果允许更一般的数组形式，那么就可能要采用运行时描述符以求得使用上的一致性。

（2）如果在作用域入口计算数组边界，则可以使用栈或堆存储。这时还需要相关的描述符。

（3）如果数组边界较灵活（即，随时可变），则必须使用堆存储并且需要内情向量或某种描述符。这样的数组可当作串来处理。

不同的数组组织区别在于如何引用单个的数组元素:

• 连续存储（Contiguous）

所有的数组元素采用连续的存储分配。最常见的编排方式有:

（1）行主序（Row Major）

这种次序对应着最右的下标变化最快，即[A(1,1),A(1,2), …, A(2,1),A(2, 2), …]。大多数现代程序设计语言（如PL/I、Algol、Pascal、C和Ada等）均采用这种次序。

（2）列主序（Column Major）

这种次序对应着最左的下标变化最快，即[A(1,1), A(2,1), …, A(1,2), A(2,2), …]。FORTRAN语言采用这种次序。

• 向量化存储（By vectors）

向量的所有的元素均相连。数组可看作是由指向子数组或向量的指针组成的向量。

在以下章节里，我们将考虑在使用不同组织形式时，如何寻址单个的数组元素。

行主序

行主序有着吸引人的特性，即我们可以将形如**array**(1..N, 1..M) **of** T的数组看成是**array**(1..N) **of** **array**(1..M) **of** T，而其中子数组也是采用行主序进行元素的连续分配。

394

假设有以下声明:

A : **array**($L_1..U_1, \ldots, L_n..U_n$) **of** T;

定义数组第j维元素的个数为:

$D_j = U_j - L_j + 1$

相对于数组第一个元素A($L_1, \cdots, L_n$)，数组元素A($i_1, \cdots, i_n$)的位置为:

$$(i_n-L_n)+(i_{n-1}-L_{n-1})D_n+(i_{n-2}-L_{n-2})D_nD_{n-1}+\cdots+(i_1-L_1)D_n\cdots D_2$$

这个计算式可以重写为:

$$i_1D_2\cdots D_n+i_2D_3\cdots D_n+\cdots+i_{n-1}D_n+i_n - (L_1D_2\cdots D_n+L_2D_3\cdots D_n+\cdots+L_{n-1}D_n+L_n)$$

上述计算公式的第二项独立于值i（数组的下标）。我们称之为con_part，因为它可以在数组的范围（L、U和D值）建立时提前计算。

上述计算公式可被再次重写为:

$$(((i_1D_2+i_2)D_3+i_3)D_4+i_4\cdots)D_n+i_n - \text{con_part} = \text{var_part} - \text{con_part}$$

于是，数组元素的地址可以表示为:

((var_part – con_part) × element_size) + array_start_adr

现在我们叙述在分析和处理下标列表时，语义例程将如何计算数组元素的地址。假设处理的是数组元素A($i_1, \cdots, i_n$)。此时var_part存放在临时变量中——很大可能是在寄存器中。下面是在处理每个下标以及右括号时产生的代码:

```
(1)     Calculate i1; var_part = i1
(2)     Calculate i2; var_part = var_part × D2+i2
...     ...
(n)     Calculate in; var_part = var_part × Dn+in
(n+1)   element_adr = array_start_adr + (var_part – con_part) × element_size
```

通常数组元素的大小为1时，就不需要在第(n+1)个计算步做乘法操作。同样，如果数组的边界是常量的话，则表达式:

array_start_adr – con_part × element_size

可以在编译时计算从而省去了最后一步中的减法操作。

395 现在我们描述用来翻译数组引用的语义例程。在语法上，数组引用开始于以下产生式：

<name> → <simple name> [<name suffix>]

而<name suffix>可能的形式有：

<name suffix> → (<expression> { , <expression> })

我们知道，在<simple name>中调用的例程new_name()（以及在其他任何版本的<name suffix>中所做的语义处理）将一个DATAOBJECT记录和<name>联系在一起。处理数组引用需用到下面这三个例程，start_index()、index()、finish_index()。它们出现在下面<name suffix>的产生式中：

```
<name suffix> → ( <expression> #start_index { , <expression> #index } )
                #finish_index
```

动作例程start_index()和index()将使用一个新的与非终结符<name suffix>关联的语义记录：

```
struct index {
   /* How many subscripts have been processed */
   unsigned int count;

   /* Address of var_part of indexed array */
   address_range var_part_adr;
};
```

例程start_index()开始为下标表达式列表计算var_part并创建描述下标计算情况的INDEX语义记录：

```
start_index(<name>, <expression>) => <name suffix>
{
   Check that the DATAOBJECT record for <name>
      describes an array
   Check that <expression>.data_object.object_type
      agrees with the type required by array's first
      subscript position
   Allocate a temporary, T, for var_part (a register,
      if possible)
   Generate code for T := <expression>
   If subscript checking is enabled, generate code to
     check that L1 ≤ <expression> ≤ U1
     (L1 and U1 are obtained from the array's dope vector)

   <name suffix> ← (struct index) {
         .count = 1;
         .var_part_adr = T; }
}
```

396 动作例程index()使用数组的DATAOBJECT记录以及由start_index()创建的INDEX记录和下一维下标表达式的DATAOBJECT记录继续进行下标计算：

```
index(<name>, <name suffix>, <expression>) => <name suffix>
{
   <name suffix>.index.count += 1;
   /* let Cnt represent this value */
   Check that Cnt does not exceed the number of
      subscripts possessed by the array
      described by <name>.data_object
   Check that <expression>.data_object.object_type
      agrees with the type required by the subscript
      at position Cnt
   Let Vp be the address stored at
        <name suffix>.index.var_part_adr
      Let DCnt represent the D value at position Cnt
         of the array's dope vector.
      Generate code for
               Vp *= DCnt ;
               Vp += <expression>;
   If subscript checking is enabled, generate code to
      check that LCnt ≤ <expression> ≤ UCnt
```

```
        (L_Cnt and U_Cnt are obtained from the array's
        dope vector.)
    <name suffix> ← the updated INDEX record
}
```

例程finish_index()完成下标计算并将一个DATAOBJECT记录和<name>相关联以描述特定的数组元素:

```
finish_index(<name>, <name suffix>) => <name>
{
    Check that <name suffix>.index.count equals the dimension
        of the array described by <name>.data_object
    Let Vp be the address stored at
          <name suffix>.index.var_part_adr
        Obtain con_part from the array's dope vector
        element_size is a constant known at compile-time
        Generate code for
            Vp -= con_part;
            Vp *= element_size;
            Vp += <name>.data_object.addr;
    <name> ← a data_object whose type is
        <name>.data_object.object_type.element_type
        and whose address is that of the temporary, Vp,
        with indirect set to TRUE (since the temporary
        contains the address of the element referenced)
}
```

397

当数组所有的边界均是编译时可计算的常量时，可以从符号表中获得所有的内情向量。此外，如果数组所有的下标均为常量，如A(1,2,3)，那么作为一种优化措施，可以折叠所有下标的计算而不生成任何代码。举例来说，考虑如下的声明:

```
A : array (1..10, 1..10, 1..20) of integer;
```

数组A所有的边界均为常量，因此con_part可由编译器计算得到:

```
con_part == 2 * 10 * 20 + 2 * 20 + 2 == 442
```

诸如A (I, J, K)形式的数组元素引用的翻译见图11-7。

```
(RANGETEST, 1, 10, I)
(ASSIGN, I, 2, t1)
(RANGETEST, 1, 10, J)
(MULTI, t1, 10, t2)
(ADDI, t2, J, t3)
(RANGETEST, 1, 20, K)
(MULTI, t3, 20, t4)
(ADDI, t4, K, t5)
(ADDI, t5, 442, t6)          -- addition of con_part
(MULTI, t6, 2, t7)           -- multiplication by element size
(ADDI, t7, Addr(A), t8)      -- t8 now contains the address of A(I,J,K)
```

图11-7 为多维数组引用产生的元组

Pascal语言（像C语言一样）认为多维数组是数组的数组。这意味着不提供全部下标的部分下标（partial indexing）情况也是合法的。此结果就是某个子数组——这是一个合法的数据对象。为处理这种特性，我们必须非常仔细且不能太快地完成下标的处理。例如，在Pascal中，A[i][j][k]是合法的且等价于A[i,j,k]，此时必须小心处理不能将第一个] 错误地解释为下标的终结。正确处理的关键就是把后面紧跟 [的] 当作等价的逗号。为此必须修改文法来推迟例程finish_index()的调用，直到出现在] 后面的记号不是 [为止。

为处理部分下标，必须修改例程finish_index()以识别可能选择子数组的情况。start_index()和index()不需要改动。但什么地址是合适的呢？对行主序数组而言，我们可以利用所有子数组均是连续分配的这一事实。因此子数组的地址就是它的第一个元素的地址。也就是说，对刚才讨论的三维数组而言，地址A[i]等价于A[i,1,1]的地址。经过适当修改的finish_index()版本如下:

398

```
finish_index(<name>, <name suffix>) => <name>
{
   Let Cnt = <name suffix>.index.count
   Check that Cnt <= n, the dimensionality of the array
       described by <name>.data_object
   If Cnt < n, then use L_{Cnt+1}, ..., L_n to complete
       calculation of the component address  (Calls to
       index() can be used to help)
   Let Vp be the address stored at
       <name suffix>.index.var_part_adr
       Obtain con_part from the array's dope vector.
       element_size is a constant known at compile-time
       Generate code for
           Vp -= con_part;
           Vp *= element_size;
           Vp += <name>.data_object.addr;
   <name> ← a DATAOBJECT record whose type is
      the type of the selected subarray
      and whose address is that of the temporary, Vp,
      with indirect set to TRUE (since the temporary
      contains the address of the element referenced)
}
```

当允许部分下标时（为得到子数组），在内情向量中包含子数组的大小信息就显得非常有用了。我们可以通过以下步骤推导出这些信息：

令S_1为整个数组的大小。

令S_2为子数组$A(L_1)$的大小。

令S_3为子数组$A(L_1, L_2)$的大小。

……

令S_n为子数组$A(L_1, \cdots, L_{n-1})$的大小。

$A(L_1, \cdots, L_n)$的大小是编译时常量，也就是数组基类型的大小。该值是存放在符号表中而不是在（运行时的）内情向量中。n维（下标）数组类型的内情向量的构成如图11-8所示。

ConPart			
D_1	L_1	U_1	S_1
⋮	⋮	⋮	⋮
D_n	L_n	U_n	S_n

图11-8 n维数组的内情向量模板

如果数组的所有边界都是常量，则上述所有信息均可存放在符号表中。否则，它们将在运行时计算并使用。如果有以下的类型声明：

```
type A is array(1..N) of Float;
```

则所有的内情向量信息，不论它们是存放在编译时的符号表中还是存放在运行时的活动记录中，都将被所有类型为A的数组共享。每一个数组都可以用它的数据的开始地址来表示。如果在变量声明中使用了数组生成器，例如：

399

```
A : array(1..N) of Float;
```

那么生成的内情向量只能用于数组A。假定有如下声明：

```
A : array(1..N,10..M) of Integer;
```

在编译时刻，我们在当前活动记录中为A的内情向量以及指向它的数据区的指针分配空间。在运行时刻，就在进入相关作用域以及必要的活动记录的操作完成后，我们执行计算数组边界的代码，填写内情向量并分配数组A的空间。假定A创建的时候，N=10，M=15。首先，我们填写这些边界，得到如图11-9所示的内情向量。

然后，我们用以下参数调用数组分配库例程：

- 数组维数（此例中为2）。
- 内情向量地址。
- 数组A的数据区指针的地址。

- 当前活动记录的stack_top的地址。在此例子中，假定stack_top值为100。
- 数组元素大小（此例中为1）。

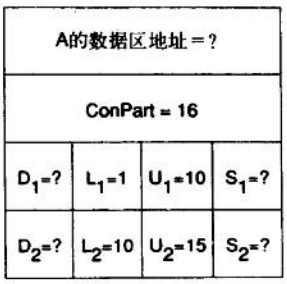
该库例程完成以下工作：

- 根据（已在内情向量中的）值L和U计算D值：$D_i = U_i - L_i + 1$。在这个例子中，$D_1=10$，$D_2=6$。
- 然后计算con_part。此例中，con_part = $L_1D_2+L_2 = 16$。
- 接着再计算值S。此例子中：

$S_2=D_2 \times element_size = 12$
$S_1=D_1 \times S_2 = 120$

- 最后，该例程为A分配的空间。它将A的数据区地址置为stack_top的当前值，然后 将stack_top增加S_1个字长。

该库例程返回时，所完成内情向量和数据区指针如图11-10所示。



（? 意味着仍未确定）

图11-9 用数组边界值初始化的内情向量

400

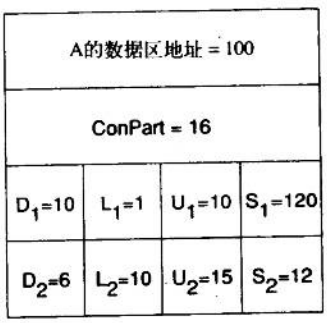


图11-10 已完成的内情向量

为处理整个数组A，我们可以使用它的地址（100）和大小（$S_1 = 120$）。在处理数组元素，如A(3,12)时，我们计算其地址为：

$(i_1D_2+i_2-con_part) \times element_size+start_address = (3\times6+12-16)\times2+100 = 128$

其中element_size是一个编译时常量（INTEGERSIZE == 2）。

为处理子数组A(3)，我们计算其地址。该地址和元素A(3,10)的地址相同：$(3 \times 6+10-16) \times 2+100 = 124$。该子数组的大小为$S_2=12$。

带有类型描述符的行主序

另一种处理多维数组的方法是将它们一律看成数组的数组（如果语言定义允许的话）。任意类型的数组在符号表中的条目给出其边界（如果已知的话）、下标类型和基类型，此基类型本身也可以是数组类型。例如，考虑以下声明：

```
A : array (1..10) of array (0..5) of array (3..4) of Integer;
```

401

上述类型将包含四个不同的类型记录条目。首先，Integer类型在其预定义条目中指出该类型的值占用一个字长。其次，类型**array** (3..4) **of** Integer在其类型记录中指出它是边界为3和4的数组类型，其基类型是整型，同时整个对象大小的是4。这个类型是*匿名的*，也就是说，它没有相应的类型名字，但我们还是要为它创建一个类型记录。如果此类型有名字T，那么T在符号表中的条目将指向它的类型记录。

为保持内容的可读性，我们将发明那么一个名字，如Type&1。当然，这个名字不会实际出现在符号表中。再者，类型**array** (0..5) **of array** (3..4) **of** Integer的类型记录（匿名类型，称为Type&2）表明它也是数组类型，边界范围为0和5，其基类型是Type&1。这种类型的对象大小为6 × 4=24。最后，A的类型将是第三个匿名类型，Type&3。它同样是数组类型，边界为1和10，基类型为Type&2，该类型对象大小为240。变量A在符号表中的条目将包括它的开始地址（在活动记录中偏移）和其他的信息，诸如，声明A的词法作用域等。（在C语言中，多于一维的数组在定义上也是数组的数组，因此可以应用上述技术。）

现在考虑编译A(3,4,4)。在看到第一个下标时，我们核对了A是数组类型以及3为合法下标。A(3)的地址是start_address(A)+sizeof(Type&2) × (3−1) = start_address(A)+48，它的类型是Type&2。在看到第二个下标时，检查了Type&2是数组类型且4是合理的下标。A(3,4)的地址为start_address(A(3)) +sizeof(Type&1) × (4−0) = start_address(A)+48+4 × 4=start_address(A) +64。类似地，A(3,4,4)的地址为start_address(A)+66，它的类型是Integer。如果分析了A(3,4)就停止处理，那我们就可以得到类型为Type&1的对象的正确地址。

402 以上我们列举了此例中最简单的情况：具有常量下标的非动态数组。如果下标为任意表达式，那么范围检查以及地址计算就必须推迟到运行时刻进行。对编译器来说，产生那些合适的代码是非常容易的。

处理动态数组时，编译器仅知道其类型而不知晓其边界。包括匿名类型在内的每一个动态类型，它们的内情向量在运行时才能构造。这些内情向量需要用来存放边界和对象大小，但不指向数组本身。我们称这些内情向量为*类型描述符*（type descriptor）。严格地讲，所有的动态或含有动态子类型的类型都需要类型描述符。若以牺牲效率为代价来换取类型处理上的一致性，那么也可以为非动态类型构造类型描述符。类型描述符就像变量一样存放在活动记录中。当进入作用域时，每个类型描述符中的边界和对象大小将被初始化。例如，假定A的声明如下：

A : **array** (1..10) **of array** (0..n) **of array** (3..4) **of** Integer;

其中Type&1不需要类型描述符，这是因为它的边界和元素大小在编译时均已明确。另一方面，Type&2 和Type&3却需要运行时的类型描述符，这是因为Type&2的边界和Type&3的大小取决于运行时n的值。图11-11a显示了在任何运行时的信息填入之前由编译器生成的类型描述符。图11-11b显示了在块入口处已初始化且包含了所有运行时信息的类型描述符，假定此时n = 23。

很多变量（如此例中的A）均在活动记录中有相应的位置。事实上，所有我们存放在那里的只是指向运行时栈中动态区域的指针。在遇到下标表达式时，编译器根据存储在类型描述符中的信息生成计算地址（和边界检查）的代码。

L= 1	U = 10	Size = ?	descriptor for Type&3
L = 0	U = ?	Size = ?	descriptor for Type&2

a) 编译器生成的类型描述符

L= 1	U = 10	Size = 960	descriptor for Type&3
L = 0	U = 23	Size = 96	descriptor for Type&2

b) 进入作用域后的类型描述符

图 11-11

我们可以很简单地实现“数组的数组”的策略。INDEX语义记录以及例程start_index()和finish_index()均不再需要。下面的产生式显示了何时必须调用修改过的index()例程：

```
<name> <name suffix>  → <name> ( <expression> #index
                                { , <expression> #index } )
```

这里，index()期待两个DATAOBJECT语义记录，其中之一代表数组，另一个则代表下标表达式。它产生描述所选数组元素的DATAOBJECT记录。 403

```
index(<name>, <expression>) => <name>
{
   Check that <name>.data_object.object_type describes
       an array.
   Check that <expression>.data_object.object_type agrees
       with the index type it requires.

   If subscript checking is enabled, generate code to
       check that L <= <expression> <= U.  L and U are
       obtained from <name>.data_object.object_type.
       The actual values may be in a run-time type
       descriptor for that type, or they may be in the
       symbol table.  If only one of the two bounds is
       known at compile-time, at least that bound might
       be kept in the symbol table, since we can
       usually generate better code with that
       knowledge.  (For machines with a range-check
       instruction, it may be more efficient to put
       even the known bound in the run-time type
       descriptor so that the instruction can be
       applied.)

   Let V be the address of the array described by
       <name>.data_object.  Let D  be the size of the
       element type of the array.  D might be known at
       compile-time or might be stored in a run-time
       type descriptor.  Allocate a temporary T and
       generate code for T = V + (<expression> - L) * D.
       Much of this computation might be done at
       compile-time.  If <expression>, L, and D are all
       known at compile-time, no code needs to be
       generated at all.

   <name> ← a DATAOBJECT record whose type is
       the element type of the array and whose address
       is that of the temporary, T, with indirect set
       to TRUE (since the temporary contains the
       address of the element referenced).
}
```

404

这种“数组的数组”方法需为每个下标做减法。在11.2.3节中为避免此类减法而提出的虚拟起始地址方法在这里并不适用，尽管在下标列表中的第一个表达式后，当index()被调用的时候，L × D可以在编译时从V的偏移中减去，假定这里V采用直接地址表示。

使用图11-11中的类型描述符，可将诸如A(I,J,K)的引用翻译如下：

```
(RANGETEST, 1, 10, I)
(SUBI, I, 1, t1)
(MULTI, t1, Type&3.size, t2)
(ADDI, Addr(A), t2, t3)              -- t3 contains the address of A(I)
(RANGETEST, 0, Type&2.upper, J)
(MULTI, J, Type&2.size, t4)
(ADDI, t3, t4, t5)                   -- t5 contains the address of A(I,J)
(RANGETEST, 3, 4, K)
(SUBI, K, 3, t6)
(MULTI, t6, 2, t7)                   -- multiply by Type&1.size
(ADDI, t5, t7, t8)                   -- t8 contains the address of A(I,J,K)
```

在此例中，使用了类型描述符中的成员，例如，Type&3.size。这样做是为了简化表达式的处理。该符号实际上将翻译为一个在编译时可计算的活动记录中的偏移，就像名为I的变量所做的那样。

向量化结构

这种结构形式有时称为代码字（codeword）方法（因为指向子数组的指针称为代码字）。

为寻址A(i_1, … i_n)，我们使用下标i_1到i_{n-1}来索引指针向量。i_n则用来索引向量元素。例如：

```
A : array(1..2,1..3,1..4) of Integer;
```

该数组的存储如图11-12所示。

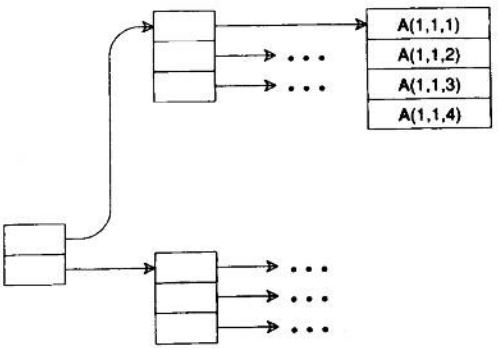


图11-12 向量化的数组结构

这种结构形式有如下特性：

- 向量不需要连续化，甚至不需要驻留在内存。此特性有助于处理非常大的数组或在地址范围或段大小受限制的机器上使用。
- 不需要做乘法操作。这种特性在乘法操作较慢时显得很有用。它也可以执行的很快，如果有特殊的指令可用来追踪那些指针。
- $D_1+D_1D_2 + \cdots + D_1 \cdots D_{n-1}$个指针的存储需要一些空间开销。

405

11.3.2 含动态对象的记录

在本节及后续的章节里，我们将考虑更复杂的可能要求包含有动态对象的记录结构。此类记录需要更复杂的处理技术。我们期望用这样一种方式来表示它们，即记录赋值可以较容易地用单个块传送指令（或等价的循环）来实现。但问题是，动态对象是用指针来表示的，如果仅拷贝这些指针，那么在被复制的记录中的动态对象可以通过源记录中指向相同对象的指针来表示，这将导致不必要的混淆。因此我们必须拷贝动态对象本身，而不是到它们的引用。

首先考虑动态数组。例如：

```
type A1 is array (1..I) of Integer;
type A2 is array (1..J) of Float;

type R is record
     B : Integer;
     C : Float;
     D : A1;
     E : A2;
     F : Boolean;
  end record;
A : R;
```

406

整个记录大小是动态变化的。可以将它存放在运行时栈中活动记录里，并留有足够的空间放置域D和E的指针。然而，在记录赋值时，这些指针将和B、C和F一起被拷贝，从而导致前面所述的不必要的混淆。一种替代方法是将整个记录放置在运行时栈上大小固定的AR之后（就像动态数组那样）。在AR中，我们将维护一个结构如图11-13所示的描述符。

此描述符可用来访问记录域和进行记录的赋值。在此记录自身的组织结构中，数组指针包含的是相

对偏移而非绝对地址。记录赋值将拷贝正常的记录域（B、C和F）和数组指针（D和E）以及动态对象本身。由于从数组指针到数组的偏移并未改变，因此所复制的数组指针也将指向这些数组的新拷贝。



图11-13 动态记录类型描述符

和以前一样，在编译时可以知道固定大小对象的位置以及数组指针相对于记录开始处的偏移。动态数组的元素放置在记录的末端。图11-14显示了记录A的结构布局。

例如，为访问A.C，我们可以：

- 使用A的描述符来得到A的开始地址。
- 加上已知的C的常量偏移值。

为访问A.E(I)，我们可以：

- 使用A的描述符来得到A的开始地址。
- 使用已知的E在记录中的偏移（此位置中存放了动态数组的偏移）和记录的开始地址计算数组的开始地址。
- 使用数组E的内情向量，和以前一样来计算(var_part-con_part) × element_size。

包含动态数组的记录和普通记录仅在以下几点有区别：

- 动态记录需要使用一层间接访问来引用动态的记录域。
- 动态记录中的数组指针包含数组元素在记录中的偏移而非绝对地址。因为这些地址是相对的，因此数组指针，作为记录的局部传送，可以毫无问题地进行赋值。

407

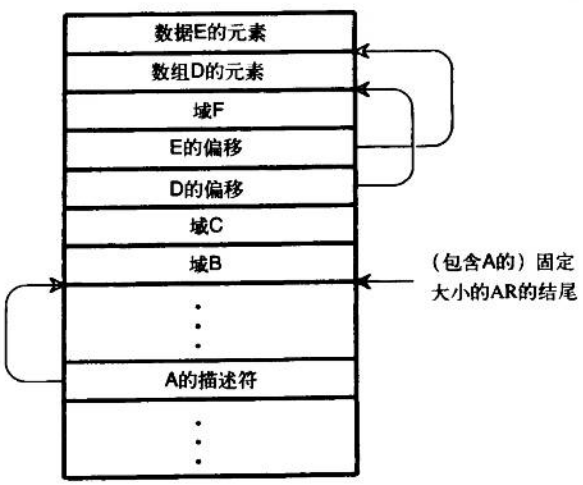


图11-14 动态记录的存储布局

含有字符串的记录的处理依然很复杂。如果使用渐增式回收或者仅允许每个指针一个引用，那么我们需要单独地拷贝每一个串。这样，记录的传送就必须翻译成一系列单独的域传送。

作为一项优化，我们可以把记录中不含串的相邻域作为一组来传送。我们甚至可以重新编排让所有非字符串的域相邻。（此重新编排的选择依赖于正被编译的语言。C语言要求结构域在内存中的放置顺序和声明中的顺序一样。然而，由于C语言没有动态数组，这个也就不成为问题，并且在拷贝结构时块传送总是可以进行的。）

如果允许串的多重引用（multiple reference）和进行垃圾回收，那么包含串的记录赋值就不需要特殊的处理。然而，在垃圾回收时，我们还是需要遍历这样的记录并标记它引用的串。类似的考虑可用
408 在字符串数组上，但这种情况下，可以很容易找到并处理那些单独的串描述符。

记录的类型描述符

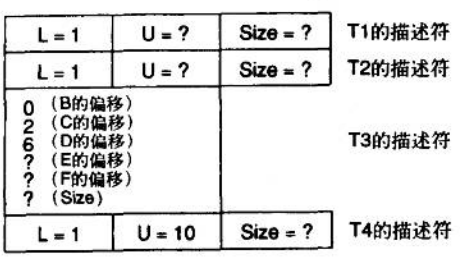
遗憾的是，刚才描述的方法在处理略为复杂的类型时就显得很不实用。例如，考虑如下的声明：

```
type T1 is array(1..I) of Integer;
type T2 is array(1..J) of Float;
type T3 is record
             B : Integer;
             C : Float;
             D : T1;
             E : T2;
             F : Boolean;
           end record;
type T4 is array(1..10) of T3;
X : T4;
```

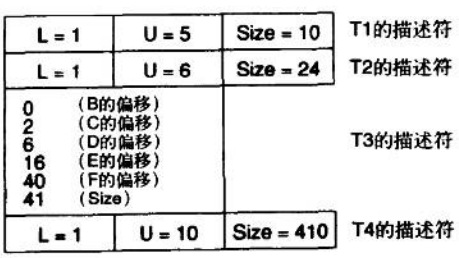
如果我们沿用前面章节里的方法，就必须在X的10个条目中的每一个中放置记录描述符来初始化X。X中每一个记录也必须初始化。如果X是动态数组，则又多了一层复杂性。那种记录和动态数组任意嵌套的一般情况通常很难理解。

相反地，我们考虑一种通用的解决方案，它使用早些时候我们在处理多维数组时所提出的运行时类型描述符。该描述符可以很好地适用于记录。在这个记录类型描述符中每个域都有单个的条目以表示该记录域到记录开始地址的偏移。它也包含存放整个记录长度的条目。如果一个域是动态的，即该记录域要么是动态数组，要么它含有一个包含某种层次的动态数组的结构，那么它后面域的偏移以及整个记录的长度在编译时就无法确定。

图11-15a 给出上述例子在块入口处尚未全部填写的运行时描述符。假定在进入块时I是5，J是6，那么这些运行时描述符的修改将如图11-15b 所示。



a) 进入块入口前的记录类型描述符



b) 初始化后的记录类型描述符

图 11-15

同先前一样，变量在活动记录中有存储位置。这些位置中保存的绝对指针指向在活动记录中固定长度部分之后为那些对象所预留的内存位置。预留内存的大小可在类型描述符的size域找到。记录赋值将整个数据块从内存的某个区域传送到另一个区域而不需要担心嵌套的指针：因为这里没有此类指针！数据块传送的长度可以在运行时从AR中的某个编译时已知的偏移位置上的合适的类型描述符中的合适的size域中找到。

一种有效实现类型描述符初始化的方法是，为它保存一个在编译时尽可能多地填写的模板。编译器创建该模板并把它放置在程序已初始化的数据区中。块的入口首先要将此模板拷贝到AR中，然后填写其中未初始化的部分。作为选择，可以只把那些编译器不知道的部分实际存放在AR中。这种做法需要小心处理，因为AR中的偏移很难追踪。 409

所有动态对象的运行时类型描述符均简化了存储管理。以下三个运行时的区域均包含了任意数据对象的信息，无论有多么复杂：类型描述符、变量空间和数据本身。类型描述符有固定的长度并在AR中已知的偏移处存放且被所有该类型的变量共享。变量空间也有固定的长度，亦在AR中已知的偏移处并指向对象的数据区。

在某个作用域中声明的类型也可以在所嵌套的作用域中使用。编译器记住每个类型的嵌套深度及其类型描述符在AR中的偏移。在运行时可以使用静态链簇或显示表来查找声明为非局部的动态类型的变量的类型描述符。

11.3.3 变体记录

编译变体记录的域比在10.3.5节中处理变体记录的声明要简单得多。事实上，只需对在11.2.3节中描述的例程field_name()稍做扩展即可处理包含在变体中的域的引用。

包含在变体中的域的引用若要合法化，那么外围标签域的值必须匹配变体的标记。为此必须产生代码以检验域引用的合法性，就像数组边界检查那样。使用在第10章为变体记录定义的复杂的type_descriptor结构可以很容易地产生那些检查代码。在attributes记录中field变体在其enclosing_variant域中包括一个指向struct variant_des的指针。

```
struct {    /* class == FIELD */
    address_range field_offset;
    struct attributes *next_field;
    struct variant_des *enclosing_variant;
    boolean is_tag;
};
```

如果这个指针为NULL，那么表明该记录域不在某个变体中且不需要做有关的检查。如果此记录域在某个变体中，struct variant_des将提供用来产生检查代码的信息：

```
struct variant_des {
    long choice_value;
    attributes *tag_field;
    attributes *field_list;
    struct variant_des *inner_variants;
    struct variant_des *next_variant;
};
```

运行时的检查将测试由tag_field引用的域的值是否等于choice_value。如果不相等，则产生一个运行时错误或异常。

由于变体可以嵌套，因此必须检查标签域attributes记录中的enclosing_variant域，以此来判别标签域本身是否包含在另一个变体中。如果是的话，必须使用标签域的外围enclosing_variant来重复相同的处理。 410

11.3.4 访问类型的引用

在Pascal语言中，指针引用显式地用符号↑来标识。在Ada和Ada/CS语言中，在访问对象的名字后
411 使用 . all作为一种<name suffix>来表示整个对象的引用。然而，被引用的对象中成员的引用不需要特别的语法标识。因此，A.D既可以表示为记录A中域D的引用，又可以表示访问对象（指针）A所指向的记录中的域D的引用。（在C语言中，A.D的意思要么是A.D，要么是A->D。）

我们首先考虑显式引用的情况：

<name> → <simple name> { <name suffix> } [. all #access_ref]

例程access_def()用来处理那些描述访问类型对象的DATAOBJECT记录。乍看起来，例程access_def()务必会生成一些少量的代码。访问对象只是个指针，因此，其地址是它所引用对象的间接地址。这样，如果在DATAOBJECT记录中地址不是间接地址，那么仅需要将indirect标志设为TRUE。如果该地址已是间接地址，则需要有关元组来执行一次间接访问。与如此计算的地址对应的类型即为访问类型所引用的类型。

不幸的是，事情不会这么简单。我们无法保证访问对象包含有效的数据对象引用。该对象可以含有空指针（在Ada中表示为**null**，在Pascal中为**nil**）。在那些对变量初始化没有语法描述以及不要求所有指针变量均要初始化的语言中（例如，Pascal和Modula-2），访问对象可能是未初始化的，因此也就包含着随机值。最后，如果允许显式的空间释放（如Pascal中的Dispose，Ada中的Unchecked_Deallocation），那么访问对象则可能指向一块已经（使用同一对象的其他引用来）释放的存储区域。在这种情况下，我们将得到一个*悬空指针*。因此，指针引用需要运行时的检查，其意义在某种程度上和数组下标表达式的边界检查一样。

如果希望仅检查指针是否为空（就像许多Ada实现版本所做的，这是因为在Ada中访问对象必须初始化且不需要检查Unchecked_Deallocation），我们可以为**null**选择一个特殊的值以便在用它来寻址对象时引起地址故障。该故障可被语言的运行时系统捕获并被转换成适当的错误信息。如果这样的故障可由硬件产生并被及时捕获，那么这类空指针的检查即可随时（免费）使用。

未初始化指针和悬空指针（包括空指针）需要更多的显式检查。UW-Pascal为此实现了一种有效的检查机制（Fisher and LeBlanc，1980）。使用该技术，访问对象由所引用的地址加上另一个称为*key*的域组成。访问对象所引用的对象也配有一个额外的称为*lock*的域。当动态对象分配时，它们均配备了惟一的lock值（模上lock域所表示的最大值）。由分配例程创建的指针在其key域中包含与之匹配的位串（见图11-16）。

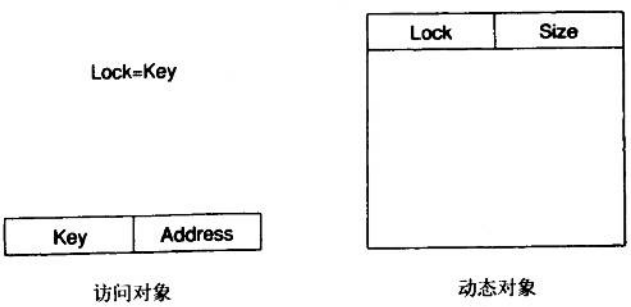


图11-16 动态对象及其指针的存储布局

指针赋值必须同时拷贝key域和address域，这样创建的新的访问对象拷贝就可以合法地访问给定
412 的动态对象了。对动态对象的访问若要合法化，它们之间的key域和lock域就必须匹配。对空指针来说，

此项检查将失败，因为任何空指针会引起地址故障。未初始化的指针可能引起地址故障，或它的地址域可能包含一个可以合法地解释为某个地址的位串。在后面一种情况下，key域和lock域的位不大可能匹配。为处理悬空指针，释放例程必须修改它所释放的任何对象的lock位串。然后，指向该对象的其余指针将由于key域和lock域的测试失败而无效。

假定指针引用涉及比改变标志indirect为TRUE更多的处理工作，那么语义例程access_ref()就必须产生元组以通知代码生成器为动态对象引用产生代码。例如，

```
(AccessRef, <name>.data_object, T)
```

其中，T是为此目的而产生的新的临时变量。

由于动态对象成员的引用缺乏显式的语法描述，因此在前面章节里讨论过的语义例程start_index()和field_name()找到的或许是由它们的某个语义记录输入参数所描述的访问类型对象，而非某个数组或记录。在这种情况下，上述每个例程可以先调用例程access_ref()，然后再完成剩余的处理工作。

11.3.5 Ada中其他名字的使用

413

在11.2.1节里，我们考虑了和下面<simple name>产生式相联系的语义例程new_name()：

```
<simple name> → <id> #new_name
```

那时，我们假定所考虑的标识符是常量或变量；这种假设意味着在语义栈上很适合用DATAOBJECT记录来表示标识符。然而，也存在着另外一些可能性，它们可被其他的语义栈条目很好地处理。标识符可以是类型名、包名、外围子程序或块名（用于限定名字引用）、函数名（用于函数调用）。如果标识符是类型名，那我们将用含有相应type_descriptor引用的TYPEREF记录来表示它。至于其他的情况，我们引入新的语义栈记录类型来简单地存放该标识符的属性引用直到该名字的使用得到进一步解析为止：

```
struct attributes_type {
   attributes *attribute_ref;
};
```

现在定义例程new_name()的扩展版本：

```
new_name(<id>) => <simple name>
{
   Find <id>.id.id in the symbol table
   Let A reference its attributes.

   if (A.class == CONST || A.class == VARIABLE)
       <simple type> ← a DATAOBJECT record with
          appropriate information extracted from the
          symbol table attributes.
   else if (A.class == TYPENAME)
       <simple type> ← (struct type_ref) {
              .object_type = A.id_type; }
   else if (A.class == PACKAGENAME
      || A.class == SUBPROGRAMNAME
      || A.class == BLOCKNAME
      || A.class == LOOPNAME)
       <simple type> ← (struct attributes_type) {
                 .attribute_ref = A; }
   else
       <simple type> ← an ERROR record
}
```

在重新定义new_name()之后，必须考虑<name suffix>的语法中可能的语义处理。首先，考虑先前讨论过的处理记录域引用的情况。

414

```
<name>                          → <simple name> { <name suffix> }
<simple name>                   → <id> #new_name
<name suffix>                   → . <selected suffix> #selected_name
<selected suffix>               → <id>
                                → <operator symbol or string>
<operator symbol or string>     → STRINGLITERAL
```

例程selected_name()的简化版本，即称为field_name()的例程曾在11.2.3节中讨论记录时出现过。现在考虑其他的选择：

```
selected_name(<name>, <selected suffix>) => <name>
{
   if (<name>.record_kind == DATAOBJECT
       && <selected suffix>.record_kind == ID)
      Process as described for field_name().
   else if (<name>.record_kind == PACKAGENAME) {
      Search the symbol table of the package for the
          identifier or operator symbol designated by
          <selected suffix> and retrieve its
          attributes.
      Process the attributes retrieved above just
          as in new_name().
   } else if (<name>.record_kind == ATTRIBUTES
       && <name>.attributes.attrs names a currently
       open scope) {
      Search for the identifier or operator symbol
          designated by <selected suffix> in
          that symbol table of the scope and
          retrieve its attributes.
      Process the attributes retrieved above just
          as in new_name().
   } else
      <name> ← an ERROR record
}
```

<name suffix>的带括号的表达式列表版本提出了更为复杂的问题。因为Ada/CS中函数调用和数组引用的形式在语法上是一样的，所以在11.2.3节、11.3.1节中提出的语义处理可能不会像它们所描述的那样简单地完成任务。如果在表达式列表前的<name>所指示的是函数而非数组，那么将调用与例程start_index()、index()和finish_index()类似的有关例程。另一种选择是在每次动作例程触发时检查<name>的语义记录，然后（为数组下标或函数参数）执行合适的动作或者收集一系列表示表达
415 式列表的DATAOBJECT记录，最后根据<name>的语义记录（再次地，区分函数名和数组引用）处理上述语义记录列表。后一种方法可能是较好的选择，因为它与函数重载所需的技术配合得很好，同时还因为在预定义的语言属性的语法形式中也使用了带括号的表达式列表。为此，我们使用以下语义动作符号的布局：

```
<name suffix>  → ( #start_expr_list <expression list> ) #name_plus_list
```

其中，<expression list>语法中的动作例程必须将DATAOBJECT记录添加到由例程start_expr_list()初始化的列表中。

例程name_plus_list()检查<name>的语义记录。如果它是数组引用，则将使用描述在11.3.1节中的技术来处理有关列表。如果是函数名，那将使用在第13章里讨论的技术来处理参数，或者为重载解析建立所需的结构，该结构将在11.3.7节中描述。每一种选择均会产生一个DATAOJBJECT语义记录。如果<name>表示的是其他东西，则产生ERROR记录。

最后剩下的一种<name suffix>形式为：

```
<name suffix>  → ' id #process_attribute
```

此语法中的标识符将命名预定义的语言属性，该属性扮演着预定义函数的角色。很多属性用来限制（或修饰）类型名，但有些也可用于数据对象。一些有参数的属性采用带括号的表达式列表的语法形式。最

好将属性作为特殊的情况加以个别处理。多数属性可以直接翻译成1个或2个元组。带有参数的属性完全可以作为编译器中函数来使用以配合name_plus_list()所做的处理，或者我们可以在该例程中建立其他的用于属性处理的措施。

最后，我们需要在<name>的产生式中添加一个动作符号：

<name> → <simple name> { <name suffix> } [. all #access_ref] #finish_name

例程finish_name()不做任何工作，除非它发现其参数是某个无参函数的属性。在这种情况下，它触发调用该函数所必需的语义处理。

11.3.6 记录和数组聚合

聚合（aggregate）是一种联合多个值组成记录或数组类型复合值的操作。在Ada/CS中，聚合的语法是： 416

```
<primary>      → <aggregate>
<aggregate>    → <name>'( <component> { , <component> } )
<component>    → [ <choice list> => ] <expression>
<choice list>  → <choice> { | <choice> }
<choice>       → <simple name>
               → <simple expression>
               → <discrete range>
               → others
```

开始聚合的<name>必须是一个记录或者约束数组类型名；这样它可以表示为TYPEREF记录。形式最简单的聚合是其中的每个成员仅由一个<expression>组成。这里仅使用按位结合。例如，如果A是一个数组类型名，其下标范围为1..5，元素类型为整型，那么A ' (1,2,3,4,5)表示一个类型A的对象，其中A(1) = 1，A(2) = 2，等等。更一般地，部分或全部的成员值可以是表达式：A' (I,I+1,I+2,4,5)。现在A(1) = I，A(2) = I+2，等等，但前3个成员的值在编译时不可用。

为编译按位聚合，我们可以简单地收集成员的值直到遇见列表结束。处理聚合的语义例程必须为类型的实例分配空间，检查是否提供了正确的成员个数和类型（对数组而言，所有成员的类型都必须相同；而记录则典型地需要不同的成员类型），然后生成使用成员的值来填写聚合的元组。编译按位聚合时的主要抉择在于决定应在何处分配聚合的空间。最一般的做法是在当前活动记录中分配空间。然而，这种做法有一个缺点：如果其中有带有常量值的成员，那么我们或者在每次创建当前作用域的活动记录时都必须初始化这些成员值，抑或我们要产生填写这些常量值的代码。另一种可供选择的方法是在我们先前称为常量区的静态数据区中为聚合分配空间。因为该数据区是静态分配的，所有只需在程序开始执行时初始化那些常量成分。此种方法的缺点是它的较大的空间需求；并且我们要为程序中的每一个聚合静态地分配空间。

当聚合的<component>被冠以前缀<choice list>时，称为*按名结合*。此时在列表中的位置不再有意义。因此，前面的第一个例子可以重写为：

```
A'(5 => 5, 4 => 4, 3 => 3, 2 => 2, 1 => 1)
```

从语法上，我们看到“选择”标签可以包括表达式、范围和关键字**others**（它仅在列表的最后出现并标注一个赋给聚合中所有剩余成员的值。）这些选择实际上仅对数组聚合有用。按位与按名结合的组合使用也是合法的，但按位结合必须先出现。 417

按名结合本质上与按位结合没有什么不同，但它更复杂一些。例如，在处理开始前必须收集更多的信息。如果从语义栈中收集列表，那么为了避免混淆成员列表在何处结束，需要对选择列表和成员列表采用不同的分隔符。除了生成元组以将指定的值赋给聚合适当的成员外，语义例程还必须检查所有的成员是否均被赋值且没有成员被多次赋值。

最后，完成的聚合可以用单个DATAOBJECT记录来表示。

11.3.7 重载解析

为允许用户定义的类型来扩展语言，可以赋予Ada中的标准运算符额外的含义来表示新类型（或现有类型的新的组合类型）。正如在11.2.2节讨论eval_unary()和eval_binary()时所提到的，这种特性称为重载，它只不过是常见子程序设计语言中的概念的外延。C++是另一个允许重载的出色的语言。

在Ada/CS中，可以重载运算符和子程序的名字。而在Ada中，亦可重载枚举文字常量。两种定义能够共享（重载）运算符或子程序名，只要它们在自变量的数目或类型，抑或是结果类型上有区别。（可将枚举文字常量看成无参函数。）至于任何含有重载名字的表达式或语句，则必须通过上下文来决定使用那些共存定义中的哪一个。如果不能做出惟一的选择，则意味着出现错误。此时程序员必须通过定义标识符或运算符的块、子程序或者包来限制它们以解决二义性（例如，Sqrt.Min(X)而非Min(X)）。我们也可以用类型限制符来保证结果类型（例如，Float'(A,B)）。

自变量的数目和类型以及结果类型恰好相同的（同名）运算符或子程序绝不能同时存在。如果其中一个定义和另外一个所在的作用域不同，那么通常的作用域规则可以发挥作用且在内层的定义将隐藏其他的定义。如果两者均在相同的作用域中定义，则会产生多重定义的错误。

在所有隐藏的情况中（包括变量、类型和常量的名字），可以使用显式的限制符来引用那些被隐藏的名字（这就是为什么块允许有名字的原因）。注意，只有运算符、子程序和枚举文字常量可被重载。所有其他的名字只能有被作用域规则确定的惟一的定义。

自底向上解析

418 在Ada语言出现前，允许重载的程序设计语言通常需要使用自变量的数目和类型（而不是结果类型）来做重载解析。这是FORTRAN、Pascal和C++中规则，即只允许对预定义的运算符和子程序进行重载，语言Algol 68也一样，只允许某些用户定义的重载。这项限制的主要优点是，重载不会干扰正常的自底向上的表达式翻译。也就是说，我们仍然可以编译表达式，翻译其中的操作数和运算符而不必关心该表达式所出现的上下文。

不幸的是，如果使用了强类型检查，那么自底向上的重载解析就略显不足了，因为类型转换必须通过使用重载机制来实现。例如，考虑A：= I + 1；，假设其中I是整型变量。如果A是浮点型变量，则赋值号右边的表达式类型也必须是浮点型（为避免类型错误），这意味着+所表达的定义的解析必须由表达式及其操作数来决定。（在C语言中，转换规则的工作方式不同。它先做整数算术加法，然后在赋值完成时将结果转换成float或double。这里重载的是赋值运算符 =，而不是加法运算符。）

自顶向下解析算法

在一个普遍采用的Ada编译模型中，语法分析器将首先建立框架式的树结构的中间表示。然后语义例程所产生的语义信息将用来装饰这棵树，最后代码生成器将遍历这棵树并产生目标代码。

以下由Cormack（1981）提出的自顶向下的递归例程采用了上述模型，它遍历表达式树并计数重载符号的可能的一致性解释。如果它返回1，则该树没有二义性；返回0，则表示没有有效的解释；若返回2个或更多，则表明树是二义的。

```
int count(type *target_type, tree_node node)
{
  int solutions = 0, parm_combos;
  if (node is a leaf) {
     if (node.type == target_type)
        return 1;
     else
```

```
        return 0;
    }
    /* node is an operator, or subprogram name */
    for (each definition, node.def, associated with node) {
       if (node.def.result_type == target_type &&
           node.def.arg_count ==
            number of subtrees of node) {
          /*
           * This definition is possible;
           * check the args, one by one.
           */
          parm_combos = 1;
          for (i = 1; i <= node.def.arg_count; i++) {
             parm_combos *=
                count(node.def.arg[i].type, node.son[i]);
          }
          solutions += parm_combos;
       }
    }
    return solutions;
}
```

419

这个算法将过程当作函数来处理并返回特殊类型Void。很容易扩充该例程以标记所发现的第一个带有有效定义的内部结点。如果返回计数为1，那么所做的标记是惟一的；否则，表达式有错误且此标记可被忽略。

自底向上解析算法

Cormack的算法简单有效。但表达式树往往是采用自底向上的方式建立或翻译的。因此我们可能希望用自底向上的算法借助于自变量的信息来解析重载并在需要时利用有关的结果信息。这样的算法已由Baker（1982）提出。他得出了两项关键性的结论：

(1) 如果子表达式没有合法的解释，那么包含它的表达式也没有。

(2) 如果子表达式有不止一种的解释，每一种解释必须在子表达式的结果类型上不同。

这些结论导致的自底向上的算法，在必要时，将建立表达式树的列表而不是单独的一棵树。每棵树有惟一的类型，用来限制必须维护的树的数量。在树出现的上下文变得明朗时，将从列表中剪除掉一些树。最后，惟一的一棵树被确定下来，或者错误被发现（无有效的或二义性的解释）。

例程build_tree()（如图11-17所示）将被渐增式地调用以建立表达式树。该例程将可能重载的运算符或子程序名以及（将被建成树的）自变量表作为它的参数。每个自变量又可以是每棵子树有惟一结果类型的子树列表。这推广了通常操作数均是惟一子树的建树例程。例程build_tree()返回树的列表，其中每棵树均有惟一的结果类型。我们假设有例程select_tree()，它以树列表和某种类型作参数，返回指针指向列表中具有该类型的（惟一的）树，或者由于不存在这样的树而返回NULL。

```
tree_list build_tree(tree_node node,
                     list_of_tree_list arg_list)
{
  tree_list result_trees = { /* empty */ };
  tree_node *P;
  /* node is an operator, or subprogram name */
  for (each definition, node.def, associated with node) {
     if (node.def.arg_count == length(arg_list)) {
        /* This definition is possible;  */
        /* process the args, one by one  */
        /* Create a new tree_node N, as follows: */
        tree_node N;
        N.def = node.def;
```

图11-17 建立表达式树的重载解析算法

```
            N.type = node.def.result_type;
            N.ambiguous = FALSE;
            /*
             * ambiguous is assigned TRUE if there is
             * more than one overload resolution that
             * returns N.type
             */
            for (i = 1; i <= node.def.arg_count; i++) {
               P = select_tree(arg_list[i],
                               node.def.arg[i].type);
               if (P == NULL) {
                  N.type = ERRORTYPE;
                  break;   /* loop */
               } else
                  N.son[i] = P;
            }
            if (N.type != ERRORTYPE) {
               tree t;
               if (select_tree(result_trees, N.type) == NULL)
                  append(result_trees, N);
               else {
                  t = select_tree(result_trees, N.type);
                  t->ambiguous = TRUE;
               }
            }
         }
      }
      for (i = length(result_trees); i >= 1; i--)
         if (result_trees[i].ambiguous)
            Remove result_trees[i] from result_trees;
      return result_trees;
   }
```

图11-17 （续）

通过调用build_tree()处理每一个子表达式，我们可以按自底向上的方式建立表达式树。当整棵树建好时，我们将调用select_tree()选择返回所期望类型的惟一的表达式树。然后释放那些在arg_list中没有被select_tree()选中的子树，这是因为我们已知这些树代表着重载符号的无效解释。如果将枚举文字常量表示为无参函数，那么build_tree()也可用于它们的重载。

420

为弄清楚build_tree()是如何工作的，可以考虑在Ada/CS中建立与表达式I+J对应的树（或树列表）的例子。参数node是运算符 +，它有以下2个与之关联的定义：

((float, float) → float)
((integer, integer) → integer)

如果我们是在编译像Pascal那样允许混合模式的表达式的语言，那么可以包括额外的float和integer参数的组合定义。参数arg_list是tree_list的列表，该列表中的每一项是操作数（tree_list）可能的解释列表。此例中的2个操作数都是简单变量，因此，每个tree_list含有单独的一棵树，且是平凡的，仅表示变量的类型：

arg_list: ((integer), (integer))

当build_tree()将node所关联的定义和arg_list做比较时，只有第二种定义匹配由arg_list提供的树。因为I+J只有一种可能的解释，所以例程build_tree()将返回仅包含一个元素的tree_list。

另一方面，如果有其他与+关联的定义，例如：

((integer, integer) → float)

那么它也将匹配arg_list上的树，此时例程build_tree()将返回包含2个选择的tree_list。这两棵树反映出在那个语言中允许符号+用于两个整数相加并产生一个整数或浮点数的事实。我们最终将

根据使用表达式的上下文来选择其中的一棵树。

练习

1. 与数据对象的符号表引用表示法相比，其面向地址的表示法的相对优点与缺点是什么？ 421 ~ 422
2. 编写代码实现11.2.1中所描述的new_nane()例程，说明如何利用符号表中的信息创建表示标识符的DATAOBJECT记录。
3. 描述由11.2.2节中的语义例程使用的select_unary_operator()和select_binary_operator()。
4. 在一个利用free_temporary()进行显式临时变量释放的编译器中，11.2.2节里的例程eval_unary()和eval_binary()应如何调用此例程？
5. 针对以下声明，为下面的每条Ada/CS语句产生相应的元组：

```
I, J, K : Integer;
X, Y, Z : Float;
```

(a) I := -(J + 5);
(b) J := I * 2 - J * 3 + K / 4;
(c) X := Y * 3.5 + Z / Float(I);

6. 给出以下记录声明中各记录域的偏移：

```
R : record
    X : Float;
    I, J : Integer;
    A : array (1..10, 5..9) of Float;
    R : record
        S : String;
        L : Integer;
      end record;
    S, T : String;
  end record;
```

7. 给出在处理图11-5b中的每条语句时所产生的动作例程的调用序列，并指出每一例程所生成的元组，如果有的话。
8. 为下面的每条语句产生相应的元组，使用图11-5a中的声明：
   (a) A1(5) := A1(I+J);
   (b) A2(ARecord.X, ARecord.Y) := J;
   (c) A1(A1(I).Y).X := J* 3;
   (d) A2(I,9) := A2(5,J);
9. 使用在11.2.4节中讨论的动态串的实现方法，描述实现串比较、子串存取和串连接所需的运行时例程。 423
10. 解释如何修改在图11-7中为访问A(I,J,K)所生成的元组，如果所有关于A的信息是从一个像图11-8中那样的内情向量中访问获得的（而不是在编译时可用的）。
11. 如果使用像图11-11中那样的运行时类型描述符实现数组，那么，
    (a) 给出为下面的数组声明生成的类型描述符：

```
type T is array (1..M) of array (2..N) of Float;
```

    (b) 如果A和B在声明T的同一个过程中声明为类型T，试给出那个过程的活动记录，并说明其中所有与A、B和T相关的对象。
    (c) 给出在以M = 5和N = 8进入此过程后，该类型描述符的示意图。
    (d) 为下面的语句产生元组：

```
A(I,J) := B(1,I) + B(J,2);
```

12. 描述为A(I,J,K)所生成的元组，如果A是采用图11-12中说明的向量化组织来存储的。
13. 采用定义在图11-15中的类型描述符，为表达式X(I).C+X(I).E(J)生成相应的元组。
14. 给定以下声明并假定I的偏移是5:

```
I : Integer;
R : record
      F1 : Integer;
      case T1 : Integer range 1..2 is
          when 1 => F2 : Integer;
          when 2 => case T2 : Integer range 3..4 is
                        when 3 => F3 : Integer;
                        when 4 => F4 : Integer;
                                  F5 : Integer;
                    end case;
    end case;
  end record;
```

那么，将为以下语句产生什么样的元组（包括变体检查）？

(a) I := R.F1;

(b) R.F1 := R.F2;

(c) R.F4 := I + R.F5;

424 15. 在Pascal程序中声明的变体记录，如果它覆盖了两个指向不同类型、不同大小的对象的指针，则它可以使我们在11.3.4节中介绍的检验动态对象指针有效性的技术失效。例如:

```
R : record
  case T1 : boolean of
    True: (P1 : ↑ array [1..10] Integer;);
    False: (P2 : ↑ array [1..100] Integer;);
  end;
```

因为可以在不改变记录中其他信息的情况下更改标签域（T1），所以在调用new(P1)以及将T1赋值为False之后就会存在某种不安全的局面。对R.P2↑ [50]的引用可以通过变体的检查、指针有效性检查以及下标检查，但它所引用的堆中位置并不是为创建该指针而调用new所分配的。解释如何扩展指针检查机制以便在这种情况下指示错误。

16. 假定运算符+有以下与之关联的定义:

```
((float, float) → float)
((integer, integer) → integer)
((integer, integer) → float)
```

425 如果I和J是整数且F是浮点数，说明如何使用图11-17中的例程build_tree()来解释表达式 I+J+F。

第12章 翻译控制结构

在本章中，我们研究那些能确定控制流的Ada/CS中相关语言特性的实现技术。这些控制结构一般可以代表出现在现代程序设计语言中的各种控制结构。我们要考虑的语句类型可分为三类：循环结构、条件执行结构和直接控制转移。循环结构包括可使用**exit**语句在任意点退出的简单**loop**语句以及通过**for**和**while**子句控制的**loop**语句的特殊实例。条件执行结构包括普遍使用的**if-then-else**语句和**case**语句。最后，直接控制转移包括上面提到的**exit**语句和异常。我们还要讨论在许多语言中都有的**goto**语句的编译，尽管在Ada/CS中没有该语句。 426

12.1 if 语句

Ada/CS包括普遍使用的**if**语句，它带有可选的**elsif**和**else**部分。这两个可选部分的组合产生以下两种一般的形式：

```
if <boolean expr 1> then
  <stmt list 1>
elsif <boolean expr 2> then
  <stmt list 2>
  ...
elsif <boolean expr N> then
  <stmt list N>
end if;
```

和

```
if <boolean expr 1> then
  <stmt list 1>
elsif <boolean expr 2> then
  <stmt list 2>
  ...
elsif <boolean expr N> then
  <stmt list N>
else <stmt list N+1>
end if;
```

如果没有**elsif**，它们将简化为在包括Pascal的大多数语言中都可找到的基本的**if**语句。Ada/CS中的**if**语句同Pascal语言中的**if**语句的重大区别是闭合关键字**end if**的使用，这不仅提高了**if**语句的可读性，同时也允许在**if**语句的每个选择分支中可以使用语句列表而不只是单个的语句。

为**if**语句产生的元组格式如图12-1所示。图12-1中给出了两种形式，分别对应带有**else**子句和不带**else**子句的**if**语句。和以往一样，元组被表示成括在括号内的一组值。散布在图中的元组描述了那些潜在的较长元组序列。这些描述放在大括号中，如{...}。在此例和本章其他示例中，某些元组前被冠以标号。在第7章定义的中间语言中，不允许元组拥有标号；相反，它提供了特殊的标号元组。尽管本章中所概述的语义例程可以正确地在相应的地方生成标号元组，但在这个例子中前缀标号的使用只是为了增加可读性。

目前，我们沿用处理表达式的一般惯例。在分析<boolean expr>时调用语义例程生成相关代码来计算布尔表达式的值为True或False，并且产生可描述结果的DATAOBJECT语义记录。在12.7节中，我们将研究另一种处理布尔表达式的方法，它生成合适的控制转移而不是显式的布尔值。 427

```
                    {Evaluate <boolean expr 1>}
                   (JUMP0, <boolean expr 1>, Else1)
                     { code for <stmt list 1> }
                   (JUMP, Out)

Else1:             {Evaluate <boolean expr 2>}
                   (JUMP0, <boolean expr 2>, Else2)
                     { code for <stmt list 2> }
                   (JUMP, Out)

                   ...

ElseN-1:           {Evaluate <boolean expr N>}
                   (JUMP0, <boolean expr N>, Out)
                     { code for <stmt list N> }
Out:               ...
```

a) 不带**else**部分的**if**语句的元组

```
                    {Evaluate <boolean expr 1>}
                   (JUMP0, <boolean expr 1>, Else1)
                     { code for <stmt list 1> }
                   (JUMP, Out)

Else1:             {Evaluate <boolean expr 2>}
                   (JUMP0, <boolean expr 2>, Else2)
                     { code for <stmt list 2> }
                   (JUMP, Out)

                   ...

ElseN-1:           {Evaluate <boolean expr N>}
                   (JUMP0, <boolean expr N>, ElseN)
                     { code for <stmt list N> }
                   (JUMP, Out)

ElseN:               { code for <stmt list N+1> }
Out:               ...
```

b) 带有**else**部分的**if**语句的元组

图12-1 带有和不带有**else**部分的**if**语句的元组

428 我们需要在一些地方产生代码和进行语义处理：

- 在每个布尔表达式后，需要根据表达式的值生成一个条件跳转。
- 在**then**部分后，如果有相应的**else**或**elsif**部分，则需要生成跳转语句跳过它们。
- 标识**else**或**elsif**部分的正确标号以及**if**语句的结尾必须由构造JUMP元组的动作例程产生。相应的LABEL元组必须放在元组序列中正确的位置上。

这些考虑导致以下的语法和动作符号的摆放：

```
<if statement>  → if #start_if <b expr> #if_test then <stmt list>
                    { elsif #gen_jump #gen_else_label <b expr> #if_test
                    then <stmts> }
                    <else part> end if ; #gen_out_label
<else part>     → else #gen_jump #gen_else_label <stmt list>
<else part>     → #gen_else_label
```

上述产生式中指定的所有动作例程都生成JUMP或LABEL元组。它们用来相互通信的新的语义记录保存着标号，时间是从这些标号被创建且用于JUMP元组一直到某个相应的LABEL元组被产生为止。假定存在某个名为`new_label()`的支持例程，它在每次被调用时创建一个惟一的标号。标号可以被表示成字符串，这个新的语义记录类型是：

```
struct if_stmt {
    string out_label, next_else_label;
};
```

动作例程start_if()构造IFSTMT语义记录并调用new_label()来创建一个可作为所有跳出**if**语句的跳转语句的目标标号。(这些跳转语句发生在**then**部分结尾。)这个新标号存放在IFSTMT记录的out_label域中。next_else_label域被初始化为空串，因为所有**else**标号在例程if_test()中创建，该例程生成一个跳过其后**then**部分的条件跳转语句:

```
start_if(void) => if
{
    if ← (if_stmt) {
            .out_label = new_label();
            .next_else_label = " "; }
}
if_test(if, <b expr>) => if
{
    Check that <b expr>.data_object.object_type == BOOLEAN
    if.if_stmt.next_else_label = new_label()
    >generate(JUMP0, <b expr>.data_object,
            if.if_stmt.next_else_label, "");
    if ← the updated IFSTMT record
}
```

429

如果有**else**或**elsif**部分跟在**then**部分后，gen_jump()例程被调用来生成跳至out_label的跳转:

```
gen_jump(if)
{
    generate(JUMP, if.if_stmt.out_label, "", "");
}
```

gen_else_label()用if_test()生成的标号来标记**else**或**elsif**部分的开始，作为前面条件跳转的目标:

```
gen_else_label(if)
{
    generate(LABEL, if.if_stmt.next_else_label, "", "");
}
```

在处理完全部语句后，gen_out_label()为start_if()创建的out_label生成LABEL元组:

```
gen_out_label(if)
{
    generate(LABEL, if.if_stmt.out_label, "", "");
}
```

我们建议读者从这些例程跟踪一些**if**语句例子(从简单的到复杂的)，来验证生成的代码和本节开头的例子是否匹配。尤其重要的是，要观察在处理**if**语句过程中**if**.if_stmt.next_else_label的值是如何使用的。

在本节中，首次研究了可能嵌套其他语句的语句。**if**语句和其他所有诸如此类的语句，它们的语义记录完全独立于它们包含的语句，因此，其他控制结构在这些语句列表中的出现不会带来任何问题。

最后，考虑在直接生成二进制代码的一遍编译器中**if**语句的处理。在这样的编译器中，刚才提出的技术不能工作，因为它们依赖下一遍编译处理去解析符号化的标号引用。待解决的基本问题是当JUMP和JUMP0元组在gen_jump()和if_test()中生成的时候，它们的目标位置尚不知晓。与当元组的目标明确时保存待生成的符号化标号相反，对目标不明确的元组的某些引用(例如序列号)必须保存在IFSTMT记录中。在发现合适的目标元组序号时，将其值作为操作数填入那些跳转元组。这种处理过程称为回填(backpatching)。

430

为在本节中提出的语义例程框架内实现回填，必须稍微改动IFSTMT记录。作为out_label域的替换，该记录中必须包含所有应在语句结尾处回填的JUMP元组的列表。类似地，next_else_label域

被last_else_jump域所取代，后者保存着上一次生成的JUMP0元组的序列号。gen_else_label()动作例程在其被调用时必须回填那个元组以跳至下一个可用元组号。在12.7节中，我们将非常详细地讨论回填。

12.2 循环

循环语句的翻译相当简单。在关键字**loop**后添加动作符号#gen_loop_label以便在循环开始的地方生成一个LABEL元组并将其保存在语义记录中。这个标号在循环结尾被语义例程loop_back()用来生成一个回到循环上部的跳转。

涉及的产生式为：

<basic loop> → **loop** #gen_loop_label <stmts> **end loop** #loop_back

需要一个新的用来保存单个循环标号的语义记录：

```
struct label {
    string label;
};
```

所涉及的两个语义例程相当简单：

```
gen_loop_label(void) => loop
{
    L = new_label();
    generate(LABEL, L, "", "");
    loop ← (label) { .label = L; }
}

loop_back(loop)
{
    generate(JUMP, loop.label.label, "", "");
}
```

431

因此，为如下简单循环生成的元组

```
loop
  <statement list>
end loop;
```

具有以下形式：

```
(LABEL, LoopStart)
  { code for <statement list> }
(JUMP, LoopStart)
```

12.2.1 while 循环

刚才讨论的一般化的**loop**语句从不终止。需要用**exit**语句来结束循环并将控制转移到跟在**loop**语句后的下一条语句。**while loop**是**loop**语句的语法扩展，它处理给定的特殊情况：在循环开始的地方测试退出条件。

while语句的语法描述是：

<while statement> → **while** #start_while <b expr> #while_test
loop <stmts> **end loop** #finish_while

需要一个新的语义记录，它非常像IFSTMT记录：

```
struct while_stmt {
    string top_label, out_label;
};
```

start_while()和while_test()很像我们已经看过的gen_loop_label()和if_test()动作例程:

```
start_while(void) => while
{
    L = new_label()
    generate(LABEL, L, "", "");
    while ← (while_stmt) {
            .top_label = L;
            .out_label = ""; }
}
while_test(while, <b expr>) => while
{
    Check that <b expr>.data_object.object_type == BOOLEAN
    while.while_stmt.out_label = new_label();
    generate(JUMP0, <b expr>.data_object,
            while.while_stmt.out_label, "");
    while ← the updated struct while_stmt
}
```

432

finish_while()看上去也很熟悉，可视它为在**if**语句结尾和简单循环结尾时调用的例程的组合:

```
finish_while(while)
{
    generate(JUMP, while.while_stmt.top_label, "", "");
    generate(LABEL, while.while_stmt.out_label, "", "");
}
```

因此，为如下**while loop**生成的元组

```
while <boolean expr> loop
  <statement list>
end loop;
```

具有以下的形式:

```
(LABEL, LoopStart)
  { code for <boolean expr> }
(JUMP0, <boolean expr>, Out)
  { code for <statement list> }
(JUMP, LoopStart)
(LABEL, Out)
```

12.2.2 for 循环

loop语句的另一个特别扩展是**for loop**，用于计数器控制的语句列表的重复。Ada和Ada/CS包含两种格式的**for loop**:

(1) "向上计数"循环，其中索引变量以升序取某个范围内的所有值:

```
for <identifier> in <range> loop
    <statement list>
end loop;
```

433

(2) "向下计数"循环，其中索引变量以降序取某个范围内的所有值:

```
for <identifier> in reverse <range> loop
    <statement list>
end loop;
```

for loop的编译很复杂，因为必须要正确地处理很多细节。特别是:

- 当进入**for loop**时，必须为循环索引创建新的作用域和数据对象。编译器必须关注循环索引创建以及可用的准确时机。例如，以下循环首部的含义依赖语言定义的细节:

  ```
  for K in K .. 10 loop ···
  ```

  它说明这个翻译问题不是惟一针对**for loop**的。类似的问题出现在变量声明中:

```
T : T;                    -- Assume a nonlocal type T exists
Int : Integer := 2*Int;   -- Assume a nonlocal integer Int exists
```

Ada和Ada/CS 通过将声明划分为两步来解决这个问题：一遇到一个标识符，这个标识符的新声明就隐藏（hide）相同标识符的非局部声明。然而，一个新声明仅在该声明完成后可用。因此，这里所有的三个例子都是非法的，因为非局部的标识符已被隐藏起来而局部的再声明还未完成。

- 必须将**in**或**in reverse**范围内的界限值存于临时变量中，以便在循环中始终可以使用这个在循环入口计算的值。因此，以下循环将执行10次：

```
L := 10;
for LoopVar in 1 .. L loop
   L := 3;
end loop;
```

- 循环可以迭代零次；必须生成处理这种情况的合适的代码。
- 必须将循环索引作为只读值加以保护。

在以下**for loop**的语义例程中，我们将循环索引放在临时变量中，原因有两条。首先，该方法要求
434 仅在循环范围内分配空间；动作例程不需要决定将循环索引看作变量并在活动记录中为它分配一个新的偏移。这一步是一种代码生成的决策。其次，被分配的临时变量经常是一个寄存器，这不仅节省活动记录空间，而且可能改进循环中代码的质量。

我们为两种**for loop**生成如图12-2所示的元组序列。这些代码序列可能看起来有些令人费解，因为它们中的每一个都包括两个不同的终止测试。然而，如果迭代的上界是给定机器上所能表示的最大整数，则这种代码结构是必要的，因为它确保了向上计数的**for loop**正确终止。（类似的问题存在于下界和向下计数循环。）一个更显而易见的代码序列将会在循环底部增值索引变量，然后跳转回循环上部以测试新值是否大于上界。该方法的缺点是：如果上界是最大整数的话，这个增值可能在最后的测试前引起上溢。在像Pascal这样的语言中，循环索引是一个从循环外可见的变量，此时，我们的方法另有目的。当索引变量是子界类型并且循环迭代范围是整个子界范围时，图12-2中的代码序列能确保该变量绝不被赋值为超出范围的值。最后，全局优化器能够在紧挨着标号Next前面的位置放置循环不变代码（见16.3.1节）而将它移出循环体，并保证除非循环至少执行一次，否则该代码将永不运行。

for语句带有动作符号的产生式如下：

```
<for statement>   → for <id> #enter_for_id in <reverse option> <discrete range>
                    #init_loop loop <stmts> end loop; #finish_loop
<reverse option> → #set_in
<reverse option> → reverse #set_reverse
```

这些产生式中指定的动作例程需要两个新的语义记录：

```
struct reverse {
    boolean reverse_flag;
};
```

和

```
struct for_stmt {
    data_object id;
    data_object limit_val;
    string next_label, out_label;
    boolean reverse_flag;
};
```

435 回想第10章中的<discrete range>的产生式，它产生一个实现为TYPEREF语义记录的子类型引用。

前三个语义例程很简单。enter_for_id()开始循环索引标识符的声明，如先前所讨论的，它指出在该声明完成前，该变量在表达式中不可用。set_in()和set_reverse()产生一个REVERSE记录，这个记录能指示循环是向上还是向下计数。

```
enter_for_id(<id>)
{
    Open a new name scope
    Enter the identifier in the symbol table in the
        new scope with attributes indicating
        that it is "unavailable"
}
set_in(void) => <reverse option>
{
    <reverse option> ← (reverse) {
                .reverse_flag = FALSE; }
}

set_reverse(void) => <reverse option>
{
    <reverse option> ← (reverse) {
                .reverse_flag = TRUE; }
}
```

436

```
                    { compute LowerBound }
                    { compute UpperBound }
                (GT, LowerBound, UpperBound, t1)
                (JUMP1, t1, Out)
                (ASSIGN, LowerBound, Index)
                (ASSIGN, UpperBound, Limit)
Next:           { code for <statement list> }
                (EQ, Index, Limit, t2)
                (JUMP1, t2, Out)
                (ADDI, Index, 1, Index)
                (JUMP, Next)
Out:            ...
```

a) 向上计数**loop**语句的元组序列

```
                    { compute LowerBound }
                    { compute UpperBound }
                (GT, LowerBound, UpperBound, t1)
                (JUMP1, t1, Out)
                (ASSIGN, LowerBound, Limit)
                (ASSIGN, UpperBound, Index)
Next:           { code for <statement list> }
                (EQ, Index, Limit, t2)
                (JUMP1, t2, Out)
                (SUBI, Index, 1, Index)
                (JUMP, Next)
Out:            ...
```

b) 向下计数**loop**语句的元组序列

图12-2 向上计数**loop**语句的元组序列以及向下计数**loop**语句的元组序列

init_loop()生成直到标号Next前的所有元组。它必须注意那些列出的细节，这包括为循环索引和界限（如果需要的话）分配临时变量。init_loop()构造一个FORSTMT语义记录以提供finish_loop()所需的信息，以便它在循环结尾处生成必要的代码。以下例程略述表明，这两个例程完成的某些处理依赖于循环的方向。

```
init_loop(<id>, <reverse option>, <discrete range>) => for
{
    data_object upper, lower, init, limit;
    struct address T;
    Change the attributes of <id> in the symbol table to
       make it "available"
    Let loop_info be a FORSTMT semantic record
    Allocate a temporary for use as the loop index, and
       create a data_object record, loop_info.id, for it.
```

```
    loop_info.id.object_type =
        <discrete range>.type_ref.object_type
    loop_info.id.addr.read_only = TRUE;
    Create data_object records upper and lower describing
       the upper and lower bounds of the index range,
       based on the struct constraint_des:
          <discrete range>.type_ref.object_type.constraint
    T = get_temporary()
    loop_info.out_label = new_label();
    generate(GT, lower, upper, T);
    generate(JUMP1, T, loop_info.out_label, "");
    loop_info.reverse_flag =
        <reverse option>.reverse.reverse_flag
    if (loop_info.reverse_flag) {
        init = upper
        limit = lower
    } else {
        init = lower
        limit = upper
    }
    generate(ASSIGN, loop_info.id, INTEGERSIZE, init);
    if (limit does not describe a static value) {
        Allocate a temporary to hold the loop limit,
            and create a data_object record,
            loop_info.limit_val, for it
        generate(ASSIGN, loop_info.limit_val, INTEGERSIZE,
                 limit);
    } else
        loop_info.limit_val = limit
    Update the attributes of <id> in the symbol table
       to be consistent with loop_info.id
    loop_info.next_label = new_label()
    generate(LABEL, loop_info.next_label, "", "");
    for ← loop_info
}
```

437

finish_loop()生成的元组包括终止测试、索引变量增值以及跳回循环体代码上部。该例程必须同时关闭（close）为循环索引创建的新的符号表作用域。

```
finish_loop(for)
{
   struct address T;
   T = get_temporary()
   generate(EQ, for.for_stmt.id, for.for_stmt.limit_val, T);
   generate(JUMP1, T, for.for_stmt.out_label, "");
   if (for.for_stmt.reverse_flag)
      generate(SUBI, for.for_stmt.id, 1, for.for_stmt.id);
   else
      generate(ADDI, for.for_stmt.id, 1, for.for_stmt.id);
   generate(JUMP, for.for_stmt.next_label, "", "");
   generate(LABEL, for.for_stmt.out_label, "", "");
   The temporaries for the loop index and limit value
       can be freed now (if the action routines
       explicitly free such temporaries)
   Terminate the current scope, discarding the symbol
       table entry for the loop index
}
```

for loop 优化

438

通常可以不需要分析控制流而对**for loop**进行优化。例如，循环索引变量可保存在寄存器中，这使得引用这些（经常在循环中出现的）变量非常快。这种技术常常工作得很好，但在某些情况下，它却使循环的执行低效或者增加了编译器的复杂性。

因为期望在穿越过程调用时仍能保留其值的寄存器必须作为调用的一部分而被保存和恢复，所以将索引变量放在寄存器中要求在循环里的每一过程都有这样的保存和恢复。因此，为特定循环将索引变量保持在寄存器中的代价依赖于循环中对该变量的引用次数、循环中过程调用的次数、保存寄存器引用的

耗费以及寄存器保存和恢复的耗费。

在允许从循环体外部引用循环索引的语言中，更会涉及这种情况。当从**for loop**中退出（即使非正常退出）时，循环索引值必须被保存到与索引对应的内存单元中，以保证退出后用到正确的值。在循环里，当一个过程被调用时，该值也必须被保存到那个单元中，以便在循环外的相应的过程体能够引用当前索引值。例如，在Pascal语言中，下面代码是合法的：

```
procedure P;
begin
  writeln (I)
end;
  ...
for I in 1 to 10 do
  P;
```

编译器可以满足这些明显的需求，方法是：如果循环索引被存放在寄存器中，则编译器在每次迭代开始的时候把循环索引的值存储到相应的内存单元中。尽管这些较复杂，但实际的保存使得循环索引在局部引用时可用，且**for loop**中这种引用的频率使得分配循环索引到寄存器中成为最常见的优化之一。

在UW-Pascal编译器中，循环索引寄存器不作为过程调用的一部分而被保存到或从相应存储位置（此例中I的位置）中恢复。相反地，在穿越调用时，寄存器的内容将被保留在一个临时存储器中。这项策略在循环仍活跃时对改变循环变量值的（非法）企图有着有趣的分支效应（ramification）。考虑下面非法的Pascal程序：

```
program Prog ( input , Output ) ;
var
  I : Integer ;
procedure P ;
  begin
    I := 0 ;
    Writeln ( I ) ;
  end ; { P }
begin
  for I := 1 to 10 do
  begin
    P ;
    Writeln ( I ) ;
  end ;
end .
```

439

编译器应当给出错误信息，但编译时很难察觉此类错误。刚才大致叙述的策略导致此程序打印：0 1 0 2 0 3 . . . 。由过程P引起的I的非法修改在过程P返回后被忽略，因为存放在循环寄存器中的I值被用作循环索引的值。尽管对循环索引的非法修改未被发现，但它们在返回循环体后即被抹掉。这种结果是在寄存器中保持循环索引带来的边缘效益。

在Ada语言中，循环索引被视为局部于循环体的有名常量（即，它们不可修改），因此，也就和前面程序没有关联。所以说，将循环索引指派到寄存器既简单又有效。

另一种优化是可能的，因为在Pascal、Ada和Ada/CS中，**for loop**的定义使得循环索引的范围完全受制于出现在循环首部中的初始和最终的循环界值。如果这些界值是常量或静态约束变量，那么循环索引可被视为循环体中的一个静态约束变量。此观察允许我们去除许多与循环索引相关联的子界或下标检查。

例如，给定以下程序：

```
A : array(1..10) of range 1..10;
  ...
for I in 1 .. 10 loop
   A(I) := I;
end loop;
```

在循环体中不需要子界或下标检查。这种优化很容易实现并可以极大提高**loop**语句代码的质量。

12.3 编译exit语句

exit语句是一种跳出循环的结构化方式。有两种形式的退出必须处理：无条件的退出和有条件的退出。**exit**语句的设计减少了前向引用问题，因为在**exit**中被引用的标号总是在它们被使用之前定义。

440 然而，**exit**所暗示的跳转目标仍然是一个未知的地方。我们需要考虑两个问题：

- **exit**可以引用显式的循环名，也可以隐式地引用包含它的最内层的循环。
- 由**exit**显式或者隐式引用的循环必须在包围**exit**的最小的外围包或子程序中。

在编译过程中的任一点，我们有一系列嵌套的、开放的（即，没有完全翻译的）循环。为处理这个翻译问题，我们为每一个循环创建一个编译时循环描述符（loop descriptor）。这个描述符包括一个指针指向直接外转/外层循环的描述符。可以被不含循环名的**exit**语句隐式引用的最内层循环将由一个称为current_loop的变量指向，该变量初始为NULL。因此，current_loop和循环描述符的链接表定义与所有开放循环相对应的记录栈。这个栈可以被单独维护，也可以被嵌入到语义栈中。本节中动作例程的概括叙述足够全面可以处理每一种选择。

循环描述符具有如图12-3所示的格式。在描述符内，label_entry指向循环标号的符号表条目；对没有标号的循环，则它为NULL。exit_label是用作**exit**跳转目标的符号化标号。containing_loop引用我们刚讨论过的直接外围循环（如果有的话）。contain- ing_proc和containing_package引用直接外围过程和包的attributes描述符。以上这些是用来检查**exit**的有效性，确信它没有指定跳出过程或包的跳转。

label_entry
exit_label
containing_loop
containing_proc
containing_package

图12-3 翻译**exit**所需的循环描述符

产生**exit**语句并带有语义动作的产生式如下：

```
<exit statement>   → exit <name option> <when option> ;
<name option>      → <name> #process_name
<name option>      → #null_name
<when option>      → when <b expr> #exit_cond
<when option>      → #exit_jump
```

441 处理**exit**所需的语义记录是：

```
typedef struct loop_des {
    id_entry *label_entry;
    string exit_label;
    struct loop_des *containing_loop;
    attributes *containing_proc, *containing_package;
} loop_descriptor;

struct loop_ref {
    loop_descriptor *descriptor_ref;
};
```

为了处理**exit**语句，必须在循环分析开始时执行的语义例程中做如下添加。必须创建LOOPDESCRIPTOR记录，并在实现循环的语义例程使用它们前将它们压入到单独的loop_descriptor栈或语义栈上。设置exit_label为新生成的标号以提供循环中任意**exit**的跳转目标。而containing_loop、containing_proc和containing_package将被分别赋值为变量current_loop、current_proc和current_package的值。current_loop接着被赋值为一个指针，指向这个新创建的描述符。如果正处理一个带标号的循环，则将为该标号创建一个符号表条目。与这个标号对应的ATTRIBUTES记录含有指向相应描述符的指针。描述符记录中的label_entry域被置为id_entry，它是为这个标号调用enter()的返回值。如果循环是未标号的，则label_entry被置为NULL。

由于使用符号化标号作为跳转目标在一遍编译器中是不可能的，因此就需要某些回填跳转地址的技术。所使用的任何技术必须处理循环中任意数量的**exit**，而不只是单个的**exit**。可以使用以下三种方法来解析跳转目标的地址：

(1) 间接跳转（Indirect Jump）：在遇到**exit**时，从常量区中分配一个位置，存储它的地址到描述符记录的exit_address域（用来替代exit_label）中，并生成到这个位置的跳转。对有条件的**exit**语句，生成的是条件跳转。当循环分析结束时，在常量区中那个分配的位置上生成到已知的目标地址的跳转。因此，**exit**语句执行了间接跳转（跳转到一个跳转指令）。

(2) 链式引用（Chaining Reference）：此外，还可以为每个**exit**语句生成形如（JUMP,Target?）的跳转，其中？表示未知的地址。将它们在一个链接表上链接在一起，在这种情况下，链接表头存放在描述符记录中的第二个域，我们称其为exit_list。这个链表中包含所有Target?待填写的跳转元组地址。一旦编译器到达循环结尾且目标地址明确，就将遍历该链表并填写Target?。

(3) 直接-间接（Direct-indirect）：间接跳转方法的一种变形是：在和第一个遇到的**exit**语句对应的位置上（不是在常量区中）生成跳转指令。然后所有后继**exit**都跳至这第一个位置。这个技巧在常量区中节省了一个字空间，并保证被处理的第一个（且可能是惟一一个）**exit**是直接跳转而非间接跳转。

第二种选择（链式引用）需要更精巧的链接式数据结构，但相比其他技术，它生成执行稍快的代码。这种选择和第三种方法都比间接跳转（第一种选择）少需要一条指令。若编译器的简单性是最重要的，那我们或许可以使用直接-间接方法，因为在最常见的情况下，它生成的代码的性能与链式引用方法相当，而同时它却使用着简单的数据结构。而且，一个能够消除跳转链的优化器或许会将间接跳转变为直接跳转。在以下语义例程的概述中，我们将讨论用作一遍目标地址解决方案的符号化标号和链式引用。其他一遍方案的实现很简单，读者可以很容易地开发它们。

process_name()和null_name()语义例程找到针对特定循环的LOOPDESCRIPTOR记录，检查**exit**语句的合法性，然后产生LOOPREF语义记录以提供对那个描述符记录的直接访问。这个struct loop_ref是例程exit_jump()或exit_cond()的输入参数，这些例程被调用来完成**exit**语句的翻译。

```
process_name(<name>) => <name option>
{
   <name> should be represented by an ATTRIBUTES
      semantic record
   If it is not so represented or if it is not a loop
      name or if it is a loop name with a null
      descriptor pointer, the exit is illegal.
   Otherwise, use the descriptor pointer to access the
      corresponding LOOPDESCRIPTOR record.
   Check that the containing_proc and containing_package
      fields in this record match the values of
      current_proc and current_package
   If either does not match, the exit is illegal,
      because the exit would cause a jump out
      of a subprogram or package.
   If the exit is legal,
      <name option> ← (loop_ref) {
         .descriptor_ref = pointer to the
            descriptor record for the
            loop being exited; }
   If the exit is not legal,
      <name option> ← an ERROR record
}
null_name(void) => <name option>
{
   Examine the LOOPDESCRIPTOR record referenced by
      current_loop
   If it is NULL or if the containing_proc and
      containing_package fields in this record do not
      match the values of current_proc and
```

```
        current_package, then the exit is illegal and
        <name option> ← an ERROR record
    If the exit is legal,
        <name option> ← (loop_ref) {
            .descriptor_ref = current_loop; }
}

exit_jump(<name option>)
{
    If symbolic exit labels are being used,
        Let L be the label found in the LOOPDESCRIPTOR
        record referenced by
        <name option>.loop_ref.descriptor_ref
          generate(JUMP, L, "", "")
    If one-pass target resolution is being used,
        generate(JUMP, Target?, "", "")
        Chain the address of this JUMP tuple onto the
          exit_list of the descriptor record referenced
          by <name option>.loop_ref.descriptor_ref
}

exit_cond(<name option>, <b expr>)
{
    Generate a JUMP1 tuple to exit the loop if the
        boolean expression is True
    The address of the jump target is handled exactly
        like that in the exit_jump() routine
}
```

当循环分析结束时，在loop_descriptor栈或语义栈中，在任何其他用来实现循环的语义记录的下面，我们找到相应的LOOPDESCRIPTOR记录。如果正在使用一遍目标解决方案，则所有通过exit_list域链接的元组地址将被填补上到next_tuple_number的跳转。如果使用了符号化标号，则此时必须生成一个用于循环的包含exit_label的LABEL元组。如果label_entry不是NULL，则我们用它来引用循环的符号表条目，并置描述符记录指针为NULL。这一步是必不可少的，因为循环名字的作用域就是包含它的块、子程序或包。尽管该名字在包含它的单元中随处可见，但使用循环名字的**exit**语句不可以在循环体外出现。接着，从栈中弹出LOOPDESCRIPTOR记录，因为该循环现在已全部编译完毕。

444

12.4 case语句

Ada和Ada/CS中有两种形式的**case**语句:

```
case <expr> is
   when <choice> | ··· | <choice> => <stmts> ;
   ···
   when <choice> | ··· | <choice> => <stmts> ;
end case;
```

和

```
case <expr> is
   when <choice> | ··· | <choice> => <stmts> ;
   ···
   when <choice> | ··· | <choice> => <stmts> ;
   when others => <stmts> ;
end case;
```

它们的区别仅在于是否包括**others**选择。

每个<choice>可以是常量表达式、上下界均为常量表达式的范围对或者其约束为常量表达式的子类型名。

Ada和Ada/CS要求**case**索引的所有可能值可被某个**when**子句的选择所包含。如果**others**是最后的

选择，它将无疑满足条件。如果**others**未使用，那么估算可能的**case**索引值范围也许比较困难。对于一个为简单变量的索引来说，其约束界限（如果它们是常量）可以从该变量的类型信息中获取。否则，将使用其底层的基类型。对于一个为表达式的**case**索引来说，即使表达式的操作数是约束的，也将难以抽取其界限。在这种情况下，多数编译器假设使用基类型的全范围值。Ada和Ada/CS允许表达式是限制的（qualified），使用符号 TypeOrSubTypeName' (<expr>)，其中，类型或子类型的上下界将限制表达式采用的值的范围。（这种限制将导致运行时检查，可能会引起一个Constraint_Error异常。）如果**case**索引不是限制的，那么一般情况下需要一个**others**子句。

通常，**case**语句的实现选择合适的使用*跳转表*（jump table）或*搜索表*（search table）的方法。我们生成的代码使用了跳转表，这可能是最常见的方法了。该方法的优势在于：它比搜索表的执行效率更高，尽管它不像搜索表方法那样通用（因为其大小必须受某种限制）。在稍后概述的语义例程之后还有更多搜索表的讨论。使用跳转表生成的通用元组形式如图12-4所示。 445

```
                  {Evaluate <expr>}
                 (LT, <expr>, MinChoice, t1)
                 (JUMP1, t1, Others)
                 (GT, <expr>, MaxChoice, t2)
                 (JUMP1, t2, Others)
                 (JUMPX, <expr>, Table-MinChoice)
L1:                { code for <statement list 1> }
                 (JUMP, Out)
                 ...
LN:                { code for <statement list N> }
                 (JUMP, Out)
Others:            { code for <statement list> in others clause }
                 (JUMP, Out)
                 --  If no others clause is present, delete the preceding two lines.
Table:           (JUMP, L1) or (JUMP, Others)
                 ...
                 (JUMP, LN) or (JUMP, Others)
Out:
```

图12-4 **case**语句的元组序列

图中的代码框架里引入了一个新的元组形式。

(JUMPX, ARG1, ARG2) Indexed jump: Add the contents of the location specified by ARG1 to the tuple address specified by ARG2 and jump to that location

在代码框架里的JUMPX元组中，ARG2的元组地址由一个标号表达式来指定，其形式为：元组标号（Table）减去一个编译时常量（MinChoice）。

一个插入了语义动作记号的**case**语句文法如下：

```
<case statement>  →  case <expr> #start_case is <when list>
                       <others option> end case; #finish_case
<when list>       →  { when <choice list> => <stmts> #finish_choice }
<others option>   →  when others #start_others => <stmts> #finish_choice
<others option>   →  #no_others
<choice list>     →  <choice> { | <choice> }
<choice>          →  <expr> #append_val_or_subtype
<choice>          →  <expr> .. <expr> #append_range
```

446

其中，动作例程所需的两个语义记录是：

```
struct case_rec {
    struct type_ref index_type;
    list_of_choice choice_list;
    /* address of the JUMPX tuple */
    tuple_index jump_tuple;
    /* target of branches out */
```

```
    string out_label;
    /* label of the code for others clause */
    string others_label;
};
```

和

```
struct choice {
    long lower_bnd, upper_bnd;
    string start_label;
};
```

调用start_case()语义例程来开始**case**语句的处理。该例程生成显示在图12-4中自JUMPX起的元组。其中的若干元组依赖一个在**case**语句开始处尚不可用的值，因此在一遍编译器中它们必须加以回填。这个未知值将在下面的语义例程概述中用 ? 标记为后缀形式。start_case()同时创建和该语句对应的CASEREC语义记录。

```
start_case(<expr>) => case
{
    Test that <expr>'s type is an enumeration or
        subtype of Integer
    OthersL = new_label():
    Generate the following tuples:
        (LT, <expr>, MinChoice?, t1)
        (JUMP1, t1, OthersL)
        (GT, <expr>, MaxChoice?, t2)
        (JUMP1, t2, OthersL)
        (JUMPX, <expr>, TableLabel?-MinChoice?)
    Let A = next_tuple_number - 1;
    /* A is address of the JUMPX tuple */
    case ← (case_rec) {
        .index_type = <expr>.data_object.object_type;
        .choice_list = NULL;
        .jump_tuple = A;
        .out_label = new_label();
        .others_label = OthersL; }
}
```

447

例程append_var_or_subtype()和append_range()处理那些标记**case**语句中“选择”的常量、范围和子类型名字。它们中的每一个都为这些标签创建描述符并将之添加到CASEREC记录里的choice_list中。

```
append_val_or_subtype(case, <expr>) => case
{
  /*
   * <expr> may be an enumeration-valued constant
   * expression or a name that is a subtype with
   * constant constraints.  We assume all constant
   * expressions are folded.
   */

  Check that <expr> is a constant enumeration value or
       the name of an enumeration type or subtype with
       constant constraints
  Create a new struct choice, C

  if (<expr> is a DATAOBJECT record) {
     Check that <expr>.data_object.object_type ==
                case.case_rec.index_type
     /*
      * A single value in a <choice list> is treated as
      * a range with equal lower and upper bounds
      */
     C.lower_bnd = <expr>.data_object.value.int_value
     C.upper_bnd = <expr>.data_object.value.int_value
  } else {     /* <expr> must be a type or subtype */
     Check that <expr>.type_ref.object_type ==
```

```
                case.case_rec.index_type
    C.lower_bnd =
      <expr>.type_ref.object_type.constraint.lower_bound
    C.upper_bnd =
      <expr>.type_ref.object_type.constraint.upper_bound
  }

  C.start_label = new_label()
  generate(LABEL, C.start_label, "", "");
  Append C onto case.case_rec.choice_list
  case ← the updated struct case_rec
}


append_range(case, <expr1>, <expr2>) => case
{
   Let lower_bound rename <expr1>.data_object
   Let upper_bound rename <expr2>.data_object
   Check that lower_bound and upper_bound are constant
      enumeration values
   Check that lower_bound.object_type ==
               case.case_rec.index_type
   Check that upper_bound.object_type ==
               case.case_rec.index_type

   Create a new struct choice, C
   /*  We assume all constant expressions are folded */
   C.lower_bnd = lower_bound.value.int_value
   C.upper_bnd = upper_bound.value.int_value
   if (C.lower_bnd > C.upper_bnd)
      /* We have a null range */
      Issue a warning message
   else {
      C.start_label = new_label()
      generate(LABEL, C.start_label, "", "");
      Append C onto case.case_rec.choice_list
      case ← the updated struct case_rec
   }
}
```

448

finish_choice()在每个选择分支尾部生成跳出**case**语句所必需的跳转。start_others()和no_others()是可选的例程，我们用其中的某一个来处理特定的**case**语句，这取决于由**others**标识的选择是否存在。no_others()设置CASEREC记录中的others_label域为空串以通知finish_case()例程该语句中没有包含**others**部分。

```
finish_choice(case)
{
    generate(JUMP, case.case_rec.out_label, "", "");
}

start_others(case)
{
    generate(LABEL, case.case_rec.others_label, "", "");
}

no_others(case) => case
{
    case ← case.case_rec with others_label
                    set to "" (a null label)
}
```

动作例程finish_case()，见图12-5，将在**case**语句的最后结尾处被调用。它处理choice_list以创建跳转表并检查索引表达式的所有可能值是否恰好选择一个分支。它回填在语句开始处的元组中必要的域，为out_label生成LABEL元组，这是所有退出跳转的目标地址。

449

如果范围min_choice .. max_choice被选择值密集地覆盖，则用跳转表方法翻译**case**语句会来得简单且迅速。如果覆盖不是非常密集，那么跳转表也许会非常浪费空间，甚至可能超出存储容量。

```
finish_case(case)
{
   Let CR rename case.case_rec.

   min_possible = CR.index_type.constraint.lower_bound
   max_possible = CR.index_type.constraint.upper_bound
   choice_list = CR.choice_list

   Sort choice_list into ascending order based on values
       of lower_bnd
   Check that there is no overlap among choices (and
       ranges) by traversing the sorted list, and
       checking that choice_list[i].upper_bnd <
       choice_list[i+1].lower_bnd
   Set min_choice to be choice_list[first].lower_bnd
   Set max_choice to be choice_list[last].upper_bnd
   if (min_choice < min_possible ||
       max_choice > max_possible)
        Issue a warning about unreachable choices
   if (case.case_rec.others_label == "") {
       /* No others clause */
       if (min_choice > min_possible ||
            max_choice < max_possible)
          Issue an error because of index values not
          covered by any when clause
       Check that there is no gap among choices (and
          ranges) by traversing the sorted list, and
          checking that choice_list[i].upper_bnd+1 ==
          choice_list[i+1].lower_bnd
   }

   Let table = next_tuple_number++;
   Let T = CR.jump_tuple
   /*
    * At the beginning of the case statement,
    * we generated these tuples:
    *     (LT, <expr>, min_choice?, t1)
    *     (JUMP1, t1, others_l)
    *     (GT, <expr>, max_choice?, t2)
    *     (JUMP1, t2, others_l)
    *     (JUMPX, <expr>, table_label?-min_choice?)
    * where T is the tuple number of the JUMPX tuple
    */
450 if (case.case_rec.others_label == "")
       The first four of these tuples can be deleted
          (since a compile-time range check has been
          performed)
   else {
       Backpatch tuple[T].ARG3 with table-min_choice
       Backpatch tuple[T-2].ARG2 with max_choice
       Backpatch tuple[T-4].ARG2 with min_choice
   }

   for (i = min_choice; i <= max_choice; i++) {
       Let C be the first struct choice on choice_list
       if (C.lower_bnd <= i && i <= C.upper_bnd)
          generate(JUMP, C.start_label, "", "");
       else
          generate(JUMP, case.case_rec.others_label, "", "");
       if (i == C.upper_bnd)
          Remove C from choice_list
   }
   generate(LABEL, case.case_rec.out_label, "", "");
}
```

图12-5 动作例程finish_case()的概述

尽管某些编译器不担心这种意外会发生，但可能的话，最好还是生成某种搜索表而不是跳转表。例如，可以生成：

```
            (JUMP,Search)
              { code for all the cases }
Table:      (ChoiceValue,Address)
            ...
            (ChoiceValue,Address)
            (—,OthersAddress)
Search:       { code to search the table and jump to the appropriate address }
```

可以线形地搜索Table（例如，使用硬件搜索指令），或基于ChoiceValue对其排序并用二分搜索法进行搜索。甚至可能使用基于ChoiceValue的哈希方案。

如果（max_choice－min_choice）< 10，或如果50%或更多可能的选择标号在范围min_choice .. max_choice中出现，则UW-Pascal编译器生成跳转表。否则，它生成搜索表并使用硬件搜索指令执行线性搜索。 451

UNIX C编译器和Wisconsin UNIX Pascal编译器（均在VAX机器上运行）在以下三种方法中做选择。如果有多于三个的选择分支且覆盖了3/4以上范围，则使用前面描述过的跳转表。（VAX机器有一条指令可以合并范围检查和变址跳转。）否则，如果多于8个选择分支，它将生成代码在选择分支表上执行内联的二分搜索。因为分支选择的值集合是固定的且在编译时已知，所以可以编写不带循环的二分搜索代码。例如，如果有2n个选择分支，且val_n表示第n个分支的值，则可能生成的代码如下：

```
                (GT, expr, val_n, t1)
                (JUMP1, t1, SearchUpper_1)
                (GT, expr, val_n/2, t2)
                (JUMP1, t2, SearchUpper_2)
                  { code to search among cases 1..n/2 }
SearchUpper_2:    { code to search among cases n/2+1..n }
SearchUpper_1:  (GT, expr, val_3n/2, t3)
                (JUMP1, t3, SearchUpper_3)
                  { code to search among cases n+1..3n/2 }
SearchUpper_3:  { code to search among cases 3n/2+1..2n }
```

此方法为每个分支生成两条指令，但其中只有log(n)实际执行，这里n为分支数量。最后，如果没有一个条件成立，则将直接执行线性搜索。

可以合并上述三种方法以创建一种如下面伪代码所描述的混合方法：

```
if (size of choice range < minimum)
     generate a linear search
else if (choices are dense enough)
     generate code for a jump table
else {
     generate code to divide the range in half (as for
     binary search) and then recursively apply this
     algorithm to generate a search of each of the
     halves of the range
}
```

12.5 编译goto语句

即使在Ada/CS中没有**goto**语句，我们还是打算讨论它，因为在每一种实际的主要语言中都有**goto**语句，包括Ada。处理**goto**和处理**exit**所遇到的问题类似，因为某些**goto**，像**exit**一样，涉及跳到尚未定义标号的前向跳转。如果在编译器使用的IR中包括了元组的符号化标号，那么一旦明确特定的**goto**语句的合适目标标号，**goto**的翻译将很容易。如果要求一遍地址解析，则再一次地，可以使用任何间接跳转、链式引用或直接-间接技术。 452

为翻译**goto**语句，我们还必须处理以下新的问题：

- 同名标号可在多于一个的名字作用域中定义。
- 跳出块或过程的**goto**也许需要恢复寄存器、更新栈顶指针和修改显示表。

标号定义

首先考虑标号定义的问题，这里使用Algol 60和Pascal语言的作用域规则。例如，我们可以在一个Algol 60程序中找到如下的标号使用情况：

```
begin
   L: begin
         goto L;
         ...
         {possible definition of L}
      end
end
```

在处理这样的**goto**语句时，我们不清楚L将在哪个作用域中定义。因此，有必要推迟解析标号的引用直到我们确定哪个作用域含有正被引用的标号。

为解决这个问题，必须将标号标识符存放在符号表中。在看见**goto**时，标号标识符可能已被解析，而在这种情况下，因为我们已经看见了标号的定义实例，所以知道**goto**的目标将转向此标号，或者若该标号标识符尚未解析，那么此时我们将维持一个位置链，在这些位置上生成那些稍后必须回填的到目标地址的跳转。这个回填是必不可少的，即使用作JUMP元组目标的是符号化标号而不是元组序号，因为一个未解析的标号标识符没有与之关联的标号元组。

对于在Ada和Algol 60中编译**goto**，有三件事值得关注：(1) 已看见**goto**，(2) 已定义标号和(3) 已离开作用域。在前两个事件中的任一个发生时，在最内层作用域对应的符号表中可能有也可能没有此标号已解析的条目。在离开一个作用域时，我们必须处理在它相应的符号表中的那些已解析或尚未解析的标号条目。每种情况处理的方式不同，如图12-6中的列表所示。

	Resolved	Unresolved	No entry
goto L	generate	append chain	new unresolved entry
<<L>>	error	resolve	new resolved entry
Close scope	flush	propagate	—

图12-6 处理Ada和Algol 60中的**goto**

453 符号表条目代表以下动作：对于generate的情况，需要生成一个到包含在已解析条目中标号元组的跳转。对于append chain的情况，需要生成一个到未知地址的跳转，该跳转元组的位置将被添加到由此类位置组成的链中，该链位于未解析标号的符号表条目中。除了必须为标号创建一个符号表条目以及将跳转元组位置作为链中第一个元素以外，new unresolved entry情况与append chain情况很像。error的情况显而易见；这种情形表示在一个作用域中有重复的标号定义。对于resolve的情况，标号在符号表中的条目已从未解析变为已解析，此时将生成LABEL元组，并且回填链中所有跳转元组使之跳转到这个新的标号元组。对于new resolved entry的情况，仅需要生成一个LABEL元组并将该标号和对应的元组地址保存到符号表中。对于flush的情况，只需要从符号表中删除相应标号。因为这通常发生在退出一个作用域时，所以不需要其他任何显式的工作。propagate的情况是惟一复杂的情况。如果正要退出一个Algol 60程序的最外层作用域，或Ada程序的过程、包或任务的最外层作用域时，此时将表明有错误产生；一个未定义的标号已被作为了跳转目标。否则，在该标号的下一个作用域层，符号表可以包含也可以不包含该标号的条目。如果没有相应条目，则创建一个新的未解析的条目，并在该条目中包含那个随着作用域退出而正消失的链。如果存在一个已解析的条目，则使用其值回填正退出的作用域中所有未解析的条目链上的元组。如果已有一个未解析的条目，则将当前链附加到它的链上。

Pascal语言要求标号在作用域的首部声明，且必须在那个作用域而不是在某个内嵌的作用域中声明。在声明标号以后，将在符号表中建立一个未解析条目。因此，在编译**goto**时，在符号表中必定有该语句的目标标号条目，且该条目不需要在最内层作用域中。图12-7中的动作表对于Pascal来说很简单，因为

该语言没有propagate的情况，并且错误也能很快地被侦测出来。

	Resolved	Unresolved	No entry
Declare L	—	error	new unresolved entry
goto L	generate	append chain	error
L:	error	resolve	error
Close scope	flush	error	—

图12-7 处理Pascal中的**goto**

我们根据声明创建一个符号表条目，并且和以前一样，将标号的引用串成链直到该标号被定义为止。然而，在作用域结束时，不需要合并引用链。相反，我们会给出有关在作用域中声明却未定义的标号的立即诊断信息。

我们也可以使用链式技术来解决其他的标号作用域问题。许多语言（如Pascal和Ada）不允许从外面跳入某些结构（**for loop**、**while loop**和**case**语句）中。强制此规则的一个简单办法是：将在这样的 454
结构中定义的所有的标号链接在一起。结构中对这些标号的局部引用将按正常方式处理。然而，可以通过把这些标号标记为不可访问的来禁止那些来自结构外部的引用。例如，考虑：

```
for I := 1 to 10 do
   goto L; {illegal}
   for J := 1 to 20 do
      goto L; {legal}
      ...
      L:
   end {for J}
end {for I}
```

这里，我们将每个**for loop**中出现的所有前向引用串成链。内层的引用在循环分析结束时被正常解析。然后，L被标记为inaccessible。在外围循环分析结束时，它对L的引用不能被合法地解析。

跳出块或过程

我们现在考虑为跳出块或过程而需要的额外代码。在跳出前，需要弹出运行时栈顶、修改显示表和恢复寄存器。

不论我们使用块级还是过程级活动记录，弹出运行时栈顶不需要什么额外的工作。每个子程序和块均有局部的stack_top。当我们离开块或过程时，我们将其恢复到先前的stack_top值，这具有隐式弹出的效果。

如果使用了静态链簇，则不会有显示表，也就不要求有显示表的修改。然而，显式表的校正依赖于正确的返回序列。引用标号的**goto**要求标号定义在可见的外层词法作用域中（标号和变量有同样的作用域规则）。因此，在**goto**执行后所需要的部分显示表已经是正确的。但是，和我们先前看到的一样，即便在显示表中未用到的部分也必须是正确的。让我们回过头再看一下在9.2.1节中用来讨论显示表的例子： 455

```
procedure A is
   procedure B is
      procedure C is
         ··· A; ···
      end C;

      ··· C; ···
   end B;

   ··· B; ···
end A;
```

如果出现以下过程调用序列（其中'用来表示在某过程第一次调用返回前的第二次调用）：

A B C A′ B′ C′

那么，图12-8显示了每次调用时有效的显示表，以及每次保存的显示表值。

Calls	A	B	C	A′	B′	C′
display[2]	??	??	C	C	C	C′
display[1]	?	B	B	B	B′	B′
display[0]	A	A	A	A′	A′	A′
Saved	—	?	??	A	B	C

图12-8 goto前创建的显示表

如果C'直接通过**goto**返回到A'，则两个显示表寄存器B和C必须分别被恢复到层次2和3。一般地，要求为在**goto**的源和目标之间的动态链簇上的每一个活动记录恢复一个显示表寄存器。幸运的是，这项额外工作仅限于非局部**goto**的情况。

静态链簇避免了这些问题；其中每一个活动记录由正确的引用环境构成，且该环境不会被进一步的过程调用所改变。可以建立一个混合方案，它保留了静态链簇并构造了显示表。在这些方案中，期待着非局部的**goto**从静态链簇中重建全部的显示表，而正常的返回仅需恢复一个显示表寄存器即可。现在，每一个过程调用必须为建立静态链簇付出少许额外的代价。

跳到标号变量或标号参数的**goto**语句可以用静态链簇非常容易地实现，也可以用显示表来处理。其中的技巧是将它们视为形式过程。

456 在离开一个块时，我们不恢复寄存器，因此转出块的跳转不需要额外的工作。而在跳出一个过程时需要恢复寄存器（例如，如果**for**的索引被保存在寄存器中）。恢复寄存器的方法取决于寄存器第一次是如何被保存的。我们会在第13章中讨论一些可供选择的办法。

12.6 异常处理

在程序设计语言Ada的各种控制结构特性中，异常处理向编译器的编写者提出了巨大的挑战。和Ada语言中其他一些复杂特性（它们大多需要更大的编译时复杂度）不同，异常处理有着显著的运行时牵连。Ada语言的异常处理特性提出了许多有趣的问题：

- 异常可以被*隐式地或显式地*引发。
- 异常传播具有静态和动态的特征。在程序块和包中的异常，如果不在本地加以处理，则可以被静态地传播到外围包含单元；但在子程序中的异常可以被动态地传播到调用者（并被重新引发）。
- 异常名遵循普通的作用域规则，但异常传播却不是这样的。因此，有可能传播一个异常到某个该异常名不可知的作用域。
- 异常传播规则依赖于恰好引发异常的地方。特别地，在声明部分和异常处理程序中引发的异常同那些在语句中引发的异常传播方式不同。
- 在*单元*（unit）（块、包或子程序）体后可以可选地声明异常处理程序。我们期望许多单元不提供异常处理程序，但提供的处理程序的可能性也不应当削弱这些单元的翻译。
- 包和子程序可以被单独编译，其中异常可以在一个编译单元中声明和引发而在另一个编译单元中加以处理。

异常处理提出的基本问题是：当程序在执行期间引发异常时，必须查找和执行合适的异常处理程序。最直接的异常处理实现方法是保存某些代表异常处理程序的运行时数据结构并利用它们找到需要的处理程序。这种方法的缺点是：无论是在进入或退出包含处理程序的作用域时，都必须修改这些数据结构，因此即使没有引发异常，也需要相当多的执行时间花费。

我们更偏好那些不会招致额外开销的实现，除非我们使用了它们支持的某些特性。本节中提出的方法在异常被引发前没有运行时代价。决定合适的异常处理程序所付出的代价只依赖于到达那个处理程序所必需的隐式子程序返回次数。这个依赖可能是最少的，因为调用模式（和有效的异常处理程序）一般 457
不可能被事先预测。这种方法的空间需求也是适度的，它和异常处理程序个数和程序中子程序个数之和成比例。

我们首先提出的方案仅涉及在单个编译单元中异常传播的静态特性。也就是说，只考虑异常可能传播出嵌套的程序块和包，但不会传播出子程序。为了正确处理异常，在程序块或包中的任意点，需要知道从哪里可以找到适合给定异常的处理程序。回想：如果没有找到用户提供的处理程序，所有的异常都将有一个默认的处理程序。这个默认处理程序的特征是与实现相关的，但可能的默认处理程序只是简单地打印一条错误信息“异常X被引发”，然后终止程序的执行。

给每个异常（预定义的或用户定义的异常）指派一个惟一的从1开始的整数。一个异常在内部用这个整数码来表示，该整数可用来索引转移向量以找到异常处理程序。在开始翻译一个编译单元时，异常转移向量仅包括预定义的异常和所有默认条目（它们可以是处理默认情况的支持例程的地址）。

对每一个已声明的异常，将在转移向量中添加新的元素。当声明一个显式的异常处理程序时，必须计算一个已更新的转移向量。我们用一个转移向量覆盖的地址区间来限制这个转移向量。也就是说，如果异常e在地址a处引发，那么用e作索引在关联地址a的转移向量中搜索，然后跳转到合适的处理程序。每个处理程序在被翻译的时候包含一个程序继续执行的地址，即跟在包含异常处理程序的单元后面的语句或包的地址。在（通过转移向量进入的）异常处理程序执行完成后，它跳到那个继续执行地址以继续程序的执行。

在最简单的情况下，编译单元的程序体中不包括异常处理程序。因此，我们仅需要用单一的地址区间去覆盖整个编译单元。所关联的转移向量是那个含有预定义异常和默认处理程序的原始向量。

考虑图12-9中含有异常处理程序的编译单元实例。在最内层块中的异常处理程序可应用于地址区间3中的语句。位于包体末尾的异常处理程序可应用于地址区间2、4和5中的语句。预定义的异常处理程序可应用于地址区间1和6中的语句。除了Singular和Constraint_Error以外，其他可在区间3中引发的异常将传播到P的异常处理程序。因为P有一个处理**others**的处理程序，所以在区间2到5中引发的异常不可能传播得更远。

如图12-10所示，我们用了三个转移向量：第一个可用于区间1和6；第二个可用于区间2、4和5；第三个可用于区间3。因为在包P中提供处理**others**的处理程序，所以我们会把它的地址从转移向量中提取出来，并把它作为向量的默认条目。也就是说，如果向量中的一个位置含有一个空条目，那就使用这个默认地址。

如果没有提供**others**的处理程序，则异常可以传播出去。这意味着：如果没有局部的异常处理程 458
序，那么将使用外围包含单元的处理程序。这种情况发生在示例中的最内层块里。根据实现方法，我们已拷贝了外围包含单元，即程序包P（作为默认）的转移向量，然后为Constraint_Error和Singular更新有关条目以反映局部处理程序声明。

当最外层单元分析完毕，所有转移向量均已知。这时，我们得到一张地址区间表，每一个都有自己的转移向量。这些区间是连续的且覆盖整个程序。共享相同转移向量的相邻区间可以被合并（例如本例中的区间4和5）。为节省空间，可利用以下事实：即某个区间的开始地址要高于它前一个区间的结束地址。这允许我们针对每个区间保存一个地址（区间的开始地址）和一个转移向量的引用。

当算术运算或区间错误隐式地引发异常时，可以在区间表（见图12-11）中进行二分搜索以便快速 459
找到合适的转移向量，通过它我们随后可以进入合适的处理程序。对于显式引发的异常，可以生成转到合适处理程序的跳转指令，该处理程序的地址可在编译时决定（使用编码在转移向量中的数据）。

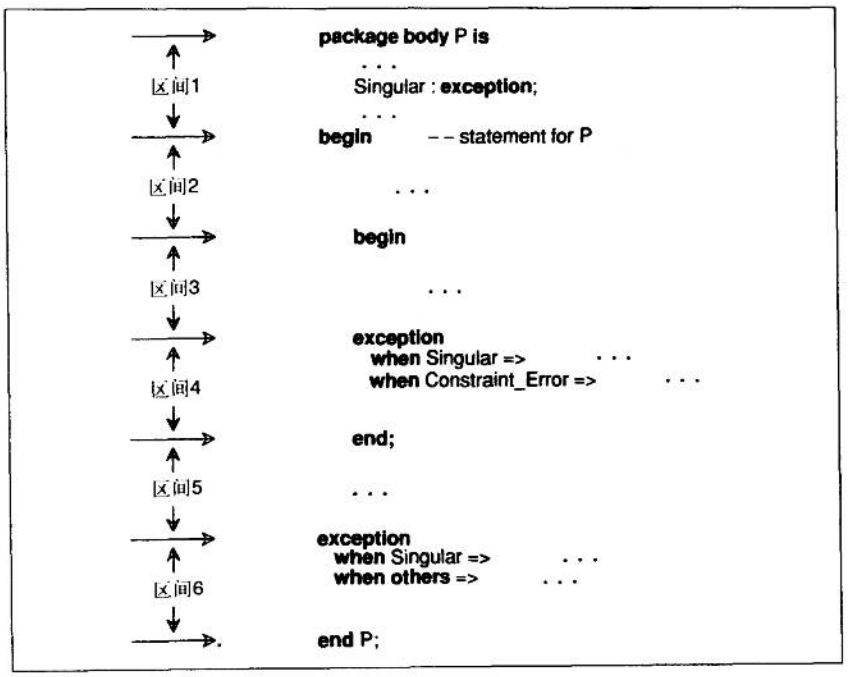


图12-9 带有异常处理程序的编译单元

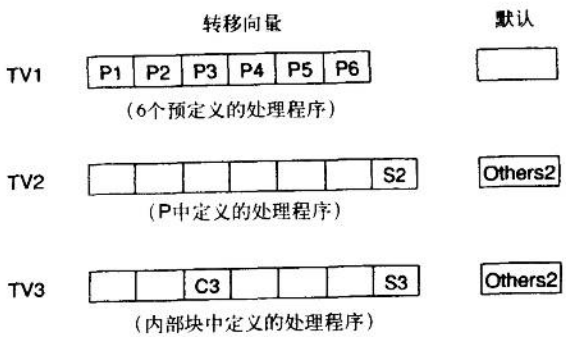


图12-10 图12-9中程序示例的转移向量

从子程序中传播异常

在开始翻译子程序时，我们创建一个覆盖整个子程序的区间，包括声明和局部定义的异常处理程序。这个区间有一个转移向量，这个转移向量的默认条目引用特殊的propagate_exception()例程。该例程执行一个正常的子程序返回并在返回时立即再次(隐式地)引发原来的异常。返回地址决定了合适的转移向量，而这个转移向量又决定了要进入的处理程序。

range 1	TV1
range 2	TV2
range 3	TV3
range 4	TV2
range 5	TV2
range 6	TV1

图12-11 图12-9中示例的区间图

460

在翻译完子程序中的所有单元时，如果声明了异常处理程序，则创建一个新的区间，包括与之关联的转移向量。在子程序中异常的传播是静态的，但如果不提供局部的处理程序，那么propagate_exception()例程将在调用点再次引发那个异常。

如果被调用的过程负责恢复寄存器，则propagate_exception()例程相当简单。我们将执行正常的子程序收尾工作（重置显示表、恢复寄存器等）。然而，我们没有跳转到返回地址，而是将它和异常码一起传递给隐式的异常处理程序，就好像一个隐式异常已被引发一样。如果由调用者来恢复寄存器，则我们必须用返回地址来定位并执行寄存器恢复代码，然后再调用隐式的异常处理程序。

表示转移向量和区间图

区间的数目不会超过子程序数和带有异常处理程序的嵌套单元数之和的两倍。因此，每个子程序区间图的表示只需要不多的存储字。包含处理程序声明的单元也有类似的存储开销。

不同转移向量的数目实际上就是声明异常处理程序的单元数。一般地，转移向量较稀疏，因为它们仅包含非默认的条目，对应着那些已提供显式的处理程序的异常。我们可以想像一个通过转移向量号（保存在区间图中）和异常码来索引的稀疏数组（sparse array）。已知有各种有效的稀疏数组（如分析表）的压缩算法。其中最合适的是双偏移索引，它提供了快速搜索和接近最优的压缩。我们将在第17章中详细讨论该方法。

分块编译

分块编译带来了另外一个问题：异常编号。回想我们先前曾给每一个异常指派一个惟一的整数码。我们允许每个编译单元独立地为在其中声明的异常指派编号。这多少有点像内部指派的地址被重定位一样，异常的编号也必须在执行前被重定位（relocated）。因此，异常的编号在编译单元之间就变得惟一，且转移向量的索引区间是程序中声明的异常总数。

压缩的转移向量数组由对应每个编译单元的子数组组成。当分块编译的模块被绑定在一起时，必须建立这个完整的数组并压缩它。

在许多情况下，在某个编译单元中声明的异常不能在另一个编译单元中访问。这种情形不会造成任何问题，并且在压缩时也不会引起任何空间惩罚。进一步地，我们采用的默认机制简单地处理了一个异常通过其名字不可知的作用域进行传播的情况。 461

在一遍编译器中的实现

在处理异常声明后，将指派给每个异常一个惟一的整数码，这就扩展了转移向量。在声明异常处理程序的时候，它在转移向量中为相关的语句列表产生一个显式的条目。每个处理程序在被翻译时，包含程序继续执行的地址。该地址是跟在包含异常处理程序的单元后面的语句或包的地址。在处理程序（通过转移向量进入）执行完以后，它跳到那个地址继续程序的执行。该跳转和**case**语句中跟在**when**子句后的出口地址的跳转类似。

在遇见**begin**时，我们会在语义栈上保存语句列表的首地址。在语句列表的末尾，可以看见**exception**（它指示处理程序定义的开始）或**end**。如果看见的是**end**，则弹出语句列表的首地址，因为这个语句列表使用与它的直接包含单元关联的转移向量。如果有**exception**，那么下一个可用的指令地址将作为从**begin**开始的区间的结束地址。在处理了紧随其后的异常处理程序声明之后，将构造一张表以包含异常码和相应异常处理程序的开始地址。在遇到**end**时，必须将这张表和相关的区间地址传播到外围包含单元的语义栈条目中，如果有的话。然后，弹出当前单元的栈条目。

在分析完最外层单元时，我们从默认向量开始，使用区间边界以及在处理嵌套单元时收集的异常处理程序表来计算所有的转移向量。然后，我们得到一张地址区间表，其中每个地址区间与一个转移向量关联（在前面的讨论中已提到过：多个地址区间可以共享一个向量）。如果在单元中保存了所有引发异常的语句列表，那么在所处理的异常不会传播到过程以外时，可以使用转移向量将raise exception n的库例程调用转换为到处理程序的直接跳转。我们还必须使用这些向量生成某种运行时表示用于处理待传播的异常，就像已描述的那样。

在图12-11中显示的区间图，通常是一种很有用的技术，除了异常处理的实现以外，它还可以应用

于其他许多问题。例如，在像Pascal这样的语言中，没有异常处理的特性，如果程序执行了某些非法操作（如企图使用非法下标值进行数组下标操作），那么它的执行将被终止。大多数编译器在这种情况下生成一条错误信息，其中包括了源程序中导致该错误的语句的行号。将代码段映射到行号的区间图，在
462 发生错误时，可用来从程序计数器的值导出有错误的行号。

12.7 短路计算布尔表达式

普通的布尔运算符**and**和**or**可以像其他所有的中缀运算符一样被处理。其中首先计算两个操作数，然后施用运算符计算结果到一个临时变量。然而，**and**的短路计算运算符**and then**以及**or**的短路计算运算符**or else**很有挑战性，它们意味着并不总是计算两个操作数。左操作数总是第一个被计算。如果那个计算足以决定表达式的值，那么就不再计算右操作数。（C语言的逻辑运算符&&和||总是采用短路方法来计算。）一个例子是：

```
if I /= 0 and then 10/I > R then
   write (10/I);
else
   write("Undefined");
end if;
```

作为*运算符*，短路计算布尔运算符可以和其他二元运算符进行组合，例如，A **and** B **and then** C **and** D。

我们生成的代码不产生代表布尔表达式的值，除非要求给一个变量赋值。该代码将根据表达式的值产生到合适元组的控制转移。在产生这样一个计算的过程中，每个操作数仅在必要时才被跳转到并计算。和为通常的布尔运算符生成的普通的*面向表达式的代码*（expression-oriented code）相比，我们把这种形式的代码称为*跳转代码*（jump code）。

为对比两种代码的形式，首先考虑A := A **and** B **and** C。面向表达式的代码可能是：

```
(AND, A, B, t1)
(AND, t1, C, t2)
(ASSIGN, t2, A)
```

而另一方面，对 A := A **and then** B **and then** C，我们可以生成以下跳转代码，它使用了条件分支元组，即如果ARG2为真则跳转到ARG3，反之如果ARG2为假则跳转到ARG4：

```
1 : (BR, A, 2, 6)
2 : (BR, B, 3, 6)
3 : (BR, C, 4, 6)
4 : (ASSIGN, True, t1)
5 : (JUMP, 7)
6 : (ASSIGN, False, t1)
7 : (ASSIGN, t1, A)
```

在此上下文中，跳转代码看起来很糟糕，但它们非常适合于那些询问布尔表达式的控制结构，如著名的
463 **if**语句和**while**循环。

对于 **if** A **and** B **and** C **then** . . . ，我们生成的面向表达式的代码可能是：

```
(AND, A, B, t1)
(AND, t1, C, t2)
(BR, t2, ThenAdr, ElseAdr)
```

类似地，对于 **if** A **and then** B **and then** C **then** . . . ，我们生成跳转代码：

```
1 : (BR, A, 2, ElseAdr)
2 : (BR, B, 3, ElseAdr)
3 : (BR, C, ThenAdr, ElseAdr)
```

上述每种情况均生成三个元组，但平均而言，后者代码执行得较快。例如，假设A、B和C为真的概率各为50%。对于面向表达式的代码，在所有的情况下我们都执行那三个元组。然而，对于跳转代码，

我们平均只执行1.75个元组，这是一个极大的改进。

正如我们先前看到的，DATAOBJECT记录在编译时包含着访问一个数据对象所需的所有数据，它还提供三种寻址模式：

(1) 直接常量。

(2) 直接地址。

(3) 间接地址。

也就是说，我们可以获得数据对象的值、它的地址或引用它的间接地址。

对于布尔值，我们添加第四种寻址模式，称其为*跳转代码*（jump code）。当 form == OBJECTJUMPCODE时，data_object包含两个域t_chain和f_chain。我们有：

```
enum object_form { OBJECTVALUE, OBJECTADDRESS,
       OBJECTJUMPCODE };

typedef struct data_object {
    type_descriptor *type_ref;
    enum object_form form;
    union {
        /* form == OBJECTVALUE */
        struct value value;

        /* form == OBJECTADDRESS */
        struct address addr;

        /* form == OBJECTJUMPCODE /
        struct {
            patch_node *t_chain, *f_chain;
        };
    };
} data_object;
```

464

t_chain和f_chain分别指示两个由patch_node记录组成的链表的表头，它们用来回填元组以跳转到的合适的位置，如果布尔表达式为真则用t_chain，若为假则用f_chain。

patch_node的定义如下：

```
typedef struct patch_node {
    /* Tuple to be backpatched */
    address_range tuple;

    /* Which field to be filled in? (1..4) */
    short field;

    /* next on chain */
    struct patch_node *next;
} patch_node;
```

我们需要通过t_chain和f_chain来回填，因为通常在布尔表达式翻译完之前我们不知道往哪里转移。

跳转代码是一种不寻常的布尔表达式计算方法，因为我们从不在寄存器或存储单元中实际存放布尔值。相反，这个值总是隐含在计算最后结束的地址里。不管怎样，这个地址模式都非常有用，因为布尔表达式经常用于控制流，而那些地方很适合于跳转代码。

对于布尔表达式，可以很容易地将其从跳转代码形式转换回直接地址形式。定义如下转换例程，convert_to_jump_code()和convert_to_boolean_value()。这两个例程生成必要的元组并创建struct data_object来描述转换的结果：

```
convert_to_jump_code(data_object *d)
{
    unsigned br_adr;
    if (d->object_form != OBJECTJUMPCODE) {
        br_adr = next_tuple_number++;
        generate(BR, d, TAdr?, FAdr?);
        d->object_form = OBJECTJUMPCODE;
```

```
        d->t_chain = (patch_node) {
                        .tuple = br_adr;
                        .field = 3;
                        .next = NULL; }
        d->f_chain = (patch_node) {
                        .tuple = br_adr;
                        .field = 4;
                        .next = NULL; }
    }
}
convert_to_boolean_value(data_object *d)
{
    address t;
    unsigned int a;

    if (d->object_form == OBJECTJUMPCODE ) {
        t = get_temporary;
        a = next_tuple_number++;
        generate(ASSIGN, TRUE, t);
        /* backpatch tuple on d.t_chain to address a */
        backpatch(d->t_chain, a);
        generate(JUMP, a+3, "");
        generate(ASSIGN, FALSE, t);
        /* backpatch tuple on d.t_chain to address a + 2 */
        backpatch(d->f_chain, a + 2);
        d->object_form = OBJECTADDRESS;
        d->addr = t;
    }
}
```

只要我们有了一个采用跳转代码形式表示的布尔表达式并需要其值（例如，在一个赋值语句中），就可以使用convert_to_boolean_value()。类似地，convert_to_jump_code()可以将布尔值转换为条件跳转（例如，在if语句中）。

下面Ada/CS文法中的产生式可以用来生成布尔表达式。从<b primary>到<factor>的产生式序列是必需的，因为not比逻辑运算符的优先级要高很多。

```
<b expr>      → <b primary> { <logical op> #start_op <b primary> #finish_op }
<b primary>   → <a expr> [ <relational op> <a expr> ]
<a expr>      → <term 1> { <adding op> <term 1> }
<term 1>      → <term>
<term>        → <factor> { <multiplying op> <factor> }
<factor>      → not <primary> #process_not
```

<b expr>、<b primary>、< a expr>、<term 1>、<term> 和<factor>具有DATAOBJECT格式的语义记录。进一步地，我们会使用下面新的语义记录来表示<logic op>:

```
struct bool_op {
    token operator;
    boolean short_circuit;
};
```

在分析了布尔运算符后将立即调用语义例程start_op()。如果运算符是短路计算运算符，它将开始生成跳转代码。

```
start_op(<b primary>, <logical op>) => <b primary>
{
    if (<logical op>.bool_op.short_circuit) {
        convert_to_jump_code(<b primary>.data_object);
        /* does nothing if <b primary> is already jump code */

        if (<logical op>.bool_op.operator == OR_OP)
            backpatch <b primary>.data_object.f_chain
                to next_tuple_number++
            /* if left operand is false, then right operand
                must be evaluated and tested. */
        else    /* operator is AND_OP */
            backpatch <b primary>.data_object.t_chain
```

```
            to next_tuple_number++
          /* if left operand is true, then right operand
            must be evaluated and tested. */
    ) else  /* not short circuit */
       convert_to_boolean_value(<b primary>.data_object);
    <b primary> ← the updated DATAOBJECT record
}
```

在分析完布尔表达式的第二个操作数后将调用语义例程finish_op()。如果生成的是跳转代码，则该例程仅需处理回填结点链；如果布尔值是想要的结果，那么该例程调用标准例程eval_binary()。

```
finish_op(<b primary1>, <logical op>, <b primary2>) => <b expr>
{
    Check that <b primary1>.data_object.object_type and
        <b primary2>.data_object.object_type are BOOLEAN
    if (<logical op>.bool_op.short_circuit) {
       convert_to_jump_code(<b primary2>.data_object);
       if (<logical op>.bool_op.operator == OR_OP) {
          <b expr> ← (data_object) {
             .t_chain =
                merge(<b primary1>.data_object.t_chain,
                      <b primary2>.data_object.t_chain);
             /*
              * The whole expression is true if the left
              * operand is true or if the left operand is
              * false and right operand is true.
              */
             .f_chain = <b primary2>.data_object.f_chain; }
             /*
              * The whole expression is false if both the
              * left and right operands are false.
              */
       } else {   /* operator == AND_OP */
          <b expr> ← (data_object) {
             .f_chain =
                merge(<b primary1>.data_object.f_chain,
                      <b primary2>.data_object.f_chain),
             /*
              * The whole expression is false if the left
              * operand is false or if the left operand is
              * true and right operand is false.
              */
             .t_chain = <b primary2>.data_object.t_chain; }
             /*
              * The whole expression is true if both the
              * left and right operands are true.
              */
       }
    } else {    /* ordinary operator */
       convert_to_boolean_value(<b primary2>.data_object);
       <b expr> ← eval_binary(<b primary1>,
                              <logical op>, <b primary2>)
    }
}
```

在生成跳转代码的时候，可以直接实现**not**运算符。例程process_not()中所需要做的仅仅是交换t_chain和f_chain指针：

```
process_not(<primary>) => <factor>
{
    Check that <primary>.data_object.object_type is BOOLEAN
    if (<primary>.data_object.form == OBJECTJUMPCODE) {
       Interchange <primary>.data_object.t_chain and
          <primary>.data_object.f_chain
       <factor> ← the updated DATAOBJECT record
    } else
       <factor> ← eval_unary(NOT_OP, <primary>)
}
```

467

为说明这些例程是如何工作的，我们将布尔表达式

If A or else not (B and then C) then ···

的追踪分析结果放在图12-12中。这其中仅包含了那些生成元组或修改t_chain和f_chain的步骤。在
468 完成所有回填后，可从图12-12中得到最终的元组是：

```
1 : (BR, A, ThenAdr, 2)
2 : (BR, B, 3, ThenAdr)
3 : (BR, C, ElseAdr, ThenAdr)
```

我们很容易看出这些代码是正确的。

	Actions	Generated Code
(1)	start_op(**or else**) convert_to_jump_code(A) A.TC = (1,3) A.FC = (1,4) Backpatch A.FC to 2 A.TC = (1,3) A.FC = NULL	1 : (BR,A,?,?) 1 : (BR,A,?,2)
(2)	start_op(**and then**) convert_to_jump_code(B) B.TC = (2,3) B.FC = (2,4) Backpatch B.TC to 3 B.TC = NULL B.FC = (2,4)	2 : (BR,B,?,?) 2 : (BR,B,3,?)
(3)	finish_op(**and then**) convert_to_jump_code(C) C.TC = (3,3) C.FC = (3,4) Result1.TC = C.TC = (3,3) Result1.FC = merge(B.FC,C.FC) = ((2,4),(3,4))	3 : (BR,C,?,?)
(4)	process_not() Result2.TC = ((2,4),(3,4)) Result2.FC = (3,3)	
(5)	finish_op(**or else**) Result3.TC = merge(A.TC,Result2.TC) = ((1,3),(2,4),(3,4)) Result3.FC = Result2.FC = (3,3)	
(6)	As we process the **If**, we Backpatch Result3.TC to the ThenAdr and Backpatch Result3.FC to the ElseAdr	

图12-12 追踪短路计算布尔表达式翻译

在表达式中可以任意地混合使用短路计算的以及正常的布尔运算符，尽管这种混合往往由于要经常来回转换跳转代码而生成较差的代码。因此，翻译下面的语句

469 **A := A and B and then C and D;**

将生成：

```
1 : (AND, A, B, t1)
2 : (BR, t1, 3, 6)
3 : (BR, C, 4, 6)
4 : (ASSIGN, True, t2)
5 : (JUMP, 7)
6 : (ASSIGN, False, t2)
7 : (AND, t2, D, t3)
8 : (ASSIGN, t3, A)
```

另外，涉及常量的短路计算表达式不容易进行常量折叠。因此，使用上述例程翻译下面的语句

A := True and then False and then True;

将生成:

```
1 : (BR, True, 2, 6)
2 : (BR, False, 3, 6)
3 : (BR, True, 4, 6)
4 : (ASSIGN, True, t1)
5 : (JUMP, 7)
6 : (ASSIGN, False, t1)
7 : (ASSIGN, t1, A)
```

上述代码虽正确但几乎没有一点优化。这个问题在于：在上下文中，例如**False and then**. . .，我们不知道右操作数是否为常量。如果是，则整个表达式可在编译时进行常量折叠。如果不是，则必须为右操作数生成代码。所以，我们必须生成BR或JUMP元组来阻止它的计算。

为了折叠短路计算表达式，可以首先建立一棵表达式树而不必生成代码，优化它（例如，通过折叠）。然后再从这棵树上生成代码。这种方法可以很好地工作，但它不太适合一遍编译器。

最后要注意的是，某些编译器为所有的布尔运算符生成跳转代码（并经常声称布尔表达式是经过优化的）。如果语言的定义不要求所有的操作数在所有情况下都要加以计算（例如，为强制可能的副作用），那么短路代码将是最好的。Pascal使这种选择成为实现相关的。因此，某些Pascal编译器通过短路代码实现布尔表达式，而其他的Pascal编译器则生成普通的表达式代码。事实上，以前的Berkeley Unix Pascal解释器和编译器（*pi*和*pc*）就区别在这一点上。UW-Pascal编译器的实现方法是，在条件语句（**if**、**while**和**repeat**）中生成短路代码，而在赋值语句、参数表达式等情况下生成普通的表达式代码。这种折中方法改进了整个代码的质量，但也容易导致意想不到的结果。例如，表达式 (I <>0) **and** (A/I > R) 在**if**语句中不会出现除法错误，但在赋值语句中却可能。因此，在这个例子中，由于缺乏统一的语言定义而导致了不一致的实现，而这妨碍了程序的可移植性。 470

在Ada语言中，运算符**and**和**or**（以及所有非短路计算运算符）都必须被计算，但计算的次序却没有指定。可以使用普通的从左到右计算次序，但作为一种优化措施也可以重新编排计算次序。

12.7.1 单地址短路计算

在前一节描述的语义例程假设生成的是元组形式的中间代码。在多数情况下，语义例程不需要改变很多即可直接生成机器代码。然而，短路计算布尔表达式的技术严重依赖于这样的假设，条件分支指令指定两个地址：一个是条件成立的分支，而另一个则是条件不成立的分支。

以下方法基于Logothetis和Mishra(1981) 所开发的技术。该技术尽可能地推迟跳转的生成，而同时又记住在最后生成的时候这些跳转应当以何面目出现。

我们假设除了普通的算术操作，目标语言还有这样的操作：(ASSIGN, loc1, loc2)，该操作拷贝loc1的内容到loc2；(NOT, loc1, loc2)，该操作将loc1中的内容（一个布尔值）的逻辑非赋值给loc2。我们同样假设有比较操作(CMP, loc1, loc2)，该操作设置条件码和条件分支(B,cond,loc)，这个分支操作根据条件码进行控制流分支转移（cond是GT、GE、LT、LE、EQ、NE和ALWAYS中的一个）。注意，CMP操作可用第7章的元组运算符通过减法操作来模拟，其结果可被临时看作一个条件码。通常，我们将推迟生成条件分支操作直到我们清楚它应当使用什么条件为止。

DATAOBJECT语义记录必须做如下扩展:

```
enum cond { LT, LE, EQ, NE, GT, GE };
enum object_form { OBJECTVALUE, OBJECTADDRESS,
       OBJECTJUMPCODE };

typedef struct data_object {
    type_descriptor *type_ref;
    enum object_form form;
    union {
        /* form == OBJECTVALUE */
```

```
        struct value value;

        /* form == OBJECTADDRESS */
        struct {
            struct address addr;
            boolean negated;
        }
        /* form == OBJECTJUMPCODE /
        struct {
            patch_node *t_chain, *f_chain;
            enum cond condition;
        };
    };
} data_object;
```

471

t_chain和f_chain同先前一样：即为位置的链表，这些位置上的分支指令在表达式确定为真（t_chain）或为假（f_chain）时将跳出表达式。域condition解释最近的CMP指令的含义如下（若有的话）：例如，假定condition == GT，那么，如果我们想在表达式为假时使控制转移到label以及表达式为真时使控制进入下一条语句，则应在已生成的代码后跟有(B, GT, label)。换句话说，condition表明这种分支所跟的代码应在条件成立时使控制流顺序流入。OBJECTADDRESS变体的negated域用来避免不必要的**not**代码。

假设我们使用以下函数来操作patch_node记录的链表：append(location, chain2)函数将一个位置添加到链表中并返回已更新的链表；merge(chain1, chain2)函数返回两个链表连接的新链表；过程backpatch(location, chain)将链中所有指令的地址域置成给定的位置。在稍后概括描述的例程中还使用了一些其他的子程序。to_jump_code()和from_jump_code()是转换例程，它们很像上一节中的那些例程：

```
to_jump_code(data_object *b)
{
    enum cond c;

    generate(CMP, b, TRUE, "");
    c = b->negated ? EQ : NE;

    b->form = OBJECTJUMPCODE;
    b->t_chain = NULL;
    b->f_chain = NULL;
    b->condition = c;
}


from_jump_code(data_object *b, address t)
{
    /* t is a temporary supplied  */
    /* by the calling routine     */

    fall_through_on_true(b);
    generate(ASSIGN, TRUE, t);
    generate(b, ALWAYS, next_tuple_number + 2, "");
    backpatch(next_tuple_number++, b->f_chain);
    generate(ASSIGN, FALSE, t, "");

    b->form = OBJECTJUMPCODE;
    b->addr = t;
    b->negated = FALSE;
}
```

472

fall_through_on_true()和fall_through_on_false()在计算单个操作数后将生成合适的分支并调整patch_node链到合适的直接控制流上：

```
/*
 * Fix up the code for b to fall through to the
 * next statement if b is true
 */

fall_through_on_true(data_object *b)
{
    if (b->form != OBJECTJUMPCODE)
        to_jump_code(b);
    append(next_tuple_number, b->f_chain);
    generate(b, b->condition, ?, "");
    backpatch(next_tuple_number, b->t_chain);
    b->t_chain = NULL;
}


/*
 * Fix up the code for b to fall through to the
 * next statement if b is false
 */

fall_through_on_false(data_object *b)
{
    if (b->form != OBJECTJUMPCODE)
        to_jump_code(b);
    append(next_tuple_number, b->t_chain);
    generate(b, complement(b->condition), ?, "");
    backpatch(next_tuple_number, b->f_chain);
    b->f_chain = NULL;
}
```

可用negate()例程来实现**not**操作而不必生成任何代码。它使用使条件翻转的例程complement()： 473

```
enum cond complement(enum cond c)
{
    switch (c) {
    case LT:    return GE;
    case LE:    return GT;
    case EQ:    return NE;
    case NE:    return EQ;
    case GT:    return LE;
    case GE:    return LT;
    }
}

negate(data_object *b)
{
    if (b->form == OBJECTVALUE)
        b->value.int_value = - b->value.int_value;
    else if (b->form == OBJECTADDRESS)
        b->negated = ! b->negated;
    else {      /* b->form == OBJECTJUMPCODE */
        exchange b->t_chain and b->f_chain;
        b->condition = complement(b->condition);
    }
}
```

为清楚起见，图12-13给出的布尔表达式文法仅包含短路计算布尔运算符。和在Ada文法中一样，**and then**和**or else**具有相同的优先级。类似地，出于简单性的考虑，我们省略了算术表达式（<a expr>）和“其他”语句（<other stmt>）的产生式和语义例程。

ftt()和ftf()语义例程主要使用先前给出的fall_through_on_true()和fall_through_on_false()例程：

```
ftt(<b primary>) => <b expr>
{
    fall_through_on_true(<b primary>.data_object)
    <b expr> ← the updated DATAOBJECT record
}
```

```
ftf(<b primary>) => <b expr>
{
    fall_through_on_false(<b primary>.data_object)
    <b expr> ← the updated DATAOBJECT record
}
```

```
<b expr>       → <b primary> { <b expr tail> }
<b expr tail>  → #ftt and then <b primary> #bool_op
<b expr tail>  → #ftf or else <b primary> #bool_op
<b primary>    → <a expr> [ <relational op> <a expr> #compare ]
<a expr>       → <term 1> { <adding op> <term 1> }
<term 1>       → <term>
<term>         → <factor> { <multiplying op> <factor> }
<factor>       → not <primary> #process_not
<stmt>         → <other stmt> ;
<stmt>         → <var> := <b expr> #b_assign ;
<stmt>         → if #start_if <b expr> #if_test then <stmt list>
                   { elsif #gen_jump #patch_else_jumps <b expr> #if_test
                   then <stmt list> } <else part> end if ; #patch_out_jumps
<else part>    → else #gen_jump #patch_else_jumps <stmt list>
<else part>    → #patch_else_jumps
<stmt>         → while #start_while <b expr> #while_test
                   loop <stmt list> end loop #finish_while
```

图12-13 布尔表达式文法

474 除了在必要时转换到跳转代码，bool_op()和process_not()的工作全部通过patch_node记录链上的操作来实现：

```
bool_op(<b expr>, <b primary>) => <b expr>
{
    if (<b primary>.data_object.form != OBJECTJUMPCODE)
        to_jump_code(<b primary>.data_object);
    <b expr> ← (data_object) {
            .form = OBJECTJUMPCODE;
            .t_chain = merge(<b expr>.t_chain,
                       <b primary>.t_chain);
            .f_chain = merge(<b expr>.f_chain,
                       <b primary>.f_chain);
            .condition = <b primary>.condition; }
}


process_not(<primary>) => <factor>
{
    negate(<primary>.data_object)
    <factor> ← the updated DATAOBJECT record
}
```

475 compare()用CMP元组来实现关系运算符并创建一个DATAOBJECT记录来描述跳转代码格式中的结果：

```
compare(<a expr1>, <relational op>, <a expr2>) => <b primary>
{
    Check types of <a expr1> and <a expr2> for compatibility
    generate(CMP, <a expr1>.data_object,
                  <a expr2>.data_object, dummy);
    <b primary> ← (data_object) {
        .form = OBJECTJUMPCODE;
        .t_chain = NULL;
        .f_chain = NULL;
        .condition = complement(<relational op>.op.operator); }
}
```

为处理短路计算布尔运算符，我们在DATAOBJECT记录中添加了一些扩展，由此带来的特殊情况需要使用b_assign()例程来处理。因为图12-13中赋值产生式实际上不区分布尔表达式和其他表达式，所以该例程的处理过程描述可以简单地通过合并标准赋值动作例程来得到：

```
b_assign(<var>, <b expr>)
{
    if (<b expr>.data_object.form == OBJECTJUMPCODE)
        from_jump_code(<b expr>.data_object,
                       <var>.data_object.addr);
    else if (<b expr>.data_object.negated)
        generate(NOT, <b expr>.data_object,
                       <var>.data_object, "");
    else
        generate(ASSIGN, <b expr>.data_object,
                       <var>.data_object, "");
}
```

这里处理if和while语句所需的语义记录和我们在本章开始看到的略有不同，因为我们现在考虑的是patch_node链表而不是标号。

```
struct if_stmt {
    patch_node *out_list, *else_list;
};

struct while_stmt {
    string top_tuple;
    patch_node *out_list;
};

start_if(void) => if
{
    if ← (if_stmt) {
                   .out_list = NULL;
                   .else_list = NULL; }
}
if_test(if, <b expr>) => if
{
    Check that <b expr>.data_object.object_type == BOOLEAN
    fall_through_on_true(<b expr>.data_object)
    if.if_stmt.else_list = <b expr>.data_object.f_chain
    if ← the updated IFSTMT record
}
```

476

gen_jump()例程生成的跳转代码可以跳过跟在then部分后的else或elsif部分：

```
gen_jump(if) => if
{
    Create a new patch_node, P, for next_tuple_number
    if.if_stmt.out_list = append(if.if_stmt.out_list, P)
    generate(B, ALWAYS, ?, "");
    if ← the updated IFSTMT record
}
```

在else或elsif部分开始处调用的patch_else_jumps()将补填else_list链表中的所有元组以跳转到next_tuple_number：

```
patch_else_jumps(if)
{
    backpatch(next_tuple_number, if.if_stmt.else_list);
}
```

在全部语句处理完后，patch_out_jumps()将补填out_list链表中的所有元组以跳转到next_tuple_number：

```
patch_out_jumps(if)
{
    backpatch(next_tuple_number, if.if_stmt.out_list);
}

start_while(void) => while
{
    L = new_label();
```

```
    generate(LABEL, L, "", "");
    while ← (while_stmt) {
        .top_tuple = L;
        .out_list = NULL; }
}

while_test(while, <b expr>) => while
{
    Check that <b expr>.data_object.object_type == BOOLEAN
    fall_through_on_true(<b expr>.data_object);
    while.while_stmt.out_list = <b expr>.data_object.f_chain
    while ← the updated struct while_stmt
}
```

477

我们对finish_while()很熟悉，它基本上是在if语句和简单循环语句结尾处调用的例程的组合：

```
finish_while(while)
{
    generate (B, ALWAYS, while.while_stmt.top_label, "")
    backpatch(next_tuple_number, while.while_stmt.out_list);
}
```

为说明这些例程是如何工作的，我们将布尔表达式

```
if A <= B and then A >= 0 or else A = 100 then A := 1; end if;
```

的追踪分析结果放在图12-14中。再一次地，其中仅包含了那些生成元组或修改t_chain和f_chain的步骤。

	Actions	Generated Code	Semantic Stack (t_chain, f_chain, cond)
(a)	#compare	1: CMP A,B	(null,null,GT)
(b)	#ftt		
	Append 2 to FC	2: B GT,?	(null,2,GT)
	Backpatch TC to 3		(null,2,GT)
(c)	#compare	3: CMP A,0	(null,null,LT) (null,2,GT)
(d)	#bool_op		(null,2,LT)
(e)	#ftf		
	Append 4 to TC	4: B GE,?	(4,2,LT)
	Backpatch FC to 5	2: B GT,5	(4,null,LT)
(f)	#compare	5: CMP A,100	(null,null,NE) (4,null,LT)
(g)	#bool_op		(4,null,NE)
(h)	#ftt		
	Append 6 to FC	6: B NE,?	(4,6,NE)
	Backpatch TC to 7	4: B GE,7	(null,6,NE)
(i)	#assign	7: ASSIGN 1,A	
(j)	#patch_out_jumps		
	Backpatch FC to 8	6: B NE,8	(null,null,NE)

478

图12-14 三地址短路计算布尔表达式的翻译过程

在完成所有回填后，可从图12-14中得到最终的元组是：

```
1: CMP   A,B
2: B     GT,5
3: CMP   A,0
4: B     GE,7
5: CMP   A,100
```

```
6: B        NE,8
7: ASSIGN  1,A
```

我们很容易看出这些代码是正确的。

练习

1. 追踪在编译下面程序片段时所调用的语义例程序列。给出生成的元组，并标明生成它的例程。假设I、J和K均为Integer变量。

a.
```
if I > J then
   K := I;
elsif J > I then
   K := J;
else
   K := 0;
end if;
```

b.
```
if I > J then
   K := I;
elsif J >= I then
   K := J;
end if;
```

2. 追踪在编译下面程序片段时所调用的语义例程序列。给出生成的元组，并标明生成它的例程。假设I、J、Limit和Sum均为Integer变量，而A是一个二维Integer数组。

```
            Sum : = 0;
            I := 0;
OuterLoop:  loop
               I := I + 1;
               exit when I > Limit;
               J := 1;
               while J <= Limit loop
                  exit OuterLoop when A(I,J) = 0;
                  Sum := Sum + A(I,J);
                  J := J + 1;
               end loop;
            end loop;
```

3. 追踪在编译下面程序片段时所调用的语义例程序列。给出生成的元组，并标明生成它的例程。假设I和Limit均为Integer变量，而B是一个一维Integer数组。 479

```
Sum := 0;
for I in 1..Limit loop
   Sum := Sum + B(I);
end loop;
```

4. 写一个包括**others**在内的有至少6个标号选择的**case**语句。标号应该包含区间以及单个值。追踪在编译你的**case**语句时所调用的语义例程序列。给出除了`finish_case()`以外的所有动作例程生成的元组，并画出为描述选择标号而创建的数据结构。
5. 使用跳转表方法，给出`finish_case()`为练习4中的**case**语句所生成的元组。
6. 使用内联二分搜索方法，给出`finish_case()`为练习4中的**case**语句所生成的元组。
7. 重写用来编译基本的**loop**和**while loop**的语义例程概述和语义记录声明，以便采用回填而不是符号标号来处理跳转地址解析的问题（提示：参见在12.1节末尾有关回填的讨论）。
8. 重写用来编译**if**语句的语义例程概述和语义记录声明，以便采用回填而不是符号标号来处理跳转地址解析的问题。
9. 描述在编译练习2中的嵌套循环时，`loop_descriptor`栈所发生的一系列变化。说明是哪个例程造成了这些变化。
10. 根据总结在图12-6中的技术，说明编译器为下面Ada程序段中的标号和**goto**语句所做的处理工作。 480

```
declare
   ...
begin
   <<L>>
```

```
declare
  ...
begin
  ...
  goto L;
  ...
  goto M;
  ...
  <<M>>
  ...
end;
...
end;
```

11. 使用12.6节中的技术，说明下面程序中重要的地址区间、异常转移向量和区间图。解释过程Q中的**raise**语句以及在Q中出现的异常Constraint_Error都是如何处理的。

```
procedure P is
  ...
  Fault : exception;

  procedure Q is
  begin
    ...
    raise Fault;
    ...
  exception
    when Fault => ...;
  end Q;

begin
  ...
  Q;
  ...
exception
  when Constraint_Error => ...;
end P;
```

481

12. 使用12.6节中的技术，说明下面程序中重要的地址区间、异常转移向量和区间图。解释当过程R分别在Q中和P中被调用时，其中的**raise**语句是如何处理的。

```
procedure P is
  ...
  Fault : exception;

  procedure R is
  begin
    ...
    raise Fault;
    ...
  end R;

  procedure Q is
  begin
    ...
    R;
    ...
  exception
    when Fault => ...;
  end Q;

begin
  ...
  Q;
  ...
  R;
  ...
exception
  when Fault => ...;
end P;
```

13. 在异常处理特性和交互式调试器之间存在着有趣的交互。正常情况下，在发生运行错误时调用这样的调试器。然而，在一个具有异常处理特性的语言中，这些错误显然被当作了预定义的异常。因此，只有在（预定义或在程序中定义的）异常没有配备处理程序的时候才会调用这种调试器。解释为了与调试器交互，应该如何修改12.6节中的异常处理实现技术。假定调试器在被授予控制权时应该带有引发未经处理的异常的程序状态视图。

14. 使用12.7节中提出的技术，针对以下语句片段，做出像图12-12那样的翻译过程的追踪。

```
if A and B and then C or else not D then ···
```

482

15. 使用12.7.1节中提出的技术，针对以下语句，做出像图12-14那样的翻译过程的追踪。

```
if A >= C or else not ( C <> A and then B < 77 ) then A := B; end if;
```

483

第13章 翻译过程和函数

过程和函数（此后通称为子程序）的翻译包括两个基本步骤：处理声明和处理调用。首先，编译器遇到的是子程序的声明。该声明中包含有其他的声明，因此在子程序声明中将构建符号表条目和相关属性记录。在子程序声明以后，即可引用子程序且必须用在声明处得到的属性信息来正确地翻译这些引用。从子程序角度看，这些引用称为调用（call）。因为子程序调用中隐含着控制转移及可能伴随的参数，所以处理子程序名字的引用比处理其他声明的引用要复杂得多。

484

13.1 简单子程序

13.1.1 声明无参子程序

我们必须处理以下几种形式的子程序声明。Ada和Ada/CS中的过程和函数具有不同的语法形式，它们允许每一种子程序在声明时可以推迟定义其过程体描述。因此，在下面的产生式中，可以看到4种子程序声明：有或无过程体的过程，有或无函数体的函数：

```
<declaration>      → <subprogram spec> <subprogram tail>

<subprogram spec>  → procedure <id> #start_proc [ <formal part> ]
                   → function <designator> #start_proc [ <formal part> ]
                     return <type name> #return_type

<subprogram tail>  → ; #end_proc_spec
                   → is #start_proc_body <decl pt>
                     begin <stmts> <exception part> end
                     [ <designator> #check_proc_id ] ;
                     #end_proc_body
```

上述产生式中的语义例程使用了一个新的语义记录类型SUBPROGRAM。此记录类型的实例由start_proc()创建并对所有处理子程序声明的例程可用。

```
typedef struct subprogram {
    attributes *old_current_proc;
    id_entry st_entry;
    string end_label;
} subprogram;
```

必须扩展attributes类型以包含下面的变体：

```
/* class == SUBPROGRAMNAME */
struct {
    level_range nesting_level;
    string start_label;
    address_range activation_rec_size;
    symbol_table local_decls;
    attributes *parameters;
    struct type_des *return_type;
        /* -- NULL except for functions */
    boolean body_declared;
};
```

485

首先，我们考虑处理无参过程和函数所需的语义例程。我们使用一个称为current_proc的全局变量，它包含一个指针，指向当前正在处理的最内层子程序的属性记录。

```
start_proc(<id>) => procedure
{
   Enter the identifier represented by <id> into the
       symbol table for the current scope.
   Due to the possibility of overloading, it may already
       be present in this scope.
   Save the id_entry returned by enter() in subprog_entry.
   Let A = (attributes) {
       .class = SUBPROGRAMNAME;
       .id = subprog_entry;
       .id_type = NULL;
       .nesting_level = current_proc.nesting_level + 1;
       .start_label = "";
       .activation_rec_size = CONTROLSIZE;
       .local_decls = create();
       .parameters = NULL;
       .return_type = NULL;
       .body_declared = FALSE; }

   procedure ← (subprogram) {
       .old_current_proc = current_proc;
       .st_entry = subprog_entry;
       .end_label = ""; }
   current_proc ← A;
}
```

在被创建用来描述一个新子程序的attributes结构里，我们用名为CONTROLSIZE的常量来初始化activation_rec_size域。该常量的值是实现相关的，它表示在每个活动记录的开始部位所存储的控制信息（参见9.1.2节）的空间大小。（在10.2.1节描述的）语义例程var_decl()将使用这个域在处理变量声明时指派偏移位置。在声明每个变量时，将使用current_proc结构的activation_rec_size域获取一个偏移值送给该变量的attributes记录中的address的var_offset域。为给该变量分配空间，我们将它的类型对象所占字节的大小加至current_proc.activation_rec_size上。

```
return_type(<type or subtype>)
{
   /*
    * Set the return_type field of the function currently
    * being compiled using the TYPEREF record
    * representing <type or subtype>
    */
   current_proc.return_type ←
       <type or subtype>.type_ref.object_type
}

end_proc_spec(procedure)
{
   /*
    * Called only for subprogram specifications
    * that appear without a body
    */
   string start
   start = new_label()
   /*
    * A corresponding label tuple will be generated
    * when the body is encountered
    */
   Record the label start as current_proc.start_label
   Call the symbol table routine set_attributes() to
       associate current_proc with
       procedure.subprogram.st_entry.
   Generate an error message if this call fails because
       st_entry already has associated attributes that
       current_proc may not overload.
   current_proc ← procedure.subprogram.old_current_proc
}
```

486

```
start_proc_body(procedure) => procedure
{
   Take the appropriate actions to make
       current_proc.local_decls the current scope.
   /*
    * This may require no action if creating a scope
    * automatically does this.  Such an implementation
    * is common for languages without packages or a
    * similar feature.
    */
   procedure.subprogram.end_label = new_label()
   generate(JUMP, procedure.subprogram.end_label, "", "");
   /*
    * Execution of enclosing scope will jump
    * over this procedure body
    */
   if (current_proc.start_label == "")
       current_proc.start_label = new_label();
   generate(LABEL, current_proc.start_label, "", "");
   generate(STARTSUBPROG, current_proc.nesting_level,
            "", "");
   procedure ← the updated SUBPROGRAM record
}
```

我们必须在处理声明列表之前生成STARTSUBPROG元组，因为Ada/CS中的声明可以导致有关代码的生成。例如，变量的初始化表达式和动态大小的数组产生那些必须在子程序体首部执行的代码。如果我们正在处理像Pascal这样声明部分不会产生代码的语言，我们也许会在语句列表开始的时候才生成该元组。我们比较偏爱这种延迟，因为它可以避免嵌套过程之间出现STARTSUBPROG-ENDSUBPROG的嵌套对。我们的代码生成器要么必须移动元组来取消这种嵌套并使过程体保持连续，要么插入跳转语句引导子程序体的执行跳过嵌套在其中的任何子程序。

```
check_proc_id(<id>)
{
   Check that the identifier on top of the semantic
   stack is the name of current_proc
}


end_proc_body(procedure)
{
   Take the appropriate actions to remove
       current_proc.local_decls as the current scope.
   Call destroy(current_proc.local_decls), since any
       names declared within the subprogram must no
       longer be accessible.
   generate(ENDSUBPROG, current_proc.activation_rec_size,
            "", "");
   generate(LABEL, procedure.subprogram.end_label, "", "");
   current_proc = procedure.subprogram.old_current_proc
}
```

13.1.2 调用无参过程

无参过程的调用由可用作语句的过程标识符组成，如下面的产生式所示：

```
<statement> → <id> #simple_proc_stmt
```

语义例程simple_proc_stmt()首先必须核实<id>确实代表一个过程。然后它使用该标识符的attributes记录中的其他信息生成两个元组。STARTCALL必须最终被翻译成分配被调过程活动记录的代码，PROCJUMP确定到此过程的实际的控制转移。在讨论带参子程序调用的13.3节里，这两个元组分别由不同的语义例程产生，在它们之间将生成用来处理参数的元组。

487 / 488

```
simple_proc_stmt(<id>)
{
   Search for <id> in the symbol table and obtain
```

```
        a reference to its Attributes, A
    Check that A.class == SUBPROGRAMNAME &&
              A.return_type == NULL
    T = get_temporary()
    generate(STARTCALL, T, A.activation_rec_size, "");
    generate(PROCJUMP, A.start_label, T, "");
}
```

13.2 向子程序传递参数

一般而言，我们在子程序的活动记录中分配空间来存放有关参数的信息。通常，调用者在参数位置上存放某些信息（如值、地址或内情向量），而被调过程使用这些位置来访问实际参数。什么样的信息应该被传递依赖于实参的类型和参数传递的模式。

根据所需的实现技术的相似程度，可以将参数传递模式分组列表如下：

- 值（拷贝）、结果和值-结果。

  实参的值被拷贝到形参中，或形参的最终值的拷贝被拷贝回实参。

- 引用（**var**）和只读（**in**）

  引用参数代表着实参的地址。对引用模式形参的改变将立即影响（改变）相应的实参。只读参数只可以读取而不能被改变。它们经常像引用参数那样通过传递地址来实现（作为选择，它们也可以通过传递拷贝来实现）。

- 名字、形式过程和标号参数

  名字参数，如果需要的话，可在每次引用时重新计算。形式过程是作为参数传递给其他过程的子程序。标号参数在事实上允许间接**goto**语句。

Ada语言的参数传递模式**in**、**out**和**in out**大致对应于只读、结果和值-结果。一个**in**参数可用作局部常量，其值由相应的实参提供。一个**out**参数可用作未初始化的局部变量，其值在子程序返回时被传送给相应的实参。一个**in out**参数除了在子程序调用时其值可由实参值初始化以外，其余都很像一个**out**参数。Ada语言定义要求，对标量而言，这三种模式可以分别采用值、结果和值-结果方式来实现。
489 对于非标量，可以通过实际拷贝参数或传递地址并实现为引用参数等方式来实现这些模式。然而，不论形参/实参对应如何实现，**in**参数必须被当作只读参数来对待。任何依赖于非标量参数模式实现的Ada程序都是错误的。

考虑在Ada/CS中出现的类型种类：

- 标量类型（整型、实型等）
- 数组（可能带动态边界）
- 字符串
- 记录

在实现数据对象（常量、变量和参数）时，必须区分在编译时一个值是如何引用的。它可以是：

- 值——例如，一个显然的常量。
- 值的地址（普通变量）。
- 包含某个值地址的单元的地址（引用参数）。

我们还必须记录一个值是如何被访问的，也就是说，它是否能被改变或仅允许读访问。

我们将引用和访问模式合起来称为访问数据（access data）。在我们的语义记录数据描述符中弄清楚这些访问数据对生成正确的代码十分重要。它也使编译变得容易，例如，若不需要做常量折叠（如，对于字符串常量），我们可以将常量作为只读变量对待。

访问数据也提醒我们在处理数据对象时要仔细。例如，一个引用参数通过引用方式再次传递给第二

个过程时，要求在调用点的代码和普通变量（或值参）通过引用方式传递时的代码不同。我们只想使用一层间接，即使是在引用参数被再次当作一个引用参数而传递的时候，这是因为，第二次调用的子程序通常不知道它的任何实参所需的准确的间接层次。

13.2.1 值、结果和值-结果参数

在值模式下，我们拷贝实参的值并把它看作局部变量。任何具有正确类型的表达式或值均可被传递。

在结果模式下，将创建一个局部的未初始化的变量。在返回时，该变量的值将被赋给实参。只有名字（即，可被用作赋值目标的表达式，有时也称为*左值*（l-value））可作为结果参数被传递，以便该赋值是合法的。

在值-结果模式下，我们将实参的值复制到一个局部拷贝中，在返回时，将其拷贝回实参中。再次地，要求用一个名字作为对应的实参，以便赋值是合法的。

要求拷贝的参数模式的一种实现方法是，将拷贝每个值的代码放在被调过程里。调用者像在引用模式中那样传递地址，位于过程的首部或尾部的代码将实际完成这些拷贝。在结果和值-结果模式下，需要两个截然不同的局部数据对象：实参的地址和参数值的局部拷贝。对值模式而言，我们可以用其中的一个对象。那个拷贝值可以重写（回）被传递的地址。也有可能让调用过程做拷贝工作。对结果或值-结果模式而言，活动记录必须被重新编排，以便调用者而不是被调者从栈中清除实参。 490

下面，我们依次考虑各种类型：

- 对标量来说，我们通常传递实参的地址并将一个值复制到或复制出局部拷贝，尽管对值模式而言，我们通常只是简单地传递实参值本身，因为这样做不会比仅传递地址的开销更多。针对在过程返回时两个涉及将值拷贝出来的情况，我们必须注意实参类型是形参类型的限制子类型的可能性。在这种情况下，我们必须确保送给实参的那个拷贝回来的值是其子类型的一个合法值。这要么涉及在被调过程中对与其地址一起传递的范围信息进行检查；要么在返回后，在调用点立即进行相关检查。
- 对数组来说，我们将（固定大小的）内情向量作为实参传递。通常，这个描述符只是指向数组数据的指针，数组的类型由它的类型描述符来描述，该描述符要么在编译时已完全知晓（对常量边界的数组而言），要么其位置（静态层次和在该活动记录中的偏移）在编译时可知（对动态边界数组而言）。对于*未约束或适应性数组*——即，不完全确定的形式数组参数（通常，数组维数、基类型和下标类型已明确地指出，但边界范围未确定）——除了传递一个指向数组数据空间的指针，还必须传递一个指向含有实参边界的类型描述符的指针。一个过程在被调用时，使用有关的边界信息为这个数组参数的局部拷贝分配空间；然后，此过程拷贝该数组（如果需要），并初始化它自己的局部内情向量。在过程返回时，如果需要，可以使用这两个内情向量将局部数组拷贝回实参。然而，对值模式而言，仅需要单个的内情向量；它可被局部拷贝的内情向量重写。
- 大小可动态变化的字符串需要一个描述符。在Ada/CS中，这样的描述符就是串的地址，而串的大小是作为串本身的一部分而存放的。因为Ada/CS中串的这种约束特征，所以这种简单方法其实是可行的。为支持更一般的字符串特性（例如允许子串作为结果参数），一个描述符除了其地址以外还必须包含串长，如图13-1所示。

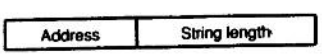


图13-1 字符串参数的描述符

对于值模式，我们可以传递描述符本身。对于结果和值-结果模式，我们可以传递描述符的地址，因为字符串的赋值可以改变描述符的内容。使用描述符，我们可以和前一种情况一样，将 491

值复制到局部拷贝或从局部拷贝中复制出来。

作为一种优化，我们可以在返回时将局部描述符（不是字符串本身）拷贝回实参。字符串的局部拷贝通常在返回时即消失。（不能对数组做此种优化，因为分配在活动记录中的局部数组的所有空间在返回时将被弹出栈。）

- 记录将按标量方式处理，因为它们也是大小固定的，但通常更好的方法是在值模式下传递记录的地址并让被调者拷贝该记录。这种技术使得调用更加紧凑，尤其是在机器没有多字（multiword）传送指令的时候。即使是动态记录（即那些带有一个动态类型的域的记录）也不需要传递显式的类型描述符；运行时的类型描述符，如果有的话，将被放在某个编译时已知且被调过程可见的位置上。

13.2.2 引用和只读参数

通常，我们传递引用或只读参数的地址：

- 对标量值而言，其引用模式的处理是在参数表中传递实参的地址。在只读模式下，我们也可以传递地址，但有时当实参是临时变量（例如，在F(A+B)中）时，这种办法就显得不是很方便。更进一步地，拷贝标量的值并不比传递它的地址开销更多，而且直接访问比间接访问要快。将只读标量参数实现为只读局部变量，可能是最好的办法了。事实上，Ada语言要求通过做实际的拷贝来实现只读的标量**in**参数。
- 对数组而言，我们传递内情向量的地址，或更简单地传递内情向量本身。如果数组没有内情向量，我们将创建一个内情向量。（作为一种优化，除了未约束数组以外，我们也许会忽略其他所有数组的内情向量。）
- 字符串的处理通过传递串描述符的地址来实现。
- 我们传递记录的地址。再次重申，即使是动态记录也不需要显式的类型描述符。

492

13.2.3 处理参数声明的语义例程

Ada/CS语言和Ada一样允许参数的传递采用任意三种模式之一：**in**、**in out**和**out**，它们在13.2节中已被定义过。在过程体中，访问参数和访问变量的方式一样；因此，必须把它们添加到符号表中。在每个参数的符号表条目中必须给出它的传递模式以便生成访问它的合适的代码。我们还必须将所有参数的属性记录按它们声明的次序链接在一起，以便在处理子程序的调用中使用它们。为此，参数名字的属性定义如下：

```
/* class == PARAMNAME */
struct {
    level_range param_level;
    address_range param_offset;
    enum parameter_mode mode;
    attributes *next_param;
};
```

其中，

```
enum parameter_mode { INMODE, OUTMODE, INOUTMODE };
```

参数声明的产生式如下：

```
<formal part>           → ( <parameter declaration>
                              { ; <parameter declaration> } )
<parameter declaration> → <id list> : <mode> <type or subtype> #param_decl
<mode>                  → [ in ] #set_in
                        → out #set_out
                        → in out #set_in_out
```

我们需要一个新的语义记录选项来保存当前处理的参数的传递模式:

```
struct mode {
    enum parameter_mode mode_kind;
};
```

用来处理模式规范的语义例程是相当简单的:

```
set_in(void) => <mode>
{
    <mode> ← (mode) { .mode_kind = INMODE; }
}
set_out(void) => <mode>
{
    <mode> ← (mode) { .mode_kind = OUTMODE; }
}

set_in_out(void) => <mode>
{
    <mode> ← (mode) { .mode_kind = INOUTMODE; }
}


param_decl(<id list>, <mode>, <type or subtype>)
{
    for (each identifier in <id list>) {
        Call enter() to put the identifier in the symbol
            table referenced by current_proc.local_decls.
        if (it is already there) {
            generate an appropriate error message
            continue;  /* go on to next identifier */
        }
        Allocate storage for the parameter, recording
        its offset in the local variable offset.  (The
        size of the block of storage to allocate is
        calculated according to how parameters of
        <type or subtype>.type_ref.object_type and <mode>
        are implemented.)

        The following expression describes the attribute
        record to be created for the parameter:
        (attributes) {
            .class = PARAMNAME;
            .id_type = <type or subtype>.type_ref.object_type;
            .id = the id_entry returned by enter();
            .param_level = current_proc.nesting_level;
            .param_offset = offset;
            .mode = <mode>.mode.mode_kind;
            .next_param = NULL; }

        In addition to associating this attribute record
        with the identifier just entered into the symbol
        table, also add it to the end of the list
        referenced by current_proc.parameters
        (constructed using next_param fields).
    }
}
```

493

494

需要修改语义例程new_name()以便允许将参数按变量方式来解释。PARAMNAME作为必须处理的另一种选择被添入以便在这种情况下生成一个DATAOBJECT记录。address记录addr中的var_level和var_offset域从param_level和param_offset中取值。根据参数类型及其模式所选择的实现方式，indirect和read_only标志被设置为合适的值。例如，如果参数模式是INMODE，PARAMNAME将总是设置read_only标志为TRUE。indirect的值取决于将实参用地址还是拷贝来表示。

因为子程序的参数列表是由每个参数的属性记录的链接表来表示，所以在调用例程end_subprog_body()释放过程的符号表的时候必须要特别注意。这些属性记录不应该在这个时候就释放掉，因为在处理过程调用的时候还必须使用它们所包含的那些信息。我们必须在从符号表中删除它们

之前拷贝它们，或在调用符号表例程destroy()前以某种其他方式对它们加以保护。

13.3 处理子程序调用和参数表

正如我们在第11章中所讨论的，Ada和Ada/CS中描述子程序调用的语法和数组引用的语法是相同的。这两种不同的特性必须通过语义例程name_plus_list()来加以区别。然而，对于本节中那些处理子程序调用的语义例程来说，它们的调用好像是由语法直接触发的，这同Pascal中的情况一样。出于讨论的目的，我们同时简化了子程序名字的语法，仅允许简单的标识符。我们所提出的语义例程仅适用于过程而非函数；当然，为处理函数而对它们进行的扩展也应该是显而易见的。

过程调用的产生式如下：

```
<statement>   → <proc id> [ <parameters> ] #gen_proc_jump
<proc id>     → <id> #start_proc_stmt
<parameters>  → ( <expression> #process_param
                  { , <expression> #process_param } )
```

这里需要一个新的语义栈记录来保存在处理参数的时候（如果有的话）那个被调用的子程序的有关信息：

```
struct proc_call {
    string start_label;
    attributes *parameters;
    address AR_ref;
};
```

495

start_proc_stmt()被调用来处理一个ID记录，将其解释为过程名并产生相应的PROCCALL记录。AR_ref域保存正被调用的子程序的活动记录的引用。在处理过程调用时，用它来填写参数信息。

```
start_proc_stmt(<id>) => <proc id>
{
   Search for <id> in the symbol table and obtain
      a reference to its attributes, A
   Check that A.class == SUBPROGRAMNAME
   T = get_temporary()
   <proc id> ← (proc_call) {
      .start_label = A.start_label;
      .parameters = A.parameters;
      .AR_ref = T; }
   generate(STARTCALL, T, A.activation_rec_size, "");
}
```

STARTCALL限定过程调用序列的开始，因此它允许函数调用可以作为实参表达式的全部或一部分出现。在例程process_param()检查实参和相应的形参类型是兼容的之后，它为每个传递给子程序的参数生成合适的元组：

```
process_param(<proc id>, <expression>) => <proc id>
{
   Verify that the <expression>.data_object.object_type
      matches the type of the first parameter on the list
      <proc id>.proc_call.parameters (Generate a
      "Too many actual parameters" error message if this
      parameter list pointer is NULL.)
   If the parameter mode requires an L-value actual
      parameter (OUTMODE or INOUTMODE), verify that the
      actual parameter describes a legal L-value
      (constants, expressions, and read-only variables
      not allowed)
   Set op to REFPARAM, COPYIN, COPYOUT or COPYINOUT,
      depending on how the parameter is to be transmitted.
   generate(op, <expression>.data_object,
            <proc id>.proc_call.parameters.param_offset,
            <proc id>.proc_call.AR_ref);
   <proc id>.proc_call.parameters =
            <proc id>.proc_call.parameters.next_param
   <proc id> ← the updated struct proc_call
}
```

496

gen_proc_jump()检查是否已提供了足够的参数，然后生成PROCJUMP元组，该元组确定到被调过程的控制转移：

```
gen_proc_jump(<proc id>)
{
   if (<proc id>.proc_call.parameters != NULL)
      Generate a "Too few actual parameters" error message.

   generate(PROCJUMP, <proc id>.proc_call.start_label,
                      <proc id>.proc_call.AR_ref, "");
}
```

示例：带参数的过程调用

假设一个Ada/CS程序包含以下声明：

```
I, J, K: Integer;
procedure P (X : in Integer; Y : in out Integer) is
 begin
  Y := X * J;
 end P;
```

假设P的嵌套层次为2且CONTROLSIZE是3，因此，X的偏移就是3，Y的偏移就是4，并且P的activation_rec_size是5。针对该过程声明将生成如下元组（产生这些元组的语义例程的名字也一并给出）：

```
start_proc_body():    (STARTSUBPROG, 2)
eval_binary():        (MULTI, X, J, t1)
assign():             (ASSIGN, t1, INTEGERSIZE, Y)
end_subprog_body():   (ENDSUBPROG, 5)
                        -- 5 is activation_rec_size for P
```

过程调用P(I+J,K)所生成的元组可能是：

```
start_proc_stmt():    (STARTCALL, t2, P.activation_rec_size)
eval_binary():        (ADDI, I, J, t3)
process_param():      (COPYIN, t3, 4, t2)
                        -- —— 第一个参数的偏移是4
process_param():      (COPYINOUT, K, 5, t2)
                        —— 第二个参数的偏移是5
gen_proc_jump():      (PROCJUMP, P.start_label, t2)
```

497

13.4 子程序调用

在本节里，我们将查看另外一些有关子程序调用的实现相关的细节，这些细节是从诸如STARTSUBPROG和ENDSUBPROG这样的元组中生成代码的时候所必需的。

13.4.1 保存和恢复寄存器

子程序使用的寄存器在穿越该子程序中其他的调用时必须要加以保存，因为被调用的其他子程序可以使用同样的寄存器而因此覆盖它们原来包含的任何值。选择用来保存寄存器的方法将影响正常过程的调用和返回以及处理非局部**goto**或异常的时空代价。某些RISC体系结构将寄存器自动保存作为过程调用指令的一部分，但大多数体系结构要求显式的保存和恢复指令。我们面临的一个抉择是，究竟选择那些调用者正在使用的寄存器（通常，仅有少数几个）、被调者可能修改的寄存器（经常有很多），还是所有的寄存器用于保存与恢复。我们面临的另一个抉择是，究竟选择调用者还是被调者来进行寄存器的保存和恢复。假设过程在不止一个地方被调用，那么让被调者执行这些操作将会使代码空间效率更高。

我们将讨论在图13-2中列出的6种方法。如果其他的情况彼此相当的话，我们可能比较偏爱左手边的列（它们保存很少的寄存器）和第一行中的栏目（不需要为每个调用复制代码）。然而，其他的情况

并非彼此相当。无论选择哪一种方法加以实现，我们必须保证过程调用和返回都是快速的（它们经常发生，因此效率是很重要的），而且实现的方式可以支持非局部**goto**和异常的传播（如果需要的话）。

	调用者的寄存器	被调者的寄存器	所有的寄存器
被调者保存	1	3	5
调用者保存	2	4	6

图13-2 可供选择的寄存器保存方法

(1) 对于被调者保存调用者的寄存器，调用指令必须包括有关正在使用的寄存器的描述（或者是位向量，也称位图；或者是字节向量）。编译器可以在处理过程调用点的地方创建其中的一种。因为测试位向量中的每一位较复杂，所以这种方法仅在有保存/恢复寄存器的硬件指令时才有用，这样的硬件指令取位向量和寄存器保存区的地址来完成所有的工作。目前某些机器中存在着这样的指令。字节向量在
498 某种程度上可以被更有效地访问，但如果仅用于保存所有寄存器（方法5），它还可以更快一些。

(2) 对调用者来说，保存它自己的寄存器并在过程返回时恢复它们是件很简单的事。然而，如果一个过程，它返回的位置不是跟在调用后面的代码（例如，通过**goto**或传播异常返回的），那么它就必须通过寻找恢复代码（或许通过返回地址）并勉强地执行该代码恢复寄存器。

(3) 编译器知道被调者用于非易变临时变量的寄存器——但这仅在过程已完全翻译且代码已生成的时候。因为寄存器必须在过程开始时加以保存，所以在过程的开始代码中可能有一个待回填的跳转可跳到代码最后，那里存放过程保存代码，且跟随其后的是跳到实际过程代码开始的跳转语句。在VAX机器上，编译器仅回填过程的第一个字单元以包含过程使用的寄存器的位向量。过程调用指令使用那个位图来保存合适的寄存器。在活动记录中也放置了该位图的一个拷贝。过程返回指令再次使用该位图来恢复寄存器。

(4) 调用者知道被调者打算使用哪些寄存器的惟一方法是采用位图技术，就像在VAX机器上所用的那样。位图仅在有硬件支持时才有效。VAX上的方法可被认为是符合方法3或方法4的。纯粹的方法4技术可能会使用在方法1中提及的保存/恢复寄存器的指令。仅保存被调者所需的寄存器将使异常传播和非局部**goto**复杂化；动态链簇中的每个活动记录都必须被检查，而且保存在记录中的寄存器值也必须被恢复。因此，AR必须不仅要指明那些旧值，还要指明那些已被保存的寄存器。许多FORTRAN的实现使用方法3。在这些实现中，非局部**goto**没有造成什么特别的问题，因为在FORTRAN中不允许它们存在。相反地，可以使用标号参数，但也只允许一层的返回。也就是说，你可以传递一个标号常量，而不是一个标号参数。

(5) 对被调者来说，保存所有的寄存器很容易，特别是当它们只有几个时（如在PDP-11机器上）。PDP-11的原始C编译器生成过程的开头和结尾代码以保存三个非易变的通用寄存器。如果寄存器是可寻址的，那么这个代码可能会使用循环，但一个展开的版本将更有效率。（实际上，C语言生成一个简短的实用例程调用来执行实际的保存和恢复，这种技术以最小的时间开销来节约代码空间。）

(6) 调用者也可以很容易地保存所有的寄存器。由于调用在大多数程序中相当常见，因此在这里也
499 很可能优先使用一个实用例程来节省代码空间。当所有的寄存器被保存后，非局部**goto**和传播的异常仅需要恢复原来的寄存器集。

13.4.2 子程序的入口和出口

现在，我们考虑子程序调用和子程序返回的机制。如果使用块级的活动记录，那么这里的例程同样适用于块的入口和出口。首先，我们给出一个方法5的算法。活动记录必须包含存放过程调用信息的位置。它的一般形式如图13-3所示。

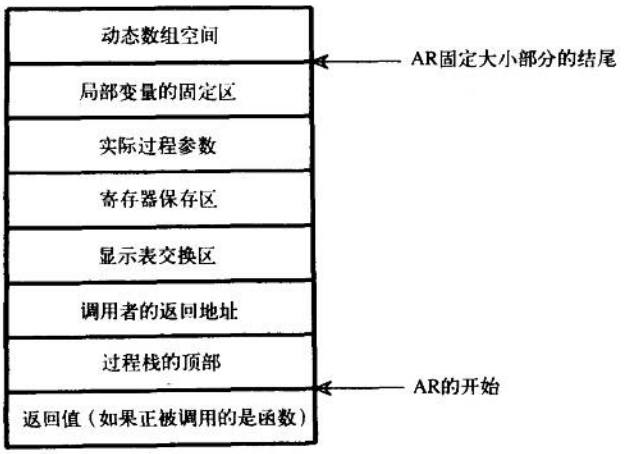


图13-3 活动记录布局示意图

假设我们正在调用一个在词法层i中声明的过程。在过程被调用前，调用者：

- 压入空间以保存：返回值（如果正被调用的是函数）、过程的stack_top值、返回地址、显示表交换区和寄存器保存区。
- 计算每个实参并将它们压入栈。
- 重置调用者的局部stack_top指向正在建立的新的活动记录底部。（函数返回值因此可以形象地表示在被调用的例程的AR的下面（below）。然而，如果操作系统中断在调用过程中发生，最好不要在由硬件栈寄存器定义的栈底之上留有任何重要信息，否则，中断将破坏该信息。）
- 把这个已更新的stack_top值放在某个已知的全局通信位置（如S）中。
- 设置返回地址并调用该过程。

500

在被调用后，该过程即被调者（callee）：

- 保存当前寄存器保存区的寄存器值（不包括显示表寄存器，它们被保存在显示表交换区中）。
- 从全局的S处（S指向活动记录的开始位置）和已知的过程活动记录的固定大小来计算它的局部stack_top。
- 在显示表交换区中保存display[i]的当前值，其中i是过程的词法层次。
- 设置display[i]以指向其活动记录的开始位置（保存在S中）。
- 如果需要的话，使用内情向量和作为实参传入的地址来按值拷贝数组和记录。
- 如果需要的话，使用局部stack_top来分配动态数组。

在过程要退出时，它：

- 从寄存器保存区恢复先前寄存器的值。
- 从显示表交换区恢复先前display[i]的值。
- 跳到调用者的返回地址。

函数的返回值位于调用者栈的顶部。如果使用了块级分配方式，那么在每个块入口和出口处，和过程调用/返回情况一样，需压入和弹出活动记录并更新和恢复显示表寄存器。

更常见的是，即使在有局部块时，也可以使用过程级的分配方式。在这种情况下，在块的入口和出口处需要以下动作：

块入口

- 从外围块或过程中拷贝stack_top的值到这个块的局部stack_top中。

对于每一个在块中分配的动态数组，

(a) 计算数组的边界和所需空间。

(b) 在栈上压入所需空间并更新局部stack_top。

(c) 更新数组的内情向量。

块出口

• 不需要做任何事!

块的出口不需要做任何工作，因为在其退出后它的外围块的局部stack_top就会变成活跃的。这种
501 方案还可以很容易地做到：（通过**goto**或**exit**语句）立刻离开许多块。

本节中使用的过程调用模型假设：被调者保存并稍后恢复所有的寄存器。选择调用者还是被调者来保存和恢复寄存器以及保存哪些寄存器均受到代码大小、速度和复杂性等因素的影响。前面的模型随时可以适应所选择的任何保存/恢复机制。

使用该模型，调用者必须分配大部分活动记录，因为实参是由调用者计算的。另一种可选的组织形式将实参放在AR下面属于调用者的部分栈里。这种方法在为过程调用及相关的AR栈操作提供硬件（或微代码）支持的机器上很常见。例如，VAX机器的指令集包括一个能完成大多数过程入口和出口工作的功能强大的过程调用指令。该指令提供额外的指针指向AR，以便被调者不需要知道已被保存的寄存器占用多少空间。这种AR的布局如图13-4所示。

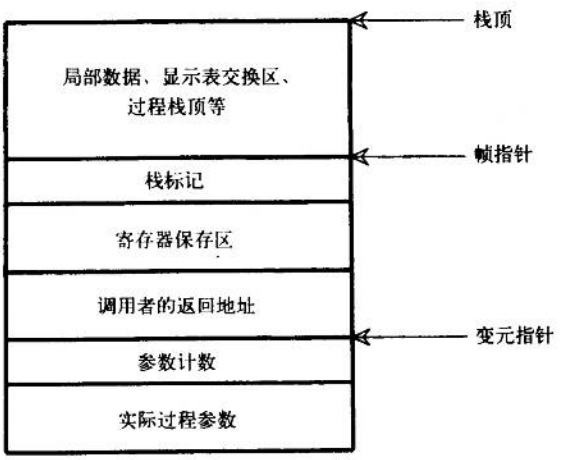


图13-4 VAX机器上活动记录的布局图

栈顶、帧指针和变元指针均为特殊的硬件寄存器。帧（frame）是活动记录的另一个称谓。调用者将实参或变元（argument）压入栈中并发出命令calls argcount,procedure（calls中的s代表栈（stack））。过程的入口点是一个位图，它指出过程使用了哪些寄存器。calls指令压入argcount和返回地址并保存由位图指示的寄存器以及变元指针和帧指针。接着它压入栈标记，那里记录着需保存的寄存器以及另外的一些信息（例如条件码）。最后，它设置变元指针和帧指针并跳转到跟在位图后面的那个字单元
502 来开始子程序的执行。过程可以使用帧指针访问局部变量以及使用变元指针访问变元（形参）。

过程可以通过执行零-操作数指令ret来返回。该指令恢复由栈标记指示的寄存器以及条件码，恢复帧指针和变元指针到它们原来的值，并将stack_top恰好设置在实参的下面（由变元指针和变元计数所指示），这样就从栈中弹出了变元（形参）。

不幸的是，这种机制不直接支持返回值和块结构（嵌套的名字作用域）。返回值可以在寄存器中被传递或通过结果参数传递回来。对于标量值，我们经常使用寄存器的方法，如在VAX程序中，寄存器R0

和R1常常用作此目的。当使用结果参数时，这个返回参数应当位于栈的第一个（最低的）位置，且不能被包括在变元计数内，因此，返回时它不会从栈中弹出。块结构可以通过显示表或静态链簇来处理，但每个AR指针必须被表示为一个值对（pair）：变元指针和帧指针。

如果过程可以像在Algol 60或Pascal语言中那样作为参数传递，那么上述方法需要做少许的扩展。Ada和Ada/CS没有这些顾虑。这样的过程，称为*形式过程*（formal procedure），我们已在9.6节中进行了讨论。

VAX机器支持的方案和本节中介绍的模型在`stack_top`指针的处理上区别很大。VAX方案支持全局指针，而这里介绍的方案，针对每个过程级的活动记录以及针对任何局部块，均有一个局部指针。不管有无硬件支持，这两种方法都可以实际工作。它们的主要差别是：全局指针方法要求在过程或块退出时恢复旧的指针值；而使用局部指针则不需要相应的操作，更确切的是，在退出发生时，某个合适的局部指针将隐式地成为活跃的`stack_top`。

13.5 标号参数

某些语言（例如FORTRAN和Algol 60）允许将语句标号作为参数传递。在一个不包括异常的语言中可以使用跳转到标号参数的**goto**来实现从过程返回的另一种选择，就像Ada所做的那样。令人惊讶的是，标号参数并不特别难实现。在实际标号L被绑定为标号参数时，我们创建一个小的形式过程（它的过程体是**goto** L），并和往常一样将它与当前环境绑定（这和我们为所有形式过程做的一样）。现在，我们只使用形式过程机制。当要跳转到形式标号参数时，我们就执行该过程，它可以恢复正确的环境并做实际的跳转。

如果使用静态链簇，则标号参数可以传递为两个指针：（1）由标号定义的代码位置（就是那个小的形式过程的**goto** L过程体）和（2）标号的词法层次的静态指针。执行这个**goto**意味着将按那个静态指针所指示的来重置当前活动记录并跳转到特定的位置。我们还必须确保有关寄存器已被恢复，但这些会使事情变得更加复杂。

503

我们可以通过图13-5中较为复杂的程序来说明使用静态链簇对过程和标号参数所进行的合适的处理（该程序摘自Pratt [1975, p. 225]）。我们修改了该程序，使它可以采用Ada式的语法，其中语句标号L被标识为<<L>>。行号将在下面的讨论中引用。

```
1   procedure B is
2      N : Integer;
3
4      procedure P(X : procedure; C : Integer) is
5         procedure R(label T) is
6         begin
7            N := N + C;
8            X(K);
9            goto T;
10        end R;
11     begin -- P
12        <<J>> if C > N then
13           X(J);
14        else
15           P(R,C+1);
16        end if;
17        <<K>> N := N + C;
18        goto L;
19     end P;
20
21     procedure Q(label T) is
```

图13-5 标号参数示例

```
22      begin
23          N := N + 1;
24          goto T;
25      end Q;
26
27  begin -- B
28      N := 2;
29      P(Q,2);
30      <<L>>   write(N);
31  end B;
```

图13-5 （续）

图13-6显示了在第24行执行前、在起初的一系列调用之后运行时栈上每个活动记录的主要内容。所有形式标号和形式过程均由两个数据项来指定：（1）地址（图中用实际过程的名字或标号来表示）和
504 （2）静态链（一个活动记录）。由AR5所表示的最近一次调用位于栈顶。对于AR1中变量N，图中包括了它所经历的所有值以及在它获得每个值时哪个活动记录位于运行时栈顶。

AR5 (Q):

static link	AR1
T	K, AR2
return	AR4, line 9

AR4 (R):

static link	AR2
T	J, AR3
return	AR3, line 17

AR3 (P):

static link	AR1
X	R, AR2
C	3
return	AR2, line 17

AR2 (P):

static link	AR1
X	Q, AR1
C	2
return	AR1, line 30

AR1 (B):

static link	none
N	5 (AR5) 4 (AR4) 2 (AR1)
return	none

图13-6 第一次**goto**执行前的活动记录

一旦到达图13-6中所描述的状态时，过程Q执行第24行的**goto** T。T在AR2中被绑定到K，因此活动记录AR5、AR4和AR3被弹出。在标号为K的语句执行后，剩下的运行时栈状态如图13-7所示。

505 接下来执行语句**goto** L。L是全局一层的，因此弹出AR2并在块B退出前打印N的值7。

AR2 (P):

static link	AR1
X	Q, AR1
C	2
return	AR1, line 30

AR1 (B):

static link	none
N	7 (AR2) 5 (AR5) 4 (AR4) 2 (AR1)
return	none

图13-7 第一次**goto**执行后的活动记录

13.6 名字参数

由Algol 60引入的*按名调用*（call by name）在每次调用中把参数传递模拟为用实参系统地替换形参。也就是说，在每次调用时，我们假装被调用的过程体是用实参替换形参而得到的一个宏展开。这在概念上看似简单，但其实很复杂，因为实参必须在调用者的环境中而不是在被调者的环境中被计算（我们仅编译过程体一次）。此外，实际的按名参数在每次引用时都必须（在调用者环境中而不是被调者环境中）被重新计算。

因为实现按名参数的复杂性，所以它们直到今天仍被视为某种非正规的方法。然而，这种模式作为一种练习来明白参数传递中的问题还是值得研究的。

对于不需要计算代码的参数（例如简单变量或常量），我们可以只传递其地址。然而，如果需要代码来计算实参，那么编译器需要将此代码封装到一个通常称之为*形实转换程序*（thunk）的内部生成的过程中。然后，这个过程连同它的环境作为实参一起被传递，使用的方法和形式过程一样。根据形参访问的上下文，也许要用这个转换程序计算地址（左值l-value）或右值(r-value)。为确保不会赋值到右值表达式，按名传递的实参表达式的转换程序必须进行有关检查，在要求*左值*却出现右值的时候，必须产生一个运行时错误。只读名字模式也可用类似的方式来实现，尽管这在实践中并不常见。 506

为看一下这些不寻常的名字模式的效果，请考虑：

```
declare
  I : Integer;
  A : array(1..2) of Integer;
  procedure P (J : name Integer) is
  begin
    Read (J);
    I := I+1;
    Read (J);
  end P;
begin
  I := 1;
  P(A(I));
end;
```

这个程序读数据到变量A(1)和A(2)，而其他的模式则将J永久绑定到A(1)。

称为*Jensen' s device*的程序说明了名字模式的一个实际的使用：

```
function Sum(Expr : name Real; Index : name Integer; Max : Integer)
return  Integer is
  Answer : Real := 0;
begin
  for i in 1 .. Max loop
    Index := i;
    Answer := Answer + Expr;
  end loop;
  return Answer;
end Sum;
write(Sum(i,i,5));        -- Sum of first five integers
writeln(Sum(j*j,j,10));       -- Sum of first 10 squares
writeln(Sum(log(sin(x/pi)),x,100));     -- integration?
```

修改**for loop**中Index的副作用是每次给Expr一个不同的值。函数Sum从1到Max逐步增加Index并在每一步中重新计算Expr，在Expr改变时Sum对其求和。

一个有益的练习是仅使用名字模式参数编写交换两个整数值的Swap例程。但方案却明显地失败了，这是因为其中一个参数（如第一个参数）首先被赋新值，而这个改变可能影响其他参数的含义。例如，如果i=1且a(1)=3，那么调用Swap(i,a(i))将终止i（为3）的正确设置，它改变的是a(3)而不是a(1)。如果首先给第二个参数赋新值，那么Swap(a(i),i)也会失败。我们需要抓住并保留两个变元的左值，然后再

修改它们的右值。该技巧使用下面例子中的一个附属例程。(然而，对于Swap(A,B[f(i)])，该方案还是会失败，这里f(i)在每次调用时返回一个不同的值。因为每个实参必须计算两次（第一次取它用来的值，第
507 二次存入它的新值），所以一个完全通用的按名调用方案看起来是不可能的。参见[Fleck 1976]。)

```
function GiveSecond(x, y : name Integer) return Integer is
-- Returns original r-value of y, gives y a new r-value from x.
   tmp : Integer;
begin
   tmp := y;
   y := a;
   return tmp;
end GiveSecond;


procedure Swap(a, b :name Integer);
begin
   a := GiveSecond(a,b);
end Swap;
```

练习

1. 解释如何修改13.1.2节中的`simple_proc_stmt()`动作例程和13.3节中的`gen_proc_jump()`例程以便同时处理函数和过程。
2. 一个无参函数的调用看起来很像对一个变量的引用。确定将修改第11章的哪个动作例程以便处理无参函数。描述所做的必要修改。
3. 除了Ada语言以外，列出你选择的语言所定义的参数传递模式。使用已定义的参数模式再加上语言的其他特性来描述如何模拟其他种类的参数传递模式。
4. Pascal的语言标准要求Pascal编译器通过引用实现**var**参数。编写一个Pascal程序，如果**var**参数是通过值-结果方式实现的，那么该程序将产生不同结果。使用任何你可用的Pascal编译器编译并执行该程序以验证编译器是否正确实现了**var**参数。
5. 编写一个Ada程序，它可以发现数组类型**in out**参数的实现。使用该程序去找出任何你可接触到的Ada编译器所使用的实现方式。
6. 给出在13.3节例子中声明的过程P的`attributes`记录的描述，包括过程参数的`attributes`记录列表。
508 7. 给出下面过程声明的`attributes`记录，包括过程参数的`attributes`记录列表。

```
type A is array (1..20) of Integer;
procedure P (X : in Integer; inArray : in A; inOutArray : in out A);
begin
   for I in 1..20 loop
      inOutArray(I) := X * inArray(I);
   end loop;
end P;
```

8. 假定数组参数通过传值或值-结果方式实现，给出为练习7中的过程所生成的元组。使用（level，offset）偶对而不是符号化名字来确定元组中的变量和参数。假设P声明的嵌套层次为2。
9. 假定数组参数通过引用方式来实现，给出为练习7中的过程所生成的元组。使用（level，offset）偶对而不是符号化名字来确定元组中的变量和参数。假设P声明的嵌套层次为2。
10. 假定下面的声明与练习7中的过程P处于相同的层次：

```
A1, A2 : A;
J, K: Integer;
```

其中A1的偏移是5。给出为以下调用生成的元组：

```
P(J*K, A1, A2);
```

假设使用练习8和练习9中参数实现方式。

11. 在13.4.2节中列出的子程序调用和返回步骤均假设被调子程序负责保存所有寄存器。为在13.4.1节中讨论的其他寄存器保存方法重写这些调用和返回序列。
12. 13.5节中那个说明标号参数的例子假设静态链簇可用来引用非局部环境。因此，一个环境仅需要由一个静态链簇描述即可。如果使用的是显示表而非静态链簇，那么什么是描述一个环境所必需的？重新考虑该例子，假设这次使用显示表。
13. 对于13.6节的第一个例子中的过程P的按名调用参数J，详细解释为实现它的每个引用所做的计算。 509

第14章 属性文法和多遍翻译

在第10～13章，我们研究了实现各种程序设计语言特性的技术。我们根据所假设的可被语法分析器直接调用的语义例程来组织并定义这些技术。这些语义例程构成编译器的重要组成部分，因为它们完成编译过程中的分析任务后又要开始综合任务。分析完成于语义例程将（从声明获得的）语义信息和所有标识符的使用联系起来以及检查程序是否满足语言的任何静态语义约束。综合开始于程序中间表示或实际目标代码的生成。语义例程的命名就是来自于此综合步骤，因为这些例程的输出必须反映由词法以及语法分析器所识别的语法结构的“含义”。

不论编译器的组织形式是否可以让语义例程直接被语法分析器调用，在语义例程部分描述的技术通常还是很有用的。本章的第一部分将详细介绍*属性文法*（attribute grammar），它曾在7.1.1节中被简要介 510
绍过。属性文法提供了描述语义处理的实用的形式化表述方式，相比之下，我们在第10～13章中使用的是非正规的伪代码描述。在7.1.2节中所介绍的任何一种编译器结构都可以使用属性文法来描述语义处理。然而，它们强大的描述能力在围绕树结构中间表示而组织的编译器中才能最好地发挥出来。为此，本章的第二个部分将研究树结构的中间表示以及支持其使用的工具。

14.1 属性文法

Knuth（1968）提出的属性文法是一种给语言的上下文无关的语法添加语义的手段。每个文法符号（终结符或非终结符）可以有固定数目的关联值，称为*属性*（attribute）。这些属性代表着与符号相关的信息，诸如它的类型、值、代码序列和符号表等。属性可以在分析输入时计算，或在语法分析器构造完分析树以后再计算。增加了属性的结果分析树代表着输入的语义。

与给定符号关联的属性可以分为两类：*综合*（synthetic）*属性*和*继承*（inherited）*属性*。简单地说，综合属性被用来沿语法树向上传递信息，而继承属性则被用来沿语法树向下传递信息。特别地：

- 终结符可能仅有综合属性。它们由词法分析器提供给终结符。
- 非终结符可以同时拥有综合属性和继承属性。在计算开始前，*开始符号*（start symbol）的所有继承属性以初始值方式提供（其实就是参数）。

每个上下文无关的产生式有关联的属性计算规则。我们必须为产生式右边出现的每个继承属性和产生式左边符号的每个综合属性提供规则。属性规则仅可以使用相应产生式中的符号所关联的属性来计算有关的值。这有助于将属性依赖“包装”在给定的产生式中。然而，产生式左部符号的继承属性和右部的综合属性并不通过给定产生式的属性规则来计算。相反，它们在其他产生式中计算并可作为上述属性计算规则的输入参数。

用于描述属性计算规则的传统记号多来自于Algol和Pascal语言；我们也借用Ada语言的记号。具体来说，:= 用作赋值，= 用作相等，& 用作字符串连接，-- 用作开始注释。

作为属性计算规则的示例，考虑非终结符A、B和C，其中A有继承属性a和综合属性b，B有综合属 511
性c，C有继承属性d。产生式 A → B C可能有如下规则：

```
C.d := B.c + 1;
A.b := A.a + B.c;
```

注意，A.a（A的属性a）和B.c在别的地方计算：

作为一个更具体的示例，考虑在图14-1中的上下文无关文法（CFG），它生成使用运算符+和*的整数常量表达式。每个表达式或子表达式都标记有表示其值的属性。

在图14-1中，某些非终结符的上标可用来区分产生式中同一个非终结符的多次出现。对于文法G1的第一和第三条产生式，这种区分有助于消除属性计算规则中对这些非终结符的二义引用。这些规则和文法中其他所有的规则仅使用综合属性，因为信息流严格地沿语法树自底向上传递。

G1: V_n = {E,T,P} 这些符号中的每一个都有单个的综合属性val

V_t = {C} 这个符号也有一个综合属性val

产生式		属性规则
E^1	$\rightarrow E^2 + T$	$E^1.val := E^2.val + T.val$
E	$\rightarrow T$	E.val := T.val
T^1	$\rightarrow T^2 * P$	$T^1.val := T^2.val * P.val$
T	$\rightarrow P$	T.val := P.val
P	$\rightarrow C$	P.val := C.val
P	$\rightarrow (E)$	P.val := E.val

图14-1 整数常量表达式的属性文法

输入12+3*6生成如图14-2所示的属性（化）语法树（符号后面跟着它们的属性val的值）。

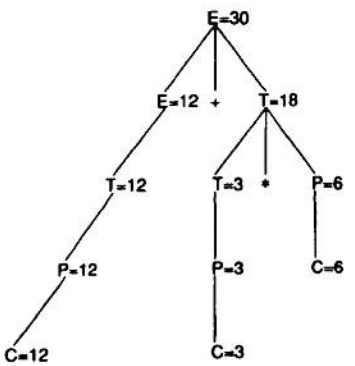


图14-2 12+3*6的属性分析树

如前所述，属性规则和信息流通常都是很简单明了的。尽管如此，属性文法基本上是一种非常强大的形式化描述方法。我们不需要为属性规则做任何假设。它们可能非常复杂，计算代价也相当昂贵，例如，计算过程可能在任何情况下都不会终止。使用这样规则的属性文法也许因此要指明那些非常耗时、甚至有时尚未定义的翻译。

属性规则可能会有副作用（诸如生成代码）或者它们可能不是其输入参数的严格函数（诸如某条规则给出下一个可用的数据地址）。如果没有预先定好属性计算次序（例如，从左到右的次序），规则的使用可能导致二义性结果。

512

因此，虽然属性规则给使用者带来了极大的方便，但也要求在定义和使用它们的时候必须要很仔细。同时，属性规则之间的函数依赖决定了信息在语法树上的流动方式。多数时候，它只是简单地自顶向下或自底向上。但有时属性规则也能导致非常复杂的流动模式。例如，考虑图14-3中的文法G2。

图14-4给出了从文法G2中推导出的惟一的语法树，其中每个结点旁边的括号里的数字标示着属性计算次序。该次序为S.A（以初始值方式提供）、Z.H、Z.G、X.C、X.D、S.B、Y.E、Y.F。

G2 : V_n = {S,X,Y,Z} V_t = {x,y,z}

属性:
S: A{inh}, B{syn}
X: C{inh}, D{syn}
Y: E{inh}, F{syn}
Z: H{inh}, G{syn}

产生式	属性规则	
S → XYZ	Z.H := S.A	{inh←inh}
	X.C := Z.G	{inh←syn}
	S.B := X.D–2	{syn←syn}
	Y.E := S.B	{inh←syn}
X → x	X.D := 2*X.C	{syn←inh}
Y → y	Y.F := Y.E*3	{syn←inh}
Z → z	Z.G := Z.H+1	{syn←inh}

图14-3 属性计算规则中复杂的信息流

这种属性流的编排使属性计算程序非常忙碌。因此，很多属性计算程序限制它们所允许的属性流种类（如从左自右）也就不足为奇了。然而，还存在着更为糟糕的属性流计算次序。如果在G2中，用Z.H := S.B替换规则Z.H := S.A，那么就会出现这样的情况：S.B间接地由其自身来定义（S.B定义Z.H，Z.H定义Z.G，Z.G定义X.C，X.C定义X.D，而X.D最后定义S.B）。在这种情况下存在着定义循环（circularity），因此也就没有合法定义的属性计算次序。

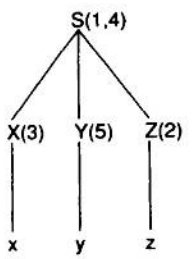


图14-4 文法G2的语法树

我们称有这种定义的属性文法是循环的（circular）并认为是非正常定义的。已知有测试循环性的算法（Jazayeri，Ogden and Rounds 1975），但其运行时花费可能是被测文法大小的指数倍。幸运的是，大多数计算方法被限制为仅接受属性文法的非循环子集（这就像分析技术被限制为接受上下文无关文法的非二义的子集一样）。 513

14.1.1 简单赋值形式和动作符号

在属性计算规则中，将某个属性值或常量赋值给另一个属性是很常见的。我们称这样的规则为拷贝规则（copy rule）。因为属性计算程序自动地处理这样的拷贝规则，所以我们也就常常使用一种称为简单赋值形式（simple assignment form）的属性文法。在这种形式的属性文法中，动作符号（action symbol）常用来实现所有非平凡的属性规则（即，所有非拷贝的属性规则）。给这些动作符号的输入值是它的继承属性，而输出值则是它的综合属性。 514

因此，给定产生式$E^1 \rightarrow E^2 + T$和属性规则E^1.val := E^2.val + T.val（该规则不是简单赋值形式），我们可能会代之以$E^1 \rightarrow E^2 + T$<add>，其中动作符号<add>带有继承属性v1和v2以及综合属性sum。所使用的拷贝规则是：

<add>.v1 := E^2.val
<add>.v2 := T.val
E^1.val := <add>.sum

使用拷贝规则，所有非平凡的属性计算均可被替换成动作符号，而这些动作符号能被实现为子程序（或语义例程）调用。属性值的拷贝很容易自动地进行。

此外，动作符号还可用来加强上下文有关的约束。我们允许动作符号指示语义错误，前提条件是它的输入值（即它的继承属性）是不正确的。这种错误的指示和语法分析错误的指示类似。如果输入值是正确的，那么动作符号将计算其结果（它的综合属性）并指示没有发生错误。

动作符号可用于属性值的检查和计算。正因为如此，它们非常适合于语义例程的抽象表示。我们稍后将看到，除了手工编码的动作符号以外，属性计算程序几乎可以自动处理每一件事。（手工编码可能是也可能不是从具有程序样式的动作符号定义开始的。）

考虑使用继承属性Max的文法G1的修改版本。Max是所允许的常量或常量表达式的最大值。那些要使用较大值的企图都将被视为属性文法中的语义错误。图14-5给出了修改后的文法G3。

G3: V_n = {E,T,P} V_t = {C}, Action Symbols = {<add>,<mult>,<check>}

E、T和P有一个继承属性Max和一个综合属性Val。

<add>和<mult>有继承属性v1、v2和Max。它们均有综合属性Result。

<check>有继承属性Val和Max以及综合属性Result。

产生式	拷贝规则
$E^1 \rightarrow E^2$ + T <add>	<add>.v1 := E^2.Val <add>.v2 := T.Val <add>.Max := E^1.Max E^1.Val := <add>.Result E^2.Max := E^1.Max T.Max := E^1.Max
E → T	E.Val := T.Val T.Max := E.Max
$T^1 \rightarrow T^2$ * P <mult>	<mult>.v1 := T^2.Val <mult>.v2 := P.Val <mult>.Max := T^1.Max T^1.Val := <mult>.Result T^2.Max := T^1.Max P.Max := T^1.Max
T → P	P.Max := T.Max T.Val := P.Val
P → C <check>	<check>.Max := P.Max <check>.Val := C.Val P.Val := <check>.Result
P → (E)	E.Max := P.Max P.Val := E.Val

动作符号定义

<add> : if (v1 + v2 > Max) ERROR else Result := v1 + v2

<mult> : if (v1 * v2 > Max) ERROR else Result := v1 * v2

<check>: if (Val > Max) ERROR else Result := Val

图14-5 属性文法的简单赋值形式

例如，使用由Farrow（1982）和Ganzinger等（1982）所开发的技术，可以将图14-5中的文法G3的完整定义自动地翻译为常量表达式的语法分析器和属性计算程序/检查程序。

作为示例，考虑30*30+125的处理，其中Max = 1000。(这是一个语义上非法的表达式。) 相应的语法树显示在图14-6中。其中，一个未定义值 (=?) 与表达式相关联，这是因为它所计算的值是非法的(因为它大于Max)。

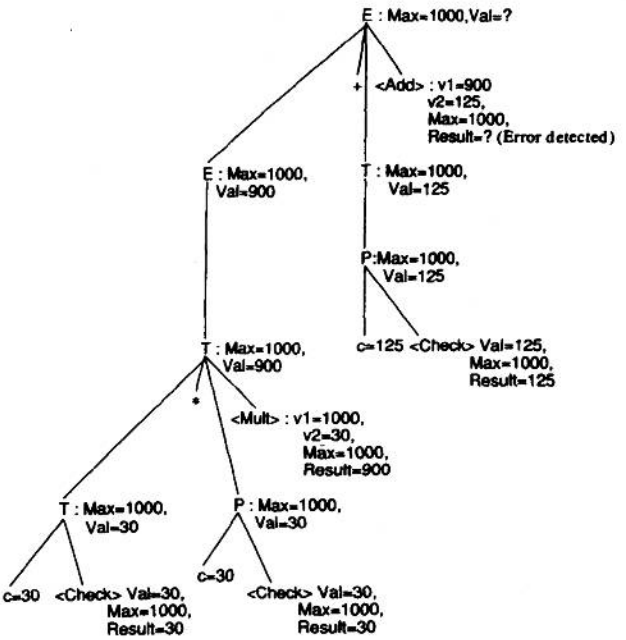


图14-6 30*30+125的属性语法树

14.1.2 树遍历的属性计算程序

本节和下一节将研究各种属性计算算法的有效性及其限制。属性计算算法可分为两类：(1)树遍历算法，这种算法一般需要多次遍历来计算所有的属性（因此需要存在一棵语法树）；(2)“直接”计算算法，该算法在分析程序的同时计算属性值。本节研究树遍历算法，在14.1.3节将研究“直接”计算算法。 515

从左至右遍历方法

现在考虑确定属性值的方式。很多属性计算方法被称为树遍历 (tree-walk) 的计算程序。这些方法假定语法树已被建立起来，并且其中已标记出开始符号的继承属性和所有终结符的综合属性。然后，这些方法将以某种方式遍历语法树直到计算出所有的属性。一个特别常见的遍历次序是深度优先、从左至右遍历。如果需要，可以进行多次（或“多遍”）遍历。 516 ~ 517

以下方法可以计算任何非循环的属性文法：

```
while (attributes remain to be evaluated)
   visit_node(S);    /* S is Start symbol */

void visit_node(node N)
{
   if (N is a nonterminal) {
      /* Assume it roots production
         N →X1 · · · Xm */
      for (i = 1; i <= m; i++) {
         if (! Xi ∈ Vt) {
            /* i.e., a nonterminal or action symbol */
            Evaluate all possible inherited
```

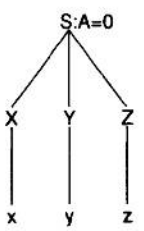
```
                attributes of X_i.
              visit_node(X_i)
            }
        }
    }
    Evaluate all possible synthetic attributes of N
}
```

只要文法是非循环的，那么在每次遍历中将至少计算一个属性。进一步，如果树有n个结点（且因此至多有O(n)个属性），那么最差情况下的时间复杂度为$O(n^2)$（与属性计算时间无关）。该算法甚至可以处理循环文法，只要在每次遍历后查验在刚才进行的那次遍历中是否已经计算了至少一个属性。

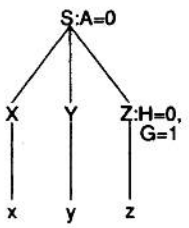
作为示例，重新考虑G2。假设S.A被初始化为0。在计算开始前，其抽象语法树如图14-7a所示。

首次遍历中的动作如下：

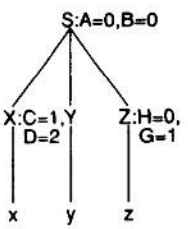
```
visit_node(S)
  X.C can't be evaluated
  visit_node(X)
      X.D can't be evaluated
  Y.E can't be evaluated
  visit_node(Y)
      Y.F can't be evaluated
  Z.H := 0
  visit_node(Z)
      Z.G := 1
  S.B can't be evaluated
```



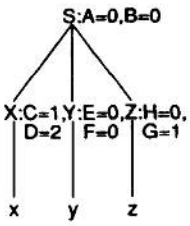
a) 初始状态



b) 第一次调用visit_node()



c) 第二次调用visit_node()



d) 第三次调用visit_node()

图14-7 文法G2中抽象语法树的属性计算过程

在第一遍计算后，语法树的状态如图14-7b所示。第二次调用visit_node(S)导致按X.C、X.D和S.B的次序计算相应的属性值，此时语法树的状态如图14-7c所示。最后，第三次遍历将计算Y的两个属性。语法树的最终状态如图14-7d所示。

518

该算法非常普通，同时也非常粗糙且常常效率不高（我们重复访问了已计算的结点）。只有当我们能证明某些固定的N遍访问即可满足计算需要时（与待计算的实际语法树无关），一种计算算法才算是真正有吸引力。

特别有趣的情况是那种在一遍访问即可满足计算的算法。在从左到右的一遍访问即可使所有属性得到计算的属性文法称为L-属性定义的（L-attributed）。注意，如果底层的CFG是LL(1)的，那么我们可以在分析进行的同时计算有关属性。属性文法是L-属性定义的，当且仅当：

- 右部符号的每一个继承属性仅依赖于左部符号的继承属性和这个给定右部符号左边的那些符号的任意属性。
- 每个左部符号的综合属性仅依赖于它的继承属性和右部符号的任意属性。
- 动作符号的每一个综合属性仅依赖于它的继承属性。

事实上，这些属性流上的限制假设产生式$X \rightarrow Y_1Y_2 \ldots Y_n$中存在如下的属性计算次序： 519

```
Evaluate X's inherited attributes.
Evaluate Y1's inherited attributes.
Call visit_node(Y1) to get Y1's synthetic attributes.
.
.
.
Evaluate Yn's inherited attributes.
Call visit_node(Yn) to get Yn's synthetic attributes.
Evaluate X's synthetic attributes.
```

注意:属性文法G1和G3是L-属性定义的（但G2不是）。

Bochmann(1976)分析了更一般的有关N遍访问什么时候可以满足计算的问题，并提出了一个可确定什么时候N遍访问将总是满足计算的算法。

对某些非循环的属性文法而言，不能为所有的语法树事先固定访问的遍数N。考虑图14-8中的属性文法，它生成a的列表并（使用一些稍有技巧的办法）对a计数。

G4: $V_t = \{a\}$ $V_n = \{L,A\}$

L有综合属性C（代表count），用于统计其子树中a的个数。

A有继承属性RC（代表right count），用于统计其右边a的个数

产生式		属性规则
L^1	$\rightarrow A\ L^2$	$A.RC := L^2.C$ $L^1.C := A.RC + 1$
L	$\rightarrow A$	$A.RC := 0$ $L.C := 1$
A	$\rightarrow a$	

图14-8 带有从右到左信息流的属性规则

图14-9显示了字符串aaa的属性语法树。注意：这里信息流方向是从右到左且需要O(n)遍来计算带有n个a的树（这意味着需要$O(n^2)$的时间）。

我们注意到，有时从右到左的深度优先的树遍历效果较好（G4的树可在从右到左的一遍访问中计算）。而这种观察导致产生从左到右和从右到左两种遍历交替使用的想法（Jazayeri and Walter 1975）。但是，不是所有的非循环属性文法都可以在固定次数的交替遍历中计算。 520

考虑图14-10中的文法G5，它是文法G4的一种推广，也可以对生成的a进行计数。这里，交替的访问遍历在信息流以Z字形穿越树的情况下并不是真的很有用。该信息流如图14-11中箭头所示。

在每遍访问中，信息仅向上传递一层，因此需要O(n)遍来访问n个A的树（再次重申，需要$O(n^2)$的时间）。

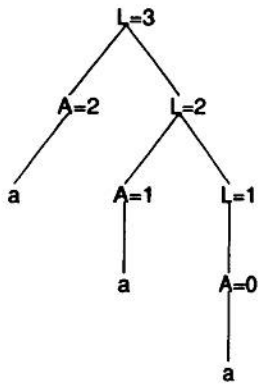


图14-9 文法G4的一个属性语法树

G5: V_t = {a} V_n = {L,L2,A}.

L和L_2有名为C（代表count）的综合属性，用于统计以它们为根的子树中a的个数

A有继承属性BC（代表brother's count），用于统计A的兄弟子树中a的个数

产生式	属性规则
L → A L2	A.BC := L2.C L.C := A.BC+1
L → A	A.BC := 0 L.C := 1
L2 → L A	A.BC := L.C L2.C := A.BC+1
L2 → A	A.BC := 0 L2.C := 1
A → a	

521

图14-10 交替信息流的属性规则

总之，深度优先的遍历（从左到右或从右到左）非常普通，除了L-属性定义外，它们在处理那些已被重复访问过的结点时相当低效，因为那些结点要么不可计算，要么已被全部计算过。

交替遍历方法

正如我们所看到的，从左到右和从右到左遍历方法在处理那些经常被访问的结点时有不足的地方，因为有时并不需要那样做。我们现在考虑一种更具有针对性的访问模式。其关键思想来自于下面的观察：每个非终结符和动作符号至少要被访问一次，但一旦被访问，在它们至少有一个其他的属性可用（即，
522 被计算出来）之前，这些符号暂时不需要被再次访问。

简言之，所有流入子树的信息将穿过它的根结点，因此子树的计算将由根属性的计算提供线索。使用这种想法，可以实质性地改进先前定义的visit_node()例程。假设我们用值State标记语法树上的每个结点。对于结点A，State(A)是表示上次访问A时所计算的A的属性集；NV则用来表示A从未被访问过。所有非终结符和动作符号均有初始状态值NV。所有终结符的初始状态值等于它们的综合属性集（因为所有这些综合属性值由词法扫描器提供）。进一步，我们将使用函数Atr。Atr(A)给出所有当前已计算的A的属性。注意，除了在State(A)=NV时，其他情况下State(A) ⊑ Atr(A) 。

现在可以考虑使用状态信息来智能地指导属性计算程序:

```
void visit_node2(node N)
{
   if (N is an Action Symbol)
      Evaluate all possible attributes in N
   else {      /* N  is a nonterminal */
      while (TRUE) {
         Evaluate all possible attributes in the
            production rooted by N;
         if (there exists an offspring, Xi, of N
               such that State(Xi) != Atr(Xi))
            visit_node2(Xi);
         else
            break;
      }
   }
   State[N] = Atr[N];
}
```

注意，结点可被访问，条件是它们还没有被访问过或从上次访问以来又有一个新的属性被计算出来。这意味着：可对每个结点做有限次数的访问，即，遍历为线性的。同样，这里也只能计算非循环的属性文法。

作为示例，再次考虑G2。我们从图14-12中的树开始。(实际的属性值没有被给出；非终结符还没有被访问；终结符也没有属性。)

从Atr(S) = {A}开始，我们调用visit_node2(S):

(1) 立即计算Z.H，因此Atr(S) = {A}，Atr{Z} = {H}。

(2) 现在Atr{Z} = Ø ≠ State(X) = {NV}，因此X被访问。

(3) visit_node2(X)。

此时没有属性被计算，因此我们设置State(X) ≠ Ø并返回。

(4) 现在State(X) = Atr(X) = Ø，因此我们访问Y。(再次地，Atr(Y) = Ø ≠ State(Y) = NV。)

(5) visit_node2(Y)。

此时没有属性被计算，因此我们设置State(Y) ≠ Ø并返回。

(6) 现在State(Y) = Atr(Y) = Ø，因此我们访问Z(Atr(Z) = {H} ≠ State(Z) = NV)。

(7) visit_node2(Z)。

此时计算Z.G，且State(Z)变为{G, H}。

(8) 现在可以计算X.C。因为Atr(X) = C ≠ State(X) = Ø，所以我们访问X。

(9) visit_node2(X)。

此时计算X.D，且State(X)变为{C, D}。

(10) 现在可以计算S.B和Y.E 。因为Atr(Y) = E ≠ State(Y) = Ø，所以我们访问Y。

(11) visit_node2(Y)。

此时计算Y.F，且State(Y)变为{E, F}。

(12) 没有更多的属性需要被计算且也没有更多的结点需要被访问 (Atr(X) = State(X)，Atr(Y) = State(Y)，...)。此时计算完毕。

注意：采用这种技术的计算比使用原来的visit_node()过程的情况更为直接且考虑更周到。

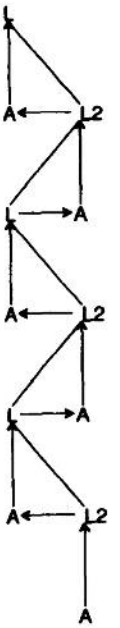


图14-11 基于文法G5的属性化树中的信息流

523

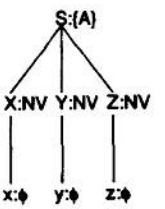


图14-12 应用visit_node2()之前文法G2的属性树

14.1.3 直接属性计算程序

不像树遍历计算程序，直接属性计算程序将属性计算与分析相结合而不是在分析后进行。这样看来，

它们是很好的一遍语法制导编译器的模型。此类计算程序在计算某些属性时将放弃那些不需要的属性值。这种翻译的方法通常可以借助于副作用（例如，代码或IR的生成）或通过把翻译构造为目标符号的综合属性来实现。然而，如果需要的话，我们可以显式地将属性值写到文件中而不是扔掉。那样，我们可以
524 根据是否需要而产生完整的属性语法树。

这种直接方法的执行和语法分析器结合在一起，且它的特征可以从两个方面来加以描述：（1）要使用的语法分析器，（2）要适应的属性流种类。通常，对于给定的分析方法，我们试着使用可能是最一般的属性流。然而，所有这些模式都至少假定有从左到右的属性流。因此前向引用（对于一遍编译器来说）是个问题。

LL(1) L-属性定义计算程序

LL(1) L-属性定义类的计算程序相当有名且功能强大（Lewis、Rosenkrantz and Stearns 1976）。我们假设属性文法是L-属性定义的(如先前所定义的)，其底层的CFG亦为LL(1) 的且文法采用*简单赋值形式*。我们曾经提及任何属性文法可以被很容易地改造成简单赋值形式。

L-属性定义的属性流可以完美地适应于基于LL(1)的属性计算。L-属性定义规则指出，在计算产生式的属性时，我们应该首先计算左部符号的继承属性，然后再（从左至右地）计算产生式右边的各个属性，最后才计算左部符号的综合属性。这就和首先预测产生式左部并接着预测和匹配产生式右部符号的LL(1)分析器很好地配合起来。

假定我们的计算程序使用属性栈（其实也就是语义栈）。当一个非终结符被预测时，它的继承属性被压入栈中。当产生式的右部符号被识别时，它们的继承属性和综合属性被先后压入栈中。最后，当整个右部被识别时，将从栈中弹出右部所有的属性并压入左部的综合属性。

因此，如果预测并识别出X → YZ，那么属性栈上的有关操作如下：

(1) 压入X的继承属性:

Stack = ··· Inh(X)

(2) 压入Y的继承属性:

Stack = ··· Inh(X) Inh(Y)

(3) 压入Y的综合属性（在分析Y后）:

Stack = ··· Inh(X) Inh(Y) Syn(Y)

(4) 压入Z的继承属性:

Stack = ··· Inh(X) Inh(Y) Syn(Y) Inh(Z)

(5) 压入Z的综合属性（在分析Z后）:

Stack = ··· Inh(X) Inh(Y) Syn(Y) Inh(Z) Syn(Z)

(6) 弹出所有右部符号的属性并压入X的综合属性:

Stack = ··· Inh(X) Syn(X)

525 由于L-属性定义的约束，因此所有属性值在需要时才存在于栈上且位于栈中（相对于栈顶的）一个已知的位置上。

为完善属性计算程序，我们仅需要提供操作属性栈的方法。为此我们引入*拷贝符号*（copy symbol）。它们本质上是进行属性移动而非属性计算的动作符号。这些拷贝符号可以从拷贝规则*自动地*生成并在继承属性或综合属性被操控时出现。

通过下面简单的示例，我们可以清楚地看到拷贝符号的使用。图14-13中给出了仅使用运算符 + 的新版本G1（称之为G1A）。该文法是LL(1)的和 L-属性定义的，且采用了简单赋值形式。用标记为#1、#2的拷贝符号替代G1A中的拷贝规则就产生了如图14-14所示的文法。

图14-15显示了在文法G1A的变换版本中分析和计算10 + 11 $ 的追踪过程。在分析完成时符号E的

综合属性（E.val）将驻留在（属性）栈上。

G1A: V_n = {E,T,T-List} V_t = {C,$,+} Action symbols = <add>

Attributes:
Syn(E) = Syn(T) = Syn(T-List) = Syn(C) = {Val}
Inh(T-List) = {LeftVal}
Inh(<add>) = {V1,V2} Syn(<add>) = {Result}

产生式		属性规则
E	→ T T-List $	E.Val := T-List.Val T-List.LeftVal := T.Val
T	→ C	T.Val := C.Val
$T\text{-List}^1$	→ + T <add> $T\text{-List}^2$	<add>.v1 := $T\text{-List}^1$.LeftVal <add>.v2 := T.Val $T\text{-List}^2$.LeftVal := <add>.Result $T\text{-List}^1$.Val := $T\text{-List}^2$.Val
T-List	→ λ	T-List.Val := T-List.LeftVal

图14-13 仅使用拷贝规则的L-属性定义文法

E → T #1 T-List $ #2
T → C #3
T-List → + T #4 <add> #1 T-List #5
T-List → #1

#1 : Push copy of Top Element
#2 : Temp := Top Element; Pop 3; Push Temp
#3 : None -- Equivalent to Temp := Top; Pop 1; Push Temp
#4 : Push copy of Top-1; Push copy of Top-1.
#5 : Temp := Top; Pop 6; Push Temp

图14-14 使用拷贝符号变换文法G1A

Attribute Stack	Parse Stack	Input
Empty	E	C:10+C:11 $
Empty	T #1 T-List $ #2	C:10+C:11 $
Empty	C:10 #3 #1 T-List $ #2	C:10+C:11 $
10	#3 #1 T-List $ #2	+C:11 $
10	#1 T-List $ #2	+C:11 $
10 10	T-List $ #2	+C:11 $
10 10	+T $ #2	+C:11 $
10 10	+T #4 <add> #2 T-List #5 $ #2	+C:11 $
10 10	T #4 <add> #1 T-List #5 $ #2	C:11 $
10 10	C:11 #3 #4 <add> #1 T-List #5 $ #2	C:11 $
10 10 11	#3 #4 <add> #1 T-List #5 $ #2	$
10 10 11	#4 <add> #1 T-List #5 $ #2	$
10 10 11 10 11	<add> #1 T-List #5 $ #2	$
10 10 11 10 11 21	#1 T-List #5 $ #2	$
10 10 11 10 11 21	#1 T-List #5 $ #2	$
10 10 11 10 11 21 21	T-List #5 $ #2	$
10 10 11 10 11 21 21 21	#5 $ #2	$
10 10 21	$ #2	$
10 10 21	#2	Empty
21	Empty	Empty

图14-15 使用G1A的属性计算追踪

注意：存在很多针对栈操作例程的可能优化。例如，在

E → T #1 T-List $ #2

中，可以通过允许T.val和T-List.LeftVal共享相同的栈位置来消除由#1所暗示的拷贝。我们稍后会看到，

526 在基于LR的属性计算程序中，这种优化是至关重要的。同样，也可以合并序列 #4 <add> #1为一个例程。可以直截了当地实现并在实践中使用这些优化。

LR S-属性定义计算程序

LR类的语法分析器较之LL分析器的优势在于：它们将产生式的识别一直推迟到整个产生式右部被识别出来为止。然而，在考虑属性计算时，这种优势限制了其所适应的属性流。特别是由于LR分析器一般不知道它正在识别的是什么产生式，因此它也就不能为符号提供继承属性。所以LR技术常常局限于S-属性定义文法，该文法仅允许非终结符有综合属性。特别地，一个属性文法是S-属性定义的（S-

527 attributed），当且仅当

- 它是L-属性定义的。
- 非终结符仅有综合属性。
- 所有动作符号（和拷贝符号）出现在产生式右部所有终结符和非终结符的右边。

考虑如图14-16 所示的G1B，它是文法G1的另一个仅使用运算符 + 的版本，同时也是S-属性定义的、LR(0)的和采用简单赋值形式的。再次使用拷贝符号变换，得到如图14-17所示的文法。

G1B: V_n = {E} V_t = {C,+} Action Symbols = {<add>}
Attributes:
Syn(E) = Syn(C) = {Val}
Syn(<add>) = {Result} Inh(<add>) = {v1,v2}

产生式	属性规则
$E^1 \rightarrow E^2$ + C <add>	<add>.v1 := E^2.val <add>.v2 := C.val E^1.val := <add>.Result
E → C	E.val := C.val

图14-16 使用拷贝规则的S-属性定义文法

E → E + C #1 <add> #2
E → C #3

#1: Push copy of Top-1; Push copy of Top-1
#2: Temp := Top; Pop 5; Push Temp
#3: No action -- In effect Temp := Top; Pop 1; Push Temp

图14-17 使用拷贝符号变换文法G1B

我们注意到那些拷贝和动作符号串中包括了自底向上编译器中的标准语义例程并且将在语法分析器指示归约的时候被调用。

LR LC-属性定义计算程序

S-属性定义类对我们没有吸引力，因为它不允许有继承属性。而在实践中，如果有可能，应该允

528 许使用继承属性。一般来说，产生式不需要在最左端（如在LL中）或它的最右端（如在LR中）被识别，但它可以在其右部中间的某个位置上被识别。事实上，产生式A → αβ 的右部可以分成两部分：左角（left corner）和尾部（trailing part）。根据定义，任何产生式可以在其左角分析处理后即被识别出来。在LL(1)中，左角总是为空；在LR(1)中，左角有时可以包括整个右部（尽管在许多LR文法中左角都很小）。这种右部的划分暗示着β（尾部）允许使用继承属性。因此，就有了以下S-属性定义和L-属性定义的混合属性流。一个属性文法是LC-属性定义的（LC-attributed），当且仅当

- 它是L-属性定义的。

• 在左角中出现的非终结符没有继承属性。

• 左角中不会出现动作符号。

实际上，左角允许S-属性流，而尾部允许L-属性流。

LC-属性文法的确是S-属性文法的改进。同样重要的是，我们可以用任何LR类的分析器来分析它们。这个想法是：从概念上我们创建一个带有能界定左角的特殊识别符号（recognition symbol）的CFG。如果属性文法是LC-属性定义的，那么动作符号和拷贝符号仅能在尾部出现，但不必在最右端。这样，我们可能有：

```
A → X Y & <A1> Z <A2>
```

（&是识别符号；LR分析器看不到它——它的出现仅为了界定左角。）对那些在最右端出现的动作符号和拷贝符号（如<A2>），和以往一样，我们在识别产生式时处理它。为处理<A1>，我们引入新的非终结符和产生式（<call A1> → λ）。因为这个（总是生成λ的）新的非终结符必须在尾部，所以我们可以肯定它将被正确地分析。事实上，尾部被定义为产生式右部的某个区域，在其中我们可以在不破坏可分析性的情况下放置新的仅产生λ的非终结符。

接着，我们将拷贝符号和动作符号与那个刚创建的λ-产生式相关联。先前的产生式因而被变换为：

```
A → XY<call A1>Z<A2>
<call A1> → <A1>
```

可以使用普通的LR类技术分析这些产生式。LC-属性定义类最初由Rowland（1977）定义。

左角中继承属性有限制的使用

LC-属性文法的使用也不完全令人满意，因为有时必须为左角符号提供继承属性。考虑如图14-18所示的GA3，它是使用运算符+的另一个版本的G3并带有最小的左角。 529

```
G3A: Vn = {E}  Vt = {+,C}  Action Symbols = {<add>,<check>}

Attributes:
   Inh(E) = {Max}   Syn(E) = {Val}
   Syn(C) = {Val}
   Inh(<add>) = {v1,v2,Max}     Syn(<add>) = {Result}
   Inh(<check>) = {Val,Max}     Syn(<check>) = {Result}

   产生式                          属性规则

E¹ → E² & + C <add>      E².Max := E¹.Max
                         <add>.v1 := E².Val
                         <add>.v2 := C.Val
                         <add>.Max := E¹.Max
                         E¹.Val := <add>.Result

E → & C <check>          <check>.Val := C.Val
                         <check>.Max := E.Max
                         E.Val := <check>.Result
```

图14-18 左角中需要继承属性的属性文法

GA3不是LC-属性定义的，因为左角符号E有继承属性。然而，在左角中继承属性所需要的拷贝符号经常会被优化掉（Watt 1977）。此例中，可以通过让属性共享相同的栈中位置而消除从E^1.Max到E^2.Max的拷贝。事实上，前i个(i > 1)栈位置的拷贝可以通过共享而被优化掉（这些属性实际上是只读的）。去除左角中的拷贝后将产生：

```
E → E & + C #1 <add> #2
E → & C #3 <check> #4

#1: Push Top-1; Push Top-1; Push Top-4;
```

```
#2: Temp := Top; Pop 6; Push Temp
#3: Push Top; Push Top-2
#4: Temp := Top; Pop 4; Push Temp
```

530 这个文法可以用前一节的方法来处理。这种拷贝优化不允许在左角中有任何的继承属性（或动作符号），但Watt声称，它足以处理一个Pascal语言的属性文法（使用LR类分析器）。

14.1.4 属性文法示例

在图14-19中，我们给出包括**if**语句和**while**循环的文法片段所对应的属性文法示例。使用这个例子有两个目的：(1) 展示一个比先前抽象例子更为真实的属性文法，(2) 举例说明在不要求一遍处理的时候如何简化语义处理算法。该例子是基于Tomasz Kowaltowski所开发的示例。在某种程度上，它比在第12章提出的布尔表达式短路计算的算法要简单些，这是因为它在计算这样的表达式的时候用到了从右到左的属性流。

```
-- The symbol "&" represents string concatenation

S → if E then L end if          E.case:=FALSE
                                E.label:=S.next
                                L.next:=S.next
                                S.code:=E.code & L.code &
                                        generate(LABEL,S.next,"","")

S → if E then L1 else L2 end if E.case:=FALSE
                                E.label:=new_label()
                                L1.next:=S.next
                                L2.next:=S.next
                                S.code:=E.code & L1.code &
                                        generate(JUMP,S.next,"","") &
                                        generate(LABEL,E.label,"","") &
                                        L2.code &
                                        generate(LABEL,S.next,"","")

S → while E loop L end loop;    E.case:=FALSE
                                E.label:=S.next
                                S.begin:=new_label()
                                L.next:=S.begin
                                S.code:=generate(LABEL,S.begin,"","") &
                                        E.code & L.code &
                                        generate(JUMP,S.begin,"","") &
                                        generate(LABEL,S.next,"","")

S → OtherS                      S.code:=OtherS.code

L → S                           S.next:=L.next
                                L.code:=S.code

L → L1 S                        L1.next:=new_label()
                                S.next:=L.next
                                L.code:=L1.code & S.code

E → E1 BoolOp E2                E2.label:=E.label
                                E2.case:=E.case
                                if BoolOp.operator = OrElseOp then
                                  E1.case := TRUE
                                  if E.case then
                                    E1.label:=E.label
                                    E.code:= E1.code & E2.code
                                  else
                                    E1.label:=new_label()
                                    E.code:= E1.code & E2.code &
                                          generate(LABEL,E1.label,"","")
                                  end if;
                                else -- BoolOp.operator = AndThenOp
                                  E1.case := FALSE
                                  if E.case then
                                    E1.label:=new_label()
```

图14-19 短路计算的属性文法

```
                                E.code:= E¹.code & E².code &
                                    generate(LABEL,E¹.label,"","")
                              else
                                E¹.label:=E.label
                                E.code:= E¹.code & E².code
                              end if;

E → not E¹                    E¹.label:=E.label
                              E¹.case:=not E.case
                              E.code:=E¹.code

E → (E¹)                      E¹.label:=E.label
                              E¹.case:=E.case
                              E.code:=E¹.code

E → id¹ RelOp id²             E.code:= if E.case then
                                    generate(BR,RelOp.operator,
                                        id¹.loc,id².loc,E.label)
                                  else
                                    generate(BR,
                                      complement(RelOp.operator),
                                      id¹.loc,id².loc,E.label)
                                  end if

E → true                      E.code:=if E.case then
                                    generate(JUMP,E.label,"","")
                                  else
                                    empty
                                  end if

E → false                     E.code:=if E.case then
                                    empty
                                  else
                                    generate(JUMP,E.label,"","")
                                  end if
```

图14-19 （续）

在这个文法中，（由非终结符S指示的）语句有继承属性next和综合属性code；属性next是在当前正被考虑的语句之后执行的某条语句的标记，而属性code则是包含为语句产生的代码串。（这些属性规则假设元组生成例程generate()以串的方式返回元组。）由非终结符E指示的每个表达式有两个继承属性：label，一个符号标号属性；case（true或false）。这些属性解释如下：如果表达式的值计算为case，那么表达式的代码将导致到label的跳转；否则，代码将“流到”下条指令。

这种方法比在第12章中见到的两种算法要简单，那些算法需要对根据表达式值的真假来回填的元组链进行维护。而这里仅有一个标号与每个表达式相关联（如果使用了回填方法，那也只有一条链和表达式关联）。这种简化的关键在于发现产生式E → E^1 BoolOp E^2相关的属性计算规则。BoolOp的属性operator是用来计算E^1.label（从右到左属性流），这样就有可能在分析E^1的时候产生合适的跳转以处理它独自决定表达式值的情况。 531

可以考虑下面的**if**语句作为使用该文法生成代码的例子：

```
if A > B or else C < D then
  OtherS¹
else
  OtherS²
end if
```

这个例子中的属性计算说明在图14-20中，它开始于以下产生式对应的结点：

S → **if** E **then** L^1 **else** L^2 **end if** 532 ~ 533

我们假设继承属性S.next := L1。这里仅包括与属性计算规则中的符号相对应的结点的访问。

```
visit_node(S):
  E.case := FALSE
  E.label := L2  /* a new label */
  L1.next := L1
  L2.next := L1

  visit_node(E): /* E → E1 BoolOP E2 */
    E2.label := L2
    E2.case := FALSE
    /* The next two attributes depend on right-to-left information flow, */
    /* since the rules to generate them consider the operator BoolOp */
    E1.case := TRUE
    E1.label := L3  /* a new label */

    visit_node(E1)
      E1.code := (BR,>,A.loc,B.loc,L3)

    visit_node(E2)
      E2.code := (BR,>=,C.loc,D.loc,L2)

    E.code := (BR,>,A.loc,B.loc,L3)
              (BR,>=,C.loc,D.loc,L2)
              (LABEL,L3)

  visit_node(L1): /* L1 → S */
    S.next := L1

    visit_node(S): /* S → OtherS */
      S.code := {code for OtherS}

    L1.code := {code for OtherS}

  visit_node(L2): /* L2 → S */
    S.next := L1

    visit_node(S): /* S → OtherS */
      S.code := {code for OtherS}

    L2.code := {code for OtherS}

  S.code := (BR,>,A.loc,B.loc,L3)
            (BR,>=,C.loc,D.loc,L2)
            (LABEL,L3)
            {code for OtherS}
            (JUMP,L1)
            (LABEL,L2)
            {code for OtherS}
            (LABEL,L1)
```

图14-20 使用图14-19中属性文法追踪属性的计算

14.2 树结构的中间表示

作为表达式树的推广，第8章里所讨论的中间表示为整个程序引入了树结构的IR想法。这样的IR以前被认为是基于*抽象语法树*（abstract syntax tree）。抽象语法树是通过消除分析树中那些没有语义含义的显式语法元素表示而得到的。抽象语法树可以很容易地被用作IR以保存由多遍属性计算程序所使用的属性。

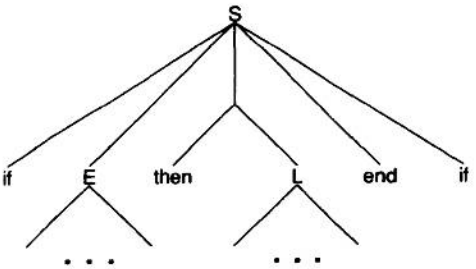
例如，可以考虑图14-19中最简单的**if**语句产生式：

S → if E then L end if

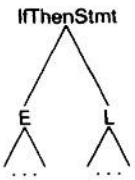
534 该产生式的分析树具有如图14-21a所示的形式。作为对比，图14-21b中显示了等价的抽象语法树表示。

这里说明一下它们两者的基本差别。首先是用显式的IfThenStmt替代了通用的语句结点S。该替换使得从这个结点的名字就可能知道它下面所期待的结构。尤其是，这种替换使得第二种差别成为可能：所有的关键字都已被从结点IfThenStmt的后代中删除。而这个结点的名字则暗示着它们在源程序中的存在。因

为除了它们的存在，这些关键字没有传递任何语义信息，所以在树中也就没有必要显式地表示它们。



a) 简单If语句的分析树



b) 简单if语句的抽象语法树

图14-21 简单If语句的分析树和抽象语法树

结点E和L的子树也根据抽象语法树的定义做了类似的变换。这种改变对以L为根的语句列表子树的意义相当大。在分析树中，这棵子树代表由产生式L → L S 和L → S产生的语句列表。因此，如果列表包含不止一条的语句，那么将创建下一级的L结点，且可以递归地应用相同的构造。而在抽象语法树中，使用StmtList结点指向显式的下属语句结点列表的表头。

535

如此描述的抽象语法树实质上就是一种比分析树更为简洁的源程序语法的表示。这样一种树可以用作带有多遍分析组成结构的编译器中的语法分析器和语义分析阶段之间的接口。当然，为便于语义分析，还必须添加更多的信息。属性的概念很适合此目的。我们可以把从结点IfThenStmt出发的指向表达式和语句列表子树的指针想像为*结构化属性*（structural attribute）。在一个完整的属性文法定义中，语句有代表名字空间（符号表）的继承属性environment，出现在该属性里的语句中的任何标识符都将被解释。environment是一个*语义属性*（semantic attribute）的例子。稍后的例子将使用另一种属性，*代码生成属性*（code generation attribute）。

除了那些需用来传递结构信息的属性外，还可以在树结点中添加其他属性以便给编译器的任一阶段提供必需的信息。例如，表达式子树综合一个描述表达式类型的语义属性。这样的属性可被用于那些检查类型兼容以加强静态语义约束的规则中，或更一般的情况是用来解析重载的运算符。语义属性和代码生成属性通常在不同的树遍历阶段计算，因为前者是与机器无关的而后者显然是与目标机器相关的。

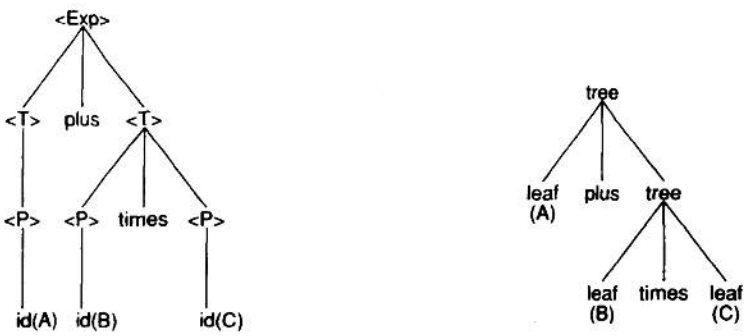
14.2.1 抽象语法树接口

通常，抽象语法树是通过使用结构来表示结点并使用指针将结点链接在一起来实现的。主要的设计选择在于，是为每种树结点定义不同的类型，还是让所有的树结点具有单一的类型。在后一种情况下，*一个结点名字*（node name）域将用来区分可能的结点类型，并且基于结点名字的变体将包含合适的属性。单一结点类型的使用是最常见的选择，这主要是因为它使得用于执行树遍历的代码非常简单。而树的遍历，当然是在抽象语法树上所做的最基本的操作。

考虑下面简单表达式的文法：

```
<Exp>      → <T> { <addop> <T> }
<T>        → <P> { <multop> <P> }
<P>        → ID
<add op>   → PLUS | MINUS
<mult op>  → TIMES | DIVIDE
```

语法上非终结符<Exp>、<T>和<P>之间的区别仅在于指明运算符的优先级。除了表示的是简单标识符或包含运算符的树之外，它们没有语义上的区别。这两种情况可以分别由称为leaf和tree的抽象语
536 法树结点来表示。四种运算符均有不同的语义含义，并且有四种适合于表示它们的不同的结点类型。运算符记号的名字可以被当作树结点的名字。因此，表达式A+B*C有如图14-22a所示的分析树，其抽象语法树如图14-22b所示。



a) A+B*C的分析树　　b) A+B*C的抽象语法树

图14-22 A+B*C的分析树和抽象语法树

可以根据采用以下声明的单一记录类型方法来构造此图中显示的抽象语法树：

```
enum node_type { TREE, LEAF, PLUS, MINUS, TIMES, DIVIDE };

typedef struct tn {
   enum node_type node_name;
   union {
      /* node_name == TREE */
      struct {
         struct tn *op, *left, *right;
537   };
      /* node_name == LEAF */
      string id;

      /* anything else, empty */
   }
} tree_node;
```

14.2.2 语法树抽象接口

如果使用了语法树抽象接口，那么先前的设计决策需要不会对编译器的其余部分产生影响。通过使用抽象接口（abstract interface），语义例程和代码生成器不需要直接访问树结点记录的域。相反地，只有通过语法树“包”中的过程或函数才能从外面访问到属性，这些过程或函数可以设置或返回属性值，而树结点的类型不必输出。这种抽象方法不仅隐藏了基本的结构化设计策略，而且还隐藏了单独属性的表示方法（例如，它是某种允许计算的显式的值或信息）。在C语言中，这种过程式接口实际上可以是一组宏。这样既允许对属性的有效访问又可以通过这些接口继续提供数据隐藏。

事实上，Diana是Ada语言的一种标准的IR。它是采用诸如刚才所描述的抽象接口而定义的一种树结构的IR。访问Diana的接口是采用描述抽象数据结构的特殊符号——接口描述语言（IDL）（Nestor, Wulf, and Lamb 1981）来定义的。因为Diana过于复杂而不便于提供简洁的使用示例，所以我们将提供图14-22b中抽象语法树的IDL描述及和它所定义的接口对应的C语言声明。在本章最后将提供一个更复杂的IDL示例。这些示例说明了访问抽象语法树的抽象接口的概念，而这在Diana中是最基本的。

简单的IDL示例

图14-23中的第一个IDL示例改编自《IDL参考手册》（IDL Reference Manual）。它是表示图14-22中简单算术表达式的抽象数据类型的定义。

```
mode AST root EXP is

   -- first we define the notion of an expression, EXP

   EXP ::= leaf | tree ;

   -- next we define the nodes and their attributes

   tree => op : OPERATOR, left : EXP, right : EXP ;
   leaf => name : String ;

   -- finally we define the notion of an OPERATOR as the union of
   -- a collection of nodes; note these particular nodes have no
   -- attributes and hence have no further definitions

   OPERATOR ::= plus | minus | times | divide ;

end
```

图14-23 表达式抽象语法树的IDL描述

EXP和OPERATOR是类名字的示例。它们用来定义树的结构但并不实际在树中作为结点出现。定义的6种结点为：tree、leaf、plus、minus、times和divide。它们当中的每一个都可以作为树的结点出现。这其中只有tree和leaf结点有属性，它们属性的名字和类型各由两行属性定义给出。leaf结点的名字属性的类型String是IDL的内建类型。而EXP（由稍后出现的tree或leaf定义）作为树的根这一事实则由标题行来指定。

538

对应图14-23中IDL描述的C语言声明如图14-24所示。

```
/* specification of AST */
typedef struct tn {
        . . .              /* contents are "private" */
} tree_node;
typedef enum {
   TREE, LEAF, PLUS, MINUS, TIMES, DIVIDE
} node_name;
/*
 * Tree constructors. Pointers are used both for
 * efficiency and due to C's call-by-value semantics.
 */
tree_node *make(node_name n);
void destroy(tree_node *t);
node_name kind(tree_node *t);
/*
 * The following procedure/function pairs are used to
 * set and access attributes of nodes.
 */
```

图14-24 与图14-23对应的C语言声明

```
void        set_op(tree_node *node, tree_node *value);
tree_node *get_op(tree_node *node);
void        set_left(tree_node *node, tree_node *value);
tree_node *get_left(tree_node *node);
void        set_right(tree_node *node, tree_node *value);
tree_node *get_right(tree_node *node);
void        set_name(tree_node *node, string value);
string    *get_name(tree_node *node);
```

图14-24 （续）

给定这些声明（见图14-24），在图14-25中的调用序列将构造与A+B*C对应的树，其中假定变量T1、T2、T3、T4和T5的类型为(tree_node*)。这些代码主要是说明那些进行语法树构造以及属性设置的过程和函数的使用。

```
T1 = make(LEAF);
set_name(T1, "A");
T2 = make(LEAF);
set_name(T2, "B");
T3 = make(LEAF);
set_name(T3, "C");
T4 = make(TREE);
set_left(T4, T2);
set_right(T4, T3);
set_op(T4, make(TIMES));  /* T4 is the tree for B*C */
T5 = make(TREE);
set_op(T5, make(PLUS));
set_left(T5, T1);
set_right(T5, T4);        /* T5 is the tree for A+B*C */
```

图14-25 使用AST例程构造的抽象语法树

图14-26中的过程postfix()使用来自图14-24中相同的例程做表达式树的遍历以及输出表达式的后缀表示。该例子所强调的是那些访问树结点属性的函数。

```
void postfix(tree_node *t)
{
    switch (kind(t)) {
    case LEAF:    printf(" %s", name(t)); break;
    case PLUS:    printf(" +"); break;
    case MINUS:   printf(" -"); break;
    case TIMES:   printf(" *"); break;
    case DIVIDE:  printf(" /"); break;
    case TREE:    postfix(get_left(t));
                  postfix(get_right(t));
                  postfix(get_op(t));
                  break;
    }
}
```

图14-26 表示为AST的表达式的后缀打印

短路计算示例的IDL定义

图14-27包含图14-19中短路计算属性文法需要的抽象语法的IDL定义。该定义的一个重要特性是将属性定义划分为两部分：定义树结构的属性和用来驱动代码生成处理的属性。Diana，这个Ada语言的IR，也采用这种定义，但Diana的定义区分四种不同的属性：结构属性、词法属性（这种信息允许精确重建源程序）、语义属性和代码生成属性。

539 在这个示例中使用了IDL的另一个新特性。StmtList结点列表的属性采用序列类型（Seq）定义。因而在与包含序列类型的IDL描述相对应的C语言声明中必须包括操作这些序列的函数。

```
mode SyntaxTree root STMT is

   STMT ::= IfThenStmt | IfThenElseStmt | WhileStmt | OtherStmt ;
   LIST ::= StmtList ;
   EXP  ::= BoolExp | Negation | Parens | RelExp | TrueExp | FalseExp ;

   -- The following attributes define the structure of the abstract syntax tree
   -- for statements

   IfThenStmt => condition : EXP, thenpart : LIST ;
   IfThenElseStmt => condition : EXP, thenpart : LIST, elsepart : LIST ;
   WhileStmt => condition : EXP, loopbody : LIST ;
   StmtList => list : Seq of STMT ;

   BoolExp => left : EXP, operator : BOOLOP, right : EXP ;
   RelExp => left : EXP, operator : RELOP, right : EXP ;
   Negation => expression : EXP ;
   Parens => expression : EXP ;

   -- the following declarations are of the attributes used by the
   -- code-generation process encoded in the attribute grammar

   IfThenStmt => next : String, code : String ;
   IfThenElseStmt => next : String, code : String ;
   WhileStmt => next : String, begin : String, code : String ;
   OtherStmt => code : String ;

   BoolExp => case : Boolean, label : String, code : String ;
   Negation => case : Boolean, label : String, code : String ;
   Parens => case : Boolean, label : String, code : String ;
   RelExp => case : Boolean, label : String, code : String ;
   TrueExp => case : Boolean, label : String, code : String ;
   FalseExp => case : Boolean, label : String, code : String ;

end
```

图14-27 表示图14-19中文法的IDL定义

图14-28中的声明是与图14-27中IDL定义相对应的C语言接口。它们是基于标准的程序模板，其中包括创建和操作树和序列所需的类型及子程序。类型node_name以及那些设置并访问属性的子程序的声明当然是基于这个特定的IDL定义的。 540

```
/* specification of SyntaxTree */

typedef struct tn {
    . . .              /* contents are "private" */
} tree_node;

typedef  . . .  seq_type;    /* also private */

typedef enum {
    IFTHENSTMT, IFTHENELSESTMT, WHILESTMT, OTHERSTMT,
    STMTLIST, BOOLEXP, NEGATION, PARENS, RELEXP, TRUEEXP,
    FALSEEXP);
} node_name;

/*
 * Tree constructors. Pointers are used both for
 * efficiency and due to C's call-by-value semantics.
 */

tree_node *make(node_name n);
void destroy(tree_node *t);
node_name kind(tree_node *t);

/* handling of list constructs (for STMTLIST) */
```

图14-28 与图14-27对应的C语言声明

```
tree_node *head(seq_type *l);
seq_type  *tail(seq_type *l);
seq_type  *seq_make(void);  /* returns an empty list */
boolean    is_empty(seq_type *l);
/* inserts t at start of l */
seq_type  *insert(seq_type *l, tree_node *t);
/* inserts t at end of l */
seq_type  *append(seq_type *l, tree_node *t);

/*
 * The following procedure/function pairs are used to set
 * and access attributes of nodes.  The first group deals
 * with the attributes that define the structure of the
 * abstract syntax tree.
 */

void    set_condition(tree_node *node, tree_node *value);
void    set_thenpart(tree_node *node, tree_node *value);
tree_node *get_condition(tree_node *node);
tree_node *get_thenpart(tree_node *node);
      .
      . /* similar subprograms for elsepart, loopbody */
      . /* list, left, right, operation and expression */
      .

/*
 * This second group of procedures and functions handles
 * the attributes that drive the code-generation algorithm.
 * Different nodes that have attributes with the same name
 * may share a pair of these routines, since all nodes are
 * tree_nodes.
 */

void   set_next(tree_node *node, string value);
string get_next(tree_node *node);
void   set_code(tree_node *node, string value);
string get_code(tree_node *node);
      .
/* similar subprograms for begin, label and case */
      .
      .
```

541
~
543

图14-28 （续）

14.2.3 实现树

通常，我们认为树结构的中间表示需要很大的存储空间。其实这种看法未必准确。如果使用最常见的编译器组织结构，即前端产生整个程序的树形表示（每个结点被表示成动态分配的变体记录），那么即使是大小适度的程序也会需要相当大的存储空间。尽管存储需求是一个重要的问题，但采用其他形式的编译器组织仍然是有可能的。

如果仅需要做有限的优化（或根本不做），那么可以这样来组织编译器，即让它的构件（像在一遍编译器中那样）作为协同例程运行。程序的某些部分（通常是过程或函数）分别由每个部件来处理而它们的树则被抛弃。这种处理将一直重复到整个程序被编译完为止。

与在任何时刻为节省空间而仅保存部分树的方法相比，有多种其他的方法可用来以更紧凑的方式保存整个树。语法树可被线性化（如采用后缀表示），例如，为了节省指针可能需要的空间，在这种表示中子结点可以被隐式地放置在与结点相关的位置上。线性化表示简化了某些形式的树遍历（通常从左到右），但所付代价是使得普通的遍历更加昂贵。线性表示作为树的外部表示形式时很有用，因为它们消除了来自典型指针实现中的内存地址的依赖性。

另一种节省空间的技术是通过共享相同的子树而将一棵语法树转换成有向无环图（dag）。最明显的可共享的结点是表示文字字面值特别是标识符的叶子结点。然而，还有更多有价值的子树可被共享，前

提条件是可以识别它们。

最后，某些平凡的结点根本不需要被表示为结点。例如，AST示例中的OPERATOR类的结点就没有属性。它们惟一的意义在于表示各种不同的运算符。对这样的tree结点的引用可以被优化掉，替代的方法是用一个存储在tree结点中实现为小整数的简单枚举类型的值来代替指向实际结点的指针。

练习

1. 根据14.2节中的示例，对于给定的描述语言具体语法的文法，非形式化地提出能推导出合适抽象语法树表示的一般算法。它能自动进行这个推导吗？
2. 在14.1节中的所有示例文法里，属性计算规则将计算完整语法树上的结点的属性。应如何扩展你在练习1中提出的算法来处理计算规则的变换以应用于导出的抽象语法树？将你的算法应用到文法G1和G3以检验它的正确性。 544
3. 使用图14-19中的文法，追踪以下程序的属性计算（使用图14-20的格式）：

```
while not (A = B) and then C < D loop
  OtherS;
  if A /= B then
    if C = D then
      OtherS;
    end if;
  else
    OtherS;
  end if;
  OtherS;
end loop;
```

4. 图14-25和图14-26中的过程说明了语法树抽象接口的使用。编写一个执行等价功能的过程，它使用14.2.1节中由tree_node直接定义的接口。
5. 借助分析或实验来确定：在使用14.2.3节中讨论的语法树的实现优化时，诸如子树共享或平凡结点删除所能带来的空间节省的量级。 545

第15章 代码生成和局部代码优化

15.1 概述

在我们所开发的编译器模型中，综合部分被划分为翻译和代码生成这两个不同的阶段。这种方法有许多优点。最为重要的是，被编入到编译器语义例程里的翻译阶段具有高度的源程序依赖性，而代码生成阶段则是与目标机器相关的。如果要进行再目标（改变目标机器），则保持它们之间的清晰界限是非常重要的。即使再目标不是最初关心的问题，但在编译器被证明是成功的之后，经常会出现将它移植到新机器上或已有机器的新版本上的要求。因此，在源程序和目标机器之间提供清晰划分的设计也就是考虑到了未来的需要。

最简单的代码生成器就是根本没有代码生成器！这看起来有些滑稽，但确实有这么一些场合要求编译器仅产生中间表示（IR）代码而非实际的目标代码。

简单编译器或许会产生可在执行阶段解释的树结构IR。这样的编译器被设计用在程序被频繁修改和重新编译的教学环境中。这时执行速度并不重要，因为程序测试仅使用相当简单的数据且只需（成功）运行若干次即可。然而，因为需要很频繁地重新编译，所以编译速度就显得很重要，这时放弃代码生成阶段和稍后的链接、装入阶段也许是有利的。该折中方案是以较慢的执行换取快速的编译，且在教学或调试环境中这种净收益颇丰。

另一个产生IR代码而非目标代码的较好的编译器示例是Pascal P-*编译器*（P-Compiler）。P-编译器被设计为高度可移植的，它通过为称作*虚拟栈机器*（Virtual Stack Machine，VSM）的假想的栈机器生成P-*代码*（P-code）来达到这一目标。P-代码的设计简单而紧凑。P-编译器采用了源形式（Pascal程序）和目标形式（P-代码）两种形式分发。

为将P-编译器移植到新机器上，需要首先编写P-代码的解释器。（这项工作估计要花大约一个月时间。）一旦P-代码解释器可用，就可以执行P-编译器的目标形式，Pascal程序也因此可以被编译并执行。进一步地，由于可以获得P-编译器的源代码，因此也就可以修改并重新编译该编译器本身。通常，下一步是针对P-代码的代码生成器的实现，这样就在新机器上产生了真正的Pascal编译器。估计现行所有Pascal编译器中的50%~70%均是这个原始的P-编译器的直系后代。

翻译IR到目标代码的最简单方法是将每个IR元组或子树宏展开为等价的目标机器指令序列。目标代码序列挑选时特有的情况分析与选择可以用多种方式来组织。正如15.3节中说明的那样，可以为每个元组或子树编写不同的代码生成器。其他一些方法包括使用一组用特殊的*编码语言*（Wilcox 1971）编写的互相递归的模板例程，以及带有模板匹配例程的模板表（Johnson 1978）。

546
~
547

宏展开方法的主要缺点是它可能相互独立地展开每一条IR指令从而产生较差质量的代码。例如，给定元组（+,A,B,C）和（*,C,D,E），“傻瓜”式的代码生成器将某个求和的值存储到C中，然后立即又从C中再次取出该值并用到下个元组的展开中。为提高代码质量，必须在展开IR代码时维持某些上下文信息或状态，以禁止那些不必要的或质量差的代码序列。

把代码生成看成IR代码宏展开的另一个问题是，有时多于一条的IR指令可被单个的目标机器指令所替代。这种情况一般发生在目标机器有丰富的寻址模式，或有能将多个操作捆绑在单个指令中的奇特指令时（例如，能完成寄存器增值、检测和条件跳转的单条循环控制指令）。

考虑Pascal语句 A := P↑;，如果有间接寻址（指令），那么该语句能被很好地翻译为单条指令。并不是所有的机器都支持间接寻址，因此编译器在IR一级也许会产生后面跟着存储指令的间接取值指令。应该由代码生成器负责判定单个目标机器指令能否获得这两条IR指令的效果。

IR代码（如元组或P-代码）通常比等价的目标机器指令更紧凑，因为它们经过特别设计可以用来表示已编译过的源代码。代码紧凑是一个优点，但如果要解释它们的话，执行速度将会相当慢。如果使用了*线索化代码*（threaded code）方法（Bell，1973），则有可能在紧凑代码和快速执行之间找到一个有趣的平衡。在线索化代码中，每个IR指令将被替换为一个实现IR指令的支持例程的调用（如果有参数，则紧随该调用之后）。在执行指令后，实现例程使用返回地址选择下一个IR指令，而那又是某个实现例程的调用。程序的控制在一系列实现所生成的IR代码的例程调用中穿过。每个IR指令仅需要一种实现，且程序大小比采用宏展开方式所得到的要小得多。同时，程序执行也比IR代码的解释执行快得多，这是因为惟一的开销就是每个IR指令所需的调用和返回（两条指令）。事实上，线索化代码是引起众多关注的创新语言Forth（Brodie 1981）的基础。

15.2 寄存器和临时变量管理

在第11章中，我们提出了一种非常抽象的临时变量分配方法。借助这种方法，临时变量的分配仅涉及为其指派惟一的索引且存储基本的大小和类型信息。所有临时变量管理的复杂工作是交由代码生成器完成的，代码生成器必须将临时变量映射到寄存器并生成有效利用寄存器的代码。

548

编译器临时变量是在有限时间里被指派的位置，目的是保存与当前计算相关的数据。通常，临时变量是寄存器，但是驻留内存的存储型临时变量有时也是必需的。编译器必须小心管理临时变量以避免在它们使用上的冲突。

通常，特定机器上可用的寄存器被划分为若干类：*可分配寄存器*（allocatable register）、*保留寄存器*（reserved register）和*易变寄存器*（volatile register）。

可以在编译时通过调用寄存器管理例程来显式地分配和释放可分配寄存器。在分配以后，除了寄存器的“拥有者”以外，我们将对寄存器进行使用保护。因此，这就保证了寄存器所包含的数据项不会被相同寄存器的其他使用不正确地修改。

可分配寄存器的请求通常很*普通*；也就是说，这样的请求是申请获得某个寄存器类中的任何一个寄存器而不是其中某个特定的寄存器。通常，一个寄存器类中的任何一个寄存器均会满足要求。而且，如果请求的特定寄存器已被占用而在同一寄存器类中还有其他许多寄存器可用，那么采用这种普通的请求将消除由此而引起的问题。

一旦分配了某个寄存器，就必须在它到特定临时变量的指派完成时释放它。通常，我们通过语义例程发出的`free_temp()`命令来释放寄存器。`free_temp()`命令也允许我们标记寄存器的最后一次使用为“*非活跃的*”（dead）。正如我们将看到的，这是一个有价值的信息，因为如果寄存器的内容不再需要保存，就有可能生成更好的代码。

另一方面，保留寄存器和易变寄存器从不会被显式地分配和释放。指派到固定函数的保留寄存器将贯穿程序执行的始终。它们包括显示表寄存器、栈顶寄存器和用于给子例程传递信息的寄存器。

易变寄存器可在任何时刻被任何例程使用。易变寄存器仅在局部代码序列中可以完全地使用，这一点是由代码生成器完全掌控的。也就是说，如果我们正生成数组上索引操作（如A(I + J)），那么此时使用易变寄存器保存数组元素的地址将是错误的，因为下标计算也许会改变那个易变寄存器。若使用可分配寄存器，则它们当然会被加以保护。

易变寄存器在以下几种场合中还是很有用的：

- 需要非常短暂时间的工作单元。（例如，在编译A := B时，我们将B的值装入某个寄存器，然后把

该寄存器内容保存到A。）使用易变寄存器将节省分配、紧接着又释放临时变量的开销。

- 有时，寄存器被创建到一条指令中。例如，许多计算机在诸如块传送的多字指令中使用指定的寄存器来保存指针和字节数。这些值必须被放置在程序员可访问的寄存器中（与程序员不可访问的内部的“微引擎”寄存器不同），是因为这样的指令可以在执行中被中断。然而，指令格式可能没有足够的域允许程序员显式地指定寄存器。为达到此目的，许多计算机的体系结构通过预留某些寄存器来解决这个潜在的问题。例如，在VAX机器上，一条多字传送指令修改0～4号寄存器。如果可用寄存器的总数较少，那么编译器可以选择避免使用这样的指令以便让更多的寄存器用于分配，或者编译器仅在某些特殊的上下文中使用这些指令，在每次使用的前后或许要保存和恢复相关的寄存器。 549

编译器设计中重要的一环是决定如何分配可用的寄存器。某些寄存器被预留用在显示表中，或用来保存运行时栈顶，或保存调用过程中的信息。而其他的寄存器则是可分配的或是易变的。如何划分寄存器的决定主要是基于可用寄存器的个数（如果不止一种分类的话，也包括它们的种类）和系统的约定。在产品化编译器中，这种选择很重要且应当谨慎处理。

15.2.1 临时变量的分类

正如我们已看到的，可以将临时变量划分成若干类。*存储型临时变量很适合保存寄存器或保持大的*数据对象。根据硬件设计，寄存器可以包括一个或多个类。（例如，Univac 1100机器有三个不同的寄存器类，其中的两个有部分重叠。）因此，我们希望对临时变量分配例程的请求要指定所需的临时变量的*数目和种类*。

我们还必须处理某类临时变量中已无可用临时变量的这种可能性。如果发生这种情况，那么作为一种替代处理办法，可用一种简单但立即响应的方式终止编译或代码生成。如果耗尽某类临时变量的可能性较小的话，这种响应或许是一种很好的办法。然而，一种更健壮的分配例程可以选择另外种类的临时变量而不只是报告失败。多数时候，当这种情况发生时将返回存储型临时变量作为对寄存器请求的回应。这样的临时变量可被用作*伪寄存器*（pseudoregister）。在看见对伪寄存器的引用时，代码生成器将其装入一个易变寄存器中，生成使用该易变寄存器的指令，然后将易变寄存器的值写回伪寄存器。为伪寄存器生成的代码的质量可能很差，但是这种方法比放弃要好得多。

临时变量从真实寄存器重新指派到伪寄存器称为*溢出*（spilling）。判断哪一个临时变量将会溢出是一件困难的事，我们准备在15.4.3节中讨论它。直觉上，我们希望溢出那些最不重要的临时变量，这里我们经常使用最近很少引用这样一种依据。在语义例程级调用的`free_temp()`极大地简化了寄存器溢出的问题，其中不再需要的临时变量被显式地标识出来。没有这样的标识，非活跃的临时变量最终仍将会溢出，而代价则是（不必要地）将它们的值转移到伪寄存器中。 550

实践中，存储型临时变量在数目上实际不受限制，但也需要对它们加以谨慎的分配。特别是：

- 存储型临时变量必须被分配在局部活动记录中，而不是全局区域中。否则，递归过程可能失败。
- 在需要的时候，可以通过在编译时扩展活动记录大小而在局部活动记录中创建空间。实际上，那些隐式的声明就是为存储型临时变量而生成的。

15.2.2 分配和释放临时变量

为分配临时变量，我们通常保持一个可用临时变量池或集合。可用寄存器可在一个集合（在C语言中可能是位图）中被维护；存储型临时变量将被保存在一个表中，这个表能够指示它们在活动记录中的大小和偏移。临时变量分配和释放通常遵守先分配后释放（LIFO或栈）的规则。也就是说，仅当R_1，R_2，…，R_{i-1}在使用时才分配R_i。然而，在某些情况下这种栈式规则会遭到破坏，特别是在使用优化的

时候。因为使用临时变量集合的更一般的分配技术非常容易编程实现，所以这是一项被推荐的技术。

在代码生成期间，寄存器临时变量处于以下三种状态之一：未分配（unallocated）、活跃（live）和非活跃（dead）。未分配的寄存器临时变量是那些尚未被指派到某个实际寄存器的临时变量。寄存器指派通常要被推迟，直到涉及该临时变量的代码必须要生成的时候。这种延迟允许引用寄存器的上下文来改变有关的指派。（例如，索引寄存器或奇数编号的寄存器可用来生成一条特定的指令。）

活跃的寄存器临时变量是那些已被分配到某个寄存器中、且其值必须要加以保护的临时变量。类似地，非活跃的寄存器临时变量是那些已被分配、但其值已不再需要的临时变量。

15.3 简单的代码生成器

现在，我们考虑一个简单代码生成器的设计，它使用元组作为IR。我们给每个元组均提供一个代码生成器，从而使全部的代码生成任务模块化。每个元组生成器负责为关联的IR元组生成可能的最佳目标代码。这样做时，它必须执行以下三个子任务：

- 指令选择。
- 地址模式选择。
- 寄存器分配。

这三个任务紧密相联。通常，目标机器指令仅允许使用可用的寻址模式的子集。例如，许多机器允许寄存器到寄存器或存储器到寄存器的加法，但不允许存储器到存储器的加法。这种限制意味着用于访
551 问操作数的地址模式将影响实现元组的指令的选择。类似地，许多指令要求使用寄存器；有时还规定使用特殊的寄存器（例如，索引寄存器或奇偶寄存器对。）如何指派寄存器到临时变量会强烈地影响可被生成的指令的种类。

元组生成器按如下方式组织。寻址模式是那些与数据对象的语义记录中的操作数（字面值、索引、间接、临时变量）相关联的模式。每个元组操作数包含这个信息，我们可以将其映射到硬件寻址模式。实际上，生成器实现的是一个决策表，该表是通过与元组操作数关联的寻址模式的组合来索引的。元组生成器与寄存器分配器相配合将临时变量绑定到实际的寄存器。

为说明可能出现的复杂情况，假设我们有一台具有寄存器、索引和立即寻址模式的机器。寄存器操作数被表示为r。索引地址被表示为d(r)，其中d是一个无符号的偏移值，而r是索引寄存器。立即地址被表示为#s，其中s是一个有符号值。我们将大致描述操作符+的基于元组的生成器，假定此时我们有寄存器到寄存器、存储单元到寄存器和字面值到寄存器的加法。也就是说，一个寄存器或者存储单元的内容或者字面值（源）可被加到一个寄存器中（目的），所求的和被存放在这个目的寄存器中。同时，我们还假定有寄存器的装入和存储指令。

+的生成器取三个操作数描述符并生成合适的代码序列。两个加数可以采用5种寻址模式中的任何一种（字面值、索引、间接、活跃寄存器和非活跃寄存器），且结果可以是4种寻址模式中的任何一个（索引、间接、活跃寄存器和未指派寄存器）。这意味着可能出现100多种不同的组合，而且如果要生成高质量代码，那么几乎每种组合都需要不同的代码序列。

穷举所有的情况不仅乏味、容易出错而且不是空间有效的。图15-1展示了一个更普通的方法，它可以根据操作数的状态建立合适的代码序列。这种方法展示了指令选择、寄存器分配和地址模式是如何相
552 互关联的，即使是在为像加法元组这样简单的元组生成代码时也是如此。

该例程仔细检查操作数模式，并生成高质量的代码。例如，给定(+,T1,10,G)，其中T1是被指派到寄存器r1的非活跃临时变量，G是采用(base, offset)寻址的普通变量，于是我们得到：

```
Add #10,r1
Store offset(base),r1
```

Generate code for integer add: (+,A,B,C)

Possible operand modes for A and B are:
(1) Literal (stored in value field)
(2) Indexed (stored in adr field as (Reg,Displacement) pair)
(3) Indirect (stored in adr field as (Reg,Displacement) pair)
(4) Live register (stored in Reg field)
(5) Dead register (stored in Reg field)

Possible operand modes for C are:
(1) Indexed (stored in adr field as (Reg,Displacement) pair)
(2) Indirect (stored in adr field as (Reg,Displacement) pair)
(3) Live register (stored in Reg field)
(4) Unassigned register (stored in Reg field, when assigned)

(a) Swap operands (knowing addition is commutative)

```
if (B.mode == DEAD_REGISTER || A.mode == LITERAL)
     Swap A and B;  /* This may save a load or store
                       since addition overwrites the
                       first operand. */
```

(b) "Target" the result of the addition directly into C (if possible).

```
switch (C.mode) {
case LIVE_REGISTER: Target = C.reg; break;

case UNASSIGNED_REGISTER:
     if (A.mode == DEAD_REGISTER)
          C.reg = A.reg; /* Compute into A's reg,
                            then assign it to C. */
     else
          Assign a register to C.reg;
     C.mode = LIVE_REGISTER;
     Target = C.reg;
     break;

case INDIRECT:
case INDEXED:
     if (A.mode == DEAD_REGISTER)
          Target = A.reg;
     else
          Target = v2;
          /* vi is the i-th volatile register. */
     break;
}
```

(c) Map operand B to right operand of add instruction (the "Source")

```
if (B.mode == INDIRECT) {
    /* Use indexing to simulate indirection. */
    generate(LOAD,B.adr,v1,"");
    /* v1 is a volatile register. */
    B.mode = INDEXED;
    B.adr = (address) { .reg = v1;
                        .displacement = 0; };
}
Source = B;
```

(d) Now generate the add instruction

```
if (A.mode == LITERAL && B.mode == LITERAL)
   /* "Fold" the addition. */
   generate(LOAD,#(A.val+B.val),Target,"");

else {
   address t;
   /* Load operand A (if necessary). */
   switch (A.mode) {
   case LITERAL:  generate(LOAD,#A.val,Target,"");
                  break;
   case INDEXED:  generate(LOAD,A.adr,Target,"");
                  break;
```

图15-1 +元组的代码生成器

```
        case LIVE_REGISTER:
                        generate(LOAD,A.reg,Target,"");
                        break;
        case INDIRECT: generate(LOAD,A.adr,v2,"");
                        t.reg = v2; t.displacement = 0;
                        generate(LOAD,t,Target,"");
                        break;
        case DEAD_REGISTER:
                        if (Target != A.reg)
                           generate(LOAD,A.reg,Target,"");
                        break;
        }
        generate(ADD,Source,Target,"");
    }
(e) Store result into C (if necessary)

    if (C.mode == INDEXED)
        generate(STORE,C.adr,Target,"");
    else if (C.mode == INDIRECT) {
        generate(LOAD,C.adr,v3,"");
        t.reg = v3; t.displacement = 0;
        generate(STORE,t,Target,"");
    }
```

图15-1 （续）

尽管我们的代码生成器在检查每种情况时相当彻底，但仍然还可以包含更多的情况。例如，对A或B是（或二者都是）文字常量0的情况，就没有加以特殊处理；同样的情况还有在一个元组中同一个操作数的出现不止一次。这些扩展可以在花费更多、更仔细的情况分析的代价后获得。为防止出现令人不愉快的事情，较为明智的做法是使元组生成器尝试各种情况以检查所生成的代码的正确性和最优性。

553
554

将代码生成组织成许多元组生成器的最大好处是代码生成模块化。很容易分离负责任意元组的例程，因此调试或者代码的改进将是直截了当的，尽管它们通常是冗长乏味的。但是这种方法的一个显著缺点是代码生成决策和机器特征的描述将混杂在一起。例如，减法的元组生成器应该和加法所使用的元组生成器类似，但实际上却不相同，因为减法是不可交换的。因此，将机器特征从代码生成的细节中清晰地分离出去是一件值得做的事情。

另外一个问题是，若采用基于逐个元组的代码生成方式，那么在元组之间无法保存任何状态。因此，表达式也许会被不必要地重新计算，会出现冗余的寄存器装入和存储，且寻址模式也没有被充分利用。接下来，我们将研究在代码生成时信息追踪的方法。我们称这种方法是*解释性的*（interpretive），因为我们将“解释”元组，即有时会生成代码而有时仅记住将会生成什么样的代码，我们这样做就是希望找到更有效的方法来产生想要的计算。

15.4 解释性代码生成

解释性代码生成器把即将扩展为真正目标代码的IR看作某个虚拟机的代码。如果IR是标准形式的，那么有可能通过使用P个前端和M个代码生成器在M个目标机器上创建P个程序设计语言的编译器。这种方法由UNCOL（Universal Compiler-oriented Language）引入，它曾被提议作为一种通用的IR形式（Steel 1961）。UNCOL或许有些超前，它被证明是不成功的。最近以来，P-代码和U-代码（Perkins and Sites 1979）已被建议作为适合解释性代码生成的IR形式。

这种代码生成就像典型的宏处理，尽管在解释器中也存在着用某些“状态”来产生较高质量的代码。U-代码因其支持解释性代码生成模型而比（通常是宏展开形式的）P-代码的表示有显著的改进。

当U-代码解释器读U-代码指令时，它更新它的状态，就像普通的CPU那样，而代码的生成则以副作用的形式体现。为了再目标U-代码编译器到新的机器，我们要重新定义与U-代码解释器内部状态的

改变相关联的目标代码序列。目标代码不是与U-代码指令而是与U-代码解释器内部状态相关联，这意味着可以实现目标代码质量上的显著改进。

我们所生成的代码必须在任何执行路径上都是正确的，因此我们将代码生成器的分析工作限制为穿越元组的简单的直线流路径。也就是说，我们将注意力限制在基本块（basic block）中，而基本块是元组的线性序列，除了在基本块的最后，其他地方不含有控制流的分支。一个基本块从其顶部开始，依次 555
执行其中所有的指令，然后以条件或无条件分支指令结束。在基本块中间不允许有分支指令。每个程序可以被表示为一系列由分支指令连接在一起基本块。

在基本块内的优化称为局部优化（local optimization），因为这些优化是由基本块的局部特征决定的，它不受其他基本块或控制流的考虑的影响。在基本块内，我们寻找、识别和消除冗余的计算以及进行地址计算的优化。冗余的计算涉及到一个已计算的表达式的重复计算，或涉及到寄存器的使用以便保持某个活跃值并减少装入和存储操作。

15.4.1 优化地址计算

地址计算在生成的代码中很常见，如果处理不当，它们的计算将会是效率低下的。因此，有必要识别那些建立地址的计算并把它们映射为目标机器所提供的特殊的寻址模式和指令。基本的想法是预先考虑常见的硬件寻址模式，除非在绝对需要时才生成建立地址的代码。例如，我们知道通常存在变址寻址模式（寄存器加偏移），因此可以将地址表示为一个（变量加常量）对。

我们也可以事先考虑其他的寻址模式，例如间接（或延迟）寻址，并且可以通过将一种寻址模式转换为另外一种寻址模式来推迟地址代码的生成。通常，我们可以通过简单地将直接地址变换为间接地址来翻译包含指针或引用型参数的表达式。例如，针对Pascal语句A := P↑ + 1，我们也许会生成：

```
(Fetch Indirect,P,T)
(Add,T,1,A)
```

在预见了间接寻址模式之后，可以通过推迟T的显式计算并将它表示为地址模式(P,indirect)的方式来改进上述元组。（如果间接寻址模式不可用，那么可以模拟这个间接访问，这已在15.3节的示例中出现过。）

就像我们在第11章中见过的，数组引用被非常仔细地翻译以便消除不必要的计算。因此，对A(I)的引用会仅要求将I加到A的常量部分（即：A的起始地址与数组下界的差值）的指令。然而A(I-1)通常比A(I)生成更多的代码。这显然不是最优的，因为“减1”可以被合并到A的常量部分。这个问题在于翻译下标表达式的例程可能不知道它是地址计算的一部分而把它作为普通的计算来处理。因此，有必要为处理表达式的语义例程提供上下文信息，以表明正在计算的是地址还是普通的值。地址可以用我们熟悉的 556
(variable, constant)对来表示，因为我们知道最终硬件寻址模式将提供一个“免费”的地址对加法操作。因而，在翻译A(I-1)时，我们将I-1表示为(I, -1)对，而A表示为(base_reg, constant_part)对。这种表示将所需代码变为：I加到基址寄存器上，其中I可能还要乘上数组元素的大小。也就是说，为计算A(I-1)的地址，我们可以生成：

```
Load    I,reg1
Mult    #element_size,reg1    -- Omit if element_size = 1
Add     base_reg,reg1         -- Usually a Display register
```

A(I-1)的地址是(reg1, constant_part-1 × element_size)。

诸如IBM 360/370系列的机器通过提供“将常量偏移与两个寄存器之和相加而形成地址”的寻址模式来允许对数组引用做进一步的优化，这两个寄存器一个是基址（base）寄存器，另一个则是描述下标的索引（index）寄存器。利用这样的寻址模式，地址必须被表示为三元组形式：(Base, Index, Offset)，而三部分的相加则要推迟到地址形成为止。（这种双寄存器模式不会在所有指令中可用，因此有时还是

需要显式地形成一个普通的(Base, Offset)对。)

VAX体系结构提供了更为详细的称为*位移变址*（indexed displacement）的寻址模式，该模式中包含了被访问对象的类型（byte、word或long）。地址依然被表示为三元组形式：(Base, Index, Offset)，但这里索引(Index)要自动乘上对象的大小（如1、2或4），然后再与基址(Base)和偏移(Offset)成员相加。现在，如果代码生成器延迟那些不成熟的地址计算，那么两个加法和一个乘法就可以隐藏在地址计算中。事实上，这里出现一个一般的计算模式。硬件寻址模式通常很不一致且经常很琐碎，因此，延迟或缓存所有的地址计算将是很有用的。当最后必须访问一个对象时，将调用一个经过特殊设计的“地址专家”来挖掘可用的寻址模式。该“专家”决定什么样的寻址模式是可行的以及如何以最佳方式建立一个到所需对象的访问路径。在获悉许多特殊情况可被充分利用的前提下，该专家卖力地工作以求优化数据访问。

在某些机器（如MC 68000）上地址和操作数寄存器是不同的。对于这样的机器，地址计算的结果必须以地址寄存器为目标。也就是说，尽管地址计算可以利用操作数寄存器，但地址计算的最后一步应当将其结果放到地址寄存器中以便可以立即使用该地址。

指令的选择因指令提供的寻址模式不一致而变得复杂。例如，许多机器要求指令中至少有一个操作数是寄存器。如果操作数当前均不在寄存器中，那么可能需要装入操作。在生成指令前，可以计算把操作数与可用寻址模式相匹配的代价。如果操作是像加法那样可以交换的，那么一个谨慎的代码生成器将
557 会考虑两个操作数的顺序，此时通常要寻找可以导致更好代码生成的那一种顺序。

VAX机器上的指令选择也因为存在两-操作数和三-操作数的指令格式而显得复杂。包含三-操作数的指令常常通过消除额外的装入和存储操作而改进代码的质量。但不幸的是，并非所有的指令都提供两-操作数或三-操作数格式，这也带来了选择上的困难。例如，乘上2的幂次方常常通过移位指令来实现。乘法操作可以出现在两-操作数或三-操作数指令格式中，而算术移位仅有三-操作数的指令格式。因此，就有必要在小而慢的乘法指令与大而快的移位指令之间做出选择。

硬件体系结构有时包含自动递增和自动递减寻址模式，即在寻址操作数时允许索引向上或向下步进。通常，在生成的代码中很难利用自动递增和自动递减，除非它们在源语言中被直接表示出来（如它们在C语言中那样）。一个问题是：它们的操作不对称——自动递增在使用索引之后增加其值，而自动递减在使用索引之前减少其值。正常情况下，源代码以递增顺序遍历数组，这就意味着在用索引访问数组元素的元组之后才能出现增加索引的元组。为此，有必要在基本块中做前向搜索，寻找那个可以作为自动递增而预先计算的加法操作。另一个问题是：自动递增和自动递减常常按照2或4个单元来执行（递增或递减），这反映出字长是2或4个字节的事实。除非我们使用一种借助重复的加法替代乘法的称为*强度削弱*（strength reduction）的优化（参见第16章），否则，那个索引每次递增1、然后再乘上字的大小2或4的事实将可能完全掩盖使用自动递增的可能性。

自动递增和自动递减在访问栈时很有用；通常我们增加栈顶并随后存放值，或者从栈顶拷贝值并随后递减栈顶。为使栈顶操作通过自动递增和自动递减来执行，有必要按照后向生长方式来组织栈。也就是说，栈必须开始于高端地址而向低端地址生长。这就允许使用自动递减来实现入栈操作，用自动递增来实现出栈操作。

有关寻址模式的最后一个问题涉及连接基本块的分支指令。硬件体系结构（如PDP-11和VAX）均提供长（long）和短（short）格式分支。短格式分支在一个字节或字中存储一个带符号的*相对偏移*（relative offset），但长格式分支建立绝对地址（非相对地址）。自然地，短格式分支由于需要的空间少而受欢迎。有人研究了长、短格式分支的最佳选择问题（Szymanski 1978），发现这是一个非常难于选择的问题。其问题在于为一个分支选择长格式或短格式可能影响到其他分支的范围。为获得最优代码，也许需要对所生成的代码做不定次数的穿越（以设置长或短格式的分支）。

通常，我们采用能提供近似最优分支格式的启发式方法。假定基本块没有被代码生成器重新排序

(尽管它们可以被某个单独的优化阶段重排)。我们首先为基本块生成代码，其中的分支指令格式尚未决 558
定。这就允许我们预测分支指令到它的目标的距离，这里假设所有分支均为长格式的。然后，我们用这个预测做出选择长格式还是短格式的最初决定，并更新分支指令与它们目标之间的范围。此外，我们还可以利用已更新的信息，重新处理任何能够转换为短格式的长分支。(这种处理的过程可以重复多次，但通常情况下至多需要一遍修正即可。)

15.4.2 避免冗余计算

在基本块中，通常有可能去判定某个特定的计算是否冗余。不需要重新计算的值称为*公共子表达式*(Common Sub Expression，CSE)。

为识别基本块中的CSE，我们必须能判定某个特定的表达式是否已被计算过，如果已被计算，那么要判断它的值是否已被注销。一个值被认为是注销的，如果重新计算它会产生一个不同的值。给表达式中的操作数赋值就注销了先前已计算的该表达式的值。

只要代码生成器被要求去计算表达式，我们就在相同的基本块中寻找该表达式是否有先前计算过且仍然活跃的值。如果找到了一个，我们就抑制有关代码的生成，取而代之的是复用那个已计算过的值。

我们仍假定使用元组作为IR。正常情况下，在元组一级计算表达式的时候，将指派给该表达式一个新的、惟一的临时变量。现在，我们要求所选择的临时变量名字惟一对应某个特定的运算符-操作数组合。即，给定元组(OP1，A1，B1，T1)和元组(OP2，A2，B2，T2)，如果(OP1=OP2)且(A1=A2)以及(B1=B2)，那么T1=T2。

该约定使发现可能的CSE变得容易，因为它们拥有相同的结果临时变量。该约定也确保了所有的已发现的CSE将共享同一个临时变量。在生成元组时可用简单的哈希方法确定合适的结果临时变量。

冗余计算是指那些计算，即一个临时变量T的值在它被重新计算的前后必须是相同的。如何能决定某个临时变量T属于这种情况呢？我们的方法是基于*值编号*(value numbering)的想法，它是由Cocke和Schwartz(1970)首先提出的。首先考虑涉及数组元素、引用参数、指针等的别名被忽略的情形。(由别名带来的复杂性在本节稍后讨论。)

对基本块中每个不同的程序变量和临时变量，用一个整型值`last_def`指向基本块中给那个变量或临时变量赋值的最新的元组。开始时，`last_def`的值设为0。如果T含有一个地址，那么我们将维护两个值：其中一个是地址，而另一个则是由该地址所引用的对象。通常，在改变存放于临时变量中的地址 559
的同时也改变了通过该地址所引用的值。

我们假定，在处理基本块时，所有生成的元组和所有优化的CSE都是正确的。因此——在忽略别名时——如果条件

```
last_def[A] < last_def[T] && last_def[B] < last_def[T]
```

成立，那么元组(OP，A，B，T)的计算必定冗余，因为从最后一次计算T以来，A和B均未改变。这种情况下，(OP，A，B，T)是CSE且可以被删除，而T中值仍然正确。当然，如果A或B自从最后一次计算T以来已另外赋值，那么T的计算就不再冗余。

在识别出CSE的值之后，我们必须确定：在基本块里，只有在这些值的最后一次使用之后，保存它们的临时变量才能被标记为“非活跃的”。这种预防保证了所需要的值将被保存起来，通常这些值是被保存在寄存器临时变量中。

考虑下面的示例，其中T↑表示通过存放在T中的地址间接引用的值。我们将删除为图15-2中的语句所生成的元组中的CSE。

图15-3中给出了为图15-2中的语句所生成的元组。后缀R表示一个可以被删除的冗余的元组。注意：从表中可以看到，`last_def`的值在某些元组中被改变。符号Ti_a表示Ti中保存的地址，Ti_v则表示通过Ti中

保存的地址所引用的值。由†标记的条目表示如果考虑了涉及数组A的别名，那么有关的值可能要改变。

```
A(I,J)   := A(I,J) + B + C;
A(I,J+1) := A(I,J) + B + D;
A(I,J)   := A(I,J) + B;
```

图15-2 简单的基本块

别名

在为基本块生成元组时，以上描述的处理很容易做到。然而，我们必须记住：别名是个大问题。首先，我们考虑数组元素的别名。给一个带下标的变量赋值也许意味着对另一个下标变量的改变，如果二者均引用相同的数组成员（例如，如果I=J的话，A(I)是A(J)的别名）。

如何处理这种情况呢？简单的方法是保持一张能寻址给定数组元素的所有临时变量的表（例如，在本例中针对数组A中元素的T1、T2和T6）。现在，通过这些临时变量中的任何一个对数组元素的赋值都会改变表中所有这样的临时变量的last_def的值。因此，在前面这个例子中，通过T1、T2或T6对数组元素的赋值将更新三者相应的last_def值，如在图15-3中用后缀†所示。因此，在使用了这个更加细致的算法后，第20号元组不再是计算冗余的。这种改变是由于通过T6（在第15号元组中）给A(I,J+1)的赋值注销了通过T2A(I,J)引用的值。当然，即使在最严格的意义下，也并不是真的需要这个动作，因为A(I,J)和A(I,J+1)根本不会引用同一个单元位置。然而，认识这个相当“深奥”的规则（对任何值的J，J≠J+1）所需的额外的复杂性已远超出了这里使用的分析的能力。

560

			I	J	B	C	D	$T1_a$	$T1_v$	$T2_a$	$T2_v$	T3	T4	T5	$T6_a$	$T6_v$	T7
(1)	(index,A,I,T1)		0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
(2)	(index,T1,J,T2)									2	2						
(3)	(index,A,I,T1)	R															
(4)	(index,T1,J,T2)	R															
(5)	(+,T2↑,B,T3)											5					
(6)	(+,T3,C,T4)												6				
(7)	(:=,T4,T2↑)								7†		7					7†	
(8)	(index,A,I,T1)	R															
(9)	(+,J,1,T5)													9			
(10)	(index,T1,T5,T6)														10	10	
(11)	(index,A,I,T1)	R															
(12)	(index,T1,J,T2)	R															
(13)	(+,T2↑,B,T3)											13					
(14)	(+,T3,D,T7)																14
(15)	(:=,T7,T6↑)								15†		15†					15	
(16)	(index,A,I,T1)	R															
(17)	(index,T1,J,T2)	R															
(18)	(index,A,I,T1)	R															
(19)	(index,T1,J,T2)	R															
(20)	(+,T2↑,B,T3)	R†															
(21)	(:=,T3,T2↑)								21†		21					21†	

图15-3 为图15-2中语句所生成的元组和值编号

在我们给一个形参或由指针引用的对象赋值的时候也会引起别名问题。我们必须以某种方式注销所有潜在的可能无效的CSE。我们所做的最为准确的事情仅仅是注销那些与形参或可能绑定到形参的变量有关的CSE。类似地，对于指针，我们可以只注销那些与所关注的指针或可能引用同一对象的其他指针

有关的CSE。处理别名时要谨慎，以确保能考虑到所有到被关注对象的路径。 561

在16.2.3节中我们将讨论如何确定可与形参构成别名的变量集合。与之类似的技术可以用来确定访问相同堆对象的指针集合。在一遍编译器中，准确判定所有潜在的别名是不现实的。为简单起见，所有包含具有合适类型的变量或堆对象指针的CSE均被注销。尽管这样做我们会丧失某些优化的机会，但这比刚才介绍的分析要简单得多并确实保证了CSE优化是安全的。

过程和函数调用可以出现在基本块中且被视为是一个复合操作，就像它们的过程（或函数）体已在调用点展开一样。当然，我们要考虑调用的副作用。正常情况下，我们只是简单地假设所有的CSE均被注销。在优化编译器中，我们可以做过程间（interprocedural）分析以便查看哪些变量可能被修改并注销相应的CSE。我们将在16.2.3节里详细讨论这种分析技术。

15.4.3 寄存器追踪

到目前为止，我们所描述的代码生成器尚未有效地利用寄存器。特别是有关寄存器内容的信息也少有维护。结果是，“值”被不必要地装入寄存器或存储到内存中。进一步地，我们还假设寄存器分配程序简单地将寄存器绑定到临时变量直到临时变量被释放。当寄存器不得不溢出时（即当寄存器的需求超出所提供的寄存器数时），这种方法就显得过于简单了。

我们将使用一个简单的能追踪基本块中可分配寄存器内容的局部寄存器分配方案，来改进15.3节中那个简单的代码生成器。这种方案允许我们将寄存器分配给那些经常访问的变量或临时变量，它是一种可以有效地减少所需寄存器装入和存入数目的方法。它还设法将存储器到寄存器方式的指令替换为更小更快的寄存器到寄存器方式的指令，从而实现代码大小和速度上的改进。

这种方案的设计简单易用。但它不是最优的，已知有许多代价高且可以为基本块产生最优方案的算法（Horwitz et al., 1966）。而在上一节里，我们最初忽略了别名和子程序调用所带来的影响。

我们从赋值操作以及那些可交换或不可交换的二元运算符的元组中生成代码。我们使用的“机器”BB1（bare-bones 1，裸机1号）有n≥2个可用于分配的寄存器。它包括以下各类机器指令：

```
(a)  Load  Storage,Reg    -- Cost = 2
(b)  Store  Storage,Reg   -- Cost = 2
(c)  OP  Storage,Reg      -- Reg := Reg OP Storage; Cost = 2
(d)  OP  Reg1,Reg2        -- Reg2 := Reg2 OP Reg1; Cost = 1
```

562

格式(a)到(c)的指令均为存储器到寄存器方式的指令且每个需要耗费2个单位生成时间。格式(d)则表示寄存器到寄存器指令，需要1个单位生成时间。存储器地址可以是直接地址或索引地址（寄存器加偏移）。

对每一个操作数寄存器，我们维持该寄存器所包含的变量或临时变量的列表，并称之为寄存器关联表（register association list）。与某个寄存器关联的每一个变量或临时变量都有两个状态标识：

(1) L（活跃）或D（非活跃）。

(2) S（待保存）或NS（不必保存）。

如果变量或临时变量是活跃的，那么在它们的值改变以前，它们将再次在基本块中被引用（也就是说，将活跃变量或临时变量的值保存在寄存器中是值得的）。变量应当总是在基本块的最后被保存，如果它们还没有被保存到内存中（即，如果变量有S标志，那么它将被保存）。临时变量在“非活跃”后通常不需要保存。

活跃状态通常可以通过对基本块进行后向扫描来决定。（也就是说，我们必须缓存基本块，做后向扫描，然后再做前向的分析以分配寄存器并生成代码。）进一步地，如果变量或临时变量的下次引用是对其进行新的赋值，那么它的状态就是(D,NS)，因为在重新定义前它将不会被使用，因而在计算新值前也就没有必要把旧值保存到内存中。对每一个寄存器，我们考虑释放它的代价（即，寄存器“失去”其

当前内容的代价）。该代价被定义为释放与寄存器关联的所有变量或临时变量的单个代价的总和。而从给定的寄存器释放关联的单个变量或临时变量的代价被定义如下：

0 如果它的状态是(D,NS)或(D,S)

（在状态为(D,S)时，变量或临时变量均不再被使用，而且它们必须要被保存，因此，这里释放寄存器并立即进行保存没有什么损失或代价。）

2 如果它的状态是(L,NS)

（需要装入指令将变量或临时变量恢复到寄存器中。）

4 如果它的状态是(L,S)

（需要存储指令来保存值，然后需要装入指令将值恢复到寄存器中。）

寄存器如果不包含（变量或临时变量）关联，则其分配代价为零。我们调用图15-4中定义的例程get_reg()来分配寄存器，该例程分配“最便宜”的可用寄存器。它的返回类型machine_reg只是0和机器的最大寄存器数减1之间的一个小整数。

```
machine_reg get_reg(void)
{
  /*
   * Any register already allocated to the current tuple
   * is NOT AVAILABLE for allocation during this call.
   */

  if (there exists some register R with cost(R) == 0)
      Choose R
  else {
     C = 2;
     while (TRUE) {
        if (there exists at least one register
            with cost == C) {
            Choose that register, R, with cost C that
              has the most distant next reference to an
              associated variable or temporary
            break;
        }
        C += 2;
     }

     Save the value of R for any associated variables or
       temporaries with a status == (L,S) or (D,S)
  }
  return R;
}
```

图15-4 基于代价的寄存器分配程序

get_reg()选择尽可能“便宜”的寄存器，当存在多个有相同的非零代价的寄存器时，它将选择其相关变量或临时变量的下次引用最远的那个寄存器。其理由是，寄存器中的值离它的下次引用越近，越应该被保存在寄存器中。反之，寄存器中的值离下次引用越远，则释放该寄存器不会造成什么直接后果
563 （而将值继续保留在该寄存器中则收效甚微）。下次引用信息可以在用于确定活跃或非活跃状态的相同的后向分析中确定。

设函数get_reg_cost()确定获得一个寄存器所需的最小代价（即，释放由get_reg()所选寄存器的代价）。一旦get_reg()分配了寄存器，我们就生成代码装入所需的变量或临时变量。接着清除该寄存器的原有关联，并根据处理当前元组之后变量或临时变量的活跃情况，将寄存器的当前状态设为(L,NS)或(D,NS)。

564 我们使用在图15-5和图15-6中定义的代码生成例程。

```
Assignment (:=,X,Y):

if (X is not already in a register)
    Call get_reg() and generate a load of X

if (Y, after this tuple, has a status of (D,S) )
     generate(STORE,Y,Reg,"")
else
     Append Y to Reg's association list with a status
        of (L,S)
     /* The generation of the STORE instruction
        has been postponed */


(OP,U,V,W) where OP is noncommutative:

if (U is not in some register, R1)
      Call get_reg() and generate code to load U
else  /* R1's current value will be destroyed */
     Generate any necessary saves of R1's value
     as indicated by the S/MS flag on R1's
     association list

if (V is in a register, R2)
     /* including the possibility that U == V */
     generate(OP,R2,R1,"")
else if (get_reg_cost() > 0 || V is dead after this tuple)
     generate(OP,V,R1,"")
else {
     /*
      * Invest 1 unit of cost so that V is
      * in a register for later use
      */
     R2 = get_reg()
     generate(Load,V,R2,"")
     generate(OP,R2,R1,"")
}

Update R1's association list to include W only.
```

图15-5 赋值语句与不可交换的二元运算符的代码生成器

```
(OP,U,V,W) where OP is commutative:

if (cost((OP,U,V,W)) <= cost((OP,V,U,W)))
      generate(OP,U,V,W);
      /* using noncommutative code generator */
else
      generate(OP,V,U,W);
      /* using noncommutative code generator */
```

图15-6 可交换的二元运算符的代码生成器

如果使用图15-5的算法，我们可以计算为(OP,U,V,W)所生成的代码的代价。它是， 565

```
cost = (U is in a register ? 0
        : get_reg_cost() + 2) /* Cost to load U into R1 */
     + cost(R1)               /* Cost of losing U */
     + (V is in a register || U == V
        ? 1 : 2)      /* Cost of register-to-register */
                      /* vs. storage-to-register */
```

如图15-6所示，我们用这个代价评估来决定在可交换的二元运算符中使用的计算次序。

作为示例，考虑图15-7中的基本块。相应的元组在图15-8中。

```
A   := B * C + D * E;
D   := C + (D - B);
F   := E + A + C;
A   := D + E;
```

图15-7 一个简单的基本块

```
(1)  (*,B,C,T1)        (8)  (+,E,A,T6)
(2)  (*,D,E,T2)        (9)  (+,T6,C,T7)
(3)  (+,T1,T2,T3)      (10) (:=,T7,F)
(4)  (:=,T3,A)
                       (11) (+,D,E,T8)
(5)  (-,D,B,T4)        (12) (:=,T8,A)
(6)  (+,C,T4,T5)
(7)  (:=,T5,D)
```

566

图15-8 与图15-7基本块相对应的元组

假设有4个寄存器且使用一个不监控寄存器内容的简单代码生成器。我们为图15-8中元组生成的代码如图15-9所示。

```
(1)  (Load  B,R1)      (12) (Load  E,R1)
(2)  (*     C,R1)      (13) (+     A,R1)
(3)  (Load  D,R2)      (14) (+     C,R1)
(4)  (*     E,R2)      (15) (Store F,R1)
(5)  (+     R2,R1)
(6)  (Store A,R1)      (16) (Load  D,R1)
                       (17) (+     E,R1)
(7)  (Load  D,R1)      (18) (Store A,R1)
(8)  (-     B,R1)
(9)  (Load  C,R2)
(10) (+     R1,R2)
(11) (Store D,R2)
```

图15-9 没有寄存器追踪的代码生成

这个代码序列用了6次装入、4次存储、6次存储器到寄存器操作和2次寄存器到寄存器操作。该序列的全部代价为34。

使用寄存器追踪的代码生成在图15-10中给出。(符号⇐标记为可交换操作所选择的操作数序列。)寄存器追踪方法生成5次装入、3次存储、1次存储器到寄存器操作和7次寄存器到寄存器操作。所生成序列的全部代价为25而不是34——这是一个显著的改进。

别名和子程序调用的影响

在追踪基本块中寄存器的内容时，必须要考虑别名和子程序调用的影响。

我们首先考虑别名的影响。设N为数据对象的别名。它可以是一个引用形参、指针或带有非常量下标的变址变量（数组元素）。我们可用16.2.3节中的技术非常仔细地计算出与N别名的数据对象集合O。O也可能简单地/就是所有的变量、堆对象或与N相对应的数组元素的集合（例如，所有具备正确类型的变量或给定数组中的所有元素）。

无论何时，只要引用N的值，我们就必须检查寄存器关联表。如果任意数据对象o ∈ O在寄存器关联表中出现且状态为S，那么相应寄存器必须要被保存到对象o中。如果N实际与o别名，那么此过程确保N将引用正确的值。

567
~
568

类似地，当对N进行赋值时，我们也必须检查寄存器关联表。如果对象o ∈ O在寄存器关联表中，那么应当从关联表中删除该对象。它反映出这样的事实：即对N的赋值可能已改变o的值，从而使当前保存在与o关联的寄存器中的值无效。

就如第13章所讨论的，通常可在子程序调用前后，由调用者或被调者保存和恢复可分配的寄存器。(在16.2.2节中，我们将讨论避免不必要的寄存器保存的方法)

如果调用者做保存与恢复的工作，将有益于在调用前清除所有的寄存器关联。也就是说，状态为S的寄存器将被保存，而其他所有寄存器将被释放。在返回时，那些所需要的寄存器的值可被逐一重新装

入。最坏情况是，我们保存某个寄存器并稍后再次装入它。尽管如此，我们仍然可以做得更好些，因为状态为NS的寄存器不需要保存，而且寄存器也只有在实际需要的时候才会被重新装入。

Tuple/Code generated	Register Associations			
	R1	R2	R3	R4
(*,B,C,T1) Cost(*,B,C,T1) = 2+2+2 ⇐ Cost(*,C,B,T1) = 2+2+2				
(Load B,R1) (Load C,R2) (* R2,R1)	B(L,NS) B(L,NS) T1(L,S)	C(L,NS) C(L,NS)		
(*,D,E,T2) Cost(*,D,E,T2) = 2+2+2 ⇐ Cost(*,E,D,T2) = 2+2+2				
(Load D,R3) (Load E,R4) (* R4,R3)	T1(L,S) T1(L,S) T1(L,S)	C(L,NS) C(L,NS) C(L,NS)	D(L,NS) D(L,NS) T2(L,S)	 E(L,NS) E(L,NS)
(+,T1,T2,T3) Cost(+,T1,T2,T3) = 0+0+1 ⇐ Cost(+,T2,T1,T3) = 0+0+1				
(+ R3,R1) -- (D,NS) associations -- can be immediately removed	T3(L,S)	C(L,NS)	T2(D,NS)	E(L,NS)
(:=,T3,A) -- The store is deferred	A(L,S)	C(L,NS)		E(L,NS)
(-,D,B,T4)				
(Load D,R3) (- B,R3) -- B is not live after this tuple	A(L,S) A(L,S)	C(L,NS) C(L,NS)	D(D,NS) T4(L,S)	E(L,NS) E(L,NS)
(+,C,T4,T5) Cost(+,C,T4,T5) = 0+2+1 Cost(+,T4,C,T5) = 0+0+1 ⇐				
(+ R2,R3)	A(L,S)	C(L,NS)	T5(L,S)	E(L,NS)
(:=,T5,D) -- Store is deferred	A(L,S)	C(L,NS)	D(L,S)	E(L,NS)
(+,E,A,T6) Cost(+,E,A,T6) = 0+2+1 Cost(+,A,E,T6) = 0+0+1 ⇐ -- A is dead after this				
(+ R4,R1)	T6(L,S)	C(L,NS)	D(L,S)	E(L,NS)
(+,T6,C,T7) Cost(+,T6,C,T7) = 0+0+1 ⇐ Cost(+,C,T6,T7) = 0+0+1				
(+ R2,R1)	T7(L,S)	C(D,NS)	D(L,S)	E(L,NS)
(:=,T7,F) (Store F,R1) -- Do store since F is not -- live in this block	T7(D,NS)		D(L,S)	E(L,NS)
(+,D,E,T8) Cost(+,D,E,T8) = 0+0+1 ⇐ Cost(+,E,D,T8) = 0+0+1				
(Store D,R3) -- Store is unavoidable (+ R4,R3)			D(L,NS) T8(L,S)	E(L,NS) E(D,NS)
(:=,T8,A) (Store A,R3) -- Store is unavoidable				

图15-10 带有寄存器追踪的代码生成

如果由被调者来保存和恢复寄存器，那么在子程序被调用的时候，我们将检查Def和Use这两个集合。Def和Use分别反映子程序调用期间所定义（所更新的）和使用（所读的）的变量集合。它们可以用16.2.3节中的技术来计算，或简单地设为该子程序能访问的所有变量的集合。在调用前，我们保存所
569 有出现在寄存器关联表中状态为S的对象o ∈ Use。类似地，我们将从寄存器关联表中删除所有对象o ∈ Def。也就是说，我们保存那些在调用过程中将被引用的值而去除调用期间可能会被赋值语句修改的变量关联。

在最简单的情况下，我们假设在调用过程中所有变量可读可写，那么这种方法就意味着在调用前保存所有状态为S的寄存器并在调用后完全清除所有的关联表。幸运的是，如果调用的是非局部的子程序，那么局部变量显然不受调用的影响。因此，我们可以确信库子程序（输入/输出、存储管理、数学函数等）不会影响到程序变量，除非使用显式的引用参数。

其他有关寄存器追踪的问题

前面提及的寄存器追踪方法代表着一类局部寄存器分配例程。此外还有许多可能的变形和扩展：

- 有算法建议溢出下次引用在最远处的寄存器（Kim 1978）。同样，还有算法已研究了对寄存器的下两次引用（Hsu 1987）。这种方法比只考虑下次引用会产生更好的代码。

  可以使用*着色算法*（coloring algorithm）（Chaitin 1982）来执行寄存器分配。首先建立*相干图*（conflict graph），把图中结点赋予即将指派到寄存器的变量或临时变量。如果两个结点所代表的对象（变量或临时变量）必须共存，则在它们之间有一条边。即，相连接的结点不能分配到相同的寄存器。寄存器分配变为图结点的着色问题：每种颜色代表一个寄存器，相连接的结点颜色不同。如果相干图所需颜色数比寄存器总数要多，就必须溢出寄存器。已有各种启发式方法通过将相干图拆分成子图来做“着色”工作。
- 对各种不同的操作，根据其指令大小或执行时间使用不同的代价描述。
- 可以包含额外的地址模式（直接、间接、变址+基址、等等）。
- 允许不同寄存器分类和分配相邻寄存器对。
- 允许使用寄存器到寄存器传送来保护寄存器内容。例如，针对(C−A)+(C+B)，我们可能产生：

570

```
(Load   C,R1)
(-      A,R1)
(Load   C,R2)
(+      B,R2)
(+      R2,R1)
```

其代价为9。而使用寄存器传送，更好的代码是：

```
(Load   C,R1)
(Move   R1,R2)
(-      A,R1)
(+      B,R2)
(+      R2,R1)
```

其代价仅为8。

- 允许包含状态为(L,NS)的变量或临时变量的寄存器的释放代价仅为1，条件是它的值可由一个存储器到寄存器指令从存储器中引用到（即，可以避免重新装入其值）。类似地，对于状态为(L,S)的变量，如果我们保存它，并稍后用一个存储器到寄存器指令引用其值，那么可能需要的释放代价仅为3（而不是4）。
- 如果进行*窥孔优化*（peephole optimization）（使用逻辑上相邻指令；参见15.5节），那么可以将指令对(Load V,Rj)，(OP Rj,Ri)替换为(OP V,Ri)，其条件是在后续指令中不从Rj中引用V的值。这种指令的合并一般发生在以下情况下：我们因V活跃而将其值装入Rj、但稍后在遇到V的引用前又

强行释放Rj另作他用。

上述讨论中的最重要一点是，我们可以建立基于代价的寄存器分配的形式化模型，即使该模型的使用范围及细节相当简单，它也能产生显著的代码改进。如果有可用的寄存器，它将“付费使用”它们。

在全局范围内我们如何进行寄存器分配呢？通常我们会假定在基本块之间没有需要保留的寄存器值（即，我们做的是局部寄存器优化）。然而，我们可以在基本块间提供有限制的寄存器使用：

- 特殊的操作数（如循环索引或过程参变量）可在循环或过程中被分配到固定的寄存器中。
- 可以向前传输寄存器状态信息到其前驱惟一的基本块。前驱惟一的基本块经常出现在**if**和**case**语句中，其中每个条件分支均有惟一的前驱。然而，在穿越基本块边界时延迟寄存器的保存并非好主意，因为我们可能在每个后继基本块而非单个的前驱基本块中做保存（这浪费了些空间，但时间不受影响）。 571

优化编译器也许会施行完全的全局寄存器分配。但此类优化要做得很好却非常困难，因为：

- 我们必须允许在所有基本块中用相同寄存器保存某个给定数据对象或做额外的传送。
- 在可能以数千计的变量和临时变量中，必须决定在小得多的寄存器集合中保存其中的一部分。这种决策涉及到控制流分析，而评估给定变量和临时变量引用频率的机制却使之复杂起来。其答案也往往带有整数程序设计问题的味道（Johnsson 1975）。

15.5 窥孔优化

为产生高质量代码，有必要了解众多的特殊情况。例如，很明显我们希望避免生成操作数与零相加的代码。但我们在哪里检查这个特殊情况呢？在每一个可能生成加法元组的例程里？或在每一个可能生成加法操作的代码生成例程里？

与其将特殊情况的知识散布在语义例程或代码生成例程中，还不如借助明显的*窥孔优化*（peephole optimization）阶段以寻找那些特殊的情况并将它们替换为改进后的代码。我们可以在元组（Tanenbaum et al., 1982）或已生成的代码（McKeeman 1965）上进行窥孔优化。如同“窥孔”一词所暗示的，我们将检查包含2～3条指令或元组的小窗口。如果在窥孔中的指令匹配特定的模式，它们将被*替换序列*（replacement sequence）所取代。替换后，新的指令将被重新考虑以便进行进一步的优化。

一般地，我们将定义窥孔优化器的众多特殊情况表示为“模式-替换”对的列表。因此，pattern ⇒ replacement就意味着如果发现一个代码或元组序列匹配模式，那么它将被替换序列所代替。如果没有任何模式可以匹配，那么该代码序列将保持不变。很明显，可以考虑的特殊情况的数量几乎是无限的，而我们这里仅仅列出了几类最常用的替换规则。通常，在元组一级表示能施用于任何机器体系结构的规则；而利用特定指令或寻址模式的规则则一般在机器-代码一级来表示。

- 常量折叠（提前计算常量表达式）。

```
(+,Lit1,Lit2,Result)                          ⇒ (:=,Lit1+Lit2,Result)
(:=,Lit1,Result1), (+,Lit2,Result1,Result2)   ⇒ (:=,Lit1,Result1),
                                                (:=,Lit1+Lit2,Result2)
```

572

- 强度削弱（将较慢的操作替换为较快的等价操作）。

```
(*,Operand,2,Result)   ⇒ (ShiftLeft,Operand,1,Result)
(*,Operand,4,Result)   ⇒ (ShiftLeft,Operand,2,Result)
```

- 空序列（删除无用操作）。

```
(+,Operand,0,Result)   ⇒ (:=,Operand,Result)
(*,Operand,1,Result)   ⇒ (:=,Operand,Result)
```

- 合并操作（将若干操作替换为一个等价的操作）。

```
Load A,Ri; Load A+1,Ri+1          ⇒ DoubleLoad A,Ri
BranchZero L1,R1; Branch L2; L1:  ⇒ BranchNotZero L2,R1
Subtract #1,R1; BranchZero L1,R1  ⇒ SubtractOneBranch L1,R1
```

- 代数规则（利用代数规则来简化或重排指令）。

```
(+,Lit,Operand,Result)   ⇒ (+,Operand,Lit,Result)
(-,0,Operand,Result)     ⇒ (Negate,Operand,Result)
```

- 特殊情形指令（利用为特殊操作设计的指令）。

```
Subtract #1,R1            ⇒ Decrement R1
Add #1,R1                 ⇒ Increment R1
Load #0,R1; Store A,R1    ⇒ Clear A
```

- 地址模式操作（利用地址模式来简化代码）。

```
Load A,R1; Add 0(R1),R2     ⇒ Add @A,R2
                                -- @A 表示间接寻址
Subtract #2,R1; Clear 0(R1) ⇒ Clear -(R1)
                                -- "-(Ri)" 表示自动递减
```

为减少替换规则的数量，可以使用能匹配任何操作符和操作数的模式变量（pattern variable）。例如，如果模式变量被冠以%前缀，那么我们可以指定单一的模式来识别可能的间接寻址的应用：

```
Load %IndirAdr,%VolatileReg; %OpCode 0(%VolatileReg),%ResultReg ⇒
   %OpCode @%IndirAdr,%ResultReg
```

573

为使模式匹配更快地进行，我们将操作符-操作数组合种类散列（映射）到可用模式。同时，窥孔窗口的大小通常也限制在2－3条指令。经过精心的哈希实现，可取得每秒钟几千条指令的处理速度（Davidson and Fraser 1984）。

这种分析思想已从实际相邻的指令推广到逻辑相邻的指令（Davidson and Fraser 1982）。如果两条指令通过控制流相连接或它们不受插入它们之间的指令的影响，则称这两条指令*逻辑上相邻*（logically adjacent）。通过分析逻辑上相邻的指令，有可能删除跳转链（到转移指令的跳转）和冗余计算（例如，不必要地设置条件码）。寻找逻辑上相邻的指令比较耗时，因此需要谨慎处理以保持窥孔优化的快速执行。

15.6 从树结构生成代码

我们已集中讨论了从IR元组生成代码。在处理表达式的时候，所翻译的元组是表达式树形结构的线形表示。可以用不同的顺序遍历并翻译表达式树；通常，使用深度优先、从左自右的方法产生元组。深度优先、从左自右的方法总能产生正确的翻译；然而，其他的遍历方式或许可以生成更好的代码。考虑(A－B) + ((C+D)+(E*F))。通常的深度优先遍历首先翻译(A-B)，将其值保留在寄存器中。然后再翻译(C+D)+(E*F)，这需要两个寄存器（每个子表达式一个）。因此，总共使用了三个寄存器。然而，如果右边的表达式((C+D)+(E*F))被首先计算，那么仅需要两个寄存器，因为这个子表达式一旦被计算，其值将保存在一个寄存器中，而另外一个可用于计算A+B。

树中结点有如下结构：

```
typedef struct expr_tr {
   struct expr_tr *left_subtree;
   struct expr_tr *right_subtree;
   int reg_count;
   boolean is_right;
   enum { ID, BINARY_OPERATOR,
          COMMUTATIVE_OPERATOR } kind;
} expression_tree;
```

现在我们考虑可以在计算任意表达式或子表达式时确定需要寄存器数最少的算法。我们暂时忽略公共子表达式和运算符的特殊性质（如交换性）。该算法在树中结点的边上标记用于计算以该结点为根的

子表达式所需最少的寄存器数目。该标记也称为Sethi-Ullman编号（Sethi-Ullman 1970）。一旦知晓计算 574
每个表达式或子表达式所需最少的寄存器数，我们将以产生最优代码的方式遍历相应的（子）表达式树（即，产生的代码使用寄存器最少，且因此寄存器的溢出也最少）。

和前面几节一样，我们假设的机器模型可提供寄存器到存储器以及存储器到寄存器的操作数模式，且第一操作数和结果总驻留在寄存器中。

算法以自底向上的方式工作，首先标记树的叶结点。如果叶结点是左操作数，它将被标记为1，因为它必须被装入到寄存器中；反之，它若是右结点则标记为0，因为它可以直接从内存中访问。对于代表二元操作的内部结点，必须考虑每个操作数的寄存器需求。如果两个操作数均需要r个寄存器，那么该运算需要r+1个寄存器，因为一旦一个操作数计算完成，其值将保存在一个寄存器中。如果两个操作数需要不同数目的寄存器，那么整个表达式所需寄存器数和两个操作数中较复杂的那个操作数所需寄存器数一样多。（首先计算较复杂的操作数并将其值存放在一个寄存器中。较简单的操作数需要少一些的寄存器，因此可以复用先前计算较复杂操作数所用的寄存器。）这种分析算法如图15-11所示。

```
void register_needs(expression_tree *T)
{
   /*
    * Mark each node in T with a field "reg_count".
    * Node.reg_count is the minimum number of registers
    * needed to evaluate the subexpression rooted by
    * Node in T.
    */
   if (T->kind == ID) {
       if (is_a_right_subtree(T))
           T->reg_count = 0;
       else
           T->reg_count = 1;
   } else {      /* T must be a binary operator. */
       register_needs(T->left_subtree);
       register_needs(T->right_subtree);
       if (T->left_subtree.reg_count ==
          T->right_subtree.reg_count)
          T->reg_count = T->right_subtree.reg_count + 1;
       else
          T->reg_count = max(T->left_subtree.reg_count,
                             T->right_subtree.reg_count);
   }
}
```

图15-11 为表达式树标记所需寄存器的算法

作为该算法的示例，图15-12中给出了register_needs()对(A-B) + ((C+D)+(E*F))的表达式树所 575
做的标记（括号里是每个算符结点所需的寄存器数reg_count）。

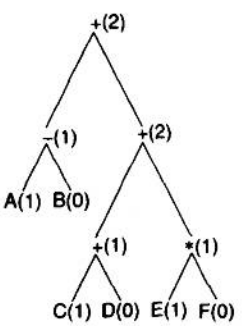


图15-12 已标记所需寄存器数的(A-B) + ((C+D)+(E*F))的表达式树

我们用reg_count标记来驱动一个简单、但是最优的代码生成器tree_code()，其定义见图15-13。tree_code()取带标记的表达式树和可用寄存器列表为输入参数。它生成计算该树的代码并将结果存于列表中的第一个寄存器里。如果tree_code()能使用的寄存器太少，它将在必要时溢出寄存器到临时存储器中。我们用到一些简单的表操纵函数[⊖]，限于篇幅，这里并未将它们列出。）

```
void tree_code(expression_tree *T, register_list reglist)
{
  /*
   * Generate code to evaluate T using registers in
   * reglist (at least two registers are assumed).
   */
  extern register_list listcat(register_list,...);
  expression_tree *left_t, *right_t;
  machine_reg R1, R2;
  register_list remaining_regs;
  address temp;

  R1 = head(reglist);
  if (T->kind == ID)
     /* Must be a left subtree or trivial expression. */
     generate(LOAD,T,R1,"");
  else {  /* T->kind must be a binary operator. */
     left_t = T->left_subtree;
     right_t = T->right_subtree;
     if (right_t->reg_count == 0) { /* right_t is an ID. */
        tree_code(left_t,reglist);
        generate(T->op,right_t,R1,"");
     } else if (left_t->reg_count >= length(reglist) &&
           right_t->reg_count >= length(reglist)) {
        /* Must spill a register */
        tree_code(right_t,reglist);
        get_storage_temp(& temp);
        generate(STORE,temp,R1,"");
        tree_code(left_t,reglist);
        generate(T->op,temp,R1,"");
     } else {
        /* One or both subtrees don't need all registers */
        R2 = head(tail(reglist));
        if (left_t->reg_count >= right_t->reg_count) {
           tree_code(left_t,reglist);
           tree_code(right_t,tail(reglist));
           generate(T->op,R2,R1,"");
        } else {
           remaining_regs = tail(tail(reglist));
           tree_code(right_t,listcat(R2,R1,remaining_regs));
           /* Leave result in R2. */
           tree_code(left_t,listcat(R1,remaining_regs));
           /* Leave result in R1. */
           generate(T->op,R2,R1,"");
        }
     }
  }
}
```

图15-13 从表达式树产生代码的算法

作为tree_code()程序的示例，我们以图15-12中的标记树和寄存器表(R1,R2)调用该例程，得到如下代码序列：

```
(1)  Load   C,R2
(2)  Add    D,R2
(3)  Load   E,R1
(4)  Mult   F,R1
(5)  Add    R1,R2
(6)  Load   A,R1
(7)  Sub    B,R1
(8)  Add    R2,R1
```

⊖ 如图15-13所示代码中的tail()、listcat()等。——译者注

tree_code()很好地诠释了以寄存器为目标（register targeting）的原则。也就是说，所生成的代码中，最终值将出现在目标寄存器中，没有不必要的传送。 576 ~ 577

如果能利用运算的可交换性，还可以改进由例程tree_code()生成的代码的质量。考虑运算可交换的情况，设子树为T1和T2。如果T1和T2均为标识符，则交换操作数显然对代码没有改进。类似地，如果T1和T2都是非平凡的表达式，则交换操作数也不会有改进，因为较复杂的表达式子树总是先被计算且两个子树均将结果存放在寄存器中。然而，当左子树T1是标识符、右子树T2为非平凡表达式的时候，交换操作数就是有益的。这是因为在原来的计算方式里，这个左操作数必须要被装入到寄存器中，而交换后，它可以从内存中访问到。

基于这种分析，我们可以定义如图15-14所示的例程commute()；它递归遍历表达式树并在发现有利于代码改进的时候交换相应的操作数。在例程commute()执行后，可以跟先前那样使用register_needs()和tree_code()。

```
void commute(expression_tree *T)
{
    /* Commute subtrees in T, where advantageous. */

    if (T->kind == BINARY_OPERATOR) {
        commute(T->left_subtree);
        commute(T->right_subtree);
        if (T->left_subtree.kind == ID
            && T->right_subtree.kind == BINARY_OPERATOR
            && commutative(T->op))
                swap(T->left_subtree, T->right_subtree);
    }
}
```

图15-14 可交换运算符的代码生成算法

15.7 从dag生成代码

在15.6节里，我们集中讨论了为简单表达式产生最优代码的问题。而在这节里，我们考虑为整个基本块产生代码这一更具挑战性的问题。为此，我们需要处理公共子表达式和赋值语句。我们也因此将表达式树推广到计算dag（computation dag）。dag是有向无环图——即，一种推广的树结构，其中的结点可以有多个父结点。而允许有多个父结点则使得表达式能够使用多次。环是不允许的，因为它们可能在计算表达式时导致无限循环。没有父结点的结点称为根结点；一般来讲，dag可以有多个根结点。 578

图15-15中显示了与(A+B)+(A+B)对应的dag。它说明了dag是如何表示共享的公共子表达式的。为表示赋值语句，可以允许算符结点被标以变量名。这种标记方法表示以算符结点为根的表达式的值已赋给那个变量。例如，我们在图15-16中所表示的语句I:=I+1的dag；注意，其中的I出现不止一次。（每棵树均是一个平凡的dag。）如果用粗黑字体对变量的当前值突出表示，则图15-16中的dag就变成了图15-17中的样子。



图15-15 与(A+B)+(A+B)对应的dag

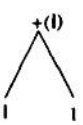


图15-16 与I:=I+1对应的dag



图15-17 与I:=I+1对应且带有变量突出表示的dag

579 在整个基本块的dag中，一个变量可以出现多次。其初值总表示为叶结点，而它的那个必须被保存的最终值则用粗黑体表示。如果变量未被赋值，那么它只出现一次（作为叶结点）且因为其值不需保存而没有被突出表示。set_label(node, label)将label（以突出表示的形式）指派给结点node，并删除dag中label的其他突出表示。

现在我们给出以组成基本块的赋值语句列表为输入、产生基本块计算dag的算法。假设每个赋值语句的形式为 ID := Expr，其中Expr是代表语句右部的子树。

首先，我们定义图15-18中的函数look_up()，它以一个树结点（可以是字面值或变量）为参数，返回相应的dag结点。利用look_up()，我们可以确定（或创建）与变量和字面值对应的dag结点。如果某变量已被多次赋值，那么look_up()将返回和突出显示最近一次赋值的值。

```
dag_node *look_up(tree_node *T)
{
  if (T is a literal) {
    if (a dag node labeled with T exists)
       Return a pointer to the node
    else
       Create a new dag node, label it with T
          and return a pointer to it
  } else {      /* T is a variable */
    if (no dag node labeled with T exists)
       Create a new dag node, label it with T
          and return a pointer to it
    else if (a highlighted node labeled with T exists)
       Return a pointer to the node
    else
       Return a pointer to the nonhighlighted
          node labeled with T
    }
}
```

图15-18 将树结点映射为dag结点的算法

在图15-19中定义的tree_to_dag()，以树为参数并将其合并到已有的dag结构中。在做的过程中，该函数提供公共子表达式的共享和赋值的效果。一个与树的值对应的dag结点指针将被返回。

```
dag_node *tree_to_dag(tree_node *N)
{
  dag_node *RHS, *l_opnd, *r_opnd, *p;

  if (N is a leaf)    /* must be a literal or variable */
      return look_up(N);
  else if (N is a ":=" operator) {
      RHS = tree_to_dag(r_sub_tree(N));
      set_label(RHS, l_sub_tree(N));
      return RHS;
  } else {
      /* N is a binary operator
         (the unary case is analogous). */
      l_opnd = tree_to_dag(l_sub_tree(N));
      r_opnd = tree_to_dag(r_sub_tree(N));
      if (there exists a dag node, P, that is the
         same operator as N and has l_opnd and
         r_opnd as left and right descendants)
            return P;
      else {
            Create a new dag node, P, that is an
               N-type operator and has l_opnd and
               r_opnd as left and right descendants
            return P;
      }
  }
}
```

图15-19 将树合并到dag结构中的算法

作为示例，考虑图15-20所示的基本块。对应的表达式树见图15-21。用tree_to_dag()依次合并图15-21中的每棵树便得到图15-22中的三个计算dag。

580

```
G   := C*(A+B)+(A+B);
C   := A+B;
A   :=(C*D)+(E-F);
```

图15-20 一个简单的基本块

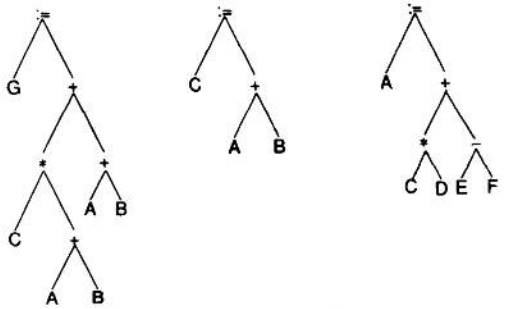


图15-21 图15-20中基本块的表达式树

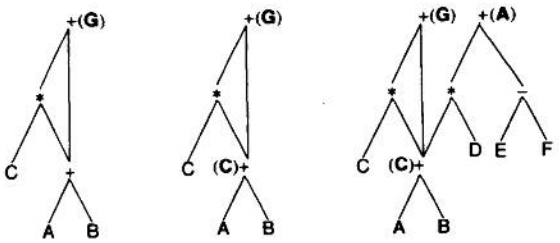


图15-22 图15-20中基本块的计算dag

有突显标记的操作符必须在基本块结尾保存它们的值。可被共享的值（像A+B）在dag中有不止一个父结点使用其值。

显然，从dag生成代码比从树产生代码要复杂得多。一个明显的问题是：如果一个值要被共享，那么其值就必须被一直保存在寄存器中直到所有使用该值的代码均被生成为止。因此，如何减少寄存器的使用就成了问题。令人惊讶的是，问题还不仅仅如此。事实上，Aho、Johnson和Ullman（1977）已证明，即使寄存器的数目无限多，最优代码的产生仍然十分复杂（目前已知的最好的算法花费dag大小的指数阶时间）。这个问题在于，用作左操作数的值通常会被“破坏”。如果一个值要被共享，那么在它用作左操作数前，其值必须被强制拷贝。优化代码的生成器也因此不得不排序dag的计算以减少寄存器拷贝。

581

我们将给出一种启发式算法，它一般可以产生dag的有效翻译。此算法不难理解。我们将尽量排序dag的计算以便操作数的上一次（last time）使用是作为左操作数来使用。此过程显然优于拷贝操作数的值：先将该操作数用作左操作数，然后再用拷贝的值作为右操作数。

582

我们定义一个名为schedule()的dag遍历算法，它调度dag中操作符的计算顺序。该例程以自顶向下的方式工作；首先被调度的操作符将是最后一个被计算的（在一个操作符被计算前，其所有的操作数必须要被全部计算完成）。

因为dag可能有多个根结点，所以我们自右向左、从最右边的根结点开始依次调度各个子dag。（结

点是从左至右地被添加到dag中，因此最右的根结点是最后被创建并加入dag中的。）我们自右向左调度是因为第一个被调度的子dag将最后被计算，而最右的子dag往往对应于基本块中最后一条语句。我们希望所生成的代码能大致和得到这些代码的语句对应。

在子dag中，如果一个操作的所有父结点都已被调度过，此时即可调度该操作。这一点保证了所有操作数在使用前将被计算。我们调度左操作数先于右操作数，因为我们需要左操作数最后被计算，而这种调度恰好代表计算次序的逆序。因此，我们有如图15-23所示的算法，它从子dag的根开始递归调度。

```
void schedule(dag_node D)
{
     if (D is an operator all of whose parents
         have been scheduled) {
          /* if D is a root, then we assume it
             is immediately schedulable. */
          Mark D as the next node scheduled
          schedule(left_operand(D));
          schedule(right_operand(D));
     }
}
```

图15-23 调度dag结点的算法

我们通过以算符结点被调度的次序来编号这些算符结点，可以描述调度的过程。对于图15-22中的dag，其调度后的结果如图15-24所示。

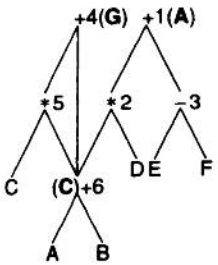


图15-24 算符结点调度后的dag

在刚才的介绍中，根结点的调度是自右向左；dag的计算次序是所选调度的逆序。在代码生成前，必须分配寄存器。此项工作分两步进行。首先，分配虚拟寄存器；然后再将它们映射到实际的寄存器。

我们使用的寄存器分配程序allocator_v_regs()，如图15-25所示。它试图将一个操作符和它的左操作数映射到相同的寄存器。该过程允许左操作数被使用（或被撤销）而无需拷贝其值。

583

```
void allocate_v_regs(void)
{
   /* Allocate virtual registers. */
   int v_reg = 1;      /* Current virtual register */
   dag_node N;
   operator left_op;

   while (all operator nodes have not been
          allocated a virtual register) {
      Choose N, the operator node with the lowest
        schedule number still unallocated
      while (TRUE) {
         Allocate v_reg to N;
         left_op = left_operand(N);
         if (left_op is an operator not yet allocated
             a virtual register and left_op's lowest
             schedule-numbered parent is N)
               N = left_op;
         else
               break;
      }
      v_reg++;
   }
}
```

图15-25 分配虚拟寄存器的算法

allocator_v_regs()试图反复地将同样的虚拟寄存器指派给一个操作符及其所有的左操作数，只要上次使用该操作数的是这个操作符即可（也就是说，只要该操作符有最小的调度编号）。再回到图15-24，我们分配虚拟寄存器后并得到如图15-26所示的dag，其中虚拟寄存器用V1、V2、…表示。 584

将虚拟寄存器映射为真实寄存器很容易。我们定义一个虚拟寄存器的使用范围是指派到或使用这个虚拟寄存器的操作符的调度编号的范围。以图15-26为例，其映射表如图15-27所示。

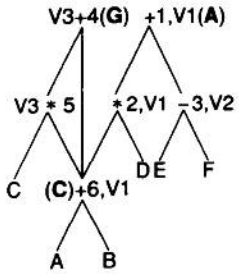


图15-26 分配虚拟寄存器后的dag

Register	Span
V1	1-6
V2	1-3
V3	4-5

图15-27 虚拟寄存器的使用范围

两个虚拟寄存器可以被映射到同一个硬件寄存器，当且仅当它们的使用范围不重叠。因此，在我们的示例中，V1可被映射到R1，而V2和V3可被映射到R2。

在给出计算次序和寄存器指派后，代码生成也算基本上完成了。余下的工作就是产生代码来保存那些在基本块中被修改的变量的值。我们已用突显的方式标记出那些接受新值的变量。如果结点N旁突显地标记有变量V，那么必须把分配给N的寄存器保存到V。但这样做存在一定的风险。如果我们过早地保存了那个值，那么我们就有可能在所有需要这个初值的操作符尚未被计算之前破坏了该变量的初值。例如，对C值的更新在编号为6的结点处完成，但C的初值在计算结点5时仍然需要。因此，我们推迟保存寄存器中的值直到它可以被丢掉为止。如果一个值在结点N处计算，它将被一直保留到该结点的最小编号的父结点P_{min}被计算为止。如果使用变量初值的所有操作的调度编号都比P_{min}大，那么仅在计算P_{min}前保存那个更新的值。否则，那个更新的值可被保存到另一个寄存器或临时存储器中，并在计算dag的代码的最后再被拷贝回变量中。 585

在我们的示例中，C的新值在结点6处被计算并被存于寄存器R1中，该值将被保留到结点2被计算为止。因为对C的初值的最后一次引用在结点5处，因而我们可以在计算结点2之前将R1直接保存到C。

为该示例生成的代码如图15-28所示，其中用到了调度、寄存器分配和刚刚讨论的变量更新。

我们所用的启发式调度算法有时由于操作数之间存在着共享而未必是最优的。例如，考虑图15-29中的两个调度后的“钻石”形dag。

```
(1)   Load   A,R1
(2)   Add    B,R1
(3)   Load   C,R2
(4)   Mult   R1,R2
(5)   Add    R1,R2
(6)   Store  G,R2
(7)   Load   E,R2
(8)   Sub    F,R2
(9)   Store  C,R1
(10)  Mult   D,R1
(11)  Add    R2,R1
(12)  Store  A,R1
```

图15-28 图15-26中dag的代码生成

左边dag的调度是次优的，因为结点4在作为右操作数之前是用作左操作数的，这样就必须做寄存器拷贝。另一方面，右边的dag虽有同样的调度，但却是最优的，因为结点4先用作右操作数然后才用作左操作数。除非我们用的调度程序检查表达式中的子操作数以及它们共享的方式，否则不会总生成最佳的（代码）执行次序。

可从许多方面对本节启发式的dag计算调度程序进行改进。例如，操作数子表达式往往是棵树而不是dag，可以用15.6节的技术来处理。对于可交换的操作符，其操作数可以交换。如果一个操作数被多

个操作符用作左操作数（将不得不进行寄存器拷贝），那么交换操作数可使左操作数变成右操作数，而代码质量可因此得到改进。其他启发式的方法可参见Aho、Johnson和Ullman（1977）。

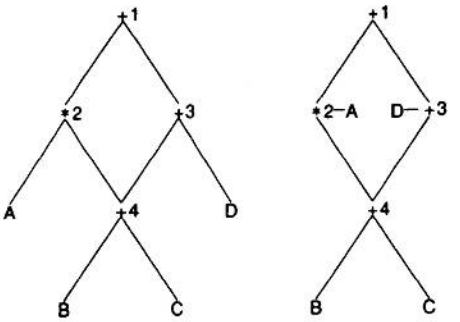


图15-29 两个调度后的“钻石”形dag

15.7.1 别名

当dag中包含别名的影响时，代码生成将变得更加复杂。例如，对一个数组元素（如A(I)）的赋值
586 可能会影响到看似不同的数组元素A(J)。生成dag时必须要考虑到别名。

为处理数组和指针，必须允许表达式产生地址。特别是，A(I)被表示成如图15-30所示，其中Ind是下标操作符。

Ind操作符产生地址。为访问由地址表达式引用的值，我们用↑作为脱引用操作符，就像在Pascal程序中的那样。例如，J := A(I)将被表示为如图15-31所示的dag。由指针和引用型参数所引用的值同样通过↑操作符来访问。

我们将用形式为%1、%2、…等的*内部名字*（internal name）惟一地标记所有地址表达式。这样做可允许地址表达式出现在赋值语句的左边。在赋值后，脱引用的地址表达式可用于标记一个结点。例如，A(I) := J将被表示为如图15-32所示的dag。

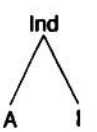


图15-30 A(I)对应的dag

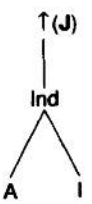


图15-31 J := A(I)对应的dag

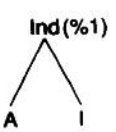


图15-32 A(I) := J对应的dag

如果没有别名问题，数组和指针处理起来将相当容易，地址表达式可以像其他公共子表达式一样被
587 共享。然而，别名还是必须要解释说明的，并且必须满足以下三个要求：

(1) 通过地址表达式的存储不能被推迟。即，A(I) := J在被计算后必须被强制存储到A(I)。

(2) 对数组元素或堆对象或引用型参数的赋值必须注销所有可能成为别名的已标记的子表达式。例如，在A(I) := J后，变量J已被标记为代表A(I)的当前值（它被表示为%1↑）。如果随后处理A(L) := K，那么必须去除J上的标记，因为如果I = L的话，A(I)的值可能已被改变。

(3) 对别名数据对象的读写次序必须加以保护。假设我们翻译A(I) := J; K := A(L); 其dag如图15-33所示。这些dag表明两条语句之间没有依赖关系，就好像A(I)和A(L)是简单变量一样。为确保正确的执行次序，我们引入*依赖弧*（dependency arc），使一个dag看起来依赖于另一个dag，

从而强制正确的计算次序。利用依赖弧，我们可以重写dag，其形式如图15-34所示。现在，对A(I)的赋值被强制先于A(L)的使用。



图15-33 A(I) := J; K := A(L); 对应的dag

借助这些改进，可对前面章节中的调度、寄存器分配等技术加以扩展以处理对数组元素、堆对象和引用型参数的引用。 588



图15-34 有依赖弧的dag

15.8 代码生成器的生成器

近年来，使用代码生成器的生成器愈加普遍起来。代码生成器的驱动程序例程遵循特定目标指令在使用时的规则来选择目标代码。如果规则改变了，则意味着要适应新的指令或体系结构。

正如人们所想像的，产生代码生成器的生成器是一件极富挑战性的工作。以下是一些必须要面对的问题：

- 机器的体系结构千差万别。代码生成器的生成器必须能够适应特定机器的“怪僻”，尽管它们可能还不清楚这些怪僻到底是什么。
- 尽管目标代码不需要最优，但对于自动生成代码的生成器而言，很重要的一点就是它产生的代码应该和手工编写的代码生成器所生成的代码在质量上不相上下。因此，就不能再使用那些将机器过度简化的方法（例如，假设机器为简单的栈或单-寄存器机器）。
- 大多数机器能通过多种方式来做同样一件事情。也就是说，一台机器可能有普通的加法指令、直接加法指令和递增指令。因此，应当避免使用那些不分具体情况而满足于生成任意代码序列或被多种可能的指令选择所迷惑而不知所措的代码生成算法。 589
- 代码生成器可能需要处理与机器无关的优化问题。例如，试图消除不必要的相同表达式重复计算的公共子表达式分析，就被认为是机器无关的。然而，对目标机器的依赖往往使事情复杂化。例如，表达式的重新计算有可能比保存它还“便宜”，而这取决于具体机器的细节。
- 代码生成器必须相当快速，同时为兼顾灵活性而在某些速度方面做出的牺牲也应当是可以接受的。

代码生成器的生成器强调描述-驱动（description-driven）技术。这种技术以目标机器上每条指令行为的精确的形式化描述为输入，将其与所需的操作相匹配以产生相应的计算代码。机器间的差别在于它们指令的效果，而将特定指令的效果与所期望的效果相匹配则是代码生成任务的实质。这点同上下文

无关分析类似——不同的文法产生不同的分析表，但底层的分析驱动程序和分析模型是固定的。

描述-驱动方法的主要优势在于：它们把实现者完全从决定为给定结构生成何种代码这样繁杂的事务中解脱出来。而实现者仅需简单地用形式化的方式精确地说明每条机器指令的作用即可。然后代码生成器搜索这些有关机器的描述以寻找能产生所需计算的单个或多个指令；模式匹配可用来取代解释和情况分析。

指令模式可以是树形或线性结构。根据选择的指令模式，匹配工作可由启发式搜索或形式化分析技术来完成。

启发式搜索技术在进行搜索时创建子目标并采用启发法来试探地选择子目标和匹配的次序。分析技术则使用简单的LR类分析方法，有时还附带有语义属性。

通过考察以下示例，可以说明描述-驱动技术是如何工作的。每条目标机指令被定义为它能计算的中间形式子树以及指令执行后存放结果的子树或结点。例如，将寄存器保存到变址内存单元的存储指令Store Const(R2), R1，其定义见图15-35。

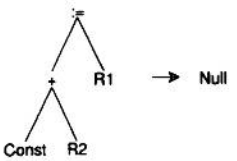


图15-35 Store Const(R2), R1的形式定义

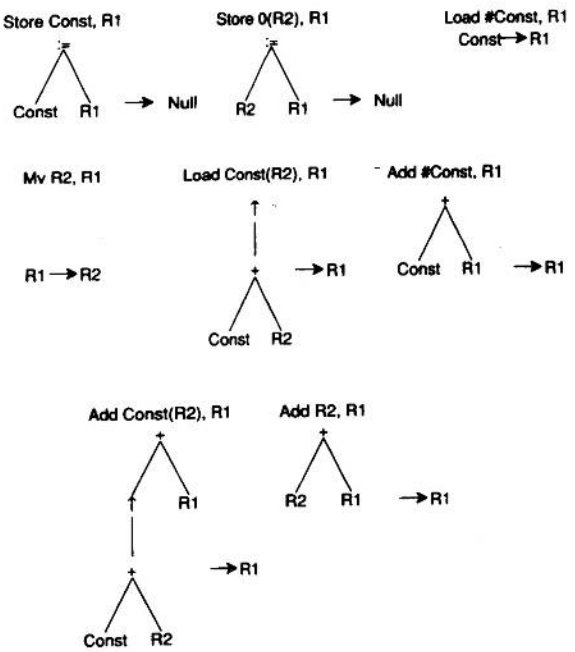


图15-36 机器指令的形式定义

这条规则说明存储指令计算图形子树且可被替换为Null结点。Null结点是用来表示整个子树已被正确地匹配。其他指令的定义如图15-36所示。#Const是值为Const的字面值；↑是间接操作符。

590 ~ 591

作为示例，我们为下面的Pascal语句生成代码：

A := P↑.B + C;　　{ P是指向记录的指针 }

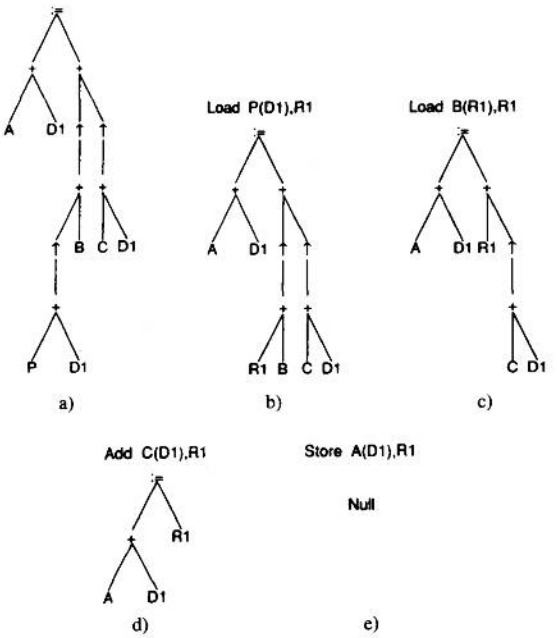


图15-37　语句A := P↑.B + C的代码生成步骤

我们从图15-37a所示的表达式树开始（D1是显示表寄存器）。为生成代码，我们将匹配子树直到产生Null结点为止。我们将在计算中尽量重复使用寄存器（如，与Load Const(R1), R2相比，我们更偏爱Load Const(R1), R1）。然而，我们不允许改写显示表寄存器。我们首先装入由P指示的记录的地址（见图15-37b），然后再装入记录域B的值（见图15-37c）。（假设模式匹配器知道+是可交换的或者已保存表示+的两个操作数次序的模式。）接着，P↑.B和C相加（见图15-37d），随后，所求和被存储到A（见图15-37e）。所生成的代码是：

```
Load    P(D1),R1
Load    B(R1),R1
Add     C(D1),R1
Store   A(D1),R1
```

对表达式树进行不同次序的匹配可能会得到不同的代码序列。例如，我们可以将操作数C装入到寄存器中，然后生成寄存器到寄存器加法，产生的代码是：

```
Load    P(D1),R1
Load    B(R1),R1
Load    C(D1),R2
Add     R2,R1
Store   A(D1),R1
```

如果描述-驱动代码生成器找到将表达式树归约为Null的办法，我们就可以生成正确的但未必是最好的代码。可以使用启发式方法（例如，我们尝试匹配尽可能大的子树以避免生成不必要的指令），但即使这样，仍然会产生问题。例如，假设我们用Add Const2(R), Const1的加法指令（存储器到存储器的加法）取代了指令Add Const(R2), R1；这种情况下，没能将P↑.B装入寄存器可能引起代码生成器阻塞（block）（即，“受骗”了）。此时，我们可能需要回溯并搜索可替代的代码序列。而回溯的介入将使代码生成器更加复杂（不太容易“收回”代码）且减慢代码生成器的速度，当回溯过于频繁时速度将更

592 慢。因此，在实现描述-驱动代码生成器时，一个重要问题就是必须了解选择各种候选代码序列的方法并避免阻塞。

15.8.1 基于文法的代码生成器

在Glanville和Graham（1978）中提到，代码模板和表达式树匹配的问题与语法分析时产生式和记号序列匹配的问题非常类似。Glanville和Graham敏锐地用语法分析的术语重新表述了模板匹配问题。首先，与直接的树匹配不同，他们采用一种线性的代码生成IR（Code-Generation IR，CGIR），也就是一种波兰前缀形式。采用这种方法，前面的示例变为：

593 := + A D1 + ↑ + ↑ + P D1 B ↑ + C D1

这种结构可以通过对表达式树进行前序遍历或者通过扩充元组、将结果临时变量替换为它们的定义而得到。由于我们希望能利用可用的地址模式，因此我们在CGIR中显式地给出了显示表寄存器和栈指针。

现在，我们将模板重新改造为普通的产生式。对于可交换的操作符，我们包括了两种操作数的顺序，如图15-38所示。

```
Null  → := + Const R2 R1      -- Store Const(R2),R1
Null  → := + R2 Const R1      -- Store Const(R2),R1
Null  → := Const R1           -- Store Const,R1
Null  → := R2 R1              -- Store 0(R2),R1
R1    → Const                 -- Load #Const,R1
R1    → ↑ + Const R2          -- Load Const(R2),R1
R2    → R1                    -- Mv R2,R1
R1    → ↑ + R2 Const          -- Load Const(R2),R1
R1    → + ↑ + Const R2 R1     -- Add Const (R2),R1
R1    → + ↑ + R2 Const R1     -- Add Const (R2),R1
R1    → + R1 ↑ + Const R2     -- Add Const (R2),R1
R1    → + R1 ↑ + R2 Const     -- Add Const (R2),R1
R1    → + Const R1            -- Add #Const,R1
R1    → + R1 Const            -- Add #Const,R1
R1    → + R2 R1               -- Add R2,R1
R1    → + R1 R2               -- Add R2,R1
```

图15-38 Graham-Glanville代码模板

和在普通的上下文无关文法中一样，像R1或Const之类的符号是占位符，代表着某些被匹配的特定符号。这里的R1并不代表硬件寄存器1，而是指那些已被分配且绑定到符号R1的寄存器。我们表示实际硬件寄存器的方式是：显示表寄存器用符号D1、D2、…表示，而操作数寄存器用Reg1、Reg2、…表示。如果一个符号在产生式中出现不止一次，那么与此符号关联的特定信息在所有情况下都是一样的。例如，在R1 → + R2 R1中，在匹配+之后，无论哪个真实机器寄存器与R1关联都将被认为保存着那个刚被计算出的和。

我们将使用与普通的基于LR的技术（见第6章）相类似的分析技术。然而，必须对分析器做少许修改以处理指令选择中的二义性。首先，如果有移进-归约冲突（分析器此时可以识别产生式或继续读入），
594 那么总是先执行移进。这种改动对应着在启发式方法中仅在需要时才生成相关代码。其次，如果有归约-归约冲突（两个或更多的产生式被识别），将考虑多个独立规则以确定哪一个指令序列更好。同样，在识别产生式前还必须检查某些限制条件。因此，在R1 → + R2 R1中，我们需要核实R1可以被修改，因为它将保存求和的结果。这些限制使我们可以保证显示表寄存器在计算过程中不会遭到破坏。

我们通过为前面的示例生成代码来举例说明该方法的工作过程。注意，分析器将自动匹配所有可行的产生式，因此，如果某个指令序列有可能被匹配，那么它就会被考虑。分析器首先读入 := + A D1。它必须和下面有关存储的产生式之一相匹配：

```
Null → := + Const R2 R1     -- Store Const(R2),R1
Null → := + R2 Const R1     -- Store Const(R2),R1
Null → := Const R1          -- Store Const,R1
Null → := R2 R1             -- Store 0(R2),R1
```

如果表达式剩余部分可由R1匹配，那么可以使用第一条产生式。其他可能的情况还有，如果我们让R2匹配常量A，则第二条产生式将被选用。但根据尽可能移进而非归约的原则，我们否决了这种可能性。（该规则使我们可以避免不必要地把A装入寄存器的操作。）类似地，第4条产生式将在R2匹配+ A D1时被使用。但这个选择再次被否决，因为我们宁可移进也不归约（并可再次避免不必要的装入）。

现在我们知道：表达式的剩余部分：

```
+ ↑ + ↑ + P D1 B ↑ + C D1
```

必须和以R1为左部的某个产生式相匹配。可能的产生式有：

```
R1 → + Const R1             -- Add #Const,R1
R1 → + R1 Const             -- Add #Const,R1
R1 → + ↑ + Const R2 R1      -- Add Const (R2),R1
R1 → + ↑ + R2 Const R1      -- Add Const (R2),R1
R1 → + R1 ↑ + Const R2      -- Add Const (R2),R1
R1 → + R1 ↑ + R2 Const      -- Add Const (R2),R1
R1 → + R2 R1                -- Add R2,R1
R1 → + R1 R2                -- Add R2,R1
```

我们使用其中的第4条产生式来匹配最长可能的前缀。这就意味着该产生式的其他部分R2 Const R1必须与

```
↑ + P D1 B ↑ + C D1
```

相匹配。此时首先使用R1 →↑ + Const R2并生成Load P(D1), Reg1。然后B匹配Const，并再次使用 595
刚才那个产生式匹配↑ + C D1。因而生成代码Load C(D1), Reg2。至此该加法产生式已全部匹配，我们生成代码Add B(Reg1), Reg2。而此时存储产生式也全部匹配，因此我们生成代码Store A(D1), Reg2。整个代码序列如下：

```
Load    P(D1),Reg1
Load    C(D1),Reg2
Add     B(Reg1),Reg2
Store   A(D1),Reg2
```

上述代码序列不是最优的，因为它用了两个寄存器，而事实上一个寄存器即可满足要求：

```
Load    P(D1),Reg1
Load    B(Reg1),Reg1
Add     C(D1),Reg1
Store   A(D1),Reg1
```

这个问题出在我们总是试图尽可能避免代码的生成。因此，我们看到：指令Load B(Reg1), Reg1可以通过将B(Reg1)作为Add指令的操作数而得以避免。在这个例子中，该策略没有成功，因为我们终结了必须生成的C(D1)的装入代码。尽管如此，我们还是注意到：如果第二操作数已在寄存器中（例如，因为它是表达式而非变量），那么避免装入B(Reg1)可能就是应该做的事情。

一般地，直接代码生成器在它们看清楚第二操作数之前就必须决定如何处理指令的第一操作数！因此，它们试图推迟那些必需的操作，尤其是装入，直到有绝对必要时。这有时会导致次优的代码。在生成代码前检查两个操作数的代码改进技术可参见文献Christopher、Hatcher和Kukuk（1984）以及Aho、Ganapathi和Tjiang（1989）。

Graham-Glanville方法好的一面是它能够在构造代码生成器时发现潜在的阻塞状态。特别是，如果基于代码生成产生式的分析器达到这么一个状态，该状态中一个产生式已部分匹配但剩余部分却不能被有效的操作数或表达式匹配，那么代码生成器就可能阻塞。例如，假定有些奇怪的原因促使我们只有寄

存器到存储器加法却没有寄存器到寄存器加法。我们或许能发现这样的分析状态，其中，+和某个寄存器已被匹配，剩下一个待匹配的存储位置。然而，如果第二操作数已在寄存器中，那么我们就会“卡”在这里。此类状态代表着一种错误的情形，那时我们不能为所有有效的输入生成正确的代码。

为解决这个问题，我们可以添加新的产生式以匹配剩余的操作数。对于我们的例子，可以添加如下
596 产生式：

```
Value → ↑Const
Value → R1        -- Store Value(0),R1
```

它们通知代码生成器，一个值可以通过提取存储单元的内容而获得。若需要的话，在寄存器中的值可以被存储到内存单元中，接着再从内存单元中提取出来充当操作数。

Graham-Glanville代码生成技术已在许多实验性编译器中测试过，并开始出现在产品编译器中。为提高代码生成速度，通常通过引入特殊符号和产生式来减少语义检查的次数。例如，像Zero、One、Two、Four和Eight等特殊符号可以表示指令中出现的“魔数”。类似地，对各种属性值，我们将用各种不同的符号而不是配备了属性（byte、long、real、longreal）的单个符号来表示它们。

使用语法来编码语义检查，其缺点是符号和产生式的数目将变得很大。（对于VAX机器，据称它的机器描述文法包含1073条产生式、219个终结符、148个非终结符和2216分析状态。）实际上，当LALR(1)生成器作用于机器描述文法时，它往往要耗费数小时来产生代码生成器表。然而，特别设计的表生成器可以显著地提高产出率（Henry et al., 1984）。

代码生成的速度主要取决于底层的分析过程，其时间多花在产生式链的分析以及分析表条目的存取和操作上。据报导，这种代码生成的速度大约比产品编译器的速度慢50%。通过精心设计，该方法所生成的代码可以和那些普通的非优化编译器所生成的代码在质量上一较高低。

15.8.2 在代码生成器中使用语义属性

Graham-Glanville方法的局限性在于它是纯语法的。也就是说，它主要通过上下文无关的方式匹配符号序列。而代码生成过程的其他方面则被非形式化地处理。例如，符号可以关联语义值，但在文法中无法显现这些值确切是什么以及在哪里计算。类似地，即使代码生成产生式匹配一个CGIR符号序列，它们也可能是不可用的。因为只有非上下文无关的约束可以使用（地址也许需要对齐到字边界，立即操作数的值也许需要在一定的范围内，等等）。这些非上下文无关的约束不能直接在产生式中表示，而必须在别的什么地方加以体现，就如同语义规则和普通的上下文无关产生式分开那样。

在Ganapathi和Fischer（1985）中，通过将*属性产生式*（attributed production）用作代码模板来解决这些问题。首先，显式的属性值被包括进文法符号中。这些属性代表着与符号相关并与之存放在一起的
597 信息。两种新的符号，即*动作符号*（action symbol）和*谓词符号*（predicate symbol），连同终结符和非终结符一起被添加到代码模板中。动作符号封装了新属性值的计算和带有副作用（尤其是代码生成）的操作。它们以#为前缀，如同调用语义例程的动作符号一样。

谓词符号类似于动作符号。然而，它们不产生属性值。相反地，它们在被调用时，产生真值或假值。如果谓词为真，那么包含此谓词的产生式的匹配工作将继续进行。如果谓词为假，那么该产生式将不再被考虑。为清楚起见，所有谓词均带有后缀 ?。谓词是一种方便的、能够精确定义产生式应用场合的机制。从代码生成的角度看，这是一项重要的改进，因为现在的机器描述是一个属性产生式，它能精确地定义（带属性的）中间形式符号与机器指令之间的对应关系。

作为示例，考虑下面的产生式：

```
Long(R) → + Long(A) Long(R) IsOne?(A) Dead?(R) #emit(IncL,R)
```

该产生式匹配两个长整型操作数相加并产生长整型结果。在应用该产生式前，必须验证它的左操作数是

否为1以及右操作数是否“死亡”（若是，则可以“撤销”它）。当且仅当这些条件满足时，将生成一个长整型增量指令IncL。

产生式中的谓词决定该产生式能否被使用。如果有多条产生式被启用（即谓词为真），那么此时必须有某种消除二义性的办法。出于代码生成的目的，可将产生式按两种顺序排列成表。其中一种是以指令大小的递增序排列，而另一种则按速度的递减序排列。根据优化的对象是代码大小还是速度，我们将无二义地从代码大小或速度列表中选择最早“启用”的产生式。

出于代码生成的目的，可将属性产生式分成三类。第一类表示地址模式产生式。它们将中间形式与目标机地址模式相匹配。它们可以产生也可以不产生代码。例如：

```
Adr(A) → Index Obj(Offset,Size) Base(Reg) #build_adr(Offset,Size,Reg,A)
```

可以和活动记录中常用的访问数据的变址操作相匹配。如果变址操作包含常量偏移和基址寄存器，那么我们只是简单地在属性A里构造一个地址描述符。然而，也有变址操作的第二个操作数不是基址寄存器的情况。下面的产生式预见了这种情况：

```
Adr(A) → Index Obj(Offset,Size) Adr(B) #get_reg(Long,R)
          #emit(MovL,B,R) #build_adr(Offset,Size,R,A)
```

此例中，我们通过调用get_reg()来获得一个寄存器，然后我们生成将第二个操作数的内容传送至该寄存器的指令，接着再构建一个地址，它包含偏移和新装入的基址寄存器。我们注意到：产生式的次序很重要，因为我们尽力避免生成不必要的代码。此外，可以创建另一对变址产生式用来覆盖那些Index操作符中操作数顺序可变的情况，其中基址寄存器或地址可以出现在偏移值的前面。 598

第二类属性产生式包含操作数传送（operand transfer）操作。它们用来处理转换，以便在使用破坏性操作时保护操作数（如在两地址的加法中）以及处理非正交性操作（即，不是所有的操作数或地址格式都可用于某个给定的操作）。这些产生式对防止阻塞至关重要。它们可以产生也可以不产生代码。例如，考虑：

```
Long(B) → Adr(B) IsLong?(B)
```

该产生式匹配代表地址的一个非终结符并查验它代表的对象是否为长整型格式。如果需要转换，可使用下面的产生式（CvtWL将字类型操作数转换为长整型操作数）：

```
Long(A) → Adr(B) ConvertLong?(B) #get_temp(Long,A) #emit(CvtWL,B,A)
```

第三类属性产生式是用来选择那些实现各种操作的指令序列。正常情况下，可能出现多条产生式，因为同一种操作可以用不同的目标指令来实现。例如，考虑如图15-39所示的产生式。

```
Long(R)  → + Long(A) Long(R) IsZero?(A)
Long(R)  → + Long(A) Long(R) IsOne?(A) Dead?(R) #emit(IncL,R)
Null     → := Long(A) + Long(A) Long(B) #emit(AddL2,B,A)
Long(R)  → + Long(A) Long(R) Dead?(R) #emit(AddL2,A,R)
Null     → := Long(C) + Long(A) Long(B) #emit(AddL3,A,B,C)
Long(R)  → + Long(A) Long(B) #get_temp(Long,R) #emit(AddL3,A,B,R)
```

图15-39 代码生成的属性模板

上述产生式按最特殊的到最一般的情况排序，其中谓词控制着各种选择的可用性。首先，测试与0相加的特殊情况。如果是这种情况，那么不会有代码生成。然后考虑把1与一个非活跃值相加的情况。接下来，列出的是加法的操作数之一是加法目的地的情况。如果加法操作数之一非活跃，那么此操作数

599 可被重写，规则中包含了这种可能性。最后，考虑的是最一般且最昂贵的加法形式——三操作数加法，它或者在赋值语句的上下文中，或者作为产生长整型结果的子表达式。

为处理可交换操作符（如+操作符），我们使用表示（两种）操作数顺序的两条产生式。我们注意到：如果一个操作的最后一条（即最一般的）产生式不包含谓词，那么就不会发生阻塞。否则，我们必须规定在所有情况下，对于某个给定的操作数，至少有一个产生式可被启用。

作为属性化代码生成的示例，重新考虑前面章节中的例子：A := P↑.B + C。当此例子被翻译成一个属性化CGIR时，我们有：

```
:= Index Obj(A,Long) Base(D1) +  Index Index Obj(P,Long) Base(D1)
    Obj(B,Long) Index Obj(C,Word) Base(D1)
```

其中，括号里的属性值表示操作数的字节长度和偏移。假定A、P和B是长整数，C是字型整数。Index操作符可用来访问被分配在活动记录中的变量或访问记录域。因为地址和值均为显式的，所以不需要插入显式的↑（我们可以那么做，如果我们希望将地址强制转换为值的话）。

首先识别那个包含A的变址操作，并产生：

```
:= Adr(Long,(A,D1)) + Index Index Obj(P,Long) Base(D1) Obj(B,Long)
    Index Obj(C,Word) Base(D1)
```

Adr的属性表明：它的操作数的长度为长整型且它的地址是(A, D1)。接下来，操作数A被识别为具有长整型格式：

```
:= Long(A,D1) + Index Index Obj(P,Long) Base(D1) Obj(B,Long)
    Index Obj(C,Word) Base(D1)
```

再接下来，P被匹配为一个Adr：

```
:= Long(A,D1) + Index Adr(Long,(P,D1)) Obj(B,Long)
    Index Obj(C,Word) Base(D1)
```

下一步，包含P和B的变址操作被匹配。因为这个操作包含的是Adr而非Base，所以我们必须分配寄存器(如，R1)，并通过生成MovL(P, D1), R1指令将其装入。现在我们有：

```
:= Long(A,D1) + Adr(Long,(B,R1)) Index Obj(C,Word) Base(D1)
```

这个操作数（Adr(Long, (B,R1))）被识别为Long操作数，且C被识别为Adr，其操作数长度为Word：

```
:= Long(A,D1) + Long(B,R1) Adr(Word,(C,D1))
```

600 此刻，因为没有混合长度的加法操作，C将被转换为长整型长度的操作数。我们分配临时变量T，并通过生成指令CvtWL (C, D1), T将C转换为长整型格式。现在我们有：

```
:= Long(A,D1) + Long(B,R1) Long(T)
```

它将被完全匹配，产生Null，且生成指令AddL3 (B, R1), T, (A,D1)。最终生成的代码序列为：

```
MovL    (P,D1),R1
CvtWL   (C,D1),T
AddL3   (B,R1),T,(A,D1)
```

在像VAX一类的体系结构上，这是语句A := P↑.B + C的一个非常理想的翻译。

属性化代码生成的好处之一在于：它很容易通过添加新的产生式、谓词或动作符号来逐步提高所生成代码的质量。因此，如果希望利用移位指令来实现某些乘法，那么所要做的仅仅是添加新的产生式，所附带的谓词将定义应用新产生式的时机。该方法其他的好处是：它使得更精细的优化成为可能。例如，有时候我们很希望推迟或禁止赋值操作。也就是说，给定A := B，我们可以推迟赋值操作。只要B值不改变，那么对A值的引用就可以被替换为对B值的引用（这称为复写传播）。

为实现此类优化，我们可以引入新的称为delay的动作符号。delay类似于emit，但它将推迟实际生成指令直到它迫不得已而为之。事实上，它起到一个过滤器作用，将指令排队于缓冲区中并在需要的时候实际生成。有趣的是，为实现该方法，我们仅需要修改emit例程来缓冲指令即可。而基本的代码生成方法根本不需改动。

基于属性的实验性代码生成器已在多种机器（如PDP-11、VAX和iAPX-86）上生成。在VAX机器上，创建一个代码生成器通常需要几分钟时间。具有丰富指令集的VAX机器和具有非正交指令集的iAPX-86机器，它们中每一个均需要大约600条产生式和1200个分析器状态。而在某种程度上较简单的PDP-11机器也需要大约400条产生式和800个分析器状态。它们的代码生成速度为每分钟数千条指令。据估计，将这种代码生成器移植到新的体系结构上约耗时一个月左右。

同本地编译器比较，这种代码生成器所生成的代码的质量非常好，与那些普通的非优化编译器所生成的代码也可以一较高低。简言之，属性化代码生成器特别擅长利用特殊指令、硬件寻址模式和寄存器。 601

15.8.3 生成窥孔优化器

文献（Fraser and Davidson，1980）中讨论了各种自动创建窥孔优化器的方法。其想法首先是在寄存器-传送层定义目标机指令的执行效果。在这一层里，指令可以修改基本硬件单元，包括内存单元（用向量M表示）、寄存器（用向量R表示）、PC（程序计数器）以及各种条件码等。例如，在寄存器-传送层，我们可以有以下指令序列（PC作为指令执行的一部分而被隐式地增加）：

```
R[3]   ← R[3] + 1                    —— 寄存器3加1
M[c]   ← 0                           —— 内存单元c置0
PC     ← (NZ = 0 ⇒ 140 else PC)      —— 如果条件码（NZ）为0
                                     —— 则跳转到140
```

一条目标机指令或许有多种执行效果，因此它在寄存器-传送层的定义需包括更多的赋值任务（或含义）：

```
Add s,d   d ← d + s;
          NZ ← d + s ? 0    —— ?是比较操作符
```

在此例中，加法指令对其操作数进行相加，把结果放入第二个操作数中，并根据结果的符号设置条件码。这些寄存器-传送的效果可以被认为是同时发生的，因为它们都是同一条指令组成的一部分。

操作数可以利用各种寻址模式，且它们也可在寄存器-传送层定义并被包括在指令中以表示指令全部执行效果。例如，指令Add 100(R2), @R3定义如下，其中@表示间接操作：

```
@R3   ← @R3 + 100(R2);
NZ    ← @R3 + 100(R2) ? 0
```

上述序列扩充为：

```
M[R[3]]   ← M[R[3]] + M[R[2]+100];
NZ        ← M[R[3] + M[R[2]+100] ? 0
```

窥孔优化器（PO）的工作方式是：考虑指令对，将它们扩充为寄存器-传送层定义，进而简化指令组合的定义，并随后搜索与组合指令对具有相同执行效果的单条指令。考虑： 602

```
SUB #2,R3   ——从R3减去2
CLR @R3     ——清零由R3所指的单元
```

首先，上述定义被替换为：

```
R[3]      ← R[3] - 2;    NZ   ← R[3] - 2 ? 0
M[R[3]]   ← 0;           NZ   ← 0 ? 0
```

我们注意到：NZ的第一次赋值可被忽略，因为在第一次的值被引用之前第二次赋值就已经重置了

NZ的值。进而，我们将第二条指令中R[3]的引用替换为第一条指令中赋予R[3]的表达式。于是有：

R[3] ← R[3] - 2; M[R[3]-2] ← 0 ; NZ ← 0 ? 0

该模式匹配使用自动递减的清零指令；因此，PO将它们替换为CLR-(R3)。

为切实可行，单条指令必须完成组合指令中所有的寄存器传送。此外，它也可以做其他一些不起作用的寄存器传送（即，这些寄存器传送不会对后续计算产生影响）。因此，在单条指令中也可以设置条件码，即使这并不需要，但只要该条件码不被后面的指令引用即可。

开始于条件分支的指令对将受到特别对待。特别地，第二条指令被冠以与初始条件相反的条件（即，仅当前面的条件分支不成立，才可以执行第二条指令）。例如，假定我们有：

```
      BZ   L1
      B    L2
L1:
```

它被扩充为：

```
      PC ← (NZ = 0 ⇒ L1 else PC)
      PC ← L2
L1:
```

603

我们包含求反条件后得到：

```
      PC ← (NZ = 0 ⇒ L1 else PC)
      PC ← (NZ ≠ 0 ⇒ L2 else PC)
L1:
```

PO做代数简化后得到：

```
      PC ← (NZ = 0 ⇒ L1 else L2)
L1:
```

然后重写得到：

```
      PC ← (NZ ≠ 0 ⇒ L2 else L1)
L1:
```

最后，我们得到：

```
      PC ← (NZ ≠ 0 ⇒ L2 else PC)
L1:
```

它被匹配为：

```
      BNZ L2
L1:
```

因此，PO就发现了一个常见的优化——后面伴随一个无条件分支的条件分支可被替换为一个原先条件求反的条件分支。

无条件分支和它的目标指令配成对。这种配对往往允许跳转链（即，跳转到另一个跳转）的“倒塌”合并。然而，如果这种指令对中的第二条指令前带有标号，则不能做上述优化。在这种情况下，应该保持原来到这些标号的跳转语句能正确工作。尽管如此，如果PO能清除对一个标号的所有引用，那么此标号本身也可被清除，这样就有可能发现新的优化机会。

上面描述的指令分析与简化实际上并不是在编译期间完成的，因为那样做的话，编译速度将会非常慢。事实上，我们已经提前对一些有代表性的实际程序的样本进行了分析，并把最常见的窥孔优化保存在一张表里。在编译时，可参考这张表来决定落在“孔”中的指令能否被优化。

Davidson和Fraser注意到：窥孔优化可用来极大地简化代码生成。这个想法是：代码生成器仅需利用最一般的指令格式和机器最基本的寻址模式，而依靠窥孔优化发现那些可被替换的特殊用途的指令。在对这种方法进行相当极端的测试中，Davidson和Fraser设想了一个生成简单的P-代码风格指令的编译

器。这些指令以宏展开的方式被扩展为PDP-11的代码，而窥孔优化紧接着被执行。在多数情况下，产生的代码在代码大小上可与本地的PDP-11编译器所生成的代码进行比较。这个结果告诉我们，这种首先生成较粗糙的代码、然后再使用窥孔优化对其精练改进的方法也许是可行的。 604

15.8.4 基于树重写的代码生成器的生成器

Cattell提出了基于树重写的代码生成器的生成器（Cattell 1980）。使用该方法时，首先利用寄存器-传送符号描述每条指令的执行效果。然后，代码生成器通过匹配指令和IR树“发现”合适的代码序列。也就是说，代码生成器寻找适当的方式将表达式树分解为特殊的*原始*（primitive）树的组合。

如前所述，有多种方法可以对树进行分解（即，有多种可以计算它的代码序列）。然而，在遇到不能约减至基本指令的树时，分解过程可能受阻。此时，需要回溯以寻找别的分解方式。尽管如此，这种代码生成速度可能慢得令人难以接受。Cattell解决这个问题的方法是将代码生成器的创建分成两部分。

第一阶段称为*选择*（select），它创建可以被扩展为目标代码的树模式的有限集合。实际上，此集合也就是要实现的树模式的一个目录。一旦选定树模式，那么将开始称为*搜索*（search）的第二阶段。“*搜索*”为每一个选定的模式寻找可能的实现。在代码生成器创建期间，它只运行一次。因此它仔细地检查可能的代码序列并选择其中最适合的一个。该序列随后被保存在一张表里并在编译时用来替换相应的树。“*选择*”阶段被设计用来保证所有可能出现的树都能被分解为子树，而与子树等价的目标代码（已被“*搜索*”阶段）保存在表中。

Cattell代码生成器的性能相当好。树能够以每秒数千条指令的速度被映射到目标代码，尽管其他的代码生成组件可能会降低代码生成的总体速度。代码生成器的生成器的速度相当慢，大约每秒10个子树。

Cattell方法中最有趣的是它的*搜索*阶段。它有潜力发掘那些对代码生成器的实现者来说是未知的代码序列（尽管代码序列是否由于难以琢磨而需要去发掘仍是一个有待争论的问题）。Cattell方法的主要缺点是：在某种程度上，它生成的代码序列是由“*选择*”阶段所选出的固定子树模板的宏展开而得到的。这意味着那些在“*选择*”阶段没被选中的特殊子树将不被“*搜索*”阶段分析，而因此可能被扩展为次优的代码。但如果要经历最终的窥孔优化阶段，这倒也不是一个问题!

练习

1. 假设要为15.4.3节中BB1体系结构生成代码，有三个寄存器可用作操作数寄存器。针对以下代码片段，你会生成何种代码？假设所有变量是静态分配的整数或整数数组。 605

```
C       := 1;
A       := B–C*D;
D       := C + B;
X(B)    := C;
X(A)    := X(A) + X(D+1);
```

BB1体系结构的后继是BB2。BB2包含BB1的所有指令格式。它同时还包括一个三寄存器指令，其格式为OP Reg1, Reg2, Reg3，其中Reg3 := Reg2 OP Reg1。针对上述相同代码片段，你会产生什么样的BB2代码？

2. 在某些机器体系结构中，寄存器分配很复杂，其原因是指令中寄存器R的使用将隐式地涉及另一个寄存器R'。例如，涉及R_i的乘法可能会将两倍（字）长度的乘积存放在R_i和R_{i+1}中。作为替代，三-地址指令，OP Reg_i, Reg_j, Reg_k，通过要求k=j+1，可被“紧缩”为两-地址格式。

针对包含隐式寄存器引用的机器，你将如何组织get_reg()例程？为使问题具体化，假设你的get_reg()使用BB3体系结构。该体系结构和15.4.3节中BB1体系结构基本相同，但在格式为OP X, Reg_i的指令中，BB3要求X为寄存器或存储地址，计算结果将存放在Reg_{i+1}而不是Reg_i中。

用你的例程为练习1中代码片段生成BB3代码。

3. 在图15-1中给出了元组(+,A, B, C)的代码生成器。推广此代码生成器以包括A或B是零的情况。进一步扩展你的代码生成器以包括A、B或C不完全相同的情况。

4. 在15.4.1节里，我们注意到所涉及的计算有时是作为地址计算的一部分。例如，IBM 360/370机器包括相加一个寄存器对、再加上常量的偏移以构成地址的寻址模式（两个寄存器中一个作为基址寄存器，另一个作为变址寄存器）。

   给出为A(I)产生的元组，其中A和I是通过显示表寄存器访问的局部变量。大致描述代码生成器将如何利用基址加变址的寻址模式?

   有时寻址模式可用来有效地计算普通的表达式。例如，考虑A*B+C*D+1。首先计算A*B和C*D并将它们的值放入寄存器R1和R2。下一步显然是将R2加至R1，然后再将1加到R1。而一个很不明显但是更好的代码序列是生成LA R1, 1(R1,R2)。这条指令使用了基址加变址的寻址模式在一步内完成
606 R1+R2+1，并将结果存到R1（LA是取地址指令）。如何扩展+的代码生成器以生成这种有效但不那么明显的代码序列?

5. 考虑以下代码片段:

```
A(I,J)          := A(I,J)/B(I,J);
B(I+1,J+1)      := B(I,J+1)*A(I,J);
```

   下面是为上述代码片段可能生成的元组:

```
(Index,A,I,T1)
(Index,T1,J,T2)
(Index,A,I,T1)
(Index,T1,J,T2)
(Index,B,I,T3)
(Index,T3,J,T4)
(/,T2↑,T4↑,T5)
(:=,T5,T2↑)

(+,I,1,T6)
(Index,B,T6,T7)
(+,J,1,T8)
(Index,T7,T8,T9)
(Index,B,I,T3)
(+,J,1,T8)
(Index,T3,T8,T10)
(Index,A,I,T1)
(Index,T1,J,T2)
(*,T10↑,T2↑,T11)
(:=,T11,T9↑)
```

   采用15.4.2节里的值-编号技术来识别并删除该元组序列中冗余的元组。

6. 我们经常需要进行运行时检查以验证数组、指针和约束变量的使用是否正确。如果“傻瓜式”生成检查代码，那么将显著地损害程序的大小和速度。例如，假设我们有格式为(TestRng, I, L, U)的元组，用来测试I是否在范围L..U内。该元组可用于检查下标和约束变量。如果L≤I≤U，(TestRng, I, L, U)不起作用；否则将引发constraint异常（可能导致程序运行终止）。“傻瓜式”代码生成器会在每个下
607 标操作前生成TestRng元组。在练习5的示例中，设A和B是10*10的数组，那么产生的代码如下:

```
(TestRng,I,1,10)
(Index,A,I,T1)
(TestRng,J,1,10)
(Index,T1,J,T2)
(TestRng,I,1,10)
(Index,A,I,T1)
(TestRng,J,1,10)
(Index,T1,J,T2)
(TestRng,I,1,10)
```

```
(Index,B,I,T3)
(TestRng,J,1,10)
(Index,T3,J,T4)
(/,T2↑,T4↑,T5)
(:=,T5,T2↑)

(+,I,1,T6)
(TestRng,T6,1,10)
(Index,B,T6,T7)
(+,J,1,T8)
(TestRng,T8,1,10)
(Index,T7,T8,T9)
(TestRng,I,1,10)
(Index,B,I,T3)
(+,J,1,T8)
(TestRng,T8,1,10)
(Index,T3,T8,T10)
(TestRng,I,1,10)
(Index,A,I,T1)
(TestRng,J,1,10)
(Index,T1,J,T2)
(*,T10↑,T2↑,T11)
(:=,T11,T9↑)
```

扩展15.4.2节里的值-编号技术来识别并删除冗余的TestRng元组。以上述元组序列为例说明你的扩展。

7. 考虑如下程序片段:

```
A   := D/(B*C);
D   := D-(B-C);
A   := A+C;
C   := C+D;
```

所生成的元组如下:

608

```
(*,B,C,T1)
(/,D,T1,T2)
(:=,T2,A)

(-,B,C,T3)
(-,D,T3,T4)
(:=,T4,D)

(+,A,C,T5)
(:=,T5,A)

(+,C,D,T6)
(:=,T6,C)
```

假设采用15.4.3节里BB1体系结构且有三个可用寄存器。使用15.4.3节里寄存器追踪技术，为上述元组生成BB1代码。重做代码生成，此次假定有4个可用寄存器。

8. 重做练习7，此次在第二和第三条语句之间加入子程序P的调用。假设我们不知道有关P使用寄存器的情况，因此对任何内容需要保护的寄存器必须在调用前后加以保存和恢复。尽管如此，如果在调用前清除寄存器关联，那么就有可能避免不必要的保存。
9. 扩展15.4.3节里寄存器追踪技术以便包括格式为Move Reg1, Reg2的寄存器传送指令。该指令拷贝Reg1的内容到Reg2，其生成开销为1个单位。使用你扩展的算法重做练习7。
10. 有时寄存器的内容被存储到内存单元中，而紧接着该值又立刻从内存单元中被再次装入寄存器中。也就是说，我们可以看到以下的代码序列:

```
Store   L,R1
Load    L,R2
```

其中R1和R2不必相同。试解释使用“窥孔优化”将如何优化这个指令对?

11. 在练习4中，我们看到：如果有基址加变址的寻址模式，那么将两个寄存器和一个常量相加便可形成一个单条指令。说明如何使用窥孔优化将显式的寄存器与常量的加法替换为利用基址加变址的寻址模式的一个单条指令。

609 12. 窥孔优化的定义经常在简单的规则中使用模式变量来描述众多的相关优化。因此，

```
Mult #2,%R  ⇒  Add %R,%R
```

将表示任意寄存器与2的相乘可以被替换为那个寄存器到自身的加法。（%R匹配任何寄存器操作数。）大致描述如何实现能匹配包含模式变量规则的窥孔优化器。

13. 假定我们正翻译以下表达式：

```
(A + (B*C*D)) / (E-F*G).
```

为该表达式创建表达式树（假定使用Ada/CS的运算符优先级），然后使用图15-11中的例程`register_needs()`标记它。接下来使用图15-13中的例程`tree_node()`为该表达式生成代码，假定有两个寄存器可用。如果使用图15-14中的例程`commute()`，那么能否改进所生成的代码？

14. 证明`register_needs()`、`tree_node()`和`commute()`的执行时间正比于表达式树中操作符（即运算符）的个数。

15. 有时，如果利用某些操作（如+和*）的结合性，可以改进为表达式生成的代码。例如，如果使用`tree_node()`翻译下面的表达式，则需要三个寄存器：

```
(A+B) * (C+D) * ((E+F) / (G-H))
```

即使采用`commute()`，还是需要三个寄存器。然而，如果利用乘法的结合性从右到左计算乘法，那么仅需要两个寄存器。（首先计算((E+F) / (G-H))，然后再计算(C+D)* ((E+F) / (G-H))，最后计算整个表达式(A+B) * (C+D) * ((E+F) / (G-H))。）

编写例程`associate()`，它可以重排可结合的操作数的操作数以改进代码质量。（提示：允许可结合的操作符拥有多于两个的操作数。）

16. 考虑下面的语句序列：

```
A := B+C*D;
B := A*(C*D);
C := (C*D)*2;
D := A+C;
```

为这些语句创建表达式树，然后使用`tree_to_dag()`（见图15-19）将表达式变换为dag。接下来

610 使用`schedule()`（见图15-23）为代码的生成调度dag，并使用`allocate_v_regs()`（见图15-25）来分配虚拟寄存器。最后，将虚拟寄存器映射到真实寄存器并产生计算dag的代码。

17. 重做练习16，此次假定B是与A别名的引用形参。

18. 例程`schedule()`（见图15-23）尝试以启发式方法调度左操作数以便在右操作数后使用它们，并随后改写它们。如图15-29所示，这种启发式方法也可能失效。大致描述应如何扩展`schedule()`以正确处理共享的子操作数。你的扩展能正确处理图15-29中的两个dag吗？

19. 解释应如何扩展15.7节中的代码生成算法以处理dag中出现的可交换操作符？你的扩展应当可以互换可交换操作符的左、右操作数，前提条件是那样能产生更好代码。

20. 编写程序实现15.8节中的树匹配代码生成器。确保你的程序不会阻塞。即，如果表达式树的一部分被匹配，而其余部分无法匹配，你的程序必须能撤销先前的匹配并尝试其他的选择直到整个表达式树被匹配。你可以假定只有正确的表达式树才被处理；因而，一定存在某些正确的匹配。用你程序测试图15-37中的示例。

21. 练习20中设计的代码生成器确保了对所有正确的表达式树都可以生成某个正确的代码序列。然而，

它却不能保证所生成的代码是最优的或质量不错的。

假定每条目标机指令已被标记有代价（其大小或速度）。推广练习20中的代码生成器以便将表达式树和那些能带来最小代价（代码最小或速度最快）指令序列的指令模式相匹配。

22. 语句A:= B+C+2可被表示为如下的前缀代码生成IR:

```
:= + A D1 + + ↑ + B D1 ↑ + C D2 2
```

其中D1和D2是显示表寄存器。将图15-38中的Graham-Glanville代码模板与这个序列匹配，并显示所产生的代码。给出所有匹配的模板并解释每一次选择特定模板的理由。

23. 扩展图15-38中的Graham-Glanville代码模板，使之包括新的模板来描述下列指令：
    - 将零加至任何操作数得到那个操作数而无需产生任何代码。
    - 寄存器与2相乘可被实现为那个寄存器到自身的加法。
    - 格式为Add Op1, Op2, Op3的三地址加法指令。其定义为Op3 := Op1+Op2。这三个操作数可以全部是寄存器，或其中之一是直接或变址地址（其他两个操作数则要求是寄存器）。

611

24. 假定我们设计的机器有两地址指令。它将有N个操作码和A种寻址模式。估计需要多少Graham-Glanville代码模板来描述该机器。

大多数真实的机器都不是正交的。即，并非所有寻址模式组合都能用于所有操作码。当引入非正交指令时，描述机器所需的Graham-Glanville代码模板集会变大还是变小？你能否给出可能需要的模板数目的上界（以N和A表示）？

25. 考虑语句A := A+B+C，其中A和C长型整数，B是字型整数。采用15.8.2节里属性化中间表示形式，该语句可以被表示为：

```
:= Index Obj(A,Long) Base(D1) + + Index Obj(A,Long) Base(D1)
     Index Obj(B,Word) Base(D2) Index Obj(C,Long) Base(D1)
```

使用15.8.2节里的技术和属性产生式为该语句生成代码。

26. 像15.8.2节那样创建属性代码模板以定义下列指令：
    - 操作数和值为2^N的常量相乘，$1 < N < Max$，可被实现为算术左移N位。
    - 将常量C，$1 < C < Max$，加至除R0外的任何寄存器，可被实现为LA C(R), R
    - 格式为Mult Opnd, R_i的乘法指令，要求i为偶数且R_{i+1}未被占用（因为乘积在R_i和R_{i+1}中构成）

27. 使用15.8.3节里的符号，无条件分支指令可被定义如下：

```
B Lab      PC ← Lab
```

其中PC是程序计数器。使用这个定义，解释窥孔优化器如何发现从一个无条件分支跳到另一个无条件分支的情况可以被“倒塌”为单个的无条件分支。

如果分支指令中的地址不是绝对地址而是相对于当前PC值的某个值，这种优化又该如何进行呢？ 612

28. 在完成一次窥孔优化后，替换原先指令的优化指令可被重新考虑并将作为另一次窥孔优化的一部分。给出可在这种级联的窥孔优化中受益的例子。 613

第16章 全局优化

16.1 概述——目标与限制

代码优化覆盖了众多的算法和启发式方法，它们试图改进由编译器生成的代码。我们已知有许多种优化的方法。其中一些优化方法非常简单而且几乎被所有的产品编译器所采用。常量表达式折叠和无用 614
指令删除是两个广泛使用的简单优化的例子。在指令顺序执行的单独基本块中我们应用了*局部优化*(local optimization)。由于局部优化不考虑控制流，因此它们较容易实现且常常与代码生成集成在一起。我们已在第15章里讨论过局部优化。

其他的优化方法往往过于复杂而只在特殊的*优化编译器*（optimizing compiler）中才使用。穿越基本块的程序变量到寄存器的指派，是一个相当难优化的例子。程序变量有很多而寄存器却很少，要最小化总体装入/存储开销将需要考虑非常多的可能的指派。那些必须处理穿越基本块的控制流的优化，我们称之为*全局优化*。它是本章的主题。

实践中，优化编译器很少生成真正最优的代码。其原因有两个，第一，优化中包含了那些已知是不可判定的问题。而不可判定问题就是那些不可能用通用算法去解决的问题。*可达性*（reachability）就是这样的一个问题。给定的代码片段在程序执行期间是否可达是不可判定的（见练习8）。可达性能影响优化行为，因为代码片段如果不可达，那么我们就可以安全地删除它。某些编译器特别采用了不可达的简单情况（例如，当条件表达式为常量值的时候），但是，一般来讲，优化算法假定程序中的所有代码在执行期间都是潜在可以到达的。

即使优化问题是可解的，其解决方案也可能非常的昂贵。例如，在第15章里，我们注意到从dag生成最优代码需要共享的子dag数目的指数级时间。通常，我们与其使用那些已知的非常昂贵的优化算法，还不如使用能产生较好但不是最好的代码的快速启发式方法。

将优化加入编译器中，其实也就是加入了一些提高生成代码质量的方法。衡量优化的准则有以下两条：*安全*和*效益*。施行优化后的程序可能不会和原先未优化的程序产生完全相同的结果。总能保证产生完全相同结果的优化是*安全的*；而那些可产生不同结果的优化则是*不安全的*。例如，一种常见的循环优化是将循环中已知为不变的（常量）表达式提出并放置到循环的首部。其想法是，仅计算一次该表达式并将其值存放到临时变量中。然而，该表达式在计算时可能会引发异常。除非我们知道原循环中总是要计算该表达式，否则将其移出循环体是不安全的。

许多优化在某些情况下是不安全的。它们包括：

- 重排可结合操作数。
- 移动程序中表达式和代码序列。
- 循环展开（与迭代执行循环不同，这是将循环扩展为循环体的一系列拷贝，但这样做可能会超出存储极限）。 615

尽管一个好的优化器应当只进行安全的优化（那些可提高性能又不会影响结果的优化），但很多有价值的优化在某些场合中却是不安全的。与其失去有潜在价值的优化，倒不如让一些编译器允许用户来做主是否进行那些不安全的优化。编译时警告或编译器文档给出了优化在哪些情况下可能危及程序的安全。

即使一种优化被认为是安全的，其最终的程序也未必能从中受益。在某些场合，这种优化实际上反

而降低了程序性能。循环不变式的移动证实了这种担心。即便我们知道循环不变式的计算不会引发异常，但实际情况却可能是：在原来的循环中该表达式从未被计算过。因此，在那种情况下，外提循环不变式并提前计算它将得不偿失。

通常，优化是被设计用来改进程序执行的平均性能，因而一般不可能在所有情况下均使程序受益。例如，对循环和过程调用进行优化，我们常常假定循环至少应迭代一次，而过程至少也要被调用一次。在那些假定条件不满足的极少数的情况下，做相关的优化将是徒劳的。

除了关心安全与效益外，我们试图去优化的对象可能发生变化，这取决于用户的需要。最常见的是我们力图去改善程序的速度和大小。有时，程序的成本（由用户埋单）或系统的开销（例如，页面调度或内外存交换）也会成为最迫切的关注对象。

通常，一种优化行为能满足所有合理的优化标准。例如，删除无用指令的优化，可以缩减程序代码大小，提高运行速度，并减少内存拥堵。然而，一些优化在改善某项性能时却会以牺牲另一项性能为代价。例如，子程序调用的内联展开提高了速度却导致程序代码大小的增加。

尽管我们是单独地介绍每一项优化，但我们也不能盲目地使用它们或者没有准备好就要开始合并使用它们。此外，施行优化的顺序也很重要，因为优化行为之间可能互相影响。例如，常量传播（在其中，我们识别出变量保存的值为常量值）可帮助我们识别不可达代码（通过折叠条件表达式）。一旦删除不可达代码，就可以传播新的常量值（如果有冲突的赋值是被删除代码的一部分）。

由优化引起的各种内部变化使得在运行时针对优化后程序的诊断调试非常混乱。例如，用于事后分析的内存卸出可能不会显示正确的变量值，因为有时候变量的值是保存在寄存器而非内存中的。类似地，代码移动也许会在与源程序列表所提示位置相差甚远的某个地方引发内存故障。

很自然地，人们可能只想优化那些正确的程序，但这往往是一厢情愿的。通常，最好的帮助是去调试那些未优化的程序版本（这常常有助于我们识别所有那些太过普通的情况，而就在其中，优化器本身可能会做不安全优化并引入错误）。调试优化代码的一般问题可参见文献Hennessy 1982和Zellweger 1983。

616

16.1.1 理想的优化编译器结构

图16-1显示了一个优化编译器的模型。这个理想的模型帮助我们将各种优化进行分类：

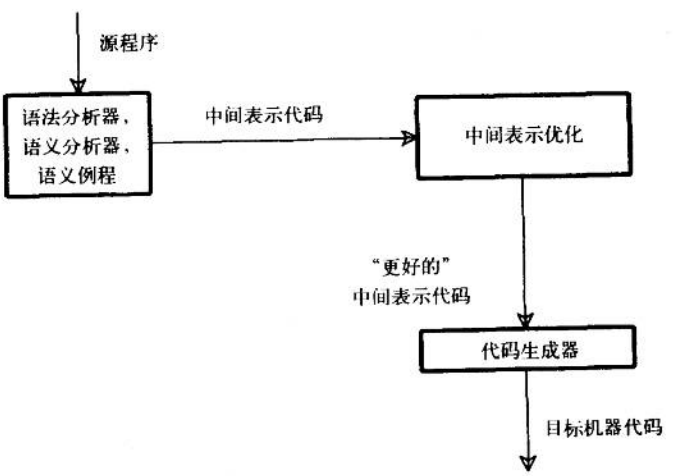


图16-1 理想的优化编译器

- **源语言优化**

  这部分优化在语义例程中完成，是语言特定的但与目标机器无关。

- **代码生成优化**

  这部分优化利用目标机体系结构，但基本上独立于源语言。

- **中间表示优化**

  理想情况下，这部分优化只依赖中间表示且可被许多用于不同源语言或目标机的编译器共享。

我们首先回顾一下在各个编译器阶段完成的各类优化。

源语言优化（在语义例程中完成）

只要有可能，语义例程就应当生成代码，这个代码可以利用出现在正被翻译的语言结构中的特殊情况。其中包括：

617

- 利用循环和数组的常量边界。
- 禁止为不可达的代码片段生成代码。
- 将循环体展开为等价的顺序代码：

```
for I in 1 .. 10 loop                A(1,1) := 2;
   A(I,I) := 2*I;          变为      A(2,2) := 4;
end loop;                            ...
```

- 压缩冗余的运行时检查。特别地，常量的循环边界允许将循环下标处理为约束子类型，这样就可能避免相应的范围或下标检查。

下面的优化可同样在语义例程中完成：

- 标记循环首部和出口以利于后面进行的流分析。
- 标记代码的分叉与回合点以利于确定直接前驱和后继。
- 标准化（规范化）操作数格式以利于公共子表达式的识别。（例如，我们可能为表达式A+B+C、A+C+B和C+A+B生成相同的代码。）

语言的设计对所生成代码的质量有着直接和主要的影响。良好设计的语言特性可使我们更容易地生成好的代码；而糟糕的设计则会带来糟糕的代码。可增强代码质量的语言结构包括：

- 有名常量（这样，变量就不必被当作常量使用）
- 赋值操作（例如C语言中的`A[i] += 1`）。冗余计算可以很一般地被识别出来。
- **case**语句，它能比等价的**if**语句产生效果更好的代码。
- 受保护的**for loop**下标，它可以被存放在寄存器中并确保被限制在固定范围内。
- 受限制的跳转和**goto**，它们使流分析更容易。

那些产生糟糕代码或禁止各种优化的语言特性包括：

- 按名传参，它被实现为“不可见”的过程调用，可用来取代按值传参或引用型参数。
- 有副作用的函数，它们使代码删除或代码移动成为不可能。
- 别名的创建，其中通过指针或引用型参数访问变量，它使冗余表达式的分析非常困难。
- 异常，它往往意想不到地(以不可见的方式)使程序控制转移到可能带来副作用的异常处理程序。Ada语言不允许在出现异常后恢复程序的正常运行；然而，PL/I语言却允许。

618

代码生成优化

我们在第15章中详细讨论了代码生成优化。在代码生成一级的优化往往利用了特定且详细的目标机知识。由代码生成器执行的典型优化包括：

- 谨慎分配与使用寄存器。
- 充分利用指令集。

- 充分利用硬件寻址模式。
- 利用特殊硬件设计（例如，流水线、高速缓存和异步功能单元）。

中间表示优化

我们需要识别出两个层次的中间表示（IR）优化：*局部优化*和*全局优化*。正如第15章所描述的，我们可以很方便地将一个程序表示为顺序代码段（称为*基本块*）的图形。基本块之间的控制流用有向边表示；程序中的所有基本块被链接在一起来体现控制流，它们共同组成*数据流图*（data flow graph）。

例如，下列代码片段：

```
if A = B then
    C := 1;
    D := 2;
else
    E : = 3;
end if;
A := 1;
```

619 其数据流图如图16-2所示。

在单个基本块内的优化称为*局部优化*。在基本块之间的优化（涉及控制流分析）称为*全局优化*。在下面的程序片段中：

```
A := B+C;
D := B+C;
if A>0 then
   E := B+C;
end if;
```

我们使用局部分析发现一个公共子表达式（Common Sub Expression，CSE），又利用全局分析发现另外一个CSE。

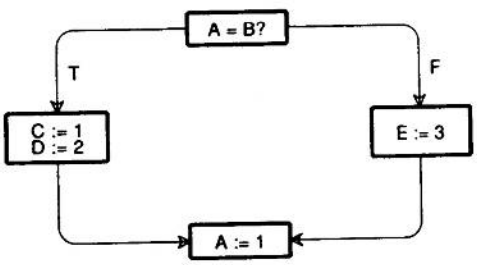


图16-2 数据流图示例

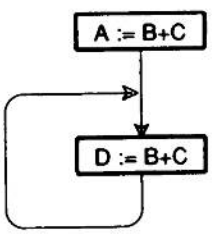


图16-3 用于全局CSE优化的数据流图

局部优化很容易进行（因为不需要流分析）且应当被大多数产品质量级的编译器所采纳。我们在讨论代码生成时包含了许多局部优化。将局部（基本块内）分析与代码生成集成在一起将允许我们改进代码质量以及更好地利用与机器相关的特性，尤其是寄存器。在这一章里，我们将集中讨论全局IR优化。这个关注点反映出这样一个事实，即：优化编译器和普通编译器的区别在于，前者要例行地进行流分析，来揭示全局优化；但后者却没有这么做。

穿越基本块的公共子表达式分析是一种常见的全局优化。例如，对于图16-3中的流图，全局分析可以确定B+C是一个CSE。

CSE优化，不论是局部的还是全局的，都是很有价值的。例如，程序中常用的数组下标就频繁地会导致公共子表达式的创建。

因为程序的大量执行时间往往花在循环体中，所以循环作为有用全局优化的一个来源，是特别重要的。

- 循环不变式可以移到循环入口且只计算一次。这是一项非常有用的优化，但如果我们不能确定此

表达式在循环中是否一定会计算，那么此优化就可能不太安全。在下面程序片段中： 620

```
while J > I loop
  A(J) := 10/I;
  J := J+2;
end loop;
```

可能出现I = 0的情况。如果10/I被移到循环入口处，那么由于我们的“优化”可能导致“被零除”的错误。即使移出的操作不会出错，仍然存在着“收益”的问题，因为如果循环执行零次的话，那么就不需要计算循环体中的表达式。

- 在**for loop**循环中，下标变量J有连续值J_0、J_0+1、J_0+2、…。如果b是循环不变式，那么形式为J*b的表达式有连续值J_0*b、J_0*b+b、J_0*b+2b、…。也就是说，该表达式的值在每个循环步后变化一个循环常量b。有鉴于此，我们可以在循环结束处用表达式的值与b的相加来删除该乘法。在这种优化里，我们说乘法被强度削弱为通常执行较快的加法。

16.1.2 优化展望

全局优化复杂、昂贵，有时还是不安全的。因此，很重要的一点是，我们通常是以一种“聚焦”的方式来使用全局优化。某些优化器只在循环上施行全局分析和优化，因为那里往往可以获得最大的收益。现代程序设计语言鼓励模块化，因此，降低子程序调用开销并提高其效率就显得十分重要。同样，仔细分析在施行其他优化时调用的效果也很重要。在本章里，我们将着重介绍子程序和循环优化，因为优化这些语言结构通常都会产生极佳的回报。

许多——实际上是大多数——程序并不需要优化。它们或是不经常使用，或是不经优化也能满足要求。但也存在至关重要的程序，它们是需要优化的对象。性能剖析程序有助于我们分析程序在运行时的行为并识别那些频繁执行的子程序和代码序列（通常是循环）。这些信息可以让优化器将其努力集中于关键的程序段上，而不是盲目地分析所有的对象。

令人惊讶的是，对于那些最关键的程序来说，光有优化是不够的。优化可以改进算法的翻译，但它却不能将一个糟糕的算法替换为一个更好的算法。好的优化器可将性能提高一个常量比率，如，可能减少程序代码大小或执行时间达20%~50%。而一个经过改进的算法却可以将一个线性算法改变成对数级算法或将一个n^2算法改变成n log n算法。因此，再多的优化也不能替代“聪明”的算法，而只能在它们被精心选择和实现后做“最后的增色”。 621

16.2 优化子程序调用

现代程序设计语言强调模块化。庞大的主程序被那些清晰定义的子程序所替代。不幸的是，许多编译器（或机器体系结构）使子程序的调用代价昂贵。在具有块结构的语言中，调用涉及引用环境中的改变、局部数据分配和参数值的传递。总之，现代的块结构的语言背负着不应有的低效率的名声。在本节里，我们将研究优化子程序调用的方法。通过谨慎处理，所获得的代码质量可以和那些无结构的语言程序的代码质量不相上下。

16.2.1 子程序调用的内联展开

在第13章里，我们讨论了将子程序翻译为*封闭子例程*（closed subroutine）的技术。在封闭子例程的情况下，调用点与返回点往往位于有着显著不同的代码体中。一种有时较有吸引力的方法是将子程序翻译为*开放子例程*（open subroutine），即在调用点*内联展开*（expanded inline）过程体。这有点类似于宏的展开，尽管简单的宏展开由于名字作用域的原因并不适合于像Pascal和Ada那样的结构化语言。某些语言（例如Ada和C++）允许程序员建议或指定哪些子程序可以用作内联展开。

子程序调用的内联展开节省了大量与调用相关的开销（保存寄存器、维护显示表、压入活动记录，等等）。事实上，此时施行其他优化也是可能的，因为调用中使用的实参在子程序体中变得可见了。特别是在实参为字面值的时候，可能有（常量）折叠和删除不可达代码的机会。已有人提出各种预先评估由内联展开所带来的节省的方法（Ball 1979）。

在讨论内联展开时，我们提三个问题：如何选择那些最值得做内联展开的调用？谁（用户或编译器）来做这个选择？以及如何正确地进行内联展开？

很明显，内联展开很大程度上是一种空间换时间的权衡，因为调用的展开几乎总是比封闭子例程调用占用更多的空间。此规律的一个明显但并不常见的例外是，子程序仅被调用一次。在这种情况下，内联展开将同时降低程序的时空需求。然而，一个递归子程序的内联展开，如果每次嵌套调用都是自身展开的话，将可能导致灾难性的后果。

为了判定子程序是否适合做内联展开，需要了解有关子程序之间相互调用的信息。这些信息可以很方便地表示在调用图（call graph）里。在调用图里，每个结点代表一个子程序或主程序。如果P调用Q，
622 那么从结点P到结点Q有一条边。对于形式过程参数的调用，将创建一条边指向每个可能与形式过程相绑定的子程序。

考虑图16-4中的调用图。主程序包含对A、B和C的调用。从A可以调用C或D。从B可以调用C。

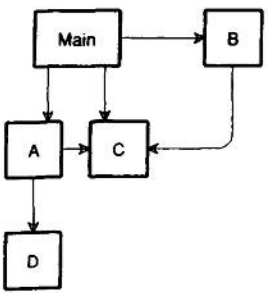


图16-4 调用图

每个结点的入边的数目表示有多少子程序可以调用它。图中的路径表示可能的调用序列。递归子程序很容易识别——在调用图中必定存在从递归例程到其自身的循环路径。类似地，只有一条边指向的例程是立即内联展开的候选者（如果调用它的那个惟一例程仅调用它一次的话）。

除了调用图以外，大小和调用频率等信息也能指导内联展开的选择。小型子程序是内联展开较好的候选，特别是那些过程体大小同普通调用所需代码大体相当的子程序。这在直觉上很简单——小型子程序若被实现为封闭子程序，那么它们花在调用上的时间会比它们完成预期功能的时间更多。事实上，那些简单的预定义的库子程序（如abs或round），通常都是以精确的内联展开来实现的，否则它们的执行速度将被调用/返回开销所吞没。

如果使用一个执行性能剖析器，则可以确定那些频繁被调用的子程序。如果没有性能剖析器，那些循环中出现的调用可能被想当然地认为是频繁执行的。频繁调用的子程序，即使它们的过程体不是特别小，它们也还是可以作为内联展开的候选者，因为不执行那些显式的调用和返回序列而节省的时间将随着“调用”次数的增加而成倍增加。

可以用前面介绍的标准来选择调用，这些调用是内联展开的候选对象。Ada语言包含一个形式为 **pragma** Inline(Name,…) 的语言指示，它允许用户选择适合做内联展开的子程序。C++允许函数被声明
623 为inline函数。但如果相应的函数不适合做内联展开，则那些由Inline语言指示或inline关键字所提供的建议将被忽略。因此，递归子程序通常不能做内联展开，若它们可以的话，则展开的深度将会受到限制。（受限制的递归子程序的内联展开允许像Factorial(5)那样的调用被全部展开，然后又收拢起来。）

一旦确定了内联展开的候选对象，我们就必须决定如何实现该内联展开。乍一看，用类似宏展开的方法可能较合适，但其实并非如此。这个问题在于，无论子程序调用被实现为普通的封闭子例程调用还是内联展开，它的调用效果必须是相同的。而宏展开由于作用域规则的原因，不能像普通的调用那样总是具有相同的语义。考虑:

```
declare
  A : Integer;
  procedure P(I : Integer) is
    J : constant Integer := I*2;
  begin
    Write(A,J);
  end P;
begin
  declare
    A : Boolean;
  begin
    P(1);
  end;
end;
```

如果用宏展开P的调用，那么将输出布尔值；而普通的调用则输出一个整数值。

与宏展开子程序调用为一段源程序不同，我们将子程序翻译为IR形式——或许是元组或许是“类似Diana的树”。非局部引用将被全部解析，以保持作用域规则；局部声明和参数的引用则被特别标记。当发生内联展开时，过程体上述的翻译形式将被替换。局部声明被处理为程序块中的声明；它们将扩展调用者的活动记录。子程序中所有的局部引用变为调用者活动记录空间内的引用。这意味着内联调用可以避免AR压入和显示表维护的开销。

调用的实参将替换子程序中的形参，替换过程中需要保持参数传递的语义。特别地，在子程序体执行前，每个参数仅被计算一次。因而，即使J在子程序体中可以被改变，A(J)的一个参数也总是表示相同的数组元素。非常量的标量参数必须被拷贝到临时变量；非标量的参数通常以引用方式传递，它们一旦被计算完毕，即被替换。 624

遵照这些规则，可将前面的例子翻译如下（尽管这是在IR一级上做的替换）：

```
B1: declare
      A : Integer;
    begin
      declare
        A : Boolean;
      begin
        declare
          J : constant Integer := 1*2;
        begin
          Write(B1.A,J); -- 引用正确的A
        end;
      end;
    end B1;
```

16.2.2 优化对封闭子例程的调用

在汇编语言程序中，常常通过一条能够保存返回地址并跳转到子例程开始地址的指令来实现对封闭子例程的调用。在结构化语言中，调用通常要昂贵得多，因为必须要维持活动记录、更新显示表、传递参数以及保存和恢复寄存器等。在本节里，我们研究在像Ada和Pascal那样的块结构语言中减少封闭子例程调用开销的方法。

一种常常很有价值的优化基于以下的观察：非递归子程序的活动记录可以采用静态分配，从而在这些子程序被调用时避免做任何AR或显示表维护工作。

在16.2.1节里，我们介绍了调用图的概念，它可以表示可能的调用序列。调用图中出现在循环路径上的任何子程序都是潜在递归的且必须在运行栈中压入或弹出它们的活动记录。调用图也表示了将AR映射到内存中的约束条件。如果存在从子程序P到子程序Q的路径，那么P可能会（直接或间接地）调用Q且因此它们的AR必须是不相交的。然而，如果P到Q（或Q到P）没有路径，则这两个子程序的AR可以覆盖。

为给活动记录分配静态地址，我们首先从调用图确认那些递归子程序，然后再从调用图中删去它们。（必须在运行栈上分配递归子程序的AR。）在删除递归子程序后，我们对调用图进行拓扑排序。拓扑排序是图中结点的一种简单列表，如果存在P到Q的边，那么P在列表中要先于Q。（只要图中没有循环，就总有可能得到这样的列表。）

625 重新考虑图16-4中调用图的例子。对此图来说，(Main,A,B,C,D)和(Main,B,A,C,D)都是它的拓扑排序。在大多数关于数据结构的书（如Knuth 1968）中都可以找到拓扑排序的算法。事实上，如果一种语言要求子程序在其被调用前要被完全声明，那么声明次序的逆序即为这些子程序的拓扑排序。

使用调用图结点的拓扑排序，我们能够形式化一个简单的用于活动记录分配的规则。它类似于在过程一级的AR中为块分配空间的方法：

(1) 根据一个拓扑排序来处理结点（即子程序），这样，一个子程序可先于它调用的任何子程序而被处理。

(2) 在分配完子程序的直接前驱所需的最大空间后立即分配该子程序所需空间。

例如，使用图16-4的调用图，我们首先处理Main。接下来，将分配A和B的活动记录，然后再分配C和D的活动记录。（静态）分配活动记录的结果如图16-5所示。

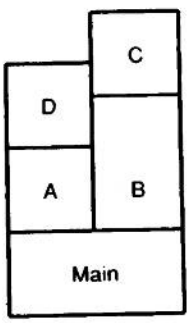


图16-5 静态分配活动记录的例子

我们可以很容易检验该分配是正确的。也就是说，两个共享空间的子程序不能同时处于活跃状态。同样，每个子程序被分配在最小可能的地址上，假定所有调用序列均是可能的。如同在过程一级为块分配空间一样，我们为活动记录静态分配的空间有时会比需要的大一些，这是因为我们将为所有可能的调用序列（其中包括那些可能从不发生的调用）预先分配空间。

动态数组和以往一样被处理并在运行栈顶被分配空间。有必要在每个子程序中保存stack_top的值，以便可以正确

626 地分配与释放动态数组和递归子程序（在运行栈上）所需的空间。

如果非递归子程序已静态分配了活动记录，其参数传递也将简化。在参数被计算后，它们可被直接放到被调者的AR中。然而，当函数出现在调用的参数列表（例如，F(a,b,c,G(x),…)中，情况会略有不同。我们不希望F和G完全相互覆盖，因为G的执行可能破坏那些已被计算并保存在F活动记录中的参数。另一方面，F和G也不需要完全不重叠，因为在调用F前，G已执行完成。

该问题的解决方案是：每个子程序在调用图中包括两个结点。一个结点表示存放子程序参数和控制信息的那部分活动记录；另一个结点表示存放子程序局部变量的那部分活动记录。如果P调用Q，那么我们建立从P的两个结点到Q的两个结点的边，它表示P的空间分配，包括参数和局部变量在内，都必须先于Q的空间分配。至于像F(a,b,c,G(x),…)的调用，我们建立从F的参数结点到G的两个结点的边，强制F的参数空间和G的空间分离。这种方法同样适用于诸如F(a,b,c, F(x,y,z,…),…)的部分递归调用。此时，可以在运行栈上分配F的参数空间，或者在不同的区域集中放置F的参数并仅在调用前将它们拷贝到相关的位置上。F的局部变量空间可以采用静态分配，因为（这个调用里）并没有真正的递归。

汇编语言程序经常用寄存器而非存储位置来传递参数，这也是一种很有价值的优化。通常，参数会被频繁地引用，因此将这些使用频繁的值指派到寄存器中是一个好主意。标量值参和引用参数的地址可以被有效地指派到寄存器中。对于这类参数，调用者只是简单地将实参装入合适的寄存器中，然后将控制转交给被调者，后者无需更多的努力就可以方便地访问这些参数。那些不能用寄存器传递的参数（非标量值参）将通过运行栈传递或直接被拷贝到被调者的活动记录中。

但是到底用哪些寄存器来传递参数呢？通常，编译器预先分配少量寄存器用作参数传递。调用的前

几个参数将传至所分配的寄存器中；而其他的参数，如果有的话，将通过运行栈传递或直接被拷贝到被调者的活动记录中。

这种方法简单但不灵活。如果我们分配更多的寄存器用于参数传递，它们将不能另作它用。如果我们分配用作参数传递的寄存器非常少，那么拥有许多参数的调用将因此“遭殃”。此外，为调用生成代码时将在寄存器使用方面存在不易察觉的冲突。考虑：

627

```
procedure P(A,B : Integer) is
begin
  ...
  Q(1,A);
  ...
end P;
```

在开始Q的调用时，参数寄存器到底包含P的参数还是包含Q的参数呢？如果我们考虑不仔细，就会在所有P的参数引用完成之前装入Q的参数。

解决办法之一是：在Q的调用前保存P的参数，并在计算Q的参数（并装入寄存器）时通过保存区引用P的参数。这种方法可行，但它会增加调用相关的寄存器保存/恢复开销。另一种办法是：按调用者和被调者使用的寄存器不重叠的方式将参数指派到寄存器。但这涉及到更多的最小化寄存器保存与恢复的一般性问题，我们将稍后讨论它们。

我们已经考虑了如何降低活动记录与显示表的维护以及参数传递的开销。最后需考虑的是寄存器的保存与恢复的开销问题。在某些情况下，此开销是不可避免的。尤其是，如果我们只有很少的寄存器，或者如果寄存器负荷很大，那么保存与恢复寄存器将不可避免。在这些情况下，我们所能做的就是使保存与恢复的开销尽可能低。这可能涉及到寄存器到保存区的块传送。一些RISC体系结构将自动地分配与释放某个寄存器集合（或窗口）作为子例程调用的一部分。其实，同局部变量一样，寄存器也有相应的“活动记录”。

如果可用寄存器数目比一个典型的子程序所需要的还多，我们可以通过细致的寄存器分配来降低寄存器保存与恢复的开销。这种想法很简单。如果过程P能够调用过程Q，我们试图在分配寄存器时使P的寄存器和Q的寄存器不相交。如果可以这么做，那么从P中调用Q时，就不需要做保存与恢复的工作了。

为减少保存/恢复的开销，我们用每一个子程序和主程序所需的临时寄存器数来标记它们。我们在分配真实寄存器之前做这项工作。就像活动记录优化那样，我们将以拓扑序遍历程序的调用图来决定如何将临时寄存器映射到真实寄存器。这分两步来做。首先，根据可能的调用路径，覆盖一些临时寄存器。如果P与Q之间不存在直接或间接调用关系，那么它们可以共享相同的临时寄存器。这与我们早前讨论的活动记录覆盖非常类似。

在我们覆盖临时寄存器后，我们可以将它们映射为实际的寄存器。如果临时寄存器比实际的寄存器要多，可以将同一个真实寄存器分配给多个临时寄存器，此时对于某些调用可能要强制生成保存/恢复代码。

作为示例，考虑图16-6，其中每一个结点用它所需的临时寄存器数来标记。整个程序需要12个临时寄存器，但如果考虑调用序列，则所需寄存器数可减少到9个。特别地，我们可以将寄存器1到4分配给Main；5到6分配给B，5、6和7分配给A，8分配给C，8和9分配给D。

628

如果有9个寄存器，相关工作可以顺利完成，根本不需要进行寄存器保存和恢复。如果寄存器数少于9个，则需要一些保存和恢复的工作。例如，假定我们有8个可分配的寄存器。那么，我们可将寄存器1重新分配给D。

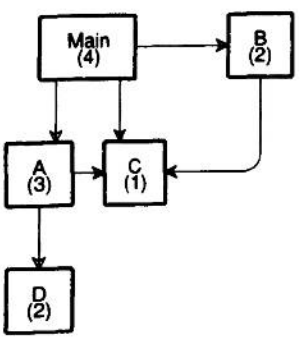


图16-6 标记所需寄存器数目的调用图

我们知道，寄存器的保存和恢复工作既可以由调用者也可以由被调者来完成，如果让调用者来完成此项工作，我们将保存和恢复那些被调者可能直接或间接使用的、但当前却由调用者占用着的寄存器。因此，在此例中，从Main中调用A将保存和恢复寄存器1，因为A可能调用D，而D使用寄存器1。

此外，如果被调者将用到的寄存器可由调用者直接或间接地使用，可以让被调者保存和恢复这些寄存器。采用这种方法，D将保存和恢复寄存器1，因为它被Main使用。如果我们让被调者做保存和恢复的工作，那么代码将缩小不少，这是因为每个子程序仅需要一个保存/恢复序列。

如何处理递归子程序呢？如果我们让被调者做寄存器的保存和恢复，那么无论分配了什么寄存器都无关紧要，因为保存和恢复工作将不可避免。如果由调用者来做的话，那么当调用者不在调用图上的任何环中时，子程序可以被分配到与环中它的最初的调用者不重叠的寄存器。这样，当子程序直接或间接调用自身时，保存和恢复工作只是为这些递归调用做的。

将参数和局部变量指派到寄存器中可以提高代码的质量。通过精心设计，还可以减少保存/恢复的
629 开销。尽管如此，仍然存在不易觉察的危险。指派到寄存器中的一个变量可被另一个子程序非局部地访问到。例如，我们有：

```
procedure P(A : Integer) is
  procedure Q is
  begin
    ...
    Write(A);
    ...
  end Q;
  ...
end P;
```

假定A被指派到寄存器。现在，根据寄存器被分配以及Q被调用（可能是间接调用）的情况，当在Q中引用A时，A的值有可能不在寄存器中。通过使用调用图来判定图中从P到Q的路径上A的寄存器是否已被重新指派，我们可以决定是否将A另存它处。假设是这样，我们如何取得A的正确值呢？简单的方法是将一个存储位置连同寄存器一起分配给A。在P中，通过它的寄存器来引用A；在P外，通过它的存储位置来引用A。

使用这种方法，无论何时在P中发生调用，A的值必须被保存在相应的存储单元（若调用过程中引用A）。我们的目标是减少调用/保存开销，为此，我们可以更灵活一点。如果我们要调用的子程序需直接或间接引用A，我们仅需将A保存到它的存储位置。如果我们能确定没有对A的引用，那么A的寄存器值则不需要被保存到它的内存单元中。在一次调用中读、写的变量集合可以通过过程间数据流分析（interprocedural data flow analysis）来确定，它是下一节要讨论的主题。

16.2.3 过程间数据流分析

在子程序的执行过程中，我们知道变量将被读取并修改。除非我们有办法判定哪个变量被“接触”，否则我们必须做最坏的假设。在第15章里，在基本块中施行局部优化时，我们这样做过。例如，我们假定在调用的过程中所有存放在寄存器中的变量的值均可能潜在地被读取或修改。类似地，我们假定在一次调用中任何变量均可能被修改，因此没有子表达式的值可以被保持并穿越此次调用（所有子表达式的值均被该调用注销）。其结果是，我们只能优化在调用之间但不穿越这些调用的基本块代码。

如果我们能够更加详细地推测一个调用的效果，我们就可以期盼做得更好一些。对调用效果的分析一般称为过程间数据流分析。在某些情况下，我们可以使用16.4节中的技术精确地分析子程序内部的控制流。而在其他情况下，我们只能使用调用图分析可能的调用序列。

630 与调用相关的两个简单但非常有用的集合是Def和Use。Def(P(A,B,…))是在P(A,B,…)被调过程中所有可被定义（即，被赋值）的变量的集合。Use(P(A,B,…))是在P(A,B,…)被调过程中所有被使用（即，

读取）的值（变量或有名常量）的集合。

Def和Use集合允许我们估计一次调用的可能效果。例如，如果在一次调用中用于计算子表达式的变量没有一个在Def集合中，那么该子表达式的值在此次调用后将保存不变。类似地，如果存放在寄存器里的变量不在某次调用的Use集合里，那么在此次调用前该变量就不需要被保存到它相应的内存单元里（尽管作为调用的一部分，可能仍然需要对它进行保存和恢复的工作）。此外，如果存放在寄存器中的变量出现在某次调用的Def集合里，那么在此次调用后这个寄存器原先的值将不再有效。因此，在调用后该寄存器要么已被释放，要么已被装入修改后的值。无论是哪一种情况，该寄存器都不会被认为是在调用点使用的，且因此不需要作为调用的一部分而加以保存或恢复。

可用下面的方法计算得到Def和Use集合。为简单起见，假定所有对象名均不相同（如果需要，我们可以创建惟一的内部名）。我们首先考虑子程序没有参数的简单情况。

开始的时候，我们确定那些由子程序读取或写入的变量和有名常量。这些集合被称为LocalDef(P)和LocalUse(P)。这两个集合在编译子程序时很容易建立。当语义例程生成访问有名常量或变量值的代码时，那些对象名将被包括在当前过程的LocalUse里。类似地，在生成修改变量值的代码时会将对象名包括进LocalDef中。由于别名问题，我们假设对数组元素的引用将波及到整个数组。也就是说，如果我们看见语句A(i) := 1;，我们并不能准确地知道哪个元素被改变了，因此，我们简单地把A包括到LocalDef中。

现在，考虑P产生的调用效果。设Called(P)是在P中直接调用的子程序的集合。该集合可简单地由调用图得到。于是有：

$$Use(P) = LocalUse(P) \bigcup_{Q \in Called(P)} Use(Q)$$

$$Def(P) = LocalDef(P) \bigcup_{Q \in Called(P)} Def(Q)$$

这些方程是递归的；我们将搜寻与这些方程相容的最小集合。可以用迭代法获得这样的解：首先，所有子程序的Use(P)和Def(P)分别近似取值为LocalUse(P)和LocalDef(P)。然后，通过包含与Called中的子程序相应的集合来更新Use和Def集合。重复进行这个更新过程直至Use和Def集合不再发生任何改变。定义在图16-7中的例程compute_use_set()使用了该方法。（compute_def_set()的算法与之类似。）有关该迭代方法正确性的证明将作为练习15留给读者来做。 631

```
void compute_use_set(void)
{
    /*
     * Compute Use sets for subprograms
     * including the effects of calls.
     */

    for (S ∈ Subprogram_set)
        Use(S) = local_use(S);

    changes = TRUE;

    while (changes) {
        changes = FALSE;
        for (S ∈ Subprogram_set) {
            for (P ∈ Called(S)) {
                if ( ! (Use(P) ⊆ Use(S)) ) {
                    Use(S) = Use(S) ∪ Use(P);
                    changes = TRUE;
                }
            }
        }
    }
}
```

图16-7 计算Use集合的算法

作为示例，考虑如下程序：

```
declare
  A,B,C,D,E : integer;

  procedure Q is
  begin
    Write(A);
  end Q;

  procedure P is
  begin
    A := B + C;
    Q;
  end P;
begin
  C := A + B;
  E := C + D;
  P;
  ...
end;
```

632

通过检查子程序体，我们得到：

LocalUse(P) = {B,C}　LocalDef(P) = {A}
LocalUse(Q) = {A}　LocalDef(Q) = ∅

我们首先将Def和Use集合近似为相应的LocalDef和LocalUse集合：

Use(P) = {B,C}　Def(P) = {A}
Use(Q) = {A}　Def(Q) = ∅

因为P调用Q，Q的定义（Def）和使用（Use）集合被包括到P的集合中：

Use(P) = {A,B,C}　Def(P) = {A}
Use(Q) = {A}　Def(Q) = ∅

由于没有其他的调用，因此对Def和Use集合不会做进一步改变，计算也就此完成。借助这个信息，我们能够确定在调用P后A+B已被注销，但C+D依然有效。此外，如果A、B、C、D和E已在寄存器中，那么A、B和C在调用前就必须要保存，这是因为它们的值要被使用，而且A由于其值已改变还必须从内存中重新装入。使用16.4节里的流分析技术，我们可以做得更好一些：因为A是定义在前、使用在后，因此在调用前保存A不是真的有必要。我们也注意到，其他的寄存器如果会被P或Q另作他用，也可能要被保存；然而，使用16.2.2节中的技术来分配寄存器有可能最小化这些额外的保存。

我们现在考虑有参数的子程序调用。设F为形参名。如果F在过程P中被使用或定义，那么我们把F包含在集合FormalUse(P)或FormalDef(P)中。我们最初把F包含在集合Formals(F)中。该集合是所有可以代表F的形参名。因为一个子程序的形参可以作为实参传递给另一个子程序，所以我们必须如下计算Formals：

```
if (A ∈ Formals(F) &&
      A在调用中作为实参出现，其所对应的形参为G) {
    Formals(F)=Formals(F)∪G
}
```

633

集合Formals(F)很有用，因为对形参G（其中，G ∈ Formals(F)）的引用，可能就是引用F。这样的间接引用必须要包括在Def和Use集合里。

我们首先考虑Ada语言中使用的三种参数传递模式：**in**、**out**和**in out**。设I为形式**in**参数。那么I将被看成是一个有名常量且只能读不能写。我们可以查看某个名字F ∈ Formals(I)是否出现在某个子程序的FormalUse集合里。如果没有出现，就说明I被不正确地使用，编译器将给出合适的诊断警告。在任何情况下，我们都假定任何作为in参数传递的变量或有名常量在调用期间被直接或间接地使用。

类似地，设O为**out**参数。O可写但不可读。我们可以查看某个名字F ∈ Formals(O)是否出现在某个子程序的FormalDef集合里。如果没有出现，就说明O被不正确地使用，编译器将给出合适的诊断警告。我们假定任何作为**out**参数传递的变量在调用期间被直接或间接地定义。

设IO为**in out**参数。IO既可读又可写。我们可以检查名字某个F ∈ Formals(IO)是否出现在某个子程序的FormalUse集合里，以及检查某个名字G ∈ Formals(IO)是否出现在某个子程序的FormalDef集合里。如果未同时发现IO的定义和使用，则IO要么未被使用，要么正被误用作**in**或**out**参数。无论是哪一种情况，编译器都将给出诊断警告。我们假定任何作为**in out**参数传递的变量在调用期间都被直接或间接地定义和使用。对于作为**in out**参数传递的结构数据对象，有时在子程序中我们仅看见对它的部分成员进行赋值的情况。虽然此时该结构数据对象的其他成员并未改变，但我们还是假定那一部分成员有隐式的使用（以便将它们拷贝到已更新的结构对象中）。

Pascal语言中的值和引用参数模式更为微妙。在Pascal中，值参类似于**in**参数。初始化为相应实参值的值参可当作局部变量来使用。就像我们为**in**参数做的那样，对于值参Val，我们将查看某个名字F ∈ Formals(Val)是否出现在某个子程序的FormalUse集合里。对Formals(Val)中的名字进行写也是可能的。使用16.4节中的技术，我们或许想查看一个值的使用是否总是先于它的任何一个定义。这项检查很有用，因为经验显示，Pascal的值参（默认参数模式）常常和**var**参数相混淆，而定义先于使用是这种混乱的有力见证。我们假定任何作为值参传递的变量或有名常量在调用期间被直接或间接地使用。

Pascal的**var**参数可用作**in**参数（以避免值参的隐式拷贝），也可用作**out**参数，还可用作**in out**参数。如果Var是Pascal的**var**参数，那么根据任意G ∈ Formals(Var)是否出现在一个FormalUse集合或一个FormalDef集合中，或是在两个集合中都出现，我们将Val描述为**in**、**out**或**in out**参数。

如前所述，我们将Pascal和Ada中的参数模式分类为**in**、**out**或**in out**。此外，还假定形参被正确地使用，即，**in**参数将总被使用，**out**参数将总被定义，**in out**参数将总被使用和定义。因此，对于作为实参传递变量和有名常量，我们根据传递它们的参数模式将它们添加到LocalDef或LocalUse集合中。

现在，我们可以将参数包括进Def和Use集合中：

634

$$Use(P(a_1,\ldots,a_k)) = Use(P) \cup \{a_i \mid \text{name } a_i \text{ is an } \textbf{in} \text{ or } \textbf{in out} \text{ parameter}\}$$
$$Def(P(a_1,\ldots,a_k)) = Def(P) \cup \{a_i \mid \text{name } a_i \text{ is an } \textbf{out} \text{ or } \textbf{in out} \text{ parameter}\}$$

扩展我们先前的示例，考虑：

```
declare
  A,B,C,D,E : Integer;

  procedure Q(Z : out Integer) is
  begin
    Z := 1;
    Write(A);
  end Q;

  procedure P(I : in Integer; J : in out Integer) is
  begin
    E := I + J;
    A := B + C;
    Q(J);
  end P;
begin
  C := A + B;
  E := C + D;
  P(B,C);
  ...
end;
```

我们首先计算Formals集合：

Formals(I) = {I}　Formals(J) = {J,Z}　Formals(Z) = {Z}

然后，我们计算FormalDef和FormalUse集合：

FormalUse(P) = {I,J}　FormalDef(P) = ∅
FormalUse(Q) = ∅　FormalDef(Q) = {Z}

I是**in**参数且由于它仅出现在FormalUse集合中，所有I的使用是正确的。类似地，Z是**out**参数且由于它仅出现在FormalDef集合中，所有Z的使用也是正确的。最后，J是**in out**参数且由于J出现在Formaluse(P)中以及Z ∈ Formals(J)出现在FormalDef(Q)中，所有J的使用也是正确的。

635 P和Q的LocalDef和LocalUse集合在我们包含作为实参传递的变量时并未改变；因此，Use和Def集合也不会改变：

Use(P) = {A,B,C}　Def(P) = {A,E}
Use(Q) = {A}　Def(Q) = ∅

在P(B,C)中，B是**in**参数而C是**in out**参数，因此

Use(P(B,C)) = {A,B,C}　Def(P(B,C)) = {A,C,E}

利用以上信息，我们可以确定在调用P后，A+B和C+D均被注销。此外，如果A、B、C、D和E已在寄存器中，那么A、B和C可能将不得不在调用前进行保存（因为会用到它们的值），而且A和C还可能不得不从内存中重新装入（因为它们的值可能已被改变）。

注意：我们计算Formals、FormalDef和FormalUse集合仅是用来检查参数是否按它们的模式所说的那样被正确地使用。为简单起见，我们可以例行假定所声明的参数模式准确地描述了如何使用实参的方式。接着，为进行分析，我们仅需初始化LocalDef和LocalUse集合（包括实参），并从它们开始计算Def和Use集合。即使形参的使用没有按照它们的模式所指示的来进行，我们的分析依然会正确，但可能过于保守些。（例如，**in out**参数被用作**in**参数，但我们还是假定它既被定义又被使用。）

16.3 循环优化

计算机科学中最著名的说法之一是“90/10规则”——程序90%的执行时间花在10%的代码上。这种认识与优化是密切相关的。与其试图优化每件事，倒不如明智地去寻找那些一旦优化必将产生极大改进的“热点”。

运用剖析工具定位那些性能至关重要的热点区域会十分理想。在优化期间，虽然缺乏实际运行数据，但是我们对循环（尤其是嵌套的循环）还是会予以特别的关注。在第12章里，我们用特殊办法对循环进行有效的翻译。不仅如此，在第15章里讨论的局部优化对于循环体而言也特别有价值。也就是说，在循环体中将变量指派到寄存器、追踪寄存器内容和避免冗余计算等优化是非常值得做的。事实上，依据90/10规则，我们有理由认为，在除循环体和其他负荷很重的代码片段以外的地方应禁止局部优化。此举将加快翻译的速度而不会显著影响代码的质量。

在下面的章节里，我们将讨论特别适用于循环的优化。我们的讨论将集中在那些能产生显著优化且
636 相对容易实现的技术上。

16.3.1 外提循环不变式

一个非常流行的循环优化是将循环不变式从循环体中外提到循环首部。简单地说，循环不变式（loop-invariant expression）就是那些在循环体中值保持不变的表达式。如果循环不变式出现在循环体中，它将在每次迭代时重复计算。一个明显的优化是将循环不变式外提到循环的首部，在那里该表达式仅被计算一次。作为示例，考虑图16-8所示的嵌套循环。假定数组采用行主序分配，且这些循环被翻译为如图16-9所示的程序样式。

```
for I in 1..100 loop
  for J in 1..100 loop
    for K in 1..100 loop
      A(I,J,K) := I*J*K;
    end loop;
  end loop;
end loop;
```

图16-8 三重嵌套循环

```
for I in 1..100 loop
  for J in 1..100 loop
    for K in 1..100 loop
      A(I)(J)(K) := (I*J)*K;
    end loop;
  end loop;
end loop;
```

图16-9 外提循环不变式之前的三重嵌套循环

在此例中，很容易发现I*J和A(I)(J)是最内层循环中的不变式，因此它们的计算可以移到中间层循环里。进一步地，A(I)在中间层循环内仍为不变式，因此它的计算可移至最外层循环中。

为了外提循环不变式，我们必须首先识别那些可以移动的表达式。同样重要的是，我们也必须考虑安全与优化的效益。借助前面章节开发的技术，我们将不难识别循环不变式。设LoopDef是那些在循环体中有定义（即，被赋值）的变量集合。LoopDef不仅包含那些在循环体中显式被赋值的变量，也包含那些在（循环内）过程调用中潜在被改变的变量以及循环下标自身。在前面的例子中，最内层循环的LoopDef是{A,K}，中间层循环的LoopDef是{A,J,K}，最外层循环的LoopDef是{A,I,J,K}。 637

表达式的*相关变量*（relevant variable）是那些用于表达式计算的变量。如果表达式的相关变量没有一个出现在集合LoopDef中，则该表达式是循环不变式。例如，A(I)(J)依赖I、J和A的地址，因此在最内层循环中是循环不变的。I*J在最内层循环中也是循环不变的。它们两个均可被移到该循环首部。因为A(I)在中间循环层为循环不变的，所以它可被再次移到中间循环的首部。

出现在循环体中的表达式集合，以及LoopDef中的变量，可在翻译循环，即计算集合Def时计算。接着，循环不变式可以被识别并在代码生成前被移出循环。图16-10中定义的例程mark_invariants()识别循环不变式集合。在图16-11中定义的例程factor_invariants()外提循环不变式到循环首部。

```
void mark_invariants(loop L)
{
    /*  Find expressions invariant in loop L. */

    Compute LoopDef for loop L;

    Mark as loop-invariant all expressions whose
     relevant variables are not members of LoopDef;
}
```

图16-10 识别循环不变式的算法

```
void factor_invariants(loop L)
{
  /*
   * Find expressions invariant in loop L
   * and move their calculation outside the loop body.
   */

  mark_invariants(L);

  for (each expression E marked as loop-invariant) {
     Allocate a new temporary T;
     Replace each occurrence of E in L with T;
     Insert T := E in L's header code immediately after
       the first loop termination test;
     /*
      * If L must iterate at least once
      * T := E may be placed immediately before L.
      */
  }
}
```

图16-11 外提循环不变式的算法

因为循环不变式可以被外提多层循环，所以factor_invariants()应当首先应用于最内层循环中，然后才是包含它的外围循环，等等。将factor_invariants()用于图16-9中的最内层循环，我们得到如图16-12所示的代码。

接着，factor_invariants()被用于中间层循环，得到如图16-13所示的代码。最后，factor_invariants()被用于外层循环，但未发现循环不变式，此时程序将保持不变。尽管可能不那么直观，但是外提图16-9中程序的循环不变式还是产生了较好的性能。原来的嵌套循环将执行300万次下标操作和200万次乘法。图16-13中优化后的代码将执行1 010 100次下标计算和1 010 000次乘法，这是非常显著的改进。

```
for I in 1..100 loop
   for J in 1..100 loop
      Temp1 := Adr(A(I)(J));
      Temp2 := I*J;
      for K in 1..100 loop
         Temp1(K) := Temp2*K;
      end loop;
   end loop;
end loop;
```

图16-12 循环不变式已移出最内层循环的嵌套循环

```
for I in 1..100 loop
   Temp3 := Adr(A(I));
   for J in 1..100 loop
      Temp1 := Adr(Temp3(J));
      Temp2 := I*J;
      for K in 1..100 loop
         Temp1(K) := Temp2*K;
      end loop;
   end loop;
end loop;
```

图16-13 循环不变式已移出的嵌套循环

不是所有的循环不变式都能被安全地或有利可图地从循环体中外提出来。特别地，如果不变式的计算在循环首部出现“故障”，将会发生什么呢？优化应当保持程序的语义，而且我们可能需要忽略或延
638 迟这些故障的处理。

在理想状况下，在外提的表达式出现故障时，我们会去截取那个故障，并在循环体使用该表达式的时候重新将它引发。这实在很“狡猾”，因为我们不想在循环体中添加代码来测试那些悬而未决的故障。例如，我们或许会将出错的表达式替换为一个在被访问时能引发故障的“错误值”。此外，我们还可以关闭对存放外提值的位置的访问。如果目标机体系结构不能使这些方法可行，我们或许可以把循环体中
639 对外提值的引用替换为执行时会引发故障的非法操作码。（这涉及到在执行期间改变程序代码，因此不适合于共享或重入式代码。）如果没有别的办法可用，我们可以设法拥有两份循环体拷贝，一个是优化的，另一个是未优化的。如果外提循环不变式计算时出错，那么错误将被忽略，且程序控制此时转移到那个未优化的代码，在那里不变式将在它原来的位置上被计算。

除了安全性以外，我们还必须考虑获益情况。循环可能执行零次（即，一次也不执行）或循环体中的控制流可能不会到达循环不变式。在其中任何一种情况下，如果外提不变式，我们将计算一个可能用不到的表达式——这简直就不是优化！在某些情况下，仅通过检查循环边界，我们就可以确定循环至少会执行一次（前面的例子就是这种情况）。如同12.2.2节里所描述的，我们能够生成**for loop**代码，使得我们可以首先测试循环体是否执行，然后再测试循环是否执行多次。如果循环不变式被移到初始测试之后，那么我们知道，仅当循环至少执行一次时它们方能计算。

while loop可以迭代零次，因此，外提不变式到**while loop**的首部可能不安全。许多编译器忽略安全考虑而执意外提不变式。更谨慎的编译器仅外提那些不会出错的表达式。

如果一个表达式的值将在所有可能的执行路径上使用（参见16.4.2节），则该表达式非常忙。对于在循环体中一个非常忙的循环不变式，如果循环执行至少一次，那么该不变式将总被使用。非常忙的循环不变式是最佳的外提候选。在我们的示例中，循环体是一个简单的基本块，因此，其中所有的表达式都非常忙。

如果表达式不是非常忙，我们或者会很小心而不把它外提，或者我们坚持外提它。可能最好的方法是使用通过剖析收集的数据来估测循环执行期间不变式的使用频度。如果所期望的使用频次大于1，那么平均而言，外提不变式将是有益的且应当在编译期间被允许进行。（很少有产品编译器实际使用剖析数据。） 640

16.3.2 循环中强度削弱

有时，代价昂贵的操作可被代价低廉的操作所取代而得以*强度削弱*（reduced in strength）。循环中，乘法有时可用加法来替换。因为乘法通常比加法慢3～10倍，因此强度削弱能够产生显著的加速。

（循环中的）*归纳变量*（induction variable）是那些其值有规律地以常量值递增或递减的变量。Pascal和Ada语言中仅允许单位步长；而其他语言（如FORTRAN）允许较大的步长。Aho等人已提出寻找循环中归纳变量的算法（Aho、Sethi和Ullman 1985，算法10.9），但这里，我们只把注意力集中在最常见的一类归纳变量——**for loop**的下标。

我们定义形式为$i*c_1+c_2$的表达式为归纳表达式，其中i是归纳变量，c_1、c_2是循环不变式。我们知道归纳变量i的值的步进序列为：i_0、i_0+s、i_0+2s、…，其中是i_0归纳变量i的初值，s是步长（Ada和Pascal中步长为±1）。我们的归纳表达式将因此通过下列的值序列步进：$i*c_1+c_2$、$i_0*c_1+c_2+s*c_1$、$i_0*c_1+c_2+2s*c_1$、…。在每一步，归纳表达式以步长$s*c_1$变化。

我们可用初值为$i_0*c_1+c_2$的临时变量取代每个归纳表达式，且在每次迭代的末尾将此临时变量的值增加$s*c_1$。这种替换是有益的，因为在原来每次迭代时执行的乘法（以及可能的加法）现已被每次迭代末尾执行的加法所取代。

为实现强度削弱，我们首先要识别归纳表达式。这比较容易，因为那些包含循环下标的表达式可被轻而易举地识别且循环不变式也可用16.3.1节中的mark_invariants()标记出来。每个归纳表达式被分配一个临时变量，且在循环首部和结尾处分别插入必要的初始化和增值代码。在循环中执行强度削弱的算法——strength_reduce()的定义如图16-14所示。

```
void strength_reduce(loop L)
{
  /*
   * Find induction expressions in loop L
   * and strength reduce them.
   */

  mark_invariants(L);

  for (each expression E in L of the form I*C+D
       where I is L's loop index and C and D involve
       only marked loop-invariants) {
    Allocate a new temporary T;
    Replace each occurrence of E in L with T;
    Insert T := I0* C + D immediately before L
      where I0 is the initial value of I in L;
    Insert T := T + S * C at the end of L's body
      where S is the step size by which I is incremented;
    /* S is negative if I is decremented. */
  }
}
```

图16-14 循环中执行强度削弱的算法

作为示例，再次考虑图16-13中已外提循环不变式的程序。表达式I*J和Temp2*K是归纳表达式。在三重嵌套循环中的每一层调用strength_reduce()，其结果代码如图16-15所示。

复写传播（copy propagation，参见16.4.4节）是一种优化，它识别在形为A := B的赋值语句后，如果A或B在赋值后均未改变，对A的引用可被替换为B。对图16-15中的代码应用复写传播，我们看到，

Temp2必然含有Temp4的值，并因此所有对Temp2的引用可被替换为Temp4。此优化给我们带来如图16-16所示的更加简单的代码。

```
for I in 1..100 loop
   Temp3 := Adr(A(I));
   Temp4 := I; -- Initial value of I*J
   for J in 1..100 loop
      Temp1 := Adr(Temp3(J));
      Temp2 := Temp4; -- Temp4 holds I*J
      Temp5 := Temp2; -- Initial value of Temp2*K
      for K in 1..100 loop
         Temp1(K) := Temp5; -- Temp5 holds Temp2*K = I*J*K
         Temp5 := Temp5 + Temp2;
      end loop;
      Temp4 := Temp4 + I;
   end loop;
end loop;
```

图16-15 强度削弱后的嵌套循环

```
for I in 1..100 loop
   Temp3 := Adr(A(I));
   Temp4 := I; -- Initial value of I*J
   for J in 1..100 loop
      Temp1 := Adr(Temp3(J));
      -- Temp4 holds I*J
      Temp5 := Temp4; -- Initial value of Temp2*K
      for K in 1..100 loop
         Temp1(K) := Temp5; -- Temp5 holds Temp2*K = I*J*K
         Temp5 := Temp5 + Temp4;
      end loop;
      Temp4 := Temp4 + I;
   end loop;
end loop;
```

图16-16 强度削弱和复写传播后的嵌套循环

另一个有价值的强度削弱隐藏在下标操作里。假定数组A被定义为A: **array**(1..100, 1..100,1..100) **of** Integer。我们知道在下标操作中“隐藏”着乘法。具体地说，Adr(A(I))可扩展为
641 A_0+(10000*I)−10000，其中A_0是A的开始地址。这个看似复杂的表达式是归纳表达式!

类似地，Adr(Temp3(J))扩展为Temp3+(100*J)−100，它也是归纳表达式。最后，Temp1(K)扩展为(Temp1+K−1)↑，其中↑是间接操作符。而Temp1+K−1也是归纳表达式。将这些扩展后的下标表达
642 式替换到图16-16的代码中，则得到如图16-17所示的代码。

```
for I in 1..100 loop
   Temp3 := A0+(10000*I)-10000;
   Temp4 := I; -- Initial value of I*J
   for J in 1..100 loop
      Temp1 := Temp3+(100*J)-100
      -- Temp4 holds I*J
      Temp5 := Temp4; -- Initial value of Temp4*K
      for K in 1..100 loop
         (Temp1+K-1)↑ := Temp5; -- Temp5 holds Temp4*K = I*J*K
         Temp5 := Temp5 + Temp4;
      end loop;
      Temp4 := Temp4 + I;
   end loop;
end loop;
```

图16-17 下标代码展开后的嵌套循环

在每个循环中应用strength_reduce()，我们得到图16-18中所示的代码。再次地，复写传播通过删除Temp1和Temp3对程序进行整理，产生如图16-19所示的代码。

```
Temp6 := A0 -- Initial value of Adr(A(I))
for I in 1..100 loop
  Temp3 := Temp6;
  Temp4 := I; -- Initial value of I*J
  Temp7 := Temp3 -- Initial value of Adr(A(I)(J))
  for J in 1..100 loop
    Temp1 := Temp7;
    Temp5 := Temp4; -- Initial value of Temp4*K
    Temp8 := Temp1 -- Initial value of Adr(A(I)(J)(K))
    for K in 1..100 loop
      Temp8↑ := Temp5; -- Temp5 holds Temp4*K = I*J*K
      Temp5 := Temp5 + Temp4;
      Temp8 := Temp8 + 1;
    end loop;
    Temp4 := Temp4 + I;
    Temp7 := Temp7 + 100;
  end loop;
Temp6 := Temp6 + 10000;
end loop;
```

图16-18 下标代码强度削弱后的嵌套循环

```
Temp6 := A0 -- Initial value of Adr(A(I))
for I in 1..100 loop
  Temp4 := I; -- Initial value of I*J
  Temp7 := Temp6 -- Initial value of Adr(A(I)(J))
  for J in 1..100 loop
    Temp5 := Temp4; -- Initial value of Temp4*K
    Temp8 := Temp7 -- Initial value of Adr(A(I)(J)(K))
    for K in 1..100 loop
      Temp8↑ := Temp5; -- Temp5 holds Temp4*K = I*J*K
      Temp5 := Temp5 + Temp4;
      Temp8 := Temp8 + 1;
    end loop;
    Temp4 := Temp4 + I;
    Temp7 := Temp7 + 100;
  end loop;
Temp6 := Temp6 + 10000;
end loop;
```

图16-19 下标代码强度削弱和复写传播后的嵌套循环

产生的代码比原来的三重嵌套循环难读得多，但它相当快——这也就是优化的目的！事实上，现在我们可以看到J和K在它们各自的循环体中都没有被引用（自身的增值除外）。这就有可能根据那些被引用（且增值变化）的值（即，那些可从J和K导出的归纳表达式）来重写循环的控制。 643

尽管本节集中讨论的是乘法的强度削弱问题，但其他操作的强度削弱也同样是可能的。例如，表达式length'(A&B)可被强度削弱为length' (A)+length' (B)。这是一个明显的优化，因为它避免了串A和串B的连接操作。Fong和Ullman(1976)对复杂操作的强度削弱进行了研究。 644

16.4 全局数据流分析

全局优化（跨越多个基本块的优化）通常依赖于对程序中所有可能的执行路径的分析。对数据值在穿越基本块时的修改情况的分析是全局数据流分析（global data flow analysis）。一般而言，我们不可能事先准确知道程序究竟会执行哪一个基本块序列。因此，我们执行的数据流分析假定通过流图的所有路径都是可能发生的。无论一个特定的程序执行何种路径，基于这种分析的优化都是有效的。

16.4.1 单路径流分析

我们通过求解涉及活跃变量识别的一个相当简单的问题来开始数据流分析的介绍。我们知道，活跃变量是那些在被赋予新值以前，其当前值还要被使用的变量。在全局活跃变量分析的语境中，我们认为，如果沿着某一条路径，一个变量在被重新定义前可能被使用，则这个变量是活跃的。

在第15章里，我们假定所有变量在基本块末尾都是活跃的。借助全局活跃变量分析，我们可以识别那些当前值已不活跃（即，变量值已“死亡”）的变量。非活跃变量的值不需要在基本块末尾加以保存。类似地，如果在子程序的入口得知某变量的值非活跃，那么我们就不需要在调用前保存这个变量的值，即使该变量的值会被这个子程序所使用。

设b为基本块索引。定义LiveIn(b)为到达基本块b入口的活跃变量集合。类似地，定义LiveOut(b)是离开基本块b出口的活跃变量集合。LiveIn和LiveOut集合不是相互独立的。设S(b)为流图中基本块b的所有后继的集合。我们有：

$$LiveOut(b) = \bigcup_{i \in S(b)} LiveIn(i)$$

即，变量在一个基本块出口处活跃，同时它也在该基本块的某些后继块的入口处活跃。如果一个基本块没有后继块，那么它的LiveOut集合为空。

设LiveUse(b)是基本块b中那些被定义前要被使用的变量集合。LiveUse(b)是常量集合（constant set），它的值只由出现在基本块b中的语句决定。很容易得知，如果$v \in LiveUse(b)$，那么$v \in$
645 LiveIn(b)；即，$LiveIn(b) \supseteq LiveUse(b)$。

设Def(b)是那些在基本块b中定义的变量的集合。同样，Def(b)也是一个常量集合，它的值也只由出现在基本块b中的语句决定。我们可在建立基本块b的时候计算Def(b)。

如果一个变量在基本块b出口处活跃，它要么在b中定义，要么在b的入口处活跃。也就是说，$LiveIn(b) \supseteq LiveOut(b) - Def(b)$。经仔细考虑，我们可以证明，变量在基本块入口处活跃的条件只能是：要么它在基本块的LiveUse集合里，要么它在基本块出口处活跃且没有在基本块里被重新定义。此分析导致以下方程：

$$LiveIn(b) = LiveUse(b) \cup (LiveOut(b) - Def(b))$$

针对每个基本块，都有这种形式的方程将LiveIn和LiveOut联系起来。全局活跃变量分析必须为每个基本块计算与它们的数据流方程解一致的LiveIn和LiveOut集合。

为使分析更具体，我们考虑以下示例：

```
A := 1;
If A=B then
      B := 1;
else
      C := 1;
end if;
D := A+B;
```

图16-20给出了该程序片段对应的流图。

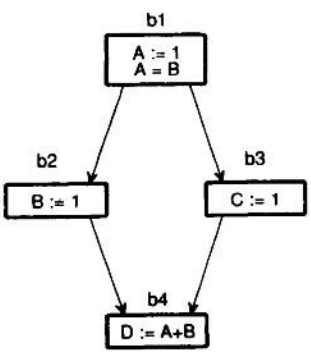


图16-20 用于活跃变量分析的流图

646

从基本块中，我们首先抽取Def和LiveUse集合：

Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	∅
b3	{C}	∅
b4	{D}	{A,B}

我们可以从最后一个基本块开始以后向方式来求解。事实上，在这个分析过程中，信息流是从变量的使用回溯到它的定义点，因此，活跃变量的检测有时也被称为是一个后向流（backward-flow）问题。我们将会看到那些信息流和控制流方向一致的数据流问题，它们是前向流（forward-flow）问题。

由于b4没有后继，因此LiveOut(b4) = ∅。因此有：

LiveIn(b4) = LiveUse(b4) = {A,B}

现在，

LiveOut(b2) = LiveIn(b4) = {A,B} 以及
LiveOut(b3) = LiveIn(b4) = {A,B}

在b2和b3中没有活跃变量的使用，因此我们有：

LiveIn(b2) = LiveOut(b2)–Def(b2) = {A,B}–{B} = {A}
LiveIn(b3) = LiveOut(b3)–Def(b3) = {A,B}–{C} = {A,B}
LiveOut(b1) = LiveIn(b2) ∪ LiveIn(b3) = {A} ∪ {A,B} = {A,B}

最后，

LiveIn(b1) = LiveUse(b1) ∪ (LiveOut(b1)–Def(b1)) = {B} ∪ ({A,B}–{A}) = {B}

我们总结得到：

Block	LiveIn	LiveOut
b1	{B}	{A,B}
b2	{A}	{A,B}
b3	{A,B}	{A,B}
b4	{A,B}	∅

该例子也说明了活跃变量分析的另一种用法。如果变量在初始基本块的块首活跃，那么该变量可在被定义前被使用——而这种定义前使用是一种常见的错误。在我们的例子中，LiveIn(b1) = {B}，变量B事实上在被定义前已被使用。

647

如果我们将注意力集中到那些有惟一开始结点（即没有前驱的结点）和多个结束结点（即没有后继的结点）的流图上，我们总能求解出那些数据流方程。其想法是，由基本块所生成的值开始（如此例中用到的LiveUse集合），在去除那些在基本块内被注销的值（如此例中用到的Def）后，将其余的值传播到前驱结点中。此过程将一直迭代到In和Out集合收敛为止。

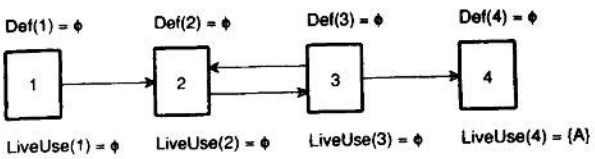


图16-21 简单循环的流图

令人惊讶的是，数据流问题的解不必惟一。要明白为什么会这样，可以考虑图16-21中与一个简单循环相对应的流图。在这个流图中没有定义任何变量，而基本块b4中却使用了变量A。最明显的解是将这个引用往后传播，那么对于所有四个基本块来说，LiveIn = {A}。不幸的是，还可能有一些毫无道理的

解。例如，下面的解是一个有效解（即满足所有的数据流方程）：

Block	LiveIn	LiveOut
b1	{A,B}	{A,B}
b2	{A,B}	{A,B}
b3	{A,B}	{A,B}
b4	{A}	∅

该解最显著的一点是B没有在任何基本块中使用过！该问题在于基本块b2和b3互为父结点，且因为B从未被定义过（所有的Def集合均为空），所以一旦把B包含到LiveIn集合里，则它将永远不会被去除。

对于数据流方程，我们既可以消极地也可以乐观地看待它们。消极的看法是假定所有变量都是活跃的，除非能在所有后继基本块中看到显式的定义。乐观的看法是仅当我们在某个后继基本块中看到活跃的引用（中间没有插入新的定义）时，假定一个变量是活跃的。

从选择含有最小可能的LiveIn和LiveOut集合的（有效解）这个意义上说，乐观的看法将是“最小的”。在练习27里，我们将证明总存在最小的解。出于优化目的，我们希望获得最小活跃变量解，因为活跃变量需要保存而非活跃变量则可忽略。换句话说，仅当在后继块中能实际看到一个变量的活跃引用
648 时，该变量才被认为是活跃的。

我们可以将寻找可能未初始化的变量的问题形式化为一个前向数据流问题。前向数据流问题以和程序执行期间的控制流一致的方向追踪信息流。

我们认为未初始化变量是那些含有非法值的变量。某些编译器——例如PL/C（Conway 1973）——将所有变量初始化为一个特殊的非法值且在使用变量前测试它是否为该值。在普通的Pascal和Ada语言编译器中，约束变量和访问型变量在使用前通常要进行有效性的检查。

我们的分析可以确定在基本块开始或结束处哪些变量可能是未初始化的。未初始化变量在使用前被测试。那些不在未初始化集合里的变量一定有合法的值且不需要在使用前进行检查。设UninitIn(b)是那些在基本块b的入口处未初始化的变量集合。类似地，设UninitOut(b)是那些离开基本块b时未初始化的变量集合。如果一个变量在离开集合P(b)中的基本块时是未初始化的，其中P(b)是流图中b的直接前驱，则该变量在基本块b入口处未初始化。即：

$$\text{UninitIn}(b) = \bigcup_{i \in P(b)} \text{UninitOut}(i)$$

如果基本块没有前驱，那么它的UninitIn集合将包含所有变量。设Init(b)是b中那些在b结尾处已经初始化的变量集合。该集合既包括那些被赋予已知有效值的变量，也包含那些在使用前被测试的变量。在后一种情况下，一个非法值可能已经导致错误的出现，然而，如果控制能到达基本块末尾，则这样的变量值就一定是有效的。由Init(b)的定义，我们得知，$\text{UninitOut}(b) \supseteq \text{UninitIn}(b) - \text{Init}(b)$。

设Uninit(b)是那些在b中成为未初始化的且随后也未在该基本块中被重新定值或测试的变量集合。一个变量成为未初始化的，可能是因为某个操作的副作用（如释放一个堆对象）而被赋予了非法值（如**null**），或者是因为该变量刚刚被创建（如程序块中局部声明的变量）。可以很容易地知道，$\text{UninitOut}(b) \supseteq \text{Uninit}(b)$。

UninitOut(b)的计算从UninitIn(b)开始，加入那些成为未初始化的变量并去除掉那些已知为初始化的变量：

$$\text{UninitOut}(b) = \text{Uninit}(b) \cup (\text{UninitIn}(b) - \text{Init}(b))$$

因为UninitOut是从UninitIn计算得来的，所以这是一个前向流问题；其信息流和控制流的方向一致。同样地，它的解不需要惟一。因此，我们采取保守的方法并假定在第一个基本块入口处所有的变量都是未初始化的。这就确保了在一个基本块入口处，某变量仍将被看成是未初始化的，除非已知此变量在到该

基本块的所有路径上均被初始化。 649

16.4.2 全路径流分析

我们刚才讨论的数据流问题均假定，一个特征如果可被证明在某个路径上成立的话，那么该特征将为真。这样，如果存在某些导致变量活跃引用的路径，则该变量将被看成是活跃的；同样，如果存在任意一条缺少适当初始化（或测试）的路径，则某个变量将被认为是未初始化的。这类数据流问题被称为*单路径*（any-path）问题。单路径问题的解不能保证所期待的特征一定成立，而仅仅是可能成立。

数据流问题也能通过那些声明所期待的特征必须在所有可能路径上成立的*全路径*（all-path）方式来求解。在全路径问题的解中，所期待的特征可确保总是成立的。

（前向流的）全路径问题的一个极佳的例子是表达式可用性的确定。如果一个表达式已被计算，而且再次计算它将是冗余的，则该表达式被称作是*可用的*（available）。表达式可用性的信息在施行全局公共子表达式优化时是至关重要的。

我们首先关注如何确定在基本块中计算的表达式在离开基本块时是否可用。起初，它看起来可以使用那个曾用于局部公共子表达式优化的`last_def`值（参见15.4.2节）。但不幸的是，它不堪此用。这个问题在于，`last_def`考虑的仅仅是表达式的直接操作数，而忽略了子操作数的赋值。为解决这个难点，对于给定的表达式，我们计算确定该表达式值所需的所有变量。这一点可以通过定义与临时变量T关联的表达式计算的*相关变量*（relevant variable）集合来做到。该集合我们称之为RelVar(T)，并按图16-22所示来计算它。

```
void compute_rel_vars(temporary T)
{
    /*
     * Compute relevant variables for expression
     * corresponding to temporary T.
     */
    RelVar(T) = SET_OF( T );
    while (there exists a temporary T' ∈ RelVar(T))
        Replace T' in RelVar(T) with the variables
        and temporaries used to compute T';
}
```

图16-22 计算相关变量的算法

`compute_rel_vars()`递归地将集合中的临时变量替换为计算它们所需的其他变量和临时变量，直到集合中只剩下变量为止。

T在出口处可用，当且仅当：

$$\forall X \in \text{RelVar(T)} : \text{last_def(X)} < \text{last_def(T)}$$ 650

定义AvailOut(b)为在基本块b出口处可用的临时变量集合。回想一下，临时变量名将惟一地与表达式关联，以便容易地确定那些潜在的冗余表达式。定义AvailIn(b)为基本块b入口处可用的临时变量集合。因为这是一个前向流问题，所以我们知道AvailIn(b)将依赖于b的前驱的AvailOut值。我们仍用P(b)作为流图中b的前驱集合。现在，我们要求临时变量应该在所有的路径上先行计算，且在所有的AvailOut集合中：

$$\text{AvailIn(b)} = \bigcap_{i \in P(b)} \text{AvailOut(i)}$$

我们很自然地假设在第一个基本块入口处没有表达式可用。只有在一、两种情况下表达式在基本块出口处可用。其一，它在基本块中计算且在最后一次计算后没有被注销。这种情况可以通过检查相关变量来确定。设Computed(b)为那些在基本块中已被计算且随后未被注销的表达式集合。

此外，一个表达式在基本块入口处可用且未在基本块中被注销。即，该表达式的相关变量没有一个在基本块中被赋值。设Kill(b)为基本块b中由于给相关变量赋值而被注销的表达式集合。因此，定义在出口处可用的表达式集合的方程是：

$$AvailOut(b) = Computed(b) \cup (AvailIn(b) - Killed(b))$$

同样存在全路径的后向数据流问题。这种情况的一个较好的例子是非常忙表达式（very busy expression）的确定。如果一个表达式在被注销前其值在所有路径上均被使用，则该表达式是非常忙的。非常忙表达式是寄存器分配的主要候选，因为我们知道它们的值一定会被使用。“非常忙”的分析也可用来指导代码的迁移。也就是说，如果在一个循环里，循环不变式是“非常忙”的话，那么我们可能愿意将此表达式提到循环的首部（假定循环至少执行一次迭代）。

设VeryBusyOut(b)是那些在基本块b结尾处非常忙的表达式集合，且设VeryBusyIn(b)是那些在基本块b的开始处非常忙的表达式集合。于是有：

$$VeryBusyOut(b) = \bigcap_{i \in S(b)} VeryBusyIn(i)$$

我们假定在最后一个基本块出口处没有非常忙的表达式。

设Used(b)是基本块b中那些被注销前使用的所有表达式的集合，而Killed(b)是基本块b中那些使用前被注销的表达式集合。于是有：

651

$$VeryBusyIn(b) = Used(b) \cup (VeryBusyOut(b) - Killed(b))$$

16.4.3 数据流问题的分类

如我们所见，数据流问题存在着很清晰的分类。每个基本块有相应的In和Out集合。对于前向流问题而言，Out集合是由基本块中的In集合计算得到，而In集合则由前驱基本块的Out集合来计算。类似地，对后向流问题而言，In集合是由基本块中的Out集合计算得到，而Out集合则由后继基本块的In集合来计算。

在基本块内，根据问题是后向流问题还是前向流问题，可用下面的方程来表示In和Out集合之间的关系：

$$In(b) = Used(b) \cup (Out(b) - Killed(b)) \quad 或$$
$$Out(b) = Used(b) \cup (In(b) - Killed(b))$$

在单路径问题中，将计算前驱或后继值的并集；而在全路径问题中，将计算前驱或后继值的交集。

最后，作为边界条件，必须指定前向流问题中初始基本块的In集合的值与后向流问题中结尾基本块的Out集合的值。通常，根据求解的问题，这些边界集合不是被设置为空集就是包含所有可能的值。

我们将数据流分析的一般模式概括在图16-23中的表格里。

	前向流	后向流
单路径	$Out(b) = Gen(b) \cup (In(b) - Killed(b))$ $In(b) = \bigcup_{i \in P(b)} Out(i)$	$In(b) = Gen(b) \cup (Out(b) - Killed(b))$ $Out(b) = \bigcup_{i \in S(b)} In(i)$
全路径	$Out(b) = Gen(b) \cup (In(b) - Killed(b))$ $In(b) = \bigcap_{i \in P(b)} Out(i)$	$In(b) = Gen(b) \cup (Out(b) - Killed(b))$ $Out(b) = \bigcap_{i \in S(b)} In(i)$

图16-23 用于数据流分析的方程

16.4.4 其他重要的数据流问题

在本节中，我们会简要地考虑一些其他数据流问题。一个单路径、前向流的分析可用来计算到达定值（reaching definition）集合。定值也就是给基本块中的变量赋值。一个变量v的定值到达v的一个引用，

前提条件是存在一条从这个v的定值到v的引用的路径，且此路径上没有对v重新定值。直观地讲，如果v的定值到达v的引用，那么那个定值可能已建立了我们即将使用的值。到达定值对于寄存器目标分配很有用。特别地，为最小化寄存器拷贝，我们很希望把某个变量能到达相同引用的所有定值都分配到相同的寄存器中。否则，该变量的当前值就可能会存放在多个寄存器中（而我们将可能不得不从它的内存单元中装入其值）。

到达定值在常量传播中也很有用。如果到达特定引用的惟一的变量定值涉及一个常量的赋值，那么该引用可被替换为那个常量值。 652

作为示例，考虑如下的代码片段：

```
A := 1;
B := C;
if A=B then
      B := 1;
else  C := 1;
end if;
A := A+B;
```

在最后一条语句中，A和B均被使用。和A的惟一定值一样，B的两次定值均可到达这条语句。因为A的定值涉及常量赋值，所以我们可以将A的引用替换为值1。类似地，如果两次B的定值把它们的值放在相同的寄存器中，那么可直接在那个寄存器上进行加1操作（假定B在寄存器中的值首先要保存，如果B在这条语句后是活跃的）。

作为数据流问题分析的标准，我们必须形式化In、Out、Gen和Killed集合的定义。In和Out集合代表可到达基本块块首与块尾的定值集合。这些集合包含那些定义变量的元组（或树）的索引（或地址）。初始基本块的In集合为空。基本块的Gen集合包含那些在基本块中出现且能到达块尾的定值。如果基本块中包含相同变量的多个定值，那么正常情况下，只有最后一个定值可以到达块尾。

对于每个在基本块中定义的变量v，Killed集合包含了v的其他所有不在Gen集合中的定值。Killed集合“擦除”了那些被基本块中的局部定值所淘汰的定值。

在像Pascal和Ada那样的语言里，由于子程序调用和别名的影响，到达定值的确定将非常复杂。特别是我们在进行过程调用或给别名对象（数组或堆对象）赋值时，一个变量可能被定义也可能不被定义。由于不清楚它们是否会实际被赋值，故而我们称这些定值为*二义性的*（ambiguous）。而那些在基本块中对变量的显式定值则是*无二义的*（unambiguous）。基本块中的二义性定值包含在Gen集合中，但它们不会注销其他的定值，因此对Killed集合没有影响。实际上，它们添加可能的新定值但是绝不会像无二义定值那样删除定值。

到达定值有时被编码在称为ud-链（use-definition chain）的数据结构里。尽管它的名字如此，但ud-链却是和变量每次使用相关联的到达定值集合。我们在优化前收集此信息并把它用在优化和代码生成阶段。 653

和ud-链同属一类的还有du-链（definition-use chain）。du-链是和每个变量定值相关联的变量使用情况的集合。也就是说，du-链可使我们找到在基本块中的特定点可能使用该值给某个变量赋值的所有元组（或树）。du-链可被用于多种用途，包括寄存器目标分配。

du-链可采用单路径、后向流分析计算得到，这类似于计算活跃变量的方法。In和Out集合代表着那些可能使用变量当前值的元组（或树）。结尾基本块的Out集合为空集。Gen(b)是那些在基本块b中在定义变量前使用那个变量的元组集合。Killed(b)是那些可以使用定义在基本块b中的变量的元组集合。

流分析还可用来确定*复写传播*（copy propagation）的可能性。有时，针对A:=B形式的拷贝语句，我们可以通过将A的引用替换为B而删除该语句。（我们在15.4.3节的寄存器指派算法中这样做过。）如果在使用A的某个地方，我们能确定A := B是公共子表达式（即，是冗余的），那么对A的引用可替换为B。如果对A的所有引用都被替换为B，那么A就成了非活跃的且那个赋值语句可被删除。

对A的引用可被替换为B，条件1是A := B是到达A的引用的惟一的定值（ud-链可用来做此项检查），而条件2是在该赋值语句执行后没有出现对B的重新赋值。后一个条件可以通过一个全路径、前向流分析来检查。设In(b)是那些已执行完且随后其左边或右边变量都没有被再赋值的拷贝语句的集合。在In(b)中的拷贝语句是复写传播的候选。可类似定义Out(b)。初始基本块的In集合为空。

Gen(b)是基本块b中的拷贝语句的集合，这些拷贝语句的左边或右边变量其后在基本块中未被重新定值。Killed(b)是那些不在基本块b中的拷贝语句的集合，同时这些语句的左边或右边的变量却在基本块b中赋值。

作为示例，考虑：

```
A := D;
if A=B then
      B := 1;
else
      C := 1;
end if;
A := A+B;
```

拷贝语句A := D可到达比较操作A=B和加法操作A+B。因为A或D均没有在使用前被再赋值，所以复写传播是可能的。此外，通过参考A:=D的du-链，得知A只在两个地方使用，而这两个地方我们都准备将其
654 替换为D。这项修改把A的引用次数减少到零，使得该赋值语句成为多余的语句。我们因此得到：

```
if D=B then
      B := 1;
else  C := 1;
end if;
A := D+B;
```

16.4.5 使用数据流信息的全局优化

在这一节里，我们会简要地考虑如何实际使用那些通过数据流分析收集的信息来实现各种全局优化。我们不打算讨论所有可能施行的优化——那太多了！相反，我们将说明在控制代码迁移或删除时如何使用由数据流分析提供的数据。

在图16-24所示的表格里，我们对已研究过的数据流分析按照它们的流方向、路径形式（全路径或单路径）以及初始条件进行了分类。前向流问题中的初始条件定义第一个基本块的In集合；而后向流问题的初始条件则定义了最后一个基本块的Out集合。正常情况下，这些集合的初值不是空集（∅）就是全集（即包含所有可能的值）。

使用这些信息，每一种数据流分析可以按需施行。接下来的问题是何时施行这些分析以及如何使用收集到的信息。在下面的各小节里，针对我们已研究的每一个数据流问题，我们将逐一关注这些问题。

	前向流		后向流	
	问题	初始值	问题	初始值
单路径	到达定值（ud-链） 未初始化的变量	∅ 所有变量	活跃变量 du-链	∅ ∅
全路径	可用表达式 复写传播	∅ ∅	非常忙 表达式	∅

图16-24 全局优化和相应的数据流分析

非常忙表达式

如16.3.1节所示，非常忙的循环不变式可作为从循环体中进行代码迁移的最佳候选。循环不变式可以被安全地从循环体中移出。由于非常忙循环不变式总要在循环中使用，因此将它们的计算放置在循环首部

也是有利的。可按图16-25所示修改16.3.1节中的factor_invariants()例程。现在，虽然该例程满足了寻找非常忙表达式的要求，但它却更加复杂了。尽管如此，现在那些被外提的不变式都确实能保证其迁移是有利可图的。 655

有时在多个基本块中计算的代码可以被移到那些基本块的共同祖先（基本块）中。这种优化称为*代码提升*（code hoisting）；它节省了空间，方法是一个表达式只在祖先基本块中计算一次而不是在不同的后继基本块中计算多次。例如，在下面程序片段中，

```
if I = 1 then
    A(I) := 0;
else
    A(I) := 1/(I-1);
end if;
```

A(I)的计算可被提升到**if**语句的首部。

```
void factor_very_busy_invariants(loop L)
{
    /*
     * Find very busy expressions invariant in loop L
     * and move their calculation outside the loop body.
     */

    mark_invariants(L);
    Compute VeryBusyInvariants, the set of marked
     expressions that are very busy at L's header.

    for (each expression E in VeryBusyInvariants) {
        Allocate a new temporary T;
        Replace each occurrence of E in L with T;
        Insert T := E in L's header code immediately
          after the first loop termination test;
        /*
         * If L must iterate at least once
         * T := E may be placed immediately before L.
         */
    }
}
```

图16-25 外提非常忙循环不变式的算法

只有非常忙表达式才应该被提升；否则，表达式的计算可能是不必要的。但我们如何定位那些可作为代码提升的目标基本块呢？一般来说，我们称基本块b*支配*（dominate）基本块集合S，前提条件是到S中基本块的所有路径都必须经过b。因为b是S中基本块必然的前驱，所以它是存放可能从S中提升出的代码的较好的目标选择。已知有计算任意流图中支配（必经）结点的算法（参见练习24），但是像Pascal和Ada那样结构化的语言提供了非常适合做代码提升的条件执行结构（**if**和**case**语句）。特别地，在**if**和**case**语句的首部将支配其语句体中的所有语句。图16-26中的code_hoist()例程利用条件语句的结构来识别和提升表达式。因为**if**和**case**语句可以嵌套，所以应当按由内到外的方式使用code_hoist()，以便代码从内层嵌套结构提升到它的外层包围结构里。 656

全局公共子表达式删除

在15.4.2节里，我们研究了局部公共子表达式（CSE）的删除问题。特别是在我们看到基本块b中不止一次计算表达式E时，那些冗余的计算将被识别并删除。在只施行局部优化时，表达式在基本块中的第一次计算决不会冗余。然而，在施行全局优化时，如果基本块中表达式已被该基本块的前驱基本块计算过，则这个表达式的首次计算可能会冗余。图16-27中的算法remove_global_cses()将识别并删除基本块中冗余的表达式首次计算。我们假定基本块中后面的冗余计算已被局部CSE优化所删除。每个识别为全局CSE的表达式被分配一个保存其值并可穿越基本块的临时变量（多数情况下这将是一个存储型临时变量）。

```
void code_hoist(conditional_statement C)
{
    /*
     * Find very busy expressions in C
     * and hoist them to the header of C.
     */
    Compute VeryBusy, the set of expressions in C's body
     that are very busy at C's header.
    for (each expression E in VeryBusy) {
        Allocate a new temporary T;
        Replace each occurrence of E in C with T;
        Insert T := E immediately before C;
    }
}
```

图16-26 提升条件语句中非常忙表达式的算法

```
void remove_global_cses(void)
{
   /*
    * Find redundant initial calculations
    * of CSEs in basic blocks and remove them.
    */

   Compute GlobalCSEs, the set of expressions that
         are computed in more than one basic block;
     /*
      * Recall each expression is hashed to a unique
      * result temp, so potential CSEs are easy to
      * identify.
      */

   for (each expression E in GlobalCSEs) {
       Do an available expression data flow
          analysis for E;
       Assign a temporary location, temp_loc(E), to E;
   }

   for (each basic block B) {
       for (each expression E in GlobalCSEs) {
           if (E is computed in B
                 && E is available on entrance to B) {
               Remove the first calculation of E in B
                  and replace it with a reference
                  to temp_loc(E);
           }
       }
   }
}
```

图16-27 删除CSE冗余初始计算的程序

活跃变量分析

活跃变量的值必须在基本块结尾加以保存；而非活跃变量的值则无需保存。类似地，如果施行全局CSE删除，那么在基本块之间的CSE的值有时必须要加以保存。特别地，如果将保存CSE值的位置看作
657 变量，那么在这些位置活跃时必须要保存CSE的值。

定义在图16-28中的remove_dead_stores()例程寻找那些存储非活跃变量的指令并删除它们。

未初始化变量分析

在诊断编译器中，识别潜在的未初始化变量非常有用。一旦识别出未初始化变量，编译器将发出警告信息，或生成运行时检查以发现对未初始化变量的非法引用。定义在图16-29中的find_uninitialized_vars()例程寻找未初始化变量的可能使用并发出编译器警告或生成代码来检测

对未初始化变量的非法引用。 658

```
void remove_dead_stores(void)
{
    /*
     * Find unnecessary stores of dead variables
     * and global CSEs in basic blocks and remove them.
     */

    for (each basic block B) {
        for (each V that is a variable or
              global CSE location) {
            if (a store into V follows all references
                 to V in B) {
                Perform a live variable analysis for V;
                if (V is not live on exit from B) {
                    Remove the store of V from B;
                }
            }
        }
    }
}
```

图16-28 删除非活跃变量存储的算法

```
void find_uninitialized_vars(void)
{
   /* Find possible uses of uninitialized variables. */

   Perform an uninitialized variable data flow analysis
   for (each basic block B) {
      for (each use of a variable V in B) {
         if ( (this is the first use of V in B
               && V is uninitialized on entrance to B)
            || instructions since the last use of V may
               have made V uninitialized) {
            Issue a warning that V may be uninitialized
              or generate code to check that V is
              properly initialized
         }
      }
   }
}
```

图16-29 寻找未初始化变量的可能使用的算法 659

常量传播和复写拷贝

通常，对编译器而言，若能知道变量在程序中的某一点含有特定的值将非常有用。这将允许用这个已知值来替换该变量，其效果就像是把该变量当作有名常量一样。这种优化称为*常量传播*（constant propagation），它在像FORTRAN这样没有有名常量的语言中特别有用。然而，即使在那些包含有名常量的语言里，常量传播仍可以改进代码质量，方法是给变量赋值为常量后就立即引用这个变量（例如，A := 100; B := B + A;）。

如16.3.2节所述，复写传播利用了在某些情况下存在的事实，即，在形式为X := Y的赋值语句后，对X的引用可能被替换为对Y的引用。这种修改可能允许删除该赋值，甚至允许删除整个X。很明显，常量传播是复写传播的特例。

图16-30中的算法propagate()识别可施行常量传播和复写传播的情况。如果可能，我们应当简化表达式以便发现新的优化机会。

```
void propagate(void)
{
  /*
   * Propagate constant and variable assignments and
   * simplify resulting expressions.
   */

   Perform a reaching definition data flow analysis;
   Perform a du-chain data flow analysis;
   Perform a copy propagation data flow analysis;

   Mark all uses of variables in the program;

   for (each marked use of a variable V) {
      Unmark this use of V;
      if (the only definition of V that reaches this use
          of V is V := C, where C is a constant) {
          Replace this use of V with C and
            try to simplify the resulting expression;
          if (this substitution and simplification
             creates a constant assignment X := K) {
             Replace the original assignment with X := K;
             Mark all uses of X that this
              assignment reaches;
          }
          Remove this use of V from the du-chain of V := C;
      } else if (the copy propagation analysis shows that
              the only definition of V that reaches this
              use of V is V := X, where X is a variable) {
          Replace this use of V with X;
          Remove this use of V from the du-chain of V := X;
      }
   }

   for (each definition of a variable V)
      if (all uses of this definition have been removed by
            constant or copy propagation)
          Remove this definition from the program;

   for (each variable V)
      if (all uses of this variable have been removed)
          Remove this variable from the program;
}
```

图16-30 施行常量传播和复写传播的算法

16.4.6 求解数据流方程

我们现在讨论前几节中所提出的数据流方程的求解问题。人们已经研究出了多种求解方法。而我们将详细考察其中的两种。第一种是迭代法，该方法计算近似解直到收敛为止。第二种方法则利用了现代程序设计语言的流图是结构化的事实，因为这样的流图就是从语言中结构化的部件（如条件和循环语句）推导而来的。从基本块的方程开始，第二种方法递归地构造语言结构化部件的方程直到整个程序或子程序的单个方程构造完毕为止。然后它使用初始条件来求解。

迭代法

我们的数据流问题是用集合来形式化的，因此，首要的问题是为数据流方程中出现的集合及其操作（并、交和差）寻找一种紧凑而有效的实现方法。

位向量是表示集合的一种好方法。向量中包含与集合中每个可能的对象相对应的位。位为1表示存在相应对象，位为0则表示相应对象不存在。事实上，这种表示法在Pascal中常用来表示集合。

集合操作完美地对应于位向量上的布尔运算。集合的并与交操作分别对应着位模式的“或运算”（C中的 | 操作符）和“与运算”（C中的 & 操作符）。集合的差操作A-B对应(Va & (~Vb))，其中，Va和

660

Vb分别是表示A和B的位向量。

连接相同基本块的In和Out集合的基本数据流方程，根据所分析的问题方向不同（后向还是前向），通常总具有如下形式：

$In(b) = Gen(b) \cup (Out(b) - Killed(b))$ 或 $Out(b) = Gen(b) \cup (In(b) - Killed(b))$

因为Gen(b)和Killed(b)是由基本块b中的语句所决定的常量，因此我们重写这些方程如下：

$In(b) = F_b(Out(b))$ 或 $Out(b) = F_b(In(b))$

其中，F_b是与基本块b对应的函数，它把Out集合映射为In集合（或把In集合映射为Out集合）。

对每个In(b)和Out(b)集合，设$In_i(b)$和$Out_i(b)$是期望解的第i次近似。我们的算法将一直迭代到收敛为止——即，我们到达第i次迭代，其中对每个基本块b均有$In_i(b) = In(b)$且$Out_i(b) = Out(b)$。

这里我们仅关注前向流问题，至于后向流可以对称地考虑。我们首先定义In_0集合的值。对于第一个基本块（我们称之为基本块0），该集合将事先给出。在前向流问题中，对所有的i，$In_i(0) = In(0)$（即，我们每次开始于In(0)的正确值）。

对单路径问题而言，其他的In_0集合被初始化为Ø。其理由是，在单路径问题里，我们使用集合的并运算来合并前驱的值，而Ø对集合的并运算是恒等值（对所有的S，$S \cup Ø = S$）。通过以Ø值开始，我们确保了迭代中不会出现不期望（或不想要的）的值。

对全路径而言，In_0集合（不同于$In_0(0)$）将被初始化为 $\cup$，即包含所有可能值的全集。针对全路径问题，我们使用集合的交运算来合并前驱的值，而 $\cup$时集合的交运算是恒等值（对所有S，$S \cap \cup = S$）。通过从 $\cup$ 值开始，我们确保了迭代中不会丢失任何正确的值。在定义In_0的值以后，我们只是简单地进行迭代直至收敛为止，迭代算法如图16-31所示。

```
Initialize all In0 sets;
i = 0;

do {
    for (each basic block b)
        Out_i(b) = F_b(In_i(b))

    for (each basic block b) {
        if (this is an any-path problem)
            In_{i+1}(b) = ∪_{j∈P(b)} Out_i(j)
        else   /* Must be an all paths problem. */
            In_{i+1}(b) = ∩_{j∈P(b)} Out_i(j)
    }
    i += 1;
} while (there is at least one block b
          for which  In_i(b) ≠ In_{i-1}(b));
```

图16-31 迭代计算In集合的算法

作为示例，考虑图16-32中的程序片段，该程序读入一系列数字并求和。相应的程序流图如图16-33所示。

```
Read(Limit);
for I in 1..Limit loop
  Read(J);
  if I=1 then
      Sum := J;
  else
      Sum := Sum + J;
  end if;
end loop;
Write(Sum);
```

图16-32 读数并求和的程序

我们将进行未初始化变量的分析。In和Out集合表示那些可能未初始化的变量。Killed集合表示那些通过给它们值而被注销的未初始化变量。Gen集合是那些成为未初始化的变量集合。在这个例子中，这种情况只出现在**for loop**结束后，此时循环

索引变成未初始化的（在Ada语言里，此时索引也是不可访问的）。对每个基本块，我们有：

661
~
662

	b0	b1	b2	b3	b4	b5	b6
Gen	∅	∅	∅	∅	∅	∅	{I}
Killed	{Limit,I}	∅	{J}	{Sum}	{Sum}	{I}	∅

在初次迭代时，In_0(b0) = {Limit,I,J,Sum}，其他的In集合为空集（这是一个单路径问题）。接着，我们使用规则Out=(In-Killed) ∪ Gen，由In集合计算Out集合。此时有：

663

	b0	b1	b2	b3	b4	b5	b6
In_0	{Limit,I,J,Sum}	∅	∅	∅	∅	∅	∅
Out_0	{J,Sum}	∅	∅	∅	∅	∅	{I}

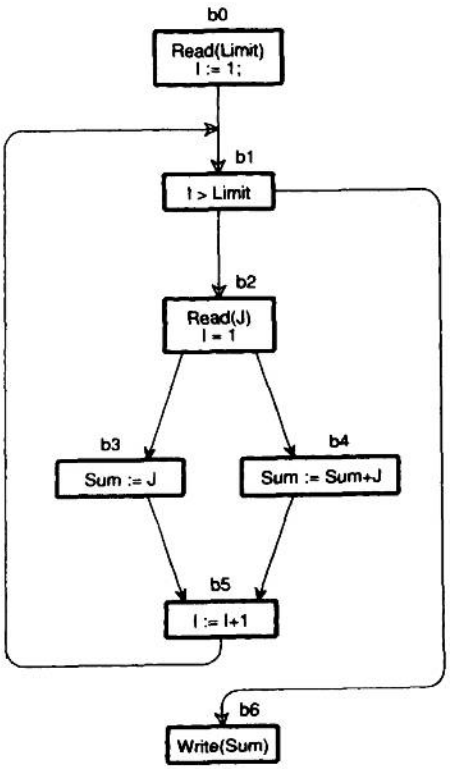


图16-33 图16-32中程序的流图

通过合并前驱基本块的Out_0值，我们计算得到In值的下次近似值In_1。然后再从In_1计算Out_1：

664

	b0	b1	b2	b3	b4	b5	b6
In_1	{Limit,I,J,Sum}	{J,Sum}	∅	∅	∅	∅	∅
Out_1	{J,Sum}	{J,Sum}	∅	∅	∅	∅	{I}

我们继续迭代：

	b0	b1	b2	b3	b4	b5	b6
In_2	{Limit,I,J,Sum}	{J,Sum}	{J,Sum}	∅	∅	∅	{J,Sum}
Out_2	{J,Sum}	{J,Sum}	{Sum}	∅	∅	∅	{J,Sum,I}

	b0	b1	b2	b3	b4	b5	b6
In_3	{Limit,I,J,Sum}	{J,Sum}	{J,Sum}	{Sum}	{Sum}	∅	{J,Sum}
Out_3	{J,Sum}	{J,Sum}	{Sum}	∅	∅	∅	{J,Sum,I}

	b0	b1	b2	b3	b4	b5	b6
In_4	{Limit,I,J,Sum}	{J,Sum}	{J,Sum}	{Sum}	{Sum}	∅	{J,Sum}
Out_4	{J,Sum}	{J,Sum}	{Sum}	∅	∅	∅	{J,Sum,I}

在第4次迭代后，所有集合均收敛。

这种流分析的值现在变得明显起来——我们在看似正确的程序中发现了一个错误。正常情况下，我们希望所有变量使用前有定义。然而，Sum ∈ In(b6)且b6中要使用Sum。这个程序可能是在循环至少会迭代一次的假设下编写的，因为那样的话，就可以保证Sum在输出前（使用前）已被定义。然而，在Pascal和Ada语言里，循环可以迭代零次，而这种情况下，正如我们分析所发现的，可能导致企图输出一个未定义的Sum值。

我们的分析也警告我们：Sum在b4入口处是潜在未初始化的。然而，这是一个“误报”，它是由于我们的分析没有办法识别b4将只能在（前一次迭代中的）b3执行后才可进入，而这一点却保证了Sum被正确初始化。

不难看见，如果我们的迭代算法终止运行，它将产生正确的解。此结论基于这样的事实：在算法终止时，所有的数据流方程均被满足。此时尚不清楚的是算法是否一定总会终止。即使它会终止，其收敛速度是否慢得难以接受呢？

所幸的是，我们并未发现这些令人担心的情况。我们的迭代算法总是在合理的时间内产生解。首先考虑算法的终止。我们知道计算Out值的函数F_b的形式为：$F_b=(In(b)-Killed(b)) \cup Gen(b)$。因为Killed(b)和Gen(b)均为常量，所以易知函数F_b是单调的（monotonic）。即，如果$s_1 \subseteq s_2$，那么$F_b(s_1) \subseteq F_b(s_2)$。

对于单路径问题，$In_{i+1}(b) = \bigcup_{j \in P(b)} Out_i(j) = \bigcup_{j \in P(b)} F_j(In_i(j))$。我们知道其中的函数F是单调的，而且集 665
合的并运算也是单调的。设In_i是所有基本块b的$In_i(b)$值的向量。我们说$In_i \subseteq In_j$，条件对所有基本块b，满足$In_i(b) \subseteq In_j(b)$。这意味着：一个集合向量包含在另一个集合向量中，如果它的每一个集合成员是另一个集合向量中对应集合成员的子集。

现在可以重写前面的方程为$In_{i+1}(b) = G_b(In_i)$，其中$G_b = \bigcup_{j \in P(b)} F_j(In_i(j))$。函数F和集合的并运算都是单调的，因此函数G也是单调的。这意味着：如果$In_i \subseteq In_{i+1}$，那么对任意的基本块b，$G_b(In_i) \subseteq G_b(In_{i+1})$。这是问题的关键所在，而其他的都好解决。

流分析的初始化规则告诉我们：$In_0(b0)$是事先提供的，而其他基本块的$In_0(b)=\varnothing$。因为$In_0(b0) = In(b0)$，易知$In_0 \subseteq In_1$。这意味着：在第一次迭代以后我们可能已向In集合中添加了值，但我们确实没有删除任何值。现在，我们利用函数G_b的单调特征。对任意基本块b，$In_1(b) = G_b(In_0)$，$In_2(b) = G_b(In_1)$。因为$In_0 \subseteq In_1$，我们得出$In_1(b) \subseteq In_2(b)$。此关系对所有的基本块均成立；因此，$In_1 \subseteq In_2$。在第二次迭代后，我们可能又在相关集合中添加了某些值而同时又没有删除任何值。按此归纳，我们得到$In_0 \subseteq In_1 \subseteq In_2 \cdots$。

每次迭代时我们添加某些值但从不删除它们。然而，In集合大小是有限的，我们不能无限制地添加值。最终我们到达某一点，那里$In_i = In_{i+1}$，且在这一点算法终止并获得解。事实上，在练习27中，我们将证明此时获得的解是最小解（minimal solution）。这意味着：如果存在两个不同的解（如，In是我们算法求得的解，而In'是其他算法求得的一个有效解），那么有$In \subseteq In'$。对数据流分析而言，可能更希望获得最小解，因为它仅包含着可从初始的In集合以及从各基本块的Gen和Killed集合推导出的数据。

对全路径问题而言，我们的算法再一次展示了其单调特征。我们从预先提供的$In_0(b0) = In(b0)$以及（针对其他块的$In_0(b) = \cup$开始。现在，我们在迭代时单调地删除集合中的值，而且以后也决不会再恢复它们。这意味着：我们到In的近似解序列将满足$In_0 \supseteq In_1 \supseteq In_2 \cdots$的关系。再者，因为In集合为有限集，所以我们的算法最终一定会终止并求得最大解（maximal solution）。最大解包含了任何其他的解。对全路径问题来说，可能希望获得最大解，因为我们开始于所有可能值并删除了那些不在所有路径上出现的值。我们希望保留那些与所有路径一致的值——即，那些满足最大解的值。

现在，我们估测数据流分析要花多长时间来分析计算一个带有B个基本块以及包含V个值的In和Out集合的程序或子程序。在每次迭代中，对每个基本块，我们从In集合来计算Out集合（或反之）。每个基本块维持一个长度为V的位向量，其时间复杂度为O(V)。而对所有B个基本块，其时间为$O(B \times V)$。我们还必须传播In或Out集合到前驱或后继。此项传播时间取决于前驱或后继的数目。该数目可变，但平均而言，它是一个不大的常量。因此，用于传播值的时间正比于V且对B个基本块而言，它是$O(B \times V)$。

为估测收敛所需的迭代次数，最重要的是要认清In和Out集合中不同值是各自独立的。即，如果我
666 们正维持一个有V个值的集合，那么我们可以实际平行地做V次不同的数据流分析。如果我们为单个值做数据流分析（可能是活跃变量或可用表达式分析），那么我们所建立的单调性质将要求不超过B次的迭代，这是因为每次迭代时至少有一个值要改变。又因为值是相互独立的，而V>1个值不会改变迭代次数的范围，所以全部分析时间不应当多于$O(B^2 \times V)$。

实践中，B可能不会超过1 000，甚至更少（子程序往往是独立分析的）。V值大约在100左右。开销常量应当不大，因为我们做的大多数操作是位操作，而且这些操作也是按字完成的（每个操作处理16个或32个值；共计处理值$1000^2 \times 100 = 10^8$）。假定处理每个值需要10微秒时间（记住，我们平行地处理16个或32个值），那么分析总时间将限定在1 000秒内，这个时间虽不短但还可以接受。我们迭代B次的估测可能过高了（它假定每个基本块都将影响其他的每一个基本块），因此，几百秒才是一个更合理的上界。

另一种求解流问题的方法（Cocke 1970）是将一个大的流图拆成许多称为区间（interval）的子图（它们往往对应于程序中的循环）。然后单独分析各个子图，并在处理完成后，将整个子图作为包含它的大图中的一个结点。在分析过程中，首先处理的是内层循环，其次是包围它的外层循环，依此类推。该方法计算的优势来自于以下事实：将非线性的算法用于许多小的问题比直接用在大的问题上要便宜得多。使用我们对$O(B^2 \times V)$的估测，如果将图拆成大小分别为B/2的两部分，那么每个子问题的时间成本减少1/4，而总时间将减少1/2。如果将图再拆成许多更小的块，则可能带来更多的时间节省。在下面的讨论中，我们考虑一个结构化的数据流分析技术，它利用了以下事实：在现代程序设计语言（如Ada和Pascal）中，语言的结构化的控制部件可导致程序流图的自然分解。

结构法

对诸如Ada和Pascal这些的现代程序设计语言来说，它们数据流图的大部分可以从结构化的控制部件推导出来。结构化控制部件展示了它们单入口/单出口的特征。这意味着：无论这些部件中控制流如何复杂，它们都具有良好定义的入口点与出口点。凡是使用单一语句的地方也都可以使用这些具有单一入口和出口的控制部件，这样就创建了使用上的一致性和清晰的嵌套层次。并非所有的语言结构都必须是单入口/单出口的。特别地，我们可以跳入或跳出某个结构，而这样将会丧失单入口/单出口的特征。

就数据流分析而言，可以独立地求解结构化的控制部件。我们可以将控制结构中各成员的Gen和
667 Killed集合分别合并为合成的用于整个结构的Gen和Killed集合。这样，整个结构就可以被看成是一个单位，而将In集合映射为Out集合，或反之。我们从基本块开始，求解那些逐步增大的控制结构的数据流方程直至整个程序或子程序求解完毕或者仅剩下无结构的部件为止。在后一种情况下，可用前面章节的技术来求解（再一次将结构化部件看作一个单位）。

在获得整个程序的方程解以后，可将此解传播回结构化部件中，为那些嵌套的结构产生解，最后再为各个基本块产生相应的解。

控制结构中最简单的当数顺序执行结构。假定我们有两个结构S1和S2。这些结构可能是我们已经分析过的基本块或控制结构。每个结构均用它们的数据流方程以标准形式来描述：Out = (In−Killed) ∪ Gen。



图16-34 S1和S2的顺序执行

图16-34中的图概括了穿越这两个结构的数据流。现在，$In = In_1$，$Out_1 = In_2$，以及$Out_2 = Out$。进一步地，$Out_1 = (In_1 - Killed_1) \cup Gen_1$，$Out_2 = (In_2 - Killed_2) \cup Gen_2$。合并后，我们得到：

$$Out = (((In - Killed_1) \cup Gen_1) - Killed_2) \cup Gen_2 =$$
$$((In - (Killed_1 \cup Killed_2)) \cup ((Gen_1 - Killed_2) \cup Gen_2)$$

对这个合并后的结构，我们有：

$$Killed = (Killed_1 \cup Killed_2) \text{ 和 } Gen = (Gen_1 - Killed_2) \cup Gen_2$$

也就是说，我们可以从任何一个结构中生成或注销值，而且在第一个结构中生成的值可在第二个结构中被注销。

条件语句执行的模型如图16-35所示。正常情况下，在S1或S2的条件执行前将计算并测试有关“谓词条件”。这种情况可以被建模为使用顺序结构来连接谓词计算和条件执行结构。条件执行结构的数据流规则是：$In = In_1 = In_2$。对单路径问题而言，$Out = Out_1 \cup Out_2$。而对全路径问题而言，$Out = Out_1 \cap Out_2$。

668

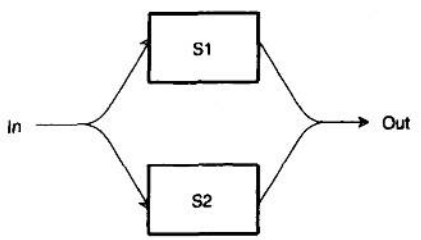


图16-35 S1或S2的条件执行

我们首先考虑单路径情况。使用S1和S2的方程，我们有：

$$\begin{aligned} Out &= ((In - Killed_1) \cup Gen_1) \cup ((In - Killed_2) \cup Gen_2) \\ &= (In - Killed_1) \cup (In - Killed_2) \cup Gen_1 \cup Gen_2 \\ &= (In - (Killed_1 \cap Killed_2)) \cup (Gen_1 \cup Gen_2) \end{aligned}$$

对于单路径情况，我们有：

$$Killed = (Killed_1 \cap Killed_2) \text{ 和 } Gen = (Gen_1 \cup Gen_2)$$

这意味着：如果一个值在条件语句中的两条路径上均被注销，那么此值才被注销；而要生成一个值，则可以在条件语句的任何一条路径上生成它即可。

对于全路径情况，我们有：

$$\begin{aligned} Out &= ((In - Killed_1) \cup Gen_1) \cap ((In - Killed_2) \cup Gen_2) \\ &= ((In - Killed_1) \cap (In - Killed_2)) \cup \\ &\quad ((In - Killed_1) \cap Gen_2) \cup ((In - Killed_2) \cap Gen_1) \cup (Gen_1 \cap Gen_2) \end{aligned}$$

通过注意到我们总是能假定Killed集合与对应的Gen集合不相交，我们可以简化这个方程。也就是说，如果我们注销一个值，然后又（在相同的基本块或结构中）产生它，那么此注销就不会产生任何差别。设对应的Killed集合与Gen集合不相交，则可以证明：

669

$$((\text{In}-\text{Killed}_1)\cap\text{Gen}_2)\subseteq((\text{In}-\text{Killed}_1)\cap(\text{In}-\text{Killed}_2))$$
$$((\text{In}-\text{Killed}_2)\cap\text{Gen}_1)\subseteq((\text{In}-\text{Killed}_1)\cap(\text{In}-\text{Killed}_2))$$

这些不等式允许上述方程简化为：

$$\begin{aligned}\text{Out} &= ((\text{In}-\text{Killed}_1)\cap(\text{In}-\text{Killed}_2))\cup\\ &\quad((\text{In}-\text{Killed}_1)\cap\text{Gen}_2)\cup((\text{In}-\text{Killed}_2)\cap\text{Gen}_1)\cup(\text{Gen}_1\cap\text{Gen}_2)\\ &= ((\text{In}-\text{Killed}_1)\cap(\text{In}-\text{Killed}_2))\cup(\text{Gen}_1\cap\text{Gen}_2)\\ &= (\text{In}-\text{Killed}_1-\text{Killed}_2)\cup(\text{Gen}_1\cap\text{Gen}_2)\end{aligned}$$

$$\text{Killed} = (\text{Killed}_1\cup\text{Killed}_2)\quad\text{和}\quad\text{Gen} = (\text{Gen}_1\cap\text{Gen}_2)$$

在全路径情况中，如果一个值在条件语句中的任何一条路径上被注销，那么此值才被注销；而要生成一个值，则必须在条件语句的两条路径上均生成它。

最后，我们考虑迭代结构。有很多形式的循环，图16-36是其中的一种。我们将集中讨论最常见的形式——**while loop**。如果用顺序结构构造器将初始化代码与之连接的话，则这种形式也可以表示**for loop**。

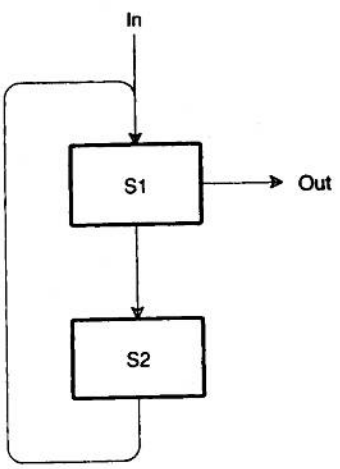


图16-36 S1和S2的迭代执行

S1代表循环终止的测试；S2则表示循环体。为理解循环中可能的路径，我们可以“展开”它。如果循环执行零次，则只有S1执行。如果循环执行一次，那么执行序列为S1;S2;S1。如果循环执行两次，那么执行序列为S1;S2;S1;S2;S1，等等。这些看起来似乎极其复杂，因为循环可以无限地迭代。幸运的是，存在一个非常漂亮的简化。设F1代表将结构S1的In集合映射为它的Out集合的函数。设G代表将结构S1;S2的In集合映射为它的Out集合的函数。由顺序执行的规则，我们知道G可以写成标准形式：((In−Killed) ∪ Gen)。与循环零次执行对应的Out集合是F1(In)。与循环执行一次对应的Out集合为F1(G(In))。与循环执行两次对应的Out集合则为F1(G(G(In)))，等等。

670

现在考虑G(G(I))。因为G是标准形式，所以我们可以将该公式写为：

$$\begin{aligned}&(((\text{I}-\text{Killed})\cup\text{Gen})-\text{Killed})\cup\text{Gen}\\ &= (\text{I}-\text{Killed})\cup(\text{Gen}-\text{Killed})\cup\text{Gen}\\ &= (\text{I}-\text{Killed})\cup\text{Gen} = \text{G(I)}.\end{aligned}$$

这个G(G(I)) = G(I)的结论意味着循环体多于一次的执行不会改变所要计算的Out集合。事实上，循环归结为两种情况：循环执行零次和循环执行一次或多次。

循环的数据流方程和条件执行的那些方程非常类似。对于单路径情况：

$$\text{In}_1 = \text{In}\cup\text{Out}_2\quad\text{Out} = \text{Out}_1\quad\text{In}_2 = \text{Out}_1$$

$$\text{Out}_2 = \text{G(In)}\cup\text{G(G(In))}\cup\text{G(G(G(In)))}\cup\cdots = \text{G(In)}$$

G是与顺序执行的S1;S2对应的函数；如前面所推导的，它是：

$$\text{Out}_2 = (\text{In}-(\text{Killed}_1\cup\text{Killed}_2))\cup((\text{Gen}_1-\text{Killed}_2)\cup\text{Gen}_2)$$

$$\text{In}_1 = \text{In}\cup\text{Out}_2 = \text{In}\cup(\text{Gen}_1-\text{Killed}_2)\cup\text{Gen}_2$$

$$Out = Out_1 = (In_1 - Killed_1) \cup Gen_1$$
$$= ((In \cup (Gen_1 - Killed_2) \cup Gen_2) - Killed_1) \cup Gen_1$$
$$= (In - Killed_1) \cup (Gen_2 - Killed_1) \cup Gen_1$$

$Killed = Killed_1$ 且 $Gen = (Gen_2 - Killed_1) \cup Gen_1$

这个方程表明：对循环的单路径分析，Out集合就是从In集合减去在S1中被注销的，加上在S1中产生的，再加上在S2中产生且没有在S1中被注销的那部分。

对全路径而言，我们注意到：In1要么对应着什么也不执行（即空循环），要么对应着序列S1;S2的一次或多次执行。我们可以使用从条件执行获得的全路径解，其中一条路径为空语句（Killed=Gen=∅），另一条路径是S1;S2（函数G代表这个序列）。因而，我们有：

671

$$In_1 = (In - (Killed_G - \varnothing)) \cup (Gen_G \cap \varnothing)$$
$$= (In - (Killed_G)) = (In - (Killed_1 \cup Killed_2))$$

为了获得Out集合，我们应用S1的数据流方程：

$$Out = (In_1 - Killed_1) \cup Gen_1$$
$$= ((In - (Killed_1 \cup Killed_2)) - Killed_1) \cup Gen_1$$
$$= (In - (Killed_1 \cup Killed_2)) \cup Gen_1$$

$Killed = (Killed_1 \cup Killed_2)$ 且 $Gen = Gen_1$

全路径解注销了那些可在循环中任意路径被注销的值（即，只有那些保持在所有路径上的值才会被保留下来）。Gen集合仅包含Gen_1，因为总要执行的只有S1。

为了说明我们已经开发出的结构化方法，让我们重新考虑图16-33中所示的例子。各个基本块的Gen和Killed集合如下表所示：

	b0	b1	b2	b3	b4	b5	b6
Gen	∅	∅	∅	∅	∅	∅	{I}
Killed	{Limit,I}	∅	{J}	{Sum}	{Sum}	{I}	∅

我们按照由内到外的方式来处理流图，并将单独的基本块合并为复合结构：

Step	Structure	Combining Rule	Killed	Gen
1	b3,b4	Conditional	{Sum}	∅
2	b2,b3,b4	Sequential	{J,Sum}	∅
3	b2,b3,b4,b5	Sequential	{J,Sum,I}	∅
4	b1,b2,b3,b4,b5	Iterative	∅	∅
5	b0,b1,b2,b3,b4,b5	Sequential	{Lim,I}	∅
6	b0,b1,b2,b3,b4,b5,b6	Sequential	{Lim,I}	{I}

In和Out集合按照由外至内的方式传播，先求解整个程序的方程，然后再解内层嵌套结构的方程。 672

Step	Block	In Set	Out Set
1	b0	{Limit,I,J,Sum}	{J,Sum}
2	b1	{J,Sum}	{J,Sum}
3	b6	{J,Sum}	{I,J,Sum}
4	b2	{J,Sum}	{Sum}
5	b3	{Sum}	∅
6	b4	{Sum}	∅
7	b5	∅	∅

结构化的数据流方法的最大优点是，该方法是线性的。分析时，基本块首先被合并。在每个合并步，基本块和结构的总数减1，因此我们仅需做B步合并，B为基本块的个数。每步花时间O(V)，V为Gen和Killed集合的大小（通常表示为位向量）。在合并阶段完成后，我们可以访问单独的基本块，使用那些已知的集合值以及基本块和合并后结构的方程来确定基本块单独的In和Out集合。这个时间正比于$O(B \times V)$，因此，算法的全部时间也是$O(B \times V)$。和迭代方法最差情况$O(B^2 \times V)$相比，这是一个明显的改进。

16.5 集成优化技术

我们已在最近这两章里讨论了各种优化。现在，我们将简要地讨论一下在编译处理中各种优化的集成问题。大多数优化编译器允许用户选择他们期望的优化级别。许多场合中并不需要优化，而其他一些场合中则可能要求适度的或广泛的优化。

代码优化的投入虽然增加了编译时的复杂性却换来了运行时的性能提高。优化代价可由多种形式体现。优化编译器体积庞大、运行慢且造价比普通的产品质量级的编译器高许多。而且，并非所有的优化都是安全的，因此优化后的程序通常不如相应的未经优化的程序健壮。

在程序开发和调试阶段，任何优化都是不需要的。只有到达了最终的产品生产成形阶段，再做与谨慎的代码生成集成在一起的局部优化才是合适的。对大多数产品生产程序而言，局部优化可能就是它们所需要的全部。在使用频繁或关键的程序中可能需要进一步的优化。但在这种情况下，“不遗余力”地进行所有可能的优化是不必要的，也是不明智的。相反，我们更倾向于使用剖析工具来检查并发现程序
673 中成为性能瓶颈的热点所在。一般地，子程序或代码片段尤其是循环，将被确定为关键部件。

接下来，我们将对这些已知的事关程序性能的部件进行全局优化。对循环来说，优化可能会涉及不变式外提或强度削弱。对子程序而言，内联展开、调用优化或过程间分析等被证明是有用的优化手段。

在关键部件优化后，需再一次剖析程序以查看是否出现新的热点，或原先的热点在优化后是否仍然是瓶颈。在后一种情况下，需要重新编程相关的代码片段或子程序。正是因为单独使用优化不是解决性能问题的万能药，所以剖析技术才成为一个相当有价值的工具。那些糟糕的算法，即使被施行了大量的优化，也还是不能替代那些现有的为程序量身定制的灵巧算法。如果算法是经过审慎选择的，那它们可能不需要优化。而在其他的情况中，对关键例程所做的仔细优化只是最后的有价值的增色而已。

无论是在为单独的子程序还是为整个程序做全局优化的时候，其中单项优化施行的次序很重要。一般来讲，那些能够揭示其他优化机会的优化行为将被首先实施，而与代码生成集成在一起的局部优化将在最后施行。特别地，我们建议按以下步骤施行优化：

- 在分析程序结构并将其综合到IR代码时，要施行能够展开程序结构的优化（典型的是循环展开和内联展开）。这就允许对展开后的结构实施进一步的分析和简化。例如，子程序实参内联展开取代形参，此举可带来合并（优化）的机会。
- 在综合IR代码时，要建立基本块。同时计算那些用在数据流分析（能够描述基本块特征）的参数（如Def和Killed集合）。
- 在建立基本块后，可以构造程序或子程序的流图。在数据流分析中，必须考虑出现在基本块里的（过程或函数）调用的影响。如果程序或子程序是各自独立分析的，那么在基本块中无论何时遇见调用都必须做出最坏情况下的假设。即，假定所有的表达式都要被注销，所有的变量可能被引用并修改，等等。

  此外，我们可以在不考虑控制流的情况下检查子程序文本以便更好地估测子程序调用所带来的影响。因此，在一次调用中，如果变量v的使用出现在子程序体文本中的任一地方，我们就假定v将在此调用期间被使用。

  如果我们在遇到调用时可以分析相应子程序的流图，我们就可以获得更准确的信息来确定调用的影响。即，我们用子程序的流图来取代它的调用，并分析整个扩展后的流图。但这是过程间（interprocedural）的流分析。由于存在递归，流图实际上是不可能被替换的。相反，我们可用一种
674 迭代的方法来近似计算调用的实际影响（参见练习22）。
- 接下来可以施行那些揭示其他优化的数据流优化。特别地，常量和复写传播的施行可以很容易地识别某些不可达的基本块，并由此将这些基本块作为“死”代码而删除。因此，在以下程序片段中，

```
if A = 1 then
  B := 1;
end if;
```

传播某个常量值到A有可能导致对B的赋值不可达。同样，删除B的赋值也可能带来其他常量传播和复写传播的机会，因此，这些优化将反复施行直到流图不再发生变化为止。

- 再接下来，施行那些能够揭示冗余代码的优化。此类优化包括CSE分析和非活跃变量的识别。删除冗余代码不会改变流图的结构，因此这类分析仅需实施一次。
- 再往下，由代码生成器将基本块翻译为目标代码。在代码生成时，可施行基本块内的局部优化。这些优化被集成到代码生成里是因为它们与寄存器分配、代码选择密切相关。在诸如PDP-11和VAX那样有长、短格式分支指令的机器上，连接基本块的分支指令格式是在基本块被翻译为目标代码之后选择的。可以在内存中重排基本块以便用短分支指令替代长分支指令。更多的时候，在基本块被翻译为目标代码时，它们才被简单地映射到内存，并在可能时将长分支替换为短分支。
- 最后，可以利用窥孔优化来甄别已生成的代码并做最后的改进。

在组织这些优化步骤时，较明智的做法是将每个阶段（代码生成除外）做成可选的。这将减轻开发与调试的负担并很容易禁止不需要的优化。

练习

1. 下列优化中哪个是不安全的？哪个我们总能从中获益？
   - 代码提升（见16.4.5节）
   - 常量传播（见16.4.5节）
   - 将标量型变量装入未使用的寄存器
   - 将下面循环

675

```
for I in 1 .. 2*N loop
 {loop body}
end loop;
```

   部分展开为：

```
for J in 1 .. N loop
   {loop body with I replaced by 2*J-1}
   {loop body with I replaced by 2*J}
end loop;
```

2. 在大多数编译器中，优化全部还是部分程序将由用户来决定，这通常可以通过设置选项或标志来进行。一种替代的办法是自动地做出选择，使得当满足某种条件时，就自动地触发并实施优化。

   应当使用什么样的条件来触发优化的施行？当施行优化的决定发生在程序作为产品使用之后，将如何修改优化过程？

3. 考虑下面的子程序：

```
type MarkedVocabulary is array(Vocabulary) of Boolean;

function MarkLambda(G : Grammar) return MarkedVocabulary is
   -- Mark those vocabulary symbols found to derive λ (directly or indirectly)
DerivesLambda : MarkedVocabulary;
Changes : Boolean := True;        -- Any changes during last iteration?
RHS_Derives_Lambda : Boolean := False;    -- Does the RHS derive λ?
NumProds : Integer := 100;    -- Number of Productions in Grammar
RHSLen : Integer;    -- Length of Current RHS
begin
   for V in Vocabulary loop
      DerivesLambda(V) := False; -- Initially, nothing is marked
   end loop;
```

```
while Changes loop
   Changes := False;
   for P in 1 .. NumProds loop
      RHS_Derives_Lambda := True;
      RHSLen := RHS_Length(P);
      for I in 1 .. RHSLen loop
         RHS_Derives_Lambda :=
            RHS_Derives_Lambda and DerivesLambda(RHS(P)(I));
      end loop;
      if RHS_Derives_Lambda and not DerivesLambda(LHS(P)) then
         Changes := True;
         DerivesLambda(LHS(P)) := True;
      end if;
   end loop;
 end loop;
end MarkLambda;
```

676

确定在此程序上可能施行的局部优化和全局优化。哪些是我们从中可以获益的？哪些是最难实施的？

4. 给出下面程序框架所对应的调用图：

```
program Main is

  procedure A is
    D;
    ...
    B;
  end A;

  procedure B is
    C;
  end B;

  procedure C is
    ...
  end C;

  procedure D is
    C;
    ...
    E;
  end D;

  procedure E is
    ...
  end E;

begin
  A;
  ...
  B;
end Main;
```

5. 能否静态分配练习4中子程序的活动记录？如果可以，给出使用最少可能空间的分配方案。
6. 证明16.2.2节里的静态活动记录分配技术总是使用最少可能的空间，假定调用图所示的所有调用序列都是可能的。

677

7. 考虑练习4中的程序。假定程序已被翻译但寄存器尚未被分配。下表中给出了每个例程所需的寄存器数：

例程	所需寄存器数
Main	3
A	1
B	2
C	1
D	3
E	2

假定有8个硬件寄存器可用于分配。你如何将这8个寄存器分配给主程序和各个子程序以最小化调用之间的寄存器保存和恢复？

8. 我们都已经知道，确定任何不进行I/O的程序是否终止的问题是不可判定的。即，我们不可能构造一个可以正确判定程序在所有情况下都终止的算法。证明这个“停机”问题可被简化为可达性问题。如果在执行期间无论是否到达程序指定语句它总是确定的，那么我们可以根据可达性来定义并求解出“停机”问题。因为知道“停机”问题不可求解，所以我们也得出结论，可达性问题必定是不可判定的。
9. 解释如何施行下列过程调用的内联展开。此内联展开还能使哪些优化成为可能？

```
A : Real := 2;
B : Real := 21;

procedure P(Flag : in Boolean; S : in out Real) is
begin
  if Flag then
      S := S **2;
  else
      S := S/A;
  end if;
end P;

P(False,B);
Write(B);
```

10. 解释在施行内联展开时，如何使用常量传播来优化值参的引用，以及如何使用复写传播来优化标量型**in out**参数的引用。
11. Ada/CS子程序的内联展开必须正确地解析参数和局部变量的引用。在程序包P中定义的子程序Q可以引用通常在包外不能引用的局部定义于P的变量。有无可能在P外内联展开Q的调用，并假定某些Q可访问的变量在调用点仍不可访问？如果可能，那么将如何处理到那些在通常情况下不可访问变量的引用呢？ 678
12. 我们注意到递归调用如果出现在内联展开的子程序中将可能引发问题。但在某些情况下，递归调用还是可以被容忍的。例如，如果Fact是以通常的递归方式实现的阶乘函数，那么调用Fact(5)可用来做内联展开，而且这个展开可以被优化到单个常量值。

    参数为常量的递归调用是否总能被安全地内联展开？如果不能，另外需要什么样的约束条件来控制涉及递归的内联展开？
13. 计算下面代码片段的Def(P(B,C))和Use(P(B,C))。使用这些Def和Use信息，判断A+B和D+F是否被调用P(B,C)所注销。

```
declare
  A,B,C,D,E,F,G : Integer;

  procedure Q(Z : out Integer; X : in Integer) is
  begin
    Z := X + F;
    Write(A);
  end Q;

  procedure P(I : in Integer; J : in out Integer) is
  begin
    E := I + J;
    A := B + C;
    Q(J,G);
    if E > F then
      P(J,E);
    end if;
  end P;
begin
  C := A + B;
  G := D + F;
  P(B,C);
  ...
end;
```

14. 如果我们已知函数没有副作用，那么将很容易优化该函数调用。由于Ada和Ada/CS仅允许In参数，所以实参不会在函数调用期间被修改。然而，变量可在调用期间被修改并有时会带来副作用。假定已计算函数调用的Def集合。解释如何使用该集合来确定调用是否带来副作用。

679

15. 证明图16-7中的算法compute_use_set()总会终止。然后证明它计算的Use集合是正确的。也就是说，如果在调用P期间使用了变量v，那么v ∈ Use(P)；反之，如果v∈Use(P)，那么变量v实际上可能在调用P期间被使用。

16. 考虑以下程序片段：

```
for I in 1 .. N loop
  for J in 1 .. N loop
    if M(J)(I) then
      for K in 1 .. N loop
        M(J)(K) := M(J)(K) or M(I)(K);
      end loop;
    end if;
  end loop;
end loop;
```

使用factor_invariants()（见图16-11）来外提循环不变式。

现在重写下标表达式以揭示执行变址所需的计算。假定M是N × N的布尔数组。使用factor_invariants()来外提包含在变址代码中的不变式。使用strength_reduce()来强度削弱包含在变址代码中的乘法。

17. 在16.3.1节里，我们注意到从**while loop**中外提循环不变式是不安全的，即使这个表达式非常忙。这是因为**while loop**可能迭代零次，且通常不可能事先预测控制循环的布尔表达式是否初始为真。另一方面，Pascal语言的**repeat-until**循环允许安全地外提非常忙循环不变式，因为它必须至少迭代一次。

证明**while loop**可被重写为一个通过**if**语句进入的**repeat-until**循环。解释如何使用这种变换来安全地外提**while loop**中的非常忙循环不变式？在什么情况下这种变换不合乎要求？

18. 建立下面子程序的数据流图：

```
type ArrayArg is array(Integer range <>) of Integer;

procedure Sort(A : in out ArrayArg) is
  Temp : Integer := 0;
begin
  for I in reverse A'First .. A'Last-1 loop
    for J in A'First .. I loop
      if A(J) > A(J+1) then
        Temp := A(J+1);
        A(J+1) := A(J);
        A(J) := Temp;

      end if;
    end loop;
  end loop;
end Sort;
```

680

19. 列出在练习18中定义的Sort所计算的表达式。对于Sort的数据流图中的每一个基本块b，确定其Computed(b)，即在b中计算且随后没有在b中被注销的表达式集合。同时还要确定Killed(b)，即在b中被注销的表达式集合。

利用Sort的数据流图和刚计算出的Computed与Killed值，对Sort实施可用表达式分析，确定每个基本块的AvailIn和AvailOut集合。采用两种方法做这个数据流分析，首先采用迭代技术，然后再使用结构化方法（均见16.4.6节）。正常情况下，两种技术应当产生相同的解。

最后，使用刚计算出的可用表达式信息在Sort中实施CSE优化。

20. 重新考虑练习18中定义的Sort。这一次做活跃变量分析。我们再次使用两种分析方法，首先采用迭代技术，然后再使用结构化方法（见16.4.6节）。检验两种方法是否产生相同的解。Sort中的任一赋值语句能否因为被赋值的变量是非活跃变量而被删除呢？

21. 在Pascal语言里，值和引用参数模式常常被混淆。特别地，因为值模式是默认的模式，所以有时值参的使用方式看起来就像在使用引用模式一样。区分这种混淆的一个标志是，值参在其值使用前一般要进行赋值。说明如何使用数据流来识别那些使用前定义的值参。

22. 假定我们希望建立程序`live_s_vars(P)`来识别在子程序P的入口处活跃的变量。当编译P的调用时，可能使用`live_s_vars(P)`来确定哪些变量必须在调用P之前加以保存。我们已经研究了活跃变量的分析，但`live_s_vars(P)`却提出了另外的关注——如何处理在P中出现的子程序调用。由于可能存在着递归，因此在调用点插入子程序的数据流图是行不通的。

    作为一种选择，可以使用迭代方法。设`live_s_vars(P)`假定所有变量在P入口处活跃，`live_s_vars(P,1)`是它的第一次近似。为计算`live_s_vars(P,2)`，使用`live_s_vars(Q,1)`来描述在P中调用Q的影响。类似地，为计算`live_s_vars(P,i)`，使用`live_s_vars(Q,i-1)`来描述在P中调用Q的影响。此过程将持续到`live_s_vars(P,i) = live_s_vars(P,i+1)`。 681

    证明这种迭代方法将终止且终止于：

    `live_s_vars(P,i) = live_s_vars(P)`.

23. 代码下沉（sinking）优化是对16.4.5节中的代码提升优化的补充。代码提升在公共前驱块中计算值，从而使各后继块中的再次计算变得冗余。类似地，代码下沉将赋值语句移到公共的后继块，从而使各前驱块中同样赋值变得冗余。例如，在下面的程序片段中，

```
if A = B then
        C := A + B;
        B := 0;
else    C := A - B;
        B := 0;
end if;
```

    将B赋值为0的语句可以移到紧接在if语句之后的地方。这就使if语句中的两条B的赋值语句成为给非活跃量的无用赋值。

    要使代码下沉可行，“下沉”的赋值语句所移至的基本块必须是原来各（相同）赋值语句必然的后继块。此外，从原来的各赋值语句到新赋值语句的所有路径上，该赋值语句右部的值一定不能被改变且左部的值也一定不能被引用。

    形式化一个数据流问题以确定出现在一个或多个前驱块中的赋值语句能否移到给定的基本块。将这个分析结果用到与图16-26中的`code_hoist()`例程类似的`code_sink()`例程中。

24. 我们知道基本块b的支配（必经）结点是任意基本块d，如果到b的所有路径上都必须经过d。计算b的必经结点集合Dom(b)的一个粗略方法是，列出从初始基本块b_0到b的所有非循环路径。而集合Dom(b)就是那些恰好出现在所有列出的非循环路径上的结点集合。

    然而，下面给出的却是一个巧妙的方法。根据定义，$Dom(b_0) = \{b_0\}$。将Dom(b)，$b \neq b_0$，近似为B，B为所有基本块的集合。很明显，该近似估计过高。我们注意到，对任意基本块b，$Dom(b) = \{b\} \cup \bigcap_{i \in P(b)} Dom(i)$。即，b总是其自身的必经结点。此外，如果c控制支配b，则它必定也控制支配所有b的直接前驱。

    建立一个迭代算法，它使用前面描述的方法来计算所有基本块的必经结点集合。证明所计算的必经结点集合是正确的。 682

25. 假定按照练习24中描述的方法计算必经结点。证明对于每个必经结点集合Dom(b)-b，存在惟一的

成员i ∈ Dom(b)−b，i是b的直接必经结点。也就是说，对所有的j ∈ Dom(b)−b，j∈Dom(i)。如果基本块i是基本块b的直接必经结点，那么i是离b最近的必经结点且因此成为最合理的地方来移动从b中外提的代码。

26. 在某些程序设计语言里，对变量的类型不做声明。相反，变量的类型可由它们的使用方式推断出。例如，在A := B+1.0中，如果加法涉及实数，它必定产生实数，于是A的类型也一定是实型。假定我们可以确定所有字面值的类型，并且没有自动的类型转换，则所有表达式的结果类型由它们的操作数类型决定。说明如何使用数据流技术来确定变量的类型。最初，所有变量应当被假定为任意类型。在程序的最后，该数据流分析将告诉我们变量所具有的确切类型，或者告诉我们没有类型可与变量的使用方式一致，或者告诉我们不能惟一地确定变量的类型（由于上下文不充分）。

27. 证明图16-31中的算法在求解单路径问题时可以计算出一个最小解。即，如果 $\overline{\text{In}}$ 是数据流方程的任意有效解，而In是图16-31中的算法所计算的解，那么 $\text{In} \subseteq \overline{\text{In}}$。（提示：初始时，$\text{In}_0 \subseteq \overline{\text{In}}$。）

28. 我们一般假定，如果有充足的未被占用的寄存器可供使用，那么把那些使用频繁的变量保存在寄存器中是较为明智的做法。因此，基于循环下标和程序参数将在循环或子程序中被频繁访问的假定，我们可以将这些对象指派到寄存器中。

    与其求助于这个真假难料的一般性，我们倒不如尽可能地估测每个变量的引用频率。如果做了这个预测，那么我们将把引用最频繁的变量指派到寄存器中，直到分配完所有空闲的寄存器为止。估测变量引用频率的最好方法是剖析程序的执行。如果没有可用的剖析数据，那么可以实施一个形式与数据流分析类似的引用频率分析。

    该分析可以通过对程序流图进行结构化分析来完成。主程序执行一次。如果一条语句估计要执行N次，那么顺序后继（语句）也将执行N次。如果条件语句执行M次，那么各个分支语句所估计的执行次数的总和一定是M次。通常在**if**语句中，假定**then**和**else**分支各执行M/2次。如果一个循环执行L次，那么它的循环体将执行L × P次，其中P是估测的循环迭代次数。对于常量循环边界的**for loop**循环，可以精确确定P；对于其他的循环，可能要估测循环的迭代次数（如假定每个循环迭代10次）。

683 一旦确定每条语句预期的执行频率，那么就很容易计算出每个变量预期的引用频率，而其中引用最频繁的变量将被指派到寄存器中。假定循环迭代10次，且**then**和**else**分支执行次数相同，请估测练习18中的子程序里的每条语句的执行频率。假定我们有三个寄存器可用于分配，那么我们的分析将建议哪些变量可以被指派到这些寄存器上？

29. 练习28中实施的分析假定特定的变量在程序或子程序中被指派到特定的寄存器上。这在某种程度上比较浪费，原因是在程序的某些地方，变量可能是不可访问的或非活跃的，因此也就不需要在那些地方被指派到寄存器中。请设想该如何使用活跃/非活跃信息来改进练习28中描述的寄存器分

684 配方案。

第17章　现实世界中的语法分析

在第3、5和6章里，我们详细研究了为现代程序设计语言构造词法分析器和语法分析器的问题。在这一章里，我们将考虑在“现实世界”分析中的两个至关重要的问题——表压缩和错误修复。

那些由词法分析器和语法分析器的自动生成器所产生的表通常是大而稀疏的。但经仔细处理，我们可以有效地压缩那些表并少许降低访问表元素的时间。17.1节的主题即为各种表压缩技术的调研与分析。

我们已提出的分析理论主要解决正确输入的有效分析。而实践中，我们还是需要某种处理错误输入的办法。这是一个复杂的问题，且不存在最佳答案。在17.2节里，我们调查了各种实际使用的方法。自动生成的错误处理程序因其特别容易使用而受到特殊的关注。

685

17.1　压缩表

存储由ScanGen、LLGen和LALRGen这样的工具产生的表的最明显的办法是采用普通的一维或二维数组。这种表示在用于二维表时空间开销比较大，特别是当表稀疏的时候更是如此。稀疏表中的大多数条目均被设置为特殊的默认值。例如，Ada/CS的LL(1)文法包含了70个终结符和138个非终结符。采用数组形式存储的LL(1)分析器动作表需要大约10 000（138 × 70）个条目。根据实际经验，这些表中一般仅含有10%左右的非出错条目，因此我们可以通过不同于普通数组的一些表示方法来获得更大的空间节省。

现在考虑其他表示二维表T（N × M）的方法。假设有E个非默认的条目，其中E<<N × M。我们的目标是：在表的表示方法中表空间大小是与E而不是与N × M成正比，且同时将保持到T[i][j]的快速访问。

我们最初考虑的方法和那些用于符号表的方法类似。普通符号表和压缩表的重要区别是：表T中的条目均为事先已知的，而符号表的条目则是在编译过程中被动态创建或撤销的。

在后面的讨论中，我们使用图17-1中的一个5 × 5的表作为示例。表中的空白表示默认的条目；其他条目则用单个字母表示。

L			P	
	Q			R
		U		
W	X			
	Y		Z	

图17-1　一个稀疏数组

二分搜索

可以根据非默认条目的索引排序来创建一张表。此表需要3 × E个条目（索引i和j以及条目的值）。所需条目可以通过基于索引的二分搜索来查找，其时间要求为O(log(E))。对于图17-1中的数组，我们可以创建如图17-2所示的二分搜索表。

假设每个条目占用一个单元的存储空间，如果3 × E<N × M，我们即获得了空间上节省。查找时间正比于log(E)。随E变大，查找有可能慢得难以接受。

686

哈希表

索引i和j可被散列到哈希表中的位置k，使用以下规则：如果位置k已被占用，我们将尝试k+1%S，随后尝试k+2%S，等等。（记住：%代表C语言中的取模运算。）S是表的大小，这种处理表中冲突的办法称为线性法（linear resolution）。此哈希表中的每个位置上要求3个条目{i,j,T[i][j]}，并且我们要求S>E（表中有一个空闲位置可以避免搜索的条目不在哈希表中时出现死循环）。

使用图17-1中的例子，我们创建了如图17-3所示的哈希表，其中S=E+1。所用的哈希函数是h(i,j)=i*j%S。

1	1	L
1	4	P
2	2	Q
2	5	R
3	3	U
4	1	W
4	2	X
5	2	Y
5	4	Z

图17-2 一个表示为二分搜索表的稀疏数组

2	5	R
1	1	L
5	2	Y
5	4	Z
1	4	P
2	2	Q
4	1	W
4	2	X
3	3	U

图17-3 一个表示为哈希表的稀疏数组

此哈希表中非默认条目的平均查找次数约为1.89次，而默认条目的平均查找次数约为4.5次。

一般地，如果3 × S<N × M，哈希表将节省空间。查找时间与表的大小、哈希函数和未使用条目的数量相关（S和E之差）。如果集合S的设置很接近E，那么查找时间将由于频繁的冲突而被延长。然而，因为T中所有条目均事先已知，所以我们可以调整S和哈希函数以降低哈希表中条目的平均查找次数。事实上，如果使用3.5节里的完美哈希技术，那么哈希表中的探针可以满足任何非默认条目的定位。

行压缩

可以将二维数组看成是指向其各行的指针的向量。即，T[i][j]可被映射为V[row[i]+j]，其中V是包含数组各行的向量，这些行是首尾相连的。row[i]给出了V中第i行的开始位置。事实上，这种映射
687 通常用于在内存中查找给定的数组元素的位置，用到的规则为row[i]= (i-1)*M。

为节省空间，我们可以删除行中的默认条目并存储每个非默认条目的值和列索引。此外还必须保存一个称为row[i]的向量，它给出在剔除默认条目后V中第i行的偏移。我们从V[row[i]]到V[row[i+1]]-1的条目中寻找索引值j。如果找到，那么此条目中的下一个值即为T[i][j]；否则，T[i][j]一定是默认值。

688 对于我们上面5 × 5的例子，我们可能会得到如图17-4所示的表。这种表示方法需要2 × E而不是3 × E个条目，再加上row向量中的N+1个条目。如果采用顺序搜索，T[i][j]的平均查找时间为ND(i)/2，其中ND(i)=row[i+1]-row[i]是T的第i行中非默认条目的个数。如果使用二分搜索，查找时间为log(ND(i))。

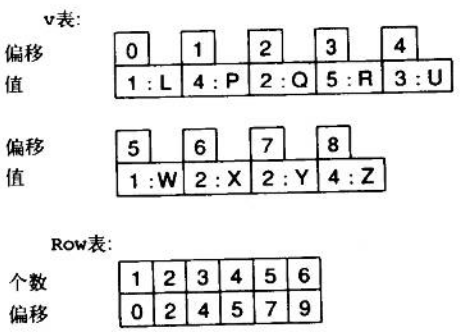


图17-4 一个采用行压缩表示的稀疏数组

双重偏移索引

如果T的某些行有较多的非默认条目，那么使用行压缩方法时查找时间可能会成为较棘手的问题。而这里介绍的双重偏移索引方法试着以适度增加所需空间为代价来确保快速查找。此想法很聪明：各行在彼此相关的偏移处相互重叠以便某一行中非默认条目总是覆盖其他行中的默认条目。这就保证了在给定的位置上仅有一个非默认条目。行号与条目存放在一起以便确定所关联的非默认条目的所在行。我们可以使用图17-5中的表格来表示前面的例子数组。

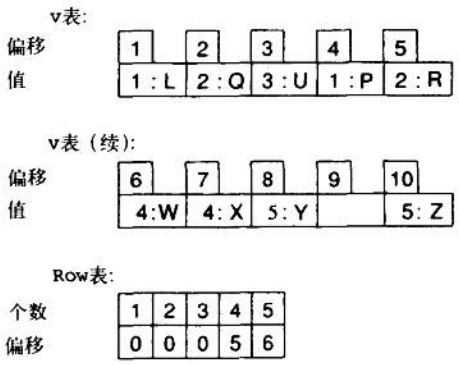


图17-5 一个使用双重偏移索引表示的稀疏数组

为得到T[i][j]，我们查找V[row[i]+j]。如果此条目为空白，则返回默认条目。如果V[row[i]+j]的第一个成员是i，那么我们将它的第二个成员作为T[i][j]的值返回；否则，我们仍然返回默认条目。在所有情况下，我们仅检查了表V中的一个条目，因此速度非常快。最佳情况下的空间需求几乎和采用行压缩的方法一样，均为2 × E+N个条目。然而，此最佳情况下的空间需求通常无法实现，而经验告诉我们实际所需的空间与此最佳情况已相当接近。

作为一项实验，我们曾研究过此双重偏移表示法可否用于Ada/CS的LL(1)分析表。那个未压缩的表大小有9 660个条目（70个终结符乘上138个非终结符）。其中，非默认条目个数E为629，占全部条目数的6.51%。当压缩长度为70的行时（每一行代表着某个特定非终结符的预测），表V中需要660个条目（超出最佳可能4.9%）。而当转置此分析稀疏矩阵的行和列后，可以压缩长度为138的行（每一行是超前搜索终结符所对应的预测），此时表V中需要879个条目（超出最佳可能39.7%）。造成这两种情况下表V大小差异的原因是较长的行（138对70）比较难以压缩。 689

表V中各行的覆盖顺序可以影响最终的表的大小。因此，如果我们在先前的例子中以1、5、2、4、3的顺序覆盖各行，则可能得到如图17-6所示的最佳的布局。

一般地，最佳方式的行覆盖问题是NP-完全的（Garey和Johnson，1979）。它意味着已知最好的算法需要被覆盖的行数的指数阶时间，这也就几乎等于尝试所有的排列。

行覆盖的一个较好的启发式方法称为最优递减法（best-fit decreasing）。各行是以所含非默认条目密度递减的顺序相互覆盖，而且所选的每次覆盖都将缩减表V的大小。其关键点在于首先覆盖带有许多非默认条目的行（它们与其他非默认条目的冲突最多），而最后覆盖那些最容易（含有少量非默认条目）的行（以填充表V中的“空洞”）。我们将在附录F中讨论使用此最优递减方法创建双重偏移索引的数组压缩实用程序。Tarjan和Yao (1979)深入讨论了这种双重偏移索引的方法。 690

17.1.1 压缩LL(1)分析表

我们可以利用LL(1)分析表的特殊性质来进一步减少空间需求而同时对查找速度的影响却甚少。我

们知道，在存储压缩表时，必须随条目一起存放用于条目标识的一个或两个索引。因此，对于有E个非默认条目的表，我们需要2 × E或3 × E个条目。在LL(1)分析表中，索引可以是待展开的非终结符，或者是超前搜索的终结符。表中默认的条目代表分析出错的情况；非默认条目则是产生式的名字。但如果T[A][a]是某个产生式的名字（通常被编码为一个整数），则A必定是那个产生式的左部。这意味着在压缩表中不需要显式地将A与T[A][a]存放在一起。相反地，我们创建可将产生式映射为它们相应左部的向量LHS。如果T[A][a]不是出错条目，那么LHS[T[A][a]]=A。如果产生式的数目小于表中非默认条目的数目（通常是这种情况），那么我们通过在每个条目中存储LHS向量而非一个索引就可以进一步获得空间上的减少。

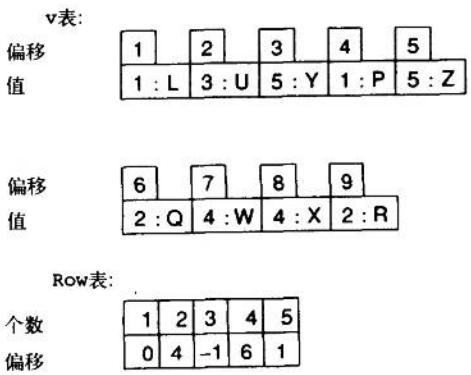


图17-6 使用双重偏移索引的稀疏数组的最佳表示

这种改进体现在当各行对应一个给定的超前搜索终结符时所使用的行压缩技术中，以及当各行对应一个给定的非终结符时所使用的双重偏移技术中。在行压缩方法中，我们检查从V[row[a]]到V[row[a+1]-1]的条目。每个条目是产生式的索引。如果LHS[V[i]]=A（其中，row[a]≤i≤row[a+1]-1），那么V[i]=T[A][a]；否则，考虑V[i+1]。

类似地，对于双重偏移技术，我们检查V[row[A]+a]。如果LHS[V[row[A]+a]]=A，那么T[A][a]=V[row[A]+a]，否则T[A][a]=error。为测量此项修改的价值，我们可以回过来再看Ada/CS的LL(1)分析表。如果我们显式地将索引及其相应的条目存放在一起且使用较短的行（因为它们能更好地覆盖），那么我们可能需要660 × 2个条目+一个row向量（大小为138）= 1458个条目。如果使用向量LHS来避免存储任何索引，那么我们现在只需要660 × 1个条目+一个row向量（大小为138）+一个LHS向量（大小为252）= 1050个条目，空间减少28%。

17.2 语法错误的恢复与修复

当编译器发现语法错误时，它通常想做的事就是试图使语法分析（或整个编译）过程继续进行下去以便发现更多的错误。而这些涉及到*错误恢复*（error recovery）或*错误修复*（error repair）的工作。

错误恢复

进行错误恢复时，我们试着重新设置语法分析器以便分析处理剩余的输入。此过程可能涉及修改语法分析栈或剩余输入。根据错误恢复完成的好坏，后续的语法错误有可能是真实的，或它们可能是受前面的语法错误株连而得到的。例如，在…A := B C + D; …中，恢复算法或许会从词法记号C处预测一条语句而重新开始语法分析。然而，这将导致在 + 处有一个虚假的语法错误。

691

衡量错误恢复例程质量好坏的主要依据是看它所导致的虚假错误或株连错误的多少；即，看它将语

法分析器和剩余输入相同步的精准程度。正常情况下，在进行了错误恢复后，语义和代码生成例程均被禁止，因为此时再去执行那些编译器的输出已没有任何意义。

最简单的错误恢复办法称为*紧急方式*（panic mode）。在紧急方式下，语法分析器尝试“紧急跳出”某个语言构造，同时寻找一个安全的符号（例如分界符或语句的首部）以重新开始分析。紧急方式通常在单独使用的时候效果并不令人满意，但它常常是其他所有办法均失败时的一个可依靠的办法。

错误修复

在进行错误修复时，我们尝试将用户的输入修复为有效的程序。此过程可能涉及修改已分析过的输入或者（更常见的是）修改剩余输入。在重新开始编译后，仍可能发现后续的错误。同样地，这些错误可能是真实的用户错误或由先前修复动作所导致的虚假错误。因此，在…A := B C + D; …中，修复算法可能在B后面插入分号（;），而此举将在符号+处引起一个虚假的或级联的语法错误。另外，如果修复算法在B后面插入的是运算符**or**，则有可能导致一个虚假的语义错误。

错误修复算法的准确性可以通过它的修复动作所导致的虚假错误的多少来衡量。修复算法实际上在尝试着修复有错误的输入，而且在修复工作完成后语义处理和代码生成阶段还将继续进行。正常情况下，可以执行带有错误修复功能的编译器的输出，尽管其中可能存在的语义错误会在运行时调用终止例程或调试器。

错误修复（error repair）有时被人称作错误校正（error correction），但我们认为这多少有些用词不当。“校正”一词使人想到的是我们可以提供用户实际想要的。而这一点恰恰是我们无法做到的。相反，我们最希望做的是能将一个非法的语言构造修复成合理的语言结构。因此，与“错误校正”相比，“错误修复”更多地是对实际发生的情况的一种描述。

一般来讲，错误修复算法要比错误恢复算法复杂得多，因为前者实际上必须尝试修复错误的输入而同时还要维护用来继续进行翻译的语义信息。相比之下，一个恢复例程只要能重启分析过程且粗略地改变（或抛弃）相关分析栈和剩余输入的内容即可。

尽管如此，错误修复算法总还是可以用于错误恢复，因为如果我们可以修复一个输入，那么我们当然可以继续分析它。

在设计错误恢复和修复算法时，我们利用了以下事实：所有我们感兴趣的语法分析技术，包括SLR(1)、LALR(1)、LR(1)和LL(1)，均有*正确前缀特征*（correct prefix property）。也就是说，如果我们有输入配置yTz（其中，输入串y已被读入，终结符T被发现在语法上是非法的，串z是剩余输入），那么我们知道y是某个合法程序的前缀[y … ∈ L(G)]而yT不是[yT … ∉ L(G)]。在此上下文中可以有三种不同类型的修复操作： 692

(1) 修改y。

(2) 在y和T之间插入串v使得yvT … ∈ L(G)。

(3) 删除T以便yz成为待处理的对象（然后可将操作(1)、(2)或(3)应用于这个已修改的输入串）。

这三种修复操作的吸引力并不相等。特别地，操作(1)无疑是较差的。通常，y因其已被分析而不是直接可用的。此外，如果即将采取错误修复动作，则y的修改在一遍编译器中一般要求取消已完成的语义和代码生成操作——而这点是很难或不可能简单而有效进行的。

恢复算法有时也（通过弹出分析栈内容）修改y，但此举被认为是过激的，因为y已经被接受且被验证为语法有效的。事实上，当y已知为有效时，我们并不需要去修改它，所有的修复操作可以限制在第(2)和第(3)类上。因此，大多数修复算法选择从不修改y（以避免改变语义），而恢复算法也只把y的修改作为最后的手段。

请注意，尽管如此，排出了对y的修改有可能也排出了某些想要的修复。例如，下面的例子中的错误是“缺少**if**”：

```
··· ; B then C:=0; end if; ···
```

正常情况下，在发现任何错误前，这里的B将被归约为某个左部。而在这一点，**if**的插入将实际修改y这个已被接受的输入前缀。这个错误在实践中很少出现而且许多修复算法也只是简单地把它给忽略掉。另一种较有吸引力的方法是采用*出错产生式*（error production）来预测此类可能性。出错产生式是添加到文法中用来预测特定错误的产生式。实际上，它扩展了语言的语法以覆盖那些已预料到的错误。当一个出错产生式被识别时，编译器将产生特殊的错误信息以描述该出错产生式所表示的修复动作。对于缺少**if**首部的这种错误，我们可以使用如下出错产生式：

```
<if> → if <b expr>
<if> → <b expr>
```

后一条产生式包含了缺少**if**首部的情况。添加出错产生式时必须要谨慎处理，因为有可能引入分析冲突，而这些冲突又必须要得到解决。

现在，我们把在错误记号的左边插入新记号的办法与删除错误记号的办法做个比较。正常情况下，
693 插入将优于删除：一般地，在用户输入周围建立正确的程序要比从那个输入中删除一部分要安全一些。实际上，在某些称为*可通过插入校正的*（insert-correctable）语言里，总有可能仅通过插入操作来修复任何出错的记号序列。Ada/CS是可通过插入校正的语言，因为它的程序是由包序列组成的。也就是说，在Ada/CS中，除结束标记以外的任何终结符都可被生成为包的一部分。当发生语法错误时，我们总是有可能完成当前包的分析并将出错符号生成为下一个包的一部分。因此，任何语法错误都可以通过适当的记号插入而得到修复。

插入校正对恢复算法有着特殊的意义，因为在那些算法中，插入操作所带来的简单性很有价值。即，一个总能正常工作且通常工作得很好的简单、便宜且紧凑的恢复算法将是理想的算法。由其他修复操作而非插入操作所引入的额外复杂性在恢复算法中未必是划算的。在另一方面，错误修复算法通常还必须允许删除操作（尽管在实践中，插入要比删除普遍得多）。

17.2.1 即时错误检测

为保持最广泛的修复操作，有必要确保语法分析器在尽可能早的时间里发现错误。理想情况下，当错误的记号首次出现在超前搜索符中时，我们就可以发现相关的语法错误。我们称在首次看到某个记号时即发现该记号为非法的语法分析器具有*即时错误检测特征*（immediate error-detection property）。但是，普通的SLR(1)、LALR(1)和LL(1)语法分析器发现错误的时机要稍迟一点——即在试图移入非法记号的时候才会发现错误。错误检测时的这种轻微延迟能够对可利用的错误修复操作的范围产生巨大的影响。

考虑图17-7中所示的文法G_1。假设我们使用LL(1)分析器分析输入串ID) ··· $。所发生的分析栈移动序列是：

$$S' \Rightarrow E\$ \Rightarrow TE'\$ \Rightarrow ID\ E'\$ \Rightarrow E'\$ \Rightarrow \$$$

此时错误被发现。发生最后一次移动是因为) ∈ Follow(E')。现在，因为分析栈中只剩下$，所以惟一可能的修复动作就是删除剩余输入中所有一直到结束标记的记号。这是一个相当极端的做法，尤其是当所分析的语言是可通过插入校正的时候。

正如5.11节所讨论的，我们在实践中使用的LL(1)分析器实际上是强LL(1)分析器。强LL(1)分析器在
694 超前搜索符号a满足a ∈ Follow(A)时预测产生式A → λ。而a是否实际有效则在预测后才加以判断。相比之下，完全LL(1)分析器不做任何预测，除非已知超前搜索符号是有效的。强LL(1)分析器所需的分析表比完全LL(1)分析器所需的分析表要明显小得多。类似地，SLR(1)、LALR(1)和优化的LR(1)分析器也不具备完全LR(1)分析器的即时错误检测特征。

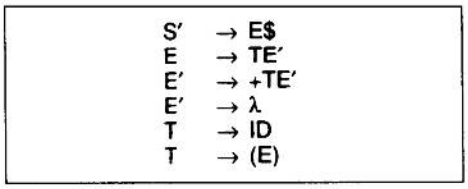


图17-7 文法G_1

尽管即时错误检测特征很有价值，但因为完全LL(1)或完全LR(1)分析器所需的分析表过于庞大，所以我们还是要避免使用它们。作为替换办法，我们可以缓冲分析移动和由超前搜索符号引起的语义例程调用。如果该符号最终被移入栈中，那么我们就知道它是合法的。于是，我们可以清除那些被缓冲的分析移动并执行被缓冲的语义例程调用。如果不能移入该超前搜索符号，那么我们可以用被缓冲的分析移动将分析栈恢复到首次使用非法的超前搜索符号时的那个栈的配置。

一般地，对于LL(1)分析器，我们将做出的预测缓冲在一个栈上直到出现成功的移入为止，而那时我们才可以清除此缓冲栈。为撤销预测，分析栈中的产生式的右部可被替换为预测它们的相应产生式的左部符号。因而，在前面的例子中，我们就可以缓冲分析移动：predict E' → λ。当发现) 是非法的时候，我们就用此缓冲来撤销那个预测，将$替换为E'$。

类似地，对于SLR(1)、LALR(1)和优化的LR(1)分析器，我们将它们归约所用的产生式保存在一个栈中。为撤销某个归约操作，我们从分析栈中弹出一个状态，接着再压入相应产生式右部每个符号的后继状态。例如，如果我们有含状态$s_1 \cdots s_n$的分析栈（s_n位于栈顶）而且希望撤销产生式A → abc的归约，那么我们可以首先弹出状态s_n而露出状态s_{n-1}。然后我们可以压入状态 $\hat{s}_n$、$\hat{s}_{n+1}$、$\hat{s}_{n+2}$，其中go_to $[s_{n-1}][a] = \hat{s}_n$，go_to$[\hat{s}_n][b] = \hat{s}_{n+1}$，go_to$[\hat{s}_{n+1}][c] = \hat{s}_{n+2}$。

如果执行的仅仅是错误恢复，那么通常不需要缓冲（尽管性能可能会降低）。但如果试图使用错误修复，那么我们就推荐缓冲技术（以撤销不合适的预测或归约）。(Mauney和Fischer[1981]讨论了另一个较有吸引力的用于强LL(1)分析器的缓冲技术。)

17.2.2 递归下降分析器中的错误恢复

Wirth（1976，5.9节）研究了一种可用于递归下降分析器的简单、一致的错误恢复方法。递归下降分析中的每个分析过程被设计用来匹配某个非终结符所生成的终结符串。出于错误恢复的目的，我们给每个分析过程提供一个能在过程返回后进行匹配的符号集合。此集合，即follow_set，可在检测到错误的时候用于重新同步输入。如果有必要，分析过程可以跳过若干输入记号直到发现集合follow_set中的符号为止。此时分析过程可以较安全地返回，因为我们知道下一个输入符号将被调用过程所匹配。

在大多数语言中有诸如**begin**和**if**那样非常重要且不能被跳过的首部符号。因此，我们把集合header_set添加到在集合follow_set中以确保首部符号能被保留下来并稍后由后续的分析过程匹配。例如，考虑下面的Ada程序片段：

695

```
··· if B then A := 1 else ···
```

Ada中的语句都必须以分号（;）结束，但此例中我们可不想为搜索这个符号而跳过**else**部分。将**else**包括进集合header_set中可以确保与**then**部分相连的**else**部分不被丢掉。

最后，因为可能存在的输入错误，分析过程不能肯定它自己一定会匹配当前的输入。为此，我们使用valid_set（它代表所有期望的合法输入）来筛选当前输入。正常情况下，集合valid_set包含集合First(A)的成员，A是分析过程匹配的非终结符。如果A可以产生λ，则集合valid_set还可包含集合Follow(A)的成员。Wirth建议在每个分析过程的入口处调用如图17-8所示的例程。

```
void check_input(terminal_set valid_set,
                 terminal_set follow_set,
                 terminal_set header_set)
{
   if (next_token() ∈ valid_set)
      return;     /* Input looks valid */
   else {
      syntax_error();   /* Mark next token as illegal */
      while (! (next_token() ∈
             (valid_set ∪ follow_set ∪ header_set)))
         skip_token();
         /* Skip this token;
            call scanner() to get next token. */
   }
}
```

图17-8 跳过非法记号的算法

在调用check_input()例程后，我们知道next_token()或者有效，或者它将被调用过程继续处理。无论哪种情况，错误恢复均已被适当地分解到了单个例程中。

我们对递归下降分析器所做的其他主要的修改涉及由过程match()的调用所发现的语法错误。我们知道，调用match(T)试图无条件地将next_token()匹配为T。如果匹配失败，我们就发现一个语法错误。为恢复由match()发现的语法错误，我们调用syntax_error()来标记此错误并返回。而next_token()无需改变并且它将被重新考虑，要么匹配要么在返回前被跳过。实际上，因为match(T)知道下一个记号必须是T，所以我们可以在它不匹配的时候插入记号T。作为特殊情况，如果还有任何剩余，match(SCANEOF)将跳过所有的剩余输入。它这么做是因为在匹配文件结束符前，所有的输入都必须被处理和消耗掉。

696

syntax_error()标记那些被认为是非法的记号。通常，一些连续的记号会被这样标记。而语法错误信息仅为第一个被标记的记号而非为其他剩余被标记的记号而产生。我们其实并不知道那些剩余的已标记的输入是否非法，我们所知道的只是它们已被跳过。有时，我们会为第一个未被标记的记号产生一条信息（“分析已恢复”）以强调那些中间的符号已被分析器所忽略。

作为示例，考虑图17-9中的分析过程。该例子曾出现在第2章中，现在添加了错误恢复的内容。

```
void statement_list(terminal_set follow_set)
{
   check_input(SET_OF( ID, READ, WRITE ),
               follow_set,
               SET_OF( SCANEOF ));
   statement(follow_set);
   while (TRUE) {
      switch (next_token()) {
      case ID:
      case READ:
      case WRITE:
         statement(follow_set);
         break;
      default:
         return;
      }
   }
}
```

697

图17-9 一个带有错误恢复内容的分析过程

在例程statement_list()的入口处，我们使用例程check_input()来筛选当前输入。valid_set是First(<statement_list>)。follow_set则是作为参数被接收，尽管对Micro文法的研究表

明：此集合总是单元集（**end**）。在内容较丰富的语言中，集合follow_set将依赖分析过程被调用时的上下文。因此，在Ada语言中，如果正被匹配的<statement list>是**if**语句的**then**部分，则集合follow_set可以包含**else**。

集合header_set中包含符号Eof，因为我们不想跳到输入结束符的后面。该集合也可以包含诸如ID和READ那样的语句首部，但这是多余的，因为那些符号已出现在集合valid_set中且不可能被跳过。在statement_list()过程体中的statement()的调用被赋予follow_set参数，这是因为正在匹配的语句可能是语句列表中的最后一句。实际上，还可以在follow_set中添加First(<statement>)，但这也没有必要，因为例程statement()将立即匹配那些在First(<statement>)中的符号。

这种方法已应用于实践而且效果还可以。像所有的紧急方式的处理技术一样，它的主要缺点在于它相当随意地跳过输入符号，从而试图将当前输入与合适的分析过程同步。对于嵌套结构而言，这种跳过输入符号的方法就不能很好地工作，这也就是为什么我们尝试添加首部符号集合的原因。

一直以来，很少有人做递归下降的错误修复工作。其问题在于分析状态是隐式地存储在分析过程的调用栈里，而且不容易决定何种修复动作可作为有效措施而被接受。此外，由于分析过程里夹杂着语法分析和语义处理，因而当不止一个修复动作看似可能的时候就不知道该测试执行哪一个了。

17.2.3 LL(1)分析器中的错误恢复

针对递归下降分析器所开发的错误恢复方法可以用在LL(1)分析器中。图17-10中所示例程将在LL(1)分析器驱动程序发现语法错误的时候调用，它跳过若干输入符号并弹出栈符号直到语法分析可以重新开始为止。

```
/*
 * Will this combination of stack_top
 * and token cause a syntax error?
 */

static boolean parse_error(symbol stack_top,
                           terminal current_token)
{
   if (stack_top∈Vn)
      return T[stack_top][current_token] == ERROR;
   else
      /* stack_top∈Vt */
      return stack_top != current_token;
}

void ll_recovery(void)
{
   /* Let S be the parse stack;
      let a be the current input token. */
   while (parse_error(top(S), a)) {
      if (top(S)∈Vn) {
         /* topl() "peeks" at top(pop(S)) */
         if (parse_error(topl(S), a)
               && ! (a ∈ header_set))
            scanner(a);  /* Skip current token */

         else    /* Remove top stack symbol */
            pop(S);

      } else {    /* top(S)∈Vt */

         if (top(S) == SCANEOF)
            scanner(& a);
            /* Never skip past endmarker */
         else
```

图17-10 用于LL分析器的简单的错误恢复例程

```
                pop(S);
                /* Match expected terminal; */
                /* then try again */
            }
        }
    }
```

图17-10 （续）

ll_recovery()必须重新设置分析器以便恢复语法分析。它可以通过跳过输入符号或弹出分析栈条目来达到此目的。如果分析栈的栈顶是终结符（它不能匹配当前的超前搜索符），我们就将此栈顶弹出而插入所需的终结符。如果栈顶是非终结符，我们要么将其弹出栈顶，要么删除当前的超前搜索符。

如果在弹出分析栈有关内容后可以恢复语法分析，那么我们就进行此类操作。这种情形类似于当一个分析过程返回后它的调用者才能恢复语法分析一样。如果这种出栈操作不能使语法分析得以恢复，同时当前的超前搜索符不是被保护的首部符号，我们就可以删除这个当前的超前搜索符。受保护的符号是不能被删除的；相反，我们要弹出分析栈中有关内容以便受保护的符号可以匹配某些位置较深的栈符号。

这种恢复算法完全是启发式的。有时它从栈中弹出条目，强制进行匹配；有时它会跳过输入，希望能和下一个记号匹配。我们需要用集合header_set来防止跳过那些重要的记号，但我们却没有精确地定义它。在17.2.4节里，我们提出一个更形式化的有关LL(1)恢复和修复的方法。特别地，在那个方法中我们做了更加仔细的分析以确定何时删除或插入符号，而且当存在多种可能的修复动作时我们还引入了一个最佳选择的概念。

17.2.4 FMQ LL(1)错误修复算法

我们现在考虑由Fischer、Milton和Quiring（1980）提出的FMQ错误修复算法。这是一种可自动产生的（automatically generable）错误修复算法，设计它是用来与可通过插入校正的LL(1)文法一起工作。在17.2.5节里，我们给出了可以和任意LL(1)文法一起工作的该算法的一种扩展。

698
~
699

因为这种算法操作在可通过插入校正的语言上，因而它可以将它的修复动作限制为终结符串的插入。这一点和先前那些偏好删除输入符号的技术形成鲜明的对比。这种只进行插入的方法，其优点是在已有记号的周围进行修复，从而也就无需定义那个受保护的首部记号集合。

如果一个语言是可通过插入校正的，那么对于任何语法错误（$S \Rightarrow^* x \cdots$和$S \not\Rightarrow^* xa \cdots$，$x \in V_t^*$，$a \in V_t$），我们总可以选择并使用终结符串$y \in V_t^*$达到修复的目的（$S \Rightarrow^* xya \cdots$）。正如先前所提及的，一些典型的程序设计语言都是非常接近于可通过插入校正的。

接下来，我们假设我们能够尽可能快地发现语法错误（即，当错误符号被首次用作超前搜索符的时候就发现这个错误）。因为使用的是普通的LL(1)分析器，所以我们还可以假设某个先前描述过的技术（如缓冲技术）可被用来将分析栈恢复到错误符号首次作为超前搜索符出现时已经存在的栈配置。

和许多修复和恢复技术不同，FMQ算法被证实具有如下优良特性：

- 它可以修复任何输入。
- 它的修复动作是通过插入代价可调的。
- 它完全是表驱动的，因此可被自动生成。
- 它的时空需求是线性的。
- 它选择的修复动作总是局部最优的。

为控制修复算法所做出的插入选择，我们将专门使用一个描述插入代价的整数向量。即，$\forall a \in V_t$，我们有$C[a] > 0$。C[a]是插入符号a的代价。C[a]的值越大，符号a被插入的可能性越小（如果有那么一个选择的话）。

插入λ等价于什么也不插入，因此C[λ] = 0。类似地，结束标记总是最后被分析的记号且从不需要被插入。因此，C[$] = ∞。最后，我们引入一个新符号? ∉ V_t，满足C[?] = ∞。对于串$X_1 \cdots X_n$ ($n \geq 0$)，$C[X_1 \cdots X_n] = C[X_1] + \cdots + C[X_n]$。(在我们的算法中，我们将使用函数C(a)来返回用于任意终结符串的C[A]值)。

为驱动修复算法，我们事先计算并存储以下两张表：

(1) $S: V \rightarrow V_t^*$

S[A] = 最小代价$z \in V_t^*$，使得$A \Rightarrow^+ z$

且对于$a \in V_t$，S[a] = a。

(2) $E: V \times V_t \rightarrow V_t^* \cup \{?\}$

对于$A \in V_n$: E[A][a] = ?，如果$A \not\Rightarrow^+ \cdots a \cdots$

否则，E[A][a] = 最小代价$w \in V_t^*$，使得$A \Rightarrow^+ wa \cdots$。

对于$a \in V_t$: E[a][b] = ?如果$a \neq b$

E[a][a] = λ

700

非正式地，S[X]给出可由X推出的最小代价串，而E[X][a]则给出允许a可由X推出的最小代价前缀。

我们可以很容易地计算这两张表并把它们保存在内存中或文件里直到修复算法需要使用它们。使用这两张表，我们可以定义一个如图17-11所示的相当简单的错误修复算法。

```
terminal_string find_insert(stack_of_symbol parse_stack,
                            terminal a)
{
   /* a is the error symbol */
   terminal_string insert, least_cost;

   insert = "?";
   least_cost = λ;
   for (i = 0; i <= depth(parse_stack) - 1; i++) {
      if (C(least_cost) >= C(insert))
         break;
         /* No lower cost insertion can be found */

      if (C(least_cost @ E[parse_stack[top-i]][a])
            < C(insert)) {
         /* A better insertion has been found */
         insert = least_cost @ E[parse_stack[top-i]][a];
      }
      least_cost = least_cost @ S[parse_stack[top-i]];
   }
   return insert;
}
```

图17-11 计算LL(1)最小插入代价的算法

在本章剩余部分提出的算法采用了比我们先前所使用的更加高级的伪代码描述。该描述添加了以下新的特征：简单的串赋值通过符号 = 进行；串比较通过 == 进行；运算符@执行串连接操作；通过指定片段的上下界将数组分段。

find_insert()简单地沿LL(1)分析栈向下搜索以寻找推出错误符号a的最便宜的方式。因为LL(1)文法是可通过插入校正的，所以我们知道某些栈符号一定可以推出a。

作为示例，让我们再次考虑那个能生成简单表达式的文法G_1（见图17-7）。首先，我们必须确定插入代价。而关于如何选择代价却没有固定的规则。一般地，如果某些终结符的插入不合乎要求，则它们的代价会比那些插入后无任何副作用的其他符号的代价更高一些。在文法G_1里，ID的插入是我们所不想要的，因为标识符一般都带有语义的内容。类似地，符号(的插入也不具有吸引力，因为稍后我们可能将不得不插入与之匹配的符号) 。尽管如此，符号) 或 + 的插入却少有麻烦。这种分析可导致如图

701 17-12所示的插入代价。

一旦选择了插入代价，就可以分别计算S表（见图17-13）和E表（见图17-14）。现在假设我们正在分析 ID +) + ID $。在遇到) 时我们检测到语法错误。此时，LL(1)分析栈包含T E' $。现在将调用例程find_insert()，而出错符号是)。文法G_1是可通过插入校正的，因此) 必定可由栈中某个符号推出。首先，考虑栈顶符号T。而E[T][')'] = (ID ；此插入的代价为4。接下来，考虑E'。S[T] = ID且E[E'][')'] = + (ID。因此，ID + (ID也是一个可能的插入串，但它由于代价过高（为7）而遭拒绝。最后，考虑的是$，但它却不能推出错误符号。因此，最终返回的最小代价的插入是 (ID。

符号	代价
(	2
)	1
+	1
ID	2

图17-12 文法G_1的插入代价

符号	最小代价的插入
S'	ID $
E	ID
E'	λ
T	ID

图17-13 文法G_1的S表

非终结符	终结符				
	ID	(	)	+	$
S'	λ	λ	(ID	ID	ID
E	λ	λ	(ID	ID	?
E'	+	+	+ (ID	λ	?
T	λ	λ	(ID	(ID	?

图17-14 文法G_1的E表

尽管算法find_insert()相当简单，但我们还是能很容易地证明先前列出的有关特征：

702
- 它能修复任何输入，因为每次find_insert()的调用都允许接受至少一个或多个输入符号。
- 仅通过改变代价向量的值并重新计算S表和E表就可以调整它的修复动作。
- 它完全是由可自动生成的S表和E表来驱动的。
- 它的执行时间可确保在线性范围内。真实的分析器通常使用*深度有限*（bounded-depth）分析栈（即，分析栈的最大深度是某个实现常量）。对于深度有限的分析栈而言，find_insert()的一次调用至多花费有限的时间，而且O(|x|)次调用仅需O(|x|)的时间。即使在允许分析栈有O(|x|)的深度的一般情况下，我们也可以创建针对所有的调用仅需要O(|x|)时间的新版本的find_insert()（见练习16）。
- 由S表和E表所选择的插入是*局部最优的*（locally optimal），因为不存在比它们所做的选择具有更低插入代价且让分析器接受错误符号的插入动作。

很明显，此算法的简单性也使得它可以有非常高效的实现。另外，如果将S表和E表保存在文件中，那么正确的程序几乎不需要为编译器内建的错误修复功能付出任何东西。

自然有人会问由find_insert()实施的修复操作究竟好在哪里。我们根据测试情况所得出的答案是：相当得好，但并不总是最好。find_insert()的优点可以总结如下：

- 由于其性能良好且极其简单，因此它是一种优秀的错误恢复技术。
- 它是一种不错的、但不是最好的修复算法。正如我们将要看到的，对find_insert()所做的一些简单扩展可以极大地提高它的修复质量且不会过多增加其代价或复杂性。

如果有必要的话，S表可以采取比我们目前所讨论的方法更为紧凑的方式来存储。也就是说，我们与其在终结符串的表上建索引，但不如将每个非终结符和从它开始能进行最小代价推导的产生式存放在一起。当需要S[A]的时候，即进行从A开始的最小代价的推导。S[A]的生成很快，且仅需存储一张从非终结符到产生式的映射表。此外，E表的条目也可以在需要时从S表的条目和文法产生式中计算得到（参见练习7b）。这种计算也很快，只消片刻即可建立E[X][a]，这里的a是一个特定的错误符号。

17.2.5 在FMQ修复算法中添加删除操作

一个高质量的错误修复算法必须不时地做一些删除以获得最佳的可能的修复。而且，如果允许删除，我们可以将FMQ算法应用于任意的LL(1)文法而不仅仅是可通过插入校正的LL(1)文法。

现在，我们研究如何将删除操作添加到FMQ算法中。就像我们为插入所做的，我们也假设有整数向量D，其中对$a \in V_t$，D[a]是删除a的代价。很自然地，我们有$D[\$] = \infty$和$D[X_1 \cdots X_n] = D[X_1] + \cdots + D[X_n]$。 703
类似地，D(a)返回任意终结符串的D[a]值。假设剩余的输入符号串是$b_1 \cdots b_m$，其中b_1是出错符号。为使$b_1 \cdots b_m$可用，有必要扫描剩余输入并把结果记号保存在一个队列里。回想一下，修复动作是由两个参数决定的：delete，待删除的输入符号数；insert，在删除后待插入的串。可以使用如图17-15所示的算法来获得delete和insert的最小代价值。即，使用此算法可以计算下式的最小值：

$$\min_{0<i<m} \min_{y \in V_t^*} \{D(b_1 \cdots b_i)+C(y) \mid xyb_{i+1} \cdots \in L(G)\}$$

其中，输入串为$xb_1 \cdots b_m$，insert = y，delete = i。

```
void ll_repair(terminal_string *ins, int *d)
{
  /*
   * Remaining input = b1 · · · bm.
   * Optimal repair is to delete d tokens,
   * then insert string ins.
   */

  *ins = "?";
  *d = 0;
  for (i = 1; i <= length(b); i++) {
     if (D(b[1 .. i-1]) >= C(*ins) + D(b[1 .. *d]))
         break;
         /*  No lower cost repair is possible */

     if (C(find_insert(parse_stack,b[i]))
                + D(b[1 .. i-1])
         < C(*ins) + D(b[1 .. *d])) {
       /* Better repair found */
       *ins = find_insert(parse_stack,b[i]);
       *d = i-1;
     }
  }
}
```

图17-15 一个计算LL(1)最小修复代价的算法

作为示例，我们再次考虑在17.2.4节中用过的那个例子。现在，我们必须既定义删除代价又定义插入代价，如图17-16所示。为简单起见，我们使用相同的插入和删除代价；而在较复杂的文法中，特定符号的插入和删除代价通常是不一样的。 704

Symbol	Cost
(	2
)	1
+	1
ID	2

图17-16 文法G_1的删除代价

当使用LL(1)分析器分析串ID +) + ID $时，我们在遇到) 时即检测到语法错误并在那时调用ll_repair()。我们首先考虑零个符号的删除，然后再考虑一个、两个……符号的删除，此过程将持续到我们确信已找到最小的修复代价为止。如果删除零个符号，也就等于我们只进行插入修复。正如我们在上一节所知道的，修复此错误的最小代价的插入串是 (ID，其代价为4。而删除) 的代价为1。现在，+ 被认为是错误符号，且find_insert()建议插入符号ID，其代价为2。此次修复的总代价是删除代价加上插入代价，其结果为3。这个代价比只进行插入修复的代价要改进一些。接下来，我们考虑) +的删除，这个代价为2。错误符号现在是ID，且find_insert()报告最小代价的插入串为λ。(插入λ等

价于什么也不插入。）此次修复的总代价为2，这又是一次改进。再接下来，我们考虑删除）+ ID，但这个删除的代价为4且不能带来一个更便宜的修复。因此，ll_repair()最终选择了删除）+ 和插入λ作为最佳的修复动作。

算法ll_repair()倒是非常的简洁易懂，但它的最差情况时间界限却是$O(|x|^2)$。这里，深度有限栈的假设不再起作用；针对$O(|x|)$个错误中的每一个错误，我们可能要反复地处理剩余输入，且我们可能需要查看所有的剩余输入。然而，对于实际感兴趣的例子，我们可以证明那个$O(|x|^2)$的执行时间不会发生。特别地，再次假设一个深度有限的分析栈且对$\forall a \in V_t$，D[a]>0（在一般情况下，D[a]=0是可能的，但不是非常有用——它使删除太容易了）。可以证明，一个给定的输入符号，在ll_repair()的多次连续调用后，至多可以被考虑删除常量次，并且由此可达线性。通过使用更复杂版本的ll_repair()，我们可以证明一般情况（$O(|x|)$的栈深度，允许D[a]=0）也具有线性特征（Fischer、Mauney和Milton 1979）。

在实践中，我们当然不会在调用ll_repair()时扫描并排列所有的剩余输入。相反，在需要考虑某些输入符号的时候，我们将渐增式地扫描并排列记号。即，要么立即删除一个记号（因为没有使它成为合法输入的插入操作），要么只检查少量的输入记号并把它们保存在分析重新启动点之上（以便验证不存在更小的修复代价）。

705 当允许删除时，FMQ修复操作的质量将变得非常好。这种方法的“闪光点”其实就在于它易于使用。编译器的作者只需要指定插入和删除的代价向量——其余的工作均可自动完成。当然，LL(1)错误修复和恢复方法的简单性和效率也是促成我们在现实世界编译器中使用LL(1)的主要因素。

17.2.6 FMQ算法的扩展

通常，可以通过将修复分成优秀（excellent）、良好（good）和较差（poor）三个等级来评价一种错误修复算法的好坏。优秀的修复正是那些人们所期望做到的。良好的修复虽不是那些人们所期望的但也是较合理的。较差的修复显然要比人们所期望做到的差，而且它们常常会导致随后的虚假错误。

FMQ算法经过一整套标准的测试（Ripley和Druseikis 1978），产生大约28%的较差的修复。很明显，我们希望减少这一数字。（良好的修复与优秀的修复之间的区分往往因人而异，但代价的调整有助于使较好的修复变得更好。）

很多研究人员（Graham、Haley和Joy 1979；Burke和Fischer 1982）均得出结论，即修复的验证在过滤那些较差的修复时特别有用。为验证一个修复动作，语法分析器将被重新启动（而语义处理则被禁止）。如果语法分析器能够接纳最小数目的记号而不引起语法错误，那么此次修复即被验证有效并可以实际用于输入的修复。如果被提议的修复没能通过验证，那么我们就会拒绝它并考虑其他的修复。

Mauney（1982）将局部最小代价的概念推广到了区域最小代价的修复。在区域最小代价修复中，代价被最小化至程序中固定大小的区域内。区域中可以包括错误的修复以及为验证修复所包含的合适的上下文。区域中也可以包含两个或多个相连错误的修复。Mauney发现，在使用大小为5个记号的区域时，采用Ripley-Druseikis测试集所得的较差修复的百分比可以减至9%以下。Mauney提议的算法由于代价过高而不能用于普通的编译器中，但它确实让我们看到了验证修复所带来的改进机会。

在FMQ修复算法中加入验证修复的机制并不是件难事。可以将LL分析栈和一个待尝试修复的前缀馈送到一个骨架型分析器中，由它最终确定是接受还是拒绝此修复。我们可用图17-17中那个基于ll_driver()的例程来验证修复。

17.2.5节里的ll_repair()可被推广至validated_ll_repair()，如图17-18所示。这两个例程除了validated_ll_repair()在修复被接受前需要v+1个符号验证外，其余的都基本相同。

validated_ll_repair()要求在将插入操作包含进一个修复前必须要对其加以验证。图17-19中所示

的例程find_validated_insert()计算允许符号串validation_prefix在分析重启后被接受的最小代价插入。在例程find_validated_insert()的设计中，需解决的问题是：如果某个最小代价插入被验证检查所拒绝，我们该怎么办？答案是，在有验证的情况下，我们需要在找到合适的修复前先考虑一系列代价逐渐升高的插入。 706

```
boolean ll_validate(stack_of_symbol parse_stack,
                    terminal_string prefix)
{
   /* If all of prefix can be parsed,
      it is considered validated. */

   i = 0; /* Initial position in prefix string */
   while (i < length(prefix) && depth(parse_stack) > 0) {
      if (top(parse_stack)∈Vn) {
         if (T[top(parse_stack)][prefix[i]] ==
                        X → Y1 · · · Ym)
            /* Expand nonterminal */
            Replace top(parse_stack) with Y1...Ym;
         else
            return FALSE;
            /* Validation attempt has failed. */

      } else {  /*  top(parse_stack)∈Vt */
         if (top(parse_stack) == prefix[i]) {
             pop(parse_stack); /* Match worked. */
             i++;
         } else
             return FALSE;
             /* Validation attempt has failed. */
      }
   }

   /* The whole prefix was correctly parsed. */
   return TRUE;
}
```

图17-17 一个验证LL修复的算法

```
void validated_ll_repair(terminal_string *ins,
                         int *d, int v)
{
  /*
   * Remaining input = b1 · · · bm.
   * Optimal repair is to delete d tokens, then insert
   * string ins.  v is a validation count: we must
   * validate the repair on bd+1 · · · bd+1+v.
   */

  *ins = "?";
  *d = 0;
  for (i = 1; i <= length(b); i++) {
    if (D(b[1 .. i-1]) >= C(*ins) + D(b[1 .. *d]))
        break;
        /* No lower cost repair is possible */

    len = min(i+v,length(b));
    if (C(find_validated_insert(parse_stack,b[i .. len]))
         + D(b[1 .. i-1]) < C(*ins) + D(b[1 .. *d])) {
        /* Better repair found */
        *ins = find_validated_insert(parse_stack,
                                     b[i .. len]);
        *d = i-1;
    }
  }
}
```

图17-18 一个计算已验证的LL修复的算法

```
terminal_string
find_validated_insert(stack_of_symbol parse_stack,
                      terminal_string validation_prefix)
{
  /*
   * Find a least-cost insertion that allows
   * the validation_prefix to be accepted.
   */
  terminal_string insert, least_cost, t;
  terminal a = validation_prefix[0];
  auto int last_soln[depth(parse_stack)] = {/* all 0s */};
  int soln_i;  /* i value used in least-cost solution */

  /* Loop until a validated insertion is found. */
  while (TRUE) {
     insert = "?";
     least_cost = λ;
     for (i = 0; i <= depth(parse_stack) - 1; i++) {
        if (C(least_cost) >= C(insert))
            break;
            /* No lower cost insertion can be found */

        t = least_cost @
            E[parse_stack[top-i]][a][last_soln[i]];
        if (C(t) < C(insert)) {
            /* A better insertion has been found */
            insert = t;
            soln_i = i;
        }
        least_cost = least_cost @ S[parse_stack[top-i]];
     }

     /* Now try to validate insert */
     if (insert == "?")
         return insert;
         /* Failure return; */
         /* no validated insertion found */
     else if (ll_validate(parse_stack,
                          insert @ validation_prefix))
         return insert;
     else
         last_soln[soln_i]++;
         /* Record rejected solution and try again */
  }
}
```

图17-19 一个计算已验证的LL插入的算法

我们可以推广E表以包含第三个参数（索引），一个整数i。非正式地，E[A][a][i]是允许a可被A推导出的第i个最便宜的前缀。即：

$E: V \times V_t \times N \rightarrow V_t^* \cup \{?\}$

对于$A \in V_n$: E[A][a][i] = w，$w \in V_t^*$是第i个最小代价前缀（$A \Rightarrow^* wa \cdots$）。

否则，E[A][a][i] = ?

707 对于$a \in V_t$，E[a][b][i] = ?如果$a \neq b$ 或 $i > 0$ E[a][a][0] = λ

注意：使用我们最初的定义，我们知道E[A][a][0]和E[A][a]具有相同的值。

我们仍然想获得有最便宜的可能插入，但它现在必须服从下一个输入符号加上v个验证符号应该是可分析的这一条件。设$X_1 \cdots X_i$是分析栈最上面的i个符号，y是已分析的输入符号，$b_1\ b_2 \cdots$是剩余的输入符号。我们必须设法求解下面的公式：

$$\min_{1 \le i \le n} \min_{0 \le j} \{C(S[X_1]) + \cdots + C(S[X_{i-1}]) + C(E[X_i][b_1][j]) \mid y\, S[X_1] \cdots S[X_{i-1}] E[X_i][b_1][j] b_1 \cdots b_{v+1} \cdots \in L(G)\}$$

这个求最小值的公式看起来很怕人，但它其实是说我们现在必须以v个符号的验证为条件来获取最小代价的插入。如果$E[X_i][b_1][0]$不起作用，我们可能要考虑$E[X_i][b_1][1]$、$E[X_i][b_1][2]$ …。find_validated_insert()求解此最小值。它发现最便宜的插入并试图验证这个插入。如果验证失败，它就尝试下一个最便宜的插入，然后是再下一个，等等，直到发现一个验证正确的插入或者直到没有代价有限的插入剩下为止。 708

在那个求最小值的公式里，插入操作可以由两个参数来描述：i，在分析栈中使用函数E的位置；j，选择第j次最便宜值的索引。我们的算法知道j = 0时得到最便宜的可能的E表的值，因此它最初考虑每个栈位置时使用j = 0，然后找到大体上最便宜的插入的位置（称它为i）。这个处理过程和我们先前定义的find_insert()所做的完全相同。如果验证失败，我们寻找下一个最便宜的插入，记住不要再试我们先前的插入。这是通过在考虑栈位置i时使用j值1来完成的。一般地，我们使用按栈位置索引的向量last_soln，它记录了上次求解时使用的j值。这种方法确保了我们按照代价最小的顺序尝试插入操作并且决不会重复已验证失败的插入。

我们还必须为E表寻找一种有效的表示方式。很显然，我们不可能预先计算E[B][b][i]的所有可能值。为使简单的修复更快一些，我们可以预先计算E[B][b][0]的值。如果考虑空间因素，这些值也只能在需要的时候才被计算。

在例程find_validated_insert()里，所有E表的引用有相同的第二个成员，即参数validation_prefix的首符号。最初，我们仅需要E[X][a][0]，其中a = validation_prefix[0])。这是一个由非终结符X索引的向量。这个向量可以从E[B][b][0]中抽取出来（前提条件是我们已预先计算过E[B][b][0]）。除此之外，还可以从S表的值和文法的产生式来计算E[X][a][0]（参见练习7b）。这种计算很快，只需片刻时间。如果有必要，我们可以逐步计算E[X][a][1]、E[X][a][2]、等等，直到发现验证正确的插入为止（参见练习19）。

还有一种方法可以计算完整的E[X][a][i]向量。随着插入代价变得愈加昂贵，它们就愈加不受欢迎，而且修复代价是否最小也变得不那么重要了。因此，我们可以考虑例程compute_E()，它能产生原始E表的值并将它们存储在E_table中，后面跟着以代价排序的单符号前缀。因为长度为1的E函数值是按需计算的，所以我们可以在原始的E表中将它们替换为?。而长度为1的条目相当常见（可以占到Pascal文法的20%），所以这是一个了不起的节约。假设我们全局定义了：

```
const terminal sorted_terminals[NUMBER_OF_TERMINALS];
```

而且将其初始化为按插入代价排序的终结符的集合。

我们现在有如图17-20所示的例程。使用此例程计算E值，且只使用错误符号之上的一个或两个符号验证，试验人员就将针对Pascal测试集的较差的修复减少到了11%以下。有趣的是，验证的窗口从一个符号增大到两个符号时并没有带来多大的区别（在评估为较差的修复中的改变低于1%）。两符号的验证有时确实能选择更好的修复，但却容易被前后两个错误相距很近的情况所抵消。在这些情况下，较大的窗口会把第二个错误曲解为一个指示而指出正在考虑尝试的修复是不合适的。 709 ~ 710

```
terminal_string compute_E(symbol A, terminal a, int i)
{
  /*
   * Find E[A][a][i], restricting
   * values for i > 0 to one symbol.
   */

  /* Index of last valid prefix found */
  int num_found = -1;
```

图17-20 一个计算E表值的算法

```
        terminal_string prefix = "?";

        if (E_table[A][a] != "?") {
           prefix = E_table[A][a];
           num_found = 0;
        }

        for (j = 0; j < NUMBER_OF_TERMINALS; j++) {
           if (i == num_found)
              return prefix;

           if (ll_validate(SET_OF( A ),
                           sorted_terminals[j] @ a)) {
              /*
               * sorted_terminals[j] is a legal prefix if
               * it and a can be parsed with A as stacktop.
               */
              prefix = sorted_terminals[j];
              num_found++;
           }
        }
        /* All possibilities examined;  */
        /* could not find E[A][a][i]    */
        return "?";
     }
```

图17-20 （续）

总之，FMQ算法的扩展带来了非常显著的修复质量的提升。当然也少不了一些“惩罚”。因为所采
711 用的搜索和验证不像表查询那样简单，所以错误修复相对较慢。然而，速度并非主要因素。该算法的一种简单实现可以一个修复平均低于一秒钟。而更仔细的实现无疑还会提高修复率，无论如何，它都足以胜任交互式或批处理的应用。我们做的最明显的改进是用compute_E()缓存状态信息以便E[A][a][i+1]的调用能从E[A][a][i]停止的地方而不是从头开始计算。

17.2.7 利用LLGen进行错误修复

我们这么详细地讨论FMQ算法的一个原因是我们的LLGen语法分析器的生成器也是一个错误修复的生成器。如果使用了选项errortables，就会生成一个包含插入和删除代价以及S表和E表的文件。根据所用的FMQ的版本不同，E表是可选的。

插入和删除代价被作为输入提供给LLGen。在*terminals一节里，每个记号的定义行可以包含一个整数插入代价和一个整数删除代价，其形式为：

token_name insertion_cost deletion_cost

图17-21给出了语言Micro的词法记号的定义，其中带有插入和删除代价。

如果没有提供有关代价，则假定默认代价为1。这意味着最小代价修复默认为最短长度修复。

修复代价的选择往往因人而异，但通常也不是那么要紧。回想一下，只有已知可能是正确的（或可能被验证确认的）修复才会被算法所选择。修复代价主要是用于在那些可被接受的候选中进行挑选。通常，我们可以更新代价以便调整算法来强制执行一个特定的修复。根据以往的经验，带有较少语义内容的分界符的插入或删除均很便宜；关键字和一些重要的分界符[像(]代价较高。因为通常在一个输入符号
712 周围构造插入要比删除它的情况好一些，所以删除代价通常要比插入代价高一些。有时一个符号（例如if）可以出现在不止一个上下文当中（例如，出现在Ada语言中作为标题和闭合符号）。如果存在着重大差异的插入代价均是合适的，那么可以创建一些截然不同的符号用于代价分析，并且单个终结符将出于分析和修复目的而被保留。

*terminals		
ID	8	15
INTLITERAL	8	15
:=	4	8
,	2	4
;	2	4
+	2	4
−	2	4
(	10	20
)	4	8
begin	7	12
end	7	12
read	7	12
write	7	12

图17-21 语言Micro中词法记号的插入和删除代价

经验表明：任何合理的代价集合加上适度的验证窗口（1～5个符号）一般可以产生高质量的修复。

17.2.8 LR错误恢复

用于LR分析器的错误恢复技术通常是紧急方式技术的变形。这些技术可以单独使用，或作为那些雄心勃勃的错误修复算法失败时可以依靠的方法。

在紧急方式恢复的最简单版本中，我们首先定义安全符号（safe symbol）集合（如;、**end**、$等）。在发现语法错误时，我们向前跳过若干输入符号直到首次出现安全符号。然后，我们从分析栈中弹出若干条目直至我们达到可以读入安全符号的状态。紧接着，我们将重新开始语法分析。

预期的效果是要跳出有错误的语言结构并在下一个（可能正确的）语言结构开始的地方恢复分析。这种处理在像FORTRAN或BASIC那样的未结构化语言中工作得最好，因为在那些语言中，很容易找到下一条语句开始的地方。

为限制在搜索安全符号时被跳过的输入符号的数量，我们可以定义能够标志语句或语句列表开始的首部符号集合（也许是**begin**、**then**和**else**等）。如果我们正在向前跳过若干输入符号而寻找安全符号时遇到一个首部符号，我们就在此首部符号前停下来并标记它。分析即将被重启（以处理那个期望的语句）。在已标记的首部符号被分析器还原后，向前的跳越将重新开始。

一个比较好的紧急方式版本出现在文献James（1972）中。这种方法自动地搜索要跳到的安全符号。

在发现错误时，此算法做以下几步工作：

（1）弹出零个或多个状态直到遇见能读入至少一个非终结符的状态。我们称那个状态为s。

（2）考虑可以通过读入各种非终结符而得到的s可能的后继状态。假设此状态集为$\{s_1, s_2, \cdots s_n\}$。

（3）向前跳过零个或多个符号到达能被$\{s_1, s_2, \cdots s_n\}$中的任一状态读入的符号T。然后压入任何可以读入T的状态并重新开始分析。

（4）如果没有找到这样的符号T，则从栈顶弹出一个状态，重试步骤1。

此算法的基本思想是通过压入某些非终结符的后继状态从而完成它们的识别。随后用跟在那些非终结符后面的终结符来重新开始分析。实际上，我们跳出了一个短语并随后立即重新开始分析。这种方案明显要比紧急方式更通用（且可以自动生成）。它可以从任何语法错误中恢复过来（最坏情况下，它弹栈到开始状态，取开始符号的后继状态，向前一直读到输入结束处的$）。此算法最大的缺点是待完成的短语的选择（即，压入哪一个后继状态）是任意的。不恰当的选择能够导致级联错误。

713

17.2.9 Yacc中的错误恢复

当一个Yacc生成的分析器（见6.6.12节）发现语法错误时，它将调用子程序`yyerror()`并停止分析。（`yyerror()`负责打印语法错误信息。）

Yacc还另外提供了用户可控的错误恢复方法。当我们把LALR (1)的文法规范提交给Yacc时，可以在产生式的右部放置一个特殊的符号error。此符号标记（出现语法错误后）进行错误恢复的理想地点。例如，产生式

```
statement : error ';' ;
```

就表示在出现语法错误后，语法分析可以通过分号的匹配而得到恢复，继而完成语句的识别。当前（出错的）语句的剩余部分将被跳过。

当发生语法错误时，Yacc分析器即进入出错处理阶段。它从分析栈中弹出若干内容直到发现可以移进error的状态。（如果没有找到这样的状态，分析将终止。）接着error被移进，且用当前（引起语法错误的）记号来恢复分析。如果能分析三个连续记号，分析器就离开出错处理阶段并恢复正常的分析工作。否则，它将删除记号直到它能分析三个连续的记号为止。

例如，假设我们正在分析 A := B C; 而且在记号C处发现语法错误。使用刚刚给出的错误恢复产生式，我们将把分析栈弹回到最初看见A时的那个状态。error被移入栈中，分析栈到达一个要求分号的状态。C随后会被删除，这就使得它后面的分号能被移入栈中。假设后面是一条有效的语句，那么分析将被正确地恢复。包含error标记的产生式也可以包含其他的语义例程以允许在错误恢复后的用户干预，诸如重新设置语义处理或发出特殊的出错信息。

error标记使用户可以控制恢复分析的地点——通常，在主要的语言结构（诸如语句、表达式或声明）的后面。例如，给定语言Micro的规范（见图6-32），下面两个产生式

```
statement : error ';' ;
expression : error ;
```

714 将允许在语句或表达式上下文的结尾处进行恢复工作。

Yacc的恢复方法简单易用且可被自动包含在所有Yacc生成的分析器中。它可以单独使用或作为在那些更精细的修复或恢复技术失败时可以依靠的方法。

17.2.10 自动生成的LR修复技术

基于LR方法的分析器所用的最小代价修复技术是由Dion（1982）开发的。此技术和用于LL(1)分析器的FMQ算法类似。即，用来恢复分析的最小代价的插入和删除序列是确定的。但是，Dion的技术要比FMQ方法复杂得多。问题在于，LL(1)分析栈中直接编码着剩余待匹配 的信息，而从LR技术中获取等价的信息却相当困难。特别地，LR分析状态中的那些单独的项目必须要加以分析且与它们在其他各种状态中的前驱链接在一起。最终的结果是追踪所有可能恢复分析的方法以便从中选择尽可能最便宜的修复。

实现Dion技术需要各种各样的数据表，对于典型的程序设计语言，这些数据表通常都需要成百上千个字节。由Dion技术所产生的最小代价修复相当令人满意，尤其是在执行验证的时候。然而，为达到实用的目的，还是应该使用较小的数据表。因此，我们将关注那些基于代价的并和Dion技术一样有效但要求空间更少的修复技术。

基于延拓的LR错误修复

我们现在考虑能够使用延拓来确定可能的修复动作的LR错误修复算法。由Roehrich（1980）引入的*延拓*（continuation）是指在发生语法错误时被插入的终结符串，它使得分析器在重新启动时没有更多的错误。在最坏情况下，可用延拓来替换剩余的输入以完成修复；更多的时候，可以插入某个前缀以使剩余输入的某个后缀能通过分析。作为示例，我们考虑文法G_2，它是一个产生简单表达式的文法：

$S \rightarrow E\$$
$E \rightarrow T \mid E+T$
$T \rightarrow ID \mid (E)$

在分析(ID + (ID ID)) $的时候，我们在到达第三个ID时发现语法错误。任何可以在我们已接受的程序前缀(ID + (ID之后插入以构成有效程序的终结符串都是延拓。LALR(1)分析器仅接受有效的程序前缀，而一个有效的延拓也必定总是存在的。实际上，对于给定的程序前缀，可以存在无穷多个不同的延拓。因此，在我们的例子中，)) $是一个有效的延拓，而)) + ID $和)) + ID + ID $以及其他许多类似的串也均为有效的延拓。在挑选延拓时，我们可能最想要的是最短或代价最小的延拓。

仅使用延拓的错误修复方法可能是不够的，因为在发生语法错误时，我们很少想删除所有的剩余输入。尽管如此，最小代价延拓的前缀常常可以用作修复中的插入。在进行这样的插入以后，我们就可以删除零个或多个输入符号，然后再试着重启分析过程。和LL中错误修复情况一样，我们采用代价尺度作为判断的标准来选择插入和删除的最佳组合，且在接受之前要先行验证这个修复。设以下函数 715

```
boolean lr_validate(stack_of_state parse_stack,
                    terminal_string prefix);
```

是确定prefix能否在由parse_stack所表示的上下文中进行分析的验证例程。

定义在图17-22中的例程choose_validated_insert()调用子程序get_continuation(parse_stack)来获得与parse_stack中的符号相对应的最小代价延拓。choose_validated_insert()检查延拓的前缀，试图寻找可使validation_prefix被接受的最便宜的插入。

```
terminal_string
choose_validated_insert(stack_of_state parse_stack,
                        terminal_string validation_prefix)
{
  /*
   * Find a least-cost terminal or continuation prefix
   * for the validation_prefix.  Assume that globally
   * we have defined an array of terminals sorted by cost:
   */
  extern const terminal sorted_terminals[NUMBER_OF_TERMINALS];

  terminal_string continuation = get_continuation(parse_stack);
  terminal_string insert = "?";

  if (lr_validate(parse_stack, validation_prefix))
     return ""; /* Best insertion is no insertion */

  for (j = 0; j < NUMBER_OF_TERMINALS; j++)
     if (lr_validate(parse_stack,
              sorted_terminals[j] @ validation_prefix)) {
        insert = sorted_terminals[j];
        break;
      }

  /* Check prefixes of length >=2 */
  for (i = 2; i <= length(continuation); i++) {
     if (C(continuation[1 .. i]) >= C(insert))
        return insert;

     if (lr_validate(parse_stack,
              continuation[1 .. i] @ validation_prefix))
        return continuation[1 .. i];
  }

    return insert;
}
```

图17-22 一个计算已验证的LALR插入的算法

延拓通常是代价最小的，这意味着程序中的一些可选的结构是可以忽略的。在我们先前的例子(ID + (ID ID)) $中，与(ID + (ID相对应的最小代价延拓是))$。一个很明显的修复是在第二个ID后面插入

+，但是+并没有出现在))$里。

Roehrich注意到了这个问题并建议包含能在出错符号前被立即插入的特殊分隔符（separator symbol）表。LeBlanc和Mongiovi（1983）通过考虑把所有在语法上合法的终结符作为插入候选而改进了上述方法。因此，choose_validated_insert()首先检查在无需插入的情况下能否分析validation_prefix。接下来，它尝试单符号的插入，从最便宜的到最贵的。最后，它尝试延拓符号串的前缀。

定义在图17-23中的例程validated_lr_repair()将考虑输入符号删除的可能性，它使用choose_validated_insert()来计算相应的插入直到发现一个验证正确的最小代价修复为止。我们总可以找到一些验证正确的修复，即使在最坏情况下也有可能删除所有的剩余输入并插入延拓的全部符号。

```
void validated_lr_repair(terminal_string *ins,
                         int *d, int v)
{
  /*
   * Remaining input = b1 · · · bn
   * Optimal repair is to delete d tokens, then insert
   * string ins.  v is a validation count:  we must
   * validate the  repair on bd+1 · · · bd+1+v.
   */
  terminal_string val_ins;

  *ins = "?";
  *d = 0;
  for (i = 1; i <= length(b); i++) {
     if (D(b[1 .. i-1]) >= C(*ins) + D(b[1 .. *d]))
         break;
         /* No lower cost repair is possible */

     len = min(i+v, length(b));
     val_ins = choose_validated_insert(parse_stack,
                                       b[i .. len]);
     if (C(val_ins) + D(b[1 .. i-1])
               < C(*ins) + D(b[1 .. *d])) {
         /* Better repair found */
         *ins = val_ins;
         *d = i-1;
     }
  }
}
```

图17-23 一个计算已验证的LALR修复的算法

作为示例，我们再来看(ID + (ID ID)) $的修复问题。为简单起见，假设所有的插入和删除代价均为1而且我们要求三符号的验证。首先，validated_lr_repair()考虑零个符号的删除。它以(ID + (ID对应的分析栈和值为ID))的validation_prefix 为参数调用choose_validated_insert()。如前所述，发现代价为1的符号+的插入。接下来，validated_lr_repair()考虑删除第三个ID。由于此代价为1，因此不能带来更便宜的修复。而更多符号的删除也是如此，所以，validated_lr_repair()选择插入+来修复此错误。

在实践中，validated_lr_repair()执行得相当好。在使用Ripley和Druseikis测试集时，有超过
716 75%的修复被评估为优秀。经过仔细地调整修复代价以及采用可适应两个相连错误的验证机制，在所有修复中，可评估为优秀修复的比例增加到81%，而评判为较差修复的比例仅占4%。

同样重要的是，这种修复技术并不昂贵。除了分析表以外，需要存储的只有用于计算延拓的延拓动作表和代价向量。修复的计算速度大约为每秒钟10个，而这种速度也适合于交互式和批处理的
717 应用。

计算延拓

在本节里，我们讨论在发生错误时如何从当时的分析栈中计算延拓。Roehrich的原始公式没有使用代价。因此，我们使用由LeBlanc和Mongiovi（1983）提出的一个扩展。为计算延拓，我们检查分析栈中的每一个状态以确定它所期望的终结符。这表面上看似简单其实却暗含陷阱。

考虑有以下基础项目的分析状态:

$A \rightarrow \alpha . \beta$
$B \rightarrow \delta . \gamma$
...
$Z \rightarrow \zeta . \omega$

这些基础项目表明产生式已被部分匹配。为完成分析，必定要完成它们当中的某一个。一个明显的方法 718
是插入与β或γ，…，或ω匹配的（可能是最小代价的）终结符串。

必须要仔细选择待插入的终结符串。假设我们有基础项目:

$A \rightarrow E . \textbf{end}$
$E \rightarrow E . + ID$

其中，插入**end**的代价比插入+ ID的代价要高一些。于是，看上去插入+ ID会比较适合，但这样会导致无限插入循环。特别地，如果插入的是+ ID，在我们归约E → E + ID且移进E之后，我们将返回到与原来相同的状态，从而又开始强制插入+ ID。正确的延拓算法必须认识到**end**的插入才是合适的，尽管它比+ ID代价高一些。

为确定延拓，我们检查LR分析状态。我们特别关注状态中项目的顺序，把状态看成一个项目的列表而不是项目的集合。如果列表完全有序，那么在列表头部的项目将总被用来确定延拓。

假设我们从文法G开始。和17.2.4节中的情况一样，设C(a)为插入终结符a的代价，S[γ]表示可从γ推导出的最小代价的终结符串。

文法G中能够共享公共左部的产生式将按照它们右部的代价重新排序。即，如果我们有A → α和A → β，且C(S[α]) < C(S[β])，那么A → α将先于A → β。这样做的目的是为了首先考虑最便宜的产生式。

LR的移进和闭包算法必须要进行更新以保留项目间的正确次序，如图17-24所示。

```
configuration_list list_closure(configuration_list L)
{
   configuration_list L' = L;

   do {
      if (B →δ . Aρ ∈ L' for A∈V_n) {
            /* Predict productions with A
               as the left-hand side. */
            Add, in order of definition, all
              configurations of the form A → . γ
              immediately after B →δ . Aρ
      }
      if (Some configurations appear more than once)
          Remove all but its first (earliest) occurrence

   } while (more new configurations can be added);

   return L';
}
```

图17-24 使用列表的LR分析状态闭包

作为示例，重新考虑使用单位插入代价来排序的文法G_2:

S → E$
E → T | E+T
T → ID | (E)

```
list_closure([S → . E$]) = [   S → .E$,
                               E → .T,
                               T → .ID,
                               T → .(E),
                               E → .E+T ]
```

和通常的LR(0)闭包算法不同，list_closure()在添加新项目时执行的是深度优先(depth-first)而不是广度优先(breadth-first)搜索。这样做的目的是为了把通过预测而添加的项目按照代价排序并且与那
719 个首次导致引入它们的项目放在一组中。

为创建初始配置列表L_0，我们预测拓广产生式并求其闭包：L_0 = list_closure ([S → •α$])。

给定配置列表L，我们使用图17-25中的函数list_go_to()计算它在符号X下的后继L'。list_go_to()以通常的方式计算后继项目并在列表中保持各项目之间的正确次序。

```
configuration_list list_go_to(configuration_list l,
                              symbol x)
{
   /* Compute a basis list, L_b: */
   configuration_list L_b = L;

   for (I in L_b) {
       /* Advance the . past the symbol X (if possible) */
       if (I is of the form A →β . Xγ)
               Replace I with A →βX . γ
       else
               Remove I from L;
   }
    return list_closure(L_b);
    /* Add new predictions to L_b via closure operation */
}
```

图17-25 LR分析表的go_to计算

按照上面的定义，项目列表与用于分析目的的项目集等价。其顺序是强制用来便于提取延拓（稍后讨论）。如Roehrich(1980)中所详细描述的那样，当项目列表被视作集合时，我们必须偶尔创建两个或更多不同的项目列表。这将稍微增加相应的CFSM的大小，但不影响分析。在随后的讨论中，我们将特定文法的LR(0)项目列表表示为L_0。

我们定义延拓值的集合CV = P ∪ V_t ∪{Accept}。CV中的产生式p表示那个产生式将被归约；终结符值t表示t将被添加为延拓中的下一个符号。Accept表示一个已完成的延拓。

我们使用函数CA（continuation action，延拓动作）将配置列表映射到延拓动作：

CA: L_0 → CV

CA的定义如下。设L ∈ L_0为配置列表。设l为L中的第一个形式为A → α•aβ或A → γ•的项目——即，第
720 一个预测终结符或完成产生式分析（归约）的项目。

```
if I = A→α . aβ then CA(L) = a;
elsif I = S→δ$ . then CA(L) = Accept;
elsif I = A→γ . then CA(L) = p, where production p is A→γ; end if;
```

作为示例，我们在图17-26中给出了G_1的CFSM，它使用的是项目列表而非项目集合。相应的延拓动作表在图17-27中给出。

我们现在可以在图17-28中定义先前被choose_validated_insert()使用的例程get_continuation()。get_continuation()取LR分析栈为参数并使用延拓动作表CA和go_to表来计算延拓。它的操作方式是追踪CA表、移进终结符和归约产生式直至最终到达一个接受动作。

作为示例，假设我们正在使用文法G_2来分析(ID + (ID ID))$。当发现语法错误时，分析栈中的状态

为0 6 7 3 6 7。例程get_continuation()执行图17-29中的步骤来计算一个延拓。

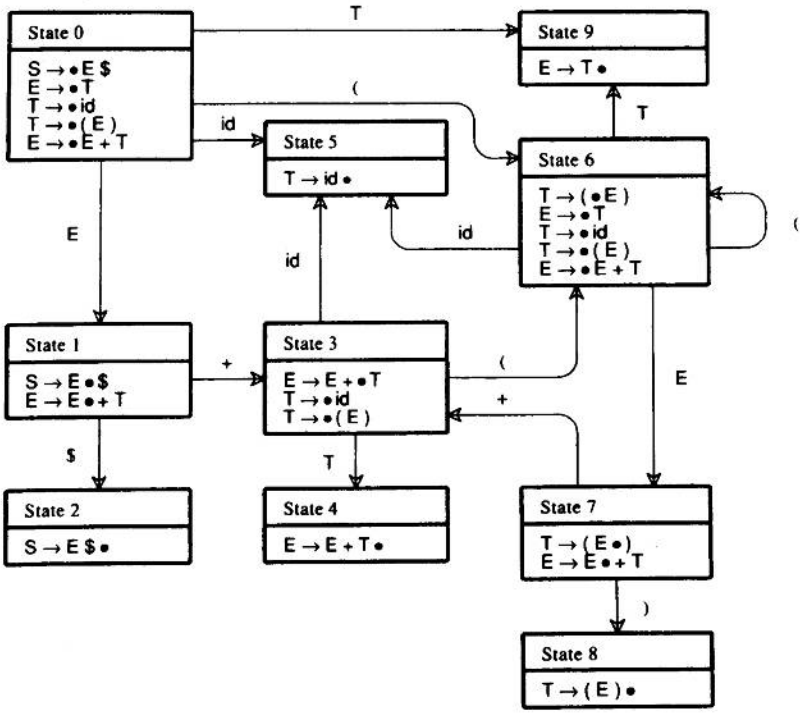


图17-26 使用项目列表的文法G_1的CFSM

State:	0	1	2	3	4	5	6	7	8	9
Continuation Action:	ID	$	Accept	ID	p5	p3	ID	)	p4	p2

图17-27 文法G_1的延拓动作表

```
terminal_string get_continuation(stack_of_state parse_stack)
{
   terminal_string continuation;
   production p;

   continuation = λ;
   while (TRUE) {
     if (CA[top(parse_stack)] == ACCEPT)
        return continuation;
     else if (CA[top(parse_stack)]∈Vt) {
        /* Add to continuation */
        continuation = continuation @ CA[top(parse_stack)];
        push(parse_stack,
             go_to[top(parse_stack)][CA[top(parse_stack)]]);
     } else {
        /* Reduce a production */
        p = CA[top(parse_stack)];
        pop(parse_stack, length(p.rhs));
        push(parse_stack,
             go_to[top(parse_stack)][p.lhs]);
     }
   }
}
```

图17-28 一个计算延拓的算法

Stack	Action	Continuation
0 6 7 3 6 7	Shift)	)
0 6 7 3 6 7 8	Reduce T → (E)	)
0 6 7 3 4	Reduce E → E + T	)
0 6 7	Shift)	))
0 6 7 8	Reduce T → (E)	))
0 9	Reduce E → T	))
0 1	Shift $	))$
0 1 2	Accept	))$

图17-29 追踪例程get_continuation()

计算出的值))$ 很明显是一个有效的延拓而且实际上也是代价最小的。我们可以很容易地证明例程get_continuation()总是产生有效的延拓。在正常情况下，那些生成的延拓也是代价最小的，尽管这一点不能总是得到保证（参见练习13）。

有效性很容易证明，因为例程get_continuation()只是简单地仿真一个分析器：移入延拓符号、执行归约直到到达接受动作。惟一可能出问题的是例程get_continuation()有可能不终止。揭示算法终止的奥秘在于分析列表中项目的次序。在闭包计算中，一个被预测的项目跟在首次引入它的那个项目的后面被添加进来。这就意味着在一个项目完成分析、归约并移入它的左部（非终结符）以后，它最初的预测者将位于分析列表的表头并将在下一次完成分析。实际上，我们可以逆向工作，首先完成某个项目然后才是它最初的预测者。这种处理避免了循环且因此可以确保计算终止。

721
~
722

17.2.11 利用LALRGen进行错误修复

我们这么详细地讨论LR错误修复算法的一个原因是我们的LALRGen语法分析器的生成器也是一个错误修复的生成器。如果使用了选项errortables，那么就会生成一个包含插入和删除代价以及延拓动作表的文件。

723

提交给LALRGen的插入和删除代价的格式与在LLGen中使用的格式相同（参见17.2.7节）。在*terminals一节里，每个记号定义行可以包含一个整数插入代价和一个整数删除代价，其形式为：

token_name insertion_cost deletion_cost

如果没有提供有关代价，则假定默认代价值为1。

17.2.12 其他LR错误修复技术

研究人员已经提出许多其他的LR错误修复技术，其中包括由Druseikis和Ripley（1976）、Mickunas和Modry（1978）、Pennello和DeRemer（1978）等人提出的技术。我们这里集中讨论几种最近的方法，它们被证实是有效且效率高的。

由Graham、Haley和Joy（1979）提出的技术可以用在Berkeley Pascal系统中。它利用由Yacc产生的分析表，并在其中添加了强有力的错误修复成分。GHJ技术首先尝试包括插入、删除或替换单个记号等简单的错误修复。它所考虑的每一种修复均被验证，所使用窗口的大小为5个记号。验证将包括一个语义成分，因为在验证过程中仍然可以进行各种类型检查。任何一个语法或语义上的错误都会使修复变得不合格。

每种修复都被赋予一个代价值，那些仅能通过部分窗口验证的修复将受到惩罚。如果上一次分析器的动作是移进，那么被移入的终结符可以被移出，并且包含单个插入、删除或替换的修复将被再次评估。

在当前记号（也可能包括前一个记号）的所有单符号修复都被评估之后，我们将选择其代价低于一个阈值的最便宜的修复。如果没有这样的修复，我们要做特殊的检查以查看是否能将单个终结符移入到当前状态中。如果可以，那么即使可能很快就会检测到另一个错误（或验证机制将此修复评估成是可以

接受的修复），此终结符也将被插入。我们需要谨慎处理以避免插入循环，因此连续的单移入的插入将被禁止。

如果单移入的插入不合适，则进行二级错误恢复。这其实就是（使用错误标记的）标准的Yacc恢复机制，并附带有Wirth的首部符号（见17.2.2节）以防止对输入符号的过度删除。特别的考虑将用于区别声明部分和语句部分。如果有必要，可以插入**begin**来打开语句部分，或者弹出若干个状态以重新打开声明部分。

GHJ技术也利用了出错产生式来处理那些超出普通错误修复范围的情况。例如，在Pascal语言里，声明部分（标号、常量、类型、变量和子程序）必须有正确的次序。不正确的出现顺序是一种较常见的错误；出错产生式将接受那些出现在错误位置上的声明。

在实践中，我们发现GHJ技术是快速、紧凑和有效的。为了包含错误标记条目，分析表仅增大约10%。修复速度为每个修复若干分之一秒，且修复质量非常好，针对Pascal测试集的修复中，有超过80%的修复被评估为良好或优秀。

724

这种方法的主要缺点在于，它包含了许多针对特定语言的组合使用的试探方法。修复程序是手工编写而不是自动生成的，这样就不容易将它移植到新的语言中。尽管如此，GHJ方法表明：在产品级的编译器中，我们可以获得那些有效且高效的错误修复。它为更自动的技术制定了参照或赶超的标准。

另一个值得关注的错误修复技术是Burke和Fisher（1982）提出的为Ada语言设计的错误修复技术。和GHJ技术的情况一样，这种技术也使用了两级修复，通过验证去除掉一些可能的修复。大多数完整Ada语言的编译器都不是一遍的；相反，它们首先创建并分析称为Diana树的中间表示，稍后才生成代码（参见第14章）。因此，BF技术不限于保存一个已经被接受的输入前缀，而且它还包括在已经分析的输入上的*回溯移动*（backward move）。

在*主要的修复阶段*，将尝试在当前（非法）记号处进行*简单修复*（simple repair）。简单修复包括*记号合并*（token merge）、单个记号插入、记号替换、*范围恢复*（scope recovery）或单个记号删除等。记号合并是将前一个记号和当前记号，或将当前记号和下一个记号拼接为一个记号（例如，：和 = 拼接为:=）。记号替换仅限于可能的拼写错误（例如，begim替换为**begin**）。范围恢复是插入一个*范围闭合符号*，如）或**end if**。可以通过检查验证能进行多远（最多到25个记号）来评估每一种可能的修复。验证最远的修复，只要它能到达最小距离（通常是3个记号）就会被选中。

如果没有找到满意的修复，这个算法就从分析栈中删除一个状态并把此状态的条目符号拼接到输入串上。实际上，此算法通过删除一个状态对应的已接受的符号而向左移动了一个符号（终结符或非终结符）。在此之后，将再次尝试并评估简单修复（不包括记号合并）。此过程将一直重复直至找到一个可接受的修复或直到算法试图回溯越过范围的开始符号（例如，**begin**或**package**）为止。此时，进入*次要修复阶段*。

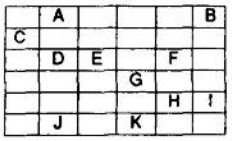
为开始次要修复阶段，分析栈被重置为在首次发现错误时就已存在的那个配置。我们从栈中弹出若干条目以寻找可以移入当前输入记号的状态（当前记号可能用必要的范围闭合符号作为前缀）。如果验证接受了至少5个记号，分析即可恢复。否则，我们将删除当前输入记号并再次弹出状态直到输入前缀可被满意地验证为止。删除将持续到发现已验证的恢复或到达文件末尾为止。对于后者，恢复将作为特殊情况而被强制执行。

研究人员已发现，BF技术可带来优秀的性能，它在Pascal测试集中90%以上的修复被评价为良好或优秀。它的速度还可以接受，约为每秒钟一个修复。

725

练习

1. 考虑以下的稀疏数组：



说明如何使用压缩行技术和双重偏移技术来表示这个数组。

2. 附录F中讨论的数组压缩工具使用了最优递减的方法来构造双重偏移数组表示。即，数组行将按照它们密度递减的顺序而被融和在一起。此外，当某一行被融入到V表中时，该行的放置将使V表被尽可能少地扩展。

   请建议另一种与最优递减法不同的方法，使得数组行可以被融和以形成双重偏移数组表示。在什么情况下，你的方法优于最优递减法?

3. 在某些情况下，数组包含了两行或更多行相同的行。解释如何扩展17.1节中的表压缩技术以利用稀疏数组中的相同行。

4. 通常，Pascal语言中的**if-then**和**if-then-else**语句可由以下形式的产生式生成:

```
<stmt list> → <stmt>
<stmt list> → <stmt list> ; <stmt>
<stmt>      → if <expr> then <stmt>
<stmt>      → if <expr> then <stmt> else <stmt>
```

   Pascal中一个常见语法错误是，“; **else**”问题——“;”后面直接跟着**else**。令人惊讶的是，这个错误难以修复。明显的解决方案是如果“;”后面直接跟着**else**，就忽略掉这个“;”。然而，在一遍Pascal编译器中，分析在**then**分支后的一个“;”将导致**if-then**语句的识别，而**else**则看似新语句的开头。

   在这个文法中添加出错产生式，以便 ; **else**序列被处理为和**else**等价。要保证更新过的文法仍是LALR(1)的。

726

5. 给2.4节里的语言Micro的语法分析过程添加`check_input()`调用。说明如何处理以下语法错误:

   (a) **begin** ID := 1 **end** $

   (b) **begin** ID := **end** $

   (c) **begin end** $

   (d) $

6. 假设我们正使用图5-5中的LL(1)分析表来分析语言Micro的程序，而且使用17.2.3节中的`ll_recovery()`例程来处理语法错误。请说明下面的语法错误是如何处理的:

   (a) **begin** ID := 1 ; ; **end** $

   (b) **begin** ID := 1 **end** $

   (c) **begin** ID := **end** $

   (d) **begin end** $

   (e) $

7. (a) 给出使用CFG和插入代价向量来计算17.2.4节中S表的算法。(*提示*: 使用可以检查每个产生式右部的迭代算法。)

   (b) 推广你在（a）中定义的算法来计算E表的值。

8. 考虑下面的文法:

```
<program>          → <statement list>
<statement list>   → <statement> <statement tail>
<statement tail>   → <statement list>
<statement tail>   → λ
```

```
<statement>        → ID := <expression> ;
<statement>        → if <expression> then <statement list> fi ;
<statement>        → null ;
<expression>       → ID <expression tail>
<expression tail>  → + ID <expression tail>
<expression tail>  → λ
```

给这个文法中的每一个终结符提供插入和删除代价。使用这些代价和你在练习7中定义的算法来计算这个文法的S表和E表。 727

9. 使用练习8中的文法和错误修复表，给出针对以下每个输入在采用图17-11中的例程find_insert()时所计算的修复：

   (a) ID := ID ID := ID ;

   (b) ID := ID ID + ID ;

   (c) ID := fi ;

   (d) ID then null; fi

   (e) if ;

   (f) then ;

   (g) fi ;

   在使用图17-15中的例程ll_repair()时，如果允许删除，那么你将得到什么样的修复？

   在使用ll_repair()计算得到的修复中，如果要求五符号的验证，那么将有哪个修复被拒绝？

10. 把下面的文法改写成Yacc所要求的格式，且添加error记号以指定错误恢复：

```
<program>          → <statement list>
<statement list>   → <statement list> <statement>
<statement list>   → <statement>
<statement>        → ID := <expression> ;
<statement>        → if <expression> then <statement list> fi ;
<statement>        → null ;
<expression>       → <expression> + ID
<expression>       → ID
```

   练习9中的错误程序将得到怎样的处理？

11. 使用17.2.10节中的技术，为练习10中的文法创建CFSM和延拓动作表。例程get_continuation()将为练习9中的每个错误程序计算什么样的延拓？

12. 为练习10中文法的每个终结符建议相应的插入和删除代价。使用在练习11中计算出的CFSM和延拓动作表，并假设采用三符号的验证，那么图17-23中的例程validated_lr_repair()将为练习9中的每个错误程序计算什么样的修复？

13. 证明由图17-28中的例程get_continuation()计算的延拓并不总是最小代价的。

14. 如果我们同时检查语义和语法的有效性，就可以改进错误修复的验证。大致描述我们应当如何设计语义例程，以便它们既可以筛选错误修复，又可以做完整的语义处理。 728

15. 设计17.2.1节中的LL(1)缓冲技术是用来处理任意LL(1)文法的。LL(1)文法常常具有这样的特征：可推出λ的非终结符都可以直接那么做。即，如果$A \Rightarrow^* \lambda$，则可以有$A \rightarrow \lambda$。如果所有λ的推导都是直接，则简要描述一种撤销为非法超前搜索符号而进行的分析器动作的较简单方法。

16. 假设我们正在分析输入x。设计find_insert()的一个版本（见图17-11），使得即使分析栈的深度达到O(|x|)时，该版本的find_insert()的O(|x|)次调用也仅需要O(|x|)的时间。

17. 证明：图17-15中的ll_repair()算法可以计算局部最优的错误修复——即，它所计算的插入串y和删除计数i满足：

$$\min_{0 \le i \le m} \min_{y \in V_t^*} \{D(b_1 \cdots b_i)+C(y) \mid xyb_{i+1} \cdots \in L(G)\}$$

其中，输入为$xb_1 \cdots b_m$。

18. 当发现语法错误时，我们通常不撤销分析动作，因为语义动作是很难或不可能撤销的。由于语义动作是由动作符号初始化的，因此将分析动作撤销至上一次处理的动作符号应该是可能的。这将使修复动作的范围更广且无需撤销语义动作。简要描述应如何修正图5-11中标准的LL(1)驱动程序，以便在发现语法错误时将分析撤销至上一次处理的动作符号。

19. 设a为某个特别的错误符号。假设我们希望使用图17-19中的例程`find_validated_insert()`而且我们已经计算了向量E[X][a][0]、E[X][a][1]、…、E[X][a][i]。解释如何使用这些向量、S表以及正在分析的文法来计算E[X][a][i+1]。

20. 衡量错误修复算法的有效性的一种方法是，取一个正确的程序并有组织地将它转变为一系列在语法上非法的程序。例如，如果程序包含n个记号，那么通过删除一个记号即可获得n个“变异”程序。类似地，通过在每个记号的最近左邻处插入一个随机记号即可得到另外n个“变异”程序。只有少量的“变异”程序在语法上是合法的且可以被忽略。所有其他的“变异”程序虽然在语法上是非法的但可以通过插入或删除单个记号而被修复。

    我们可以通过有多少“变异”程序被修复为原始程序来评估一个错误修复算法。使用这种“变异”分析方法来自动评估你喜欢的错误修复算法。

729

附录A Ada/CS语言定义

A.1 简介

Ada/CS是计算机科学中的Ada子集（a Computer Science subset of Ada），用来说明程序设计语言及其编译器设计和实现中的概念。本附录有两个目的。一是提供语言定义，用于使用本书授课的课程中的编译器实现设计。二是作为本书第10～13章所讨论的语言特性的手册，以及作为本书程序设计示例中所使用的伪代码的基础。

A.2 词法因素

标准字符集是依赖于实现的，但它必须包含表示Ada/CS词法记号所需的特殊符号。注释由“--”（双减号）和行结束符界定。在能出现行结束符的任何地方都可以有注释，它们被词法分析器忽略。在除字符串外的任何词法记号中，大小写字母等价。空白符（空格、制表符和行结束符）要么作为单独的词法记号，要么被忽略。空白符不能出现在除字符串常量（见下）之外的任何词法记号中。行结束符不能在任何词法记号中出现。行的最大允许长度是由实现定义的。

A.3 词法记号

令letter = 'A'.. 'Z', 'a' .. 'z'; digit = '0' .. '9';

(1) 下列为关键字。它们均被保留。

abs	access	all	and	array	begin
body	case	constant	declare	else	elsif
end	exception	exit	for	function	if
in	is	loop	mod	not	null
of	or	others	out	package	pragma
private	procedure	raise	range	record	return
reverse	subtype	then	type	use	when
while					

(2) 文字常量是整数、浮点数或字符串。

整数文字常量仅包含数字和下划线，且由数字开头。下划线可以出现在任意两个数字之间；它用来将数字分组以增加可读性。它不影响所表示的实际值。例如，100万可以写成1_000_000或1000000，甚至1_0_0_0_0_0_0。

浮点数文字常量中在小数点前面和后面都必须至少有一个数字。下划线允许出现在浮点数文字常量中。浮点数也可以使用带字母E或e的科学计数表示法。下列都是有效的浮点数文字常量：

```
1.0  1.0E-5  1_000_000e4
```

字符串文字常量以双引号（"）开头和结尾，并包含可打印字符（printable character）的任意序列。不可打印字符必须通过Char属性（见下）创建。如果一个双引号出现在字符串常量中，它必须被重复（例如，""""Help!"" he cried"）。

整数值的允许范围和Float值的范围及精度是由实现定义的。

(3) 利用正则表达式词法记号，其中：‘,’表示并集，‘.’表示集合的积（连接），‘*’表示Kleene闭包，λ表示空字符串，Not表示补集，-表示差集，文字常量由引号（"）界定，而括号用于分组，则以上定义可以更为精确：

```
Literal = IntLiteral , FloatLiteral , StrLiteral

IntLiteral = digit . (digit , '_')*

FloatLiteral =
  IntLiteral . '.' . IntLiteral ,
  IntLiteral . ('E','e') . (λ, '+', '-') . IntLiteral ,
  IntLiteral . '.' . IntLiteral . ('E','e') . (λ, '+', '-') . IntLiteral

731
StrLiteral = '"' . (Not('"') , '"' . '"')* . '"'
```

(4) 标识符是以一个字母开头的字母、数字和下划线的字符串。

```
Identifier = (letter . (letter , digit , '_')*) - Reserved
```

对于标识符的长度没有限制（除了它必须被放在同一行中以外）。

(5) 提供下列各类运算符：

MultOps	*, /, **mod**
UnaryOps	**abs, not**
PlusOps	+, -
RelOps	=, /=, <, <=, >, >=
LogicalOps	**and, or, and then, or else**

(6) 下列为其余的运算符和分隔符（$表示文件结束符）：

```
Delim = '"' , '.' , ':' , ';' , ',' , '.' , '|' , '&' , '(' , ')' , '**' , ':=' , '<>' , '=>' , '$'
```

相邻的标识符、保留字和数值文字常量必须由空白符（或注释）分隔。该规则防止词法记号扫描方式的二义性（即，begina是一个标识符，而不是**begin**后面跟着a）。

A.4 Pragma

形如**pragma** ID; 的*pragma*可以出现在包之间，以及声明或语句可以出现的任何地方。pragma是一个编译器指示。pragma的效果以及合法命令ID的集合是依赖于实现的。典型的pragma命令包括：ListOn、Optimize、Pack和Inline。

A.5 类型

Ada/CS包含四种在Standard包中预定义的类型（Standard包中含有创建标准环境所需的所有声明）。

Boolean: 与Pascal和Ada中的一样

Float: 像Pascal中的Real

732 Integer: 与Pascal和Ada中的一样

String: 一个变长字符串（与Snobol IV中的一样）。最大可能的字符串长度通常由实现因素决定（但原则上是无限的）。应当避免较小的串长限制（如256或1024）。这个字符串的定义比大多数Ada实现中的定义更通用。

这些类型名不是保留字，因而可以在局部名字作用域中被重新定义。

Ada/CS包含四种创建新类型的*类型生成器*（type generator）：

(1) 枚举（与Pascal中的一样）形如

```
type ID is (ID, ID, . . .);
```

该生成器创建一个新的枚举类型，括号中的那些ID作为该类型的常量。在类型定义列表中ID的顺序确定了值的全序关系，列表中第一个ID的值最小，最后一个ID的值最大。类型Integer和Boolean都被认为是枚举类型（type Boolean is (False, True);）。Float不是枚举类型。Float和枚举类型被称为标量（scalar）类型。Integer和Float，但不包括枚举类型，被称为算术（arithmetic）类型，在算术类型上可以应用预定义的算术运算符（如“+”和“*”）。

(2) 数组定义为

```
type ID is array( <bounds list> ) of <component type name>;
```

在<bounds list>中，（由多个“,”分隔的）单独的界限形式为<expr> .. <expr>或<type name>。区间表达式可以包含变量。<type name>必须是一个（可能受限的）枚举类型的名字。如果任意数组边界包含变量，则数组是动态的（dynamic）。在进入包含适当数组、枚举和子类型定义的作用域时对边界进行求值（并固定）。数组的成员类型可以是包括**array**在内的任意类型。

<bounds list>的元素也可以拥有<type name> **range** <>这样的形式，它定义一个非约束数组类型（unconstrained array type），其中下标边界是未指定的。这种类型对于指定作为参数传递给子程序的数组类型尤其有用。（见子程序一节对非约束数组类型的进一步讨论。）

例如:

```
type Arr1 is array (1..9) of Integer;
type Arr2 is array (Boolean) of Arr1;
type Matrix is array (Integer range <>, Integer range <>) of Float;
```

733

(3) 记录的形式为

```
type ID is
  record
    <component list>
  end record;
```

带有变体的记录形式为

```
type ID is
  record
    <component list>
    <variant part>
  end record;
```

例如

```
type R1 is
  record
    I, J : Float;
    C  : Integer;
  end record;

type R2 is
  record
    C : String;
    E : Integer;
    F : R1;
  end record;

type R3 is
  record
    K : Integer;
    case T : Integer is
      when 1 => A : Integer;
      when 2 => B : Float;
```

```
        when 3 => C : Integer;
                  D : Float;
    end case;
  end record;
```

记录中的一个成员声明在语法上与变量声明相同（见下）。记录中的所有数组都必须有常量边界。记录中的域名局部于当前记录并且在当前记录中必须是惟一的。

变体记录允许在一个记录类型中指定那些可选择的成员。每个变体描述与（Ada中）判别式或
734 （Pascal和Ada/CS中）标签域的一个（或多个）特定值相对应的记录成员。为简化关于编译变体记录技术的介绍，Ada/CS的变体记录特性比Ada的变体记录特性要简单一些。正如我们所注意到的，它是基于标签域的使用，这一点就像在Pascal中一样。此外，每个变体可以仅有一个单独的标签（而不是一系列标签）。

没有域的记录（仅作为占位符）的声明如下:

```
type R3 is
  record
    null;
  end record;
```

(4) 访问类型的形式为

```
type ID is access <object type name> ;
```

在Ada/CS中动态分配的对象由访问类型引用，这些访问类型等价于其他语言中的指针或引用类型。在Ada/CS中，对整个动态对象的引用由访问对象名后面的一个**.all**后缀表示（例如**A.all**）。然而，对被引用对象的成员的引用不需要特殊的语法表示。因此，A.D可以表示对记录A的域D的引用，也可以是对访问对象A所指向的记录的域D的引用。

在Ada/CS中（像在Ada中一样）包含不完整的类型声明，用于处理访问类型声明中的前向引用，比如在定义递归类型时就是这样。例如

```
type Cell;                    ——不完整的类型声明
type Link is access Cell;     ——必须已经声明Cell

type Cell is
  record
    Value : Integer;
    Succ : Link;
    Pred : Link;
  end record;
```

A.6 子类型

子界类型定义为

```
subtype ID is <type name> <constraint>
```

<constraint>的一种形式为**range** Lower .. Upper，称为区间约束。不像Pascal中那样，Lower和Upper可以是（产生相同枚举类型的）表达式。这些表达式中可以含有变量，因此可能无法在编译时求
735 值。<constraint>是可选的；如果它被省略，则子类型就是原类型的一个简单的重命名。<type name>也是可选的；如果它被省略，则它是Lower到Upper区间界限的类型。

子类型是限制在区间约束所规定的约束内的原类型（或基类型）。因此，仅有在Lower和Upper界限约束之间的值可以被赋予这种子类型的对象。可以声明子类型的子类型，只要约束区间是适当的（例如，子类型不能包含其父类型所不允许的任何值）。

约束的第二种形式是下标约束：(<range list>)。下标约束只能应用于非约束数组类型。像数组类型定义中的界限一样，区间约束中的区间表达式也可以包含变量。变量不能声明为非约束数组类型。它必

须使用约束类型或数组子类型，以便变量的大小可由适当的声明定义。

理解子类型并不创建新类型这一点很重要。子类型名代表着约束和类型的组合，其中的约束规定了类型中的哪些值可以被赋给声明为该子类型的变量的进一步的限制。

A.7 变量声明

变量由以下形式的声明引入：

ID, ID, . . ., ID : <type> := <expr>;

其中:= <expr>是可选的初始化部分。

例如：

```
subtype Week is Integer range 1 .. 7;
A, B : Week := 1;
```

A.8 变量表示

记录的域通过在记录标识符后添加限定来表示。例如，如果使用前面的记录R2的定义得到

A,B : **array**(1 .. 10) **of** R2,

则下列所有名字都是合法的

A(1).C, A(1).F.C, B(I).F

但下列这些名字是非法的

B(1).I, A.E, A.E(1), F.J.C

736

多维数组不能是部分索引的。例如，给定

type AR **is array**(Boolean,1..10) **of** Float; C : AR;

C(True, 3)是合法的，但C(True)(3)和C(False)是非法的。

与Pascal不同，多维数组不是数组的数组（但数组的数组也是允许的）。函数可以返回数组和记录，而且它们可以被限定。因此，如果F和G是函数，则F(1).A和G(2)(1,2)都是合法的。

A.9 有名常量

标识符可以被给定初值并被声明为常量。其语法非常接近用于变量声明的语法：

ID, ID, . . ., ID : **constant** <type> := <expr>;

如果<expr>仅包含文字常量、显式常量和预定义的运算符和函数，则该常量是显式的（manifest）并且可以在编译时求值。否则，该常量（实际上）是一个只读对象，其值在运行时确定但不能在初始化之后被改变。在定义之后，显式常量可以用于（相同类型的）文字常量能出现的任何地方。

A.10 运算符和表达式

运算符（以优先级的降序排列）有：

(1) 域限定符（'.'）和下标（'('和')'）

(2) 指数和一元运算符：**, **abs**, **not**

(3) 乘法运算符：*, /, **mod**

(4) 一元加和一元减：+, −

(5) 加运算符：+, −, &

(6) 关系运算符：=, /=, <, <=, >, >=

(7) 逻辑运算符：**and, or, and then, or else**

相同优先级的运算符从左到右求值。除**和关系运算符（它们根本不可结合）之外，所有运算符
737 都是左结合的。因此，X ** Y ** 2是非法的（尽管(X ** Y) ** 2和X ** (Y ** 2)都是合法的）。

A.11 运算符描述

(1) 域限定符（略）⊖。

(2) 指数和一元运算符。

如果A是整数或实数，且B是非负整数，则A ** B有定义。特别地，A ** 0等于1或1.0（依赖于A是整数还是实数）。对于B > 0，A ** B等于A*A*...*A（B个A）。

abs是绝对值运算符；**not**是布尔求反。

(3) 乘法运算符。

运算符*是乘；运算符/是除。两个操作数必须是浮点数或整数；结果类型与操作数类型相同。（回想一下，整型的约束子类型也是整型。）

mod运算符仅应用于整数。等式

```
A = (A/B)*B + (A mod B)
```

总是成立。(A **mod** B)和B有相同的符号，并且其绝对值小于B的绝对值。

(4) 一元加和一元减。

这些运算符定义于所有算术类型之上，而且总是返回操作数的类型。+是恒等运算符；-是求补运算符。

(5) 加运算符。

运算符+是加；运算符-是减。两个操作数必须是浮点数或整数；结果类型与操作数类型相同。&运算符是字符串连接。

(6) 关系运算符。

对于运算符<、<=、>、>=、=、/=，两个操作数的类型必须相同；结果是布尔型。相等和不相等(=和/=)对于所有类型均有定义。其余运算符针对标量类型和字符串有定义。对于字符串，比较按照字典序进行。

(7) 逻辑运算符。

and是布尔与；**or**是布尔或。操作数必须是布尔型。两个操作数总是被求值，因而**and**和**or**符合交换律。

运算符**and then**是布尔条件与；运算符**or else**是布尔条件或。操作数必须是布尔型。对于这两个运算符，左边的操作数先求值。右边的操作数仅在必要时才求值。这两个运算符都不符合交换律。

A.12 赋值相容性

值只能被赋给与之具有相同基类型的对象。如果目标类型是约束子类型，则赋给它的任何值都必须
738 满足其约束。否则，子类型将与其基类型以及相同基类型的其他子类型赋值相容。两种不同的类型定义，即使结构上相同，也被认为是不同的。该规则被称为类型的名字等价。

例如

⊖ 原书这里缺少第一点。——译者注

```
type T1 is array(1..10) of Float;
type T2 is array(1..10) of Float;
A, B: T1; C, D: T2;
```

A:=B和C:=D是合法的，但A:=C不合法。

A.13 空语句

空语句表示为 **null**。

A.14 赋值语句

赋值与在Algol 60或Pascal中类似，假定赋值的左边和右边相容（如上面所定义）。任意类型的相容对象都可以被赋值。求值顺序由实现定义。

A.15 块

块结构，（变量、有名常量、类型和子程序等）名字的作用域和动态分配与Algol 60中相同。块结构的形式为

```
declare
   Type, variable, constant and subprogram declarations
begin
   Statement list
exception
   Exception handler declarations
end;
```

如果没有声明，则关键字**declare**可以被省略。如果没有异常处理程序，关键字**exception**可以被省略。块可以出现在语句能出现的任何地方（像Algol 60中一样）。块必须以形如“ID:”的块标识符作为前缀；同样的标识符必须跟在结尾的**end**之后。

739

例如：

```
B1:  begin
        ...
     end B1;
```

A.16 if语句

if语句的形式为

```
if boolean expression then
   sequence of statements
elsif boolean expression then
   sequence of statements
   ...
elsif boolean expression then
   sequence of statements
else
   sequence of statements
end if
```

else子句和**elsif**子句可以被省略。条件按顺序进行求值，直到其中一个为真（将**else**看成**elsif** True **then**）。随后执行相应的语句序列。如果没有任何条件为True，则不执行其中任何一个语句序列。

A.17 loop语句

loop语句有三种形式。第一种形式为

```
loop
  sequence of statements
end loop;
```

740

这是一个“无限循环”，但**exit**可用于终止迭代（见下）。

第二种循环形式是传统的**while loop**：

```
while <boolean expr> loop
  sequence of statements
end loop;
```

第三种形式基本上是一个像Pascal一样的**for loop**：

```
for ID in <range> loop
  sequence of statements
end loop;

for ID in reverse <range> loop
  sequence of statements
end loop;
```

<range>是显式区间对（即Lower .. Upper）或者是一个枚举类型或子类型的名字。在后一种情况下，Lower和Upper从类型或子类型的限制或约束中提取。ID在循环体中*隐式声明*，因此它在循环体外不可用。ID的类型由<range>表达式决定。

在第一种形式中，迭代自ID被置为Lower时开始（如果发生迭代的话），并在ID = Upper之后终止。<range>表达式仅在进入循环时被计算一次。零次迭代也是有可能的（如果初始时ID > Upper）。在每次迭代的末尾，ID接受<range>序列中的下一个值。

在第二种形式中，迭代自ID被置为Upper时开始（如果发生迭代的话），并在ID = Lower之后终止。<range>表达式仅在进入循环时被计算一次。零次迭代也是有可能的（如果初始时ID < Lower）。在每次迭代的末尾，ID接受<range>序列中以*逆序*遍历的下一个值。

需要将**for** ID（即循环变量）视为只读的。它不能被**赋值**，不能用作**out**参数，等等。像块一样，（任意形式的）循环可以加标号，并在循环*结尾*处加上相同的标号。

A.18 exit语句

Ada/CS没有**goto**语句。但它有形式为

```
exit;
```

或者

```
exit ID;
```

的**exit**语句，作为**goto**语句的受限形式。不带标号的**exit**离开包含该**exit**语句的**最内层循环结构**。带标号的**exit**语句离开标记有相应标识符的循环。在这两种情况下，所退出的循环都必须在包含**exit**语句的同一子程序或包内。**exit**不意味着从子程序中返回。

741

一个含条件的**exit**可以由**when**子句表示，其形式为

```
exit ID when <bool expr>;
```

exit仅在布尔表达式为True时执行。（也可以使用**if**语句，但我们认为**exit-when**语法的可读性更好。）

A.19 case语句

case语句的形式为

```
case <expr> is
  when <choice> | . . . | <choice> => sequence of statements;
  when <choice> | . . . | <choice> => sequence of statements;
    . . .
end case;
```

它计算表达式的值，并执行与表达式的值相匹配的选项所对应的语句序列。每个选项要么是常量表达式，要么是一个像为**for**循环所定义的<range>指示符，但**case**语句中所使用的所有区间的边界必须在编译时确定。最后一条**when**可以使用关键字**others**作为其特有的选项。这表示所有未被其他选项子句涵盖的值。<expr>所有可能的值都必须仅被一个选项涵盖。任意枚举类型或子类型都可以用于**case**语句中，当然，表达式的类型和**when**选项的类型必须匹配。

A.20 Read

在程序设计语言的设计中，一个反复出现的问题是，是将Read和Write作为语句类型还是作为预定义的过程。两种方式都有一些问题。考虑（像Ada那样）将Read和Write作为预定义的过程并不真正令人满意，因为不像普通的过程，我们想让它们取不定数目的参数，而且可能允许非标准的参数语法（例如Pascal的宽度规范）。另一方面，保留Read和Write看起来会阻止用户去扩展这些语句来处理新的数据类型。

在Ada/CS中，我们将基本上把Read和Write看作重载的预定义过程。为允许输入或输出项的列表（Ada并不真正支持），我们将引入对过程调用语法的一个记号上的扩展。如果愿意，你可以想像一个将我们的扩展翻译为标准语法的预处理器。（C语言用这种方式处理#include、#define等。）Ada中的参数 742
由逗号分隔。我们将允许由分号分隔的参数组（parameter group）。每个参数组由预处理器提取并插入一个单独的过程调用。因此Write("Date ="; month; day; year);会被展开成

```
Write("Date ="); Write(month); Write(day); Write(year);
```

该扩展可由所有过程使用，而不仅限于Read和Write。

在Ada/CS中，只能读标量和字符串。即，标准的包中含有接受Integer、Float、Boolean和String值的Read定义。当定义枚举类型时，Read被自动重载以处理该新类型。Read没有被预定义用来处理数组或记录，尽管我们可以为这些类型编写用户定义的例程。

Read能够处理结构类型成员，只要该成员是字符串或标量。输入是以自由格式读入，而字符串值需放在引号中。我们的预处理器强制Read一项一项地处理，因此，举例来说，Read(I; A[I]);是合法的。

标识符Next（预定义枚举类型的一部分）可以出现在读列表中。这将导致立即跳到下一个输入行的开始。

A.21 Write

所有标量值（和表达式）以及字符串都由预定义的Write过程处理。每个标量类型都有足够打印该类型任意值而不需截断的默认输出宽度（由实现决定）。字符串的默认输出宽度是其长度。

在写列表中的任何变量或表达式后面可以跟随表示显式输出宽度的第二个整数参数。

- 对于标量，如果宽度值i足够打印输出值而不需截断，则该值使用i列打印，（如果必要的话）空格被填充在左边。如果宽度不足以打印标量值而不需截断，则该宽度参数将被重载为允许打印该值的最小宽度值。
- 对于字符串，如果宽度值i > 字符串长度，则字符串使用i列打印，左边填充空格。如果i < 字符串长度，则打印最左边的i个字符。如果i < 0，则不打印任何符号。
- 对于浮点数，使用的格式（指数形式，定点十进制等）是与实现相关的，何时打印宽度会导致截

断的定义也与实现相关。

预定义的标识符Next可以出现在输出列表中，Next将导致立即跳到下一个输出行。如果输出行过
743 长而不能打印在一个物理行中，它可以在没有显式Next出现的情况下被截断。

宽度表达式可以被（可选地）包含在参数组中：

```
Write("Date = "; month,2; day,3; year,5);
```

它被展开成

```
Write("Date ="); Write(month,2); Write(day,3); Write(year,5);
```

A.22 子程序

就像在Pascal中一样，子程序可以作为过程或函数使用。过程调用的形式为P(Arg, Arg2,. . .);而无参过程调用的形式为P;(空参数表是不需要的)。

函数调用出现在表达式中；例如A + F(3,A)。无参函数以F表示（看起来很像变量）。两种形式的子程序都可以递归。函数可以返回任意类型（包括记录和数组——甚至动态数组）。

每个子程序的开始是其形参列表。对每个参数，都指定其位置、名字、类型和模式（in、**out**或**in out**，默认是**in**）。如果子程序是一个函数，则函数头以**return** <type name>结束；它定义了由该函数所计算的值的类型。

例如，考虑

```
procedure P(X : Float; Y,Z : in out Integer);
```

过程P中声明了三个形参(X, Y, Z)。所有形参都被认为是局部于子程序体的。所有形参的类型以及返回值类型（如果是函数的话）必须以类型名指定。不可以使用显式的类型生成器或区间规范。

Ada/CS子程序遵循与Pascal相同的作用域规则：每个变量在定义该变量的块中的任意函数、过程或块中自动可见，除非某个内部作用域包含相同变量的另一个声明。Ada/CS不允许前向引用，因此声明的作用域实际上从其定义点延伸到其包含作用域的结尾。

744 过程像块一样，也可以声明局部常量、变量、类型和子程序。子程序体定义的一般结构与块相似：

```
procedure ID ( <formals list>) is
  Type, variable, constant and subprogram declarations
begin
  Statement list
exception
  Exception handler declarations
end;
```

(<formals list>)和**exception**部分都是可选的。

子程序可以接受一个单独的数组类型（其界限是固定的）作为参数，而若能接受整个一类数组类型（其不同仅在于所分配的数组分量数目不同）则通常会更有用。在Ada/CS中，允许定义未指定下标界限的非约束数组。非约束数组的一个例子是

```
type Matrix is array(Integer range <>, Integer range <>) of Float;
```

非约束数组可以用于子程序中来表示一类合法的实参，所有实参都拥有相同的下标和分量类型，但其实际边界不同。因此

```
procedure Invert (M_In: in Matrix; M_Out: out Matrix);
```

可以接受任意大小相容的二维矩阵。

如果一个代表普通数组而不是非约束数组的类型名被用于参数定义，则仅有正确类型的数组会被传

递。因此，参数可以被限制为特定数组类型，或者允许整个一类拥有共同结构的数组。

函数的定义如下：

```
function <name> ( <formals list>) return <type name> is
  Type, variable, constant and subprogram declarations
begin
  Statement list
exception
  Exception handler declarations
end;
```

函数能返回任意类型。仅有**in**参数是允许的，这反映函数不应该有副作用的观点（尽管它们可以改变非局部变量）。函数必须通过执行

```
return <expr>;
```

来返回。其中<expr>是可赋值给函数的类型；该表达式的值作为函数调用的值被返回。过程通过执行"**return**;"而返回。在每个子程序的结尾处都有一个隐式的**return**。

745

函数名（当然）可以是一个标识符。它也可以是（像字符串文字常量一样）以引号引用的下列运算符号之一：**、*、/、**mod**、+、-、**abs**、**not**、&、=、<、<=、>、>=、**and**、**or**。带运算符名的函数必须取一个或两个参数，具体视运算符是一元还是二元（或者既是一元又是二元）而定。'='运算符必须返回一个boolean值。无论何时定义'='，'/='都自动被定义为其非运算。

A.23 参数模式

当调用一个过程时，实参与相应的形参相匹配。

- **in**模式的参数是在调用时被初始化为相应实参的局部常量。实参类型必须可赋值给形参类型，且形参在任意区间上的限定（如果必要）在调用点被检查。如果形参是非约束数组，则实参必须是该形参类型的约束子类型。作为**in**参数传递的实参可以是适当类型的任意表达式。**in**参数是最安全的类型，因为它们不能被改变。这也是它们是默认类型的原因。
- **out**模式的参数是可被赋值但不能读的未初始化局部变量，在返回时，它们的值被赋给相应实参。形参类型必须可赋值给实参类型，而形参在任意区间上的限定（如果必要）在返回点被检查。如果形参是非约束数组，则实参必须是该形参类型的约束子类型。作为**out**参数传递的实参必须是一个（可能带有限定的）变量名，因为它将成为赋值的目标。
- **in out**模式的参数是在调用点初始化为实参值的局部变量。在返回点，该局部变量的值被赋给相应的实参。形参的类型和实参的类型必须是互相可赋值的。此外，对形参和实参的区间限定（如果必要）在调用点和返回点被检查。如果形参是非约束数组，则实参必须是该形参类型的约束子类型。作为**in out**参数传递的实参必须是一个（可能带有限定的）变量名，因为它将成为赋值的目标。

in、**out**和**in out**参数显然可以通过执行复制操作来实现。事实上，这对标量来说是必需的。对于记录、数组和字符串，复制操作可能是昂贵的。因此，Ada/CS允许这样的参数通过传递地址而不是执行实际的复制来实现。依赖于非标量参数如何实现（复制还是地址）的程序是非法的。

746

A.24 前向引用

在Ada/CS中不允许前向引用。如果一个常量、变量、类型或子程序要被使用，则它必须是已经被定义的。可能发生子程序必须在其被定义前被调用的情况（例如，如果A调用B且B调用A）。由于这个原因，子程序的声明可以与其实际定义分离。为声明一个子程序，只要提供指定其名字、参数和返回值类

型（如果有返回值的话）的子程序头即可。在同一名字作用域的后面，必须提供（包含子程序头的）完整的子程序定义。

A.25 预定义函数和语言属性

Ada/CS包括普通的预定义函数（如Substr），这些函数被包含在标准环境中。Ada/CS也包含语言属性（language attribute）的概念，它提供一个对象或类型的某种性质。属性表达式的形式为Name'Attribute。Name是对象或类型的名字；Attribute是我们所希望的特定属性。因此，T'First给出名字为T的类型或子类型所允许的值区间中的第一个值。

(1) A' Len

在A是一个字符串表达式时有定义。它返回一个代表该字符串中字符数的整数值。

(2) Substr(A,S,L)

A必须是一个字符串表达式，S是正整数，且L是非负整数。它返回A中从位置S开始、长度为L个字符的子串。如果该子串不存在，则出错。A最左端字符的位置为1。

(3) T' Succ(A)

A必须是枚举类型T的一个值。其结果为其直接后继（如果存在的话）。否则，抛出Constraint_Error异常。

(4) T' Pred(A)

A必须是枚举类型T的一个值。其结果为其直接前驱（如果存在的话）。否则，抛出Constraint_Error异常。

(5) I' Char

I必须是一个整数。结果是相应于该整数的（在字符序列中）长度为1的字符串；如果不存在这样的字符，则抛出Constraint_Error异常。

(6) T' Val(A)

A是任意枚举类型的一个值，而T是任意枚举类型和子类型的名字。A以如下方式被转换为枚举类型
747 的相应值：

所有枚举都有一个由该类型值的枚举顺序所确定的序数值。对于除整数之外的枚举类型，该值从0开始。对于整数，值i是其自身的序数值。两个值相对应当且仅当它们拥有相同的序数值。因此，给定**type** Boolean **is** (False, True)和**type** Color **is** (Red, Blue, Yellow, Green)，我们有

```
Integer'Val(False) = 0
Boolean'Val(Blue) = True
```

如果在枚举类型中没有对应于A的值，则抛出Constraint_Error异常。(Boolean' Val(-1)和Boolean' Val(Green)将抛出异常。)

(7) T'First

如果T是一个枚举类型或子类型，T' First是T所允许的区间中的第一个值。如果T是一个数组类型或对象，T' First给出第一个下标的下界；T' First(J)给出第J个下标的下界。

(8) T' Last

如果T是一个枚举类型或子类型，T' Last是T所允许的区间中的最后一个值。如果T是一个数组类型或对象，T' Last给出第一个下标的上界；T' First(J)给出第J个下标的上界。

(9) T(E)

如果T是一个算术类型或子类型，且E是一个算术表达式，则E被转换为类型T。

(10) T' E

T必须是一个类型名；E是任意表达式。E必须被求值以产生一个T类型的值。这种形式的类型指定对于解决重载冲突是有用的。

A.26 重载

如下所述，包被设计为方便用户定义新的类型和操作。除非允许重载，否则新定义的类型不能使用已有的运算符和子程序名——那将导致不一致的和难以使用的程序。

在Ada/CS中，运算符和子程序名可以被重载。两个定义可以共享（即重载）一个运算符或子程序名，前提是它们的参数数目、参数类型或者返回值类型不同。我们将从上下文中确定使用共存定义中的哪一个定义。如果不能做出惟一选择，则程序有错误。这样的错误通常可以通过用标识符或运算符在其中定义的块、子程序或包来限定它们而得到解决（例如，Sqrt.Min(X)而不是Min(X)）。拥有相同参数数目和相同参数类型以及相同结果类型的运算符或子程序不能共存。如果一个定义和另一个定义处于不同 748
的作用域中，则应用通常的作用域规则，而且内部定义将遮蔽外部定义。如果两个定义处于同一作用域，则程序具有一个多重定义错误。

在所有情况的遮蔽（包括变量、类型和常量名）中，显式限定可被用来引用一个被遮蔽的名字（这也是为什么允许为块命名的原因）。只有运算符和子程序名能被重载。其他所有名字都必须拥有由作用域规则所确定的惟一定义。

A.27 包

实际的程序设计语言需要某种机制将程序模块化。模块化对于将大的程序组织为可管理的单元、允许程序片段组成的库以及允许程序单元的分离编译来说是必需的。在Ada/CS中，模块化的单位是包(package)。一个Ada/CS程序就是一个或多个包的序列。包的形式为

```
package ID is
   Type, variable, constant and subprogram headers
private
   Type declarations
body
   Type, variable, constant and subprogram declarations
begin
   Statement list
exception
   Exception handler declarations
end;
```

如果没有私有类型或异常处理程序，则**private**和**exception**部分可以被省略；语句列表部分和之前的**begin**也是可选的。

在Ada中，一个包必须被划分为独立的声明和主体部分，如下所示。在Ada/CS中，这种划分是允许的但不是必需的。

```
package ID is
   Type, variable, constant and subprogram headers
private
   Type declarations                                          749
end;
package body ID is
   Type, variable, constant and subprogram declarations
begin
   Statement list
exception
   Exception handler declarations
end;
```

这种分离有两个目的。如果两个包必须互相引用，我们能够通过首先列出包的声明并随后列出主体部分来避免前向引用，如同对子程序所做的那样。更重要的是，包声明部分包含编译对包组件的引用所需的所有信息。这意味着各个包的主体可以被预编译并链接起来以形成一个可执行的目标程序。

出现在包声明部分的类型、变量、常量和子程序头可以对其他包可见。它们因此被称为包的可见部分（visible part）。对象可以通过用它所在的包名来限定它的对象名这种方式来引用。因此，P.A指示包P中的对象A。访问包声明部分里的对象的另一种方式是使用如下所述的**use**语句。

我们希望使用包来实现抽象数据类型，但这种用法导致一个问题。如果将类型定义放在包的可见部分，则它的实现在包外可见。这种可见性不是我们所希望的，而我们希望的是通过抽象数据类型的操作而不是其实现来刻画它。放在包主体中的声明在包外永远不可见，因此不能将抽象数据类型的定义放在那里。包声明的**private**部分用于解决这个问题。在包的可见部分，类型被声明为**private**，类型的实现被隐藏在**private**部分中。因此，在以下声明中：

```
package Set_Stuff is
type Set is private;
private
   type Set is array(1 .. Max_Set) of Boolean;
end Set_Stuff;
```

类型名Set是可见的，但它被实现为数组这一事实是不可见的。假如所有声明都在可见部分，则其实现应当是可见的。通过使类型私有，可以保证它们仅由包自身所定义的操作、加上自动提供给私有类型的赋值和相等运算符来操纵。包主体中的所有声明对包的外部都是完全隐藏的。规范部分的声明在主体部分中可见，而主体部分负责实现在规范部分声明的子程序。在声明部分中声明的子程序必须在包的主体
750 部分中定义。如果没有声明子程序，则包主体部分是不需要的。例如

```
package Set_Stuff is
type Set is private;
function In (I:Integer; S:Set) return Boolean;
private
   type Set is array(1 .. Max_Set) of Boolean;
body
   function In (I:Integer; S:Set) return Boolean;
   begin
      return S(I);
   end;
end Set_Stuff;
```

跟在声明后面的包主体部分中的语句（如果有的话）在执行包主体时被执行。包主体以其组成主程序的顺序执行。如果一个包是单独编译的，我们插入桩代码

```
package body ID is separate;
```

来标记在哪里插入单独编译的主体。通常，除最后一个包主体外的所有包主体都只包含初始化代码。最后一个包主体事实上是主程序，因此该包最有可能会拥有跟在声明后面的语句。

在包中声明的所有变量和常量（但不包括包内子程序中声明的变量和常量）都是静态的。也就是说，它们在包被执行时被创建，并在包主体被执行之后继续存在。这种设计是必要的，因为后面的包可能引用前面这些包中的对象。

A.28 use子句

use子句（它可选地放在声明段前）可以指定一个或多个包的可见部分为可访问的。声明

```
use p1, p2,. . ., pn;
```

（其中p1，p2，. . .，pn指定包名）导致包的所有可见定义成为可访问的（就像它们已经在本地定义）。然

而，有特殊规则控制名字冲突：

- 如果一个名字可以利用通常的作用域规则找到（排除由包所提供的名字），则它总是可应用的。（来自包的名字仅在不遮蔽其他已有名字时才被允许）。
- 如果**use**列表中的多个包提供相同的名字，则将不包含这些冲突的包定义中的任何一个。（该规则确保**use**列表中指定的包的顺序是无关紧要的）。 751

use子句可以引入重载，只要没有声明被遮蔽并且没有冲突的声明被引入。例如，假定

```
function "+" (X,Y : Matrix) return Matrix;
```

处于**use**子句所指定的某个包的可见部分。只要在两个Matrix对象上的“+”定义先前不存在或它不会出现在**use**子句所指定的其他包中，则该“+”定义将被允许重载已有的定义。

A.29 异常

程序有时必须处理未预料到的情况或出错的情况。为此目的，Ada/CS提供了异常。异常被声明在块、子程序或包的声明部分里，例如

```
Illegal_Data, Symbol_Table_Full : exception;
```

Ada/CS提供了许多预定义的异常：

- Constraint_Error

  在一个数组下标越界或者违反了区间约束时抛出。

- Numeric_Error

  在上溢、下溢、被零除以及指数非法时抛出。

- Storage_Error

  在存储请求不能满足时抛出。

- Time_Limit

  在达到执行时间限制时抛出。

- Eof_Error

  在试图越过文件结束符读数据时抛出。

- Invalid_Input

  在试图读取无效输入项时抛出。

预定义的异常可以在执行时作为运行时错误的结果而被自动抛出。所有异常都可以由**raise**语句显式地抛出，例如：

```
raise Invalid_Input;
```

当一个异常在某条语句执行过程中被抛出时，则语句执行被中断，并在直接外围块、子程序或包的异常部分查找*异常处理程序*（exception handler）。异常部分看起来非常像一条**case**语句，只是其选项都标以异常名： 752

```
exception
   when exception | . . . | exception => sequence of statements;
   when exception | . . . | exception => sequence of statements;
     . . .
```

像**case**语句一样，最后一个**when**可以使用选项**others**来覆盖所有没有被显式指定的异常。

如果找到一个异常处理程序，则执行相应的语句，并且随后退出该块、子程序或包。所执行的语句遵守与块、子程序或包中的其他语句相同的作用域规则。在抛出异常的点的执行不被恢复。

如果没有找到异常处理程序，则传播该异常。当异常从一个块中传播出来时，将检查直接包含该块

的程序块、子程序或包的异常部分。当异常从一个子程序中传播出来时，将返回到调用点并检查直接包含该调用的块、子程序或包的异常部分。当异常从一个包中传播出来时，将用一条指定所抛出异常名的出错信息来终止执行。

要处理在声明或异常处理程序的执行期间所抛出的异常，方法是将该异常从直接包含出现异常的声明或异常处理程序的块、子程序或包中传播出来。

A.30 Ada/CS文法

下面列出Ada/CS的扩展BNF文法。非终结符号被置于“<”和“>”中。保留字用粗黑体表示，词法记号类是没有被界定的单词，所有其他符号（除[、]、{和}之外）将表示它们自身。符号→分隔产生式的左部和右部。如果没有左部，则认为左部为前面一个产生式的左部。

753

<compilation> → <pragma list> <compilation unit>
{ <pragma list> <compilation unit> }

<pragma list> → { <pragma> }

<pragma> → **pragma** id ;

<compilation unit> → <package declaration>

<package declaration> → **package** <package spec or body> ;

<package spec or body> → <id> **is** { <spec declaration> } [<private part>]
<body option> **end** <id option> ;
→ **body** <id> **is** { <body declaration> }
[**begin** { <statement> }] [<exception part>]
end <id option> ;

<body option> → [**body** { <body declaration> }
[**begin** { <statement> }] [<exception part>]]

<id option> → [<id>]

<spec declaration> → <private type declaration>
→ <declaration>

<private type declaration> → **type** <id> **is private** ;

<private part> → **private** <private item> { <private item> }

<private item> → **subtype** <id> **is** <subtype definition> ;
→ **type** <id> **is** <type definition> ;

<body declaration> → <subprogram body decl>
→ <declaration>

<declaration> → <object declaration>
→ <type declaration>
→ <subtype declaration>
→ <pragma>
→ <subprogram declaration>
→ **use** <name list> ;
→ <id list> **exception** ;

<object declaration> → <id list> : <constant option> <type or subtype>
<initialization option> ;

<id list> → <id> { , <id> }

<id> → Identifier

<constant option> → [**constant**]

<type or subtype> → <type>
→ <subtype definition>

<initialization option> → [:= <expression>]

<type declaration> → **type** <id> **is** <type definition> ;
→ <incomplete type decl>

<type> → <type name>
→ <type definition>

<type name> → <id>

<type definition> → <record type definition>
→ <array type definition>
→ <enumeration type def>
→ **access** <subtype>

<incomplete type decl> → **type** <id> ;

<record type definition> → **record** <component list>
end record

<component list> → <component declaration> { <component declaration> }
→ { <component declaration> } <variant part>
→ **null** ;

<component declaration> → <id list> : <type or subtype> <initialization option> ;

<variant part> → **case** <id> : <type name> **is** <variant>
{ <variant> } **end case** ;

<variant> → **when** <v choice> => <component list>

<v choice> → <simple expression>

<array type definition> → <unconstrained array def>
→ <constrained array def>

<unconstrained array def> → **array** <unconstrained index list> **of** <element type>

<unconstrained index list> → (<index subtype def> {, <index subtype def>})

<index subtype def> → <type name> **range** <>

<constrained array def> → **array** <constrained index list> **of** <element type>

<constrained index list> → (<discrete range> {, <discrete range>})

<element type> → <type or subtype>

<enumeration type def> → (<enumeration id list>)

<enumeration id list> → <id> { , <id> }

<subtype declaration> → **subtype** <id> **is** <subtype definition> ;

<subtype> → <type name>
→ <subtype definition>

<subtype definition> → [<type name>] <range constraint>
→ <type name> <index constraint>

<range constraint> → **range** <range>

<range> → <simple expression> .. <simple expression>

<index constraint> → (<discrete range> {, <discrete range>})

<discrete range> → <subtype>
→ <range>

<subprogram declaration> → <subprogram specification> ;

<subprogram body decl> → <subprogram specification> **is** { <body declaration> }
begin { <statement> } [<exception part>]
end <id option> ;

<subprogram specification> → **procedure** <id> <formal part opt>
→ **function** <designator> <formal part opt>

<designator> → <id>
→ <operator symbol>

<operator symbol> → StrLiteral

<formal part> → (<parameter declaration list>)

<formal part opt> → [<formal part>]

<parameter declaration list> → <parameter decl> { ; <parameter decl> }

<parameter decl> → <id list> : [<mode>] <type or subtype>

<mode> → **in** [**out**]
→ **out**

<exception part> → **exception** { <exception handler> }

<exception handler> → **when** <exception when tail>

<exception when tail> → **others** => { <statement> }
→ <name> { | <name> } => { <statement> }

<statement> → <pragma>
→ <null statement>
→ <assignment statement>
→ <call statement>
→ <block>
→ <loop statement>
→ <if statement>
→ <exit statement>
→ <return statement>
→ <case statement>
→ <raise statement>

<null statement> → **null** ;

<assignment statement> → <name> := <expression> ;

<call statement> → <name> ;

<block> → [<id> :] [<decl part>] **begin** { <statement> }
{ <exception part> } **end** [<id>] ;

<decl part> → **declare** { <body declaration> }

<return statement> → **return** [<expression>] ;

<raise statement> → **raise** <name option> ;

<if statement> → **if** <b expr> **then** { <statement> }
{ **elsif** <b expr> **then** { <statement> } } [<else part>]
end if ;

<else part> → **else** { <statement> }

<loop statement> → [<id> :] [<iteration clause>] <basic loop> ;

<basic loop> → **loop** { <statement> } **end loop**

<iteration clause> → **while** <b expr>
→ **for** <id> **in** [**reverse**] <discrete range>

<exit statement> → **exit** [<name>] [**when** <b expr>] ;

<case statement> → **case** <expression> **is** <when list> <others option>
end case;

<when list> → { **when** <choice list> => { <statement> } }

<others option> → [**when others** => { <statement> }]

<choice list> → <choice> { | <choice> }

<choice> → <expression>
→ <expression> .. <expression>

<b expr> → <expression>

<expression> → <relation> { <logical op> <relation> }
→ <relation> { **and then** <relation> }
→ <relation> { **or else** <relation> }

<relation> → <simple expression>
[<relational op> <simple expression>]

<simple expression> → [<unary adding op>] <term> { <adding op> <term> }

<term> → <factor> { <multiplying op> <factor> }

<factor> → <primary> [** <primary>]
→ **not** <primary>
→ **abs** <primary>

<primary> → <literal>
→ <name>
→ (<expression>)
→ aggregate

<literal> → IntLiteral
→ FloatLiteral
→ StrLiteral

<logical op> → **and**
→ **or**

<relational op> → =
→ /=
→ <
→ <=
→ >
→ >=

<adding op> → +
→ -
→ &

<unary adding op> → +
→ -

<multiplying op> → *
→ /
→ **mod**

<name> → <simple name> { <name suffix> } [. **all**]

<simple name> → <id>

<name suffix> → . <selected suffix>
→ (<expression> { , <expression> })
→ ' <id>

<selected suffix> → <id>
→ <operator symbol>

<aggregate> → <name> ' (<component> { , <component> })

<component> → [<agg choice list> =>] <expression>

<agg choice list> → <agg choice> { | <agg choice> }

<agg choice> → <simple name>
→ <simple expression>
→ <discrete range>
→ **others**

<name list> → <name> { , <name> }

754
~
758

附录B ScanGen

ScanGen由Gary Sevitsky编写并随后由Robert Gray进行了增强。最近的改动由Charles Fischer完成。ScanGen接受以正则表达式书写的词法记号描述并产生可用于驱动词法分析器的表格。

B.1 输入规范

输入格式将通过带注释的示例进行介绍，由图B-1开始。

```
Options
    tables,list
Class
    letter          = 'A'..'Z', 'a'..'z';
    digit           = '0'..'9';
    blank           = ' ';
Definition
    Token emptyspace {0}    = blank+;
    Token identifier {1}    = letter.(letter, digit)*;
    Token number {2}        = digit+;
```

图B-1 一个简单的ScanGen规范

ScanGen的规范包含三部分：所选择的选项、类定义和正则表达式定义。

选项部分是可选的。如果有选项部分，则它会以保留字Options开头，后面跟一个或多个选项名（不是保留字）。选项名可以以任何顺序出现，并以空格或逗号分隔。选项的完整列表在B.3节中给出。 759

类定义指定那些组成正则表达式所用字母表的字符类。字符类是如图B-1中那样通过使用引号中的单独的字符或字符区间来定义的字符集。为指定引号字符，使用''''。不可打印字符通过其等价的十进制数表示。例如，换行符可以通过linefeed = 10（或者你的字符集中的相应数字）来指定。

如果在类定义中没有提到某个字符，则它不能在输出表格中创建任何词法分析器动作：它将被忽略。生成器将未提及的字符放在字符类Epsilon中。每个字符至多可被赋给一个字符类。

要构造用于指定词法记号的正则表达式定义，需要利用字符类和下列运算（以优先级降序列出）：正闭包（+）和Kleene闭包（*），连接（.）和联合（,）。可以通过使用括号来改变优先级。每个词法记号名后面跟一个词法记号编号。词法记号编号出现在输出表格中，使得词法分析器可以在识别词法记号时返回一个词法记号编号。

图B-2举例说明了定义正则表达式的更复杂方式。第二个定义，即Letter的定义，没有定义词法记号；它定义了一个可用于后续定义的辅助正则表达式。词法记号也可以用于后续定义中；例如，IntLit可以用于RealLit的定义中。

该规范语言的另一个特性是*异常列表*（exception list），它可以用于Identifier的定义中。异常列表由称为*保留字*的字符串组成，长度小于等于12，后面跟词法记号编号。异常列表不会影响输出表格；它被单独存储以便词法分析器可以访问它。

在IntLit、RealLit和StrLit的定义中，有两个词法记号编号跟在词法记号名后面，它们是：*主词法记号编号*（major token number）和*次词法记号编号*（minor token number）。主词法记号编号可以用于定义一个词法记号类，次词法记号编号可以用于指定该类的成员。如果没有指定次词法记号编号，则

生成器将提供默认值50。词法记号编号必须是非负整数。相同的词法记号编号可以用于不同的词法记号。习惯上，将要被删除的词法记号（注释、空格、制表符）的主词法记号编号为0。

```
Options
    List, tables

Class
    E             = 'E', 'e';
    OtherLetter   = 'A'..'D','F'..'Z','a'..'d','f'..'z';
    Digit         = '0'..'9';
    Blank         = ' ';
    Dot           = '.';
    Plus          = '+';
    Minus         = '-';
    Quote         = '"';
    Linefeed      = 10;

Definition

Token EmptySpace (0)    = (Blank, Linefeed)+;
Letter                  = E, OtherLetter;
Token Identifier (1)    = Letter.(Letter,Digit)*
                           Except
                              'begin' (4),
                              'end' (5);
Token IntLit (2,1)      = Digit+;
Token RealLit (2,2)     = IntLit.Dot.IntLit.
                          (Epsilon, E.(Epsilon, Plus, Minus).IntLit);
Token StrLit (2,3)      = Quote(Toss).
                          (Not(Quote, Linefeed),Quote(Toss).Quote)*
                           . Quote(Toss);
Token RunOnString (3)   = Quote(Toss).
                          (Not(Quote, Linefeed), Quote(Toss).Quote)*
                           . Linefeed(Toss);
```

图B-2 一个更复杂的ScanGen规范

在RealLit的定义中使用了字符类Epsilon，使得输出表格将识别没有指数部分的数和带无符号指数的数。

Not运算在StrLit和RunOnString的定义中使用。该运算仅可用于求一个字符类联合的补集。补集的取得相对于类定义中所指定的类。换句话说，字符类Epsilon不在补集中。

760 Toss特性出现在StrLit和RunOnString的定义中。该特性用于告诉词法分析器是否把一个字符追加到它正在创建的词法记号串的末尾。如果不追加一个字符，则将Toss放在词法记号定义中其字符类名的后面。Toss只能出现在字符类名或Not(…)后面。Toss特性的误用会导致一个丢弃/保存冲突。例如，如果StrLit由

```
Quote(Toss) . (Not(Quote, Linefeed), Quote.Quote(Toss))*
  . Quote(Toss)
```

定义，则会发生一个丢弃/保存冲突。

该冲突可以通过比较字符串'a'和'a''b'的词法分析器动作而被发现。在第一种情况下，词法分析器被告知丢弃跟在a后面的引号；但在第二种情况下，该字符将被保存。当发生一个丢弃/保存冲突时，生成器将打印一条出错信息。

761 输入规范可以用每行最多132个字符的自由格式书写。非法符号会被忽略，并打印一条警告信息。标识符可以为任意长度，但只有前12个字符被检查。标识符的大小写会被忽略（即，ABC和abc是相同的标识符）。保留字Class、Definition、Epsilon、Except、Not、Token、Options和Toss中的大小写也会被忽略。当发现一个语法错误时，生成器将打印一条出错信息并终止。

B.2 输出

当使用tables选项时，将产生外部文件tables。该文件由下列5段组成：

第1段：词法分析器参数

一系列由空格隔开的项，每项含5个整数。

NumStates

最小DFA的状态数。

StartState

最小DFA的初始状态。

NumClasses

由用户定义的字符类数目（不包括Epsilon）。

NumResWords

在Except子句中定义的保留字总数。

NumLists

拥有异常列表的词法记号名的数目。

第2段：字符类映射

一系列由空格隔开的项，每项含N个整数，指定每个字符被赋予的字符类。N由配置ScanGen所使用的字符集确定（对ASCII，N=128；对EBCDIC，N=256）。列表中第i个元素是赋给其值（ord）为i的字符的类编号。字符类Epsilon的编号为0，因此，不显式赋予任何类的字符将被指定字符类编号为0。

第3段：保留字到词法符号的映射

包含NumLists条记录，每条记录的形式为

Major Minor FirstRSW LastRSW

Major和Minor是以保留字FirstRSW到LastRSW作为异常的词法记号类的数目。FirstRSW和LastRSW是组成该输出文件第4段的保留字表中的索引。保留字编号从1开始。该段中的信息仅当多个词法记号定义拥有异常列表时才是重要的。

762

第4段：保留字列表

包含NumResWords 条记录，每条记录的形式为

Word (columns 1-12) Major Minor

Major和Minor是保留字的词法记号编号，以空格分隔。

第5段：最小确定有限自动机的转换表

转换表被写成NumStates × NumClasses个条目的数组。状态和类的编号都从1开始。数组以行主序书写，这意味着类编号变化得更快。

每个条目的第一个整数指示出转换类型：Error、MoveAppend、MoveNoAppend、HaltAppend、HaltNoAppend和HaltReuse。这个整数后面跟0～2个整数（具体数目依赖于转换类型），它们给出关于转换的更多信息。

转换具有以下几种形式：

0	—— Error	
1 NextState	—— MoveAppend:	移动到NextState并将当前字符追加到正在组装的词法记号的末尾。
2 NextState	—— MoveNoAppend:	移动到NextState并消耗当前字符，但不将其追加到当前词法记号的末尾。
3 Major Minor	—— HaltAppend:	停止，返回词法记号编号Major和Minor，并将当前字符追加到当前词法记号的末尾。

4 Major Minor	—— HaltNoAppend:	停止，返回词法记号编号Major和Minor，并消耗当前字符，但不将其追加到当前词法记号的末尾。
5 Major Minor	—— HaltReuse:	停止，返回词法记号编号Major和Minor，保存当前字符以便在下一个词法记号中复用，并且不将其追加到当前词法记号的末尾。

763

B.3 杂项

这个词法分析器生成器产生仅使用一个超前搜索字符的表格。该表格不能直接处理Pascal和Ada/CS的".."问题或FORTRAN的".EQ."问题。然而，多字符超前搜索问题可以通过保存在扫描一个词法记号时所访问的状态序列并回退到其中最后的终结状态来处理。

该表格总是识别最长可能的词法记号串。如果整个词法记号类每次都是以另一个类的某个成员的前缀形式出现，则可能无法识别该词法记号类。

如果一个状态是多个词法记号类的终结状态，则在输入文件中定义的第一个词法记号类将是被接受的一个类。如果一个词法记号类包含着另一个词法记号类，则此约定是很重要的。例如，在用于Ada/CS的定义（见下）中，`RealLit`的定义包含整数常量。然而，因为`IntLit`在`RealLit`之前，所以数字序列将被正确地扫描为整数而不是实数常量。

根据要处理的正则表达式的大小和复杂度，ScanGen中常量`MaxSubsets`的值可以被改变（其初始设定为75000）。减小`MaxSubsets`可以显著地缩小程序的大小；增加它会允许处理更大和更复杂的定义。

B.4 ScanGen选项

`List:`	在标准输出文件中列出输入规范。
`Dfa:`	打印词法分析器生成器所构造的最小DFA。
`Report:`	打印由以下部分组成的报告： 1) 字符类名和字符类编号的对应关系。 2) 字符到字符类编号的映射。 3) 保留字和它们被赋予的词法记号类编号的列表。
`Test:`	使用文件`testfile`作为词法分析器的样本输入来测试DFA。词法分析器将列出在扫描每个符号时的动作并在到达文件结尾时终止。建议测试文件保持短小。也可以把选项`Dfa`的输出放在旁边对照。
`Tables:`	产生文件`tables`，其中包含字符类映射、保留字列表和转换表。
`Optimize:`	优化由ScanGen所创建的表格。在产生用于词法分析的表格时必须总是使用该选项。在调试运行中不必使用该选项。

764

用于ScanGen自身调试的其他选项在ScanGen的源码中说明。

B.5 Ada/CS的ScanGen定义

下列定义表示一个ASCII版本的Ada/CS；EBCDIC版本在特定字符编码的赋值上会有不同。

```
Options
 tables,optimize
Class
E              = 'E','e';
OtherLetter    = 'A'..'D','F'..'Z','a'..'d','f'..'z';
```

```
Digit          = '0'..'9';
Linefeed       = 10;
Blank          = ' ';
Amper          = '&';
Quote          = '"';
SingleQuote    = '''';
LParenChar     = '(';
RParenChar     = ')';
Star           = '*';
PlusChar       = '+';
MinusChar      = '-';
DotChar        = '.';
Slash          = '/';
ColonChar      = ':';
Semi           = ';';
Less           = '<';
Equal          = '=';
Greater        = '>';
BarChar        = '|';
CommaChar      = ',';
UnderScore     = '_';
Tab            = 9;
Unprintable    = 0..8,11..31,127;
Illegal        = '!','#','$','%','?','@','\','~','`', '{','}', '[', ']','^';

Definition

Letter                  = E,OtherLetter;
Token  Ampersand{6}     = Amper{Toss};
Token  Bar{5}           = BarChar{Toss};
Token  Box{23}          = Less{Toss} . Greater{Toss};
Token  Becomes{8}       = ColonChar{Toss}.Equal{Toss};
Token  Choose{24}       = Equal{Toss}. Greater{Toss};
Token  Colon{19}        = ColonChar{Toss};
Token  Comma{15}        = CommaChar{Toss};
Token  Dot{16}          = DotChar{Toss};
Token  DotDot{17}       = DotChar{Toss}.DotChar{Toss};
Token  LParen{26}       = LParenChar{Toss};
Token  Minus{21}        = MinusChar{Toss};
Token  MultOp{22}       = Star{Toss};
Token  DivOp{25}        = Slash{Toss};
Token  Plus{20}         = PlusChar{Toss};
Token  LT{9}            = Less{Toss};
Token  LE{11}           = Less{Toss}.Equal{Toss};
Token  EQ{7}            = Equal{Toss};
Token  NotEQ{13}        = Slash{Toss}.Equal{Toss};
Token  GE{12}           = Greater{Toss}.Equal{Toss};
Token  GT{10}           = Greater{Toss};
Token  RParen{27}       = RParenChar{Toss};
Token  Semicolon{18}    = Semi{Toss};
Token  Tic{4}           = SingleQuote{Toss};
Token  ToThe{14}        = Star{Toss}.Star{Toss};
Token  IntLit{2,1}      = Digit.(Digit,UnderScore{Toss})*;
Token  RealLit{2,2}     = IntLit.(Epsilon,DotChar.IntLit).
                          (Epsilon,(E.(Epsilon,PlusChar,MinusChar)
                          .IntLit));
Token  StringLit{3}     = Quote{Toss}.
                          (Not(Quote,Linefeed,Tab,Unprintable),
                          Quote{Toss}. Quote)*.Quote{Toss};
Token  EmptySpace{0}    = (Blank{Toss},Linefeed{Toss},Tab{Toss})+;
Token  Comment{0}       = MinusChar{Toss}.MinusChar{Toss}.
                          (Not Linefeed{Toss})*.
                          Linefeed{Toss};
Token  Identifier{1}    = Letter.(Letter,Digit,UnderScore)*
                              Except
                              'abs' {28},
                              'and' {29},
                              'array' {30},
                              'begin' {31},
                              'body' {32},
                              'case' {33},
                              'constant' {34},
                              'declare' {35},
                              'else' {36},
                              'elsif' {37},
```

```
'end' (38),
'exception' (39),
'exit' (40),
'for' (41),
'function' (42),
'if' (43),
'in' (44),
'is' (45),
'loop' (46),
'mod' (47),
'not' (48),
'null' (49),
'of' (50),
'or' (51),
'others' (52),
'out' (53),
'package' (54),
'pragma' (55),
'private' (56),
'procedure' (57),
'raise' (58),
'range' (59),
'record' (60),
'return' (61),
'reverse' (62),
'access' (63),
'subtype' (64),
'then' (65),
'type' (66),
'use' (67),
'when' (68),
'while' (69),
'all' (70);
```

765
~
767

附录C　LLGen用户手册

LLGen接受上下文无关文法规范并产生用于分析指定语言的表格。它能对任意LL(1)文法产生表格，并对非LL(1)文法提供简单的冲突解决机制。LLGen是FMQ的一个子集，是一个LL(1)分析器/错误修正器的生成器。FMQ由Jon Mauney编写。

该报告描述了一个语言翻译工具——具体说，就是用于分析上下文无关语言的工具。LLGen是一个表生成器。它接受用以下所描述格式指定的LL(1)文法并产生可用于语法分析的表格。通过语义动作编号，LLGen也提供与用户提供的语义动作的接口。这些编号在LLGen的输入中被指定并出现在所产生的表格中。

LLGen的一个典型使用过程如下：

1) 创建一个指定所要文法的文件。

2) 运行LLGen，将文法文件定向到标准输入。LLGen将把可选的输出和出错信息（如果有的话）发送到终端（或任何形式的标准输出）并创建一个文件ptableout或ptablebin（见下面C.2节）。

3) 如果文法不能被LLGen接受，重复1和2。

4) 在LL(1)分析器驱动程序中使用ptableout或ptablebin。

C.1　LLGen的输入

LLGen的输入主要有3段：运行所需的选项、文法的终结符号以及文法的产生式规则。输入的一般形式为


```
    <comments>
*fmq
    <options>
*define
    <constant definitions>
*terminals
    <terminal specifications>
*productions
    <production specifications>
*end
    <comments>
```

示例见图C-2。

在下面的叙述中，*符号*是指要生成表格的文法中的符号，而*词法记号*是指LLGen的输入中的实体。

通过三个简单规则，可以将LLGen的输入划分为若干个词法记号：

- 所有词法记号必须由一个或多个空白、制表符或行结束符分隔。
- 词法记号不能包含空白或制表符，除非该词法记号被置于尖括号<和>中。词法记号不能跨越行边界。
- 如果一个词法记号以<开始，则它必须以>结束。

也就是说，输入中的任何东西——选项名、保留字、文法符号——都必须由空白符分隔。在一个符号中含有空白符时可以使用尖括号，但仅当第一个字符是<的情况下它们才有特殊意义。出现在其他任何环境中的尖括号是合法的，但没有特殊含义（在这种情况下将给出一条警告）。其他符号中惟一被赋予特殊性质的是#，它是所有*动作符号*（action symbol）的开始字符。动作符号由#后面直接跟随无符号整数或（如下所述的）已定义常量组成。

大写和小写字母被认为是不同的；然而，被保留的词法记号和选项无论大写、小写或大小写混合都能被识别。图C-1中给出的示例说明了上述规则。

词法记号	注释
ABC	OK
abc	OK，与ABC不同。
123	OK
< Expr >	OK
<id list>	OK
<>	OK
&:=	OK
"<"	合法，得到一条警告
—>	合法，得到一条警告
<not<>equal>	合法，得到一条警告
much<<lessthan	合法，得到一条警告
<LHS>::=<RHS>	合法，得到一条警告。这是一个词法记号，而不是三个。
2<two tokens>	合法，两个词法记号，两条警告 (<仅当首先出现时才是特殊的)
*fmq	保留
*FMQ	保留，与*fmq相同
*Fmq	保留，与*fmq相同
#13	合法，例程13的动作符号
#add	合法，add应当已经在*define段中被定义
*fmq*terminals	合法，一个词法记号，没有警告
<=	不合法，没有结束的尖括号
<	不合法，没有结束的尖括号
<bad>token	同上，(>后面必须跟随空格)

图C-1 LLGen词法记号的示例

下列词法记号是被保留的：

```
*fmq    *define    *terminals    *productions    *end
::=     ...        <Goal>        $$$
```

行结束符作为如下所述的终结符规范、产生式和常量定义的结束符是必需的。除此之外，LLGen的输入是自由格式的。

在*fmq之前或*end之后的任何东西都将被认为是注释并被忽略。然而，注释不能包含上述任意被保留的词法记号。跟随在*fmq后面的是零个或多个选项的列表，和通常一样以空白、制表符或行结束符
769 分隔。所有选项都具有开关的形式并可通过在选项列表中包含其名字而激活。一个被激活的选项可以通过在其名字前放置no而被禁止（no与选项名之间没有空格）；因此，为阻止分析表的构造，可使用noparsetable。除了用于checkreduce和text的选项之外的所有选项在初始时都是被禁止的。大写和小写的选项都能被识别。可用的选项有

(1) bnf

打印文法规则。

(2) first

打印所有非终结符的First集。

(3) follow

打印所有非终结符的Follow集。

(4) parsetable

以表格形式打印分析动作表。表中的数字指示对产生式的预测；空白条目指示错误。该表可能相当大。

(5) checkreduce

检查文法是否可归约；报告所有不能产生终结符串的符号以及那些从开始符号无法到达的符号。如果文法不可归约，且激活了checkreduce，则不会生成表格。checkreduce通常是被激活的。 770

(6) resolve

如果给定的文法不是LL(1)的，仍然会生成分析表，并通过优先选择较早出现在输入中的产生式来逐对解决分析冲突。该选项应当谨慎使用；见“LLGen中的错误处理”一节的讨论。如果resolve被禁止，则在存在分析冲突的情况下将停止计算分析表。

(7) shortline, longline

控制易被人阅读的输出（vocab、parsetable等）中打印行的长度。shortline导致每行少于80个字符（适合屏幕显示），而longline是132个字符（适合打印机）。shortline和nolongline是同义语，反之亦然。默认选项是shortline。

(8) statistics

打印文法的分类统计信息。

(9) vocab

打印给定语言的符号。

(10) text, binary

由LLGen所创建的表可以被写成文本形式（字符文件）、二进制形式（整数文件）或同时写成这两种形式。文本输出被写到文件ptableout；二进制输出被写到ptablebin。二进制文件往往更大一些，至少在32位机器上要大一些（在VAX机器UNIX™操作系统下大约大30%），但它们通常读起来更快。默认选项是text和nobinary。

常量定义段是可选的。如果存在，则以保留的词法记号*define开始，由一系列定义组成，每条定义都处在单独一行中。每条定义的形式为

<const name> <integer value>

其中<const name>是上述的一个词法记号，而<integer value>是一个无符号整数（即仅含数字的词法记号）。该常量可以随后用于任何需要整数的地方：在后续常量定义中，以及用作语义例程编号。注意：该特性并不像它最初看起来那么好，因为LLGen的输出列表将使用数字值，而不是常量名。

保留的词法记号*terminals开始终结符号列表。终结符规范段由一系列这样的规范组成：每条规范处于单独一行中。所有终结符都必须出现在该列表中。

词法记号*productions将产生式和终结符分隔开。产生式由一列规则指定，每条规则处于单独一行中。 771

产生式规范的形式为

<lhs> ::= <rhs>

<lhs>或<rhs>都可以被省略。<lhs>是表示非终结符号的词法记号。如果没有<lhs>，则使用前面一条产生式的<lhs>。<rhs>是词法记号串，其中包含产生式的文法符号以及指示当到达产生式的适当点时将被调用的语义例程的动作符号。动作符号由#后面跟随一个无符号整数或已定义常量组成，其间没有空白。

如果没有<rhs>或其中只含有动作符号，则<lhs>推出空串。可以通过以保留的词法记号“”. . .”” 开始一行使<lhs>在那些后续的行中继续（仅有产生式可以这样续行）。

产生式以*end终止。在所有产生式都被处理之后将添加拓广产生式。两个符号<Goal>和$$$以及一个产生式

```
<Goal> ::= <S> $$$
```

被添加到文法中，其中<S>是指定的第一个产生式左边的符号，<Goal>是开始符号，而$$$是结束标记。

C.2 LLGen的输出

由上述选项控制的输出被写到标准Pascal文件output中。此外，包含分析表的文件也将被创建。分析表被写到ptableout（文本格式）或ptablebin（二进制格式）中。这些文件可以被指定或重定向，具体情况要依操作系统而定。

不论输出文件被创建为text文件还是binary文件，输出文件的形式都是相同的。在binary文件中，字符值被写成该字符的ord（序号）。

文件ptableout和ptablebin包含下列表：所有产生式的右部、语法分析器动作表、能够推出空串的符号列表以及文法中所有符号的符号化表示。

文件格式如下所示：

标题行：

第一行给出各个表的大小。它包含语言中终结符的数目（numterms）、文法符号的数目（numsymbols）、文法中产生式的数目（numprods）、符号映像的字符串大小（stringsize）以及一个指
772 示是否产生了错误修复表的标志。如果创建了修复表，则该标志为字符T，否则为字符F。

产生式：

给出所有产生式的右部；所有右部都是逆序存放的，并能够以那个给定的顺序压入分析栈。每个产生式由一个长度以及后面跟随的相应符号的编号组成。动作符号包含在产生式右部，并编码为文法中给定编号的相反数。

分析表：

对每个非终结符号的LL(1)预测由分析表给出。对每个非终结符的预测被存储在一个整数对列表中。列表中的第一个整数对是一个零以及后面跟随的非终结符的编号。后续的整数对的形式为

```
terminal prediction
```

其中prediction是当terminal出现在超前搜索符号中时预测的产生式。该列表在下一个列表的开始处终止。整个分析表以0 0终止。

lambda产生式：

Lambda产生式是那些右部没有（除动作符号外的）符号的产生式。该列表由一个长度n以及后面跟随的n个代表产生式编号的整数表示。（该表对普通的LL(1)分析器是不必要的，但它对于完成错误修复时避免不必要的分析器动作是有用的。）

字符串表：

符号表信息有两种形式。第一种是索引，由numsymbols个整数对组成。每对中第一个整数是字符串中符号的开始点，第二个整数是符号的长度。跟随在索引后面的是stringsize个字符，每行132个。

C.3 LLGen中的错误处理

LLGen输入中的语法错误将由错误恢复例程处理。如果在输入中存在错误，表生成将被中止。

试图使用符号<Goal>和$$$（开始符号和结束标记）将被视为语法错误。

所有终结符必须被列在*terminals段。如果任何终结符没有被列出或一个非终结符没有出现在任何产生式的左部，则该符号会被标记，且不会生成任何表。类似地，被声明为终结符的符号不能出现在产生式左部。

如果所指定的文法不是LL(1)的，那么将报告所有的冲突。如果选项resolve被激活，将按产生式的出现次序确定它们的优先级（第一条指定的产生式有最高的优先级）。因此Pascal和其他语言中的悬空else可以通过

773

```
<if stmt>     ::= if <expr> then <stmt> <else part>
<else part>   ::= else <stmt>
              ::=
```

进行分析。冲突将通过优先选择第一种形式的语句来解决，即把else与最近出现的if相匹配。

该解决机制应当谨慎使用。必须仔细地检查冲突以保证所采取的分析动作是所希望的动作。例如，将上述两条<else part>产生式的顺序颠倒也会很好地被LLGen所接受，但会产生灾难性的后果。当else出现在超前搜索符号中时，所采取的分析动作将总是预测<else part>推导出空串；else将永远不会被接受。

如果指定的文法被证明对于LLGen的限制来说太大了，则程序将会打印一条信息来描述所越过的限制且程序将会终止。LLGen必须随后用增加的限制被重新编译。通常，超过一个限制意味着其他限制也会被突破，一次增加所有的限制将会节省重新编译的时间。注意：LLGen按顺序处理终结符、产生式和分析表。因此，如果报告文法中产生式的数量超过了限制，则终结符的数量必定在限制之内，因为它们已被完全处理了。特定文法的某些方面的问题较容易被发现；而其他的问题则必须通过经验和试验来处理。容易确定的是终结符数目、符号数目（terminals+nonterminals）和产生式数目。不容易确定但容易估算的是产生式中的符号总数和所有不同符号中的字符总数。

C.4 使用LLGen

文法规范被从标准输入文件中读入，而易被人阅读的输出则被写到标准输出中。产生的分析表被写到ptableout或ptablebin中。上述名字是在程序头文件中声明的内部名字，并且可以由运行程序的系统环境来修改。

774

图C-2给出LLGen的一个输入规范示例；图C-3给出将会生成的ptableout文件。

```
Grammar for DCL, a desk calculator language

(the action symbols here don't mean anything,
 they just illustrate how they might be used)

*fmq
statistics
checkreduce
vocab bnf
text  binary
*terminals
    id
    constant
    end
    ;
    :=
    (
    )
```

图C-2 LLGen输入示例

```
    +
    -
    *
    /
    write
    read
    ,

*productions
<prog>          ::= <st list> end
<st list>       ::= <st> #1 <st list tail>
<st list tail>  ::= ; <st list>
                ::=
<st>            ::= id #2 := <expr> #5
                ::= read  #3 ( <id list> #6 #14 )
                ::= write  #4 ( <expr list> #6 )
<expr>          ::= <term> <e tail>
<e tail>        ::= + <term> #20 <e tail>
<e tail>        ::= - <term> #20 <e tail>
                ::=
<term>          ::= <primary> <t tail>
<t tail>        ::= * <primary> #21 <t tail>
<t tail>        ::= / <primary> #21 <t tail>
                ::=
<primary>       ::= - <primary> #7
<primary>       ::= ( <expr> )
                ::= id #8
                ::= constant #9
<expr list>     ::= <expr> #30 <e list tail>
<e list tail>   ::= , <expr list>
                ::=
<id list>       ::= id #15 <id list tail>
<id list tail>  ::= , <id list>
                ::=

*end
```

图C-2 （续）

```
 15 29 26 243 F
2 3 17 3 19 -1 18 2 17 4 0 5 -5 20 5 -2 1 7 7
-14 -6 21 6 -3 13 6 7 -6 22 6 -4 12 2 24
23 4 24 -20 23 8 4 24 -20 23 9 0 2 26
25 4 26 -21 25 10 4 26 -21 25 11 0 3 -7 25 9 3
7 20 6 2 -8 1 2 -9 2
3 27 -30 20 2 22 14 0 3 28 -15 1 2 21 14 0 2 15 16
0 16 13 1 12 1 1 1 0 17 13 2 12 2 1 2 0 18 12 7 13 6 1 5
0 19 3 4 4 3 0 20 9
8 6 8 2 8 1 8 0 21 1 23 0 22 9 20 6 20 2 20 1 20 0 23 9
12 6 12 2 12 1 12 0 24 14 11 7 11 4 11 3 11 9 10 8 9 0 25
2 19 1 18 6 17 9 16 0 26 14 15 9 15 8 15 7 15 4 15 3 15 11
14 10 13 0 27 7 22 14 21 0 28 7 25 14 24 0 29 13 26 12 26
1 26 0 0 5 4 11 15 22 25 95 2 97 8 105 3 108 1 109 2 111 1
112 1 113 1 114 1 115 1 116 1 117 5 122 4 126
1 7 3 127 6 133 9 142 4 146 14 160 6 166 9 175 11 186 6 192
8 200 9 209 8 217 13 230 14 1 6
<Goal>$$$ grammarforDCL(theactionsymbolsheredon'tmeananything,
theyjustillustratehowtheymightbeidconstantend;:=()+-*/wr
iteread,<prog><st list><st ><st list tail>< expr><id list><exp
r list><term><e tail><primary><t tail><e list tail><id list tail>
```

775
~
776

图C-3 给定图C-2的输入文件，由LLGen产生的ptableout

附录D LALRGen用户手册

LALRGen接受上下文无关文法规范并产生用于分析指定语言的表格。它能对任意LALR(1)文法产生表格，并对非LALR(1)文法提供简单的冲突解决机制。LALRGen是ECP的一个子集，是一个LALR(1)分析器/错误修正器的生成器。ECP由Jon Mauney编写。

该报告描述了一个语言翻译工具——具体说，就是用于分析上下文无关语言的工具。LALRGen是一个表生成器。它接受以下面描述的格式所指定的LALR(1)文法并产生可用于语法分析的表格。通过语义动作编号，LALRGen也提供一个和用户提供的语义动作的接口。这些编号在LALRGen的输入中指定并出现在所产生的表格中。

LALRGen的一个典型使用过程如下:

(1) 创建一个能够指定所要文法的文件。

(2) 运行LALRGen，将文法文件定向到标准输入。LALRGen将把可选的输出和出错信息（如果有的话）发送到终端（或任何形式的标准输出）并创建一个文件`ptableout`或`ptablebin`（见下面D.2节）。

(3) 如果文法不能被LALRGen接受，重复1和2。

(4) 在LALR(1)分析器驱动程序中使用`ptableout`或`ptablebin`。

D.1 LALRGen的输入

LALRGen的输入主要有3段：运行所需的选项、文法的终结符号以及文法的产生式规则。输入的一般形式为:


```
    <comments>
*ecp
    <options>
*define
    <constant definitions>
*terminals
    <terminal specifications>
*productions
    <production specifications>
*end
    <comments>
```

示例见图D-2。

在下面的叙述中，*符号*是指要生成表格的文法中的符号，而*词法记号*是指LALRGen的输入中的实体。

通过三个简单规则，可以将LALRGen的输入划分为若干个词法记号:

- 所有词法记号必须由一个或多个空白、制表符或行结束符分隔。
- 词法记号不能包含空白或制表符，除非该词法记号被置于尖括号<和>中。词法记号不能跨越行边界。
- 如果一个词法记号以<开始，则它必须以>结束。

也就是说，输入中的任何东西——选项名、保留字、文法符号——都必须由空白符分隔。在一个符号中含有空白符时可以使用尖括号，但仅当第一个字符是<的情况下它们才有特殊意义。出现在其他任何环境中的尖括号是合法的但没有特殊含义（在这种情况下将给出一条警告）。

大写和小写字母被认为是不同的；然而，被保留的词法记号和选项无论大写、小写或大小写混合都能被识别。图D-1中给出的示例说明了上述规则。

词法记号	注释
ABC	OK
abc	OK，与ABC不同。
123	OK
< Expr >	OK
<id list>	OK
<>	OK
&:=	OK
"<"	合法，得到一条警告
—>	合法，得到一条警告
<not<>equal>	合法，得到一条警告
much<<lessthan	合法，得到一条警告
<LHS>::=<RHS>	合法，得到一条警告。这是一个词法记号，而不是三个。
2<two tokens>	合法，两个词法记号，两条警告
	(<仅当它首先出现时才是特殊的)
*ecp	保留
*ECP	保留，与*ecp相同
*Ecp	保留，与*ecp相同
<=	不合法，没有结束的尖括号
<	不合法，没有结束的尖括号
<bad>token	同上，(>后面必须跟随空格)

图D-1 LALRGen词法记号的示例

下列词法记号是被保留的:

```
*ecp   *define   *terminals   *productions   *end   ##
::=    ...       --           <Goal>         $$$
```

行结束符作为如下所述的终结符规范、产生式和常量定义的结束符是必需的。除此之外，LALRGen的输入是自由格式的。

在*ecp之前或*end之后的任何东西都将被认为是注释并被忽略。然而，注释不能包含上述任意被保留的词法记号。注释也可以被放在任意行的尾部；在记号——到行尾之间的所有文本都将被忽略。

跟随在*ecp后面的是零个或多个选项的列表，以空白、制表符或行结束符分隔。所有选项都具有
778 开关的形式并通过在选项列表中包含其名字而被激活。一个被激活的选项可以通过在其名字前放置no被禁止（no与选项名之间没有空格）；因此，为阻止构造分析表，可使用noparsetable。除了用于checkreduce和text的选项之外的所有选项在初始时都是被禁止的。注意：对于真正程序设计语言的文法大小来说，大多数输出选项都会创建大量输出。在每个输出选项后面括号中的数字给出了所打印的行数的数量级，以及对于拥有69个终结符、258个产生式和226个状态的Pascal文法所打印的实际行数。大写或小写的选项都能被识别。可用的选项有

(1) bnf

打印文法规则。(产生式的数目，Pascal = 258)

(2) cfsm

打印带有LALR(1)超前搜索符号集的文法的特征化有限状态机。(项的数目，Pascal = 2893)

(3) links

如果links和cfsm都被激活，则对一个状态中的每一项，列出其后继项。(项的数目，Pascal = 5809 + cfsm的大小)

779

(4) first

打印所有非终结符的First集。(非终结符的数目，Pascal = 175)

(5) parsetable

以表格形式打印分析动作表。在分析表中，未标记的条目表示到给定状态的转换。标记为L的条目表示由给定编号的产生式所进行的超前归约，标记为R的条目表示简单归约。空白条目指示错误。(状态数乘以终结符数目，Pascal = 1855)

(6) checkreduce

检查文法是否可归约；报告所有不能产生终结符串的符号以及那些从开始符号无法到达的符号。如果文法不可归约，且激活了checkreduce，则不会生成表格。checkreduce通常是被激活的。

(7) resolve

如果给定的文法不是LALR(1)的，仍然会生成分析表，并通过优先选择较早出现在输入中的产生式来逐对解决分析冲突。该选项应当谨慎使用；见“LALRGen中的错误处理”一节的讨论。如果resolve被禁止，则在存在分析冲突的情况下将停止计算分析表。

(8) shortline, longline

控制易被人阅读的输出(vocab、cfsm、parsetable等)中打印行的长度。shortline导致每行少于80个字符(适合屏幕显示)，而longline是132个字符(适合打印机)。shortline和nolongline是同义语，反之亦然。默认选项是shortline。

(9) statistics

打印文法的分类统计信息。对于所有的运行，报告产生式、符号和状态的数目。如果statistics被激活，像基本项的平均数目、闭包图路径长度以及执行时间等额外信息将被打印。(常数，25)

(10) vocab

打印语言的符号。(符号数目，Pascal = 124)

(11) text, binary

由LALRGen所创建的表可以被写成文本形式(字符文件)、二进制形式(整数文件)或同时写成这两种形式。文本输出被写到文件ptableout；二进制输出被写到ptablebin。二进制文件往往更大一些，至少在32位机器上要大一些(在VAX机器UNIX™操作系统下大约大30%)，但它们通常读起来更快。默认选项是text和nobinary。

常量定义段是可选的。如果存在，则以被保留的词法记号*define开始，由一系列定义组成，每条定义都处在单独一行中。每条定义的形式为

<const name> <integer value>

780

其中<const name>是上述的一个词法记号，而<integer value>是一个无符号整数(即仅含数字的词法记号)。该常量可以随后用于任何需要整数的地方：在后续常量定义中，以及用作语义例程编号。注意：该特性并不像它最初看起来那么好，因为LALRGen的输出列表将使用数字值，而不是常量名。

被保留的词法记号*terminals开始终结符号列表。终结符规范段由一系列这样的规范组成，每条规范处于单独一行中。所有终结符都必须出现在该列表中。

词法记号*productions将产生式和终结符分隔开。产生式由一列规则指定，每条规则处于单独一行中。

产生式规范的形式为

```
<lhs> ::= <rhs> <semantic routine #>
```

<lhs>、<rhs>和<semantic routine #>中的任何一个都可以被省略。<lhs>是表示非终结符号的一个词法记号。如果没有<lhs>，则使用前面一条产生式的<lhs>。<rhs>是词法记号串，表示以空白隔开的文法符号。如果没有<rhs>或其中只含有动作符号，则<lhs>推出空串。可以通过以被保留的词法记号"..."开始一行使<lhs>在那些后续的行中继续（仅有产生式可以这样续行）。

<semantic routine #>的形式为

```
## <number>
```

并且指定在识别该产生式时将被调用的语义例程。<number>是一个无符号整数或者预定义常量。如果没有<number>，将使用零。

产生式以*end终止。在所有产生式都被处理之后添加拓广产生式。两个符号<Goal>和$$$以及一个产生式

```
<Goal> ::= <S> $$$
```

被添加到文法中，其中<S>是指定的第一个产生式左部的符号，<Goal>是开始符号，$$$是结束标记。

D.2 LALRGen的输出

由上述选项控制的输出被写到标准Pascal文件output中。此外，包含分析表的文件也被创建。分析表被写到ptableout（文本格式）或ptablebin（二进制格式）中。这些文件可以被指定或重定向，具体情况要依操作系统而定。

不论输出文件被创建为text文件还是binary文件，输出文件的形式都是相同的。在binary文件中，
781 字符值被写成该字符的ord（序号）。

文件ptableout和ptablebin包含下列表：编码的分析动作表、产生式右部的长度、产生式的左部符号、与产生式关联的语义例程编号、给出文法符号字符串表示的符号表以及CFSM中每个状态的条目符号。文法符号被编码为整数。终结符以其在终结符规范段所列出的顺序从1开始编号。结束标记$$$是编号最高的终结符。非终结符以其在文法中出现的顺序来编号，从结束标记编号加1开始。目标符号<Goal>是拥有最高编号的非终结符。文件格式如下所示。

标题行：

第一行给出各个表的大小。它包含CFSM的状态数（numstates）、文法符号的数目（numsymbols）、产生式的数目（numprods）、符号表的字符串大小（stringsize）、分析表中非错误条目数以及一个指示是否为该文法创建了错误修正表的标志（errortables）。如果创建了修正表，则该标志为字符T，否则为字符F。

分析动作：

每个状态的分析动作将以一系列符号/动作对列表的形式给出。每个列表以零/状态数的整数对开头；动作表以一对零结束。如果一个符号没有出现在针对某个状态的列表中，则针对那个状态和符号的动作是出错动作。动作按如下方式编码：

n > 2000:

通过产生式p = n − 2000超前归约。从栈中弹出rhslength(p)个状态但不消耗当前输入符号。

2000 > n > 1000:

通过产生式p = n−1000简单归约。当前符号完成产生式p。弹出rhslength(p)−1个状态并消耗当前输入符号。

1000 > n > 0:

转换到状态n。将n压入栈中。消耗当前输入符号。

rhs长度:

跟随在动作矩阵后面的是numprods个整数，指示相应产生式右部的符号数。

lhs:

接下来是numprods个整数，给出每条产生式左部的符号。

语义编号:

numprods个整数给出与每条产生式相关联的语义例程编号。

字符串表:

符号表信息有两种形式。第一种是索引，由numsymbols个整数对组成。每对中第一个整数是字符串中符号的开始点，第二个整数是符号的长度。跟随在索引后面的是stringsize个字符，每行80个。在二进制形式中，每个字符利用ord（序号）被写成一个字。 782

条目符号:

最后，有numstates个整数，给出进入每个状态时所移进的符号。

D.3 LALRGen中的错误处理

LALRGen输入中的语法错误将由错误恢复例程处理。如果在输入中存在错误，表生成将被中止。

试图使用符号<Goal>和$$$（开始符号和结束标记）将被视为语法错误。

所有终结符必须被列在*terminals段。如果任何终结符没有被列出或一个非终结符没有出现在任何产生式的左边，则该符号会被标记，且不会生成任何表格。类似地，被声明为终结符的符号不能出现在产生式左部。

如果所指定的文法不是LALR(1)的，那么将报告所有的冲突。如果选项resolve被激活，将按产生式的出现次序指定它们的优先级（第一条指定的产生式有最高的优先级）。因此，Pascal和其他语言中的悬空else可以通过

```
<if stmt>   ::= if <expr> then <stmt> else <stmt>
            ::= if <expr> then <stmt>
```

进行分析。

冲突将通过优先选择第一种形式的语句来解决，即把else与最近出现的if相匹配。有相同基本产生式的两项之间的冲突将通过优先进行归约来解决。下列二义文法可以用于分析包含+和id的表达式，左结合被强制执行，因为总是选择归约而不是移进。

```
E   ::= E + E
    ::= id
```

该解决机制应当谨慎使用。必须仔细地检查冲突以保证所采取的分析动作是所希望的动作。例如，在上面的if语句文法中，交换两条产生式的顺序对于生成器是可以接受的，但这将导致归约优先于移进。这将对分析产生灾难性的后果，因为else将永远不会被接受。 783

如果指定的文法被证明对于LALRGen的限制来说太大了，则程序将会打印一条信息来描述所超过的限制且程序将会终止。LALRGen必须随后用增加的限制被重新编译。通常，超过一个限制意味着其

他限制也会被突破，一次增加所有的限制将会节省重新编译的时间。注意：LALRGen按顺序处理终结符、产生式和分析表。因此，如果报告CFSM中的状态数超过了限制，终结符的数量必定还在限制之内，因为它们已被完全处理了。特定文法的某些方面的问题较容易被发现；而其他的问题则必须通过经验和试验来处理。容易确定的是终结符数目、符号数目（terminals+nonterminals）和产生式数目。不容易确定但容易估算的是产生式中的符号总数、穿过闭包图的路径数以及在所有不同符号中的字符总数。经验表明状态数≈产生式数；对于Pascal，项数≈状态数的12倍（≈2500）；对于Pascal，链接数≈项数的2倍（≈5000）。

D.4 使用LALRGen

文法规范被从标准输入文件中读入，而易于阅读的输出则被写到标准输出中。产生的分析表被写入ptableout或ptablebin中。上述名字是在程序头文件中声明的内部名字，并且可以由运行程序的系统环境来修改。

784 图D-2给出LALRGen的一个输入规范示例；图D-3给出将会生成的ptableout文件。

```
Grammar for DCL, a desk calculator language


*ecp
     vocab bnf
     nobinary text  — only generate text files
*define
     <do assn> 2 — semantic actions
     add 5
     subtract 6
*terminals
    id
    constant
    end
    ;
    :=
    (
    )
    +
    -
    *
    /
    write
    read
    ,

*productions
<prog>        ::= <st list> end
<st list>     ::= <st list> ; <st>
              ::= <st>
<st>          ::= id := <expr> ## <do assn>
              ::= <read> ( <id list> ) ## 3
              ::= <write> ( <expr list> ) ## 4
              ::=
<expr>        ::= <expr> + <term> ## add
              ::= <expr> - <term>
                    ... ## subtract
              ::= <term> ## 7
<term>        ::= <term> * <primary> ## 8
              ::= <term> / <primary> ## 9
              ::= <primary> ## 10
<primary>     ::= - <primary> ## 11
              ::= ( <expr> )
```

图D-2 LALRGen输入示例

```
                    ::= id ## 1
                    ::= constant ## 12
<write>             ::= write ## 20
<read>              ::= read ## 21
<id list>           ::= <id list> , id ## 23
                    ::= id ## 24
<expr list>         ::= <expr list>
                        ... , <expr>
                    ::= <expr>

*end
```

图D-2 （续）

```
27 26 24 180 126 F
0 1 13 1019 12 1018 1 6 20 5 22 4 4 2007 3 2007 18 1003 17 3
16 2 0 2 15 1024 0 3 4 27 3 1001 0 4 6 22 0 5 6 19
0 6 5 7 0 7 9 11 6 10 1 1016 2 1017 25 1013 24 9 19 8 0 8 8
14 9 13 4 2004 3
2004 0 9 10 17 11 16 14 2010 9 2010
8 2010 7 2010 4 2010 3 2010 0 10 9 11 6 10 1 1016 2 1017 25
1013 24 9 19 12 0 11 9 11 6 10 1 1016 2 1017 25 1014 0 12 8 14
9 13 7 1015 0 13 9 11 6 10 1 1016 2 1017 25 1013 24 18 0 14 9
11 6 10 1 1016 2 1017 25 1013 24 15 0 15 10 17 11 16 14 2008
9 2008 8 2008 7 2008 4 2008 3 2008 0 16 9 11 6 10 1 1016 2
1017 25 1012 0 17 9 11 6 10 1 1016 2 1017 25 1011 0 18 10 17 11 16
14 2009 9 2009 8 2009 7 2009 4 2009 3 2009 0 19 1 1021 21 20 0
20 14 21 7 1005 0 21 1 1020 0 22 9 11 6 10 1 1016 2 1017 25 1013
24 9 19 24 23 23 0 23 14 25 7 1006 0 24 8 14 9 13 14 2023 7 2023
0 25 9 11 6 10 1 1016 2 1017 25 1013 24 9 19 26 0 26
8 14 9 13 14 2022 7 2022 0 27 13 1019 12 1018 1 6 20 5 22 4 4 2007
3 2007 18 1002
0 0 2 3 1 3 4 4 0 3 3 1 3 3
1 2 3 1 1 1 1 3 1 3 1 2 16 17 17 18 18 18 18 19 19 19 24 24 24 25
25 25 25 22 20 21 21 23 23 26 0 0 0 2
3 4 0 5 6 7 8 9 10 11 0 1 12 20 21 23 24 0 0 -1 76 2 78 8 86 3
89 1 90 2 92 1 93 1 94 1 95 1 96 1
97 1 98 5 103 4 107 1 7 3 108 6 114 9 123 4 127 6 133 6 139 9 148
7 155 11 166 6 172 9 1 6
<Goal>$$$grammarforDCL,DeskCalculatoroneltwo2three3<do assn>2add5
subtract6idcon stantend;::=()←*/writeread,<prog><st list><s
t><expr><read><id list><write><expr list><term><primary>
15 16 17 22 20 1 5 19
24 6 9 19 9 8 24 11 10 24 6 21 14 6 23 19 14 19 4
```

图D-3 给定图D-2的输入文件，由LALRGen产生的ptableout

785
|
786

附录E　LLGen和LALRGen错误修复特性

LLGen和LALRGen不仅能够生成分析表，还可以生成错误修复表。该文档仅描述错误修复特性。所有分析选项和表都与附录C和附录D中所描述的相同。

那些编码在错误修复表中的动作由所指定的上下文无关文法和一系列修复成本来确定。你必须选择修复成本；否则会应用一个默认值。还没有任何一种算法能用于选择最佳的成本集合。然而，这里有一些有益的启发：开始一个结构的符号（如**if**或**begin**）应当拥有相对较高的成本，因为插入或删除这样的符号会导致更多的后续错误。结束结构的符号（如**end**或）可以有较低的成本。非常低的成本可以被赋予仅在有限上下文中出现的符号，例如Pascal中的“..”。以LLGen和LALRGen所需的形式列出的Ada/CS修复成本表见图E-1。

```
*terminals
<id>                    4  2
<numeric literal>       2  2
<character string>      5  2
'                       1  3
|                       1  3
&                       3  1
=                       1  2
:=                      2  2
<less than>             2  2
>                       2  2
<less than or equal>    2  2
>=                      2  2
/=                      2  2
**                      3  1
,                       1  2
.                       3  2
..                      3  2
;                       2  4
:                       2  3
+                       1  1
-                       1  1
*                       2  1
<>                      1  3
=>                      1  3
/                       3  1
(                       8  4
)                       4  6
abs                     3  1
access                  5  3
all                     5  2
and                     3  1
array                   8  2
begin                   8  4
body                    6  4
case                    8  2
constant                6  4
declare                 8  4
else                    5  4
elsif                   5  4
end                     4  5
exception               6  4
exit                    5  2
```

图E-1　Ada/CS的简单修复成本

```
for             8  2
function        4  6
if              8  2
in              6  4
is              1  3
loop            5  4
mod             3  1
not             3  1
null            5  3
of              5  4
or              3  1
others          6  4
out             6  4
package         4  6
pragma          6  2
private         5  3
procedure       4  6
raise           5  2
range           4  2
record          8  2
return          4  2
reverse         5  4
subtype         6  4
then            5  4
type            6  4
use             6  2
when            8  2
while           8  2
```

图E-1 （续）

被保留的词法记号*terminals开始终结符号列表以及其相应的插入和删除成本。每个终结符号的规范的形式为

<terminal symbol> <insert cost> <delete cost>

其中<terminal symbol>是一个词法记号，而<insert cost>和<delete cost>是无符号整数或已定义常量。修复成本是用户提供的值，用于控制和微调修复算法的动作。终结符规范段由一系列这样的规范组成，每条规范处于单独一行中。所有终结符都必须出现在该列表中。

787
788

E.1 LL(1)的错误修复选项和输出格式

LLGen生成可用于FMQ风格LL(1)修复算法（见17.2.4节～17.2.7节）的表格。下列选项是可用的：

errortables: 创建FMQ风格的最小成本错误修复所需的表。该选项通常是被禁止的。如果text选项被激活，则错误修复表被写到etableout中；如果binary选项被激活，则错误修复表被写到etablebin中。（默认选项为text，nobinary。）这些文件可以被指定或重定向，具体情况要依操作系统而定。

如果计算了errortables，可以利用下面所描述的选项将它们打印出来。如果已经计算了单独的表，也可以将它们打印出来，因此下列选项需要errortables被激活。

s: 打印可从每个非终结符推导出的最小成本字符串（S表）。

e: 打印用于从非终结符推导出终结符的最小成本前缀（E表）。该表通常相当大。

所有FMQ风格修复算法都需要S表。该表足够小，可以被很容易地存储在主存中。某些FMQ风格修复算法还需要E表。该表则可能相当大（对于Pascal，大约有40K字节）。因为对于任何一个修复而言，仅有其中一部分是必需的，所以你可能希望将它放在一个文件中并只读取直接需要的那部分。

789

如果已创建文件etableout和etablebin，则它们有如下形式：

标题：　一行，其中包含3个整数：语言中的终结符数目、符号（终结符 + 非终结符）数目以及最大修复成本

成本：　所有终结符号的插入和删除成本

S表：　可从每个非终结符推导出的最小成本字符串（S表）

E表：　从每个非终结符推导出每个终结符所需的最小成本前缀（E表）。前缀值 = ?不包含在该表中。

错误表文件的格式总结于下表中。在此表中，名字对应于文件中的整数。(string) *<name>意指把(string)的内容重复<name>次。(string)*意指该字符串重复未知次数（该列表以-1终止）。"name:" 标记文件的逻辑部分而不实际出现在文件中。表的注释（不出现在文件中）被置于{和}中。

标题：　NumberOfTerminals NumberOfSymbols Infinity

成本：　(InsertCost DeleteCost)*NumberOfTerminals

S表：　(Cost Length (InsertSymbol)*Length

　　或-1{如果与前一项相同；不保证进行该优化}

　)*NumberOfNonterminals

　{其中NumberOfNonterminals = NumberOfSymbols−NumberOfTerminals}

E表：　0 TerminalSymbol 0

　(NonterminalSymbol Cost Length (InsertSymbol)*Length)*

　　{其中上面的重复由下一个

　　0 TerminalSymbol 0序列或

　　终结序列0 0 0来结束}

　)*NumberOfTerminals

　0 0 0

E.2 LALR(1)的错误修复选项和输出格式

LALRGen生成可以和基于延拓的LALR(1)修复算法（见17.2.10节 ~ 17.2.11节）一同使用的表。下列选项是可用的：

`errortables:`　创建基于延拓的错误修复所需的表。该选项通常是被禁止的。如果text选项被激活，则错误修复表被写到etableout中；如果binary选项被激活，则错误修复表被写到etablebin中。（默认选项为text，nobinary。）这些文件可以被指定或重定向，具体情况依操作系统而定。

790

如果计算了errortables，可以利用下面所描述的选项将它们打印出来。如果已经计算了单独的表，也可以将它们打印出来，因此下列选项需要errortables被激活。

`ca:`　打印延拓动作表。

如果已创建文件etableout和etablebin，则它们有如下形式：

标题：　一行，其中包含3个整数：语言中的终结符数目、CFSM状态数以及最大修复成本

成本：　所有终结符号的插入和删除成本

ca表：　相应于每个状态的延拓动作

错误表文件的格式总结于下表中。记号m * [n]意指一列包含n项的类型m的项列表。

标题：　NumberOfTerminals NumberOfStates Infinity

成本: (InsertCost DeleteCost) * [NumberOfTerminals]

ca表: ContinuationAction * [NumberOfStates]

延拓动作表中的条目按如下方式编码。令ca = catable(state)。如果ca为

1 .. NumberOfTerminals —— 插入下标为ca的终结符号

1000 < CA < 2000 —— 用产生式ca－1000来简单归约

791 > 2000 —— 用产生式ca－2000来超前归约

附录F 编译器开发实用工具

许多以Pascal语言编写的实用工具程序可用于辅助编译器的开发和调试。其中包括一个双偏移压缩例程和一个目标机器解释器。可用的例程还有：执行算术运算并能同时捕捉错误（例如溢出）的例程，将位串分别在整数和实数变量之间进行转换并绕过强类型检查的例程，以及压缩和解压缩Ada/CS机器指令和字符串的例程。

F.1 数组压缩工具

ArrayComp是一个实用工具程序，它读取一个稀疏矩阵并为利用双偏移索引（见17.1节）访问它而对其进行压缩。它可以用于在由ScanGen、LLGen和LALRGen产生词法分析表和语法分析表之后对这些表进行压缩。

从标准输入文件中读取输入。所读的数据为

```
N M Default
InArray（元素为行主序）
```

N和M是要被压缩的数组的边界。即，InArray是**array**(1..N; 1..M) **of** Integer。Default是一个元素，它会被假定为压缩表示中的默认值。为使这个压缩有效，InArray的大多数元素必须为默认值。

压缩形式的InArray被写到文件outtable中。所写的数据为

```
N M Default CompressedLen
Row（包含N个值）
OutArray（包含CompressedLen个条目）
```

压缩表以如下形式访问：

为访问InArray(i, j):

```
if OutArray(Row(i)+2*i) = j
then InArray(i,j) = OutArray(Row(i)+2*i+1)
else InArray(i,j) = Default
```

描述数据结构大小和所获得的压缩度的统计数据被写到标准输出文件中。

F.2 Ada/CS目标机器

Ada/CS目标机器拥有与IBM 360/370系列类似的简单的面向寄存器的体系结构。为该机器生成代码简化了编译器开发，因为它提供了一个带有调试选项的解释器。此外，用于编译器开发的实际机器和操作系统的细节被隐藏起来，使得更容易利用任何可用的机器。

Ada/CS目标机器有若干个字数的存储器（依赖于所购买的模型），编号从0到MaxAdr。每个字包含32位。（Ada/CS机器不同于VAX、PDP-11和IBM机器，这些机器都是每字节单独编址的。）

有64个寄存器，它们覆盖了主存的前64个字（0~63）：寄存器i存储在Memory[i]中。指令是32位长，其形式为

操作码（Op Code）	寄存器（Register）	基址（Base）	位移（Displacement）
6位	6位	6位	14位

一条指令的有效地址（EAdr）定义如下：

```
EAdr =
  if Base = 0 then
    Displacement
  else
    Displacement + Memory[Base]
  end
```

Displacement和Base是指令相应域的内容。我们遵循IBM 360的约定，Base = 0意味着不使用基址寄存器，而不是指基址寄存器0，因此仅有寄存器1-63可以用作基址寄存器。Displacement被视为有符号的14位量，可通过最左端一位的复制扩展到整字量。随后其值（用2的补码算术）与指定基址寄存器（如果有的话）的内容相加。负的Displacement是合法的而且非常有用。

793 我们将令R代表Memory[Reg]，这是由指令的Reg域所选择的寄存器的内容；令M代表Memory[EAdr]。内存单元中最右端的位是最低有效位；最左端的位是最高有效位。

我们使用下列数据格式：

整数：32位，2的补码。

实数：与运行Ada/CS目标机器解释器的机器上32位实数的格式相同。

布尔数：存储在一个字的最低位（最右端一位）。

字符：单个字符，表示为那些运行Ada/CS目标机器解释器的机器所使用的字符集，存放在一个字的低有效位部分（最右端）。

字符串：一个（32位）字存储长度，后面跟随字符，每四个压缩在一个字中，（在字内）从左向右存放。如果字符数不是4的倍数，则最后一个字内右部以零填充。

下列机器操作是可用的：

操作码	助记符	定义
1	FIX	将实数转换为整数：R := Integer(R)
2	NEGI	R := −R（整数）
7	NEGR	R := −R（实数）
3	FLOAT	将整数转换为实数：R := Float(R)（有限精度）

4	NOTL	**if** EAdr > 0 **then**
		取R最右边EAdr位的1的补码
		else
		取R的最左边Abs(EAdr)位的1的补码
		end if

6	B	PC := EAdr（跳转到EAdr）
8	BZ	**if** R = 0 **then** PC := EAdr **end if**
10	BNZ	**if** R ≠ 0 **then** PC := EAdr **end if**
12	BGZI	**if** R > 0（整数）**then** PC := EAdr **end if**
13	BGZR	**if** R > 0（实数）**then** PC := EAdr **end if**
14	BNGZI	**if** R < 0（整数）**then** PC := EAdr **end if**
15	BNGZR	**if** R < 0（实数）**then** PC := EAdr **end if**
16	BLZI	**if** R < 0（整数）**then** PC := EAdr **end if**
17	BLZR	**if** R < 0（实数）**then** PC := EAdr **end if**

18	BNLZI	**if** R > 0（整数）**then** PC := EAdr **end if**
19	BNLZR	**if** R > 0（实数）**then** PC := EAdr **end if**
20	BAL	转移并链接：R := PC; PC := EAdr
21	BKT	块移动： R = 源地址 EAdr = 目标地址 Memory[Reg + 1] = 移动的字数 操作数从左向右处理 每次移动一个字
22	SVC	超级用户调用：解释器返回Result = EAdr。 EAdr = 0意味着停机。
23	TRNG	测试区间，若下式不成立则中断： Memory[EAdr] ≤ R ≤ Memory[EAdr + 1]
24	SHL	R := R左移EAdr位（以零填充）
26	SHR	R := R右移EAdr位（以零填充）
28	LDA	R := EAdr
30	ST	M := R
31	STPC	M := PC
32	LD	R := M
34	ADDI	R := R + M（整数）
35	ADDR	R := R + M（实数）
36	SUBI	R := R - M（整数）
33	SUBU	R := R - M（整数） 减法是不进行检查的，因而不会导致溢出异常。 在所有情况下，结果的符号都是正确的，尽管如果SUBI产生溢出，则其结果将会不正确。
37	SUBR	R := R - M（实数）
38	MULI	R := R * M（整数）
39	MULR	R := R * M（实数）
40	DIVI	R := R / M（整数）
41	DIVR	R := R / M（实数）
42	EXP	R := R ** M（整数）
43	EXPRE	R := R ** M（实数 ** 整数）
44	ANDL	R := R **and** M（位运算）
46	ORL	R := R **or** M（位运算）
48	BXOR	R := R **xor** M（位运算）

输入是自由格式的。每行中可能有多个数据项。无效项将产生一个异常。有下列输入操作可用：

47	RDLN	跳到下一个输入行的开始
49	RDCH	把一个字符读到M中（M的其余位被清零）
50	RDL	**if** 下面5个输入字符是FALSE（忽略大小写） **then** 将M的最右一位清零

794

elsif 下面4个输入字符是TRUE（忽略大小写）
then 置M的最右一位为1
else 产生一个无效输入异常
end if

51 RDR 将一个实数读入M
52 RDI 将一个整数读入M
53 RSTR 将一个字符串读入M。
该字符串可以出现在输入行的任何位置。
它必须被置于引号（'）中。

795

有下列输出操作可用：

54 WL **if** M的最右一位 = 1
then 打印TRUE
else 打印FALSE
end if
55 WR 以实数格式打印M
56 WI 以整数格式打印M
57 WRCH 以字符格式打印M
5 WBITS 以位串格式打印M
58 WSTR 以字符串格式打印M
如果必要，插入换行符。
60 SKP **if** EAdr < 0 **then**
writeln
elsif 0 < EAdr < 57 **then**
跳过EAdr行
elsif 57 < EAdr **then**
跳到新的一页
end if

注意：

(1) WL、WR、WI和WRCH：
如果Reg = 0，那么使用默认的输出格式；否则，使用R作为输出域的最小宽度（如果需要，在左边填充空白符）。

(2) WL、WR、WI、WRCH和WBITS：
如果输出项不能填满整个当前行，则跳至下一行。

(3) WSTR：
if Reg = 0 **then** L := len(str) **else** L := R **end if**。打印max(0, L − len(str))个空白符，后面跟着串的最左的min(L,len(str))个字符。

提供了下列字符串操作：

59 CAT M := S1和S2的连接，
其中Memory[Reg]指向S1，而Memory[Reg + 1]指向S2
61 BSUBSTR M := S中从位置P开始、长度为L的子串的复本。
S的地址在R中，P和L分别存储于

Memory[Reg + 1]和Memory[Reg + 2]中。

62 STEQ S1在Memory[R]处。
if S1 = M（作为字符串）
then PC := PC + 1;（跳过下一条指令）
end if

63 STLSS 同上，但在S1 < M（以词典序）时跳过下一条指令 796

下列硬件异常将导致Ada/CS机器解释器返回到调用程序，并将作为标志的负值放入Result参数中。

-1 无效指令（0, 9, 11, 25, 27, 29或45）
-2 地址越界
-3 上溢（实数或整数）
-4 定时器中断
-5 输入文件结束
-6 被零除（实数或整数）
-8 实数下溢
-9 TRNG指令失败
-10 无效输入项
-11 EXP的第二个操作数为负数

Ada/CS机器解释器模拟一个虚拟计算机，其内存为MaxAdr + 1个字的数组，以Memory(0..MaxAdr)表示。为运行该解释器，分配一个整数数组作为机器的模拟内存，对其进行适当的初始化并调用外部过程

```
procedure interp(Store : in out Memory; LastAdr : in integer;
                 InFile : in out text; PC, Time : in out integer;
                 ReturnCode : out integer; Trace : in Boolean);
```

其中

```
Memory : array (0..MaxAdr) of integer;
```

Memory: 模拟的内存。
LastAdr: Memory中可被访问而不会导致非法地址错误的最后一个地址。必须满足LastAdr ≤ MaxAdr。
InFile: 一个打开的文件，Ada/CS机器从中获得（RDLN、RDCH以及类似指令的）输入。
PC: 初始的程序计数器（内存中将被执行的第一条指令的地址）。如果发生异常或超级用户调用（SVC）指令，PC将指向该指令。
Time: 时间限制。每次执行一条指令时，Time减1。当Time = 0时，发生计时器中断（且PC指向将被执行的下一条指令）。
Result: 返回代码：如果Result > 0，则Result是SVC指令的EAdr。否则，Result是中断码。
Trace: 指出一个应当产生指令踪迹。

F.3 其他实用工具

算术例程：

```
procedure integerop(Op : oper; A,B : integer; C : out integer; Flag : out status);
```
797

```
procedure realop(Op : oper; A,B : real;  C : out real; Flag : out status);
```

其中，

```
oper = (AddOper, SubOper, MulOper, DivOper);
status = (OK, Overflow, Underflow, ZeroDivide);
```

如果Flag = OK，那么C = A Op B。

否则，C的值是不可预测的。

压缩和解压缩例程：

```
function packinst(OpCode, Reg, Base, Offset : integer) return integer;
procedure unpackin(Inst : integer; OpCode, Reg, Base, Offset : out integer);
```

对于压缩，所有参数都被截断至适当的长度。对于解压缩，Offset是以符号位扩展，其他所有参数都以零扩展。

```
function packchar(C1,C2,C3,C4 : char) return integer;
procedure unpackch(n : integer; C1,C2,C3,C4 : out char);
```

C1处于字的高端（最左端），C4处于低端（最右端）。（这也是Ada/CS机器所期望的顺序。）

忽略强类型检查的例程：

```
function intcast(R : shortreal) return integer;
function realcast(I : integer) return shortreal;
```

798

参考文献

Aho, Alfred V.; Ganapathi, M.; and Tjiang, S. 1989. "Code generation using tree matching and dynamic programming." *ACM Transactions on Programming Languages and Systems* 11(4): 491–516.

Aho, Alfred V.; Johnson, S. C.; and Ullman, Jeffrey D. 1975. "Deterministic parsing of ambiguous grammars." *Communications of the ACM* 18(8): 441–52.

———. 1977. "Code generation for expressions with common subexpressions." *Journal of the ACM* 24(1): 146–60.

Aho, Alfred V.; Sethi, Ravi; and Ullman, Jeffrey D. 1985. *Compilers: Principles, Techniques, and Tools*. Reading, Mass.: Addison-Wesley.

Aho, Alfred V., and Ullman, Jeffrey D. 1972. *The Theory of Parsing, Translation and Compiling, Vol. 1*. Englewood Cliffs, N.J.: Prentice-Hall.

———. 1977. *Principles of Compiler Design*. Reading, Mass.: Addison-Wesley.

Aigrain, P.; Graham, Susan L.; Henry, R. R.; McKusick, K.; and Pelegri-Liopart, E. 1984. "Experience with a Graham-Glanville style code generator." *SIGPLAN Notices* 19(6): 13–24.

ANSI. 1989. *Standard for the programming language C*. New York, New York: American National Standards Institute.

Appel, Andrew W. 1985. "Semantics-directed code generation." In *Twelfth Annual ACM Symposium on Principles of Programming Languages* 315–24.

Aretz, F.E.J. 1989. "A new approach to Earley's parsing algorithm." *Science of Computer Programming* 12: 105–21.

Archer, James, and Conway, Richard. 1981. "COPE: a cooperative programming environment." TR 81-459. Ithaca, N.Y.: Cornell University.

Baker, T. 1982. "A one-pass algorithm for overload resolution in Ada." *ACM Transactions on Programming Languages and Systems* 4(4): 601–14.

Ball, J. 1979. "Predicting the effects of optimization on a procedure body." *SIGPLAN Notices* 14(8): 214–20.

Bell, J. R. 1973. "Threaded code." *Communications of the ACM* 16(6): 370–72.

Bjorner, D., and Jones, C. 1978. "The Vienna Development Method: the metalanguage." In *Lecture Notes in Computer Science, #61*. New York: Springer-Verlag.

Bochmann, G. V. 1976. "Semantic evaluation from left to right." *Communications of the ACM* 19(2): 55–62.

Brodie, L. 1981. *Starting FORTH*. Englewood Cliffs, N.J.: Prentice-Hall.

Burke, Michael, and Fisher, Gerald A. 1982. "A practical method for syntactic error diagnosis and repair." *SIGPLAN Notices* 17(6): 67–78.

Cattell, R.G.G. 1980. "Automatic derivation of code generators from machine

descriptions." *ACM Transactions on Programming Languages and Systems* 2(2): 173–90.

Chaitin, G. J. 1982. "Register allocation and spilling via graph coloring." *SIGPLAN Notices* 17(6): 98–105.

Christopher, T. W.; Hatcher, P. J.; and Kukuk, R. C. 1984. "Using dynamic programming to generate optimized code in a Graham-Glanville style code generator." In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction* 25–36.

Cichelli, R. J. 1980. "Minimal perfect hash functions made simple." *Communications of the ACM* 23(1): 17–19.

Cocke, J. 1970. "Global common subexpression elimination." *SIGPLAN Notices* 5(7): 20–24.

Cocke, J., and Schwartz, J. T. 1970. *Programming Languages and Their Compilers: Preliminary Notes, Second Revised Version.* New York: Courant Institute of Mathematical Science.

Conway, R. W. 1972. *The ASAP Systems Reference Manual.* Ithaca, N.Y.: Compuvisor, Inc.

Conway, R. W., and Wilcox, T. R. 1973. "Design and implementation of a diagnostic compiler for PL/I." *Communications of the ACM* 16: 169–79.

Cormack, G. V. 1981. "An algorithm for the selection of overloaded functions in Ada." *SIGPLAN Notices* 16(2): 48–52.

Davidson, J. W., and Fraser, C. W. 1982. "Eliminating redundant object code." *In Ninth Annual ACM Symposium on Principles of Programming Languages* 128–32.

———. 1984. "Automatic generation of peephole optimizations." *SIGPLAN Notices* 19(6): 111–16.

DeRemer, Frank L. 1969. "Practical translators for LR(k) languages." Ph.D. dissertation, M.I.T.

———. 1971. "Simple LR(k) grammars." *Communications of the ACM* 14: 453–60.

DeRemer, Frank L., and Pennello, T. 1982. "Efficient computation of LALR(1) look-ahead sets." *ACM Transactions on Programming Languages and Systems* 4(4): 615–49.

Dion, Bernard A. 1982. *Locally Least-cost Error Correctors for Context-free and Context-sensitive Parsers.* Ann Arbor: UMI Research Press.

800

Druseikis, Frederick C., and Ripley, G. David. 1976. "Extended SLR(k) parsers for error recovery and repair." In *Proceedings of the ACM Annual Conference* 396–400.

Earley, J. 1970. "An efficient context-free parsing algorithm." *Communications of the ACM* 13(2): 94–102.

Farrow, R. 1982. "Linguist-86: yet another translator writing system based on attribute grammars." *SIGPLAN Notices* 17(6): 160–71.

Fischer, C. N.; Johnson, G. F.; Mauney, J.; Pal, A.; and Stock, D. L. 1984. "The POE language-based editor project." In *ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*

Fischer, Charles N., and LeBlanc, Richard J. 1977. *UW-Pascal Reference Manual.* Madison, Wis.: Madison Academic Computer Center.

———. 1980. "The implementation of run-time diagnostics in Pascal." *IEEE Transactions on Software Engineering* SE-6(4): 313–19.

Fischer, Charles N.; Mauney, Jon; and Milton, Donn R. 1979. "A locally least-cost LL(1) error corrector." Technical Report #371. Madison, Wis.: University of Wisconsin.

Fischer, Charles N.; Milton, Donn R.; and Quiring, Sam B. 1980. "Efficient LL(1) error correction and recovery using only insertions." *Acta Informatica* 13(2): 141–54.

Fleck, A. C. 1976. "On the impossibility of content exchange through the by-name parameter transmission mechanism." *SIGPLAN Notices* 11(11): 38–41.

Fong, A. C., and Ullman, Jeffrey D. 1976. "Induction variables in very high-level languages." In *Third Annual ACM Symposium on Principles of Programming Languages* 104–12.

Fraser, C. W., and Davidson, J. W. 1980. "The design and application of a retargetable peephole optimizer." *ACM Transactions on Programming Languages and Systems* 2(2): 191–202.

Ganapathi, M., and Fischer, Charles N. 1985. "Affix grammar-driven code generation." *ACM Transactions on Programming Languages and Systems* 4(7): 560–99.

Ganzinger, H.; Giegerich, R.; Moncke, U.; and Wilhelm, R. 1982. "A truly generative semantics-directed compiler generator." *SIGPLAN Notices* 17(6): 172–84.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W.H. Freeman.

Glanville, R. S., and Graham, Susan L. 1978. "A new method for compiler code generation." In *Fifth Annual ACM Symposium on Principles of Programming Languages*

Goos, G., and Wulf, W. A. 1981. "Diana Reference Manual." CMU-CS-81-101. Pittsburgh: Department of Computer Science, Carnegie-Mellon University.

Graham, Susan L.; Haley, Charles B.; and Joy, William N. 1979. "Practical LR error recovery." *SIGPLAN Notices* 14(8): 168–75.

Graham, Susan L.; Harrison, Michael A.; and Ruzzo, Walter L. 1980. "An improved context-free recognizer." *ACM Transactions on Programming Languages and Systems* 2(3): 415–62.

Graham, Susan L.; Joy, William; and Roubine, Olivier. 1979. "Hashed symbol tables for languages with explicit scope control." *SIGPLAN Notices* 14(8): 50–57.

Gray, Robert W. 1988. "γ-GLA: A generator for lexical analyzers that programmers can use." In *Proceedings of the Summer Usenix Conference*.

Gries, David. 1981. *The Science of Programming*. New York: Springer-Verlag.

Hennessy, J. 1982. "Symbolic debugging of optimized code." *ACM Transactions on Programming Languages and Systems* 4(3): 323–44.

Hopcroft, John E., and Ullman, J. D. 1969. *Formal Languages and Their Relation to Automata*. Reading, Mass.: Addison-Wesley.

———. 1979. *Introduction to Automata Theory, Languages, and Computation*. Reading, Mass.: Addison-Wesley.

801

Horwitz, L. P.; Karp, R. M.; Miller, R. E.; and Winograd, S. 1966. "Index register allocation." *Journal of the ACM* 13(1): 43–61.

Hsu, Wei-Chung. 1987. "Register allocation and code scheduling for load/store architectures." Ph.D. dissertation, University of Wisconsin, Madison.

James, L. R. 1972. "A syntax-directed error recovery method." Technical Report CSRG-13. Toronto: Computer Systems Research Group, University of Toronto.

Jacobsen, Van. 1987. "Tuning UNIX Lex, or, it's not true what they say about Lex." In *Proceedings of the Winter Usenix Conference* 163–64.

Jazayeri, M.; Ogden, W. F.; and Rounds, W. C. 1975. "The intrinsic exponential complexity of the circularity problem for attribute grammars." *Communications of the ACM* 18(12): 697–706.

Jazayeri, M., and Walter, K. G. 1975. "Alternating semantic evaluator." In *Proceedings of the ACM Annual Conference* 230–34.

Johnson, S. C. 1975. "Yacc—yet another compiler compiler." C.S. Technical Report #32. Murray Hill, N.J.: Bell Telephone Laboratories.

———. 1978. "A portable compiler: theory and practice." In *Fifth Annual ACM Symposium on Principles of Programming Languages* 97–104.

Johnsson, R. K. 1975. "An approach to global register allocation." Ph.D. dissertation, Carnegie-Mellon University.

Kernighan, Brian W., and Ritchie, Dennis M. 1978. *The C Programming Language*. Englewood Cliffs, N.J.: Prentice-Hall.

802

———. 1988. *The C Programming Language*, 2d ed. Englewood Cliffs, N.J.: Prentice-Hall.

Kim, J. 1978. "Spill placement optimization in register allocation for compilers." IBM Research report RC 7251.

Knuth, Donald E. 1965. "On the translation of languages from left to right." *Information and Control* 8(6): 607–39.

———. 1968. "Semantics of context-free languages." *Mathematical Systems Theory* 2(2): 127–45.

———. 1973. *The Art of Computer Programming*, 2d ed. Vol. 1. Reading, Mass.: Addison-Wesley.

LeBlanc, Richard J., and Mongiovi, Roy J. 1983. "A practical method for automated LR error repair." Technical Report GIT-ICS-83/18. Atlanta: School of Information and Computer Science, Georgia Institute of Technology.

Lesk, M. E., and Schmidt, E. 1975. "Lex—a lexical analyzer generator." In *UNIX Programmer's Manual* 2. Murray Hill, N.J.: AT&T Bell Laboratories.

Lewis, P. M.; Rosenkrantz, D. J.; and Stearns, R. E. 1976. *Compiler Design Theory*. Reading, Mass.: Addison-Wesley.

Logothetis, George, and Mishra, Prateek. 1981. "Compiling short-circuit boolean expressions in one pass." *Software-Practice and Experience* 11: 1197–1241.

Magnusson, K. 1982. "Identifier references in Simula 67 programs." *Simula Newsletter* 10(2):

Mauney, Jon. 1982. "Least-cost error repair using extended right context." Technical Report 495. Madison, Wis.: University of Wisconsin.

Mauney, Jon, and Fischer, Charles N. 1981. "An improvement to immediate error detection in Strong LL(1) parsers." *Information Processing Letters* 12(5): 211–12.

McCarthy, John. 1965. *Lisp 1.5 Programmer's Manual.* Cambridge, Mass.: MIT Press.

McKeeman, W. M. 1965. "Peephole optimization." *Communications of the ACM* 8(7): 443–44.

Mickunas, M. D., and Modry, J. A. 1978. "Automatic error recovery for LR parsers." *Communications of the ACM* 21: 459–65.

Nestor, J. R.; Wulf, W. A.; and Lamb, D. A. 1981. "IDL—interface description language: formal description." Technical Report CMU-CS-81-139. Pittsburgh: Department of Computer Science, Carnegie-Mellon University.

Notkin, D. 1985. "The Gandalf project." *Journal Systems Software* 5: 91–106.

Pager, D. 1977. "A practical general method for constructing LR(k) parsers." *Acta Informatica* 7: 249–68.

803

Patterson, D. 1985. "Reduced instruction set computers." *Communications of the ACM* 28(1): 8–21.

Paxson, Vern. 1990. "Flex users manual." Ithaca, N.Y.: Cornell University.

Pennello, Thomas J., and DeRemer, Frank L. 1978. "A forward move algorithm for LR error recovery." In *Fifth Annual ACM Symposium on Principles of Programming Languages* 241–54.

Perkins, D. R., and Sites, R. L. 1979. "Machine independent Pascal code optimization." *SIGPLAN Notices* 14(8): 201–7.

Pratt, T. W. 1975. *Programming Languages: Design and Implementation.* Englewood Cliffs, N.J.: Prentice-Hall.

Ripley, G. David, and Druseikis, Frederick C. 1978. "A statistical analysis of syntax errors." *Computer Languages* 3: 227–40.

Roehrich, Johannes. 1980. "Methods for the automatic construction of error-correcting parsers." *Acta Informatica* 13(2): 115–39.

Rowland, Bruce R. 1977. "Combining parsing and the evaluation of attributed grammars." Ph.D. dissertation, University of Wisconsin.

Sethi, Ravi. 1983. "Control flow aspects of semantics-directed compiling." *ACM Transactions on Programming Languages and Systems* 5(4): 554–95.

Sethi, Ravi, and Ullman, Jeffrey D. 1970. "The generation of optimal code for arithmetic expressions." *Journal of the ACM* 17(4): 715–28.

Soisalon-Soininen, E. 1982. "Inessential error entries and their use in LR parser optimization." *ACM Transactions on Programming Languages and Systems* 4(2): 179–95.

Stallman, Richard M. 1989. "Using and porting GNU CC." Free Software Foundation, Inc.

Steel, T. B., Jr. 1961. "A first version of UNCOL." *Proceedings of the WJCC* 19: 371–78.

Stroustrup, Bjarne. 1986. *The C++ Programming Language.* Reading, Mass.: Addison-Wesley.

Sussman, G. 1981. "Scheme-79, Lisp on a chip." *IEEE Computer* 14(7): 10–21.

Szymanski, T. G. 1978. "Assembling code for machines with span-dependent instructions." *Communications of the ACM* 21(4): 300–8.

Tanenbaum, A. S.; van Straveren, H.; and Stevenson, J. W. 1982. "Using peephole optimization on intermediate code." *ACM Transactions on Programming Languages and Systems* 4(1): 21–36.

Tanenbaum, A. S. 1974. "Implications of structured programming for machine architecture." *Communications of the ACM* 21(3): 237–42.

Tarjan, R. E., and Yao, A. C. 1979. "Storing a sparse table." *Communications of the ACM* 22(11): 606–11.

804

Teitelbaum, T., and Reps, Thomas. 1981. "The Cornell program synthesizer: a syntax-directed programming environment." *Communications of the ACM* 24(9): 563–73.

Valiant, L. 1975. "General context-free recognition in less than cubic time." *Journal Computer and Systems Sciences* 10(2): 308–15.

Wand, M. 1982. "Deriving target code as a representation of continuation semantics." *ACM Transactions on Programming Languages and Systems* 4(3): 496–517.

WATFIV. 1981. *WATFIV Users Guide.* Waterloo, Ontario: University of Waterloo.

Watt, David A. 1977. "The parsing problem for affix grammars." *Acta Informatica* 8: 1–20.

Wetherell, Charles, and Shannon, A. 1981. "LR—automatic parser generator and LR(1) parser." *IEEE Transactions on Software Engineering* SE-7(3): 274–78.

Wilcox, T. R. 1971. "Generating machine code for high-level programming languages." Technical Report 71-103. Ithaca, N.Y.: Cornell University.

Wirth, Niklaus. 1976. *Algorithms + Data Structures = Programs.* Englewood Cliffs, N.J.: Prentice-Hall.

Wirth, Niklaus, and Weber, H. 1966. "Euler: a generalization of Algol and its formal definition: Part 1." *Communications of the ACM* 9(1): 13–23.

Wulf, W. 1981. "Compilers and computer architecture." *IEEE Computer* 14(7): 41–7.

Zellweger, Polle. 1983. "An interactive high-level debugger for control-flow optimized programs." *SIGPLAN Notices* 18(8): 159–71.

805

索　引

索引中的页码为英文原书的页码，与书中边栏页码一致。

A

abstract data type（抽象数据类型），273，538
abstract definition（抽象定义），255
abstract interface（抽象接口），379，537
abstract semantic stack（抽象语义栈），236
abstract syntax tree（抽象语法树），217，249-250，252，325，518，534-536，544
abstract syntax（抽象语法），250，539
access types（存取类型），364-366，411，413
action symbol（动作符号），39-43，121-123，177-178，226-229，514-515，598
action-controlled（动作控制的），336
action-controlled semantic stack（动作控制的语义栈），233-236，244-245，323
activation pointer（活动指针），305-306，308-309，311
activation record（活动记录），289-296，305-311，349-351，455-456，497-505
activation register（活动寄存器），305
address mode（地址模式），21，221，465，552-553，556-558，570，573，594，598
aggregate（聚合），76，335，337，416
Aho，A.V.（阿尔弗莱德·阿霍，美国著名计算机科学家），175，185，204，580，585，596，641
alias（别名），13-14，406-407，560-563，567，586-587，631，654
all path data flow analysis（全数据流分析），654，656
allocatable register（可分配寄存器），549，562，568
ambiguous grammar（二义文法），127，139，184，198，206
analysis（分析），8，110，216，219-224，260，510，562，712
any path data flow analysis（单路径数据流分析），652，654，656
Appel，A.W.（人名），14
Archer，J.（人名），19
argument pointer（实参指针），502
array declarations（数组声明），338-340，349-358
array references（数组引用），389-406
Aretz，F.E.J.（人名），209
associativity（结合性），31，184，187-188，217，380，578，611
attribute evaluation rules（属性计算规则），512
attribute grammar（属性文法），511-534
attribute propagation（属性传播），218
attribute（属性），10-12，42，217-219，254-256，320-323，327-328，511-545，597
attribute-based code generation（基于属性的代码生成），597-601
attributed syntax tree（属性语法树），512，524
augmenting production（拓广产生式），145，150，154，182-183，190-191，206
available expression data flow analysis（可用表达式数据流分析），658
available on exit（退出时可用），651
axiomatic definition（公理定义），13

B

backpatching（回填），430，442，453，480-481，532
backtrack（回溯），592，605
backward analysis（后向分析），175
backward move（后向移动），724
backward-flow analysis（后向数据流分析），654，656
Baker，T.（人名），420
Ball，J.（人名），622
base register（基址寄存器），303，390，557，599，607
base-plus-index address mode（基址加变址地址模式），607，610
basic block（基本块），555-556
basis item（基本项），718
Bell，J.R.（人名），548
best-fit（最佳适配），300，690，725
binary trees（二叉树），256-257
bit-map（位图），301，499
Bjorner,D.（人名），13
blank token（空白记号），62
block structured symbol tables（块结构符号表），261-266
Bockmann，G.V.（人名），520
bootstrap（自举），4

bottom-up（自底向上），141
bottom-up parser（自底向上的语法分析器），93，98，144
boundary tag（边界标记），301
bounded state（有界状态），215
Brodie，L.（人名），548
buffering（缓冲，缓存），69，75，77，191，695，698
Burke，M.（人名），706，724

C

cactus stack（cactus栈），292，626
call graph（调用图），622，625-626，628
call optimization（调用优化），673
case analysis（实例分析），547，553
Cattell, R.G.G.（人名），605
CFSM（Characteristic Finite State Machine，特征化有限状态自动机），147-148
chained references（链式引用），442
Chaitin，G.J.（人名），570
character class（字符类），61-62，68，72
Christopher, T.W.（人名），209，596
Cicheli, R.J.（人名），71
circular first fit（循环首次适配），300
circularity（环），513
closed subroutine（封闭子例程），622-623，625
closure（闭包），53-54，146-147，156，307，719-720，723
Cocke, J.（人名），559，667
code generation（代码生成）
  from dags（从dag生成代码），578-589
  from trees（从树生成代码），574-578
  interpretive（解释性代码生成），555-572
  simple（简单代码生成），551-555
code generation attribute（代码生成属性），534，536
code generation IR（代码生成中间表示），594
code generator generators（代码生成器的生成器），589-601
code hoisting（代码提升），656
codeword（代称），405
cognate（同心的），166-167，206
collision（冲突），257-258，687
coloring algorithm（着色算法），570
comments（注释），24，26，56，62，77-79，129-130，179，181
common prefix（公共前缀），123，126，139，144，199
common subexpression analysis（公共子表达式分析），559-562，578，620-621，657
compaction（压缩），61，300，302，315，461-462，685-686，690，725
compiler directive（编译器指示），72-73
compiler generator（编译器生成器），12
compiler writing tool（编译器编写工具），12
compiler-compiler（编译器的编译器），12，183
computation dag（dag计算），578
concrete syntax（具体语法），545
conditional expression（条件表达式），48-49，72，177，615，617
configuration set（项目集），147，156
configuration（项目），145，155，202
conflict graph（冲突图），570
conflict resolution（冲突解决），132
constant declaration（常量声明），340-343
constant folding（常量合并），48
constrained array（约束数组），338，347，349，352-355，357，417
constraint（约束），333，345-349，357，445，598，625
context-free grammar（上下文无关文法），10，91-100，594
context-free language（上下文无关语言），92
context-free parser（上下文无关语法分析器），206
context-sensitive grammar（上下文相关文法），95
continuation address（连续地址），458
continuation（连续性），715-720
control flow analysis（控制流分析），620
Conway, R.W.（人名），18，19，649
copy propagation（复写传播），601，641，643，654-655，659-660，674
copy rule（拷贝规则），514-515，526
copy symbol（拷贝符号），525-526，528
core（核心），166-167，170，194-196，202
Cormack, G.V.（人名），419
correct prefix property（正确前缀性质），692
cross-compiler（交叉编译器），21
current scope（当前作用域），74，261

D

dag evaluation（dag计算），580
dag（有向无环图），250，545，578-587，615
dag-traversal algorithm（dag遍历算法），581
dangling pointer（悬空指针），297，412
data area（数据区），5，288-289，349，355，381，389-401，410

data flow analysis（数据流分析），572，618-621，655，660，665-667，674
data flow graph（数据流图），620，667
Davidson, J.W.（人名），574，602
dead register（非活跃寄存器），551
dead variable（非活跃变量），645，658，675
deep binding（深绑定），307
default entry（默认入口），459，461，687
default reduction（默认归约），191
denotational semantics（指称语义），13
dependency arc（依赖弧），587
DeRemer, F.L.（人名），161，166，176，723
derivation tree（推导树），96
derivation（推导），31-32，91-96，98，101，112，133，141-146，160，163，197，703
deterministic finite automaton（确定的有限自动机），79，82
diagnostic compiler（诊断编译器），18，658
Diana（人名），224，250，537，541
dictionary（字典），254
Dion, B. A.（人名），714
directed acyclic graph（有向无环图）
See dag（参见dag）
display（显示表），293-295，305-311，500-502，549-550，624-628
display-swap area（显示表交换区），501
display-swap location（显示表交换场所），500
dominator（必经结点），656，682
dope vector（内情向量），290-291，349-351，401-404
Druseikis, F.C.（人名），706，723
du-chain（du链表，定值－引用链），654，660
dynamic array（动态数组），287，290-291，349-351，403，407，409，501，626
dynamic chain（动态链簇），305-307，456，499
dynamic link（动态链），305-306，309，315，323
dynamic-typed（动态类型的），17

E

Earley's algorithm（Earley算法），206
Early, J.（人名），206
environment（环境），307-313，456，506，534
error production（错误产生式），724
error recovery（错误恢复），65，75，77，691，695-699，713-714
error repair（错误修复），191，691-693，699-712，715-725
error token（错误记号），78
exception handler, predefined（预定义的异常处理程序），459
exception handler（异常处理程序），458-462，482
exceptions（异常），457
export rule（导出规则），269
expressions（表达式），382-387
extended BNF（扩展BNF），30-31，98

F

Farrow, R.（人名），515
finite automata（有限自动机），9，52，55，61，78-79，109，148
First set（First集，首记号集），102，109，113
first-fit（首次适配），300
Fischer, C.N.（人名），18，19，412，598，695，698，704
Fisher,G.A.（人名），706，724
Fleck, A.C.（人名），507
fluid binding（可变绑定），7
Follow set（Follow集，后继记号集），104，113
Fong, A.C.（人名），645
forward references（前向引用），282-284
forward-flow analysis（前向数据流分析），652，654，656
frame pointer（帧指针），502
frame（帧，栈帧），430，510
Fraser, C.W.（人名），574，602

G

Ganapathi, M.（人名），596，598
Ganzinger, H.R.（人名），515
garbage collection（垃圾收集），297-300，302
Garey, M.R.（人名），690
Glanville, R.S.（人名），594
global data flow analysis（全局数据流分析），645
global live variable analysis（全局活跃变量分析），645
global optimization（全局优化），224，621，655，674，677
goal symbol（目标符号），91，141，153，161，207，524
Goos,G.（人名），537
Graham, S.L.（人名），209，267，594，706，723
grammar analysis algorithms（文法分析算法），101
Gray, R.W.（人名），69
Greibach Normal Form（Greibach范式），126

Gries, D.（人名），13

H

Haley, C.B.（人名），706，723
handle（句柄），95，141，144，149，153，160，203，205
handshaking convention（握手协议），299
Harrison, M.A.（人名），209
hash function（哈希函数），71-72，257，259，687
hash tables（哈希表），257-259
Hatcher, P.J.（人名），207，596
heap（堆），288，296-304
Hennessy, J.（人名），617
Henry, R.R.（人名），597
Hopcroft, J.E.（人名），86，96
Hopper, G.M.（人名），2
Horwitz, L.P.（人名），563
Hsu, W.-C.（人名），570

I

immediate error detection property（立即错误发现特性），693
implicit deallocation（隐式存储单元释放），297
implicit declarations（隐式声明），279-280
import rule（导入规则），267，274
importing scope（导入作用域），274
inaccessible（不可达），275，280，282，293，298
index constraint（下标约束），354，357
index register（变址寄存器），551-552，607
indirect jump（间接跳转），442
induction expression（归纳表达式），641-642，644
induction variable（归纳变量），641
inherited attribute（继承属性），219，511-512
inline expansion（内联展开），616，622-624，673
insert-correctable（插入校正），693-694，698，701，703
intermediate representation（中间表示），8，38，217，222-225，246-251，534-546
internal name（内部名），586，631
interpreter（解释器），4，6-7，8，13-14，16，38，48，247，297，470，547，555
interpretive code generation（解释性代码生成），555
interprocedural analysis（过程间分析），673
interprocedural data flow analysis（过程间数据流分析），630
item（项，项目），145，155

J

Jacobsen, V.（人名），69
James, L.R.（人名），713
Jazayeri, M.（人名），513，521
Jensen's device（Jensen程序），507
Johnson, D.S.（人名），690
Johnson, S.C.（人名），178，185，547，580，585
Johnsson, R.K.（人名），572
Jones, C.（人名），13
Joy, W.N.（人名），267，706，723

K

keyword（关键字），70
Kim, J.（人名），570
Knuth, D.E.（人名），144，257，511，625
Kowaltowski, T.（人名），531
Kukuk, R.C.（人名），207，596

L

LALR(1) grammar（LALR(1)文法），199
LALR(1) machine（LALR(1)机器），166-167，192，194
LALR(1) parser（LALR(1)语法分析器），165-177，693，715
Lamb, D.A.（人名），537
L-Attributed attribute flow（L属性的属性流），529
least-cost derivation（最小代价推导），703
LeBlanc, R.J.（人名），18，412，717
left corner（左角），529
left hand side（产生式左边），512，525
left recursion（左递归），123，139，199
leftmost derivation（最左推导），101
left-recursive（左递归的），124
Lesk, M.E.（人名），59
Lewis, P.M.（人名），525
Lex（一个词法分析器生成器），52，59，66-70，77，183
lexical analyzer（词法分析器），50
library routine（库例程），401
live register（活跃寄存器），551
live variable analysis（活跃变量分析），648，658
LL parser（LL语法分析器），127，228，244-245，527
LL(1) grammar（LL(1)文法），34-35，111-116，123-126，686，698，703
LL(1) parse table（LL(1)语法分析表），111，115-116，120，131，201，689
LL(1) parser（LL(1)语法分析器），111，116-137，177，

199-200, 240, 525, 686, 697-711
LL(k) parser (LL(k)语法分析器), 135
local optimization (局部优化), 675
Logothetis, G. (人名), 471
lookahead (超前搜索, 超前搜索符号), 101, 112, 115-117, 123-124, 128, 134-136, 144, 149-151, 154-177, 190-199, 202-206, 690-691, 693-694, 697
loop invariant expressions (循环不变表达式), 637-640
loop optimizations (循环优化), 636-645
LR parser (LR语法分析器), 144-161, 194-198, 229, 237, 527-529, 712-725
LR(1) configuration (LR(1)项目), 166
LR(1) FSM (LR(1)有限状态机), 156
LR(1) machine (LR(1)机器), 156-161, 166-167, 170, 195, 197

M

machine-independent IR (与机器无关的IR), 224
machine-independent optimization (与机器无关的优化), 224, 384
Mauney, J. (人名), 695, 704
maximal solution (最大解集), 666
McCarthy, J. (人名), 16
McKeeman, W.M. (人名), 572
Mickunas, M.D. (人名), 723
Micro (Micro语言), 23-24, 30, 113, 132, 179-180, 188-189
Milton, D.R. (人名), 698, 704
minimal solution (最小解集), 666
Mishra, P. (人名), 471
Modry, J.A. (人名), 723
Mongiovi, R.J. (人名), 717
multi-dimensional array references (多维数组引用), 394-406
multi-pass analysis (多遍分析), 222
multi-pass synthesis (多遍综合), 222, 224

N

name field (名字域), 259
named association (命名关联), 417
nested comments (嵌套注释), 78
Nestor, J.R. (人名), 537
nondeterministic finite automaton (非确定有限自动机), 82
non-importing scope (非导入作用域), 274
non-procedural programming (非过程式程序设计), 52
nonterminal (非终结符), 30, 91, 96
Notkin, D. (人名), 19

O

Ogden, W.F. (人名), 513
one-pass compiler (一遍编译器), 12, 18, 220, 562
open scope (开放式作用域), 261, 415
open subroutine (开放子例程), 622
operator precedence (算符优先), 31, 187-188
operator precedence parsing (算符优先分析), 203-205
optimization of finite automata (有限自动机优化), 84-85
optimization of parse tables (语法分析表优化), 190-194
optimization of subprogram calls (子程序调用优化), 622-636
optimizing compiler (优化编译器), 19, 20, 572, 615-617, 621, 673
overlay (叠加), 288
overlayed vectors (向量叠加), 689, 691
overloading (重载), 26, 223, 279-280, 321, 345, 384, 418-420, 538

P

package declarations (包声明), 366-370
Pager, D. (人名), 193, 195
parameters (参数), 296, 488-497, 503-508, 586-589
  array parameters (数组参数), 491
  formal procedure parameters (过程形参), 307-313, 489
  IN OUT parameters (IN OUT参数), 489, 634
  IN parameters (IN参数), 276, 485, 489, 492, 634, 680
  label parameters (标号参数), 457, 489, 499, 503
  name parameters (名字参数), 16, 489, 505, 507, 510
  OUT parameters (OUT参数), 489, 634
  readonly parameters (只读参数), 489, 492
  reference parameters (引用参数), 489-490, 492
  result parameters (结果参数), 491
  string parameters (串参数), 492
  value parameters (值参), 490, 634
  value-result parameters (值-结果参数), 490
  VAR parameters (VAR参数), 634
parse state (语法分析状态), 715, 718
parser (语法分析器), 10, 33-38, 111-121, 133-137, 141-154, 198-208
parser generators (语法分析器生成器), 129-133, 180-190
parser-controlled semantic stack (语法分析器控制的语义

栈)，236-246
parsing procedure（语法分析过程)，34，112，117-118，695，697
Patterson, D.（人名)，20，628
Paxson, V.（人名)，69
P-code（P-代码)，4，547-548，555，605
P-compiler（P-代码编译器)，268，547
peephole optimization（窥孔优化)，11，221，572-574，602-604
Pennello, T.J.（人名)，176，723
perfect hashing（完美哈希)，71
Perkins, D.R.（人名)，555
phrase（阶段)，95
pointer types（指针类型）
See access types（参见access types）
postfix notation（后缀表示)，249
Pratt, T.W.（人名)，504
precedence relation（优先关系)，205
Predict function（Predict函数，预测函数)，112-116
prediction pointer（预测指针)，206
private part（私有部分)，370
procedures（过程）
See subprograms（参见subprograms）
production（产生式)，30，379，91-98，112-117，145-147，718
propagate link（传播链)，171

Q

quadruple（四元式)，38，247
qualification（限定)，269，277，418，445，595
Quiring, S.B.（人名)，698

R

range map（范围图)，461
reaching definition（到达定义)，652，654，660
read-only（只读)，274，276-277，301-302，341，434
recognition symbol（识别符号)，529
recognizer（识别器)，98
record number（记录号)，268，277
recursive descent parsing（递归下降语法分析)，33-38，117-119，695-697
reduce action（归约动作)，149-150，159，166，190-193
reduce-reduce conflict（归约-归约冲突)，151，160，168，187，194-197，595
reduction successor（归约后继)，175
reference count（引用计数)，298，316
register association list（寄存器关联表)，563，567-568，570
register preference（寄存器优先选择)，387
register spilling（寄存器溢出)，551，575
register targeting（以寄存器为分配、使用目标)，576，652，654
register tracking（寄存器跟踪)，567，570，609
register-save area（寄存器保留区)，500
regular expression（正则表达式)，9，52-57，59-62，66-72，78-80，84
regular grammar（正则文法)，95，109
regular set（正则集)，9，53-54，95，138
relative offset（相对偏移)，408，558
relevant variable（相关变量)，638，650
Reps, T.（人名)，19
reserved register（预留寄存器)，549
reserved word（保留字)，8，24，27-28，51，62，68，70-72
resolved label（已解析标号)，453
resolved（已解析的)，150，168，282-284，414，453
retarget（再目标)，10-11，19，221-222，379，547，555
right hand side（产生式右边)，30，35，101，419，512，525-529，676
right sentential form（右句型)，94，141，149，153，160，163
rightmost derivation（最右推导)，101
rightmost parse（最右语法分析)，101，143
right-recursive（右递归的)，109
right-to-left attribute flow（从右至左的属性流)，532
Ripley, G.D.（人名)，706，723
Roehrich, J.（人名)，715，720
Rosenkrantz, P.M.（人名)，525
Roubine, O.（人名)，267
Rounds, W.C.（人名)，513
Rowland, B.R.（人名)，529
run-time semantics（运行时语义)，12，21
run-time stack（运行时栈)，289-296，299-300，305-311，455-456，504-505，625
Ruzzo, W.L.（人名)，209
r-value（右值)，506

S

ScanGen（一个词法分析器生成器)，52，59-66，69，77，84
scanner（词法分析器)，25-29，50-79

scanner generator (词法分析器生成器), 59, 66
Schmidt, E. (人名), 59
Schwartz, J.T. (人名), 559
scope number (作用域编号), 264
scope stack (作用域栈), 270
semantic analysis, during error correction (错误校正时的语义分析), 725
semantic analysis (语义分析), 119, 225, 250, 534
semantic attribute (语义属性), 218-219, 227-228, 534, 590
semantic error (语义错误), 515
semantic hook (语义钩子), 229
semantic record (语义记录), 40-41, 230-246, 329, 373-374
semantic routines (语义例程), 39, 45, 177-178, 217-228
access_ref, 412-413, 416
access_type, 365-366
append_index, 347, 352-353, 357
append_range, 446, 448
append_val_or_subtype, 446, 448
array_def, 338-340, 351, 354
array_subtype, 347, 356-357
assign, 325, 368
body_present, 367, 369
bool_op, 475
check_data_object, 381, 383
check_package_id, 367, 369
check_proc_id, 485, 488
compare, 475-476
end_package, 367-368
end_proc_body, 485, 488, 494
end_proc_spec, 369, 485, 487
end_record, 335, 338
end_variant_part, 359, 363
end_visible_part, 367-368
enter_for_id, 435-436
enum_id, 343, 345
eval_binary, 381-385, 418, 423, 467
eval_unary, 382-384, 418, 423, 468
exit_cond, 441, 444
exit_jump, 441, 444
field_decl, 335-337, 359-361
field_name, 388-390, 411
finish, 397
finish_case, 446, 450
finish_choice, 446, 449
finish_enum_type, 343, 345
finish_index, 396, 399
finish_loop, 435, 438
finish_name, 416
finish_op, 466-467
finish_while, 432-433, 475, 478
ftf, 476, 478
ftt, 476, 478
gen_else_labal, 429
gen_else_label, 429-430, 477
gen_jump, 429-430, 475, 477
gen_loop_label, 431
gen_out_label, 429-430, 477
gen_proc_jump, 496, 498
if_test, 429-430, 475, 477
incomplete_type, 365-366
index, 396-398, 403-405
init_loop, 435, 437
is_constant, 340-341
loop_back, 431
lower_bound, 338-339
name_plus_list, 416
new_name, 380, 388, 396, 414-415, 422, 494
new_variant, 359, 361-362
next_id, 325-326
no_initialization, 340-341
no_others, 446, 449
not_constant, 340-341
null_name, 441, 444
object_decl, 340, 342-343, 351
param_decl, 493-494
patch_else_jumps, 475, 477
patch_out_jumps, 475, 477
private_type_decl, 369-370
process_attribute, 416
process_id, 323-324, 380
process_index, 390-391
process_literal, 381
process_name, 441, 443
process_not, 466, 468, 475
process_op, 382-383
push_mark, 323-324
range_pair, 347-348
range_subtype, 347-348
return_type, 485
selected_name, 415
set_in, 435-437, 493

set_in_out, 493-494
set_out, 493-494
set_reverse, 435-437
simple_proc_stmt, 487-488
_startcase, 446-477
start_expr_list, 416
start_id_list, 325-326
start_if, 429, 475
start_index, 396, 398
start_index_list, 348, 352-353, 356-357
start_op, 466-467
start_others, 446, 449
start_package, 367
start_package_body, 367-368
start_proc, 485-486
start_proc_body, 485, 487
start_proc_stmt, 496-497
start_record, 335-336, 359
start_while, 432, 475, 477
subtype_decl, 347-348
tag_field, 359, 361-362
type_decl, 330-332, 370
type_reference, 330-331
unconstrained_index, 352
upper_bound, 338-339
var_decl, 326-328, 332, 486
while_test, 432-433, 475, 477
semantic stack (语义栈), 230, 235, 237, 244, 252, 336, 414
semantic attribute (语义属性), 536
sentential form (句型), 92-94, 109
Sethi, R. (人名), 14, 575, 641
Sethi-Ullman numbering (Sethi-Ullman编号方法), 575
shallow binding (浅绑定), 307
Shannon, A. (人名), 197
shift action (移进动作), 150, 153, 190
shift-reduce conflict (移进-归约冲突), 186, 195
shift-reduce parser (移进-归约语法分析器), 177-178, 194
short-circuit evaluation (短路计算), 463-478, 531-534
simple assignment form (简单赋值形式), 525
Simple LR(1) (简单LR(1)), 161
simple precedence (简单优先级), 205
single-pass analysis (单遍分析), 223
Sites, R.L. (人名), 555
SLR(1) parser (SLR(1)语法分析器), 693
Soisalon-Soininen, E. (人名), 193
Stallman, R.M. (人名), 12
start symbol (开始符号), 516
statements (语句)
assignment statement (赋值语句), 45, 248-249, 392-393, 578-585
case statement (case语句), 444-450
exit statement (exit语句), 440-444, 501
for loop (for循环), 279-280, 433-440, 618, 621, 636-645, 670-673
goto statement (goto语句), 452-457
if statement (if语句), 72, 127-128, 226, 229, 427-430, 531-534
loop statement (loop语句), 431
while loop (while loop), 432
with statement (with语句), 277
static arrays (静态数组), 338-340, 389-392
static chain (静态链簇), 305-310, 455-457, 502
static link (静态链), 305, 308, 504
Stearns, R.E. (人名), 525
Steel, T.B., Jr. (人名), 555
storage allocation (存储分配)
heap allocation (堆分配), 296-301
stack allocation (栈分配), 289-296
static allocation (静态分配), 288
storage temporary (存储器临时变量), 439, 550, 585, 658
strength reduction (强度削弱), 641-645
string space (串空间), 259-261
string operations (串操作), 393
strong compatibility (强兼容性), 197
Strong LL(1) parser (强LL(1)语法分析器), 137, 694
subprogram declarations (子程序声明), 485-488, 494-495
subprogram invocation (子程序调用), 488, 495-503
subtype (子类型), 345-349, 352-358
successor item (后继项), 720
successor state (后继状态), 713
successor (后继), 85, 147, 153, 160, 195
Sussman, G. (人名), 20
symbol table (符号表), 17, 42, 74, 254-284, 327-328, 335, 379-380, 389, 453
syntax error (语法错误), 77, 153, 698, 715, 726
syntax errors (语法错误), 48
syntax tree (语法树), 217, 219, 226, 511, 516, 520, 537
syntax (语法), 51, 70, 147, 188
synthesis (综合), 216, 220, 222-223, 225, 510

synthetic attribute（综合属性），511-512
Szymanski, T.G.（人名），558

T

Tanenbaum, A.S.（人名），4，572
target computer（目标计算机），19，52，301
Tarjan, R.E.（人名），690
Teitelbaum, T.（人名），19
temporary allocation（临时变量分配），550
temporary（临时变量），39，43，248-251，386-388，548-551，559
terminal（终结符），30，91-95，129-132，181-184
threaded code（线索化代码），548
three-address code（三地址代码），247
thunk，16，506，508
Tjiang, S.（人名），596
token merge（记号合并），724
top-down parser（自顶向下的语法分析器），98-99，117-121
topological sort（拓扑排序），625
trailing part（尾部），529
transducer（转换器），58
transfer vector（转移向量），303，458-462，481
transition diagrams（转换图），54
transition table（转换表），55-57，61，63，72
tree-structured intermediate representation（树结构中间表示），511
tree-walk（树遍历），516
triple（三元组），171，247-248，253，557
type declarations（类型声明），331-333
type descriptor（类型描述符），322，330-331，402-411，491
type-0 grammar（0型文法），95
typeless（无类型的），17

U

U-code（U-代码），555
ud-chain（ud-链表，引用-定值链表），654
Ullman, J.D.（人名），86，96，185，204，575，580，585，641，645
unallocated register（未分配的寄存器），551
UNCOL（Universal Compiler-oriented Language，面向编译器的通用语言），555
uninitialized variable analysis（未初始化变量分析），658，661
unique invertibility（惟一可逆性），203
unreachable nonterminal（不可达非终结符），110
unreachable（不可达），96，674
unresolved label（未解析标号），453
useless nonterminal（无用非终结符），96，109

V

validation window（有效窗口），712
validation（有效性），706-710，715，717，723
Valliant, L.（人名），207
value numbering（值编号方法），559，608
variable declarations（变量声明），326-329，340-343
variable references（变量引用），380
variant record（变体记录），359，363，411
very busy analysis（非常忙分析），651
very busy expression（非常忙表达式），651，656
viable prefix（活前缀），149，152-153，160，197
virtual machine（虚拟机），3-5，8，20，24，38，225，247，555
virtual origin（虚拟起始地址），390，405
virtual stack machine（虚拟栈机器），4，547
visibility rule（可见性规则），261
vocabulary（词汇表），53，91-92，101-104
volatile registers（易变寄存器），549

W

Walter, K.G.（人名），521
Wand, M.（人名），14
Watt, D.A.（人名），530
weak compatibility（弱兼容性），195
Weber, H.（人名），203
Wetherell, C.（人名），197
Wilcox, T.R.（人名），18，547
Wirth, N.（人名），203，695
with clause（with子句），278
Wulf, W.A.（人名），19，537

Y

Yacc（Yet Another Compiler Compiler，一个语法分析器生成器），183-186，713-714
Yao, A.C.（人名），690

Z

Zellweger, P.（人名），617