

信息科学与技术丛书

裘巍 编著

编译器 设计之路

- ▶ 完整介绍一个实际编译器Neo Pascal的设计与实现
- ▶ 详细讲述LL(1)分析器、符号表系统、优化等核心话题
- ▶ 阐述系统软件的设计观点
- ▶ 探讨现代编译技术热点
- ▶ 提供编译器全部源代码



 机械工业出版社
CHINA MACHINE PRESS

信息科学与技术丛书

编译器设计之路

裘 巍 编著



机械工业出版社

本书系统地介绍了一个实际的 Pascal 编译器 Neo Pascal 的设计与实现。结合 Neo Pascal 的源代码,详细讲述了 LL(1)语法分析器、符号表系统、中间表示、类型系统、优化技术、运行时刻的存储管理、代码生成器等编译器设计的核心话题。各章都附有少量以实践应用为主的练习题,既可作为阅读思考题,也可作为课程设计选题。

与国内其他介绍编译技术的图书相比,本书更关注的是编译器的实现细节,而不仅仅局限于理论阐述。本书可供从事编译器设计相关工作的工程人员阅读,也可作为高等院校计算机专业的编译原理课程参考书。

读者可在 <http://neopascal.sourceforge.net> 获得 Neo Pascal 的源代码及相关文档。

图书在版编目(CIP)数据

编译器设计之路 / 裘巍编著. —北京:机械工业出版社, 2010.10

(信息科学与技术丛书)

ISBN 978-7-111-32164-4

I. ①编… II. ①裘… III. ①编译程序—程序设计 IV. ①TP314

中国版本图书馆 CIP 数据核字(2010)第 196394 号

机械工业出版社(北京市百万庄大街 22 号 邮政编码 100037)

策划编辑:车 忱

责任编辑:车 忱

责任印制:乔 宇

三河市宏达印刷有限公司印刷

2011 年 1 月第 1 版·第 1 次印刷

184mm×260mm·28.75 印张·713 千字

0001—3000 册

标准书号:978-7-111-32164-4

定价:59.00 元

凡购本书,如有缺页、倒页、脱页,由本社发行部调换

电话服务

网络服务

社服务中心:(010) 88361066

门户网:<http://www.cmpbook.com>

销售一部:(010) 68326294

教材网:<http://www.cmpedu.com>

销售二部:(010) 88379649

读者服务部:(010) 68993821

封面无防伪标均为盗版

出版说明

随着信息科学与技术的迅速发展，人类每时每刻都会面对层出不穷的新技术和新概念。毫无疑问，在节奏越来越快的工作和生活中，人们需要通过阅读和学习大量信息丰富、具备实践指导意义的图书来获取新知识和新技能，从而不断提高自身素质，紧跟信息化时代发展的步伐。

众所周知，在计算机硬件方面，高性价比的解决方案和新型技术的应用一直备受青睐；在软件技术方面，随着计算机软件的规模和复杂性与日俱增，软件技术不断地受到挑战，人们一直在为寻求更先进的软件技术而奋斗不止。目前，计算机在社会生活中日益普及，随着Internet延伸到人类世界的方方面面，掌握计算机网络技术和理论已成为大众的文化需求。由于信息科学与技术 在电工、电子、通信、工业控制、智能建筑、工业产品设计与制造等专业领域中已经得到充分、广泛的应用，所以这些专业领域中的研究人员和工程技术人员越来越迫切需要汲取自身领域信息化所带来的新理念和新方法。

针对人们了解和掌握新知识、新技能的热切期待，以及由此促成的人们对语言简洁、内容充实、融合实践经验的图书迫切需要的现状，机械工业出版社适时推出了“信息科学与技术丛书”。这套丛书涉及计算机软件、硬件、网络和工程应用等内容，注重理论与实践的结合，内容实用、层次分明、语言流畅，是信息科学与技术领域专业人员不可或缺的参考书。

目前，信息科学与技术的发展可谓一日千里，机械工业出版社欢迎从事信息技术方面工作的科研人员、工程技术人员积极参与我们的工作，为推进我国的信息化建设作出贡献。

机械工业出版社

前 言

与其他自然科学相比，计算机科学的发展历史并不久远，是较新的学科体系，尚有许多未知的领域有待探索。因此，本专业学生或工程技术人员仅仅满足于学习或应用几门程序设计语言是远远不够的。一些看似抽象的课程才是提高专业人员“内功”修为的秘技，例如数据结构、操作系统、编译原理、计算机系统结构、计算机网络等。不过，经典课程的学习并不是一蹴而就的，如何学习与理解课程的精髓是值得关注的。本书将从编译器设计的角度，为读者揭示编译原理的精髓。

学习编译技术的意义

有人认为，编译技术似乎已经相当成熟了，继续深入研究是没有任何意义的。实际上，任何科学技术都是发展变化的。表面上看，编译器设计的高层问题似乎已经形成了完美的体系，但当我们深入其内核就会发现事实并非如此。现代编译器设计面临的挑战是来自目标计算机系统结构、新颖程序设计语言及本身的计算资源等多方面的。其中，任何一方面的因素都足以颠覆某些传统理论与算法。例如，在现代编译器设计中，计算资源的增加允许设计者采用更耗费时间、空间的算法，而不必过分关注其代价。对于优化算法设计者而言，这是令人兴奋的。为了追求目标代码的更优，即使设计一些时空复杂度稍高的算法也是可以接受的。

当然，从更高的层次上讲，学习编译器设计的目的还不仅仅局限于其本身的理论与技术。作为一个系统软件的设计学科，其解决问题的思路与方法更是值得读者细细品味的。这可能是一个漫长而艰辛的历程，不过，这才是经典学科的魅力所在。以品味经典为目的来学习与研究操作系统、数据库技术、计算机网络、编译技术等学科是笔者多年来的努力方向。

从 20 世纪 50 年代中期以来，编译器设计就一直是计算机科学界的一个重要研究领域。Fortran 语言之父 John Backus 认为，除非编译器生成的代码与手工编写的机器代码的性能非常接近，否则程序员就不会放弃汇编语言程序设计的思想与方法，这就是编译技术研究的源动力。从学术研究的角度讲，众所周知，图灵奖被誉为“计算机界的诺贝尔奖”，自 1966 年设立至今，54 位获奖者中就有 16 位是由于程序设计语言及编译技术的研究成果而获此殊荣的。编译技术在计算机科学领域的地位由此可见一斑。

本书的特点

国内讲述编译原理的书籍并不少见，但是基于一个实际编译器系统讨论相关实现的资料却相当稀缺。同样，国内高校的编译原理课程也存在着一个普遍的现象：讲述的内容与实际编译器设计之间存在着鸿沟。对许多专业学生而言，编译原理可能更像是一门充满数学符号与定义的计算机课程，因此很难得到学生的认可。

本书摒弃了传统教材只讲理论的不足，以笔者开发的编译器 Neo Pascal 为例，从词法分析、语法分析、语义分析、IR 生成、代码优化、目标代码生成等角度全面、系统地阐述了编译器设计与实现中的许多经典话题，包括 LL(1) 语法分析器的实现、符号表的设计与实现、类型系统的实现、IR 的设计、迭代数据流分析、IR 优化、运行时刻管理、基于模式匹配的代码生成器的实现等。同时，笔者也引入了一些现代编译技术中的观点，对传统的算法

进行了一定的改进，以便达到相对更优的结果。与传统编译教材的最主要区别在于，本书并不是一本以阐述编译技术的相关理论与算法框架为主的原理性教材，而是一本依托实际编译器项目讨论其实现的书籍。因此，本书将不涉及与 Neo Pascal 实现无关的算法或话题。

在笔者看来，源代码是一种实现，也是设计的载体。因此，本书不是一本简单的源代码注释文本。其中，更多体现的是以提出问题并设计解决方案为核心的思想。笔者试图传达给读者的并不局限于编译器源代码的实现，而是其内核的设计原理。

笔者坚信，只有深刻理解算法分析与设计的过程，并实际参与编译器的设计，才可能体会编译技术的真谛。通过启发式的讲解，引领读者进入编译技术的殿堂正是撰写本书的目的。无论是对于在校的相关专业学生还是专业技术人员，本书都是非常适合自学使用的。对于有志于涉足编译技术领域的专业人员，本书的许多观点与方法更是值得思考。

本书涉及的 Neo Pascal 是一个开源的编译器系统，读者可以登录 <http://neopascal.sourceforge.net>，获得最新的项目源代码及相关文档。

本书的读者

本书预期的读者是至少掌握了 C++ 语言及数据结构的计算机专业人员或在校学生。对于有汇编语言程序设计、计算机系统结构、操作系统、编译原理等课程基础的读者，学习本书将事半功倍。当然，在涉及相关知识时，本书将作简单介绍。

内容概览

第 1 章 概述

本章将介绍一些关于程序设计语言、编译器的概念性话题，例如，解释与编译、编译器的架构等，这是讨论后续主题的基础。另外，本章也将简单介绍本书待实现的高级语言 Pascal。

第 2 章 词法分析

本章将介绍词法分析器的实现。与传统教材不同，本章只涉及词法分析器的设计与实现，并不讨论有限自动机的相关理论。

第 3 章 语法分析

本章将讨论形式语言描述方法及语法分析器的实现。形式语言是一种语言语法的描述形式，是语法分析器实现的基础。在语法分析器方面，本章将深入讨论 LL(1) 语法分析器的原理与实现。这是一种非常经典的语法分析方法，广泛应用于许多经典编译器中。

第 4 章 符号表系统

符号表是编译器的中心数据库。不过，关于符号表的话题，传统教材却涉及甚少。虽然符号表的功能与普通数据表类似，但两者结构特点相差甚远。

第 5 章 中间表示

中间表示是一种由设计者定义与使用的内部语言。事实上，经典的中间表示形式非常多。本章将介绍一种源自 lcc 的中间表示形式。同时，笔者将结合个人的体会，介绍一些关于中间表示的设计经验及其评价机制。另外，各种经典语句的翻译也是本章讨论的重点。

第 6 章 表达式语义

本章将涉及类型系统、表达式翻译等复杂的话题。其中，类型系统将讨论类型描述、类型兼容、类型转换、类型推断等，而表达式翻译更多关注的是数组、指针、结构、函数调用等语言机制的处理。



第 7 章 优化技术

优化技术是现代编译器设计关注的重点。鉴于优化的重要性，本书不惜耗费大量篇幅详细阐述各种基本优化算法的原理与实现。与传统教材相比，本书不局限于原理的介绍，更配以详细的源代码实现，便于读者理解，这是本书的特色之一。当然，本书并不是一本以介绍优化算法为主的高级教材，所以，并不涉及那些理论抽象、实现复杂的算法。

第 8 章 运行时刻的存储管理

本章将重点介绍栈、堆式存储分配机制的原理及其实现技术。这是一个实用性非常强的话题，故笔者将结合一些常见的语言现象进行深入剖析，包括动态数组、字符串、可变参数、调用约定等。

第 9 章 目标代码生成

本章将依托 i386 目标机实现一个完整的代码生成器。笔者采用了一种基于模式匹配实现的代码生成技术，这是一项非常高效且灵活的生成技术。同时，寄存器的分配也是本章讨论的重点。与传统教材基于模型机讨论代码生成相比，基于目标机实现代码生成器可能复杂得多，包括寄存器的分配、指令的选择等都需要考虑许多硬件结构的限制。

第 10 章 GCC 内核与现代编译技术概述

了解 GCC 内核是深入学习先进编译技术的基础，本章将以有限的篇幅介绍几个 GCC 最核心的话题，包括 GIMPLE、SSA、RTL 等。可以毫不夸张地说，它们正是 GCC 的魅力所在。其次，笔者还将介绍两个现代编译技术的研究热点，即动态编译技术与并行编译技术。并且还将提供一些资源与材料，供读者参考使用。

致谢

首先，要感谢上海交通大学张冬茱老师给予的极大鼓励与帮助，是她引领我走上了编译器设计之路。其次，要感谢我的同事潘松，在著书过程中，他给予了我最大程度的鼓励与支持。

联系方式

由于笔者水平有限，书中难免存在一些错误，殷切希望广大读者不吝批评指正。笔者的电子信箱是 qiuwei-qiuwei@163.com，QQ 号是 331341714。

目 录

出版说明

前言

第 1 章 概述	1	2.6 大师风采——Dennis M. Ritchie	45
1.1 编译技术概述	1	第 3 章 语法分析	47
1.1.1 程序设计语言基础	1	3.1 程序设计语言的语法描述	47
1.1.2 程序设计语言的翻译机制	4	3.1.1 上下文无关文法	47
1.1.3 编译器的基本结构	5	3.1.2 推导	52
1.2 Pascal 语言基础	8	3.1.3 语法树	54
1.2.1 Pascal 语言简介	8	3.1.4 归约简介	57
1.2.2 Pascal 程序基本组成	9	3.2 语法分析概述	58
1.2.3 Pascal 的声明部分	10	3.2.1 语法分析的任务	58
1.2.4 Pascal 的类型	12	3.2.2 自上而下的语法分析法	59
1.2.5 Pascal 的运算符	15	3.2.3 构造语法分析器	64
1.2.6 Pascal 的语句	17	3.3 语法分析器的实现	71
1.3 开发环境与 Delphi 基础	18	3.3.1 文法定义	71
1.3.1 开发环境与文件列表	18	3.3.2 语法分析表	76
1.3.2 Delphi 基础	19	3.3.3 源代码实现	86
1.4 深入学习	24	3.4 深入学习	90
1.5 实践与思考	25	3.5 实践与思考	91
1.6 大师风采——Niklaus Wirth	25	3.6 大师风采——Edsger Wybe Dijkstra	92
第 2 章 词法分析	26	第 4 章 符号表系统	93
2.1 词法分析概述	26	4.1 语义分析概述	93
2.1.1 词法分析的任务	26	4.1.1 程序设计语言的语义	93
2.1.2 单词的分类	28	4.1.2 语义分析与 IR 生成的任务	94
2.2 词法分析器的设计	28	4.1.3 语法制导翻译	95
2.2.1 识别单词	28	4.2 符号表设计	98
2.2.2 转换图	29	4.2.1 符号表概述	98
2.2.3 构造词法分析器	31	4.2.2 符号表的逻辑结构	99
2.3 词法分析器的实现	35	4.2.3 符号表的实例分析	109
2.3.1 词法定义	35	4.3 声明部分的实现	111
2.3.2 构造转换图与转换表	36	4.3.1 相关数据结构	111
2.3.3 相关数据结构	38	4.3.2 主程序首部声明	113
2.3.4 源代码实现	40	4.3.3 包含文件声明部分	114
2.4 深入学习	44	4.3.4 标号声明部分	118
2.5 实践与思考	45		



4.3.5 常量声明部分	119	5.11 大师风采——Kenneth E. Iverson	209
4.3.6 类型声明部分	120	第6章 表达式语义	210
4.3.7 变量声明部分	149	6.1 表达式概述	210
4.3.8 过程、函数声明部分	152	6.2 类型系统基础	211
4.4 深入学习	154	6.2.1 类型基础	211
4.5 实践与思考	155	6.2.2 类型系统	212
4.6 大师风采——John Backus	155	6.2.3 类型转换	217
第5章 中间表示	156	6.3 类型系统的实现	218
5.1 IR 概述	156	6.3.1 类型系统的设计	218
5.1.1 IR 的作用	156	6.3.2 IR 的操作数	221
5.1.2 IR 设计及其级别	157	6.3.3 类型相容的实现	222
5.1.3 设计 IR 的重要意义	159	6.3.4 类型推断的实现	223
5.2 IR 生成	160	6.4 表达式翻译	226
5.2.1 三地址代码概述	160	6.4.1 表达式翻译基础	226
5.2.2 Neo Pascal三地址代码的实现	164	6.4.2 深入表达式翻译	229
5.2.3 翻译机制概述	168	6.4.3 表达式翻译的实现	230
5.3 语句翻译概述	170	6.5 操作数翻译	247
5.3.1 语句翻译基础	170	6.5.1 操作数的地址与形态	247
5.3.2 翻译辅助函数及其实现	173	6.5.2 操作数翻译基础	248
5.4 if 语句	176	6.5.3 简单变量操作数的翻译	252
5.4.1 if 语句的翻译	176	6.5.4 记录字段操作数的翻译	262
5.4.2 源代码实现	177	6.5.5 数组翻译基础	265
5.5 while/repeat 语句	181	6.5.6 数组元素操作数的翻译	270
5.5.1 while 语句的翻译	181	6.5.7 指针运算的翻译	280
5.5.2 源代码实现	181	6.6 深入学习	286
5.5.3 repeat 语句的翻译	184	6.7 实践与思考	286
5.6 for 语句	184	6.8 大师风采——Alan Kay	287
5.6.1 for 语句的翻译	184	第7章 优化技术	288
5.6.2 源代码实现	186	7.1 优化概述	288
5.7 case 语句	192	7.1.1 什么是优化	288
5.7.1 case 语句的翻译	192	7.1.2 优化级别	289
5.7.2 源代码实现	193	7.2 控制流分析	290
5.8 其他语句	199	7.2.1 流图与基本块	290
5.8.1 break、continue 语句的翻译	199	7.2.2 流图的数据结构	292
5.8.2 goto 语句的翻译	201	7.2.3 流图的构造	293
5.8.3 asm 语句的翻译	204	7.2.4 优化的分类	297
5.9 深入学习	208	7.3 数据流分析	298
5.10 实践与思考	208	7.3.1 数据流的相关概念	298

7.3.2 数据流分析的策略	298	8.2.2 i386 栈式存储分配	360
7.3.3 活跃变量分析	299	8.2.3 深入理解栈式存储分配	365
7.3.4 ud 链与 du 链	301	8.3 存储分配的实现	368
7.3.5 更多数据流问题	302	8.4 存储优化	372
7.4 数据流分析的实现	303	8.4.1 存储优化基础	372
7.4.1 定值点与引用点分析的基础	303	8.4.2 存储优化的实现	374
7.4.2 定值点、引用点分析的相关 数据结构	305	8.5 深入学习	381
7.4.3 定值点、引用点分析的实现	307	8.6 实践与思考	382
7.4.4 活跃变量分析的实现	312	8.7 大师风采——Bjarne Stroustrup	382
7.4.5 ud 链、du 链分析的实现	314	第 9 章 目标代码生成	383
7.5 常量传播与常量折叠	321	9.1 目标代码生成概述	383
7.5.1 常量传播基础	321	9.1.1 目标代码生成基础	383
7.5.2 常量传播的实现	324	9.1.2 指令选择	384
7.6 复写传播	328	9.1.3 寄存器分配	385
7.6.1 复写传播的基础	328	9.2 目标机简介	386
7.6.2 复写传播的实现	330	9.2.1 目标机结构	386
7.7 代数简化	333	9.2.2 浮点处理单元	387
7.7.1 代数简化基础	333	9.2.3 操作数寻址方式	391
7.7.2 代数简化的实现	334	9.2.4 ptr 操作符	392
7.8 跳转优化	339	9.2.5 一个完整的汇编程序	393
7.8.1 跳转优化基础	339	9.3 构造代码生成器	393
7.8.2 条件跳转优化的实现	341	9.3.1 自动代码生成器基础	393
7.8.3 连续跳转优化的实现	343	9.3.2 指令模板	394
7.9 冗余代码删除	345	9.3.3 寄存器描述	397
7.9.1 冗余代码删除基础	345	9.3.4 寄存器分配	398
7.9.2 死代码删除的实现	346	9.3.5 代码生成器的基本结构	402
7.9.3 不可到达代码删除的实现	348	9.4 深入学习	413
7.10 深入学习	350	9.5 实践与思考	413
7.11 实践与思考	350	9.6 大师风采——Peter Naur	413
7.12 大师风采——Richard Stallman	351	第 10 章 GCC 内核与现代编译 技术概述	414
第 8 章 运行时刻的存储管理	352	10.1 编译技术的现状及发展	414
8.1 存储管理概述	352	10.2 GCC 内核分析	415
8.1.1 存储区域	352	10.2.1 GCC 的基本结构	415
8.1.2 存储布局	354	10.2.2 GENERIC	416
8.1.3 存储分配基础	356	10.2.3 GIMPLE	416
8.2 栈式存储分配	357	10.2.4 SSA	426
8.2.1 栈式存储分配基础	357	10.2.5 RTL 概述	428



编译器设计之路

10.2.6 RTX	430	10.4.1 并行编译技术基础	441
10.3 动态编译技术简介	436	10.4.2 并行计算机及其编译系统	443
10.3.1 动态编译技术基础	436	10.5 深入学习	446
10.3.2 运行时特定化	437	10.6 大师风采——Alan Perlis	447
10.3.3 动态二进制翻译	439	参考文献	448
10.4 并行编译技术简介	441		

第 1 章 概 述

Reliable and transparent programs are usually not in the interest of the designer.

——Niklaus Wirth

1.1 编译技术概述

1.1.1 程序设计语言基础

Intel 公司的 David Kuck 院士曾经将编译器誉为“计算机科学与技术的皇后”，它是应用与系统之间的一座桥梁。在国内，由于计算机基础科学相对薄弱，人们通常更多投身于应用领域的研究，编译技术并不太受到人们的关注。因此，基于这方面的研究成果也相对较少。除了早期一些大型机的 Fortran、Algol 编译器之外，并没有真正的产品级编译器。不过，这并不意味着研究编译技术是毫无价值可言的。事实上，作为计算机科学的组成部分之一，编译技术有着极其崇高的地位，其辉煌的历史可能是其他许多学科无法比拟的。众所周知，图灵奖被誉为“计算机界的诺贝尔奖”，自 1966 年设立至今，54 位获奖者中就有 16 位是由于程序设计语言或编译技术的研究成果而获此殊荣的，见表 1-1。虽然有些大师的身影已经渐渐远去，但是他们的研究成果却为人津津乐道。

表 1-1 因程序设计语言或编译技术而获图灵奖的科学家

年份	获 奖 者	获 奖 原 因
1966	Alan J. Perlis	先进编程技术和编译架构方面的贡献
1971	John McCarthy	Lisp 语言、程序语义、程序理论、人工智能方面的贡献
1972	E. W. Dijkstra	对开发 Algol 做出了原理性贡献
1977	John Backus	在高级语言方面所做出的具有广泛和深远意义的贡献，特别是在 Fortran 语言方面
1979	Kenneth E. Iverson	在编程语言的理论和实践方面（特别是 APL）所进行的开创性的工作
1980	C. A. R. Hoare	在编程语言的定义和设计方面的贡献，如 case 语句、公理语义学等
1983	Ken Thompson, Dennis M. Ritchie	在通用操作系统理论研究，特别是 UNIX 操作系统的实现上的贡献。开发实现了 C 语言
1984	Niklaus Wirth	开发了 Euler、Algol-W、Modula 和 Pascal 一系列崭新的计算语言
1991	Robin Milner	在可计算函数逻辑（LCF）、ML 和并行理论（CCS）这三个方面突出的贡献
2001	Ole-Johan Dahl、Kristen Nygaard	面向对象编程始发于他们基础性的构想，这些构想集中体现在他们所设计的编程语言 Simula I 和 Simula 67 中
2003	Alan Kay	在面向对象语言方面的原创性思想，领导了 Smalltalk 的开发团队，以及对 PC 的基础性贡献
2005	Peter Naur	在设计 Algol 60 程序设计语言上的贡献。Algol 60 语言定义清晰，是许多现代程序设计语言的原型
2006	Frances Allen	对于优化编译器技术的理论和实践做出的开创性贡献，这些技术为现代优化编译器和自动并行执行打下了基础
2008	Barbara Liskov	在计算机程序语言设计方面的开创性工作。开发了面向对象编程语言 CLU

注：见《ACM 图灵奖：计算机发展史的缩影（1966-2006）》（第三版），吴鹤龄，崔林，高等教育出版社。



从 20 世纪 50 年代到 80 年代末，一些程序设计语言的兴起，为编译技术的发展提供了新的契机。Ada、Fortran、Algol、Pascal、C、C++ 等高级语言编译器如雨后春笋一般诞生。作为编译领域的经典之作，“龙书”也是撰写于这一时期的，它的出现将原先模糊的理论体系与实现技巧完全梳理清晰了。正在人们隐约感到编译技术（尤其是前端技术）已相对成熟之际，一个崭新的时代正悄然来临。自 20 世纪 90 年代至今，计算机硬件体系的飞速发展，把编译技术的研究推向了新的高峰。

编译器是将一种程序设计语言编写的源程序等价地转换为另一种程序设计语言编写的源程序的系统软件。习惯上，将前者称为源语言，而将后者称为目标语言。读者应该对语言并不陌生，汉语、英语、C、Java 等都是语言。那么，它们的联系与区别是什么呢？

大千世界的语言一般可以分为两类，即自然语言和人工语言。自然语言就是日常生活中使用的语言，如汉语、英语。自然语言通常是在交流过程中逐渐完善的，并不是刻意定义的。人工语言则是为了特定目的、用途，而人为创造出来的语言。例如，在计算机程序设计中使用语言、工程技术中的符号、图形语言等。实际上，读者可以将人工语言理解成为解决某一特殊问题而定义的一种语言，例如 C、Java 等就是为了解决计算机程序设计而定义的人工语言。不过，自然语言与人工语言之间并非如人们所想象的那样存在着严格的界限。虽然自然语言没有太多的人为主观因素，但是它确实也是由人类创造的。因此，读者只需了解这两个概念即可，不必过多深究。

早在 20 世纪 50 年代，计算机科学家就开始对语言处理进行了深入的研究，其中包含最重要的两个领域：一是自然语言理解与处理，二是程序设计语言及其编译技术。前者主要研究计算机如何才能正确识别与处理自然语言及其语义。而后者主要研究如何设计一种人工语言，使之有效地完成人与计算机之间的无障碍交流，计算机之所以能普及到千家万户与程序设计语言的发展是分不开的。不过，两者的研究至今仍然尚待完善，例如，至今人类还无法使用计算机自动完成英语句子到汉语句子准确无误的翻译。这主要是由于无法让计算机准确理解输入英语句子的语义，同时，也无法让计算机将预定语义使用汉语表达输出。这就需要在自然语言理解与处理领域的继续探索。

自然语言理解与处理不是本书的研究对象，笔者不多作讨论。下面，来谈谈程序设计语言及其编译技术的话题。这方面的研究主要包括：程序设计语言设计与定义、高级语言编译器设计、编译优化技术、并行编译技术、嵌入式编译技术、动态编译技术等。读者阅读完本书之后，可以根据个人兴趣选择其中某些领域作深入学习与研究。

程序设计语言是一种人工语言，而人工语言的一个重要特点就是人们对语言作一些严格的规定，从而不必过多关注语言表达形式的不确定性。例如，在 C 语言中，表达式 $a=10$ 只能用于描述“变量 a 是否等于 10”这一种语义。然而，在自然语言中，一个句子存在两种以上语义的情况并不罕见。例如，“我看见他太激动了。”这个句子表达的含义到底是什么？

可以理解为： 我看见他后，我太激动了。

也可理解为： 我看见他时，他非常激动。

这里“看见”的宾语选择使句子存在二义性。然而，同样的问题在程序设计语言中却很罕见。例如，在 C 语言中，对于 $a+++b$ 这样的表达式，读者可能会疑惑运算结果到底应该

是 $(a++) + b$ 还是 $a + (++b)$ 。但根据 C 语言的约定, 运算结果一定是 $(a++) + b$ 。由此可见, 在设计一门程序语言及其编译器时, 避免二义性是设计者考虑的首要因素。至于为什么一定是这样的结果呢? 当学习完第 2 章, 读者一定会了解 C 语言这一约定的现实意义与必要性。当然, 这并不意味着程序设计语言中是绝对不存在二义性的。以 C 语言为例, $(++a) + (++a)$ 的取值就将因编译器而异。

迄今为止, 程序设计语言仍是人类与计算机交流的主要途径, 它的应用领域也仅限于操纵与控制计算机。从第一台计算机诞生之日起, 人类就始终在探索一种有效的方式与计算机进行对话交流, 使之能为人类服务。虽然时隔数十年, 计算机能识别与处理的语言仍然是二进制形式的机器语言描述的源程序。当然, 不可否认二进制机器语言的优点非常多, 但机器语言的易用性差也是不可回避的。即使是计算机专家想直接使用机器语言与计算机进行交流也是非常困难的。在 20 世纪 50 年代, 计算机科学家们就已经意识到必须解决这一棘手的问题, 否则计算机将无法得到普及。经过多年努力, 汇编语言、C、Pascal 等程序设计语言终于横空出世。根据语言的形式与特点, 习惯上, 将机器语言与汇编语言(一种比较接近机器语言的设计语言)称为低级语言, 将其余的 C、Pascal 之类的语言称为高级语言。读者必须注意, 低级语言与高级语言之分并不是说明语言本身的优劣, 仅仅是说明语言的形式与机器语言的相似程度。所谓低级语言指的是与机器语言比较类似的语言, 而高级语言指的是与机器语言差别较大而与自然语言比较类似的语言。由于这些非机器语言的诞生, 也就出现了将非机器语言等价翻译成机器语言的需求。显然, 将非机器语言翻译成机器语言的工作不能由手工完成, 否则, 非机器语言的产生就没有任何意义了。因此, 人们试图借助于一个程序工具自动完成翻译工作。根据语言不同, 翻译工具的复杂程度也不尽相同。比如, 汇编语言比较接近机器语言, 所以其翻译工具较易实现。而高级语言与机器语言差别较大, 所以其翻译工具的实现也较为复杂。习惯上, 将前者称为汇编器, 而将后者称为编译器。当然, 有些书上对于汇编器与编译器并没有严格区分, 都将其称为编译器, 反正这只是一个名词而已, 读者不必深究。编译器的源语言是一种较为高级的程序设计语言, 而目标语言可以是汇编语言、机器语言或者另一种高级语言。笔者必须澄清一点, 人们普遍认为编译器的目标语言就是低级语言, 这个观点的确没有错, 但并不完整。实际上, 有些编译器的目标语言可能是某一种已存在的高级语言, 例如, 第一个由 Bjarne Stroustrup 开发的 C++ 编译器的目标语言就是当时的 C 语言。假设未来的计算机 CPU 能直接处理 C 或者 Java 语言, 那么, 低级语言、高级语言及编译器的概念也将被重新诠释。

今天, 虽然计算机的 CPU 可以达到天文级的处理速度, 但是识别的指令数量却不会超过 1000 条(PC 的指令规模一般只有 300~500 条左右)。无论多么深奥、华丽的高级语言源代码最终必将被这近千条机器指令等价替换。相对于自然语言而言, 程序设计语言的表达能力要小得多。不过, 程序设计语言也有其自身的特性, 比如, 程序设计语言必须简单易学。不可能要求程序员用十年磨一剑的精神像学习英语一样去学习 C 语言。过于复杂的语言并不是经典的程序设计语言, 更不能广为流传。在程序设计语言发展的历程中, 这种例子并不罕见。

以上主要讨论了语言及程序设计语言的一些基本常识。程序设计语言的设计是一门技术, 更是一门艺术, 优秀的程序设计语言可以广为流传数十年之久。实际上,



Algol、Lisp、Fortran、Pascal、C、Smalltalk、APL、ML、Simula、Modula-1/2、PL360 等经典语言都是历经了时间的考验才传承至今的。设计程序设计语言及编译器是一项富有挑战性且能带来无限成就感的工作，希望本书能引领读者进入程序设计语言的殿堂。

1.1.2 程序设计语言的翻译机制

在日常生活中，对于两个不同体系的自然语言而言，翻译工作可能是相当复杂的。众所周知，在语法、语义等体系上，不同的自然语言之间都存在着巨大差异，因此，至今自然语言的翻译工作仍然是以手工完成为主。当然，随着对自然语言理解的深入研究，或许在不久的将来计算机准确翻译自然语言会成为可能。那么，作为一种人工语言，程序设计语言的翻译又是如何进行的呢？

程序设计语言的复杂性远不及自然语言，因此，试图让计算机完成程序设计语言的翻译的想法并非奢望。从 20 世纪 60 年代开始，许多计算机大师就着眼于高级程序设计语言及其翻译工具的研究，成就了不少优秀的程序语言，例如读者熟知的 C、Pascal 等。在没有任何理论基础的情况下，大师们进行了艰苦的探索，构造了程序设计语言翻译体系的雏型。在经典编译技术中，程序设计语言的翻译模型主要有两种：解释与编译。

首先，笔者简单介绍一下“解释”。运用解释方法进行翻译的高级语言工具被称为解释程序。解释程序是高级语言翻译程序的一种，它将某一高级语言源程序作为输入，解释一句后就提交计算机执行一句，直至源程序读取完毕。解释程序在计算机应用中并不少见，例如，早期的 BASIC 语言的翻译程序就是解释程序。另外，网络浏览器也是一种解释程序，它是将 HTML 以及 JavaScript 等脚本程序进行解释执行的。再如，UNIX/Linux 的 Shell 也是解释执行的。

下面，再来谈谈“编译”。运用编译方法进行翻译的高级语言翻译器称为编译器（或称为编译程序），这也是本书讨论的重点。虽然编译器的目标语言并不唯一，但是，汇编语言或机器语言是最为常用的目标语言。通常，编译器指的就是把用高级语言源程序翻译成等价的计算机汇编语言或机器语言的目标程序的翻译程序。编译器把高级语言源程序作为输入，而以汇编语言或机器语言的目标程序作为输出。因此，在编译完成后，编译器会生成相应的目标程序。编译器是程序设计语言翻译中应用极广的软件，例如，早期的 Turbo C、Turbo Pascal，现代的 Delphi、VC++等都是编译器。

这里，笔者必须指出，“解释”与“编译”只是两种翻译方法或者说是方案，并不能直接与某种程序设计语言挂钩。例如，有些书认为 BASIC 语言是解释语言、C 语言是编译语言。严格地说，这种说法并不正确。笔者认为程序设计语言本身并不存在解释、编译之分，C 语言一样可以运用解释的方法设计它的翻译程序，BASIC 语言同样可以运用编译方法设计它的翻译程序。

随着人们对程序设计语言翻译技术的不断深入研究，有一种新的翻译方法逐步被人们接受，就是编译与解释相结合的方法。这种翻译方法的思路是先通过编译生成一个目标程序，但这一目标程序并不是真正的机器语言程序，而是一种编译器设计人员定义的一个较低级语言描述的程序（可能近似于汇编语言）。这一程序再经过相应的解释器解释执行，如图 1-1 所示。早期的 Lisp 语言就是应用这种翻译方式实现的，它使用一种类似于语法树形式的语

言作为目标程序的描述语言。

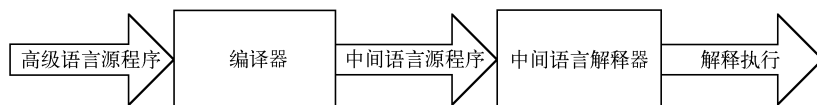


图 1-1 “编译+解释”翻译过程

近年来，人们热衷于编译与解释相结合的方法探索，提出了两种改进方案：即时编译、动态编译。

即时编译：先把源程序翻译成一种比较低级的内部语言描述的的程序，然后，在执行装载程序时，即时地将程序编译为目标机器的本地机器语言程序，然后直接执行机器语言程序。这种实现方案被广泛应用，例如，Java 语言编译器和微软的.NET 开发平台都是应用这种方法实现的。Java 生成的目标程序是称为字节码的一种较低级语言的程序。而.NET 开发平台生成的目标程序是面向 CLR（Common Language Runtime）的程序。

动态编译：为了避免即时编译的运行代价，开始时正常解释执行，在执行中，检查执行的热点（如循环、递归调用等），发现热点后动态编译这部分代码，生成本机的机器语言程序以提高执行效率。

关于三种翻译方式的优缺点在学术界争论已久，本书不进行深入分析，只是列出几个公认的观点，见表 1-2。笔者认为，作为一位有志于深入学习与研究编译技术的读者，了解翻译方式的基本知识还是有必要的。

表 1-2 翻译方式的特点

	执行效率	执行过程中的灵活性	源代码保密性
纯“解释”方式	慢	高	完全公开
纯“编译”方式	快	低	保密
“编译+解释”方式	一般	一般	一般

1.1.3 编译器的基本结构

作为系统软件，编译器的设计与实现是非常复杂的。对于一个相对复杂的系统，通常的解决方法是将系统分解成若干较小且便于处理的小系统，分别实现后将其组织成一个完整的复杂系统，这就是“分治法”的思想。实际上，计算机科学家正是运用这种思想来设计与实现编译器、操作系统、网络通信协议等复杂的大型系统软件的。

编译器的翻译过程是非常复杂的，但就过程本身而言，与自然语言翻译却有不少相近之处。例如，把英语句子翻译为汉语句子时，通常需要经过下列几个步骤：

- 1) 对句子中的每个英语单词进行识别。
- 2) 对句子的语法结构进行分析。
- 3) 分析句子的基本含义，进行初步翻译。
- 4) 修饰译文，使之更加符合汉语的表达习惯。
- 5) 将译文整理书写记录。



编译器的工作过程与自然语言翻译过程比较类似，亦可划分为五个阶段：词法分析、语法分析、语义分析与中间表示生成、代码优化、代码生成。

1. 词法分析

词法分析的任务就是对输入的源程序进行扫描分析，识别出一个个的单词 (Token)，并进行归类。这里的“单词”可以理解为源程序中具有独立含义的不可分割的字符序列，与自然语言中的单词概念有一定区别。一般而言，根据程序设计语言的特点，单词可以分为五类：关键字、标识符、常量、运算符、界符。以一个 C 语言的条件语句为例：

```
if (aa && 10==0) aa=100;
```

词法分析的结果是识别出如下的单词符号：

关键字	界符	标识符	运算符	常量	运算符
if	(aa	&&	10	==
常量	界符	标识符	运算符	常量	界符
0)	aa	=	100	;

这里，读者只需了解词法分析的任务即可。其算法实现将在第 2 章中详述。

2. 语法分析

语法分析的任务就是在词法分析的基础上，根据程序设计语言的语法规则（文法），把单词流分解成各类语法单位（语法范畴），如“语句”、“表达式”等。理论上讲，通过语法分析，编译器可以准确无误地判断输入源程序是否满足语言的语法规则。例如，语法分析可以判断如下语句是错误的。

```
if aa %% 10==9 aa=100;
for(i<0) i++;
```

不过，实际情况并非完全如此，这主要与文法定义的细化程度有直接的关系。当程序设计语言的设计人员把文法定义得比较宽泛时，也就意味着依据此语法规则，编译器不能在语法分析阶段发现所有的语法错误，只能将错误遗留给后续阶段处理。表面上看，语法分析并不像词法分析有直观的输出结果，而仅仅完成了输入源程序的语法判定工作。实际上，语法分析是编译器前面三个阶段（合称为前端）的主控模块。语法分析的设计思想与算法实现是经典编译理论重点讨论的话题，将在第 3 章中详述。

3. 语义分析与中间表示生成

语义分析与中间表示生成的任务就是在语法分析的基础上，分析各语法单位的含义，并进行初步的翻译，即生成中间表示形式。有时，这两个任务是密不可分的，故通常将其合并为一个阶段讨论。语义分析主要是检查输入源程序的语义是否正确，例如，变量使用前是否定义、同一作用域内变量是否重名等。中间表示生成将根据输入源程序的语义生成语义等价中间表示形式。中间表示是一种由编译器设计人员定义、使用的形式，对于用户是完全透明的。中间表示形式的定义是值得深入研究的，因为它直接决定了编译器前、后端的设计复杂度，也决定了编译器前端与目标语言之间的耦合程度。中间表示的形式也非常多，包括四元组、三元组、语法树、DAG 图等，并不一定是读者理

解的通常的代码形式。例如，lcc 的中间表示就是一种 DAG 的形式。当然，近似于汇编指令形式的四元组、三元组可能是最为常见的中间表示形式。由于这个阶段的内容繁多，笔者将以符号表系统、中间表示、表达式语义三个主题予以介绍。相关内容分布在本书的第 4~6 章中。

4. 代码优化

代码优化的任务就是对生成的中间表示进行优化处理，得到占用空间较小、执行效率较高的目标代码。由于中间表示是由计算机自动生成的，不可能像手工编码那样精炼，因此，编译器需要借助优化处理对目标代码进行精简。随着前端技术的相对成熟，优化技术逐渐成为编译技术领域的核心研究课题之一。第 7 章将详细讨论优化的相关话题。

5. 代码生成

代码生成的任务就是将中间表示形式翻译成目标语言描述的程序代码。本阶段是与目标计算机硬件系统结构密切相关的，其工作也非常复杂，如寄存器的调度、指令的选择、指令的调度等。并且，这个阶段还涉及许多与目标计算机硬件系统有关的优化技术，称为“指令级优化”。通常，目标代码的形式可以是汇编语言代码、机器语言代码、其他语言的代码。早期，目标代码多为机器语言代码。然而，随着汇编器、链接器的成熟，汇编语言代码逐渐取代了机器语言代码，成为目标代码的首选。近年来，随着虚拟机技术的出现与发展，目标代码的概念可能还会改变。例如，由于 .NET Framework 的盛行，许多编译器（如 Delphi .NET、ML、SmallTalk 等）都可以生成面向 CLR 的目标程序。第 8~9 章将详细讨论运行时环境与代码生成的相关技术。

以上五个阶段是一种典型的分类观点。事实上，并非所有的编译器都包含这五个阶段。例如，有些简单语言的编译器完全可以省去中间表示形式，直接生成目标代码。再如，一些并不苛求优化的编译器完全可以省去优化阶段。这里所提及的五个阶段只是大多数编译器的设计经验而已。除了以上五个阶段之外，有些编译器还包括两个非常重要的组成部分：符号表管理、出错处理。

6. 符号表管理

符号表是一系列用于记录各个分析阶段所获取信息（如变量名、作用域、函数形参等）的数据表格，这些表格的维护贯穿于整个编译过程。显然符号表的设计和管理是编译器构造过程中的一项极其重要的任务。这里，设计者更多关注的是表格的完整性与访问效率。第 4 章将详细讨论符号表的构造。

7. 出错处理

对于输入源程序的各种错误，编译器必须给出比较准确的出错报告，以使用户及时准确地定位、修改。编译过程的每一个阶段都可能检测出错误，其中，绝大多数错误可以在编译的前三个阶段检测出来。当然，真正的商用编译器并不限于此，可能还涉及更复杂的出错恢复。出错恢复主要是当编译器检测到错误之后，尽可能按照语义修正错误。注意，出错恢复的目的并不在于真正修复用户程序，而只是试图一次检测更多的错误。当然，这是基于编译器预测机制实现的。

图 1-2 描述了编译器各个阶段之间的关系，读者只需参考结构图了解各个阶段的任务及其输入、输出情况即可。

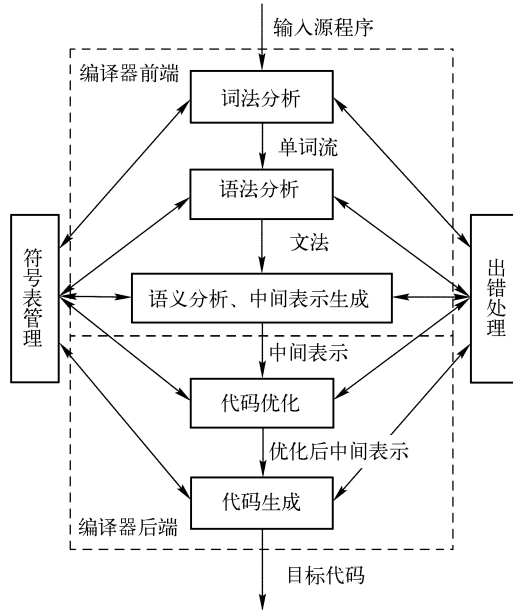


图 1-2 编译器结构图

这里，还有必要了解一个比较重要的概念：端（End）。

根据完成任务不同，可以将编译器的组成部分划分为前端（Front End）与后端（Back End）。前端主要指与源语言有关但与目标机无关的部分，包括词法分析、语法分析、语义分析与中间表示生成。后端主要指与目标机有关的部分，包括代码优化和目标代码生成等。“端”概念的提出对于编译技术的发展起到了至关重要的作用，它使编译器的框架更明晰，更利于集成与构造。

1.2 Pascal 语言基础

1.2.1 Pascal 语言简介

本书描述的编译器是笔者开发的 Neo Pascal，这是一个基于标准 Pascal 的实际编译器。Pascal 是由瑞士苏黎士理工学院 Niklaus Wirth 教授在 1971 年设计实现的，它的语法结构源于 Algol 语言。Wirth 教授以 17 世纪法国著名数学家 Pascal 的名字为这种优美严谨的程序设计语言命名。Pascal 盛行 30 年，成为 PC 平台上最受欢迎的程序设计语言之一。不过，随着 C 语言体系（C 语言、C++、Java 等）的不断成熟壮大，对于某些读者而言，Pascal 语言可能已经渐渐淡出了人们的视野。笔者之所以仍然选择标准 Pascal 语言作为设计蓝本，主要有如下几个原因：

- (1) Pascal 语言是一门严谨且优美的程序设计语言。
- (2) Pascal 语言功能非常强大，适合各种系统软件、应用软件设计。
- (3) Pascal 语言数据类型非常丰富。例如，集合类型、函数类型、指针类型等。
- (4) Pascal 语言的语义复杂度不如 C 语言，故有利于编译器设计与实现。同时，便于初

学者学习理解。

正由于上述优点，Pascal 仍然是算法设计的主要描述语言，也是 IOI（国际奥林匹克信息学竞赛）三种参赛程序设计语言之一。相信读者读完本书后，一定能够领略到 Pascal 的优美。

下面简单介绍一下 Pascal 语言，有 Pascal 语言基础的读者可以跳过这些内容。不过，Pascal 语言程序设计并不是本书的主题，因此，不可能花大量篇幅详述。笔者将对照 C 语言进行讲解，以便读者快速入门。当然，读者也可以参考《Pascal 语言程序设计》、《Turbo Pascal 用户手册》等。

1.2.2 Pascal 程序基本组成

请读者先阅读下列两段程序，比较计算圆周长程序的 C 与 Pascal 的描述。

例 1-1 计算圆的周长，代码见表 1-3。

表 1-3 C 与 Pascal 程序的比较

C 语言程序	Pascal 语言程序
1 float r,c;	1 program main(input,output); //程序首部
2 float Calc(float r)	2 var r,c:real; //声明部分
3 {	3 function Calc(r:real):real;
4 return 2*r*3.1416;	4 begin
5 }	5 result:=2*r*3.1416;
6 main()	6 end;
7 {	7 begin //执行部分
8 scanf("%f",&r);	8 read(r);
9 c=Calc(r);	9 c:=Calc(r);
10 printf("%f",c);	10 write(r);
11 }	11 end.

下面简单分析上述 Pascal 程序的基本结构。

一个完整的 Pascal 程序包含三个部分：程序首部、声明部分、执行部分。不同的 Pascal 编译器，程序注释的形式也有所不同，Neo Pascal 用“//”或“(* *)”表示。

1. 程序首部

第 1 行称为程序首部。program 是保留字，接着是程序名，Pascal 的程序名是任意的，并不一定是 main。其后的 input、output 是主程序参数，但这里的参数与 C 语言里的 argv 不同，并不是用于传递命令行参数的。而且不同的编译器对主程序参数的支持是不同的，Neo Pascal 并不关注这两个参数，仅仅是为了形式上兼容标准 Pascal 而已。

2. 声明部分

声明部分依次包含程序的单元声明、标号声明、常量声明、类型声明、变量声明、函数或过程声明。这里，简单说明三点：

- (1) 声明部分是可选的，对于简单的 Pascal 程序可以没有声明部分。
- (2) Pascal 程序的声明是有顺序要求的，必须严格按照上述顺序进行声明，不可倒置。



(3) 函数或过程声明就是用于声明自定义函数的。

而函数或过程声明又包含函数首部（例 1-1 中程序第 3 行）、声明部分（例 1-1 中省略了）、执行部分（例 1-1 中程序第 4~6 行）。在主程序声明部分声明的对象是在全局有效的。在子函数或过程声明部分声明的对象只在其声明函数或过程内有效，即局部有效。

3. 执行部分

执行部分就是用于描述函数或过程的语句序列。在例 1-1 中第 8 行、第 10 行就是调用输入/输出函数完成 IO，这与 C 语言的 scanf、printf 相似。第 9 行是调用 Calc 函数并将返回值赋给变量 c。Pascal 的语句种类与 C 语言非常类似，后续章节中将详细讲述。

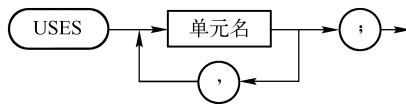
1.2.3 Pascal 的声明部分

声明部分主要包含单元声明、标号声明、常量声明、类型声明、变量声明、函数或过程声明。

1. 单元声明

单元声明用于声明程序所需包含的其他单元文件，功能与 C 语言的#include 类似，但是其处理与#include 是完全不同的。#include 是由预处理器识别的，而 Pascal 的单元声明 USES 是由 Pascal 编译器处理的。

声明格式：



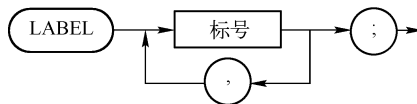
例如：

USES aa,bb,cc;

2. 标号声明

标号声明用于声明程序所包含的标号，功能与 C 语言的标号非常相似，只是 C 语言的标号并不需要事先声明就可以直接使用。

声明格式：



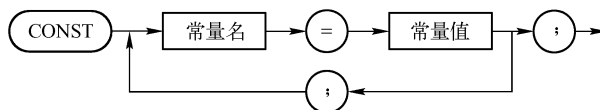
例如：

LABEL L1,L2,L3;

3. 常量声明

常量声明用于声明程序所使用的常量，与 C 语言的#define 宏定义是有所不同的。Pascal 的常量指的就是数学常量。

声明格式：



例如：

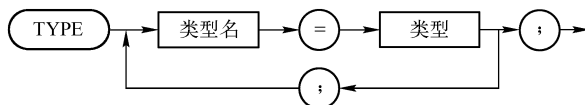
CONST a=12; b=1.2+3.1;c="r";

注意：常量声明部分必须以关键字 `CONST` 开始，等号右边的常量既可以是普通的常量，也可以是常量表达式，由编译器在编译阶段计算得到。标准 Pascal 中对于常量的类型是由编译器根据常量值自动判定的，并且只允许使用整型、实型、字符型、字符串型四类常量。不过，一些商用编译器对常量声明进行了扩展，例如，允许由用户指定常量的类型或者允许声明构造类型的常量等。

4. 类型声明

类型声明用于声明用户自定义的类型，功能与 C 语言的 `typedef` 非常相似。

声明格式：



例如： `TYPE a=INTEGER;b=ARRAY [1..10] OF INTEGER;`

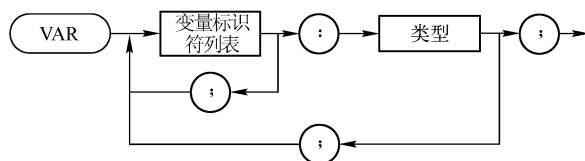
C 语言声明： `typedef int a;typedef int b[10];`

类型声明部分必须以关键字 `TYPE` 开始。“类型”可以是基本类型名用户自定义类型名，亦可以是匿名类型定义。关于类型定义的详细说明参见后续章节。

5. 变量声明

变量声明用于声明程序所使用的变量，功能与 C 语言的变量声明类似。

声明格式：



例如： `VAR a,b:INTEGER;`
`c:RECORD a,b:INTEGER;END;`

C 语言声明： `int a;`
`struct { int a,b; } b;`

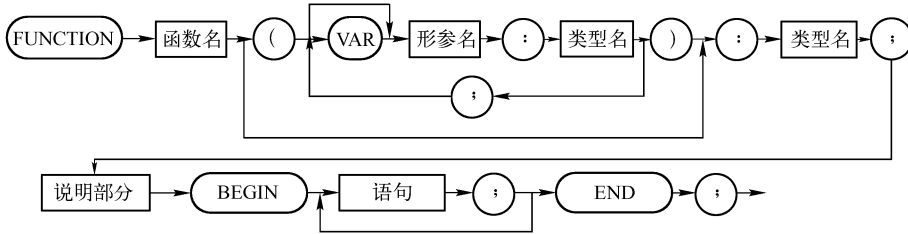
变量声明部分必须以关键字 `VAR` 开始。“变量标识符列表”是一个用逗号隔开的标识符序列。冒号后面的“类型”可以是类型标识符，也可以是匿名类型声明，如上例的 `c` 变量声明形式。

6. 函数或过程声明

函数或过程声明用于声明程序所包含的用户自定义函数或过程。函数或过程声明一般包含声明首部、声明部分、执行体部分三部分。其中，声明部分与主程序的声明部分类似，同样包含变量、常量、类型等声明部分。那么，函数或过程的声明部分中是否还可以包含函数或过程声明呢？答案是肯定的，标准 Pascal 规定可以嵌套声明函数或过程，即函数或过程仍然可以嵌套声明子函数或过程。然而，C 语言是不允许函数嵌套声明的。函数或过程嵌套声明的优劣在此不便评说，但包括 C、C++、BASIC、Java 等在内的许多高级语言都不允许。与其他高级语言比，Pascal 嵌套声明机制并无明显优势，反倒使编译器的存储分配算法变得复杂。这或许正是其他高级语言不允许这一声明机制的重要因素之一。考虑到读者可能是初

学编译技术，为了简化设计与实现，故 Neo Pascal 编译器不允许函数或过程嵌套声明，这与标准 Pascal 有一定区别。

声明格式：



例如：
`FUNCTION aa(a:INTEGER;b:CHAR):REAL;
 BEGIN ... END;`

C 语言声明：
`float aa(int a,char b) { ... }`

注意：以上的声明形式与标准 Pascal 规范是存在一定差异的。标准 Pascal 规定过程或函数的形参除了值参数、变量参数（与 C++的引用参数类似），还可以是过程参数、函数参数。Neo Pascal 参考 Delphi 采用函数过程指针代替函数、过程参数。标准 Pascal 语言规定函数或过程的形参类型或返回类型必须是类型名，不允许为匿名类型。形如“FUNCTION AA(A:ARRAY[1..10] OF INTEGER):BOOLEAN;”的函数首部是非法的。

另外，Pascal 语言还有两个特点：Pascal 语言是大小写不敏感的，这与 C 语言是不同的；Pascal 语言与 C 语言一样对于用户程序的阶梯型缩进是忽略的，这与 Python 语言是不同的。

1.2.4 Pascal 的类型

1. 类型简介

Pascal 语言的数据类型非常丰富。关于 Pascal 语言数据类型的分类标准并不唯一，笔者从编译器设计观点出发，将数据类型分为三类：基本类型（也可以称为简单类型）、构造类型、指针类型。

基本类型主要包含：整型、实型、字符型、字符串型、布尔型、枚举型。

构造类型主要包含：数组、记录（相当于 C 语言的结构）、集合、文件、函数过程等。

指针类型主要包含：数据指针。

标准 Pascal 语言中引入了函数过程类型，该类型源于函数式语言，在命令式语言中并不多见。Pascal 语言的函数过程类型与 C 语言的函数指针比较类似，实现机制相似，但还是存在概念的差别。不过，一些商用 Pascal 编译器已经不再区别函数指针与函数过程类型，统一使用函数指针实现。函数指针是现代程序设计语言的一个重要机制，对于面向对象编译技术研究具有现实意义，故后续章节中将深入讨论函数指针的实现技术。

2. 基本类型

根据不同的精度要求，C 编译器的实型、整型还可以细化为长整型、短整型、32 位整

型、单精度实型、双精度实型等。同样，各种版本的 Pascal 编译器也对整型、实型进行了细化。Neo Pascal 编译器参考了 Delphi 语言的数据类型，实现了其中最为常用的 12 种基本类型，见表 1-4。

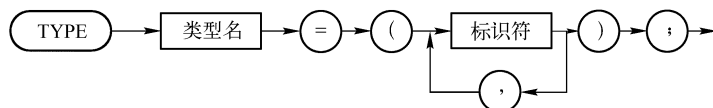
表 1-4 Neo Pascal 基本数据类型

整 型			实 型		
BYTE	unsigned char	单字节无符号整型	SINGLE	float	单精度型
WORD	unsigned short	双字节无符号整型	REAL	double	双精度型
LONGINT	unsigned int	四字节无符号整型	字符型		
CARDINAL	unsigned int	四字节无符号整型	CHAR	char	字符型
SHORTINT	char	单字节有符号整型	布尔型		
SMALLINT	short	双字节有符号整型	BOOLEAN	C++的 bool	布尔型
INTEGER	int	四字节有符号整型	字符串型		
			STRING	类似 C 的 char*	字符串型

3. 枚举类型

枚举类型（enum）通过预定义列出所有值的标识符来定义一个有序集合，这些值的次序和枚举类型说明中的标识符的次序是一致的。Pascal 的枚举与 C 语言的枚举基本类似。Pascal 规定枚举类型的枚举值个数不允许超过 255 个。

声明格式：

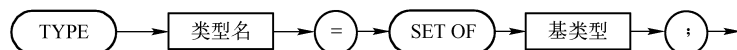


例如：VAR a:(SUN,MON,TUE,WED,THU,FRI,SAT)

4. 集合类型

Pascal 的集合（set）类型源自数学的集合概念。集合是以已知有序类型值的列表构造而成的，称已知序数类型为基类型。Pascal 对集合元素类型有严格的规定，集合元素的类型必须是有序类型，且所有元素值必须界于 0~255 之间。因此，基本类型中的 BYTE、CHAR 类型及枚举类型是有效的集合基类型。在标准 Pascal 中，还可以是子界类型，而 Neo Pascal 并不支持子界类型。与数学的集合类似，集合元素之间是不存在顺序关系的，即集合[2, 3]与[3, 2]是相等的。

声明格式：



例如：a:set of char;

表示一个集合值的最通用的方法是逐个枚举集合的元素，并用方括号[]括起来。

例如：

[3, 9, 15, 23]: 表示由 3, 9, 15, 23 组成的集合。



[l, 'p', 'z']: 表示由字符 l, p, z 组成的集合。

集合的运算:

+ : 集合“并”运算。

* : 集合“交”运算。

- : 集合“差”运算。

= : 检查两个集合所包含的元素是否相同。

<> : 检查两个集合是否不相等。

<=: 检查第一个集合中的元素是否都在第二个集合中出现。

>=: 检查第一个集合中的元素是否包含第二个集合中的所有元素。

in : 检查集合基类型的一个元素是否属于集合。

例如: [A, B, C]+[D]等于[A, B, C, D]。

[A, B, C]*[A]等于[A]。

[A, B, C]-[A]等于[B, C]。

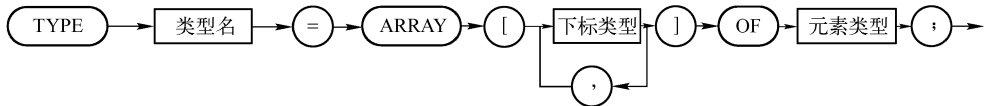
[A, B, C]=[A, B, C]等于 TRUE。

[A, B, C]<>[C, B, A]等于 FALSE。

5. 数组类型

数组类型 (array) 是程序中最常用的构造数据类型, 用来描述由固定数量的同一类型的元素组成的数据结构。Pascal 的数组与 C 语言的数组基本类似, 唯一的不同就是 Pascal 的数组下标可以由用户定义。例如, 用户可以定义一个下标从 20 开始的数组。而 C 语言规定数组下标从 0 开始。

声明格式:



“下标类型”可以是任意有序类型。由于其他有序类型作为数组下标的实际意义不大, 所以数组最常用的声明形式就是将子界类型作为“下标类型”的一种声明形式。不熟悉 Pascal 的读者对于子界类型可能不了解, 在 Neo Pascal 中也省略了子界类型, 这里读者只需了解子界类型实际上就是某一有序类型的一个子集。一般使用“下限..上限”形式表示子界类型。使用子界类型作为数组的下标类型的优点就是可以由用户指定数组的上、下限, 这一特性是 C 语言不具备的。

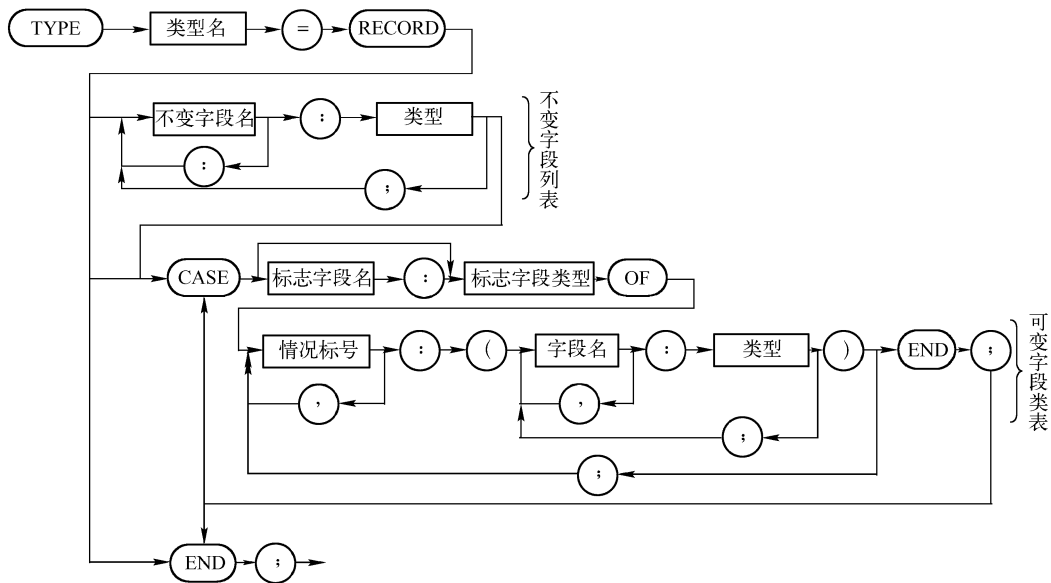
例如: a:ARRAY [0..10, 0..20] OF INTEGER;

C 语言声明: int a[11][21];

6. 记录类型

记录类型 (record) 是描述同一对象的一组类型可能不同的数据集合。使用记录类型实现了数据逻辑关系和存放形式上的一致。Pascal 的记录类型综合了 C 语言的结构、联合的基本功能, 其包含两类字段: 不变字段、可变字段。记录类型的声明形式比较复杂, 可能会对初次阅读产生障碍, 读者只需了解即可, 不必深究。

声明格式:



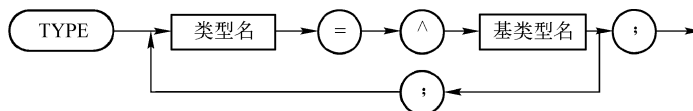
例如: a:RECORD a, b:INTEGER;END;

C 语言声明: struct { int a, b;} a;

7. 指针类型

指针 (pointer) 变量中存放的是某个存储单元的地址, 即指针变量指向的存储单元的地址。一个指针变量仅能指向某一种特定类型的存储单元, 该数据类型是在指针声明时确定的, 称为指针类型的基类型。标准 Pascal 并不提供无类型的指针, 不过, 一些商用编译器对此进行了扩展。

声明格式:



例如: a:^INTEGER;

C 语言声明: int *a;

指针的运算:

^: 间接访问运算。获取指针所指向存储单元里的数据, 与 C 语言的 “*” 运算类似。

@: 地址运算。获取变量的地址, 与 C 语言的 “&” 运算类似。

例如:

```

VAR    P:^INTEGER;
        I:INTEGER;
...
P:=@I;
P^:=65;
    
```

1.2.5 Pascal 的运算符

表达式是描述计算规则的一种语言结构。它由操作数、运算符及圆括号组成。操作数包



括变量、常量及函数等。运算符可以分为算术运算符、逻辑运算符、集合运算符、关系运算符以及其他运算符。Pascal 只有单目、双目两种类型的运算符。表 1-5 详细介绍了 Pascal 语言的运算符。

表 1-5 Pascal 的运算符

分类	运算符	目数	优先级	运算类型	返回类型	功能说明
算术	-	单目	1	整、实型	整、实型	取负
	NOT	单目	1	整型	整型	位非
	*	双目	2	整、实型	整、实型	乘法运算符
	/	双目	2	整、实型	实型	实数除
	DIV	双目	2	整型	整型	整数除
	MOD	双目	2	整型	整型	取模（取余）
	AND	双目	2	整型	整型	位与
	SHL	双目	2	整型	整型	左移
	SHR	双目	2	整型	整型	右移
	+	双目	3	整、实型	整、实型	加法
	-	双目	3	整、实型	整、实型	减法
	OR	双目	3	整型	整型	位或
	XOR	双目	3	整型	整型	位异或
逻辑	NOT	单目	1	布尔型	布尔型	逻辑非
	AND	双目	2	布尔型	布尔型	逻辑与
	OR	双目	3	布尔型	布尔型	逻辑或
	XOR	双目	3	布尔型	布尔型	逻辑异或
关系	=	双目	4	基本类型	布尔型	等于
	<>	双目	4	基本类型	布尔型	不等于
	<	双目	4	基本类型	布尔型	小于
	>	双目	4	基本类型	布尔型	大于
	<=	双目	4	基本类型	布尔型	小于或等于
	>=	双目	4	基本类型	布尔型	大于或等于
集合	*	双目	2	集合类型	集合类型	集合交
	+	双目	3	集合类型	集合类型	集合并
	-	双目	3	集合类型	集合类型	集合差
	=	双目	4	集合类型	布尔型	两个集合是否相等
	<>	双目	4	集合类型	布尔型	两个集合是否不相等
	>=	双目	4	集合类型	布尔型	左集合是否包含了右集合
	<=	双目	4	集合类型	布尔型	右集合是否包含了左集合
IN	双目	4		布尔型	左操作数是否为右集合的元素	
其他	+	双目	3	字符、字串	字符、字串	字符串连接
	@	单目	1	任意类型	指针型	取得操作数变量的地址
	^	单目	1	指针型	任意类型	取得指针变量指向存储单元的值

表 1-5 中优先数越小的运算符优先级最高。相对于 C 语言的 15 级优先级, Pascal 的优先级简单得多。

1.2.6 Pascal 的语句

程序设计语言主要可以分为命令式语言、函数式语言、逻辑式语言和面向对象语言。大多数命令式语言提供的语句种类比较相似, 见表 1-6。

表 1-6 Pascal 的语句

语 句 名 称	语 法 格 式	示 例
赋值语句	<变量名> := <表达式>;	a:=32;
IF 语句	IF <布尔表达式> THEN <语句>;	IF a=3 THEN a:=4;
	IF <布尔表达式> THEN <语句 1> ELSE <语句 2>;	IF a=3 THEN a:=4 ELSE a:=5;
WHILE 语句	WHILE <布尔表达式> DO <语句>;	WHILE a<3 DO a:=a+1;
REPEAT 语句	REPEAT <语句 1>; ... <语句 n>; UNTIL <布尔表达式>;	REPEAT a:=a+1; b:=b+1; UNTIL a>b;
FOR 语句	FOR <循环变量名> := <初值> TO <终值> DO <语句>;	FOR i:=1 TO 10 DO j:=j+1;
	FOR <循环变量名> := <初值> DOWNTO <终值> DO <语句>;	FOR i:=10 TO 1 DO j:=j+1;
BREAK 语句	BREAK;	
CONTINUE 语句	CONTINUE;	
GOTO 语句	GOTO <标号>;	GOTO L1;
复合语句	BEGIN <语句序列> END;	BEGIN a:=1;b:=2; END;
CASE 语句	CASE <分支表达式> OF <常量 1>:<语句 1>; ... <常量 n>:<语句 n>; END;	CASE a OF 1:i:=a+1; 2:i:=a+2; 3:i:=a+3; END;
内嵌汇编语句	ASM <汇编程序段字符串>, <参数信息> END;	ASM 'MOV AX, BX,' END;

内嵌汇编不是标准 Pascal 的语句, 而是笔者参考 Delphi 为 Neo Pascal 增加的一种功能强大的语句。它可以在用户程序中嵌入汇编代码, 以完成与硬件相关的功能。

至此, 笔者已经对 Pascal 语言的语法机制作了简单介绍。由于本书的目的不是以介绍 Pascal 语言为主, 所以没有必要更详尽地讲述 Pascal 语言程序设计的相关内容。对于某些语法细节, 笔者将在后续章节中解释。



1.3 开发环境与 Delphi 基础

1.3.1 开发环境与文件列表

Neo Pascal 是用 Visual Studio C++ 2005 实现的，因此，本节将以介绍 Visual Studio 2005 开发环境为主。同时，考虑到读者的应用习惯，笔者也提供了 VC++ 6.0、VC++ 2003、VC++ 2008 等不同版本的源代码工程，便于读者调试使用。

编译器项目的调试、编译并不复杂，与普通应用程序的开发类似，如图 1-3 所示。

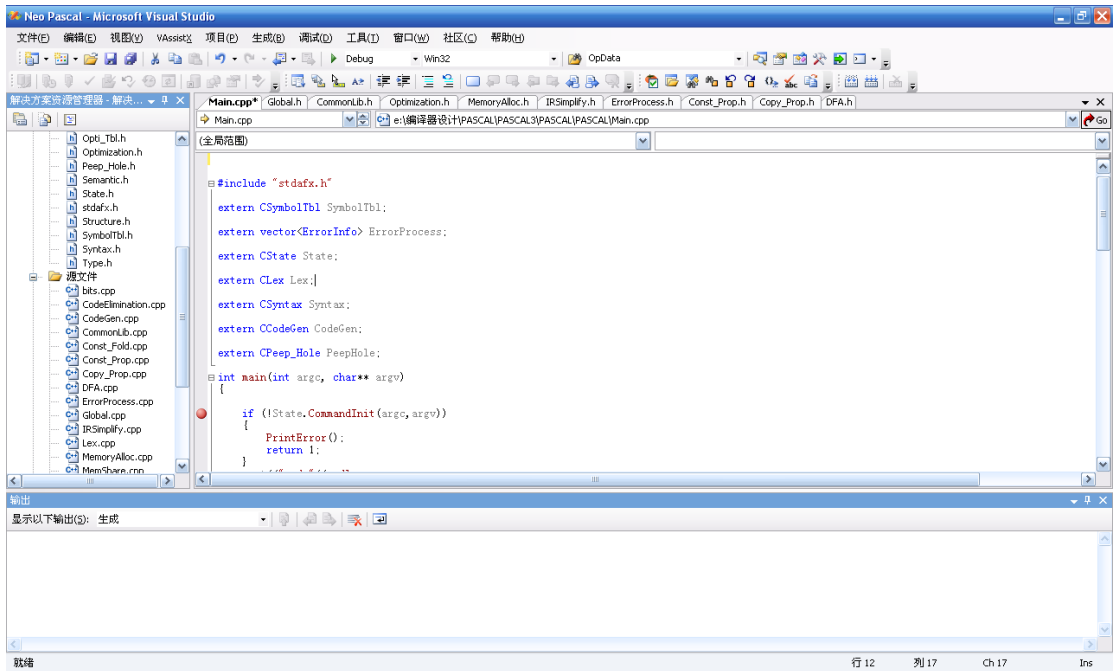


图 1-3 Neo Pascal 开发环境

关于 Visual C++ 环境的使用，相信读者已经熟悉，笔者就不再重复了。下面对 Neo Pascal 项目的文件列表进行简单说明，见表 1-7。

表 1-7 Neo Pascal 文件列表

文件名	相关描述
bits.cpp	位向量类，用于描述优化迭代中的位向量结构
CodeElimination.cpp	代码删除，用于删除不可到达代码及死代码
CodeGen.cpp	代码生成，用于生成最终的目标代码
CommonLib.cpp	公用函数库，用于描述一些公用的函数实现
Const_Fold.cpp	常量折叠，属于代码优化阶段的算法
Const_Prop.cpp	常量传播，属于代码优化阶段的算法

(续)

文 件 名	相 关 描 述
Copy_Prop.cpp	复写传播, 属于代码优化阶段的算法
DFA.cpp	数据流分析, 属于代码优化阶段的算法
ErrorProcess.cpp	出错处理
Global.cpp	全局函数接口的定义
IRSimplify.cpp	代码简化, 属于代码优化阶段的算法
Lex.cpp	词法分析器
MemoryAlloc.cpp	存储分配算法
MemShare.cpp	存储优化, 属于空间优化算法
Optimization.cpp	优化主控模块
Peep_Hole.cpp	窥孔优化
Main.cpp	主程序
Semantic.cpp	语义分析及 IR 生成模块
State.cpp	命令行处理
Structure.cpp	公用结构、类的实现
SymbolTbl.cpp	符号表
Syntax.cpp	语法分析器
Type.cpp	类型系统

1.3.2 Delphi 基础

1. Delphi 简介

本书以 Pascal 语言为蓝本讨论编译器的设计与实现, 因此, 建议读者安装一个通用的 Pascal 编译器作为参考。目前, 可供选择的 Pascal 编译器并不少, 例如, Turbo Pascal 7.0、Free Pascal、GNU Pascal (GPC)、Delphi 7 等。这里, 笔者还是推荐 Delphi 7。相对于其他编译器而言, Delphi 毕竟是一个曾经风靡一时的商用编译器。虽然 Delphi 本身的确存在一定不足, 尤其是 IDE 环境, 但是这丝毫不影响其成为一代经典之作。同时, Delphi 7 又是 Delphi 家族中口碑最好、应用最广的一个产品。因此, 笔者认为还是值得尝试的。鉴于有些读者可能对 Delphi 比较陌生, 下面, 笔者将对 Delphi 语言及其开发环境作简单介绍。

Delphi 作为一种可视化的编程环境, 提供了方便、快捷的 Windows 应用程序开发工具。Delphi 使用了 Microsoft Windows 图形用户界面的许多先进特性和设计思想, 采用了可重用的完整的面向对象的程序语言 (Object-Oriented Language), 是当今最快的编译器, 拥有领先的数据库技术。对于广大的程序开发人员而言, 使用 Delphi 开发应用软件, 无疑会大大提高编程效率。

Delphi 1.0 诞生于 1995 年, 作为 Borland 公司主推的 Windows 开发工具, 其设计初衷就是以 Visual Basic 作为竞争对手的。历经 4 年发展, Borland 终于推出了 Delphi 历史上的第一



个经典版本 Delphi 5.0。由此 Delphi 也一度成为最流行的编程语言之一，几乎可以与盛极一时的 C/C++媲美。

2002 年 8 月，Delphi 7 正式登上历史舞台，其灵活、易用、快捷等特性吸引了大批国人的眼球，成为当时应用软件开发的主要工具平台。在国内，其市场份额是 Visual Basic 无法比拟的。可惜的是 Delphi 7 却成了后继版本无法超越的传奇。此后，Borland 公司也陆续推出了 Delphi 8、Delphi 2005、Delphi 2006、Delphi 2007 等。不过，随着核心开发人才的流失，产品的竞争力有所降低，根本无法面对 Java、C#等新语言的挑战。由于开发工具领域的激烈竞争，Borland 公司也无心恋战，以至于 Delphi 失去了当年的辉煌。

2008 年 5 月 7 日，Borland 公司正式将其负责开发工具研制的子公司 CodeGear 出售给 Embarcadero 公司。目前，Embarcadero 公司研发的 CodeGear RAD Studio 2010 是最新的 Delphi 版本。

2. 集成开发环境

Delphi 7 的 IDE（如图 1-4 所示）主要包括 7 个部分：主窗口、组件面板、工具栏、窗体设计器、代码编辑器、对象观察器（Object Inspector）和代码浏览器。程序员借助于 IDE 可以很方便地完成创建、调试、修改应用程序等各种操作。

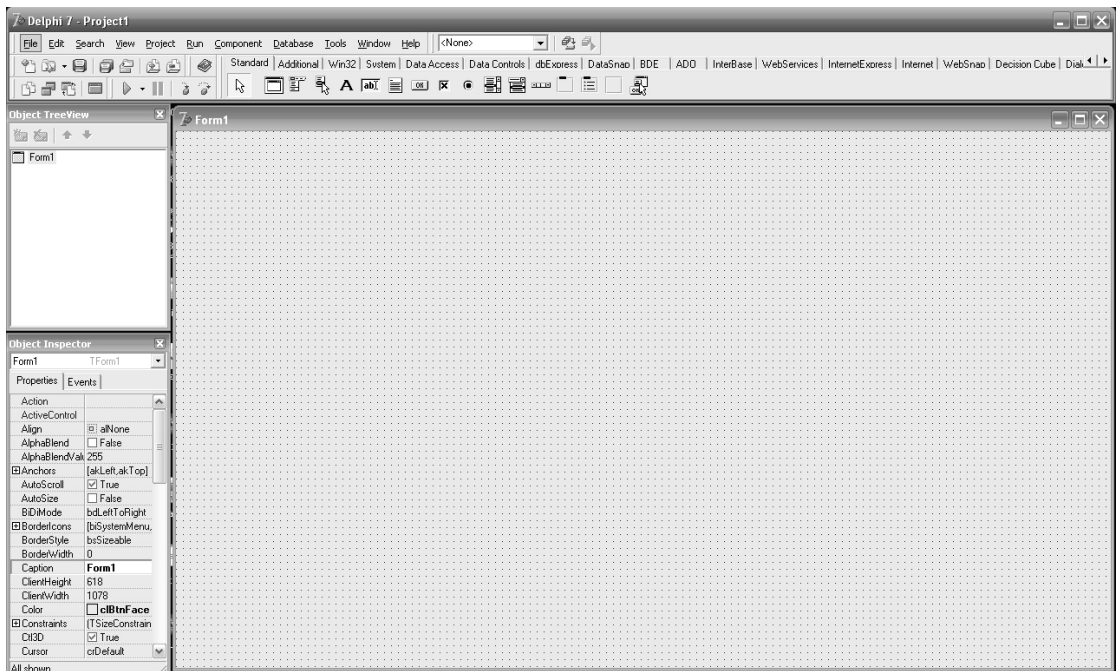


图 1-4 Delphi 7 的 IDE

(1) 主窗口。主窗口（如图 1-5 所示）是 Delphi IDE 的控制核心。它具有其他 Windows 应用程序的主窗口所具有的大多数特性。主窗口主要包括 4 部分：菜单栏、工具栏、标题栏和组件面板。

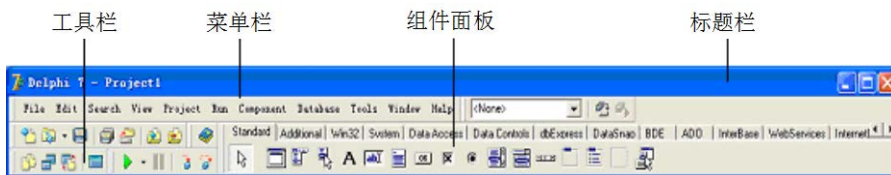


图 1-5 Delphi 7 的主窗口

(2) 窗体设计器。Delphi 是著名的 RAD 工具，可以方便地开发所见即所得的 Windows 应用程序。而 Windows 的窗口界面设计主要就是在窗体设计器内完成的。在创建新的项目时，窗体设计器是一个空白的窗口。可以把窗体设计器看成一块画布，在这块画布上可以描绘出各种各样的 Windows 应用程序。应用程序的用户界面正是由窗体实现的。只要从组件面板上选择一个组件并把它放到窗体上，就能够实现与窗体设计器的交互。可以用鼠标调整组件在窗体设计器上的位置和大小，还可以用对象观察器和代码编辑器来控制组件的外观和行为。

(3) 对象观察器。利用对象观察器（如图 1-6 所示），可以修改窗体或组件的属性，或者使它们能够响应不同的事件。属性（Property）是一些数据，如高度、颜色、字体等，它们决定了组件在屏幕上的外观。事件（Event）则是一种消息处理机制，它能够捕捉某种情况的发生并做出反应，像鼠标单击和窗口打开就是两种典型的事件。对象观察器包括 Properties 选项卡和 Events 选项卡，切换时只需在窗口上部单击所需选项卡的标签即可。至于对象观察器中显示哪个组件的属性和事件，取决于选中的组件。

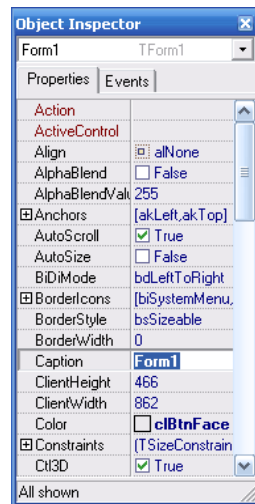


图 1-6 对象观察器

(4) 代码编辑器。代码编辑器是输入代码来指定应用程序行为的地方，也是 Delphi 根据应用程序中的组件自动生成代码的地方。代码编辑器每一页对应着一个源代码模块或文件。当向应用程序中加入一个窗体时，Delphi 会自动创建一个新的单元，并添加到代码编辑器顶部的标签中。当进行编辑的时候，快捷菜单提供了很多选项，如关闭文件、设置书签等。

(5) 代码浏览器。代码浏览器以一种树状视图的方式显示了列在代码编辑器中的单元文件。通过代码浏览器，可以方便地在单元文件中漫游或在单元文件中加入新的元素或者把已有的文件改名。要记住代码浏览器和代码编辑器有一一对应的关系。在代码浏览器中右击一个结点即可以看到该结点的可用选项。也可以通过修改主菜单 Tools→Environment Options 中的 Explorer 选项卡来控制代码浏览器的行为，如排序和过滤等。

3. 项目开发与调试

关于 Delphi 语言的细节，笔者无法在此尽解。下面，笔者详细讲述一个简单的控制台程序的开发过程，以便读者对 Delphi 项目开发及 IDE 的使用有初步了解。这里，之所以选择控制台程序是因为其更接近标准 Pascal。

(1) 新建工程。单击“File”菜单→“New”→“Other...”，即出现“New Items”窗口，

如图 1-7 所示。选择“Console Application”后，即显示代码编辑窗口。

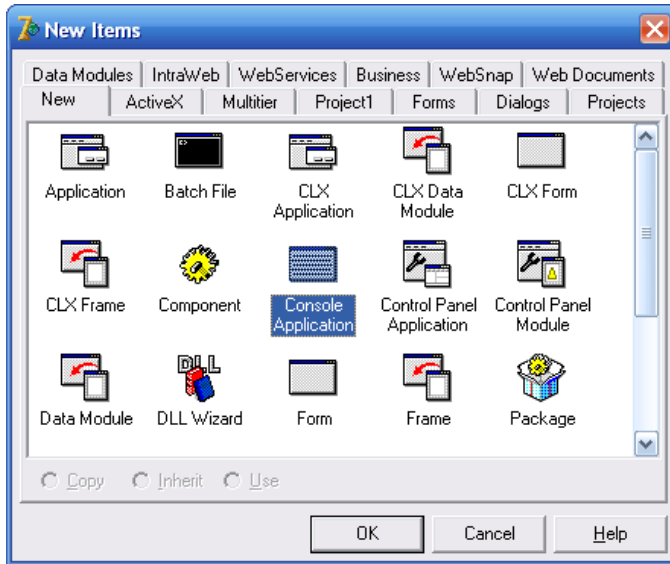


图 1-7 New Items 窗口

控制台程序是最有利于剖析编译器内核的，这绝不是信口开河。读者所见到的窗体、界面都是 Windows 提供的 API 接口而已，编译器需要做的只是像调用普通函数一样调用系统接口即可。至于 RAD 的效果完全是 IDE 的任务，并不是编译器所关心的。事实上，一切界面最终都将以源程序的形式输入编译器，编译器则根据约定传参调用系统 API 生成相应的界面。因此，学习编译器的最好办法就是避开那些外围的附件，直击其内核。

(2) 编写程序。在代码编辑窗口中，读者参照如图 1-8 所示的程序段，录入相应的程序代码。

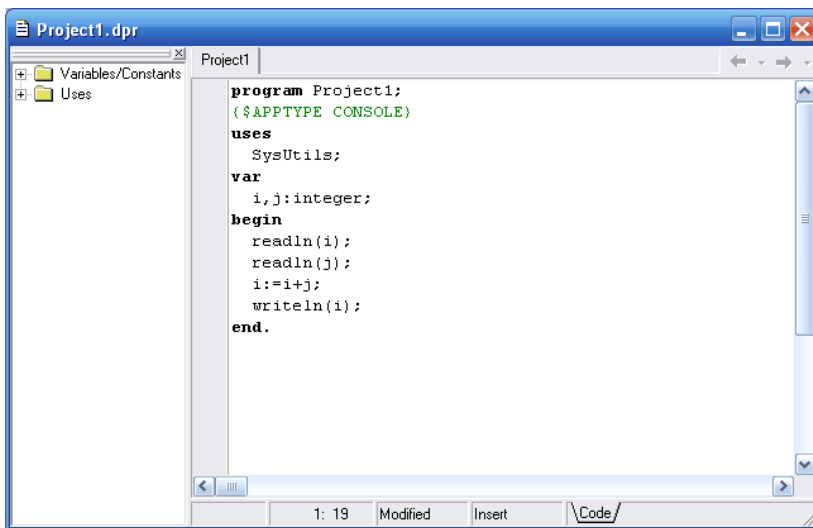


图 1-8 代码编辑窗口

这个程序的功能是从终端读入两个整数后，计算这两个整数的和并输出。这里，值得注意的是，Delphi 与标准 Pascal 还是有一定差异的。例如，第 2 行的“{\$APPTYPE CONSOLE}”是 Delphi 特有的编译指令，表示该项目是控制台程序。再如，第 4 行的“SysUtils”是 Delphi 的系统库，读者不必过多关注。

(3) 运行与调试。代码输入完毕，只需按“F9”键即可运行程序，亦可通过单击“Run”菜单→“Run”运行。Delphi 提供的调试工具是比较强大的，包括断点、监视、调用栈、线程等，读者可以通过“View”菜单→“Debug Windows”选择使用。下面，笔者介绍一下与编译器设计相关的调试功能——汇编调试。

在程序运行过程中，使用功能键“Ctrl+Alt+C”即可打开 CPU 窗口，如图 1-9 所示。

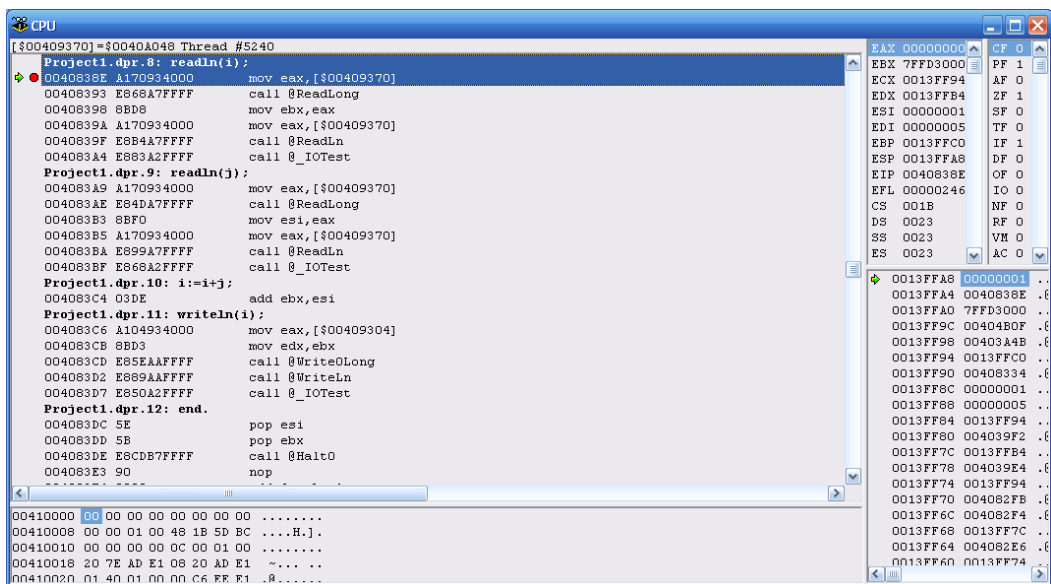


图 1-9 Delphi 的 CPU 窗口

在应用 Delphi 开发实际系统时，CPU 窗口主要的功能是帮助用户进行汇编级的调试，它非常直观地显示了编译得到的汇编代码，并且允许用户监视通用寄存器、内存、标志寄存器资源的状况。这与 Visual Studio IDE 的反汇编调试是非常类似的，相信有经验的读者应该比较熟悉。这里，笔者介绍 Delphi 汇编级调试的目的就是希望读者能够结合实例程序学习 Delphi 的编译机制，这对于深入理解本书的内容将是非常有利的。不过，可惜的是 Delphi 并不提供输出汇编程序代码的功能，因此，只能借助 CPU 窗口来查看编译得到的汇编程序代码。

(4) 联机帮助。单击“Help”菜单→“Delphi Help”即可打开联机帮助。与许多联机帮助一样，Delphi 的联机帮助也是一部非常经典的教科书。不熟悉 Pascal 语言的读者可以阅读其中的“Delphi Language Guide”，它对学习 Pascal 语言是有一定帮助的。当然，更重要的是它还提供了一份完整的 Delphi 文法，读者可以参见“Delphi Language Guide”→“Object Pascal grammar”部分，如图 1-10 所示。这是 Neo Pascal 文法的一个重要来源，因此，建议读者参考品味。

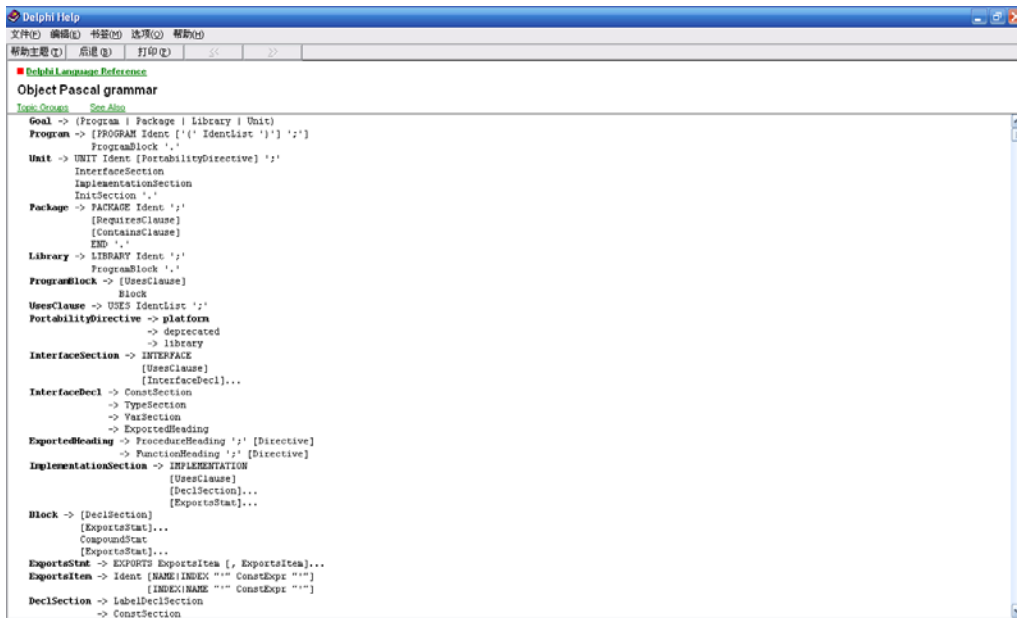


图 1-10 Object Pascal 文法

1.4 深入学习

为了帮助读者更好地学习与理解，在每一章结尾处，笔者将结合讲述内容推荐一些书籍或资源，供读者参考。

- | | | |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------|-------------------|
| 1、History of Programming Languages | Richard Wexelblat | Academic Press |
| 说明：本书是以介绍程序设计语言历史为主的材料。 | | |
| 2、The History of Programming Languages, Volume 2 | Thomas Bergin, Richard Gibson | ACM Press |
| 说明：本书是以介绍程序设计语言历史为主的材料。 | | |
| 3、Compilers and Compiler Generators | P. D. Terry | Rhodes University |
| 说明：本书是很不错的编译技术的入门教材，相对于“紫龙书”简单一些。 | | |
| 4、Structure and Interpretation of Computer Programs | Harold Abelson | The MIT Press |
| 说明：本书是麻省理工学院的经典教材，以介绍程序设计语言原理与翻译技术为主，初学者阅读有一定难度。 | | |
| 5、Pascal User Manual and Report | Kathleen Jensen, Niklaus Wirth | |
| 说明：Wirth 写的一本 Pascal 用户手册，是 Pascal 的权威著作。 | | |
| 6、Pascal ISO 7158:1990 | | |
| 说明：ISO 撰写的 Pascal 标准。 | | |
| 7、编译原理（紫龙书） | Alfred V. Aho, Monica S. Lam,
Ravi Sethi, Jeffrey D. Ullman | 机械工业出版社 |
| 说明：在编译技术中，“龙书”堪称最经典的著作。“龙书”原著共有三个版本，即 1977 年第一版（习称“绿龙书”，目前并没有中文版）、1986 年第二版（习称“红龙书”，中文版于 2003 年由机械工业出版社出版），2007 年第三版（习称“紫龙书”，中文版于 2009 年由机械工业出版社出版）。在“红龙书”的基础上，“紫龙书”增加了关于过程间分析、指令级并行等研究热点的讨论。同时，还引入了一些关于经典编译技术的最新观点。 | | |
| 8、Compiler Construction | Niklaus Wirth | Addison-Wesley |
| 说明：本书是 Wirth 于 1996 年撰写的以介绍编译器构造为主的材料。但本书内容比较精炼，不太适合作为入门级读物。 | | |

1.5 实践与思考

1. 结合本章所述的观点，试分析读者所熟悉的程序设计语言的特点。
2. 结合本章所述的观点，试比较 C#与 Java 语言的特点，重点关注翻译机制的差异。
3. 请列举 3 个 Pascal 与 C 语言的差异，并试述 Pascal 语言的主要特性。

1.6 大师风采——Niklaus Wirth

Niklaus Wirth: 瑞士计算机科学家，Pascal、Modula 语言创始人，结构化程序设计思想的提出者。1934 年 2 月 15 日出生于瑞士温特图尔。1958 年取得瑞士苏黎世工学院学士学位，1960 年取得加拿大莱维大学硕士学位，1963 年获得美国加州大学伯克利分校博士学位。毕业后进入斯坦福大学计算机科学系工作，在此期间，他实现了平生第一个语言 Euler。1967 年，Wirth 拒绝了斯坦福大学的挽留，首先回到瑞士苏黎世大学任职，而后又于 1968 年回到了他的母校苏黎世工学院工作。

“算法 + 数据结构 = 程序”这个著名的公式正是由 Wirth 提出的，他也因此获得了 1984 年图灵奖。Wirth 被誉为“Pascal 之父”。在程序设计语言与编译器设计领域，Wirth 的主要贡献在于：

- 设计与实现了 Pascal。
- 设计与实现了 Modula 与 Modula-2。
- 提出了结构化程序设计思想。
- 对著名的 BNF 进行了扩充，提出了 EBNF (Extended BNF)。

20 世纪 80 年代后期，Wirth 又致力于“Oberon 计划”的研究，设计并实现了一个超越 Pascal 与 Modula 的 Oberon 语言。与 Pascal、Modula 相比，Oberon 语言更安全，它引入了数组越界检查、垃圾回收、强类型检查等机制。当然，最令人期待的是 Oberon 不但支持应用软件设计，还支持操作系统的设计。2007 年，Wirth 实现了 Oberon-07，其目标机是 32 位 ARM 处理器。

第2章 词法分析

A language that doesn't have everything is actually easier to program in than some that do.

——Dennis M. Ritchie

2.1 词法分析概述

2.1.1 词法分析的任务

就编译器基本结构而言，词法分析（lexical analysis）是编译过程的第一阶段。实际上，在词法分析之前，有些编译器模型存在编译预处理阶段，C语言就是一个典型的例子。不过，大多数Pascal编译器都不存在预处理器。本书对编译预处理不作讨论。词法分析的任务是从左到右扫描与分析构成源程序的字符流（字符串），把字符流分解为多个单词（token）。每个单词都是具有独立含义且不可再分割的字符序列。在编译器框架中，完成词法分析任务的模块称为词法分析器。

读者可能对“单词”感到有点疑惑，不明白到底什么才是词法分析中所说的“单词”。试图回答这个问题就必须了解几个基本概念。这里，引入几个程序设计语言相关的名词。

- (1) 标识符：用户自定义的变量名、函数名等字符串。
- (2) 关键字：具有特殊含义的标识符。
- (3) 运算符：例如+、-、*、/等。
- (4) 常量：例如3.24、92等。
- (5) 界符：具有特殊含义的符号，如分号、括号等。

通常，程序设计语言的关键字也就是它的保留字，编译器不允许此类单词另作他用。不过，这并不是绝对的。有些语言关键字并不一定是保留字，例如，Fortran语言允许用户定义与关键字相同的标识符。因此，读者应该注意关键字与保留字是有区别的。

上述五类元素是最基本的原子，任何表达式、语句、函数、程序都是由这几类元素排列组合而成的。词法分析所讨论的“单词”就是这五类元素的总和。

例 2-1 C语言程序段的词法分析结果见表 2-1。

表 2-1 词法分析的单词流

源程序字符流	词法分析的逻辑结果								
int i,j;	int	i	,	j	;	for	(i	
for (i=1;i<10;i++)	=	1	;	i	<	10	;	i	
j=j+1;	++)	j	=	j	+	1	;	

注意，表 2-1 的单词流并不是词法分析器真正的实际输出结果，只是一种逻辑表示而

已。更详细的形式将在后续章节中讨论。根据单词的分类标准，可以将单词作如下归类，见表 2-2。

表 2-2 例 2-1 单词流的分类

关键字	int	for		
标识符	i	j		
运算符	=	++	<	+
常量	10	1		
界符	,	;	()

这里，读者可能会有两个疑问：

(1) 为什么“++”运算符不会分解为两个“+”运算符呢？

(2) 为什么将“int i”分解为“int”和“i”，而不是“int i”呢？

下面，笔者来解释一下这两个问题。在实际编译器设计中，任何词法分析器都必须满足一个原则，就是在符合词法定义的情况下进行超前搜索识别。例如，当 C 语言词法分析器读入了一个字符“+”后，由于 C 语言中存在“++”、“+=”运算符，那么，词法分析器会继续读入下一个字符。如果下一个字符是“+”或“=”时，词法分析器就将这两个字符作为一个运算符。然而，如果下一个字符不是“+”或“=”时，词法分析器就将前一个字符“+”作为一个运算符记录下来后，继续识别下一个单词。

根据这个原则，就可以解释为什么“int”没有被识别为“i”、“n”、“t”了。根据 C 语言标识符（关键字只是有特殊含义的标识符）定义的规则，标识符必须以字母或下画线开头，后跟字母、数字、下画线的任意组合。因此，当读入“i”后，继续读入“n”，由于“in”是合法的标识符，则继续读入“t”。直到读到“ ”时，发现“int”不满足标识符的定义，则将“int”记录下来即可。

那么，为什么需要超前搜索识别呢？这个问题非常简单，超前搜索保证了词法识别能遵守单词的极大搜索原则。也就是说，在满足词法定义的情况下，尽可能使单词的长度最长。那么，通过超前搜索一个字符，直到读入这个字符后，单词不能满足词法定义，也就完成了一个单词的识别了。如果没有极大搜索原则，当读入类似“int”的字符串时，词法分析器就将分解成“i”、“n”，“t”三个单词。因为，它们都是满足 C 语言标识符的定义规范的。这样，永远也不可能识别出长度大于 1 的标识符了。显然，这样的结果是不能接受的，因此，超前搜索识别的存在是必然的。

不过，词法分析器的设计难度很大程度上依赖于程序设计语言本身的规范。例如，C 语言中“_name”是完全合法的标识符，而在早期的 Pascal 语言中，“_name”却是非法的标识符，因为 Pascal 语言的标识符定义原则是以字母开始，后跟字母、数字的任意组合。再如，Fortran 语言规定除字符串中的空格以外，编译器将忽略其他空格，这意味着在 Fortran 中“int i”与“inti”是完全相同的。虽然这只是一个很小的规定，但是，读者可以尝试使用超前搜索识别单词。不难发现，这个规定给词法分析的设计带来很大的困难。

当然，程序设计语言毕竟就是人工语言，是人为定义而成的。在设计程序设计语言时，完全可以避免某些不利于编译器实现的语言规则。C、Pascal 等语言的设计就吸取了先辈的教训，避免了一些人为造成的不必要的麻烦。因此，在设计一门程序设计语言时，应该尽可能避免关

键字非保留字、空格忽略等类似情况的发生，否则将给词法、语法分析造成相当的障碍。

2.1.2 单词的分类

前面，笔者已经介绍了单词的五种类型，即标识符、关键字、运算符、常量、界符。从程序设计语言设计的角度而言，这种分类方式是一种非常合理的观点。不过，从编译器设计的角度而言，这种分类却是值得商榷的。例如，Pascal 语言的“DIV”、C 语言的“sizeof”应该归入关键字还是运算符？再如，C 语言的“，”应该归入运算符还是界符呢？“，”既可以是逗号运算符，也可以是表达式列表的分隔界符。

根据词法定义规则，编译器或许可以将“DIV”、“sizeof”归类，但绝对不可能将“，”准确归类。实际上，编译器并不需要关心“DIV”、“sizeof”等到底是关键字还是运算符。故笔者提倡将关键字、运算符、界符合并为一类，称为系统单词类。也就是说，从编译器的观点来看，只需将单词分为标识符、常量、系统单词三类即可。

2.2 词法分析器的设计

2.2.1 识别单词

通常，设计一个完整的词法分析器需要考虑三个部分：输入源程序字符流、按照词法定义识别单词、输出单词流。本节将围绕这三个部分讨论词法分析器的设计与实现。

首先讨论如何输入源程序字符流。读者或许认为源程序字符流输入无非就是将源程序读入内存的过程，这应该是非常简单的，这个想法不无道理。在 2GB 内存成为标准配置的今天，将源程序文件完全读入内存确实并非难事。不过，在早期硬件条件相对较差时，就不允许编译器将源程序文件完全读入内存了。许多编译器设计者必须考虑如何让编译器在极为有限的内存中完成编译工作，同时，还要尽可能将有限的内存留给符号表使用。因此，初始化时不可能将源程序文件完全读入内存。那么，是不是每次都是直接从外存读取数据呢？众所周知，内、外存数据交换是有代价的，尤其是外存访问速率较低时，频繁的内、外存数据交换可能耗时较长。由此，设计者必须在时间与空间之间作出抉择。通常的解决方法就是设置一个内存缓冲区，每次以数据块为单位读取源程序文本存入内存缓冲区中，以备识别程序处理。这样，编译器设计者就必须考虑数据块的大小以及缓冲区的访问策略等问题。不过，在今天看来，无论是内存容量还是操作系统支持，笔者认为详细讨论此类话题的实际意义已经不大了。

前面，笔者简单引入了缓冲区的概念。下面，将切入正题，重点讨论单词的识别。

实际上，识别单词就是一种特殊的子串搜索过程。关于字符串子串搜索算法原理与实现，读者应该并不陌生。或许有些读者还会联想到经典的 KMP 模式匹配算法。那么，词法分析器到底是使用哪种算法的呢？要回答这个问题，就必须理解词法分析器与传统子串搜索算法的区别：

(1) 起始位置是已知的。在词法分析器搜索一个单词之前，子串在父串中的起始位置是已知的。单词的起始位置就是上一个单词结束时搜索指针的位置加上其后的无关字符个数。无关字符应视具体的实现语言而定。例如，C 语言中的不是出现在字符、字符串常量中的空格、回车、Tab 等都视为无关字符。然而，Python 的语句行前部的空格、Tab 对于程序语义

是有影响的，因此，就不能视为无关字符。对于有编译预处理器的编译器而言，剔除无关字符可以由编译预处理器完成。不过，编译预处理器并不能将源程序文件中的所有无关字符一概剔除，必须保证单词与单词之间存在必要的分割字符。例如，“int a;”中的无关字符是不能在编译预处理时剔除的，因为其中的空格对于词法分析是有实际意义的。

如图 2-1 所示，根据超前搜索识别的原则，当搜索指针移动到“e”后面一个字符时，发现单词“while”识别完成并将其提交。对于 C 语言而言，本例中的两个空格是无关字符，故搜索指针直接跳过这两个空格即可。当搜索指针移动到“(”时，由于“(”不是无关字符，那么，下一个待识别单词的起始位置就是“(”的位序。搜索指针从起始位置开始识别单词。假设将“(”字符替换为“@”，显然“@”不是任何有效的 C 语言单词的起始字符。同时，也不是 C 语言规定的无关字符，因此，并不能跃过“@”继续识别，只能中断分析并提示出错。

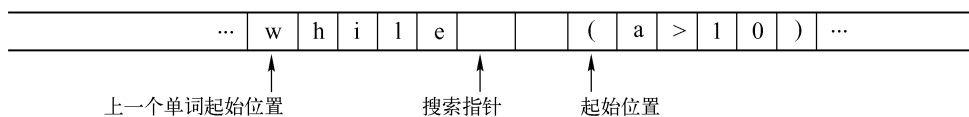


图 2-1 词法分析器起始位置

(2) 子串是未知的。在开始识别一个单词之前，编译器是无法预知下一个单词的种类及其形式的。因此，词法分析器不可能像传统子串搜索算法那样给定子串，更不可能像 KMP 算法那样预先分析处理子串。

现在可以设计一个识别标识符的小程序。不同的程序设计语言关于标识符的词法定义是不尽相同的，例如，C 语言规定标识符可以是以字母、下画线开头后跟字母、下画线、数字的任意组合。而 Pascal 语言规定标识符可以是以字母开头后跟字母、数字的任意组合。由于在识别单词前，待识别单词的起始位置是已知的，因此，不必过多考虑。这个小程序的设计应该是非常容易的。下面，以 Pascal 语言为例，给出简单的设计流程，如图 2-2 所示。

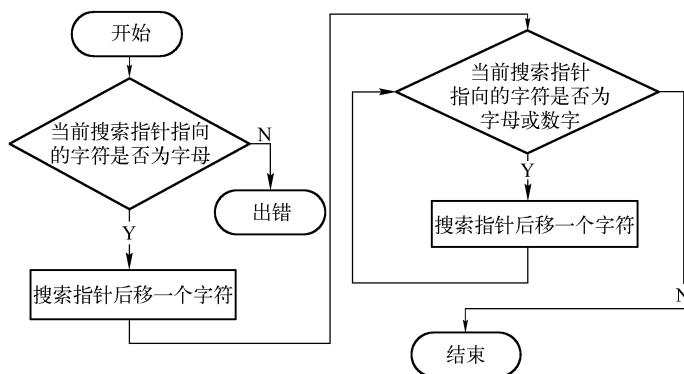


图 2-2 识别 Pascal 标识符的程序流程图

2.2.2 转换图

从图 2-2 中，不难发现，其中只有“搜索指针后移一个字符”一种处理动作（方框）。

那么，读者不妨想象一下，词法分析器是否就只有这种处理动作呢？仔细分析手工识别单词的过程后，就可以发现事实确实如此。既然词法分析器的流程中条件判断（菱形框）比较复杂，而处理动作非常单一，因此，可以将普通流程图改造成一种专门用于描述条件判断的流程图。具体改造步骤如下：

- 1) 把图 2-2 中所有上、下菱形（判断）之间的箭头用圆表示。
- 2) 把图 2-2 中所有的菱形直接用箭头线表示，箭头上写上原菱形的判断成立与否的条件，即可得到图 2-3。

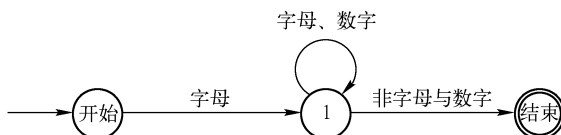


图 2-3 识别 Pascal 标识符的状态转换图

这里省略了出错处理，由于词法分析器的出错处理可以统一完成，所以无需重复考虑。

那么，将图 2-2 改造成图 2-3 的意义是什么呢？实际上，主要针对词法分析器的流程特点，突出了流程中最为复杂的部分，省略或简化相对简单的部分，这样的改造便于更直观地分析问题的本质。这种分析问题的方法在软件系统分析与设计过程中较为常用，读者应该善于改进一些已有的工具，使之更有利于分析问题。

图 2-3 对于词法分析器设计具有非常重要的作用。在形式语言学中将这种图称为转换图（transition diagrams），亦称为状态转换图。

转换图是一个有向图。在转换图中，圆表示状态，状态之间用箭弧连接。箭弧上的标记表示在箭弧始结点的状态下读入给定字符或字符集时，当前状态转到箭弧终结点所示状态。例如，图 2-3 在开始状态下，若输入字母，则转换到状态 1。一张转换图只包含有限个状态（即有限个结点），其中，有一个是初始状态（图 2-3 中的“开始”状态），而且必须至少有一个终止状态（用双圈表示，读者可以将终止状态理解为可终止状态，而其他状态即为不可终止状态）。那么，如何运用转换图完成单词的识别呢？实际上，识别一个单词就是从初始状态经过数次转换到达终止状态的过程。如果某一个字符串的识别过程不能到达终止状态，说明输入字符串不能被该转换图接收。

下面，应用转换图来分析一个复杂的实例。在 Pascal 词法分析器中最复杂的就是识别实型常量的逻辑流程。Pascal 的实型常量有两种形式：普通小数形式、科学记数法的小数形式，例如，2.34、23.4E32、46.1e-43、30.e12 等都是合法的实型常量，而诸如 .32、31.3e、32.3E13.2 等形式都是非法的实型常量。读者可以参考图 2-4。

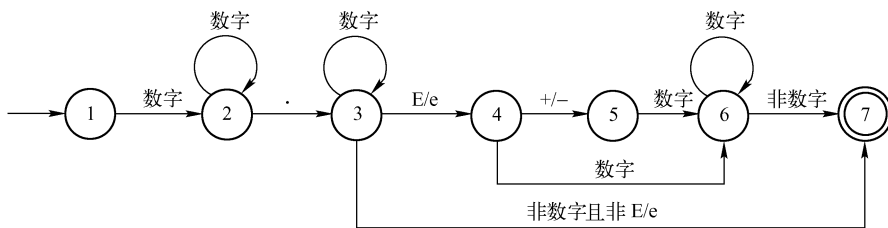


图 2-4 识别 Pascal 实型常量的转换图

读者可能会有疑惑，为什么图 2-4 没有考虑实型常量的+/-号？实际上，在词法分析时，只能将“-”识别成运算符，但无法确定其到底是用作负号还是减号。这个工作将在语法分析阶段完成。因此，词法分析器无法确定是否将“-”直接与后续常量组合。仅从识别实型常量的角度而言，图 2-4 已经足够完整了。但从实际编译器设计的角度而言，可能还需作少量改进。这里，值得注意的是标准 Pascal 中的数组声明形式，例如：

```
ARRAY [1..10] OF INTEGER;
```

这种声明形式对图 2-4 提出了挑战。在上例中，当词法分析器读取第二个“.”时，将直接转到状态 7，随即完成单词“1.”的识别，这与原始的语义是完全相悖的。实际上，正确的识别结果应该是得到“1”、“.”、“10”三个单词。

下面，再来看看如何应用图 2-4 识别字符串“92.3E1”。这是一个合法的实数常量表示形式。首先，从开始箭头进入状态 1。在状态 1 时，读取到数字即转入状态 2。在状态 2 时，读取第二个字符“2”，根据圆弧箭头指示，状态 2 遇到数字时仍然停留在状态 2。在状态 2 时，读取第三个字符“.”，状态 2 即转入状态 3。在状态 3 时，读取第四个字符“3”，状态 3 遇到数字时仍然停留在状态 3。在状态 3 时，读取第五个字符“E”，状态 3 即转入状态 4。在状态 4 时，读取第六个字符“1”，状态 4 转换为状态 6。根据超前搜索识别原则，词法分析器继续读取下一个字符，且下一个字符必定不是数字。在状态 6 时，读取一个非数字字符即转入状态 7。至此，状态转换停留在状态 7 上，状态 7 是终止状态（即双圈状态），故判定输入字符串合法有效。表 2-3 给出几个实例分析，请读者仔细理解。

表 2-3 识别实型常量的实例分析

输入字符串	状态转换	识别结果
1234	1, 2, 2, 2, 2, 7	有效（到达终止态 7）
192.32	1, 2, 2, 2, 3, 3, 7	有效（到达终止态 7）
192a2	1, 2, 2, 2	无效（到达非终止态 2）
A12s	1	无效（到达非终止态 1）
1.104E19	1, 2, 3, 3, 3, 3, 4, 6, 6, 7	有效（到达终止态 7）
12E12.98		
-12.12		
+32.E23		
43.1E3.9		

请读者自行补充表 2-3 中空白区域，以加深对转换图使用方法的理理解。只有理解了转换图与普通流程图之间的联系，才能进行词法分析器的设计。Neo Pascal 语言的完整转换图将在 2.3 节详细分析。

2.2.3 构造词法分析器

前面已经阐述了词法分析器如何识别单词及如何应用转换图描述词法分析器等问题。下面将详述如何高效地应用转换图实现词法分析器。



转换图是由程序流程图改进而成的。同样，转换图也可以等价地转换为程序流程图。完整的转换图就是词法分析器的程序流程图，是实现词法分析器的依据。早期的编译器都是采用这种方式手工编码实现的。这种实现方式对编译器设计者的编程技巧要求非常高。如果非常机械地完全按照转换图的描述编码，所得到的词法分析器的执行效率可能比较低，必须运用编程技巧使词法分析器尽可能精悍高效。因此，这种编码实现方式存在一个严重的不足：程序的耦合度高。当词法定义发生改变时，词法分析器的改动将非常大，甚至是颠覆性的。然而，为什么早期的编译器大多采用这种方式编码实现呢？最主要的原因就是这种方式实现的词法分析器运行时所需的存储空间的代价相对较小。当然，这与大师们的精湛技艺是分不开的。下面，笔者介绍一种更简单的实现方法，它是词法分析器核心理论研究的成果。

经过计算机科学家的多年探索，设计一个能够根据转换图模仿人工应用转换图识别单词的过程完成词法分析的程序并非完全不可能的。读者可以想象一下，模仿人工识别的过程用程序实现并不复杂。主要的问题就是如何将转换图输入计算机，并且可以由词法分析器进行处理。

显然，转换图的形式是不利于输入计算机的，众所周知，最便于程序处理的数据结构就是表格。如果能将转换图转化为表格或者类似的数据结构，那么，词法分析器就可以识别并处理转换图了。事实上，将转换图等价转换为二维表格的过程可以参照以下四个步骤：

- 1) 将转换状态集合作为行，将所有箭弧上的字符集合作为列。
- 2) 按箭弧填写表格中的单元格。填写方法如下：在箭弧始结点状态所在的行和箭弧上的标记所在的列所指示的单元格中填入箭弧终结点状态。依此法将转换图中的所有箭弧填入表格中。
- 3) 在空单元格内填入另一个特殊标记“-1”，表示该行状态下不允许读入该列“字符”。
- 4) 用特殊标记标识转换状态集中的开始状态与终止状态，这里分别使用“*”与“#”。

按照上面的步骤，可以将图 2-4 转换为表 2-4 的形式。

表 2-4 Pascal 实型常量转换表

	0	1	2	3	4	5	6	7	8	9	+	-	E	e	.	其他
1*	2	2	2	2	2	2	2	2	2	2	-1	-1	-1	-1	-1	-1
2	2	2	2	2	2	2	2	2	2	2	-1	-1	-1	-1	3	-1
3	3	3	3	3	3	3	3	3	3	3	7	7	4	4	7	7
4	6	6	6	6	6	6	6	6	6	6	5	5	-1	-1	-1	-1
5	6	6	6	6	6	6	6	6	6	6	-1	-1	-1	-1	-1	-1
6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7
7#	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

注意，“#”、“*”之类的特殊标记可以任选。另外，这里的“其他”是简略书写，完整

的转换表应该罗列该语言可以接受的所有字符（除数字、+/-、E/e 外）。

表 2-4 的功能完全等价于图 2-4 的转换图。表格的行表示原转换图的状态集合，列表示转换图中所有字符集合。表格中的单元格表示所在行状态下读入一个所在列的字符时转入的状态。读者可以自行选择实例进行分析。

下面，就来设计一个基于转换表进行自动识别单词的程序，即简单的词法分析器。程序 2-1 并非完整的 Neo Pascal 词法分析器，只是旨在说明如何设计一个基于转换表的词法分析器。理解了此算法思想，也就理解了词法分析器的核心。

程序 2-1 Lex.cpp

```

1  int LexTbl[128][100];
2  void LexParse(string szSourceTxt)
3  {
4      int iPos=0;
5      bool bTag=true;
6      int iState=0;
7      string szBuffer="";
8      while (iPos<szSourceTxt.length() && bTag)
9      {
10         szBuffer=szBuffer+szSourceTxt.at(iPos);
11         int iNextState=LexTbl[szSourceTxt.at(iPos)][iState];
12         if (iNextState== -1)
13             bTag=false;
14         else
15             if (!IsTerminal(iNextState))
16                 iPos= iNextState;
17             else
18                 {
19                     iPos--;
20                     szBuffer.erase(szBuffer.end()-1);
21                     ProcessToken();
22                     szBuffer="";
23                     iState=0;
24                 }
25     }
26 }
```

第 1 行：声明转换表，第一维表示语言的字母表，第二维表示状态。

第 4 行：iPos 表示源程序当前读取位置，即读取指针。

第 6 行：iState 表示当前状态，初始设置为 0，即表示开始状态为 0。

第 7 行：单词缓存区。

第 8~25 行：循环读取源程序文本字符，直至源程序文本尾。bTag 标志指示当前单词是否出错。

第 10 行：将源程序的当前字符加入单词缓存区。

第 11 行：查询当前状态下读取当前字符时的转换状态，这是程序的核心部分。



第 12 行：如果查询得到的转换状态为-1，表示当前状态下不允许读取当前字符。

第 13 行：置出错标志 `bTag` 为 `true`。

第 15 行：调用 `IsTerminal` 函数判断转换状态是否为终止状态。

第 16 行：将当前状态设为查询得到的转换状态。

第 17 行：当前状态下读取当前字符时，转换状态为终止状态，即得到一个单词。

第 19 行：由于超前搜索识别，必须将当前读取位置回退一个字符。

第 20 行：将单词缓存区回退一个字符。

第 21 行：调用 `ProcessToken` 处理单词。

第 22 行：将单词缓存区清空。

第 23 行：当前状态置为 0。

程序 2-1 虽然只有 26 行，却是词法分析器的核心。请读者务必理解该程序的设计思想。另外，为了便于设计与实现，本书所有的程序都将使用 C++ 描述。在程序 2-1 中没有实现 `ProcessToken` 函数，这个函数的主要功能是处理与输出单词。下面，就详细讨论这个问题。

词法分析器识别得到的单词流是语法分析器的输入，那么，词法分析器需要提供哪些信息给语法分析器呢？以什么方式传递给语法分析器呢？这些问题是需要设计者回答的。

在编译器设计中，最常见的单词存储形式就是三元组（单词 ID，单词备注，单词行号）。

单词行号一般包含两部分信息：单词所在的源程序文件名、单词在源程序文件中的行号。这些信息主要用于出错处理，可以提供错误的位置信息，便于用户定位修改源程序。行号信息还是程序调试的重要依据。调试信息中主要包含的就是源程序与目标程序之间的对应关系，而建立这些行号对应关系的基础数据就是单词行号。

其次，对于编译器而言，单词可以分为三类：标识符、常量、系统单词。下面，分别讨论这三类单词的表示形式。

标识符的表示形式：（标识符的 ID，用户标识符名，单词行号）。通常，所有用户标识符都使用同一 ID。而在“单词备注”域中存储用户标识符的实际名字。

常量的表示形式：（常量的 ID，在常量表中的位置，单词行号）。通常，常量的 ID 是根据常量的类型确定的，即相同数据类型的常量的 ID 是相同的。当然，这里的数据类型并不是真正的详细数据类型，在词法分析阶段，只将常量分为整型、实型、字符串等类型，无需详细区别是哪一种详细类型（例如，`INTEGER`、`SHORTINT` 等）。常量 ID 可用于指示常量的数据类型。在词法分析时，对识别所得到的常量还必须登记入常量表，便于语法分析使用。在常量的三元组中必须记录该常量在常量表中的位置。

系统单词（即关键字、运算符、界符）的表示形式：（ID，不使用，单词行号）。通常，程序设计语言的每一个系统单词都拥有一个唯一的 ID 编号。语法分析器只需根据 ID 就可以区别是哪个系统单词了。而系统单词的三元组中“单词备注”域是不使用的。不难发现，词法分析器对于关键字、运算符、界符单词的处理是完全相同的，因此，笔者才将这三类合并为系统单词类。关于 Neo Pascal 的系统单词及其 ID 可参见 2.3.1 节。

最后，简单讨论一下单词流是如何传递给语法分析器的。这主要取决于词法分析是作为一个独立阶段还是将它设计为独立的一遍。词法分析作为一个独立阶段就是指将词法分析器作为一个函数，由语法分析器调用。每次调用词法分析器只识别一个单词，然后将单词直接传递给语法分析器处理。整个过程由语法分析器控制，在语法分析过程中，进行单词识别。词法分析作为独立的一遍就是指词法分析器将对整个源程序文件扫描一次，识别出所有的单词，然后将源程序以单词流的方式传递给语法分析器处理。这两种方式各有利弊，且各有成功的应用案例。为了使读者更清晰地理解编译器的架构，故 Neo Pascal 将词法分析作为独立的一遍扫描。Neo Pascal 词法分析器的模型示意如图 2-5 所示。

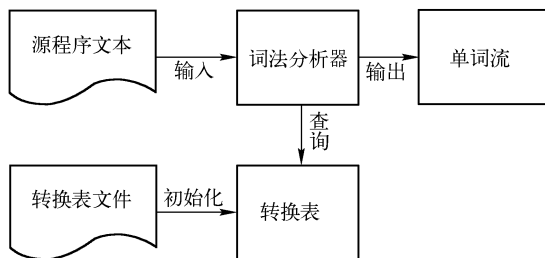


图 2-5 词法分析器模型

至此，已经详细阐明了词法分析器的设计，读者对词法分析的基本理论、设计思想与算法核心应该有所了解，这是剖析 Neo Pascal 词法分析的基础。

2.3 词法分析器的实现

2.3.1 词法定义

设计词法分析器之前，必须明确程序设计语言的词法定义。词法定义是一门程序设计语言的必要部分。同时，词法定义也直接关系到词法分析器的复杂程度，甚至并非所有的词法定义都可以实现。例如，假设 C 语言的词法定义规定标识符中可以出现“*”，则这样的词法定义是很难实现的。

表 2-5 详细描述了 Neo Pascal 的词法定义。

表 2-5 Neo Pascal 词法定义

标识符	以字母开头，后跟字母与数字的任意组合的字符串 ID: 001									
运算符	Neo Pascal 一共包含 13 个运算符，分别是：									
	ID	单词	ID	单词	ID	单词	ID	单词	ID	单词
	016	=	017	:=	018	<	019	<>	020	<=
	021	>	022	>=	023	+	024	-	025	*
	026	/	027	^	028	@				



界符	Neo Pascal 一共包含 9 个界符, 分别是:									
	ID	单词	ID	单词	ID	单词	ID	单词	ID	单词
	007	.	008	(009)	010	;	011	,
	012	:	013	..	014	[015]		
关键字	关键字是特殊的标识符, 必须符合标识符的定义。Neo Pascal 一共包含 54 个关键字, 分别是:									
	ID	单词	ID	单词	ID	单词	ID	单词	ID	单词
	040	ASM	041	AND	042	ARRAY	043	BEGIN	044	BOOLEAN
	045	BYTE	046	BREAK	047	CARDINAL	048	CASE	049	CHAR
	050	CONST	051	CONTINUE	052	DIV	053	DO	054	DOWNT0
	055	ELSE	056	END	057	FILE	058	FOR	059	FORWARD
	060	FUNCTION	061	GOTO	062	IF	063	IN	064	INTEGER
	065	LABEL	066	LONGWORD	067	MOD	068	NIL	069	NOT
	070	OF	071	OR	072	PROCEDURE	073	PROGRAM	074	REAL
	075	RECORD	076	REPEAT	077	SET	078	SHL	079	SHORTINT
	080	SHR	081	SINGLE	082	SMALLINT	083	STRING	084	THEN
	085	TO	086	TYPE	087	UNTIL	088	USES	089	VAR
	090	WITH	091	WHILE	092	WORD	093	XOR		
	常量	字符串常量, ID: 002。 例: 'I am a boy.' 整数常量, ID: 003。 例: 324、32 普通形式实型常量, ID: 004。 例: 43.53、62.034 带+/-号的科学记数形式实型常量, ID: 005。 例: 43.12E-23 不带+/-号的科学记数形式实型常量, ID: 006。 例: 41.09E12								

表中的 ID 是笔者自行编制的。为了方便管理, 尽可能将同类单词编在同一段 ID 中。在进行 ID 编号时, 应注意预留一定 ID 空间, 以便扩展。例如, 029~039 便预留给了运算符和界符。当然, 也可以直接在 093 之后扩展, 但这样扩展会出现同类单词不在同一段 ID 中的现象。另外, 笔者将 Neo Pascal 的常量分成了五种类型, 为其分配了不同的 ID。这种处理便于语法分析器只需简单判断单词 ID 就可以断定常量的类型, 而不需要详细查询常量表。例如, Neo Pascal 中的数组声明规定上、下限必须是整型常量, 当语法分析器分析到上、下限单词时, 只需判断单词 ID 是否为 003 即可。

词法分析器主要包括: 构造转换图与转换表、设计词法分析器算法。词法分析器的核心就是依据转换图识别单词。不过, 事实并非完全如此。由于某些程序设计语言的词法定义缘故, 仅仅依据程序 2-1 的算法是不足以完成词法分析的。下面, 将详细分析 Neo Pascal 的词法分析器设计与实现细节。

2.3.2 构造转换图与转换表

笔者根据 Neo Pascal 的词法定义绘制了完整的转换图, 如图 2-6 所示。作为一门完整的程序设计语言, Neo Pascal 的转换图必定是比较复杂的, 读者可以配合词法定义阅读。

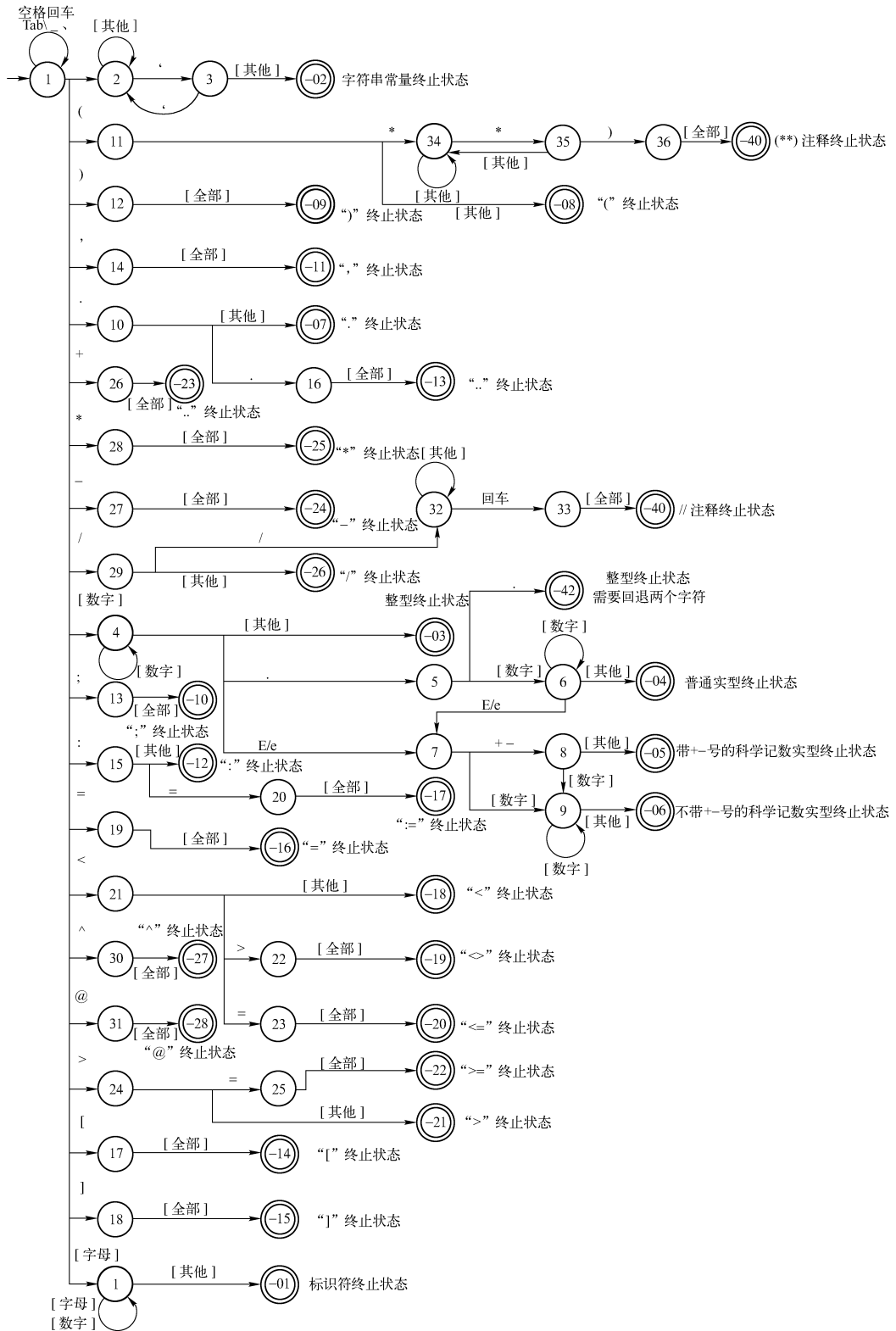


图 2-6 Neo Pascal 词法转换图



为了便于读者理解，这里对图 2-6 作几点说明：

(1) 图中 [全部]、[其他]、[字母]、[数字] 是字符集的简化表示形式。例如，[字母] 表示箭弧射出状态下输入任意字母时都转入箭弧射入状态。

(2) 终止状态都以“-”开头，以区别非终止状态。

(3) 由于 Neo Pascal 是不带编译预处理器的，所以注释也将作为单词识别，只是不将识别得到的单词输出而已。Neo Pascal 接受两种注释形式，即 (* *) 与 //，功能与 C 语言的 /* */ 与 // 完全一样。

下面通过一个特例来说明程序 2-1 算法的不足。

在标准 Pascal 语言中，一般会使用如下形式的语句声明数组：

【声明 2-1】

```
VAR i:ARRAY [1..10] OF INTEGER;
```

那么，当词法分析器识别完单词 “[” 后，返回状态 1，当前字符位置指向“1”，准备识别下一个单词。在状态 1 下读取“1”转入状态 4，当前字符位置指向第一个“.”。在状态 4 下读取“.”转入状态 5，当前字符位置后移一位指向第二个“.”。在状态 5 下读取“.”转入状态-42，当前字符位置后移一位指向“1”。由于状态-42 是终止状态，根据程序 2-1 的算法，缓冲区与当前字符位置都将回退一个字符。此时缓冲区的字符串为“1.”而当前字符位置指向第二个“.”。显然，将字符串“1.”识别为单词“1.”与“.”都不正确，而应该识别为单词“1”与“..”。实际上，主要是由于在识别“..”时需要超前搜索两个字符，而通常情况只超前搜索一个字符。因此，在处理终止状态-42 时，必须将缓冲区与当前字符位置回退两个字符，故程序 2-1 无法正确识别单词。

超前搜索几个字符与词法定义有关，有些设计不精良的语言可能需要超前搜索三个甚至四个字符才能正确识别单词。反复回溯搜索会大大降低词法分析器的执行效率，故在设计一门程序设计语言时，应尽可能做到只需超前搜索一个字符就可以完成所有单词的识别。读者不妨考虑一下，C 语言是否也存在超前搜索两个字符的情况呢？

通常，完全可以通过修改程序设计语言的词法定义避免超前搜索两个字符。但为了与标准 Pascal 兼容，Neo Pascal 仍然继承标准 Pascal 的词法定义。

从转换图导出转换表的算法是非常简单的，却比较费时费力，具体过程请参见 2.2 节，读者可在 Neo Pascal 的电子文档中找到完整的转换表。

2.3.3 相关数据结构

下面就来看看词法分析器的相关数据结构。词法分析器相关的数据结构主要包括：转换表、关键字表、常量表及其单词流。常量表暂不作详述，将在第 4 章详细分析。

1. 转换表

转换表是词法分析器的核心。通常用二维数组表示，数组元素可以根据实际情况而定，Neo Pascal 的转换表元素为整型。转换表的声明形式如下：

【声明 2-2】

```
int m_szLexTbl[50][130];
```

第1维表示状态集，第2维表示 Neo Pascal 允许输入的字符集。Neo Pascal 允许输入的字符集是指源程序文本中允许输入字符的集合。由于 Neo Pascal 语言中组成关键字、运算符、界符、标识符的字符 ASCII 码都小于或等于 127，而 ASCII 码大于 127 的字符只可能出现在字符串或注释中，因此，所有 ASCII 码大于 127 的字符在转换表中用一列表示。这样，转换表可以表示为一个 42 行 128 列的二维表。

简单估算一下转换表所占的内存空间大约为 21KB，不难发现，使用转换表实现的词法分析器的最大缺点就是所消耗的存储空间比较大。这是早期编译器不使用这一方案实现词法分析器的最重要原因。然而，对于今天的计算机而言，21KB 的存储耗费是完全可以接受的。当然，有时也可以应用一些空间压缩技术以节省存储消耗，不过，这可能是以更多的时间耗费为代价的。

2. 关键字表

关键字表用于存储 Neo Pascal 的系统关键字及其 ID。由于关键字也是符合标识符的定义的，故在词法分析时完全按照标识符的词法定义识别单词。完成标识符识别之后（即进入终止状态-01），再查找关键字表，确定该标识符是否为关键字。如果是，则返回其 ID。故关键字表一般包含两个字段：关键字名、ID。

由于每次识别完一个标识符后，都必须查找关键字表，故关键字表的查找效率至关重要。常用的查找算法比较多，例如，顺序、折半、哈希表、B-树、B+树等。这里，选择 STL 中的 map 结构来描述关键字表。map 的内核是一种红黑树结构，其查找效率近似于哈希表，属于比较高效的。关键字表的声明形式如下：

【声明 2-3】

```
map<string, int> m_KeywordTbl;
```

3. 单词流

将词法分析作为独立的一遍，则必须借助于单词流。单词的声明形式如下：

【声明 2-4】

```
struct CToken
{
    int m_iKind;           //单词的类型，即单词的 ID
    string m_szContent;    //单词备注
    LineInfo m_LineInfo;  //单词行号信息
};
```

其中 LineInfo 结构用于描述单词行号信息，单词行号信息的声明形式如下：

【声明 2-5】

```
struct LineInfo
{
    int m_iRow;           //单词所在行号
    string m_szFileName; //单词所在的源程序文件名
};
```




由于单词流的所有元素都是由单词组成的，为了使用方便，这里使用 C++ 的 `vector` 容器代替传统的数组。`vector` 容器是 C++ 的 STL 的类，单词流的声明形式如下：

【声明 2-6】

```
vector<CToken> TokenList;
```

2.3.4 源代码实现

程序 2-2 Lex.h

```
1 class Clex
2 {
3 private:
4     int m_szLexTbl[50][130];           //转换表
5     string m_szSource;                 //源代码字符串
6     string m_szFileName;               //源程序文件名
7     map<string,int> m_KeywordTbl;      //关键字表
8     string m_szBuffer;                 //单词缓存区
9     int m_iRow;                         //当前行号
10    int m_iNonTerminal;                 //当前非终结状态编号
11    int m_iPos;                          //源代码当前读取位置
12    vector<CToken> *m_pTokenList;       //单词流
13    bool Process(int iTag);              //词法分析处理
14    void EmitToken(const int iKind,const string szContent,const int iRow); //记录单词函数
15    void EmitToken(const int iKind,const int iContent,const int iRow); //记录单词函数
16    bool SearchKeyword(string szKeyWord,int &iPosition); //查询关键字表函数
17    void BufferBack();                   //单词缓存区回退函数
18 public:
19     CLex();                             //默认构造函数
20     ~CLex();                             //默认析构函数
21     bool GenToken(vector<CToken> *pTokenList); //生成单词流函数
22     void SetKeywords(string szStr);       //设置关键字表函数
23     void SetLexTbl(string szStr);        //设置转换表函数
24     void SetFileName(string szStr);      //设置源程序文件名函数
25 };
```

在一些现代编译器设计中，有时，词法分析器不仅需要记录行号信息，还有可能需要记录列号信息，具体情况视调试器而定。当然，记录列号信息要比行号信息复杂一些，尤其是在有预处理阶段的编译器中。

程序 2-3 Lex.cpp

```
1 bool CLex::GenToken(vector<CToken> *pTokenList)
2 {
3     bool bTag=true;
4     m_iPos=0;
```

```

5     m_pTokenList=pTokenList;
6     m_szSource=FileRead(m_szFileName.c_str());
7     m_szSource.push_back(' ');
8     m_iNonTerminal=0;
9     m_szBuffer="";
10    m_iRow=1;
11    int TmpPos=0;
12    m_pTokenList->clear();
13    while (m_iPos<m_szSource.length() && bTag)
14    {
15        if (m_szSource[m_iPos]=='\n'&& TmpPos!=m_iPos)
16        {
17            m_iRow++;
18            TmpPos=m_iPos;
19        }
20        m_szBuffer.push_back(m_szSource.at(m_iPos));
21        bTag=Process(m_szLexTbl[m_iNonTerminal][m_szSource[m_iPos]<128?m_szSource[m_iPos]:128]);
22        if (!bTag)
23        {
24            EmitError(m_szFileName+"("+itos(m_iRow)+"):词法分析错误, 请检查单词");
25            return false;
26        }
27        m_iPos++;
28    }
29    return bTag;
30 }

```

第 3 行：用于判定单词是否正确的标志，初始置为 true。

第 4 行：源程序文本的当前读取位置，初始置为 0。

第 6 行：读取 m_szFileName 指定的文件，将文件以字符串形式返回置入 m_szSource。

第 7 行：在 m_szSource 尾部添加一个空格，以便处理最后一个单词时，也能运用超前搜索识别。

第 8 行：转换表的开始状态置入 m_iNonTerminal，即当前状态。

第 9 行：单词缓存区清空。

第 10 行：源程序当前行号为 1，表示正在处理第 1 行。

第 11 行：单词流清空。

第 13 行：未读到源程序尾并且无出错单词，则继续识别单词。

第 15 行：如果当前读入字符是换行符，则将源程序当前行号累加 1。

第 20 行：将当前读字符加入单词缓存区。

第 21 行：查询当前状态下读取当前字符时的转换状态，当前字符 ASCII 码大于 128 时，取 128 即可。

第 27 行：当前读取位置后移一个字符。



程序 2-4 Lex.cpp

```
1  bool CLex::Process(int iTag)
2  {
3      int iTmp=0;
4      if (iTag== -99)
5          return false;
6      if (iTag<0)
7          {
8              BufferBack();
9              m_szBuffer=trim(m_szBuffer);
10             if (iTag== -1)
11                 {
12                     m_szBuffer=UpperCase(m_szBuffer);
13                     if (SearchKeywords(m_szBuffer,iTmp)==true)
14                         EmitToken(iTmp+40,NULL,m_iRow);
15                     else
16                         {
17                             if (m_szBuffer.compare("TRUE")==0||m_szBuffer.compare("FALSE")==0)
18                                 EmitToken(3,SymbolTbl.RecConstTbl(m_szBuffer,7),m_iRow);
19                             Else
20                                 EmitToken(1,m_szBuffer,m_iRow);
21                         }
22                 }
23             if (iTag>= -6 && iTag<= -2)
24                 EmitToken(-iTag,SymbolTbl.RecConstTbl(m_szBuffer,-iTag),m_iRow);
25             if (iTag>= -15 && iTag<= -7)
26                 EmitToken(-iTag,NULL,m_iRow);
27             if (iTag>= -28 && iTag<= -16)
28                 EmitToken(-iTag,NULL,m_iRow);
29             if (iTag== -42)
30                 {
31                     BufferBack();
32                     m_szBuffer=trim(m_szBuffer);
33                     EmitToken(3,SymbolTbl.RecConstTbl(m_szBuffer,3),m_iRow);
34                 }
35             m_szBuffer="";
36             m_iNonTerminal=0;
37         }
38     else
39         m_iNonTerminal=iTag;
40     return true;
41 }
```

第 3 行：临时变量初始化为 0。

第 4 行：下一个转入状态等于 -99，表示当前状态下不允许读取当前字符，即单词识别

出错。

第 6 行：下一个转入状态小于 0，表示当前状态下读取当前字符时进入终止状态，即单词识别完成。

第 8 行：由于超前搜索识别，单词缓存区、当前读取位置回退一个字符。

第 9 行：滤去单词缓存区的左右空格。

第 10 行：终止状态为-1，表示当前单词是标识符。

第 12 行：注意，这里将字符串转成大写的原因是 Pascal 语言标识符是大小写不敏感的。

第 13 行：判断当前单词是否为关键字。

第 14 行：将单词标识为关键字，并加入单词流。

第 17 行：判断当前单词是否为 TRUE/FALSE（布尔常量）。

第 18 行：将单词标识为常量，并加入单词流，登记常量表。

第 20 行：将单词标识为标识符，并加入单词流。

第 23 行：判断当前单词是否为常量。

第 24 行：将单词标识为常量，并加入单词流，登记常量表。

第 25 行：判断当前分析的单词是否为界符。

第 26 行：将单词标识为界符，并加入单词流。

第 27 行：判断当前单词是否为运算符。

第 28 行：将单词标识为运算符，并加入单词流。

第 29 行：判断终止状态是否为-42，在这种情况下，需要处理两次回退。

第 31 行：单词缓存区、当前读取位置回退一个字符。

第 32 行：滤去单词缓存区的左右空格。

第 33 行：将单词标识为整型常量，并加入单词流，登记常量表。

第 35 行：将单词缓存区清空，准备识别下一个单词。

第 36 行：将当前状态置为转换表的开始状态。

第 40 行：下一个转入状态大于或等于 0，表示当前状态下读取当前字符时进入非终止状态，即单词未识别完成，将当前状态置为 iTag 指示的下一个状态。

程序 2-5 Lex.cpp

```
1  bool CLex::SearchKeyword(string szKeyWord,int &iPosition)
2  {
3      map<string,int>::iterator It=m_KeywordTbl.find(szKeyWord);
4      if (It!=m_KeywordTbl.end())
5          {
6              iPosition=It->second;
7              return true;
8          }
9      return false;
10 }
```

第 3 行：在关键字表中查找字符串。



第 4 行：字符串是关键字。

第 6 行：返回关键字的 ID。

程序 2-6 Lex.cpp

```
1 void CLex::BufferBack()
2 {
3     m_szBuffer.erase(m_szBuffer.end()-1);
4     m_iPos--;
5 }
```

第 3 行：将单词缓存区的最末一个字符删除，即实现退回一个字符的功能。

第 4 行：将当前位置回退一个字符。

程序 2-7 Lex.cpp

```
1 void CLex::EmitToken(int iKind,string sContent,int iRow)
2 {
3     m_pTokenList->push_back(CToken(iKind,sContent,iRow,m_szFileName));
4 }
5 void CLex::EmitToken(const int iKind,const int iContent,const int iRow)
6 {
7     string szTmp;
8     char cBuffer[5];
9     itoa(iContent,cBuffer,10);
10    szTmp=cBuffer;
11    EmitToken(iKind,szTmp,iRow);
12 }
```

程序 2-6 和 2-7 是用于提交单词的，即把识别得到的单词加入单词流中。

2.4 深入学习

词法分析理论与技术的应用是现代计算机科学的一个相当成功的故事。无论是手工构造，还是自动生成词法分析器都是完全可行的。本书以介绍手工构造为主，不过，实践证明，依据一定的规则，自动生成词法分析器也是一个不错的选择。其中，最经典的实例就是 UNIX 平台上的 Lex，它是基于有限自动机理论实现的。Lex 根据输入的正则表达式，生成相应的词法分析器。当然，手工编码实现词法分析器的方法也是可行的，经典实例是不胜枚举的。两者并不存在绝对的优劣之分，更多关注的是是否有利于源语言的实现。这里，笔者推荐一个词法分析器构造程序——Flex，它的基本思想与 Lex 是一脉相承的。Flex 诞生于 1987 年，经常与 Bison 一起使用，生成词法、语法分析器。读者可以访问<http://www.gnu.org/software/flex/>，以获得更多的相关信息。当然，正则表达式作为形式语言和自动机理论的早期雏型，为搜索与扫描技术的发展提供了源动力，广泛应用于文本编辑、网络检索、编译器、信息安全等领域。

自动机理论与形式语言是计算机科学中两个非常重要的理论研究方向，许多计算理论书

籍都会涉及这两个话题。对于有志于学习与研究自动机与形式语言的读者而言,《自动机理论、语言与计算导论》是该领域一本较新的著作,值得深入阅读。

1、编译器工程 Keith D. Cooper, Linda Torczon 机械工业出版社

说明:这是一本以介绍编译器实践工程为主的教材,堪称“龙书”之后的另一本经典著作。

2、Lex – A lexical Analyzer Generator M. E. Lesk, E. Schmidt

说明:Lex 是一个基于有限自动机理论实现的词法分析器的生成器,也是早期自动化生成技术的一个经典范例。

3、ISO/IEC 9945

说明:这是一个正则表达式的标准,它是对 UNIX 正则表达式规范的扩充及完善。

4、自动机理论、语言与计算导论(原书第2版) John E. Hopcroft, Rajeew Motwani 机械工业出版社

说明:这本书详细阐述了自动机理论与形式语言的相关问题,同时还讨论了不可判定性等疑难话题。

2.5 实践与思考

1. Pascal 语言是大小写不敏感的,而 C 语言是大小写敏感的。从词法分析器的实现角度而言,试问两者的差异是什么?

2. 一个 STL 的声明语句如下:

```
map<int, vector<int>> tmp;
```

试问词法分析器应该如何处理其中的“>>”呢?根据极大搜索原则,应该将其识别为“>>”(右移运算符),但在此显然是不适用的。在《C++ Primer》中,明确指出程序员必须在两个“>”之间输入至少一个分隔符,否则视为非法的。不过,在 VC++ 中,对此并没有要求。(提示:考虑回溯识别)

3. 在经典编译器(如 lcc、GCC 等)中,通常,并不是借助关键字表来实现关键字判断的,而是直接在词法分析器的转换图中加以描述。实际上,这种实现虽然会使得转换图复杂一些,但是它却比本章介绍的实现方法更优。试问这种设计的优点是什么?

4. 在 C 语言中,单词的长度是有限制的。C89 规定单词最大长度不超过 509 个字符,而 C99 的最大长度是 4096。然而,Pascal 并没有明确的规定,因此,Neo Pascal 没有限制。不过,有些商用 Pascal 编译器为了提高词法分析的效率,对此有一定限制。如果 Neo Pascal 限制单词的最大长度为 4096,试问应该如何修改程序以提高单词识别的效率呢?

2.6 大师风采——Dennis M. Ritchie

Dennis M. Ritchie: 美国计算机科学家,UNIX、C 语言创始人之一。1941 年 9 月 9 日出生于纽约。在哈佛大学获得物理学学士学位后,又于 1968 年获得应用数学博士学位。1967 年,Ritchie 加入著名的贝尔实验室,参与了 Multics 项目。正是 Multics 项目,为 UNIX 的产生打下了基础。

1969 年,Dennis M. Ritchie 与 Ken Thompson、Douglas McIlroy 设计与实现了著名的 UNIX 操作系统。最初 UNIX 是用汇编语言编写,而其中一些应用是由 B 语言和汇编语言混合实现的。不过,由于 B 语言在系统编程方面的缺陷,因此,Dennis M. Ritchie 与



编译器设计之路

Ken Thompson 对其进行了完善，并与 1971 年共同设计实现了 C 语言。1973 年，他们用 C 语言重写了 UNIX。在计算机科学史上，UNIX 与 C 语言是具有里程碑意义的。1977 年，Dennis M. Ritchie 发表了不依赖于具体机器系统的 C 编译器论文《可移植的 C 语言编译程序》。1978 年，Ritchie 与 B. W. Kernighan 合作撰写了著名的《The C Programming Language》一书，这是 ANSI C 标准的基础。

Dennis M. Ritchie 与 Ken Thompson 也因为 UNIX 及 C 语言的成功而获得了 1983 年图灵奖。

第3章 语法分析

The effort of using machines to mimic the human mind has always struck me as rather silly. I would rather use them to mimic something better.

——Edsger Wybe Dijkstra

3.1 程序设计语言的语法描述

3.1.1 上下文无关文法

一种语言通常由语法、语义、语用三个部分组成。其中，语法是本章关注的重点，语义与语用将在后续章节中阐述。语言的语法指的是构成语言句子的各个符号之间的组合规律。同样，程序设计语言的语法就是描述构成各种语句的单词的组合规律，它是判断程序合法性的标准之一，但不是唯一标准。读者必须注意，符合程序设计语言语法的程序未必是合法的，但不符合语法的程序必定是非合法的。这主要是由于程序既要符合语言的语法，又要符合语言的语义。

程序设计语言的语法体系与该语言是否能得到普及发展有着密切的联系。同时，也是评价一门语言的主要标准。通常，一门程序设计语言是否能得到普及发展主要取决于这几个方面：程序设计思想、应用领域、复杂程度、可移植性、语言实现、经济及其他非语言因素等。

(1) 程序设计思想。实践证明，某种激进的程序设计思想往往与实现这种思想的程序设计语言密不可分。当然，那些实现先进设计思想的程序设计语言也会随着该设计思想的流行而获得广泛应用。例如，Niklaus Wirth 提出结构化程序设计思想，同时也实现了 Pascal 语言。Pascal 语言是其设计思想的一种体现。可以毫不夸张地说，完全是 IF、WHILE、FOR 等语法结构的出现成就了 Pascal、C 语言风靡几十年的神话。

(2) 应用领域。程序设计语言在其定义之初就应该明确其应用领域。不同的应用领域对于程序设计语言的功能要求是不尽相同的。当然，应用领域也影响了该程序设计语言的普及程度。例如，C 语言的应用领域是系统软件设计，而在数据库相关设计中就很少使用 C 语言。HTML 语言的应用领域是 Web 设计，它就不具备传统结构化程序设计语言的 IF、WHILE 等语句。脱离具体的应用领域评价程序设计语言是不科学的。不过，优秀的程序设计语言往往能在相关领域内脱颖而出。

(3) 复杂程度。与其他方面的因素相比，程序设计语言的复杂程度似乎显得更为重要。使用者总是偏爱那些简单易学的语言。简单易学的程序设计语言可以一定程度上弥补功能或其他方面的不足，BASIC 语言的成功就是最好的例证。无论是程序设计思想还是功能实现，BASIC 语言都确实存在一定不足之处，但是它的简单易学却足以弥补这一切。实际上，现存



的大多数语言都是由于其简单易学而得以从当年的“乱世”中传承至今。实践证明，在一门功能强大的复杂语言和一门功能相对稍弱的简单语言之间，人们往往更乐意选择后者。当然，这种弥补是有一个度的。通常，功能强大与简单易学是矛盾的。以失去功能为代价，一味地追求语言的简单易学并不可取。语言设计者必须准确把握这个度，否则可能适得其反。

(4) 可移植性。在当今程序设计语言领域，关于可移植性的争论是比较激烈的。在软件复用被尤为重视之际，一门语言是否能跨平台使用直接决定了其是否能普及发展。在这个问题上，C++与Java之争的结果似乎已经证明了一切。

(5) 语言实现。Fortran 语言的成功可能就是归功于其优秀的编译器实现。早期 Fortran 编译器的设计宗旨就是生成特别高效的代码。Fortran 语言的开发者为此投入了大量的时间与金钱，虽然以牺牲一定的语言功能（例如，Fortran 不支持递归）为代价，但是他们最终实现了这一目标。正是出众的效率才使得功能、设计思想都不是特别优秀的 Fortran 语言能传承至今。当然，语言实现不局限于编译器，还包括一些环境支持工具（如 IDE、项目管理工具等）。

(6) 经济及其他非语言因素。经济及其他非语言因素也是决定一门程序设计语言能否普及的重要因素。例如，语言支持、惯性发展等。经济因素是比较重要的，实践证明，免费的产品总会比收费的产品更容易得到使用者的青睐。编译器、操作系统等都是如此，Pascal、C、Java 等都是由于免费才让更多的程序员、专家认识了它们。Linux 能与操作系统霸主 Windows 一争高下，一个重要原因也是免费。另外，强有力的语言支持是保证语言发展的一个因素。程序设计语言的设计与实现是一项费时耗力的长期工程，如果没有强有力的后盾支持（人力、财力），可能是无法完成的。例如，COBOL 和 PL/I 的生存很大程度上与 IBM 的支持有关。Ada 语言的成功就是因为它出生在美国国防部。今天，评论 C#、Java 是否成功可能为时过早，但是 C#、Java 的发展与微软、Sun 的支持是分不开的。再者，语言的惯性发展也是人们选择语言时考虑的因素之一。先入为主的思想广泛存在，人们通常更喜欢选择在某些现存语言基础上扩展而得的语言，而不喜欢选择一些全新定义的语言。尤其在今天几大语言体系初现端倪之际，人们更希望选择自己熟悉的语言及其扩展形式。

以上各个因素可能贯穿了整个程序设计语言及其编译器设计的过程。其中，前面三个因素与程序设计语言的语法有着密切的联系。因此，程序设计语言的语法定义是至关重要的，它是评价一门程序设计语言的重要因素之一。程序设计语言的语法定义通常包括两个过程：建模、描述。

建模就是构造一门程序设计语言的语法体系，这个过程并非想象中的那样简单，它涉及许多程序设计语言的相关理论。笔者举个简单的例子：C 语言只定义了“与”、“或”、“非”、“异或”四个逻辑运算符，然而，这四个逻辑运算符的不同组合却可以描述任何复杂逻辑关系，但是，无论缺少哪一个逻辑运算符都无法实现这一描述功能。这使得 C 语言能借助较少的语言结构实现更多的功能，保证语言结构尽可能不重复，这种特性被称为程序设计语言的正交性。然而，在建模过程中，保证程序设计语言的正交性有时并不是非常容易的，需要程序设计语言相关理论的支持。由于 Neo Pascal 语言只是 Pascal 语言的扩展，所以并不存在完整的建模过程。程序设计语言的定义与设计也不是编译技术讨论的范畴，故不再赘述。

描述即语法的形式描述。其重点就是讨论如何描述程序设计语言语法模型。实际上，语

言语法的描述是一门独立的学科，即形式语言学。形式语言是一种元语言的形式，是一套形式化描述语言（主要指的就是语言的语法）的方法。读者只要将形式语言理解为一种描述语言的语言即可。关于语言语义的描述并不是形式语言的讨论范畴，将在第 4~6 章中说明。形式语言在自然语言研究中起步，却在计算机科学中得到了应用，如编译技术、计算复杂性、通信技术、图像处理、人工智能等领域。其中，最主要被应用于编译技术中。编译技术是一门与语言有关的学科，所以形式语言自然就成为编译技术讨论的范畴了。在编译技术中，形式语言主要用于描述程序设计语言的语法。下面，就谈谈形式语言与程序设计语言语法之间的联系。

虽然人们每时每刻都离不开语言，但是，在日常生活或工作中，语言的语法往往为人们所忽视。读者可以想象一下，在学习汉语时，是不是从汉语语法开始的呢？显然，答案是肯定的。语言学家研究表明，人们学习与掌握一门新语言的最主要途径就是实践经验。也就是说，人们是通过大量实践运用一门语言，以达到熟能生巧的境界。学习程序设计语言同样如此，也是通过大量编程实践最终才能掌握一门程序设计语言的。如果没有真正接触到语言的形式描述，即使能熟练应用某些程序设计语言，那也只是处于对语言的感性认识阶段。包括大多数程序设计书籍在内的编程方面的资料都是使用自然语言形式描述程序设计语言语法及其语义的。正如读者所知的，自然语言是可能存在不同程度的二义性的。显然，这种模糊的、不确定的方式是无法精确定义一门程序设计语言的。正因如此，在 20 世纪 50 年代，美国的语言学家 Noam Chomsky 提出了“转换—生成语法”的概念，深入阐述了语法结构的相关理论，这就是“形式语言学”原型，它是具有里程碑意义的。

编译器设计是一门与程序设计语言有着密切联系的学科，因此，仅仅从感性的角度认识一门程序设计语言是不够的。当然，必须设计一种准确无误地描述程序设计语言的语法结构，这就是将形式语言应用于编译技术的初衷。通常，将这种严谨、简洁、易读的形式规则描述的语言结构模型称为文法（grammar）。

最著名的文法描述形式是由 Backus 在定义 Algol 60 语言时提出的 Backus-Naur 范式（Backus-Naur Form，简称为“BNF”）及其扩展形式（简称为“EBNF”）。实际上，EBNF 与 BNF 还是存在细微差别的，而且 EBNF 应用比 BNF 更为广泛。不过，由于 BNF 太著名，人们并不严格区别 EBNF 与 BNF，统称为 BNF。有些书籍甚至已不再严格区别 BNF 与文法的概念，本书后续提及文法指的就是文法的 BNF 形式。当然，除了 BNF 之外，还有一些其他的文法描述形式。其中，较为常见的就是语法图，第 1 章中描述 Pascal 语法的图形工具就是语法图。BNF、语法图都只是文法的一种描述形式而已，但与自然语言描述方式相比，它们的优点就是不存在二义性。然而，BNF 之所以更为著名，就是由于它能以一种简洁、灵活的方式描述语言的语法。下面，笔者给出一个 BNF 的实例。

例 3-1 自然语言文法实例。

【文法 3-1】

句子	→	主语 谓语 宾语部分
主语	→	你
主语	→	我
谓语	→	是
宾语部分	→	定语 宾语



宾语部分	→	宾语
定语	→	可爱的
定语	→	美丽的
定语	→	帅气的
宾语	→	男孩
宾语	→	女孩

文法 3-1 是一个简化的自然语言文法。通常，将其中的每一行称为一个产生式，而一个完整的文法可以由一个或多个产生式组成。每个产生式中有且只有一个箭头（有时，亦可记作“::”），表示“定义为”。箭头将一个产生式分成左右两个部分，而整个产生式即表示将其左部定义为其右部。读者不难发现，产生式的右部可以由一个或多个符号（或称为项）组成。其中，有些符号是整个文法中某些产生式的左部（为了便于讲解，笔者将这些符号使用斜体表示），有些符号则不是。在 BNF 中，将斜体符号称为非终结符，也就是说，它可以用其他符号串来定义。实际上，非终结符就是一个语法变量，是符号的有序集合。而其他非斜体符号则称为终结符，它是语言的最基本符号，即不可再分割的原子符号。

文法 3-1 描述了一个简化的自然语言模型，也就是说文法 3-1 完全可以验证某个句子是否属于该语言模型。下面，尝试验证“你是可爱的男孩”是否为合法的句子。实际上，如果可以运用上述产生式推导出这个句子（就是从“句子”出发，反复把上述规则中“→”左边的符号替换成右边的符号的过程。关于推导的详细内容可参见 3.1.2 节。），则表明这个句子符合该文法的描述，即为合法的句子，否则为不合法的句子。事实上，“你是可爱的男孩”是一个合法的句子，推导过程如下：

句子=>主语 谓语 宾语部分
=>你 谓语 宾语部分
=>你 是 宾语部分
=>你 是 定语 宾语
=>你 是 可爱的 宾语
=>你 是 可爱的 男孩

从“句子”这个非终结符出发（通常，称为文法开始符），经过了多次推导，最终得到了“你是可爱的男孩”这个句子，表明它是合法的句子。回头再来看看到底什么是推导？一次推导就是将非终结符替换为合适的产生式右部符号串的过程，可以用“=>”表示推导操作。从文法开始符出发，可以通过推导得到的句子集合就是这个文法所描述的语言。例如，“我是帅气的男孩”、“你是美丽的女孩”都是文法 3-1 的合法句子，而“我是男孩帅气的”、“我帅气的男孩是”都不是文法 3-1 的合法句子。读者不难发现，文法描述的语言都是终结符的集合，不包含任何非终结符。对于任何包含非终结符的符号串都只是推导的一个步骤，并不是文法描述的句子。在形式语言学中，将包含非终结符的符号串称为句型，将不包含非终结符的符号串称为句子。

值得注意的是，文法 3-1 是一个简化的自然语言文法，因此，它所描述的语言并不是汉语的全集。实际上，完整的自然语言文法是非常复杂的。其形式化描述是自然语言理解领域的重要研究课题，这里不作讨论。程序设计语言的语法比自然语言简单得多，使用文法描述程序设计语言是完全可行的。

形式语言理论的开创者 Chomsky 按文法描述能力将文法分为四种类型，由高到低分别为：0 型（短语文法）、1 型（上下文相关文法）、2 型（上下文无关文法）、3 型（正规文法）。实际上，大多数程序设计语言的文法都是上下文无关文法，文法 3-1 也是一个上下文无关文法。因此，上下文无关文法是编译技术讨论的重点。至于其他类型的文法，读者可参考相关书籍，这里不再赘述。

上下文无关文法（context-free grammar）由终结符、非终结符、文法开始符和产生式组成。

(1) 终结符 (terminal): 语言的最基本符号，即不可再分割的原子符号。在程序设计语言中，终结符必定是词法分析所得到的单词。例如，关键字 if、then 等都是终结符。

(2) 非终结符 (nonterminal): 语法变量，它可以定义为一个符号串（由终结符和非终结符组成的符号串）。例如，文法 3-1 的“宾语”、“主语”等都是非终结符。

(3) 文法开始符 (start symbol): 在一个完整的文法中，有且仅有一个非终结符可以被指定为文法开始符。

(4) 产生式 (production): 是一种定义语法规则的形式。产生式的形式：非终结符→符号串。

通常，有些书把上下文无关文法表示为一个四元组：(V_T, V_n, S, P)，其中 V_T 为终结符集合，V_n 为非终结符集合，S (S ∈ V_n) 为文法开始符，P 为产生式集合。

为了便于描述，除特殊说明外，本书的文法将采用如下约定：

非终结符用斜体字表示。

终结符用正体字表示。

若无特殊说明，文法的第一个产生式的左部非终结符就是文法的开始符。

可以使用小写希腊字母表示文法符号串。

如果存在如下文法：

$$A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_k$$

则可以简化写成如下形式：

$$A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_k$$

我们将 α_1 、 α_2 、……、 α_k 称为 A 的候选式。

例 3-2 使用上述简写约定，文法 3-1 可以简写为：

【文法 3-2】

句子	→ 主语 谓语 宾语部分
主语	→ 你 我
谓语	→ 是
宾语部分	→ 定语 宾语 宾语
定语	→ 可爱的 美丽的 帅气的
宾语	→ 男孩 女孩

语言的文法是研究程序设计语言的基础，也是编译器前端的重要理论基础之一。读者应该熟悉文法描述形式，这对于学习编译器与程序设计语言设计是至关重要的。编译器设计者



可以不关心程序设计语言的建模过程，但必须关心建模的结果，即语言模型的形式化描述。下面，笔者借助一个实例来讨论如何运用文法描述程序设计语言的算术表达式。

例 3-3 算术表达式的文法。

【文法 3-3】

```

E    →  E op1 T | T
T    →  T op2 F | F
F    →  ( E ) | -F | I
I    →  var | const
op1  →  + | -
op2  →  * | /
    
```

注意：var 表示变量，const 表示常量，它们都是终结符。

在初学一门程序设计语言时，最复杂的语法机制可能就是表达式了。就 C 语言而言，34 种运算符及其 15 个优先级分类可能足以让初学者琢磨许久。表达式不但文法复杂（优先级）、运算符众多，而且还涉及类型系统等繁复的语义处理。虽然文法 3-3 是一个简化的算术表达式文法，但是它揭示了算术表达式乃至表达式的核心。文法 3-3 中包含了括号、单目运算符、双目运算符，同时，还涉及运算符的优先级。独立设计程序设计语言文法的机会未必很多，但理解文法所描述的语法规则是非常必要的。运用文法 3-3 验证表达式“3*5+(1-i)”合法性的详细推导过程如下：

```

E    => E op1 T           => T op1 T
     => T op2 F op1 T     => F op2 F op1 T
     => I op2 F op1 T     => 3 op2 F op1 T
     => 3 * F op1 T       => 3 * I op1 T
     => 3 * 5 op1 T       => 3 * 5 + T
     => 3 * 5 + F         => 3 * 5 + ( E )
     => 3 * 5 + ( E op1 T ) => 3 * 5 + ( T op1 T )
     => 3 * 5 + ( F op1 T ) => 3 * 5 + ( I op1 T )
     => 3 * 5 + ( 1 op1 T ) => 3 * 5 + ( 1 - T )
     => 3 * 5 + ( 1 - F )   => 3 * 5 + ( 1 - I )
     => 3 * 5 + ( 1 - i )
    
```

从文法开始符 E 经过 21 步推导可以得到表达式 3*5+(1-i)，故此表达式是合法的。

3.1.2 推导

前面通过实例简单介绍了推导的大致过程，但并没有明确阐述推导的定义。本小节将从推导的定义出发，深入研究推导及其与编译器设计的联系。推导是用于描述文法定义语言的过程，其核心思想是把产生式的左部非终结符替换为右部的符号串。推导的方法是程序设计语言语法分析的核心方法之一。下面，给出推导的公理化定义：

如 $\alpha \rightarrow \beta$ 是文法 $G=(V_t, V_n, S, P)$ 的规则（或说是 P 中的一条产生式）， γ 和 δ 是 $(V_n \cup V_t)$ 中的任意符号，若有符号串 v, w 满足：

$$v = \gamma \alpha \delta, w = \gamma \beta \delta$$

则说 v (应用规则 $\alpha \rightarrow \beta$) 直接产生 w , 或说 w 是 v 的直接推导 (direct derivation), 记作 $v \Rightarrow w$ 。

如果 $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$, 则称其为 α_1 到 α_n 的推导 (derivation)。通常, 可以用 “ \Rightarrow ” 表示 “一步推导”, 用 “ $\stackrel{*}{\Rightarrow}$ ” 表示 “零步或多步推导”, 用 “ $\pm \Rightarrow$ ” 表示 “一步或多步推导”。在不关心推导细节的情况下, 可以将例 3-3 的推导过程简写为 $E \stackrel{*}{\Rightarrow} 3 * 5 + (1 - i)$ 。

对于文法开始符号为 S 的文法 G , 如果 $S \stackrel{*}{\Rightarrow} \alpha$, 则称 α 为 G 的句型 (sentential form), 其中 α 可能包含非终结符。不含非终结符的句型称为句子 (sentence)。

关于推导的过程与实例, 笔者认为没有必要再次阐述了, 请读者参见例 3-3。推导能运用文法验证用户输入的程序, 那么, 如何设计一个算法能让计算机自动完成推导验证程序的工作呢? 这正是本书研究推导的目的。目前, 手工推导验证程序完全是凭借个人经验完成的, 这种随机性较大的工作流程是很难通过算法实现的。因此, 有必要深入分析手工推导的过程, 得到其工作模型, 才能通过算法最终实现。读者仔细分析手工推导过程, 不难发现每一步推导通常都存在两个选择:

- (1) 选择哪个非终结符被替换。
- (2) 选择哪个候选式替换该非终结符。

以例 3-3 第二步推导为例, 第二步推导过程是从符号串 “E op1 T” 推导得到符号串 “T op1 T”。在符号串 “E op1 T” 中存在三个非终结符, 即 E、op1、T。由于每一步推导只能替换一个非终结符, 故必须只选择替换其中一个非终结符。实际上, 无论选择替换哪个非终结符, 其最终推导得到的句子都是一样的。笔者在此不作理论证明, 只通过实例说明。读者可以参考例 3-4 的推导过程, 例 3-3 的第二步选择对非终结符 E 进行推导, 而例 3-4 在第二步推导中选择替换非终结符 T。两个推导过程完全不同, 但是其最终推导得到的句子是相同的。当然, 读者也可以作其他尝试, 可以发现最终结果是始终不变的。实践证明, 选择哪个非终结符被替换并不影响最终的推导结果。那么, 例 3-3 中选择对非终结符 E 进行推导, 非终结符 E 的候选式有两个, 即 E op1 T 和 T。此时, 面临选择哪个候选式替换非终结符 E 的问题。注意, 在推导过程中, 通常某一步推导所能选择的候选式中往往只有一个可以正确推导得到给定句子, 其余的都是错误的。当然, 某些文法可能出现某一步推导所能选择的候选式中存在两个或以上的候选式能够正确推导得到给定句子的情况, 必须通过变换文法的方法消除这类情况。如果某些文法经过变换后仍存在某一步推导能选择超过一个的正确候选式, 则说明文法本身存在歧义, 通常称为二义性。笔者将在 3.1.3 节中讨论文法的二义性问题, 这里先不作深究。总之, 读者必须明确一点, 就是对于文法的任何一个合法句子而言, 文法必须保证每一步推导过程中有且仅有一个候选式能够最终正确推导得到该句子。因此, 选择哪个候选式替换非终结符最终得到的结果是不同的。要想正确推导得到句子, 在每一步推导过程中, 都必须选择那个唯一正确的候选式进行深入推导, 否则, 无法推导得到所需的句子。由于候选式的选择可能直接决定了是否能正确推导出给定句子, 那么, 如果选择了不合适的候选式可能会使推导过程回溯。例 3-3 中选择了 E op1 T 候选式替换非终结符 E, 这是唯一正确的选择。读者可以尝试选择候选式 T 替换非终结符 E, 经过 n 步推导仍然无法推导出给定表达式。

推导过程可以判定给定句子是否符合文法。那么, 模拟推导过程实现语法分析器的想法也就形成了。前面, 已经分析了推导的过程, 也发现了推导过程中的核心难点, 即两个选



择。构造基于推导的语法分析器就必须解决两个选择的问题，即非终结符的选择与候选式的选择。由于非终结符的选择并不会影响推导的结果，所以选择非终结符的问题就比较容易了。可以约定每一步推导都选择替换最左边或者最右边的非终结符，通常，这是最简单的选择方案。习惯上，将每次都选择替换最左边非终结符的推导过程称为最左推导（leftmost derivation），例 3-3 的推导过程就是最左推导。而将每次都选择替换最右边非终结符的推导过程称为最右推导（rightmost derivation）。为了对句子进行确定性分析，编译技术只考虑最左推导、最右推导。至于候选式的选择就比较复杂了，笔者将在 3.2 节中详细讲述。

例 3-4 最右推导实例。使用文法 3-3 验证表达式 “3*5+(1-i)” 是否合法？

$ \begin{aligned} E & \Rightarrow E \text{ op1 } T \\ & \Rightarrow E \text{ op1 } (E) \\ & \Rightarrow E \text{ op1 } (E \text{ op1 } F) \\ & \Rightarrow E \text{ op1 } (E \text{ op1 } I) \\ & \Rightarrow E \text{ op1 } (T-i) \\ & \Rightarrow E \text{ op1 } (I-i) \\ & \Rightarrow E+(1-i) \\ & \Rightarrow T \text{ op2 } F+(1-i) \\ & \Rightarrow T \text{ op2 } 5+(1-i) \\ & \Rightarrow F*5+(1-i) \\ & \Rightarrow 3*5+(1-i) \end{aligned} $	$ \begin{aligned} & \Rightarrow E \text{ op1 } F \\ & \Rightarrow E \text{ op1 } (E \text{ op1 } T) \\ & \Rightarrow E \text{ op1 } (E \text{ op1 } I) \\ & \Rightarrow E \text{ op1 } (E-i) \\ & \Rightarrow E \text{ op1 } (F-i) \\ & \Rightarrow E \text{ op1 } (1-i) \\ & \Rightarrow T+(1-i) \\ & \Rightarrow T \text{ op2 } I+(1-i) \\ & \Rightarrow T*5+(1-i) \\ & \Rightarrow I*5+(1-i) \end{aligned} $
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

3.1.3 语法树

通常，推导过程是研究语法分析过程的重要依据。为了便于描述，可以使用一些图形化方式描述推导过程。其中，语法树（syntax tree）是最重要的方式之一。它是推导的图形描述形式，有助于理解一个句子语法结构的层次，简称语法层次。语法层次这个概念在有些书籍上被替换成了优先级，确实如此，在表达式文法中的语法层次就是运算符的优先级。但是，由于作为一门实际程序设计语言的文法不仅包含了表达式，还包含了许多其他的语法结构，这些语法结构之间是可能存在一定的层次关系的，故笔者认为语法层次的说法比优先级更为精准。

与数据结构课程中的树结构类似，语法树也是一棵根在上、枝叶在下的倒长树，语法树的根结点是文法的开始符号。随着推导的展开，当某个非终结符被它的某个候选式替换时，将候选式的所有符号以此非终结符的子结点形式表示。在构建语法树的过程中的任何时刻，将所有的叶结点自左至右排列起来就表示此文法的一个句型。当推导过程完成后，将所有的叶结点自左至右排列起来就表示此文法的一个句子。

例 3-5 用语法树描述例 3-4 的最右推导过程。

构造语法树步骤：

- 1) 文法开始符为 E，将 E 构造为语法树的根。
- 2) 第一步推导将 E 替换为候选式 E op1 T，将这三个符号作为 E 的子结点。
- 3) 第二步推导将 T 替换为候选式 F，将符号 F 作为 T 的子结点。

- 4) 第三步推导将 F 替换为候选式 (E)，将这三个符号作为 F 的子结点。
- 5) 依次按照每一步推导构造语法树 (如图 3-1 所示)，直至结束。

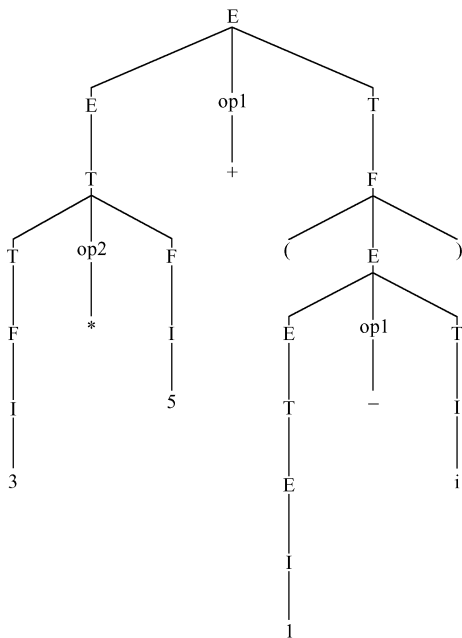


图 3-1 $3*5+(1-i)$ 的语法树

6) 最终，语法树所有叶结点自左至右排列就是 $3*5+(1-i)$ ，表明 $3*5+(1-i)$ 是文法的句子。

详细的构造过程如图 3-2 所示。

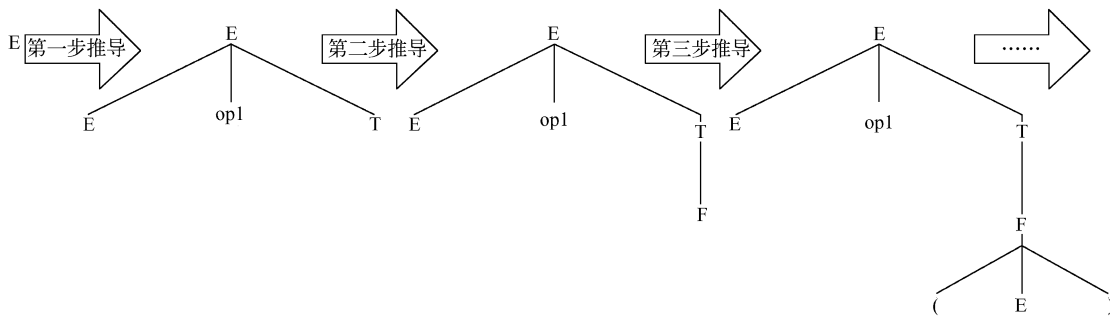


图 3-2 构造语法树示意图

同样，读者可以尝试依据例 3-3 的最左推导过程构造语法树，一定会发现得到的语法树与图 3-1 完全相同。语法树可以表示一个句型或句子的种种可能的不同推导过程，当然包括最左（最右）推导。语法树的步步成长和推导的步步展开之间是完全一致的。

语法树不但描述了推导过程，还说明了句子的语法层次，这对于理解句子的语义是至关重要的。下面，笔者对图 3-1 的语法树作深入讨论。读者不难发现语法树的叶子分布在树的不同层次上。实际上，深度越深的叶子的语法层次就越高，这是为什么呢？这与语法树的

构造顺序有关。熟悉数据结构的读者不难理解手工依据推导过程构造语法树与树的深度优先遍历比较类似。然而，文法 3-3 的表达式文法中的语法层次也就是运算符的优先级。在图 3-1 中，根据语法层次，先计算 $1-i$ ，再计算 $3*5$ ，最后计算两者的和。这个语法层次（即优先级）完全符合运算习惯。

语法树是一种语法层次的描述，在语法分析阶段是非常重要的。经典编译理论认为语法树是编译器的必不可少的输出，许多实际编译器的语法分析阶段的主要任务之一就是构建语法树。当然，实际编译器并不一定需要构造显式的语法树。隐式语法树是现代编译技术提出的观点，在基于推导方式的语法分析器中尤为常见，并在一定程度上得到了发展。这主要是由于推导的过程就是一次树的深度遍历过程，所以并不一定需要显式构建语法树。

那么，对于一个文法而言，语法树与一个句型或句子的推导之间是否存在一一对应关系呢？事实上，在大多数情况下，语法树与一个句型或句子的推导是一一对应的。不过，也确实存在例外，请参考下例分析。

例 3-6 已知文法如下：

【文法 3-4】

$$E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid \text{var} \mid \text{const}$$

句子 $3+4*6$ 存在两个不同的最左推导及语法图，如图 3-3 所示。

推导步骤				推导步骤			
推导步骤		选择候选式		推导步骤		选择候选式	
E	\Rightarrow	E + E	E \rightarrow E + E	E	\Rightarrow	E * E	E \rightarrow E * E
	\Rightarrow	3 + E	E \rightarrow var		\Rightarrow	E + E * E	E \rightarrow E + E
	\Rightarrow	3 + E * E	E \rightarrow E * E		\Rightarrow	3 + E * E	E \rightarrow var
	\Rightarrow	3 + 4 * E	E \rightarrow var		\Rightarrow	3 + 4 * E	E \rightarrow var
	\Rightarrow	3 + 4 * 6	E \rightarrow var		\Rightarrow	3 + 4 * 6	E \rightarrow var

图 3-3 二义性文法推导

注意，读者千万不要将本例的推导与例 3-3、例 3-4 的推导混淆。虽然本例表面上与例 3-3、例 3-4 非常相似，都是不同的推导步骤推出了相同的句子，但是，读者一定要深入理解其本质。例 3-3、例 3-4 是由于推导过程中选择不同的非终结符替换所致的。例 3-3 是最左推导，即每次替换最左的非终结符，而例 3-4 是最右推导，即每次替换最右的非终结符。由于最左或最右推导所构造的语法树是相同的，也就是说最左或最右推导并不影响句子的语法层次。然而，本例的两个推导都是最左推导，只是由于选择了不同的产生式推导所致。不难看出本例所产生的语法树是不同的，也就是说，本例的两个推导的语法层次结构也是不同的。左图中乘号的语法层次高于加号（表达式文法中的语法层次指的就是运算符优先级），

这是符合表达式的运算习惯的。而右图中加号的语法层次高于乘号，显然不符合运算习惯。右图把表达式 $3+4*6$ 看成 $(3+4)*6$ ，这是完全错误的。

这种给定文法 G ，如果任何基于文法的 G 的句子存在两棵或两棵以上的语法树，称 G 是存在二义性 (ambiguous) 的，或称为二义文法。二义性问题的产生并不是推导的错误，而是文法本身的问题。这类文法的语言就像自然语言一样存在二义性，二义性是计算机所不能理解与处理的。严格地说，任何程序设计语言原则上是不允许存在任何二义性的。但这并不是绝对的。在程序设计语言文法中，典型的二义性例子就是 IF-ELSE 结构，大多数程序设计语言都存在 IF ELSE 语句的二义性问题，所以编译器设计者作了 ELSE 就近匹配的约定以解决二义性问题。在 3.3 节中将详述 IF ELSE 语句的二义性及其解决方法。

3.1.4 归约简介

许多数学运算都定义了相应的逆运算或者逆操作。例如，乘法的逆运算就是除法，导数的逆运算就是不定积分。同样，推导过程也存在相应的逆过程，称为归约 (reduction)。归约思想主要源于一种自下而上的语法分析方法——“移进-归约”法。这种方法的思想是借助于一个暂存符号的栈把输入符号一个个地移进到栈里，当栈顶符号串形成某个非终结符的一个候选式时，把栈顶这部分符号串替换为该产生式的左部非终结符。通常，将这种栈顶的可归约的符号串称为句柄(handle)。对于合法的输入符号串而言，这种从输入符号串出发，经过 n 次替换最终得到文法开始符的操作就是归约。当然，如果输入的符号串无法归约得到文法开始符，即说明该符号串是非法的。

归约与推导类似，同样也存在最左归约与最右归约之分。最左归约对应的逆过程就是最右推导，而最右归约对应的逆过程正是最左推导。在形式语言中，通常将最左归约称为规范归约，而将最右推导称为规范推导。读者必须注意，这只是一个形式语言学中的专有名词而已，并不表示规范归约与规范推导就是语法分析器的核心方法。本书所讨论的基于推导的语法分析器恰恰是使用最左推导方法实现的。下面，笔者通过一个实例讲解归约的过程。

例 3-7 使用最左归约的方法验证句子 $3*5+(1-i)$ 。归约步骤如下所示：

句型	句柄	产生式
$3 * 5 + (1 - i)$	3	$I \rightarrow \text{const}$
$I * 5 + (1 - i)$	I	$F \rightarrow I$
$F * 5 + (1 - i)$	F	$T \rightarrow F$
$T * 5 + (1 - i)$	*	$op2 \rightarrow *$
$T op2 5 + (1 - i)$	5	$I \rightarrow \text{const}$
$T op2 I + (1 - i)$	I	$F \rightarrow I$
$T op2 F + (1 - i)$	T op2 F	$T \rightarrow T op2 F$
$T + (1 - i)$	T	$E \rightarrow T$
$E + (1 - i)$	+	$op1 \rightarrow +$
$E op1 (1 - i)$	1	$I \rightarrow \text{const}$
$E op1 (I - i)$	I	$F \rightarrow I$



$E \text{ op1 } (F-i)$	F	$T \rightarrow F$
$E \text{ op1 } (T-i)$	T	$E \rightarrow T$
$E \text{ op1 } (E-i)$	-	$\text{op1} \rightarrow -$
$E \text{ op1 } (E \text{ op1 } i)$	i	$I \rightarrow \text{var}$
$E \text{ op1 } (E \text{ op1 } I)$	I	$F \rightarrow I$
$E \text{ op1 } (E \text{ op1 } F)$	F	$T \rightarrow F$
$E \text{ op1 } (E \text{ op1 } T)$	E op1 T	$E \rightarrow E \text{ op1 } T$
$E \text{ op1 } (E)$	(E)	$F \rightarrow (E)$
$E \text{ op1 } F$	F	$T \rightarrow F$
$E \text{ op1 } T$	E op1 T	$E \rightarrow E \text{ op1 } T$
E		

归约是一种重要的语法分析思想，而基于归约的语法分析法主要有 LR 分析法、算符优先分析法等。由于 Neo Pascal 语法分析器是基于推导实现的，因此，本书只详细讨论基于推导的语法分析器的设计与实现。如果读者对基于归约的语法分析器感兴趣，可以参考“龙书”或其他相关资料。

3.2 语法分析概述

3.2.1 语法分析的任务

语法分析是编译过程的第二阶段，也是整个编译过程的核心阶段。语法分析的任务就是在词法分析识别得到单词流的基础上，分析并判定程序的语法结构是否符合预定义的文法。语法分析在编译器中的地位如图 3-4 所示。

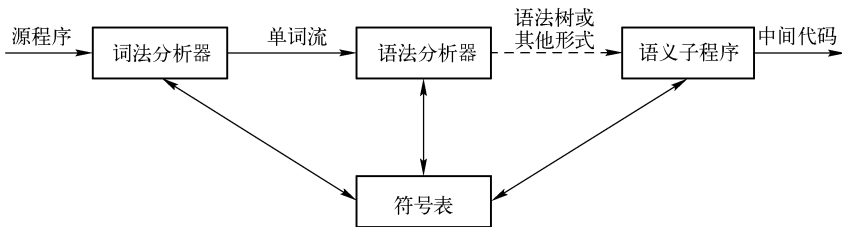


图 3-4 语法分析在编译器中的地位

语法分析器的输入就是词法分析器输出的单词流。当然，读者应该已经明确了词法分析既可以作为独立阶段，也可以作为独立的一遍处理。如果词法分析作为独立的阶段，那么词法分析器将由语法分析器调用，每次调用获得一个单词。如果词法分析作为独立的一遍，那么词法分析器将识别完成所有单词并以单词流的方式传输出至语法分析器。

语法分析器没有统一的形式输出形式。事实上，很多编译器将语法分析器作为一个主控模块，由语法分析器驱动语义分析器的方式完成语义分析及中间代码生成。在这种情况下，

语法分析器并没有显式的输出。图 3-3 中描述的语法树只是经典编译器设计中一种较为常用的输出形式。这一观点的提出主要是由于基于归约方式的语法分析器通常借助于语法树描述整个输入源程序的语法层次。不过，一些现代编译器设计的观点认为基于推导方式的语法分析器并非一定要显式构造语法树，更偏向于在语法分析过程中直接驱动语义分析子程序，而整个驱动语义分析子程序的过程正是一个遍历语法树的过程。这种观点比较适用于基于推导的语法分析法，对于基于归约的语法分析法则稍难实现。当然，是否显式构造语法树并不是完全依赖于编译器设计者的个人意志的，有时，也必须考虑程序设计语言本身的因素。关于显式构造语法树的利弊，“龙书”的描述相对比较权威，有兴趣的读者可以参阅。

前面，笔者已经引入了推导与归约两种分析方法。读者自然会问，是否可以模仿推导或归约方式设计语法分析器呢？答案是肯定的。

主流的语法分析法可以分为两类：一种是基于推导的语法分析法，另一种是基于归约的语法分析法。这两种分析法构造语法树的过程正好相反，前者的构造是从树根开始逐步向树叶发展，而后者是从树叶开始最终生成树根。因此，习惯上将前者称为自上而下的语法分析法（top-down parse），将后者称为自下而上的语法分析法(bottom-up parse)。这两类语法分析法各有优缺点。前者比较轻巧，但对程序设计语言的文法要求较高，适用于手工构造。后者比较完备，但较复杂，一般适用于工具自动生成。实际上，无论是自上而下还是自下而上的语法分析法对程序设计语言的文法都有一定的要求。尤其是自上而下的语法分析法对文法的要求比较高，对于不符合要求的文法是不能使用自上而下的语法分析法的。而自下而上的语法分析法对文法也有一定要求，虽然绝大多数程序设计语言都可以满足此要求，但还是不具有真正的通用性。除了这两类语法分析法，还有一些通用语法分析法（如 Earley Parsing）。虽然这些通用语法分析法可以构造适用于任何文法的语法分析器，但是，它们的算法复杂度过高且效率较低，故一般情况下不作考虑。相对于自下而上的语法分析算法而言，自上而下的语法分析算法似乎更受到经典编译器设计者的青睐。笔者分析了一些主流编译器后，发现大多数编译器都是采用自上而下的语法分析算法的，其中最著名的就是 GCC 与 lcc。虽然 Stallman 最初设计的 GCC 是使用 yacc 实现的，但最新的 GCC 4.x 版本都是采用了自上而下的语法分析算法。Neo Pascal 也是应用自上而下的语法分析法实现的。下面将介绍自上而下的语法分析法及其实现。

3.2.2 自上而下的语法分析法

自上而下的语法分析法的核心思想就是模拟最左推导方式设计语法分析器。为什么是最左推导呢？这个问题非常简单，参考了例 3-4 后就不难发现，如果使用最右推导构造语法分析法，终结符的匹配将从输入单词流的最末端开始，这样的处理对于处理符号表、语义分析都是非常困难的。实际上，读者可以想象，如果逆向阅读源程序的话，对于理解程序是非常不便的。同样，再比较例 3-3 的最左推导，实际上就是模拟正向阅读源程序的过程，这种方式将便于语义分析等。这就是为什么自上而下的语法分析法都是模拟最左推导的原因。

在讲述文法推导时，笔者提到了推导过程中有两个选择，即选择哪个非终结符被替换及如何选择合适的候选式替换该非终结符。前面，只是凭直觉试探地选择候选式替换非终结符，并没有从算法上讨论这个问题。针对这个问题，熟悉算法设计的读者最先想到的一种解



决办法就是回溯法。暂且不论回溯法的反复试探会大大降低语法分析的效率，回溯法本身就无法解决这个问题。通常，回溯法是在有限的集合中反复试探，当然，有时集合的数量级可能会很大。对有限语言的文法，在不考虑效率的情况下，回溯法或许是可行的。但是，程序设计语言的文法通常表示的句子集合是无限的，回溯法将无法试探穷举所有的可能句子，因此，使用回溯法就存在一定问题。下面，将详细讨论如何设计一个算法来解决这个问题。

例 3-8 简单语法分析算法的设计。

【文法 3-5】

$$\begin{aligned} G &\rightarrow aA \mid bB \\ A &\rightarrow aA \mid \epsilon \\ B &\rightarrow bB \mid \epsilon \end{aligned}$$

这个文法非常简单，对于任意输入单词流，程序 3-1 就是基于文法 3-5 的分析算法。

程序 3-1

```
1  bool B()
2  {
3      char c=getc();
4      if (c=='b')
5          return B();
6      else
7          if (c=='0')
8              return true;
9          else
10             return false;
11 }
12 bool A()
13 {
14     char c=getc();
15     if (c=='a')
16         return A();
17     else
18         if (c=='0')
19             return true;
20         else
21             return false;
22 }
23 bool G()
24 {
25     char c=getc();
26     if (c=='a')
27         return A();
28     else
29         if (c=='b')
30             return B();
31     else
```

```
32
33 }
```

```
return false;
```

程序 3-1 的实现不算太复杂，对于有递归程序设计经验的读者来说只是小菜一碟。不过，读者可能会对程序 3-1 的算法实现提出疑问：文法 3-5 所描述的语言是一个非常简单的体系，似乎根本不需要使用如此复杂的递归算法实现其分析功能。就文法 3-5 本身而言，使用此算法实现的确有杀鸡用牛刀之感。但是，笔者给出此算法实现的目的并不在于简单实现文法 3-5 的语法分析，而旨在揭示一种通用的算法实现方式。实现上，这个算法实现是一种语法分析器的模型，读者仔细分析算法实现与文法形式，不难发现文法形式与算法实现的结构有惊人的相似之处。事实上，这种语法分析器就是将文法机械地变换为等价的算法实现。由于这种语法分析器中必定存在递归，故称之为递归下降分析器。大家不要小看这种语法分析器，许多经典编译器正是使用递归下降分析器实现的，著名的 GCC、lcc 就位列其中。递归下降分析器是自上而下语法分析器的鼻祖，本书讨论的语法分析器也是递归下降分析器的一种改进形式，称为 LL(1)语法分析器。这里，笔者将从超前搜索一个单词的递归下降分析器着手，深入讨论语法分析器的构造。

显而易见，在例 3-7 中，可以通过超前搜索一个单词从而解决正确选择候选式的问题。那么，选择候选式的问题是不是真正解决了呢？事实并非如此，如果将文法 3-5 改写成文法 3-6 的形式，将无法构造类似程序 3-1 的递归下降分析器。

【文法 3-6】

$$\begin{aligned} G &\rightarrow y a A \mid y b B \\ A &\rightarrow a A \mid 0 \\ B &\rightarrow b B \mid 0 \end{aligned}$$

这是什么原因呢？比较以上两个文法，不难发现，因为 G 的两个候选式的首符号都是 y ，使用超前搜索一个单词无法作出正确选择。针对这种情况，是否必须改写分析算法呢？就此文法而言，确实可以通过改写分析算法使之超前搜索两个单词从而完成正确分析。但是，读者设想一下，由于文法产生式具有不确定性，如果 G 的两个候选式的首部有 100 个 y ，是否需要超前搜索 100 个单词呢？显然，修改分析算法的方案可行性较差。

换一个角度考虑，是否可以通过等价地改写文法解决这个问题？答案是肯定的。可以将文法改写成如文法 3-7 的形式，文法 3-7 与文法 3-5 的形式非常相似。当然，就可以构造出类似程序 3-1 的递归下降分析器，从而实现了文法 3-7 的语法分析功能。

【文法 3-7】

$$\begin{aligned} G &\rightarrow y F \\ F &\rightarrow a A \mid b B \\ A &\rightarrow a A \mid 0 \\ B &\rightarrow b B \mid 0 \end{aligned}$$

显然，通过增加一个非终结符及两个产生式就完成解决了此问题。这种处理方法与数学中提取公因子比较类似，通常称为提取公共左因子。通过前面两个文法的分析，不难发现，当文法的所有产生式右部首符号都是终结符时，就可以根据文法构造出类似程序 3-1 的递归下降分析器，从而完成语法分析的工作。但是，对于文法而言，这个条件似乎不太合理。请

参考文法 3-8:

【文法 3-8】

$$G \rightarrow A a \mid b B$$

$$A \rightarrow a A \mid 0$$

$$B \rightarrow b B \mid 0$$

就语法形式而言（暂不考虑文法描述的语言），文法 3-8 与文法 3-5、3-6、3-7 的最大区别就是 G 的一个产生式的右部首符号不是终结符而是非终结符。这个变化对于分析算法就提出了挑战。如果依然使用原来的方法构造递归下降分析器，那么得到的分析器将出现死递归现象，这显然是不正确的。由于无法确定产生式 $G \rightarrow A a$ 可以推导得到的第一个终结符，分析算法也就无法运用超前搜索一个输入单词的方法确定选择哪个产生式。读者应该不难想到，如果能够确定非终符 A 所能推导得到的所有句子的首终结符的集合，那么，就可以借助于这个集合确定何时该选择产生式 $G \rightarrow A a$ 进行推导，通常将这个集合称为 First 集合。剩下的问题关键就在于如何确定 First 集合，这里就文法 3-8 进行讨论。在文法 3-8 中非终结符 A 可以推导得到的句子为“0”、“a0”、“aa0”……等等，不难看出非终结符 A 可以推导得到的所有句子的首符号的 First 集合就是 $[0, a]$ 。由于程序设计语言的文法是预知的，可以依此方法事先求出每个产生式的 First 集合，而分析算法只需根据 First 集合选择候选式即可。

最后，再来讨论一种相对复杂的文法情况，请参考文法 3-3。就语法形式特点而言，文法 3-3 与文法 3-8 似乎有点类似，它们都存在右部以非终结符为首的产生式。但是，读者可以尝试使用文法 3-8 的处理方法变换文法 3-3，一定会发现即使求出产生式 $E \rightarrow E op1 T$ 和 $T \rightarrow T op2 F$ 的 First 集合也没有任何意义。因为，产生式 $E \rightarrow E op1 T$ 和产生式 $E \rightarrow T$ 的 First 集合必定是相同的，对于非终结符 E 而言，根据 First 集合选择候选式的算法将无法作出决策。如果同一个非终结符的候选式的 First 集合存在交集，即产生了分析冲突，表示该程序设计语言的文法不合适使用这种语法分析法。

如果诸如例 3-3 的标准表达式文法也不能分析，那么，本书讨论的语法分析法似乎已经失去了任何价值。显然，必须通过一些加工处理，使文法 3-3 满足语法分析方法的需求。

仔细分析产生式 $E \rightarrow E op1 T$ ，一定能够发现这个产生式的特点是产生式右部的第一个非终结符与产生式左部的非终结符相同，这种文法称为左递归文法。注意，读者必须考虑间接左递归文法的情况，例如：

【文法 3-9】

$$A \rightarrow B a \mid 0$$

$$B \rightarrow A b \mid 0$$

文法 3-9 表面上看并不是左递归文法，但将 $B \rightarrow A b$ 产生式的右部代入 $A \rightarrow B a$ 产生式的非终结符 B 时，即产生了左递归文法。左递归文法的最主要问题在于使用最左推导时，由于无法判定递归出口，将陷入死递归。

那么，左递归文法应该如何等价变换呢？必须明确指出：任何程序设计语言的文法必定存在递归（可以是左递归或者右递归）。因为没有递归的文法只能描述有限的语言（即语言的句子是可以枚举列出的），然而，任何程序设计语言必定是无限的语言。因此，不可能试图消除左递归文法的递归形式，只能将左递归形式等价变换为右递归形式或者其他形式，这

种变换通常称为消除左递归。通常，使用如下公式将左递归文法等价变换为右递归文法：

$$P \rightarrow P a | b \quad \xrightarrow{\text{等价变换}} \quad \begin{array}{l} P \rightarrow b R \\ R \rightarrow a R | \varepsilon \end{array} \quad (\text{"}\varepsilon\text{" 表示一个空字})$$

右边是一个右递归文法。左右文法描述的语言是等价的，而右递归文法不存在左递归文法在最左推导时的不足之处。当然，消除左递归的变换也带来了一个新的问题，那就是 ε （空字）的出现。确实，这将给语法分析带来一定的不便，但并不像左递归文法那样存在不可逾越的障碍。

消除左递归的变换中，最重要的就是如何准确辨别给定产生式中哪一部分是公式中的 a 部分，而哪一部分是公式中的 b 部分。在实际文法中， a 、 b 通常可能是符号串形式，而不是简单的单一符号。下面，通过实例说明如何辨别产生式的 a 、 b 部分。

例 3-9 消除左递归变换。

$$\begin{array}{l} E \rightarrow E \underline{op1} T | \underline{T} \\ T \rightarrow T \underline{op2} F | \underline{F} \\ F \rightarrow (E) | -F | I \\ I \rightarrow \text{var} | \text{const} \\ op1 \rightarrow + | - \\ op2 \rightarrow * | / \end{array}$$

注意：将单下画线部分看作 a 部分，将双下画线部分看作 b 部分。
经消除左递归后变成：

【文法 3-10】

$$\begin{array}{l} E \rightarrow T E1 \\ E1 \rightarrow op1 T E1 | \varepsilon \\ T \rightarrow F T1 \\ T1 \rightarrow op2 F T1 | \varepsilon \\ F \rightarrow (E) | -F | I \\ I \rightarrow \text{var} | \text{const} \\ op1 \rightarrow + | - \\ op2 \rightarrow * | / \end{array}$$

下面，再来看看右部为 ε 的候选式。当候选式右部为 ε 时，显然无法简单地求得此候选式的 First 集合。原因非常简单，源程序单词流中是不可能存在 ε 的，所以将 ε 加入 First 集合是无意义的。那么，何时选择 ε 候选式进一步推导呢？必须寻找一种有效的方法解决 ε 候选式的选择问题。

候选式的右部为 ε 即表示候选式的左部非终结符可以用 ε 替换。那么，候选式右部出现 ε 又意味着什么呢？先来看一个例子：

【文法 3-11】

$$\begin{array}{l} G \rightarrow a R b \\ R \rightarrow \varepsilon | a \end{array}$$

这个文法比较简单，它只包含两个句子，即 ab 和 aab 。显而易见，要推导出句子 ab ，符号串 $a R b$ 中的非终结符 R 只能被 ε 替换。在符号串中某个非终结符被 ε 替换时，实际上就



是将该非终结符从符号串中直接剔除。

理解了非终结符推导出 ϵ 的含义后,就可以继续讨论何时选择 ϵ 候选式了。图 3-5 是选择非终结符 R 的候选式时的指针状态示意图,指针 1 指向产生式符号串的 R ,而指针 2 由于超前搜索指向单词 b 。显然,候选式 $R \rightarrow a$ 的 First 集合是 $[a]$,指针 2 指向的单词不在候选式的 First 集合内,故不能选该候选式。那么,唯有 ϵ 候选式可选,由于单词流 ab 是合法的句子,这里先假设选择 ϵ 候选式。这样,产生式符号串的非终结符 R 被直接剔除了,指针 1 就后移至符号 b ,并继续分析匹配。不难看出,当选择 ϵ 候选式时,实际上是非终结符 R 后的符号与单词流的当前符号比较。如果能事先分析得到非终结符 R 之后允许出现的符号串集合,便可求得此集合能推导出的所有句子的首终结符集合,记作 Follow 集合。那么,Follow 集合就是选择 R 的 ϵ 候选式的依据。在文法 3-11 中,非终结符 R 后允许出现的符号串只有 b ,而由于 b 是终结符,所以它的 Follow 集合就是 $[b]$ 。也就是说,在选择非终结符 R 的候选式时,当单词流中超前搜索得到的单词是 b 时,则选择 R 的 ϵ 候选式进行推导。由于一个非终结符的 ϵ 候选式最多只能有一条,故一个非终结符的 Follow 集合是唯一的。

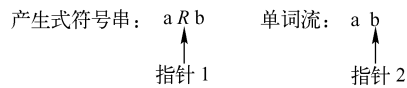


图 3-5 选择非终结符 R 的候选式时的指针状态示意图

至此,就可以借助于 First 集合与 Follow 集合辅助选择候选式了。在选择非终结符的候选式时,主要根据超前搜索得到的输入单词判定。当输入单词属于当前非终结符的某个候选式的 First 集合时,则选择该候选式进行推导。当输入单词属于当前非终结符的 Follow 集合时,则选择当前非终结符的 ϵ 候选式进行推导。必须注意两点:

- (1) 如果非终结符没有 ϵ 候选式,则它的 Follow 集合为空集。

- (2) 对于一个非终结符而言,它的全部 First 集合与 Follow 集合之间都是两两不相交的,否则说明文法不适用于超前搜索一个单词的语法分析法。

这里,引入了两个重要的集合: First 集合与 Follow 集合,这是自上而下语法分析法的核心概念。目前,读者不必深究如何计算一些复杂文法的 First 集合与 Follow 集合,只需深入理解引入 First 集合、Follow 集合的目的及其解决的问题,这远比后续学习如何计算重要得多。

综上所述,通过等价变换文法或少量修改分析算法等方式已经实现了一个语法分析器的简单原型。此原型目前还存在许多不足之处,将在后续章节中逐步完善。通过本小节学习,读者应该掌握根据文法的特点,提取公共左因子和消除左递归。另外,读者还学会了如何构造 First 集合、Follow 集合来辅助选择候选式进行推导,它们是构造语法分析器的基础。

3.2.3 构造语法分析器

Neo Pascal 的语法分析器是基于自上而下的语法分析法构造的,所以本书以详细介绍自上而下的语法分析器相关理论及其构造为主。自上而下的语法分析器短小精悍,适于手工构造。而自下而上的语法分析器的分析表非常复杂,几乎不可能手工完成。许多经典编译器偏向于手工编码完成,故自上而下的语法分析器成为不少经典编译器的不二之选。构造自上而下的语法分析器主要包括三项工作:

- 1) 变换程序设计语言的文法,使之满足自上而下的语法分析法的要求。

- 2) 构造文法的 First 集合与 Follow 集合，作为分析算法选择候选式的依据。
- 3) 编码实现自上而下的语法分析算法。

1. LL(1)文法

自上而下的语法分析法适用于大多数程序设计语言，但确实也有一些特殊的文法不适用于自上而下的语法分析法。如果给定文法符合如下两个条件（读者先不必深究）：

- (1) 文法不含左递归。
- (2) 对于文法中每一个非终结符 A 的各个候选式的 First 集合、Follow 集合两两不相交。即若

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$$

则

$$\text{First}(\alpha_i) \cap \text{First}(\alpha_j) = \phi \quad (i \neq j) \quad \text{且} \quad \left(\bigcup_{i=1}^n \text{First}(\alpha_i) \right) \cap \text{Follow}(A)$$

则称为 LL(1)文法。笔者先解释一下 LL(1)的含义，第一个“L”是 scanning from left to right 的意思，即自左向右扫描输入单词。第二个“L”是 leftmost derivation 的意思，即最左推导。而“(1)”表示超前搜索一个单词。

前面，已经讨论了如何将一个普通的文法变换为 LL(1)文法。不过，并不是任何文法都可以变换为 LL(1)文法的。通过任何方法都无法使其满足上述两个条件的文法就是真正的非 LL(1)，也就不适用于 LL(1)语法分析法了。当然，LL(1)文法也是存在不足之处的。读者不难发现，经过变换后的 LL(1)文法的可读性、可理解性远不及原始文法，这也是自上而下的语法分析法的最大缺点。

2. First 集合

下面，笔者将给出 First 集合的定义。

如果 A 是任意的文法符号串，则将从 A 推导出的串的开始符号的终结符集合称为 First(A)，即 $\text{First}(A) = \{a \mid A \xRightarrow{*} a \dots, a \text{ 是终结符}\}$ 。如果 $A \xRightarrow{*} \epsilon$ ，则 ϵ 也属于 First(A)。

当候选式不为空字时，First 集合是语法分析器选择候选式的重要依据。如果当前需被替换的非终结符为 A，而超前搜索得到的输入单词 T 属于 $\text{First}(A \rightarrow B)$ 集合，则选择候选式 B 替代非终结符 A。这里，主要讨论如何计算 LL(1)文法的 First 集合。

对于任意文法符号 A，可以应用下列规则计算 First(A)：

- 1) 如果 A 是终结符，则 First(A) 就是 {A}。
- 2) 如果 $A \rightarrow \epsilon$ 是一个候选式，则将 ϵ 加入 First(X) 中。
- 3) 如果 A 是非终结符，且 $A \rightarrow Y_1 Y_2 \dots Y_n$ 是一个产生式，则
 - a) First(Y_1) 中的所有符号加入 First(A) 中。
 - b) 如果对于某个 i 而言， $a \in \text{First}(Y_i)$ 并且 $\epsilon \in \text{First}(Y_1), \dots, \text{First}(Y_{i-1})$ ，即 $Y_1 \dots Y_{i-1} \xRightarrow{*} \epsilon$ ，则将 a 加入 First(A) 中。
 - c) 若对于所有的 $j=1, 2, \dots, k$ ， ϵ 属于 First(Y_j)，则将 ϵ 加入 First(X) 中。
- 4) 重复以上过程，直到没有终结符或 ϵ 可加到某个 First 集合为止，即迭代过程结束。

3. Follow 集合

前面已经介绍了 Follow 集合的作用。在正式给出 Follow 集合的定义之前，不妨先思考一个问题。读者应该还记得在设计词法分析器时，为了便于词法分析器超前搜索算法的实



现，统一在整个源程序文本的尾部自动添加了一个空格字符。这样，词法分析器就可以运用超前搜索算法顺利识别源程序文本的最后一个单词。那么，在设计语法分析器时会不会存在相同的问题呢？的确如此，设计语法分析器时需要定义一个特殊的结束单词以表示源程序终结，一般可以称为文法结束符，记作“\$”（有些书上记作“#”）。一门程序设计语言的文法结束符是由语言设计者定义的，文法结束符有时可以在文法中体现，例如，Pascal 中规定“.”就是文法结束符。文法结束符有时并不是显式存在的，例如，C、BASIC、Java 等并没有明确定义文法结束符，这就需要编译器设计者在设计语法分析器时隐式定义。文法结束符是一个终结符，它保证了超前搜索算法的一致性。由于 Neo Pascal 显式定义了结束符，因此，并不需要考虑隐式定义的相关技巧。

如果 A 是一个非终结符，则 Follow(A)是包含所有在句型中紧跟在 A 后面的终结符 a 的集合，即 $Follow(A) = \{a | S \xRightarrow{*} \dots Aa \dots, a \text{ 是终结符}\}$ 。如果 $S \xRightarrow{*} \dots A$ ，则 $\$ \in Follow(A)$ 。实际上，Follow(X)表示非终结符 X 后紧接着可以出现的终结符或文法终结符的集合。当候选式为 ϵ 时，Follow 集合是语法分析器选择候选式的重要依据。如果当前需被替换的非终结符为 A，而超前搜索得到的输入单词 T 属于 Follow(A)集合且不属于任何 First(A)集合时，则选择 ϵ 候选式进行推导。

对于文法符号 A，可以应用下列规则计算 Follow(A)：

- 1) 如果 A 是文法开始符，则将 \$ 置于 Follow(A)中。其中，\$ 表示文法的结束符。
- 2) 若存在产生式 $A \rightarrow aBb$ ，则将 First(b)中除了 ϵ 以外的符号都放入 Follow(B)中。
- 3) 若存在产生式 $A \rightarrow aB$ ，或者 $A \rightarrow aBb$ ，其中 First(b)中包含 ϵ （即 $b \xRightarrow{*} \epsilon$ ），则将 Follow(A)中的所有符号都放入 Follow(B)中。
- 4) 重复以上过程，直到没有终结符或 ϵ 可加到某个 Follow 集合为止，即迭代过程结束。

例 3-10 计算文法 3-10 的 First 集合与 Follow 集合。见表 3-1。

表 3-1 文法 3-10 的 First 集合与 Follow 集合

序号	产生式	First 集合	Follow 集合
1	$E \rightarrow T E1$	{(, var, const)	
2	$E1 \rightarrow op1 T E1$	{+, -}	
3	$E1 \rightarrow \epsilon$		{\$, }
4	$T \rightarrow F T1$	{(, var, const)	
5	$T1 \rightarrow op2 F T1$	{*, /}	
6	$T1 \rightarrow \epsilon$		{+, -, \$, }
7	$F \rightarrow (E)$	{(}	
8	$F \rightarrow - F$	{-}	
9	$F \rightarrow I$	{var, const}	
10	$I \rightarrow var$	{var}	
11	$I \rightarrow const$	{const}	
12	$op1 \rightarrow +$	{+}	
13	$op1 \rightarrow -$	{-}	
14	$op2 \rightarrow *$	{*}	
15	$op2 \rightarrow /$	{/}	

计算 First 集合:

为了便于计算, 首先计算右部首符号为终结符的产生式的 First 集合。由于首符号为终结符的产生式的 First 集合就是本身, 所以, $\text{First}(\text{op2} \rightarrow /) = \{/ \}$ 、 $\text{First}(\text{op2} \rightarrow *) = \{*\}$ 、 $\text{First}(\text{op1} \rightarrow +) = \{+\}$ 、 $\text{First}(\text{op1} \rightarrow -) = \{- \}$ 、 $\text{First}(\text{I} \rightarrow \text{var}) = \{\text{var}\}$ 、 $\text{First}(\text{I} \rightarrow \text{const}) = \{\text{const}\}$ 、 $\text{First}(\text{F} \rightarrow -\text{F}) = \{- \}$ 、 $\text{First}(\text{F} \rightarrow (\text{E})) = \{(\}$ 。

然后, 计算右部首符号为非终结符的产生式的 First 集合。例如, $\text{First}(\text{F} \rightarrow \text{I}) = \{\text{var}, \text{const}\}$, 因为 I 是非终结符, $\text{First}(\text{F} \rightarrow \text{I})$ 是 I 的所有产生式的 First 集合的并集。同理, 可以计算得到其他右部首符号为非终结符的产生式的 First 集合。

最后, 右部为 ϵ 候选式的 First 集合不存在的, 则无需计算。

计算 Follow 集合:

理论上讲, 每个非终结符都有一个且只有一个 Follow 集合。不过, 由于 Follow 集合仅可用于选择 ϵ 候选式, 因此, 只需计算存在 ϵ 候选式的非终结符的 Follow 集合。这里, 只需要计算 E1 和 T1 的 Follow 集合即可。另外, 计算非终结符的 Follow 集合时, 经常用到文法开始符的 Follow 集合, 为了方便计算, 文法开始符的 Follow 集合也需要事先计算。

首先计算文法开始符 E 的 Follow 集合。由于 E 是文法开始符, 故先将文法结束符 \$ 加入 $\text{Follow}(\text{E})$ 。同时, 非终结符 E 也出现在产生式 7 的右部符号串中, 不难发现, 产生式 7 的右部符号串中紧接着 E 后的是终结符 “)”, 根据计算规则 2, 终结符 “)” 也被加入 $\text{Follow}(\text{E})$ 。计算得到 $\text{Follow}(\text{E}) = \{\$, \})\}$ 。

下面, 计算 E1 的 Follow 集合。非终结符 E1 出现在产生式 1、2 的右部符号串中。由于产生式 2 的左部是 E1 本身, 不需要处理, 那么, 只需考虑产生式 1。不难发现, 产生式 1 的形式与计算规则 3 描述的产生式形式类似, 根据计算规则 3, 必须将 $\text{Follow}(\text{E})$ 集合添加到 $\text{Follow}(\text{E1})$ 中。所以, $\text{Follow}(\text{E1}) = \text{Follow}(\text{E}) = \{\$, \})\}$ 。

最后, 计算 T1 的 Follow 集合。非终结符 T1 出现在产生式 4、5 的右部符号串中。由于产生式 5 的左部是 T1 本身, 不需要处理, 那么, 只需考虑产生式 4。产生式 4 的形式与计算规则 3 描述的产生式形式类似, 根据计算规则 3, 必须将 $\text{Follow}(\text{T})$ 集合添加到 $\text{Follow}(\text{T1})$ 中, 故必须先计算 $\text{Follow}(\text{T})$ 集合。非终结符 T 出现在产生式 1、2 的右部符号串中, 而紧接着 T 后的非终结符都是 E1。根据计算规则 2, 先将 $\text{First}(\text{E1})$ 集合 (除了 ϵ 外) 添加到 $\text{Follow}(\text{T})$ 中, 此时, $\text{Follow}(\text{T}) = \{+, -\}$ 。同时, 由于 E1 存在空字候选式, 故必须将 $\text{Follow}(\text{E1})$ 添加到 $\text{Follow}(\text{T})$ 中。这样就得到了 $\text{Follow}(\text{T1}) = \text{Follow}(\text{T}) = \{+, -, \$, \})\}$ 。

计算结果见表 3-1。

4. LL(1)语法分析器

至此, 已经完成了构造语法分析器的准备工作, 主要包括变换 LL(1)文法、计算 First 集合及 Follow 集合等。当然, 运用类似于例 3-7 的方法借助于文法的 First 集合与 Follow 集合完全可以构造递归下降分析器。递归下降分析器非常有利于手工构造语法分析器, 它使语法分析器短小精悍、灵活多变。但递归下降分析器存在两个缺点:

- (1) 算法本身过多依赖程序设计语言文法。
- (2) 该算法必须有实现递归的程序设计语言编译器的支持。

虽然大多数通用机的编译器都支持递归调用, 但不支持递归调用的情况并不少见。不支

持递归调用的原因很多，大部分是由于程序设计语言或者编译器设计本身的问题，例如，早期的 Fortran 就是由于编译器采用了静态存储分配方式，从而无法实现递归调用。但是，有时也可能是由于计算机硬件系统没有系统栈结构而不支持递归调用的。在第 8 章中将详细讨论系统栈的问题。

在数据结构中，许多算法都有递归与非递归两种描述形式，有些数据结构教材甚至还讲述了如何将递归算法改写为非递归算法。同样，也可以将递归下降分析器改写成非递归的自上而下的语法分析器。递归算法的本质就是利用系统栈暂存函数的运行时环境（如局部变量、返回地址、参数等），而系统栈对于递归程序本身是透明的，用户在编写递归程序时并不需要考虑系统栈的管理。如果试图摒弃递归描述形式，就必须定义一种数据结构来模拟系统栈，栈结构当然就是不二之选了。

LL(1)语法分析器有一个输入缓冲区、一个分析栈、一张分析表和一个输出流。通常，输入缓冲区就是词法分析器的输出单词流，输出流主要用于输出语法分析过程中的出错信息。分析表是一张依据文法 First、Follow 集合构造的二维表，它是 LL(1)语法分析器的核心。而分析栈是一个用户定义的栈结构，栈的元素是文法的符号，既可以是终结符也可以是非终结符。

LL(1)语法分析器如图 3-6 所示，由一个主控程序控制，主控程序根据分析栈栈顶元素 S（符号）和输入缓冲区的当前单词符号 a 决定语法分析器的下一步动作：

- 1) 如果 $S=a=\$$ ，则语法分析成功完成。
- 2) 如果 $S=a\neq \$$ ，则语法分析器弹出分析栈栈顶符号 E，并将输入缓冲区的指针后移一个符号。
- 3) 如果 S 是非终结符，则查询分析表 T 的 $T[S, a]$ 项。分析表 T 中的项只有两种值，即 S 的某个候选式或出错标志。如果 $T[S, a]$ 项是非终结符 S 的某个候选式，则先弹出分析栈栈顶符号 S，再将 $T[S, a]$ 项的候选式右部符号串反序入栈（如果候选式右部符号为 ϵ ，则不需要入栈）。如果 $T[S, a]$ 项是出错标志，则表示输入源程序不正确，语法分析器调用错误处理程序或提示出错信息。

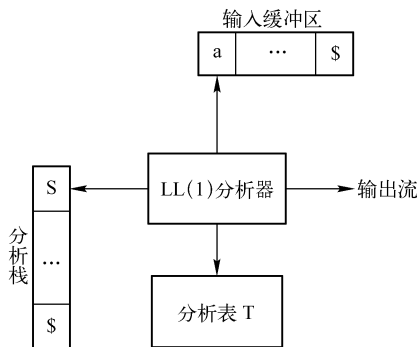


图 3-6 LL(1)语法分析器模型

4) 如果 S 是终结符，但 $S\neq a$ ，则表示输入源程序不正确，语法分析器调用错误处理程序或提示出错信息。

下面就来看看 LL(1)语法分析器的伪代码实现。

程序 3-2 LL(1)语法分析器的伪代码实现

```
1 //假设输入缓冲区 Token[]
2 把文法结束符“$”入栈;
3 把文法开始符号入栈;
4 i=1; //输入缓冲区的指向当前符号的指针
5 a=Token[i];
6 while (栈不空 && 输入符号未读完)
7 {
8     弹出栈顶符号并置入 S 中;
9     if ( $S \in V_T$ )
10    {
11        if ( $S == a$ )
12            i++;
13        else
14            出错处理或提示出错信息;
15    }
16    else
17    {
18        if ( $S == '$'$ )
19        {
20            if ( $S == a$ )
21                break;
22            else
23                出错处理或提示出错信息;
24        }
25        else
26        {
27            if ( $T[S, a]$ 是 S 的某个候选式)
28                将该候选式右部符号串反序入栈 (除  $\epsilon$  之外);
29            else
30                出错处理或提示出错信息;
31        }
32    }
33 }
34 if(栈不空 || 输入符号未读完)
35     出错处理或提示出错信息;
```

这里只给出了算法的伪代码，在后文剖析 Neo Pascal 语法分析器源代码时，将详细给出预测分析器的 C++源代码。

仔细分析 LL(1)语法分析器的算法，应该不难理解分析表的功能。分析表的作用是：当非终结符 S 遇到输入符号 a 时，辅助语法分析器选择哪条候选式进一步推导。如果无法选择合适的候选式推导，则表示输入源程序不正确。所以，分析表的表项只有两种情况，即候选式或出错标志。

前面已经介绍了递归下降分析器选择候选式的依据是文法的 First、Follow 集合。在构造分析表时，同样也不少了它们的帮助。下面通过实例讲解如何构造分析表。

例 3-11 根据文法 3-10 构造预测分析表，见表 3-2。

表 3-2 预测分析表

	+	-	*	/	var	const	()	\$
E	err	$E \rightarrow T E1$	err	err	$E \rightarrow T E1$	$E \rightarrow T E1$	$E \rightarrow T E1$	err	Err
E1	$E1 \rightarrow op1 T E1$	$E1 \rightarrow op1 T E1$	err	err	err	err	err	$E1 \rightarrow \epsilon$	$E1 \rightarrow \epsilon$
T	err	$T \rightarrow F T1$	err	err	$T \rightarrow F T1$	$T \rightarrow F T1$	$T \rightarrow F T1$	err	Err
T1	$T1 \rightarrow \epsilon$	$T1 \rightarrow \epsilon$	$T1 \rightarrow op2 F T1$	$T1 \rightarrow op2 F T1$	err	err	err	$T1 \rightarrow \epsilon$	$T1 \rightarrow \epsilon$
F	err	$F \rightarrow -F$	err	err	$F \rightarrow I$	$F \rightarrow I$	$F \rightarrow (E)$	err	Err
I	err	Err	err	err	$I \rightarrow var$	$I \rightarrow const$	err	err	Err
op1	$op1 \rightarrow +$	$op1 \rightarrow -$	err	err	err	err	err	err	Err
op2	err	Err	$op2 \rightarrow *$	$op2 \rightarrow /$	err	err	err	err	Err

首先，以非终结符为行，终结符为列，构造一张空表。

然后，依次处理文法中每个产生式，以左部非终结符定位行，以产生式的 First 集合中的终结符定位列，将该产生式填入相应单元格中。以产生式 $E \rightarrow T E1$ 为例，已经计算得到该产生式的 First 集合为 $\{(-, var, const)\}$ 。首先，定位到非终结符 E 所在的行，然后，将产生式分别填入 E 行的“(”、“-”、“var”、“const”列。对于 ϵ 产生式（例如 $T1 \rightarrow \epsilon$ ），则以 Follow 集合中的终结符定位列，将产生式填入相应单元格中。

最后，在所有空单元格中，填入“err”，表示“出错标志”。

至此，从语法分析器设计的角度重新审视先前提到的 LL(1)文法的两个条件，就不难理解其用意了。条件 1 规定 LL(1)文法必定不含左递归，关于这个条件，笔者已经详细分析了左递归文法为什么会使语法分析器失效。而条件 2 规定 LL(1)文法的任意一个非终结符的各个候选式的 First 集合及其 Follow 集合两两不相交，从构造分析表的过程分析就不难理解此条件的必然性了。如果文法中某一个非终结符的各个候选式的 First 集合及其 Follow 集合存在交集，那么，在构造分析表时，必定会有冲突（即一个单元格中填入多个产生式的情况）。显然，语法分析器就无法依据分析表选择适合的候选式，这是语法分析器无法接受的。

综上所述，笔者已经详细阐述了 LL(1)语法分析法的相关理论与技术，这是剖析 Neo Pascal 语法分析器源代码的基础。LL(1)语法分析法并不是唯一的语法分析方法，但是，它比其他语法分析法更适合于手工构造。对于一门实际的程序设计语言而言，计算 First、Follow 集合及构造预测分析表有时可能比较烦琐，但是与自下而上语法分析法的分析表相比还是简单的。当然，任何事物都存在两面性，LL(1)语法分析法也存在一定不足之处，就是文法的限制较多。且不论是否任何程序设计语言的文法都可以等价地消除左递归，即使是消除了左递归的文法也已经面目全非，可读性远不及原始的文法了。相比之下，自下而上语法分析法对文法的限制远小于自上而下语法分析法。在某些情形下，自下而上语法分析法也可以实现非常优美的语法分析器。LR 分析法是最著名的自下而上语法分析法，除了不适用于手工编码实现分析器外，它可以是一种近乎完美的语法分析法。关于 LR 的更多内容，不再赘述，读者可以参考编译原理教材。

3.3 语法分析器的实现

3.3.1 文法定义

在实现 Neo Pascal 语法分析器前，必须有一套完备的语言文法。Pascal 自 1971 年诞生后，广泛应用于教学、科研等领域。1990 年，Pascal 语言标准（ISO 7185）正式提出。这个标准的主要制定者就是 Pascal 语言创始者 Niklaus Wirth。标准详细阐述了 Pascal 语言的相关规范及约定，并完整给出了文法的 EBNF 描述形式。在过去的二十年中，这个标准得到了广泛的认同，成为现代 Pascal 编译器的设计规范。笔者在设计 Neo Pascal 文法时，主要也是参考这个标准完成的，同时加入了少量现代 Delphi 语言的语法元素。为了节省篇幅，笔者省略了原始的 Neo Pascal 文法定义，只给出 Neo Pascal 的 LL(1)文法定义形式，这是构造语法分析器的依据，见表 3-3。

表 3-3 Neo Pascal 文法

	产生式左部		产生式右部
001	程序	→	程序头 程序块 002 .
002	程序头	→	program 标识符 001 (标识符列表) ;
003	程序块	→	说明部分 048 语句部分
004	说明部分	→	包含文件说明 声明部分
005		→	声明部分
006	包含文件说明	→	uses 093 标识符列表 094 ;
007	声明部分	→	标号声明部分 声明部分 1
008		→	声明部分 1
009	声明部分 1	→	常量声明部分 声明部分 2
010		→	声明部分 2
011	声明部分 2	→	类型声明部分 声明部分 3
012		→	声明部分 3
013	声明部分 3	→	变量声明部分 声明部分 4
014		→	声明部分 4
015	声明部分 4	→	095 过程函数声明部分
016	标号声明部分	→	label 标识符 003 标号声明列表 ;
017	标号声明列表	→	, 标识符 003 标号声明列表
018		→	ε
019	常量声明部分	→	const 常量定义 ; 常量声明列表
020	常量声明列表	→	常量定义 ; 常量声明列表
021		→	ε
022	常量定义	→	标识符 102 = 表达式 004
023	类型声明部分	→	type 类型定义 ; 类型定义列表
024	类型定义列表	→	类型定义 ; 类型定义列表
025		→	ε



(续)

	产生式左部		产生式右部
026	类型定义	→	标识符 006 = 类型
027	变量声明部分	→	var 变量定义 ; 变量定义列表
028	变量定义列表	→	变量定义 ; 变量定义列表
029		→	ϵ
030	变量定义	→	013 标识符列表 040 : 类型
031	过程函数声明部分	→	过程声明部分 ; 过程函数声明部分
032		→	函数声明部分 ; 过程函数声明部分
033		→	ϵ
034	过程声明部分	→	过程头 ; 过程函数声明部分 1
035	函数声明部分	→	函数头 ; 过程函数声明部分 1
036	过程函数声明部分 1	→	043 044 程序块 047
037		→	来源指向 044 047
038	来源指向	→	045 forward
039		→	字符串常量 046
040	语句部分	→	begin 语句序列 end
041	过程头	→	procedure 标识符 041 形参列表 039
042	函数头	→	function 标识符 042 形参列表 : 038 类型 039 039 067
043	形参列表	→	(参数说明 参数部分)
044		→	ϵ
045	参数部分	→	; 参数说明 参数部分
046		→	ϵ
047	参数说明	→	035 标识符列表 036 : 类型
048		→	var 037 标识符列表 036 : 类型
049	类型	→	015 基本类型 010
050		→	015 函数类型 010
051		→	015 构造类型 010
052		→	015 指针类型 010
053		→	015 字符串类型 010
054		→	015 标识符 023 010
055	基本类型	→	有序类型
056		→	无序类型
057	有序类型	→	char 007
058		→	boolean 007
059		→	枚举类型
060		→	整数类型
061	枚举类型	→	(011 标识符列表) 014
062	整数类型	→	integer 007
063		→	byte 007
064		→	shortint 007
065		→	smallint 007

(续)

	产生式左部		产生式右部
066		→	word 007
067		→	longword 007
068		→	cardinal 007
069	无序类型	→	real 007
070		→	single 007
071	函数类型	→	procedure 021 形参列表 039 044
072		→	function 022 形参列表 : 038 类型 039 039 044
073	构造类型	→	数组类型
074		→	记录类型
075		→	集合类型
076		→	文件类型
077	数组类型	→	array 016 [数组界限 数组界限列表] of 类型 025
078	数组界限列表	→	, 数组界限 数组界限列表
079		→	ϵ
080	数组界限	→	整数常量 017 .. 整数常量 018
081	集合类型	→	set 024 of 类型 025
082	记录类型	→	record 019 字段列表 end 020
083	字符串类型	→	string 007
084	字段列表	→	字段固定部分 字段列表
085		→	字段可变部分 字段列表
086		→	ϵ
087	字段固定部分	→	标识符列表 : 类型 ;
088	字段可变部分	→	case 标识符 029 : 类型 of 033 字段可变部分 1 034 end 030 ;
089	字段可变部分 1	→	常量 031 : (字段列表) 032 ; 字段可变部分 1
090		→	ϵ
091	文件类型	→	file of 027 类型 028
092	指针类型	→	^ 简单类型
093	简单类型	→	标识符 008
094		→	char 009
095		→	boolean 009
096		→	integer 009
097		→	byte 009
098		→	shortint 009
099		→	smallint 009
100		→	word 009
101		→	longword 009
102		→	cardinal 009
103		→	real 009
104		→	single 009
105	语句序列	→	语句行 语句序列



(续)

	产生式左部		产生式右部
106		→	ϵ
107	语句行	→	语句 ;
108	语句	→	标识符 061 标识符起始语句
109		→	其他语句
110		→	ϵ
111	标识符起始语句	→	变量 1 053 := 表达式 064
112		→	: 062 语句
113			过程调用语句
114	过程调用语句		(101 实参列表) 066
115		→	ϵ 005
116	其他语句	→	goto 标识符 063
117		→	break 081
118		→	continue 082
119		→	begin 语句序列 end
120		→	while 071 表达式 073 do 语句 072
121		→	repeat 074 语句序列 until 表达式 075
122		→	for 标识符 076 := 表达式 077 for 语句后部
123		→	if 表达式 068 then 语句 if 语句后部 070
124		→	case 表达式 084 of case 分支 end 089
125		→	asm 字符串常量 , 字符串常量 083 end
126		→	with 090 变量 053 091 do 语句 092
127	for 语句后部	→	to 表达式 078 do 语句 079
128		→	downto 表达式 080 do 语句 079
129	if 语句后部	→	else 069 语句
130		→	ϵ
131	case 分支	→	常量列表 087 : 语句 088 ; case 分支
132		→	ϵ
133	表达式	→	项 表达式 1
134	表达式 1	→	关系运算符 050 项 051 表达式 1
135		→	ϵ
136	项	→	因式 项 1
137	项 1	→	低优先级运算符 050 因式 051 项 1
138		→	ϵ
139	因式	→	因子 因式 1
140	因式 1	→	高优先级运算符 050 因子 051 因式 1
141		→	ϵ
142	关系运算符	→	=
143		→	<>
144		→	<
145		→	>

(续)

	产生式左部		产生式右部
146		→	<=
147		→	>=
148		→	in
149	低优先级运算符	→	+
150		→	-
151		→	or
152		→	xor
153	高优先级运算符	→	*
154		→	/
155		→	div
156		→	mod
157		→	shr
158		→	shl
159		→	and
160	因子	→	变量 053
161		→	常量 049
162		→	(表达式)
163		→	nil 100
164		→	+ 050 因子 052
165		→	- 050 因子 052
166		→	not 050 因子 052
167		→	@ 变量 053 060
168		→	[096 表达式列表 097]
169	变量	→	标识符 054 变量 1
170	变量 1	→	[098 表达式列表 099] 059 变量 1
171		→	. 标识符 056 变量 1
172		→	^ 058 变量 1
173		→	过程调用语句
174	表达式列表	→	表达式 055 表达式列表 1
175	表达式列表 1	→	, 表达式 057 表达式列表 1
176		→	ε
177	常量列表	→	常量 085 常量列表 1
178	常量列表 1	→	, 常量 086 常量列表 1
179		→	ε
180	实参列表	→	表达式 065 实参列表 1
181		→	ε
182	实参列表 1	→	, 表达式 065 实参列表 1
183		→	ε
184	标识符列表	→	标识符 012 标识符列表 1
185	标识符列表 1	→	, 标识符 012 标识符列表 1

	产生式左部		产生式右部
186		→	ϵ
187	常量	→	整数常量
188		→	普通形式实数常量
189		→	不带十一号的科学记数形式实型常量
190		→	带十一号的科学记数形式实型常量
191		→	字符串常量

实践证明，试图凭空设计一个优秀的语言是非常困难的，一门经典的语言往往是从某些现存的语言发展继承而来的。例如，C 语言来自 B 语言、Pascal 来自 Algol 等。不过，即便如此，设计一门严谨且完美的程序设计语言并不简单，离不开程序设计语言相关理论的支持，这是一个复杂的研究课题，在此不再赘述。

下面对 Neo Pascal 的文法作简单说明。

首先，必须明确文法的非终结符集合与终结符集合。为了使文法描述形式更清晰、易于理解，笔者尽可能使用中文词组作为非终结符名，符号与符号之间以空格分隔。此文法中除了“标识符”、“整数常量”、“普通形式实数常量”、“不带十一号的科学记数形式实型常量”、“带十一号的科学记数形式实型常量”、“字符串常量”外，其他含有中文的符号都是非终结符。当然，除了非终结符与 ϵ 之外，剩下的符号就是文法的终结符了。

其次，读者可能会对文法中出现的大量数字感到疑惑。笔者在介绍文法的概念时并未提及这些数字的作用，实际上，它们既不是终结符也不是非终结符，当然，更不是标准文法的组成部分。在此，读者不必深究，稍后笔者将解释其作用。

最后，从文法形式来看，不难看出此文法是一个标准的 LL(1)文法。它既不存在公共左因子，也不存在左递归形式。至于 First 集合与 Follow 集合是否存在交集，读者可以先尝试计算验证，3.3.2 节将给出答案。

3.3.2 语法分析表

本小节将详细分析 Neo Pascal 语法分析表的构造过程及结构设计，同时，还将着手解决实际语法分析表构造过程中的一些常见问题。

前面，笔者介绍的 LL(1)语法分析器只是一个抽象模型，或者说只是一个伪代码描述的算法而已，与实际的语法分析器还存在一定差异。其中，最主要的问题就是语法分析表的构造及结构设计。

1. 构造语法分析表

仅从文法的篇幅而论，Neo Pascal 的文法的确比先前所分析的任何例子都要长得多。显然，这会构造其语法分析表带来一些麻烦，不过，其核心方法却是不变的。首先，准确计算得到文法的 First 集合及 Follow 集合。然后，根据 First 集合及 Follow 集合构造语法分析表。其中，在构造语法分析器时，计算文法的 First 集合及 Follow 集合是最烦琐的。不仅需要熟练掌握其计算方法，还需要有相当耐心才能完成。当然，读者也可以应用一些自动生成工具构造语法分析表或者语法分析器。有关自动生成工具的使用并不是本书讨论的重点，读者可以参考相关教材。

下面，笔者给出 Neo Pascal 文法的 First 集合及 Follow 集合，供读者参考。
First 集合见表 3-4。

表 3-4 Neo Pascal 文法的 First 集合

编 号	First 集合
001	program
002	program
003	label const type var procedure function begin uses
004	uses
005	label const type var procedure function begin
006	uses
007	label
008	const type var procedure function begin
009	const
010	type var procedure function begin
011	type
012	var procedure function begin
013	var
014	procedure function begin
015	procedure function begin
016	label
017	,
018	ϵ
019	const
020	标识符
021	ϵ
022	标识符
023	type
024	标识符
025	ϵ
026	标识符
027	var
028	标识符
029	ϵ
030	标识符
031	procedure
032	function
033	ϵ
034	procedure
035	function
036	label const type var procedure function begin
037	forward 字符串常量



(续)

编 号	First 集合
038	forward
039	字符串常量
040	begin
041	procedure
042	function
043	(
044	ϵ
045	;
046	ϵ
047	标识符
048	var
049	char boolean (integer byte shortint smallint word longword cardinal real single
050	procedure function
051	array record set file
052	^
053	string
054	标识符
055	char boolean (integer byte shortint smallint word longword cardinal
056	real single
057	char
058	boolean
059	(
060	integer byte shortint smallint word longword cardinal
061	(
062	integer
063	byte
064	shortint
065	smallint
066	word
067	longword
068	cardinal
069	real
070	single
071	procedure
072	function
073	array
074	record
075	set
076	file
077	array

(续)

编 号	First 集合
078	,
079	ϵ
080	整数常量
081	set
082	record
083	string
084	标识符
085	case
086	ϵ
087	标识符
088	case
089	整数常量 普通形式实数常量 不带+一号的科学记数形式实型常量 带+一号的科学记数形式实型常量 字符串常量
090	ϵ
091	file
092	^
093	标识符
094	char
095	boolean
096	integer
097	byte
098	shortint
099	smallint
100	word
101	longword
102	cardinal
103	real
104	single
105	标识符 goto break continue begin while repeat for case asm with if;
106	ϵ
107	标识符 goto break continue begin while repeat for case asm with if;
108	标识符
109	goto break continue begin while repeat for case asm with if
110	ϵ
111	:= [, ^
112	:
113	(ϵ
114	(
115	ϵ
116	goto
117	break



(续)

编 号	First 集合
118	continue
119	begin
120	while
121	repeat
122	for
123	if
124	case
125	asm
126	with
127	to
128	downto
129	else
130	ϵ
131	整数常量 普通形式实数常量 不带十号的科学记数形式实型常量 带十号的科学记数形式实型常量 字符串常量
132	ϵ
133	标识符 整数常量 普通形式实数常量 不带十号的科学记数形式实型常量 带十号的科学记数形式实 型常量 字符串常量 (nil + - not @ [
134	= < < < < > > > = in
135	ϵ
136	标识符 整数常量 普通形式实数常量 不带十号的科学记数形式实型常量 带十号的科学记数形式实 型常量 字符串常量 (nil + - not @ [
137	+ - or xor
138	ϵ
139	标识符 整数常量 普通形式实数常量 不带十号的科学记数形式实型常量 带十号的科学记数形式实 型常量 字符串常量 (nil + - not @ [
140	* / div mod shr shl and
141	ϵ
142	=
143	<
144	<
145	>
146	<=
147	>=
148	in
149	+
150	-
151	or
152	xor
153	*
154	/
155	div
156	mod

(续)

编 号	First 集合
157	shr
158	shl
159	and
160	标识符
161	整数常量 普通形式实数常量 不带+一号的科学记数形式实型常量 带+一号的科学记数形式实型常量 字符串常量
162	(
163	nil
164	+
165	-
166	not
167	@
168	[
169	标识符
170	[
171	.
172	^
173	(* / div mod shr shl and + - or xor = <> << >> = in , end until else do of then) to downto := ;
174	标识符 整数常量 普通形式实数常量 不带+一号的科学记数形式实型常量 带+一号的科学记数形式实 型常量 字符串常量 (nil + - not @ [
175	,
176	ϵ
177	整数常量 普通形式实数常量 不带+一号的科学记数形式实型常量 带+一号的科学记数形式实型常量 字符串常量
178	,
179	ϵ
180	标识符 整数常量 普通形式实数常量 不带+一号的科学记数形式实型常量 带+一号的科学记数形式实 型常量 字符串常量 (nil + - not @ [
181	ϵ
182	,
183	ϵ
184	标识符
185	,
186	ϵ
187	整数常量
188	普通形式实数常量
189	不带+一号的科学记数形式实型常量
190	带+一号的科学记数形式实型常量
191	字符串常量

表 3-4 的编号列与表 3-3 中 Neo Pascal 文法的编号是对应的, 即为产生式的编号。而表中 First 集合列则表示对应产生式的 First 集合。

Follow 集合见表 3-5。



表 3-5 Neo Pascal 文法的 Follow 集合

非终结符	Follow 集合
程序	.
标号声明列表	;
常量声明列表	type var procedure function begin
类型定义列表	var procedure function begin
变量定义列表	procedure function begin
过程函数声明部分	begin
形参列表	;:
参数部分)
数组界限列表]
字段列表	end)
字段可变部分 1	end
语句序列	end until
语句	;end until else
标识符起始语句	;end until else
过程调用语句	* / div mod shr shl and + - or xor = < <=> >= in , end until else do of then]) to downto := ;
if 语句后部	;end until
case 分支	end
表达式 1	, end until else do of then]) to downto ;
项 1	= < <=> >= in , end until else do of then]) to downto ;
因式 1	+ - or xor = < <=> >= in , end until else do of then]) to downto ;
表达式列表 1]
常量列表 1	:
实参列表)
实参列表 1)
标识符列表 1	:);

一个非终结符的 Follow 集合是唯一的，并且 Follow 集合仅在该非终结符存在空字候选式时才使用，故只需计算此类非终结符的 Follow 集合即可。

如果读者有手工构造 Neo Pascal 的 Follow 集合的经历，一定会发现 Follow (if 语句后部) = {; end until else} 与表 3-5 的 Follow (if 语句后部) = {; end until} 描述不尽相同。实际上，这并非计算错误。按照 Follow 集合的计算方法，Follow (if 语句后部) = {; end until else} 是完全正确的。那么，到底是什么原因导致了两个不同的结果呢？这就是文法二义性造成的。文法的二义性在语法分析表构造时主要的表现就是冲突，即一个单元格中填入多个产生式的情况。如果 Follow (if 语句后部) = {; end until else}，则非终结符“if 语句后部”的 Follow 集合与其相关 First 集合将存在交集，这将使语法分析表出现冲突。不过，值得注意的是，这并不意味着当一个单元格中填入了多个产生式时，就必定存在冲突。

实际上，这种现象并不罕见，例如，C、C++ 等同样存在类似的问题。在学习 C 语言时，读者应该记得 C 语言中 else 的就近匹配原则。如图 3-7 所示。

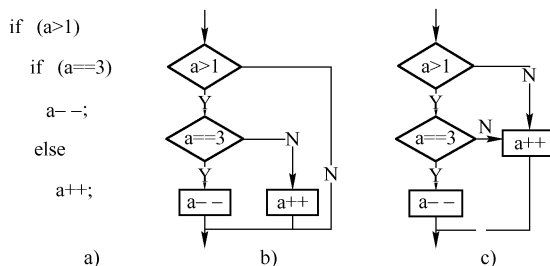


图 3-7 if-else 的流程示意图

a) 代码 b) 流程图1 c) 流程图2

就程序的形式而言，以上两个流程图应该都是正确的，既可以将 `else` 与 `if (a>1)` 匹配，也可以将 `else` 与 `if (a==3)` 匹配。显然，从这两个流程图的分析，不同的匹配方式对程序流程及运行结果是完全不同的。换句话说，就意味着程序的语义是存在二义性的，这是程序设计语言不允许的。产生二义性的原因无非就是程序设计语言文法本身的原因。读者可以尝试编写一个类似上例的 Pascal 程序，然后，参考 Neo Pascal 文法推导验证此程序。不难发现，对于 Neo Pascal 的文法非终结符“if 语句后部”遇到终结符“else”时，无论使用哪个候选式推导都是正确的。不过，生成的语法树是完全不同的，这就表明了二义性的存在。

那么，如何才能避免二义性呢？通常有两种解决方式，即修改文法或强制约定。

(1) 修改文法也就是通过改变 `if` 语句的结构来避免产生二义性。例如，BASIC 就是通过 `if` 语句结尾处增加 `endif` 关键字避免了产生二义性。

(2) 强制约定也就是在各种语义中约定一种语义是正确的，将其他的语义排除。例如，C 语言的 `else` 的就近匹配原则就是一种强制约定，它约定 `else` 只能匹配之前最近的一个 `if`。根据这个原则，在上例中，`else` 唯一能匹配的就是 `if (a==3)`。这样，强制排除其他的语义，也就避免了二义性。

同样，Pascal 语言也遵守 `else` 就近匹配原则，其约定与 C 语言完全一致。那么，这一约定如何在语法分析上体现呢？无非就是在非终结符“if 语句后部”遇到终结符“else”时，确定选择一个候选式。这样，就能避免二义性了。那么，如何确定选择哪个候选式进行下一步推导才能满足 `else` 就近匹配原则呢？最简单的处理方法就是尝试用一个小例程推导一棵语法树，结果就非常明确了。答案就是当非终结符“if 语句后部”遇到终结符“else”时，选择“if 语句后部→else 语句”这个候选式推导即可。由于没有选择使用 ϵ 产生式推导，因此必须将“else”从 Follow (if 语句后部) 中剔除。

至此，`if` 语句的二义性问题的就完美解决了。程序设计语言的二义性现象并不常见，类似 `if` 语句的二义性情况是非常特殊的。二义性既不利于语言的表述，也不利于编译器的设计。故在设计定义一门程序设计语言时，应该尽可能避免二义性。

计算得到了 Neo Pascal 的 First 集合与 Follow 集合后，就可以依此构造语法分析表了。这个工作并不复杂，先前，笔者也已经举例详述了其构造过程，故不再赘述。同时，囿于篇幅，笔者也不打算给出完整的语法分析表，读者可以参考本书的相关文档。

2. 语法分析表的结构设计

顾名思义，语法分析表就是一张静态二维表，在编译器运行过程中仅作查询操作，其数

据并不发生任何变化。显然，应用静态的二维数组存储语法分析表是比较理想的。由于 C/C++ 的二维数组下标只能是整型，因此，有必要对文法的符号进行编码，即为每个文法符号分配一个唯一的编号（ID）。

文法的符号主要包括终结符、 ϵ 、非终结符。实际上，就 LL(1) 算法而言，不难发现，文法的终结符就是词法分析阶段的单词，因此，终结符的编号完全可以沿用词法分析中的单词 ID，这是最简单且有效的方式。有关词法分析中的单词 ID，读者可以参考表 2-4。在表 2-4 中，将终结符编号归入了 001~093 之间。为了便于终结符的适当扩展，可以从 100 开始为非终结符编号，如表 3-6 所示。那么，剩下的 ϵ 只需编码为 0 即可。

表 3-6 Neo Pascal 文法非终结符表

ID	非终结符	ID	非终结符	ID	非终结符
100	程序	128	函数头	156	过程调用语句
101	程序头	129	形参列表	157	其他语句
102	程序块	130	参数部分	158	for 语句后部
103	说明部分	131	参数说明	159	if 语句后部
104	包含文件说明	132	类型	160	case 分支
105	声明部分	133	基本类型	161	表达式
106	声明部分 1	134	有序类型	162	表达式 1
107	声明部分 2	135	枚举类型	163	项
108	声明部分 3	136	整数类型	164	项 1
109	声明部分 4	137	无序类型	165	因式
110	标号声明部分	138	函数类型	166	因式 1
111	标号声明列表	139	构造类型	167	关系运算符
112	常量声明部分	140	数组类型	168	低优先级运算符
113	常量声明列表	141	数组界限列表	169	高优先级运算符
114	常量定义	142	数组界限	170	因子
115	类型声明部分	143	集合类型	171	变量
116	类型定义列表	144	记录类型	172	变量 1
117	类型定义	145	字符串类型	173	表达式列表
118	变量声明部分	146	字段列表	174	表达式列表 1
119	变量定义列表	147	字段固定部分	175	常量列表
120	变量定义	148	字段可变部分	176	常量列表 1
121	过程函数声明部分	149	字段可变部分 1	177	实参列表
122	过程声明部分	150	文件类型	178	实参列表 1
123	函数声明部分	151	指针类型	179	标识符列表
124	过程函数声明部分 1	152	语句序列	180	标识符列表 1
125	来源指向	153	语句行	181	常量
126	语句部分	154	语句	182	简单类型
127	过程头	155	标识符起始语句		

以上是标准文法的三类符号（非终结符、终结符、 ϵ ）的编码，这是构造与应用语法分析表的准备工作。

下面，着手处理语法分析表的表项设计。前面已经介绍过了，语法分析表的表项通常是文法产生式或出错标志（在 Neo Pascal 中记作“-01”）。那么，应该如何设计语法分析表的表项结构呢？由于文法产生式是一串符号的集合，因此，将语法分析表的表项声明为字符串类型是可以接受的。不过，这种设计却给语法分析表带来了大量的冗余，字符串类型作为语法分析表的表项类型并不是非常合理的。那么，应当如何设计，使语法分析表的表项结构更为合理呢？通常设计一张单独表格来存储产生式及其编号，而在语法分析表的表项中只需存储产生式的编号或者出错标志即可，如图 3-8 所示。这样，就可以将语法分析表的表项声明为整型。与字符串类型的表项相比，整型表项的冗余度较小，占用的空间也较少。稍有数据库设计经验的读者不难想到这种设计方式与数据库设计的关系分解比较类似。关系分解是数据库原理课程的相关知识，读者可以参考相关书籍。第 4 章还会涉及一些数据库设计方面的理论。

图 3-8 是语法分析表的逻辑结构示意图，描述了检索语法分析表的过程。这里，以非终结符“语句”遇到终结符“标识符”为例说明。首先，以终结符的编号定位列，即图中终结符“标识符”的编号为 001，故定位表格第 1 列。然后，以非终结符的编号定位行。由于非终结符的编号是从 100 开始的，但实际语法分析表没有必要从第 100 开始存储，因此，必须将非终结符的编号减去起始编号 100 后才进行行的定位。图中非终结符“语句”的编号为 154，减去起始编号 100 为 54，即定位到第 54 行。这样，语法分析器检索得到 108，即表示选择编号为 108 的产生式进行下一步推导分析。这个检索过程并不复杂，读者结合图 3-8 应该可以理解。

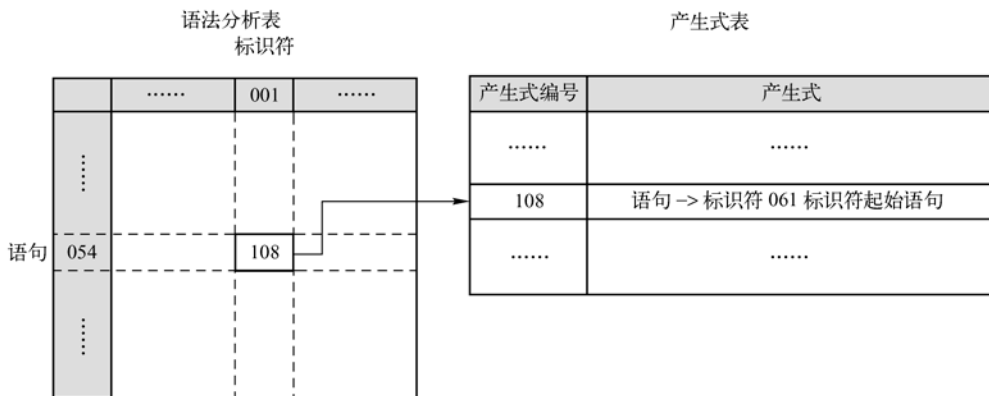


图 3-8 语法分析表与产生式表的示意图

最后，再来讨论产生式表的存储形式。首先，读者仔细思考语法分析器选择候选式进行推导的过程。语法分析器先将栈顶非终结符弹出，然后，将候选式的右部逆序入栈。整个推导过程始终只涉及候选式的右部，而没有使用候选式的左部。故方便起见，只需将产生式的右部存入产生式表即可。当然，产生式也同样需要编码。前面，已经给出了标准产生式右部的三类符号（终结符、非终结符、 ϵ ）的编码，这里，就可以直接运用了。然而，读者应该还记得 Neo Pascal 文法产生式中的那些奇怪数字。这里，笔者暂不解释这些数字的作用，只



需将其加上 300 后编码即可。根据以上产生式编码的规则，图 3-8 中编号为 108 的产生式的编码为“001361155”。

综上所述，语法分析表是一个二维数组，数组元素的类型为整型。当元素值大于或等于 0 时，表示该元素值是产生式的编号（即产生式表的下标索引），否则表示该元素值是出错标志。产生式表是一个字符串列表，每个字符串表示一个产生式的编码序列。在产生式编码序列中，可以将三个 ASCII 字符作为一组，用于表示一个符号的编码。下面，笔者给出语法分析表、产生式表的结构声明：

【声明 3-1】

```
int m_iParseTbl[120][120];      //语法分析表
vector<string> m_szProductList; //产生式表
```

下面简单介绍一下 Neo Pascal 文法中的那些数字的作用。

前面提到过语法分析器并没有统一的输出形式，其中较为常见的输出形式是语法树。然而，至今所讨论的语法分析器似乎并没有考虑任何输出形式。构造的语法分析器也只是应用于验证输入源程序的语法正确性而已，显然，这与语法分析器的目标是不符的。而 Neo Pascal 中的数字正是实现语法分析器输出的重要元素——语义子程序编号。

什么是语义子程序呢？顾名思义，就是完成语义分析处理的子程序。文法不同位置的语义几乎是完全不同的，例如，for 语句的语义与 if 语句的语义是完全不同的。语义分析就必须根据不同的文法位置完成相应的分析工作。通常，根据语义分析的需求，在文法的适合位置插入相应的语义子程序，以实现语义分析的目的。在语法分析过程中，由语法分析器根据输入源程序调用相应的语义子程序，完成相应的语义分析工作。

那么，Neo Pascal 文法中的数字正是语义子程序的编号。在此，读者不必深究语义分析的理论及语义子程序的实现，笔者将在第 4~6 章中详细讨论。只需理解文法中的数字是语义子程序的编号，将其看成产生式的一种特殊的符号即可。它与其他终结符或非终结符一样，在语法分析过程，同样需要压入分析栈。然而，当分析栈栈顶元素为语义子程序的编号时，语法分析器则将其出栈并调用相应的语义子程序，从而实现语义分析功能。

不难发现，在 Neo Pascal 文法中，语义子程序的编号与终结符的编号存在一定的冲突，即存在相同编号的现象。为了避免同号冲突，在构造语法分析表时，将语义子程序的编号加上 300 以避免冲突。而在语法分析过程中，当分析栈栈顶元素为语义子程序的编号时（即栈顶元素的值大于或等于 300），只需将此编号值减去 300 后调用相应的语义子程序即可。各类符号的编号范围如表 3-7 所示。

表 3-7 各类符号的编号范围

编号范围	符号	编号范围	符号	编号范围	符号	编号范围	符号
0	ε	1~93	终结符	100~182	非终结符	300~420	语义子程序编号

3.3.3 源代码实现

前面已经介绍了语法分析表的构造及逻辑结构的设计，这是语法分析器的核心，它将指导与控制语法分析器的工作流程。了解了语法分析表的作用后，阅读 Neo Pascal 的语法分析

器的源代码就非常容易了。实际上，Neo Pascal 语法分析器只是 3.2.3 节中介绍的 LL(1)算法模型的扩充而已。语法分析器的整体流程如图 3-9 所示。

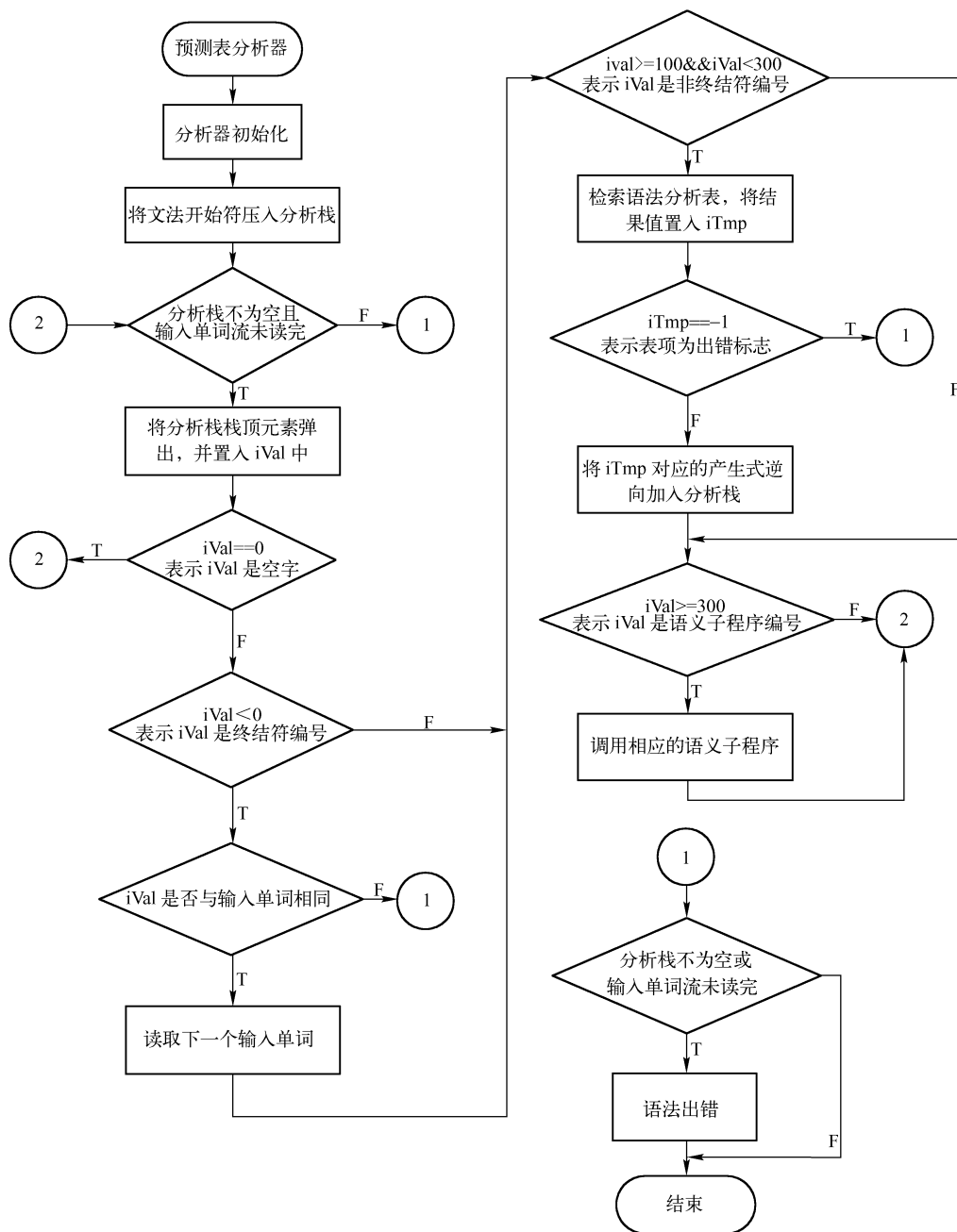


图 3-9 Neo Pascal 语法分析器算法流程设计

程序 3-3 Syntax.h

```
1 class Csyntax
```




```
2  {
3  private:
4      int m_iParseTbl[120][120];           //语法分析表
5      vector<string> m_szProductList;     //产生式列表
6      stack<int> m_ParseStack;           //分析栈，即语法分析栈
7      void EnStack(string szProduct);    //将产生式右部符号逆向推入分析栈
8      bool DeStack(int &iTop);           //将分析栈顶符号弹出，置入 iTop 变量中
9      void ClearStack();                 //将分析栈清空，初始化时使用
10 public:
11     CSyntax(void);
12     bool SyntaxParse();                //语法分析器主控函数
13     ~CSyntax(void);
14 };
```

程序 3-4 Syntax.cpp

```
1  bool CSyntax::SyntaxParse()
2  {
3      int iRow=0;
4      int iCol=0;
5      int iVal=0;
6      iListPos=0;
7      ClearStack();
8      m_ParseStack.push(100);
9      while(!m_ParseStack.empty() && iListPos<TokenList.size())
10     {
11         if (DeStack(iVal))
12         {
13             if (iVal==0)
14             {
15                 continue;
16             }
17             if (iVal<100)
18             {
19                 if (iVal==TokenList.at(iListPos).m_iKind)
20                 {
21                     iListPos++;
22                 }
23                 else
24                 {
25                     EmitError("语法错误",TokenList.at(iListPos-1));
26                     break;
27                 }
28             }
29             if (iVal>=100 && iVal<300)
30             {
31                 iCol=TokenList.at(iListPos).m_iKind;
```

```

32         iRow=iVal-100;
33         int iTmp=m_iParseTbl[iRow][iCol];
34         if (iTmp==-1)
35         {
36             EmitError("语法错误",TokenList.at(iListPos-1));
37             break;
38         }
39         Else
40         {
41             EnStack(m_szProductList[iTmp]);
42         }
43     }
44     if (iVal>=300)
45     {
46         if (!FuncList[iVal-300]())
47             break;
48     }
49 }
50 }
51 if (m_ParseStack.empty() && iListPos==TokenList.size())
52     return true;
53 else
54     return false;
55 }

```

第 3 行：临时变量（分析表行号）初始化。

第 4 行：临时变量（分析表列号）初始化。

第 5 行：临时变量（存储分析栈栈顶符号）初始化。

第 6 行：输入单词流的当前读取指针（即当前读取位置）。

第 7 行：分析栈清空。

第 8 行：将文法开始符压入分析栈。

第 9 行：循环结构条件为分析栈为空或单词流读取完毕。

第 11 行：分析栈栈顶符号出栈，置入 iVal 中。

第 13~16 行：栈顶符号为空字，不作处理，直接进行下一步分析。

第 17~28 行：栈顶符号为终结符，需要判断当前输入单词与 iVal 是否相等。如果相等，则读取指针后移一个字符，即读取下一个单词，如果不相等，则语法分析出错。

第 29~43 行：栈顶符号为非终结符，则根据该非终结符与当前单词检索语法分析表。如果检索结果为-1，则表示当前状态下不允许读取当前单词。如果检索结果不为-1，则将产生式右部符号逆序压入分析栈。

第 44~48 行：调用相应的语义子程序，这里使用函数指针的数组实现。语义子程序是无参函数，返回值为 bool 类型。如果语义子程序返回 true，则表示语义分析正确，否则表示语义分析出错。

第 51~54 行：分析栈为空且输入单词流读取完毕，即 while 是正常结束的，则表示语



法分析正确，否则语法分析出错。

程序 3-5 Syntax.cpp

```
1 void CSyntax::EnStack(string szProduct)
2 {
3     if(szProduct.length()<=0)
4     {
5         return;
6     }
7     for(int i=szProduct.length()-3;i>=0;i=i-3)
8     {
9         m_ParseStack.push(atoi(szProduct.substr(i,3).c_str()));
10    }
11 }
12 void CSyntax::ClearStack()
13 {
14     m_ParseStack.c.clear();
15 }
16 bool CSyntax::DeStack(int &iTop)
17 {
18     if(m_ParseStack.empty())
19         return false;
20     else
21     {
22         iTop=m_ParseStack.top();
23         m_ParseStack.pop();
24         return true;
25     }
26 }
```

第 1 行：EnStack 函数的作用是将产生式右部符号逆向压入分析栈。参数 szProduct 就是产生式字符串，其中每 3 个字符表示一个符号的编号。

第 3 行：如果产生式字符串为空，则无需处理。

第 7~10 行：截取产生式字符串中相应符号的编号，并将其转换为整型值后推入分析栈。

第 12 行：ClearStack 函数的作用是清空分析栈。

第 16 行：DeStack 函数的作用是将分析栈的栈顶符号弹出，并置入 iTop 变量中。

3.4 深入学习

自 20 世纪五六十年代以来，语法分析器的构造一直是编译技术中最炙手可热的话题之一。虽然它日趋完美，但是，这似乎并没有打消人们研究的热情。无论是从构造技术，还是从形式语言理论而言，关于语法分析器的资源都是最丰富的。从早期的递归下降分析器到近乎完美的 LR(k)分析器，它承载的更像是一部完整的编译技术发展史。

与词法分析器的构造类似，自动生成技术同样适用于构造语法分析器。与 Lex 完美结合

的就是 Yacc (yet another compiler compiler)，它是基于 LR 分析法实现的自动生成工具。如果读者需要了解更多信息关于 Yacc 的信息，可以访问 <http://dinosaur.compilertools.net/yacc/>。

这里，笔者推荐一个更有效的自动生成工具——ANTLR。从某种程度而言，ANTLR 可能比 Yacc 更实用，只不过 ANTLR 的诞生年代较晚，因此，不如 Yacc 那样著名。ANTLR 是基于 LL(k)分析法实现的自动生成工具，它生成的语法分析器可读性较好，而且执行效率也不错，还可以自动生成 AST。在现代编译器设计中，ANTLR 已经逐渐被设计者接受。读者可以访问 ANTLR 的主页 <http://www.antlr.org/>，以获取更多的资料。当然，与 Yacc 相比，ANTLR 凸现的是其灵活性，而 Yacc 就显得更学术派。

- 1、语言与机器：计算机科学理论导论 Thomas A. Sudkamp 机械工业出版社
说明：这本书重点介绍了形式语言相关的话题，包括上下文无关文法、下推自动机、图灵机、乔姆斯基层次、LL(k)文法等。
- 2、Yacc: Yet Another Compiler Compiler Johnson
说明：可以访问 <http://dinosaur.compilertools.net/yacc/>，以获得 yacc 的更多信息。
- 3、Bison, The YACC-compatible Parser Generator Donnelly, R. Stallman
说明：可以访问 <http://www.gnu.org/software/bison/manual/>，以获得 Bison 的更多信息。
- 4、A programmer-friendly LL(1) parser generator D. Grune, C. Jacobs
说明：这是一个基于 LL(1)实现的语法分析器的生成器，可以访问 <http://www.cs.vu.nl/~ceriel/LLgen.html>，以获取更多的信息。
- 5、形式语言与自动机导论 Peter Linz 机械工业出版社
说明：这是形式语言理论方面的经典著作，目前被很多国内高校选作形式语言课程的教材。
- 6、Bibliography on Syntax Error Handling in Language Translation Systems J. A. Dain
说明：论文详细阐述了语法分析出错处理的相关技术，可以访问 <http://compilers.iecc.com/comparch/article/91-04-050>。
- 7、Report on the Algorithmic Language Algol 60 Peter Naur, et al.
说明：论文详细阐述了文法描述形式在 Algol 60 设计中的应用。
- 8、Three models for the description of language N. Chomsky
说明：Chomsky 在论文中详细阐述了上下文无关文法的相关内容，当然，其最初的目的只是作为自然语言研究的一个分支。
- 9、编译原理及实践 Kenneth C. Louden 机械工业出版社
说明：该书描述了一个小型编译器实例——TINY，与本书的 Neo Pascal 是一脉相承的，值得初学者研究。

3.5 实践与思考

1. 本书详细阐述了 LL(1)分析法，不过，LL(1)分析法并不能解决所有上下文无关文法的语法分析问题。请读者试想 C 语言中哪些语法机制是 LL(1)分析法不能实现的。

2. 本书没有考虑出错处理及恢复机制，请读者试想如何修改与完善 Neo Pascal，以便其支持错误恢复机制。

3. 试比较 LL(1)与递归下降分析法的优劣，并构造一个分析器，用于识别例 3-3 所示文法。

4. 应用词法分析、语法分析的相关知识，设计一个简单的整型计算器，要求考虑正确的结合律及优先级。

5. 请读者为如下前缀表达式文法设计一个语法分析器。

$$\begin{aligned} \text{expr} &\rightarrow \textit{number} | (\textit{op seq}) \\ \textit{op} &\rightarrow + | - | * \end{aligned}$$



seq → expr seq expr | expr

前缀表达式是一种常用的形式，例如，表达式 $58-39*41$ 的前缀形式为 $(- 58 (* 39 41))$ 。

6. 语法树是一种较为常见的中间表示形式，请读者尝试为 Neo Pascal 构造一个语法树生成模块。

3.6 大师风采——Edsger Wybe Dijkstra

Edsger Wybe Dijkstra: 荷兰著名计算机科学家。1930年5月11日出生于荷兰鹿特丹，父亲是一位化学家，而母亲是一位数学家。1948年，他考入了 Leyden 大学，并取得了学士学位。1951年9月，他参加了英国剑桥大学开设的一个电子计算机程序设计的课程，讲师是著名的 M. V. Wilkes。这次学习为他的职业生涯打下了基础。从1984年到2000年，Dijkstra 在德克萨斯州立大学奥斯汀分校计算机科学学院任全职教授。

Dijkstra 与 Donald E. Knuth 并称为“20世纪最伟大的计算机科学家”。Dijkstra 在程序设计语言、编译技术、操作系统、数据结构与算法、程序设计、分布计算、形式验证等众多领域都有非常杰出的贡献，其中，最著名的研究成果如下：

- Algol 60。
- 操作系统 THE。
- 信号量和 PV 原语。
- 最短路径算法。

由于在 Algol 语言及编译器方面的成就，Dijkstra 获得了 1972 年图灵奖。Dijkstra 还是分布计算研究的先驱。

2002年8月6日，Dijkstra 病逝于荷兰家中，享年 72 岁。

第 4 章 符号表系统

I think conventional languages are for the birds. They're just extensions of the Von Neumann computer, and they keep our noses in the dirt of dealing with individual words and computing addresses, and doing all kinds of silly things like that, things that we've picked up from programming for computers; we've built them into programming languages; we've built them into Fortran; we've built them in PL/I; we've built them into almost every language.

——John Backus

4.1 语义分析概述

4.1.1 程序设计语言的语义

语义 (semantic) 是语言的重要组成部分之一。一门语言, 仅有词法、语法规则是不够的。虽然语法规则能够描述语言各个符号之间的组合规律, 但是始终无法说明各种符号组合的意义及其作用。换句话说, 仅有语法规则的语言只不过是一系列符号的有序序列而已, 这对于语言本身及其使用者是没有任何意义的。当然, 语言的表述能力也根本无从谈起了。如果暂且不考虑形式上的区别, 仅从语法角度而言, if 语句和 for 语句是不存在任何差异的。在语法分析器看来, 它们只是源语言的合法句子而已。由于语法描述能力的局限性, 所以需要一种元素来补充说明语言的含义, 这种元素就是语言的语义。

实际上, 语言的语义也是一组规则的集合。它们既可用于描述语言的含义, 也对语言本身作了一定的限制。与语法类似, 语义同样具有评价语言句子合法性的功能。例如, 从语法角度而言, “我走饭。” 这个句子是完整正确的。它完全符合汉语“主语+谓语+宾语”的语法规则。不过, 在日常生活中, 绝不会认为这是一个正确的句子。因为谓语“走”与宾语“饭”的搭配是不符合汉语语义的, 所以该句子是不具备任何实际含义的。这里, 以汉语句子为例, 旨在帮助读者深刻理解语法与语义的关系, 实际上, 自然语言的语义远比这个例子复杂得多。无论是人类学习语言, 还是计算机处理自然语言, 主要的障碍就是来自复杂的语义。以英语学习为例, 中国学生最精通的可能就是英语语法了, 精通程度甚至超过了母语是英语的人, 但是习惯表达、固定搭配等却是学习英语的最大障碍。很多时候, 中国学生无法写出地道的英语文章绝对不是因为语法不过关, 而是由于汉语习惯思维所致。通常, 学生会将汉语的习惯表达直接翻译成英语, 并不考虑使用英语的习惯表达来描述。由于人类至今也无法精准地描述自然语言的语义, 当然, 也就无法通过一个算法让计算机理解与处理人类的自然语言。在计算机科学中, 自然语言理解是人工智能的一个重要分支, 这里, 就不再深入讨论。下面, 笔者将重点关注程序设计语言的语义与编译器设计的联系。

早在 20 世纪 60 年代, 语法与语义的概念就被加以区分了。在程序设计语言领域, Algol 60 修订报告首次明确了语法、语义的概念, 使用 BNF 定义语言的语法, 而使用自然语



言说明及实例分析等非形式化方式描述语言的语义。在讨论 Algol 60 时, BNF 的创立者 Backus 说: 现在我们已经有了办法严格定义语言的语法了, 希望今后不久就能严格定义语言的语义。计算机科学家随即对程序设计语言语义展开了深入的研究, 试图开发一种形式化的语义描述方法, 以解决程序设计语言语义定义及语义分析器自动生成等一系列问题。经过十多年的努力, 形式语义研究取得了重大的进展, 提出了指称语义、操作语义、代数语义、公理语义等理论与方法。虽然形式语义研究硕果累累, 但是迄今为止还没有开发出一种完美、功能强大、易于理解与使用的语义定义系统。因此, 在目前状况下, 形式语义理论对编译器设计的意义是有限的。不过, 在不久的将来, 有理由相信形式语义学的发展会给编译器的设计与自动生成带来质的飞跃。

当然, 读者不要将语法与语义视作两个完全孤立的范畴。就像计算机科学中许多经典问题一样, 语法与语义之间并不存在非常严格的界线。在一些特殊情况下, 两者是可以互相转化的, 这主要是取决于文法的定义。一些宽泛的文法可能将语法问题滞留到语义分析阶段。

4.1.2 语义分析与 IR 生成的任务

读者已经知道, 编译器前端主要包括词法分析、语法分析、语义分析与 IR 生成。与词法分析、语法分析完全不同, 从算法、构造工具等角度而言, 语义分析与 IR 生成模块既没有比较成熟的通用算法, 也没有自动生成工具, 一般都是由手工编码构造的。因此, 语义分析与 IR 生成模块可能也是编译器前端较复杂的模块, 它涉及符号表设计、中间代码设计、类型系统等高级话题, 本书将在后续章节中详细分析。

编译器的任务是把输入程序翻译为等价的目标程序, 所谓“等价”指的就是语义相同, 也就是说, 尽管两个程序的语法结构与形式完全不同, 但它们所表述的含义与运行的结果必定是相同的。从经典编译技术来说, 语义分析与 IR 生成模块主要完成如下两项任务:

- (1) 对每个语法结构进行静态语义检查, 检查符合语法结构的程序是否具有实际含义。
- (2) 如果静态语义正确, 就进行程序的翻译, 即生成等价的 IR。

由于两者之间的界线可能是比较模糊的, 通常, 编译器会将它们组织在一遍中完成。

本书讨论的语义分析通常是指静态语义检查。所谓“静态语义检查”(static semantic check) 就是指在编译阶段进行的语义错误检查, 主要涉及如下几个方面:

(1) 类型检查。例如, C 语言中的“%”运算符的两个操作数类型只能是整型, 否则语义分析必须有出错提示。再如, 函数实参的类型必须与形参类型兼容等。

(2) 语句相关性检查。例如, default、case 只能出现在 switch 结构中。

(3) 一致性检查。例如, 同一作用域内的变量不允许重名、枚举常量不能重名等。

实际上, 由于程序设计语言的差异, 静态语义也不尽相同。这里只是列举了一些 C 语言中较为常见的例子而已。实际上, 在程序设计语言中, 与静态语义对应的范畴就是动态语义。笔者有必要指出, 动态语义检查(dynamic semantic check) 并不能在编译阶段完成, 因此, 也不属于经典编译技术讨论的范畴。例如, 当数组下标是一个变量时, 数组访问越界就不是静态语义检查能够判断的。通常, 动态语义检查都是在程序执行时进行的, 需要在目标程序中加入相应的语义检查代码。由于动态语义检查需要耗费运行时资源, 因此, C、Pascal 等经典编译器都不进行动态语义检查。在动态语义检查方面, Delphi 做得不错。当然, 对于解释型或混合型的翻译程序而言, 实现动态语义检查是相对容易的, 例如, Java、C#、

Visual Basic 等。

最后简单介绍一下 IR 的相关话题。IR 是输入源程序的一种内部表示形式，通常是由编译器设计者定义与使用的，对于编译器用户是透明不可见的。前端将输入源程序等价地翻译成 IR，然后，再由后端各模块将 IR 翻译成目标代码，如图 4-1 所示。

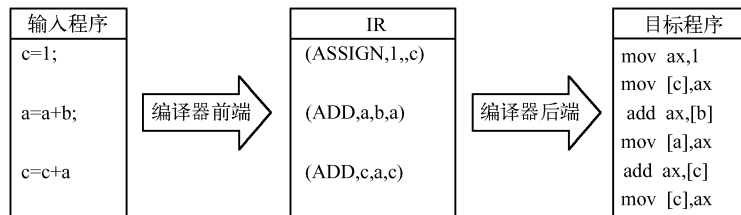


图 4-1 中间表示形式

IR 是由编译器设计者定义与使用的，根据源语言与目标语言的不同特点，设计者定义各种便于构造编译器的 IR 形式。关于 IR 形式的评价，从来就没有正确或错误之分，而只有适合或不适合。不过，无论使用哪种 IR，设计者都必须遵循一个原则：IR 要足以表达源语言的任何语义，否则，某些源语言的语义可能会在转换成 IR 的过程中丢失，语义等价的转换也就无从谈起了。

早期编译器设计中并没有使用 IR，通常是将输入源程序直接转换为目标程序。虽然在编译效率方面，生成 IR 的编译器的确无法与直接生成目标代码形式的编译器媲美，但是前者的优点却是后者无法企及的。关于 IR 的更多内容将在第 5 章中详述。

4.1.3 语法制导翻译

前面，主要讨论了语义分析与 IR 生成模块的基本任务，本小节将继续深入研究其设计与构造的相关技术。实际上，语义就是语法单位的含义，故脱离了语法空谈语义是没有任何意义的。例如，for (i=1; i<10; i++); 语句中三个表达式的含义是完全不同的，其中表达式 i<10 含有循环的终止条件的语义。但是，如果将这个表达式单独提到 for 语句之外考虑，该表达式也就不具备循环终止条件的含义了。

由于语义与语法之间存在着一定的关系，很多编译器设计者试图从语法分析器的角度思考语义分析与 IR 生成模块的设计与构造。经过不断探索，提出一种有效且通用的方案——语法制导翻译 (syntax-directed translation)。可以毫不夸张地说，目前所能见到的绝大多数编译器都是基于语法制导翻译实现的。Neo Pascal 同样如此，因此，读者有必要了解一下语法制导翻译的相关概念。

语法制导翻译就是在上下文无关文法产生式的适当位置插入相应的程序片段（称为语义子程序或语义动作），随着语法分析过程的深入，根据每个产生式中的语义子程序进行翻译的方式。

例 4-1 使用语法制导计算表达式 4+2*3 的值。

【文法 4-1】

$$E \rightarrow TE1$$

$$E1 \rightarrow op1 T \{$$

Data2=DataStack.pop();



```

    Data1= DataStack.pop();
    Op=OpStack.pop();
    switch (Op)
    {
        case '+': Rslt=Data1+Data2;break;
        case '-': Rslt=Data1-Data2;break;
        case '*': Rslt=Data1*Data2;break;
        case '/': Rslt=Data1/Data2;break;
    }
    DataStack.push(Rslt);
} E1 | ε

T → FT1
T1 → op2 F { ① ... } T1 | ε
F → (E) | -F { ②

    Data1=DataStack.pop();
    Rslt=-Data1;
    DataStack.push(Rslt);
} I

I → const { ③ DataStack.push(常量值);}
op1 → + { ④ OpStack.push('+);} | - { ⑤ OpStack.push('-);}
op2 → * { ⑥ OpStack.push('*);} | / { ⑦ OpStack.push('/);}

```

说明：

(1) 文法 4-1 是文法 3-10 的扩充，在文法 3-10 的适当位置插入了相应的语义子程序，以实现算术表达式的自动计算。

(2) 文法 4-1 仅应用于常量表达式的计算，暂不考虑变量，所以，文法 4-1 中将原来文法 3-10 的 $I \rightarrow \text{var}$ 产生式删除了。

(3) DataStack、OpStack 是两个栈结构，DataStack 的元素是常量操作数，OpStack 的元素是表达式的运算符。push、pop 是入栈和出栈的方法。

(4) 大括号之间的程序段是一个整体元素，即称为语义子程序。为了便于讲解，为每个语义子程序进行了编号。如果某些大括号的编号相同，表示其语义子程序完全相同。计算过程如图 4-2 所示。

实际上，语法分析扫描输入单词流的过程就是顺序阅读输入源程序的过程，而语义子程序就是在阅读到特定位置时需执行的动作或者需完成的工作。语法制导的实现并不复杂，也不存在太多的理论基础。这可能就是语法制导能够被广泛应用于编译器设计的主要原因，即使它不是一个完备形式系统。

语法制导翻译，顾名思义，整个翻译过程是由语法分析器控制的，所有的语义子程序同样是由其驱动的。因此，不同的语法分析方法对语义子程序的要求也是完全不同的。通常，自上而下的语法分析是一个从全局到局部的分析过程，而自下而上的语法分析是一个从局部到全局的分析过程。当然，两者各有特点，也都存在不足。国内编译原理教材大多数都是以后者的语义子程序为例讲解的。不过，笔者更倾向于前者，觉得此方法更适合手工构造。

步骤	分析栈	表达式	所用产生式
1	\$ E	4+2*3\$	
2	\$ E1 T	4+2*3\$	E->TE1
3	\$ E1 T1 F	4+2*3\$	T->FT1
4	\$ E1 T1 I	4+2*3\$	F->I
5	\$ E1 T1 {3} const	4+2*3\$	I->const {3}
6	\$ E1 T1 {3}	+2*3\$	
{3}执行结果			
7	\$ E1 T1	+2*3\$	
8	\$ E1	+2*3\$	T1->ε
9	\$ E1 {1} T op1	+2*3\$	E1->op1 T {1} E1
10	\$ E1 {1} T {4} +	+2*3\$	Op1->+{4}
11	\$ E1 {1} T {4}	2*3\$	
{4}执行结果			
12	\$ E1 {1} T	2*3\$	
13	\$ E1 {1} T1 F	2*3\$	T->FT1
14	\$ E1 {1} T1 I	2*3\$	F->I
15	\$ E1 {1} T1 {3} const	*3\$	I->const {3}
16	\$ E1 {1} T1 {3}	*3\$	
{3}执行结果			
17	\$ E1 {1} T1	*3\$	
18	\$ E1 {1} T1 {1} F op2	*3\$	T1->op2 F {1} T1
19	\$ E1 {1} T1 {1} F {6} *	*3\$	Op2->*{6}
20	\$ E1 {1} T1 {1} F {6}	3\$	
{6}执行结果			
21	\$ E1 {1} T1 {1} F	3\$	
22	\$ E1 {1} T1 {1} I	3\$	F->I
23	\$ E1 {1} T1 {1} {3} const	3\$	I->const {3}
24	\$ E1 {1} T1 {1} {3}	\$	
{3}执行结果			
25	\$ E1 {1} T1 {1}	\$	
{1}执行结果		Rslt=2*3	
第 1 步: 操作数、操作符出栈 第 2 步: 计算 第 3 步: 将计算结果压入操作数栈			
26	\$ E1 {1} T1	\$	
27	\$ E1 {1}	\$	T1->ε
{1}执行结果		Rslt=4*6	
第 1 步: 操作数、操作符出栈 第 2 步: 计算 第 3 步: 将计算结果压入操作数栈			
28	\$ E1	\$	
29	\$	\$	E1->ε

图 4-2 语法制导翻译计算表达式 4+2*3 的过程

4.2 符号表设计

4.2.1 符号表概述

符号表 (symbol table) 是一种供编译器存储关于源程序各种元素信息的数据结构。在编译过程中, 编译器自动收集所需的信息, 将其组织存储在符号表中, 同时, 依据这些信息完成代码优化和目标代码生成等。

符号表的功能包括以下几个方面:

(1) 收集源程序中各元素的信息。在编译过程中, 编译器根据源程序声明语句收集各类语法元素的信息, 并在符号表中建立相应的表项记录。

(2) 上下文语义相关性检查的依据。例如, 表达式 `a % 8` 中, 编译器必须通过检索符号表获取标识符 `a` 的信息, 尤其是其类型信息, 只有当 `a` 是整型变量或符号常量时才符合取模运算的语义。

(3) 存储分配的依据。在生成目标代码时, 编译器必须最终为每个变量 (包括用户变量和编译器产生的临时变量) 分配存储空间。由于符号表完整地记录了所有变量的信息, 因此它是目标代码生成器进行存储分配的重要依据之一。

实际上, 符号表就是编译器的中心信息库。符号表收集信息的完备与否, 直接影响编译器后续决策的质量。尤其是在优化代码及目标代码生成时, 某些符号信息对于个别算法是极为重要的。例如, C 语言中 `volatile` 修饰符对优化算法就提出了一定的要求。

从应用的角度来说, 寄希望于从符号表获得一切关于输入源程序的符号信息的思想是可以理解的, 但是, 当符号表成为一本万宝全书时, 问题也就随之而来了。为了达到这一目标, 不得不以牺牲存储空间与检索访问时间为代价, 这并不是设计者愿意看到的结果。早期编译器设计者对此极其关注, 因为效率有时决定了该编译器的命运。优秀的编译器不仅要完成正确的翻译工作, 时间、空间复杂度也是评价一个编译器的重要指标。虽然今天看来这些因素的重要性有所弱化, 但是, 它们仍然是系统软件 (操作系统、编译器) 的重要性能指标。通常, 编译器设计者必须权衡两者的关系, 并作出抉择。

那么, 哪些信息需要保存到符号表中呢? 这是根据具体程序设计语言而定的。不同程序设计语言的语法、语义规则都可能存在较大差异。对于不同范型的语言而言, 由于它们的语言模型差别巨大, 应用领域与设计目标也各不相同, 因此, 符号表的结构也必定存在差异。例如, C 语言中可以使用 `static`、`register` 等修饰变量, 而 Pascal 语言则不存在这种语法机制。因此, C 编译器的符号表必须存在描述此信息的字段, 而 Pascal 编译器则不必考虑。由于 Neo Pascal 语言是一种命令式 (或称为过程式) 语言, 所以本书所讨论的符号表更多情况下适用于命令式语言 (例如, C、Fortran 等) 编译器的构造。

通常, 编译器的符号表可能包含如下信息:

- (1) 变量 (名字、类型描述、作用域、种类、存储地址等)
- (2) 常量 (名字、常量类型、作用域、值等)
- (3) 函数 (名字、形参列表、返回类型、存储地址、种类、中间代码列表等)
- (4) 标号 (名字、作用域等)

符号表的信息收集工作主要集中在词法分析、语义分析两个阶段，其中，词法分析阶段主要是将识别得到的字面常量直接登记入常量表，其余更多的信息收集工作都是由语义分析完成的。在编译过程中，各个模块都可以从符号表中检索获取所需的信息，并不局限于编译器本身。有些编译器也会将符号表输出成一些标准的格式文件，供汇编器、链接器、调试器使用。

4.2.2 符号表的逻辑结构

本小节主要讨论设计者以何种形式来组织符号信息。符号表的设计很大程度上是一门艺术而不是科学。有时，面对几种设计方案，很难判定孰优孰劣。与数据库逻辑设计类似，在实施验证之前，仅仅凭借理论分析评价一个数据库模型可能并不是非常科学的。当然，笔者并不否认理论分析的重要性，但是太多实践经验让人们不得不承认这个观点。

总体而言，符号表的逻辑组织形式主要有两种：单表、多表。

所谓单表形式，就是指把所有的符号都组织在一张符号表中。这种组织方式的优点是管理集中。而其缺点是由于不同符号的属性字段并不相同，使用符号表的表项长度会因为符号的不同而不同，这是数据库设计者通常无法接受的。这种组织形式多见于一些早期编译器。

与单表形式相比，多表组织形式相对优越一些。

所谓多表形式，就是指把类型相似的符号（即属性字段基本相同）组织在一张符号表中，在整个编译过程中，编译器可能需要维护若干张类型不同的符号表。这种形式使得类型差别明显的符号分散在不同的符号表中。例如，可以将变量符号集中在一张变量表中，编译器在检索变量符号时，只需按变量名和作用域在变量表中检索即可，无需额外判定符号类型。从数据库设计观点而言，这种方式更符合数据库设计的思想。虽然未必所有的符号表都能符合 3NF 要求，但它的确有利于降低管理复杂度。现代编译器设计观点更倾向于这种组织形式，当然，关于符号相似程度并没有统一的标准，完全是凭借设计者经验、具体语言与目标系统而定的。

下面详细谈谈 Neo Pascal 符号表的逻辑结构。Neo Pascal 的符号表采用多表组织形式，根据 Pascal 语言的特点，Neo Pascal 设置了六张符号表：过程信息表、变量信息表、常量信息表、枚举类型信息表、类型信息表、标号信息表。笔者提出的这种分类方法未必是最佳的设计，只能说是一种可行的设计方案而已。另外，需要说明一点，为了照顾一些初学者的需要，笔者选择了 STL 的 map 模板类作为符号表的主要描述形式，而避免使用链表之类的复杂结构。map 模板类的内核实现是红黑树，它是一种极其高效的平衡树结构，在一些特殊场合下，其性能可以接近或超过 hash_map（即散列表）。因此，本书符号表中涉及的指针都是以类似于下标形式的关键字描述的，不过，为了便于讲解，本书仍然将沿用“指针”这一名称。

1. 变量信息表

变量信息表主要用于存储变量符号的相关信息。这里讨论的变量一般包括两类：用户声明的变量、编译器产生的临时变量。与用户声明的变量类似，临时变量也具有类型、所属过程等信息，但是在编译过程中由编译器自动产生的，对于用户是完全透明的。根据 Pascal 语言的特点，Neo Pascal 变量信息表表项的结构如下：

【声明 4-1】

```
struct VarInfo:ObjectInfo
```



```
{
    int m_iTypeLink;
    enum Rank {PARA,VAR,RET} m_eRank;
    bool m_bRef;
    int m_iMemoryAlloc;
    string m_szReg;
    bool m_bUsed;
    VarInfo();
}
```

m_szName: 变量名。对于临时变量而言，Neo Pascal 采用统一的命令规则，即为“_T×××”。其中，“×××”表示一个任意长度的正整数，这个值由编译器顺序编码，以此保证都不存在重名现象。而变量名以下画线开头，则保证了临时变量不可能与用户定义的变量名冲突，因为 Pascal 语言规则变量名必须以字母开头。如果是 C 语言，则必须用其他字符开头来命名临时变量，以区别 C 语言中下画线开头的变量。注意，m_szName 属性是继承自 ObjectInfo 的。

m_iProcIndex: 变量所属过程的指针。根据 Pascal 语言的语义，变量必须隶属于某一个过程，即使是全局变量也是隶属于主程序的，这与 C 语言是不同的。注意，m_iProcIndex 属性是继承自 ObjectInfo 的。

m_iTypeLink: 变量的类型描述指针。在分析变量声明时，必须在符号表中将变量类型完整地进行描述，这是后续语义分析及代码翻译的决策依据。由于类型描述比较复杂，类型符号与变量符号的属性信息也差异较大，因此，笔者将类型描述信息构造成一张独立的符号表，即类型信息表。而 m_iTypeLink 就是指向该表类型描述信息的指针。

m_eRank: 变量的类别，取值范围为 {VAR, PARA, RET}，分别表示普通变量、形参、返回值。严格地说，过程、函数的形参和返回值都不是用户声明的变量，但与用户声明的变量非常类似。对于编译器而言，同样需要对它们作语义检查、类型分析，也要分配相应的存储空间。当然，与普通用户定义的变量相比，形参、返回值毕竟又具有一定的特殊性。例如，Pascal 语言规定返回值只能作为右值。因此，符号表必须对形参、返回值加以标识。

m_bRef: 用于标识该变量的寻址方式是直接寻址还是间接寻址。

m_MemoryAlloc: 这个属性将在存储分配中使用，主要用于标识变量的逻辑存储地址。语义分析阶段并不需要关注这个属性。

m_bUsed: 标识变量是否被使用。在整个程序范围内，如果该变量没有被使用，则不必分配存储空间。

注意，VarInfo 结构是继承自 ObjectInfo 结构的。这是由于 m_szName 与 m_iProcIndex 是大多数符号共有的属性，因此，笔者将其抽象到 ObjectInfo 结构中，而其他各类具体符号的表项结构都是 ObjectInfo 的派生结构。

2. 常量信息表

常量信息表主要用于存储常量符号的相关信息。常量信息表中的常量不一定局限于词法分析器所识别获得的常量，可能还会包括编译过程中产生的一些常量值。例如，编译器在分析表达式 $a=3+5$ 时，为了提高目标程序的运行效率，通常会自动计算 $3+5$ 的值，将计算结果 8 直接赋给变量 a，而不需要生成具体的计算指令。这时，编译器必须将常量 8 记录在常量

信息表中，以便后续处理。从程序设计语言的角度来说，常量与变量是完全不同的，因为常量在目标程序中是不予分配存储空间的，而变量必须占用一定存储空间。即便如此，对于编译器而言，常量信息仍然是非常重要的，它是代码生成的依据。因此，设置常量信息表是有现实意义的。常量信息表表项的声明形式如下所示：

【声明 4-2】

```
struct ConstInfo:ObjectInfo
{
    string m_szVal;
    ConstType m_ConstType;
    StoreType m_StoreType;
    long long m_iVal;
    bool m_bVal;
    double m_fVal;
    int m_iEnumIdx;
    string m_szSet;
    bool m_bUsed;
    ConstInfo();
}
```

m_ConstType：常量的基本类型。与变量类似，常量也具有类型信息，例如，字符串型常量、整型常量等。在词法分析阶段，编译器能够识别常量符号，并将其登记到符号表中，但是编译器无法也没必要精确断言常量的实际类型，因为这项工作通常是由类型系统完成的。**m_ConstType** 的取值为{STRING, REAL, INTEGER, EREAL, BOOLEAN, ENUM, SET, PTR}，即字符串型、实型、整型、科学记数法的实型、布尔型（True、False）、枚举型、集合型、指针型（NIL）。当然，这里讨论的常量可能并不等同于 C 语言中由 **const** 修饰符指示的常量。在大多数编译器中，复杂类型的常量信息一般并不记录在常量信息表中，例如，记录类型常量、数组常量等。

m_StoreType：常量的实际类型。在编译过程中，根据常量的取值，编译器将自动计算确定常量的实际类型。例如，在分析常量 123.12 时，词法分析器可以识别它是整型常量，但无法准确断言它是到底是 INTEGER、BYTE、SHORTINT、SMALLINT、WORD、LONGWORD 中的哪一类型。类型断言的工作通常由类型系统完成。

m_szVal、**m_iVal**、**m_bVal**、**m_fVal**、**m_szSet**：常量的值。根据常量基本类型，将值存储于相应的字段中。例如，12.31 是实型常量，则将 12.31 存储在 **m_fVal** 字段中。

m_iEnumIdx：枚举类型描述指针。在大多数语言中，枚举类型数据都通常是以整型值进行存储的。例如：

```
TYPE
    num=(one,two,three,four,five,six,seven,eight,nine,ten);
    date=(Sun,Mon,Tue,Wed,Thu,Fri,Sat);
VAR
    a:num;
    b:date;
BEGIN
```



```
a:=(three);b:=(Tue);  
END.
```

在分析 `num` 类型声明时，通常会为 `num` 中的每个枚举常量分配一个唯一的序号。同样，编译器也会为 `date` 中的每一个枚举常量顺序编号。而 `a`, `b` 中存储的值则是枚举常量的编号。例如，执行完该程序段后，`a` 中的值为 2 (`three` 的编号为 2)，`b` 中的值也是 2 (`Tue` 的编号为 2)。当然，Pascal 语言的枚举类型不允许用户像 C 语言那样为枚举常量指定序号。对于枚举常量 `three` 与 `Tue` 而言，它们的实际存储值是完全相同的，然而，它们所属类型却是完全不同的。因此，在常量信息表中，登记枚举常量时必须记录其所属的枚举类型信息，以便后续阶段分析处理。

`m_bUsed`: 标识常量是否被使用。在整个程序范围内，如果该常量没有被使用，则可以不予分配关注。

3. 过程信息表

过程信息表主要用于描述输入源程序中过程、函数的相关信息。通常包括形参列表、返回值类型以及参数传递方式等。除了普通的过程、函数信息之外，如果输入源程序中使用了函数指针类型，那么，相关信息也将在过程信息表中体现。由于函数指针的属性与函数非常相似，所以，笔者将其组织在过程信息表中。早期，函数指针或者函数类型的概念主要是源于函数式语言。标准 Pascal 是较早接受函数指针概念的语言，之后一些 Pascal 编译器又参考了 C 语言的经验，改进了函数指针及其实现细节。因此，Neo Pascal 也有必要考虑函数指针的相关实现。过程信息表表项的声明形式如下所示：

【声明 4-3】

```
struct ProcInfo  
{  
    string m_szName;  
    enum Rank {Main,Sub} m_eRank;  
    enum Type {Procedure,Function,Type} m_eType;  
    vector<ParaInfo> m_ParaTbl;  
    int m_iReturnType;  
    int m_iReturnVar;  
    enum Flag {Forward,Extern,None} m_eFlag;  
    int m_iExternStr;  
    int m_iProcDefToken;  
    enum CallType {Stdcall,Fastcall} m_eCallType;  
    vector<IRCode> m_Codes;  
    set<int> m_TmpMemShare;  
    int m_ValSize;  
    int m_TmpLink;  
    bool m_bUsed;  
    bool m_bTmpUsed;  
    ProcInfo();  
}
```

`m_szName`: 过程、函数名字。注意，由于 Neo Pascal 并不允许过程、函数的嵌套定

义，所以，过程、函数名字必须唯一。

m_eRank: 过程等级。标识该过程或函数是主程序还是子过程。

m_eType: 表项类型。标识该表项是过程表项、函数表项还是函数指针类型表项。

m_ParaTbl: 形参列表信息。存储该过程或函数的形参信息，包括形参名称、形参类型描述、形参变量、形参传递方式等。

m_iReturnType: 返回类型描述指针。

m_iReturnVar: 返回变量指针。与形参类似，函数的返回值也与普通变量处理比较相似，通常，也需要在变量信息表中建立一个符号表项。

m_eFlag: 过程修饰标志。不少语言允许过程、函数的声明与使用不在同一个源程序文件中，甚至允许用户程序调用一些外部定义的过程、函数。例如，用户程序调用系统接口、库函数等。无论是哪种调用方式，编译器都必须知道被调函数的形参列表和返回值类型。熟悉 `dll` 库或 `Windows API` 的读者应该不难理解，在调用一个函数时，必须事先在程序中显式声明函数头部。在 `Pascal`、`C` 语言中，除了常规的函数声明调用以外，还允许用户程序调用以下两种函数：其他源文件中的函数、其他二进制文件中的函数。`Extern` 标志表示该表项是一个外部函数的头部声明。`Forward` 标志表示该表项是一个内部函数的向前引用声明。注意，前者是现代编译器普遍支持的，它是动态扩展编译器功能的基础。而后者是实现一遍扫描编译的基本要求，其目的就是告知编译器函数的实际定义可能在本文件稍后位置或其他源文件中，保证编译器能够在尚未完整分析函数定义之前，就能知道该如何准确调用此函数。实际上，这样的设计只是为了便于编译器能够在一遍扫描过程中完成编译。当然，在现代编译器实现中，有时也会使用多遍扫描，此时，`Forward` 引用将是完全多余的。

m_iExternStr: 外部修饰说明。仅当 **m_eFlag** 为 `Extern` 时，此字段存储外部函数相关说明信息。例如，外部函数所属的文件名、函数参数传递方式等。为了便于管理，通常将修饰信息统一存储在外部修饰信息表中，而 **m_iExternStr** 则是指向相应信息表项的指针。

m_iProcDefToken: 原型单词流。有时，编译器需要判定过程或函数声明与定义的头信息是否完全相同，仅仅依据其名字、形参列表、返回类型等语义元素作出判断并不一定正确。尤其在输入源程序中存在向前声明时，编译器必须严格判断前后两个声明形式是否完全一致，对于不完全一致的情况，编译器必须给出错误或警告提示。在 `Pascal` 语言中，这种判断是相对严格的，有时并不局限于语义层次上的一致。例如，在如下两个函数声明中，其语义是完全相同的，但是当前者是后者的向前引用时（当然，需增加 `forward` 关键字），大多数 `Pascal` 编译器并不认可，而是提示命名冲突错误。

(1) `function aa(a:integer):integer;`

(2) `function aa(a1:integer):integer;`

在 `C` 语言中，这种判断则相对较弱。`C` 编译器更多关注的是语义层次上的一致，例如，参数类型、参数个数、返回值类型等。

m_Codes: 中间代码序列，将在第 5~6 章中讨论。

m_TmpMemShare: 可共享存储区域的临时变量集合。

以下几个属性都将在第 8 章中讨论：

m_ValSize: 相关局部变量的空间大小。

m_TmpLink: 临时变量的指针。



m_bUsed: 表示过程或函数是否被使用。在整个程序范围内，如果都没有被使用，则不必分配关注。

m_bTmpUsed: 表示过程或函数中是否使用了临时变量。

下面简单介绍一下形参列表表项结构及其意义。

【声明 4-4】

```
struct ParaInfo
{
    string m_szName;
    int m_iParaType;
    int m_iParaVar;
    enum AssignType {VAL,VAR} m_eAssignType;
};
```

m_szName: 形参名。

m_iParaType: 形参类型指针。

m_iParaVar: 指向形参所关联变量的指针。

m_eAssignType: 传递类型。用于描述形参的传递方式。在程序设计语言中，参数的传递方式主要有三类：值传递、址传递、引用传递。C 语言只支持值传递，Pascal 语言支持值传递、引用传递（也就是 Pascal 中俗称的“变参”）。

4. 类型信息表

类型系统是程序设计语言的一个研究课题。在编译技术中，更多讨论的是类型系统的实现及其与编译器设计的关系。

类型可能是高级程序设计语言与汇编语言的最主要区别之一。类型的出现确实使得程序设计语言更加丰富多彩，同时也大大提高了程序设计语言的易用性。但是类型也向程序设计语言及其编译器的设计提出了新的挑战。与变量、常量、过程、函数等其他语法元素相比，类型可能是最为复杂的。本章只涉及类型系统中关于类型描述的问题。后续的章节中，还将从其他角度详细讲述类型系统。

类型描述，简而言之，就是设计一种数据结构用于描述程序设计语言的类型。在编译器设计中，由于类型描述最终体现在符号表结构中，因此，有必要在符号表设计中加以考虑。与变量、常量等不同，类型的复杂之处在于其种类繁多且组合灵活。虽然语言本身提供的基本类型（原子类型）有限，但是类型的复合却使其变得异常复杂。例如：

【声明 4-5】

```
struct
{
    union { int* a[10]; }* b;
} c[10]
```

声明 4-5 并不算太复杂，但编译器试图准确无误地描述 c 变量的类型却也不是非常容易的。从程序设计语言设计的角度来说，对于具有实际语义的类型复合，通常是不作太多限制的，只是不同的语言对类型复合的描述形式可能不尽相同。例如，Pascal 中规定形参列表中的类型只能是基本类型或用户自定义类型，却不能是匿名类型，而 C 语言中则相对宽松得多。

虽然类型复合比较复杂，但是任何复杂的类型复合的语义是确定唯一的。在语义可能存在二义性时，编译器就必须设法回避。例如，在 C 语言中，经常会借助于括号来区别指向函数的指针和返回指针的函数。

不同编译器的类型描述可能存在一定的差异，但它们的内核却是惊人地相似，都是使用类型链的形式加以描述的。其基本思想就是以一种线性结构来精准地描述类型的修饰关系，这是一个非常有效且灵活的描述形式。当然，类型链并不是唯一的描述形式。在早期 C 语言编译器中，出现过位串形式的类型描述。不过，这个方式存在一定的局限，它不足以描述任意形式的类型复合，也不能用于描述数组等类型。

类型链就是按照语义层次将复合形式的类型逐一分解成单一的类型结点，并将这些类型结点依次链接形成一个线性结构。这里，读者必须注意两点：第一，类型链是按语义层次展开的，也就是说，类型链表元素的顺序不同，其描述的类型是不同的。第二，不同程序设计语言的类型表示形式差别较大，C 语言是前缀形式，而 Pascal 则是后缀形式。因此，编译器设计者应该根据源语言的特点确定类型语义的层次。例如，`int* a[10]`的语义是声明一个含有 10 个元素的数组，数组元素的基类型是指向整型的指针，所以类型链的描述为“数组->指针->整型”。

不同编译器类型链的实现方式可能差异较大，实际上，这个问题的核心就是讨论线性结构的存储形式。关于这个问题，读者可以参考数据结构教材，本书不再赘述。Neo Pascal 的类型链是采用了一种类似于顺序线性表的形式，也就是类型信息表。下面，笔者将详细分析类型信息表的表项结构。

【声明 4-6】

```
struct TypeInfo:ObjectInfo
{
    StoreType m_eDataType;
    int m_iLink;
    vector<FieldInfo> m_FieldInfo;
    vector<ArrayInfo> m_ArrayInfo;
    StoreType m_eBaseType;
    string m_szContent;
    int m_iState;
    int m_iSize;
    TypeInfo();
};
```

m_szName: 类型名。与变量声明不同，C、Pascal 等语言都支持匿名类型。一般而言，即使是匿名类型，系统也会为其自动分配一个唯一的名字。

m_eDataType: 数据类型。是一个枚举类型的字段，用于描述该表项的类型标识。取值如下所示：

T_NONE	T_CHAR	T_BOOLEAN	T_INTEGER	T_BYTE	T_SHORTINT
T_SMALLINT	T_WORD	T_LONGWORD	T_CARDINAL	T_REAL	T_SINGLE
T_ENUM	T_ARRAY	T_SET	T_RECORD	T_STRING	T_FILE
T_POINTER	T_LONG8	T_FUNC	T_PROC	T_USER	



这里有必要作两点说明：

(1) T_NONE 与 C 语言中的 void 是不同的，void 是 C 语言提供了一种特殊数据类型，而 T_NONE 仅用于标识该表项未被使用而已。

(2) T_LONG8 是一个特殊的数据类型，表示 8 字节的有符号整型。由于一些经典程序设计语言的标准都是在 16 位、32 位机时代形成的，包括 Pascal 在内的许多语言都是不支持 8 字节整型的。不过，大多数通用机的编译器内部都是支持 8 字节整型的。这是由于许多程序设计语言的整型都存在有、无符号之分。编译器在处理无符号整型值与有符号整型值进行比较类运算时，必须将无符号整型值转换为等值的有符号整型值。这是因为汇编比较指令要求两个操作数的符号类型是相同的，可以同为有符号数或同为无符号数，但不允许一个有符号数与一个无符号数进行比较。这种情况下，就必须进行强制类型转换。显然，将有符号整型转成无符号整型是不可能的，因此，只能将无符号整型转换为有符号整型。但是，就正整数而言，无符号整型的表示范围要远大于有符号整型。例如，4 字节的无符号整型最大能表示的值为 4 294 967 295，而 4 字节的有符号整型最大能表示的值为 2 147 483 647。为了不影响运算结果，编译器通常将两者同时转换成 8 字节的有符号整型进行比较。

m_iLink：指向下一个类型结点的指针，即为下一个类型结点的下标。不过，如果当前表项为枚举类型时，由于枚举类型无法继续进行类型复合，所以 **m_iLink** 就复用作为指向枚举常量列表的指针。Neo Pascal 有一张全局符号表用于存储枚举常量值。稍后，笔者将详细举例说明。在 Pascal 中，枚举常量的名字与普通标识符类似，是共享同一个名字空间（命名空间）的。例如，声明代码如下：

```
VAR a:(a1,a2,a3); a2:INTEGER;
```

编译器通常会提示 a2 标识符已被使用。

m_FieldInfo：记录类型字段列表。前面读者已经对 Pascal 记录类型有了简单了解。实际上，Pascal 的记录类型并不复杂，只是对于大多数读者而言，更习惯使用 C 语言的结构、联合类型而已。

m_ArrayInfo：数组类型的维度列表。对于数组类型而言，编译器在类型信息表中详细记录数组的维度及其上、下限。与 C 语言不同，Pascal 允许用户定义数组的上、下限。关于数组类型的维度描述，通常还有一种比较常见的形式，那就是不使用列表描述多维数组，而是将多维数组视作多个一维数组的复合，也就是说，将使用 n 个类型结点来描述一个 n 维数组。

m_eBaseType：基本类型。使用类型链描述类型几乎是公认的，但是对于类型链长度问题的观点却并不一致。类型链的长度也就是指类型结点的细化程度。一般而言，笔者并不支持极短的类型链形式，因为这种形式失去了构造类型链的意义，但也不表示过长的类型链是绝对占优的。在设计符号表时，应尽可能保证类型链中每个结点的基本类型都可以独立描述。不能满足此条件的表项，可拆分成两个或多个表项，并以类型链形式钩链。

m_szContent：备注。

m_iState：处理状态。这是一个非常重要的标志字段。下面，笔者通过一个 C 语言的实例来说明 **m_iState** 字段的作用，如表 4-1 所示。

表 4-1 递归声明的实例

A	B	C
<pre>struct aa { int a1,a2; aa a3; };</pre>	<pre>struct aa { int a1,a2; aa* a3; };</pre>	<pre>TYPE aa=^Node; Node=RECORD a1:INTEGER; a2:aa; END;</pre>

表面上看，A 列的声明并没有什么错误。可上机调试后，读者会发现，C 语言并不允许这样的声明形式。由于结构 aa 的 a3 字段的类型是 aa 本身，因此，也将这种声明形式称为“递归声明”。实际上，包括 C、Pascal 等绝大多数程序设计语言都不支持递归声明。因为编译器无法准确计算这种类型的变量所需的存储空间大小，那么，也就无法进行存储空间的分配了。解决这个问题的方法并不复杂，编译器在处理一个类型时，只需在类型信息表中作一个标记，标识该类型是否已经构造完毕。如果类型没有构造完毕，则不允许对其引用。例如，编译器分析到 aa a3 时，发现 aa 类型本身并没有完成构造，这时编译器会给出错误提示。不过，语义子程序无法根据类型信息表判断类型是否已构造完成，因此笔者在类型信息表项中增加了 m_iState 字段，就是用于标识该类型信息是否已完成构造。只有当语义子程序完成整个类型分析后，才将 m_iState 置为 1，否则为 0。然而，B 列的声明形式却是特例。在 C 语言的结构中，允许声明指向本身的指针字段。由于指针字段占用的存储空间大小是由目标机决定的，与具体指向的类型无关，因此，这种声明并不会影响编译器的存储分配。虽然不同的程序设计语言对这种声明的支持不尽相同，但是这种声明形式却是常见的。在 Pascal 语言中，不允许像 C 语言那样直接在结构体内声明，但支持形如 C 列的声明。在这种情况下，编译器通常会作特殊处理。以 C 列声明为例，在分析 aa=^Node 时，实际上，Node 类型名是未知的，按编译器一贯的处理方式，这时应该给予无效类型名提示。不过，在处理指针类型（变量）声明时，编译器并不会立即判断基类型是否已声明，而通常是在事后某一特定时刻进行判断。如果编译器无法从后续声明中发现 Node 类型的声明，则依然会报告出错信息。

m_iSize: 占用存储空间的大小。在编译过程中，编译器必须能准确计算该类型变量所需存储空间的大小，以便进行存储分配。因此，这个字段在存储分配及目标代码生成中具有极其重要的意义。

下面再来谈谈记录类型字段列表、数组类型的维度列表的结构。

记录类型字段列表并不是全局符号表。通常，一个记录类型结点有一张独立的记录类型字段列表，这与枚举类型信息表是不同的。声明形式如下：

【声明 4-7】

```
struct FieldInfo:ObjectInfo
{
    int m_iLink;
    int m_iSize;
    string m_szVarFieldFlag;
    string m_szVarFieldConst;
    int m_iState;
    int m_iOffset;
```

```
FieldInfo();
};
```

m_iLink: 字段类型指针。字段的类型可能是非常复杂的，因此，字段类型与变量类型的描述形式是完全一致的。

m_iSize: 字段占用的字节数。

m_szVarFieldFlag: 可变部分的标志字段名。先来看一个简单的例子，如图 4-3 所示。

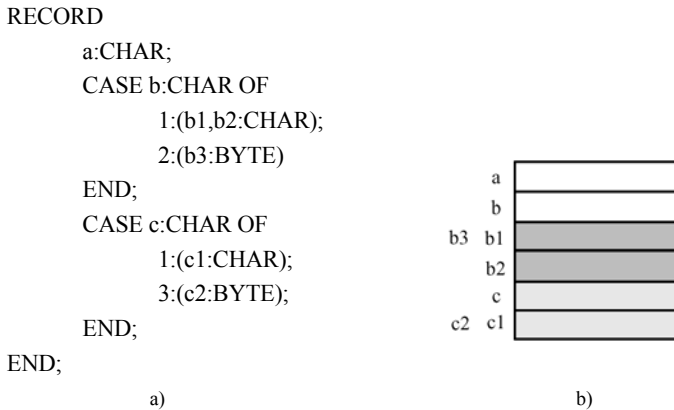


图 4-3 可变字段存储分配示意图

a) 代码 b) 存储分配情况

这里，暂且将 CASE 结构后的字段称为“可变部分的标志字段”，它也是一个独立的字段。从存储分配的情况来分析，实际上，编译器将一组 CASE 结构视为一个共用体进行分配。不同 CASE 结构之间是独立分配存储空间的，并不是共用的。CASE 标号之间的字段组共用同一块存储区域，例如，(b1,b2)与(b3)就是共用一片存储区域。而字段组内的字段分量则不共用存储区域，例如，b1 与 b2 都是独立分配存储空间的。CASE 标号仅用于标识字段组，也就是说，一组 CASE 结构内的标号相同的字段组成一个字段组。然而，CASE 标号本身却没有实际意义，将上例中的标号“3:”改成“4:”是没有区别的。从编译器的角度而言，并不关注 CASE 标志字段与标号本身。编译器只需要对记录类型中的字段组进行标识，并且记录哪些字段组可以共用同一块存储区域即可。在记录类型信息表中，就是使用 **m_szVarFieldFlag** 字段来标识哪些字段隶属于同一个可变部分。在 Neo Pascal 中，存储的就是 CASE 标志字段的名称，如上例中的 b、c。

m_szVarFieldConst: 可变字段的标号。在 Neo Pascal 中，该字段存储的就是 CASE 标号的值，**m_szVarFieldFlag**、**m_szVarFieldConst** 都相同的记录集就表示一个字段组，字段组内部的分量是不共用存储区域的。

m_iState: 处理状态。将在后续章节中详细讨论。

m_iOffset: 字段偏移量。即字段首地址相对于记录首地址的字节偏移量。

下面，再来看看数组类型的维度列表。数组类型的维度列表就是用于描述数组维度的上限、下限信息的辅助符号表。与记录类型字段列表类似，它也不是全局符号表，一个数组类型有一张独立的维度列表。与字段列表的结构相比，维度列表要简单得多，只有上限、下限两个字段信息。声明形式如下：

【声明 4-8】

```
struct ArrayInfo
{
    int m_iStart;
    int m_iEnd;
    ArrayInfo();
};
```

5. 枚举信息表

在介绍类型信息表时，已经提到了枚举信息表。枚举信息表主要用于存储枚举类型的枚举常量信息。与 C 语言相比，Pascal 的枚举类型简单一些。因此，Neo Pascal 的枚举信息表结构也比较简单。此处不再详细给出其声明形式，请读者自行参考 Neo Pascal 源代码。

6. 标号信息表

标号信息表主要用于存储输入源程序的标号信息。虽然 Pascal 创始人 Wirth 教授并不提倡使用 goto 语句，甚至在其结构化程序设计思想中也竭力反对 goto 语句，不过，Pascal 语言仍然提供 goto 及标号机制。Pascal 规定标号使用前必须进行声明，这与 C 语言是不同的。当然，值得注意的是，标号信息表中不仅包含用户声明的标号，还需要包含编译器自动生成的临时标号。声明形式如下：

【声明 4-9】

```
struct LabelInfo:ObjectInfo
{
    bool m_bDef;
    bool m_bUse;
};
```

m_szName: 标号名。

m_bDef: 是否已定义。在 Pascal 中，如果用户声明了一个标号，那么，该标号在过程、函数体内的位置定义点的个数必须小于或等于 1。也就是说，Pascal 允许声明的标号没有具体的位置定义点，当然，这种标号是不允许被 goto 语句使用的。在分析源程序的过程中，发现一个标号的位置定义点时，将标号信息表中对应的表项的 m_bDef 字段置 true，以免重复。

m_bUse: 是否已使用。标号的使用是根据源程序而定的。Pascal 语言允许用户声明了一个标号后却不使用。对于未使用的标号，通常，编译器会在分析过程中予以标识，便于后端优化处理。当源程序中 goto 语句使用了一个标号后，编译器则将标号信息表中对应的表项的 m_bUse 字段置 true，表示该标号已被使用。

至此，已经分析了 Neo Pascal 的符号表结构。符号表的结构对于理解编译器的设计具有极其深远的意义。因此，笔者认为有必要通过完整的实例分析 Neo Pascal 的符号表系统，希望读者能结合下一小节的实例深刻理解符号表的设计思想。

4.2.3 符号表的实例分析

由于 Neo Pascal 是基于 C++ 开发的，较之早期的经典编译器，Neo Pascal 的符号表系统是比较简单的。为了突出符号表的核心内容，笔者把更多的与编译器设计无关的实现细节交

由 C++ 编译器完成了。

例 4-2 符号表实例 (图 4-4)。

```

PROGRAM aa(input,output);
LABEL L1,L2;
CONST
    c1=123;    c2=12.3;
TYPE
    t1=RECORD
        t1,t2:INTEGER;
    END;
VAR
    v1:ARRAY [1..10] of INTEGER;
    v2:(enum1,enum2);
BEGIN
    .....
END.
    
```

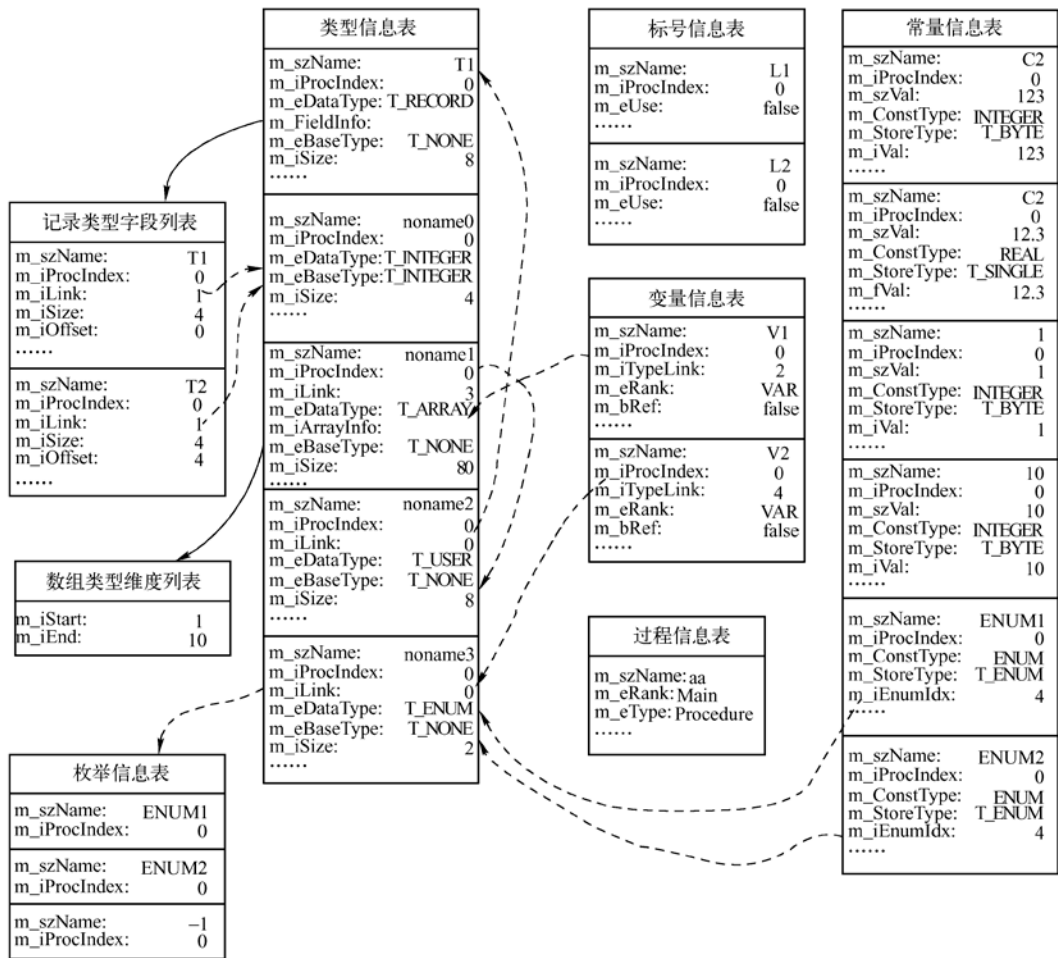


图 4-4 符号表实例示意图

在图 4-4 中，用虚线箭头表示位序编号方式的指针，即箭头起点的位序编号指向箭头终点的表项。实线箭头表示箭头终点的符号表隶属于箭头起点的表项。为了突出重点，图 4-4 省略了 `m_iProcIndex` 与过程信息表的虚线箭头。

读者可能有疑问，枚举类型的枚举常量一共才两个，为什么在枚举信息表中有三个表项呢？实际上，最后一个表项是结束标记。类型信息表的 `m_iLink` 仅指出了枚举值在枚举信息表中的起始位置，却没有记录结束位置，这可能会给编译器处理枚举类型带来不便，因此，在枚举信息表中，笔者为每个枚举类型设置一个结束标记。这样，从类型信息表的 `m_iLink` 所指向的位置起直到结束标记之间的枚举值就是该枚举类型的所有枚举常量。

通过以上的分析，读者对 Neo Pascal 符号表结构应该有了感性的认识了。囿于篇幅，无法详细举例说明符号表的各种情况。如果尚有不明之处，读者可以直接调试 Neo Pascal 源代码观察符号表，这是最简单且直观的方法。

4.3 声明部分的实现

从本章开始，读者将大量接触 Neo Pascal 的源代码。在分析源代码之前，笔者想与读者分享一点个人体会。在阅读了一些编译器方面的书籍后，发现语义分析与中间代码生成一直不是国内编译原理教材的讨论重点，不少教材往往轻描淡写或是泛泛而谈，令初学者找不到头绪。当然，笔者不可否认一点，详细阐述语义分析与中间代码生成构造并不容易。它是由一组庞大且耦合的语义子程序构成的，给编写教材带来了困难。因此，笔者相信阅读与分析实际编译器的源代码可能是最佳的学习方法。当然，阅读源代码可能会比较痛苦，笔者对此也深有体会。不过，想深入学习系统软件（操作系统、编译器、网络协议等）的读者可能不得不在绝望中寻找希望。与其他教材上的伪代码片段相比较，本书的优势在于 Neo Pascal 是一个完整的可运行、调试的编译器。读者结合本书的分析阅读源代码一定能事半功倍。

Neo Pascal 语义分析器是由 100 多个语义子程序构成，而语义子程序的调用完全由语法分析器驱动，即语法制导。根据完成的功能不同，语义子程序通常分为两个部分：符号表管理、IR 生成。其中，IR 生成又可以细分为语句生成和表达式生成。本小节主要分析符号表管理部分的源代码，而 IR 生成部分源代码将在第 5、6 章中详述。整个语义分析器的源代码规模在 3000 行左右，对于不少读者而言，直接给出完整的源代码及注释可能意义并不大，相信能一气呵成读完 3000 行枯燥代码的读者不会太多。因此，笔者将语法结构（即文法）进行适当分类，然后，详细分析该结构相关的语义子程序。不过，这是一项艰难的工作，因为语法结构并不存在明显的界线，可能有时并不是特别精确，请读者谅解。

4.3.1 相关数据结构

前面已经详细讲述了符号表的相关结构与设计。符号表是整个编译器的中心数据库，其重要性不言而喻。实际上，除了符号表以外，语义分析器还涉及一些全局数据结构。这些数据结构主要是辅助语义分析器完成相关的分析功能的。其中，大部分是栈结构，这主要是由于 Neo Pascal 是基于自上而下的语法分析方法构造语法分析器的。下面，就来谈谈相关的数据结构。

首先，最重要的数据结构就是符号表。下面就给出符号表完整声明。



程序 4-1 SymbolTbl.h

```
1 class CSymbolTbl
2 {
3 public:
4     vector<ProcInfo> ProcInfoTbl;           //过程信息表
5     stack<int> ProcStack;                   //过程分析栈
6     map<int,LabelInfo> LabelInfoTbl;       //标号信息表
7     map<int,ConstInfo> ConstInfoTbl;      //常量信息表
8     map<int,TypeInfo> TypeInfoTbl;        //类型信息表
9     vector<EnumInfo> EnumInfoTbl;        //枚举类型信息表
10    map<int,VarInfo> VarInfoTbl;           //变量信息表
11    vector<ProcDefToken> ProcDefTokenTbl; //过程原型单词流表
12    vector<TypeSysInfo> TypeSysTbl;       //类型系统表
13    vector<UseFile> UseFileTbl;          //包含文件信息表
14    vector<AsmPara> AsmParaTbl;          //汇编参数信息表
15 public:
16     //默认构造函数。
17     CSymbolTbl(void);
18     //检索标号信息表函数。检索成功，返回标号信息所在的表项位序号。检索失败，返回-1。
19     int SearchLabelInfoTbl(int iProcIndex,string sName);
20     //检索常量信息表函数。检索成功，返回常量信息所在的表项位序号。检索失败，返回-1。
21     int SearchConstInfoTbl(int iProcIndex,string sName);
22     //检索类型信息表函数。检索成功，返回类型信息所在的表项位序号。检索失败，返回-1。
23     int SearchTypeInfoTbl(int iProcIndex,string sName,bool bState=true);
24     //检索枚举类型信息表函数。检索成功，返回枚举信息所在的表项位序号。检索失败，返回-1。
25     int SearchEnumInfoTbl(int iProcIndex,string sName);
26     //检索过程信息表函数。检索成功，返回过程信息所在的表项位序号。检索失败，返回-1。
27     int SearchProcInfoTbl(string szName,bool bIsForward=true);
28     //检索类型系统表函数。检索成功，返回类型系统信息所在的表项位序号。检索失败，返回-1。
29     int SearchTypeSysTbl(int iOp,int iOp1Type,int iOp2Type=0);
30     //检索变量信息表函数。检索成功，返回变量信息所在的表项位序号。检索失败，返回-1。
31     int SearchVarInfoTbl(int iProcIndex,string szName);
32     //检索包含文件信息表函数。检索成功，返回包含文件信息所在的表项位序号。检索失败，返回-1。
33     int SearchUseFileTbl(string szName);
34     //判断两个过程原型单词流是否完全一致。
35     bool ProcDefTokenTblCompare(vector<CToken> List1,vector<CToken> List2);
36     //注册常量函数。将一个常量登记加入常量信息表。（重载形式）
37     int RecConstTbl(const string szValue,int iType);
38     //注册常量函数。将一个常量登记加入常量信息表。（重载形式）
39     int RecConstTbl(const ConstInfo Value);
40     //注册集合常量函数。将一个集合常量登记加入常量信息表。
41     int RecConstSetTbl(const string szValue);
42     //计算类型占用存储空间大小。
43     int CalcTypeSize(int iPos);
44     //检查指针声明是否合法。
45     bool PtrCheck(int &iPos);
```

```
46 //申请一个类型为 eStoreType 的临时变量。返回临时变量在变量信息表中的位序号。（重载形式）
47 int GetTmpVar(int iProcIndex,StoreType eStoreType);
48 //申请一个临时变量。返回临时变量在变量信息表中的位序号。（重载形式）
49 int GetTmpVar(int iProcIndex);
50 //申请一个临时变量。返回临时变量在变量信息表中的位序号。（重载形式）
51 int GetTmpVar(int iProcIndex,OpType eOpType);
52 //复制一个变量，作为临时变量。返回临时变量在变量信息表中的位序号。（重载形式）
53 int CopyTmpVar(int iValIdx);
54 //获取类型描述的实际类型。即略去类型链中的 T_USER 类型元。
55 int GetRealType(int iTypeLink);
56 //申请一个临时标号。
57 int GetTmpLabel(int iProcIndex);
58 //判断变量是否为形参变量。
59 bool IsVarPara(string szName,int iProcIndex);
60 //判断变量是否为临时变量。
61 bool IsTmpVar(int iPose);
62 //打印 IR 列表。
63 void PrintIR();
64 //判断变量的类型是否为过程。
65 bool IsProcVar(string szName);
66 //打印基本块信息。
67 void PrintBasicBlock();
68 //清空符号表。
69 void Clear();
70 //添加标号。
71 void AddLabel(LabelInfo Tmp);
72 //添加常量。
73 void AddConst(ConstInfo Tmp);
74 //添加类型。
75 void AddType(TypeInfo Tmp);
76 //添加变量。
77 void AddVar(VarInfo Tmp);
78 //默认析构函数。
79 ~CSymbolTbl(void);
80 };
```

CSymbolTbl 的数据成员也就是 Neo Pascal 的符号表，而其中的成员函数主要的作用就是维护符号表及处理符号，在此，读者先不必深究其实现细节。

4.3.2 主程序首部声明

主程序首部声明是比较简单的。在处理这部分声明时，编译器需要收集的信息仅仅是主程序名而已，也就是 program 关键字后面的一个标识符信息。同时，semantic001 也是第一个被调用的语义子程序，一些初始化工作也是由其完成的。至于其后的“(标识符列表)”部分在 Neo Pascal 中并没有实际的语义，仅仅是为了兼容标准 Pascal 文法而设置的。因此，并不需要专门的语义子程序来分析处理。



程序 4-2 semantic.cpp

相关文法:

程序头 → program 标识符 001 (标识符列表);

```
1  bool semantic001()
2  {
3      semantic000();
4      ProcInfo Tmp;
5      Tmp.m_szName=TokenList.at(iListPos-1).m_szContent;
6      Tmp.m_eRank=ProcInfo::Rank::Main;
7      Tmp.m_eType=ProcInfo::Type::Procedure;
8      Tmp.m_ParaTbl.clear();
9      SymbolTbl.ProcInfoTbl.push_back(Tmp);
10     SymbolTbl.ProcStack.push(SymbolTbl.ProcInfoTbl.size()-1);
11     iIdListFlag.push(0);
12     iTypeFlag.push(0);
13     return true;
14 }
```

第 3 行: 调用 semantic000 函数, 进行初始化。主要涉及符号表、辅助全局变量等。

第 5~9 行: 获取主程序名。在获取主程序名后, 生成过程信息表项, 并加入过程信息表中。

第 10 行: 将当前过程指针压入 ProcStack 栈。ProcStack 的栈顶元素就是用于标识当前正在分析的过程。设置 ProcStack 栈的主要意图就是解决过程嵌套声明的问题。与 C 语言不同的是 Pascal 允许嵌套声明过程。虽然 Neo Pascal 并不支持过程嵌套声明, 但是主程序中嵌套子过程的形式还是需要兼容的。

第 11~12 行主要完成标志处理, 将在后续章节中详解, 这里可以先略过。

4.3.3 包含文件声明部分

结构化、模块化程序设计的思想最早就是由 Pascal 的创始人 Wirth 教授提出来的, 因此, 在多文件编译方面, Pascal 是有一定优势的。实际上, Pascal 的 USES 关键字与 C 语言的#include 从功能上讲是比较类似的。不过, 从编译器实现的角度而言, 却存在一定的差别。

单文件源程序的时代已经渐渐远去了, 对于现代编译器而言, 处理多文件源程序是一个必备的能力。这里, 笔者觉得有必要简单谈谈多文件编译的相关内容。读者在编写多文件程序时, 可能并没有仔细考虑过多文件编译的内核。多文件编译的方法很多, 最常见的模式有以下两类: 合并后编译、编译后合并。

所谓“合并后编译”, 就是在编译前或编译过程中某一适当的时刻, 将多个文件按需求合并为一个源文件, 然后再由编译器编译此文件。此时, 多文件形式对于编译过程是完全透明的。读者熟悉的编译预处理就是如此。在正式编译之前, 先由预处理器将包含的头文件直接合并入某一主文件中。然后, 再由编译器进行统一编译。值得注意的

是，这里仅讨论编译预处理，并不表示 C 编译器默认就是这样完成多文件编译的。当然，程序员可以通过一些特殊的编程技巧，使 C 编译器使用“合并后编译”的方式进行编译。如果程序员在工程项目中只包含一个主程序文件，而使用 `#include` 方式将其他的 C 文件（不是头文件）包含入主程序文件，在这种情况下，C 编译器就会在编译预处理时将所有的 C 文件合并到主程序文件中，再进行统一编译。编译预处理器是一种比较有效的处理手段，不过，并不是所有编译器都拥有预处理器的。大多数 Pascal 编译器（包括 Neo Pascal）都没有真正意义上的预处理器，因此，源文件合并的工作通常是由编译器完成的。

所谓“编译后合并”，以 Windows 环境为例，就是将多个源程序分别编译成汇编文件，再由汇编器生成相应的 `obj` 文件，最后，由链接器将多个 `obj` 文件链接成可执行文件。与前者相比，这种方式的主要优点在于比较容易实现部分编译的需求。虽然这种方式也存在一些缺点，例如，某些源程序的错误可能会被隐藏，直到链接阶段才能发现，但对结构化思想而言，这种方式的提出可能是一种质的飞跃。最终，这种方式就演化形成了库文件的概念。由于库函数几乎不可能以源码形式公开，这种方式是必然之选。库文件的源程序通常经编译、汇编后生成 `obj` 文件，然后用户程序根据需要选择使用。随着 `obj` 数量的增加，人们开始考虑着手按功能将 `obj` 文件进行归类。实际上，就是将 `obj` 文件按一定格式组织合并成一个文件，也就是常见的 `lib` 文件。大多数 C 编译器都是采用这种方式进行编译生成的，虽然它们也支持编译预处理。这种方式的优点明显，但是对语言的语法结构是有一定要求的，它要求语言本身支持编写库。虽然 Pascal 的模块化设计思想堪称经典，但笔者不得不承认其在模块化方面做得不够。避开某些细枝末节不谈，就标准 Pascal 本身而言，它并不支持编写库文件，这可能就是其先天的不足之处。同时，也是 Pascal 备受争议的重要原因之一。当然，一些商用 Pascal 编译器扩展了相关的语法结构，最终实现了这一功能。

虽然 Neo Pascal 支持多文件编译，但也有一定的限制。Neo Pascal 只允许用户在非主文件（不是主程序所在的程序文件）中声明过程、函数，不允许声明全局常量、变量等。这与 Delphi 等商用编译器有所不同。

关于多文件编译的理论先谈到这里，下面详细分析一下 Neo Pascal 相关实现的细节。前面，已经了解了 Neo Pascal 的多文件编译主要是由编译器进行动态合并后进行统一编译的。既然要在编译过程中完成多文件合并，那么就必须将其视为一种语法结构，否则编译器将无法处理。因此，在 Neo Pascal 语法体系（BNF）中，存在 USES 声明的描述，而在 C 语言标准文法中并没有关于 `#include` 预处理指令的相关描述。

在 Pascal 语言中，USES 通常只是用于说明包含文件的列表，但它并不是合并文件的插入点，这与 `#include` 指令也是不同的。原因非常简单，如果在 USES 声明处插入文件的实体内容（过程、函数），那么，合并后的程序结构并不是一个合法的 Pascal 程序。而合法的插入点应该位于过程函数声明部分之前。实际上，语义子程序在分析 USES 声明结构时，只需记录包含文件的列表，以备后用即可。



程序 4-3 semantic.cpp

相关文法:

包含文件说明 → uses 093 标识符列表 094;

```
1 bool semantic093()
2 {
3     iIdListFlag.push(7);
4     return true;
5 }
6 bool semantic094()
7 {
8     iIdListFlag.pop();
9     return true;
10 }
```

以上产生式中只有 semantic093、semantic094 两个语义子程序，而且从这两个语义子程序来看，除了对 iIdListFlag 栈进行了一次入栈、出栈的操作外，根本没有完成其他任何语义动作。这里，先不必关注 iIdListFlag 栈的作用，稍后会作详细介绍。读者不难发现，产生式中的“标识符列表”是一个非终结符。也就是说在推导分析的过程中，如果“标识符列表”推导得到的产生式右部含有语义子程序，它们同样会被编译器执行。从 Neo Pascal 文法来说，“标识符列表”仅有的候选产生式如下：

【文法 4-2】

标识符列表 → 标识符 012 标识符列表 1

那么，随着推导过程的深入进行，编译器一定会执行 semantic012 语义子程序。同样地，由于“标识符列表 1”也是一个非终结符，它的相关语义子程序也会被调用。“标识符列表 1”的候选式有如下两个：

【文法 4-3】

标识符列表 1 → , 标识符 012 标识符列表 1
 → ε

如果在实际语法分析过程中，选择第一个产生式推导“标识符列表 1”，则其中的 semantic012 将会被调用。而如果选择第二个产生式推导“标识符列表 1”，由于其中没有包含任何语义子程序，编译器将不作任何调用。例如，编译器在分析 USES file1,file2;声明结构时，其语义子程序调用顺序必定是 semantic093→ semantic012 → semantic013 → semantic014。这里，笔者不打算详细分析产生该调用顺序的过程了，这是一个非常简单的语法分析问题，读者可以自行参考自上而下的语法分析的特点。下面，就来看看 semantic012 相关源代码及其与 semantic093、semantic094 的关系。

程序 4-4 semantic.cpp

相关文法:

标识符列表 → 标识符 012 标识符列表 1

标识符列表 1 → , 标识符 012 标识符列表 1

```
1 bool semantic012()
2 {
3     .....
4     if (iIdListFlag.top()==7)           //判断是否正在处理包含文件声明，即判断标志栈
5     {
6         string Tmp=TokenList.at(iListPos-1).m_szContent;
7         if (SymbolTbl.SearchUseFileTbl(Tmp)==-1)
8         {
9             SymbolTbl.UseFileTbl.push_back(UseFile(Tmp,false));
10            return true;
11        }
12        else
13        {
14            EmitError(Tmp.append("模块已经包含"),TokenList.at(iListPos-1));
15            return false;
16        }
17    }
18    return true;
19 }
```

上述的 semantic012 并不完整，这里仅列出了与 USES 声明相关的源代码。从整个程序的结构来说，读者不难发现它是以 iIdListFlag 栈顶元素作为分支条件。因此，笔者有必要对 iIdListFlag 栈作简单介绍。实际上，这与非终结符“标识符列表”有关。仔细观察 Neo Pascal 的文法，不难发现“标识符列表”这个非终结符在产生式右部多次出现。例如：

【文法 4-4】

变量定义 → 013 标识符列表 040: 类型

枚举类型 → (011 标识符列表)014

.....

在选用上述候选式推导时，根据语法制导的特点，编译器同样会去调用 semantic012 语义子程序。显然，此时 semantic012 的语义动作与其在“包含文件说明→ uses 093 标识符列表 094;”候选式中的语义动作是完全不同的。实际上，这个问题的矛盾主要是源于非终结符的复用，导致某些语义子程序可能同时需要处理若干个不同的语义情景。当然，要解决这个问题并不复杂，主要有两种处理方式：

(1) 避免非终结符的复用。虽然这样可能会导致文法冗余，却是一个有效且可行的办法。实际上，在一个完整的语言文法中，非终结符的复用并不会太频繁。因此，导致的文法冗余的规模是可以接受的。

(2) 设置标志辅助分析。实际上，在递归下降语法分析中，编译器可以借助于传参



方式实现标志管理，这是一种常见且高效的方法。然而，本书采用的 LL(1)是一种非递归的语法分析方法，因此，标志的管理必须借助于用户栈结构实现。这种方式不需要修改文法，只需在复用非终结符的候选式中加入维护标志栈的语义动作即可。而被复用的非终结符的语义子程序则根据标志栈的栈顶元素判断执行相应的语义动作。而 `iIdListFlag` 就是一个标志栈。

下面简单分析一下 `semantic012` 用于处理包含文件声明的语义动作。

第 7 行：为了保证模块文件不被重复引用，在处理包含文件声明时，必须加以判断。`SearchUseFileTbl` 是一个简单的查找函数。

第 9 行：保存包含文件名，以备后用。严格地说，包含文件信息表 (`UseFileTbl`) 并不是一张符号表，只是一个辅助数据结构。它的结构非常简单，声明如下：

【声明 4-10】

```
struct UseFile
{
    string m_szFileName;           //文件名
    bool m_bFlag;                 //标志
    UseFile(string szFileName,bool Flag); //构造函数
};
vector<UseFile> UseFileTbl;      //包含文件信息表
```

这里简单解释一下 `m_bFlag` 标志。实际上，编译器在处理包含文件时，有时难以避免模块文件的循环引用或重复引用的情况，这可能会导致非常严重的后果。因此，编译器通常会设计一个简单的数据结构来避免类似情况发生。虽然编译器的判断可能与用户的设计初衷相悖，但也是无可奈何的。在文件合并时，编译器将利用 `m_bFlag` 标志避免上述情况发生。

4.3.4 标号声明部分

分析标号声明并不复杂，编译器的主要工作就是将标号声明部分中的标号信息逐一加入 `LabelInfoTbl` 中。

程序 4-5 semantic.cpp

相关文法：

标号声明部分 → label 标识符 003 标号声明列表；

```
1 bool semantic003()
2 {
3     LabelInfo Tmp;
4     Tmp.m_szName=TokenList.at(iListPos-1).m_szContent;
5     Tmp.m_bDef=false;
6     Tmp.m_bUse=false;
7     Tmp.m_iProcIndex=SymbolTbl.ProcStack.top();
8     if ( SymbolTbl.SearchLabelInfoTbl(Tmp.m_iProcIndex,Tmp.m_szName)==-1 &&
9         SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack
10            .top()).m_szName.compare(Tmp.m_szName)!=0)
11     {
```



```

12         SymbolTbl.LabelInfoTbl.push_back(Tmp);
13         return true;
14     }
15     else
16     {
17         EmitError(Tmp.m_szName.append("标识符已经存在"),TokenList.at(iListPos-1));
18         return false;
19     }
20 }

```

第 4~7 行：生成一个 `LabelInfo` 类型的标号信息，并按实际情况填写。

第 8~10 行：用于判断当前标号的名字是否已被占用。这里，程序仅对 `LabelInfoTbl` 表进行了搜索。读者可能疑惑为什么不需要搜索其他符号表呢？笔者曾经提到，说明部分依次包含程序的单元声明、标号声明、常量声明、类型声明、变量声明、函数或过程声明。`Pascal` 语言允许缺少其中某些声明，但不允许改变它们的次序。从 `Pascal` 文法角度来看，这就更显而易见了。在分析标号信息时，编译器根本没有分析到其他任何声明，所以并不需要搜索其他符号表。有些读者可能还会疑惑，根据 `Neo Pascal` 文法，`semantic003` 不但需要处理主程序的标号信息，同时还需要处理子程序（即过程或函数）的标号信息。那么，当处理子程序的标号信息时，是否需要考虑该标号名字与主程序各标识符之间的重名问题呢？显然不必考虑，在 `Pascal` 语言中，子程序的名字空间与所属主程序的名字空间是相对独立的。当子程序与主程序的标识符重名时，对于子程序而言，此时主程序中声明的标识符将是不可见的。在这种情况下，标识符对象的作用域与可见域就不完整重叠了。当然，除此之外，标号的名字也不能与所隶属的过程、函数名冲突。

4.3.5 常量声明部分

常量声明部分的语义处理主要完成常量信息表的登记。前面已经讲过，词法分析器可以识别得到字面常量值，并将其登记入常量信息表。不过，词法分析器却无法获得常量的符号信息。维护、补充常量信息的工作就必须由语义处理来完成。常量声明的相关文法如下：

【文法 4-5】

```

常量声明部分 → const 常量定义；常量声明列表
常量声明列表 → 常量定义；常量声明列表
                → ε
常量定义     → 标识符 = 常量 004

```

其中只包含了一个语义子程序 `semantic004`。

程序 4-6 semantic.cpp

相关文法：

```
常量定义     → 标识符 = 常量 004
```

```

1 bool semantic004()
2 {
3     string szID=TokenList.at(iListPos-3).m_szContent;

```




```
4   if ( SymbolTbl.SearchLabelInfoTbl(SymbolTbl.ProcStack.top(),szID)==-1 &&
5       SymbolTbl.SearchConstInfoTbl(SymbolTbl.ProcStack.top(),szID)==-1 &&
6       SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_szName.compare(szID)!=0)
7   {
8       ConstInfo *pTmp;
9       pTmp=&SymbolTbl.ConstInfoTbl.at(atoi(TokenList
10          .at(iListPos-1).m_szContent.c_str()));
11       pTmp->m_szName=szID;
12       pTmp->m_iProcIndex=SymbolTbl.ProcStack.top();
13       return true;
14   }
15   else
16   {
17       EmitError(szID.append("标识符已经存在"),TokenList.at(iListPos-3));
18       return false;
19   }
20 }
```

第 3 行：获取常量的名字标识符。根据 semantic004 在候选式中的位置，必须向前搜索 3 个单词，才能获得常量的名字标识符。根据语法制导的特点，这样的搜索绝对是安全的，所以不必作任何判断。

第 4~6 行：判断标识符是否已被占用。这里的判定条件与标号声明的判定条件有一定区别。由于文法定义顺序的原因，对于同一级别的过程而言，编译器将在分析完标号声明部分之后，再分析常量声明部分。因此，这里必须增加对标号信息表的重名判断。

第 9 行：申请一个临时指针指向常量信息表中的相应表项。在讨论词法分析器时，笔者介绍过词法分析器是如何处理字面常量的。首先，将常量值记录在常量信息表中。然后，利用单词的 `m_szContent` 属性将该常量值在常量信息表中的位序号传递给后续阶段。这里，就是利用单词的 `m_szContent` 属性获取常量信息表中的相应表项，便于更新相关属性。

第 11 行：更新该常量表项的名字信息。

第 12 行：更新该常量表项的所属过程信息。不仅在 Pascal 语言中，大多数程序设计语言都是如此，常量与变量类似，也存在作用域、可见域等。

4.3.6 类型声明部分

本小节将讨论类型声明部分的语义分析。前面，笔者已经从符号表结构的角度阐述了类型描述的相关问题，读者可能已经感觉有些复杂。确实如此，在编译器设计中，只要和类型扯上关系，通常许多问题可能就變得比较复杂。下面将分析类型声明部分语义分及其实现。

实际上，本小节的讨论目的就是构造一组语义子程序，使之根据输入源程序自动完成类型信息表及其相关符号表的登记、维护工作。当然，这个过程是稍显繁杂的。

【文法 4-6】

类型声明部分 → type 类型定义 ; 类型定义列表

类型定义列表 → 类型定义 ; 类型定义列表

→ ϵ
类型定义 → 标识符 006 = 类型

以上是类型声明部分的文法，其中“类型”非终结符是最为复杂的，稍后将详细分类讲述。这里，先来看看 semantic006 的功能。semantic006 的主要功能就是生成类型信息表项，并填写类型名字等属性。

程序 4-7 semantic.cpp

相关文法：

类型定义 → 标识符 006 = 类型

```
1 bool semantic006()
2 {
3     TypeInfo Tmp;
4     Tmp.m_szName=TokenList.at(iListPos-1).m_szContent;
5     Tmp.m_iProcIndex=SymbolTbl.ProcStack.top();
6     if (SymbolTbl.SearchTypeInfoTbl(SymbolTbl.ProcStack.top(),Tmp.m_szName)==-1 &&
7         SymbolTbl.SearchConstInfoTbl(SymbolTbl.ProcStack.top(),Tmp.m_szName)==-1 &&
8         SymbolTbl.SearchLabelInfoTbl(SymbolTbl.ProcStack.top(),Tmp.m_szName)==-1 &&
9         SymbolTbl.SearchEnumInfoTbl(SymbolTbl.ProcStack.top(),Tmp.m_szName)==-1 &&
10        SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_szName.compare(Tmp.m_szName)!=0)
11     {
12         SymbolTbl.TypeInfoTbl.push_back(Tmp);
13         iTypePos.push(SymbolTbl.TypeInfoTbl.size()-1);
14         return true;
15     }
16     else
17     {
18         EmitError(Tmp.m_szName.append("标识符已经存在"),TokenList.at(iListPos-1));
19         return false;
20     }
21 }
22
```

第 6~10 行：这个条件判断就比较复杂了，除了常量信息表、标号信息表、类型信息表之外，类型的名字还可能与枚举常量冲突，因此，必须加以考虑。

第 13 行：将指向当前类型表项的指针压入 iTypePos 栈。iTypePos 栈的元素类型就是普通的整型，而该元素指向的类型信息表项就是当前正在分析的类型信息。那么，为什么需要使用 iTypePos 栈呢？下面，笔者就对这个问题稍作解释。

虽然 semantic006 填写了一部分类型信息表项的属性，并将其加入了类型信息表，但是，这些信息是远远不够的，需要后续语义分析子程序加以补充。读者可能会疑惑，为什么不采用类似 semantic004 的方式，向前搜索多个单词以获取类型名字呢？实际上，这里所面临的问题是不同的，候选式“常量定义 → 标识符 = 常量”中包含的“常量”虽是非终结符，但是它推导得到的单词数量是固定的。而“类型定义 → 标识符 006 = 类型”中非终结符“类型”所能推导得到的单词数量是无法确定的。因此，试图在分析完毕后回填名字等



信息将是比较复杂的。

然而，应用栈结构描述的意图是显而易见的。当然，就是考虑到类型嵌套声明的原因，这是一个很现实的问题。通常，处理这种嵌套问题的最好办法就是递归，但是，LL(1)分析法恰恰将递归的问题转化为非递归的形式了。因此，原本可以借助于递归传参的属性只能通过用户栈来模拟实现了。

下面，将着眼于类型相关的语义分析，这是一个充满挑战的话题。与其他程序设计语言相比，Pascal 的类型系统还不算特别复杂，但足以阐述类型描述的相关问题。

1. 基本类型

基本类型的语义分析比较简单，基本不需要处理复杂的类型链。下面，先来看看基本类型相关的候选式。

【文法 4-7】

```
类型      → 015 基本类型 010
基本类型 → 有序类型 | 无序类型
有序类型 → char 007 | boolean 007 | 整数类型 | 枚举类型
无序类型 → real 007 | single 007
整数类型 → integer 007 | byte 007 | shortint 007 | smallint 007 | word 007 | longword 007 |
           cardinal 007
枚举类型 → (011 标识符列表 )014
```

这里，semantic015、semantic010 与基本类型的语义分析无关，先不作考虑。这两个语义子程序主要用于处理一些复杂类型的分析。

而 semantic007 涉及的语境比较复杂，它不但需要处理基本类型声明，还需要处理许多复杂的语义。这里，笔者只剖析其中的部分代码，更多的复杂处理，待后续章节详述。

程序 4-8 semantic.cpp

相关文法：

```
有序类型 → char 007 | boolean 007 | 整数类型 | 枚举类型
无序类型 → real 007 | single 007
整数类型 → integer 007 | byte 007 | shortint 007 | smallint 007 | word 007 | longword 007 | cardinal 007
```

```
1  bool semantic007()
2  {
3      .....
4      if (!iTypePos.empty() && iTypePos.top() != -1 &&
5          SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_eDataType == StoreType::T_NONE)
6      {
7          StoreType TmpStoreType = CType::TokenToEnum(TokenList.at(iListPos-1).m_iKind);
8          SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_eDataType = TmpStoreType;
9          SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_eBaseType = TmpStoreType;
10         SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_iState = 1;
11     }
12     return true;
13 }
```

第 4 行：判断 `iTypePos` 栈是否为空。

第 7 行：`TokenToEnum` 是类型系统的一个静态成员函数，它的功能是根据单词值返回实际的类型枚举值，即 `StoreType` 的值。详细源代码稍后给出。

第 8~10 行：基本类型的信息非常简单，只需填写 `m_eDataType` 及 `m_eBaseType` 两个属性即可。注意，完成后必须将 `m_iState` 状态标志置为 1，表示这个类型表项已经分析完毕，可以对其引用了。这个标志非常重要，它可以控制类型信息表中哪些类型表项可以被引用，而哪些则不可以。这里的引用，不仅包括变量对其引用，还包括其类型内部对其引用。

下面，再来看看 `TokenToEnum` 函数的源代码。

程序 4-9 Type.cpp

```
1 StoreType CType::TokenToEnum(int iToken)
2 {
3     StoreType tmp;
4     switch (iToken)
5     {
6         case 49:tmp=StoreType::T_CHAR;           break;
7         case 44:tmp=StoreType::T_BOOLEAN;       break;
8         case 64:tmp=StoreType::T_INTEGER;       break;
9         case 45:tmp=StoreType::T_BYTE;          break;
10        case 79:tmp=StoreType::T_SHORTINT;      break;
11        case 82:tmp=StoreType::T_SMALLINT;     break;
12        case 92:tmp=StoreType::T_WORD;         break;
13        case 66:tmp=StoreType::T_LONGWORD;     break;
14        case 47:tmp=StoreType::T_CARDINAL;     break;
15        case 74:tmp=StoreType::T_REAL;         break;
16        case 81:tmp=StoreType::T_SINGLE;       break;
17        case 83:tmp=StoreType::T_STRING;       break;
18    }
19    return tmp;
20 }
```

前面，笔者已经详细分析了整型、实型、字符型、布尔型的相关语义子程序，它们并不是特别复杂。下面，讨论另一种基本类型——枚举类型。与枚举类型有关的候选式如下：

【文法 4-8】

枚举类型 → (011 标识符列表) 014

较之先前讨论的基本类型，枚举类型可能稍复杂。枚举类型的语义处理不但涉及类型信息表，还需要填写枚举信息表。虽然枚举类型是一种基本类型，但是，从编译器设计的角度而言，它的语义处理可能更接近于复杂类型。仔细分析该候选式，不难发现，候选式中主要是由一对终结符括号、两个语义子程序和一个非终结符“标识符列表”组成。关于非终结符“标识符列表”，曾经在 4.3.3 节中简单介绍过，它的相关产生式包含了一个语义子程序——`semantic012`。因此，枚举类型的语义处理主要就是由 `semantic011`、`semantic012`、`semantic014` 三个语义子程序共同完成的。下面，就来看看相关源码的实现。



程序 4-10 semantic.cpp

相关文法:

枚举类型 → (011 标识符列表) 014

```
1  bool semantic011()
2  {
3      iEnumSize= SymbolTbl.TypeInfoTbl.size();
4      if (iTypeFlag.top()==2)
5      {
6          SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_iLink=iEnumSize;
7          SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_iState=1;
8          TypeInfo Tmp;
9          Tmp.m_iState=1;
10         Tmp.m_szName="_noname";
11         Tmp.m_szName.append(GetSerialId());
12         Tmp.m_iProcIndex=SymbolTbl.ProcStack.top();
13         Tmp.m_eDataType=StoreType::T_ENUM;
14         Tmp.m_iLink=SymbolTbl.EnumInfoTbl.size();
15         iIdListFlag.push(1);
16         SymbolTbl.TypeInfoTbl.push_back(Tmp);
17         return true;
18     }
19     SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_eDataType=StoreType::T_ENUM;
20     SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_iLink=iEnumSize;
21     iIdListFlag.push(1);
22     return true;
23 }
```

第 3 行: `iEnumSize` 是一个全局变量, 用于记录枚举信息表的表长。由于枚举信息表是一张全局符号表, 各种枚举类型的枚举常量信息统一顺序存储在枚举信息表内, 因此, 在分析枚举常量列表之前, 保存表长的值也就是记录当前类型的枚举常量列表在枚举信息表中的起始位置。其中, 两个要点有必要说明:

(1) 枚举类型声明内部是不允许嵌套其他类型声明的, 故并不需要考虑应用栈结构来保存起始位置。

(2) 保存起始位置的目的在于便于计算该枚举类型的枚举常量个数。由于 Pascal 规定枚举类型的常量不允许超过 256 个, 因此, 语义子程序应该加以判断。

第 6~17 行: 处理枚举类型作为集合类型的基类型。集合类型的类型链构造比较特殊, 是由基类型语义子程序处理, 而不是集合类型本身的语义子程序处理的。根据 Pascal 的特点, 集合类型的基类型只能是少数的有序类型及枚举类型, 因此, 为了便于设计, 笔者没有使用 `iTypePos` 栈加以处理。

第 20 行: 指向枚举信息表尾, 因为这个位置就是当前枚举类型的枚举常量列表的开始。

第 21 行: 前面已经提到枚举类型语义处理将涉及“标识符列表”相关语义子程序。然而, 在整个文法体系中, 由于非终结符“标识符列表”存在多处复用, 因此, `semantic012`

语义子程序只能依赖 `iIdListFlag` 标志栈，才能保证语义动作的正确执行。

程序 4-11 semantic.cpp

相关文法：

标识符列表 → 标识符 012 标识符列表 1

标识符列表 1 → , 标识符 012 标识符列表 1

```

1  bool semantic012()
2  {
3      if (iIdListFlag.top()==1)
4      {
5          EnumInfo Tmp;
6          Tmp.m_szName=TokenList.at(iListPos-1).m_szContent;
7          Tmp.m_iProcIndex=SymbolTbl.ProcStack.top();
8          if (SymbolTbl.SearchTypeInfoTbl(SymbolTbl.ProcStack.top(),Tmp.m_szName,false)==-1 &&
9              SymbolTbl.SearchConstInfoTbl(SymbolTbl.ProcStack.top(),Tmp.m_szName)==-1 &&
10             SymbolTbl.SearchLabelInfoTbl(SymbolTbl.ProcStack.top(),Tmp.m_szName)==-1 &&
11             SymbolTbl.SearchEnumInfoTbl(SymbolTbl.ProcStack.top(),Tmp.m_szName)==-1 &&
12             SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_szName.compare(Tmp.m_szName)!=0)
13             {
14                 if ((SymbolTbl.EnumInfoTbl.size()-iEnumSize)<=256)
15                 {
16                     SymbolTbl.EnumInfoTbl.push_back(Tmp);
17                     return true;
18                 }
19             }
20             else
21             {
22                 EmitError("枚举类型的标号个数不得大于 256",TokenList.at(iListPos-1));
23                 return false;
24             }
25             else
26             {
27                 EmitError("标识符名已经存在，或该标识符已定义",TokenList.at(iListPos-1));
28                 return false;
29             }
30         }
31         .....
32     }

```

第 3 行：判断 `iListFlag` 标志栈。当非终结符“标识符列表”被复用时，标志栈是 `semantic012` 得以正确处理各类语义的保证。

第 6、7 行：填写枚举信息表表项。在处理枚举常量列表时，需注意具体源语言关于枚举常量名字空间的约定，不同的语言可能存在一定的差异。

第 14 行：判断当前类型的枚举常量的个数是否已超过 256。枚举类型的枚举常量个



数不超过 256 是标准 Pascal 的规范，绝大多数 Pascal 编译器（包括实验室级、商用级）都遵守这一规范。正因如此，枚举类型变量的物理存储空间通常就是 1B。

程序 4-12 semantic.cpp

相关文法:

枚举类型 → (011 标识符列表) 014

```
1  bool semantic014()
2  {
3      EnumInfo Tmp;
4      int i=iEnumSize;
5      int j=0;
6      while (i<SymbolTbl.EnumInfoTbl.size())
7      {
8          ConstInfo TmpConst;
9          TmpConst.m_szName=SymbolTbl.EnumInfoTbl.at(i).m_szName;
10         TmpConst.m_iProcIndex=SymbolTbl.ProcStack.top();
11         TmpConst.m_iVal=j++;
12         TmpConst.m_iEnumIdx=SymbolTbl.TypeInfoTbl.size()-1;
13         TmpConst.m_ConstType=ConstType::ENUM;
14         TmpConst.m_StoreType=StoreType::T_ENUM;
15         SymbolTbl.ConstInfoTbl.push_back(TmpConst);
16         i++;
17     }
18     Tmp.m_szName="-1";
19     SymbolTbl.EnumInfoTbl.push_back(Tmp);
20     iIdListFlag.pop();
21     return true;
22 }
23
```

第 6~17 行：遍历类型的相关枚举常量的表项，并生成相应的常量信息。以枚举常量名作为常量信息表项的名字，以一个由编译器顺序分配的整型值作为常量的值，并加入常量信息表。这样做的优点是显而易见的，便于后续语义子程序的处理。因为后续语义子程序不必再关注枚举常量与普通常量之间的区别，只需检索常量信息表即可获得枚举常量的相关信息，包括实际的取值、枚举类型信息等。这里需说明两点：

- (1) 常量信息表的 `m_iEnumIdx` 属性用于指向该枚举常量所属类型的描述信息。
- (2) 在 Pascal 语言中，枚举常量的存储值是由编译器分配的，不允许用户指定。

第 18、19 行：设置结束标志，并将其加入枚举信息表。由于枚举信息表是顺序存储且全局共享的，所以编译器有必要记录各个枚举类型的枚举值列表在枚举信息表中的起始和结束位置。类型信息表的 `m_iLink` 属性已经记录了该类型在枚举信息表中的起始位置，但是编译器却始终无法得到结束位置。当然，在类型信息表中增加一个属性用于记录结束位置是完全可行的。不过，笔者并不主张这种修改方案。比较理想的处理方式就是在枚举信息表中设置结束标志，这样并不需要修改符号表的结构。在 Neo Pascal 中，将“-1”作为结束标志加

入枚举信息表内。这样，编译器可以方便地获得某一类型的枚举值列表，只需从类型的 `m_iLink` 属性开始顺序向下读取，直到“-1”结束标志终止即可。

至此，笔者已经分析了基本类型声明的相关语义处理。下面，将继续讲解数组类型声明及其语义处理。

2. 数组类型

前面，读者应该已经了解了 Pascal 数组类型的一些基本特点，以及与 C 或 Java 的不同之处。下面看看数组类型的相关候选式。

【文法 4-9】

类型	→ 015 构造类型 010
构造类型	→ 数组类型
数组类型	→ array 016 [数组界限 数组界限列表] of 类型 025
数组界限列表	→ , 数组界限 数组界限列表 ε
数组界限	→ 整数常量 017.. 整数常量 018

与基本类型语义处理不同，数组类型是允许类型嵌套声明的。从文法角度而言，不难发现，非终结符“类型”存在右递归现象。此时，编译器是无法预知递归实际深度的。下面看一个实例分析。

例 4-3 匿名类型声明。

【声明 4-11】

TYPE

```
a=ARRAY [1..20] OF INTEGER;
b=ARRAY [1..10] OF a;
```

声明 4-11 并不复杂，先不考虑语义子程序的实现。填写符号表、构造类型链应该并不困难。整个过程可以描述为三步：填写 a 的类型信息；填写 b 的类型信息；将 b 的 `m_iLink` 属性指向 a 的类型表项。参见图 4-5。

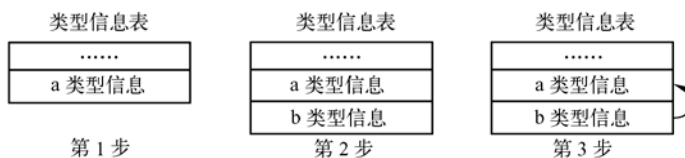


图 4-5 声明 4-10 的语义分析过程示意

以上过程虽然涉及类型链的构造，但是整个过程并不复杂。由于 b 类型的基类型 a 是一个实名类型，编译器只需检索类型信息表并钩链成类型链即可。右递归的处理似乎消失于无形之中了，然而问题并非如此简单。请读者分析如下声明：

【声明 4-12】

TYPE

```
b=ARRAY [1..10] OF ARRAY [1..20] OF INTEGER;
```

声明 4-12 的语义与声明 4-11 完全相同，只是声明形式略有不同。不过，这样的变化可能会给语义分析带来不少困难。通常，可以将整个分析处理过程描述为三步：先填写 a 的类

型信息；再生成一个匿名类型表项，填写相应的属性，并将其加入符号表；最后将 b 的 m_iLink 属性指向 a 的类型表项。请参见图 4-6。

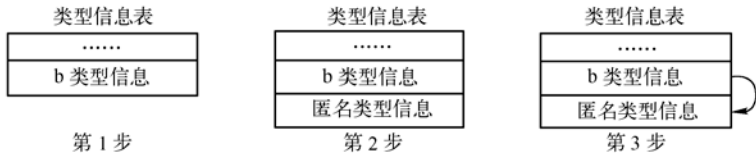


图 4-6 声明 4-2 的语义分析过程示意

以上过程，乍一看与先前的三个步骤比较类似，但是，其差别在于编译器主属类型在符号表中的位置是不同的。在类型复合声明中，两个类型结点之间通常是存在主从关系的。例如，在声明 4-11 中，b 类型就是主属类型，而在 b 类型的复合声明中，a 就是从属类型。值得注意的是，主从关系是针对某一特定的复合声明而言的，并不是基于整个源程序讨论的。通常，在处理某一类型声明时，编译器可能需要反复徘徊于主从类型之间进行分析。在记录类型声明的语义处理时，这种情况尤为突出。根据先前的符号表结构，编译器可以方便地从主属类型信息获取其从属类型的信息。不过，从某一个实际类型信息表项获取其主属类型的详细信息可能就不大容易了。形如声明 4-11，编译器很容易地确定主属类型，因为主属类型通常就是类型信息表的最末表项。而形如声明 4-12，编译器就无法简单地确定主属类型，因为编译器永远也无法预知主属类型表项后将存在多少从属类型的表项。当然，在语法上作严格限制，确实可以避免匿名类型的产生，同时也大大降低了编译器设计的复杂度。不过，这种处理方案并不是一个好主意。实际上，在很多情况下，人们更倾向于使用语言的匿名机制，包括匿名类型、匿名参数、匿名函数等，它们可以大大方便使用。

那么，可行的解决方法就是利用栈结构保存当前正在分析类型的指针（在 Neo Pascal 中也就是类型信息表项的位序号），可以将这个栈简称为“当前类型指针栈”。在 Neo Pascal 中，当前类型指针栈声明如下：

【声明 4-13】

```
stack<int> iTypePos;
```

栈顶元素即为当前类型的指针，而次元素即为当前类型的主属类型的指针，依此类推。这种解决方案的提出并不仅仅是为解决数组声明的问题，而是旨在应对一切复杂的类型复合声明。虽然语法分析器是非递归实现的，但是，推导文法的过程仍然是一个递归的过程。在递推过程中，语义分析器将类型的指针逐一压栈，而在回归过程中，则将类型的指针逐一恢复出栈。当正确完成一次推导后，iTypePos 栈应当仍然为空。

例 4-4 复合类型声明的处理。

【声明 4-14】

```
TYPE
    a=ARRAY [1..10] OF RECORD
        a: ARRAY [1..10] OF INTEGER;
        b: INTEGER;
    END;
```

本例是一个复合类型声明的实例。这里，笔者先不讨论详细的语义动作实现。只讨论编译器如何确定主属类型的问题，这是一个重要的话题，也是理解后续语义处理源代码的基础。

先前，读者应该已经基本理解了设置 `iTypePos` 的意图。下面，就来看看如何利用栈结构保存主属类型指针。请读者参见以下声明：

【声明 4-15】

TYPE

```
a= ↓ ARRAY [1..10] OF ↓ RECORD
      a: ↓ ARRAY [1..10] OF ↓ INTEGER ↑ ↑ ;
      b: ↓ INTEGER ↑ ;
END ↑ ↑ ;
```

声明 4-15 在声明 4-14 的基础增加了增加了一些箭头。注意，这些箭头只是为了便于笔者讲解而设置的一种标记，与指针或者其他的语法机制无关。假设“↓”表示将当前类型信息表的指针入栈，而“↑”表示将栈顶指针弹出。仅此两个语义动作，编译器就可以跟踪推导过程中的类型变化情况。详细的处理过程并不复杂，不再赘述。

当然，仅在源程序中加入入栈、出栈的动作，显然是不够的。下面来看如何在文法的适当位置安插语义子程序，使之在语法制导分析的过程中实现对栈的维护。当然，能使编译器正确完成语义分析工作是必须保证的前提。

要弄清楚源程序与文法之间的联系，归约的方法可能是最有效的，它比较适用于分析此类情况。本书并不打算详解归约的相关理论与技术。这里，仅以示意图形式描述其过程，参见图 4-7。

```
a= ↓ ARRAY [1..10] OF ↓ RECORD a: ↓ ARRAY [1..10] OF ↓ INTEGER ↑ ↑ ; b: ↓ INTEGER ↑ ; END ↑ ↑ ;
```

图 4-7 类型声明及相关语义动作

在图 4-7 中，笔者用五条下画线描述了整个归约过程，最终，每条下画线上的单词都将被归约为非终结符“类型”。而下画线的序号则表示归约的顺序。从归约的过程分析，可以得到如下结论：通常，在一组完整的类型声明中，将入栈动作插入声明首部，而将出栈动作插入尾部。换句话说，语义分析器就是在处理一组类型声明之初，执行入栈动作；在结束一组类型声明之际，执行出栈动作。实际上，这种栈操作方式与计算机中断现场保护非常类似。栈结构在计算机软、硬件中都有广泛应用。在后续章节中，还将涉及计算机硬件结构中的系统栈，它对程序设计语言的发展起到了至关重要的作用。

以上，笔者分析了复合类型语义处理的基本思想。从算法本身而言，可能并不是很复杂。有些读者可能会认为这些棘手问题的产生完全是由于自上而下的语法分析机造成的。关于这个问题，读者的看法并非全无道理。事实上，如果采用自下而上的语法分析机制的确可以回避此类问题。不过，基于归约实现的编译器同样会遇到一些新的问题，而这些问题可能是前者无需考虑的。实践证明，从设计语义分析器的角度而言，两种语法分析机制实现的语义分析器都不是绝对完美的。

下面，继续讨论数组声明的相关语义处理。先来看看 `semantic016` 与 `semantic025` 的源代



码实现，它们是“数组类型”的最外层语义子程序。

程序 4-13 semantic.cpp

相关文法：

数组类型 → array 016 [数组界限 数组界限列表] of 类型 025

```
1 bool semantic016()
2 {
3     SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_eDataType=StoreType::T_ARRAY;
4     iTypeFlag.push(3);
5     return true;
6 }
```

第 3 行：设置类型信息表项的 `m_eDataType` 属性，而 `iTypePos` 栈顶元素即为当前类型指针。

第 4 行：类型标志压栈，用于标识当前正在分析的类型是数组类型。数组、记录等类型的文法远比基本类型复杂，非终结符、产生式的复用的情况也较为常见。因此，编译器有必要进行标识，以便后续语义子程序处理。

程序 4-14 semantic.cpp

相关文法：

数组类型 → array 016 [数组界限 数组界限列表] of 类型 025

```
1 bool semantic025()
2 {
3     iTypeFlag.pop();
4     return true;
5 }
```

第 3 行：弹出类型标志，与 `semantic016` 的压栈动作是对应的。

下面，再来看看“数组界限”及其语义子程序（`semantic017`、`semantic018`）的实现。

程序 4-15 semantic.cpp

相关文法：

数组界限 → 整数常量 017.. 整数常量 018

```
1 bool semantic017()
2 {
3     ArrayInfo Tmp;
4     int i=atoi(TokenList.at(iListPos-1).m_szContent.c_str());
5     Tmp.m_iStart=atoi(SymbolTbl.ConstInfoTbl.at(i).m_szVal.c_str());
6     SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_ArrayInfo.push_back(Tmp);
7     return true;
8 }
```

第 4 行：获取整数常量的表项指针。在词法分析阶段，编译器已经将整数常量加入了常

量信息表中，并借助于单词的 `m_szContent` 属性将该常量信息表项位序号传递给了语法分析器及后续阶段。

第 5 行：从常量信息表中，获取数组的下限。

第 7 行：将数组维度信息表项加入符号表，便于 `semantic018` 填写上限（即 `m_iEnd` 属性）。注意，数组维度信息列表是一张局部符号表，结构也非常简单。

程序 4-16 semantic.cpp

相关文法：

数组界限 → 整数常量 017.. 整数常量 018

```
1 bool semantic018()
2 {
3     int i=atoi(TokenList.at(iListPos-1).m_szContent.c_str());
4     int iTmp=atoi(SymbolTbl.ConstInfoTbl.at(i).m_szVal.c_str());
5     ArrayInfo* tmp=&(SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_ArrayInfo.
6                     at(SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_ArrayInfo.size()-1));
7     if(tmp->m_iStart<=iTmp)
8     {
9         tmp->m_iEnd=iTmp;
10        return true;
11    }
12    else
13    {
14        EmitError("数组定义上限小于下限",TokenList.at(iListPos-1));
15        return false;
16    }
17 }
```

第 6 行：令 `tmp` 指针指向当前正在处理的数组维度信息表项。定位数组维度信息表项的工作主要分为两步：

1) 借助于 `iTypePos` 栈查找到当前类型信息表项。

2) 当前类型信息表项的 `m_ArrayInfo` 列表中的最末表项即为当前正在处理的数组维度信息表项。

第 8 行：数组界限的语义判断。在 Neo Pascal 中，数组声明必须满足下限小于或等于上限的规定，否则报错。在标准 Pascal 中，数组上、下限是允许为负数的，故 Neo Pascal 并没有严格规定上、下限整数的取值范围。

讨论至此，编译器似乎并没有对数组的基类型作任何语义处理。那么，数组的类型链到底是由谁构造的呢？似乎一直忽略了 `semantic015` 及 `semantic010` 的存在。在分析基本类型声明的语义处理时，笔者有意略过了这两个语义子程序。其原因就是这两个语义子程序是用于处理类型链的，而基本类型声明根本不涉及类型链问题，因此，没有必要深入阐述。然而，处理数组类型声明的过程中，编译器就无法回避类型链构造的问题了。下面，就来看看语义子程序是如何构造类型链的。



程序 4-17 semantic.cpp

相关文法:

类型 → 015 构造类型 010

```
1  bool semantic015()
2  {
3      if (iTypeFlag.top()==2 || iTypeFlag.top()==5 || iTypeFlag.top()==6 || iTypeFlag.top()==7)
4      {
5          iTypePos.push(SymbolTbl.TypeInfoTbl.size()-1);
6          return true;
7      }
8      if (iTypeFlag.top()==3 || iTypeFlag.top()==4)
9      {
10         TypeInfo Tmp;
11         Tmp.m_szName="_noname";
12         Tmp.m_szName.append(GetSerialId());
13         Tmp.m_iProcIndex=SymbolTbl.ProcStack.top();
14         Tmp.m_iState=0;
15         SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_iLink=SymbolTbl.TypeInfoTbl.size();
16         SymbolTbl.TypeInfoTbl.push_back(Tmp);
17         iTypePos.push(SymbolTbl.TypeInfoTbl.size()-1);
18         return true;
19     }
20     .....
21     return true;
22 }
```

第 3~7 行：根据当前类型标志判断是否为集合类型、过程类型、函数类型、函数返回类型，如果满足条件则将当前类型信息表项的位序号压栈。这里，读者先不必深究，稍后将给出详细分析。

第 8 行：判断当前类型标志是否为数组类型或文件类型。从文法上而言，在类型声明部分，此语义子程序被多次复用。因此，设置 `iTypeFlag` 的目的就是用于区别不同类型声明的语境。实际上，`semantic015` 的主要功能就是构造类型链。读者应该知道，不同复合类型的类型链结构是不尽相同的，例如，基本类型并不存在类型链。而记录类型的类型链则是一对多的关系，当然，设置记录类型字段列表的目的也仅仅是为了更好地解决其一对多的存储结构问题。然而，数组类型的类型链则是一一对一的关系，即数组类型的基类型是唯一的，不可能同时存在多个基类型。而这一特点与文件类型比较类似。

第 11、12 行：设置类型名字。从应用的角度而言，`semantic015` 所处理的数组基类型必定是匿名类型。不过，为了便于后续处理，编译器依然会为其分配一个默认且唯一的名字。当然，编译器还必须保证这个名字不与任何合法的用户语法元素（包括类型、变量、常量等等）的命名冲突，即不可占用任何用户的名字空间。这个条件看似比较苛刻，却并不难实现。通常，用户定义的语法元素的名字必须是合法的标识符。那么，编译器自动分配的名字只需破坏这一原则，自然就不可能产生任何冲突了。在 Neo Pascal 中，自动分配的名字以

“_noname”开头，后接一个顺序编号即可。由于合法的 Pascal 标识符是不允许以“_”开头的，所以不可能存在冲突。而顺序编号是由编译器自动产生的，以保证不存在重名现象。

第 14 行：设置 `m_iState` 为 0，表示当前类型未分析完成。这个属性保证了未完成分析的类型是不能被引用的，即使其类型信息表项已经被加入了符号表。

第 17 行：将当前类型信息表项的指针压栈，以便后续语义子程序继续完善该类型信息表项。最终，实现类型链的构造。

程序 4-18 semantic.cpp

相关文法：

类型 → 015 构造类型 010

```

1  bool semantic010()
2  {
3      if (!iTypePos.empty() && iTypePos.top() != -1)
4      {
5          SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_iState=1;
6          iTypePos.pop();           //iTypePos 栈顶元素出栈
7      }
8      return true;
9  }
```

第 3 行：为了使程序逻辑比较严谨，这里有必要作相应判断。

第 6 行：修改类型信息表项的 `m_iState` 属性，即表示该类型信息可以被引用了。当然，匿名类型是不可能被后续声明显式引用的。

`semantic015` 和 `semantic010` 是两个非常重要的语义子程序，在后续章节中，还将继续详细分析 `semantic015` 的剩余部分源代码。关于数组声明的处理，暂且讨论至此。实际上，现在读者再来回顾复合类型声明的语义处理可能已经并不是那么复杂了。在此，笔者并不想过多纠缠于源代码实现，更愿意将解决问题的思路与读者分享，以便读者举一反三，构造自己的编译器。

3. 记录类型

Pascal 的记录类型或者 C 语言的结构类型都是比较特殊的类型，它们是面向对象语言产生与发展的重要基础。早期的面向对象语言的设计初衷就是在记录或结构的基础上，引入函数成员，即所谓的操作或方法，以便将数据与基于数据的函数视作一个整体，称为“类”。虽然类的思想并不足以囊括面向对象设计的全部，也未必称得上是面向对象思想的精髓，但它是面向对象时代的先行者。下面，就来看看记录类型的相关文法的候选式。

【文法 4-10】

类型	→ 015 构造类型 010
构造类型	→ 记录类型
记录类型	→ record 019 字段列表 end 020
字段列表	→ 字段固定部分 字段列表 字段可变部分 字段列表 ε
字段固定部分	→ 标识符列表 : 类型 ;
字段可变部分	→ case 标识符 029 : 类型 of 033 字段可变部分 1 034 end 030 ;



字段可变部分 1 → 常量 031:(字段列表)032; 字段可变部分 1 |ε

记录类型的文法相对比较复杂。不过，其构造类型链的核心思想与数组声明的处理是非常相似的。笔者先从 semantic019 和 semantic020 这一对语义子程序着手开始分析记录类型声明的相关源代码实现。事实上，这两个语义子程序与先前的 semantic016、semantic025（数组声明部分的语义子程序）是非常类似的。

程序 4-19 semantic.cpp

相关文法:

记录类型 → record 019 字段列表 end 020

```
1 bool semantic019()
2 {
3     SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_eDataType=StoreType::T_RECORD;
4     iTypeFlag.push(1);
5     iIdListFlag.push(2);
6     return true;
7 }
8 bool semantic020()
9 {
10    iTypeFlag.pop();
11    iIdListFlag.pop();
12    return true;
13 }
```

第 6 行：在处理记录类型声明时，必然涉及“标识符列表”。根据先前的讨论，由于“标识符列表”存在多处复用，因此，不得不运用 iIdListFlag 栈加以标识。

Pascal 记录类型的字段可以分为两类：固定字段、可变字段。首先，笔者介绍有关固定字段的语义处理。从文法来看，“字段固定部分”并没有设置语义子程序，实际并非如此。由于“标识符列表”、“类型”都是非终结符，因此，“标识符列表”及“类型”相关产生式中包含的语义子程序都有可能在分析过程中被调用。在此过程中，编译器必须协调处理非终结符复用的情况，即通过设置相应的标志以标识分析状态。虽然先前已经讨论了这两个非终结符的相关语义子程序，却并没有完整地给出 semantic012、semantic015 的源代码。下面，笔者就将讨论这两个语义子程序中与记录类型声明相关的语义动作。

程序 4-20 semantic.cpp

相关文法:

标识符列表 → 标识符 012 标识符列表 1

标识符列表 1 → , 标识符 012 标识符列表 1

```
1 bool semantic012()
2 {
3     .....
```



```
4     if (iIdListFlag.top()==2)
5     {
6         FieldInfo Tmp;
7         Tmp.m_szName=TokenList.at(iListPos-1).m_szContent;
8         Tmp.m_iState=0;
9         Tmp.m_iProcIndex=SymbolTbl.ProcStack.top();
10        for(int i=SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_FieldInfo.size()-1;i>=0;i--)
11        {
12            string szTmp=SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_FieldInfo.at(i).m_szName;
13            if (szTmp.compare(Tmp.m_szName)==0)
14            {
15                EmitError(Tmp.m_szName.append("域名已经存在"),TokenList.at(iListPos-1));
16                return false;
17            }
18        }
19        SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_FieldInfo.push_back(Tmp);
20    }
21    .....
22 }
```

第 8 行：设置 `m_iState` 属性。注意，这里的 `m_iState` 并不是隶属于类型信息表项的。在讨论符号表结构时，笔者并没有详细讨论该属性的功能。这个属性并不是一个符号表的实义属性，仅仅是为了便于语义子程序登记符号表而设置的一个临时属性。也就是说，当语义分析器完成符号表登记后，这个属性也就失去了实际意义，它不会为后续编译阶段提供任何有价值的信息。至于设置 `m_iState` 属性的目的也并不复杂，主要用于处理多个字段隶属于同一类型的情况。例如：

```
TYPE
    c=RECORD aa,bb,cc:INTEGER;    END;
```

根据符号表的设计，这 3 个字段的 `m_iLink` 将指向同一个类型信息表项。当然，这种声明方式可能会给语义分析带来一些小小的不便。由于编译器从左向右扫描输入源程序，在处理 `aa`、`bb` 两个字段表项时，编译器将无法预知其类型，当然，也无法填写 `m_iLink` 指针。当分析到 `cc` 字段时，编译器才可以获得其类型表项的指针。注意，至此编译器仍无法预知类型的详情，仅能知道其类型表项指针而已。不过，凭借此信息也足以填写 `m_iLink` 指针了。此时，编译器必须将获得的指针信息同时回填到以上 3 个字段的 `m_iLink` 属性中，因此，编译器就必须根据一个特殊的标志属性来判断哪些字段需要回填。设置 `m_iState` 属性的目的正在于此。

第 11 行：这是一个循环判断的过程，主要用于判断字段名是否已被占用。根据 Pascal 的规定，字段的可见域仅限于其直属记录类型范围内。因此，编译器只需在当前类型的字段列表范围内判断当前符号名是否已被占用即可。



程序 4-21 semantic.cpp

相关文法:

类型 → 015 构造类型 010

```
1  bool semantic015()
2  {
3      .....
4      if (iTypeFlag.top()==1)
5      {
6          TypeInfo Tmp;
7          Tmp.m_szName="_noname";
8          Tmp.m_szName.append(GetSerialId());
9          Tmp.m_iProcIndex=SymbolTbl.ProcStack.top();
10         Tmp.m_iState=0;
11         SymbolTbl.TypeInfoTbl.push_back(Tmp);
12         int i=SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_FieldInfo.size()-1;
13         int iTmpTypePos=iTypePos.top();
14         iTypePos.push(SymbolTbl.TypeInfoTbl.size()-1);
15         while (i>=0)
16         {
17             if (SymbolTbl.TypeInfoTbl.at(iTmpTypePos).m_FieldInfo.at(i).m_iState==0)
18             {
19                 SymbolTbl.TypeInfoTbl.at(iTmpTypePos).m_FieldInfo.at(i).m_iLink=iTypePos.top();
20                 SymbolTbl.TypeInfoTbl.at(iTmpTypePos).m_FieldInfo.at(i).m_iState=1;
21             }
22             i--;
23         }
24     }
25     .....
26 }
```

第 4 行：根据 `iTypeFlag` 标志栈，完成相应的语义动作。

第 7、8 行：设置临时表项的 `m_szName` 的名字。前面，笔者已经详细分析了匿名类型的处理，这里不再赘述。

第 11 行：将类型信息表项加入类型信息表中。

第 15~23 行：循环回填字段的 `m_iLink` 属性。这是一个逆向处理的过程，从当前类型的字段信息表表末开始，逐一向上查询至表首。

至此，完成了记录类型固定部分的语义分析及符号表登记工作。下面，继续来讨论可变部分的相关语义子程序。可变部分的文法看似比较繁复，但实际上其语义处理却不复杂。

这里，有必要说明一点，严格地说，Neo Pascal 的记录类型与标准 Pascal 还是存在一定差别的。主要的区别在于可变部分的 `case` 结构，标准 Pascal 并没有明确规定 `case...of` 之间必须是“标识符：类型”结构，通常允许省略“标识符：”。然而，Neo Pascal 的文法却不允许省略“标识符：”。下面，就从语义分析的角度来看看其相关的实现。

程序 4-22 semantic.cpp

相关文法:

字段可变部分 → case 标识符 029: 类型 of 033 字段可变部分 1 034 end 030 ;

```
1 bool semantic029()
2 {
3     FieldInfo Tmp;
4     Tmp.m_szName=TokenList.at(iListPos-1).m_szContent;
5     Tmp.m_iState=0;
6     Tmp.m_iProcIndex=SymbolTbl.ProcStack.top();
7     for(int i=SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_FieldInfo.size()-1;i>=0;i--)
8     {
9         string szTmp=SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_FieldInfo.at(i).m_szName;
10        if (szTmp.compare(Tmp.m_szName)==0)
11        {
12            EmitError(Tmp.m_szName.append("域名已经存在"),TokenList.at(iListPos-1));
13            return false;
14        }
15    }
16    szVarFieldFlag.push(Tmp.m_szName);
17    SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_FieldInfo.push_back(Tmp);
18    return true;
19 }
```

semantic029 与 semantic012（部分）非常相似，除了第 16 行外，可以说是完全相同的。semantic029 主要就是用于处理位于 case...of 结构之间的字段（暂且称为“可变部分的标志字段”）。从用户的角度而言，可变部分的标志字段同样被视作普通的固定字段。不过，这个字段同时还承担着标识一个可变字段集的功能。从 Neo Pascal 符号表的结构而言，实际上，字段列表中的 m_szVarFieldFlag 属性存储的就是所属 case 结构的可变区域标识字段名。

第 16 行：将可变部分的标志字段入栈。根据符号表的设计，该字段所属 case 结构中的所有字段的 m_szVarFieldFlag 属性都将填入该字段名。显然，在 semantic029 中是无法完成的，编译器必须将该字段名暂存，以备后用。至于为什么使用栈结构保存该字段名？原因非常简单，记录类型的可变部分同样存在复杂的类型嵌套。根据先前的讨论，栈结构非常有利于处理这种嵌套引起的递推及回归问题。

semantic030 的功能非常简单，主要工作就是将先前压入的可变区域字段名出栈，保证栈的平衡即可。

程序 4-23 semantic.cpp

相关文法:

字段可变部分 → case 标识符 029: 类型 of 033 字段可变部分 1 034 end 030 ;

```
1 bool semantic030()
2 {
3     szVarFieldFlag.pop();
4     return true;
5 }
```



semantic033、semantic034 的功能非常单一，主要工作就是处理 iIdListFlag 标志栈。字段可变部分的字段名的处理与固定部分的字段名的处理动作是不同的，主要体现在两者所需填写的符号表属性不完全一致。通常，都借助于 iIdListFlag 标志栈以区分不同的语义动作，这种解决方式已经多次使用了，读者应该并不陌生。

程序 4-24 semantic.cpp

相关文法：

字段可变部分 → case 标识符 029: 类型 of033 字段可变部分 1 034 end 030 ;

```
1 bool semantic033()
2 {
3     iIdListFlag.push(3);
4     return true;
5 }
6 bool semantic034()
7 {
8     iIdListFlag.pop();
9     return true;
10 }
```

既然提到了 iIdListFlag 标志栈，就有必要分析一下非终结符“标识符列表”相关语义子程序当 iIdListFlag 栈顶标志为“3”时的具体处理动作。根据前面的经验，非终结符“标识符列表”有关语义子程序就是 semantic012。实际上，与先前固定部分的处理类似，可变部分处理的要点同样在于此。下面，就给出 semantic012 中的相关源代码。

程序 4-25 semantic.cpp

相关文法：

标识符列表 → 标识符 012 标识符列表 1

标识符列表 1 → , 标识符 012 标识符列表 1

```
1 bool semantic012()
2 {
3     .....
4     if (iIdListFlag.top()==3)
5     {
6         FieldInfo Tmp;
7         Tmp.m_szName=TokenList.at(iListPos-1).m_szContent;
8         Tmp.m_iState=0;
9         Tmp.m_iProcIndex=SymbolTbl.ProcStack.top();
10        Tmp.m_szVarFieldConst=szVarFieldConst.top();
11        Tmp.m_szVarFieldFlag=szVarFieldFlag.top();
12        for(int i=SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_FieldInfo.size()-1;i>=0;i--)
13        {
14            FieldInfo *pTmp=&SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_FieldInfo.at(i);
15            if (pTmp->m_szName.compare(Tmp.m_szName)==0)
```

```

16     {
17         EmitError(Tmp.m_szName.append("域名已经存在"),TokenList.at(iListPos-1));
18         return false;
19     }
20     if (pTmp->m_iState==1 &&
21         pTmp->m_szVarFieldConst.compare(Tmp.m_szVarFieldConst)==0 &&
22         pTmp->m_szVarFieldFlag.compare(Tmp.m_szVarFieldFlag)==0 )
23     {
24         EmitError(Tmp.m_szName.append("常量值已经存在"),TokenList.at(iListPos-1));
25         return false;
26     }
27     }
28     SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_FieldInfo.push_back(Tmp);
29 }
30 .....
31 }

```

实际上，读者不难发现，这部分与先前固定部分相关处理的源代码结构还是比较类似的。下面，笔者只针对其中的差异稍作解释，其余部分的分析请读者参见先前章节。

第 10、11 行：设置符号表表项的 `m_szVarFieldConst`、`m_szVarFieldFlag` 属性。分别将 `szVarFieldConst`、`szVarFieldFlag` 两个栈的栈顶元素赋给了这两个属性。`szVarFieldFlag` 栈前面已作说明，而 `szVarFieldConst` 栈的功能就是用于暂存各 case 常量，具体的结构与前者是非常类似的。

第 20~26 行：这个 if 语句主要是用于判断 `m_szVarFieldConst`、`m_szVarFieldFlag` 属性的值是否有效。根据 Pascal 语言的规定，同一个可变区域（即 case 结构）内的常量值是不允许重复的，但不同可变区域内的常量值是允许重复的。

最后看看 `semantic031`、`semantic032` 这两个语义子程序的相关实现。前面，笔者已经分析了 `semantic029`、`semantic030`，读者应该了解它们的主要功能就是管理 `szVarFieldFlag` 栈，但至今未提到 `szVarFieldConst` 栈的相关管理。因此，也就不难想到这两个语义子程序可能与 `szVarFieldConst` 栈的管理有关。

程序 4-26 semantic.cpp

相关文法：

字段可变部分 1 → 常量 031 : (字段列表) 032 ; 字段可变部分 1 | ε

```

1 bool semantic031()
2 {
3     int i=atoi(TokenList.at(iListPos-1).m_szContent.c_str());
4     szVarFieldConst.push(SymbolTbl.ConstInfoTbl.at(i).m_szVal);
5     return true;
6 }
7 bool semantic032()
8 {

```



```

9      szVarFieldConst.pop();
10     return true;
11  }
```

4. 指针类型

熟悉 C 语言的读者对指针类型应该并不陌生，有不少人坚持认为强大、灵活的指针机制可以为程序设计提供不少便利。不过，指针是一把双刃剑，自诞生之日起，就一直饱受争议。有些程序设计语言学家认为指针是一种极不安全的语法结构，一些新型语言（例如，Java、C#、Python 等）甚至摒弃了这一机制。关于指针的讨论可谓众说纷纭，在此，笔者也不便多作评说。不过，从研究一门程序设计语言的角度而言，指针机制还是值得深入剖析的，很多经典的论题都是在指针基础上讨论的。下面，就先来看看指针类型声明的相关文法。

【文法 4-11】

```

类型      → 015 指针类型 010
指针类型  → ^ 简单类型
简单类型  → 标识符 008 | char 009 | boolean 009 | integer 009 | byte 009 | shortint
           009 | smallint 009 | word 009 | longword 009 | cardinal 009 | real 009 | single 009
```

这里，引入了一个非终结符“简单类型”，其主要原因就是由于标准 Pascal 不支持将匿名类型作为指针的基类型。例如：

【声明 4-16】

```
TYPE a=^ARRAY [1..10] OF INTEGER;
```

【声明 4-17】

```
TYPE b=ARRAY [1..10] OF INTEGER;
      a=^b;
```

声明 4-16 是非法的 Pascal 声明形式，只有将其改写成声明 4-17 的形式，才能被编译器接受。这是由于 Pascal 语言是一门强类型语言，Pascal 编译器必须严格检查指针及其所指向元素的类型是否兼容。显然，依据这一规定，基类型为匿名类型的指针将失去任何实用价值。实际上，不同语言对于类型兼容的定义是存在差异的。这里，暂且不必深入讨论该话题，读者只需了解一点，即 Pascal 处理类型的基本策略是从严处理的，这是与 C 语言截然不同的。而 C 程序员不适应 Pascal 的主要原因也在于其强类型机制。

从 Neo Pascal 的文法可知，与指针类型声明相关的语义子程序主要有 semantic008、semantic009。下面，笔者就详细分析这两个语义子程序。

显而易见，semantic008 是用于处理指向用户自定义类型的指针声明，语义子程序就是根据用户自定义类型的名字检索符号表，判断该名字是否有效，如有效则准备构造类型链。

程序 4-27 semantic.cpp

相关文法:

简单类型 → 标识符 008

```
1 bool semantic008()
2 {
3     SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_eDataType=StoreType::T_POINTER;
4     SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_szContent=TokenList.at(iListPos-1).m_szContent;
5     int i;
6     i=SymbolTbl.SearchTypeInfoTbl(SymbolTbl.ProcStack.top(),
7                                 TokenList.at(iListPos-1).m_szContent,false);
8     if (i!=-1)
9     {
10         if (SymbolTbl.TypeInfoTbl.at(i).m_iState!=1)
11         {
12             EmitError("指针指向的类型声明不完整",TokenList.at(iListPos-1));
13             return false;
14         }
15     }
16     return true;
17 }
```

第 4 行：设置类型信息表项的 `m_szContent` 属性。读者可能有疑问，为什么需要设置这个属性？为什么在 `semantic008` 中没有设置 `m_iLink` 属性（即构造类型链）？

在讲解符号表结构时，笔者曾经提到过 `m_iState` 属性的作用及其指针类型声明的特殊性。根据先前的分析，读者应该已经知道，指针类型的基类型并不一定是在指针类型之前声明的。这是指针类型的一个非常特殊的性质，是其他复合类型都不具备的。

在分析指针类型声明时，由于指针类型的基类型可能出现在后续某一个位置上，编译器是无法预知的，这种情况与先前讨论的匿名类型是完全不同的。例如，声明形式如下：

【声明 4-18】

```
A=ARRAY [1..10] OF ARRAY [1..20] OF INTEGER;
```

编译器在分析 `A` 类型时，可以确定 `A` 是数组类型，并且其基类型是一个匿名类型。虽然编译器无法预知其后的匿名类型的复合层次，但可以知道其在源程序中的位置。因为根据数组类型的文法，如果其基类型为匿名类型，则必须出现在“OF”之后。不过，这一约定却并不适用于指针类型。

在分析指针类型声明时，通常只是将基类型的名字记录下来，待类型声明分析完毕后，重新遍历类型信息表，根据记录下来的名字检索类型信息表，回填 `m_iLink` 属性，构造类型链。`m_szContent` 属性就是用于暂存基类型的名字。

第 10 行：注意这个判断并不是多余的，主要用于避免指针引用自身类型。类似于“`A=^A;`”的声明形式。虽然这种指针引用形式本身并不存在语义错误，但标准 Pascal 却并不支持。



那么，编译器又是如何回填 `m_iLink` 属性的？根据前面所说的，编译器将在适当的时刻完成此项工作。那么，到底何为适当的时刻呢？实际上，在分析完一个过程或函数的声明部分后，编译器就将调用 `CSymbolTbl.PtrCheck` 函数，以完成 `m_iLink` 回填的工作。下面，笔者将详细分析 `CSymbolTbl.PtrCheck` 函数的实现。先看看回填 `m_iLink` 属性的具体过程。

程序 4-28 Symbol.cpp

```
1  bool CSymbolTbl::PtrCheck(int &iPos)
2  {
3      int i=0;
4      for (i=0;i<TypeInfoTbl.size();i++)
5      {
6          if(TypeInfoTbl.at(i).m_eDataType==StoreType::T_POINTER
7              && !TypeInfoTbl.at(i).m_szContent.empty())
8              {
9                  int j=SearchTypeInfoTbl(TypeInfoTbl.at(i).m_iProcIndex,
10                                         TypeInfoTbl.at(i).m_szContent,false);
11                  if (j==-1)
12                      j=SearchTypeInfoTbl(0,TypeInfoTbl.at(i).m_szContent,false);
13                  if (j!=-1)
14                      {
15                          TypeInfoTbl.at(i).m_szContent="";
16                          TypeInfoTbl.at(i).m_iLink=j;
17                      }
18                  else
19                      {
20                          iPos=i;
21                          return false;
22                      }
23              }
24      }
25      return true;
26  }
```

第 6、7 行：判断该类型信息表项是否为 `T_POINTER` 类型且是否需要回填 `m_iLink`。当表项的 `m_szContent` 属性不为空时，即表示该表项需要回填 `m_iLink` 属性。

第 9、10 行：检索当前过程的类型信息表。根据 Pascal 语言名字空间的规定，编译器将优先检索直接所属过程的类型信息表项。

第 12 行：当检索直接所属过程的类型信息表失败后，编译器将继续检索主程序的类型信息表。当然，由于 Neo Pascal 不允许过程、函数（除主程序外）的嵌套声明，所以检索工作变得相对容易。如果按照标准 Pascal 的规定，编译器将由内向外逐级检索。

第 20 行：当两次检索都失败后，表明该指针指向的用户自定义类型名字并不存在，即表示该指针的声明是非法的。函数必须回传该指针表项的指针（即通过 `iPos` 引用参数实

现)，便于语义子程序报告错误。

程序 4-29 semantic.cpp

相关文法:

简单类型 → char 009 | boolean 009 | integer 009 | byte 009 | shortint 009 | smallint 009
| word 009 | longword 009 | cardinal 009 | real 009 | single 009

```
1 bool semantic009()
2 {
3     SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_eDataType=StoreType::T_POINTER;
4     SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_iLink=SymbolTbl.TypeInfoTbl.size();
5     TypeInfo Tmp;
6     Tmp.m_iState=1;
7     Tmp.m_szName="_noname";
8     Tmp.m_szName.append(GetSerialId());
9     Tmp.m_iProcIndex=SymbolTbl.ProcStack.top();
10    Tmp.m_eDataType=CType::TokenToEnum(TokenList.at(iListPos-1).m_iKind);
11    Tmp.m_eBaseType=CType::TokenToEnum(TokenList.at(iListPos-1).m_iKind);
12    SymbolTbl.TypeInfoTbl.push_back(Tmp);
13    return true;
14 }
```

第 6 行：与 semantic008 的处理完全不同，由于指针基类型都是基本类型，所以构造类型链变得非常容易。语义子程序只需为基类型增加一个表项，然后通过 m_iLink 属性将两者链接起来即可。

第 8 行：由于基类型是一个基本类型，又是一个匿名类型。编译器并不需要过多理会其 m_iState 属性。

5. 集合类型

虽然 Pascal 的集合类型可能是对传统数据类型的一种突破与丰富，但是笔者个人认为 Pascal 的集合类型的设计并不成功。由于它的限制较多，因此应用领域非常之小，许多 Pascal、Delphi 程序员很少使用集合类型。即便如此，Pascal 集合类型却成为了先驱，是后来程序设计语言完美实现集合类型的奠基石。其中，C++ 的 set 模板类就是一个成功的实例。当然，考虑到尽可能兼容标准 Pascal 语言，所以 Neo Pascal 依然保留了集合类型。集合类型的相关文法非常简单，只有一条产生式。下面，笔者就简单分析一下集合类型声明的相关源代码实现。

程序 4-30 semantic.cpp

相关文法:

集合类型 → set 024 of 类型 025

```
1 bool semantic024()
2 {
3     SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_eDataType=StoreType::T_SET;
4     iTypeFlag.push(2);
```




```
5     return true;
6   }
7   bool semantic025()
8   {
9     iTypeFlag.pop();
10    return true;
11  }
```

集合类型声明的语义处理比较简单，主要是基于 `iTypeFlag` 标志栈的操作。在介绍数组声明时，笔者已经详细分析了 `iTypeFlag` 标志栈的作用以及构造类型链的过程，这里不再赘述。根据先前的讲述，类型链的构造将在非终结符“类型”的相关推导过程中完成。

在标准 Pascal 语言中，集合类型的基类型必须满足两个条件：一是有序类型；二是类型的允许取值总数不能超过 256。在 Pascal 语言中，集合类型通常指的是一个有限的集合，即集合所允许的最大元素个数是限定的，这是由 Pascal 集合类型的实现内核决定的。实际上，Pascal 集合类型的实现内核并不复杂，每个集合类型变量都有 32B（即 256 位）的存储空间，每个位表示一个集合元素，使用位与、位或等操作来模拟集合的交、并、差运算。从 Pascal 集合类型的实现机制分析，不难发现，它是比较低效的，并且又有一定的局限性。即便如此，这种实现机制还一直被 Turbo Pascal、Delphi 等编译器沿用至今。Neo Pascal 为了尽可能兼容标准 Pascal，同样采用这种并不高效的实现机制。当然，随着人们的研究深入，不少更为高效的实现方式随之产生。有兴趣的读者可以参考研究 C++ 的相关实现，相信必定能获益良多。

了解了 Pascal 集合类型的特性后就不难发现，构造集合类型链相关的语义子程序主要包括：`semantic015`、`semantic007`、`semantic011`、`semantic023`。由于这些语义子程序都存在多重复用，所以必须根据 `iTypeFlag` 标志栈的标识执行相应的语义动作。`semantic015`、`semantic011` 已在数组类型声明章节中详细分析过，不再赘述。下面，主要来看看 `semantic007` 的相关语义动作，而 `semantic023` 暂且不作讨论。

`semantic007` 主要用于处理基类型是基本类型（枚举类型除外，由 `semantic011` 处理）的集合类型声明。

程序 4-31 semantic.cpp

相关文法：

有序类型 → `char 007` | `boolean 007` | 整数类型 | 枚举类型

无序类型 → `real 007` | `single 007`

整数类型 → `integer 007` | `byte 007` | `shortint 007` | `smallint 007` | `word 007`
| `longword 007` | `cardinal 007`

```
1 bool semantic007()
2 {
3     if (iTypeFlag.top()==2)
4     {
5         SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_iLink=SymbolTbl.TypeInfoTbl.size();
6         SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_iState=1;
7         TypeInfo Tmp;
```

```
8      Tmp.m_iState=1;
9      Tmp.m_szName="_noname";
10     Tmp.m_szName.append(GetSerialId());
11     Tmp.m_iProcIndex=SymbolTbl.ProcStack.top();
12     Tmp.m_eDataType=CType::TokenToEnum(TokenList.at(iListPos-1).m_iKind);
13     Tmp.m_eBaseType=CType::TokenToEnum(TokenList.at(iListPos-1).m_iKind);
14     if (Tmp.m_eBaseType==StoreType::T_BOOLEAN ||
15         Tmp.m_eBaseType==StoreType::T_BYTE ||
16         Tmp.m_eBaseType==StoreType::T_CHAR ||
17         Tmp.m_eBaseType==StoreType::T_SHORTINT )
18     {
19         SymbolTbl.TypeInfoTbl.push_back(Tmp);
20         return true;
21     }
22     else
23     {
24         EmitError("集合类型的基类型不正确",TokenList.at(iListPos-1));
25         return false;
26     }
27 }
28 .....
29 }
```

第 5 行：设置表项的 `m_iLink` 属性，构造类型链。集合类型的类型链是比较简单的，只可能有两个类型结点，即主属类型结点和基类型结点。与数组声明不同的是集合类型的基类型结点并不是由 `semantic015` 产生的，而是由基类型相关的语义子程序直接生成的。例如，这里的 `semantic007` 就是基类型的语义子程序，读者只要比较 `semantic007` 中针对不同类型标志的语义动作，便可理解其中的差别了。当然，采用与数组声明处理类似的方式也是完全可行的，只是程序逻辑可能复杂一些。

第 7~13 行：生成一个临时类型结点，将其设置为基类型结点。

第 14~17 行：用于判断集合类型的基类型是否为合法类型。这里，不必考虑枚举类型的情况，因为 `semantic007` 并不需要兼顾枚举类型声明的相关语义处理。枚举类型的语义处理由 `semantic011` 完成，先前笔者已作了详细分析。

6. 字符串类型

字符串类型是一种常用的数据类型，其应用领域极为广泛。随着计算机的广泛普及，从应用角度而言，字符串处理的重要性可能已经凌驾于传统数值处理之上了，例如，办公服务、Web 应用、信息检索等等。实际上，编译技术也与字符串处理有着密切的联系。不过，在 20 世纪六七十年代，计算机的主要应用还局限在数值计算领域，不少程序设计语言对于字符串处理的支持都不尽理想。当然，一个重要的事实也不可否认，那就是当时的硬件环境也并不允许程序设计语言支持过于复杂的字符串处理操作。

严格地说，无论是 C 语言的字符指针还是 Pascal 的字符数组都不属于真正的字符串类型。它们将字符串的逻辑、物理结构完全暴露在用户面前，其唯一目的就是希望由用户为某



些复杂的字符串操作负责，至于这些操作的效率与性能更多取决于用户的编程技巧与经验。然而，Delphi、C#等商用编译器在字符串应用方面就做得比较好，可能都是出自天才设计师 Anders Hejlsberg 之手的原因吧，两者字符串的内核实现机制（这里主要指的是逻辑、物理结构，并不考虑.NET 的托管机制）也非常类似，用户可以方便地应用字符串而不必过多关注其内核实现。C++在字符串处理方面同样也是比较出色的，基于 Windows 平台的 VC++编译器更是提供了两种不同的字符串类型，即 MFC 的 CString 和 STL 的 string。不论是何种字符串类型，究其本质仍然是字符指针的不同封装形式而已。由此可见，字符串类型已经得到广泛的应用，软、硬件领域都在字符串处理方面作了不少探索。很少有一个现代编译器是不支持字符串类型的，故笔者在 Neo Pascal 中增加了对字符串类型的支持，这与标准 Pascal 是有区别的。

字符串类型声明的语义处理是非常简单的，从文法上来看，字符串类型实际上就是一个基本类型，它的语义动作与普通的整型、实型无异。读者可以参考基本类型声明的相关分析。

下面，简单讨论一下 Neo Pascal 字符串类型的内核实现。Neo Pascal 字符串类型的内核就是一个字符指针，这与 Delphi 的 string 类型是有区别的。笔者这样设计的主要原因是为了兼容 Windows 的标准 API 函数。由于 Windows 大部分 API 接口主要兼容 C 语言，所以其 API 的参数都是以字符指针类型替代字符串类型的。像 Delphi 的 string 类型等是不符合这一标准的，所以在调用 API 时，不得不进行一些类型转换操作。为了避免这些麻烦，笔者并没有采用 Delphi string 的实现方案。实际上，Neo Pascal 的 string 更像是 Delphi 语言的 PChar 类型，只是在其基础上增加了一些字符串运算操作而已。

不论是哪种内核实现，在不考虑效率的情况下，编译器已经完全有能力将字符串类型封装成一个使用极其方便的基本类型，对于用户而言，可以像处理普通数值类型一样处理字符串类型。但是，字符串处理始终没有走出字符指针的影子。即便是今天，某些字符串的处理已经被直接引入到 CPU 的指令集中，对于字符串的支持仍然不够完美，在大量涉及字符串操作时，其处理效率仍然不尽如人意，尤其是字符串连接、子串检索等。计算机科学也始终没有中止对字符串的研究与探索，当然，并不仅仅局限于软件、算法领域。因此，有理由相信未来计算机科学在字符串处理领域一定会有所突破的。

7. 用户自定义类型

前面，笔者花费了大量的篇幅逐一介绍了各种类型声明的语义处理动作。现在讨论类型声明部分的最后一个话题——用户自定义类型。这种声明形式是常见的，例如：

【声明 4-19】

TYPE

```
A=CHAR;  
B=SET OF A;
```

B 是一个集合类型，而该集合类型的基类型是另一个用户自定义类型 A。在分析 B 的声明时，必须将其类型结点的 m_iLink 属性设为 A 类型信息表项的位序号，以构成 B 的类型链。不过，处理这种声明形式的语义动作与之前讨论的是不同的。此前，编译器更多关心的是声明右部是匿名类型的情况，即声明右部是一个具体类型声明，而不是一个现已存在的用

户自定义类型名。在处理匿名类型时，构造类型链的工作通常会变得比较复杂。主要原因就是编译器无法预知匿名类型的嵌套层次，分析过程是递归向下进行的。虽然从表面上来看，语义分析器并不存在真正的递归调用，它只不过是利用栈模拟递归的一种实现方式而已，其本质还是递归的思想。与匿名类型的处理相比，处理声明右部是用户自定义类型名字的情况就容易得多了，其过程可能更多涉及的是类型信息的检索。

在声明 4-19 中，分析 B 类型声明的过程如下：当分析到 B 的基类型是 A 时，便检索符号表，如果 A 是合法的类型名字，则将两者构成类型链，否则报错（指针类型声明除外）。这种语义动作非常简单，并不需要理会任何因模拟递归而设置的临时栈结构。下面，就来看看用户自定义声明形式的文法：

【文法 4-12】

类型 → 015 标识符 023 010

其中 semantic015、semantic010 已作分析，这里，主要关注 semantic023 的相关源代码实现。

程序 4-32 semantic.cpp

相关文法：

类型 → 015 标识符 023 010

```

1  bool semantic023()
2  {
3      string szTmp=TokenList.at(iListPos-1).m_szContent;
4      int i=SymbolTbl.SearchTypeInfoTbl(SymbolTbl.ProcStack.top(),szTmp);
5      if (i== -1)
6      {
7          i=SymbolTbl.SearchTypeInfoTbl(0,szTmp);
8      }
9      if (i!= -1)
10     {
11         if (iTypeFlag.top()==2)
12         {
13             int j=i;
14             string szTmp1;
15             while (SymbolTbl.TypeInfoTbl.at(j).m_eDataType==StoreType::T_USER)
16             {
17                 if (SymbolTbl.TypeInfoTbl.at(j).m_iLink== -1)
18                 {
19                     EmitError("系统分析出错",TokenList.at(iListPos-1));
20                     return false;
21                 }
22                 szTmp1=SymbolTbl.TypeInfoTbl.at(SymbolTbl.
23                     TypeInfoTbl.at(j).m_iLink).m_szName;
24                 j=SymbolTbl.SearchTypeInfoTbl(SymbolTbl.ProcStack.top(),szTmp1);
25             }

```



```
26         if (SymbolTbl.TypeInfoTbl.at(j).m_eBaseType==StoreType::T_BOOLEAN ||
27             SymbolTbl.TypeInfoTbl.at(j).m_eBaseType==StoreType::T_BYTE ||
28             SymbolTbl.TypeInfoTbl.at(j).m_eBaseType==StoreType::T_CHAR ||
29             SymbolTbl.TypeInfoTbl.at(j).m_eBaseType==StoreType::T_SHORTINT ||
30             SymbolTbl.TypeInfoTbl.at(j).m_eBaseType==StoreType::T_SMALLINT ||
31             SymbolTbl.TypeInfoTbl.at(j).m_eDataType==StoreType::T_ENUM)
32         {
33             SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_iLink=i;
34             return true;
35         }
36         else
37         {
38             EmitError("集合类型的基类型不正确",TokenList.at(iListPos-1));
39             return false;
40         }
41         .....
42     }
43     SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_iLink=i;
44     SymbolTbl.TypeInfoTbl.at(iTypePos.top()).m_eDataType=StoreType::T_USER;
45     return true;
46 }
47 else
48 {
49     EmitError(szTmp.append("类型未定义或定义不完整"),TokenList.at(iListPos-1));
50     return false;
51 }
52 return true;
53 }
```

第 3 行：获得用户自定义类型的名字。用户自定义类型的名字可以直接从单词流中获得。

第 4~9 行：两次检索类型信息表，判断该用户自定义类型的名字是否合法。程序设计语言关于符号的可见域、作用域都有严格的规定，而且有时可能是比较复杂的。例如，C 语言允许声明代码块级的变量，而标准 Pascal 允许过程嵌套声明，这些情况将使可见域、作用域的分析变得复杂。由于 Neo Pascal 不支持过程嵌套声明，所以符号的可见域、作用域问题就变得相对简单。

第 11 行：这是一个重要的标志判断。处理集合类型声明时，由于集合类型的基类型是一个用户自定义类型，所以编译器需要对其作一定的合法性判断。例如：

【声明 4-20】

TYPE

A=ARRAY [1..10] OF INTEGER;

B=SET OF A;

根据 Pascal 语言的规定，这样的声明也是不正确的。编译器并不能因为声明形式的改变，而忽略合法性检查。

第 13~24 行：获得基类型的实际类型（非用户自定义类型）结点。任何用户自定义类型最终还将链接到实际类型结点。这个循环的目的就是过滤所有用户自定义类型的结点，直接获得实际类型的结点，便于后续合法性判断。

第 26~40 行：判定集合类型的基类型是否合法，若合法则构造类型链，否则报错。

第 43、44 行：处理其他引用用户自定义类型的声明，即生成类型结点，构造类型链。

应该注意的是，这个语义子程序并不适用于指针类型基类型的处理，根据前面所分析的例程，指针类型基类型的语义处理通常有一定的特殊性，即基类型名字的合法性是待事后判断的。

8. 更多话题

至此，除了函数类型之外，笔者已经详细介绍了 Neo Pascal 各种类型的声明形式及其语义处理动作。在阅读源代码的过程中，读者难免会感到有点疑惑，这也并不足为奇。类型系统是高级程序设计语言有别于低级程序设计语言的重要标志之一，因此，它对高级程序设计语言的发展有着极其深远的影响。对于有志于学习程序设计语言及编译技术的读者而言，类型系统还是值得深入研究的。

学习本小节的关键在于理解类型链的概念及其构造方法。类型链并不是 Neo Pascal 所特有的产物，也不是笔者凭空想象出来的。类型链是一种比较通用的类型描述方法，得到了编译器设计者的普遍认可。当然，不同编译器的具体实现可能存在一定的差异，但是理解其概念与设计思想对于学习分析其他编译器是有一定帮助的。实际上，从访问效率及存储的角度而言，Neo Pascal 的类型链、符号表的设计并不完美。例如，进一步可以为符号表设计一个索引，那样将大大提高检索的效率。有兴趣的读者完全可以自行对符号表结构及其操作稍加修改，实现这种索引机制。

还需指出一点，本书所讨论的类型仅仅是类型系统的具体实现而已。实际上，这里讨论的类型系统与计算机科学中的类型理论是有一定区别的。类型理论研究是程序设计语言领域的一个重要课题，它侧重于应用数学的方法从抽象层次上研究类型系统的设计、推理与实现。由于类型理论已经超出了本书讨论的范围，因此，不再深入讨论，笔者推荐两部著作供读者学习。在类型理论领域，Benjamin C. Pierce 撰写的《Types and Programming Languages》和《Advanced Topics in Types and Programming Languages》是公认的经典著作，前者有中文译本《类型与程序设计语言》，而后者目前暂时没有中文译本。有兴趣的读者可以参考阅读。

4.3.7 变量声明部分

Pascal 的变量是相对比较简单，仅支持全局变量和局部变量两类。Pascal 不允许用户声明块内变量（例如，C 语言中可以将变量声明置于某一对大括号内，变量的生存期则仅限于大括号所示的程序块内）。虽然块内变量的优越性显而易见，但是标准 Pascal 并不支持，因此可以忽略。

有了先前的基础，变量声明部分的语义处理将是比较简单的。从变量信息表的结构来看，不难发现，变量信息中需要填写的属性并不多（一共只有 6 个）。实际上，对于用户声明的变量而言，m_eBank、m_bRef 两个属性的取值是固定不变的，总是“VAR”和



“false”，因为这两个属性主要适用于一些形参变量或临时变量。而 `m_iMemoryAlloc` 也是存储分配相关的属性，因此也不必考虑。故需要补充填写的属性只有 3 个。下面先介绍变量声明部分相关的文法及其主要的语义动作。

【文法 4-13】

变量声明部分 → var 变量定义 ; 变量定义列表
变量定义列表 → 变量定义 ; 变量定义列表 | ε
变量定义 → 013 标识符列表 040 : 类型

根据前面的经验，很容易得到以下结论：变量声明部分的主要语义动作必定存在于非终结符“标识符列表”相关的产生式中，而 `semantic013`、`semantic040` 的主要工作就是标志栈的管理。下面先来看看 `semantic013`、`semantic040` 的源代码。

程序 4-33 semantic.cpp

相关文法：

变量定义 → 013 标识符列表 040 : 类型

```
1 bool semantic013()
2 {
3     iIdListFlag.push(6);           //将“6”压入 iIdListFlag 栈
4     return true;
5 }
```

第3行：将“6”压入 `iListFlag` 栈，便于推导“标识符列表”时进行相关语义判断。

程序 4-34 semantic.cpp

相关文法：

变量定义 → 013 标识符列表 040 : 类型

```
1 bool semantic040()
2 {
3     TypeInfo Tmp;
4     Tmp.m_szName="_noname";
5     Tmp.m_szName.append(GetSerialId());
6     Tmp.m_iProcIndex=SymbolTbl.ProcStack.top();
7     Tmp.m_iLink=-1;
8     SymbolTbl.TypeInfoTbl.push_back(Tmp);
9     iTypePos.push(SymbolTbl.TypeInfoTbl.size()-1);
10    iIdListFlag.pop();
11    return true;
12 }
```

第 4、5 行：设置临时类型信息表项的 `m_szName` 属性。实际上，变量声明中的类型分为两种情况：非匿名类型与匿名类型。所谓非匿名类型，就是指变量的类型是一个用户自定义类型，例如：

【声明 4-21】

```

TYPE
    AA=ARRAY [1..10] OF INTEGER;

VAR
    A,B:AA;

```

AA 并不是 Pascal 的标准类型，而是一个用户自定义类型的名字。除了非匿名类型之外，其他的形式都可以看作匿名类型。当变量声明中的“类型”是一个匿名类型时，其语义动作非常简单，只需建立一个类型结点供后续“类型”相关的语义子程序填写信息即可。对于非匿名类型而言，其语义动作就稍有不同。根据符号表及类型链的设计，比较好的处理方式是直接将变量信息表项的 `m_iTypeLink` 指向该用户自定义类型的类型链的首结点。由于这两种处理方式存在一定的差异，可能给语义子程序的设计带来小小的不便，为了便于设计与编码，笔者将两者的处理方式进行了统一，即都采用前者的处理方案。这样的改变也是有代价的，就是会造成非匿名类型的变量描述存在一个冗余的类型结点，不过，这丝毫不会给编译器分析源程序带来任何障碍。当然，通过修改设计方案完全可以避免此事的发生，从而使之比较完美，但是程序设计的复杂度却可能有所增加，效率也会降低。

第 7 行：`m_iLink` 属性初始化为-1，而其实际值将由类型声明相关的语义子程序来填写。

第 9 行：设置 `iTypePos` 栈，即当前类型结点，以便后续“类型”相关的语义子程序处理。

下面，再来看看 `semantic012` 是如何生成变量信息表项的。

程序 4-35 semantic.cpp

相关文法：

```

标识符列表 → 标识符 012 标识符列表 1
标识符列表 1 → , 标识符 012 标识符列表 1

```

```

1  bool semantic012()
2  {
3  .....
4      if (iIdListFlag.top()==6)
5      {
6          VarInfo Tmp;
7          Tmp.m_eRank=VarInfo::VAR;
8          Tmp.m_iProcIndex=SymbolTbl.ProcStack.top();
9          Tmp.m_iTypeLink=SymbolTbl.TypeInfoTbl.size();
10         Tmp.m_szName=TokenList.at(iListPos-1).m_szContent;
11         if (SymbolTbl.SearchTypeInfoTbl(SymbolTbl.ProcStack.top(),Tmp.m_szName,false)==-1 &&
12             SymbolTbl.SearchConstInfoTbl(SymbolTbl.ProcStack.top(),Tmp.m_szName)==-1 &&
13             SymbolTbl.SearchLabelInfoTbl(SymbolTbl.ProcStack.top(),Tmp.m_szName)==-1 &&
14             SymbolTbl.SearchEnumInfoTbl(SymbolTbl.ProcStack.top(),Tmp.m_szName)==-1 &&
15             SymbolTbl.SearchVarInfoTbl(SymbolTbl.ProcStack.top(),Tmp.m_szName)==-1 &&
16             SymbolTbl.SearchProcInfoTbl(Tmp.m_szName)==-1)
17         {
18             SymbolTbl.VarInfoTbl.push_back(Tmp);
19         }
20         return true;

```




```
20         }
21     else
22     {
23         EmitError(Tmp.m_szName.append("标识符名已经存在"),TokenList.at(iListPos-1));
24         return false;
25     }
26 }
27 .....
28 }
```

第 7 行：设置 `m_eRank` 属性。这里，只需考虑用户变量声明的情况，故统一将 `m_eRank` 属性设为 `VAR` 即可。

第 9 行：设置 `iTypeLink` 属性。由于无论是匿名类型还是非匿名类型，`semantic040` 都会在表尾追加一个类型信息表项，因此这里只需将 `iTypeLink` 指向表尾信息项即可。

第 10 行：获取变量名字。

第 11~25 行：检查变量名字是否有效。根据实际情况，依次检索符号表即可。

4.3.8 过程、函数声明部分

过程、函数也是程序设计语言最基本的语法元素之一。而 Pascal 中的过程就是指无返回值的函数。由于 Pascal 没有类似于 C 语言的 `void` 类型，因此，Wirth 教授提出了过程的概念以区别于普通的函数。实际上，从严格意义上来说，过程与函数还是存在一定差别的。例如，在 Pascal 中，过程体内严格禁止存在对 `RESULT` 的赋值，而函数体内却必须存在 `RESULT` 的赋值。这些语义在 C 语言中并不是特别强调的，C 语言对于 `return` 语句的限制比较灵活。即使如此，很多人仍然质疑 Pascal 过程的作用，认为它的存在并没有太大的意义。当然，笔者也并不否定这个观点。确实，拥有类似于 C 语言的函数机制已经足以满足相关应用的需求了，过程作为一种特殊函数的存在价值并不大。不过，从设计编译器的角度而言，笔者非常尊重 Wirth 教授的设计思想，同时为了尽可能兼容标准 Pascal，故仍保留过程机制。需要指出的是，在计算机科学中，过程与函数的概念并没有非常明确的界限。许多著作经常使用“过程”泛指一切子程序。因此，在后续章节中，在没有明确说明的情况下，过程与函数是不作区分的。

这里，笔者从三个方面对 Pascal 的过程、函数声明作一个简单的回顾，以便进一步讨论相关语义子程序的实现。

(1) 参数传递方式。习惯上，参数传递方式可以分为三种：值传递、址传递、引用传递。其中引用传递可以视为址传递的一种特殊实现，因此，可以认为其与址传递是一脉相承的。读者最熟知的“引用传递”应该就是由 C++ 实现的，在 C++ 中，引用传递与传统的址传递（在 C++ 中可以通过传递指针实现）的形式还是存在一定差异的。使用址传递方式时，由于该参数是一个指针，程序员在使用该参数时，必须通过指针运算才能获取实际所需参数值。而使用引用传递方式时，程序员不必考虑许多指针运算方面的问题，可以像使用实参一样，直接访问形参，而不必在意其指针的本质。对于程序员而言，引用传递就像是直接将实参的名字传递给了形参，可以像引用实参一样引用形参，所以引用传递亦称为名字传递。Pascal、C++、Java 都支持两种参数传递方式，即值传递、引用传递。而 C 语言仅支持值传

递。最后，笔者需指出一点，在程序设计语言领域，关于引用传递方式实际上是存在一定争议的。支持者固然是大多数，但反对者的理由也并非无中生有，他们认为引用传递可能会使程序员忽略调用函数可能产生的副作用。C#语言中加入了 `out` 方式的参数。除了一些很细小的差别之外，`out` 方式的内核与引用方式是基本类似的。

(2) 扩展信息描述。除了参数、返回值类型等信息外，实际上，过程、函数声明中还可能包含一些其他辅助信息，这些信息大多与外部函数相关。事实上，除了用户自定义过程、函数之外，还有两类函数可供用户调用，即库函数、系统函数。库函数既可以是编译器提供的预定义库函数，也可以是用户编写生成的自定义库函数。而系统函数主要指的是操作系统提供的一些系统调用或系统接口函数。库函数和系统函数都是以二进制形式存在的，它们并不需要重新编译生成。由于这些函数都是独立于本项目而存在的，因此通常将其称为外部函数。注意，关于内部、外部函数的分类标准并不统一，这里的说法仅限于本书。当然，使用外部函数的优势非常明显，本书不可能一一列举，只能简单提几点。例如，抽象与封装、功能扩展、实现跨语言共享等。而这里提到的扩展信息主要是指那些与外部函数相关的信息，例如，外部函数的来源、调用方式等信息。下面，就针对这些信息作简单介绍。

外部函数的来源一般就是指外部函数所属的库文件名及路径等信息。外部函数是以库文件形式组织的，一个库文件通常可以包含多个外部函数，例如，读者熟知的 `DLL` 文件、`Lib` 文件皆是如此。从理论来说，用户在本项目中调用某一外部函数时，必须告知编译器所调用的外部函数所属库文件的准确路径，以便编译器生成正确的目标程序。不过，这不是绝对的。在实际情况下，来源仅仅指的就是库文件名字而已，而库文件路径信息一般是预定义的，由编译器或操作系统自动检索识别。例如，在 `Windows` 中，`DLL` 文件的路径就是由操作系统自动检索的，`Windows` 定义了一套明确的检索顺序。虽然来源信息并不复杂，却是非常重要的，不正确的来源信息会导致编译链接出错，甚至目标程序异常。

外部函数的调用方式主要就是讨论一个话题，即传参约定。对于支持系统栈的目标机器来说，通常都是借助于系统栈传参的，调用点通常将参数逐一压入系统栈，被调函数从系统栈内获取参数信息。传参约定的目的就是便于访问与实现外部函数，为外部函数提供一个统一调用接口是非常必要的。实际上，函数调用是一次协作的过程，调用点及被调函数双方必须遵守同样的传参约定才能保证被调函数正确获取实参。对于调用本项目的函数、过程，由于调用点与被调函数都是由同一个编译器生成目标代码的，编译器设计者完全可以定义一种内部的传参约定供双方遵守，唯一的目标只要保证正确传参即可。不过，当被调函数是外部函数时，编译器就无法作类似的内部约定了，因为外部函数已经是二进制代码形式存在了，不可能对其作任何改变。再者，由于编写外部函数的源语言、所使用的编译器都是未知的，如果传参约定完全是由语言及编译器设计者自定义的，那么，其随意性将导致外部函数失去存在的价值。当然，计算机科学家并没有否定各种新颖的内部传参约定的存在意义，也不限制其使用范围。在这种情况下，早期的一些语言及编译器设计者仅对外部函数的传参方式作了一些约定，规定任何编译器生成外部函数、调用外部函数相关目标代码时，必须遵守相关的约定，否则无法保证调用的正确性。但这些约定并不影响各种内部函数的传参方式。例如，读者熟悉的 `stdcall`、`cdecl`、`fastcall` 就是最常见的传参约定。下面，笔者就来谈谈传参方式约定的相关组成。

(1) 实参压栈的顺序。实参压栈的顺序直接关系到被调函数访问的正确性。栈式结构的特点就是先进后出，如果调用点以顺序方式将实参压栈，则被调函数将以逆序方式访问参



数。为了便于被调函数顺序访问实参，通常调用点是以逆序方式压栈的。逆序压栈得到了绝大多数编译器的共识，几乎不存在任何分歧。

(2) 栈复位点。试图借助于系统栈传参，就必须保证函数调用前后系统栈的情景一致，否则程序可能产生异常。这里，有个关键问题需说明：栈式传参的过程与普通的栈应用可能存在一定的差别。虽然传参方式的通常理解是一个实参赋值给形参的过程，但事实却并非如此。根据栈式传参方式的观点，调用点将实参压栈后，被调函数直接通过计算栈基址偏移访问参数，并不需要将参数出栈，或作其他任何操作。因此，从理论上而言，直至被调函数返回并不会改变系统栈的状况，也就是说，调用函数时压入的参数仍存在于系统栈内。如此便使得函数调用前后的系统栈的状况发生了变化，这是谁都不愿意看到的，因为这可能导致函数返回异常。故目标程序就不得不将原来因函数调用需要而压栈的参数弹出，即栈复位。栈式传参方式需要做栈复位的工作是得到普遍共识的，并不存在任何异议。关键问题是由谁来完成栈复位工作。实践证明，调用点及被调函数都是可以完成这一工作的，某些观点认为应由前者完成，而另一些则支持由后者完成。例如，WINAPI 的标准调用方式 `stdcall` 规定栈复位由被调函数完成。而 C 语言最常用的调用方式 `cdecl` 则规定栈复位由调用点完成。关于这个问题，笔者觉得并不存在本质区别，只是实现方式不同而已。

(3) 各种类型数据以何种形式压栈。高级语言与汇编语言的一个重要区别就是丰富的数据类型，如数组、结构、字符串等。程序是将这类变量的首地址压栈，还是将整个存储块压栈，这是一个值得商榷的问题。

至此，简单讨论了函数传参问题，旨在说明外部函数调用方式的特点，关于更多的传参话题将在后续章节中详述。关于函数、过程的符号表处理的问题，与变量、常量类似，这里就不再详细讨论了。

4.4 深入学习

符号表与声明部分的语义处理一直是编译器实现中两个比较复杂的话题。遗憾的是，即使是“龙书”，关于符号表设计的篇幅也非常有限，更多关注的是符号表的检索效率，并没有涉及符号表的描述能力的相关讨论。

这里，笔者向读者推荐《高级编译器设计与实现》（即“鲸书”）第 3 章，其中，比较详细地阐述了符号表设计中的一些高级话题，尤其是关于描述能力方面的观点是值得学习的。至于符号表实例分析，最经典的莫过于《可变目标 C 编译器——设计与实现》了。

哈希技术也是符号表设计的另一个重要话题，1973 年，Knuth 给出了哈希方法的详细分析。当然，在此之后，人们还提出了许多优秀的哈希函数设计观点。

- | | | |
|-------------------------------------------------------------|---------------------------------------|---------|
| 1、计算机程序设计艺术 | D.E.Knuth | 清华大学出版社 |
| 说明：这本书是算法设计的经典著作，1999 年被列为 20 世纪 12 部最佳学术专著之一。 | | |
| 2、编译原理 | 吕映芝，张素琴 | 清华大学出版社 |
| 说明：书中提到了符号表的结构设计问题，并评价了单表、多表等结构的优劣，也是国内教材中的经典之作。 | | |
| 3、高级编译器设计与实现 | Steven S.Muchnick | 机械工业出版社 |
| 说明：这本书被誉为“鲸书”，其中涉及许多编译器设计中的高级话题，当然，它最值得关注的就是优化技术。 | | |
| 4、可变目标 C 编译器——设计与实现 | Christopher W. Fraser, David.R.Hanson | 电子工业出版社 |
| 说明：该书所描述的 lcc 是一个值得读者深入学习的实例系统，它的很多设计思想被广泛应用于许多相关领域，如操作系统等。 | | |
| 5、算法导论 | T.H.Cormen | 机械工业出版社 |
| 说明：这本书是 MIT 数据结构与算法设计课程的教材，其经典程度足以与《计算机程序设计艺术》媲美。 | | |

4.5 实践与思考

1. 请读者使用更优的哈希算法改写 Neo Pascal 的符号表系统。
2. 根据 C 语言的特点，试设计一套完整的符号表系统，用于描述 C 语言的符号结构。
3. 符号表的输入、输出也是非常重要的话题，尤其是在生成调试信息时，经常要涉及相关应用。请读者设计一组输入、输出接口，以便 Neo Pascal 的符号表能够以文件形式存储。
4. 在现代编译器中，有时把复数类型视为一个原子类型，试问该如何修改 Neo Pascal，以实现复数数据类型？
5. 如果 Neo Pascal 需要支持类似于 C 语言的块内变量声明，那么应该如何修改符号表结构？

4.6 大师风采——John Backus

John Warner Backus: 美国计算机科学家，Fortran 语言创始人之一，BNF 发明者之一。1924 年 12 月 3 日出生于美国费城。1943 年，进入弗吉尼亚大学主修化学，中途参军。战后移居纽约，对数学发生了兴趣，并进入了哥伦比亚大学学习数学，于 1949 年获硕士学位。

1950 年，Backus 加入 IBM 公司，成为一名程序员。在 IBM 公司期间，Backus 的第一个主要项目是设计一个用于计算月球位置的程序。

1953 年，他开发了基于 IBM 计算机的第一个高级语言——Speedcoding。1954 年，Backus 和他的团队为 IBM 704 定义与开发了 Fortran 语言，这是人类历史上第一个广泛应用的高级语言。Fortran 语言的诞生将程序编译技术正式推上了历史的舞台。

20 世纪 50 年代后期，Backus 致力于设计 Algol 58 和 Algol 60 语言。在此期间，Backus 提出了著名的 Backus-Naur 范式 (BNF)，在程序设计语言、形式语言等研究领域，这是具有里程碑意义的。Backus 也因此获得了 1977 年的图灵奖。

在此之后，Backus 开始关注函数式语言的研究。在他职业生涯的后期，主要的研究项目就是 FL (Function Level)。他提出的许多理论与概念被后来的 J 语言 (1990 年由 Ken Iverson 与 Roger Hui 开发) 实现了。

2007 年 3 月 17 日，Backus 病逝于美国阿什兰的家中，享年 83 岁。

第5章 中间表示

The initial motive for developing APL was to provide a tool for writing and teaching. Although APL has been exploited mostly in commercial programming, I continue to believe that its most important use remains to be exploited: as a simple, precise, executable notation for the teaching of a wide range of subjects.

——Kenneth Eugene Iverson

5.1 IR 概述

5.1.1 IR 的作用

中间表示，通常称为 IR（Intermediate Representation，简称 IR），是编译器设计中的另一个重要话题，它是编译器前端的产物。在编译技术发展历程中，IR 的提出具有里程碑的意义，它是划分编译器前、后端的一个重要标准。IR 的应用使得编译器可以方便地适应源语言、目标语言的变化与扩展。在理想状态下，与源语言相关的变化只需改变编译器的前端，而与目标语言或目标机相关的变化只需修改编译器的后端即可。图 5-1 是两种经典编译器的模型示意图。

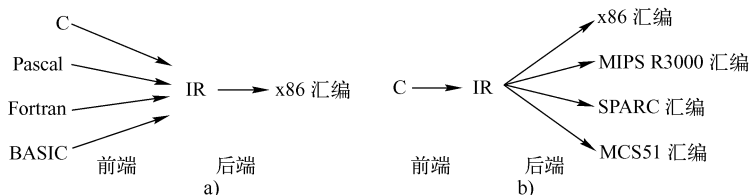


图 5-1 编译器前、后端模型示意图

a) 模型 A b) 模型 B

模型 A 允许输入多种不同源语言编写的程序，将其翻译成同一目标代码。为了便于源语言的扩展，通常前端将其编译生成一种具有一定通用意义的 IR。而后端（例如，优化、目标代码生成等）并不需要理会编译器到底存在几种不同的源语言，对于后端而言，唯一的源语言就是 IR。VS .NET 就是基于这种模型的编译器，只不过它的后端是 .NET Framework 的动态编译框架，而前、后端的接口就是 IL。

模型 B 描述的是一种常见的编译器，称为“可变目标编译器”，即编译器的后端可以生成不同的目标代码。使用这种编译器的源语言编写程序的优点在于可以方便地将同一个程序移植到各种目标机上运行，用户不必过多关注各种目标机体系结构的细节。不过，需要指出的是，可变目标编译器并不是读者所熟悉的跨平台编译器。可变目标编译器关注的是目标机的变化，而不仅仅是操作平台的变化。当然，在现代编译领域，这种模型也用于解决目标程

序跨平台的问题。

在实际应用中，还存在一种更为特殊的模型，它可能存在多种源语言与目标语言。这类编译器的 IR 可能更像是一种标准接口，分别提供给前、后端使用。事实上，能够完美实现这一目标的经典编译器并不多见，可能唯有 GCC 才当之无愧，GCC 支持 C、C++、Java、Fortran、Pascal 等多种源语言及 i386、MIPS、ARM 等数十种目标机。

最后，笔者有必要指出，在很多情况下，编译器前、后端之间并不存在非常明确的界线。尤其是源语言的专业性比较强时，前、后端的界线可能是非常模糊的。因为，在这种情况下，编译器设计者没有必要过多考虑源语言、目标语言的横向扩展，当然，也没有必要为此付出巨大的精力。

5.1.2 IR 设计及其级别

一个实际编译器的 IR 通常是由该编译器设计者定义且仅供该编译器使用的一种语言，对于编译器的用户及程序设计语言本身都是透明的。除了一些特殊情况外，编译器的 IR 是不会公开的。在笔者看来，将 IR 视作一种资源或者财富是绝不为过的。编译器后端的算法设计及其性能很大程度上都是依赖于其 IR 的，因此，除了一些开源或者早期实验室级的编译器外，很少有商用编译器愿意公开其 IR。

即便如此，关于 IR 设计可以借鉴的经验还是比较多的，也有一些设计原则可以指导编译器设计者定义 IR。不少编译器设计大师都认为定义 IR 是一门艺术而不是科学，因为其优劣在很大程度上是无法使用非常科学或精确的方法来评价的。根据对经典编译器的理解并结合一些大师著作的观点，笔者总结得到如下几点建议，供读者参考。

(1) 充分考虑目标机体系结构。IR 的主要服务对象就是编译器后端，因此设计 IR 必须充分考虑目标机体系结构。当然，目标机的指令系统是其中一个最重要因素，但并不仅仅局限于此。例如，存储结构、总线结构、中断系统等也是比较重要的。不过，必须指出一点，充分考虑目标机体系结构，并不等同于过多依赖于目标机体系结构。恰恰相反，通常 IR 不应该包含太多目标机的细节，否则移植性、通用性都会大大降低。例如，有些目标机的指令系统不支持乘、除法，如果因此编译器的 IR 也不包含乘、除法操作，笔者认为这是完全没有必要的。再比如，在定义 IR 时，可以对寄存器分配的方案加以考虑，但不应该细化到某一种目标机的具体寄存器（如 i386 的 EAX~EDX 等）。简而言之，应充分考虑目标机的共性，舍弃目标机中过于个性的细节，除非是针对某一特定的目标机而设计专用编译器。

(2) 兼顾优化算法的需要。优化技术是现代编译技术中最为复杂的话题之一，优化算法众多，其中一部分优化算法是基于 IR 实现的，即优化算法的输入、输出都将是 IR。然而，正所谓众口难调，由于各种优化算法所面临的问题和解决的方案都存在较大的差异，它们对于 IR 的需求也不尽相同。例如，有些优化算法希望输入的 IR 尽可能高级，即比较接近于源语言形式，最大程度上保留源语言的特性。而有些优化算法则希望输入的 IR 尽可能低级，即更接近于目标语言形式，以便优化过程中更多地考虑目标机的特性。

在一些编译原理教材中，经常以经典的五个阶段为基础讨论编译技术。这样可能会给读者造成一个误区：编译过程中只存在一种 IR 形式。实际上，这种说法通常是错误的，因为一个编译器使用多种 IR 的情况并不罕见。设置多种 IR 的目的就是为了更好地适应各种优化算法的需求。一些商用编译器为了尽可能改善目标代码的性能，不惜生成多遍 IR 以适应各



种算法的需要。当然，这个过程通常是串行的，以免编译过程对存储空间资源消耗过大。

(3) 结构不宜过于复杂，应具有一定灵活易变性。根据经验，一些经典 IR 的逻辑结构都是比较简单的，如数组、树、DAG 等。这样设计的主要目的有两个：第一，IR 经常会以文件形式存储到外存中，供其他程序使用。逻辑结构过于复杂可能会给读取、存储增加难度，而且影响处理效率。第二，IR 往往不是一成不变的，随着后端设计的细化，有时 IR 会有一些变化。通常，很难认为这种情况是由于初期设计不周所致，毕竟编译器设计的难度要远高于普通应用系统。从一些编译器大师的回忆录中不难发现，即使是 Turbo Pascal、lcc、Delphi 等经典编译器也是如此。因此，在这种情况下，能做的仅是使 IR 适应后端的需要。

(4) 表达能力必须充分。前面已经讲过，编译过程就是将源语言程序等价变换成目标语言程序的过程，其中最核心的要点就是“等价”。如果翻译过程导致或者可能导致源语言程序与目标语言程序之间存在语义差异，那么本次编译过程是失败的。这也是编译优化技术讨论的前提，任何优化算法都不能违背这个原则，否则宁可不用，也不能使编译器存在任何隐患。IR 是编译器后端的输入，也就是说，编译器后端可能已经无法知晓源语言程序的真实语义了，只能从输入的 IR 了解源语言程序的语义信息。在这种情况下，IR 应该足以表述源语言所允许的任何语义，否则一定会违背“等价”原则。举个比较极端的例子，源语言存在 IF、FOR 等结构，而 IR 只允许存在顺序结构。那么，无论使用怎样的翻译方案也无法保证“等价”原则。

当然，参考一些现存的经典 IR 也是一种不错的选择，如 GCC 的 RTL、SSA、lcc 的 DAG、Sun SPARC 的 SunIR、GEM 的 CIL 和 EIL、Intel 的 IL 等。针对当前的源语言与目标机，虽然现存的 IR 未必是最优的解决方案，但是它们可以让一些经验不足的设计者方便地构造一套较为完整的 IR。在这种情况下，设计者需要更多考虑这种 IR 对目标编译器的各种适应性问题及实现它所需付出的代价。

下面简单介绍一些常用 IR。按 IR 的逻辑结构，通常可以分为如下几种：语法树、后缀表达式、DAG、三地址代码等。除了三地址代码之外，读者应该对提到的其他逻辑结构并不陌生，它们都是常见的数据结构。这里，笔者就不再给出其具体形式，只想就这些 IR 的特性简单评说几句，供读者参考。

语法树、DAG（有向无环图）：这两种形式者是比较直观的，在计算机科学中，基于树、图的算法资源比较丰富，关于它们的存储结构也存在许多经典的方案可供参考。语法树形式能较好地保存源程序的结构，所以常作为自下而上的语法分析过程的暂存数据结构。而 DAG 更多地被作为三地址代码的一种变体形式，在某些优化算法中，DAG 有着其他形式所不具备的优势。

后缀表达式：也称为逆波兰表达式，这种形式简单明晰，便于存储。在处理表达式翻译时，后缀表达式有着其他形式无法比拟的优势。不过，由于后缀表达式的应用领域比较单一，所以很少独立作为一个实际编译器的 IR 存在。

三地址代码：也称为“四元组”，即操作符和三个操作数地址。这是一种最为常见的 IR。甚至有些书籍认为 IR 就是中间代码（即三地址代码）。笔者认为这一说法并不是非常准确。从形式上而言，三地址代码与汇编代码比较类似，都是以代码列表形式存在的。所谓的三地址，指的就是每一行代码通常包含三个地址信息，即操作数 1、操作数 2、结果操作数。例如，(ADD A, 1, C) 这行三地址代码的含义就是 $A+1 \rightarrow C$ 。这种形式初看与汇

编语言有点类似，但是在生成三地址代码时，编译器并不需要计算操作数的实际地址，只是以标号形式给出即可。另外，读者也不必苛求三地址代码与汇编指令格式的统一，这是没有意义的。有些书上也提到了二地址代码（三元组）的形式，它和三地址代码比较类似，只是省略了结果操作数地址，而是将操作结果存入操作数 1 或操作数 2 的存储空间内。实际上，二地址代码形式可能更贴近于 x86 汇编指令的格式，但是它的通用性却不及三地址代码，因此并不常用。三地址代码的优点是便于编译器生成目标代码，编译器不必为复杂的 IR 结构的转换付出高昂的代价，而且也基本上能满足常用优化算法的需要。当然，三地址代码也不是完美的，由于它是相对离散的，在分析源程序结构方面，它就不及语法树便捷。Neo Pascal 的 IR 也是三地址代码，在后续章节中，笔者还将详细讲解。

最后，再来谈谈 IR 的级别，即 IR 依赖于目标机的程度。按级别分类，可将 IR 分成三类：高级形式（HIR）、中级形式（MIR）、低级形式（LIR），也可称为高级中间语言、中级中间语言、低级中间语言。

高级形式（HIR）是一种尽可能保持了源语言程序结构的 IR，这种形式能较好地保留源程序的原始语义信息。由于高级形式太接近源语言程序结构，所以很少有编译器将其独立作为 IR 传递给后端。

中级形式（MIR）既要以一种与语言无关的方式在一定程度上反映源语言的特性，又要能够适应多种体系结构的 IR。中级形式是一种比较常用的 IR，它兼顾了源语言、目标机的特性，又能适用于大多数优化算法。当一个编译器仅设计一种 IR 时，中级形式是较理想的选择。

低级形式（LIR）就是在一定程度上包含某些目标机特性的 IR，比目标语言稍高级，常作为一些机器相关的优化算法的输入。不过，实际上，除了一些较大型的编译器需要使用低级形式之外，低级形式并不是很常见。因为更多编译器设计者更愿意直接基于目标语言作优化。

级别的概念在国内编译原理书籍中并不多见，不过，笔者却觉得这种分类比较科学，它揭示了 IR 的核心。有些国内编译原理书籍在讲述 IR 时，过份强调其逻辑结构（例如，后缀式、树、DAG、三元组等）。实际上，逻辑结构只是数据结构的讨论范畴而已，并不是设计 IR 的核心，从编译器设计角度而言，应该更关注 IR 的级别。在此，笔者不想过多讨论 IR 的各种逻辑结构，这可能是徒劳的。因为，在笔者看来，这个话题或许会让读者对 IR 产生错误的理解，认为优秀的 IR 所采用的逻辑结构就是书上所提及的那几种。实际上，出于对整体架构的考虑，一些经典编译器的 IR 可能是很奇特的，根本无法将其严格归类。即使如此，它却依然是经典的。例如，GCC 的 RTL，有些书坚持认为 RTL 是三地址代码，但笔者认为这种观点值得商榷，毕竟它与三地址代码的形式相差甚远。

5.1.3 设计 IR 的重要意义

IR 在编译器设计中的地位与作用是极其重要的。不过，正如前面所提到的，随着“端”概念的出现，有些设计方案认为编译器设计者只需设计一个编译器的前端，将编译产生的 IR 作为 GCC 之类的编译器后端的输入，由后端完成将 IR 编译成目标代码的过程。对于一些编译技术爱好者而言，这种方法的优点是显而易见的，可以大大降低编译器设计者的工作量。例如，对于熟悉 GCC 接口的读者而言，联系本书第 2~5 章相关前端理论与技术，构造基于 GCC 的



编译器前端应该并不算太困难。因此，有些观点认为：随着 IR 的日趋标准化、规范化，在现代编译技术中，研究 IR 已经意义不大了。不过，笔者并不认同，原因有如下两点：

(1) 从研究与学习编译技术的角度而言，这种方法无法对编译器作全面了解。实际上，这种方法只完成了编译器的前端，后端的核心技术并不是掌握在设计者的手中。即使是 GCC 这样的开源后端，鉴于其庞大的规模，一般读者也很难完全理解其设计核心的。这种情况并不利于理解编译器的核心技术，也不利于针对目标机的某些特性进行后端的修改或重构。在实践中，这种情况并不少见，虽然 GCC 的设计堪称完美，提供了灵活性很高的目标机描述文件（即 md 文件），但是针对不同目标机的一些奇怪特性，即使 Stallman 恐怕也无法预知。这就使得编译器设计者不得不对 GCC 的后端作一些个性化的修改，在不能完全理解 GCC 后端的情况下，修改工程将是非常复杂与危险的。

(2) 从企业行为的角度而言，这种方法的可行性不大。一般来说，商用编译器的源代码及相关文档必定是开发商最为核心的技术资料，开发商通常不会轻易公开。在开源运动（Open Source）热火朝天的今天，开发商能承受的底线可能也就是免费提供编译器而已。

Neo Pascal 的设计目标是完成一个完整的编译器，而不仅仅是一个编译器的前端，因此，笔者还是倾向于自行设计一套 IR 供 Neo Pascal 使用。当然，笔者水平有限，不可能设计出 RTL、SSA 之类的经典形式。Neo Pascal 的 IR 只是一个可行的方案而已，并不苛求完美。关于 RTL、SSA 的话题，将在第 10 章中详细讨论。

5.2 IR 生成

5.2.1 三地址代码概述

前面已经提到了三地址代码的概念。在众多 IR 中，三地址代码是最为常用的。由于 Neo Pascal 的 IR 也是一种三地址代码的形式，因此有必要详细讨论这种 IR。所谓的“三地址”指的就是两个运算分量（操作数 1、操作数 2）及目标操作数三个对象的地址。例如，(ADD 5, A, B) 的含义是 $5+A \rightarrow B$ ，其中“ADD”称为操作码，而 5 和 A 是两个运算分量，B 是目标操作数。注意，三地址代码的书面表示形式并不唯一，本书将使用“四元组”的形式表示三地址代码。

计算机科学家提出三地址代码的理由如下：三地址代码是一种线性 IR。由于输入源程序及输出目标程序都是线性的，因此，线性 IR 有着其他形式无法比拟的优势。另外，相对于其他表示形式而言，程序员对于线性表示形式通常会有一种莫名的亲切感，编译器设计者当然也不例外。早期编译器设计者往往都是汇编语言程序设计的高手，可以非常自然、流畅地阅读线性的三地址代码形式。同时，线性表示形式也会降低输入输出的实现难度。随着编译器“端”、“遍”等概念的出现，IR 已经不仅仅是一种存储在内存中的数据结构。有时它也需要以文件形式转存输出，作为接口供其他系统读取使用。

那么，一定有读者会心存疑问：为什么将其设计为“三地址”的形式呢？实际上，这是计算机科学家经过多年实践探索后才得到共识的。三地址代码并不是唯一的线性 IR，只能说是最为常见的而已。在编译技术领域，二地址代码、单地址代码（即栈式机代码）都曾出现过，也曾在某些应用领域盛行一时，尤其是单地址代码。

二地址代码比较简单，就是选择其中一个对象同时充当运算分量与目标操作数。在早期，二地址代码主要就是着眼于 x86 机器而提出的。不过，实践证明，这只是人们的一厢情愿而已，即使是针对 x86 机器，二地址的优势也并不明显，它反而可能会给编译器带来一定的麻烦，所以这种表示形式已经逐步被淘汰了。

然而，单地址代码的情况则截然不同了，在现代编译器设计中，单地址代码也是应用比较广泛的一种 IR。尤其是近年随着混合语言的日渐壮大，单地址代码也重新进入了人们的视野。由于执行单地址代码程序的栈式机架构相对比较简单，可以非常方便地构造相关的解释器或虚拟机，所以单地址代码深受混合语言设计者的欢迎。读者熟悉的 Java 字节码、.NET 的 IL 都是单地址代码。栈式机或者单地址代码与常见的 x86 体系结构相差甚远，可能读者所知不多。不过，单地址代码还是一种比较有意思的表示形式，因此，笔者想通过一个简单的实例让读者对单地址代码有所了解。

例 5-1 单地址代码实例。

表 5-1 是一个 .NET IL 汇编的实例，左侧是 C# 的源程序片段，右侧是 IL 汇编的片段（使用 `ildasm.exe` 将可执行文件反汇编即得到该结果）。当然，注释是笔者事后加上的。

表 5-1 IL 汇编实例

C#源程序片段	汇编片段
.....	.maxstack 2
int a, b;	.locals init ([0] int32 a,
a = 1;	[1] int32 b) //局部变量列表, 0、1 为编号
b = 2;	IL_0000: nop
a = a + b;	IL_0001: ldc.i4.1 //将立即数 1 压入计算栈
.....	IL_0002: stloc.0 //将计算栈顶元素置入 0 号变量(a)
	IL_0003: ldc.i4.2 //将立即数 2 压入计算栈
	IL_0004: stloc.1 //将计算栈顶元素置入 1 号变量(a)
	IL_0005: ldloc.0 //将 0 号变量压入计算栈
	IL_0006: ldloc.1 //将 1 号变量压入计算栈
	IL_0007: add //将栈顶两个元素出栈相加, 并将结果压入计算栈
	IL_0008: stloc.0 //将栈顶元素置入 0 号变量(a)
	IL_0009: ret //返回

某些读者可能对 IL 汇编语言比较陌生。相信无论熟悉与否，借助笔者的详细注释，要读懂这段简单的 IL 汇编程序并不困难。设置这个实例的目的只是让读者了解一下栈式机及单地址代码的形式。这里，笔者必须澄清一点：实际上，IL 汇编语言并不简单，读者千万不要被本例误导。当然，对有志于深入研究 .NET 编译器的读者而言，学习 IL 汇编语言还是非常有必要的，它是深入研究 .NET 编译器的基础。

三地址代码是在二地址代码的基础上发展而来的。二地址代码的不足之处在于它通常会给其中一个源操作分量带来一定副作用。当然，这种设计的灵感最初是来源于 x86 指令系统的，但是却忘了一个重要的区别：x86 指令中往往都是以寄存器作为暂存空间的。而暂存空间对于二地址代码却是一个棘手的问题。为了解决二地址代码的不足，人们提出了一个对源操作分量不产生任何副作用的形式，那就是三地址代码。也就是说，在一行三地址代码中，任何运算都不会改变两个源操作分量。这是三地址代码与二地址代码的主要区别。这个特性是非常重要的，它将使得编译器更自由地复用名字与值，不必考虑代码带来的副作用。

一般来说，三地址代码的大多数操作都是由四项组成，即一个操作码和三个地址。不过，三地址代码同样存在级别差异。随着语言复杂性的提高，在现代编译器设计中，三地址



代码的级别概念显得尤其重要。根据编译器设计的需要，有些三地址代码可能近似于源语言，而有些三地址代码则更接近于目标语言。当然，级别主要就是取决于三地址代码的操作符及操作分量的复杂性。下面，笔者就操作符及操作分量这两个话题来讨论三地址代码。

操作符是用于标识三地址代码操作含义的元素。根据源语言、目标语言的特点，三地址代码操作符的集合以及抽象程度是各不相同的。其中，抽象程度是三地址代码设计中的重要因素之一。一般而言，三地址代码将包含大部分低级操作，即目标机所支持的指令。不过，这并不意味着三地址代码就是机器指令系统的映射。设计者应该从便于后端处理的角度考虑，尽可能地发挥三地址代码作为中间语言的作用。

在 Pascal 语言中，集合变量的赋值或者传参实际上就是存储区的复制操作，这与指针、数组等是完全不同的。一个集合变量所需的存储空间是 32B（256 位），集合变量的赋值就是一次 32B 的存储区拷贝。对于大多数机器而言，通常都不提供类似的指令。那么，针对这种情况，应该如何设计三地址代码呢？可能有两种方案可选。第一，只生成一条三地址代码，设置一个独立的操作符（例如，MOV_SET），其含义就是将源地址起始的 32 个字节连续存储区中的数据复制到目标地址起始的存储区中。这种方案更多是从源语言的角度考虑，尽可能在 IR 中体现源语言的语义。至于如何由该三地址代码生成相应的目标代码是由目标代码生成器完成的，目标代码生成器可以使用一段目标语言的指令序列来替换这一条三地址代码，从而实现等价转换。第二，从目标语言的角度考虑，按目标机的字长生成若干条标准字长的赋值代码，例如，以 32 位目标机为例，就需要生成 8 条三地址代码。这种方案使得目标代码生成比较简单，编译器只需用一两条指令替换一条三地址代码即可。不过，就三地址代码本身而言，它更多地只是承载了目标机的特性，却很难明确地表达源语言的语义。对于单一 IR 的编译器而言，笔者并不提倡使用第二种方案。然而，对于存在多种 IR 的编译器而言，可根据不同的需要选用，两种方案是各具特色的。

读者可能会有疑问，第一种方案的优势又是什么呢？实际上，其优势就是便于后端的优化处理。这里，暂且不详细讨论优化，笔者只是通过一个简单的实例来阐明三地址代码与优化技术的联系。

例 5-2 三地址代码与优化技术。

三地址代码与优化技术如图 5-2 所示。

<pre>a:= [2]; b:= [3]; c:= a+b;</pre>	<pre>mov_set [2], ,a mov_set [3], ,b set_add a,b,c</pre>	<pre>mov_set [2], ,a mov_set [3], ,b mov_set [2,3], ,c</pre>	<pre>mov 4, ,a+0 mov 0, ,a+8 mov 0, ,a+28 mov 8, ,b+0 mov 0, ,b+8 mov 0, ,b+28 or a+0,b+0,c+0 or a+8,b+8,c+8 or a+28,b+28,c+28</pre>
a)	b)	c)	d)

图 5-2 三地址代码与优化技术示意

a) 源程序 b) 方案 1 的三地址代码 c) 优化后的三地址代码 d) 方案 2 的三地址代码

本例的三地址代码只是一种逻辑形式而已，并不是由 Neo Pascal 实际生成的。为了突出重点，笔者简化了一些细枝末节的元素。

本例是一个 Pascal 集合变量相关的源程序，其中 a、b、c 都是集合类型的变量。程序运行结果：a 的值为[2]，b 的值为[3]，c 的值为[2, 3]。注意，在 Pascal 语言中，当运算分量为集合类型变量时，加法运算的语义即为集合的并运算。先来比较一下(b)、(c)两段三地址代码的差异，不难发现，(b)代码段是完全按照 $c:=a+b$ 的语义生成的，而(c)代码段是直接 will 集合加计算的结果[2, 3]赋给 c 的。这里，暂且不论实际生成的目标代码形式如何，仅从这个简单的变换而言，相信不难得到如下结论：省去了集合加运算，程序的执行效率是有所提高的。如果考虑实际编译器生成(b)代码段时，还可能产生临时变量的情况，那么优化的效果就更为明显了。当然，不得不承认编译过程必须为优化付出一定的时间代价。不过，实践证明，大多数用户愿意用编译时间来换取运行效率的提高。显然，(b)代码段的语义非常明确，对于优化算法来说，它可以非常方便地获得这三条三地址代码所承载的语义，这对优化是非常有利的。

再来看看(d)代码段，它是根据方案 2 生成的三地址代码列表。注意，Pascal 语言规定类型的基类型元素的总和必须小于 256 个，这是由集合类型的物理结构决定的。一个集合变量占用 256 位 (32B) 存储空间，每一位表示一个元素的状态，置位表示元素存在于集合中，否则表示元素不存在于集合中。若基类型为 BYTE (允许最小元素为 0)，那么，集合[2, 3]可表示为(1100)b，即第 2、3 位置 1，其余位置 0。因此，在(d)代码段中，集合[2]的十进制表示就是 4，而集合[3]的十进制表示就是 8。从(d)代码段中，似乎已经很难看到源程序的实际语义了，更多的只是考虑目标机的物理结构及目标语言的语义。在这种情况下，优化算法要想实现(d)→(c)的转换也就变得非常困难。当然，这并不意味着这种转换是不可能实现的，只是代价较大而已。

通过例 5-2 的讲述，读者应该已经了解了三地址代码操作符抽象性的意义所在。当然，本例提到的优化只是冰山一角而已，不同的优化算法对于三地址代码的需求也是不尽相同的，这里就不再展开了。

下面，笔者再从操作分量的角度来讨论三地址代码。操作分量是三地址代码的另一个组成元素。讨论操作分量的目的就是明确一个操作分量需要承载哪些信息。

汇编语言指令的操作分量是非常简单的，在大多数情况下，汇编语言中只需描述操作分量的地址及操作分量的宽度（即占用空间大小）即可。例如：

```
mov eax, dword ptr [aa]
```

该指令的左操作分量是一个寄存器，寄存器的地址及宽度是机器的固有属性，所以不必显式说明。而指令的右操作分量是一个内存空间，地址即为标号 aa 所指出的地址，dword ptr 表示其宽度为 4B。在汇编语言中，操作分量的相关属性并不多，这可能是仅有的几个属性。通常，汇编器并不关心该存储空间中的数据到底是什么类型，也就是说，aa 所标识内存空间中数据的类型可能只有程序员本人可以解释。当然，笔者仅针对那些经典汇编语言及汇编器进行讨论，而 HLA 之类的高级汇编语言暂不做考虑。

同样，三地址代码作为一种中间语言，也有必要说明其操作分量的相关属性，这是决定三地址代码级别的关键因素。实际上，从操作分量上来说，不难发现，高级语言与汇编语言最主要的差异就在于类型。高级语言的操作分量通常都是有类型的，而汇编语言则不具备这一特点。然而，类型又是一个非常复杂的元素。在很多情况下，三地址代码的级别就是取决

于其是否支持以及如何支持类型。这个问题并不好回答，人们也很难评价这两种三地址代码孰优孰劣。不过，它却影响着后续工作的进行。在国内编译原理书籍中，很少提及关于三地址代码操作分量的类型描述等话题。读者可能会有一个错觉：三地址代码就是一种近似于汇编语言的 IR。这是一种比较常见的情况，但并不是绝对的。作为一种模型，完全可以避开类型描述之类的繁复细节，只需要关心操作分量的名字等主要属性即可。不过，就实际编译器设计而言，这恐怕是无法回避的。本书提及这一话题的目的就是希望读者对此给予足够关注。

最后，笔者简单谈谈三地址代码的物理存储结构。三地址代码是一种线性表示形式，所以有许多经典的线性表结构可以参考。关于线性表的结构，这里就不再深入讨论。不过，有个概念是需要说明的，根据一些经典编译器的设计经验，三地址代码的物理结构并非自始至终一成不变的。在多遍编译器中，使用不同物理存储结构来实现不同阶段的三地址代码是有意义的。例如，在前端，主要就是致力于生成三地址代码，顺序结构可能较优。而在后端，可能会涉及一些三地址代码的修改、调度，链式结构就优势明显了。当然，根据应用的需要，创新设计一些更适合本系统的物理结构是完全可以接受的。

至此，笔者已经详细讨论了三地址代码的相关理论话题，从实践设计角度，阐明了某些现代编译技术的观点。囿于篇幅，笔者省略了其中的某些常识性的概念，有需要的读者可以参见相关书籍。

5.2.2 Neo Pascal 三地址代码的实现

前面读者应该已经了解了三地址代码的基本概念。下面，就来看看 Neo Pascal 三地址代码的实现，本书将其简称为“NPIR”（Neo Pascal IR）。三地址代码设计主要包括两个方面：操作符与操作数。NPIR 的操作符并不是完全由笔者凭空想象出来的，其设计灵感来源于 lcc 的 IR。笔者觉得 lcc IR 的优势比较显著，尤其是其操作符的设计经验是值得与各位读者分享的。下面给出完整的 NPIR 的操作符列表，供读者参考，见表 5-2。

表 5-2 NPIR 的操作符

相等 (EQU)					
EQU_1	1 字节	EQU_S	字符串	EQU_A	数组
EQU_4	4 字节	EQU_8	8 字节	EQU_8F	8 字节浮点数
EQU_4F	4 字节浮点数	EQU_2	2 字节	EQU_32	32 字节
不等 (NEQU)					
NEQU_1	1 字节	NEQU_S	字符串	NEQU_4	4 字节
NEQU_8	8 字节	NEQU_8F	8 字节浮点数	NEQU_4F	4 字节浮点数
NEQU_2	2 字节	NEQU_32	32 字节	NEQU_A	数组
小于 (LS)					
LS_1	1 字节	LS_S	字符串	LS_4	4 字节
LS_8	8 字节	LS_8F	8 字节浮点数	LS_4F	4 字节浮点数
LS_2	2 字节	LS_1U	1 字节无符号	LS_2U	2 字节无符号
LS_4U	4 字节无符号				

(续)

大于(MR)					
MR_1	1 字节	MR_S	字符串	MR_4	4 字节
MR_8	8 字节	MR_8F	8 字节浮点数	MR_4F	4 字节浮点数
MR_2	2 字节	MR_1U	1 字节无符号	MR_2U	2 字节无符号
MR_4U	4 字节无符号				
小于或等于(LE)					
LE_1	1 字节	LE_S	字符串	LE_4	4 字节
LE_8	8 字节	LE_8F	8 字节浮点数	LE_4F	4 字节浮点数
LE_2	2 字节	LE_1U	1 字节无符号	LE_2U	2 字节无符号
LE_4U	4 字节无符号				
大于或等于(ME)					
ME_1	1 字节	ME_S	字符串	ME_4	4 字节
ME_8	8 字节	ME_8F	8 字节浮点数	ME_4F	4 字节浮点数
ME_2	2 字节	ME_1U	1 字节无符号	ME_2U	2 字节无符号
ME_4U	4 字节无符号				
集合包含(IN)					
IN_C	CHAR 值	IN_I	INTEGER 值	IN_B	BOOLEAN 值
IN_SH	SHORTINT 值	IN_SM	SMALLINT 值	IN_W	WORD 值
IN_LW	LONGWORD 值	IN_E	ENUM 值		
减法(SUB)					
SUB_4	4 字节	SUB_8F	8 字节浮点	SUB_8	8 字节
SUB_SET	集合差	SUB_2	2 字节		
加法(ADD)					
ADD_S	字符串连接	ADD_4	4 字节	ADD_8F	8 字节浮点数
ADD_SET	集合并	ADD_2	2 字节		
或(OR)					
OR_0	位或	OR_4	4 字节	OR_1	1 字节
OR_2	2 字节				
异或(XOR)					
XOR_0	位异或	XOR_4	4 字节	XOR_1	1 字节
XOR_2	2 字节				
乘法(MUL)					
MUL_4	4 字节	MUL_8F	8 字节浮点数	MUL_SET	集合交
除法(DIV)					
DIV_8F	8 字节浮点数	DIV_4	4 字节		
取模(MOD)					
MOD_4	4 字节				
右移(SHR)					
SHR_4	4 字节	SHR_1U	1 字节无符号	SHR_1	1 字节
SHR_2	2 字节	SHR_2U	2 字节无符号	SHR_4U	4 字节无符号



(续)

左移(SHL)					
SHL_4	4 字节	SHL_1U	1 字节无符号	SHL_1	1 字节
SHL_2	2 字节	SHL_2U	2 字节无符号	SHL_4U	4 字节无符号
与(AND)					
AND_0	位与	AND_4	4 字节	AND_1	1 字节
AND_2	2 字节				
取负(NEG)					
NEG_4	4 字节	NEG_1	1 字节	NEG_2	2 字节
NEG_8F	8 字节浮点	NEG_4F	4 字节浮点数		
取非(NOT)					
NOT_0	位非	NOT_4	4 字节	NOT_1	1 字节
NOT_2	2 字节				
赋值(ASSIGN)					
ASSIGN_1	1 字节	ASSIGN_S	字符串	ASSIGN_4	4 字节
ASSIGN_2	2 字节	ASSIGN_8	8 字节	ASSIGN_N	多字节
ASSIGN_32	32 字节				
其他					
JMP	无条件跳转	LABEL	标号定义	PARA	传递参数
RETV	传递返回值	CALL	函数调用	JT	条件真跳转
JNT	条件假跳转	JE_1	1 字节相等跳转	JE_2	2 字节相等跳转
JE_4	4 字节相等跳转	CALLPTR	调用函数指针	ASM	汇编
GETADDR	获得变量地址	SET_ADDITEM	元素加入集合	GETPROCADDR	获得函数地址

仅就表 5-2 的规模而言, 可能读者会觉得 NPIR 是一个非常庞大的体系。从 IR 设计的角度而言, 这种设计似乎并不是非常合理。不过, 有理由相信这种来自 lcc 的设计思想是有优势的。那么, 读者不妨思考一下其优势何在? 这种方案的目的在于得到的三地址代码可以同时作为 MIR、LIR 使用, 并且不必考虑 LIR 到 MIR 的转换代价。有些读者可能会对笔者的观点有所怀疑。下面解释一下其中的原因。

首先, 按功能将这 130 余个操作符分为 21 大类, 即“相等”、“不等”、“小于”等表头。如果将前 20 个类别(即除“其他”类之外)分别用 1 个操作符(即表头括号中字符)表示, 那么, 这 20 个操作符和最后一类的 15 个操作符就形成了一套 MIR。这个操作符集合与目标机几乎是无关的, 因为它既没有涉及具体目标机的特性, 也没有过多考虑物理存储结构。

然而, 仅考虑大类下属的具体操作符, 那么, 它就是一套 LIR。其中涉及一些目标机特性, 例如, 处理机字长、物理存储结构等。

那么, NPIR 又是如何实现 LIR 与 MIR 之间无代价转换的呢? 实际上, 这是非常简单的。NPIR 操作符集合是一组完整的枚举值。笔者通过一种编码规则对各枚举常量进行编号, 根据枚举常量的编码就可以对类别及其所属操作符加以区分。例如, EQU 的编码为 $1 \ll 4$, 而 EQU_1、EQU_S 的枚举值分别为 $(1 \ll 4) + 1$ 和 $(1 \ll 4) + 2$, 依此类推, 将 EQU 类别的所有所属操作符加以编号。也就是说, 操作符编号二进制值的高 5 位(由于一共有 21

类，需要 5 位二进制值表示）表示其类别，低 4 位（由于同一类别中最多不超过 16 种操作符）表示操作符在该类别中的顺序编号。通过这样的编码方式，就可以方便地将 LIR 转换为 MIR。注意，在 Neo Pascal 语义分析过程中，提交的三地址代码是 MIR 形式，而经过类型系统处理后的最终形式是 LIR。这里，讨论 LIR 转换到 MIR 的目的就是考虑到很多中间代码优化算法是基于 MIR 实现的。通过简单地屏蔽操作符编码的低 4 位就可以获得 MIR 的结果是设计者乐意见到的。

讨论完 NPIR 的设计目的之后，笔者针对 NPIR 操作符的特点作两点说明：

(1) 在带字节指示的操作符中，除了浮点数及明确说明无符号数的操作符之外，都默认为是有符号数的操作符，即三地址代码的操作数都是有符号类型的值或变量。

(2) NPIR 中设置了 32B 的操作符的目的是处理集合类型变量及数据的，因为 Pascal 集合类型数据所占的存储空间就是 32B（256 位）。

实际上，详细分析 NPIR 操作符的含义是没什么意义的，对汇编语言稍有了解的读者可以非常轻松地理解。

至此，笔者已经详细地讲解了 NPIR 的操作符的相关内容。下面，再来简单看看操作数及其存储结构。NPIR 的操作数是非常简单的，主要由以下三个字段描述：`m_iType`、`m_iLink`、`m_bRef`。结构声明如下所示：

【声明 5-1】

```
struct OpInfo
{
    enum {CONST, VAR, PTR, LABEL, NONE, PROC} m_iType;
    int m_iLink;
    .....
    bool m_bRef;
};
```

其中，`m_iType` 描述该操作数的类型，即常量、变量、指针、标号、过程等。`m_iLink` 是指向实际操作数的指针，例如，操作数类型为变量，则 `m_iLink` 的值指向该变量在变量信息表中的位序。`m_bRef` 表示该操作数是否需要引用访问（或者称为间接访问）。对于 NPIR 而言，操作数的结构仅此而已。

当然，与声明 5-1 相比，Neo Pascal 源代码中的操作数声明较复杂。这里，读者可以暂且略过。下面，再来看看 NPIR 的声明形式：

【声明 5-2】

```
struct IRCode
{
    OpType m_eOpType;           //操作符，即称为操作类型
    OpInfo m_Op1,m_Op2,m_Rslt;  //操作数 1，操作数 2，结果操作数
    .....
};
vector<IRCode> m_Codes;
```

实际上，`m_Codes` 并不是一个全局变量，而是过程信息表项的一个属性，也就是说



NPIR 是以函数或过程为单位组织的。下面，笔者给出一个完整的实例分析，以便读者对 NPIR 有感性的认识。

例 5-3 NPIR 实例分析，见表 5-3。

表 5-3 NPIR 实例分析

Neo Pascal 源程序	NPIR
program main(input,output);	(ByteToInt , 1 ,, _T3)
var i,j:integer;	(ASSIGN_4 , _T3 ,, 1)
begin	(LABEL , _L1 ,,)
for i:=1 to 10 do	(ByteToInt , 10 ,, _T5)
j:=j+i;	(LE_4 , 1, _T5 ,, _T4)
begin	(JNT , _T4 ,, _L0)
end;	(ADD_4 , J, 1, _T6)
end.	(ASSIGN_4 , _T6 ,, J)
	(LABEL , _L2 ,,)
	(ByteToInt , 1 ,, _T7)
	(ADD_4 , 1, _T7 ,, 1)
	(JMP , _L1 ,,)
	(LABEL , _L0 ,,)

这个实例比较简单，对于熟悉汇编语言的读者，相信并不需要太多的注释。不过，读者可能对其中的 ByteToInt 操作符有所疑惑。确实，在先前操作符列表中，笔者并没有提到 ByteToInt。实际上，从字面上来看，也不难猜出其类型转换的功能。读者的关键问题可能是想知道它到底是不是操作符？答案是肯定的。实际上，表 5-2 所列出的操作符尚不完整，还有一部分类型转换的操作符并不在其中。关于这类操作符，将在第 6 章中详细阐述。

关于 NPIR 的优点，先前已经讲了不少了，选用这种形式还是具有一定积极意义的。下面，笔者想客观地谈谈 NPIR 设计的不足之处。由于 NPIR 主要定位是 MIR、LIR，所以源程序中的某些非常明显的语言结构可能会被隐藏，例如，数组访问、记录访问等在 NPIR 中都是以指针引用形式出现。当然，从语义的角度而言，这是完全等价的，但是这种形式却并不利于某些别名分析算法的实现。实际上，这就是前面所说的级别问题，相对而言，抽象语法树等 HIR 在这方面的表现就比较有优势了。

由于 NPIR 是 Neo Pascal 的一个重要组成部分，因此，深入理解 NPIR 对于后续学习是有重要意义的。囿于篇幅，就不再举例分析了。对于某些尚不熟悉 NPIR 的读者，可以自己调试 Neo Pascal 源代码，将 NPIR 输出与源程序对照学习。

5.2.3 翻译机制概述

本小节将讨论编译器的翻译机制。实际上，先前讨论的词法、语法分析及生成符号表等工作无非都是为程序翻译提供决策支持的，翻译工作才是编译器的核心。读者应该对自然语言的翻译并不陌生，甚至不敢想象复杂的翻译工作可以由机器自动完成。自然语言翻译确实是一项复杂的工作，迄今为止，人们还无法实现一个精准的自然语言翻译系统。然而，程序设计语言与自然语言不同，机器翻译程序设计语言是完全可能的，并且已经有非常成功的实例。不过，笔者必须指出，编译器自动生成的目标代码的质量是无法达到或超越经过精雕细琢的手工代码的，即使是具有强大优化机制的 GCC、Intel C++等也无法保证目标代码是最优的。当然，与一些计算机科学的经典问题类似，编译技术中的“代码最优化”也是难以衡量与评价的。

经过前面的讲解，编译器的翻译机制已经并不神秘了。编译器设计者就是通过文法中插入相应的语义子程序，使之完成语法制导的翻译。在分析符号表处理时，笔者详细讲解了语法制导的理论与技术。不过，仅仅了解语法制导的翻译过程是远远不够的。这里，将从翻译方案入手，讨论翻译机制的实现。

翻译方案就是讨论源语言结构与 IR 之间的联系。例如，讨论 case 结构对应的 IR 序列、函数的 IR 等。翻译方案对编译行为、后端优化器设计及效能都有着很大的影响。目前，读者所熟悉的编译器都不是真正智能的（包括动态编译器）。编译过程中的每一步翻译都是有严格的算法定义的，编译器本身并不进行学习。当然，就编译器设计初衷而言，设计者也并不期望编译器独立思考与创新。在生成三地址代码的阶段，翻译方案就是设计算法的基础。下面看一个翻译方案的实例，见表 5-4。

表 5-4 for 语句翻译方案实例

for 语句结构	翻译方案 A	翻译方案 B
<pre>for 变量:=初值 to 终值 do begin end;</pre>	<pre>变量 ← 初值 Lx1: if !(变量>终值) goto Lx0 变量 ← 变量 + 1 goto Lx1 Lx0:</pre>	<pre>变量 ← 初值 Lx1: if (变量<=终值) goto Lx2 goto Lx0 Lx2: 变量 ← 变量 + 1 goto Lx1 Lx0:</pre>

以上的实例是 for 语句的翻译方案，虽然两个方案结果都是正确的，但是无论是从代码的篇幅还是代码的效率而言，方案 A 明显优于方案 B。翻译方案设计的主要工作就是得到相对较优的 IR 结构。这项工程对于熟悉三地址代码或者机器汇编的设计者来说，可能并不复杂。即便如此，翻译方案的设计同样是具有一定挑战性的。

以 case 结构为例，众所周知，case 语句（即 C 语言中的 switch 语句）是一种多分支结构。然而，由于大多数目

标机指令系统都不支持多分支结构，因此，通常是将其翻译成多级嵌套的 if-then-else 结构。不过，针对图 5-2 所示的例程，不难发现，这个例程的特点是分支非常紧密。在这种情况下，继续使用 if-then-else 结构进行翻译就可能会显得比较臃肿。实际上，这种情况更适用于查表方案。这是一种经典的处理方式，有兴趣的读者可以参考汇编语言的教程。图 5-2 的例程是查表方案最简单的处理，实际上，查表方案也可以应用于一些分支非常多的情况。当然，这种情况下，编译器也需要为此付出一定的空间代价。这里，笔者的目的不在于讨论查表技术，而是想说明翻译方案选择的问题。

针对不同的实际情况，存在不同的翻译方案，当然，有时它们是相对较优的。编译器如何选择翻译方案呢？例如，认定例程是分支紧密型的标准是什么？多少分支才认为是多分支情况？等等。通常，这些问题都需要编译器设计者经过一定的理论证明及实验总结后才能获得答案。因此，翻译方案的设计并不是一项容易的事，为了达到相对最优的翻译方案，需要付出的努力还是比较大的。

```
case I of
1..5: Caption := 'Low';
6..9: Caption := 'High';
0, 10..99: Caption := 'Out of range';
end;
```

图 5-2 case 语句实例



5.3 语句翻译概述

5.3.1 语句翻译基础

前面，笔者已经讲解了一些与生成 IR 有关的理论知识，这些理论知识是分析 IR 生成模块的基础。在本小节中，笔者将着眼于分析编译器的翻译机制，看看编译器是如何将输入源程序翻译成 IR 的。对大多数程序设计语言而言，编译器的翻译主要涉及两个领域：语句翻译、表达式翻译。结构化语句与表达式是高级语言最重要的两个组成元素，同时也是区别于低级语言（汇编语言）的主要标志。

程序设计语言的语句是一个非常精致的体系结构，它们的组合可以诠释任何复杂的逻辑。一门程序设计语言的语句少则数种，多则十几种，就是这样一个体系规模却创造出了一个千变万化的奇妙世界。在笔者看来，“精致”一词仍不足以描绘其本质。读者熟悉的这个语句结构模型非常经典，无论是结构化语言还是面向对象语言都没有颠覆其主导地位。

习惯上，将语句结构分为三类：顺序结构、选择结构、循环结构，这恐怕是每位程序员早已烂熟于心的。本章将从编译器设计的角度，让读者理解各种语句结构的内核实现。Neo Pascal 一共提供了 12 种语句结构，分别是赋值语句、过程函数调用语句、goto、continue、复合语句、while、repeat、if、for、case、asm、with。Neo Pascal 的语句结构是完全参考标准 Pascal 实现的，并在标准 Pascal 的基础上加入了 asm 语句。

笔者先给出 Neo Pascal 语句相关的文法，以便后续章节中针对各类语句结构进行详细的分析。

【文法 5-1】

语句序列	→ 语句行 语句序列 ϵ
语句行	→ 语句 ;
语句	→ 标识符 061 标识符起始语句 其他语句 ϵ
标识符起始语句	→ 变量 1 053 := 表达式 064 : 062 语句 过程调用语句
过程调用语句	→ (101 实参列表) 066 ϵ 005
其他语句	→ goto 标识符 063 → break 081 → continue 082 → begin 语句序列 end → while 071 表达式 073 do 语句 072 → repeat 074 语句序列 until 表达式 075 → for 标识符 076 := 表达式 077 for 语句后部 → if 表达式 068 then 语句 if 语句后部 070 → case 表达式 084 of case 分支 end 089 → asm 字符串常量, 字符串常量 083 end → with 090 变量 053 091 do 语句 092
for 语句后部	→ to 表达式 078 do 语句 079 downto 表达式 080 do 语句 079
if 语句后部	→ else 069 语句 ϵ
case 分支	→ 常量列表 087 : 语句 088 ; case 分支 ϵ

从 Neo Pascal 的文法来看, 笔者将 Neo Pascal 的语句结构分为两类: 标识符起始语句、其他语句 (也就是关键字起始语句)。其中, 标识符起始语句只有三种, 即赋值语句、过程调用语句以及标号语句, 而其余的语句都属于后者。这里, 读者应该注意一个细节, 在 Pascal 语言中, 复合语句 (BEGIN...END 结构) 处理与其他普通语句是一样的, 所以复合语句结束处仍需要标注分号。

下面, 先来谈谈 Neo Pascal 语句翻译的总体实现。根据 IR 的设计目标, 高级语言的任何语句结构都将转换成等价的 IR 形式, 而这种转换工作完全是由若干个语义子程序协调完成的。然而, NPIR 是一种 MIR, 仅提供了一些跳转操作符, 所以编译器不得不将 Neo Pascal 的各种语句结构转换成由跳转、标号等组成的 IR 序列。虽然各种编译器设计 IR 形式不尽相同, 但是, 使用跳转、标号组合实现源语言的语句结构却是得到共识的。

语句的各种嵌套组合是不可预知的, 设计者通常是无法穷尽各种存在可能性的, 转换工作的复杂性可能是难以想象的。那么, 面对这个复杂的问题, 通常的解决方案就是分而治之。初期, 并不需要过多考虑嵌套组合等复杂情况, 只需实现基于各种基本语句的等价翻译即可。待解决了最基本语句的翻译后, 再逐步考虑各种复杂情况的存在。实际上, 基本语句的翻译是非常简单的, 这里。笔者通过一个 if 语句的实例来看看编译器是如何翻译基本语句的。

例 5-4 if 语句的 IR 实例, 见表 5-5。

表 5-5 if 语句的 IR 实例

Pascal 源程序	IR 序列
if i>j then	(MR_4 , I, J, _T0)
i:=1	(JNT , _T0, null, _L1)
else	(ByteToInt , I, null, _T3)
j:=1;	(ASSIGN_4 , _T3, null, I)
	(JMP , _L2, null, null)
	(LABEL , _L1, null, null)
	(ByteToInt , I, null, _T4)
	(ASSIGN_4 , _T4, null, J)
	(LABEL , _L2, null, null)

这是一个 if 语句的 IR 实例, 读者可以先忽略 IR 序列中的非粗体代码。除了临时变量名、标号名可能存在一定的变化之外, 粗体所示的 IR 结构正是 if 语句的标准翻译方案。如果省略了 else 子句, 则只需忽略翻译方案 (黑体所示的 IR 序列) 中的 2、4 两行 IR 即可。就本例而言, 读者可能觉得从源程序到 IR 的转换过程似乎并没有想象中的复杂。实际上, 确实如此, 当问题规模很小的时候, 一些原本表面上比较复杂的问题就会变得简单, 将语句翻译的问题细化到各个语句结构时, 就可以指导编译器很机械地完成等价转换的工作。从上例不难发现, if 语句翻译方案是非常简单的, 只要稍有汇编语言基础的读者都可以轻松理解。类似地, 设计各种基本语句结构的翻译方案并不是非常困难的, 详细的翻译方案将在后续章节中逐一描述。

解决了基本语句的翻译之后, 就得考虑各种语句之间的嵌套情况。根据程序设计语言的语法、语义规定, 语句的嵌套并不是任意为之的, 它必须满足一定的层次条件。抽象语法树的观点认为任何复杂的语句嵌套情况都可以借助于树的形式加以描述。确实, 不得不承认



用抽象语法树可以使语句翻译变得相对容易，它很好地描述了语句、表达式之间的联系。不过，由于 Neo Pascal 并不会显式构造抽象语法树，所以不得不借助于其他数据结构实现。根据先前的经验，栈结构就是不二之选。那么，就来详细看看 Neo Pascal 又是如何实现这个栈结构的呢？在 Neo Pascal 中，称其为“当前语句栈”，声明如下：

【声明 5-3】

```
stack<Statement> CurrentStatement;
```

CurrentStatement: 栈顶元素描述的就是当前（控制）语句的相关信息。与符号表不同的是 **CurrentStatement** 只是一个暂存结构，而符号表却是一套贯穿于整个编译过程的数据结构。由于各种语言结构的不同，编译器需要得到的信息也不尽相同。同样，也不可能要求 **Statement** 结构应对各种应用需要，但是仅针对 Pascal 语言来说，**Statement** 结构所描述的信息已经足以完成编译工作了。读者必须注意，**Statement** 结构的功用并不是承载语句语义，而是存储一些与编译相关的信息或者属性。

【声明 5-4】

```
struct Statement
{
    enum {IF,WHILE,FOR,REPEAT,CASE} m_eType;
    struct LabelIdx
    {
        enum LabelType{TrueLabel,FalseLabel,ExitLabel,EntryLabel,CaseLabel} m_LabelType;
        OpInfo m_Label;
        int m_iIdx;
        int m_iConst;
    };
    vector<LabelIdx> m_Labels;
    int m_iLoopVar;
    bool m_bIsDownto;
    bool m_bIsElse;
    OpInfo m_CaseExp;
    .....
};
```

m_eType: 用于描述该语句结构的类型。注意，这里并不需要考虑那些不允许嵌套的语句，如 **break**、**goto** 等。虽然它们也可能产生跳转，但是却不可能出现因嵌套导致的复杂标号联系。根据 Neo Pascal 语言的特点，这里只需设置五个枚举值即可。

m_Labels: 这是一个标号列表，用于存储与当前语句相关的所有标号信息。关于 **LabelIdx** 结构的解释，笔者将稍后给出。

m_iLoopVar: 如果当前语句是 **for** 语句时，则 **m_iLoopVar** 就是指向循环变量的指针。在 Pascal 语言中，循环变量对于 **for** 语句是尤为重要的，循环变量的增、减或者比较运算并没有显式的表达式，而是隐含于语句结构的语义中，这就需要编译器加以识别处理。然而，在 C 语言中，循环变量的作用就完全被 **for** 语句的三个表达式替代了，而这三个表达式是由用户设计编写的，因此，只需将表达式翻译成目标代码即可。当然，**m_iLoopVar** 属性在

case 语句中还有一个特殊的作用，待后续章节中详述。

m_bIsDownto: 这是一个标志属性。前面，读者应该已经了解了 Pascal 的 for 语句与 C 语言 for 语句的区别，标准 Pascal 的 for 语句规定每次循环完毕循环变量自增或自减 1。而编译器则是通过 for 语句中的 to 或 downto 关键字加以区分。注意，在标准 Pascal 中，并没有循环步长的概念，唯一的步长就是 1。而 m_bIsDownto 属性就是用于标识当前 for 语句是否为自减模式。

m_bIsElse: 用于标识当前 if 语句是否存在 else 子句。

m_CaseExp: 用于存储一个实际的操作数（case 表达式）。由于 case 表达式对于翻译 case 语句是非常重要的，所以有必要加以记录。

下面解释一下 LabelIdx 结构及其各属性的作用。

LabelIdx: 这是一个内嵌结构类型声明。根据前面的讲述，标号信息是语句翻译过程中极为重要的元素，例如，语句结构的入口标号、出口标号等。实际上，控制语句翻译的关键就是正确生成与管理各种标号。下面，笔者详细分析其中各属性的含义。

m_labelType: 这个属性的作用是标识该标号的类型。根据常见的高级语言控制结构的特点，一般可以将标号分为如下五种类型：TrueLabel（真分支标号）、FalseLabel（假分支标号）、ExitLabel（出口标号）、EntryLabel（入口标号）、CaseLabel（Case 标号）。其中，真、假分支标号主要是指标识 if 语句真、假分支的标号。而出、入口标号主要是指标识循环结构出、入口的标号，当然出口标号也常见于其他控制结构中。严格区别五种标号类型的主要目的是为了提高程序的可读性。

m_Label: 这个属性用于存储一个实际标号操作数。

m_iIdx: 指向该标号的指针。

m_iConst: 这个属性仅当所属 Statement 实例的 m_eType 的值是 CASE（即表示当前语句是 case 语句）时才有效。在 Pascal 语言中规定一个 case 分支只能关联一个常量值，而不是一个常量列表。通常，在翻译 case 语句时，会为每一个 case 分支分配一个标号，而 m_iConst 就是用于标识当前标号所关联的常量值在常量信息表中的位序号。

总体来说，Pascal 的语句结构是比较精简的，但也是比较经典的结构化语言模型，因此，Pascal 语句翻译机制的设计对于其他语言编译器设计是具有一定指导意义的。

5.3.2 翻译辅助函数及其实现

前面介绍了 Neo Pascal 语句翻译过程中两个非常重要的数据结构，即 Statement 与 IRCode。其中 Statement 是翻译过程中的一个辅助结构，其目的是临时存储当前语句的翻译信息。而 IRCode 也就是 IR 代码的基类型，在介绍三地址代码时，已经作了相关讨论。本小节将讲述一些翻译辅助函数的功能及其实现。从表面上看，这些函数的功能可能并不复杂，例如，IR 代码管理、语句标号生成等。不过，在整个系统设计中，它们的地位却是极其重要的。实际上，Neo Pascal 的翻译辅助函数并不少，这里只能选择部分比较重要的函数予以分析。

1. IR 生成函数——EmitIR

IR 生成的工作将贯穿于整个语义处理阶段，然而由各个语义子程序直接访问 m_Codes 列表可能会大大增加 m_Codes 列表的管理成本，因此，设置 IR 生成函数的目的就是为了让



于接口统一。当 `m_Codes` 类型或结构变化时，不至于大费周章地检查与修改每一个 `m_Codes` 的引用点。下面，笔者就介绍一组通用函数 `EmitIR`，这是一组参数重载函数，其功能就是将传入的参数组合生成一个 `IRCode` 对象返回。

程序 5-1 CommonLib.cpp

```
1  IRCode EmitIR(OpType eOpType,OpInfo Op1,OpInfo Op2,OpInfo Rslt)
2  {
3      IRCode Tmp;
4      Tmp.m_eOpType=eOpType;
5      Tmp.m_Op1=Op1;
6      Tmp.m_Op2=Op2;
7      Tmp.m_Rslt=Rslt;
8      Tmp.m_bEnabled=true;
9      return Tmp;
10 }
11 IRCode EmitIR(OpType eOpType,OpInfo Op1,OpInfo Rslt)
12 {
13     IRCode Tmp;
14     Tmp.m_eOpType=eOpType;
15     Tmp.m_Op1=Op1;
16     Tmp.m_Rslt=Rslt;
17     Tmp.m_bEnabled=true;
18     return Tmp;
19 }
20 IRCode EmitIR(OpType eOpType,OpInfo Op1)
21 {
22     IRCode Tmp;
23     Tmp.m_eOpType=eOpType;
24     Tmp.m_Op1=Op1;
25     Tmp.m_bEnabled=true;
26     return Tmp;
27 }
```

函数的功能是非常简单的，笔者觉得关于源码的任何解释可能都是多余的。至于三种参数重载形式的目的也是显而易见的，实际上，三地址代码并不一定需要三个完整的操作数，有些操作符只需要 1~2 个操作数即可。然而，在 `Neo Pascal` 中，并不存在零操作数的操作符，所以并不需要相应的参数重载形式。

2. 临时对象生成函数

临时对象指的就是由编译器在编译过程中自动生成的标号、变量、常量等。注意，这些对象并不一定是透明的，它们在目标代码中是可见的。至于为什么需要生成这些临时对象呢？答案非常简单，它们是目标代码中不可或缺的元素。例如，在翻译 `if` 语句时，编译器不得不为此自动生成一些标号以实现语义的等价变换。当然，临时对象还不仅限于临时标号，可能还包括一些临时的变量、常量等。

在编译过程中，生成临时对象是非常频繁的操作。输入源程序规模相对较大时，其编译

过程中产生的临时对象将是一个非常庞大的集合。由此所付出的资源代价是不能忽视的。

生成临时对象可能并不像读者所想象的那样简单。在生成临时对象时，任何编译器都必须注意以下两个问题：

(1) 必须保证临时对象的命名空间与用户命名空间之间不存在任何交集。这个问题对于像 Pascal、C 语言等而言，并不复杂。实际上，只需突破具体语言的标识符定义规则即可。例如，C 语言规定标识符必须以下划线或字母开头，那么，临时对象的名字就可以以“#”或“*”开头，这样就可以有效避免与用户标识符的冲突。由于临时对象的名字并不需要通过词法分析识别，所以它可以完全颠覆词法分析器对于标识符的限制，即使将临时对象命名为以数字开头字符串也是完全可行的。然而，这种命名规则却是词法分析器无法接受的。

(2) 必须保证目标代码中的临时对象名字不存在任何冲突。由于各个语义子程序都是离散的，所以统一的命名管理是必不可少的。通常，顺序种子、长随机编码等都是比较常见的解决方案。

除了上面谈到的两个问题之外，临时对象的声明过程中还有一项重要的工作，即登记符号表。对于编译器而言，编译过程中产生的临时对象同样需要准确登记符号信息。实际上，分析临时对象的符号信息并不简单，编译器不可能像分析用户对象一样从显式声明中获取临时对象的相关信息，而只能从输入源程序的上下文语义中分析获得。临时标号、临时常量的信息比较容易获取。然而，临时变量则相对复杂，其类型信息只能通过表达式运算符的语义加以推断。

本小节将介绍两类生成函数，分别是临时标号生成函数、临时变量生成函数。下面，就来看看这两类函数的实现。

程序 5-2 SymbolTbl.cpp

```
1  int CSymbolTbl::GetTmpLabel(int iProcIndex)
2  {
3      LabelInfo Tmp;
4      Tmp.m_iProcIndex=iProcIndex;
5      Tmp.m_szName="_L";
6      Tmp.m_szName.append(GetVarId());
7      Tmp.m_bDef=true;
8      Tmp.m_bUse=true;
9      LabelInfoTbl.push_back(Tmp);
10     return LabelInfoTbl.size()-1;
11 }
```

GetTmpLabel 的主要功能就是生成一个临时标号信息，并将其加入标号信息表。根据 Neo Pascal 的设计，临时标号的命名规则为“_L×××”，其中“×××”表示一个由编译器顺序分配的 id 号，以保证系统生成的标号名字是唯一的。而临时标号以“_L”开头的目的就是为了避免与用户标号出现重名现象。当然，针对 C 语言或者其他语言，这种处理就不可行了。

与临时标号生成函数相比，临时变量生成函数稍复杂，除了变量名字等常规信息之外，还需要考虑类型等相关信息的生成。



程序 5-3 SymbolTbl.cpp

```
1  int CSymbolTbl::GetTmpVar(int iProcIndex,StoreType eStoreType)
2  {
3      VarInfo Tmp;
4      Tmp.m_iProcIndex=iProcIndex;
5      Tmp.m_iTypeLink=TypeInfoTbl.size();
6      Tmp.m_szName="_T";
7      Tmp.m_szName.append(GetVarId());
8      Tmp.m_eRank=VarInfo::VAR;
9      VarInfoTbl.push_back(Tmp);
10     TypeInfo TmpType;
11     TmpType.m_szName="_noname";
12     TmpType.m_szName.append(GetSerialId());
13     TmpType.m_eDataType=eStoreType;
14     TmpType.m_eBaseType=eStoreType;
15     TypeInfoTbl.push_back(TmpType);
16     return VarInfoTbl.size()-1;
17 }
```

GetTmpVar 的主要功能就是生成一个临时变量信息，并将其加入变量信息表。根据 Neo Pascal 的设计，临时变量的命名规则为“_T×××”，其中“×××”表示一个由编译器顺序分配的 id 号，以保证系统生成的变量名字是唯一的。

这里，笔者给出了 GetTmpVar 函数的最基本实现，根据 eStoreType 参数，生成一个相应类型的临时变量。在实际应用中，编译器可能无法准确提供临时变量的类型信息，因此就需要一些其他的重载形式。囿于篇幅，不再详细分析重载形式的源代码实现，请读者自行参考 Neo Pascal 源代码。当然，如何确定临时变量的类型是一个复杂的话题，通常，这是由类型系统决策的。关于类型系统的相关话题，将在介绍表达式翻译时详细阐述，此处并不需过于关注临时变量的实际类型。

5.4 if 语句

5.4.1 if 语句的翻译

if 语句是现代程序设计语言最基本的语句结构之一。从结构化语言到面向对象语言，从脚本语言到非过程式语言，从命令式语言到函数式语言，通常都能见到 if 语句的身影。至于 if 语句的基本结构与功能，笔者觉得没有任何必要在此重复了。

if 语句的标准文法如下：

【文法 5-2】

其他语句 → if 表达式 068 then 语句 if 语句后部 070
if 语句后部 → else 069 语句 | ε

从文法上来看，if 语句的文法并不复杂，结构相对也比较清晰。这里，一个话题是值得

思考的，那就是如何处理非终结符“表达式”。

不难发现，文法中存在非终结符“表达式”。其目的是显而易见的，希望通过这个非终结符推导各种复杂的表达式。当然，它也确实达到了这一目标，而具体的实现细节正是下一章的核心论点。不过，由于 if 语句的翻译将涉及其他内容，这里不得不作简单说明。

实际上，表达式翻译的实现是各不相同的，有些表达式翻译借助于栈实现，而有些表达式翻译可能借助于抽象语法树实现。前者是比较常见的实现方案，也是一种比较经典的方法，其核心思想在一些数据结构教材中亦有描述。虽然这种实现方案简单灵活，但并不是万能的。在一些特殊的应用场合，前者的实现可能并不简单，甚至会大大增加实现的复杂度。关于利用栈结构实现表达式处理的算法，读者应该并不陌生。编译器应用栈结构翻译表达式的过程与数据结构的相关算法是比较类似的，只是编译器并不一定直接将计算结果常量压栈，而是可能将产生的临时变量压栈。这样，可以得到一个结论：编译器处理完一个表达式后，栈顶元素可能是一个常量或者变量。如果栈顶元素是常量，则表示该表达式的运算结果。如果栈顶元素是变量，则意味着该变量所存储的值就是表达式的运算结果。理解了这个结论以后，非终结符“表达式”将不再成为语句翻译的障碍。

表达式的翻译通常是独立于语句翻译而存在的，在语句翻译过程中，更多时候只是对表达式的运算结果作一定的类型检查。当然，这并不是绝对的，正如前面所说的，某些语言特性可能要求编译器将表达式与语句翻译统一规划。

根据 if 语句的语义，可以得到如表 5-6 所示的翻译方案。

表 5-6 if 语句的翻译方案

if 语句	翻译方案
if <表达式> then	<表达式翻译>
<语句 1>	(JNT , <表达式结果>, null, __L1)
else	
<语句 2>	(JMP , __L2 , null, null)
	(LABEL , __L1 , null, null)
	<语句 2>
	(LABEL , __L2 , null, null)

如果省略了 else 部分，那么只需将翻译方案中第 4~6 行语句省略，并将第 7 行的“__L2”替换为“__L1”即可。semantic068、semantic069、semantic070 主要的功能就是根据翻译方案翻译输入的 if 语句。也就是说，试图依靠这三个语义子程序，完成翻译方案中黑体语句的生成。在上述翻译方案中，可以暂且将“__L1”称为“假分支标号”，而将“__L2”称为“出口标号”。另外，需注意一点，当输入语句是 if-then 结构时，第 7 行语句的标号不应该取出口标号，而应该取假分支标号，因为此时并不存在真正意义的假分支，因此，可以将假分支标号当作出口标号使用。

5.4.2 源代码实现

在分析源代码之前，笔者先简单介绍一下各语义子程序的功能，以便读者理解。

semantic068: 1、判断表达式结果是否为布尔类型。

2、生成 JNT 语句，目标标号就是假分支标号。



semantic069: 1、生成 JMP 语句，目标标号就是出口标号。

2、生成假分支的入口 LABEL 语句，即翻译方案中的第一个 LABEL 语句。

semantic070: 1、生成 if 语句的出口 LABEL 语句，即翻译方案中的第二个 LABEL 语句。

下面，就来看看 if 语句相关的语义子程序。

程序 5-4 semantic.cpp

相关文法:

其他语句 → if 表达式 068 then 语句 if 语句后部 070

```
1 bool semantic068()
2 {
3     if (OpData.empty())
4     {
5         EmitError("语义错误",TokenList.at(iListPos-1));
6         return false;
7     }
8     OpInfo Tmp=OpData.top();
9     OpData.pop();
10    if (CType::GetOpType(Tmp)!=StoreType::T_BOOLEAN)
11    {
12        EmitError("IF 语句判定表达式必须为 BOOLEAN 类型",TokenList.at(iListPos-1));
13        return false;
14    }
15    else
16    {
17        Statement TmpS;
18        Statement::LabelIdx TmpLblIdx;
19        OpInfo TmpLbl;
20        TmpS.m_eType=Statement::IF;
21        TmpS.m_bIsElse=false;
22        TmpLbl.m_iType=OpInfo::LABEL;
23        TmpLbl.m_iLink=SymbolTbl.GetTmpLabel(SymbolTbl.ProcStack.top());
24        TmpLblIdx.m_Label=TmpLbl;
25        TmpLblIdx.m_LabelType=Statement::LabelIdx::FalseLabel;
26        TmpLblIdx.m_idx=TmpLbl.m_iLink;
27        TmpS.m_Labels.push_back(TmpLblIdx);
28        TmpLbl.m_iType=OpInfo::LABEL;
29        TmpLbl.m_iLink=SymbolTbl.GetTmpLabel(SymbolTbl.ProcStack.top());
30        TmpLblIdx.m_Label=TmpLbl;
31        TmpLblIdx.m_LabelType=Statement::LabelIdx::ExitLabel;
32        TmpLblIdx.m_idx=TmpLbl.m_iLink;
33        TmpS.m_Labels.push_back(TmpLblIdx);
34        CurrentStatement.push(TmpS);
35        SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
36            OpType::JNT,Tmp,TmpS.GetLabel(Statement::LabelIdx::FalseLabel),-1));
37        return true;
```

```
38     }
39     return true;
40 }
```

第 3 行: OpData 栈就是表达式翻译时使用的辅助栈结构。if 语句中的条件表达式的运算结果就是 OpData 的栈顶元素。因此,程序必须判断 OpData 是否为空。

第 10 行: 这个条件语句判断 OpData 栈顶元素是否为 BOOLEAN 类型。在 Pascal 中,if 语句的条件表达式类型必须为 BOOLEAN 类型,这里的类型检查是非常严格的。

第 17 行: 声明一个 Statement 实例。

第 21 行: 在翻译 if 语句时, m_bIsElse 是非常有用的。当然, m_bIsElse 的缺省值应该是 false,仅当编译器分析到相应的 else 子句时,才将该属性重置为 true。

第 22~26 行: 生成一个临时标号,并将其设置为当前 if 语句的假分支标号。实际上,在任何编译器中,临时标号都是静态的且全局有效的,并不是临时存在的,称其为“临时标号”仅因为它是编译过程中由编译器自动产生的而已。Neo Pascal 调用了 SymbolTbl 的静态方法 GetTmpLabel 生成一个临时标号,其返回值即为该临时标号在标号信息表中的位序号。GetTmpLabel 方法的源程序列表稍后给出。

第 27 行: m_Labels 是 TmpS 对象的标号集合,主要用于存储当前语句相关的标号信息。m_Labels 集合中的标号不存在先后顺序。

第 28~33 行: 主要的工作是生成出口标号,其过程与假分支标号处理基本类似。

第 35~36 行: 生成 JNT 指令,并将其加入 IR 列表。通常, JNT 指令需要提供两个操作数,即条件变量(常量)及跳转标号,程序调用 EmitIR 函数统一生成 IR 指令。

程序 5-5 semantic.cpp

相关文法:

```
if 语句后部 → else 069 语句 | ε
1  bool semantic069()
2  {
3      if (CurrentStatement.empty() || CurrentStatement.top().m_eType!=Statement::IF)
4      {
5          EmitError("ELSE 子句必须与 IF 配合使用",TokenList.at(iListPos-1));
6          return false;
7      }
8      Statement TmpS;
9      CurrentStatement.top().m_bIsElse=true;
10     TmpS=CurrentStatement.top();
11     SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
12         OpType::JMP,TmpS.GetLabel(Statement::LabelIdx::ExitLabel)));
13     SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
14         OpType::LABEL,TmpS.GetLabel(Statement::LabelIdx::FalseLabel)));
15     return true;
16 }
```



第 3 行：有效性判断。实际上，语法分析完全有能力保证 else 子句位置有效性，因此。这个判断是可以省略的。

第 9 行：设置 m_bIsElse 属性的目的就是便于 semantic070 生成适合的出口标号。仔细分析后，不难发现，semantic070 可能遇到两种情景，即 if-then 和 if-then-else 结构。针对这两种结构，semantic070 生成的出口标号是不同的，所以需要明确地标识输入语句结构是 if-then 还是 if-then-else。

第 11~12 行：生成 JMP 语句。这个 JMP 语句的目标标号就是 if 结构的出口标号。

第 13~14 行：生成 LABEL 语句。这个 LABEL 语句的标号就是 if 结构的假分支标号，即对应翻译方案中的第一个 LABEL 语句的“_L1”。

程序 5-6 semantic.cpp

相关文法：

其他语句 → if 表达式 068 then 语句 if 语句后部 070

```
1 bool semantic070()
2 {
3     if (CurrentStatement.empty() || CurrentStatement.top().m_eType!=Statement::IF)
4     {
5         EmitError("语义错误",TokenList.at(iListPos-1));
6         return false;
7     }
8     Statement TmpS;
9     TmpS=CurrentStatement.top();
10    if (TmpS.m_bIsElse)
11        SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
12            OpType::LABEL,TmpS.GetLabel(Statement::LabelIdx::ExitLabel)));
13    else
14        SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
15            OpType::LABEL,TmpS.GetLabel(Statement::LabelIdx::FalseLabel)));
16    CurrentStatement.pop();
17    return true;
18 }
```

第 3 行：有效性判断。

第 10 行：判断当前正在分析的语句是否含有 else 子句。如果是 if-then-else 结构，标志是由 semantic069 设置的。

第 11、12 行：生成 LABEL 语句。当输入语句为 if-then-else 结构时，则 LABEL 语句的标号即为 if 语句的出口标号。

第 14、15 行：生成 LABEL 语句。当输入语句为 if-then 结构时，则 LABEL 语句的标号即为 if 语句的假分支标号。

第 16 行：当前 if 语句分析完毕，将 CurrentStatement 栈顶元素弹出，这个步骤是非常重要的。

5.5 while/repeat 语句

5.5.1 while 语句的翻译

while 语句也是程序设计语言的最基本结构之一。Pascal 的 while 语句的语义与 C 语言完全一致，这里，笔者不再多作解释了。

while 语句的标准文法如下：

【文法 5-3】

其他语句 → while 071 表达式 073 do 语句 072

下面，先谈谈 while 语句的翻译方案及其与语义子程序的联系。根据 while 语句的语义，可以得到如表 5-7 所示的翻译方案。

表 5-7 while 语句的翻译方案

while 语句	翻译方案
while <表达式> do <语句>	(LABEL ,__L1,null,null) <表达式翻译> (JNT ,<表达式结果>,null,__L0) <语句> (JMP ,__L1,null,null) (LABEL ,__L0,null,null)

在翻译 while 语句时，必须注意两个分支标号，即入口标号、出口标号。在表 5-7 的翻译方案中，可以将“__L1”称为入口标号，而将“__L0”称为出口标号。由于 while 语句的翻译比较简单，笔者就不多作说明了。

5.5.2 源代码实现

先简单介绍一下各相关语义子程序的功能，以便读者理解和阅读源代码。

semantic071: 1、生成入口标号语句，即翻译方案中的第一个 LABEL 语句。

semantic073: 1、判断表达式的结果是否为布尔类型。
2、生成 JNT 语句，目标标号为出口标号。

semantic072: 1、生成 JMP 语句，目标标号为入口标号。
2、生成出口标号语句，即翻译方案中的第二个 LABEL 语句。

下面，就来看看 while 语句相关的语义子程序。

程序 5-7 semantic.cpp

相关文法：

其他语句 → while 071 表达式 073 do 语句 072

```

1 bool semantic071()
2 {
3     Statement TmpS;
4     Statement::LabelIdx TmpLblIdx;
```



```
5     OpInfo TmpLbl;
6     TmpS.m_eType=Statement::WHILE;
7     TmpLbl.m_iType=OpInfo::LABEL;
8     TmpLbl.m_iLink=SymbolTbl.GetTmpLabel(SymbolTbl.ProcStack.top());
9     TmpLblIdx.m_Label=TmpLbl;
10    TmpLblIdx.m_LabelType=Statement::LabelIdx::ExitLabel;
11    TmpLblIdx.m_iIdx=TmpLbl.m_iLink;
12    TmpS.m_Labels.push_back(TmpLblIdx);
13    TmpLbl.m_iType=OpInfo::LABEL;
14    TmpLbl.m_iLink=SymbolTbl.GetTmpLabel(SymbolTbl.ProcStack.top());
15    TmpLblIdx.m_Label=TmpLbl;
16    TmpLblIdx.m_LabelType=Statement::LabelIdx::EntryLabel;
17    TmpLblIdx.m_iIdx=TmpLbl.m_iLink;
18    TmpS.m_Labels.push_back(TmpLblIdx);
19    SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
20        OpType::LABEL,TmpS.GetLabel(Statement::LabelIdx::EntryLabel)));
21    CurrentStatement.push(TmpS);
22    return true;
23 }
```

semantic071 的语义动作比较简单，就不再逐行分析。这里，读者应该注意一个要点，那就是严格区别入口标号与出口标号。相对而言，while 结构比 if 结构更容易处理，因为并不需要区别类似于 else 的子句。Pascal 的 while 是遵循经典 while 结构设置的，当然，在某些现代程序设计语言中，while 结构可能会稍复杂，读者也应该注意。

程序 5-8 semantic.cpp

相关文法：

其他语句 → while 071 表达式 073 do 语句 072

```
1 bool semantic073()
2 {
3     if (CurrentStatement.empty() || CurrentStatement.top().m_eType!=Statement::WHILE)
4     {
5         EmitError("语义错误",TokenList.at(iListPos-1));
6         return false;
7     }
8     if (OpData.empty())
9     {
10        EmitError("语义错误",TokenList.at(iListPos-1));
11        return false;
12    }
13    OpInfo Tmp=OpData.top();
14    OpData.pop();
15    if (CType::GetOpType(Tmp)!=StoreType::T_BOOLEAN)
16    {
```

```

17         EmitError("WHILE 语句判定表达式必须为 BOOLEAN 类型",TokenList.at(iListPos-1));
18         return false;
19     }
20     else
21     {
22         Statement TmpS;
23         TmpS=CurrentStatement.top();
24         SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
25             OpType::JNT,Tmp,TmpS.GetLabel(Statement::LabelIdx::ExitLabel)));
26     }
27     return true;
28 }

```

这里，读者应注意有效性检查，主要是判断表达式结果的类型是否为布尔类型。这是 Pascal 语义强制规定的，不必评说其优劣。对于编译器设计者，这是应当遵守的设计准则。

程序 5-9 semantic.cpp

相关文法:

其他语句 → while 071 表达式 073 do 语句 072

```

1  bool semantic073()
2  {
3      if (CurrentStatement.empty() || CurrentStatement.top().m_eType!=Statement::WHILE)
4      {
5          EmitError("语义错误",TokenList.at(iListPos-1));
6          return false;
7      }
8      Statement TmpS;
9      TmpS=CurrentStatement.top();
10     SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
11         OpType::JMP,TmpS.GetLabel(Statement::LabelIdx::EntryLabel)));
12     SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
13         OpType::LABEL,TmpS.GetLabel(Statement::LabelIdx::ExitLabel)));
14     CurrentStatement.pop();
15     return true;
16 }

```

在很多情况下，语句翻译的语义子程序是存在一定的相似的。从 while 语句与 if 语句翻译的语义子程序中，就不难得到这一结论。实际上，这种现象在翻译控制结构时较为常见。例如，for、while、repeat、if 等都存在一定的雷同，即使它们本身的语义存在着较大的差异。因此，针对这几类控制结构的翻译，笔者不再对源代码进行逐行解释。在理解了 while 语句翻译的基础上，相信读者应该可以顺利地阅读相关源代码。



5.5.3 repeat 语句的翻译

repeat 语句也是程序设计语言的最基本结构之一。不过，Pascal 的 repeat 语句与 C 语言的 do-while 语句稍有不同。C 语言的 do-while 语句规定其结束循环的条件是循环表达式的计算结果为假（即为 0），而 Pascal 的 repeat 语句规定其结束循环的条件是循环表达式的计算结果为真（即为 true）。

repeat 语句的标准文法如下：

【文法 5-4】

其他语句 → repeat 074 语句序列 until 表达式 075

下面，先来看看 repeat 语句的翻译方案及其与语义子程序的联系。根据 repeat 语句的语义，可以得到表 5-8 中的翻译方案：

表 5-8 repeat 语句的翻译方案

repeat 语句	翻译方案
repeat <语句> until <表达式>	(LABEL , __L0,null,null) <语句> <表达式翻译> (JNT ,<表达式结果>,null,__L0) (LABEL , __L1,null,null)

翻译 repeat 语句时，必须注意两个分支标号，即入口标号、出口标号。在翻译方案中，可以将“__L0”称为入口标号，而将“__L1”称为出口标号。这里，有读者可能有疑问：标号“__L1”是否多余？当然不是。虽然就翻译方案而言，“__L1”并未被使用，但是，必须考虑循环结构中出现 break 语句的情况。实际上，通常将 break 语句翻译为 JMP 指令，其目标标号就是循环的出口标号。

由于 repeat 语句与 while 语句的翻译非常类似，这里，笔者就不再详细给出源代码列表了，请读者自行参考 Neo Pascal 的实现。

5.6 for 语句

5.6.1 for 语句的翻译

Pascal 的 for 语句与 C 语言的 for 语句存在较大差别。标准 Pascal 的 for 仅作为计数循环使用，所以其语义限制较多。例如，标准 Pascal 的 for 语句仅允许使用简单变量作为循环变量，不允许将数组元素或记录字段作为循环变量。其次，标准 Pascal 的 for 语句的步长只能取 1 或-1。这些差异是 Pascal 初学者值得注意的。

for 语句的标准文法如下：

【文法 5-5】

其他语句 → for 标识符 076 := 表达式 077 for 语句后部

for 语句后部 → to 表达式 078 do 语句 079
 → downto 表达式 080 do 语句 079

这里，先解释一下 for 语句的特点：

(1) 在标准 for 结构中有两个表达式，通常将前者称为“循环初始表达式”，而将后者称为“循环终止表达式”，这两个表达式的计算结果类型必须是有序类型。

(2) 标准 Pascal 规定 for 结构有两种形式：即 for-to-do、for-downto-do。两种形式的差别就在于其步长不同，前者的步长为 1，而后的步长为-1。

(3) 标准 Pascal 规定 for 语句的循环变量不得在循环体内改变其值。不过，在 Neo Pascal 中，并没有作这一限制。

下面，再来看看 for 语句的翻译方案及其与语义子程序的联系。根据 for 语句的语义，可以得到如表 5-9 所示的翻译方案：

表 5-9 for 语句的翻译方案

for 语句	翻译方案
for <循环变量>:=<表达式 1> to <表达式 2> do <语句>	<表达式 1 翻译> (ASSIGN_4 ,<表达式 1 结果>,null,<循环变量>) (LABEL , __L1,null,null) <表达式 2 翻译> (LE_4 ,<循环变量>,<表达式 2 结果>,_T) (JNT , _T,null, __L0) <语句> (LABEL , __L2,null,null) (ADD_4 ,<循环变量>,1,<循环变量>) (JMP , __L1,null,null) (LABEL , __L0,null,null)
for <循环变量>:=<表达式 1> downto <表达式 2> do <语句>	<表达式 1 翻译> (ASSIGN_4 ,<表达式 1 结果>,null,<循环变量>) (LABEL , __L1,null,null) <表达式 2 翻译> (ME_4 ,<循环变量>,<表达式 2 结果>,_T) (JNT , _T,null, __L0) <语句> (LABEL , __L2,null,null) (SUB_4 ,<循环变量>,1,<循环变量>) (JMP , __L1,null,null) (LABEL , __L0,null,null)

翻译方案中“_T”表示一个由编译器分配的临时变量。在翻译 for 语句时，同样需要注意循环的入口、出口分支的标号。例如，翻译方案中的“__L1”就是入口分支的标号，而“__L0”就是出口分支的标号。另外，读者应该注意 for 语句的入口分支标号的位置，根据标准 Pascal 的语义，入口标号只能置于循环初始表达式计算及循环变量赋初值之后。在翻译方案中，“__L2”是 continue 语句的目标标号。对于 continue 语句而言，for 语句的处理与 while、repeat 语句的处理是不同的。在处理 for 语句时，必须将 continue 的目标标号置于循环变量修改之前，否则与 continue 语句通用语义不符。



5.6.2 源代码实现

笔者先简单介绍一下各语义子程序的功能，以便读者理解和阅读源代码。

semantic076: 1、检查循环变量是否有效。

2、生成 Statement 对象。

semantic077: 1、生成循环变量赋初值语句。

2、生成入口标号语句，即翻译方案中的第一个 LABEL 语句。

semantic078: 1、有效性检查。

2、生成 LE_4 语句。

3、生成 JNT 语句，目标标号为出口标号。

semantic080: 1、有效性检查。

2、生成 ME_4 语句。

3、JNT 语句，目标标号为出口标号。

semantic079: 1、生成 ADD_4/SUB_4 语句，以改变循环变量。

2、生成 JMP 语句，目标标号为入口标号。

3、生成出口标号语句，即翻译方案中的第二个 LABEL 语句。

下面，再来看看 for 语句的相关语义子程序的实现。

程序 5-10 semantic.cpp

相关文法:

其他语句 → for 标识符 076 := 表达式 077 for 语句后部

```
1  bool semantic076()
2  {
3      string szTmp=TokenList.at(iListPos-1).m_szContent;
4      int i=SymbolTbl.SearchVarInfoTbl(SymbolTbl.ProcStack.top(),szTmp);
5      if (i== -1)
6      {
7          i=SymbolTbl.SearchVarInfoTbl(0,szTmp);
8          if (i== -1)
9          {
10             EmitError("循环变量未定义",TokenList.at(iListPos-1));
11             return false;
12         }
13     }
14     Statement TmpS;
15     Statement::LabelIdx TmpLblIdx;
16     OpInfo TmpLbl;
17     TmpS.m_eType=Statement::FOR;
18     TmpS.m_iLoopVar=i;
19     TmpLbl.m_iType=OpInfo::LABEL;
20     TmpLbl.m_iLink=SymbolTbl.GetTmpLabel(SymbolTbl.ProcStack.top());
21     TmpLblIdx.m_Label=TmpLbl;
22     TmpLblIdx.m_LabelType=Statement::LabelIdx::ExitLabel;
```

```

23     TmpLblIdx.m_idx=TmpLbl.m_iLink;
24     TmpS.m_Labels.push_back(TmpLblIdx);
25     TmpLbl.m_iType=OpInfo::LABEL;
26     TmpLbl.m_iLink=SymbolTbl.GetTmpLabel(SymbolTbl.ProcStack.top());
27     TmpLblIdx.m_Label=TmpLbl;
28     TmpLblIdx.m_LabelType=Statement::LabelIdx::EntryLabel;
29     TmpLblIdx.m_idx=TmpLbl.m_iLink;
30     TmpS.m_Labels.push_back(TmpLblIdx);
31     TmpLbl.m_iType=OpInfo::LABEL;
32     TmpLbl.m_iLink=SymbolTbl.GetTmpLabel(SymbolTbl.ProcStack.top());
33     TmpLblIdx.m_Label=TmpLbl;
34     TmpLblIdx.m_LabelType=Statement::LabelIdx::TrueLabel;
35     TmpLblIdx.m_idx=TmpLbl.m_iLink;
36     TmpS.m_Labels.push_back(TmpLblIdx);
37     CurrentStatement.push(TmpS);
38     return true;
39 }

```

第 4、5 行：检索当前过程函数的变量信息表，判断循环变量是否为局部变量。

第 7、8 行：检索主函数的变量信息表，判断循环变量是否为全局变量。注意，两次检索存在优先级差别。标准 Pascal 规定循环变量只能是简单变量，故不必考虑复杂类型的情况。

第 18 行：设置 `m_iLoopVar` 属性。这个属性的值就是循环变量在变量信息表中的位序号。这个属性对于 `for` 语句的翻译是非常重要的。

第 19~24 行：生成 `for` 语句的出口标号。

第 25~29 行：生成 `for` 语句的入口标号。

第 31~35 行：生成 `for` 语句的 `continue` 目标标号。

程序 5-11 semantic.cpp

相关文法：

其他语句 → `for` 标识符 076 := 表达式 077 `for` 语句后部

```

1  bool semantic077()
2  {
3      if (OpData.empty() || CurrentStatement.empty() ||
4          CurrentStatement.top().m_eType!=Statement::FOR)
5      {
6          EmitError("语义错误",TokenList.at(iListPos-1));
7          return false;
8      }
9      OpInfo Op1,Rslt;
10     Op1=OpData.top();
11     OpData.pop();
12     Rslt.m_iType=OpInfo::VAR;
13     Rslt.m_iLink=CurrentStatement.top().m_iLoopVar;
14     if (!CType::IsOrd(Op1) || CType::TypeCompatible(Op1,Rslt,17)==-1)
15     {

```



```
16         EmitError("循环变量的初值不为有序类型或者类型不兼容",TokenList.at(iListPos-1));
17         return false;
18     }
19     if (!GenIR(Op1,Rslt,BasicOpType::MOV))
20         return false;
21     SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
22         OpType::LABEL,CurrentStatement.top().GetLabel(Statement::LabelIdx::EntryLabel)));
23     return true;
24 }
```

第 14 行：循环初值表达式的结果类型必须为有序类型且与循环变量类型兼容。这里，调用了两个函数来完成判断，即 `IsOrd`、`TypeCompatible`。这两个函数是类型系统的相关实现，这里不必深究，只需了解其基本功能即可。

第 19 行：生成循环变量赋初值语句。这里，笔者调用 `GenIR` 函数来实现这一功能。实际上，读者千万不要将循环变量赋初值的问题想得过于简单。如果循环初值表达式的类型与循环变量的类型是完全一致的，问题可能简单一些。不过，当两者类型仅仅是兼容的，则不得不考虑类型转换，这可能使实现复杂度大大增加。

第 22 行：生成 LABEL 指令。

`semantic077` 中调用了三个函数，即 `IsOrd`、`TypeCompatible`、`GenIR`。下面，笔者对这三个函数的参数及功能作简单介绍，并不深究其实现。

【声明 5-5】

```
bool CType::IsOrd(OpInfo Op)
```

该函数主要用于判断操作数 `Op` 的类型是否为有序类型。在 Pascal 语言中，不少语句、表达式对类型是否有序是有一定限制的。因此，有序类型的判断是有现实意义的。`Op` 可以是常量、变量等。在标准 Pascal 中，关于有序类型有非常详细的定义。

【声明 5-6】

```
int CType::TypeCompatible( OpInfo Op1, OpInfo Op2, int Op)
```

该函数主要用于判断两个操作数（即 `Op1`、`Op2`）的类型是否兼容。在类型系统中，讨论类型是否兼容通常是针对某一运算而言的。例如，`integer` 和 `real` 类型的变量进行比较运算时，两者是兼容的。但如果试图用 `real` 类型的变量为 `integer` 类型的变量赋值时，那么，两者就是不兼容的。这里，参数 `Op` 就是用于说明两个操作数所处的运算情景。注意，`TypeCompatible` 允许传递的运算情景就是运算符 ID（与词法分析器中的定义一致）。在 `semantic077` 第 14 行中，`CType::TypeCompatible(Op1, Rslt, 17)` 的作用就是判断将 `Op1` 赋给 `Rslt` 时，类型是否兼容。根据词法分析器的定义，赋值运算符“`:=`”的 ID 就是 17。

下面解释一下 `TypeCompatible` 函数返回值的含义。当函数返回 -1 时，即表示两者在给定运算情景下是不兼容的。当函数返回其他值（可能是 1、2、3 等）时，则表示两者是兼容的，至于不同正整数常量的具体含义将在第 6 章详述。

实际上，在类型系统中，`TypeCompatible` 函数还存在几个不同形式的重载，以满足不同的需要。这里，不再逐一列举。

【声明 5-7】

```
bool GenIR(OpInfo Op1, OpInfo Op2, OpInfo Rslt, BasicOpType Op)
```

GenIR 函数是一个用于生成 IR 的重要接口，其主要功能就是根据操作数类型与相应的语义生成所需的 IR。由于 IR 操作数个数的不同，所以 GenIR 函数也存在几个不同形式的重载，以满足相应的需要。那么，为什么要设计 IR 生成接口呢？

在高级语言编译器中，由于类型种类繁多，这可能使得 IR 生成变得比较复杂。在编译过程中，即使是在操作数类型兼容（但不等价）的情况下，仍需要考虑隐式类型转换（强制类型转换）、IR 指令选择等问题。设计 GenIR 函数的目的就是使得语义子程序只需关心表达式的语义及各操作数的来源，而不必关注隐式类型转换、IR 操作符选择等细节。

这里，笔者解释一下其中的一个参数——BasicOpType Op。注意，这里的 Op 并不是 IR 的操作符，而只是一个描述运算情景的枚举常量。例如，BasicOpType::MOV 就表示其语义是将 Op1 赋给 Rslt，至于最终生成的 IR 操作符是 ASSIGN_S 还是 ASSIGN_4 或者其他操作符，完全是由传入的操作数类型与类型系统决定的。更多细节说明请读者参考第 6 章。

程序 5-12 semantic.cpp

相关文法：

for 语句后部 → to 表达式 078 do 语句 079

```

1  bool semantic078()
2  {
3      if (OpData.empty() || CurrentStatement.empty() ||
4          CurrentStatement.top().m_eType!=Statement::FOR)
5      {
6          EmitError("语义错误",TokenList.at(iListPos-1));
7          return false;
8      }
9      OpInfo Op2,Op1;
10     Op2=OpData.top();
11     OpData.pop();
12     Op1.m_iType=OpInfo::VAR;
13     Op1.m_iLink=CurrentStatement.top().m_iLoopVar;
14     CurrentStatement.top().m_bIsDownto=false;
15     int TmpResult=CType::TypeCompatible(Op1,Op2,20);
16     if (!CType::IsOrd(Op2) || TmpResult== -1)
17     {
18         EmitError("循环变量的终值不为有序类型或者类型不兼容",TokenList.at(iListPos-1));
19         return false;
20     }
21     OpInfo TmpRslt;
22     TmpRslt.m_iType=OpInfo::VAR;
23     TmpRslt.m_iLink=SymbolTbl.GetTmpVar(SymbolTbl.ProcStack.top(),
24         (StoreType)SymbolTbl.TypeSysTbl.at(TmpResult).m_iRsltType);
25     OpVarSemantic(Op1,Op2,TmpRslt,SymbolTbl.TypeSysTbl.at(TmpResult));
26     SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
27         OpType::JNT,TmpRslt,CurrentStatement.top().GetLabel(

```



```
28         Statement::LabelIdx::ExitLabel));
29     return true;
30 }
```

第 16 行：判断循环终值表达式的结果是否为有序类型。并且判断在进行比较运算时循环终值类型与循环变量类型是否兼容。注意，for-to-do 结构的循环条件是循环变量小于或等于循环终值表达式的结果，所以 `TypeCompatible(Op1, Op2, 20)` 中的 20 就是终结符 “<=” 的 ID。

第 22~25 行：生成一个临时变量，将其作为循环变量与终值表达式比较语句的结果变量，并且生成比较语句。注意，这里调用了 `OpVarSemantic` 函数实现比较语句的生成。在此，不必深究 `OpVarSemantic` 函数的实现，只需理解其功能与 `GenIR` 函数是类似的即可。

第 26~28 行：生成 JNT 语句，其目标标号为循环的出口标号。

程序 5-13 semantic.cpp

相关文法：

for 语句后部 → to 表达式 080 downto 语句 079

```
1  bool semantic080()
2  {
3      if (OpData.empty() || CurrentStatement.empty() ||
4          CurrentStatement.top().m_eType!=Statement::FOR)
5      {
6          EmitError("语义错误",TokenList.at(iListPos-1));
7          return false;
8      }
9      OpInfo Op2,Op1;
10     Op2=OpData.top();
11     OpData.pop();
12     Op1.m_iType=OpInfo::VAR;
13     Op1.m_iLink=CurrentStatement.top().m_iLoopVar;
14     CurrentStatement.top().m_bIsDownto=true;
15     int TmpResult=CType::TypeCompatible(Op1,Op2,22);
16     if (!CType::IsOrd(Op2) || TmpResult===-1)
17     {
18         EmitError("循环变量的终值不为有序类型或者类型不兼容",TokenList.at(iListPos-1));
19         return false;
20     }
21     OpInfo TmpRslt;
22     TmpRslt.m_iType=OpInfo::VAR;
23     TmpRslt.m_iLink=SymbolTbl.GetTmpVar(SymbolTbl.ProcStack.top(),
24         (StoreType)SymbolTbl.TypeSysTbl.at(TmpResult).m_iRsltType);
25     OpVarSemantic(Op1,Op2,TmpRslt,SymbolTbl.TypeSysTbl.at(TmpResult));
26     SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
27         OpType::JNT,TmpRslt,CurrentStatement.top().GetLabel(
28         Statement::LabelIdx::ExitLabel)));
29     return true;
30 }
```

semantic080 与 semantic078 有两个主要区别:

(1) 第 14 行中 m_bIsDownto 属性的设置不同。由于两个语义子程序所处的文法不同, 所以该属性的值是完全不同的。

(2) 第 15 行中调用 TypeCompatible 函数的传参不同, 即第三个参数的值不同。在 for-to-do 结构中, 循环条件是循环变量的值小于或等于循环终止表达式的结果, 所以将终结符 “<=” 的 ID(20) 作为调用 TypeCompatible 函数的实参。而在 for-downto-do 结构中, 循环条件是循环变量的值大于或等于循环终止表达式的结果, 所以将终结符 “>=” 的 ID(22) 作为调用 TypeCompatible 函数的实参。

程序 5-14 semantic.cpp

相关文法:

for 语句后部 → to 表达式 080 downto 语句 079
 → to 表达式 078 do 语句 079

```

1  bool semantic079()
2  {
3      if (CurrentStatement.empty() || CurrentStatement.top().m_eType!=Statement::FOR)
4      {
5          EmitError("语义错误",TokenList.at(iListPos-1));
6          return false;
7      }
8      OpInfo Op1,Op2;
9      Op1.m_iType=OpInfo::VAR;
10     Op1.m_iLink=CurrentStatement.top().m_iLoopVar;
11     Op2.m_iType=OpInfo::CONST;
12     Op2.m_iLink=SymbolTbl.RecConstTbl("1",3);
13     SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
14         OpType::LABEL,CurrentStatement.top().GetLabel(
15             Statement::LabelIdx::TrueLabel)));
16     if (CurrentStatement.top().m_bIsDownto)
17     {
18         if (!GenIR(Op1,Op2,Op1,BasicOpType::SUB))
19             return false;
20     }
21     else
22     {
23         if (!GenIR(Op1,Op2,Op1,BasicOpType::ADD))
24             return false;
25     }
26     SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
27         OpType::JMP,CurrentStatement.top().GetLabel(
28             Statement::LabelIdx::EntryLabel)));
29     SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
30         OpType::LABEL,CurrentStatement.top().GetLabel(
31             Statement::LabelIdx::ExitLabel)));
32     CurrentStatement.pop();
33     return true;
34 }
```




第 13~15 行：生成 continue 语句的目标标号。

第 16~25 行：根据 for 的结构，调用 GenIR 函数生成自减或自增 1 语句。请注意传入 GenIR 函数的第三个参数。

第 26~28 行：生成 JMP 语句，其目标标号为循环入口标号。

第 29~31 行：生成 for 语句的出口标号语句。

第 32 行：当前语句分析完毕，弹出 CurrentStatement 栈顶元素，以保证分析的正确性。

至此，笔者已经详细讲述了 for 语句翻译方案及其实现。当然，由于其中某些函数涉及表达式翻译的话题，所以没有剖析其实现细节。

5.7 case 语句

5.7.1 case 语句的翻译

大多数程序设计语言都支持 case 或 switch 语句。无论是 C 语言的 switch，还是 Pascal 的 case，两者都是作为多路分支结构存在的。先来看看 Pascal 的 case 语句文法。

【文法 5-6】

其他语句	→	case 表达式 084 of case 分支 end 089
case 分支	→	常量列表 087 : 语句 088 ; case 分支 ε
常量列表	→	常量 085 常量列表 1
常量列表 1	→	, 常量 086 常量列表 1 ε

与 C 语言的 switch 语句相比，标准 Pascal 的 case 语句主要有两个特点：

(1) 除了“case”分支外，标准 Pascal 不支持包括“default”分支在内的其他任何形式的分支。不过，由于 C 语言“default”分支的应用还是非常广泛的，因此，有些商用 Pascal 编译器提供了“else”分支，即用于表示 case 语句的默认分支，以满足用户的需要。

(2) 分支是不允许贯穿的。而在 C 语言中，除非显式使用 break 语句，否则 case 分支默认是贯穿的。

了解了 case 语句的特点后，就来看看如何翻译 case 语句。与先前讨论的语句结构不同，case 语句的翻译方案并不是唯一的。在现代编译技术中，有三种比较成熟的翻译方案可供选择。下面简单介绍一下各翻译方案及其评价。

最直观的翻译方案就是将 case 结构转换成一系列条件跳转语句。这种方案是最简单、最易于理解与实现的。不过，该方案生成的 IR 或者汇编代码不够精简，尤其是翻译 C 语言的 switch 语句时，可能会产生许多冗余的跳转及标号。

第二种是散列表方式。散列表方式并不是一种通用的 case 语句翻译方案，将其推广为任意形式的 case 语句的翻译并不容易。散列表方式就是通过构造一个散列表，将分支语句的标号存储在散列表内，以分支值作为关键字进行检索。虽然这是一种比较高效的实现方法，但是以分支值为关键字构造散列表却并不简单。

最后也是应用最广泛的一种翻译方案就是跳转表。跳转表方案的基本思想与散列表方式比较类似，都是通过构造分支值与分支标号的关系，以实现分支跳转的目的。然而，主要的

区别在于跳转表方案生成的目标代码以顺序检索方式扫描跳转表，匹配分支值并实现跳转。在实际编译器设计中，这种翻译方案的应用非常广泛，包括 Delphi、VC++ 等商用编译器都无一例外。不过，各种商用编译器的具体的实现方案还是存在一定差异的。由于 Neo Pascal 也是应用跳转表实现 case 语句翻译的，故本节将着重阐述跳转表翻译方案的实现。下面，就来看一种应用跳转表实现的标准翻译方案，见表 5-10。

表 5-10 跳转表的翻译方案

case 语句	翻译方案
case <表达式> of	//跳转表部分
<常量 1>:<语句 1>;	(JE , <表达式>的结果>, <常量 1>, __L1)
<常量 2>:<语句 2>;	(JE , <表达式>的结果>, <常量 2>, __L2)
.....
<常量 n>:<语句 n>;	(JE , <表达式>的结果>, <常量 n>, __Ln)
end;	(JMP , __L0, null, null)
	//执行语句部分
	(LABEL , __L1, null, null)
<语句 1>	(JMP , __L0, null, null)
	(LABEL , __L2, null, null)
<语句 2>	(JMP , __L0, null, null)

	(LABEL , __Ln, null, null)
<语句 n>	(JMP , __L0, null, null)
	//case 语句的出口标号
	(LABEL , __L0, null, null)

注：有些书籍将 case 语句中的<表达式>称为“选择子”，本书也将沿用这个名词。

关于跳转表方式的说明如下：

(1) 应用跳转表处理 case 分支贯穿的问题（例如，C 语言的 switch 语句）也是非常容易的，只需将执行语句部分中的 JMP 省略即可。

(2) 跳转表的具体形式可能存在差异。有时，跳转表也使用 JZ 或者其他指令描述。用 JZ 指令描述跳转表稍复杂，必须将分支值按升（降）序排列，逐一应用加（减）运算及判零实现分支跳转。

(3) 跳转表的位置并不一定处于执行语句体的上部，有时可能是处于执行语句体的下部，通常视具体的实现而定。

5.7.2 源代码实现

前面，已经讨论了跳转表的基本形式，以及如何应用跳转表实现 case 语句的翻译。本小节将着眼于 Neo Pascal 的实现。

由于跳转表结构的特点，使得编译器不得不把原先纠结在一起的 case 结构拆分成两个部分，即跳转表部分、执行语句部分。如何在一遍扫描过程中完成整个 case 结构的翻译是解决这个问题的关键所在。当然，编译器不得不借助于一些外部容器暂存某些重要的临时 IR



或者符号，以便后续使用。

case 语句翻译的基本步骤如下：

- (1) 在翻译 case 语句前，记录当前 IR 的行号。
- (2) 充分利用 CurrentStatement 的实例，记录 case 分支值与分支标号的对应关系。
- (3) 在一遍扫描过程中，只需将执行语句部分正确无误地生成 IR 即可。
- (4) 在 case 语句翻译完毕后，按记录得到的 case 分支值与分支标号的对应关系生成跳转表，并将其插入执行语句部分之前。

首先，来看看各语义子程序主要功能，以便读者理解和阅读源代码。

semantic084: 1、检查选择子类型是否有效，必须为有序类型。

- 2、生成 Statement 对象。
- 3、分配一个出口标号。

semantic085: 1、检查分支常量与选择子类型是否兼容。检查分支常量是否重复定义。

- 2、分配一个分支标号。
- 3、建立分支标号与分支常量之间的相应关系。

semantic086: 1、检查分支常量与选择子类型是否匹配。检查分支常量是否重复定义。

- 2、建立分支标号与分支常量之间的相应关系。注意，与 semantic085 的区别就是在于 semantic086 不需要分配新的分支标号。

semantic087: 1、生成分支标号语句。

semantic088: 1、生成 JMP 语句，目标标号为出口标号。

semantic089: 1、生成出口标号语句。

- 2、根据分支标号与分支常量的相应关系，生成跳转表，并将其插入到 case 语句相应 IR 序列的开始位置。

程序 5-15 semantic.cpp

相关文法：

其他语句 → case 表达式 084 of case 分支 end 089

```
1  bool semantic084()
2  {
3      if (OpData.empty() || !CType::IsOrd(OpData.top()))
4      {
5          EmitError("CASE 的选择子必须为有序类型",TokenList.at(iListPos-1));
6          return false;
7      }
8      OpInfo Tmp,TmpLbl;
9      Tmp=OpData.top();
10     OpData.pop();
11     Statement TmpS;
12     TmpS.m_eType=Statement::CASE;
13     TmpS.m_CaseExp=Tmp;
14     TmpS.m_iLoopVar=SymbolTbl.ProcInfoTbl.at(
15         SymbolTbl.ProcStack.top()).m_Codes.size();
16     Statement::LabelIdx TmpLblIdx;
17     TmpLbl.m_iType=OpInfo::LABEL;
```

```

18     TmpLbl.m_iLink=SymbolTbl.GetTmpLabel(SymbolTbl.ProcStack.top());
19     TmpLblIdx.m_Label=TmpLbl;
20     TmpLblIdx.m_LabelType=Statement::LabelIdx::ExitLabel;
21     TmpLblIdx.m_idx=TmpLbl.m_iLink;
22     TmpS.m_Labels.push_back(TmpLblIdx);
23     CurrentStatement.push(TmpS);
24     return true;
25 }

```

第 3 行：选择子类型的有效性判断。Pascal、C 等都规定选择子必须为有序类型。

第 13 行：在生成跳转表时，选择子是必不可少的操作数，所以必须将其暂存。

第 14 行：记录当前 IR 的行号。编译器先翻译 case 语句的执行语句部分，再生成跳转表，并插入执行语句部分之前。因此，在正式翻译执行语句部分之前，必须记录当前 IR 的位置，即为以后跳转表的插入点。

第 19~25 行：生成 case 语句的出口标号。

程序 5-16 semantic.cpp

相关文法：

常量列表 → 常量 085 常量列表 1

```

1  bool semantic085()
2  {
3      OpInfo Tmp;
4      Tmp.m_iType=OpInfo::CONST;
5      Tmp.m_iLink=atoi(TokenList.at(iListPos-1).m_szContent.c_str());
6      if (CurrentStatement.empty() || CurrentStatement.top().m_eType!=Statement::CASE)
7      {
8          EmitError("语义错误",TokenList.at(iListPos-1));
9          return false;
10     }
11     if (CType::TypeCompatible(Tmp,CurrentStatement.top().m_CaseExp,17)==-1)
12     {
13         EmitError("case 子句的常量与选择子的类型不匹配",TokenList.at(iListPos-1));
14         return false;
15     }
16     int iConst1=SymbolTbl.ConstInfoTbl.at(Tmp.m_iLink).m_iVal;
17     int iConst2;
18     for (int i=0;i<CurrentStatement.top().m_Labels.size();i++)
19     {
20         if (CurrentStatement.top().m_Labels.at(i).m_LabelType!=
21             Statement::LabelIdx::CaseLabel)
22             continue;
23         iConst2=SymbolTbl.ConstInfoTbl.at(CurrentStatement.top().
24             m_Labels.at(i).m_iConst).m_iVal;
25         if (iConst1==iConst2)
26         {
27             EmitError("case 分支常量重复定义",TokenList.at(iListPos-1));

```



```
28         return false;
29     }
30 }
31 OpInfo TmpLbl;
32 Statement::LabelIdx TmpLblIdx;
33 TmpLbl.m_iType=OpInfo::LABEL;
34 TmpLbl.m_iLink=SymbolTbl.GetTmpLabel(SymbolTbl.ProcStack.top());
35 TmpLblIdx.m_Label=TmpLbl;
36 TmpLblIdx.m_LabelType=Statement::LabelIdx::CaseLabel;
37 TmpLblIdx.m_iIdx=TmpLbl.m_iLink;
38 TmpLblIdx.m_iConst=atoi(TokenList.at(iListPos-1).m_szContent.c_str());
39 CurrentStatement.top().m_Labels.push_back(TmpLblIdx);
40 return true;
}
```

第 3~4 行：获取分支常量。由于后续程序将调用 `TypeCompatible` 函数判断分支常量与选择子的类型是否兼容，因此需要将分支常量打包成 `OpInfo` 对象。

第 11 行：调用 `TypeCompatible` 函数判断分支常量与选择子的类型是否兼容。

第 17~29 行：判断分支常量是否存在重复定义。在大多数高级语言中，都不允许类似于 `case` 的多分支结构中存在重值分支常量。这里，必须通过遍历已有的分支标号信息，检查是否存在分支常量重值的现象。注意，遍历时只需检查类型为 `CaseLabel` 的标号及其相关信息，不需要也不能够检查其他类型的标号，否则可能导致 Neo Pascal 执行异常。因此，第 19~20 行的判断是非常重要的，绝不容忽视。

第 32~38 行：生成 `case` 语句的分支标号。`TmpLblIdx` 对象就是用于存储分支标号与分支常量的关系，其中，`m_iIdx` 属性即表示编译器分配的临时标号在标号信息表中的位序号，而 `m_iConst` 属性则用于记录分支常量。

程序 5-17 semantic.cpp

相关文法：

常量列表 1 → , 常量 086 常量列表 1

```
1  bool semantic086()
2  {
3      OpInfo Tmp;
4      Tmp.m_iType=OpInfo::CONST;
5      Tmp.m_iLink=atoi(TokenList.at(iListPos-1).m_szContent.c_str());
6      if (CurrentStatement.empty() || CurrentStatement.top().m_eType!=Statement::CASE)
7      {
8          EmitError("语义错误",TokenList.at(iListPos-1));
9          return false;
10     }
11     if (CType::TypeCompatible(Tmp,CurrentStatement.top().m_CaseExp,17)==-1)
12     {
13         EmitError("case 子句的常量与选择子的类型不匹配",TokenList.at(iListPos-1));
14         return false;
15     }
```

```

15     }
16     int iConst1=SymbolTbl.ConstInfoTbl.at(Tmp.m_iLink).m_iVal;
17     int iConst2;
18     for (int i=0;i<CurrentStatement.top().m_Labels.size();i++)
19     {
20         if (CurrentStatement.top().m_Labels.at(i).m_LabelType!=
21             Statement::LabelIdx::CaseLabel)
22             continue;
23         iConst2=SymbolTbl.ConstInfoTbl.at(CurrentStatement.top().
24             m_Labels.at(i).m_iConst).m_iVal;
25         if (iConst1==iConst2)
26         {
27             EmitError("case 分支常量重复定义",TokenList.at(iListPos-1));
28             return false;
29         }
30     }
31     Statement::LabelIdx TmpLbIdx;
32     TmpLbIdx.m_Label=CurrentStatement.top().m_Labels.at(
33         CurrentStatement.top().m_Labels.size()-1).m_Label;
34     TmpLbIdx.m_LabelType=Statement::LabelIdx::CaseLabel;
35     TmpLbIdx.m_iIdx=CurrentStatement.top().m_Labels.at(
36         CurrentStatement.top().m_Labels.size()-1).m_iIdx;
37     TmpLbIdx.m_iConst=atoi(TokenList.at(iListPos-1).m_szContent.c_str());
38     CurrentStatement.top().m_Labels.push_back(TmpLbIdx);
39     return true;
40 }

```

仔细分析 case 语句的语法，不难发现 semantic086 主要用于处理一个 case 分支所属的分支常量个数大于 1 的情况。那么，也就不难理解为什么 semantic086 不需要分配临时标号了，这是与 semantic085 最主要的差异。semantic086 将分支常量与当前正在分析的分支标号（即当前语句标号集中最末的标号对象）关联。

程序 5-18 semantic.cpp

相关文法：

```

case 分支      →      常量列表 087 : 语句 088 ; case 分支
1  bool semantic087()
2  {
3      if (CurrentStatement.empty() || CurrentStatement.top().m_eType!=Statement::CASE)
4      {
5          EmitError("语义错误",TokenList.at(iListPos-1));
6          return false;
7      }
8      SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(
9      EmitIR(OpType::LABEL,CurrentStatement.top().m_Labels.back().m_Label));
10     return true;
11 }

```



以上代码生成 case 分支的 LABEL 语句。而这里的分支标号就是由 semantic085 分配的。

程序 5-19 semantic.cpp

相关文法:

case 分支 → 常量列表 087: 语句 088; case 分支

```
1  bool semantic088()
2  {
3      if (CurrentStatement.empty() || CurrentStatement.top().m_eType!=Statement::CASE)
4      {
5          EmitError("语义错误",TokenList.at(iListPos-1));
6          return false;
7      }
8      SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
9      OpType::JMP,CurrentStatement.top().GetLabel(Statement::LabelIdx::ExitLabel));
10     return true;
11 }
```

由于 Pascal 的 case 分支之间是不允许贯穿的，所以当语义子程序分析完一个 case 分支后，必须生成一句跳转到出口标号的 JMP 语句，以保证 case 分支不被贯穿。

程序 5-20 semantic.cpp

相关文法:

其他语句 → case 表达式 084 of case 分支 end 089

```
1  bool semantic089()
2  {
3      if (CurrentStatement.empty() || CurrentStatement.top().m_eType!=Statement::CASE)
4      {
5          EmitError("语义错误",TokenList.at(iListPos-1));
6          return false;
7      }
8      OpInfo Op1,Op2;
9      Op1=CurrentStatement.top().m_CaseExp;
10     int iOpSize=0;
11     if (!Op1.m_iDetailType.empty())
12         iOpSize=SymbolTbl.TypeInfoTbl.at(Op1.m_iDetailType.top().m_iLink).m_iSize;
13     else
14         iOpSize=SymbolTbl.TypeInfoTbl.at(CType::GetRealType(SymbolTbl.VarInfoTbl.at(
15         Op1.m_iLink).m_iTypeLink)).m_iSize;
16     if (iOpSize>4)
17     {
18         EmitError("CASE 语句的条件变量必须为有序类型",TokenList.at(iListPos-1));
19         return false;
20     }
21     Op2.m_iType=OpInfo::CONST;
22     vector<IRCode> TmpIRList;
23     SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
```

```
24         OpType::LABEL,CurrentStatement.top().GetLabel(  
25         Statement::LabelIdx::ExitLabel));  
26     for (int i=CurrentStatement.top().m_Labels.size()-1;i>0;i--)  
27     {  
28         Op2.m_iLink=CurrentStatement.top().m_Labels.at(i).m_iConst;  
29         TmpIRList.push_back(EmitIR(OpType(OpType::JE_1+iOpSize/2),Op1,Op2,  
30         CurrentStatement.top().m_Labels.at(i).m_Label));  
31     }  
32     TmpIRList.push_back(EmitIR(OpType::JMP,CurrentStatement.top().GetLabel(  
33         Statement::LabelIdx::ExitLabel));  
34     vector<IRCode>::iterator  
35     it=SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.begin();  
36     it=it+CurrentStatement.top().m_iLoopVar;  
37     SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.insert(it,  
38         TmpIRList.begin(),TmpIRList.end());  
39     CurrentStatement.pop();  
40     return true;  
41 }
```

第 11~15 行：在 Neo Pascal 中，有序类型变量的存储空间可以是 1、2、4 字节。然而，根据 NPIR 的设计，必须依据选择子的实际类型生成不同的 JE 语句（例如，JE_1、JE_2、JE_4）。因此，需要计算选择子所占的存储空间，以便正确生成合适的 JE 语句。这里，读者不必深究这个 if 结构的含义。

第 16~20 行：由于有序类型变量所占的空间只能是 1、2、4 字节，所以必须作严格判断。

第 22 行：临时 IR 列表，用于暂存跳转表。

第 23~25 行：生成 LABEL 语句，即 case 结构中的出口标号的 LABEL 语句。

第 26~31 行：遍历当前 case 语句的所有分支常量，生成分支跳转表，并将其暂存于 TmpIRList 中。

第 32 行：生成跳转表表末的 JMP 语句，其目标标号就是 case 语句的出口标号，以保证跳转表正常工作，不至于贯穿到执行语句部分。

第 35~36 行：查找跳转表的插入位置，并将跳转表加入原 IR 序列。使用迭代器插入时，必须注意顺序容器的插入特性。

5.8 其他语句

5.8.1 break、continue 语句的翻译

ISO/IEC 7185-1990 标准（Pascal 语言标准）并没有提供关于 break、continue 语句的任何描述。不过，由于这两种语句结构确实为编程带来了极大的便利，所以得到了程序员的广泛支持。因此，笔者参考了 C 语言及 Delphi 的实现，将 break、continue 作为 Neo Pascal 的标准语句。Neo Pascal 的 break、continue 语法、语义与 C 语言完全一致，这里就不多作解释了。

在程序设计语言中，类似于 `break`、`continue` 的语句结构并不少见，例如，`Ruby` 语言的 `next`、`redo`、`retry` 等等。通过使用这类语句，程序员可以方便地改变循环控制结构的执行流程。当然，除了 `break`、`continue` 之外，有些高级语言还提供了 `exit`、`quit` 之类的过程（函数）的出口语句。这类语句的文法结构比较简单，所以翻译过程也并不复杂，关键是明确跳转的目标标号。

实际上，了解 `C` 语言 `break`、`continue` 语句的读者应该都明白这两个结构的本质就是一次无条件跳转，只是跳转的目标不同而已。根据 `C` 语言的语义，`break` 语句的跳转目标是最内层循环的出口标号，而 `continue` 语句的跳转目标是最内层循环的入口标号。当然，这并不是绝对的，例如，`Ada` 语言的循环终止语句 `exit` 就允许程序员定义欲跳出的循环名（缺省情况为最内层循环）。处理这类特殊应用时，编译器设计者不得不跟踪输入程序循环的嵌套情况，并记录一些必要的信息。在 `Neo Pascal` 中，笔者使用 `CurrentStatement` 栈详细记录了各语句分析的状况信息，因此，即使 `Pascal` 存在类似于 `Ada` 语言的 `exit` 结构，也可以比较方便地分析与处理。

下面就来看看 `break`、`continue` 语句的具体实现。

程序 5-21 semantic.cpp

相关文法:

```
其他语句      →      break 081

1  bool semantic081()
2  {
3      Statement* pState;
4      int i=CurrentStatement.size()-1;
5      while (i>=0)
6      {
7          pState=&CurrentStatement.c.at(i);
8          if (pState->m_eType==Statement::FOR ||
9              pState->m_eType==Statement::WHILE ||
10             pState->m_eType==Statement::REPEAT)
11             break;
12         i--;
13     }
14     if (i<0)
15     {
16         EmitError("BREAK 语句只能出现在 FOR、WHILE、REPEAT 语句块内"
17                 ,TokenList.at(iListPos-1));
18         return false;
19     }
20     SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
21         OpType::JMP,pState->GetLabel(Statement::LabelIdx::ExitLabel)));
22     return true;
23 }
```

第 5~13 行：遍历 `CurrentStatement` 栈，检索最内层循环。注意，最内层循环并不一定是 `CurrentStatement` 的栈顶元素，所以不能依此判断。在 `Pascal` 语言中，标准的循环结构有

三种：for、while、repeat。

第 14 行：判断是否检索到最内层循环，如果检索失败（即 $i < 0$ ），表示 break 语句没有出现在任何循环体内，编译器必须报错。

第 20~21 行：生成 JMP 语句，目标标号就是最内层循环的出口标号。

程序 5-22 semantic.cpp

相关文法：

```
其他语句      →      continue 082

1  bool semantic082()
2  {
3      Statement* pState;
4      int i=CurrentStatement.size()-1;
5      while (i>=0)
6      {
7          pState=&CurrentStatement.c.at(i);
8          if (pState->m_eType==Statement::FOR ||
9              pState->m_eType==Statement::WHILE ||
10             pState->m_eType==Statement::REPEAT)
11             break;
12         i--;
13     }
14     if (i<0)
15     {
16         EmitError("CONTINUE 语句只能出现在 FOR、WHILE、REPEAT 语句块内"
17                 ,TokenList.at(iListPos-1));
18         return false;
19     }
20     if (pState->m_eType==Statement::FOR)
21     {
22         SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
23                                     OpType::JMP,pState->GetLabel(Statement::LabelIdx::TrueLabel)));
24     }
25     }
26     else
27     {
28         SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
29                                     OpType::JMP,pState->GetLabel(Statement::LabelIdx::EntryLabel)));
30     }
31     return true;
32 }
```

continue 语句的处理与 break 语句类似，主要差别就在于 continue 语句的目标标号。对于 for 语句而言，continue 语句的目标标号是 TrueLabel。然而，对于 repeat、while 语句而言，continue 语句的目标标号是 EntryLabel。

5.8.2 goto 语句的翻译

众所周知，结构化程序设计并不提倡使用 goto 或类似语句。不过，即便如此，包括



Pascal、C 在内的大多数高级语言都提供了 goto 语句。只要正确设计、合理应用，对于某些编程高手（尤其是汇编高手）而言，goto 语句还是有着其他控制结构无法比拟的灵活性与优越性。

下面，笔者先来谈谈 goto 语句翻译的几个重点。

(1) 向前跳转与向后跳转。通常，goto 语句的跳转分为两种情况：第一种，目标标号的位置定义点在 goto 语句之前，也称为“向前跳转”。第二种，目标标号的位置定义点在 goto 语句之后，也称为“向后跳转”。实际上，对于 Pascal 而言，两者的分析与处理并不存在差异。然而，C 语言编译器在分析 goto 语句时，就必须严格区别这两种情形。C 语言没有标号声明机制，在分析向后跳转的 goto 语句时，由于目标标号的定义点位于 goto 语句之后，使编译器无法获得目标标号的任何信息。在这种情况下，编译器又该如何处理呢？无论何种处理方案都必须保证任何 goto 语句的目标标号都是存在唯一定义点的。基于这个前提，通常的处理方案是将目标标号登记到符号表中，并设置特定的标志用于标识此标号存在引用点但不存在定义点。编译器一旦获得该标号的定义点信息，及时更新该标识的符号表信息即可。当编译器分析完输入源程序后，必须逐一检查符号表中的标号信息，判断是否存在仅有引用点却没有定义点的标号信息，如果存在，则给出相应的出错提示。当然，有些编译原理书籍也介绍了一些其他的处理方案，如回填链等。

(2) 过程内跳转与过程间跳转。这是一个比较复杂的问题，当然，函数也存在同样的问题。现代高级语言对此有比较严格的限制，即 goto 语句仅能用于实现过程内跳转，而不允许进行过程间跳转。熟悉汇编语言的读者可能会有疑惑，有些目标机的 JMP 指令是可以实现全空间跳转的，为什么 goto 语句只能进行过程内跳转呢？这是因为汇编语言程序设计中经常使用静态存储分配机制。而编译器生成的目标代码更多使用的是栈式存储分配机制。过程间跳转涉及一些复杂的存储管理方面的问题，在一般情况下，编译器是很难实现真正意义上的过程间跳转的。当然，这也并非完全不可能的。在早期，有些静态分配存储空间的编译器就可以比较方便地实现过程间跳转，而不需要付出太大的代价。

下面，再来看看 goto 语句的相关语义子程序。

程序 5-23 semantic.cpp

相关文法：

其他语句 → goto 标识符 063

```
1  bool semantic063()
2  {
3      string szTmp=TokenList.at(iListPos-1).m_szContent;
4      int i=SymbolTbl.SearchLabelInfoTbl(SymbolTbl.ProcStack.top(),szTmp);
5      if (i!=-1)
6      {
7          OpInfo Tmp;
8          Tmp.m_iType=OpInfo::LABEL;
9          Tmp.m_iLink=i;
10         SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(
11             EmitIR(OpType::JMP,Tmp));
12         SymbolTbl.LabelInfoTbl.at(i).m_bUse=true;
13         return true;
```

```

14     }
15     else
16     {
17         EmitError("标号未声明",TokenList.at(iListPos-1)); //出错提示
18         return false;
19     }
20     return true;
21 }

```

第 5 行：在标号信息表中，检索目标标号是否已声明。注意，检索范围仅限于本过程（函数）内，即使是主程序中声明的标号也不允许在子过程中被引用。Pascal 语言明确规定标号定义或引用前必须在声明部分中给出相应的说明，而 C 语言并没有对应的要求。

第 10、11 行：生成 JMP 语句，并设置目标标号。

第 12 行：设置目标标号的 m_bUse 属性，表示该标号已经存在引用点。

最后简单说明一下 Neo Pascal 如何保证变量声明、定义的有效性。实际上，有以下几种情况需要考虑，请参考表 5-11。

表 5-11 标号有效性列表

声明	定义	引用	处理方式
无	无	有	goto 语句处理时提示出错
无	有	无	定义标号时提示出错
无	有	有	定义标号、goto 语句处理时都提示出错
有	无	无	Pascal 语言允许
有	无	有	全局检查

注意，前三种情况在分析 goto 语句或定义标号时都能够识别与处理。不过，由于 goto 语句的向后跳转的情形，使得最后一种检查不能即时进行。在分析完一个输入源程序后，编译器会对标号信息表作一次全局分析，确定哪些标号仅有引用没有定义。

程序 5-24 semantic.cpp

相关文法：

程序 → 程序头 程序块 002

```

1  bool semantic002()
2  {
3      for (int i=0;i<SymbolTbl.LabelInfoTbl.size();i++)
4      {
5          if (SymbolTbl.LabelInfoTbl[i].m_szName[0]!='_' &&
6              !SymbolTbl.LabelInfoTbl[i].m_bDef &&
7              SymbolTbl.LabelInfoTbl[i].m_bUse)
8          {
9              EmitError(SymbolTbl.LabelInfoTbl[i].m_szName+"标号未定义"
10                      ,TokenList.at(iListPos-1));
11              return false;
12          }

```



```
13     }  
14     return true;  
15 }
```

这是整个语义分析阶段最后被调用的语义子程序，它的主要作用就是检查是否存在仅有引用没有定义的标号。注意，编译器分配的临时标号是不需要作检查的，只需跳过即可。

5.8.3 asm 语句的翻译

asm 语句的功能是在 Neo Pascal 程序中内嵌汇编代码。在实践中，内嵌汇编机制也被程序员广泛应用于各种场合，尤其是系统、硬件方面的开发。

内嵌汇编的优点如下：

- (1) 内嵌汇编可以完成一些高级语言无法实现的功能。
- (2) 内嵌汇编可以实现一些对空间、时间要求较高的应用。
- (3) 有时，内嵌汇编可以简化实现的复杂程度。在实践应用中，这种情况也并不少见。

下面，就来讨论内嵌汇编机制是如何实现的。在现代编译技术中，比较常见的实现方式主要有如下两种：

(1) 将汇编语言的文法完全引入高级语言文法中，由编译器统一进行词法、语法、语义等分析。实际上，这种实现方式使得编译器必须实现汇编的词法、语法、语义分析等功能，从而增加编译器实现的复杂度。不过，其优点在于一些内嵌汇编代码中的错误可以尽早被发现。当然，也有利于一些指令级的优化算法得以进行。经典的 Turbo Pascal、Delphi 都是基于这种方式实现内嵌汇编的。

(2) 将汇编代码作为字符串直接嵌入目标代码中，由汇编器统一进行汇编分析。这种方式实现比较简单，编译器不需要关注内嵌汇编的具体代码，只需将其作为普通的单词即可。不过，这种方式会使得一些指令级的优化算法失效，也许这就是易于实现的代价吧。从纠错方面而言，两种方式的纠错能力是相同的，仅有的差别在于报错的阶段不同而已。著名的 GCC 就是基于这种方式实现内嵌汇编的。同样地，Neo Pascal 也是选择了这种实现方式。

下面谈谈 Neo Pascal 的内嵌汇编机制。一般来说，在处理内嵌汇编时，编译器设计者关注的是如何实现内嵌汇编与输入源程序的信息交换。当然，符号命名规则是保证内嵌汇编与输入源程序顺利实现信息交换的主要因素。换句话说，内嵌汇编访问输入源程序中的变量时，只能是依据该符号的实际名字（即编译器分配的名字），而不是由用户指定的名字。因此，在编译器设计中，用户声明的符号的实际命名并不是完全随机的，为了便于内嵌汇编的访问，统一的命名规则是必不可少的。不过，出于安全考虑，编译器很少允许内嵌汇编直接根据命名规则访问用户变量。因为这种不受限的访问会给编译带来很大的不可控性，这并不是编译器设计者愿意见到的。笔者从 GCC 的设计中得到灵感，引入了汇编参数的设计方案。

在 Neo Pascal 中，内嵌汇编主要是由两个部分组成：汇编程序、汇编参数。顾名思义，汇编程序就是内嵌汇编的主体程序段，编译器最终会将其稍作变换后插入到目标代码中。这里，值得注意的是汇编参数。汇编参数同样是一个字符串常量，其主要作用就是指出汇编程序中访问的用户变量。同时，内嵌汇编程序也就是通过这个接口与高级语言程序进行数据交换的。例如：

【声明 5-8】

```
VAR
    i, j, k:integer;
BEGIN
    asm 'mov eax, %0
        mov %1, eax',
        'r@i, w@j'
    end;
END.
```

注意，第 6 行的字符串就是一个汇编参数列表。在本例中，一共有两个参数，即 *i* 和 *j*。其中，“@”符号前的字母用于标识参数的读写状态，“r”表示只读，“w”表示只写，“a”表示即读又写。而汇编程序则通过“%X”的方式引用参数。在本例中，“%0”即表示引用 *i*，而“%1”则表示引用 *j*。注意，由于编译器并不分析汇编程序本身的语义，因此，也不严格判定参数中读写状态与实际引用的方式是否一致。因两者不一致而造成的后果，完全是由用户承担的。注意，正确设置读写状态是非常重要的，在后续章节的优化处理中，这个状态标志是具有非常重要的意义的。

下面详细分析内嵌汇编的相关实现。

程序 5-25 semantic.cpp

相关文法:

其他语句 → asm 字符串常量,字符串常量 083 end

```
1  bool semantic083()
2  {
3      string szPara=SymbolTbl.ConstInfoTbl[atol(TokenList.at(iListPos-1)
4          .m_szContent.c_str())].m_szName;
5      string szAsm=SymbolTbl.ConstInfoTbl[atol(TokenList.at(iListPos-3)
6          .m_szContent.c_str())].m_szName;
7      string szTmp="";
8      int iAsmParaLink=SymbolTbl.AsmParaTbl.size();
9      AsmPara TmpAsmPara;
10     for(int i=1;i<szPara.length()-1;i++)
11     {
12         if (szPara[i]!='@' || szPara[i]!=';')
13         {
14             szTmp=trim(szTmp);
15             if (szTmp.empty())
16             {
17                 EmitError("内嵌汇编语句的参数信息不正确",TokenList.at(iListPos-1));
18                 return false;
19             }
20             if (szPara[i]!='@')
21             {
22                 TmpAsmPara.flag=AsmPara::N;
23             }
```



```
24         if (szTmp[0]=='w')
25             TmpAsmPara.flag=AsmPara::W;
26         if (szTmp[0]=='a')
27             TmpAsmPara.flag=AsmPara::RW;
28         if (szTmp[0]=='r')
29             TmpAsmPara.flag=AsmPara::R;
30
31         if (TmpAsmPara.flag==AsmPara::N)
32         {
33             EmitError("内嵌汇编语句的参数读写属性不正确"
34                 ,TokenList.at(iListPos-1));
35             return false;
36         }
37         szTmp="";
38     }
39     if (szPara[i]==';')
40     {
41         szTmp=UpperCase(trim(szTmp));
42         TmpAsmPara.m_iLink=SymbolTbl.SearchVarInfoTbl(SymbolTbl
43             .ProcStack.top(),szTmp);
44         if (TmpAsmPara.m_iLink==-1)
45         {
46             TmpAsmPara.m_iLink=SymbolTbl.SearchVarInfoTbl(0,szTmp);
47             if (TmpAsmPara.m_iLink==-1)
48             {
49                 EmitError("变量"+szTmp+"不存在",TokenList.at(iListPos-1));
50                 return false;
51             }
52         }
53         TmpAsmPara.m_szName=szTmp;
54         SymbolTbl.AsmParaTbl.push_back(TmpAsmPara);
55         szTmp="";
56     }
57 }
58 else
59     szTmp=szTmp+szPara[i];
60 }
61 for (int i=0;i<szAsm.length();i++)
62 {
63     if (szAsm[i]=='%')
64     {
65         i++;
66         string szNum;
67         while (szAsm[i]<='9' && szAsm[i]>='0')
68             szNum=szNum+szAsm[i++];
69         if (szNum.empty() ||
```



```
70             (iAsmParaLink+atoi(szNum.c_str())>SymbolTbl.AsmParaTbl.size()-1))
71         {
72             EmitError("内嵌汇编参数引用不正确",TokenList.at(iListPos-1));
73             return false;
74         }
75     }
76 }
77 OpInfo Op1,Op2,Op3;
78 Op1.m_iType=OpInfo::CONST;
79 Op1.m_iLink=atoi(TokenList.at(iListPos-3).m_szContent.c_str());
80 Op2.m_iType=OpInfo::NONE;
81 Op2.m_iLink=iAsmParaLink;
82 Op3.m_iType=OpInfo::NONE;
83 Op3.m_iLink=SymbolTbl.AsmParaTbl.size()-1;
84 SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(
85     EmitIR(OpType::ASM,Op1,Op2,Op3));
86 return true;
87 }
```

第 3~4 行：获取汇编参数字符串。

第 5~6 行：获取汇编程序字符串。注意，汇编程序段在前，而汇编参数字符串在后。

第 8 行：`AsmParaTbl` 是一张全局向量表，用于存储内嵌汇编的参数信息。当然，根据它的实际用途，将其理解为一个辅助的数据结构可能更合适。

第 10~60 行：分析汇编参数字符串。注意，这里完成的工作主要包括：分析读写状态标志、分析变量。实际上，这里的本质就是字符串处理。根据先前定义的汇编参数的格式，应该不难理解。下面，笔者对此作简单分析。

第 20~38 行：分析读写状态标志。

第 39~57 行：分析变量信息。这里主要有两项工作需要完成：获取变量名、变量名的有效性检查。注意，在进行有效性检查时，同样要关注两级命名空间的问题。

第 61~76 行：遍历汇编程序，判断汇编程序中引用参数是否存在越界，即引用参数的编号是否超过汇编参数列表给出的参数个数。这里的检查非常必要，否则代码生成可能会出错。值得注意的是，汇编参数列表中的参数可以不被引用，但越界引用却是不允许的。

第 77~85 行：生成 ASM 指令。从 IR 设计的角度来说，ASM 指令只有一个操作数，即汇编程序的字符串常量。不过，汇编参数也是一个非常重要的信息，虽然它可能很难直接通过 `OpInfo` 传递到后续阶段。这里，笔者借助于操作数 2、结果操作数的 `m_iLink` 属性传递汇编参数信息。其中，操作数 2 的 `m_iLink` 属性用于标识参数列表在 `AsmParaTbl` 表中的起始位置，而结果操作数的 `m_iLink` 属性则用于标识参数列表在 `AsmParaTbl` 表中的结束位置。后续阶段就可以通过这两种属性访问汇编参数信息了。注意，这里只是借助于这两个操作数而已，但它们本身并不是作为真正的操作数存在于 IR 中，因此，笔者将两个操作数的 `m_iType` 属性都设置为 `OpInfo::NONE`，以免给后续处理带来不便。

至此，笔者已经详细阐述了语句翻译的理论的实现，这是 IR 生成的核心部分之一。语句翻译的重点就是建立翻译方案与语义子程序之间的联系，因此，设计翻译方案将是极其重

要的。在第 6 章中，笔者将揭示 IR 生成的另一个重要部分——表达式翻译。其中，将涉及类型系统、数组计算、指针引用等复杂而有趣的话题。

5.9 深入学习

本章讨论的重点就是 IR，因此，有必要推荐几种经典的 IR，供读者参考学习。

Java 的字节码是近几年比较流行的一种 IR 形式，有兴趣的读者可以参考相关资料。

“鲸书”第 4 章也提出了一种不错的 IR 形式，并进行了详细的阐述。

RTL 是 GCC 的 IR 形式，GCC 内核白皮书对此有详细的解释。

MSIL 是 .Net 的一种 IR 形式，也是近几年来比较流行的一种低级语言，是渴望了解与学习 .Net 内核的读者的必备知识。

UNCOL 是一种诞生于 20 世纪 50 年代中期的通用 IR 形式，在《The problem of programming communication with changing machines: a proposed solution》一文中最先提到了 UNCOL 的概念。

lcc 的 DAG 也是一种非常经典的 IR 形式，《可变目标 C 编译器——设计与实现》一书对此有详尽说明。

SunIR 是 Sun 为 SPARCE 提供的 C、C++、Fortran、Pascal 等编译器的一种前端 IR 形式。

C-- 也是目前比较流行的一种前端 IR 形式，是一种比 C 语言更低级的形式，当然，它仍然是符合 C 语言标准的，这是 C-- 的最大优势。这种 IR 形式可以被编译成可执行文件，更有利于前端调试与验证。

1、Java intermediate bytecodes

James Gosling

说明：本文是关于 Java 字节码最权威的资料，是由 Sun 微系统实验室的 James Gosling 于 1995 撰写的。

2、GNU Compiler Collection Internals

说明：这是关于 GCC 内核最权威的资料，其中，详细描述了 RTL 的话题，读者可以访问 <http://gcc.gnu.org/onlinedocs/gccint/>。

3、Microsoft .NET IL 汇编语言程序设计

Serge Lidin

机械工业出版社

说明：这是目前可以见到的唯一一本关于 MSIL 的书，不过，由于成书较早，涉及的内容与微软最新的技术有一定差距。

4、The problem of programming communication with changing machines: a proposed solution

说明：最先提出了 IR 设计的思想与 UNCOL 的提议。

5、可变目标 C 编译器——设计与实现

Christopher W. Fraser, David R. Hanson

电子工业出版社

说明：该书所描述的 lcc 是一个值得读者深入学习的实例系统，它的很多设计思想被广泛应用于许多相关领域，如操作系统等。

6、高级编译器设计与实现

Steven S. Muchnick

机械工业出版社

说明：该书第 4 章给出了一种有效的 MIR 形式，而第 21 章中详细描述了 SunIR。

7、C-- specification (Version 2)

说明：这是 C-- 语言的参考手册，读者可以访问 <http://www.cminusminus.org/>，以获得 C-- 的更多资料。

5.10 实践与思考

1. 请结合本章提到的几种 IR 形式，试分析与评价 NPIR 的优缺点，并思考改进的方案。
2. Neo Pascal 并不显式构造语法树，如果需要显式构造与输出语法树，试问该如何修改？

3. 实际上, 将一种高级语言程序编译为另一种高级语言程序的应用并不罕见, 其中, 最著名的就是 C++, 最早的 C++ 编译器的目标代码就是 C 语言。如果需要将 Neo Pascal 的目标代码改为 C 语言, 那么, 该如何修改与完善?

5.11 大师风采——Kenneth E. Iverson

Kenneth Eugene Iverson: 加拿大计算机科学家, APL 语言创始人之一。1920 年 12 月 17 日出生于加拿大 Camrose 的一个农民家庭。少年时期, Iverson 就酷爱数学, 自学微积分。二战后, 他进入了皇后大学 (Queen's University) 主修数学与物理, 并于 1950 年获得学士学位。1951 年, Iverson 获得了哈佛大学数学硕士学位。1954 年, Iverson 又获得了哈佛大学应用数学博士学位, 毕业后留校 5 年任助理教授。

1960 年, Iverson 进入 IBM 公司从事基于 IBM 360 的程序设计语言开发。1980 年, 他离开 IBM 公司, 进入加拿大的 I. P. Sharp Associates 从事 APL 语言的设计与开发, 直到 1987 年退休。

1989 年夏天, Iverson 与 Roger Hui、Arthur Whitney 开始着手 J 语言的设计与实现, 其原型就是 APL。在以后的 15 年中, Iverson 与 Roger Hui 一直致力于 J 语言的研究。Iverson 于 2004 年去世。

Iverson 是 1979 年图灵奖获得者。

第 6 章 表达式语义

Actually I made up the term “object-oriented”, and I can tell you I did not have C++ in mind.

—Alan Curtis Kay

6.1 表达式概述

表达式是程序设计语言的一个重要组成部分，包括汇编语言在内的绝大多数语言都引入了表达式的概念。通常，表达式主要由两个部分组成，即操作数与运算符。有些读者可能会认为一门语言的运算符个数也是非常有限的，因此，仅从语义角度而言，表达式并不难理解。不过，不能忽略一个重要的元素——类型系统。当类型系统被引入表达式之后，很多看似简单的问题可能会变得异常复杂，尤其是试图借助于一个软件系统实现相关功能时。例如，Pascal 表达式 $i+4$ 中的加法运算符的精确语义是什么呢？这个问题并不好回答，根据 i 的实际类型，加法运算符至少可能存在两种不同的语义，即无符号整数加法、有符号整数加法。对于编译器设计者而言，仅仅回答这个运算符为加号是远远不够的。通常，需要了解更深刻的语义。在上例中，如果考虑语言的详细数据类型，由于各种数据类型的存储空间不同，可能会得到更多关于加法运算符的语义理解。理论上讲，两种不同类型数据之间的加法运算都存在类型转换。不过，其中只有很小一部分转换是显式的，即需要程序员编码完成的，大多数转换都是隐式的，完全是编译器自动分析完成的。不过，在编程过程中，程序员一般很少理会隐式转换机制，甚至有时根本没有意识到隐式转换的存在，因为有太多太多的工作是由编译器代劳的。程序员可能很难想象如果书写每个表达式都是需要显式完成类型转换的情形。本章将从类型入手，讨论几个与表达式分析相关的话题，引导读者从编译器设计的视角重新认识表达式。下面，简单谈谈本章将涉及的几个核心问题：

(1) 类型系统。这是表达式翻译的核心，没有设计相对精良的类型系统，就绝对无法进行表达式的翻译。笔者将从类型描述、类型等价、类型相容、类型转换等方面阐述编译器类型系统的设计，并详细介绍 Neo Pascal 的相关设计思想与实现。

(2) 数组、结构、指针等复杂变量的寻址。实际上，这个问题的本质就是讨论操作数的寻址翻译。在翻译表达式之前，编译器必须得到所需的操作数寻址的相关 IR，否则表达式的 IR 是无法得到的。简单变量的寻址是比较容易的，而数组、结构、指针等复杂变量的寻址就比较困难了。当然，更为复杂的就是这些元素的嵌套使用，如何正确无误地生成 IR 是编译器设计者必须关注的。

(3) 常量的计算。实际上，这是一个优化方面的技术（即常量折叠），但是并不意味着语义分析阶段就不必理会了。常量折叠的思想是很简单的，它试图将操作数皆为常量的表达式直接由编译器计算得到相应的结果，以备后用。例如，表达式 $i=5+3$ ，由于加法运算的两个操作数都是常量，所以 $5+3$ 的计算就可以由编译器完成，并不需要生成相应的加法 IR，

而是直接翻译为 $i=8$ 相应的 IR 语句即可。当然，读者同样不要小觑常量折叠。由于类型系统的存在，要完成真正意义上的常量折叠是有些难度。

(4) 表达式翻译。最后，就是如何正确理解与翻译表达式了。读者可能立刻会联想到繁复的运算符优先级，难道这是表达式翻译的核心吗？答案是否定的。虽然，每位 C 程序员曾几何时必定都被 C 语言的 15 级运算符优先级折磨得苦不堪言，但是，运算符优先级却不是表达式翻译所关注的。甚至读者会发现表达式翻译时对其根本不需要关注，这是非常神奇的。当然，在使用递归下降分析法时，有时会利用运算符优先级来简化程序设计，那是一种非常精巧的设计结构。有兴趣的读者可以参考 lcc 的实现。这里就不再深入讨论了。

6.2 类型系统基础

6.2.1 类型基础

有程序设计经验的读者应该对类型并不陌生，类型已经成为高级语言的基本组成元素之一。即便如此，学界关于类型的定义却是存在一定分歧的，笔者在此只能给出一个比较普遍的观点。在大多数程序设计语言中，每一个数据值通常都关联一系列的性质，而这样的一系列性质就是这个值的类型。类型描述的就是基于该类型的数据值所共有的一组性质。例如，integer 类型的数据必定在 $[-2^{31}, 2^{31}]$ 范围内取值，而 char 类型的数据必定在 $[0, 255]$ 内取值。而程序设计语言中的类型可以分为两类，即基本类型和复杂类型。基本类型指的就是程序设计语言预定义的、不可再分割的类型，如 integer、char、real 等，有时也称为“原子类型”。而复杂类型通常指的是由程序员基于基本类型复合构造而成的类型，如记录类型、数组类型等，有时也称为“复合类型”或“构造类型”。

通俗地讲，程序设计语言中的类型系统指的就是语言的类型集合及其描述程序行为的规则。从程序设计语言的观点而言，引入类型系统的目的就是便于在语义层次上更精准地描述程序的动作行为。从编译器的观点而言，引入类型系统的主要目的就是尽可能保证程序运行的安全。

在程序设计语言中，不同运算符对于其运算对象的类型是有一定要求的。当然，这个标准并不统一，有些语言可能比较宽松，而有些语言就比较严格。对于不满足要求的表达式，通常应该报告出错或警告信息，否则是不安全的。通常，在这种情况下，很难预测将会发生什么样的情况，甚至可能威胁到操作系统的稳定性。在编译器设计中，严格的类型检查是必不可少的。实际上，所谓类型检查就是基于类型系统进行的一个处理过程，其目的就是保证每个操作或运算都是针对一组数目正确、类型合适的元素进行的，从而保证了目标程序安全、有效地执行。类型检查是编译器的一个重要组成部分。

根据检查的时刻不同，类型检查一般分为两类：静态类型检查、动态类型检查。

静态类型检查指的是由编译器在编译过程中完成的类型检查。

动态类型检查指的是在目标代码中插入相应的类型检查子程序，以便在目标程序运行时进行各种类型检查。

从理论上来说，在不考虑效率及资源的情况下，只要目标代码中存在足够的类型信息，动态类型检查可以实现一切类型检查，而静态类型检查则不然。例如，通用的数组越界检



查、子界类型赋值合法性等问题是无法实现静态检查的。不过，程序设计语言的绝大多数类型相关错误都可以由静态检查发现。

最后，笔者需要说明一点：本章仅讨论了类型系统中最基础的部分。事实上，程序设计语言的类型系统是一个非常深奥的话题，包括许多复杂的理论。目前，国内类型系统领域的研究还处于基础阶段，对于有志于研究程序设计语言的读者而言，类型系统理论的重要性不言而喻。

6.2.2 类型系统

现代程序设计语言的类型系统通常包括如下四个部分：基本类型、新类型构造规则、类型等价或相容规则、类型推断规则。很多语言还提供了一些类型隐式转换的规则。下面，从编译器设计角度来详细描述各部分的作用。

1. 基本类型

绝大多数语言的基本类型都是从三个原子类型（即数值、字符、布尔值）引申得到的，其中数值是最为复杂的，它可以细化成各种基本类型。例如，数值可以分为整数类型、实数类型。而根据不同的程序设计语言，又将整数类型定义为 `integer`、`byte`、`word`、`longword` 等整数类型。而实数类型又可以细化为 `float`、`double` 等。一般来说，现代大多数程序设计语言都提供了非常丰富的基本类型，以便程序员有更多的选择。

程序设计语言通常从两个方面细化整数类型，即长度、符号位。符号位相对比较简单，而长度就复杂些了。有些语言使用绝对长度描述基本类型，即在语言的类型系统中明确规定各种基本类型占用的 `bit` 或 `byte` 的数量，如 `Fortran`、`Ada` 等。而有些则是以相对长度描述基本类型，即只描述各种基本类型之间的相对关系，具体的实现是由编译器设计者规定的。例如，C 语言规定 `long long` 类型的长度是 `long` 类型的 2 倍，但并没有严格定义 `long` 或者 `long long` 的实际长度。

当然，针对实数类型而言，实数的表示形式也是一个可细化的方面。例如，实数的尾数的精确位数、阶码的表示范围等。大多数语言的实数类型的表示形式都是遵守 IEEE 754 标准的，所以其表示形式还是比较一致的。

以上从语言及编译器设计的角度对基本类型作了简单介绍。由于基本类型比较简单，笔者就不再耗费篇幅了。

2. 新类型构造规则

程序设计语言的基本类型通常是硬件支持的数据存储方式的一种抽象，但是这却不足以满足程序员处理复杂数据结构的需要，例如，图、树、表、栈、数组、串等。这些数据结构与基本类型的主要区别在于它们通常需要一组对象加以描述，如何将基本类型组合或复合为这些复杂的数据结构是高级语言设计者必须思考的问题。

在程序设计语言中，有些复杂类型需要语言本身的支持，如数组、串、枚举、结构、指针等。而复杂类型可以由前者合成，如树、图等。语言设计者通常将前者作为类型系统的一部分，明确说明其构造原则。下面，笔者针对语言设计中最常见的数组作简单分析，以便读者了解如何在类型系统中构建复杂类型。

通常，在类型系统中，数组类型需明确说明如下几点：

(1) 数组的基类型。有些语言规定数组的基类型只能是语言支持的基本类型，而更多的

语言则允许基类型是任意类型，甚至是数组。

(2) 行（列）优先原则。例如，Fortran 是列优先，而 C、Pascal 都是行优先。

(3) 数组操作的支持。有些语言支持对整个或部分数组赋值，例如，当 a 、 b 、 c 是大小、形状相同且基类型支持加法运算的数组时，Fortran 90 允许使用 $a=b+c$ 将 b 、 c 各单元元素相加后送到 a 的相应单元中。而 C、Pascal 对于数组的操作则相对比较简单。

(4) 最大维度。数组的最大维度理论上是没有限制的。大多数语言设计者也认同这一观点。

(5) 物理存储形式。即如何分配数组的物理存储空间，以及以什么形式存储数组元素。

(6) 数组声明的语法、语义形式。

不难发现，上述说明并不复杂，都是一些最基本的元素。对了解程序设计的读者来说，应该是非常简单的。因此，不必花较大篇幅评说各种复杂类型。注意，类型系统是程序设计语言的一部分，类型系统的描述可以是形式化的，也可以是非形式化的。从编译器设计的角度来说，不必拘泥于小节，只要能明确、清晰说明语言的类型即可。

3. 类型等价或相容规则

实际上，类型等价和类型相容是不同的概念，所谓“类型等价”就是指对于程序设计语言而言，两个类型是相同的。而所谓“类型相容”就是指对于程序设计语言而言，两个类型是兼容的，即可以通过一定的隐式转换使两个类型等价。类型等价是类型相容的充分条件。类型相容的规则更多应用于确定特定类型的对象是否满足特定上下文的需要。下面，先来谈谈类型等价的话题。

程序设计语言判断两个类型是否等价的标准是什么呢？不妨先分析图 6-1 的实例。

```
struct Node                struct TableNode
{
    int data;
    struct Node * link;
}

{
    int data;
    struct TableNode * link;
}
```

图 6-1 类型等价实例

在图 6-1 中，Node 类型与 TableNode 类型是否等价呢？这就引出了两个等价标准，即名字等价和结构等价。

名字等价的观点是：两个类型等价的基本条件是类型名字必须相同。根据这一条件，图 6-1 的类型是不等价的。严格意义上的名字等价是不可取的，因为当项目规模较大时，名字的一致性是很难维护的。通常，实际编译器很少采用严格的名字等价。不过，有些语言支持类型命名机制，那么，由此可能产生类型别名。如果将类型别名考虑在内，名字等价还是具有实际意义的，这种机制有时称为“宽松的名字等价”。例如，在 C 语言中声明 `typedef struct p q;`（其中 p 是一个已声明的结构类型），即表示 q 是类型 p 的别名。在这种情况下，按严格的名字等价机制来说， q 与 p 显然不是等价类型。不过，由于 q 只是 p 的一个别名，按宽松的名字等价机制来说， q 与 p 是等价类型。注意，实际编译器中所讨论的名字等价都是宽松的名字等价。名字等价的优点在于实现比较简单。

结构等价的观点是：两个类型等价的基本条件是它们的结构必须相同。根据这个条件，



图 6-1 的类型是等价的。结构等价的判断标准要比名字等价宽松得多，也更灵活。不过，很少有程序设计语言选择结构等价作为判断标准。结构等价表面上看似乎比名字等价更合理，实则不然。主要有如下三个原因：

(1) 无法区别设计本意不同但结构完全相同的两个类型。如图 6-2 所示。

```

struct rectangle    //长方形
{
    float x;        //长
    float y;        //宽
}

struct ellipse      //椭圆形
{
    float x;        //长轴
    float y;        //短轴
}

```

图 6-2 结构等价实例

不难发现，根据结构等价的观点，图 6-2 所示的两个类型是等价的。不过，这个判断结果似乎明显违背了程序员的本意。这种歧义等价结果不但没有太大的意义，甚至还有可能引发一些问题。

(2) 结构等价的实现也比较复杂，而且根据源语言的特点，可能还需要考虑递归类型声明的情况。尤其当基本类型比较丰富时，判断算法的效率可能比较低。

(3) 由于不同程序设计语言的特点不尽相同，结构等价的定义也就各有不同了。例如，有些语言并不强调结构类型中域的顺序，这样判断算法就会非常复杂。

下面，举两个实际语言的例子来说明类型等价问题。

C 语言明确指出其使用宽松的名字等价规则，结构、联合以及 typedef 皆是如此，而 C 语言的基本类型都是互不等价的。只不过由于 C 语言是弱类型语言，有较丰富的类型相容规则，读者千万不要认为 C 语言类型就是互相等价的。

Pascal 的类型等价问题就比较模糊了，Pascal 报告中并没有明确提到“类型等价”的概念。一般来说，Pascal 语言的类型等价问题更多依赖于具体编译器的实现。这里，笔者以 Turbo Pascal 为例，说明 Pascal 的类型等价问题。

例 6-1 Turbo Pascal 的类型等价。

```

TYPE
    T1=^INTEGER;
    T2=^INTEGER;
    T3=T1;
VAR
    a,b    :T1;
    c      :T2;
    d      :^INTEGER;
    e      :^INTEGER;
    f      :T3;
.....

```

在本例中，a, b, c, d, e 中有几组类型等价的变量呢？答案只有一组，即 (a, b, f)。为什么呢？从理论上分析，可能并不容易理解其中的原因。实际上，从符号表设计的角度就非常容易得到这个答案。根据前面所介绍的符号表结构,可以得到如图 6-3 所示的符号表结构。

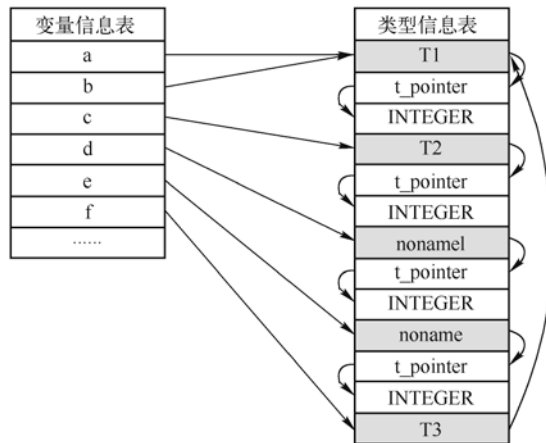


图 6-3 符号表示意图

从符号表中，不难得到一个结论：Turbo Pascal 判定类型等价的标准是从变量的类型指针开始，跃过所有类型别名结点（例如，本例中的 T1、T2、T3、noname1、noname2）后，如果指针指向同一个类型结点时，则认为两个变量的类型是等价的，否则认为它们是不等价的。例如，判断 f 与 a 变量的类型等价的过程如下：分别跃过别名结点 T3、T1，最终都指向了第二行的 t_pointer 结点，由此推断它们是等价的。这种观点得到许多 Pascal 编译器设计者的认同。Neo Pascal 的类型等价规则与 Turbo Pascal 是一致的。

讨论完类型等价问题，再来看看类型相容的话题。

前面已经简单介绍了类型相容的概念。从实践的角度来说，类型相容更多关注的是类型之间的隐式转换是否可行。例如，在 Pascal 中，讨论 INTEGER 与 REAL 类型是否相容的问题，就是明确将哪个类型作为标准类型。如果将 REAL 作为标准类型，那么，将 INTEGER 类型隐式转换为 REAL 是完全可行的，在这种情况下两者是相容的。但是，如果将 INTEGER 作为标准类型，而将 REAL 类型隐式转换为 INTEGER 则是不可行的，这种情况下两者就不相容了，而只能借助于显式转换。

通常，程序设计语言讨论类型相容是基于一个特定的上下文环境讨论的。只有在一定的上下文环境下，才有标准类型（或者称为上下文期望的类型）的概念。一般而言，需要在以下几种上下文环境下考虑类型相容：

赋值语句：右部表达式类型必须与左部表达式的类型相容。此时，左部表达式的类型作为标准类型。

运算表达式：根据不同的运算表达式的需求，操作数的类型必须与运算表达式期望的类型相容。

函数传参：实参的类型必须与形参类型相容，此时，形参类型作为标准类型。

函数返回值：实际返回值表达式的类型必须与函数声明的返回值类型相容，此时，声明类型作为标准类型。

关于类型相容性的定义，不同的程序设计语言有各自的理解。有些语言比较严格，而有些语言就比较宽松。C 语言就是一种非常宽松的语言实例，在 C 语言中，所有的数值类



型都是相容的。即使用 double 变量给 char 变量赋值，依然是合法的，其中的隐式转换完全由编译器完成。然而，Pascal 语言就严格一些。关于类型相容的宽严之争，学界并没有统一的观点。

4. 类型推断规则

在 C 语言中，已知 a、b 的类型是 unsigned char，是否可以判断 a+b 的类型呢？读者可能会不假思索地认为结果类型也是 unsigned char。这个答案正确吗？假设 a、b 的值都是 255 的情况下，a+b 的值为 510，显然已经远远超过了 unsigned char 类型的表示范围。有读者可能会觉得运算结果溢出应该由程序员控制，而不应当由编译器处理。这种说法并非没有道理。确实，在实际编程过程中，有一些数据溢出的错误是应该由程序员自行解决的。如果程序员将 a+b 的结果赋给了一个 char 类型的变量，因此所产生的溢出完全应由程序员负责。但是，如果程序员将 a+b 的结果赋给了一个 int 类型的变量，计算结果却由于 a+b 运算结果的溢出而不正确，难道这也要程序员买单吗？显然，这是令人无法接受的。那么，如何才避免此类溢出的发生？通常，需要编译器根据每一个表达式的操作数类型，推断获得其运算结果的类型，这种推断机制就是类型推断。

那么，类型推断又是如何达到预期目标的呢？实际上，在设计程序设计语言时，设计者根据类型系统及运算符的实际状况，推导出类型与运算符各种组合之下的结果类型。然后，将这些推导结果整理形成一种映射关系，这就是类型推断规则。编译器就是根据这些规则机械地完成类型推断工作的。

下面，先看一个简单的类型推断规则的实例。这是部分 Neo Pascal 类型推断规则。从表 6-1 分析，不难想象，完整的类型推断规则的规模应该是非常庞大的。一般而言，可以得到如下结论：

$$R \geq Op \times T \times T$$

其中，R 表示类型推断的规则数量，Op 表示双目运算符的种类数，T 表示语言预定义类型种类数。Neo Pascal 的规则数量大约为 1300 条左右。然而，有些语言引入了类型提升的概念，从编译器设计的角度而言，它可以将类型推断描述为一个规则函数的形式。

表 6-1 类型推断规则

a 类型	b 类型	a+b 类型
T_INTEGER	T_INTEGER	T_INTEGER
T_INTEGER	T_BYTE	T_INTEGER
T_INTEGER	T_SHORTINT	T_INTEGER
T_INTEGER	T_SMALLINT	T_INTEGER
T_INTEGER	T_WORD	T_INTEGER
T_INTEGER	T_LONGWORD	T_INTEGER
T_INTEGER	T_CARDINAL	T_INTEGER
T_INTEGER	T_REAL	T_REAL
T_INTEGER	T_SINGLE	T_REAL
T_INTEGER	T_POINTER	T_POINTER

以上只讨论了类型推断的一个话题，即表达式类型推断。当然，这是类型推断的主要工

作。实际上，除此之外，类型推断的工作还包括推断常量的类型。在编程过程中，程序员不可避免地会使用常量。但是，常量到底是什么类型的呢？程序设计语言无法要求程序员像声明变量一样，说明常量的类型。有读者可能会反驳，在 C 语言中，声明常量通常也需要指出类型的。实际上，这种形式的常量是符号常量，的确有些语言要求程序员为符号常量指明类型，但这并不是普遍的，Pascal 语言的符号常量是不需要类型信息的。然而，读者可能忽略了一种更常用的常量，例如，`cos(3.4)`，其中，3.4 就是一个常量。针对这种情况，相信没有一种语言要求程序员指明其类型信息的。当然，3.4 也不能没有类型信息，而它的类型必须由编译器推断得到。较之表达式的推断，常量类型的推断要简单得多，只需根据常量的取值选择合适的类型即可。当然，常量类型推断也有一定技巧，例如，对于正整数而言，将其推断为无符号类型还是有符号类型是存在一定区别的。不同程序设计语言及编译器对于这些细节的定义与实现是不尽相同的。在常量推断方面，有些语言的处理非常简单，即尽可能选择高级类型描述常量，而并不要求精确推断。

最后，笔者还必须指出一点，对于 C、Pascal 之类的结构化语言，类型推断机制是很简单的。有些语言特性却可能会使类型推断变得极其复杂，例如，BASIC 语言不强制要求变量声明，编译器只能从上下文中收集线索并作出推断。再如，面向对象语言的派生类、多态性、重载、泛型、作用域规则等特性同样会使得类型推断变得复杂，C++ 的类型推断机制正是如此。而函数式语言 ML 在类型推断方面是研究最深刻且做得最好的语言之一。

6.2.3 类型转换

相信读者对于类型转换应该并不陌生，在 C 语言中，将类型转换分为隐式转换、显式转换两类。C 语言是弱类型语言，对于类型转换的限制很少，当然，其中也包括了一些不安全的转换。这里，笔者不谈显式转换，因为显式转换的安全性是由程序员保证的。对于编译器来说，显式转换实际上就是一次函数调用的过程。而隐式转换是由编译器自动完成的类型转换，所以有时也称为强制类型转换。隐式转换并不需要程序员授权，而转换的安全性也是由编译器保证的。一般而言，程序设计语言对隐式转换的基本要求就是保证安全，对于一些存在一定不安全因素的隐式转换宁可不做。Pascal 语言在隐式转换方面是非常谨慎的，例如，`real` 转 `integer`、`integer` 转 `char` 等转换都不作为隐式转换的范畴，需要程序员编码完成。

在实现隐式类型转换时，语言及编译器设计者应当注意如下几点：

(1) 损失精度处理。例如，C 语言中，`float` 转 `int` 时，小数部分应该如何处理呢？选择舍尾法取整还是四舍五入取整是语言设计者应该考虑的问题。

(2) 有符号类型与无符号类型互转。在这种情况下，必须考虑如何处理符号位。通常的做法是忽略符号位的存在。例如，无符号字符类型变量 `a` 转换成有符号字符类型变量时，直接将其按字节复制即可。当 `a` 的值为 200 时，转换结果为 -55。

(3) 有符号扩展的处理。例如，将 `signed char` 转为 `int` 时，一定要注意按符号位扩展。

最后，笔者有一句忠告，语言中广泛应用隐式转换并非上上之策。实际上，C 语言的隐式转换在学界是受到质疑的，它可能会使一些本来非常明显的错误被隐藏。对于程序员而言，这是非常可怕的，因为无法预知“定时炸弹”何时引爆。正确评价与利用隐式转换机制是有现实意义的，安全性是需要考虑的最重要因素。

6.3 类型系统的实现

6.3.1 类型系统的设计

前面详细讨论了类型系统的一些理论基础，包括类型等价、类型相容、类型推断以及隐式转换等话题，不少读者可能已经看得一头雾水。类型系统理论确实有些抽象，不太容易理解。不过，幸运的是编译器设计所需要的类型理论相对有限。同时，借助于 Neo Pascal 的源代码及相关设计文档，相信并不算太抽象。下面，就从一个实际编译器的类型系统实例入手，分析类型系统设计与实现的某些细节。

从编译器设计的角度而言，类型系统又是什么呢？实际上，简言之，主要就是实现类型检查和类型转换的机制。那么，如何才能真正实现这两个类型机制呢？先来看一段程序伪代码，如图 6-4 所示。

```
.....
if (Op==MOD)
{
    if (GetType(Op1)==INT && GetType(Op2)==INT)
        .....
    if (GetType(Op1)==INT && GetType(Op2)==CHAR)
        .....
    if (GetType(Op1)==FLOAT && GetType(Op2)==CHAR)
    {
        printf("取模运算不允许操作数类型为 float");
        exit;
    }
    .....
}
.....
```

图 6-4 类型系统的伪代码实现

这是一段非常简单的类型检查程序，其功能是检查取模运算的操作数类型。有些读者认为，可以依样画葫芦实现所有的类型检查，而且，一些早期经典编译器的成功也足以证明这种方案是完全可行的，但是，是否考虑过这项工程的复杂性？以 Neo Pascal 为例，可以估算一下手工实现类型检查机制需要的 if 判断数量。Neo Pascal 有 18 个双目运算符、4 个单目运算符以及 16 种类型，在不考虑单目运算符的情况下，至少存在 $16 \times 16 \times 18 = 4608$ 种组合。当然，在实现过程中，并不需要考虑这么多种组合情况。因为，设计者更多关心的是合法情况时所需执行的语义动作，比如，类型转换、类型推断等。对于非法的组合，只需报错即可。在这种情况下，只需建立一个集合，该集合描述的是各种运算符与类型的正确组合。在类型检查时，编译器按输入表达式的实际运算符及类型与集合中的元素匹配，如果存在则表示类型检查合法，直接执行相应的语义动作。如果不存在于集合中，则表示类型检查是非法

的，由编译器统一报告出错信息。即使如此，根据 Neo Pascal 的实际情况，依然不得不面临 1300 余种组合需要编译器处理。读者能否想象在程序中出现 1300 多个 if 是什么样的情况？随着源语言基本类型的扩展，程序的规模还会呈非线性增长，这是比较可怕的。同时，这种程序的可读性、可维护性必定大大降低。

不过，笔者需要指出一点，类型系统是编译器前端的一个核心元素，这 1300 余个类型判断是必不可少的，关键在于如何有效地实现。如果试图以任何理由回避类型系统的实现可能是徒劳的。

下面，介绍一种应用表格实现类型系统的方法。这种方法可统筹地考虑类型检查、类型转换以及类型推断等，甚至类型优化也得到了兼顾。实际上，编译器类型系统的工作流程是机械的、模式化的。无论是类型检查、类型转换还是类型推断都是由相应规则指导的，并不需要编译器作太多的决策或加工。以取模运算为例，可以将其总结为表 6-2 所示。

表 6-2 取模运算类型系统实例

左操作数	右操作数	结果操作数	左操作数类型转换	右操作数类型转换
char	char	char	/	/
int	int	int	/	CharToInt
word	int	word	WordtoInt	/
⋮				

借助表 6-2（表中并没有完整地列出取模运算的所有类型规则），编译器可以非常方便地实现类型检查、类型转换以及类型推断。首先，编译器根据取模表达式的左、右操作数类型检索表 6-2，检索成功，则表示输入表达式的操作数类型合法，即实现了类型检查机制。然后，编译器根据当前表项的信息，判断左、右操作数是否需要类型转换，需要则生成相应的 IR。最后，编译器根据结果操作数类型，生成相应的运算 IR。而结果操作数的类型就是根据左、右操作数类型的不同组合，加以推断得到的。至此，类型检查、类型转换、类型推断都已经非常直观地显示在表 6-2 中。下面，笔者就类型相容的实现作简单讨论。实际上，类型相容是非常简单的。读者不妨仔细考虑一下，如何应用类型系统描述类型相容呢？一般而言，A 类型与 B 类型相容就是指 A 类型可以通过类型转换后得到 B 类型。这种情况恰好与赋值运算类型检查类似，赋值运算的类型检查主要是核查源操作数类型是否与结果操作数类型相容。例如，在 Pascal 语言中，不允许将 REAL 类型的变量直接赋给 INTEGER 类型的变量，原因就是 REAL 与 INTEGER 是不相容的。因此，完全可以将类型相容问题转化为赋值运算类型检查问题来处理。至此，表 6-2 已经足以实现类型检查、类型转换、类型推断及类型相容等机制了。这里，笔者暂时将类似于表 6-2 的表格形式称为“类型系统表”。类型系统表的优点就在于编译器设计者只需将类型规则加入表中，并编码实现简单的表检索及处理动作，即可完成繁复的类型系统。

当然，也不可否认一个事实，那就是类型系统表的规模可能比较庞大，尤其是实际编译器项目中，视类型系统复杂程度不同，类型系统表的记录量可能达到数千甚至上万。

最后，再来看看 Neo Pascal 类型系统表的实现。实际上，先前讨论的表 6-2 只是一个类型系统表的简单模型，它与真正实现还是存在一定差异的。下面，笔者给出 Neo Pascal 类型系统表结构，如图 6-5 所示。



操作符	操作类型1	操作类型2	结果类型	变量操作码	常量操作码	中间码	类型转换1	类型转换2
=	T_CHAR	T_CHAR	T_BOOLEAN	#0001	#0001	EQU_1	None	None
=	T_CHAR	T_STRING	T_BOOLEAN	#0002	#0001	EQU_S	ChrToStr	None
=	T_BOOLEAN	T_BOOLEAN	T_BOOLEAN	#0001	#0001	EQU_1	None	None
=	T_INTEGER	T_INTEGER	T_BOOLEAN	#0001	#0002	EQU_4	None	None
=	T_INTEGER	T_BYTE	T_BOOLEAN	#0003	#0002	EQU_4	ByteToInt	None
=	T_INTEGER	T_SHORTINT	T_BOOLEAN	#0003	#0002	EQU_4	ShortToInt	None
=	T_INTEGER	T_SMALLINT	T_BOOLEAN	#0003	#0002	EQU_4	SmallToInt	None
=	T_INTEGER	T_WORD	T_BOOLEAN	#0003	#0002	EQU_4	WordToInt	None
=	T_INTEGER	T_LONGWORD	T_BOOLEAN	#0005	#0002	EQU_8	IntToLong8	LongToLong8
=	T_INTEGER	T_CARDINAL	T_BOOLEAN	#0005	#0002	EQU_8	IntToLong8	LongToLong8
=	T_INTEGER	T_REAL	T_BOOLEAN	#0002	#0002	EQU_8F	IntToReal	None
=	T_INTEGER	T_SINGLE	T_BOOLEAN	#0002	#0002	EQU_4F	IntToSingle	None
=	T_BYTE	T_INTEGER	T_BOOLEAN	#0002	#0002	EQU_4	ByteToInt	None
=	T_BYTE	T_BYTE	T_BOOLEAN	#0001	#0002	EQU_1	None	None
=	T_BYTE	T_SHORTINT	T_BOOLEAN	#0005	#0002	EQU_2	ByteToSmall	ShortToSmall
=	T_BYTE	T_SMALLINT	T_BOOLEAN	#0002	#0002	EQU_2	ByteToSmall	None
=	T_BYTE	T_WORD	T_BOOLEAN	#0002	#0002	EQU_2	ByteToWord	None

图 6-5 Neo Pascal 类型系统表结构示例

图 6-5 是 Neo Pascal 类型系统表的部分截图，先来看看其中的某些字段属性的含义。

变量操作码、常量操作码字段表示的是语义处理的动作，即语义处理程序根据操作码值的不同，完成相应的语义动作。这里，读者必须注意一点，为了提高目标程序的效率，当表达式的操作数都是常量时，编译器会直接计算结果并将其记录，而不会按照常规方式生成 IR。那么，就必须分两种情况考虑，即不存在变量操作数（都是常量操作数）、存在变量操作数。这里，常量操作码的意义就是标识前者的语义动作，而变量操作码就是后者的语义动作标识。

中间码字段表示的是该运算的三地址代码的操作符，编译器根据该字段生成 IR。

类型转换 1、类型转换 2 表示的是操作数 1、操作数 2 的类型转换信息，编译器根据这两个字段生成类型转换的 IR。

Neo Pascal 类型系统表的结构声明如下：

【声明 6-1】

```

struct TypeSysInfo
{
    int m_iOp;           //操作符
    int m_iOp1Type;     //操作数 1 的类型编号
    int m_iOp2Type;     //操作数 2 的类型编号
    int m_iRsltType;    //结果操作数的类型编号
    int m_iProcessId;   //常量操作码
    int m_iVarProcessId; //变量操作码
    int m_iIR;          //IR 操作符
    int m_Convert1;    //操作数 1 的类型转换
    int m_Convert2;    //操作数 2 的类型转换
};

```

最后，笔者还需要指出一点，在 Neo Pascal 中，由于类型系统将以文件形式存储在外存中，所以实际使用的类型系统表是图 6-5 所示的代码化形式，这是由源码附带的工具自动完成的，对于编译器设计是透明的。

6.3.2 IR 的操作数

在正式讲解类型系统的代码实现之前，必须重新认识一下 NPIR 的操作数，它是讨论表达式分析的重要前提。在第 5 章中，已经对 OpInfo 作过简单讲解。不过，这并不能满足表达式翻译的需要。由于 Neo Pascal 没有设计 AST 结构，所以 IR 的操作数将承担更多工作，它必须足以记录整个表达式分析过程中的类型跟踪信息。不过，这些类型跟踪信息仅仅在表达式翻译过程中有效，一旦完成了表达式的翻译之后，对于语句的翻译或者后续的优化、代码生成等是没有任何意义的。下面，就来看看 OpInfo 的完整声明形式：

【声明 6-2】

```
struct OpInfo
{
    enum {CONST,VAR,PTR,LABEL,NONE,PROC} m_iType;           //操作数类型
    int m_iLink;                                           //操作数指针
    stack<VarType> m_iDetailType;                          //操作数详细类型指针
    bool m_bRef;                                           //操作数是否为间接寻址
    vector<int> m_udChain;                                  //操作数的ud链
    vector<int> m_duChain;                                  //操作数的du链
    bool m_bUninit;                                        //操作数是否初始化
};
```

这里，主要讨论一下 m_iDetailType。这是一个用于跟踪类型变化的栈，其声明形式如下：

【声明 6-3】

```
struct VarType
{
    StoreType m_StoreType;
    int m_iLink;
};
stack<VarType> m_iDetailType;
```

VarType 结构有两个属性，分别表示基本类型及指向类型描述的指针（即类型描述信息在类型信息表中的位序号），而栈顶元素记录的则是操作数的当前类型。假设有声明形式如下：

【声明 6-4】

```
TYPE
    T3=record i,j,k:integer;end;
    T2=^T3;
    T1=array [1..3] of T2;
    T0=RECORD p: ^T1;end;
VAR
```



a:array [1..10] of T0;

表达式的某一操作数为 $a[i].p^3.k$ ，这个形式比较复杂，而隐藏在这个操作数背后的类型信息就更复杂了。根据标准 Pascal 的约定，这个操作数的最终类型是一个 integer 型数据。不过，在开始分析时，编译器是不可能准确断言的。其类型变化的过程如图 6-6 所示。

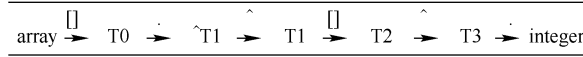


图 6-6 类型推断的过程示意

精准记录类型变化的详细过程对于编译器分析操作数的类型是非常有利的。

6.3.3 类型相容的实现

前面已经讲述了类型相容的相关基础理论，本小节将分析 Neo Pascal 的相关源代码实现。在 Neo Pascal 表达式处理中，类型等价被视作一种特殊的类型相容问题，所以不作专门讨论。下面，就来看看 Neo Pascal 是如何验证类型相容的。

程序 6-1 Type.cpp

```

1  int CType::TypeCompatible( OpInfo Op1,OpInfo Op2,int Op)
2  {
3      StoreType Op1Type,Op2Type;
4      Op1Type=GetOpType(Op1);
5      Op2Type=GetOpType(Op2);
6      return SymbolTbl.SearchTypeSysTbl(Op,(int)Op1Type,(int)Op2Type);
7  }
  
```

第 4~5 行：调用了 GetOpType 函数获取操作数的实际类型。有两种方式可以获取操作数类型：1、m_iDetailType 栈的栈顶元素。2、根据 m_iLink 属性获得变量信息表表项，再由变量信息表表项的 m_iTypeLink 获取变量的类型信息。无论使用哪种方式，都必须解决一个问题，那就是用户自定义类型的处理。Pascal 的用户自定义类型与 C 语言的类型别名是一脉相承的。例如：

【声明 6-5】

```

TYPE
    T1=integer;
    T2=T1;
    T3=T2;
VAR
    i:integer;
    j:T3;
  
```

在 C 或 Pascal 中，通常认为 i 与 j 的类型是相容的。虽然，j 存在多重的类型别名定义，但是，这丝毫不应该影响编译器的判断。在大多数语言中，类型相容都是基于实际类型讨论的，即忽略了那些类型别名信息后的实质类型。在 Pascal 语言中，与类型等价相比，类型相容的条件要宽松得多。稍后将给出 GetOpType 函数的具体实现。

第 6 行：调用 `SearchTypeSysTbl` 函数检索类型系统表 `TypeSysTbl`。前面，笔者已经介绍了类型系统表的作用。类型相容是基于某一特定的运算符讨论的，否则一切是徒劳的。这里的 `Op` 就是运算符单词的 ID，是由词法分析器生成的。

下面，再来看看 `GetOpType` 函数的实现。

程序 6-2 Type.cpp

```
1 StoreType CType::GetOpType(OpInfo Op)
2 {
3     if(Op.m_iType==OpInfo::CONST)
4         return SymbolTbl.ConstInfoTbl.at(Op.m_iLink).m_StoreType;
5     if(Op.m_iType==OpInfo::NONE)
6         return StoreType::T_NONE;
7     if(Op.m_iType==OpInfo::PTR)
8         return StoreType::T_POINTER;
9     if(Op.m_iType==OpInfo::VAR)
10    {
11        int i;
12        if(!Op.m_iDetailType.empty())
13            {
14                i=Op.m_iDetailType.top().m_iLink;
15                while (SymbolTbl.TypeInfoTbl.at(i).m_eDataType==StoreType::T_USER)
16                    i=SymbolTbl.TypeInfoTbl.at(i).m_iLink;
17                return SymbolTbl.TypeInfoTbl.at(i).m_eDataType;
18            }
19        i=SymbolTbl.VarInfoTbl.at(Op.m_iLink).m_iTypeLink;
20        while (SymbolTbl.TypeInfoTbl.at(i).m_eDataType==StoreType::T_USER)
21            i=SymbolTbl.TypeInfoTbl.at(i).m_iLink;
22        return SymbolTbl.TypeInfoTbl.at(i).m_eDataType;
23    }
24 }
```

第 3~4 行：操作数是常量，则返回常量的类型。

第 5~6 行：操作数为空，则返回 `T_NONE`。

第 7~8 行：操作数是指针，则返回 `T_POINTER`。

第 9~23 行：操作数是变量，则返回其实际类型。读者可能注意到了，Neo Pascal 分别从两个来源获取类型信息。`m_iDetailType` 栈仅在表达式翻译过程中有效。实际上，确实存在一些变量根本不需要进行表达式翻译。在这种情况下，它们的 `m_iDetailType` 自然是空的，但不能因此而不允许进行类型相容的验证，例如，for 的循环变量。为了安全起见，所以同时考虑了两个类型来源。当然，优先考虑 `m_iDetailType` 栈顶元素。仅在某些特殊情况下，当 `m_iDetailType` 栈空时，再从变量信息表取相应的类型信息。

6.3.4 类型推断的实现

这里，主要讨论常量的类型推断机制，或者说是常量的类型估计。至于表达式的类型推断将在表达式翻译章节中讨论。常量类型的推断没有太多的理论，实现也是比较简单的。一



一般而言，常量类型的推断主要有两种观点：

(1) 尽可能使用值域较大的类型。以常量 23.2 为例，既可以将其推断为 float，也可以将其推断为 double。按照本原则，编译器会将 23.2 推断为 double。

(2) 尽可能使用值域较小的类型。这种推断方式的本意就是以值域能够最小适应常量值的类型推断。以 18 为例，既可以将其推断为 int，也可以将其推断为 char。按照本原则，编译器会将 18 推断为 char。

在编译器设计中，无论采用哪一种观点都是可以接受的。在常量类型推断方面，不同编译器的实现是不尽相同的。Neo Pascal 是根据第二个观点实现的。实际上，简言之，常量类型推断就是常量值检索各种类型的值域，找到最适合（视推断观点而定）的类型即可。

程序 6-3 Type.cpp

```
1 void CType::ProcessConstType(int iPos)
2 {
3     if (SymbolTbl.ConstInfoTbl.at(iPos).m_StoreType!=StoreType::T_NONE)
4         return;
5     string szTmpVal=SymbolTbl.ConstInfoTbl.at(iPos).m_szVal;
6     switch (SymbolTbl.ConstInfoTbl.at(iPos).m_ConstType)
7     {
8         case CType::INTEGER:
9             {
10                int i=atoi(szTmpVal.c_str());
11                if (i<0)
12                    {
13                        if (i>=-128 && i<=127)
14                            SymbolTbl.ConstInfoTbl.at(iPos).
15                                m_StoreType=StoreType::T_SHORTINT;
16                        else
17                            if (i>=-32768 && i<=32767)
18                                SymbolTbl.ConstInfoTbl.at(iPos).
19                                    m_StoreType=StoreType::T_SMALLINT;
20                        else
21                            if (i>=-2147483648 && i<=2147483647)
22                                SymbolTbl.ConstInfoTbl.at(iPos).
23                                    m_StoreType=StoreType::T_INTEGER;
24                    }
25                else
26                    {
27                        if (i<=255)
28                            SymbolTbl.ConstInfoTbl.at(iPos).
29                                m_StoreType=StoreType::T_BYTE;
30                        else
31                            if (i<=65535)
32                                SymbolTbl.ConstInfoTbl.at(iPos).
33                                    m_StoreType=StoreType::T_WORD;
34                        else
```

```
35         if (i<=4294967295)
36             SymbolTbl.ConstInfoTbl.at(iPos).
37                 m_StoreType=StoreType::T_LONGWORD;
38     }
39     SymbolTbl.ConstInfoTbl.at(iPos).m_iVal=i;
40     SymbolTbl.ConstInfoTbl.at(iPos).
41         m_fVal=SymbolTbl.ConstInfoTbl.at(iPos).m_iVal;
42 };break;
43 case ConstType::BOOLEAN:
44     {
45         SymbolTbl.ConstInfoTbl.at(iPos).m_StoreType=StoreType::T_BOOLEAN;
46         if (SymbolTbl.ConstInfoTbl.at(iPos).m_szVal.compare("TRUE")==0)
47             SymbolTbl.ConstInfoTbl.at(iPos).m_bVal=true;
48         else
49             SymbolTbl.ConstInfoTbl.at(iPos).m_bVal=false;
50     };break;
51 case ConstType::STRING:
52     {
53         if (szTmpVal.length()==1)
54             SymbolTbl.ConstInfoTbl.at(iPos).m_StoreType=StoreType::T_CHAR;
55         else
56             SymbolTbl.ConstInfoTbl.at(iPos).
57                 m_StoreType=StoreType::T_STRING;
58     };break;
59 case ConstType::EREAL:
60 case ConstType::REAL:
61     {
62         SymbolTbl.ConstInfoTbl.at(iPos).
63             m_fVal=atof(SymbolTbl.ConstInfoTbl.at(iPos).m_szVal.c_str());
64         if (SymbolTbl.ConstInfoTbl.at(iPos).m_fVal>1.5e-45 &&
65             SymbolTbl.ConstInfoTbl.at(iPos).m_fVal<3.4e38)
66             SymbolTbl.ConstInfoTbl.at(iPos).
67                 m_StoreType=StoreType::T_SINGLE;
68         else
69             SymbolTbl.ConstInfoTbl.at(iPos).m_StoreType=StoreType::T_REAL;
70     };break;
71     }
72 }
```

整型常量的类型推断是最复杂的，因为其实现类型丰富，又存在符号的差异。另外，值得注意的是常量的 `m_ConstType` 信息，它并不是常量的实际类型，它是词法分析时得到的类型。例如，词法分析器能够分析得到 32.12 是实型，但无法准确获得它到底是单精度还是双精度。本程序的功能就是根据词法分析得到的一些初步类型信息及常量字面值进一步推断其实际的类型。经过了类型推断的常量与变量一样具有了非常精准的类型信息，这有利于表达式翻译与类型检查等。



综上所述，主要讨论了常量的类型推断。在静态类型语言中，用户变量的类型是完全由用户声明的，所以并不需要编译器推断。编译器需要推断的仅仅是一些临时变量的类型。实际上，临时变量类型的推断并不复杂。临时变量的作用就是临时存储表达式的中间结果，也就是说，它的类型实际上就是表达式的类型。然而，在 Neo Pascal 中，表达式类型的推断规则已经在类型系统表中有详尽的描述，所以可以通过检索类型系统表轻松实现临时变量的类型推断。关于如何应用类型系统表实现表达式的翻译，将在下一节中详述。

6.4 表达式翻译

6.4.1 表达式翻译基础

Pascal 语言所支持的表达式是比较有限的，从形式及功能上而言，Pascal 表达式也是比较规范的，虽然不及 C 语言灵活，但也足以满足实际编程的需要。下面，就来看看 Pascal 表达式的文法：

【文法 6-1】

表达式	→ 项 表达式 1
表达式 1	→ 关系运算符 050 项 051 表达式 1 ε
项	→ 因式 项 1
项 1	→ 低优先级运算符 050 因式 051 项 1 ε
因式	→ 因子 因式 1
因式 1	→ 高优先级运算符 050 因子 051 因式 1 ε
因子	→ 变量 053
	→ 常量 049
	→ (表达式)
	→ nil 100
	→ + 050 因子 052
	→ - 050 因子 052
	→ not 050 因子 052
	→ @ 变量 053 060
	→ [096 表达式列表 097]
关系运算符	→ = <> < > <= >= in
低优先级运算符	→ + - or xor
高优先级运算符	→ * / div mod shr shl and

在表达式文法中，没有将非终结符“变量”展开。因为“变量”涉及数组、结构等比较复杂的话题，所以将在下一节中详细分析。

笔者先来谈几个与表达式有关的话题。

(1) 运算符的种类。一般而言，运算符的种类并非越多越好。运算符过多，必定会使语言的易学、易用性降低，对语言的推广是不利的。当然，运算符的种类也不能过少，那样会导致语言功能的不足。那么，设计多少个运算符为宜呢？这个问题并没有非常科学的定论，更多地依赖于语言设计者的经验。

(2) 运算符的优先级。高级语言的运算符通常都是有优先级差异的。在设置运算符优先

级时，语言设计者主要考虑两个方面的因素：

1) 运算习惯。这是最重要的因素，例如，加、减运算符的优先级通常是低于乘、除法运算符的。这是一个数学常识，不应该颠覆。

2) 运算优先级别不应过多。在实际编程中，有经验的程序员很少依赖语言本身的运算优先级，为了安全起见，他们更愿意使用括号。尤其当优先级过于复杂时，大多数程序员可能更不信赖优先级了。而且优先级别过多会给初学者带来不便。虽然 C 语言如此经典，但是，在优先级设置方面，它却不是一个经典范例。相对而言，Pascal 的优先级别的数量比较合适，但是 Pascal 同样不是一个成功的例子。根据标准 Pascal 的规定，Pascal 的关系运算符优化级低于逻辑运算符，这是一个非常奇怪的规定，与人们一贯的逻辑思维方式是不符的。例如，书写 $i > 10 \text{ and } i < 100$ 的本意就是判断 i 的值是否介于 (10,100) 之间。不过，根据标准 Pascal 的规定，由于 and 优先级高于关系运算符，所以 $10 \text{ and } i$ 被优先计算，这就使程序员不得不使用括号来改写这个表达式。

在第 5 章中，笔者已经提到了表达式翻译的基本思想。利用栈结构分析与翻译表达式的方式是比较常见的，由于栈结构分析表达式算法是数据结构课程的相关内容，这里，只作简单介绍，不作深入讨论。

实际上，就表达式计算算法而言，编译技术中讨论的算法要比数据结构课程简单一些。当然，只是因为有些琐事是由词法分析器、语法分析器完成的，并不需要语义子程序关心。下面，就通过一个完整的实例来看看表达式计算的过程。

例 6-2 计算表达式 $3+5 \times 2-4$ 的值。

根据 Neo Pascal 文法，可以将表达式的推导过程抽象为语法分析树，如图 6-7 所示。在语法分析树中，笔者特意用虚线框标识语义子程序，以便与终结符区别。事实上，通常意义的语法分析树只是用于描述语法的推导过程，并不需要标注语义子程序。

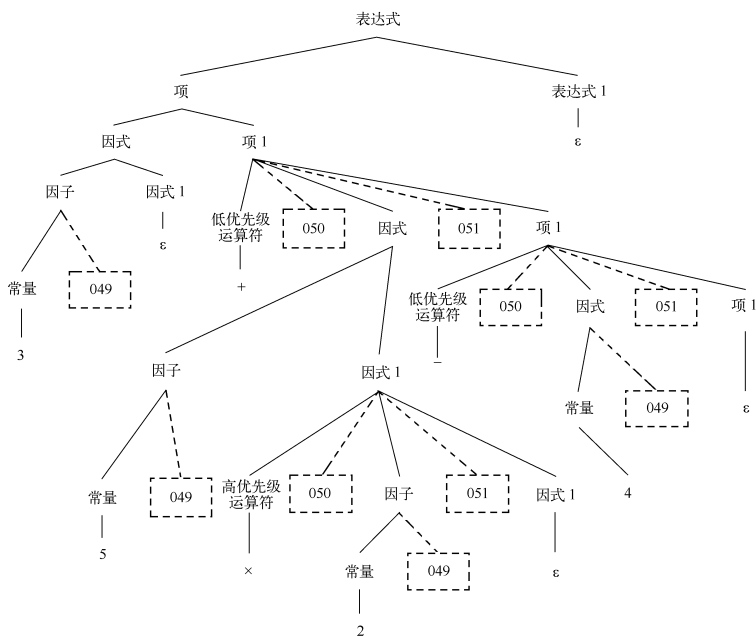


图 6-7 表达式 $3+5 \times 2-4$ 的语法分析树



不难发现，在语法分析树中只出现了三个语义子程序（049、050、051），现在问题的关键就是赋予这三个子程序一定的语义动作，使之达到预定的目标。那么，暂且定义这三个子程序的语义动作如下：

049：将当前单词压入操作数栈。

050：将当前单词压入运算符栈。

051：从操作数栈弹出两个操作数，从运算符栈弹出一个运算符，进行相应的计算，并将计算结果压入操作数栈。这里，值得注意的是先出栈的元素是第二操作数，而后出栈的元素是第一操作数。在处理一些不符合交换律的运算时，必须加倍注意。

下面，就来看看表达式处理的过程。

对语法分析树进行后序遍历，忽略非终结符与“ε”后，可以得到如下的序列：

3 049 + 050 5 049 × 050 2 049 051 - 050 4 049 051

根据先前定义的语义动作完成相应的操作，最终，操作数栈的栈顶元素即为表达式的结果。

如果分析过程完全正确且不考虑表达式组等情况，操作数栈内有且仅有一个元素，而运算符栈应该为空。笔者就不再模拟详细的入、出栈操作了，这应该是非常简单的。

读者可能会有疑问，包括 Neo Pascal 在内的许多编译器并不显式生成语法分析树，那么，又如何进行后序遍历呢？实际上，是否显式生成语法分析树并不影响语义动作的执行。前面曾经提到，自上而下语法分析的过程本身就是一次后序遍历，因此，并不会影响语义子程序的调用顺序。

上述常量表达式的计算实例，旨在说明编译器是如何计算表达式的。下面，将通过一个更一般化的例子，来谈谈变量表达式的处理。当然，含有变量的表达式是不可能由编译器在编译过程中计算得到结果的，那么，计算表达式的问题就变成了如何将表达式翻译成 IR 的问题了。实际上，这是非常简单的，只需对 051 的语义动作稍作修改即可，具体修改如下所示：

051：从操作数栈弹出两个操作数，从运算符栈弹出一个运算符，申请一个临时变量用于存储计算结果。生成形如 (Op, Op1, Op2, T) 的 IR，并将临时变量压入操作数栈。

以表达式 i+j×k-6 为例。这个表达式对应的语法分析树与图 6-7 是非常类似的，笔者就不再重绘了。后序遍历语法分析树后，可以得到如下的序列：

i 049 + 050 j 049 × 050 k 049 051 - 050 6 049 051

根据先前讨论的语义动作，可以得到如下的 IR 序列：

```
(MUL      j      ,      k      ,      T1)
(ADD      I      ,      T1     ,      T2)
(SUB      T2     ,      6      ,      T3)
```

这里的 IR 形式并不是 NPIR，只是一种逻辑表示形式而已。T1、T2、T3 是编译器临时变量，在表达式翻译过程中，每次运算的结果都必须用临时变量保存。有些读者可能会问：这样处理是否会产生大量的临时变量呢？确实如此，不过，在优化阶段及存储分配阶段，编译器会进行相关分析与优化，因此，并不会影响目标程序的效率。

至此，笔者已经详细解释了手工实现表达式的翻译的过程，并获得了一个相对完整的

IR 序列。这个过程就是实际表达式翻译的抽象模型，值得推敲。

6.4.2 深入表达式翻译

前面，提出了一个表达式翻译的基本模型，并通过实例分析了翻译模型的工作过程。本小节将在此基础上讨论如何应用这个翻译模型生成 NPIR，只有生成了 NPIR 才算是真正意义上实现了表达式的翻译。

实际上，在先前提出的翻译模型中，忽略了一个重要的元素——类型。数据结构课程讨论表达式计算时，并不涉及类型，当然，也不需要涉及类型。不过，对于编译器而言，这却是不可忽视的重要因素。在前面章节中，笔者已经介绍了类型系统的相关理论与实现，但始终没有将其应用到实际翻译中。这里，讨论的重点就是把类型系统引入到表达式翻译中，让读者明白类型系统并不是游离于表达式而存在的。

与翻译模型不同，作为一个实际编译器，Neo Pascal 在生成运算 IR 之前，通常还需要完成几项工作：

(1) 评估操作数的相容性。操作数在给定的运算符环境中是否相容是编译器设计者关心的。继续翻译生成 IR 的前提就是操作数必须相容。如果操作数不相容，编译器就必须终止分析并报错，否则后续的翻译与分析是没有任何意义的。

(2) 生成类型转换 IR。类型等价与类型相容的语义处理是不同的。在类型等价的情况下，编译器不必考虑类型隐式转换等问题。不过，在类型相容的情况下，编译器通常需要生成类型转换 IR，以保证运算 IR 操作数的类型匹配。

(3) 推断结果操作数的类型。在表达式翻译中，编译器通常需要使用临时变量来存储运算的中间结果，那么，临时变量的类型（即运算结果的类型）就需要由编译器推断得到。由于表达式经常出现在赋值语句中，有些读者可能误以为被赋值对象的类型就是表达式运算结果的类型，这是完全错误的。读者应该明确一个概念，表达式结果操作数的类型只取决于操作数的类型与运算符本身。那么，是不是表达式结果都是用临时变量存储的呢？的确如此，至少在翻译表达式时，编译器必须这么做。即使编译器在翻译 $A:=B+C$ 时，也不会生成直接类似于 (ADD, B, C, A) 这样的 IR。而是申请一个临时变量 T，将 B+C 的计算结果暂存于 T 中，然后，再考虑将 T 的值赋给 A。有些读者可能会质疑为什么需要 T 作为暂存呢？事实上，除了 T 与 A 的类型完全等价的情况之外，将 T 的值赋给 A 时，编译器还需要考虑两者之间的隐式转换问题，所以临时变量 T 是有存在必要的。在表达式翻译阶段，即使是 T 与 A 的类型完全等价的情况下，为了使表达式翻译规程统一，也不必考虑将 T 省去。当然，设计者也不希望产生冗长的临时变量，在后续优化阶段，借助于一些专门的 IR 优化算法解决这类问题是非常有效的。

(4) 常量折叠。简言之，常量折叠就是针对那些所有操作数都是常量的运算或表达式，编译器会在编译阶段直接计算得到其运算结果，而不必生成相应的运算 IR。严格地说，这个话题应该属于编译优化的范畴。不过，在表达式翻译过程中，对于编译器来说，无论采用何种语法分析方式，发现与处理这类问题都是相对简单的，而其产生的效果也是非常显著的。因此，编译器设计者都会考虑将常量折叠纳入表达式翻译方案中。当然，表达式翻译时可以完成的常量折叠是有限的，它并不能完全替代优化阶段的常量折叠算法。常量折叠的思想并不复杂，但是，具体实现却比较容易出错。除了常量的类型推断之外，编译器设计者应该特



别注意编译环境与运行环境不一致可能导致的错误。

至此，笔者详细讨论了 Neo Pascal 表达式翻译的基本问题。不过，这些问题并非程序设计语言表达式翻译的全部内容。试图设计一个完备的模型以应对一切程序设计语言复杂且多变的应用需求可能是徒劳的。在编译器设计中，设计者经常需要对一些经典的翻译模型进行改进与完善以适应不同语言的需要，例如，C 语言的布尔表达式的短路机制。这是编译原理课程中一个非常经典的翻译话题，有兴趣的读者可以参考相关书籍，其基本思想是值得借鉴与学习的。

6.4.3 表达式翻译的实现

这里，先简单讨论一下 Neo Pascal 表达式翻译的相关数据结构。先前提到的翻译模型主要涉及两个栈结构，即操作数栈、运算符栈。声明形式如下：

【声明 6-6】

```
stack<OpInfo> Operand;      //操作数栈
stack<int> Operation;      //运算符栈
```

Operation 栈的元素就是运算符的 ID，所以其类型就是 int。这两个栈结构比较简单，不再多作解释。下面，笔者来解释各语义子程序主要功能，以便读者理解和阅读源代码。

- semantic050: 1. 将当前单词压入 Operation 栈。
- semantic049: 1. 生成常量操作数，并压入 Operand 栈。
- semantic051: 1. 有效性检查，即判断操作数个数是否大于或等于 2。
2. 类型相容性检查。
3. 生成双目运算的 IR 语句或进行常量折叠。
4. 将结果操作数压入 Operand 栈。
- semantic052: 1. 有效性检查，即判断操作数个数是否大于或等于 1。
2. 类型相容性检查。
3. 生成单目运算的 IR 语句或进行常量折叠。
4. 将结果操作数压入 Operand 栈。
- semantic060: 1. 生成@运算的 IR 语句。
- semantic053: 1. 生成变量操作数，并压入 Operand 栈。
- semantic096: 1. 设置 iExpListFlag、iExpListNum 等标志栈，用于收集集合形式的操作数。
- semantic097: 1. 生成集合形式的操作数及其 IR，并压入 Operand 栈。

本小节只介绍前 4 个子程序的实现，而 semantic060、semantic053、semantic096 与 semantic097 将在后续章节中讨论。

程序 6-4 semantic.cpp

相关文法:

- 表达式 1 → 关系运算符 050 项 051 表达式 1 | ε
- 项 1 → 低优先级运算符 050 因式 051 项 1 | ε
- 因式 1 → 高优先级运算符 050 因子 051 因式 1 | ε
- 因子 → + 050 因子 052
- 因子 → - 050 因子 052
- 因子 → not 050 因子 052

```
1  bool semantic050()
2  {
3      Operation.push(TokenList.at(iListPos-1).m_iKind);
4      return true;
5  }
```

semantic050 获得当前单词的 ID（即运算符的 ID），并将其压入 Operation 栈。除了“@”运算符外，其他运算符都必须由 semantic050 压入 Operation 栈，并无需区分是单目运算符还是双目运算符。

程序 6-5 semantic.cpp

相关文法:

因子 → 常量 049

```
1  bool semantic049()
2  {
3      OpInfo Tmp;
4      Tmp.m_iType=OpInfo::CONST;
5      Tmp.m_iLink=atoi(TokenList.at(iListPos-1).m_szContent.c_str());
6      Tmp.m_bRef=false;
7      Operand.push(Tmp);
8      return true;
9  }
```

semantic049 的功能就是为常量操作数生成一个 OpInfo 的实例，并将其压入 Operand 栈供翻译表达式使用。设置 m_iType 属性为 OpInfo::CONST 表示这个操作数是常量操作数。当然，其 m_iLink 属性的值就是该常量符号在常量信息表中的位序号，即指向该常量符号的指针。m_bRef 标志对于常量操作数是没有任何意义的。

程序 6-6 semantic.cpp

相关文法:

表达式 1 → 关系运算符 050 项 051 表达式 1 | ϵ

项 1 → 低优先级运算符 050 因子 051 项 1 | ϵ

因子 1 → 高优先级运算符 050 因子 051 因子 1 | ϵ

```
1  bool semantic051()
2  {
3      OpInfo Tmp1,Tmp2;
4      int TmpResult;
5      int TmpOperation;
6      if (!Operand.empty())
7          Tmp2=Operand.top();
8      else
9          {
10         EmitError("缺少操作数",TokenList.at(iListPos-1));
11         return false;
12     }
```




```
13     Operand.pop();
14     if (!Operand.empty())
15         Tmp1=Operand.top();
16     else
17     {
18         EmitError("缺少操作数",TokenList.at(iListPos-1));
19         return false;
20     }
21     Operand.pop();
22     if (!Operation.empty())
23         TmpOperation=Operation.top();
24     else
25     {
26         EmitError("缺少操作数",TokenList.at(iListPos-1));
27         return false;
28     }
29     Operation.pop();
30     TmpResult=CType::TypeCompatible(Tmp1,Tmp2,TmpOperation);
31     if (TmpResult!=-1)
32     {
33         if (Tmp1.m_iType==OpInfo::CONST && Tmp2.m_iType==OpInfo::CONST)
34         {
35             return OpConstSemantic(Tmp1,Tmp2,SymbolTbl.TypeSysTbl.at(TmpResult)
36                 .m_iProcessId,SymbolTbl.TypeSysTbl.at(TmpResult).m_iRsltType);
37         }
38         else
39         {
40             OpInfo TmpRslt;
41             TmpRslt.m_iType=OpInfo::VAR;
42             TmpRslt.m_iLink=SymbolTbl.GetTmpVar(SymbolTbl.ProcStack.top(),
43                 (StoreType)SymbolTbl.TypeSysTbl.at(TmpResult).m_iRsltType);
44             if (OpVarSemantic(Tmp1,Tmp2,TmpRslt,
45                 SymbolTbl.TypeSysTbl.at(TmpResult))=false)
46                 return false;
47             Operand.push(TmpRslt);
48         }
49         return true;
50     }
51     else
52     {
53         EmitError("操作数类型不兼容",TokenList.at(iListPos-1));
54         return false;
55     }
56     return true;
57 }
```

第 6~29 行：从 Operand、Operation 栈获取第一、二操作数及运算符。这个过程是比较简单的，读者只需注意以下两个要点：第一，操作数的顺序。由于栈的特性，所以先出栈的元素是第二操作数，后出栈的元素才是第一操作数。第二，栈访问的安全性。这里笔者进行了严格的栈空检查。有些读者可能会有疑问，这里的三个栈空检查是否有必要？从理论上来说，确实可以省略这三个栈空检查，因为语法分析器完全有能力保证栈访问的安全性。不过，这并不是是一种良好的编程习惯，所以笔者不推荐使用。

第 30 行：根据第一、二操作数类型及运算符 ID，调用 TypeCompatible 函数检索类型系统表。当 TypeCompatible 函数返回-1 时，则表示两个操作数在给定运算符环境下是不相容的。当 TypeCompatible 函数返回大于或等于 0 的值时，则表示两个操作数在给定运算符环境下是相容的，且给出了相应的翻译规则。所谓“翻译规则”就是指类型系统表的一个表项。前面已经详细解释了类型系统表的相关结构与声明，不再赘述。

第 33 行：判断两个操作数是否都是常量，如果都是常量，则进行常量折叠，否则进行表达式翻译。由于前面已经检查了操作数的相容性，所以这里并不需要额外判断。

第 35、36 行：调用 OpConstSemantic 完成常量折叠。这里，只需将两个操作数、类型系统表表项的 m_iProcessId 属性和 m_iRsltType 属性作为参数传递给 OpConstSemantic 函数即可。关于 OpConstSemantic 函数的详细实现稍后分析。

第 41~43 行：申请一个临时变量用以存储表达式的计算结果，而临时变量的类型由类型系统表表项的 m_iRsltType 属性决定。值得注意的是，C、Pascal 的表达式都比较简单，其计算结果类型只可能是基本类型，而不可能是复杂类型（数组、结构等），故 m_iRsltType 属性足以描述结果类型。不过，在 C++等面向对象语言中，允许表达式的结果类型是复杂类型甚至类，因此推断结果类型可能会复杂一些。

第 44、45 行：调用 OpVarSemantic 完成表达式翻译。将两个操作数、运算符及类型系统表表项、结果操作数作为参数传递给 OpConstSemantic 函数即可，生成 IR 的工作由 OpVarSemantic 函数完成。关于 OpVarSemantic 函数的详细实现稍后分析。

下面，再来看看 OpVarSemantic、OpConstSemantic 的实现细节。

如果操作数满足常量折叠的要求，则调用 OpConstSemantic 函数计算操作结果，否则调用 OpVarSemantic 函数生成计算的 IR。当表达式的操作数不都为常量时，也就无法进行常量折叠。OpVarSemantic 的设计思想非常简单，即根据类型系统表表项的相关描述生成相应的 IR。简言之，就是根据类型系统表表项的 m_iVarProcessId 属性，完成相应的语义动作。在 Neo Pascal 的类型系统表中，一共设置了 5 种语义动作，分别以编号 1~5 标识。由于实际的 Neo Pascal 类型系统表规模太大，笔者仅截取了其中数行表项为例，见表 6-3。

表 6-3 Neo Pascal 类型系统表示例

操作符	操作数 1 类型	操作数 2 类型	结果操作数类型	变量操作码	IR 操作符	类型转换 1	类型转换 2
=	T_CHAR	T_CHAR	T_BOOLEAN	1	EQU_1	None	None
=	T_CHAR	T_STRING	T_BOOLEAN	2	EQU_S	ChrToStr	None
=	T_INTEGER	T_SHORTINT	T_BOOLEAN	3	EQU_4	ShortToInt	None
=	T_ENUM	T_ENUM	T_BOOLEAN	4	EQU_1	None	None
=	T_BYTE	T_SHORTINT	T_BOOLEAN	5	EQU_2	ByteToSmall	ShortToSmall



注意，为了便于阅读，这里给出的类型系统表是未经编码的，并且直接以字段的中文含义标识列名。其中，变量操作码即为 `m_iVarProcessId` 属性。下面，就针对这 5 种语义动作逐一解释，如表 6-4 所示。

表 6-4 变量操作码对应的语义动作

变量操作码	语义动作说明
1	<p>表示无需进行类型转换。当操作数 1、操作数 2 的类型完全等价或其他特殊情况时，编译器无需作任何类型转换，直接按照“IR 操作符”生成 IR 即可。例如，第 1 行表项就是如此。该表项用于描述两个 <code>char</code> 类型操作数在进行等号运算时的相关类型处理信息。由于两个 <code>char</code> 类型操作数进行等号运算时并不需要任何转换，所以“类型转换 1”、“类型转换 2”都是“None”。在翻译这类表达式时，主要完成如下工作：</p> <ol style="list-style-type: none"> 1. 根据“结果操作数类型”申请一个 <code>boolean</code> 类型的临时变量 <code>_T</code>。 2. 依据“IR 操作符”生成一条 IR 指令： (<code>EQU_1, Op1, Op2, _T</code>)
2	<p>表示操作数 1 需要进行类型转换。这种情况表明操作数之间是相容而不等价的，编译器需要对操作数 1 作隐式类型转换后，才能生成 IR。例如，第 2 行表项就是如此。该表项用于描述一个 <code>char</code> 类型操作数与一个 <code>string</code> 类型操作数在进行等号运算时的相关类型处理信息。由于 <code>char</code> 类型与 <code>string</code> 类型数据的存储形式完全不同，一般而言，需要将 <code>char</code> 类型数据转换为 <code>string</code> 类型数据后，才能进行后续分析，所以“类型转换 1”为“<code>ChrToStr</code>”。在翻译这类表达式时，主要完成如下工作：</p> <ol style="list-style-type: none"> 1. 根据“结果操作数类型”申请一个 <code>boolean</code> 类型的临时变量 <code>_T</code>。 2. 申请一个 <code>string</code> 类型的临时变量 <code>_T1</code>。 3. 根据“类型转换 1”生成一条类型转换 IR 指令： (<code>ChrToStr, Op1, NULL, _T1</code>) 4. 根据“IR 操作符”生成一条 IR 指令： (<code>EQU_S, _T1, Op2, _T</code>)
3	<p>表示操作数 2 需要进行类型转换。这种情况表明操作数之间是相容而不等价的，编译器需要对操作数 2 作隐式类型转换后，才能生成 IR。例如，第 3 行表项就是如此。该表项用于描述一个 <code>integer</code> 类型操作数与一个 <code>shortint</code> 类型操作数在进行等号运算时的相关类型处理信息。由于 <code>integer</code> 类型与 <code>shortint</code> 类型数据的存储形式不同，一般而言，需要将 <code>shortint</code> 类型数据转换为 <code>integer</code> 类型数据后，才能进行后续分析，所以“类型转换 2”为“<code>ShortToInt</code>”。在翻译这类表达式时，主要完成如下工作：</p> <ol style="list-style-type: none"> 1. 根据“结果操作数类型”申请一个 <code>boolean</code> 类型的临时变量 <code>_T</code>。 2. 申请一个 <code>integer</code> 类型的临时变量 <code>_T1</code>。 3. 根据“类型转换 2”生成一条类型转换 IR 指令： (<code>ShortToInt, Op2, NULL, _T1</code>) 4. 根据“IR 操作符”生成一条 IR 指令： (<code>EQU_4, Op1, _T1, _T</code>)
4	<p>这是一个比较特殊的语义动作，它的处理对象必定是两个枚举类型操作数。在 Pascal 语言中，虽然枚举值最终是以整型存储的，但是却不能因为这个原因忽视枚举类型的特殊性。两个枚举值进行关系运算的前提是这两个操作数所属的枚举类型必须一致。例如，</p> <pre> VAR p:(a,b,c); q:(d,e,f); p=a; q=d; </pre> <p>显然，<code>p</code> 与 <code>q</code> 实际存储的数据是同样的，但是讨论 <code>p</code> 与 <code>q</code> 的关系运算是没有意义的。先前的类型相容性判断是不能达到这一目标的，所以在处理枚举类型变量运算时，需要进行相应的有效性判断。请读者参见表 6-3 第 4 行示例，详细的翻译方式与语义动作 1 类似，这里不再赘述</p>
5	<p>表示操作数 1、操作数 2 都需要进行类型转换。这种情况表明操作数之间是相容而不等价的，编译器需要对操作数 1、操作数 2 作隐式类型转换后，才能生成 IR。例如，第 5 行表项就是如此。该表项用于描述一个 <code>byte</code> 类型操作数与一个 <code>shortint</code> 类型操作数在进行等号运算时的相关类型处理信息。由于 <code>byte</code> 类型与 <code>shortint</code> 类型都是 1 字节变量，但是前者是无符号形式，而后者是有符号形式，所以无法直接比较。一般而言，需要同时将 <code>byte</code> 和 <code>shortint</code> 类型数据同时转换为 2 字节有符号类型数据后，才能进行后续分析，所以“类型转换 1”为“<code>ByteToSmall</code>”，而“类型转换 2”为“<code>ShortToSmall</code>”。当然，Pascal 语言的这种处理方式是比较保守且安全的，但并不排除有些语言可能采用一些比较激进的方式来处理这类问题。在翻译这类表达式时，主要完成如下工作：</p> <ol style="list-style-type: none"> 1. 根据“结果操作数类型”申请一个 <code>boolean</code> 类型的临时变量 <code>_T</code>。 2. 申请一个 <code>Smallint</code> 类型的临时变量 <code>_T1</code>。 3. 根据“类型转换 1”生成一条类型转换 IR 指令： (<code>ByteToSmall, Op1, NULL, _T1</code>)

(续)

变量操作码	语义动作说明
5	4. 申请一个 Smallint 类型的临时变量 T2。 5. 根据“类型转换 2”生成一条类型转换 IR 指令： (ShortToSmall, Op2, NULL, T2) 6. 根据“IR 操作符”生成一条 IR 指令： (EQU_2, T1, T2, T)

无论是双目运算表达式，还是单目运算表达式，都是由 OpVarSemantic 函数生成 IR 的，而生成 IR 的依据就是类型系统表。下面，再来看 OpVarSemantic 函数的源代码实现。

程序 6-7 semantic.cpp

```

1  bool OpVarSemantic(OpInfo Op1,OpInfo Op2,OpInfo Rslt,TypeSysInfo Tmp)
2  {
3      vector<IRCode> *P=&SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes;
4      switch (Tmp.m_iVarProcessId)
5      {
6      case 1:
7          {
8              P->push_back(EmitIR(OpType(Tmp.m_iIR),Op1,Op2,Rslt));
9          };break;
10     case 2:
11         {
12             OpInfo TmpOp;
13             TmpOp.m_iType=OpInfo::VAR;
14             TmpOp.m_iLink=SymbolTbl.GetTmpVar(SymbolTbl.ProcStack.top(),
15             (OpType)Tmp.m_Convert1);
16             P->push_back(EmitIR((OpType)Tmp.m_Convert1,Op1,TmpOp));
17             P->push_back(EmitIR((OpType)Tmp.m_iIR,TmpOp,Op2,Rslt));
18         };break;
19     case 3:
20         {
21             OpInfo TmpOp;
22             TmpOp.m_iType=OpInfo::VAR;
23             TmpOp.m_iLink=SymbolTbl.GetTmpVar(SymbolTbl.ProcStack.top(),
24             (OpType)Tmp.m_Convert1);
25             P->push_back(EmitIR((OpType)Tmp.m_Convert1,Op2,TmpOp));
26             P->push_back(EmitIR((OpType)Tmp.m_iIR,Op1,TmpOp,Rslt));
27         };break;
28     case 4:
29         {
30             if (CType::GetOpTypeLink(Op1)==CType::GetOpTypeLink(Op2))
31                 P->push_back(EmitIR(OpType(Tmp.m_iIR),Op1,Op2,Rslt));
32             else
33                 {
34                     EmitError("枚举类型不匹配",TokenList.at(iListPos-1));
35                     return false;

```



```
36         }
37     };break;
38     case 5:
39     {
40         OpInfo TmpOp1,TmpOp2;
41         TmpOp1.m_iType=OpInfo::VAR;
42         TmpOp1.m_iLink=SymbolTbl.GetTmpVar(SymbolTbl.ProcStack.top()),
43         (OpType)Tmp.m_Convert1);
44         TmpOp2.m_iType=OpInfo::VAR;
45         TmpOp2.m_iLink=SymbolTbl.GetTmpVar(SymbolTbl.ProcStack.top()),
46         (OpType)Tmp.m_Convert2);
47         P->push_back(EmitIR((OpType)Tmp.m_Convert1,Op1,TmpOp1));
48         P->push_back(EmitIR((OpType)Tmp.m_Convert2,Op2,TmpOp2));
49         P->push_back(EmitIR((OpType)Tmp.m_iIR, TmpOp1, TmpOp2,Rslt));
50     };break;
51     }
52     return true;
53 }
```

理解了类型系统表 `m_iVarProcessId` 属性的含义及其相应的语义动作后，分析表达式翻译函数 `OpVarSemantic` 的源代码就变得非常容易了。详细的源代码注释似乎已经没什么意义了，这里，笔者只想对其中的 `SymbolTbl.GetTmpVar` 函数稍作解释。

在 `OpVarSemantic` 函数中，调用了一个 `SymbolTbl.GetTmpVar` 函数。不过，这个函数是 `GetTmpVar` 的另一种重载形式，它的第 2 个参数的类型是 `OpType`（即 IR 操作符）。根据这个 `OpType` 类型的参数，函数得到相应的类型信息，并以此类型信息申请临时变量。注意，函数对第 2 个参数是有一定限制的，必须是 `NPIR` 中类型转换相关操作符，因为只有这类操作符才包括类型信息，例如，`ChrToStr`、`ByteToInt` 等。获得了临时变量的类型信息后，函数就按照标准的申请流程完成临时变量的申请即可。由于相关源代码比较简单，就不再详细列出。

下面，再来谈谈常量折叠的相关实现。这里的常量折叠的基本思想与表达式翻译是比较类似的，它也是基于类型系统表完成的，只不过它需要处理的分支情况要多得多。在 `Neo Pascal` 表达式分析过程中，合法的常量折叠情形一共有 47 种。不过，这 47 种分支的工作流程都是同样的，即根据运算符及不同类型的操作数，由编译器在编译阶段计算结果，并将结果压入 `Operand` 栈，常量折叠的过程并不生成任何 IR。

程序 6-8 Semantic.cpp

```
1 bool OpConstSemantic(OpInfo Op1,OpInfo Op2,int iProcessId,int iReturnType)
2 {
3     OpInfo Rslt;
4     if (!OpConstFold(Op1,Op2,iProcessId,iReturnType,Rslt))
5         return false;
6     Operand.push(Rslt);
7     return true;
```

```

8   }
9   bool OpConstFold(OpInfo Op1,OpInfo Op2,int iProcessId,int iReturnType,OpInfo &Rslt)
10  {
11      ConstInfo RsltConst,Op1Const,Op2Const;
12      Rslt.m_iType=OpInfo::CONST;           //设置结果操作数的 m_iType 属性
13      RsltConst.m_StoreType=(StoreType)iReturnType; //设置结果操作数的 m_StoreType 属性
14      Op1Const=SymbolTbl.ConstInfoTbl.at(Op1.m_iLink); //获取操作数 1 的常量信息
15      if (Op2.m_iType!=OpInfo::NONE)
16          Op2Const=SymbolTbl.ConstInfoTbl.at(Op2.m_iLink);//获取操作数 2 的常量信息
17      switch (iProcessId) //根据 m_iProcessId 属性，执行相关的语义动作
18      {
19      case 1: //1、boolean=boolean 2、string(char)=string(char) 3、pointer=pointer
20          {
21              RsltConst.m_bVal=Op1Const.m_szVal.compare(Op2Const.m_szVal)==0;
22              RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
23          };break;
24      case 2: //整型(实型)=整型(实型)
25          {
26              RsltConst.m_bVal=Op1Const.m_fVal==Op2Const.m_fVal;
27              RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
28          };break;
29      case 3: //1、boolean<>boolean 2、string(char)<>string(char) 3、pointer<>pointer
30          {
31              RsltConst.m_bVal=Op1Const.m_szVal.compare(Op2Const.m_szVal)!=0;
32              RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
33          };break;
34      case 4: //整型(实型)<>整型(实型)
35          {
36              RsltConst.m_bVal=Op1Const.m_fVal!=Op2Const.m_fVal;
37              RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
38          };break;
39      case 5: //string(char)<string(char)
40          {
41              RsltConst.m_bVal=Op1Const.m_szVal.compare(Op2Const.m_szVal)<0;
42              RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
43          };break;
44      case 6: //boolean<boolean
45          {
46              RsltConst.m_bVal=Op1Const.m_bVal<Op2Const.m_bVal;
47              RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
48          };break;
49      case 7: //整型(实型)<整型(实型)
50          {
51              RsltConst.m_bVal=Op1Const.m_fVal<Op2Const.m_fVal;
52              RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
53          };break;

```



```
54     case 8:           //string(char)>string(char)
55         {
56             RsltConst.m_bVal=Op1Const.m_szVal.compare(Op2Const.m_szVal)>0;
57             RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
58         };break;
59     case 9:           //boolean>boolean
60         {
61             RsltConst.m_bVal=Op1Const.m_bVal>Op2Const.m_bVal;
62             RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
63         };break;
64     case 10:          //整型(实型)>整型(实型)
65         {
66             RsltConst.m_bVal=Op1Const.m_fVal>Op2Const.m_fVal;
67             RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
68         };break;
69     case 11:          //string(char)<=string(char)
70         {
71             RsltConst.m_bVal=Op1Const.m_szVal.compare(Op2Const.m_szVal)<=0;
72             RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
73         }
74     case 12:          //boolean<=boolean
75         {
76             RsltConst.m_bVal=Op1Const.m_bVal<=Op2Const.m_bVal;
77             RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
78         };break;
79     case 13:          //整型(实型)<=整型(实型)
80         {
81             RsltConst.m_bVal=Op1Const.m_fVal<=Op2Const.m_fVal;
82             RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
83         };break;
84     case 14:          //string(char)>=string(char)
85         {
86             RsltConst.m_bVal=Op1Const.m_szVal.compare(Op2Const.m_szVal)>=0;
87             RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
88         };break;
89     case 15:          //boolean>=boolean
90         {
91             RsltConst.m_bVal=Op1Const.m_bVal>=Op2Const.m_bVal;
92             RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
93         };break;
94     case 16:          //整型(实型)>=整型(实型)
95         {
96             RsltConst.m_bVal=Op1Const.m_fVal>=Op2Const.m_fVal;
97             RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
98         };break;
99     case 17:          //整型(实型)-整型(实型)
```

```
101     {
102         RsltConst.m_fVal=Op1Const.m_fVal-Op2Const.m_fVal;
103         char cBuffer[20];
104         gcvt(RsltConst.m_fVal,10,cBuffer);
105         RsltConst.m_szName=cBuffer;
106     };break;
107 case 18:         //string(char)+string(char)
108     {
109         RsltConst.m_szVal.append(Op1Const.m_szVal);
110         RsltConst.m_szVal.append(SymbolTbl.ConstInfoTbl.at(Op2.m_iLink).m_szVal);
111         RsltConst.m_szName=RsltConst.m_szVal;
112     };break;
113 case 19:         //整型(实型)+整型(实型)
114     {
115         RsltConst.m_fVal=Op1Const.m_fVal+Op2Const.m_fVal;
116         char cBuffer[50];
117         gcvt(RsltConst.m_fVal,10,cBuffer);
118         RsltConst.m_szName=cBuffer;
119     };break;
120 case 20:         //boolean or boolean
121     {
122         RsltConst.m_bVal=Op1Const.m_bVal || Op2Const.m_bVal;
123         RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
124     };break;
125 case 21:         //整型 or 整型
126     {
127         RsltConst.m_iVal=Op1Const.m_iVal | Op2Const.m_iVal;
128         char cBuffer[50];
129         RsltConst.m_fVal=RsltConst.m_iVal;
130         itoa(RsltConst.m_iVal,cBuffer,10);
131         RsltConst.m_szName=cBuffer;
132     };break;
133 case 22:         //boolean xor boolean
134     {
135         RsltConst.m_bVal=Op1Const.m_bVal ^ Op2Const.m_bVal;
136         RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
137     };break;
138 case 23:         //整型 xor 整型
139     {
140         RsltConst.m_iVal=Op1Const.m_iVal ^ Op2Const.m_iVal;
141         char cBuffer[50];
142         RsltConst.m_fVal=RsltConst.m_iVal;
143         itoa(RsltConst.m_iVal,cBuffer,10);
144         RsltConst.m_szName=cBuffer;
145     };break;
146 case 24:         //整型(实型) * 整型(实型)
```




```
147     {
148         RsltConst.m_fVal=Op1Const.m_fVal*Op2Const.m_fVal;
149         char cBuffer[50];
150         gcvt(RsltConst.m_fVal,10,cBuffer);
151         RsltConst.m_szName=cBuffer;
152     };break;
153 case 25:    //整型(实型)/整型(实型)
154     {
155         if (Op2Const.m_fVal==0)
156         {
157             EmitError("除数不能为",TokenList.at(iListPos-1));
158             return false;
159         }
160         RsltConst.m_fVal=Op1Const.m_fVal/Op2Const.m_fVal;
161         char cBuffer[50];
162         gcvt(RsltConst.m_fVal,10,cBuffer);
163         RsltConst.m_szName=cBuffer;
164     };break;
165 case 26:    //整型 div 整型
166     {
167         if (Op2Const.m_iVal==0)
168         {
169             EmitError("除数不能为",TokenList.at(iListPos-1));
170             return false;
171         }
172         RsltConst.m_iVal=Op1Const.m_iVal / Op2Const.m_iVal;
173         char cBuffer[50];
174         RsltConst.m_fVal=RsltConst.m_iVal;
175         itoa(RsltConst.m_iVal,cBuffer,10);
176         RsltConst.m_szName=cBuffer;
177     };break;
178 case 27:    //整型 mod 整型
179     {
180         if (Op2Const.m_iVal==0)
181         {
182             EmitError("除数不能为",TokenList.at(iListPos-1));
183             return false;
184         }
185         if (Op1Const.m_iVal<0)
186         {
187             EmitError("被求模数不允许小于",TokenList.at(iListPos-1));
188             return false;
189         }
190         RsltConst.m_iVal=Op1Const.m_iVal % Op2Const.m_iVal;
191         char cBuffer[50];
192         RsltConst.m_fVal=RsltConst.m_iVal;
```

```
193         itoa(RsltConst.m_iVal,cBuffer,10);
194         RsltConst.m_szName=cBuffer;
195     };break;
196 case 28:         //整型 shr 整型
197     {
198         if (Op2Const.m_iVal<0 || Op2Const.m_iVal>255)
199             {
200                 EmitError("移位量只能介于 0~255 之间",TokenList.at(iListPos-1));
201                 return false;
202             }
203         RsltConst.m_iVal=Op1Const.m_iVal >> Op2Const.m_iVal;
204         char cBuffer[50];
205         RsltConst.m_fVal=RsltConst.m_iVal;
206         itoa(RsltConst.m_iVal,cBuffer,10);
207         RsltConst.m_szName=cBuffer;
208     };break;
209 case 29:         //整型 shl 整型
210     {
211         if (Op2Const.m_iVal<0 || Op2Const.m_iVal>255)
212             {
213                 EmitError("移位量只能介于 0~255 之间",TokenList.at(iListPos-1));
214                 return false;
215             }
216         RsltConst.m_iVal=Op1Const.m_iVal << Op2Const.m_iVal;
217         char cBuffer[50];
218         itoa(RsltConst.m_iVal,cBuffer,10);
219         RsltConst.m_fVal=RsltConst.m_iVal;
220         RsltConst.m_szName=cBuffer;
221     };break;
222 case 30:         //整型 and 整型
223     {
224         RsltConst.m_iVal=Op1Const.m_iVal & Op2Const.m_iVal;
225         char cBuffer[50];
226         RsltConst.m_fVal=RsltConst.m_iVal;
227         itoa(RsltConst.m_iVal,cBuffer,10);
228         RsltConst.m_szName=cBuffer;
229     };break;
230 case 31:         //boolean and boolean
231     {
232         RsltConst.m_bVal=Op1Const.m_bVal && Op2Const.m_bVal;
233         RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
234     };break;
235 case 32:         //set * set
236     {
237         RsltConst.m_szSet=SetMul(Op1Const.m_szSet,Op2Const.m_szSet);
238         RsltConst.m_szName=RsltConst.m_szSet;
```



```
239         };break;
240     case 33:         // 一整型(实型)
241     {
242         RsltConst.m_fVal=-Op1Const.m_fVal;
243         char cBuffer[50];
244         gcvt(RsltConst.m_fVal,10,cBuffer);
245         RsltConst.m_szName=cBuffer;
246     };break;
247     case 34:         // not 整型
248     {
249         RsltConst.m_iVal=-Op1Const.m_iVal;
250         char cBuffer[50];
251         RsltConst.m_fVal=RsltConst.m_iVal;
252         itoa(RsltConst.m_iVal,cBuffer,10);
253         RsltConst.m_szName=cBuffer;
254     };break;
255     case 35:         //not boolean
256     {
257         RsltConst.m_bVal=!Op1Const.m_bVal;
258         RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
259     };break;
260     case 36:         //enum = enum
261     {
262         if (Op1Const.m_iEnumIdx!=Op2Const.m_iEnumIdx)
263         {
264             EmitError("枚举类型不兼容",TokenList.at(iListPos-1));
265             return false;
266         }
267         RsltConst.m_bVal=Op1Const.m_szName.compare(Op2Const.m_szName)==0;
268         RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
269     };break;
270     case 37:         //enum <> enum
271     {
272         if (Op1Const.m_iEnumIdx!=Op2Const.m_iEnumIdx)
273         {
274             EmitError("枚举类型不兼容",TokenList.at(iListPos-1));
275             return false;
276         }
277         RsltConst.m_bVal=Op1Const.m_szName.c_str()!=Op2Const.m_szName.c_str();
278         RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
279     };break;
280     case 38:         //set = set
281     {
282         RsltConst.m_bVal=Op1Const.m_szSet.compare(Op2Const.m_szSet)==0;
283         RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
284     };break;
```

```
286     case 39:           //set <> set
287         {
288             RsltConst.m_bVal=Op1Const.m_szSet.compare(Op2Const.m_szSet)!=0;
289             RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
290         };break;
291     case 40:           //整型 in set
292         {
293             if (Op1Const.m_iVal<0 || Op1Const.m_iVal>255)
294             {
295                 EmitError("集合元素只能介于 0~255 之间",TokenList.at(iListPos-1));
296                 return false;
297             }
298             RsltConst.m_bVal=Op2Const.m_szSet.at(Op1Const.m_iVal)=='1';
299             RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
300         };break;
301     case 41:           //set - set
302         {
303             RsltConst.m_szSet=SetDel(Op1Const.m_szSet,Op2Const.m_szSet);
304             RsltConst.m_szName=RsltConst.m_szSet;
305         };break;
306     case 42:           //set + set
307         {
308             RsltConst.m_szSet=SetAdd(Op1Const.m_szSet,Op2Const.m_szSet);
309             RsltConst.m_szName=RsltConst.m_szSet;
310         };break;
311     case 43:           //char in set
312         {
313             if (Op1Const.m_szVal[0]<0 || Op1Const.m_szVal[0]>255)
314             {
315                 EmitError("集合元素只能介于 0~255 之间",TokenList.at(iListPos-1));
316                 return false;
317             }
318             RsltConst.m_bVal=Op2Const.m_szSet.at(Op1Const.m_szVal[0]=='1';
319             RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
320         };break;
321     case 44:           //enum > enum
322         {
323             if (Op1Const.m_iEnumIdx!=Op2Const.m_iEnumIdx)
324             {
325                 EmitError("枚举类型不兼容",TokenList.at(iListPos-1));
326                 return false;
327             }
328             RsltConst.m_bVal=Op1Const.m_iVal>Op2Const.m_iVal;
329             RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
330         };break;
331     case 45:           //enum >= enum
```



```
332     {
333         if (Op1Const.m_iEnumIdx!=Op2Const.m_iEnumIdx)
334         {
335             EmitError("枚举类型不兼容",TokenList.at(iListPos-1));
336             return false;
337         }
338         RsltConst.m_bVal=Op1Const.m_iVal>=Op2Const.m_iVal;
339         RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
340     };break;
341 case 46:         //enum < enum
342     {
343         if (Op1Const.m_iEnumIdx!=Op2Const.m_iEnumIdx)
344         {
345             EmitError("枚举类型不兼容",TokenList.at(iListPos-1));
346             return false;
347         }
348         RsltConst.m_bVal=Op1Const.m_iVal<Op2Const.m_iVal;
349         RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
350     };break;
351 case 47:         //enum <= enum
352     {
353         if (Op1Const.m_iEnumIdx!=Op2Const.m_iEnumIdx)
354         {
355             EmitError("枚举类型不兼容",TokenList.at(iListPos-1));
356             return false;
357         }
358         RsltConst.m_bVal=Op1Const.m_iVal<=Op2Const.m_iVal;
359         RsltConst.m_szName=RsltConst.m_bVal?"TRUE":"FALSE";
360     };break;
361     }
362 //分析得到计算结果的类型
363 switch ((StoreType)iReturnType)
364     {
365 case StoreType::T_BOOLEAN:RsltConst.m_ConstType=ConstType::BOOLEAN;break;
366 case StoreType::T_CHAR:
367 case StoreType::T_STRING:RsltConst.m_ConstType=ConstType::STRING;break;
368 case StoreType::T_REAL:
369 case StoreType::T_SINGLE:RsltConst.m_ConstType=ConstType::REAL;break;
370 default:RsltConst.m_ConstType=ConstType::INTEGER;RsltConst.m_iVal=RsltConst.m_fVal;break;
371     }
372 Rslt.m_iLink=SymbolTbl.RecConstTbl(RsltConst);           //申请临时常量信息
373 return true;
374 }
```

由于类型的存在，常量折叠的问题就有许多复杂变化。笔者将常量折叠的核心语义动作从 `OpConstSemantic` 函数中抽象出来了，即建立了 `OpConstFold` 函数。其目的就是为了使

OpConstFold 函数可以被优化阶段相关算法复用。虽然 OpConstFold 函数规模比较庞大，但是逻辑却比较简单。读者阅读相关源代码时，只需把握以下两个要点即可：

(1) 分支情景。了解分支设置的意义及其所处理的代码情景是理解源码的前提，笔者已作了详细的注释说明。

(2) 常量的形式。读者应该了解各种类型的常量在常量信息表中的表示形式，即 ConstInfo 对象各属性的取值情况，如表 6-5 所示，同时，还需注意以下两点：

1) 整型常量的值可以从 m_fVal 和 m_iVal 属性获取，这样可以使 OpConstFold 函数节省许多判断。

2) Pascal 语言中唯一的指针常量就是 nil。

表 6-5 常量的值域属性

类 型	相 关 属 性	类 型	相 关 属 性	类 型	相 关 属 性
整型	m_iVal、m_fVal	实型	m_fVal	set	m_szSet
boolean	m_bVal	enum	m_szName	string	m_szVal
char	m_szVal	pointer	m_szVal		

最后看看单目运算表达式的语义子程序 semantic052 的实现细节。

程序 6-9 Semantic.cpp

相关文法：

表达式 1 → 关系运算符 050 项 051 表达式 1 | ε
 项 1 → 低优先级运算符 050 因式 051 项 1 | ε
 因式 1 → 高优先级运算符 050 因子 051 因式 1 | ε

```

1  bool semantic052()
2  {
3      OpInfo Tmp1;
4      int TmpResult;
5      int TmpOperation;
6      if (!Operand.empty())
7          Tmp1=Operand.top();
8      else
9      {
10         EmitError("缺少操作数",TokenList.at(iListPos-1));
11         return false;
12     }
13     Operand.pop();
14     if (!Operation.empty())
15         TmpOperation=Operation.top();
16     else
17     {
18         EmitError("缺少操作数",TokenList.at(iListPos-1));
19         return false;
20     }

```



```
21     Operation.pop();
22     TmpResult=CType::TypeCompatible(Tmp1,TmpOperation);
23     if (TmpResult!=-1)
24     {
25         if (Tmp1.m_iType==OpInfo::CONST)
26         {
27             return OpConstSemantic(Tmp1,SymbolTbl.TypeSysTbl.at(TmpResult).
28                 m_iProcessId,SymbolTbl.TypeSysTbl.at(TmpResult).m_iRsltType);
29         }
30         else
31         {
32             OpInfo TmpRslt;
33             TmpRslt.m_iType=OpInfo::VAR;
34             TmpRslt.m_iLink=SymbolTbl.GetTmpVar(SymbolTbl.ProcStack.top(),
35                 (StoreType)SymbolTbl.TypeSysTbl.at(TmpResult).m_iRsltType);
36             OpVarSemantic(Tmp1,TmpRslt,SymbolTbl.TypeSysTbl.at(TmpResult));
37             Operand.push(TmpRslt);
38         }
39         return true;
40     }
41     else
42     {
43         EmitError("操作数类型不兼容",TokenList.at(iListPos-1));
44         return false;
45     }
46     return true;
47 }
```

semantic052 的主要工作就是完成单目运算表达式的翻译与处理。由于 semantic052 源代码的实现与 semantic051 是非常类似的，所以不再详细分析。这里，笔者只想谈一个关于单目运算符“+”（正号）的小问题。“+”是一个比较特殊的运算符，无论是表达式翻译，还是常量折叠，编译器都不需要为此作任何处理或生成任何 IR。当然，更不可以产生类似“+ -i 等于 i”的常识性错误。

至此，笔者已经详细介绍了表达式处理的相关实现，包括表达式翻译及常量折叠两个部分。表达式处理的程序并不庞大，结构也比较简单，应该不难理解。由于 Neo Pascal 的表达式处理完全是基于庞大的类型系统表实现的，所以读者应该紧密结合类型系统表阅读相关源代码，否则，一切都是没有意义的。类型系统的设计方案也并不唯一，类型系统表只是一种易于实现、便于理解的方式。而这种方式的缺点就是二维表的结构需要耗费一定存储空间，处理效率不高。对于一些类型系统复杂的语言来说，构建类型系统表可能是一项极其庞大与惊人的工程，这是 Neo Pascal 甚至 C 语言都无法比拟的。在这种情况下，就应该考虑采用一些更为高效的算法。

6.5 操作数翻译

6.5.1 操作数的地址与形态

本节将介绍语义处理的最后一个话题——操作数翻译。前面花费了较大的精力讲述表达式的翻译，以便读者了解如何将一个表达式翻译成相应的 IR 序列。不过，在此期间却略过了一个问题，就是语法中非终结符“变量”的相关语义子程序。换句话说，讲述表达式翻译时，讨论的前提就是表达式的操作数都是以 OpInfo 对象的形式存在于 Operand 栈内的。其中，笔者只分析了 semantic049 是如何生成常量操作数的 OpInfo 对象的，但并没有解释编译器是如何为其他操作数构造 OpInfo 对象的。然而，本节的重点就是研究编译器如何从输入源程序中捕捉相关信息，并为各种可能的情况构造 OpInfo 对象。这是一个非常复杂的过程，也是表达式翻译的精髓所在。

与常量操作数不同，当变量参与表达式运算时，由于操作数的值是在程序运行过程中确定的，因此，在绝大多数情况下，编译阶段是无法得到的。不过，这并不意味着编译器完全无能为力了。虽然变量的值不能在编译阶段确定，但是变量的地址在某种意义上却是可以确定的。注意，笔者只是将其界定为“在某种意义上”。在很多情况下，变量地址的管理和组织是由编译器与操作系统及目标机共同完成的。因此，编译阶段所能确定的地址只是变量的逻辑地址，并不是实际的物理地址。

在目标程序运行过程中，任何一个变量都需要占用一定的存储空间，即使是编译器自动分配的临时变量也不例外。由此，编译器必定会为每个变量分配一片逻辑地址空间，这片空间的容量是根据变量的实际类型而定的。这里，读者并不需要关注这些逻辑空间块的组织形式，可以将它们理解为是完全离散的，彼此并不存在任何联系。然而，对于操作数翻译而言，这些逻辑空间块的首地址显然是非常重要的，它是在运行时获取变量值的唯一途径。不过，在语义处理阶段，遗憾的是编译器可能无法得到这些逻辑空间块的首地址。那么，编译器又如何生成相关的寻址 IR 呢？最有效的解决方法就是为每个逻辑地址空间定义一个名字，将其与相应逻辑空间块的首地址关联。在语义处理阶段，一切对变量逻辑空间块首地址的访问都转换成对其名字的寻址即可。例如，(ADD, a, 1, b) 的含义是将 a 逻辑空间块中存储的值加 1，并保存到 b 逻辑空间块中。当然，在后续阶段，编译器最终会将 a、b 替换为相应逻辑空间块的首地址。

不过，仅有逻辑空间块的首地址还不足以应对任意形式的操作数。编译器还必须计算所需访问的空间在逻辑空间块内的偏移量。对于简单变量操作数来说，该偏移量当然就是 0。然而，当操作数是数组元素、结构字段等形式时，该偏移量就不一定是 0，例如，a.b、a[10]、a[i] 等。编译器通常将整个数组或结构作为一个逻辑空间块，而其中的元素只能表示为相对于逻辑空间块首地址的偏移量的形式。同样，块内偏移量也可以分为两种情况：常量偏移、变量偏移。常量偏移就是指该偏移量可以在编译阶段计算得到的，例如，a.b、a[10] 等。而变量偏移就是指该偏移量只能在运行阶段确定的，例如，a[i]、a[a[j]] 等。

了解了操作数地址的概念后，就来看看其形态设计。实际上，操作数的形态就是指 IR 中关于操作数的描述形式。换句话说，就是讨论如何运用 IR 序列表示操作数。在编译技术



中，关于操作数的形态并没有统一的观点，与 IR 设计类似，设计操作数的形态也是一项基于经验的工程。当然，其形态的抽象层次对于 IR 的级别也有较大影响。

鉴于 Neo Pascal 采用了中、低级别的 IR，在设计其操作数形态时，笔者主要考虑如下两个方面：

(1) 简单地址形式。前面提到了操作数偏移量的问题，无论是常量偏移，还是变量偏移，它们都是基于首地址而言的。然而，在通用机编译器中，逻辑空间块的首地址是不可能编译阶段确定的。在设计 IR 时，为了便于后续处理，笔者尽可能细化寻址过程的每个步骤，将其转换为相应的 IR 输出，见表 6-6。

表 6-6 简单地址形式

Pascal 声明	获取 a.b 字段值的 IR 序列
a:RECORD a,b:char; END;	(ASSIGN_4, 1, NULL, _T1) //计算常量偏移
	(GETADDR, a, NULL, _T2) //获取 a 的首地址
	(ADD_4, _T1, _T2, _T3) //计算 a.b 的地址
	(ASSIGN_4, @_T3, NULL, _T4) //获取 a.b 的值

不难发现，表 6-6 中操作数的形式都比较简单，关于操作数寻址的描述是以 IR 指令形式给出，而没有将操作数的抽象滞留在较高的层次上。这种方案的主要特点就是 IR 中的每个 OpInfo 对象的形态都是简单地址形式，不存在任何偏移信息。较低级的操作数地址的形态是非常有利于优化及代码生成的。虽然这个方案可能会产生较多的临时变量，但是这个结果却是可以接受的。因为编译器可以通过存储分配及代码优化等算法来减少临时变量的使用。例如，可以通过数据流的分析获得各个临时变量的使用周期（即定值点到引用点的范围），将使用周期互不重叠的临时变量分配在一个逻辑地址空间中，以节省临时变量的空间开销。在后续章节中将详细介绍相关设计思想及算法实现。

(2) 无类型支持。在讨论 IR 的抽象级别时，曾经谈过这个话题。实际上，关于 IR 是否提供类型信息，完全是由 IR 的抽象级别决定的，并不能一概而论。针对不同的代码优化与生成算法，编译器设计者可能需要设计几种形式、级别不同的 IR。实践证明，完备的类型信息反而会加大某些类型无关的优化算法的复杂程度。因此，在通常情况下，编译器设计者都会构造一种无类型的 IR 形式供后端使用。不过，这并不意味着无类型的 IR 必须在语义处理阶段生成。如果有特殊需要，在语义处理阶段，编译器完全可以生成有类型的较高级的 IR。随后，在特定的时刻，将其转换为其他形式的 IR 与无类型的 IR，这项工程应该并不复杂。

必须指出，任何 IR 形式的评价都是基于具体应用环境（如代码优化与生成等）的，脱离了应用环境讨论 IR 的优劣是没有意义的。

6.5.2 操作数翻译基础

与表达式翻译类似，操作数翻译同样是基于语法机制实现的。因此，操作数的形式将是设计语义子程序的重要依据。在本小节中，略过类型不谈，而从操作数的形式入手讨论一般化的操作数翻译。操作数相关的文法形式如下：

【文法 6-2】

因子	→ 变量 053
变量	→ 标识符 054 变量 1
变量 1	→ [098 表达式列表 099] 059 变量 1
	→ . 标识符 056 变量 1
	→ ^ 058 变量 1
	→ 过程调用语句
过程调用语句	→ (101 实参列表) 066
	→ ε 005

从文法来说，可以将操作数分为五种基本形式：

(1) 简单变量操作数：操作数中没有任何界符，也就是直接使用变量名来表示一个操作数，例如 `a`、`b` 等。注意，这里所说的“简单变量操作数”并不一定是简单类型的变量。在进行操作数翻译时，可能更多关注的是操作数的形式，而不是操作数的类型。这是因为语法推导的过程及语义子程序调用的顺序并不是由操作数的类型决定的，而是取决于操作数的形式。当然，这并不意味着操作数的类型没有意义。操作数的类型是语义子程序内部逻辑所关注的元素。简单变量操作数的语义处理比较容易，只需将 `OpInfo` 对象的 `m_iLink` 属性指向变量信息表中相应的表项，并正确设置其他相关属性即可。

(2) 记录字段操作数：操作数中包含界符“.”，也就是使用记录类型变量的某一个字段作为一个操作数，例如 `house.address`、`house.tel` 等。众所周知，记录字段相对于首地址的偏移是常量。当然，偏移的值可能会因为记录的存储布局而变化，甚至出现一些令人费解的情形。不过，有一点应该是无可争议的，那就是偏移的值必定是一个常量。当然，在处理记录字段操作数时，不能忽略多级字段操作数的形式，例如 `house.address.road`。根据实际语义，多级字段操作数的地址相对于记录类型变量首地址的偏量同样是一个常量。

(3) 数组元素操作数：操作数中包含界符“[]”，也就是使用数组类型变量的某一个元素作为一个操作数，例如 `source[i]`、`class[2]` 等。传统的数组都是顺序存储的，所以数组元素的地址通常可以被描述成相对于首地址的偏移。不过，与记录字段不同，这个偏移量可能是常量，也可能是变量，这是取决于数组元素的下标形式。数组元素操作数的语义处理比较复杂，由于可能存在变量偏移的情况，所以编译器就需要生成 IR 来计算相应的偏移。

(4) 指针目标操作数：操作数中包含界符“^”，也就是将指针指向的目标元素作为一个操作数，例如 `aa^`、`aa.bb^` 等。指针目标操作数本身并不需要处理偏移量。它的语义处理比较简单，实际上，就是一次间接寻址的过程，只需正确设置 `m_bRef` 属性即可。不过，在处理指针时，一定不能忽略左值的问题。在任何支持指针的语言中，指针运算的对象通常只能是左值变量。因此，左值判断是非常必要的。

(5) 函数返回值操作数：在 `Pascal`、`C` 等语言中，函数返回值也可以直接作为表达式的操作数。值得注意的是：标准 `Pascal` 规定，程序员调用过程或函数时，“()”只能在参数列表非空的情况下运用。也就是说，调用无参过程或函数时，只需引用过程或函数名即可。不过，随着“@”运算符的引入，就不得不解决一个二义性的问题。例如：

```
type
  f=function :integer;
```



```
var
    b:^integer;      c:f;
function a:integer;
begin
    .....
end;
begin
    b:=@a;    //取函数 a 的返回值的地址
    c:=@a;    //取函数 a 的地址
end.
```

这是一个合法的 Pascal 程序。主程序中两个赋值语句的右部都为@*a*，但它们的语义却是完全不同的。对于语义分析来说，这种情况是非常不利的。显然，表达式“@*a*”本身的语义并不唯一，它依赖于赋值号左边对象的类型。从语言实现的角度而言，这种机制也将给编译器设计带来不小的麻烦。实践证明，这种得不偿失的做法并没有得到大多数语言设计者的认可。在设计 C 语言时，Ritchie 很好地解决了这一问题。因为 C 语言的函数调用语句必须加“()”，即使无参函数也是如此。这样，就避免了类似的二义性问题。开源编译器 Free Pascal 就参考了 C 语言的实现，提出了类似的修改方案，规定调用过程或函数时必须加“()”。为了避免二义性，便于编译器的实现，Neo Pascal 同样采用了这个方案。

以上简单说明了五种操作数形式，这是讨论操作数语义处理及翻译的基础。关于操作数的概念、分类都没有非常明确的定论，所以笔者给出的操作数分类名仅适用于本书。另外，再次强调，在进行操作数语义处理时，区分操作数形式是比较重要的，因为翻译操作数的语义动作与其文法形式是密切相关的。了解了操作数的形式后，就来讨论操作数翻译的一般化过程。

读者已经知道了，操作数翻译与表达式翻译的接口就是 Operand 栈。前者借助于 Operand 栈向后者传递信息，而后者则根据表达式的语义生成相应的 IR。然而，这里主要关注的是生成 OpInfo 对象的过程。

在讨论翻译过程之前，有一个概念需要明确：即使语法分析器选用“因子→变量”这个产生式进行后续推导，也并不意味着当前标识符必定是变量名或者函数名。事实上，很多语言都提供了一种符号常量机制，例如 Pascal 中的“CONST”为首的相关声明。虽然符号常量的形式与简单变量比较类似，但是它的语义处理却与普通的常量无异。如果当前标识符是符号常量名，那么“[]”或“.”运算对于符号常量都是没有任何意义的。处理符号常量操作数的过程比较简单，编译器只需为其生成一个 OpInfo 对象，将相应的 *m_iType* 属性设为 OpInfo::Const，并压入 Operand 栈即可。

下面重点讨论变量操作数的语义处理过程。

处理变量操作数的关键就在于如何计算偏移量。针对不同形式的变量操作数，区别常量偏移、变量偏移是极其重要的。由于变量偏移依赖运行时刻的状态，所以，对于变量偏移来说，编译器只能通过生成相应的目标代码来实现偏移量的计算。至于如何生成更优的 IR 或目标代码，则完全是代码优化问题。

不过，常量偏移的情况就有些不同了。在语义分析过程中，常量偏移的计算完全是由编译器完成的。当然，生成精简且高效的 IR 就是编译器设计者关注的。这里，笔者先给出一

个简单的例子:

```
VAR
    house:RECORD
        name:    ^char;           //4 字节, 偏移为 0
        address:RECORD           //偏移为 4
            No:    integer;       //4 字节, 偏移为 4
            road:  ^char;         //4 字节, 偏移为 8
        END;
    END;
```

根据先前讨论的结论可知, `house.address.road` 的偏移必定是一个常量偏移。当然, 也可以非常容易地得到这个常量偏移的值即为 8。当运用相关文法推导“`house.address.road`”时, 就不难发现 `semantic056` 将被两次调用, 分别用于处理 `address` 与 `road` 字段。换句话说, 如果不考虑效率, 编译器可能得到如下的 IR 序列:

```
(GETADDR, house, NULL, _T1) //获得 house 的首地址
(ADD, _T1, 4, _T2) //分析 address 字段, 生成计算 address 偏移的 IR
(ADD, _T2, 4, _T3) //分析 road 字段, 生成计算 road 偏移的 IR
```

从语义上来说, 这样的 IR 序列是完全正确的, 确实也足以精确描述整个寻址的过程。不过, 这个结果却是不可接受的。实际上, 在本例中, 生成计算 `address` 字段偏移的 IR 完全没有意义。本例只需直接计算得到 `road` 字段的偏移量 8, 并生成相应的 IR 即可。当然, 有些读者认为编译器可以借助优化算法将后两句 IR 合并。虽然笔者也承认优化算法可以解决这个问题, 但并不推荐这种实现。理由非常简单, 任何优化算法都只能处理某种或某几种代码情况, 不可能以不变应万变, 应对任意代码情景。由于编译器自动生成 IR 的随意性较大, 设计一个能在任意代码情景中将后两句 IR 合并的算法可能需要付出较大的代价, 所以其可行性较差。然而, 在操作数翻译过程中, 要实现这个目标就比较容易了。

在前面例子中, 生成冗余的 IR 的主要原因就是没有完成常量偏移的折叠, 而是直接生成 IR 输出。解决这个问题的关键就是如何在语义分析过程中完成常量偏移的折叠。为了便于讲解, 笔者暂且将翻译操作数得到的 IR 序列分成两个部分: 偏移计算部分、操作数寻址部分。顾名思义, 偏移计算部分就是专门用于计算操作数偏移的 IR 序列。而操作数寻址部分就是指那些用于获取首地址及计算操作数逻辑地址的 IR 序列。

所谓的“操作数寻址部分”的 IR 序列主要指的就是两条指令, 即获取首地址的 IR 指令、首地址与偏移相加的 IR 指令。当然, 如果偏移为常量 0 时, 编译器就不需要生成第二条 IR 指令了。因此, 试图优化操作数寻址部分的 IR 可能是徒劳的。

然而, “偏移计算部分”的 IR 序列的情况就比较复杂了, 同时, 它也为代码的优化创造了条件。如果完全按照语义生成 IR, 就可能得到类似于上例中的冗余结果。实际上, 编译器在生成“偏移计算部分”时, 需要解决的核心问题就是常量偏移的折叠。

翻译操作数的基本思想大致如下: 可以将生成操作数寻址 IR 的动作置于 `semantic053` 中, 即在分析完整个操作数之后, 再生成操作数寻址 IR。因此, 编译器就不得不借助于一个暂存结构, 跟踪偏移量的变化情况, 并予以记录。在此过程中, 遵循常量偏移折叠的原则, 尽可能将常量偏移在暂存结构中合并, 直到出现变量偏移为止。当然, 自出现变量偏移之



后，就不必再考虑后续更复杂的常量偏移折叠的情况。例如，`b[i].a.p[3]`，实际上，“`.a.p[3]`”相对于“`b[i]`”也是一个常量偏移。不过，由于 `i` 是一个变量，所以 `b[i]` 相对于 `b` 就是一个变量偏移。基于 `b[i]` 继续讨论常量偏移折叠可能复杂一些，故后续出现的常量偏移折叠就不予考虑了。具体的源码实现，请读者参考后续章节。

最后，笔者给出一个重要的数据结构，就是先前提及的“暂存结构”。暂存结构的设计是比较重要的，必须既能满足操作数翻译的实际需要，结构也不能过于复杂。在 Neo Pascal 中，该暂存结构的声明形式如下：

【声明 6-7】

```
struct Var
{
    stack<VarType> m_VarTypeStack;
    int m_iVarLink;
    OffsetType m_eOffsetType;
    int m_iOffsetLink;
    int m_iDim;
    bool m_bRef;
    Var();
};
stack<Var> CurrentVar;
```

m_VarTypeStack：操作数类型栈，主要用于跟踪操作数分析过程中类型信息的变化情况。这是一个非常重要的字段，因为操作数类型分析的结果将直接影响编译的正确性。

m_iVarLink：所属变量信息。由于 `Var` 结构只处理变量操作数，所以有必要将所属变量的指针暂存。实际上，这个字段对于编译器获取变量首地址信息是非常有用的。

m_eOffsetType：操作数的偏移类型，这是一个枚举类型的字段。该枚举类型的允许取值分别为 `ConstOffset`（常量偏移）、`VarOffset`（变量偏移）、`NoneOffset`（无偏移）。

m_iOffsetLink：操作数的偏移指针。根据 `m_eOffsetType` 的取值，该字段的含义是不同的。当 `m_eOffsetType` 的值为 `ConstOffset` 时，`m_iOffsetLink` 中存储的是常量在常量信息表中的位序。当 `m_eOffsetType` 的值为 `VarOffset` 时，`m_iOffsetLink` 中存储的是变量在变量信息表中的位序。当 `m_eOffsetType` 的值为 `NoneOffset` 时，`m_iOffsetType` 是没有意义的。

m_iDim：数组维度信息。在分析当前操作数时，该字段主要用于记录当前正在分析数组的维度信息，详细实现将在后续章节中讨论。

m_bRef：引用寻址标志。

当然，在语义分析过程中，还需要考虑操作数嵌入引用的情况，例如，`a[b.c].p` 等。因此，暂存结构实际上是一个栈式结构，用于跟踪整个操作数的分析过程。

6.5.3 简单变量操作数的翻译

前面已经对“简单变量”的概念作了相关的说明。本小节将关注简单变量的语义处理动作的设计及其实现。那么，直接引用一个标识符来表示的操作数必定是简单变量操作数吗？答案是否定的，事实上，`with` 结构和过程调用语句是不可忽略的。下面对这两个话题作简单讨论。

with 结构是一种比较特殊的语法机制，其设置的目的是为了便于程序员对记录字段的引用。因此，在处理简单变量操作数时，需要区分通过 with 结构引用记录字段的情形。当然，从某种角度而言，with 结构可能降低程序的可读性，尤其是多层嵌套的 with 结构可能导致程序的结构比较零乱。因此，包括 C 语言在内的很多程序设计语言是不提供 with 结构的。不过，作为一个 Pascal 编译器，Neo Pascal 还是继承了标准 Pascal 的 with 结构。

先简单介绍一下 Pascal 语言的 with 结构。如图 6-8 所示，这是一个关于 with 结构的例子，程序中有 3 条赋值语句，不过，它们的赋值对象却是完全不同的。这是由 with 结构的优先引用原则决定的。在 with 结构中，当字段名与其他对象重名时，字段名将被优先引用，这和全局变量、局部变量重名时的处理原则是一致的。另外，当 with 结构嵌套时，在内、外层字段名重名的情况下，内层字段将屏蔽外层字段。设计者需要关注 with 结构的两个基本特性，它们是编译器设计的准则。虽然编译器设计者对 with 结构其他特性的理解可能存在一定的差异，但这两个基本原则似乎是不可突破的底线。

除了 with 结构之外，还需要考虑函数调用语句的情况。判断标识符是不是函数名应该并不复杂，只需遍历符号表即可。如果是函数调用，那么，编译器就应该生成相应的调用函数并获取返回值的 IR。在处理函数调用时，还有一个问题不得不处理，那就是函数调用与 with 结构的优先关系。从理论上来说，函数调用与 with 结构之间是不应该存在二义性的，即使存在重名的情况亦是如此。不过，遗憾的是，标准 Pascal 的语义却并非如此。图 6-8 所示的程序，包括 Delphi 在内的许多 Pascal 编译器都不支持。换句话说，在 with 结构中，当字段名与其他对象（包括过程、函数）重名时，字段名将被优先引用。

了解了 with 结构及函数调用语句后，也就明确了哪些标识符属于简单变量操作数的讨论范畴了。而那些属于记录字段或者函数名的标识符当然就由其他语义子程序处理。由于操作数翻译涉及 with 结构处理的相关数据结构——WithStack 栈，因此，笔者对此结构作简单说明。WithStack 栈的声明形式如下：

【声明 6-8】

```
struct WithField
{
    Var m_Var;           //开域记录信息
    int m_iRestoreIR;    //冗余 IR 的起始位置
};
stack<WithField> WithStack;
```

```
var
    a,b:record c,d: integer ;end;
    c: integer;
function d(i : integer) : integer;
begin
    result:=1;
end;
begin
    with a do
    begin
        c:=1;           // 优先引用 a.c
        with b do
        begin
            c:=2;       // 优先引用 b.c
            c:=d(1);    // 语义错误
        end;
    end;
    c:=3;               // 优先引用 c
end;
```

图 6-8 with 结构的优先级



对于一个 with 结构而言，最重要的信息应该就是开域记录的相关信息。从语法上而言，标准 Pascal 的开域记录可以是由“变量”推导得到的任何形式，所以其形式还是比较复杂的。那么，以什么数据结构暂存开域记录的相关信息（编译信息）呢？当然，使用 Var 结构来暂存开域记录的信息应该是不错的选择。相对于 OpInfo 结构而言，Var 结构可以提供更多有用的信息，而且便于被操作数翻译的相关语义子程序共享访问。通过 m_Var 字段，编译器可以轻松获得开域记录相关的类型信息，在 semantic054 中，就将涉及这方面的应用。当然，由于标准 Pascal 允许 with 结构嵌套声明，所以编译器必须借助于 WithStack 栈将每一层次开域记录的信息都予以保存。关于 with 结构的实现细节，将在后续章节中讨论。下面，再来看看简单变量操作数的相关文法：

【文法 6-3】

- | | | |
|--------------|---|------------------------------------------------------------------------------------------|
| 因子 | → | 变量 053 |
| 变量 | → | 标识符 054 变量 1 |
| 变量 1 | → | 过程调用语句 |
| 过程调用语句 | → | ϵ 005 |
| semantic054: | | 1. 输入标识符的有效性判断。
2. 构造 Var 对象，并根据输入标识符设置 m_iVarLink 字段。
3. 将 Var 对象压入 CurrentVar 栈。 |
| semantic053: | | 1. 生成取首地址 IR。
2. 生成寻址 IR，用于计算变量首地址与常量偏移的和。 |
| semantic005: | | 1. 处理函数调用的相关语义。 |

程序 6-10 semantic.cpp

相关文法：

变量 → 标识符 054 变量 1

```
1  bool semantic054()
2  {
3      Var Tmp;
4      string szTmp=TokenList.at(iListPos-1).m_szContent;
5      int j=WithStack.size()-1;
6      for (;j>=0;j--)
7      {
8          int i;
9          for(i=0;i<SymbolTbl.TypeInfoTbl[WithStack.c.at(j).m_Var.m_VarTypeStack.top()
10             .m_iLink].m_FieldInfo.size();i++)
11          {
12              if (SymbolTbl.TypeInfoTbl[WithStack.c.at(j).m_Var.m_VarTypeStack.top()
13                 .m_iLink].m_FieldInfo.at(i).m_szName.compare(szTmp)==0)
14                  break;
15          }
16          if (i!=SymbolTbl.TypeInfoTbl[WithStack.c.at(j).m_Var.m_VarTypeStack.top()
17             .m_iLink].m_FieldInfo.size())
18              break;
19      }
20      if (j>=0)
```

```
21 {
22     CurrentVar.push(WithStack.c.at(j).m_Var);
23     return semantic056();
24 }
25 int i=i=SymbolTbl.SearchProcInfoTbl(szTmp,true);
26 if (i!=-1)
27 {
28     bool bConstProcess;
29     i=SymbolTbl.SearchVarInfoTbl(SymbolTbl.ProcStack.top(),szTmp);
30     if (i!=-1)
31     {
32         i=SymbolTbl.SearchConstInfoTbl(SymbolTbl.ProcStack.top(),szTmp);
33         if (i!=-1)
34         {
35             i=SymbolTbl.SearchVarInfoTbl(0,szTmp);
36             if (i!=-1)
37             {
38                 i=SymbolTbl.SearchConstInfoTbl(0,szTmp);
39                 if (i!=-1)
40                 {
41                     if (szTmp.compare("SIZEOF")==0)
42                     {
43                         ProcCall Tmp;
44                         Tmp.m_iProcId=-1;
45                         CurrentProcCall.push(Tmp);
46                         return true;
47                     }
48                     else
49                     {
50                         EmitError("标识符未声明或非法引用"
51                                 ,TokenList.at(iListPos-1));
52                         return false;
53                     }
54                 }
55             }
56             else
57                 bConstProcess=true;
58         }
59         else
60             bConstProcess=false;
61     }
62     else
63         bConstProcess=true;
64 }
65     bConstProcess=false;
66     if (bConstProcess)
```




```
67     {
68         OpInfo Tmp;
69         Tmp.m_iType=OpInfo::CONST;
70         Tmp.m_iLink=i;
71         Operand.push(Tmp);
72         bConstFlag=true;
73     }
74     else
75     {
76         PushVarType(SymbolTbl.VarInfoTbl[i].m_iTypeLink,Tmp);
77         Tmp.m_iVarLink=i;
78         Tmp.m_iDim=0;
79         Tmp.m_bRef=SymbolTbl.VarInfoTbl[i].m_bRef;
80         CurrentVar.push(Tmp);
81         bConstFlag=false;
82     }
83 }
84 else
85 {
86     ProcCall Tmp;
87     OpInfo TmpRet;
88     TmpRet.m_iType=OpInfo::VAR;
89     TmpRet.m_iLink=SymbolTbl.ProcInfoTbl.at(i).m_iReturnVar;
90     Tmp.m_Return=TmpRet;
91     Tmp.m_iProcId=i;
92     Tmp.m_eCallType=ProcCall::Call;
93     CurrentProcCall.push(Tmp);
94 }
95 return true;
96 }
```

第 5~19 行：遍历 WithStack 栈，判断当前输入标识符是否为开域字段名。根据标准 Pascal 的语义，编译器将由内向外逐层检索，即从 WithStack 栈顶向栈底逐一检索各个开域记录的字段列表。

第 20~24 行：如果当前输入标识符是开域字段名，则调用 semantic056 完成相应的语义动作。这里，先不必深究第 22 行、第 23 行的功能，笔者将在后续章节中详述。

第 25 行：检索过程信息表，判断当前输入标识符是否为过程名。

第 26~83 行：由于当前输入标识符不是过程名，即为常量符号名或变量符号名。因此，语义子程序主要完成两项工作：

(1) 确定输入标识符是常量符号名还是变量符号名。同时，还需要确定该符号是全局符号还是局部符号。由于同一过程或函数内的符号常量名与变量名是不允许重名的，因此检索的先后顺序并没有本质差异。不过，根据全局符号与局部符号重名的相关规范，编译器应优先检索当前过程所属的局部符号列表。这个检索顺序是由语言规范确定的，编译器设计者不可随意改变。第 29~65 行的代码主要就是完成这项工作。

(2) 根据当前输入标识符的性质，完成相应的语义动作。

如果该标识符是常量符号名，则直接生成一个 `OpInfo` 对象，设置 `m_iType` 属性为 `CONST`，并设置 `m_iLink` 属性。最后，将其压入 `Operand` 栈。注意，常量符号与普通常量类似，无需借助于 `Var` 对象完成操作数翻译。这里，值得注意一点，第 72 行将 `bConstFlag` 标志设为 `true` 是比较重要的，有助于后续语义子程序的合法性判断。通常，程序员试图对一个常量符号进行 “[]”、“.” 或 “^” 运算都是没有意义的。不过，根据文法来说，程序员却可以书写类似的表达式。然而，因为常量符号并不生成 `Var` 对象，所以编译器试图在语义子程序中发现与解决这些问题也并不容易。而 `bConstFlag` 标志就可以轻松实现这一功能，它的主要思想就是略过非终结符“变量”推导过程中执行的语义子程序，直到 `semantic053` 为止。

如果该标识符是变量名，根据先前讨论的原则，生成 `Var` 对象暂存变量的相关信息。第 76~79 行代码就是用于设置 `Tmp` 的各个属性。其中，调用了 `PushVarType` 函数将变量的类型信息压入了 `Tmp.m_VarTypeStack` 栈。第 80 行代码将 `Tmp` 压入 `CurrentVar` 栈。

第 85~94 行：这段代码主要用于处理输入标识符为过程名时的相关语义动作。与操作数翻译类似，过程调用的翻译同样需要栈结构的支持，因为编译器不得不考虑某一函数返回值出现在另一个过程（函数）调用语句的实参列表中的情况，例如，`a(b(2))`。因此，笔者设置了一个 `ProcCall` 结构及一个 `CurrentProcCall` 栈用于处理过程（函数）调用的语句的翻译。在操作数翻译中，主要涉及一些将过程相关信息保存到 `ProcCall` 对象中的语义处理动作。在此，笔者不打算详细解释各字段属性的含义与作用，读者只需大概了解该程序段的作用即可。过程（函数）调用的处理将在后续章节中详细讨论。

前面已经分析了 `semantic054` 的实现细节。下面看看 `semantic053` 的相关源代码实现。`semantic053` 是操作数翻译的核心语义子程序，它的主要工作就是根据 `Var` 对象的相关信息，生成操作数寻址 `IR`。

程序 6-11 semantic.cpp

相关文法：

因子 → 变量 053

```
1 bool semantic053()
2 {
3     if (bConstFlag)
4     {
5         bConstFlag=false;
6         return true;
7     }
8     if (CurrentVar.empty())
9     {
10        return true;
11    }
12    if (bWithFlag)
13    {
14        TmpWithField.m_Var=CurrentVar.top();
15        TmpWithField.m_iRestoreIR=SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top())
16        .m_Codes.size()-1;
```



```
17     }
18     OpInfo Tmp;
19     Tmp.m_iType=OpInfo::VAR;
20     Tmp.m_iLink=CurrentVar.top().m_iVarLink;
21     Tmp.m_iDetailType=CurrentVar.top().m_VarTypeStack;
22     if (CurrentVar.top().m_VarTypeStack.top().m_StoreType==StoreType::T_POINTER)
23         Tmp.m_bRef=false;
24     else
25         Tmp.m_bRef=CurrentVar.top().m_bRef;
26     if (CType::IsOffsetVar(CurrentVar.top().m_VarTypeStack) &&
27         CurrentVar.top().m_eOffsetType!=OffsetType::NoneOffset)
28     {
29         OpInfo Op1,Rslt;
30         int iConstOffset=0;
31         for(vector<OffsetStruct>::iterator it=CurrentVar.top().m_OffsetVec.begin();
32             it!=CurrentVar.top().m_OffsetVec.end();it++)
33         {
34             if (it->m_eOffsetType==OffsetType::ConstOffset)
35             {
36                 iConstOffset+=SymbolTbl.ConstInfoTbl[it->m_iOffsetLink].m_iVal;
37             }
38         }
39         if (CurrentVar.top().m_eOffsetType==OffsetType::ConstOffset)
40         {
41             Op1.m_iType=OpInfo::CONST;
42             iConstOffset+=SymbolTbl.ConstInfoTbl[CurrentVar.top()
43                 .m_iOffsetLink].m_iVal;
44             Op1.m_iLink=SymbolTbl.RecConstTbl(itos(iConstOffset),3);
45             iConstOffset=0;
46         }
47         else
48         {
49             Op1.m_iType=OpInfo::VAR;
50             Op1.m_iLink=CurrentVar.top().m_iOffsetLink;
51         }
52         Rslt.m_iType=OpInfo::VAR;
53         Rslt.m_bRef=false;
54         Rslt.m_iDetailType=Tmp.m_iDetailType;
55         Rslt.m_iLink=SymbolTbl.GetTmpVar(SymbolTbl.ProcStack.top(),StoreType::T_POINTER);
56         if (!SymbolTbl.IsTmpVar(Tmp.m_iLink))
57         {
58             if (!SymbolTbl.IsVarPara(SymbolTbl.VarInfoTbl[Tmp.m_iLink].m_szName
59                 ,SymbolTbl.VarInfoTbl[Tmp.m_iLink].m_iProcIndex) &&
60                 CurrentVar.top().m_bRef==false)
61             {
62                 OpInfo TmpRslt=Rslt;
```

```
63         TmpRslt.m_iLink=SymbolTbl.GetTmpVar(SymbolTbl.ProcStack.top()
64             ,StoreType::T_POINTER);
65         SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes
66             .push_back(EmitIR(OpType::GETADDR,Tmp,TmpRslt));
67         Tmp=TmpRslt;
68     }
69     else
70     {
71         Tmp.m_bRef=false;
72     }
73 }
74 SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes
75     .push_back(EmitIR(OpType::ADD_4,Tmp,Op1,Rslt));
76 if (iConstOffset!=0)
77 {
78     Op1.m_iType=OpInfo::CONST;
79     Op1.m_iLink=SymbolTbl.RecConstTbl(itos(iConstOffset),3);
80     SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes
81         .push_back(EmitIR(OpType::ADD_4,Rslt,Op1,Rslt));
82 }
83 for(vector<OffsetStruct>::iterator it=CurrentVar.top().m_OffsetVec.begin()
84     ;it!=CurrentVar.top().m_OffsetVec.end();it++)
85 {
86     if (it->m_eOffsetType==OffsetType::VarOffset)
87     {
88         Op1.m_iType=OpInfo::VAR;
89         Op1.m_bRef=false;
90         Op1.m_iLink=it->m_iOffsetLink;
91         SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes
92             .push_back(EmitIR(OpType::ADD_4,Rslt,Op1,Rslt));
93     }
94 }
95 Rslt.m_bRef!=(CurrentVar.top().m_VarTypeStack.top().m_StoreType==StoreType::T_POINTER);
96 Tmp=Rslt;
97 }
98 Operand.push(Tmp);
99 CurrentVar.pop();
100 return true;
101 }
```

第 3~7 行：根据 `bConstFlag` 标志，判断是否需要完成后续工作。`semantic053` 是用于生成变量操作数的寻址 IR 的，对于常量符号是没有任何意义的，必须略过。`semantic053` 是非终结符“变量”推导过程中的最后一个语义子程序，因此，在第 5 行中，将 `bConstFlag` 设回初始状态 `false`。实际上，当输入标识符为常量符号名时，“变量”推导过程中的所有语义子程序都将被略过，直至执行完 `semantic053` 后。至此，读者应该能够理解 `bConstFlag` 标志的意义了。



第 12~17 行：根据 `bWithFlag` 标志，完成 `with` 结构的相关语义处理。

第 18~25 行：根据 `CurrentVar` 栈顶元素，即当前操作数的 `Var` 信息，生成 `OpInfo` 对象。实际上，也就是将变量的首地址打包成一个 `OpInfo` 对象。这里，值得注意的是指针变量的特殊语义处理。

第 26、27 行：判断当前操作数是否存在偏移信息。注意，根据当前操作数的 `Var` 信息，判断偏移信息是否有效。遵守如下两条判断标准：

(1) `m_eOffsetType` 属性不为 `NoneOffset`。这里，可能并不需要关心常量偏移或变量偏移，而只需界定是否存在偏移即可。

(2) `m_VarTypeStack` 栈中是否包含记录类型或数组类型描述。只有数组元素或记录字段可能存在偏移信息，其他数据类型根本不需要关注该属性。因此，仅当用于跟踪类型变化的 `m_VarTypeStack` 栈中存在记录类型或数组类型描述信息时，操作数的偏移信息才可能是有效的。当然，这里并没有强调该类信息的详细位置，而只关注其存在与否。

第 28~97 行：根据当前操作数的偏移信息，生成操作数寻址 IR，这是 `semantic053` 的核心代码段。简而言之，操作数寻址部分的 IR 主要由以下两部分组成：

(1) 获取符号首地址的 IR 指令 (`GETADDR`)。获取首地址 IR 指令的操作数是输入程序的某个数组或记录对象，其结果是一个存储地址的临时指针变量。

(2) 叠加偏移的 IR 指令 (`ADD_4`)。无论是常量偏移还是变量偏移，只要存在偏移，就需要生成加法指令来计算首地址与偏移的和。

例如，`house.address.road` 操作数相应的 IR 序列如下：

```
(GETADDR, house, NULL, _T1) //获得 house 的首地址
(ADD_4, _T1, 8, _T1) //分析 road 字段，生成计算 road 偏移的 IR
```

操作数翻译与表达式翻译的接口就是 `Operand` 栈，两者是通过 `Operand` 栈传递数据的。前者将得到的操作数压入 `Operand` 栈。而后者根据运算符的目数，从 `Operand` 栈弹出操作数，生成相应的 IR，并将结果操作数再次压入 `Operand` 栈，以备后用。根据前面的分析，不难发现，将操作数压栈的动作将由 `semantic053` 完成。这里，值得注意的是无偏移量的操作数与有偏移量的操作数的处理是不尽相同的。以先前的“`_T1`”为例，根据 IR 的语义，临时变量 `_T1` 中存储的值是一个地址 (`house` 首地址+8)。按照无偏移量的操作数的处理方式，通常是将“`_T1`”直接压入 `Operand` 栈。实际上，这种做法是不正确的。

以 `i:=house.address.road-1` 为例，编译器可能得到如下 IR 序列：

```
(GETADDR, house, NULL, _T1) //获得 house 的首地址
(ADD_4, _T1, 8, _T1) //分析 road 字段，生成计算 road 偏移的 IR
//将 _T1 直接压入 Operand 栈
(SUB_4, _T1, 1, i) //计算 i:=house.address.road-1
```

非常明显，第 3 句 IR 的含义是将临时变量 `_T1` 的值减 1 后赋给 `i`。不过，由于 `_T1` 中的值是一个地址，这个结果并不是用户所预期的。实际上，从输入源程序的语义来说，期望得到的是 `_T1` 所示地址指向的存储空间内的值，换句话说，就是需要对 `_T1` 进行一次间接寻址后再访问。因此，对于编译器而言，区分直接寻址访问或间接寻址访问是非常重要的。同一个操作数在不同 IR 中的访问方式也可能是不同的。在上例中，前两句 IR 中的 `_T1` 都

是直接寻址访问，而最后一句 IR 中的 `_T1` 应该是间接寻址访问。在 Neo Pascal 中，笔者使用 `m_bRef` 来标识 IR 操作数的寻址访问方式，`m_bRef` 为 `true` 表示该操作数为间接寻址访问，反之则表示该操作数为直接寻址访问。在生成 IR 序列时，设计者应该谨慎处理 `m_bRef` 标志，否则将导致程序异常。在本书的 IR 文本中，使用“@”符号表示间接寻址访问，例如：

```
(SUB_4,  @_T1,  1,      i )
```

即表示将临时变量 `_T1` 的内容作为地址寻址得到的值减 1 后送入 `i`。读者必须注意与标准 Pascal 的“@”运算符的区别。

这里，讨论了一种具有通用意义的操作数寻址方式，使用这种方式的前提就是对象的物理地址在编译阶段是不可知的，其物理地址是由操作系统分配的。不过，假设在某些特定的机器环境下，物理地址（首地址）在编译阶段是已知的，那么，也可以通过更简洁的 IR 序列实现同样的语义。事实上，这种假设并非无稽之谈，一些嵌入式系统并没有真正意义的操作系统，其程序空间的分配都是在编译阶段完成的。在这种情况下，物理地址的分配通常是由链接器完成的，而不是由编译器本身直接完成的。至少在语义处理阶段，物理地址仍然是未知的。因此，在类似的系统结构下，如何生成更为简洁的 IR 序列（略去取首地址的 IR 语句）是一个值得深入研究的问题。关于这个话题，笔者就不再深入阐述。了解了操作数寻址 IR 的处理流程后，阅读与理解相关源码并不复杂。下面，针对 Neo Pascal 的源代码，笔者谈三个实现细节：

（1）临时变量的取地址。通常，取地址运算（GETADDR）只可能在两种情形下出现：

1、翻译含有“@”运算符的表达式。2、获取复杂类型变量的首地址。事实上，这两种情形都不可能实现对临时变量进行取地址运算。不过，有一种特殊的情形是值得注意的，那就是对复杂变量的某个元素取地址，例如，`@a[1]`、`@(house.road)`等。这里，以`@(house.road)`为例，按照先前讨论的翻译规程，编译器将产生如下 IR 序列：

```
(GETADDR,  house,  NULL,  _T1)  //获得 house 的首地址
(ADD_4,    _T1,    8,     _T1)  //分析 road 字段，生成计算 road 偏移的 IR
(GETADDR,  @_T1,   NULL,  _T2)  //获得@_T1 的地址，赋给_T2
```

注意，IR 序列中“`@_T1`”表示 `_T1` 是间接寻址操作数。从语义上来说，第 3 行 IR 的翻译并没有错，它的含义是将 `_T1` 值所指示的存储空间的地址赋给 `_T2`。不过，这种翻译方式是非常繁琐的。事实上，`_T1` 中存放的数据本身就是 `house.road` 的地址，根本不需要进行间接寻址后再取地址。理想的方案是将第 3 行 IR 替换成如下形式：

```
(ASSIGN_4,  _T1,  1,      _T2)  //将_T1 的地址赋给_T2
```

（2）变参的取地址。变参是一种比较特殊的传参方式。其本质就是将实参的地址传递给形参。不过，对于程序员而言，这是完全透明的。在引用形参时，并不需要对形参作“^”运算，这个过程是由编译器处理的。因此，对形参取地址时，编译器只需直接将形参的值赋给目标对象即可，并不需要生成取地址相关的 IR。

（3）叠加 `m_OffsetVec` 向量所存储的常量偏移。在此，笔者暂且不介绍 `m_OffsetVec` 向量的作用，读者只需了解 `m_OffsetVec` 中的常量同样作为偏移处理即可。关于 `m_OffsetVec`

向量，将在后续章节中详解。在本例中，第 31~38 行即用于处理 `m_OffsetVec` 向量。

第 98 行：将结果操作数压入 `Operand` 栈，供表达式翻译相关语义子程序使用。

第 99 行：至此，一个完整的操作数就分析完了，故将 `CurrentVar` 栈顶元素弹出。

6.5.4 记录字段操作数的翻译

记录是由若干已知类型的数据元素组合而成的一种复合数据类型，C 语言中称之为“结构”。众所周知，记录类型有一个非常重要的特点，就是记录变量的字段（分量）是连续存储的。因此，根据操作数的字段名，编译器是可以得到相应的常量偏移的。如图 6-9 所示。

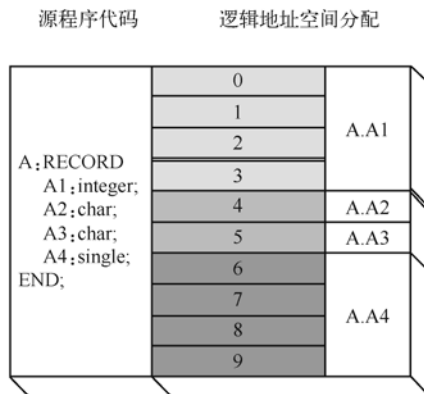


图 6-9 记录存储分配示意图（未经过数据字对齐处理）

这是记录类型变量 `A` 的标准存储形式，从图中可以看到编译器是按字段声明的先后顺序，依次组织各字段数据的。由此，编译器就可以根据符号表相关记录计算得到一个相对于逻辑空间块首地址的常量偏移。实际上，上图所示并不一定是记录类型变量 `A` 的实际存储形式。在存储分配时，根据字段的类型信息，分配算法将得到一个更合理的分配方案，未必是简单地按字段声明先后来组织字段的存储结构的。存储分配是一个比较复杂的话题，笔者将在后续章节中详细讨论。不过，无论记录类型变量的内部组织形式如何，有一个事实是不可否认的，那就是字段相对于记录变量首地址的偏移在编译阶段是已知的常量。迄今为止，绝大多数程序设计语言仍然是遵守这个约定的。虽然一些新型语言对传统的记录类型的理解或实现可能存在一定的差异，但始终没有突破这一底线。

在记录类型的字段信息表中专门有一个属性 (`m_iOffset`) 用于描述字段相对于记录变量首地址的偏移。这里，读者并不需要关心编译器如何计算 `m_iOffset` 属性，只要知道如何应用即可。下面，先来看看记录字段操作数的相关文法：

【文法 6-4】

- 因子 → 变量 053
- 变量 → 标识符 054 变量 1
- 变量 1 → . 标识符 056 变量 1

笔者已经详细分析了 `semantic053` 与 `semantic054` 的实现。由于简单变量操作数并不涉及偏移量的问题，所以不需要生成计算偏移的 IR 语句。然而，本小节将更多关注操作数偏移

的相关话题，这是因为记录字段操作数是需要偏移信息的。实际上，操作数偏移的问题远比操作数寻址复杂得多，针对不同的代码情景，编译器设计者必须做非常周详的考虑，否则很难保证编译的正确性。例如， $a[i].p$ 、 $a[1]^p$ 或者 a^p 等形式的处理。从语法上来说，不难发现，处理操作数偏移的动作就是由 semantic056 完成的。因此，详细分析 semantic056 的实现对于理解操作数偏移处理的基本思想是非常重要的。

根据当前操作数信息（即 CurrentVar 的栈顶元素），通常分为如下三种情形处理：

(1) 当前操作数没有偏移，即 CurrentVar 栈顶元素的 m_eOffsetType 值为 NoneOffset。由于记录字段的偏移是常量偏移，所以只需将常量偏移登记在当前操作数信息中即可，以便 semantic053 统一处理。

(2) 当前操作数的偏移类型为常量偏移，即 CurrentVar 的栈顶元素的 m_eOffsetType 值为 ConstOffset。由于当前操作数的原偏移与当前记录字段的偏移都是常量偏移，那么，编译器就必须将两个常量偏移相加折叠。

(3) 当前操作数的偏移类型为变量偏移，即 CurrentVar 的栈顶元素的 m_eOffsetType 值为 VarOffset。在这种情况下，不存在常量折叠的可能性，所以编译器只能生成相应的 IR 语句，以完成偏移值的累加。

程序 6-12 semantic.cpp

相关文法：

变量 1 → . 标识符 056 变量 1

```
1  bool semantic056()
2  {
3      if (bConstFlag || CurrentVar.empty())
4      {
5          EmitError("语义错误",TokenList.at(iListPos-1));
6          return false;
7      }
8      Var TmpVar;
9      string szTmp=TokenList.at(iListPos-1).m_szContent;
10     TmpVar=CurrentVar.top();
11     int i;
12     for(i=0;i<SymbolTbl.TypeInfoTbl[TmpVar.m_VarTypeStack.top()
13         .m_iLink].m_FieldInfo.size();i++)
14     {
15         if (SymbolTbl.TypeInfoTbl[TmpVar.m_VarTypeStack.top().m_iLink]
16             .m_FieldInfo.at(i).m_szName.compare(szTmp)==0)
17             break;
18     }
19     if (i!=SymbolTbl.TypeInfoTbl[TmpVar.m_VarTypeStack.top()
20         .m_iLink].m_FieldInfo.size())
21     {
22         FieldInfo *pFieldInfo=&(SymbolTbl.TypeInfoTbl[TmpVar.m_VarTypeStack
23             .top().m_iLink].m_FieldInfo.at(i));
24         int iTmpOffset=pFieldInfo->m_iOffset;
25         OffsetType TmpOffsetType=OffsetType::ConstOffset;
```




```
26     if (CurrentVar.top().m_eOffsetType==OffsetType::NoneOffset)
27     {
28         iTmpOffset=SymbolTbl.RecConstTbl(itos(iTmpOffset),3);
29     }
30     if (CurrentVar.top().m_eOffsetType==OffsetType::ConstOffset)
31     {
32         iTmpOffset=SymbolTbl.RecConstTbl(itos(iTmpOffset+SymbolTbl.ConstInfoTbl
33             [CurrentVar.top().m_iOffsetLink].m_iVal),3);
34     }
35     if (CurrentVar.top().m_eOffsetType==OffsetType::VarOffset)
36     {
37         TmpOffsetType=OffsetType::VarOffset;
38         OpInfo Op1,Op2,Rslt;
39         Op1.m_iType=OpInfo::VAR;
40         Op1.m_iLink=CurrentVar.top().m_iOffsetLink;
41         Op2.m_iType=OpInfo::CONST;
42         Op2.m_iLink=SymbolTbl.RecConstTbl(itos(iTmpOffset),3);
43         Rslt.m_iType=OpInfo::VAR;
44         Rslt.m_iLink=SymbolTbl.GetTmpVar(SymbolTbl.ProcStack.top()
45             ,StoreType::T_INTEGER);
46         iTmpOffset=Rslt.m_iLink;
47         if (!GenIR(Op1,Op2,Rslt,BasicOpType::ADD))
48             return false;
49     }
50     TmpVar.m_eOffsetType=TmpOffsetType;
51     TmpVar.m_iOffsetLink=iTmpOffset;
52     TmpVar.m_VarTypeStack.push(VarType(SymbolTbl.TypeInfoTbl[CType::GetRealType
53         (pFieldInfo->m_iLink)].m_eDataType
54         ,CType::GetRealType(pFieldInfo->m_iLink)));
55     CurrentVar.pop();
56     CurrentVar.push(TmpVar);
57     return true;
58 }
59 else
60 {
61     EmitError("字段未声明",TokenList.at(iListPos-1));
62     return false;
63 }
64 }
```

第 3~7 行：如果当前主符号是常量符号，那么，对常量进行“.”运算是无意义的。

第 12~18 行：遍历当前操作数的字段列表，以获取相应的字段信息。编译器以输入标识符为关键字检索当前操作数所属记录类型的字段列表，以此判断输入标识符的有效性。

第 19~20 行：输入标识符的有效性检查。根据文法形式，输入标识符必定是当前操作数所属记录类型的某一字段，否则即存在语义错误。

第 22~24 行：根据检索结果，令 pFieldInfo 指向相应字段，以便后续处理。同时，获

取该字段的偏移信息。

第 26~29 行：如果当前操作数的偏移类型为无偏移，则只需直接修改当前操作数相关属性即可。

第 30~34 行：如果当前操作数的偏移类型为常量偏移，则完成常量偏移的折叠。也就是用两个常量偏移的和生成一个新的常量，并将该常量的位序信息存入 `iTmpOffset` 变量中，以备后用。注意，`iTmpOffset` 变量就是用于存储操作数偏移的位序信息，并不严格区分常量信息表或变量信息表，而是视结果偏移信息的实际情况而定的。

第 35~49 行：如果当前操作数的偏移类型为变量偏移，则直接生成计算偏移的 IR。实际上，就是生成一条操作码为“ADD_4”的 IR 语句，将当前操作数的偏移与当前字段的常量偏移相加后暂存于一个临时变量中。这里，分别声明了三个 `OpInfo` 对象，即 `Op1`、`Op2`、`Rslt`，主要用于表示当前操作数的偏移、字段的常量偏移及结果操作数。当然，在生成 IR 后，必须注意 `Operand` 栈的维护。

第 50~56 行：生成一个 `Var` 对象，设置完相应的属性后压入 `Operand` 栈。值得注意的是 `m_VarTypeStack` 栈的设置。`m_VarTypeStack` 栈主要是用于跟踪操作数类型的变化情况，并予以记录。其中，栈顶元素描述的就是当前操作数的类型信息。实际上，在调用 `semantic056` 之前，当前操作数是一个记录类型变量。然而，在调用以后，当前操作数就是一个记录字段变量。为了跟踪类型变化的过程，通常会将当前操作数的最新类型信息压入 `m_VarTypeStack` 栈，而不是简单地覆盖原始类型信息。

至此，笔者已经分析了记录字段操作数的翻译与处理过程。其中，常量偏移的折叠是关键所在，了解了常量偏移的处理对于理解后续章节是相当有用的。事实上，严格来说，常量偏移的折叠是常量折叠的一种特例，也是 IR 优化讨论的范畴。在语义处理中，完全可以将其忽略，直至 IR 优化再作处理。不过，笔者却不认为这是一个值得提倡的做法。在设计一些类似于编译器之类的复杂系统时，应该尽可能避免将即时出现的问题延后到后期处理，因为期间可能发生的变化有时是很难预见的。虽然 IR 优化作为单独的一遍存在于整个编译过程中，但是，并不意味着必须把所有的优化问题集中到这一点上处理。笔者的观点是适时而动，即在最有利于解决某一问题的时刻处理该问题是最优的选择，而不必过多拘泥于模块结构的限制。在 C 编译器中，逻辑表达式的短路问题亦是如此。当然，也可以将短路问题视作 IR 优化的范畴，不过，可能因此而付出巨大的代价。

6.5.5 数组翻译基础

相比记录类型而言，数组可能更深入人心。笔者相信可能存在不支持记录类型的语言，却不可能存在不支持数组的语言。关于数组的基本概念，似乎已经没有阐述的必要了。这里，笔者将从编译器实现的角度来深入剖析数组的特性，让读者从一个崭新的视角来重新审视这种最通用的构造类型。

1. 数组的存储布局

数组是一种连续存储的数据结构，用户通常以下标形式存取数组的元素。那么，下标与数组元素逻辑地址的映射关系就是编译器所关心的。然而，众所周知，物理存储区都是线性编址的。数组的存储布局主要就是讨论各种形态的数组在线性物理存储区中的组织形式，这是将下标引用转换为逻辑地址的基础。由于一维数组的形态本身就是线性结构，其存储布局

与本身的形态基本一致，故不再详述。这里，先以二维数组为例，引入数组存储布局的相关问题。然后，再将其结论推广到更一般的多维数组中。

二维数组的映射方案主要有如下三个：行优先映射、列优先映射、间接向量映射。对于读者来说，前两种方案应该并不陌生。几乎任何一个程序设计课程都会对此作相关说明，因此，笔者就不再赘述了。较之前两者，在过去很长一段时期内，间接向量映射的概念却并没有得到普及，直至 Java 的出现。

所谓“间接向量映射”就是使用间接向量把所有多维数组压缩在一个向量集合中。以一个普通的二维数组为例，采用间接向量映射的存储布局如图 6-10 所示。

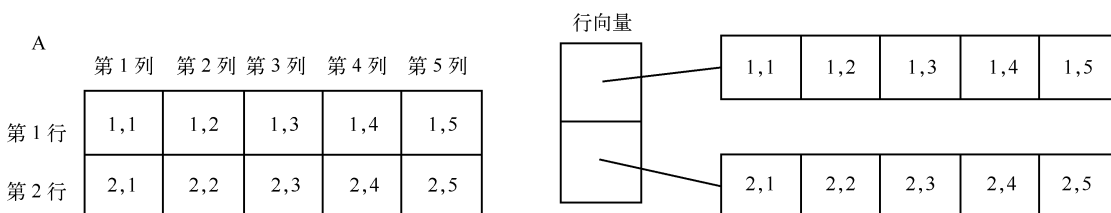


图 6-10 间接向量映射的存储布局

仔细观察图 6-10，不难发现这种存储布局的特点就是每一行元素是连续存储的，行之间是完全离散的。编译器借助一个行向量完成行寻址操作，而行内元素的寻址与一维数组完全一样。同样，间接向量映射也存在行优先或列优先之分。

从存储布局来看，这带来了两个新问题。

(1) 增加了间接向量的空间开销。传统行（列）优先映射并不需要借助间接向量引用元素，而这种方案却不得不借助间接向量来完成元素寻址。事实上，随着数组维数的增大，间接向量的开销是非常可观的。以一个 $3 \times 3 \times 4$ 的数组为例，将多消耗 9 个存储空间用于存储间接向量。

(2) 建立间接向量需要大量的初始化代码，用于处理所有间接向量内部的指针。

当然，间接向量映射方式也有其自身的魅力，是传统映射方案无法比拟的。它降低了目标程序对连续存储区的需求是毋庸置疑的。其次，间接向量映射对于处理那些异形数组的优势比较明显。传统映射方案也可以处理异形数组，但代价较大。

在目前流行的程序设计语言中，行优先映射方案是应用最广的，例如，C、Pascal、C++ 等采用这种映射方案，只有一个经典的例外就是 Fortran，它使用了列优先映射方案。而一些较新的语言都支持间接向量的映射方案，最典型例子的就是 Java。

最后，将二维数组的映射方案推广到多维数组中。与二维数组类似，多维数组也有三种映射方案，即左下标优先映射、右下标优先映射、间接向量映射。由于多维数组并没有行、列之分，所以也就不存在行（列）优先之说。实际上，左（右）下标优先是一种更通用的观点。所谓“左下标优先”就是指将数组元素逐一映射到存储区时，左侧下标的变化速度比右侧下标的变化速度快。例如， $A[1,1,1]$ 、 $A[2,1,1]$ 、 $A[3,1,1]$ 、 $A[1,2,1]$。实际上，二维数组的列优先映射就是一种左下标优先方案。同样，所谓“右下标优先”就是指将数组元素逐一映射到存储区时，右侧下标的变化速度比左侧下标的变化速度快。二维数组的行优先映射就是一种右下标优先方案。而多维数组的间接向量映射方案与二维数组的类似，只是其间接

向量将比二维数组多得多。

2. 数组元素的引用

下面继续讨论如何将源程序中下标引用形式翻译成该元素的逻辑地址，或者说相对于该数组首地址的偏移量。这个计算过程依赖语言所选择的数组映射方案，不同映射方案的翻译方式是完全不同的。本书以最常见的行优先映射方式为例，深入剖析下标引用转换成逻辑地址的过程。

首先讨论最简单的一维数组的情形。假设 A 是一个长度为 n 的数组，而其每个元素占用的存储空间为 w 。那么， $A[i]$ 的地址为：

$$\text{Base}_A + (i - \text{low}_1) \times w$$

其中， Base_A 表示数组 A 的首地址， low_1 表示数组 A 第 1 维的下限。在 C 语言中，规定数组维度的下限都是 0，所以不必考虑 low_1 的取值。而 Pascal 的下限是由用户定义的，所以 low_1 的取值是不可以忽略的。读者只需通过简单的手工演算，不难发现这个地址公式是完全可行的。

下面再来看看二维数组的情形。假设 A 是一个 2 行 5 列的二维数组，各维的下限都是 1，其形式即为 $A[1..2, 1..5]$ 。那么， $A[2, 3]$ 元素的地址是多少呢？根据行优先映射的特点，不难得到如下的计算过程：

$$\begin{aligned} A[2, 3] \text{ 的地址} &= \text{Base}_A + (2 - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) \times w + (3 - \text{low}_2) \times w \\ &= \text{Base}_A + (2 - 1) \times (5 - 1 + 1) \times w + (3 - 1) \times w \\ &= \text{Base}_A + 5 \times w + 2 \times w = \text{Base}_A + 7 \times w \end{aligned}$$

其中， high_2 为第 2 维的上限。可以将整个计算过程分为三步：

1) 求元素所在行的首地址。换句话说，就是求所在行第 1 列元素之前的所有元素的个数 N 。假设当前行之前的行数为 L ，而该二维数组的列数为 C 。显然，不难得到如下等式：

$$N = L \times C$$

其中， L 可以用（当前行号 - low_1 ）表示，而 C 则可以用（ $\text{high}_2 - \text{low}_2 + 1$ ）表示。这样，在考虑元素占用存储空间 w 的情况下，就不难得到计算式中的第 2 项。

2) 求元素相对于所在行首地址的偏移。这个过程与一维数组元素的计算相同，并不难求得。同样，就可以轻松得到计算式中的第 3 项。

3) 将两者相加就是元素相对于数组首地址的偏移。

至此，就可以得到如下的计算公式（其中， $\text{len}_2 = \text{high}_2 - \text{low}_2 + 1$ ）：

$$A[c_1, c_2] \text{ 的地址} = \text{Base}_A + (c_1 - \text{low}_1) \times \text{len}_2 \times w + (c_2 - \text{low}_2) \times w$$

假设各维下限都为 0 时，可以将该公式简化为：

$$A[c_1, c_2] \text{ 的地址} = \text{Base}_A + (c_1 \times \text{len}_2 + c_2) \times w$$

这两个公式应该并不难理解，但是编译器设计者却不满足于此。这里，考虑更一般的情况，那就是 c_1 、 c_2 都是未知的变量，而不是常量。不难发现，由于存在变量，整个公式的计算过程都将依赖于目标代码完成，甚至 len_2 的计算也可能由目标程序完成，这并不是预期的结果。事实上，由于数组的上、下限都是已知的，所以 len_2 的计算是完全可以在编译阶段完成的。换句话说，若希望得到更精简的目标代码，就需要在计算的过程中尽可能将已知的常量折叠。确实，表面上看，公式本身并没有提供太多常量折叠的机会。不过，可以做如下公式变换：



$$A[c_1, c_2] \text{的地址} = \text{Base}_A + (c_1 - \text{low}_1) \times \text{len}_2 \times w + (c_2 - \text{low}_2) \times w \\ = \text{Base}_A - (\text{low}_1 \times \text{len}_2 + \text{low}_2) \times w + (c_1 \times \text{len}_2 + c_2) \times w$$

读者可能会对公式变换的意义有所疑惑。事实上，公式变换的目的就是为常量折叠创造机会。不难发现，将变量 c_1 、 c_2 集中到了公式的第3项中，而将可能的常量集中提取到了第2项中。如果数组上、下限已知，那么，第2项的值是在编译过程中计算得到的。虽然这样的变换可能会使公式的含义并不那么明晰，但是，它对生成精简的目标代码是有现实意义的。公式变换的思想并不是由笔者提出的，在经典编译技术中，对此早有详细记载。许多书中将第1项、第2项合称为公式的“不变部分”，而将第3项称为公式的“可变部分”。借助公式变换来实现常量折叠的思想印证了笔者先前提到的一个观点，即在最有利于解决某一问题的时刻处理该问题是最优的选择，不要拘泥于功能模块的划分。从理论上来说，完全可以按照原始的公式生成复杂的IR序列，而将精简优化的问题交由IR优化模块来处理。不过，届时所需付出的努力将是巨大的，而优化却未必能达到预期的效果。

最后，将二维数组的地址计算公式推广到多维数组中。根据多维数组的特性，可以得到如下的计算公式（其中， $p_i = c_i - \text{low}_i$ ）：

$$A[c_1, c_2, \dots, c_n] \text{的地址} = \text{Base}_A + (\dots((p_1 \times \text{len}_2 + p_2) \times \text{len}_3 + p_3) \dots) \times \text{len}_n + p_n \times w \\ = \text{Base}_A - (\dots((\text{low}_1 \times \text{len}_2 + \text{low}_2) \times \text{len}_3 + \text{low}_3) \dots) \times \text{len}_n + \text{low}_n \times w \\ + (\dots((c_1 \times \text{len}_2 + c_2) \times \text{len}_3 + c_3) \dots) \times \text{len}_n + c_n \times w$$

这里，笔者就不再详细解释此公式的推导过程，有兴趣的读者可以借助实例验证其正确性。这个具有一般意义的数组元素地址计算公式是基于如下两个基本条件讨论的：

(1) 数组的映射方式是右下标优先。

(2) 在编译过程中，数组各维度的上、下限是已知的。注意，如果上、下限是未知的，那么，试图通过公式变换来实现常量折叠是没有任何意义的。关于上、下限未知数组的相关问题，将稍后讨论。

3. 越界访问的检测

所谓“越界访问”就是指引用了数组预定义界外的元素，这种异常的访问会给程序带来不可预知的后果，甚至会导致整个程序崩溃。不过，不同语言对越界访问的检测力度是不同的。有些语言进行静态检测，即在编译过程中，判定访问是否越界。有些语言则进行动态检测，即在目标程序执行过程中，判定访问是否越界。然而，也有些语言不进行越界检测。其中，C语言对数组访问既不做静态越界检测，也不做动态越界检测。

静态越界检测的实现比较容易，只需根据符号表的相关信息，判断下标是否越界即可。不过，在实际编译器设计中，静态越界检测的功效并不显著。这是因为数组访问的下标通常是一个变量，而不是常量。下标的值是依赖实际运行环境的。在这种情况下，静态越界检测就完全失效了。

动态越界检测的实现稍复杂，大致的思想就是在目标代码的适当位置插入越界检测的程序段，并且在每次数组访问的代码之前，调用越界检测子程序。如果检测发现异常访问，则中断程序执行，并给出相应的出错提示。通常，动态越界检测程序段必须依赖数组的内情向量（就是用于描述数组信息的数据结构）完成检测，因此，编译器需要将内情向量以适当的形式置入目标程序中。

虽然动态越界检测可以很大程度上保证数组访问的安全性，但它的实现却是需要付出代

价的。动态越界检测机制不但会使目标程序的体积增大，而且还会延长目标程序执行的时间。其中的利弊是值得语言设计者斟酌的。在 Pascal 语言中，关于数组越界的检测并没有非常明确的规定，因此，不同的编译器实现存在着一定的差别。例如，Turbo Pascal 动态越界检测机制是可选的，但默认情况下只进行静态越界检测，并不做动态越界检测。

4. 动态数组的实现

所谓“动态数组”就是指数组的上、下限在编译阶段是不能确定的，换句话说，就是数组的上、下限在运行过程中是可变的，有时，亦称为“可变数组”。动态数组的历史远没有静态数组久远，早期的经典程序设计语言（如 C、Pascal 等）都不支持这一机制。随着软、硬件技术飞速发展，动态数组机制受到了越来越多语言的青睐，例如，C++、Java、C#等。

从应用的角度来说，除了可变上、下限之外，动态数组与静态数组的差异并不大。不过，从实现机制来说，两者的差异是非常巨大的。动态数组与指针却有几分相似之处。这里就来分析一下动态数组的内核实现。

数组都是在编译过程中由编译器静态分配存储空间的，在执行过程中，操作系统只需将逻辑存储区映射到实际的物理空间中即可。当然，实现静态分配的前提就是所需分配的空间大小在编译过程中是预知的。然而，动态数组所需空间大小在编译过程中是未知的，完全是由目标程序在运行过程中动态分配存储空间的。

编译器在处理动态数组类型的变量时，通常会为该变量分配一个数据结构，用于管理与维护动态数组。该数据结构主要包含如下两个部分：

(1) 内情向量。实际上，就是一套用于描述数组信息的数据结构，其中包含了数组的维度及上、下限信息。在静态数组中，如果不考虑动态越界检测，编译器不一定会将数组的内情向量写入目标代码，而只需要保留在符号表中即可。而动态数组则不然。假设 A 是一个行优先映射的动态数组，当试图计算 $A[2, 3]$ 的首地址时，编译器则必须按照先前的计算公式生成 IR，原公式中的不变部分也需要依靠指令来计算。原因很简单，公式中存在不变部分的前提就是数组各维度的上、下限是已知的。因此，目标程序必须使用一个数据结构（即内情向量）来保存此类信息，以便运行过程中动态计算元素的地址。一般来说，内情向量的元素个数是视数组的维数而定的。然而，在编译阶段，动态数组的维数通常是可以确定的。因此，不必担心内情向量长度无法确定的问题。

(2) 首地址。这里所讨论的“首地址”指的就是数组元素实际存储区域的首地址。通常，数组元素实际存储区域是由目标程序根据实际的需要调用操作系统接口申请获得的动态存储区，该区域中存储的才是数组的实际数据。这个过程是由目标程序与操作系统之间协调完成的，编译器是不可能获得任何有关该存储区的线索的。因此，编译器只能静态分配一个存储空间用于存放该区域的首地址，以便通过间接寻址方式访问数组元素。实际上，首地址的作用与 C、Pascal 语言中的指针是非常类似的。

当然，除了内情向量及首地址信息之外，为了便于编译器生成更为精简的目标代码，这个数据结构中还可能包含一些辅助信息，不再赘述。

最后，简单谈谈动态数组存储区管理。其存储布局如图 6-11 所示。笔者主要想讨论以下两个问题：

(1) 存储区的空间大小。一般来说，它是根据用户设置的数组上、下限而定的。理论上讲，只有显式设置了动态数组的上、下限后，用户才可以访问数组的元素。不过，并非所有

语言都明确规定用户必须显式设置动态数组的各维上、下限，很多语言都提供了默认的上、下限值。例如，STL 的 `vector` 就允许用户不显式设置上限，而使用默认值。那么，这个默认值的大小是多少较为适合呢？这是需要设计者斟酌的。

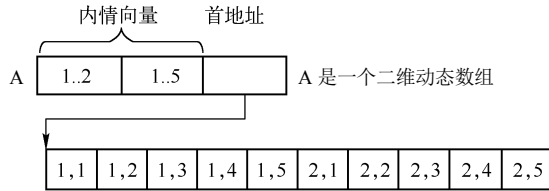


图 6-11 动态数组的存储布局

(2) 存储空间的扩展与回收。设置动态数组的目的就是便于用户动态改变数组的上、下限，因此，存储空间的扩展与回收是非常重要的。当然，重设存储空间的大小是必要的，却并不仅限于此。一般来说，用户改变数组上、下限时，并不希望现存的数据丢失。由于数组形态发生了变化，现存的数据元素的映射位置也必须随即改变。用户不可能也没必要了解数组的物理存储布局，因此，这项工作对于用户来说是完全透明的。尤其对于支持异形数组的语言，这方面的设计显得极其重要。另外，为了便于用户使用，有些语言将动态数组扩展、回收等交由编译器处理或者封装成库的形式。例如，当用户向 `vector` 对象添加元素时，是否关注过 `vector` 对象的容量呢？实际上，STL 为此做了许多幕后工作，当用户调用 `push_back` 方法向一个 `vector` 对象添加元素时，许多辅助的程序片段将被执行，以便在数据溢出时自动扩展 `vector` 对象的动态存储区。至于一次扩展空间的大小是非常有讲究的，不同的编译器对此的理解不同。这个数字过大可能会导致存储空间的浪费，过小可能会使扩展操作频繁执行导致效率降低。

6.5.6 数组元素操作数的翻译

简言之，数组元素操作数的翻译就是按照地址计算公式将下标引用形式转换为逻辑地址形式。在正式分析源码之前，先来看一个实例。

例 6-3 根据输入源程序生成相应的 IR 序列。

输入源程序如下：

```

program aa(input, output);
var a: array[1..10, 1..5] of integer;
    i, j: integer;
begin
    a[i, j]:=1;
end.

```

在表 6-7 中，列出了两个完整的 IR 序列。其中，左列是不经过任何优化的 IR 序列，它就是由语义子程序直接生成的，并没有作任何加工。而右列是经过优化后的 IR 序列，从优化效果来看，主要是简化了一些冗余的类型转换 IR。虽然右列的 IR 比较精简，但是它丝毫不失为分析数组元素翻译的好例子。下面将基于右列的 IR 作相关讨论。计算公式如下：

表 6-7 数组寻址的实例

优化前的 IR 序列	优化后的 IR 序列
0: (ByteToInt ,5,null,_T1)	0: (MUL_4 ,I,5,_T0)
1: (MUL_4 ,I,_T1,_T0)	1: (ADD_4 ,J,_T0,_T2)
2: (ADD_4 ,J,_T0,_T2)	2: (SUB_4 ,_T2,6,_T3)
3: (ByteToInt ,6,null,_T4)	3: (MUL_4 ,_T3,4,_T5)
4: (SUB_4 ,_T2,_T4,_T3)	4: (GETADDR ,A,null,_T7)
5: (ByteToInt ,4,null,_T6)	5: (ADD_4 ,_T7,_T5,_T7)
6: (MUL_4 ,_T3,_T6,_T5)	6: (ASSIGN_4 ,1,null,@_T7)
7: (GETADDR ,A,null,_T7)	
8: (ADD_4 ,_T7,_T5,_T7)	
9: (ByteToInt ,1,null,_T8)	
10: (ASSIGN_4 ,_T8,null,@_T7)	

$$\begin{aligned}
 A[c_1, c_2 \dots c_n] \text{ 的地址} &= \text{Base}_A + ((\dots((p_1 \times \text{len}_2 + p_2) \times \text{len}_3) + p_3) \dots) \times \text{len}_n + p_n) \times w \\
 &= \text{Base}_A - ((\dots((\text{low}_1 \times \text{len}_2 + \text{low}_2) \times \text{len}_3 + \text{low}_3) \dots) \times \text{len}_n + \text{low}_n) \times w \\
 &\quad + ((\dots((c_1 \times \text{len}_2 + c_2) \times \text{len}_3 + c_3) \dots) \times \text{len}_n + c_n) \times w
 \end{aligned}$$

第 0 行：计算 $c_1 \times \text{len}_2$ ，并将结果送入 $_T0$ 。 c_1 即为第 1 维的下标 i ，而 len_2 即为第 2 维的长度。

第 1 行：计算 $c_2 + _T0$ ，并将结果送入 $_T2$ 。其中， c_2 即为第 2 维的下标 j 。

第 2 行：计算公式中第 3 项与公式中第 2 项的差，将结果送入 $_T3$ 。其中，第 2 项常量的值即为 6。

第 3 行：将 $_T3 \times w$ ，并将结果送入 $_T5$ 。其中， w 即为 `integer` 类型数据占用的空间数。

第 4 行：获取数组 A 的首地址，即为 Base_A ，将结果送入 $_T7$ 。

第 5 行：将 $_T5$ 与 $_T7$ 相加，即获得了 $A[i, j]$ 的地址，将计算结果送回 $_T7$ 。

第 6 行：通过对 $_T7$ 的间接寻址，将常量送入 $A[i, j]$ 中。

至此，通过一个实例，读者应该已经了解了数组元素的寻址问题。下面，就来看看数组元素操作数的相关文法：

【文法 6-5】

因子 → 变量 053

变量 1 → [098 表达式列表 099] 059 变量 1

表达式列表 → 表达式 055 表达式列表 1

表达式列表 1 → , 表达式 057 表达式列表 1

→ ϵ

semantic055: 1. 对当前下标表达式进行有效性检查。

2. 根据下标表达式，生成 IR。若存在常量折叠的条件，则进行常量折叠。

semantic057: 1. 对当前下标表达式进行有效性检查。

2. 根据下标表达式，生成 IR。若存在常量折叠的条件，则进行常量折叠。

semantic059: 1. 有效性检查。

2. 计算公式中第 3 项的常量。

3. 根据地址计算公式，生成 IR。

semantic098: 1. 设置 `iExpListFlag` 标志。

semantic099: 1. 撤销 `iExpListFlag` 标志。



其中，数组下标表达式的有效性检查主要包括以下几项：

- (1) 根据 Pascal 语言的规定，数组的下标必须为整型表达式。
- (2) 越界访问检测。Neo Pascal 只进行静态检测，不进行动态检测。

(3) 下标的个数必须与数组的维数一致。例如，试图用“A[1]”或“A[1, 2, 3]”的方式引用二维数组 A 都是非法的。

此外，由于“表达式列表”是一个复用非终结符，所以需要 semantic098、semantic099 根据实际情况设置语义处理标志。这种思想在处理“标识符列表”相关语义时，已经详细解释过了，读者应该并不陌生。

程序 6-13 semantic.cpp

相关文法：

表达式列表 → 表达式 055 表达式列表 1

```
1  bool semantic055()
2  {
3      if (iExpListFlag.top()==1)
4      {
5          if (bConstFlag || Operand.empty() || CurrentVar.empty())
6          {
7              EmitError("语义错误",TokenList.at(iListPos-1));
8              return false;
9          }
10         OpInfo Tmp,Op1,Op2,TmpRslt;
11         IRCode TmpIR;
12         int iTypeLink;
13         Var* TmpVar;
14         Tmp=Operand.top();
15         Operand.pop();
16         TmpVar=&CurrentVar.top();
17         TmpVar->m_iDim=1;
18         iTypeLink=TmpVar->m_VarTypeStack.top().m_iLink;
19         if (Tmp.m_iType==OpInfo::CONST)
20         {
21             if (CType::IsInt(CType::GetOpType(Tmp)) &&
22                 SymbolTbl.TypeInfoTbl[iTypeLink].m_ArrayInfo.at(0).m_iStart
23                 <=SymbolTbl.ConstInfoTbl[Tmp.m_iLink].m_iVal &&
24                 SymbolTbl.TypeInfoTbl[iTypeLink].m_ArrayInfo.at(0).m_iEnd
25                 >=SymbolTbl.ConstInfoTbl[Tmp.m_iLink].m_iVal)
26             {
27                 int i;
28                 if (SymbolTbl.TypeInfoTbl[iTypeLink].m_ArrayInfo.size()>=2)
29                 {
30                     i=SymbolTbl.TypeInfoTbl[iTypeLink].m_ArrayInfo.at(1).m_iEnd;
31                     i=i-SymbolTbl.TypeInfoTbl[iTypeLink].m_ArrayInfo.at(1).m_iStart+1;
32                     i=SymbolTbl.ConstInfoTbl[Tmp.m_iLink].m_iVal*i;
33                 }
```

```
34     else
35         i=SymbolTbl.ConstInfoTbl[Tmp.m_iLink].m_iVal;
36     Op2.m_iLink=SymbolTbl.RecConstTbl(itos(i),3);
37     Op2.m_iType=OpInfo::CONST;
38     if (TmpVar->m_eOffsetType==OffsetType::ConstOffset ||
39         TmpVar->m_eOffsetType==OffsetType::VarOffset)
40     {
41         TmpVar->m_OffsetVec.push_back(OffsetStruct(
42             TmpVar->m_eOffsetType,TmpVar->m_iOffsetLink));
43     }
44     Operand.push(Op2);
45 }
46 else
47 {
48     EmitError("越界访问或数组下标类型不正确",TokenList.at(iListPos-1));
49     return false;
50 }
51 }
52 else
53 {
54     if (!CType::IsInt(CType::GetOpType(Tmp)))
55     {
56         EmitError("数组下标类型不正确",TokenList.at(iListPos-1));
57         return false;
58     }
59     if (SymbolTbl.TypeInfoTbl[iTypeLink].m_ArrayInfo.size()>=2)
60     {
61         Op2.m_iLink=SymbolTbl.RecConstTbl(itos(SymbolTbl.TypeInfoTbl[iTypeLink]
62             .m_ArrayInfo.at(1).m_iEnd-SymbolTbl.TypeInfoTbl[iTypeLink].m_ArrayInfo.at(1)
63             .m_iStart+1),3);
64         Op2.m_iType=OpInfo::CONST;
65         TmpRslt.m_iLink=SymbolTbl.GetTmpVar(SymbolTbl.ProcStack.top()
66             ,StoreType::T_INTEGER);
67         TmpRslt.m_iType=OpInfo::VAR;
68         if (!GenIR(Tmp,Op2,TmpRslt,BasicOpType::MUL))
69             return false;
70         Tmp=TmpRslt;
71     }
72     if (TmpVar->m_eOffsetType==OffsetType::ConstOffset ||
73         TmpVar->m_eOffsetType==OffsetType::VarOffset)
74     {
75         TmpVar->m_OffsetVec.push_back(OffsetStruct(TmpVar->m_eOffsetType
76             ,TmpVar->m_iOffsetLink));
77     }
78     Operand.push(Tmp);
79 }
```



```
80         return true;
81     }
82     if (iExpListFlag.top()==2)
83     {
84         iExpListNum.top()++;
85     }
86     return true;
87 }
```

从完整的文法来看，不难发现，`semantic055` 的另一个功用就是处理集合变量的表达式列表。因此，借助于 `iExpListFlag` 栈传递处理标志，告知 `semantic055` 所需完成的语义动作。当 `iExpListFlag` 栈顶元素为 1 时，表示“表达式列表”是由“变量 1”的候选式推导出来的，此时，`semantic055` 的语义动作就是处理数组的下标。否则，表示“表达式列表”是由“因子”的候选式推导出来的，那么，`semantic055` 的语义动作就是处理集合变量的值。

第 5~9 行：语义有效性检查。其中，`Operand` 栈中保存的是下标表达式的结果操作数，而 `CurrentVar` 栈中保存的是数组对象的信息。另外，对常量符号取下标也是无意义的。

第 14 行：获取 `Operand` 栈顶元素，该元素即为当前下标表达式的结果操作数。

第 16 行：获取 `CurrentVar` 栈顶元素，该元素即为当前操作数。由于 `semantic055` 处理的表达式必须是数组第 1 维的下标，因此，当前操作数的 `m_VarTypeStack` 栈顶类型一定是数组。与 C 语言不同，Pascal 严格限制下标运算只能应用于数组。

第 17 行：设置 `m_iDim` 属性。这个属性主要用于记录当前已经分析完成的下标表达式个数。根据地址计算公式，不难发现，其中 `leni` 的取值完全依赖于下标表达式的序号。因此，将下标表达式的个数暂存在 `m_iDim` 属性中是有必要的。

第 18 行：获取当前变量的类型信息。

第 19 行：判断下标表达式是否为常量。由于 `semantic055` 仅用于处理第 1 维的下标表达式，所以只需要考虑当前下标表达式是否为常量即可。

第 21~25 行：下标表达式的有效性检查，包括两项判断：第一项，下标的类型是否为整型；第二项，下标是否越界访问。由于下标表达式是常量，所以可以进行静态越界访问检测。

第 28~35 行：计算偏移量。注意，这里需要考虑当前维度是否为最末的维度。根据公式可知，除最末维度之外，都需要乘以下一维度的 `len`。由于 `semantic055` 只处理第一维的偏移，因此，只需判断数组维度是否大于或等于 2 即可。

第 36、37 行：生成常量操作数。

第 38~43 行：这个条件语句是非常重要的。先前所讨论的情况都只是以数组首地址为基准考虑偏移的，并没有涉及数组首地址本身的偏移。换句话说，数组的首地址也可能是相对于某个变量符号的偏移。例如，数组作为记录的一个字段，那么，数组的首地址就是相对于该记录变量的一个常量偏移。因此，如果当前符号本身存在偏移时，就需要将其保存在一个暂存空间中。在计算数组首地址时，再将其取出。

第 44 行：将常量操作数压入 `Operand` 栈。

第 54~79 行：处理下标表达式不是常量的情况。

第 54~58 行：下标表达式的有效性检查。由于表达式不是常量，因此不需要考虑静态

越界检查。

第 61~70 行：生成 IR 用于计算 $C_n \times \text{len}_{n+1}$ 的值。

第 72~77 行：与第 38~43 行的作用相同，暂存数组本身的偏移。

第 78 行：将常量操作数压入 Operand 栈。

程序 6-14 semantic.cpp

相关文法：

表达式列表 1 \rightarrow , 表达式 057 表达式列表 1

```

1  bool semantic057()
2  {
3      if (iExpListFlag.top()==1)
4      {
5          if (bConstFlag || Operand.size()<2 || CurrentVar.empty())
6          {
7              EmitError("语义错误",TokenList.at(iListPos-1));
8              return false;
9          }
10         OpInfo Tmp1,Tmp2,Op1,Op2,TmpRslt;
11         IRCode TmpIR;
12         Var* TmpVar;
13         Tmp1=Operand.top();
14         Operand.pop();
15         Tmp2=Operand.top();
16         Operand.pop();
17         TmpVar=&CurrentVar.top();
18         int j=++TmpVar->m_iDim;
19         if (SymbolTbl.TypeInfoTbl[TmpVar->m_VarTypeStack.top().m_iLink].m_ArrayInfo.size()<j)
20         {
21             EmitError("数组访问维数大于声明维数",TokenList.at(iListPos-1));
22             return false;
23         }
24         if (Tmp1.m_iType==OpInfo::CONST && Tmp2.m_iType==OpInfo::CONST)
25         {
26             if (CType::IsInt(CType::GetOpType(Tmp1)) && CType::IsInt(CType::GetOpType(Tmp2)) &&
27                 SymbolTbl.TypeInfoTbl[TmpVar->m_VarTypeStack.top().m_iLink].m_ArrayInfo.at
28                 (j-1).m_iStart
29                 <=SymbolTbl.ConstInfoTbl[Tmp1.m_iLink].m_iVal &&
30                 SymbolTbl.TypeInfoTbl[TmpVar->m_VarTypeStack.top().m_iLink].m_ArrayInfo.at
31                 (j-1).m_iEnd
32                 >=SymbolTbl.ConstInfoTbl[Tmp1.m_iLink].m_iVal)
33             {
34                 int i;
35                 if (SymbolTbl.TypeInfoTbl[TmpVar->m_VarTypeStack.top().m_iLink]
36                     .m_ArrayInfo.size()>j)
37                     i=(SymbolTbl.ConstInfoTbl[Tmp2.m_iLink].m_iVal+SymbolTbl
38                         .ConstInfoTbl[Tmp1.m_iLink].m_iVal)*(SymbolTbl.TypeInfoTbl[

```



```
39         TmpVar->m_VarTypeStack.top().m_iLink].m_ArrayInfo.at(j).m_iEnd
40         -SymbolTbl.TypeInfoTbl[TmpVar->m_VarTypeStack.top().m_iLink]
41         .m_ArrayInfo.at(j).m_iStart+1);
42     else
43         i=(SymbolTbl.ConstInfoTbl[Tmp2.m_iLink].m_iVal+SymbolTbl
44         .ConstInfoTbl[Tmp1.m_iLink].m_iVal);
45     char cBuffer[10];
46     itoa(i,cBuffer,10);
47     Op2.m_iLink=SymbolTbl.RecConstTbl(cBuffer,3);
48     Op2.m_iType=OpInfo::CONST;
49     Operand.push(Op2);
50     }
51     else
52     {
53         EmitError("数组访问越界或数组下标类型不正确",TokenList.at(iListPos-1));
54         return false;
55     }
56 }
57 else
58 {
59     if (!CType::IsInt(CType::GetOpType(Tmp1)) || !CType::IsInt(CType::GetOpType(Tmp2)))
60     {
61         EmitError("数组下标类型不正确",TokenList.at(iListPos-1));
62         return false;
63     }
64     TmpRslt.m_iLink=SymbolTbl.GetTmpVar(SymbolTbl.ProcStack.top(),StoreType::T_INTEGER);
65     TmpRslt.m_iType=OpInfo::VAR;
66     if (!GenIR(Tmp1,Tmp2,TmpRslt,BasicOpType::ADD))
67         return false;
68     Operand.push(TmpRslt);
69     if (SymbolTbl.TypeInfoTbl[TmpVar->m_VarTypeStack.top().m_iLink].m_ArrayInfo.size(>j)
70     {
71         Op2.m_iLink=SymbolTbl.RecConstTbl(itos(SymbolTbl.TypeInfoTbl
72         [TmpVar->m_VarTypeStack.top().m_iLink].m_ArrayInfo.at(j)
73         .m_iEnd-SymbolTbl.TypeInfoTbl[TmpVar->m_VarTypeStack.top()
74         .m_iLink].m_ArrayInfo.at(j).m_iStart+1),3);
75         Op2.m_iType=OpInfo::CONST;
76         Op1=Operand.top();
77         Operand.pop();
78         TmpRslt.m_iLink=SymbolTbl.GetTmpVar(SymbolTbl.ProcStack.top()
79         ,StoreType::T_INTEGER);
80         TmpRslt.m_iType=OpInfo::VAR;
81         if (!GenIR(Op1,Op2,TmpRslt,BasicOpType::MUL))
82             return false;
83         Operand.push(TmpRslt);
84     }
```

```
85     }
86     return true;
87 }
88 if (iExpListFlag.top()==2)
89 {
90     iExpListNum.top()++;
91 }
92 return true;
93 }
```

semantic057 的功能与 semantic055 是基本类似的，不过，需要考虑叠加前一维计算得到的偏移量。同样，semantic055 也存在复用的情况，因此，也需要借助于 iExpListFlag 栈传递处理标志，其基本思想与 semantic055 是完全一致的。

第 5~9 行：语义有效性检查。注意，根据文法，在这种情况下，Operand 栈中必须至少存在两个操作数，即前一维计算得到的偏移量及当前的下标表达式。否则，是不符合语义的。而 CurrentVar 栈中保存的是数组对象的信息。当然，对常量符号取下标也是无意义的。

第 13 行：获取下标表达式的结果操作数。根据栈的性质，这是必然的。

第 15 行：获取前一维计算得到的偏移量操作数。

第 17 行：获取 CurrentVar 栈顶元素，该元素即为当前操作数。

第 18 行：设置 m_iDim 属性。这个属性主要用于记录当前已经分析完成的下标表达式个数。根据地址计算公式，不难发现，其中 len_i 的取值完全是依赖于下标表达式的序号。因此，将下标表达式的个数暂存在 m_iDim 属性中是有必要的。

第 19~23 行：判断是否存在维度越界的情况，即访问的维度大于声明的维度。在处理 semantic055 时，并不需要考虑，这是因为数组至少是一维的。

第 24 行：判断是否满足常量折叠的条件，即前一维计算得到的偏移与当前下标表达式的结果都是常量。注意，仅有这种情况才能进行常量折叠，否则，只能通过生成 IR 计算偏移。

第 26~30 行：下标表达式的有效性检查，包括两项判断：第一项，下标的类型是否为整型；第二项，下标是否越界访问。由于下标表达式是常量，所以可以进行静态越界访问检测。

第 32~47 行：处理与 semantic055 类似，根据公式求得当前维度的偏移值。这里同样需要考虑当前维度是否为最末的维度。根据公式可知，除最末维度之外，都需要乘以下一维度的 len。

第 57~61 行：下标表达式的有效性检查。由于表达式不是常量，因此不需要考虑静态越界检查。

第 62~65 行：生成前一维的偏移与当前下标的求和 IR，并申请一个临时变量用于保存这个计算结果。

第 66 行：将计算结果操作数压入 Operand 栈。

第 67~82 行：基本思想与第 35~39 行的处理类似，只不过这里生成的是 IR 指令，而不是常量偏移值。

至此，笔者详细分析了 semantic055、semantic057 的主要功能及实现细节。最后，再来



看看 semantic059 的基本实现。与先前两个语义子程序相比，semantic059 稍复杂，它是数组元素寻址翻译的最后一个部分。

程序 6-15 semantic.cpp

相关文法:

变量 1 → [098 表达式列表 099] 059 变量 1

```
1  bool semantic059()
2  {
3      if (bConstFlag || Operand.empty() || CurrentVar.empty())
4      {
5          EmitError("语义错误",TokenList.at(iListPos-1));
6          return false;
7      }
8      Var TmpVar=CurrentVar.top();
9      CurrentVar.pop();
10     if (SymbolTbl.TypeInfoTbl[TmpVar.m_VarTypeStack.top().m_iLink].m_ArrayInfo.size()>TmpVar.m_iDim)
11     {
12         EmitError("数组访问维数小于声明维数",TokenList.at(iListPos-1));
13         return false;
14     }
15     int iArrayLowC=0;
16     int i;
17     for (i=0;i<SymbolTbl.TypeInfoTbl[TmpVar.m_VarTypeStack.top().m_iLink].m_ArrayInfo.size();i++)
18     {
19         iArrayLowC=iArrayLowC+SymbolTbl.TypeInfoTbl[TmpVar.m_VarTypeStack.top().m_iLink]
20             .m_ArrayInfo.at(i).m_iStart;
21         if (i+1<SymbolTbl.TypeInfoTbl[TmpVar.m_VarTypeStack.top().m_iLink].m_ArrayInfo.size())
22             iArrayLowC=iArrayLowC*(SymbolTbl.TypeInfoTbl[TmpVar.m_VarTypeStack.top()
23                 .m_iLink].m_ArrayInfo.at(i+1).m_iEnd-SymbolTbl.TypeInfoTbl[TmpVar
24                 .m_VarTypeStack.top().m_iLink].m_ArrayInfo.at(i+1).m_iStart+1);
25     }
26     int iArrayType=SymbolTbl.CalcTypeSize(SymbolTbl.TypeInfoTbl[TmpVar.m_VarTypeStack
27         .top().m_iLink].m_iLink);
28     OpInfo Tmp;
29     if (Operand.top().m_iType==OpInfo::CONST)
30     {
31         int j=(SymbolTbl.ConstInfoTbl[Operand.top().m_iLink].m_iVal-iArrayLowC)*iArrayType;
32         TmpVar.m_VarTypeStack.push(VarType(SymbolTbl.TypeInfoTbl[CType::GetRealType(SymbolTbl
33             .TypeInfoTbl[TmpVar.m_VarTypeStack.top().m_iLink].m_iLink)].m_eDataType,CType::
34             GetRealType(SymbolTbl.TypeInfoTbl[TmpVar.m_VarTypeStack.top().m_iLink].m_iLink)));
35         TmpVar.m_eOffsetType=OffsetType::ConstOffset;
36         TmpVar.m_iOffsetLink= SymbolTbl.RecConstTbl(itos(j),3);
37         CurrentVar.push(TmpVar);
38         Operand.pop();
39     }
40     else
```

```
41     {
42         OpInfo TmpRslt, TmpOp1, TmpOp2;
43         TmpOp2.m_iType=OpInfo::CONST;
44         TmpOp2.m_iLink=SymbolTbl.RecConstTbl(itos(iArrayLowC),3);
45         TmpRslt.m_iType=OpInfo::VAR;
46         TmpRslt.m_iLink=SymbolTbl.GetTmpVar(SymbolTbl.ProcStack.top(),StoreType::T_INTEGER);
47         TmpOp1=Operand.top();
48         if (!GenIR(TmpOp1, TmpOp2, TmpRslt, BasicOpType::SUB))
49             return false;
50         TmpOp1=TmpRslt;
51         TmpOp2.m_iLink=SymbolTbl.RecConstTbl(itos(iArrayType),3);
52         TmpRslt.m_iLink=SymbolTbl.GetTmpVar(SymbolTbl.ProcStack.top(),StoreType::T_INTEGER);
53         if (!GenIR(TmpOp1, TmpOp2, TmpRslt, BasicOpType::MUL))
54             return false;
55         Operand.pop();
56         VarType tmp;
57         int iTmpLink=TmpVar.m_VarTypeStack.top().m_iLink;
58         tmp.m_StoreType=SymbolTbl.TypeInfoTbl[iTmpLink].m_iLink==1?SymbolTbl
59             .TypeInfoTbl[CType::GetRealType(iTmpLink)].m_eDataType:SymbolTbl.TypeInfoTbl[
60             CType::GetRealType(SymbolTbl.TypeInfoTbl[iTmpLink].m_iLink)].m_eDataType;
61         tmp.m_iLink=SymbolTbl.TypeInfoTbl[iTmpLink].m_iLink==1?iTmpLink:
62             CType::GetRealType(SymbolTbl.TypeInfoTbl[iTmpLink].m_iLink);
63         TmpVar.m_VarTypeStack.push(tmp);
64         TmpVar.m_eOffsetType=OffsetType::VarOffset;
65         TmpVar.m_iOffsetLink=TmpRslt.m_iLink;
66         CurrentVar.push(TmpVar);
67     }
68     return true;
69 }
```

第 3~7 行：语义有效性检查。当然，对常量符号取下标也是无意义的。

第 8 行：获取 `CurrentVar` 栈顶元素，该元素即为当前操作数。

第 10~14 行：判断访问维度与数组声明维度是否一致，Pascal 语言对此是有严格要求的。

第 17~25 行：根据公式计算其中的不变部分，将其临时保存在 `iArrayLowC` 变量中。

第 26 行：由类型系统计算得到每个数组元素所占用的空间大小。

第 31~39 行：处理偏移为常量的情况。

第 32~34 行：更新当前操作数的类型，也就是将数组元素的类型压入 `m_VarTypeStack` 栈。注意，这里的类型跟踪是非常重要的，有助于编译器及时了解当前操作数的实际类型。

第 35~36 行：更新当前操作数的偏移信息。这里，生成一个常量符号作为偏移量即可。

第 42~66 行：处理偏移为变量的情况。

第 42~49 行：根据公式生成 IR，计算偏移操作数与 `iArrayLowC` 的差值。

第 50~54 行：获取数组元素所占用空间的大小。根据公式生成 IR，计算最终的实际偏移。

关于数组元素操作数的语义处理，暂且讨论至此。



6.5.7 指针运算的翻译

指针一度被誉为程序设计语言的精髓所在。不过，学术界对于指针的应用一直存在争议。本书不对指针机制本身作过多的评价，因为这是非常困难的。然而，从编译器设计的角度来说，指针却是一个非常不错的语法机制。由于指针的存在，很多原先看似平凡的问题变得丰富多彩，当然，这可能是以实现难度的增加作为代价的。这里，主要关注指针的一些常用运算的相关语义处理。在 C 语言中，与指针相关的运算主要有“*”和“&”，这是指针的两个基本运算，当然，读者可能还会想到“++”、“--”等。不过，由于其他运算相对简单，与普通单、双目运算非常类似，因此，没有必要作深入讨论。

标准 Pascal 关于指针及其运算的描述比较模糊，因此，各种版本的编译器对此的理解与处理是不尽相同的。早期的 Pascal 编译器并不支持地址运算（“@”），在这种情况下，指针只允许指向运行时刻用户程序动态申请的空间，而不能指向普通的变量。虽然这种处理是以牺牲指针的灵活性为代价的，但是大大降低了实现的复杂度。随着 C 语言的风靡，C 的指针机制逐渐成为了一个标准，许多商用 Pascal 编译器都竞相效仿，引入了“@”运算符。在正式讨论指针相关运算的实现之前，先简单解释一下左值、右值的概念。

1. 左值与右值

在程序设计语言及编译器设计中，左值、右值是一对非常重要的概念。不过，国内的教材关于这一概念的阐述却非常罕见，甚至一些有多年经验的程序员对这个概念都比较模糊。不能深刻理解左值、右值的概念，对于分析指针的运算是非常不利的。实际上，在国外的一些专业书籍中，左值、右值是出现频率极高的两个名词。因此，笔者认为有必要深入了解这两个概念，以便以后阅读一些国外的经典教材。

实际上，通俗地讲，左值就是指可以出现在赋值号左边的值，也就是指那些可以被修改的值。而右值就是指可以出现在赋值号右边的值。从表面上来看，这两个概念的描述似乎是轻描淡写的。不过，其内含却并不简单。注意，左值、右值是讨论表达式的一种分类方法而已，并不意味着左值必定是出现在赋值号左边的。先前，笔者只是强调左值“可以”出现在赋值号左边，但并不是必须出现在赋值号左边的。

在早期，左值、右值的概念的提出就是用于区别一个值是否可以改变。通常，左值是可以改变的，而右值是不能改变的。不过，这种观点并不太准确。在现代 C++ 语言中，左值、右值的概念已经失去了其原有的意义。C++ 的观点认为左值通常是指可以通过具体的名字或引用来指定的对象。除了左值之外，其余的都可以视为右值。下面，先来看几个例子，见表 6-8。

表 6-8 表达式左值

表 达 式	左 值
x = 421	x
*ptr = 23	*ptr
a++	a++
b[0] = 1000	b[0]
const int m = 1000	m
int& f()	f()的返回值

这里简单说明两点：

(1) `const` 限定词修饰的符号仍然可以视为左值。在这种情况下，试图修改符号的值可能是非法的，但并不影响其成为左值，因为它可以通过名字引用的。

(2) 返回引用的函数也可以视为左值。事实上，除了返回引用之外，其他类型的返回值都不是左值。其实这是很好理解的。如果一个函数的返回值是一个普通对象，那么这个普通对象必定是临时的，当然，也不可能通过一个显式的名字来访问。但如果一个函数返回引用，那么，它的返回值就没有意义了，因为它是另一个名字的别名，通过这个别名，就可以直接访问被别名的对象，因此，返回引用就是左值。

理论上讲，左值是可以转换为右值的，但右值却不一定能转换为左值。那么，讨论左值、右值对于编译器设计又有什么意义呢？在程序设计语言中，左值、右值的限定是非常频繁的，仅以指针处理为例，左值、右值的意义就比较显著。例如，C 语言规定 `&` 运算符的操作数只能是左值或函数名字。那么，从编译器设计的角度，又如何理解左值问题呢？实际上，在所有变量符号中，只有一类变量符号是不能作左值的，即非间接寻址访问的临时变量。读者一旦体会到这一本质，确定左值与否将变得非常容易。相对于左值而言，右值的问题就非常容易了，并没有太多的理论。至此，笔者已经详细阐述了左值与右值的相关话题，这两个概念比较重要，建议读者仔细推敲。

2. 间接访问运算

间接访问运算是一种基本的指针运算。在 Pascal 中，间接访问运算符 `^` 通过指针进行间接访问，它与地址运算符 `@` 是互逆的。如果 `x` 是一个变量，那么，表达式 `(@x)^` 与 `x` 是相同的。间接访问运算规定操作数必须是指针，其运算结果就是引用这个指针所指向的对象。间接访问运算的结果是一个左值。

当然，间接访问运算是否安全是由程序员确定的。编译器通常不会静态检测指针是否为空指针或野指针，因这类指针而导致的不安全寻址是由程序员负责的。实际上，Pascal 的间接访问运算应用与 C 语言的 `*` 运算符是极其相似的，这里就不再深入讨论了。下面就来看看 Neo Pascal 的相关实现。

【文法 6-6】

因子	→	变量 053
变量	→	标识符 054 变量 1
变量 1	→	<code>^</code> 058 变量 1

程序 6-16 semantic.cpp

相关文法：

表达式列表 1 → , 表达式 057 表达式列表 1

```
1 bool semantic058()
2 {
3     if (bConstFlag || CurrentVar.empty())
4     {
5         EmitError("语义错误",TokenList.at(iListPos-1));
6         return false;
7     }
8     Var TmpVar=CurrentVar.top();
```



```
9     if (TmpVar.m_VarTypeStack.empty() || TmpVar.m_VarTypeStack.top().m_StoreType!=StoreType::
10         T_POINTER)
11     {
12         EmitError("^运算符只能运用于指针类型",TokenList.at(iListPos-1));
13         return false;
14     }
15     int iTypeLink=CType::GetRealType(SymbolTbl.TypeInfoTbl[TmpVar.m_VarTypeStack.top()
16         .m_iLink].m_iLink);
17     TmpVar.m_VarTypeStack.pop();
18     TmpVar.m_VarTypeStack.push(VarType(SymbolTbl.TypeInfoTbl[iTypeLink].m_eDataType,iTypeLink));
19     semantic053());
20     OpInfo TmpOp=Operand.top();
21     Operand.pop();
22     if (TmpOp.m_bRef)
23     {
24         OpInfo TmpOp1,TmpRslt;
25         TmpOp1.m_iType=OpInfo::VAR;
26         TmpOp1.m_iLink=TmpOp.m_iLink;
27         TmpOp1.m_bRef=true;
28         TmpRslt.m_iType=OpInfo::VAR;
29         TmpRslt.m_iLink=SymbolTbl.GetTmpVar(SymbolTbl.ProcStack.top(),StoreType::T_POINTER);
30         TmpRslt.m_bRef=false;
31         SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
32             OpType::ASSIGN_4,TmpOp1,TmpRslt));
33         TmpOp.m_bRef=false;
34         TmpOp.m_iLink=TmpRslt.m_iLink;
35     }
36     TmpVar.m_iVarLink=TmpOp.m_iLink;
37     TmpVar.m_bRef=true;
38     TmpVar.m_eOffsetType=OffsetType::NoneOffset;
39     TmpVar.m_iOffsetLink=-1;
40     CurrentVar.push(TmpVar);
41     return true;
42 }
```

第 3~7 行：语义有效性检查。当然，对常量符号取下标也是无意义的。

第 8 行：获取 `CurrentVar` 栈顶元素，该元素即为当前操作数。

第 9~13 行：间接访问运算符只能应用于指针类型变量。注意，间接访问运算并没有规定其操作数必须为左值，因此，不需要判断当前变量是否为左值。

第 14~15 行：获取当前变量的类型链的指针。

第 16 行：将当前变量的 `m_VarTypeStack` 栈的栈顶元素弹出，也就是将指针类型的描述信息弹出。

第 17 行：将指针所指向的基类型压入当前变量的 `m_VarTypeStack`。这里，值得注意的是类型的变化过程，一次间接访问会将当前变量类型由原来的指针类型转换为该指针的基类型。编译器则借助于 `m_VarTypeStack` 栈跟踪寻址过程中的类型的变化信息。

第 18 行：调用 `semantic053` 语义子程序，分析当前变量的相关语义，这是一个非常重要的语义动作。在 Pascal 或 C 语言中，间接访问运算的结果本身也是一个左值，因此，大多数运算都可以基于其结果进行，当然，也包括许多单目运算。例如，`a[i]`、`a.p` 等形式都是合法的，这与 C 语言的*运算是类似的。以 `a[i]` 为例，设计者必须注意 `a[i]` 与 `a[i]` 的差异。显然，`a[i]` 是相对于 `a` 变量首地址的偏移，而 `a[i]` 是相对于指针变量 `a` 所指向的空间区域首地址的偏移。换句话说，对于一个完整的操作数而言，间接访问运算符前后两部分并没有直接的联系，只是后者是依赖于前者计算得到的符号内的值而已。因此，这里调用 `semantic053` 的目的就是用于分析间接访问运算符前面部分信息，即存储于 `CurrentVar` 栈中的当前变量的相关信息。然后，这些信息（主符号、偏移等）传递也就到此为止了，并不需要继续向后传递。而间接访问运算符之后部分的相关分析仅仅依赖于对 `semantic053` 函数得到的结果操作数的间接寻址访问，以此作为后续分析的基址。

第 19~20 行：获取 `semantic053` 函数所生成的结果操作数，将其赋给临时对象 `TmpOp`，并将 `Operand` 栈顶元素（即结果操作数）弹出。

第 21~34 行：如果 `TmpOp` 为间接寻址操作数，则需要特殊处理。实际上，在这种情况下，若试图获得间接访问运算符后部符号的基址，就必须对 `TmpOp` 进行二重间接访问寻址。不过，绝大多数目标机只支持一次间接寻址，因此，试图在一个操作数中表示二重间接访问寻址并不是一个令人满意的解决方案。这里，不得不借助于一个临时指针将二重间接访问寻址转化为两次间接寻址的形式，通常，可以生成赋值 IR 完成这一功能。在生成赋值 IR 时，应该特别注意各操作数 `m_bRef` 属性的设置，即源操作数的 `m_bRef` 为真，而目标操作数的 `m_bRef` 为假。

第 35~40 行：设置 `TmpVar` 对象的相关属性，并将其压入 `CurrentVar` 栈。值得注意的是，实际上，此时的 `TmpVar.m_bRef` 属性必定为 `false`，无论原始的状况是 `true` 还是 `false`。如果原始的 `TmpVar.m_bRef` 为真，则必须由第 21~34 行生成一行赋值 IR，以实现一次间接寻址的过程。然而，第 36 行再次将 `TmpVar.m_bRef` 设置为 `true` 的目的是为了表示该操作数是间接访问。

3. 地址运算

地址运算是一种基本的指针运算。实际上，标准 Pascal 并没有对其作非常明确的解释。这主要是因为早期的 Pascal 语言只允许指针指向用户动态申请的存储空间，而不能是用户变量或参数等。不过，鉴于 C 及一些商用 Pascal 编译器的设计，笔者最终采用了“@”符号作为地址运算符，这主要源于 Turbo Pascal 及 Delphi 的设计。

通常，地址运算的安全性是由编译器保证的。当然，它的前提就是地址运算的操作数必须是左值。也就是说，只有左值表达式才可以进行取地址。实际上，这并不难理解。例如，`&(a+b)` 是非法表达式的原因就是 `(a+b)` 不是左值。而地址运算的结果操作数就是一个地址或者指针。实际上，Pascal 的地址运算应用与 C 语言的 `&` 运算符是极其相似的，这里就不再深入讨论了。下面就来看看 Neo Pascal 的相关实现。

【文法 6-7】

因子

→ @ 变量 053 060



程序 6-17 semantic.cpp

相关文法:

因子 → @ 变量 053 060

```
1  bool semantic060()
2  {
3      OpInfo Tmp,Op1,Rslt;
4      if (Operand.empty())
5      {
6          EmitError("语义错误",TokenList.at(iListPos-1));
7          return false;
8      }
9      Tmp=Operand.top();
10     Operand.pop();
11     if (Tmp.m_iType==OpInfo::CONST ||
12         (Tmp.m_iType==OpInfo::VAR && SymbolTbl.IsTmpVar(Tmp.m_iLink) && Tmp.m_bRef==false))
13     {
14         EmitError("地址运算的操作数必须为左值",TokenList.at(iListPos-1));
15         return false;
16     }
17     TypeInfo TmpType;
18     if (Tmp.m_iType==OpInfo::PROC)
19     {
20         if (SymbolTbl.ProcInfoTbl.at(Tmp.m_iLink).m_eType==ProcInfo::Function)
21             TmpType.m_eDataType=StoreType::T_FUNC;
22         else
23             TmpType.m_eDataType=StoreType::T_PROC;
24         TmpType.m_iLink=Tmp.m_iLink;
25         SymbolTbl.AddType(TmpType);
26         Rslt.m_iDetailType=Tmp.m_iDetailType;
27         Rslt.m_iDetailType.push(VarType(TmpType.m_eDataType,SymbolTbl.TypeInfoTbl.size()-1));
28         int i=SymbolTbl.GetTmpVar(SymbolTbl.ProcStack.top(),StoreType::T_POINTER);
29         SymbolTbl.VarInfoTbl[i].m_iTypeLink=SymbolTbl.TypeInfoTbl.size()-1;
30         Rslt.m_iType=OpInfo::VAR;
31         Rslt.m_iLink=SymbolTbl.VarInfoTbl.size()-1;
32         Rslt.m_bRef=false;
33         SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
34             OpType::GETPROCADDR,Op1,Rslt));
35         Operand.push(Rslt);
36         return true;
37     }
38     Rslt=Tmp;
39     TmpType.m_eDataType=StoreType::T_POINTER;
40     if (Tmp.m_iDetailType.empty())
41     {
42         Rslt.m_iDetailType.push(VarType(SymbolTbl.TypeInfoTbl[SymbolTbl.VarInfoTbl[Rslt.m_iLink]
43             .m_iTypeLink].m_eDataType,SymbolTbl.VarInfoTbl[Rslt.m_iLink].m_iTypeLink));
```

```
44     TmpType.m_eBaseType=SymbolTbl.TypeInfoTbl[SymbolTbl.VarInfoTbl[Rslt.m_iLink].m_iTypeLink]
45         .m_eDataType;
46     TmpType.m_iLink=SymbolTbl.VarInfoTbl[Rslt.m_iLink].m_iTypeLink;
47 }
48 else
49 {
50     TmpType.m_eBaseType=SymbolTbl.TypeInfoTbl[Rslt.m_iDetailType.top().m_iLink].m_eDataType;
51     TmpType.m_iLink=Rslt.m_iDetailType.top().m_iLink;
52 }
53 TmpType.m_szName="_noname";
54 TmpType.m_szName.append(GetSerialId());
55 SymbolTbl.AddType(TmpType);
56 Rslt.m_iDetailType.push(VarType(StoreType::T_POINTER,SymbolTbl.TypeInfoTbl.size()-1));
57 Rslt.m_bRef=false;
58 Rslt.m_iType=OpInfo::VAR;
59 if (!Tmp.m_bRef)
60 {
61     Op1=Tmp;
62     int i=SymbolTbl.GetTmpVar(SymbolTbl.ProcStack.top(),StoreType::T_POINTER);
63     SymbolTbl.VarInfoTbl[i].m_iTypeLink=SymbolTbl.TypeInfoTbl.size()-1;
64     Rslt.m_iLink=SymbolTbl.VarInfoTbl.size()-1;
65     if (SymbolTbl.IsVarPara(SymbolTbl.VarInfoTbl[Op1.m_iLink].m_szName,SymbolTbl
66         .VarInfoTbl[Op1.m_iLink].m_iProcIndex))
67     {
68         SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
69             OpType::ASSIGN_4,Op1,Rslt));
70     }
71     else
72     {
73         SymbolTbl.ProcInfoTbl.at(SymbolTbl.ProcStack.top()).m_Codes.push_back(EmitIR(
74             OpType::GETADDR,Op1,Rslt));
75     }
76 }
77 Operand.push(Rslt);
78 return true;
79 }
```

第 4~8 行：Operand 栈的有效性检查。

第 9~10 行：将 Operand 栈顶元素赋给 Tmp，并将其弹出。

第 11~16 行：左值判断。在 C、Pascal 中，地址运算的操作数必须是左值表达式。关于左值的判断条件，笔者先前已作了说明。

第 18~37 行：处理对过程取地址的运算。这里必须生成一个指向该过程的指针变量，并生成相关的 IR，用于对过程取地址。

第 38~58 行：实际上，除了过程取地址之外，只可能存在变量取地址的情况，因此并不需要进行额外的判断。对于变量取地址的情况，主要的处理有两部分：



(1) 变量的原类型的变换。即在原类型链的链首增加一个指针结点，用于表示地址运算结果操作数的类型是一个指向原类型的指针。

(2) 根据实际情况，生成地址运算的 IR。

其中，第一部分主要是由第 38~58 行完成的。同时，这里还需要考虑 `m_iDetailType` 栈为空的情况。当然，类型的变化过程也需要在 `m_iDetailType` 栈中予以体现。

第 59~76 行：处理非间接寻址的情况。对于非间接寻址的操作数则需要生成地址运算的 IR。注意，变参的情况是需要特殊处理的。虽然变参并不是指针，但是变参空间内存储的却是实参的地址。因此，在处理变参地址运算时，只需要生成赋值 IR (ASSIGN) 即可。然而，除了变参之外，其他的情况都是需要生成取地址 IR (GETADDR) 的。当然，值得注意的是，对于间接寻址的情况，由于操作数存储的就是目标对象的地址，因此，只需将操作数的 `m_bRef` 设置 `false` 即可（即表示非间接寻址）。

第 77 行：将 `Rslt` 操作数压入 `Operand` 栈。

至此，关于指针运算的相关语义处理已基本讨论完了。处理指针运算的关键就是理解左值、右值的概念，因为并不是任何操作数都是满足运算需求的。

6.6 深入学习

类型理论是一个非常复杂的话题，它也一直是计算机科学理论的重要研究领域。类型对于程序设计语言、编译技术的研究是至关重要的。这里，笔者推荐几本类型理论相关的经典著作，供学有余力的读者参考使用。

- | | | |
|-----------------------------------------------------|--------------------|-----------|
| 1、Advanced Topics in Types and Programming Language | Benjamin C. Pierce | MIT Press |
| 说明：这是类型理论方面的经典著作，不过，由于难度极高，不适合初学者使用。 | | |
| 2、Type Theory and Functional Programming | Simon Thompson | |
| 说明：书中提到了类型理论与函数式语言的相关话题，学习难度较大。 | | |
| 3、程序设计语言理论基础 | John C. Mitchell | 电子工业出版社 |
| 说明：这本书是程序设计语言领域的著作，适合初学者阅读。 | | |
| 4、程序设计语言的形式语义 | Glynn Winskel | 机械工业出版社 |
| 说明：这本书是国内外许多高校形式语义学的教材。 | | |
| 5、计算机语言的形式语义 | 陆汝钤 | 科学出版社 |
| 说明：陆汝钤院士是国内形式语义领域的权威，该书完整地阐述了形式语义相关理论与概念，是不可多得经典著作。 | | |

6.7 实践与思考

1. 与 C 语言不同，Pascal 语言并不允许数组与指针之间进行类型转换，试问如何修改与完善 Neo Pascal，使之可以支持数组与指针之间的类型转换？

2. 请结合 C++ 的 `set` 模板，完善 Neo Pascal 的集合类型。

3. Neo Pascal 的类型转换类型并不完整，请读者根据实际需要予以补充。

4. 试评价 Neo Pascal 类型系统的不足之处。

5. 与 C 语言不同，Pascal 语言并不支持逻辑表达式的短路，试问如何修改与完善，使之可以支持逻辑表达式的短路？

6.8 大师风采——Alan Kay

Alan Curtis Kay: 美国计算机科学家，面向对象程序设计思想的提出者，Smalltalk 创始人之一。1940 年 5 月 17 日出生于美国。1966 年，获得犹他州立大学工程学院硕士及博士学位。在此期间，他与 Internet 先驱 Ivan Sutherland 一起工作。

1970 年，Kay 加入了 Xerox 公司的 Palo Alto 研究中心，从事网络工作站原型的研究与开发。他的研究被苹果公司商用化，用于 Macintosh 及 Lisa。

当然，Kay 最具影响力的研究恐怕就是提出了面向对象程序设计思想。可以毫不夸张地说，这种思想影响了整整一代程序员。至于 Smalltalk 语言，严格地说，并不是 Kay 个人的成果，而是由一个团体共同完成的。

1984 年，Kay 加入苹果公司。2001 年，他创建了 Viewpoints Research Institute，从事儿童学习研究和先进软件开发。

由于面向对象程序设计思想及对 Smalltalk 语言的贡献，Kay 荣获 2003 年的图灵奖。

第7章 优化技术

Control over the use of one's ideas really constitutes control over other people's lives; and it is usually used to make their lives more difficult.

——Richard Stallman

7.1 优化概述

7.1.1 什么是优化

优化（Optimization）是现代编译技术中一个非常重要的话题。仅就研究现状而言，随着其他阶段的理论研究与实现技术的逐步完善，有理由相信优化技术将是评价未来编译器优劣的唯一标准。

经过先前章节的学习，读者应该知道 IR 生成及目标代码生成都是一种机械的翻译过程，它们更多考虑的是如何实现等价地转换，而往往忽略了目标代码的品质因素。随着编译器的广泛应用，人们似乎已经不能接受这种平凡的翻译过程了。无论何时何地，目标程序的执行效率仍然是程序员关注的重要指标。由此，编译器设计者引入了编译优化的概念，试图通过一些特殊的算法使得编译器产生更优的目标代码。这里的“更优”主要体现在以下三个方面：代码执行的效率、代码执行的所需空间大小、代码本身的大小。一般而言，以上三者之间并不存在矛盾。但有时却并非如此，在一些特殊情况下，即使是程序员可能也无法准确评估并作出最佳抉择，更何况是编译器。举个简单的例子，通常认为指令行数越少（即目标程序所占用空间越小），则执行效率就必定越高（不考虑循环跳转）。不过，对于某些目标机而言，不同指令所需的机器周期可能不同。在这种情况下，代码执行效率及其本身所占用空间之间的关系可能是非常微妙的，甚至会出现一种矛盾的状态。实际上，在繁多的体系结构中，这种情况并不罕见。因此，试图从理论上分析论证优化算法的成效可能是比较复杂的。不过，当明确了应用目标后，这个问题就会变得相对容易了。例如，针对当今的 x86 目标机来说，代码执行效率可能是程序员更关心的因素，为此他们往往愿意以牺牲一定的存储空间作为代价。在这种情形下，代码执行效率就是评价算法的最主要因素。然而，对于一些嵌入式系统的编译器而言，情况却恰恰相反。此类目标机的存储空间有限，而用户却并不太关注程序的执行时间，因为嵌入式系统的程序一般是后台运行的。因此，耗用存储空间多少将是评价此类编译器优化算法的主要因素。

经过计算机科学家的不懈努力，无论是理论研究还是实践应用，优化技术都取得了较大的进展。即便如此，仍然不可能达到“最优”的目标，编译器能做到的仅仅是改善代码的性能而已。因此，从来没有一本编译原理的书将“Optimization”译成“最优”。当然，人们也很难使用一个量化的标准来衡量优化成果。对于不同的输入，优化的成果是不可判定的。在

一些特殊的代码情形下，某些优化算法甚至可能降低目标程序的性能。

寄希望于通过优化算法尽可能地改善目标代码的思想是完全可以接受的，但有一条底线是不能突破的，那就是代码的正确性。以牺牲正确性为代价讨论优化算法是没有任何意义的。即使错误仅仅是“理论上”存在，也绝不能忽视。任何优化的前提都是必须保证不会将一个正确的程序转换成不正确的程序。这是优化中一个非常重要的原则——保守安全原则。实际上，为了达到这一目的，编译器设计者不得不面临许多复杂的问题，例如，数据流分析、控制流分析、别名分析、依赖分析等。这些分析算法本身并不会改善代码的性能，却为许多优化算法提供了非常重要的信息资源。在某些情况下，分析算法的完善可能会对优化算法产生极其深远的影响。由于两者之间的关系非常密切，在现代编译技术中，经常将两者都作为优化技术讨论的话题。

下面，通过一个 Neo Pascal 的实例分析，让读者对优化有一个感性的认识。如表 7-1 所示，仔细观察 B、C 两列的 IR，不难发现，C 列的代码品质、执行性能较 B 列有显著的改善。就本例而言，已经得到了一个理论最优解，由此可见，优化的意义是毋庸置疑的。

表 7-1 IR 优化实例分析

(A) 输入源程序	(B) 优化前 IR	(C) 优化后 IR
...	0:(ByteToInt , 10 , null , _T0)	0:(PARAM, 10 , null , I)
var	1:(ASSIGN_4 , _T0 , null , J)	1:(CALL, AA , null , null)
j,k,p:integer;	2:(ASSIGN_4 , J , null , K)	
a:integer;	3:(ADD_4 , K , J , _T1)	
...	4:(ASSIGN_4 , _T1 , null , P)	
begin	5:(PARAM , P , null , I)	
j:=10;	6:(CALL , AA , null , null)	
k:=j;		
p:=k;		
aa(p);		
end.		

最后，简单讨论一下关于优化可行性的话题。读者应该知道，任何算法的执行都需要付出时间及空间的代价。人们通常能够容忍在编译阶段耗费时空资源来改善代码的品质，最终达到相对最优的运行效果。不过，这并不是绝对的。编译器设计者通常需要考虑以下两个问题：

第一，如果优化所付出的代价无法或很难用运行时的收益来弥补，那么，该优化算法的可行性是值得商榷的。

第二，根据硬件环境及编译模型进行可行性分析。对于有些编译模型而言，花费大量的时间或空间进行优化可能是无法接受的。尤其是在设计动态编译器时，算法的时空耗费可能是决定算法可行性的最关键因素。

7.1.2 优化级别

根据一些经典编译器的设计经验，编译优化往往不是一遍完成的。理想的状态是，在编译过程中每遍代码生成后，都存在相应的优化环节，尽可能保证得到相对最优的中间形式。这种优化组织方式的优点就在于能够针对不同的代码形式选用最适合的优化算法，以达到相



对最优的结果。

通常，编译过程主要涉及三种代码形式：输入源程序、IR、目标代码。一般来说，针对这三种形式，就需要三个优化环节，或称为“三遍优化”。当然，这里并不局限于编译器的优化。广义而言，程序员对输入源程序的改善也可以视为一种优化，是一种基于源程序的人工优化。例如，完全可以通过运用一些编程技巧来节省程序运行的时空消耗，其中，最经典的例子可能就是两个整型变量的原地交换算法。当然，软件工程却不提倡这种编程习惯，它可能会大大降低程序的可读性、可维护性。因此，编译器设计者并不能对输入源程序的性能抱有太大的希望，死代码、死变量、不可到达代码可能是无处不在的。

真正的编译优化主要包括两个级别，即 IR 优化与指令级优化。这种划分的依据是优化算法对目标机的依赖程度。所谓“IR 优化”指的就是那些与目标机特性无关的优化算法，有时亦可称为“机器无关优化”（machine-independent optimization），它的优化对象就是 IR。而所谓“指令级优化”指的就是那些与目标机特性相关的优化算法，有些书籍将其称为“目标代码优化”或“机器相关优化”（machine-dependent optimization），它的优化对象就是目标代码。在优化技术中，IR 优化的理论与技术相对成熟，早期，编译器设计者在这个领域研究中投入了大量的精力，提出许多著名的 IR 优化算法。而指令级优化的发展稍晚些。当然，随着一些新型编译器模型的提出，指令级优化已经逐渐成为一个崭新的研究领域。

本章将着重讨论 IR 优化，关于指令级优化的话题，将在后续章节中详述。

7.2 控制流分析

7.2.1 流图与基本块

输入源程序的逻辑可能是非常复杂的，是编译器设计者无法预知的。语法分析阶段获得的是输入程序的语法结构，而语义处理阶段获得的是符号表以及 IR。但是，仅依据这些信息仍不足以让编译器一窥输入源程序的全貌。无论是 HIR 还是 LIR，它们为编译器提供的关于输入程序逻辑结构的信息是非常有限的。那么，编译器到底希望得到哪些信息呢？回答这个问题并不容易，在理想情况下，编译器希望得到关于输入源程序的一切信息，无论是有用或无用。不过，这仅仅是一种理想状态。根据一些经典编译器的设计经验，发现有两类信息对于优化是非常重要的，即数据流、控制流。这里，笔者简单解释一下相关概念。

数据流（data flow）。就是用于描述数据处理相关的全局信息，例如，一个变量在哪些 IR 指令中被赋值，而在哪些 IR 指令中被引用等。这些信息对于优化算法是非常重要的，编译器通常在实施优化前，必须分析获得这些信息，否则讨论优化可能是不安全的。很多算法都是基于数据流分析的结果进行代码优化的。

控制流（control flow）。主要用于描述程序结构的相关信息。一般来说，控制流分析将完成两项工作：循环结构分析、流图构造。在编译技术中，有些优化算法是基于循环结构分析实现的，例如，循环优化、循环倒置等。除此之外，构造流图供数据流分析及其他优化算法使用也是非常必要的。由于 Neo Pascal 不涉及循环优化算法，所以并不需要进行循环结构分析。本节的重点就是讲解流图的概念及其构造。

流图（flow graph）。是程序结构的有向图描述形式。它的形式与程序流程图类似，却又

比程序流程图的结构更为清晰明了。与程序流程图相比，流图更关注程序的结构描述。例如，流图关注的是选择结构有几个分支及其流向，可能并不关心具体哪个是真（假）分支。在实际编译器中，流图只是优化、代码生成的一种辅助工具，通常，可能并不需要显式地输出，因此，很多时候，流图对程序员而言是完全透明的。图 7-1 是一段 IR 实例，它的流图描述形式如图 7-2 所示。下面，结合图 7-2 介绍流图的主要组成。

基本块(basic block)。在流图中，使用方框表示一系列动作，通常将其称为“基本块”。基本块内的动作是自上而下顺序执行的，不存在任何跳转。有时，为了更突出程序结构的特点，也可以将基本块退化为一个结点形式。

边(edge)。在流图中，使用有向线来表示程序结构中的分支。一般来说，可以将跳转语句分为两类：无条件跳转语句、条件跳转语句。因此，基本块的出度一般不会超过 2，而入度却是不定的。当基本块末语句是条件跳转语句时，该基本块的出度必定为 2。在其他情况下，基本块（除出口基本块外）的出度必定是 1。

```

0: (MR_4      , I, 10, _T0)
1: (JNT      , _T0, null, _L2)
2: (ASSIGN_4 , 1, null, I)
3: (LABEL    , _L5, null, null)
4: (LE_4     , I, 10, _T8)
5: (JNT      , _T8, null, _L4)
6: (ADD_4    , I, 1, J)
7: (LABEL    , _L6, null, null)
8: (ADD_4    , I, 1, I)
9: (JMP      , _L5, null, null)
10: (LABEL   , _L4, null, null)
11: (JMP     , _L3, null, null)
12: (LABEL   , _L2, null, null)
13: (ASSIGN_4 , 1, null, J)
14: (LABEL   , _L3, null, null)
15: (ADD_4   , I, J, I)
    
```

图 7-1 IR 实例

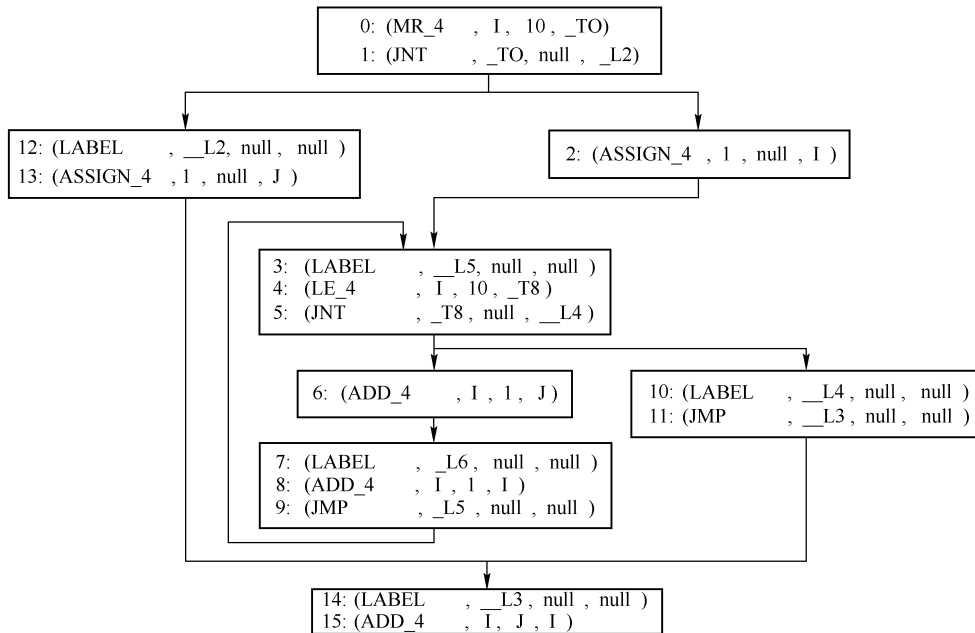


图 7-2 与图 7-1 对应的流图形式

入口基本块(entry basic block)。流图必定存在一个唯一的入口，即程序首语句所在的基本块，入口基本块的入度必定是 0。不过，值得注意的是，入度为 0 的基本块却不一定是入



口基本块。实际上，通常将入度是 0 的基本块（除入口基本块外）称为不可到达代码。也就是说，在任何情况下，它们都无法被调用执行。

出口基本块（exit basic block）。出口基本块的数量与源语言的特点有着密切的关系。由于源语言的不同，流图的出口基本块的数量是不同的。例如，C 语言的函数允许使用 return 语句直接返回，因此，C 语言函数的出口与实际 return 语句的数量有关。而标准 Pascal 语言并不允许使用 return 语句返回，这种情况下，Pascal 的出口就是唯一的。根据一些经典编译器设计的经验，人们发现当流图的出口不唯一时，许多优化问题就变得非常复杂了。为了降低优化的实现难度，通常会为流图添加一个冗余基本块，并且令原图中所有的出口基本块添加一条流向该冗余基本块的边。这样，该冗余基本块就成了流图的唯一出口。有些书将该冗余基本块称为“扩展基本块”（extended basic block）。由于 Pascal 函数的出口是唯一的，所以并不需要建立扩展基本块，这里就不再深入讨论。

流图主要描述的对象是 IR 或目标代码。理论上讲，一张流图或许可以描述一个完整的程序。不过，却很少进行这样的尝试。通常，编译器是以过程为单位生成流图的，因为这样构造得到的流图规模适中，便于其他优化算法使用。

事实上，构建流图的关键就在于划分基本块。下面，就来看看如何将一个 IR 序列划分成若干基本块。

划分基本块可以遵循如下算法：

(1) 首先，确定基本块的入口语句。其判定条件如下：

- 1) 程序第一个语句。
- 2) 任意能够由条件跳转语句或无条件转移语句转移到的语句。
- 3) 紧跟在跳转语句后面的语句。

当一个语句满足上述三个判定条件之一时，则该语句即为基本块入口语句。

(2) 从某个入口语句开始，直到下一个入口语句（但不含该入口语句）或程序结束语句之间的所有语句即组成了一个基本块。

完成基本块划分后，只需根据基本块末的语句的类型构建流图的边即可。如果块末语句为无条件跳转时，则构造一条指向目的标号所在基本块的边。如果块末语句不是跳转语句时，则构造一个指向后继语句所在的基本块的边即可。这个过程并不复杂，读者参考图 7-2、图 7-3 分析构造过程，这里不再详细讲述。

7.2.2 流图的数据结构

前面，已经详细介绍了流图的概念及其构造方法。在本小节中，将开始分析 Neo Pascal 中相关源代码的实现。

下面，先来看看 Neo Pascal 中基本块的结构定义。

【声明 7-1】

```
typedef struct CBasicBlock
{
    int iStart;           //基本块入口语句位序号
    int iEnd;           //基本块出口语句位序号
    vector<int> DownFlow; //当前基本块的后继基本块集合
```

```

vector<int> UpFlow;           //当前基本块的前趋基本块集合
CBits *Def;                 //定值变量集合
CBits *Use;                 //引用变量集合
CBits *InSet;              //入口变量集合
CBits *OutSet;             //出口变量集合
};

```

iStart、iEnd 两个属性是用于描述基本块入、出口语句的位序号。由于 Neo Pascal 的 IR 是以顺序线性表形式存储的，所以只需两个数值类型的变量足以描述基本块入、出口。

DownFlow、UpFlow 两个属性用于描述基本块之间的关系，即流图中的边。由于流图是一个有向图，有读者可能认为根本不需要使用两个属性描述流图的边，而只需使用 DownFlow 描述其后继基本块即可。不过，这却不是一个好主意。实际上，很多优化算法不仅需要关心当前基本块的后继基本块，有时也需要了解当前基本块的前趋基本块。所以，比较理想的基本块结构一般都需要存储其前趋基本块的信息。

这里，值得读者注意的是基本块的结构定义很大程度上取决于 IR 的组织形式。例如，编译器设计者使用链式结构组织 IR，那么，基本块的结构也就可能复杂些。

剩下的 4 个属性都是用于描述基本块数据流相关信息的，在此，读者并不需要深入研究。

下面，再来看看流图的结构定义。实际上，流图就是基本块的集合。在 Neo Pascal 中，流图是以过程为单位生成的，也就是说，一张流图描述的是一个过程（函数）的控制结构，但并不描述过程之间的依赖关系。因此，针对整个输入程序的控制流分析可能会生成多张流图。在 Neo Pascal 中，流图的结构声明如下：

【声明 7-2】

```
map<int,vector<CBasicBlock>> BasicBlock; //流图的结构定义
```

这是一个 map 表结构，它存储的是整个输入程序的流图集合。其中，一个表项即表示一张流图，以过程序号为关键字进行索引。

7.2.3 流图的构造

在本小节中，将讨论 Neo Pascal 的流图构造算法。经过前面的讲解，读者已经知道构造流图的关键就在于分析基本块的入口。根据先前的分析，算法的逻辑是比较明确的，而实现的难度也应该相对较低。

不过，在编译过程中，构造流图的次数将是非常可观的，并不是通常认为的一次或者几次。实际上，任何优化算法都是针对某种或者几种特定的情景进行分析与优化的。然而，各种优化算法的执行必定存在先后次序。后续的优化算法执行完毕后，可能导致代码情景发生一定变化。这种情况下，有可能为先前已执行完毕的某些优化算法创造了新的条件。一般而言，优化算法需要经过多次迭代，直至达到一个相对稳定的状况。在此过程中，由于代码情景的不断变化，经常需要重新构造流图。因此，编译器设计者不得不考虑流图构造算法的性能。

Neo Pascal 的流图构造算法如程序 7-1 所示。



程序 7-1 DFA.cpp

```
1 void CDataFlowAnalysis::GetBasicBlock(int iProcIndex)
2 {
3     vector<CBasicBlock> TmpBasicBlocks; //临时流图
4     map<int,int> Lbl_Block; //散列表, 描述记录标号及其 IR 行号的相应关系
5     int iStart=0; //基本块入口
6     map<int,int> ReWrite; //重写散列表
7     int i;
8     for(i=0;i<SymbolTbl.ProcInfoTbl[iProcIndex].m_Codes.size();i++) //遍历 IR 列表
9     {
10        CBasicBlock TmpBb;
11        IRCode * TmpIR=&SymbolTbl.ProcInfoTbl[iProcIndex].m_Codes[i]; //获取当前 IR
12        Opti_Tbl* TmpOpti=SearchOptiTbl(TmpIR->m_eOpType); //获取处理方案
13        if (TmpOpti!=NULL && TmpOpti->eJumpType!=Opti_Tbl::None)
14        {
15            switch(TmpOpti->eJumpType)
16            {
17                case Opti_Tbl::Lbl: //当前 IR 是标号
18                {
19                    if (i!=0 && i!=iStart) //判断是否构成了基本块
20                    {
21                        TmpBb.iStart=iStart; //设置基本块的入口
22                        TmpBb.iEnd=i-1; //设置基本块的出口
23                        //如果当前 IR 不是最末语句, 则设置后继基本块
24                        if (i<SymbolTbl.ProcInfoTbl[iProcIndex].m_Codes.size())
25                            TmpBb.DownFlow.push_back(TmpBasicBlocks.size()+1);
26                        TmpBasicBlocks.push_back(TmpBb); //将临时基本块加入临时流图中
27                        iStart=i; //设置下一基本块的入口
28                    }
29                    //将标号与 IR 行号的对应关系加入 Lbl_Block 中
30                    Lbl_Block.insert(pair<int,int>
31                                    (TmpIR->m_Op1.m_iLink,TmpBasicBlocks.size()));
32                };
33                break;
34                case Opti_Tbl::CondJump: //当前 IR 是条件跳转
35                case Opti_Tbl::NonCondJump: //当前 IR 是非条件跳转
36                {
37                    TmpBb.iStart=iStart; //设置基本块入口
38                    TmpBb.iEnd=i; //设置基本块出口
39                    //获取跳转的目的标号
40                    int iLbl=(TmpOpti->eJumpType==Opti_Tbl::CondJump)?
41                        TmpIR->m_Rslt.m_iLink:TmpIR->m_Op1.m_iLink;
42                    //按标号检索 Lbl_Block
43                    map<int,int>::iterator it=Lbl_Block.find(iLbl);
44                    if (it!=Lbl_Block.end()) //检索成功
45                        TmpBb.DownFlow.push_back(it->second); //建立与后继块的关系
```

```

46         else
47             //设置 ReWrite 表
48             Rewrite.insert(pair<int,int>(TmpBasicBlocks.size(),iLbl));
49             if (TmpOpti->eJumpType==Opti_Tbl::CondJump) //是否为条件跳转
50                 {
51                     //如果当前 IR 不是最末语句，则设置后继基本块
52                     if (i!=SymbolTbl.ProcInfoTbl[iProcIndex].m_Codes.size()-1)
53                         TmpBb.DownFlow.push_back(TmpBasicBlocks.size()+1);
54                 }
55             TmpBasicBlocks.push_back(TmpBb); //将临时基本块加入流图
56             iStart=i+1; //设置下一基本块的入口
57         }
58         break;
59     }
60 }
61 }
62 if (iStart<i) //设置最末基本块
63 {
64     CBasicBlock TmpBb;
65     TmpBb.iStart=iStart; //设置基本块入口
66     TmpBb.iEnd=i-1; //设置基本块出口
67     TmpBasicBlocks.push_back(TmpBb); //将临时基本块加入流图
68 }
69 //遍历 ReWrite 表，回填基本块的后继关系
70 for(map<int,int>::iterator it=Rewrite.begin();it!=Rewrite.end();it++)
71     TmpBasicBlocks.at(it->first).DownFlow.push_back(Lbl_Block[it->second]);
72 //遍历基本块，根据基本块的后继集合，回填基本块的前趋集合
73 for (int i=0;i<TmpBasicBlocks.size();i++)
74     for (int j=0;j<TmpBasicBlocks.at(i).DownFlow.size();j++)
75         TmpBasicBlocks.at(TmpBasicBlocks.at(i).DownFlow[j]).UpFlow.push_back(i);
76 //将流图加入全局流图集合
77 BasicBlock.insert(pair<int,vector<CBasicBlock>>(iProcIndex,TmpBasicBlocks));
78 //设置当前流图
79 CurrentBasicBlock=&BasicBlock[iProcIndex];
80 }

```

在 Neo Pascal 中，笔者并没有设置独立的控制流分析模块，而是将流图构造算法置于数据流分析模块中。下面，来详细分析流图构造算法的源代码实现。

判定基本块入口是本算法的关键所在，不过，这并不是算法的难点。难点在于如何在一遍扫描中完成基本块的识别及块间关系的分析。实际上，使用两遍扫描分别完成这两项工作是非常简单的。不过，这却不是一种高效的方案。那么，一遍扫描完成流图分析的难点何在呢？当跳转语句的目的标号出现在跳转语句之后，算法就很难获取两者之间的关系。Neo Pascal 采用了一种回填的机制，即实现了一遍扫描构造流图的算法。整个程序主要由三部分组成：遍历 IR 列表、回填基本块后继关系、回填基本块的前趋关系。

第 8~61 行：这个循环主要用于遍历 IR 列表，根据 IR 的操作符，判断基本块的入口。



这里，Neo Pascal 设置了一张二维表 `opti_Tbl`，这张表中记录了各种 IR 操作符在各类优化算法中所需执行的动作，也就是“处理方案”。无论是构造流图或者是其他优化算法，它们的工作对象都是 IR。通常，算法需要对 IR 操作符进行判断，并执行相应的处理方案。下列程序就是 `Opti_Tbl` 的声明形式，而该表的初始化值请参见 `Opti_Tbl.h` 文件，这里就不再逐一列出。

【声明 7-3】

```
struct Opti_Tbl
{
    OpType Op;                //IR 操作符
    int eJumpType;           //跳转类型
    int eExpType;            //表达式类型
    int eConstPropType;     //常量传播类型
    int eOpToken;           //单词类型
    int eDeadCode;          //死代码操作类型
    int eAlgebraicProcess;   //代数简化类型
    int eCommutative         //是否满足交换律
}
```

第 11、12 行：以 IR 操作符为关键字检索处理方案表，检索成功时，则返回相应的 `Opti_Tbl` 结构，否则返回 `null`。这里需要注意一点，从设计的初衷来说，`Opti_Tbl` 表的记录与 IR 操作符是一一对应的，也就是说，检索永远不会失败。但事实上，有些优化算法根据需要只为每一大类操作符设置一个表项，以便将一些雷同的操作符归类处理。因此，需要考虑检索失败的情况。

第 13 行：根据检索得到的处理方案的 `eJumpType` 属性，完成相应的处理动作。`eJumpType` 属性用于描述 IR 的跳转情况，其可能的取值包括：`None`、`Lbl`、`CondJump`、`NonCondJump`。在流图构造算法中，需要关心的 IR 操作符主要是 3 类，即标号、条件跳转、无条件跳转。至于其他的任何 IR 操作符都不是算法所关注的，因为它们不可能是基本块的入口，所以其他的 IR 操作符相应的 `eJumpType` 取值为 `None`，表示无需处理这类指令，直接跃过即可。

第 15 行：根据 `eJumpType` 的取值，分别进行相应的处理。

第 17~33 行：处理标号类型 IR。标号是基本块的可能入口。除第一行是标号 IR 的情况之外，出现标号就意味着其上一行 IR 应该就是某个基本块的结束，标号本身又是新基本块的开始。而这两个基本块的关系也非常明确，即前者是后者的前趋。不过，值得注意的是，需要记录标号及其所标识的基本块的序号关系（登记在 `Lbl_Block` 表中），以便处理以该标号为目的的跳转语句。

第 34~58 行：处理跳转类型 IR。在 Neo Pascal 中，包括 `JMP`、`JNT`、`JT`、`JE_1`。实际上，区别无条件跳转、条件跳转的目的就是确定基本块之间的关系。由于 Neo Pascal 的过程（函数）是不可能以跳转类型 IR 结束的，因此出现跳转语句就意味着一个基本块的结束，而其后的一句 IR 就是一个新基本块的入口。这两个基本块的关系就依赖于 IR 的跳转类型，如果是无条件跳转，那么这两个基本块之间就可能不存在关系（除非后者的首行 IR 正好是跳转的目标标号）。如果是条件跳转，则前者将是后者的前趋基本块。当然，对于跳转类型 IR

的另一个处理动作就是建立跳转 IR 所在基本块与跳转目的基本块的关系。这一过程需要考虑两种情况，即向前跳转、向后跳转。向前跳转的情况比较容易，由于算法是顺序遍历 IR，所以目的标号必定已经识别，并登记在 Lbl_Block 表中，只需检索获取标号对应的基本块序号建立关系即可。而向后跳转的情况复杂一些，由于无法预知目标标号所在的基本块，因此，仅能将其记录在 ReWrite 表中，以便遍历后续程序回填相应的关系。当然，在不考虑效率的情况下，也可以将此过程分为两遍处理。

第 62~68 行：主要用于处理最末一个基本块。因为最末一个基本块的结束可能无法通过标号或者跳转标识，一般需要在遍历完毕后进行特殊处理。

第 70~71 行：遍历 ReWrite 表回填所有 DownFlow 信息。此时，已经不需要考虑目的标号不存在等情况。即使是输入源程序存在一定的错误，应该已经在语义分析阶段完成出错处理了。任何优化算法都是建立在 IR 列表绝对正确的前提下讨论的。

第 73~75 行：根据基本块的 DownFlow 集合的信息，填写相应的 UpFlow 集合。笔者觉得这是一个非常简单的处理过程。当然，可以将其集成在整个 IR 遍历过程中完成，但那样做可能会使问题变得稍复杂。

第 77 行：将该过程（函数）的流图放到流图集合中。前面已经说过，分析算法是以过程（函数）为单位构造流图，因此，一个完整的程序通常可以存在若干张流图。

7.2.4 优化的分类

在编译技术中，通常将优化算法分为三类：过程间优化、过程内优化、块内优化（即基本块内优化）。

下面先来看看过程内优化、块内优化。在很多编译原理的书籍中，将前者称为“全局优化”，而将后者称为“局部优化”。相对而言，局部优化的实现难度要远低于全局优化，从理论上来说，局部优化能达到的优化效果都可以由相应的全局优化实现。那么，是否意味着局部优化已经失去了存在的价值呢？答案是否定的。实际上，在某些场合，考虑到全局优化的实现代价或可行性等因素，编译器设计者更愿意选择局部优化而不是全局优化。

绝大部分全局优化都是基于数据流、控制流实现的，这也是它与局部优化最主要的差异。数据流分析是一项比较复杂的工程，尤其是处理指针、引用等元素时，可能很难精准地分析其引用、定值情况。当然，有时为了便于实现，一些消极的分析方法也是可以接受的。

相对而言，局部优化则较少依赖于数据流信息。例如，基于 DAG 的代码重写就是一个典型的局部优化算法，它的优化过程几乎很少需要数据流、控制流信息的支持，但它却依然能够实现块内的常量折叠、常量传播、复写传播等优化效果。对于那些不考虑数据流、控制流分析的小型编译器而言，基于 DAG 实现的代码重写算法是一个不错的选择，它的优化效果也是令人满意的。而这种代码重写算法的实现代价视具体 IR 形式而定，一般来说，它更偏向于树型结构的 IR。

在早期，由于过程间优化的实现代价及执行效率都相对较大，因此很多研究认为过程间分析、优化都是徒劳的。当然，更多的原因是鉴于过程间优化的效率。20 世纪 80 年代末，Richardson、Ganapathi 就研究分析了过程间优化的作用及其效率，研究发现过程间优化的成效是显著的，但是其效率却是相对较低的，常常使编译速度大幅度降低。同时，发现具有一般意义的过程间优化是相对复杂的，对编译器的设计与实现提出了较高的要求。因此，在经



典编译技术中，很少涉及过程间优化的话题。不过，过程间优化在并行编译领域的价值是令人兴奋的。

Neo Pascal 涉及的 IR 优化算法大多属于全局优化。由于全局优化需要处理大量复杂的数据流分析，所以数据流分析将是本章的一个重要话题。下面，将详细讲解数据流分析算法及其实现。

7.3 数据流分析

7.3.1 数据流的相关概念

实际上，数据流的概念在很多学科中都有论述，并不是编译技术的专有名词。当然，在不同的应用领域中，数据流的含义也存在较大的差异。即使是同一学科的两个专业，对数据流的理解也不尽相同，读者千万不要将本书讨论的数据流与软件工程中的数据流混淆。在讨论数据流的概念之前，先引入几个相关的概念。

定值点：变量 A 的定值点是一句 IR，而执行该句 IR 可能会对 A 的值产生影响，那么，该 IR 的位置称为 A 的定值点。最常见的定值点就是对 A 的赋值，当然，必须考虑指针、引用等对变量可能产生的定值。在整个程序范围内，一个变量的定值点可能是非常多的。

引用点：变量 A 的引用点是一句 IR，而该句 IR 可能会引用变量 A 的值。那么，该 IR 的位置称为 A 的引用点。同样，也必须考虑指针、引用等机制带来的可能引用情况。

引用一定值链（ud 链）：假设在程序中某点 u 引用了变量 A 的值，则把能到达 u 的 A 的所有定值点的集合称为变量 A 能够到达 u 引用点的引用一定值链。

定值—引用链（du 链）：假设在程序中某点 d 是变量 A 的定值点，则把该定值能到达的对 A 的所有引用点的集合称为变量 A 的 d 定值点的定值—引用链。

活跃变量（live variable）：亦称活变量。对程序中的变量 A 和某点 p 而言，如果存在一条从 p 开始的通路，其中引用了 A 在 p 点的值，则称 A 在 p 点是活跃的。讨论活跃变量的主要意义就在于删除死变量、死代码，以节省存储空间的消耗。死变量可能是用户声明，也可能是编译器产生的临时变量。

7.3.2 数据流分析的策略

数据流分析（data-flow analysis）指的是一组用来获取有关数据如何沿着程序执行路径流动变化的相关信息的技术，这些信息对于优化算法设计是极其重要的。从程序执行的角度而言，数据流分析就是把每个程序点和一个数据流值关联起来的过程。而这个数据流值就是该程序点所能观察到的所有程序状态的集合（即程序所有变量值的集合）的抽象形式。同时，希望得到程序中每个位置点上的数据流值。在优化编译器中，数据流分析的主要目的就是在保证安全的情况下定位优化的机会。告知优化算法哪些位置可以进行代码变换，以获得更优的执行效果。

与优化算法类似，通常可以将数据流分析分为两类，即基本块内分析与基本块间分析。由于基本块内的 IR 都是自上而下顺序执行的，根据 IR 的语义，确定每句 IR 的数据流值并不是非常困难的。当然，对于某些支持指针或引用的编译器来说，问题可能会复杂一些。

基本块间的数据流分析是基于流图讨论的。块间分析主要是根据流图建立相应的数据流方程组，并设计迭代算法实现求解方程组的过程。块间数据流分析也称为“全局数据流分析”，是优化技术中一个重要的研究领域。本章将重点讨论这一话题。

无论是块内数据流分析还是块间数据流分析，都不得不面临一个问题，那就是必须严格区分“可能”还是“必然”。

如图 7-3 所示，请问哪些关于变量 j 的定值 IR 能够到达第 5 行的引用处？第 4 行 IR 中 j 的定值是必然能够到达的，这是毋庸置疑的。那么，第 2 行 IR 的 j 的定值是否能够到达呢？实际上，这只是一个可能到达的定值点。如果指针 k 指向变量 j 时，第 3 行 IR 的定值将影响变量 j 的值，那么，此时第 2 行的定值就不可能到达第 5 行。然而，仅依据图 7-3，编译器根本无法判断 k 具体指向的变量。针对这种情况，编译器设计者只能认定第 2 行的定值是无法到达的，否则可能导致优化算法不安全。

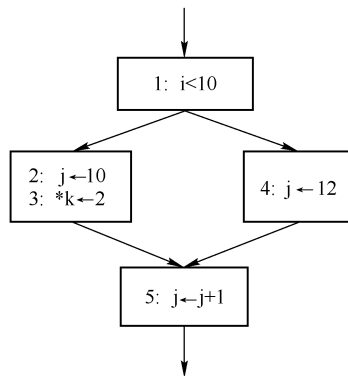


图 7-3 可能定值的流图

当然，读者可能会想到对 k 的引用情况进行分析，这是完全可行的。不过，其分析过程却是比较复杂的，因为设计者不得不考虑二重或多重的指针引用情况。在现代编译技术中，针对指针、引用、别名等语言机制，通常有两种策略：积极的和消极的。所谓“积极”就是对引用、指针、别名进行完整的分析，然后，基于分析结果实现更精准的数据流分析。所谓“消极”就是以一种比较保守的态度处理引用、指针、别名等，充分考虑存在“可能”的条件。相对而言，消极方式的实现要简单一些，它不必准确地进行引用、别名等分析。虽然基于消极方式实现的数据流分析可以被大多数编译器接受，但也不可否认一个事实：这种分析方法必然会影响到结果的精准程度，也不可避免地会对优化效果有一定副作用。

最后，笔者再次强调：安全性是一个原则问题，是不容置疑的。无论使用何种优化策略或分析方法，每一个转换动作执行的前提就是保证转换的正确性。针对“可能”还是“必然”的问题，考虑安全性可能比其他任何因素都重要。优化失效是可以接受的，但不安全的优化却是不能接受的。尤其是在分析一些复杂的指针、引用操作时，这是最容易忽视的。例如，引用参数可能给实参带来的副作用、多重指针的真实目标等都是需要考虑的。

7.3.3 活跃变量分析

全局数据流分析的关键就在于根据流图及基本块内的数据流值来分析得到整个流图各点的数据流值。那么，全局数据流问题的求解过程必定是读者关心的。

根据数据流的实际问题，通常是以数据流方程作为求解与算法实现的依据。事实上，很多数据流问题都可以以方程的形式加以描述。根据实际编译器设计的经验，计算机科学家提出了许多经典的数据流模式，例如，活跃变量分析、到达一定值分析等。其中，活跃变量分析是一个经典实例，它具有深远的现实意义。本小节将重点讨论活跃变量的相关话题。

假设一个基本块 s ，其入口处的活跃变量集合为 $\text{in}[s]$ ，而其出口处的活跃变量集合为 $\text{out}[s]$ 。那么，必定有下式成立：

$$\text{out}[s] = \bigcup \text{in}[i] \quad (\text{其中 } i \text{ 是 } s \text{ 的后继基本块}) \quad \text{【式 7-1】}$$



这个等式的含义就是基本块 s 出口处的活跃变量必定是其后继基本块入口处活跃变量集合的并集。同样，可以将等式描述为某一基本块入口处活跃变量的集合等于其前驱基本块出口处活跃变量集合的并集。

令 use[s]为基本块 s 中所引用的变量集合。注意，这里讨论的“引用”指的是在某一基本块内，某些点引用的变量的定值来自当前基本块外。在数据流问题中，仅关心这类特定的引用，而不需要关心那些来自当前基本块的引用点变量的定值。

同样，令 def[s]为基本块 s 中所定值的变量集合。那么，不难得到下式：

$$\text{in}[s] = (\text{out}[s] - \text{def}[s]) \cup \text{use}[s] \quad \text{【式 7-2】}$$

这个等式的讨论对象就是基本块入、出口处的活跃变量与块内引用、定值的关系。式 7-2 的观点是基本块入口处的活跃变量集合由以下两部分组成：

- (1) 基本块内所引用的变量集合。
- (2) 基本块出口处的活跃的且在块内没有定值的变量集合。

值得注意的是,千万不要误认为 $\text{in}[s]=\text{use}[s]$ ，这样就忽视了 s 的后继基本块对 $\text{in}[s]$ 的影响。有了这两个等式之后，就明确了活跃变量集合在基本块内及块间的变化关系，那么，求解活跃变量问题就变得比较容易了。下面，通过一个实例来分析活跃变量的求解过程。

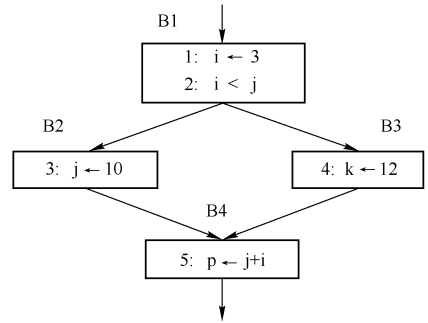


图 7-4 求解活跃分析示例流程图

例 7-1 参考图 7-4，求解该流图的活跃变量集合。

- (1) 求解 use、def 集合。

求解 use、def 集合完全依赖于 IR 的实际语义，本例中并未涉及指针引用等复杂语义。求解的结果如表 7-2 所示。注意：use[B1]中没有 i 的原因是 i 在引用前已在本基本块内被定值，所以 i 不需要加入 use[B1]集合。

表 7-2 与图 7-4 对应的 use、def 集合

基本块	use 集合	def 集合
B1	{j}	{i}
B2	空集	{j}
B3	空集	{k}
B4	{i, j}	{p}

- (2) 求解 in、out 集合。

不难发现，从最后一个基本块开始向前计算可能更容易。假设变量 p 在 B4 后继块中是不活跃的，即 $\text{out}[B4]=\{\}$ 。那么，可以分别得到如下集合：

$$\begin{aligned} \text{in}[B4] &= \text{use}[B4] = \{i, j\} \\ \text{out}[B3] &= \text{out}[B2] = \text{in}[B4] = \{i, j\} \\ \text{in}[B2] &= \text{out}[B2] - \text{def}[B2] = \{i\} \\ \text{in}[B3] &= \text{out}[B3] - \text{def}[B3] = \{i, j\} \end{aligned}$$

$$\begin{aligned} \text{out}[B1] &= \text{in}[B2] \cup \text{in}[B3] = \{i, j\} \\ \text{in}[B1] &= (\text{out}[B1] - \text{def}[B1]) \cup \text{use}[B1] = \{j\} \end{aligned}$$

根据上述计算，可以得到如表 7-3 所示的结果。

表 7-3 与图 7-4 对应的 in、out 集合

基本块	in 集合	out 集合
B1	{j}	{i, j}
B2	{i}	{i, j}
B3	{i, j}	{i, j}
B4	{i, j}	空集

从本例中，不难发现，根据数据流方程，计算活跃变量集合并不复杂。由于图 7-4 是一个无环图，因此，一次迭代即可顺利完成分析过程。如果流图存在环时，迭代过程会复杂一些，可能需要数次迭代才能完成。不过，不必担心迭代无法终止的情形，事实上，计算机科学家已经用数学方法证明了数据流方程是有解的。

最后，读者的疑问可能就是：计算活跃变量集合的意义是什么？在此，笔者稍作简单说明。非活跃变量即死变量，也就是说该变量在该集合所处位置点后不会有任何引用。从优化的角度而言，可以删除基本块内所有对块出口处已死变量的定值操作，因为这类定值是没有任何实际意义的。例如，在图 7-4 中，发现 B4 中对 p 的定值以及 B3 中对 k 的定值都是无意义的。如果将这两条语句删除后再求解一次活跃变量集合，则会发现出现了新的死变量，这就是优化算法通常需要多次迭代的原因所在。事实上，根据图 7-4 所示的流图，最终优化算法可以删除所有的语句。

7.3.4 ud 链与 du 链

ud 链、du 链是数据流分析中两个非常重要的概念。不过，这两个名词却经常被初学者混淆。下面从程序设计语言的角度来诠释这两个概念。

ud 链、du 链仅仅是两个概念的代名词而已，它们揭示了两类不同的信息。读者需要关注的是其背后的概念，而非名词本身。从 IR 定值角度而言，数据流分析需要关心的可能是这次定值会影响哪些 IR 的引用。而从 IR 引用角度而言，数据流分析需要关心的可能是这次引用值的可能定值点的位置。在经典编译技术中，将前者称为 du 链（即定值—引用链），而将后者称为 ud 链（即引用—定值链）。

实际上，du 链是活跃变量分析结果的一种扩展。活跃变量仅关心变量的一次定值后是否存在可能被引用的情况，却不关心具体引用点的位置或存在多少引用点。而 du 链却需要获得关于各个引用点的详细位置，而不仅仅是有或无的回答。通常，du 链是针对某条 IR 而言的，如果该 IR 是定值语句，那么，其 du 链描述的就是本次定值可能到达的所有引用点。同理，ud 链描述的就是一次引用的可能到达的定值点。

与很多数据流问题类似，ud 链、du 链分析的基本原则也是既保守又激进的。以定值问题为例，在很多情况下，一个定值是否能实际到达某一特定程序点是不可判定的。当然，有时还依赖于特定的外部输入。因此，在通常情况下，编译器可以分析得到哪些定值是可能到达的，却很难断言哪些定值是必然到达的。即使如此，在某些特殊情形下，定值、引用问题



仍然是很难断言的。不过，当出现这类情形时，编译器可能将无法保证程序的正确性，虽然这并不是编译器设计者的初衷。

关于 **ud** 链、**du** 链的话题，暂且讨论至此。稍后笔者还将详细阐述 Neo Pascal 的 **ud** 链、**du** 链分析，届时，将介绍 **ud** 链、**du** 链分析算法的基本思想及其实现细节。

最后，来看一个奇怪的 C 语言程序，如图 7-5 所示。如果读者使用 VC++ 编译这个例程，可以发现一个奇怪的问题，那就是 **debug** 和 **release** 版程序的运行结果是不同的。其中，**debug** 版程序的运行结果是 3，而 **release** 版程序的运行结果是 6。究其原因就是数据流分析算法是否将后两句 “*p++=1” 视为对 **b**、**c** 的定值。**debug** 版程序是几乎不做任何优化的，而 **release** 版程序是经过编译优化的。因此，**debug** 版程序即使无法预知对 **b**、**c** 的定值，但按照程序的实际语义翻译后，仍可以得到正确的结果。

```
int test(int a,int b,int c)
{
    int *p=&a;
    a=1;    b=2;    c=3;
    *p++=1; //a的定值
    *p++=1; //b的定值
    *p++=1; //c的定值
    return a+b+c;
}
```

图 7-5 一个奇怪的 C 语言程序

而经过优化的程序则不然，由于数据流分析无法从 “*p++=1” 语句中分析获得任何关于 **b**、**c** 的定值线索，故认为 **test** 函数的返回值就是 1+2+3。如果在此基础上再进行了过程间优化，则编译器就将整个 **test** 函数优化为一个常量 6。读者可以从编译器生成的汇编文件中证实这个事实。由于 C 语言标准中只说明了函数参数的存储布局以及参数取址操作的有效性，却没有明确指出这种指针引用方式的安全性，故不能以此界定优化算法是不正确的。当然，也很难就此认定是程序逻辑的错误，作为程序员来说，能做的就是尽可能避免而已。事实上，这个现象不仅限于参数，同样适用于讨论局部变量。不过，由于不同版本程序的局部变量的存储布局是不一样的，所以较难验证这一结论。在分配 **debug** 版程序的局部变量存储空间时，编译器会加入大量调试信息，以便程序员调试源码。

7.3.5 更多数据流问题

至此已经讲述了数据流分析的两个重要话题。本小节想对数据流问题的一些高级话题作简单说明。先前，仅仅介绍了数据流问题中的两个实例，并没有解释其中的一些细节问题。实际上，有些读者可能会对前面讲述的内容有不少疑问，例如：

- (1) 如何证明迭代算法是收敛的？
- (2) 如何证明数据流方程的解是正确或精确的？

这里只是列了其中的两个问题，实际上，还有一些类似的问题需要回答与论证。在编译技术中，随着计算机科学家对数据流分析的深入研究，提出了一套完整的理论体系——格。它主要是从抽象的角度论证数据流方程的相关论题。这里，只是想让读者消除心中的疑虑，大可不必怀疑先前结论的正确性，因为它们都是经过严格证明的。

还有必要说明一点，本书提到的数据流分析方法是一种最常见的方法，称为“迭代数据流分析”。在现代编译技术中，还有几种数据流分析方法是值得关注的，其中比较经典的就是基于区间的数据流分析方法。这是一种全新的思想与方法，通常需要相应的控制流分析方法配合完成。数据流分析方法的大致思想如下：

- (1) 它将流图划分为各种类型的区域，具体的类型视源语言而定。例如，经典的结构分

析方法就将区域分为八类（块、if-then、if-then-else、case、自我循环、while 循环、自然循环、非正常区间）。

(2) 算法将区域蜕化为一个结点，由此反复多次蜕化，最终得到一个抽象流图（仅包含一个结点）。

(3) 在变换过程中，根据不同类型的区域对流值的变换是有一定规律的，算法对流值施加不同的变换函数，最终得到整个数据流分析结果。

本小节引出了两个数据流分析的相关话题（格、区间分析），它们都是现代编译技术中的经典理论。不过，囿于篇幅，只能简单提及。

7.4 数据流分析的实现

7.4.1 定值点与引用点分析的基础

定值点与引用点的分析是实现数据流分析的基础，通常涉及的绝大多数数据流问题都是基于这两者讨论的。在经典编译技术中，关于定值点、引用点的讨论通常是比较精炼的，并没有太多的理论与技术。确实，基于一种模型语言讨论定值点、引用点的分析是相对比较容易的，它规避了许多现实中的细节问题。然而，对于一门实际语言而言，试图设计一个较完备的分析算法却并不简单，需要关注的问题远比模型语言复杂得多。下面，来谈一些与实际语言相关的定值点、引用点问题。

1. 全局变量

与局部变量不同，全局变量的定值点、引用点可以散布于整个程序范围内。因此，讨论全局变量的定值点、引用点并不是一件容易的事。这应该不难理解，因为任何一次有效的函数调用都可能引用或改变全局变量的值。要准确分析全局变量的定值、引用信息，就必须分析过程、函数的调用关系，只有基于完整的调用关系图才可能分析得到较为精准的结果。这种分析算法就是通常所说的过程间数据流分析。事实上，过程间数据流分析不仅解决了全局变量引用、定值的问题，同时也可以为分析函数对实参的影响提供必要的决策信息。不过，过程间数据流分析也是一项具有挑战性的工程，其难度主要体现在如下两个方面：

(1) 当源语言支持函数指针之类的语法结构时，问题可能会变得异常复杂。

(2) 多文件编译机制使得过程间数据流分析不得不在链接阶段完成。

试图完美地实现过程间数据流分析是需要付出一定代价的。这是因为过程间数据流分析是由链接器完成的，而不是由编译器完成的。其中，最主要的原因就是多文件编译机制的存在。下面，笔者通过一个实例来分析多文件编译对过程间数据流分析提出的挑战。

如图 7-6 所示，这个 C 语言项目包含 a.c、b.c 两个源程序文件，右列则详细描述了项目的编译过程。绝大多数 C 编译器都是依据这个过程完成编译、链接的。首先，编译器分别编译 a.c 和 b.c，并生成相应的汇编文件。然后，由汇编器生成对应的 obj 文件。最后，由链接器将多个 obj 文件链接成一个可执行文件。因此，试图在编译阶段实现准确的过程间数据流分析可能是徒劳的。例如，在编译 a.c 时，编译器根本不可能预测 aa() 函数对全局变量的影响，也无法得到 aa() 函数内部的调用关系图。在整个处理过程中，只有链接阶段才能对整个项目有一个全局的概括。因此，也只有链接器才可能实现真正的过程间数据流分析。不

过，obj 文件中存储的是二进制机器码及重定位、符号等信息，故仅仅依据 obj 文件的信息，试图在链接器中实现数据流分析是非常复杂的。因此，一些商用编译器开发商自主研发汇编器及链接器的一个重要原因就是试图让链接器可以从一些由编译器生成的格式文件中获取更多关于输入源程序的信息，而不仅仅依赖于 obj 文件。以微软的编译器为例，它提供了一个“/GL”选项，有兴趣的读者可以参阅 MSDN 相关文档，其中明确说明了启用全程序优化（即过程间优化）前后，生成的 obj 文件是完全不同的。

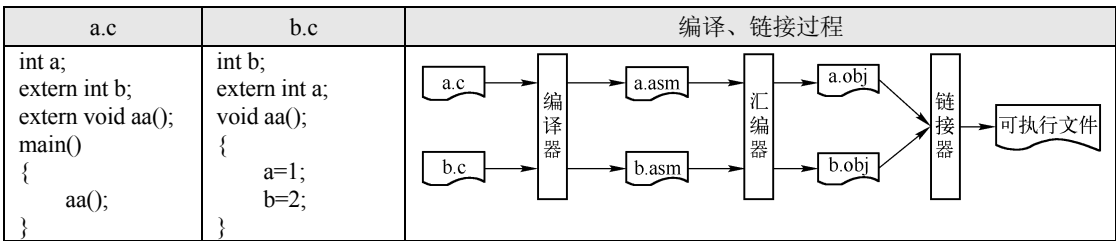


图 7-6 多文件编译示意图

鉴于语言的特性，Neo Pascal 并没有考虑过程间数据流分析，只是用了一个消极的策略，除了普通的定值、引用形式之外，假设每一个有效的过程、函数调用点都可能对全局变量定值、引用。虽然这种处理方式并不够完美，但也是工程领域可以接受的。

2. 内嵌汇编

一些经典程序设计语言都支持内嵌汇编机制，例如，C、Pascal 等。从表面上来看，内嵌汇编机制的实现似乎是非常简单的，只需将相应的汇编源代码复制到目标代码中即可。不过，事实却远比想象的复杂。Neo Pascal 的处理方式与 GCC 类似，即编译阶段不识别内嵌汇编，而直接将内嵌汇编粘贴到目标代码中。换句话说，编译器根本不需要关注内嵌汇编的语法与语义，而是统一由汇编器处理。不过，相应的问题也就由此产生了，即编译器如何估计内嵌汇编对程序变量的影响。事实上，对于不识别内嵌汇编的编译器来说，是根本不可能预计内嵌汇编对程序变量的定值、引用情况的。在 GCC 中，编译器将这项烦心的工作交由用户完成。也就是说，用户以参数形式显式说明当前内嵌汇编中的定值、引用信息，尤其是对 C 变量的影响。

虽然这种处理方式表面上看起来并不完美，但也不失为一种高效的解决方案。编译器只需根据用户的参数标记定值、引用信息，即使内嵌汇编中存在额外的定值、引用也不必理会。如果因此产生副作用而导致优化不安全，其中的责任完全是由用户承担。当然，不同的编译器对内嵌汇编的处理方式是不尽相同的。有些编译器可能会在编译阶段识别与分析内嵌汇编。在这种情况下，编译器就可以根据得到的汇编源代码的实际情形分析定值、引用信息，例如，Delphi 就采用了这种处理方式。

3. 指针引用

在现代编译技术中，关于指针引用的分析是一个非常复杂的话题。相信对于了解 C 语言指针的读者来说，其中的原因不必多作解释了。事实上，在很多情况下，即使是人工分析指针的引用关系都是困难重重的，更何况是由编译器来完成。早期的 Pascal 规定指针所指向的对象只能是动态申请的，不允许使用“@”运算符对普通变量取地址。这样做的目的也就是为了便于处理指针引用的问题。在这种情况下，编译器不必关注指针与普通变量的差异，只

需一视同仁地分析定值、引用信息即可。因为无论指针如何设置，它指向的对象都是在执行过程中动态申请存储空间的，因此不可能影响用户声明的普通变量。

如果语言提供的指针机制比较完善，即允许指针指向普通变量。那么，这种情况下，分析指针引用或可能引用的信息是非常必要的。在数据流分析中，指针处理的复杂性主要体现在对指针的间接寻址上。例如：

```
(GETADDR      ,I,null,P)
...
(GETADDR      ,J,null,P)
...
(ASSIGN_4     ,I,null,@P)
```

在最后一句 IR 中，形式上似乎是对 P 的定值。不过，由于结果操作数是对 P 的间接寻址，所以其实际定值是对 P 所指向的变量而言的，而不是 P 本身。反而这句 IR 却是 P 的一个引用点，因为间接寻址访问依赖 P 所存储的值（即所指向对象的地址）。那么，编译器又该如何确定哪些变量可能被 P 引用呢？在现代编译技术中，别名分析的话题已经并不陌生了，但是，试图准确分析这个集合仍然不是一件简单的事情。

与指针引用类似，变参传递同样存在间接寻址的问题。当然，试图分析变参引用对实参的影响可能比指针引用的分析更为复杂。变参引用不但需要考虑函数体内的逻辑，还可能涉及过程间的分析。

这个问题，一些研究型的编译器通常很少涉及，即使是商用编译器的实现也并不完美。出于实现代价的考虑，Neo Pascal 并没有采用精准的引用分析策略来解决指针引用的问题，而是采用一种比较保守的策略。例如，假设上例的 IR 序列中不存在其他的取址 IR（即 GETADDR 指令）时，试图精确了解 P 指向的目标变量就必须分析程序的控制流，甚至还不能忽略指针之间赋值带来的副作用。不过，在不考虑“精确”的情况下，这个问题可能就不难解决了。可以肯定地得到一个 P 可能指向的目标变量集（I, J），因为整个 IR 序列中只存在对 I、J 的取址操作。当然，由于指针之间可能存在赋值，因此，安全且保守的做法就是假设所有的被取址变量对于任意指针都是有效的。在上例中，即使第 1 行 IR 的结果操作数不是 P，而是其他的指针变量 R，依然不会影响 P 的可能目标变量集。虽然这种策略并不激进，但它却得到了许多编译器设计者的认可。甚至还被一些早期的商用编译器采用。

7.4.2 定值点、引用点分析的相关数据结构

在分析源代码之前，本小节将介绍几个与定值、引用相关的数据结构。其中，有些数据结构不仅涉及数据流分析的实现，甚至贯穿整个优化过程的设计。

1. 变量映射表

Neo Pascal 的数据流分析主要是基于迭代思想实现的。在优化算法的实现中，将大量涉及位向量（变量表的一种映射形式）的迭代。不过，由于 Neo Pascal 的优化算法都是以过程为单位讨论的，一个过程可能涉及的变量仅为全局变量和本过程内的局部变量，所以没有必要将符号表内的所有变量都映射成位向量形式。建立变量映射表的主要目的就是為了便于使用位向量描述变量集合。变量映射表的声明形式如下：



【声明 7-4】

```
map<int,int> VarMap;
```

在分析每个过程时，根据当前过程的信息，从符号表中提取出当前过程相关的变量集合，主要包含全局变量及当前过程的局部变量，并生成相应的 VarMap。其中，第 1 个整型值用于描述变量在符号表中的位序号，而第 2 个整型值则用以描述变量在新构造的变量集合中的位序号。后续的优化算法大多数都是基于该变量集合讨论的，因此，VarMap 是一个非常重要的数据结构。

2. 定值集、引用集

在编译技术中，定值点、引用点的问题通常是基于变量讨论的。所谓“定值集”就是用于描述变量及其定值点集合的数据结构。在 Neo Pascal 中，定值集的声明形式如下：

【声明 7-5】

```
map<string, vector<int>> VarDef;
```

其中，前者的字符串即为定值集的关键字。实际上，就是变量名的一种符号化形式，其描述形式为“过程位序号\$变量位序号”。例如，过程 aa 和其过程内的局部变量 a 在符号表中的位序号分别为 2、5，那么，a 的关键字就是“2\$5”。

vector<int>存储的就是该变量的所有定值点，也就是定值点 IR 在 m_Codes 中的位序号。

所谓“引用集”就是用于描述变量及其引用点集合的数据结构。在 Neo Pascal 中，引用集的声明形式如下：

【声明 7-6】

```
map<string, vector<int>> VarAllUse;
```

引用集的数据结构与定值数是完全一致的，不再赘述。

3. 定值位向量、引用位向量

前面介绍了定值集、引用集的基本结构。这里再引入两个概念：定值位向量、引用位向量。这两个概念是基于基本块讨论的，每个基本块都有一个定值位向量和一个引用位向量。

所谓“位向量”就是指向量中的每个元素都是一个位（只能取“0”或“1”）。使用位向量而不是集合是为了适应迭代算法的需要。

定值位向量主要是应用位向量的形式描述所属基本块中变量定值的信息。在某些应用场合，相比定值的位置来说，编译器可能更关注的是在某个基本块中有哪些变量被定值了，而不关心详细的定值位置。最典型的是稍后将谈到的活跃变量分析。通常，定值位向量中的每个位元素都与 VarMap 中的一个变量关联。位值为 0，则表示相应的变量在该定值位向量所属的基本块中没有定值，否则表示存在定值或可能定值。

引用位向量与定值位向量的结构基本一致，引用位向量主要是通过位向量的形式描述所属基本块中变量引用的信息。在 Neo Pascal 中，这两个位向量都是隶属于 CBasicBlock 结构的，声明形式如下：

【声明 7-7】

```
struct CBasicBlock
```

```

{
    ...
    CBits *Def;          //定值位向量
    CBits *Use;         //引用位向量
    ...
};

```

由于不同过程的 VarMap 的个数是不同的，所以定值、引用位向量都是在编译过程中动态生成。注意，CBits 是笔者设计的一个位向量类，同时，封装了一些在编译器设计中经常涉及的位向量操作。其源代码的实现并不复杂，读者可以自行参考阅读，这里就不再详细列出。

7.4.3 定值点、引用点分析的实现

前面已经介绍了定值点、引用点相关的一些基本概念及数据结构。而程序 7-2 的目标就是获取相关过程的定值集、引用集、基本块定值位向量、基本块引用位向量等信息。

程序 7-2 DataFlowAnalysis.cpp

```

1 void CDataFlowAnalysis::Def_Use_Analysis(int iProcIndex)
2 {
3     vector<int> RefVar;
4     CBits InitBit(VarMap.size());
5     vector<CBasicBlock*> pBbs=&BasicBlock[iProcIndex];
6     for (int i=0;i<SymbolTbl.ProcInfoTbl[iProcIndex].m_Codes.size();i++) //计算 RefVar
7     {
8         if (SymbolTbl.ProcInfoTbl[iProcIndex].m_Codes[i].m_eOpType==OpType::GETADDR)
9             RefVar.push_back(SymbolTbl.ProcInfoTbl[iProcIndex]
10                 .m_Codes[i].m_Op1.m_iLink);
11     }
12     for(int i=0;i<pBbs->size();i++)
13     {
14         if (!pBbs->at(i).Def)
15             delete pBbs->at(i).Def;
16         pBbs->at(i).Def=new CBits(VarMap.size());
17         if (!pBbs->at(i).Use)
18             delete pBbs->at(i).Use;
19         pBbs->at(i).Use=new CBits(VarMap.size());
20     }
21     for(int i=0;i<pBbs->size();i++)
22     {
23         CBits Def(VarMap.size());
24         CBits Use(VarMap.size());
25         for(int j=pBbs->at(i).iStart;j<=pBbs->at(i).iEnd;j++)
26         {
27             IRCode* TmpIR=&SymbolTbl.ProcInfoTbl[iProcIndex].m_Codes[j];
28             Opti_Tbl* TmpOpti;

```



```
29     if (TmpIR->m_eOpType<0)
30         TmpOpti=SearchOptiTbl(OpType::TypeCast);
31     else if (TmpIR->m_eOpType>>4==21)
32         TmpOpti=SearchOptiTbl(TmpIR->m_eOpType);
33     else if (TmpIR->m_eOpType>>4<21)
34         TmpOpti=SearchOptiTbl((OpType)(TmpIR->m_eOpType>>4<<4));
35     if (TmpIR->m_eOpType==OpType::CALL||TmpIR->m_eOpType==OpType::GETPROCADDR)
36         SymbolTbl.ProcInfoTbl[TmpIR->m_Op1.m_iLink].m_bTmpUsed=true;
37     if (TmpIR->m_eOpType==OpType::CALL || TmpIR->m_eOpType==OpType::CALLPTR)
38     {
39         if (TmpIR->m_eOpType!=OpType::CALL ||
40             SymbolTbl.ProcInfoTbl[TmpIR->m_Op1.m_iLink]
41                 .m_eFlag!=ProcInfo::Extern)
42         {
43             vector<VarInfo>::iterator it=SymbolTbl.VarInfoTbl.begin();
44             for(;it!=SymbolTbl.VarInfoTbl.end();it++)
45             {
46                 if (it->m_iProcIndex!=0)
47                     continue;
48                 if (SymbolTbl.IsTmpVar(it-SymbolTbl.VarInfoTbl.begin()))
49                     continue;
50                 OpInfo TmpOp;
51                 TmpOp.m_iType=OpInfo::VAR;
52                 TmpOp.m_iLink=it-SymbolTbl.VarInfoTbl.begin();
53                 TmpOp.m_bRef=false;
54                 RecordUse(VarMap,Use,Def,TmpOp,RefVar,j,iProcIndex);
55                 if (SymbolTbl.TypeInfoTbl[it->m_iTypeLink]
56                     .m_eDataType!=StoreType::T_PROC &&
57                     SymbolTbl.TypeInfoTbl[it->m_iTypeLink]
58                     .m_eDataType!=StoreType::T_FUNC)
59                     RecordDef(VarMap,Use,Def,TmpOp,RefVar,j,iProcIndex);
60             }
61         }
62     }
63     if (TmpOpti!=NULL && TmpOpti->eExpType!=Opti_Tbl::None)
64     {
65         if (TmpIR->m_eOpType==OpType::ASM)
66         {
67             for(int k=TmpIR->m_Op2.m_iLink;k<=TmpIR->m_Rslt.m_iLink;k++)
68             {
69                 OpInfo TmpOp;
70                 TmpOp.m_iType=OpInfo::VAR;
71                 TmpOp.m_iLink=SymbolTbl.AsmParaTbl[k].m_iLink;
72                 TmpOp.m_bRef=false;
73                 switch (SymbolTbl.AsmParaTbl[k].flag)
74                 {
```

```

75         case AsmPara::RW:
76             RecordUse(VarMap,Use,Def,TmpOp,RefVar,j,iProcIndex);
77             RecordDef(VarMap,Use,Def,TmpOp,RefVar,j,iProcIndex);
78             break;
79         case AsmPara::W:
80             RecordDef(VarMap,Use,Def,TmpOp,RefVar,j,iProcIndex);
81             break;
82         case AsmPara::R:
83             RecordUse(VarMap,Use,Def,TmpOp,RefVar,j,iProcIndex);
84             break;
85     }
86 }
87     continue;
88 }
89 if (TmpIR->m_Op2.m_iType==OpInfo::VAR )
90 {
91     RecordUse(VarMap,Use,Def,TmpIR->m_Op2,RefVar,j,iProcIndex);
92 }
93 if (TmpIR->m_Op1.m_iType==OpInfo::VAR )
94 {
95     RecordUse(VarMap,Use,Def,TmpIR->m_Op1,RefVar,j,iProcIndex);
96     if (TmpIR->m_eOpType==OpType::PARAM && SymbolTbl.VarInfoTbl
97         [TmpIR->m_Rslt.m_iLink].m_bRef)
98         RecordDef(VarMap,Use,Def,TmpIR->m_Op1,RefVar,j,iProcIndex);
99 }
100 if (TmpIR->m_eOpType!=OpType::PARAM &&
101     TmpIR->m_Rslt.m_iType==OpInfo::VAR )
102 {
103     RecordDef(VarMap,Use,Def,TmpIR->m_Rslt,RefVar,j,iProcIndex);
104 }
105 }
106 }
107     pBbs->at(i).Def->copy(&Def);
108     pBbs->at(i).Use->copy(&Use);
109 }
110 for(map<string,vector<int>>::iterator it=VarDef.begin();it!=VarDef.end();it++)
111 {
112     sort(it->second.begin(),it->second.end());
113 }
114 for(map<string,vector<int>>::iterator it=VarAllUse.begin();it!=VarAllUse.end();it++)
115 {
116     sort(it->second.begin(),it->second.end());
117 }
118 }

```

第5行：获取当前过程的基本块信息。



第 6~11 行：遍历 IR 序列，判断 IR 操作数是否为 GETADDR，如果为真，则表示 Op1 所指的变量被取地址。这种情况下，则将该变量的位序号置入 RefVar 中。

第 12~20 行：初始化各基本块的定值位向量、引用位向量。

第 21~109 行：遍历过程的基本块集。

第 23、24 行：初始化 Def、Use 位向量。

第 25~106 行：遍历基本块内 IR 列表。

第 29~34 行：获取处理方案。

第 35、36 行：如果操作符为 CALL 或 GETPROCADDR，则设置过程引用标志。这是一项非常重要的工作。编译器借助于过程引用标志可以轻松地识别不可到达的过程。所谓“不可到达过程”就是指那些永远无法被调用的过程。如果一个过程既没有被调用，也没有被取地址，那么，这个过程必定是不可到达过程。对于编译器而言，通常不需要为不可到达过程生成目标代码。当然，在多文件编译时，由于外部过程的原因，试图确定一个不可到达过程是相当复杂的。一般来说，不可到达过程删除必须是基于过程间分析进行讨论的。详细的原因，笔者先前已作相关说明。

第 37~62 行：主要处理过程调用 IR 对全局变量定值、引用所产生的副作用。实际上，程序的含义非常明确，即假设被调用过程将对所有的全局变量定值、引用都产生副作用。这种处理并不精确，却是最安全的策略。当然，读者需要关注几种特殊的情况：

(1) 调用外部过程是不影响全局变量的定值、引用的。这里的外部过程并不是指那些来自其他源文件的过程，而是指那些来自二进制库文件的过程。由于这种外部过程不可能给本地全局变量带来任何副作用，因此，这种过程调用不是设计者所关注的。第 37 行主要就是完成这一判定。

(2) 调用过程并不会给临时全局变量带来副作用。由于临时全局变量是由编译器自动生成的，用户程序是无法访问的，因此，过程的调用并不会影响临时全局变量。第 48 行主要就是完成这一判定。

第 54 行：调用 RecordUse 函数，登记引用点信息。稍后将详细分析 RecordUse 函数。

第 59 行：调用 RecordDef 函数，登记定值点信息。稍后将详细分析 RecordDef 函数。

第 65~88 行：处理内嵌汇编的定值点、引用点。

第 67~86 行：遍历内嵌汇编的参数信息，根据参数信息，登记定值点、引用点。当 flag 为 RW 时，表示参数所示变量在内嵌汇编中被定值且引用。当 flag 为 W 时，表示参数所示变量在内嵌汇编中被定值。当 flag 为 R 时，表示参数所示变量在内嵌汇编中被引用。

第 89~92 行：Op2 所示的变量被引用，登记变量引用点信息。

第 93~99 行：Op1 所示的变量被引用，登记变量引用点信息。有一种特殊的情况不能忽视，那就是变参传递。如果 IR 是变参传递指令，该 IR 也将是 Op1 所示变量的定值点。

第 96~98 行：主要用于判断当前 IR 是否为变参传递，为真则登记变量定值集。

第 100~104 行：Rslt 所示的变量被定值，登记变量定值点信息。当前 IR 为传参指令时跃过即可。

第 107、108 行：将 Def、Use 位向量复制到基本块的 Def、Use 属性中，以便后续优化算法使用。

第 110~113 行：将每个变量的定值集按从小到大的顺序排序，便于后续应用。

第 114~118 行：将每个变量的引用集按从小到大的顺序排序，便于后续应用。
最后，再来看看 RecordDef、RecordUse 函数的实现。

程序 7-3 DataFlowAnalysis.cpp

```

1 void CDataFlowAnalysis::RecordDef(map<int,int> VarMap,CBits &Use,CBits &Def,OpInfo Op,
2 vector<int> RefVar,int iPos,int iProcIndex)
3 {
4     if (!Op.m_bRef)
5     {
6         Def.set(VarMap[Op.m_iLink]);
7         VarDefUse(VarDef,Op.m_iLink,iPos,iProcIndex);
8     }
9     else
10    {
11        VarDefUse(VarAllUse,Op.m_iLink,iPos,iProcIndex);
12        if (!Def.value(VarMap[Op.m_iLink]))
13            Use.set(VarMap[Op.m_iLink]);
14        for(int p=0;p<RefVar.size();p++)
15            {
16                Def.set(VarMap[RefVar.at(p)]);
17                VarDefUse(VarDef,RefVar.at(p),iPos,iProcIndex);
18            }
19    }
20 }
```

第 4 行：判断 Op 是否为间接寻址操作数。

第 5~8 行：如果操作数不是间接寻址，则主要完成如下几项工作：

(1) 设置 Def 位向量。

(2) 调用 VarDefUse 设置变量定值集。实际上，VarDefUse 函数是根据引用的参数而定的，并不区分设置的是 VarAllUse 还是 VarDef。

第 10~19 行：如果操作数是间接寻址，则主要完成如下几项工作：

(1) 由于操作数是间接寻址，所以操作数本身只是引用，而间接寻址的目标变量才是可能被定值。因此，调用 VarDefUse 设置变量引用集。

(2) 设置 Use 位向量。注意，如果在一个基本块内，某一个变量引用之前存在对该变量的定值，那么，这样的引用并不是编译器所关心的。

(3) 根据 RefVar 集合，调用 VarDefUse 设置变量定值集。RefVar 集合中的成员都是被指针引用的变量。根据先前讨论的安全策略，这里假设间接寻址会对 RefVar 中所有的变量产生副作用。

程序 7-4 DataFlowAnalysis.cpp

```

1 void CDataFlowAnalysis::RecordUse(map<int,int> VarMap,CBits &Use,CBits &Def,OpInfo Op,
2 vector<int> RefVar,int iPos,int iProcIndex)
3 {
4     VarDefUse(VarAllUse,Op.m_iLink,iPos,iProcIndex);
```




```
5     if (!Def.value(VarMap[Op.m_iLink]))
6     {
7         Use.set(VarMap[Op.m_iLink]);
8     }
9     if (Op.m_bRef)
10    {
11        for(int p=0;p<RefVar.size();p++)
12        {
13            Use.set(VarMap[RefVar.at(p)]);
14            VarDefUse(VarAllUse,RefVar.at(p),iPos,iProcIndex);
15        }
16    }
17 }
```

第 4 行：调用 `VarDefUse` 设置变量引用集。

第 5~8 行：设置 `Use` 位向量。

第 9~17 行：如果 `Op` 是间接寻址操作数，则根据 `RefVar` 集合，调用 `VarDefUse` 设置变量定值集。`RefVar` 集合中的成员都是被指针引用的变量。根据先前讨论的安全策略，这里假设间接寻址会对 `RefVar` 中所有的变量产生副作用。

至此，笔者已经详细剖析了定值点、引用点分析的算法实现。定值点、引用点是后续数据流分析、优化算法的实现基础。囿于篇幅，笔者无法对该算法作更深入的阐述。另外，Neo Pascal 编译器也没有对过程间数据流作深入分析，因此无法得到更准确的结果。

7.4.4 活跃变量分析的实现

活跃变量分析是数据流分析的另一个重要问题。简而言之，活跃变量分析的目的就是求出各基本块出口之后的活跃变量集。前面，已经详细讲述了有关活跃变量分析的基本思想及相关理论。在本小节中，将关注活跃变量分析算法的设计与实现。

事实上，活跃变量分析算法就是利用数据流方程求解的过程。在例 7-1 中，应用迭代方式手工求解活跃变量的过程就是算法设计的基础。读者不妨再次参阅例 7-1 详细的求解过程，可能更有助于理解算法设计的思想。

先来谈谈活跃变量分析的相关数据结构。由于算法的数据结构较少且比较简单，故笔者并没有为其单独设置章节。在活跃变量分析中，主要涉及四个基于基本块的集合，即 `in`、`out`、`def`、`use`。为了节省空间且便于实现，通常使用位向量来描述这四个集合。在 Neo Pascal 中，它们是隶属于 `CBasicBlock` 类的，具体声明形式如下：

【声明 7-8】

```
struct CBasicBlock
{
    ...
    CBits *Def;           //定值位向量
    CBits *Use;          //引用位向量
    CBits *InSet;        //入口位向量
    CBits *OutSet;       //出口位向量
}
```

};

其中，Def、Use 位向量是由先前的 Def_Use_Analysis 函数分析得到的，因此，这里就不必关注了。而本小节的目标就是基于流图计算 InSet、OutSet 位向量。下面，就来看看详细的算法实现。

程序 7-5 DataFlowAnalysis.cpp

```

1 void CDataFlowAnalysis::In_Out_Analysis(int iProcIndex)
2 {
3     CBits InitBit(VarMap.size());
4     vector<CBasicBlock>* pBbs=&BasicBlock[iProcIndex];
5     bool bChange=true;
6     CBits NewInSet(VarMap.size());
7     for(int i=0;i<pBbs->size();i++)
8     {
9         if(!pBbs->at(i).InSet)
10            delete pBbs->at(i).InSet;
11        if(!pBbs->at(i).OutSet)
12            delete pBbs->at(i).OutSet;
13        pBbs->at(i).InSet=new CBits(InitBit);
14        pBbs->at(i).OutSet=new CBits(InitBit);
15    }
16    while (bChange)
17    {
18        bChange=false;
19        for (int i=0;i<pBbs->size();i++)
20        {
21            CBits Tmp(VarMap.size());
22            for (int j=0;j<pBbs->at(i).DownFlow.size();j++)
23            {
24                Tmp.Or(pBbs->at(pBbs->at(i).DownFlow[j]).InSet);
25            }
26            pBbs->at(i).OutSet->copy(&Tmp);
27            Tmp.Sub(pBbs->at(i).Def);
28            Tmp.Or(pBbs->at(i).Use);
29            NewInSet.copy(&Tmp);
30            if(!(NewInSet==*(pBbs->at(i).InSet)))
31            {
32                bChange=true;
33                pBbs->at(i).InSet->copy(&NewInSet);
34            }
35        }
36    }
37 }

```



第 4 行：获取当前过程的基本块信息。

第 7~15 行：初始化各基本块的 InSet、OutSet 位向量为空集。

第 16~36 行：bChange 为迭代结束的标志。在这个算法中，迭代结束的条件就是前后两次迭代的过程中，各基本块的 InSet 位向量都没有发生变化。实际上，这种情况表示迭代已经进入了稳定状态，而这个稳定状态就是所需要的最终结果，因此，就没有继续迭代的意义了。

第 18 行：进入一次求解过程之前，重置 bChange 标志。若求解过程中没有改变 bChange 标志，则表示当前状态已稳定，迭代即可终止。

第 19~35 行：遍历各基本块，依次计算各基本块的 InSet、OutSet。

第 22~26 行：计算当前基本块的 OutSet，其依据就是如下数据流公式：

$$\text{out}[s] = \bigcup \text{in}[i] \quad (\text{其中 } i \text{ 是 } s \text{ 的后继基本块})$$

第 27~29 行：计算当前基本块的 InSet，其依据就是如下数据流公式：

$$\text{in}[s] = (\text{out}[s] - \text{def}[s]) \cup \text{use}[s]$$

第 30~34 行：判断新求得的 InSet 与基本块的原 InSet 是否相等，若不相等，则使用新求得的 InSet 作为基本块的 InSet，并置 bChange 为 true，表示本次迭代过程中状态发生了变化，即未进入稳定状况，迭代不能终止。

虽然算法的规模很小，但是要深入理解其实现却并不容易。在分析代码的过程中，部分读者可能会觉得这个算法的逻辑比较晦涩。确实如此，在日常程序设计中，迭代的应用可能没有递归广泛，因此，对迭代算法的思想比较生疏也不足为奇。

7.4.5 ud 链、du 链分析的实现

在 Neo Pascal 中，许多优化算法都是基于 ud 链、du 链信息实现的。相比先前的数据流分析算法而言，ud 链、du 链分析可能稍复杂。

1. du 链分析算法

du 链分析算法的基本步骤是：

1) 在基本块内，分析当前 IR 之后是否存在其他 IR 对当前变量的定值。如果存在，则 du 链仅包含两条定值 IR 之间的对当前变量的所有引用点，并结束分析。如果不存在，则执行步骤 2。

2) 沿流图向下流的边，查找各条可能路径中最先出现的对当前变量定值的 IR 位置。如果查找成功，则将查找该 IR 过程中所经过的所有对当前变量的引用点都加入 du 链。如果查找失败，则将整个查找路径中的所有对当前变量引用的 IR 位置加入 du 链中。

du 链分析算法的关键就在于搜索流图的可能路径。实际上，搜索流图路径的算法是图深度遍历的一种特殊应用。对于不熟悉图深度遍历算法的读者，建议阅读数据结构相关章节。设计搜索流图路径算法时，有个细节值得注意，那就是环路的处理。

流图并不是 DAG 结构，在很多情况下，流图中可能出现有向环。路径搜索算法应该注意环状结构的处理，避免进入循环搜索。在图深度遍历算法中，通常是借助于一个顶点集合来避免结点的重复访问。即把已访问过的结点存放到一个顶点集合中，每次访问一个结点前，依据顶点集合判定该结点是否已访问。这里，值得注意的是，在 ud 链、du 链分析中，起始结点（即当前 IR 所在的结点）是需要特殊处理的。与其他结点不同，起始结点并不是

在第一次访问时，就将其加入顶点集合的。其中最主要的原因就是流图可能存在类似于图 7-7 所示的特殊环路，即某一个结点存在一条流向自身的边。以图 7-7 为例，讨论 P2 处关于 i 的 du 链。如果起始结点的处理与其他结点一样，那么，搜索算法首次访问起始结点时，首先将其加入顶点集合，再依据 du 链分析算法从当前 IR 所在位置（即 P2）开始向下搜索。显然，本次访问并不会涉及 P1 处的 IR。当 P2 之后不存在任何对 i 的定值点时，du 链分析算法就沿流图向下流的边搜索可能的路径。此时，搜索算法将第二次访问到起始结点，但是该结点已存在于顶点集合中，即表明该结点已被访问过了，因此就直接略过了。至此，该路径也就搜索完毕了。不过，整个过程中始终没有处理 P1 处的 IR。根据流图所示，显然 P2 处对 i 的定值是可以到达 P1 的。最简单的解决方法就是在第二次访问起始结点时，将其加入顶点集合。下面将详细分析 du 链分析算法的源代码实现。

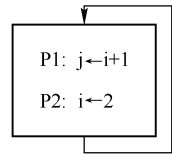


图 7-7 特殊的环路

程序 7-6 DataFlowAnalysis.cpp

```

1  vector<int> CDataFlowAnalysis::GetDu(int iVar,int iBb,int iStart,int iProcIndex)
2  {
3      vector<int> tmp;
4      int i,iEnd;
5      if (bHasVisit.find(iBb)!=bHasVisit.end())
6          return tmp;
7      else
8      {
9          if (bFirst)
10             bFirst=false;
11         else
12             bHasVisit.insert(iBb);
13     }
14     i=BackwardHasDef(iVar,iStart+1,CurrentBasicBlock->at(iBb).iEnd,iProcIndex);
15     iEnd=i== -1?CurrentBasicBlock->at(iBb).iEnd;i;
16     vector<int> TmpDu=VarAllUse[itos(iProcIndex)+"$" + itos(iVar)];
17     for(int j=0;j<TmpDu.size();j++)
18     {
19         if (iStart<TmpDu[j] && TmpDu[j]<=iEnd)
20             tmp.push_back(TmpDu[j]);
21     }
22     if (i!=-1)
23     {
24         return tmp;
25     }
26     else
27     {
28         for(int j=0;j<CurrentBasicBlock->at(iBb).DownFlow.size();j++)
29         {
30             int i=CurrentBasicBlock->at(iBb).DownFlow[j];

```



```
31         vector<int> DownFlowDu=GetDu(iVar,i,CurrentBasicBlock->at(i)
32                                     .iStart-1,iProcIndex);
33         if (!DownFlowDu.empty())
34             tmp.insert(tmp.end(),DownFlowDu.begin(),DownFlowDu.end());
35     }
36 }
37 return tmp;
38 }
```

第 5、6 行：判断当前基本块是否已访问。其中，bHasVisit 即为先前讨论的顶点集合，其声明形式如下：

【声明 7-9】

```
set<int> bHasVisit;
```

如果当前基本块的位序号已存在于 bHasVisit 中，即表示该基本块已被访问，故返回即可。

第 8~13 行：将当前基本块的位序号加入 bHasVisit 集合中。其中,bFirst 是一个布尔变量，用于满足起始基本块特殊处理的需要。

第 14 行：调用 BackwardHasDef 分析当前基本块内自 iStart 之后是否有其他 IR 对当前变量进行定值。如果存在，BackwardHasDef 将返回该定值点的位置，否则返回-1。关于 BackwardHasDef 函数的实现稍后详解。

第 15~21 行：如果 i!=-1，即当前基本块内自 iStart 之后有其他 IR 对当前变量进行定值，那么，将两条 IR 之间所有关于当前变量的引用都加入 tmp 向量中。否则，将当前基本块内自 iStart 之后所有关于当前变量的引用都加入 tmp 向量中。

第 22~25 行：如果 i!=-1，则无需沿流图向下边搜索可能的路径，返回 tmp 即可。

第 28~35 行：沿流图向下边搜索可能的路径，并将搜索过程中求得的引用点集合加入 tmp 向量中。值得注意的是第 31 行递归调用语句。有读者可能会问：为什么第 3 个实参需要在 CurrentBasicBlock->at(i).iStart 的基础上减 1？原因很简单，主要是为了适应当前 IR 处理的特殊需要。在起始基本块中，搜索引用点是从当前 IR 之后开始的，而不是从当前 IR 开始的。举个简单的例子，讨论 $a \leftarrow a+1$ 中 a 的 du 链时，在不考虑循环的情况下，可以断定对 a 的定值是不可能到达本句 IR 中对 a 的引用点的。因此，在第 14 行中，调用 BackwardHasDef 函数搜索当前引用点集合时，第 2 个参数是 iStart+1，而不是 iStart。但除了起始基本块外，其他基本块却无需考虑这些特殊处理。为了兼容 BackwardHasDef 函数，这里不得不预减 1。

下面来看 BackwardHasDef 的源代码实现。

程序 7-7 DataFlowAnalysis.cpp

```
1 int CDataFlowAnalysis::BackwardHasDef(int iVar,int iStart,int iEnd,int iProcIndex)
2 {
3     vector<int> tmp;
4     map<string,vector<int>>::iterator it=VarDef.find(itos(iProcIndex)+"$"+itos(iVar));
5     if (it==VarDef.end() || iStart>iEnd)
6         return -1;
```

```

7     else
8     {
9         int i;
10        for(i=0;i<it->second.size();i++)
11        {
12            if (it->second[i]>iEnd)
13                return -1;
14            if (it->second[i]>=iStart && it->second[i]<=iEnd)
15                break;
16        }
17        if (i!=it->second.size())
18            return it->second[i];
19        else
20            return -1;
21    }
22 }

```

第 4 行：从 VarDef 中获取关于给定变量的定值集。

第 5 行：如果给定变量的定值集为空或者给定检索范围为空，则返回-1，表示检索失败。

第 10~16 行：检索给定范围内的定值点。注意，定值集是按 IR 的位序号升序排列的，这是由定值、引用点分析算法处理的。因此，算法的实现就相对简单。

第 17~21 行：返回检索结果。如果检索成功，则返回定值点信息，否则返回-1。

2. ud 链分析算法

ud 链分析算法的基本步骤是：

(1) 在基本块内，分析当前 IR 之前是否存在其他 IR 对当前变量的定值。如果存在，则 ud 链仅包含该定值点，并结束分析。如果不存在，则执行步骤 2。

(2) 沿流图向上流的边查找各条可能路径中最后出现的对当前变量定值的 IR 位置。如果查找成功，则将该定值点加入 ud 链。如果查找失败，则不做任何处理。

这里，主要讨论 ud 链分析算法在编译器设计中的一个特殊应用，即判定变量使用前是否初始化的问题。在大多数语言中，关于变量使用前是否必须初始化的问题，并没有提出明确的规定。在实际编程中，程序员应该杜绝此类情形的发生。由于各种编译器的实现差异，故此类异常是随机的，这是非常可怕的。即便如此，编译器最多只能对这类现象给出语义警告而已，却不能强制禁止，除非语言有明确的约定。当然，并不要求编译器对此类问题一定要给出显式的警告或者其他提示。事实上，有些编译器对此并不关注。不过，数据流分析必须得到精准的分析结果，否则可能影响编译正确性。

严格地说，变量使用前未初始化的问题可以归入语义分析中讨论。但是，在语义分析阶段，编译器试图判定输入程序是否存在这类情形是非常困难的。因此，比较理想的解决方法就是在 ud 链分析算法中实现。那么，如何从 ud 链分析中获取相关的信息呢？假设从某个变量引用点出发开始分析该变量的 ud 链，在分析过程中，可以发现存在至少一条从流图的入口到当前引用点的路径，且该路径上没有任何对该变量的定值点。那么，也就表示该变量可能存在未初始化的情形。



ud 链分析算法的大致框架与 du 链类似。从算法实现来说，ud 链分析可能更简单些。下面，就来看看相关源代码的实现。

程序 7-8 DataFlowAnalysis.cpp

```
1  vector<int> CDataFlowAnalysis::GetUd(int iVar,int iBb,int iEnd,int iProcIndex,bool &bUninit)
2  {
3      vector<int> tmp;
4      int i;
5      if (bHasVisit.find(iBb)!=bHasVisit.end())
6          return tmp;
7      else
8      {
9          if (bFirst)
10             bFirst=false;
11         else
12             bHasVisit.insert(iBb);
13     }
14     i=ForwardHasDef(iVar,CurrentBasicBlock->at(iBb).iStart,iEnd-1,iProcIndex);
15     if (i==-1 && iBb==0)
16         bUninit=true;
17     if (i!=-1)
18     {
19         tmp.push_back(i);
20         return tmp;
21     }
22     else
23     {
24         for(int j=0;j<CurrentBasicBlock->at(iBb).UpFlow.size();j++)
25         {
26             int i=CurrentBasicBlock->at(iBb).UpFlow[j];
27             vector<int> UpFlowUd=GetUd(iVar,i,CurrentBasicBlock->at(i)
28                                     .iEnd+1,iProcIndex,bUninit);
29             if (!UpFlowUd.empty())
30                 tmp.insert(tmp.end(),UpFlowUd.begin(),UpFlowUd.end());
31         }
32     }
33     return tmp;
34 }
```

第 5、6 行：判断当前基本块是否已访问。这与 du 链分析算法是类似的。

第 8~13 行：将当前基本块的位序号加入 bHasVisit 集合中。其中，bFirst 是一个布尔变量，用于满足起始基本块特殊处理的需要。

第 14 行：调用 ForwardHasDef 分析当前基本块内自 iStart 之后是否有其他 IR 对当前变量进行定值。如果存在，则 ForwardHasDef 函数将返回该定值点的位置，否则返回-1。

第 15、16 行：如果已经遍历到流图的入口结点且仍没有找到定值点，则表示存在变量

使用前未初始化的情形。

第 17~20 行：如果 $i!=-1$ ，则无需继续沿流图向下边搜索，返回 `tmp` 即可。

第 23~32 行：沿流图向上边搜索可能的路径，将路径中最后一个定值点加入 `tmp` 向量中。由于遍历 IR 的过程是自下而上的，因此，路径中的最后一个定值点也就是遍历过程中的第一个定值点。如果检索成功，则不再继续遍历本条路径。在第 27 行中，递归调用 `GetUd` 函数时，第 3 个实参的情况与 `GetDu` 函数类似，这里就不再解释其原因了。

`ForwardHasDef` 的源代码实现与 `BackwardHasDef` 非常类似，这里，笔者仅给出源代码，详细的解释请读者参阅 `BackwardHasDef` 函数。

程序 7-9 DataFlowAnalysis.cpp

```

1  int CDataFlowAnalysis::ForwardHasDef(int iVar,int iStart,int iEnd,int iProcIndex)
2  {
3      vector<int> tmp;
4      map<string,vector<int>>::iterator it=VarDef.find(itos(iProcIndex)+"$"+itos(iVar));
5      if (it==VarDef.end() || iStart>iEnd)
6          return -1;
7      else
8          {
9              int i;
10             for(i=it->second.size()-1;i>=0;i--)
11                 {
12                     if (it->second[i]<iStart)
13                         return -1;
14                     if (it->second[i]>=iStart && it->second[i]<=iEnd)
15                         break;
16                 }
17             if (i!= -1)
18                 return it->second[i];
19             else
20                 return -1;
21         }
22     }

```

前面，已经详细讨论了 `ud` 链、`du` 链分析算法的实现细节。这两个算法是针对一个特定的程序点讨论某一变量在该程序点上的 `ud` 链、`du` 信息。最后，来看看 `ud` 链、`du` 链分析的主控函数，以便了解主控函数是如何调用 `GetUd`、`GetDu` 函数实现 `ud` 链、`du` 链分析的。

程序 7-10 DataFlowAnalysis.cpp

```

1  void CDataFlowAnalysis::ud_du_Analysis(int iProcIndex)
2  {
3      vector<CBasicBlock>* pBbs=&BasicBlock[iProcIndex];
4      IRCode* TmpIR;
5      for(int i=0;i<SymbolTbl.ProcInfoTbl[iProcIndex].m_Codes.size();i++)
6          {
7              SymbolTbl.ProcInfoTbl[iProcIndex].m_Codes[i].m_Op1.m_udChain.clear();

```




```
8      SymbolTbl.ProcInfoTbl[iProcIndex].m_Codes[i].m_Op1.m_duChain.clear();
9      SymbolTbl.ProcInfoTbl[iProcIndex].m_Codes[i].m_Op2.m_udChain.clear();
10     SymbolTbl.ProcInfoTbl[iProcIndex].m_Codes[i].m_Op2.m_duChain.clear();
11     SymbolTbl.ProcInfoTbl[iProcIndex].m_Codes[i].m_Rslt.m_udChain.clear();
12     SymbolTbl.ProcInfoTbl[iProcIndex].m_Codes[i].m_Rslt.m_duChain.clear();
13     SymbolTbl.ProcInfoTbl[iProcIndex].m_Codes[i].m_Op1.m_bUninit=false;
14     SymbolTbl.ProcInfoTbl[iProcIndex].m_Codes[i].m_Op2.m_bUninit=false;
15     SymbolTbl.ProcInfoTbl[iProcIndex].m_Codes[i].m_Rslt.m_bUninit=false;
16     }
17     for(int i=0;i<pBbs->size();i++)
18     {
19         CurrentBasicBlock=pBbs;
20         for(int j=pBbs->at(i).iStart;j<=pBbs->at(i).iEnd;j++)
21         {
22             TmpIR=&SymbolTbl.ProcInfoTbl[iProcIndex].m_Codes[j];
23             Opti_Tbl* TmpOpti;
24             if (TmpIR->m_eOpType<0)
25                 TmpOpti=SearchOptiTbl(OpType::TypeCast);
26             else if (TmpIR->m_eOpType>>4>=21)
27                 TmpOpti=SearchOptiTbl(TmpIR->m_eOpType);
28             else if (TmpIR->m_eOpType>>4<21)
29                 TmpOpti=SearchOptiTbl((OpType)(TmpIR->m_eOpType>>4<<4));
30             if (TmpOpti!=NULL && TmpOpti->eExpType!=Opti_Tbl::None)
31             {
32                 if (TmpIR->m_eOpType==OpType::ASM)
33                 {
34                     continue;
35                 }
36                 if (TmpIR->m_Op2.m_iType==OpInfo::VAR && !TmpIR->m_Op2.m_bRef)
37                 {
38                     bHasVisit.clear();
39                     bFirst=true;
40                     TmpIR->m_Op2.m_udChain=GetUd(TmpIR->m_Op2.m_iLink,i,j,iProcIndex
41                                             ,TmpIR->m_Op2.m_bUninit);
42                 }
43                 if (TmpIR->m_Op1.m_iType==OpInfo::VAR && !TmpIR->m_Op1.m_bRef)
44                 {
45                     bHasVisit.clear();
46                     bFirst=true;
47                     TmpIR->m_Op1.m_udChain=GetUd(TmpIR->m_Op1.m_iLink,i,j,iProcIndex
48                                             ,TmpIR->m_Op1.m_bUninit);
49                 }
50                 if (TmpIR->m_Rslt.m_iType==OpInfo::VAR && !TmpIR->m_Rslt.m_bRef)
51                 {
52                     bHasVisit.clear();
53                     bFirst=true;
```

```

54         TmpIR->m_Rslt.m_duChain=GetDu(TmpIR->m_Rslt.m_iLink,i,j,iProcIndex);
55     }
56 }
57 }
58 }
59 }

```

第3行：获取当前过程的基本块信息。

第5~16行：遍历当前过程的IR列表，清空各操作数的ud链、du链。

第17~58行：遍历流图中各基本块。

第20~57行：遍历基本块内的IR序列。

第24~29行：获取处理方案。

第36~42行：调用GetUd函数，分析m_Op2的ud链信息。注意，在ud链分析中，不分析间接寻址变量的ud链信息。由于Neo Pascal并没有打算做指针相关的优化，因此，分析间接寻址变量的ud链信息的意义不大。当然，目前的定值、引用点信息也不足以完成分析间接寻址变量的ud链。

第43~49行：调用GetUd函数，分析m_Op1的ud链信息。

第50~55行：调用GetDu函数，分析m_Rslt的du链信息。

至此，已经详细阐述了经典编译技术中关于数据流分析的一些基本思想与算法实现，并结合Neo Pascal剖析了几个最为常见的算法模型。数据流分析是优化技术中一个重要且复杂的话题。虽然笔者尽可能结合程序实例与Neo Pascal源代码解释一些经典的数据流问题，但是，对于没有学习过编译原理的读者来说，理解上述内容可能还是有一定难度的。针对尚未深入理解算法实现的读者，笔者的建议如下：暂且不必关注源代码实现，可以先把握两个要点。第一，理解数据流分析的一些名词概念；第二，理清Neo Pascal数据流分析算法的主要数据结构。这两个要点是学习后续章节的基础，相信读者阅读完本章内容之后，可能会对数据流分析有更深刻的认识和理解。

7.5 常量传播与常量折叠

7.5.1 常量传播基础

常量传播(constant propagation)是一种代码转换，其基本思想如下：对于给定关于某个变量 v 和一个常量 c 的赋值 $v \leftarrow c$ ，在没有出现其他关于 v 定值的程序范围内，编译器则用 c 来替代出现的 v 的引用。常量传播是一种实现代价较小且效果显著的转换，它为进一步的常量折叠创造了新的契机。下面，通过一个简单的实例来解释常量传播的方法及意义。

例7-2 局部常量传播的实例，见表7-4。

这是一个简单的常量传播的实例。试比较(b)、(c)两列，不难发现，IR的语句规模似乎并没有任何减小。请读者注意，常量传播的基本思想仅仅指出了其核心目标是尽可能用常量替代变量的引用，而不是减少IR指令。其根据基本思想的描述，从(b)到(c)的转换似乎已经

达到了预期的效果，例如，第 0 行对 `_T0` 的定值已经分别传播到了第 1、4 行 IR 中，而第 2 行对 `_T1` 的定值已经分别传播到了第 3、4 行 IR 中。那么，这种传播的意义又是什么呢？仔细观察(c)列，不难发现，第 4 行 IR 的两个操作数都变成了常量，这是常量折叠的契机。换句话说，在优化阶段讨论常量传播的主要目的之一就是为常量折叠创造优化的条件。

表 7-4 局部常量传播实例分析

(a) 输入源程序片段	(b) 原始的 IR	(c) 常量传播后的 IR
<code>var i,j:integer;</code>	0: (ByteToInt ,20,null,_T0)	0: (ByteToInt ,20,null,_T0)
<code>begin</code>	1: (ASSIGN_4 ,_T0,null,I)	1: (ASSIGN_4 ,20,null,I)
<code> i:=20;</code>	2: (ByteToInt ,30,null,_T1)	2: (ByteToInt ,30,null,_T1)
<code> j:=30;</code>	3: (ASSIGN_4 ,_T1,null,J)	3: (ASSIGN_4 ,30,null,J)
<code> j:=i+j;</code>	4: (ADD_4 ,I,J,_T2)	4: (ADD_4 ,20,30,_T2)
<code> result:=j;</code>	5: (ASSIGN_4 ,_T2,null,J)	5: (ASSIGN_4 ,_T2,null,J)
<code>end;</code>	6: (ASSIGN_4 ,J,null,RESULT)	6: (ASSIGN_4 ,J,null,RESULT)

下面，再来看一个稍复杂的实例。

例 7-3 全局常量传播的实例，见表 7-5。

表 7-5 全局常量传播实例分析

(a) 输入源程序	(b) 原始的 IR	(c) 常量传播后的 IR
<code>var i,j:integer;</code>	0: (ByteToInt ,30,null,_T0)	0: (ByteToInt ,30,null,_T0)
<code>begin</code>	1: (ASSIGN_4 ,_T0,null,J)	1: (ASSIGN_4 ,30,null,J)
<code> j:=30;</code>	2: (MR_4 ,I,J,_T1)	2: (MR_4 ,I,30,_T1)
<code> if i>j then</code>	3: (JNT ,_T1,null,_L2)	3: (JNT ,_T1,null,_L2)
<code> i:=20</code>	4: (ByteToInt ,20,null,_T4)	4: (ByteToInt ,20,null,_T4)
<code> else</code>	5: (ASSIGN_4 ,_T4,null,I)	5: (ASSIGN_4 ,20,null,I)
<code> i:=20;</code>	6: (JMP ,_L3,null,null)	6: (JMP ,_L3,null,null)
<code> j:=i+j;</code>	7: (LABEL ,_L2,null,null)	7: (LABEL ,_L2,null,null)
<code> result:=j;</code>	8: (ByteToInt ,20,null,_T5)	8: (ByteToInt ,20,null,_T5)
<code>end;</code>	9: (ASSIGN_4 ,_T5,null,I)	9: (ASSIGN_4 ,20,null,I)
	10: (LABEL ,_L3,null,null)	10: (LABEL ,_L3,null,null)
	11: (ADD_4 ,I,J,_T6)	11: (ADD_4 ,20,30,_T6)
	12: (ASSIGN_4 ,_T6,null,J)	12: (ASSIGN_4 ,_T6,null,J)
	13: (ASSIGN_4 ,J,null,RESULT)	13: (ASSIGN_4 ,J,null,RESULT)

从实现的角度而言，例 7-3 与例 7-2 的主要差别就在于例 7-3 的常量传播的范围已经突破了一个基本块的限制，是在整个过程范围内进行的。在这种情况下，需要考察从入口至当前 IR 的每一条可能到达路径的定值信息后，才能决策是否传播以及如何传播。在例 7-3 中，由于 if 语句真、假分支中都存在对 `i` 的常量定值，且常量的值都是 20，因此，该常量值是可以传播到 `j:=i+j` 中的。假设某一个分支中不存在对 `i` 的常量定值，或者两个常量的值不同，此时，进行常量传播是不允许的。本书将重点讨论关于过程内常量传播的实现，即全局常量传播。为了便于讲解，除了有明确说明之外，本书提及的“常量传播”都是指全局常量传播，而不是局部常量传播。

讨论常量传播的关键在于确定常量传播是否安全，这是至关重要的。从 ud 链的角度来思考这个问题就变得非常简单了。ud 链描述的就是可以到达某一变量引用点的所有关于该变量的定值信息。也就是说，讨论某一 IR 的操作数是否满足常量传播的条件，就只需要考察该操作数 ud 链中的每个定值点即可。如果定值点都满足以下四个条件：

- (1) IR 的操作码是赋值或类型转换运算。
- (2) Op1 皆为值相同的常量。
- (3) 结果操作数不是间接寻址方式。
- (4) 不存在未初始化的路径。

那么，常量传播就是安全的。对于前 3 个条件，一般读者不会有什么异议。而第 4 个条件通常是容易忽视的。在这个问题上，即使是经典的 GCC，也存在百密一疏之处。

除了常量传播的安全性之外，判定两个常量值是否相等的标准也是编译器设计者感兴趣的。例如，10 与 10.0 是否相等，10.0000001 与 10.000000 是否相等，这些问题都必须由编译器设计者来明确回答。在实践中，不同的编译器关于这类问题的解释与处理是不尽相同的。很多时候，这个标准是依赖于运行机或者目标机的系统结构而定的。譬如，浮点数的表示、浮点指令集等。下面，笔者通过实例来分析 GCC 中关于常量传播的一个疏漏之处。

例 7-4 GCC 常量传播的实例分析。

如果读者使用 GCC 3.2 的-O2 参数编译表 7-6 中(a)列源程序后，不难发现，生成的可执行文件的运行结果永远是“23”，并不依赖于 aa()函数的返回值。实际上，从(c)列的汇编程序片段中，读者是很容易证明这一结论的。请读者注意(c)列第 13 行与(b)列第 17 行的区别，这是调用 printf 之前的传参指令，不难发现，(c)列程序中是直接将常量“23”压栈，而(b)列程序中是将“-8(%ebp)”所示存储单元中的数据压栈。显然，从程序的原始语义来说，(c)列程序是不正确的。也就是说，在这种情形下，GCC 的常量传播算法是不安全的。

表 7-6 GCC 常量传播的实例分析

(a) 输入源程序	(b) 未优化的汇编程序片段	(c) 带优化的汇编程序片段
<code>#include "stdio.h"</code>	1 <code>_main:</code>	<code>_main:</code>
<code>int aa()</code>	2 <code> pushl %ebp</code>	<code> pushl %ebp</code>
<code>{</code>	3 <code> movl %esp,%ebp</code>	<code> xorl %eax,%eax</code>
<code> int a;</code>	4 <code> subl \$24,%esp</code>	<code> movl %esp,%ebp</code>
<code> scanf("%d",&a);</code>	5 <code> andl \$-16,%esp</code>	<code> pushl %edx</code>
<code> return a;</code>	6 <code> movl \$0,%eax</code>	<code> pushl %edx</code>
<code>}</code>	7 <code> movl %eax,-12(%ebp)</code>	<code> andl \$-16,%esp</code>
<code>main()</code>	8 <code> movl -12(%ebp),%eax</code>	<code> call __alloca</code>
<code>{</code>	9 <code> call __alloca</code>	<code> call __main</code>
<code> int a,b;</code>	10 <code> call __main</code>	<code> call _aa</code>
<code> if(aa())</code>	11 <code> call _aa</code>	<code> pushl %eax</code>
<code> b=23;</code>	12 <code> testl %eax,%eax</code>	<code> pushl %eax</code>
<code> printf("%d",b);</code>	13 <code> je L5</code>	<code> pushl \$23</code>
<code>}</code>	14 <code> movl \$23,-8(%ebp)</code>	<code> pushl \$LC0</code>
	15 <code>L5:</code>	<code> call _printf</code>
	16 <code> subl \$8,%esp</code>	<code> movl %ebp,%esp</code>
	17 <code> pushl -8(%ebp)</code>	<code> popl %ebp</code>
	18 <code> pushl \$LC0</code>	<code> ret</code>
	19 <code> call _printf</code>	
	20 <code> addl \$16,%esp</code>	
	21 <code> leave</code>	
	22 <code> ret</code>	



问题的本质在于 GCC 的常量传播并没有考虑未初始化路径的因素。从 ud 链角度分析，很容易理解这个错误。在整个程序中，b 的唯一值点就是 b=23。因此，不考虑未初始化路径的情况下，编译器必然认为这种情形是满足常量传播要求，以至于将常量“23”直接传播到了 printf 语句中参数 b 的引用处。

至此，已经讨论了常量传播的基本话题。常量传播的目的就是为常量折叠创造优化的条件。因此，在很多情况下，为了保证优化效果，常量折叠是紧接着常量传播进行的。

7.5.2 常量传播的实现

在 Neo Pascal 中，常量传播是基于过程内讨论的。下面将详细分析 Neo Pascal 中常量传播算法的源代码实现。

程序 7-11 Const_Prop.cpp

```
1 void CConst_Prop::Const_Prop(int iProcIndex)
2 {
3     bool bChange=true;
4     while (bChange)
5     {
6         bChange=false;
7         for (int i=0;i<SymbolTbl.ProcInfoTbl.at(iProcIndex).m_Codes.size();i++)
8         {
9             IRCode* TmpIR=&SymbolTbl.ProcInfoTbl.at(iProcIndex).m_Codes[i];
10            Opti_Tbl* TmpOpti;
11            if (TmpIR->m_eOpType<0)
12                TmpOpti=SearchOptiTbl(OpType::TypeCast);
13            else if (TmpIR->m_eOpType>>4>=21)
14                TmpOpti=SearchOptiTbl(TmpIR->m_eOpType);
15            else if (TmpIR->m_eOpType>>4<<21)
16                TmpOpti=SearchOptiTbl((OpType)(TmpIR->m_eOpType>>4<<4));
17            if (TmpOpti!=NULL && TmpOpti->eExpType!=Opti_Tbl::None)
18            {
19                if (TmpIR->m_Op2.m_iType==OpInfo::VAR && !TmpIR->m_Op2.m_bRef
20                    && !TmpIR->m_Op2.m_bUninit)
21                {
22                    bChange|=GenConstOp(TmpIR->m_Op2,iProcIndex);
23                }
24                if (TmpIR->m_Op1.m_iType==OpInfo::VAR && !TmpIR->m_Op1.m_bRef
25                    && !TmpIR->m_Op1.m_bUninit)
26                {
27                    bChange|=GenConstOp(TmpIR->m_Op1,iProcIndex);
28                }
29            }
30        }
31    }
32 }
```

第3行：初始化常量传播的结束标志。与许多优化算法一样，常量传播通常也会进行多遍处理，直到进入稳定状态为止。所谓“稳定状态”就是指在一遍扫描过程中，代码没有发生任何改进的状态。实际上，多遍扫描处理的目的是为了使得常量传播的范围更广，效果更显著。

第4行：判断常量传播是否结束。

第7~31行：遍历 IR 列表。

第9行：获取当前 IR。

第11~16行：获取处理方案。

第19、20行：判断 `m_Op2` 操作数是否需要常量传播优化。通常，要满足如下三个条件：

- (1) 操作数必须为变量，即 `m_iType==OpInfo::VAR`。
- (2) 操作数不允许是间接寻址，即 `m_bRef` 为 `false`。
- (3) 不允许存在未初始化的路径，即 `m_bUninit` 为 `false`。

第22行：调用 `GenConstOp` 函数进行常量传播。`GenConstOp` 函数声明形式如下：

【声明 7-10】

```
bool CConst_Prop::GenConstOp(OpInfo &Op,int iProcIndex)
```

注意，参数 `Op` 为引用传递，也就是说，如果给定操作数的所有 `ud` 点都满足常量传播的要求，那么，`GenConstOp` 将通过修改 `Op` 的属性，以实现常量传播的目的。函数的返回值用于标识实参操作数在常量传播过程中是否被修改。关于 `GenConstOp` 函数的实现稍后详解。

第24~28行：判断 `m_Op1` 操作数是否需要常量传播优化。如满足以上三个条件，即调用 `GenConstOp` 函数进行常量传播。

下面来看看 `GenConstOp` 函数的实现，这是常量传播的核心。

程序 7-12 Const_Prop.cpp

```
1  bool CConst_Prop::GenConstOp(OpInfo &Op,int iProcIndex)
2  {
3      bool first=true;
4      int ConstOpType=-1;
5      OpInfo* TmpOp;
6      ConstInfo TmpConstInfo;
7      for(int i=0;i<Op.m_udChain.size();i++)
8      {
9          IRCode* TmpIR=&SymbolTbl.ProcInfoTbl.at(iProcIndex).m_Codes[Op.m_udChain[i]];
10         Opti_Tbl* TmpOpti;
11         if (TmpIR->m_eOpType<0)
12             TmpOpti=SearchOptiTbl(TmpIR->m_eOpType);
13         else if (TmpIR->m_eOpType>>4==21)
14             TmpOpti=SearchOptiTbl(TmpIR->m_eOpType);
15         else if (TmpIR->m_eOpType>>4<21)
16             TmpOpti=SearchOptiTbl((OpType)(TmpIR->m_eOpType>>4<<4));
17         if (TmpOpti!=NULL && TmpOpti->eConstPropType!=Opti_Tbl::None)
```



```
18     {
19         if (TmpIR->m_Op1.m_iType==OpInfo::CONST)
20         {
21             if (TmpIR->m_Rslt.m_bRef)
22                 return false;
23             if (first)
24             {
25                 TmpConstInfo=SymbolTbl.ConstInfoTbl.at(TmpIR->m_Op1.m_iLink);
26                 TmpOp=&TmpIR->m_Op1;
27                 ConstOpType=TmpOpti->eConstPropType;
28                 first=false;
29             }
30             else
31             {
32                 if (TmpConstInfo.m_ConstType!=SymbolTbl.ConstInfoTbl
33                     .at(TmpIR->m_Op1.m_iLink).m_ConstType)
34                 {
35                     return false;
36                 }
37                 switch (TmpConstInfo.m_ConstType)
38                 {
39                     case ConstType::BOOLEAN:
40                     case ConstType::PTR:
41                     case ConstType::SET:
42                     case ConstType::STRING:
43                         if (TmpConstInfo.m_szName!=SymbolTbl.ConstInfoTbl
44                             .at(TmpIR->m_Op1.m_iLink).m_szName)
45                             return false;
46                         break;
47                     default:
48                         if (fabs(TmpConstInfo.m_fVal-SymbolTbl.ConstInfoTbl
49                             .at(TmpIR->m_Op1.m_iLink).m_fVal)>10e-30)
50                             return false;
51                         break;
52                 }
53             }
54         }
55         else
56             return false;
57     }
58     else
59         return false;
60 }
61 if (ConstOpType==-1)
62     return false;
63 Op.m_iType=OpInfo::CONST;
```

```
64     if (ConstOpType==Opti_Tbl::ToAssign)
65     {
66         Op.m_iLink=TmpOp->m_iLink;
67     }
68     else
69     {
70         Op.m_iLink=SymbolTbl.ConstInfoTbl.size();
71     }
72     Op.m_udChain.clear();
73     switch(ConstOpType)
74     {
75     case Opti_Tbl::ToInt:
76         TmpConstInfo.m_ConstType=ConstType::INTEGER;
77         TmpConstInfo.m_StoreType=StoreType::T_INTEGER;
78         break;
79     case Opti_Tbl::ToReal:
80         TmpConstInfo.m_ConstType=ConstType::REAL;
81         TmpConstInfo.m_StoreType=StoreType::T_REAL;
82         break;
83     case Opti_Tbl::ToSingle:
84         TmpConstInfo.m_ConstType=ConstType::REAL;
85         TmpConstInfo.m_StoreType=StoreType::T_SINGLE;
86         break;
87     case Opti_Tbl::ToSmall:
88         TmpConstInfo.m_ConstType=ConstType::INTEGER;
89         TmpConstInfo.m_StoreType=StoreType::T_SMALLINT;
90         break;
91     case Opti_Tbl::ToWord:
92         TmpConstInfo.m_ConstType=ConstType::INTEGER;
93         TmpConstInfo.m_StoreType=StoreType::T_WORD;
94         break;
95     case Opti_Tbl::ToLong:
96         TmpConstInfo.m_ConstType=ConstType::INTEGER;
97         TmpConstInfo.m_StoreType=StoreType::T_LONGWORD;
98         break;
99     case Opti_Tbl::ToShort:
100        TmpConstInfo.m_ConstType=ConstType::INTEGER;
101        TmpConstInfo.m_StoreType=StoreType::T_SHORTINT;
102        break;
103     case Opti_Tbl::ToByte:
104        TmpConstInfo.m_ConstType=ConstType::INTEGER;
105        TmpConstInfo.m_StoreType=StoreType::T_BYTE;
106        break;
107     case Opti_Tbl::ToLong8:
108        TmpConstInfo.m_ConstType=ConstType::INTEGER;
109        TmpConstInfo.m_StoreType=StoreType::T_LONG8;
```




```
110         break;
111     }
112     SymbolTbl.ConstInfoTbl.push_back(TmpConstInfo);
113     bOptiChanged=true;
114     return true;
115 }
```

第 7 行：遍历操作数的 ud 链信息。

第 9 行：根据 ud 链中的定值点信息，获取相应的 IR。

第 11~16 行：获取处理方案。

第 19~54 行：逐一判断 ud 链各定值点是否满足常量传播的基本要求，主要判断如下三个条件：

(1) 第 19 行：判断定值点 IR 的 m_Op1 是否为常量。

(2) 第 21 行：判断定值点 IR 的 m_Rslt 是否为间接寻址方式。

(3) 第 23~53 行：判断各定值点的常量值是否相同。这里，值得注意的是，整型、实型常量比较 m_fVal 属性，而其他类型常量直接比较 m_szName 属性。除了比较常量值之外，编译器还严格比较常量的类型信息，即 m_ConstType 属性。

在逐一判断比较的过程中，如出现不满足常量传播要求的情形，则立刻返回 false。也就是说，这个 for 循环能够正常结束即表明给定操作数 ud 链的各定值点完全满足常量传播的要求。

第 61、62 行：这个 if 结构主要用于处理操作数的 ud 链为空的情况。这种情况是不需要进行常量传播的。

第 63~112 行：通过改变参数 Op 的属性，完成常量传播。

第 63 行：设置 m_iType 属性为 OpInfo::CONST。

第 64~71 行：设置 m_iLink 属性。根据常量传播的特点，操作数定值点的 IR 的操作符可能有两种情况，即赋值运算或类型转换运算。如果 IR 是赋值运算，那么，只需将 m_iLink 直接指向该常量符号即可。如果 IR 是类型转换运算，就必须由编译器完成常量类型的转换，即生成一个新的常量符号用于存储转换结果。并且将 m_iLink 属性指向新的常量符号，以保证常量传播后 IR 操作数的类型仍然是兼容的。

第 73~112 行：根据定值点的 IR 操作符，设置新生成常量符号的类型信息。

至此，已经详细讨论了 Neo Pascal 中常量传播的实现细节。然而，常量传播的一些理论问题并未作深入阐述。有兴趣的读者可以参考“龙书”的相关章节。

7.6 复写传播

7.6.1 复写传播的基础

复写传播 (copy propagation) 是一种代码转换，其基本思想如下：对于给定的关于某个变量 v 和 s 的赋值 $v \leftarrow s$ ，在没有出现其他关于 v 定值的程序范围内，编译器用 s 来替代出现的 v 的引用。从定义上来说，不难发现复写传播与常量传播有着惊人的相似之处。的

确如此，这正是传播优化的基本思想。下面，通过一个简单的实例来解释复写传播的方法及意义。

例 7-5 局部复写传播的实例。见表 7-7。

表 7-7 局部复写传播的实例分析

(a) 输入源程序片段	(b) 原始的 IR	(c) 复写传播后的 IR
var	0: (ASSIGN_4 ,A,null,B)	0: (ASSIGN_4 ,A,null,B)
a,b,c,d:integer;	1: (ASSIGN_4 ,B,null,C)	1: (ASSIGN_4 ,A,null,C)
e,f,g:integer;	2: (ASSIGN_4 ,C,null,D)	2: (ASSIGN_4 ,A,null,D)
begin	3: (ASSIGN_4 ,D,null,E)	3: (ASSIGN_4 ,A,null,E)
b:=a;	4: (ASSIGN_4 ,E,null,F)	4: (ASSIGN_4 ,A,null,F)
c:=b;	5: (ASSIGN_4 ,F,null,G)	5: (ASSIGN_4 ,A,null,G)
d:=c;	6: (ASSIGN_4 ,G,null,RESULT)	6: (ASSIGN_4 ,A,null,RESULT)
e:=d;		
f:=e;		
g:=f;		
result:=g;		
end;		

从形式上来说，这个实例可能有些极端，但非常有利于说明复写传播的方法及意义。比较(b)、(c)两列，不难发现，(c)列中所有指令的第 1 个操作数都被替代为“A”。根据复写传播的定义，由于(b)列中第 0 行是 $B \leftarrow A$ 的赋值，而整个程序中并不存在其他关于 B 的定值，编译器则可以用 A 来替代所有出现的对 B 的引用。因此，将第 1 行替换为 $C \leftarrow A$ 的赋值。以此类推，不难得到(c)列的结果。关于复写传播的方法，应该并不难理解。

复写传播的现实意义是什么呢？或者说，相比(b)列代码而言，(c)列代码的优势是什么呢？实际上，复写传播的意义就是在于尽可能减少程序中的活跃变量的个数。从一个函数的角度来说，只有 RESULT 是活跃变量，因为它是用于存储函数返回值的，其他的变量在离开函数体后，都是没有意义的。当然，除了 RESULT 之外，与 RESULT 定值相关的变量也将是活跃的。先来分析(b)列代码的活跃变量的个数。由于 RESULT 值是从 G 中得到的，故 G 也是活跃的。而 G 的值又是从 F 中获得的，因此 F 也将是活跃的。以此类推，(b)列代码中 A、B、C、D、E、F、G 都将是活跃变量。以同样的方法分析(c)列代码，不难发现，其中只有 A 是活跃变量。根据先前关于活跃变量的讨论，对于非活跃变量的赋值操作都是没有意义的，即可删除。这样的话，(c)列代码最终就只剩下最后一行 IR 了，这个成果是令人可喜的。不仅如此，就本例而言，当删除了(c)列中冗余的赋值之后，由于整个程序中不存在任何关于 B、C、D、E、F、G 的引用与定值，因此，编译器不必再为这些变量分配存储空间。

同样，复写传播也可以分为局部遍与全局遍，即局部复写传播与全局复写传播。前者是基于基本块内实现的，而后者是基于整个过程讨论的。这里，就不再详细分析全局复写传播的实例了。复写传播虽然成效显著，但并不是万能的。在一些特殊的情况下，复写传播是可能失效的。正如先前讨论的，优化算法都是针对某类特殊问题的一种解决方案，不可能要求一个优化能做到以不变应万变。



在实践中，有些初学者可能试图使用复写传播优化如图 7-8 所示的结构。实际上，这是完全徒劳的。这种代码形态已经不是复写传播讨论的范畴了。读者仔细推敲复写传播的定义，不难发现，定义中非常明确地指出了复写传播的前提只是 $v \leftarrow s$ 的赋值，而其中并不能含有其他运算。当然，在图 7-8 中，形如基本块 B2、B3 中的 $i \leftarrow j+1$ 也是不能接受的冗余。在现代编译技术中，通常使用一种称为尾融合的算法来实现两个复写赋值的合并。尾融合的基本思想就是将 B2、B3 这两个基本块尾部相同的代码提取出来，移至另一个新生成的基本块 B5 中。删除 B2、B3 与 B4 的后继关系。同时，令 B5 成为 B2、B3 的唯一后继基本块。而 B5 的后继基本块即为 B2、B3 的原后继 B4。

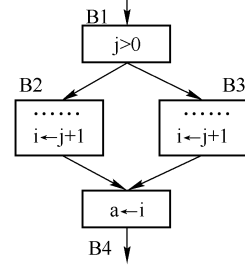


图 7-8 尾融合的实例

7.6.2 复写传播的实现

复写传播对于 IR 的级别并没有太高的要求，无论是 HIR 还是 LIR，复写传播通常都能适应。当然，复写传播的实现方式也并不唯一。Neo Pascal 使用了一种易于理解且便于实现的方式，虽然它的效率可能不是最优的。就算法实现而言，Neo Pascal 的复写传播与常量传播是极其类似的，也是基于 ud 链完成的。下面，就来详细分析相关实现。

程序 7-13 Copy_Prop.cpp

```
1 void CCopy_Prop::Copy_Prop(int iProcIndex)
2 {
3     bool bChange=true;
4     while (bChange)
5     {
6         bChange=false;
7         for (int i=0;i<SymbolTbl.ProcInfoTbl.at(iProcIndex).m_Codes.size();i++)
8         {
9             IRCode* TmpIR=&SymbolTbl.ProcInfoTbl.at(iProcIndex).m_Codes[i];
10            Opti_Tbl* TmpOpti;
11            if (TmpIR->m_eOpType==OpType::ASM)
12                continue;
13            if (TmpIR->m_eOpType<0)
14                TmpOpti=SearchOptiTbl(OpType::TypeCast);
15            else if (TmpIR->m_eOpType>>4==21)
16                TmpOpti=SearchOptiTbl(TmpIR->m_eOpType);
17            else if (TmpIR->m_eOpType>>4<21)
18                TmpOpti=SearchOptiTbl((OpType)(TmpIR->m_eOpType>>4<<4));
19            if (TmpOpti!=NULL && TmpOpti->eExpType!=Opti_Tbl::None)
20            {
21                if (TmpIR->m_Op2.m_iType==OpInfo::VAR && !TmpIR->m_Op2.m_bRef)
22                {
23                    bChange|=GenCopyOp(TmpIR->m_Op2,iProcIndex);
24                }
25            }
26        }
27    }
28 }
```

```

25         if (TmpIR->m_Op1.m_iType==OpInfo::VAR && !TmpIR->m_Op1.m_bRef)
26         {
27             bChange|=GenCopyOp(TmpIR->m_Op1,iProcIndex);
28         }
29     }
30 }
31 }
32 }

```

第 3 行：初始化复写传播的结束标志。与许多优化算法一样，复写传播通常会进行多遍扫描处理，直到进入稳定状态为止。

第 4 行：判断复写传播是否结束。

第 7~30 行：遍历 IR 列表。

第 9 行：获取当前 IR。

第 11~12 行：略过内嵌汇编代码。

第 13~18 行：获取处理方案。

第 21 行：判断 `m_Op2` 是否需要进行复写传播优化。通常，要满足如下两个条件：

(1) 操作数必须为变量，即 `m_iType==OpInfo::VAR`。

(2) 操作数不允许是间接寻址，即 `m_bRef` 为 `false`。

第 23 行：调用 `GenCopyOp` 函数进行复写传播。`GenCopyOp` 函数声明形式如下：

【声明 7-11】

```
bool CCopy_Prop::GenCopyOp(OpInfo &Op,int iProcIndex)
```

注意，参数 `Op` 为引用传递，也就是说，如果给定操作数的所有 `ud` 点都满足复写传播的要求，那么，`GenCopyOp` 将通过修改 `Op` 的属性，以实现复写传播的目的。函数的返回值用于标识实参操作数在复写传播过程中是否被修改。关于 `GenCopyOp` 函数的实现，稍后详解。

第 25~28 行：判断 `m_Op1` 是否需要进行复写传播优化。如满足以上两个条件，即调用 `GenCopyOp` 函数进行复写传播。

下面来看看 `GenCopyOp` 函数的实现，这是复写传播的核心。

程序 7-14 Copy_Prop.cpp

```

1  bool CCopy_Prop::GenCopyOp(OpInfo &Op,int iProcIndex)
2  {
3      bool first=true;
4      int iVal=-1;
5      for(int i=0;i<Op.m_udChain.size();i++)
6      {
7          IRCode* TmpIR=&SymbolTbl.ProcInfoTbl.at(iProcIndex)
8              .m_Codes[Op.m_udChain[i]];
9          Opti_Tbl* TmpOpti;
10         if (TmpIR->m_eOpType<0)
11             TmpOpti=SearchOptiTbl(OpType::TypeCast);

```



```
12         else if (TmpIR->m_eOpType>>4>=21)
13             TmpOpti=SearchOptiTbl(TmpIR->m_eOpType);
14         else if (TmpIR->m_eOpType>>4<21)
15             TmpOpti=SearchOptiTbl((OpType)(TmpIR->m_eOpType>>4<<4));
16         if (TmpOpti!=NULL && TmpOpti->eConstPropType!=Opti_Tbl::None)
17             {
18                 if (TmpIR->m_Op1.m_iType==OpInfo::VAR)
19                     {
20                         if (TmpIR->m_Op1.m_bRef || TmpIR->m_Rslt.m_bRef)
21                             return false;
22                         if (first)
23                             {
24                                 iVal=TmpIR->m_Op1.m_iLink;
25                                 first=false;
26                             }
27                         else
28                             if (iVal!=TmpIR->m_Op1.m_iLink)
29                                 {
30                                     return false;
31                                 }
32                             }
33                         else
34                             return false;
35                     }
36                 else
37                     return false;
38             }
39         if ( iVal==1)
40             return false;
41         Op.m_udChain.clear();
42         Op.m_iLink=iVal;
43         bOptiChanged=true;
44         return true;
45     }
46 }
```

第 5 行：遍历操作数的 ud 链信息。

第 7 行：根据 ud 链中的定值点信息，获取相应的 IR。

第 10~15 行：获取处理方案。

第 16~37 行：逐一判断 ud 链各定值点是否满足复写传播的基本要求，主要判断如下三个条件：

(1) 第 18 行：判断定值点 IR 的 m_Op1 是否为变量。

(2) 第 20 行：判断定值点 IR 的 m_Op1、m_Rslt 是否为间接寻址。

(3) 第 22~31 行：判断各定值点的 m_Op1 所示的变量是否相同，只需比较 m_iLink 属性即可。

在逐一判断比较的过程中，如出现不满足复写传播要求的情形，则立刻返回 `false`。也就是说，这个 `for` 循环能够正常结束即表明给定操作数 `ud` 链的各定值点完全满足复写传播的要求。

第 39~40 行：这个 `if` 结构主要用于处理操作数的 `ud` 链为空的情况。这种情况是不需要进行复写传播的。

第 41 行：清空 `Op` 的 `ud` 链。

第 42 行：设置 `m_iLink` 属性。

7.7 代数简化

7.7.1 代数简化基础

代数简化 (algebraic simplification) 也称“代数化简”，其基本思想就是利用运算符或运算符及操作数的特殊组合的代数性质来简化表达式的形式。初次接触这个定义的读者可能会联想起深奥的数学，难道代数简化背后蕴含着复杂的数学理论吗？事实恰恰相反。虽然笔者尚不敢确定代数简化是不是最简单的优化算法，但至少也是最简单的优化算法之一。

先来看一些简单等式：

$$i+0=i、i-0=i、i*1=i、i/1=i、b \text{ or } \text{true}=\text{true}、b \text{ and } \text{false}=\text{false}、b \text{ shr } 0=b$$

.....

从数学上来说，这些式子都是成立的。而代数简化就是利用这些简单等式的性质，达到简化代码的效果。例如，可以将 `y←x or true` 转换为 `y←true`。同样，也可以将 `p←q+0` 转换为 `p←q`。当然，在数学中，类似的等式是常见的，因此，其规模可能非常庞大，试图逐一枚举恐怕并非易事。这里，笔者并不打算列举各种代数简化等式，当然，这样做的本身意义也并不大。代数简化的意义就是简化表达式的运算，这应该不难理解。

有些代数简化也可以看成是强度削弱，即用一种计算速度较快的运算来替代另一种相对较慢的运算。例如，可以将 `i*8` 转换为 `i shl 3`。这种转换的主要原因就是移位运算的速度要远远超过乘、除运算。在一些 RISC 结构的目标机中，强度削弱的应用是非常广泛的。由于目标机指令系统可能不支持某些复杂的运算指令，此时，强度削弱的替换思想就是不错的应对之策。最常见的是用一串移位和加、减运算来替代乘、除运算。例如，`t←i*5` 可以用

$$t \leftarrow i \text{ shl } 2、t \leftarrow t + i$$

代替。而 `t←i*7` 可以用

$$t \leftarrow i \text{ shl } 3、t \leftarrow t - i$$

来代替。当然，强度削弱并不局限于优化阶段。实践证明，有些强度削弱的思想应用在代码生成阶段的效果可能会更佳。下面，来看一个代数简化的实例。

例 7-6 代数简化的实例。见表 7-8。

代数简化通常只针对一条 IR 进行操作，并不会影响上下文。这是代数简化的一个重要特点。在这个实例中，代数简化算法主要作用于三条 IR 上，即第 2、5、9 行 IR。下面对每一次化简稍作解释。



表 7-8 代数简化的实例分析

(a) 输入源程序	(b) 原始的 IR	(c) 代数简化后的 IR
var a,b:integer;	0: (ByteToInt ,1,null,_T1)	0: (ByteToInt ,1,null,_T1)
begin	1: (MR_4 ,A,_T1,_T0)	1: (MR_4 ,A,1,_T0)
if (a>1) and false then	2: (AND_0 ,_T0,FALSE,_T2)	2: (ASSIGN_1 ,FALSE,null,_T2)
a:=1*b;	3: (JNT ,_T2,null,_L3)	3: (JNT ,_T2,null,_L3)
a:=b*4;	4: (ByteToInt ,1,null,_T6)	4: (ByteToInt ,1,null,_T6)
result:=a;	5: (MUL_4 ,_T6,B,_T5)	5: (ASSIGN_4 ,B,null,_T5)
end;	6: (ASSIGN_4 ,_T5,null,A)	6: (ASSIGN_4 ,_T5,null,A)
	7: (LABEL ,_L3,null,null)	7: (LABEL ,_L3,null,null)
	8: (ByteToInt ,4,null,_T8)	8: (ByteToInt ,4,null,_T8)
	9: (MUL_4 ,B,_T8,_T7)	9: (SHL_4 ,B,2,_T7)
	10: (ASSIGN_4 ,_T7,null,A)	10: (ASSIGN_4 ,_T7,null,A)
	11: (ASSIGN_4 ,A,null,RESULT)	11: (ASSIGN_4 ,_T7,null,RESULT)

第 2 行的化简依据就是等式 $i \text{ and } \text{false} = \text{false}$ ，这显然是成立的。

第 5 行的化简依据就是等式 $i * 1 = i$ 。当然，这是基于常量传播后才能化简的。

第 9 行的化简依据就是等式 $i * 4 = i \text{ shl } 2$ 。同样，这也是基于常量传播后才能化简的。

仅从这个实例分析，似乎并没有看到代数简化的明显优势。除了强度削弱之外，更重要的是，这一遍代数简化为常量传播创造了新的机会。例如，将第 2、5 行都替换成了赋值形式，这种形式最有利于进行常量传播。在实践中，这种渐进式的改善是非常有效的，许多看似棘手的问题可能在无意间就被化解了。在本例中，最终可以借助于这种渐进式优化的思想将整个 if 语句删除，这个效果应该是令人满意的。

代数简化一般都是针对某一条 IR 进行的，所以不必过多考虑基本块或其他流图的因素。当然，代数简化也不太关注 IR 的级别，一般而言，HIR 或 LIR 都是可以接受的。不过，笔者还是推荐使用 HIR 或 MIR 实现代数简化，那样可能会得到更好的优化效果。

7.7.2 代数简化的实现

从算法实现上来说，代数简化是一种代价极小的优化算法，甚至不需要数据流分析的支持。设计代数简化算法时，有个问题需要注意，就是不要把代数简化过于神化了。事实上，试图运用代数简化解决一切运算化简的问题是徒劳的。这种想法不但会大大增加算法设计的难度，而且也会影响算法执行的效率。这里，Neo Pascal 只实现了一个简单的算法框架，并不力求解决那些复杂的运算化简问题。

另外，还应该注意一些特定的代数性质，例如运算的交换律、结合律等。其中，交换律是最常见的代数性质。在很多情况下，如果交换律处理得不好，就会使源代码变得极其冗长。当然，要解决这类问题的方法是非常简单的，笔者将在源代码分析中详述。

程序 7-15 IRSimplify.cpp

```

1 void CIRSimplify::AlgebraicSimplify(int iProcIndex)
2 {
3     if (Num2Bit.empty())
4     {

```

```

5         int j=1;
6         for(int i=1;i<31;i++)
7         {
8             j=j*2;
9             Num2Bit.insert(pair<int,int>(j,i));
10        }
11    }
12    for (int i=0;i<SymbolTbl.ProcInfoTbl.at(iProcIndex).m_Codes.size();i++)
13    {
14        IRCode* TmpIR=&SymbolTbl.ProcInfoTbl.at(iProcIndex).m_Codes[i];
15        Opti_Tbl* TmpOpti;
16        if (TmpIR->m_eOpType<0)
17            TmpOpti=SearchOptiTbl(OpType::TypeCast);
18        else if (TmpIR->m_eOpType>>4>=21)
19            TmpOpti=SearchOptiTbl(TmplR->m_eOpType);
20        else if (TmpIR->m_eOpType>>4<21)
21            TmpOpti=SearchOptiTbl((OpType)(TmpIR->m_eOpType));
22        if (TmpOpti!=NULL && TmpOpti->eAlgebraicProcess!=Opti_Tbl::None)
23        {
24            if (TmpOpti->eCommutative==Opti_Tbl::Commutative)
25            {
26                if (TmpIR->m_Op1.m_iType==OpInfo::CONST)
27                {
28                    OpInfo TmpOp=TmpIR->m_Op1;
29                    TmpIR->m_Op1=TmpIR->m_Op2;
30                    TmpIR->m_Op2=TmpOp;
31                }
32            }
33            if ((TmpIR->m_eOpType>>4<<4)==OpType::ADDI)
34            {
35                if (TmpIR->m_Op2.m_iType==OpInfo::CONST &&
36                    SymbolTbl.ConstInfoTbl[TmpIR->m_Op2.m_iLink].m_fVal==0)
37                {
38                    TmpIR->m_eOpType=(OpType)TmpOpti->eAlgebraicProcess;
39                    TmpIR->m_Op2.m_iType=OpInfo::NONE;
40                    bOptiChanged=true;
41                    continue;
42                }
43            }
44            if ((TmpIR->m_eOpType>>4<<4)==OpType::SUBI)
45            {
46                if (TmpIR->m_Op2.m_iType==OpInfo::CONST &&
47                    SymbolTbl.ConstInfoTbl[TmpIR->m_Op2.m_iLink].m_fVal==0)
48                {
49                    TmpIR->m_eOpType=(OpType)TmpOpti->eAlgebraicProcess;
50                    TmpIR->m_Op2.m_iType=OpInfo::NONE;

```




```
51         bOptiChanged=true;
52         continue;
53     }
54 }
55 if((TmpIR->m_eOpType>>4<<4)==OpType::DIVI ||
56    (TmpIR->m_eOpType>>4<<4)==OpType::MOD ||
57    (TmpIR->m_eOpType>>4<<4)==OpType::MULI )
58 {
59     if (TmpIR->m_Op2.m_iType==OpInfo::CONST &&
60         SymbolTbl.ConstInfoTbl[TmpIR->m_Op2.m_iLink].m_fVal==1)
61     {
62         TmpIR->m_eOpType=(OpType)TmpOpti->eAlgebraicProcess;
63         TmpIR->m_Op2.m_iType=OpInfo::NONE;
64         bOptiChanged=true;
65         continue;
66     }
67 }
68 if((TmpIR->m_eOpType==OpType::MUL_4|| TmpIR->m_eOpType==OpType::DIV_4)&&
69    (TmpIR->m_Op2.m_iType==OpInfo::CONST))
70 {
71     map<int,int>::iterator it=Num2Bit.find(SymbolTbl
72         .ConstInfoTbl[TmpIR->m_Op2.m_iLink].m_iVal);
73     if (it!=Num2Bit.end())
74     {
75         ConstInfo TmpConst=SymbolTbl.ConstInfoTbl[TmpIR->m_Op2.m_iLink];
76         TmpConst.m_iVal=it->second;
77         TmpConst.m_szName=itos(it->second);
78         TmpIR->m_Op2.m_iLink=SymbolTbl.RecConstTbl(TmpConst);
79         if (TmpIR->m_eOpType==OpType::MUL_4)
80             TmpIR->m_eOpType=OpType::SHL_4;
81         else
82             TmpIR->m_eOpType=OpType::SHR_4;
83         continue;
84     }
85 }
86 if((TmpIR->m_eOpType>>4<<4)==OpType::SHL ||
87    (TmpIR->m_eOpType>>4<<4)==OpType::SHR)
88 {
89     if (TmpIR->m_Op2.m_iType==OpInfo::CONST &&
90         SymbolTbl.ConstInfoTbl[TmpIR->m_Op2.m_iLink].m_iVal==0)
91     {
92         TmpIR->m_eOpType=(OpType)TmpOpti->eAlgebraicProcess;
93         TmpIR->m_Op2.m_iType=OpInfo::NONE;
94         bOptiChanged=true;
95         continue;
96     }
97 }
```

```

97     }
98     if((TmpIR->m_eOpType>>4<<4)==OpType::OR)
99     {
100         if (TmpIR->m_Op2.m_iType==OpInfo::VAR &&
101             TmpIR->m_Op1.m_iType==OpInfo::VAR &&
102             TmpIR->m_Op1.m_iLink==TmpIR->m_Op1.m_iLink)
103         {
104             TmpIR->m_eOpType=(OpType)TmpOpti->eAlgebraicProcess;
105             TmpIR->m_Op2.m_iType=OpInfo::NONE;
106             bOptiChanged=true;
107             continue;
108         }
109         if (TmpIR->m_Op2.m_iType==OpInfo::CONST &&
110             SymbolTbl.ConstInfoTbl[TmpIR->m_Op2.m_iLink].m_bVal==false)
111         {
112             TmpIR->m_eOpType=(OpType)TmpOpti->eAlgebraicProcess;
113             TmpIR->m_Op2.m_iType=OpInfo::NONE;
114             bOptiChanged=true;
115             continue;
116         }
117         if ((TmpIR->m_Op2.m_iType==OpInfo::CONST &&
118             SymbolTbl.ConstInfoTbl[TmpIR->m_Op2.m_iLink].m_bVal==true))
119         {
120             TmpIR->m_eOpType=(OpType)TmpOpti->eAlgebraicProcess;
121             TmpIR->m_Op1=TmpIR->m_Op2;
122             TmpIR->m_Op2.m_iType=OpInfo::NONE;
123             bOptiChanged=true;
124             continue;
125         }
126     }
127     if((TmpIR->m_eOpType>>4<<4)==OpType::AND)
128     {
129         if (TmpIR->m_Op2.m_iType==OpInfo::VAR &&
130             TmpIR->m_Op1.m_iType==OpInfo::VAR &&
131             TmpIR->m_Op1.m_iLink==TmpIR->m_Op1.m_iLink)
132         {
133             TmpIR->m_eOpType=(OpType)TmpOpti->eAlgebraicProcess;
134             TmpIR->m_Op2.m_iType=OpInfo::NONE;
135             bOptiChanged=true;
136             continue;
137         }
138         if (TmpIR->m_Op2.m_iType==OpInfo::CONST &&
139             SymbolTbl.ConstInfoTbl[TmpIR->m_Op2.m_iLink].m_bVal==true)
140         {
141             TmpIR->m_eOpType=(OpType)TmpOpti->eAlgebraicProcess;
142             TmpIR->m_Op2.m_iType=OpInfo::NONE;

```



```
143         bOptiChanged=true;
144         continue;
145     }
146     if (TmpIR->m_Op2.m_iType==OpInfo::CONST &&
147         SymbolTbl.ConstInfoTbl[TmpIR->m_Op2.m_iLink].m_bVal==false)
148     {
149         TmpIR->m_eOpType=(OpType)TmpOpti->eAlgebraicProcess;
150         TmpIR->m_Op1=TmpIR->m_Op2;
151         TmpIR->m_Op2.m_iType=OpInfo::NONE;
152         bOptiChanged=true;
153         continue;
154     }
155 }
156 }
157 }
158 }
```

第 3~10 行：主要用于初始化 Num2Bit 映射表。这个表描述的就是乘、除运算中特定常量与相应移位运算的位数之间的关系。例如， $n*8=n \text{ shl } 3$ ，就可以将 $\langle 8, 3 \rangle$ 存入 Num2Bit 备用。有读者可能有疑问，为什么不直接使用 log 函数计算呢？主要是考虑到 log 函数的结果是浮点数，而细小误差可能会导致非常严重的错误。

第 12 行：遍历 IR 列表。

第 14 行：获取当前 IR。

第 16~21 行：获取处理方案。

第 24~32 行：如果运算是满足交换律的，则将 IR 中常量操作数统一置于操作数 2 中。这样做可以避免编写许多不必要的判断及处理。当然，这项工作也可以在生成 IR 时完成。理论上讲，操作数 1、操作数 2 都是常量的 IR 是不存在的，应该已经被常量折叠优化了。

第 35~42 行：依据等式 $i+0=i$ ，将 IR 转换为赋值形式。这里，不必考虑 $0+i=i$ 的情况，因为在第 24~32 行中已经进行了操作数交换。以下满足交换律的等式的处理方法类似。

第 46~53 行：依据等式 $i-0=i$ ，将 IR 转换为赋值形式。

第 59~66 行：依据等式 $i*1=i$ 、 $i \text{ div } 1=i$ 、 $i \text{ mod } 1=i$ ，将 IR 转换为赋值形式。

第 71~84 行：将乘以、除以一个常量 2^n 的 IR 转换为移位 IR。

第 89~96 行：依据等式 $i \text{ shl } 0=i$ 、 $i \text{ shr } 0=i$ ，将 IR 转换为赋值形式。

第 100~108 行：依据等式 $i \text{ or } i=i$ ，将 IR 转换为赋值形式。

第 109~116 行：依据等式 $i \text{ or } \text{false}=i$ ，将 IR 转换为赋值形式。

第 117~125 行：依据等式 $i \text{ or } \text{true}=\text{true}$ ，将 IR 转换为赋值形式。

第 129~137 行：依据等式 $i \text{ and } i=i$ ，将 IR 转换为赋值形式。

第 138~145 行：依据等式 $i \text{ and } \text{true}=i$ ，将 IR 转换为赋值形式。

第 146~164 行：依据等式 $i \text{ and } \text{false}=\text{false}$ ，将 IR 转换为赋值形式。

关于代数简化的实现，就讨论至此。

7.8 跳转优化

7.8.1 跳转优化基础

严格地说，跳转优化（jump optimization）并不是特指某一个优化算法，而是一类优化算法的总称，它们能够完成大多数分支、跳转相关的优化工作，有时，亦称为“分支优化”。与先前讨论的优化算法不同，跳转优化是比较典型的控制流优化算法。因此，它基本上都是基于流图进行的，很少依赖于数据流分析的支持。在实践中，虽然跳转优化经常被应用于 IR 优化及目标代码优化中，但并不太适用于 HIR。现代编译技术的观点认为跳转优化的最佳时刻是在代码最终形态基本确定之后。

实际上，跳转优化的思想并不复杂，已经成为了编程的规范，只是读者并没有察觉而已。下面，来谈两个最常见的跳转优化方案。

1. 条件跳转优化

有一个事实是不得不承认的，那就是条件跳转的复杂程度要远远大于无条件跳转。实际上，条件跳转优化的思想是非常简单的，就是将永真或永假的条件跳转替代为无条件跳转，或者将其删除。首先，读者可能会有一个疑问，即永真或永假的条件跳转是否存在？回答是肯定的。在 IR 生成过程中，编译器设计者会避免永真或永假的条件跳转，保证原始生成的 IR 不存在此类跳转。但是，常量传播、常量折叠以及代数简化等算法为我们做了很多工作，因此，现阶段的 IR 与原始生成的 IR 相比，早已是面目全非了。

一般来说，针对永真或永假的条件跳转，主要采用如表 7-9 所示的优化方式。从表面上来看，这种跳转优化的意义可能难以理解，尤其是将 JT、JNT 转换为 JMP，似乎并没有节省任何 IR。不过，这种优化对于检测不可到达代码的作用是非常显著的。举个简单的例子，读者应该不难想象，假设把 JT 转换为 JMP，那么，就意味着 JT 跳转的假分支代码极有可能变成不可到达代码。同样，JNT 的情况也是类似的。下面，来看一个条件跳转优化的实例。

表 7-9 条件跳转的优化方式

原始 IR	优化方式
JNT true Label	删除当前 IR
JNT false Label	替换为 JMP Label
JT true Label	替换为 JMP Label
JT false Label	删除当前 IR

例 7-7 条件跳转优化的实例。见表 7-10。

正如先前所述，基于原始 IR 直接讨论条件跳转优化的意义并不大。通常，条件跳转优化的基本环境是由其他 IR 优化算法创造的，而不是来自原始 IR。因此，笔者并没有给出原始的 IR 形式。在本例中，不难发现，(c)列中的第 3、4、5 行 IR 都是不可到达代码，也就是在任何情况下都无法执行的代码。在稍后的不可到达代码删除优化中，就可以将这部分代码删除。

表 7-10 条件跳转优化的实例方式

(a) 输入源程序	(b) 条件跳转优化前的 IR	(c) 条件跳转优化后的 IR
var a,b,c,d:integer;	0: (ASSIGN_4 ,2,null,A)	0: (ASSIGN_4 ,2,null,A)
begin	1: (ASSIGN_1 ,FALSE,null,_T1)	1: (ASSIGN_1 ,FALSE,null,_T1)
a:=2;	2: (JNT ,FALSE,null,_L3)	2: (JMP ,_L3,null,null)
if (a<1) then	3: (ADD_4 ,B,C,_T5)	3: (ADD_4 ,B,C,_T5)
a:=b+c+d;	4: (ADD_4 ,_T5,D,_T6)	4: (ADD_4 ,_T5,D,_T6)
result:=a;	5: (ASSIGN_4 ,_T6,null,A)	5: (ASSIGN_4 ,_T6,null,A)
end;	6: (LABEL ,_L3,null,null)	6: (LABEL ,_L3,null,null)
	7: (ASSIGN_4 ,A,null,RESULT)	7: (ASSIGN_4 ,A,null,RESULT)

2. 连续跳转优化

从一条跳转指令转移到另一条跳转指令的情况是极其常见的，尤其是当采用简单的代码生成策略时。例如：

```

JNT <条件> L1
.....
L1:
JMP L2
.....
L2:
.....

```

不难发现，由于 JNT 指令的跳转目标 L1 处的指令是另一条无条件跳转，因此，完全可以将 JNT 指令的目标直接设为 JMP 指令的目标，以减少目标程序中冗余的跳转指令。这种替换并不依赖于 JNT 指令中的条件值。与条件跳转优化不同，连续跳转优化并不会创造新的不可到达代码情形，它仅仅是通过优化跳转的形式，以提高目标程序的执行效率。

连续跳转优化既可以适用于 IR，也可以适用于目标代码。这里，只讨论 IR 优化中的连续跳转优化问题。在 IR 列表中，试图检测代码中的连续跳转情形并不复杂，只需要简单地查看每一条跳转的目标是不是另一条跳转指令即可。一般来说，可以将连续跳转的情形分为如表 7-11 所示的几种。

表 7-11 连续跳转优化的方案[⊖]

	JMP L1 L1: JMP L2 L2:	JMP L1 L1: JNT <条件> L2 L2:	JNT <条件> L1 L1: JMP L2 L2:	JNT <条件> L1 L1: JNT <条件> L2 L2:
原始 IR				
跳转优化后的 IR	JMP L2 L1: JMP L2 L2:	JNT <条件> L2 L1: JNT <条件> L2 L2:	JNT <条件> L2 L1: JMP L2 L2:	根据实际情况而定

⊖ 这里仅以 JNT 指令代表条件跳转。

前三种连续跳转情形的优化方案是不需要作任何论证的，它也不依赖于 JNT 指令的条件值，因此，相关实现也是非常简单的。下面，来分析最后一种情形。

最后一种情形比较复杂，并不是在任何情况下都可以优化的。仅当两个条件都为真时，才可能优化。看似这个要求并不苛刻，不过，稍加思考后，不难发现，在编译阶段，试图确定两个条件是否同时为真并不是一件容易的事情。当然，这里并不考虑两个条件为布尔常量的情况，假设编译器是无法计算出这两个条件的实际取值的。那么，在很多情况下，编译器将无法作出两个条件是否同为真的判定。注意，这个问题的关键就在于“很多情况”，而不是绝对不可能。有读者可能会有疑惑，既然两个条件的实际取值都不知道，那么，如何判定它们是否同为真呢？事实上，在特定的情况下，这是可能的。例如：

```

if a==1 goto L1
.....
L1: if a>=0 goto L2
.....

```

编译器并不能预知 $a==1$ 、 $a>=0$ 的实际取值，但可以得到一个结论，即当 $a==1$ 为真，则 $a>=0$ 必定为真。因此，可以将第 1 行代码优化为“if a==1 goto L2”。当然，在更多情况下，这个问题是不可判定的。例如：

```

if a==0 goto L1
.....
L1: if b==0 goto L2
.....

```

编译器试图确定 $a==0$ 与 $b==0$ 的取值关系是不可能的。理论上说，在特殊条件下，这种优化方式是可行的。但从实现的角度来说，在无法计算实际取值的情况下，设计一个判断两个条件是否同时为真的算法是非常困难的。而且其优化的效果也十分有限。因此，即使是 VC++ 之类的商用编译器也没有做相关分析与优化。

7.8.2 条件跳转优化的实现

笔者必须说明一点，在 Neo Pascal 中，虽然 JE 指令也是条件跳转 IR，但是并不做优化处理。因为 JE 指令仅仅应用于 CASE 语句翻译，而 CASE 语句的优化思想与条件跳转优化是大相径庭的，所以本小节并不涉及相关内容与技术。下面，来看看条件跳转优化的源代码实现。

程序 7-16 IRSimplify.cpp

```

1 void CIRSimplify::IRSimplify(int iProcIndex)
2 {
3     vector<IRCode> TmpCodes;
4     bool bFlag=false;
5     for (int i=0;i<SymbolTbl.ProcInfoTbl.at(iProcIndex).m_Codes.size();i++)
6     {
7         IRCode* TmpIR=&SymbolTbl.ProcInfoTbl.at(iProcIndex).m_Codes[i];
8         if (TmpIR->m_eOpType==OpType::JNT)

```



```
9      {
10          if (TmpIR->m_Op1.m_iType==OpInfo::CONST)
11          {
12              if (SymbolTbl.ConstInfoTbl.at(TmpIR->m_Op1.m_iLink).m_bVal==false)
13              {
14                  TmpIR->m_eOpType=OpType::JMP;
15                  TmpIR->m_Op1=TmpIR->m_Rslt;
16                  TmpIR->m_Rslt.m_iType=OpInfo::NONE;
17                  bOptiChanged=true;
18              }
19              else
20              {
21                  bFlag=true;
22                  continue;
23              }
24          }
25      }
26      if (TmpIR->m_eOpType==OpType::JT)
27      {
28          if (TmpIR->m_Op1.m_iType==OpInfo::CONST)
29          {
30              if (SymbolTbl.ConstInfoTbl.at(TmpIR->m_Op1.m_iLink).m_bVal==true)
31              {
32                  TmpIR->m_eOpType=OpType::JMP;
33                  TmpIR->m_Op1=TmpIR->m_Rslt;
34                  TmpIR->m_Rslt.m_iType=OpInfo::NONE;
35                  bOptiChanged=true;
36              }
37              else
38              {
39                  bFlag=true;
40                  continue;
41              }
42          }
43      }
44      TmpCodes.push_back(*TmpIR);
45  }
46  if (bFlag)
47      SymbolTbl.ProcInfoTbl.at(iProcIndex).m_Codes=TmpCodes;
48  }
```

第 5~45 行：遍历 IR 序列，逐一判断 IR 是否满足条件跳转优化的要求，如满足则进行优化。

第 7 行：获取当前 IR。

第 8 行：判断当前 IR 是否为 JNT 指令。

第 10 行：判断当前 IR 的 m_Op1 是否为常量。如果是变量，则不满足跳转优化的要求。

第 12 行：判断 `m_Op1` 的值是否为 `false`。

第 13~18 行：当前 IR 的形式为“JNT FALSE 标号”，因此，可进行如下优化：

JNT FALSE 标号 → JMP 标号

其中，将 `bOptiChanged` 置为 `true` 表示本次优化过程中代码发生了变化，以便编译器进一步优化。

第 21~23 行：删除当前 IR。这里并没有直接将当前 IR 删除，而是通过 `continue` 语句实现相应功能。这里是借助于临时代码列表 `TmpCodes` 实现 IR 删除的，即在遍历 IR 序列的过程中，将所有需要的 IR 加入到 `TmpCodes` 中（第 44 行），而略过不需要的 IR。然后，遍历完成后，则使用 `TmpCodes` 中的 IR 列表替代程序的实际 IR 序列（第 47 行）。因此，删除一条 IR，只需使用 `continue` 语句，保证程序不执行第 44 行代码即可。

第 26 行：判断当前 IR 是否为 JT 指令。

第 28 行：判断当前 IR 的 `m_Op1` 是否为常量。如果是变量，则不满足跳转优化的要求。

第 30 行：判断 `m_Op1` 的值是否为 `true`。

第 31~36 行：当前 IR 的形式为“JT TRUE 标号”，因此，可进行如下优化：

JT TRUE 标号 → JMP 标号

其中，将 `bOptiChanged` 置为 `true` 表示本次优化过程中代码发生了变化，以便编译器进一步优化。

第 38~41 行：删除当前 IR。

第 44 行：将当前 IR 插入 `TmpCodes` 中。

第 46 行：根据 `bFlag` 标志，判断是否重置实际的 IR 列表。注意，只有当遍历过程中出现了删除 IR 的动作，才需要重置，仅修改操作符、操作数是不需要重置的。

7.8.3 连续跳转优化的实现

相比条件跳转优化而言，连续跳转优化的实现稍复杂。不过，读者只要理解了先前的各种连续跳转情况及其优化策略，相信掌握其源代码实现是比较容易的。当然，Neo Pascal 也只处理了前三种连续跳转的情况，并没有考虑第四种情况。下面，就来详细看看其源代码实现。

程序 7-17 IRSimplify.cpp

```

1 void CIRSimplify::JumpSimplify(int iProcIndex)
2 {
3     vector<IRCode> TmpCodes;
4     bool bFlag=false;
5     map<int,int> LabelMap;
6     set<int> bHasVisit;
7     for (int i=0;i<SymbolTbl.ProcInfoTbl.at(iProcIndex).m_Codes.size();i++)
8     {
9         IRCode* TmpIR=&SymbolTbl.ProcInfoTbl.at(iProcIndex).m_Codes[i];
10        if (TmpIR->m_eOpType==OpType::LABEL)
11            LabelMap.insert(pair<int,int>(TmpIR->m_Op1.m_iLink,i));
12    }

```




```
13     for (int i=0;i<SymbolTbl.ProcInfoTbl.at(iProcIndex).m_Codes.size();i++)
14     {
15         bHasVisit.clear();
16         IRCode* TmpIR=&SymbolTbl.ProcInfoTbl.at(iProcIndex).m_Codes[i];
17         if (TmpIR->m_eOpType>>4!=21)
18             continue;
19         Opti_Tbl* TmpOpti=SearchOptiTbl(TmpIR->m_eOpType);
20         if (TmpOpti->eJumpType==Opti_Tbl::CondJump ||
21             TmpOpti->eJumpType==Opti_Tbl::NonCondJump)
22             {
23                 int j=TmpOpti->eJumpType==Opti_Tbl::NonCondJump?
24                     TmpIR->m_Op1.m_iLink:TmpIR->m_Rslt.m_iLink;
25                 while (bHasVisit.find(j)==bHasVisit.end())
26                     {
27                         int k=LabelMap.find(j)->second;
28                         while (SymbolTbl.ProcInfoTbl.at(iProcIndex)
29                             .m_Codes[k].m_eOpType==OpType::LABEL)
30                             {
31                                 bHasVisit.insert(SymbolTbl.ProcInfoTbl.at(iProcIndex)
32                                     .m_Codes[k].m_Op1.m_iLink);
33                                 k++;
34                                 if (k==SymbolTbl.ProcInfoTbl.at(iProcIndex).m_Codes.size())
35                                     {
36                                         break;
37                                     }
38                             }
39                         if (k==SymbolTbl.ProcInfoTbl.at(iProcIndex).m_Codes.size() ||
40                             SymbolTbl.ProcInfoTbl.at(iProcIndex).m_Codes[k].m_eOpType!=OpType::JMP)
41                             {
42                                 if (TmpOpti->eJumpType==Opti_Tbl::NonCondJump)
43                                     TmpIR->m_Op1=SymbolTbl.ProcInfoTbl
44                                         .at(iProcIndex).m_Codes[k-1].m_Op1;
45                                 else
46                                     TmpIR->m_Rslt=SymbolTbl.ProcInfoTbl
47                                         .at(iProcIndex).m_Codes[k-1].m_Op1;
48                             }
49                         else
50                             j=SymbolTbl.ProcInfoTbl.at(iProcIndex).m_Codes[k].m_Op1.m_iLink;
51                     }
52             }
53     }
54 }
```

第 5 行：LabelMap 主要用于存储标号与其实际位置的映射。例如，L1 在标号信息表中的序号是 6，而 LABEL L1 语句在 IR 列表中的实际位置是 8，那么，就用<6, 8>来描述标号与其实际位置的映射关系。

第 6 行: bHasVisit 主要用于标识一次连续跳转优化过程中所有已被访问的标号, 以免一个标号被两次访问。这主要是为了防止在优化过程中处理环状跳转结构时出现死循环。设置 bHasVisit 集合就可以保证一个标号不会被第二次访问。

第 7~12 行: 遍历 IR 序列, 设置标号的映射关系。

第 13 行: 遍历 IR 序列。

第 15 行: 开始一次新的连续跳转优化之前, 先清空 bHasVisit 集合。

第 16 行: 获取当前 IR。

第 19 行: 获取处理方案。

第 23、24 行: 根据 IR 的跳转类型, 获取标号的序号。在 JMP 跳转 IR 中, 标号存储于 m_Op1 中。而在 JNT、JT 跳转 IR 中, 标号存储于 m_Rslt 中。

第 25 行: 检索连续跳转的终点标号。

第 27 行: 根据 LabelMap 的映射, 获取当前跳转 IR 的目的标号的位置。

第 28~38 行: 从目的标号 IR 开始, 逐一向后检索第一条非标号 IR。在此过程中, 将所有访问过的标号记录到 bHasVisit 集合中。如果检索直至 IR 列表表尾, 则退出 while 循环。

第 39、40 行: 如果检索到表尾或第一条非标号 IR 不是 JMP, 那么, 本次连续跳转优化不必继续进行了, 直接用新获取的标号替换原 IR 中的标号即可。

第 42~47 行: 直接使用顺序检索过程中最后访问的标号替代原 IR 中的标号即可。

第 50 行: 如果检索到的语句是 JMP, 那么, 按该 JMP 语句的目的标号继续进行优化。

7.9 冗余代码删除

7.9.1 冗余代码删除基础

事实上, 冗余代码是一个很宽泛的概念, 它可以泛指程序中一切无用代码。这里, 只讨论两种较常见的形式, 即死代码和不可到达代码。在编译技术中, 死代码 (dead code) 与不可到达代码 (unreachable code) 是完全不同的概念。死代码就是指那些确实对计算结果不起作用的代码。而不可到达代码就是指那些无论输入数据是什么都不可能执行的代码。这里, 举两个 C 语言的实例, 以便读者更直观地理解这两个概念。见表 7-12。

表 7-12 死代码、不可到达代码的实例

(a) 死代码实例	(b) 不可到达代码实例
<pre>#include "stdio.h" main() { int i,j; scanf("%d",&i); j=2+i; i=4; //死代码 printf("%d",j); }</pre>	<pre>#include "stdio.h" main() { int i=1,j; goto L1; i=2; //不可到达代码 L1: j=4+i; printf("%d",j); }</pre>



从程序流程来说，(a)列中的“i=4”是可以被执行到的，不过，是否执行该语句对运行结果是没有影响的。而(b)列中的“i=2”是不可能被执行的，当然，讨论该语句是否会影响运行结果是没有意义的。

在许多编译器经典著作中，关于这两个概念有非常明确的阐述。可惜的是许多国内的编译原理书籍往往将这两个概念混为一谈，最常见的观点认为死代码就泛指一切冗余代码，而对不可到达代码的概念就完全淡化了，这样的理解是值得商榷的。实际上，读者并不需要关注这两个概念本身，它们只不过是两个名词而已。而设计者关注的是如何删除冗余代码。不过，由于各种冗余代码的删除算法往往是不同的，因此，针对不同的实际问题设计相应的算法，这才是笔者强调区分死代码与不可到达代码的意义所在。下面，就分别讨论死代码及不可到达代码删除的相关问题。

1. 死代码删除

如果编译器有比较完善的数据流分析机制，死代码分析就将变得比较容易了。最简单的方法就是根据定值 IR 的 du 链信息判断其是否为死代码。如果 du 链为空，则表示本次定值不能到达任何引用点，即可作为死代码删除。

值得注意的是，编译器必须保证被删死代码没有执行任何附带动作，例如，设置标志位、触发中断等。如果死代码的附带动作会影响到其他活跃代码，那么，即便两者之间不存在任何引用关系，该死代码也是不能被删除。

这里，只讨论基于全路径的死代码删除问题。事实上，死代码删除并不一定是基于全路径讨论的。在现代优化技术中，基于部分路径的死代码删除是一个比较热门的研究领域，即某一定值 IR 在部分路径上是死代码，而在其他的路径上可能不是死代码。在这种情况下，讨论死代码的分析与删除是比较复杂的，将涉及许多代码变换方面的工作。早在 1994 年，Knoop、Rüthing、Steffen 就提出了部分路径死代码删除的思想。近年来，还有一些部分路径死代码删除算法是基于区域分析实现的。与全路径的死代码删除相比，部分路径的死代码删除的适应能力更强，当然，优化效果也更好。有兴趣的读者可以参阅相关论文及学术资料。

2. 不可到达代码删除

分析不可到达代码的基础思想并不复杂，就是从流图的入口基本块开始，沿向下流的边遍历整个流图，所有能够被访问到的基本块即为可达的，而其余的基本块则是不可到达的。至此，请读者仔细考虑一个问题：是否可以依据基本块的 InSet 集合来判定不可到达代码呢？那样，算法实现就会更简单。读者可能认为通过判断 InSet 集合是否为空来确定该基本块是否可达似乎也是可行的。实际上，也不能认为这种观点是错误的，只是不全面而已。因为有些不可到达代码是无法通过 InSet 集合判定的。例如，如图 7-9 所示，if 真分支内的语句明显是不可到达代码。但是，在整个流图中，除了入口基本块以外，其他基本块的 InSet 都是不为空的。因此，仅依据 InSet 集合判定不可到达代码是不彻底的。

```
a=2;
if (a<0) then
{
L1:
    a=1;
    goto L1;
}
b=a+1;
```

图 7-9 不可到达代码

7.9.2 死代码删除的实现

在 Neo Pascal 中，判定一条赋值 IR 是否为死代码的标准就是其 du 链是否为空。这个判

定标准是相对简单的，因此，算法实现也不复杂。不过，这种简单的实现方法并不完美，主要是由于它的效率相对较低，整个优化过程可能需要多次迭代才能完成。下面，就来看看相关源代码的实现细节。

程序 7-18 CodeElimination.cpp

```

1 void CDeadCodeElimination::DeadCodeElimination(int iProcIndex)
2 {
3     bool bChange=true;
4     while (bChange)
5     {
6         bChange=false;
7         for (int i=0;i<SymbolTbl.ProcInfoTbl.at(iProcIndex).m_Codes.size();i++)
8         {
9             IRCode* TmpIR=&SymbolTbl.ProcInfoTbl.at(iProcIndex).m_Codes[i];
10            Opti_Tbl* TmpOpti;
11            if (TmpIR->m_eOpType<0)
12                TmpOpti=SearchOptiTbl(OpType::TypeCast);
13            else if (TmpIR->m_eOpType>>4>=21)
14                TmpOpti=SearchOptiTbl(TmpIR->m_eOpType);
15            else if (TmpIR->m_eOpType>>4<21)
16                TmpOpti=SearchOptiTbl((OpType)(TmpIR->m_eOpType>>4<<4));
17            if (TmpOpti!=NULL && TmpOpti->eDeadCode!=Opti_Tbl::None)
18                {
19                    if(TmpIR->m_Rslt.m_iType==OpInfo::VAR && !TmpIR->m_Rslt.m_bRef &&
20                        SymbolTbl.VarInfoTbl[TmpIR->m_Rslt.m_iLink].m_eRank==VarInfo::VAR &&
21                        TmpIR->m_Rslt.m_duChain.size()==0)
22                        {
23                            bChange=true;
24                            bOptiChanged=true;
25                            TmpIR->m_bEnabled=false;
26                        }
27                }
28            }
29            if (bChange)
30                DeleteCode(iProcIndex);
31        }
32    }

```

第 3 行：初始化优化结束标志。

第 6 行：重置优化结束标志。

第 7~27 行：遍历 IR 列表。

第 9 行：获取当前 IR。

第 11~16 行：获取处理方案。

第 19~21 行：判断当前 IR 的 du 链是否为空，为空则表示当前 IR 是死代码。当然，这里并不考虑结果操作数间接寻址访问的情况。



第 23 行：设置 bChange 标志。

第 24 行：设置 bOptiChanged 标志。

第 25 行：将当前 IR 的 m_bEnabled 设为 false，以便后续模块进行统一删除。。

第 30 行：根据 IR 的 m_bEnabled 标志，将冗余 IR 实际删除。

7.9.3 不可到达代码删除的实现

前面已经讲述了不可到达代码分析是基于流图遍历实现的。通常，不可到达代码都是以基本块为单位分析的，也就是说，如果某一行 IR 是不可到达代码，那么，其所属的基本块必定也是不可到达的。因此，从算法实现来说，可能更多关注的是不可到达基本块。分析不可到达基本块大致有如下几步：

1) 从流图入口基本块开始，沿向下流的边（即结点的出度）遍历整个流图。

2) 遍历过程中，记录所有访问过的基本块（结点），将其暂存于一个集合中，即 Neo Pascal 中的 ReachableBlock 集合。

3) 最后，遍历所有基本块，如果基本块属于暂存集合，则表示该基本块是可到达的。反之，则表示该基本块是不可到达的。

显然，遍历流图是分析算法的核心。流图是标准的图结构，因此，数据结构中关于图遍历的算法都是适用的。注意，编译器并不关注到底采用什么方式实现遍历，因为遍历顺序对分析结果不会有任何影响。更多关注的是遍历算法的实现代价及效率。在分析流图遍历算法之前，先引入两个全局数据结构，声明形式如下：

【声明 7-12】

```
set<int> ReachableBlock;    //可到达基本块的集合
set<int> VisitedBlock;     //已访问基本块的集合
```

ReachableBlock 集合描述的是从流图入口出发可到达的所有基本块（序号）。也就是说，不存在于 ReachableBlock 集合中的基本块就是不可到达的基本块。而优化算法就是依此集合实现冗余删除的。

VisitedBlock 集合的主要作用是标志已访问过的基本块，避免因流图中的环路而导致遍历算法死循环。

下面，就来看看流图遍历算法的实现细节。

程序 7-19 CodeElimination.cpp

```
1 void ReachPath(vector<CBasicBlock>* pBbs,int iPos)
2 {
3     if (VisitedBlock.count(iPos)!=0)
4         return;
5     else
6         VisitedBlock.insert(iPos);
7     vector<int>::iterator it=pBbs->at(iPos).DownFlow.begin();
8     for(;it!=pBbs->at(iPos).DownFlow.end();it++)
9     {
10        ReachableBlock.insert(*it);
```

```

11         ReachPath(pBbs,*it);
12     }
13 }

```

第3行：判断当前基本块（即 `iPos` 所标识的基本块）是否已经被访问过了。

第6行：将当前基本块序号加入 `VisitedBlock` 集合中。

第7~12行：依次递归调用 `ReachPath` 函数，深度遍历当前基本块的各向下流边的目的基本块。

图遍历算法是相对比较简单，不再赘述。下面继续讨论不可到达代码删除的问题。

程序 7-20 CodeElimination.cpp

```

1 void CDeadCodeElimination::DeadBlockElimination(int iProcIndex)
2 {
3     vector<CBasicBlock>* pBbs=&BasicBlock[iProcIndex];
4     ReachableBlock.clear();
5     VisitedBlock.clear();
6     ReachPath(pBbs,0);
7     if (ReachableBlock.size()!=pBbs->size())
8     {
9         for (int i=1;i<pBbs->size();i++)
10        {
11            if (ReachableBlock.count(i)==0)
12            {
13                for(int j=pBbs->at(i).iStart;j<=pBbs->at(i).iEnd;j++)
14                    SymbolTbl.ProcInfoTbl[iProcIndex].m_Codes[j].m_bEnabled=false;
15                bOptiChanged=true;
16            }
17        }
18        DeleteCode(iProcIndex);
19    }
20 }
21 }

```

第3行：获取当前过程的基本块信息。

第4行：清空 `ReachableBlock` 集合。

第5行：清空 `VisitedBlock` 集合。

第6行：遍历流图，计算可到达基本块集合。

第7行：判断是否存在不可到达的基本块。

第9~17行：在遍历流图基本块集合的过程中，将不可到达基本块的所有 IR 的 `m_bEnabled` 属性置为 `false`，即表示该 IR 是冗余的。

第18行：调用 `DeleteCode` 函数删除所有冗余 IR。

7.10 深入学习

在过去的数十年中，优化技术绝对是编译器设计领域中最炙手可热的话题。关于优化技术的经典论文不胜枚举，有兴趣的读者可以参考“鲸书”各章的参考文献，其中，详尽列出了各种优化技术发展历程中的经典论文与著作，本书不再赘述。首先，笔者推荐几个优化编译器的资源。

Open64 是一个开源的编译器后端。1999 年，SGI 发布了一个工业化的并行优化编译器 Pro64(TM)及其源代码，后来经过全世界多个编译器研究小组的改进，最终命名为 Open64。Open64 的设计结构优秀，分析优化全面，是高级编译优化技术的理想平台。Open64 的前端主要源于 GCC 2.95，目前，支持 C、C++、Fortran 90/95 等前端。读者可以访问 <http://www.open64.net/>，以获得更多的资源。

SUIF 是由 Monica Lam 和她在斯坦福大学的同事一起开发的一个研究型编译器。SUIF 并不是一个产品级系统，它只是用于优化技术与并行编译的研究。目前，SUIF 共有两个版本，分别是 SUIF1 和 SUIF2，SUIF1 是 1994 年诞生的，而 SUIF2 是 1999 年开发完成的。SUIF 是一个将 C、Fortran 77 转换为 MIPS 代码的编译器，当然，SUIF2 已经实现了 Compaq、Alpha、x86 的代码生成器。其中，还包括了数组依赖关系分析库、循环转换库、矩阵和线性不等式运算库、并行代码生成器和运行库、标量优化器等资源。读者可以访问 <http://suif.stanford.edu/>，以获得相关信息。

LANCE 也是一个优秀的开源编译器平台。它为开发一个高效的 C 编译器提供了一个系统平台，这主要是依赖于其精巧的设计结构。LANCE 包括 C 编译器前端、机器无关的优化组件、IR 分析与处理库、代码生成器等组件。LANCE 的主要目标就是嵌入式 C 编译器，因此，在工程领域，LANCE 获得广泛应用。读者可以访问 <http://www.lancecompiler.com/>，以获得相关资料。LANCE 的源代码是可以通过认可申请的。

最后，笔者再来推荐几本优化方面的经典著作，供读者参考学习。

- | | | |
|------------------------------------------------------|--------------------------------|-----------|
| 1、现代体系结构的优化编译器 | Randy Allen, Ken Kennedy | 机械工业出版社 |
| 说明：本书是以并行机为基础讨论优化技术的经典著作。该书涉及的理论与技术较新，不过，并不太适合初学者学习。 | | |
| 2、The Compiler Design Handbook | Y. N. Srikant, Priti Shankar | CRC Press |
| 说明：本书是以优化与代码生成为主的高级手册，详细阐述了基于不同目标机的优化与代码生成技术。 | | |
| 3、高级编译器设计与实现 | Steven S. Muchnick | 机械工业出版社 |
| 说明：这本书被誉为“鲸书”，其中涉及许多编译器设计的高级话题，当然，它最值得关注的就是优化技术。 | | |
| 4、编译器工程 | Keith D. Cooper, Linda Torczon | 机械工业出版社 |
| 说明：这是一本以介绍编译器实践工程为主的教材，堪称继“龙书”之后的另一本经典著作。 | | |

7.11 实践与思考

1. 请尝试基于 Neo Pascal 实现简单的别名分析机制。

2. 在优化技术中，如果试图得到更精准的活跃信息，可以基于 IR 迭代，而不是基于基本块迭代。请读者尝试修改 Neo Pascal 的活跃分析算法。

3. 从 IR 优化的角度，请读者思考 HIR 与 LIR 的优劣。
4. 请尝试基于 Neo Pascal 实现控制流分析算法。
5. 请尝试基于控制流分析结果实现循环优化。

7.12 大师风采——Richard Stallman

Richard Stallman: 美国著名的自由软件倡议者和计算机程序员，GNU 项目创始人，GCC、GDB、Emacs 等著名软件的开发者。1953 年 3 月 16 日出生于纽约。从高中时代开始，他就从事编程工作，如开发数据分析程序、APL 语言的编辑器、PL/I 的预处理器等。

1971 年，他进入哈佛大学学习，并成为麻省理工学院人工智能实验室的程序员。在此期间，他正式加入黑客阵营，成为职业黑客。1974 年，取得了哈佛大学学士学位后，他进入麻省理工学院主修物理。在人工智能实验室期间，Stallman 开发了 TECO、Emacs、基于 Lisp 机的操作系统，成为世界上最著名的黑客。

1983 年 9 月，Stallman 发起了 GNU 项目，并开始研发可自由使用的类 UNIX 操作系统。从此，GNU 正式登上了历史舞台。

第 8 章 运行时刻的存储管理

There are only two kinds of programming languages: those people always bitch about and those nobody uses.

——Bjarne Stroustrup

8.1 存储管理概述

8.1.1 存储区域

众所周知，在程序执行过程中，变量、常量、程序代码都需要占用一定的存储区域，有效地组织与分配存储区域是一个值得讨论的话题。通常，存储管理与具体目标机的体系结构是密切联系的，凭空讨论存储管理没有太大意义。在经典编译技术中，关于存储管理的理论并不太丰富，更多是凭借个人的经验与技巧完成的。因此，很多设计者更愿意将其视为一种管理艺术，而非科学技术。

本章涉及的存储管理是以讨论目标程序运行过程中的存储空间为重点，故也称为“运行时存储管理”。事实上，在编译器设计中，深入研究目标程序运行时的存储空间远比讨论编译器本身的存储空间更有意义。目标程序要实现一次运行，除了必要的可执行代码，还必须依托一个运行时环境。而许多与目标程序运行相关的工作都必须依赖这个环境，包括源程序中各种对象的存储分配及访问机制、参数传递、操作系统接口等。

首先引入几个重要的概念，这可能更有助于读者学习本章内容。

(1) 逻辑地址与物理地址。在通用机平台上，编译器生成的目标程序通常不是独享整个系统的存储资源的，其享有的存储空间是由操作系统依据特定的算法分配的。对于物理存储资源调配情况，编译器是完全无能为力的。那么，目标程序如何实现对存储空间的访问呢？事实上，这个过程是由编译器与操作系统甚至目标机协作完成的。下面，笔者举一个汇编程序存储分配的例子，如图 8-1 所示。

从 ASM 文件到实际运行的过程可以分为以下两个阶段：

第一，汇编、链接阶段。ASM 文件中并没有明确指出 A、B 符号的地址，只是说明了 A、B 符号所需的存储空间。经过汇编、链接之后，形成了可执行文件。一个可执行文件通常是以段的形式组织的，其中一部分用于存储数据（如变量、常量等），而另一部分用于存储可执行代码。在假设各段起始地址为 0 的情况下，汇编器会为各符号计算逻辑地址，即相对于段首地址的偏移。实际上，汇编、链接是两个不同的阶段，它们的工作职责也是完全不同的。囿于篇幅，本书不再深入阐述两者的差异。当然，并不排除用户在 ASM 中指定符号逻辑地址的可能性，在这种情况下，汇编器就只能按照用户的实际需要分配逻辑地址。注意，只是逻辑地址而已。

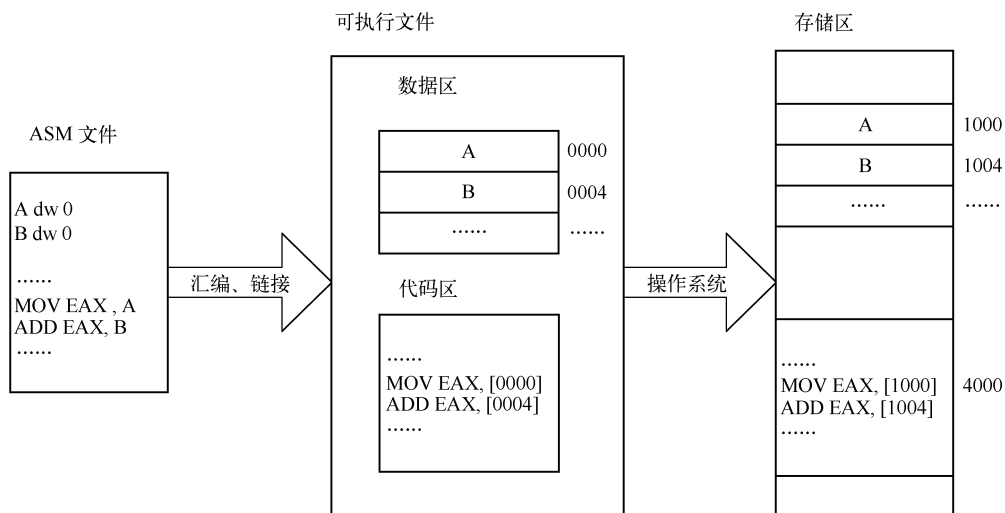


图 8-1 汇编程序的存储分配示例

第二，装入阶段。当一个可执行文件被执行时，操作系统会按照一定的调度顺序将可执行文件中的段读入物理存储空间中。通常，物理存储空间是共享的，完全是由操作系统管理与分配的。当某一数据段被装入到物理存储空间之后，操作系统会为各符号计算物理地址，即实际分配得到的存储地址，并修改相应的引用。至于操作系统如何确定引用点的问题并不难解决。通常，引用点的信息是由汇编器产生的，并写入可执行文件中。操作系统只需按照约定的格式读取。即可获得各引用点的信息。在操作系统中，这个过程被称为“重定位”。关于重定位更详细的内容属于操作系统讨论的范畴，这里就不再赘述。

(2) 逻辑地址空间的组织形式。用户程序的逻辑地址空间的管理与组织是由编译器、操作系统、目标机共同完成的。这里，笔者介绍一种最普遍的模式，如图 8-2 所示。通常，编译器会将逻辑地址空间划分为五个区域，即代码区、静态区、堆区、栈区、空闲区，并将用户程序中的各种程序数据分类存储。



图 8-2 逻辑地址空间模型

代码区主要用于存储程序代码。程序代码在程序运行过程中是保证不变的，对于拥有可写 ROM 的目标机结构来说，代码区一般是存放在 ROM 中。这样既保证了代码安全，也节省了有限的 RAM 空间。

静态区主要用于描述用户程序的静态数据，包括全局变量、静态变量等。通常，静态区允许用户程序通过地址直接进行寻址。

栈区、堆区主要用于解决程序运行过程中动态存储分配的问题。通常，用户程序的非静态局部变量都是有一定生存周期的，当某一函数执行完毕后，隶属于该函数的非静态局部变量的存储空间就会被释放。在程序设计语言中，这种动态分配、回收的例子并不少见，例如指针目标空间的分配、回收等。在静态区中，实现空间的动态分配、回收有相当的难度。然而，在栈式结构或堆式结构中实现这一功能是比较容易的，因此，设计者更愿意

选择栈、堆结构，而非静态结构。那么，动态存储分配是否必须基于栈或堆结构实现呢？严格来说，的确如此。不过，栈、堆结构并不是所有目标机都支持的，有些目标机可能只支持代码区、静态区。那么，编译器就不得不在静态区中模拟某些动态分配、回收的过程，以满足相应需求。即便如此，这种分配策略的本质仍然是静态的，是在编译过程中由编译器完成的。

8.1.2 存储布局

在存储分配之前，编译器还必须完成一项重要的工作，就是计算每个数据对象所需占用的存储空间大小。通常，数据对象的存储空间大小依赖于其数据类型。对于简单类型来说，计算空间大小是非常容易的。不过，对于复杂类型来说，就需要一定的技巧。数据对象的存储布局受目标机寻址约束的影响很大，合理的存储布局对提高程序的执行效率是非常有效的。这里，介绍一种比较常见的方案——整字对齐。

笔者先来介绍一下整字对齐的背景。所谓“字长”，简而言之，就是 CPU 一次读、写数据的最大长度。例如，32 位机指的就是一次读、写数据的最大长度为 32 位（4 字节）。在现代计算机体系结构中，为了便于设计，CPU 通常是按整字访问数据的，即地址必须能整除机器字长。如果数据不按规定存储，CPU 就不得不耗费多个周期进行读、写数据，并按实际情况进行拼接，这样会大大降低程序的执行效率。以 32 位机为例，如果用户将 4 字节数据存储在 0006h 为首地址的单元中，那么，CPU 就必须花费两个周期分别寻址 0004h（获取字的高两个字节，即 0006h、0007h 单元的字节数据，舍弃低两个字节数据）、0008h（获取字的低两个字节，即 0008h、0009h 单元中的字节数据，舍弃高两个字节数据），然后将相应的数据拼合成一个完整的 4 字节数据。

因此，避免这种情形的发生是非常有必要的。如果编译器仅仅依照声明的次序及数据对象的实际大小来安排存储布局，那么，发生这类问题的概率是非常大的。如图 8-3 所示。

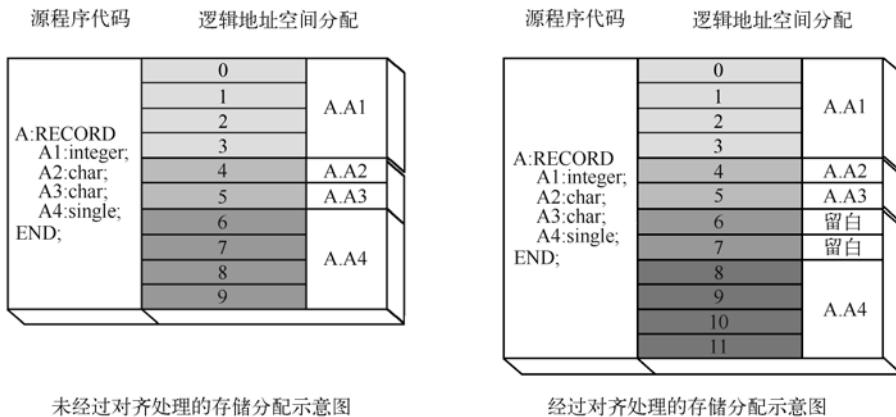


图 8-3 存储布局的比较

在图 8-3 的左图中，不难发现 A.A4 这个 4 字节数据的起始地址是不符合整字对齐要求的。最简单的解决方法就是适当地留白，实践证明，以少量的存储区域来换取效率的提高是完全值得的。不过，实际情况却并非如此简单。例如，声明 8-1 如下：

【声明 8-1】

```
A:RECORD
  A1:char;
  A2:integer;
  A3:char;
  A4:integer;
END;
```

根据先前的解决方法，编译器必须为 A1、A3 两个数据对象之后分别留白 3B 的空间。有效的数据空间仅为 10B，而留白的空间却达到了 6B，这恐怕是程序员与编译器设计者都不能接受的。因此，这个存储布局方案是值得商榷的。

在现代编译器设计中，人们提出了一种更有效的存储布局方案，就是打破声明次序的先后原则，按照数据对象占用空间的实际大小顺序，由小至大（或由大至小）地依次分配。这样，就能有效地避免因用户声明次序的不合理导致留白过多的问题。这种处理方法的实现代价较小，但对节省存储空间却是效果显著的。

下面，再来看看更一般化的存储布局问题。这里，读者应该了解一个概念，编译器存储分配的基本单位是什么？实际上，几乎所有的编译器都是以过程或函数为单位来分配存储区的。也就是说，编译器将一个过程或函数内的所有局部变量收集起来，组成一个数据块。而全局变量则是独立组成一个或多个数据块，这可能是根据目标机体系结构而定的。然后，编译器以数据块为单位进行存储分配。C、Pascal 等都是如此。当然，这里并不考虑优化技术产生的影响。在优化的前提下，讨论变量的存储分配是不合适的。那么，编译器为什么以过程或函数为单位进行存储分配呢？这里先不作解释，当读者阅读完本章内容后就明白了。

了解了存储分配的基本单位后，讨论基于整个用户程序的存储布局问题就变得非常容易了。实际上，编译器只需将一个过程或函数内的所有局部变量都看作是某一虚拟记录类型的字段列表，这样就可以依据记录类型变量的存储布局原则讨论所有用户变量的存储布局了。通常，可以得到若干以过程、函数为单位的逻辑数据块以及若干全局变量的数据块，这些就是存储分配的对象。为了便于讨论，本书假设只存在一个全局变量的数据块。

前面，主要讨论了存储布局的一些基本问题。无论是记录字段，还是局部变量的存储布局都涉及对数据对象的重新排序。因此，有两个要点是值得注意的：

(1) 重排的安全性。事实上，重排的安全性更多是取决于高级语言本身的设计规范，即语言各种数据对象的存储布局是否公开。如果语言向用户完全公开数据组织与存储布局结构，例如，规定记录的字段列表的组织顺序即为声明顺序，那么，编译器是不能随意更改其存储布局的，否则，一切不安全性都应该由编译器负责。然而，如果语言并没有明确公开存储布局，而用户仅仅是凭借经验推断，那么，重排可能带来的某些不安全性就应该由用户承担。典型的例子就是应用指针的自增方式来遍历函数局部变量或记录字段。

(2) 重排的方式。最常见的重排方式就是以占用空间大小排序，这几乎是没有争议的。不过，到底是以逆序还是顺序重排，不同编译器的实现是存在一定差异的。事实上，两者各有优缺点，并不能一概而论。针对不同的声明形式，读者只需进行少量实验，即可了解两种重排方式的优缺点。本书就不再深入讨论了。

至此，已经详细讨论了存储布局的相关话题，了解了离散的数据对象是如何被组织成数

据块的，并且明确了存储分配的基本单位是数据块。存储布局的设计思想更多是源于实践总结，并没有太多的理论支持。随着高级语言功能的丰富，存储布局的问题也是与时俱进的，因此，深入研究这一话题是有现实意义的。例如，托管机制、索引器等都对传统编译器的存储布局机制提出了挑战。不过，笔者想再次强调的是，存储布局通常只是讨论合理与否，要做到真正的最优却并不容易。

8.1.3 存储分配基础

本小节将详细分析基于数据块的存储分配，其中将涉及一些非常有趣的话题。那么，解决存储分配问题的关键是什么？简单来说，就是保证用户程序的各种数据对象都可以被正确地存取访问，不至于产生冲突。当然，这是基于语言的相关规则而言的，也就是说，在数据对象的作用域内，必须保证存取是有效的。下面看一个例子，如图 8-4 所示。

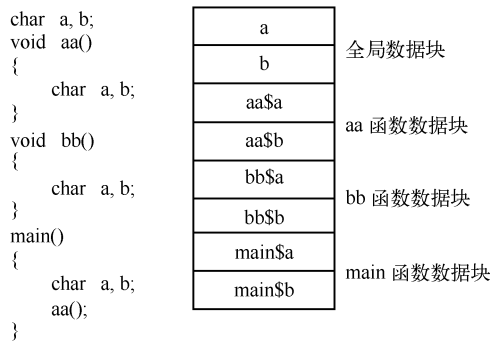


图 8-4 存储分配示例

根据先前的分析，编译器会为这个程序分配四个数据块，即全局数据块、aa 函数数据块、bb 函数数据块、main 函数数据块。这里，为了便于讨论，在局部变量的名字前冠以函数名，全局变量则不作更名。下面，就来看看这个程序的存储分配方案。

首先看一个最容易想到的解决方案，在不考虑空间回收的情况下，将各数据块（共 8 字节）依次分配到静态区即可。当然，此时的全局变量、局部变量的差异仅仅是一种语义范畴，它们的实际生存期是完全一样的。这种方案既不能有效地利用存储空间，也不能很好地体现变量生存期的差异，因此，其可行性比较差。

那么，是否可以考虑对这一分配方案进行合理的优化呢？由于局部变量的生存期仅限于函数执行期间，因此，某些函数数据块是可以共享同一存储区的。不过，这种共享是有前提的，就是必须保证函数之间不可以存在任何调用关系，包括直接调用或间接调用。在上例中，main、aa 之间是存在调用关系的，所以不能共享。而它们与 bb 之间是不存在调用关系的，故它们与 bb 是可以共享同一数据块的。因此，为上例分配 6B 的存储空间是可以保证程序正确性的。至于分析函数的调用关系对于编译器来说应该是易如反掌的。在编译技术中，通常，将这种分配方式称为“静态存储分配”。早期的 Fortran 编译器就是采用这种分配方案的，其可行性较先前要好一些。当然，在现代编译技术中，静态存储分配的实现可能比这里所讨论的方案要更优，并不满足于仅仅分析函数的调用关系，而是更深入地分析函数的控制流，试图在更大程度上实现空间的共享。然而，这种分析算法的实现却不简单，尤其是当语

言支持指针时，编译器可能需要付出更大的代价。在 i386 体系结构的目标机上，讨论静态存储分配的意义不是很大，这是因为这种方案本身仍然存在一定的不足之处。不过，在一些不支持堆栈结构的目标机上，静态存储分配可能是唯一的可选方案。

从表面上来看，静态存储分配方案似乎解决了存储分配的基本问题，事实却并非如此。这种分配方案的不足之处就在于它无法解决函数递归调用的问题。这是因为递归调用的深度是编译器无法预知的，当然，编译器也就不可能以静态方式分配存储空间。实践证明，即使是经典的 Fortran 编译器同样无法解决这一问题，因此，必须寻求更合适的动态分配方案。在现代编译技术中，提出了两种动态分配策略：栈式存储分配、堆式存储分配。

栈式存储分配：将过程或函数的局部数据对象分配在栈空间中。申请与回收存储空间付出的代价较小，便于实现。

堆式存储分配：与栈式存储不同，堆的管理通常离不开操作系统的协助。一般而言，操作系统会提供一个接口，用于申请与释放堆的空间。编译器同样生成相应的语句调用这个接口，实现空间的申请与释放。

在实际应用中，关于栈式存储分配的观点比较一致，一般都是由编译器管理、维护的，很少有语言会将栈的管理接口开放给程序员的，因此，对于程序员来说，栈结构是透明不可见的。不过，关于堆式存储分配的观点就不太一致了，不同的语言在这个问题上的实现差异比较大。有些语言将申请与释放堆空间的接口显式地暴露给程序员，一切基于堆的管理完全由程序员控制，相应的责任也由程序员承担，典型的例子就是 C、Pascal。然而，有些语言仅开放申请接口，而释放空间的琐事就由垃圾回收机制来完成，以减轻程序员维护堆的负担，例如，Java、C#等。当然，也有些语言通过某些特殊语法机制隐藏了堆的管理，让程序员感觉不到堆的存在。这两种分配策略并不是互斥的，更多时候，它们是相互合作、互为补充的。下面详细分析这两种分配策略的基本思想。

8.2 栈式存储分配

8.2.1 栈式存储分配基础

这里所讨论的栈主要指的是控制栈。实际上，控制栈与传统意义上的栈是有差异的，主要体现在两个方面：

(1) 可访问元素。传统的栈可访问的元素只有一个，也就是栈顶。无论入栈、出栈，其操作的对象只是栈顶而已，如果想访问栈的其他非栈顶元素是不可能实现的。然而，控制栈不但需要支持传统的入、出栈访问方式，还必须支持通过直接、间接等寻址方式访问栈的任意元素，否则将无法满足不同策略的需求。

(2) 栈的大小。用户在声明一个栈时，必须说明其大小。然而，编译器却不需要过多考虑控制栈的大小。换句话说，如果一个程序能够将系统栈耗尽，那么，这个程序的逻辑可能是值得商榷的，

在编译原理书籍中，一般很少讨论以上话题，这是因为常见的通用机所提供的系统栈都满足这两个条件。不过，笔者认为讨论以上话题是具有一定现实意义的。事实上，并不是所有的系统栈都适合作为控制栈的。在一些嵌入式系统中，有些栈结构可能容量比较小，有些

栈结构则不支持寻址访问或者入、出栈访问，这样的栈都不适合作为控制栈。

栈式存储分配的基本思想就是每当进入一个过程时，就在控制栈顶为该过程分配所需的数据空间，当一个过程工作完毕返回时，它在栈顶的数据空间也随即释放。习惯上，将那块与过程有关的数据空间称为活动记录（activation record），亦称为帧（frame），缩写为 AR。AR 的空间一般由以下几部分组成：过程局部变量的空间、实参列表、返回值空间、返回地址、机器状态等。在一次过程调用中，控制栈的变化是一个非常精巧的过程，要求编译器经过非常严谨的计算，方才得以实现。

根据所实现语言的不同，AR 的内容与布局也是不尽相同的。事实上，包括很多书在内，关于 AR 的观点也存在一定的差异。这里，笔者参考了“龙书”关于 AR 的描述，并结合 Neo Pascal 的实现，介绍 AR 的各基本组成部分。如图 8-5 所示。

(1) 实参。从经典编译技术的观点来说，实参是由过程调用点压入控制栈，供过程内部读取使用的。不过，在现代编译器设计中，参数传递更多地依赖于寄存器，而不是通过 AR 传递的。即便如此，编译器仍然需要在 AR 中预留适当的实参空间，想通过寄存器传参解决任意参数列表的传递问题是不可能的。Delphi 在这方面的处理是比较出色的，第一、二个参数通过 ecx、edx 传递，其他参数依然压入控制栈。这就是 fastcall 的调用约定，是继 stdcall、cdecl 之后另一种经典的调用约定。

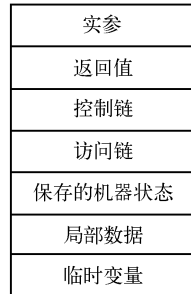


图 8-5 AR 结构示意图

(2) 返回值。返回值是由函数执行完毕后，返回给调用点的值。由于返回值的类型不同，返回值的空间大小各异。不过，一个函数最多只允许存在一个返回值，因此，借助于寄存器传递返回值相对容易得多。在现代编译器设计中，通过 eax 传递返回值几乎是公认的。

(3) 保存的机器状态。用于保存当前过程调用之前的机器状态信息，包括返回地址、通用寄存器的值等。其中，有些信息可能是由调用点或被调过程保存的，而有些信息则可能是由机器自动保存的。在 i386 体系结构中，返回地址就是由机器自动压栈保存的。当过程调用完毕返回时，必须将保存的机器状态恢复，以便调用点继续执行后续程序。

(4) 局部数据与临时变量。这两部分都是与过程的相关局部数据对象，从编译器设计的角度来说，在存储分配中，并不需要严格区分局部数据与临时变量。当然，如何存储布局？如何删除死变量？这些问题还是值得注意的。

(5) 控制链。指向调用者的 AR。

(6) 访问链。用于指向其他 AR 中可访问的非局部数据对象。

由于 Neo Pascal 和 C 都不支持嵌套过程声明，所以，本书将略过控制链与访问链的话题。不过，它们对于实现标准 Pascal、Alogl 60、ML 等是有实际意义的，有兴趣的读者可以参考相关资料。下面通过一个实例来分析过程调用的详细步骤。

例 8-1 过程调用的分析。代码如下所示。

```
char aa(char p1,char p2)
{
    char a,b;
    .....
}
```

```

main()
{
    char c,d;
    .....
    aa(c,d);
}

```

函数调用的 AR 变化如图 8-6 所示。

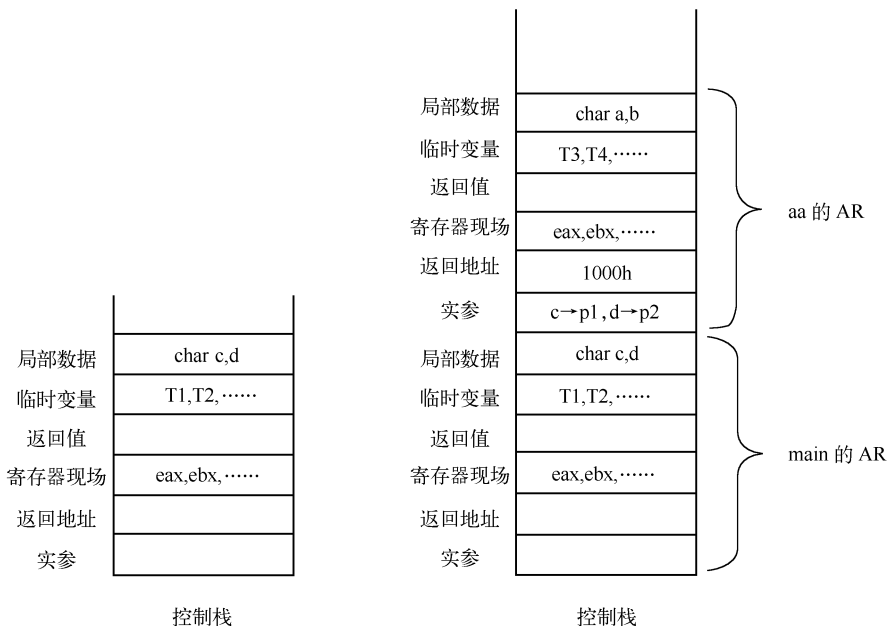


图 8-6 函数调用的 AR 变化

a) main 调用 aa 前的栈示意图 b) main 调用 aa 后的栈示意图

图 8-6a 的控制栈是调用 `aa` 前的状态，注意，`main` 函数也有一个 AR。当然，其中某些项允许为空，编译器不必分配相应的空间。这里，为了便于读者理解，故未将空表项删除。在本例中，临时变量是笔者假设存在的。`main` 函数在执行前同样需要保护寄存器现场。当然，对于 `main` 来说，这个动作看起来可能意义并不大。请读者结合图 8-6b，理解调用 `aa` 函数的大致过程。可以分为如下几步：

1) 参数传递。这里，假设使用 `stdcall` 方式传递参数。编译器会将实参自右向左依次压栈。实际上，实参与形参是共享同一片存储区的，`aa` 函数体内对形参的访问就直接变成了对实参的存取，这将是非常方便的。

2) 返回地址压栈。在正式执行 `aa` 函数之前，必须将返回地址压栈。前面，笔者已经介绍了返回地址压栈的方式很多，有些是由用户程序负责的，有些则是由机器自动完成的。不仅如此，压栈的地址也可能存在差异，有些是将 `aa` 函数调用点 `call` 指令所在地址压栈，而更多的是将 `aa` 函数调用点 `call` 指令之后的一条指令所在的地址压栈。这主要依赖于目标机的体系结构。本例假设返回地址是 `1000h`。

3) 转入 `aa` 函数。进入 `aa` 函数的程序体执行。注意，在转入 `aa` 函数执行时的栈顶是



“返回地址”。那么，当 aa 函数执行完后，返回 main 函数前，也必须保证栈顶仍然是“返回地址”。否则，将导致异常。

4) aa 函数的调用代码序列。转入 aa 函数后，率先执行的并不是 aa 函数的实际程序，而是编译器在 aa 函数体前插入的一段特定代码序列。这段代码序列主要用于生成 AR。其中，包括保护寄存器现场、分配临时数据、局部变量等存储空间。待 AR 构造完毕，就得到了一个完整的 AR 布局示意，如图 8-6b 控制栈所示。

5) aa 函数的返回代码序列。当 aa 函数完全执行完毕，必须执行一段由编译器生成插入在返回指令之前的代码序列。这段代码序列主要完成空间回收，并恢复寄存器现场。当然，也包括返回值处理。事实上，所谓“空间回收”并不是真正的数据清空，而只需将栈顶调整到“返回地址”即可，以保证函数能正常返回。

6) 返回 main 函数。当执行到返回指令时，从栈中弹出返回地址，根据返回地址返回调用点。原则上，弹出返回地址的动作应该由压栈者负责。也就是说，如果返回地址是由机器自动压的，那么，就由机器处理返回地址的弹出。否则，就由用户程序负责。

7) 清理现场。虽然程序得以继续正常执行，但是控制栈里却多了几个实参数据，这是不能接受的。通常，需要将栈顶恢复到传参之前的状况。那么，由谁来完成这项工作呢？关于这个问题存在两种观点，一种认为由调用者清理，一种认为由被调用者清理。cdecl 方式是前者的经典实例，而 stdcall 方式则实现了后者。从理论上来说，更规范、灵活的清理方式是前者，处理个数不定的参数列表时，它显得更方便。被调函数只关心如何引用参数，不需要过问参数的数量。而后者在处理个数不定的参数列表时，不得不将参数的个数同时传递给被调函数，因为其清理工作将在函数返回时工作。当然，后者也不是没有任何优势的，在处理个数确定的参数列表时，调用者可以省去很多烦心事。WINAPI 的标准调用方式就是 stdcall。

关于被调者清理现场的方案，还有一个问题需要了解，那就是如果目标机的返回指令功能比较弱时，实现由被调用者清理现场将是比较复杂的。这是因为无论是调用者还是被调用者清理现场，都必须保证清理现场是在返回指令执行后进行的。理由很简单，因为返回地址位于实参之上，一旦清理了现场，返回地址就将丢失。为了解决这个矛盾，现代目标机体系结构一般不仅支持简单的 RET 指令，还支持 RET X 指令（即返回后，同时退 X 字节的栈）。这样，由被调用者清理现场也并不困难。不过，并非所有的目标机都支持 RET X 指令。如果基于不支持这类指令的目标机，那么，就必须在返回指令之前，将“返回地址”暂存后清理现场，清理完后再将“返回地址”压栈，以保证正常返回调用点。

实际上，无论是普通调用，还是递归调用，都可以借助于这个模型实现。从理论上来说，只要栈空间允许，任意多级嵌套调用都是合法的。C、Pascal 等语言并没有规定函数调用的级数或递归的层次等。

8.2.2 i386 栈式存储分配

前面已经讲述了关于栈式存储分配的基本思想，本小节将基于一个具体的目标机模型深入讨论栈式存储分配的一些细节实现。为了便于读者更好地理解，先简单介绍一下 i386 系统栈相关的话题。

在 i386 体系结构中，系统栈是由 CPU 直接管理与维护的，它主要涉及两个寄存器：

ss、esp。在保护模式下，ss 寄存器存放的是段选择器，用户模式程序不应对其进行修改。在实模式下，ss 寄存器存放的就是系统栈的段首地址，通常是由用户程序指定的。而 esp 寄存器存放的是指向栈内特定位置的一个 32 位偏移值。一般来说，esp 寄存器指向的就是最后压入到系统栈上的数据。在实模式下，由于栈元素的地址只有 16 位，因此，只需要使用 esp 的低 16 位（即 sp 寄存器）即可。相对于实模式而言，现代编译器更多关注的是保护模式下的相关话题，因此，本书将重点讨论 32 位保护模式下的系统栈机制。关于系统栈的操作，主要包括两类：压栈操作、出栈操作。

压栈操作将 esp 指针减 4，并将值复制到 esp 指向的位置，如图 8-7 所示。

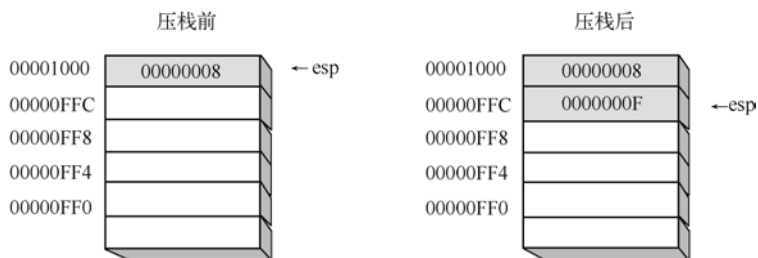


图 8-7 i386 系统栈压栈示意图

出栈操作将 esp 指向位置的值复制出来，并将 esp 指针加 4，如图 8-8 所示。

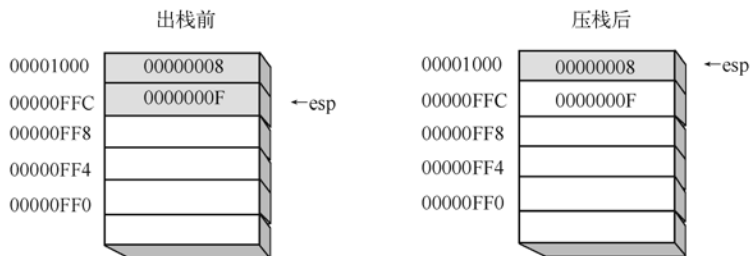


图 8-8 i386 系统栈出栈示意图

针对 i386 的系统栈，笔者作如下三点简单说明：

(1) 系统栈区域的初始状态。理论上说，系统栈区域的初始状态为空。不过，实际却并非如此。在 i386 体系结构中，系统栈区域的初始状态是随机的，其实际存储的值是完全取决于存储器的当前状态的。因此，为了避免出现意外错误，在函数初始化过程中，有些编译器设计者会考虑系统栈初始化的问题，例如，VC++ 是将“0xCC”填充即将使用的栈区域。注意，并不是初始化整个系统栈。

(2) 系统栈的扩展方向。关于这个问题，从图 8-7 及图 8-8 中，读者应该已经看出来。的确，与数据结构课程中描述的栈结构相比，i386 系统栈的扩展方向是向下扩展的，即随着栈内元素的增加，esp 指向的地址却是逐步减小的。有些读者可能会疑惑向下扩展系统栈是否有特殊的原因。事实上，这是没有任何原因的，只是 Intel 设计者的个人喜好而已。

(3) 出栈元素并不清除。与传统的栈结构类似，CPU 并不会刻意去清除那些已出栈的元素所在的存储区域。当有新的元素需要入栈时，CPU 就直接覆盖原有数据即可。以图 8-8 为例，虽然 00000FFC 单元的数据已经出栈，但 CPU 只对 esp 作了修改，却并不真正清除该数



据。直到有新的元素入栈时，CPU 就将其写入 00000FFC 单元中，此时，并不关注是否存在数据，而是直接无条件覆盖即可。

下面，笔者通过一个完整实例来介绍 i386 栈式分配的内核，以便读者深入理解这一精巧的处理过程。

例 8-2 i386 栈分配的实例，见表 8-1。

表 8-1 输入源程序与相应 IR 序列

输入源程序	IR 序列
program aa(input,output);	函数: AA
var i:integer;	0: (PARA ,2,null,I)
function	1: (PARA ,1,null,I)
add(i:integer;j:integer):integer;	2: (CALL ,ADD,null,null)
begin	3: (RETV ,null,null,_T1)
result:=i+j;	4: (ASSIGN_4 ,_T1,null,I)
end;	
begin	函数: ADD
i:=add(1,2);	0: (ADD_4 ,I,J,_T0)
end.	1: (ASSIGN_4 ,_T0,null,RESULT)

表 8-1 所示的程序是非常简单的，读者应该不难理解其 IR 序列的意义。这里，笔者特意关闭了优化选项（包括汇编级的窥孔优化），目的就是为了更好地还原调用过程的全貌。下面，笔者就详细列出相应的汇编程序代码，如程序 8-1 所示。

程序 8-1

```

1  .686P
2  .model flat,stdcall
3  option casemap: none
4  include PasLib.INC
5  .data                                ;数据段
6  ;ConstSymbolList:                   ;常量声明列表
7  __Const0=1
8  __Const1=2
9  ;VarSymbolList                       ;变量声明列表
10 __cvt DWORD 1 DUP(0)                ;变量转换使用的临时空间
11 __ADD$I=8                            ;add 函数的参数 i 的偏移
12 __ADD$J=12                           ;add 函数的参数 j 的偏移
13 __ADD$RESULT=-8                     ;add 函数的返回值的偏移
14 __ADD$_T0=-4                         ;add 函数的临时变量_T0 的偏移
15 __AA$I DWORD 1 DUP(0)                ;全局变量 i
16 __AA$_T1 DWORD 1 DUP(0)              ;全局临时变量_T1
17 .code                                ;代码段
18 ;function:AA
19 ;process of start
20 start:
21 __AA proc                             ;主程序段

```

```

22  push ebp                ;保护 ebp
23  mov ebp,esp            ;将 esp 的值赋给 ebp
24  sub esp,0              ;申请存储空间
25  push ebx               ;保护 ebx
26  push esi               ;保护 esi
27  push edi               ;保护 edi
28  ;(PARAM                ,2,null,J)
29  push dword ptr[___Const1 ] ;将常量 2 压入系统栈
30  ;(PARAM                ,1,null,I)
31  push dword ptr[___Const0 ] ;将常量 1 压入系统栈
32  ;(CALL                  ,ADD,null,null)
33  call _ADD              ;调用 add 函数
34  ;(RETV                  ,null,null,_T1)
35  mov dword ptr[_AA$_T1 ],eax ;获取返回值
36  ;(ASSIGN_4              ,_T1,null,I)
37  mov eax,dword ptr[_AA$_T1 ]
38  mov dword ptr[_AA$I ],eax
39  ;process of end
40  pop edi                ;恢复 edi
41  pop esi                ;恢复 esi
42  pop ebx                ;恢复 ebx
43  mov esp,ebp           ;恢复 esp
44  pop ebp                ;恢复 ebp
45  ret 0                  ;返回
46  _AA endp
47  ;function:ADD
48  ;process of start
49  _ADD proc              ;add 函数段
50  push ebp              ;保护 ebp
51  mov ebp,esp           ;将 esp 的值赋给 ebp
52  sub esp,8             ;申请存储空间
53  push ebx              ;保护 ebx
54  push esi              ;保护 esi
55  push edi              ;保护 edi
56  ;(ADD_4                 ,I,J,_T0)
57  mov eax,dword ptr[ebp+_ADD$I ] ;获取 add 函数的参数 i
58  add eax,dword ptr[ebp+_ADD$J ] ;将 add 函数的参数 i 与参数 j 相加
59  mov dword ptr[ebp+_ADD$_T0 ],eax ;将和赋给临时变量_T0
60  ;(ASSIGN_4              ,_T0,null,RESULT)
61  mov eax,dword ptr[ebp+_ADD$_T0 ]
62  mov dword ptr[ebp+_ADD$RESULT ],eax
63  ;process of end
64  mov eax,[ebp+_ADD$RESULT ] ;将返回值置入 eax
65  pop edi                ;恢复 edi
66  pop esi                ;恢复 esi
67  pop ebx                ;恢复 ebx

```



```

68   mov esp,ebp           ;恢复 esp
69   pop ebp              ;恢复 ebp
70   ret 8                 ;返回
71   _ADD endp
72   end start

```

这是一个标准的宏汇编程序，使用 MASM 6.15 汇编、链接得到可执行文件。笔者并不打算深入讲述汇编语言方面的话题，有兴趣的读者可以参考《Intel 汇编语言程序设计》一书。这里，假设 esp 的初始值为 1000H。下面，就来看看函数调用的详细过程，如图 8-9 所示。

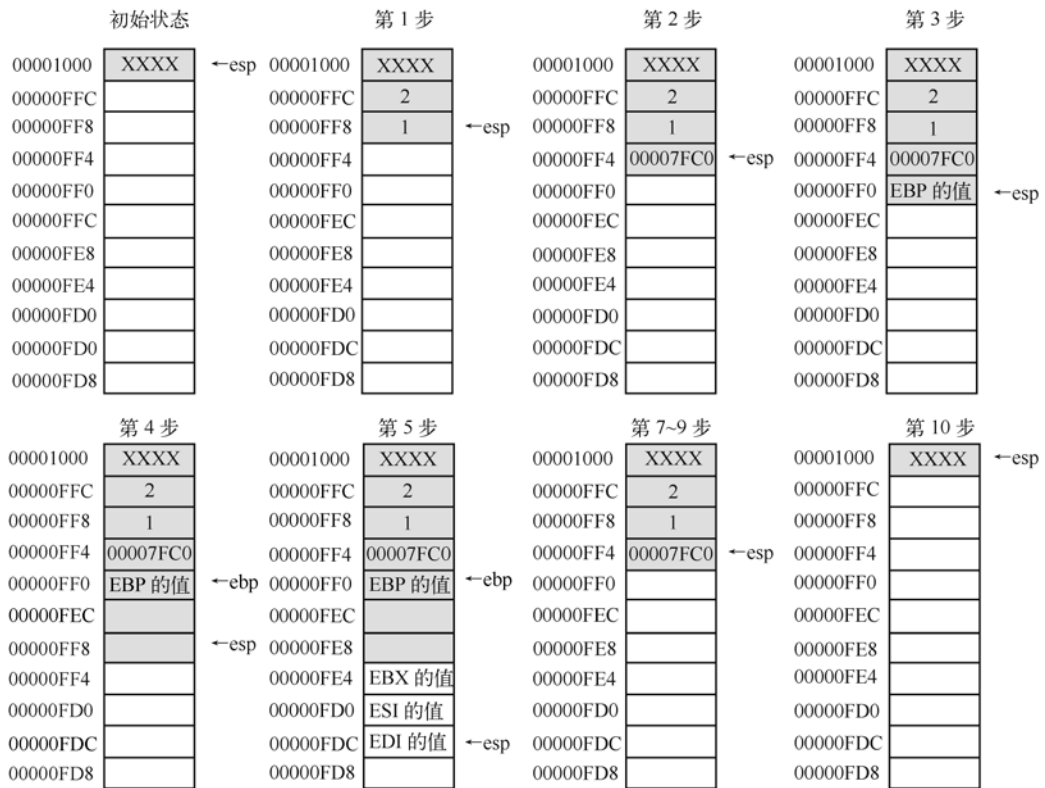


图 8-9 程序 8-1 的栈变化过程

(1) 参数压入系统栈。注意，为了便于被调函数访问实参，参数压栈的顺序恰好与参数书写顺序相反。

(2) 调用_ADD 过程。首先，CPU 会自动将调用点（即 call 指令）之后首条指令的地址压入系统栈。这里，假设第 35 行指令的地址为 00007FC0h。然后，CPU 转入_ADD 过程执行。

(3) 保护 ebp。由于 ebp 将在程序中用于变址寻址，故必须将 ebp 压入系统栈。然后，将 esp 的值置入 ebp 中。注意，由于 esp 是栈顶的指针，压、出栈将时刻影响 esp 的值，因此，不便于后续局部变量的寻址处理。

(4) 申请存储空间。_ADD 过程有两个 4B 的变量，故需要分配 8B 的存储空间。注

意，由于系统栈是向下扩展的，因此，`esp` 的值减 8 即表示向下预留 8B 的存储空间。

(5) 保护 `ebx`、`esi`、`edi` 的值。一般编译器都考虑将这三个寄存器保护，以便后续处理。

(6) 局部变量（包括参数）只需以 “[`ebp`+该变量的偏移]” 形式引用即可。

(7) 恢复 `ebx`、`esi`、`edi` 的值。

(8) 恢复 `esp` 的值。

(9) 恢复 `ebp` 的值。

(10) 清理参数并返回。在 i386 系统中，提供了 “`ret X`” 指令来清理参数，这是一种非常有效的机制。

在图 8-9 中，有一个事实是显而易见的，那就是函数调用前后的系统栈是平衡的，因此，本次调用可以视为是安全的，读者千万不可忽视这一要点。在应用系统栈时，平衡可能比其他任何因素都显得重要。一些商用编译器甚至不惜通过在目标程序中插入检测代码，以保证栈的平衡。除了栈平衡之外，编译器需要关心的另一个问题就是局部变量、参数等数据对象的寻址，这是被调函数正常运行的关键所在。仔细分析图 8-9，读者不难发现以下两个结论：

(1) 实参存储区就是以 `ebp+4` 为基准地址向上扩展的存储区域。

(2) 局部变量存储区就是以 `ebp-4` 为基准地址向下扩展的存储区域。

有了以上两个结论以后，寻址问题将变得非常容易。例如，可以通过 [`ebp+8`]（即 [`ebp+4+4`]）访问参数 `i`。当然，在这个过程中，访问的越界控制完全是由编译器处理的，编译器必须准确计算每个对象的偏移，以及预留的空间等。这是一项非常精巧的工程，稍有不慎，就将导致编译结果不正确。值得注意的是，关于系统栈的布局形态并不唯一，因此，以上两个结论未必适用于任何编译器。不过，笔者可以肯定地说，栈分配的基本思路必定如此，差异仅可能存在于一些实现的细节中。例如，需要保护的寄存器个数可能有差异。甚至有些优化编译器不必借助于 `ebp` 寻址，而是直接使用 `esp`。那样的话，由于整个函数范围内 `esp` 是可能发生变化的（嵌套调用其他函数），因此，计算偏移的问题将变得异常复杂。有兴趣的读者可以尝试。而且，可以肯定的是，这种做法是绝对可行的。

最后简单解释一下关于存储区布局的问题。实参存储区的布局是根据实参的压栈顺序而定的。一般而言，为了便于函数顺序访问实参，在传递实参时，将实参逆序压栈是通用的解决方案，这是不存在任何异议的。而变量存储区的布局则完全是由编译器决定的，编译器根据函数局部变量的个数及所占空间的大小分配相应的存储区。至于局部变量到底是如何映射到存储区中的问题，已经在 8.1.2 节中作了详细讨论，不再赘述。

8.2.3 深入理解栈式存储分配

在编译器设计中，栈式存储分配是一种应用极广的存储分配策略。本小节将结合几个比较特殊的语言机制讨论栈式存储分配的应用，旨在使读者对这个极其重要的机制有更深刻的理解。当然，这里仅从原理的角度加以剖析，不会过多关注其中的实现细节。

1. 可变参数

熟悉 C 语言的读者对于可变参数应该并不陌生。虽然在实际编程中，刻意应用可变参数的场合并不算太多，但是，接触可变参数的机会却并不少。最常见的例子可能就是 `printf`、`scanf` 了，其声明形式如下：



【声明 8-2】

```
int printf(const char* _Format,...);  
int scanf(const char* _Format,...);
```

从传参的角度来说，可变参数的传递与普通参数的传递并不存在明显的差异，都是将实参逆序压栈实现的。而可变参数的关键问题就在于函数本身是无法得知实参存储区大小的，这个信息只能由调用点计算确定。因为各调用点传递的参数个数可能是不同的，故所需的实参存储区的空间数量也是不同的。在这种情况下，实参存储区的清理工作就只能由调用点完成了，如图 8-10 所示。

```
printf("%d,%d,%d",a,b,c);  
00401000 mov     eax,dword ptr [c (40338Ch)]  
00401005 mov     ecx,dword ptr [b (403388h)]  
0040100B mov     edx,dword ptr [a (403390h)]  
00401011 push   eax  
00401012 push   ecx  
00401013 push   edx  
00401014 push   offset string "%d,%d,%d" (4020F4h)  
00401019 call   dword ptr [__imp_printf (4020A0h)]  
0040101F add     esp,10h
```

图 8-10 printf 调用点的反汇编程序

在处理可变参数时，C 编译器只会遵守 `_cdecl` 调用约定，即使用户显式地将调用约定设置为 `_stdcall` 也是无效的。原因很简单，`_stdcall` 规定清栈工作是由被调函数完成的，因此，这是不满足可变参数的基本条件的。

值得注意的是，在实现可变参数时，由调用点完成清栈工作是最常见且便捷的处理，但并不是唯一的方案。实际上，试图由函数本身完成清栈工作同样是可行的，其基本思想就是调用点将计算得到的实参存储区的实际大小也以参数的形式传递给被调函数，而被调函数则根据该参数的值在返回前完成清栈工作。当然，这个过程是需要双方协调的，目前，并没有标准的调用协议支持。

2. 变长数据的分配

在一些程序设计语言中，变长数据的应用是非常广泛的，例如，字符串、变长数组等。当然，相对于定长数据而言，变长数据的灵活程度是用户更愿意接受的主要原因。不过，变长数据对编译器的设计提出了挑战，原因不难理解。对于栈式存储分配机制而言，试图在一个函数体内改变 AR 的长度或者布局是不可能实现的。因为一个函数的 AR 完全是根据局部变量及参数的实际情况计算得到的，这个计算过程是由编译器完成的。因此，也就不能接受在运行过程中动态改变局部变量的实际长度。

实际上，解决变长数据的问题并不复杂。在讲述逻辑地址空间时，笔者引入了堆的概念。在 i386 中，堆的管理通常是由操作系统监控的，而操作系统仅仅以系统调用的形式暴露一些管理接口供用户使用。在早期，C、Pascal 等编译器并不会将用户数据分配到堆空间中，而是直接为用户提供了 `malloc`、`free` 之类的堆管理接口，以使用户动态申请与释放存储

空间。不过，在近二十多年中，变长数据类型已经逐渐成为了程序设计语言不可缺少的组成部分。因此，编译器设计者也有必要关注这方面的话题。

图 8-11 描述了 s0、s1 两个字符串变量的内部存储形式。在系统栈中，编译器并不会根据字符串的实际长度分配存储空间，而只是为其分配一个指针空间，该空间中存储的就只是一个位于堆中的数据块的地址，而该数据块中存储的才是字符串的实际值。访问字符串值时，通过间接寻址访问即可。关于堆的话题，笔者作三点说明：

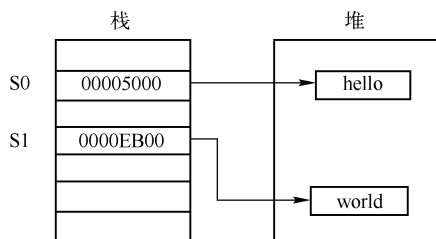


图 8-11 变长数据的存储示意图

(1) 数据块与堆。事实上，一次系统调用申请得到一块存储区即为一个数据块，可以视作当前进程从操作系统得到的一块存储资源。除了显式释放或进程退出之外，数据块是不会自动被操作系统回收的。至于数据块的分配策略完全是由操作系统决定的，编译器、用户进程都是无法掌控的。在不引起冲突的情况下，通常，操作系统是允许用户进程动态扩展数据块容量的。当然，这种操作还是比较危险的，对环境的耦合程度太高，因此不推荐使用。

(2) 越界访问可能会引起异常中断。与系统栈的访问不同，越界访问数据块可能是致命的。主要的监管工作是由操作系统完成的。当然，不同的操作系统对此的理解并不一致，管理的力度也不尽相同，但仍然不建议读者尝试。

(3) 堆是线性结构。图 8-11 似乎并不能表现堆的实际形态，这是因为笔者更想突出堆的黑箱特性。必须澄清的是堆的内部结构仍然是线性的，只不过其分配策略对于用户是完全透明的。另外，值得注意的是，这里的堆与数据结构中的堆是完全不同的概念，仅仅名字相同而已。

3. 悬空引用

在 C、Pascal 等支持指针的程序设计语言中，悬空引用可能是非常危险的。虽然悬空引用可以被视作一种语义错误，但几乎没有一个编译器可以在编译阶段分析出此类错误。更可怕的是，由于存储块释放后仍可能被重新分配，因此，悬空引用的错误可能是致命的且不可预知的。例如，如声明 8-3 所示：

【声明 8-3】

```
int * aa()
{
    int i=34;
    return &i;
}
main()
```




```
{
    int *p=aa();
}
```

aa 函数返回局部变量 i 的地址。不过，从 aa 函数返回到 main 函数时，由于 aa 函数的 AR 被释放，因此，p 指针的引用就是悬空引用。

但悬空引用也不是一无是处的，在处理一些复杂结构的返回值时，经常会借助指针，避免冗余的数据块复制。当然，用户必须保证这种“悬空引用”是绝对安全的。如声明 8-4 所示：

【声明 8-4】

```
int * aa()
{
    int i[10]={1,2,3,4,5,6,7,8,9,10};
    return &i;
}
main()
{
    int *p=aa();
    int j;
    .....
    j=*(p+2);
}
```

在编译器设计中，使用这种引用机制处理复杂结构的返回值是非常有效且方便的。

8.3 存储分配的实现

运行时的存储分配是独立存在于 IR 优化与目标代码生成之间的。通常，关于存储空间方面的 IR 优化应该是在存储分配之前基本完成的。换句话说，本阶段涉及的有效变量、常量的集合是相对稳定的，至少在 IR 层次上是不再进行任何相关优化的。否则，存储分配将不得不面临多次迭代的命运。事实上，这是完全没有必要的。

通过前面的学习，读者应该已经理解栈式分配的基本思想了。编译器可以通过计算符号相对过程局部数据块首的偏移，最终实现寻址。

程序 8-2 CodeGen.cpp

```
1 void CCodeGen::MemoryAlloc()
2 {
3     int iStackOffset=8;
4     int iTmp=0;
5     for(int i=0;i<SymbolTbl.TypeInfoTbl.size();i++)
6         SymbolTbl.CalcTypeSize(i);
7     RecOpAsm(".data");
8     RecContentAsm(";ConstSymbolList.");
9     for(int i=0;i<SymbolTbl.ConstInfoTbl.size();i++)
10    {
```

```
11     if (!SymbolTbl.ConstInfoTbl[i].m_bUsed)
12         continue;
13     switch (SymbolTbl.ConstInfoTbl[i].m_ConstType)
14     {
15     case ConstType::SET:
16         RecOpAsm("__Const"+itos(i)+" db "+SetValue(SymbolTbl.ConstInfoTbl[i].m_szName));
17         break;
18     case ConstType::BOOLEAN:
19         if (SymbolTbl.ConstInfoTbl[i].m_szName=="TRUE")
20             RecOpAsm("__Const"+itos(i)+"=1");
21         else
22             RecOpAsm("__Const"+itos(i)+"=0");
23         break;
24     case ConstType::PTR:
25         RecOpAsm("__Const"+itos(i)+"=0");
26         break;
27     case ConstType::STRING:
28         {
29             string szTmp=GetConstStr(SymbolTbl.ConstInfoTbl[i].m_szName.substr
30                                     (1,SymbolTbl.ConstInfoTbl[i].m_szName.length()-2));
31             if (szTmp=="\\")
32                 RecOpAsm("__Text"+itos(i)+" db 0");
33             else
34                 RecOpAsm("__Text"+itos(i)+" db "+szTmp+",0");
35             RecOpAsm("__Const"+itos(i)+" dword 0");
36             break;
37         }
38     case ConstType::ENUM:
39     case ConstType::INTEGER:
40         RecOpAsm("__Const"+itos(i)+"="+itos(SymbolTbl.ConstInfoTbl[i].m_iVal));
41         break;
42     case ConstType::EREAL:
43     case ConstType::REAL:
44         RecOpAsm("__Const"+itos(i)+" dword "+rtos(SymbolTbl.ConstInfoTbl[i].m_fVal));
45         break;
46     default:
47         RecOpAsm("__Const"+itos(i)+"="+SymbolTbl.ConstInfoTbl[i].m_szName);
48     }
49 }
50 RecContentAsm(";VarSymbolList");
51 RecOpAsm("__cvt DWORD 1 DUP(0)");
52 for(int i=0;i<SymbolTbl.ProcInfoTbl.size();i++)
53 {
54     if (SymbolTbl.ProcInfoTbl[i].m_bUsed==false)
55         continue;
56     iTmp=iStackOffset;
```



```
57     if (SymbolTbl.ProcInfoTbl.at(i).m_szName=="_noname")
58         continue;
59     for(int j=0;j<SymbolTbl.VarInfoTbl.size();j++)
60     {
61         if (SymbolTbl.VarInfoTbl[j].m_iProcIndex==i
62             && SymbolTbl.VarInfoTbl[j].m_eRank==VarInfo::PARA)
63         {
64             SymbolTbl.VarInfoTbl[j].m_iMemoryAlloc=iTmp;
65             int iSize=SymbolTbl.TypeInfoTbl[SymbolTbl.VarInfoTbl[j]
66                 .m_iTypeLink].m_iSize;
67             if (iSize % 4!=0)
68                 iSize=(4-iSize % 4)+iSize;
69             if (i!=0)
70                 RecOpAsm("_"+SymbolTbl.ProcInfoTbl.at(i).m_szName+"$"+
71                     SymbolTbl.VarInfoTbl[j].m_szName+"="+itos(iTmp));
72             else
73                 RecOpAsm("_"+SymbolTbl.ProcInfoTbl.at(i).m_szName+"$"+SymbolTbl
74                     .VarInfoTbl[j].m_szName+" DWORD "+itos(iSize/4)+" DUP(0)");
75             iTmp=iTmp+iSize;
76         }
77     }
78 }
79 for(map<int,VarInfo>::iterator it=SymbolTbl.VarInfoTbl.begin();
80     it!=SymbolTbl.VarInfoTbl.end();it++)
81 {
82     if (SymbolTbl.ProcInfoTbl[it->second.m_iProcIndex].m_bUsed==false)
83         continue;
84     if (it->second.m_iMemoryAlloc!=-1 && it->second.m_iProcIndex!=0
85         && it->second.m_eRank!=VarInfo::PARA)
86     {
87         iTmp=-SymbolTbl.ProcInfoTbl.at(it->second.m_iProcIndex).m_ValSize;
88         RecOpAsm("_"+SymbolTbl.ProcInfoTbl.at(it->second.m_iProcIndex).m_szName+"$"+
89             it->second.m_szName+"="+itos(iTmp+it->second.m_iMemoryAlloc));
90     }
91 }
92 for(map<int,VarInfo>::iterator it=SymbolTbl.VarInfoTbl.begin();
93     it!=SymbolTbl.VarInfoTbl.end();it++)
94 {
95     if (SymbolTbl.ProcInfoTbl[it->second.m_iProcIndex].m_bUsed==false)
96         continue;
97     if (((it->second.m_iMemoryAlloc!=-1 && it->second.m_iProcIndex==0))
98         {
99         iTmp=SymbolTbl.TypeInfoTbl[it->second.m_iTypeLink].m_iSize;
100        string szTmp="";
101        if (iTmp==1)        szTmp="BYTE";
102        if (iTmp==2)        szTmp="WORD";
```

```
103         if (iTmp>=3)         szTmp="DWORD";
104         if (iTmp % 4!=0)     iTmp=(4-iTmp % 4)+iTmp;
105         RecOpAsm("_"+SymbolTbl.ProcInfoTbl.at(it->second.m_iProcIndex)
106             .m_szName+"$"+it->second.m_szName+" "+szTmp+" "+itos(iTmp/4)+" DUP(0));
107     }
108 }
109 }
```

第 5、6 行：调用 CalcTypeSize 函数，重新计算各种类型所需的存储空间大小。在存储分配之前，这项工作是非常有意义的。在处理记录类型时，必须考虑字段布局的问题。因为不同的字段布局对最终的空间大小是有一定影响的。

第 7 行：生成数据段声明的汇编代码。

第 9~49 行：遍历常量信息表，生成常量声明的汇编代码。

第 11~12 行：略过未引用的常量符号。

第 13~48 行：根据常量的类型，生成常量声明代码。

第 15~17 行：生成集合类型常量的声明代码。应该注意两点：

(1) 常量对应的汇编符号是以 “_ConstXXX” 形式命名的，其中，XXX 即为常量符号的位序号。

(2) 在标准 Pascal 中，集合常量可以用连续 32 个字节的值表示。这里，SetValue 函数就是将集合常量的字面值转换为由 32 个独立字节形式组成的字符串。

第 18~23 行：生成布尔类型常量的声明代码。在 Neo Pascal 中，布尔类型常量只有两种取值，即 TRUE 和 FALSE。根据 Pascal 的标准，TRUE 用数值“1”表示，而 FALSE 则用“0”表示。

第 24~26 行：生成指针类型常量的声明代码。在 Neo Pascal 中，指针类型常量只有 NIL，其取值为“0”。

第 27~37 行：生成字符串类型常量的声明代码。应该注意以下三点：

(1) 字符串结束符即数值“0”。

(2) 为了统一字符串的寻址处理，编译器为字符串常量分配了一个指针存储区。该指针指向的存储区才是实际的字符串值。这样，无论是字符串常量还是变量寻址，都必须进行一次间接寻址。虽然这种处理不得不以牺牲一些存储空间为代价，但显著减轻了程序设计的负担。

(3) 处理字符串中的换行符。在 MASM 宏汇编声明中，不含换行符的字符串是允许出现在数据段声明中的，当然，并不支持 C 语言的转义字符。因此，有必要对符号表中的字符串常量中的换行符作特殊处理。

以字符串“aa\nbb”为例，其对应的汇编声明如下：

```
_Text1 db "aa", 10, "bb", 0
_Const1 dword <_Text1 的地址>
```

在 Neo Pascal 中，_Const1 单元存储的值是由初始化程序统一设置的。

第 38~41 行：枚举类型、整型常量只需直接获得其实际取值即可。

第 42~45 行：实型常量只需要直接获得其实际取值即可。不过，值得注意的是，在实



践中，很多汇编器是不支持实型的，经常需要编译器完成实型值的内码计算。其中 IEEE 754 是最常见的实数内码标准。

第 52~78 行：遍历过程信息表，生成形参的声明代码。分配算法的关键就是计算变量相对于 AR 的偏移，AR 是以过程为单位讨论的。

第 54、55 行：未被调用的过程不参与存储分配。

第 56 行：初始化偏移基值。根据系统栈的布局，形参的初始偏移基值就是 8。

第 57、58 行：匿名过程信息不参与存储分配。

第 59~78 行：遍历变量信息表，生成当前过程所属形参的声明。

第 61、62 行：判断当前变量是否为本过程所属的形参。

第 64 行：为变量符号分配存储区。这里，只需在符号信息中记录偏移值即可。

第 65 行：获得当前变量符号所需存储空间的大小。以计算下一个符号的偏移。

第 67、68 行：处理整字对齐。这里，只考虑 4 字节的整字对齐情况。

第 69~74 行：生成符号的汇编声明。当 i 等于 0 时，即表示该符号为主程序的形参，应该作为全局变量进行静态分配。当然，目前笔者并没有为主程序提供形参机制。

第 75 行：更新偏移计数变量 $iTmp$ 。

第 79~91 行：遍历变量信息表，生成不可优化的变量符号的声明。这里，值得注意的是，并非所有的有效符号都需要分配存储空间。实际上，在分配存储空间之前，编译器还会进行一次存储空间方面的优化，以实现空间的共享。如果变量符号的 $m_iMemoryAlloc$ 的值为 -1，则表示该符号不需要分配空间。

第 87~89 行：计算各过程的局部变量符号的偏移值，并生成汇编声明。通常，存储寻址都是向上扩展的，而系统栈的扩展却恰好相反，为了便于处理，Neo Pascal 以局部变量存储区的最低地址为基准逐一向上传递。这里， $m_iMemoryAlloc$ 的值是变量相对于 0 的偏移，而 $m_ValSize$ 是整个过程的局部变量存储区的大小。这两个值都是由存储优化算法计算得到的。

第 92~108 行：生成全局变量符号的静态声明。前面已经讲述了全局变量是静态分配存储空间的。因此，这里只需根据变量所需空间大小分配即可。

关于存储分配的实现，就暂且讨论至此。笔者还要说明一点，在 Neo Pascal 中，存储分配算法与存储优化是密不可分的，严格地说，存储优化是必不可少的。然而，这个观点并不适用于一般化的情形，在很多编译器中，存储优化完全是可选的。不过，存储优化本身并不影响调试，因此，可选或必选并不太重要。

8.4 存储优化

8.4.1 存储优化基础

第 7 章中所涉及的优化不论是简化计算，还是冗余删除，其目的都是试图通过优化算法提高代码执行效率。本小节将引领读者从另一个视角来思考有关优化的问题，那就是存储优化。所谓“存储优化”就是通过优化算法以改善目标程序在执行过程中对存储资源的耗费。在优化技术中，读者可能有机会接触到一个概念，即存储层次优化。注意，存储层次优化与这里所讨论的存储优化是两个完全不同的概念。存储层次优化主要是关注如何利用存储器的

层次（例如，常见的三级存储结构）来提高代码执行效率的一种优化算法。而存储优化并不关注代码执行的效率，换句话说，它本身不会对代码执行效率产生任何影响。下面，通过一个简单的实例来看看存储优化的基本思想。

如图 8-11 所示，读者不必关注 `aa` 函数是怎样实现的，假设 `aa` 是一个非空函数即可。在这个实例中，`i`、`j`、`k` 的作用域都是整个 `main` 函数体。显然，根据存储分配的原则，编译器必须分配三个存储单元用于分别存储 `i`、`j`、`k` 的值。不过，从程序逻辑本身的特点来说，不难发现，即使将 `i`、`j`、`k` 都分配在同一个存储单元中，也不会给程序执行结果带来任何影响。这种依赖于程序逻辑实现的存储共享策略就是存储分配的基本思想。

```
main()
{
    int i,j,k;
    scanf("%d",&k);
    if (k>0)
        aa(i);
    else
        aa(j);
}
```

图 8-11 存储优化引例

存储优化的成果就是提出一个存储分配的策略或者方案，其主要描述的就是哪些变量可以共享同一个存储单元。而存储分配算法就可以根据这份方案来完成实际的存储分配工作。

通用的存储优化算法是有一定难度的，由于篇幅所限，笔者并不打算深入讨论。这里，只针对编译器产生的临时变量进行分配优化，这是相对容易且有效的。在 IR 生成阶段，编译器始终没有关注如何避免临时变量冗余的问题。在很多情况下，即使是一个非常简单的源程序，其对应的目标程序所涉及的临时变量数目也是非常可观的，甚至可以用“庞大”二字来形容，这是用户不能接受的。笔者可以举一个简单的例子。

例 8-2 临时变量存储分配优化实例，见表 8-2。

表 8-2 临时变量存储分配优化

(a) 输入源程序	(b) 优化前的 IR	(c) 优化后的 IR
<code>var a,b,c:integer;</code>	0: (ADD_4 ,A,B,_T0)	0: (ADD_4 ,A,B,_T0)
<code>begin</code>	1: (ADD_4 ,_T0,C,_T1)	1: (ADD_4 ,_T0,C,_T1)
<code>a=a+b+c;</code>	2: (ASSIGN_4 ,_T1,null,A)	2: (ADD_4 ,_T1,B,_T2)
<code>b=a+b+c;</code>	3: (ADD_4 ,A,B,_T2)	3: (ADD_4 ,_T2,C,_T3)
<code>c=a+b+c;</code>	4: (ADD_4 ,_T2,C,_T3)	4: (ADD_4 ,_T1,_T3,_T4)
<code>result:=c;</code>	5: (ASSIGN_4 ,_T3,null,B)	5: (ADD_4 ,_T4,C,_T5)
<code>end;</code>	6: (ADD_4 ,A,B,_T4)	6: (ASSIGN_4 ,_T5,null,RESULT)
	7: (ADD_4 ,_T4,C,_T5)	
	8: (ASSIGN_4 ,_T5,null,C)	
	9: (ASSIGN_4 ,C,null,RESULT)	

这样一个简单的程序，也需要耗用 6 个临时变量，如果编译一个实际的应用程序，消耗数千甚至更多的临时变量并不罕见。因此，寻找一种较优的分配策略是极其重要的。注意，本例中(c)列只是在(b)列的基础上做了常量传播、复写传播等优化后的结果。虽然代码的规模有所减小，但是临时变量的个数并没有任何变化。存储分配的优化并不会改善代码本身的形式，只是提出了一份较优的存储分配的策略。在这个例子中，优化算法提出的临时变量分配方案即为：`_T0`、`_T2`、`_T3`、`_T4`、`_T5` 共享一个存储单元，`_T1` 独享一个存储单元。如果读者根据这个分配方案仔细推敲(c)列的 IR，可以发现，程序依然是正确且安全的。然而，临时变量的使用数量却大大减少了，这是令人欣慰的。那么，为什么 `_T1` 不能与其他临时变量共享同一个存储单元呢？这个问题非常简单，如果共享同一个存储单元，那么，第 2 行、



第 3 行的定值将影响 `_T1` 本身的定值（即由第 1 行 IR 所置入的值）。然而，由于第 4 行中又存在 `_T1` 的引用点，此时，`_T1` 的值可能已经不是原来由第 1 行 IR 所确定的，而是由第 3 行所确定的，这显然与原始的语义不符。

至此，笔者通过一个实例简单介绍了存储优化的概念及其意义。存储优化的前提就是保证访问的安全性，而这也正是其算法的关键所在。实际上，本书讨论的存储优化模型与通用的存储优化算法的差异就在于冲突检测机制。讨论临时变量的冲突检测远比普通的用户变量容易得多。通常，冲突可以利用图或者网结构描述，如常见检测算法就是基于图着色思想实现的。在经典编译技术中，图着色更多应用于寄存器分配算法中，主要用于描述寄存器的使用冲突。

8.4.2 存储优化的实现

前面，读者已经了解了存储分配优化的基本概念，本小节将从实现的角度来详细分析相关细节。无论存储优化算法如何激进，变量访问不冲突是不可逾越的底线，这是编译安全性的保证。下面，笔者只讨论一种特殊的情况，即临时变量的存储优化。

临时变量是由编译器自动产生的，因此，编译器对其的控制力度要远胜于普通变量。针对临时变量的特点，讨论存储优化就简单得多了。在 Neo Pascal 中，为了便于算法实现，算法讨论的临时变量必须满足如下三个条件：

- (1) 在整个程序范围内，只能存在一次定值和一次引用。
- (2) 一次定值和一次引用是在同一个基本块内的。
- (3) 定值点位于引用点之前。

也就是说，只有满足这三个条件的临时变量才可能进行存储优化。当然，这只是优化之前的一项准备工作而已。那么，这三个条件是否可以被大多数临时变量接受呢？虽然三个条件看似都比较苛刻，但对于许多临时变量来说，却是可以接受的。事实上，现行的 IR 生成机制所创建的临时变量是非常容易达到这一要求的。在例 8-2 中，不难发现，除了 `_T1` 之外，其他的临时变量都是本优化算法的工作对象。实际上，读者可以尝试更多的例程，能满足这三个条件的临时变量应该超过 80%，这样的结果是令人兴奋的。因此，基于这三个条件讨论存储分配优化是具有现实意义的。注意，本小节后续篇幅中不作特殊说明的“变量”都是指满足上述条件的临时变量。下面，就来讨论临时变量存储优化的实现细节。

首先，就来谈谈共享存储区的问题。Neo Pascal 并没有设置太多的共享存储区，在通常情况下，每个过程只拥有一个共享存储区。而共享存储区的大小是根据实际使用该共享区的变量的类型而定的，即为各个变量实际容量的最大值。事实上，针对临时变量存储优化的问题设置一个共享存储区是可以接受的。虽然未必能得到最优的分配方案，但应该是工程应用领域可以接受的次优解。而设置一个共享存储区的主要优点就是便于算法实现。

其次，就要收集本过程中可以共享存储区的临时变量。由于本小节涉及的临时变量的定值、引用必定位于同一基本块内，因此，位于不同基本块内且满足条件的临时变量是可以共享存储区的。这里，编译器只需要确定在一个基本块内可共享存储区的临时变量集合即可。假设临时变量集合 `s` 的所有元素的定值、引用点都位于同一个基本块内，且它们可以共享同一存储区。如果试图将变量 `v` 加入集合 `s`，那么，`v` 必须满足两个条件：

- (1) 变量 `v` 的定值、引用点也必须位于该基本块内。

(2) 集合 s 中的任意变量 i 都与 v 不冲突。所谓“冲突”就是指两个变量从定值点到引用点之间的 IR 所构成的集合的交集为空。

最后，选择临时变量的顺序也是值得关注的。这主要与 Neo Pascal 的实现有关，因为 Neo Pascal 只考虑为每个过程分配一个临时变量的共享存储区。在这种情况下，优化的效果与选择临时变量的顺序是密切相关的。例如，已知 5 个临时变量的定值、引用点如下：

```
_T1(def:1,use:5)
_T2(def:2,use:3)
_T3(def:1,use:1)
_T4(def:4,use:6)
_T5(def:6,use:8)
```

假设选择变量的顺序为 ($_T1$ 、 $_T2$ 、 $_T3$ 、 $_T4$ 、 $_T5$)，那么，分析过程如下：

- 1) 清空集合 s 。
- 2) $_T1$ 与集合 s 中的变量不冲突， $s=\{_T1\}$ 。
- 3) $_T2$ 与集合 s 中的 $_T1$ 冲突，略过 $_T2$ 。
- 4) 同理， $_T3$ 、 $_T4$ 与 $_T1$ 也是冲突的，同样略过。
- 5) $_T5$ 与集合 s 中的 $_T1$ 不冲突， $s=\{_T1, _T5\}$ 。

就本例而言，优化算法确定 $\{_T1, _T5\}$ 是可以共享存储区的，而其他临时变量则必须独立分配存储区。不过，如果选择变量的顺序为 ($_T2$ 、 $_T3$ 、 $_T4$ 、 $_T5$ 、 $_T1$) 时，那么，最终得到的 s 就是 $\{_T2, _T3, _T4, _T5\}$ 。由于临时变量的类型都是基本类型，不同变量所需的存储空间差异并不明显，因此，后者的方案是较优于前者的。换句话说，两个方案的差别主要就在于集合 s 中的元素个数。

这里，应用了一个特殊的排序方式，即按每个临时变量的 use-def 得到的差值由小至大排序。实际上，其目的就是优先选择引用点与定值点较接近的临时变量，只有这样才能尽可能使更多变量加入共享集合中。

下面，就来详细看看 Neo Pascal 相关源代码的实现。

程序 8-3 MemShare.cpp

```
1 void CMemShare::TmpMemShare()
2 {
3     vector<TmpInfo> TmpVar;
4     map<int,vector<LiveArea>> Conflict;
5     CDataFlowAnalysis::DataFlowAnalysis();
6     for (int i=0;i<SymbolTbl.ProcInfoTbl.size();i++)
7     {
8         map<int,int> VarMap=GetVar2IdMap(i,false);
9         SymbolTbl.ProcInfoTbl.at(i).m_TmpMemShare.clear();
10        TmpVar.clear();
11        Conflict.clear();
12        for(map<int,int>::iterator it=VarMap.begin();it!=VarMap.end();it++)
13        {
14            if(!SymbolTbl.IsTmpVar(it->first))
15                continue;
```




```
16         if (VarDef[itos(i)+"$"+itos(it->first)].size()==1 &&
17             VarAllUse[itos(i)+"$"+itos(it->first)].size()==1)
18         {
19             int iDef=VarDef[itos(i)+"$"+itos(it->first)].at(0);
20             int iUse=VarAllUse[itos(i)+"$"+itos(it->first)].at(0);
21             int iDefBlock=GetBlock(i,iDef);
22             int iUseBlock=GetBlock(i,iUse);
23             if (iDefBlock==iUseBlock && iDef<=iUse)
24                 TmpVar.push_back(TmpInfo(it->first,iDef,iUse,iDefBlock));
25         }
26     }
27     sort(TmpVar.begin(),TmpVar.end(),TmpInfoCmp);
28     for(vector<TmpInfo>::iterator it=TmpVar.begin();it!=TmpVar.end();it++)
29     {
30         map<int,vector<LiveArea>>::iterator ConflictIt=Conflict.find(it->m_iBlock);
31         if (ConflictIt!=Conflict.end())
32         {
33             vector<LiveArea>::iterator VecIt=ConflictIt->second.begin();
34             for(;VecIt!=ConflictIt->second.end();VecIt++)
35             {
36                 if (it->m_LiveArea.m_iDef<VecIt->m_iUse &&
37                     it->m_LiveArea.m_iDef>VecIt->m_iDef)
38                     break;
39                 if (it->m_LiveArea.m_iUse<=VecIt->m_iUse &&
40                     it->m_LiveArea.m_iUse>VecIt->m_iDef)
41                     break;
42                 if (it->m_LiveArea.m_iUse>=VecIt->m_iUse &&
43                     it->m_LiveArea.m_iDef<=VecIt->m_iDef)
44                     break;
45             }
46             if (VecIt==ConflictIt->second.end())
47             {
48                 ConflictIt->second.push_back(LiveArea(it->m_LiveArea.m_iDef,
49                                                         it->m_LiveArea.m_iUse));
50                 SymbolTbl.ProcInfoTbl.at(i).m_TmpMemShare.insert(it->m_iLink);
51             }
52         }
53     }
54     else
55     {
56         vector<LiveArea> TmpVec;
57         SymbolTbl.ProcInfoTbl.at(i).m_TmpMemShare.insert(it->m_iLink);
58         TmpVec.push_back(LiveArea(it->m_LiveArea.m_iDef,it->m_LiveArea.m_iUse));
59         Conflict.insert(pair<int,vector<LiveArea>>(it->m_iBlock,TmpVec));
60     }
61 }
62 }
```

第 3 行：声明临时变量集合，即该集合内的临时变量是共享同一存储区的。其中，TmpInfo 结构用于描述临时变量的相关信息，声明如下：

【声明 8-5】

```
struct TmpInfo
{
    int m_iLink;           //指向临时变量符号信息的指针
    int m_iBlock;         //指向临时变量所属基本块信息的指针
    LiveArea m_LiveArea; //活跃区域
};
```

m_iLink：指向临时变量符号信息的指针，即符号在变量信息表中的位序号。

m_iBlock：指向临时变量所属基本块信息的指针。

m_LiveArea：用于描述临时变量的活跃区域，即临时变量的定值点、引用点信息。由定值点到引用点构成的一个 IR 序列的区域即为活跃区域。如果两个临时变量的活跃区域存在交集，则表明这两个临时变量是冲突的，也就无法共享同一存储区。LiveArea 结构的声明形式如下：

【声明 8-6】

```
struct LiveArea
{
    int m_iDef;
    int m_iUse;
};
```

第 4 行：声明基本块冲突映射表。由于本算法并不是以基本块为主线遍历的，因此需要考虑应用 Conflict 映射表来描述整个过程的临时变量冲突信息。其中，Conflict 是以基本块的指针作为关键字，而 vector<LiveArea>则用来描述该基本块内已占用的活跃区域。当一个临时变量试图加入共享集合时，必须保证该变量的活跃区域与所属基本块的所有活跃区域都不存在冲突（即活跃区域不存在交集）。

第 5 行：调用函数完成数据流分析。

第 6~26 行：以过程为单位，分析共享存储区的临时变量集合。

第 8 行：获取当前过程的变量集合。

第 9~11 行：初始化相关数据结构。其中，m_TmpMemShare 就是用于描述该过程的共享集合。

第 12~26 行：遍历变量集合，将满足存储优化基本条件的临时变量加入 TmpVar 中。

第 14~15 行：判断是否为临时变量。

第 16~17 行：判断该临时变量是否只存在一次定值与一次引用。

第 19~22 行：根据临时变量的定值、引用信息，获取定值、引用点所属的基本块号。

第 23 行：判断定值、引用点是否位于同一基本块内，定值点的位置是否先于引用点。

第 24 行：满足上述条件的临时变量就是算法的工作对象，将其加入 TmpVar 集合。

第 27 行：对 TmpVar 集合进行排序，按活跃区域由小至大排列。

第 28~61 行：依次选择 TmpVar 集合中的临时变量，将其加入 Conflict 中，当然，前提



就是不发生活跃区域的冲突。

第 30 行：获得当前临时变量所属的基本块的活跃区域集合。

第 31 行：判断 Conflict 中是否存在该基本块的活跃区域集合。

第 34~45 行：遍历基本块的活跃区域集合，判断是否存在冲突。

第 46~51 行：如果当前临时变量的定值及引用点都不包含在该基本块所有活跃区域中，即表示不存在任何冲突，则可以将其加入该基本块的活跃区域集合。

第 50 行：将当前临时变量加入过程 m_TmpMemShare 集合中。

第 54~59 行：由于 Conflict 中不存在关于所属基本块的活跃区域集合的描述信息，则需要重新生成一个活跃区域集合，并将当前临时变量的活跃区域加入该集合。同时，将变量本身加入过程的 m_TmpMemShare 集合中。

至此，已经得到了一个关于过程的可共享存储区的临时变量集合，这个集合是后续存储优化分配的关键所在。编译器将根据这个集合确定哪些变量需要独立分配空间，而哪些变量则是共享同一片存储区域的，以及决策共享存储区的设置与相关的应用策略。下面，就来看看 Neo Pascal 是如何根据过程的 m_TmpMemShare 集合实现优化分配的。结合 8.3 节存储分配算法的详细分析，可能更有利于深入理解。

程序 8-4 MemoryAlloc.cpp

```
1 void CMemoryAlloc::MemoryAlloc()
2 {
3     for(int i=0;i<SymbolTbl.TypeInfoTbl.size();i++)
4     {
5         SymbolTbl.CalcTypeSize(i);
6     }
7     for(int i=0;i<SymbolTbl.VarInfoTbl.size();i++)
8     {
9         SymbolTbl.VarInfoTbl[i].m_iMemoryAlloc=-1;
10    }
11    for(vector<ProcInfo>::iterator it=SymbolTbl.ProcInfoTbl.begin();
12          it!=SymbolTbl.ProcInfoTbl.end();it++)
13    {
14        for(vector<IRCode>::iterator it1=it->m_Codes.begin();it1!=it->m_Codes.end();it1++)
15        {
16            if(it1->m_Op1.m_iType==OpInfo::CONST)
17                SymbolTbl.ConstInfoTbl[it1->m_Op1.m_iLink].m_bUsed|=true;
18            if(it1->m_Op2.m_iType==OpInfo::CONST)
19                SymbolTbl.ConstInfoTbl[it1->m_Op2.m_iLink].m_bUsed|=true;
20            if(it1->m_Rslt.m_iType==OpInfo::CONST)
21                SymbolTbl.ConstInfoTbl[it1->m_Rslt.m_iLink].m_bUsed|=true;
22        }
23    }
24    for(int i=0;i<SymbolTbl.ProcInfoTbl.size();i++)
25    {
26        if (SymbolTbl.ProcInfoTbl.at(i).m_eFlag!=ProcInfo::None)
27            continue;
```

```
28     vector<VarMem> Tmp;
29     int iMaxTmp=-1;
30     for(int j=0;j<SymbolTbl.VarInfoTbl.size();j++)
31     {
32         if (i!=SymbolTbl.VarInfoTbl[j].m_iProcIndex ||
33             SymbolTbl.VarInfoTbl[j].m_eRank==VarInfo::PARA )
34             continue;
35         if (i==0 && SymbolTbl.IsProcVar(SymbolTbl.VarInfoTbl[j].m_szName))
36             continue;
37         if (SymbolTbl.VarInfoTbl[j].m_eRank==VarInfo::VAR )
38         {
39             map<string,vector<int>>::iterator it=VarDef.find(itos(i)+"$"+itos(j));
40             map<string,vector<int>>::iterator it1=VarAllUse.find(itos(i)+"$"+itos(j));
41             if ((it==VarDef.end() || it->second.empty()) && (it1==VarAllUse.end() ||
42                 it1->second.empty()))
43                 continue;
44         }
45         Tmp.push_back(VarMem(j,SymbolTbl.TypeInfoTbl[SymbolTbl.VarInfoTbl[j]
46             .m_iTypeLink].m_iSize));
47         if (SymbolTbl.ProcInfoTbl.at(i).m_TmpMemShare.count(j)>0)
48         {
49             if (iMaxTmp==-1)
50             {
51                 iMaxTmp=Tmp.size()-1;
52                 Tmp.at(iMaxTmp).m_bMaxTmp=true;
53             }
54             else
55             {
56                 if (Tmp.at(Tmp.size()-1).m_iSize>Tmp.at(iMaxTmp).m_iSize)
57                 {
58                     Tmp.at(iMaxTmp).m_bMaxTmp=false;
59                     iMaxTmp=Tmp.size()-1;
60                     Tmp.at(iMaxTmp).m_bMaxTmp=true;
61                 }
62             }
63         }
64     }
65     sort(Tmp.begin(),Tmp.end(),VarMemCmp);
66     int iOffset=0;
67     int iMaxTmpOffset=-1;
68     int iMaxLink=iMaxTmp!=-1?Tmp.at(iMaxTmp).m_iLink:-1;
69     vector<int> TmpAlloc;
70     for(int j=0;j<Tmp.size();j++)
71     {
72         if (SymbolTbl.ProcInfoTbl.at(i).m_TmpMemShare.count(Tmp.at(j).m_iLink)==0 ||
73             Tmp.at(j).m_bMaxTmp)
```



```
74     {
75         int iEmpty;
76         switch(Tmp.at(j).m_iSize)
77         {
78             case 3:
79                 case 1:iEmpty=1;break;
80                 case 2:iEmpty=2;break;
81                 default:iEmpty=4;break;
82             }
83             if (iOffset%iEmpty!=0)
84                 iOffset=iOffset+4-iOffset%iEmpty;
85             if (Tmp.at(j).m_bMaxTmp)
86                 iMaxTmpOffset=iOffset;
87             SymbolTbl.VarInfoTbl[Tmp.at(j).m_iLink].m_iMemoryAlloc=iOffset;
88             iOffset+=Tmp.at(j).m_iSize;
89         }
90     else
91     {
92         TmpAlloc.push_back(j);
93     }
94 }
95 for(int j=0;j<TmpAlloc.size() && iMaxTmpOffset!=-1;j++)
96 {
97     SymbolTbl.VarInfoTbl[Tmp.at(TmpAlloc.at(j)).m_iLink].m_iMemoryAlloc=-1;
98 }
99 SymbolTbl.ProcInfoTbl.at(i).m_TmpLink=iMaxLink;
100 SymbolTbl.ProcInfoTbl.at(i).m_ValSize=iOffset%4==0?iOffset:iOffset+4-iOffset%4;
101 }
102 }
```

第 3~6 行：在存储分配之前，重新计算类型的实际占用空间大小。

第 7~10 行：初始化所有变量符号的 `m_iMemoryAlloc` 属性。在 8.3 节中，读者应该已经了解了 `m_iMemoryAlloc` 属性就是用于存储变量符号相对于本过程数据块的偏移。

第 11~23 行：遍历 IR 序列，根据操作数的类别，分析常量的引用情况。

第 24~101 行：遍历过程信息表，计算变量符号的 `m_iMemoryAlloc` 属性。

第 26、27 行：略过未被调用的过程。

第 30~64 行：遍历变量信息表，收集需要分配存储空间的变量符号。由于全局用户变量是静态分配的，因此，这里只考虑局部变量及全局临时变量的分配即可，并不涉及全局用户变量的分配。

第 32~34 行：略过非当前过程的变量符号。

第 35、36 行：略过全局用户变量符号。

第 37~44 行：略过没有定值点及引用点的变量符号。

第 45、46 行：将当前变量加入 `Tmp` 向量中。`Tmp` 向量是一个存储分配的辅助结构。

第 47~64 行：当前变量符号属于 `m_TmpMemShare` 集合，则表示该变量是可以分配在

共享存储区的。

第 49 行：`iMaxTmp` 变量用于标识 `m_TmpMemShare` 集合中所需存储空间最多的变量。`iMaxTmp` 的初始值就是-1。

第 50~62 行：如果当前变量是到目前为止占用空间最多的临时变量，则将 `iMaxTmp` 指向该变量信息，并设置该变量的 `m_bMaxTmp` 属性为 `true`。

第 65 行：按占用存储空间的大小重排 `Tmp` 向量中的变量信息。在讨论存储布局时，笔者已经详细解释了重排变量的意义了。

第 70~94 行：遍历 `Tmp` 向量，计算各变量符号的 `m_iMemoryAlloc` 属性。

第 72、73 行：判断当前变量是否需要分配存储空间。需要分配存储空间的变量必定满足如下两个条件之一：

(1) 不属于 `m_iMemoryAlloc` 集合，即为不可共享存储区的变量。

(2) 是占用空间最多的可共享存储区的临时变量。

第 76~84 行：整字对齐处理，即根据变量占用的空间大小，计算留白的字节。

第 85、86 行：如果当前变量是占用空间最多的可共享存储区的临时变量，则使用 `iMaxTmpOffset` 变量暂存当前的偏移。

第 87 行：设置变量符号的 `m_iMemoryAlloc` 属性，即相对于过程数据块的偏移值。

第 92 行：处理可共享存储区的临时变量（不是占用空间最多的临时变量），将变量符号的位序号加入 `TmpAlloc` 向量中。

第 95~98 行：遍历 `TmpAlloc` 向量，回填共享存储区的变量的 `m_iMemoryAlloc` 属性。

第 99 行：设置当前过程的 `m_TmpLink` 属性，该属性记录的是占用空间最多的可共享存储区的临时变量的位序。

第 100 行：设置当前过程的 `m_ValSize` 属性，该属性记录的是当前过程数据块占用空间的字节数。

在现代编译器设计中，关于存储优化的算法并不罕见，在并行、嵌入式编译等领域，都有广泛的应用，其理论与实现技术可能是相对复杂的。不过，算法的核心仍然在于检测冲突与描述依赖关系。以怎样的数据结构及算法才能更有效地检测存储空间的访问冲突是一个值得深入研究的话题。

8.5 深入学习

通常，基于一个实际系统讨论运行时刻存储管理的问题是比较适合的。这里，笔者推荐几本程序设计语言及汇编程序设计方面的图书，供读者参考学习。

1、程序设计语言原理(原书第 7 版)

Robert W. Sebesta

机械工业出版社

说明：这本书是程序设计语言方面的巨著，深入细致地讲解了命令式语言的主要结构及其设计与实现。

2、Intel 汇编语言程序设计

Kip R. Irvine

电子工业出版社

说明：这本书是汇编语言程序设计方面的著作，以 IA-32 体系结构为例，并详细阐述了汇编语言程序设计的技巧。

3、高级编译器设计与实现（第 5 章）

Steven S. Muchnick

机械工业出版社

说明：这本书被誉为“鲸书”，涉及许多编译器设计中的高级话题，当然，它最值得关注的就是优化技术。

8.6 实践与思考

1. 请读者思考仅依赖 esp 实现寻址的技术难点是什么？是否可以在 Neo Pascal 的基础上扩展实现？
2. 标准 Pascal 并不支持可变参数机制。请读者尝试在 Neo Pascal 的基础上进行扩展，使之支持类似于 C 语言的可变参数机制。
3. 试问为什么 C 语言不支持函数返回类型为数组？
4. 请读者在 Neo Pascal 的基础上，实现存储优化算法，使其能支持跨基本块、多次引用定值的临时变量的优化。
5. 如何分析与描述普通变量的冲突信息？

8.7 大师风采——Bjarne Stroustrup

Bjarne Stroustrup: 计算机科学家，C++语言创始人。1950年12月30日出生于丹麦。1975年，Stroustrup 获得丹麦奥尔胡斯大学数学和计算机科学硕士学位，并于1979年获得英国剑桥大学计算机科学博士学位。他是 AT&T 大规模程序设计研究部门负责人。目前，任教于美国德州农机大学计算机工程学院。

Stroustrup 最令世人瞩目的研究成果就是 C++语言的设计与实现。1979年，他开始研发 C++语言，最初的 C++编译器的目标程序就是 C 语言。在过去的 20 年中，C++成为了最炙手可热的编程语言之一。C++的盛行使面向对象程序设计思想从理论研究正式走向了实践中，让软件开发人员切实体会到了面向对象程序设计的优点。

第 9 章 目标代码生成

The raising of the status of programmers suggested by the Theory Building View will have to be supported by a corresponding reorientation of the programmer education.

—Peter Naur

9.1 目标代码生成概述

9.1.1 目标代码生成基础

目标代码生成，亦称为代码生成，是经典编译模型中最后一个阶段。其输入是经过语法、语义分析及优化后得到的 IR，而其输出则是特定的目标语言程序。随着汇编语言的成熟，现代高级语言编译器已经很少直接将源程序转换为机器代码了，通常会选择汇编语言作为编译器的目标语言。不过，即使如此，代码生成器的设计仍然很大程度上依赖于目标机的体系结构。因此，设计者不必过多顾虑该模块与目标机的耦合程度，这是不可避免的。

编译器对代码生成器的要求是比较严格的。通常，必须遵守如下几个原则：

(1) 语义等价。这是代码生成器的设计底线，准确且等价的转换才是有意义的。语义等价是编译器设计的基本原则，笔者已经多次声明，其重要性不言而喻。

(2) 相对高效。理论上的最优代码是不可判定的。当然，很大程度上也是不可达到的。不过，这并不意味着设计者可以任意放纵资源耗费。即使是硬件资源不再稀缺的今天，目标程序的性能仍然是评价编译器设计的一个重要指标。为了得到相对较优的代码，编译器还引入了一些基于目标代码实现的优化算法，通常称为指令级优化。关于指令级优化的话题，笔者将在本章中稍作介绍。除了指令级优化之外，存储层的优化也是提高目标程序效率的有效途径。众所周知，寄存器、cache 的访问速度远胜于内存，因此，如何合理分配寄存器以及如何提高 cache 的命中率是编译器设计需要关注的。当然，就算法实现而言，这可能是比较复杂的。

(3) 扩展性好。在通用机编译器中，读者可能不太容易体会其优越性。因为要真正做到通用机层次上的跨目标机应用，除了编译器的因素之外，还必须考虑操作系统的差异。程序员更愿意接受像 Java、.NET 之类的跨平台应用，这可能更有现实意义。不过，在一些嵌入式系统的编译器中，可变目标的编译器可以使程序员省去很多烦心事。最著名的可变目标编译器就是 GCC，它的经典与完美已在前文阐述。当然，扩展性也是一柄双刃剑，实践证明，即使是经典的可变目标的编译器也很难与专用编译器媲美。例如，在 i386 结构上，无论是目标程序的性能还是编译器本身的效率，GCC 都无法与 Intel C++ 相比。

鉴于以上几点，试图设计一个相对完备的代码生成器并不是一项简单的工程。通常，现代编译技术的观点认为，代码生成器主要完成如下三项任务：



- (1) 指令选择：也称为指令筛选，即选择一个实现 IR 操作的目标机指令序列。
- (2) 寄存器分配：考虑如何有效地利用有限的寄存器资源，以提高代码的执行效率。
- (3) 指令调度：也称为指令排序，就是确定指令的执行顺序。

以上三个话题所涉及的理论与技术是非常丰富的，并不亚于编译优化技术，因此，本书无法详细阐述相关细节。鉴于 Neo Pascal 的实现及本书篇幅，本章只涉及指令选择、寄存器分配的一些基本问题，不讨论指令调度的相关技术与实现。

9.1.2 指令选择

指令选择的难易程度主要取决于目标机的指令集，并不能一概而论。实践经验证明，CISC 指令集比 RISC 指令集更有利于编译器实现，这是因为所拥有的资源（指令）相对较丰富。例如，试图在没有浮点运算指令的目标机上实现浮点数运算，那么，编译器设计者就不得不考虑如何使用整数运算指令模拟实现。当然，这并不是绝对的。在资源相对丰富的情况下，设计者可能会面临新的问题，那就是如何合理调配资源，使得目标结果相对更优。这就意味着，可以达到相同功能的指令或指令序列可能并不唯一时，如何选择更优的方案是值得考虑的。

在不考虑目标程序的效率的情况下，指令选择的实现是非常简单的。只需要根据 IR 的实际语义，为每种 IR 设计一个相对独立的目标代码片段。在代码生成期间，按照 IR 与目标代码片段的一一对应关系，将 IR 转换为语义等价的目标代码片段即可。在设计目标代码片段时，由于并不考虑各句 IR 之间的联系，因此，这种处理方案是完全可以保证安全的。那么，这种方案的缺点是什么呢？笔者引入一个简单的例子。例如，a、b、c 都是 integer 型的全局变量，则 IR (ADD_4 a, b, c) 的目标代码序列如下：

```
mov    eax, a
add    eax, b
mov    c, eax
```

在单机情况下，这种翻译方案始终是安全的。不过，这种代码生成方案的缺点是代码质量较差，极易产生冗余代码。例如，IR 序列如下：

```
(ADD_4 a, b, c)
(ADD_4 c, b, a)
```

那么，得到的目标代码序列如下：

```
mov    eax, a
add    eax, b
mov    c, eax
mov    eax, c
add    eax, b
mov    a, eax
```

仔细观察这段代码序列，可以发现如下几个问题：

- (1) 第 4 条指令是冗余的。当第 3 条指令执行完成后，eax 寄存器中存储的值就是 c 变

量的值，因此，再次从 `c` 变量内取值是冗余的。

(2) 如果编译器可以确定以后将不存在 `c` 的引用，则第 3 条指令也是冗余的。计算机完全可以依赖 `eax` 内的计算结果直接参与运算，而不需要将其值映射到 `c` 单元中。

理论上讲，评价目标代码质量的标准是显而易见的，即执行效率与代码长度，这是指令选择需要关注的问题。不过，实践证明，有时这种评价是很难进行的。例如，代码长度的准确定义是什么？到底是指汇编指令的行数，还是机器指令的字节数？当然，理想情况下，人们更希望以后者作为评估标准。然而，对于某些指令码不等长的指令系统而言，试图准确预计机器指令的大小是比较困难的。有时，即使设计者竭尽所能，也是徒劳无功的。因此，退而求其次也是可以接受的。

9.1.3 寄存器分配

寄存器是目标机处理器提供的一种有限的资源，即使是今天，寄存器依然是非常稀缺的。除了可以作为存储层次中最高等级的通用存储资源之外，更重要的是，寄存器还是指令系统的一个重要组成部分。换句话说，许多指令对某些寄存器的使用都是有特殊限制的。例如，在 `i386` 系统中，`idiv` 指令的运算结果必须存放在 `eax`、`edx` 中。事实上，这种情况是比较常见的，并且是因目标机指令系统而异的。

那么，寄存器分配的目标是什么呢？简而言之，就是尽可能充分利用寄存器保存有用（有效）数据且设法使有用数据尽可能长久地保存在寄存器中。值得注意的是，这里讨论的数据仅局限于有用数据，对于无用数据而言，讨论寄存器分配是无意义的。而哪些才是有用数据呢？这就要依赖数据流分析结果了。

即使是基于理想的虚拟机讨论，试图得到最优的寄存器分配方案也是非常困难的，这是一个 `NP` 完全问题。然而，实际目标机情况可能更复杂，因为还需考虑指令系统、操作系统等因素对寄存器使用的限制。下面举一个简单的例子，说明寄存器分配的原理。

例 9-1 假设存在 `eax`、`ebx` 两个寄存器，设计寄存器分配方案，并生成目标代码。

```
(ADD_4 a,b,e)
(ADD_4 c,e,b)
(ADD_4 d,e,e)
```

在考虑寄存器分配的情况下，得到的目标代码如下：

```
mov    eax,a        ;eax 保存的是 a 的值
add    eax,b        ;eax 保存的是 e 的值
mov    ebx,eax      ;eax、ebx 保存的都是 e 的值
add    eax,c        ;eax 保存的是 b 的值
mov    b,eax        ;如果以后不使用，可以省略
add    ebx,d        ;eax 的值已无用，ebx 保存的是 e 的值
mov    e,ebx        ;如果以后不使用，可以省略
```

无论是经典编译技术还是现代编译技术，关于寄存器分配策略的研究一直是人们所关注的。当然，这是一个有趣且复杂的问题，有兴趣的读者可以参考“鲸书”或《The Compiler Design Handbook》。



9.2 目标机简介

9.2.1 目标机结构

正如人们所知的，代码生成是一种与目标机密切相关的技术，因此，笔者有必要对目标机结构作简单介绍。通常，IA-32 体系结构指的就是从 Intel 386 到最新的 32 位奔腾 4 处理器，是 Intel 经典的 32 位机系统。从程序员及编译器设计者的角度来看，除了性能的提高及引进了一套用于提升多媒体处理的高性能指令集之外，IA-32 的体系与最初的 Intel 386 仍然是一脉相承的。

1. 操作模式

i386 处理器有三种基本操作模式：保护模式、实模式、系统管理模式。而虚拟 8086 模式可以视作是保护模式下的一种特例。

保护模式 (protected mode) 是处理器的主要状态，尤其是处于操作系统托管之下的应用程序。在保护模式下，用户程序被赋予了独立的内存区域 (称为“段”)，试图访问该区域之外的存储单元将被视为非法的。在这种情况下，用户程序试图访问绝对物理地址或硬件接口是很难的。

虚拟 8086 模式 (virtual-8086 mode) 是为了解决在保护模式下兼容实模式程序而设计的一种处理方案。早期的程序大多数都是实模式的，为了这些程序同样可以运行在保护模式之下，设计者提供了一个虚拟 8086 模式。严格地说，它并不是一个独立的模式。

实模式 (real-address mode) 也称为“实地址模式”，允许用户程序直接访问与控制存储器或硬件设备。同时，在实模式情况下，允许程序员进行模式切换等特殊应用。Intel 处理器都是从实模式引导的。不过，此后大多数操作系统会将其切换到保护模式。Windows 98 是支持实模式的，而之后的 Windows 操作系统都不支持了，即使是运行所谓的“MS-DOS”，也只是虚拟 8086 模式而已。从操作系统设计角度而言，实模式是不安全的，而且难以监控。

系统管理模式 (system management mode) 也称为“系统管控模式”，是提供给操作系统用以实现电源管理及系统安全等功能的机制。通常，自定义系统启动过程等特殊应用可能会涉及该模式。

一般来说，编译器设计者及程序员需要考虑的就是保护模式下的程序设计，实模式的应用涉及较少。

2. 基本寄存器

在 i386 系统中，通常提供了 8 个通用寄存器、6 个段寄存器、1 个标志寄存器 (EFLAGS) 和 1 个指令指针 (EIP)。

通用寄存器：主要用于进行运算和数据传递。在实际目标机中，除了用作数据存储之外，有时，通用寄存器还有一些特殊的功能与应用限制，这是编译器设计者需要考虑的因素。

段寄存器：主要用于存储段，如代码段、数据段、栈段等。在保护模式下，段寄存器的处理是由操作系统完成的，通常不需要程序员考虑。

标志寄存器：存储控制 CPU 操作或反映 CPU 状态的独立二进制位。

指令指针：用于标识下一条待执行指令的地址。任意改变该寄存器的值，可能导致程序执行的异常，因此，这个寄存器的值通常是受控访问的。

下面，笔者重点介绍 8 个 32 位通用寄存器。它们分别是 `eax`、`ebx`、`ecx`、`edx`、`ebp`、`esp`、`edi`、`esi`。其中，`eax`、`ebx`、`ecx`、`edx` 都可以进行 8 位、16 位、32 位寻址。而 `ebp`、`esp`、`edi`、`esi` 只可以进行 16 位、32 位寻址，不过，它们的 16 位寻址仅限于实模式下。

以 `eax` 为例，其低 16 位称为 `ax`，`ax` 的高 8 位称为 `ah`，低 8 位称为 `al`。而 `eax` 的高 16 位是不可以独立寻址的。这种交织的关系同样适用于 `ebx`、`ecx`、`edx`。如图 9-1 所示。

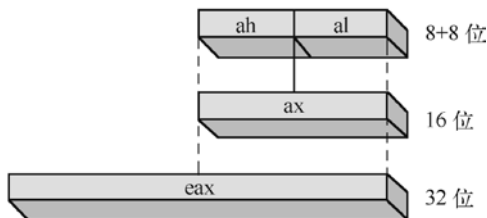


图 9-1 通用寄存器寻址示例

最后，再来谈谈通用寄存器的一些特殊应用场合。

- (1) 在乘法和除法指令中，`eax`、`edx` 被自动应用，赋予了特定的作用。
- (2) 在字符串处理、循环处理等场合中，`ecx` 作为循环计数器使用。
- (3) `esp` 是寻址堆栈的栈顶指针寄存器，通常不能另作他用。
- (4) 在字符串块数据传输时，`esi`、`edi` 通常作为源和目标块的指针寄存器。
- (5) 在程序设计中，`ebp` 经常用于处理函数参数和局部变量的寻址。关于 `ebp` 的应用，第 8 章中已作了详细阐述。

9.2.2 浮点处理单元

IA-32 结构的处理器都提供了浮点处理单元 (FPU)。在 i486 之前的处理器中，FPU 是独立于主处理器而存在的，通常称为“协处理器”。在 i486 之后，由于主处理器运算性能的提高，FPU 也被集成到了主处理器中。不过，这对于程序员似乎是透明的。事实上，直到今天，从应用的角度来说，FPU 仍然是以一个逻辑整体的形式存在的。由于 8086 没有 FPU，因此，关于 FPU 的原理及其应用，国内汇编语言教材很少涉及。在此，笔者有必要稍作详解。

FPU 共有 8 个 80 位的浮点数据寄存器，分别命名为 `R0`、`R1`、 \dots 、`R7`。不过，从程序员的角度来看，这 8 个寄存器的逻辑标识为 `ST(0)`、`ST(1)`、`ST(2)`、`ST(3)`、`ST(4)`、`ST(5)`、`ST(6)`、`ST(7)`。两种命名没有必然的联系，这是非常奇怪的，笔者稍后解释。当然，FPU 还拥有独立的指令指针、数据指针、标记寄存器、控制寄存器、状态寄存器。

1. 浮点数据寄存器

在 i386 处理器结构中，FPU 是一个模拟的栈式机器。与通用寄存器不同，浮点数据寄存器是以循环堆栈的形式组织的，如图 9-2 所示。图中 `TOP` 指向的是栈顶寄存器，它占用了状态寄存器中的 3 位。由于 `TOP` 栈顶可能是任意的，因此，要直接访问 `TOP` 指向的物理寄存器就必须经过变址寻址，同时，还要考虑偏移的取模处理。为了便于程序员访问，处理器将根据 `TOP` 的当前位置，为每个寄存器分配一个别名。将

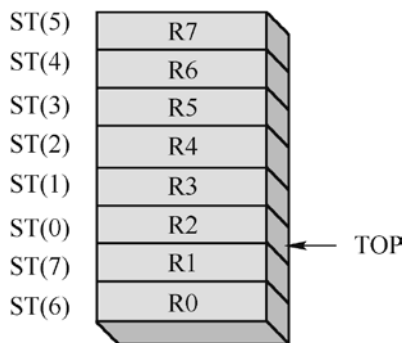


图 9-2 FPU 寄存器堆栈

TOP 指向的栈顶寄存器命名为 ST(0)，然后，沿栈顶向栈底的方向，依次标识为 ST(1)~ST(7)。随着 TOP 的变化，处理器将自动改变寄存器的别名。

2. 浮点运算的过程

众所周知，浮点数通常是依据 IEEE 754 格式存储的。关于 IEEE 754 格式的规范，笔者就不再详述了。浮点运算的过程大致分为三步：

- 1) 将操作数装入浮点数据寄存器堆栈。
- 2) 执行浮点运算指令。根据指令所需的操作数个数，从堆栈获取操作数，并将运算结果再次压入堆栈。
- 3) 将运算结果存储至目标单元。

值得注意的是，在 IA-32 结构中，FPU 与主 CPU 之间的操作数传递只能借助内存进行，而不能直接从主 CPU 的寄存器传递给 FPU。

例 9-2 浮点运算实例，编程实现计算表达式 $(8.12 \times 2.3) + (12.3 \times 9.2)$ 。

```
.data
a dd 8.12
b dd 2.3
c dd 12.3
d dd 9.2
e dd ?
.code
main proc
    finit                ;初始化 FPU，这不是必需的
    fld a                ;将 8.12 压入浮点数据寄存器堆栈，即 ST(0)=8.12
    fmul b                ;ST(0)=ST(0) * 2.3=18.676
    fld c                ;将 12.3 压入浮点数据寄存器堆栈，即 ST(0)=12.3, ST(1)=18.676
    fmul d                ;ST(0)=ST(0) * 9.2=113.16
    fadd                 ;ST(0)=ST(0)+ST(1)=113.16+18.676=131.836
    fstp e               ;将 ST(0)的值存入 e
    exit
main endp
end main
```

3. FPU 指令集

表 9-1 是数据传递操作指令列表。

表 9-1 数据传递操作指令列表

指令格式	指令含义	执行的操作
FLD src	装入实数到 st(0)	st(0) <- src (mem32/mem64/mem80)
FILD src	装入整数到 st(0)	st(0) <- src (mem16/mem32/mem64)
FBLD src	装入 BCD 数到 st(0)	st(0) <- src (mem80)
FLDZ	将 0.0 装入 st(0)	st(0) <- 0.0
FLD1	将 1.0 装入 st(0)	st(0) <- 1.0
FLDPI	将 pi 装入 st(0)	st(0) <- ?(ie, pi)
FLDL2T	将 log ₂ (10)装入 st(0)	st(0) <- log ₂ (10)

(续)

指令格式	指令含义	执行的操作
FLDL2E	将 $\log_2(e)$ 装入 $st(0)$	$st(0) \leftarrow \log_2(e)$
FLDLG2	将 $\log_{10}(2)$ 装入 $st(0)$	$st(0) \leftarrow \log_{10}(2)$
FLDLN2	将 $\log_e(2)$ 装入 $st(0)$	$st(0) \leftarrow \log_e(2)$
FST dest	保存实数 $st(0)$ 到 $dest$	$dest \leftarrow st(0)$ (mem32/mem64)
FSTP dest		$dest \leftarrow st(0)$ (mem32/mem64/mem80); 然后再执行一次出栈操作
FIST dest	将 $st(0)$ 以整数保存到 $dest$	$dest \leftarrow st(0)$ (mem32/mem64)
FISTP dest		$dest \leftarrow st(0)$ (mem16/mem32/mem64); 然后再执行一次出栈操作
FBST dest	将 $st(0)$ 以 BCD 保存到 $dest$	$dest \leftarrow st(0)$ (mem80)
FBSTP dest		$dest \leftarrow st(0)$ (mem80); 然后再执行一次出栈操作

表 9-2 是数据比较指令列表。

表 9-2 数据比较指令列表

指令格式	指令含义	执行的操作
FCOM	实数比较	将标志位设置为 $st(0) - st(1)$ 的结果标志位
FCOM op	实数比较	将标志位设置为 $st(0) - op$ (mem32/mem64) 的结果标志位
FICOM op	和整数比较	将 Flags 值设置为 $st(0) - op$ 的结果 op (mem16/mem32)
FICOMP op	和整数比较	将 $st(0)$ 和 op 比较 op (mem16/mem32) 后; 再执行一次出栈操作
FTST	零检测	将 $st(0)$ 和 0.0 比较
FUCOM $st(i)$		比较 $st(0)$ 和 $st(i)$ [486]
FUCOMP $st(i)$		比较 $st(0)$ 和 $st(i)$, 并且执行一次出栈操作
FUCOMPP $st(i)$		比较 $st(0)$ 和 $st(i)$, 并且执行两次出栈操作
FXAM		测试 $st(0)$, 如正值、负值等

表 9-3 是数据运算指令列表。

表 9-3 数据运算指令列表

指令格式	指令含义	执行的操作
加法		
FADD	加实数	$st(0) \leftarrow st(0) + st(1)$
FADD src		$st(0) \leftarrow st(0) + src$ (mem32/mem64)
FADD $st(i), st$		$st(i) \leftarrow st(i) + st(0)$
FADDP $st(i), st$		$st(i) \leftarrow st(i) + st(0)$; 然后执行一次出栈操作
FIADD src	加上一个整数	$st(0) \leftarrow st(0) + src$ (mem16/mem32)
减法		
FSUB	减去一个实数	$st(0) \leftarrow st(0) - st(1)$
FSUB src		$st(0) \leftarrow st(0) - src$ (reg/mem)



(续)

指令格式	指令含义	执行的操作
FSUB st(i),st		st(i) <-st(i) - st(0)
FSUBP st(i),st		st(i) <-st(i) - st(0), 然后执行一次出栈操作
FSUBR st(i),st	用一个实数来减	st(0) <- st(i) - st(0)
FSUBRP st(i),st		st(0) <- st(i) - st(0), 然后执行一次出栈操作
FISUB src	减去一个整数	st(0) <- st(0) - src (mem16/mem32)
FISUBR src	用一个整数来减	st(0) <- src - st(0) (mem16/mem32)
乘法		
FMUL	乘以一个实数	st(0) <- st(0) * st(1)
FMUL st(i)		st(0) <- st(0) * st(i)
FMUL st(i),st		st(i) <- st(0) * st(i)
FMULP st(i),st		st(i) <- st(0) * st(i), 然后执行一次出栈操作
FIMUL src	乘以一个整数	st(0) <- st(0) * src (mem16/mem32)
除法		
FDIV	除以一个实数	st(0) <-st(0) /st(1)
FDIV st(i)		st(0) <- st(0) /t(i)
FDIV st(i),st		st(i) <-st(0) /st(i)
FDIVP st(i),st		st(i) <-st(0) /st(i), 然后执行一次出栈操作
FIDIV src	除以一个整数	st(0) <- st(0) /src (mem16/mem32)
FDIVR st(i),st	用实数除	st(0) <- st(i) /st(0)
FDIVRP st(i),st		FDIVRP st(i),st
FIDIVR src	用整数除	st(0) <- src /st(0) (mem16/mem32)
FSQRT	平方根	st(0) <- sqrt st(0)
FSCALE	按 2 的 st(1)次方缩放 st (0)	st(0) <- st(0)*(2 ^{ST(1)})
EXTRACT	分解浮点数的指数与有效数值	st(0) <- st(0)的幂; st(0) <- st(0)的有效数字
FPREM	取余数	st(0) <-st(0) MOD st(1)
FPREM1	取余数 (IEEE), 同 FPREM, 但是使用 IEEE 标准[486]	
FRNDINT	取整 (四舍五入)	st(0) <- INT(st(0)); depends on RC flag
FABS	求绝对值	st(0) <- ABS(st(0)); removes sign
FCHS	改变符号位(求负数)	st(0) <-st(0)
F2XM1	计算 2 ^x -1	st(0) <- (2 ^{st(0)}) - 1
FYL2X	计算 Y * log ₂ (X)	st(0)为 Y; st(1)为 X; 将 st(0)和 st(1)变为 st(0)*log ₂ (st(1))的值
FCOS	余弦函数 Cos	st(0) <- COS(st(0))
FPTAN	正切函数 tan	st(0) <- TAN(st(0))
FPATAN	反正切函数 arctan	st(0) <- ATAN(st(0))
FSIN	正弦函数 sin	st(0) <- SIN(st(0))
FSINCOS	sincos 函数	
FYL2XP1	计算 Y * log ₂ (X+1)	st(0)为 Y; st(1)为 X; 将 st(0)和 st(1)变为 st(0)*log ₂ (st(1)+1)的值
处理器控制指令		

(续)

指令格式	指令含义	执行的操作
FINIT	初始化 FPU	
FSTSW AX	保存状态字的值到 AX	AX<- MSW
FSTSW dest	保存状态字的值到 dest	dest<-MSW (mem16)
FLDCW src	从 src 装入 FPU 的控制字	FPU CW <-src (mem16)
FSTCW dest	将 FPU 的控制字保存到 dest	dest<- FPU CW
FCLEX	清除异常	
FSTENV dest	保存环境到内存地址 dest 处 保存状态字、控制字、标志字和异常指针的值	
FLDENV src	从内存地址 src 处装入保存的环境	
FSAVE dest	保存 FPU 的状态到 dest 指向的 94 个字节的存储单元中	
FRSTOR src	从 src 处装入由 FSAVE 保存的 FPU 状态	
FINCSTP	增加 FPU 的栈指针值	st(6) <-st(5); st(5) <-st(4),...,st(0) <-?
FDECSTP	减少 FPU 的栈指针值	st(0) <-st(1); st(1) <-st(2),...,st(7) <-?
FFREE st(i)	标志寄存器 st(i)未被使用	
FNOP	空操作, 等同于 CPU 的 nop	st(0) <-st(0)
WAIT/FWAIT	同步 FPU 与 CPU: 停止 CPU 的运行, 直到 FPU 完成当前操作码	
FXCH	交换指令, 交换 st(0)和 st(1)的值	

9.2.3 操作数寻址方式

操作数寻址方式一直是汇编语言程序设计的一个重要话题。i386 系统提供了丰富的寻址方式, 如立即数寻址、直接内存寻址、寄存器寻址、间接寻址、变址寻址。

1. 立即数寻址

立即数寻址的操作数就包含在指令中, 它是指令码的一部分。值得注意的是, 在处理多字节立即数寻址时, 高字节的数值存放在高地址存储单元, 而低字节的数值存储在低地址存储单元内。在汇编语言程序设计中, 通常称之为“小尾顺序”。例如, `mov eax, 10203040h` 指令的执行结果如图 9-3 所示。

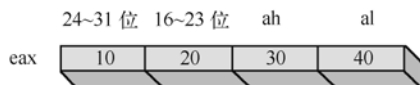


图 9-3 多字节立即数寻址示例

2. 直接内存寻址

直接内存寻址即操作数存储于内存中, 指令中包含的是该操作数所在存储单元的地址。注意, 这里的地址并不是真正的绝对地址, 实际上, 只是数据段内的偏移地址。而数据段的基地址则存储于 `ds` 寄存器中。例如, `mov eax, [1234h]` 指令的含义是将逻辑地址 1234h 存储单元内的数据存放到 `eax` 中。

3. 寄存器寻址

寄存器寻址即操作数存储于寄存器内, 指令中包含的是该操作数所在寄存器号。寄存器



寻址的速度相对于直接内存寻址快得多，因此，这也是寄存器分配的初衷。例如，`mov eax, ebx` 指令的含义是将 `ebx` 内的数据存储在 `eax` 中。

4. 间接寻址

间接寻址即实际的操作数存储于内存中，而操作数的地址存放在一个 32 位通用寄存器中。在汇编语言中，间接寻址就是使用方括号括起来的通用寄存器名表示。例如，`mov eax, [ebx]`。

在实模式下，只能用 `si`、`di`、`bx`、`bp` 作为间接寻址寄存器。其中，`bp` 常用来寻址堆栈段，而其他三个都是寻址数据段的。

在保护模式下，允许使用任意 32 位通用寄存器作为间接寻址寄存器。不过，间接寻址的范围仅限程序所属的数据段。当寻址超出该区域，则会产生保护故障。因此，在 Windows 环境下的高级语言中，对野指针、空指针的寻址可能是会导致异常的。然而，在实模式编程中，处理器是不会关注此类异常的。

5. 变址寻址

变址寻址即将一个常量的值与一个寄存器的值相加，产生一个有效地址，再根据这个有效地址获取实际数据值。这种寻址方式与高级语言的一维数组类似。这里的常量通常是一个数据块的首地址。而寄存器的值是一个相对于首地址的偏移，也称为“指针寄存器”。例如：

```
.data
    array dd 10h, 11h, 12h
.code
    mov ebx, 1
    mov eax, [array+ebx] ;在 masm 中，也可以写成 mov eax, array[ebx]
```

`array` 是数组的基址，`ebx` 中的值是相对于 `array` 的偏移。执行完该指令后，`eax` 的值为 `11h`。注意，在实模式下，只能用 `si`、`di`、`bx`、`bp` 作为指针寄存器，与间接寻址类似，`bp` 是用于堆栈段寻址的。值得注意的是，`mov eax, [array+1]` 指令并不是变址寻址，而是直接内存寻址，应该加以区别。两者的区别在于是否可以在汇编阶段计算得到操作数的地址。变址寻址是不可以计算得到的，而直接内存寻址是可以计算得到的。

9.2.4 ptr 操作符

`ptr` 操作符可以用于重载操作数的默认尺寸，这在 32 位汇编程序设计中是非常有用的。例如，用户书写 `mov al, aa` 指令的本意就是将 `aa` 的低 8 位送 `al` 寄存器。如果 `aa` 是一个 2 字节的变量时，由于 `mov` 指令左右操作数大小不匹配，这样的指令是非合法的。同样，`mov eax, aa` 指令也是不允许的。在这种情况下，需要将指令改写成如下形式：

```
mov al, byte ptr aa
mov eax, dword ptr aa
```

注意，`ptr` 必须和汇编器的标准数据类型联合使用，可以是 `byte`、`sbyte`、`word`、`sword`、`dword`、`sdword`、`fword`、`qword`、`tbyte`。在编译器设计中，`ptr` 操作符的应用是非常广泛的，它可以保证不同大小操作数之间的运算是合法的。

不过，值得注意的是，在使用 `ptr` 操作符将较小操作数向较大的操作数转换时，由于“小尾顺序”原则，会导致非预期结果。例如：

```
.data
aa dw    1234h, 5678h
bb db    12h, 34h, 56h, 78h
.code
...
mov eax, dword ptr aa    ;eax 的结果为 56781234h
mov ebx, dword ptr bb    ;ebx 的结果为 78563412h
```

9.2.5 一个完整的汇编程序

最后，笔者简单介绍一下完整的汇编程序的基本组成结构，这里，以保护模式下的汇编程序为例。一个完整的汇编程序通常包括以下几个部分：文件包含声明、数据段声明、代码段声明。下面是一个完整的汇编程序实例。

例 9-3 实现三个整数的加减运算。

```
.686P                ;指出该程序要求的最低 CPU。
.model flat,stdcall  ;指示编译器为保护模式程序生成的代码，stdcall 是调用约定。
include PasLib.inc   ;所需包含文件的列表，与 C 语言“#include”的功能类似。
.data                ;数据段声明。
    a dd 3000h       ;声明变量 a，初值为 3000h。
    b dd 2000h       ;声明变量 b，初值为 2000h。
    c dd 1000h       ;声明变量 c，初值为 1000h。
    d dd ?           ;声明变量 d，初值未定。使用“?”表示无需初始化该单元。
.code                ;.code 伪指令用来标记代码段声明的开始，语句必须置于其后。
aa proc              ;proc 伪指令标识 aa 过程开始。这里的 aa 是一个空过程。
;                    ;这里可以插入指令。
aa endp              ;endp 伪指令标记 aa 过程声明已结束。
start:               ;start 标号是具有特殊意义的，程序将从 start 开始执行。
    main proc        ;main 过程开始。但并不限制必须使用“main”作为启动过程。
    mov eax,a         ;将 a 的值装载到 eax 中。
    sub eax,b         ;计算 eax-b，结果存于 eax 中。
    add eax,c         ;计算 eax+c，结果存于 eax 中。
    mov d,eax         ;将 eax 的运算结果存储到 d 中。
    main endp        ;endp 伪指令标记 main 过程声明已结束。
end main             ;end 伪指令标明该行是程序的最末行，其后必须为启动过程名。
```

9.3 构造代码生成器

9.3.1 自动代码生成器基础

所谓“自动代码生成”指的是编译器设计者将与目标机指令相关的翻译规则抽象成外部文件（习惯上，将该文件称为“模式文件”），然后，按照 IR 的实际形式，由代码生成器进行模式匹配，检索合适的翻译规则，用于生成目标代码。应用模式文件实现目标代码生成器的优点就是便于修改、维护与移植。虽然这种目标代码生成器的设计较手工编码实现的生成



器稍复杂，但它的优点是后者无法比拟的。

本质上讲，IR 对生成正确的代码并没有决定性的影响，但它却是影响选择代码生成方法的主要因素。针对 DAG、树、四元式、三元式、前缀波兰代码等，寻找合适的代码生成方法是不容易的。其中，最著名的自动代码生成方法是 Graham-Glanville 方法，其基本思想是运用形式化的规则来描述目标机器的操作，并依据相应的指令模板进行代码生成。此过程与构造上下文无关文法分析器是类似的。

Graham-Glanville 代码生成器由三部分组成，即中间语言转换、模式匹配器和代码生成。代码生成的过程可为三步：

- 1) 将 IR 转换为适合匹配的模式串形式。
- 2) 根据模式串匹配指令模板库，确定指令序列。
- 3) 根据指令序列生成相应的目标代码。

注意，Graham-Glanville 代码生成器的默认输入 IR 是前缀波兰代码，这是早期代码生成器广泛应用的一种 IR 形式。不可否认的是，一些经典的自动代码生成器更热衷基于树、DAG、前缀波兰代码等形式的 IR 实现，而不是传统的三地址代码。这种设计的主要优点是可以最大程度上减少代码生成器对 IR 形式的要求。众所周知，AST、DAG 或其他变体形式是较高级的 IR，实际上，Graham-Glanville 算法第 1 步关注的就是将高级形式转换为 LIR 形式，然后，再从 LIR 形式中分析得到模式串。如果输入的 IR 是 MIR 或 LIR，那么，算法可以最大程度上适应 IR 的变化。本书将讨论基于三地址代码的自动代码生成器的实现，这种实现方法更多关注的是算法的第 2、3 步。

9.3.2 指令模板

在经典编译技术中，指令模板的实现技术方式主要有以下两种：

(1) 使用自动代码生成器的生成器来实现基于指令模板的代码生成。这种方法与语法分析的生成器 yacc 比较类似。根据生成器规则文件的要求，逐一书写模板规则，再由自动代码生成器的生成器将模板规则文件转换为相应的 C/C++ 源代码。这就是代码生成器的源代码，只需将其加入编译器项目重新编译即可。最著名的自动代码生成器的生成器称为“lburg”，它习惯上将规则文件称为“紧缩规范”。

(2) 手工编码实现指令模板。这种方法主要是在代码生成阶段从外部文件动态读入指令模板库。相比前者，这种方式的实现代价可能大一些，同时，由于必须借助外部 IO，代码生成的效率可能低一些。然而，这种方式较灵活，尤其在指令选择方案的灵活变化方面是有优势的。由于 Neo Pascal 是基于这种方法实现的，因此，本书将详细讨论其设计与实现的细节。

下面，笔者就来介绍 Neo Pascal 的指令模板形式。

由于 NPIR 是一种较低级的 MIR 形式，因此，给指令模板的实现带来了便利。Neo Pascal 并不需要考虑 AST 或 DAG 到 LIR 的转换。在指令模板的设计中，最复杂的描述可能就是操作数的来源，不同的指令系统对此的限制是非常奇怪的。例如，在 i386 中，浮点指令规定数据一定是来自内存的，而主 CPU 的大多数指令最多只允许存在一个内存操作数。类似的情况在指令模板的设计中必须予以体现，否则无法保证代码生成的正确性。设计指令模板是一项极富有挑战性的工作，而设计出经典的指令模板就更困难了，即使是经典的

Graham-Glanville 方法也没有太多的建议。下面，就来看看 Neo Pascal 指令模板的实例。

```
{
ADD_4 %V.1% %V.2% %V.3%
@1,4
@2,4
@3,4
add %V.1%,dword ptr %V.2%
}
```

大括号：一个完整的指令模板必须用大括号括起来，以便代码生成器解析。

模式串：指令模板的第一行是模板的模式串，用于匹配模板。在代码生成过程中，生成器根据 IR 的实现情形，可以分析得到类似的模式串。然后，根据得到的模式串，在指令模板库中检索。这个过程与数据库检索比较类似，而这里的关键字就是模式串。

操作数来源标记：ADD_4 一共有 3 个操作数。以“@”开头的行文本正是用于标识操作数的来源。其中，逗号之后的数值是用于标识操作数的字节数。而逗号之前的数值则用于标识操作数的来源，详细的说明见表 9-4。

指令序列：其中，除了“%V.1%”、%V.2%等之外的文本将在目标代码中输出，而“%V.1%”等标记将在代码生成中被替换。

表 9-4 指令模板的操作数来源标记

操作数	标记值	说明
操作数 1	0	直接内存寻址。如果操作数存在于寄存器中，则必须将其存到内存中，然后再从内存直接取值
	1	寄存器寻址。如果操作数不存在于寄存器中，则必须先将其传输至寄存器，然后再从该寄存器取值
	2	优先寄存器寻址。如果操作数存在于寄存器中，则优先从寄存器取值，否则从内存取值
操作数 2	0	直接内存寻址。如果操作数存在于寄存器中，则必须将其存到内存中，然后再从内存直接取值
	1	寄存器寻址。如果操作数不存在于寄存器中，则必须先将其传输至寄存器，然后再从该寄存器取值
	2	优先寄存器寻址。如果操作数存在于寄存器中，优先从寄存器取值，否则从内存取值
操作数 3	0	直接内存寻址。将运算结果回存到内存中
	1	申请一个寄存器，用于存储运算结果
	2	将运算结果存放在操作数 2 的寄存器中
	3	将运算结果存放在操作数 1 的寄存器中

下面再举一个指令模板的例子。

```
{
EQU_1 %V.1% %V.2% %V.3%
@1,1
@2,1
@lah,1
#ah,
cmp %V.1%,byte ptr %V.2%
lahf
}
```



```
and ah,40h
shr ah,6
}
```

这是一个比较特殊的应用。其中，`lahf` 指令的功能是将标志寄存器值存入 `ah` 寄存器中，因此，并不需要重新申请一个寄存器用于存储运算结果。其中，“`@1ah,1`”表示该操作数所属的寄存器是特指的。在本例中，即表示存储操作数 3 的寄存器只能是 `ah`，而不需要也不能申请其他寄存器。而本例中的“`#ah`”则表示该指令模板对 `ah` 寄存器有特殊需要，因此，在应用该指令模板生成代码时，不能将 `ah` 寄存器挪作他用。

最后，笔者通过一个实例来解释基于指令模板的代码生成过程。

例 9-4 基于指令模板的代码生成。

```
IR:      ADD_4   A,B,C
        ADD_4   C,A,B
```

处理 (`ADD_4 A,B,C`):

1) 根据 `ADD_4` 的指令模板，操作数 1 必须为寄存器寻址，由于变量 `A` 没有绑定任何寄存器，因此，代码生成器必须为 `A` 绑定一个 4 字节的寄存器（这里，假设为 `eax`）。并生成指令将 `A` 的值装入 `eax` 中。

指令：`mov eax,A`

2) 操作数 2 是优先寄存器寻址。`B` 变量不存在绑定寄存器，因此，只需从内存寻址即可。

3) 处理模板中的指令序列，将文本中的“`%V.1%`”、“`%V.2%`”替换为实际的操作数。

指令：`add eax,dword ptr [B]`

4) 根据 `ADD_4` 的指令模板，运算结果存放在操作数 1 所属的寄存器中，至此，`eax` 寄存器绑定的变量是 `C`，而不是 `A`。

处理 (`ADD_4 C,A,B`):

5) 根据 `ADD_4` 的指令模板，操作数 1 必须为寄存器寻址，`C` 与 `eax` 寄存器是绑定的。

6) 操作数 2 是优先寄存器寻址。`B` 变量不存在绑定寄存器，因此，只需从内存寻址即可。

7) 处理模板中的指令序列，将文本中的“`%V.1%`”、“`%V.2%`”替换为实际的操作数。

指令：`add eax,dword ptr [A]`

8) 根据 `ADD_4` 的指令模板，运算结果存放在操作数 1 所属的寄存器中，至此，`eax` 寄存器绑定的变量是 `B`，而不是 `C`。

生成的代码序列如下：

```
mov eax,A
add eax,dword ptr [B]
add eax,dword ptr [A] ;eax 寄存器所绑定的变量是 B
```

通过例 9-4 的讲解，笔者试图揭示基于指令模板进行代码生成的基本过程。不过，这个过程比较抽象，其中，忽略了许多实现细节，例如，指令序列的文本替换、寄存器的绑定、寄存器的分配、寄存器装入与回存等，这些细节将在后续章节中讨论。

9.3.3 寄存器描述

要有效进行寄存器分配、寄存器绑定等操作，代码生成器就必须依赖于一套完整的寄存器描述信息。当然，其复杂性很大程度上是依赖于目标机系统结构的。通常，代码生成器需要关注的信息包括寄存器的大小、寄存器的名称、寄存器的关系等。下面引入几个有用的数据结构，描述寄存器的相关信息。

【声明 9-1】

```
enum
{
    EAX=0, EBX, ECX, EDX, AX, BX, CX, DX, AH, BH, CH, DH, AL, BL, CL, DL, ESI, EDI
};
```

声明 9-1 列出的就是可参与分配绑定的寄存器，该列表中未表现的寄存器（如 `ebp`、`esp` 等）是不参与寄存器分配的。值得注意的是，浮点数据寄存器也是不可绑定的。

【声明 9-2】

```
static char Size2Reg[3][8]={
    {AL, BL, CL, DL, AH, BH, CH, DH},
    {AX, BX, CX, DX, -1, -1, -1, -1},
    {EAX, EBX, ECX, EDX, EDI, ESI, -1, -1}
};
```

`Size2Reg` 数组用于描述从字节数到寄存器号的映射关系。第 0 行记录的是 1 字节的寄存器，第 1 行记录的是 2 字节的寄存器，第 2 行记录的是 4 字节的寄存器。其中，“-1”表示空项。

【声明 9-3】

```
static char Reg2Size[]={4, 4, 4, 4, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 4, 4};
```

`Reg2Size` 数组用于描述从寄存器到字节数的映射关系。`Reg2Size` 与 `Size2Reg` 是互逆的映射，在分配寄存器时，这两个映射关系是非常有用的。

【声明 9-4】

```
static char RegRela[18][5]={
    {AL, AX, AH, -1, -1},           //与 eax 相关的寄存器
    {BL, BX, BH, -1, -1},         //与 ebx 相关的寄存器
    {CL, CX, CH, -1, -1},         //与 ecx 相关的寄存器
    {DL, DX, DH, -1, -1},         //与 edx 相关的寄存器

    {AL, EAX, AH, -1, -1},         //与 ax 相关的寄存器
    {BL, EBX, BH, -1, -1},         //与 bx 相关的寄存器
    {CL, ECX, CH, -1, -1},         //与 cx 相关的寄存器
    {DL, EDX, DH, -1, -1},         //与 dx 相关的寄存器

    {EAX, AX, -1, -1, -1},         //与 ah 相关的寄存器
```



```
{EBX, BX, -1, -1, -1}, //与 bh 相关的寄存器
{ECX, CX, -1, -1, -1}, //与 ch 相关的寄存器
{EDX, DX, -1, -1, -1}, //与 dh 相关的寄存器

{EAX, AX, -1, -1, -1}, //与 al 相关的寄存器
{EBX, BX, -1, -1, -1}, //与 bl 相关的寄存器
{ECX, CX, -1, -1, -1}, //与 cl 相关的寄存器
{EDX, DX, -1, -1, -1}, //与 dl 相关的寄存器

{-1, -1, -1, -1, -1}, //与 esi 相关的寄存器
{-1, -1, -1, -1, -1}, //与 edi 相关的寄存器
};
```

RegRela 数组用来描述寄存器之间的依赖关系。在介绍 i386 结构时，笔者曾提到过关于寄存器交织访问的问题。乍一看，这是一个比较奇怪的特性，但在实际应用中却是非常有用的。这里，需要关注的是寄存器之间的依赖关系。例如，eax 寄存器值的改变不仅影响 eax 本身，还会影响 ax、ah、al 三个寄存器的状态。同样，ax 寄存器值的改变也会影响 eax、ah、al 三个寄存器的状态。那么，这个依赖关系的作用是什么呢？实际上，代码生成器并不关注某一运行时刻的寄存器的实际存储值，当然，这也是不可能实现的。不过，寄存器在某一运行时刻所绑定的具体变量或存储空间却是代码生成器关心的。而 RegRela 的依赖关系对于寄存器的绑定是有重要意义的，例如，假设 eax 被绑定到一个 4 字节变量时，为了保证代码生成的安全，eax、ax、ah、al 四个寄存器的原有绑定必须全部解除。不过，值得注意的是，当 ah 被绑定时，eax、ax、ah 三个寄存器是必须解除原有绑定的，但 al 的原有绑定是不受影响的。其中，“-1”表示空项。

实际上，就可变目标的应用而言，更理想的做法是将寄存器的描述信息以外部文件格式形式存储，与指令模板库类似，代码生成器在初始化时将描述信息读入。不过，为了便于处理，本书略去了这部分实现。

9.3.4 寄存器分配

本书并不打算深入讨论如何寻求相对更优的寄存器分配方案，这将是一个复杂的话题。这里，关注的是一种简单且相对有效的寄存器分配方式。它的目标任务是根据指令模板中操作数的相关描述信息，分配一个能够满足应用需求的寄存器。为了跟踪寄存器的绑定情况，引入一个用于记录寄存器绑定信息的数据结构，声明形式如下：

【声明 9-5】

```
static map<int, list<OpInfo>> RegVal;
```

其中，RegVal 的关键字是寄存器号。需要说明的是，一个寄存器是可能同时与多个操作数绑定的。不过，这种情况必须谨慎处理。下面，就来看看寄存器分配的基本思想。

Neo Pascal 寄存器分配的基本思想是比较简单的。根据寄存器的大小，将寄存器分为 3 个组，即 1 字节寄存器组、2 字节寄存器组、4 字节寄存器组。并且设置一组游标变量，用于分配寄存器，即 AllocCursor 数组，如图 9-4 所示。

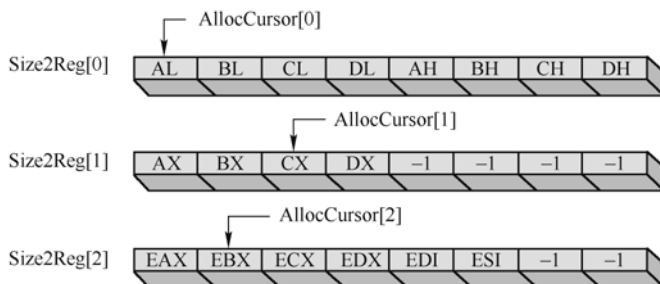


图 9-4 寄存器分配示意

以 1 字节寄存器分配为例，AllocCursor[0]指向的就是当前可分配的寄存器，而每一次分配请求之后，AllocCursor[0]游标后移一位。当 AllocCursor[0]游标移到行末时，则重新指向行首。这种分配策略与操作系统中的存储分配比较类似，当然，这里并不需要考虑回收的问题。值得注意的是，对分配算法而言，可以将寄存器资源分为三类，即“未绑定”寄存器、“已绑定”寄存器、“不可用”寄存器。

“未绑定”寄存器就是指未绑定具体变量且未被禁用的寄存器资源。这类寄存器是分配算法优先考虑的。

“已绑定”寄存器就是指已绑定了具体变量的寄存器资源。对于这类寄存器资源，原则上尽可能不作分配。不过，当“未绑定”寄存器已分配殆尽时，这类寄存器依然是可以参与分配的。

“不可用”寄存器就是指当前指令模板中有特殊限制的寄存器资源，这些寄存器是绝对不能参与分配的，否则会导致所生成的代码不正确。

下面，就来看看寄存器分配的源代码实现。

程序 9-1 Target.cpp

```

1  int CTarget::RegAlloc(int iSize,int iTarget)
2  {
3      if (iTarget!=-1 && iTarget!=-2)
4          return iTarget;
5      int j=iSize/2;
6      int iReg=-1;
7      set<int> TmpSet;
8      static int AllocCursor[3]={0,0,0};
9      int iTmpCursor=AllocCursor[j];
10     int iTmpReg=-1;
11     for (;TmpSet.count(iTmpReg)==0;iTmpCursor=((iTmpCursor+1)%8))
12     {
13         if(iTmpReg!=-1)
14             TmpSet.insert(iTmpReg);
15         if (Size2Reg[j][iTmpCursor] == -1)
16             continue;
17         if (iTarget==-2
18             && (Size2Reg[j][iTmpCursor]==ESI

```




```
19         || Size2Reg[j][iTmpCursor]==EDI))
20         continue;
21         iTmpReg=Size2Reg[j][iTmpCursor];
22         if (Forbid.count(iTmpReg)==0
23             && (RegVal.count(iTmpReg)==0
24                 || RegVal[iTmpReg].empty()))
25             {
26                 iReg=iTmpReg;
27                 break;
28             }
29     }
30     TmpSet.clear();
31     iTmpReg=-1;
32     if (iReg==-1)
33     {
34         for (;TmpSet.count(iTmpReg)==0;iTmpCursor=((iTmpCursor+1)%8))
35         {
36             if(iTmpReg!=-1)
37                 TmpSet.insert(iTmpReg);
38             if (Size2Reg[j][iTmpCursor] == -1)
39                 continue;
40             if (iTarget==-2
41                 && (Size2Reg[j][iTmpCursor]==ESI
42                     || Size2Reg[j][iTmpCursor]==EDI))
43                 continue;
44             iTmpReg = Size2Reg[j][iTmpCursor];
45             if (Forbid.count(iTmpCursor)==0)
46             {
47                 iReg=iTmpReg;
48                 break;
49             }
50         }
51     }
52     AllocCursor[j]=(iTmpCursor+1)%18;
53     if (iReg==-1)
54         EmitError("out of register");
55     return iReg;
56 }
```

第 3、4 行：参数 `iTarget` 是目标寄存器的编号。在默认情况下，`iTarget` 的值为-1，即表示由代码生成器随机分配寄存器。如果 `iTarget` 的值为-2，即表示由代码生成器随机分配可以交织寻址的寄存器。其他情况则表示必须分配参数 `iTarget` 所指定的寄存器。

第 6 行：`iReg` 用于记录分配得到的寄存器号。

第 7 行：`TmpSet` 用于记录算法曾经考察过的寄存器号，避免同一寄存器被重复考察。

第 8 行：`AllocCursor` 是用于分配寄存器的一组游标。

第 9 行：根据所需分配寄存器的大小，iTmpCursor 指向相应的游标。

第 11~25 行：考察大小满足条件的寄存器组，检索是否存在“未绑定”寄存器，如果存在未绑定寄存器，则直接予以分配。

第 13、14 行：在循环考察寄存器组时，TmpSet 集合用于避免同一寄存器被重复考察。如果当前寄存器号已存在于 TmpSet 集合中，则结束循环考察。

第 17~20 行：如果 iTarget 为-2，则不能分配 ESI、EDI。

第 22~28 行：判断当前寄存器是否为“未绑定”寄存器。Forbid 是一个全局集合，其中记录的是“不可用”寄存器。

第 32~51 行：如果 iReg 为-1，则表示当前已无“未绑定”寄存器可以分配。在这种情况下，就必须从“已绑定”寄存器中选择合适的寄存器予以分配，此时，就产生了寄存器溢出的情况。当然，“不可用”寄存器仍然是不参与分配的。

第 52 行：修改游标的指向。

第 53、54 行：如果没有适合的寄存器予以分配，则表示寄存器已耗尽。

本节所讨论的寄存器分配算法主要应用于较低层次的实现。对于更优的寄存器分配算法，这种修改并不会对当前的实现造成太大的影响。最经典的寄存器分配算法就是“图着色”算法，当所有“未绑定”寄存器均被占用时，如何选择溢出寄存器是该算法所关注的。寄存器溢出是有代价的，因此，需要评估决策。Neo Pascal 对此并没有作太多考虑，有兴趣的读者可以参考“鲸书”。

最后，再来看看寄存器溢出的相关实现。

程序 9-2 Target.cpp

```
1 void CTarget::SaveRegs(int iReg,bool bFlg)
2 {
3     for(int i=0;i<5 && RegRela[iReg][i]!=-1;i++)
4     {
5         SaveReg(RegRela[iReg][i],bFlg);
6         RegVal[RegRela[iReg][i]].clear();
7     }
8 }
```

以上代码遍历与当前寄存器有依赖关系的所有寄存器，并调用 SaveReg 函数，将寄存器的值回存到存储器中。

程序 9-3 Target.cpp

```
1 void CTarget::SaveReg(int iReg,bool bFlg)
2 {
3     set<int> TmpSet;
4     for(list<OpInfo>::iterator it=RegVal[iReg].begin();it!=RegVal[iReg].end();)
5     {
6         if (it->m_iType!=OpInfo::VAR || (bFlg && IsDeadVar(*it)))
7         {
8             it++;
```



```
9             continue;
10         }
11         if (TmpSet.count(it->m_iLink)==0)
12         {
13             TmpSet.insert(it->m_iLink);
14             int iSize=Reg2Size[iReg];
15             vector<string> ParaStr;
16             ParaStr.push_back(RegName[iReg]);
17             ParaStr.push_back("");
18             ParaStr.push_back(GetOpDataStr(*it));
19             IRtoAsmList("#MOV_"+itos(iSize)+" %V.1% %V.3%",ParaStr);
20         }
21         RegVal[iReg].erase(it++);
22     }
23 }
```

第 4~22 行：遍历指定寄存器的所有绑定变量信息，生成相应的 mov 指令，以便将寄存器的值回存到存储器中。

第 6~10 行：在 bFlg 为真的情况下，如果溢出变量为死变量，则不需要回存。在其他情况下，都需要回存。

第 11~20 行：调用 IRtoAsmList 函数生成 mov 指令，其中 ParaStr 用于传递参数。在指令模板库中，专门设置了三个模板（即#MOV_1、#MOV_2、#MOV_4）用于将寄存器的值回存到内存，这三个指令模板仅限寄存器溢出使用。当然，这里需要借助于 TmpSet 集合避免出现多次回存的情况。

9.3.5 代码生成器的基本结构

前面，已经详细介绍了寄存器分配的基本思想与实现。本小节将关注代码生成器主体结构的源码实现。通过本程序的源代码分析，读者可以了解到许多代码生成的实现细节，如指令模板的应用、寄存器的调度、寄存器溢出等。在详细分析源代码之前，笔者先引入一个指令模板的相关数据结构 Pattern。

【声明 9-6】

```
struct Pattern
{
    string szPattern;
    vector<Code> CodeList;
    vector<RegFlg> Flg;
    vector<int> SaveReg;
};
```

szPattern：模式串，这是指令模板的关键字。

CodeList：指令序列，这是目标代码的实体，Code 结构说明如声明 9-6 所示。

Flg：操作数的来源信息，包括来源标记的相关信息，RegFlg 结构说明如声明 9-7 所示。

SaveReg：不可用寄存器列表，即使用本指令模板前必须回存的寄存器。

【声明 9-7】

```
struct Code
{
    string Label;
    string Op;
    vector<string> Operand1;
    vector<string> Operand2;
    vector<string> Operand3;
    string Comment;
};
```

Label: 指令的标号。

Op: 指令的操作码。

Operand1: 指令的操作数 1。

Operand2: 指令的操作数 2。

Operand3: 指令的操作数 3。

Comment: 指令的注释。

【声明 9-8】

```
struct RegFlg
{
    char cFlg;
    char cReg;
    int iSize;
};
```

cFlg: 操作数的来源标志。

cReg: 如果操作数的值必须来源于某一特定寄存器, 则 cReg 记录的是该寄存器的编号。

iSize: 操作数的大小。

前面主要讨论了指令模板的相关数据结构, 这是剖析代码生成器源码的重要基础。代码生成器是以操作数来源作为主线实现的, 下面就来看看代码生成器的实现细节。

程序 9-4 Target.cpp

```
1 bool CTarget::IRtoASM()
2 {
3     CDataFlowAnalysis::DataFlowAnalysis();
4     pCurrBlock = NULL;
5     GenInit(State.m_szSysDir+"AsmScheme.txt");
6     CurrProcId=-1;
7     CurrAsmCodeList=GetAsmCodeList(CurrProcId);
8     IRtoAsmList("FILESTART",ParaStr1);
9     GetLibAsm();
10    MemoryAlloc();
11    RecOpAsm(".code");
12    for (int i=0;i<SymbolTbl.ProcInfoTbl.size();i++)
```



```
13     {
14         if (SymbolTbl.ProcInfoTbl[I].m_bUsed==false)
15             continue;
16         if (SymbolTbl.ProcInfoTbl.at(i).m_eFlag==ProcInfo::Extern)
17             continue;
18         if (SymbolTbl.ProcInfoTbl.at(i).m_szName=="_noname")
19             continue;
20         CurrProcId=I;
21         CurrAsmCodeList=GetAsmCodeList(CurrProcId);
22         RecContentAsm(";function:"+SymbolTbl.ProcInfoTbl.at(i).m_szName);
23         RecContentAsm(";process of start");
24         if (I==0)
25             IrtoAsmList("MAINSTART",ParaStr1);
26         ParaStr1.clear();
27         ParaStr1.push_back(itos(GetProcVarSize(i)));
28         ParaStr1.push_back("_"+SymbolTbl.ProcInfoTbl.at(i).m_szName);
29         IrtoAsmList("FUNCSTART %V.1% %V.2%",ParaStr1);
30         vector<CbasicBlock> Blocks=BasicBlock[I];
31         for(int j=0;j<Blocks.size();j++,RegClearAll())
32             {
33                 pCurrBlock = &(Blocks[j]);
34                 for (int p=Blocks[j].iStart;p<=Blocks[j].iEnd;p++)
35                     {
36                         int iSize,iReg1,iReg2,iReg3;
37                         vector<int> ExtReg;
38                         CurrIR = p;
39                         IRCode * tmp=&(SymbolTbl.ProcInfoTbl[I].m_Codes[p]);
40                         RecContentAsm(";"+GetIRStr(*tmp));
41                         if (tmp->m_eOpType==PARA)
42                             {
43                                 GenParaAsm(*tmp);
44                                 continue;
45                             }
46                         if (tmp->m_eOpType==ASSIGN_N)
47                             {
48                                 GenAssignNAsm(*tmp);
49                                 continue;
50                             }
51                         if (tmp->m_eOpType==ASM)
52                             {
53                                 string szIncludeAsm=SymbolTbl.ConstInfoTbl[
54                                     tmp->m_Op1.m_iLink].m_szName;
55                                 RecOpAsm(szIncludeAsm.substr(1,szIncludeAsm.length()-2)
56                                     ,tmp->m_Op2);
57                                 continue;
58                             }
59                     }
```

```
59         if (tmp->m_eOpType==RETV)
60         {
61             GenRetAsm(*tmp);
62             continue;
63         }
64         string szPatternStr=GetOpStr(tmp->m_eOpType);
65         if (tmp->m_eOpType>>4<<4 == OpType::IN)
66             szPatternStr="IN";
67         string szTmp=GetOpDataPattern(tmp->m_Op1);
68         if (szTmp!="")
69             szPatternStr=szPatternStr+" "+szTmp+".1%";
70         szTmp=GetOpDataPattern(tmp->m_Op2);
71         if (szTmp!="")
72             szPatternStr=szPatternStr+" "+szTmp+".2%";
73         szTmp=GetOpDataPattern(tmp->m_Rslt);
74         if (szTmp!="")
75             szPatternStr=szPatternStr+" "+szTmp+".3%";
76         Pattern AsmPattern;
77         if (CodePattern.SearchPattern(szPatternStr,AsmPattern))
78         {
79             ParaStr1.clear();
80             ExtReg.clear();
81             Forbid.clear();
82             for(int i=0;i<AsmPattern.SaveReg.size();i++)
83             {
84                 SaveRegs(i);
85                 RegForbid(i);
86             }
87             for(int i=3;i<AsmPattern.Flg.size();i++)
88             {
89                 int j=RegAlloc(AsmPattern.Flg[i].iSize
90                             ,AsmPattern.Flg[i].cReg);
91                 SaveRegs(j);
92                 RegForbid(j);
93                 ExtReg.push_back(j);
94             }
95             if (tmp->m_Op1.m_iType==OpInfo::VAR)
96             {
97                 iSize = AsmPattern.Flg[0].iSize;
98                 if (tmp->m_Op1.m_bRef)
99                 {
100                     iReg1=OpRef(tmp->m_Op1,AsmPattern.Flg[0].cReg,iSize);
101                     RegForbid(iReg1);
102                     ParaStr1.push_back(RegName[iReg1]);
103                 }
104             }
105             else
```



```
106         {
107     107         if (AsmPattern.Flg[0].cFlg==0)
108     108         {
109     109             iReg1=RegSeek(tmp->m_Op1,iSize);
110     110             if (iReg1!=-1)
111     111                 SaveReg(iReg1);
112     112             ParaStr1.push_back(GetOpDataStr(
113     113                 tmp->m_Op1));
114     114             iReg1=-1;
115     115         }
116     116         if (AsmPattern.Flg[0].cFlg==1)
117     117         {
118     118             iReg1=RegSeek(tmp->m_Op1,iSize);
119     119             if (iReg1===-1)
120     120             {
121     121                 iReg1=RegAlloc(iSize,AsmPattern.
122     122                 Flg[0].cReg);
123     123                 SaveRegs(iReg1);
124     124                 RegForbid(iReg1);
125     125                 LoadReg(iReg1,tmp->m_Op1);
126     126                 SetVal(iReg1,tmp->m_Op1);
127     127             }
128     128             else
129     129             {
130     130                 RegForbid(iReg1);
131     131             }
132     132             ParaStr1.push_back(RegName[iReg1]);
133     133         }
134     134         if (AsmPattern.Flg[0].cFlg==2)
135     135         {
136     136             iReg1=RegSeek(tmp->m_Op1,iSize);
137     137             if (iReg1===-1)
138     138                 ParaStr1.push_back(GetOpDataStr(
139     139                 tmp->m_Op1));
140     140             else
141     141                 ParaStr1.push_back(RegName[iReg1]);
142     142         }
143     143     }
144     144     }
145     145     else
146     146     {
147     147         if (tmp->m_Op1.m_iType == OpInfo::CONST
148     148             && AsmPattern.Flg[0].cFlg!=0)
149     149         {
150     150             iSize = AsmPattern.Flg[0].iSize;
151     151             vector<string> TmpPara;
```

```
152         iReg1=RegAlloc(iSize);
153         SaveRegs(iReg1);
154         RegForbid(iReg1);
155         TmpPara.push_back(GetOpDataStr(tmp->m_Op1));
156         TmpPara.push_back("");
157         TmpPara.push_back(RegName[iReg1]);
158         IRtoAsmList(
159             "#MOV_" + itos(iSize) + " %V.1%
160             %V.3%", TmpPara);
161         ParaStr1.push_back(RegName[iReg1]);
162     }
163     else
164     {
165         if (tmp->m_Op1.m_iType == OpInfo::NONE)
166             ParaStr1.push_back("");
167         else
168             ParaStr1.push_back(GetOpDataStr(
169                 tmp->m_Op1));
170     }
171 }
172 if (tmp->m_Op2.m_iType==OpInfo::VAR)
173 {
174     iSize = AsmPattern.Flgl[1].iSize;
175     if (tmp->m_Op2.m_bRef)
176     {
177         iReg2=OpRef(tmp->m_Op2,AsmPattern.Flgl[1].
178             cReg,i Size);
179         RegForbid(iReg2);
180         ParaStr1.push_back(RegName[iReg2]);
181     }
182     else
183     {
184         if (AsmPattern.Flgl[1].cFlg==0)
185         {
186             iReg2=RegSeek(tmp->m_Op2,iSize);
187             if (iReg2!=-1)
188                 SaveReg(iReg2);
189             ParaStr1.push_back(GetOpDataStr(
190                 tmp->m_Op2));
191             iReg2=-1;
192         }
193         if (AsmPattern.Flgl[1].cFlg==1)
194         {
195             iReg2=RegSeek(tmp->m_Op2,iSize);
196             if (iReg2==-1)
197             {
```




```
198         iReg2=RegAlloc(iSize,AsmPattern.  
199         Flg[1].cReg);  
200         SaveRegs(iReg2);  
201         RegForbid(iReg2);  
202         LoadReg(iReg2,tmp->m_Op2);  
203         SetVal(iReg2,tmp->m_Op2);  
204     }  
205     else  
206     {  
207         RegForbid(iReg2);  
208     }  
209     ParaStr1.push_back(RegName[iReg2]);  
210 }  
211 if (AsmPattern.Flg[1].cFlg==2)  
212 {  
213     iReg2=RegSeek(tmp->m_Op2,iSize);  
214     if (iReg2== -1)  
215         ParaStr1.push_back(GetOpDataStr  
216         (tmp->m_Op2));  
217     else  
218         ParaStr1.push_back(RegName[iReg2]);  
219 }  
220 }  
221 }  
222 else  
223 {  
224     if (tmp->m_Op2.m_iType==OpInfo::CONST)  
225     {  
226         iSize = AsmPattern.Flg[1].iSize;  
227         vector<string> TmpPara;  
228         iReg2=RegAlloc(iSize);  
229         SaveRegs(iReg2);  
230         RegForbid(iReg2);  
231         TmpPara.push_back(GetOpDataStr(tmp->m_Op2));  
232         TmpPara.push_back("");  
233         TmpPara.push_back(RegName[iReg2]);  
234         IRtoAsmList(  
235             "#MOV_" + itos(iSize) + " %V.1% %V.3%",  
236             TmpPara);  
237         ParaStr1.push_back(RegName[iReg2]);  
238     }  
239     else  
240     {  
241         if (tmp->m_Op2.m_iType == OpInfo::NONE)  
242             ParaStr1.push_back("");  
243     }  
else
```

```
245         ParaStr1.push_back(GetOpDataStr(tmp->m
246             _Op2));
247     }
248 }
249 if (tmp->m_Rslt.m_iType==OpInfo::VAR)
250 {
251     iSize = AsmPattern.Flg[2].iSize;
252     vector<string> TmpPara;
253     string szTmp = GetOpDataStr(tmp->m_Rslt);
254     if (tmp->m_Rslt.m_bRef)
255     {
256         iReg3=RegAlloc(4);
257         SaveRegs(iReg3);
258         SetVal(iReg3,tmp->m_Rslt);
259         TmpPara.push_back(GetOpDataStr(tmp->m_Rslt));
260         TmpPara.push_back("");
261         TmpPara.push_back(RegName[iReg3]);
262         RegRefSave();
263         IRtoAsmList("#MOV_4 %V.1% %V.3%",TmpPara);
264         TmpPara.clear();
265         TmpPara.push_back("[ "+RegName[iReg3]+" ]");
266     }
267     else
268     {
269         if (AsmPattern.Flg[2].cFlg==0)
270             iReg3=-1;
271         if (AsmPattern.Flg[2].cFlg==1)
272         {
273             iReg3=RegAlloc(iSize,AsmPattern.Flg[2].
274                 cReg);
275             SaveRegs(iReg3);
276             RegForbid(iReg3);
277             SetVal(iReg3,tmp->m_Rslt);
278             szTmp=RegName[iReg3];
279         }
280         if (AsmPattern.Flg[2].cFlg==2)
281             SetVal(iReg2,tmp->m_Rslt);
282         if (AsmPattern.Flg[2].cFlg==3)
283             SetVal(iReg1,tmp->m_Rslt);
284     }
285     ParaStr1.push_back(szTmp);
286 }
287 else
288 {
289     ParaStr1.push_back(GetOpDataStr(tmp->m_Rslt));
290 }
```



```
291         for(vector<int>::iterator
292             it=ExtReg.begin();it!=ExtReg.end();it++)
293             ParaStr1.push_back(Name2Reg[*it]);
294         IRtoAsmList(szPatternStr,ParaStr1);
295         LabelMap.clear();
296     }
297 }
298 }
299 RecContentAsm(";process of end");
300 if (SymbolTbl.ProcInfoTbl.at(i).m_eType==ProcInfo::Type::Function)
301 {
302     int iRetSize=SymbolTbl.TypeInfoTbl[SymbolTbl.ProcInfoTbl.at(i)
303         .m_iReturnType].m_iSize;
304     ParaStr1.clear();
305     ParaStr1.push_back(
306         "ebp+_ "+SymbolTbl.ProcInfoTbl.at(i).m_szName+"$RESULT");
307     if (iRetSize<=4)
308         IRtoAsmList("FUNCRET4 %V.1%",ParaStr1);
309     else if (iRetSize<=8)
310         IRtoAsmList("FUNCRET8 %V.1%",ParaStr1);
311     else
312         IRtoAsmList("FUNCRETA %V.1%",ParaStr1);
313 }
314 ParaStr1.clear();
315 ParaStr1.push_back(itos(GetProcParaSize(i)));
316 ParaStr1.push_back("_ "+SymbolTbl.ProcInfoTbl.at(i).m_szName);
317 IRtoAsmList("FUNCEND %V.1% %V.2%",ParaStr1);
318 }
319 CurrProcId=-2;
320 CurrAsmCodeList=GetAsmCodeList(CurrProcId);
321 IRtoAsmList("FILEEND",ParaStr1);
322 return true;
323 }
```

第 3 行：调用 `CDataFlowAnalysis::DataFlowAnalysis` 函数完成数据流分析。代码生成器对数据流信息的依赖程度是比较大的，尤其在评估死变量时。因此，数据流信息是必不可少的。

第 4 行：`pCurrBlock` 指针用于指向当前正在分析的基本块，在算法实现中，`pCurrBlock` 指针是非常有用的。

第 5 行：调用 `GenInit` 函数，初始化指令模板库。代码生成器将从系统文件夹中的 `AsmScheme.txt` 文件读取指令模板库。关于指令模板的格式，请参考 9.3.2 节。

第 6 行：`CurrProcId` 用于记录当前正在分析的函数符号。

第 7 行：`CurrAsmCodeList` 用于指向目标代码的插入点。关于目标代码的数据结构，笔者稍后详解。

第 8 行：调用 `IRtoAsmList` 函数，生成目标代码。`IRtoAsmList` 函数的第 1 个参数就是用

于匹配指令模板的模式串。而第 2 个参数是根据 IR 产生的参数列表，这个参数列表将在匹配到指令模板后，用于替换模板中指令序列的参数，以便最终生成目标代码。“FILESTART”模板主要用于在汇编代码列表中生成文件头部声明。

第 9 行：调用 GetLibAsm 函数，在汇编代码列表中生成库引用的声明。

第 10 行：调用 MemoryAlloc 函数，在汇编代码列表中生成变量、常量等符号声明。

第 11 行：调用 RecOpAsm 函数，在汇编代码列表中生成“.code”文件。

第 12~306 行：遍历过程信息表。代码生成是以过程为单位进行的。

第 14、15 行：忽略未被引用的过程。这些过程是不需要生成目标代码的。

第 16、17 行：忽略外部过程或函数，这些过程也是不需要生成目标代码的。

第 18、19 行：忽略匿名过程或函数，这些过程可以视作未实例化的过程类型。

第 20 行：令 CurrProcId 指向当前过程符号。

第 21 行：令 CurrAsmCodeList 指针指向当前过程的汇编代码列表。

第 22、23 行：在汇编代码列表中生成关于当前过程的注释说明。

第 24、25 行：如果当前过程是主程序，则应用“MAINSTART”模板生成“start:”标号。

第 27~29 行：应用“FUNCSTART”模板生成函数起始部分的相关代码，主要是运行时栈的维护代码。其中，ParaStr1 列表的第 2 个参数是当前过程或函数所属局部变量的空间大小。

第 30 行：获取当前过程的基本块列表。

第 31~286 行：遍历当前过程的所有基本块。基本块是寄存器分配与管理的单位。在 Neo Pascal 中，跨越基本块的寄存器管理不予考虑。

第 34~285 行：遍历当前基本块的 IR 序列，逐一生成相应的目标代码。

第 39 行：获取当前 IR。

第 40 行：在汇编代码列表中生成关于当前 IR 的注释说明。

第 41~45 行：如果当前 IR 的操作码为 PARA，则调用 GenParaAsm 生成相应的目标代码。

第 46~50 行：如果当前 IR 的操作码为 ASSIGN_N，则调用 GenAssignNAsm 生成相应的目标代码。

第 51~58 行：如果当前 IR 的操作码为 ASM，则将内嵌汇编文本直接插到目标代码中。

第 59~63 行：如果当前 IR 的操作码为 RETV，则调用 GenRetAsm 生成相应的目标代码。

第 64~75 行：根据 IR 的操作码、操作数等情况，生成相应模式串。

第 78~283 行：运用模式串匹配指令模板库，如果检索成功，则根据模板指示生成代码。

第 81 行：清空 ExtReg 向量。值得注意的是，ExtReg 向量的作用就是标识指令模板中的辅助寄存器。事实上，通用的指令模板仍有一定局限，对于较复杂的应用，不得不借助于辅助的寄存器实现。因此，在应用此类指令模板时，也必须关注因辅助寄存器而产生的寄存器溢出。这里，借助于 ExtReg 向量记录当前模板的辅助寄存器。

第 82 行：清空 Forbid 集合。Forbid 集合用于记录指令模板中的“不可用”寄存器。

第 83~87 行：遍历当前指令模板的 SaveReg 集合。SaveReg 集合中记录的正是当前指令模板中“不可用”的寄存器号，也就是说，指定寄存器在模板中有特殊应用。因此，需要调用 SaveRegs 函数，生成代码将指定寄存器的值回存到相应的内存中，如果指定寄存器并



没有绑定变量，则不作处理。

第 88~95 行：根据指令模板的辅助寄存器列表，生成相应的汇编代码。

第 96~166 行：根据操作数 1 生成相应的参数，以便替换指令模板中的“%V.1%”。

第 99~104 行：如果操作数 1 的 `m_bRef` 为真，则必须调用 `OpRef` 函数，并生成用于间接寻址访问的汇编代码。值得注意的是，为了保证安全，这里涉及的寄存器将在当前 IR 翻译过程中不可再分配，因此，也可以视为“不可用”寄存器。

第 107~113 行：指令模板中操作数来源标记为 0，则表示该操作数必须为直接内存寻址。如果操作数存在于寄存器中，则必须将其存到内存中，然后再从内存直接取值。

第 115~131 行：指令模板中操作数来源标记为 1，则表示该操作数必须为寄存器寻址。如果操作数不存在于寄存器中，则必须先将其传输至寄存器，然后再从该寄存器取值。其中，`LoadReg` 函数用于生成将变量值读入寄存器的指令代码。而 `SetVal` 函数用于将变量与寄存器绑定。

第 132~139 行：指令模板中操作数来源标记为 2，则表示该操作数为优先寄存器寻址。如果操作数存在于寄存器中，则优先从寄存器取值，否则从内存取值。

第 144~158 行：如果操作数 1 为常量且操作数来源不为直接内存寻址，则生成代码将操作数 1 的值传送到寄存器。

第 160~164 行：在其他情况下，根据操作数 1 生成参数相应的参数字符串即可。

第 167~236 行：根据操作数 2，生成相应的参数，以便替换指令模板中的“%V.2%”。这部分源码的实现与操作数 1 的处理类似，不再赘述。

第 237~278 行：根据操作数 3，生成相应的参数，以便替换指令模板中的“%V.3%”。

第 242~255 行：如果操作数 3 的 `m_bRef` 为真，则必须调用 `OpRef` 函数，并生成用于间接寻址访问的汇编代码。

第 260~267 行：指令模板中操作数来源标记为 1，则表示需要申请一个寄存器，用于存储运算结果。

第 268~269 行：指令模板中操作数来源标记为 2，则表示将运算结果存放在操作数 2 的寄存器中。因此，这里需要调用 `SetVal` 函数将结果变量与操作数 2 的寄存器绑定。

第 270~271 行：指令模板中操作数来源标记为 3，则表示将运算结果存放在操作数 1 的寄存器中。因此，这里需要调用 `SetVal` 函数将结果变量与操作数 1 的寄存器绑定。

第 279~281 行：将 `ExtReg` 集合中指示的寄存器名也加入 `ParaStr1` 中，供 `IRtoAsmList` 函数使用。

第 282 行：调用 `IRtoAsmList` 函数，根据当前指令模板及传入参数列表 `ParaStr1`，生成相应的目标代码。

第 288~301 行：如果当前过程存在返回类型，则需要处理返回值的传递。根据返回类型的大小，分别应用“`FUNCRET4`”、“`FUNCRET8`”、“`FUNCRETA`”模板进行代码生成。

第 302~305 行：应用“`FUNCEND`”模板生成函数结尾部分的相关代码，主要是运行时栈的维护代码。其中，`ParaStr1` 列表的第 2 个参数是当前过程或函数所属参数的空间大小。

第 307~309 行：应用“`FILEEND`”模板生成汇编文件结尾部分的相关代码。

至此，笔者已经详细介绍了代码生成器的基本结构。本程序的源代码篇幅较长，阅读理

解可能有一定难度，建议读者仔细推敲。

9.4 深入学习

关于代码生成的话题，《The Compiler Design Handbook》是一本非常权威的书。另外，《可变目标 C 编译器——设计与实现》描述的代码生成器也是一个不错的选择，只不过它是基于 DAG 形式的 IR 进行讨论的。

- | | | |
|---------------------------------------------------|---------------------------------------------|-------------------|
| 1、Compilers and Compiler Generators | P. D. Terry | Rhodes University |
| 说明：本书是很不错的编译技术的入门教材，比“紫龙书”简单一些。 | | |
| 2、编译原理（紫龙书） | Alfred V. Aho, Monica S. Lam,
Ravi Sethi | 机械工业出版社 |
| 说明：编译技术的经典著作，不过，对于初学者而言，书中部分内容有一定难度。 | | |
| 3、The Compiler Design Handbook | Y. N. Srikant, Priti Shankar | CRC Press |
| 说明：本书是以优化与代码生成为主的高级手册，详细阐述了基于不同目标机的优化与代码生成技术。 | | |
| 4、编译器工程 | Keith D. Cooper, Linda Torczon | 机械工业出版社 |
| 说明：这是一本以介绍编译器实践工程为主的教材，堪称继“龙书”之后的另一本经典著作。 | | |
| 5、高级编译器设计与实现 | Steven S. Muchnick | 机械工业出版社 |
| 说明：这本书被誉为“鲸书”，其中涉及许多编译器设计中的高级话题，当然，它最值得关注的就是优化技术。 | | |

9.5 实践与思考

1. 请读者尝试使用图着色算法完善 Neo Pascal 的寄存器分配机制。
2. 简述指令调度的意义与基本思想，并举例说明其在 i386 体系结构中的应用。

9.6 大师风采——Peter Naur

Peter Naur: 丹麦著名计算机科学家。1928 年 10 月 25 日出生于丹麦。他于 1949 年从哥本哈根大学获得天文学硕士学位，1957 年获得了天文学博士学位。1959 年，Naur 加盟丹麦第一个计算机公司——Regnecentralen，并且领导了 Algol 60 语言的定义。1969 年，Naur 成为哥本哈根大学的教授，直到 1998 年退休。Naur 是著名的 BNF 范式的发明者之一。

Naur 提出的 BNF 范式为程序设计语言研究及编译技术的发展奠定了基础，他也因此获得了 2005 年图灵奖。除了程序设计语言方面的成果，Naur 在软件工程、算法设计等领域也有杰出的成就。即便如此，Naur 却是一个非常谦虚的人，他反对将他与 BNF 联系在一起，而更愿意将该成果归功于 Backus。

第 10 章 GCC 内核与现代编译技术概述

Some programming languages manage to absorb change, but withstand progress.

—Alan Jay Perlis

10.1 编译技术的现状及发展

在前面的章节中，笔者已经详细阐述了 Neo Pascal 设计与实现的相关话题。通过本书的学习，相信读者对构造一个实际编译器的过程应该有所了解。从实践的角度而言，设计一个完整编译器可能远比学习枯燥的编译原理课程有趣得多。当然，在此过程中，设计者所享受的成就感也不是一门考试成绩所能替代的。由于撰写本书的目的并不在于讨论太多关于编译理论的话题，因此，也无法保证读者通过阅读本书就足以从容应对编译原理的考试。不过，笔者坚信，通过本书的阅读，读者对编译器乃至系统软件的设计与实现一定会有新的认识与提高，这可能是编译原理课程学习无法达到的。在庆幸的同时，新的征程才刚刚开始。对于计算机科学的一个核心领域而言，本书所涉及的话题只是冰山一角而已。

在国内，由于计算机基础学科相对薄弱，人们通常更愿意投身于应用领域的研究，编译技术并不太受到人们的关注，因此，基于这方面的研究成果也相对较少。除了早期一些大型机的 Fortran、Algol 编译器之外，并没有真正的产品级编译器，这不得不说是一种遗憾。本书的目的就是引领更多有志之士踏上编译器设计这片神奇的土地。

有人认为，编译技术已经成熟，继续研究是没有意义的。对此，笔者并不赞同，任何一门学科研究都是永无止境的。在本书剩余的篇幅中，笔者将涉及一些现代编译技术的话题。下面就针对各领域的现状与发展趋势作简单介绍，以便读者了解发展与努力的方向。当然，有些看法纯属笔者个人观点，仅供参考。

在前端技术中，自动生成技术、IR 设计仍将是最炙手可热的话题。

从 20 世纪六七十年代至今，前端自动生成技术一直是人们努力的方向之一，即使目前还只局限于词法分析器与语法分析器领域。以前有人甚至认为完全依赖于计算机实现前端的自动生成是一个可遇而不可求的理想状态，在很多人看来，这种猜测并非完全是无稽之谈。虽然目前无法证明这种观点的正确性，但是，解决这一难题同样是非常困难的。经过了数十年探索之后，试图让计算机自动构造语义分析器的目标仍然很难实现。基于属性文法与语法制导实现前端仍然是绝大多数编译器设计者的选择，这不得不说是一种遗憾。

除此之外，IR 设计也是前端研究的一个热点。关于 IR 的评价，笔者已经在前面详细阐述了。虽然人们并不奢求获得所谓的“最优”IR 形式，但不可否认的是 IR 仍然是设计者津津乐道的话题之一。当然，这种研究还是有价值的，任何基于 IR 的精雕细凿都是值得的。在本章中，笔者将介绍一下 RTL，让读者体会经典 IR 的魅力所在。

在后端技术中，优化、新型编译器实现将是未来的研究热点。

自编译技术诞生之日起，关于优化的研究始终没有停息，当然，这个过程同样将延续至

未来。原因非常简单，除非编译器生成代码的质量能与最高效的手工代码媲美，否则优化技术将是一个永恒的话题。在未来一段时期内，优化技术可能有以下几个主要的发展趋势：

- 1) 过程间分析、别名分析将是两个热点。
- 2) SSA 形式将被越来越多的优化算法采用，其主要优点就是允许原来针对局部的优化算法作用于整个过程。
- 3) 随着并行技术的发展，并行编译的优化技术将是未来十年内一个重要的研究领域。
- 4) 高速缓存优化、数据依赖分析、软流水技术也将取得显著的成果。
- 5) 针对嵌入式目标机的空间优化技术将是值得关注的焦点。

由于本书并非以介绍优化技术为主，因此，笔者不打算在本章中深入讨论有关优化的话题。

其次，新型编译器设计相关的技术也是值得关注的，主要包括并行编译技术、嵌入式编译技术、动态编译技术等研究领域。在本章中，笔者将对动态编译技术及并行编译技术作简单介绍。

10.2 GCC 内核分析

10.2.1 GCC 的基本结构

GCC 的大名，相信已经不必再解释。在本小节中，笔者想针对 GCC 的一些内核实现作简单介绍。事实上，试图详尽解释 GCC 是一项非常庞大的工程，或者说这可能是无法实现的。据笔者所知，除了 GNU 的官方内核白皮书之外，目前尚无一本关于 GCC 内核实现的书籍。不过，对有志于从事编译器设计的专业人员而言，深入学习 GCC 源代码可能是必不可少的。这里，笔者仍然想对 GCC 的内核稍作介绍。

就 GCC 内核层而言，整个编译过程主要应用了三种 IR 形式：GENERIC、GIMPLE、RTL。

GENERIC 是一种前端 IR 形式。由于 GCC 支持的前端多达数十种，而各种语言机制之间是存在差异的，因此，GCC 需要使用一种语言无关的 IR 形式表示前端的输出，这就是 GENERIC。

而 GIMPLE 与 RTL 都是后端 IR 形式，它们更多关注的是优化与代码生成。

事实上，GIMPLE 就是一种较低级的 AST 形式，当然，比普通的 AST 存在更多限制与约束。基于 GIMPLE 实现的优化比较多，如常量传播、冗余删除等。在 GCC 中，基于 GIMPLE 实现的优化也称为“SSA 树优化”。在前面的章节中，并没有提到“SSA”这一名词，实际上，对于初学者而言，SSA 的话题有一定的难度。不过，简单了解一下这个名词的基本含义应该并不困难。关于 SSA 的话题，笔者将在 10.2.4 节中阐述。与 GENERIC 相比，GIMPLE 似乎显得更规范，更适用于生成 RTL。

与 GIMPLE 相比，RTL (register transfer language) 从名字上看，显然是一种更低级的 IR 形式，因为它所关注的是寄存器传输相关的描述。有人认为它就是一种三地址代码，笔者认为这种观点还是有些牵强的，毕竟 RTL 与普通的三地址代码有很大差异。当然，RTL 是一种线性 IR 形式是毋庸置疑的。GCC 的编译过程如图 10-1 所示。值得注意的是，这里 GIMPLE、RTL 等 IR 形式只是一个统称，事实上，它们还存在多种级别的变体，以应对不同的优化及代码生成器的需求。



下面，笔者将从 IR 设计的观点着手，简单介绍一下 GENERIC、GIMPLE 及 RTL 的相关话题，以便读者了解与学习经典编译器的 IR 设计经验。

10.2.2 GENERIC

GENERIC 的设计目标就是将基于源语言书写的代码描述成一种与语言无关的形式，以便针对不同的前端实现都能得到一个相对独立的 IR 形式。这种形式的描述能力必须是充分的，否则等价转换将无从谈起。当然，是否将语句描述成树的形式在学术界是存在争议的。

早期编译器设计中，人们并不提倡将语句描述成树的形式，更多地只是将表达式翻译成 AST 形式。原因主要是由于这种以描述表达式为主的 AST 的实现代价较小。然而，在 GENERIC 中，语句通常被视作没有返回值的表达式，依此即可将其描述成 AST，也就是一个完整的函数将以一个 AST 的形式存在。通常，基于 GCC 前端源语言的扩展都只需实现到 GENERIC 即可，而优化、代码生成都不需要考虑太多。

10.2.3 GIMPLE

从结构上来看，GIMPLE 是 GENERIC 的一种较低级的子集形式。最初，GIMPLE 受到了 McCAT（McGill 大学的一个编译器项目）的 SIMPLE IL 的影响。不过，最好不要将 GIMPLE 理解成一种普通的 AST，因为它已经具备了一些线性 IR 的特性了，例如，它使用临时变量保存表达式的中间结果，这种做法主要应用于三地址码，而 AST 中是很少见的。

GIMPLE 是以函数为单位描述的。通常，并不建议将语言相关的元素引入 GIMPLE 或者后端，也就是说，前端扩展尽可能应用现存的与语言无关的 GENERIC 形式，而避免涉及 GIMPLE 及 RTL 的修改。但是，这并不是绝对的。有时，某些源语言的特性是非常奇特的，编译器不得不将它们传递到目标代码，在这种情况下，语言相关的元素就必须引入 GIMPLE 及 RTL 了。当然，值得注意的是，这种“长距离”传递带来的问题可能是很多的，尤其对于缺乏 GCC 开发经验的设计者而言，这可能是致命的。如果设计者试图将语言相关的特性提供给后端，那么，就必须提供一个 LANG_HOOKS_GIMPLIFY_EXPR 的接口定义。在 GCC 中，虽然 GIMPLE 的存储形式是一种 AST 结构，但它通常是以一种类 C 程序的形式输出的。读者可以使用“-fdump-tree-gimple”参数输出 GIMPLE。下面，先来看一个 GIMPLE 的实例。

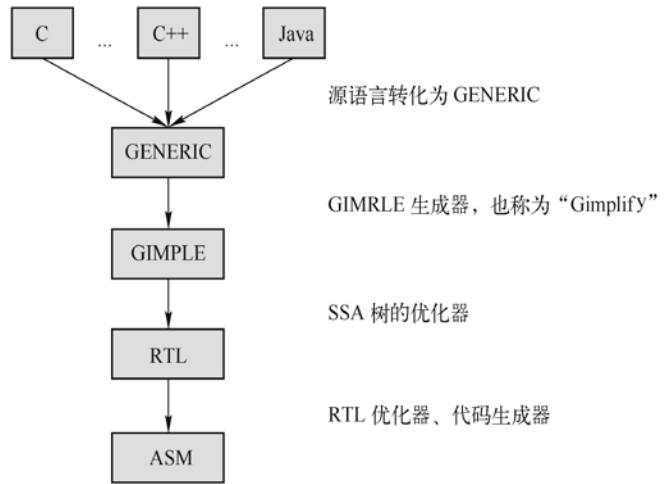
例 10-1 数组最大值求解程序的 GIMPLE 形式。

【源程序】

```
main()
{
```

【GIMPLE】

```
main ()
{
```



```
int nMax=0;
unsigned int Data[10];
int i;
for(i = 0;i < 10;i++)
    if (nMax < Data[i])
        nMax = Data[i];
printf("the max number is %d",nMax);
}
```

```
unsigned int nMax.0;
int i.1;
unsigned int D.2031;
int i.2;
unsigned int D.2033;
int nMax;
unsigned int Data[10];
int i;
```

```
nMax = 0;
i = 0;
goto <D2027>;
<D2026>;
nMax.0 = (unsigned int) nMax;
i.1 = i;
D.2031 = Data[i.1];
if (nMax.0 < D.2031)
{
    i.2 = i;
    D.2033 = Data[i.2];
    nMax = (int) D.2033;
}
else
{

}
i = i + 1;
<D2027>;
if (i <= 9)
{
    goto <D2026>;
}
else
{
    goto <D2028>;
}
<D2028>;
printf (&"the max number is %d"[0], nMax);
}
```

实际上，从本例来看，GIMPLE 的输出形式与先前提到的 C 比较类似。当然，二者也存在不同，C 的程序是符合标准 C 的，而 GIMPLE 的输出只是一种类 C 代码而已。下面，笔者将结合实例详细阐述 GIMPLE 的一些特性。

1. 表达式

GIMPLE 的表达式通常由一个操作符与若干形式简单的操作数组成，这些操作数必须是右值符号或寄存器变量。在 GIMPLE 中，复杂的运算表达式被处理成若干简单的子表达式，



并以临时变量暂存中间结果。这种处理与三地址码的生成比较相似，如例 10-2 所示。

例 10-2 表达式的处理。

【源程序】

```
main()
{
    int i,j,k;
    i = i + j + k;
}
```

【GIMPLE】

```
main ()
{
    int D.2026;
    int i;
    int j;
    int k;

    D.2026 = i + j;
    i = D.2026 + k;
}
```

(1) 复杂对象

这里主要指复杂对象的寻址处理，如数组元素、结构字段等。事实上，在复合左值处理上，GIMPLE 并没有将其中的复合左值转换为“简单地址”形式，如例 10-3 所示。

例 10-3 复杂对象的处理。

【源程序】

```
main()
{
    struct
    {
        int a;
        int b[10];
    }c;
    c.b[c.a] = 1;
}
```

【GIMPLE】

```
main ()
{
    int D.2027;
    typedef struct
    {
        int a;
        int b[10];
    } struct struct
    {
        int a;
        int b[10];
    };
    struct
    {
        int a;
        int b[10];
    } c;

    D.2027 = c.a;
    c.b[D.2027] = 1;
}
```

(2) 条件表达式

条件表达式的处理与 if 语句是一样的，如例 10-4 所示，只是真、假分支的被赋值变量都是相同的。

例 10-4 条件表达式的处理。

【源程序】

```
main()
{
    int i,j,k;
    i = j > 1 + k ? j : k;
}
```

【GIMPLE】

```
main ()
{
    int iftmp.0;
    int D.2027;
    int i;
    int j;
    int k;

    D.2027 = k + 1;
    if (D.2027 < j)
    {
        iftmp.0 = j;
    }
    else
    {
        iftmp.0 = k;
    }
    i = iftmp.0;
}
```

(3) 逻辑表达式

逻辑表达式的处理需要关注短路的实现。不过，在 GIMPLE 中，逻辑表达式并没有使用真、假链翻译技术，而使用了数值翻译方案，也就是通常所说的使用 0、1 表示 true、false 的翻译方案。如例 10-5 所示。

例 10-5 逻辑表达式的处理。

【源程序】

```
main()
{
    int i,j,k;
    i = j && k;
}
```

【GIMPLE】

```
main ()
{
    int iftmp.0;
    int i;
    int j;
    int k;

    if (j == 0)
    {
        goto <D2027>;
    }
    else
    {
        }
    if (k == 0)
```



```

    {
        goto <D2027>;
    }
else
    {

    }
iftmp.0 = 1;
goto <D2028>;
<D2027>;
iftmp.0 = 0;
<D2028>;
i = iftmp.0;
}

```

值得注意的是，当 GIMPLE 的 if 结构缺少假分支时，以空语句替代。

(4) 逗号表达式

例 10-6 逗号表达式将被处理成一系列简单的赋值语句。

【源程序】

```

main()
{
    int i,j,k;
    i = j++, k++;
}

```

【GIMPLE】

```

main ()
{
    int i;
    int j;
    int k;

    i = j;
    j = j + 1;
    k = k + 1;
}

```

2. 语句

对于编译器而言，赋值语句通常被视为赋值表达式，因此，并不是语句翻译所关注的。这里，笔者主要想阐述有关控制结构类语句的处理。

(1) 循环语句

目前，GIMPLE 中的循环语句都将以 goto 跳转的形式描述，如例 10-7 所示。

例 10-7 循环语句的处理。

【源程序】

```

main()
{
    int i,j;
    for(i = 0; i < 10; i++)
        j = j + i;
}

```

【GIMPLE】

```

main ()
{
    int i;
    int j;

    i = 0;
    goto <D2026>;
}

```

```
<D2025>;  
j = j + i;  
i = i + 1;  
<D2026>;  
if (i <= 9)  
{  
    goto <D2025>;  
}  
else  
{  
    goto <D2027>;  
}  
<D2027>;  
}
```

(2) 选择语句

选择语句都是以 `goto` 跳转形式描述的。不过，当条件表达式作用于选择语句时，短路问题是值得注意的，如例 10-8 所示。而例 10-9 则是一个 `switch` 结构的翻译实例。

例 10-8 `if` 语句的处理。

【源程序】

```
main()  
{  
    int i,j;  
    if (i && j || i)  
        i = 1;  
}
```

【GIMPLE】

```
main ()  
{  
    int i;  
    int j;  
  
    if (i == 0)  
    {  
        goto <D2025>;  
    }  
    else  
    {  
  
    }  
    if (j != 0)  
    {  
        goto <D2026>;  
    }  
    else  
    {  
        goto <D2025>;  
    }  
    <D2025>;  
    if (i != 0)  
    {  
        goto <D2026>;  
    }  
}
```



```

else
{
    goto <D2027>;
}
<D2026>;
i = 1;
<D2027>;
}
    
```

在本例中，读者应该特别注意真、假链的翻译机制。由于 Neo Pascal 使用了数值翻译方案处理布尔表达式，因此，没有涉及这个话题。在布尔表达式的翻译中，真、假链是一种非常经典的方案，有兴趣的读者可以参考“龙书”相关章节。不过，这种翻译方案有一定难度，会令一些初学者望而却步。

例 10-9 switch 语句的处理。

【源程序】

```

main()
{
    int i;
    switch (i)
    {
        case 1:i = 1;break;
        case 3:i = 2;break;
        case 2:i = 3;break;
    }
}
    
```

【GIMPLE】

```

main ()
{
    int i;
    switch (i)
    {
        case 1: goto <D2025>;
        case 2: goto <D2028>;
        case 3: goto <D2027>;
        default : goto <D2029>;
    }
    <D2025>;
    i = 1;
    goto <D2026>;
    <D2027>;
    i = 2;
    goto <D2026>;
    <D2028>;
    i = 3;
    goto <D2026>;
    <D2029>;
    <D2026>;
}
    
```

特别注意的是，switch 的跳转表是升序排列的，这样可以避免许多不必要的麻烦。

(3) 跳转语句

跳转语句主要指的就是 goto 及 return，如例 10-10 所示。

例 10-10 跳转语句的处理。

【源程序】

```

main()
{
    
```

【GIMPLE】

```

main ()
{
    
```

```

goto L1;
return 1;
L1:
return 2;
}

int D.2024;
void L1 = <<< error >>>;

goto L1;
D.2024 = 1;
return D.2024;
L1;
D.2024 = 2;
return D.2024;
}

```

(4) 异常处理

在较高级的 GIMPLE 中，异常处理是以 try 语句形式出现的。然而在较低级的 GIMPLE 中，异常处理最终被解释为 goto 跳转。

3. GIMPLE 文法的框架

最后，笔者给出一份摘自《GNU Compiler Collection Internals》的 GIMPLE 文法框架，以便读者对 GIMPLE 的形式有更深刻的理解。

```

function : FUNCTION_DECL
          DECL_SAVED_TREE -> compound-stmt

```

```

compound-stmt: STATEMENT_LIST
              members -> stmt

```

```

stmt : block
      | if-stmt
      | switch-stmt
      | goto-stmt
      | return-stmt
      | resx-stmt
      | label-stmt
      | try-stmt
      | modify-stmt
      | call-stmt

```

```

block : BIND_EXPR
      BIND_EXPR_VARS -> chain of DECLs
      BIND_EXPR_BLOCK -> BLOCK
      BIND_EXPR_BODY -> compound-stmt

```

```

if-stmt : COND_EXPR
        op0 -> condition
        op1 -> compound-stmt
        op2 -> compound-stmt

```

```

switch-stmt : SWITCH_EXPR
            op0 -> val

```




op1 -> NULL
op2 -> TREE_VEC of CASE_LABEL_EXPRs
The CASE_LABEL_EXPRs are sorted by CASE_LOW,
and default is last.

goto-stmt : GOTO_EXPR
op0 -> LABEL_DECL | val

return-stmt : RETURN_EXPR
op0 -> return-value

return-value : NULL
| RESULT_DECL
| MODIFY_EXPR
op0 -> RESULT_DECL
op1 -> lhs

resx-stmt : RESX_EXPR

label-stmt : LABEL_EXPR
op0 -> LABEL_DECL

try-stmt : TRY_CATCH_EXPR
op0 -> compound-stmt
op1 -> handler
| TRY_FINALLY_EXPR
op0 -> compound-stmt
op1 -> compound-stmt

handler : catch-seq
| EH_FILTER_EXPR
| compound-stmt

catch-seq : STATEMENT_LIST
members -> CATCH_EXPR

modify-stmt : MODIFY_EXPR
op0 -> lhs
op1 -> rhs

call-stmt : CALL_EXPR
op0 -> val | OBJ_TYPE_REF
op1 -> call-arg-list

call-arg-list: TREE_LIST
members -> lhs | CONST

```
addr-expr-arg: ID
              | compref

addressable : addr-expr-arg
            | indirectref

with-size-arg: addressable
            | call-stmt

indirectref : INDIRECT_REF
            op0 -> val

lhs        : addressable
            | bitfieldref
            | WITH_SIZE_EXPR
            op0 -> with-size-arg
            op1 -> val

min-lval   : ID
            | indirectref

bitfieldref: BIT_FIELD_REF
            op0 -> inner-compref
            op1 -> CONST
            op2 -> val

compref    : inner-compref
            | TARGET_MEM_REF
            op0 -> ID
            op1 -> val
            op2 -> val
            op3 -> CONST
            op4 -> CONST
            | REALPART_EXPR
            op0 -> inner-compref
            | IMAGPART_EXPR
            op0 -> inner-compref

inner-compref: min-lval
            | COMPONENT_REF
            op0 -> inner-compref
            op1 -> FIELD_DECL
            op2 -> val
            | ARRAY_REF
            op0 -> inner-compref
```



```

        op1 -> val
        op2 -> val
        op3 -> val
    | ARRAY_RANGE_REF
        op0 -> inner-compref
        op1 -> val
        op2 -> val
        op3 -> val
    | VIEW_CONVERT_EXPR
        op0 -> inner-compref

condition : val
    | RELOP
        op0 -> val
        op1 -> val

val : ID
    | invariant ADDR_EXPR
        op0 -> addr-expr-arg
    | CONST

rhs : lhs
    | CONST
    | call-stmt
    | ADDR_EXPR
        op0 -> addr-expr-arg
    | UNOP
        op0 -> val
    | BINOP
        op0 -> val
        op1 -> val
    | RELOP
        op0 -> val
        op1 -> val

| COND_EXPR
    op0 -> condition
    op1 -> val
    op2 -> val
```

10.2.4 SSA

在现代编译技术中，SSA 是一个不得不提及的话题。对于初学者而言，深入讨论 SSA 可能并不太适合。不过，笔者觉得领略一下 SSA 的魅力还是非常有必要的。这里，笔者并不打算讨论基于 SSA 的优化技术，只想结合 GCC 的实现，简单介绍一下 SSA 的基本概念与形式。

1991 年，在 R. Cytron、J. Ferrante、B. Rosen、M. Wegman、K. Zadeck 撰写的《Efficiently

Computing Static Single Assignment Form and Control Dependence Graph》一文中首先提出了 SSA (static single assignment form) 的概念。因此, 与其他 IR 方案相比, SSA 是相对较新的 IR 形式。所谓 SSA 就是指一个函数内每个被赋值的变量只存在一次赋值, 那么, 这个函数满足 SSA 的形式。SSA 的形式有效地将程序中运算的值与它们的存储位置分离, 对数据流分析及某些优化算法的实现是有利的, 例如, 常量传播、值编号、冗余删除、强度削弱等。

事实上, 真正满足 SSA 形式的实例几乎是不存在的, 也就是说, 在实际程序中对一个变量多次赋值的情况是不可避免的。那么, 如何将普通的程序转换成 SSA 的形式呢? 事实上, 当需要对某个变量进行多次赋值时, 编译器会为其生成该变量的新副本。当然, 这些关于同一个变量的副本集合最终仍然会被合并。在 GCC 中, SSA 是基于 GIMPLE 实现的, 可以使用 “-fdump-tree-ssa” 将程序的 ssa 形式输出。下面, 笔者就来举一个 SSA 的实例, 以便读者理解。

例 10-11 SSA 实例分析。

【源程序】

```
main()
{
    int x,y,z;
    x=1;
    if (z>3)
    {
        x=1;
        if (z>2)
            y=x+1;
        else
            goto L1;
    }
    else
        x=2;
L1:
    x=x-3;
    x=4;
    z=x+7;
}
```

【SSA】

```
main ()
{
    int z;
    int y;
    int x;

<bb 2>:
    x_2 = 1;
    if (z_3(D) > 3)
        goto <bb 3>;
    else
        goto <bb 6>;

<bb 3>:
    x_4 = 1;
    if (z_3(D) > 2)
        goto <bb 4>;
    else
        goto <bb 5>;

<bb 4>:
    y_5 = x_4 + 1;
    goto <bb 7> (L1);

<bb 5>:
    goto <bb 7> (L1);

<bb 6>:
    x_6 = 2;
```



```
# x_1 = PHI <x_4(5), x_4(4), x_6(6)>
L1:
x_7 = x_1 + -3;
x_8 = 4;
z_9 = x_8 + 7;
return;
}
```

分析本例后，不难发现，SSA 的转换就是为每一个赋值的变量带上一个下标，并在汇合入口处（例如，标号 L1 处）使用一个 PHI 函数（在书面资料上通常写作“ ϕ ”），以标识对一个变量的多次赋值形式。每个 PHI 函数的参数个数与其所处点的关于该变量的版本个数一致。值得注意的是，本例中所有的被赋值变量只可能存在一次被赋值的情形，这是 SSA 的基本要求。然而，读者不要理解成 SSA 中的所有变量只存在一次被引用的情况，显然，SSA 并没有对引用的次数作限制，在本例中，z_3 就被多次引用了。

SSA 的形式对于优化是非常有利的，其中，最令人满意的是，许多基于 SSA 设计的块内局部优化可以很方便地被移植到过程内使用。当然，笔者对这个话题就不再展开了，有兴趣的读者可以参考《编译器工程》及《高级编译器设计》。

在处理完成后，编译器最终还会将 SSA 的形式还原到原始状态。实际上，这个过程就需要删除 PHI 函数，因为它们只是一个形式上的工具而已，并不具有实际的作用。这里，必须明确一点，SSA 中的程序赋值个数通常会比原始形式的赋值个数多一些，一般是 1.3~3.8 倍左右。不过，SSA 的优化效果较好，因此这种影响并不太明显。

10.2.5 RTL 概述

与 GIMPLE 相比，即将介绍的 RTL 名气要大得多。在 GCC 中，大多数工作都是基于 RTL 完成的。在 GIMPLE 被引入 GCC 之前，所有的优化工作都集中在 RTL 上进行。因此，GCC 内核中最重要的 RTL 是值得读者深入学习的。

RTL 是一种线性结构的 IR，但并不是人们所想象的三地址码形式。RTL 的灵感源于 Lisp 的 lists 结构，它也有两种形式，即内部形式与文本形式。其中，内部形式是以结构、指针等数据结构存储的，主要用于 GCC 内部使用与管理。而文本形式则是专门用于屏幕输出的形式。这与 GIMPLE 是一样的，只不过 GIMPLE 的文本形式是一种类似 C 程序而已。笔者先举一个例子，以便读者对 RTL 有感性的认识。在 GCC 中，读者可以使用“-fdump-rtl-all”参数输出所有的 RTL。注意，参数中的“all”是一个粒度选项，也可以根据实际需要设置，详细的选项说明请参考 GCC 网站。

例 10-12 RTL 实例。

【源程序】

```
main()
{
    int x,y;
    x = x - y;
}
```

【RTL】

```

;; Function main (main)

;; Generating RTL for tree basic block 2

;; x = x - y
(insn 8 6 9 (set (reg:SI 60)
  (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
    (const_int -4 [0xffffffffc])) [0 y+0 S4 A32])) -1 (nil)
  (nil))

(insn 9 8 0 (parallel [
  (set (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
    (const_int -8 [0xffffffff8])) [0 x+0 S4 A32])
    (minus:SI (mem/c/i:SI (plus:SI (reg/f:SI 54
virtual-stack-vars)
      (const_int -8 [0xffffffff8])) [0 x+0 S4 A32])
      (reg:SI 60)))
  (clobber (reg:CC 17 flags))
]) -1 (nil)
  (expr_list:REG_EQUAL (minus:SI (mem/c/i:SI (plus:SI (reg/f:SI 54
virtual-stack-vars)
    (const_int -8 [0xffffffff8])) [0 x+0 S4 A32])
    (mem/c/i:SI (plus:SI (reg/f:SI 54 virtual-stack-vars)
    (const_int -4 [0xffffffffc])) [0 y+0 S4 A32]))
  (nil)))

```

囿于篇幅，笔者并没有给出该源程序的完整 RTL 形式。相对于 GIMPLE 而言，RTL 的形式似乎比较复杂。的确，从形式上而言，即使是文本形式输出的 RTL 也不太直观。当然，主要原因还是初次见面，稍显生疏。相信稍加练习后，这一切将变得非常简单。事实上，对于有志于研究 GCC 内核的读者而言，熟练阅读 RTL 是必不可少的。关于本例 RTL 的含义，暂不作解释。当阅读完本书后，理解本例中的 RTL 应该并不困难。下面，笔者将详细阐述 RTL 的结构与设计。

值得注意的是，与其他 IR 不同，RTL 是一种比较依赖于目标机的设计方案。这是一种非常重要的 IR 设计理念，它对传统的 IR 设计提出了挑战。虽然在过去的二十年中，大多数编译器设计者对于这种观点仍保持谨慎，但始终没有对 RTL 的能力表示怀疑。RTL 在可变目标方面的表现是令人满意的，这恐怕是最有力的例证。

RTL 主要使用了 5 种对象：表达式、整数、宽整数、字符串、向量。

RTL 表达式（简称为“RTX”）是一种类似于 C 语言表达式的结构，RTX 内部形式就是一棵表达式树。而 RTX 的文本形式是以 Lisp 的 lists 输出，实际上，这种形式与数据结构中的广义表的文本形式是一脉相承的。RTX 是 RTL 的核心，也是最复杂的部分。

整数、宽整数也就是普通的整数常量，通常用十进制形式表示。其中，整数就是 C 语言的 int，而宽整数是一种合成对象。



字符串的内核与 C 语言的字符指针是一样的，或者说，RTL 就是参考 `char*` 实现的。不过，RTL 的字符串是不会为空值的。如果试图描述一个空字符串，它将被描述成一个空指针，而不是一个指向空字符的指针，请读者仔细琢磨其中的差异。

向量也就是用于存储指向表达式指针的一维数组，向量通常被书写为 `[exp1 ... expn]`，当然，与数组类似，长度为 0 的向量是不存在的。

完整的 RTL 就是由这 5 种对象组合而成的。实际上，其中最核心的就是 RTX，其他的对象并不复杂。下面，笔者将详细谈谈 RTX 的相关话题。

10.2.6 RTX

RTX 的书写形式如下：

(RTX 代码 [/标志 1.../标志 n] [:机器模式] 操作数 1 ... 操作数 n)

其中，RTX 代码也就是三地址码中的操作码，主要用于描述该 RTX 的功能作用。在 GCC 中，RTX 代码是在 `rtl.def` 文件中定义的，是一组类似于枚举常量的值。在 GCC 中，通常可以使用 `GET_CODE(x)` 和 `PUT_CODE(x)` 宏来获取与修改 RTX 中的代码值。

标志是可选的，主要是作为 RTX 的辅助说明。标志是非常有用的，但又是非常复杂的。

机器模式是可选的，主要描述的就是 RTX 的数据对象的大小。在 RTL 的文本形式中，机器模式紧跟在 RTX 代码之后，其间用冒号隔开。例如，REG 表达式 (`reg:SI 38`)。关于机器模式的详细内容将稍后深入。

操作数通常是一个指向另一个对象的指针，根据 RTX 代码的不同，操作数个数是不同的。这与三地址码是类似的，应该不难理解。

1. RTX 代码的分类

目前，在 `rtl.def` 文件中，将 RTX 代码分成如下几类：

RTX_OBJ：实际的对象，如寄存器 (`reg`)、存储单元 (`mem`、`symbol_ref`)。

RTX_CONST_OBJ：常量对象，包括一些基本转换等，如 `HIGH`、`ADDRESSOF` 等。

RTX_COMPARE：不满足交换律的比较，包括 `LE`、`LT`、`GE`、`GT`、`LEU`、`LTU`、`GEU`、`GTU`。

RTX_COMM_COMPARE：满足交换律的比较，包括 `EQ`、`NE`、`ORDERED` 等。

RTX_UNARY：单目操作，如 `NEG`、`NOT`、`ABS`、值扩展、整型及浮点型转换。

RTX_COMM_ARITH：满足交换律的双目操作，如 `PLUS`、`AND` 等。

RTX_BIN_ARITH：不满足交换律的双目操作，如 `MINUS`、`DIV`、`ASHIFTRT` 等。

RTX_BITFIELD_OPS：位域操作，通常有三个操作数。

RTX_TERNARY：其他的三目操作，目前只支持 `IN_THEN_ELSE`、`VEC_MERGE`。

RTX_INSN：表示一个完整的指令，如 `INSN`、`JUMP_INSN`、`CALL_INSN` 等。

RTX_MATCH：表示指令内的匹配对象，如 `MATCH_DUP`。通常只存在于机器描述中。

RTX_AUTOINC：表示自增的寻址模式，如 `POST_INC`。

RTX_EXTRA：所有其他的 RTX 代码，包括一些用于机器描述的代码（如 `DEFINE_*`）、所有描述副作用的代码（如 `SET`、`USE`）、出现在指令中的非指令代码（如 `NOTE`、`CODE_LABEL`）。

2. RTX 格式描述

与三地址码类似，不同的 RTX 代码对其操作数是有一定限制的。在 GCC 中，对此有一套相对完整的描述体系。例如，在 `rtl.def` 中，PLUS 的格式描述如下：

```
DEF_RTL_EXPR(PLUS, "plus", "ee", RTX_COMM_ARITH)
```

其中，第 1 项表示 RTX 代码，第 2 项是文本输出形式，第 4 项是所属的类别，而第 3 项就是用于描述格式的。在 GCC 中，格式描述代码“e”就表示操作数可以是一个表达式，而两个“e”就表示该 RTX 可以且必须有两个表达式操作数。

下面，就来看看常用的 RTX 格式描述代码的含义：

e：表达式（实际上为指向表达式的指针）。

i：整数。

w：宽整数。

s：字符串。

E：表达式向量。

u：除了在调试信息中不同，“u”等价于“e”，表示指向指令的指针。

n：除了在调试信息中不同，“n”等价于“i”，表示 `note` 指令行号或代码编号。

S：说明一个可选字符串。在内部表示的 RTL 对象里，“S”等价于“s”。但当从机器描述文件中读出一个对象时，这个操作数的值可能被省略，被省略的字符串被认为是空串。

V：说明一个可选向量。在内部表示的 RTL 对象里，它等价于“E”。但当从机器描述文件中读出一个对象时，其值可能被省略，被省略的向量相当于没有元素的向量。

B：一个指向基本块结构的指针。

0：一个内容不适合任何正常类型的跟踪形式。“0”跟踪信息在 RTL 调试输出中不显示，通常在编译器中有某些特殊作用。

在 GCC 中，可以使用以下几个宏来获取一些与 RTX 代码相关的信息。

GET_RTX_LENGTH(code)：获取给定 RTX 代码的操作数个数。

GET_RTX_FORMAT(code)：获取给定 RTX 代码的格式字符串。

GET_RTX_CLASS(code)：获取给定 RTX 代码所属的类别的缩写形式（一个字符表示）。

关于各个 RTX 代码的格式描述就不再详细列举了，读者可以参考 `rtl.def` 文件。

3. RTX 机器模式

机器模式主要用于描述数据对象的大小及其表示。从实现上而言，机器模式表示成枚举类型 `enum machine_mode`，这个类型的声明在 `machmode.def` 文件中。例如：

```
INT_MODE(QI, 1);
```

这是一个单字节整型模式的描述。“QI”的含义就是“Quarter-Integer”。在 GCC 的整型模式中，通常是以 4 字节 `int` 作为基准进行描述。例如，“HI”即表示“Half-Integer”，也就是两字节型。而第 2 项“1”则表示其所需的字节数，这是依赖于 `INT_MODE` 宏的定义。在 `machmode.def` 文件中，关于 `INT_MODE` 宏的描述为“`INT_MODE (MODE, BYTESIZE);`”。再如：

```
FLOAT_MODE(SF, 4, ieee_single_format);
```




```
FLOAT_MODE (DF, 8, icee_double_format);
```

这两个模式分别是单精度浮点型及双精度浮点型的描述，其中，第 3 项所描述的就是它们所采用的格式标准。在 `machmode.def` 文件中，关于 `FLOAT_MODE` 宏的描述为“`FLOAT_MODE (MODE, BYTESIZE, FORMAT);`”。

根据 `machmode.def` 文件的注释，可以非常容易地理解机器模式相关声明的含义。这里，笔者不再举例说明了。在 RTL 中，机器模式是可选的，却是非常重要的。参考例 10-12 就不难发现，在 RTL 中，机器模式的使用是极其频繁的。值得注意的是，由于 GCC 的目标机不局限于 i386，还包括向量机、嵌入式计算机等，因此，并非所有模式在 i386 中都是支持的。

4. RTX 的标志

一个 RTX 通常可以包含若干标志，应用于一些特殊的表达式中，对其进行辅助说明。在 RTL 内核中，标志是以位形式存储的，而访问则通过 GCC 定义的宏完成。标志是比较繁琐的话题，且同一标志在不同的表达式中的含义是不同的。在 RTX 文本模式中，标志是紧接在 RTX 后面书写的，通常以“/”开头。下面，笔者将详细说明各标志的含义。

(1) `call`。在 `mem` 中，置 1 表示内存引用不受限。在 RTL 文本模式中，记作“/c”。

(2) `frame_related`。在 `insn` 或 `set` 中，置 1 表示它是一个函数的首部，并需要设置栈指针。包括重置栈的指针、保护现场寄存器等。在介绍运行时环境时，笔者已经详细阐述了相关理论与实现。

在 `symbol_ref` 表达式中，置 1 表示引用的地址是该函数的字符串常量池。

在 `mem` 表达式中，置 1 表示引用一个标准对象。

在 RTL 文本模式中，记作“/f”。

(3) `in_struct`。在 `mem` 表达式中，置 1 表示引用的数据对象依赖于一个数组或结构，置 0 表示引用的数据对象依赖于一个标准类型的变量。这个标准对别名分析是非常有用的，它有助于确定别名引用的关系。

在 `reg` 表达式中，置 1 表示该寄存器的整个生存期被包含在循环条件表达式中。

在 `subreg` 表达式中，置 1 表示此 `subreg` 访问的对象的方式是由一个宽方式提升而来。

在 `label_ref` 表达式中，置 1 表示所引用的标号的定义点位于包含此标号引用的最内层循环之外。

在 `code_label` 表达式中，置 1 表示该标号不会被删除。这主要用于描述没有被 `goto` 引用的标号。这种标号不一定是冗余的，有可能是分支优化造成的。

在 RTL 文本模式中，记作“/s”。

(4) `integrated`。在 `insn`、`insn_list`、`const` 中，置 1 表示该 RTL 是由内联函数产生的。

在 `reg` 表达式中，置 1 表示此寄存器包含当前函数的返回值。对于用寄存器传参的机器而言，当相同的寄存器被用于传参时，不设置该标志。

在 `symbol_ref` 表达式中，置 1 表示对该符号弱引用。这里，笔者简单介绍一下“弱引用 (weak reference)”的概念。弱引用与强引用主要的区别是在垃圾回收机制中体现的。在 Java、.Net 之类的语言中，被弱引用的对象仍然可以被垃圾回收。弱引用机制对有效利用缓存资源是积极的，但是弱引用有时也是危险的。

在 RTL 文本模式中，记作 “/i”。

(5) jump。在 mem 表达式中，置 1 表示当访问一个对象时，该 mem 的别名集合不变。

在 set 表达式中，置 1 表示是一个返回。

在 call_insn 中，置 1 表示一个同属调用。这里，笔者简单介绍一下“同属调用”的概念。实际上，这是一个编译优化的情形，如下所示：

```
int aa(int x,int y)
{
...
return(bb(x+1,y));
}
```

在这个程序中，如果 return 之前的语句不会产生任何副作用，那么，编译器可以将对 aa 的调用直接处理成对 bb 的调用。这样，编译器不必再为 aa 分配帧空间，而只需为 bb 分配帧空间。同属调用优化是一种比较高级的优化技术，而且算法实现也较复杂。

在 RTL 调试输出中，此标志写成 “/j”。

(6) unchanging。在 reg、mem 表达式中，置 1 表示表达式的值不变化。

在 subreg 表达式中，置 1 表示 subreg 访问的无符号对象的方式是由一个宽方式提升而来。

在 insn、jump_insn 中，置 1 表示一个废弃的分支。

在 symbol_ref 表达式中，置 1 表示此符号引用函数常数池中的地址。

在 call_insn、note、expr_list 中，置 1 表示一个常数和纯函数的调用。

在 RTL 调试输出中，此标志写成 “/u”。

(7) used。通常，used 只临时用在一个函数的 RTL 生成完成之时，用于计算一个表达式在指令中出现的次数。在 GCC 中，对于多次出现的表达式，根据结构共享规则将被复制。

在 symbol_ref 表达式中，置 1 表示符号的外部说明已输出。简单地说，就是保证外部说明只输出一次。

在 reg 表达式中，用于重新编码叶子寄存器，以保证每个寄存器仅被重新编码一次。

(8) volatile。在 mem、asm_operands 或 asm_input 表达式中，若内存引用是易变 (volatile) 的，则易变内存引用不能被删除、归并或重新排序。

在 symbol_ref 表达式中，它是与具体机器相关的标志。

在 reg 表达式中，置 1 表示用户级的变量值。置 0 表示编译器内部临时变量。

在 insn 中，置 1 表示此指令被删除。

在 label_ref、reg_label 表达式中，置 1 表示引用一个非局部标号。

在 RTL 调试输出中，写成 “/v”。

5. RTX 代码

前面，笔者已经详细讲述了 RTX 结构相关的话题，包括机器模式、标志等。下面将关注一下 RTX 的代码。这是学习与理解 RTL 的关键，也是体会经典编译器设计的契机。

表 10-1 列出了绝大部分常用的 RTX 代码，供读者参考。

表 10-1 常用的 RTX 代码

RTX 表达式形式	功能描述
(const_int i)	表示一个整型值 i
(const_double:m addr i0 i1)	表示一个浮点型常量或一个大整型常量
(const_fixed:m addr)	表示一个定点型常量
(const_vector:m [x0 x1...])	表示一个向量常量
(const_string str)	表示一个字符串常量
(symbol_ref:mode symbol)	表示一个数据段中的汇编标号的值
(label_ref:mode label)	表示一个代码段中的汇编标号的值
(const:m exp)	表示一个汇编阶段可计算的常量
(high:m exp)	表示一个表达式的高 n 位
(reg:m n)	当 n < FIRST_PSEUDO_REGISTER 时, 表示引用一个物理寄存器 当 n < FIRST_PSEUDO_REGISTER 时, 表示引用一个逻辑寄存器
(subreg:m reg bytenum)	表示引用一个机器模式下的寄存器
(scratch:m)	根据当前指令的需要, 选取一个寄存器
(mem:m addr alias)	表示引用一个主存单元
(addressof:m reg)	获取一个寄存器的地址。当然, 这里主要指逻辑寄存器
(concatm rtx rtx)	将两个 RTX 连结, 主要用于声明、代码生成相关的 RTL 中, 不能出现在指令链中
(concatnm [rtx...])	将多个 RTX 连结, 主要用于声明中, 不能出现在指令链中
(plus:m x y) (ss_plus:m x y) (us_plus:m x y)	两个表达式相加。主要的差异在于: plus 将值的宽度置于 m 中, 而 ss_plus、us_plus 分别将最大的有符号值、无符号值置于 m 中
(lo_sum:m x y)	将 x 与 y 的低 n 位相加
(minus:m x y) (ss_minus:m x y) (us_minus:m x y)	两个表达式相减
(compare:m x y)	两个表达式的值比较
(neg:m x) (ss_neg:m x) (us_neg:m x)	将表达式的值取负
(mult:m x y) (ss_mult:m x y) (us_mult:m x y)	两个表达式的值相乘
(div:m x y) (ss_div:m x y)	两个有符号表达式的值相除
(udiv:m x y) (us_div:m x y)	两个无符号表达式的值相除
(mod:m x y) (umod:m x y)	两个表达式的值取模
(umin:m x y) (umax:m x y)	比较两个无符号表达式的值, 返回较大(小)的值
(smin:m x y) (smax:m x y)	比较两个有符号表达式的值, 返回较大(小)的值
(not:m x)	对表达式的值取逻辑非
(and:m x y)	对两个表达式的值取逻辑与
(ior:m x y)	对两个表达式的值取逻辑或
(xor:m x y)	对两个表达式的值取异或
(ashift:m x c) (ss_ashift:m x c) (us_ashift:m x c)	对表达式的值算术左移, 移位位数取决于 c
(lshiftrt:m x c)	对表达式的值逻辑右移, 移位位数取决于 c

(续)

RTX 表达式形式	功能描述
(ashifrt:m x c)	对表达式的值算术右移, 移位位数取决于 c
(rotate:m x c)	对表达式的值循环左移, 移位位数取决于 c
(rotatert:m x c)	对表达式的值循环右移, 移位位数取决于 c
(abs:m x)	对表达式的值取绝对值
(sqrt:m x)	对表达式的值取平方根
(ffs:m x)	对表达式 x 的最低有效位加 1
(clz:m x)	统计 x 首部置 0 位的个数
(ctz:m x)	统计 x 尾部置 0 位的个数
(popcount:m x)	统计 x 中置 1 位的个数
(parity:m x)	统计 x 中置 1 位的个数, 并将统计结果模 2
(bswap:m x)	将 x 按字节翻转
(eq:m x y)	判断两个表达式的值是否相等
(ne:m x y)	判断两个表达式的值是否不等
(gt:m x y)	判断 x 是否大于 y (有符号)
(gtu:m x y)	判断 x 是否大于 y (无符号)
(lt:m x y)	判断 x 是否小于 y (有符号)
(ltu:m x y)	判断 x 是否小于 y (无符号)
(ge:m x y)	判断 x 是否大于或等于 y (有符号)
(geu:m x y)	判断 x 是否大于或等于 y (无符号)
(le:m x y)	判断 x 是否小于或等于 y (有符号)
(leu:m x y)	判断 x 是否小于或等于 y (无符号)
(if_then_else cond then else)	条件表达式, 与 C 语言的“?:”类似
(cond [test1 value1...] default)	多路条件表达式, 与 switch 结构类似, 但这并不是 switch 的翻译方案
(sign_extract:m loc size pos)	引用一个按符号位扩展的位域, loc 表示寄存器或字节地址, size 表示位域大小, pos 表示位偏移
(zero_extract:m loc size pos)	引用一个按 0 扩展的位域, loc 表示寄存器或字节地址, size 表示位域大小, pos 表示位偏移
(vec_merge:m vec1 vec2 items)	合并两个向量。哪些元素合并取决于 items
(vec_select:m vec1 section)	选择部分向量元素
(vec_concat:m vec1 vec2)	连接两个向量
(vec_duplicate:m vec)	将一个小向量以元素复制的形式转换为一个大向量
(sign_extend:m x)	将 x 按符号扩展
(zero_extend:m x)	将 x 按 0 扩展
(float_extend:m x)	浮点数扩展
(truncate:m x) (ss_truncate:m x) (us_truncate:m x) (float_truncate:m x)	截尾转换
(float:m x)	定点数转换为有符号浮点数
(unsigned_float:m x)	定点数转换为无符号浮点数
(fix:m x)	浮点数转换为有符号定点数
(unsigned_fix:m x)	浮点数转换为无符号定点数



(续)

RTX 表达式形式	功能描述
(fract_convert:m x)	将定点数、有符号整数、浮点数转换为指定机器模式的浮点数。当发生溢出时，结果未定义
(sat_fract:m x)	将定点数、有符号整数、浮点数转换为指定机器模式的浮点数。当发生溢出时，结果取最大或最小值
(unsigned_fract_convert:m x)	将定点数、无符号整数、浮点数转换为指定机器模式的浮点数。当发生溢出时，结果未定义
(unsigned_sat_fract:m x)	将定点数、无符号整数、浮点数转换为指定机器模式的浮点数。当发生溢出时，结果取最大或最小值
(set lval x)	将 x 的值置入 lval 中
(return)	函数返回
(call function nargs)	函数调用，function 是一个 mem 表达式，nargs 用于传参
(clobber x)	存储或者可能存储一个不确定的值到 x 中
(use x)	使用 x 的值
(cond_exec [cond expr])	有条件执行表达式。当 cond 非零时，expr 执行，否则不执行
(sequence [insns...])	指令序列
(asm_input s)	内嵌汇编

6. insn 序列

在 GCC 中，一个函数的 RTL 是以一个双向链表结构存储的，称为“insn 序列”。而具体的 insn 结构基本上就是类似于一条指令，RTL 是嵌套，但位于“顶层”的 RTL 只有 insn，它们以空行隔开。有些 insn 表示可执行的指令，有些表示 switch 语句的分支列表，有些则表示变量的声明。

除了拥有具体的数据之外，每条 insn 都有唯一的编号。通常，用户可以通过一些预定义的宏来获取指定编号的 insn 及其前驱、后继 insn。在 GCC 中，insn 可以分为 6 类：

insn: 用于描述非跳转及函数调用的指令。表达式序列通常是包含在 insn 中的。

jump_insn: 用于描述跳转的指令，通常与 label_ref 表达式一起使用。

call_insn: 用于描述函数调用的指令。

code_label: 用于描述标号的指令。

note: 用于描述一些调试及声明信息。

barrier: 置于指令序列中，使控制流不能通过的指令，如 exit 等。

10.3 动态编译技术简介

10.3.1 动态编译技术基础

动态编译 (dynamic compilation) 技术的发展历史不过短短的二十几年。相信谁也没有想到当年籍籍无名的动态编译会成为今天最热门的研究领域之一。目前，有关动态编译技术的讨论也仅限于论文资源，尚无书籍涉及此类话题。

实际上，动态编译技术并不神秘，其最初的应用领域就是程序移植。在 20 世纪 80 年代末，随着 CPU 产业竞争的日益激烈，许多 CPU 开发商都以兼容其他经典 CPU 指令集为卖

点，大力推广自己的产品。除了某些 CPU 的确是结构兼容之外，大部分 CPU 都很难做到这一点。当然，技术并不是唯一的原因，商用因素也是不容忽视的。那么，所谓的“兼容”又是如何做到的呢？事实上，通常的做法就是指令转换，即把用其他 CPU 指令集描述的程序动态转换成用自身 CPU 指令集描述的形式。这种转换有些是硬件实现的，有些则是软件实现的。硬件实现的转换通常会设计一组专用的处理器凌驾于主 CPU 之上，完成指令的实时转换。而软件实现的转换则是将转换程序烧录到 BIOS 中，在装入用户程序之前，完成动态转换。当然，无论是哪种实现技术，可以肯定的是它们都比操作系统更低级。值得注意的是，在最初的设计方案中，这种转换并不一定是动态的，也可能是静态的。不过，由于应用领域的特殊性，动态转换比静态转换的优势明显得多。在现代编译技术中，通常将这类转换技术视作一种特殊的编译方式，只不过其源程序与目标程序都是二进制形式的，因此，习惯上将其称为“二进制翻译”。二进制翻译是动态编译技术中一个非常重要的领域，稍后将作相关介绍。下面先介绍动态编译技术的一些基本概念。

所谓的“动态编译器”就是指将部分编译工作延后至运行时进行的编译器。而动态编译技术研究的就是如何花费较少的时间，以获得更优化的目标代码。因此，在一定程度上，对于动态编译器而言，编译时间可能比代码质量更重要。这完全打破了静态优化技术的传统观念，许多耗时费力的分析算法是很难被动态编译器接受的。说到这里，读者应该树立一个观点，那就是动态编译通常并不苛求一次编译完成，将整个编译过程分成多次进行是可以接受的。

目前，动态编译技术的主要应用领域有如下几个：

(1) 动态语言的实现。例如，Lisp、Perl、Python、Smalltalk、Ruby 等都属于动态语言。这类语言的虚拟机或者运行环境通常就是一个动态编译器，它们的工作就是在程序运行过程中，实现将低级 IR 描述的可执行程序动态编译成二进制形式。当然，这个过程对于用户是完全透明的。

(2) 动态二进制翻译。这类应用需求与动态编译的最初原型比较相似。不过，在今天的硬件体系中，实现转换已经不再是终极目标了，如何得到更优的目标代码是人们所关注的。这种技术在目标机移植方面是非常有用的，也是 CPU 设计厂商重点关注的技术。目前，Intel、IBM、HP 等知名公司都正在致力于该领域的研究。

(3) 动态优化技术。编译优化通常是指静态优化，不过，有些特殊的代码情景是静态优化无能为力的，笔者稍后举例。在这种情况下，如果试图进一步优化，那么，动态优化将是唯一的选择。

目前，关于动态编译的技术资源并不太多，仅限于一些论文研究。因此，本书也不打算深入探讨动态编译的相关理论话题，只想结合几个经典的研究实例，阐述相关的基本概念，以便读者对动态编译有一定的认识。

10.3.2 运行时特定化

特定化 (specialization) 是一种依据程序及上下文环境完成的程序变换技术。根据执行时刻的不同，程序特定化又可以分为编译时特定化和运行时特定化。其中，编译时特定化是静态优化的话题，暂不作讨论。这里，只关注运行时特定化的相关技术。

运行时特定化 (run-time specialization) 指的就是该特定化的工作是在程序运行过程中完

成的。那么，研究运行时特定化的意义是什么呢？事实上，其意义是非常深远的，这里简单提两点：

(1) 程序封装技术限制了静态编译器的分析、优化范围，使得编译器无法得到关于程序整体的结构信息。例如，用户程序依赖于某个二进制共享库。那么，在静态编译中，这个共享库对于编译器是一个黑盒。对其进行任何数据流、控制流分析都是徒劳的，因此，这种情况对于优化是非常不利的。即使是过程间优化对此也是束手无策的。

(2) 面向对象技术的动态绑定机制也为静态分析带来了不便。

在现代编译器设计中，类似的应用需求将日益增多。这里，笔者举一个简单的实例，如表 10-2 所示。

表 10-2 动态优化示例

(a)	(b)
<pre>extern int aa(); main() { printf("%d",aa()); }</pre>	<pre>int aa() { return 10; }</pre>

如果(a)、(b)两个函数是在同一个项目中，那么，应用过程间优化就很容易实现常量传播。但如果(b)存在于一个二进制的共享库（注意，不是 obj 文件）中，任何静态优化都将无能为力。当然，在不苛求的情况下，即使不进行优化也是可以接受的。不过，在动态编译技术中，这类程序通常可以得到更优的效果。在运行过程中，动态分析 aa()函数的返回值是完全可能的，因此，进一步的常量传播是有意义的。而运行时特定化的工作就是尽可能分析得到这种特殊值的集合，以便产生更优的目标代码。

在运行时特定化领域中，UW Dynamic Compilation（缩写为“DyC”）是一个比较经典的实例，它是由华盛顿大学动态编译器小组设计与实现的。下面，简单介绍一下 DyC。

DyC 是一个基于运行时特定化实现的动态 C 编译器，它借助于一些用户标记实现运行时的特定化。DyC 包括一个静态编译组件和一个动态编译组件，结构如图 10-2 所示。

其中，静态编译组件根据输入源程序生成三个输出部分：机器码模板、设置代码、指导命令。机器码模板中包含了一些空洞，以回填存储运行时计算得到的常量值。而设置代码则用于计算运行时常量的值。指导命令则用于指导如何应用模板及设置代码生成可执行程序。动态编译组件则遵循指导命令来复制机器码模板，并将常量值填充到模板的空洞中。

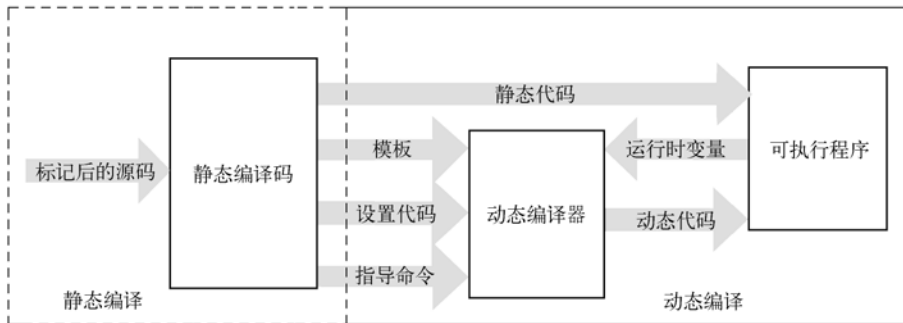


图 10-2 DyC 的总体结构

不过，值得注意的是，动态优化是不能也不可能代替静态优化的。由于动态优化对时间的要求比较苛刻，因此，设计者通常会在静态编译时尽可能提高代码的质量，以减轻动态优化的工作。事实上，动态优化是在静态优化的基础上追求更完美的过程，而不是将动态优化变成编译器的主体优化。在 DyC 中，静态编译组件会基于机器模板进行大量的静态全局优化，这些优化对于运行时特定化是相当有用的。

在 DyC 中，静态编译组件只编译未被用户标识为动态区域（dynamic region）的函数，而对于动态区域，静态编译组件的处理如下：

(1) 标识动态区域中的变量、表达式在运行时刻将是常量。在算法实现中，DyC 是基于 SSA 形式进行数据流分析的。DyC 采用了两遍数据流分析，第 1 遍用于标识运行时刻的常量，而第 2 遍则用于分析运行时刻常量的到达定值信息。

(2) 将每个动态区域划分为设置子图和模板代码子图，并替换区域原始的子图。

(3) 应用标准的优化算法进行控制流相关优化。

(4) 生成机器代码及指导命令。

不过，手工插入指导性的注释来标识程序是一项麻烦的工作，Calpa 系统解决了这个问题，实现了自动注释的功能。它结合了 profile 信息和程序分析来自动获得注释。关于 DyC 的更多资源，读者可以访问<http://www.cs.washington.edu/research/dyncomp/>。

10.3.3 动态二进制翻译

二进制翻译技术，也称为“BT (Binary Translation)”，是一种即时编译技术，它的工作是将源体系结构编译生成的二进制代码动态翻译为可以在目标体系结构上运行的代码。动态二进制翻译及优化系统使得程序可以在无需重新编译的情况下进行目标机移植，对丰富新兴目标机的软件系统是有积极意义的。目前，比较著名的二进制翻译系统包括 Intel 的 IA-32 Execution Layer、IBM 的 DAISY、Transmeta 的 CMS 等。该技术也被应用于虚拟机中，如著名的 VMware。下面，先来看看动态二进制翻译系统的基本结构，如图 10-3 所示。

结合图 10-3，笔者简单介绍一下动态二进制翻译的过程，可以分为如下几步：

(1) 程序加载。当用户执行一个源目标机程序时，目标机平台将调用加载器装入程序。注意，这个加载器可能是建立在操作系统上的，也可能是烧录在 BIOS 中的，甚至是依赖于硬件实现的。当然，如果操作系统已经实现了目标机移植，那么，将二进制翻译建立在操作系统之上也是完全可行的。在这种情况下，操作系统会将该程序的控制权完全交给二进制翻译系统。

(2) 解释执行。由 BT 控制器启动解释器进行解释执行。与普通的解释器不同，在这个过程中，解释器需要收集源程序的相关信息，以便进行动态翻译与优化。根据程序执行的 90-10 定律，程序执行的“热点”是解释器必须关注的。当然，收集工作有时也可以由硬件完成，一些现代 CPU 提供了比较丰富的状态寄存器，以便翻译器动态分析程序执行的过程。

(3) 提取程序片段。“热点”是动态编译与优化的主要工作对象，因此，提取与分析“热点”片段是非常重要的。

(4) 动态翻译程序片段。将“热点”翻译成本地代码执行是动态翻译器的任务，在此过

程中，动态翻译器将对“热点”片段进行一些动态优化。翻译生成的本地代码片段将被保存在 cache 中，以便下次调用该程序片段时，直接执行本地代码片段，以提高程序执行效率。这里的 cache 是指动态编译器拥有的一组存储资源，并不一定是硬件体系中的 cache。在实际系统中，cache 通常被分为“block cache”与“trace cache”。值得注意的是，cache 一般不会太大，因为动态翻译器与用户争夺存储资源是不能接受的。因此，就需要考虑 cache 管理的问题了。与操作系统页面管理类似，cache 管理也存在一个淘汰机制。事实上，哪些片段应该被淘汰有时是很难评估的。当然，有些动态翻译器也会将本地代码片段永久存放在外存上，在外存允许的情况下，这种方案可能更好。

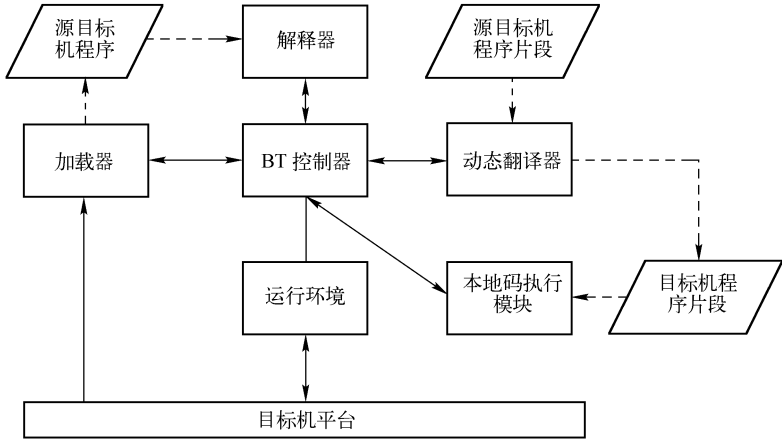


图 10-3 动态二进制翻译系统的基本结构

这里，笔者简单谈谈动态二进制翻译的几个关键问题：

- (1) 存储系统。根据目标机的结构，二进制翻译通常会涉及寄存器、cache、主存等存储系统。对于不同目标机之间的映射关系是设计者需要考虑的。当前，寄存器、cache 相对复杂一些。在使用寄存器时，必须兼顾专用寄存器、状态寄存器、标志寄存器等。而 cache 管理中，更多关注的是 cache 的命中率。在获取 cache 命中率的信息方面，动态编译器比静态编译器有利得多。在一些现代体系结构的目标机中，甚至还设置了一些用于记录 cache 命中率、cache 失效率、程序热点等的寄存器组，以便应用程序能更高效地获取相关信息。这类信息对于动态优化是极其重要的，在此之前，动态编译器必须依赖于软件方式记录。在主存方面，数据映射是一个问题。如 x86 平台采用的是小尾端的形式，而 MIPS 既支持小尾端，也支持大尾端，而另外一些平台可能只支持大尾端。
- (2) 实时性问题。正如前面所说的，实时性对于二进制翻译是极其重要的，因此，在某些情况下，实时性很强的代码需要使用启发式的算法进行提前预测，并进行相关翻译生成。
- (3) 空间耗费。动态编译器本身对于存储资源的耗费也是关键的问题。由于动态编译器与用户程序是运行在同一时刻的，因此，动态编译器对存储资源的耗费可能会对用户程序的运行产生一定的影响。当然，这也许是不可避免的，但并不意味着是无限制的。无论是优化还是代码生成，动态编译器设计者都必须时时关注这个指标，绝不可能像静态编译器那样不计成本地耗用户的存储空间。
- (4) 效率问题。在动态二进制翻译中，研究代码的重用率是提高翻译效率的有效方法。

理想情况下，如果一次翻译生成的代码被执行的次数较多，那么，就可以在在一定程度上降低优化及生成代码的代价。因此，有效地利用 cache 管理代码片段是非常必要的。不过，当缓存被填满时，代码片段的淘汰将是一个复杂的问题。与操作系统页面淘汰问题类似，试图寻找一个优化的方案并不容易。

(5) 体系结构。不同结构目标机之间的差异通常是非常大的，可能还不仅局限于存储系统、指令系统方面。有时，中断、流水线等也是值得关注的。这些因素对指令调度、代码生成等都是有一定影响的。在笔者看来，硬件结构的问题并不能依赖于想象，只有实践证明过的才是可靠的。

在动态二进制翻译领域中，Queensland 大学的 UQDBT 是一个不错的多源多目标的研究系统。在国内，动态二进制翻译主要应用于“龙芯”的平台移植，由中科院软件所研发的 DigitalBridge 就是在 Linux/MIPS 平台上动态翻译执行 Linux/x86 的程序。与先前讨论的模型不同，DigitalBridge 并不解释执行源程序，而是直接翻译生成未优化的本地代码片段，将其存放于 block cache 中。然后，在执行本地代码片段的过程中，分析程序“热路径”，再进行优化并生成优化后的片段，将其存放于 trace cache 中。这种处理方法在二进制翻译中的应用很广泛，主要原因就是解释器的执行效率通常不能令人满意。

10.4 并行编译技术简介

10.4.1 并行编译技术基础

高性能体系结构的发展对传统编译器提出了挑战。在过去的二十多年里，并行计算机的发展是惊人的。如今，在高性能计算领域，并行计算机的应用非常广泛。当然，基于并行计算机的软件系统的研发也是极其重要的，如并行操作系统、并行编译器等。相对于动态编译技术而言，并行编译的历史稍久远些。并行编译系统是并行计算机系统软件中的一个十分重要的部分，也是现代编译技术领域的一个研究热点。

在 20 世纪 70 年代，并行体系结构发展飞快，在不同层次上，硬件以不同方式实现了多种并行处理机制，因此，如何充分利用硬件的并行处理能力是对软件设计提出的新要求。在并行系统软件领域，在过去的 20 年中，人们对并行编译技术的研究热情超过了并行操作系统。熟悉并行计算的读者应该明白基于并行计算机进行程序设计是一项艰巨的任务，需要考虑与处理的细节是非常复杂的。因此，人们更希望像在串行机上编写高级语言程序一样完成并行程序设计，这就是并行编译器设计的源动力。

1. 并行编译系统的基本结构

一个完整的并行编译系统包括并行化工具、并行编译器、并行运行库等。其基本结构如图 10-4 所示。

并行化工具的主要功能就是将串行程序并行化，它可以独立于并行编译器存在，也可以嵌入在并行编译器中。

并行编译器包括预处理器、前端、主处理器、后端 4 个部分。预处理器的输入是并行化后的源程序，其功能是根据并行编译命令对源程序进行改写，这与普通编译器的预处理器是比较类似的。前端则是对预处理后的源程序进行词法、语法分析，将其转换为 IR 形式。主

处理器的工作则是完成基于 IR 的处理与优化。后端是将 IR 转换为并程序，同时完成面向并行体系结构的优化。

当然，除了并行编译器外，并行运行库也是必不可少的，其功能与普通的运行库类似。

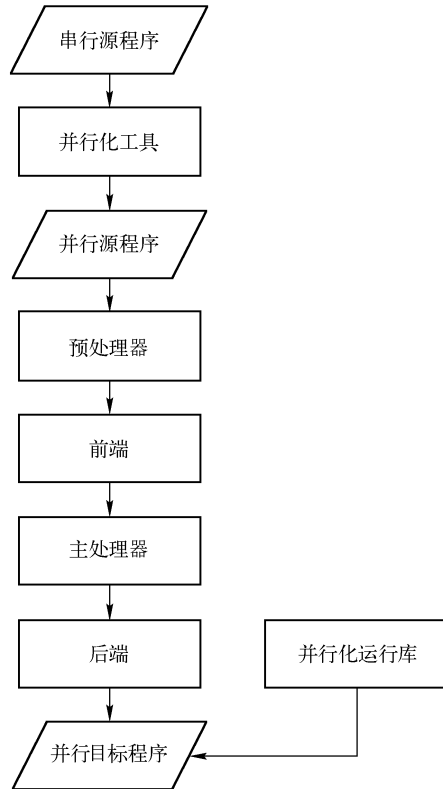


图 10-4 并行编译系统的基本结构

2. 并行编译系统的核心技术

并行编译系统的核心技术包括：依赖关系分析、程序并行化、并行编译技术、并行运行库。下面，笔者对此作简单介绍。

(1) 依赖关系分析。通常，程序的各个部分之间是存在一定依赖关系的，并行计算就是在不破坏原有依赖关系的基础上并行执行的。因此，依赖关系分析的理论与技术是并行计算的基础，也是并行编译技术讨论的重点之一。没有依赖关系的辅助，讨论并行优化技术或者生成技术都是没有意义的。例如，两个函数都访问一个全局变量，那么，则认为它们之间是存在依赖关系的，如果随意并行化，将无法保存程序的正确性。

(2) 程序并行化。在并行处理过程中，尽可能挖掘程序的并行性是编译器的职责。正如先前讨论的，存在依赖关系的部分是不能并行化的。然而，让用户在程序设计过程中尽可能减少依赖关系又是非常困难的。因此，编译器不得不通过一些等价的程序变换，以消除某些依赖关系，以便进行最大限度的并行化，这种等价变换就称为“程序并行化”。在程序并行化中，循环的处理是比较棘手的，因为它的依赖关系比顺序结构复杂得多。然而，由于循环对程序执行的性能影响是比较大的，因此，又是最需要并行处理的部分。在程序并行化技术

中，人们重点关注的就是循环的并行化处理。从变换的角度而言，主要可以分为两种形式：语句层次的变换和迭代层次的变换。其中，语句层次的变换主要着眼于改变循环体中的语句，以消除依赖关系。而迭代层次的变换则更关注改变循环体中迭代的执行顺序。当然，任何程序并行化的前提都是安全。与优化类似，语义等价是编译技术永恒的话题。

(3) 并行编译技术。这里主要指的是用于实现并行编译器的相关技术，包括向量语言处理、并行语言处理、并行目标程序组织等。从用户角度而言，主要有两个方面的需求：程序可移植性、并行执行的效率。用户通常希望程序的可移植性较好，也就是说，在设计与实现语言时，尽可能减小对目标机器的依赖。不过，从提高程序执行效率的角度来说，要更好地利用目标机器的资源，发挥目标机器的特性，就不得不在设计语言时充分考虑相关的因素。因此，这两方面矛盾的需求对并行编译器的设计提出了巨大的挑战。

(4) 并行运行库。在并行操作系统、并行语言及并行编译技术都还不成熟的情况下，运行库提供了一种并行程序设计的手段。从并行程序设计技术角度来说，并行运行库可以分为三类：支持多任务并行程序设计的运行库、支持消息传递并行程序设计的运行库、支持数据并行程序设计的运行库。

3. 并行编译技术的发展趋势

实践证明，一门学科、一项技术的发展与成熟与它的应用是分不开的。因此，并行编译技术的发展很大程度上依赖于并行语言的成熟与流行。在并行语言方面，Fortran 是占绝对主导地位的。20 世纪 70 年代中后期，推出了多个在串行 Fortran 语言上扩充而来的向量语言。1991 年 ANSI 和 ISO Fortran 90 标准的发布标志着向量 Fortran 语言完成了标准化工作。在并行语言中，由于在科学计算中的优势，Fortran 的研究远胜于 C 或其他语言。

有些读者可能认为：并行计算是实验室的产物，离普通用户的应用需求相距甚远。的确，就目前的研究成果而言，笔者并不否认这一观点。不过，很多研究已经表明，基于局域网或互联网的分布并行计算将在未来的 10 年中得到迅速发展。最终，这一技术将进入普通用户的视野，而不再停留在实验室中。与一味追求峰值性能的超级计算机不同，网络分布并行计算更关注一些有实际应用价值的因素，如负载平衡、共享计算资源、网络安全、可靠性等。届时，并行程序设计、并行编译技术将迎来新的纪元。

10.4.2 并行计算机及其编译系统

在高性能计算领域，并行体系结构主要可分为三类：向量计算机、共享存储器并行计算机、分布存储器并行计算机。下面，笔者对这三类并行结构及其编译系统作简单介绍。

1. 向量计算机及其编译系统

向量是一个数学概念，就是指类型相同的数据项的集合。向量的概念看似有些神秘，不过，它的实现形式却早已深入人心。在程序设计语言中，向量的实现形式就是数组。例如，对于向量语言的标准化具有里程碑意义的 Fortran 90 也被称为“数组语言”。其主要原因就是从体系结构和编译器实现来看，数组操作就是向量操作的一种实现形式。

而向量计算机就是指具有高效的向量处理能力的计算机系统。有读者可能疑惑，似乎普通计算机也同样具有向量处理能力，那么，它们是不是向量计算机呢？答案显然是否定的。在普通标量计算机中，向量处理是基于软件实现的。例如，在 C 语言中，计算两个向量的和就必须依赖程序实现。而这里所说的向量处理能力通常是指硬件层次上的，也就是说，向量

计算机可以通过硬件指令直接完成复杂的向量运算，而不需要借助软件实现。对于超级计算技术而言，向量计算对提高超大规模科学计算的性能是极其重要的，尤其在大型机、巨型机等系统中。

与普通标量计算不同，向量计算对计算机的寄存器组、流水线处理、存储访问控制都提出了新的需求。通常，需要计算机能够一次读取、存入一组数据并执行相关的计算处理。因此，除了标量功能部件之外，向量计算机还专门设有向量寄存器、向量长度寄存器、向量屏蔽寄存器、向量流水功能部件和向量指令系统等。这里，以两个向量求和为例作简单说明，如图 10-5 所示。

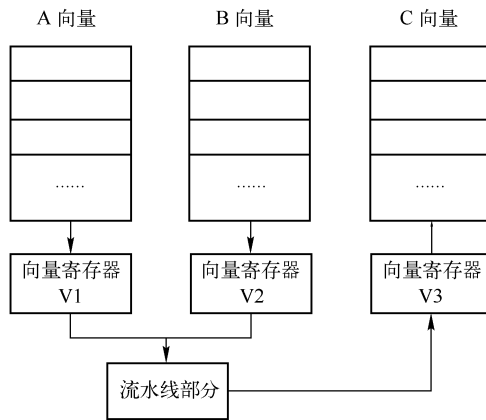


图 10-5 向量求和的处理过程

在图 10-5 中，假设 A、B、C 三个向量的长度都是 64，那么，此求和操作需要进行的迭代次数为 1，这比普通标量计算机需要的 64 次迭代高效得多。Fortran 90 的描述形式如下：

$$C(1:64) = A(1:64) + B(1:64)$$

但是，更多时候，用户习惯于书写如下的简单循环形式：

```
DO I=1,64
  C(I) = A(I) + B(I)
ENDDO
```

当然，这两种形式是完全等价的，因此，并行化的任务就是将后者转换为前者。不过，这种转换是非常复杂的，不像读者想象的那样简单。

下面，笔者再给出一个例子：

```
DO I=1,64
  A(I+3) = A(I) + B(I)
ENDDO
```

试想一下，是否可以将其简单地将其转换为：

$$A(4:67) = A(1:64) + B(1:64)$$

答案是否定的，原因非常简单，每一个迭代完的结果将影响后续迭代的过程，因此，这种存在依赖关系的情况是不可量化的。那么，哪些情况可以转换，而哪些情况不可以转换呢？这就是依赖关系分析要解决的问题。

2. 共享存储器的并行计算机及其编译系统

共享存储器的并行计算机主要由多个处理机、一个共享的中央存储器及专门的同步通信部件组成。处理机可以是向量机，也可以是标量机。系统一般是 MIMD（多指令流多数据流）体系结构。如图 10-6 所示。

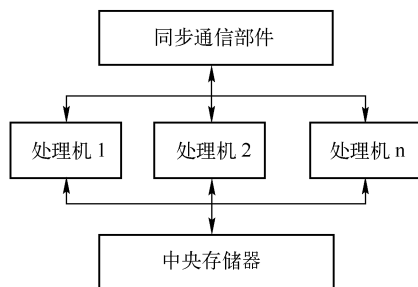


图 10-6 共享存储器的并行计算机的基本结构

在共享存储器的并行计算机中，虽然，物理上多个处理机是共享一个中央存储器的，但是，一个处理机究竟可以访问哪些存储数据及一个数据应该被哪个处理机优先访问等都是由用户控制的，硬件机制并没有严格的限制。而其中的同步通信部件只是为处理机之间的同步通信提供硬件支持，例如，用于实现同步原语的共享信号灯等。

与向量计算机相比，共享存储器的并行计算机在更大程度提高了并行处理的能力。多处理机的结构使得计算机可以并行执行多个循环迭代、程序块或函数。如果所属的处理机都是向量计算机，那么，该系统的并行能力将更为出色。在低层次上，由向量计算机完成向量计算方面的迭代，而高层次上，则由共享存储器的并行计算机提供服务。

基于共享存储器的并行计算机的编译系统通常需要完成如下工作：

(1) 串程序并行化。识别与分析输入串行源程序的可并行化的部件，将整个程序任务划分为多个可并行执行的部分，以便多处理机并行计算。当然，这项工作是基于依赖关系分析完成的。正如先前讨论的，并行化的主要对象仍然是循环迭代，也就是将循环迭代形式转换为等价的可并行处理的形式，或者插入并行编译指导命令。

(2) 处理并行语法机制。根据并行语言的特点，将源程序编译成相应的目标程序。这里主要指的是与并行机相关的语法成分，如并行循环结构、并行段结构、并行区结构等。除此之外，编译器还需关注任务调度、处理机分配、任务同步等方面的工作。

3. 分布存储器的并行计算机及其编译系统

分布存储器的并行机是由很多相对独立的结点（通常就是独立的计算机）通过网络连接而成的，每个结点都有自己的处理机和存储器。结点数是庞大的。如图 10-7 所示。

这类计算机的特点是存储访问时间不一致，即处理机对远程存储器的访问时间比对本本地存储器的访问时间要长得多。分布存储器的并行计算机得以发展主要有两方面的原因：

(1) 系统组建的价格相对便宜。与向量计算机、共享存储器的并行计算机相比，分布存

储器的并行计算机主要是依赖于微型计算机的，因此，其成本有其他两类并行计算机无法比拟的优势。

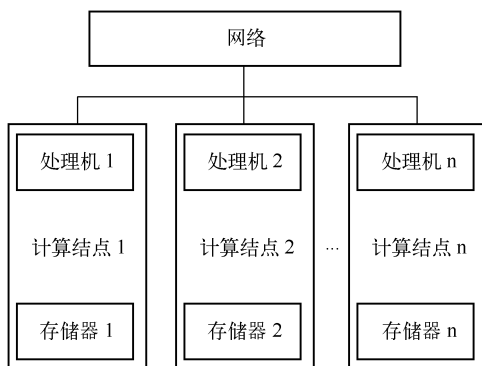


图 10-7 分布存储器的并行计算机的基本结构

(2) 分布存储器结构突破了共享存储器的性能瓶颈。虽然共享存储器结构的并行计算机可以在计算处理方面达到高数值，但存储器的访问却是这类并行计算机的瓶颈。而分布存储器结构的方案却不存在这种瓶颈，从而可以追求更高的峰值。不过，值得注意的是，这种方案也不是绝对完美的，网络性能将在很大程度上约束并行执行及存储器访问的效率。

随着互联网的普及与发展，分布存储器的并行计算机结构已经成为当今最热门的话题之一。近年来，被炒得热火朝天的“云计算”也可以视为这种计算机结构的商业化产物。而分布计算的主要研究对象就是这种计算机结构。

在分布存储器的并行计算机中，数据并行程序设计语言是主要的并行语言，它扩充了普通语言在数据分布与并行处理方面的描述能力。例如，Split-C、CM Fortran 等。数据并行语言便于用户简单且直观地编写并行程序，而不必关注并行处理的细节问题。

目前，在分布存储器的并行计算机上的编译系统主要是用于处理数据并行语言的，其工作包括以下几项：

(1) 数据分布。为了减少通信资源耗费，提高执行的效率，并行任务与相关数据的分布是编译器需要关注的。虽然无论使用何种分配算法，保证并行任务与相关数据绝对被分配在同一个计算结点是不现实的，但这却是编译器设计者的努力方向。根据计算结点的分布情况，合理安排任务与数据的布局，以减少通信资源的耗费，这是完全可行的。

(2) 任务划分。如何合理地划分任务，也是实现计算与参与计算的数据尽可能被分配到同一个计算结点的一种有效策略。通常的任务划分原则是拥有者计算，即数据在哪个计算结点上，则优先考虑由哪个结点完成计算任务。

(3) 同步与通信。主要工作包括确定同步与通信点、插入相应的并行库子程序调用、同步通信优化等。

10.5 深入学习

关于 GCC 的学习，唯一建议就是深入阅读源代码。笔者相信没有任何形式的文档可以

与源代码媲美。迄今为止，所能见到的关于 GCC 的最权威参考就是《GNU Compiler Collection Internals》，读者可以在 GCC 的网站免费下载。其中非常详细地描述了 GCC 的内核实现。不过，该书并不适合作为教材阅读，而更多地是在阅读源代码过程中作为辅助参考手册。

关于动态编译的书籍很难找到，对动态编译感兴趣的读者只能以论文资料为主。而本书中介绍的 DyC 也是一个非常不错的选择，读者不妨关注一下其相关的资源，或许对于了解动态编译的前沿技术有一定帮助。

相对而言，关于并行编译技术的资源与文献就相对较多。这里，笔者推荐一些并行编译方面的著作。

- 1、并行编译方法
沈志宇、胡子昂
国防科技大学出版社
说明：国防科技大学是国内并行编译领域的先驱，因此，本书堪称国内该领域的经典著作。
- 2、Parallel Computing Fortran Forum(<http://portal.acm.org/citation.cfm?id=122392>)
说明：并行 Fortran 语言的规范。
- 3、Parallel Computing: Theory and Practice
M. J. Quinn
McGraw-Hill
说明：本书以介绍并行计算的理论与实现技术为主，适合初学者学习参考。
- 4、编译原理（紫龙书）
Alfred V. Aho, Monica S. Lam, Ravi Sethi
机械工业出版社
说明：“紫龙书”对并行编译及其优化技术也作了相关介绍。
- 5、Optimizing Compilers for Modern Architectures, A Dependence-Based Approach
Randy Allen, Ken Kennedy
Elsevier Science
说明：这是并行编译领域最权威的教材。

10.6 大师风采——Alan Perlis

Alan Jay Perlis: 美国著名计算机科学家，也是计算机程序设计语言领域的先驱。1922 年 4 月，Perlis 出生于宾夕法尼亚州匹兹堡的一个犹太人家庭。1943 年，他获得卡内基理工学院化学学士学位。二战参军期间，Perlis 对数学产生了兴趣，并于 1949 年、1950 年分别获得了麻省理工学院数学硕士、博士学位。

1956 年，Perlis 加入了卡内基理工学院，任计算机科学系主任。1966 年，为表彰其在高级程序设计语言及编译器设计领域的成就，Perlis 获得了第一届图灵奖。他的研究成果对于后来的 Algol 语言的设计与实现有深远的意义。

1971 年，Perlis 加入了耶鲁大学计算机科学系，任计算机科学系主任。他还当选为美国艺术和科学院院士及美国工程院院士。1990 年 2 月 7 日，Perlis 因心脏病在康涅狄格州纽哈芬去世。

参 考 文 献

- [1] Alfred V Aho, Monica S Lam, Ravi Sethi, Jeffrey D Ullman. Compilers: Principles, Techniques, and Tools [M]. 2nd ed. New York: Pearson Education, Inc, 2009.
- [2] Christopher W Fraser, David R Hanson. 可变目标 C 编译器——设计与实现 [M]. 王挺, 黄春, 译. 北京: 电子工业出版社, 2005.
- [3] Steven S Muchnick. Advanced Compiler Design and Implementation [M]. San Francisco: Morgan Kaufmann Publishers, 1997.
- [4] Y N Srikant, Priti Shankar. The Compiler Design Handbook: Optimizations and Machine Code Generation [M]. 2nd ed. CRC Press, 2007.
- [5] 侯文永, 张冬荣. 编译原理 [M]. 北京: 电子工业出版社, 2002.
- [6] 陈火旺, 刘春林, 谭庆平, 赵克佳, 刘越. 程序设计语言编译原理 [M]. 3 版. 北京: 国防工业出版社, 2000.
- [7] Kathleen Jensen, Niklaus Wirth. PASCAL User Manual and Report [C]. New York: Springer-Verlag, 1974.
- [8] Andrew W Appel, Jens Palsberg. Modern Compiler Implementation in Java [M]. 2nd ed. New York: Cambridge University Press, 2002.
- [9] Robert W Sebesta. 程序设计语言原理 [M]. 5 版. 张勤, 译. 北京: 机械工业出版社, 2004.
- [10] John C Mitchell. 程序设计语言理论基础 [M]. 许满武, 徐建, 袁宜, 杨群, 译. 北京: 电子工业出版社, 2006.
- [11] Glynn Winskel. 程序设计语言的形式语义 [M]. 宋国新, 邵志清, 译. 北京: 机械工业出版社, 2005.
- [12] Kip R Irvine. Intel 汇编语言程序设计 [M]. 4 版. 温玉杰, 张家生, 罗生彬, 译. 北京: 电子工业出版社, 2005.
- [13] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual [M]. 2nd ed. Intel Corporation, 2007.
- [14] Richard M Stallman, the GCC Developer Community. GNU Compiler Collection Internals [OL]. <http://gcc.gnu.org/onlinedocs/gccint/>.
- [15] ISO Org. Pascal ISO 7185:1990 [S/OL]. <http://www.iso.org/>.
- [16] Harold Abelson, Gerald Jay Sussman, Julie Sussman. Structure and Interpretation of Computer Programs [M]. 2nd ed. The MIT Press, 1996.
- [17] Keith D Cooper, Linda Torczon. Engineering a Compiler [M]. 冯速, 译. 北京: 机械工业出版社, 2006.
- [18] Benjamin C Pierce. Advanced Topics in Types and Programming Languages [M]. The MIT Press, 2005.
- [19] P D Terry. Compilers and Compiler Generators: an introduction with C++ [M]. International Thomson Computer Press, 1997.
- [20] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, Susan J Eggers. DyC: An Expressive Annotation-Directed Dynamic Compiler for C [C]. Technical Report, 1999.

- [21] 沈志宇, 胡子昂, 廖湘科, 吴海平, 赵克佳, 卢宇彤. 并行编译方法 [M]. 北京: 国防工业出版社, 2000.
- [22] 张素琴, 吕映芝, 蒋维杜, 戴桂兰. 编译原理 [M]. 2 版. 北京: 清华大学出版社, 2008.
- [23] Benjamin C Pierce. 类型和程序设计语言 [M]. 马世龙, 蛙跃飞, 译. 北京: 电子工业出版社, 2005.
- [24] 白童心, 冯晓兵, 武成岗, 张兆庆. 优化动态二进制翻译器 DigitalBridge [J]. 计算机工程, 2005, 31(10):103-105.
- [25] Randy Allen, Ken Kenedy. 现代体系结构的优化编译器 [M]. 张兆庆, 等译. 北京: 机械工业出版社, 2004.

ISBN 978-7-111-32164-4

策划编辑：车 忱

封面设计：



子时文化
Zi Shi Culture

编译器 设计之路

内容简介

本书系统地介绍了一个实际的Pascal编译器Neo Pascal的设计与实现。结合Neo Pascal的源代码，详细讲述了LL(1)语法分析器、符号表系统、中间表示、类型系统、优化技术、运行时刻的存储管理、代码生成器等编译器设计的核心话题。

与国内其他介绍编译技术的图书相比，本书更关注的是编译器的实现细节，而不仅仅局限于理论阐述。



读者可在 <http://neopascal.sourceforge.net>

获得Neo Pascal的源代码及相关文档。

地址：北京市百万庄大街22号
电话服务
社服务中心：(010)88361066
销售一部：(010)68326294
销售二部：(010)88379649
读者服务部：(010)68993821

邮政编码：100037
网络服务
门户网：<http://www.cmpbook.com>
教材网：<http://www.cmpedu.com>
封面无防伪标均为盗版

定价：59.00元

上架建议 计算机/程序设计

ISBN 978-7-111-32164-4



9 787111 321644 >