



请加入群570693764

免费为程序员提供电子书订制服务

只要您要的书籍，我们都会尽量为您做出电子版

一百多万册的数字图书馆，随时免费为您下载

请加入群570693764

DIY Compiler and Linker

自己动手写 编译器、链接器

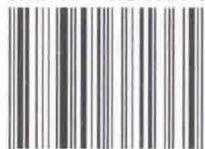
王博俊 张宇 编著



清华大学出版社

本书特色

- 本书展示了从语言定义，到进行词法分析、语法分析、语义分析的编译器完整开发过程。
- 本书介绍的SCC编译器，没有借助Lex与Yacc编译器自动生成工具，纯手工编写而成，更便于学习和理解。
- 为了使编译生成的成果可以直接运行，本书还实现了一个链接器，链接生成可以直接运行的EXE文件。
- 编写编译器用到的知识之广是编写一般程序所无法比拟的，通过本书你将学到编译原理、数据结构与算法、Intel x86汇编语言、机器语言、目标文件格式、可执行文件格式等知识内容。
- 从给微机编写BASIC语言编译器起家的比尔·盖茨，在世界首富宝座上稳坐多年，编写编译器的技术积淀究竟在他的成功中占多大份量，通过阅读本书，你会从技术层面上有所洞悉。
- 有了编译器的编写经历，你将拿到一把学习任何计算机语言的万能钥匙，你会发现所有的计算机语言原来都是相通的，各种语言核心内容其实也就是词法、语法、语义那么点事。
- 如果你想了解各类计算机语言设计思路，编译器、链接器实现过程，本书将使你豁然开朗。



DIY Compiler and Linker

自己动手写 编译器、链接器

王博俊 张宇 编著

清华大学出版社

内 容 简 介

本书讲述了一个真实编译器的开发过程,源语言是以 C 语言为蓝本,进行适当简化定义的一门新语言,称之为 SC 语言(简化的 C 语言),目标语言是大家熟悉的 Intel x86 机器语言。在本书中,读者将看到从 SC 语言定义,到 SCC 编译器开发的完整过程。本书介绍的 SCC 编译器,没有借助 Lex 与 Yacc 这些编译器自动生成工具,纯手工编写而成,更便于学习和理解。为了生成可以直接运行 EXE 文件,本书还实现了一个链接器。读完本书读者将知道一门全新的语言如何定义,一个真实的编译器、链接器如何编写。

本书适合各类程序员、程序开发爱好者阅读,也可作为高等院校编译原理课程的实践教材。

郑重声明: 本书源代码作者已申请版权,仅供读者用于学习研究之目的。未经作者允许,严禁任何组织与个人将其在网络上传播或用于商业用途。对于侵权行为,作者保留提起法律诉讼的权利。源代码相关问题,请与作者联系,作者邮箱:1259809207@qq.com。

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。

版权所有,侵权必究。侵权举报电话:010-62782989 13701121933

图书在版编目(CIP)数据

自己动手写编译器、链接器/王博俊, 张宇编著. --北京: 清华大学出版社, 2015

ISBN 978-7-302-38136-5

I. ①自… II. ①王… ②张… III. ①C 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字(2014)第 227835 号

责任编辑: 袁勤勇 徐跃进

封面设计: 傅瑞学

责任校对: 李建庄

责任印制: 李红英

出版发行: 清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址: 北京清华大学学研大厦 A 座 邮 编: 100084

社 总 机: 010-62770175 邮 购: 010-62786544

投稿与读者服务: 010-62776969, c-service@tup.tsinghua.edu.cn

质量反馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者: 北京富博印刷有限公司

装 订 者: 北京市密云县京文制本装订厂

经 销: 全国新华书店

开 本: 185mm×260mm **印 张:** 22.25 **字 数:** 557 千字

版 次: 2015 年 2 月第 1 版 **印 次:** 2015 年 2 月第 1 次印刷

印 数: 1~2000

定 价: 44.50 元

产品编号: 059284-01

序 言

因为工作的关系，我经常和各企业的技术负责人交流。话题谈着谈着常常会转到他们目前共同的难题——技术人员招聘。这时不少人都会感慨，中国能做系统软件开发的技术人员太少，这方面的人太难找了。随着中国企业的发展，做系统和平台的需求不断增加，这种供需矛盾将越来越明显。

究其原因，很容易想到的是我们的高校教育、课程设置。美国顶尖大学计算机系基础课程教学里都非常重视项目实践，操作系统课往往要真的开发一个像模像样的操作系统原型，编译器课也真的要自己设计并实现一门有创新性的小语言……

在计算机科学的各门课程中，编译器的设计实践有着特殊的重要性。“龙书”的主要作者、哥伦比亚大学教授 Alfred V. Aho 曾经列举过编译器实践有诸多好处：

- 能让学生领悟到理论与实践的完美结合。比如编译原理所涵盖的正则表达式和自动机，在各种场合的应用是极其广泛的，对正则的掌握程度，从某种意义上讲甚至可以作为技术人员水平的一种尺度。
- 深入探索计算思维的多样性。与人类语言一样，不同类型的编程语言其实代表了不同的思维方式。只用过命令式语言的人可能没有想到，开启了大数据领域的 Map 与 Reduce，其实在函数式语言是一种非常常见的东西。

的确，深入了解编译器和编译原理，对于技术人员更好地理解和掌握自己最常用的语言和系统，从而提升自己的内力是有极大好处的。另一方面，随着 DSL（领域特定语言）的流行，需要技术人员开发自己语言的机会也越来越多。

然而，编译原理是计算机科学里公认比较难的一门课。虽然目前国外比较重要的编译理论教材（比如龙书的《编译原理》、虎书《现代编译原理》的 C 语言和 Java 版本、鲸书《高级编译器设计与实现》）基本上都有了中文版和英文影印版，但这些书往往更偏重理论，而且门槛较高，不太适合指导一线技术人员实践和自学。我认识的一位美籍华人技术专家 Ronald Mak 在 Wiley 出版过一本基于 Java 的“Writing Compilers and Interpreters”，比较贴近实践，但部头较大，而且没有看到中文版。

偶然的机会，我得知王博俊在工作之余，写了一本以简化的 C 语言为例子讲述编译器和链接器实践的书。浏览了初稿之后，感觉全书内容简明，容易上手，又不失全面和系统，正好弥补了这方面的空白。特向大家推荐。

CSDN 暨《程序员》杂志总编 刘江

2015 年 1 月

自序

纸上得来终觉浅，绝知此事要躬行。

——陆游

编译原理与技术的一整套理论在整个计算机科学领域占有相当重要的地位,学习它对程序设计人员有很大的帮助。我们考究历史会发现那些人人称颂的程序设计大师都是编译领域的高手,像写出 BASIC 语言的比尔·盖茨,Sun 公司的 Java 之父等,在编译领域都有很深的造诣。曾经在世界首富宝座上稳坐多年的比尔·盖茨也是从给微机编写 BASIC 语言编译器起家的,也正是这个 BASIC 编译器为比尔·盖茨和保罗·艾伦的微软帝国奠定了基础。这个编写 BASIC 语言编译器的经历,开启了比尔·盖茨的辉煌职业生涯。

编译器是一种相当复杂的程序,编写甚至读懂这样的一个程序都非易事,大多数的计算机科学家和专业人员也从来没有编写过一个完整的编译器。但是,几乎所有形式的计算都要用到编译器,而且任何一个与计算机打交道的专业人员都应掌握编译器的基本结构和操作。除此之外,计算机应用程序中经常遇到的一个任务就是命令解释程序和界面程序的开发,这比编译器要小,但使用的却是相同的技术。因此,掌握这一技术具有非常大的实际意义。

李国杰院士说:“随着微处理器技术的飞速发展,处理器性能在很大程度上取决于编译器的质量,编译技术成为计算机的核心技术,地位变得越来越重要。我国要发展自己的微处理器事业,必然要有自己的编译技术作为后盾。”

回过头来说一说是什么样的原因使我萌生了写这样一本书的想法。作者学习其他计算机课程感觉没有特别难懂的,唯独看编译原理的教材,看完了云里雾里的,感觉一知半解,我感觉可能是学的教材过于理论化,于是到书店把所有跟编译原理有关的书籍统统买回家,当然这也包括大家公认的编译原理三大经典书籍(龙书、虎书、鲸书)在内,每一本我都从头到尾翻一遍,好像什么都懂了,又感觉要真的自己动手写个编译器仍然是只有大师才能完成,对自己还是可望而不可即的事情。并且作者也了解到许多关于编译原理实践的悲观论调:“现有的编译器都是用 Lex 和 Yacc 构造的,从头开始手工编写一个完整的编译器几乎是不可能的。”可作者偏偏是那种“明知山有虎,偏向虎山行”的人,要知道早期的编译器可都是纯手工构造的,苦辣酸甜的征程就此开始,可是写个什么语言的编译器?这个编译器怎么定位?这一切都很茫然。

我开始研究编译原理书上的样例,希望能从中找到灵感,给上述问题找到答案。世界著名计算机科学家 N. Worth 编写的 PL/0 语言的编译程序是作者最先研究的编译器,它功能简单、结构清晰、可读性强,被认为是一个非常合适的小型编译程序的学习模型,可这个编译程序不支持数组、结构体、字符串,并且是以假想的栈式机器为例来编写的,而不是直接生成在某种 CPU,某种操作系统环境下直接可以运行的目标语言程序。“PL/0 语言的编译程序”作为编译器的学习模型,也只能算“矬子里面拔将军”,因为没有更好的,也只好将就着用了。至此,编译器定位问题算有了些眉目,作者希望构造一个更适合学习的编译器。

可是,另一个问题接踵而至,为什么那么多开源编译器不能直接用作编译器学习模型呢?我开始研究各个开源编译器的源代码,其中包括GCC的源代码,由于GCC支持多个前端语言和各种后端机器平台、AST(Abstract Syntax Tree)和RTL(Register Transfer Language)又成了绕不过去的坎,还没学会怎么编写针对一种源语言、一种目标机器的编译器,就要去学习支持多种源语言多个机器平台的编译器,就好比一个婴儿还没学会走路就要学跑,这注定是要跌跟头的。

一面是过于简化的编译器学习模型,另一面是过于复杂的开源编译器,作为学习模型都不太合适。到这里,编译器定位问题算是彻底想清楚了,作者要构造一个教大家如何自己动手写编译器的学习模型。这个模型包括两大部分,第一部分是语言定义,第二部分是这个语言编译器的实现,这个编译器只支持一种源语言,目标语言也只支持一种。这个语言应该具备目前流行的高级语言的最主要特征。这个编译器要结构清晰,代码量要尽可能少,要能体现编写一个实用的编译器的完整过程与技术。这个编译器可以生成在操作系统中直接运行的exe文件,只要双击或在命令行执行就能看到结果的那种。

接下来作者开始思考另一个问题,编写个什么语言的编译器?作者研究了目前最流行的几种编程语言C、C++、C#、Objective-C、Java,其中C语言是最简单的了,只有32个关键字,但是作者研究发现,C语言还是有许多冗余的成分,作为学习模型还可以更简单一些。作者最终以C语言为蓝本,进行适当简化定义了一门新的语言,仅有15个关键字,称为SC语言。目标语言选择大家熟悉的Intel x86机器语言,编译器命名为SCC编译器。

在本书中,读者将看到从SC语言定义,到SCC编译器开发的完整过程。读完本书你将知道一门全新的语言如何定义,一个真实的编译器如何编写,这些对你来说将不再神秘,编译原理讲的理论与本书中讲述的SC语言定义及SCC编译器开发过程,是理论联系实际在编译领域的最好阐释。

如本书作为编译原理实践教材,作者建议安排10学时讲授。

本书投稿后,有幸请CSDN暨《程序员》杂志总编、刘江老师阅读了本书的初稿,并为本书做序,在此向刘老师表示最衷心的感谢。

本书临近出版之际,承蒙清华大学王生原老师阅读了本书终稿,并对书稿做了中肯评价:“本书特色鲜明,内容有深度,文笔也很不错,很值得出版。本书最大的特色是所选的目标平台,即x86处理器以及微软系统的COFF目标文件格式,这在教材中很少见到,可为国内的编译教学实践提供别具一格的素材。”同时,王老师还对本书提出了宝贵建议。在这里,向王老师表示由衷的敬意和最诚挚的感谢。

我还要感谢我的家人,他们的支持与鼓励是本书得以完成的保障。

要列出所有对本书出版有所帮助的人名是不可能的,因为有些困难是通过互联网解决的,我甚至不知道他们的名字。在此,谨向他们一并表示感谢!

最后,回想本书6年的写作历程,愿以蒲松龄的一副对联与读者共勉:

有志者,事竟成,破釜沉舟,百二秦关终属楚;

苦心人,天不负,卧薪尝胆,三千越甲可吞吴。

王博俊

2015月1月

目 录

第 1 章 引言	1
1.1 HelloWorld 编译过程分析	1
1.1.1 HelloWorld 程序源文件	1
1.1.2 词法分析	2
1.1.3 语法分析	3
1.1.4 语义分析	3
1.1.5 链接器	4
1.2 SCC 编译器简介	7
1.2.1 SCC 编译器架构	7
1.2.2 SCC 编译器开发环境	7
1.2.3 SCC 编译器运行环境	8
第 2 章 文法知识	10
2.1 语言概述	10
2.2 形式语言	11
2.2.1 字母表和符号串	11
2.2.2 文法与语言的形式定义	12
2.2.3 文法与语言的类型	13
2.2.4 程序设计语言描述工具	15
2.3 词法分析方法	16
2.3.1 词法定义例举	17
2.3.2 状态转换图	17
2.3.3 词法分析程序流程图	17
2.4 语法分析方法	18
2.4.1 LL 分析器	18
2.4.2 LL(k)文法	19
2.4.3 LL(1)文法	19
2.4.4 递归子程序法	21
2.4.5 文法的等价变换	24

第 3 章 SC 语言定义	26
3.1 SC 语言的蓝本选择	26
3.1.1 K&R C	26
3.1.2 C89	26
3.1.3 C99	27
3.2 SC 语言对 C89 简化原则	27
3.3 SC 语言的字符集	27
3.3.1 基本字符集	28
3.3.2 扩展字符集	28
3.4 SC 语言词法定义	29
3.4.1 关键字	29
3.4.2 标识符	30
3.4.3 整数常量	31
3.4.4 字符常量	31
3.4.5 字符串常量	32
3.4.6 运算符及分隔符	32
3.4.7 注释	33
3.5 SC 语语法定义	33
3.5.1 外部定义	33
3.5.2 语句	35
3.5.3 表达式	39
3.6 SC 语言与 C 语言功能对比	46
3.6.1 关键字	46
3.6.2 数据类型	46
3.6.3 存储类型	47
3.6.4 常量	47
3.6.5 变量	47
3.6.6 函数	48
3.6.7 语句	48
3.6.8 表达式	50
第 4 章 SC 语言词法分析	52
4.1 词法分析任务的官方说法	52
4.2 单词编码	53
4.3 词法分析用到的数据结构	55
4.3.1 动态字符串	56
4.3.2 动态数组	58
4.3.3 哈希表	61

4.3.4 单词表	62
4.4 错误处理,未雨绸缪	67
4.5 词法分析过程.....	72
4.5.1 词法分析主程序	72
4.5.2 预处理	76
4.5.3 解析标识符	79
4.5.4 解析整数	80
4.5.5 解析字符串	80
4.5.6 词法分析流程图	82
4.6 词法着色.....	84
4.7 控制程序.....	85
4.8 词法分析成果展示.....	86
第5章 SC 语言语法分析	87
5.1 外部定义.....	87
5.1.1 翻译单元	87
5.1.2 外部声明	88
5.1.3 类型区分符	90
5.1.4 结构区分符	92
5.1.5 函数调用约定	95
5.1.6 结构成员对齐	95
5.1.7 声明符	96
5.1.8 初值符.....	100
5.2 语句	101
5.2.1 复合语句.....	102
5.2.2 表达式语句.....	103
5.2.3 选择语句.....	104
5.2.4 循环语句.....	104
5.2.5 跳转语句.....	105
5.3 表达式	107
5.3.1 赋值表达式.....	108
5.3.2 相等类表达式.....	109
5.3.3 关系表达式.....	109
5.3.4 加减类表达式.....	110
5.3.5 乘除类表达式.....	111
5.3.6 一元表达式.....	112
5.3.7 后缀表达式.....	113
5.3.8 初值表达式.....	114
5.4 语法缩进	116

5.4.1 用到的全局变量及枚举.....	116
5.4.2 语法缩进程序.....	117
5.5 总控程序	118
5.6 成果展示	119
第 6 章 符号表.....	120
6.1 符号表简介	121
6.1.1 收集符号属性.....	121
6.1.2 语义的合法性检查.....	122
6.2 符号表用到的主要数据结构	123
6.2.1 栈结构.....	123
6.2.2 符号表结构.....	127
6.2.3 数据类型结构.....	132
6.2.4 存储类型.....	133
6.3 符号表的构造过程	134
6.3.1 外部声明.....	134
6.3.2 类型区分符.....	137
6.3.3 结构区分符.....	138
6.3.4 声明符.....	144
6.3.5 变量初始化.....	149
6.3.6 复合语句.....	150
6.3.7 sizeof 表达式	150
6.3.8 初等表达式.....	152
6.4 控制程序	153
6.5 成果展示	155
第 7 章 生成 COFF 目标文件.....	157
7.1 COFF 文件结构	157
7.1.1 基本概念.....	157
7.1.2 总体结构.....	158
7.1.3 COFF 文件头	158
7.1.4 节头表.....	161
7.1.5 代码节内容.....	168
7.1.6 数据节与导入节内容.....	168
7.1.7 COFF 符号表	169
7.1.8 COFF 字符串表	173
7.1.9 COFF 重定位信息	173
7.2 生成 COFF 目标文件	175
7.2.1 生成节表.....	176

7.2.2 生成符号表.....	178
7.2.3 生成重定位信息.....	182
7.2.4 生成目标文件.....	183
7.3 成果展示	185
第 8 章 x86 机器语言	187
8.1 x86 机器语言简介	187
8.2 通用指令格式	188
8.2.1 指令前缀.....	188
8.2.2 操作码.....	190
8.2.3 ModR/M 字节	190
8.2.4 SIB 字节	191
8.2.5 偏移量与立即数.....	193
8.3 x86 寄存器	193
8.3.1 数据寄存器:.....	193
8.3.2 变址寄存器.....	193
8.3.3 指针寄存器.....	194
8.3.4 段寄存器.....	194
8.3.5 指令指针寄存器.....	194
8.3.6 标志寄存器.....	195
8.4 指令参考	196
8.4.1 符号说明.....	196
8.4.2 数据传送指令.....	198
8.4.3 算术运算指令.....	200
8.4.4 逻辑运算指令.....	203
8.4.5 控制转移指令.....	205
8.4.6 串操作指令.....	208
8.4.7 处理器控制指令.....	208
8.5 生成 x86 机器语言	208
8.5.1 操作数栈.....	209
8.5.2 生成通用指令.....	210
8.5.3 生成数据传送指令.....	213
8.5.4 生成算术与逻辑运算指令.....	217
8.5.5 生成控制转移指令.....	221
8.5.6 寄存器使用.....	224
8.5.7 本章用到的全局变量.....	227
8.6 成果展示	227
第 9 章 SCC 语义分析.....	229
9.1 外部定义	229

9.1.1	声明与函数定义	229
9.1.2	初值符	232
9.1.3	函数体	234
9.2	语句	237
9.2.1	表达式语句	237
9.2.2	选择语句	238
9.2.3	循环语句	239
9.2.4	跳转语句	241
9.3	表达式	244
9.3.1	赋值表达式	244
9.3.2	相等类表达式	245
9.3.3	关系表达式	246
9.3.4	加减类表达式	248
9.3.5	乘除类表达式	249
9.3.6	一元表达式	250
9.3.7	后缀表达式	253
9.3.8	初值表达式	257
9.4	成果展示	259
第 10 章	链接器	261
10.1	链接方式与库文件	261
10.2	PE 文件格式	263
10.2.1	总体结构	263
10.2.2	DOS 部分	264
10.2.3	NT 头	265
10.2.4	节头表	272
10.2.5	代码节	272
10.2.6	数据节	274
10.2.7	导入节	274
10.3	链接器代码实现	278
10.3.1	生成 PE 文件头	278
10.3.2	加载目标文件	281
10.3.3	加载引入库文件	282
10.3.4	解析外部符号	285
10.3.5	计算节区的 RVA 地址	288
10.3.6	重定位符号地址	291
10.3.7	修正需要重定位的地址	292
10.3.8	写 PE 文件	293
10.3.9	生成 EXE 文件	295

10.4 SCC 编译器、链接器总控程序	297
10.5 成果展示.....	301
10.6 全书代码架构.....	302
第 11 章 SC 语言程序开发	304
11.1 SC 语言程序开发流程	304
11.2 SCC 编译器测试程序	304
11.2.1 表达式测试.....	304
11.2.2 语句测试.....	308
11.2.3 结构体测试.....	310
11.2.4 函数参数传递测试.....	312
11.2.5 字符串测试.....	314
11.2.6 全局变量测试.....	315
11.3 语言举例.....	316
11.3.1 可接收命令行参数的控制台程序.....	316
11.3.2 可接收命令行参数的 Win32 应用程序	317
11.3.3 HelloWindows 窗口程序	318
11.3.4 文件复制程序.....	323
11.3.5 九九乘法表.....	325
11.3.6 打印菱形.....	326
11.3.7 屏幕捕捉程序.....	328
参考文献.....	336
附录 A SC 语言文法定义中英文对照表	337

第1章

引言

世上无难事，只怕有心人

——民谚

编译器是将一种语言翻译为另一种语言的计算机程序。编译器将源语言编写的程序作为输入，而产生用目标语言编写的等价程序。通常，源语言为高级语言（面向人的语言），如 C、C++、FORTRAN 等，而目标语言为机器语言（面向目标机的语言），如 Intel x86、ARM、MIPS、SPARC 等。

本书将和读者一起编写一个完整的 SCC(Simplified C Compiler) 编译器，源语言是新定义的一门语言，称为 SC(Simplified C) 语言，也就是简化的 C 语言，目标语言是 Intel x86 机器语言。SCC 编译器编译过程如图 1.1 所示。



图 1.1 SCC 编译器编译过程

让我们一起踏上编写 SCC 编译器的美妙旅程，一路上可能鲜花烂漫，也可能荆棘丛生，作者作为本次旅行的导游，接下来要先给大家讲一下旅行指南，让大家对本次旅程有个大概了解。

1.1 HelloWorld 编译过程分析

马克思主义认识论认为，认识是一个在实践基础上，由感性认识上升到理性认识，又由理性认识回到实践的辩证发展过程。本书的开头首先分析 SC 语言编写的 HelloWorld 程序的编译过程，以便对本书将要实现的 SCC 编译器的各个阶段的功能有个感性认识，第 2 章学习编写 SCC 编译器用到的编译原理知识，从第 3 章开始 SCC 编译器的实践过程。

1.1.1 HelloWorld 程序源文件

HelloWorld 作为所有编程语言的入门程序，占据着无法改变的地位，这个例程是从 Kernighan & Ritchie 合著的 *The C Programme Language* 开始有的，因为它的简洁、实用，可谓麻雀虽小，五脏俱全，一门语言的 HelloWorld 程序可以看作这门语言语法结构的一个缩影，因此后来几乎所有学习各种计算机语言的书都以 HelloWorld 程序作为学习这门语言的入门程序。下面就先看看用 SC 语言编写的 HelloWorld 程序。

```

1. /*****
2. * HelloWorld.c 源文件
3. *****/
4. int main()
5. {
6.     printf("Hello World!\n");
7.     return 0;
8. }
9.
10. void _entry()
11. {
12.     int ret;
13.     ret=main();
14.     exit(ret);
15. }
16.

```

上面的程序似乎与 C 语言编写的没什么区别。但是，仔细一看会发觉用 SC 语言编写的 HelloWorld 程序多出一个 `_entry` 函数，它是干什么用的？从字面上理解应该是程序的入口点，没错，`_entry` 是上面 SC 语言 HelloWorld 程序的真正入口点。可能你认为，看来 SC 语言程序与 C 语言的人口点是有区别的，SC 语言程序的人口点是 `_entry`，C 语言程序的人口点是 `main` 函数。

讲述 C 语言的书上都说“`main` 是 C 语言程序的入口”，不知道大家是否对这句话的正确性怀疑过，是否深入探究过。在 Visual C++ 下，控制台程序的入口函数是 `mainCRTStartup`，由 `mainCRTStartup` 调用用户编写的 `main` 函数；图形用户界面（GUI）程序的入口函数是 `WinMainCRTStartup`，由 `WinMainCRTStartup` 调用用户编写的 `WinMain` 函数；`mainCRTStartup` 及 `WinMainCRTStartup` 函数的代码封装在已经编译好的 lib 库中，由链接器自动链接到生成的可执行文件中，所有这一切都是链接器悄悄干的，当然悄悄干的可不一定都是坏事，也可能无名英雄做好事。`mainCRTStartup` 及 `WinMainCRTStartup` 函数就是这样的无名英雄，它们将该类程序开始都要做的一些必备且有技术含量的工作，悄悄地做了，并且背着广大程序员“悄悄”地做。SCC 编译器是一个学习模型，目的是让可执行文件中的每一函数都由用户自己的代码产生，不要有那么多“潜规则”，这样更有助于对编译过程的深入理解。

从 SC 语言编写的 HelloWorld 程序源代码中，大致可以了解以下内容：SC 语言还是保持了 C 语言的绝大多数功能，支持函数，函数以 { 开始，以 } 结束，支持变量声明，变量可以为 `int` 型，函数可以返回 `int` 型值，也可以返回 `void` 表示没有返回值，支持函数调用，支持字符串的处理。

1.1.2 词法分析

词法分析器读入 `HelloWorld.c` 源程序字符流，将它们组织为一系列具有词法含义的单词，词法分析器将给每个单词编上号，以便为语法分析提供便利。HelloWorld 程序的单词编码表如表 1.1 所示。

表 1.1 HelloWorld 单词编码表

单 词	编 码	单 词	编 码
关 键 字		常 量	
int	KW_INT	"Hello World!\n"	TK_CSTR
return	KW_RETURN	0	TK_CINT
void	KW_VOID		标标识符
运算符,分隔符		main	TK_IDENT+0
(TK_OPENPA	printf	TK_IDENT+1
)	TK_CLOSEPA	_entry	TK_IDENT+2
{	TK_BEGIN	ret	TK_IDENT+3
:	TK_SEMICOLON	exit	TK_IDENT+4
}	TK_END		
=	TK_ASSIGN		

单词列比较清楚,就是程序中一个个独立单词,编码列中 KW_INT、TK_OPENPA、TK_CSTR、TK_IDENT 等,代表什么呢,它们都是单词枚举类型中的标识符,当成整型常量来理解就可以了。从表 1.1 中可以看出单词将被分为关键字、运算符、常量及标识符,每一个标识符都将有一个独立编码。

HelloWorld 单词编码看上去有些枯燥,好像也并不是很有意思,那么词法分析完成时有什么拿得出手的成果吗?请看图 1.1,这里将关键字显示为绿色,运算符分隔符显示为红色,常量显示为黄色。

1.1.3 语 法 分 析

语法分析是编译程序的核心部分。语法分析的作用是识别由词法分析给出单词序列是否是给定文法的正确句子(程序)。如果在语法分析阶段语法分析程序仅仅告诉我们“你的 HelloWorld.c 程序符合 SC 语言语法”,恐怕多少有点令人沮丧,花了多少天辛辛苦苦写出来的语法分析程序,就能干这点事,恐怕我们一点成就感都没有,因为 HelloWorld.c 符合 SC 语言语法,我们早就知道的。那么费不少工夫写出来的语法分析程序能不能干点有技术含量的活呢?请看图 1.3,语法分析程序实现了对 HelloWorld 源程序语法缩进功能。图 1.3 与图 1.2 有什么区别呢?请读者仔细对比一下。有了语法缩进程序,用 SC 语言写的代码再乱也不用担心了,哪怕程序完全写在一行也没关系,只要用语法自动缩进程序处理一下就美观了。

1.1.4 语 义 分 析

编译器必须要忠实地将 SC 语言写的程序编译成机器指令,在语义分析阶段 SCC 编译器要当好“傀儡”,循规蹈矩,将“主子”(SC 语言)的意思忠实的传达给 Intel x86 处理器。在语义分析阶段将生成目标文件 HelloWorld.obj,它保存了语义分析的成果,来看一下目标文件生成过程,如图 1.4 所示。

```

E:\自己动手写编译器\代码样例\第四章>scc.exe HelloWorld.c

int main()
{
    printf("Hello World!\n");
    return 0;
}

void _entry()
{
    int ret;
    ret = main();
    exit(ret);
}
EndOfFile
代码行数: 16行
HelloWorld.c 词法分析成功!

```

图 1.2 HelloWorld 词法着色成果展示

```

E:\自己动手写编译器\代码样例\第五章>scc.exe HelloWorld.c

int main()
{
    printf("Hello World!\n");
    return 0;
}

void _entry()
{
    int ret;
    ret = main();
    exit(ret);
}
EndOfFile
HelloWorld.c 语法分析通过!

```

图 1.3 HelloWorld 语法缩进成果展示

```

E:\自己动手写编译器\代码样例>scc -o HelloWorld.obj -c HelloWorld.c
HelloWorld.c 编译成功

```

图 1.4 HelloWorld 编译生成目标文件

图 1.5 是 HelloWorld. obj 文件内容,是不是看上去跟天书一样,这是我们看到的第一封天书,第 7 章、第 8 章和第 9 章将对这封天书从不同侧面进行全方位解读。

1.1.5 链接器

严格地说,链接器不属于 SCC 编译器的工作范畴,但 SCC 编译器如果就停留在此处,恐怕有些扫兴。上面的 HelloWorld. obj 文件只能算是个半成品,还没法直接执行,就像一家汽车生产企业,将轮胎、发动机、车身等零部件生产出来了,但是还没有组装,这样的汽车当然没法跑起来,链接器充当着组装车间的角色,它将 HelloWorld. obj 与 C 运行时库链接生成 HelloWorld. exe 可执行文件。这个链接及执行过程如图 1.6 所示。

这可是令人激动的时刻,具有里程碑的意义。SCC 编译器最终可以生成可执行文件,执行 HelloWorld. exe 就会在命令行显示“Hello World!”,同样的 HelloWorld. exe 执行结果,但心情大不一样,因为这可是用自己亲手写的 SCC 编译器编译生成的。

HelloWorld. exe 既熟悉,又神秘,熟悉的是,运行 HelloWorld. exe 就会在命令行打印出“Hello World!”,神秘的是这个文件中到底存着什么,这个文件为什么能够“指挥”计算机

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00000000h:	4C	01	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000010h:	00	00	00	00	2E	74	65	78	74	00	00	00	00	00	00	00
00000020h:	00	00	00	00	46	00	00	00	54	01	00	00	00	00	00	00
00000030h:	00	00	00	00	00	00	00	00	20	00	00	20	2E	64	61	74
00000040h:	61	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050h:	9A	01	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000060h:	40	00	00	C0	2E	72	64	61	74	61	00	00	00	00	00	00
00000070h:	00	00	00	00	OE	00	00	00	9A	01	00	00	00	00	00	00
00000080h:	00	00	00	00	00	00	00	00	40	00	00	40	2E	69	64	61
00000090h:	74	61	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000A0h:	A8	01	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000B0h:	40	00	00	C0	2E	62	73	73	00	00	00	00	00	00	00	00
000000C0h:	00	00	00	00	00	00	00	00	A8	01	00	00	00	00	00	00
000000D0h:	00	00	00	00	00	00	00	00	80	00	00	C0	2E	72	65	6C
000000e0h:	00	00	00	00	00	00	00	00	00	28	00	00	00	00	00	00
000000f0h:	A8	01	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000100h:	00	08	00	40	2E	73	79	6D	74	61	62	00	00	00	00	00
00000110h:	00	00	00	90	00	00	00	D0	01	00	00	00	00	00	00	00
00000120h:	00	00	00	00	00	00	00	00	08	00	40	2E	73	74	72	00
00000130h:	74	61	62	00	00	00	00	00	00	00	00	2B	00	00	00	00
00000140h:	60	02	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000150h:	00	08	00	40	55	89	E5	81	EC	00	00	00	B8	00	00	00
00000160h:	00	00	50	E8	FC	FF	FF	83	C4	04	B8	00	00	00	00	00
00000170h:	E9	00	00	00	00	8B	E5	5D	C3	55	89	E5	81	EC	04	00
00000180h:	00	00	E8	FC	FF	FF	89	45	FC	88	45	FC	50	E8	FC	..
00000190h:	FF	FF	FF	83	C4	04	8B	E5	5D	C3	48	65	6C	6C	6F	20
000001a0h:	57	6F	72	6C	64	21	0A	00	0A	00	00	03	00	00	00	00
000001b0h:	01	06	10	00	00	05	05	00	00	01	14	2F	00	00	00	00
000001c0h:	04	00	00	00	01	14	3B	00	00	07	00	00	01	14	00	00
000001d0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001e0h:	00	00	01	00	00	00	00	00	00	00	00	00	00	00	02	00
000001f0h:	00	00	03	00	07	00	00	00	00	00	00	00	00	00	00	00
00000200h:	05	00	00	03	00	0C	00	00	01	00	00	00	00	00	00	00
00000210h:	00	00	03	00	00	03	00	13	00	00	00	00	00	00	00	00
00000220h:	00	00	00	00	01	20	00	02	18	00	00	00	00	00	00	00
00000230h:	00	00	00	00	00	00	00	20	00	02	00	1F	00	00	00	00
00000240h:	00	00	00	25	00	00	00	01	00	20	00	02	00	26	00	00
00000250h:	00	00	00	00	00	00	00	00	00	00	00	00	20	00	02	00
00000260h:	00	2E	64	61	74	61	00	2E	62	73	73	00	2E	72	64	61
00000270h:	74	61	00	6D	61	69	6E	00	70	72	69	6E	74	66	00	5F
00000280h:	65	6E	74	72	79	00	65	78	69	74	00					entry.exit.

图 1.5 HelloWorld.obj 文件内容

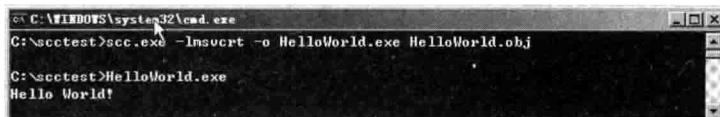


图 1.6 HelloWorld.obj 文件内容

打印出“Hello World!”。

表 1.2 就是 HelloWorld.exe 的文件内容,省略了部分全 0 的内容,这是我们见到的第二封天书,这封天书将在第 10 章进行全方位解读,请大家拭目以待。

表 1.2 HelloWorld.exe 文件内容

文件偏移	文件数据内容(十六进制表示)	数据代表的 ASCII 码字符
00000000h	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ?.....
00000010h	B8 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00	?.....@.....
00000020h	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000030h	00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00 €...
00000040h	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..? .??L?Th
00000050h	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
00000060h	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00000070h	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode.... \$

续表

1.2 SCC 编译器简介

1.2.1 SCC 编译器架构

通过上述 HelloWorld 编译过程分析,大家已经对 SCC 编译器有了大致了解,本节讨论 SCC 编译器整体架构,参见图 1.7。

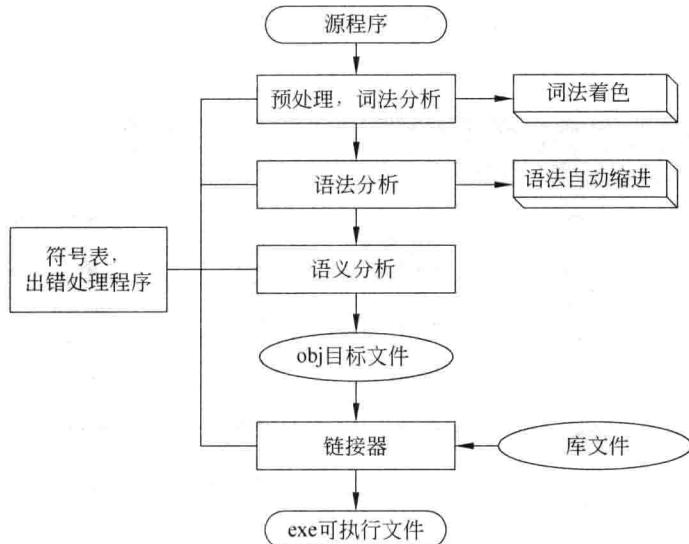


图 1.7 SCC 编译器架构

读者可能对这个图很熟悉,感觉没有什么新意,每本讲编译原理的书中都会有这个编译过程图。但大多是纸上谈兵,本书可是要实现图 1.1 中所有功能。有了前面对 HelloWorld 编译过程的分析,上面的 SCC 编译器架构也很容易理解,所以这里不多解释。给出这个图的目的,可用古人的话来表达为“不谋万世者,不足谋一时;不谋全局者,不足谋一域。”,SCC 编译器实现过程是一项复杂的、整体的过程,各个阶段既相对独立,又紧密相关,要求在每个阶段的程序设计上要考虑后续阶段能够方便使用。

1.2.2 SCC 编译器开发环境

SCC 编译器是在 Windows 操作系统中,使用 Visual Studio 6.0 中的 Visual C++ 6.0 开发的,读者可能会问为什么不用 Visual Studio .Net 呢? Visual Studio 6.0 虽然是 Microsoft 公司开发环境的老版本,但是鉴于其后继版本的主要功能变化都是为了支持 .Net 平台,并且安装后身躯庞大,体态臃肿,所以对于开发非. Net 平台的程序,这个经典稳定的开发环境仍然是首选。

有必要对 Visual Studio 6.0 开发环境做一个简单介绍,以便读者对 Visual Studio 6.0 与 Visual C++ 6.0 的关系有个清晰的认识。Visual Studio 6.0 是微软公司在 1998 年前后推出的一个编程组件,Visual Studio 6.0 中含有 Visual Basic 6.0、Visual C++ 6.0、Visual J++ 6.0、Visual FoxPro 6.0、Visual SourceSafe 6.0 等,而 SCC 编译器开发只用到了其中

的 Visual C++ 6.0。

Visual C++ 6.0 IDE(Integrated Development Environment, 即集成开发环境)界面如图 1.6 所示。IDE 是用于程序开发环境的应用程序,一般包括代码编辑器、编译器、链接器、调试器和图形用户界面工具等,是集成了代码编写功能、分析功能、编译功能、调试功能等一体化的开发软件套件。人们习惯上把 IDE 称为编译器,这个称呼有些名不符实,并且会形成误导。其实在 Visual C++ 6.0 IDE 中,编译时 IDE 会安排 CL. EXE 来编译,当链接时 IDE 会指挥 LINK. EXE 来链接,这时 IDE 整个就是一甩手掌柜,被称作编译器完全是“浪得虚名”。CL. EXE 与 LINK. EXE 才是真正的幕后英雄,通过图 1.8 可看到在 IDE 背后一直默默奉献的两位“无名英雄”。



图 1.8 VC6 中的编译器和链接器

讲完 Visual C++ 6.0 IDE, 这里讲一下 SCC 编译器使用的开发语言, VC6 编译器支持 C 语言及 C++ 语言的编译, 当源文件后缀为. c 时按 C 语言来编译, 当源文件后缀为. cpp 时, 按 C++ 来编译, SCC 编译器则完全使用 C 语言来开发。

1.2.3 SCC 编译器运行环境

这里说一下 SCC 编译器的运行环境,主要包括支持的处理器和操作系统两方面。Intel 80x86 平台和 Windows 是桌面计算机上最流行的配置仍是不争的事实,所以 SCC 编译器目前只支持 Windows 操作系统,处理器只支持兼容 Intel x86 指令集的处理器。

这里介绍一下 x86 指令集处理器,Intel 8086/8088/80186/80286 CPU 都为 16 位处理器,在市面上的 PC 中这些很多年前就已经销声匿迹,所以 SCC 编译器将不支持这些 16 位处理器的指令系统。1985 年,Intel 公司正式公布了 32 位处理器 80386,它采用 32 位指令系统,有 32 条地址线。80386 处理器在设计的时候考虑了多用户及多任务的需要,在芯片中增加了保护模式、优先级、任务切换和片内的存储单元管理等硬件单元。直到现在,运行于 80x86 处理器之上的多任务操作系统都是以 80386 的运行模式为基础的。本书中,x86 指兼容 80386 指令集的处理器。

从 80386 开始,在 Intel 公司向市场大量推出处理器芯片的同时,其他一些电脑公司和厂商如 AMD 和 Cyrix 等,也纷纷投入大量的人力财力进行处理器的开发和研制,并很快把研制出的产品推向市场。这些 CPU 芯片和 80386 芯片兼容,在编程上可以使用与 Intel 处理器相同的指令集。Intel 公司后来推出的 80486 及奔腾、赛扬、酷睿系列 CPU 都兼容 80386 指令集。

所谓 SCC 支持的操作系统有两层含义:一是 SCC 编译器所运行的操作系统;二是用 SCC 编译器编译生成的可执行文件所运行的操作系统。SCC 编译器所运行的操作系统及生成的可执行文件所运行的操作系统皆为 Win 32 操作系统或者可以兼容运行 Win 32 应用程序的操作系统,即 PC 上装的 Windows 2000、Windows XP、Windows Server 2003、Windows Vista、Windows 7、Windows 8 都支持。

第 2 章

文 法 知 识

宜未雨而绸缪，毋临渴而掘井。

——朱柏庐

在正式开始编写编译器之前，需要学习一点编译原理的基础知识，这里不会像《编译原理》书籍那样长篇大论面面俱到地讲那些枯燥的理论。本书对理论知识讲授本着够用就行的原则，对于不好理解的知识，还会附以生动形象的例子帮助理解。

在正式介绍文法的知识之前，先来看一下西天取经团队成员的文法定义，以便对文法有个感性认识。

- ① <西天取经团队成员> ::= <师父> | <徒弟成员>
- ② <师父> ::= "唐僧"
- ③ <徒弟成员> ::= "孙悟空" | "猪八戒" | "沙和尚" | "白龙马"

不用多解释，大家也知道上面文法的含义吧。西天取经团队成员是师父或徒弟，师父是“唐僧”，徒弟是“孙悟空”、“猪八戒”、“沙和尚”或“白龙马”。

问大家一个问题，西天取经团队成员中，有一位他的名字中第一字是“孙”，问这位成员是谁？读者可能会说，这么弱智的问题还好意思拿出来问，当然是“孙悟空”。

再举一个例子：

<陈述句> ::= <陈述句内容>。

再问大家一个问题，根据<陈述句>的方法定义，<陈述句内容>以什么结尾？当然是以句号结尾。

如果上述两个问题你都答对了，那么恭喜你，你读这本书的词法分析和语法分析部分，将不会遇到太大的困难，因为本书的词法语法部分，用到的就是这个原理。后面文绉绉的对First 集、Follow 集的定义，其实描述的就是这点事。请大家带着如下两个问题来阅读本章内容，第一如何定义一门语言，第二如何对一门语言进行词法分析、语法分析。

2.1 语 言 概 述

语言是由句子组成的集合，是由一组符号所构成的集合。汉语是所有符合汉语语法的句子的全体。英语是所有符合英语语法的句子的全体。程序设计语言是所有符合该语言语法定义的程序的全体。

语言有两方面来构成：语法和语义。语法表示构成语言句子的各个记号之间的组合规律，语义表示按照各种表示方法所表示的各个记号的特定含义，即各个记号和记号所表示的对象之间的关系。每种语言具有两个可识别的特性，即语言的形式和该形式相关联的意义。

下面以<句子>的定义为例来说明一下语法与语义的关系。

- ① <句子> ::= <主语><谓语><宾语>
- ② <主语> ::= <名词>
- ③ <宾语> ::= <名词>
- ④ <谓语> ::= <动词>
- ⑤ <名词> ::= "人" | "狗" | "花" | "鸟" | "虫" | "鱼" | "水"
- ⑥ <动词> ::= "吃" | "喝" | "打" | "咬" | "睡觉" | "游泳" | "开"

根据上面的语法定义,如果说“人打狗”,“狗喝水”,“花开”,“鸟睡觉”,“虫咬人”,“鱼游泳”这些句子符合语法,语义上也没问题。但是如果说“狗打人”,“水喝狗”,“花咬鱼”,“水睡觉”,“虫游泳鸟”,“鱼打狗”,这些句子语法确实没有问题,但是语义上非常荒谬。

下面给出语法与语义关系的官方描述:语法能够描述程序设计语言的大部分语法但不是全部,例如,标识符的先声明后使用无法用上下文无关文法描述。因此,语法分析器接受的语言是程序设计语言的超集。必须通过语义分析来剔除一些符合文法、但不合法的程序。语言是语义和语法的统一,语法结构是外表,语义结构是内在。

上述官方描述,现在有些不太理解没关系,等读完本章就完全理解了。

2.2 形式语言

如果不考虑语义,即只从语法这一侧面来看语言,这种意义上的语言称作形式语言。形式语言抽象地定义为一个数学系统。“形式”是指这样的事实:语言的所有规则只以什么符号串能出现的方式来陈述。形式语言理论是对符号串集合的表示法、结构及其特性的研究,是程序设计语言语法分析研究的基础。

2.2.1 字母表和符号串

正如英语是由句子组成的集合,而句子又是由单词和标点符号组成的序列那样。SC 程序设计语言,是由一切 SC 程序所组成的集合,而 SC 程序是由 if、else、for 等关键字符,+, -、*、/ 等运算符,;、,、{、} 等分隔字符,字母数字及下划线组成的标识符等基本符号所组成。从字面上看,每个程序都是一个“基本符号”串,设有一基本符号集,那么 SC 语言可看成在这个基本符号集上定义的,按一定规则构成的一切基本符号串组成的集合,因此有必要将有关符号串的一些概念做一下介绍,作为文法和语言的形式定义的预备知识。

字母表: 字母表是元素的非空有穷集合,把字母表中的元素称为符号,因此字母表也称为符号集。不同语言可以有不同的字母表,例如汉语的字母表中包括汉字、数字及标点符号等。这里读者可要注意了,“字母表”可不要只机械地理解成英文字母表的 26 个英文字母。对于 SC 语言定义来说,语法定义时的字母表是指经词法分析识别出的一个个单词符号,如 if、else、for、+、-、*、/、;、,、{、} 等。词法分析的字母表则是 a~z、A~Z、_、+、-、*、/ 等源码字符。

符号: 字母表中的元素,语法分析时指一个个单词,词法分析指一个个源码字符。

符号串: 由字母表中的符号组成的任何有穷序列称为符号串。语法分析时 int abc=1 代表一个符号串。词法分析时 abc 就代表一个符号串。在符号串中,符号的顺序是很重要的,例如符号串 abc 就不同于 cba。可以使用字母表示符号串,如 x=abc 表示“x 是由符号

a、b 和 c，并按此顺序组成的符号串”。

空字符串：无任何符号的字符串或长度为零的字符串，记为 ϵ 。

字符串集合：若集合 A 中的一切元素都是某字母表上的字符串，则称 A 为该字母表上的字符串集合。

字符串相等：若 x, y 是集合上的两个字符串，则 $x = y$ 当且仅当组成 x 的每一个符号和组成 y 的每一个符号依次相等。

字符串的长度： x 为字符串，其长度 $|x|$ 等于组成该字符串的字符个数。

例： $x = abc, |x| = 3$

字符串的连接：若 x, y 是定义在 Σ 上的字符串，且 $x = XY, y = YX$ ，则 x 和 y 的连接 $xy = XYYX$ 也是 Σ 上的字符串。注意，一般 $xy \neq yx$ ，而 $\epsilon x = x\epsilon$ 。

字符串集合的乘积运算：令 A, B 为字符串集合，定义

$$AB = \{xy \mid x \in A, y \in B\}$$

字符串集合的幂运算：有字符串集合 A ，定义

$$A_0 = \{\epsilon\}, A_1 = A, A_2 = AA, A_3 = AAA, \dots, A_n = A_{n-1}A = AA_{n-1}, n > 0$$

字符串集合的闭包运算：设 A 是字符串集合，定义

$A^+ = A_1 \cup A_2 \cup A_3 \cup \dots \cup A_n \cup \dots$ 称为集合 A 的正闭包。

$A^* = A_0 \cup A^+$ 称为集合 A 的闭包。

例： $A = \{x, y\}$

$$A^+ = \{x, y, xx, xy, yx, yy, xxx, xxy, xyx, xyy, \dots\}$$

$$A_1 \quad \quad \quad A_2 \quad \quad \quad A_3$$

$$A^* = \{\epsilon, x, y, xx, xy, yx, yy, xxx, xxy, xyx, xyy, \dots\}$$

$$A_0 \quad A_1 \quad , \quad A_2 \quad \quad \quad A_3$$

若 A 为 SC 语言的基本字符集

$$A = \{a, b, \dots, z, A, B, \dots, Z, 0, 1, \dots, 9, +, -, *, /, (,), =, \dots\}$$

B 为 SC 语言单词集

$$B = \{\text{char, short, if, for, break, \dots, abc, x1, y1, \dots}\}$$

则 $B \subset A^*$ 。

SC 语言程序是定义在 B 上的字符串。若令 C 为 SC 语言程序集合，则 $C \subset B^*$ ，程序 $\subset C$ 。

2.2.2 文法与语言的形式定义

文法是对语言结构的定义与描述，即从形式上用于描述和规定语言结构的称为文法（或称为“语法”）。

当表述一种语言时，就是要说明这种语言的句子。如果语言只含有有穷多个句子，则只需列出句子的有穷集。如果语言含有无穷多个句子，则存在如何给出它的有穷表示的问题。这需要一种规则，用这些规则来描述语言的结构，可以把这些规则看成一种元语言，这些规则就称为文法。下面给出文法的定义。进而在文法定义的基础上，给出推导的概念，句型、句子和语言的定义。

定义 文法 G 定义为四元组 (V_N, V_T, P, S) 。其中 V_N 为非终结符（是可以被取代的符

号)集; V_T 为终结符(语言中用到的基本元素,不能再被分解成更小的单位)集; P 为产生式(也称规则)的集合; V_N, V_T 和 P 是非空有穷集。 S 称作识别符号或开始符号,它是一个非终结符,至少要在一条产生式中作为左部出现。

V_N 和 V_T 不含公共的元素,即 $V_N \cap V_T = \emptyset$ 。通常用 V 表示 $V_N \cup V_T$, V 称为文法 G 的字母表或字汇表。其中产生式,是形如 $\alpha \rightarrow \beta$ 或 $\alpha ::= \beta$ 的 (α, β) 有序对,其中 α 是字母表 V 的正闭包 V^+ 中的一个符号, β 是 V^* 中的一个符号。 α 称为规则的左部, β 称为规则的右部。

为定义文法所产生的语言,还需要引入推导的概念,即定义 V^* 中的符号之间的关系:直接推导 \Rightarrow 、长度为 $n(n \geq 1)$ 的推导 $\stackrel{*}{\Rightarrow}$ 和长度为 $n(n \geq 0)$ 的推导 $\stackrel{*}{\Rightarrow}$ 。

定义 如 $\alpha \rightarrow \beta$ 是文法 $G = (V_N, V_T, P, S)$ 的规则(或说是 P 中的一产生式), γ 和 δ 是 V^* 中的任意符号,若有符号串 v, w 满足:

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

则说 v (应用规则 $\alpha \rightarrow \beta$)直接产生 w ,或者说, w 是 v 的直接推导,也可以说, w 直接归约到 v ,记作 $v \Rightarrow w$ 。

定义 如果存在直接推导的序列:

$$v = w_0 \Rightarrow w_1 \Rightarrow w_2 \cdots \Rightarrow w_n = w, (n > 0)$$

则称 v 推导出(产生) w (推导长度为 n),或称 w 归约到 v 。记作 $v \stackrel{*}{\Rightarrow} w$ 。

定义 若有 $v \stackrel{*}{\Rightarrow} w$,或 $v = w$,则记作 $v \stackrel{*}{\Rightarrow} w$ 。

定义 设 $G[S]$ 是一文法,如果符号串 x 是从识别符号推导出来的,即有 $S \stackrel{*}{\Rightarrow} x$,则称 x 是文法 $G[S]$ 的句型。若 x 仅由终结符号组成,即 $S \stackrel{*}{\Rightarrow} x, x \in V_T^*$,则称 x 为 $G[S]$ 的句子。

定义 文法 G 所产生的语言定义为集合 $\{x \mid S \stackrel{*}{\Rightarrow} x\}$,其中 S 为文法识别符号,且 $x \in V_T^*$ 。可用 $L(G)$ 表示该集合。

从这个定义看出两点:第一,符号串 x 可从识别符号推出,也即 x 是句型。第二, x 仅由终结符号组成,即 x 是文法 G 的句子。也就是说,文法描述的语言是该文法一切句子的集合。

定义 若 $L(G_1) = L(G_2)$,则称文法 G_1 和 G_2 是等价的。也就是说,如果两个文法定义的语言一样,则称这两个文法是等价的。

定义 如果在推导的任何一步 $v \Rightarrow w$,其中 v, w 是句型,都是对 v 中的最左(最右)非终结符进行替换,则称这种推导为最左(最右)推导。

2.2.3 文法与语言的类型

1956 年,乔姆斯基(Chomsky)建立了形式语言理论,这种理论对计算机科学有着深刻的影响,特别是对程序设计语言的设计、编译方法和计算复杂性等方面更有重大的作用。

2.2.3.1 文法分类

乔姆斯基把文法分成 4 种类型,即 0 型、1 型、2 型和 3 型。这几类文法的差别在于对产生式施加不同的限制。下面看一下 4 种类型文法的定义。

0 型文法: 设 $G = (V_N, V_T, P, S)$,如果它的每个产生式 $\alpha \rightarrow \beta$ 是这样一种结构: $\alpha \in (V_N \cup V_T)^*$ 且至少含有一个非终结符,而 $\beta \in (V_N \cup V_T)^*$,则 G 是一个 0 型文法。一个非常重要的理论结果是: 0 型文法的能力相当于图灵机(Turing)。或者说,任何 0 型文法语言都

是递归可枚举的；反之，递归可枚举集必定是一个 0 型语言。0 型文法对文法规则的表示形式不作任何限制，从而能使定义的语言提供充分的描述功能。但 0 型文法不保证语言的递归性，即不能确保语句合法的可判性，所以很少用于定义自然语言。

1型文法：1型文法也称上下文有关文法，其可描述的语言为上下文有关语言，其对应的识别器为线性有界自动机。它是在 0 型文法的基础上，规定对每一个 $\alpha \rightarrow \beta$ ，都有 $|\beta| \geq |\alpha|$ 。这里的 $|\beta|$ 表示的是 β 的长度。自然语言是上下文有关的语言，文法规则允许其左部有多个符号（至少包括一个非终结符），以指示上下文相关性；但要求规则右部符号的个数不少于左部，以确保语言的递归性（即语句合法的可判性）。

2型文法：2型文法也称上下文无关文法，其可描述的语言为上下文无关语言，其对应的识别器为下推自动机。下推自动机比下面讲到的识别 3 型方法的有限状态自动机复杂：除了有限状态组成部分外，还包括一个长度不受限制的栈。2型文法是在 1型文法的基础上，再满足：每一个 $\alpha \rightarrow \beta$ 都有 α 是非终结符。如 $A \rightarrow Ba$ ，符合 2型文法要求。如 $Ab \rightarrow Bab$ 虽然符合 1型文法要求，但不符合 2型文法要求，因为其 $\alpha = Ab$ ，而 Ab 不是一个非终结符。上下文无关文法有足够的能力描述现今程序设计语言的语法结构，例如描述各种表达式、描述各种语句等。上下文无关文法及其语言已广泛应用于定义程序设计语言，这种文法的规则限定其左部只能是单一的非终结符，即非终结符通过文法规则的扩展性重写是相互独立的，不受其他符号的影响，所以称为上下文无关。

3型文法：3型文法也称正规文法，其可描述的语言为正则语言，其对应的识别器为有限状态自动机。它是在 2型文法的基础上满足： $A \rightarrow \alpha | \alpha B$ （右线性）或 $A \rightarrow \alpha | B\alpha$ （左线性）。可以看出，3型文法规则表示形式高度受限，使得正则语言可以用有限状态自动机程序作高效的句法分析。有限状态自动机有若干状态，其中必有一个为起始状态，并至少有一个结束状态；自动机的输入会导致状态变化，并在到达目标状态时停机。面向正则语言句法分析的有限状态自动机就以文法规则左部的非终结符指示当前状态，文法规则右部的终结符作为输入，终结符后的非终结符就是自动机将到达的下一状态；若终结符后无非终结符，则自动机在当前状态下停机。若输入结束且此时自动机处于结束状态，则输入就作为一个合法语句而接受；否则输入的是非法语句。尽管正则文法简单并易于分析，但文法规则的表示太受限制，使其无法用于描述哪怕是人工制定的语言。正则文法主要用于西文（如英语）的词法分析阶段，如切分单词和识别非法字符。

注意：上面例子中的大写字母表示的是非终结符，而小写字母表示的是终结符。

4 种类型文法的规则受限情况、相应的语言类型、相对应的识别器之间的关系如表 2.1 所示。

表 2.1 文法、语言及相应识别器之间的关系

乔姆斯基文法层次	文法别名	语言类型	规则限制	识别器
3型文法	正则文法	正则语言	左部必须是单一非终结符，右部必须是单一终结符或单一终结符后跟单一非终结符	有限状态自动机
2型文法	上下文无关文法	上下文无关语言	左部必须是单一非终结符	下推自动机
1型文法	上下文有关文法	上下文有关语言	左部至少包括一个非终结符，右部符号的个数不少于左部	线性有界自动机
0型文法	无限制文法	递归可枚举语言	无	图灵机

2.2.3.2 几类语言之间的关系

现在已经明白了上述 4 类文法相应代表的 4 种语言类型,那么这 4 种语言之间是什么关系呢?参见图 2.1。

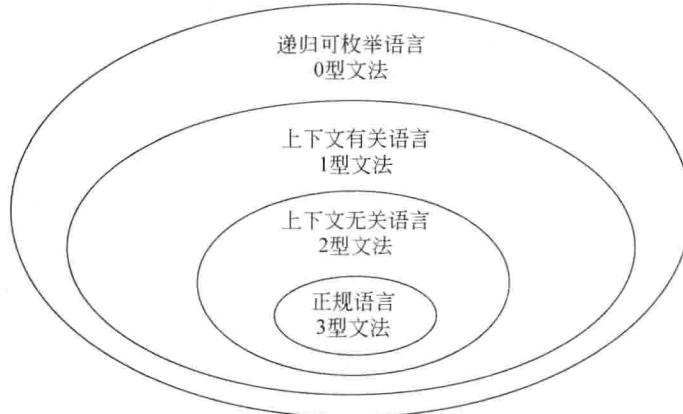


图 2.1 几类语言之间的关系

从图 2.1 可以看出,正规语言类包含于上下文无关语言类,上下文无关语言类包含于上下文有关语言类,上下文有关语言类包含于递归可枚举语言类。这里的包含都是集合的真包含关系,也就是说,存在递归可枚举语言不属于上下文有关语言类,存在上下文有关语言不属于上下文无关语言类,存在上下文无关语言不属于正规语言类。

2.2.4 程序设计语言描述工具

程序设计语言可以使用上下文无关文法或者巴科斯范式来描述,巴科斯范式的表达能力等价于上下文无关文法。上下文无关方法前面已经介绍过,下面来认识一下巴科斯范式。

巴科斯范式(Backus-Naur Form,BNF)是由 John Backus 和 Peter Naur 首次引入一种形式化符号来描述给定语言的语法(最早用于描述 ALGOL 60 编程语言)。BNF 类似一种数学游戏:从一个符号开始(称为起始标志,实例中常用 S 表示),然后给出替换前面符号的规则。BNF 语法定义的语言只不过是一个字符串集合,可以按照下述规则书写,这些规则称为书写规范(产生式规则),形式如下:

```
symbol ::= alternative1 | alternative2 ...
```

每条规则声明 ::= 左侧的符号必须被右侧的某一个可选项代替。BNF 只包括 3 种元符号,如表 2.2 所示。

表 2.2 BNF 元符号表

元 符 号	表 示 含 义	备 注
::=	定义为	有的书上用 ->, 或 :=
	或者	
< >	尖括号用于括起非终结符	

明白了 BNF 表示语法的书写规则,下面来看一下用 BNF 表示的有符号整数,

```

<有符号整数> ::= <无符号整数> | <正负号> <无符号整数>
<无符号整数> ::= <数字> | <无符号数> <数字>
<数字> ::= 0|1|2|3|4|5|6|7|8|9
<正负号> ::= + |-
```

不知道你对上面的<有符号整数>文法定义是否满意,感觉是否够简洁,是否容易理解,是否想过还有没有更好的定义方法?可能你还来不及想这些,我们先来看一下前人对BNF的看法:BNF有着可选项和重复不能直接表达的问题。作为替代,它们需要利用中介规则或两选一规则,对于可选项,定义要么是空的,要么是可选的产生式的规则;对于重复,递归的定义要么是被重复的产生式,要么是自身的规则。既然前人认识到了这些问题,那么这些问题有没有用更好的方法来解决呢?下面就看一下那些不安于现状的前辈为了解决这些问题又研究出了什么新鲜玩意儿。

扩展巴科斯范式(EBNF)是表达作为描述计算机编程语言和形式语言的正规方式的上下文无关文法的元语法符号表示法。它是基本巴科斯范式(BNF)元语法符号表示法的一种扩展。它最初由尼克劳斯·维尔特开发,最常用的EBNF变体由ISO-14977标准所定义。下面通过表2.3看一下EBNF的元符号及其含义。

表2.3 EBNF元符号表

EBNF元符号	含 义	EBNF元符号	含 义
::=	定义为,推导为	()	括号内看作一项
	或	.	一条生成规则的结束
{}	含0次在内任意多次重复	<>	非终结符
[]	含0次和1次	""	终结符

BNF原来只有3个元符号,EBNF成了8个元符号,想必EBNF比BNF功能强大多了。这里要特别提示一下,EBNF在定义语言方面并不比BNF更强大,只是更方便。凡是用EBNF写的东西都可以转换成BNF的形式。下面再看一下<有符号整数>的EBNF定义:

```

<有符号整数> ::= [<正负号>] <有符号整数>
<无符号整数> ::= <数字> {<数字>}
<数字> ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
<正负号> ::= "+"|"-"
```

用EBNF定义文法是不是比BNF更方便,也更容易理解。引入EBNF除了上述优点之外,另一个更重要原因是为更方便地把文法映射到递归下降分析程序的真实代码。当需要手动构造递归下降分析程序的时候,通常把上下文无关文法改写为EBNF是必需的。请大家务必记住上面这句话,如果现在不理解没关系,可以在后面的章节中逐步理解。

多数编程语言标准都使用EBNF来定义语言的语法,这样做的好处是在语言的语法上没有争议,而且有助于编译器的编写。

2.3 词法分析方法

为了设计词法分析器,可以根据正规文法定义建立状态转换图。状态转换图是个有向图,其中的结点表示状态,通常用整数或字母命名;每条有向边其标记表示从该边起点的状态到终点状态的转换规则。

态转换到终点的状态需要的条件。对应词法分析器规格说明中的每个正规定义,分别建立一个状态转换图,其中每条有向边的标记分别表示识别词汇时从输入缓冲区接收一个输入字符。开始识别一个词形时对应的状态,称为开始状态。识别完一个词形,词法分析器应该返回一个词汇时对应的状态,称为接收状态。从开始状态到接收状态形成的路径上,每条有向边的标记形成该状态转换图识别的词形。为了区分,对接收状态用带双圈的结点表示,其他状态都用带单圈的结点表示。下面以一个简单的例子来说明词法分析器是如何构造的。

2.3.1 词法定义例举

支持整数四则运算计算器的词法定义:

```
<IDENTIFIER> ::= <Alphabet> {<Alphabet>}
<INTEGER> ::= <Digit> {<Digit>}
<TK_PLUS> ::= "+" 
<TK_MINUS> ::= "-" 
<TK_STAR> ::= "*" 
<TK_DIVIDE> ::= "/" 
<TK_ASSIGN> ::= "="
```

2.3.2 状态转换图

图 2.2 为词法分析器的状态转换图。

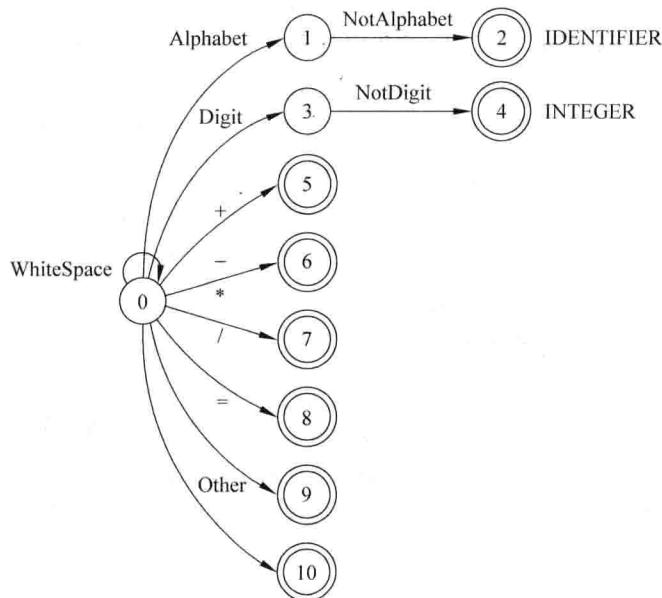


图 2.2 词法分析器的状态转换图

2.3.3 词法分析程序流程图

有了上述状态转换图,很容易就可以得到词法分析器的算法框图,请大家自己观察一下

状态转换图 2.2 与程序流程图 2.3 之间的关系。

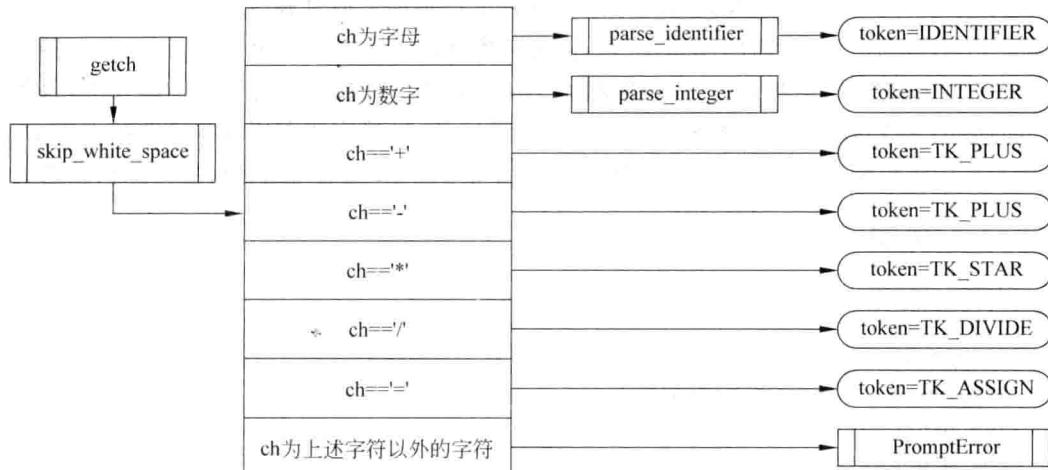


图 2.3 词法分析程序流程图

2.4 语法分析方法

语法分析的作用是识别由词法分析给出的单词符号序列是否是给定文法的正确句子(程序),目前语法分析常用的方法有自顶向下分析和自底向上分析两大类。所谓**自顶向下分析法**,是从文法的开始符号出发,反复使用各种产生式,寻找“匹配”于输入符号串的推导。**自底向上分析法**,则是从输入符号串开始,逐步进行“归约”到文法的开始符号。由于本书的语法分析方法采用的是自顶向下分析法,所以这里只介绍自顶向下分析法。

自顶向下分析法也称面向目标的分析方法,也就是从文法的开始符号出发企图推导出与输入的单词串完全相匹配的句子。若输入串是给定文法的句子,则必能推出,反之必然出错。自顶向下分析法对文法有一定的限制,但由于实现方法简单、直观,便于手工构造,因而仍是目前最常用的方法之一。自顶向下分析法又分为两类,不确定的自顶向下分析方法和确定的自顶向下分析方法。**不确定的自顶向下分析方法**是带回溯的分析,“不确定”的意思:当某个非终结符的产生式有多个候选,而面临当前的输入符号无法确定选用哪个产生式,从而引起回溯。这种方法实际上是一种穷举的试探方法,因此效率低,代价高,因而极少使用。**确定的自顶向下分析方法**,是从文法的开始符号出发,考虑如何根据当前的输入符号(单词符号)唯一地确定选用哪个产生式替换相应非终结符以下推导。确定的自顶向下分析方法:对文法有一定的限制,但实现方法简单、直观,便于手工构造或自动生成语法分析器,目前仍是常用的方法之一。

2.4.1 LL 分析器

LL 分析器是一种自顶向下的上下文无关语法分析器。第一个 L 是指 Left to right,即从左至右扫描输入串,第二个 L 是指 Leftmost derivation,即分析过程中将使用最左推导,能以此方法分析的语法称为 LL 语法。

一个 LL 分析器若被称为 LL(k) 分析器, 表示它向前看 k 个符号才可决定如何推导, 即选择哪个产生式(规则)进行推导。若对某个语法而言, 存在一个分析器可以在不用回溯的情况下处理这个语法, 则这个语法称为 LL(k) 语法。在这些语法中较严格的 LL(1) 语法相当受欢迎, 由于它的分析器只需要多看一个符号就可以产生分析结果。那些需要很大的 k 才能产生分析结果的编程语言, 在分析时的需求也较高。

下面首先举例说明 LL(k) 文法, 然后研究一下适合用确定的自顶向下分析方法分析的文法需要具备的条件。

2.4.2 LL(k) 文法

我们先来看一下 LL(0) 文法, 它根本不需要检查终结符, 总能够选择适当的产生式。这种情况只发生在所有符号只有一个替换符的情形, 而如果只有一个替换符就意味着语言只有一个字符串, 也就是说 LL(0) 没有意义。下面介绍一个 LL(0) 文法的例子:

```
<光杆司令> ::= <军长>
<军长> ::= <师长>
<师长> ::= <旅长>
<旅长> ::= <团长>
<团长> ::= <营长>
<营长> ::= <连长>
<连长> ::= <排长>
<排长> ::= <班长>
<班长> ::= <士兵>
<士兵> ::= "王老五"
```

上面是个<光杆司令>的文法定义, 由于每个产生式右边只有一个产生式, 闭着眼睛都不会选错, 所以是个 LL(0) 文法, 但是这个<光杆司令>的定义实在有些无聊, 一点实际意义都没有。

再来看一个 LL(1) 文法的例子:

```
<三国桃园结义成员> ::= "刘备" | "关羽" | "张飞"
```

只要向前看一个字符, 是“刘”、“关”或是“张”, 就可以从右侧 3 个可选项中做出正确选择。

再来看一个 LL(2) 文法的例子:

```
<某班级成员> ::= "张三" | "李四" | "王五" | "王一"
```

如果向前看一个字符, 不能保证在右侧 4 个选项中选择绝对正确, 例如已经知道某一位班级成员姓“王”, 那么这个成员可能是“王五”, 也可能是“王一”, 也就是说, 最不利的情况下必须向前看两个字符才能保证在上述 4 个选项中做出正确选择。

其他 LL(k) 文法的例子, 依此类推。

2.4.3 LL(1) 文法

LL(1) 文法在实际当中应用最多, 所以这里重点研究一下 LL(1) 文法。LL(1) 的含义

是：第一个 L 表明自顶向下分析是从左向右扫描输入串，第 2 个 L 表明分析过程中将使用最左推导，1 表明只须向右看一个符号便可决定如何推导，即选择哪个产生式（规则）进行推导。

2.4.3.1 LL(1)文法的判别

LL(1)文法的判别需要依次计算 FIRST 集、FOLLOW 集和 SELECT 集，然后判断是否为 LL(1)文法，最后再进行句子分析。

定义 设 $G = (V_N, V_T, P, S)$ 是上下文无关文法 $\text{FIRST}(\alpha) = \{a \mid \alpha \xrightarrow{*} a\beta, a \in V_T, \alpha, \beta \in V^*\}$ ，若 $\alpha \xrightarrow{*} \epsilon$ ，则规定 $\epsilon \in \text{FIRST}(\alpha)$ ，称 $\text{FIRST}(\alpha)$ 为 α 的开始符号集或首字符集。

定义 设 $G = (V_N, V_T, P, S)$ 是上下文无关文法， $A \in V_N$ ， S 是开始符号， $\text{FOLLOW}(A) = \{a \mid S \xrightarrow{*} \mu A\beta, \text{且 } a \in V_T, a \in \text{FIRST}(\beta), \mu \in V_T^*, \beta \in V^*\}$ ，若 $S \xrightarrow{*} \mu A\beta$ ，且 $\beta \xrightarrow{*} \epsilon$ ，则 $\# \in \text{FOLLOW}(A)$ 。这里用 '#' 作为输入串的结束符。

定义 给定上下文无关文法的产生式 $A \xrightarrow{*} \alpha, A \in V_N, \alpha \in V^*$ ，若 $\alpha \nRightarrow \epsilon$ ，则 $\text{SELECT}(A \xrightarrow{*} \alpha) = \text{FIRST}(A)$ 。如果 $\alpha \xrightarrow{*} \epsilon$ ，则 $\text{SELECT}(A \xrightarrow{*} \alpha) = (\text{FIRST}(\alpha) - \{\epsilon\}) \cup \text{FOLLOW}(A)$ 。

定义 一个上下文无关文法是 LL(1) 文法的充分必要条件是，对每个非终结符 A 的两个不同产生式， $A \xrightarrow{*} \alpha, A \xrightarrow{*} \beta$ ，满足 $\text{SELECT}(A \xrightarrow{*} \alpha) \cap \text{SELECT}(A \xrightarrow{*} \beta) = \emptyset$ 。其中 α, β 同时能 $\xrightarrow{*} \epsilon$ 。

2.4.3.2 计算 FIRST 集

根据 FIRST 集定义对每一文法符号 $X \in V$ 计算 $\text{FIRST}(X)$ ：

(1) 若 $X \in V_T$ ，则 $\text{FIRST}(X) = \{X\}$ 。

(2) 若 $X \in V_N$ ，且有产生式 $X \xrightarrow{*} a \cdots, a \in V_T$ ，则 $a \in \text{FIRST}(X)$ 。

(3) 若 $X \in V_N, X \xrightarrow{*} \epsilon$ ，则 $\epsilon \in \text{FIRST}(X)$ 。

(4) 若 $X \in V_N; Y_1, Y_2, \dots, Y_i \in V_N$ ，且有产生式 $X \xrightarrow{*} Y_1 Y_2 \cdots Y_n$ ；当 $Y_1 Y_2 \cdots Y_{i-1}$ 都 $\xrightarrow{*} \epsilon$ 时，(其中 $1 \leq i \leq n$)，则 $\text{FIRST}(Y_1) \cup \text{FIRST}(Y_2) \cup \dots \cup \text{FIRST}(Y_{i-1})$ 的所有非 $\{\epsilon\}$ 元素和 $\text{FIRST}(Y_i)$ 都包含在 $\text{FIRST}(X)$ 中。

(5) 当(4)中所有 $Y_i \xrightarrow{*} \epsilon, (i=1, 2, \dots, n)$ ，则

$$\text{FIRST}(X) = \text{FIRST}(Y_1) \cup \text{FIRST}(Y_2) \cup \dots \cup \text{FIRST}(Y_n) \cup \{\epsilon\}$$

反复使用上述(4)~(5)步直到每个符号的 FIRST 集合不再增大为止。

2.4.3.3 计算 FOLLOW 集

根据 FOLLOW 集定义，对文法中每一个文法符号 $A \in V_N$ ，计算 $\text{FOLLOW}(A)$ ：

(1) 设 S 为文法中开始符号，把 $\{\#\}$ 加入 $\text{FOLLOW}(S)$ 中。

(2) 若 $A \xrightarrow{*} \alpha B \beta$ 是一个产生式，则把 $\text{FIRST}(\beta)$ 的非空元素加入 $\text{FOLLOW}(B)$ 中。

如果 $\beta \xrightarrow{*} \epsilon$ 则把 $\text{FOLLOW}(A)$ 也加入 $\text{FOLLOW}(B)$ 中。

(3) 反复使用(2)直到每个非终结符的 FOLLOW 集不再增大为止。

2.4.3.4 计算 SELECT 集

根据 SELECT 集定义,对文法中每一个形如 $A \rightarrow \alpha$ 产生式,计算 $\text{SELECT}(A \rightarrow \alpha)$:

- (1) 求 $\text{FIRST}(\alpha)$;
- (2) 若 $\epsilon \notin \text{FIRST}(\alpha)$,则令 $\text{SELECT}(A \rightarrow \alpha) = \text{FIRST}(\alpha)$,否则求 $\text{FOLLOW}(A)$,并令 $\text{SELECT}(A \rightarrow \alpha) = \text{FIRST}(\alpha) \cup \text{FOLLOW}(A)$ 。

下面给出一个计算 FIRST 集、FOLLOW 集、SELECT 集的实例,设有文法:

```

E → TE'
E' → + TE' | ε
T → FT'
T' → * FT' | ε
F → i | ( E )

```

求其 3 种集合。

解:

```

FIRST(E) = FIRST(T) = FIRST(F) = {i, ()}
FIRST(E') = {+, ε}
FIRST(T') = {* , ε}
FOLLOW(E) = FOLLOW(E') = {(), #}
FOLLOW(T) = FOLLOW(T') = {+, ), #}
FOLLOW(F) = {* , + , ), #}
SELECT(E → TE') = FIRST(T) = {i, ()}
SELECT(E' → + TE') = FIRST(+ TE') = {+}
SELECT(E' → ε) = FIRST(ε) ∪ FOLLOW(E') = {ε, ), #}
SELECT(T → FT') = FIRST(F) = {i, ()}
SELECT(T' → * FT') = FIRST(* FT') = {*}
SELECT(T' → ε) = FIRST(ε) ∪ FOLLOW(T') = {ε, + , ), #}
SELECT(F → i) = FIRST(i) = {i}
SELECT(F → ( E )) = FIRST(( E )) = {(())

```

2.4.4 递归子程序法

确定的自顶向下分析方法分为两种,一种是递归子程序法;另一种是预测分析法。递归子程序法是比较简单直观且易于手工构造的一种语法分析方法,本书 SCC 编译器语法分析采用的就是递归子程序法。它要求文法满足 LL(1) 文法,它的实现思想是对文法中每个非终结符编写一个递归过程,每个过程的功能是识别由该非终结符推出的串,当某非终结符的产生式有多个候选时能够按 LL(1) 形式可唯一地确定选择某个候选进行推导。由于递归子程序法对每个过程可能存在直接或间接递归调用,所以对某个过程在退出之前可能又被调用,因此有些信息需要保留,通常在入口时要保留某些信息,出口时需恢复。由于递归过程是遵循先进后出规律,所以通常开辟先进后出栈来处理。下面以一个简单的例子来说明如何利用递归子程序法来构造语法分析程序,这个例子包括 3 部分内容:语法定义、语法描述图和语法分析程序。

2.4.4.1 语法定义举例

```

<program> ::= <statement> { statement } "#" 
<statement> ::= <expression> "\n" 
<expression> ::= <multiplicative_expression> { 
    "+" <multiplicative_expression> 
    | "-" <multiplicative_expression> } 
<multiplicative_expression> ::= <primary_expression> { 
    "* " <primary_expression> 
    | "/" <primary_expression> } 
<primary_expression> ::= <INTEGER> | "(" <expression> ")" 
  
```

2.4.4.2 语法描述图

本例的语法描述图如图 2.4 所示。

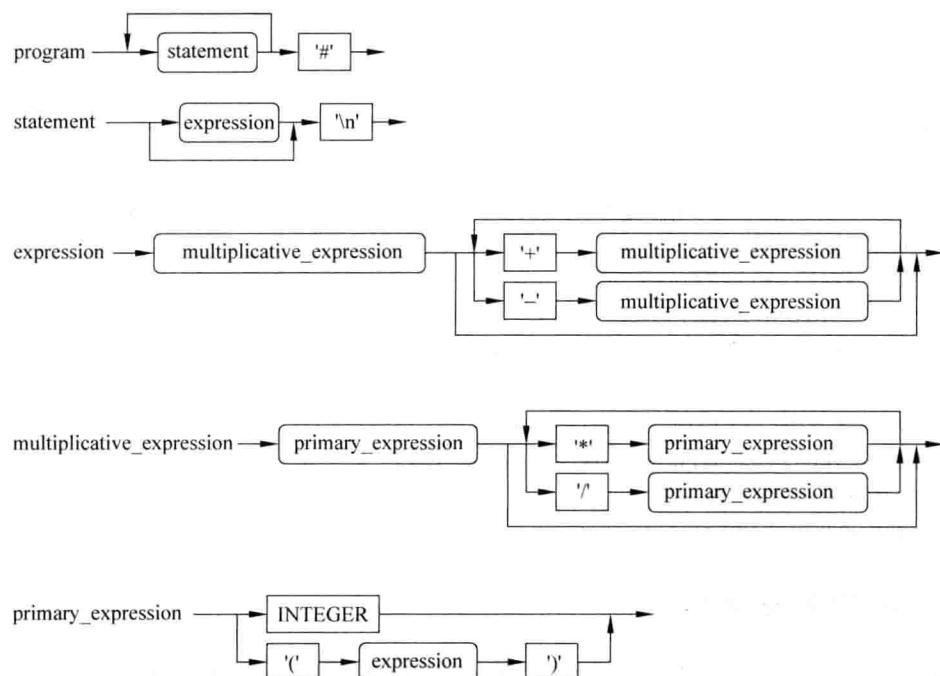


图 2.4 语法描述图

2.4.4.3 构造语法分析程序

```

void main()
{
    get_token();
    program();
}
void program()
  
```

```
{  
    while(token != '#')  
    {  
        statement();  
    }  
}  
  
void statement()  
{  
    if (token != '\n')  
    {  
        expression();  
    }  
    else  
    {  
        get_token();  
    }  
}  
  
void expression()  
{  
    multiplicative_expression();  
    if (token=='+' || token=='-')  
    {  
        get_token();  
        multiplicative_expression();  
    }  
}  
  
void multiplicative_expression()  
{  
    primary_expression();  
    if (token=='*' || token=='/')  
    {  
        get_token();  
        primary_expression();  
    }  
}  
  
void primary_expression()  
{  
    primary_expression();  
    if (token==INTEGER)  
    {  
        get_token();  
    }  
}
```

```

    }
    else if (token=='(')
    {
        get_token();
        expression();
        get_token();
        if (token==')')
            error("缺少右括号");
    }
    else
    {
        error("报错");
    }
}

```

请大家自己观察思考一下语言的文法定义、文法描述图和递归子程序法构造的语法分析程序之间的关系,第5章语法分析用到的就是这部分知识。

2.4.5 文法的等价变换

确定的自顶向下分析法要求给定语言的文法必须是 LL(1)形式,然而,不一定每个语言都是 LL(1)文法,对一个语言的非 LL(1)文法是否能变换为等价的 LL(1)形式以及如何变换是这里讨论的主要问题。由 LL(1)文法的定义可知若文法中含有左递归或含有左公共因子,则该文法肯定不是 LL(1)文法,因而,设法消除文法中的左递归,提取左公共因子对文法进行等价变换。

2.4.5.1 提取左公共因子

若文法中含有形如: $A \rightarrow \alpha\beta | \alpha\gamma$ 的产生式,这导致了对相同的产生式右部的 FIRST 集相交。即有 $\text{SELECT}(A \rightarrow \alpha\beta) \cap \text{SELECT}(A \rightarrow \alpha\gamma) \neq \emptyset$ 不满足 LL(1)文法的充要条件。

现将产生式 $A \rightarrow \alpha\beta | \alpha\gamma$ 等价交换为 $A \rightarrow \alpha(\beta | \gamma)$,可进一步引入非终结符 A' ,使产生式变换为 $A \rightarrow \alpha A', A' \rightarrow \beta | \gamma$ 。写成一般形式为 $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \dots | \alpha\beta_n$ 提取左公因子后变为 $A \rightarrow \alpha(\beta_1 | \beta_2 | \dots | \beta_n)$ 再引进非终结符 A' ,变为 $A \rightarrow \alpha A', A' \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$ 。若在 $\beta_i, \beta_j, \beta_k \dots$ 中仍含有左公共因子,可再进行提取,这样反复进行提取直到所引进的新非终结符的有关产生式均无左公共因子为止。

2.4.5.2 消除左递归

一个文法含有下列形式的产生式之一时:

- (1) $A \rightarrow A\beta, A \in V_N, \beta \in V^*$ 。
- (2) $A \rightarrow B\beta, B \rightarrow A\alpha, A, B \in V_N, \alpha, \beta \in V^*$ 。

则称该文法是左递归的。一个文法是左递归时,不能采取自顶向下分析法。为了使某些含有左递归的文法经过等价变换消除左递归后可能变为 LL(1)文法,可采取如下方法消除左递归。

① 消除直接左递归,把直接左递归改写为右递归,如对文法 $G: S \rightarrow Sa, S \rightarrow b$ 可改写为

$$S \rightarrow bS', \quad S' \rightarrow aS' | \epsilon$$

一般情况下,假定关于 A 的全部产生式是: $A \rightarrow A\alpha_1 | A\alpha_2 | \cdots | A\alpha_m | \beta_1 | \beta_2 | \cdots | \beta_n$, 其中, $\alpha_i (1 \leq i \leq m)$ 不等于 ϵ , $\beta_j (1 \leq j \leq n)$ 不以 A 开头, 消除直接左递归后改写为

$$A \rightarrow \beta_1 A' | \beta_2 A' | \cdots | \beta_n A', A' \rightarrow \alpha_1 A' | \alpha_2 A' | \cdots | \alpha_m A' | \epsilon$$

② 消除间接左递归。

对于间接左递归的消除需先将间接左递归变为直接左递归, 然后再按①消除直接左递归。

第3章

SC语言定义

合抱之木，生于毫末；九层之台，起于累土；千里之行，始于足下。

——老子

盖一栋大楼之前，要先绘出这幢大楼的蓝图来指导施工。搞社会主义现代化建设，要由社会主义现代化的宏伟蓝图来指引前进的方向。本章就要勾勒出 SC 语言的蓝图，来指导 SCC 编译器的开发。有了第 2 章关于文法知识的铺垫，本章要定义一下 SC 语言。以前用 C 语言写程序，只有老老实实地按 C 语言的语法规则写。现在我们由言听计从的规则执行者，变成了规则制定者，这种感觉很是不错，可是肩上也多了一份沉甸甸的责任，因为要对 SC 语言负责，这个语言有没有生命力，语言功能定位非常重要。

3.1 SC 语言的蓝本选择

SC 语言以 C 语言为蓝本进行简化，但 C 语言也有多个不同的版本，那么 SC 语言以 C 语言哪个版本为蓝本呢。下面简单介绍一下 C 语言自诞生到现在，出现的几个主要版本。

3.1.1 K&R C

早期的 C 语言还没有标准化，源自 Kernighan & Ritchie 合著的 *The C Programme Language* 的 C 描述可算作“正式”的标准，所以此时的 C 也称为 K&R C。之后 C 语言一直不断地发生细微的变化，各编译器厂商也有自己的扩展，这个过程一直持续到 20 世纪 80 年代末。

3.1.2 C89

考虑到标准化的重要，ANSI(American National Standards Institute)制定了第一个 C 标准，在 1989 年被正式采用(American National Standard X3.159—1989)，故称为 C89，也称为 ANSI C。该标准随后被 ISO 采纳，成为国际标准(ISO/IEC 9899:1990)。

C89 的主要改动：

- 定义了 C 标准库；
- 新的预处理命令和特性；
- 函数原型(prototype)；
- 新关键字 const、volatile、signed；
- 宽字符、宽字符串和多字节字符；
- 转化规则、声明(declaration)、类型检查的改变。

3.1.3 C99

这是目前最新的标准,由 ISO 于 1999 年制定(ISO/IEC 9899:1999),故称为 C99。

C99 的主要改动反映在以下几个方面:

- 复数(complex);
- 整数(integer)类型扩展;
- 变长数组;
- Boolean 类型;
- 非英语字符集的更好支持;
- 浮点类型的更好支持;
- 提供全部类型的数学函数;
- C++ 风格注释(//)。

C99 是当前的标准,但它仍未得到广泛支持,虽然标准发布已经多年。C99 对 C89 的改动非常大,如果编写 C99 的代码,那么可移植性必然受到限制。此外,个人认为 C99 的一些新特性在大多数程序设计中并不是必须的。C89 目前仍然使用最广泛,并得到所有主流编译器的支持。K&R C 现在只会在一些非常旧的代码中才能见到了,除非要维护旧代码,否则不应该再使用它。

所以,我们最终决定 SC 语言以 C89 为蓝本进行简化。

3.2 SC 语言对 C89 简化原则

下面首先说一下 SC 语言对 C 语言进行简化的原则:

- (1) 支持单字节、双字节、四字节的基本数据类型;
- (2) 支持数组、结构体;
- (3) 支持字符串;
- (4) 支持函数、局部变量、全局变量;
- (5) 支持条件语句、循环语句;
- (6) 支持基本的算术运算、关系运算;
- (7) 能用多种方式实现的功能,只保留一种,例如 C 语言的循环语句有 for 循环、do 循环、while 循环,只保留一种;
- (8) 原来 C 语言中绝大多数人用不到的一些功能去掉,例如 auto 关键字恐怕没有人用到过。

3.3 SC 语言的字符集

字符是组成语言的最基本的元素。SC 语言字符集由字母、数字、空格、标点和特殊字符组成。在字符常量、字符串常量和注释中还可以使用汉字或其他可表示的图形符号。

SC 语言包括两个字符集,一个是用于书写 SC 语言源文件的字符集,称为源码字符集;另一个是 SC 语言编译后在执行环境中解释的字符集,称为执行字符集。除在字符常量及

字符串常量中出现的转义字符外,两个字符集完全相同。

两个字符集中成员又分为两类,一类为基本字符集,这些字符可以出现在代码的任何位置;另一类为扩展字符集,扩展字符集包含基本字符集,扩展字符集中除基本字符集外的其他字符只能出现在源码的注释和字符串中。在字符常量或字符串常量中,执行字符集中的成员应当用对应的源字符集成员或者用由一个反斜线字符\后紧跟一个字符所组成的转义序列来表示。在基本执行字符集中应有一个其字节的所有位均置为0的,称为空字符的字符,空字符用来终止一个字符串。

3.3.1 基本字符集

基本源字符集和基本执行字符集均应至少具有下列成员。

1. 字母与下划线

(1) 字母。

- 英文字母表中的 26 个小写字母。

a	b	c	d	e	f	g	h	i	j	k	l	m
n	o	p	q	r	s	t	u	v	w	x	y	z

- 英文字母表中的 26 个大写字母。

A	B	C	D	E	F	G	H	I	J	K	L	M
N	O	P	Q	R	S	T	U	V	W	X	Y	Z

(2) 下划线。

2. 10 个数字

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

3. 标点和特殊字符

+	-	*	/	%	=	!	<	>	.	&
()	[]	{	}	;	,	\	"	'

4. 空白符

空格符、制表符、换行符等统称为空白符。空白符只在字符常量和字符串常量中起作用。在其他地方出现时,只起间隔作用,编译程序对它们忽略不计。因此在程序中使用空白符与否,对程序的编译不发生影响,但在程序中适当的地方使用空白符将增加程序的清晰性和可读性。

5. 空字符

字节的所有位均置为0的,称为空字符,空字符用来终止一个字符串。

3.3.2 扩展字符集

源字符集中可包含多字节字符,用于表示扩展字符集的成员。执行字符集中也可包含扩展字符集。下列条件对两个字符集均应成立:

- 应包含基本字符集所定义的单字节字符;
- 任何附加成员的存在、含义和表示均是地域特定的,是跟编译器的实现相关的,编译器有权决定支持哪些附加成员。

3.4 SC语言词法定义

SC语言的单词可以分为以下几类：关键字、标识符、整数常量、字符常量、字符串常量、运算符及分隔符、注释。

3.4.1 关键字

1. 定义

```
<char 关键字> ::= "char"
<short 关键字> ::= "short"
<int 关键字> ::= "int"
<void 关键字> ::= "void"
<struct 关键字> ::= "struct"
<if 关键字> ::= "if"
<else 关键字> ::= "else"
<for 关键字> ::= "for"
<continue 关键字> ::= "continue"
<break 关键字> ::= "break"
<return 关键字> ::= "return"
<sizeof 关键字> ::= "sizeof"
<__cdecl 关键字> ::= "__cdecl"
<__stdcall 关键字> ::= "__stdcall"
<__align 关键字> ::= "__align"
```

2. 语义

关键字是由语言规定的具有特定意义的字符串，通常也称为保留字，用户定义的标识符不应与关键字相同。SC语言这些关键字的功能如表3.1所示。

表3.1 SC语言关键字表

分 类	名 称	含 义
数据类型关键字	char	声明字符型变量或函数
	short	声明短整型变量或函数
	int	声明整型变量或函数
	void	声明函数无返回值，声明无类型指针
	struct	声明结构体变量
控制语句关键字	if	条件语句
	else	条件语句否定分支(与 if 连用)
	for	循环语句
	continue	结束当前循环，开始下一轮循环
	break	跳出当前循环
	return	子程序返回语句(可以带参数，也可不带参数)

续表

分 类	名 称	含 义
类型长度计算关键字	sizeof	计算类型长度
函数调用约定关键字	__cdecl	__cdecl 调用约定
	__stacall	__stacall 调用约定
结构成员对齐关键字	__align	__align(n) 强制结构成员对齐到 n

3.4.2 标识符

1. 定义

```

<标识符> ::= <非数字> {<数字>} | <非数字>
<非数字> ::= "_" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n"
          | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
          | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M"
          | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
<数字> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

```

2. 解释

标识符只能是字母(A~Z,a~z)、数字(0~9)、下划线(_)组成的字符串，并且其第一个字符必须是字母或下划线。

3. 语义

标识符可以代表很多实体：函数、变量、结构体名称、结构体成员、数组名。变量是一个存储区域，它的解释依赖于两个主要属性：它的存储类型和它的数据类型。存储类型决定了与该标识符相关联的存储区域的生命周期，数据类型决定了该对象值的含义。一个名字还有一个作用域，作用域即程序中可见此名字的区域。

4. 举例

以下标识符是合法的：

a、x、x3、BOOK_1、sum5。

以下标识符是非法的：

3s 以数字开头。

s * T 出现非法字符 *。

bowy-1 出现非法字符 - (减号)。

5. 使用标识符需要注意的问题

(1) 在标识符中，大小写是有区别的，例如 BOOK 和 book 是两个不同的标识符。

(2) 标识符虽然可由程序员随意定义，但标识符是用于标识某个量的符号。因此，命名应尽量有相应的意义，以便于阅读理解，做到“顾名思义”。

(3) SC 语言的关键字不能用作标识符名称。

3.4.3 整数常量

1. 定义

```
<整数常量> ::= <数字> {<数字>}
<数字> ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

2. 解释

整数常量由十进制数字(0~9)序列组成。

3. 语义

整数常量的值以10为基计算,词法上的第一个数字是最高有效位。

3.4.4 字符常量

1. 定义

```
<字符常量> ::= '<C-字符>'
<C-字符> ::= <转义序列> | 源字符集中除单引号'、反斜线字符\或新行字符外的任何字符
```

<转义序列>为下列序列之一:

```
\' \\" \\ \a \b \f \n \r \t \v
```

2. 解释

字符常量是由单引号括起来的一个或多个多字节字符组成的序列,例如'x'或'ab'。注意,'a'和'A'是不同的字符常量。除后面将详细描述的几种例外情况外,该序列中的元素可是源字符集中的任何成员,它们以一种实现定义的方式映射到**执行字符集**上。

除了以上形式的字符常量外,SC语言还允许用一种特殊形式的字符常量,就是以一个字符\开头的字符序列。例如,前面已经遇到过的,在printf函数中的\n,它代表一个“换行”符。这是一种“控制字符”,在屏幕上是不能显示的,在程序中也无法用一个一般形式的字符表示,只能采用特殊形式来表示。常用以\开头的特殊字符见表3.2。

表3.2 SC语言转义字符表

源码字符形式	含 义	执行字符 ASCII 码
\0	空字符	0
\a	响铃	7
\b	退格,将当前位置移到前一列	8
\t	水平制表(跳到下一个Tab位置)	9
\n	换行,将当前位置移到下一行开头	10
\v	纵向制表转义字符	11
\f	换页,将当前位置移到下页开头	12
\r	回车,将当前位置移到本行开头	13
\"	代表一个双引号字符	34
\'	代表一个单引号(撇号)字符	39
\\\	代表一个反斜杠字符\	92

表 3.2 中列出的字符称为“转义字符”，意思是将反斜杠\后面的字符转换成另外的意义。如\n 中的 n 不代表字母 n 而作为“换行”符。

3. 语义

整型字符常量的类型是 int，如字符常量包含单个字符或转义序列，则它的值是将该单字符或转义序列值的 char 类型的对象转换为 int 类型时所得到的值。包含多于一个字符，或包含一个不在基本执行字符集中表示的字符或转义序列的整型字符常量的值是实现定义的。

3.4.5 字符串常量

1. 定义

```
<字符串常量> ::= "<串字符>"
<串字符> ::= <转义字符> | 源字符集中除双引号字符"、反斜线字符\或新行字符外的任何字符
```

2. 解释

字符串常量是由双引号括起来的零个或多个多字节字符组成的序列，例如“盗版严重阻碍了中国的经济发展和自主创新”。

3. 语义

字符串常量实际上是字节数组，该数组的元素类型为 char，用该多字节字符序列中的各个字节初始化数组的各个元素。该数组长度为表示该字符串的多字节字符序列的长度加 1，多出的这最后一个字节的值为 0，标识字符串的结束。

3.4.6 运算符及分隔符

1. 定义

```
<加号> ::= "+"
<减号> ::= "-"
<星号> ::= "*"
<除号> ::= "/"
<取余号> ::= "%"
<等于号> ::= "==""
<不等于号> ::= "!="
<小于号> ::= "<"
<小于等于号> ::= "<="
<大于号> ::= ">"
<大于等于号> ::= ">="
<赋值等号> ::= "="
<箭头> ::= "->"
<点号> ::= "."
<与号> ::= "&"
<左小括号> ::= "("
<右小括号> ::= ")"
<左中括号> ::= "["
```

```
<右中括号> ::= "]"
<左大括号> ::= "{"
<右大括号> ::= "}"
<分号> ::= ";"
<逗号> ::= ","
<省略号> ::= "..."
```

2. 约束

算符[]、()和{}应成对出现。

3. 语义

算符说明要执行的运算,该运算产生一个值,或一个副作用(有关副作用解释,见3.5.3节),或它们的组合。操作数是算符所作用的实体。

分隔符是具有独立的语法和语义含义的符号,但并不指定一个执行后获得值的运算。依赖于上下文,同一符号也可表示一个算符或某个算符的一部分。

3.4.7 注释

除出现在字符常量、字符串常量或注释中外,字符对/*引入一注释。对注释内容的检查仅为了辨识多字节字符以及发现终止注释的字符对*/ ,这里要注意注释不允许嵌套。

3.5 SC语言语法定义

写了这么多,连语言还没定义完,想必大家已经迫不及待了,但是摩天大楼可不是一天建成的。从可行性研究到初步设计,再到施工图设计,最后进入施工,前面可研及设计这些准备工作可能比施工周期还要长。因为如果前期准备不足,任何一个环节出问题,最后只能建成一栋烂尾楼;所以请大家还是要耐着点性子,打起精神,来看SC语言的语法定义。

3.5.1 外部定义

1. 定义

```
<翻译单元> ::= {<外部声明>} <文件结束符>
<外部声明> ::= <函数定义> | <声明>
```

2. 语义

每个SC源文件是一个翻译单元,它由一系列外部声明所组成。将这些描述为“外部的”是因为它们出现在任何函数之外,因而具有全局作用域。

3.5.1.1 函数定义

1. 定义

```
<函数定义> ::= <类型区分符> <声明符> <函数体>
<函数体> ::= <复合语句>
```

2. 语义

函数定义中的声明符指定要定义的函数名及形式参数列表。

3. 约束

在函数定义中声明的标识符(即函数名)应为函数类型,该类型由函数定义的声明符部分所说明。

3.5.1.2 声明

1. 定义

```
<声明> ::= <类型区分符> [<初值声明符表>] <分号>
<初值声明符表> ::= <初值声明符> {<逗号> <初值声明符>}*
<初值声明符> ::= <声明符> | <声明符> <赋值运算符> <初值符>
```

2. 语义

声明规定一组标识符的解释和属性,而同时还导致为标识符所代表的对象或函数保留存储空间的声明则是定义。

类型区分符指示声明符所代表实体的类型部分。初值声明符表是由逗号分隔的声明符序列,其中每个声明符均可有附加的类型信息或初值符,或二者兼有。

3. 约束

在同一作用域,对该标识符的声明不应超过一个,涉及同一对象或函数的所有声明均应为相容的类型。

3.5.1.3 类型区分符

1. 定义

```
<类型区分符> ::= <void 关键字> | <char 关键字> | <short 关键字> |
    <int 关键字> | <结构区分符>
<结构区分符> ::= <struct 关键字> <标识符> <左大括号> <结构声明表> <右大括号> |
    <struct 关键字> <标识符>
<结构声明表> ::= <结构声明> {<结构声明>}
<结构声明> ::= <类型区分符> {<结构声明符表>} <分号>
<结构声明符表> ::= <声明符> {<逗号> <声明符>}
```

2. 语义

结构是一种由一系列成员变量所组成的类型,各成员的存储区按顺序分配。结构区分符中出现的结构声明表声明了在一个翻译单元中的一种新类型。结构声明表是一系列对结构成员的声明,直至终止结构声明表的}之前,该新类型是不完整的。结构的成员可以是任何对象类型。

3. 约束

结构中不应包含函数类型的成员,也不应包含其自身的实例,但可包含指向其自身实例的指针。

例如:

```

struct node
{
    int value;
    struct node cur;           //不合法
    struct nod * next;         //合法
}

```

3.5.1.4 声明符

1. 定义

<声明符> ::= {<指针>} [<调用约定>] [<结构成员对齐>] <直接声明符>
 <调用约定> ::= <__cdecl 关键字> | <__stdcall 关键字>
 <结构成员对齐> ::= <__align 关键字> <左小括号> <整数常量> <右小括号>
 <指针> ::= <星号>
 <直接声明符> ::= <标识符> <直接声明符后缀>
 <直接声明符后缀> ::= {<左中括号> <右中括号>
 | <左中括号> <整数常量> <右中括号>
 | <左小括号> <右小括号>
 | <左小括号> <形参表> <右小括号> }
 <参数声明> ::= <类型区分符> <声明符>

2. 语义

每个声明符声明一个标识符，并表明当某个表达式中出现与该声明符形式相同的操作数时，则该操作数指示一个函数或对象，它们具有由声明区分符所指出的作用域、存储期及类型。

3. 约束

调用约定只对函数起作用，结构体成员对齐只对结构体成员变量声明起作用。

3.5.1.5 初值符

1. 定义

<初值符> ::= <赋值表达式>

2. 语义

初值符规定存储在对象中的初始值，若未显式初始化一个自动存储期的对象，则其值不确定。

3.5.2 语句

1. 定义

<语句> ::= {<复合语句> |
 <if 语句> |
 <for 语句> |
 <break 语句> |
 <continue 语句> |
 <return 语句> |

<表达式语句>}

2. 语义

语句规定所要执行的动作,除非特别规定外,语句均按顺序执行。

3.5.2.1 复合语句

1. 定义

<复合语句> ::= <左大括号> {<声明>} {<语句>} <右大括号>

2. 语义

复合语句(也称为块)允许将一系列语句组合成一个语法单位,该语法单位可以有其自己的一组声明。

3.5.2.2 表达式语句与空语句

1. 定义

<表达式语句> ::= [<expression>] <分号>

2. 语义

表达式语句中的表达式将按 void 表达式求值以获得其副作用(例如赋值以及具有副作用的函数调用等)。

空语句仅由一个分号组成,不执行任何操作。

3.5.2.3 选择语句

1. 定义

<if 语句> ::= <if 关键字> <左小括号> <表达式> <右小括号> <语句> [<else 关键字> <语句>]

2. 语义

依据控制表达式的值,选择语句在一系列语句中作选择。对两种形式的 if 语句,若表达式与零比较结果不相等,则都执行第一个子语句。对 else 形式的语句,若表达式与零比较结果相等,则执行第二个子语句。else 与在同一块中的、词面上仅先于它的无 else 的 if 相关联,即就近关联原则。

3. if 语句在实际应用中的 3 种形式:

(1)

`if(表达式) 语句`

(2)

`if(表达式) 语句 1 else 语句 2`

(3)

`if(表达式 1) 语句 1`

`else if(表达式 2) 语句 2`

`else if(表达式 m) 语句 m`

`else 语句 n`

3种形式的程序流程图如图3.1所示。

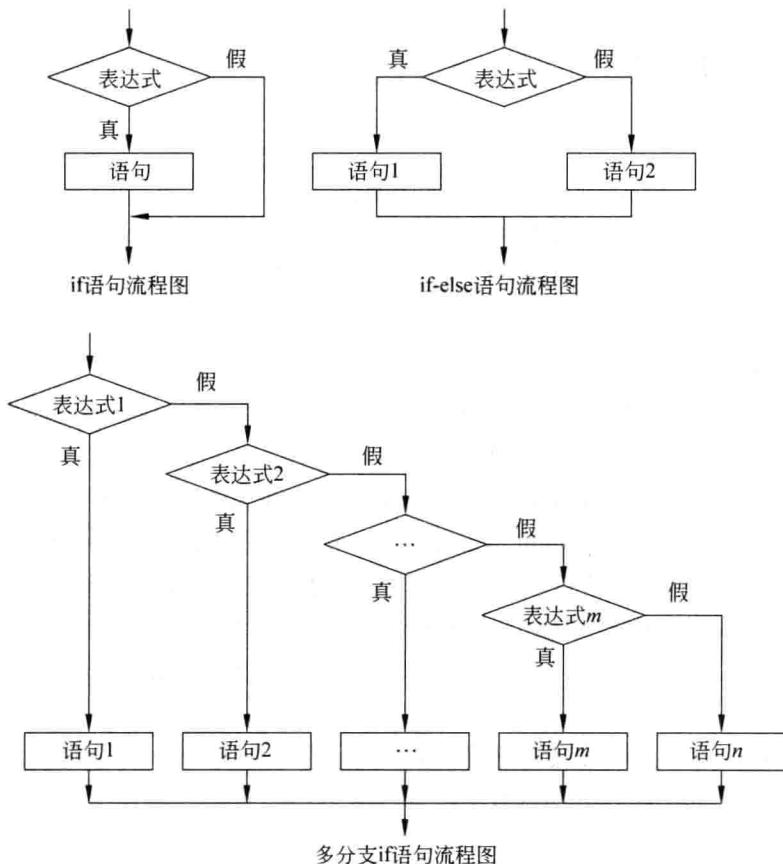


图3.1 3种形式的if语句流程图

3.5.2.4 循环语句

1. 定义

```
<for语句> ::= <for关键字><左小括号><表达式语句><表达式语句><表达式>
<右小括号><语句>
```

2. 语义

循环语句使得称为循环体的语句重复执行直至控制表达式与零比较结果相等。

3. for语句在实际应用中的形式

`for(表达式 1; 表达式 2; 表达式 3) 语句`

其程序流程图如图3.2所示。

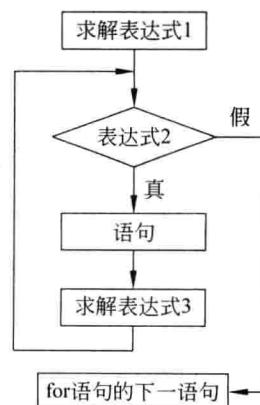


图3.2 for语句程序流程图

表达式 1 指定了对循环的初始化。表达式 2, 即控制表达式, 规定在每次循环前求值一次, 以便循环的执行得以继续, 直至该表达式与零比较结果相等。表达式 3 规定在每次循环后执行的一个操作(例如增量)。表达式 1 和表达式 3 都可省略, 每个都按 void 表达式求值, 省略的表达式 2 可用非零常量替换。

3.5.2.5 跳转语句

1. 定义

```
<continue 语句> ::= <continue 关键字><分号>
<break 语句> ::= <break 关键字><分号>
<return 语句> ::= <return 关键字><expression><分号>
```

2. 语义

跳转语句导致控制无条件地转向另一处。

continue 语句导致控制转向包含它的最小循环语句的继续循环部分, 即转向循环体的末尾。break 语句终止包含它的最小循环语句的执行。

return 语句终止当前函数的执行并将控制返回其调用者。一个函数可有任何数量的 return 语句, return 语句可以带也可不带表达式。执行带表达式的 return 语句, 该表达式的值将作为函数调用表达式的值返回给调用者。若该表达式的类型与它所在的函数的类型不同, 则将对它进行转换, 如同将它赋于一个该类型的对象一样。

若执行不带表达式的 return 语句, 且调用者要使用该函数调用的值, 则行为是未定义的。

3. 约束

continue 语句、break 语句应仅在循环体内或作为循环体出现, 带表达式的 return 语句不应出现在其返回类型为 void 的函数中。

4. 举例

continue 语句在实际应用中的形式如下:

```
for(表达式 1; 表达式 2; 表达式 3
{
    :
    if(表达式 4) continue;
    :
}
```

其程序流程图如图 3.3 所示。

break 语句在实际应用中的形式如下:

```
for(表达式 1; 表达式 2; 表达式 3
{
    :
    if(表达式 4) break;
    :
}
```

其程序流程图如图 3.4 所示。

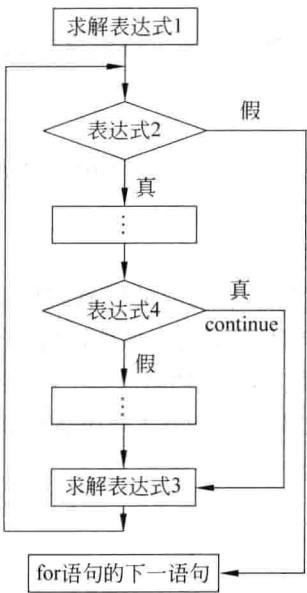


图 3.3 含 continue 语句的 for 循环流程图

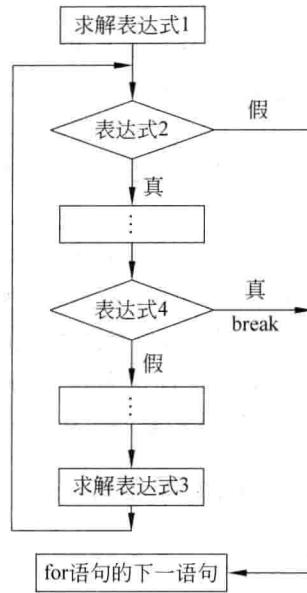


图 3.4 含 break 语句的 for 循环流程图

3.5.3 表达式

1. 定义

<表达式> ::= <赋值表达式> {<逗号><赋值表达式>}

2. 语义

表达式是由算符和操作数组成的序列,它规定了对一个值的计算,或产生一种副作用,或它们的组合。在两个相继的顺序点之间,最多可通过对表达式求值将一个对象的存储值修改一次。这里有两个概念要重点解释一下:表达式的副作用和顺序点。

1) 表达式的副作用

一般说计算一个表达式的值需要引用一些变量,在表达式求值过程中,需要提取这些变量的值,但并不改变这些变量的值,这样的表达式称为无副作用的表达式。从传统意义上讲,表达式的作用就是计算,它除了产生一个计算结果外,不应该改变参与计算过程的任何变量的值或产生其他效应。换句话说,传统意义上的表达式是不应该有副作用的。但是,如果一个表达式在求值过程中,对使用的变量不但引用,对它们的值还加以改变,这样的表达式称为有副作用的表达式,或者称这个表达式是有副作用的。

例如:

```
20 //这个表达式的值是 20;它没有副作用,因为它没有改变任何变量的值
x=5 //这个表达式的值是 5;它有一个副作用,因为它改变了变量 x 的值
```

2) 顺序点

顺序点的意思是在一系列步骤中的一个“结算”的点,语言要求这一时刻的求值和副作

用全部完成,才能进入下面的部分。在 SC 语言有以下几种顺序点:

- 每个表达式的结束处。
- 函数调用中,在所有参数求值完成后,函数体开始执行前有一个顺序点,而参数间的逗号处则没有顺序点。
- 逗号表达式中每一个逗号处有一个顺序点。
- 在每一个完整的变量声明处有一个顺序点,例如 int i, j; 中逗号和分号处分别有一个顺序点。
- for 循环控制条件中的两个分号处各有一个顺序点。

对于任意一个顺序点,它之前的所有副作用都已经完成,它之后的所有副作用都尚未发生。在两个顺序点之间,子表达式求值和副作用的顺序是不同步的。如果代码的结果与求值和副作用发生顺序相关,称这样的代码有不确定的行为。而且,假如期间对一个内建类型执行一次以上的写操作,则是未定义行为。任意两个顺序点之间的副作用的发生顺序都是未定义的。

3.5.3.1 赋值表达式

1. 定义

```
<赋值表达式> ::= <相等类表达式>
| <一元表达式><赋值等号><赋值表达式>
```

2. 语义

赋值算符在其左操作数所指示的对象中存储一个值。对于简单赋值(=),右操作数的值转换为赋值表达式的值,并用它替换存储在由左操作数所指示的对象中的值。应在前一个和下一个顺序点之间发生副作用,以更新其左操作数的存储值。

3. 约束

赋值算符的左操作数应为可修改的左值。

这里要注左值的概念,左值(lvalue)是可以赋值的,说明它是一个变量,它在内存中一定存在,一定有地址。所以 &lvalue 是有效的,能取到其在内存中的地址。

3.5.3.2 相等类表达式

1. 定义

```
<相等类表达式> ::= <关系表达式> {
    <等于号><关系表达式>
    | <不等于号><关系表达式> }
```

2. 语义

$=$ =(等于)和 $!=$ (不等于)算符除优先级较低外,其余均与关系算符类似。

注:由于优先级的关系,只要 $a < b$ 和 $c < d$ 的真值相同,则 $a < b == c < d$ 为 1。

若两个指向对象类型的指针均为空指针,则它们比较时相等。若两个指向对象的指针比较时相等,则它们或者都是空指针,或者都指向同一对象,或者都指向刚超越同一数组对象最后一个元素的位置。若两个指向函数类型的指针均为空指针或都指向同一函数,则它们比较时相等。若两个指向函数类型的指针比较时相等,则它们或者都是空指针,或者都指

向同一函数。若其中一个运算数是指向对象的指针,而另一个操作数是 void 类型指针,则前者将转换为后者的类型。

3. 约束

下列条件之一应成立:

- 两个操作数均为算术类型;
- 两个操作数均为指向相容对象类型的指针;
- 一个操作数是指向某对象指针,另一个操作数是 void 类型指针;
- 一个操作数是指针,另一个操作数是空指针常量。

3.5.3.3 关系表达式

1. 定义

```
<关系表达式> ::= <加减类表达式> {
    <小于号><加减类表达式>
    |<大于号><加减类表达式>
    |<小于等于号><加减类表达式>
    |<大于等于号><加减类表达式>
}
```

2. 语义

若两个操作数均为算术类型,则执行一般算术转换。

对于这类算符,指向非数组对象的指针的行为如同指向长度为 1、以该对象类型为其元素类型的数组中第一个元素的指针一样。

将两个指针作比较时,结果取决于它们所指向的对象在地址空间的相对位置。若所指向的对象是同一结构对象的成员,则在对指针作比较时,后声明的结构成员的指针高于在该结构中先声明的成员的指针;下标值较大的数组元素的指针高于同一数组中下标值较小的元素的指针。

若两个对象的指针均指向同一对象,或均指向刚超越同一数组对象最后一个元素的位置,则它们比较时相等;反过来,若两个对象的指针比较时相等,则它们都指向同一对象,或都指向刚超越同一数组对象最后一个元素的位置。

算符<(小于)>(大于)<=(小于等于)和>=(大于等于)中的每一个所规定的关系为真时都应得 1,否则都应得 0。结果的类型是 int。

注:表达式 $a < b < c$ 不按一般数学意义解释。正如其语法所规定的,它意味着 $(a < b) < c$; 换言之,即“若 $a < b$ 用 1 与 c 比较,否则用 0 与 c 比较”。

3. 约束

下列条件之一应成立:

- 两个操作数均为算术类型;
- 两个操作数均为指向相容对象类型的指针。

3.5.3.4 加减类表达式

1. 定义

```
<加减类表达式> ::= <乘除类表达式> {
```

<加号><乘除类表达式>
|
<减号><乘除类表达式>}

2. 语义

若两个操作数均为算术类型，则对它们执行一般算术转换。

二元+算符的结果是其操作数之和。

二元-算符的结果是从第一操作数中减去第二操作数所得的差。

对这些算符，指向非数组对象的指针的行为与指向长度为 1、以该对象类型为其元素类型的数组中第一个元素的指针相同。

当指针加上或减去一个整型表达式时，结果的类型取指针操作数的类型。若该指针操作数指向某个数组对象的元素，且该数组足够大，则结果指向一个与原始元素有一定偏移的元素，该元素与原始的元素的下标之差等于该整型表达式。换言之，若表达式 p 指向某数组对象的第 i 个元素，则表达式 $(p)+n$ 和 $(p)-n$ 分别指向该数组对象的第 $i+n$ 和 $i-n$ 个元素，只要这样的元素存在。而且若表达式 p 指向某数组对象的最后一个元素，则表达式 $(p)+1$ 指向刚超越该数组对象最后一个元素的位置；若表达式 Q 指向刚超越某数组对象最后一个元素的位置，则表达式 $(Q)-1$ 指向该数组对象的最后一个元素。

当两个指向同一数组对象的指针相减时，结果是两个数组元素下标之差。若表达式 P 和 Q 分别指向某数组对象的第 i 和第 j 个元素，则表达式 $(P)-(Q)$ 的值就等于 $i-j$ 。

另一种实现指针运算的方法是首先将指针转换为字符指针：以这种模式，则要与转换后的指针相加或从转换后的指针中减去的整型表达式首先乘以原先所指向的对象的尺寸，结果的指针再转换回原先的类型。对于指针相减，作为结果的两个字符指针的差也将类似地除以原先所指向的对象的尺寸。

3. 约束

对于加，应是两个操作数均为算术类型，或者一个操作数是指向某对象类型的指针，而另一个操作数是整型。

对于减，下列之一应成立：

- 两个操作数均为算术类型；
- 两个操作数均为指向相容对象类型的指针；
- 左操作数是指向一个对象类型的指针，右操作数是整型。

3.5.3.5 乘除类表达式

1. 定义

<乘除类表达式> ::= <一元表达式> {
 <星号><一元表达式>
 | <除号><一元表达式>
 | <取余运算符><一元表达式>}

2. 语义

二元 * 算符的结果是其操作数的积。

/ 算符的结果是其第一操作数除以第二操作数所得的商；而 % 算符的结果是余数。在上

述两种运算中,若第二操作数的值为零,则行为是未定义的。

当整数相除且不能整除时,若两个操作数均为正,则/算符的结果是小于其代数商的最大整数,且%算符的结果为正。若两个操作数均为负,则/算符的结果是小于等于其代数商的最大整数还是大于等于其代数商的最小整数由实现定义,%算符结果的符号也由实现定义。

3. 约束

每个操作数均应为算术类型,%算符的操作数应为整型。

3.5.3.6 一元表达式

1. 定义

```
<一元表达式> ::= <后缀表达式>
    | <与号><一元表达式>
    | <星号><一元表达式>
    | <加号><一元表达式>
    | <减号><一元表达式>
    | <sizeof 表达式>
<sizeof 表达式> ::= <sizeof 关键字> (<类型区分符>)
```

2. 地址与间接算符

1) 语义

一元&(地址)算符的结果是其操作数所指示的对象或函数的指针。若其操作数的类型是某 type,则结果的类型是“指向该 type 的指针”。

一元*算符表示间接取值。若其操作数指向一个函数,则结果是一个函数指示符;若其操作数指向一个对象,则结果是一个指示该对象的左值;若其操作数的类型是“指向某 type 的指针”,则结果的类型就是该 type;若已将一个无效值赋予了该指针,则一元*算符的行为是未定义的。

2) 约束

一元&算符的操作数应是一个函数指示符或者表示一个对象的左值。

一元*算符的操作数应是指针类型。

3. 一元算术算符

1) 语义

一元+算符的结果是其操作数的值。

一元-算符的结果是其操作数取负,即 0 减该操作数所得值。

2) 约束

一元+算符或一元-算符的操作数应为算术类型。

4. sizeof 算符

1) 语义

sizeof 算符得出其操作数的(按字节计的)尺寸,其操作数必须是类型名。结果是一个整数常量。

当 sizeof 算符应用于 char 类型的操作数时,结果是 1;当应用于结构类型的操作数时,结果是这类对象的总字节数,包括内部和结尾的填充字节在内。

2) 约束

不应将 sizeof 算符应用于函数类型的表达式。

3.5.3.7 后缀表达式

1. 定义

```
<后缀表达式> ::= <初等表达式> {
    <左中括号><expression><右中括号>
    |<左小括号><右小括号>
    |<左小括号><实参表达式表><右小括号>
    |<点号>IDENTIFIER
    |<箭头>IDENTIFIER
<实参表达式表> ::= <赋值表达式> {<逗号><赋值表达式>} }
```

2. 下标数组

1) 语义

在其后跟有一个在方括号内的表达式的后缀表达式是数组对象元素加下标的指示。下标算符[]的定义是：E1[E2]与(* (E1 + (E2)))完全相同。由于应用于二元+算符的转换规则，若 E1 是一个数组对象（或等价地，是指向数组对象第一个元素的指针），E2 是一个整数，则 E1[E2]指示 E1 的第 E2 个元素（从 0 开始计数）。连续的下标算符指示多维数组对象的一个元素。若 E 是一个 n 维数组 ($n \geq 2$)，其维数是 $i \times j \times \dots \times k$ ，则 E（不用作左值）将转换为指向各维是 $j \times \dots \times k$ 的 $(n-1)$ 维数组的指针。若对此指针显式地或作为下标应用的结果隐式地应用一元 * 算符，则结果是所指向的 $(n-1)$ 维数组，且若结果不用作左值，则该结果也转换为一个指针。由此可见，数组是按行优先的顺序存储的，即最后一个下标变化最快。

示例

考虑由声明：

```
int x[3][5];
```

所定义的数组对象，其中 x 是一个 3×5 的 int 数组；更确切地说，x 是有 3 个元素对象的数组，每个元素对象又是 5 个 int 的数组。在表达式 x[i]（等价于 (* (x + (i)))) 中，x 首先转换为指向第一个 5 个 int 的数组，然后按照 x 的类型调整 i，这在概念上是将 i 乘以该指针所指向的对象的尺寸，在此即是 5 个 int 对象的数组的尺寸。将乘得的结果与 x 相加，得到一个 5 个 int 的数组。当用在表达式 x[i][j] 中时，上述结果又转换为指向一个 int 的指针，于是 x[i][j] 得到一个 int。

2) 约束

其中一个表达式的类型应是“指向对象 type 的指针”，而另一个表达式的类型应为整型，结果的类型取 type。

3. 函数调用

1) 语义

其后跟有括号()，括号中包含可能为空的或以逗号分隔的表达式表的后缀表达式是一

个函数调用。该后缀表达式表示被调函数,而括号内的表达式表规定了函数的实参。

若在函数调用中加括号的表达式表之前的表达式仅由一个标识符组成,且若对该标识符无可见的声明,则该标识符是隐式声明的,效果如同在包含该函数调用的最内部的块中出现声明“int 标识符();”时一样,即该标识符标识一个块作用域的,且被声明为具有外部链接,类型为无形参信息、并返回 int 的函数。如果事实上该函数并非定义为“返回 int 的函数”类型,则行为是未定义的。

实参可以是任何对象类型的表达式。在准备调用函数之前,先对实参数求值,并将实参数赋给每个相对应的形参。函数可以改变其形参的值,但这些改变不应影响实参数。另外,可以传递指向一个对象的指针,而函数可以改变该指针所指向的对象的值。声明为数组类型或函数类型的形参将转换为指针类型的形参。

若实参数目与形参数目不一致,则行为是未定义的。若实参数类型与形参类型不相容,则行为也是未定义的。

对函数指示符、实参及实参中子表达式的求值顺序未作规定,但在实际调用前有一个顺序点,应允许函数递归调用,直接递归或通过任何其他函数调用链的间接递归均允许。

2) 约束

标记被调函数的表达式的类型应是指向一个函数的指针,该函数返回 void 类型,或者返回除数组及结构体类型外的对象类型。实参的数目应与形参的数目一致。实参的类型应与实参所对应的形参类型一致。

4. 结构成员

1) 语义

跟有一个圆点. 和一个标识符的后缀表达式指示结构对象的一个成员,其值是所命名成员的值,且第一个表达式是左值,该值也是一个左值。

跟有一个向右箭头->和一个标识符的后缀表达式也指示结构对象的一个成员,其值是由第一个表达式所指向的对象成员的值,且是一个左值。

2) 约束

- . 算符的第一操作数应为结构类型,第二操作数应为该结构类型的一个成员。

- >算符的第一操作数类型应为指向结构的指针,第二操作数应为该结构类型的一个成员。

3.5.3.8 初等表达式

1. 定义

<初等表达式> ::= <标识符> | <整数常量> | <字符串常量> | <字符常量> | (<表达式>)

2. 语义

只要标识符已被声明为指示一个对象(此时它是一个左值),或函数(此时它是一个函数指示符),则标识符就是一个初等表达式。

一个常量是初等表达式,其类型取决于它的形式和值。

一个字符串是初等表达式,它是一个 char 类型数组。

加括号的表达式也是初等表达式。其类型和值与未加括号的表达式完全相同。

3.6 SC 语言与 C 语言功能对比

3.6.1 关键字

关键字是由语言规定的具有特定意义的字符串,通常也称为保留字,用户定义的标识符不应与关键字相同。通过下面的榜单(见表 3.3)看一下 SC 关键字与 C 语言关键字对比情况。

表 3.3 SC 语言与 C 语言关键字对比

SC 语言支持的关键字		SC 语言不支持的关键字			
光荣榜		黑马榜	落第榜		
char	short	__cdecl	auto	double	long
int	void	__stdcall	switch	case	enum
struct	if	__align	register	typedef	extern
else	for		union	const	float
continue	break		unsigned	signed	default
return	sizeof		goto	volatile	do
			static	while	

榜单分为 3 部分。左侧的光荣榜为原来的 C 语言关键字,本次又被选上的,有 12 个。中间的黑马榜中的几个以前不是 C 语言关键字,是被破格录用的三匹黑马。也就是说 SC 语言总共有 $12 + 3 = 15$ 个关键字,比 C 语言少一半。右侧的落第榜中,是落选的 C 语言关键字,共有 20 个。你可能觉得 C 语言已经足够简洁了,怎么还能去掉这么多,别忘了 SC 语言的定位,是对 C 语言进行最简化的学习语言。

3.6.2 数据类型

1. 支持的数据类型

(1) 基本数据类型最主要的特点是,其值不可以再分解为其他类型。也就是说,基本数据类型是自我说明的。在 SC 语言中,基本数据类型有 int、char、short,上述 3 种基本类型均为有符号类型,其表示的数据范围如表 3.4 所示。

表 3.4 SC 语言基本数据类型

类型名称	类型符号	字节数	比特数	所表示的数值范围
整型	int	4	32	$-2^{147} 483 648 \sim 2^{147} 483 647$ 即 $-2^{31} \sim (2^{31} - 1)$
短整型	short	2	16	$-32 728 \sim 32 727$ 即 $-2^{15} \sim (2^{15} - 1)$
字符型	char	1	8	$-128 \sim 127$ 即 $-2^7 \sim (2^7 - 1)$

(2) 构造数据类型是根据已定义的一个或多个数据类型用构造的方法来定义的。也就是说,一个构造类型的值可以分解成若干个“成员”或“元素”。每个“成员”都是一个基本数据类型或又是一个构造类型。在 SC 语言中,构造类型有数组类型、结构体类型两种。

(3) 指针是一种特殊的,同时又是具有重要作用的数据类型。其值用来表示某个变量在内存存储器中的地址。虽然指针变量的取值类似整型量,但这是两个类型完全不同的量,因

此不能混为一谈。

(4) 在调用函数值时,通常应向调用者返回一个函数值。这个返回的函数值是具有一定的数据类型的,应在函数定义及函数说明中给以说明,例如函数 int max(int a,int b)返回

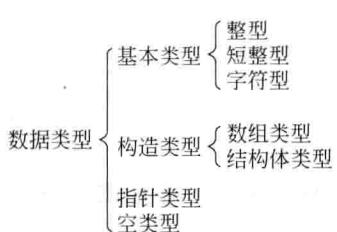


图 3.5 SC 语言支持的数据类型

值为整型量,函数 char upper(char c)返回值为字符型。但是,也有一类函数,调用后并不需要向调用者返回函数值,这种函数可以定义为“空类型”。其类型说明符为 void,例如,void print_string(char * str)返回值为 void 类型。

SC 语言支持的数据类型如图 3.5 所示。

2. SC 语言不支持的数据类型

(1) 基本类型中不支持单精度浮点型(float)、双精度浮点型(double)、枚举类型。

(2) 不支持 unsigned、signed、long 类型关键字。

(3) 构造类型中不支持共用体类型(union)。

3.6.3 存储类型

SC 语言变量只有全局变量和局部变量两种类型,全局变量都属于静态存储类型,局部变量均为自动变量。

C 语言所有的存储类型关键字 SC 语言都不支持:

(1) auto 声明自动变量,一般不使用。

(2) extern 声明变量是在其他文件中声明(也可以看做引用变量)。

(3) register 声明寄存器变量。

(4) static 声明静态变量。

3.6.4 常量

在程序运行过程中,其值不能被改变的量称为常量。SC 语言支持如下常量类型:

(1) 整型常量,如 12、0、-3。

(2) 字符常量,如'a'、'B'、'1'。

(3) 字符串常量,"Hello World\n","我爱你,中国!"。

SC 语言不支持浮点常量,如 1.5。SC 语言中不支持用标识符来代表常量,如 #define PI 3.1415926。

3.6.5 变量

其值可以改变的量称为变量,一个变量应该有一个名字,在内存中占据一定的存储单元。在该存储单元中存放变量的值。请注意区分变量名和变量值这两个不同的概念,见图 3.6。

变量名实际上是一个符号地址,在对程序编译连接时由系统给每一个变量名分配一个内存地址。在程序中从变量中取值,实际上是通过变量名找到相应的内存地址,从其

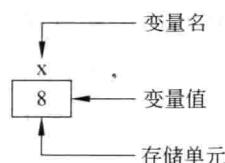


图 3.6 变量名、变量值、变量存储单元之间的关系

存储单元中读取数据。变量必须先定义后使用。

和 C 语言一样,用来标识变量名的标识符,只能由字母、数字、下划线 3 种字符组成,且第一个字母必须为字母或下划线。标识符还可用来表示函数名、数组名、函数形参和结构体成员,其命名规则与变量名完全一样。

变量赋初值,数组变量只能单个赋初值,这点要特别注意。

3.6.6 函数

SC 语言函数与 C 语言函数没什么区别。这里主要讲一下函数定义及调用的一般形式和形参与实参。

- 函数定义的一般形式为

```
类型标识符 调用约定 函数名(形参列表)
{
    声明部分
    语句
}
```

形参列表可以为空,即没有形参。调用约定可省略,默认为 __cdecl 调用约定。支持可变参数的形参列表,如 int sum(num,...);

- 函数调用的一般形式为:

函数名(实参列表)

如果调用无参函数,则“实参列表”可以没有,但括号不能省略。如果实参列表包含多个实参,则各参数间用逗号隔开。

- 形式参数和实际参数

在调用有参函数里,主调函数和被调用函数之间有数据传递关系。在定义函数函数名后面括号中的变量名称为“形式参数”(简称“形参”)。在主调函数中调用一个函数时,函数名后面括号中的参数称为“实际参数”(简称“实参”)。实际参数可以是常量、变量、表达式。

3.6.7 语句

SC 程序的执行部分是由语句组成的,程序的功能也是由执行语句实现的。语句是程序指令,除非特别说明,语句都按顺序执行。SC 语言的语句类型有复合语句、表达式语句、空语句、选择语句、循环语句和跳转语句。

3.6.7.1 复合语句

把多个语句用括号{}括起来组成的一个语句称复合语句。在程序中应把复合语句看成是单条语句,而不是多条语句,例如

```
{
    x=y+z;
    a=b+c;
```

```

    printf("%d%d", x, a);
}

```

是一条复合语句。复合语句内的各条语句都必须以分号;结尾;此外,在括号}外不能加分号。

3.6.7.2 表达式语句

表达式语句由表达式加上分号;组成。

其一般形式为: 表达式;执行表达式语句就是计算表达式的值。

例如:

```

x=y+z;a=520;      //赋值语句
y+z;                //加法运算语句,但计算结果不能保留,无实际意义

```

3.6.7.3 空语句

只有分号;组成的语句称为空语句,空语句是什么也不执行的语句,在程序中空语句可用来作空循环体。

例如,for(; getchar() != '\n');这两句代码功能是,只要从键盘输入的字符不是回车则重新输入。这里有两个空语句,for 循环的第一个表达式为空语句,循环体也为空语句。

3.6.7.4 选择语句

选择语句只支持 if-else 语句,不支持 switch-case 语句。在讲 SC 语言对 C 语言的简化原则时,说过原来 C 语言能用多种方式实现的功能,SC 语言只保留一种。C 语言的选择语句有 if-else 和 switch-case 两种,两种选择语句在应用范围上有所区别。if-else 可以表达所有的条件选择情况,通用性最好,用 switch-case 处理的都可以用 if-else 处理,所以 SC 语言选择语句只保留 if-else 语句。下面来看一个 switch-case 语句等价变换为 if-else 语句的例子。

```

/* 按照考试成绩的等级打印出百分制分数段 */
switch(grade)
{
    case 'A': printf("85-100\n"); break;
    case 'B': printf("70-84\n");   break;
    case 'C': printf("60-69\n");   break;
    case 'D': printf("< 60\n");    break;
    default:   printf("error!\n");
}

```

等价变换为

```

if(grade=='A')      printf("85-100\n");
else if(grade=='B') printf("70-84\n");
else if(grade=='C') printf("60-69\n");
else if(grade=='D') printf("< 60\n");
else               printf("error!\n");

```

3.6.7.5 循环语句

循环语句只支持 for 循环,不支持 while 循环和 do-while 循环。所有的 while 循环、do-while 循环都可以用 for 循环来表示。通过表 3.5,来看一下 while 循环、do-while 循环等价转换为 for 循环的例子。

表 3.5 循环的等价转换

循环语句	等价转换为 for 循环
while(表达式 A) 语句 B	for(;表达式 A;) 语句 B
do 语句 C while(表达式 D)	for(;;) { 语句 C if(表达式 D == 0) Break; }

3.6.7.6 跳转语句

跳转语句只支持 continue、break、return,不支持 goto 语句。

这里来看一下学术界对 goto 语句的看法。goto 语句的争论已经有很长的历史,在 20 世纪 60 年代末和 70 年代初,关于 goto 语句的用法争论比较激烈。主张从高级程序语言中去掉 goto 语句的人认为,goto 语句是对程序结构影响最大的一种有害的语句,他们的主要理由是: goto 语句使程序的静态结构和动态结构不一致,从而使程序难以理解,难以查错。去掉 goto 语句后,可直接从程序结构上反映程序运行的过程。这样,不仅使程序结构清晰,便于理解,便于查错,而且也有利于程序的正确性证明。持反对意见的人认为,goto 语句使用起来比较灵活,而且有些情形能提高程序的效率。若完全删去 goto 语句,有些情形反而会使程序过于复杂,增加一些不必要的计算量。后来,G·加科皮尼和 C·波姆从理论上证明:任何程序都可以用顺序、分支和重复结构表示出来。这个结论表明,SC 语言去掉 goto 语句并不影响它的编程能力,而且编写的程序结构更加清晰。

3.6.8 表达式

表达式由运算符、常量及变量构成,运算符是告诉编译程序执行特定算术或逻辑操作的符号,常量及变量为操作数。每个运算符在优先级层次结构中都有指定的优先级,并包含一个计算方向,运算符的计算方向就是运算符结合性。具有高优先级的运算符先于低优先级的运算符进行计算,如果复杂的表达式有多个运算符,则运算符优先级将确定执行操作的顺序,执行顺序可能对结果值有明显的影响。某些运算符具有相同的优先级,如果表达式包含多个具有相同的优先级的运算符,则按照从左到右或从右到左的方向进行运算。表 3.6 按从高到低的顺序列出了 SC 语言运算符的优先级,同一层上的运算符具有相同的优先级。

表3.6 SC语言运算符优先级表

优先级	运算符	含 义	要求运算对象的个数	结合方向
1	() [] -> .	圆括号 下标运算符 指向结构体成员运算符 结构体成员运算符		自左至右
2	- * & sizeof	负号运算符 指针运算符 地址与运算符 长度运算符	1 (单目运算符)	自右至左
3	*	乘法运算符	2	自左至右
	/	除法运算符		
	%	求余运算符	(双目运算符)	
4	+	加法运算符	2	自左至右
	-	减法运算符	(双目运算符)	
5	< <=	小于	2	自左至右
	>	小于等于		
	>=	大于	(双目运算符)	
		大于等于		
6	==	等于运算符	2	自左至右
	!=	不等于运算符	(双目运算符)	
7	=	赋值运算符	2	自右至左
8	,	逗号运算符 (顺序求值运算符)		自左至右

SC不支持的C语言运算符如表3.7所示。

表3.7 SC语言不支持的C语言运算符表

运算符	含 义	运算符	含 义
++	后置自增操作符		逻辑或操作符
--	后置自减操作符	?=	三元条件操作符
!	逻辑取反操作符	+ =	复合赋值操作符(加法)
~	按位取反(按位取补)	- =	复合赋值操作符(减法)
++	前置自增操作符	* =	复合赋值操作符(乘法)
--	前置自减操作符	/ =	复合赋值操作符(除法)
(类型)	类型转换操作符	% =	复合赋值操作符(取余)
<<	按位左移操作符	>> =	复合赋值操作符(按位与)
>>	按位右移操作符	<<=	复合赋值操作符(按位异或)
&	按位与操作符	& =	复合赋值操作符(按位或)
^	按位异或操作符	^ =	复合赋值操作符(按位左移)
	按位或操作符	=	复合赋值操作符(按位右移)
&&	逻辑与操作符	,	逗号操作符

第 4 章

SC 语言词法分析

不积跬步，无以至千里；不积小流，无以成江海。

——荀子

回想一下学习汉语的过程，首先学习点、横、竖、撇、捺等基本笔画，然后由这些基本笔画组成了一个个汉字。再回想一下学习英语的过程，首先学习 26 个英文字母，然后学习一个个单词。汉字的最小单位是这些笔顺，英文的最小单位是这些英文字母。本章的词法分析过程将实现从一串字节流中分析出一个个单词。看一下图 4.1 对本章要做的事情，及这件事情在整个编译过程扮演什么角色就非常清楚了。

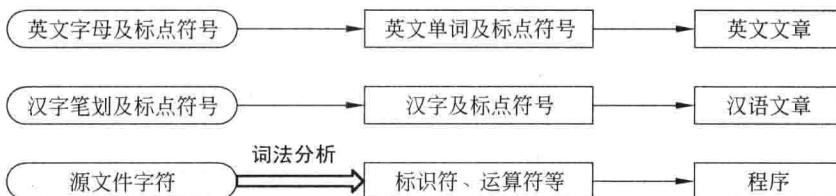


图 4.1 语言分析过程类比图

4.1 词法分析任务的官方说法

源程序是由字符序列构成的，词法分析器扫描源程序字符串，根据语言的词法规则分析并识别具有独立意义的最小语法单位：单词（包括关键字、运算符、标识符等），并以某种编码形式输出。词法分析器不关心单词之间的关系（属于语义分析的范畴），举例来说：词法分析器能够将括号识别为单词，但并不保证括号是否匹配。

词法分析是编译过程的第一步，是编译过程的基础。词法分析除了为语义分析提供单词流，滤掉编译过程不关心的内容以外，还有一个重要的作用是有了词法分析可以大大提高编译的效率。可能有人曾有过疑问，为什么一定要有词法分析？词法分析和语义分析的关系与其他编译过程有些不同，如语义分析，生成代码在编译过程中是独立的步骤与其他步骤有明显的区别。而词法分析和语义分析在形式上很相似，都要用文法去定义语言的结构。可以想象一下，如果把词法分析和语义分析合并会有什么不同，也就是说直接对源代码做语义分析，如 SC 源代码中当遇到一个字符 c，这时它可能是关键字 char、标识符 cl，等等。如果是 char 关键字那么接下来要分析一个字符变量或字符串定义，如果是标识符那要看它的具体上下文而定。这样的话与一个字符 c 有可能对应的情况太多了，所以要先做一遍词法分析把源程序的基础组成单位先分析出来。词法分析是语义分析的一个缓冲，可以大大提

高编译效率。

4.2 单词编码

下面来确定一下单词编号规则,可能你会问为什么要对单词进行编号,不编号不行吗?这就像每个人都有名字,为什么要有身份证号,答案是:这样操作管理起来更方便。并且每个人的编号还不止一个,上学了我们还会有学号,工作了还会有工号,这些编号是服务于不同时期不同目的的。单词编号也不止一个,一个是流水号,在程序中先到先得,当然这个规则只对标识符适用,关键字标点符号等属于特权阶层,它们在编译开始前就已经排在了单词表的最前面,排队规则对它们不适用。这就像我们去看一场演出,要求大家按顺序排队入场,这个规则也是针对普通观众的,对于演职人员跟VIP嘉宾是不会跟着大家这么拥挤着入场的。另一个编号是单词的哈希码,在4.3.3节会讲到。

流水号生成起来倒是简单,可是后面使用起来不直观,对我们的记忆力也是个考验,因此对编号采用助记符来代替。下面就来看看单词编码表,如表4.1所示。

表4.1 单词编码表

词法记号类型标识	含 义	词法记号类型标识	含 义
TK_PLUS	"+"	TK_CINT	整数
TK_MINUS	"-"	TK_CCHAR	一对单引号括起来的单个字符
TK_STAR	"*"	TK_CSTR	一对双引号括起来的字符序列
TK_DIVIDE	"/"	TK_EOF	文件结束符
TK_MOD	"%"	KW_CHAR	"char"
TK_EQ	"=="	KW_SHORT	"short"
TK_NEQ	"!="	KW_INT	"int"
TK_LT	"<"	KW_VOID	"void"
TK_LEQ	"<="	KW_STRUCT	"struct"
TK_GT	">"	KW_IF	"if"
TK_GEQ	">="	KW_ELSE	"else"
TK_ASSIGN	"="	KW_FOR	"for"
TK_POINTSTO	"->"	KW_CONTINUE	"continue"
TK_DOT	". "	KW_BREAK	"break"
TK_AND	"&."	KW_RETURN	"return"
TK_OPENPA	"("	KW_SIZEOF	"sizeof"
TK_CLOSEPA	")"	KW_CDECL	"__cdecl"
TK_OPENBR	"["	KW_STDCALL	"__stdcall"
TK_CLOSEBR	KW_PACK	"__pack"	

续表

词法记号类型标识	含 义	词法记号类型标识	含 义
TK_BEGIN	"{"	TK_IDENT	标识符,由字母、数字及下划线组成,且第一个字符必须为字母或下划线
TK_END	TK_IDENT+1		
TK_SEMICOLON	";"	:	
TK_COMMA	", "	TK_IDENT+n	

编码表已经看到了,可是 TK_PLUS 这些单词编号助记符怎么跟编号关联起来呢,我们希望 TK_PLUS 代表 0,TK_MINUS 代表 1,以此类推,怎么来实现呢?方法有多种,第 1 种将这些助词符逐个定义全局变量。第 2 种用 const 将这些助记符逐个定义为全局常量。第 3 种利用#define 对这些助记符进行宏定义。第 4 种利用枚举对这些助记符进行定义。这里采用第 4 种方法,好处读者可以自己思考总结一下。

```
/* 单词编码 */
enum e_TokenCode
{
    /* 运算符及分隔符 */
    TK_PLUS,           // + 加号
    TK_MINUS,          // - 减号
    TK_STAR,           // * 星号
    TK_DIVIDE,         // / 除号
    TK_MOD,            // % 求余运算符
    TK_EQ,              // == 等于号
    TK_NEQ,             // != 不等于号
    TK_LT,              // < 小于号
    TK_LEQ,             // <= 小于等于号
    TK_GT,              // > 大于号
    TK_GEQ,             // >= 大于等于号
    TK_ASSIGN,           // = 赋值运算符
    TK_POINTSTO,        // -> 指向结构体成员运算符
    TK_DOT,              // . 结构体成员运算符
    TK_AND,              // & 地址与运算符
    TK_OPENPA,           // ( 左圆括号
    TK_CLOSEPA,          // ) 右圆括号
    TK_OPENBR,           // [ 左中括号
    TK_CLOSEBR,          // ] 右中括号
    TK_BEGIN,             // { 左大括号
    TK_END,              // } 右大括号
    TK_SEMICOLON,        // ; 分号
    TK_COMMA,             // , 逗号
    TK_ELLIPSIS,          // ... 省略号
    TK_EOF,               // 文件结束符
};
```

```

/* 常量 */
TK_CINT,           //整型常量
TK_CCHAR,          //字符常量
TK_CSTR,           //字符串常量

/* 关键字 */
KW_CHAR,           //char 关键字
KW_SHORT,          //short 关键字
KW_INT,            //int 关键字
KW_VOID,           //void 关键字
KW_STRUCT,         //struct 关键字
KW_IF,              //if 关键字
KW_ELSE,            //else 关键字
KW_FOR,             //for 关键字
KW_CONTINUE,        //continue 关键字
KW_BREAK,           //break 关键字
KW_RETURN,          //return 关键字
KW_SIZEOF,          //sizeof 关键字

KW_ALIGN,           //__align 关键字
KW_CDECL,           //__cdecl 关键字
KW_STDCALL,         //__stdcall 关键字

/* 标识符 */
TK_IDENT
};


```

4.3 词法分析用到的数据结构

在设计词法分析用到的数据结构时要考虑以下几个问题。

(1) 单词长度差别很大,有单字节的,也有多达几十字节,单词字节个数在SC词法描述中并没有规定上限,理论上可以任意长度,因此单词字符串客观上要求用动态字符串来存储。当然我们是编译器的设计者,有权力在语言词法描述基础上,对单词长度加以限制,例如规定最多64字节,即使这样如果所有单词都采用定长字符串来存储也会造成内存的巨大浪费。

(2) 常量字符串长度,无法预知,可能为空字符串,也可能特别长,因此也必须采用动态字符串来存储。

(3) 单词个数无法预知,可能只有一个单词,也可能成千上万个。对于单词的存储结构,为了贯彻“厉行节约,反对浪费”的精神,这里决定不搞“一刀切”(统一分配一个足够大的静态数组),而是采用按需分配的动态数组。

(4) 单词表中不存储重复单词,因此每遇到一个单词都要到单词表中去查找,查找操作将是非常频繁的,如果采用普通的遍历查找效率将是非常低的,因此借助哈希表来提高查找

效率,它可以快速地查询一个对象在内存里的位置,进行引用。

下面一起来设计动态字符串、动态数组、哈希表结构及其支持的相应操作,在本节的最后压轴介绍将是单词表,可以看到它是由动态数组和哈希表复合组成的。

4.3.1 动态字符串

C语言字符串是以连续的字节流表示的,即字符数组,并且以'\0'结尾,C语言标准库中也提供了很多函数来操作这种形式的字符串,例如,求字符串长度 `strlen()`、求子串 `strstr()`、字符串拷贝 `strcpy()`、字符串连接 `strcat()`,等等,但是,这些函数并不安全,很可能给系统或应用程序带来严重的问题,如栈溢出等,C语言字符串中并没有记录操作系统为其分配的长度,用户必须自己将字符串长度保存在其他变量中,很明显如果操作不当就会产生错误,如缓冲区溢出。

C语言提供的字符串及其操作函数,对我们使用起来并不怎么方便,每追加一个字符都要考虑内存分配,缓冲区溢出问题,这对我们来说是个巨大的负担。可是C语言标准库中并没有提供动态字符串结构及其操作函数,下面我们来自己实现一个简单的动态字符串,那么将采用什么样的数据结构既支持动态扩展,又可以避免缓冲区溢出问题呢?请看下面定义的动态字符串数据结构:

```
/* 动态字符串定义 */
typedef struct DynString
{
    int count;           //字符串长度
    int capacity;        //包含该字符串的缓冲区长度
    char * data;         //指向字符串的指针
} DynString;
```

在动态字符串结构中,`count`记录了字符串长度,`data`为指向字符串的指针,这两个变量比较容易理解。可是为什么还有`capacity`字符串的缓冲区长度这个成员变量?后面字符串内存分配相关函数会看到这个成员变量有什么用。

下面来写用到的一组动态字符串操作函数,包括初始化动态字符串存储容量 `dynstring_init`、释放动态字符串使用的内存空间 `dynstring_free`、重置动态字符串 `dynstring_reset`、重新分配字符串容量 `dynstring_realloc`、追加单个字符到动态字符串对象 `dynstring_chcat`。

```
*****
* 初始化动态字符串存储容量
* pstr: 动态字符串存储结构
* initSize: 字符串初始化分配空间
*****
void dynstring_init(DynString * pstr, int initSize)
{
    if(pstr != NULL)
    {
        pstr->data = (char *)malloc(sizeof(char) * initSize);
```

```
pstr->count=0;
pstr->capacity=initsize;
}

}

/***********************
* 释放动态字符串使用的内存空间
* pstr: 动态字符串存储结构
*************************/
void dynstring_free(DynString * pstr)
{
    if(pstr !=NULL)
    {
        if(pstr->data)
            free(pstr->data);
        pstr->count=0;
        pstr->capacity=0;
    }
}

/***********************
* 重置动态字符串,先释放,重新初始化
* pstr: 动态字符串存储结构
*************************/
void dynstring_reset(DynString * pstr)
{
    dynstring_free(pstr);
    dynstring_init(pstr,8);      //字符串初始化分配空间 8 字节
}

/***********************
* 重新分配字符串容量
* pstr: 动态字符串存储结构
* new_size: 字符串新长度
*************************/
void dynstring_realloc(DynString * pstr, int new_size)
{
    int capacity;
    char * data;

    capacity=pstr->capacity;
    while(capacity <new_size)
        capacity=capacity * 2;
    data=realloc(pstr->data, capacity);
    if(!data)
```

```

        error("内存分配失败");
        pstr->capacity=capacity;
        pstr->data=data;
    }

/***********************
 * 追加单个字符到动态字符串对象
 * pstr: 动态字符串存储结构
 * ch: 所要追加的字符
*************************/
void dynstring_chcat(DynString * pstr, int ch)
{
    int count;
    count=pstr->count +1;
    if(count >pstr->capacity)
        dynstring_realloc(pstr, count);
    ((char *)pstr->data)[count -1]=ch;
    pstr->count=count;
}

```

上面每个函数都有注释,这里不再逐一解释。这里着重说一下动态字符串的内存分配,动态字符串的“动态”内存分配归根结底是靠 realloc 函数实现的,这个函数每执行一次都要把原先的数据向新申请的内存位置拷贝一遍,这个过程将是非常影响效率的。因此动态字符串初始化函数 dynstring_init 的第二个参数 initsize 可以由调用者来指定初始化大小,如果估计 10 字节容量就够了,你就调用 dynstring_init 函数时将初始化大小设为 10。可能你会问那万一要是超出我的初始化大小呢,这时我们看到在 dynstring_chcat 函数中,每追加一个字符都会进行字符串实际长度与缓冲区容量的对比,如果发现字符串缓冲区容量已经不足以容纳要存储的字符串,就会调用 dynstring_realloc 动态扩展缓冲区容量,并且扩展时不是需要一个字节就扩展一个字节,而是成倍扩展,需要 10 个就扩展为 20 字节,可能有的读者要较真儿了,前面不是说了动态字符串内存实行按需分配嘛,这里不还是多分配了吗?这里说一下按需分配是原则,需要 10 个给 20 个是策略问题,是权衡了避免内存浪费与 realloc 频繁重分配导致的效率下降,矛盾斗争的结果。

4.3.2 动态数组

熟悉了前面的动态字符串实现,动态数组就好理解多了,下面来看一下的动态数组结构定义:

```

/* 动态数组定义 */
typedef struct DynArray
{
    int count;           //动态数组元素个数
    int capacity;        //动态数组缓冲区长度
    void **data;         //指向数据指针数组
} DynArray;

```

可以看到,动态数组定义与动态字符串结构的唯一区别是动态字符串结构的字符数组,而动态数组存储的指针数组,这些指针可以指向任意类型的数据。可能有的读者会问为什么动态数组不直接存储数据数组呢?这个问题等大家看完了下面的动态数组操作函数,作者再回答。

```
*****  
* 重新分配动态数组容量  
* parr: 动态数组存储结构  
* new_size: 动态数组最新元素个数  
*****  
void dynarray_realloc(DynArray * parr, int new_size)  
{  
    int capacity;  
    void * data;  
  
    capacity=parr->capacity;  
    while(capacity < new_size)  
        capacity=capacity * 2;  
    data=realloc(parr->data, capacity);  
    if(!data)  
        error("内存分配失败");  
    parr->capacity=capacity;  
    parr->data=data;  
}  
  
*****  
* 追加动态数组元素  
* parr: 动态数组存储结构  
* data: 所要追加的新元素  
*****  
void dynarray_add(DynArray * parr, void * data)  
{  
    int count;  
    count=parr->count +1;  
    if(count * sizeof(void*) >parr->capacity)  
        dynarray_realloc(parr, count * sizeof(void*));  
    parr->data[count-1]=data;  
    parr->count=count;  
}  
  
*****  
* 初始化动态数组存储容量  
* parr: 动态数组存储结构  
* initsize: 动态数组初始化分配空间  
*****
```

```

void dynarray_init(DynArray * parr, int initsize)
{
    if(parr !=NULL)
    {
        parr->data= (void**)malloc(sizeof(char) * initsize);
        parr->count=0;
        parr->capacity=initsize;
    }
}

/****************************************
* 释放动态数组使用的内存空间
* parr: 动态数组存储结构
****************************************/
void dynarray_free(DynArray * parr)
{
    void **p;
    for(p=parr->data; parr->count; ++p, --parr->count)
        if(* p)
            free(* p);
    free(parr->data);
    parr->data=NULL;
}

/****************************************
* 动态数组元素查找
* parr: 动态数组存储结构
* key: 要查找的元素
****************************************/
int dynarray_search(DynArray * parr, int key)
{
    int i;
    int **p;
    p= (int**)parr->data;
    for(i=0; i <parr->count; ++i, p++)
        if(key==*p)
            return i;
    return -1;
}

```

现在来回答为什么动态数组存储的是指针数组，而不是数据数组。可能有的读者首先想到的是各种数据类型尺寸不一样，没法通用，这个回答可不算个好理由，因为数据类型尺寸不一样，可以再加一个成员变量来存储数据类型长度不就可以了。真正的理由还是看一下重新分配动态数组容量 `dynarray_realloc` 函数，可以看到调用 `realloc` 进行内存重新分配时，移动的仅仅是指针数组，而数据没有移动，这种情况对于存储数据类型长度比较大的数

据优势就很明显了,因为内存重分配的次数及每次重分配时移动的数据量并没有随存储数据长度的增大而增加。

在本节的最后,为了让大家对动态字符串及动态数组有个更直观的认识,更清楚地看到有什么区别,这里给出动态字符串及动态数组的存储结构图(见图 4.2),请大家看一看。

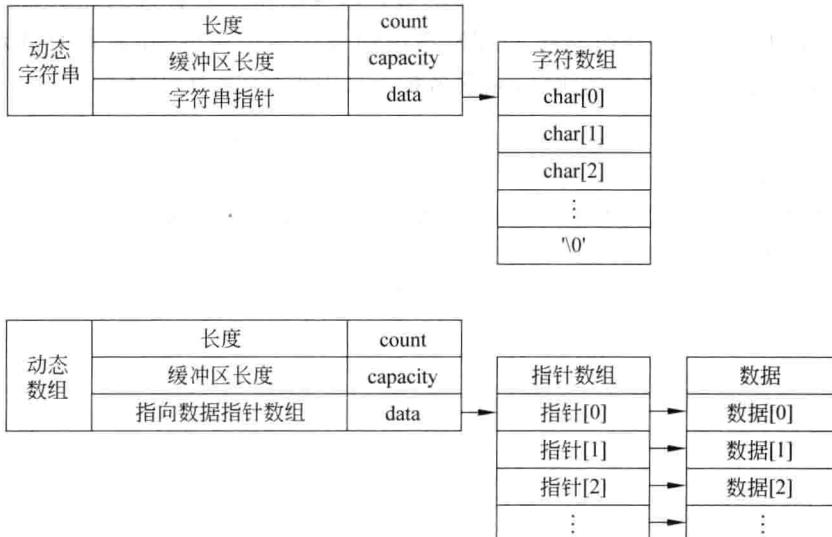


图 4.2 动态字符串与动态数组存储结构图

4.3.3 哈希表

学过数据结构的,都应该知道,线性表和树中,记录在结构中的相对位置是随机的,记录和关键字之间不存在明确的关系,因此在查找记录的时候,需要进行一系列的关键字比较,这一类查找方法建立在“比较”的基础上,查找的效率与比较次数密切相关。理想的情况是能直接找到需要的记录,因此必须在记录的存储位置和它的关键字之间建立一确定的对应关系 f ,使每个关键字和结构中一个唯一的存储位置相对应。因而查找时,只需根据这个对应关系 f 找到给定值 K 的像 $f(K)$ 。若结构中存在关键字和 K 相等的记录,则必定在 $f(K)$ 的存储位置上,由此不需要进行比较便可直接取得所查记录。在此,称这个对应关系 f 为哈希函数,按这个思想建立的表为哈希表(又称为杂凑法或散列表)。

哈希表不可避免冲突(collision)现象:对不同的关键字可能得到同一哈希地址,即 $\text{key1} \neq \text{key2}$,而 $f(\text{key1}) = f(\text{key2})$ 。具有相同函数值的关键字对该哈希函数来说称为同义词(synonym)。因此,在建造哈希表时不仅要设定一个好的哈希函数,而且要设定一种处理冲突的方法。常用的方法有开放地址法、再哈希法、链地址法、建立一个公共溢出区法。我们决定采用链地址法,它将所有关键字为同义词的记录存储在同一线性链表中。在 4.3.4 节的单词存储结构定义中,会看到记录同义词的成员变量。

有了上面的铺垫,可以如下描述哈希表:根据设定的哈希函数 $f(\text{key})$ 和所选中的处理冲突的方法,将一组关键字映像到一个有限的、地址连续的地址集(区间)上并以关键字在地址集中的“像”作为相应记录在表中的存储位置,这种表称为哈希表。这个函数 $f(\text{key})$ 为哈希函数。哈希函数是一个映像,即将关键字的集合映射到某个地址集合上,它的设置很灵

活,只要这个地址集合的大小不超出允许范围即可。现实中哈希函数是需要构造的,并且构造得好才能使用得好。

我们需要的哈希函数是以单词字符串为关键字进行哈希值计算,字符串哈希函数非常多,常见的主要有 Simple_hash、RS_hash、JS_hash、PJW_hash、ELF_hash、BKDR_hash、SDBM_hash、DJB_hash、AP_hash、CRC_hash 等。这些哈希函数各自的优缺点作者没有详细考究,其中,ELF_Hash 函数是在 UNIX 系统中被广泛使用的,作者最终选定了 ELF_Hash,为了与本书函数命名规范统一,函数名统一采用小写,定为 elf_hash。

```
/**************************************************************************
 * 计算哈希地址
 * key: 哈希关键字(为了与本书中 sc 语言关键字区分开,此处称为哈希关键字)
 * MAXKEY: 哈希表长度
 **************************************************************************/
int elf_hash(char * key)
{
    int h=0, g;
    while(*key)
    {
        h= (h<<4)+ *key++;
        g=h & 0xf0000000;
        if(g)
            h ^=g>>24;
        h &= ~g;
    }
    return h%MAXKEY;
}
```

通过上面的函数就可以得到上节提到的单词的另一个编码——哈希码。哈希表就介绍到这里,由于哈希表不是一个独立的数据结构,只是对原始数据记录的一个索引,没有独立的操作函数,相关的插入,查找操作函数及带有哈希表的单词表长什么样,将在 4.3.4 节中介绍。

4.3.4 单词表

下面开始介绍词法分析最重要的数据结构——单词表,首先给出单词存储结构定义如下:

```
/* 单词存储结构定义 */
typedef struct TkWord
{
    int tkcode;                      //单词编码
    struct TkWord * next;           //指向哈希冲突的同义词
    char * spelling;                //单词字符串
    struct Symbol * sym_struct;     //指向单词所表示的结构定义
    struct Symbol * sym_identifier; //指向单词所表示的标识符
} TkWord;
```

从上面定义可以看出,每个单词信息由5部分组成,tkcode为4.2节介绍的单词流水号,next记录哈希冲突的其他单词,spelling为指向单词字符串的指针,另外两个成员变量分别指向单词所表示的结构定义和标识符,具体作用将在6.2.2.2节介绍。接下来介绍单词表用到的全局变量及相关操作函数。

全局变量:

```
*****
#define MAXKEY 1024          //哈希表容量
TkWord *tk_hashtable[MAXKEY]; //单词哈希表
DynArray tktable;           //单词表
*****
```

单词表操作函数:

```
*****
* 功能: 运算符、关键字、常量直接放入单词表
*****
TkWord * tkword_direct_insert(TkWord * tp)
{
    int keyno;
    dynarray_add(&tktable, tp);
    keyno=elf_hash(tp->spelling);
    tp->next=tk_hashtable[keyno];
    tk_hashtable[keyno]=tp;
    return tp;
}

*****
* 功能: 在单词表中查找单词
* p: 要查找的单词
* keyno: 要查找单词的哈希值
*****
TkWord * tkword_find(char * p, int keyno)
{
    TkWord * tp=NULL, * tp1;
    for(tp1=tk_hashtable[keyno];tp1;tp1=tp1->next)
    {
        if(!strcmp(p,tp1->spelling))
        {
            token=tp1->tkcode;
            tp=tp1;
        }
    }
    return tp;
}
```

```

/***********************
* 功能：标识符插入单词表，先查找，查找不到再插入单词表
***********************/
TkWord * tkword_insert(char * p)
{
    TkWord * tp;
    int keyno;
    char * s;
    char * end;
    int length;

    keyno=elf_hash(p);
    tp=tkword_find(p,keyno);
    if(tp==NULL)
    {
        length=strlen(p);
        tp=(TkWord *)mallocz(sizeof(TkWord)+length+1);
        tp->next=tk_hashtable[keyno];
        tk_hashtable[keyno]=tp;
        dynarray_add(&tkttable,tp);
        tp->tkcode=tkttable.count-1;
        s=(char *)tp+sizeof(TkWord);
        tp->spelling=(char *)s;
        for(end=p+length;p<end;)
        {
            *s++=*p++;
        }
        *s=(char)'\0';
    }
    return tp;
}

/***********************
* 功能：分配堆内存并将数据初始化为'0'
* size: 分配内存大小
***********************/
void * mallocz(int size)
{
    void * ptr;
    ptr=malloc(size);
    if(!ptr && size)
        error("内存分配失败");
    memset(ptr,0,size);
    return ptr;
}

```

从代码可以看出,tkword_direct_insert 函数是关键字、运算符等特权阶层专用的进入单词表的入口,tkword_insert 函数是普通标识符这些平民阶层进入单词表的入口,看来森严的等级制度不仅存在于社会领域,编程语言的单词也是要分三六九等的。再看一下 tkword_find 函数,就是前面说的哈希表查找函数,这个函数有两个参数,第一个参数是要查找的单词,第二个参数是单词的哈希码,根据哈希码从哈希表中得到单词的指针,如果指针为空说明没找到返回空指针,不为空时与要查找单词匹配直接返回,如果不匹配沿着该单词的 next 链顺藤摸瓜地去找哈希冲突的同义词,直到 next 为 NULL。

词法分析的初始化函数本应该放在 4.5 节词法分析过程中讲,但是这个函数其实就干了一件事——单词表的初始化,所以这个函数也放在本节来讲一下。

```
/****************************************************************************
 * 功能: 词法分析初始化
 ****/
void init_lexer()
{
    TkWord * tp;
    static TkWord keywords[] = {
        {TK_PLUS,      NULL,    "+",           NULL,   NULL},
        {TK_MINUS,     NULL,    "-",           NULL,   NULL},
        {TK_STAR,      NULL,    "*",           NULL,   NULL},
        {TK_DIVIDE,    NULL,    "/",           NULL,   NULL},
        {TK_MOD,       NULL,    "%",           NULL,   NULL},
        {TK_EQ,        NULL,    "==",          NULL,   NULL},
        {TK_NEQ,       NULL,    "!=" ,          NULL,   NULL},
        {TK_LT,        NULL,    "<",          NULL,   NULL},
        {TK_LEQ,       NULL,    "<=",         NULL,   NULL},
        {TK_GT,        NULL,    ">",          NULL,   NULL},
        {TK_GEQ,       NULL,    ">=",         NULL,   NULL},
        {TK_ASSIGN,    NULL,    "=" ,          NULL,   NULL},
        {TK_POINTSTO,  NULL,    "->",          NULL,   NULL},
        {TK_DOT,       NULL,    ".",            NULL,   NULL},
        {TK_AND,       NULL,    "&",          NULL,   NULL},
        {TK_OPENPA,    NULL,    "(" ,          NULL,   NULL},
        {TK_CLOSEPA,   NULL,    ")" ,          NULL,   NULL},
        {TK_OPENBR,    NULL,    "[" ,          NULL,   NULL},
        {TK_CLOSEBR,   NULL,    "]" ,          NULL,   NULL},
        {TK_BEGIN,     NULL,    "{" ,          NULL,   NULL},
        {TK_END,       NULL,    "}" ,          NULL,   NULL},
        {TK_SEMICOLON, NULL,    ";" ,          NULL,   NULL},
        {TK_COMMMA,    NULL,    "," ,          NULL,   NULL},
        {TK_ELLIPSIS,  NULL,    "...",         NULL,   NULL},
        {TK_EOF,       NULL,    "End_Of_File", NULL,   NULL},
    };
}
```

```

    {TK_CINT,      NULL,  "整型常量",    NULL,  NULL},
    {TK_CCHAR,     NULL,  "字符常量",    NULL,  NULL},
    {TK_CSTR,      NULL,  "字符串常量",  NULL,  NULL},

    {KW_CHAR,      NULL,  "char",        NULL,  NULL},
    {KW_SHORT,     NULL,  "short",       NULL,  NULL},
    {KW_INT,        NULL,  "int",         NULL,  NULL},
    {KW_VOID,       NULL,  "void",        NULL,  NULL},
    {KW_STRUCT,    NULL,  "struct",      NULL,  NULL},

    {KW_IF,         NULL,  "if",          NULL,  NULL},
    {KW_ELSE,       NULL,  "else",        NULL,  NULL},
    {KW_FOR,        NULL,  "for",         NULL,  NULL},
    {KW_CONTINUE,   NULL,  "continue",    NULL,  NULL},
    {KW_BREAK,      NULL,  "break",       NULL,  NULL},
    {KW_RETURN,     NULL,  "return",      NULL,  NULL},
    {KW_SIZEOF,     NULL,  "sizeof",      NULL,  NULL},
    {KW_ALIGN,      NULL,  "__align",     NULL,  NULL},
    {KW_CDECL,      NULL,  "__cdecl",     NULL,  NULL},
    {KW_STDCALL,   NULL,  "__stdcall",   NULL,  NULL},
    {0,             NULL,  NULL,          NULL,  NULL}
};

dynarray_init(&tktable, 8);
for(tp=&keywords[0];tp->spelling !=NULL;tp++)
    tkword_direct_insert(tp);

}

```

单词表的初始化都做了什么工作呢,好像对关键字、运算符等做了些什么特殊操作,是的,上面这段代码让关键字、运算符这些特权阶层在正式词法分析之前优先进入单词表,坐在了最前面的嘉宾席,后面会看到普通的标识符是如何排队进入单词表的。为了让大家对单词表存储结构有个更直观的认识,作者为大家绘制了单词表存储结构图(见图 4.3)供大家品鉴。

从图 4.3 可以看出,单词表是由单词动态数组和哈希表组成的一个复合结构。这里以标识符 X 为例讲一下单词插入符号表及查找过程,在源代码中遇到一个标识符 X,经计算其哈希码为 N,取出哈希表的第 N 个元素值,如果这个值为 NULL,说明单词表中还没有名为 X 的标识符,我们利用 tkword_insert 做两件事,第一件事是将标识符 X 放在单词动态数组中,第二件将哈希表的第 N 个元素值设为指向单词 X 的指针。等下次再遇到标识符 X,用 tkword_find 函数查找时发现,tk_hashtable[n] 值不为空,通过 tk_hashtable[N] 得到单词指针,就可以从单词动态数组中找到单词 X。

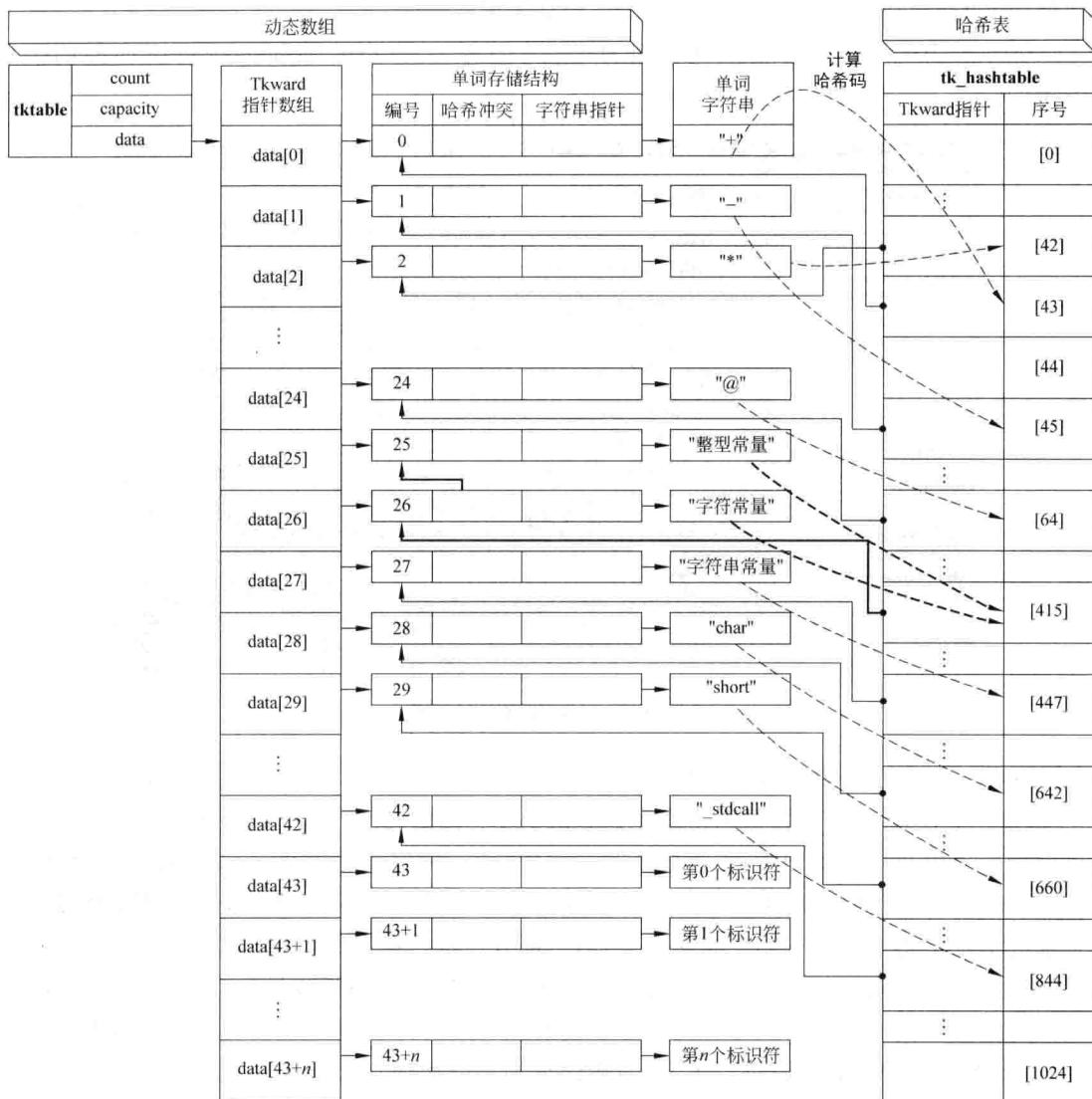


图 4.3 单词表存储结构图

4.4 错误处理,未雨绸缪

试想写了一个 10 000 行的程序让编译器给编译一下,编译过程中遇到错误直接退出了,没有给出任何错误提示,怎么去找错误?在这么长程序里去找个错误,这活儿的难度堪比大海捞针,其难度可想而知!为了让大家对错误处理在编译程序中的角色地位有个更直观的认识,再来打个比方,如果把我们的词法分析、语法分析、语义分析比作一支部队的主力军,错误处理程序则是这支部队的后勤服务部门,你可能会感觉后勤部门不就是打杂的吗?可不要小看这些打杂的工作,试想一支部队如果没有负责运送粮草给养的,没有烧火做饭的这些后勤部门,这支部队能打胜仗吗?看到这里相信大家已经认识到错误处理程序作

为编译器的后勤服务部门,不是可有可无,而是不可或缺的。

先不着急写 SC 编译器的错误处理程序,先看一下 VC6 如何进行错误处理的(见图 4.4)。

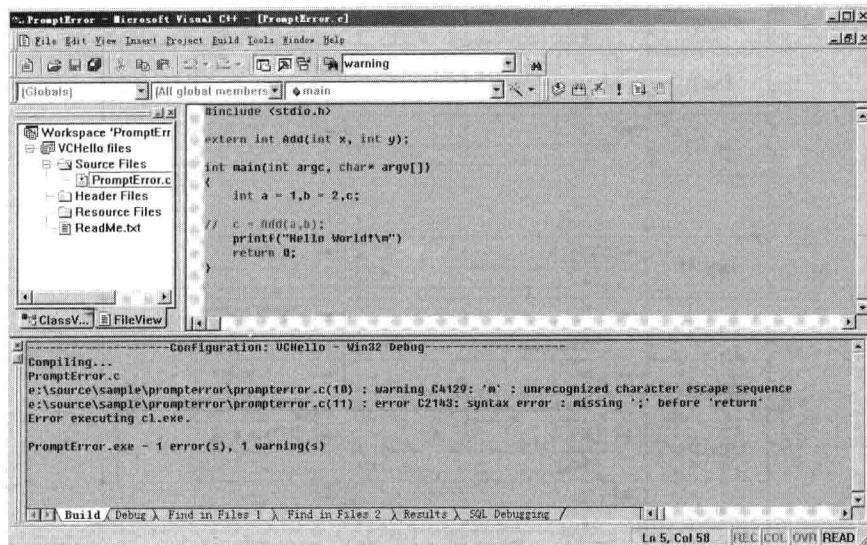


图 4.4 VC6 编译错误处理

从图 4.4 可以看出,编译器提示 prompterror.c 文件中,第 10 行有一个警告,第 11 行有一个错误。第 10 行的警告是因为编译器不认识'\m'后面的转义字符'm',检查一下程序,原来是因为本来想写个"\n"回车符,写成了"\m"。第 11 行的错误是因为 return 前缺少';'分号,检查一下前一行代码结尾果然缺个分号。来分析一下 VC 编译过程错误由几部分组成,第一部分提示了错误发生在哪个源文件中,第二部分提示了错误发生在第几行,第三部分表示了错误发生的级别,即分为警告与错误,另外还对错误进行了编码,上面的 C4129、C2143,第四部分描述了错误原因。这里有两个小小的疑问,疑问一,为什么要对错误进行编码,疑问二,编码用错误编号表示就可以了,为何错误编码的前面还加个字母"C"。现将光标定位在第一条错误处,按 F1 键,弹出 MSDN 关于 C4149 的帮助,如图 4.5 所示。

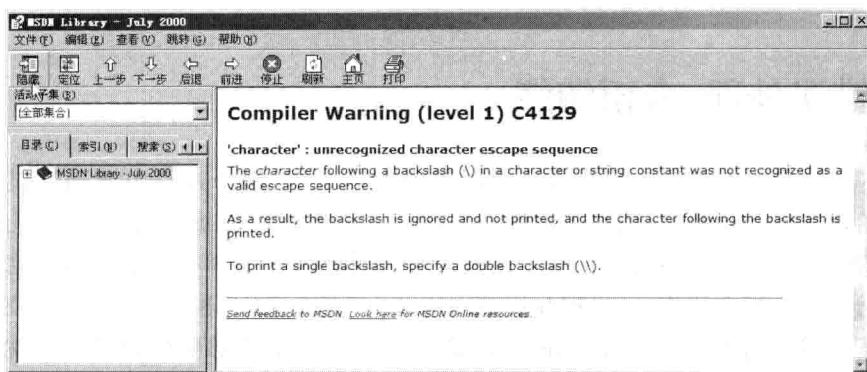


图 4.5 VC6 错误处理帮助

看了图4.5所示的第一个疑问基本上就有答案了,对错误进行编码是为了在编译器的帮助下进一步详细解释这个编号的错误是什么原因引起的,编译器会针对此错误做什么处理。疑问二留待讲解链接错误时一起讲,接下来将前面的代码稍微修改一下,再看一下图4.6所示的错误提示。

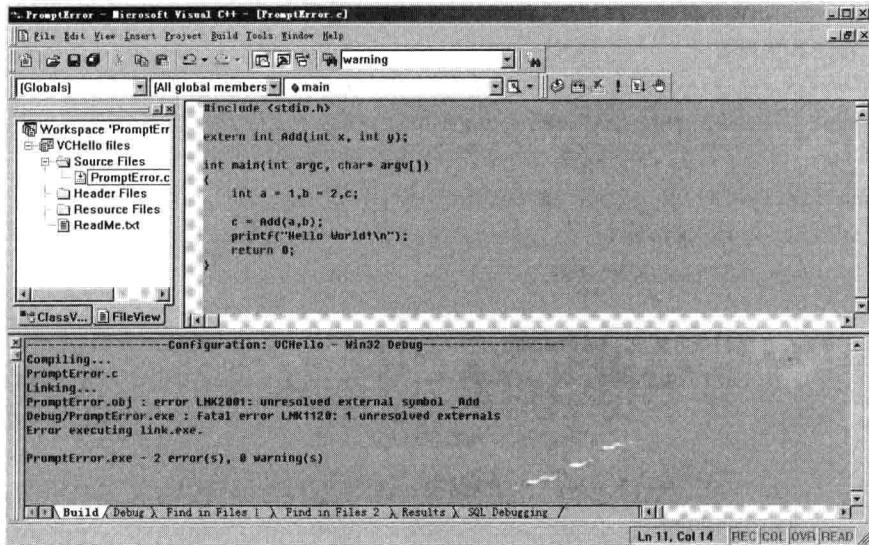


图4.6 VC6链接错误处理

图4.6是一个典型的链接错误提示,提示“_Add外部符号未找到”,从这个提示可以知道,这个错误是因为Add只给出了声明,而从链接的各个库文件中找不到这个函数的定义。现在对比一下链接错误编码和前面的编译错误编码,发现编译的错误编码以“C”开头,链接错误的错误编码以“LNK”开头,联想一下“C”不就是“Compile”编译这个单词的首字母,“LNK”不就是“Link”这个单词的缩写,到这里前面的疑问二也就解开了。所以错误提示中其实是包含了阶段信息,即错误是在编译阶段发生的,还是在链接阶段出现的。分析清楚VC编译器的错误处理,下面来编写SC编译器的错误处理程序。

```
/* 错误处理程序用到的枚举定义 */
/* 错误级别 */
enum e_ErrorLevel
{
    LEVEL_WARNING,
    LEVEL_ERROR,
};

/* 工作阶段 */
enum e_WorkStage
{
    STAGE_COMPILE,
    STAGE_LINK,
};
```

```

*****  

* 异常处理  

* stage: 编译阶段还是链接阶段  

* level: 错误级别  

* fmt: 参数输出的格式  

* ap: 可变参数列表  

*****/  

void handle_exception(int stage,int level,char * fmt,va_list ap)
{
    char buf[1024];
    vsprintf(buf,fmt,ap);
    if(stage==STAGE_COMPILE)
    {
        if(level==LEVEL_WARNING)
            printf("%s(第%d行): 编译警告: %s!\n",filename,line_num,buf);
        else
        {
            printf("%s(第%d行): 编译错误: %s!\n",filename,line_num,buf);
            exit(-1);
        }
    }
    else
    {
        printf("链接错误: %s!\n",buf);
        exit(-1);
    }
}

*****  

* 编译警告处理  

* fmt: 参数输出的格式  

* ap: 可变参数列表  

*****/  

void warning(char * fmt,...)
{
    va_list ap;

    va_start(ap,fmt);
    handle_exception(STAGE_COMPILE,LEVEL_WARNING,fmt,ap);
    va_end(ap);
}

*****  

* 编译致命错误处理

```

```
* fmt: 参数输出的格式
* ap: 可变参数列表
*****
void error(char * fmt,...)
{
    va_list ap;
    va_start(ap,fmt);
    handle_exception(STAGE_COMPILE,LEVEL_ERROR,fmt,ap);
    va_end(ap);
}

*****
* 提示错误,此处缺少某个语法成分
* msg: 需要什么语法成分
*****
void expect(char * msg)
{
    error("缺少%s",msg);
}

*****
* 跳过单词 c,取下一单词,如果当前单词不是 c,提示错误
* c: 要跳过的单词
*****
void skip(int c)
{
    if(token !=c)
        error("缺少'%s'",get_tkstr(c));
    get_token();
}

*****
* 功能:取得单词 v 所代表的源码字符串
* v:    单词编号
*****
char * get_tkstr(int v)
{
    if(v>tkttable.count)
        return NULL;
    else if(v>=TK_CINT && v<=TK_CSTR)
        return sourcestr.data;
    else
        return ((TkWord*)tkttable.data[v])->spelling;
}
```

```
*****
* 链接错误处理
* fmt: 参数输出的格式
* ap: 可变参数列表
*****
void link_error(char * fmt, ...)
{
    va_list ap;
    va_start(ap,fmt);
    handle_exception(STAGE_LINK, LEVEL_ERROR, fmt, ap);
    va_end(ap);
}
```

通过图 4.7 来看一下 SCC 编译器的错误处理系统, 这张图是对前面代码的最好解释, 从图中可以看出错误分为 3 大类: 编译警告、编译致命错误、链接错误。

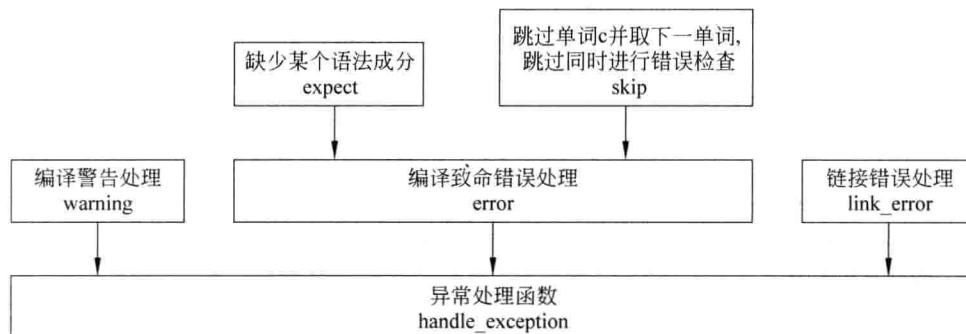


图 4.7 SCC 编译器的错误处理系统

4.5 词法分析过程

词法分析的核心功能是为语法分析程序提供一个个单词符号, 取单词程序命名为 `get_token()`, 这个函数委托其他一些函数来完成各种单词的解析工作, 这些函数主要有注释解析函数 `parse_comment`, 空白字符处理函数 `skip_white_space`, 标识符解析函数 `parse_identifier`, 整型常量解析函数 `parse_num`, 字符常量、字符串常量解析函数 `parse_string`。下面来看一下这些词法分析函数, 每个函数都有注释, 没有特别需要说明的作者不再逐一解释。

4.5.1 词法分析主程序

```
*****
* 功能： 取单词
*****
void get_token()
{
    preprocess();
    switch(ch)
```

```
{  
    case 'a': case 'b': case 'c': case 'd': case 'e': case 'f': case 'g':  
    case 'h': case 'i': case 'j': case 'k': case 'l': case 'm': case 'n':  
    case 'o': case 'p': case 'q': case 'r': case 's': case 't':  
    case 'u': case 'v': case 'w': case 'x': case 'y': case 'z':  
    case 'A': case 'B': case 'C': case 'D': case 'E': case 'F': case 'G':  
    case 'H': case 'I': case 'J': case 'K': case 'L': case 'M': case 'N':  
    case 'O': case 'P': case 'Q': case 'R': case 'S': case 'T':  
    case 'U': case 'V': case 'W': case 'X': case 'Y': case 'Z':  
    case '_':  
    {  
        TkWord * tp;  
        parse_identifier();  
        tp=tkword_insert(tkstr.data);  
        token=tp->tkcode;  
        break;  
    }  
    case '0': case '1': case '2': case '3':  
    case '4': case '5': case '6': case '7':  
    case '8': case '9':  
        parse_num();  
        token=TK_CINT;  
        break;  
    case '+':  
        getch();  
        token=TK_PLUS;  
        break;  
    case '-':  
        getch();  
        if(ch=='>')  
        {  
            token=TK_POINTSTO;  
            getch();  
        }  
        else  
            token=TK_MINUS;  
        break;  
    case '/':  
        token=TK_DIVIDE;  
        getch();  
        break;  
    case '%':  
        token=TK_MOD;  
        getch();  
        break;  
    case '=':  
        getch();
```

```

if(ch=='=')
{
    token=TK_EQ;
    getch();
}
else
    token=TK_ASSIGN;
break;

case '!':
getch();
if(ch=='=')
{
    token=TK_NEQ;
    getch();
}
else
    error("暂不支持'!'(非操作符)");
break;

case '<':
getch();
if(ch=='=')
{
    token=TK_LEQ;
    getch();
}
else
    token=TK_LT;
break;

case '>':
getch();
if(ch=='=')
{
    token=TK_GEQ;
    getch();
}
else
    token=TK_GT;
break;

case '.':
getch();
if(ch=='.')
{
    getch();
    if(ch != '.')
        error("省略号拼写错误");
    else
        token=TK_ELLIPSIS;
}

```

```
    getch();
}
else
{
    token=TK_DOT;
}
break;
case '&':
    token=TK_AND;
    getch();
    break;
case ';':
    token=TK_SEMICOLON;
    getch();
    break;
case ']':
    token=TK_CLOSEBR;
    getch();
    break;
case ')':
    token=TK_END;
    getch();
    break;
case '[':
    token=TK_OPENBR;
    getch();
    break;
case '{':
    token=TK_BEGIN;
    getch();
    break;
case ',':
    token=TK_COMMA;
    getch();
    break;
case '(':
    token=TK_OPENPA;
    getch();
    break;
case '*':
    token=TK_STAR;
    getch();
    break;
```

```

        case '\'':
            parse_string(ch);
            token=TK_CCHAR;
            tkvalue= * (char *)tkstr.data;
            break;
        case '\"':
        {
            parse_string(ch);
            token=TK_CSTR;
            break;
        }
        case EOF:
            token=TK_EOF;
            break;
        default:
            error("不认识的字符: \\x%02x", ch);
            getch();
            break;
    }
}

/******************
* 功能: 从 sc 源文件中读取一个字符
*****************/
void getch()
{
    ch=getc(fin);           //文件尾返回 EOF, 其他返回实际字节值
}

```

4.5.2 预处理

预处理程序的作用是忽略空白字符(空格符、制表符、换行符)及注释。

```

/******************
* 功能: 预处理, 忽略空白字符及注释
*****************/
void preprocess()
{
    while(1)
    {
        if(ch==' ' || ch=='\t' || ch=='\r')
            skip_white_space();
        else if(ch=='/')
        {
            //向前多读一个字节看是否是注释开始符, 猜错了把多读的字符再放回去
            getch();
            if(ch=='*')
            {

```

```
    parse_comment();
}
else
{
    ungetc(ch, fin);           //把一个字符退回到输入流中
    ch = '/';
    break;
}
}
else
{
    break;
}
}
```

4.5.2.1 注释处理

```
/**************************************************************************
* 功能：解析注释
*************************************************************************/
void parse_comment()
{
    getch();
    do
    {
        do
        {
            if (ch == '\n' || ch == '*' || ch == CH_EOF)
                break;
            else
                getch();
        }while(1);
        if (ch == '\n')
        {
            line_num++;
            getch();
        }
        else if (ch == '*')
        {
            getch();
            if (ch == '/')
            {
                getch();
                return;
            }
        }
    }
}
```

```

        error("一直到文件尾未看到配对的注释结束符");
        return;
    }
}while(1);
}

```

4.5.2.2 空白字符处理

```

/***********************
* 功能：忽略空格、Tab 和回车
***********************/
void skip_white_space()
{
    while(ch==' ' || ch=='\t' || ch=='\r') //忽略空格、Tab 和回车
    {
        if(ch=='\r')
        {
            getch();
            if(ch != '\n')
                return;
            line_num++;
        }
        printf("%c",ch);
        getch();
    }
}

```

这里着重讲一下回车'\r'、换行'\n':

- 回车\r本义是光标重新回到本行开头，r的英文return，控制字符可以写成CR，即Carriage Return。
- 换行\n本义是光标往下一行（不一定到下一行行首），n的英文newline，控制字符可以写成LF，即Line Feed。

可能大家有个疑问，回车换行不就是一个意思，一个动作嘛，干吗还搞出两个字符，下面就讲一下这个问题的历史渊源。

在计算机还没有出现之前，有一种称为电传打字机（Teletype Model 33）的玩意儿，每秒钟可以打10个字符。但是它有一个问题，就是打完一行换行的时候，要用去0.2秒，正好可以打两个字符。要是在这0.2秒里面，又有新的字符传过来，那么这个字符将丢失。于是，研制人员想了个办法解决这个问题，就是在每行后面加两个表示结束的字符。一个称为“回车”，告诉打字机把打印头定位在左边界；另一个称为“换行”，告诉打字机把纸向下移一行。这就是“换行”和“回车”的来历，从它们的英语名字上也可以看出一二。后来，发明了计算机，这两个概念也就被搬到了计算机上。那时，存储器很贵，一些科学家认为在每行结尾加两个字符太浪费了，加一个就可以。于是，就出现了分歧。在不同的操作系统这两个字符表现不同，例如在Windows系统下，这两个字符就是表现的本义；在UNIX类系统，换行\n就表现为光标下一行并回到行首；在MAC上，\r就表现为回到本行开头并往下一行。至于

Enter键的定义是与操作系统有关的：

- \n\r: Windows系统行末结束符。
- \n: UNIX系统行末结束符。
- \r: MAC OS系统行末结束符。

一个直接后果是,UNIX/Mac系统下的文件在Windows里打开,所有文字会变成一行;而Windows里的文件在UNIX/Mac下打开,在每行的结尾可能会多出一个^M符号。不过大家也不必担心,很多文本/代码编辑器带有换行符转换功能,使用这个功能可以将文本文件中的换行符在不同格式间互换。由于SCC编译器目前只考虑在Windows系统中运行,从UNIX或Mac OS编写完成的源代码,如果有换行符问题,请转换后再进行编译。

4.5.3 解析标识符

```
/************************************************************************/
* 功能：判断c是否为字母(a~z,A~Z)或下划线(_)
* c:    字符值
/************************************************************************/
int is_nodigit(char c)
{
    return (c>='a' && c<='z') ||
           (c>='A' && c<='Z') ||
           c=='_';
}

/************************************************************************/
* 功能：判断c是否为数字
* c:    字符值
/************************************************************************/
int is_digit(char c)
{
    return c>='0' && c<='9';
}

/************************************************************************/
* 功能：解析标识符
/************************************************************************/
void parse_identifier()
{
    dynstring_reset(&tkstr);
    dynstring_chcat(&tkstr, ch);
    getch();
    while(is_nodigit(ch) || is_digit(ch))
    {
        dynstring_chcat(&tkstr, ch);
        getch();
    }
}
```

```

    }
    dynstring_chcat(&tkstr, '\0');
}

```

4.5.4 解析整数

```

/****************************************************************************
* 功能： 解析整型常量
****************************************************************************/
void parse_num()
{
    dynstring_reset(&tkstr);
    dynstring_reset(&sourcestr);
    do{
        dynstring_chcat(&tkstr, ch);
        dynstring_chcat(&sourcestr, ch);
        getch();
    }while(is_digit(ch));
    if(ch=='.')
    {
        do{
            dynstring_chcat(&tkstr, ch);
            dynstring_chcat(&sourcestr, ch);
            getch();
        }while(is_digit(ch));
    }
    dynstring_chcat(&tkstr, '\0');
    dynstring_chcat(&sourcestr, '\0');
    tkvalue=atoi(tkstr.data);
}

```

4.5.5 解析字符串

```

/****************************************************************************
* 功能： 解析字符常量、字符串常量
* sep:      字符常量界符标识为单引号 (')
            字符串常量界符标识为双引号 ("")
****************************************************************************/
void parse_string(char sep)
{
    char c;
    dynstring_reset(&tkstr);
    dynstring_reset(&sourcestr);
    dynstring_chcat(&sourcestr, sep);
    getch();
}

```

```
for(;;)
{
    if(ch==sep)
        break;
    else if(ch=='\\')
    {
        dynstring_chcat(&sourcestr,ch);
        getch();
        switch(ch)           //解析转义字符
        {
            case '0':
                c='\0';
                break;
            case 'a':
                c='\a';
                break;
            case 'b':
                c='\b';
                break;
            case 't':
                c='\t';
                break;
            case 'n':
                c='\n';
                break;
            case 'v':
                c='\v';
                break;
            case 'f':
                c='\f';
                break;
            case 'r':
                c='\r';
                break;
            case '\"':
                c='\"';
                break;
            case '\'':
                c='\'';
                break;
            case '\\':
                c='\\';
                break;
            default:
                c=ch;
                if(c>='!' && c<='~')
                    warning("非法转义字符: \\%c",c);
        }
    }
}
```

```

        else
            warning("非法转义字符: \\x%x\\", c);
            break;
    }
    dynstring_chcat(&tkstr, c);
    dynstring_chcat(&sourcestr, ch);
    getch();
}
else
{
    dynstring_chcat(&tkstr, ch);
    dynstring_chcat(&sourcestr, ch);
    getch();
}

}
dynstring_chcat(&tkstr, '\0');
dynstring_chcat(&sourcestr, sep);
dynstring_chcat(&sourcestr, '\0');
getch();
}

```

在讲 SC 语言扩展字符集时,讲过出现在注释、字符常量、字符串常量中的扩展字符集的附加成员是跟编译器的实现相关的,编译器有权决定支持哪些附加成员。SCC 编译器扩展字符集支持 GB2312 字符集。GB2312 的码表范围从 A1A0 至 FEFF,完全兼容 ASCII,不会跟 ASCII 区的编码产生冲突。这里对 GB2312 字符集做个简单介绍。GB2312 是中国国家标准简体中文字符集,全称《信息交换用汉字编码字符集-基本集》,由中国国家标准总局发布,1981 年 5 月 1 日实施。GB2312 标准共收录 6763 个汉字,其中一级汉字 3755 个,二级汉字 3008 个;同时收录了包括拉丁字母、希腊字母、日文平假名及片假名字母、俄语西里尔字母在内的 682 个字符。

你可能会说,SCC 编译器既然支持 GB2312 字符集,那还有哪些字符不支持。作者要讲一下另一个字符集 GBK 字符集,GBK 编码是在 GB2312 标准基础上的内码扩展规范,使用了双字节编码方案,其编码范围从 8140 至 FEFE(剔除 xx7F),共 23 940 个码位,共收录 21 003 个汉字,完全兼容 GB2312—80 标准。GBK 编码方案于 1995 年 10 月制定,1995 年 12 月正式发布。这里要注意 GBK 的编码范围内,低位字节的编码范围为 40~7F,这与 SC 语言基本字符集中的成员编码可能会产生冲突,所以低位字节在这个范围的字符出现在源码中时,会影响 SCC 编译器的正常运行。当然 SCC 编译器支持 GBK 字符集也不是什么困难的事,这里 SCC 编译器支持 GB2312 字符集,不完全支持 GBK 字符集,主要是为了让大家对语言的字符集,及编译器支持的字符集,两者的区别有更加深刻的理解。

4.5.6 词法分析流程图

本节词法分析程序的代码还是比较简单的,这里没有逐一解释,仅给出词法分析流程图(见图 4.8),一张图胜过千言万语。

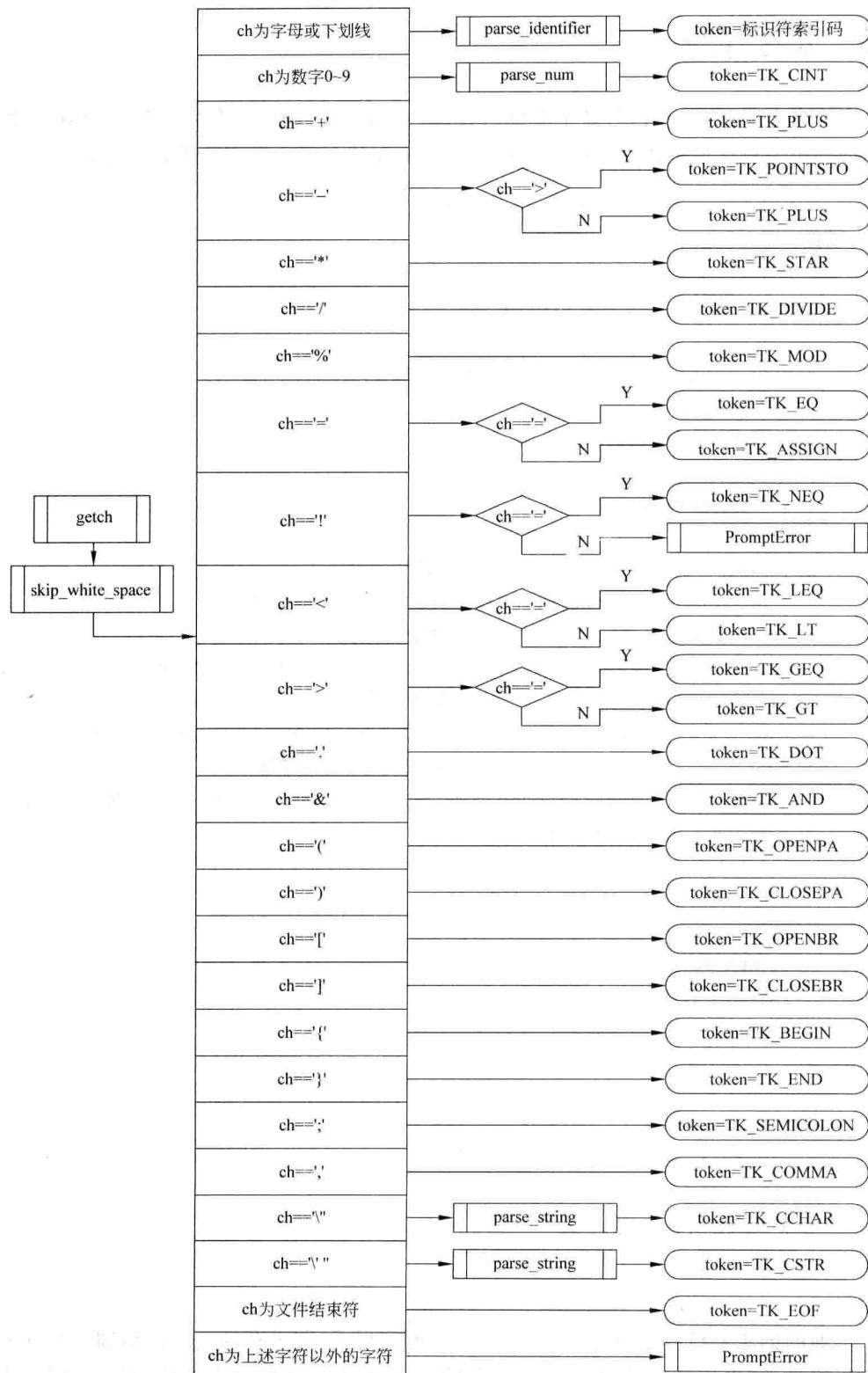


图 4.8 SCC 编译器词法分析流程图

4.6 词法着色

有的书上叫“语法着色”，是不准确的，这里特别强调“词法着色”的目的，是让读者了解在词法分析完成后就可以写代码着色程序了。下面就来看一看词法着色程序：

```
/**************************************************************************
 * 功能：词法着色
 **************************************************************************/
void color_token(int lex_state)
{
    HANDLE h=GetStdHandle(STD_OUTPUT_HANDLE);
    char * p;
    switch(lex_state)
    {
        case LEX_NORMAL:
        {
            if(token>=TK_IDENT)      /* 标识符为灰色 */
                SetConsoleTextAttribute(h,FOREGROUND_INTENSITY);
            else if(token>=KW_CHAR)   /* 关键字为绿色 */
                SetConsoleTextAttribute(h,FOREGROUND_GREEN|FOREGROUND_INTENSITY);
            else if(token>=TK_CINT)   /* 常量为黄色 */
                SetConsoleTextAttribute(h,FOREGROUND_RED|FOREGROUND_GREEN);
            else                      /* 运算符及分隔符为红色 */
                SetConsoleTextAttribute(h,FOREGROUND_RED|FOREGROUND_INTENSITY);
            p=get_tkstr(token);
            printf("%s",p);
            break;
        }
        case LEX_SEP:
        {
            printf("%c",ch);
            break;
        }
    }
    /* 词法状态枚举定义 */
enum e_LexState
{
    LEX_NORMAL,
    LEX_SEP
};
```

上面的词法着色程序只有不到 30 行代码就实现了，看来在词法分析基础上实现个词法着色程序是件“水到渠成”的事。词法着色的例子，在图 1.2 已经看到过，但这个例子还不够有代表性。在本章的最后，将用一个更具代表性的例子来介绍词法着色程序。

4.7 控制程序

最后介绍一下词法分析的控制程序,控制程序包括3个函数,main主函数、init初始化函数和cleanup扫尾清理函数。

```
*****  
* 功能: main 主函数  
*****  
int main(int argc, char **argv)  
{  
    fin=fopen(argv[1],"rb");  
    if(!fin)  
    {  
        printf("不能打开 SC 源文件!\n");  
        return 0;  
    }  
    init();  
  
    getch();  
    do  
    {  
        get_token();  
        color_token(LEX_NORMAL);  
    }while(token !=TK_EOF);  
    printf("\n 代码行数: %d 行\n",line_num);  
  
    cleanup();  
    fclose(fin);  
    printf("%s 词法分析成功!",argv[1]);  
    return 1;  
}  
  
*****  
* 功能: 初始化  
*****  
void init()  
{  
    line_num=1;  
    init_lex();  
}  
  
*****  
* 功能: 扫尾清理工作  
*****  
void cleanup()  
{  
    int i;
```

```

printf("\ntktable.count=%d\n", tktable.count);
for(i=TK_IDENT;i<tktable.count;i++)
{
    free(tktable.data[i]);
}
free(tktable.data);
}

```

这里要注意单词表 tktable 中必须从 TK_IDENT 开始释放,因为 TK_IDENT 以下的关键字、运算符、分隔符等单词没有放在堆中,而是放在静态存储区。

这里给出本章用到的全局变量。

```

TkWord * tk_hashtable[MAXKEY]; //单词哈希表
DynArray tktable; //单词表中放置标识符,包括变量名,函数名,结构定义名
DynString tkstr; //单词字符串
DynString sourcestr; //单词源码字符串
int tkvalue; //单词值(单词为整型常量)
char ch; //当前取到的源码字符
int token; //单词编码
int line_num; //行号

```

4.8 词法分析成果展示

下面用 color_token_demo.c 程序来测试一下本章写的词法分析程序和词法着色功能。如图 4.9 所示,可以看出,所有的关键字 struct、int、void、for、if、else、continue 都是绿色,所有的标识符都是灰色,整型常量和字符串常量都是黄色,运算符及分隔符都是红色。并且图中的最后一行赫然写着“color_token_demo.c 词法分析成功!”,说明这个程序词法上没有什么问题。至此,SCC 编译器第一阶段词法分析任务就算完成了。

```

E:\自己动手写编译器\代码例子\第四章>scc.exe color_token_demo.c
struct point
{
    int x;
    int y;
};
void main()
{
    int arr[10];
    int i;
    struct point pt;
    pt.x = 1024;
    pt.y = 768;
    for(i = 0; i < 10; i = i + 1)
    {
        arr[i] = 1;
        if (i == 6)
        {
            continue;
        }
        else
        {
            printf("arr[%d]=%d\n", i, arr[i]);
        }
    }
    printf("pt.x = %d, pt.y = %d\n", pt.x, pt.y);
}

EndOfFile
代码行数: 31行
color_token_demo.c 词法分析成功!

```

图 4.9 词法分析成果展示

第 5 章

SC 语言语法分析

横看成岭侧成峰，远近高低各不同。

——苏轼

语法分析是编译过程的第二个阶段，也是其核心的部分。语法分析的基本任务是在词法分析识别出单词符号的基础上，分析源程序的语法结构，即分析由这些单词如何组成各种语法成分，如“声明”、“函数”、“语句”、“表达式”等，并分析判断程序的语法结构是否符合语法规则。

语法分析的方法第 2 章已经讲过，这里再简单回顾一下，常用的语法分析方法有自顶向下分析和自底向上分析两大类。自顶向下分析又分为确定的和不确定的两种，这里采用的是确定的自顶向下分析方法，也就是不带回溯的那种。确定的自顶向下分析方法又有两种，一种是递归子程序法；另一种是预测分析方法，这里采用的是递归子程序法。图 5.1 清晰地表示了本章采用的递归子程序法在整个语法分析方法中所占的位置。

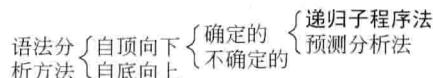


图 5.1 SCC 使用的语法分析方法

本章的主要学习任务是让大家理解由语言的语法定义，如何利用递归子程序法生成语法分析程序。在语法分析阶段可以实现函数及变量统计、生成语法分析图、语法缩进等功能，本章只选择实现比较实用的语法缩进程序。由于语法缩进程序穿插在语法分析过程中，与缩进有关代码用灰色底纹表示，单纯写语法分析程序这些代码是可以去掉的。语法缩进用到的两个全局变量，这里先简单介绍一下，syntax_state 控制语法状态决定是应该换行，还是空格分隔等，syntax_level 控制缩进级别。更详细的语法缩进程序讲述见 5.4 节。

每个语法成分的讲述至少包含 3 部分内容，即文法定义、代码实现和文法描述图。穿插着讲一下第 2 章讲的语法分析方法在本章如何应用。对于文法定义与语法分析函数能够严格对应的语法成分，一般不再解释。另外，这里要说一下第 2 章的语法定义全部采用的是中文，由于中文是不能用作函数名的，所以本章看到的文法定义作为函数注释全部采用了英文格式，在附录 A 中列出了 SC 语言文法定义中英文对照表供大家参考。下面就开始编写语法分析程序。

5.1 外部定义

5.1.1 翻译单元

翻译单元的代码如下：

```

/*****************
 * <translation_unit> ::= {<external_declaration>} <TK_EOF>
/*****************/
void translation_unit()
{
    while(token != TK_EOF)
    {
        external_declaration(SC_GLOBAL);
    }
}

```

文法描述图如图 5.2 所示。

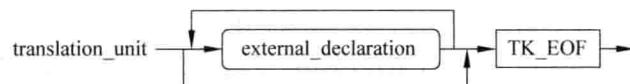


图 5.2 translation_unit 文法描述图

请大家观察一下，void translation_unit() 函数与 <translation_unit> 语法定义及文法描述图之间的关系。这里大致可以得出如下结论：

- <translation_unit> 语法定义的左侧非终结符作为函数名。
- <translation_unit> 的语法描述图基本可以看作程序流程图。
- 语法定义中 {} 表示 <external_declaration> 可以重复多次，对应 while 循环，循环终止的条件是遇到 {} 后面紧跟的 <TK_EOF> 终结符，注意 <TK_EOF> 在语法分析阶段作为终结符，在词法分析阶段则作为非终结符。

现在大家是不是逐步体会到，作者前面花了那么大的气力来对 SC 语言进行形式定义的良苦用心了。分析完 translation_unit 函数，大家可能在想，看来写语法分析程序也不是什么困难的事。先别高兴得太早，因为紧接着的 external_declaration 就是一“刺头”，对于第一次写语法分析程序的人来说，写起来可没那么简单。

5.1.2 外部声明

请大家思考一下已知 <external_declaration> ::= <function_definition> | <declaration> 这个文法定义，如何写出 external_declaration 语法分析函数。这个函数是所有语法分析函数中最难写的，难在什么地方？难在 <external_declaration> 文法定义不符合 LL(1) 文法定义，需要进行文法的等价变换，变换为 LL(1) 文法才能使用自顶向下的语法分析方法。估计很多想自己写编译器的朋友，都在这里望而却步了。在第 2 章中已经学过文法等价转换方法，下面就来试着转换一下。

将下面一组文法定义：

```

<external_declaration> ::= <function_definition> | <declaration>
<function_definition> ::= <typeSpecifier> <declarator> <funcbody>
<declaration> ::= <typeSpecifier> <TK_SEMICOLON>
    | <typeSpecifier> <init_declarator_list> <TK_SEMICOLON>
<init_declarator_list> ::= 

```

```

<init_declarator> {<TK_COMM><init_declarator>}
<init_declarator> ::==
    <declarator> {<TK_ASSIGN><initializer>}

```

等价转换为：

```

<external_declaration> ::==
    <typeSpecifier> (<TK_SEMICOLON>
        |<declarator><funcbody>
        |<declarator> [<TK_ASSIGN><initializer>]
        {<TK_COMM><declarator> [<TK_ASSIGN><initializer>]}<TK_SEMICOLON>)

```

现在可以利用改写后的`<external_declaration>`文法定义，写`external_declaration`语法分析函数了。

```

/*****************
* 功能：解析外部声明
* l： 存储类型，局部的还是全局的
*****************/
void external_declaration(int l)
{
    if(!typeSpecifier())
    {
        expect("<类型区分符>");
    }

    if(token==TK_SEMICOLON)
    {
        get_token();
        return;
    }
    while(1)      //逐个分析声明或函数定义
    {
        declarator();
        if(token==TK_BEGIN)
        {
            if(l==SC_LOCAL)
                error("不支持函数嵌套定义");
            funcbody();
            break;
        }
        else
        {

            if(token==TK_ASSIGN)
            {
                get_token();

```

```

        initializer();

    }

    if(token==TK_COMMA)
    {
        get_token();
    }
    else
    {
        syntax_state=SNTX_LF_HT;
        skip(TK_SEMICOLON);
        break;
    }
}
}

```

文法描述图如图 5.3 所示。

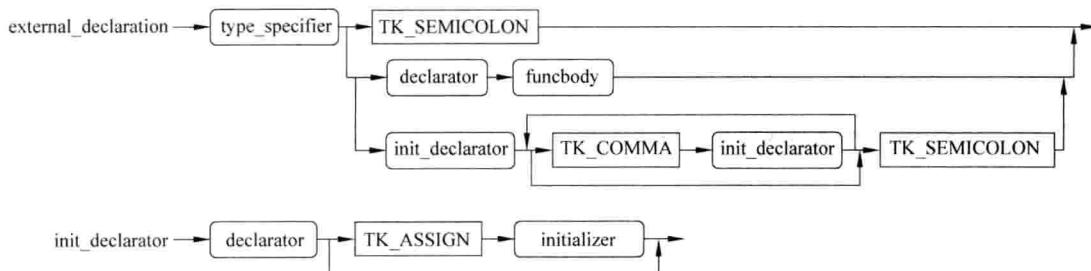


图 5.3 external declaration 文法描述图

下面通过一组例子来解释一下 external declaration 函数。

```
int g_a;                                /* 全局变量声明 */
int add(int x,int y);                  /* 函数声明 */
int add(int x,int y)                  /* 函数定义 */
{
    int z;                                /* 局部变量声明 */
    z=x+y;
    return x+y;
}
```

上面这组声明与函数定义,字体加粗部分的代码都是由函数 `external_declaration` 进行解析的。由于全局变量与局部变量的声明的文法定义没有任何区别,所以都在这个函数中完成,以避免重复编码,通过给这个函数传入参数以示区别。参数 1 为 `SC_GLOBAL` 表示处于函数外部解析状态,参数 1 为 `SC_LOCAL` 表示函数内部解析状态。

5.1.3 类型区分符

类型区分符的代码如下所示。

```
*****
* 功能：解析类型区分符
* 返回值：是否发现合法的类型区分符
*
* <typeSpecifier> ::= <KW_INT>
*   | <KW_CHAR>
*   | <KW_SHORT>
*   | <KW_VOID>
*   | <structSpecifier>
*****
int typeSpecifier()
{
    int type_found=0;
    switch(token)
    {
        case KW_CHAR:
            type_found=1;
            syntax_state=SNTX_SP;
            get_token();
            break;
        case KW_SHORT:
            type_found=1;
            syntax_state=SNTX_SP;
            get_token();
            break;
        case KW_VOID:
            type_found=1;
            syntax_state=SNTX_SP;
            get_token();
            break;
        case KW_INT:
            syntax_state=SNTX_SP;
            type_found=1;
            get_token();
            break;
        case KW_STRUCT:
            syntax_state=SNTX_SP;
            structSpecifier();
            type_found=1;
            break;
        default:
            break;
    }
    return type_found;
}
```

文法描述图如图 5.4 所示。

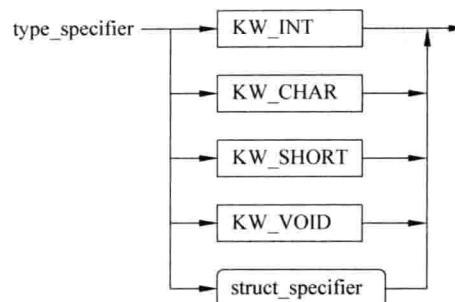


图 5.4 type_specifier 文法描述图

这里讲一下,当有多个产生式如何选择的问题,用到了第 2 章的求产生式的 SELECT 集的问题。下面求各个产生式的 SELECT 集:

```

SELECT(<type_specifier>→<KW_INT>) = FIRST(<KW_INT>) = {<KW_INT>}
SELECT(<type_specifier>→<KW_CHAR>) = FIRST(<KW_CHAR>) = {<KW_CHAR>}
SELECT(<type_specifier>→<KW_SHORT>) = FIRST(<KW_SHORT>) = {<KW_SHORT>}
SELECT(<type_specifier>→<KW_VOID>) = FIRST(<KW_VOID>) = {<KW_VOID>}
SELECT(<type_specifier>→<struct_specifier>) = FIRST(<struct_specifier>
= {<KW_STRUCT>}

```

看一下各个 SELECT 集相交情况:

```

SELECT(<type_specifier>→<KW_INT>) ∩ SELECT(<type_specifier>→<KW_CHAR>) = ∅
SELECT(<type_specifier>→<KW_SHORT>) ∩ SELECT(<type_specifier>→<KW_VOID>) = ∅

```

等,既然上面几个候选式的 SELECT 集两两交集都为空,分析起来当然就非常容易了。

5.1.4 结构区分符

结构区分符的代码如下所示。

```

/*****************
* <struct_specifier> ::= *
*   <KW_STRUCT><IDENTIFIER><TK_BEGIN><struct_declaraction_list><TK_END>
*   |<KW_STRUCT><IDENTIFIER>
/*****************/
void struct_specifier()
{
    int v;

    get_token();
    v = token;

    syntax_state = SNTX_DELAY; // 延迟到取出下一单词后确定输出格式
    get_token();
}

```

```

if(token==TK_BEGIN)           //适用于结构体定义
    syntax_state=SNTX_LF_HT;
else if(token==TK_CLOSEPA)    //适用于 sizeof(struct struct_name)
    syntax_state=SNTX_NUL;
else                         //适用于结构变量声明
    syntax_state=SNTX_SP;
syntax_indent();

if(v<TK_IDENT)               //关键字不能作为结构名称
    expect("结构体名");

if(token==TK_BEGIN)
{
    struct_declaration_list();
}
}

```

文法描述图如图 5.5 所示。

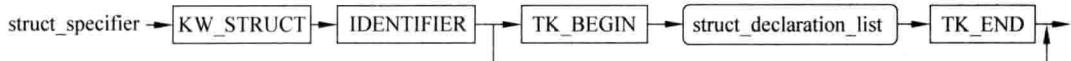


图 5.5 struct_specifier 文法描述图

5.1.4.1 结构声明符表

结构声明符表的代码如下所示。

```

/***********************
* <struct_declaration_list> ::= <struct_declarator> {<struct_declarator>}
************************/
void struct_declaration_list()
{
    int maxalign,offset;

    syntax_state=SNTX_LF_HT;      //第一个结构体成员与'{'不写在一行
    syntax_level++;              //结构体成员变量声明,缩进增加一级

    get_token();
    while(token !=TK_END)
    {
        struct_declarator(&maxalign,&offset);
    }
    skip(TK_END);

    syntax_state=SNTX_LF_HT;
}

```

文法描述图如图 5.6 所示。

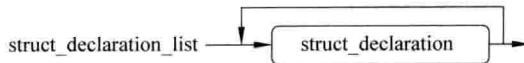


图 5.6 struct_declaration_list 文法描述图

5.1.4.2 结构声明

结构声明的代码如下所示。

```

/*
* <struct_declarator>::=
*     <type_specifier><declarator><TK_SEMICOLON> *
* <declarator_list>::=<declarator>{<TK_COMMMA><declarator>} *
*
* 等价转换后文法：：
* <struct_declarator>::=
*     <type_specifier><declarator>{<TK_COMMMA><declarator>}
*     <TK_SEMICOLON>
*/
void struct_declarator()
{
    typeSpecifier();
    while(1)
    {
        declarator();

        if(token==TK_SEMICOLON)
            break;
        skip(TK_COMMMA);
    }
    syntax_state=SNTX_LF_HT;
    skip(TK_SEMICOLON);
}

```

文法描述图如图 5.7 所示。

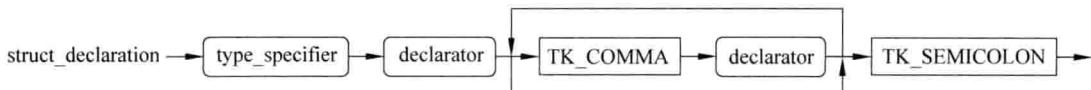


图 5.7 struct_declarator 文法描述图

上面的两个文法定义，其实不合并分别写两个语法分析函数也可以，不过写成一个更简洁，不啰唆。这里想说明这样一个问题，语言定义与语法分析时对文法要求的侧重点不一样。

语言定义时主要考虑：

(1) 每个非终符尽量代表一个有意义的语法成分；

(2) 文法描述要尽量简洁，看上去要通俗易懂。不一定完全考虑语法分析的方便性，就像前面讲的<external_declaration>改写后的文法虽然有利于语法分析，但是如果拿它作为语言定义就不合适了，因为许多原本有意义的语法成分大家看不到了，理解起来比较困难。

语法分析时主要考虑：

(1) 这个文法要有利于指导构造语法分析函数；

(2) 语法分析程序的简洁性及高效性，用一个函数就很容易表达的，不一定非得按语法定义产生好几个函数，几个函数之间又需要通过参数传递信息而降低了运行效率。

通过上面的分析，主要是想告诉大家已知语言的文法定义，在用递归子程序法构造语法分析程序时，注意这种方法的灵活运用，不要生搬硬套。

5.1.5 函数调用约定

函数调用约定的代码如下所示。

```
*****  
* <function_calling_convention> ::= <KW_CDECL> | <KW_STDCALL>  
* 用于函数声明上，用在数据声明上忽略掉  
*****  
void function_calling_convention(int * fc)  
{  
    * fc=KW_CDECL;  
    if(token==KW_CDECL || token==KW_STDCALL)  
    {  
        * fc=token;  
        syntax_state=SNTX_SP;  
        get_token();  
    }  
}
```

文法描述图如图 5.8 所示。

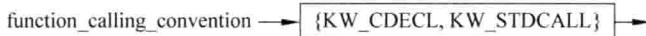


图 5.8 function_calling_convention 文法描述图

5.1.6 结构成员对齐

结构成员对齐的代码如下所示。

```
*****  
* <struct_member_alignment> ::= <KW_ALIGN> <TK_OPENPA> <TK_CINT> <TK_CLOSEPA>  
*****  
void struct_member_alignment()  
{
```

```

if(token==KW_ALIGN)
{
    get_token();
    skip(TK_OPENPA);
    if(token==TK_CINT)
    {
        get_token();
    }
    else
        expect("整数常量");
    skip(TK_CLOSEPA);
}
}

```

文法描述图如图 5.9 所示。



图 5.9 struct_member_alignment 文法描述图

5.1.7 声明符

声明符的代码如下所示。

```

/****************************************************************************
 * <declarator>::={<pointer>}[<function_calling_convention>]
 *   [<struct_member_alignment>]<direct_declarator>
 * <pointer>::=<TK_STAR>
 *
 * 等价转换后文法：
 * <declarator>::={<TK_STAR>}[<function_calling_convention>]
 *   [<struct_member_alignment>]<direct_declarator>
 *****/
void declarator()
{
    int fc;
    while(token==TK_STAR)
    {
        get_token();
    }
    function_calling_convention(&fc);
    struct_member_alignment();
    direct_declarator();
}

```

文法描述图如图 5.10 所示。

这里对文法定义灵活运用, 将两条文法定义合成了一条, 再定义一个 pointer 函数, 就完全

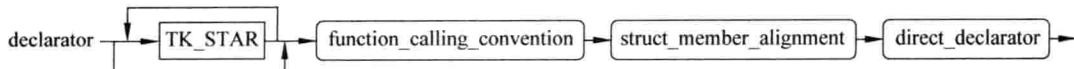


图 5.10 declarator 文法描述图

没有必要了，语法定义时单独定义<pointer>这个语法成分，目的就是明确表明这是个指针，如果直接用<TK_STAR>含义就比较模糊，因为它既可以代表指针，也可以代表乘号。

5.1.7.1 直接声明符

直接声明符的代码如下所示。

```

*****
* <direct_declarator> ::= <IDENTIFIER><direct_declarator_postfix>
*****
void direct_declarator()
{
    if(token>=TK_IDENT)
    {
        get_token();
    }
    else
    {
        expect("标识符");
    }
    direct_declarator_postfix();
}

```

文法描述图如图 5.11 所示。



图 5.11 direct_declarator 文法描述图

5.1.7.2 直接声明符后缀

直接声明符后缀的代码如下所示。

```

*****
* <direct_declarator_postfix> ::= {<TK_OPENBR><TK_CINT><TK_CLOSEBR>
*           |<TK_OPENBR><TK_CLOSEBR>
*           |<TK_OPENPA><parameter_type_list><TK_CLOSEPA> .
*           |<TK_OPENPA><TK_CLOSEPA> }
*****
void direct_declarator_postfix()
{
    int n;
}
```

```

if(token==TK_OPENPA)
{
    parameter_type_list();
}
else if(token==TK_OPENBR)
{
    get_token();
    if(token==TK_CINT)
    {
        get_token();
        n=tkvalue;
    }
    skip(TK_CLOSEBR);
    direct_declarator_postfix();
}
}

```

文法描述图如图 5.12 所示。

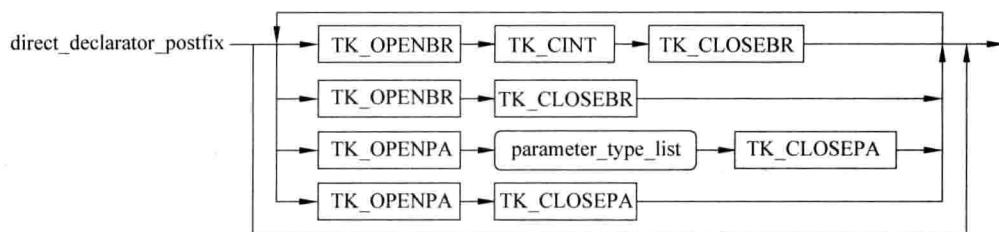


图 5.12 direct_declarator_postfix 文法描述图

5.1.7.3 形参类型表

形参类型表的代码如下所示。

```

/****************************************************************************
 * 功能：解析形参类型表
 * func_call: 函数调用约定
 *
 * <parameter_type_list> ::= <parameter_list>
 *     | <parameter_list><TK_COMMA><TK_ELLIPSIS>
 * <parameter_list> ::= <parameter_declaration>
 *     {<TK_COMMA><parameter_declaration>}
 * <parameter_declaration> ::= <typeSpecifier>{<declarator>}
 *
 * 等价转换后文法：
 * <parameter_type_list> ::= <typeSpecifier>{<declarator>}
 * {<TK_COMMA><typeSpecifier>{<declarator>}}<TK_COMMA><TK_ELLIPSIS>
 ****
 void parameter_type_list(int func_call)

```

```

{
    get_token();
    while(token !=TK_CLOSEPA)
    {
        if(!typeSpecifier())
        {
            error("无效类型标识符");
        }
        declarator();
        if(token==TK_CLOSEPA)
            break;
        skip(TK_COMMA);
        if(token==TK_ELLIPSIS)
        {
            func_call=KW_CDECL;
            get_token();
            break;
        }
    }
    syntax_state=SNTX_DELAY;
    skip(TK_CLOSEPA);
    if(token==TK_BEGIN)      //函数定义
        syntax_state=SNTX_LF_HT;
    else                      //函数声明
        syntax_state=SNTX_NUL;
    syntax_indent();
}

```

文法描述图如图 5.13 所示。

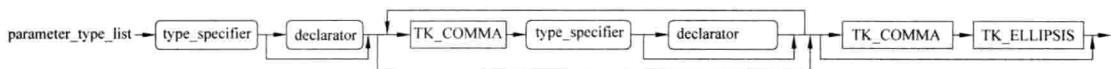


图 5.13 parameter_type_list 文法描述图

5.1.7.4 函数体

函数体的代码如下所示。

```

*****
* <funcbody> ::= <compound_statement>
*****
void funcbody()
{
    compound_statement();
}

```

文法描述图如图 5.14 所示。

在 funcbody() 函数中,只有一句代码就是调用 compound_statement(),为何不把调用 funcbody() 的地方直接改为调用 compound_statement()? 将 funcbody() 函数去掉岂不更简洁。如果思考到了这一步,证明前面 5.1.4.2 节的知识你已经掌握了。但这里遇到的是另一种情况,这种情况用陈澹然[清代]的经典名言“不谋万世者,不足谋一时;不谋全局者,不足谋一域。”描述再合适不过了。如果单看语法分析程序,前面的思考确实是对的,但是编写 SCC 编译器要经过词法分析、语法分析、语义分析 3 个阶段,目前刚到第二个阶段,我们希望 SCC 编译器到语义分析阶段能够完全采用语法分析时定义的架构,尽可能少做修改。下面说明为什么把 funcbody() 与 compound_statement() 分开定义。举个例子来说明:

```
void main()
{
    //函数体开始,也可以说是复合语句开始
    int arr[10]
    int i;
    for(i=0;i<10;i++)
    {
        //复合语句开始
        arr[i]=i * i;
        printf("arr[%d]=%d\n",i,arr[i]);
    }
    //复合语句结束
}
//函数体结束,也可以说是复合语句结束
```

从这个例子可以看出<复合语句>的概念比<函数体>概念要大,<函数体>属于一类比较特殊的<复合语句>,在语义分析阶段与普通<复合语句>相比,除了干普通<复合语句>的活,还有些特殊任务要执行,如生成函数的开始、结尾代码。具体都做了哪些特殊任务,在 9.1.3 节函数体语义分析时大家将看到。

5.1.8 初值符

初值符的代码如下所示。

```
*****
* <initializer> ::= <assignment_expression>
*****
void initializer()
{
    assignment_expression();
}
```

文法描述图如图 5.15 所示。

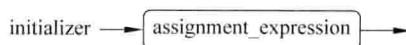


图 5.14 funcbody 文法描述图

图 5.15 initializer 文法描述图

又是一个只有一行代码的函数,为什么要单独写成一个函数,情况跟<函数体>基本一样,这里不再赘述。

5.2 语句

与语句相应的代码如下所示。

```
*****  
* <statement> ::= <compound_statement>  
*   | <if_statement>  
*   | <return_statement>  
*   | <break_statement>  
*   | <continue_statement>  
*   | <for_statement>  
*   | <expression_statement>  
*****  
void statement(int *bsym, int *csym)  
{  
    switch(token)  
    {  
        case TK_BEGIN:  
            compound_statement(bsym, csym);  
            break;  
        case KW_IF:  
            if_statement(bsym, csym);  
            break;  
        case KW_RETURN:  
            return_statement();  
            break;  
        case KW_BREAK:  
            break_statement(bsym);  
            break;  
        case KW_CONTINUE:  
            continue_statement(csym);  
            break;  
        case KW_FOR:  
            for_statement(bsym, csym);  
            break;  
        default:  
            expression_statement();  
            break;  
    }  
}
```

文法描述图如图 5.16 所示。

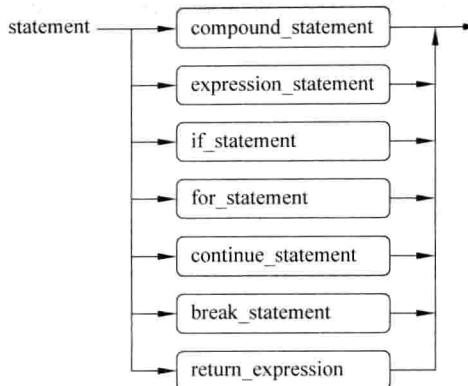


图 5.16 statement 文法描述图

5.2.1 复合语句

复合语句的代码如下所示。

```

/***********************
* <compound_statement> ::= <TK_BEGIN> {<declaration>} {<statement>} <TK_END>
************************/
void compound_statement()
{
    syntax_state=SNTX_LF_HT;
    syntax_level++;           //复合语句,缩进增加一级

    get_token();
    while(is_type_specifier(token))
    {
        external_declaration(SC_LOCAL);
    }
    while(token != TK_END)
    {
        statement();
    }

    syntax_state=SNTX_LF_HT;
    get_token();
}
  
```

文法描述图如图 5.17 所示。

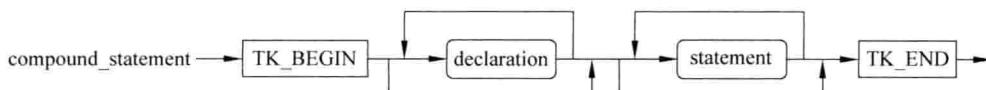


图 5.17 compound_statement 文法描述图

上面用到的 is_type_specifier 函数有必要在这里介绍一下。从图 5.17 可以看出，复合语句由声明和语句组成，声明在前，语句在后。声明解析函数 external_declaration 和语句解析函数 statement 已经在前面介绍过，它们的职责很明确，在应该出现声明的地方解析声明，在应该出现语句的地方解析语句。这里需要一个根据取到的当前单词判断是否是声明的函数，is_type_specifier 函数就是做这个工作的，它的代码如下所示。

```
*****  
* 功能：判断是否为类型区分符  
* v: 单词编号  
*****  
int is_typeSpecifier(int v)  
{  
    switch(v)  
    {  
        case KW_CHAR:  
        case KW_SHORT:  
        case KW_INT:  
        case KW_VOID:  
        case KW_STRUCT:  
            return 1;  
        default:  
            break;  
    }  
    return 0;  
}
```

5.2.2 表达式语句

与表达式语句相关的代码如下所示。

```
*****  
* <expression_statement> ::= <TK_SEMICOLON> | <expression><TK_SEMICOLON>  
*****  
void expression_statement()  
{  
    if(token != TK_SEMICOLON)  
    {  
        expression();  
    }  
    syntax_state=SNTX_LF_HT;  
    skip(TK_SEMICOLON);  
}
```

文法描述图如图 5.18 所示。

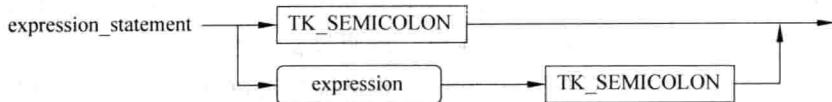


图 5.18 expression_statement 文法描述图

5.2.3 选择语句

选择语句的代码如下所示。

```

*****
* <if_statement> ::= <KW_IF><TK_OPENPA><expression>
*   <TK_CLOSEPA><statement> [<KW_ELSE><statement>]
*****
void if_statement()
{
    syntax_state=SNTX_SP;
    get_token();
    skip(TK_OPENPA);
    expression();
    syntax_state=SNTX_LF_HT;
    skip(TK_CLOSEPA);
    statement();
    if(token==KW_ELSE)
    {
        syntax_state=SNTX_LF_HT;
        get_token();
        statement();
    }
}

```

文法描述图如图 5.19 所示。

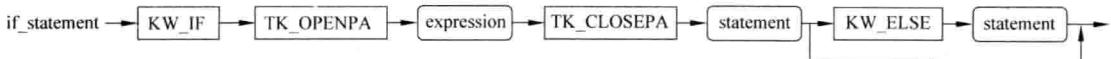


图 5.19 if_statement 文法描述图

5.2.4 循环语句

循环语句的代码如下所示。

```

*****
* <for_statement> ::= <KW_FOR><TK_OPENPA><expression_statement>
*   <expression_statement><expression><TK_CLOSEPA><statement>
*****
void for_statement()

```

```

{
    get_token();
    skip(TK_OPENPA);
    if(token !=TK_SEMICOLON)
    {
        expression();
    }
    skip(TK_SEMICOLON);
    if(token !=TK_SEMICOLON)
    {
        expression();
    }
    skip(TK_SEMICOLON);
    if(token !=TK_CLOSEPA)
    {
        expression();
    }
    syntax_state=SNTX_LF_HT;
    skip(TK_CLOSEPA);
    statement();
}

```

文法描述图如图 5.20 所示。



图 5.20 for_statement 文法描述图

5.2.5 跳转语句

跳转语句包括 continue 语句、break 语句和 return 语句。

5.2.5.1 continue 语句

continue 语句的代码如下所示。

```

/***********************
*<continue_statement> ::= <KW_CONTINUE><TK_SEMICOLON>
************************/
void continue_statement()
{
    get_token();
    syntax_state=SNTX_LF_HT;
    skip(TK_SEMICOLON);
}

```

文法描述图如图 5.21 所示。

这里大家要重点理解一下第 2 章讲到的“语法能够描述程序设计语言的大部分语法但



图 5.21 continue_statement 文法描述图

不是全部”这句话,这里的 continue 语句严格来说必须出现在 for 循环内,这种限制无法用上下文无关语言来描述。上面产生的 continue_statement() 没有对 continue 语句出现的位置进行限制与判断,因此就会造成图 5.22 所示的情况。

```

C:\WINDOWS\system32\cmd.exe
E:\自己动手写编译器\代码样例\第五章>scc.exe syntax_continue_break_test.c
void main()
{
    int arr[10];
    int i;
    for(i = 0; i < 10; i = i + 1)
    {
        if (i == 6)
            continue;
        if (i == 8)
            break;
        arr[i]=i;
    }
    continue;
    break;
}
End Of File
syntax_continue_break_test.c 语法分析通过!

```

图 5.22 continue 语句应用举例

可以看出 continue 语句无论出现在 for 循环内,还是 for 循环外都编译通过了。按照 SC 语言的语法定义,确实上面的程序语法上是正确的。但是语言除了语法定义,还有语义的要求,continue 语句在语义上要求必须出现在 for 循环内。因此在语义分析阶段将对这种情况加以限制。请大家再加深一下对这句话的理解:“语法分析器接受的语言是程序设计语言的超集。必须通过语义分析来剔除一些符合文法、但语义不合法的程序。”

5.2.5.2 break 语句

break 语句的代码如下所示。

```

/*****************
* <break_statement> ::= <KW_BREAK><TK_SEMICOLON>
*****************/
void break_statement()
{
    get_token();
    syntax_state=SNTX_LF_HT;
    skip(TK_SEMICOLON);
}

```

文法描述图如图 5.23 所示。



图 5.23 break_statement 文法描述图

这里 break 语句想说的跟 continue 语句一模一样,例子也一并举过了,不再赘述。

5.2.5.3 return 语句

return 语句的代码如下所示。

```
*****  
* <return_statement> ::= <KW_RETURN><TK_SEMICOLON>  
*           | <KW_RETURN><expression><TK_SEMICOLON>  
*****  
void return_statement()  
{  
    syntax_state=SNTX_DELAY;  
    get_token();  
    if(token==TK_SEMICOLON)      //适用于 return;  
        syntax_state=SNTX_NUL;  
    else                         //适用于 return<expression>;  
        syntax_state=SNTX_SP;  
    syntax_indent();  
  
    if(token !=TK_SEMICOLON)  
    {  
        expression();  
    }  
    syntax_state=SNTX_LF_HT;  
    skip(TK_SEMICOLON);  
}
```

文法描述图如图 5.24 所示。

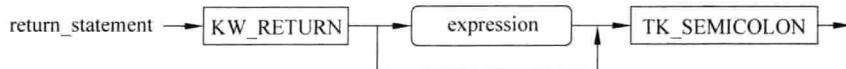


图 5.24 return_statement 文法描述图

5.3 表达式

表达式的相关代码如下所示。

```
*****  
* <expression> ::= <assignment_expression> {<TK_COMMA><assignment_expression>}  
*****  
void expression()  
{  
    while(1)  
    {
```

```

        assignment_expression();
        if(token != TK_COMMA)
            break;
        get_token();
    }
}

```

文法描述图如图 5.25 所示。

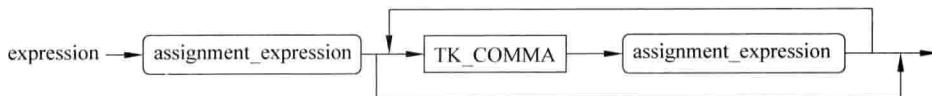


图 5.25 expression 文法描述图

5.3.1 赋值表达式

赋值表达式的代码如下所示。

```

*****
* <assignment_expression> ::= <equality_expression>
*     | <unary_expression><TK_ASSIGN><assignment_expression>:
*
* 非等价变换后文法：
* <assignment_expression> ::= <equality_expression>
*     {<TK_ASSIGN><assignment_expression>}:
*****
void assignment_expression()
{
    equality_expression();
    if(token==TK_ASSIGN)
    {
        get_token();
        assignment_expression();
    }
}

```

文法描述图如图 5.26 所示。

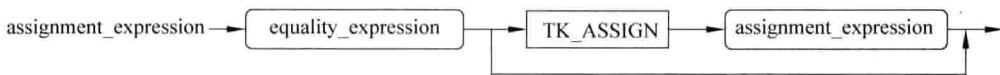


图 5.26 assignment_expression 文法描述图

这个函数有必要重点讲一下，因为以前只听说对语法进行等价变换，而等价变换算是换汤没换药，语言还是原来的语言。而这里竟然对<assignment_expression>文法定义进行非等价变换，要知道非等价变换后语法分析程序可以接受的语言可是发生了变换，作者莫非是吃了豹子胆。先不要着急评论，看一下上面的文法非等价变换后可表示的语言范围发生

了怎样的变化。把第二个候选式的 $\langle\text{unary_expression}\rangle$ 替换为 $\langle\text{equality_expression}\rangle$ ，相当于扩大了可表示语言的范围。下面来看一下变换前后可接受语言的例子：

变换前

例 1	$5=a$	$/*$ 语法合理,语义不合理 $*/$
例 2	$-8=9$	$/*$ 语法合理,语义不合理 $*/$
例 3	$a=6$	$/*$ 语法合理,语义合理 $*/$
例 4	$\text{arr}[1]=1$	$/*$ 语法合理,语义合理 $*/$
例 5	$\text{sizeof}(\text{int})=8$	$/*$ 语法合理,语义不合理 $*/$
例 6	$\text{add}(5,6)=7$	$/*$ 语法合理,语义不合理 $*/$
例 7	$*a=8$	$/*$ 语法合理,语义合理 $*/$
例 8	$\&a=9$	$/*$ 语法合理,语义不合理 $*/$
例 9	$a+b=6$	$/*$ 语法不合理,语义不合理 $*/$
例 10	$a*b=c$	$/*$ 语法不合理,语义不合理 $*/$
例 11	$a>b=8$	$/*$ 语法不合理,语义不合理 $*/$

可以看到变换前能接受的语言中,语义上不合理就有好多,那么变换后不就是又多了几种语法合理,语义不合理的情况,应该也无妨。变换后例 9、例 10 也变成了语法合理,语义不合理,跟例 1、例 2、例 5、例 6、例 8 可以平起平坐了。再分析一下 $\langle\text{赋值表达式}\rangle$ 等号左侧语义是否合理的本质,就是左侧必须为左值。反正到语义分析时也要通过左值过滤判断那些语法合理,语义不合理的情况,把新增加的情况捎带判断一下不就可以了。

5.3.2 相等类表达式

相等类表达式的代码如下所示。

```
/*
* <equality_expression> ::= <relational_expression>
*   {<TK_EQ><relational_expression>
*   |<TK_NEQ><relational_expression>}
*/
void equality_expression()
{
    relational_expression();
    while(token==TK_EQ || token==TK_NEQ)
    {
        get_token();
        relational_expression();
    }
}
```

文法描述图如图 5.27 所示。

5.3.3 关系表达式

关系表达式的代码如下所示。

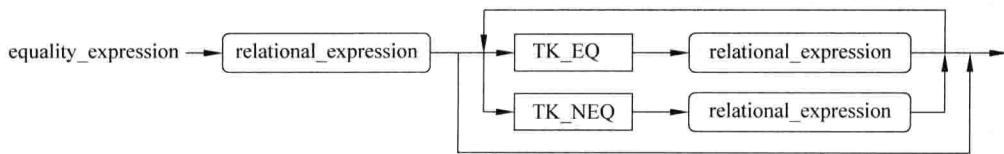


图 5.27 equality_expression 文法描述图

```

/*
* <relational_expression> ::= <additive_expression> {
*     <TK_LT><additive_expression>
*     |<TK_GT><additive_expression>
*     |<TK_LEQ><additive_expression>
*     |<TK_GEQ><additive_expression>
* }
void relational_expression()
{
    additive_expression();
    while((token==TK_LT || token==TK_LEQ) ||
          token==TK_GT || token==TK_GEQ)
    {
        get_token();
        additive_expression();
    }
}
  
```

文法描述图如图 5.28 所示。

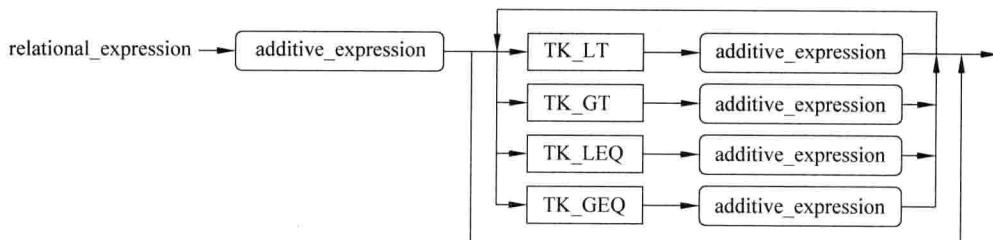


图 5.28 relational_expression 文法描述图

5.3.4 加减类表达式

加减类表达式的代码如下所示。

```

/*
* <additive_expression> ::= <multiplicative_expression>
*     {<TK_PLUS><multiplicative_expression>
*      <TK_MINUS><multiplicative_expression>}
* }
void additive_expression()
{
}
  
```

```

multiplicative_expression();
while(token==TK_PLUS || token==TK_MINUS)
{
    get_token();
    multiplicative_expression();
}
}

```

文法描述图如图 5.29 所示。

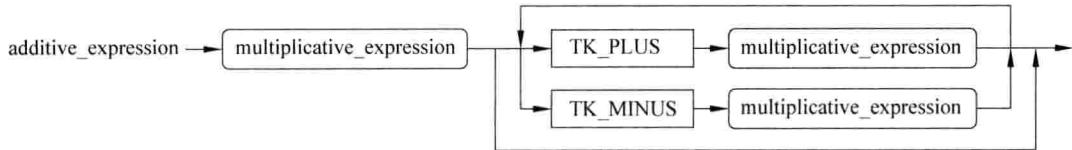


图 5.29 additive_expression 文法描述图

5.3.5 乘除类表达式

乘除类表达式的代码如下所示。

```

/****************************************************************************
 * <multiplicative_expression> ::= <unary_expression>
 *      {<TK_STAR><unary_expression>
 *       |<TK_DIVIDE><unary_expression>
 *       |<TK_MOD><unary_expression>}
 *****/
void multiplicative_expression()
{
    unary_expression();
    while(token==TK_STAR || token==TK_DIVIDE || token==TK_MOD)
    {
        get_token();
        unary_expression();
    }
}

```

文法描述图如图 5.30 所示。

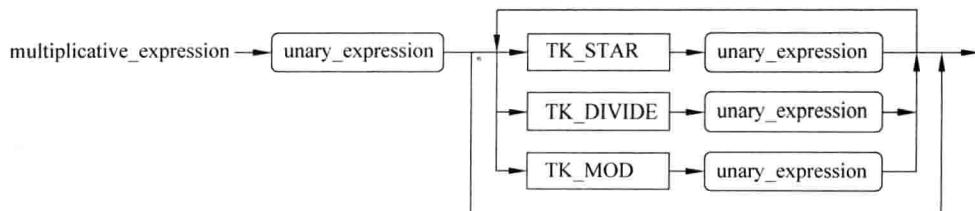


图 5.30 multiplicative_expression 文法描述图

5.3.6 一元表达式

一元表达式的代码如下所示。

```
*****<unary_expression> ::= <postfix_expression>
*      | <TK_AND><unary_expression>
*      | <TK_STAR><unary_expression>
*      | <TK_PLUS><unary_expression>
*      | <TK_MINUS><unary_expression>
*      | <sizeof_expression>
*****
void unary_expression()
{
    switch(token)
    {
        case TK_AND:
            get_token();
            unary_expression();
            break;
        case TK_STAR:
            get_token();
            unary_expression();
            break;
        case TK_PLUS:
            get_token();
            unary_expression();
            break;
        case TK_MINUS:
            get_token();
            unary_expression();
            break;
        case KW_SIZEOF:
            sizeof_expression();
            break;
        default:
            postfix_expression();
            break;
    }
}
```

文法描述图如图 5.31 所示。

sizeof 表达式的代码如下所示。

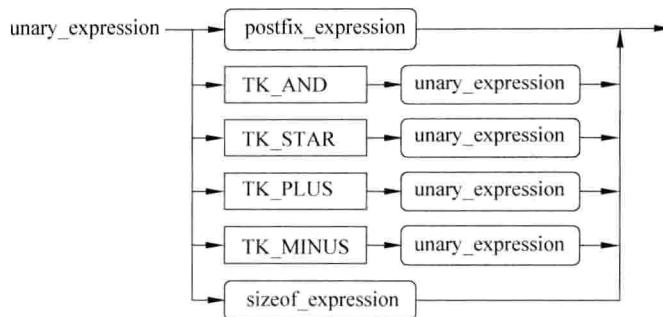


图 5.31 unary_expression 文法描述图

```

/******************
* <sizeof_expression> ::= 
*   <KW_SIZEOF><TK_OPENPA><type_specifier><TK_CLOSEPA>
/******************/

void sizeof_expression()
{
    get_token();
    skip(TK_OPENPA);
    type_specifier();
    skip(TK_CLOSEPA);
}
  
```

文法描述图如图 5.32 所示。

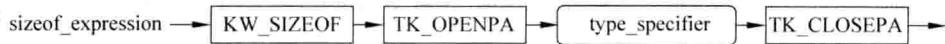


图 5.32 sizeof_expression 文法描述图

5.3.7 后缀表达式

后缀表达式的代码如下所示。

```

/******************
* <postfix_expression> ::= <primary_expression>
*   | <TK_OPENBR><expression><TK_CLOSEBR>
*   | <TK_OPENPA><TK_CLOSEPA>
*   | <TK_OPENPA><argument_expression_list><TK_CLOSEPA>
*   | <TK_DOT><IDENTIFIER>
*   | <TK_POINTSTO><IDENTIFIER>
/******************/

void postfix_expression()
{
    primary_expression();
    while(1)
  
```

```

    {
        if(token==TK_DOT || token==TK_POINTSTO)
        {
            get_token();
            token |= SC_MEMBER;
            get_token();
        }
        else if(token==TK_OPENBR)
        {
            get_token();
            expression();
            skip(TK_CLOSEBR);
        }
        else if(token==TK_OPENPA)
        {
            argument_expression_list();
        }
        else
            break;
    }
}

```

文法描述图如图 5.33 所示。

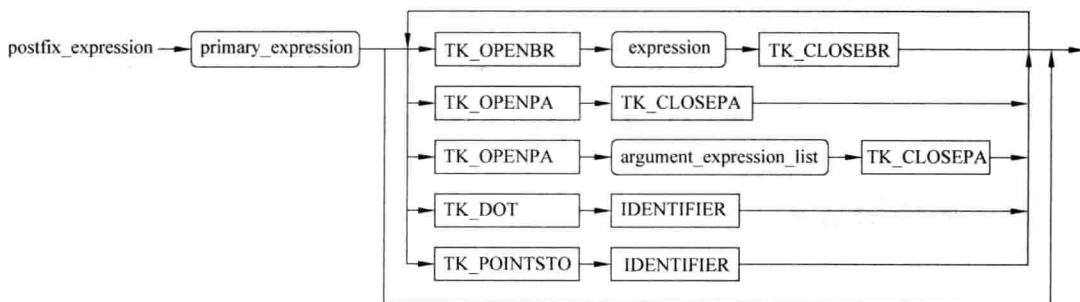


图 5.33 postfix_expression 文法描述图

5.3.8 初值表达式

初值表达式的代码如下所示。

```

/*****************
* <primary_expression> ::= <IDENTIFIER>
*   | <TK_CINT>
*   | <TK_CSTR>
*   | <TK_CCHAR>
*   | <TK_OPENPA><expression><TK_CLOSEPA>
*****************/

```

```

void primary_expression()
{
    int t;
    switch(token)
    {
        case TK_CINT:
        case TK_CCHAR:
            get_token();
            break;
        case TK_CSTR:
            get_token();
            break;
        case TK_OPENPA:
            get_token();
            expression();
            skip(TK_CLOSEPA);
            break;
        default:
            t=token;
            get_token();
            if(t<TK_IDENT)
                expect("标识符或常量");
            break;
    }
}

```

文法描述图如图 5.34 所示。

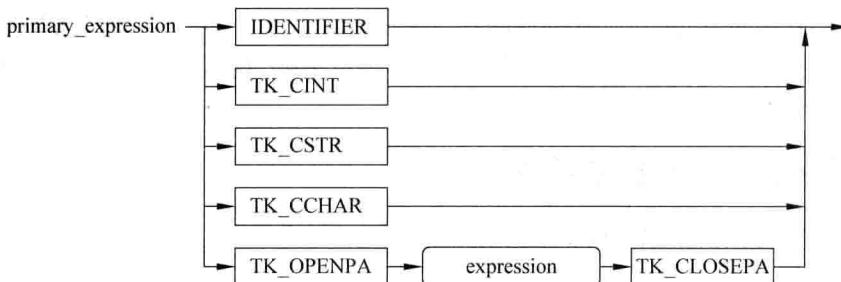


图 5.34 primary_expression 文法描述图

实参表达式表的代码如下所示。

```
/**************************************************************************  
* <argument_expression_list>::=<assignment_expression>  
*     {<TK_COMMA><assignment_expression>}  
*****
```

```

{
    get_token();
    if(token != TK_CLOSEPA)
    {
        for(;;)
        {
            assignment_expression();
            if(token==TK_CLOSEPA)
                break;
            skip(TK_COMMA);
        }
    }
    skip(TK_CLOSEPA);
}

```

文法描述图如图 5.35 所示。

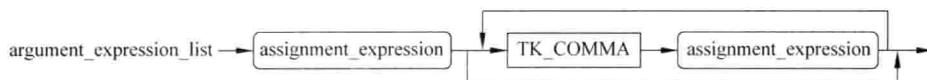


图 5.35 argument_expression_list 文法描述图

5.4 语法缩进

语法缩进就是对代码进行排版。代码的排版虽然不影响代码的编译和程序的运行,但是对于开发人员来说,它同样很重要。当人们看到排版整齐质量又高的代码的时候,会像欣赏艺术品一样去欣赏它;而如果代码排版混乱,哪怕思想再好的代码也会让你看了心烦。下面来讲一下语法缩进程序,由于本章的侧重点不在语法缩进上,这个功能只能算是语法分析程序的一个副产品,所以看上去比较简陋,算是抛砖引玉吧,相信读者可以实现功能更加齐全的代码排版程序。

5.4.1 用到的全局变量及枚举

语法缩进功能主要由以下两个全局变量来控制:

```
*****
int syntax_state;          //语法状态
int syntax_level;          //缩进级别
*****
```

语法状态 syntax_state 全局变量可取以下枚举值:

```
/* 语法状态枚举值 */
enum e_SynTaxState
{
    SNTX_NUL,           //空状态,没有语法缩进动作
```

```

SNTX_SP,           //空格
SNTX_LF_HT,       //换行并缩进,每一个声明、函数定义、语句结束都要置为此状态
SNTX_DELAY        //延迟到取出下一单词后确定输出格式
};

}

```

5.4.2 语法缩进程序

穿插在语法分析过程中的语法缩进的代码,在讲语法分析程序时已经一并给出了,下面来讲一下语法缩进程序。为了不因语法缩进这个辅助功能而对语法分析造成过多干扰,语法缩进功能主要放在 get_token 函数中进行调用,这样我们就能很方便地边取单词,边进行缩进输出。增加语法缩进功能的 get_token 函数如下:

```

*****+
* 功能: 取单词
*****/
void get_token()
{
    :
    syntax_indent();
}

```

语法缩进函数通过 syntax_state 判断语法状态,进行相应的格式输出:

```

*****+
* 功能: 语法缩进
*****/
void syntax_indent()
{
    switch(syntax_state)
    {
        case SNTX_NUL:
            color_token(LEX_NORMAL);
            break;
        case SNTX_SP:
            printf(" ");
            color_token(LEX_NORMAL);
            break;
        case SNTX_LF_HT:
        {
            if(token==TK_END)      //遇到'}',缩进减少一级
                syntax_level--;
            printf("\n");
            print_tab(syntax_level);
        }
        color_token(LEX_NORMAL);
    }
}

```

```

        break;
    case SNTX_DELAY:
        break;
}
syntax_state=SNTX_NUL;
}

```

使用 Tab 键控制缩进级别,下面是 Tab 缩进函数:

```

/***********************
 * 功能: 缩进 n 个 Tab 键
 * n: 缩进个数
*********************/
void print_Tab(int n)
{
    int i=0;
    for(;i<n;i++)
        printf("\t");
}

```

5.5 总控程序

总控程序包括 3 个函数,即 main 主函数、init 初始化函数和 cleanup 扫尾清理函数,init 及 cleanup 函数与词法分析阶段完全一样,这里不再列出。

```

/***********************
 * 功能: main 主函数
*********************/
int main(int argc,char **argv)
{
    fin=fopen(argv[1],"rb");
    if(!fin)
    {
        printf("不能打开 SC 源文件!\n");
        return 0;
    }

    init();
    getch();
    get_token();
    translation_unit();
    cleanup();
    fclose(fin);
    printf("\n%s 语法分析通过!\n",argv[1]);
    return 1;
}

```

自顶向下分析,就像一棵倒挂的树,分析的入口是语法定义的开始符号,即 translation_unit。

5.6 成果展示

下面用 syntax_indent_demo.c 程序来测试一下本章写的语法分析程序和语法缩进功能。如图 5.36 所示,可以看出,语法缩进前写的乱七八糟的代码,经语法缩进处理后井然有序,看上去舒服多了。并且图中的最后一行赫然写着“syntax_indent_demo.c 语法分析通过!”,说明这个程序语法上没有什么问题。至此,SCC 编译器第二阶段的任务——语法分析已经完成。

```

E:\自己动手写编译器\代码样例\第5章>type syntax_indent_demo.c
/****************************************************************************
 * syntax_indent_demo.c
 ****
 struct point{int x; int y;};
void main()
{
    int arr[10]; int i;    struct point pt;
    pt.x =1024;pt.y=768;
    for(i= 0; i < 10; i = i + 1)  {arr[i]=i;
    if(i == 6){continue;
    }
    else printf("arr[%d]=%d\n",i,arr[i]);}
    printf("pt.x = %d, pt.y = %d\n",pt.x,pt.y);
}

E:\自己动手写编译器\代码样例\第5章>scc.exe syntax_indent_demo.c
struct point
{
    int x;
    int y;
};
void main()
{
    int arr[10];
    int i;
    struct point pt;
    pt.x =1024;
    pt.y=768;
    for(i = 0; i < 10; i = i + 1)
    {
        arr[i]=i;
        if (i == 6)
        {
            continue;
        }
        else
        {
            printf("arr[%d]=%d\n",i,arr[i]);
        }
    }
    printf("pt.x = %d, pt.y = %d\n",pt.x,pt.y);
}
End_Of_File
syntax_indent_demo.c 语法分析通过!

```

图 5.36 语法分析成果展示

第6章

符 号 表

彼节者有间，而刀刃者无厚；以无厚入有间，恢恢乎其于游刃必有余地矣。

——庄子

本章开始进行语义分析，由于语义分析内容很多，全部放在一章会比较零乱，所以语义分析分4章来讲，第6~8章算是语义分析前的准备工作，第9章正式进行语义分析。本章介绍SCC编译器用到的符号表，关于符号表在整个SCC编译器中占有什么地位，看一下图6.1就非常清楚了，从图中可以看出符号表可是SCC编译器的一个核心结构，它与编译的各个阶段都息息相关。

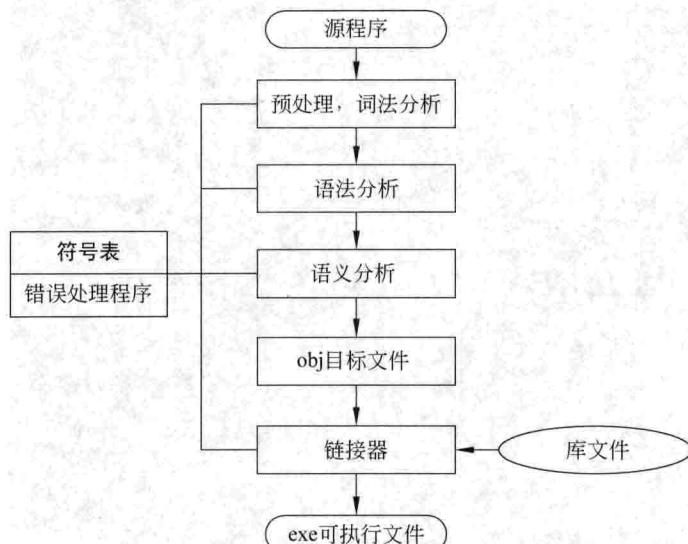


图 6.1 符号表在 SCC 编译器中的地位

在正式介绍符号表之前，先来看一段程序，以便大家对符号表有个初步认识。

```
/**************************************************************************  
* homonym.c 源文件——同名异义符号  
**************************************************************************/  
#include<stdio.h>  
  
int x;  
  
struct x
```

```

{
    int x;
    int y;
};

int add(int x,int y)
{
    return x+y;
}

int main()
{
    int x=1;
    int sum;
    {
        int x=2;
        {
            int x=3;
            sum=add(x,1);
        }
    }
    printf("sum=%d\n",sum);
    return 1;
}

```

上面这段程序能否编译通过？是不是会有符号重复定义的错误提示？但看到图 6.2，你的疑虑就会完全打消，不但没有错误提示，连个警告都没有。说明以上程序在语法上、语义上都是没有任何问题的。并且上面的程序 SC 语言也是支持的，那么请大家思考一个问题，同一个标识符 x，如何表示上面这些同名不同义的符号，在符号表中如何存储、如何区分呢？

6.1 符号表简介

在编译程序中符号表用来存放程序中出现的有关标识符的属性信息，这些信息集中反映了标识符的语义特征属性。编译过程中不断累积和更新表中的信息，编译的各个阶段也按各自的需要从表中获取不同的属性信息。符号表的功能归结为以下几个主要方面。

6.1.1 收集符号属性

在分析语言程序中标识符说明部分时，编译程序根据说明信息收集有关标识符的属性，并在符号表中建立符号的相应属性信息。例如，编译程序分析下述几个说明语句：

```

int A;
char B[8];
struct C

```



图 6.2 同名符号程序举例

```

{
    int m;
    int n;
}

int Add(int x,int y)
{
    return x+y;
}

```

在符号表中收集到符号 A 的属性是一个整型变量;符号 B 的属性是具有 8 个字符元素的一维数组;符号 C 的属性是一个结构定义,它有两个成员变量,第一个成员变量 m 为 int 类型,第二个成员变量 n 也为 int 类型;符号 Add 的属性是一个函数,它的返回值为 int 类型,它有两个参数,第一个参数 x 为 int 类型,第二个参数也为 int 类型。

6.1.2 语义的合法性检查

在语义分析中,符号表所登记的内容将用于语义检查和产生目标代码。通过符号表中

属性记录可进行相应上下文的语义检查。

例如,在 SC 语言中变量必须先声明后使用。

```
int A()
{
    x=1;
}
```

编译过程将提示: 'x'未定义。

又例如,在 SC 语言中,数组名不能用作左值。

```
:
char A[8];
A=1;
:
```

编译过程将提示: 此处 A 不能作为左值。

又例如,两个指针变量相加运算没有意义。

```
int * p1, * p2, * p3;
:
p3=p1+p2;
```

编译过程将提示: 指针之间不能进行加法运算。

6.2 符号表用到的主要数据结构

6.2.1 栈结构

6.2.1.1 为什么要用栈结构

下面先来看一段程序:

```
1 int main()
2 {
3     int x=1;
4     int sum1,sum2,sum3;
5     {
6         int x=2;
7         {
8             int x=3;
9             sum3=x+1;
10        }
11        sum2=x+1;
12    }
13    sum1=x+1;
14    printf("sum1=%d,sum2=%d,sum3=%d\n",sum1,sum2,sum3);
```

```

15     return 1;
16 }

```

运行结果如下：

```
sum1=2,sum2=3,sum3=4
```

这个运行结果只要学过 C 语言的想必都很容易理解，主要用到不同范围内的变量有不同作用域。上面有 3 个 x 变量声明，第 8 行定义的 x 是最后定义的，在第 10 行最早出了自己的作用域，第 6 行定义的 x，在第 12 行出了自己的作用域，第 3 行最先定义的 x 则是最后出自己的作用域，这个特征用 4 个字概括就是“后进先出”，那么具有这个特征的数据用什么结构描述最合适呢？当然是“栈”了。好，接下来我们写个栈结构，用于存储编译过程中用到的符号。

6.2.1.2 动态栈实现

在第 3 章已介绍过动态数组，动态栈的数据存储结构跟动态数组一样，只是在操作方式有所区别，下面来看一下动态栈结构定义：

```

/* 动态栈结构定义 */
typedef struct Stack
{
    void **base;      // 栈底指针
    void **top;       // 栈顶指针
    int stacksize;   // 栈当前可使用的最大容量，以元素个数计
} Stack;

/*****************
 * 功能：      初始化栈存储容量
 * stack：      栈存储结构
 * initSize：  栈初始化分配空间
 *****************/
void stack_init(Stack *stack, int initSize)
{
    stack->base = (void **)malloc(sizeof(void *) * initSize);
    if (!stack->base)
    {
        error("内存分配失败");
    }
    else
    {
        stack->top = stack->base;
        stack->stacksize = initSize;
    }
}

/*****************
 * 功能：      插入元素 element 为新的栈顶元素

```

```

* stack: 栈存储结构
* element: 要插入栈顶的新元素
* size: 栈元素实际数据尺寸
* 返回值: 栈顶元素
*****
void * stack_push(Stack * stack,void * element,int size)
{
    int newsize;
    if(stack->top>=stack->base + stack->stacksize)
    {
        newsize=stack->stacksize * 2;
        stack->base= (void **)realloc(stack->base,
                                       (sizeof(void **) * newsize));
        if(!stack->base)
        {
            return NULL;
        }
        stack->top=stack->base + stack->stacksize;
        stack->stacksize=newsize;
    }
    * stack->top= (void **)malloc(size);
    memcpy(* stack->top,element,size);
    stack->top++;
    return * (stack->top-1);
}

*****
* 功能: 弹出栈顶元素
* stack: 栈存储结构
*****
void stack_pop(Stack *stack)
{
    if(stack->top>stack->base)
    {
        free(* (--stack->top));
    }
}

*****
* 功能: 得到栈顶元素
* stack: 栈存储结构
* 返回值: 栈顶元素
*****
void * stack_get_top(Stack * stack)
{

```

```

void **element;
if(stack->top>stack->base)
{
    element=stack->top - 1;
    return * element;
}
else
{
    return NULL;
}

}

/*****判断栈是否为空*****
* 功能： 判断栈是否为空
* stack: 栈存储结构
* 返回值： 1 表示栈为空,0 表示栈非空
*****/
int stack_is_empty(Stack * stack)
{
    if(stack->top==stack->base){
        return 1;
    }
    else
    {
        return 0;
    }
}

/*****销毁栈*****
* 功能： 销毁栈
* stack: 栈存储结构
*****/
void stack_destroy(Stack * stack)
{
    void **element;

    for(element=stack->base;element<stack->top;element++)
    {
        free(* element);
    }
    if(stack->base)
    {
        free(stack->base);
    }
    stack->base=NULL;
}

```

```

    stack->top=NULL;
    stack->stacksize=0;
}

```

动态栈的数据存储结构与 4.3.2 节的动态数组存储结构完全一样,所以数据存储结构及内存动态增长方式相关介绍见 4.3.2 节,这里不再赘述。这里主要介绍一下栈操作,入栈通过 stack_push,入栈元素总是放在栈顶。出栈通过 stack_pop,出栈元素也总是栈顶元素。通过这两个成对操作,实现了元素的“后进先出”。另外,介绍一下 stack_is_empty 函数,可通过判断栈底指针与栈顶指针是否相等来判断栈是否为空,在这种情况下要注意,栈顶指针指向的是下一个进栈元素要插入位置,并不指向真正的栈顶元素,如图 6.3 所示的动态栈存储结构图。

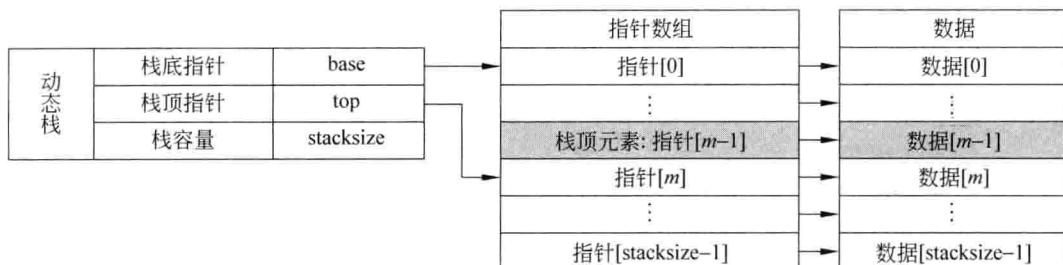


图 6.3 动态栈存储结构图

6.2.2 符号表结构

符号表采用刚介绍的动态栈结构存储,由于符号有全局与局部之分,全局符号的作用域是 SC 源文件,局部符号作用域是函数作用域或复合语句作用域,因此将符号表分成两张表,分别存储在下面的全局变量中。

```

Stack global_sym_stack,           //全局符号栈
      local_sym_stack;           //局部符号栈

```

下面开始介绍符号存储结构定义及符号表的相关操作函数。

6.2.2.1 符号存储结构定义

```

/* 符号存储结构定义 */
typedef struct Symbol
{
    int v;                                //符号的单词编码
    int r;                                //符号关联的寄存器
    int c;                                //符号关联值
    Type type;                            //符号数据类型
    struct Symbol * next;                 //关联的其他符号
    struct Symbol * prev Tok;             //指向前一定义的同名符号
} Symbol;

```

6.2.2.2 符号的登录

```

/************************************************************************/
* 功能： 将符号放在符号栈中
* v:    符号编号
* type: 符号数据类型
* c:    符号关联值
/************************************************************************/
Symbol * sym_direct_push(Stack * ss,int v,Type * type,int c)
{
    Symbol s,* p;
    s.v=v;
    s.type.t=type->t;
    s.type.ref=type->ref;
    s.c=c;
    s.next=NULL;
    p=(Symbol *)stack_push(ss,&s,sizeof(Symbol));
    return p;
}

/************************************************************************/
* 功能： 将符号放在符号栈中,动态判断是放入全局符号栈还是局部符号栈
* v:    符号编号
* type: 符号数据类型
* r:    符号存储类型
* c:    符号关联值
/************************************************************************/
Symbol * sym_push(int v,Type * type,int r,int c)
{
    Symbol * ps,**pps;
    TkWord * ts;
    Stack * ss;

    if(stack_is_empty(&local_sym_stack)==0)
    {
        ss=&local_sym_stack;
    }
    else
    {
        ss=&global_sym_stack;
    }
    ps=sym_direct_push(ss,v,type,c);
    ps->r=r;
}

//不记录结构体成员及匿名符号

```

```

if((v & SC_STRUCT) || v<SC_ANOM)
{
    //更新单词 sym_struct 或 sym_identifier 字段
    ts=(TkWord *)tktable.data[(v & ~SC_STRUCT)];
    if(v & SC_STRUCT)
        pps=&ts->sym_struct;
    else
        pps=&ts->sym_identifier;
    ps->prev_tok= * pps;
    * pps=ps;
}
return ps;
}

/***********************
* 功能： 将函数符号放入全局符号表中
* v:      符号编号
* type:   符号数据类型
*************************/
Symbol * func_sym_push(int v,Type * type)
{
    Symbol * s,**ps;
    s=sym_direct_push(&global_sym_stack,v,type,0);

    ps=&((TkWord *)tktable.data[v])->sym_identifier;
    //同名符号,函数符号放在最后->->...
    while(* ps !=NULL)
        ps=&(* ps)->prev_tok;
    s->prev_tok=NULL;
    * ps=s;
    return s;
}

Symbol * var_sym_put(Type * type,int r,int v,int addr)
{
    Symbol * sym=NULL;
    if((r & SC_VALMASK)==SC_LOCAL)           //局部变量
    {
        sym=sym_push(v,type,r,addr);
    }
    else if(v &&(r & SC_VALMASK)==SC_GLOBAL) //全局变量
    {
        sym=sym_search(v);
    }
}

```

```

    if(sym)
        error("%s 重定义\n", ((TkWord*)tkttable.data[v])->spelling);
    else
    {
        sym=sym_push(v,type,r|SC_SYM,0);
    }
}

//else 字符串常量符号
return sym;
}

/*************************************************************************
 * 功能：将节名称放入全局符号表
 * sec: 节名称
 * c: 符号关联值
*************************************************************************/
Symbol * sec_sym_put(char * sec,int c)
{
    TkWord * tp;
    Symbol * s;
    Type type;
    type.t=T_INT;
    tp=tkword_insert(sec);
    token=tp->tkcode;
    s=sym_push(token,&type,SC_GLOBAL,c);
    return s;
}

```

这里大家看到函数符号通过 func_sym_push 函数插入符号表，总是放在全局符号栈中。变量符号通过 var_sym_put 函数进入符号表，变量符号插入符号表时根据变量是全局的还是局部的插入相应表中，全局变量插入全局符号栈中，局部变量插入局部符号栈中。另外，大家还看到一个函数 sec_sym_put，将节名称放在符号栈中，什么是“节”我们将在第 7 章介绍 COFF 目标文件结构时讲述，目前大家只要知道程序中遇到的常量字符串都存放在这个符号中，例如 printf("Hello World!\n") 中，"Hello World!\n" 就存放在这个符号中。

另外，这里介绍一下单词表与符号表的关系，回顾一下第 4 章单词存储结构定义：

```

/* 单词存储结构定义 */
typedef struct TkWord
{
    int tkcode;                                //单词编码
    struct TkWord * next;                      //指向哈希冲突的其他单词
    char * spelling;                            //单词字符串
    struct Symbol * sym_struct;                //指向单词所表示的结构定义
    struct Symbol * sym_identifier;            //指向单词所表示的标识符
} TkWord;

```

其中,最后两个成员变量 sym_struct 和 sym_identifier 当时没讲。对于 sym_push 函数而言,在变量符号插入符号表的同时,还可以设置单词的这两个变量。通过这个机制能够实现从标识符字符串很容易找到与之相关的符号,符号结构定义中又记忆了单词编码,由符号访问单词字符串也很方便。通过这种设计,实现了单词与符号的双向查询,避免了查找遍历的过程。通过下面这个例子来理解一下这两个成员变量。

```
struct point      //单词 point 的 sym_struct 字段指向 point 结构定义
{
    short a;
    short b;
}
struct point pt; //单词 pt 的 sym_identifier 字段指向 pt 变量定义
```

6.2.2.3 符号的删除

```
/**************************************************************************
 * 功能: 弹出栈中符号直到栈顶符号为'b'
 * ptop: 符号栈栈顶
 * b:    符号指针
 **************************************************************************/
void sym_pop(Stack *ptop,Symbol *b)
{
    Symbol *s,**ps;
    TkWord *ts;
    int v;

    s=(Symbol *)stack_get_top(ptop);
    while(s !=b)
    {
        v=s->v;
        //更新单词表中 sym_struct sym_identifier
        if((v & SC_STRUCT) || v<SC_ANOM)
        {
            ts=(TkWord *)tktable.data[(v & ~SC_STRUCT)];
            if(v & SC_STRUCT)
                ps=&ts->sym_struct;
            else
                ps=&ts->sym_identifier;
            *ps=s->prev_tok;
        }
        stack_pop(ptop);
        s=(Symbol *)stack_get_top(ptop);
    }
}
```

6.2.2.4 符号的查找

```
*****
* 功能：查找结构定义
* v:    符号编号
*****
Symbol * struct_search(int v)
{
    if(v>=tktable.count)
        return NULL;
    else
        return((TkWord *)tktable.data[v])->sym_struct;
}

*****
* 功能：查找结构定义
* v:    符号编号
*****
Symbol * sym_search(int v)
{
    if(v>=tktable.count)
        return NULL;
    else
        return((TkWord *)tktable.data[v])->sym_identifier;
}
```

6.2.3 数据类型结构

如果 SC 语言的数据类型是有限可枚举的几种，那么类型定义起来就比较简单了，但是我们的 SC 语言的数据类型系统是开放的，是可扩展的，是不可枚举的，主要体现在以下几个方面：

- (1) 可以定义各种结构体；
- (2) 可以定义各种类型的数组，数组的维数理论上不加限制；
- (3) 可以定义各种类型的指针；
- (4) 可以定义各种函数。

所以我们希望寻找一种通用机制来描述存储 SC 语言编译过程中遇到的各种类型，下面给出这种通用类型结构的定义。

```
/* 类型结构定义 */
typedef struct Type
{
    int t;                      //数据类型
    struct Symbol * ref;        //引用符号
} Type;
```

数据类型 t 可取下面的枚举值,前面 7 种基础类型,互相独立,这几种基本类型可以与 T_ARRAY 类型组合,代表这些基本类型的数组。注意本章介绍的基本类型要与第 3 章 SC 语言定义时介绍的基本类型区别开来。

```
/* 数据类型编码 */
enum e_TypeCode
{
    T_INT      = 0,           //整型
    T_CHAR     = 1,           //字符型
    T_SHORT    = 2,           //短整型
    T_VOID     = 3,           //空类型
    T_PTR      = 4,           //指针
    T_FUNC     = 5,           //函数
    T_STRUCT   = 6,           //结构体

    T_BTYPE   = 0x000f,       //基本类型掩码
    T_ARRAY   = 0x0010,       //数组
};

};
```

具体到上面的通用类型结构如何应用,如何存储编译过程中遇到的各种数据类型,在符号表的构造过程中会逐一讲到。

与数据类型相关的全局变量:

```
Type char_pointer_type,          //字符串指针
int_type,                      //int 类型
default_func_type;             //缺省函数类型
```

6.2.4 存储类型

符号除了具有数据类型外,还有存储类型,存储类型为如下枚举值,注意这里的存储类型与 3.6.3 节介绍的存储类型有所区别。

```
/* 存储类型 */
enum e_StorageClass
{
    SC_GLOBAL  = 0x00f0,        //包括整型常量、字符常量、字符串常量、全局变量、函数定义
    SC_LOCAL   = 0x00f1,        //栈中变量
    SC_LLOCAL  = 0x00f2,        //寄存器溢出存放栈中
    SC_CMP     = 0x00f3,        //使用标志寄存器
    SC_VALMASK = 0x00ff,        //存储类型掩码
    SC_LVAL    = 0x0100,        //左值
    SC_SYM     = 0x0200,        //符号

    SC_ANOM    = 0x10000000,    //匿名符号
    SC_STRUCT  = 0x20000000,    //结构体符号
    SC_MEMBER  = 0x40000000,    //结构成员变量
};
```

```
    SC_PARAMS = 0x80000000, //函数参数
};
```

6.3 符号表的构造过程

6.3.1 外部声明

```
/****************************************************************************
 * 功能： 声明与函数定义
 * l: 存储类型，局部的还是全局的
 *****/
void external_declarator(int l)
{
    Type btype,type;
    int v,has_init,r,addr;
    Symbol * sym;
    if(!type_specifier(&btype))
    {
        expect("<类型区分符>");
    }

    if(btype.t==T_STRUCT && token==TK_SEMICOLON)
    {
        get_token();
        return;
    }
    while(1)
    {
        type=btype;
        declarator(&type,&v,NULL);

        if(token==TK_BEGIN)          //函数定义
        {
            if(l==SC_LOCAL)
                error("不支持函数嵌套定义");

            if((type.t & T_BTYPE)!=T_FUNC)
                expect("<函数定义>");

            sym=sym_search(v);
            if(sym)                  //函数前面声明过，现在给出函数定义
            {
                if((sym->type.t & T_BTYPE)!=T_FUNC)
                    error("'%" ,get_tkstr(v));
                sym->type=type;
            }
        }
    }
}
```

```
    }
    else
    {
        sym=func_sym_push(v,&type);
    }
    sym->r=SC_SYM|SC_GLOBAL;
    funcbody(sym);
    break;
}
else
{
    if((type.t & T_BTYPY)==T_FUNC) //函数声明
    {
        if(sym_search(v)==NULL)
        {
            sym=sym_push(v,&type,SC_GLOBAL|SC_SYM,0);
        }
    }
    else //变量声明
    {
        r=0;
        if(!(type.t & T_ARRAY))
            r |= SC_LVAL;

        r |= l;
        has_init=(token==TK_ASSIGN);

        if(has_init)
        {
            get_token();
            initializer(&type);
        }
        sym=var_sym_put(&type,r,v,addr);
    }
    if(token==TK_COMMA)
    {
        get_token();
    }
    else
    {
        skip(TK_SEMICOLON);
        break;
    }
}
```

}

从上面的代码能够解读出如下几条信息。

(1) 数据类型解析不是在一个函数中完成的,通过几个函数的协调配合,才能形成完整的数据类型。

(2) 只有结构体定义的结尾加分号是合法的,其他类型标识符直接加分号没有意义,例如

```
struct point {int x; int y}; //合法  
int; //非法
```

(3) 同一函数标识符允许先声明再定义,但是声明与函数定义必须一致,否则会提示“重定义”,例如

```
int add; //声明  
int add(int x, int y) {...} //add 重定义
```

(4) 数组名不能用作左值。

(5) 函数放入符号表使用 `func_sym_push` 函数,这个函数保证函数符号都存放在全局符号栈,变量放入符号表通过 `var_sym_put` 函数,这个函数会根据变量是局部变量还是全局变量,放入相应的符号栈中。通过两个例子,我们来看一下符号是如何在内存中存储的。函数符号 `int __cdecl func1(char x, short y);` 的存储结构图如图 6.4 所示,变量符号 `char **g_ppstr;` 存储结构图如图 6.5 所示。

Name	Value	
sym	0x00374db8	
u	0x00000030	func1 函数名
r	0x000002f0	SC_SYM SC_GLOBAL = 0x02f0
c	0x00000000	
type	{ ... }	
t	0x00000005	T_FUNC = 5, // 函数
ref	0x00374d60	??? 待后面讲
next	0x00000000	
prev Tok	0x00000000	

图 6.4 函数符号存储结构图

Name	Value
sym	0x00374F18
v	0x00000033
r	0x000003F0
c	0x00000000
type	{...}
t	0x00000004
ref	0x00374e68
next	0x00000000
prev Tok	0x00000000

图 6.5 变量符号存储结构图

6.3.2 类型区分符

```
*****
* 功能:          类型区分符
* type(输出):  数据类型
* 返回值:        是否发现合法的类型区分符
*****
```

```
int typeSpecifier(Type * type)
{
    int t,type_found;
    Type type1;
    t=0;
    type_found=0;
    switch(token)
    {
        case KW_CHAR:
            t=T_CHAR;
            type_found=1;
            get_token();
            break;
        case KW_SHORT:
            t=T_SHORT;
            type_found=1;
            get_token();
            break;
        case KW_VOID:
            t=T_VOID;
            type_found=1;
            get_token();
            break;
        case KW_INT:
            t=T_INT;
            type_found=1;
            get_token();
            break;
        case KW_STRUCT:
            structSpecifier(&type1);
            type->ref=type1.ref;
            t=T_STRUCT;
            type_found=1;
            break;
        default:
            break;
    }
}
```

```

    }
    type->t=t;
    return type_found;
}

```

通过这个函数可以得到基本类型,即这个函数完成如下例子加粗部分的解析:

```

int x;
short arr[10]
char * str
struct point {int x;int y};
int add(int x,int y)

```

这个函数只完成基本类型的解析,如果这个函数刚解析完我们就给数据类型下结论,就像“盲人摸象”,还是非常片面的,因为离数据类型解析完整得出结论,还有一段距离。

6.3.3 结构区分符

```

/****************************************************************************
 * 功能:          解析区分符
 * type(输出):   结构类型
 *****/
void struct_specifier(Type * type)
{
    int v;
    Symbol * s;
    Type type1;

    get_token();
    v=token;

    get_token();

    if (v<TK_IDENT)           //关键字不能作为结构名称
        expect("结构体名");
    s=struct_search(v);
    if (!s)
    {
        type1.t=KW_STRUCT;
        // -1 赋值给 s->c,标识结构体尚未定义
        s=sym_push(v|SC_STRUCT,&type1,0,-1);
        s->r=0;
    }

    type->t=T_STRUCT;
}

```

```

    type->ref=s;

    if(token==TK_BEGIN)
    {
        struct_declaratoin_list(type);
    }
}
}

```

6.3.3.1 结构声明符表

```

/**************************************************************************
* 功能：      解析
* type(输出)： 结构类型
*************************************************************************/
void struct_declaratoin_list(Type *type)
{
    int maxalign,offset;
    Symbol * s,**ps;
    s=type->ref;
    get_token();
    if(s->c != -1)                                //s->c 记录结构体尺寸
        error("结构体已定义");
    maxalign=1;
    ps=&s->next;
    offset=0;
    while(token != TK_END)
    {
        struct_declaratoin(&maxalign,&offset,&ps);
    }
    skip(TK_END);

    s->c=calc_align(offset,maxalign); //结构体大小
    s->r=maxalign;                  //结构体对齐
}

/**************************************************************************
* 功能：  计算字节对齐位置
* n:      未对齐前值
* align: 对齐粒度
*************************************************************************/
int calc_align(int n,int align)
{
    return((n+align-1)&(~(align-1)));
}

```

这个函数中引用的 calc_align 函数有必要在这里讲一下，这个函数实现什么功能，表 6.1 给出该函数的一组输入输出值，大家研究一下就知道这个函数的确切作用了。

表 6.1 calc_align 输入输出举例

输入参数 n	输入参数 align	calc_align 返回值	输入参数 n	输入参数 align	calc_align 返回值
9	1	9	7	1	7
9	2	10	7	2	8
9	4	12	7	4	8

6.3.3.2 结构体成员声明

```
/****************************************************************************
 * 功能：          解析结构体成员声明
 * maxalign(输入,输出)： 成员最大对齐粒度
 * offset(输入,输出)： 偏移量
 * ps(输出)：       结构定义符号
 *****/
void struct_declaration(int *maxalign,int *offset,Symbol ***ps)
{
    int v,size,align;
    Symbol *ss;
    Type type1,btype;
    int force_align;
    typeSpecifier(&btype);
    while(1)
    {
        v=0;
        type1=btype;
        declarator(&type1,&v,&force_align);
        size=type_size(&type1,&align);

        if(force_align & ALIGN_SET)
            align=force_align & ~ALIGN_SET;

        *offset=calc_align(*offset,align);

        if(align>*maxalign)
            *maxalign=align;
        ss=sym_push(v|SC_MEMBER,&type1,0,*offset);
        *offset +=size;
        **ps=ss;
    }
}
```

```

* ps=&ss->next;

if(token==TK_SEMICOLON)
    break;
skip(TK_COMMA);
}
skip(TK_SEMICOLON);
}

/* 强制对齐标志宏定义 */
#define ALIGN_SET 0X100

```

这里通过 struct s1 结构定义,来看一下上面几个函数对结构体定义解析后,最终在符号表中是如何存储的,如图 6.6 所示。

```

struct s1
{
    char a;
    short b;
    int c;
};

```

Name	Value
type	0x0012Fe48
t	0x00000006 T_STRUCT = 6, // 结构体
ref	0x00374818
v	0x2000002c s1
r	0x00000004 结构体对齐粒度
c	0x00000008 结构体尺寸
type	{...}
t	0x00000020 KW_STRUCT = 0x20
ref	0xcccccccc
next	0x003748c8
v	0x4000002d a
r	0x00000000
c	0x00000000 成员变量a偏移位置
type	{...}
t	0x00000001 T_CHAR = 1, // 字符型
ref	0xcccccccc
next	0x00374978
v	0x4000002e b
r	0x00000000
c	0x00000002 成员变量b偏移位置
type	{...}
t	0x00000002 T_SHORT = 2, // 短整型
ref	0xcccccccc
next	0x00374a28
v	0x4000002f c
r	0x00000000
c	0x00000004 成员变量c偏移位置
type	{...}
t	0x00000000 T_INT = 0, // 整型
ref	0xcccccccc
next	0x00000000
prev_tok	0xcccccccc
prev_tok	0xcccccccc
prev_tok	0x00000000

图 6.6 结构体定义存储结构图

6.3.3.3 结构成员对齐

```
*****
* 功能：          解析结构成员对齐
* force_align(输出)：强制对齐粒度
*****
void struct_member_alignment(int * force_align)
{
    int align=1;
    if(token==KW_ALIGN)
    {
        get_token();
        skip(TK_OPENPA);
        if(token==TK_CINT)
        {
            get_token();
            align=tkvalue;
        }
        else
            expect("整数常量");
        skip(TK_CLOSEPA);
        if(alin!=1 && align!=2 && align!=4)
            align=1;
        align |= ALIGN_SET;
        *force_align=align;
    }
    else
        *force_align=1;
}
```

这里重点讲一下数据结构对齐的概念，数据结构对齐是数据在内存中分配与访问的方式。具体来说包括数据对齐与数据结构填充两方面，这两方面相互独立但又相互关联。当计算机在特定的内存地址读写数据时，将以机器字长为单位进行（32位系统为4字节）。数据对齐意味着数据放置的地址与起始地址的距离为机器字长的整数倍，这种做法更利于CPU对内存的访问，因此能够提升系统性能。有些数据结构大小不是机器字长的整数倍，因而为了达到数据对齐的目的，需要在这些数据的末尾到下一个有效数据之间填充空白字节，这就是所谓的数据结构填充。

估计很多人对数据结构对齐这个概念比较陌生，又似曾相识，好像在哪见过，看看图6.7大家是不是很熟悉？只不过绝大多数人对结构对齐的相关设置可以视而不见，因为一般情况下大家没必要关心这个，只要按编译器默认设置就可以了。

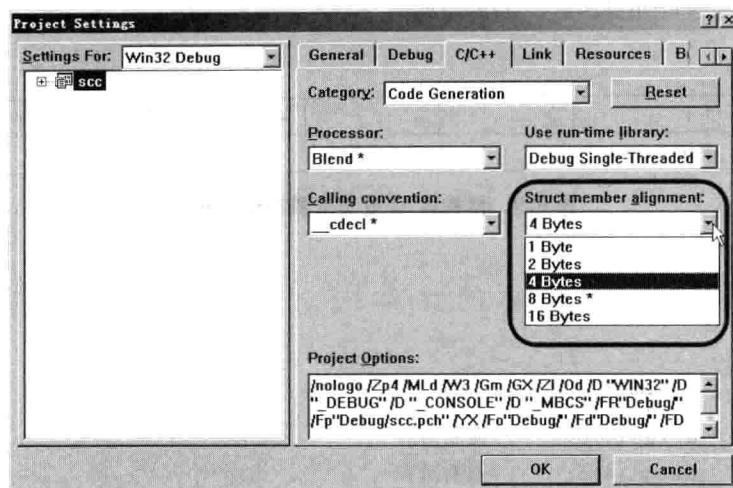


图 6.7 VC6 结构成员对齐设置

结构成员默认对齐要求,如表 6.2 所示,通过 type_size 函数实现默认对齐要求,该函数将在本章讲述 sizeof 表达式如何求类型尺寸时给出。

表 6.2 不同类型数据的默认对齐边界

类 型	对 齐 方 式	类 型	对 齐 方 式
char	与字节边界对齐	int (32 位)	与 32 位边界对齐
short(16 位)	与偶数字节边界对齐	结构	任何成员的最大对齐要求

通过下面 3 个结构定义及相应存储结构图,可以理解结构对齐的含义,参见图 6.8~图 6.10(灰色底纹表示填充字节)。

示例 1:

```
struct s1      //按表 6.2 默认对齐规则对齐
{
    char a;
    short b;
    int c;
};
```

0	1	2	3	4	5	6	7
a		b		c			

图 6.8 结构体 s1 成员变量存储结构图

示例 2:

```
struct s2
{
    char a;          //强制对齐
    short __align(1)b;
```

```
int __align(1)c; //强制对齐
};
```

0	1	2	3	4	5	6
a	b		c			

图 6.9 结构体 s2 成员变量存储结构图

示例 3：

```
struct s3
{
    char a;
    short __align(4)b; //强制对齐
    int __align(4)c; //强制对齐
};
```

0	1	2	3	4	5	6	7	8	9	10	11
a				b					c		

图 6.10 结构体 s3 成员变量存储结构图

6.3.4 声明符

```
/****************************************************************************
 * 功能：          解析
 * type:          数据类型
 * v(输出)：       单词编号
 * force_align(输出)： 强制对齐粒度
*/
void declarator(Type * type, int * v, int * force_align)
{
    int fc;
    while(token==TK_STAR)
    {
        mk_pointer(type);
        get_token();
    }
    function_calling_convention(&fc);
    if(force_align)
        struct_member_alignment(force_align);
    direct_declarator(type,v,fc);
}
```

这里举例来看一下指针类型是如何存储的，指针类型 `char ** ppstr;` 的存储结构图如图 6.11 所示。

Name	Value
type	0x0012fec4
t	0x00000004 T_PTR = 4
ref	0x00374dc8
v	0x10000000 SC_ANOM = 0x10000000, // 匿名符号
r	0x00000000
c	0xffffffff
type	{...}
t	0x00000004 T_PTR = 4
ref	0x00374d70
v	0x10000000 SC_ANOM = 0x10000000, // 匿名符号
r	0x00000000
c	0xffffffff
type	{...}
t	0x00000001 T_CHAR = 1
ref	0xcccccccc
next	0x00000000
prev_tok	0xcccccccc
next	0x00000000
prev_tok	0xcccccccc

图 6.11 指针类型存储结构图

6.3.4.1 函数调用约定

```
*****
* 功能：解析函数调用约定
* fc(输出)：调用约定
* 用于函数声明上，用在数据声明上忽略掉
*****
void function_calling_convention(int * fc)
{
    * fc=KW_CDECL;
    if(token==KW_CDECL || token==KW_STDCALL)
    {
        * fc=token;
        get_token();
    }
}
```

函数调用约定有__cdecl 和 __stdcall 两种方式，默认为 __cdecl 方式。

6.3.4.2 直接声明符

```
*****
* 功能：      解析直接声明符
* type(输入,输出)： 数据类型
* v(输出)：      单词编号
* func_call:     函数调用约定
*****
void direct_declarator(Type * type,int * v,int func_call)
{
```

```

if(token>=TK_IDENT)
{
    * v=token;
    get_token();
}
else
{
    expect("标识符");
}
direct_declarator_postfix(type,func_call);
}

```

该函数很重要，在声明中处于核心地位，因为声明中其他部分都是用来形容这个标识符的，请看下面一组声明的例子，加粗的部分就是本函数负责识别的标识符。

```

int x;
int arr[10];
char * str
int add(int x,int y)

```

6.3.4.3 声明后缀

```

/****************************************************************************
 * 功能：          直接声明符后缀
 * type(输入,输出)： 数据类型
 * func_call：      函数调用约定
****************************************************************************/
void direct_declarator_postfix(Type * type,int func_call)
{
    int n;
    Symbol * s;

    if(token==TK_OPENPA)
    {
        parameter_type_list(type,func_call);
    }
    else if(token==TK_OPENBR)
    {
        get_token();
        n=-1;
        if(token==TK_CINT)
        {
            get_token();
            n=tkvalue;
        }
    }
}
```

```

skip(TK_CLOSEBR);
direct_declarator_postfix(type, func_call);
s=sym_push(SC_ANOM, type, 0, n);
type->t=T_ARRAY|T_PTR;
type->ref=s;
}
}

```

在该函数中,主要实现了数组的解析,下面举例来看一下数组类型在符号表中是如何存储的。数组类型 `int arr[8][10]`; 存储结构图如图 6.12 所示。

Name	Value
type	0x0012fec4
t	0x00000014 T_ARRAY T_PTR = 0x0014
ref	0x00374a28
v	0x10000000 匿名符号
r	0x00000000
c	0x00000006 第一维的元素个数
type	{...}
t	0x00000014 T_ARRAY T_PTR = 0x0014
ref	0x003749d0
v	0x10000000 匿名符号
r	0x00000000
c	0x00000008 第二维的元素个数
type	{...}
t	0x00000001 T_CHAR = 1
ref	0xcccccccc
next	0x00000000
prev Tok	0xcccccccc
next	0x00000000
prev Tok	0xcccccccc

图 6.12 数组类型存储结构图

6.3.4.4 形参类型表

```

/****************************************************************************
 * 功能:          解析形参类型表
 * type(输入,输出): 数据类型
 * func_call:      函数调用约定
 *****/
void parameter_type_list(Type * type, int func_call)
{
    int n;
    Symbol **plast, * s, * first;
    Type pt;

    get_token();
    first=NULL;
    plast=&first;

    while(token !=TK_CLOSEPA)

```

```

{
    if(!type_specifier(&pt))
    {
        error("无效类型标识符");
    }
    declarator(&pt,&n,NULL);
    s=sym_push(n|SC_PARAMS,&pt,0,0);
    *plast=s;
    plast=&s->next;
    if(token==TK_CLOSEPA)
        break;
    skip(TK_COMMA);
    if(token==TK_ELLIPSIS)
    {
        func_call=KW_CDECL;
        get_token();
        break;
    }
}
skip(TK_CLOSEPA);

//此处将函数返回类型存储,然后指向参数,最后将 type 设为函数类型,引用的相关信息
//放在 ref 中
s=sym_push(SC_ANOM,type,func_call,0);
s->next=first;
type->t=T_FUNC;
type->ref=s;
}

```

这里还是用一个例子来说明函数类型在符号表中是如何存储的：函数类型为“`int __cdecl func1(char x, short y);`”，存储结构图如图 6.13 所示。

6.3.4.5 函数体

```

*****
* 功能：解析函数体
* sym: 函数符号
*****
void funcbody(Symbol * sym)
{
    /* 放一匿名符号在局部符号表中 */
    sym_direct_push(&local_sym_stack,SC_ANOM,&int_type,0);
    compound_statement(NULL,NULL);
    /* 清空局部符号栈 */
}

```

Name	Value	
type	0x0012Fec4	
t	0x00000005	T_FUNC = 5
ref	0x00374c68	
u	0x10000000	匿名符号
r	0x00000029	调用约定 KW_CDECL = 0x29
c	0x00000000	
type	{...}	
t	0x00000000	返回值类型 T_INT = 0
ref	0xffffffff	
next	0x00374bb8	
u	0x80000031	形参x SC_PARAMS = 0x80000000
r	0x00000000	
c	0x00000000	
type	{...}	
t	0x00000001	形参x类型 T_CHAR = 1
ref	0xffffffff	
next	0x00374c68	
u	0x80000032	形参y SC_PARAMS = 0x80000000
r	0x00000000	
c	0x00000000	
type	{...}	
t	0x00000002	形参y类型 T_SHORT = 2
ref	0xffffffff	
next	0x00000000	
prev_tok	0x80000000	
prev_tok	0x00000000	
prev_tok	0xffffffff	

图 6.13 函数类型存储结构图

```
    sym_pop(&local_sym_stack,NULL);
```

放入匿名符号后的局部符号栈，如图 6.14 所示。通过局部符号栈的栈顶及栈底指针，用下面的公式求得目前局部符号栈中符号个数：

`local_sym_stack.top - local_sym_stack.base = 0x0037241c - 0x00372418 = 1`
`sizeof(void *) 4`

从图 6.14 看出目前局部符号栈中确实只有一个匿名符号。

Name	Value
local_sym_stack.base	0x00372418
local_sym_stack.top	0x0037241c
[-] (Symbol*)local_sym_stack.base[0]	0x003724f0
[-] v	0x10000000 SC_ANOM = 0x10000000, //匿名符号
[-] r	0xffffffff
[-] c	0x00000000
[+] type	{...}
[+] next	0x00000000
[+] prev Tok	0xffffffff

图 6.14 放入匿名符号后的局部符号栈

6.3.5 变量初始化

```
/* 功能：解析初值符
 * type: 变量类型
 */
void initializer(Type * type)
```

```
{
    if(type->t & T_ARRAY)
    {
        get_token();
    }
    else
    {
        assignment_expression();
    }
}
```

6.3.6 复合语句

```
*****
* 功能：解析复合语句
* bsym: break 跳转位置
* csym: continue 跳转位置
*****
void compound_statement(int *bsym,int *csym)
{
    Symbol *s;
    s=(Symbol*)stack_get_top(&local_sym_stack);

    get_token();
    while(is_type_specifier(token))
    {
        external_declaration(SC_LOCAL);
    }
    while(token !=TK_END)
    {
        statement(bsym,csym);
    }
    sym_pop(&local_sym_stack,s);
    get_token();
}
```

6.3.7 sizeof 表达式

```
*****
* 功能：解析 sizeof 表达式
*****
void sizeof_expression()
{
    int align,size;
```

```

Type type;

get_token();
skip(TK_OPENPA);
typeSpecifier(&type);
skip(TK_CLOSEPA);

size=type_size(&type,&align);
if(size<0)
    error("sizeof 计算类型尺寸失败");
}

/******************
 * 功能：返回类型长度
 * t: 数据类型指针
 * a: 对齐值
*****************/
int type_size(Type *t,int *a)
{
    Symbol *s;
    int bt;
    //指针类型长度为 4 字节
    int PTR_SIZE=4;

    bt=t->t & T_BTYPE;
    switch(bt)
    {
        case T_STRUCT:
            s=t->ref;
            * a=s->r;
            return s->c;

        case T_PTR:
            if(t->t & T_ARRAY)
            {
                s=t->ref;
                return type_size(&s->type,a) * s->c;
            }
            else
            {
                * a=PTR_SIZE;
            }
    }
}

```

```

        return PTR_SIZE;
    }

    case T_INT:
        *a=4;
        return 4;

    case T_SHORT:
        *a=2;
        return 2;

    default:      //char,void,function
        *a=1;
        return 1;
    }
}

```

6.3.8 初等表达式

```

/***********************
 * 功能：解析初等表达式
*************************/
void primary_expression()
{
    int t, addr;
    Type type;
    Symbol * s;
    switch(token)
    {
        case TK_CINT:
        case TK_CCHAR:
            get_token();
            break;
        case TK_CSTR:
            t=T_CHAR;
            type.t=t;
            mk_pointer(&type);
            type.t |=T_ARRAY;
            var_sym_put(&type, SC_GLOBAL, 0, addr);
            initializer(&type);
            break;
        case TK_OPENPA:
            get_token();

```

```

expression();
skip(TK_CLOSEPA);
break;
default:
    t=token;
    get_token();
    if(t<TK_IDENT)
        expect("标识符或常量");
    s=sym_search(t);
    if(!s)
    {
        if(token !=TK_OPENPA)
            error("'%s'未声明\n",get_tkstr(t));
        s=func_sym_push(t,&default_func_type); //允许函数不声明,直接引用
        s->r=SC_GLOBAL|SC_SYM;
    }
    break;
}
}

```

在初等表达式中,有两种情况需要操作符号表,第一种情况是函数引用了常量字符串,例如 printf("Hello World\n") 中直接引用的字符串;第二种情况是函数使用前没有声明。这里要注意函数允许不声明直接引用,变量必须先声明后使用。

6.4 控制程序

控制程序包括 main 主函数、init 初始化函数和 cleanup 扫尾清理函数 3 个函数。

```

*****
* 功能: main 主函数
*****
int main(int argc,char **argv)
{
    fin=fopen(argv[1],"rb");
    if(!fin)
    {
        printf("不能打开 SC 源文件!\n");
        return 0;
    }

    init();
    getch();
    get_token();
    translation_unit();

```

```

cleanup();
fclose(fin);
return 1;
}

//*****************************************************************************
* 功能：初始化
*****
void init()
{
    line_num=1;
    init_lex();

    stack_init(&local_sym_stack,8);
    stack_init(&global_sym_stack,8);
    sym_sec_rdata=sec_sym_put(".rdata",0);

    int_type.t=T_INT;
    char_pointer_type.t=T_CHAR;
    mk_pointer(&char_pointer_type);
    default_func_type.t=T_FUNC;
    default_func_type.ref=sym_push(SC_ANOM,&int_type,KW_CDECL,0);
}

```

从 init 函数看出, 正式开始编译前, 全局符号表中已经预先放入了几个符号, 此时的全局符号栈如图 6.15 所示, 通过全局符号栈的栈顶和栈底指针, 用下面的公式求得, 目前局部符号栈中放入了 3 个符号。

$$\text{global_sym_stack.top} - \text{global_sym_stack.base} = \frac{0x0037252c - 0x00372520}{\text{sizeof(void *)}} = 3$$

Name	Value
global_sym_stack.base	0x00372520
global_sym_stack.top	0x0037252c
(Symbol*)global_sym_stack.base[0]	0x003725e8 .rdata
v	0x0000002b
r	0x000000f0
c	0x00000000
type	{...}
t	0x00000000
ref	0xffffffffcc
next	0x00000000
prev Tok	0x00000000
(Symbol*)global_sym_stack.base[1]	0x00372638
v	0x10000000 SC_ANOM = 0x10000000
r	0x00000000
c	0xffffffff
type	{...}
t	0x00000001 T_CHAR = 1
ref	0x00000000
next	0x00000000
prev Tok	0xffffffffcc
(Symbol*)global_sym_stack.base[2]	0x00372690
v	0x10000000 SC_ANOM = 0x10000000
r	0x00000029
c	0x00000000
type	{...}
t	0x00000000 T_INT = 0
ref	0x00000000
next	0x00000000
prev Tok	0xffffffffcc

图 6.15 初始化后的全局符号栈

```
*****
* 功能：扫尾清理工作
*****
void cleanup()
{
    int i;
    sym_pop(&global_sym_stack, NULL);
    stack_destroy(&local_sym_stack);
    stack_destroy(&global_sym_stack);

    printf("\ntktable.count=%d\n", tktable.count);
    for(i=TK_IDENT;i<tktable.count;i++)
    {
        free(tktable.data[i]);
    }
    free(tktable.data);
}
```

6.5 成果展示

下面通过一个简单而有代表性的例子展示一下本章代码产生的成果。代码源文件为 symtab_demo.c，其符号表存储结构如图 6.16 所示。通过下面两个公式：

$$\text{global_sym_stack.top} - \text{global_sym_stack.base} = \frac{0x00372540 - 0x00372520}{\text{sizeof(void *)}} = 8$$

$$\text{local_sym_stack.top} - \text{local_sym_stack.base} = \frac{0x003724c8 - 0x003724b8}{\text{sizeof(void *)}} = 4$$

我们知道全局符号表中共有 8 个符号，前 3 个（第 0~2）已经在 6.4 节讲过。第 3 个是 struct point 结构定义，第 4 个是成员变量 x，第 5 个是成员变量 y，第 6 个记录了 main 函数返回值类型，第 7 个是 main 函数符号。局部符号表中共有 4 个符号，第 0 个符号已经在 6.3.4.5 节讲过，第 1 个是局部变量 pt，其类型引用全局符号表 struct point 结构定义，第 2 个是 char * 指针类型定义，第 3 个是局部变量 pstr。

```
*****
* symtab_demo.c 源代码
*****
struct point
{
    short x;
    short y;
};

void main()
{
    struct point pt;
    char * pstr;
}
```

Name	Value	Name	Value
global_sym_stack.base	0x00372528	local_sym_stack.base	0x00372ab8
global_sym_stack.top	0x00372540	local_sym_stack.top	0x00372ac8
(Symbol*)global_sym_stack.base[0]	0x003725e0	(Symbol*)local_sym_stack.base[0]	0x00374ad8
(Symbol*)global_sym_stack.base[1]	0x00372638	(Symbol*)local_sym_stack.base[1]	0x00374bf8
(Symbol*)global_sym_stack.base[2]	0x00372698	u	0x00000000
(Symbol*)global_sym_stack.base[3]	0x00374818	r	0x000000f1
v	0x20000002c	c	SC_LOCAL SC_LOCAL = 0x01ff
r	0x00000002	t	0xcccccccc
c	0x00000004	t	{...}
type	{...}	ref	0x00000006 T_STRUCT = 6
t	0x00000008	next	0x00374818
ref	0x00000008	prev_tok	0x00000000
next	0x00000008	(Symbol*)local_sym_stack.base[2]	7
prev_tok	0x00000008	u	0x00000000 SC_ANOM = 0x10000000
(Symbol*)global_sym_stack.base[4]	0x003748c8	r	0x10000000
(Symbol*)global_sym_stack.base[5]	0x00374978	c	0x00000000
(Symbol*)global_sym_stack.base[6]	0x00374aa8	type	0xfffffff
u	0x10000000	{...}	{...}
r	0x000000029	t	0x00000001 T_CHAR = 1
c	0x00000000	ref	0xcccccccc
type	{...}	next	0x00000000
t	0x00000003	prev_tok	0x00000000
ref	0x00000000	(Symbol*)local_sym_stack.base[3]	0x00374e98
next	0x00000000	u	0x00000031
prev_tok	0x00000000	r	0x000000f1 SC_LOCAL SC_LOCAL = 0x01ff
(Symbol*)global_sym_stack.base[7]	0x00374a80	c	0xcccccccc
main	0x00000002f	type	{...}
u	0x00000002f	t	0x00000004 T_PTR = 4
r	0x00000000	ref	0x003740e0
c	0x00000000	next	0x00000000
type	{...}	prev_tok	0x00000000
t	0x00000005	T_FUNC = 5	0x00000000
ref	0x00000000	next	0x00374a28
next	0x00000000	prev_tok	0x00000000

图 6.16 符号表成果展示

第 7 章

生成COFF目标文件

逢山开路，遇水叠桥

——罗贯中

读者可能会觉得语法分析讲完了，符号表也讲完了，是不是该介绍语义分析了，但还得等一等，因为在介绍语义分析之前还有一些准备工作需要做。首先需要解决的是语义如何表达、用什么来表达的问题，这就涉及 x86 目标语言。另外，如果生成了 x86 目标语言代码，放在什么地方？我们去打水之前总得先把水桶或者水杯准备好吧。盛放目标语言代码的容器有好多种，Windows 帝国的人们普遍用 COFF 格式，Linux 帝国的人们喜欢用 ELF 格式。SC 语言虽然是跨平台的，但 SCC 编译器目前只考虑在 Windows 平台上运行，所以 COFF 格式就成为我们的不二选择。COFF 这个水桶结构及使用方法多少有些复杂，如果不看说明书，我们甚至连水从哪倒进去都不知道，所以还要耐着性子学习一下。

7.1 COFF 文件结构

Windows 使用的目标文件格式为 COFF(Common Object File Format)格式，中文名称为通用对象文件格式，使用的可执行文件格式为 PE 文件格式，第 10 章链接器链接生成的就是 PE 文件格式。PE 格式与 COFF 格式有一些共用内容，将在本章一并讲述，在第 10 章用到时，请读者参考本章相应章节。

7.1.1 基本概念

表 7.1 为本章用到的基本概念。

表 7.1 本章用到的基本概念

名 称	说 明
属性证书	用来将可校验的声明与映像关联起来的证书。有许多不同的可校验声明可以与文件关联，最常用的一种就是软件制造商用来指明映像的消息摘要是什么的声明。消息摘要与校验和类似，但想要伪造它却极其困难。因此对一个文件进行修改并保持它的消息摘要与原始文件一致是非常困难的。正如制造商所做的那样，可以使用公钥或私钥加密机制来校验声明
日期/时间戳	由于不同目的而用于 PE 或 COFF 文件中好几个地方的戳。戳的格式与 C 运行时库时间函数所使用的戳的格式相同

续表

名 称	说 明
文件指针	某项内容在文件被链接器处理前(如果是目标文件)或者被加载器处理前(如果是映像文件)在文件自身中的位置。换句话说,这是一个位于存储在磁盘上的文件中的位置
链接器	指的是随 Microsoft Visual Studio 提供的链接器
映像文件	可执行文件;或者是 EXE 文件,或者是 DLL 文件。映像文件可以被认为是“内存映像”。术语“映像文件”经常用来代替“可执行文件”,因为后者有时仅用来指代 EXE 文件
目标文件	作为链接器的输入文件。链接器生成一个映像文件,而这个映像文件又作为加载器的输入
保留,必须为 0	对一个域的这种描述表明,对于生成这个域的程序来说必须将这个域设置为 0,对于使用这个域的程序来说必须忽略它的值
RVA	相对虚拟地址。对于映像文件来说,它是某项内容被加载进内存后的地址减去映像文件的基址。某项内容的 RVA 几乎总是与它在磁盘上的文件中的位置(文件指针)不同 对于目标文件来说,RVA 并没有什么意义,因为内存位置尚未分配。在这种情况下,RVA 是一个节(后面将要描述)中的地址,这个地址在以后链接时要被重定位。为了简单起见,编译器应该将每个节的首个 RVA 设置为 0
节	节是 PE 或 COFF 文件中代码或数据的基本单元。例如一个目标文件中所有代码可以被组合成单个节,或者(依赖于编译器的行为)每个函数独占一个节。增加节的数目会增加文件开销,但是链接器在链接代码时有更大的选择余地。节与 Intel 8086 体系结构中的段非常相似。一个节中的所有原始数据必须被加载到连续的内存中。另外,映像文件可能包含一些具有特殊用途的节,例如.tls 节或.reloc 节
VA	VA 虚拟地址。除了不减去映像文件的基址外,与 RVA 相同。这个地址之所以被称为“虚拟地址”是因为 Windows 为每个进程创建一个私有的虚拟地址空间,它独立于物理内存。无论出于何种目的,VA 都只应该被认为是一个地址。VA 并不能像 RVA 那样能够预先得到,因为加载器可能不把映像加载到它的首选位置上

7.1.2 总体结构

图 7.1 解释了 Microsoft COFF 目标文件格式。

7.1.3 COFF 文件头

在目标文件的开头,或者紧跟着映像文件签名之后,是一个如下格式的标准 COFF 文件头,其中各成员变量的说明见表 7.2。

```
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD   TimeDateStamp;
    DWORD   PointerToSymbolTable;
    DWORD   NumberOfSymbols;
```

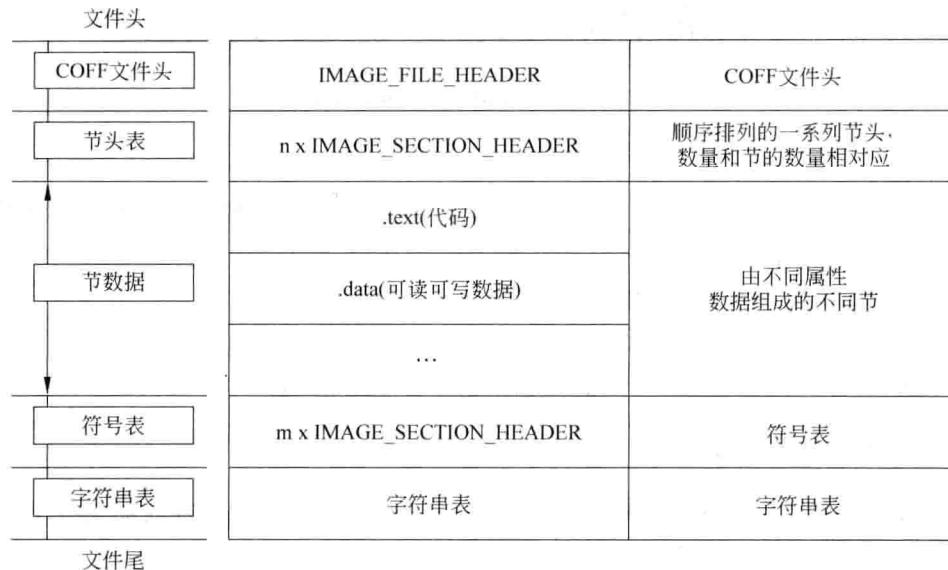


图 7.1 典型的 COFF 目标文件格式

```

WORD  SizeOfOptionalHeader;
WORD  Characteristics;
} IMAGE_FILE_HEADER, * PIMAGE_FILE_HEADER;

```

表 7.2 COFF 文件头成员变量说明

偏移	大小	域	描述
0	2	Machine	标识目标机器类型的数字。要获取更多信息，请参考 7.1.3.1 节“机器类型”
2	2	NumberOfSections	节的数目。它给出了节表的大小，而节表紧跟着文件头。注意 Windows 加载器限制节的最大数目为 96
4	4	TimeDateStamp	从 UTC 时间 1970 年 1 月 1 日 00:00 起的总秒数（一个 C 运行时 time_t 类型的值）的低 32 位，它指出文件何时被创建
8	4	PointerToSymbolTable	符号表的文件偏移。如果不存在 COFF 符号表，此值为 0。对于映像文件来说，此值应该为 0，因为已经不赞成使用 COFF 调试信息了
12	4	NumberOfSymbols	符号表中的元素数目。由于字符串表紧跟符号表，所以可以利用这个值来定位字符串表。对于映像文件来说，此值应该为 0，因为已经不赞成使用 COFF 调试信息了
16	2	SizeOfOptionalHeader	可选文件头的大小。可执行文件需要可选文件头而目标文件并不需要。对于目标文件来说，此值应该为 0。要获取可选文件头格式的详细描述，请参考 10.2.6 节“可选文件头（仅适用于映像文件）”
18	2	Characteristics	指示文件属性的标志。对于特定的标志，请参考 7.1.3.2 节“文件属性标志”

我们来解读一下 HelloWorld.obj 的文件头，文件头如图 7.2 所示。

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00000000h:	4C	01	08	00	00	00	00	00	00	01	00	00	08	00	00	00
00000010h:	00	00	00	00	2E	74	65	78	74	00	00	00	00	00	00	00

图 7.2 HelloWorld.obj (00h-1Fh)

其中：

- Machine=0x14c，从 7.1.3.1 节可以知道，它表示机器类型为 386 或后继处理器及其兼容处理器。
- NumberOfSections=8，代表文件中包含 8 个节。
- NumberOfSymbols=8，代表符号表中有 8 个符号条目。
- SizeOfOptionalHeader=0，代表此文件没有可选文件头。

7.1.3.1 机器类型

Machine 域可以取以下各值中的一个来指定 CPU 类型。映像文件仅能运行于指定处理器或者能够模拟指定处理器的系统上(见表 7.3)。

表 7.3 机器类型

常量	值	描述
IMAGE_FILE_MACHINE_UNKNOWN	0x0	适用于任何类型处理器
IMAGE_FILE_MACHINE_AM33	0x1d3	Matsushita AM33 处理器
IMAGE_FILE_MACHINE_AMD64	0x8664	x64 处理器
IMAGE_FILE_MACHINE_ARM	0x1c0	ARM 小尾处理器
IMAGE_FILE_MACHINE_EBC	0xebc	EFI 字节码处理器
IMAGE_FILE_MACHINE_I386	0x14c	386 或后继处理器及其兼容处理器
IMAGE_FILE_MACHINE_IA64	0x200	Itanium 处理器家族
IMAGE_FILE_MACHINE_M32R	0x9041	Mitsubishi M32R 小尾处理器
IMAGE_FILE_MACHINE_MIPS16	0x266	MIPS16 处理器
IMAGE_FILE_MACHINE_MIPSFPU	0x366	带 FPU 的 MIPS 处理器
IMAGE_FILE_MACHINE_MIPSFPU16	0x466	带 FPU 的 MIPS16 处理器
IMAGE_FILE_MACHINE_POWERPC	0x1f0	PowerPC 小尾处理器
IMAGE_FILE_MACHINE_POWERPCFP	0x1f1	带符点运算支持的 PowerPC 处理器
IMAGE_FILE_MACHINE_R4000	0x166	MIPS 小尾处理器
IMAGE_FILE_MACHINE_SH3	0x1a2	Hitachi SH3 处理器
IMAGE_FILE_MACHINE_SH3DSP	0x1a3	Hitachi SH3 DSP 处理器
IMAGE_FILE_MACHINE_SH4	0x1a6	Hitachi SH4 处理器
IMAGE_FILE_MACHINE_SH5	0x1a8	Hitachi SH5 处理器
IMAGE_FILE_MACHINE_THUMB	0x1c2	Thumb 处理器
IMAGE_FILE_MACHINE_WCEMIPSV2	0x169	MIPS 小尾 WCE v2 处理器

7.1.3.2 文件属性标志

Characteristics 域包含指示目标文件或映像文件属性的标志。当前定义了以下值，参见表 7.4。

表 7.4 文件属性标志

标 志	值	说 明
IMAGE_FILE_RELOCS_STRIPPED	0x0001	仅适用于映像文件,适用于Windows CE、Microsoft Windows NT_及其后继操作系统。它表明此文件不包含基址重定位信息,因此必须被加载到其首选基址上。如果地址不可用,加载器会报错。链接器默认会移除可执行(EXE)文件中的重定位信息
IMAGE_FILE_EXECUTABLE_IMAGE	0x0002	仅适用于映像文件,它表明此映像文件是合法的,可以被运行。如果未设置此标志,表明出现了链接器错误
IMAGE_FILE_LINE_NUMS_STRIPPED	0x0004	COFF行号信息已经被移除,不赞成使用此标志,它应该为0
IMAGE_FILE_LOCAL_SYMS_STRIPPED	0x0008	COFF符号表中有关局部符号的项已经被移除,不赞成使用此标志,它应该为0
IMAGE_FILE.Aggressive_WS_TRIM	0x0010	此标志已经被舍弃,它用于调整工作集。Windows 2000 及其后继操作系统不赞成使用此标志,它应该为0
IMAGE_FILE_LARGE_ADDRESS_AWARE	0x0020	应用程序可以处理大于2GB的地址
	0x0040	此标志保留供将来使用
IMAGE_FILE_BYTES_REVERSED_LO	0x0080	小尾:在内存中,最不重要位(LSB)在最重要位(MSB)前面。不赞成使用此标志,它应该为0
IMAGE_FILE_32BIT_MACHINE	0x0100	机器类型基于32位字体系结构
IMAGE_FILE_DEBUG_STRIPPED	0x0200	调试信息已经从此映像文件中移除
IMAGE_FILE_Removable_Run_From_SWAP	0x0400	如果此映像文件在可移动介质上,完全加载它并把它复制到交换文件中
IMAGE_FILE_Net_Run_From_SWAP	0x0800	如果此映像文件在网络介质上,完全加载它并把它复制到交换文件中
IMAGE_FILE_SYSTEM	0x1000	此映像文件是系统文件,而不是用户程序
IMAGE_FILE_DLL	0x2000	此映像文件是动态链接库(DLL)。这样的文件总被认为是可执行文件,尽管它们并不能直接被运行
IMAGE_FILE_UP_SYSTEM_ONLY	0x4000	此文件只能运行于单处理器机器上
IMAGE_FILE_BYTES_REVERSED_HI	0x8000	大尾:在内存中,MSB在LSB前面。不赞成使用此标志

7.1.4 节头表

节头表的每一行等效于一个节头。这个表紧跟可选文件头(如果存在)。之所以必须在这个位置是因为文件头中并没有一个直接指向节表的指针,节表的位置是通过计算文件头后的第一个字节的位置来得到的。一定要确保使用文件头中指定的可选文件头的大小来进行计算。节表中的元素数目由文件头中的 NumberOfSections 域给出,而元素的编号是从1开始的。表示代码节和数据节的元素的顺序由链接器决定。在映像文件中,每个节的 VA

值必须由链接器决定。这样能够保证这些节位置相邻且按升序排列，并且这些 VA 值必须是可选文件头中 SectionAlignment 域的倍数。每个节头(节表项)格式如下，共 40 个字节，其中各成员变量的说明见表 7.5。

```
#define IMAGE_SIZEOF_SHORT_NAME 8
typedef struct _IMAGE_SECTION_HEADER {
    BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD PhysicalAddress;
        DWORD VirtualSize;
    } Misc;
    DWORD VirtualAddress;
    DWORD SizeOfRawData;
    DWORD PointerToRawData;
    DWORD PointerToRelocations;
    DWORD PointerToLinenumbers;
    WORD NumberOfRelocations;
    WORD NumberOfLinenumbers;
    DWORD Characteristics;
} IMAGE_SECTION_HEADER, * PIMAGE_SECTION_HEADER;
```

表 7.5 节头成员变量说明

偏移	大小	域	说 明
0	8	Name	这是一个 8 字节的 UTF-8 编码的字符串，不足 8 字节时用 NULL 填充。如果它正好是 8 字节，那就没有最后的 NULL 字符。如果名称更长，这个域中是一个斜杠(/)后跟一个用 ASCII 码表示的十进制数，这个十进制数表示字符串表中的偏移。可执行映像不使用字符串表也不支持长度超过 8 字节的节名。如果目标文件中有长节名的节最后要出现在可执行文件中，那么相应的长节名会被截断
8	4	VirtualSize	当加载进内存时这个节的总大小。如果此值比 SizeOfRawData 大，那么多出的部分用 0 填充。这个域仅对可执行映像是合法的，对于目标文件来说，它应该为 0
12	4	VirtualAddress	对于可执行映像来说，这个域的值是这个节被加载进内存之后它的第一个字节相对于映像基址的偏移地址。对于目标文件来说，这个域的值是没有重定位之前其第一个字节的地址；为了简单起见，编译器应该把此值设置为 0；否则这个值是个任意值，但是在重定位时应该从偏移地址中减去这个值
16	4	SizeOfRawData	(对于目标文件来说)节的大小或者(对于映像文件来说)磁盘文件中已初始化数据的大小。对于可执行映像来说，它必须是可选文件头中 FileAlignment 域的倍数。如果它小于 VirtualSize 域的值，余下的部分用 0 填充。由于 SizeOfRawData 域要向上舍入，但是 VirtualSize 域并不舍入，因此可能出现 SizeOfRawData 域大于 VirtualSize 域的情况。当节中仅包含未初始化的数据时，这个域应该为 0

续表

偏移	大小	域	说 明
20	4	PointerToRawData	指向 COFF 文件中节的第一个页面的文件指针。对于可执行映像来说,它必须是可选文件头中 FileAlignment 域的倍数。对于目标文件来说,要获得最好的性能,此值应该按 4 字节边界对齐。当节中仅包含未初始化的数据时,这个域应该为 0
24	4	PointerToRelocations	指向节中重定位项开头的文件指针。对于可执行文件或者没有重定位项的文件来说,此值应该为 0
28	4	PointerToLinenumbers	指向节中行号项开头的文件指针。如果没有 COFF 行号信息,此值应该为 0。对于映像来说,此值应该为 0,因为已经不赞成使用 COFF 调试信息了
32	2	NumberOfRelocations	节中重定位项的个数。对于可执行映像来说,此值应该为 0
34	2	NumberOfLinenumbers	节中行号项的个数。对于映像来说,此值应该为 0,因为已经不赞成使用 COFF 调试信息了
36	4	Characteristics	描述节特征的标志。要获取更多信息,请参考 7.1.4.1 节“节标志”

节中已初始化的数据就是简单的字节块,但是对于那些仅包含 0 的节来说,节中的数据就没有必要包含到文件中。

每个节中的数据都位于节头的 PointerToRawData 域指定的文件偏移处,数据的大小由 SizeOfRawData 域给出。如果 SizeOfRawData 小于 VirtualSize,那么余下的部分用 0 填充。

在映像文件中,节中的数据必须按可选文件头的 FileAlignment 域指定的边界对齐。节中的数据必须按相应节的 RVA 值的大小顺序出现在文件中(节表中单个的节头也是如此)。

如果可选文件头的 SectionAlignment 域的值小于相应平台的页面大小,那么映像文件有一些附加的限制。对于这种文件,当映像被加载到内存中时,节中数据在文件中的位置必须与它在内存中的位置匹配,因此节中数据的物理偏移与 RVA 相同。

下面来解读一下 HelloWorld.obj 的节头表,节头表如图 7.3 所示。

00000010h:	00 00 00 00 2E 74 65 79 74 00 00 00 00 00 00 00 00 00 :	text.....
00000020h:	00 00 00 00 46 00 00 00 54 01 00 00 00 00 00 00 00 00 :	...F...T.....
00000030h:	00 00 00 00 00 00 00 00 20 00 00 20 2E 64 61 74 :dat
00000040h:	61 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 :	a.....
00000050h:	9A 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 :	3.....
00000060h:	40 00 00 C0 2E 72 64 61 74 61 00 00 00 00 00 00 00 00 :	0..?rdata.....
00000070h:	00 00 00 00 0E 00 00 00 9A 01 00 00 00 00 00 00 00 00 :?.....
00000080h:	00 00 00 00 00 00 00 00 40 00 00 40 2E 69 64 61 :0..0..ida
00000090h:	74 61 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 :	ta.....
000000a0h:	A8 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 :	?.....
000000b0h:	40 00 00 C0 2E 62 73 73 00 00 00 00 00 00 00 00 00 00 :	0..?bss.....
000000c0h:	00 00 00 00 00 00 00 00 A8 01 00 00 00 00 00 00 00 00 :?.....
000000d0h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 2E 72 65 6C :	e..?rel
000000e0h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 28 00 00 00 :{....}
000000f0h:	A8 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 :	?.....
00000100h:	00 08 00 40 2E 73 79 6D 74 61 62 00 00 00 00 00 00 00 :	..0..symtab.....
00000110h:	00 00 00 00 90 00 00 00 00 00 01 00 00 00 00 00 00 00 :?....
00000120h:	00 00 00 00 00 00 00 00 00 00 08 00 40 2E 73 74 72 :0..str
00000130h:	74 61 62 00 00 00 00 00 00 00 00 00 00 2B 00 00 00 :	tab.....+...
00000140h:	60 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 :
00000150h:	00 08 00 40 55 89 E5 81 EC 00 00 00 00 B8 00 00 :	..0u文俊....?

图 7.3 HelloWorld.obj (10h-15Fh)

可以看到,HelloWorld.obj 中含有 8 个节头,目标文件的节头我们重点关注 Name、SizeOfRawData、PointerToRawData、Characteristics 成员变量,各个节的这些成员变量值如表 7.6 所示。

表 7.6 HelloWorld.obj 节头表解读

Name 节名称	SizeOfRawData 节的大小	PointerToRawData 文件中偏移位置	Characteristics 节属性
.text	0x46	0x154	0x20000020
.data	0x0	0x19A	0xC0000040
.rdata	0x0E	0x19A	0x40000040
.idata	0x0	0x1A8	0xC0000040
.bss	0x0	0x1A8	0xC0000080
.rel	0x28	0x1A8	0x40000800
.symtab	0x90	0x1D0	0x40000800
.strtab	0x2B	0x260	0x40000800

7.1.4.1 节属性

节头中的 Characteristics 域指出了节的属性(见表 7.7)。

表 7.7 节属性

标 志	值	说 明
	0x00000000	保留供将来使用
	0x00000001	保留供将来使用
	0x00000002	保留供将来使用
	0x00000004	保留供将来使用
IMAGE_SCN_TYPE_NO_PAD	0x00000008	从这个节结尾到下一个边界之间不能填充。此标志被舍弃,它已经被 IMAGE_SCN_ALIGN_1BYTES 标志取代,此标志仅对目标文件合法
	0x00000010	保留供将来使用
IMAGE_SCN_CNT_CODE	0x00000020	此节包含可执行代码
IMAGE_SCN_CNT_INITIALIZED_DATA	0x00000040	此节包含已初始化的数据
IMAGE_SCN_CNT_UNINITIALIZED_DATA	0x00000080	此节包含未初始化的数据
IMAGE_SCN_LNK_OTHER	0x00000100	保留供将来使用
IMAGE_SCN_LNK_INFO	0x00000200	此节包含注释或者其他信息。.directve 节具有这种属性,此标志仅对目标文件合法
	0x00000400	保留供将来使用
IMAGE_SCN_LNK_REMOVE	0x00000800	此节不会成为最终形成的映像文件的一部分,此标志仅对目标文件合法

续表

标 志	值	说 明
IMAGE_SCN_LNK_COMDAT	0x00001000	此节包含 COMDAT 数据,此标志仅对目标文件合法
IMAGE_SCN_GPREL	0x00008000	此节包含通过全局指针(GP)来引用的数据
IMAGE_SCN_MEM_PURGEABLE	0x00020000	保留供将来使用
IMAGE_SCN_MEM_16BIT	0x00020000	保留供将来使用
IMAGE_SCN_MEM_LOCKED	0x00040000	保留供将来使用
IMAGE_SCN_MEM_PRELOAD	0x00080000	保留供将来使用
IMAGE_SCN_ALIGN_1BYTES	0x00100000	按 1 字节边界对齐数据,此标志仅对目标文件合法
IMAGE_SCN_ALIGN_2BYTES	0x00200000	按 2 字节边界对齐数据,此标志仅对目标文件合法
IMAGE_SCN_ALIGN_4BYTES	0x00300000	按 4 字节边界对齐数据,此标志仅对目标文件合法
IMAGE_SCN_ALIGN_8BYTES	0x00400000	按 8 字节边界对齐数据,此标志仅对目标文件合法
IMAGE_SCN_ALIGN_16BYTES	0x00500000	按 16 字节边界对齐数据,此标志仅对目标文件合法
IMAGE_SCN_ALIGN_32BYTES	0x00600000	按 32 字节边界对齐数据,此标志仅对目标文件合法
IMAGE_SCN_ALIGN_64BYTES	0x00700000	按 64 字节边界对齐数据,此标志仅对目标文件合法
IMAGE_SCN_ALIGN_128BYTES	0x00800000	按 128 字节边界对齐数据,此标志仅对目标文件合法
IMAGE_SCN_ALIGN_256BYTES	0x00900000	按 256 字节边界对齐数据,此标志仅对目标文件合法
IMAGE_SCN_ALIGN_512BYTES	0x00A00000	按 512 字节边界对齐数据,此标志仅对目标文件合法
IMAGE_SCN_ALIGN_1024BYTES	0x00B00000	按 1024 字节边界对齐数据,此标志仅对目标文件合法
IMAGE_SCN_ALIGN_2048BYTES	0x00C00000	按 2048 字节边界对齐数据,此标志仅对目标文件合法
IMAGE_SCN_ALIGN_4096BYTES	0x00D00000	按 4096 字节边界对齐数据,此标志仅对目标文件合法
IMAGE_SCN_ALIGN_8192BYTES	0x00E00000	按 8192 字节边界对齐数据,此标志仅对目标文件合法
IMAGE_SCN_LNK_NRELOC_OVFL	0x01000000	此节包含扩展的重定位信息
IMAGE_SCN_MEM_DISCARDABLE	0x02000000	此节可以在需要时被丢弃
IMAGE_SCN_MEM_NOT_CACHED	0x04000000	此节不能被缓存
IMAGE_SCN_MEM_NOT_PAGED	0x08000000	此节不能被交换到页面文件中

续表

标 志	值	说 明
IMAGE_SCN_MEM_SHARED	0x10000000	此节可以在内存中共享
IMAGE_SCN_MEM_EXECUTE	0x20000000	此节可以作为代码执行
IMAGE_SCN_MEM_READ	0x40000000	此节可读
IMAGE_SCN_MEM_WRITE	0x80000000	此节可写

7.1.4.2 特殊的节

典型的 COFF 节包含的是普通代码或数据,链接器与 Microsoft Win32 加载器并不需要知道其中的内容就可以处理它们。这些内容只与将要链接的或执行的应用程序有关。

但是目标文件与映像文件中的某些 COFF 节却有特殊含义。由于在节头中为这些节设置了特殊的标志,或者可选文件头中的某些域指向这些节,或者节名本身指出了节的特殊作用,所以工具和加载器可以识别它们。

表 7.8 描述了保留的节以及它们的属性,后面是对出现在可执行文件中的节以及包含元数据用于扩展目的的节的详细描述。

表 7.8 特殊的节

节 名	内 容	特 征
.bss	未初始化的数据(自由格式)	IMAGE_SCN_CNT_UNINITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_WRITE
.cormeta	CLR 元数据,它表明目标文件中包含托管代码	IMAGE_SCN_LNK_INFO
.data	已初始化的数据(自由格式)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_WRITE
.debug \$ F	生成的 FPO 调试信息(仅适用于目标文件,仅用于 x86 平台,现已被舍弃)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_DISCARDABLE
.debug \$ P	预编译的调试类型信息(仅适用于目标文件)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_DISCARDABLE
.debug \$ S	调试符号信息(仅适用于目标文件)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_DISCARDABLE
.debug \$ T	调试类型信息(仅适用于目标文件)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_DISCARDABLE
.directive	链接器选项	IMAGE_SCN_LNK_INFO
.edata	导出表	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ

续表

节名	内 容	特征
.idata	导入表	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_WRITE
.idlsym	包含已注册的SEH(仅适用于映像文件),它们用以支持IDL属性	IMAGE_SCN_LNK_INFO
.pdata	异常信息	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ
.rdata	只读的已初始化数据	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ
.reloc	映像文件的重定位信息	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_DISCARDABLE
.rsrc	资源目录	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ
.sbss	与GP相关的未初始化数据(自由格式)	IMAGE_SCN_CNT_UNINITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_WRITE IMAGE_SCN_GPREL 其中,IMAGE_SCN_GPREL标志仅用于IA64平台,不能用于其他平台。此标志只能用于目标文件,当映像文件中出现这种类型的节时,一定不能设置这个标志
.sdata	与GP相关的已初始化数据(自由格式)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_WRITE IMAGE_SCN_GPREL 其中,IMAGE_SCN_GPREL标志仅用于IA64平台,不能用于其他平台。此标志只能用于目标文件,当映像文件中出现这种类型的节时,一定不能设置这个标志
.srdata	与GP相关的只读数据(自由格式)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_GPREL 其中,IMAGE_SCN_GPREL标志仅用于IA64平台,不能用于其他平台。此标志只能用于目标文件,当映像文件中出现这种类型的节时,一定不能设置这个标志
.sxdata	已注册的异常处理程序数据(自由格式,仅适用于目标文件,仅用于x86平台)	IMAGE_SCN_LNK_INFO 这个节中包含目标文件中的代码所涉及的所有异常处理程序在符号表中的索引。这些符号可以是IMAGE_SYM_UNDEFINED类型的符号,也可以是定义在那个模块中的符号
.text	可执行代码(自由格式)	IMAGE_SCN_CNT_CODE IMAGE_SCN_MEM_EXECUTE IMAGE_SCN_MEM_READ
.tls	线程局部存储(仅适用于目标文件)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_WRITE
.tls\$	线程局部存储(仅适用于目标文件)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_WRITE

续表

节名	内 容	特 征
. vsdata	与 GP 相关的已初始化数据 (自由格式, 仅适用于 ARM、SH4 和 Thumb 平台)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ IMAGE_SCN_MEM_WRITE
. xdata	异常信息(自由格式)	IMAGE_SCN_CNT_INITIALIZED_DATA IMAGE_SCN_MEM_READ

表 7.8 列出的一些节被标记为“仅适用于目标文件”或“仅适用于映像文件”，它们表示这些节的特殊含义只是分别对于目标文件或者映像文件来说的。标记为“仅适用于映像文件”的节仍然可以首先出现在目标文件中，最后再成为映像文件的一部分，但这个节对于链接器来说并没有特殊含义，它仅对于映像文件加载器才有特殊含义。

7.1.5 代码节内容

代码节存储了编译器生成的可执行代码，下面来看一下 HelloWorld. obj 代码节的内容（见图 7.4）。

```

00000010h: 00 00 00 00 2E 74 65 78 74 00 00 00 00 00 00 00 ; .....text.....
00000020h: 00 00 00 00 46 00 00 00 54 01 00 00 00 00 00 00 ; .....F...T....
00000030h: 00 00 00 00 00 00 00 00 20 00 00 20 2E 64 61 74 ; ..... . . .dat
00000040h: 61 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; a. .....
00000050h: 9A 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ?. .....
00000060h: 40 00 00 C0 2E 72 64 61 74 61 00 00 00 00 00 00 ; @..?rdata. .....
00000070h: 00 00 00 00 0E 00 00 00 9A 01 00 00 00 00 00 00 ; .....?.
00000080h: 00 00 00 00 00 00 00 00 40 00 00 40 2E 69 64 61 ; .....@..@.ida
00000090h: 74 61 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ta. .....
000000a0h: A8 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ?. .....
000000b0h: 40 00 00 C0 2E 62 73 73 00 00 00 00 00 00 00 00 ; @..?bss. .....
000000c0h: 00 00 00 00 00 00 00 00 A8 01 00 00 00 00 00 00 ; .....?.
000000d0h: 00 00 00 00 00 00 00 00 80 00 00 C0 2E 72 65 6C ; .....€..?rel
000000e0h: 00 00 00 00 00 00 00 00 00 00 00 00 28 00 00 00 ; .....(. .
000000f0h: A8 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ?. .....
00000100h: 00 08 00 40 2E 73 79 6D 74 61 62 00 00 00 00 00 ; ...@.symtab. .....
00000110h: 00 00 00 00 90 00 00 00 00 D0 01 00 00 00 00 00 00 ; ....?..?.
00000120h: 00 00 00 00 00 00 00 00 00 08 00 40 2E 73 74 72 ; .....@.str
00000130h: 74 61 62 00 00 00 00 00 00 00 00 00 00 00 00 00 ; tab. .....
00000140h: 60 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .
00000150h: 00 08 00 40 55 89 E5 81 EC 00 00 00 00 B8 00 00 ; ...@.友伎....?.
00000160h: 00 00 50 E8 FC FF FF B8 C4 04 B8 00 00 00 00 ; ..P様 趣.?.
00000170h: E9 00 00 00 00 B8 E5 5D C3 55 89 E5 81 EC 04 00 ; ?..嫡]胱友伎..
00000180h: 00 00 E8 FC FF FF B9 45 FC B8 45 FC 5D E8 FC ; ..様 境界E是
00000190h: FF FF FF B3 C4 D4 B8 E5 5D C3 48 65 6C 6C 6F 20 ; 塵.嫡]禽el

```

图 7.4 HelloWorld. obj(10h-19Fh)

这里看到 .text 节头中, PointerToRawData = 0x154, SizeOfRawData = 0x46, 即代码节内容是位于文件偏移位置 0x154, 连续的 0x46 个字节, 代码节的这些内容代表什么含义, 将在第 8 章讲完 Intel x86 机器语言后进行解读。

7.1.6 数据节与导入节内容

.data 节、.rdata 节和.bss 节都属于数据节, 从 7.1.4.2 节可知,.data 节存储可读可写需要初始化的数据,.rdata 节存储只读数据,.bss 节存储可读可写不需要进行初始化的数据,.idata 节是导入节, 存储导入数据。下面来看一下 HelloWorld. obj 数据节的内容(见图 7.5)。

这里看到 .data 节、.idata 节和.bss 节的 SizeOfRawData 都为 0, 即这些节中没有数据。只有.rdata 节中有数据, 该节头的 PointerToRawData=0x19A, SizeOfRawData=0xE, 即只

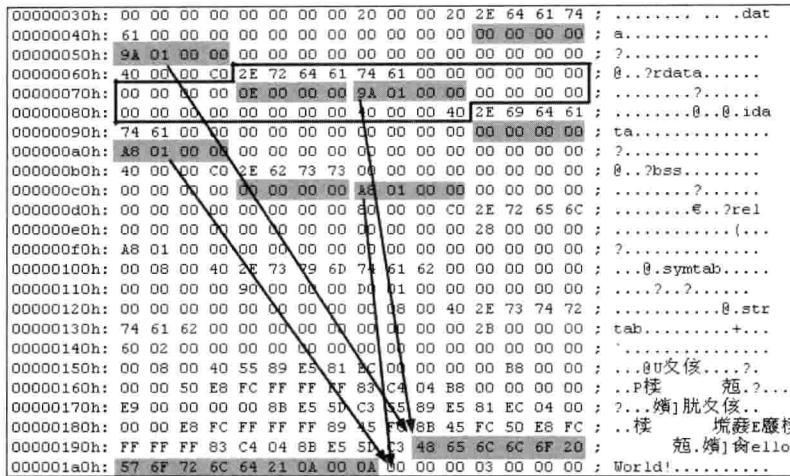


图 7.5 HelloWorld.obj(30h-1aFh)

读数据节是位于文件偏移位置 0x19A，连续的 0xE 个字节，其存储的具体内容是"Hello World!\n"常量字符串。

7.1.7 COFF 符号表

符号表的位置由 COFF 文件头给出。符号表是一个由记录组成的数组，每个记录长 18 字节。它们或者是标准符号表记录，或者是辅助符号表记录。标准符号表记录定义了一个符号或名称，格式如下：

```
typedef struct _IMAGE_SYMBOL {
    union {
        BYTE ShortName[8];
        struct {
            DWORD Short;           //if 0,use LongName
            DWORD Long;           //offset into string table
        } Name;
        PBYTE LongName[2];
    } N;
    DWORD Value;
    SHORT SectionNumber;
    WORD Type;
    BYTE StorageClass;
    BYTE NumberOfAuxSymbols;
} IMAGE_SYMBOL;
```

这里对符号表记录格式做一个小修改，按照 COFF 文件规范，当符号名小于长度 8 时，将符号名直接存在 ShortName 中，当大于 8 时，Short 字段为 0，Long 字段表示了符号名称在字符串表中的偏移位置。现在改为无论符号名称长度是否大于 8，符号表名称都放在字符串表中，Name 字段表示符号名称在字符串表中的偏移位置，新增加的 Next 字段用来保存哈希冲突链表。修改后的符号表记录格式如下，各成员变量说明见表 7.9。

```

typedef struct CoffSym
{
    DWORD Name;
    DWORD Next;
    DWORD Value;
    short SectionNumber;
    WORD Type;
    BYTE StorageClass;
    BYTE NumberOfAuxSymbols;
} CoffSym;

```

表 7.9 符号表记录成员变量说明

偏移	尺寸	域	说 明
0	4	Name(*)	符号名称,字符串表中的一个偏移地址
4	4	Next	用于保存哈希冲突链表
8	4	Value	Value 与 符号 相关的 值。其 意义 依 赖于 SectionNumber 和 StorageClass 这两个 域，它 通 常 表 示 可重 定位 的 地址
12	2	SectionNumber	这个 带 符号 整数 是 节 表 的 索引 (从 1 开始)，用 以 标识 定义 此 符号 的 节。一 些 值 有 特殊 的 含义，详 细 信 息 请 参 考 7.1.6.2 节 “SectionNumber 域 的 值”
14	2	Type	一 个 表 示 类型 的 数字。Microsoft 的 工具 将 它 设置 为 0x20 (如 果 是 函 数) 或者 0x0 (如 果 不是 函 数)。要 获取 更多 信 息，请 参 考 7.1.6.3 节 “类 型 表 示”
16	1	StorageClass	这 是 一 个 表 示 存储 类别 的 枚举 类型 值。要 获取 更多 信 息，请 参 考 7.1.6.4 节 “存 储 类别”
17	1	NumberOfAuxSymbols	跟 在 本 记 录 后 面 的 辅 助 符 号 表 项 的 个 数

SCC 编译器将 COFF 符号表存储在 .symtab 节中,下面来看一下 HelloWorld. obj 符号表的内容(见图 7.6)。

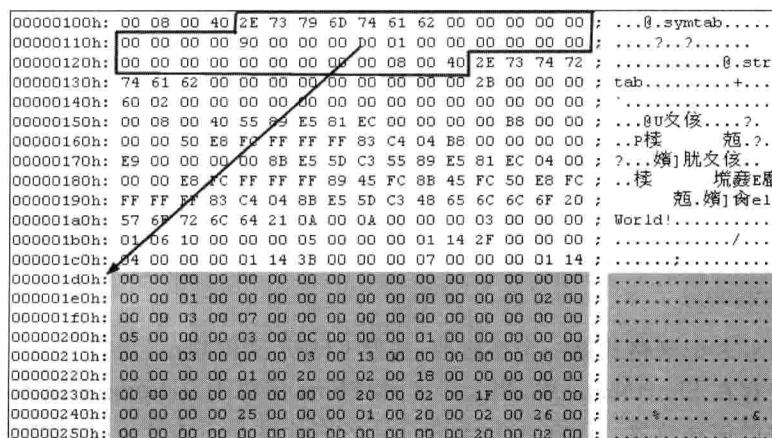


图 7.6 HelloWorld. obj(100h-25Fh)

这里看到.symtab节头中,PointerToRawData=0x1D0,SizeOfRawData=0x90,即符号表内容是位于文件偏移位置0x1D0,连续的0x90=144字节,符号个数=144÷sizeof(CoffSym)=144÷18=8,这8个符号是什么需要通过字符串表才能知道。

7.1.7.1 SectionNumber域的值

通常符号表项中的SectionNumber域是节表的索引(从1开始)。但是这个域是带符号整数,因此它可以为负值。下面这些小于1的值有特殊含义,见表7.10。

表7.10 SectionNumber域特殊取值含义

常量	值	说明
IMAGE_SYM_UNDEFINED	0	尚未为此符号记录分配一个节。这个零值表明引用了一个定义在其他地方的外部符号;而非零值则表明是一个普通符号,其大小由Value域给出
IMAGE_SYM_ABSOLUTE	-1	此符号是个绝对符号(不可重定位),并且不是地址
IMAGE_SYM_DEBUG	-2	此符号提供普通类型信息或者调试信息,但它并不对应于某一个节。Microsoft的工具将.file记录(存储类别为FILE)设置为这个值

7.1.7.2 类型表示

SCC编译器使用这个域来指出这个符号是不是函数,Type域只有0x0和0x20这两个值,其宏定义如下:

```
#define CST_FUNC      0x20          //函数
#define CST_NOTFUNC   0             //非函数
```

7.1.7.3 存储类别

符号表中的StorageClass域指出符号具体的存储类别。表7.11列出了所有可能的取值。注意StorageClass域是长度为1字节的无符号整数。因此如果这个域的值为-1,实际上应该被看作是与它相等的无符号数,也就是0xFF。表7.11中出现的Value都表示符号表记录中的Value域(它的意义依赖于存储类别的值)。

表7.11 存储类型取值

常量	值	说明以及对Value域的解释
IMAGE_SYM_CLASS_END_OF_FUNCTION	-1 (0xFF)	表示函数结尾的特殊符号,用于调试
IMAGE_SYM_CLASS_NULL	0	未被赋予存储类别
IMAGE_SYM_CLASS_AUTOMATIC	1	自动(堆栈)变量,Value域指出此变量在栈帧中的偏移

续表

常量	值	说明以及对 Value 域的解释
IMAGE_SYM_CLASS_EXTERNAL	2	Microsoft 的工具使用此值来表示外部符号。如果 SectionNumber 域为 0(IMAGE_SYM_UNDEFINED), 那么 Value 域给出大小; 如果 SectionNumber 域不为 0, 那么 Value 域给出节中的偏移
IMAGE_SYM_CLASS_STATIC	3	符号在节中的偏移。如果 Value 域为 0, 那么此符号表示节名
IMAGE_SYM_CLASS_REGISTER	4	寄存器变量, Value 域给出寄存器编号
IMAGE_SYM_CLASS_EXTERNAL_DEF	5	在外部定义的符号
IMAGE_SYM_CLASS_LABEL	6	模块中定义的代码标号, Value 域给出此符号在节中的偏移
IMAGE_SYM_CLASS_UNDEFINED_LABEL	7	引用的未定义的代码标号
IMAGE_SYM_CLASS_MEMBER_OF_STRUCT	8	结构体成员, Value 域指出是第几个成员
IMAGE_SYM_CLASS_ARGUMENT	9	函数的形式参数(形参)。Value 域指出是第几个参数
IMAGE_SYM_CLASS_STRUCT_TAG	10	结构体名
IMAGE_SYM_CLASS_MEMBER_OF_UNION	11	共用体成员, Value 域指出是第几个成员
IMAGE_SYM_CLASS_UNION_TAG	12	共用体名
IMAGE_SYM_CLASS_TYPE_DEFINITION	13	Typedef 项
IMAGE_SYM_CLASS_UNDEFINED_STATIC	14	静态数据声明
IMAGE_SYM_CLASS_ENUM_TAG	15	枚举类型名
IMAGE_SYM_CLASS_MEMBER_OF_ENUM	16	枚举类型成员, Value 域指出是第几个成员
IMAGE_SYM_CLASS_REGISTER_PARAM	17	寄存器参数
IMAGE_SYM_CLASS_BIT_FIELD	18	位域, Value 域指出是位域中的第几个位
IMAGE_SYM_CLASS_BLOCK	100	.bb(beginning of block, 块开头)或. eb 记录(end of block, 块结尾)。Value 域是代码位置, 它是一个可重定位的地址
IMAGE_SYM_CLASS_FUNCTION	101	Microsoft 的工具用此值来表示定义函数范围的符号记录, 这些符号记录分别是. bf(begin function, 函数开头)、. ef(end function, 函数结尾)以及. lf(lines in function, 函数中的行)。对于. lf 记录来说, Value 域给出了源代码中此函数所占的行数。对于. ef 记录来说, Value 域给出了函数代码的大小

续表

常量	值	说明以及对 Value 域的解释
IMAGE_SYM_CLASS_END_OF_STRUCT	102	结构体末尾
IMAGE_SYM_CLASS_FILE	103	Microsoft 的工具以及传统 COFF 格式都使用此值来表示源文件符号记录。这种符号表记录后面跟着给出文件名的辅助符号表记录
IMAGE_SYM_CLASS_SECTION	104	节的定义 (Microsoft 的工具使用 STATIC 存储类别代替)
IMAGE_SYM_CLASS_WEAK_EXTERNAL	105	弱外部符号
IMAGE_SYM_CLASS_CLR_TOKEN	107	表示 CLR 记号的符号,它的名称是这个记号的十六进制值的 ASCII 码表示

7.1.8 COFF 字符串表

COFF 符号表后紧跟的是 COFF 字符串表。它的位置可以通过将 COFF 文件头中符号表的地址加上符号总数乘以每个符号的大小得到。存储着以'\0'结尾的字符串,COFF 符号表中的符号指向这些字符串。下面来看一下 HelloWorld.obj 符号表的内容(见图 7.7)。

00000120h: 00 00 00 00 00 00 00 00 00 08 00 40 2E 73 74 72 ;8.str
00000130h: 74 61 62 00 00 00 00 00 00 00 00 00 2B 00 00 00 ; tab.....+.
00000140h: 60 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000150h: 00 08 00 40 55 89 E5 81 EC 00 00 00 00 B8 00 00 ; ...EU友俟....?
00000160h: 00 00 50 E8 FC FF FF F8 83 C4 04 B8 00 00 00 00 ; ..P様.....題.?.
00000170h: 29 00 00 00 00 8B E5 5D C3 55 89 E5 81 EC 04 00 ; ?...燒1臘友俟..
00000180h: 00 00 E8 FC FF FF F8 89 45 FC 8B 45 FC 50 E8 FC ; ..様 燒臘友俟.
00000190h: FF FF FF F8 83 C4 04 8B E5 5D C3 48 65 6C 6C 6F 20 ; 脑.燒1食el.
000001a0h: 57 6F 72 6C 64 21 0A 00 0A 00 00 00 03 00 00 00 ; World!.
000001b0h: 01 06 10 00 00 00 05 00 00 00 01 14 2F 00 00 00 ;/
000001c0h: 04 00 00 00 01 14 3B 00 00 00 07 00 00 00 01 14 ;;
000001d0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;;
000001e0h: 00 01 00 00 00 00 00 00 00 00 00 00 00 00 02 00 ;;
000001f0h: 00 03 00 07 00 00 00 00 00 00 00 00 00 00 00 00 ;;
00000200h: 05 00 00 00 03 00 00 00 00 00 00 01 00 00 00 00 ;;
00000210h: 00 03 00 00 00 03 00 03 13 00 00 00 00 00 00 00 ;;
00000220h: 00 00 00 01 00 20 00 00 00 00 18 00 00 00 00 00 ;;
00000230h: 00 00 00 00 00 00 00 00 20 00 02 00 1F 00 00 00 ;;
00000240h: 00 00 00 25 00 00 00 01 00 30 00 02 00 26 00 ;%;
00000250h: 00 00 00 00 00 00 00 00 00 00 00 00 20 00 02 00 ;;
00000260h: 00 21 64 61 14 61 00 2E 62 73 72 00 2E 72 01 61;data.bss.rda
00000270h: 74 61 00 00 61 69 6E 00 73 72 69 6E 74 66 00 5F ; ta.main.printf_.
00000280h: 65 6E 74 72 79 00 65 78 74 73 74 00 ; entry.exit.

图 7.7 HelloWorld.obj(12Fh-28Fh)

7.1.7 节可知 HelloWorld.obj 的符号表中共有 8 个符号条目，通过字符串表我们来看一下这 8 个符号分别是什么？第 1 个符号是空字符串，第 2 个符号是 ".data"，第 3 个符号是 ".bss"，第 4 个符号是 ".rdata"，第 5 个符号是 "main"，第 6 个符号是 "printf"，第 7 个符号是 "_entry"，第 8 个符号是 "exit"。

7.1.9 COFF 重定位信息

目标文件中包含 COFF 重定位信息，它用来指出当节中的数据被放进映像文件以及将来被加载进内存时应如何修改这些数据。

映像文件中并不包含 COFF 重定位信息,这是因为所有被引用的符号都已经被赋予了一个在平坦内存空间中的地址。映像文件中包含的重定位信息是位于 .reloc 节中的基址重定位信息(除非映像具有 IMAGE_FILE_RELOCS_STRIPPED 属性)。

对于目标文件中的每个节,都有一个由长度固定的记录组成的数组来保存此节的 COFF 重定位信息。此数组的位置和长度在节头中指定。数组的每个元素格式如下:

```
typedef struct _IMAGE_RELLOCATION {
    union {
        DWORD VirtualAddress;
        DWORD RelocCount;
    };
    DWORD SymbolTableIndex;
    WORD Type;
} IMAGE_RELLOCATION;
```

这里也对重定位记录格式做一个小修改,将 Type 这个成员变量的两个字节拆分为两个变量,一个字节保存需要重定位数据的节编号;另一个字节存储重定位类型。修改后的重定位信息记录格式如下,各成员变量说明见表 7.12。

```
typedef struct CoffReloc {
    DWORD offset;           // 需要进行重定位的代码或数据的地址
    DWORD cfsym;           // 符号表的索引(从 0 开始)
    BYTE section;           // 需要重定位数据的节编号
    BYTE type;              // 重定位类型
} CoffReloc;
```

表 7.12 重定位信息各成员变量说明

Offset	Size	Field	Description
0	4	offset	需要进行重定位的代码或数据的地址。这是从节开头算起的偏移,加上节的 RVA/Offset 域的值,参见 7.1.4 节。例如,如果节的第一个字节的地址是 0x10,那么第三个字节的地址就是 0x12
4	4	cfsym	符号表的索引(从 0 开始)。这个符号给出了用于重定位的地址。如果这个指定符号的存储类别为节,那么它的地址就是第一个与它同名的节的地址
8	1	section	需要重定位数据的节编号
9	1	type	重定位类型。合法的重定位类型依赖于机器类型,请参考 7.1.9.1 节“类型指示符”

SCC 编译器将 COFF 重定位信息存储在“.rel”节中,下面来看一下 HelloWorld.obj 重定位表的内容见图 7.8。

在“.rel”节头中,PointerToRawData=0x1A8,SizeOfRawData=0x28,即重定位表是位于文件偏移位置 0x1A8,连续的 0x28=40 字节,重定位条目个数=40 ÷ sizeof(CoffSym)=40 ÷ 10=4。

```

000000d0h: 00 00 00 00 00 00 00 00 80 00 00 CO 2E 72 65 6C ; .....€..?rel
000000e0h: 00 00 00 00 00 00 00 00 00 00 00 28 00 00 00 ; .....
000000f0h: 48 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ?.....
00000100h: 00 08 00 40 2E 73 79 6D 74 61 62 00 00 00 00 00 ; ...@.syntab....
00000110h: 00 00 00 00 90 00 00 00 D0 01 00 00 00 00 00 00 00 ; ....?.....
00000120h: 00 00 00 00 00 00 00 00 00 00 00 08 00 40 2E 73 74 72 ; .....@.str
00000130h: 74 61 62 00 00 00 00 00 00 00 00 00 00 00 2B 00 00 00 ; tab.....
00000140h: 60 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000150h: 00 08 00 40 55 89 E5 81 EC 00 00 00 00 B8 00 00 ; ...@.rva....?.
00000160h: 00 00 50 E8 FC FF FF 83 C4 04 B8 00 00 00 00 ; ..P棲 趕?.
00000170h: E9 00 00 00 00 BB E5 SD C3 55 89 E5 81 EC 04 00 ; ?..嫡]肮爻.....
00000180h: 00 00 E8 FC FF FF 89 45 FC 8B 45 FC 50 E8 FC ; ..棲 壢薪E層
00000190h: FF FF FF 83 C4 04 BB 45 SD C3 48 65 6C 6C 6F 20 ; 趕.嫡]龠el
000001a0h: 57 6F 72 6C 64 21 0A 00 DA 00 00 00 03 00 00 00 ; World!.....
000001b0h: 01 06 10 00 00 05 00 00 00 01 14 2F 00 00 00 00 ; .....
000001c0h: 04 00 00 00 01 14 3B 00 00 00 07 00 00 00 01 14 ; .....

```

图 7.8 HelloWorld.obj(d0h-1cfh)

类型指示符

重定位记录的 Type 域指出了重定位类型。对于每种类型的机器都定义了不同的重定位类型。为 Intel 386 及其兼容处理器定义了以下重定位类型指示符,见表 7.13。

表 7.13 重定位类型指示符取值

常量	值	说明
IMAGE_REL_I386_ABSOLUTE	0x0000	重定位被忽略
IMAGE_REL_I386_DIR16	0x0001	不支持
IMAGE_REL_I386_REL16	0x0002	不支持
IMAGE_REL_I386_DIR32	0x0006	重定位目标的 32 位 VA
IMAGE_REL_I386_DIR32NB	0x0007	重定位目标的 32 位 RVA
IMAGE_REL_I386_SEG12	0x0009	不支持
IMAGE_REL_I386_SECTION	0x000A	包含重定位目标的节的 16 位索引,用于支持调试信息
IMAGE_REL_I386_SECREL	0x000B	重定位目标相对于它所在节开头的 32 位偏移,用于支持调试信息和静态线程局部存储
IMAGE_REL_I386_TOKEN	0x000C	CLR 记号
IMAGE_REL_I386_SECREL7	0x000D	相对于重定位目标所在节地址的 7 位偏移
IMAGE_REL_I386_REL32	0x0014	重定位目标的 32 位相对偏移,用于支持 x86 的相对分支和 CALL 指令

7.2 生成 COFF 目标文件

学习完 COFF 目标文件结构,下面来看生成 COFF 目标文件的相关代码。代码较难理解的地方作者都做了注释,后面的代码主要用到了 7.1 节讲的内容,如果没有特殊需要说明的地方作者将不做过多解释,下面是本节内容阅读指南:

- (1) 生成节表,结合 7.1.4 节来阅读。

- (2) 生成符号表,结合 7.1.6 节~7.1.7 节来阅读。
- (3) 生成重定位信息,结合 7.1.8 节来阅读。
- (4) 生成目标文件,结合 7.1 节来阅读。
- (5) SCC 编译器生成的目标文件格式,如图 7.9 所示,这就是本节代码要实现的目标。



图 7.9 SCC 编译器生成的 COFF 文件格式

7.2.1 生成节表

7.2.1.1 节结构定义

```

/* 节结构定义 */
typedef struct Section
{
    int data_offset;           //当前数据偏移位置
    char * data;               //节数据
    int data_allocated;        //分配内存空间
    char index;                //节序号
    struct Section * link;    //关联的其他节,符号节关联字符串节
    int * hashtab;             //哈希表,只用于存储符号表
    IMAGE_SECTION_HEADER sh;   //节头
} Section;

```

7.2.1.2 新建节

```
*****
* 功能：          新建节
* name:          节名称
* Characteristics: 节属性
* 返回值：        新增加节
* nsec_image:    全局变量,映像文件中节个数
*****
Section * section_new(char * name,int Characteristics)
{
    Section * sec;
    int initsize=8;
    sec=mallocz(sizeof(Section));
    strcpy(sec->sh.Name,name);
    sec->sh.Characteristics=Characteristics;
    sec->index=sections.count +1;      //节表序号从1开始
    sec->data=mallocz(sizeof(char) * initsize);
    sec->data_allocated=initsize;
    if(!(Characteristics & IMAGE_SCN_LNK_REMOVE))
        nsec_image++;
    dynarray_add(&sections,sec);
    return sec;
}
```

如前所述,IMAGE_SCN_LNK_REMOVE 代表“此节不会成为最终形成的映像文件的一部分。此标志仅对目标文件合法”,通过这个标志可判断该节是否需要放入映像文件中,如果需要放入映像文件,nsec_image 自动加 1,在所有节新建完成后, nsec_image 变量记录了映像文件的节个数。

7.2.1.3 给节数据预留空间

```
*****
* 功能：        给节数据预留至少 increment 大小的内存空间
* sec:          预留内存空间的节
* increment:   预留的空间大小
* 返回值：      预留内存空间的首地址
*****
void * section_ptr_add(Section * sec,int increment)
{
    int offset,offset1;
    offset=sec->data_offset;
    offset1=offset +increment;
    if(offset1>sec->data_allocated)
        section_realloc(sec,offset1);
```

```

    sec->data_offset=offset;
    return sec->data + offset;
}

/******************
 * 功能：      给节数据重新分配内存，并将内容初始化为 0
 * sec:       重新分配内存的节
 * new_size:   节数据新长度
*****************/
void section_realloc(Section * sec,int new_size)
{
    int size;
    char * data;

    size=sec->data_allocated;
    while(size<new_size)
        size=size * 2;
    data=realloc(sec->data,size);
    if(!data)
        error("内存分配失败");
    memset(data + sec->data_allocated,0,size - sec->data_allocated);
                           /* 节数据初始化为 0 */
    sec->data=data;
    sec->data_allocated=size;
}
}

```

7.2.2 生成符号表

7.2.2.1 创建 COFF 符号节

```

/******************
 * 功能：      新建存储 COFF 符号表的节
 * symtab:     COFF 符号表名
 * Characteristics: 节属性
 * strtab_name: 与符号表相关的字符串表
 * 返回值：     存储 COFF 符号表的节
*****************/
Section * new_coffsym_section(char * symtab_name,int Characteristics,
                               char * strtab_name)
{
    Section * sec;
    sec=section_new(symtab_name,Characteristics);
    sec->link=section_new(strtab_name,Characteristics);
    sec->hashtab=malloc(sizeof(int) * MAXKEY);
}
```

```

    return sec;
}

```

每个符号节要有其相应的字符串节,字符串节负责保存符号名称,可以看到Section结构的link字段记录了符号表关联的字符串节。

7.2.2.2 增加COFF符号

```

/****************************************************************************
 * 功能:      增加 COFF 符号
 * symtab:    保存 COFF 符号表的节
 * name:      符号名称
 * val:       与符号相关的值
 * sec_index: 定义此符号的节
 * type:      Coff 符号类型
 * StorageClass: Coff 符号存储类别
 * 返回值:    符号 COFF 符号表中序号
 ****
 int coffsym_add(Section * symtab, char * name, int val, int sec_index,
                  short type, char StorageClass)
{
    CoffSym * cfsym;
    int cs, keyno;
    char * csname;
    Section * strtab=symtab->link;
    int * hashtab;
    hashtab=symtab->hashtab;
    cs=coffsym_search(symtab, name);
    if(cs==0)
    {
        cfsym=section_ptr_add(symtab, sizeof(CoffSym));
        csname=coffstr_add(strtab, name);
        cfsym->Name=csname - strtab->data;
        cfsym->Value=val;
        cfsym->Section=sec_index;
        cfsym->Type=type;
        cfsym->StorageClass=StorageClass;
        cfsym->Value=val;
        keyno=elf_hash(name);
        cfsym->Next=hashtab[keyno];
    }
    cs=cfsym - (CoffSym * )symtab->data;
    hashtab[keyno]=cs;
}
return cs;
}

```

```

/****************************************************************************
 * 功能：      增加或更新 COFF 符号,更新只适用于函数先声明后定义的情况
 * s:          符号指针
 * val:        符号值
 * sec_index: 定义此符号的节
 * type:       COFF 符号类型
 * StorageClass: COFF 符号存储类别
****************************************************************************/
void coffsym_add_update(Symbol * s,int val,int sec_index,
                        short type,char StorageClass)
{
    char * name;
    CoffSym * cfsym;
    if(!s->c)
    {
        name= ((TkWord *)tktable.data[s->v])->spelling;
        s->c=coffsym_add(sec_symtab,name,val,sec_index,type,StorageClass);
    }
    else           //函数先声明后定义
    {
        cfsym=&((CoffSym *)sec_symtab->data)[s->c];
        cfsym->Value=val;
        cfsym->Section=sec_index;
    }
}

/****************************************************************************
 * 功能：      增加 COFF 符号名字符串
 * strtab:     保存 COFF 字符串表的节
 * name:       符号名称字符串
 * 返回值：    新增 COFF 字符串
****************************************************************************/
char * coffstr_add(Section * strtab,char * name)
{
    int len;
    char * pstr;
    len=strlen(name);
    pstr=section_ptr_add(strtab,len+1);
    memcpy(pstr,name,len);
    return pstr;
}

```

下面通过表 7.14,来理解一下上面这段代码。

表 7.14 HelloWorld.obj 中符号表及字符串表

表名称	数据内容(十六进制)																	数据代表的 ASCII 码字符	
	Name		Next		Value		...												
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
.symtab(符号表)	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	01	00	00	00	00	00	00	00	00	00	00	02	00	00	00	03	00	00	
	07	00	00	00	00	00	00	00	00	00	00	05	00	00	00	03	00	00	
	0C	00	00	00	01	00	00	00	00	00	00	03	00	00	00	03	00	00	
	13	00	00	00	00	00	00	00	00	00	00	01	00	20	00	02	00	00	
	18	00	00	00	00	00	00	00	00	00	00	00	00	00	20	00	02	00	
	1F	00	00	00	00	00	00	25	00	00	00	01	00	20	00	02	00	00	
	26	00	00	00	00	00	00	00	00	00	00	00	00	00	20	00	02	00	
.strtab(字符串表)	00	2E	64	61	74	61	00	2E	62	73	73	00	2E	72	64	61	74	61	.. data.. bss.. rdata.
	00	6D	61	69	6E	00	70	72	69	6E	74	66	00	5F	65	6E	74	72	main, printf, _entry, exit.
	79	00	65	78	69	74	00												

可以看到, 符号中共有 8 个条目, 第一个条目是空条目, 第二个条目, Name 字段值为 1, 从字符串表中可以看到偏移位置为 1 的字符串是 ".data", 第三个条目, Name 字段值为 7, 查得符号名是 ".bss", 第三个条目, Name 字段值为 C, 转换为十进制为 12, 查得符号名是 ".rdata", 另外注意到第四个条目的 Next 字段为 1, 说明与第一个符号名称出现了哈希冲突。其他 Next 都为 0, 而符号表中第 0 条为空条目, 表示没有出现哈希冲突, 这就是为什么符号表的开头要放个空条目的原因。

7.2.2.3 查找 COFF 符号

```
/*
 * 功能:      查找 COFF 符号
 * symtab:    保存 COFF 符号表的节
 * name:      符号名称
 * 返回值:    符号 COFF 符号表中序号
 */
int coffsym_search(Section * symtab, char * name)
{
    CoffSym * cfsym;
    int cs, keyno;
    char * csname;
    Section * strtab;

    keyno=elf_hash(name);
    strtab=symtab->link;
    cs=symtab->hashtab[keyno];
    while(cs)
    {
        cfsym=(CoffSym *)symtab->data+cs;
```

```

        csname=strtab->data +cfsym->Name;
        if(!strcmp(name,csname))
            return cs;
        cs=cfsym->Next;
    }
    return cs;
}

```

7.2.3 生成重定位信息

```

/******************
 * 功能： 增加重定位条目
 * section: 符号所在节
 * sym: 符号指针
 * offset: 需要进行重定位的代码或数据在其相应节的偏移位置
 * type: 重定位类型
*****************/
void coffreloc_add(Section * sec,Symbol * sym,int offset,char type)
{
    int cfsym;
    char * name;
    if(!sym->c)
        coffsym_add_update(sym,0,IMAGE_SYM_UNDEFINED,CST_FUNC,
                           IMAGE_SYM_CLASS_EXTERNAL);
    name=((TkWord *)tktable.data[sym->v])->spelling;
    cfsym=coffsym_search(sec_syntab,name);
    coffreloc_direct_add(offset,cfsym,sec->index,type);
}

/******************
 * 功能： 增加 COFF 重定位信息
 * offset: 需要进行重定位的代码或数据在其相应节的偏移位置
 * cfsym: 符号表的索引
 * section: 符号所在节
 * type: 重定位类型
*****************/
void coffreloc_direct_add(int offset,int cfsym,char section,char type)
{
    CoffReloc * rel;
    rel=section_ptr_add(sec_rel,sizeof(CoffReloc));
    rel->offset=offset;
    rel->cfsym=cfsym;
    rel->section=section;
    rel->type=type;
}

```

7.2.4 生成目标文件

7.2.4.1 初始化程序

```
*****  
* 功能：COFF 初始化  
* 本函数用到全局变量：  
DynArray sections;           //节数组  
Section * sec_text,          //代码节  
        * sec_data,           //数据节  
        * sec_bss,            //未初始化数据节  
        * sec_idata,          //导入表节  
        * sec_rdata,          //只读数据节  
        * sec_rel,             //重定位信息节  
        * sec_symtab,          //符号表节  
        * sec_dynsymtab;       //链接库符号节  
int nsec_image;              //映像文件节个数  
*****  
void init_coff()  
{  
    dynarray_init(&sections,8);  
    nsec_image=0;  
  
    sec_text=section_new(".text",  
                         IMAGE_SCN_MEM_EXECUTE|IMAGE_SCN_CNT_CODE);  
    sec_data=section_new(".data",  
                         IMAGE_SCN_MEM_READ|IMAGE_SCN_MEM_WRITE |  
                         IMAGE_SCN_CNT_INITIALIZED_DATA);  
  
    sec_rdata=section_new(".rdata",  
                         IMAGE_SCN_MEM_READ|IMAGE_SCN_CNT_INITIALIZED_DATA);  
    sec_idata=section_new(".idata",  
                         IMAGE_SCN_MEM_READ|IMAGE_SCN_MEM_WRITE |  
                         IMAGE_SCN_CNT_INITIALIZED_DATA);  
    sec_bss=section_new(".bss",  
                         IMAGE_SCN_MEM_READ|IMAGE_SCN_MEM_WRITE |  
                         IMAGE_SCN_CNT_UNINITIALIZED_DATA);  
    sec_rel=section_new(".rel",  
                         IMAGE_SCN_LNK_REMOVE|IMAGE_SCN_MEM_READ);  
  
    sec_symtab=new_coffsym_section(".symtab",  
                                   IMAGE_SCN_LNK_REMOVE|IMAGE_SCN_MEM_READ,".strtab");  
    sec_dynsymtab=new_coffsym_section(".dynsym",
```

```

        IMAGE_SCN_LNK_REMOVE|IMAGE_SCN_MEM_READ, ".dynstr");

coffsym_add(sec_symtab, "", 0, 0, 0, IMAGE_SYM_CLASS_NULL);
coffsym_add(sec_symtab, ".data", 0, sec_data->index, 0, IMAGE_SYM_CLASS_STATIC);
coffsym_add(sec_symtab, ".bss", 0, sec_bss->index, 0, IMAGE_SYM_CLASS_STATIC);
coffsym_add(sec_symtab, ".rdata", 0, sec_rdata->index, 0, IMAGE_SYM_CLASS_STATIC);
coffsym_add(sec_dynsymtab, "", 0, 0, 0, IMAGE_SYM_CLASS_NULL);
}

}

```

7.2.4.2 收尾清理程序

```

/******************
* 功能：释放所有节数据
*****************/
void free_sections()
{
    int i;
    Section * sec;
    for(i=0;i<sections.count;i++)
    {
        sec=(Section *)sections.data[i];
        if(sec->hashtab !=NULL)
            free(sec->hashtab);
        free(sec->data);
    }
    dynarray_free(&sections);
}

```

7.2.4.3 输出目标文件

```

/******************
* 功能：输出目标文件
* name: 目标文件名
*****************/
void write_obj(char * name)
{
    int file_offset;
    FILE * fout=fopen(name,"wb");
    int i,sh_size,nsec_obj=0;
    IMAGE_FILE_HEADER * fh;

    nsec_obj=sections.count - 2;
    sh_size=sizeof(IMAGE_SECTION_HEADER);
    file_offset=sizeof(IMAGE_FILE_HEADER)+nsec_obj * sh_size;
    fpad(fout,file_offset);
}

```

```

fh=malloc(sizeof(IMAGE_FILE_HEADER));
for(i=0;i<nsec_obj;i++)
{
    Section * sec=(Section *)sections.data[i];
    if(sec->data==NULL)continue;
    fwrite(sec->data,1,sec->data_offset,fout);
    sec->sh.PointerToRawData=file_offset;
    sec->sh.SizeOfRawData=sec->data_offset;
    file_offset+=sec->data_offset;
}
fseek(fout,SEEK_SET,0);
fh->Machine=IMAGE_FILE_MACHINE_I386;
fh->NumberOfSections=nsec_obj;
fh->PointerToSymbolTable=sec_syntab->sh.PointerToRawData;
fh->NumberOfSymbols=sec_syntab->sh.SizeOfRawData/sizeof(CoffSym);
fwrite(fh,1,sizeof(IMAGE_FILE_HEADER),fout);
for(i=0;i<nsec_obj;i++)
{
    Section * sec=(Section *)sections.data[i];
    fwrite(sec->sh.Name,1,sh_size,fout);
}

free(fh);
fclose(fout);
}

/******************
* 功能： 从当前读写位置到 new_pos 位置用 0 填补文件内容
* fp： 文件指针
* new_pos： 填补终点位置
*****************/
void fpad(FILE * fp,int new_pos)
{
    int curpos=fteill(fp);
    while(++curpos<=new_pos)
        fputc(0,fp);
}

```

7.3 成果展示

本章 7.1 节, 我们已经解读过 HelloWorld.obj 各个组成部分, 由于是肢解开来讲的, 大家可能对这个文件内容没有建立起整体的概念。本章的成果展示, 我们来看一下 HelloWorld.obj 的完整解读如图 7.10 所示, 这也是本章代码的最终输出成果。

IMAGE_FILE_HEADER NumberOfSections = 8	4C 01 08 00 00 00 00 00 D0 01 00 00 08 00 00 00 00 00 00 00	L.....
.text节头 SizeOfRawData= 0x46 = 70	2E 74 65 78 74 00 00 00 00 00 00 00 00 00 00 00 00 00 46 00 00 00 54 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 20	.text..... F...T.....
.data节头 SizeOfRawData= 0x0 = 0	2E 64 61 74 61 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 9A 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0	.data.....?..... . @..
.rdata节头 SizeOfRawData= 0xE = 14	2E 72 64 61 74 61 00 00 00 00 00 00 00 00 00 00 00 00 0E 00 00 00 9A 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40	.rdata.....?..... . @..@
.idata节头 SizeOfRawData= 0x0 = 0	2E 69 64 61 74 61 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 A8 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0	.idata.....?..... . @..
.bss节头 SizeOfRawData= 0x0 = 0	2E 62 73 73 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 A8 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 C0	.bss.....?..... €..
.rel节头 SizeOfRawData= 0x28 = 40	2E 72 65 6C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 28 00 00 00 A8 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 08 00 40	.rel.....(.....@.....)
.symtab节头 SizeOfRawData= 0x90 = 144	2E 73 79 6D 74 61 62 00 00 00 00 00 00 00 00 00 00 00 90 00 00 00 D0 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 08 00 40	.symtab.....?.....@.....
.strtab节头 SizeOfRawData= 0x2B = 43	2E 73 74 72 74 61 62 00 00 00 00 00 00 00 00 00 00 00 2B 00 00 00 60 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 08 00 40	.strtab..... +...@.....
.text (代码) 节内容	55 89 E5 81 EC 00 00 00 00 B8 00 00 00 00 50 E8 FC FF FF 83 C4 04 B8 00 00 00 E9 00 00 00 00 8B E5 5D C3 55 89 E5 81 EC 04 00 00 E8 FC FF FF FF 89 45 FC 8B 45 FC 50 E8 FC FF FF 83 C4 04 8B E5 5D C3
.data (可读可写数据) 节内容		
.rdata (只读数据) 节内容	48 65 6C 6C 6F 20 57 6F 72 6C 64 21 0A 00	Hello World!
.idata (导入表) 节内容		
.bss (未初始化数据) 节内容		
.rel (重定位表) 节内容 sizeof(CoffReloc)*4 = 10*4 = 40	0A 00 00 00 03 00 00 00 01 06 10 00 00 00 05 00 00 00 01 14 2F 00 00 00 04 00 00 00 00 14 3B 00 00 00 07 00 00 00 00 14
.symtab (符号表) 节内容 sizeof(CoffSym)*8 = 18*8 = 144	00 01 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00 03 00 07 00 00 00 00 00 00 00 00 00 00 00 00 05 00 00 00 03 00 0C 00 00 00 01 00 00 00 00 00 00 00 00 03 00 00 00 03 00 13 00 00 00 00 00 00 00 00 00 00 00 00 01 00 20 00 02 00 18 00 00 00 00 00 00 00 00 00 00 00 00 00 00 20 00 02 00 1F 00 00 00 00 00 00 00 00 25 00 00 00 01 00 20 00 02 00 26 00 00 00 00 00 00 00 00 00 00 00 00 00 00 20 00 02 00
.strtab (字符串表) 节内容	00 2E 64 61 74 61 00 2E 62 73 73 00 2E 72 64 61 74 61 00 6D 61 69 6E 00 70 72 69 6E 74 66 00 5F 65 6E 74 72 79 00 65 78 69 74 00	..data..bss..r data.main.p rintf_.entry. exit.

图 7.10 HelloWorld.obj 文件结构解读图

第 8 章

x86 机器语言

不积跬步，无以至千里；不积小流，无以成江海。

——荀子

如果不学习 x86 目标语言就开始语义分析，就像一个从来没有学过英语的人去当英语翻译，拿着英语辞典及语法书现学现卖，最后翻译的结果可想而知，肯定是词不达意。本章介绍 SCC 编译器生成的目标语言——x86 机器语言，在讲 x86 机器语言之前，先熟悉机器语言、指令系统的概念。

机器语言是用二进制代码表示的计算机能直接识别和执行的一种机器指令系统的集合。它是计算机的设计者通过计算机的硬件结构赋予计算机的操作功能。机器语言具有灵活、直接执行和速度快等特点。一条指令就是机器语言的一个语句，它是一组有意义的二进制代码。指令的基本格式包括操作码字段和地址码字段，其中操作码指明了指令的操作性质及功能，地址码则给出了操作数或操作数的地址。

指令系统是计算机硬件的语言系统，也称机器语言，它是软件和硬件的主要界面，从系统结构的角度看，它是系统程序员看到的计算机的主要属性。因此指令系统表征了计算机的基本功能，也决定了指令的格式和机器的结构。

从上面机器语言及指令系统的定义来看，二者完全是同一概念的不同称谓而已，所以下文中读者对这两个概念不必加以区分。

8.1 x86 机器语言简介

x86 或 80x86 是 Intel 公司首先开发制造的一种微处理器体系结构的泛称，该系列较早期的处理器名称是以数字来表示，并以 86 作为结尾，包括 Intel 8086、80186、80286、80386 以及 80486，因此其架构称为“x86 架构”。

x86 架构在 1978 年推出的 Intel 8086 中央处理器中首度出现，它是从 Intel 8008 处理器中发展而来的，而 8008 则是发展自 Intel 4004 的。8086 在三年后为 IBM PC 所选用，之后 x86 便成为了个人电脑的标准平台，成为了历来最成功的 CPU 架构。

其他公司也有制造 x86 架构的处理器，即 Cyrix(现为威盛电子所收购)、恩益禧集团、IBM、IDT 以及 Transmeta。Intel 公司以外最成功的制造商为 AMD，其早先产品 Athlon 系列处理器的市场份额仅次于 Intel Pentium。

8086 是 16 位处理器；直到 1985 年 32 位的 80386 的开发，这个架构都维持是 16 位。接着一系列的处理器表示了 32 位架构的细微改进，推出了数种扩充，直到 2003 年 AMD 对于这个架构发展了 64 位的扩充，并命名为 AMD64。后来英特尔公司也推出了与之兼容的

处理器，并命名为 Intel 64。两者一般统称为 x64，开创了 x86 的 64 位时代。

8.2 通用指令格式

Intel x86 通用指令格式如图 8.1 所示，可以看到通用指令格式分为以下 6 部分。

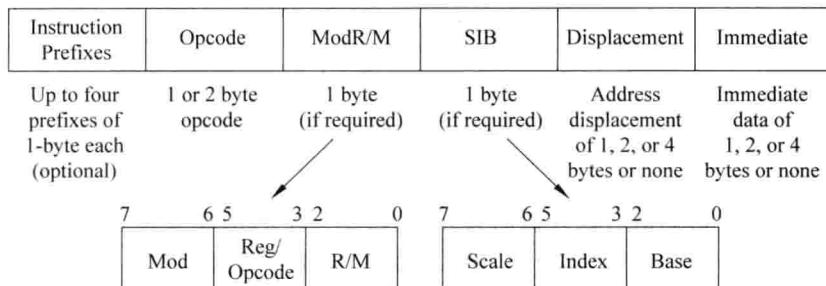


图 8.1 Intel x86 指令格式

- (1) Prefix(指令前缀)；
- (2) Opcode(操作码)；
- (3) ModR/M 字节；
- (4) SIB 字节；
- (5) Displacement(偏移量)；
- (6) Immediate(立即数)。

这 6 部分中，只有第 2 部分是必须的，其他都不是必须的。接下来将对指令的各个组成部分分别进行介绍。

8.2.1 指令前缀

指令前缀有 4 种，而且一条指令可以有多种前缀，每一个前缀占一个字节，在 32 位指令里，前缀种类的排列顺序不作规定。下面是它们的名称和机器码。

8.2.1.1 操作数长度前缀

对于 32 位指令系统而言，默认寄存器都是 32 位，但是我们不可避免地会使用其他长度的寄存器。如果要使用 16 位长度的寄存器，只需在指令前加 66H，即用操作数长度前缀标记。然而对于 8 位长度的寄存器，不是通过操作数前缀标记的。

举例：

```
mov eax,1 = B8 01000000
mov ax,1 = 66 B8 0100
mov al,1 = B0 01
```

另外还有一点要说明，就是像 movzx 这样，操作数与被操作数长度不相等的指令，则以被操作数的长度为标准，选择是否添加前缀。

举例：

```
movzx eax,ax    = 0FB7C0
movzx eax,al    = 0FB6C0
movzx ax,al     = 66 0FB6C0
```

8.2.1.2 地址长度前缀

这个前缀和上一个前缀用法差不多,只不过这个前缀是标记内存地址长度的。学过汇编语言的人,还应该知道操作数和被操作数中,至多一个是内存操作数。在32位指令系统下,如果内存操作数的长度为16位,则需在指令前加“地址长度前缀”,即67H。

举例:

```
mov eax,[bx]   = 67 8B07
mov eax,[ebx]  = 8B07
```

8.2.1.3 段超越前缀

当使用内存操作数时,无论那种内存操作数寻址都有默认的段寄存器,当内存操作数不使用默认段寄存器时,就需要使用段超越前缀。各个段寄存器代码如下:

- cs=2EH;
- ds=3EH;
- es=26H;
- fs=64H;
- gs=65H;
- ss=36H。

举例:

```
mov eax,[eax]      = 8B00
mov eax,cs:[eax]   = 2E 8B00
mov eax,es:[eax]   = 26 8600
mov eax,fs:[eax]   = 64 8600
mov eax,gs:[eax]   = 65 8600
mov eax,ss:[eax]   = 36 8600
```

8.2.1.4 重复前缀

进行串操作时,经常使用重复指令,该指令就是在机器码前加重复前缀,全部重复前缀编码如下:

- rep=F3;
- repe=F3;
- repz=F3;
- repne=F2;
- repnz=F2。

举例:

```
stosb      = AA
rep stosb  = F3 AA
```

8.2.1.5 锁定前缀

锁定前缀和协处理器有关，在指令前加 F0，表示访问独享。

举例：

```
add [eax].eax      = 3100
Lock add [eax],eax = F0 3100
```

8.2.2 操作码

操作码(Opcode)其实就是指令序列号，用来告诉CPU需要执行哪一条指令。指令系统的每一条指令都有一个操作码，它表示该指令应进行什么性质的操作。大多数通用指令的 Opcode 是单字节，最多是 2 字节，但是对有些浮点运算指令和 SSE 等 midea 指令来说是 3 字节的，Opcode 码代表指令是做什么操作。一些 Opcode 并不是完整的 Opcode 码，它需要 ModR/M 字节进行辅助。

这里请大家思考两个问题，一个操作码只对应一个机器指令吗？一个机器指令只对应一个操作码吗？下面通过表 8.1、表 8.2 来看一下。

表 8.1 操作码与机器指令对应关系

操作码	机器指令
0x90	NOP
0x90	XCHG AX, AX
0x90	XCHG EAX, EAX

表 8.2 机器指令与操作码对应关系

机器指令	操作码
ADD EAX, 1	0x83C001
ADD EAX, 1	0x0501000000
ADD EAX, 1	0x81C001000000

从表 8.1 中可以看出，同一个操作码可以对应多个机器指令。从表 8.2 中可以看出，同一个机器指令也可以对应多个操作码。

8.2.3 ModR/M 字节

某些操作码还需要 ModR/M 字节进行辅助，ModR/M 的结构分为 3 部分，各部分含义见表 8.3。

表 8.3 ModR/M 字节构成

部分名称	位域	功能
Mod 部分	ModRM [7:6]	与 R/M 一起组成 32 种可能的值—8 个寄存器加 24 种寻址模式
Reg/Opcode 部分	ModRM [5:3]	要么指定一个寄存器的值，要么指定 Opcode 中额外的 3 个比特的信息，具体作用在主操作码中指定
R/M 部分	ModRM [2:0]	可以指定一个寄存器作为操作数，或者和 mod 部分合起来表示一个寻址模式

表 8.4 为带 ModR/M 的 32 位寻址模式。

表 8.4 带 ModR/M 的 32 位寻址模式

r8(/r)		AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)		AX	CX	<td>BX</td> <td>SP</td> <td>BP</td> <td>SI</td> <td>DI</td>	BX	SP	BP	SI	DI
r32(/r)		EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)		MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)		XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
/digit(Opcode)	0	1	2	3	4	5	6	7	
REG=	000	001	010	011	100	101	110	111	
Effective Address	Mod	R/M	Value of ModR/M Byte(in Hexadecimal)						
[EAX]	00	000	00	08	10	18	20	28	30
[ECX]		001	01	09	11	19	21	29	31
[EDX]		010	02	0A	12	1A	22	2A	32
[EBX]		011	03	0B	13	1B	23	2B	33
[--][--] ¹	100	04	0C	14	1C	24	2C	34	3C
disp32 ²	101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36
[EDI]		111	07	0F	17	1F	27	2F	37
disp8[EAX] ³	01	000	40	48	50	58	60	68	70
disp8[ECX]		001	41	49	51	59	61	69	71
disp8[EDX]		010	42	4A	52	5A	62	6A	72
disp8[EBX]		011	43	4B	53	5B	63	6B	73
disp8[--][--]	100	44	4C	54	5C	64	6C	74	7C
disp8[EBP]		101	45	4D	55	5D	65	6D	75
disp8[ESI]		110	46	4E	56	5E	66	6E	76
disp8[EDI]		111	47	4F	57	5F	67	6F	77
disp32[EAX]	10	000	80	88	90	98	A0	A8	B0
disp32[ECX]		001	81	89	91	99	A1	A9	B1
disp32[EDX]		010	82	8A	92	9A	A2	AA	B2
disp32[EBX]		011	83	8B	93	9B	A3	AB	B3
disp32[--][--]	100	84	8C	94	9C	A4	AC	B4	BC
disp32[EBP]		101	85	8D	95	9D	A5	AD	B5
disp32[ESI]		110	86	8E	96	9E	A6	AE	B6
disp32[EDI]		111	87	8F	97	9F	A7	AF	B7
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0
ECX/CX/CL/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	FF

注：(1) [—][—] 表示 ModR/M 后跟 SIB 字节。

(2) "disp32" 表示 SIB 字节后跟随一个 32 位的偏移量，该偏移量被加至有效地址。

(3) "disp8" 表示 SIB 字节后跟随一个 8 位的偏移量，该偏移量将被符号扩展，然后被加至有效地址。

8.2.4 SIB 字节

SIB 字节包括以下 3 部分。

- scale 部分：指定 scale 因子。
- index 部分：指定索引寄存器的号码。
- base 部分：指定基址寄存器的号码这两个字节都是用来提供指令操作数，实现操作数的寻址。

表 8.5 为带 SIB 字节的 32 位寻址模式。

表 8.5 带 SIB 字节的 32 位寻址模式

r32		EAX	ECX	EDX	EBX	ESP	[*]	ESI	EDI
Base=	0	1	2	3	4	5	6	7	
Base=	000	001	010	011	100	101	110	111	
Scaled Index	SS	Index	Value of SIB Byte(in Hexadecimal)						
[EAX]	00	000	00	01	02	03	04	05	06
[ECX]		001	08	09	0A	0B	0C	0D	0E
[EDX]		010	10	11	12	13	14	15	16
[EBX]		011	18	19	1A	1B	1C	1D	1E
none		100	20	21	22	23	24	25	26
[EBP]		101	28	29	2A	2B	2C	2D	2E
[ESI]		110	30	31	32	33	34	35	36
[EDI]		111	38	39	3A	3B	3C	3D	3E
[EAX*2]	01	000	40	41	42	43	44	45	46
[ECX*2]		001	48	49	4A	4B	4C	4D	4E
[EDX*2]		010	50	51	52	53	54	55	56
[EBX*2]		011	58	59	5A	5B	5C	5D	5E
none		100	60	61	62	63	64	65	66
[EBP*2]		101	68	69	6A	6B	6C	6D	6E
[ESI*2]		110	70	71	72	73	74	75	76
[EDI*2]		111	78	79	7A	7B	7C	7D	7E
[EAX*4]	10	000	80	81	82	83	84	85	86
[ECX*4]		001	88	89	8A	8B	8C	8D	8E
[EDX*4]		010	90	91	92	93	94	95	96
[EBX*4]		011	98	99	9A	9B	9C	9D	9E
none		100	A0	A1	A2	A3	A4	A5	A6
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE
none		100	E0	E1	E2	E3	E4	E5	E6
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE

注：(1) “[*]”记号表示：若 MOD=00B 表示没有基址，且带有一个 32 位的偏移量；否则，表示 disp8 或 disp32+ [EBP]。即提供如下的寻址方式：

```

disp32[index]           (MOD=00)
disp8[EBP][index]       (MOD=01)
disp32[EBP][index]      (MOD=10)

```

8.2.5 偏移量与立即数

偏移量与立即数直接嵌在指令编码中。偏移量需要 ModRM 甚至 SIB 提供寻址，而立即数在 Opcode 里提供寻址。偏移量与立即数可为 1、2、4 个字节。

这里有必要讲一个概念——字节序，顾名思义就是字节的顺序，再多说两句就是大于一个字节类型的数据在内存中的存放顺序（一个字节的数据当然就无需谈顺序的问题了）。其实大部分人在实际的开发中都很少会直接和字节序打交道。但搞编译器开发字节序就是一个必须被考虑的问题。字节序分为两类：Big-Endian 和 Little-Endian，其定义如下：

- Little-Endian 就是低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。0x12345678，在内存的 Little-Endian 序列是 78 56 34 12。
- Big-Endian 就是高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。0x12345678，在内存的 Big-Endian 序列是 12 34 56 78。

x86 指令的编码在内存中是以 little endian 存储的，这主要体现在偏移量和立即数部分。在编码序列里，指令前缀在低端，依次往上，最后的立即数部分在最高端。

8.3 x86 寄存器

32 位 CPU 所含有的寄存器有：

- 4 个数据寄存器(EAX、EBX、ECX 和 EDX)。
- 2 个变址和指针寄存器(ESI 和 EDI)及 2 个指针寄存器(ESP 和 EBP)。
- 6 个段寄存器(ES、CS、SS、DS、FS 和 GS)。
- 1 个指令指针寄存器(EIP)和 1 个标志寄存器(EFlags)。

8.3.1 数据寄存器

数据寄存器主要用来保存操作数和运算结果等信息，从而节省读取操作数所需占用总线和访问存储器的时间。32 位 CPU 有 4 个 32 位的通用寄存器 EAX、EBX、ECX 和 EDX。对低 16 位数据的存取，不会影响高 16 位的数据。这些低 16 位寄存器分别命名为 AX、BX、CX 和 DX，它和先前的 CPU 中的寄存器相一致。4 个 16 位寄存器又可分割成 8 个独立的 8 位寄存器(AX:AH-AL、BX:BH-BL、CX:CH-CL、DX:DH-DL)，每个寄存器都有自己的名称，可独立存取。程序员可利用数据寄存器的这种“可分可合”的特性，灵活地处理字/字节的信息。

在 32 位 CPU 中，其 32 位寄存器 EAX、EBX、ECX 和 EDX 不仅可传送数据、暂存数据保存算术逻辑运算结果，而且也可作为指针寄存器，所以，这些 32 位寄存器更具有通用性。

8.3.2 变址寄存器

32 位 CPU 有两个 32 位通用寄存器 ESI 和 EDI。其低 16 位对应先前 CPU 中的 SI 和 DI，对低 16 位数据的存取，不影响高 16 位的数据。寄存器 ESI、EDI、SI 和 DI 称为变址寄存器(Index Register)，它们主要用于存放存储单元在段内的偏移量，用它们可实现多种存储器操作数的寻址方式，为以不同的地址形式访问存储单元提供方便。变址寄存器不可分

割成 8 位寄存器。作为通用寄存器,也可存储算术逻辑运算的操作数和运算结果。它们可作一般的存储器指针使用。在字符串操作指令的执行过程中,对它们有特定的要求,而且还具有特殊的功能。

8.3.3 指针寄存器

32 位 CPU 有两个 32 位通用寄存器 EBP 和 ESP。其低 16 位对应先前 CPU 中的 BP 和 SP,对低 16 位数据的存取,不影响高 16 位的数据。寄存器 EBP、ESP、BP 和 SP 称为指针寄存器(Pointer Register),主要用于存放堆栈内存储单元的偏移量,用它们可实现多种存储器操作数的寻址方式,为以不同的地址形式访问存储单元提供方便。指针寄存器不可分割成 8 位寄存器。作为通用寄存器,也可存储算术逻辑运算的操作数和运算结果。它们主要用于访问堆栈内的存储单元,并且规定:BP 为基指针(Base Pointer)寄存器,用它可直接存取堆栈中的数据;SP 为堆栈指针(Stack Pointer)寄存器,用它只可访问栈顶。

8.3.4 段寄存器

段寄存器是根据内存分段的管理模式而设置的。内存单元的物理地址由段寄存器的值和一个偏移量组合而成的,这样可用两个较少位数的值组合成一个可访问较大物理空间的内存地址。CPU 内部的段寄存器包括:

- CS——代码段寄存器(Code Segment Register),其值为代码段的段值;
- DS——数据段寄存器(Data Segment Register),其值为数据段的段值;
- ES——附加段寄存器(Extra Segment Register),其值为附加数据段的段值;
- SS——堆栈段寄存器(Stack Segment Register),其值为堆栈段的段值;
- FS——附加段寄存器(Extra Segment Register),其值为附加数据段的段值;
- GS——附加段寄存器(Extra Segment Register),其值为附加数据段的段值。

在 16 位 CPU 系统中,它只有 4 个段寄存器,所以,程序在任何时刻至多有 4 个正在使用的段可直接访问;在 32 位计算机系统中,它有 6 个段寄存器,所以,在此环境下开发的程序最多可同时访问 6 个段。32 位 CPU 有两个不同的工作方式:实方式和保护方式。在每种方式下,段寄存器的作用是不同的。有关规定简单描述如下。

- 实方式:前 4 个段寄存器 CS、DS、ES 和 SS 与先前 CPU 中的所对应的段寄存器的含义完全一致,内存单元的逻辑地址仍为“段值:偏移量”的形式。为访问某内存段内的数据,必须使用该段寄存器和存储单元的偏移量。
- 保护方式:在此方式下,情况要复杂得多,装入段寄存器的不再是段值,而是称为“选择子”(Selector)的某个值。

8.3.5 指令指针寄存器

32 位 CPU 把指令指针扩展到 32 位,并记作 EIP,EIP 的低 16 位与先前 CPU 中的 IP 作用相同。指令指针 EIP、IP(Instruction Pointer)是存放下次将要执行的指令在代码段的偏移量。在具有预取指令功能的系统中,下次要执行的指令通常已被预取到指令队列中,除非发生转移情况。所以,在理解它们的功能时,不考虑存在指令队列的情况。在实方式下,由于每个段的最大范围为 64KB,所以,EIP 中的高 16 位肯定都为 0,此时,相当于只用其低

16位的IP来反映程序中指令的执行次序。

8.3.6 标志寄存器

SCC编译器用到了标志寄存器的进位标志(Carry Flag,CF)、奇偶标志(Parity Flag,PF)、辅助进位标志(Auxiliary Carry Flag,AF)、零标志(Zero Flag,ZF)、符号标志(Sign Flag,SF)、溢出标志(Overflow Flag,OF)、方向标志(Direction Flag,DF),下面逐一介绍这些标志位的功能。

8.3.6.1 进位标志 CF

进位标志CF主要用来反映运算是否产生进位或借位。如果运算结果的最高位产生了一个进位或借位,那么,其值为1;否则,其值为0。使用该标志位的情况有多字(字节)数的加减运算、无符号数的大小比较运算、移位操作、字(字节)之间移位、专门改变CF值的指令等。

8.3.6.2 奇偶标志 PF

奇偶标志PF用于反映运算结果中1的个数的奇偶性。如果1的个数为偶数,则PF的值为1;否则,其值为0。利用PF可进行奇偶校验检查,或产生奇偶校验位。在数据传送过程中,为了提供传送的可靠性,如果采用奇偶校验的方法,就可使用该标志位。

8.3.6.3 辅助进位标志 AF

在发生下列情况时,辅助进位标志AF的值被置为1;否则,其值为0。

- (1) 在字操作时,发生低字节向高字节进位或借位时;
- (2) 在字节操作时,发生低4位向高4位进位或借位时。

对以上6个运算结果标志位,在一般编程情况下,标志位CF、ZF、SF和OF的使用频率较高,而标志位PF和AF的使用频率较低。

8.3.6.4 零标志 ZF

零标志ZF用来反映运算结果是否为0。如果运算结果为0,则其值为1;否则,其值为0。在判断运算结果是否为0时,可使用此标志位。

8.3.6.5 符号标志 SF

符号标志SF用来反映运算结果的符号位,它与运算结果的最高位相同。在计算机系统中,有符号数采用补码表示法,所以,SF也就反映运算结果的正负号。运算结果为正数时,SF的值为0;否则,其值为1。

8.3.6.6 溢出标志 OF

溢出标志OF用于反映有符号数加减运算所得结果是否溢出。如果运算结果超过当前运算位数所能表示的范围,则称为溢出,OF的值被置为1;否则,OF的值被清为0。

8.3.6.7 方向标志 DF

方向标志 DF 用来决定在串操作指令执行时有关指针寄存器发生调整的方向。

8.4 指令参考

80x86 指令系统,指令按功能可分为以下 7 部分。

- (1) 数据传送指令;
- (2) 算术运算指令;
- (3) 逻辑运算指令;
- (4) 控制转移指令;
- (5) 串操作指令;
- (6) 处理器控制指令;
- (7) 保护方式指令。

本节将对 SCC 编译器用到的指令逐一介绍,每个指令介绍分为操作码、指令形式、对标志位影响、应用举例 4 个部分。

8.4.1 符号说明

本节先对指令介绍中用到的一些符号说明一下。

8.4.1.1 操作码符号说明

- /digit——0~7 之间的数字,用于只使用寄存器/内存操作数的指令的 ModR/M 字节,作为指令扩展码。
- /r——表明指令的 ModR/M 字节既包含寄存器操作数,又包含内存操作数。
- cb——相对于下一条指令的字节偏移值。
- cw——相对于下一条指令的字偏移值。
- cd——相对于下一条指令的双字偏移值。
- cp——绝对远指针。
- ib、iw、id——ib 为 1 字节立即数;iw 为 2 字节立即数;id 为 4 字节立即数。
- +rb、+rw、+rd——寄存器编码,从 0~7,被加到操作数字节,各个寄存器编码如表 8.6 所示。

表 8.6 寄存器编码表

rb	rw	rd
AL = 0	AX = 0	EAX = 0
CL = 1	CX = 1	ECX = 1
DL = 2	DX = 2	EDX = 2
BL = 3	BX = 3	EBX = 3

续表

rb		rw		rd	
AH	=	4	SP	=	4
CH	=	5	BP	=	5
DH	=	6	SI	=	6
BH	=	7	DI	=	7

寄存器编码通过下面的枚举定义实现：

```
/* 寄存器编码 */
enum e_Register
{
    REG_EAX,
    REG_ECX,
    REG_EDX,
    REG_EBX,
    REG_ESP,
    REG_EBP,
    REG_ESI,
    REG_EDI,
    REG_ANY
};

#define REG_IRET REG_EAX //指定 EAX 为存放返回值的寄存器
```

8.4.1.2 汇编指令符号说明

- rel8——8位相对地址，范围为-128~127。
- rel16、rel32——同一段内的相对地址。
- ptr16:16、ptr16:32——远指针，典型应用于跨段指令。
- r8——8位通用寄存器。
- r16——16位通用寄存器。
- r32——32位通用寄存器。
- imm8——8位立即数。
- imm16——16位立即数。
- imm32——32位立即数。
- r/m8——8位通用寄存器或内存字节。
- r/m16——16位通用寄存器或内存字。
- r/m32——32位通用寄存器或内存双字。
- m——16位或32位内存操作数。
- m8——由DS:(E)SI或ES:(E)DI指向的内存字节。
- m16——由DS:(E)SI或ES:(E)DI指向的内存字。
- m32——由DS:(E)SI或ES:(E)DI指向的内存双字。

- m16:16,m16:32——包含两个数据的内存指针操作数,引号前面的是段地址,引号后面的是偏移地址。
- m16&32,m16&16,m32&32——包含数据对的内存操作数。
- moffs8,moffs16,moffs32——分别表示类型为字节、字、双字的内存变量。
- Sreg——段寄存器,ES=0, CS=1, SS=2, DS=3, FS=4, GS=5。
- m16int,m32int,m64int——字型、双字型、四字型内存整数操作数。

8.4.2 数据传送指令

SCC 编译器用到的数据传送指令有 MOV、MOVsx、PUSH、POP、LEA,下面逐一介绍(见表 8.7~表 8.11)。

8.4.2.1 MOV 指令

表 8.7 MOV 指令介绍

操作码	指令形式	标志位	说明	应用举例
88 /r	MOV r/m8,r8	不影响标志位 传送指令		MOV [00459AF0],AL
89 /r	MOV r/m16,r16			MOV [00459AF0],AX
89 /r	MOV r/m32,r32			MOV [00459AF0],EAX
8A /r	MOV r8,r/m8			MOV AL,[00459AF0]
8B /r	MOV r16,r/m16			MOV AX,[00459AF0]
8B /r	MOV r32,r/m32			MOV EAX,[00459AF0]
8C /r	MOV r/m16,Sreg			MOV AX,ES
8E /r	MOV Sreg,r/m16			MOV ES,AX
A0	MOV AL,moffs8			MOV AL,ES:[459A]
A1	MOV AX,moffs16			MOV AX,ES:[459A]
A1	MOV EAX,moffs32			MOV EAX,ES:[00459AF0]
A2	MOV moffs8,AL			MOV ES:[459A],AL
A3	MOV moffs16,AX			MOV ES:[459A],AX
A3	MOV moffs32,EAX			MOV ES:[00459AF0],EAX
B0+rb	MOV r8,imm8			MOV AL,F0
B8+rw	MOV r16,imm16			MOV AX,9AF0
B8+rd	MOV r32,imm32			MOV EAX,00459AF0
C6 /0	MOV r/m8,imm8			MOV BYTE PTR [00459AF0],F0
C7 /0	MOV r/m16,imm16			MOV WORD PTR [00459AF0],9AF0
C7 /0	MOV r/m32,imm32			MOV DWORD PTR [00459AF0],00459AF0

8.4.2.2 MOVSX 指令

表 8.8 MOSX 指令介绍

操作码	指令形式	标志位	说 明	应用举例
0F BE /r	MOVSX r16,r/m8	不影响 标志位	带符号扩展传送指令	MOVSX AX,BL
0F BE /r	MOVSX r32,r/m8			MOVSX EAX,BL
0F BF /r	MOVSX r32,r/m16			MOVSX EAX,BX

8.4.2.3 PUSH 指令

表 8.9 PUSH 指令介绍

操作码	指令形式	标志位	说 明	应用举例
FF /6	PUSH r/m16	不影响 标志位	16位数据压栈	PUSH WORD PTR [006387EA]
FF /6	PUSH r/m32		32位数据压栈	PUSH DWORD PTR [006387EA]
50+rw	PUSH r16		16位寄存器数据压栈	PUSH AX
50+rd	PUSH r32		32位寄存器数据压栈	PUSH EAX
6A	PUSH imm8		8位立即数据压栈	PUSH EA
68	PUSH imm16		16位立即数据压栈	PUSH 87EA
58	PUSH imm32		32位立即数据压栈	PUSH 006387EA
0E	PUSH CS		寄存器 CS 数据压栈	PUSH CS
16	PUSH SS		寄存器 SS 数据压栈	PUSH SS
1E	PUSH DS		寄存器 DS 数据压栈	PUSH DS
06	PUSH ES		寄存器 ES 数据压栈	PUSH ES
0F A0	PUSH FS		寄存器 FS 数据压栈	PUSH FS
0F A8	PUSH GS		寄存器 GS 数据压栈	PUSH GS

8.4.2.4 POP 指令

表 8.10 POP 指令介绍

操作码	指令形式	标志位	说 明	应用举例
8F /0	POP r/m16	不影响 标志位	16位数据出栈	POP WORD PTR [006387EA]
8F /0	POP r/m32		32位数据出栈	POP DWORD PTR [006387EA]
58+rw	POP r16		16位数据出栈到寄存器	POP AX
58+rd	POP r32		32位数据出栈到寄存器	POP EAX
1F	POP DS		数据出栈到寄存器 DS	POP DS
07	POP ES		数据出栈到寄存器 ES	POP ES
17	POP SS		数据出栈到寄存器 SS	POP SS
0F A1	POP FS		数据出栈到寄存器 FS	POP FS
0F A9	POP GS		32位数据出栈到寄存器 GS	POP GS

8.4.2.5 LEA 指令

表 8.11 LEA 指令介绍

操作码	指令形式	标志位	说 明	应用举例
8D /r	LEA r16,m	不影响	将源操作数的有效地址送 r16	
8F /0	LEA r32,m	标志位	将源操作数的有效地址送 r32	

8.4.3 算术运算指令

SCC 编译器用到的算术运算指令有 ADD、SUB、IMUL、IDIV、CMP、CWD、CDQ，下面逐一介绍(见表 8.12~表 8.18)。

8.4.3.1 ADD 指令

表 8.12 ADD 指令介绍

操作码	指令形式	标志位	说明	应用举例
04 ib	ADD AL, imm8	设置 AF CF OF SF PF ZF 加法	ADD AL, 1F	
05 iw	ADD AX, imm16		ADD AX, 4F80	
05 id	ADD EAX, imm32		ADD EAX, 00004F80	
80 /0 ib	ADD r/m8, imm8		ADD BYTE PTR [006387EA], 39	
81 /0 iw	ADD r/m16,imm16		ADD WORD PTR [006387EA], 1039	
81 /0 id	ADD r/m32,imm32		ADD DWORD PTR [006387EA], 00001039	
83 /0 ib	ADD r/m16,imm8		ADD WORD PTR [006387EA], 39	
83 /0 ib	ADD r/m32,imm8		ADD DWORD PTR [006387EA], 39	
00 /r	ADD r/m8,r8		ADD [006387EA], AL	
01 /r	ADD r/m16,r16		ADD [006387EA], AX	
01 /r	ADD r/m32,r32		ADD [006387EA], EAX	
02 /r	ADD r8,r/m8		ADD AL, [006387EA]	
03 /r	ADD r16,r/m16		ADD AX, [006387EA]	
03 /r	ADD r32,r/m32		ADD EAX, [006387EA]	

8.4.3.2 SUB 指令

表 8.13 SUB 指令介绍

操作码	指令形式	标志位	说明	应用举例
2C ib	SUB AL, imm8	设置 AF CF OF SF PF ZF 减法	SUB AL, 1F	
2D iw	SUB AX, imm16		SUB AX, 4F80	

续表

操作码	指令形式	标志位	说明	应用举例
2D id	SUB EAX, imm32	设置 AF CF OF SF PF ZF	减法	SUB EAX, 00004F80
80 /5 ib	SUB r/m8, imm8			SUB BYTE PTR [006387EA], 39
81 /5 iw	SUB r/m16,imm16			SUB WORD PTR [006387EA], 1039
81 /5 id	SUB r/m32,imm32			SUB DWORD PTR [006387EA], 00001039
83 /5 ib	SUB r/m16,imm8			SUB WORD PTR [006387EA], 39
83 /5 ib	SUB r/m32,imm8			SUB DWORD PTR [006387EA], 39
28 /r	SUB r/m8,r8			SUB [006387EA], AL
29 /r	SUB r/m16,r16			SUB [006387EA], AX
29 /r	SUB r/m32,r32			SUB [006387EA], EAX
2A /r	SUB r8,r/m8			SUB AL,[006387EA]
2B /r	SUB r16,r/m16			SUB AX,[006387EA]
2B /r	SUB r32,r/m32			SUB EAX,[006387EA]

8.4.3.3 IMUL 指令

表 8.14 IMUL 指令介绍

操作码	指令形式	标志位	说 明	应用举例
F6 /5	IMUL r/m8	设置 CF OF (SF ZF AF PF 未定义)	有符号乘法: AX←AL * r/m8	IMUL CL
F7 /5	IMUL r/m16		有符号乘法: DX: AX←AX * r/m16	IMUL CX
F7 /5	IMUL r/m32		有符号乘法: EDX: EAX ← EAX * r/m32	IMUL ECX
0F AF /r	IMUL r16, r/m16		有符号乘法: r16 ← r16 * r/m16	IMUL AX, BX
0F AF /r	IMUL r32, r/m32		有符号乘法: r32 ← r32 * r/m32	IMUL EAX, EBX
6B /r ib	IMUL r16,r/m16,imm8		有符号乘法: r16 ← r/m16 * imm8	IMUL AX, BX, 39
6B /r ib	IMUL r32,r/m32,imm8		有符号乘法: r32 ← r/m32 * imm8	IMUL EAX, EBX, 39
6B /r ib	IMUL r16, imm8		有符号乘法: r16 ← r16 * imm8	IMUL AX, 37
6B /r ib	IMUL r32, imm8		有符号乘法: r32 ← r32 * imm8	IMUL EAX, 37
69 /r iw	IMUL r16,r/m16,imm16		有符号乘法: r16 ← r/m16 * imm16	IMUL AX, BX, 387E
69 /r id	IMUL r32,r/m32,imm32		有符号乘法: r32 ← r/m32 * imm32	IMUL EAX, EBX, 006387EA
69 /r iw	IMUL r16, imm16		有符号乘法: r16 ← r16 * imm16	IMUL AX, 387E
69 /r id	IMUL r32, imm32		有符号乘法: r32 ← r32 * imm32	IMUL EAX, 006387EA

8.4.3.4 IDIV 指令

表 8.15 IDIV 指令介绍

操作码	指令形式	标志位	说 明	应用举例
F6 /7	IDIV r/m8	AF CF OF PF SF ZF 未定义	有符号除法	IDIV BL; AX 除以 BL, 商在 AL 中, 余数在 AH 中
F7 /7	IDIV r/m16			IDIV BX; DX: AX 除以 BX, 商在 AX 中, 余数在 DX 中
F7 /7	IDIV r/m32			IDIV EBX; EDX: EAX 除以 BX, 商在 EAX 中, 余数在 EDX 中

8.4.3.5 CMP 指令

表 8.16 CMP 指令介绍

操作码	指令形式	标志位	说 明	应用举例
3C ib	CMP AL, imm8	设置 AF CF OF PF SF ZF	比较大小, 然后设置标志位	CMP AL, 1F
3D iw	CMP AX, imm16			CMP AX, 4F80
3D id	CMP EAX, imm32			CMP EAX, 00004F80
80 /7 ib	CMP r/m8, imm8			CMP BYTE PTR [006387EA], 39
81 /7 iw	CMP r/m16,imm16			CMP WORD PTR [006387EA], 1039
81 /7 id	CMP r/m32,imm32			CMP DWORD PTR [006387EA], 00001039
83 /7 ib	CMP r/m16,imm8			CMP WORD PTR [006387EA], 39
83 /7 ib	CMP r/m32,imm8			CMP DWORD PTR [006387EA], 39
38 /r	CMP r/m8,r8			CMP BYTE PTR [006387EA], AL
39 /r	CMP r/m16,r16			CMP WORD PTR [006387EA], AX
39 /r	CMP r/m32,r32			CMP DWORD PTR [006387EA], EAX
3A /r	CMP r8,r/m8			CMP AL, [006387EA]
3B /r	CMP r16,r/m16			CMP AX, [006387EA]
3B /r	CMP r32,r/m32			CMP EAX, [006387EA]

8.4.3.6 CWD 指令

表 8.17 CWD 指令介绍

操作码	指令形式	标志位	说 明	应用举例
99	CWD	不影响标志位	将 AX 带符号扩展到 DX:AX	CWD

8.4.3.7 CDQ 指令

表 8.18 CDQ 指令介绍

操作码	指令形式	标志位	说 明	应用举例
99	CDQ	不影响标志位	将 EAX 值带符号扩展到 EDX:EAX	CDQ

8.4.4 逻辑运算指令

SCC 编译器用到的逻辑运算指令有 XOR、TEST、SETCC，下面逐一介绍（见表 8.19～表 8.21）。

8.4.4.1 XOR 指令

表 8.19 XOR 指令介绍

操作码	指令形式	标志位	说 明	应用举例
34 ib	XOR AL, imm8	设置 CF OF PF SF ZF	逻辑异或	XOR AL, 1F
35 iw	XOR AX, imm16			XOR AX, 4F80
35 id	XOR EAX, imm32			XOR EAX, 00004F80
80 /6 ib	XOR r/m8, imm8			XOR BYTE PTR [006387EA], 39
81 /6 iw	XOR r/m16,imm16			XOR WORD PTR [006387EA], 1039
81 /6 id	XOR r/m32,imm32			XOR DWORD PTR [006387EA], 00001039
83 /6 ib	XOR r/m16,imm8			XOR WORD PTR [006387EA], 39
83 /6 ib	XOR r/m32,imm8			XOR DWORD PTR [006387EA], 39
30 /r	XOR r/m8,r8			XOR BYTE PTR [006387EA], AL
31 /r	XOR r/m16,r16			XOR WORD PTR [006387EA], AX
31 /r	XOR r/m32,r32			XOR DWORD PTR [006387EA], EAX
32 /r	XOR r8,r/m8			XOR AL, [006387EA]
33 /r	XOR r16,r/m16			XOR AX, [006387EA]
33 /r	XOR r32,r/m32			XOR EAX, [006387EA]

8.4.4.2 TEST 指令

表 8.20 TEST 指令介绍

操作码	指令形式	标志位	说 明	应用举例
A8 ib	TEST AL, imm8	设置 CF OF PF SF ZF	逻辑与 测 试，但是不 改变目的操 作数，只设 置相关标 志位	TEST AL, 1F
A9 iw	TEST AX, imm16			TEST AX, 4F80
A9 id	TEST EAX, imm32			TEST EAX, 00004F80

续表

操作码	指令形式	标志位	说 明	应用举例
F6 /0 ib	TEST r/m8, imm8	设置 CF OF PF SF ZF	逻辑与测试,但是不改变目的操作数,只设置相关标志位	TEST BYTE PTR [006387EA], 39
F7 /0 ib	TEST r/m16,imm16			TEST WORD PTR [006387EA], 1039
F7 /0 ib	TEST r/m32,imm32			TEST DWORD PTR [006387EA], 00001039
84 /r	TEST r/m8,r8			TEST BYTE PTR [006387EA], AL
85 /r	TEST r/m16,r16			TEST WORD PTR [006387EA], AX
85 /r	TEST r/m32,r32			TEST DWORD PTR [006387EA], EAX

8.4.4.3 SETcc 指令

表 8.21 SETcc 指令介绍

操作码	指令形式	标志位	说 明	应用举例
0F 97	SETA r/m8	高于(CF=0 and ZF=0)	条件设置指令,如果条件满足则 r/m8 = 1;否则,r/m8=0	SETA AL
0F 93	SETAE r/m8	高于等于(CF=0)		SETAE AL
0F 92	SETB r/m8	低于(CF=1)		SETB AL
0F 96	SETBE r/m8	低于等于(CF=1 or ZF=1)		SETBE AL
0F 92	SETC r/m8	有进位(CF=1)		SETC AL
0F 94	SETE r/m8	等于(ZF=1)		SETE AL
0F 9F	SETG r/m8	大于(ZF=0 and SF=OF)		SETG AL
0F 9D	SETGE r/m8	大于等于(SF=OF)		SETGE AL
0F 9C	SETL r/m8	小于(SF<>OF)		SETL AL
0F 9E	SETLE r/m8	小于等于(ZF=1 or SF<>OF)		SETLE AL
0F 96	SETNA r/m8	不高于(CF=1 or ZF=1)		SETNA AL
0F 92	SETNAE r/m8	不高等于(CF=1)		SETNAE AL
0F 93	SETNB r/m8	不低于(CF=0)		SETNB AL
0F 97	SETNBE r/m8	不低等于(CF=0 and ZF=0)		SETNBE AL
0F 93	SETNC r/m8	无进位(CF=0)		SETNC AL
0F 95	SETNE r/m8	不等于(ZF=0)		SETNE AL
0F 9E	SETNG r/m8	不大于(ZF=1 or SF<>OF)		SETNG AL
0F 9C	SETNGE r/m8	不大等于(SF<>OF)		SETNGE AL

续表

操作码	指令形式	标志位	说 明	应用举例
0F 9D	SETNL r/m8	不小于(SF=OF)		SETNL AL
0F 9F	SETNLE r/m8	不小等于(ZF=0 and SF=OF)		SETNLE AL
0F 91	SETNO r/m8	无溢出(OF=0)		SETNO AL
0F 9B	SETNP r/m8	非偶数(PF=0)		SETNP AL
0F 99	SETNS r/m8	非负数(SF=0)		SETNS AL
0F 95	SETNZ r/m8	非零(ZF=0)		SETNZ AL
0F 90	SETO r/m8	溢出(OF=1)		SETO AL
0F 9A	SETP r/m8	偶数(PF=1)		SETP AL
0F 9A	SETPE r/m8	偶数(PF=1)		SETPE AL
0F 9B	SETPO r/m8	奇数(PF=0)		SETPO AL
0F 98	SETS r/m8	负数(SF=1)		SETS AL
0F 94	SETZ r/m8	为零(ZF=1)		SETZ AL

8.4.5 控制转移指令

SCC 编译器用到的控制转移指令有 JMP、CALL、RET、JCC，下面逐一介绍（见表 8.22～表 8.25）。

8.4.5.1 JMP 指令

表 8.22 JMP 指令介绍

操作码	指令形式	标志位	说 明	应用举例
EB cb	JMP rel8	不影响标志位	无条件转移指令	
E9 cw	JMP rel16			
E9 cd	JMP rel32			
FF /4	JMP r/m16			
FF /4	JMP r/m32			
EA cd	JMP ptr16:16			
EA cp	JMP ptr16:32			
FF /5	JMP m16:16			
FF /5	JMP m16:32			

8.4.5.2 CALL 指令

表 8.23 CALL 指令介绍

操作码	指令形式	标志位	说 明	应用举例
E8 cw	CALL rel16	不影响 标志位	子程序调用(16位相对寻址)	
E8 cd	CALL rel32		子程序调用(32位相对寻址)	
FF /2	CALL r/m16		子程序调用(16位间接寻址)	
FF /2	CALL r/m32		子程序调用(32位间接寻址)	
9A cd	CALL ptr16:16		子程序调用(直接绝对寻址)	
9A cp	CALL ptr16:32		子程序调用(直接绝对寻址)	
FF /3	CALL m16:16		子程序调用(间接绝对寻址)	
FF /3	CALL m16:32		子程序调用(间接绝对寻址)	

8.4.5.3 RET 指令

表 8.24 RET 指令介绍

操作码	指令形式	标志位	说 明	应用举例
C3	RET	恢复压栈 的标志位	子过程返回(Near)	RET
CB	RET		子过程返回(Far)	RET
C2 iw	RET imm16		子过程返回(Near),并从堆栈弹出 imm16 字节	RET 08
CA iw	RET imm16		子过程返回(Far),并从堆栈弹出 imm16 字节	RET 08

8.4.5.4 Jcc 指令

表 8.25 Jcc 指令介绍

操作码	指令形式	标志位	说 明	应用举例
77 cb	JA rel8	高于(CF=0 and ZF=0)	条件转移指令	
73 cb	JAE rel8	高于等于(CF=0)		
72 cb	JB rel8	低于(CF=1)		
76 cb	JBE rel8	低于等于(CF=1 or ZF=1)		
72 cb	JC rel8	有进位(CF=1)		
E3 cb	JCXZ rel8	CX=0 则跳		
E3 cb	JECXZ rel8	ECX=0 则跳		
74 cb	JE rel8	等于(ZF=1)		
7F cb	JG rel8	大于(ZF=0 and SF=OF)		

续表

操作码	指令形式	标志位	说 明	应用举例
7D cb	JGE rel8	大于等于(SF=OF)	条件转移指令	
7C cb	JL rel8	小于(SF<>OF)		
7E cb	JLE rel8	小于等于(ZF=1 or SF<>OF)		
76 cb	JNA rel8	不高于(CF=1 or ZF=1)		
72 cb	JNAE rel8	不高等于(CF=1)		
73 cb	JNB rel8	不低于(CF=0)		
77 cb	JNBE rel8	不低等于(CF=0 and ZF=0)		
73 cb	JNC rel8	无进位(CF=0)		
75 cb	JNE rel8	不等于(ZF=0)		
7E cb	JNG rel8	不大于(ZF=1 or SF<>OF)		
7C cb	JNGE rel8	不大等于(SF<>OF)		
7D cb	JNL rel8	不小于(SF=OF)		
7F cb	JNLE rel8	不小等于(ZF=0 and SF=OF)		
71 cb	JNO rel8	无溢出(OF=0)		
7B cb	JNP rel8	非偶数(PF=0)		
79 cb	JNS rel8	非负数(SF=0)		
75 cb	JNZ rel8	非零(ZF=0)		
70 cb	JO rel8	溢出(OF=1)		
7A cb	JP rel8	偶数(PF=1)		
7A cb	JPE rel8	偶数(PF=1)		
7B cb	JPO rel8	奇数(PF=0)		
78 cb	JS rel8	负数(SF=1)		
74 cb	JZ rel8	为零(ZF=1)		
0F 87 cw/cd	JA rel16/32	高于(CF=0 and ZF=0)		
0F 83 cw/cd	JAE rel16/32	高于等于(CF=0)		
0F 82 cw/cd	JB rel16/32	低于(CF=1)		
0F 86 cw/cd	JBE rel16/32	低于等于(CF=1 or ZF=1)		
0F 82 cw/cd	JC rel16/32	有进位(CF=1)		
0F 84 cw/cd	JE rel16/32	等于(ZF=1)		
0F 84 cw/cd	JZ rel16/32	为零(ZF=1)		
0F 8F cw/cd	JG rel16/32	大于(ZF=0 and SF=OF)		

8.4.6 串操作指令

SCC 编译器用到的串操作指令有 MOVS/MOVSB/MOVSW/MOVSD, 下面逐一介绍(见表 8.26)。

表 8.26 MOVS/MOVSB/MOVSW/MOVSD 指令介绍

操作码	指令形式	标志位	说 明	应用举例
A4	MOVS m8, m8	不影响标志位	字符串传送,每次传送 1 个字节	MOVS STRING1, STRING2; 源串: DS:(E)SI, 目的串: ES:(E)DI
A5	MOVS m16, m16		字符串传送,每次传送 1 个字	MOVS STRING1, STRING2; 源串: DS:(E)SI 目的串: ES:(E)DI
A5	MOVS m32, m32		字符串传送,每次传送 1 个双字	MOVS STRING1, STRING2; 源串: DS:(E)SI, 目的串: ES:(E)DI (386+)
A4	MOVSB		字符串传送,每次传送 1 个字节	MOVSB; 源串: DS:(E)SI, 目的串: ES:(E)DI
A5	MOVSW		字符串传送,每次传送 1 个字	MOVSW; 源串: DS:(E)SI, 目的串: ES:(E)DI
A5	MOVSD		字符串传送,每次传送 1 个双字	MOVSD; 源串: DS:(E)SI, 目的串: ES:(E)DI (386+)

8.4.7 处理器控制指令

SCC 编译器用到的处理器控制指令只有 NOP(见表 8.27)。

表 8.27 NOP 指令介绍

操作码	指令形式	标 志 位	说 明	应用举例
90	NOP	不影响标志位	空操作	NOP

8.5 生成 x86 机器语言

学完 x86 机器语言,下面来看生成 x86 机器语言的相关代码。代码较难理解的地方作者都做了注释,后面的代码主要用到了本章前几节介绍的内容,如果没有特殊需要说明的地方作者将不做过多解释,下面是本节内容阅读指南:

- (1) 生成机器语言用到的辅助数据结构,操作数栈;
- (2) 生成通用指令,结合 8.2 节来阅读;
- (3) 生成数据传送指令,结合 8.4.2 节来阅读;

- (4) 生成算术与逻辑运算指令,结合 8.4.3 节~8.4.4 节、9.3.2 节~9.3.5 节来阅读;
- (5) 生成控制转移指令,结合 8.3.5 节、9.2.2 节~9.2.4 节来阅读;
- (6) 寄存器分配,结合 8.3 节来阅读。

8.5.1 操作数栈

这里先介绍一下为什么要用到操作数栈? 请大家看一下表达式 $3+5*6$ 的计算过程(见图 8.2)。

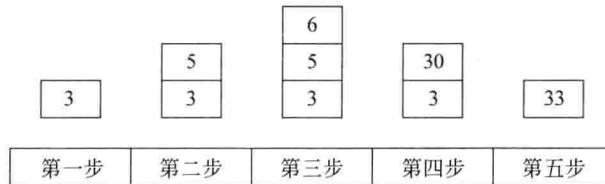


图 8.2 表达式 $3+5*6$ 计算过程图

大家观察一下上面的计算过程,是不是用“后进先出”栈结构存储最合适不过了,所以本章的操作数统一放在操作数栈中。

8.5.1.1 操作数存储结构

```
/* 操作数存储结构,存放在语义栈中 */
typedef struct Operand
{
    Type type;                                //数据类型
    unsigned short r;                          //寄存器或存储类型
    int value;                                 //常量值
    struct Symbol * sym;                      //关联符号
} Operand;
```

8.5.1.2 操作数栈相关操作

```
*****
* 功能: 操作数入栈
* type: 操作数数据类型
* r: 操作数存储类型
* value: 操作数值
*****
void operand_push(Type * type, int r, int value)
{
    if(optop>=opstack+(OPSTACK_SIZE-1))
        error("内存分配失败");
    optop++;
    optop->type= * type;
    optop->r=r;
    optop->value=value;
```

```

}

/******************
 * 功能：弹出栈顶操作数
*****************/
void operand_pop()
{
    optop--;
}

/******************
 * 功能：交换栈顶两个操作数顺序
*****************/
void operand_swap()
{
    Operand tmp;

    tmp=optop[0];
    optop[0]=optop[-1];
    optop[-1]=tmp;
}

/******************
 * 功能： 操作数赋值
 * t:      操作数数据类型
 * r:      操作数存储类型
 * value: 操作数值
*****************/
void operand_assign(Operand *opd, int t, int r, int value)
{
    opd->type.t=t;
    opd->r=r;
    opd->value=value;
}

```

这里给出程序中用到的全局变量及宏定义：

```

#define OPSTACK_SIZE 256           //操作数栈容量
Operand opstack[OPSTACK_SIZE],   //操作数栈
        * optop;                  //操作数栈栈顶

```

可以看到操作数采用了一个存储容量固定的静态栈，栈容量最大值预设为 256，大家可能担心这个容量是不是太小了，是不是很容易就会溢出，作者试验过，正常情况下写程序，栈中元素个数连 10 都超不过，预留 256 的容量已经非常充足了。

8.5.2 生成通用指令

```

/******************

```

```
* 功能： 向代码节写入一个字节
* c:    字节值
*****
void gen_byte(char c)
{
    int ind1;
    ind1=ind+1;
    if(ind1>sec_text->data_allocated)
        section_realloc(sec_text, ind1);
    sec_text->data[ind]=c;
    ind=ind1;
}
```

8.5.2.1 生成指令前缀

```
*****
* 功能： 生成指令前缀
* opcode: 指令前缀编码
*****
void gen_prefix(char opcode)
{
    gen_byte(opcode);
}
```

8.5.2.2 生成操作码

```
*****
* 功能： 生成单字节指令
* opcode: 指令编码
*****
void gen_opcode1(char opcode)
{
    gen_byte(opcode);
}

*****
* 功能： 生成双字节指令
* first: 指令第一个字节
* second: 指令第二个字节
*****
void gen_opcode2(char first, char second)
{
    gen_byte(first);
    gen_byte(second);
}
```

8.5.2.3 生成 ModR/M 字节

```
/************************************************************************/
 * 功能：      生成指令寻址方式字节 ModR/M,
 * mod:        ModR/M [7:6]
 * reg_opcode: ModR/M [5:3] 指令的另外 3 位操作码 源操作数(叫法不准确)
 * r_m:        ModR/M [2:0] 目标操作数(叫法不准确)
 * sym:        符号指针
 * c:          符号关联值
/************************************************************************/
void gen_modrm(int mod, int reg_opcode, int r_m, Symbol * sym, int c)
{
    mod<<=6;
    reg_opcode<<=3;
    if(mod==0xc0)
    {
        //mod=11 寄存器寻址
        //89 E5 (mod=11 reg_opcode=100ESP r=101 EBP)MOV EBP, ESP
        gen_byte(mod | reg_opcode | (r_m & SC_VALMASK));
    }
    else if((r_m & SC_VALMASK)==SC_GLOBAL)
    {
        //mod=00 r=101 直接寻址
        //8b 05 50 30 40 00  MOV EAX,DWORD PTR DS:[403050]
        gen_byte(0x05 | reg_opcode);
        gen_addr32(r_m, sym, c);
    }
    else if((r_m & SC_VALMASK)==SC_LOCAL)
    {
        if(c==(char)c)
        {
            //mod=01 r=101 disp8[EBP]
            //89 45 fc  MOV DWORD PTR SS:[EBP-4],EAX
            gen_byte(0x45 | reg_opcode);
            gen_byte(c);
        }
        else
        {
            //mod=10 r=101 disp32[EBP]
            //89 85 A0FDFFFF  MOV DWORD PTR SS:[EBP-260],EAX
            gen_byte(0x85 | reg_opcode);
            gen_dword(c);
        }
    }
    else
}
```

```
{
    //mod=00
    //89 01(mod=00 reg_opcode=000EAX r=001ECX)MOV DWORD PTR DS:[ECX],EAX
    gen_byte(0x00 | reg_opcode | (r_m & SC_VALMASK));
}
}
```

8.5.2.4 生成操作数

```
*****
* 功能：生成 4 字节操作数
* c: 4 字节操作数
*****
void gen_dword(unsigned int c)
{
    gen_byte(c);
    gen_byte(c>>8);
    gen_byte(c>>16);
    gen_byte(c>>24);
}

*****
* 功能：生成全局符号地址，并增加 COFF 重定位记录
* r: 符号存储类型
* sym: 符号指针
* c: 符号关联值
*****
void gen_addr32(int r, Symbol * sym, int c)
{
    if(r & SC_SYM)
        coffreloc_add(sec_text, sym, ind, IMAGE_REL_I386_DIR32);
    gen_dword(c);
}
```

这里要注意对于全局符号，由于在代码生成时还无法知道符号地址，所以需要增加一个重定位条目记一下，其通俗理解就是编译器嘱咐链接器，“嘿，伙计！我这做了重定位标记的，链接生成 EXE 时，别忘了修改这个符号地址”。

8.5.3 生成数据传送指令

```
*****
* 功能：将操作数 opd 加载到寄存器 r 中
* r: 符号存储类型
* opd: 操作数指针
*****
void load(int r, Operand *opd)
```

```

{
    int v, ft, fc, fr;

    fr=opd->r;
    ft=opd->type.t;
    fc=opd->value;

    v=fr & SC_VALMASK;
    if(fr & SC_LVAL)
    {
        if((ft & T_BTYPEn)==T_CHAR)
        {
            //movsx--move with sign-extention
            //0F BE /r    movsx r32,r/m8    move byte to doubleword,sign-extention
            gen_opcode2(0x0f,0xbe);
        }
        else if((ft & T_BTYPEn)==T_SHORT)
        {
            //movsx--move with sign-extention
            //0F BF /r    movsx r32,r/m16   move word to doubleword,sign-extention
            gen_opcode2(0x0f,0xbf);
        }
        else
        {
            //8B /r    mov r32,r/m32    mov r/m32 to r32
            gen_opcode1(0x8b);
        }
        gen_modrm(ADDR_OTHER,r, fr, opd->sym, fc);
    }
    else
    {
        if(v==SC_GLOBAL)
        {
            //B8+rd    mov r32,imm32      mov imm32 to r32
            gen_opcode1(0xb8+r);
            gen_addr32(fr, opd->sym, fc);
        }
        else if(v==SC_LOCAL)
        {
            //8D /r    LEA r32,m      Store effective address for m in register r32
            gen_opcode1(0x8d);
            gen_modrm(ADDR_OTHER,r, SC_LOCAL, opd->sym, fc);
        }
        else if(v==SC_CMP)           //适用于 c=a>b 情况
        {

```

```

/* c=a>b 生成代码
00401384 39C8          CMP EAX,ECX
00401386 B8 00000000    MOV EAX,0
0040138B 0F9FC0         SETG AL
0040138E 8945 FC        MOV DWORD PTR SS:[EBP-4],EAX
*/
/* B8+rd mov r32,imm32      mov imm32 to r32 */
gen_opcode1(0xb8+r);     /* mov r, 0 */
gen_dword(0);
//OF 9F           SETG r/m8      Set byte if greater(ZF=0 and SF=OF)
//OF 8F cw/cd     JG rel16/32  jump near if greater(ZF=0 and SF=OF)
gen_opcode2(0x0f,fc+16);
gen_modrm(ADDR_REG,0,r,NULL,0);
}
else if(v !=r)
{
    //89 /r    MOV r/m32,r32    Move r32 to r/m32
    gen_opcode1(0x89);
    gen_modrm(ADDR_REG,v,r,NULL,0);
}
}

/* 寻址方式 */
enum e_addrForm
{
    ADDR_OTHER,           //寄存器间接寻址
    ADDR_REG=3            //寄存器直接寻址
};

/***********************
 * 功能：将寄存器'r'中的值存入操作数'opd'
 * r: 符号存储类型
 * opd: 操作数指针
***********************/
void store(int r, Operand * opd)
{
    int fr, bt;

    fr=opd->r & SC_VALMASK;
    bt=opd->type.t & T_BTYP;
    if(bt==T_SHORT)
        gen_prefix(0x66);      //Operand-size override, 66H
    if(bt==T_CHAR)
        //88 /r    MOV r/m,r8    Move r8 to r/m8
        gen_opcode1(0x88);
}

```

```

else
    //89 /r      MOV r/m32,r32      Move r32 to r/m32
    gen_opcode1(0x89);

if(fr==SC_GLOBAL || fr==SC_LOCAL || (opd->r & SC_LVAL))
{
    gen_modrm(ADDR_OTHER,r, opd->r, opd->sym, opd->value);
}
}

/******************
 * 功能：将栈顶操作数加载到'rc'类寄存器中
 * rc:  寄存器类型
 * opd: 操作数指针
 *****************/
int load_1(int rc, Operand * opd)
{
    int r;
    r=opd->r & SC_VALMASK;
    //需要加载到寄存器中情况
    //栈顶操作数目前尚未分配寄存器
    //栈顶操作数已分配寄存器,但为左值 * p
    if(r>=SC_GLOBAL ||
       (opd->r & SC_LVAL)
    )
    {
        r=allocate_reg(rc);
        load(r, opd);
    }
    opd->r=r;
    return r;
}

/******************
 * 功能：将栈顶操作数加载到'rc1'类寄存器,将次栈顶操作数加载到'rc2'类寄存器
 * rc1: 栈顶操作数加载到的寄存器类型
 * rc2: 次栈顶操作数加载到的寄存器类型
 *****************/
void load_2(int rc1, int rc2)
{
    load_1(rc2,optop);
    load_1(rc1,&optop[-1]);
}

/******************

```

```

* 功能：将栈顶操作数存入次栈顶操作数中
*****
void store0_1()
{
    int r,t;
    r=load_1(REG_ANY,optop);
    //左值如果被溢出到栈中，必须加载到寄存器中
    if((optop[-1].r & SC_VALMASK)==SC_LLOCAL)
    {
        Operand opd;
        t=allocate_reg(REG_ANY);
        operand_assign(&opd, T_INT, SC_LOCAL | SC_LVAL, optop[-1].value);
        load(t, &opd);
        optop[-1].r=t | SC_LVAL;
    }
    store(r, optop-1);
    operand_swap();
    operand_pop();
}

```

8.5.4 生成算术与逻辑运算指令

```

*****
* 功能：生成二元运算，对指针操作数进行一些特殊处理
* op: 运算符类型
*****
void gen_op(int op)
{
    int u, bt1, bt2;
    Type type1;

    bt1=optop[-1].type.t & T_BTYP;
    bt2=optop[0].type.t & T_BTYP;

    if(bt1==T_PTR || bt2==T_PTR)
    {
        if(op>=TK_EQ && op<=TK_GEQ)           //关系运算
        {
            gen_ope(op);
            optop->type.t=T_INT;
        }
        else if(bt1==T_PTR && bt2==T_PTR)      //两个操作数都为指针
        {
            if(op !=TK_MINUS)
                error("两个指针只能进行关系或减法运算");
        }
    }
}
```

```

        u=pointed_size(&optop[-1].type);
        gen_opi(op);
        optop->type.t=T_INT;
        operand_push(&int_type, SC_GLOBAL, u);
        gen_op(TK_DIVIDE);
    }
    else //两个操作数一个是指针,另一个不是指针,并且非关系运算
    {
        if(op != TK_MINUS && op != TK_PLUS)
            error("指针只能进行关系或加减运算");
        //指针作为第一操作数
        if(bt2==T_PTR)
        {
            operand_swap();
        }
        type1=optop[-1].type;
        operand_push(&int_type, SC_GLOBAL, pointed_size(&optop[-1].type));
        gen_op(TK_STAR);

        gen_opi(op);
        optop->type=type1;
    }
}
else
{
    gen_opi(op);
    if(op>=TK_EQ && op<=TK_GEQ)
    {
        //关系运算结果为 T_INT 类型
        optop->type.t=T_INT;
    }
}
}

/*********************************************
* 功能：生成整数运算
* op: 运算符类型
********************************************/
void gen_opi(int op)
{
    int r, fr, opc;

    switch(op)
    {
        case TK_PLUS:

```

```
opc=0;
gen_opi2(opc,op);
break;

case TK_MINUS:
opc=5;
gen_opi2(opc,op);
break;

case TK_STAR:
load_2(REG_ANY, REG_ANY);
r=top[-1].r;
fr=top[0].r;
operand_pop();

//IMUL——有符号乘法
//OF AF/r  IMULr32,r/m32  doubleword register<-doubleword register * r/m doubleword
gen_opcode2(0x0f,0xaf);
gen_modrm(ADDR_REG,r,fr,NULL,0);
break;

case TK_DIVIDE:
case TK_MOD:
opc=7;
load_2(REG_EAX, REG_ECX);
r=top[-1].r;
fr=top[0].r;
operand_pop();
spill_reg(REG_EDX);

//CWD/CDQ--Convert Word to Doubleword/Convert Doubleword to Qword
//99    CWQ    EDX:EAX<-sign_extended EAX
gen_OPCODE1(0x99);

//IDIV——有符号除法
//F7 /7    IDIV r/m32    Signed divide EDX:EAX    by r/m doubleword
//EDX:EAX 被除数 r/m32 除数
//结果：EAX=商 EDX=余数
gen_OPCODE1(0xf7);
gen_modrm(ADDR_REG,opc,fr,NULL,0);

if(op==TK_MOD)
    r=REG_EDX;
else
    r=REG_EAX;
top->r=r;
break;

default:
```

```

        opc=7;
        gen_opi2(opc,op);
        break;
    }
}

//*************************************************************************
/* 功能：生成整数二元运算
 * opc: ModR/M [5:3]
 * op: 运算符类型
 *****/
void gen_opi2(int opc,int op)
{
    int r, fr, c;
    if((optop->r & (SC_VALMASK | SC_LVAL | SC_SYM))==SC_GLOBAL)
    {
        r=load_1(REG_ANY,&optop[-1]);
        c=optop->value;
        if(c==(char)c)
        {
//ADC--Add with Carry 83 /2 ib ADC r/m32,imm8 Add with CF sign-extended imm8 to r/m32
//ADD--Add          83 /0 ib ADD r/m32,imm8 Add sign-extended imm8 from r/m32
//SUB--Subtract     83 /5 ib SUB r/m32,imm8 Subtract sign-extended imm8 to r/m32
//CMP--Compare Two Operands 83 /7 ib CMP r/m32,imm8 Compare imm8 with r/m32
            gen_opcode1(0x83);
            gen_modrm(ADDR_REG,opc,r,NULL,0);
            gen_byte(c);
        }
        else
        {
//ADD--Add          81 /0 id ADD r/m32,imm32 Add sign-extended imm32 to r/m32
//SUB--Subtract     81 /5 id SUB r/m32,imm32 Subtract sign-extended imm32 from r/m32
//CMP--Compare Two Operands 81 /7 id CMP r/m32,imm32 Compare imm32 with r/m32
            gen_opcode1(0x81);
            gen_modrm(ADDR_REG,opc,r,NULL,0);
            gen_byte(c);
        }
    }
    else
    {
        load_2(REG_ANY, REG_ANY);
        r=optop[-1].r;
        fr=optop[0].r;

//ADD--Add          01 /rADD r/m32,r32 Add r32 to r/m32
    }
}

```

```

//SUB--Subtract          29 /rSUB r/m32,r32  Subtract r32 from r/m32
//CMP--Compare Two Operands 39 /rCMP r/m32,r32  Compare r32 with r/m32
gen_opcode1((opc<<3) | 0x01);
gen_modrm(ADDR_REG,fr,r,NULL,0);
}
operand_pop();
if(op>=TK_EQ && op<=TK_GEQ)
{
    optop->r=SC_CMP;
    switch(op)
    {
        case TK_EQ:
            optop->value=0x84;
            break;
        case TK_NEQ:
            optop->value=0x85;
            break;
        case TK_LT:
            optop->value=0x8c;
            break;
        case TK_LEQ:
            optop->value=0x8e;
            break;
        case TK_GT:
            optop->value=0x8f;
            break;
        case TK_GEQ:
            optop->value=0x8d;
            break;
    }
}
}
}

```

8.5.5 生成控制转移指令

8.5.5.1 生成跳转指令

```

*****
* 功能：生成向高地址跳转指令，跳转地址待定
* t: 前一跳转指令地址
*****
int gen_jmpforward(int t)
{
    //JMP--Jump
    //E9 cd  JMP rel32Jump near,relative,displacement relative to next instruction
}
```

```

        gen_opcode1(0xe9);
        return makelist(t);;
    }

/************************************************************************/
/* 功能：生成向低地址跳转指令，跳转地址已确定
 * a: 跳转到的目标地址
 */
void gen_jmpbackward(int a)
{
    int r;
    r=a-ind-2;
    if(r==(char)r)
    {
        //EB cb  JMP rel8 Jump short,relative,displacement relative to next instruction
        gen_opcode1(0xeb);
        gen_byte(r);
    }
    else
    {
        //E9 cd  JMP rel32Jump short,relative,displacement relative to next instruction
        gen_opcode1(0xe9);
        gen_dword(a-ind-4);
    }
}

/************************************************************************/
/* 功能：生成条件跳转指令
 * t: 前一跳转指令地址
 * 返回值：新跳转指令地址
 */
int gen_jcc(int t)
{
    int v;
    int inv=1;

    v=optop->r & SC_VALMASK;
    if(v==SC_CMP)
    {
        //Jcc--Jump if Condition Is Met
        //...
        //0F 8F cw/cd      JG rel16/32      jump/near if greater (ZF=0 and SF=OF)
        //...
        gen_opcode2(0x0f,optop->value ^ inv);
        t=makelist(t);
    }
}

```

```

    }
else
{
    if((optop->r & (SC_VALMASK | SC_LVAL | SC_SYM)) == SC_GLOBAL)
    {
        t=gen_jmpforward(t);
    }
else
{
    v=load_1(REG_ANY,optop);

    //TEST--Logical Compare
    //85 /r TEST r/m32,r32 AND r32 with r/m32, set SF,ZF,PF according to result

    gen_opcode1(0x85);
    gen_modrm(ADDR_REG,v,v,NULL,0);

    //Jcc--Jump if Condition Is Met
    //...
    //0F 8F cw/cd JG rel16/32 jump near if greater(ZF=0 and SF=OF)
    //...
    gen_opcode2(0x0f,0x85 ^ inv);
    t=makelist(t);
}
}

operand_pop();
return t;
}
}

```

8.5.5.2 生成函数调用

```

*****
* 功能:生成函数调用代码,先将参数入栈,然后生成 call 指令
* nb_args: 参数个数
*****
void gen_invoke(int nb_args)
{
    int size, r, args_size, i, func_call;

    args_size=0;
    //参数依次入栈
    for(i=0;i<nb_args; i++)
    {
        r=load_1(REG_ANY,optop);
        size=4;

```

```

//PUSH-- Push Word or Doubleword Onto the Stack
//50+rd PUSH r32 Push r32
gen_opcode1(0x50+r);                                //push r
args_size+=size;
operand_pop();
}
spill_regs();
func_call=optop->type.ref->r;                      //得到调用约定方式
gen_call();
if(args_size && func_call !=KW_STDCALL)
    gen_addsp(args_size);
operand_pop();
}

/******************
* 功能：生成函数调用指令
******************/
void gen_call()
{
    int r;
    if((optop->r & (SC_VALMASK | SC_LVAL)) == SC_GLOBAL)
    {
        //记录重定位信息
        coffreloc_add(sec_text, optop->sym, ind+1, IMAGE_REL_I386_REL32);

        //CALL--Call Procedure E8 cd
        //CALL rel32    call near,relative,displacement relative to next instrution
        gen_opcode1(0xe8);                           /* call im */
        gen_dword(optop->value-4);
    }
    else
    {
        //FF /2 CALL r/m32 Call near, absolute indirect, address given in r/m32
        r=load_1(REG_ANY,optop);
        gen_opcode1(0xff);                          //call/jmp * r
        gen_opcode1(0xd0+r);                        //d0=11 010 000
    }
}

```

8.5.6 寄存器使用

8.5.6.1 寄存器分配

```

/******************
* 功能：寄存器分配,如果所需寄存器被占用,先将其内容溢出到栈中
*****************/

```

```

* rc: 寄存器类型
*****
int allocate_reg(int rc)
{
    int r;
    Operand * p;
    int used;

    /* 查找空闲的寄存器 */
    for(r=0;r<=REG_EBX;r++)
    {
        if(rc & REG_ANY || r==rc)
        {
            used=0;
            for(p=opstack;p<=optop;p++)
            {
                if((p->r & SC_VALMASK)==r)
                    used=1;
            }
            if(used==0) return r;
        }
    }

    //如果没有空闲的寄存器,从操作数栈底开始查找到第一个占用的寄存器溢出到栈中
    for(p=opstack;p<=optop;p++)
    {
        r=p->r & SC_VALMASK;
        if(r<SC_GLOBAL && (rc & REG_ANY || r==rc))
        {
            spill_reg(r);
            return r;
        }
    }
    return -1;
}

```

8.5.6.2 寄存器溢出

```

*****
* 功能: 将寄存器'r'溢出到内存栈中,并且标记释放'r'寄存器的操作数为局部变量
* r: 寄存器编码
*****
void spill_reg(int r)
{
    int size, align;
    Operand * p, opd;

```

```

Type * type;

for (p=opstack;p<=optop;p++)
{
    if ((p->r & SC_VALMASK)==r)
    {
        r=p->r & SC_VALMASK;
        type=&p->type;
        if(p->r & SC_LVAL)
            type=&int_type;
        size=type_size(type, &align);
        loc=calc_align(loc-size,align);
        operand_assign(&opd, type->t, SC_LOCAL | SC_LVAL, loc);
        store(r, &opd);
        if(p->r & SC_LVAL)
        {
            p->r=(p->r & ~ (SC_VALMASK)) | SC_LLOCAL;           //标识操作数放在栈中
        }
        else
        {
            p->r=SC_LOCAL | SC_LVAL;
        }
        p->value=loc;
        break;
    }
}
}

/*****************
 * 功能：将占用的寄存器全部溢出到栈中
 *****************/
void spill_regs()
{
    int r;
    Operand * p;
    for(p=opstack;p<=optop; p++)
    {
        r=p->r & SC_VALMASK;
        if(r<SC_GLOBAL)
        {
            spill_reg(r);
        }
    }
}

```

8.5.7 本章用到的全局变量

下面给出本章用到的全局变量。

```

int rsym;                                //记录 return 指令位置
int ind=0;                                 //指令在代码节位置
int loc;                                   //局部变量在栈中位置
int func_begin_ind;                      //函数开始指令
int func_ret_sub;                         //函数返回释放栈空间大小
Symbol * sym_sec_rdata;                  //只读节符号
Operand opstack[OPSTACK_SIZE];           //操作数栈
Operand *optop;                           //操作数栈栈顶

```

8.6 成果展示

学习完 x86 机器语言,下面解读一下 HelloWorld. obj 中代码节的内容,看一看 SC 源语言程序与 x86 目标语言之间的对应关系,如表 8.28 所示。

表 8.28 HelloWorld. obj 中代码节内容解读

SC 语言	指令地址	x86 机器语言	汇编语言
main 函数			
int main()	00	55	PUSH EBP
{	01	89 E5	MOV EBP,ESP
printf("Hello World!\n");	03	81 EC 00000000	SUB ESP,0
	09	B8 00000000	MOV EAX,XXXXXXXX
	0E	50	PUSH EAX
return 0;	0F	E8 FCFFFFFF	CALL XXXXXXXX(调用 printf)
}	14	83C4 04	ADD ESP,4
	17	B8 00000000	MOV EAX,0
	1C	E9 00000000	JMP hello. 00000021
	21	8BE5	MOV ESP,EBP
	23	5D	POP EBP
	24	C3	RETN

续表

SC 语言	指令地址	x86 机器语言	汇编语言
_entry 函数			
void _entry()	25	55	PUSH EBP
{	26	89E5	MOV EBP,ESP
int ret;	28	81EC 04000000	SUB ESP,4
ret=main();	2E	E8 FCFFFFFF	CALL XXXXXXXX(调用 main)
	33	8945FC	MOV DWORD PTR SS:[EBP-4],EAX
exit(ret);	36	8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]
	39	50	PUSH EAX
	3A	E8 FCFFFFFF	CALL XXXXXXXX(调用 exit)
}	3F	83C4 04	ADD ESP,4
	42	8BE5	MOV ESP,EBP
	44	5D	POP EBP
	45	C3	RETN

第 9 章

SCC语义分析

欲穷千里目，更上一层楼

——王之涣

编译原理书上有这样一句话：“紧接在词法分析和语法分析之后，编译程序要做的工作是进行语义分析。”这句话说得轻松，做起来可没那么容易，我们从第 5 章语法分析完之后，在正式语义分析之前用了第 6 章、第 7 章、第 8 章 3 章进行了语义分析的准备工作，这就是为什么大多数人写个语法分析器来玩一下还可以，真要想进行语义分析就玩不转了。

词法分析和语法分析主要用来解决单词和语言成分的识别及词法和语法结构的检查。但是仅仅完成这两部分还是不够的，因为程序不但要在词法和语法结构上满足要求，它的语义也必须正确。也就是说，源程序和编译后的目标程序可以在语法结构上不同，但是它们所要表达的语义必须是一致的，它们所产生的结果必须一样，否则编译过程就失去了意义。SCC 编译器语义分析的核心任务是当好“傀儡”，将 SC 源语言这个“主子”的意思中规中矩地翻译成 x86 机器语言。

有了前几章的铺垫，用户会发现本章新增的代码并不是很多，如果前几章的内容都消化了，本章阅读起来还是比较轻松的。另外说明一下在前几章已经出现的代码函数，如果没有变化的本章将不再赘述。

9.1 外部定义

9.1.1 声明与函数定义

声明与函数定义的代码如下所示。

```
*****
 * 功能：解析外部声明
 * l： 存储类型，局部的还是全局的
*****
void external_declaration(int l)
{
    :
    Type btype, type;
    int v, has_init, r,addr;
    Symbol * sym;
    Section * sec=NULL;
```

```

    :
else                                //变量声明
{
    r=0;
    if(!(type.t & T_ARRAY))
        r |=SC_LVAL;

    r |=1;
    has_init=(token==TK_ASSIGN);

    if(has_init)
    {
        get_token();
    }

    sec=allocate_storage(&type,r,has_init,v,&addr);
    sym=var_sym_put(&type,r,v,addr);
    if(l==SC_GLOBAL)
        coffsym_add_update(sym,addr,sec->index,0,IMAGE_SYM_CLASS_EXTERNAL);

    if(has_init)
    {
        initializer(&type,addr,sec);
    }
}
:
}

/*****
 * 功能：      分配存储空间
 * type:       变量类型
 * r:          变量存储类型
 * has_init:   是否需要进行初始化
 * v:          变量符号编号
 * addr(输出): 变量存储地址
 * 返回值:     变量存储节
 *****/
Section * allocate_storage(Type * type, int r, int has_init, int v, int * addr)
{
    int size, align;
    Section * sec=NULL;
    size=type_size(type,&align);

    if(size<0)
    {
        if(type->t&T_ARRAY && type->ref->type.t==T_CHAR)

```

```

    {
        type->ref->c=strlen((char*)tkstr.data)+1;
        size=type_size(type, &align);
    }
    else
        error("类型尺寸未知");
}

//局部变量在栈中分配存储空间
if((r & SC_VALMASK)==SC_LOCAL)
{
    loc=calc_align(loc-size,align);
    *addr=loc;
}
else
{
    if(has_init==1)           //初始化的全局变量在.data节分配存储空间
        sec=sec_data;
    else if(has_init==2)      //字符串常量在.rdata节分配存储空间
        sec=sec_rdata;
    else                      //未初始化的全局变量在.bss节分配存储空间
        sec=sec_bss;

    sec->data_offset=calc_align(sec->data_offset,align);
    *addr=sec->data_offset;
    sec->data_offset+=size;

    //为需要初始化的数据在节中分配存储空间
    if(sec->sh.Characteristics & IMAGE_SCN_CNT_INITIALIZED_DATA &&
       sec->data_offset>sec->data_allocated)
        section_realloc(sec, sec->data_offset);

    if(v==0)//常量字符串
    {
        operand_push(type, SC_GLOBAL | SC_SYM, *addr);
        optop->sym=sym_sec_rdata;
    }
}
return sec;
}

```

这里主要做了3项工作：

- (1) 为变量分配存储空间，局部变量存放在栈中，全局变量分两类，声明时进行赋值的存放在.data数据节中，声明时不进行赋值的存放在.bss节中。

- (2) 将全局变量放入 COFF 符号表。
- (3) 对声明时进行赋值的变量进行初始化。

9.1.2 初值符

```
*****
* 功能: 解析初值符
* type: 变量类型
* c: 变量相关值
* sec: 变量所在节
*****
void initializer(Type * type, int c, Section * sec)
{
    if(type->t & T_ARRAY && sec)
    {
        memcpy(sec->data+c, tkstr.data, tkstr.count);
        get_token();
    }
    else
    {
        assignment_expression();
        int_variable(type, sec, c, 0);
    }
}

*****
* 功能: 变量初始化
* type: 变量类型
* sec: 变量所在节
* c: 变量相关值
* v: 变量符号编号
*****
void init_variable(Type * type, Section * sec, int c, int v)
{
    int bt;
    void * ptr;

    if(sec)
    {
        if((optop->r & (SC_VALMASK | SC_LVAL)) != SC_GLOBAL)
            error("全局变量必须用常量初始化");

        bt=type->t & T_BTYP;
        ptr=sec->data+c;
    }
}
```

```

switch(bt)
{
    case T_CHAR:
        * (char *)ptr=optop->value;           //适用于 char g_c='a';
        break;
    case T_SHORT:
        * (short *)ptr=optop->value;          //适用于 short g_s=123;
        break;
    default:
        if(optop->r & SC_SYM)
        {
            //适用于 char * g_pstr="g_pstr_Hello";
            coffreloc_add(sec, optop->sym, c, IMAGE_REL_I386_DIR32);
        }
        * (int *)ptr=optop->value;
        break;
    }
    operand_pop();
}
else
{
    if(type->t&T_ARRAY)
    {
        operand_push(type, SC_LOCAL|SC_LVAL,c);
        operand_swap();
        spill_reg(REG_ECX);

        //B8+rd mov r32,imm32      mov imm32 to r32
        gen_opcode1(0xB8+REG_ECX);           //move ecx,n
        gen_dword(optop->type.ref->c);
        gen_opcode1(0xB8+REG_ESI);
        gen_addr32(optop->r,optop->sym,optop->value);           //move esi<-
        operand_swap();

        //LEA--Load Effective Address
        //8D /r    LEA r32,m    Store effective address for m in register r32
        //lea edi, [ebp-n]
        gen_opcode1(0x8D);
        gen_modrm(ADDR_OTHER,REG_EDI,SC_LOCAL,optop->sym,optop->value);

        //Instruction prefix F3H--REPE/REPZ prefix (used only with string instructions)
        gen_prefix(0xf3);                  //rep movs byte
    }
}

```

```
//MOVMS/MOVSB/MOVSW/MOVSD--Move Data from String to String
//A4      MOVS B     Move byte at address DS:(E)SI to address ES:(E)SI
gen_opcode1(0xA4);
optop-=2;
}
else
{
    operand_push(type, SC_LOCAL|SC_LVAL, c);
    operand_swap();
    store0_1();
    operand_pop();
}
}
```

从上面的代码可以看出，字符串初始化实际上是将字符串内容拷贝到为字符串预留的内存空间中。变量初始化分两种，全局变量初始化将不产生可执行代码，只是将数据存入 .data 数据节相应位置，局部变量初始化需要通过代码进行赋值操作。

9.1.3 函数体

```
*****  
* 功能：解析函数体  
* sym: 函数符号  
*****  
void funcbody(Symbol * sym)  
{  
    ind=sec_text->data_offset;  
    coffsym_add_update(sym, ind, sec_text->index, CST_FUNC, IMAGE_SYM_CLASS_EXTERNAL);  
    /* 放一匿名符号在局部符号表中 */  
    sym_direct_push(&local_sym_stack, SC_ANOM, &int_type, 0);  
    gen_prolog(&sym->type);  
    rsym=0;  
    compound_statement(NULL,NULL);  
    backpatch(rsym,ind);  
    gen_epilog();  
    sec_text->data_offset=ind;  
    sym_pop(&local_sym_stack, NULL); /* 清空局部符号栈 */  
}  
  
*****  
* 功能：生成函数开头代码  
* func_type: 函数类型  
*****  
void gen_prolog(Type * func_type)  
{
```

```

int addr, align, size, func_call;
int param_addr;
Symbol * sym;
Type * type;

sym=func_type->ref;
func_call=sym->r;
addr=8;
loc=0;
func_begin_ind=ind;
ind+=FUNC_PROLOG_SIZE;
//SUB ESP, ??函数解析结束确定了需预留的栈空间大小反填该位置
if(sym->type.t==T_STRUCT)
    error("不支持返回结构体,可以返回结构体指针");
//参数定义
while((sym=sym->next)!=NULL)
{
    type=&sym->type;
    size=type_size(type, &align);
    size=calc_align(size, 4);           //压栈时要求每个参数必须按4字节对齐
    //结构体作为指针传递
    if((type->t & T_BTYPE)==T_STRUCT)
    {
        size=4;
    }

    param_addr=addr;
    addr+=size;

    sym_push(sym->v & ~SC_PARAMS, type,
             SC_LOCAL | SC_LVAL, param_addr);
}

func_ret_sub=0;
//_stdcall调用约定,函数本身负责清理堆栈
if(func_call==KW_STDCALL)
    func_ret_sub=addr-8;
}

/******************
 * 功能:生成函数结尾代码
 *****************/
void gen_epilog()

```

```

{
    int v, saved_ind, opc;

    //8B /r      mov r32,r/m32      mov r/m32 to r32
    gen_opcode1(0x8b);                      /* mov esp, ebp */
    gen_modrm(ADDR_REG,REG_ESP,REG_EBP,NULL,0);

    //58+   rd      POP r32
    gen_opcode1(0x58+REG_EBP);                /* pop ebp */

    if(func_ret_sub==0)
    {
        //C3      RET
        gen_opcode1(0xc3);                  //ret
    }
    else
    {
        //C2 iw      RET imm16
        gen_opcode1(0xc2);                //ret n
        gen_byte(func_ret_sub);
        gen_byte(func_ret_sub>>8);
    }

    v=calc_align(-loc,4);
    saved_ind=ind;
    ind=func_begin_ind;

    //PUSH-- Push Word or Doubleword Onto the Stack
    //50+rd    PUSH r32    Push r32
    gen_opcode1(0x50+REG_EBP);                //push ebp

    //89 /r      MOV r/m32,r32      Move r32 to r/m32
    gen_opcode1(0x89);                      //mov ebp, esp
    gen_modrm(ADDR_REG,REG_ESP,REG_EBP,NULL,0);

    //SUB-- Subtract    81 /5 id      SUB r/m32,imm32
    gen_opcode1(0x81);                      //sub esp, stacksize
    opc=5;
    gen_modrm(ADDR_REG,opc,REG_ESP,NULL,0);
    gen_dword(v);
    ind=saved_ind;
}

```

请大家结合表 9.1 中的例子来理解上面的代码。

表 9.1 空函数生成目标语言举例

SC 语 言	x86 机 器 语 言	汇 编 语 言
void function_analysis() { } }	55 89E5 81EC 00000000 8BE5 5D C3	PUSH EBP MOV EBP,ESP SUB ESP,0 MOV ESP,EBP POP EBP RETN
void cdecl_analysis(int x) { int a; }	55 89E5 81EC 04000000 8BE5 5D C3	PUSH EBP MOV EBP,ESP SUB ESP,4 MOV ESP,EBP POP EBP RETN
void __stdcall stdcall_analysis(int x) { int a,b; }	55 89E5 81EC 08000000 8BE5 5D C2 0400	PUSH EBP MOV EBP,ESP SUB ESP,8 MOV ESP,EBP POP EBP RETN 4

在这里重点探讨两个问题：

(1) 栈空间分配问题，每个函数需要的栈空间大小由函数用到的局部变量决定，function_analysis 函数没有局部变量，“SUB ESP,0”表明预留栈空间为 0 字节；cdecl_analysis 函数有一个 int 型局部变量，占 4 字节，“SUB ESP,4”表明预留栈空间为 4 字节；cdecl_analysis 函数有两个 int 型局部变量，占 8 字节，“SUB ESP,8”表明预留栈空间为 8 字节。

(2) 谁恢复栈的问题，即调用函数还是被调用函数弹出压栈参数。可以看到 cdecl_analysis、stdcall_analysis 函数都传入一个 int 型参数，但是函数返回时，cdecl_analysis 函数用 RETN 指令返回，对压栈参数占用的栈置若罔闻，而 stdcall_analysis 函数用 RETN 4 指令返回，回收了压栈参数占用的栈空间。

9.2 语句

9.2.1 表达式语句

```
/*
 * 功能：解析表达式语句
 */
void expression_statement()
{
    if(token != TK_SEMICOLON)
    {
        expression();
    }
}
```

```

    operand_pop();
}
skip(TK_SEMICOLON);
}

```

9.2.2 选择语句

```

/****************************************************************************
 * 功能：解析 if 语句
 * bsym: break 跳转位置
 * csym: continue 跳转位置
 ****
void if_statement(int *bsym, int *csym)
{
    int a, b;
    get_token();
    skip(TK_OPENPA);
    expression();
    skip(TK_CLOSEPA);
    a=gen_jcc(0);
    statement(bsym, csym);
    if(token==KW_ELSE)
    {
        get_token();
        b=gen_jmpforward(0);
        backpatch(a,ind);
        statement(bsym, csym);
        backpatch(b,ind); /* 反填 else 跳转 */
    }
    else
        backpatch(a,ind);
}

```

请大家结合表 9.2 中的例子来理解上面的代码。

表 9.2 if 语句生成目标语言举例

SC 语言	内存地址	x86 机器语言	汇编语言
void if_stmt_analysis()	0040121D	55	PUSH EBP
{	0040121E	89E5	MOV EBP,ESP
int a,b,c;	00401220	81EC 0C000000	SUB ESP,0C
if(a>b)	00401226	8B45 F8	MOV EAX,DWORD PTR SS:[EBP-8]
	00401229	8B4D FC	MOV ECX,DWORD PTR SS:[EBP-4]
	0040122C	39C1	CMP ECX,EAX
	0040122E	0F8E 0B000000	JLE scc_anal. 0040123F
c=a;	00401234	8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]

续表

SC语言	内存地址	x86机器语言	汇编语言
else	00401237	8945 F4	MOV DWORD PTR SS:[EBP-C],EAX
c=b;	0040123A	E9 06000000	JMP scc_anal. 00401245
}	00401242	8B45 F8	MOV EAX,DWORD PTR SS:[EBP-8]
	00401245	8945 F4	MOV DWORD PTR SS:[EBP-C],EAX
	00401247	8BE5	MOV ESP,EBP
	00401248	5D	POP EBP
		C3	RETN

上面程序的执行流程图如图 9.1 所示,大家可以结合流程图想一下,生成的目标代码的真出口跳转、假出口跳转是如何实现的?

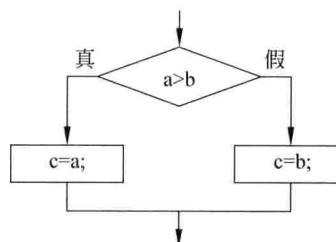


图 9.1 if 语句例子的执行流程图

9.2.3 循环语句

```

/*
 * 功能: 解析 for 语句
 * bsym: break 跳转位置
 * csym: continue 跳转位置
 */
void for_statement(int *bsym, int *csym)
{
    int a, b, c, d, e;
    get_token();
    skip(TK_OPENPA);
    if(token != TK_SEMICOLON)
    {
        expression();
        operand_pop();
    }
    skip(TK_SEMICOLON);
    d=ind;
    c=ind;
    a=0;
    b=0;
    if(token != TK_SEMICOLON)
    {

```

```

        expression();
        a=gen_jcc(0);
    }
    skip(TK_SEMICOLON);
    if(token !=TK_CLOSEPA)
    {
        e=gen_jmpforward(0);
        c=ind;
        expression();
        operand_pop();
        gen_jmpbackward(d);
        backpatch(e,ind);
    }
    skip(TK_CLOSEPA);
    statement(&a, &b);
    gen_jmpbackward(c);
    backpatch(a,ind);
    backpatch(b,c);
}

```

请大家结合表 9.3 中的例子来理解上面的代码。

表 9.3 for 语句生成目标语言举例

SC 语言	内存地址	x86 机器语言	汇编语言
void for_stmt_analysis()	00401249	55	PUSH EBP
{	0040124A	89E5	MOV EBP,ESP
int i,arr[10];	0040124C	81EC 2C000000	SUB ESP,2C
for(i=0;	00401252	B8 00000000	MOV EAX,0
i<10;	00401257	8945 FC	MOV DWORD PTR SS:[EBP-4],EAX
i=i+1)	0040125A	8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]
{	0040125D	83F8 0A	CMP EAX,0A
arr[i]=i;	00401260	0F8D 27000000	JGE scc_anal. 0040128D
}	00401266	E9 0B000000	JMP scc_anal. 00401276
arr[i]=i;	0040126B	8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]
}	0040126E	83C0 01	ADD EAX,1
arr[i]=i;	00401271	8945 FC	MOV DWORD PTR SS:[EBP-4],EAX
}	00401274	EB E4	JMP SHORT scc_anal. 0040125A
arr[i]=i;	00401276	B8 04000000	MOV EAX,4
}	0040127B	8B4D FC	MOV ECX,DWORD PTR SS:[EBP-4]
arr[i]=i;	0040127E	0FAFC8	IMUL ECX,EAX
}	00401281	8D45 D4	LEA EAX,DWORD PTR SS:[EBP-2C]
arr[i]=i;	00401284	01C8	ADD EAX,ECX
}	00401286	8B4D FC	MOV ECX,DWORD PTR SS:[EBP-4]
arr[i]=i;	00401289	8908	MOV DWORD PTR DS:[EAX],ECX
}	0040128B	EB DE	JMP SHORT scc_anal. 0040126D
arr[i]=i;	0040128D	8BE5	MOV ESP,EBP
}	0040128F	5D	POP EBP
arr[i]=i;	00401290	C3	RETN

大家可以结合图 9.2 来理解上面生成的目标代码。

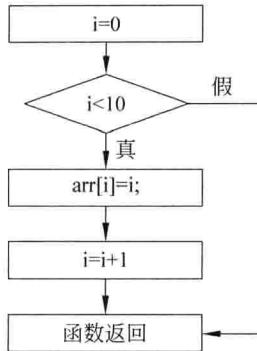


图 9.2 for 语句例子的执行流程图

9.2.4 跳转语句

```

/******************
 * 功能：解析 continue 语句
 * csym: continue 跳转位置
 *****/
void continue_statement(int *csym)
{
    if(!csym)
        error("此处不能用 continue");
    *csym=gen_jmpforward(*csym);
    get_token();
    skip(TK_SEMICOLON);
}

/******************
 * 功能：解析 break 语句
 * bsym: break 跳转位置
 *****/
void break_statement(int *bsym)
{
    if(!bsym)
        error("此处不能用 break");
    *bsym=gen_jmpforward(*bsym);
    get_token();
    skip(TK_SEMICOLON);
}

/******************
 * 功能：解析 return 语句
 *****/
void return_statement()

```

```

{
    get_token();

    if(token != TK_SEMICOLON)
    {
        expression();
        load_1(REG_IRET,optop);
        operand_pop();
    }
    skip(TK_SEMICOLON);
    rsym=gen_jmpforward(rsym);
}

/****************************************
 * 功能：记录待定跳转地址的指令链
 * s: 前一跳转指令地址
 *****/
int makelist(int s)
{
    int ind1;
    ind1=ind+4;
    if(ind1>sec_text->data_allocated)
        section_realloc(sec_text, ind1);
    *(int *) (sec_text->data+ind)=s;
    s=ind;
    ind=ind1;
    return s;
}

/****************************************
 * 功能：回填函数,把 t 为链首的各个待定跳转地址填入相对地址
 * t: 链首
 * a: 指令跳转位置
 *****/
void backpatch(int t, int a)
{
    int n, *ptr;
    while(t)
    {
        ptr=(int *) (sec_text->data+t);
        n= *ptr;                      //下一个需要回填位置
        *ptr=a-t-4;
        t=n;
    }
}

```

请大家结合表9.4中的例子来理解上面的代码。

表9.4 跳转语句生成目标语言举例

SC语言	内存地址	x86机器语言	汇编语言
int jump_analysis()	00401291	55	PUSH EBP
{	00401292	89E5	MOV EBP,ESP
int i,arr[10];	00401294	81EC 2C000000	SUB ESP,2C
for(i=0;	0040129A	B8 00000000	MOV EAX,0
i<10;	0040129F	8945 FC	MOV DWORD PTR SS:[EBP-4],EAX
}	004012A2	8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]
i=i+1)	004012A5	83F8 0A	CMP EAX,0A
{	004012A8	0F8D 5F000000	JGE scc_anal. 0040130D
if(i==2)	004012AE	E9 0B000000	JMP scc_anal. 004012BE
continue;	004012B3	8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]
if(i==6)	004012B6	83C0 01	ADD EAX,1
break;	004012B9	8945 FC	MOV DWORD PTR SS:[EBP-4],EAX
if(i==8)	004012BC	EB E4	JMP SHORT scc_anal. 004012A2
return 0;	004012BE	8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]
arr[i]=i;	004012C1	83F8 02	CMP EAX,2
}	004012C4	0F85 05000000	JNZ scc_anal. 004012CF
}	004012CA	E9 E4FFFFFF	JMP scc_anal. 004012B3
return 1;	004012CF	8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]
}	004012D2	83F8 06	CMP EAX,6
	004012D5	0F85 05000000	JNZ scc_anal. 004012E0
	004012DB	E9 2D000000	JMP scc_anal. 0040130D
	004012E0	8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]
	004012E3	83F8 08	CMP EAX,8
	004012E6	0F85 0A000000	JNZ scc_anal. 004012F6
	004012EC	B8 00000000	MOV EAX,0
	004012F1	E9 21000000	JMP scc_anal. 00401317
	004012F6	B8 04000000	MOV EAX,4
	004012FB	8B4D FC	MOV ECX,DWORD PTR SS:[EBP-4]
	004012FE	0FAFC8	IMUL ECX,EAX
	00401301	8D45 D4	LEA EAX,DWORD PTR SS:[EBP-2C]
	00401304	01C8	ADD EAX,ECX
	00401306	8B4D FC	MOV ECX,DWORD PTR SS:[EBP-4]
	00401309	8908	MOV DWORD PTR DS:[EAX],ECX
	0040130B	EB A6	JMP SHORT scc_anal. 004012B3
	0040130D	B8 01000000	MOV EAX,1
	00401312	E9 00000000	JMP scc_anal. 00401317
	00401317	8BE5	MOV ESP,EBP
	00401319	5D	POP EBP
	0040131A	C3	RETN

大家可以结合图9.3来理解上面生成的目标代码。

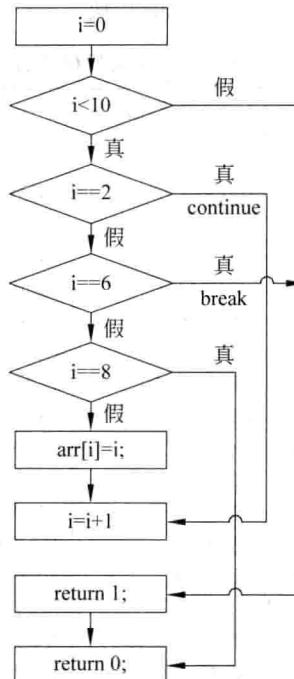


图 9.3 跳转语句例子的执行流程图

9.3 表达式

```
*****  
* 功能：解析表达式  
*****  
  
void expression()  
{  
    while(1)  
    {  
        assignment_expression();  
        if(token != ',' )  
            break;  
        operand_pop();  
        get_token();  
    }  
}
```

9.3.1 赋值表达式

```
/* 功能：解析赋值表达式  
***** */
```

```
void assignment_expression()
{
    equality_expression();
    if(token==TK_ASSIGN)
    {
        check_lvalue();
        get_token();
        assignment_expression();
        store0_1();
    }
}

/******************
 * 功能：检查栈顶操作数是否为左值
 *****************/
void check_lvalue()
{
    if(!(optop->r & SC_LVAL))
        expect("左值");
}
```

请大家结合表 9.5 中的例子来理解上面的代码。

表 9.5 赋值运算的生成目标语言举例

SC 语言	x86 机器语言	汇编 语 言
void assign_analysis()	55	PUSH EBP
{	89E5	MOV EBP,ESP
	81EC 0C000000	SUB ESP,0C
char a='a';	B8 61000000	MOV EAX,61
	8845 FF	MOV BYTE PTR SS:[EBP-1],AL
short b=6;	B8 06000000	MOV EAX,6
	66:8945 FE	MOV WORD PTR SS:[EBP-2],AX
int c=8;	B8 08000000	MOV EAX,8
	8945 FC	MOV DWORD PTR SS:[EBP-4],EAX
char str1[]{"abc";	B9 04000000	MOV ECX,4
	BE 00304000	MOV ESI,scc_anal.00403000; ASCII "abc"
	8D7D F8	LEA EDI,DWORD PTR SS:[EBP-8]
	F3:A4	REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
char * str2="XYZ";	B8 04304000	MOV EAX,scc_anal.00403004; ASCII "XYZ"
}	8945 F4	MOV DWORD PTR SS:[EBP-C],EAX
	8BE5	MOV ESP,EBP
	5D	POP EBP
	C3	RETN

9.3.2 相等类表达式

```
/* 功能：解析相等类表达式
```

```
*****
void equality_expression()
{
    int t;
    relational_expression();
    while(token==TK_EQ || token==TK_NEQ)
    {
        t=token;
        get_token();
        relational_expression();
        gen_op(t);
    }
}
```

请大家结合表 9.6 中的例子来理解上面的代码。

表 9.6 相等运算生成的目标语言举例

SC 语言	x86 机器语言	汇编语言
void equality_analysis()	55	PUSH EBP
{	89E5	MOV EBP,ESP
int a,b,c;	81EC 0C000000	SUB ESP,0C
c=a==b;	8B45 F8	MOV EAX,DWORD PTR SS:[EBP-8]
	8B4D FC	MOV ECX,DWORD PTR SS:[EBP-4]
	39C1	CMP ECX,EAX
	B8 00000000	MOV EAX,0
	0F94C0	SETE AL
	8945 F4	MOV DWORD PTR SS:[EBP-C],EAX
	8B45 F8	MOV EAX,DWORD PTR SS:[EBP-8]
	8B4D FC	MOV ECX,DWORD PTR SS:[EBP-4]
	39C1	CMP ECX,EAX
	B8 00000000	MOV EAX,0
	0F95C0	SETNE AL
	8945 F4	MOV DWORD PTR SS:[EBP-C],EAX
	8BE5	MOV ESP,EBP
	5D	POP EBP
	C3	RETN

9.3.3 关系表达式

```
*****
* 功能：解析关系表达式
*****
void relational_expression()
{
    int t;
```

```

additive_expression();
while((token==TK_LT || token==TK_LEQ) ||
      token==TK_GT || token==TK_GEQ)
{
    t=token;
    get_token();
    additive_expression();
    gen_op(t);
}
}

```

请大家结合表9.7中的例子来理解上面的代码。

表9.7 关系运算生成的目标语言举例

SC语言	x86机器语言	汇编语言
void relation_analysis()	55	PUSH EBP
{	89E5	MOV EBP,ESP
int a,b,c;	81EC 0C000000	SUB ESP,0C
c=a>b;	8B45 F8	MOV EAX,DWORD PTR SS:[EBP-8]
	8B4D FC	MOV ECX,DWORD PTR SS:[EBP-4]
	39C1	CMP ECX,EAX
	B8 00000000	MOV EAX,0
	0F9FC0	SETG AL
	8945 F4	MOV DWORD PTR SS:[EBP-C],EAX
c=a>=b;	8B45 F8	MOV EAX,DWORD PTR SS:[EBP-8]
	8B4D FC	MOV ECX,DWORD PTR SS:[EBP-4]
	39C1	CMP ECX,EAX
	B8 00000000	MOV EAX,0
	0F9DC0	SETGE AL
	8945 F4	MOV DWORD PTR SS:[EBP-C],EAX
c=a<b;	8B45 F8	MOV EAX,DWORD PTR SS:[EBP-8]
	8B4D FC	MOV ECX,DWORD PTR SS:[EBP-4]
	39C1	CMP ECX,EAX
	B8 00000000	MOV EAX,0
	0F9CC0	SETL AL
	8945 F4	MOV DWORD PTR SS:[EBP-C],EAX
c=a<=b;	8B45 F8	MOV EAX,DWORD PTR SS:[EBP-8]
}	8B4D FC	MOV ECX,DWORD PTR SS:[EBP-4]
	39C1	CMP ECX,EAX
	B8 00000000	MOV EAX,0
	0F9EC0	SETLE AL
	8945 F4	MOV DWORD PTR SS:[EBP-C],EAX
	8BE5	MOV ESP,EBP
	5D	POP EBP
	C3	RETN

9.3.4 加减类表达

```
/*
 * 功能：解析加减类表达式
 */
void additive_expression()
{
    int t;
    multiplicative_expression();
    while(token==TK_PLUS || token==TK_MINUS)
    {
        t=token;
        get_token();
        multiplicative_expression();
        gen_op(t);
    }
}
```

请大家结合表 9.8 中的例子来理解上面的代码。

表 9.8 加法、减法运算生成目标语言举例

SC 语言	x86 机器语言	汇编语言
void add_analysis() { int a,b,c,d,e; c=a+b;	55 89E5 81EC 14000000 8B45 F8 8B4D FC 01C1 894D F4	PUSH EBP MOV EBP,ESP SUB ESP,14 MOV EAX,DWORD PTR SS:[EBP-8] MOV ECX,DWORD PTR SS:[EBP-4] ADD ECX,EAX MOV DWORD PTR SS:[EBP-C],ECX
d=a+8;	8B45 FC 83C0 08 8945 F0	MOV EAX,DWORD PTR SS:[EBP-4] ADD EAX,8 MOV DWORD PTR SS:[EBP-10],EAX
e=6+8;	B8 06000000 83C0 08 8945 EC	MOV EAX,6 ADD EAX,8 MOV DWORD PTR SS:[EBP-14],EAX
c=a-b;	8B45 F8 8B4D FC 29C1 894D F4	MOV EAX,DWORD PTR SS:[EBP-8] MOV ECX,DWORD PTR SS:[EBP-4] SUB ECX,EAX MOV DWORD PTR SS:[EBP-C],ECX
d=a-8;	8B45 FC 83E8 08 8945 F0	MOV EAX,DWORD PTR SS:[EBP-4] SUB EAX,8 MOV DWORD PTR SS:[EBP-10],EAX
e=6+8;	B8 06000000 83C0 08 8945 EC 8BE5 5D C3	MOV EAX,6 ADD EAX,8 MOV DWORD PTR SS:[EBP-14],EAX MOV ESP,EBP POP EBP RETN

9.3.5 乘除类表达式

```
/*
 * 功能：解析乘除类表达式
 */
void multiplicative_expression()
{
    int t;
    unary_expression();
    while(token==TK_STAR || token==TK_DIVIDE || token==TK_MOD)
    {
        t=token;
        get_token();
        unary_expression();
        gen_op(t);
    }
}
```

请大家结合表9.9中的例子来理解上面的代码。

表9.9 乘法运算生成目标语言举例

SC语言	x86机器语言	汇编语言
void mul_analysis() { int a,b,c,d,e; c=a * b;	55 89E5 81EC 14000000 8B45 F8 8B4D FC 0FAFC8 894D F4	PUSH EBP MOV EBP,ESP SUB ESP,14 MOV EAX,DWORD PTR SS:[EBP-8] MOV ECX,DWORD PTR SS:[EBP-4] IMUL ECX,EAX MOV DWORD PTR SS:[EBP-C],ECX
d=a * 8;	B8 08000000 8B4D FC 0FAFC8 894D F0	MOV EAX,8 MOV ECX,DWORD PTR SS:[EBP-4] IMUL ECX,EAX MOV DWORD PTR SS:[EBP-10],ECX
e=6 * 8;	B8 08000000 B9 06000000 0FAFC8 894D EC	MOV EAX,8 MOV ECX,6 IMUL ECX,EAX MOV DWORD PTR SS:[EBP-14],ECX
c=a/b;	8B4D F8 8B45 FC 99 F7F9 8945 F4	MOV ECX,DWORD PTR SS:[EBP-8] MOV EAX,DWORD PTR SS:[EBP-4] CDQ IDIV ECX MOV DWORD PTR SS:[EBP-C],EAX

续表

SC 语 言	x86 机 器 语 言	汇 编 语 言
d=a/8;	B9 08000000 8B45 FC 99 F7F9 8945 F0	MOV ECX,8 MOV EAX,DWORD PTR SS:[EBP-4] CDQ IDIV ECX MOV DWORD PTR SS:[EBP-10],EAX
e=6/8;	B9 08000000 B8 06000000 99 F7F9 8945 EC	MOV ECX,8 MOV EAX,6 CDQ IDIV ECX MOV DWORD PTR SS:[EBP-14],EAX
c=a%b;	8B4D F8 8B45 FC 99 F7F9 8955 F4	MOV ECX,DWORD PTR SS:[EBP-8] MOV EAX,DWORD PTR SS:[EBP-4] CDQ IDIV ECX MOV DWORD PTR SS:[EBP-C],EDX
d=a%8;	B9 08000000 8B45 FC 99 F7F9 8955 F0	MOV ECX,8 MOV EAX,DWORD PTR SS:[EBP-4] CDQ IDIV ECX MOV DWORD PTR SS:[EBP-10],EDX
e=6%8; }	B9 08000000 B8 06000000 99 F7F9 8955 EC 8BE5 5D C3	MOV ECX,8 MOV EAX,6 CDQ IDIV ECX MOV DWORD PTR SS:[EBP-14],EDX MOV ESP,EBP POP EBP RETN

9.3.6 一元表达式

```
/*
 * 功能：解析一元表达式
 */
void unary_expression()
{
    switch(token)
    {
        case TK_AND:
            get_token();
            unary_expression();
    }
}
```

```

if((optop->type.t & T_BTYPE) != T_FUNC &&
   !(optop->type.t & T_ARRAY))
    cancel_lvalue();
mk_pointer(&optop->type);
break;

case TK_STAR:
get_token();
unary_expression();
indirection();
break;

case TK_PLUS:
get_token();
unary_expression();
break;

case TK_MINUS:
get_token();
operand_push(&int_type, SC_GLOBAL, 0);
unary_expression();
gen_op(TK_MINUS);
break;

case KW_SIZEOF:
sizeof_expression();
break;

default:
postfix_expression();
break;
}

}

/***** 功能：取消栈顶操作数的左值属性 *****
* 亦即得到栈顶操作数地址,因为后面生成代码时通过左值属性来判断是地址还是数据
***** */

void cancel_lvalue()
{
    check_lvalue();
    optop->r &= ~SC_LVAL;
}

/***** 功能：生成指针类型 *****
* t: 原数据类型
***** */

```

```

void mk_pointer(Type * t)
{
    Symbol * s;
    s=sym_push(SC_ANOM, t, 0, -1);
    t->t=T_PTR;
    t->ref=s;
}

/*********************************************
功能: 间接寻址
********************************************/
void indirection()
{
    if((optop->type.t & T_BTYPE) != T_PTR)
    {
        if((optop->type.t & T_BTYPE) == T_FUNC)
            return;
        expect("指针");
    }
    if((optop->r & SC_LVAL))
        load_1(REG_ANY,optop);
    optop->type= * pointed_type(&optop->type);

    //数组与函数不能为左值
    if(! (optop->type.t & T_ARRAY)
       && (optop->type.t & T_BTYPE) != T_FUNC)
    {
        optop->r |=SC_LVAL;
    }
}

/*********************************************
* 功能: 解析 sizeof 表达式
********************************************/
void sizeof_expression()
{
    int align, size;
    Type type;

    get_token();
    skip(TK_OPENPA);
    typeSpecifier(&type);
    skip(TK_CLOSEPA);

    size=type_size(&type, &align);
}

```

```

if(size<0)
    error("sizeof计算类型尺寸失败");
operand_push(&int_type, SC_GLOBAL, size);
}

```

请大家结合表 9.10 中的例子来理解上面的代码。

表 9.10 一元运算生成目标语言举例

SC 语言	x86 机器语言	汇编语言
void unary_analysis()		
{	55	PUSH EBP
int a, * pa,n;	89E5	MOV EBP,ESP
a=+8;	81EC 0C000000 B8 08000000 8945 FC	SUB ESP,0C MOV EAX,8 MOV DWORD PTR SS:[EBP-4],EAX
a=-8;	B8 00000000 83E8 08 8945 FC	MOV EAX,0 SUB EAX,8 MOV DWORD PTR SS:[EBP-4],EAX
a= * pa;	8B45 F8 8B08 894D FC	MOV EAX,DWORD PTR SS:[EBP-8] MOV ECX,DWORD PTR DS:[EAX] MOV DWORD PTR SS:[EBP-4],ECX
pa=&a;	8D45 FC 8945 F8	LEA EAX,DWORD PTR SS:[EBP-4] MOV DWORD PTR SS:[EBP-8],EAX
n=sizeof(int);	B8 04000000 8945 F4 8BE5 5D C3	MOV EAX,4 MOV DWORD PTR SS:[EBP-C],EAX MOV ESP,EBP POP EBP RETN
}		

9.3.7 后缀表达式

```

/******************
* 功能：解析后缀表达式
*****************/
void postfix_expression()
{
    Symbol * s;
    primary_expression();
    while(1)
    {
        if(token==TK_DOT || token==TK_POINTSTO)
        {
            if(token==TK_POINTSTO)
                indirection();
        }
    }
}

```

```

cancel_lvalue();
get_token();
if((optop->type.t & T_BTYPE) != T_STRUCT)
    expect("结构体变量");
s=optop->type.ref;
token |= SC_MEMBER;
while((s=s->next) != NULL)
{
    if(s->v==token)
        break;
}
if(!s)
    error("没有此成员变量:%s", get_tkstr(token & ~SC_MEMBER));
/* 成员变量地址=结构变量指针+成员变量偏移 */
optop->type=char_pointer_type;
operand_push(&int_type, SC_GLOBAL, s->c);
gen_op(TK_PLUS);
/* 变换类型为成员变量数据类型 */
optop->type=s->type;
/* 数组变量不能充当左值 */
if(!(optop->type.t & T_ARRAY))
{
    optop->r |= SC_LVAL;
}
get_token();
}
else if(token==TK_OPENBR)
{
    get_token();
    expression();
    gen_op(TK_PLUS);
    indirection();
    skip(TK_CLOSEBR);
}
else if(token==TK_OPENPA)
{
    argument_expression_list();
}
else
    break;
}
}

```

请大家结合表9.11中的例子来理解上面的代码。

表9.11 后缀表达式生成目标语言举例

SC语言	x86机器语言	汇编语言
void postfix_analysis()	55	PUSH EBP
{	89E5	MOV EBP,ESP
struct point	81EC 34000000	SUB ESP,34
{		
int m_x;		
int m_y;		
};		
int x;		
struct point pt;		
int arr[10];		
x=pt.m_x;	B8 01000000 B9 00000000 0FAFC8 8D45 F4 01C8 8B08 894D FC	MOV EAX,1 MOV ECX,0 IMUL ECX,EAX LEA EAX,DWORD PTR SS:[EBP-C] ADD EAX,ECX MOV ECX,DWORD PTR DS:[EAX] MOV DWORD PTR SS:[EBP-4],ECX
x=arr[1];	B8 04000000 B9 01000000 0FAFC8 8D45 CC 01C8 8B08 894D FC 8BE5 5D C3	MOV EAX,4 MOV ECX,1 IMUL ECX,EAX LEA EAX,DWORD PTR SS:[EBP-34] ADD EAX,ECX MOV ECX,DWORD PTR DS:[EAX] MOV DWORD PTR SS:[EBP-4],ECX MOV ESP,EBP POP EBP RETN
}		

```
/*
 * 功能：解析实参表达式表
 */
void argument_expression_list()
{
    Operand ret;
    Symbol * s, * sa;
    int nb_args;
    s=optop->type.ref;
    get_token();
    sa=s->next;           //first parameter
    nb_args=0;
    ret.type=s->type;
    ret.r=REG_IRET;
    ret.value=0;
    if(token !=TK_CLOSEPA)
    {

```

```

    for(;;)
    {
        assignment_expression();
        nb_args++;
        if(sa)
            sa=sa->next;
        if(token==TK_CLOSEPA)
            break;
        skip(TK_COMMA);
    }
    if(sa)
        error("实参数少于函数形参数");
    skip(TK_CLOSEPA);
    gen_invoke(nb_args);
    /* 返回值 */
    operand_push(&ret.type, ret.r, ret.value);
}

```

请大家结合表 9.12 中的例子来理解上面的代码。

表 9.12 函数调用生成目标语言举例

SC 语言	内存地址	x86 机器语言	汇编语言
int add(int x, int y)	0040131B	55	PUSH EBP
{	0040131C	89E5	MOV EBP,ESP
return x+y;	0040131E	81EC 00000000	SUB ESP,0
}	00401324	8B45 0C	MOV EAX,DWORD PTR SS:[EBP+C]
	00401327	8B4D 08	MOV ECX,DWORD PTR SS:[EBP+8]
	0040132A	01C1	ADD ECX,EAX
	0040132C	E9 00000000	JMP scc_anal. 00401331
00401331	8BE5		MOV ESP,EBP
	00401333	5D	POP EBP
	00401334	C3	RETN
void argument_analysis()	00401335	55	PUSH EBP
{	00401336	89E5	MOV EBP,ESP
int a,b,c;	00401338	81EC 0C000000	SUB ESP,0C
c=add(a,b);	0040133E	8B45 F8	MOV EAX,DWORD PTR SS:[EBP-8]
	00401341	50	PUSH EAX
	00401342	8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]
	00401345	50	PUSH EAX
	00401346	E8 D0FFFFFF	CALL scc_anal. 0040131B
	0040134B	83C4 08	ADD ESP,8
	0040134E	8945 F4	MOV DWORD PTR SS:[EBP-C],EAX
printf("Hello");	00401351	B8 08304000	MOV EAX,scc_anal. 00403008 ; ASCII "Hello"
	00401356	50	PUSH EAX
	00401357	E8 01010000	CALL <JMP. &msvert. printf>
	0040135C	83C4 04	ADD ESP,4
}	0040135F	8BE5	MOV ESP,EBP
	00401361	5D	POP EBP
	00401362	C3	RETN

9.3.8 初值表达式

```
*****  
* 功能：解析初等表达式  
*****  
void primary_expression()  
{  
    int t, r, addr;  
    Type type;  
    Symbol * s;  
    Section * sec=NULL;  
  
    switch(token)  
    {  
        case TK_CINT:  
        case TK_CCHAR:  
            operand_push(&int_type, SC_GLOBAL, tkvalue);  
            get_token();  
            break;  
        case TK_CSTR:  
            t=T_CHAR;  
            type.t=t;  
            mk_pointer(&type);  
            type.t |=T_ARRAY;  
            sec=allocate_storage(&type, SC_GLOBAL, 2, 0, &addr);  
            var_sym_put(&type, SC_GLOBAL, 0, addr);  
            initializer(&type, addr, sec);  
            break;  
        case TK_OPENPA:  
            get_token();  
            expression();  
            skip(TK_CLOSEPA);  
            break;  
        default:  
            t=token;  
            get_token();  
            if(t<TK_IDENT)  
                expect("标识符或常量");  
            s=sym_search(t);  
            if(!s)  
            {  
                if(token !=TK_OPENPA)  
                    error("' %s '未声明\n", get_tkstr(t));  
            }  
    }  
}
```

```

        s=func_sym_push(t, &default_func_type);      //允许函数不声明,直接引用
        s->r=SC_GLOBAL | SC_SYM;
    }
    r=s->r;
    operand_push(&s->type, r, s->c);
    /* 符号引用,操作数必须记录符号地址 */
    if(optop->r & SC_SYM)
    {
        optop->sym=s;
        optop->value=0;
    }
    break;
}
}

```

请大家结合表 9.13 中的例子来理解上面的代码。

表 9.13 初值表达式生成目标语言举例

SC 语句	x86 机器语言	汇编语言
char g_char = 'a';	55	PUSH EBP
short g_short=123;	89E5	MOV EBP,ESP
int g_int =123456;	81EC 0C000000	SUB ESP,0C
char g_str1[]{"g_str1"};		
char * g_str2="g_str2";		
void primary_analysis()		
{		
char a='a';	B8 61000000 8845 FF	MOV EAX,61 MOV BYTE PTR SS:[EBP-1],AL
short b=8;	B8 08000000 66:8945 FE	MOV EAX,8 MOV WORD PTR SS:[EBP-2],AX
int c=6;	B8 06000000 8945 FC	MOV EAX,6 MOV DWORD PTR SS:[EBP-4],EAX
char str1[]{"str1"};	B9 05000000 BE 15304000 8D7D F7 F3:A4	MOV ECX,5 MOV ESI,scc_anal.00403015; ASCII "str1" LEA EDI,DWORD PTR SS:[EBP-9] REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
char * str2="str2";	B8 1A304000 8945 F4	MOV EAX,scc_anal.0040301A; ASCII "str2" MOV DWORD PTR SS:[EBP-C],EAX
a=g_char;	0FBEO5 00204000 8845 FF	MOVSX EAX,BYTE PTR DS:[402000] MOV BYTE PTR SS:[EBP-1],AL
b=g_short;	0FBF05 02204000 66:8945 FE	MOVSX EAX,WORD PTR DS:[402002] MOV WORD PTR SS:[EBP-2],AX
c=g_int;	8B05 04204000 8945 FC	MOV EAX,DWORD PTR DS:[402004] MOV DWORD PTR SS:[EBP-4],EAX
printf(g_str1);	B8 08204000 50	MOV EAX,scc_anal.00402008; ASCII "g_str1" PUSH EAX

续表

SC语言	x86机器语言	汇编语言
	E8 98000000	CALL <JMP. &-msvcrt.printf>
	83C4 04	ADD ESP,4
printf(g_str2);	8B05 10204000	MOV EAX,DWORD PTR :[402010];scc_anal.0040300E
	50	PUSH EAX
	E8 89000000	CALL <JMP. &-msvcrt.printf>
	83C4 04	ADD ESP,4
printf(str1);	8D45 F7	LEA EAX,DWORD PTR SS:[EBP-9]
	50	PUSH EAX
	E8 7D000000	CALL <JMP. &-msvcrt.printf>
	83C4 04	ADD ESP,4
printf(str2);	8B45 F4	MOV EAX,DWORD PTR SS:[EBP-C]
}	50	PUSH EAX
	E8 71000000	CALL <JMP. &-msvcrt.printf>
	83C4 04	ADD ESP,4
	8BE5	MOV ESP,EBP
	5D	POP EBP
	C3	RETN

9.4 成果展示

语义分析程序终于写完了,从第6章开始我们已经在黑暗中前行了很长一段时间,终于可以看到语义分析成果了,通过图9.4看一下HelloWorld.c编译过程。



图9.4 HelloWorld.c编译过程

可以看到,“HelloWorld.c编译成功”,生成了HelloWorld.obj目标程序,其中的内容在第7章、第8章已经从不同侧面解读过,本章的成果展示我们来立体化解读一下这个文件,看一下它是如何完整保存语义分析成果的(见图9.5)。

这张图立体化地表示了代码节、COFF重定位表、COFF符号表、COFF字符串表之间的关联关系,这张图看上去好像有些复杂,其实就是为了解决一个问题,“记录暂时还不能确定地址的符号引用,以便符号地址确定后调整为真实的符号地址”。

到这里SCC编译器就算大功告成了。但是语义分析生成的目标文件,还没有办法直接运行,接下来的活由链接器完成。

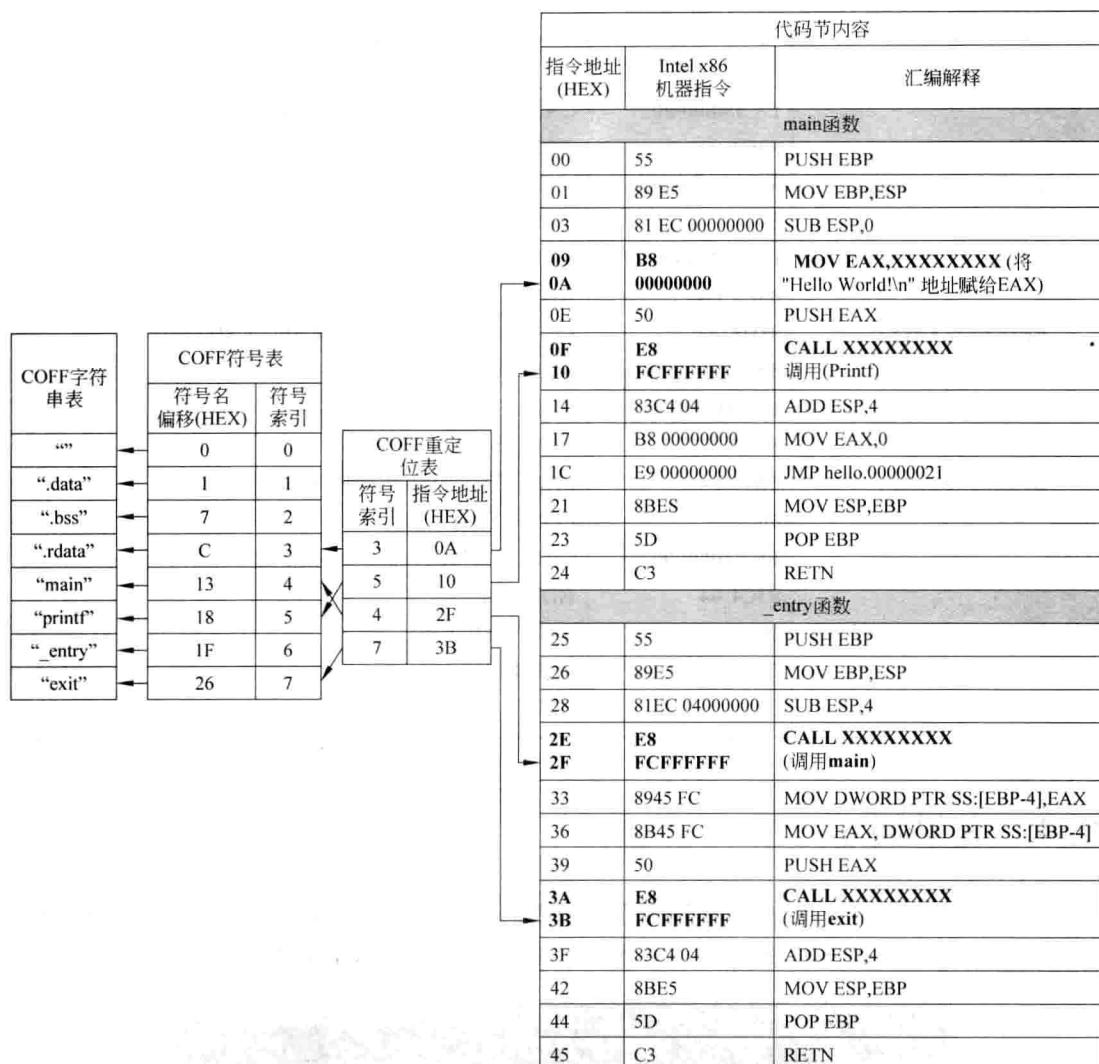


图 9.5 HelloWorld.obj 立体化解读

第 10 章

链 接 器

无限风光在险峰。

——毛泽东

SCC 编译器已经写完了,但似乎没有那种发自内心的激动与喜悦,因为生成的目标文件没法运行,所以看不到运行结果。如果 SCC 编译器就此止步,就太扫兴了。所以本章要实现一个简单的链接器,让编译器生成的成果真正能够运行起来。

链接器的功能是,将一个或多个由编译器生成的目标文件及库链接为一个可执行文件。链接器的工作就是解析未定义的符号引用,将目标文件中的占位符替换为符号的地址。链接器还要完成程序中各目标文件的地址空间的组织及重定位工作。

本章内容分 5 部分:第一部分介绍链接相关的一些基本概念,第二部分介绍链接生成的 PE 文件格式,第三部分介绍链接器的代码实现,第四部分介绍整个 SCC 编译器及链接的总控程序,第五部分是成果展示。

10.1 链接方式与库文件

链接有两种方式,一种是静态链接,另一种是动态链接,这两种链接方式各有好处。

所谓静态链接就是在编译链接时直接将需要的执行代码拷贝到调用处,优点是在程序发布的时候就不需要依赖库,也就是不再需要带着库一块发布,程序可以独立执行,但是体积可能会相对大一些。

所谓动态链接就是在编译的时候不直接拷贝可执行代码,而是通过记录一系列符号和参数,在程序运行或加载时将这些信息传递给操作系统,操作系统负责将需要的动态库加载到内存中,然后程序在运行到指定的代码时,去共享执行内存中已经加载的动态库可执行代码,最终达到运行时连接的目的。优点是多个程序可以共享同一段代码,而不需要在磁盘上存储多个拷贝,缺点是由于是运行时加载,可能会影响程序的前期执行性能。

上面的都是一些概念性的,那么对于在 Windows 操作系统中具体的实现方式是什么样的呢?这里引入另外一组概念,即静态库(Static Library)、动态链接库(Dynamic Linking Library,DLL)及引入库。

静态库,函数和数据被编译进一个二进制文件(通常扩展名为 LIB)。使用静态库的情况下,在链接生成可执行文件时,链接器从库中复制这些函数和数据并把它们和应用程序的其他模块组合起来创建最终的可执行文件(EXE 文件)。

在使用动态链接库的时候,往往提供两个文件:一个引入库(通常扩展名为 LIB)和一个 DLL。引入库包含被 DLL 导出的函数和变量的符号名,DLL 包含实际的函数和数据。

在链接生成可执行文件时,只需要链接引入库,DLL 中的函数代码和数据并不复制到可执行文件中,在运行的时候,再去加载 DLL,访问 DLL 中导出的函数。在运行 Windows 程序时,它通过一个称为“动态链接”的进程与 Windows 相接。一个 Windows 的 EXE 文件拥有它使用不同动态链接库的引用,所使用的函数即在那里。当 Windows 程序被加载到内存中时,程序中的调用被指向 DLL 函数的入口,如果 DLL 不在内存中,系统就将其加载到内存中。

这里要注意静态库与引入库在 Windows 操作系统下一般扩展名都为 LIB,但里面存的内容是有差异的。静态库的 LIB 文件包括符号表和二进制可执行代码,可以被静态连接。引入库的 LIB 文件只有符号表,也就是只有动态库的符号导出信息,通过这些信息可以在程序运行时定位到动态库中,最终实现动态连接。

本章链接器使用自定义的导入库格式,要求如下:导入库的文件名为相应 DLL 文件名,后缀为.slib(意思是简化的 LIB 库);文件内容为 DLL 中的所有导出函数,每个函数占文件一行。下面以 msvert.dll 的导入库 msvert.slib 为例来看一下导入库格式(见图 10.1)。

那么从 DLL 怎么生成相应的导入库呢?答案是通过 VC6 自带的工具 dumpbin.exe 程序来产生,通过图 10.2 看一下这个程序放在什么位置。

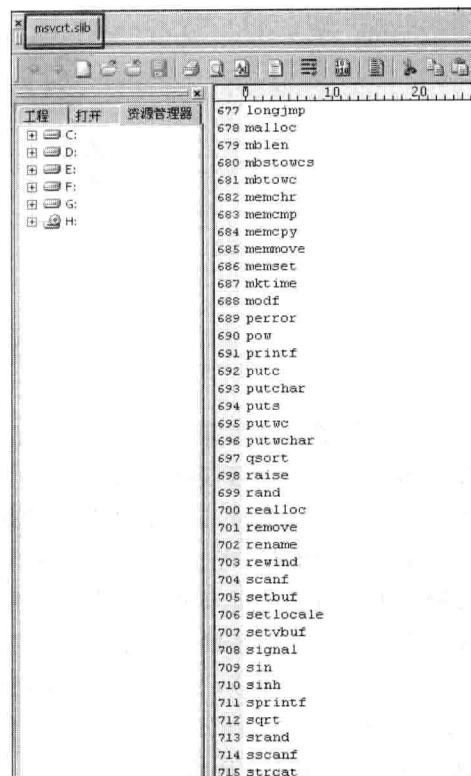


图 10.1 链接器使用的导入库格式

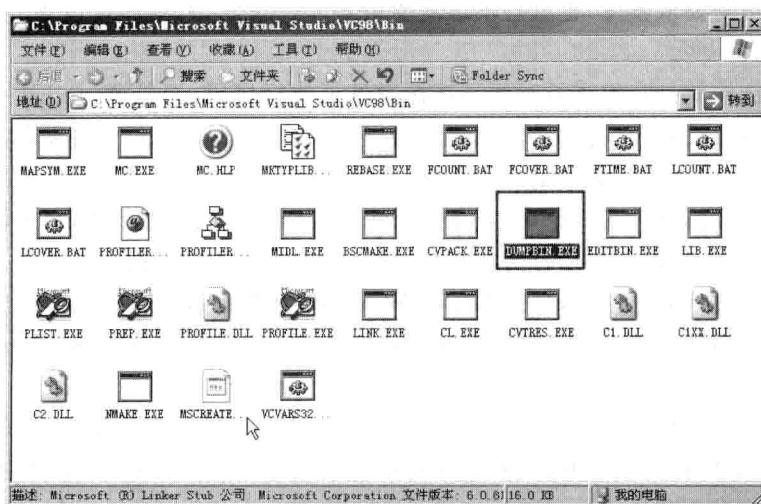


图 10.2 VC6 自带的 dumpbin.exe 工具程序

由 DLL 生成导入库的方式如图 10.3 所示,生成的导入库还要简单的手工处理一下才能得到上面所需要的导入库格式。



图 10.3 利用 dumpbin.exe 程序生成导入库

10.2 PE 文件格式

Windows 使用的可执行文件格式为 PE (Portable Executable) 格式,中文名称为可移植可执行文件格式。PE 格式与第 7 章讲的 COFF 文件格式有一些相同的格式内容,相同的部分本章将不再赘述,只给出需要参考第 7 章的那部分内容。

10.2.1 总体结构

PE 文件由 DOS 部分、NT 头、节头表、节数据组成,其总体结构如图 10.4 所示。

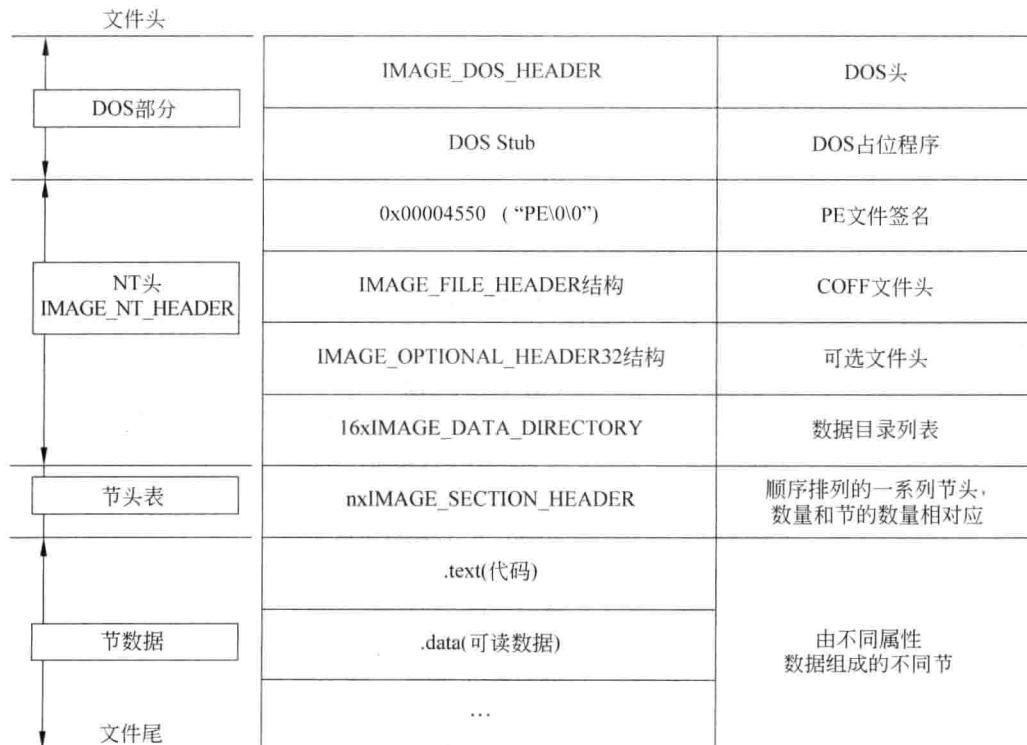


图 10.4 PE 文件的基本结构

10.2.2 DOS 部分

10.2.2.1 DOS 头

DOS 头格式如下,其中各成员变量见表 10.1。

```
typedef struct _IMAGE_DOS_HEADER {
    WORD    e_magic;
    WORD    e_cblp;
    WORD    e_cp;
    WORD    e_crlc;
    WORD    e_cparhdr;
    WORD    e_minalloc;
    WORD    e_maxalloc;
    WORD    e_ss;
    WORD    e_sp;
    WORD    e_csum;
    WORD    e_ip;
    WORD    e_cs;
    WORD    e_lfarlc;
    WORD    e_ovno;
    WORD    e_res[4];
    WORD    e_oemid;
    WORD    e_oeminfo;
    WORD    e_res2[10];
    LONG   e_lfanew;
} IMAGE_DOS_HEADER, * PIMAGE_DOS_HEADER;
```

表 10.1 DOS 头

偏移	大小	域	说 明
0	2	e_magic	EXE 标志,"MZ"
2	2	e_cblp	最后(部分)页中的字节数
4	2	e_cp	文件中的全部和部分页数
6	2	e_crlc	重定位表中的指针数
8	2	e_cparhdr	头部尺寸,以段落为单位
10	2	e_minalloc	所需的最小附加段
12	2	e_maxalloc	所需的最大附加段
14	2	e_ss	初始的 ss 值
16	2	e_sp	初始的 sp 值
18	2	e_csum	补码校验值
20	2	e_ip	初始的 IP 值
22	2	e_cs	初始化 CS 值
24	2	e_lfarlc	重定位表的字节偏移量
26	2	e_ovno	覆盖号
28	8	e_res	保留字

续表

偏移	大小	域	说 明
36	2	e_oemid	OEM 标志符
38	2	e_oemif	OEM 信息
40	20	e_res2	保留字
60	4	e_lfanew	NT 头(很多书上称 PE 头)相对于文件的偏移地址

HelloWorld.exe 的 DOS 头如图 10.5 所示。

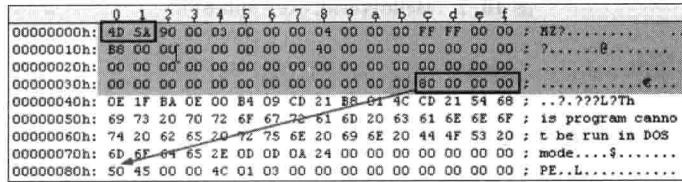


图 10.5 HelloWorld.exe (00h-8Fh)

对于 DOS 头我们只关注两个字段,可以看到 e_magic 为“MZ”, e_lfanew=0x80 通过这个字段可以定位 PE 头。e_lfanew 位于 0x3C 处,此信息的存在使得即使映像文件中有 10.2.2.2 节的 MS-DOS 占位程序,Windows 仍然能够正常执行它。这个文件偏移是在链接时被放在 0x3C 处的。

10.2.2.2 MS-DOS 占位程序

MS-DOS 占位程序是一个运行于 MS-DOS 下的合法应用程序,它被放在 EXE 映像的最前面。链接器在这里放一个默认的占位程序,当映像运行于 MS-DOS 时,这个占位程序显示 This program cannot be run in DOS mode 这条消息,翻译为中文为“此程序不能在 DOS 模式下运行”。

图 10.6 为 HelloWorld.exe 的 DOS 占位程序。

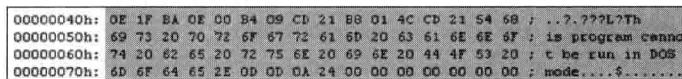


图 10.6 HelloWorld.exe (40h-7Fh)

10.2.3 NT 头

NT 头(又称 PE 头)包括 3 部分内容:PE 文件签名、COFF 文件头、COFF 可选文件头。其格式定义如下:

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

10.2.3.1 PE 文件签名

在 MS-DOS 占位程序后面、在偏移 0x3c 处指定的文件偏移处，是一个 4 字节的签名，它用来标识文件为一个 PE 格式的映像文件。这个签名是“PE\0\0”(字母 P 和 E 后跟着两个空字节)。

图 10.7 为 HelloWorld.exe 的 PE 签名。

```
00000080h: 50 45 00 00 4C 01 03 00 00 00 00 00 00 00 00 00 ; PE..L.....
```

图 10.7 HelloWorld.exe (80h-8Fh)

可以看到，HelloWorld.exe 中 PE 签名为“PE\0\0”。

10.2.3.2 COFF 文件头

COFF 文件头的格式定义及相关说明见 7.1.3 节，这里看一下 HelloWorld.exe 的 COFF 文件头。

图 10.8 为 HelloWorld.exe 的 COFF 文件头。

```
00000080h: 50 45 00 00 4C 01 03 00 00 00 00 00 00 00 00 00 ; PE..L.....  
00000090h: 00 00 00 00 E0 00 0F 03 0B 01 06 00 00 00 00 00 ; ....?....
```

图 10.8 HelloWorld.exe (80h-9Fh)

其中，

- Machine=0x14c，从 7.1.3.1 节可知，它表示机器类型为 386 或后继处理器及其兼容处理器。
- NumberOfSections=3，代表文件中包含 3 个节。
- SizeOfOptionalHeader=0xE0，代表可选文件头的大小为 0xE0。
- Characteristics=0x30F

= IMAGE_FILE_RELOCS_STRIPPED	//0x0001 不包含基址重定位信息
IMAGE_FILE_EXECUTABLE_IMAGE	//0x0002 可以被运行
IMAGE_FILE_LINE_NUMS_STRIPPED	//0x0004 COFF 行号信息已经被移除
IMAGE_FILE_LOCAL_SYMS_STRIPPED	//0x0008 COFF 符号表中有关局部符的项已经被移除
IMAGE_FILE_32BIT_MACHINE	//0x0100 机器类型基于 32 位字体系结构
IMAGE_FILE_DEBUG_STRIPPED	//0x020 调试信息已经从此映像文件中移除

10.2.3.3 COFF 可选文件头

每个映像文件都有一个可选文件头，用于为加载器提供信息。这个文件头之所以说是可选的，是相对于目标文件并不包含它这层意义上来说的。对于映像文件来说，这个文件头是必须的。

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    //标准域
    WORD Magic;
```

```

    BYTE      MajorLinkerVersion;
    BYTE      MinorLinkerVersion;
    DWORD     SizeOfCode;
    DWORD     SizeOfInitializedData;
    DWORD     SizeOfUninitializedData;
    DWORD     AddressOfEntryPoint;
    DWORD     BaseOfCode;
    DWORD     BaseOfData;

    //特定域
    DWORD     ImageBase;
    DWORD     SectionAlignment;
    DWORD     FileAlignment;
    WORD      MajorOperatingSystemVersion;
    WORD      MinorOperatingSystemVersion;
    WORD      MajorImageVersion;
    WORD      MinorImageVersion;
    WORD      MajorSubsystemVersion;
    WORD      MinorSubsystemVersion;
    DWORD     Win32VersionValue;
    DWORD     SizeOfImage;
    DWORD     SizeOfHeaders;
    DWORD     CheckSum;
    WORD      Subsystem;
    WORD      DllCharacteristics;
    DWORD     SizeOfStackReserve;
    DWORD     SizeOfStackCommit;
    DWORD     SizeOfHeapReserve;
    DWORD     SizeOfHeapCommit;
    DWORD     LoaderFlags;
    DWORD     NumberOfRvaAndSizes;
    //数据目录
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;

```

1. 可选文件头中的标准域

可选文件头的前 8 个域为标准域，它们被所有 COFF 实现所定义。这些域包含对加载和运行可执行文件有用的常规信息（见表 10.2）。

表 10.2 可选文件头中的标准域

Offset	Size	Field	Description
0	2	Magic	这个无符号整数指出了映像文件的状态。最常用的数字是 0x10B，它表明这是一个正常的可执行文件。0x107 表明这是一个 ROM 映像，0x20B 表明这是一个 PE32+ 可执行文件
2	1	MajorLinkerVersion	链接器的主版本号

续表

Offset	Size	Field	Description
3	1	MinorLinkerVersion	链接器的次版本号
4	4	SizeOfCode	代码节(.text)的大小。如果有多个代码节,它是所有代码节的和
8	4	SizeOfInitializedData	已初始化数据节的大小。如果有多个这样的数据节,它是所有这些数据节的和
12	4	SizeOfUninitializedData	未初始化数据节(.bss)的大小。如果有多个.bss节,它是所有这些节的和
16	4	AddressOfEntryPoint	当可执行文件被加载进内存时其入口点相对于映像基址的偏移地址。对于一般程序映像来说,它就是启动地址。对于设备驱动程序来说,它是初始化函数的地址。入口点对于 DLL 来说是可选的。如果不存在入口点,这个域必须为 0
20	4	BaseOfCode	当映像被加载进内存时代码节的开头相对于映像基址的偏移地址
24	4	BaseOfData	当映像被加载进内存时数据节的开头相对于映像基址的偏移地址

图 10.9 为 HelloWorld.exe 的可选文件头标准域。

000000090h: 00 00 00 00 E0 00 0F 03 0B 01 06 00 00 00 00 00 00 ; ... , ? ,
0000000A0h: 00 00 00 00 00 00 00 25 10 00 00 00 10 00 00 ; %
0000000B0h: 00 00 00 00 00 00 40 00 00 10 00 00 00 02 00 00 ; 0

图 10.9 HelloWorld.exe (90h-bFh)

其中,

- Magic=0x10B, 表明这是一个正常的可执行文件。
- AddressOfEntryPoint=0x1025, 表示当可执行文件被加载进内存时其入口点相对于映像基址的偏移地址。
- BaseOfCode=0x1000, 表示当映像文件被加载进内存时代码节的开头相对于映像基址的偏移地址。

2. 可选文件头中的 Windows 特定域

接下来的 21 个域是 COFF 可选文件头格式的扩展。它们包含 Windows 中的链接器和加载器所必需的附加信息(见表 10.3)。

表 10.3 可选文件头中的 Windows 特定域

偏移	大小	域	说 明
28	4	ImageBase	当加载进内存时映像的第一个字节的首选地址。它必须是 64K 的倍数。DLL 默认是 0x10000000。Windows CE EXE 默认是 0x00010000。Windows NT、Windows 2000、Windows XP、Windows 95、Windows 98 和 Windows Me 默认是 0x00400000

续表

偏移	大小	域	说 明
32	4	SectionAlignment	当加载进内存时节的对齐值(以字节计)。它必须大于或等于 FileAlignment。默认是相应系统的页面大小
36	4	FileAlignment	用来对齐映像文件的节中的原始数据的对齐因子(以字节计)。它应该是界于 2^8 和 2^{16} 之间的 2 的幂(包括这两个边界值)。默认是 512。如果 SectionAlignment 小于相应系统的页面大小, 那么 FileAlignment 必须与 SectionAlignment 匹配
40	2	MajorOperatingSystemVersion	所需操作系统的主版本号
42	2	MinorOperatingSystemVersion	所需操作系统的次版本号
44	2	MajorImageVersion	映像的主版本号
46	2	MinorImageVersion	映像的次版本号
48	2	MajorSubsystemVersion	子系统的主版本号
50	2	MinorSubsystemVersion	子系统的次版本号
52	4	Win32VersionValue	保留, 必须为 0
56	4	SizeOfImage	当映像被加载进内存时的大小(以字节计), 包括所有的文件头。它必须是 SectionAlignment 的倍数
60	4	SizeOfHeaders	MS-DOS 占位程序、PE 文件头和节头的总大小, 向上舍入为 FileAlignment 的倍数
64	4	CheckSum	映像文件的校验和。计算校验和的算法被合并到了 IMAGEHLP.DLL 中。以下程序在加载时被校验以确定其是否合法: 所有的驱动程序、任何在引导时被加载的 DLL 以及加载进关键 Windows 进程中的 DLL
68	2	Subsystem	运行此映像所需的子系统。要获取更多信息, 请参考后面的“Windows 子系统”部分
70	2	DllCharacteristics	DLL 特征
72	4	SizeOfStackReserve	保留的堆栈大小。只有 SizeOfStackCommit 指定的部分被提交; 其余的每次可用一页, 直到到达保留的大小为止
76	4	SizeOfStackCommit	提交的堆栈大小
80	4	SizeOfHeapReserve	保留的局部堆空间大小。只有 SizeOfHeapCommit 指定的部分被提交; 其余的每次可用一页, 直到到达保留的大小为止
84	4	SizeOfHeapCommit	提交的局部堆空间大小
88	4	LoaderFlags	保留, 必须为 0
92	4	NumberOfRvaAndSizes	可选文件头其余部分中数据目录项的个数, 每个数据目录描述了一个表的位置和大小

可选文件头的 Subsystem 域定义了以下值以确定运行映像所需的 Windows 子系统(见表 10.4)。

表 10.4 Windows 子系统

常量	值	说明
IMAGE_SUBSYSTEM_UNKNOWN	0	未知子系统
IMAGE_SUBSYSTEM_NATIVE	1	设备驱动程序和 Native Windows 进程
IMAGE_SUBSYSTEM_WINDOWS_GUI	2	图形用户界面(GUI)子系统
IMAGE_SUBSYSTEM_WINDOWS_CUI	3	字符模式(CUI)子系统
IMAGE_SUBSYSTEM_POSIX_CUI	7	字符模式子系统
IMAGE_SUBSYSTEM_WINDOWS_CE_GUI	9	Windows CE
IMAGE_SUBSYSTEM_EFI_APPLICATION	10	可扩展固件接口(EFI)应用程序
IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER	11	带引导服务的 EFI 驱动程序
IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER	12	带运行时服务的 EFI 驱动程序
IMAGE_SUBSYSTEM_EFI_ROM	13	EFI ROM 映像
IMAGE_SUBSYSTEM_XBOX	14	XBOX

图 10.10 为 HelloWorld.exe 的可选文件头中的 Windows 特定域。

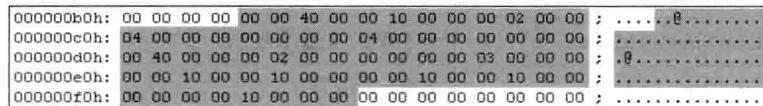


图 10.10 HelloWorld.exe (b0h-ffh)

其中，

- ImageBase=0x00400000, 表示当映像文件加载进内存时第一个字节的首选地址。
- SectionAlignment=0x1000, 表示当映像文件加载进内存时节的对齐值为 0x1000。
- FileAlignment=0x200, 表示映像文件的节中的原始数据的对齐因子为 0x200。
- Subsystem=3=IMAGE_SUBSYSTEM_WINDOWS_CUI, 表示运行此映像所需的子系统为字符模式(CUI)子系统。
- NumberOfRvaAndSizes=0x10=16, 表示数据目录项有 16 个。

3. 可选文件头中的数据目录

每个数据目录给出了 Windows 使用的表或字符串的地址和大小。这些数据目录项全部被加载进内存以备系统运行时使用。数据目录是按照如下格式定义的一个 8 字节结构：

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD VirtualAddress;
    DWORD Size;
} IMAGE_DATA_DIRECTORY, * PIMAGE_DATA_DIRECTORY;
```

第一个域——VirtualAddress, 实际上是表的 RVA。这个 RVA 是当表被加载进内存时相对于映像基址的偏移地址。第二个域给出了表的大小(以字节计)。数据目录组成了可选文件头的最后一部分, 它被列于表 10.5 中。

注意目录的数目是不固定的。在查看一个特定的目录之前, 先检查可选文件头中的 NumberOfRvaAndSizes 域。同样, 不要想当然地认为在这个表中的 RVA 指向节的开头, 或者认为包含特定表的节有特定的名字。

表 10.5 可选文件头中的数据目录

Offset	Size	Field	Description
96	8	Export Table	导出表的地址和大小
104	8	Import Table	导入表的地址和大小。要获取更多信息,请参考 10.2.10 节“.idata 节”
112	8	Resource Table	资源表的地址和大小
120	8	Exception Table	异常表的地址和大小
128	8	Certificate Table	属性证书表的地址和大小
136	8	Base Relocation Table	基址重定位表的地址和大小
144	8	Debug	调试数据起始地址和大小
152	8	Architecture	保留,必须为 0
160	8	Global Ptr	将被存储在全局指针寄存器中的一个值的 RVA,这个结构的 Size 域必须为 0
168	8	TLS Table	线程局部存储(TLS)表的地址和大小
176	8	Load Config Table	加载配置表的地址和大小
184	8	Bound Import	绑定导入表的地址和大小
192	8	IAT	导入地址表的地址和大小。要获取更多信息,请参考 10.2.10.4 节“导入地址表”
200	8	Delay Import Descriptor	延迟导入描述符的地址和大小
208	8	CLR Runtime Header	TCLR 运行时头部的地址和大小
216	8		保留,必须为 0

```

#define IMAGE_DIRECTORY_ENTRY_EXPORT          0    //Export Directory
#define IMAGE_DIRECTORY_ENTRY_IMPORT          1    //Import Directory
#define IMAGE_DIRECTORY_ENTRY_RESOURCE        2    //Resource Directory
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION       3    //Exception Directory
#define IMAGE_DIRECTORY_ENTRY_SECURITY        4    //Security Directory
#define IMAGE_DIRECTORY_ENTRY_BASERELOC      5    //Base Relocation Table
#define IMAGE_DIRECTORY_ENTRY_DEBUG           6    //Debug Directory
//      IMAGE_DIRECTORY_ENTRY_COPYRIGHT      7    //(X86 usage)
#define IMAGE_DIRECTORY_ENTRY_ARCHITECTURE    7    //Architecture Specific Data
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR       8    //RVA of GP
#define IMAGE_DIRECTORY_ENTRY_TLS              9    //TLS Directory
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG     10   //Load Configuration Directory
#define IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT    11   //Bound Import Directory in headers
#define IMAGE_DIRECTORY_ENTRY_IAT              12   //Import Address Table
#define IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT    13   //Delay Load Import Descriptors
#define IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR  14   //COM Runtime descriptor

```

图 10.11 为 HelloWorld.exe 的可选文件头中的数据目录。

可以看到,只有导入表和导入地址表中有数据,表 10.6 为这两个数据目录表项。

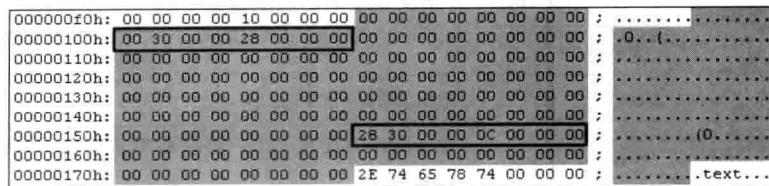


图 10.11 HelloWorld.exe (f0h-17fh)

表 10.6 HelloWorld.exe 导入表目录和导入地址表目录

IMAGE_DATA_DIRECTORY DataDirectory	VirtualAddress	Size
IMAGE_DIRECTORY_ENTRY_IMPORT	0x3000	0x28
IMAGE_DIRECTORY_ENTRY_IAT	0x3028	0xC

10.2.4 节头表

节头表的格式定义及相关说明见 7.1.4 节, 图 10.12 为 HelloWorld.exe 的节头表。

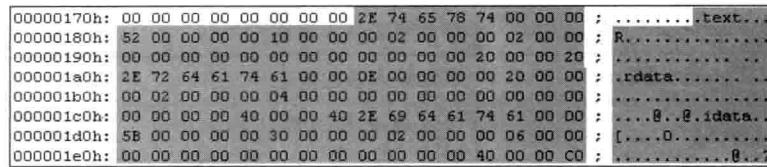


图 10.12 HelloWorld.exe (170h-1efh)

可以看到,HelloWorld.exe 节头表中有 3 项,分别是.text..rdata..idata,我们研究一下这些节用到的关键属性字段(见表 10.7)。

表 10.7 HelloWorld.exe 节头表解读

Name	.text(代码节)	.rdata(只读数据节)	.idata(导入节)
VirtualSize	0x52	0xE	0x5B
VirtualAddress	0x1000	0x2000	0x3000
SizeOfRawData	0x200	0x200	0x200
PointerToRawData	0x200	0x400	0x600
Characteristics	0x20000020	0x40000040	0xC0000040

10.2.5 代码节

代码节存储的是可执行代码,图 10.13 为 HelloWorld.exe 代码节的内容。

从图 10.13 可知,代码节内容是从 0x200 开始的连续 0x200 个字节,这与 10.2.4 节介绍的节头表中内容能够对应起来,另外大家注意到没有,代码节后面的内容都是用 0 填充的,这就涉及前面介绍的可选文件头中的 FileAlignment 字段,由于它的值为 0x200,所以在文件中以 0x200 粒度对齐。下面来解读一下代码节中有意义的内容(见表 10.8)。

00000200h:	55 89 E5 81 EC 00 00 00 00 B8 00 20 40 00 50 E8	; U文孩....? 0.P?
00000210h:	32 00 00 00 B3 C4 04 B8 00 00 00 E9 00 00 00	; 2...瓶,?...?..
00000220h:	00 0B E5 5D C3 55 89 E5 81 EC 04 00 00 00 E8 CD	; .嫡]胱文俟...奈
00000230h:	FF FF FF 89 45 FC 8B 45 FC 50 E8 0D 00 00 00 B3	; 壳疑E疑?..
00000240h:	C4 04 8B E5 5D C3 FF 25 28 30 40 00 FF 25 2C 30	; ?殃]2% (08. %,0
00000250h:	40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	; 8.....
00000260h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
00000270h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
00000280h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
00000290h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
000002a0h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
000002b0h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
000002c0h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
000002d0h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
000002e0h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
000002f0h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
00000300h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
00000310h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
00000320h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
00000330h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
00000340h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
00000350h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
00000360h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
00000370h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
00000380h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
00000390h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
000003a0h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
000003b0h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
000003c0h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
000003d0h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
000003e0h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;
000003f0h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	;

图 10.13 HelloWorld.exe (200h-3ffh)

表 10.8 HelloWorld.exe 代码节内容解读

代码节内容					
指令地址 (HEX)	Intel x86 机器指令	汇编解释	指令地址 (HEX)	Intel x86 机器指令	汇编解释
main 函数		_entry 函数			
1000	55	PUSH EBP	1025	55	PUSH EBP
1001	89 E5	MOV EBP,ESP	1026	89E5	MOV EBP,ESP
1003	81 EC 00000000	SUB ESP,0	1028	81EC 04000000	SUB ESP,4
1009	B8 00204000	MOV EAX, 00204000 (将 "Hello World!\n" 地址赋给 EAX)	102E	E8 CDFFFF	CALL 00401000 (调用 main)
100E	50	PUSH EAX	1033	8945 FC	MOV DWORD PTR SS:[EBP-4],EAX
100F	E8 32000000	CALL < JMP. &msvert.printf > 调用 (printf)	1036	8B45 FC	MOV EAX, DWORD PTR SS:[EBP-4]
1014	83C4 04	ADD ESP,4	1039	50	PUSH EAX
1017	B8 00000000	MOV EAX,0	103A	E8 0D000000	CALL < JMP. &msvert.exit > (调用 exit)
101C	E9 00000000	JMP hello.00000021	103F	83C4 04	ADD ESP,4
1021	8BE5	MOV ESP,EBP	1042	8BE5	MOV ESP,EBP
1023	5D	POP EBP	1044	5D	POP EBP
1024	C3	RETN	1045	C3	RETN
			1046	FF25 28304000	JMP DWORD PTR DS:[<&.msvert.printf>]
			104C	FF25 2C304000	JMP DWORD PTR DS:[<&.msvert.exit>]

请大家将这张表与 8.6 节 HelloWorld.obj 代码节内容解读认真对比一下,会发现本表加粗显示的部分就是不同点。这几个不同点用一句话来概念就是“解析了未定义的符号引用,目标文件中的占位符替换为符号的地址”,这些就是链接器功劳。

10.2.6 数据节

数据节包括 .data 节、.rdata 节和 .bss 节,.data 存储可读可写需要初始化的数据,.rdata 存储只读数据,.bss 存储可读可写不需要进行初始化的数据。HelloWorld.exe 只有 .rdata 数据节,我们来看一下该数据节的内容(见图 10.14)。

```

00000400h: 48 65 6C 6C 6F 20 57 6F 72 6C 64 21 0A 00 00 00 ; Hello World!
00000410h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000420h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000430h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000440h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000450h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000460h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000470h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000480h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000490h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000004a0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000004b0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000004c0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000004d0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000004e0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000004f0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000500h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000510h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000520h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000530h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000540h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000550h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000560h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000570h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000580h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000590h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000005a0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000005b0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000005c0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000005d0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000005e0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000005f0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;

```

图 10.14 HelloWorld.exe (400h-5ffh)

可以看到,该节存储的具体内容是“Hello World!\n”常量字符串。为了保证节内容在文件中以 0x200 粒度对齐,该节后面也有大量用 0 填充的字节。

10.2.7 导入节

所有导入符号的映像文件,实际上几乎包括所有的可执行(EXE)文件,都有 .idata 节。文件中导入信息的典型布局如图 10.15 所示。

HelloWorld.exe 的导入表如图 10.16 所示。

图 10.16 看上去比较乱,可通过图 10.17 来解读导入节的内容及各部分内容之间的关联,请大家结合图 10.17 来看 10.2.7.1~10.2.7.4 几节的内容。

为了大家对导入节有更直观的认识,可用 VC6 自带的 Dependency.exe 程序来看一下 HelloWorld.exe 对其他模块的依赖关系(见图 10.18)。

可以看到,HelloWorld.exe 直接引用了 msvert.dll,引用了其中的 printf 及 _exit 函数。是不是感觉非常清爽。代码引用了哪个模块,引用了其中的哪个函数一清二楚。不像 VC6

目录表
空目录项
DLL1的导入查找表
空表项
DLL2的导入查找表
空表项
...
DLL1的导入地址表
空表项
DLL2的导入地址表
空表项
...
DLL1名称
DLL2名称
...
提示/名称表

图 10.15 典型的导入节布局

```

00000600h: 34 30 00 00 00 00 00 00 00 00 00 00 40 30 00 00 ; 40.....00..
00000610h: 28 30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; (0.....
00000620h: 00 00 00 00 00 00 00 00 00 00 48 30 00 00 54 30 00 00 ; .....KO..TO..
00000630h: 00 00 00 00 48 30 00 00 54 30 00 00 00 00 00 00 ; .....KO..TO..
00000640h: 60 73 76 63 72 74 2E 64 6C 6C 66 00 00 70 72 69 ; msvcrt.dll..pri
00000650h: 6E 74 66 00 00 00 65 78 69 74 00 00 00 00 00 00 ; ntf..exit...
00000660h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000670h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000680h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000690h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000006a0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000006b0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000006c0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000006d0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000006e0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000006f0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000700h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000710h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000720h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000730h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000740h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000750h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000760h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000770h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000780h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00000790h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000007a0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000007b0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000007c0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000007d0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000007e0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
000007f0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;

```

图 10.16 HelloWorld.exe (600h-7ffh)

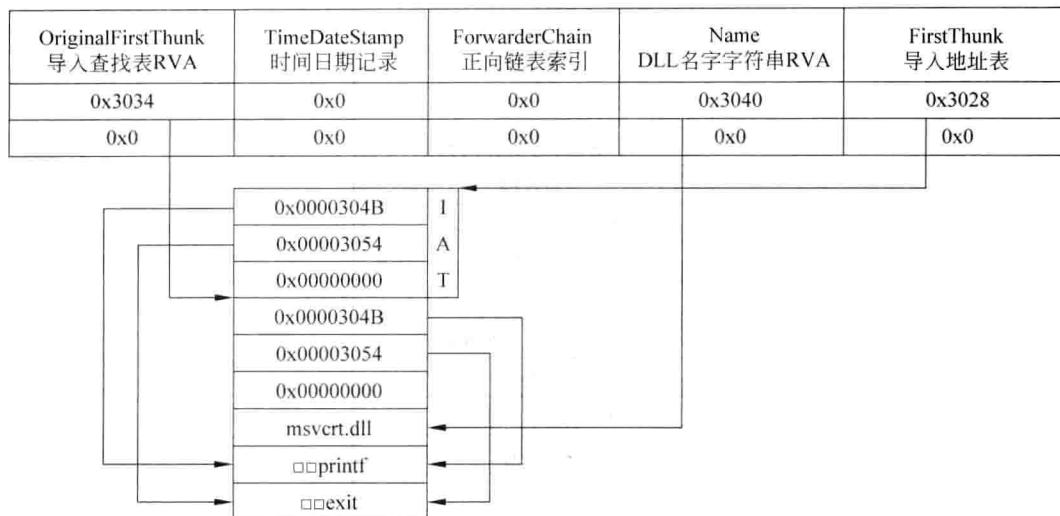


图 10.17 HelloWorld.exe 导入节内容解读

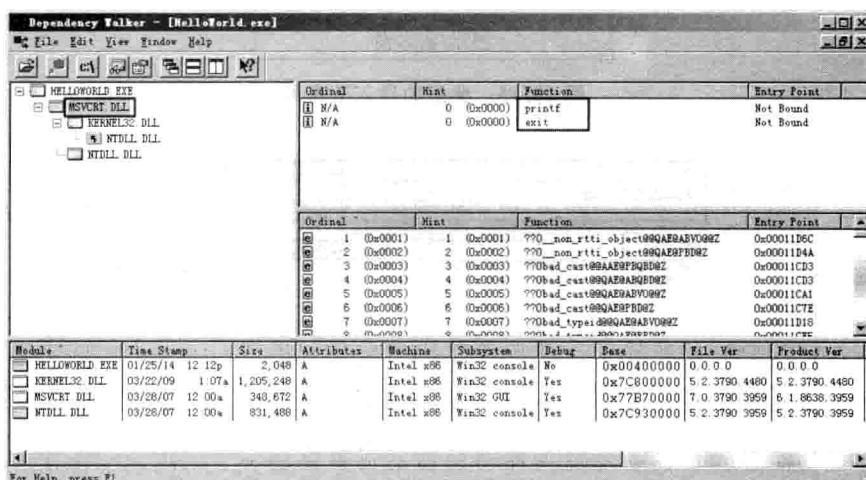


图 10.18 HelloWorld.exe 对其他模块的依赖关系

自动生成的 HelloWorld.exe 有一些潜规则, 引用了哪些模块, 引用了哪些函数一点都不透明。

10.2.7.1 导入目录表

导入目录表是导入信息的开始部分, 它描述了导入信息中其余部分的内容。导入目录表包含地址信息, 这些地址信息用来修正对 DLL 映像中的相应函数的引用。导入目录表是由导入目录项组成的数组, 每个导入目录项对应着一个导入的 DLL。最后一个导入目录项是空的(全部域的值都为 0), 用来指明目录表的结尾。每个导入目录项的格式如下, 其各个成员变量说明见表 10.9。

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
```

```

union {
    DWORD Characteristics;
    DWORD OriginalFirstThunk;
};

DWORD TimeDateStamp;
DWORD ForwarderChain;
DWORD Name;
DWORD FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR;

```

表 10.9 导入目录表

偏移	大小	域	说 明
0	4	OriginalFirstThunk	导入查找表的 RVA,这个表包含了每一个导入符号的名称或序数
4	4	TimeDateStamp	这个域一直被设置为 0,直到映像被绑定。当映像被绑定之后,这个域被设置为这个 DLL 的日期/时间戳
8	4	ForwarderChain	第一个转发项的索引
12	4	Name	包含 DLL 名称的 ASCII 码字符串相对于映像基址的偏移地址
16	4	FirstThunk	导入地址表的 RVA,这个表的内容与导入查找表的内容完全一样,直到映像被绑定

10.2.7.2 导入查找表

导入查找表是由长度为 32 位的数字组成的数组,其中每一个元素都是位域,其格式如表 10.10 所示。在这种格式中,位 31 是最高位。这些项描述了从给定的 DLL 导入的所有符号。最后一个项被设置为 0,用来指明表的结尾。

表 10.10 导入查找表

位	大小	位 域	说 明
31	1	Ordinal/Name Flag	如果这个位为 1,说明是通过序数导入的;否则,是通过名称导入的。测试这个位的掩码为 0x80000000
15-0	16	Ordinal Number	序数值(16 位长),只有当 Ordinal/Name Flag 域为 1(即通过序数导入)时才使用这个域,位 30~15 必须为 0
30-0	31	Hint/Name Table RVA	提示/名称表项的 RVA(31 位长),只有当 Ordinal/Name Flag 域为 0(即通过名称导入)时才使用这个域

10.2.7.3 提示/名称表

一个提示/名称表就能满足整个导入节的需要。提示/名称表中的每一个元素结构如下,其各个成员变量说明见表 10.11。

```

typedef struct _IMAGE_IMPORT_BY_NAME {
    WORD Hint;
}

```

```

    BYTE      Name[1];
} IMAGE_IMPORT_BY_NAME, * PIMAGE_IMPORT_BY_NAME;

```

表 10.11 提示/名称表

偏移	大 小	域	说 明
0	2	Hint	导出名称指针表的索引,当搜索匹配字符串时首选使用这个值。如果匹配失败,再在 DLL 的导出名称指针表中进行二进制搜索
2	可变	Name	包含导入符号名称的 ASCII 码字符串,这个字符串必须与 DLL 导出的公用名称匹配。同时这个字符串区分大小写并且以 NULL 结尾
*	0 或 1	Pad	为了让提示/名称表的下一个元素出现在偶数地址,这里可能需要填充 0 个或 1 个 NULL 字节

10.2.7.4 导入地址表

导入地址表的结构和内容与导入查找表完全一样,直到文件被绑定。在绑定过程中,用导入符号的 32 位地址覆盖导入地址表中的相应项。这些地址是导入符号的实际内存地址,尽管技术上仍把它们称为“虚拟地址”,加载器通常会处理绑定。

10.3 链接器代码实现

学习完 PE 可执行文件结构,下面来写生成 EXE 的链接器代码。代码较难理解的地方作者都做了注释,后面的代码主要用到了本章前两节及 7.1 节介绍的内容,如果没有特殊需要说明的地方作者将不做过多解释。

10.3.1 生成 PE 文件头

10.3.1.1 MS-DOS 头

```

/*
 * 声明 PE 头变量,并进行初始化
 */
IMAGE_DOS_HEADER dos_header = {
    /* IMAGE_DOS_HEADER doshdr */ 
    0x5A4D, /* WORD e_magic;           DOS 可执行文件标记,为'MZ' */
    0x0090, /* WORD e_cblp;            Bytes on last page of file */
    0x0003, /* WORD e_cp;              Pages in file */
    0x0000, /* WORD e_crlc;            Relocations */

    0x0004, /* WORD e_cparhdr;          Size of header in paragraphs */
    0x0000, /* WORD e_minalloc;         Minimum extra paragraphs needed */
    0xFFFF, /* WORD e_maxalloc;         Maximum extra paragraphs needed */
    0x0000, /* WORD e_ss;               DOS 代码的初始化堆栈段 */

    0x00B8, /* WORD e_sp;               DOS 代码的初始化堆栈指针 */
}

```

```

0x0000, /* WORD e_csum;           Checksum */
0x0000, /* WORD e_ip;             DOS 代码的入口 IP */
0x0000, /* WORD e_cs;             DOS 代码的入口 CS */
0x0040, /* WORD e_lfarlc;        File address of relocation table */
0x0000, /* WORD e_ovno;           Overlay number */
{0,0,0,0}, /* WORD e_res[4];        Reserved words */
0x0000, /* WORD e_oemid;          OEM identifier(for e_oeminfo) */
0x0000, /* WORD e_oeminfo;        OEM information; e_oemid specific */
{0,0,0,0,0,0,0,0,0,0}, /* WORD e_res2[10];       Reserved words */
0x00000080 /* DWORD   e_lfanew;      指向 PE 文件头 */
};

}

```

10.3.1.2 MS-DOS 占位程序

```

BYTE dos_stub[0x40]={
/* BYTE dosstub[0x40] */

/* 14 code bytes+ "This program cannot be run in DOS mode.\n$"+7 * 0x00 */

0x0e,0x1f,0xba,0x0e,0x00,0xb4,0x09,0xcd,0x21,0xb8,0x01,0x4c,0xcd,0x21,
0x54,0x68,0x69,0x73,0x20,0x70,0x72,0x6f,0x67,0x72,0x61,0x6d,0x20,0x63,
0x61,0x6e,0x6e,0x6f,0x74,0x20,0x62,0x65,0x75,0x6e,0x20,0x69,0x6e,0x20,0x44,
0x65,0x64,0x2e,0x0d,0x0a,0x24,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
};

}

```

图 10.19 为 MS-DOS 占位程序解读。

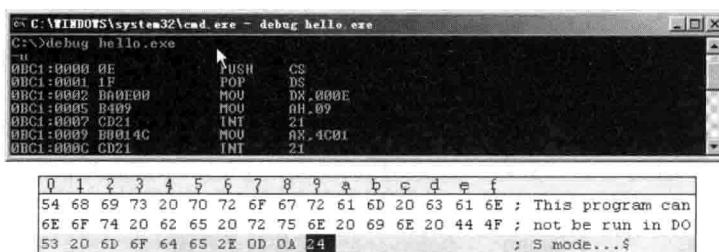


图 10.19 MS-DOS 占位程序解读

10.3.1.3 NT 头

```

IMAGE_NT_HEADERS32 nt_header={
0x00004550,      /* DWORD Signature=IMAGE_NT_SIGNATURE PE 文件标识 */
{
/* IMAGE_FILE_HEADER FileHeader */
0x014C,          /* WORD Machine; 运行平台 */
0x0003,          /* WORD NumberOfSections; 文件的节数目 */
0x00000000,      /* DWORD TimeDateStamp; 文件的创建日期和时间 */
0x00000000,      /* DWORD PointerToSymbolTable; 指向符号表(用于调试) */
}

```

```

    0x00000000, /* DWORD  NumberOfSymbols; 符号表中的符号数量(用于调试) */
    0x00E0,      /* WORD   SizeOfOptionalHeader; IMAGE_OPTIONAL_HEADER32 结构长度 */
    0x030F      /* WORD   Characteristics; 文件属性 */
},
{
    /* IMAGE_OPTIONAL_HEADER OptionalHeader */
    /* 标准域 */
    0x010B,      /* WORD   Magic; */
    0x06,        /* BYTE  MajorLinkerVersion; 链接器主版本号 */
    0x00,        /* BYTE  MinorLinkerVersion; 链接器次版本号 */
    0x00000000, /* DWORD  SizeOfCode; 所有含代码段的总大小 */
    0x00000000, /* DWORD  SizeOfInitializedData; 已初始化数据段的总大小 */
    0x00000000, /* DWORD  SizeOfUninitializedData; 未初始化数据段的大小 */
    0x00000000, /* DWORD  AddressOfEntryPoint; 程序执行入口的相对虚拟地址 */
    0x00000000, /* DWORD  BaseOfCode; 代码段的起始 RVA */
    0x00000000, /* DWORD  BaseOfData; 代码段的起始 RVA */

    /* NT 附加域 */
    0x00400000, /* DWORD  ImageBase; 程序的建议装载地址 */
    0x00001000, /* DWORD  SectionAlignment; 内存中段的对齐粒度 */
    0x00000200, /* DWORD  FileAlignment; 文件中段的对齐粒度 */
    0x0004,      /* WORD   MajorOperatingSystemVersion; 操作系统的主版本号 */
    0x0000,      /* WORD   MinorOperatingSystemVersion; 操作系统的次版本号 */
    0x0000,      /* WORD   MajorImageVersion; 程序的主版本号 */
    0x0000,      /* WORD   MinorImageVersion; 程序的次版本号 */
    0x0004,      /* WORD   MajorSubsystemVersion; 子系统的主版本号 */
    0x0000,      /* WORD   MinorSubsystemVersion; 子系统的次版本号 */
    0x00000000, /* DWORD  Win32VersionValue; 保留,设为 0 */
    0x00000000, /* DWORD  SizeOfImage; 内存中整个 PE 映像尺寸 */
    0x00000200, /* DWORD  SizeOfHeaders; 所有头+节表的大小 */
    0x00000000, /* DWORD  CheckSum; 校验和 */
    0x0003,      /* WORD   Subsystem; 文件的子系统 */
    0x0000,      /* WORD   DllCharacteristics; */
    0x00100000, /* DWORD  SizeOfStackReserve; 初始化时堆栈大小 */
    0x00001000, /* DWORD  SizeOfStackCommit; 初始化时实际提交的堆栈大小 */
    0x00100000, /* DWORD  SizeOfHeapReserve; 初始化时保留的堆大小 */
    0x00001000, /* DWORD  SizeOfHeapCommit; 初始化时实际提交的堆大小 */
    0x00000000, /* DWORD  LoaderFlags; 保留,设为 0 */
    0x00000010, /* DWORD  NumberOfRvaAndSizes; 下面的数据目录结构的数量 */

    /* IMAGE_DATA_DIRECTORY DataDirectory[16]; 数据目录 */
    {{0,0}, {0,0}, {0,0}, {0,0}, {0,0}, {0,0}, {0,0}, {0,0},
     {0,0}, {0,0}, {0,0}, {0,0}, {0,0}, {0,0}, {0,0}, {0,0}}
};

}

```

10.3.2 加载目标文件

```
*****  
* 功能： 加载目标文件  
* fname: 目标文件名  
*****  
  
int load_obj_file(char * fname)  
{  
    IMAGE_FILE_HEADER fh;  
    Section **secs;  
    int i,sh_size,nsec_obj=0,nsym;  
    FILE * fobj;  
    CoffSym * cfsyms;  
    CoffSym * cfsym;  
    char * strs, * csname;  
    void * ptr;  
    int cur_text_offset;  
    int cur_rel_offset;  
    int * old_to_new_syms;  
    int cfsym_index;  
    CoffReloc * rel, * rel_end;  
  
    sh_size=sizeof(IMAGE_SECTION_HEADER);  
    fobj=fopen(fname,"rb");  
    fread(&fh,1,sizeof(IMAGE_FILE_HEADER),fobj);  
    nsec_obj=fh.NumberOfSections;  
    secs=(Section * *)sections.data;  
    for(i=0; i<nsec_obj; i++)  
    {  
        fread(secs[i]->sh.Name,1,sh_size,fobj);  
    }  
  
    cur_text_offset=sec_text->data_offset;  
    cur_rel_offset=sec_rel->data_offset;  
    for(i=0; i<nsec_obj; i++)  
    {  
        if(!strcmp(secs[i]->sh.Name,".symtab"))  
        {  
            cfsyms=mallocz(secs[i]->sh.SizeOfRawData);  
            fseek(fobj,SEEK_SET,secs[i]->sh.PointerToRawData);  
            fread(cfsyms,1,secs[i]->sh.SizeOfRawData,fobj);  
            nsym=secs[i]->sh.SizeOfRawData / sizeof(CoffSym);  
            continue;  
        }  
        if(!strcmp(secs[i]->sh.Name,".strtab"))
```

```

    {
        strs=mallocz(secs[i]->sh.SizeOfRawData);
        fseek(fobj,SEEK_SET,secs[i]->sh.PointerToRawData);
        fread(strs,1,secs[i]->sh.SizeOfRawData,fobj);
        continue;
    }
    if(!strcmp(secs[i]->sh.Name,".dynsym") || !strcmp(secs[i]->sh.Name,".dynstr"))
        continue;
    fseek(fobj,SEEK_SET,secs[i]->sh.PointerToRawData);
    ptr=section_ptr_add(secs[i], secs[i]->sh.SizeOfRawData);
    fread(ptr,1,secs[i]->sh.SizeOfRawData,fobj);
}
old_to_new_syms=mallocz(sizeof(int) * nsym);
for(i=1; i<nsym; i++)
{
    cfsym=&cfsyms[i];
    csname=strs+cfsym->Name;
    cfsym_index=coffsym_add(sec_symtab,csname,cfsym->Value,
                           cfsym->Section,cfsym->Type,cfsym->StorageClass);
    old_to_new_syms[i]=cfsym_index;
}

rel=(CoffReloc *) (sec_rel->data+cur_rel_offset);
rel_end=(CoffReloc *) (sec_rel->data+sec_rel->data_offset);
for(; rel<rel_end; rel++)
{
    cfsym_index=rel->cfsym;
    rel->cfsym=old_to_new_syms[cfsym_index];
    rel->offset+=cur_text_offset;
}

free(cfsyms);
free(strs);
free(old_to_new_syms);
fclose(fobj);
return 1;
}

```

10.3.3 加载引入库文件

```

/*****************
* 功能：加载静态库
* name: 静态库文件名,不包括路径和后缀
*****************/
int pe_load_lib_file(char * name)

```

```

{
    char libfile[MAX_PATH];
    int ret=-1;
    char line[400], * dllname, * p;
    FILE * fp;
    sprintf(libfile,"%s%s.slib",lib_path,name);
    fp=fopen(libfile, "rb");
    if(fp)
    {
        dllname=get_dllname(libfile);
        dynarray_add(&array_dll, dllname);
        for(;;)
        {
            p=get_line(line, sizeof line, fp);
            if(NULL==p)
                break;
            if(0==*p || ';'==*p)
                continue;
            coffsym_add(sec_dynsymtab, p, array_dll.count, sec_text->index,
                        CST_FUNC, IMAGE_SYM_CLASS_EXTERNAL);
        }
        ret=0;
        if(fp)
            fclose(fp);
    }
    else
    {
        link_error("\\"%s\\" 文件打开失败",libfile);
    }
    return ret;
}

/******************
* 功能：由 libfile 得到相应 dll 名
* libfile: 静态库文件名
*****************/
char * get_dllname(char * libfile)
{
    int n1,n2;
    char * libname, * dllname, * p;
    n1=strlen(libfile);
    libname=libfile;
    for(p=libfile+n1; n1>0; p--)
    {
        if(*p=='\\')

```

```

    {
        libname=p+1;
        break;
    }
}

n2=strlen(libname);
dllname=malloc(sizeof(char) * n2);
memcpy(dllname,libname,n2-4);
memcpy(dllname+n2-4,"dll",3);
return dllname;
}

/******************
* 功能：从静态库文件中读取一行，并删除无效字符
* line: 数据存储位置
* size: 读取的最大字符数
* fp: 已打开的静态库文件指针
*****************/
char * get_line(char * line, int size, FILE * fp)
{
    char * p, * pl;
    if(NULL==fgets(line, size, fp))
        return NULL;

    //删除左空格
    p=line;
    while(*p && isspace(*p))
        p++;

    //删除右空格及回车符
    pl=strchr(p,'0');
    while(pl>p && pl[-1]<=' ')      //不能用 isspace 因为还有 0D 0A 回车符
        pl--;
    *pl='0';

    return p;
}

/******************
* 功能：加载需要链接的静态库
*****************/
void add_runtime_libs()
{
    int i;
}

```

```

int pe_type=0;

for(i=0; i<array_lib.count; i++)
{
    pe_load_lib_file(array_lib.data[i]);
}
}

```

10.3.4 解析外部符号

```

/****************************************************************************
 * 功能：解析程序中用到的外部符号
 * pe: PE 信息存储结构指针
 ****
int resolve_coffsyms(struct PEInfo *pe)
{
    CoffSym * sym;
    int sym_index, sym_end;
    int ret=0;
    int found=1;

    sym_end=sec_symtab->data_offset / sizeof(CoffSym);
    for(sym_index=1; sym_index<sym_end; sym_index++)
    {
        sym=(CoffSym*)sec_symtab->data+sym_index;
        if(sym->Section==IMAGE_SYM_UNDEFINED)
        {
            char * name=sec_symtab->link->data+sym->Name;

            unsigned type=sym->Type;
            int imp_sym=pe_find_import(name);
            struct ImportSym * is;
            if(0==imp_sym)
                found=0;
            is=pe_add_import(pe, imp_sym, name);
            if(!is)
                found=0;

            if(found && type==CST_FUNC)
            {
                int offset=is->thk_offset;
                char buffer[100];

```

```

        offset=sec_text->data_offset;
        /* FF /4 JMP r/m32 Jump near,absolute indirect,address given in r/m32 */
        * (WORD *)section_ptr_add(sec_text, 6)=0x25FF;
        sprintf(buffer, "IAT.%s", name);
        is->iat_index=coffsym_add(sec_symtab,buffer,0,sec_idata->index,
        CST_FUNC,IMAGE_SYM_CLASS_EXTERNAL);
        coffreloc_direct_add(offset+2, is->iat_index, sec_text->index,
        IMAGE_REL_I386_DIR32);
        is->thk_offset=offset;

        sym= (CoffSym* )sec_symtab->data+sym_index;
        sym->Value=offset;
        sym->Section=sec_text->index;
    }
    else
    {
        link_error("'"%s'未定义", name);
        ret=1;
    }
}
return ret;
}
/* PE 信息存储结构 */
struct PEInfo
{
    Section * thunk;
    const char * filename;
    DWORD entry_addr;
    DWORD imp_offs;
    DWORD imp_size;
    DWORD iat_offs;
    DWORD iat_size;
    Section **secs;
    int sec_size;
    DynArray imps;
};

/* 导入符号内存存储结构 */
struct ImportSym
{
    int iat_index;
    int thk_offset;
};

```

```
IMAGE_IMPORT_BY_NAME imp_sym;
};
```

请大家结合图 10.20 来理解程序中用到的外部符号是如何解析的。

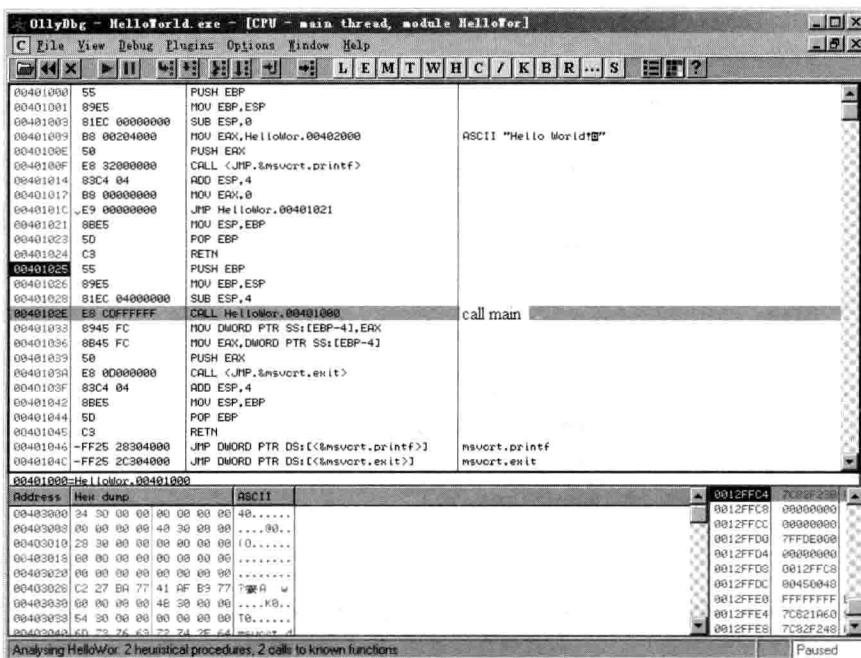


图 10.20 外部函数调用方式

```
/****************************************************************************
 * 功能：查找导入函数
 * symbol：符号名
 */
int pe_find_import(char *symbol)
{
    int sym_index;
    sym_index=coffsym_search(sec_dynsymtab, symbol);
    return sym_index;
}

/****************************************************************************
 * 功能：增加导入函数
 * pe：      PE 信息存储结构指针
 * sym_index：符号表的索引
 * name：    符号名
 */
struct ImportSym * pe_add_import(struct PEInfo * pe, int sym_index, char
* name)
{
    int i;
```

```

int dll_index;
struct ImportInfo * p;
struct ImportSym * s;

dll_index= ((CoffSym * )sec_dynsymtab->data+sym_index)->Value;
if(0==dll_index)
    return NULL;

i=dynarray_search(&pe->imps, dll_index);
if(-1 !=i){
    p=pe->imps.data[i];
}
else
{
    p=mallocz(sizeof * p);
    dynarray_init(&p->imp_syms,8);
    p->dll_index=dll_index;
    dynarray_add(&pe->imps, p);
}

i=dynarray_search(&p->imp_syms,sym_index);
if(-1 !=i)          //找到直接返回,找不到则填加
{
    return (bstruct ImportSym* )p->imp_syms.data[i];
}
else
{
    s=mallocz(sizeof(bstruct ImportSym)+strlen(name));
    dynarray_add(&p->imp_syms, s);
    strcpy((char *) &s->imp_sym.Name, name);
    return s;
}
/* 导入模块内存存储结构 */
struct ImportInfo
{
    int dll_index;
    DynArray imp_syms;
    IMAGE_IMPORT_DESCRIPTOR imphdr;
};

```

10.3.5 计算节区的 RVA 地址

```

*****
* 功能：计算节区的 RVA 地址
*****
```

```
* pe: PE 信息存储结构指针
*****
int pe_assign_addresses(struct PEInfo * pe)
{
    int i;
    DWORD addr;
    Section *sec,**ps;

    pe->thunk=sec_idata;

    pe->secs=(Section** )mallocz(nsec_image * sizeof(Section *));
    addr=nt_header.OptionalHeader.ImageBase+1;

    for(i=0; i<nsec_image; ++i)
    {
        sec=(Section*)sections.data[i];
        ps=&pe->secs[pe->sec_size];
        *ps=sec;

        sec->sh.VirtualAddress=addr=pe_virtual_align(addr);

        if(sec==pe->thunk)
        {
            pe_build_imports(pe);
        }

        if(sec->data_offset)
        {
            addr+=sec->data_offset;
            ++pe->sec_size;
        }
    }
    return 0;
}

*****  

* 功能：计算内存对齐位置
* n: 未对齐前位置
*****
DWORD pe_virtual_align(DWORD n)
{
    DWORD SectionAlignment=nt_header.OptionalHeader.SectionAlignment;
                                            //内存中段的对齐粒度
    return calc_align(n,SectionAlignment);
}
```

```

/***********************
* 功能：创建导入信息(导入目录表及导入符号表)
* pe: PE 信息存储结构指针
***********************/
void pe_build_imports(struct PEInfo * pe)
{
    int thk_ptr, ent_ptr, dll_ptr, sym_cnt, i;
    DWORD rva_base = pe->thunk->sh.VirtualAddress - nt_header.OptionalHeader
        .ImageBase;
    int ndlls=pe->imps.count;

    for(sym_cnt=i=0; i<ndlls; ++i)
        sym_cnt+=((struct ImportInfo * )pe->imps.data[i])->imp_syms.count;

    if(0==sym_cnt)
        return;

    pe->imp_offs=dll_ptr=pe->thunk->data_offset;
    pe->imp_size=(ndlls+1) * sizeof(IMAGE_IMPORT_DESCRIPTOR);
    pe->iat_offs=dll_ptr+pe->imp_size;
    pe->iat_size=(sym_cnt+ndlls) * sizeof(DWORD);
    section_ptr_add(pe->thunk, pe->imp_size+2 * pe->iat_size);
                                            //预留 DLL 导出表及 IAT 表空间

    thk_ptr=pe->iat_offs;
    ent_ptr=pe->iat_offs+pe->iat_size;

    for(i=0; i<pe->imps.count; ++i)
    {
        int k, n, v;
        struct ImportInfo * p=pe->imps.data[i];
        char * name=array_dll.data[p->dll_index-1];

        /* 写入 dll 名称 */
        v=put_import_str(pe->thunk, name);

        p->imphdr.FirstThunk=thk_ptr+rva_base;
        p->imphdr.OriginalFirstThunk=ent_ptr+rva_base;
        p->imphdr.Name=v+rva_base;
        memcpy(pe->thunk->data+dll_ptr,&p->imphdr,sizeof(IMAGE_IMPORT_DESCRIPTOR));
                                            //增加导入表条目

        for(k=0, n=p->imp_syms.count; k<=n; ++k)
        {
            if(k<n)

```

```

{
    struct ImportSym * is=(struct ImportSym*)p->imp_syms.data[k];
    DWORD iat_index=is->iat_index;
    CoffSym * org_sym=(CoffSym *)sec_symtab->data+iat_index;

    org_sym->Value=thk_ptr;
    org_sym->Section=pe->thunk->index;
    v=pe->thunk->data_offset+rva_base;

    section_ptr_add(pe->thunk, sizeof(is->imp_sym.Hint));
    put_import_str(pe->thunk, is->imp_sym.Name);
}

else
{
    v=0; /* last entry is zero */
}

* (DWORD *) (pe->thunk->data+thk_ptr)=
* (DWORD *) (pe->thunk->data+ent_ptr)=v;
thk_ptr+=sizeof(DWORD);
ent_ptr+=sizeof(DWORD);
}

dll_ptr+=sizeof(IMAGE_IMPORT_DESCRIPTOR);
dynarray_free(&p->imp_syms);
}

dynarray_free(&pe->imps);
}

/******************
 * 功能：向导入符号表中写入符号字符串(dll 名称或导入符号)
 * sec: 导入符号所在节
 * sym: 符号名称
 *****************/
int put_import_str(Section * sec, char * sym)
{
    int offset, len;
    char * ptr;
    len=strlen(sym)+1;
    offset=sec->data_offset;
    ptr=section_ptr_add(sec, len);
    memcpy(ptr, sym, len);
    return offset;
}

```

10.3.6 重定位符号地址

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

```

* 功能：重定位符号地址
*****
void relocate_syms()
{
    CoffSym * sym, * sym_end;
    Section * sec;

    sym_end= (CoffSym *) (sec_symtab->data+ sec_symtab->data_offset);
    for(sym= (CoffSym *) sec_symtab->data+1; sym<sym_end; sym++)
    {
        sec= (Section *) sections.data[sym->Section-1];
        sym->Value+= sec->sh.VirtualAddress;           //加上节区的 RVA 地址
    }
}

```

10.3.7 修正需要重定位的地址

```

***** 
* 功能：修正需要进行重定位的代码或数据的地址
*****
void coffrelocs_fixup()
{
    Section * sec, * sr;
    CoffReloc * rel, * rel_end, * qrel;
    CoffSym * sym;
    int type, sym_index;
    char * ptr;
    unsigned long val, addr;
    char * name;

    sr=sec_rel;
    rel_end= (CoffReloc *) (sr->data+sr->data_offset);
    qrel= (CoffReloc *) sr->data;
    for(rel=qrel; rel<rel_end; rel++)
    {
        sec= (Section *) sections.data[rel->section-1];

        sym_index= rel->cfsym;
        sym= &((CoffSym *) sec_symtab->data)[sym_index];
        name= sec_symtab->link->data+ sym->Name;
        val= sym->Value;
        type= rel->type;
        addr= sec->sh.VirtualAddress+ rel->offset;
    }
}

```

```
ptr=sec->data+rel->offset;
switch(type)
{
    case IMAGE_REL_I386_DIR32:           //全局变量
        * (int *)ptr+=val;
        break;
    case IMAGE_REL_I386_REL32:            //函数调用
        * (int *)ptr+=val-addr;
        break;
}
}
```

10.3.8 写 PE 文件

```
*****  
* 功能: 写 PE 文件  
* pe: PE 信息存储结构指针  
*****  
  
int pe_write(struct PEInfo * pe)  
{  
    int i;  
    FILE * op;  
    DWORD file_offset, r;  
    int sizeofheaders;  
  
    op=fopen(pe->filename, "wb");  
    if(NULL==op)  
    {  
        link_error("'%s'生成失败", pe->filename);  
        return 1;  
    }  
  
    sizeofheaders=pe_file_align(  
        sizeof(dos_header)  
        +sizeof(dos_stub)  
        +sizeof(nt_header)  
        +pe->sec_size * sizeof(IMAGE_SECTION_HEADER)  
    );  
  
    file_offset[sizeofheaders];  
    fpad(op, file_offset);  
  
    for(i=0; i<pe->sec_size; ++i)  
    {  
        if(file_offset[i]<0x400)  
            file_offset[i]=0x400;  
        else if(file_offset[i]>0x7FFF)  
            file_offset[i]=0x7FFF;  
        else  
            file_offset[i]=file_offset[i];  
    }  
    fwrite(file_offset, 1, sizeofheaders, op);  
    fclose(op);  
}
```

```

{
    Section * sec=pe->secs[i];
    char * sh_name=sec->sh.Name;
    unsigned long addr = sec -> sh.VirtualAddress - nt_header.OptionalHeader
        .ImageBase;
    unsigned long size=sec->data_offset;
    IMAGE_SECTION_HEADER * psh=&sec->sh;

    if(!strcmp(sec->sh.Name,".text"))
    {
        nt_header.OptionalHeader.BaseOfCode=addr;
        nt_header.OptionalHeader.AddressOfEntryPoint=addr+pe->entry_addr;
    }
    else if(!strcmp(sec->sh.Name,".data"))
        nt_header.OptionalHeader.BaseOfData=addr;
    else if(!strcmp(sec->sh.Name,".idata"))
    {
        if(pe->imp_size)
        {
            pe_set_datadir(IMAGE_DIRECTORY_ENTRY_IMPORT,
                pe->imp_offs+addr, pe->imp_size);
            pe_set_datadir(IMAGE_DIRECTORY_ENTRY_IAT,
                pe->iat_offs+addr, pe->iat_size);
        }
    }

    strcpy((char *)psh->Name, sh_name);

    psh->VirtualAddress=addr;
    psh->Misc.VirtualSize=size;
    nt_header.OptionalHeader.SizeOfImage=pe_virtual_align(
        size+addr
    );

    if(sec->data_offset)
    {
        psh->PointerToRawData=r=file_offset;
        if(!strcmp(sec->sh.Name,".bss"))
        {
            sec->sh.SizeOfRawData=0;
            continue;
        }
        fwrite(sec->data, 1, sec->data_offset, op);
        file_offset=pe_file_align(file_offset+sec->data_offset);
    }
}

```

```

        psh->SizeOfRawData=file_offset-r;
        fpad(op, file_offset);
    }
}

nt_header.FileHeader.NumberOfSections=pe->sec_size;
nt_header.OptionalHeader.SizeOfHeaders=sizeofheaders;

nt_header.OptionalHeader.Subsystem=subsystem;

fseek(op, SEEK_SET, 0);
fwrite(&dos_header, 1, sizeof dos_header, op);
fwrite(&dos_stub, 1, sizeof dos_stub, op);
fwrite(&nt_header, 1, sizeof nt_header, op);
for(i=0; i<pe->sec_size;++i)
    fwrite(&pe->secs[i]->sh, 1, sizeof(IMAGE_SECTION_HEADER), op);
fclose(op);
return 0;
}

/****************************************
* 功能：计算文件对齐位置
* n: 未对齐前位置
****************************************/
DWORD pe_file_align(DWORD n)
{
    DWORD FileAlignment=nt_header.OptionalHeader.FileAlignment;
                                //文件中段的对齐粒度
    return calc_align(n,FileAlignment);
}

/****************************************
* 功能：设置数据目录
* dir: 目录类型
* addr: 表的 RVA
* size: 表的大小(以字节计)
****************************************/
void pe_set_datadir(int dir, DWORD addr, DWORD size)
{
    nt_header.OptionalHeader.DataDirectory[dir].VirtualAddress=addr;
    nt_header.OptionalHeader.DataDirectory[dir].Size=size;
}

```

10.3.9 生成 EXE 文件

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

```

* 功能：生成 EXE 文件
* filename: EXE 文件名
*****
int pe_output_file(char * filename)
{
    int ret;
    struct PEInfo pe;

    memset(&pe, 0, sizeof(pe));
    dynarray_init(&pe.imps, 8);
    pe.filename=filename;
    /* 加载需要链接的静态库 */
    add_runtime_libs();
    /* 计算程序入口点 */
    get_entry_addr(&pe);
    /* 解析程序中用到的外部符号 */
    ret=resolve_coffsyms(&pe);
    if(0==ret)
    {
        /* 计算节区的 RVA 地址 */
        pe_assign_addresses(&pe);
        /* 重定位符号地址 */
        relocate_syms();
        /* 修正需要进行重定位的代码或数据的地址 */
        coffrelocs_fixup();
        /* 写 PE 文件 */
        ret=pe_write(&pe);
        free(pe.secs);
    }
    return ret;
}

*****
* 功能：计算程序入口点
* pe: PE 信息存储结构指针
*****
void get_entry_addr(struct PEInfo * pe)
{
    unsigned long addr=0;
    int cs;
    CoffSym * cfsym_entry;
    cs=coffsym_search(sec_symtab,entry_symbol);
    cfsym_entry=(CoffSym *)sec_symtab->data+cs;
    addr=cfsym_entry->Value;
    pe->entry_addr=addr;
}

```

10.4 SCC 编译器、链接器总控程序

```
*****
* 功能: main 主函数
*****
int main(int argc, char ** argv)
{
    int i,opind;
    char * ext;

    init();
    opind=process_command(argc,argv);
    if(opind==0)
    {
        cleanup();
        return 0;
    }

    for(i=0; i<src_files.count; i++)
    {
        filename=src_files.data[i];
        ext=get_file_ext(filename);
        if(!strcmp(ext,"c"))
            compile(filename);
        if(!strcmp(ext,"obj"))
            load_obj_file(filename);
    }
    if(output_type==OUTPUT_OBJ)
        write_obj(outfile);
    else
        pe_output_file(outfile);

    cleanup();
    return 1;
}

*****
* 功能: 初始化
*****
void init()
{
    output_type=OUTPUT_EXE;
    subsystem=IMAGE_SUBSYSTEM_WINDOWS_CUI;
```

```

dynarray_init(&src_files,1);
dynarray_init(&array_lib,4);
dynarray_init(&array_dll,4);
init_lex();

stack_init(&local_sym_stack,8);
stack_init(&global_sym_stack,8);
sym_sec_rdata=sec_sym_put(".rdata",0);           //IMAGE_SYM_CLASS_EXTERNAL

int_type.t=T_INT;
char_pointer_type.t=T_CHAR;
mk_pointer(&char_pointer_type);
default_func_type.t=T_FUNC;
default_func_type.ref=sym_push(SC_ANOM, &int_type, KW_CDECL, 0);

opstack=opstack-1;

init_coff();

lib_path=get_lib_path();
}

/******************
 * 功能：得到放置静态库的目录
 *****************/
char * get_lib_path()
{
    /* 假定编译需要链接的静态库放在与编译器同级目录的 lib 文件夹下
     此处需要讲一下静态库
     */
    char path[MAX_PATH];
    char * p;
    GetModuleFileNameA(NULL, path, sizeof(path));
    p=strrchr(path,'\\');
    *p='\0';
    strcat(path,"\\lib\\");
    return strdup(path);
}

/******************
 * 功能：扫尾清理工作
 *****************/
void cleanup()
{
    int i;
}

```

```
sym_pop(&global_sym_stack, NULL);
stack_destroy(&local_sym_stack);
stack_destroy(&global_sym_stack);
free_sections();

for(i=TK_IDENT; i<tkttable.count; i++)
{
    free(tkttable.data[i]);
}
free(tkttable.data);
dynarray_free(&array_dll);
free(src_files.data);
free(array_lib.data);
}

/***********************
* 功能：处理命令行选项
* argc: 命令行参数个数
* argv: 命令行参数数组
***********************/

int process_command(int argc, char **argv)
{
    int i;
    for(i=1; i<argc; i++)
    {
        if(argv[i][0]=='-')
        {
            char * p=&argv[i][1];
            int c= * p;
            switch(c)
            {
                case 'o':
                    outfile=argv[++i];
                    break;
                case 'c':
                    dynarray_add(&src_files, argv[++i]);
                    output_type=OUTPUT_OBJ;
                    return 1;
                case 'l':
                    dynarray_add(&array_lib, &argv[i][2]);
                    break;
                case 'G':
                    subsystem=IMAGE_SUBSYSTEM_WINDOWS_GUI;
                    break;
                case 'v':
```

```

        printf("SCC Version%.2f", scc_version);
        return 0;
    case 'h':
        printf("usage: scc [-c infile] [-o outfile] [-llib] [infile1 infile2...]
               \n");
        return 0;
    default:
        printf("不支持的命令行选项");
        return 0;
    }
}
else
{
    dynarray_add(&src_files, argv[i]);
}

}
return 1;

}
/* 输出类型枚举定义 */
enum e_OutType
{
    OUTPUT_OBJ,           //目标文件
    OUTPUT_EXE,           //EXE 可执行文件
};

/*****************
 * 功能：得到文件扩展名
 * fname: 文件名称
 *****************/
char * get_file_ext(char * fname)
{
    char * p;
    p=strrchr(fname,'.');
    return p+1;
}

/*****************
 * 功能：编译 SC 源文件
 * fname: SC 源文件名
 *****************/
void compile(char * fname)
{
    fin=fopen(fname,"rb");
}

```

```

if(!fin)
    printf("不能打开 SC 源文件!\n");
getch();
line_num=1;
get_token();
translation_unit();
fclose(fin);
printf("%s 编译成功\n", fname, line_num);
}

```

下面给出本章用到的全局变量：

FILE * fin=NULL;	//源文件指针
char * filename;	//源文件名称
DynArray src_files;	//源文件数组
char * outfile;	//输出文件名
int output_type;	//输出文件类型
float scc_version=1.00;	//SCC 编译器版本号
char * entry_symbol="_entry";	//入口函数
DynArray array_dll;	//动态库数组
DynArray array_lib;	//引入库数组
char * lib_path;	//引入库所在路径
short subsystem;	//子系统

10.5 成果展示

至此，链接器也完成了，SCC 编译器编译生成的目标文件与库文件链接就可以生成真正可以运行的 EXE 程序了！现在就把 HelloWorld. obj 与 C 运行时库链接生成 HelloWorld. exe，看一下 HelloWorld. exe 的运行结果（见图 10.21）。

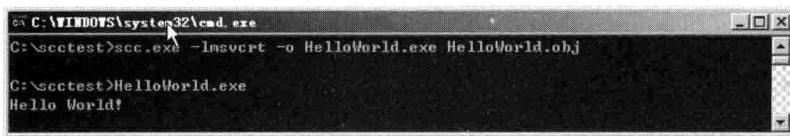


图 10.21 链接生成 HelloWorld. exe 并运行

这可是令人激动的时刻，具有里程碑的意义。SCC 编译器、链接器最终可以生成可执行文件了，执行 HelloWorld. exe 就会在命令行显示“Hello World！”，同样的 HelloWorld. exe 执行，心情却大不一样。因为这可是用自己亲手写的 SCC 编译器、链接器生成的。

HelloWorld. exe 的各个组成部分，在 10.2 节介绍 PE 文件格式时已经讲过，估计你现在对 HelloWorld. exe 的印象还是支离破碎的，本章的最后要展示一下 HelloWorld. exe 文件结构简图（见图 10.22）。

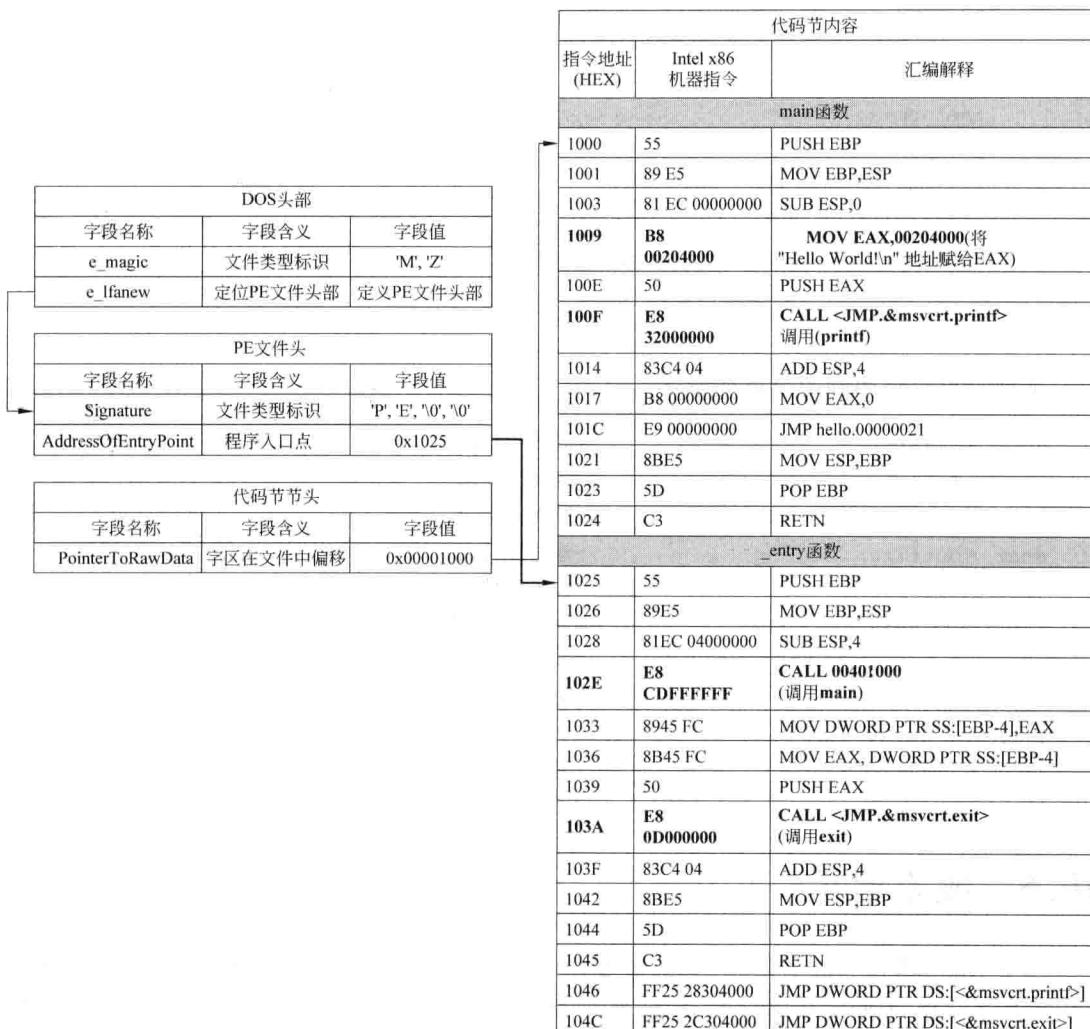


图 10.22 HelloWorld.exe 文件结构简图

10.6 全书代码架构

到这里 SCC 编译器、链接器的编码工作就算结束了，通过图 10.23，来看一下全书的代码架构。

可以看到，代码基本是按章来分的，功能划分还是比较清晰的，dynarray.c、dynstring.c、error.c、lex.c 是在词法分析（第 4 章）介绍的，grammar.c 比较特殊，它是语法分析（第 5 章）、符号表（第 6 章）、语义分析（第 9 章）代码的复合体，stack.c 及 symbol.c 是在符号表（第 6 章）中介绍的，outcoff.c 是在生成 COFF 文件（第 7 章）介绍的，operand.c 及 gencode.c 是在生成 x86 机器语言（第 8 章）介绍的，outpe.c 是在链接生成 PE 文件（第 10 章）介绍的，scc.c 中是各章的控制程序，scc.h 负责存放结构定义、枚举定义、宏定义及各个模块需要相互引用的函数声明和全局变量声明。

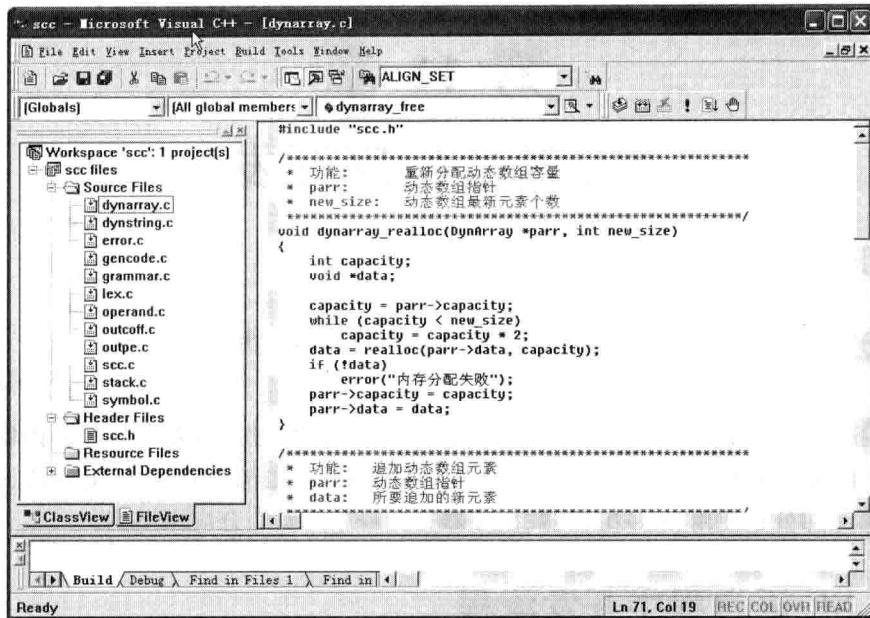


图 10.23 SCC 编译器、链接器代码架构

第 11 章

SC 语言程序开发

宝剑锋从磨砺出，梅花香自苦寒来

——《警世贤文》

SCC 编译器、链接器算是写完了。本章做一些收尾的工作，主要有 3 部分内容，第一部分介绍使用 SCC 编译器、链接器进行程序开发的流程，第二部分对 SCC 编译器、链接器进行测试，第三部分是使用 SCC 编译器、链接器进行程序开发综合应用举例。

11.1 SC 语言程序开发流程

使用 SCC 编译器、链接器进行 SC 语言程序开发的流程如图 11.1 所示，好像跟使用 VC6 开发 C、C++ 程序没什么本质区别。编辑器大家可以使用记事本、UltraEdit 等，当然也可以把 VC6 的 IDE 当成编辑器来使用也没问题，但似乎有点大材小用。

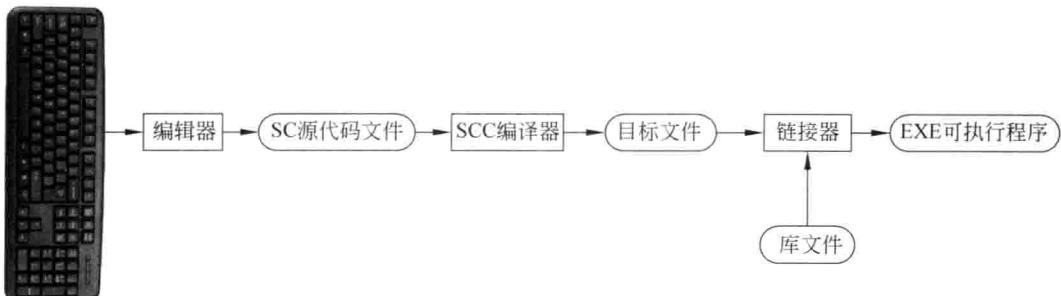


图 11.1 用 SCC 进行程序开发流程

11.2 SCC 编译器测试程序

本节精心设计了一些测试用例，可用来对 SCC 编译器进行测试。

11.2.1 表达式测试

```
/* expr_test.c 源文件 */
/****************************************************************************
 * 功能：算术表达式测试
 ****/
void expr arithmetic test()
```

```
{  
    int a=6, b=2, c, d, e, f, g;  
    c=a+b;  
    d=a-b;  
    e=a * b;  
    f=a / b;  
    g=a%b;  
    printf("%d+ %d=%d\n",a,b,c);  
    printf("%d- %d=%d\n",a,b,d);  
    printf("%d * %d=%d\n",a,b,e);  
    printf("%d / %d=%d\n",a,b,f);  
    printf("%d%%%d=%d\n",a,b,g);  
}  
  
/**************************************************************************/  
* 功能：指针运算测试  
/**************************************************************************/  
void pointer_op_test()  
{  
    char      * p1=0, * p2;  
    short   * p3=0, * p4;  
    int      * p5=0, * p6;  
  
    p2=p1+1;  
    p4=p3+1;  
    p6=p5+1;  
    printf("p1=% .8x, p2=% .8x\n",p1,p2);  
    printf("p3=% .8x, p4=% .8x\n",p3,p4);  
    printf("p5=% .8x, p6=% .8x\n",p5,p6);  
}  
  
/**************************************************************************/  
* 功能：比较运算测试  
/**************************************************************************/  
void expr_cmp_test()  
{  
    int a, b,c;  
  
    a=123;  
    b=456;  
    printf("%d==%d=%d\n", a, a, a==a);  
    printf("%d !=%d=%d\n", a, a, a !=a);  
  
    printf("%d<%d=%d\n", a, b, a<b);  
    printf("%d<=%d=%d\n", a, b, a<=b);  
    printf("%d<=%d=%d\n", a, a, a<=a);  
    printf("%d>=%d=%d\n", b, a, b>=a);
```

```
printf("%d>=%d=%d\n", a, a, a>=a);
printf("%d>%d=%d\n", b, a, b>a);
}

/******************
 * 功能：数组测试
 *****************/
void array_test()
{
    int arr1[2];
    int arr2[2][2];

    arr1[0]=100;
    arr1[1]=200;

    arr2[0][0]=10;
    arr2[0][1]=20;
    arr2[1][0]=30;
    arr2[1][1]=40;

    printf("arr1[0]=%d, arr1[1]=%d\n", arr1[0], arr1[1]);
    printf("arr2[0][0]=%d, arr2[0][1]=%d\n", arr2[0][0], arr2[0][1]);
    printf("arr2[1][0]=%d, arr2[1][1]=%d\n", arr2[1][0], arr2[1][1]);
}

/******************
 * 功能：正负号测试
 *****************/
void plus_minus_test()
{
    int a=+7;
    int b=-9;

    printf("a=%d, b=%d\n", a, b);
}

/******************
 * 功能：类型自动转换测试
 *****************/
int cast_test()
{
    int c;
    short b;
    char a;

    a=68;
    b=a;
```

```

c=b;
printf("a=%d, b=%d, c=%d\n", a, b, c);
}

/******************
* 功能：main 函数
*****************/
int main()
{
    expr_arithmetic_test();
    pointer_op_test();
    expr_cmp_test();
    array_test();
    plus_minus_test();
    cast_test();

    return 0;
}

/******************
* 功能：入口函数
*****************/
void _entry()
{
    int ret;
    ret=main();
    exit(ret);
}

```

expr_test.c 编译及运行结果如图 11.2 所示。

```

C:\scctest>scc.exe -lsvcrt -o expr_test.exe expr_test.c
expr_test.c 编译成功

C:\scctest>expr_test.exe
6 + 2 = 8
6 - 2 = 4
6 * 2 = 12
6 / 2 = 3
6 % 2 = 0
p1=00000000, p2=00000001
p3=00000000, p4=00000002
p5=00000000, p6=00000004
123 == 123 = 1
123 != 123 = 0
123 < 456 = 1
123 <= 456 = 1
123 <= 123 = 1
456 >= 123 = 1
123 >= 123 = 1
456 > 123 = 1
arr1[0] = 100, arr1[1] = 200
arr2[0][0]=10, arr2[0][1]=20
arr2[1][0]=30, arr2[1][1]=40
a = 7, b = -9
a = 68, b = 68, c = 68

```

图 11.2 表达式测试

11.2.2 语句测试

```
/* stmt_test.c 源文件 */
//****************************************************************************
* 功能：复合语句, 变量作用域测试
*****
void compound_test()
{
    int a=10;

    printf("a=%d\n",a);
    {
        int a=20;
        printf("a=%d\n",a);
        {
            int a=30;
            printf("a=%d\n",a);
        }
        printf("a=%d\n",a);
    }
    printf("a=%d\n",a);
}

//****************************************************************************
* 功能：if 语句测试
*****
void if_test()
{
    int a=3, b=5;

    if(a>b)
        printf("a(%d)>b(%d)\n", a, b);
    else
        printf("a(%d)<b(%d)\n", a, b);

    if(50>30)
        printf("50>30\n");
    else
        printf("50<30\n");
}

//****************************************************************************
* 功能：for 语句测试, 包含 continue 语句, break 语句测试
*****
void for_test()
```

```
{  
    int i;  
  
    for(i=0; i<10; i=i+1)  
    {  
        if(i==5)  
            continue;  
        if(i==8)  
            break;  
        printf("i=%d\n",i);  
    }  
}  
  
/**************************************************************************/  
* 功能: return 语句测试  
/**************************************************************************/  
int return_test()  
{  
    return 2014;  
}  
  
/**************************************************************************/  
* 功能: main 函数  
/**************************************************************************/  
int main()  
{  
    compound_test();  
    if_test();  
    for_test();  
    return_test();  
  
    return 0;  
}  
  
/**************************************************************************/  
* 功能: 入口函数  
/**************************************************************************/  
void _entry()  
{  
    int ret;  
    ret=main();  
    exit(ret);  
}
```

stmt_test.c 编译及运行结果如图 11.3 所示。

```

C:\> C:\WINDOWS\system32\cmd.exe
C:\>scctest>cc.exe -lmsvcrt -o stmt_test.exe stmt_test.c
stmt_test.c 编译成功

C:\>scctest>stmt_test.exe
a=10
a=20
a=30
a=20
a=10
a<3 < b<5
50 > 30
i = 0
i = 1
i = 2
i = 3
i = 4
i = 6
i = 7

```

图 11.3 语句测试

11.2.3 结构体测试

```

/* struct_test.c 源文件 */
/****************************************************************************
 * 功能：结构体测试
 ****/
struct point
{
    int x;
    int y;
};

void struct_test()
{
    struct point pt;
    struct point * ppt;

    ppt=&pt;
    pt.x=1024;
    pt.y=768;
    printf("pt.x=%d, pt.y=%d\n",pt.x,pt.y);
    printf("ppt->x=%d, ppt->y=%d\n",ppt->x,ppt->y);
}

/****************************************************************************
 * 功能：求类型长度测试
 ****/
void sizeof_test()
{
    printf("sizeof(char)=%d, sizeof(short)=%d, sizeof(int)=%d,
           sizeof(struct point)=%d\n",
           sizeof(char),sizeof(short),sizeof(int),

```

```
        sizeof(struct point));  
    }  
  
/**********************************************************/  
 * 功能：强制对齐测试  
/**********************************************************/  
void struct_align_test()  
{  
    struct s1  
    {  
        char    m1;  
        short   m2;  
        int     m3;  
    };  
  
    struct s2  
    {  
        char    m1;  
        short __align(1)m2;  
        int     __align(1)m3;  
    };  
  
    struct s3  
    {  
        char m1;  
        short __align(4)m2;  
        int  __align(4)m3;  
    };  
  
    printf("sizeof(struct s1)=%d\n",sizeof(struct s1));  
    printf("sizeof(struct s2)=%d\n",sizeof(struct s2));  
    printf("sizeof(struct s3)=%d\n",sizeof(struct s3));  
}  
  
/**********************************************************/  
 * 功能：main 函数  
/**********************************************************/  
int main()  
{  
    struct_test();  
    sizeof_test();  
    struct_align_test();  
  
    return 0;  
}
```

```
*****  
* 功能：入口函数  
*****  

void _entry()  
{  
    int ret;  
    ret=main();  
    exit(ret);  
}
```

struct_test.c 编译及运行结果如图 11.4 所示。

```
C:\WINDOWS\system32\cmd.exe  
C:\scctest>scc.exe -lncvrt -o struct_test.exe struct_test.c  
struct_test.c 编译成功  
  
C:\scctest>struct_test.exe  
pt.x = 1024, pt.y = 768  
ppt->x = 1024, ppt->y = 768  
sizeof<char>=1, sizeof<short>=2, sizeof<int>=4,  
                                         sizeof<struct point>=8  
sizeof<struct s1> = 8  
sizeof<struct s2> = ?  
sizeof<struct s3> = 12
```

图 11.4 结构体测试

11.2.4 函数参数传递测试

```
/* param_test.c 源文件 */  
*****  
* 功能：参数传值测试  
*****  

int add(int x, int y)  
{  
    int z;  
  
    z=x+y;  
    return z;  
}  
  
*****  
* 功能：参数传地址测试  
*****  

void param_ptr(int * px)  
{  
    * px=123;  
}  
  
*****  
* 功能：函数返回值作用实参测试  
*****
```

```
int func_as_param_test()
{
    int sum, a=1, b=2, c;

    sum=add(c=add(a,b),add(1,2));
    printf("sum=%d\n",sum);
}

/*****************
 * 功能：可变参数测试
 *****************/
void __cdecl sum(int num,...)
{
    int sum=0, i=0;
    int * p1;
    int * end;

    p1=&num;
    end=p1+num;
    for(p1=p1+1; p1<=end; p1=p1+1)
    {
        sum=sum+ * p1;
    }
    printf("sum=%d\n",sum);
}

/*****************
 * 功能：main 函数
 *****************/
int main()
{
    int n=0;

    printf("add(3,5)=%d\n",add(3,5));
    param_ptr(&n);
    printf("n=%d\n",n);
    sum(5,1,2,3,4,5);

    return 0;
}

/*****************
 * 功能：入口函数
 *****************/
void _entry()
```

```
{
    int ret;
    ret=main();
    exit(ret);
}
```

param_test.c 编译及运行结果如图 11.5 所示。

```
C:\>scctest>scc.exe -lmsvrt -o param_test.exe param_test.c
param_test.c 编译成功

C:>scctest>param_test.exe
add<3,5> = 8
n = 123
sum = 15
```

图 11.5 函数参数传递测试

11.2.5 字符串测试

```
/* string_test.c 源文件 */
/*****************/
/* 功能：字符集测试
/*****************/
void characterset_test()
{
    printf("Hello 哇,饭已 OK 了,下来めし吧!\n");
}

/*****************/
/* 功能：字符串测试
/*****************/
void string_test()
{
    char str1[]="abc\n";
    char * str2 ="XYZ\n";
    printf(str1);
    printf(str2);
}

/*****************/
/* 功能：字符测试
/*****************/
void char_test()
{
    char c1='E';
    char c2='GH';
    printf("c1=%c, c2=%c\n", c1, c2);
}
```

```

/*************
 * 功能: main 函数
 *****/
int main()
{
    characterset_test();
    string_test();
    char_test();

    return 0;
}

/*************
 * 功能: 入口函数
 *****/
void _entry()
{
    int ret;
    ret=main();
    exit(ret);
}

```

string_test.c 编译及运行结果如图 11.6 所示。

```

C:\WINDOWS\system32\cmd.exe
C:>ccctest>cc.exe -lmsvcrt -o string_test.exe string_test.c
string_test.c 编译成功

C:>ccctest>string_test.exe
Hello哇,饭已OK了,下来めし吧!
XYZ
c1 = E, c2 = G

```

图 11.6 字符串测试

11.2.6 全局变量测试

```

/* globalvar_test.c 源文件 */

/*************
 * 功能: 全局变量测试
 *****/
char g_char      = 'a';
short g_short   = 123;
int  g_int      = 123456;
char g_str1[]   = "SC 语言及 SCC 编译器会成为风靡全球的编译原理最佳教学用例吗? \n";
char * g_str2   = "也许会吧!\n";
void global_var_test()
{
    printf("g_char=%c, g_short=%d, g_int=%d\n",

```

```

        g_char,g_short,g_int);
printf(g_str1);
printf(g_str2);
}

/*****************
* 功能：main 函数
*****************/
int main()
{
    global_var_test();
    return 0;
}

/*****************
* 功能：入口函数
*****************/
void _entry()
{
    int ret;
    ret=main();
    exit(ret);
}

globalvar_test.c 编译及运行结果如图 11.7 所示。

```

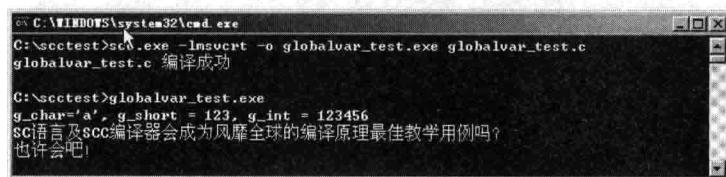


图 11.7 全局变量测试

11.3 语言举例

11.3.1 可接收命令行参数的控制台程序

```

/* console_entry.c 源文件 */
/*****************
* 功能：main 主函数
*****************/
int main(int argc, char **argv, char **env)
{
    printf(argv[1]);
    printf("\n");
}

```

```

    return 0;
}

//*****************************************************************************
* 功能：入口函数
*****
void _entry()
{
    int argc; char **argv; char **env; int ret;
    int start_info=0;

    __getmainargs(&argc, &argv, &env, 0, &start_info);
    ret=main(argc, argv, env);
    exit(ret);
}

```

Console_entry.c 编译及运行结果如图 11.8 所示。

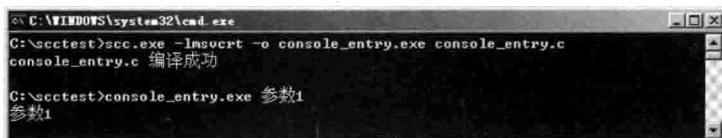


图 11.8 控制台入口测试

11.3.2 可接收命令行参数的 Win32 应用程序

```

/* win_entry.c 源文件 */
/* 程序用到的外部函数声明 */
Int __stdcall MessageBoxA(void * hWnd, char * lpText, char * lpCaption, int uType);
int __stdcall GetModuleHandleA(char * lpModuleName);
char * __stdcall GetCommandLineA();
***** 
* 功能：WinMain 主函数
*****
int __stdcall WinMain(void * hInstance, void * hPrevInstance,
                      char * szCmdLine, int iCmdShow)
{
    MessageBoxA(0,szCmdLine,"title",0);
    return 1;
}

***** 
* 功能：入口函数
*****
int _entry()
{

```

```

char * szCmd;

szCmd=GetCommandLineA();
exit(WinMain(GetModuleHandleA(0), 0, szCmd,10));
}

```

Win_entry.c 编译及运行结果如图 11.9 所示。



图 11.9 Win32 应用程序入口测试

11.3.3 HelloWindows 窗口程序

```

/* HelloWindows.c 源文件 */
/* 程序用到的结构定义 */
struct tagPOINT
{
    int x;
    int y;
};

struct tagMSG
{
    void * hwnd;
    int message;
    int wParam;
    int lParam;
    int time;
    struct tagPOINT pt;
};

struct tagWNDCLASSA
{
    int style;
    void * lpfnWndProc;
    int cbClsExtra;
    int cbWndExtra;
    void * hInstance;
    void * hIcon;
    void * hCursor;
}

```

```
void * hbrBackground;
char * lpszMenuName;
char * lpszClassName;
};

struct tagRECT
{
    int      left;
    int      top;
    int      right;
    int      bottom;
};

struct tagPAINTSTRUCT
{
    void *          hdc;
    int             fErase;
    struct tagRECT rcPaint;
    int             fRestore;
    int             fIncUpdate;
    char            rgbReserved[32];
};

struct _STARTUPINFOA
{
    int      cb;
    char *  lpReserved;
    char *  lpDesktop;
    char *  lpTitle;
    int      dwX;
    int      dwY;
    int      dwXSize;
    int      dwYSize;
    int      dwXCountChars;
    int      dwYCountChars;
    int      dwFillAttribute;
    int      dwFlags;
    short   wShowWindow;
    short   cbReserved2;
    char *  lpReserved2;
    void *  hStdInput;
    void *  hStdOutput;
    void *  hStdError;
};

};
```

```

/* 程序用到的函数声明 */
void * __stdcall LoadIconA(void * hInstance, char * lpIconName);
void * __stdcall LoadCursorA(void * hInstance, char * lpCursorName);
void * __stdcall GetStockObject(int c);
short __stdcall RegisterClassA(struct tagWNDCLASSA * lpWndClass);
int __stdcall MessageBoxA(void * hWnd, char * lpText, char * lpCaption, int uType);
void * __stdcall CreateWindowExA(int dwExStyle, char * lpClassName,
                                char * lpWindowName, int dwStyle, int X, int nWidth, int nHeight,
                                void * hWndParent, void * hMenu, void * hInstance, void * lpParam);
int __stdcall ShowWindow(void * hWnd, int nCmdShow);
int __stdcall UpdateWindow(void * hWnd);
int __stdcall GetMessageA(struct tagMSG * lpMsg, void * hWnd,
                          int wMsgFilterMin, int wMsgFilterMax);
int __stdcall TranslateMessage(struct tagMSG * lpMsg);
int __stdcall DispatchMessageA(struct tagMSG * lpMsg);
void * __stdcall BeginPaint(void * hWnd, struct tagPAINTSTRUCT * lpPaint);
int __stdcall GetClientRect(void * hWnd, struct tagRECT * lpRect);
int __stdcall DrawTextA(void * hDC, char * lpString, int nCount,
                      struct tagRECT * lpRect, int uFormat);
int __stdcall EndPaint(void * hWnd, struct tagPAINTSTRUCT * lpPaint);
int __stdcall GetModuleHandleA(char * lpModuleName);
char * __stdcall GetCommandLineA();
void GetStartupInfoA(struct _STARTUPINFOA * lpStartupInfo);
void __stdcall PostQuitMessage(int nExitCode);
int __stdcall DefWindowProcA(void * hWnd, int Msg, int wParam, int lParam);
int __stdcall WndProc(void * a, int b, int c, int d);

/* 程序用到的常量定义 */
char * szAppName = "HelloWin";
int NULL = 0;
int CS_VREDRAW = 1;
int CS_HREDRAW = 2;
int IDI_APPLICATION = 32512;
int IDC_ARROW = 32512;
int WHITE_BRUSH = 0;
int MB_ICONERROR = 16;
int WS_OVERLAPPEDWINDOW = 13565952; /* 0xcf0000 */
int CW_USEDEFAULT = 2147483648; /* 0x80000000 */
int WM_PAINT = 15;
int WM_DESTROY = 2;
int DT_SINGLELINE = 32;
int DT_CENTER = 1;
int DT_VCENTER = 4;
int SW_SHOWDEFAULT = 10;

```

```
*****  
* 功能：窗口消息处理函数  
*****  
int __stdcall WndProc(void * hwnd, int message, int wParam, int lParam)  
{  
    void *             hdc;  
    struct tagPAINTSTRUCT ps;  
    struct tagRECT     rect;  
  
    if(message==WM_PAINT)  
    {  
        hdc=BeginPaint(hwnd, &ps);  
        GetClientRect(hwnd, &rect);  
        DrawTextA(hdc, "Hello,world!", -1, &rect,  
                  DT_SINGLELINE+DT_CENTER+DT_VCENTER);  
        EndPaint(hwnd, &ps);  
        return 0;  
    }  
    else if(message==WM_DESTROY)  
    {  
        PostQuitMessage(0);  
        return 0;  
    }  
  
    return DefWindowProcA(hwnd, message, wParam, lParam);  
}
```

```
*****  
* 功能：WinMain 主函数  
*****  
int __stdcall WinMain(void * hInstance, void * hPrevInstance,  
                      char * szCmdLine, int iCmdShow)  
{  
    void *             hwnd;  
    struct tagMSG      msg;  
    struct tagWNDCLASSA wndclass;  
  
    wndclass.style      = CS_VREDRAW+CS_HREDRAW;  
    wndclass.lpfnWndProc = WndProc;  
    wndclass.cbClsExtra = 0;  
    wndclass.cbWndExtra = 0;  
    wndclass.hInstance  = hInstance;  
    wndclass.hIcon      = LoadIconA(NULL, IDI_APPLICATION);  
    wndclass.hCursor    = LoadCursorA(NULL, IDC_ARROW);
```

```
wndclass.hbrBackground =GetStockObject(WHITE_BRUSH);
wndclass.lpszMenuName =0;
wndclass.lpszClassName =szAppName;

if(RegisterClassA(&wndclass)==0)
{
    MessageBoxA(NULL, "Cann't RegisterClass!", szAppName, MB_ICONERROR);
    return 0;
}

hwnd=CreateWindowExA(0,szAppName,
                     szCmdLine,
                     WS_OVERLAPPEDWINDOW,
                     CW_USEDEFAULT,
                     CW_USEDEFAULT,
                     CW_USEDEFAULT,
                     CW_USEDEFAULT,
                     NULL,
                     NULL,
                     hInstance,
                     NULL);

if(hwnd==NULL)
    MessageBoxA(NULL,"error",szCmdLine,0);

ShowWindow(hwnd, iCmdShow);
UpdateWindow(hwnd);

for(;GetMessageA(&msg, NULL, 0, 0);)
{
    TranslateMessage(&msg);
    DispatchMessageA(&msg);
}

return msg.wParam;
}

/****************************************
* 功能：入口函数
****************************************/
int _entry()
{
    char * szcmd;

    szcmd=GetCommandLineA();
    exit(WinMain(GetModuleHandleA(0), NULL, szcmd,SW_SHOWDEFAULT));
}
```

HelloWindows.c 编译及运行结果如图 11.10 所示。



图 11.10 HelloWindows 窗口程序

11.3.4 文件复制程序

```
/* CopyFile.c 源文件 */  
  
/* 程序用到的结构定义 */  
struct _iobuf  
{  
    char * _ptr;  
    int   _cnt;  
    char * _base;  
    int   _flag;  
    int   _file;  
    int   _charbuf;  
    int   _bufsiz;  
    char * _tmpfname;  
};  
  
/*********************  
* 功能：main 主函数  
*****/  
int main(int argc, char **argv, char **env)  
{  
    int ch;  
    struct _iobuf * srcfile;  
    struct _iobuf * destfile;  
    char srcfname[300];  
    char destfname[300];
```

```
int i=0;
char buf[2000];
int NULL=0;
int EOF = -1;
int SEEK_SET=0;

printf("源文件名:");
scanf("%s",srcfname);

printf("目标文件名:");
scanf("%s",destfname);

srcfile=fopen(srcfname,"rb");
if(srcfile==NULL)
{
    printf("读入文件未找到!\n");
    printf("按任意键继续...");
    getchar();
    exit(1);
}

destfile=fopen(destfname, "wb");
if(destfile==NULL)
{
    printf("写入文件未找到!\n");
    printf("按任意键继续...");
    getchar();
    fclose(srcfile);
    exit(1);
}

for(;(ch=getc(srcfile))!=EOF;)
{
    putchar(ch);
    fwrite(&ch,1,1,destfile);
}

fclose(srcfile);
fclose(destfile);

printf("按任意键继续...");
getchar();
return 0;
}
```

```
*****  
* 功能：入口函数  
*****  
void _entry()  
{  
    int argc; char **argv; char **env; int ret;  
    int start_info=0;  
  
    __getmainargs(&argc, &argv, &env, 0, &start_info);  
    ret=main(argc, argv, env);  
    exit(ret);  
}
```

CopyFile.c 编译及运行结果如图 11.11 所示。

```
C:\>scctest>CopyFile.exe -lmsvcrt -o CopyFile.exe CopyFile.c
CopyFile.c 编译成功

C:\>scctest>CopyFile.exe
源文件名: console_entry.c
目标文件名: console_entry.txt
/* console_entry.c 源文件 */
*****  
* 功能:      main主函数  
*****  
int main<int argc, char **argv, char **env>
{
    printf(argv[1]);
    printf("\n");
    return 0;
}

*****  
* 功能:      入口函数  
*****  
void _entry()
{
    int argc; char **argv; char **env; int ret;
    int start_info = 0;

    __getmainargs(&argc, &argv, &env, 0, &start_info);
    ret = main(argc, argv, env);
    exit(ret);
}>按任意键继续...
```

图 11.11 文件复制举例

11.3.5 九九乘法表

```
/* multiplication_table.c 源文件 */  
*****  
* 功能：全局变量测试  
*****  
void multiplication_table()  
{  
    int i,j;  
    for(i=1; i<=9; i=i+1)  
    {  
        for(j=1; j<=i; j=j+1)  
        {  
            printf("%d * %d=%d ",j,i,j * i);  
        }  
    }  
}
```

```

    }
    printf("\n");
}
}

/*****************
* 功能：main 函数
*****************/
int main()
{
    multiplication_table();
    return 0;
}

/*****************
* 功能：入口函数
*****************/
void _entry()
{
    int ret;
    ret=main();
    exit(ret);
}

```

multiplication_table.c 编译及运行结果如图 11.12 所示。

```

C:\scctest>gcc.exe -fmsvcrt -o multiplication_table.exe multiplication_table.c
multiplication_table.c 编译成功

C:\scctest>multiplication_table.exe
1*1=1
1*2=2 2*2=4
1*3=3 2*3=6 3*3=9
1*4=4 2*4=8 3*4=12 4*4=16
1*5=5 2*5=10 3*5=15 4*5=20 5*5=25
1*6=6 2*6=12 3*6=18 4*6=24 5*6=30 6*6=36
1*7=7 2*7=14 3*7=21 4*7=28 5*7=35 6*7=42 7*7=49
1*8=8 2*8=16 3*8=24 4*8=32 5*8=40 6*8=48 7*8=56 8*8=64
1*9=9 2*9=18 3*9=27 4*9=36 5*9=45 6*9=54 7*9=63 8*9=72 9*9=81

```

图 11.12 九九乘法表

11.3.6 打印菱形

```

/* diamond.c 源文件 */
/*****************
* 功能：打印菱形
*****************/
void diamond()
{
    int i, j; /* 定义循环变量 */
    int k; /* 定义变量保存用户的输入 */

```

```

printf("请输入菱形大小:");
scanf("%d",&k);

/* 循环开始打印出空心菱形的上半部分,
外循环控制行数,内循环控制每行的输出 */
for(i=0; i<k; i=i+1)
{
    /* 控制菱形左边的空格输出 */
    for(j=0; j<k-1-i; j=j+1)
    {
        printf(" ");
    }

    /* 如果是菱形的边界用 "*" 输出,否则用空格填充 */
    for(j=0; j<2 * i+1; j=j+1)
    {
        if(0==j)
        {
            printf("* ");
        }
        else if(2 * i==j)
            printf("* ");
        else
        {
            printf(" ");
        }
    }

    printf("\n");           /* 每行结束的时候换行 */
}

/* 循环开始打印出空心菱形的下半部分,
外循环控制行数,内循环控制每行的输出 */
for(i=k-2; i>=0; i=i-1)
{
    /* 控制菱形左边的空格输出 */
    for(j=0; j<k-1-i; j=j+1)
    {
        printf(" ");
    }

    /* 如果是菱形的边界用 "*" 输出,否则用空格填充 */
    for(j=0; j<2 * i+1; j=j+1)
    {
}

```

```

        if(0==j)
        {
            printf(" * ");
        }
        else if(2 * i==j)
            printf(" * ");
        else
        {
            printf(" ");
        }
    }
    printf("\n");           /* 每行结束的时候换行 */
}

*******/

* 功能: main 函数
*******/

int main()
{
    diamond();
    return 0;
}

*******/

* 功能: 人口函数
*******/

void _entry()
{
    int ret;
    ret=main();
    exit(ret);
}

```

diamond.c 编译及运行结果如图 11.13 所示。

11.3.7 屏幕捕捉程序

看完上面的程序大家可能感觉还不够过瘾,可能在想能不能举个更综合、更有代表性,并且具有一定实用价值的程序?大家的这点心思作者早就猜到了,作者为大家精心准备了本书的压轴程序,用 SC 语言编写一个屏幕捕捉程序。

```

/* screen_capture.c 源文件 */
/* 程序用到的结构定义 */
struct tagWNDCLASSA {

```

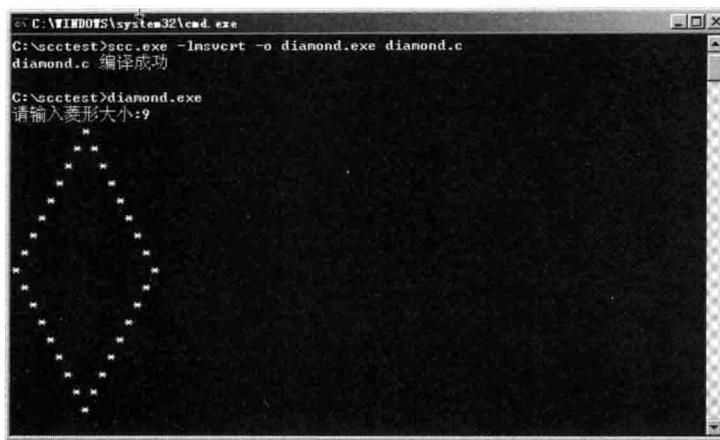


图 11.13 打印菱形

```

    int      style;
    void * lpfnWndProc;
    int      cbClsExtra;
    int      cbWndExtra;
    void * hInstance;
    void * hIcon;
    void * hCursor;
    void * hbrBackground;
    char * lpszMenuName;
    char * lpszClassName;
};

struct HBITMAP__           { int unused; };
struct HDC__                { int unused; };
struct HWND__               { int unused; };
struct HINSTANCE__          { int unused; };
struct _iobuf {
    char * _ptr;
    int   _cnt;
    char * _base;
    int   _flag;
    int   _file;
    int   _charbuf;
    int   _bufsiz;
    char * _tmpfname;
};

struct BITMAPINFOHEADER{
    int      biSize;
    int      biWidth;

```

```

        int      biHeight;
        short    biPlanes;
        short    biBitCount;
        int      biCompression;
        int      biSizeImage;
        int      biXPelsPerMeter;
        int      biYPelsPerMeter;
        int      biClrUsed;
        int      biClrImportant;
};

struct RGBQUAD {
    char      rgbBlue;
    char      rgbGreen;
    char      rgbRed;
    char      rgbReserved;
};

struct BITMAPINFO {
    struct BITMAPINFOHEADER    bmiHeader;
    struct RGBQUAD             bmiColors[1];
};

struct BITMAPFILEHEADER {
    short     bfType;
    int       __align(2)      bfSize;
    short     bfReserved1;
    short     bfReserved2;
    int       __align(2)      bfOffBits;
};

struct OPENFILENAMEA {
    int          lStructSize;
    struct HWND__ *      hwndOwner;
    struct HINSTANCE__ * hInstance;
    char *        lpstrFilter;
    char *        lpstrCustomFilter;
    int           nMaxCustFilter;
    int           nFilterIndex;
    char *        lpstrFile;
    int           nMaxFile;
    char *        lpstrFileTitle;
    int           nMaxFileTitle;
    char *        lpstrInitialDir;
    char *        lpstrTitle;
};

```

```
int             Flags;
short           nFileOffset;
short           nFileExtension;
char *          lpstrDefExt;
int             lCustData;
void *          lpfnHook;
char *          lpTemplateName;
};

struct tagWNDCLASSA * test_return_struct_pointer()
{
    struct tagWNDCLASSA     wndclass;
    wndclass.style         = 654321;
    wndclass.lpfnWndProc  = 0;
    wndclass.cbClsExtra   = 0;
    return &wndclass;
}

int test_structassign()
{
    struct tagWNDCLASSA     wndclass;
    wndclass.style         = 3;
    wndclass.lpfnWndProc  = 0;
    wndclass.cbClsExtra   = 0;
    return 1;
}

/* 程序用到的常量定义 */
int MB_OK=0;           /* #define MB_OK 0x00000000L */
int MB_ICONERROR=16;
int BI_RGB=0;
int DIB_RGB_COLORS=0;
int SM_CXSCREEN=0;
int SM_CYSCREEN=1;
int SRCCOPY=13369376; /* #define SRCCOPY      (DWORD) 0x00CC0020 */
int NULL=0;
int OFN_HIDEREADONLY=4;
int OFN_PATHMUSTEXIST=2048; /* #define OFN_PATHMUSTEXIST      0x00000800 */

/* 程序用到的函数声明 */
struct HDC__ *        __stdcall GetDC();
struct HWND__ *        __stdcall GetDesktopWindow();
struct HBITMAP__ *     __stdcall CreateCompatibleBitmap();
int     __stdcall GetDIBits();
int     __stdcall MessageBoxA();
```

```

int      __cdecl    fwrite();
int  __stdcall ReleaseDC();
int  __stdcall GetSaveFileNameA();
int  __stdcall GetSystemMetrics();
void*   __stdcall SelectObject();
int   __stdcall BitBlt();
int   __stdcall DeleteDC();
int   __stdcall DeleteObject();
void* __cdecl     memset();
void  __cdecl     free();
int   __cdecl     fclose();
char* __cdecl     strcpy();
int   __cdecl     abs();

/******************
 * 功能：    保存位图文件
 * szFilename: 位图文件名
 * hBitmap:    位图句柄
*****************/
void SaveBitmap(char * szFilename,struct HBITMAP __* hBitmap)
{
    struct HDC __*                      hdc=NULL;
    struct FILE *                       fp=NULL;
    void*                                pBuf=NULL;
    struct BITMAPINFO                   bmpInfo;
    struct BITMAPFILEHEADER             bmpFileHeader;
    for(;;)
    {
        hdc=GetDC(NULL);

        memset(&bmpInfo,0,sizeof(struct BITMAPINFO));
        bmpInfo.bmiHeader.biSize=sizeof(struct BITMAPINFOHEADER);
        GetDIBits(hdc,hBitmap,0,0,NULL,&bmpInfo,DIB_RGB_COLORS);

        if(bmpInfo.bmiHeader.biSizeImage<=0)
            bmpInfo.bmiHeader.biSizeImage = bmpInfo.bmiHeader.biWidth * abs(bmpInfo.bmiHeader.biHeight) * (bmpInfo.bmiHeader.biBitCount+7)/8;

        if((pBuf=malloc(bmpInfo.bmiHeader.biSizeImage))==NULL)
        {
            MessageBoxA(NULL,"Unable to Allocate Bitmap Memory","Error",MB_OK+MB_ICONERROR);
            break;
        }
    }
}

```

```
    bmpInfo.bmiHeader.biCompression=BI_RGB;
    GetDIBits(hdc,hBitmap,0,bmpInfo.bmiHeader.biHeight,pBuf,&bmpInfo,DIB_
RGB_COLORS);

    fp=fopen(szFilename,"wb");
    if(fp==NULL)
    {
        MessageBoxA(NULL,"Unable to Create Bitmap File","Error",MB_OK+MB_
ICONERROR);
        break;
    }

    bmpFileHeader.bfReserved1=0;
    bmpFileHeader.bfReserved2=0;
    bmpFileHeader.bfSize = sizeof ( struct BITMAPFILEHEADER ) + sizeof ( struct
BITMAPINFOHEADER) + bmpInfo.bmiHeader.biSizeImage;
    bmpFileHeader.bfType=19778;           /* 0x424d "BM" */
    bmpFileHeader.bfOffBits=sizeof (struct BITMAPFILEHEADER) + sizeof (struct
BITMAPINFOHEADER);

    fwrite(&bmpFileHeader,sizeof(struct BITMAPFILEHEADER),1,fp);
    fwrite(&bmpInfo.bmiHeader,sizeof(struct BITMAPINFOHEADER),1,fp);
    fwrite(pBuf,bmpInfo.bmiHeader.biSizeImage,1,fp);
    break;
}

if(hdc)
    ReleaseDC(NULL,hdc);

if(pBuf)
    free(pBuf);

if(fp)
    fclose(fp);
}

/**************************************************************************
* 功能: main 函数
*************************************************************************/
int main()
{
    struct             OPENFILENAMEA      ofn;
    char               strFileName[512];
    int                nWidth;
    int                nHeight;
```

```

    struct HDC__ *          hBmpFileDC;
    struct HWND__ *         hDesktopWnd;
    struct HDC__ *          hDesktopDC;
    struct HBITMAP__ *      hBmpFileBitmap;
    struct HBITMAP__ *      hOldBitmap;
    struct tagWNDCLASSA *   pwndclass;

    strcpy(strFileName,"ScreenShot.bmp");
    memset(&ofn,0,sizeof(struct OPENFILENAMEA));
    ofn.lStructSize = sizeof(struct OPENFILENAMEA);
    ofn.Flags=OFN_HIDEREADONLY+OFN_PATHMUSTEXIST;
    ofn.lpstrFilter="Bitmap Files (*.bmp) | *.bmp";
    ofn.lpstrDefExt="bmp";
    ofn.lpstrFile=strFileName;
    ofn.nMaxFile=512;
    ofn.hwndOwner=NULL;
    if(GetSaveFileNameA(&ofn)==0)      return 1;

    hDesktopWnd=GetDesktopWindow();
    hDesktopDC=GetDC(hDesktopWnd);
    nWidth=GetSystemMetrics(SM_CXSCREEN);
    nHeight=GetSystemMetrics(SM_CYSCREEN);
    hBmpFileDC=CreateCompatibleDC(hDesktopDC);
    hBmpFileBitmap=CreateCompatibleBitmap(hDesktopDC,nWidth,nHeight);
    hOldBitmap=SelectObject(hBmpFileDC,hBmpFileBitmap);
    BitBlt(hBmpFileDC,0,0,nWidth,nHeight,hDesktopDC,0,0,SRCCOPY);
    SelectObject(hBmpFileDC,hOldBitmap);

    SaveBitmap(ofn.lpstrFile,hBmpFileBitmap);

    DeleteDC(hBmpFileDC);
    DeleteObject(hBmpFileBitmap);
    return 0;
}

/****************************************
 * 功能：入口函数
****************************************/
void _entry()
{
    int ret;
    ret=main();
    exit(ret);
}

```

screen_capture.c 编译及运行结果如图 11.14 所示。

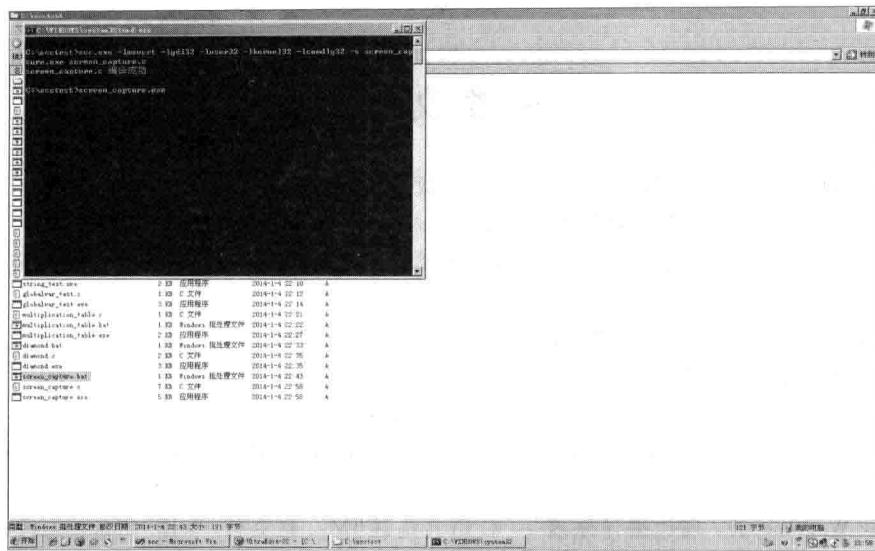


图 11.14 屏幕捕捉程序

至此,SCC 编译器、链接器就算大功告成了。作者作为本次旅行的导游,非常依依不舍地跟大家说:“本次旅行圆满结束,感谢大家的一路相伴”。

参 考 文 献

- [1] 张素琴,吕映之,蒋维杜,戴桂兰. 编译原理(第2版)[M]. 北京: 清华大学出版社, 2005.
- [2] 王生原,董渊,杨萍,张素琴. 编译原理[M]. 北京: 人民邮电出版社, 2010.
- [3] 陈火旺,刘春林,谭庆平,赵克佳,刘越. 程序设计语言编译原理(第3版)[M]. 北京: 国防工业出版社, 2011.
- [4] 陈意云,张昱. 编译原理[M]. 2 版. 北京: 高等教育出版社, 2008.
- [5] 温敬和. 编译原理实用教程[M]. 北京: 清华大学出版社, 2005.
- [6] 李冬梅,施海虎. 编译原理[M]. 北京: 人民邮电出版社, 2006.
- [7] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman 著. 编译原理[M]. 赵建华, 等译, 北京: 机械工业出版社, 2009.
- [8] Andrew W. Appel 著, 现代编译原理——C 语言描述[M]. 赵克佳, 等译. 北京: 人民邮电出版社, 2006.
- [9] Steven Muchnick 著. 高级编译器设计与实现[M]., 北京: 机械工业出版社, 2003.
- [10] Thomas Pittman, James Peters 著. 编译程序设计艺术理论与实践[M]. 李文军,高晓燕译. 北京: 机械工业出版社, 2010.
- [11] Kenneth C. Louden, Compiler Construction Principles and practice[M]. 北京: 机械工业出版社, 2002.
- [12] Brian W. Kernighan, Dennis M. Ritchie 著. C 程序设计语言[M]. 2 版. 徐宝文, 等译. 北京: 机械工业出版社, 2011.
- [13] 吴哲辉,吴振寰. 形式语言与自动机理论[M]. 北京: 机械工业出版社, 2007.
- [14] 谭浩强. C 程序设计[M]. 北京: 清华大学出版社, 1999.
- [15] IA Software Developer's Manual Volumne2-Instruction Set Reference. 2004.
- [16] 于渊. 自己动手写操作系统[M]. 北京: 电子工业出版社, 2005.
- [17] Visual Studio, Microsoft Portable Executable and Common Object File Format Specification, 2006.
- [18] 罗云彬. 琢石成器——Windows 环境下 32 位汇编语言程序设计[M]. 北京: 电子工业出版社, 2009.
- [19] 赵树升,杨建军. DOS/Windows 汇编语言程序设计教程[M]. 北京: 清华大学出版社, 2005.
- [20] John R. Levine 著. 链接器和加载器[M]. 李勇译. 北京: 北京航空航天大学出版社, 2009.
- [21] Jeffrey Richter 著. Windows 核心编程[M]. 王建华, 等译. 北京: 机械工业出版社, 2000.

附录 A

SC语言文法定义中英文对照表

序号	英 文 表 示	中 文 表 示
词 法 定 义		
1	<code><KW_CHAR> ::= "char"</code>	<code><char 关键字> ::= "char"</code>
2	<code><KW_SHORT> ::= "short"</code>	<code><short 关键字> ::= "short"</code>
3	<code><KW_INT> ::= "int"</code>	<code><int 关键字> ::= "int"</code>
4	<code><KW_VOID> ::= "void"</code>	<code><void 关键字> ::= "void"</code>
5	<code><KW_STRUCT> ::= "struct"</code>	<code><struct 关键字> ::= "strut"</code>
6	<code><KW_IF> ::= "if"</code>	<code><if 关键字> ::= "if"</code>
7	<code><KW_ELSE> ::= "else"</code>	<code><else 关键字> ::= "else"</code>
8	<code><KW_FOR> ::= "for"</code>	<code><for 关键字> ::= "for"</code>
9	<code><KW_CONTINUE> ::= "continue"</code>	<code><continue 关键字> ::= "continue"</code>
10	<code><KW_BREAK> ::= "break"</code>	<code><break 关键字> ::= "break"</code>
11	<code><KW_RETURN> ::= "return"</code>	<code><return 关键字> ::= "return"</code>
12	<code><KW_SIZEOF> ::= "sizeof"</code>	<code><sizeof 关键字> ::= "sizeof"</code>
13	<code><KW_CDECL> ::= "__cdecl"</code>	<code><__cdecl 关键字> ::= "__cdecl"</code>
14	<code><KW_STDCALL> ::= "__stdcall"</code>	<code><__stdcall 关键字> ::= "__stdcall"</code>
15	<code><KW_PACK> ::= "__pack"</code>	<code><__pack 关键字> ::= "__pack"</code>
16	<code><KW_ALIGN> ::= "__align"</code>	<code><__align 关键字> ::= "__align"</code>
17	<code><IDENTIFIER> ::= <nondigit> {<nondigit> <digit>}</code>	<code><标识符> ::= <非数字> { <非数字> <数字> }</code>
18	<code><TK_CINT> ::= <digit> {<digit>}</code>	<code><整数常量> ::= <数字> {<数字>}</code>
19	<code><nondigit> ::= "_" "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z" "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"</code>	<code><非数字> ::= "_" "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z" "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"</code>

续表

序号	英文表示	中文表示
20	<code><digit> ::= "0" "1" "2" "3" "4" "5" "6" "7" "8" "9"</code>	<code><数字> ::= "0" "1" "2" "3" "4" "5" "6" "7" "8" "9"</code>
21	<code><TK_CSTR> ::= " " {<s-char>} "</code>	<code><常量字符串> ::= " {<串字符>} "</code>
22	<code><s-char> ::= <escape-sequence> any member of the source character set except the double-quotation mark ("'), backslash (\), or newline character</code>	<code><串字符> ::= <转义字符> 源字符集中除双引号字符"、反斜线字符\或新行字符外的任何字符</code>
23	<code><TK_CCHAR> ::= ""<c-char>""</code>	<code><字符常量> ::= '<C-字符>'</code>
24	<code><c-char> ::= <escape-sequence> Any member of the source character set except the single quotation mark ('), backslash (\), or newline character</code>	<code><C-字符> ::= <转义序列> 源字符集中除单引号'、反斜线字符\或新行字符外的任何字符</code>
25	<code><escape-sequence> ::= "\\" "\"" "\\\" "\\a" "\\b" "\\f" "\\n" "\\r" "\\t" "\\v"</code>	<code><转义序列> ::= "\\" "\"" "\\\" "\\a" "\\b" "\\f" "\\n" "\\r" "\\t" "\\v"</code>
26	<code><TK_PLUS> ::= "+"</code>	<code><加号> ::= "+"</code>
27	<code><TK_MINUS> ::= "-"</code>	<code><减号> ::= "-"</code>
28	<code><TK_STAR> ::= "*"</code>	<code><星号> ::= "*"</code>
29	<code><TK_DIVIDE> ::= "/"</code>	<code><除号> ::= "/"</code>
30	<code><TK_MOD> ::= "%"</code>	<code><取余号> ::= "%"</code>
31	<code><TK_EQ> ::= "=="</code>	<code><等于号> ::= "=="</code>
32	<code><TK_NEQ> ::= "!="</code>	<code><不等于号> ::= "!="</code>
33	<code><TK_LT> ::= "<"</code>	<code><小于号> ::= "<"</code>
34	<code><TK_LEQ> ::= "<="</code>	<code><小于等于号> ::= "<="</code>
35	<code><TK_GT> ::= ">"</code>	<code><大于号> ::= ">"</code>
36	<code><TK_GEQ> ::= ">="</code>	<code><大于等于号> ::= ">="</code>
37	<code><TK_ASSIGN> ::= "="</code>	<code><赋值等号> ::= "="</code>
38	<code><TK_POINTSTO> ::= "->"</code>	<code><箭头> ::= "->"</code>
39	<code><TK_DOT> ::= ". "</code>	<code><点号> ::= ". "</code>
40	<code><TK_AND> ::= "&."</code>	<code><与号> ::= "&."</code>
41	<code><TK_OPENPA> ::= "("</code> <code><TK_CLOSEPA> ::= ")"</code>	<code><左小括号> ::= "("</code> <code><右小括号> ::= ")"</code>
42	<code><TK_OPENBR> ::= "["</code>	<code><左中括号> ::= "["</code>
43	<code><TK_CLOSEBR> ::= "]"</code>	<code><右中括号> ::= "]"</code>
44	<code><TK_BEGIN> ::= "{"</code>	<code><左大括号> ::= "{"</code>
45	<code><TK_END> ::= "}"</code>	<code><右大括号> ::= "}"</code>

续表

序号	英 文 表 示	中 文 表 示
46	<TK_SEMICOLON> ::= ";"	<分号> ::= ";"
47	<TK_COMMA> ::= ","	<逗号> ::= ","
48	<TK_ELLIPSIS> ::= "..."	<省略号> ::= "..."
语 法 定 义		
1	<translation_unit> ::= { external_declarator }<TK_EOF>	<翻译单元> ::= { <外部声明> } <文件结束符>
2	<external_declarator> ::= <function_definition> <declaration>	<外部声明> ::= <函数定义> <声明>
3	<function_definition> ::= <typeSpecifier><declarator><funcbody>	<函数定义> ::= <类型区分符> <声明符> <函数体>
4	<funcbody> ::= <compound_statement>	<函数体> ::= <复合语句>
5	<declaration> ::= <typeSpecifier> <TK_SEMICOLON> <typeSpecifier> <init_declarator_list> <TK_SEMICOLON>	<声明> ::= <类型区分符> [<初值声明符表>] <分号>
6	<init_declarator_list> ::= <init_declarator> { <TK_COMMA> <init_declarator> }	<初值声明符表> ::= <初值声明符> { <逗号> <初值声明符> }
7	<init_declarator> ::= <declarator> <declarator> <TK_ASSIGN> <initializer>	<初值声明符> ::= <声明符> <声明符> <赋值运算符> <初值符>
8	<typeSpecifier> ::= <KW_INT> <KW_CHAR> <KW_SHORT> <KW_VOID> <structSpecifier>	<类型区分符> ::= <void 关键字> <char 关键字> <short 关键字> <int 关键字> <结构区分符>
9	<structSpecifier> ::= <KW_STRUCT> <IDENTIFIER> <TK_BEGIN> <struct_declaration_list> <TK_END> <KW_STRUCT> <IDENTIFIER>	<结构区分符> ::= <struct 关键字> <标识符> <左大括号> <结构声明表> <右大括号> <struct 关键字> <标识符>
10	<struct_declaration_list> ::= <struct_declaration> { <struct_declaration> }	<结构声明表> ::= <结构声明> { <结构声明> }
11	<struct_declaration> ::= <typeSpecifier> <struct_declarator_list> <TK_SEMICOLON>	<结构声明> ::= <类型区分符> { <结构声明符表> } <分号>
12	<struct_declarator_list> ::= <declarator> { <TK_COMMA> <declarator> }	<结构声明符表> ::= <声明符> { <逗号> <声明符> }
13	<declarator> ::= { <pointer> } [<function_calling_convention>] [<struct_member_alignment>] <direct_declarator>	<声明符> ::= { <指针> } [<调用约定>] [<结构成员对齐>] <直接声明符>
14	<function_calling_convention> ::= <KW_CDECL> <KW_STDCALL>	<调用约定> ::= <__cdecl 关键字> <__stdcall 关键字>

续表

序号	英 文 表 示	中 文 表 示
15	<code>< struct _ member _ alignment > ::= = < KW _ ALIGN ><TK_OPENPA><TK_CINT><TK_CLOSEPA></code>	<code><结构成员对齐> ::= = <__align 关键字><左小括号><整数常量><右小括号></code>
16	<code><pointer> ::= = <TK_STAR></code>	<code><指针> ::= = <星号></code>
17	<code><direct_declarator> ::= = <IDENTIFIER></code> <code><direct_declarator_postfix></code>	<code><直接声明符> ::= = <标识符><直接声明符后缀></code>
18	<code>< direct _ declarator _ postfix > ::= = { < TK _ OPENBR ><TK_CINT><TK_CLOSEBR></code> <code><TK_OPENBR> <TK_CLOSEBR></code> <code><TK_OPENPA><parameter_type_list></code> <code><TK_CLOSEPA></code> <code><TK_OPENPA> <TK_CLOSEPA> }</code>	<code><直接声明符后缀> ::= = {<左中括号><右中括号></code> <code> <左中括号><整数常量><右中括号></code> <code> <左小括号><右小括号></code> <code> <左小括号><形参表><右小括号> }</code>
19	<code><parameter_type_list> ::= = <parameter_list> </code> <code><parameter_list> < TK _ COMMA >< TK _ ELLIPSIS ></code>	<code><形参类型表> ::= = <形参表> <形参表><逗号><省略号></code>
20	<code><parameter_list> ::= = <parameter_declaration></code> <code>{<TK_COMMA><parameter_declaration>}</code>	<code><形参表> ::= = <参数声明> {<逗号><参数声明>}</code>
21	<code><parameter_declaration> ::= = <typeSpecifier></code> <code><declarator> <typeSpecifier></code>	<code><参数声明> ::= = <类型区分符><声明符></code>
22	<code><initializer> ::= = <assignment_expression></code>	<code><初值符> ::= = <赋值表达式></code>
23	<code><statement> ::= = <compound_statement> </code> <code><if_statement> < return _ statement > </code> <code><break_statement> <continue_statement> </code> <code><for_statement> <expression_statement></code>	<code><语句> ::= = {<复合语句> <if语句> </code> <code><for语句> <break语句> <continue语句> <return语句> <表达式语句> }</code>
24	<code><compound_statement> ::= = <TK_BEGIN></code> <code>{<declaration>}<statement><TK_END></code>	<code><复合语句> ::= = <左大括号> {<声明>}</code> <code>{<语句>}<右大括号></code>
25	<code>< expression _ statement > ::= = < TK _ SEMICOLON > < expression > < TK _ SEMICOLON ></code>	<code><表达式语句> ::= = [<表达式>]<分号></code>
26	<code><if_statement> ::= = <KW_IF><TK_OPENPA></code> <code><expression><TK_CLOSEPA><statement></code> <code>[<KW_ELSE><statement>]</code>	<code><if语句> ::= = <if关键字><左小括号></code> <code><表达式><右小括号><语句> [<else关键字><语句>]</code>
27	<code><for _ statement> ::= = < KW _ FOR >< TK _ OPENPA ></code> <code>< expression _ statement ></code> <code>< expression_statement >< expression >< TK _ CLOSEPA >< statement ></code>	<code><for语句> ::= = <for关键字><左小括号></code> <code><表达式语句><表达式语句><表达式></code> <code><右小括号><语句></code>
28	<code><continue_statement> ::= = <KW_CONTINUE></code> <code><TK_SEMICOLON></code>	<code><continue语句> ::= = <continue关键字></code> <code><分号></code>
29	<code><break_statement> ::= = <KW_BREAK></code> <code><TK_SEMICOLON></code>	<code><break语句> ::= = <break关键字><分号></code>

续表

序号	英 文 表 示	中 文 表 示
30	$\langle \text{return_statement} \rangle ::= \langle \text{KW_RETURN} \rangle \langle \text{TK_SEMICOLON} \rangle$ $\langle \text{KW_RETURN} \rangle \langle \text{expression} \rangle \langle \text{TK_SEMICOLON} \rangle$	$\langle \text{return 语句} \rangle ::= \langle \text{return 关键字} \rangle \langle \text{expression} \rangle \langle \text{分号} \rangle$
31	$\langle \text{expression} \rangle ::= \langle \text{assignment_expression} \rangle \{ \langle \text{TK_COMMA} \rangle \langle \text{assignment_expression} \rangle \}$	$\langle \text{表达式} \rangle ::= \langle \text{赋值表达式} \rangle \{ \langle \text{逗号} \rangle \langle \text{赋值表达式} \rangle \}$
32	$\langle \text{assignment_expression} \rangle ::= \langle \text{equality_expression} \rangle$ $\langle \text{unary_expression} \rangle$ $\langle \text{TK_ASSIGN} \rangle \langle \text{assignment_expression} \rangle :$	$\langle \text{赋值表达式} \rangle ::= \langle \text{相等类表达式} \rangle$ $\langle \text{一元表达式} \rangle \langle \text{赋值等号} \rangle \langle \text{赋值表达式} \rangle$
33	$\langle \text{equality_expression} \rangle ::= \langle \text{relational_expression} \rangle \{$ $\langle \text{TK_EQ} \rangle \langle \text{relational_expression} \rangle$ $\langle \text{TK_NEQ} \rangle \langle \text{relational_expression} \rangle \}$	$\langle \text{相等类表达式} \rangle ::= \langle \text{关系表达式} \rangle \{$ $\langle \text{等于号} \rangle \langle \text{关系表达式} \rangle$ $\langle \text{不等于号} \rangle \langle \text{关系表达式} \rangle \}$
34	$\langle \text{relational_expression} \rangle ::= \langle \text{additive_expression} \rangle \{$ $\langle \text{TK_LT} \rangle \langle \text{additive_expression} \rangle$ $\langle \text{TK_GT} \rangle \langle \text{additive_expression} \rangle$ $\langle \text{TK_LEQ} \rangle \langle \text{additive_expression} \rangle$ $\langle \text{TK_GEQ} \rangle \langle \text{additive_expression} \rangle \}$	$\langle \text{关系表达式} \rangle ::= \langle \text{加减类表达式} \rangle \{$ $\langle \text{小于号} \rangle \langle \text{加减类表达式} \rangle$ $\langle \text{大于号} \rangle \langle \text{加减类表达式} \rangle$ $\langle \text{小于等于号} \rangle \langle \text{加减类表达式} \rangle$ $\langle \text{大于等于号} \rangle \langle \text{加减类表达式} \rangle \}$
35	$\langle \text{additive_expression} \rangle ::= \langle \text{multiplicative_expression} \rangle \{$ $\langle \text{TK_PLUS} \rangle \langle \text{multiplicative_expression} \rangle $ $\langle \text{TK_MINUS} \rangle \langle \text{multiplicative_expression} \rangle \}$	$\langle \text{加减类表达式} \rangle ::= \langle \text{乘除类表达式} \rangle \{$ $\langle \text{加号} \rangle \langle \text{乘除类表达式} \rangle $ $\langle \text{减号} \rangle \langle \text{乘除类表达式} \rangle \}$
36	$\langle \text{multiplicative_expression} \rangle ::= \langle \text{unary_expression} \rangle \{$ $\langle \text{TK_STAR} \rangle \langle \text{unary_expression} \rangle$ $\langle \text{TK_DIVIDE} \rangle \langle \text{unary_expression} \rangle$ $\langle \text{TK_MOD} \rangle \langle \text{unary_expression} \rangle \}$	$\langle \text{乘除类表达式} \rangle ::= \langle \text{一元表达式} \rangle \{$ $\langle \text{星号} \rangle \langle \text{一元表达式} \rangle$ $\langle \text{除号} \rangle \langle \text{一元表达式} \rangle$ $\langle \text{取余运算符} \rangle \langle \text{一元表达式} \rangle \}$
37	$\langle \text{unary_expression} \rangle ::= \langle \text{postfix_expression} \rangle$ $\langle \text{TK_AND} \rangle \langle \text{unary_expression} \rangle$ $\langle \text{TK_STAR} \rangle \langle \text{unary_expression} \rangle$ $\langle \text{TK_PLUS} \rangle \langle \text{unary_expression} \rangle$ $\langle \text{TK_MINUS} \rangle \langle \text{unary_expression} \rangle$ $\langle \text{sizeof_expression} \rangle$	$\langle \text{一元表达式} \rangle ::= \langle \text{后缀表达式} \rangle$ $\langle \text{与号} \rangle \langle \text{一元表达式} \rangle$ $\langle \text{星号} \rangle \langle \text{一元表达式} \rangle$ $\langle \text{加号} \rangle \langle \text{一元表达式} \rangle$ $\langle \text{减号} \rangle \langle \text{一元表达式} \rangle$ $\langle \text{sizeof 表达式} \rangle$
38	$\langle \text{sizeof_expression} \rangle ::= \langle \text{KW_SIZEOF} \rangle \langle \text{TK_OPENPA} \rangle \langle \text{type_specifier} \rangle \langle \text{TK_CLOSEPA} \rangle$	$\langle \text{sizeof 表达式} \rangle ::= \langle \text{sizeof 关键字} \rangle \langle \text{类型区分符} \rangle$

续表

序号	英 文 表 示	中 文 表 示
39	$\begin{aligned} <\text{postfix_expression}> ::= & <\text{primary_expression}> \{ \\ & <\text{TK_OPENBR}> <\text{expression}> <\text{TK_CLOSEBR}> \\ & <\text{TK_OPENPA}><\text{TK_CLOSEPA}> \\ & <\text{TK_OPENPA}><\text{argument_expression_list}> \\ & <\text{TK_CLOSEPA}> \\ & <\text{TK_DOT}> <\text{IDENTIFIER}> \\ & <\text{TK_POINTSTO}> <\text{IDENTIFIER}> \} \end{aligned}$	$\begin{aligned} <\text{后缀表达式}> ::= & <\text{初等表达式}> \{ \\ & <\text{左中括号}><\text{expression}><\text{右中括号}> \\ & <\text{左小括号}><\text{右小括号}> \\ & <\text{左小括号}><\text{实参表达式表}><\text{右小括号}> \\ & <\text{点号}> \text{ IDENTIFIER} \\ & <\text{箭头}> \text{ IDENTIFIER} \} \end{aligned}$
40	$\begin{aligned} <\text{argument_expression_list}> ::= & <\text{assignment_expression}> \{ \\ & <\text{TK_COMMA}><\text{assignment_expression}> \} \end{aligned}$	$<\text{实参表达式表}> ::= <\text{赋值表达式}> \{ <\text{逗号}><\text{赋值表达式}> \}$
41	$\begin{aligned} <\text{primary_expression}> ::= & <\text{IDENTIFIER}> \mid \\ & <\text{TK_CINT}> \mid <\text{TK_CSTR}> \mid <\text{TK_CCHAR}> \mid \\ & <\text{TK_OPENPA}> <\text{expression}> <\text{TK_CLOSEPA}> \end{aligned}$	$\begin{aligned} <\text{初等表达式}> ::= & <\text{标识符}> \mid <\text{整数常量}> \mid <\text{字符串常量}> \mid <\text{字符常量}> \mid \\ & (<\text{表达式}>) \end{aligned}$