

“十二五”普通高等教育本科国家级规划教材配套参考书

编译原理(第3版)习题精选与解析

陈意云　张　昱　编著

高等教育出版社

编译原理(第3版)习题精选与解析

本书特色：

- 强调对概念和方法的理解而不是强记，因此灵活运用所学知识的题目较多，极少出现按教材上的算法进行机械求解的题目。对其中有深度的题目，除了给出答案外，还有详细的分析，有利于学生理解和检查理解深度。
- 鼓励学生用所学的知识去分析和解决实际问题，因此从学习、使用编程语言和编译技术时碰到的实际问题中抽象出来的题目占据大半篇幅。这些题目对学生把握编程语言和理解编译技术很有帮助，能激发学生学习编译原理的兴趣。
- 题目涉及面广、灵活性强、重复性低、对配套教材的依附少，因此也适用于选择其他编译原理教材的教学和学生的自学。

ISBN 978-7-04-040579-8

9 787040 405798 >

定价 19.00 元

“十二五”普通高等教育本科国家级规划教材配套参考书

编译原理(第3版)习题精选与解析

Bianyi Yuanli(Di-san Ban)Xiti Jingxuan yu Jiexi

陈意云　张　昱　编著

高等教育出版社·北京

内容提要

本书是普通高等教育“十二五”国家级规划教材《编译原理》(第3版)的配套参考书。作者从主教材的习题和近年来所设计的各种试题中,精选出200多道题目,并将多年讲授这门课程的一些经验和体会写入本书,为便于结合教学来使用,本书各章的名称和主教材一致,并对难度较大的题目注上了“*”号。本书的习题涉及面广、灵活性强、重复性少,对学习编译原理课程很有帮助。

本书是本科生学习编译原理和技术的参考书,也可供计算机软件工程技术人员参考使用。

图书在版编目(CIP)数据

编译原理(第3版)习题精选与解析/陈意云,张昱编著. -- 北京:高等教育出版社,2014.9

ISBN 978-7-04-040579-8

Ⅰ.①编… Ⅱ.①陈… ②张… Ⅲ.①编译程序-程序设计-高等学校-教学参考资料 Ⅳ.①TP314

中国版本图书馆CIP数据核字(2014)第160011号

策划编辑 刘 艳 责任编辑 刘 艳 封面设计 于文燕 版式设计 范晓红
插图绘制 郝 林 责任校对 刁丽丽 责任印制 毛斯璐

出版发行 高等教育出版社
社 址 北京市西城区德外大街4号
邮政编码 100120
印 刷 北京玥实印刷有限公司
开 本 787mm×1092mm 1/16
印 张 13
字 数 280千字
购书热线 010-58581118
咨询电话 400-810-0598

网 址 http://www.hep.edu.cn
http://www.hep.com.cn
网上订购 http://www.landraco.com
http://www.landraco.com.cn

版 次 2014年9月第1版
印 次 2014年9月第1次印刷
定 价 19.00元

本书如有缺页、倒页、脱页等质量问题,请到所购图书销售部门联系调换
版权所有 侵权必究
物料号 40579-00

前　言

本书是普通高等教育“十二五”国家级规划教材《编译原理》(第3版)(高等教育出版社,2014年出版)一书的配套习题解答。

本书前版于2005年出版,时隔9年时间,完成改版更新工作。与上版相比,本版主要有以下几点改动。

(1)添加了30多道习题,新增习题的挑选原则与第一版的一致。

(2)对于因机器不同或编译器版本不同而可能造成答案不一样的情况,在解答中都增加了说明。例如对于C语言的语句printf("%d,%d,%d\n"),编译器一般都不报告错误,运行时输出3个从程序无法判断其值的整数。现在有些编译器会对printf调用的参数进行分析,对该语句会警告格式控制字符串与随后的实参表达式不匹配。

(3)从学习编程语言和编译技术时碰到的实际问题中抽象出来的习题大大增加,这些习题对把握编程语言和理解编译技术很有帮助。

习题解答对学生来说是一把双刃剑,使用得当则受益,反之则受害。怎样使用才得当则又因人而异,无论如何不能用它来不动脑筋地完成教师布置的书面作业。

限于时间和学识水平,书中难免存在不妥和错误之处。若发现本书有错误或对本书有建议,欢迎给作者发送电子邮件(yiyun@ustc.edu.cn,yuzhang@ustc.edu.cn)批评指正,作者将与出版社联系,在再次印刷或再版本书时改正错误和采纳合理的建议。

作者

于中国科学技术大学

2014年7月

目　录

第 1 章　引论 …… 1
第 2 章　词法分析 …… 3
第 3 章　语法分析 …… 21
第 4 章　语法制导的翻译 …… 50
第 5 章　类型检查 …… 72
第 6 章　运行时存储空间的组织和管理 …… 88
第 7 章　中间代码生成 …… 117
第 8 章　代码生成 …… 136
第 9 章　独立于机器的优化 …… 152
第 10 章　依赖于机器的优化 …… 172
第 11 章　编译系统和运行时系统 …… 179
第 12 章　面向对象语言的编译 …… 190
第 13 章　函数式语言的编译 …… 195

第1章 引 论

一、主要知识点

《编译原理》(第3版)(以下简称“主教材”)中第1章通过简要介绍编译器的各个逻辑阶段,对全书的内容做一个概述。由于此章中出现的大部分概念在以后各章会有详细介绍,因此不要求在学习此章时就都能理解这些概念。主要应掌握下面两点。

1. 基本概念:源语言、目标语言、翻译器、编译器、解释器。
2. 编译器的各个逻辑阶段,各阶段的主要功能。

二、习题精选与解析

1.1 解释器和编译器有什么区别?

答 编译器将高级语言源程序翻译成低级语言程序(经常是机器语言程序),然后由虚拟机(或者是硬件)执行编译的结果程序。在30多年前的BASIC语言阶段,解释器的功能是这样介绍的:它将高级语言的源程序翻译成一种中间语言程序,然后对中间语言程序进行解释执行。因为在那个年代,解释器的两个功能(编译和解释)是合在一个程序中的,这个程序被称为解释器。进入Java语言时代,解释器的上述两个功能分离成两个程序,前一个程序称编译器,它把Java语言的程序翻译成一种中间语言程序,这种中间语言叫做字节码;后一个程序称为解释器,它对字节码程序进行解释执行。

为了避免混淆,用编译执行和解释执行这两个术语来加以区别。一般来说,解释执行的效率低于编译执行的效率,究竟相差多少,和所用的中间语言关系很大。一个极端是,没有编译阶段,直接对源程序进行解释执行,这时的执行效率最低。另一个极端是,没有解释阶段,编译器将源程序直接翻译成机器语言程序,这时的执行效率最高。实际的解释执行介于这两个极端之间,选择一种合适的中间语言。

为什么说解释执行的效率低,以上面的第一种极端情况作解释。对于编译执行来说,对源程序的词法分析、语法分析和语义分析只要进行一次。而对于解释执行来说,每次执行到源程序的某个语句,都要对它进行一次词法分析、语法分析和语义分析,确定了这个语句的含义后,才能执行该含义指定的计算。显然,反复分析循环体降低了解释执行的效率。所以解释执行都要寻找一种适合于解释的中间语言,以减少反复分析需要的时间。反过来,如果源语言没有循环构造,

如历史上的作业控制语言,那么解释执行的效率最高,因为它省去了复杂的代码生成和代码优化等工作。

像 Java 语言这种解释方式的优点是,与机器和平台无关的中间语言使得中间语言程序能通过网络传到其他站点上运行,只要那里有一个中间语言的解释器就可以了。

1.2 编译器的逻辑阶段可以怎样分组?

答 编译器的阶段从逻辑上可以分成两组:一是由词法分析、语法分析和语义分析构成的编译器的分析部分,二是由中间代码生成、代码生成和代码优化构成的编译器的综合部分。

另一种分组方式是分成前端和后端两部分,前端是指编译器中完成从源程序到中间表示的那部分程序,后端是指编译器中完成从中间表示到目标语言程序的那部分程序。它和上面的分组方法是有区别的,例如,有些处理从逻辑上看属于综合部分,但它可能是放在前端完成的。一个具体事例是,从逻辑上看,变量的存储分配属于综合部分,但编译器的前端知道了变量的类型后,也就知道了该变量需要多少存储单元,因此通常是在前端完成变量的存储分配。

还有一种分组方式是按遍来分。一个编译过程可由一遍、两遍或多遍来完成。每一遍扫描的处理可完成一个阶段或多个阶段的工作。对于多遍的编译器,第一遍的输入是用户写的源程序,最后一遍的输出是目标语言程序,其余情况下则为上一遍的输出是下一遍的输入。

第2章 词法分析

一、主要知识点

主教材第2章主要应掌握下面一些内容。

1. 词法分析器的作用和接口,用高级语言编写词法分析器等,它们与词法分析器的实现有关。大部分教材上都有这方面的例子,这些问题比较适合作为实践题,因此本书没有安排这方面的习题,但不要认为进行这样的实践对编译原理的学习不重要。

2. 掌握以下概念,它们之间转换的技巧、方法或算法。

- 非形式描述的语言 ↔ 正规式(↔ 表示两个方向的转换都要掌握)
- 正规式 → NFA(非确定的有限自动机)
- 非形式描述的语言 ↔ NFA
- NFA → DFA(确定的有限自动机)
- DFA → 最简 DFA
- 非形式描述的语言 ↔ DFA(或最简 DFA)

作为习题来说,第2章的难点是为非形式描述的语言寻找一种形式描述(正规式、确定的或非确定的有限自动机),因为不存在这样的转换算法。

二、习题精选与解析

2.1 字母表 $\Sigma = \{ (,) \}$ 上的语言 $\{ (), (()()), ((())), ()()()()() \}$ 是不是正规语言?为什么?

答 该语言只包括有限个句子,它可以用正规式定义如下:

$$() \mid (()()) \mid ((())) \mid () () () () ()$$

所以该语言是正规语言。

分析 该语言的句子虽然都是配对括号串,但是它只有四个句子,因此它是正规语言。若该语言包括所有配对括号串,则它就不是正规语言了。

2.2 叙述由正规式 $0\,(0 \mid 1)^{*}\,0$ 和 $((\varepsilon \mid 0)\,1^{*})^{*}$ 描述的语言。

答 正规式 $0\,(0 \mid 1)^{*}\,0$ 表示字母表 $\{0, 1\}$ 上以0开始并以0结尾的长度大于1的所有串

的集合。正规式((ε | 0) 1*)* 表示字母表{0, 1}上所有串的集合。

分析 因为正规式(0 | 1)* 表示字母表{0, 1}上所有串的集合,显然,正规式 0 (0 | 1)* 0 只对串的前后两个字符做了限定,而中间的字符可以任意。

再分析正规式((ε | 0) 1*)* 表示的语言。可以看出正规式(ε | 0) 1* 描述的语言是集合{ε, 1, 11, 111, …, 0, 01, 011, 0111,…},它含长度为 1 的两个串 0 和 1。那么,(0 | 1)* 所描述的语言是((ε | 0) 1*)* 所描述的语言的子集。因为(0 | 1)* 表示字母表{0, 1}上所有串的集合,因此((ε | 0) 1*)* 所描述的语言不可能再有更多的句子,因而也是表示字母表{0, 1}上所有串的集合。

2.3 叙述正规式(00 | 11)*((01 | 10) (00 | 11)*(01 | 10) (00 | 11)*)* 描述的语言。

答 该正规式所描述的语言是:所有 0 和 1 的个数都是偶数的串的集合。另外,和该正规式等价的正规式有(00 | 11 | ((01 | 10) (00 | 11)*(01 | 10)))*。

分析 从 2.2 题已经看出,叙述正规式描述的语言并没有一种统一的办法,只能是通过对正规式的具体分析去总结。

该正规式的一个重要特点是,它是两个字符一组来考虑的。正规式(00 | 11)* 表示的串的长度是偶数,每两个字符一组的话,不是 00 就是 11。再看正规式(01 | 10) (00 | 11)*(01 | 10),它表示的串由 01 或 10 开始,中间有若干组 00 或 11,最后出现 01 或 10。这样的串,其 0 和 1 的个数仍然都是偶数,只不过第一组是 01 或 10 的话,那么一定还要有一组 01 或 10 才能保证它们的偶数性。显然,正规式(01 | 10) (00 | 11)*(01 | 10) (00 | 11)* 表示的串也仍然能保证 0 和 1 的个数都是偶数。这样,可以断定题目所给正规式表示的语言的每个句子都是 0 和 1 的个数是偶数的串。

反过来还需要考虑,任何 0 和 1 的个数都是偶数的串是否都在这个语言中。这实际上是问,每个这样的串,其结构是否都符合正规式(00 | 11)*((01 | 10) (00 | 11)*(01 | 10) (00 | 11)*)* 所做的刻画。可以这样叙述,0 和 1 的个数都是偶数的串,从左向右,每两个字符一组地考察,则

1. 由若干个(强调一下,可以是零个)00 或 11 开始(这由正规式(00 | 11)* 描述)。

2. 一旦出现一个 01 或 10,那么经过若干个 00 或 11 后,一定会出现一个 01 或 10。这第二个 01 或 10 的后面可能还有若干个 00 或 11,一直到串的结束,或者到再次出现 01 或 10 为止。如果再次出现 01 或 10 的话,就是重复出现这里所描述的结构(所以由((01 | 10) (00 | 11)* (01 | 10) (00 | 11)*)* 来描述)。

因此正规式(00 | 11)*((01 | 10) (00 | 11)*(01 | 10) (00 | 11)*)* 描述的是 0 和 1 的个数都是偶数的串。

可能会提一个问题,这样的串是否能用更简单的观点来看待,也就是该语言是否能用更简洁的正规式描述? 这是可能的,正规式如下,

(00 | 11 | ((01 | 10) (00 | 11)*(01 | 10)))*

它是基于这样的考虑,满足要求的最简单的串有三种形式(空串除外):

1. 00

2. 11

3. (01 | 10) (00 | 11)*(01 | 10)

它们任意多次的重复构成的串仍然满足要求。

2.4 一个语言的非形式定义是:字母表{0, 1}上所有不含子串 001 的 0 和 1 的串,写出定义该语言的正规式。

答 定义该语言的正规式是(1 | 01)* 0*。

分析 根据该语言的规定,在任意一个句子中,每个 1 的前面,紧挨着这个 1 的 0 至多只能有一个,所以有(1 | 01)*的结构。在最后一个 1 的后面可以有任意多个 0,因此该语言的正规式是(1 | 01)* 0*。

2.5 写出语言"0 的个数是偶数并且 1 的个数是奇数的所有 0 和 1 的串"的正规定义。

答 *even_0_even_1* → (00 | 11)*((01 | 10) (00 | 11)*(01 | 10) (00 | 11)*)*

even_0_odd_1 → 1 *even_0_even_1* | 0 (00 | 11)*(01 | 10) *even_0_even_1*

分析 有了 2.3 题的结果,这个问题应该容易解决。首先给 2.3 题的正规式起个名字:

even_0_even_1 → (00 | 11)*((01 | 10) (00 | 11)*(01 | 10) (00 | 11)*)*

对于 0 的个数是偶数并且 1 的个数是奇数的串,其第一个字符可能是 0 或 1。

1. 如果是 1,那么剩下的部分一定是 0 和 1 的个数都是偶数。

2. 如果是 0,那么经过若干个 00 或 11,一定会出现一个 01 或 10,才能保证 0 的个数是偶数,1 的个数是奇数。若串还没有结束,剩余部分一定是 0 和 1 的个数都是偶数。

因此,正确的正规定义是:

even_0_odd_1 →1 *even_0_even_1* | 0 (00 | 11)*(01 | 10) *even_0_even_1*

***2.6** C 语言的注释是以/*开始并且以*/结束的任意字符串,但它的任何前缀(本身除外)不以*/结尾。为 C 语言的注释写一个正规定义。

答 *other* → *a* | *b* | ⋯ 注:除了*以外,该右部的选择包括 C 的其他所有字符

other1 → *a* | *b* | ⋯ 注:除了*和/以外,该右部的选择包括 C 的其他所有字符

comment → /* *other*(* *)* *other1* *other*)* *** */

分析 对于这个问题,先构造接受该语言的有限自动机(见习题 2.11),然后利用其他教材介绍的从有限自动机得到正规式的算法,可能显得比较直观一些。在此更愿意通过分析直接得到结果,这样有助于提高大家的分析能力。

按照题目的规定,注释中间出现*时,它的后面不能直接跟/。因此,在从左向右检查注释中的字符序列时,主要关心字符*和/,所以才有 *other* 和 *other1* 的定义。在掠过可能有的若干个非

* 字符(即 $other^*$)后,碰到若干个 *,它的后继一定不能是/(即只能取 *other1*),然后就可以是任意字符了,直至又碰到 *,若它不是表示结束的 *,也不是一直持续到结束的第一个 *,那么应该重复刚才的过程。

由此可以写出上面的正规定义。

***2.7** 写出语言"所有相邻数字都不相同的非空数字串"的正规定义。

答

$no_0\text{-}8 \rightarrow 9$

$no_0\text{-}7 \rightarrow (8 \mid no_0\text{-}8\ 8)(no_0\text{-}8\ 8)^*(no_0\text{-}8 \mid \varepsilon) \mid no_0\text{-}8$

注:只含 8 和 9,且相邻数字都不相同的非空串

$no_0\text{-}6 \rightarrow (7 \mid no_0\text{-}7\ 7)(no_0\text{-}7\ 7)^*(no_0\text{-}7 \mid \varepsilon) \mid no_0\text{-}7$

注:只含 7、8 和 9,且相邻数字都不相同的非空串,下同

$no_0\text{-}5 \rightarrow (6 \mid no_0\text{-}6\ 6)(no_0\text{-}6\ 6)^*(no_0\text{-}6 \mid \varepsilon) \mid no_0\text{-}6$

$no_0\text{-}4 \rightarrow (5 \mid no_0\text{-}5\ 5)(no_0\text{-}5\ 5)^*(no_0\text{-}5 \mid \varepsilon) \mid no_0\text{-}5$

$no_0\text{-}3 \rightarrow (4 \mid no_0\text{-}4\ 4)(no_0\text{-}4\ 4)^*(no_0\text{-}4 \mid \varepsilon) \mid no_0\text{-}4$

$no_0\text{-}2 \rightarrow (3 \mid no_0\text{-}3\ 3)(no_0\text{-}3\ 3)^*(no_0\text{-}3 \mid \varepsilon) \mid no_0\text{-}3$

$no_0\text{-}1 \rightarrow (2 \mid no_0\text{-}2\ 2)(no_0\text{-}2\ 2)^*(no_0\text{-}2 \mid \varepsilon) \mid no_0\text{-}2$

$no_0 \rightarrow (1 \mid no_0\text{-}1\ 1)(no_0\text{-}1\ 1)^*(no_0\text{-}1 \mid \varepsilon) \mid no_0\text{-}1$

$answer \rightarrow (0 \mid no_0\ 0)(no_0\ 0)^*(no_0 \mid \varepsilon) \mid no_0$

分析 刚拿到这个问题,一定不知从哪儿下手。其实和前面一样,关键是用一种合适的观点来看待这种句子结构。可以按这样的观点:每个这样的句子由若干个 0 把它分成若干段,如

1230313571067803590123

可以看成

123, 0, 313571, 0, 678, 0, 359, 0, 123

由 0 隔开的每一段,如 313571,它不含 0,并且又可以看成由若干个 1 把它分成多段。如此下去,就能找到该语言的正规定义。

按这个思路,上面的正规定义应该逆序看。

$answer \rightarrow (0 \mid no_0\ 0)(no_0\ 0)^*(no_0 \mid \varepsilon) \mid no_0$

表示一个句子由若干个 0 分成若干段,特殊情况是整个句子不含 0。在这个正规定义中,所引用的 *no_0* 表示不含 0 的串,它的定义和这个定义的形式一样,因为串的形式是一样的,只不过没有数字 0。所以有

$no_0 \rightarrow (1 \mid no_0\text{-}1\ 1)(no_0\text{-}1\ 1)^*(no_0\text{-}1 \mid \varepsilon) \mid no_0\text{-}1$

其中,*no_0-1* 表示不含 0 和 1 的串。以此类推,最后 *no_0-8* 是表示不含 0, …, 8 的、没有重复数字的串,它只可能是单个 9。

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

答 见图 2.1。

分析　对于这样的问题，不要急于去尝试画 DFA，先把问题分析一下，这里要接受的是 0 和 1 的个数都是偶数的串。一个自然数不是偶数就是奇数，因此，对于任意一个 0 和 1 的串，不论其 0 和 1 的个数有多少，总归属于下面四种情况之一：

0：0 和 1 的个数都是偶数；

1：0 的个数是偶数，1 的个数是奇数；

2：0 的个数是奇数，1 的个数是偶数；

3：0 和 1 的个数都是奇数。

并且不管一个串处于上面哪一种情况，给它再添加一个 0 或 1 后，它总是处于上面另一种情况。由此分析可以知道，DFA 只需四个状态就够了，并且状态转换图也可以很容易地画出来。答案中的四个状态对应于这里的四种情况。空串是属于 0 和 1 的个数都是偶数的情况，因此状态 0 是开始状态。因为是接受 0 和 1 的个数都是偶数的串，因此状态 0 也是接受状态。

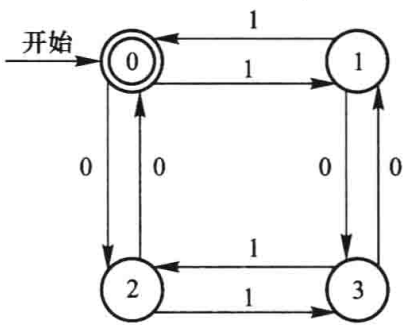


图 2.1　接受 0 和 1 的个数都是偶数的 DFA

2.9　构造一个 DFA，它接受 $\Sigma = \{0, 1\}$ 上能被 5 整除的二进制数。为使问题稍微简单一些，00101 这样的形式也被看成是合法的二进制数，即允许前面有无效的 0。

答　见图 2.2。

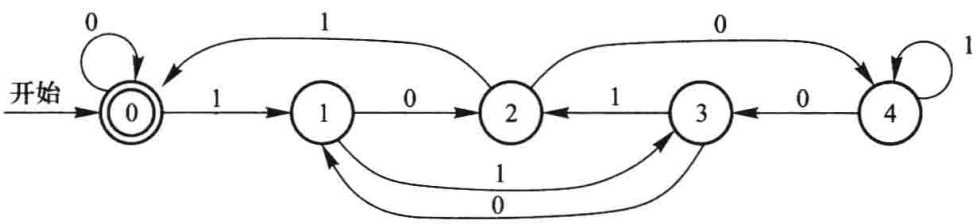


图 2.2　接受能被 5 整除的二进制数的 DFA

分析　由上题可以知道，构造 DFA 之前，首先搞清楚问题的状态空间。即想明白应该有多少个状态，状态之间的转换条件，以及针对该问题的开始状态和接受状态。

对于本题目，任意一个二进制数除以 5 时，只有余数为 0（即整除），1，2，3 和 4 这五种情况。图 2.2 中的五个状态也是这样命名的。一个二进制数的后面添上一个 0 意味着其值变成原来的两倍，而后面添上一个 1 意味着其值变成原来的两倍再加 1。不管是哪一种情况，都很容易从原来的余数决定值变化后的余数。这样，可以很快得出所有的状态转换。例如考虑状态 4，任何一个余 4 的数，两倍后一定余 3，两倍再加 1 后一定还是余 4。所以，状态 4 遇 0 转换到状态 3，遇 1 转换到本身。显然，状态 0 既是开始状态又是接受状态。

需要注意的是，考虑状态空间时，还要检查所取的是否为最简情况（即状态数极小）。例如，对于本题目，假如从这样的观点出发，每个二进制数都可以转换成一个十进制数。十进制数的末位有 0~9 这 10 种情况，其中末位为 0 和 5 是能被 5 整除的情况。这样很可能会构造 10 个状态的 DFA，接受状态有两个。这也是一种解，但它不是最简的 DFA。

2.10 构造一个 DFA，它接受 $\Sigma = \{0, 1\}$ 上所有大于 5 的二进制整数。

答 见图 2.3。为直观起见，状态名直接体现到达该状态时，所读过的字符串可能构成的十进制整数。整数 5 的二进制表示是 101。

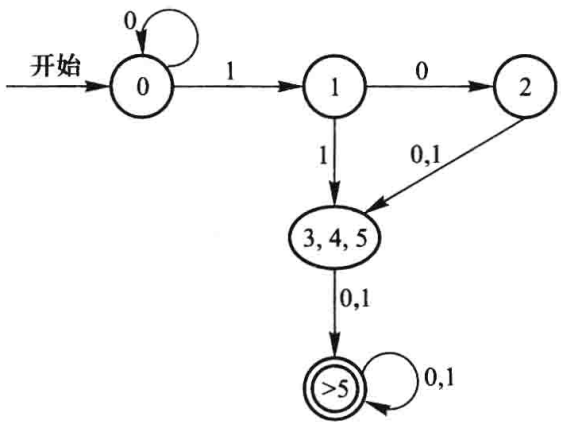


图 2.3 接受大于 5 的二进制整数的 DFA

分析 因为 3,4 和 5(二进制分别是 11,100 和 101)的 2 倍都大于 5，因此读进 3,4 和 5 后应该到达同一个状态。该状态表示，再读进字符的话，则值一定大于 5 了。

2.11 由/*开始，并由*/结束的串构成 C 语言的注释，注释中间没有*/。画出接受这种注释的 DFA 的状态转换图。

答 见图 2.4。标记为 others 的边是指字符集中未被从同一状态出发的别的边指定的任意其他字符。

分析 这个 DFA 的状态数及含义并不难确定，见下面的五个状态说明。

状态 1：注释开始状态。

状态 2：进入注释体前的中间状态。

状态 3：表明目前正在注释体中的状态。

状态 4：离开注释前的中间状态。

状态 5：注释结束状态，即接受状态。

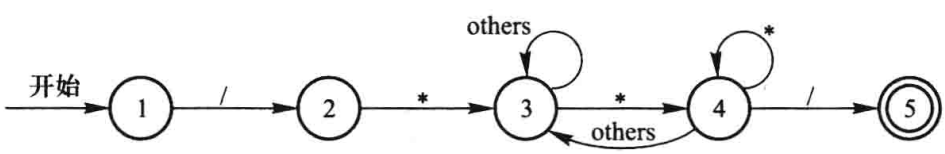


图 2.4 接受注释的 DFA

在这个 DFA 中，最容易忽略的是状态 4 到本身的*转换。这条边的含义是：在离开注释前的中间状态，若下一个字符是*，那么把刚才读过的*看成是注释中的一个字符，而把当前这个字符看成可能是表明注释结束的那个*。若没有这条边，那么像

/ * * * * This is a comment * * * * /

这样的注释就被拒绝。

另外，上面的状态转换图并不完整。例如，对于状态 1，没有指明遇到其他字符怎么办。要把状态转换图画完整，还需引入一个死状态 6，进入这个状态就再也出不去了。因为它不是接受状态，因此进入这个状态的串肯定不被接受。完整的状态转换图见图 2.5，其中 all 表示任意字符。在能够说清问题时，通常省略死状态和所有到它的边。

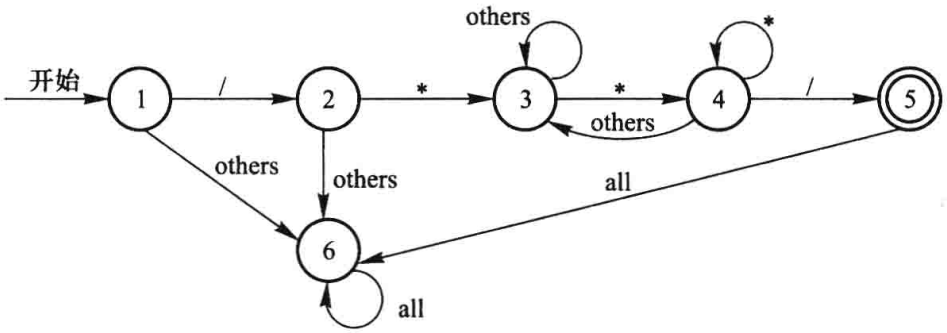


图 2.5 接受注释的完整 DFA

2.12 某操作系统下合法的文件名为

device : name . extension

其中第一部分(device :)和第三部分(. extension)都可省略，若 device，name 和 extension 都是字母串，长度不限，但至少为 1，画出识别这种文件名的 DFA。

答 见图 2.6，图中的标记 *d* 表示任意字母。

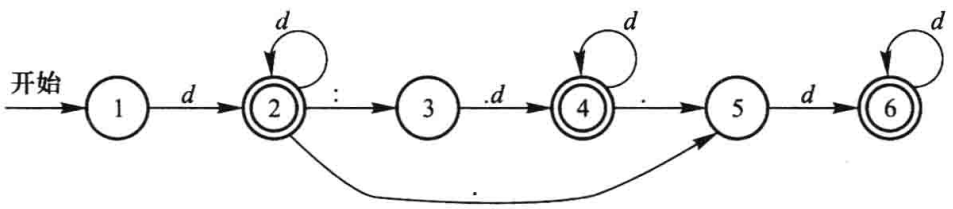


图 2.6 接受文件名的 DFA

分析 这个 DFA 和一些教材上接受无符号数的 DFA 有类似的地方。首先考虑 device: 和 . extension 全都出现的情况。这时的 DFA 比较容易构造，见图 2.7。

然后考虑省略情况。因为 . extension 可省略，因此把状态 4 也作为接受状态。因为 name 和 device 一样，都是字母序列，因此当 device: 省略时，把到状态 2 为止得到的字母序列看成是

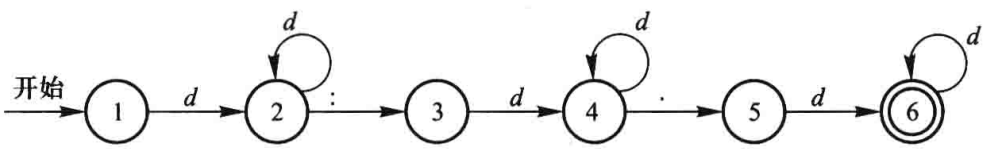


图 2.7 文件名的三部分都出现的 DFA

name,所以从状态 2 画一条转换边到状态 5,标记为“.”(如果构成 name 和 device 的字符完全不一样,那么可以画一条从状态 1 到状态 4 的边,其标记同状态 3 到状态 4 的标记一样)。由于 device:和 . extension 都可省略,因此把状态 2 也作为接受状态。

2.13 为正规式$(a \mid b)^* a\ (a \mid b)\ (a \mid b)$构造 NFA。

答 该 NFA 的状态转换图见图 2.8。

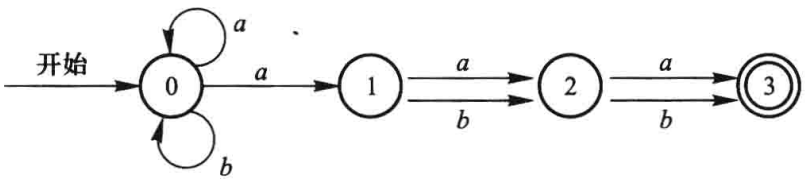


图 2.8 接受正规式$(a|b)^* a(a|b)(a|b)$的 NFA

分析 各种教材在介绍有限状态自动机和正规式的等价时,都给出了从正规式构造等价的 NFA 的算法。各种构造算法虽然不一样,但有一个共同的特点,都或多或少地引入了 ε 转换,使状态转换图变得复杂。尤其是,如果题目还要求画出 DFA 时,那么状态数的增多使得手工完成从 NFA 到 DFA 的确定化过程变得更容易出错。因此,既要会用教材上的算法构造 NFA,也要会手工构造更简洁一些的 NFA,尽量避免在 NFA 中出现 ε 转换。这在大多数情况下是可以做到的,本题就是一个例证。

2.14 用状态转换图表示接受正规式$(a \mid b)^* aa$ 的 DFA。

答 状态转换图见图 2.9。

分析 对于$(a \mid b)^* aa$ 这样的正规式,构造 NFA 是容易的,但是要构造 DFA,需要多少个状态和每个状态的含义就不是立即能看清楚的。在此,不走先构造 NFA 然后把它确定化的道路,而是直接构造 DFA。大家既要会按教材上的算法从 NFA 的确定化得到 DFA,也要会直接构造 DFA,这一点比较重要。通过本题和下面两题,可以看出直接构造 DFA 也并不困难。

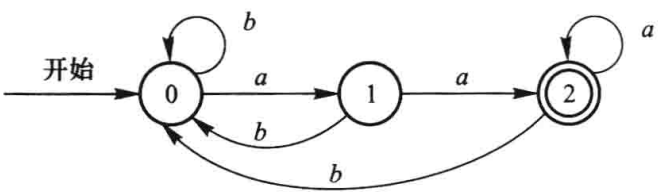


图 2.9 接受正规式$(a|b)^* aa$ 的 DFA

该正规式表示的语言是,字母表 $\Sigma = \{a, b\}$ 上最后两个字符都是 a 的串的集合。抓住这个特点,首先画出构造过程中的第一步,见图 2.10。它是接受最简单的句子 aa 的 DFA。

然后,因为在第一个 a 前可以有若干个 b,因此状态 0 有到自身的 b 转换。在最后两个字符都是 a 的串的末尾添加若干个 a 的话,它能够保持串的这个性质,因此状态 2 有到自身的 a 转换。这样就得到图 2.11。

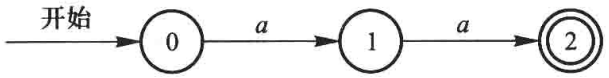


图 2.10　构造过程中的第一步

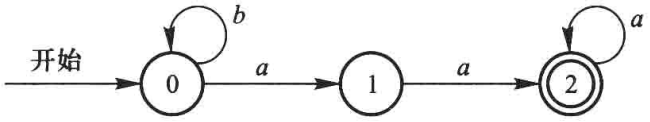


图 2.11　构造过程中的第二步

最后，在状态 1 和状态 2 碰到 b 时，不管前面刚连续读过多少个 a，它们都不可能作为句子结尾的那两个字符 a，因此状态 1 和状态 2 的 b 转换都回到状态 0。

所有状态的 a 转换和 b 转换都已给出，这就得到了最后结果。

现在可以看出，状态 0、1 和 2 的含义分别是刚连续读过 0 个、1 个和不少于 2 个 a。

2.15　画出接受$(1 \mid 01)^* 0^*$所描述语言的最简 DFA 的状态转换图。

答　由 2.4 题知道，该正规式表示所有不含子串 001 的 0 和 1 的串。接受该语言的最简 DFA 的状态转换图见图 2.12。

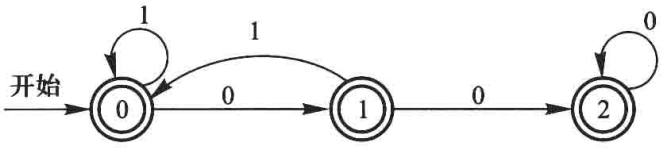


图 2.12　接受正规式$(1101)^* 0^*$的 DFA

分析　根据上面的非形式说明，在此关心的是，出现两个连续的 0 后不能再出现 1。根据上一题的经验，这三个状态的含义是：

状态 0：上一步读过的字符不是 0；

状态 1：连续读过一个 0；

状态 2：已经连续读过了不少于两个 0。

显然，所有面临 1 的转换都是到状态 0，但状态 2 不能有这样的转换。

直观上可以判断这是最简 DFA，因为必须有这样三个状态来分别表示所关心的连续读 0 的三种情况。

2.16　用子集构造法给出由图 2.13 的 NFA 得到的 DFA 的转换表。

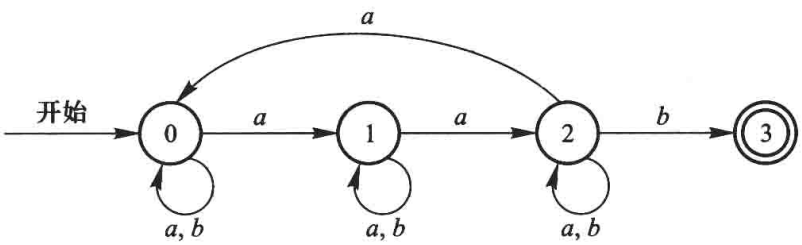


图 2.13　一个不确定的有限自动机

答 DFA 的转换表见表 2.1。DFA 各状态所代表的 NFA 的状态子集分别是：$A=\{0\}$，$B=\{0, 1\}$，$C=\{0, 1, 2\}$，$D=\{0, 1, 2, 3\}$。

表 2.1 DFA 的转换表

状 态	输 入 符 号	
	a	b
A	B	A
B	C	B
C	C	D
D	C	D

分析 解答这样的题目没有什么技巧，老老实实按照子集构造法来计算，尤其是当不能简洁地描述该 NFA 所接收的语言时。

***2.17** 用状态转换图表示接受正规式$(a \mid b)^* a\ (a \mid b)\ (a \mid b)$的 DFA。

答 状态转换图见图 2.14。

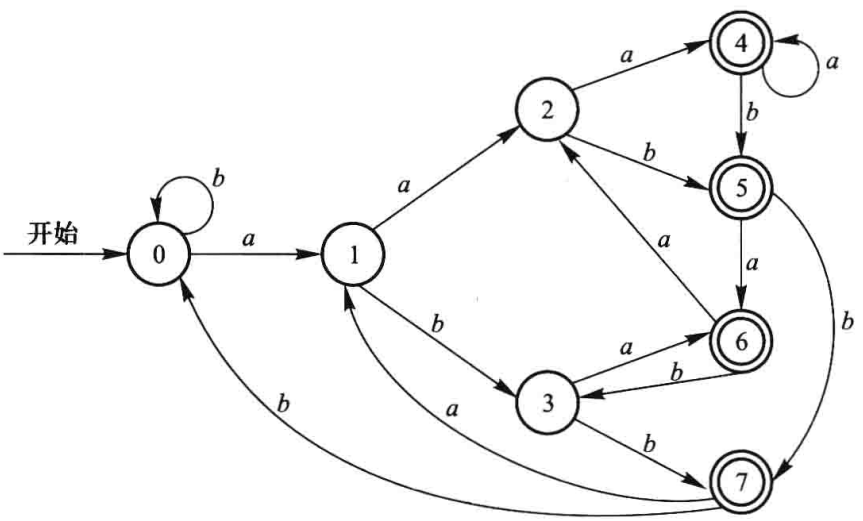


图 2.14 接收$(a|b)^*a(a|b)(a|b)$的 DFA

分析 该正规式表示的语言是，字母表 $\Sigma=\{a, b\}$ 上倒数第三个字符是 a 的串的集合。根据上两题的经验，首先画出图 2.15。因为最后两个字符任意，因此有这样的分支，并有四个接受状态。

现在考虑这四个接受状态上的转换。

1. 状态 4：该状态表示最后三个字符是 aaa，若再添加一个 a，最后三个字符仍是 aaa，因此状态 4 的 a 转换到本身。若添加的是 b，那么最后三个字符是 aab，而状态 5 表示最后三个字符是 aab，因此状态 4 的 b 转换到状态 5。

2. 状态 5：该状态表示最后三个字符是 aab，若再添加一个 a，最后三个字符成了 aba，而状态 6 表示最后三个字符是 aba，因此状态 5 的 a 转换到状态 6。若添加的是 b，那么最后三个字符是

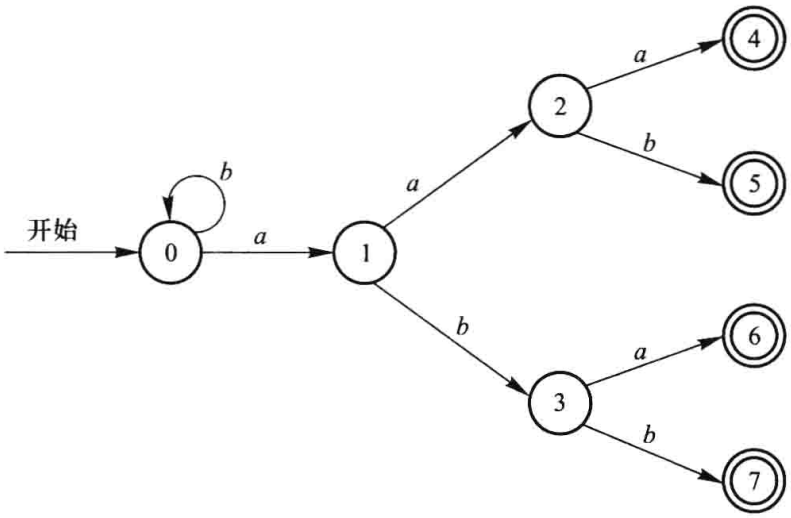


图 2.15　构造过程的第一步

abb，而状态 7 表示最后三个字符是 abb，因此状态 5 的 b 转换到状态 7。

3. 状态 6：该状态表示最后三个字符是 aba，若再添加一个 a，最后三个字符成了 baa，其由 a 开始的后缀是 aa，因此状态 6 的 a 转换到状态 2（因为从状态 0 出发经 aa 是到状态 2）。若添加的是 b，那么最后三个字符是 bab，其由 a 开始的后缀是 ab，因此状态 6 的 b 转换到状态 3。

4. 状态 7：该状态表示最后三个字符是 abb，若再添加一个 a，最后三个字符成了 bba，其由 a 开始的后缀是 a，因此状态 7 的 a 转换到状态 1。若添加的是 b，那么最后三个字符是 bbb，不存在由 a 开始的后缀，因此状态 7 的 b 转换到状态 0。

这样，所有状态的 a 转换和 b 转换都已给出，最后结果自然也就得到了。

2.18　将 2.13 题得到的 NFA 变换成 DFA。

答　所求的 DFA 就是 2.17 题的结果。

分析　之所以选这个题目，是为了比较一下，从正规式到 NFA，再把 NFA 确定化，这样得的结果同 2.17 题直接构造 DFA 的结果是否一样。

按照教材上的子集构造法，作为结果的 DFA 并不难得到。另外由于没有 ε 转换，构造过程相对简单了很多。

NFA 的开始状态是 0，因此首先从 NFA 的状态集合{0}开始，它是 DFA 的开始状态，命名为状态 0′。它的 a 转换和 b 转换所得到的 NFA 的状态集合见下面第一行。根据子集构造法所得的 DFA 的所有状态和它们的转换函数都列在下面：

状态 0′：{0}	$move(\{0\}, a) = \{0, 1\}$	$move(\{0\}, b) = \{0\}$
状态 1′：{0, 1}	$move(\{0, 1\}, a) = \{0, 1, 2\}$	$move(\{0, 1\}, b) = \{0, 2\}$
状态 2′：{0, 1, 2}	$move(\{0, 1, 2\}, a) = \{0, 1, 2, 3\}$	$move(\{0, 1, 2\}, b) = \{0, 2, 3\}$
状态 3′：{0, 2}	$move(\{0, 2\}, a) = \{0, 1, 3\}$	$move(\{0, 2\}, b) = \{0, 3\}$
状态 4′：{0, 1, 2, 3}	$move(\{0, 1, 2, 3\}, a) = \{0, 1, 2, 3\}$	$move(\{0, 1, 2, 3\}, b) = \{0, 2, 3\}$

状态 5′:{0, 2, 3}　$move(\{0, 2, 3\}, a) = \{0, 1, 3\}$　$move(\{0, 2, 3\}, b) = \{0, 3\}$

状态 6′:{0, 1, 3}　$move(\{0, 1, 3\}, a) = \{0, 1, 2\}$　$move(\{0, 1, 3\}, b) = \{0, 2\}$

状态 7′:{0, 3}　$move(\{0, 3\}, a) = \{0, 1\}$　$move(\{0, 3\}, b) = \{0\}$

状态 4′, 5′, 6′和 7′中都含原 NFA 的接受状态 3,因此它们都是 DFA 的接受状态。不难看出,所得的 DFA 和 2.17 题的结果是同构的,仅状态名不一样。

***2.19**　将图 2.16 的 DFA 极小化。

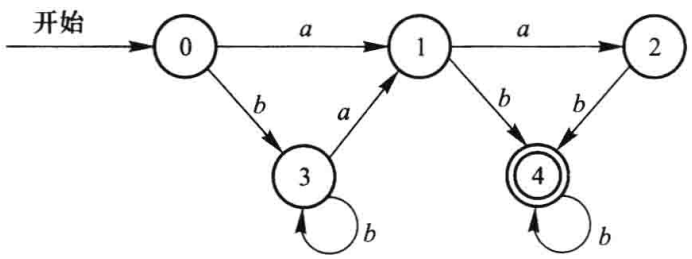


图 2.16　一个 DFA

答　最简 DFA 见图 2.17。

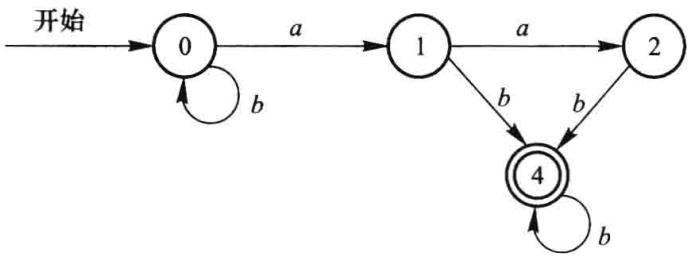


图 2.17　最简 DFA

分析　本题要注意的是,在使用极小化算法前,一定要检查一下,看状态转换函数是否为全函数,即看每个状态对每个输入符号是否都有转换。若不是全函数,需加入死状态,然后再应用极小化算法。有些教材上没有强调这一点,有的习题解答参考书上的示例甚至忽略了这一点,本题将说明,这一点是重要的。本题加入死状态 5 后的状态转换图见图 2.18。

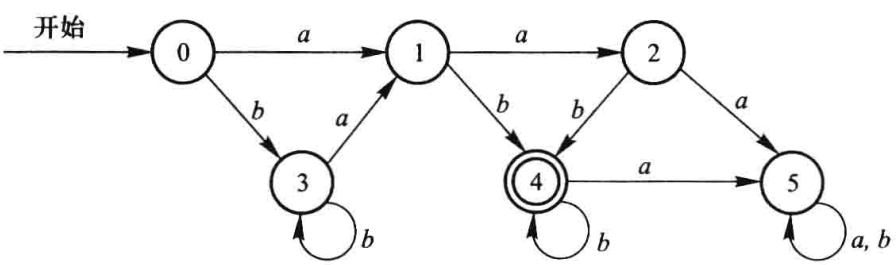


图 2.18　加入死状态后的 DFA

使用极小化算法,先把状态集分成非接受状态集{0, 1, 2, 3, 5}和接受状态集{4}这两个子集。

1. 集合{4}不能再分解,再看集合{0, 1, 2, 3, 5}。

$move(\{0, 1, 2, 3, 5\}, a) = \{1, 2, 5\}$　　$move(\{0, 1, 2, 3, 5\}, b) = \{3, 4, 5\}$

由于 b 转换的结果$\{3, 4, 5\}$不是最初划分的某个集合的子集,因此$\{0, 1, 2, 3, 5\}$需要再分,由于状态 1 和状态 2 的 b 转换都到状态 4。因此状态集合的进一步划分是:

$\{1, 2\}$,$\{0, 3, 5\}$和$\{4\}$

2. 由于

$move(\{1, 2\}, a) = \{2, 5\}$　　$move(\{1, 2\}, b) = \{4\}$

$move(\{0, 3, 5\}, a) = \{1, 5\}$　　$move(\{0, 3, 5\}, b) = \{3, 5\}$

显然$\{1, 2\}$和$\{0, 3, 5\}$需要再分,分别分成:

$\{1\}$和$\{2\}$以及$\{0, 3\}$和$\{5\}$

3. 由于

$move(\{0, 3\}, a) = \{1\}$　　$move(\{0, 3\}, b) = \{3\}$

因此不需要再分。

这样,状态 0 和状态 3 合并成一个状态,取 0 为代表,再删去死状态 5,就得到结果。

如果不加入死状态,极小化算法会是什么结果? 最初的划分是$\{0, 1, 2, 3\}$和$\{4\}$。

1. 状态集合的进一步划分是:

$\{1, 2\}$,$\{0, 3\}$和$\{4\}$

2. 忽略了死状态的影响,会认为它们都不需要再分,从而得到如图 2.19 所示的 DFA。

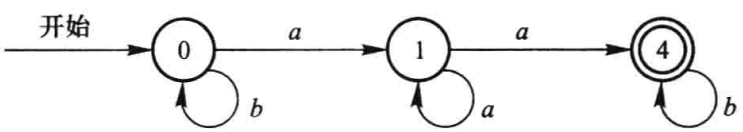


图 2.19　一个不正确的结果

显然,它和原来的 DFA 不等价。

2.20　将习题 2.17 结果的 DFA 极小化。

答　化简的结果仍是习题 2.17 结果的 DFA,即该 DFA 已是最简 DFA。这说明手工构造最简 DFA 是完全可能的。

分析　简要说明执行过程。

初始时将状态集分成两组:$\{0, 1, 2, 3\}$和$\{4, 5, 6, 7\}$。

1. 由于

$move(\{0, 1, 2, 3\}, a) = \{1, 2, 4, 6\}$　　$move(\{0, 1, 2, 3\}, b) = \{0, 3, 5, 7\}$

$move(\{4, 5, 6, 7\}, a) = \{1, 2, 4, 6\}$　　$move(\{4, 5, 6, 7\}, b) = \{0, 3, 5, 7\}$

因此进一步分成$\{0, 1\}$、$\{2, 3\}$、$\{4, 5\}$和$\{6, 7\}$四组。

2. 由于

$move(\{0, 1\}, a) = \{1, 2\}$　　$move(\{0, 1\}, b) = \{0, 3\}$

因此{0, 1}还要进一步分,其他几组也是这样。因此最终分成了每个集合中都只有一个状态。

3. 所以,原来的 DFA 已经是最简的 DFA 了。

2.21 写出字母表 $\Sigma = \{a, b\}$ 上语言 $L = \{w \mid w$ 中 a 的个数是偶数$\}$ 的正规式,并画出接受该语言的最简 DFA。

答 语言 L 的正规式是 $(a\ b^*a \mid b)^*$ 或 $b^*(a\ b^*a\ b^*)^*$,接受 L 的最简 DFA 见图 2.20。

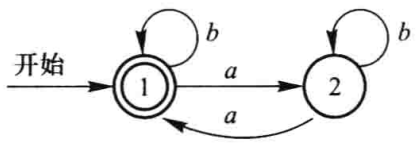


图 2.20 接受 $(a\ b^*a \mid b)^*$ 的 DFA

分析 第一个正规式的观点是这样的:除了空串外,b 和 $a\ b^*a$ 是结构最简单的句子,这样的句子经任意次连接后得到的仍然是句子。任何句子都可以按这种观点进行分解。

第二个正规式是基于这样的观点:第一个 a 之前可能有若干个 b,每个 a 后面可能跟若干个 b,因为要保证 a 的个数是偶数,所以要以两个 a 为单位来考虑结构的重复出现。

2.22 写出字母表 $\Sigma = \{a, b\}$ 上语言 $L = \{w \mid w$ 的最后两个字母是 aa 或 $bb\}$ 的正规式,并画出接受该语言的最简 DFA。

答 语言 L 的正规式是 $(a \mid b)^*(aa \mid bb)$,接受 L 的最简 DFA 见图 2.21。

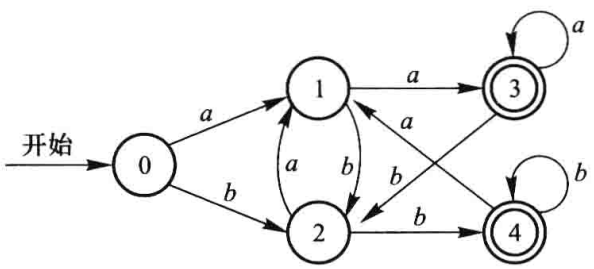


图 2.21 接受 $(a \mid b)^*(aa \mid bb)$ 的 DFA

分析 这个正规式应该很容易写出,参考 2.17 题应该能够画出这个 DFA。

2.23 用正规式表示字母表 $\{a, b\}$ 上 a 不会相邻的所有句子的集合,并给出接受该语言的最简 DFA。

答 该语言的正规式是 $b^*(abb^*)^*(a \mid \varepsilon)$。接受该语言的最简 DFA 见图 2.22。

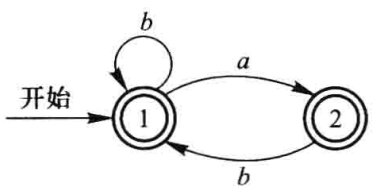


图 2.22 接受 $b^*(abb^*)^*(a \mid \varepsilon)$ 的 DFA

分析 该语言也可以用另一种方式来表达:每个 a 之后一定有 b,除非它是句子的最后一个字符。直觉上可以判断这是最简 DFA,因为状态 1 和 2 不可以合并。

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

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

其中,x^{R}表示 x 的逆序。

答 若能定义出接受语言 L'的一个 NFA,那么 L'就是正规语言。因为 L 是正规语言,那么一定存在接受 L 的 DFA M,现在就基于 M 来构造接受 L'的 NFA M',并且基于 M 的状态转换图来叙述。

1. 将状态转换图上的所有边改变方向,边上的标记不变。

2. 将原来的开始状态改成接受状态。

3. 增添一个状态作为开始状态,从它有 ε 转换到原来的每一个接受状态,并把原来的这些接受状态都改成普通的状态。

所得到的新状态转换图就是 M'的状态转换图。

分析 简单说,判断一个串是否为 L'的句子,只要在原来的状态转换图上逆行就可以了。由于 M 可能有多个接受状态,而不管是 DFA 还是 NFA 都只有唯一的开始状态,因此需要上面的第 3 步。

注意,经过上面的改造,得到的是一个 NFA 的状态转换图。除了上面第 3 步引入 ε 转换外,第 1 步也可能引入不确定性。

本题介绍的证明语言正规性的方法在下一题表现得更加充分,叙述也比本题严格。

***2.25** 若 L 是正规语言,证明下面 $\frac{1}{2}L$ 语言也是正规语言。$\frac{1}{2}L$ 语言的定义是

$$\frac{1}{2}L = \{x \mid \exists y.\, xy \in L \ \&\ |x| = |y|\}$$

答 因为 L 是正规语言,那么存在一个 DFA $M = (S, \Sigma, \delta, s, F)$,它接受语言 L。接受语言 $\frac{1}{2}L$ 的 DFA $M' = (S', \Sigma, \delta', s', F')$可如下构造。

1. S'的每个状态是一个二元组$\langle s_1, S_1\rangle$,其中 $s_1 \in S$,$S_1 \subseteq S$。

2. $\delta'(\langle s_1, S_1\rangle, a) = \langle s_2, S_2\rangle$,其中

$$s_2 = \delta(s_1, a)$$

$$S_2 = \{s_2 \mid \exists s_1 \in S_1.\exists b \in \Sigma.\, \delta(s_2, b) = s_1\}$$

3. $s' = \langle s, F\rangle$

4. $F' = \{\langle s_1, S_1\rangle \mid s_1 \in S_1\}$

可以证明这是接受语言 $\frac{1}{2}L$ 的 DFA,因此 $\frac{1}{2}L$ 是正规语言(证明比较复杂,在此略去)。

分析 这个题目超出了编译课程的范围。但是如果理解了这儿的解答，那就掌握了证明正规语言的一种很重要的技巧。下面说明为什么这样定义 DFA M'。

长度为 n 的串 w，若它是语言 L 的某个句子的前缀，那么从 M 的开始状态 s，经 n 步转换，到达某个状态 s_n。该串是否属于$\frac{1}{2}L$，取决于是否存在从 s_n 开始到某个接受状态的路径，其长度也为 n。

怎样判断是否存在这样的到某个接受状态的路径呢？可以在 M 的状态转换图上按下面的步骤执行。

1. 首先找出从任意接受状态逆向任意走一步能到达的所有状态，把这个状态集合称做 R_1。
2. 再找出从 R_1 的任意状态逆向任意走一步能到达的所有状态，把这个状态集合称做 R_2。
3. 这样逆向操作 n 步后，得到状态集合 R_n。
4. 若 s_n 在 R_n 中，那么串 w 属于$\frac{1}{2}L$，否则不是。

可以看出，上面定义的 M' 是在 M 上同时跟踪从开始状态出发的正向状态转换和从接受状态集合出发的逆向搜索一步能到达的所有状态。再看一下 M' 的定义。

1. S' 的每个状态之所以是一个二元组 $\langle s_1, S_1\rangle$，是因为需要用 s_1 来表示正向的状态转换，用 S_1 来表示逆向搜索。

2. 再看状态转换的定义 $\delta'(\langle s_1, S_1\rangle, a) = \langle s_2, S_2\rangle$，其中 $s_2 = \delta(s_1, a)$ 表示了 M 上正向一步转换到达的状态，$S_2 = \{s_2 \mid \exists s_1 \in S_1 . \exists b \in \Sigma . \delta(s_2, b) = s_1\}$ 表示了 M 上逆向一步搜索到达的状态集合。

3. 开始状态 $s' = \langle s, F\rangle$ 表示在 M 上的跟踪是从开始状态和接受状态集合这两端启动。

4. 接受状态 $F' = \{\langle s_1, S_1\rangle \mid s_1 \in S_1\}$ 表达的意思是，若在 M 上从开始状态经 n 步转换到 s_1，那么一定存在长度为 n 的从 s_1 到某个接受状态的路径。

下面以 2.8 题的正规语言为例。不难理解，该语言的$\frac{1}{2}$语言应该是字母表 $\Sigma = \{0, 1\}$ 上的所有串。可以用上面的方法来构造接受它的 DFA。该 DFA 的开始状态 $s_0 = \langle 0, \{0\}\rangle$。

状态 s_0：$\langle 0, \{0\}\rangle$　　$\delta(s_0, 0) = \langle 2, \{1, 2\}\rangle$　　$\delta(s_0, 1) = \langle 1, \{1, 2\}\rangle$

状态 s_1：$\langle 2, \{1, 2\}\rangle$　　$\delta(s_1, 0) = \langle 0, \{0, 3\}\rangle$　　$\delta(s_1, 1) = \langle 3, \{0, 3\}\rangle$

状态 s_2：$\langle 1, \{1, 2\}\rangle$　　$\delta(s_2, 0) = \langle 3, \{0, 3\}\rangle$　　$\delta(s_2, 1) = \langle 0, \{0, 3\}\rangle$

状态 s_3：$\langle 0, \{0, 3\}\rangle$　　$\delta(s_3, 0) = \langle 2, \{1, 2\}\rangle$　　$\delta(s_3, 1) = \langle 1, \{1, 2\}\rangle$

状态 s_4：$\langle 3, \{0, 3\}\rangle$　　$\delta(s_4, 0) = \langle 1, \{1, 2\}\rangle$　　$\delta(s_4, 1) = \langle 2, \{1, 2\}\rangle$

从 s_0 到 s_4 的这五个状态，每个状态的第一元都在第二元的集合中，因此这五个状态都是接受状态。这个 DFA 的图形表示见图 2.23（图中用状态 0，1，2，3 和 4 代替了这里对应的状态）。显然，它能接受字母表 $\Sigma = \{0, 1\}$ 上所有的串，化简这个 DFA 可以得到只有一个状态的 DFA。

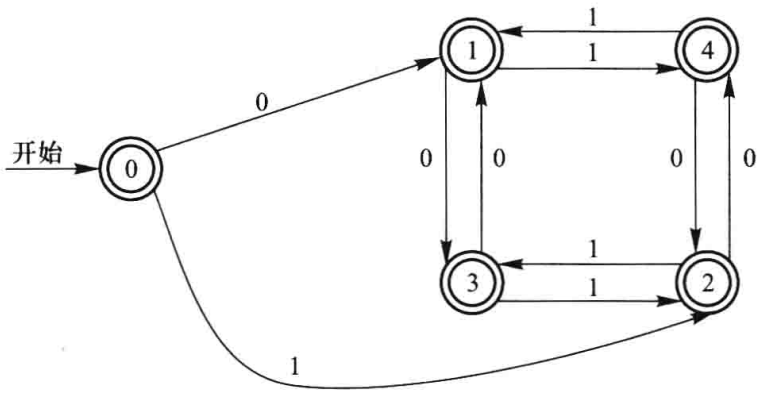


图 2.23　接受 $\Sigma=\{0,1\}$ 上所有串的一个 DFA

2.26　保留字、关键字和标准标识符之间的区别是什么？

答　保留字是语言预先确定了含义的词法单元，程序员不能对这样的词法单元重新声明它的含义。如 Pascal 语言的 type 和 procedure 等。词法分析器读到一个符合标识符拼写的词法单元时，都要和保留字进行比较，以确定该词法单元是标识符还是保留字。

很多语言的关键字是保留的，因此和上面的保留字没有区别，如 C 语言和 Java 语言。但是 FORTRAN 语言的关键字不是保留的，如 IF，当它作为语句的第一个词法单元时，很可能是关键字，但也不排除是用户定义的标识符。这给词法分析带来很大困难，因为识别词法单元和该词法单元所处的上下文有关。现在的语言都不会像 FORTRAN 这样来定义关键字了。

标准标识符是预先确定了含义的词法单元，但是程序员可以重新声明它的含义。在这个声明的作用域内，程序员声明的含义起作用，而预先确定的含义消失，如 Pascal 语言的 integer 和 true 等。词法分析器对标准标识符没有什么特别的处理，由符号表管理来解决这件事。

2.27　词法分析器能查出源程序中什么样的错误？举例说明。

答　词法分析器对源程序采取非常局部的观点，因此像 C 语言的语句

```
fi (a == f (x)) …
```

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

Pascal 语言要求作为实型常量的小数点后面必须有数字，如果程序中出现小数点后面没有数字情况，则由词法分析器报错。

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

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

误写成

```
/* then part
```

那么该编译器发现不了遗漏分号的错误。这是为什么?

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

答 此时编译器认为

```
    /* then part
    return q
else
    /* else part */
```

是程序的注释,因此它不可能再发现 else 前面的语法错误。

分析 这是注释用配对符号表示时的一个问题。注释是在词法分析时被忽略的,而词法分析器对程序采取非常局部的观点。当进入第一个注释后,词法分析器忽略输入符号,一直到出现注释的结束符号为止,由于第一个注释缺少结束符号,所以词法分析器在读到第二个注释结束符号时,才认为第一个注释处理结束。

为克服这个问题,后来的语言一般都不用配对符号来表示注释。例如 Ada 语言的注释始于双连字符(--),随行的结束而终止。如果用 Ada 语言的注释格式,那么上面函数应写成

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

第3章　语法分析

一、主要知识点

主教材第3章主要应掌握下面一些内容。

1. 文法和语言的基本知识。有些教材把这方面内容单独作为一章，但是把它和语法分析方法放在一起学习，可以理解得更深刻一些。

2. 自上而下的语法分析方法：预测分析器（递归下降分析方法），非递归的预测分析器（分析表方法），LL(1)文法。能够编写简单语言的预测分析递归程序。

3. 自下而上的语法分析方法：SLR(1)方法，规范LR(1)方法和LALR(1)方法。在这三种方法中，重点是SLR(1)方法，SLR(1)方法理解透彻了，其他两种方法就容易理解了。对于简单的语言，要能够手工构造这三种分析方法的分析表。同时还要知道它们的区别所在，便于直接判断一些简单文法属于哪一类文法而无须构造分析表。

4. LR方法如何用于二义文法。

5. 语法分析的错误恢复方法。

6. 现在编译器已很少使用算符优先分析方法，因此有些教材已不介绍这种方法。在习题中保留了它和其他分析方法的比较。

本章的难点是写文法。本课程虽不要求掌握如何验证文法产生的语言，但了解这方面的技巧对判断所写文法正确与否是极有帮助的，本章有这方面的例子。

更难的是要求所写的文法还需满足一定的约束[非二义、LR(1)]等，本章有多个详细分析的例子。

二、习题精选与解析

3.1　文法

$$S \rightarrow a\,S\,b\,S \mid b\,S\,a\,S \mid \varepsilon$$

产生的语言是什么？该文法是否二义？

答　该文法产生的语言是 a 的个数和 b 的个数相等的串的集合。该文法二义，例如句子 $abab$ 有两种不同的最左推导。

$$S \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSbS \Rightarrow ababS \Rightarrow abab$$

$$S \Rightarrow aSbS \Rightarrow abSaSbS \Rightarrow abaSbS \Rightarrow ababS \Rightarrow abab$$

分析　为什么能判断该文法产生的语言是 a 的个数和 b 的个数相等的串的集合？因为三个产生式的右部，在除去非终结符后，都正好是 a 的个数和 b 的个数相等的情况，而右部的非终结符又正好都是这唯一的非终结符。因此可以按推导步数进行归纳，来证明该文法推导出的句子一定是 a 的个数和 b 的个数相等的串。另一方面还需要判断，每个这样的串，是否都能由这个文法产生。以 a 开始的串为例，它一定能分解成 aw_1bw_2 的形式，其中 w_1 中 a 的个数和 b 的个数相等，w_2 也是这样。如果找不到这样的分解，那么该串中 a 的个数和 b 的个数就一定不相等。这样可以按串长进行归纳，得出所有这样的串都由该文法产生。习题 3.42 专门介绍怎样证明一个文法是某个语言的一个文法。

怎样判断该文法二义，并能找一个最简单的句子来说明问题？以 a 开始的串为例，需要找尽可能简单的句子，使得它按 aw_1bw_2 形式的分解不是唯一的，那么该句子一定有两棵不同的语法树。空串和串 ab 都不可能。那么再看串长为 4 的几种情况：$aabb$，$abab$ 和 $abba$，只有 $abab$ 有可能，这时 $w_1 = \varepsilon$ 且 $w_2 = ab$ 或者 $w_1 = ba$ 且 $w_2 = \varepsilon$。因此 $abab$ 一定有两种不同形式的最左推导，在上面的答案中已经给出了这两种推导。

3.2　下面的二义文法描述命题演算公式的语法，为它写一个等价的非二义文法。

$S \rightarrow S$ **and** $S \mid S$ **or** $S \mid$ **not** $S \mid p \mid q \mid (S)$

答　非二义文法的产生式如下：

$E \rightarrow E$ **or** $T \mid T$

$T \rightarrow T$ **and** $F \mid F$

$F \rightarrow$ **not** $F \mid (E) \mid p \mid q$

分析　和教材上算术表达式的二义文法一样，由于该文法中没有体现算符 **and**、**or** 和 **not** 的优先次序和结合规则，因此该文法二义。如 p **and** q **or** p 可以分解成两个子表达式 p **and** q 和 p 进行 **or** 运算，或者分解成 p 和 q **or** p 进行 **and** 运算，见图 3.1 的两棵不同语法树。

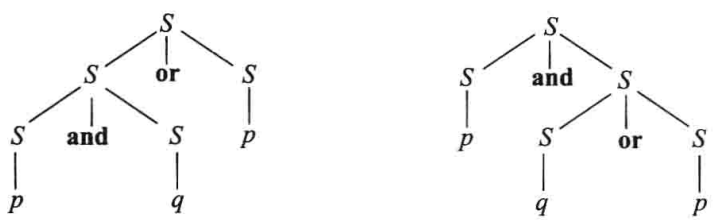


图 3.1　p **and** q **or** p 的两棵不同语法树

要使得一个布尔表达式的分解唯一，就必须体现算符的优先次序和结合规则。可以把一个表达式看成是若干个项进行 **or** 运算，并且 **or** 运算左结合，每个项看成是若干个因子进行 **and** 运算，**and** 运算也左结合，因子有四种情况：p，q，加括号的表达式，或者加 **not** 前缀的因子。按这种观点，一个表达式的结构分解只有一种可能性，因而不会有二义性。上面的文法就是按这种观点写的。

一个易犯的困惑是，究竟用产生式 $F \rightarrow$ **not** E 还是用 $F \rightarrow$ **not** F？在这种由算符优先级体现的层次结构中，若高层结构（如 E）直接作为低层算符（如 **not**，运算的优先级高）的运算对象，很可能会使文法产生二义。在上面的文法中，如果使用 $F \rightarrow$ **not** E 代替 $F \rightarrow$ **not** F，则 **not** p **and**

q 有两棵不同的语法树，见图 3.2（为节约版面，用虚线省略了部分推导）。

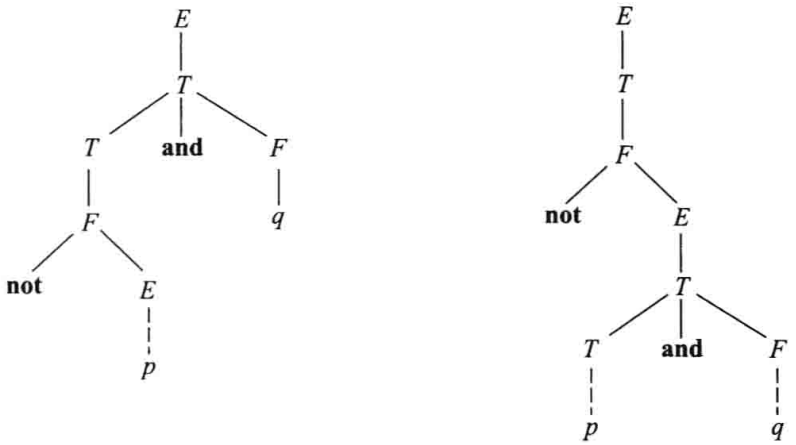


图 3.2　**not** p **and** q 的两棵不同语法树

3.3　文法

$$R \rightarrow R\ \text{`|'}\ R \mid RR \mid R* \mid (R) \mid a \mid b$$

产生字母表 $\{a, b\}$ 上所有不含 ε 的正规式。注意：第一条竖线是正规式的符号“或”，而不是文法产生式右部各选择之间的分隔符；另外“ * ”在这儿是一个普通的终结符。该文法是二义的。为该文法写一个等价的非二义文法。

答　有了上一题的经验，读者应该自己能写出答案。根据正规表达式的约定，“ * ”的优先级最高，连接（无算符）次之，“|”的优先级最低。非二义文法如下：

$$E \rightarrow E\ \text{`|'}\ T \mid T$$

$$T \rightarrow T\ F \mid F$$

$$F \rightarrow F* \mid (E) \mid a \mid b$$

分析　也许你会担心，上一题的 **not** 是右结合的算符，而本题的“ * ”是左结合算符，设计这两个文法时是否要考虑这一点区别。这个问题在设计语言时已考虑了，右结合的一元算符放在运算对象的前面，而左结合的一元算符放在运算对象的后面。3.2 题 **not not** q 和本题 a ** 的语法树见图 3.3，它们分别体现了 **not** 的右结合和“ * ”的左结合。

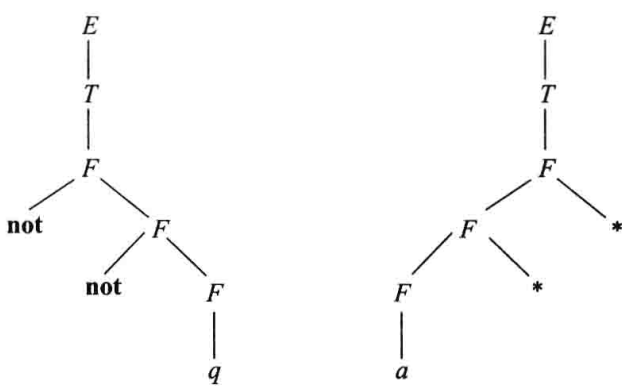


图 3.3　**not not** q 和 a ** 的语法树

3.4 为下面的语言写一个无二义的文法。

ML 语言中用分号分隔语句的语句块，例如：

((*s* ; *s*) ; (*s* ; *s* ; *s*) ; *s*) ; (*s* ; *s*)

答 无二义的文法如下：

$L \rightarrow S \mid L \ ; \ S$

$S \rightarrow s \mid (\ L \ )$

分析 从所给例子可以看出，一对括号及其中的语句块看成一个语句。因此语句有两种选择，简单语句 *s* 和加括号的语句块，即答案中第二行的产生式。答案中第一行的产生式表明语句块是由分号分隔的语句。

* **3.5** 下面的条件语句文法：

stmt → **if** *expr* **then** *stmt* | *matched_stmt*

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

试图消除悬空 *else* 的二义性。请证明该文法仍然是二义的。

答 句型 **if** *expr* **then if** *expr* **then** *matched_stmt* **else if** *expr* **then** *matched_stmt* **else** *stmt* 有两种不同的最左推导（见下面），因此该文法仍然是二义的。

stmt ⇒ *matched_stmt*

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

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

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

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

和

stmt ⇒ **if** *expr* **then** *stmt*

⇒ **if** *expr* **then** *matched_stmt*

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

⇒ **if** *expr* **then if** *expr* **then** *matched_stmt* **else** *matched_stmt*

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

分析 在这个文法中，*matched_stmt* 本意是表示 **then** 和 **else** 配对的语句，但是其 **if** *expr* **then** *matched_stmt* **else** *stmt* 不能保证这一点，因为 **else** 后面是任意的 *stmt*，而不是 *matched_stmt*。找到了可能产生二义的地方，但怎样找一个最短的句型来验证该文法的二义呢？

可以肯定地说，需要用 **then** 比 **else** 多的句型，这样才有可能出现 **else** 和不同的 **then** 配对的可能。最简单的情况是两个 **then** 和一个 **else**，可以肯定，要用产生式 *matched_stmt* → **if** *expr* **then** *matched_stmt* **else** *stmt* 来产生这个 **else**，这个 **else** 只可能和该产生式右部的那个 **then**，而不可能和句子中的另一个 **then** 配对，因此这种情况下无二义。估计三个 **then** 和一个 **else** 的情况也不

出现二义，如果出现，则两个 **then** 和一个 **else** 也应该会出现。现在考虑三个 **then** 和两个 **else** 的情况，句型是：

if *expr* **then if** *expr* **then** *matched_stmt* **else if** *expr* **then** *matched_stmt* **else** *stmt*

为了保证 **else** 和它最靠近的还没有配对的 **then** 相匹配，这里在 **else** 的前面用的都是 *matched_stmt*。用这个句型做尝试，发现了两种不同的最左推导。

解决这样的问题，一定要先把问题分析清楚，然后再去尝试，以减少盲目性。

3.6 下面是用正规式表示的变量声明：

(int | float) **id** (, **id**)* ;

请改用上下文无关文法表示，也就是写一个上下文无关文法，它和该正规式等价。

答 和该正规式等价的一个上下文无关文法如下：

$D \rightarrow T\ L$;

$T \rightarrow$ int | float

$L \rightarrow L$, **id** | **id**

分析 该正规式的前半部分是类型选择，后半部分是用分号隔开的 **id** 表，最后跟上一个分号，句子的结构比较简单。

***3.7** 为字母表{a, b}上的下列每个语言设计一个文法。

(a) 每个 a 后面至少有一个 b 的所有串。

(b) a 和 b 的个数相等的所有串。

(c) a 和 b 的个数不相等的所有串。

(d) 形式为 xy 且 $x \neq y$ 的所有串。

答 (a) $S \rightarrow abS \mid bS \mid \varepsilon$

分析 (a) 该语言是一个正规语言，它可以用正规式$(ab|b)^*$表示。上面的文法对应到该正规式的结构。

答 (b) $S \rightarrow aB \mid bA \mid \varepsilon$

$A \rightarrow aS \mid bAA$

$B \rightarrow bS \mid aBB$

分析 (b)字母表{a, b}上的任何串，从其所含的 a 和 b 的个数来分，属于下面三种情况之一：

- a 和 b 的个数相等；
- a 比 b 的个数多，可以把这样的串分成若干个 a 比 b 的个数多一个的子串；
- b 比 a 的个数多，可以把这样的串分成若干个 b 比 a 的个数多一个的子串。

根据上面的分析，使用三个非终结符：

- 用 S 来推导 a 和 b 的个数相等的串(显然 S 应该作为开始符号)；

- 用 A 来推导 a 比 b 多一个的串；
- 用 B 来推导 b 比 a 多一个的串。

当把三个非终结符的含义确定后，产生式就不难写了。对于 S 来说，显然有产生式 $S \to aB \mid bA \mid \varepsilon$。对于 A 来说，若第一个字符是 a，则剩下的是 a 和 b 的个数相等的串；若第一个字符是 b，则剩下的是 a 比 b 多两个的串，可以分成两个 a 比 b 多一个的子串。所以 A 的产生式是 $A \to aS \mid bAA$。B 的情况类似。

3.1 题的文法和这个文法等价，它们产生同样的语言。在 3.1 题的文法中，S 也是用来推导 a 和 b 的个数相等的串。在 a 和 b 的个数相等的非空串中，在适当的位置去掉一个 a 和一个 b 后，则剩下的部分可分成 a 和 b 的个数相等的两个串（包括空串），所以那个文法只有一个非终结符 S。

本题的文法仍然是二义的，习题 3.8 讨论该语言的非二义文法。

答 （c）利用上一小题的结果，即保留前面的产生式（它们被列在下面右边），但是 S 不再作为开始符号。另外需增加几个产生式（它们被列在下面左边），并且 S' 作为开始符号。

$$S' \to A' \mid B' \qquad S \to aB \mid bA \mid \varepsilon$$

$$A' \to AA' \mid A \qquad A \to aS \mid bAA$$

$$B' \to BB' \mid B \qquad B \to bS \mid aBB$$

分析 （c）新增的三个非终结符的含义如下：

- S' 用来推导 a 和 b 的个数不相等的串（显然 S' 应该作为开始符号）；
- 用 A' 来推导 a 比 b 多的串；
- 用 B' 来推导 b 比 a 多的串。

当它们的含义确定后，产生式就很容易写了。

答 （d）$S \to S_{\text{odd}} \mid S_{\text{even}}$

$$S_{\text{odd}} \to aE \mid bE$$

$$E \to aaE \mid abE \mid baE \mid bbE \mid \varepsilon$$

$$S_{\text{even}} \to S_a S_b \mid S_b S_a$$

$$S_a \to aS_a a \mid aS_a b \mid bS_a a \mid bS_a b \mid a$$

$$S_b \to aS_b a \mid aS_b b \mid bS_b a \mid bS_b b \mid b$$

分析 （d）这是一个非常难的习题。还没有找到更简洁一点的结果。

首先，如果串长为奇数的话，那么就不可能分成两个长度相同的子串，因此所有长度为奇数的串都满足条件。用 S_{odd} 来推长度为奇数的所有串。

对于长度为偶数的串，把它分成长度相等的左右两个子串。只要这两个子串在某个相同位置上，它们的字符不一样，那么这两个子串就肯定不同，如图 3.4（a）所示。但这并没有提示怎样写产生式来产生这样的串。如果把图 3.4（a）中间的 i 和 j 交换一下，如图 3.4（b）所示，就出现了转机。图 3.4（b）前一部分的特点是，中间的字符是 a，其左右两边的字符任意，但长度一样，可以用非终结符 S_a 来推这样的串。图 3.4（b）的后一部分类似，但中间的字符是 b，可以用非终结符 S_b 来推这样的串。这样，$S_a S_b$ 和 $S_b S_a$ 就能推出长度为偶数并且符合条件的所有串，非终结

符 S_{even} 用来表示这两种选择。

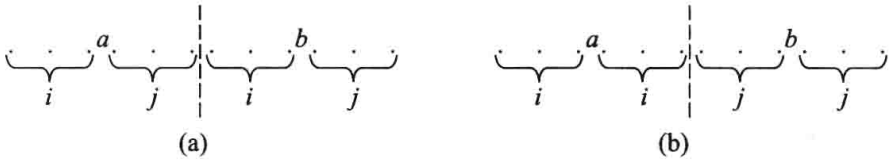


图 3.4　两种不同拆分方式

***3.8**　下面的文法表示的语言是含 a 和 b 个数相等的所有串(见 3.7 题)。

$S \to aB \mid bA \mid \varepsilon$　　S:推导 a 和 b 的个数相等的串

$A \to aS \mid bAA$　　A:推导 a 比 b 多一个的串

$B \to bS \mid aBB$　　B:推导 b 比 a 多一个的串

该文法是二义的,通过直观分析就能判断出来。产生式 $A \to bAA$ 右部的 AA 表示 a 比 b 多两个的串,这样的串分成两个 a 比 b 多一个的子串时,分解方法可能不是唯一的,如 $aaba$ 可分成 a, aba 和 aab, a 两种情况。

请你略微修改上面的文法(不增加新的非终结符,S 的含义不变,A 和 B 的含义在原基础上加以限制),得到一个接受同样语言的非二义文法。

答　$S \to aBS \mid bAS \mid \varepsilon$

$A \to a \mid bAA$

$B \to b \mid aBB$

分析　题目已经告知,出现二义是因为在把 a 比 b 多两个的串分成两个 a 比 b 多一个的子串时,分法可能不是唯一的。b 比 a 多时情况类似。要想消除二义就必须寻找唯一的分解办法。从本题目给出的例子可以看出,$aaba$ 之所以有两种分法,是因为中间两个字符构成的子串 ab 是一个 a 和 b 个数相等的子串,它可以分到前面的子串,也可以分到后面的子串。如果规定它只能出现在后面子串(从而后面部分可能还需细分),也就是规定 A 推出的串中不能有 a 和 b 个数相等的非空后缀。因此 S,A 和 B 的含义如下:

S:推导 a 和 b 的个数相等的串。

A:推导 a 比 b 多一个的串,并且推出的串中不含 a 和 b 个数相等的非空后缀。

B:推导 b 比 a 多一个的串,并且推出的串中不含 a 和 b 个数相等的非空后缀。

这时再看答案中的产生式就不难理解了。用这个文法推导 $abaabb$ 的步骤如下。

$S \Rightarrow aBS \Rightarrow abS \Rightarrow abaBS \Rightarrow abaaBBS \Rightarrow abaabBS \Rightarrow abaabbS \Rightarrow abaabb$

3.9　写一个非二义的文法,它定义的语言是 3.8 题文法定义的语言的子集:每个句子除了 a 和 b 的个数相等外,还不存在 b 比 a 个数多的前缀。例如,$abbbabaaaabb$ 不是这个语言的句子。

答　$S \to aSbS \mid \varepsilon$

分析　非终结符 S 用来推出 a 和 b 个数相等的串,且串中不存在 b 比 a 个数多的前缀。这个性质很容易用归纳法来证明。用归纳法同样可以证明,每个满足这两个条件的串都可以从 S

推出。习题 3.42 有类似问题归纳证明的详细描述。

3.10 构造下面文法的 LL(1)分析表。

$D \rightarrow TL$

$T \rightarrow$ **int** | **real**

$L \rightarrow$ **id** R

$R \rightarrow$, **id** $R \mid \varepsilon$

答 LL(1)分析表见表 3.1。

表 3.1 LL(1)分析表

非终结符	输入符号				
	int	**real**	**id**	,	$
D	$D \rightarrow TL$	$D \rightarrow TL$			
T	$T \rightarrow$ **int**	$T \rightarrow$ **real**			
L			$L \rightarrow$ **id** R		
R				$R \rightarrow$, **id** R	$R \rightarrow \varepsilon$

分析 虽然这个文法很简单,还是从求开始符号集合和后继符号集合开始。

$FIRST(D) = FIRST(T) =$ {**int**, **real**}　　$FOLLOW(D) = FOLLOW(L) =$ {$}

$FIRST(L) =$ {**id**}　　$FOLLOW(T) =$ {**id**}

$FIRST(R) =$ {, , ε}　　$FOLLOW(R) =$ {$}

有了上面每个非终结符的 *FIRST* 集合,填写分析表时要计算一个产生式右部α的 $FIRST(\alpha)$ 就不是件难事了。

填表时唯一要小心的是,ε 是产生式 $R \rightarrow \varepsilon$ 右部的一个开始符号,而 $ 在 $FOLLOW(R)$ 中,所以 $R \rightarrow \varepsilon$ 填在输入符号 $ 的栏目中。

3.11 构造下面文法的 LL(1)分析表。

$S \rightarrow aBS \mid bAS \mid \varepsilon$

$A \rightarrow bAA \mid a$

$B \rightarrow aBB \mid b$

答 各非终结符的开始符号集合和后继符号集合见表 3.2,LL(1)分析表见表 3.3。

表 3.2 开始符号和后继符号

	开始符号	后继符号
S	a, b, ε	$
A	a, b	a, b, $
B	a, b	a, b, $

表 3.3 LL(1)分析表

	a	b	$
S	$S \rightarrow aBS$	$S \rightarrow bAS$	$S \rightarrow \varepsilon$
A	$A \rightarrow a$	$A \rightarrow bAA$	
B	$B \rightarrow aBB$	$B \rightarrow b$	

3.12 下面文法是否为 LL(1)文法？说明理由。

$S \rightarrow AB \mid PQx$　　$A \rightarrow xy$　　$B \rightarrow bc$

$P \rightarrow dP \mid \varepsilon$　　$Q \rightarrow aQ \mid \varepsilon$

答 该文法不是 LL(1)文法，理由如下。

在题目所给产生式中，只有三个非终结符有两个选择。

1. P 的两个右部 dP 和 ε 的开始符号肯定不相交。

2. Q 的两个右部 aQ 和 ε 的开始符号肯定不相交。

3. 对 S 来说，因为有 $x \in FIRST(AB)$，同时还有 $x \in FIRST(PQx)$（因为 P 和 Q 都可能为空）。

所以该文法不是 LL(1)文法。

3.13 Pascal 语言的语法分析器通常采用预测分析方法，但是 Pascal 语言的文法并不是 LL(1)文法，因为赋值语句和过程语句都由标识符开始，见下面文法：

〈语句〉→〈赋值语句〉|〈过程调用语句〉| …

〈赋值语句〉→〈变量〉:=〈表达式〉

〈变量〉→〈变量标识符〉…

〈变量标识符〉→〈标识符〉

〈过程调用语句〉→〈过程标识符〉…

〈过程标识符〉→〈标识符〉

你用什么办法来克服这个问题给预测分析带来的困难？

答 首先，可以用主教材上关于提取公共左因子的办法进行文法变换，将文法变换成 LL(1)文法。但 Pascal 语言编译器代码大都不这么做，因为这会降低编译器的可读性。

另一种办法是让词法分析器变得聪明一点，它通过查标识符表来判断标识符是否为过程标识符。若是，它返回区别于一般标识符的不同记号，比如有 **id** 和 **procid** 的区分，这就相当于不需要上面最后一行的产生式了。按这种办法，词法分析器每读到一个标识符，都需要检查一下它是否为过程标识符，这会降低编译器的效率，所以编译器一般也不这么做。

实际的做法是在语句的分析过程中，通过查符号表来获得标识符的属性，由此来决定分支，见下面语句分析过程的框架。不过，下面的过程已不是纯的语法分析过程了，它使用了静态语义分析的信息来决定语法分析的分支。

```
void statement( ) {
    if (lookahead == 标识符)          {// 如果下一个记号是标识符
        if (查符号表得知 lookahead 是过程标识符)
            call_statement( );          // 调用过程调用语句的分析过程
        else
            assign_statement( );        // 调用赋值语句的分析过程
```

```
    } else
        . . .                          // 其他语句
}
```

3.14 下面是类型表达式的语法：

type → **integer** | **boolean** | **array**[**num**] **of** *type* | **record** *field_list* **end** | ↑ *type*

field_list → **id** : *type* | **id** : *type* ; *field_list*

若规定：在记录类型中不能出现数组类型(包括不能出现数组的指针类型)。请重新设计一个文法，把该约束体现在文法中，即它和上述文法的区别就是所定义的语言满足这个约束。

答 体现所给约束的文法如下。

type→ **integer** | **boolean** | **array**[**num**] **of** *type* | **record** *field_list* **end** | ↑ *type*

field_list → **id** : *field_type* | **id** : *field_type* ; *field_list*

field_type → **integer** | **boolean** |**record** *field_list* **end** | ↑ *field_type*

分析 这些约束比较简单，可以体现在上下文无关文法中。在主教材 3.2“语言和文法”一节已经介绍了多个不能用上下文无关文法表达的语言约束。

3.15 根据主教材 3.3.3 节的递归下降预测分析方法，给 3.14 题题目(而不是答案)中的非终结符 *field_list* 写一个递归过程(若需要变换文法，则先变换文法)。

答 可以将文法修改如下：

field_list → **id** : *type* [; *field_list*]

递归过程如下：

```
void field_list( ) {
    match(id); match(':'); type( );
    if(lookahead == ' ;'){
        match(' ;'); field_list( );
    }
}
```

3.16 算符优先分析方法是一种自下而上的移进-归约方法，举例说明它不是一种规范归约方法。

答 例如，表达式的文法

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow \mathbf{id} \mid (E)$

是算符优先文法。对于表达式 **id** + **id**，如果用基于该文法构造的算符优先关系进行分析，那么

第一个 **id** 到 E 的归约仅一步，而不是从 **id** 归约成 F，再归约成 T，最后归约成 E。即右部是单个非终结符的产生式 $E \rightarrow T$ 和 $T \rightarrow F$ 的归约都被省略，因此它不是规范归约。

3.17 给出接受文法

$$S \rightarrow (L) \mid a \qquad L \rightarrow L, S \mid S$$

的活前缀的一个 DFA。

答 首先将文法进行如下拓广，其中 S' 是拓广文法的开始符号。

$$S' \rightarrow S \qquad S \rightarrow (L) \mid a \qquad L \rightarrow L, S \mid S$$

接受该文法活前缀的一个 DFA 的状态转换图见图 3.5，开始状态是 I_0，所有的状态都是接受状态。

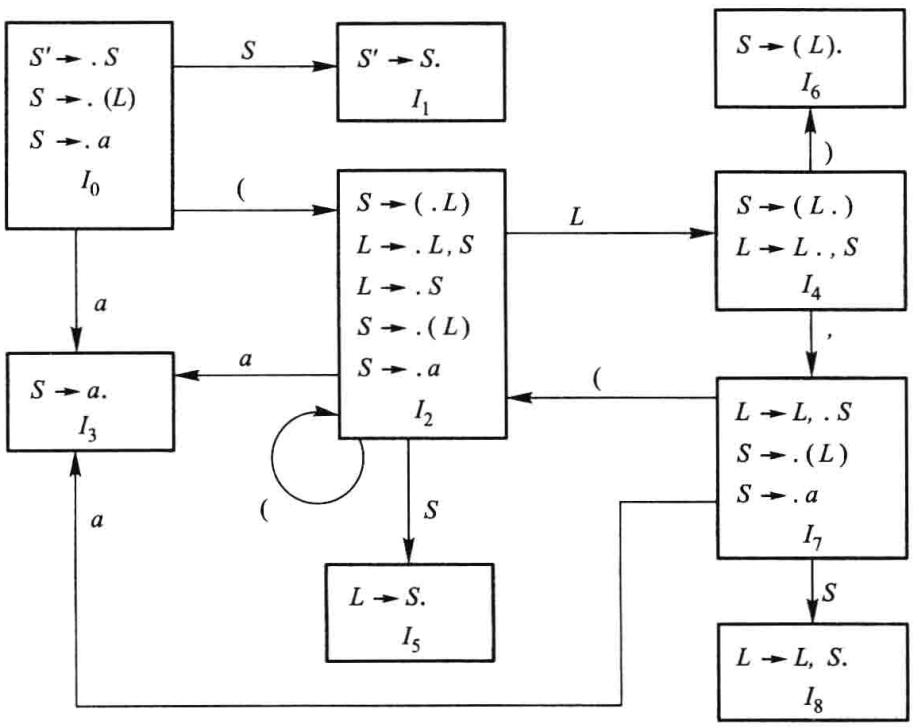


图 3.5 识别文法活前缀的 DFA

分析 文法活前缀是 LR 分析的一个重要概念，接受文法活前缀的 DFA 是构造 LR 分析表过程中的重要一步。一些同学在构造 LR 分析表时会画这样的状态转换图，但是却未理解这是接受文法活前缀的 DFA。

构造 SLR(1)分析表、规范 LR(1)分析表和 LALR(1)分析表时都需要先构造这样的 DFA，由于用 SLR(1)方法得到的 DFA 的状态数最少[实际上这个 DFA 就是用 LR(0)方法构造的 DFA]，因此用 SLR(1)方法。

3.18 一个文法如下：

$$S \rightarrow (S)$$

$S \rightarrow a$

请给出该文法中对活前缀(((有效的 LR (1)项目。

答 该文法中对活前缀(((有效的 LR (1)项目如下:

$S \rightarrow (\ .\ S),\quad)$

$S \rightarrow .\ (S),\quad)$

$S \rightarrow .\ a,\quad)$

分析 构造识别该文法活前缀的 DFA 时,实际上只要画出三个 LR(1)项目集就可以看出结果了,这三个项目集见图 3.6。

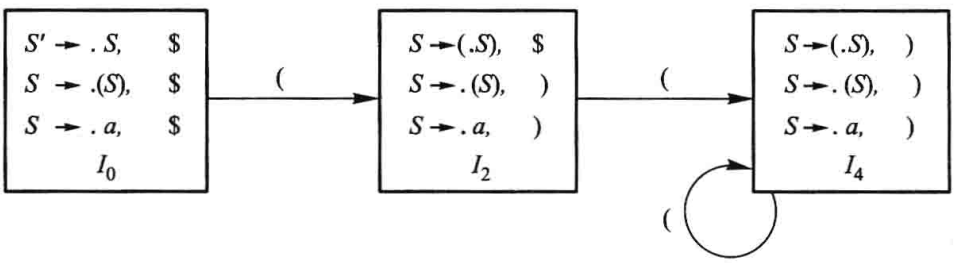


图 3.6 识别文法活前缀的三个项目集

3.19 接受文法

$S \rightarrow Aa \mid bAc \mid dc \mid bda$

$A \rightarrow d$

活前缀的 DFA 见图 3.7。请根据这个 DFA 来构造该文法的 SLR(1)分析表,并说明该文法为什么不是 SLR(1)文法。

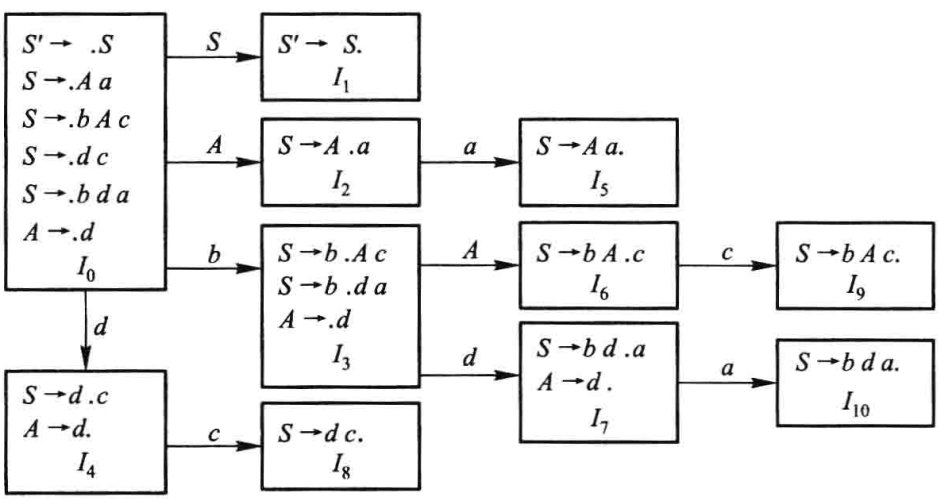


图 3.7 识别文法活前缀的 DFA

答 分析表见表 3.4,注意 $FOLLOW(A) = \{a, c\}$。因为有两处有移进-归约冲突,所以该文法不是 SLR(1)文法。

表 3.4　为 3.19 题的文法构造的分析表

状态	动作					转移	
	a	*b*	*c*	*d*	$	*S*	*A*
0		*s*3		*s*4		1	2
1					*acc*		
2	*s*5						
3				*s*7			6
4	*r*5		*s*8, *r*5				
5					*r*1		
6			*s*9				
7	*s*10, *r*5						
8					*r*3		
9					*r*2		
10					*r*4		

3.20　证明文法

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

是 LL(1)文法,但不是 SLR(1)文法。

答　先证明它是 LL(1)文法。因为仅非终结符 S 有两个不同的选择,因此只需考察它的两个右部。由于

$$FIRST(AaAb) = \{ a \} \qquad FIRST(BbBa) = \{ b \}$$

它们的交集为空,所以该文法是 LL(1)文法。

这里不是通过构造 SLR(1)分析表,而是通过简单的分析来判断该文法不是 SLR(1)。根据该文法,不管对什么句子,在面临第一个符号时进行空归约。由于 $FOLLOW(A)$ 和 $FOLLOW(B)$ 都等于 $\{a, b\}$,那么就不能确定是把 ε 归约成 A 还是归约成 B,因此出现归约-归约冲突,所以该文法不是 SLR(1)的。

分析　其实,该语言只有两个句子 ab 和 ba,这个冲突是很容易解决的。但是由于用 SLR 方法构造分析表时,不是根据向前搜索符,而是根据所有可能的后继符号来填归约项目,这种粗糙性决定了会出现这样的归约-归约冲突。

不要从这个例子得出 LL(1)方法比 SLR(1)方法强的错误结论。例如,习题 3.16 中的表达式文法是 SLR(1)的,但它不是 LL(1)的,因为它有左递归。

3.21 证明文法

$S \rightarrow Aa \mid bAc \mid dc \mid bda$

$A \rightarrow d$

是 LALR(1)文法,但不是 SLR(1)文法。

答 通过构造分析表来回答这个问题并不困难(见 3.23 题),但是在文法很简单时,也应该能直观地分析出来。先说明该文法不是 SLR(1)文法。从产生式很容易看出 $FOLLOW(A) = \{a, c\}$。若句子是 dc,在 d 进分析栈后,面临的是 c,这时出现移进-归约冲突,因为项目 $[S \rightarrow d.c]$要求移进,而由于 c 在 $FOLLOW(A)$中,项目$[A \rightarrow d.]$要求归约(这两个项目出现在同一项目集中)。

上面的移进-归约冲突在规范 LR(1)情况下不存在,因为只有在面临 a 时才进行 d 到 A 的归约。

上面的文法还有另一个移进-归约冲突,在 bd 进栈后,面临 a 的情况,其冲突的原因和上面的类似。该冲突在规范 LR(1)情况下也不存在。这样,该文法是 LR(1)文法。

显然,该文法的规范 LR(1)项目集的集合中没有同心项目集,因此该文法也是 LALR(1)文法。

3.22 证明文法

$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$

$A \rightarrow d$

$B \rightarrow d$

是 LR(1)文法,但不是 LALR(1)文法。

答 该文法产生的语言仅含四个句子:da,bdc,dc 和 bda。不难看出,在规范 LR(1)的情况下,d 在什么情况下归约成 A、什么情况下归约成 B 是能区分清楚的,所以它是 LR(1)文法。

在该文法的规范 LR(1)项目集的集合中,对活前缀 d 有效的项目集是

$\{[A \rightarrow d.\ ,\ a],\ [B \rightarrow d.\ ,\ c]\}$

对活前缀 bd 有效的项目集是

$\{[A \rightarrow d.\ ,\ c],\ [B \rightarrow d.\ ,\ a]\}$

这两个集合都不含冲突,并且它们同心。合并同心集后变成

$\{[A \rightarrow d.\ ,\ a / c],\ [B \rightarrow d.\ ,\ a / c]\}$

出现归约-归约冲突,所以该文法不是 LALR(1)文法。

3.23 说明下面的文法不是 SLR(1)文法,并重写一个等价的 SLR(1)文法。

$S \rightarrow Aa \mid bAc \mid dc \mid bda$

$A \rightarrow d$

答 分析表的部分状态见图 3.8,虽然只给出部分状态,但可能有的冲突都表现出来了。

因为 a 是 A 的后继符号之一,因此在上面最右边一个项目集中有移进-归约冲突。类似地,下面中间那个状态也有移进-归约冲突。

等价的 SLR(1)文法是:

$S \rightarrow da \mid bdc \mid dc \mid bda$

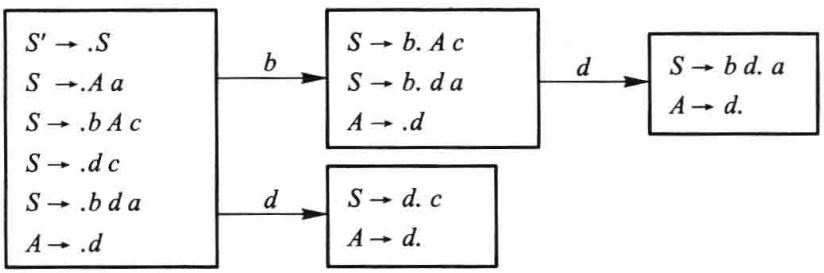


图 3.8 分析表的有关状态

3.24 下面的文法不是 LR(1)的,对它略做修改,使之成为一个等价的 SLR(1)文法。

program →**begin** *declist* ; *statement* **end**

declist → *d* ; *declist* | *d*

statement →*s* ; *statement* | *s*

答 修改后的文法如下:

program →**begin** *declist statement* **end**

declist → *d* ; *declist* | *d* ;

statement →*s* ; *statement* | *s*

分析 该文法产生的句子的形式是:

begin *d* ; *d* ; … ; *d* ; *s* ; *s* ; … ; *s* **end**

即主要是由分号分隔的若干个声明构成的声明表和由分号分隔的若干个语句构成的语句表,两表之间用分号分隔。由于分号既作为声明的分隔符,又作为声明表和语句表的分隔符,这就出现一个问题。该文法的第一步归约是将最后一个 d 归约成 *declist*。当分析器将 d 压入分析栈,面临";"时,它不能判断该 d 是否为最后一个 d,因此有移进-归约冲突。

如果把 d 后面的分号压入栈后再决定动作,移进-归约冲突就会消失。按这样的想法来修改文法,修改后的文法把声明 d 后的分号看成是 d 的结束符,因此把原来分隔声明表和语句表的分号看成是最后一个声明 d 的结束符。这时就不会有移进-归约冲突了。

***3.25** SLR 分析表、LALR 分析表和 LR 分析表都分成动作表和转移表两部分,为什么在分析过程中访问它们的转移表时决不会进入出错条目?

答 这些分析方法都具有活前缀性质,即已经移入分析栈的文法符号一定构成该文法的一个活前缀。这些分析方法只有在将分析栈顶的若干符号归约成一个非终结符时才会访问转移

表，由于这一步归约仍然保持活前缀性质，因此访问转移表时决不会进入出错条目。

3.26 一个非 LR(1)的文法如下：

$L \rightarrow M\ L\ b \mid a$

$M \rightarrow \varepsilon$

请给出有移进-归约冲突的所有规范 LR(1)项目集，以说明该文法确实不是 LR(1)的。

答 首先将文法进行如下拓广，其中 L'是拓广文法的开始符号。

$L' \rightarrow L$

$L \rightarrow M\ L\ b \mid a$

$M \rightarrow \varepsilon$

该文法有移进-归约冲突的规范 LR(1)项目集是图 3.9 中的 I_0，I_2和 I_5。它们的移进-归约冲突都一样，就是在面临 a 时，是移进 a 呢还是进行 ε 归约？

分析 如果完整地构造接受活前缀的 DFA，然后找出有冲突的项目集并不是一件困难的事。在此希望的是，从开始项目集 I_0，怎样尽快地把有冲突的所有项目集找出来，并避免构造所有的项目集，以节约答题时间。这需要对该文法进行分析。从图 3.10 句子 *abbb* 的分析树很容易看出，该文法不是二义的，而且该文法也不是 LR(1)的。

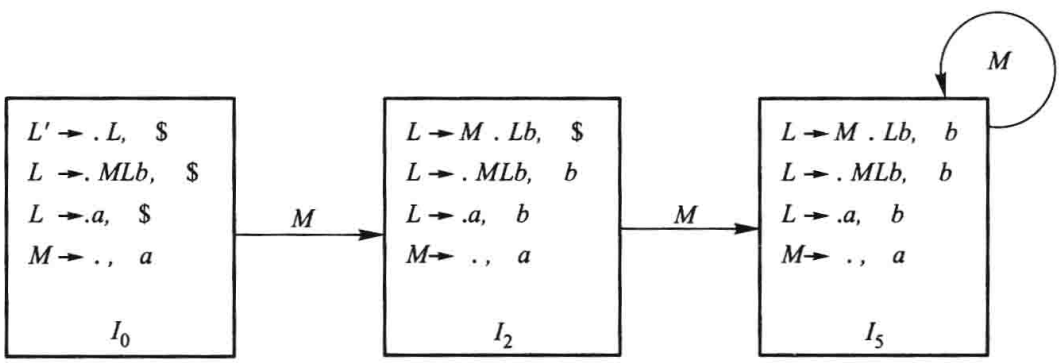


图 3.9 有移进-归约冲突的项目集

从这个句子的分析树可以看出，分析器在面临 a、未读入一个符号的情况下要做若干次 ε 归约，ε 归约的次数和 a 后面 b 的个数一致。但是分析器在面临 a 时，也就是只向前看到第一个符号的情况下，是不可能知道后面有多少个 b 的，因此在未读入任何符号时就出现移进-归约冲突。一旦移进 a 后，就不再有冲突了，也就是在其他场合没有冲突。

这样，开始状态会出现冲突（因为未读入任何符号就出现移进-归约冲突），开始状态的 M 转换到达的状态可能有冲突（因为 M 转换是 ε 归约引起的，仍然未读入任何符号），从开始状态出发的其他任何转换都不会出现冲突。故在此只画 M 转换能到达的状态。

另外，对任意的 k，该文法也不是 LR(k)的。因为总有 b 的个数比 k 还多的句子，使得向前看 k 个符号也不知道一共有多少个 b，因此仍然决定不了要做多少次 ε 归约。

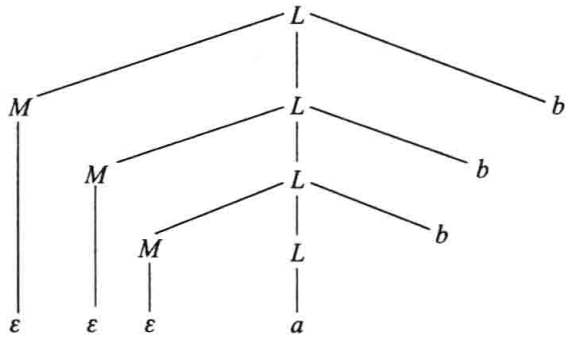


图 3.10　句子 *abbb* 的分析树

3.27　(a)为下面文法构造规范 LR(1)分析表,只要画出状态转换图就可以了。

$S \rightarrow V = E \mid E$

$V \rightarrow *E \mid \mathbf{id}$

$E \rightarrow V$

(b)上述状态转换图有同心项目集吗? 若有,合并同心项目集后是否会出现动作冲突?

答　(a)状态转换图见图 3.11。

(b)合并同心项目集(I_4和I_{11},I_5和I_{12},I_7和I_{13},I_8和I_{10})后没有动作冲突。

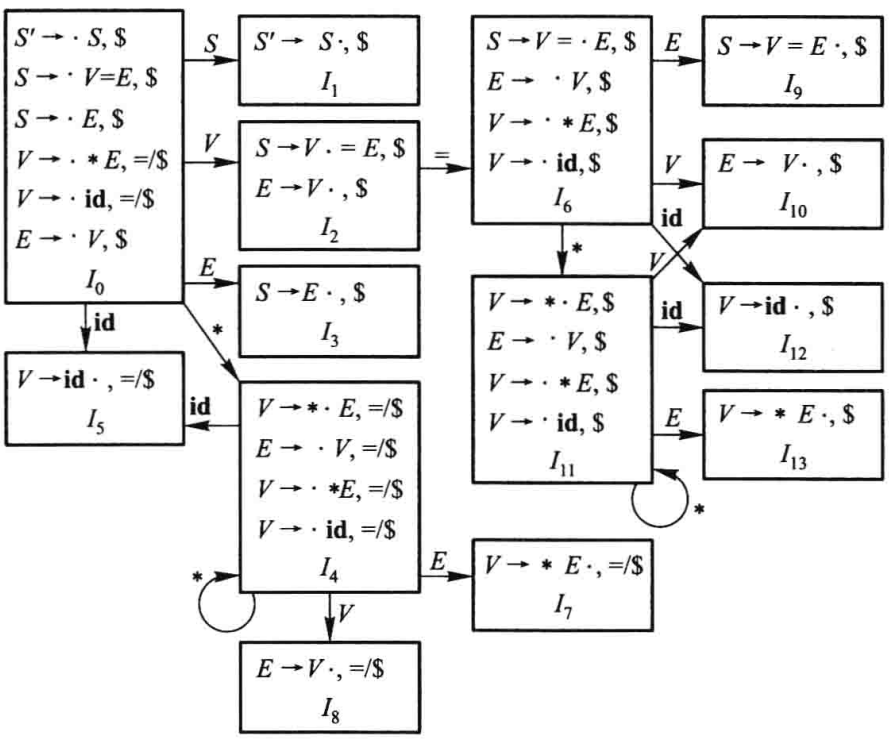


图 3.11　状态转换图

3.28　考虑下面的表达式文法,它包括数组访问、加和赋值:

$E \rightarrow E[E] \mid E + E \mid E = E \mid (E) \mid \mathbf{id}$

该文法是二义的。请写一个接受同样语言的 LR(1)文法,其优先级从高到低依次是数组访问、加和赋值,并且加运算是左结合,赋值是右结合。不需要给出你的文法是 LR(1)文法的证明。

答 符合条件的文法如下:

$E \rightarrow T = E \mid T$

$T \rightarrow T + F \mid F$

$F \rightarrow F[E] \mid (E) \mid \mathbf{id}$

分析 本题和3.2题类似,结果文法和主教材上的算术表达式LR(1)文法也类似,难的是怎么断定该文法是LR(1)的。有把握的做法是通过构造LR(1)分析表来判断,但这比较费时。直观的判断基于下面两点,再加上一点猜测。

1. 它同3.2题的布尔表达式文法和3.16题的算术表达式文法十分相似,仅增加了 $F[E]$ 作为基本运算对象。

2. 在 $F[E]$ 中,F 的后继符号是[,E 的后继符号是],都是新符号,因此对文法的其他部分来说,加基本运算对象 $F[E]$ 同加一个基本运算对象 **const** 没有多少区别。

3.29 为下面的语言写一个LR(1)文法(不要超过6个产生式):

所有的配对括号串,每个配对括号串最内层的配对括号内出现由逗号分隔的 **id** 列表。例如(( ) (**id**, **id**, **id**))(**id**)。

答 该语言的LR(1)文法如下:

$S \rightarrow (S)\ S \mid (L)\ S \mid \varepsilon$

$L \rightarrow \mathbf{id} \mid L\ ,\ \mathbf{id}$

分析 这是把一般教材上都会出现的配对括号文法和 **id** 列表的文法结合起来,所以写出该文法并判断它是LR(1)的都不应该有什么困难。

3.30 说明文法:

$S \rightarrow aSa \mid bSb \mid \varepsilon$

不是LR(1)文法。

答 该文法接受的语言是:

$L = \{\ ww^{R} \mid w \in (a \mid b\ )^{*}\}$

其中,w^{R} 表示 w 的逆序。直观上说,对于该语言的任何一个句子,如 $abaaba$,扫描前一半字符时是压栈,扫描后一半字符时是先做 ε 归约,然后将后一半字符和栈中的字符通过归约来逐个进行比较,以保证后一半是前一半的逆。问题是,向前看一个字符无法判断是否已经到达串的中点,因此该文法不是LR(1)的。构造分析表时肯定会出现移进-归约冲突。

分析 事实上,对于任意大的 k,总能找到一个句子,就是向前看 k 个字符也无法判断是否应该做 ε 归约了,因此该文法也不是LR(k)的。

还可以判断,该语言不是用 LR 文法能描述的语言。习题 3.21 的文法虽然是非 LR(1)的,但是很容易看出,存在描述那个语言的 LR(1)文法。

如果把这个语言稍微改动一下:

$$L' = \{ wcw^R \mid w \in (a \mid b)^* \}$$

这个语言就是 LR(1)的,因为中间的 c 使得分析器知道什么时候该从移进转为归约。

***3.31** 由文法二义引起的 LR(1)分析动作冲突,可以依据消除二义的规则而得到该文法的 LR(1)分析表,根据此表可以正确识别输入串是否为相应语言的句子。对于非二义非 LR(1)文法引起的 LR(1)分析动作的冲突,是否也可以依据某种规则来消除 LR(1)分析动作的冲突而得到 LR(1)分析表,并且根据此表识别相应语言的句子?若可以,你是否可以给出这样的规则?

答 不存在这样的规则。

分析 文法二义之所以会引起动作冲突,是因为有些句子有不同的分析树。消除分析表中的动作冲突,实际上是将每个句子的分析树限定到唯一。消除二义引起的分析动作的冲突并不会改变能接受的句子集合。

对于非二义非 LR(1)文法,在有移进-归约冲突的地方,若保留移进,它表示接受某种结构的句子;而反过来,若保留归约,它表示接受另一种结构的句子。因此不管怎样解决冲突,总会丢失一部分句子不能接受。

例如,对于习题 3.26 的文法,对于项目集 I_0,若保留移进,则该分析器只接受一个句子 a。若该文法的三个有冲突的项目集都保留归约,那么该分析器会不停地做 ε 到 M 的归约,永远不停止,什么句子也不接受。

3.32 现有句型 $\gamma l\beta bw$。其规范推导是 $S \Rightarrow \cdots \Rightarrow \gamma Abw \Rightarrow \gamma l\beta bw$,最后一步用的产生式是 $A \rightarrow l\beta$。分别指出 LL(1)方法和 LR(1)方法在扫描到句型 $\gamma l\beta bw$ 的什么位置时决定用产生式 $A \rightarrow l\beta$?

答 见图 3.12。

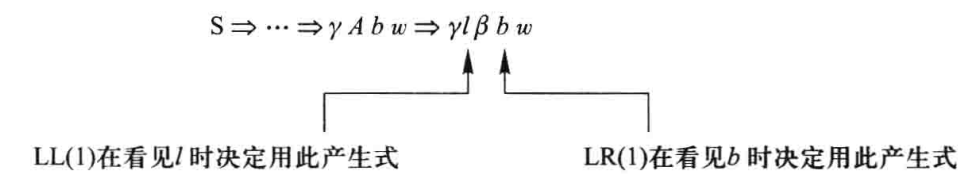


图 3.12 LL(1)和 LR(1)的比较

分析 LL(1)方法在看见右部 $l\beta$ 的第一个符号时决定用这个产生式,而 LR(1)方法在看见右部 $l\beta$ 的后继符号时决定用这个产生式。显然,和 LL(1)方法相比,LR(1)方法是在掌握了更多的信息后才决定用哪个产生式,因此 LR(1)方法能力较强。

*3.33 尽你的可能，对 LL(1)分析方法、算符优先分析方法和 LR 分析方法做一些直观的比较。

答 见表 3.5。

表 3.5 三种分析方法的比较

	LR(1)方法	算符优先方法	LL(1)方法
建立分析树的方式	自下而上	自下而上 仅得到分析树的骨架	自上而下
归约还是推导	规范归约	不规范归约	最左推导
决定使用产生式的时机	看见产生式整个右部推出的内容后才能决定用哪个产生式进行归约	看见产生式整个右部推出的内容后才能决定用哪个产生式进行归约	看见产生式右部推出的第一个终结符后便可确定用哪个产生式进行推导
对文法的显式限制	对文法没有限制	对文法有限制：必须是算符文法	对文法有限制：无左递归、无公共左因子
分析表比较	状态×文法符号 分析表大	终结符×终结符 分析表小	非终结符×终结符 分析表小
分析栈比较	状态栈（为了直观起见，主教材上将文法符号也入栈） 通常状态比文法符号包含更多信息	终结符栈	文法符号栈
确定句柄	根据栈顶状态和下一个符号便可以确定句柄和归约所用产生式	在分析栈顶查找满足 <·= = =… =·>关系的符号序列	无句柄概念
语法错误	决不会将出错点后的符号移入分析栈	有可能将出错点后的符号移入分析栈	和 LR 一样，决不会读过出错点而不报错

3.34 下面两个文法中哪一个不是 LR(1)文法？对非 LR(1)的那个文法，给出有移进-归约冲突的规范的 LR(1)项目集。

$S \rightarrow aAc$　　$S \rightarrow aAc$

$A \rightarrow Abb \mid b$　　$A \rightarrow bAb \mid b$

答 上面右边的文法不是 LR(1)文法，有移进-归约冲突的规范 LR(1)项目集是图 3.13 的状态 I_6。

分析 上面两个文法接受同样的语言。对于 b 的归约，左边的文法是先将第一个 b 归约成 A，然后每两个 b 归约一次。而右边的文法是先将中间的 b 归约成 A，然后每次前后各取一个 b 归约。问题是向前看一个符号没法确定是否到了中间那个 b，所以右边的文法不是 LR(1)文法。

怎样尽快找到移进-归约冲突项目集，并且避免画出所有的项目集？前面已经举过例子。

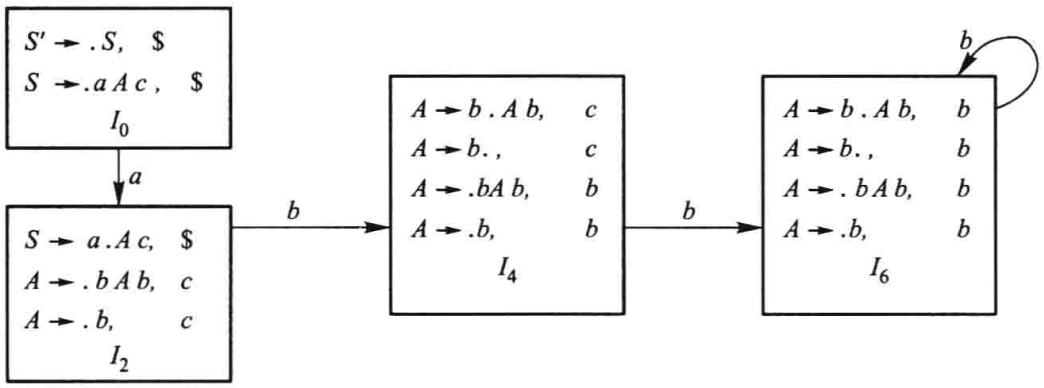


图 3.13　部分 LR(1)项目集

不难看出,如果句子不止一个 b,那么冲突出现在第二个 b 进栈以后,此时不知该将栈顶的 b 归约还是该将后面的 b 移进栈。根据这个判断,在此仅画出上面四个项目集。注意,上面的状态 I_4 没有冲突,$[A \to b.\ ,c]$ 是指在面临 c 时进行归约,这是句子只含一个 b 的情况。

3.35　下面的二义文法产生由逗号隔开的标识符表:

$L \to L,\ L \mid \mathbf{id}$

你打算如何解决移进-归约冲突,以便得到该文法的规范 $LR(1)$ 分析表?

答　优先归约,它体现出“,”是左结合的算符。

分析　该文法产生的句子是逗号隔开的 **id** 序列。把逗号看成像“+”一样的二元算符,那么该文法的二义是因为没有确定该算符的结合规则引起的。如果取左结合,那么应该优先归约;如果取右结合,那么应该优先移进。在此用优先归约来解决移进-归约冲突,那么对任何一个句子的移进和归约动作,同根据下面文法得到的移进和归约动作是一致的。

$L \to L,\ \mathbf{id} \mid \mathbf{id}$

3.36　现有字母表 $\Sigma = \{\ a\ \}$,写一个和正规式 a^* 等价的上下文无关文法,要求所写的文法既不是 LR 文法,也不是二义文法。

答　满足条件的一个文法如下:

$S \to aSa \mid a \mid \varepsilon$

3.37　(a)通过构造识别活前缀的 DFA 和构造分析表,来证明文法 $E \to E + \mathbf{id} \mid \mathbf{id}$ 是 SLR(1)文法。

(b)下面左右两个文法都和(a)中的文法等价:

$E \to E + M\ \mathbf{id} \mid \mathbf{id}$　　$E \to M\ E + \mathbf{id} \mid \mathbf{id}$

$M \to \varepsilon$　　$M \to \varepsilon$

请指出其中有几个文法不是 LR(1)文法,并给出它们不是 LR(1)文法的理由。

答 (a)接受该文法活前缀的 DFA 见图 3.14,SLR 分析表见表 3.6。表 3.6 中没有多重定义的条目,因此该文法是 SLR(1)的。

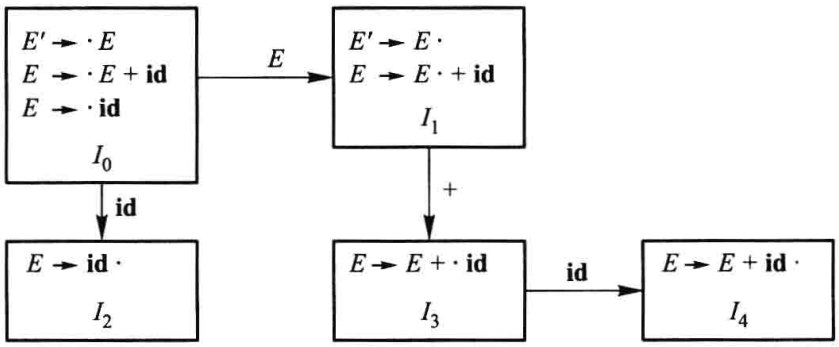


图 3.14 接受文法活前缀的 DFA

表 3.6 SLR 分析表

状态	动作			转移
	id	+	$	*E*
0	*s*2			1
1		*s*3	*acc*	
2		*r*2	*r*2	
3	*s*4			
4		*r*1	*r*1	

(b)只有文法

$E \rightarrow M\ E$ + **id** | **id**

$M \rightarrow \varepsilon$

不是 LR(1)文法。因为对于句子 **id**+**id**+…+**id** 来说,分析器在面临第一个 **id** 时需要做的空归约次数和句子中+**id** 的个数一样多,而此时句子中+**id** 的个数是未知的。

***3.38** 为语言

$L = \{\ a^m b^n \mid n > m \geqslant 0\ \}$

写三个文法,它们分别是 LR(1)的、二义的和非二义且非 LR(1)的。

答 该语言的一个 LR(1)文法是:

$S \rightarrow AB$

$A \rightarrow aAb \mid \varepsilon$

$B \rightarrow Bb \mid b$

该语言的一个二义文法是:

$S \rightarrow aSb \mid Sb \mid b$

该语言的一个非二义且非 LR(1)的文法是

$S \rightarrow aSb \mid B$

$B \rightarrow Bb \mid b$

分析 这种问题看起来很困难,但是细心分析的话也并不难解决。

该语言的句子是 $aa \cdots abb\cdots b$ 的形式,其中后面 b 的个数比前面 a 的个数多。为了保证 b 出现在 a 的后面,并且 b 的个数不少于 a 的个数,那么应该有 $S \rightarrow aSb$ 这样的产生式。为了能推导出更多的 b,应该有形如 $S \rightarrow Sb$ 的产生式。前一个产生式是把后面的 b 和前面的 a 进行配对,由于 b 的个数比 a 的多,配对方式可以有多种。但是如果产生式把这种配对方式限定到唯一,那么文法就不会是二义的。

如果将 $a^m b^n$ 的前 m 个 b 和 a^m 配对,如图 3.15 所示,那么它的文法就是上面的 LR(1)文法。因为做语法分析时,首先是把所有的 a 都压进栈,遇到第一个 b 时,开始退栈和配对(即 aAb 归约成 A),配对结束后将剩余的 b 归约成 B。

如果将 $a^m b^n$ 的后 m 个 b 和 a^m 配对,如图 3.16 所示,那么它的文法就是上面的非二义且非 LR(1)文法。因为做语法分析时,在所有的 a 进栈后,要将中间的若干个 b 归约成 B,剩下的 b 的个数要和前面 a 的个数一致。向前看一个符号是不可能确定将中间多少个 b 归约成 B 的。

如果让 $a^m b^n$ 中的 a 和 b 有不止一种配对方式,如图 3.17 所示,那么它的文法就是上面的二义文法。该示例两个不同的最左推导如下:

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb$

$\Rightarrow aaaSbbbb \Rightarrow aaabbbbb$

$S \Rightarrow aSb \Rightarrow aSbb \Rightarrow aaSbbb$

$\Rightarrow aaaSbbbb \Rightarrow aaabbbbb$

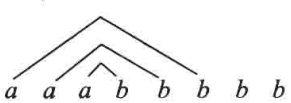


图 3.15 一种配对方式

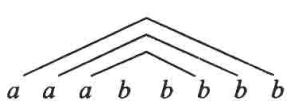


图 3.16 另一种配对方式

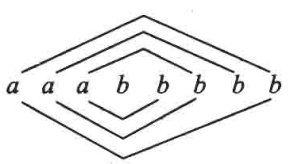


图 3.17 第三种配对方式

* **3.39** 为语言

$L = \{ a^m b^n \mid m > n \geqslant 0 \}$

写三个文法,它们分别是 LR(1)的、二义的和非二义且非 LR(1)的。

答 该语言的一个 LR(1)文法是

$S \rightarrow aS \mid aA$

$A \rightarrow aAb \mid \varepsilon$

该语言的一个二义文法是

$S \rightarrow aSb \mid aS \mid a$

该语言的一个非二义且非 LR(1)的文法是

$S \rightarrow AB$

$A \rightarrow Aa \mid a$

$B \rightarrow aBb \mid \varepsilon$

分析 下面还有两个类似的题目，之所以给出好几个这样的题目，是因为这些题目有助于认识 LR 的本质，而不会把对 LR 的认识局限在构造分析表的算法上。

本题和上题十分类似，因此不再给出具体分析。

***3.40** 为语言

$L = \{a^m b^n \mid 0 \leqslant m \leqslant 2n\}$（即 a 的个数不超过 b 的个数的两倍）

写三个文法，它们分别是 LR(1)的、二义的和非二义且非 LR(1)的。

答 该语言的一个 LR(1)文法是

$S \rightarrow AB$

$A \rightarrow aaAb \mid ab \mid \varepsilon$

$B \rightarrow Bb \mid \varepsilon$

该语言的一个二义文法是

$S \rightarrow AASb \mid \varepsilon$

$A \rightarrow a \mid \varepsilon$

该语言的一个非二义且非 LR(1)的文法是

$S \rightarrow aaSb \mid A$

$A \rightarrow aBb \mid B$

$B \rightarrow Bb \mid \varepsilon$

分析 先看 LR(1)文法。和 3.38 题相比，$A \rightarrow aaAb$ 用来将两个 a 和一个 b 同时归约，以保证 a 的个数不会超过 b 的个数的两倍。因为 a 的个数可能是奇数，所以有产生式 $A \rightarrow ab$，并且 a 的个数是奇数时，第一步就将 ab 归约成 A。

再看二义文法。用类似于 3.38 题的思想来写一个类似的文法是可能的，那就是

$S \rightarrow aaSb \mid Sb \mid ab \mid \varepsilon$

但是这儿用了完全不同的方法。$S \rightarrow AASb \mid \varepsilon$ 使得 S 推出的 A 的个数是 b 的两倍。由于 A 可以用 a 或 ε 来重写，因此最终 a 的个数不会超过 b 个数的两倍。由于产生式没有限制一定是哪些 A 重写成 a，哪些 A 重写成 ε，因此该文法是二义的。

最后看非二义且非 LR(1)的文法。该文法的特点是，在空归约后，先做若干次一个 b 的归约，然后可能做一次一个 a 和一个 b 的归约，最后做若干次两个 a 和一个 b 的归约（和前面一样，此处只关心终结符的归约，而忽略了产生式右部非终结符的归约）。问题是，向前看一个符号无法确定究竟先做多少次一个 b 的归约才是合适的。

***3.41** 为语言

$L = \{ w \mid w \in (a \mid b)^* $并且在 w 的任何前缀中，a 的个数不少于 b 的个数$\}$

写三个文法，它们分别是 LR(1)的、二义的和非二义且非 LR(1)的。

答 该语言的一个 LR(1)文法是

$S \rightarrow aS \mid aYbS \mid \varepsilon$

$Y \rightarrow aYbY \mid \varepsilon$

该语言的一个二义文法是

$S \rightarrow aS \mid aSbS \mid \varepsilon$

该语言的一个非二义且非 LR(1)的文法是

$S \rightarrow aS \mid XS \mid \varepsilon$

$X \rightarrow aYb$

$Y \rightarrow aYbY \mid \varepsilon$

分析 先对句子的特点进行分析。一个句子可以分解成若干个子句，每个子句都属于语言 L。这样的子句有两类，一类是单个的 a；另一类是配对的 a 和 b 的串(若把 a 看成左括号，把 b 看成右括号，就相当于配对括号串)。另外，这两类子句的任意连接，得到的串都是语言 L 的句子(严格证明见题 3.42)。

上面的 LR(1)文法就是依据这样的观点。从 $S \rightarrow aS \mid \varepsilon$ 知道单个 a 是 L 的句子，从 $S \rightarrow aYbS \mid \varepsilon$知道配对的 a 和 b 的串也是 L 的句子(并不难证明 Y 推出配对的 a 和 b 的串，从而 aYb 也推出配对的 a 和 b 的串)。这三个产生式放在一起表示它们的任意次连接仍然是 L 的句子。

也许有人会问，把 $S \rightarrow aYbS$ 改成 $S \rightarrow YS$ 不是更简洁一些吗？但是这会引起二义，例如句型 $aabbabS$ 中的 $aabbab$ 可以看成是由 $S \Rightarrow YS$ 中 Y 推出的；也可以看成是由 $S \Rightarrow YS \Rightarrow YYS$ 中两个 Y 推出的，其中前一个 Y 推出 $aabb$，后一个 Y 推出 ab。而采用产生式 $S \rightarrow aYbS$ 则限定 aYb 推出的 a 和 b 的配对串一定是第一个 a 和最后一个 b 配对。因此句型 $aabbabS$ 只有一种最左推导。

$S \Rightarrow aYbS \Rightarrow aaYbYbS \Rightarrow aabYbS \Rightarrow aabbS \Rightarrow aabbaYbS \Rightarrow aabbabS$

和 LR(1)文法相比，上面的二义文法是基于对句子结构的同样认识写出来的。但是产生式 $S \rightarrow aSbS$ 右部的前一个 S[在 LR(1)文法中用的是 Y]的使用，使得像句子 aab 的 b 也可以选择和前面的那个 a 配对，因而出现了二义。句子 aab 的两棵语法树如图 3.18 所示。

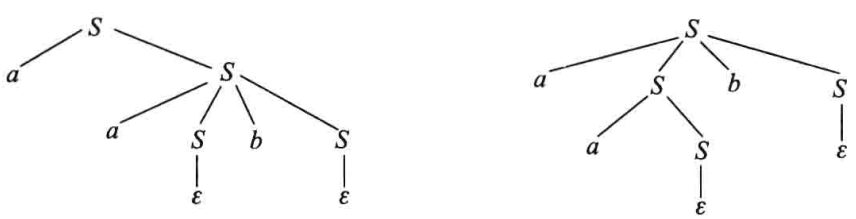


图 3.18 句子 aab 的两棵语法树

和 LR(1)文法及二义文法相比,上面的非二义且非 LR(1)的文法也是基于对句子结构的同样认识写出来的。和 LR(1)文法相比,它多了一步 aYb 到 X 的归约。这步归约导致该文法不是 LR(1)的。例如对于句型 $aaaYba\cdots$,当 $aaaYb$ 在分析栈中,面临下一个 a 时,LR 分析不知道是应该把栈顶的 aYb 归约成 X 呢还是把这个 aYb 看成是 $aYbY$ 的前缀,因而移进 a。如果句型是 $aaaYba$,那么 aYb 应该归约成 X;如果句型是 $aaaYbabb$,那么应该移进 a。

***3.42** 证明

$$S \to aS \mid aYbS \mid \varepsilon$$

$$Y \to aYbY \mid \varepsilon$$

是语言

$$L = \{ w \mid w \in (a \mid b)^* \text{并且在 } w \text{ 的任何前缀中,} a \text{ 的个数不少于 } b \text{ 的个数}\}$$

的一个文法。

答 第一步先证明

$$Y \to aYbY \mid \varepsilon$$

是语言:

$$L' = \{ w \mid w \in (a \mid b)^*,$$

w 中 a 和 b 的个数相等,并且在 w 的任何前缀中,a 的个数不少于 b 的个数}

的一个文法。

首先按推导的步数进行归纳,证明 Y 推出的终结符串都属于语言 L'。对于归纳基,可以看出,从 Y 经过一步推导能得到的终结符串仅有空串一种情况,显然空串属于 L'。

假定所有少于 n 步的推导都能得到的终结符串都属于 L',现在考虑 n 步最左推导。该推导必定是下面这种形式:

$$Y \Rightarrow aYbY \Rightarrow \cdots \Rightarrow aw_1bY \Rightarrow \cdots \Rightarrow aw_1bw_2$$

由于从 Y 到 w_1 和 w_2 的推导都少于 n 步,那么由归纳假设,w_1 和 w_2 都属于 L'。不难看出,aw_1bw_2 也属于 L'。这就证明了从 Y 推出的终结符串都属于 L'。

下面证明,L'的任何句子都可以从 Y 推出。这时按串长进行归纳。对于归纳基,空串是可以从 Y 推出的。

现在假定长度小于 $2n(n \geqslant 1)$ 的 L'的句子都可以从 Y 推出,考虑长度为 $2n$ 的 L'的句子 w。可以肯定,w 一定由 a 开始,令 aw_1b 是有相同个数的 a 和 b 的最短前缀,那么 w 可以写成 aw_1bw_2,其中 w_1 和 w_2 都属于 L',且长度都小于 $2n$。由归纳假设,它们都可以从 Y 推出,因此可以得到 w 的推导:

$$Y \Rightarrow aYbY \Rightarrow \cdots \Rightarrow aw_1bY \Rightarrow \cdots \Rightarrow aw_1bw_2 = w$$

即 w 可以从 Y 推出。这就证明了 L'的任何句子都可以从 Y 推出。

第二步证明题目所给文法是语言 L 的一个文法。同样需要从两个方面去证明。

首先按推导的步数进行归纳,证明 S 推出的终结符串都属于语言 L。对于归纳基,可以看

出，从 S 经过一步推导能得到的终结符串仅有空串一种情况，显然空串属于 L。

假定所有少于 n 步的推导能得到的终结符串都属于 L，现在考虑 n 步最左推导。该推导必定是下面两种形式之一：

$$S \Rightarrow aS \Rightarrow \cdots \Rightarrow aw_1$$

$$S \Rightarrow aYbS \Rightarrow \cdots \Rightarrow aw_2bS \Rightarrow \cdots \Rightarrow aw_2bw_3$$

由于从 S 到 w_1 和 w_3 的推导都少于 n 步，那么由归纳假设，w_1 和 w_3 都属于 L。由前面的证明可知，w_2 属于 L'，不难证明 aw_1 和 $aw_2\,bw_3$ 都属于 L。这就证明了从 S 推出的终结符串都属于 L。

下面证明，L 的任何句子都可以从 S 推出。这时按串长进行归纳。对于归纳基，空串是可以从 S 推出的。

现在假定长度小于 $n(n \geqslant 1)$ 的 L 的句子都可以从 S 推出，考虑长度为 n 的 L 的句子 w。在此寻找 w 中有相同个数的 a 和 b 的非空最短前缀 aw_2b。

若存在这样的前缀，那么在 aw_2b 中，这前后的 a 和 b 一定配对，否则 aw_2b 不是最短前缀。另外，w_2 属于 L'。因此，w 可以写成 $aw_2\,bw_3$ 的形式，其中 w_3 属于 L 且长度小于 n。由归纳假设，w_3 可以从 S 推出。因此可以得到 w 的推导如下：

$$S \Rightarrow aYbS \Rightarrow \cdots \Rightarrow aw_2bS \Rightarrow \cdots \Rightarrow aw_2bw_3 = w$$

即 w 可以从 S 推出。

若 w 中不存在有相同个数的 a 和 b 的非空前缀，那么在任何非空前缀中，a 的个数都大于 b 的个数。因此 w 可以写成 aw_1 的形式，其中 w_1 属于 L 且长度小于 n。由归纳假设，w_1 可以从 S 推出。因此可以得到 w 的推导如下：

$$S \Rightarrow aS \Rightarrow \cdots \Rightarrow aw_1 = w$$

即 w 可以从 S 推出。

这就证明了 L 的任何句子都可以从 S 推出。

分析 编译原理课程并不要求掌握这样的证明技术。但是，当为一个语言设计文法时能用这样的方式去思考问题，设计出正确文法的可能性就大得多。

从题 3.41 和本题的文法和证明，可以启发为题 3.7 的语言“a 和 b 的个数相等的所有串”写出另一个非二义的文法：

$$S \rightarrow aAbS \mid bBaS \mid \varepsilon$$

$$A \rightarrow aAbA \mid \varepsilon$$

$$B \rightarrow bBaB \mid \varepsilon$$

读者应该能证明该文法的正确性。

***3.43** 各类文法能定义的语言集合是不一样的。例如，SLR(1)文法能定义的语言集合是 LR(1)文法能定义语言集合的一个真子集。请你根据文法类能定义的语言集合的包含关系给出各种 LL 和 LR 文法类的层次关系。

答 层次关系如图 3.19 所示，在此不加证明。

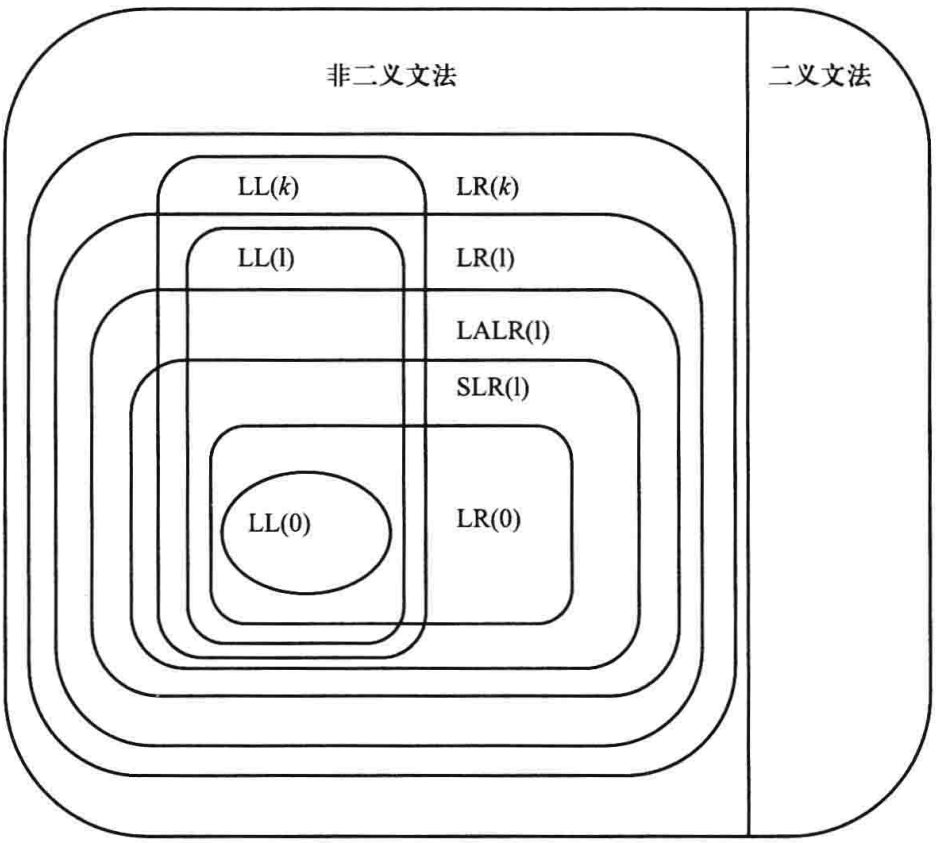


图 3.19　文法类的层次图

*3.44　说明每个 SLR(1)文法都是 LALR(1)文法。

答　此问题的核心有两点：

1. 对一个文法来说，构造 SLR(1)分析表过程中得到的接受该文法活前缀的 DFA，和构造 LALR(1)分析表过程中得到的接受该文法活前缀的 DFA 是同构的。

2. 在 LALR(1)任何一个项目集中，其归约项目的向前搜索符集合是该项目产生式的左部非终结符的后继符号集合的子集。

有了第一点，一个文法的 SLR(1)分析表和 LALR(1)分析表的大小是一样的，动作表中转移动作是一样的，转移表是完全一样的。

唯一有区别的是动作表中的归约动作，根据上面第二点，LALR(1)分析表的归约动作和 SLR(1)分析表的没有矛盾，但是可能没有 SLR(1)分析表的那么多。

这样，如果一个文法的 SLR(1)分析表没有动作冲突，那么它的 LALR(1)分析表一定也没有动作冲突。所以，每个 SLR(1)文法都是 LALR(1)文法。

教材上一般都有例子指出，存在不是 SLR(1)文法的 LALR(1)文法，即 SLR(1)文法集合是 LALR(1)文法集合的真子集。

3.45　一个 C 语言的源程序文件如下：

```
long gcd(long p, long q) {
```

```
    if (p%q == 0)
        return q;
    else
        return gcd(q, p%q);
}
```

基于 LALR(1)方法的一个编译器的报错情况如下。

如果缺少第 1 行中的逗号,编译器扫描第 1 行时会报告 expected ';' , ',' or ')' before 'long'。如果缺少第 2 行中的右括号,编译器扫描第 3 行时会报告 expected ')' before 'return'。这两个示例表明 LALR(1)方法能及时发现错误,且不会把出错点后面的符号移进分析栈(活前缀性质)。

如果第 2 行的 if 误写成 fi,编译器扫描第 3 行时会报告 expected ';' before 'return',扫描第 4 行时报告'else' without a previous 'if'。此时是否违反了活前缀性质。

答 编译器把 fi 看成标识符,把整个 fi (p%q == 0)看成是函数调用,因此没有违反活前缀性质,并且期望之后有分号。报告第 4 行的'else' without a previous 'if'是因为未能为 else 找到配对的 if。

分析 在编译源程序时,一定要把程序员观点下的程序语法错误(上例中是 if 拼写错)和编译器观点下的程序语法错误(上例中期望分号和未找到配对的 if)区分开,否则有时不能很快找到程序错误。

第4章　语法制导的翻译

一、主要知识点

主教材第4章主要应掌握下面一些内容。

1. 语义规则的两种描述方法:语法制导的定义和翻译方案。语法制导的定义没有指明语义规则的计算次序,而翻译方案显式地给出语义规则(或称语义动作)的计算次序和位置。

2. 综合属性定义(S属性定义)和L属性定义。

3. 设计一些简单问题的语法制导定义和翻译方案。

4. 语义规则的三种计算方法:分析树方法、基于规则的方法和忽略规则的方法。部分习题的分析中会指明它们所用的是哪种方法。

5. S属性的自下而上计算。它一边进行语法分析,一边进行属性计算。由于只能有综合属性,故它是忽略规则的方法。

6. L属性的自上而下计算。它也是一边进行语法分析,一边进行属性计算。由于它根据各规则的特点来决定其计算时机,因此它是基于规则的方法。

7. L属性的自下而上计算。它介绍的是,什么样的继承属性可以在自下而上的分析过程中完成,因此属于忽略规则的范畴。

8. 递归计算。它先进行语法分析,然后进行属性计算。由于它根据对语义规则的分析来安排语义规则的计算次序,因此它是基于规则的方法。主教材更新后,这部分内容被略去,本章保留了这种方式的一个习题,见习题4.24。

本章的重点和难点是设计语法制导定义和翻译方案。这种设计可看成是一种程序设计,是一种事件驱动方式的程序设计,因此它比一般的程序设计要难得多。这里的事件是指句子中各种语法结构的识别。最难的情况是必须使用继承属性的情况。

二、习题精选与解析

4.1　简单台式计算器的语法制导定义如下:

$L \rightarrow E$ **n**	print ($E.val$)
$E \rightarrow E_1+T$	$E.\ val = E_1.val+T.val$
$E \rightarrow T$	$E.val = T.val$

$T \rightarrow T_1 * F$ $\quad T.\ val = T_1.\ val \times F.\ val$

$T \rightarrow F$ $\quad T.\ val = F.\ val$

$F \rightarrow (E)$ $\quad F.\ val = E.\ val$

$F \rightarrow \mathbf{digit}$ $\quad F.\ val = \mathbf{digit}.\ lexval$

为输入表达式(4 *7+1) *2 构造注释分析树。

答 该表达式的注释分析树见图 4.1。

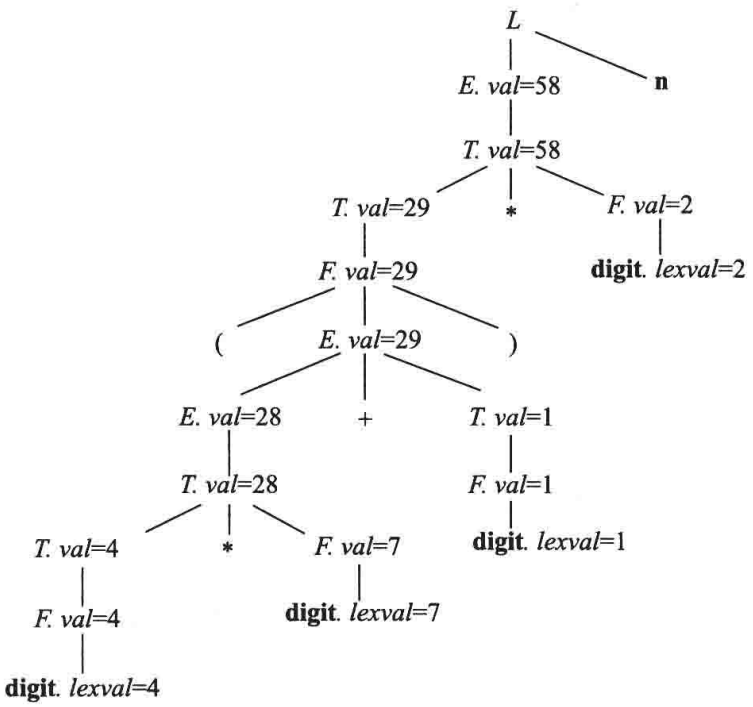


图 4.1 (4 *7+1) *2 的注释分析树

分析 在该语法制导定义中,除了 L 的属性是虚拟属性外,其余文法符号的属性都是综合属性。因此,在自下而上地构造分析树的过程中,同时完成文法符号属性的注释并不困难。

4.2 构造表达式语法树(有些教材上称为抽象语法树)的翻译方案如下:

$E \rightarrow T \{R.i = T.nptr;\} R \{E.nptr = R.s;\}$

$R \rightarrow + T \{R_1.i = mkNode('+', R.i, T.nptr);\} R_1 \{R.s = R_1.s;\}$

$R \rightarrow - T \{R_1.i = mkNode('-', R.i, T.nptr);\} R_1 \{R.s = R_1.s;\}$

$R \rightarrow \varepsilon \{R.s = R.i;\}$

$T \rightarrow (E) \{T.nptr = E.nptr;\}$

$T \rightarrow \mathbf{id} \{T.nptr = mkLeaf(\mathbf{id}, \mathbf{id}.entry);\}$

$T \rightarrow \mathbf{num} \{T.nptr = mkLeaf(\mathbf{num}, \mathbf{num}.val);\}$

其中,属性 $nptr$、i 和 s 是语法树结点的指针,$mkNode$ 和 $mkLeaf$ 分别是构造语法树内部结点和叶结点的函数。为表达式 $(a) + b$ 构造分析树和语法树。

答 见图 4.2。分析树由短虚线表示，生成的语法树用实线表示，长虚线表示文法符号的属性所指向的语法树结点。分析树上最右边 R 的 s 属性指向所产生的语法树。为使得图形清晰，图 4.2 省略了最右边 R 的 s 属性向它的父结点 R 的 s 属性的复写，及最后向根结点 E 的 *nptr* 属性的复写。

另外，图中左边文法符号的属性值很多都是指向左下角的语法树结点的指针。这些属性值的复写次序是这样：从 *T. nptr*（下面的 *T*）到 *R. i*（下面的 *R*），到 *R. s*（下面的 *R*），到 *E. nptr*（下面的 *E*），到 *T. nptr*（上面的 *T*），最后到 *R. i*（中间的 *R*）。

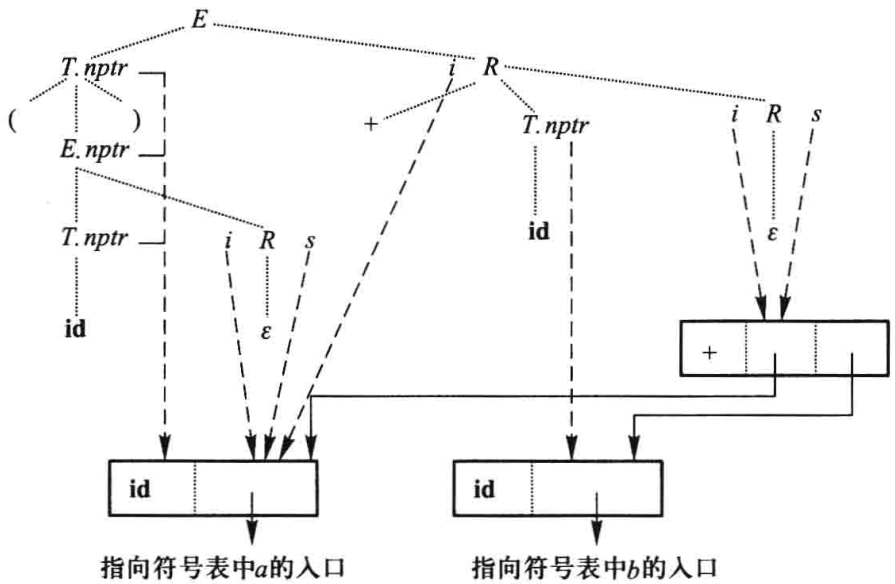


图 4.2 （*a*）+ *b* 的分析树和语法树

分析 该翻译方案中 R 有继承属性。因此出现了语义动作在文法符号之间的情况。这个例子根据翻译方案自上而下和自左向右地构造分析树。碰到产生式右部的文法符号时，则在分析树上加结点；而碰到语义动作时，执行该动作，完成语法树结点的建立或语法树结点指针值的复写。

另外，对于这个问题，文法左递归并且只有综合属性的语法制导定义如下。这个语法制导定义比上面的翻译方案要直观得多。上面的继承属性是因为对下面的文法消除左递归而产生的。

$E \rightarrow E_1 + T$	$E.nptr = mkNode('+', E_1.nptr, T.nptr)$
$E \rightarrow E_1 - T$	$E.nptr = mkNode('-', E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr = T.nptr$
$T \rightarrow (E)$	$T.nptr = E.nptr$
$T \rightarrow \mathbf{id}$	$T.nptr = mkLeaf(\mathbf{id}, \mathbf{id}.entry)$
$T \rightarrow \mathbf{num}$	$T.nptr = mkLeaf(\mathbf{num}, \mathbf{num}.val)$

4.3 为文法：

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

(a) 写一个语法制导定义，它输出括号的对数。

(b)写一个语法制导定义，它输出括号嵌套的最大深度。

答 (a) 拓广文法，加上新的开始符号 S'和新的产生式 $S' \rightarrow S$。输出括号对数的语法制导定义如下：

$S' \rightarrow S$	print ($S.num$)
$S \rightarrow (L)$	$S.num = L.num + 1$
$S \rightarrow a$	$S.num = 0$
$L \rightarrow L_1, S$	$L.num = L_1.num + S.num$
$L \rightarrow S$	$L.num = S.num$

(b) 像上面那样拓广文法，输出括号嵌套最大深度的语法制导定义如下：

$S' \rightarrow S$	print ($S.max$)
$S \rightarrow (L)$	$S.max = L.max + 1$
$S \rightarrow a$	$S.max = 0$
$L \rightarrow L_1, S$	$L.max =$ if ($L_1.max > S.max$) $L_1.max$ else $S.max$
$L \rightarrow S$	$L.max = S.max$

分析 从本题开始，给出一些只有综合属性的语法制导定义的例子，并且先从简单的抽象例子开始。

在写语法制导定义前，首先分析清楚需要为文法符号定义哪些属性，然后看这些属性的值是否可以由文法符号本身所推出的串决定。若是，则应该只用综合属性就能解决。

对于(a)，非终结符 S 和 L 需要表示它们所含括号对数的属性，而它们自身所含的括号对数显然是能从它们推出的串确定的，因此应该只用综合属性就能解决。(b)的情况类似。

另外要注意的是，拓广文法的目的是为了使打印动作唯一并且打印的值是括号总对数[(a)小题]或括号嵌套最大深度[(b)小题]。否则，不管打印语句作为哪个产生式的语义规则，都有可能出现多次执行打印动作而不知哪个值是所要的结果的情况。

4.4 下面是产生字母表 $\Sigma = \{0, 1, 2\}$ 上数字串的一个文法：

$S \rightarrow DSD \mid 2$

$D \rightarrow 0 \mid 1$

写一个语法制导定义，它判断一个句子是否为回文数(一个数字串，从左向右读和从右向左读都一样时，称它为回文数)，并输出该判断。

答 判断是否为回文数的语法制导定义如下：

$S' \rightarrow S$	print($S.val$)
$S \rightarrow D_1 S_1 D_2$	$S.val = (D_1.val == D_2.val)\ \&\&\ S_1.val$
$S \rightarrow 2$	$S.val = true$
$D \rightarrow 0$	$D.val = 0$
$D \rightarrow 1$	$D.val = 1$

分析 首先需要说明一点,每个句子中间都是数字 2 是为了保证这个文法是 LR(1)文法。若文法改成:

$S \rightarrow DSD \mid \varepsilon$

$D \rightarrow 0 \mid 1$

则它是一个非二义非 LR(1)文法。

在该语法制导定义中,S 的 val 属性是布尔型,D 的 val 属性是整型。该定义的做法是,从中间向两头逐对地比较数字,一旦出现不相同情况,$S.val$ 的值就被置为假。

4.5 算符 θ 作用于表达式 $e_1, e_2, \cdots, e_k$ 的前缀形式是 $\theta\ p_1\ p_2 \cdots\ p_k$,其中 p_i 是 e_i 的前缀形式。

(a)写出 $a * -(b + c)$ 的前缀形式(其中"-"是一元减运算符)。

(b)给出把中缀表达式翻译成前缀形式的翻译方案,中缀表达式的文法如下:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow \mathbf{id} \mid (E)$

答 (a) 表达式 $a *-(b + c)$ 的前缀形式是 $*a\ -+\ b\ c$。

(b) 用综合属性 code(串类型)来保存产生的前缀表达式,那么产生前缀表达式的翻译方案如下,其中算符"‖"表示串的连接运算。

$E \rightarrow E_1 + T$	$\{E.code = '+' \parallel E_1.code \parallel T.code;\}$
$E \rightarrow T$	$\{E.code = T.code;\}$
$T \rightarrow T_1 * F$	$\{T.code = '*' \parallel T_1.code \parallel F.code;\}$
$T \rightarrow F$	$\{T.code = F.code;\}$
$F \rightarrow \mathbf{id}$	$\{F.code = \mathbf{id}.place;\}$
$F \rightarrow (E)$	$\{F.code = E.code;\}$

分析 因为是用 $code$ 属性来保存前缀表达式,这个翻译方案就很容易写。但是若希望一边分析一边打印前缀表达式,这几乎是不可能的。以产生式 $E \rightarrow E_1 + T$ 为例来说明。对符合该产生式的输入串,首先应该打印"+"。但是"+"出现在 E_1 之后,因此必须先分析 E_1 后才有可能看见输入串中的"+",而在分析 E_1 时,对应它的前缀表达式应该输出,这样它们在"+"之前输出,也就是说"+"不可能最先输出。

若要避免串类型作为属性类型,比较实际的做法是,先生成语法树,然后对语法树进行遍历,产生前缀表达式。另外,有了语法树当然也不难产生其他形式的中间表示了。

* **4.6** 用 S 的综合属性 *val* 给出下面文法中 S 产生的二进制数的值。例如，输入 101.101 时，$S.val = 5.625$。要求仅用综合属性来计算 $S.val$。

$S \rightarrow L.L \mid L$

$L \rightarrow LB \mid B$

$B \rightarrow 0 \mid 1$

答 给文法符号 B、L 和 S 以综合属性 *val*，另外再给 L 一个表示其长度的综合属性 *length*。所求的语法制导定义如下：

$S \rightarrow L_1.L_2$	$S.val = L_1.val + L_2.val / 2^{L_2.length}$
$S \rightarrow L$	$S.val = L.val$
$L \rightarrow L_1B$	$L.val = L_1.val \times 2 + B.val$; $L.length = L_1.length + 1$
$L \rightarrow B$	$L.val = B.val$; $L.length = 1$
$B \rightarrow 0$	$B.val = 0$
$B \rightarrow 1$	$B.val = 1$

分析 在这个问题中比较困难的是有小数点的情况。在 $S \rightarrow L_1.L_2$ 中，L_1 和 L_2 分处小数点的两侧，为了避免使用继承属性，在此对 L_1 和 L_2 使用同样的计算方法，而在把它们的值相加之前，将 L_2 的值除以 $2^{L_2.length}$。要想避免这样的指数运算和除法，就要对 L_1 和 L_2 分别使用不同的方式求值。这时需要用到继承属性，因为如果 L 没有继承属性的话，到了 L 的产生式中就无法区分该 L 究竟是出现在小数点的左边还是右边。习题 4.19 将给出使用继承属性的解决办法。

4.7 下面的文法定义语言 $L = \{ a^n b^n c^m \mid m, n \geqslant 1 \}$。写一个语法制导定义，其语义规则的作用是：对不属于语言 L 的子集 $L_1 = \{ a^n b^n c^n \mid n \geqslant 1 \}$ 的句子，打印出错信息。

$S \rightarrow DC$

$D \rightarrow aDb \mid ab$

$C \rightarrow Cc \mid c$

答 语法制导的定义如下：

$S \rightarrow DC$	if ($D.length$! = $C.length$) print ("error")
$D \rightarrow ab$	$D.length = 1$
$D \rightarrow aD_1b$	$D.length = D_1.length + 1$
$C \rightarrow c$	$C.length = 1$
$C \rightarrow C_1c$	$C.length = C_1.length + 1$

D 的综合属性 *length* 用来记住 a 和 b 的个数，C 的综合属性 *length* 用来记住 c 的个数。

4.8 为下面文法写一个语法制导的定义，它完成一个句子的 while-do 最大嵌套层次的计算并输出计算结果。

$S \rightarrow E$

$E \rightarrow$ **while** E **do** E | **id** := E | $E + E$ | **id** | (E)

答 语法制导的定义如下：

$S \rightarrow E$	print($S.loop$)
$E \rightarrow$ **while** E_1 **do** E_2	$E.loop = \max(E_1.loop, E_2.loop) + 1$
$E \rightarrow$ **id** := E	$E.loop = E_1.loop$
$E \rightarrow E_1 + E_2$	$E.loop = \max(E_1.loop, E_2.loop)$
$E \rightarrow$ **id**	$E.loop = 0$
$E \rightarrow (E_1)$	$E.loop = E_1.loop$

非终结符的综合属性 *loop* 用来记住其 **while-do** 的最大嵌套层次。

4.9 下列文法产生由“+”作用于整常数或实常数的表达式。两个整数相加时，结果是整型，否则是实型。

$E \rightarrow E+T \mid T$

$T \rightarrow$ **num. num** | **num**

(a)给出确定每个子表达式类型的语法制导定义。

(b)扩充(a)的语法制导定义，既要确定类型，又把表达式翻译成后缀表示。使用一元算符 **inttoreal** 把整数变成等价的实数，使得后缀式中“+”的两个对象有同样的类型。

答 (a) 给文法符号 E 和 T 以综合属性 *type*，用来表示它们的类型，类型值分别用 *int* 和 *real* 来表示。所求语法制导定义如下：

$E \rightarrow E_1 + T$	if ($E_1.type$ == *int* && $T.type$ == *int*)
	  $E.type$ = *int*
	else
	  $E.type$ = *real*

$E \rightarrow T$	$E.type = T.type$
$T \rightarrow$ **num. num**	$T.type$ = *real*
$T \rightarrow$ **num**	$T.type$ = *int*

(b)下面的语法制导定义将表达式的后缀表示打印输出，其中 *lexeme* 属性表示单词的拼写：

$E \rightarrow E_1 + T$	if ($E_1.type$ == *real* && $T.type$ == *int*) {
	  $E.type$ = *real*; print ($T.lexeme$); print ('**inttoreal**')
	} else if ($E_1.type$ == *int* && $T.type$ == *real*) {
	  $E.type$ = *real*; print ('**inttoreal**'); print ($T.lexeme$)
	} else {
	  $E.type = E_1.type$; print ($T.lexeme$)
	}

	print ('+')
$E \rightarrow T$	$E.type = T.type$; print ($T.lexeme$)
$T \rightarrow$ **num. num**	$T.type = real$; $T.lexeme =$ **num. num**. $lexeme$
$T \rightarrow$ **num**	$T.type = int$; $T.lexeme =$ **num**. $lexeme$

分析 类型检查很容易定义,在此主要分析后缀表达式的输出。这里没有在 **num** 和 **num. num** 向 T 归约的时候输出该运算对象。如果那样做的话,带来的问题是,在 $E_1 + T$ 向 E 归约时做插入转换算符的动作有可能太晚,因为这时两个运算对象都已输出,而若要为第一个运算对象插入转换算符,则已经来不及。这个语法制导的定义推迟运算对象的输出,放在 T 向 E 归约的时候完成,以克服上面所说的困难。

另一点要注意的是,在 $E_1 + T$ 向 E 归约时,该加法运算的第一个运算对象已经输出,所以上面 $E \rightarrow E_1 + T$ 的语义规则没有输出 E_1 运算对象的动作。

4.10 由下列文法产生的表达式包括赋值表达式

$S \rightarrow E$

$E \rightarrow E=E \mid E+E \mid (E) \mid$ **id**

表达式的语义和 C 语言的一样,即 $b = c$ 是把 c 的值赋给 b 的赋值表达式,而且 $a=(b=c)$ 把 c 的值赋给 b,然后再赋给 a。构造一个语法制导定义,它检查赋值表达式的左部是否为左值。

答 用综合属性 val 表示 E 是否为左值表达式,其取值可以是 $lvalue$ 和 $rvalue$,分别表示左值表达式和右值表达式。那么所求语法制导定义如下:

$S \rightarrow E$	
$E \rightarrow E_1 = E_2$	if ($E_1.val$ == $rvalue$) print ("error"); $E.val = rvalue$
$E \rightarrow E_1 + E_2$	$E.val = rvalue$
$E \rightarrow (E_1)$	$E.val = E_1.val$
$E \rightarrow$ **id**	$E.val = lvalue$

分析 这里仍然用综合属性来解决这个问题。在赋值表达式 $E_1 = E_2$ 中,E_1 必须是左值表达式。E_1 是否为左值表达式,从 E_1 本身的结构是可以判断出来的,因此只用综合属性肯定可以求解。

另外,如果 E 是左值表达式,那么(E)仍然是左值表达式,这是 C 语言的规定。

4.11 在 C 语言中,3++和(id+id)++这样的表达式被编译时,编译器都会报告如下的错误:

invalid lvalue in increment

现有如下简化的 C 语言表达式文法:

$E \rightarrow E+E \mid (E) \mid E{+}{+} \mid$ **id** $\mid$ **num**

请写一个语法制导定义或翻译方案,它检查++的运算对象是否合法。

答 给非终结符 E 一个综合属性 val,其值可取 $lvalue$ 或 $rvalue$,分别表示 E 是左值表达式和

右值表达式,那么语法制导定义如下(无输出则表示无错):

$E' \rightarrow E$

$E \rightarrow E_1 + E_2$ $\quad E.val = rvalue$

$E \rightarrow (E_1)$ $\quad E.val = E_1.val$

$E \rightarrow E_1{+}{+}$ $\quad$ if ($E_1.val == rvalue$) printf("invalid lvalue in increment");
$E.val = rvalue$

$E \rightarrow$ **id** $\quad E.val = lvalue$

$E \rightarrow$ **num** $\quad E.val = rvalue$

分析 本题和4.10题没有多少区别,都是指某个运算的某个对象必须是左值表达式。

***4.12** 给出把中缀表达式翻译成没有冗余括号的中缀表达式的语法制导定义。例如,因为+和*是左结合,((a *(b+c)) *(d))可以重写成 a *(b+c) *d。

答 用 *code* 属性表示没有冗余括号的代码,用 *op* 属性表示算符,它可以取值 *plus*、*times* 和 *id*,那么所求语法制导定义如下。注意,"‖"在本例中既出现在逻辑运算中,也作为字符串之间的连接运算符。

$S' \rightarrow E$ $\quad$ print ($E.code$)

$E \rightarrow E_1 + T$ $\quad$ if ($T.op == plus$)
$E.code = E_1.code \parallel$ "+" $\parallel$ "(" $\parallel T.code \parallel$ ")"
else
$E.code = E_1.code \parallel$ "+" $\parallel T.code$;
$E.op = plus$

$E \rightarrow T$ $\quad E.code = T.code$; $\quad E.op = T.op$

$T \rightarrow T_1 * F$ $\quad$ if ($F.op == plus \parallel F.op == times$)
if ($T_1.op == plus$)
$T.code =$ "(" $\parallel T_1.code \parallel$ ")" $\parallel$ "*" $\parallel$ "(" $\parallel F.code \parallel$ ")"
else
$T.code = T_1.code \parallel$ "*" $\parallel$ "(" $\parallel F.code \parallel$ ")"
else if ($T_1.op == plus$)
$T.code =$ "(" $\parallel T_1.code \parallel$ ")" $\parallel$ "*" $\parallel F.code$
else
$T.code = T_1.code \parallel$ "*" $\parallel F.code$;
$T.op = times$

$T \rightarrow F$ $\quad T.code = F.code$; $\quad T.op = F.op$

$F \rightarrow$ **id** $\quad F.code =$ **id**$.lexeme$; $\quad F.op = id$

$F \rightarrow (E)$ $\quad F.code = E.code$; $\quad F.op = E.op$

分析 消除冗余括号有两种办法，其一是先把表达式的括号都去掉，然后在必要的地方再加括号。其二是只去掉表达式中的冗余括号，保留必要的括号。

用第二种办法的困难是，在产生式 $F \rightarrow (E)$ 的语义规则中要决定是否保留这对括号。但是它不是仅由 E 推出的串就能决定的，还必须有 F 的上下文信息，因此必定使用继承属性。本题不用这种办法，把它留到习题 4.16。

对于第一种办法，可以像 4.2 题那样先构造表达式的语法树，语法树是没有括号的。然后用一个过程来遍历语法树，把表达式还原成线性表示，并加上必要的括号。这里采用另一种做法，在语法分析的同时，去掉原表达式的所有括号，直接构造表达式的线性表示，同时加上必要的括号。加括号是对子表达式加括号，例如 $T_1 * F$ 中的 T_1 和 F。根据表达式的算符、子表达式的主算符和该子表达式是作为左运算对象还是右运算对象，就可以决定是否给该子表达式加括号，因此这时不需要使用继承属性。

4.13 为文法

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

(a) 写一个翻译方案，它输出每个 a 的嵌套深度。例如，对于句子 $(a,(a,a))$，输出的结果是 1 2 2。

(b) 写一个翻译方案，它打印出每个 a 在句子中是第几个字符。例如，当句子是 $(a,(a,(a,a),(a)))$ 时，打印的结果是 2 5 8 10 14。

答 (a) 用继承属性 *depth* 表示嵌套深度，所求的翻译方案如下：

$S' \rightarrow \{S.depth = 0;\} S$

$S \rightarrow \{L.depth = S.depth + 1;\} (L)$

$S \rightarrow a \{\text{print}(S.depth);\}$

$L \rightarrow \{L_1.depth = L.depth;\} L_1, \{S.depth = L.depth;\} S$

$L \rightarrow \{S.depth = L.depth;\} S$

(b) 给文法符号 S 和 L 一个继承属性 *in* 和一个综合属性 *out*，分别表示在句子中，该文法符号推出的字符序列的前面已经有多少个字符，以及该文法符号推出的字符序列的最后一个字符在句子中是第几个字符。那么所求的翻译方案如下：

$S' \rightarrow \{S.in = 0;\} S$

$S \rightarrow \{L.in = S.in + 1;\} (L) \{S.out = L.out + 1;\}$

$S \rightarrow a \{S.out = S.in + 1; \text{print}(S.out);\}$

$L \rightarrow \{L_1.in = L.in;\} L_1, \{S.in = L_1.out + 1;\} S \{L.out = S.out;\}$

$L \rightarrow \{S.in = L.in;\} S \{L.out = S.out;\}$

分析 从本题开始给出一些需要继承属性的翻译方案(或语法制导定义)的例子，先从简单的抽象例子开始。

(a) 首先判断是否一定要用继承属性。由于 a 的嵌套深度不是由 a 本身能决定的,所以一定要用继承属性。另外,a 的嵌套深度从 a 左边的部分足以判断,因此可以用 L 属性定义解决。

(b) 同样,字符 a 在句子中的位置也不可能是由 a 本身决定的,所以一定要用继承属性。另外,a 在句子中的位置从 a 左边的部分足以判断,因此可以用 L 属性定义解决。

可以用另一个综合属性 *total* 来代替 *out*,它的含义是由 S 或 L 推出的字符总数(这样,*out* 就是 *in* +*total*)。那么翻译方案改成:

$S' \rightarrow \{S.in = 0;\}\ S$

$S \rightarrow \{L.in = S.in + 1;\}\ (L)\ \{S.total = L.total + 2;\}$

$S \rightarrow a\ \{S.total = 1; print(S.in + 1);\}$

$L \rightarrow \{L_1.in = L.in;\}\ L_1,\ \{S.in = L.in + L_1.total + 1;\}$
$S\ \{L.total = L_1.total + 1 + S.total;\}$

$L \rightarrow \{S.in = L.in;\}\ S\ \{L.total = S.total;\}$

4.14 程序的文法如下:

$P \rightarrow D$

$D \rightarrow D\ ;\ D \mid \textbf{id}\ :\ T \mid \textbf{proc id}\ ;\ D\ ;\ S$

(a) 写一个语法制导定义,打印该程序一共声明了多少个 **id**。

(b) 写一个翻译方案,打印该程序每个变量 **id** 的嵌套深度。

答 (a) 用综合属性 i 表示 D 中声明的 **id** 的个数。

$P \rightarrow D$	print($D.i$)
$D \rightarrow D_1\ ;\ D_2$	$D.i = D_1.i + D_2.i$
$D \rightarrow \textbf{id}\ :\ T$	$D.i = 1$
$D \rightarrow \textbf{proc id}\ ;\ D_1\ ;\ S$	$D.i = D_1.i + 1$

(b) 用继承属性 l 表示 D 中声明的变量 **id** 的嵌套深度。

$P \rightarrow \{D.l = 1;\}\ D$

$D \rightarrow \{D_1.l = D.l;\}\ D_1;\ \{D_2.l = D.l;\}\ D_2$

$D \rightarrow \textbf{id}\ :\ T\ \{\text{print}(\textbf{id}.name, D.l);\}$

$D \rightarrow \textbf{proc id}\ ;\ \{D_1.l = D.l + 1;\}\ D_1;\ S$

4.15 改用继承属性完成习题 4.10。

答 用继承属性 *val* 表示,根据 E 的上下文是否限定 E 为左值表达式。其取值可以是 *left* 和 *any*,分别表示只能是左值表达式和可以是任意表达式。那么所求的翻译方案如下:

$S \rightarrow \{E.val = any;\}\ E$

$E \rightarrow \{\text{if}\ (E.val == left)\ \text{print}(\text{"error"});\ E_1.val = left;\}\ E_1 = \{E_2.val = any;\}\ E_2$

$E \rightarrow \{\text{if}\ (E.val == left)\ \text{print}(\text{"error"});\ E_1.val = any;\}\ E_1 + \{E_2.val = any;\}\ E_2$

$E \rightarrow \{\ E_1.val = E.val;\ \}\ (\ E_1)$

$E \rightarrow \mathbf{id}$

分析 对习题 4.10 改用综合属性实现的思路是，从每个表达式本身的结构可以判断它是否为左值表达式，然后在只能出现左值表达式的地方，检查表达式是否为左值。也可以用另一种观点来分析，从 $E \rightarrow E_1 = E_2$ 知道 E_1 必须是左值表达式，然后带着这个限定去检查 E_1 的语法结构，若不是可作为左值的结构则报错。本题用的是后一种观点。也可以这么说，第一种方法是基于自下而上的方法来考察分析树，而第二种方法是基于自上而下的方法考察分析树。

***4.16** 改用继承属性完成习题 4.12。

答 给 E，T 和 F 两个继承属性 *left_op* 和 *right_op*，分别表示它们左右两侧算符的优先级，定义综合属性 *self_op* 表示自身算符的优先级，综合属性 *code* 表示没有冗余括号的代码。

分别用整数 1 和 2 表示加和乘的优先级，用整数 3 表示 **id** 和 (E) 的优先级。另外用 0 表示左右两边没有运算对象需要比较的情况。这些是属性 *self_op* 、*left_op* 和 *right_op* 可取的值，用整数表示是为了简化语义动作的描述。所求语法制导定义如下，它是一个 L 属性定义。

$S' \rightarrow E$	$E.left_op = 0;\ E.right_op = 0;$ print $(E.code)$
$E \rightarrow E_1 + T$	$E_1.left_op = E.left_op;\ E_1.right_op = 1;$
	$T.left_op = 1;\ T.right_op = E.right_op;$
	$E.code = E_1.code \parallel$ "+" $\parallel T.code\ ;\ E.self_op = 1;$
$E \rightarrow T$	$T.left_op = E.left_op;\ T.right_op = E.right_op;$
	$E.code = T.code;\ E.self_op = T.self_op$
$T \rightarrow T_1 * F$	$T_1.left_op = T.left_op;\ T_1.right_op = 2;$
	$F.left_op = 2;\ F.right_op = T.right_op;$
	$T.code = T_1.code \parallel$ "*" $\parallel F.code\ ;\ T.self_op = 2;$
$T \rightarrow F$	$F.left_op = T.left_op;\ F.right_op = T.right_op;$
	$T.code = F.code;\ T.self_op = F.self_op$
$F \rightarrow \mathbf{id}$	$F.code =$ **id**$.lexeme;\quad F.self_op = 3$
$F \rightarrow (E)$	$E.left_op = 0;\ E.right_op = 0;$
	$F.self_op =$
	if $(F.left_op < E.self_op \ \&\&\ E.self_op >= F.right_op)$
	$E.self_op$
	else
	3
	$F.code =$
	if $(F.left_op < E.self_op \ \&\&\ E.self_op >= F.right_op)$
	$E.code$

else

"(" || $E.code$ || ")"

分析 这里的出发点是只去掉表达式中的冗余括号，保留必要的括号，那么在产生式 $F \rightarrow (E)$ 的语义规则中要决定是否保留这对括号。因为它不是仅由 E 推出的串就能决定的，还必须考虑 F 的上下文信息，因此必定使用继承属性。一个括号是否应该保留，是将括号中表达式的主算符（指看成语法树时的那个根算符）的优先级和括号外两边的算符的优先级作比较后才能确定的，所以需要有继承属性 *left_op* 和 *right_op*。

本题的语义规则较多，为使直观上清晰起见，在此给出的是语法制导定义。这是一个 L 属性定义，把它写成翻译方案并不困难。

4.17 语句的文法如下：

$S \rightarrow$ **id** := E | **if** E **then** S | **while** E **do** S | **begin** S ; S **end** | **break**

写一个翻译方案，其语义动作的作用是：若发现 break 语句没有出现在循环语句中，及时报告错误。

答 翻译方案如下：

$S' \rightarrow$ {$S.loop = false$;} S

$S \rightarrow$ **id** := E

$S \rightarrow$ **if** E **then** {$S_1.loop = S.loop$;} S_1

$S \rightarrow$ **while** E **do** {$S_1.loop = true$;} S_1

$S \rightarrow$ **begin** {$S_1.loop = S.loop$;} S_1 ; {$S_2.loop = S.loop$;} S_2 **end**

$S \rightarrow$ **break** {if (! $S.loop$) print("error") ;}

分析 非终结符 S 的属性 *loop* 是一个继承属性，在遇到 break 语句时，若继承属性告知没有出现在循环语句中，则报告错误。

4.18 C 语言和 Pascal 语言的变量声明的文法分别列在下面左右两边：

$D \rightarrow TL$	$D \rightarrow L : T$
$T \rightarrow$ **int**	$L \rightarrow L$, **id**
$T \rightarrow$ **real**	$L \rightarrow$ **id**
$L \rightarrow L$, **id**	$T \rightarrow$ **integer**
$L \rightarrow$ **id**	$T \rightarrow$ **real**

分别为它们构造一个翻译方案，把标识符的类型填入符号表中。

答 C 语言的翻译方案如下，其中 **id** 的 *entry* 属性是符号表条目的指针，过程 *addType* 是将类型信息填入符号表。

$D \rightarrow T$ {$L.in = T.type$;} L

$T \rightarrow$ **int** {$T.type = integer$;}

$T \rightarrow$ **real** { $T.type = real$; }

$L \rightarrow$ { $L_1.in = L.in$; } L_1, **id** { $addType$ (**id**. $entry$, $L.in$); }

$L \rightarrow$ **id** { $addType$ (**id**. $entry$, $L.in$); }

对于 Pascal 语言,先将文法改写如下:

$D \rightarrow$ **id** L

$L \rightarrow$, **id** L | : T

$T \rightarrow$ **integer** | **real**

下面给出改写文法的翻译方案:

$D \rightarrow$ **id** L { $addType$ (**id**. $entry$, $L.type$); }

$L \rightarrow$, **id** L_1 { $L.type = L_1.Type$; $addType$ (**id**. $entry$, $L_1.type$); }

$L \rightarrow$: T { $L.type = T.type$; }

$T \rightarrow$ **integer** { $T.type = integer$; }

$T \rightarrow$ **real** { $T.type = real$; }

分析 先说明为什么要重新设计 Pascal 语言的变量声明的文法。Pascal 语言变量声明的形式和 C 语言对应形式的重要区别是,Pascal 语言的类型信息出现在变量的后面。从产生式 $D \rightarrow L$: T 看,类型信息要从右边流向左边。因此针对该文法是写不出 L 属性定义的,因为非终结符 L 的属性计算需要它右边的符号 T 的属性。

如何克服困难? 从实现该翻译方案的手工编程所用的数据结构可以得到一点启发。由于不知道 L 会推出多少个变量标识符,因此很可能把读到的变量标识符都压到一个栈中。然后当有了类型信息后,再退栈并把类型信息填入符号表。那么,如何让翻译方案也体现这种先压栈后退栈的操作顺序呢? 自下而上的分析是移进-归约方式,如果让读到标识符时都做移进动作,读到类型时才开始做归约动作,那就能把类型信息填入符号表。因此将文法重新设计,一直到句子全读完时才做归约。从图 4.3 的 **id**, **id**, **id** : **real** 的分析树可以看出这一点。

可以这样小结,如果在句子中,信息是从右向左流动的,那么文法应该设计成从右向左归约(如图 4.3 所示),此时属性计算很可能只用综合属性就能完成,就像上面改造后的 Pascal 语言变量声明的翻译方案那样。

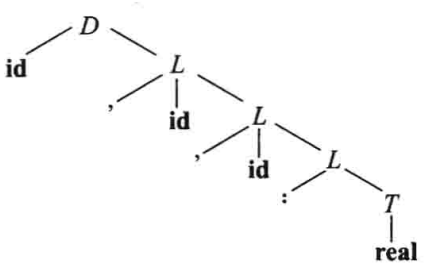


图 4.3 **id**, **id**, **id** : **real** 的分析树

S 属性定义容易写,也容易理解,而且像 Yacc 这样的生成器就能胜任处理 S 属性定义的工作。那么,上面 C 语言变量声明的翻译方案能否也通过改造文法而写出只用综合属性的翻译方案? 回答是肯定的。先将文法改造如下:

$D \rightarrow D$, **id**

$D \rightarrow T$ **id**

$T \rightarrow$ **int**

$T \rightarrow$ **real**

变量声明 **int id**, **id**, **id** 的分析树见图 4.4。然后加上语义动作而构成的翻译方案如下：

$D \rightarrow D_1$, **id**　　{ *addType* (**id**. *entry*, D_1. *type*); *D*. *type* = D_1. *type*; }

$D \rightarrow T$ **id**　　{ *addType* (**id**. *entry*, *T*. *type*); *D*. *type* = *T*. *type*; }

$T \rightarrow$ **int**　　{*T*. *type* = *integer*;}

$T \rightarrow$ **real**　　{*T*. *type* = *real*;}

上面的小结需要进行补充。如果在句子中，信息是从左向右流动的，那么文法应该设计成从左向右归约（如图 4.4 所示），此时属性计算一般都可以只用综合属性完成，上面改造后的 C 语言变量声明的翻译方案就是这样。

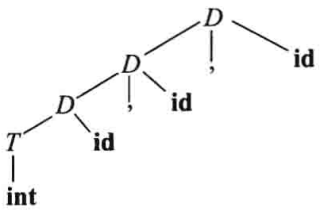


图 4.4　**int id**, **id**, **id** 的分析树

把这两个例子结合起来得到的一点启示是，在设计语言时，要考虑让归约方向和句子中信息流动的方向尽量保持一致，这时语法制导定义比较容易设计。

也许有人会说，改造后的文法可读性较差。这没有关系，语言实现时用的文法并非一定要和用户使用的文法一样。

***4.19**　像习题 4.6 那样，用语法制导定义决定 *S*. *val*。在该定义中，*B* 的唯一综合属性是 *c*，它给出由 *B* 产生的那一位对最终值的贡献。例如，101.101 的最左一位和最右一位对值的贡献分别是 4 和 0.125。

$S \rightarrow L.L \mid L$

$L \rightarrow LB \mid B$

$B \rightarrow 0 \mid 1$

答　先将文法改写成

$S \rightarrow L.R \mid L$

$L \rightarrow BL \mid B$

$R \rightarrow RB \mid B$

$B \rightarrow 0 \mid 1$

所求语法制导定义如下，其中 *i* 是 *B* 的继承属性，*val* 和 *c* 是综合属性。

$S \rightarrow L.R$	$S.val = L.val + R.val$
$S \rightarrow L$	$S.val = L.val$
$L \rightarrow BL_1$	$B.i = L_1.c \times 2;\ L.c = L_1.c \times 2;\ L.val = L_1.val + B.c$
$L \rightarrow B$	$B.i = 1;\ L.c = 1;\ L.val = B.c$
$R \rightarrow R_1B$	$B.i = R_1.c / 2;\ R.c = R_1.c / 2;\ R.val = R_1.val + B.c$
$R \rightarrow B$	$B.i = 0.5;\ R.c = 0.5;\ R.val = B.c$
$B \rightarrow 0$	$B.c = 0$
$B \rightarrow 1$	$B.c = B.i$

分析 首先,从 B 推出的字符是不可能知道这个 B 在串中的具体位置的,因此也不可能计算它的贡献。这样必须为 B 设置继承属性,以便计算 B 的贡献。对于一个二进制数,如 101.101,对于小数点前的 1,需要知道它处在从右向左数的第几位上;而对于小数点后的 1,需要知道它处在从左向右数的第几位上。考查原来的文法,从下面图 4.5 的分析树可以看出,不管数字 1 位于小数点前还是小数点后,它都是一种自左向右的归约,适合于自左向右地计数。这样,小数点左边就需要用继承属性来计数,从而提高了难度。而且,由于小数点左右都使用非终结符 L,那么,L 就既要有综合属性用于从自左向右地计数,又要有继承属性用于自右向左地计数,显然语法制导定义的难度会大大增加。所以这里采用改写文法的办法,让分析树是图 4.6 的形式。这样,小数点两边的归约方式不一样,适合于不同方向的计数,使得语法制导定义能简洁得多。

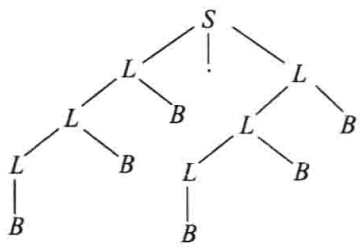


图 4.5 都是自左向右归约的分析树

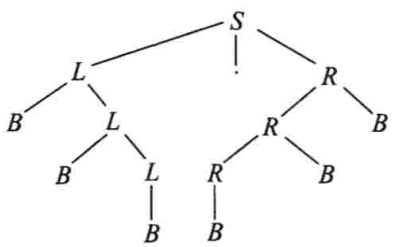


图 4.6 不同归约方式的分析树

由于在分析树中一个 B 对应一个 L(小数点左边)或一个 R(小数点右边),那么用 L 和 R 的 c 属性来表示对每一位的贡献,然后传给 B 的继承属性 i。如果 B 是 1,那么 B 的 c 属性就等于它的 i 属性,否则就等于 0。

如果不限制 B 的综合属性的个数及其含义,那么可以写出更简单的语法制导定义,无须继承属性(见 4.20 题)。另外,为了使语义规则简明,本题使用了非 L 属性的语法规则,它出现在产生式 $L \rightarrow B\ L$ 的规则集中。

从本习题可以再次看出,掌握改写文法的技巧对设计语法制导定义是很有用的。

4.20 为下面文法写一个语法制导的定义,用 S 的综合属性 val 给出下面文法中 S 产生的二进制数的值。例如,输入 101.101 时,$S.val = 5.625$。不得修改文法,但属性使用没有限制。

$S \rightarrow L.R \mid L$

$L \rightarrow LB \mid B$

$R \rightarrow BR \mid B$

$B \rightarrow 0 \mid 1$

答 用 val 属性表示各非终结符推出二进制数所代表的值(注意 R 处于小数点右面),那么所求语法制导定义如下:

$S \rightarrow L.R$	$S.val = L.val + R.val$
$S \rightarrow L$	$S.val = L.val$

$L \rightarrow L_1 B$　　$L.val = L_1.val \times 2 + B.val$

$L \rightarrow B$　　$L.val = B.val$

$R \rightarrow BR_1$　　$R.val = (R_1.val + B.val)/2$

$R \rightarrow B$　　$R.val = B.val/2$

$B \rightarrow 0$　　$B.val = 0$

$B \rightarrow 1$　　$B.val = 1$

分析　当对属性使用没有限制时，这个文法是适合于写语法制导定义的文法，仅需要一个综合属性就够了。注意，在产生式 $L \rightarrow L_1B$ 和 $R \rightarrow BR_1$ 中，B 在右部的位置是不同的。这是为了适应以下的情况：在小数点左边，离小数点越远的 1 的值越大；而在小数点的右边，离小数点越远的 1 的值越小。

4.21　为下面的表达式文法写一个只用综合属性的翻译方案（所有的语义动作都是在归约时完成），它把有加和减的中缀表达式翻译成后缀表达式。

$E \rightarrow TR$

$R \rightarrow +TR \mid -TR \mid \varepsilon$

$T \rightarrow \textbf{num}$

答　在文法中加入标记非终结符 M 和 N，修改后的文法的翻译方案如下：

$E \rightarrow TR$

$R \rightarrow +TMR_1$

$R \rightarrow -TNR_1$

$R \rightarrow \varepsilon$

$M \rightarrow \varepsilon$　　{ printf ('+'); }

$N \rightarrow \varepsilon$　　{ printf ('-'); }

$T \rightarrow \textbf{num}$　　{ printf (**num**.*val*); }

分析　从本题开始，考虑语法制导定义和翻译方案的实现。

若不改造文法，则无法写出符合题目要求的翻译方案，下面根据表达式 9-5+2 的最右推导（规范归约的逆）具体解释。

$$E \Rightarrow TR \Rightarrow T\text{-}TR \Rightarrow T\text{-}T\text{+}TR \Rightarrow T\text{-}T\text{+}T \Rightarrow T\text{-}T\text{+}\textbf{num}$$
$$\Rightarrow T\text{-}\textbf{num}\text{+}\textbf{num} \Rightarrow \textbf{num}\text{-}\textbf{num}\text{+}\textbf{num}$$

从上面的例子可以看出，对于产生式 $R \rightarrow +TR_1$ 和 $R \rightarrow -TR_1$，它们的+和-必须在进行 R_1 推出的部分向 R_1 归约前输出。在原文法上做不到这一点。

前面已经介绍过改写文法的一些技巧。本题的方法和前面的不同，它并不对文法做大的改动，而是在原文法上增加一些推出空串的非终结符，称之为标记非终结符。

对于产生式 $R \rightarrow +TR_1$，因为在分析 R_1 前必须输出+，而动作只能在归约时完成，因此分析 R_1 前必须做一次归约，以输出+。所以增加标记非终结符 M，强迫进行一次 ε 归约。产生式 $R \rightarrow$

$-TR_1$的情况类似。

在加标记非终结符的情况下，最容易犯的错误是在设计形式为 $M \rightarrow \varepsilon$ 的产生式的语义动作时，引用的属性不属于本产生式的文法符号。因为稍不注意就会把嵌入了 M 的产生式的文法符号的属性用到了 $M \rightarrow \varepsilon$ 的语义动作中。

加标记非终结符时容易犯的另一个错误是，改写后的文法不再是 LR(1)文法。可以举下面这样一个简单的例子。

为下面的表达式文法写一个只用综合属性的翻译方案，它将中缀表达式翻译成前缀表达式。

$E \rightarrow E+T$

$E \rightarrow E-T$

$E \rightarrow T$

$T \rightarrow (E)$

$T \rightarrow$ **num**

很可能采用在文法中加入标记非终结符 M 和 N，得到如下的翻译方案：

$E \rightarrow ME_1 + T$

$E \rightarrow NE_1 - T$

$E \rightarrow T$

$M \rightarrow \varepsilon$ { printf ('+'); }

$N \rightarrow \varepsilon$ { printf ('-'); }

$T \rightarrow (E)$

$T \rightarrow$ **num** { printf (**num**. *val*); }

该翻译方案很漂亮，很可惜它的基础文法不是 LR(1)的。读者应该能看出其中的归约-归约冲突。

第 7 章还会有加标记非终结符的例子。

另外，本题属于忽略规则的方法。因为题目已经告知属性只能在归约时完成，那么只能按照这种限定去构造翻译方案。因为属性的计算时机和次序不受所写规则的影响，因此该方法称为忽略规则的方法。

4.22 下面是构造语法树的一个 S 属性定义。将这里的语义规则翻译成 LR 翻译器的栈操作代码段。

$E \rightarrow E_1+T$	$E.nptr = mknode('+', E_1.nptr, T.nptr)$
$E \rightarrow E_1-T$	$E.nptr = mknode('-', E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr = T.nptr$
$T \rightarrow (E)$	$T.nptr = E.nptr$
$T \rightarrow$ **id**	$T.nptr = mkleaf($**id**, **id**. *entry*$)$
$T \rightarrow$ **num**	$T.nptr = mkleaf($**num**, **num**. *val*$)$

答 该语法制导定义的语义规则翻译成的 LR 翻译器的栈操作代码段如下：

$E \rightarrow E_1 + T$　　$val[ntop] = mknode('+', val[top-2], val[top])$

$E \rightarrow E_1 - T$　　$val[ntop] = mknode('-', val[top-2], val[top])$

$E \rightarrow T$

$T \rightarrow (E)$　　$val[ntop] = val[top-1]$

$T \rightarrow \mathbf{id}$　　$val[ntop] = mkleaf(\mathbf{id}, val[top])$

$T \rightarrow \mathbf{num}$　　$val[ntop] = mkleaf(\mathbf{num}, val[top])$

分析 上面的答案中，*val* 是存放属性值的栈，*top* 是栈顶指针，*ntop* 的值为 *top*−*r*+1，其中 *r* 等于待归约的产生式的右部符号个数。这样，*ntop* 就是归约后的栈顶。完成归约后，*ntop* 的值赋给 *top*。

本题比较简单，每个非终结符都只有一个综合属性 *nptr*，所以认为属性栈的元素的类型和 *nptr* 的类型是一样的。如果非终结符有多个综合属性，那么属性栈的元素的类型是这多个综合属性的记录。此时，上面的栈操作代码段应该写成下面的形式：

$E \rightarrow E_1 + T$　　$val[ntop].nptr = mknode('+', val[top-2].nptr, val[top].nptr)$

如果不同非终结符的综合属性的类型不同的话，那么属性栈的元素的类型就是这些类型的共用体类型。

***4.23** 构造表达式语法树的翻译方案如下（就是习题 4.2 的翻译方案，但将加号和减号统一用 addop 表示）：

$E \rightarrow T \{R.i = T.nptr;\} R \{E.nptr = R.s;\}$

$R \rightarrow \mathbf{addop}\ T \{R_1.i = mknode(\mathbf{addop}.lexeme, R.i, T.nptr);\} R_1 \{R.s = R_1.s;\}$

$R \rightarrow \varepsilon \{R.s = R.i;\}$

$T \rightarrow (E) \{T.nptr = E.nptr;\}$

$T \rightarrow \mathbf{id} \{T.nptr = mkleaf(\mathbf{id}, \mathbf{id}.entry);\}$

$T \rightarrow \mathbf{num} \{T.nptr = mkleaf(\mathbf{num}, \mathbf{num}.val);\}$

将这个翻译方案翻译成一个预测翻译器。

答 非终结符 *E*、*R* 和 *T* 的翻译函数分别如下：

```
syntaxTreeNode *E() {
syntaxTreeNode *nptr, *nptr1, *i1, *s1;
    nptr1 = T();
    i1 = nptr1;
    s1 = R(i1);
    nptr = s1;
    return nptr;
}
```

```
syntaxTreeNode * R (syntaxTreeNode * i) {
    syntaxTreeNode * nptr, * i1, * s1, * s;
    char addoplexeme;
    if (lookahead == addop) { /*产生式 R → addop T R */
        addoplexeme = lexval;
        match (addop);
        nptr = T();
        i1 = mkNode (addoplexeme, i, nptr);
        s1 = R ( i1 );
        s = s1;
    } else {
        s = i; /*产生式 R → ε */
    }
    return s;
}

syntaxTreeNode * T() {
    syntaxTreeNode * nptr, * nptr1;
    int val;
    syntaxTreeNode * entry;
    if (lookahead == ' (' ){ /*产生式 T → (E) */
        match (' (' );
        nptr1 = E();
        match (') ' );
        nptr = nptr1;
    }else if (lookahead == num) { /*产生式 T → num */
        val = lexval;
        match (num);
        nptr = mkleaf (num, val);
    } else if (lookahead == id) { /*产生式 T → id */
        entry = lexval;
        match (id);
        nptr = mkleaf (id, entry);
    }else {
```

```
            error;
        }
        return nptr;
    }
```

说明 预测翻译器不同于预测分析器的地方是,它增加了语义计算部分,并且是边分析边计算的。

预测分析器为每个非终结符构造一个分析过程,而预测翻译器为每个非终结符构造一个翻译函数。该函数为这个非终结符的每个继承属性声明一个形式参数,并以这个非终结符的综合属性的类型作为函数的结果类型。该函数还为这个非终结符的产生式的每个文法符号的每个属性声明一个局部变量(这个非终结符的继承属性除外,因为已经把它们声明为形式参数了)。

然后,把翻译方案中的语义动作翻译成属性计算的语句,并根据它们在翻译方案中的不同位置,把它们插在语法分析动作的不同地方(形成运行时的边分析边计算),最后把这个非终结符的综合属性作为函数值返回。

这种构造预测翻译器的方法,也是属于忽略规则的方法。因为它是机械地把所给出的翻译方案翻译成一个个翻译函数,而构造翻译方案的人必须保证其写出的翻译方案是能用这种自上而下的翻译器来完成属性计算的。

***4.24** Pascal 语言的类型声明的语法及把类型填入符号表的语法制导定义如下:

$D \rightarrow L : T$	$L.in = T.type$
$L \rightarrow L_1$, **id**	$addType$ (**id**.$entry$, $L.in$); $L_1.in = L.in$
$L \rightarrow$ **id**	$addType$ (**id**.$entry$, $L.in$)
$T \rightarrow$ **integer**	$T.type = integer$
$T \rightarrow$ **real**	$T.type = real$

用先分析后翻译的方法,构造非终结符 D 和 L 的翻译函数。

答 用函数 child (m, i)表示在语法树上引用结点 m 的第 i 个子结点,用函数 getAttribute(m)表示在语法树上引用叶结点 m 的属性,把 integer 和 real 看成是枚举类型 datatype 的值,另外,规定符号表条目的指针的类型是 idEntry。所求的两个翻译函数如下:

```
void D(syntaxTreeNode n) {
    datatype in, type;
    type = T(child ( n, 3));
    L (child (n, 1 ), type );
}

void L(syntaxTreeNode n, datatype in) {
    datatype in1;
```

```
    idEntry entry;
    switch (在结点 n 的产生式) {
        case' L → L1, id' :
            entry = getAttribute (child (n, 3));
            addType (entry, in);
            in1 = in;
            L (child (n, 1), in1);
            break;
        case ' L → id' :
            entry = getAttribute (child (n, 1));
            addType (entry, in);
            break;
        default:
            error;
    }
}
```

说明 注意,本题的求解方法是先生成分析树,然后再进行语义计算。和上一题相比,本题的翻译函数有下面一些区别。

1. 多了一个参数,它是语法树的结点;
2. 不含语法分析部分;
3. 子结点的访问次序不一定是从左到右。

上面 D 函数中,子结点的访问次序是从右到左;L 函数中,对于产生式 $L \rightarrow L_1$, **id** 的情况,子结点的访问次序也是从右到左。正是先有了分析树,才使得可以用不同的次序遍历分析树,进行属性计算。

本题的方法是属于基于规则的方法。在构造编译器时对语义规则进行静态分析,决定语义规则的计算次序。如果这些计算能在语法分析的同时完成,那么通常让编译器边分析边计算;否则,编译时一般采用先进行语法分析、构造语法树,然后进行语义规则计算的方法。

其实,有了分析树后,再难一点的语法制导定义也能完成属性计算。例如,若对分析树的一次遍历尚不能完成计算,则可以采用对分析树作多次遍历的方法。对于需多次遍历的情况,就不在此举例了。

第5章 类型检查

一、主要知识点

主教材第5章主要应掌握下面一些内容。

1. 静态检查所涉及的内容:类型检查、控制流检查、唯一性检查和关联名字检查等。重点是类型检查。

2. 语言的哪些规定不可能静态地完成检查,而必须用动态检查。

3. 有关类型化语言、程序执行错误和安全语言的概念,类型化语言的优点。

4. 定型规则、类型系统、类型检查和类型推断概念。

5. 设计完成类型检查的语法制导定义。

6. 多态函数和重载函数的概念以及它们的区别。

7. 类型表达式的结构等价和名字等价。

类型系统是在程序中强加抽象层次的一种语法方法,它从类型论观点出发对程序设计语言及其类型系统进行研究,在软件工程、语言设计、高性能的编译器和软件安全性等方面有重要的应用。很多教材对这一部分介绍得很少,因此本章给出的习题也比较简单,实际编程语言的类型系统要复杂得多。

本章的难点是多态类型和多态函数的类型推导,其类型推导之所以困难是因为要使用合一算法。

从本章开始,每一章都用一些C语言的小程序作为例子来帮助理解和巩固所学的知识。另一方面,C程序设计的教材一般都忽略C语言中一些深层次的内容,在此给出这样一些例子有助于读者更好地掌握C语言。本章的例子能使读者对C语言类型系统的认识达到一个新的高度。

对于以下各章涉及C语言小程序的习题,习题中所描述的编译或运行时的报错信息、生成的汇编代码及运行结果等,可能在你的机器上不能再现,那是因为编译器、操作系统或机器不同引起的。

二、习题精选与解析

5.1 下面的Pascal语言程序能否通过编译,若能,该程序的运行结果是什么?

```
program trick(output);
```

```
const   j = 10;
procedure abc;
const   i = j;
        j = 100;
begin
    writeln (i)
end;
begin
    abc
end.
```

答　使用不同的 Pascal 语言编译器可能会有不同的结果。若编译器不报错,该程序的运行结果是 10。

分析　这个问题和 Pascal 语言的作用域规定有关。先解释一下 Pascal 语言的分程序概念,分程序(Block,在 C 语言中翻译成程序块)是由局部声明和可执行语句构成的一种程序构造。例如,从第 2 行的常量声明一直到最后一行的 **end**(不包括 **end** 之后的那个点)是主程序 trick 的分程序,而从第 4 行的常量声明开始到第 8 行的 **end**(不包括 **end** 之后的分号)是过程 abc 的分程序。如果把分程序中标识符的作用域规定为从定义点到分程序的结束,那么对于过程 abc 中的 **const** i = j,这个 j 的引用性出现对应的定义是主程序中的 **const** j = 10。因此常量 i 的值是 10,程序的运行结果也是 10。历史上,使得结果为 10 的编译器确实存在。

这种规定对 Pascal 语言的一遍扫描实现也是有利的。遇到标识符的引用性出现时,查符号表,若找不到该标识符的定义,则报告该标识符没有定义。若能找到它的定义,则按作用域嵌套规则,决定该引用性出现对应该标识符哪个定义。不管后面的程序正文是什么内容,这个结论不会改变。

如果把分程序中标识符的作用域规定为整个分程序,那么过程 abc 中的 **const** i = j,这个 j 的引用性出现对应的定义是它下面的 **const** j = 100。这时常量 i 的值是 100,程序的运行结果是 100。作者过去用过不少 Pascal 语言的编译器,还没有碰到过这种情况。

也可以把分程序中标识符的作用域规定为整个分程序,但是该分程序中这个标识符的定义点前不能有同样拼写的标识符的引用性出现(递归定义的类型除外)。若采用这样的规定,那么过程 abc 中的 **const** i = j,这个 j 的引用性出现就违反了上面的限定,因而编译器会报错。确实存在这样的编译器,它报告 **const** i = j 中的 j 的含义不清。

对于后两种情况,给编译器带来的困难是,一些静态语义错误的发现会滞后。像上面的 **const** i = j,在读完这个声明后,并不能保证它没错,要等后面的常量声明都读完后才能最后确定。

历史上,由于 Pascal 语言作用域规定得不准确,或者说实现者的理解不一样,造成对这样一个简单程序有多种不同的编译结果。

这个例子展现了准确定义语言的重要性和难度。

5.2 下面的 Pascal 程序在某些环境中运行时,报告内存访问错,为什么?

```
program strange (output);
    var p: ↑integer;
    begin
        p:= nil;
        if ( p <> nil) and (p ↑= 10)
            then writeln (10)
            else writeln ('nothing')
    end.
```

答 因为 Pascal 语言的布尔表达式采用的是完全计算语义,导致 p ↑是一个悬空的引用。

分析 从直观上看,布尔表达式(p <> nil) **and** (p ↑= 10)肯定为假,因为(p <> nil)为假。但是 Pascal 语言规定布尔表达式必须完整地计算,也就是说不能因已经能断定布尔表达式的值而放弃剩余部分的计算。回到这个例子,(p ↑= 10)必须计算,由于 p 是空指针,因此报告访问错。

C 语言的布尔表达式采用的是部分计算,即一旦能确定布尔表达式的值,则放弃剩余部分的计算。因此相应的 C 程序是不会报错的。

5.3 举一个关联名字检查的例子。

答 在 Ada 语言中,循环可以命名,例如:

```
swap:
    while … loop
    …
    end loop swap;
```

这个循环命名为 swap,在该循环的结束也加上了这个名字。前后这一对 swap 是关联名字。在循环有多重嵌套时,用关联名字便于程序员确定一个循环的结尾。如果使用关联名字,编译器要检查关联名字是否一致。

5.4 下面是一个 C 语言程序,其中的形参采用传统的声明方式。虽然 main 函数中两次调用函数 gcd 的参数个数都不对,但是该程序却能够通过编译并且连接装配而形成一个目标程序。试问为什么 C 编译器和连接装配器没能发现这样的错误?

```
long gcd(p,q) long p,q; {
    if (p%q == 0)
        return q;
```

```
else
    return gcd(q, p%q);
}
main(){
    printf("%ld,%ld\n",gcd(5),gcd(5,10,20));
}
```

答 这是因为C编译器和连接装配器没有进行函数调用和函数定义的类型是否一致的检查。

分析 C语言及其实现的最初设计,考虑了以文件为单位的独立编译,一个程序可以由若干个文件组成。这样就有可能出现:函数的定义在一个文件中,而函数的调用在另一个文件中。那么在编译函数调用时,就有可能因缺少函数声明的信息,而无法完成函数调用和函数定义的类型是否一致的检查。一种办法是放弃这种检查,由程序员自己负责,对于本例这种语法形式的函数定义,C编译器一般都是这么做的。另一种办法是程序员在有函数调用的文件中加上被调用函数的参数个数和类型以及结果类型的声明,编译器进行类型检查,C++编译器就是这么做的(有关函数定义和调用问题在5.14题有进一步的说明)。

C编译器不做这样检查的另一个原因是,C语言允许参数个数可变的情况。例如,printf的参数个数是可变的。它和Pascal语言中write语句的参数个数可变是不同的。Pascal语言中write语句的实现是不可能用Pascal语言本身写出来的,然而C语言的printf函数的实现是可以用C语言来描述的。

基于这样的一些原因,即使函数定义和函数调用出现在同一个文件中,并且是先定义后调用,C编译器也不做函数调用与函数定义的类型是否一致的检查。

为了帮助程序员检查类型方面的问题,UNIX系统中的工具lint提供了比C编译器强得多的类型系统。它完成函数调用和函数定义的类型是否一致的检查。

***5.5** C语言是一种类型化语言,但它不是静态类型化语言,因为运行前的类型检查不能保证所接受的程序没有运行时的类型错误。例如,编译时的类型检查一般不能保证运行时下标表达式的值都不会越出数组的边界。请你再举一个例子说明C语言不是静态类型化语言。

答 例如共用体(union)的类型检查一般也不可能在运行前完成,虽然下面这个例子是可静态判断类型错误的。

```
union U { int u1; int *u2;} u;
    int *p;
    u.u1 = 10;
    p = u.u2;
    *p = 0;
```

分析 对于上面的共用体类型,如果在条件语句的一个分支给u1赋值,而在另一个分支给

u2 赋值,那么,在该条件语句的条件表达式的值不能静态确定的情况下,条件语句之后 u 所含数据的类型不可能在运行前确定。

另外,指针的类型分析一般来说也不是静态可完成的。

但是,不要把能做到而没有去做的事情和做不到的事情混淆,例如 5.4 题的参数个数和类型问题,实际可以静态做得更好一些,但是过去的 C 编译器没有这样去做(有关这个问题在 5.14 题有进一步的说明)。

5.6 举一个例子说明编译时的控制流检查。

答 用 continue 语句作为例子。下面是一个简单的 C 程序(所在的文件是 continue.c):

```
main( ) {
    printf( "\n% ld\n",gcd(4,12) );
    continue;
}
```

一个编译器编译时的报错如下:

```
continue.c:  In function ‘main’:
continue.c:4: continue statement not within a loop
```

分析 引起控制流离开一个语言构造的语句必须指出控制流应该转到的地方。例如,C 语言的 break 语句引起控制离开最小包围的 while,for 或 switch 语句,continue 语句类似。如果这样的包围语句不存在,就是一个错误。

编译器以某种方式记住程序中语句的嵌套关系就能完成这些检查。

5.7 举一个例子说明编译时的唯一性检查。

答 用 C 语言的 switch 语句作为例子。下面是一个 C 程序(所在的文件是 switch.c),其分支语句的常量有重复。

```
main( ){
int i;

    switch(i) {
    case 10: printf( "% d\n", 10); break;
    case 20: printf( "% d\n", 20); break;
    case 10: printf( "% d\n", 10); break;
    }
}
```

在 x86/Linux 系统上,一个 C 编译器报错如下。

```
switch.c: In function ‘main’:
```

switch. c:8: duplicate case value

switch. c:6: this is the first entry for that value

分析 唯一性检查是指对象在某些场合只能被定义一次。例如：

- 在一个作用域中，一个标识符只能被声明一次；
- 在 Pascal 的分情况语句和 C 的 switch 语句中，各分支语句的常量必须有区别；
- 枚举类型的元素不能重复。

编译器适当地建立名字表或常量表就能完成这样的检查。

5.8 为下面的类型写出类型表达式：

(a) 指向实数的指针数组，数组的下标从 0~99。

(b) 元素是整型的二维数组(即数组的数组)，它的行下标从 0~9，列下标从 0~19。

(c) 一个函数，它的定义域是从整数到整数指针的函数，它的值域是由一个整数和一个字符组成的记录。

答 (a) $array(0..99, pointer(real))$

(b) $array(0..9, array(0..19, integer))$

(c) 假定作为值域的记录类型的两个域分别记为 i 和 c，那么该函数的类型表达式是

$(integer \rightarrow pointer(integer)) \rightarrow record((i \times integer) \times (c \times char))$

5.9 假如有下列 C 语言的声明：

```
typedef struct {
        int a, b;
} CELL, *PCELL;
CELL foo[100];
PCELL bar(x, y) int x; CELL y ;{ … }
```

为类型 foo 和 bar 写出类型表达式。

答 foo 的类型表达式是

$array(0..99, record((a \times integer) \times (b \times integer)))$

bar 的类型表达式是

$(integer \times record((a \times integer) \times (b \times integer))) \rightarrow$
$pointer(record((a \times integer) \times (b \times integer)))$

5.10 下面的文法定义字面常量的表。该文法中符号的解释和通常的解释一致，但增加了类型 list，它表示类型 T 的元素表：

$P \rightarrow D ; E$

$D \rightarrow D ; D \mid \mathbf{id} : T$

$T \rightarrow$ **list of** T | **char** | **integer**

$E \rightarrow (L)$ | **literal** | **num** | **id**

$L \rightarrow E , L \mid E$

写一个翻译方案,以确定表达式 E 和表 L 的类型。

答

$P \rightarrow D ; E$

$D \rightarrow D ; D$

$D \rightarrow$ **id** : T { *addType* (**id**. *entry*, *T*. *type*) ; }

$T \rightarrow$ **char** { *T*. *type* = *char*; }

$T \rightarrow$ **integer** { *T*. *type* = *integer*; }

$T \rightarrow$ **list of** T_1 { *T*. *type* = *list*(T_1. *type*) ; }

$E \rightarrow$ **literal** { *E*. *type* = *char*; }

$E \rightarrow$ **num** { *E*. *type* = *integer*; }

$E \rightarrow$ **id** { *E*. *type* = *lookup*(**id**. *entry*) ; }

$E \rightarrow (L)$ { *E*. *type* = *list*(*L*. *type*) ; }

$L \rightarrow E$ { *L*. *type* = *E*. *type*; }

$L \rightarrow E , L_1$ { *L*. *type* = **if** (L_1. *type* == *E*. *type*) L_1. *type* **else** *type_error*; }

说明 *addType* (**id**. *entry*, *T*. *type*)表示把类型信息 *T*. *type* 添入符号表的 **id** 栏目中。*lookup* (**id**. *entry*)是从符号表中查找 **id** 的类型信息。

5.11 为下面表达式文法写一个翻译方案,它完成表达式的类型检查。该类型检查发现错误时打印错误信息,并继续检查,好像所期望的类型已经看见一样。

$E \rightarrow$ **literal** | **num** | **id** | E **mod** E | $E [E]$ | $E \uparrow$

答

$E \rightarrow$ **literal** { *E*. *type* = *char*; }

$E \rightarrow$ **num** { *E*. *type* = *integer*; }

$E \rightarrow$ **id** { *E*. *type* = *lookup*(**id**. *entry*) ; }

$E \rightarrow E_1$ **mod** E_2 {**if** (E_1. *type* ! = *integer* || E_2. *type* ! = *integer*)
print("At least the type of one operand of **mod** is error";) ;
if (E_1. *type* == *integer* && E_2. *type* == *integer*)
E. *type* = *integer*;
else
E. *type* = *type_error*; }

$E \rightarrow E_1 [E_2]$ {**if** (E_1. *type* ! = *array*(*s*, *t*))
print("The type of E_1 should be an array type. ") ;
if (E_2. *type* ! = *integer*)
print("The type of E_2 should be integer. ") ;

```
                    if ( E2.type == integer && E1.type == array(s, t) )
                        E.type = t;
                    else
                        E.type = type_error; }
E → E1 ↑            { if ( E1.type != pointer(t) )
                        print( "The type of E1 should be a pointer type. ") ;
                    if ( E1.type == pointer(t) )
                        E.type = t;
                    else
                        E.type = type_error; }
```

5.12 拓展 5.10 题的类型检查规则，使之能包含记录。有关记录部分的类型表达式和表达式的语法如下：

$T \rightarrow$ **record** *fields* **end**

fields $\rightarrow$ *fields* ; *field* | *field*

field $\rightarrow$ **id** : T

$E \rightarrow E$. **id**

答 有关记录部分的类型检查的翻译方案如下，其中函数 *lookup*(*recordType*, *name*) 是从记录类型 *recordType* 中寻找名字为 *name* 的域的类型作为返回值。若找不到这样的域名，则返回 *type_error*。

```
T → record fields end     { T.type = record(fields.type) ; }
fields → fields ; field   { fields.type = fields.type × field.type; }
fields → field            { fields.type = field.type; }
field → id : T            { field.type = id.name × T.type; }
E → E1 . id               { E.type = if ( E1.type == record( s × t) )
                                        lookup( E1.type, id.name)
                                    else
                                        type_error; }
```

5.13 推广下面给出的函数类型检查规则，使之能处理 n 元函数（本题所给的文法看成是在上面几题的基础上增加了有关函数声明和函数调用的部分）。

```
T → T1 '→' T2             { T.type = T1.type →T2.type }
E → E1 ( E2 )             { E.type = if ( E2.type == s && E1.type == s→t)
                                        t
                                    else
```

$type_error$; }

答 先将文法修改如下：

$T \rightarrow T_List\ '\rightarrow'\ T$

$T_List \rightarrow T_List, T \mid T$ // 形式参数类型表文法

$E \rightarrow E(E_List)$

$E_List \rightarrow E_List, E \mid E$ // 实在参数表文法

下面是函数类型检查的翻译方案：

$T_List \rightarrow T$ { $T_List.type = T.type$; }

$T_List \rightarrow T_List_1, T$ { $T_List.type = T_List_1.type \times T.type$; }

$T \rightarrow T_List\ '\rightarrow'\ T_1$ { $T.type = T_List.type \rightarrow T_1.type$; }

$E_List \rightarrow E$ { $E_List.type = E.type$; }

$E_List \rightarrow E_List_1, E$ { $E_List.type = E_List_1.type \times E.type$; }

$E \rightarrow E_1(E_List)$ { $E.type =$ **if** ($E_List.type == s$ && $E_1.type = s \rightarrow t$)

t

else

$type_error$; }

***5.14** 在文件 stdlib.h 中，关于 qsort 的外部声明如下：

```
extern void qsort(void *, size_t, size_t, int (*)(const void *, const void *));
```

下面 C 程序所在的文件是 type.c，用某个 C 编译器编译时，错误信息如下：

```
type.c:18: warning: passing argument 4 of 'qsort' from incompatible pointer type
```

请你对该程序略作修改，使得该警告错误消失，并且不改变程序的结果。

注：程序中关于变量 astHypo 和 n 的赋值以及其他部分被略去。

```
#include <stdlib.h>
typedef struct{
    int     Ave;
    double  Prob;
}HYPO;
HYPO    *astHypo;
int     n;
int HypoCompare(HYPO *stHypo1, HYPO *stHypo2) {
    if (stHypo1->Prob>stHypo2->Prob){
        return(-1);
    }else if (stHypo1->Prob<stHypo2->Prob) {
        return(1);
```

```
    }else{
        return(0);
    }
}/* end of function HypoCompare */
main() {
    qsort (astHypo,n,sizeof(HYPO),HypoCompare);
}
```

答 首先需要说明一点,历史上,C 语言中有参函数定义的一般形式是:

类型标识符 函数名(形式参数名字列表)

形式参数类型声明

对于这种形式的声明,C 语言编译器是不做实在参数和形式参数的个数和类型是否一致的检查的。

现在 ANSI 新标准主张使用另一种方法来声明形式参数,即在函数名后的括号中列出形式参数名字的同时,声明这些形式参数的类型(ANSI 新标准也保留了原来的函数定义形式)。本题中的函数就是用这种方式声明的。现在的 C 编译器对于这种形式的函数声明是要进行实在参数和形式参数的个数和类型是否一致的检查的(现在的 C 程序设计教材上都提到这两种参数声明形式,但却未讲它们的区别)。本题就是因为这种检查才出现这个警告错误的。

这里提供两种修改方式。第一种是修改函数 HypoCompare 的类型。

```
int HypoCompare(const void *stHypo1, const void *stHypo2) {
    if (((HYPO *) stHypo1)->Prob>((HYPO *) stHypo2)->Prob){
        return(-1);
    }else if (((HYPO *) stHypo1)->Prob<((HYPO *) stHypo2)->Prob) {
        return(1);
    }else{
        return(0);
    }
}/* end of function HypoCompare */
```

第二种方式是修改 main 函数中 qsort 调用的第四个参数。

```
main() {
    qsort ( astHypo,n,sizeof(HYPO),
        (int (*)(const void * , const void *))HypoCompare);
}
```

分析 该程序在编译时有类型警告错误,是因为函数调用 qsort(astHypo, n, sizeof(HYPO), HypoCompare)的第四个实参 HypoCompare 的类型和 qsort 函数原型声明的第四个形参的类型 int (*) (const void *, const void *)不一致。解决办法无非是改函数 HypoCompare 的类型,或

者是在调用时强制 HypoCompare 的类型。

对于第一种办法,在函数 HypoCompare 的首部将形参的类型改成和 qsort 要求的一致,另外在函数体中必须把这些形参的类型强制转换回来。在加类型强制时,注意所需要加的括号。

对于第二种办法,只需要强制 qsort 调用中 HypoCompare 的类型,让它能通过编译就可以了。

由于 C 程序设计的教材上一般只提到函数定义的两种语法形式,但没有指出编译器在类型检查上的区别,因此容易造成一个印象:这只是语法上的区别。

***5.15** 在 C 程序设计的教材上,称 & 为地址运算符,&a 为变量 a 在内存的地址。但是教材上没有强调表达式 &a 的类型是什么。另外,教材上说,数组名 a 代表数组 a 中首元素的地址,但是也没有强调表达式 a 的类型。它们所带来的一个问题是,如果 a 是一个数组名,那么表达式 a 和 &a 的值都是数组 a 中首元素的地址,但是它们的使用是有区别的,初学时很难掌握。

下面给出 4 个 C 文件,请根据某版本的 GCC 编译器报错信息和程序运行结果,写出表达式 a 和 &a 的类型表达式。了解了它们在类型上的区别,正确地使用它们就不困难了。

(1)文件 1:

```
typedef  int  A[10][20];
A  a;
A  *fun() {
     return(a);
}
```

该函数在 x86/Linux 系统上用 gcc 编译时,报告的类型错误如下:

第 4 行:warning: return from incompatible pointer type

(2)文件 2:

```
typedef  int  A[10][20];
A  a;
A  *fun() {
     return(&a);
}
```

该函数在 x86/Linux 系统上用 gcc 编译时,没有类型方面的错误。

(3)文件 3:

```
typedef  int  A[10][20];
typedef  int  B[20];
A  a;
B  *fun() {
     return(a);
}
```

该函数在 x86/Linux 系统上用 gcc 编译时,没有类型方面的错误。

(4)文件 4:

```
typedef  int  A[10][20];
A  a;
fun() {
    printf("%d,%d,%d\n", a, a+1, &a+1);
}
main() {
    fun();
}
```

该程序的运行结果是:

134518112, 134518192, 134518912

答 做一个对多维数组也适用的答案。

对一个元素类型为 t 的数组 $a[i_1][i_2]\cdots[i_n]$ 来说,表达式 &a 的类型是:

pointer(array(0.. i_1 - 1, ⋯ array(0.. i_n - 1, t)⋯))

而表达式 a 的类型是:

pointer(array(0.. i_2 - 1, ⋯ array(0.. i_n - 1, t)⋯))

用文字粗略地表达二者之间的区别是,表达式 &a 的类型是指向整个数组的指针类型,而表达式 a 是指向数组元素的指针类型(把该数组看成一维数组时的数组元素)。

分析 从文件 1 的编译时报错信息可以看出,数组名 a 的类型不是数组类型 A 的指针类型。从文件 2 顺利通过编译可以判断,&a 的类型是数组类型 A 的指针类型。从文件 3 顺利通过编译可以判断,数组名 a 的类型是数组类型 B 的指针类型,而 a[i](0≤i≤9)的类型就是数组类型 B。文件 4 的运行结果进一步确认了以上的判断是正确的,因为 a 所指对象的大小是 80 个字节,而 &a 所指对象的大小是 800 字节。

把这些结论推广到一般数组,就是上面的答案。

可再用一个例子来把一些和数组有关的表达式的类型区分得更清楚。在下面的 C 语言源文件中,所有 return 后面的表达式的值都等于数组 a 中首元素的地址。该文件编译时不会出现类型错误,那么从这三个函数的返回值类型可以清楚地知道这些表达式的类型。

```
typedef int A[3][4];
typedef int B[4];
A  a;
A  *fun2() {
    return &a;
}
B  *fun1() {
```

```
    return a;
    return a+0;
    return &a[0];
}
int *fun0() {
    return a[0];
    return *(a+0);
    return &a[0][0];
}
```

5.16 假定类型名 link 和 cell 的定义如下：

```
type link = ↑cell;
    cell = record
        info: integer;
        next: link
    end;
```

下面的表达式哪些结构等价？哪些名字等价？

(a) link

(b) *pointer* (cell)

(c) *pointer* (link)

(d) *pointer*(*record*((info × *integer*) ×(next × *pointer*(cell))))

答 link, *pointer* (cell) 和 *pointer*(*record*((info × *integer*) ×(next × *pointer*(cell)))) 三者结构等价。没有名字等价的类型。

分析 从 link 和 cell 的定义知道，link 代表的类型表达式是 *pointer*(cell)，cell 代表的类型表达式是 *record*((info × *integer*) ×(next × link))。题中(a)和(b) 的类型表达式结构等价是显然的。将 cell 类型表达式中的类型名 link 用 pointer (cell)代换后，得到 *record*((info × *integer*) × (next ×*pointer*(cell)))，因此(b)和(d)的类型表达式是结构等价的。

注意，link 和 cell 的定义是相互递归的，结构等价测试程序会发现这种递归，以避免测试过程无穷递归。

名字等价要求两个类型表达式完全相同，并且把类型表达式中的类型名看成是一个可辨别的类型，因此本题不存在名字等价的类型表达式。

5.17 一个 C 语言程序如下：

```
void fun(struct {int x; double r;} val) {   }
main() {
```

```
    struct {int x; double r;} val;
    fun(val);
}
```

该程序在 x86/Linux 系统上用某版本的 GCC 编译器编译时，报告的错误信息如下：

```
4: error: incompatible type for argument 1 of 'fun'
1: note: expected 'struct <anonymous>' but argument is not of type 'struct <anonymous>'
```

请问，报告错误的原因是什么？如何修改程序，使得编译时不再出现这个错误信息？

答 C 语言对所有的类型都采用结构等价，唯有结构体类型例外，采用名字等价。这里的类型不相容是因为作为形参和实参的两个 val 的类型不是名字等价的。

要消除这个错误，包括所有的警告，将程序修改如下：

```
struct s{
    int x;
    double r;
};
void fun(struct s val) {   }
main()          {
    struct s val;
    fun(val);
}
```

分析 C 程序设计教材上都简单地使用类型一致或类型不一致这样的概念，避免介绍 C 语言的类型等价概念，这也许是出于避免让初学者接触较高级的语言概念的目的。作为学过编译原理的同学来说，对 C 语言的掌握应该提高到一个新的高度。

5.18 对于下面的 C 语言程序，在 x86/Linux 系统上，某版本的 GCC 编译器报告第 11 行有错误：

```
incompatible types in return
```

在 C 语言中，数组和结构体都是构造类型，为什么下面第二个函数有类型错误，而第一个函数没有？

```
typedef int A1[10];
typedef int A2[10];
A1 a;
typedef struct {int i;}S1;
typedef struct {int i;}S2;
S1 s;
A2 *fun1() {
```

```
    return(&a);
}
S2 fun2() {
    return(s);
}
```

答 C 语言对除结构体类型以外的所有类型都使用结构等价，而对结构体类型使用名字等价。第一个函数能通过结构等价的检查，而第二个函数不能通过名字等价的检查。

5.19 使用类型变量表示下列函数的类型：

(a) 函数 *ref*。它取任意类型的对象作为变元，返回该对象的指针。

(b) 函数 *arrayderef*。它以一个数组作为变元，数组的下标是整型，数组的元素是某类型的指针。该函数返回一个数组，该数组的元素是变元数组的元素所指的对象。

答 (a) $ref: \forall\alpha, \alpha \rightarrow pointer(\alpha)$

(b) $arrayderef: \forall\alpha, array(integer, pointer(\alpha)) \rightarrow array(integer, \alpha)$

5.20 函数符号的重载概念同多态函数概念的区别是什么？

答 重载的函数符号在不同的上下文中代表不同的函数，这些函数有不同的类型。即重载的函数符号的含义依赖于上下文，根据上下文给出的类型限制来决定重载函数符号的特定出现代表哪个函数。重载这个概念也可以用到算符。

例如，程序设计语言中的算符“+”既可以代表函数+：$integer \times integer \rightarrow integer$，也可以代表函数+：$real \times real \rightarrow real$，根据“+”的两个运算对象的类型来确定“+”的一个特定出现是代表哪个加运算。为了区别下面的多态类型，需要强调的是，整数加和实数加是不同的函数，因为它们有不同的实现。

在 Ada 语言中，括号()是重载的，表达式 A(I)可能表示下标变量，也可能是函数调用，这取决于 A 的类型。

C++语言允许用户定义的函数符号重载。

多态函数是指其变元可以取不同类型的函数，虽然变元类型可以不同，但函数体只有一个，这样，该函数体中的语句可以在不同类型的变元下执行。这个概念也可以用到算符。

例如，Pascal 语言中取指针指向的对象的操作↑是一个多态的操作，因为不管它的操作对象是什么类型，取它所指向对象的操作序列是一样的。Pascal 语言的数组索引也是一个多态操作，因为不管数组是什么类型，它的下标是什么类型，取下标变量的操作序列是一样的。

通常函数式程序设计语言允许定义多态类型。

从函数使用角度看，好像重载和多态没有什么区别，都可认为是一个函数可以作用于不同类型的变元。但是从实现角度看，在重载情况下，对不同类型的变元，用不同的函数体；而在多态情况下，不管变元是什么类型，用的是同一个函数体。

5.21 下面是一个 C 语言文件的内容。

```
long a1[10][10][10];
long a2[ ][10][10];
long a3[ ][ ][10];
f (a4) long a4[ ][10][10]; {
    a1[2][2][2] = a4[2][2][2];
}
main( ) {
    f (a1);
}
```

把它提交给 GCC 编译器(版本(GNU) 4.2.3 (Debian 4.2.3-5))时,编译器对第二行给出警告:array 'a2' assumed to have one element。对第三行报告错误:array type has incomplete element type。

(a) 为什么第二行给出警告,而第三行报告错误?

(b) 第四行的 a4 和第二行的 a2 的描述一样,为什么编译器对 a4 没有警告?

(c) 若在 f 函数中增加输出表达式 sizeof(a2)和 sizeof(a4)的值,你认为编译时会报错吗?

答 (a) a2 的元素是 10×10 的长整型数组,而 a3 的元素的类型是不清楚的,所以对 a3 要报告错误。a2 数组的元素类型虽然清楚,但数组大小不明确,所以给出警告。

(b) a4 是形参,是元素类型为 10×10 的长整型数组的指针,数组的大小由实参决定,因此形参 a4 第一维的大小是否指明没有关系,所以对 a4 不报错。

(c) 计算 a2 的字节数时会报错,因为 a2 的大小没有指明。计算 a4 的字节数时结果为 4,因为 a4 是一个指针。

第6章　运行时存储空间的组织和管理

一、主要知识点

主教材第6章主要应掌握下面一些内容。

1. 影响存储分配策略的语言特征。

（1）过程能否递归。（FORTRAN语言早期版本不允许，所以可以用静态分配策略）

（2）当控制从一个活动的过程返回时，局部变量的值是否要保留。（C语言通过静态局部变量声明，FORTRAN通过save语句来提出保留要求。这样的变量一般都采用静态分配）

（3）过程能否访问非局部且非全局的变量。（各种语言一般都允许访问全局变量和局部变量，Pascal语言的过程嵌套声明导致除了全局变量和局部变量外，还有非局部且非全局的变量，因而出现静态链）

（4）过程调用的参数传递方式。（C语言以文件为单位的编译和FORTRAN早期版本的过程独立编译导致编译器一般不检查实在参数和形式参数的个数和类型一致问题）

（5）过程能否作为参数被传递。（一般都允许，但是Pascal语言由于有过程声明嵌套，实现起来相对复杂一些）

（6）过程能否作为结果值传递。（C语言允许。Pascal语言的过程嵌套和栈式分配方式，使得它不能提供这种功能）

（7）存储块能否在程序控制下动态地分配。（FORTRAN早期版本不允许。这样的存储块通常分配在堆区）

（8）存储块是否必须显式地释放。（大多数过程语言对程序显式申请的空间要显式地释放。函数式语言一般通过无用存储单元收集程序来隐式完成）

2. 活动记录中各种数据域的作用和安排。

3. 各种存储分配策略，主要了解静态分配和动态栈式分配。

4. 非局部数据访问的实现方法。

5. 各种参数传递方式及其实现。

本章内容是全书的一个重点，不了解运行时存储空间的组织和管理，等于对目标程序在机器上的运行情况一无所知。反过来，掌握了这些知识对程序设计、测试和调试等会有很大帮助。本章用很多实际的例子来帮助大家掌握本章的内容。

本章的难点是栈式分配情况下的存储空间的组织和管理，特别是在允许过程嵌套声明的场合下。

二、习题精选与解析

6.1 一个 FORTRAN 程序如下,请说明该程序的输出结果。

```
CALL SUB
CALL SUB
END

SUBROUTINE SUB
INTEGER I
DATA  I/10/
WRITE (*,*) I
I = 100
END
```

答 该主程序两次调用子例程 SUB,第一次调用输出 10,第二次调用输出 100。

分析 FORTRAN 语言的实现通常采用静态存储分配的策略,因此 DATA 语句中的变量 I 是静态置初值的。这样第一次执行 WRITE 语句时,I 存储单元中的值是 10。然后 I 存储单元的值被重新置成 100,而这个值一直保存到第二次调用 SUB 的 WRITE 语句,所以第二次调用时输出 100。

这样,静态存储分配的 FORTRAN 语言的一个重要特征是,控制再次进入一个过程(子例程或函数)时,局部变量的值等于控制上次离开该过程时它们的值。

值得注意的是,上面的说法是由 FORTRAN 的实现保证的,而 FORTRAN 的语义并不保证这个说法,因为并没有规定 FORTRAN 的实现必须采用静态存储分配。

如果想从语义上保证这一点,需要在上面的过程 SUB 中加上 SAVE 语句。

```
SUBROUTINE SUB
INTEGER I
SAVE I
DATA  I/10/
WRITE (*,*) I
I = 100
END
```

6.2 考虑下面的 C 程序:

```
main( ) {
```

```
    char *cp1, *cp2;
    cp1 = "12345";
    cp2 ="abcdefghij";
    strcpy(cp1,cp2);
    printf("cp1 =%s\ncp2 = %s\n", cp1, cp2);
}
```

该程序经早先某些 C 编译器的编译,其目标程序运行的结果是

```
cp1 = abcdefghij
cp2 = ghij
```

试分析,为什么 cp2 所指向的字符串被修改了?

答 因为字符串常量"12345"和"abcdefghij"连续分配在常数区,执行两个赋值语句后,cp1 和 cp2 的值分别等于字符 1 和 a 所在的地址,即

```
1 2 3 4 5 \0 a b c d e f g h i j \0
↑            ↑
cp1          cp2
```

执行串复制语句后,常数区变成

```
a b c d e f g h i j \0 f g h i j \0
↑          ↑
cp1        cp2
```

而 cp1 和 cp2 所指的地址没有变,所以 cp2 所指的字符串被修改了。

分析 编译原理的教材中几乎都没有提到编译器是怎样给程序中的常量分配空间的。实际上,对于不能直接出现在指令中的常数和字符串常量等,编译器把它们分配在常数区(在生成的汇编程序中,通常由伪指令 .data 指明常数区),并且一般来说,是按它们在程序中出现的先后次序来安排它们在数据区中的次序。所以本题程序中的两个字符串正好连续存放。

现在的编译器大都不会出现题目所讲的情况,而是在编译时或运行时报告错误。因为现在的编译器把程序中的字符串常量单独分配在一个段中,把它们和其他的常数分开,并且该段的内容在程序运行时不能被修改。这样,该程序在执行串复制时,会报告段错误(Segmentation fault)而异常终止。下面是 x86/Linux 系统上某版本的编译器为本程序生成的汇编代码,其中由标号 main 开始的指令部分被略去。从这段汇编代码能看到,程序中的三个字符串常量(包括 printf 调用中的字符串常量)都连续分配在只读数据区中(rodata 指只读数据)。

```
    .file "copy.c"
    .version "01.01"
gcc2_compiled.:
.section.rodata
.LC0:
```

```
        .string "12345"
.LC1:
        .string "abcdefghij"
.LC2:
        .string "cp1 = %s\ncp2 = %s\n"
.text
        .align 4
.globl main
        .type main,@function
main:
        ...
```

6.3 C 程序设计的教材上说,可以用两种形式表示字符串,其一是用字符数组存放一个字符串,另一种是用字符指针指向一个字符串。教材上同时介绍了这两种形式的很多共同点和不同点,但是有一种可能的区别没有介绍。下面是一个包含这两种形式的 C 程序:

```
char c1[ ] = "good! ";
char * c2 = "good! ";
main( ) {
    c1[0] = 'G';
    printf("c1 = %s\n", c1);
    c2[0] = 'G';
    printf("c2 = %s\n", c2);
}
```

该程序在 Ubuntu 12.04.2 LTS (GNU/Linux 3.2.0-42-generic x86_64)系统上,经过编译器 GCC:(Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3 编译后,运行时的信息如下:

```
c1 = Good!
Segmentation fault (core dumped)
```

请问,出现 Segmentation fault 的原因是什么?

答 c1 是字符数组,c1[ ] = "good!"看成是对这个数组的元素逐个地赋值。c2 是字符指针,它所指向的"good!"被看成是一个字符串常量,常量的值不允许修改,因此编译器把这个字符串常量分配在只读数据区。

当执行赋值 c2[0] = 'G' 时,试图对只读数据区赋值,因此报告错误。

6.4 一个 C 语言程序如下:

```
typedef struct  _a {
```

```
    char        c1;
    long        i;
    char        c2;
    double      f;
} a;
typedef struct  _b {
    char        c1;
    char        c2;
    long        i;
    double      f;
} b;
main() {
    printf("Size of double, long, char = %d, %d, %d\n",
                sizeof(double),sizeof(long), sizeof(char));
    printf("Size of a, b = %d,%d\n", sizeof(a), sizeof(b));
}
```

该程序在早先的 SPARC/Solaris 系统上的运行结果如下：

```
Size of double, long, char = 8, 4, 1
Size of a, b = 24,16
```

在 x86/Linux 系统上经编译器 GCC:（GNU） egcs-2.91.66 19990314/Linux（egcs-1.1.2 release）编译后，运行时第二行输出不一样，是

```
Size of a, b = 20,16
```

结构体类型 a 和 b 的域都一样，仅次序不同，为什么给它们分配的存储空间不一样？为什么在不同的机器上情况又不一样？

答 这是由于数据对齐（alignment）问题引起的。通过简单的分析不难看出，字符型数据可以分配到任何地址，即对齐到 1；整型数据对齐到 4（即从能被 4 整除的地址开始分配）；双精度型数据对齐到 8。因此对于类型 a 的结构体，其 4 个域的相对地址依此是 0,4,8,16，总共需 24 个字节。对于类型 b 的结构体，其 4 个域的相对地址依此是 0,1,4,8，总共需 16 个字节。

不同机器的对齐要求可能是不一样的。对于在 x86/Linux 系统上的运行结果，通过分析可以判断，它和上面的区别是，双精度型数据也对齐到 4。

分析 从下面的 C 程序及它在 x86/Linux 系统上汇编代码可以看出，这个判断是正确的。

```
static struct_a {
    char        c1;
    long        i;
    char        c2;
```

```
    double    f;
} a = {'A',10,'B',1.0};
```

上面的 C 程序生成的汇编代码如下：

```
    .file "static.c"
    .version "01.01"
gcc2_compiled.:
.data
    .align 4
    .type a,@object
    .size a,20                    // a 所需空间是 20 字节
a:
    .byte 65                      // 字符'A'
    .zero 3                       // 空三个字节,都置为 0
    .long 10                      // 长整数 10
    .byte 66                      // 字符'B'
    .zero 3                       // 空三个字节,都置为 0
    .long 0x0,0x3ff00000          // 双精度数 1.0,它占两个长整数的空间
    .ident    "GCC:(GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

*6.5 下面给出一个 C 语言程序及其在 x86/Linux 系统上的编译结果(编译器版本见汇编代码最后一行)。根据所生成的汇编程序来解释程序中四个变量在存储分配、作用域、生存期和置初值方式等方面的区别。

```
static long aa = 10;
short bb = 20;
func() {
    static long cc = 30;
    short dd = 40;
}
```

该 C 程序生成的汇编代码如下。

```
    .file "static.c"
    .version "01.01"
gcc2_compiled.:
.data
    .align 4
    .type aa,@object
```

```
        .size aa,4
aa:
        .long 10
.globl bb
        .align 2
        .type bb,@object
        .size bb,2
bb:
        .value 20
        .align 4
        .type cc.2,@object
        .size cc.2,4
cc.2:
        .long 30
.text
        .align 4
.globl func
        .type func,@function
func:
        pushl %ebp
        movl %esp,%ebp
        subl $4,%esp
        movw $40,-2(%ebp)
.L1:
        leave
        ret
.Lfe1:
        .size func,.Lfe1-func
        .ident "GCC:(GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

答 从C语言的规定可以知道,aa是静态外部变量,bb是外部变量,cc是静态局部变量,dd是自动变量。

首先看存储分配。从汇编程序中的标号aa、bb和cc.2以及.long 10、.value 20和.long 30可以看出,aa、bb和cc都分配在静态数据区(由.data伪指令开始)中。而从movw $40,-2(%ebp)等指令可以看出,dd分配在栈区,所以程序中没有出现以dd为标号的静态存储单元。

再看作用域。bb由伪指令.globl指明为全局的,用于解决其他文件中对bb的外部引用,而

aa 只能由本文件引用,因此没有伪指令 .globl。在源程序中,cc 的作用域是函数 func 的体。而在目标文件中,因为 cc 分配在静态数据区,那么所分配数据单元的标号 cc 的作用域就是整个文件。为避免同源文件中的外部变量和其他函数的静态局部变量的名字冲突,所以要对它进行改名,改成了 cc.2。

下面讨论生存期。因为 aa、bb 和 cc 都分配在静态数据区,因此它们的生存期都是整个程序。因为 dd 分配在栈区,因此它的生存期是函数 func 的激活期间。如果 func 是个递归函数的话,准确地说应该是这样:func 的每次激活都在栈上给 dd 分配存储单元,它的生存期就是相应激活的活动期间。

最后看置初值方式。dd 是通过指令 movw $40,-2(%ebp)来置初值的,因此这是动态置初值。aa、bb 和 cc 的置初值由伪指令 .long 10、.value 20 和 .long 30 完成,因此对它们的置初值是在目标程序加载时完成,这是静态置初值的方式。

***6.6** 一个 C 语言程序如下:

```
typedef struct _a {
    short   i;
    short   j;
    short   k;
}a;
typedef struct _b {
    long    i;
    short   k;
}b;
main() {
        printf("Size of short, long, a and b = %d,%d,%d,%d\n",
                    sizeof(short), sizeof(long),sizeof(a),sizeof(b));
}
```

该程序在 Ubuntu 12.04.2 LTS (GNU/Linux 3.2.0-42-generic x86_64)系统上,经过编译器 GCC:(Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3 编译后,运行结果如下:

```
Size of short, long, a and b = 2, 8, 6, 16
```

已知 short 类型和 long 类型分别对齐到 2 的倍数和 8 的倍数。试问,为什么类型 b 的 size 会等于 16?

答 数组 size 的计算原则是该数组元素的 size 乘以数组元素的个数。

对于结构体类型 b 来说,它的一个变量只需 10 个字节就够了。如果声明结构体类型 b 的一个数组,有两种可能:

① 结构类型体 b 的 size 是 10,数组元素之间空 6 个字节,以保证每个数组元素的地址都是 8

的倍数(因为第一个域 i 的类型是 long,要求对齐到 8 的倍数)。

② 结构体类型 b 的 size 是 16,以保证它作为数组元素的类型时,数组元素之间不用再考虑对齐问题。

方法①违反了一开始提到的原则,因此只能取方法②。

6.7 下面是 C 语言两个函数 f 和 g 的概略(它们不再有其他的局部变量):

```
int f (int x) { int i;…return i +1;…}
int g (int y) {int j;…f (j +1);…}
```

请参照主教材上例 6.4 中图 6.11 的形式,画出函数 g 调用 f,f 的函数体正在执行时,活动记录栈的内容及相关信息,并按教材图 6.10 左侧箭头方式画出控制链。假定函数返回值是通过寄存器传递的。

答 见图 6.1。

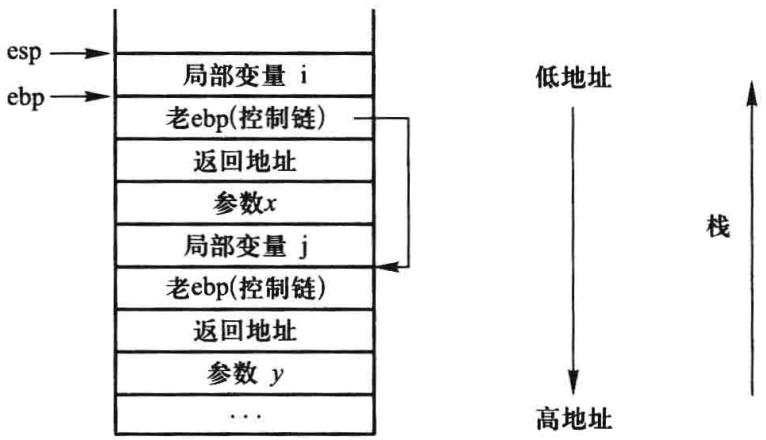


图 6.1 活动记录栈的内容及相关信息

6.8 C 语言函数 f 的定义如下:

```
int f (int x, int *py, int **ppz) {
    **ppz +=1; *py +=2; x +=3; return x + *py + **ppz;
}
```

变量 a 是指向 b 的指针,变量 b 是指向 c 的指针,c 是整型变量并且当前值是 4。那么执行 f (c, b, a)的返回值是多少?

答 参数传递使得 x 的值和 c 的值一样,都是 4;py 的值和 b 的值一样,指向 c;ppz 的值和 a 的值一样,指向 b。

ppz +=1 的执行使得 c 的值为 5。*py +=2 的执行使得 c 的值为 7。x +=3 的执行使得 x 的值为 7。这时 x、*py 和ppz 的值都是 7,因此返回值是 21。

6.9 下面是一个 C 语言程序:

```
main() {
```

```
    long i;
    long a[0][4];
    long j;
    i = 4; j = 8;
    printf("%d, %d\n", sizeof(a), a[0][0]);
}
```

虽然出现 long a[0][4]这样的声明,但在 x86/Linux 系统上,用编译器 GCC:(GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)编译时,该程序能够通过编译并生成目标代码。请回答下面两个问题:

(a) sizeof(a)的值是多少,请说明理由。

(b) a[0][0]的值是多少,请说明理由。

答 (a)根据数组 size 的计算公式,sizeof(a)的值一定是 0。

(b)a[0][0]的值是 4。虽然 a 的 size 是 0,但它仍然有起始地址,并且 a[0][0]的地址等于 a 的起始地址。另一方面,由于 a 的 size 是 0,因此 a 的起始地址肯定同 i 或 j 的地址一样。由于 x86/Linux 系统上,活动记录栈向低地址方向增长,因此 a[0][0]的地址和 i 的地址一致,其值就和 i 的值一样,等于 4。

分析 不同编译器的处理方式是不一样的。在 Ubuntu 12.04.2 LTS (GNU/Linux 3.2.0-42-generic x86_64)系统上,经过编译器 GCC:(Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3 编译后,运行时输出的两个值都是 0。因为对于 size 为 0 的数组,编译器默认它有一个元素。

6.10 假定使用(a)值调用、(b)引用调用、(c)换名调用,下面的程序打印的结果是什么?

```
program main(input, output);
    var a, b: integer;
    procedure p(x, y, z: integer);
        begin
            y := y+1;
            z := z+x;
        end;
    begin
        a := 2;
        b := 3;
        p(a+b, a, a);
        print a;
    end.
```

答 (a)值调用的结果是 2,(b)引用调用的结果是 8,(c)换名调用的结果是 9。

分析 （a）对值调用而言，由于被调用过程中对形式参数的赋值不会改变实在参数的值，因此对过程 p 的调用不改变 a 的值，因此 a 仍然是 2。

（b）在引用调用的时候，p(a+b, a, a)相当于 p(5, a, a)（注意，5 的值存放在临时单元中，传送的是临时单元的地址）。在被调用过程中，任何对形式参数的引用和赋值，也就是对相应实在参数的引用和赋值。因此 y := y+1 的执行结果使得 a 等于 3，进一步执行 z := z+x 使得 a 等于 8。

（c）换名调用等价于下面的执行序列：

```
a := a+1;
a := a+(a+b);
```

所以 a 的最后结果是 9。

6.11 一个 C 语言程序如下：

```
func(i1,i2,i3) long i1,i2,i3; {
    long j1,j2,j3;
    printf("Addresses of i1,i2,i3 = %o,%o,%o\n",&i1,&i2,&i3);
    printf("Addresses of j1,j2,j3 = %o,%o,%o\n",&j1,&j2,&j3);
}
main() {
    long i1,i2,i3;
    func(i1,i2,i3);
}
```

该程序在 x86/Linux 系统上，经某编译器编译后的运行结果如下：

```
Addresses of i1,i2,i3 = 27777775460,27777775464,27777775470
Addresses of j1,j2,j3 = 27777775444,27777775440,27777775434
```

从上面的结果可以看出，func 函数的三个形式参数的地址依次升高，而三个局部变量的地址依次降低。试说明为什么会有这样的区别。注意，输出的数据是八进制的。

答 图 6.2 给出了栈式分配的活动记录的一般情况，通常临时数据区在活动记录栈中靠近栈顶的地方。由于 C 语言对实在参数表达式的计算是采用自右向左的方式，因此在 main 函数中，实在参数进栈的次序是 i3，i2 和 i1。如果栈是向低地址方向增长，那么 i1，i2 和 i3 的地址是逐渐增大的。

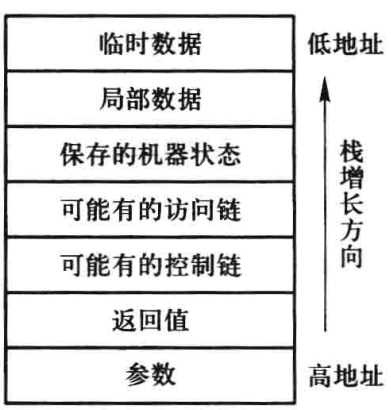


图 6.2 活动记录的一般情况

对于局部变量 j1，j2 和 j3 来说，它们在局部数据区的分配是按照它们在变量声明中出现的次序进行的。如果栈是向低地址方向增长，那么 j1，j2 和 j3 的地址是逐渐减小的。因此出现这种不一致的情况。

*6.12 下面的C语言程序中，函数printf的调用仅含格式控制字符串一个参数，该程序在x86/Linux系统上，经某编译器编译后，运行时输出三个整数。试从运行时存储空间的组织和printf的实现来分析，为什么此程序会有三个整数输出。

```
main( ) {
    printf("%d, %d, %d\n");
}
```

答 由此C语言编译器是不做实在参数和形式参数的个数和类型是否一致的检查的，而是由程序员自己来保证它们的一致性，所以printf的实现并不知道其调用者提供多少个参数。但是，C语言的实现保证被调用函数能准确地取到第一个实在参数。因此printf的实现首先取到第一个参数——格式控制字符串，然后分析它的格式控制要求，根据格式控制中的格式说明，到栈中取第二、第三个参数等，而不管调用者是否真正提供了这些参数。所以该程序有三个整数输出。

现在的一些C语言编译器，在编译时对printf调用的参数进行分析。对于本题的程序，编译器会给出警告错误。

*6.13 下面是一个C语言程序及其在早先的SPARC/SunOS系统上经某编译器编译后的运行结果。从运行结果看，函数func中四个局部变量i1,j1,f1,e1的地址间隔和它们类型的大小是一致的，而四个形式参数i,j,f,e的地址间隔和它们类型的大小不一致，试分析产生这种不一致的原因。注意，输出的数据是八进制的。

```
func(i,j,f,e) short i,j; float f,e; {
    short i1,j1; float f1,e1;
    printf("Address of i,j,f,e = %o, %o, %o, %o\n", &i,&j,&f,&e);
    printf("Address of i1,j1,f1,e1 = %o, %o, %o, %o\n", &i1,&j1,&f1,&e1);
    printf("Sizes of short, int, long, float, double = %d, %d, %d, %d, %d\n",
            sizeof(short),sizeof(int),sizeof(long),sizeof(float),sizeof(double));
}
main( ) {
    short i,j; float f,e;
    func(i,j,f,e);
}
```

运行结果是：

```
Address ofi,j,f,e = 35777772536, 35777772542, 35777772544, 35777772554
Address of i1,j1,f1,e1 = 35777772426, 35777772424, 35777772420, 25777772414
Sizes of short,int, long, float, double = 2, 4, 4, 4, 8
```

答 上一章已提到过,C 语言编译器是不做实在参数和形式参数的个数和类型是否一致的检查的,由程序员自己保证它们的一致性。但是对于形式参数和实在参数是不同的整型(如一个是 short 型,另一个是 long 型),或者是不同的实型,编译器则试图保证目标代码运行时能得到正确的结果,条件是,当需要把高级别类型数据(指需要较多空间)向低级别类型数据(指需要较少空间)转换时,不会产生溢出。这样,C 编译器做的约定是,当整型或实型数据作为实在参数时,将它们分别提升到 long 和 double 类型的数据再传递到被调用函数。被调用函数根据形式参数所声明的类型,决定是否要将实在参数向低级别类型转换。

在本题中,long 类型数据占 4 B,而 short 类型数据只占 2 B。因此被调用函数把实在参数的低位字节的内容当成是自己所需的数据,见图 6.3。注意,对于 Sun 工作站来说,低地址存放整型数据的高位。

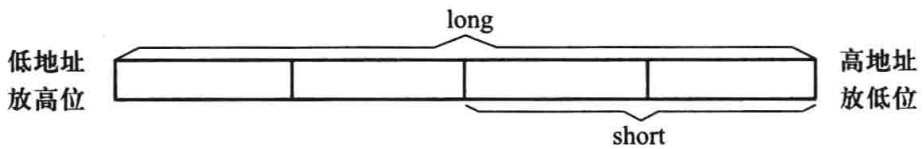


图 6.3 长整型和短整型

对于实型数据来说,double 类型占 8 B,而 float 类型占 4 B。被调用函数把实在参数的前 4 B 作为自己所需的数据,因为 double 后面 4 B 的内容都是为了增加精度用的,见图 6.4。

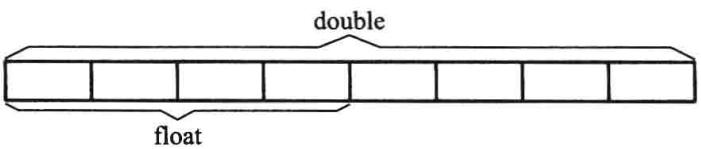


图 6.4 双精度型和浮点型

这样,在 main 函数中依次将参数类型提升后反序进栈,大小分别为 8, 8, 4, 4。在 func 函数中,按形式参数的类型,把这些存储单元的一部分看成是形式参数的存储单元,见图 6.5。从这个图不难理解为什么形式参数的地址间隔和它们的类型大小不一致了。

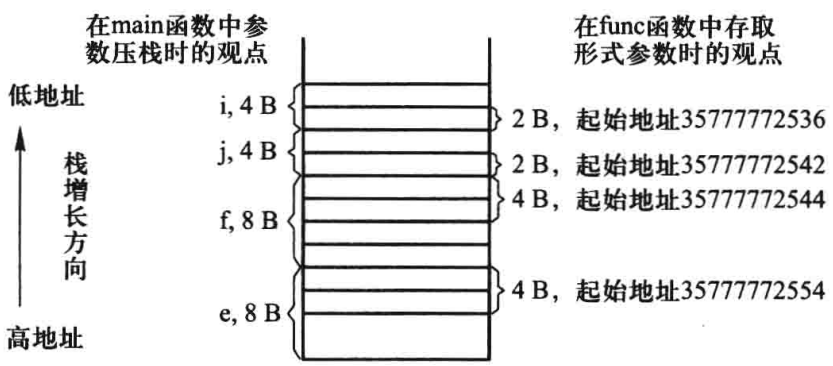


图 6.5 参数在栈中的情况

注意,现在有些编译器将需要进行类型转换的形式参数(类型是 char、short、float 等)另行分

配在局部数据区，当控制进入被调用过程时，首先将调用过程压栈的、需要进行类型转换的实在参数取出，把它们存入所分配的局部数据区存储单元，并同时完成必要的数据类型的转换。在这种情况下，不会出现本题的不一致问题。

另外，在 SPARC 工作站上，整型数据的高位存放在低地址，而低位存放在高地址。如果是 x86 机器，数据的高低位不是按这样的次序存放，那么也不会出现本题的不一致问题。

下面是某个编译器的类型提升函数，供读者理解用（备注：int 和 long 的大小一样）。

```
Type promote(Type ty) {
    switch(ty->op){
        case ENUM:
            return inttype;
        case INT:
            if (ty->size < inttype->size)
                return inttype;
            break;
        case UNSIGNED:
            if (ty->size < inttype->size)
                return inttype;
            break;
        case FLOAT:
            if (ty->size < doubletype->size)
                return doubletype;
    }
    return ty;
}
```

6.14 一个 C 文件 array.c 仅有下面两行代码：

```
char a[ ][4] = { "123", "456"};
char *p[ ]   = { "123", "456"};
```

在 x86/Linux 系统上编译生成的汇编代码列在下面（编译器的版本可见最后一行汇编代码），从中可以看出对数组 a 和指针 p 的存储分配是不同的。试依据这里的存储分配，为置了初值后的数组 a 和指针 p 写出类型表达式。

```
    .file "array.c"
    .globl a
        .data
        .type a, @object
```

```
        .size a, 8
a:
        .string "123"
        .string "456"
        .section .rodata
.LC0:
        .string "123"
.LC1:
        .string "456"
.globl p
        .data
        .align 4
        .type p, @object
        .size p, 8
p:
        .long .LC0
        .long .LC1
        .section.       note.GNU-stack, "", @progbits
        .ident      "GCC: (GNU) 3.3.5 (Debian 1:3.3.5-13)"
```

答 置了初值后的数组 a 和指针 p 的类型表达式分别是：

array(0..1, string)和 array(0..1, pointer(string))

或者写成：

array(0..1, array(0..3, char))和 array(0..1, pointer(array(0..3, char)))

***6.15** 一个 C 语言的函数如下：

```
func(c,l) char c;long l; {
    func(c,l);
}
```

在 x86/Linux 系统上编译生成的汇编代码如下(编译器版本请见汇编代码最后一行)：

```
        .file "parameter.c"
        .version "01.01"
gcc2_compiled.:
.text
        .align 4
.globl func
```

```
        .type func,@function
func:
        pushl %ebp              // 将老的基地址指针压栈
        movl %esp,%ebp          // 将当前栈顶指针作为基地址指针
        subl $4,%esp            // 分配空间
        movl 8(%ebp),%eax
        movb %al,-1(%ebp)
        movl 12(%ebp),%eax
        pushl %eax
        movsbl -1(%ebp),%eax
        pushl %eax
        call func
        addl $8,%esp
.L1:
        leave                   // 和下一条指令一起完成恢复老的基地址指针,将栈顶
        ret                     // 指针恢复到调用前参数压栈后的位置,并返回调用者
.Lfe1:
        .size func,.Lfe1-func
        .ident "GCC:(GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

请说明字符型参数和长整型参数在参数传递和存储分配方面有什么区别(小于长整型 size 的整型参数的处理方式和字符型参数的处理方式是一样的)。

答 在调用过程中,字符型实在参数被转换为长整型后再压栈。控制进入被调用过程时,从参数区把长整型数据取出,将其转换为字符型,存入为对应形式参数分配的空间(和局部变量一样都分配在局部数据区)。在被调用过程中,需要访问这样的形式参数时,使用为它们分配在局部数据区的数据单元。

对于长整型,没有上面那样的类型转换。另外在被调用过程中,在局部数据区没有为长整型形式参数另行分配空间,需要访问这样的形式参数时,直接访问参数区。

分析 对于题目中那几条没有加注释的指令,可以注释如下。

```
movl 8(%ebp),%eax          // 在参数区取参数 c
movb %al,-1(%ebp)          // 取其字节值,存入分配给 c 的存储单元
movl 12(%ebp),%eax         // 在参数区取参数 l
pushl %eax                 // 将 l 压栈
movsbl -1(%ebp),%eax       // 取 c 的值,并转换成长整型
pushl %eax                 // 将 c 压栈
call func                  // 调用函数
```

```
addl  $8,%esp                    // 栈顶指针恢复到参数压栈前的位置
```

6.16 从6.7题知道,C程序执行时只用到了控制链(动态链),不需要使用访问链(静态链)。为什么Pascal程序执行时需要使用访问链,而C程序不需要?

答 Pascal语言允许过程嵌套声明,这样,当Pascal程序执行时,除了访问局部数据和全局数据外,还可能访问非局部且非全局的数据。访问链用来帮助确定数据所在的活动记录在活动记录栈中的位置。

C语言的过程声明不能嵌套,这样C程序在执行时,只能访问全局变量(它们分配在静态数据区)和局部变量(它们分配在栈顶的那个活动记录中),因此C程序执行时无须使用访问链。

6.17 下面是求阶乘的Pascal程序。画出程序第三次进入函数factor时的活动记录栈和静态链。

```
program fact(input, output);
    var f, n : integer;
    function factor(n : integer) : integer;
        begin
            if n=0 then factor := 1
            else factor := n *factor(n-1)
        end;
    begin n := 5; f := factor(n); write(f)
    end.
```

答 见图6.6。

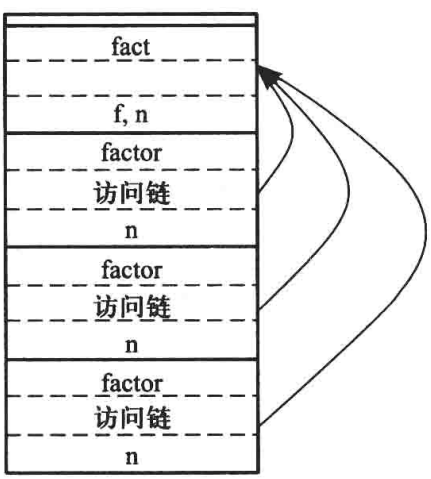


图6.6 活动记录栈和静态链

*__6.18__ 在下面假想的程序中,第(11)行语句f := a调用函数a,a传递函数addm作为返回值。

(a) 画出该程序执行的活动树。

(b) 假定非局部名字使用静态作用域,为什么该程序在栈式分配情况下不能正确工作?

(c) 在堆分配策略下,该程序的输出是什么?

```
(1)    program ret (input, output);
(2)      var f: function (integer): integer;
(3)      function a: function (integer): integer;
(4)        var m: integer;
(5)        function addm (n: integer): integer;
(6)          begin return m+n end;
(7)        begin m:= 0; return addm end;
(8)      procedure b (g: function (integer): integer);
(9)        begin writeln (g(2)) end;
(10)   begin
(11)     f := a; b(f)
(12)   end.
```

答 (a) 该程序执行的活动树见图 6.7。

(b) 在执行第(6)行的 m+n 时,由于分配给 m 的存储空间已经被释放,因此该计算无法正确完成。

(c) 在堆分配策略下,该程序的输出是 2。

分析 (a) 画活动树前,先要搞清楚函数 addm 是在什么时候执行的。在执行第(11)行的第一个语句时,函数变量 f 的值是函数 addm。在执行该行的第二个语句 b(f)时,通过实在参数 f 和形式参数 g 的结合,使得函数变量 g 的值是函数 addm。这样在执行第(9)行 writeln (g(2))时,执行函数 addm,也就是说函数 addm 是在过程 b 中被调用的。

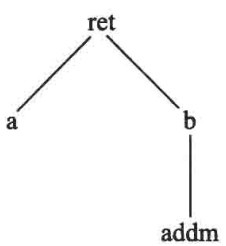


图 6.7 活动树

(b) 按照静态作用域,第(6)行 m+n 中的 m 引用对应于外围函数 a 第(4)行的 m 声明。从图 6.7 的活动树可以看出,在执行函数 addm 时,活动记录栈中依次只有 addm、b 和 ret 的活动记录,a 的活动记录已经释放,因此 m+n 的计算无法正确完成。

(c) 在堆分配策略下,活动记录的释放不受栈式分配的限制。在执行 m+n 时,只要 a 的活动记录还没有释放,程序的输出就应该是 2。

***6.19** 为什么 C 语言允许函数类型作为函数的返回值类型,而 Pascal 语言却不允许?

答 对于 Pascal 语言来说,如果允许一个函数的返回值是函数,那么有可能出现这样的情况:作为返回值的函数执行时,需访问的非局部非全局的数据已经不存在,习题 6.18 已经说明了这一点。所以 Pascal 语言不允许函数类型作为函数的返回值类型。

C 语言的函数声明不能嵌套,因此 C 的函数无论在什么情况下被激活,在执行它的代码时要访问的数据都会分成以下两种情况。

1. 非静态局部变量(包括形式参数),它们分配在活动记录栈顶部的那个活动记录中。

2. 外部变量(包括定义在其他源文件中的外部变量)和静态的局部变量,它们都分配在静态数据区。

因此 C 语言不会出现 Pascal 语言碰到的这种困难。

6.20 一个 C 语言程序如下:

```
int n;
int f(g) int g(); {
    int m;
    m = n;
    if(m == 0) return 1;
    else {
        n = n-1; return m * g(g);
    }
}
main() {
    n = 5; printf("%d factorial is %d\n", n, f(f));
}
```

该程序在 Ubuntu 12.04.2 LTS (GNU/Linux 3.2.0-42-generic x86_64)系统上,经过编译器 GCC:(Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3 编译后,运行结果不是所期望的

```
5 factorial is 120
```

而是

```
0 factorial is 120
```

试说明原因。

答 因为对于 C 语言的函数调用,实在参数是逆序计算进栈的。因此 printf 调用首先计算 f(f),得到结果 120,同时在这个计算过程中,n 的值被减为 0。因此再把参数 n 的值进栈时,得到的是 0。

***6.21** 下面程序在早先的 SPARC/SunOS 系统上运行时陷入死循环,试说明原因。如果将第 5 行的 long *p 改成 short *p,并且将第 15 行 long k 改成 short k,则 loop 中的循环体执行一次后便停止了。试说明原因。

```
main() {
    addr();
```

```
    loop();
}
long *p;
loop() {
    long i,j;
    j=0;
    for(i=0;i<10;i++){
        (*p)--;
        j++;
    }
}
addr() {
    long k;
    k=0;
    p=&k;
}
```

答 程序运行时陷入死循环的原因是由于 p 指向分配给 i 的存储单元引起的。循环体执行一次便停止是由于 p 指向分配给 i 的低位字节引起的。

分析 由于 C 语言的实现采用的是栈式分配，使得函数 addr 和函数 loop 的活动记录先后从同样的存储单元开始分配，导致长整数 k 和 i 先后分配到同样的存储单元中，因此 p 指向分配给 i 的存储单元。这样 loop 函数中的循环就相当于下面的程序段，所以程序陷入死循环。

```
for(i=0;i<10;i++){
    i--;
    j++;
}
```

SPARC/SunOS 系统中整数的存储方式是低地址放高位字节，而高地址放低位字节。另外，活动记录栈是从高地址向低地址方向增长的。这样，当 k 改为短整型且 p 改为短整型指针后，p 指向由 i 的两个低位字节组成的短整数。执行(*p)--使得这两个字节构成的短整数等于-1。而以整个 4 个字节看 i 时，是 65 535，远远大于 10。所以循环体执行一次便停止。

如果这个程序在 x86/Linux 系统上运行，则不会陷入死循环。由于 x86 机器上低地址放低位字节，当 k 改为短整型且 p 改为短整型指针后，循环体并非执行一次便停止，但也不会陷入死循环。读者可以自行分析。

***6.22** 一个 C 语言程序如下：

```
main() {
```

```
    func();
    printf("Return from func\n");
}
func() {
    char s[4];
    strcpy(s, "12345678");
    printf("%s\n",s);
}
```

在 Ubuntu 12.04.2 LTS (GNU/Linux 3.2.0-42-generic x86_64) 系统上,该程序经过编译器 GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3 编译后,运行结果如下:

```
12345678
Return from func
```

若将语句 strcpy(s,"12345678")改成 strcpy(s,"123456789"),目标程序 a.out 的运行结果及运行流产的报告如下:

```
123456789
* * * stack smashing detected * * * : a.out terminated
========Backtrace:==========
注:Backtrace 的具体内容略去
========Memery map:========
注:Memery map 的具体内容略去
Aborted (core dumped)
```

试分析为什么会出现这样的运行情况。

答 该程序流产的原因是,func 函数中的串复制出现数组越界,引起 func 函数活动记录的内容遭到破坏,活动记录中保存的控制链和/或返回地址遭到破坏,因此报告活动记录栈被击毁。

使用 strcpy(s, "12345678")时其实也出现了数组越界,但这个越界并没有破坏 func 函数活动记录中的控制链和返回地址,因此它能继续执行,返回 main 函数,输出 Return from func,再继续返回操作系统。能返回 main 函数,则说明 func 函数活动记录中的返回地址完好;能返回操作系统,则说明 func 函数活动记录中的控制链完好。

分析 可以用该程序在另一个系统上的运行结果来支持这个答案。在 x86/Linux 系统上,经一个较低版本的 gcc 编译器编译后,运行结果如下:

```
12345678
Return from func
Segmentation fault (core dumped)
```

对于这个系统,可以用主教材上图 6.11 活动记录的内容来解释。func 函数中的串复制引起

控制链被破坏,返回地址也被改写了一个字节,写成0(串结束标记)。但是这个字节被重写并没有造成灾难,所以还可以从 func 函数返回到 main 函数。当试图从 main 函数返回时出错。出错的原因是 main 函数活动记录的基地址指针未被正确恢复,因为 func 函数活动记录中所保存的控制链遭到了破坏。

若把串复制语句改成

```
strcpy(s, "123456789");
```

程序的输出就成了

```
123456789
Segmentation fault (core dumped)
```

即不能正确返回到 main 函数中继续执行,因为这时保留在 func 函数活动记录中的返回地址被重写了2个字节,使得返回地址被修改了。

6.23 一个C语言程序如下:

```
int fact(i) int i;{
    if(i==0)
        return 1;
    else
        return i * fact(i-1);
}
main() {
    printf("%d\n", fact(5));
    printf("%d\n", fact(5,10,15));
    printf("%d\n", fact(5.0));
    printf("%d\n", fact());
}
```

该程序在 x86/Linux 系统上,用编译器 GCC:(GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)编译后,运行结果如下:

```
120
120
1
Segmentation fault (core dumped)
```

请解释下面问题:

(a)对第二个 fact 调用,结果为什么没有受参数过多的影响?

(b)对第三个 fact 调用,为什么用浮点数 5.0 作为参数时结果变成 1?

(c)对第四个 fact 调用,为什么没有提供参数时会出现 Segmentation fault 错误?

答 (a) 参数表达式逆序计算并进栈,fact 能够取到第一个参数。

(b) 参数 5.0 转换成双精度数进栈,占 8 B,其低地址的 4 B 看成整数时正好是 0 或 1。

(c) 由于没有提供参数,fact 把老 ebp(控制链)(main 的活动记录中保存的 ebp)当成参数,它一定是一个很大的整数,使得活动记录栈溢出。

分析 在 Ubuntu 12.04.2 LTS (GNU/Linux 3.2.0-42-generic x86_64)系统上,该程序经过编译器 GCC:(Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3 编译后,运行时第四次调用 fact 的结果也是 1。这是因为这个编译器对活动记录的安排(活动记录中有些什么数据域以及它们的摆放次序)已经大不一样,使得第四次 fact 调用所取到的参数也是 0 或 1,因而结果是 1。

6.24 下面是一个 C 语言程序:

```
long f1(i) long i; {
    return(i * 10);
}
long f2(long i) {
    return(i * 10);
}
main() {
    printf("f1 = %d, f2 = %d\n", f1(10.0), f2(10.0));
}
```

其中函数 f1 和 f2 仅形式参数的描述方式不一样。该程序在 x86/Linux 系统上,用编译器 GCC:(GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)编译后,运行结果如下:

```
f1 = 0, f2 = 100
```

请解释为什么用同样的实在参数调用这两个函数的结果会不一样。

答 历史上,C 语言中有参函数定义的一般形式是:

类型标识符 函数名(形式参数列表)

形式参数类型声明

对于这种形式的声明,C 语言编译器是不做实在参数和形式参数的个数和类型是否一致的检查的。如本题中函数 f1 的声明是这种形式。

现在,ANSI 新标准允许使用另一种方法来声明形式参数,即在函数名后的括号中列出形式参数的同时,声明形式参数的类型。本题中函数 f2 的声明是这种形式。C 语言编译器对于这种形式的函数声明是要进行实在参数和形式参数的个数和类型是否一致的检查的。已经出现几个有关这个区别的习题了。

在编译函数调用 f2(10.0)时,会把浮点数 10.0 转换成整数 10,并产生将整数 10 压栈的指令。因此函数调用 f2(10.0)的结果是 100。

而在编译函数调用 f1(10.0)时,会把浮点数 10.0 转换成双精度型(参见 6.13 题),并产生将该双精度型数压栈的指令。双精度数占 8 B,其低地址的 4 B 被看成整数时正好是 0。因此函数调用 f1(10.0)的结果是 0(参见 6.23 题)。

6.25 下面 Pascal 程序的输出是什么?若用动态作用域代替静态作用域,程序的输出又是什么?

```
program dynamic(input, output);
    var r : real;
    procedure show;
        begin write(r:5:3) end;
    procedure small;
        var r : real;
        begin r := 0.125; show end;
    begin
        r := 0.25;
        show; small; writeln;
        show; small; writeln
    end.
```

答 该程序的输出如下:

0.250 0.250

0.250 0.250

若用动态作用域代替静态作用域,该程序的输出如下:

0.250 0.125

0.250 0.125

分析 主要考虑动态作用域的情况。在静态作用域下,确定非局部名字对应于哪个声明是按静态嵌套的层次逐层向外查找的。而在动态作用域下,确定非局部名字到存储单元的绑定是按过程激活的次序,在当前活动着的过程中由后向前查找的。这样,从图 6.8 的活动树就不难确定该程序在动态作用域下的结果了。

show 过程本身没有局部变量 r。在第一次调用 show 时,从活动树上可以看出,它访问 dynamic 中的 r,因此输出 0.250。在第二次调用 show 时,从活动树上可以看出,它访问 small 中的 r,因为它首先找到 small,而 small 中有 r 的声明,因此输出 0.125。第三和第四次调用的情况分别同第一和第二次。show、small 和 writeln 语句重复一遍的目的是提请注意,第三次调用 show 时,它不受曾经激活过而现已终止的 small 的影响。

这样可以看出动态作用域的一个性质:程序运行时,一个名字 a 实施它的影响,直到含 a 声明的一个过程开始执行时暂停;此过程停止时,其影响恢复。

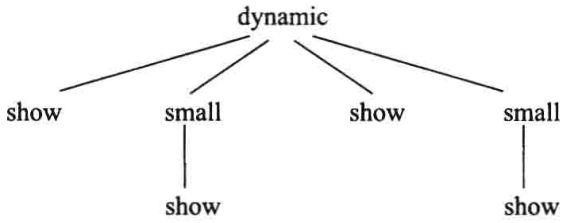


图 6.8　程序 dynamic 的活动树

6.26　一个过程作为实在参数传递的时候，它被激活时的计算环境有三种可能的规定。第一种是使用词法环境，即该过程激活时的计算环境是依据其定义点的静态作用域得到。Pascal 和 C 都采用这种方式。第二种是使用传递环境，即该过程激活时的计算环境是依据其作为实在参数的调用点的静态作用域得到。第三种是使用活动环境，即该过程激活时的计算环境是依据其激活点的静态作用域得到。

以下面的 Pascal 程序为例来解释。考虑该程序第(11)行 f 作为实在参数传递的情况，对应上面三种规定，第(8)行的非局部变量名 m 分别在第(6)、(10)和(3)行的作用域内。在这三种情况下，该程序的输出分别是什么？

```
(1)   program param(input, output);
(2)     procedure b(function h(n: integer):integer);
(3)         var m: integer;
(4)         begin m := 3; writeln(h(2)) end;        {b}
(5)     procedure c;
(6)         var m: integer;
(7)         function f(n: integer): integer;
(8)             begin f := m+n end;                 {f}
(9)         procedure r;
(10)            var m: integer;
(11)            begin m := 7; b(f) end;             {r}
(12)        begin m := 0; r end;                    {c}
(13)    begin
(14)        c
(15)    end.
```

答　从题目的叙述中应该很快可以得出，对于这三种方式，程序的输出依次是 2,9 和 5。

***6.27**　通常，函数调用的返回值是简单类型时，用寄存器传递函数值。当返回值是结构体类型时需要采用其他的方式。下面是一个 C 语言文件和它在 x86/Linux 上编译(编译器版本见汇编代码最后一行)生成的汇编代码。(备注，该汇编码略经修改，以便于阅读。该修改没有影

响结果)

(a) 请你分析这些代码,总结出函数返回值是结构体类型时,返回值的传递方式。

(b) 若 m 函数的语句 s = f(10)改成 s.i = f(10).i + f(20).i + f(30).i,你认为 m 函数的局部存储分配应该怎样修改,以适用该语句的计算?

源文件 return.c 的内容如下:

```
typedef struct{long i;}S;
S f(k) long k; {
    S s;
    s.i =k;
    return s;
}
m() {
    S s;
    s.i = 20;
    s = f(10);
}
```

汇编文件 return.s 的内容如下:

```
    .file"return.c"
    .text
.globl f
    .typef, @function
f:
    pushl    %ebp
    movl     %esp, %ebp
    subl     $4, %esp
    movl     8(%ebp), %eax
    movl     12(%ebp), %edx
    movl     %edx, -4(%ebp)
    movl     -4(%ebp), %edx
    movl     %edx, (%eax)
    leave
    ret
    .sizef, .-f
.globl m
    .typem, @function
```

```
m:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        movl    $20, -4(%ebp)
        leal    -4(%ebp), %eax
        pushl   $10
        pushl   %eax
        call f
        addl    $8, %esp
        leave
        ret
        .sizem, .-m
        .section.  note.GNU-stack,"",@progbits
        .ident    "GCC:(GNU) 3.3.5 (Debian 1:3.3.5-13)"
```

答 (a) 调用函数 m 把实在参数压栈后,将存放返回值的地址压栈,然后调用 f。f 在返回前,将结果送到 m 提供的存放返回值的地址。

(b) m 要为三次 f 调用分配存放结果值的空间,并且这三次存放结果值的空间不重叠。在每次调用 f 时,将相应的存放返回值的地址压栈。

***6.28** 一个 C 语言的文件如下:

```
func(long i, long j, long k) {
    k = (i + j)-(i-j-f(k));
}
```

经 x86/Linux 系统上编译(编译器版本见汇编代码最后一样)得到的汇编代码分两列在下面给出:

```
        .file   "call.c"
        .text
.globl func
        .type   func, @function
func:
        pushl   %ebp
        movl    $esp, %ebp
        pushl   %esi
        pushl   %ebx
        subl    $4, %esp
        movl    12(%ebp), %eax
        movl    8(%ebp), %esi
        addl    %eax, %esi
        movl    12(%ebp), %edx
        movl    8(%ebp), %eax
        movl    %eax, %ebx
        subl    %edx, %ebx
        movl    16(%ebp), %eax
```

```
movl    %eax, (%esp)
call    f
subl    %eax, %ebx
movl    %ebx, %eax
subl    %eax, %esi
movl    %esi, %eax
movl    %eax, 16(%ebp)
addl    $4, %esp
popl    %ebx
popl    %esi
popl    %ebp
ret
.sizefunc, .-func
.section    .note.GNU-stack, "",@progbits
.ident    "GCC: (GNU)  3.4.6"
```

请根据上述汇编代码进行总结:为适应函数调用引起的跨函数执行,该编译器在寄存器的值的保护方面有些什么约定?

答 1. esp 是由调用者和被调用者共同维护的栈顶地址寄存器。

2. ebp 是活动记录的基地址寄存器,各函数在入口和出口对它原来的值进行保存和恢复。即 ebp 是由被调用者保护的寄存器。

3. 一个函数代码若使用 esi 和 ebx 这样的寄存器来保存表达式计算的中间结果,则在入口和出口对它原来的值进行保存和恢复。即 esi 和 ebx 是由被调用者保护的寄存器。

4. 一个函数使用 eax 和 edx 这样的寄存器时,不进行上述方式的保存与恢复。即 eax 和 edx 是由调用者保护的寄存器。

5. 函数的返回值存于 eax 中。

为每个函数产生目标代码时应该按上述准则来分配和使用寄存器。

6.29 下面是一个以函数作为参数的 C 语言程序。

```
int f(int g()) {return g(g);}
main(){f(f);}
```

(a) 对于函数类型的形式参数,调用时的参数传递传什么?

(b) 该程序执行时,系统报告 Segmentation fault,请回答是什么原因。

答 (a)函数作为参数时,只需要传递函数代码的入口地址。

(b)该程序会不断地递归调用,导致因活动记录栈溢出而报错。

6.30 有人认为,下面 C 程序中结构体类型 record 的定义方式可用来动态生成其中 a 数组的大小不一样的结构体,以适应某些编程场合的需要。你认为这样的程序能够通过 C 编译器的类型检查吗? 请说明理由。

```
#include <malloc.h>
typedef struct {double r; int n; float a[];} record;
main() {
    record * p;
```

```
    p = malloc(sizeof(record) + sizeof(float) * 5);
    p->n = 5; p->a[4] = 100.0;…
}
```

答 这是一种编程技巧。编译器不会认为这其中有什么类型错误。对于含 malloc 调用的那个语句,只要赋值号的左边是指针类型的左值表达式就可以了。对于赋值语句 p->a[4] = 100.0,由于 C 编译器不进行下标表达式是否越界的检查,因此也不会报告错误。

6.31 (a) Java 语言的编译器通常把数组分配到堆上。Java 数组一般不能静态确定大小,这不应该作为其将数组分配到堆上的原因,因为主教材中图 6.12 给出了一种将不能静态确定大小的数组动态地分配到活动记录栈上的方法。那么,Java 数组一般不能分配在活动记录栈上的原因是什么呢?

(b) 将数组分配到活动记录栈上和分配到堆上会给程序运行带来什么区别?

答 (a) Java 的数组是一种引用类型,数组变量之间的赋值 b = a 会导致 b 也指向 a 所指向的数组。由此,在一个方法 m 中,不管是用像 int arr[ ] = {1, 2, 3, 4}这样的方式还是用像 int arr[ ] = new int[4]这样的方式创建数组,在 m 的执行结束后该数组都还有可能被引用,因此不能将其分配到活动记录栈上。

(b) 分配到活动记录上,则可及时回收数组占用的空间并且回收开销小。分配到堆上,则在垃圾收集器工作时才回收并且回收开销大。

***6.32** Java 语言的实现通常把对象和数组都分配到堆上,把指向它们的指针分配到栈上,依靠运行时的垃圾收集器来回收堆上那些从栈不可达的空间(垃圾)。这种方式提高了语言的安全性,但是增加了运行开销。编译时能否采用一些技术,来降低垃圾收集器的运行开销?概述你的方案。

答 从如下两方面考虑。

1. 减少垃圾。用程序分析技术确定数据的生存期是否超过它的静态作用域所对应的生存期,若没有超过则将这样的数据分配到栈上,而不必分配到堆上。

2. 尽量静态确定垃圾。用程序分析技术把能静态确定的垃圾都标注出来,可使运行时对它们回收开销很小。

第7章　中间代码生成

一、主要知识点

主教材第7章主要应掌握下面一些内容。

1. 中间代码的几种形式，它们之间的相互转换。
2. 符号表的组织和作用域信息的保存。
3. 数组元素地址计算的翻译方案，布尔表达式的两种不同实现方式。
4. 赋值语句和各种控制流语句的中间代码格式和生成中间代码的翻译方案。
5. 过程调用的中间代码格式和生成中间代码的翻译方案。

除了数组元素地址计算的翻译方案比较难理解外，本章没有什么困难的地方。

二、习题精选与解析

7.1　把算术表达式-(a + b) *(b + c)翻译成(a)语法树、(b)有向无环图 DAG、(c)后缀表示及(d)三地址代码。

答　(a) 该表达式的语法树见图 7.1(a)。

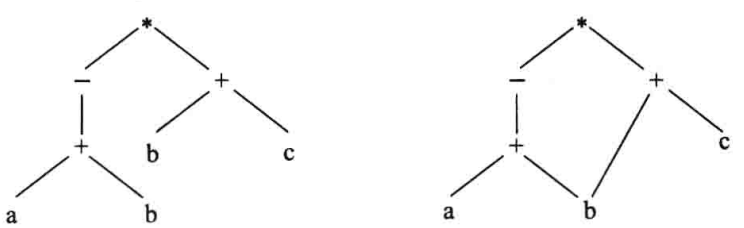
(b) 有向无环图见图 7.1(b)。

(c) 后缀表示是 a b + -b c + *(如果“-”同时作为二元算符的话，那么在后缀式中应当要考虑区分它们)。

(d) 三地址代码如下：

$t_1 := a + b$

$t_2 := -t_1$



(a) 语法树　　(b) DAG

图 7.1　语法树和有向无环图

$t_3 := b + c$

$t_4 := t_2 * t_3$

7.2 为下面文法写一个翻译方案，它完成符号表的建立（包括作用域信息的保存）、一般变量和记录域名的相对地址的分配（假定过程不能递归。为简单起见，过程不能递归和记录不能有过程域的检查不必写在本翻译方案中）。

$P \rightarrow D$

$D \rightarrow D;\ D \mid \mathbf{id} : T \mid \mathbf{procid};\ D;\ S$

$T \rightarrow \mathbf{integer} \mid \mathbf{real} \mid \mathbf{record}\ D\ \mathbf{end}$

答 翻译方案如下，其中增加标记非终结符 M 和 N。

$P \rightarrow M\ D$ { *addwidth* (*top* (*tblptr*), *top* (*offset*));
pop (*tblptr*); *pop* (*offset*); }

$M \rightarrow \varepsilon$ { t = *maketable* (*nil*); *push* (t, *tblptr*); *push* (0, *offset*); }

$D \rightarrow D_1;\ D_2$

$D \rightarrow \mathbf{procid};\ N\ D_1;\ S$ { t = *top* (*tblptr*); *addwidth* (t, *top* (*offset*));
pop (*tblptr*); *pop* (*offset*);
enterproc (*top*(*tblptr*), **id**. *name*, t); }

$N \rightarrow \varepsilon$ { t = *mktable* (*top* (*tblptr*));
push (t, *tblptr*); *push* (0, *offset*); }

$D \rightarrow \mathbf{id} : T$ { *enter* (*top*(*tblptr*), **id**. *name*, T. *type*, *top* (*offset*));
top (*offset*) = *top* (*offset*) + T. *width*; }

$T \rightarrow \mathbf{integer}$ { T. *type* = *integer*; T. *width* = 4; }

$T \rightarrow \mathbf{real}$ { T. *type* = *real*; T. *width* = 8; }

$T \rightarrow \mathbf{record}\ M\ D\ \mathbf{end}$ { T. *type* = *record* (*top* (*tblptr*));
T. *width* = *top* (*offset*); *pop* (*tblptr*); *pop* (*offset*); }

其中使用的几个过程解释如下。

1. *mktable* (*previous*)：建立新的符号表，返回新符号表指针。变元 *previous* 指向先前建立的符号表。

2. *enter* (*table*, name, type, offset)：在 *table* 指向的符号表中为名字 *name* 建立新条目。把类型 *type* 和相对地址 *offset* 置于该条目的域中。

3. *addwidth* (*table*, *width*)：把符号表 *table* 中的所有条目的累加宽度记录在该符号表的首部。

4. *enterproc* (*table*, *name*, *newtable*)：为过程 *name* 在 *table* 指向的符号表中建立新条目，变元 *newtable* 指向过程 *name* 本身的符号表。

分析 为便于理解，图 7.2 给出该翻译方案为下面程序生成的符号表。为直观起见，该例子程序用具体语法，而不是题目所给的抽象文法。在图 7.2 给出的各个符号表中，首部的整数表示

给过程或记录分配的字节数,过程中变量或记录中域名右边的整数是指给它们的分配相对地址。

```
program sample (input, output);
    var x : integer; y : real;
    procedure sub;
        var x : integer;
            a : record
                    a : integer;
                    b : record
                            b : integer; c : real
                        end
                end;
```

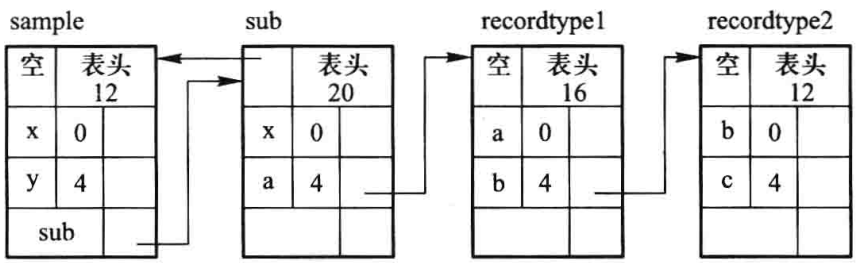


图 7.2 符号表

7.3 若 7.2 题的 Pascal 程序中,在子程序 sub 中有下面的程序段。试说明编译器如何用 7.2题为声明语句建立的符号表来查下面程序段中名字的引用性出现对应于哪个声明。

```
(1) with a do begin
(2)     x := a;
(3)     with b do begin
(4)         a := b;
(5)         y := c
(6)     end
(7) end
```

答 从 7.2 题可知,每个过程的局部名字构成一张符号表,另外,每个记录类型的域名构成一张符号表。可以作一种比较直观的解释,假定分析到程序某一点时,当前能访问的符号表的指针依次存在一个栈中。那么在进入第(1)行时,从栈底到栈顶的符号表指针依次是 sample 和 sub 的指针(严格说,栈底还应有标准标识符表的指针)。遇到标识符的引用性出现时,从栈顶到栈底依次查各表,寻找其定义。

这样,对于第(1)行的 a,在 sub 的表中确定它是记录类型 recordtype1 的变量,with 语句要求把 recordtype1 符号表的指针压入栈。对于第(2)行,查表可以知道,x 的声明在 sub 中,而 a 是 recordtype1 记录类型的一个域名。对于第(3)行,查表知 b 是 recordtype1 记录类型的一个域名,

而该域是记录类型,这样把 recordtype2 的符号表的指针压入栈。对于第(4)行,查表可以知道,a 是 recordtype1 记录类型的一个域名,而 b 是 recordtype2 记录类型的一个域名。对于第(5)行,查表可以知道,y 的声明在 sample 中,而 c 是 recordtype2 记录类型的一个域名。

到第(6)行,从栈中退出 recordtype2 的符号表的指针。到第(7)行,从栈中退出 recordtype1 的符号表的指针。符号表指针栈恢复到进入第(1)行 **with** 语句前的情况。

分析 从这里还可以看出,如果不用 **with** 语句,比方说出现 a. a,编译器首先在 sub 的表中知道 a. a 的前一个 a 是 recordtype1 记录类型的变量。那么,对于后一个 a,只查 recordtype1 记录类型的名字表,如果查不到,那就报告错误。

7.4 布尔表达式的文法如下:

$E \rightarrow E$ **or** $E \mid E$ **and** $E \mid$ **not** $E \mid (E) \mid$ **id** < **id** $\mid$ **true** $\mid$ **false**

请写出一个语法制导定义,该语法制导定义产生短路计算的三地址代码,并且把 $E \rightarrow$ **id** < **id** 翻译成 **if** $\mathbf{id}_1 \geqslant \mathbf{id}_2$ **goto**…的形式。

答 语法制导定义如下,其中“ ‖ ”是字符串并置算符。

$E \rightarrow E_1$ **or** E_2	$E_1.true = E.true$;
	$E_1.false$ = newlable;
	$E_2.true = E.true$;
	$E_2.false = E.false$;
	$E.code = E_1.code \parallel gen('goto', E_1.true) \parallel gen(E_1.false, ':') \parallel E_2.code$
$E \rightarrow E_1$ **and** E_2	$E_1.true$ = newlable;
	$E_1.false = E.false$;
	$E_2.true = E.true$;
	$E_2.false$ = E.$false$;
	$E.code = E_1.code \parallel gen(E_1.true, ':') \parallel E_2.code$
$E \rightarrow$ **not** E_1	$E_1.true = E.false$;
	$E_1.false$ = E.$true$;
	$E.code = E_1.code$
$E \rightarrow (E_1)$	$E_1.true = E.true$;
	$E_1.false = E.false$;
	$E.code = E_1.code$
$E \rightarrow \mathbf{id}_1 < \mathbf{id}_2$	$E.code = gen('if', \mathbf{id}_1.place, '\geqslant', \mathbf{id}_2.place, 'goto', E.false)$
$E \rightarrow$ **true**	$E.code = gen('goto', E.true)$
$E \rightarrow$ **false**	$E.code = gen('goto', E.false)$

分析 对于产生式 $E \rightarrow \mathbf{id}_1 < \mathbf{id}_2$,这里的语义规则实际上是产生“假转”指令,指的是,若条

件不能成立就转走，否则就执行它的后继指令。

这里大部分产生式的语义规则都能理解，可能需要说明的是 $E \rightarrow E_1$ **or** E_2 的最后一条规则

$$E.code = E_1.code \parallel gen('goto', E_1.true) \parallel gen(E_1.false, ':') \parallel E_2.code$$

由于 E_1 的代码后面跟的是 E_2 的代码，它是 E_1 为假时要执行的代码，而条件判断 $\mathbf{id_1 < id_2}$ 的代码处理成自己的后继代码应该是为真时要执行的代码，为解决这个矛盾，需要在 E_1 的代码后面加上一条语句 goto $E_1.true$。当然，这条语句有可能是多余的，比方若 E_1 就是 true，那么它的代码就是 goto $E_1.true$，后面再跟 goto $E_1.true$ 显然是多余的。但是若 E_1 是 $\mathbf{id_1 < id_2}$，那么该语句就是必不可少的。表达式 E_1 and E_2 没有这个问题，因为 E_1 为真时正好就是要执行 E_2。

***7.5** 下面的 C 语言程序

```
main( ) {
    int i,j;
    while ( (i || j) && (j > 5) ) {
        i = j;
    }
}
```

在 x86/Linux 系统上编译生成的汇编代码如下(编译器版本见汇编代码最后一行)：

```
    .file "bool.c"
    .version "01.01"
gcc2_compiled.:
.text
    .align 4
.globl main
    .type main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    subl $8,%esp
    nop
    .p2align 4,,7
.L2:
    cmpl $0,-4(%ebp)
    jne .L6
    cmpl $0,-8(%ebp)
    jne .L6
```

```
        jmp .L5
        .p2align 4,,7
.L6:
        cmpl $5,-8(%ebp)
        jg .L4
        jmp .L5
        .p2align 4,,7
.L5:
        jmp .L3
        .p2align 4,,7
.L4:
        movl -8(%ebp),%eax
        movl %eax,-4(%ebp)
        jmp .L2
        .p2align 4,,7
.L3:
.L1:
        leave
        ret
.Lfe1:
        .size main,.Lfe1-main
        .ident "GCC:(GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

在该汇编代码中有关的指令后加注释,将源程序中的操作和生成的汇编代码对应起来,以判断确实是用短路计算来完成布尔表达式计算的,并说明代码生成是否完全遵照 7.4 题答案的语法制导定义。

答 在有关的指令后加注释如下:

```
main:
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp
        nop
        .p2align 4,,7
.L2:
        cmpl $0,-4(%ebp)          // 计算 i 的布尔值
        jne .L6                   // 真转 L6
```

```
        cmpl  $0,-8(%ebp)          // 计算 j 的布尔值
        jne . L6                   // 真转 L6
        jmp . L5                   // ( i || j )为假时转 L5
        . p2align 4,,7
    . L6:
        cmpl  $5,-8(%ebp)          // 计算表达式 j > 5 的布尔值
        jg . L4                    // 真转 L4
        jmp . L5                   // 假转 L5
        . p2align 4,,7
    . L5:
        jmp . L3                   // 表达式( i || j ) && (j > 5)为假时转 L3,它和
        . p2align 4,,7             // while 结构的翻译有关
    . L4:
        movl -8(%ebp),%eax
        movl %eax,-4(%ebp)
        jmp . L2
    . L3:
```

首先可以看出,布尔表达式是用短路计算方式来完成计算的。其次可以看出,代码生成不是遵照 7.4 题答案的语法制导定义,而是遵照 7.4 题分析的最后一段提出的语法制导定义,但并非完全一样。若完全一样,那么从 L2~L6 之间的代码如下。

```
    . L2:
        cmpl  $0,-4(%ebp)          // 计算 i 的布尔值
        jne . L6                   // 真转 L6
        jmp . L11                  // 假转 L11
    . L11:
        cmpl  $0,-8(%ebp)          // 计算 j 的布尔值
        jne . L6                   // 真转 L6
        jmp . L12                  // 假转 L12
    . L12:
        jmp . L5                   // ( i || j )为假时转 L5
        . p2align 4,,7
    . L6:
```

这里冗余的无条件转移指令也许是在代码生成的窥孔优化时删除的,或者是用更细致一些的语法制导定义,使得这样的指令根本不能生成。

*7.6 Pascal 语言的标准将 for 语句

for v := initial **to** final **do** stmt

定义成和下面的代码序列有同样的含义：

```
begin
    t1 := initial;
    t2 := final;
    if t1 <= t2 then begin
        v := t1;
        stmt;
        while v <> t2 do begin
            v := succ(v);
            stmt
        end
    end
end
```

请为 for 语句设计一种合理的中间代码结构，并写出产生中间代码的翻译方案。

答 Pascal 语言 for 语句的中间代码结构如下：

```
    t1 := initial
    t2 := final
    if t1 > t2 goto L1
    v := t1
L2: stmt
    if v = t2 goto L1
    v := v + 1
    goto L2
    L1:
```

用 V、S 和 E 分别表示变量、语句和表达式，以符合写翻译方案时的习惯。一个比较直观的翻译方案如下：

$S \rightarrow$ **for** $V := E_1$ **to** E_2 **do** S_1 { $S.next = newlable$;
$S_1.begin = newlable$;
$S_1.next = newlable$;
$S.code = E_1.code \parallel E_2.code \parallel$
$gen('if', E_1.place, '>', E_2.place, 'goto', S.next) \parallel$
$gen(V.place, ':=', E_1.place) \parallel$
$gen(S_1.begin, ':') \parallel S_1.code \parallel gen(S_1.next, ':') \parallel$

gen ('if', $V.place$, '=', $E_2.place$, 'goto', $S.next$) ||
gen ($V.place$, ':=', $V.place$, '+ 1') ||
gen ('goto', $S_1.begin$)}

分析 设计中间代码结构时要注意的是 initial、final 和 stmt 在中间代码结构中仅出现一次，以免产生的中间代码序列中有它们的代码重复出现。对于 initial 和 final 来说，更重要的原因是，从上面的语义解释知道，它们只在循环前计算一次，若在代码中多次出现就可能计算多次。

另一点要注意的是它们三者的次序不应该颠倒，因为在一般情况下，initial、final 和 stmt 的代码在中间代码序列中的次序应该和它们在源程序中的次序一致，以利于中间代码的生成。这两点是容易忽略的地方。也许有人会问，题目给出的 for 语句的语义解释中 stmt 就出现了两次。但这是语义解释，它考虑的是给出准确的含义，而中间代码生成应该考虑代码的长度和效率等问题。那么，是否有 stmt 仅出现一次的语义解释呢？也许你会想到下面这样的解释：

```
begin
    t1 := initial;
    t2 := final;
    v := t1;
    while v <= t2 do begin
        stmt;
        v := succ(v)
    end
end
```

很可惜，这个解释是有问题的。如果 final 刚好等于 MAXINT(语言的实现所允许的最大整数)，那么就会出现求 MAXINT 的后继，这会导致一个越界错误。所以 Pascal 语言标准所给的解释是经过认真考虑的。

再对上面的翻译方案做些解释。

1. 对应 t_1 := initial 和 t_2 := final 的中间代码分别是 $E_1.code$ 和 $E_2.code$，属性 $E_1.place$ 和 $E_2.place$就是 initial 和 final 存放的地方。

2. *newlable* 是申请一个新标号的函数。

3. S 的 *next* 属性表示 S 的代码执行后应该执行的第一条中间代码语句的标号。

4. 和所给出的中间代码结构相比，仅增加了一个标号 $S_1.next$。这是因为语句 S_1中可能有跳转语句转到 S_1的后继。

为提高编译速度，实际的编译器不会把生成的代码放在属性中。更接近实际编译器的一个翻译方案如下，它是逐条语句地生成中间代码，把它们放入中间代码序列中。

先将 for 语句写成两个产生式：

$S \rightarrow R$ **do** S

R $\rightarrow$ **for** V := E **to** E

产生上述中间代码的翻译方案如下：

$R \rightarrow$ **for** $V := E_1$ **to** E_2 {*R.nextlist* = *makelist*(*nextstmt*);
emit ('if', E_1.*place*, '>', E_2.*place*, 'goto_');
emit (*V.place*, ':=', E_1.*place*);
R.next = *nextstmt*;
R.eplace = E_2.*place*;
R.vplace = *V.place*; }

$S \rightarrow R$ **do** S_1 {*backpatch* (S_1.*nextlist*, *nextstmt*);
S.nextlist = *merge* (*R.nextlist*, *makelist*(*nextstmt*));
emit ('if', *R.vplace*, '=', *R.eplace*, 'goto_');
emit (*R.vplace*, ':=', *R.vplace*, '+ 1');
emit ('goto', *R.next*); }

这里用的技术就是有些教材上介绍的回填技术。有关这个翻译方案的几点解释如下。

1. 因为在 E_1 和 E_2 的中间代码产生后，应及时产生 if $t_1 > t_2$ goto L1 等中间代码（在产生 S_1 的中间代码以前），所以需要把原来的产生式分成两个，在扫描完 **for** $V := E_1$ **to** E_2 后做一次归约，产生这些中间代码。

2. *nextstmt* 给出在输出的中间代码序列中下一条中间代码语句的下标。*emit* 在产生一条语句后使 *nextstmt* 的值加 1。

3. *R* 的几个综合属性是用来记住一些还要用的数据，以便在 $S \rightarrow R$ **do** S_1 的规则中使用。

4. *makelist* 函数构造只有一个表元的表，这唯一的表元就是所提供的参数，它是一个尚不完整的语句（转移的目标标号尚不知道）的下标。

5. *merge* 函数是将两张表合并。

6. *backpatch* (*p*, *i*) 把 *i* 作为目标标号，填到表 *p* 的每一个语句中。

***7.7** C 语言的 for 语句有下列形式：

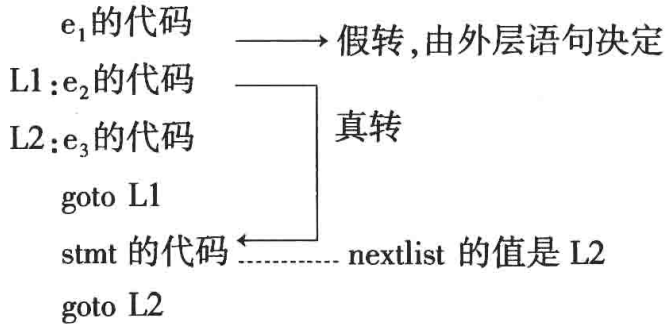
for (e_1;e_2;e_3) stmt

它和

```
e1;
while (e2) do begin
    stmt;
    e3;
end
```

有同样的含义。构造一个翻译方案，把 C 风格的 for 语句翻译成三地址代码。

答 首先给出 for 语句的中间代码结构如下：



然后直接给出接近实际编译器的一个翻译方案如下。

$S \rightarrow$for (E_1 ; M_1 E_2 ; M_2 E_3) N S_1 { *emit* ('goto', M_2.*next*);
backpatch (E_2.*truelist*, N.*next*);
backpatch (S_1.*nextlist*, M_2.*next*);
backpatch (N.*nextlist*, M_1.*next*);
S.*nextlist* = E_2.*falselist*; }

$M \rightarrow \varepsilon$ { M.*next* = *nextstmt*; }

$N \rightarrow \varepsilon$ { N.*nextlist* = *makelist* (*nextstmt*);
emit ('goto '); N.*next* = *nextstmt*; }

分析 如果用 code 属性来保存生成的代码,则很容易给出翻译方案。这里还是逐条语句地生成中间代码,并用上一题的回填技术。

注意这里插入的几个空归约的作用。M_1是用来记住 E_2的开始地址 L1,M_2是用来记住 E_3的开始地址 L2。N 的目的是用来产生语句 goto L1,它必须在分析 S_1之前产生。从上面的中间代码结构来看,产生 goto L1 时,L1 的地址应该是已知的了。但是 goto L1 是在 $N \rightarrow \varepsilon$ 归约时产生的,这时不能访问 M_1的属性(L1 的值记在 M_1的属性中),因此这个转移的目标地址要回填。

7.8 考虑下面的 for 语句

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

对这样的 for 语句可以有三种不同的语义定义。一种可能的含义是,步长表达式 10-j 和终值表达式 10 *j 都仅在循环前计算一次,例如 PL/I 语言。这样,如果在循环前 j = 5,那么该循环体执行 10 次。第二种语义完全不同,要求每次通过循环体时,都要计算步长表达式和终值表达式。例如,若在循环前 j = 5,那么该循环将不会终止,C 语言的 for 语句就属于这种情况,其表达式 e_2和 e_3也是要重复计算的(见习题 7.7)。第三种含义由 ALGOL 这样的语言给出。当步长是负数时,该循环终止的测试是 i<10 *j,而不是 i>10 *j。分别写出这三种定义下的中间代码结构。

答 先写出这种 for 语句的语法;

for i := E_1 **step** E_2 **until** E_3 **do** S

对于第一种定义,其中间代码结构和 Pascal 语言 for 语句的类似,只不过 Pascal 语言把步长限定为 1。

$t_1 := E_1$

$t_2 := E_2$

$t_3 := E_3$

if $t_1 > t_3$ goto L1

$v := t_1$

L2: S

if $v = t_3$ goto L1

$v := v + t_2$

if $v \leqslant t_3$ goto L2

L1:

对于第二种定义,其中间代码结构如下。由于 C 语言 for 语句没有循环控制变量的概念,因此两者的差异还是较大的。

$t_1 := E_1$

goto L1

L2: $t_2 := E_2$

$t_1 := t_1 + t_2$

L1: $t_3 := E_3$

if $t_1 > t_3$ goto L3

$v := t_1$

S

goto L2

L3:

对于第三种定义,其中间代码结构如下。由于步长可正可负,因此要根据步长的正负来决定所用的终止测试条件。

$t_1 := E_1$

L1: $t_2 := E_2$

$t_3 := E_3$

if $t_2 < 0$ goto L2

if $t_1 > t_3$ goto L3

goto L4

L2: if $t_1 < t_3$ goto L3

L4: $t_1 := t_1 + t_2$

$v := t_1$

S

goto L1

L3:

7.9 从第6章的习题可以看到C语言编译器为函数调用所生成的目标代码的结构。现在请写一个翻译方案,为过程调用生成中间代码。它和C语言的实现相同的地方是实在参数的值被压进栈,不同的地方是实在参数按从左到右的顺序计算和压栈。

答 过程调用的语法如下:

$S \rightarrow$ **call id** (*Elist*)

$Elist \rightarrow Elist$, E

$Elist \rightarrow E$

过程调用的中间代码结构如下:

$t_1 := E_1$

push t_1

. . .

$t_n := E_n$

push t_n

call id

所求的翻译方案如下:

$S \rightarrow$ **call id** (*Elist*)	{ *emit* ('call' , **id**. *place*) ; }
$Elist \rightarrow Elist_1$, E	{ *emit* ('push' , E. *place*) ; }
$Elist \rightarrow$ E	{ *emit* ('push' , E. *place*) ; }

分析 题目之所以没有要求按C语言的方式,让实在参数从右向左计算,是考虑到一边分析一边产生中间代码的困难性。如果那样要求,那么只能用 *code* 属性保存产生的中间代码,前面有过这样的例子。

实际的C编译器,通常中间代码是语法树形式,实在参数表是一棵语法树,限定这种情况到先计算右子树就可以实现实在参数的从右向左计算。

也许会认为按照第4章所介绍的,若要求实在参数从右向左计算的话,把上面的文法改成从右向左归约就可以了:

$S \rightarrow$ **call id** (*Elist*)	{ *emit* ('call' , **id**. *place*) ; }
$Elist \rightarrow E$, $Elist_1$	{ *emit* ('push' , E. *place*) ; }
$Elist \rightarrow E$	{ *emit* ('push' , E. *place*) ; }

注意,这里的 E 是非终结符。这个改动做到了实在参数的结果按从右向左的次序压栈,但各个实在参数表达式 E 的计算还是从左向右的。

7.10 某语言有两种语句:

$S \rightarrow$过程调用语句 | 下标变量赋值语句

过程调用语句的形式是:**id**(**id**, **id**, …, **id**),即过程名加上置于圆括号中的变量表。下标变量赋值语句的形式是:**id**(**id**, **id**, …, **id**) := **id**(**id**, **id**, …, **id**),赋值号两边都是数组名加置于圆括号中的变量表。

(a) 请完成过程调用语句和下标变量赋值语句的文法设计,得到一个以语句 S 为开始符号的 LR(1)文法。不得超过 6 个产生式,不需要证明所给出的文法是 LR(1)文法的。

(b) 如果想在 LR 分析的同时完成语义分析和中间代码生成,基于所设计的文法有什么困难?

答 (a) 这两种语句的 LR(1)文法是:

$S \rightarrow Call \mid Assign$

$Call \rightarrow$ **id**(*id_list*)

$Assign \rightarrow$ **id**(*id_list*) := **id**(*id_list*)

id_list → *id_list*, **id** | **id**

(b) 由于过程参数和下标表达式的中间代码是不一样的,在 **id** 和 *id_list*, **id** 向 *id_list* 归约时,不知按实在参数还是下标表达式来生成中间代码。

分析 从移进-归约的角度来看,不管是这两种语句的哪一种,都是先把 **id**(**id**, **id**, …, **id**)中的 **id**, **id**, …, **id** 归约成 *id_list*,然后再根据后面是赋值号还是结束标记来决定是移进赋值号还是归约为 Call 语句。

对于中间代码生成来说,对作为实在参数的 **id** 将产生压栈语句,而对作为下标的 **id** 将产生下标变量地址计算的语句。因此若无其他信息参考,很难产生恰当的代码。

主教材上所给的文法

$S \rightarrow Call \mid Assign$

$Call \rightarrow$ id(*parameter_list*)

parameter_list → *parameter_list*, **id** | **id**

$Assign \rightarrow$ **id**(*expr_list*) := **id**(*expr_list*)

expr_list →*expr_list*, **id** | **id**

是从便于产生中间代码考虑的,但可惜该文法是二义的。

当和语义分析结合起来考虑时,对于这样的问题,编译器有多种解决办法。例如,让词法分析器变得聪明一点,对于过程名,它返回 **procid** 给语法分析器。这样上面的文法就需要修改,两种情况也就区别开了。另一种方式不改文法,根据查符号表得到的 **id** (*id_list*)中 **id** 的不同属性来决定执行的不同语义动作。3.13 题已经讨论过这个问题。

7.11 下面的翻译方案比 7.2 题的简单,因为没有过程嵌套,也没有记录类型。

$P \rightarrow \{ offset := 0 \}\ D$

$D \rightarrow D\ ;\ D$

$D \rightarrow$ **id** : T { $enter($**id**$.name,\ T.type,\ offset);\ offset = offset + T.width;$ }

$T \rightarrow$ **integer** { $T.type =$ integer; $T.width = 4;$ }

$T \rightarrow$ **real** { $T.type = real;\ T.width = 8;$ }

该翻译方案使用了变量 *offset*。请重写该翻译方案，完成同样的工作，但只使用文法符号的属性，而不使用变量 *offset*。

答 $P \rightarrow \{D.offset_1 = 0;\}\ D\ \{P.offset = D.offset_2;\}$

$D \rightarrow \{D_1.offset_1 = D.offset_1;\}\ D_1;$

$\{D_2.offset_1 = D_1.offset_2;\}\ D_2\ \{D.offset_2 = D_2.offset_2;\}$

$D \rightarrow$ **id**: $T\ \{enter(\textbf{id}.name, T.type, D.offset_1); D.offset_2 = D.offset_1 + T.width;\}$

$T \rightarrow$ **integer** $\{T.type = \text{integer}; T.width = 4;\}$

$T \rightarrow$ **real** $\{T.type = real; T.width = 8;\}$

分析 文法符号 *D* 的属性 $offset_1$ 是继承属性，代表在分析 D 前原来使用的变量 *offset* 的大小；属性 $offset_2$ 是综合属性，代表在分析 *D* 后原来使用的变量 *offset* 的大小。P 的属性 *offset* 是综合属性，记录该过程所分配的空间。

7.12 Pascal 语言的赋值语句并非仅对简单类型而言，例如若有类型声明 a，b：array [1..100] of integer，则赋值 a:=b 是允许的。同样，若 a 和 b 是同一类型的两个记录，则赋值 a:=b也是允许的（对 C 语言的结构体类型，这样的赋值也是可以的）。

用主教材给出的三地址语句，能否为这种赋值产生中间代码？若持肯定态度，请给出对应这种赋值的中间代码序列；否则请为这种赋值设计一种三地址语句。所选用或设计的三地址语句要便于目标代码的生成。

答 主教材上的中间代码有形如 x := y 的复写语句，用它就可以了。但是生成目标代码时，必须考虑 x 和 y 的类型。若 x 和 y 不是简单类型，则应该根据它们类型的 size，产生一连串的值传送指令。具体的机器指令可以参考 8.7 题。

7.13 下面是一个 C 语言程序和在 x86/Linux 系统上编译（版本较低的 GCC 编译器，并且未使用优化）该程序得到的汇编代码（为便于理解，略去了和讨论本问题无关的部分，并改动了一个地方）。

（a）为什么会出现一条指令前有多个标号的情况，如 .L2 和 .L4，还有 .L5、.L3 和 .L1？从控制流语句的中间代码结构角度加以解释。

（b）每个函数都有这样的标号 .L1，它的作用是什么，为什么本函数没有引用该标号的地方？

```
main() {
    long i,j;
    if (j)
        i++;
    else
        while (i) j++;
}
```

```
main:
    pushl %ebp                  // 将老的基地址指针压栈
    movl %esp,%ebp              // 将当前栈顶指针作为基地址指针
    subl $8,%esp                // 为局部变量分配空间
    cmpl $0,-8(%ebp)
    je .L2
    incl -4(%ebp)
    jmp .L3
.L2:
.L4:
    cmpl $0,-4(%ebp)
    jne .L6
    jmp .L5
.L6:
    incl -8(%ebp)
    jmp .L4
.L5:
.L3:
.L1:
    leave                       // 和下一条指令一起完成恢复老的基地址指针,将栈顶
    ret                         // 指针恢复到调用前参数压栈后的位置,并返回调用者
```

答 (a) 条件语句和循环语句的中间代码结构如下:

```
if (E) then S1 else S2          while (E) do S
    E 的代码                    L4: E 的代码
    假转 L2                         真转 L6
    S1的代码                        转   L5
    转 L3                       L6: S 的代码
L2:S2的代码                         转   L4
L3:                             L5:
```

用这种代码结构,当 while 语句作为条件语句的 S_2 时,就会出现题目所给的这种情况。

(b) .L1 标号定义的入口是返回调用者时要执行的指令,在函数内部有 return 语句时就会跳转到 .L1。

分析 从对版本较高的编译器给出的汇编代码的分析可以看出,这些编译器不会产生一条指令前有多个标号的情况。若无必要,它们也不会产生像 .L1 这样的标号。

7.14 赋值语句 A[x, y]:= z(其中 A 是 10×5 的数组)的注释分析树如图 7.3。请根据主教材上 7.3.3 节的翻译方案,把图 7.3 中的属性值都补上(像主教材上的图 7.10 那样),并且把每步归约产生的中间代码写在相应产生式的旁边。

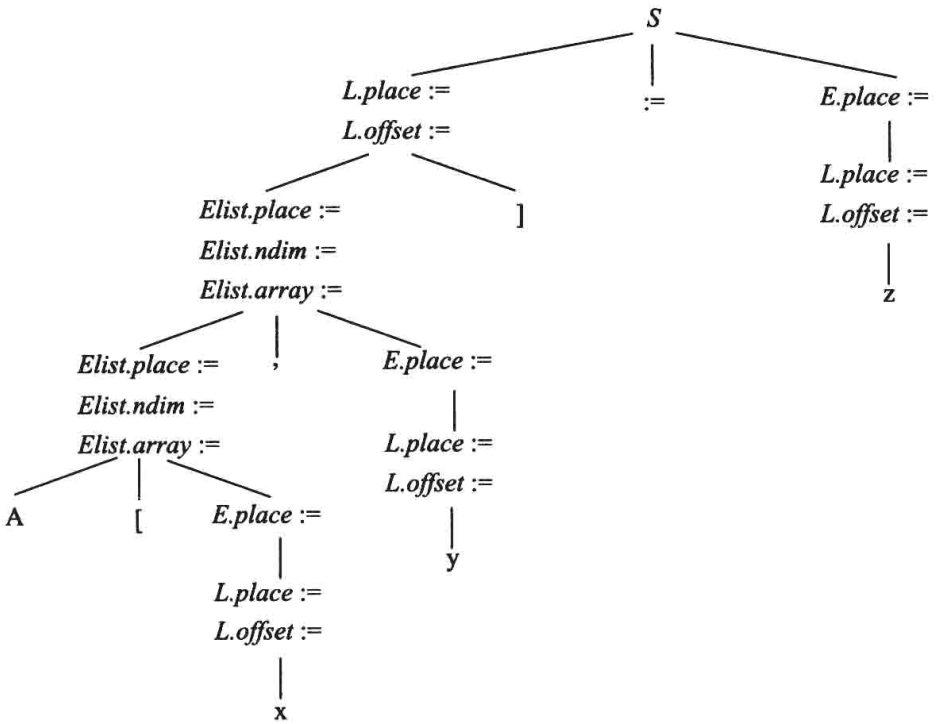


图 7.3 A[x,y]:=z 的注释分析树

答 见图 7.4。

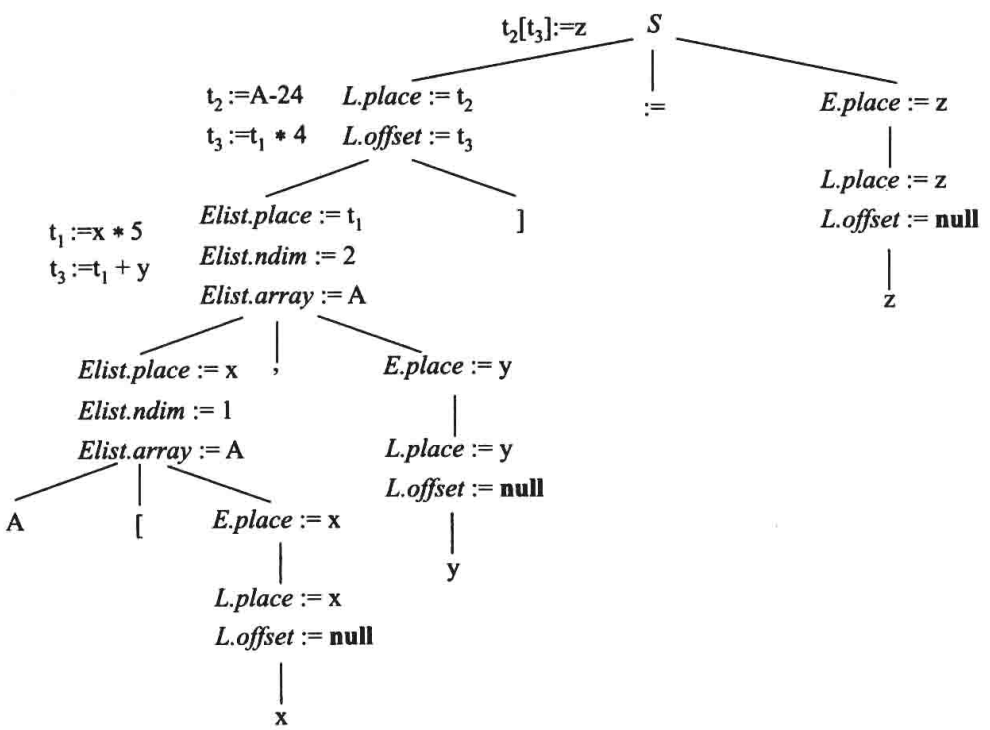


图 7.4 A[x,y]:=z 的已经标注中间代码的注释分析树

7.15 C 语言和 Java 语言的数组声明和数组元素引用的语法形式同主教材上 7.3 节讨论的不一样，例如 float A[10][20]和 A[i+1][j-1]，并且每一维的下界都是 0。若适应这种情况的赋值语句的文法如下：

$S \rightarrow L := E$

$E \rightarrow E + E \mid (E) \mid L$

$L \rightarrow L[E] \mid \mathbf{id}$

(a) 重新设计主教材上 7.3.2 节数组元素的地址计算公式，以方便编译器产生数组元素地址计算的中间代码。不要忘记每一维的下界都是 0。

(b) 重新设计数组元素地址计算的翻译方案。只需写出产生式 $L \rightarrow L[E] \mid \mathbf{id}$ 的翻译方案，但要能和 7.3.3 节中产生式 $S \rightarrow L := E$ 和 $E \rightarrow E + E \mid (E) \mid L$ 的翻译方案衔接。若翻译方案中引入新的函数调用，要解释相关函数的含义。

答 (a)以二维数组为例，A[i_1][i_2]的地址计算公式是

$$base + i_1 \times w_1 + i_2 \times w_2$$

其中，w_1是存放一行元素需要的字节数，而 w_2是存放行中一个元素需要的字节数。对于 k 维数组来说，A[i_1][i_2]…[k]的地址计算公式是

$$base + i_1 \times w_1 + i_2 \times w_2 + \cdots + i_k \times w_k$$

其中 $w_j(1 \leqslant j \leqslant k)$是二维情况的 w_1和 w_2的推广。

(b)翻译方案如下，其中函数 *getwidth*(p, k)表示到符号表中去取该维一个元素需要的字节数(其中 p 是数组名在符号表中指针，k 表示第几维)。

$L \rightarrow L_1[E]$ {*L.array* = L_1.*array*; *L.ndim* = L_1.*ndim*+1;
w = *getwidth*(*L.array*, *L.ndim*);
if (*L.ndim* == 1) **begin**
L.offset = *newtemp*;
emit (*L.offset*, '=', *E.place*, '*', *w*);
end else begin
t = *newtemp*; *L.offset* = *newtemp*;
emit (*t*, '=', *E.place*, '*', *w*);
emit (*L.offset*, '=', L_1.*offset*, '+', *t*);
end}

$L \rightarrow \mathbf{id}$ {*L.place* = **id**.*place*; *L.offset* = **null**; *L.ndim* = 0; *L.array* = **id**.*place*;}

7.16 (a) 主教材上图 7.6 的翻译方案不能用于允许递归过程的情况(见 7.2.2 节的解释)。对图 7.6 的文法及其翻译方案作最少修改(不改变该文法定义的语言)，使得修改后的翻译方案能够用于有直接递归调用的情况(可以修改或增加语义动作)。

(b)对于允许过程嵌套的语言,结合主教材上图 7.6 和图 7.9 的翻译方案,像主教材上图 2.10 和 2.11 那样给出函数 *lookup*(*name*)的描述。

答 (a) 过程声明的文法修改如下(不必修改的产生式及其语义动作没有给出):

$D \rightarrow ProcName\ D_1;\ S$ { *t* = *top*(*tblptr*); *addWidth*(*t*, *top*(*offset*)); *pop*(*tblptr*);
pop(*offset*); *linkProcTable*(*top*(*tblptr*), *ProcName.name*, *t*); }

ProcName → **proc id** ; { *enterProcName*(*top*(*tblptr*), **id**.*lexeme*); *t* = *mkTable*(*top*(*tblptr*));
push(*t*, *tblptr*); *push*(0, *offset*); *ProcName.name* = **id**.*lexeme*; }

其中,

1. *enterProcName*(*table*, *name*):在 *table* 指向的符号表中为过程名 *name* 建立新条目;

2. *linkProcTable*(*table*, *name*, *newTable*):在 *table* 指向的符号表中找到过程名 *name* 的条目,让该条目中的符号表指针等于 *newTable*(即指向过程 *name* 的符号表)。

(b) *lookup*(*name*) {

```
    t = 符号表指针栈的栈顶元素;
    while( t 是符号表指针) {
        p = localLookup(name);
        if (p! =nil) return p;
        else t = 符号表指针栈中下一个符号表指针;
    }
    return nil;
}
```

其中,*localLookup* 的功能与主教材 7.3.1 节的 *lookup* 的功能一样。

第8章 代码生成

一、主要知识点

主教材第8章主要应掌握下面一些内容。

1. 代码生成器设计中的主要问题:存储管理、计算次序的选择、寄存器的分配、指令的选择等。其中有关存储管理这一部分,在第6章给出的一些目标程序中,已经了解得比较清楚了。

2. 目标机器几种常用的地址模式和一些常用的指令。

3. 基本块和程序流图。

4. 一些简单的代码生成算法。

这一章比上一章更容易学,但是用本章所学的代码生成的思想和方法来解释一些实际编程中碰到的问题却不是件容易的事情。

二、习题精选与解析

***8.1** 在早先的SPARC/SunOS系统上,下面的程序经某编译器编译后,其运行结果是120。但是如果把第10行的abs(1)改成1的话,则程序结果是1。试分析为什么会有这不同的结果。

```
int fact( ) {
    static int i=5;
    if(i==0) {
        return(1);
    }
    else {
        i=i-1;
        return((i+abs(1)) * fact( ));
    }
}
main( ) {
    printf("factor of 5 = %d\n", fact( ));
}
```

答 这个问题主要由编译器对表达式的代码生成策略引起的。在代码生成过程中,出于寄存器分配优化的需要或其他原因,编译器可能会对不同的表达式选择不同的计算次序。如果表达式各部分的计算相互影响,那么不同的计算次序会有不同的结果。在这个问题中,表达式(i+abs(1)) * fact()的左子表达式引用 i,而右子表达式的函数调用会修改 i 的值。

不难分析,程序结果是 120 时,该表达式的实际计算次序是先完成左子表达式的计算,然后完成右子表达式的计算。

如果先计算右子表达式的话,那么不断进行 fact 函数调用,一直到 i 等于 0 为止。这样,最后一个 fact 调用的返回值是 1。由于 i 已经等于 0,i+abs(1)等于 1,因此倒数第二个 fact 调用的返回值还是 1。因此可以判断结果是 1。

这里需要解释的是,为什么把(i+abs(1)) * fact()中的求绝对值函数 abs 去掉后,计算次序会发生变化。在有的代码生成方法中,为了便于使用寄存器分配的优化算法,在一棵表达式树上,先为所有的函数调用生成代码,使得表达式树上不再有这样复杂的操作,然后再考虑寄存器分配的优化。这样,对于表达式(i +1) * fact(),运行时先计算右子表达式 fact 函数调用,然后计算左子表达式。而对于表达式(i+abs(1)) * fact(),由于左右子表达式都有函数调用,所以就先计算左子表达式。

从这段分析可以猜测,不同的编译器可能会导致这个程序的结果不一样。事实也是这样,比如在 x86/Linux 系统上,对于 abs 函数调用,GCC 编译器是用一段代码将它展开的,因此有无 abs 不会影响这个表达式的计算次序。

***8.2** 一个 C 语言程序如下:

```
main() {
    long i;
    i = 0;
    printf("%ld\n", (++i)+(++i)+(++i) );
}
```

该程序在 x86/Linux 系统上编译后运行结果是 7(编译器版本是 GCC:(GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)),而在早先的 SPARC/SunOS 系统上编译后的运行结果是 6。试分析运行结果不同的原因。

答 先看图 8.1 表达式(++i)+(++i)+(++i)的中间代码。首先,i +1 不能作为公共子表达式,因为每个这样的运算后都是对 i 的重新赋值。其次,按一般的代码生成,i = i +1(指++i)的计算结果应该保留在某个寄存器中,用于上一层计算,因此这三个 i = i +1 的计算次序不会影响最终的结果。结果应该是 6。

从另一方面看,对于结果是 7 的情况,一定是某个 i = i +1 的结果未保留在寄存器中,上层计算对它的引用落在计算另一个 i = i +1 的后面。一般来说,现在的机器都有足够的寄存器来保留这样简单表达式的中间结果。因此,只有一种可能,i = i +1 的计算根本没有用寄存器。如

果机器有 INC 指令的话,编译器极可能产生一条 INC 指令来完成 i = i +1。x86/Linux 系统上果真是这么做的,下面是该程序在 x86/Linux 系统上的编译结果(编译器版本见汇编代码最后一行),从中可以看到 INC 指令的使用。

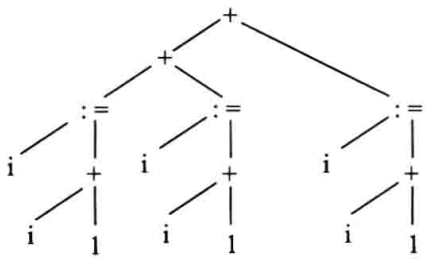


图 8.1 (++i)+(++i)+(++i)的中间代码

```
        .file "jiajia.c"
        .version    "01.01"
gcc2_compiled.:
.section    .rodata
.LC0:
        .string    "%ld\n"
.text
        .align 4
.globl main
        .type main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl  $4,%esp
        movl  $0,-4(%ebp)
        incl -4(%ebp)
        incl -4(%ebp)
        movl -4(%ebp),%eax
        movl -4(%ebp),%edx
        addl %edx,%eax
        incl -4(%ebp)
        addl -4(%ebp),%eax
        pushl %eax
        pushl  $.LC0
```

```
        call printf
        addl  $8,%esp
.L1:
        leave
        ret
.Lfe1:
        .size main,.Lfe1-main
        .ident  "GCC:(GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

应该说,这样的代码生成没有准确体现C语言的语义。正确的做法是,(++i)的结果不马上使用的话,应该保留在不会受其他计算影响的地方。SPARC/SunOS机器上编译器产生的汇编码中就没有这样的INC指令。

由此可以想到,将程序改成下面形式:

```
main() {
        long i;
        i = 0;
        printf("%ld\n",(++i)+((++i)+(++i)));
}
```

在x86/Linux系统上,结果会不同。确实如此,运行结果是9。(++i)+((++i)+(++i))的中间代码见图8.2,编译的结果在下面(无关部分已略去),读者可以自行分析。

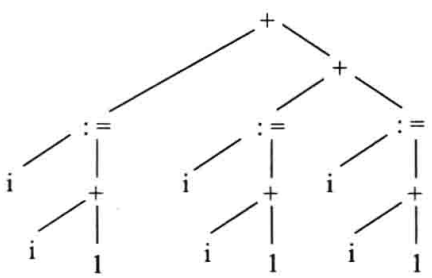


图8.2 (++i)+((++i)+(++i))的中间代码

```
main:
        pushl %ebp
        movl %esp,%ebp
        subl  $4,%esp
        movl  $0,-4(%ebp)
        incl -4(%ebp)
        incl -4(%ebp)
        incl -4(%ebp)
        movl -4(%ebp),%eax
        movl -4(%ebp),%edx
        addl %edx,%eax
        movl %eax,%edx
        addl -4(%ebp),%edx
        pushl %edx
        pushl  $.LC0
        call printf
```

```
        addl  $8,%esp
.L1:
        leave
        ret
```

*8.3 一个 C 语言程序如下,运行时输出 105。

```
main( ) {
        long i;
        i=10;
        i=(i+5) + (i=i*5);
        printf("%d\n",i);
}
```

该程序在 x86/Linux 系统上编译后生成的汇编代码如下(编译器版本见汇编代码最后一行),从生成的汇编代码看出,表达式(i+5) + (i = i*5)的右子树先计算,你能猜测有关的代码生成策略吗?

```
        .file "expression.c"
        .version "01.01"
gcc2_compiled.:
.section    .rodata
.LC0:
        .string    "%d\n"
.text
        .align 4
.globl main
        .type main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl  $4,%esp
        movl  $10,-4(%ebp)
        movl -4(%ebp),%edx
        movl %edx,%eax
        sall  $2,%eax
        addl %edx,%eax
        movl %eax,%edx
```

```
        movl %edx,-4(%ebp)
        leal 5(%edx),%eax
        addl %eax,-4(%ebp)
        movl -4(%ebp),%eax
        pushl %eax
        pushl $.LC0
        call printf
        addl $8,%esp
.L1:
        leave
        ret
.Lfe1:
        .size main,.Lfe1-main
        .ident "GCC:(GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

答 一些教材上介绍了一种表达式树的最优代码(在某种限定下条件下可得到最优代码)生成算法。该算法先自下而上分析表达式各子树完成计算至少需要的寄存器个数,然后根据这些信息来决定先产生哪棵子树的代码(也就是运行时先计算哪棵子树)。不熟悉该算法的读者可以参考陈火旺等编的《程序设计语言编译原理》(国防工业出版社)。

从题目所提供的汇编代码可以看出,除了基地址寄存器外,右子树(i = i * 5)的计算最少需要两个寄存器,而左子树(i+5)的计算最少需要一个寄存器(虽然上面的汇编码中用了两个寄存器,但实际上只用一个也是可行的)。这样,按陈火旺等编的教材上的最优代码生成算法,先生成右子树的代码。

这是一种可接受的解释,虽然该编译器的代码生成算法实际上可能更复杂。重要的是,通过本题又一次看到表达式的计算次序受到代码生成算法的影响。

该程序在 Ubuntu 12.04.2 LTS (GNU/Linux 3.2.0-42-generic x86_64)系统上,经过编译器 GCC:(Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3 编译后,运行结果是 65。可能的情况是,编译器不调整一个表达式中各子表达式的计算次序,或者确认子表达式之间不存在数据依赖关系时才调整计算次序。

8.4 一个 C 语言程序如下:

```
main() {
    int i;
    i=50;
    switch(i * i){
        case 10: i=10; break;
```

```
            case 80: i=80; break;
            case 50: i=50; break;
            case 70: i=70; break;
            case 20: i=20; break;
            default: i=40;
        }
        switch(i*i){
            case 7: i=7; break;
            case 1: i=1; break;
            case 6: i=6; break;
            case 9: i=9; break;
            case 5: i=5; break;
            case 10: i=10; break;
            case 2: i=2; break;
            default: i=40;
        }
    }
```

它在 x86/Linux 系统上编译后生成的汇编代码如下(编译器版本见汇编代码最后一行),请根据所生成的汇编代码,写出程序中两个 switch 语句的目标代码结构的特点。

```
        .file   "switch.c"
        .version    "01.01"
gcc2_compiled.:
.text
        .align 4
.globl main
        .type main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $4,%esp
        movl $50,-4(%ebp)
        movl -4(%ebp),%eax
        imull -4(%ebp),%eax
        cmpl $50,%eax
        je .L5
```

```
    cmpl $50,%eax
    jg .L10
    cmpl $10,%eax
    je .L3
    cmpl $20,%eax
    je .L7
    jmp .L8
    .p2align 4,,7
.L10:
    cmpl $70,%eax
    je .L6
    cmpl $80,%eax
    je .L4
    jmp .L8
    .p2align 4,,7
.L3:
    movl $10,-4(%ebp)
    jmp .L2
    .p2align 4,,7
.L4:
    movl $80,-4(%ebp)
    jmp .L2
    .p2align 4,,7
.L5:
    movl $50,-4(%ebp)
    jmp .L2
    .p2align 4,,7
.L6:
    movl $70,-4(%ebp)
    jmp .L2
    .p2align 4,,7
.L7:
    movl $20,-4(%ebp)
    jmp .L2
    .p2align 4,,7
```

```
.L8:
    movl $40,-4(%ebp)
.L2:
    movl -4(%ebp),%edx
    imull -4(%ebp),%edx
    leal -1(%edx),%eax
    cmpl $9,%eax
    ja .L19
    movl .L20(,%eax,4),%eax
    jmp *%eax
    .p2align 4,,7
.section    .rodata
    .align 4
    .align 4
.L20:
    .long .L13
    .long .L18
    .long .L19
    .long .L19
    .long .L16
    .long .L14
    .long .L12
    .long .L19
    .long .L15
    .long .L17
.text
    .p2align 4,,7
.L12:
    movl $7,-4(%ebp)
    jmp .L11
    .p2align 4,,7
.L13:
    movl $1,-4(%ebp)
    jmp .L11
    .p2align 4,,7
```

```
.L14:
    movl $6,-4(%ebp)
    jmp .L11
    .p2align 4,,7
.L15:
    movl $9,-4(%ebp)
    jmp .L11
    .p2align 4,,7
.L16:
    movl $5,-4(%ebp)
    jmp .L11
    .p2align 4,,7
.L17:
    movl $10,-4(%ebp)
    jmp .L11
    .p2align 4,,7
.L18:
    movl $2,-4(%ebp)
    jmp .L11
    .p2align 4,,7
.L19:
    movl $40,-4(%ebp)
.L11:
.L1:
    leave
    ret
.Lfe1:
    .size main,.Lfe1-main
    .ident "GCC:(GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

答 对于 case 比较少的 switch 语句,编译器生成的代码通常是将 switch 表达式的值和 case 常量依次逐个比较,寻找要执行的 case。在最坏的情况下,要和所有的 case 常量比较后才能决定要执行的 case。

从本题可以看出该编译器至少进行了两点优化。当 case 情况较多时,产生的代码用折半查找来寻找要执行的 case。本题第一个 switch 语句的代码就是这样。

如果 case 比较多,并且 case 常量集中在一个较小的区间内,那么可以用更快的方式找到要

执行的 case。本题第二个 switch 语句的代码就是这样，它寻找要执行的 case 的过程如下。

1. 计算 switch 表达式 i * i $\Rightarrow R_{edx}$。

2. R_{edx}的值减 1(即 i * i-1)$\Rightarrow R_{eax}$。

3. 若 R_{eax}的值不在 0~9 的区间内，则执行 default 语句(入口地址 L19)；否则，转到地址(L20+R_{eax}的值×4)的内容所给出的标号去执行(也就是对应的 case)。

以标号 L20 开始的是如图 8.3 所示的一张地址表。

L13:	case 1
L18:	case 2
L19:	default
L19:	default
L16:	case 5
L14:	case 6
L12:	case 7
L19:	default
L15:	case 9
L17:	case 10

图 8.3　入口地址表

8.5　(a) 为下面的算术表达式文法写一个语法制导的翻译方案，它将每个子表达式 E 的符号(即值大于零还是小于零)记录在属性 $E.sign$ 中(属性值分别用 POS 或 NEG 表示)。可以假定所有的整数都不为零，这样就不用担心零的符号问题了。

$E \rightarrow E * E \mid +E \mid -E \mid unsigned_integer$

(b) 为上面的表达式产生栈机器代码。代码执行后，表达式的值留在栈上。请自己设计所需的栈机器指令，并写清楚指令的含义。

答　(a) 所求语法制导的翻译方案如下：

$E \rightarrow E_1 * E_2$ { **if** $E_1.sign$ = $E_2.sign$ **then** $E.sign$:= POS **else** $E.sign$:= NEG }

$E \rightarrow +E_1$ { $E.sign$:= $E_1.sign$ }

$E \rightarrow -E_1$ { **if** $E_1.sign$ = POS **then** $E.sign$:= NEG **else** $E.sign$:= POS }

$E \rightarrow unsigned_integer$ { $E.sign$:= POS }

(b) 各指令的解释如下：

PUSH　值：　　将值压栈。

NEG：　　将栈顶值取出，计算其相反数，把结果压入栈。

MUL：　　将栈顶和次栈顶的值取出，将它们相乘，把结果压栈。

产生代码的翻译方案如下：

$E \rightarrow E_1 * E_2$　　{ emit('MUL') }

$E \rightarrow +E_1$　　{ }

$E \rightarrow -E_1$　　{ emit('NEG') }

$E \rightarrow$ unsigned_integer { emit('PUSH', $unsigned_integer.lexval$) }

8.6　一个 C 语言程序如下：

```
extern int a;
static int b;
int c;
main( ) {
```

```
    b = a;
}
```

它在 x86/Linux 系统上编译后生成的汇编代码如下(编译器版本见汇编代码最后一行),请说明编译时对 extern 变量的处理和外部变量的处理有什么区别?

```
        .file "extern.c"
        .version    "01.01"
gcc2_compiled.:
.text
        .align 4
.globl main
        .type main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        movl a,%eax
        movl %eax,b
.L1:
        leave
        ret
.Lfe1:
        .size main,.Lfe1-main
        .local  b
        .comm   b,4,4
        .comm   c,4,4
        .ident  "GCC:(GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

答 从汇编代码可以看出,外部变量 c 和静态外部变量 b 都在本汇编文件中分配存储单元,它们用 .comm 伪指令分配空间。而 extern 变量 a 在本汇编文件中没有分配存储单元,但本汇编文件中有对它的引用。在和其他文件连接时,若在其他文件中找不到这个 extern 变量的定义点,那么报告该 extern 变量未定义。

8.7 在 C 语言中,若 a 和 b 是相同的结构体类型,那么赋值 a=b 是允许的。但是编译器对这种赋值的实现方式可能和它们的字节数(size)有关。下面是两个 C 语言程序及其在 x86/Linux 系统上经某版本的 GCC 编译器编译生成的汇编代码(略去了和本题目无关的部分)。扼要叙述这些汇编代码中体现出的实现方式上的区别。(下面汇编程序中给出的注释仅供参考)

第一个 C 程序及对应汇编程序:

```
struct {long i,j; double m;}a,b;
main( ) {
    a=b;
}

main:
    pushl %ebp
    movl %esp,%ebp
    movl $b,%eax
    movl $a,%edx
    movl (%eax),%ecx
    movl %ecx,(%edx)
    movl 4(%eax),%ecx
    movl %ecx,4(%edx)
    movl 8(%eax),%ecx
    movl %ecx,8(%edx)
    movl 12(%eax),%eax
    movl %eax,12(%edx)
.L1:
    leave
    ret
```

第二个 C 程序及对应汇编程序：

```
struct {long i,j; double m,n;}a,b;
main( ) {
    a=b;
}

main:
    pushl %ebp
    movl %esp,%ebp
    pushl %edi
    pushl %esi
    movl $a,%edi
    movl $b,%esi
    cld                    // 设定方向标志,使地址指针自动增量
```

```
    movl $6,%ecx        // 设定重复次数
    rep                 // 表示重复下面的指令
    movsl               // 传送指令
.L1:
    leal -8(%ebp),%esp
    popl %esi
    popl %edi
    leave
    ret
```

答 在结构体的字节数较少时,则为该结构体各域分别产生值传送指令。在结构体的字节数较多时,则根据值传送的源地址(b 的地址)、目的地址(a 的地址)和该结构体的字节数折合成的字数,产生简洁的重复传送指令,以提高效率。

分析 两个程序的区别在于,所用结构体的字节数肯定不一样,因为第二个程序的结构体多了一个 double 类型的域。从第一个程序的汇编代码可以看出它是逐个域地传送,最后一个域是 double 类型,因此传送两个字。

从第二个程序的汇编代码和所给出的注释可以判断出它的传送方式,而且可以看出,结构体的字节数再多也就这几条指令。

8.8 在 C 语言中,若 a 和 b 是相同的结构体类型,那么赋值 a=b 是允许的。这句话对下面程序中出现的赋值 p = q 是否有效? 即编译器会报错吗? 若编译器不报错,程序的输出是什么?

```
typedef struct{int n; float a[ ];} record;
record p = {2, {1.0, 2.0} }, q = {3, {1.0, 2.0, 3.0} };
main( ) {
    p = q;
    printf("%d, %f, %f, %f\n", p.n, p.a[0], p.a[1], p.a[2]);
    printf("%d, %f, %f, %f\n", q.n, q.a[0], q.a[1], q.a[2]);
}
```

答 由于 p 和 q 属于同一个结构体类型,因此编译时不会报错。

上述程序在 Ubuntu 12.04.2 LTS (GNU/Linux 3.2.0-42-generic x86_64)系统上,经 GCC:(Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3 编译后,运行结果如下 :

```
3, 1.000000, 2.000000, 3.000000
1077936128, 1.000000, 2.000000, 3.000000
```

q.n 原先的值经结构体赋值后被覆盖。之所以变成了 1077936128,是因为它和浮点数 3.0 的机器编码一致。

分析 p 和 q 虽然属同一个类型,但是它们的 size 不一样,应该以不能赋值为好。上面这个

结果或许是语言定义的疏漏,或许是编译器的错误。下面收集更多信息来分析一下。

1. int 和 float 的 size 都是 4。

2. sizeof(record)、sizeof(p)和 sizeof(q)的值都是 4。显然,在计算 record 类型的 size 时,把 float[ ]类型的 size 当成了 0,另外认为 p、q 的 size 和它们类型的 size 一样。

3. 若有代码 typedef float array[ ]; array a = {}, b = {1.0, 2.0, 3.0},则 sizeof(a)和 sizeof(b)分别等于 0 和 12。但对于 sizeof(array),编译器会报错,指出 sizeof 用到了一个不完整的类型上。由此看出,a 和 b 虽然都是 array 类型的变量,它们的 size 可以不一样,而这时它们类型的 size 被认为未定义。

单独的 float[ ]类型被认为 size 没有定义,而处于结构体类型中的 float[ ]类型被认为 size 为 0。显然这是一个矛盾,它可能是编译器的错误,也可能是语言定义的疏漏。

***8.9** 从第 6 章和本章给出的汇编程序例子已经看出,C 语言的外部变量(静态分配)和自动变量(动态栈式分配)的存储分配方式是不一样的,但是大部分编译原理的教材在介绍代码生成时,为简单起见,没有涉及动态栈式分配变量的地址描述。另外,大部分编译原理的教材在介绍代码生成时,也没有涉及用表达式左值和右值的区别。

图 8.4 是赋值表达式 a[i] = b +1 的中间代码树,其中 a 和 i 是动态栈式分配的,b 是静态分配的。该中间代码树上已体现动态栈式分配变量的地址描述和左右值的区别,它们有助于代码生成。

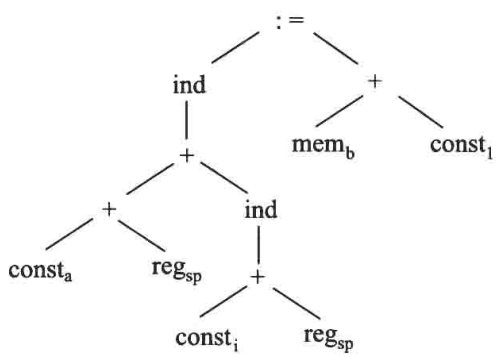


图 8.4 赋值表达式 a[i] = b +1 的中间代码树

请你对图 8.4 的中间代码树做出解释。

答 先对结点标记作出解释。

- reg_{sp}表示栈的基址寄存器;
- const 表示常数,其中 $const_1$表示常数 1,$const_a$和 $const_i$分别表示 a 和 i 在活动记录中的相对地址,它们是常数;
- mem 表示内存地址,mem_b表示分配给 b 的内存地址;
- ind 表示它的子树是一个左值表达式,当需要左值时,直接用 ind 下面的子树,当需要右值时,把 ind 下面的子树作为地址,取该地址的内容,因此 ind 又可表示间接访问的意思。

有了这个解释，就不难理解表达式树的含义了。$mem_b + const_1$表示 b+1，$const_i + reg_{sp}$表示 i 的地址，$const_a + reg_{sp}$表示 a 的起始地址。$(const_a + reg_{sp}) + ind(const_i + reg_{sp})$表示 a[i]的地址，注意这是一个左值(a 的起始地址)和一个右值(i 的值)相加，得到的是一个左值。$ind((const_a + reg_{sp}) + ind(const_i + reg_{sp}))$在赋值号的左边，因此生成代码时直接用最外层括号中的左值表达式。

很多 C 语言编译器都采用这种形式的中间代码树，给代码生成提供了丰富的信息。

第9章　独立于机器的优化

一、主要知识点

主教材第9章主要应掌握下面一些内容。

1. 对各类优化的理解，包括常量合并、公共子表达式删除、复写传播、死代码删除、循环优化（代码外提、归纳变量删除、强度削弱）等。

2. 数据流分析：对四种数据流分析（到达-定值分析、活跃变量分析、可用表达式分析和常量传播）方程的认识和区分，和对这些方程迭代求解方法的理解。数据流方程迭代求解的算法并不复杂，但手工根据算法进行演算求解将非常烦琐，所以强调的是对这些方法的理解。

3. 控制流分析：掌握建立程序流图的算法和流图中自然循环的识别算法。

4. 了解数据流分析的理论基础和一般框架，了解部分冗余删除。

和第8章类似，本章的重点是在用优化的基本知识来分析和解决实际问题。

二、习题精选与解析

9.1　一个C语言程序如下：

```
main ( ) {
        int i, j, k;
        i = 5;
        j = 1;
        while (j<100) {
                k=i+1;
                j=j+k;
        }
}
```

在x86/Linux系统上经优化编译后（编译器版本见汇编代码最后一行），生成的代码如下：

```
        .file "optimize.c"
        .version "01.01"
gcc2_compiled.:
```

```
.text
    .align 4
.globl main
    .type main,@function
main:
    pushl %ebp
    movl %esp,%ebp
    movl $1,%eax
    movl $6,%edx
    .p2align 4,,7
.L4:
    addl %edx,%eax
    cmpl $99,%eax
    jle .L4
    leave
    ret
.Lfe1:
    .size main,.Lfe1-main
    .ident "GCC:(GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

试说明编译器对这个程序作了哪些种类的优化(只需要说明复写传播、删除公共子表达式等,不需要说明是怎样完成这些优化的)。

答 所用到的优化有复写传播、常量合并、代码外提、删除无用赋值,另外,优化使得对 i,j 和 k 的赋值都成为无用赋值而被删除,因此为 i,j 和 k 分配内存单元也成为多余,从而被取消。优化后的目标程序对应下面的 C 程序。

```
main ( ) {
    register int j, k;
    j=1;
    k=6;
    while(j<=99){
        j=j+k;
    }
}
```

9.2 优化编译器对下面程序的局部变量 i 和 j 不分配空间,为什么?

```
main ( ) {
```

```
    long i, j;
    i = 5;
    j = i * 2;
    printf("%d\n", i+j);
}
```

答 和上一题一样,优化时,通过复写转播和常量合并等会发现,对 i 和 j 的赋值都是无用赋值,可以删除,从而对 i 和 j 的存储分配也就没有必要了。

***9.3** 下面是用 C 语言写的求最大公约数的函数:

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

其中的递归调用称为尾递归(即 return 后的表达式是一个递归调用,而其他地方没有递归调用)。对于尾递归,编译器可以产生和一般的函数调用不同的代码,使得目标程序运行时,这种递归调用所需的存储空间大大减少,也缩短了运行时间。对于尾递归,编译器应怎样产生代码,简述你的想法(若用源语言一级的优化来回答此问题,则不合题目要求)。

答 对于尾递归调用,运行时不用建立新的活动记录,就在当前活动记录完成计算。针对这个例子的具体步骤叙述如下。

1. 计算调用的实在参数 q 和 p%q,存放在不同的寄存器中。

2. 将上述寄存器中实在参数的值存入当前活动记录中形式参数 p 和 q 的存储单元。

3. 转到本函数第一条语句的起始地址继续执行(注意,不是本函数的入口地址,因为函数入口是一段调用序列代码,然后才是第一条语句的代码)。

分析 由于进行尾递归调用得到的结果,就是函数的当前执行应该返回给调用者的值。因此当尾递归调用的实在参数计算完毕后,当前活动记录中的局部数据不再使用,当前活动记录可以被覆盖作为尾递归调用要创建的新的活动记录。

上面答案中的改进本质上是把递归计算变换成迭代计算。

下面给出这个程序在 x86/Linux 系统上优化编译的结果(编译器版本见汇编代码最后一行)。从中可以看出,该优化的结果和答案中所说的是一致的,只不过它做了更多的优化,整个迭代计算不用形式参数 p 和 q 的存储单元,而是用它们在寄存器中的值。

```
    .file "gcd.c"
    .version "01.01"
gcc2_compiled.:
```

```
.text
    .align 4
.globl gcd
    .type gcd,@function
gcd:
    pushl %ebp
    movl %esp,%ebp
    pushl %esi
    pushl %ebx
    movl 8(%ebp),%esi
    movl 12(%ebp),%ebx
.L4:
    movl %esi,%eax
    cltd
    idivl %ebx
    movl %edx,%ecx
    testl %ecx,%ecx
    je .L2
    movl %ebx,%esi
    movl %ecx,%ebx
    jmp .L4
    .p2align 4,,7
.L2:
    movl %ebx,%eax
    leal -8(%ebp),%esp
    popl %ebx
    popl %esi
    leave
    ret
.Lfe1:
    .size gcd,.Lfe1-gcd
    .ident "GCC:(GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

9.4 考虑下面的三地址语句序列：

```
b = 1
```

```
        b = 2
        if w <= x goto L2
        e = b
        goto L2
L1:goto L3
L2:c = 3
    b = 4
    c = 6
L3:if y <= z goto L4
    goto L5
L4:g = g + 1
    h = 8
    goto L1
L5:h = 9
```

(a) 在该代码中用横线将代码分成基本块,并给每个基本块赋一个序号。

(b) 画出该代码的控制流图,每个基本块就用(a)中的序号表示。

(c) 若有循环,则列出构成每个循环的结点。

答 (a) 基本块的划分如下:

```
        b = 1
        b = 2
        if w <= x goto L2          (1)
        e = b
        goto L2                    (2)
L1:goto L3                         (3)
L2:c = 3
    b = 4
    c = 6                          (4)
L3:if y <= z goto L4               (5)
    goto L5                        (6)
L4:g = g + 1
    h = 8
    goto L1                        (7)
L5:h = 9                           (8)
```

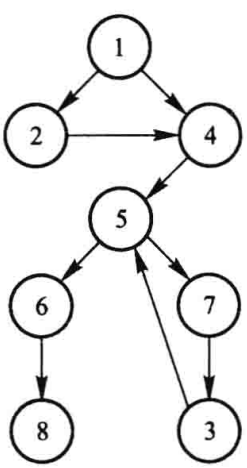


图 9.1 习题 9.4 的流图

(b) 代码的控制流图见图 9.1。

(c) 结点 5、7 和 3 构成一个循环,其中 5 是入口结点。

9.5 对图 9.2 的流图：

(a) 计算支配关系。

(b) 找出一种深度优先排序。

(c) 对(b)的结果,标明前进边、后撤边和交叉边。

(d) 说明该流图是否可归约。

(e) 计算该流图的深度。

(f) 找出该流图的自然循环。

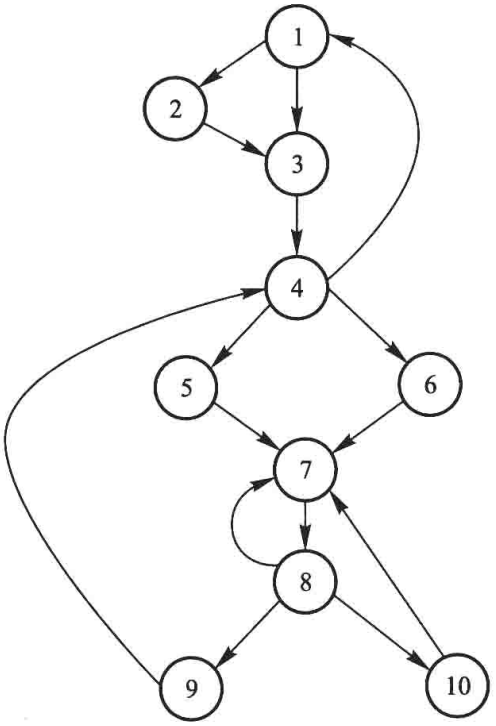


图 9.2 习题 9.5 的流图

答 (a) 各结点的支配集如下：

$D(1) = \{1\}, D(2) = \{1, 2\}, D(3) = \{1, 3\}, D(4) = \{1, 3, 4\}, D(5) = \{1, 3, 4, 5\}, D(6) = \{1, 3, 4, 6\}, D(7) = \{1, 3, 4, 7\}, D(8) = \{1, 3, 4, 7, 8\}, D(9) = \{1, 3, 4, 7, 8, 9\}, D(9) = \{1, 3, 4, 7, 8, 10\}$。

(b) 一种深度优先的排序如下：

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

图 9.2 流图的深度优先表示见图 9.3。

(c) 图 9.3 上的实线边都是前进边,后撤边有 4 → 1、8 → 7、9 → 4 和 10 → 7。交叉边有 2 → 3 和 5 → 7。

(d) 该流图可归约,因为上述 4 个后撤边都是回边。

(e) 流图的深度是 2,因为路径 9 → 4 → 1 有两条后撤边。图中不存在含更多后撤边的无环路径。

(f) 流图的自然循环有 3 个：

回边 8 → 7 和 10 → 7 确定同一个自然循环,其结点集合是{7, 8, 10}。回边 9 → 4 和 4 → 1 所确定自然循环的结点集分别是{4, 5, 6, 7, 8, 9, 10}和{1, 2,3, 4, 5, 6, 7, 8, 9, 10}。

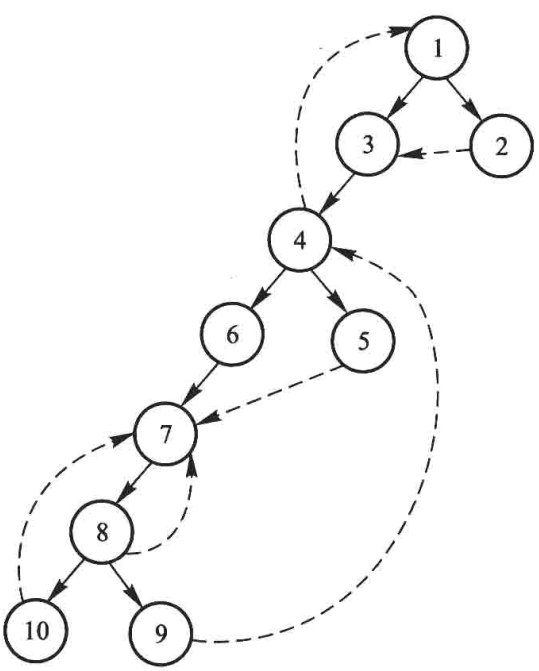


图 9.3 图 9.2 流图的深度优先表示

9.6 下面是一个 C 语言程序：

```
main( ) {
    long i,j;
    while (i) {
        if (j) { i = j; }
    }
}
```

它的编译结果如下(编译器版本见汇编代码最后一行)：

```
        .file "control.c"
        .version "01.01"
gcc2_compiled.:
.text
        .align 4
.globl main
        .type main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp
        nop
        .p2align 4,,7
.L2:
        cmpl $0,-4(%ebp)
        jne .L4
        jmp .L3
        .p2align 4,,7
.L4:
        cmpl $0,-8(%ebp)
        je .L5
        movl -8(%ebp),%eax
        movl %eax,-4(%ebp)
.L5:
        jmp .L2
        .p2align 4,,7
.L3:
.L1:
        leave
        ret
.Lfe1:
        .size main,.Lfe1-main
        .ident "GCC:(GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

它的优化编译结果如下：

```
        .file "control.c"
```

```
        .version "01.01"
gcc2_compiled.:
.text
        .align 4
.globl main
        .type main,@ function
main:
        pushl %ebp
        movl %esp,%ebp
.L7:
        testl %eax,%eax
        je .L3
        testl %edx,%edx
        je .L7
        movl %edx,%eax
        jmp .L7
        .p2align 4,,7
.L3:
        leave
        ret
.Lfe1:
        .size main,.Lfe1-main
        .ident "GCC:(GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

请你分析优化编译器所做的控制流优化。

答 从非优化的编译结果，可以写出 while E_1 do S_1语句和 if E_2 then S_2语句的目标代码结构，它们分别列在下面的左右两边。

```
L2:E1的代码          E2的代码
   真转 L4           假转 L5
   无条件转 L3       S2的代码
L4:S1的代码          L5:
   JMP L2
L3:
```

当这两个语句嵌套时，代码结构变成

```
L2:E1的代码
   真转 L4
```

无条件转 L3

L4:E_2的代码

假转 L5

S_2的代码

L5:JMP L2

L3:

显然,“假转 L5”和“JMP L2”构成连续跳转,应该可以优化。另外 while 语句的“真转 L4”和“无条件转 L3”应该能优化成一个假转。优化编译器所做的控制流优化正是针对这两点进行的。

9.7 UNIX 下的 C 编译命令 cc 的选择项 g 和 O 的解释如下,其中 dbx 的解释是“dbx is an utility for source-level debugging and execution of programs written in C”。试说明为什么用了选择项 g 后,选择项 O 便被忽略。

-g Produce additional symbol table information for dbx(1) and dbxtool(1) and pass-lg option to ld(1) (so as to include the g library, that is: /usr/lib/libg.a).
When this option is given, the-O and-R options are suppressed.

-O[level] Optimize the object code. Ignored when either-g, -go, or-a is used. ...

答 dbx 是一个源语言级的调试工具,它要求编译器产生的目标代码能允许程序员按照源程序的控制结构,逐条语句地观察程序变量值的变化情况。这样,该目标代码的执行流程必须准确地对应到按源程序的控制结构逐条语句地执行源程序。

优化有可能改变程序的执行流程(如代码外提、无用赋值删除)。另外,复写传播导致的无用赋值删除还可能使得变量的存储单元的值没有及时更新(从源程序执行流程的观点看),其更新的值在某个寄存器中。

这样,C 编译命令的这两个选择项的要求是矛盾的,因此只能保留一个。

9.8 请利用代码优化的思想(代码外提和强度削弱等),改写下面 C 语言程序中的循环,得到优化后的 C 语言程序。

```
main( ) {
    int i,j;
    int r[20][10];
    for(i=0; i<20; i++){
        for(j=0; j<10; j++){
            r[i][j] = 10 * i * j;
        }
    }
}
```

答 优化后的 C 语言程序如下：

```
main( ) {
    int i,j,m,n,k;
    int *p;
    int r[20][10];
    p=&r[0][0];
    n=0;
    for(i=0; i<20; i++){
        k=n; n=n+10; m=0;
        for(j=0; j<10; j++){
            *p = m; p++; m=m+k;
        }
    }
}
```

分析 本题用到的代码优化思想主要有代码外提和强度削弱。10 * i 在内循环中是不变计算，可以外提，那么在外循环中就有 k =i * 10，在内循环中有 r[i][j] =k * j。然后把这两个乘法进行强度削弱，在循环中用加法。现在的编译器并不一定要把乘改成加，这取决于目标机器上这两种运算在时间上究竟有多大区别。

另外，在循环中出现下标表达式的地址计算时，会有很多优化的机会。在此若用 C 语言编程，则利用 C 语言提供的指针可以指向程序变量和进行指针算术运算的便利，将下标变量访问部分彻底重写。Pascal 语言做不了这样的优化。不过这种优化是以牺牲程序的可读性为代价的。

9.9 程序中的变量，若在第一次引用前没有置初值的话，则称它为未初始化变量。请运用数据流分析技术，给出计算程序的未初始化变量集合的方法。

答 可以用活跃变量的概念来解决这个问题。如果从某程序点 p 开始的某条路径上有对 x 的引用，并且从点 p 到这个引用点之间没有对 x 的定值，那么变量 x 在点 p 是活跃的。因此，整个程序流图的起始基本块开始点的活跃变量集合，就是未赋初值的变量集合。

在建立数据流方程前，先做一些约定，不考虑过程调用、指针变量和别名等复杂情况，即认为中间代码的一个语句对哪个变量定值是确信无疑的，不存在歧义。

下面开始建立数据流方程。定义 IN[B]是基本块 B 开始点的活跃变量集合，定义 OUT[B]是基本块 B 结束点的活跃变量集合。定义 def_B是基本块 B 中定值且在该定值前没有引用的变量集合，定义 use_B是基本块 B 中引用且在该引用前没有定值的变量集合。

每个基本块的 *def* 和 *use* 可以从自身基本块得到，IN 和 OUT 需要通过数据流方程组求解。方程组如下：

$$\mathrm{IN}[B] = use_B \cup (\mathrm{OUT}[B] - def_B)$$

$$\mathrm{OUT}[B] = \bigcup_{S \text{ is a successor of } B} \mathrm{IN}[S]$$

程序起始块的 IN 集合就是未赋初值的变量集合。

还需要说明一点的是，如果程序中的某个变量没有置初值，但是对该变量没有任何引用，则它不在未初始化变量集合中。另外，若程序中的某个变量没有置初值，对它的引用出现在某个分支中并且这个分支在程序运行时根本不执行，那么这样的变量还是进入了未初始化变量集合，因为数据流分析采取的是稳妥策略。

9.10 如果从程序某点 p 开始的任何一条路径上，在对表达式 e 的任何运算对象定值前，都要计算表达式 e，那么称 e 在 p 点是非常忙的。给出计算非常忙表达式的数据流方程。

答 定义 IN[B]是基本块 B 开始点的非常忙表达式集合，定义 OUT[B]是基本块 B 结束点的非常忙表达式集合。定义 gen_B是基本块 B 中生成的非常忙表达式集合，这里的非常忙表达式 e 是指，B 中计算 e，从 B 的开始点到计算 e 的前一点，没有对 e 的运算对象定值。定义 $kill_B$是基本块 B 注销的非常忙表达式集合。一个表达式 e 被 B 注销，如果 B 中有对 e 的对象赋值。

每个基本块的 *gen* 可以从自身基本块计算。从流图中的所有表达式和一个基本块的赋值语句，可以计算该基本块的 *kill* 集合。IN 和 OUT 需要通过数据流方程组求解。仍然用上一题为简化讨论所做的约定，即中间代码的一个语句对哪个变量定值是确信无疑的，不存在歧义。求解 IN 和 OUT 的数据流方程组如下：

$$\mathrm{IN}[B] = gen_B \cup (\mathrm{OUT}[B] - kill_B)$$

$$\mathrm{OUT}[B] = \bigcap_{S \text{ is a successor of } B} \mathrm{IN}[S]$$

9.11 C 语言程序引用 sizeof(求字节数运算符)时，该运算是在编译该程序时完成，还是在运行该程序时完成？说明理由。

答 sizeof 的参数只能是类型名或者变量名，它求该类型(或该变量的类型)的元素所需空间的大小。而 C 语言所有数据类型的大小都是静态可确定的，因此完全可以在编译时计算 sizeof，事实上，C 的编译器也是这么做的。

9.12 除了用于编译器的代码优化外，控制流分析和数据流分析还能用在什么地方？

答 许多静态程序分析工具需要用到控制流分析和数据流分析技术，甚至更复杂的分析技术，下面举两个例子说明。

程序员经常需要理解别人编写的程序，可能是因为需要扩充功能或进行调试等。理解别人编写的程序不是件容易的事情，需要利用一些方法和工具来帮助理解程序。程序理解是指通过分析、抽象和概括来获取软件系统的知识的演绎过程。程序理解的方法和工具通过帮助软件工

程师从低级的代码中抽取高级的信息,来简化程序理解的复杂性。这些工具可以免除软件工程师很多烦琐的、手工的和易出错的工作,如读代码、查找等。一个简单的例子是切片(slicing),这种技术确定影响变量在某点的值的程序子集,或者由变量在某点的值影响的程序子集。通过分离这样的程序子集,理解一个大程序所需的时间会大大减少。显然,这个例子需要用到控制流分析和数据流分析技术(或是类似的分析技术)。

和程序动态行为有关的错误,如无效的指针引用、不完善的存储分配、使用未置初值的存储单元、在资源上的不适当操作(如试图关闭一个已经关闭了的文件)等,要想找出和定位它们是困难的并且代价较大,通常需要细致的测试。这样就出现了查找程序动态错误的静态分析器,它们经由源程序和存储建模来跟踪执行路径,报告可能的动态错误。这样的静态分析器需要对源程序进行控制流分析。例如,一个 C 语言程序如下:

```
(1)  #include <stdlib.h>
(2)  #include <stdio.h>
(3)  char   * f(int size) {
(4)       char * result;
(5)       if (size > 0)
(6)            result = (char *) malloc(size);
(7)       if (size == 1)
(8)            return NULL;
(9)       result[0] = 0;
(10)      return result;
(11) }
```

现在的静态分析器能发现如下的错误。

1. 如果第 5 行和第 7 行的条件都为真,则会引起内存泄漏(memory leak),其执行路径是 5,6,7 和 8。

2. 如果第 5 行的条件为假,则第 7 行的条件也为假。这时会引起访问没有初始化的指针 result 所指向的对象,其执行路径是 5,7 和 9。

3. 如果第 5 行的条件为真,第 7 行的条件为假,并且内存已经用尽(malloc 返回 0),这时第 9 行的指针使用是不正确的,其执行路径是 5,6,7,9。

9.13 一个 C 语言的函数如下:

```
func(i) long i; {
    long j;
    j = i-1;
    func(j);
}
```

下面左右两边的汇编代码是两个不同版本 GCC 编译器为该函数产生的代码。左边的代码在调用 func 之前将参数压栈，调用结束后将参数退栈。右边代码对参数传递的处理方式没有实质区别。请叙述右边代码对参数传递的处理方式并推测它带来的优点。

```
func:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $4, %esp
    movl    8(%ebp), %edx
    decl    %edx
    movl    %edx, -4(%ebp)
    movl    -4(%ebp), %eax
    pushl   %eax
    call    func
    addl    $4, %esp
    leave
    ret
```

```
func:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    movl    8(%ebp), %eax
    decl    %eax
    movl    %eax, -4(%ebp)
    movl    -4(%ebp), %eax
    movl    %eax, (%esp)
    call    func
    leave
    ret
```

答 左边的编译器版本：一般只为局部变量分配空间。调用函数前，用若干次 pushl 指令将参数压栈，返回后用 addl $n, %esp 一次将所有参数退栈（常数 n 根据调用前做了多少次 pushl 来决定）。可见主教材例 6.4。

右边的编译器版本：除了为局部变量分配空间外，同时还为本函数中出现的函数调用的参数分配空间，并且参数所用空间靠近栈顶。调用函数前，用 movl 指令将参数移入栈顶，调用结束后无需参数退栈指令。

其优点是每次函数调用结束后不需要执行 addl $n, %esp 指令，另外提高了优化的可能性。

***9.14** 考虑一个类 Pascal 的语言，其中所有的变量都是整型（不需要显式声明），并且仅包含赋值语句、读语句、写语句、条件语句和循环语句。下面的产生式定义了该语言的语法（其中 **lit** 表示整型常量；**OP** 的产生式没有给出，因为它和下面讨论的问题无关）。

Program→*Stmt*

Stmt→**id** := *Exp*

Stmt→**read** (**id**)

Stmt→**write** (*Exp*)

Stmt→*Stmt* ; *Stmt*

Stmt→ **if** (*Exp*) **then begin** *Stmt* **end else begin** *Stmt* **end**

Stmt→**while** (*Exp*) **do begin** *Stmt* **end**

Exp→**id**

Exp→**lit**

Exp→*Exp* **OP** *Exp*

定义 *Stmt* 的两个属性：*MayDef* 表示它可能定值的变量集合，*MayUse* 表示它可能引用的变量集合。

(a) 写一个语法制导定义或翻译方案，它计算 *Stmt* 的 *MayDef* 和 *MayUse* 属性。

(b) 基于 *MayDef* 和 *MayUse* 属性，说明 $Stmt_1$；$Stmt_2$ 和 $Stmt_2$；$Stmt_1$ 在什么情况下有同样的语义。

答 (a) 计算 *Stmt* 的 *MayDef* 和 *MayUse* 属性的语法制导定义如下：

Program→*Stmt*

Stmt→**id** := *Exp*

{ *Stmt.MayDef* = {**id**.*name*}; *Stmt.MayUse* = *Exp.MayUse*; }

Stmt→**read** (**id**)

{*Stmt.MayUse* = ∅; *Stmt.MayDef* = {**id**.*name*}; }

Stmt→**write** (*Exp*)

{ *Stmt.MayDef* := ∅; *Stmt.MayUse* = *Exp.MayUse*; }

$Stmt \rightarrow Stmt_1$; $Stmt_2$

{ $Stmt.MayUse = Stmt_1.MayUse \cup Stmt_2.MayUse$;

$Stmt.MayDef = Stmt_1.MayDef \cup Stmt_2.MayDef$; }

Stmt→ **if** (*Exp*) **then begin** $Stmt_1$ **end else begin** $Stmt_2$ **end**

{ $Stmt.MayUse = Stmt_1.MayUse \cup Stmt_2.MayUse \cup Exp.MayUse$;

$Stmt.MayDef = Stmt_1.MayDef \cup Stmt_2.MayDef$; }

Stmt→**while** (*Exp*) **do begin** $Stmt_1$ **end**

{ $Stmt.MayUse = Stmt_1.MayUse \cup Exp.MayUse$;

$Stmt.MayDef = Stmt_1.MayDef$;}

Exp→**id**{ *Exp.MayUse* = {**id**.*name*}; }

Exp→**lit**{ *Exp.MayUse* = ∅; }

$Exp \rightarrow Exp_1$ OP Exp_2 { $Exp.MayUse = Exp_1.MayUse \cup Exp_2.MayUse$; }

(b) 满足下面的条件时，交换 $Stmt_1$ 和 $Stmt_2$ 的次序不会影响语义。

$Stmt_1.MayDef \cap Stmt_2.MayUse = \varnothing$并且

$Stmt_2.MayDef \cap Stmt_1.MayUse = \varnothing$并且

$Stmt_1.MayDef \cap Stmt_2.MayDef = \varnothing$

9.15 考虑一个类 Pascal 的语言，其中所有的变量都是整型(不需要显式声明)，并且仅包含赋值语句、读语句、写语句、条件语句和循环语句。下面的产生式定义了该语言的语法(其中 **lit** 表示整型常量；**OP** 的产生式没有给出，因为它和所讨论的问题无关)。

Program→*Stmt*

Stmt→**id** := *Exp*

Stmt→**read** (**id**)

Stmt→**write** (*Exp*)

Stmt→*Stmt* ; *Stmt*

Stmt→ **if** (*Exp*) **then begin** *Stmt* **end else begin** *Stmt* **end**

Stmt→**while** (*Exp*) **do begin** *Stmt* **end**

Exp→**id**

Exp→**lit**

Exp→*Exp* **OP** *Exp*

定义 *Stmt* 的两个属性：*Def* 表示在 *Stmt* 中一定会定值且在该定值前没有引用的变量集合，*MayUse* 表示在 *Stmt* 中有引用且在该引用前可能没有定值的变量集合。

(a) 写一个语法制导定义或翻译方案，它计算 *Stmt* 的 *Def* 和 *MayUse* 属性。

(b) 基于上面的计算，程序可能未赋初值的变量集合从哪儿可以得到？

可能未赋初值的变量是这样定义的：若存在从程序开始点到达变量 a 某引用点的一条路径，在这条路径上没有对变量 a 赋值，则变量 a 属于程序可能未赋初值的变量集合。

答 (a) 计算 *Stmt* 的 *Def* 和 *MayUse* 属性的语法制导定义如下：

Program→*Stmt* { *Program*. *MayUse* = *Stmt*.*MayUse* }

Stmt→**id** := *Exp*

{ *Stmt*.*Def* = {**id**.*name*} − *Exp*.*MayUse* ; *Stmt*.*MayUse* = *Exp*.*MayUse*; }

Stmt→read (**id**)

{ *Stmt*.*MayUse* = ∅; *Stmt*.*Def* = {**id**.*name*} ; }

Stmt→write (*Exp*)

{ *Stmt*.*Def* = ∅; *Stmt*.*MayUse* = *Exp*.*MayUse*; }

$Stmt \rightarrow Stmt_1$; $Stmt_2$

{ $Stmt.MayUse = Stmt_1.MayUse \cup (Stmt_2.MayUse - Stmt_1.Def)$;

$Stmt.Def = Stmt_1.Def \cup (Stmt_2.Def - Stmt_1.MayUse)$; }

Stmt→ **if** (*Exp*) **then begin** $Stmt_1$ **end else begin** $Stmt_2$ **end**

{ $Stmt.MayUse = Stmt_1.MayUse \cup Stmt_2.MayUse \cup Exp.MayUse$;

$Stmt.Def = (Stmt_1.Def \cap Stmt_2.Def) - Exp.MayUse$; }

Stmt→**while** (*Exp*) **do begin** $Stmt_1$ **end**

{ $Stmt.MayUse = Stmt_1.MayUse \cup Exp.MayUse$;

$Stmt.Def = Stmt_1.Def - Exp.MayUse$; }

Exp→**id** { *Exp*.*MayUse* = {**id**.*name*} ; }

Exp→**lit** { *Exp*.*MayUse* = ∅; }

$Exp \rightarrow Exp_1$ **OP** Exp_2 { $Exp.MayUse = Exp_1.MayUse \cup Exp_2.MayUse$; }

(b) Program 的属性 *MayUse* 就是程序可能未赋初值的变量集合。

9.16 考虑一个简单语言,其中所有的变量都是整型(不需要显式声明),并且仅包含赋值语句、读语句和写语句。下面的产生式定义了该语言的语法(其中 **lit** 表示整型常量;**OP** 的产生式没有给出,因为它和所讨论的问题无关)。

Program→*StmtList*

StmtList →*Stmt StmtList* | *Stmt*

Stmt→**id** := *Exp*; | *read* (**id**); | **write** (*Exp*);

Exp→**id** | **lit** | *Exp* **OP** *Exp*

把不影响 write 语句输出值的赋值(包括通过 read 语句来赋值)称为无用赋值,写一个语法指导定义,它确定一个程序中出现过赋予无用值的变量集合(不需要知道无用赋值的位置)和没有置初值的变量集合(不影响 write 语句输出值的未置初值变量不在考虑之中)。

非终结符 *StmtList* 和 *Stmt* 用到下面三个属性(可根据需要来定义其他文法符号的属性):

uses_in:在本语句表或语句入口点的引用变量集合,它们的值影响在该程序点后的输出。

uses_out:在本语句表或语句出口点的引用变量集合,它们的值影响在该程序点后的输出。

useless:本语句表或语句中出现的无用赋值变量集合。

答 *Exp* 的属性 *uses* 表示它引用的变量集合。*Program* 的 *useless* 和 *no_initial* 分别表示程序的无用赋值变量集合和未置初值变量集合。

Program→ *StmtList*	$StmtList.uses_out = \varnothing$;
	$Program.useless = StmtList.useless$;
	$Program.no_initial = StmtList.uses_in$;
StmtList → *Stmt* $StmtList_1$	$StmtList_1.uses_out = StmtList.uses_out$;
	$Stmt.uses_out = StmtList_1.uses_in$;
	$StmtList.uses_in = Stmt.uses_in$;
	$StmtList.useless = StmtList_1.useless \cup Stmt.useless$;
StmtList→ *Stmt*	$Stmt.uses_out = StmlList.uses_out$;
	$StmlList.uses_in = Stmt.uses_in$;
	$StmtList.useless = Stmt.useless$
Stmt → **id** := *Exp*;	$Stmt.useless =$ **if** (**id**.*lexeme* $\in Stmt.uses_out$) $\varnothing$ **else** { **id**.*lexeme* };
	$Stmt.uses_in =$ **if** (**id**.*lexeme* $\in Stmt.uses_out$) ($Stmt.uses_out$ − { **id**.*lexeme* }) $\cup$ $Exp.uses$ **else** $Stmt.uses_out$;
Stmt →read (**id**);	$Stmt.useless =$ **if** (**id**.*lexeme* $\in Stmt.uses_out$) $\varnothing$ **else** { **id**.*lexeme* };

	$Stmt.uses_in = Stmt.uses_out - \{\textbf{id}.lexeme\}$
$Stmt \rightarrow$write (Exp) ;	$Stmt.useless = \varnothing$, $Stmt.uses_in = Stmt.uses_out \cup Exp.uses$
$Exp \rightarrow$ **id**	$Exp.uses = \{\textbf{id}.lexeme\}$
$Exp \rightarrow$ **lit**	$Exp.uses = \varnothing$
$Exp \rightarrow Exp_1\ OP\ Exp_2$	$Exp.uses = Exp_1.uses \cup Exp_2.uses$

9.17 下面左边的函数被编译器 GCC:(GNU) 3.3.5 (Debian 1:3.3.5-13)优化成右边的代码。若认为该优化结果不对或该优化不合理,请阐述理由。若认为该优化结果是对的,则请说明实施了哪些优化,并解释循环优化结果的合理性。

```
f(a,b,c,d,x,y,z)int a,b,c,d,x,y,z; {
    while(a<b) {
        if (c<d)
            x = x + z;
        else
            x = y -z;
    }
}
```

```
f:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jge.L9
.L7:
    jl.L7
.L9:
    popl %ebp
    ret
```

答 该优化结果是对的。它主要使用了代码外提、无用赋值删除和其他无用代码删除技术。对于该函数中的循环来说:

1. 由于 c<d 是循环不变计算,条件语句每次执行选择的是同一个分支。由于其他地方没有对 x 的引用,很容易判断条件语句的两个分支的 x = x + z 和 x = y –z 都是无用赋值,可以删除。进一步,条件判断 c<d 也可以删除,导致循环体为空。

2. 循环控制的判断 a<b 也是循环不变计算,外提为 movl 8(%ebp), %edx; movl 12(%ebp), %eax; cmpl %eax, %edx 三条指令。

3. 循环部分剩余的代码变成基于标志位判断的两个分支,jge .L9 表示循环结束,jl .L7 表示继续执行循环。

9.18 下面左栏的 C 语言程序计算 5!,结果是 120。经优化编译器(编译器版本见汇编代码最后一行)生成的汇编程序跟在源程序的后面(先左栏后右栏),其中在省略号处略去了 main 函

数的汇编程序。请问优化编译器对 factorial 函数进行了怎样的优化?

```
factorial(long n, long res) {
    if (n==0) return res;
    else return factorial(n-1, n * res);
}
main() {
    printf("%d\n", factorial(5,1));
}
```

汇编程序:

```
    .file"recursion.c"
    .text
    .p2align 4,,15
.globl factorial
    .type    factorial, @function
factorial:
    pushl    %ebp
    movl     %esp, %ebp
    movl     8(%ebp), %edx
    movl     12(%ebp), %eax
    testl    %edx, %edx
    je.      .L3
    .p2align 4,,7
.L6:
    imull    %edx, %eax
    subl     $1, %edx
    jne.     .L6
.L3:
    popl     %ebp
    ret
    .size    factorial, .-factorial
    ...      ...
    .ident   "GCC:(GNU) 4.2.3 (Debian 4.2.3-5)"
    .section.note.GNU-stack,"",@progbits
```

答 使用了尾递归优化。对于尾递归调用,运行时不用建立新的活动记录,就在当前活动记录完成计算。针对这个例子叙述具体步骤如下:

1. 计算调用的实在参数 n-1 和 n * res,仍存放在相应的寄存器中(eax 存放函数返回值)。

2. 转到本函数第一条语句的起始地址继续执行(注意,不是本函数的入口地址,因为函数入口是一段调用序列代码,然后才是第一条语句的代码)。

9.19 对于主教材上例 6.4 的程序(见下面 C 代码),用编译器 GCC:(GNU) 4.2.3 (Debian 4.2.3-5)编译后所生成的优化代码(见下面中间栏的汇编代码)同教材中例 6.4 给出的代码(见下面左栏的汇编代码)略有不同。请指出该较新版本编译器完成的优化(或采用的不同代码生成策略)。给一点提示:若把调用语句改成 func(j, j),则所生成的优化代码和下面中间的汇编代码的区别见下面右栏的说明。

```
func(i) long i; {
    long j; j=i-1; func(j);
}
```

```
func:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $4, %esp
    movl    8(%ebp), %edx
    decl    %edx
    movl    %edx, -4(%ebp)
    movl    -4(%ebp), %eax
    pushl   %eax
    call    func
    addl    $4, %esp
.L1:
    leave
    ret
```

```
func:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $4, %esp
    movl    8(%ebp), %eax
    subl    $1, %eax
    movl    %eax, (%esp)
    call    func
    leave
    ret
```

```
    该指令改成
subl $8, %esp
    在该指令和下条
指令间增加
movl %eax, 4(%esp)
```

答 主教材例 6.4 所用的编译器一般只为局部变量分配空间。调用函数前,用若干次 pushl 指令将参数压栈,返回后用 addl $n, %esp 一次将所有参数退栈(常数 n 根据调用前做了多少次 pushl 来决定)。

本题所用编译器除了为局部变量分配空间外,同时还为函数体中出现的函数调用的参数分配空间,并且参数所用空间靠近栈顶,以邻近被调用者。在调用函数之前用 movl 指令将参数移入栈顶,在函数调用之后无须用 addl $n, %esp 指令来退栈。免去退栈指令是这种策略的一个优点。

所做的优化主要是用寄存器 eax 保存 j 的值,不必为 j 分配空间。

***9.20** 下面的 C 程序分别经非优化编译和 2 级以上(含 2 级)的优化编译后,生成的两个目标程序运行时的表现不同(编译器是 GCC:(GNU) 4.2.3 (Debian 4.2.3-5))。请回答它们运行时的表现有何不同,并说明原因。

```
int f(int g( ) ) {
    return g(g);
}
main( ) {
    f(f);
}
```

答 该程序不会终止,陷入无穷的递归调用。在非优化的情况下,无穷的递归调用导致运行栈的溢出,引起系统报告 Segmentation fault。在高级全局优化的情况下,可以知道函数 f 的 return

语句中的函数调用是尾递归调用。函数 f 被优化为把当前活动记录作为原本要新增的活动记录,并把调用指令改为跳转指令,导致程序陷入死循环但不会出现运行栈的溢出。

下面是函数 f 在 2 级优化后的目标代码:

```
f:
        pushl   %ebp
        movl    %esp,%ebp
        movl    8(%ebp),%ecx
        popl    %ebp
        jmp     *%ecx
```

其中 8(%ebp)是存放参数的单元,其值就是 f 的入口地址。

第 10 章　依赖于机器的优化

一、主要知识点

由于课时的限制，主教材第 10 章内容在编译原理课程中一般都不介绍，在体系结构的课程中有可能介绍得更多一些。本章主要应了解以下内容。

1. 提取并行的数据相关性分析。
2. 代码调度的基本概念。
3. 基本块调度和全局调度的技术。
4. 软件流水线技术。
5. 循环程序的并行化和数据局部性。

二、习题精选与解析

10.1　严格按照括号指定的次序，计算表达式((u + v) + (w + x)) + (y + z)，也就是不利用交换律或结合律来改变加法的计算次序。像主教材上例 10.3 那样给出提供最大可能并行的寄存器级的机器代码。

答　如果对每个中间结果使用不同寄存器的话，则完成该表达式的计算只需要 4 步，具体的并行方式见图 10.1。

R1 = u	R2 = v	R3 = w	R4 = x	R5 = y	R6 = z
R7 = R1 + R2	R8 = R3 + R4	R9 = R5 + R6			
R10 = R7 + R8					
R11 = R10 + R9					

图 10.1　((u + v) + (w + x)) + (y + z)的并行计算

10.2　为下面的表达式完成习题 10.1 的要求。

(a)(u + (v + (w + x))) + (y + z)

(b)(u + (v + w)) + (x + (y + z))

若用最小化寄存器数来取代最大化并行，将需要多少步计算？用最大化并行节省了几步？

答　(a)对于表达式(u + (v + (w + x))) + (y + z)，用最大化并行方式只需要 5 步就可以

完成计算，见图 10.2。

R1 = u	R2 = v	R3 = w	R4 = x	R5 = y	R6 = z
R7 = R3 + R4	R8 = R5 + R6				
R9 = R2 + R7					
R10 = R1 + R9					
R11 = R10 + R8					

图 10.2 (u + (v + (w + x))) + (y + z)的并行计算

用最小化寄存器数方式的代码如下，需要 4 个寄存器和 11 步计算。

```
LD      R1,u
LD      R2,v
LD      R3,w
LD      R4,x
ADD     R3,R3, R4
ADD     R2,R2,R3
ADD     R1,R1,R2
LD      R2,y
LD      R3,z
ADD     R2,R2,R3
ADD     R1,R1,R2
```

最大化并行方式节省了 6 步计算。

(b) 对于表达式(u + (v + w)) + (x + (y + z))，用最大化并行方式只需要 4 步就可以完成计算，见图 10.3。

R1 = u	R2 = v	R3 = w	R4 = x	R5 = y	R6 = z
R7 = R2 + R3	R8 = R5 + R6				
R9 = R1 + R7	R10 = R4 + R8				
R11 = R9 + R10					

图 10.3 (u + (v + w)) + (x + (y + z))的并行计算

用最小化寄存器数方式的代码如下，需要 4 个寄存器和 11 步计算。

```
LD      R1,u
LD      R2,v
LD      R3,w
ADD     R2,R2, R3
ADD     R1,R1,R2
LD      R2,x
```

```
LD      R3,y
LD      R4,z
ADD     R3,R3,R4
ADD     R2,R2,R3
ADD     R1,R1,R2
```

最大化并行方式节省了 7 步计算。

10.3 习题 10.1 的表达式可以由下面的指令序列来执行。如果所需要的并行都能得到，那么这些指令的执行需要几步？

```
LD R1, u            // R1 = u
LD R2, v            // R2 = v
ADD R1, R1, R2      // R1 = R1 + R2
LD R2, w            // R2 = w
LD R3, x            // R3 = x
ADD R2, R2, R3      // R2 = R2 + R3
ADD R1, R1, R2      // R1 = R1 + R2
LD R2, y            // R2 = y
LD R3, z            // R3 = z
ADD R2, R2, R3      // R2 = R2 + R3
ADD R1, R1, R2      // R1 = R1 + R2
```

答 在不修改代码的情况下，由于数据相关的约束，上述指令序列的执行需要 8 步，见图 10.4。

分析 若多使用一个寄存器，并采用较好的寄存器分配策略，则仅需 5 步即可，具体代码如下，并行计算见图 10.5。

```
LD R1, u            // R1 = u
LD R2, v            // R2 = v
ADD R1, R1, R2      // R1 = R1 + R2
LD R3, w            // R3 = w
LD R4, x            // R4 = x
ADD R3, R3, R4      // R3 = R3 + R4
ADD R1, R1, R3      // R1 = R1 + R3
LD R2, y            // R2 = y
LD R4, z            // R4 = z
ADD R2, R2, R4      // R2 = R2 + R4
ADD R1, R1, R2      // R1 = R1 + R2
```

R1 = u	R2 = v	R3 = x
R1 = R1 + R2		
R2 = w		
R2 = R2 + R3		
R1 = R1 + R2	R3 = z	
R2 = y		
R2 = R2 + R3		
R1 = R1 + R2		

图 10.4 ((u + v) + (w + x)) + (y + z) 的并行计算

R1 = u	R2 = v	R3 = w	R4 = x
R1 = R1 + R2	R3 = R3 + R4		
R1 = R1 + R3	R2 = y	R4 = z	
R2 = R2 + R4			
R1 = R1 + R2			

图 10.5　习题 10.3 代码修改后的并行计算

10.4　画出图 10.6 各代码片段的数据依赖图。

LDR1, a	LDR1, a	LDR1, a
LDR2, b	LDR2, b	LDR2, b
SUB R3, R1, R2	SUB R1, R1, R2	SUB R3, R1, R2
ADD R2, R1, R2	ADD R2, R1, R2	ADD R4, R1, R2
ST a, R3	ST a, R1	ST a, R3
ST b, R2	ST b, R2	ST b, R4
(a)	(b)	(c)

图 10.6　习题 10.4 的机器代码

答　各代码片段的数据依赖图见图 10.7,其中资源预留表略去。

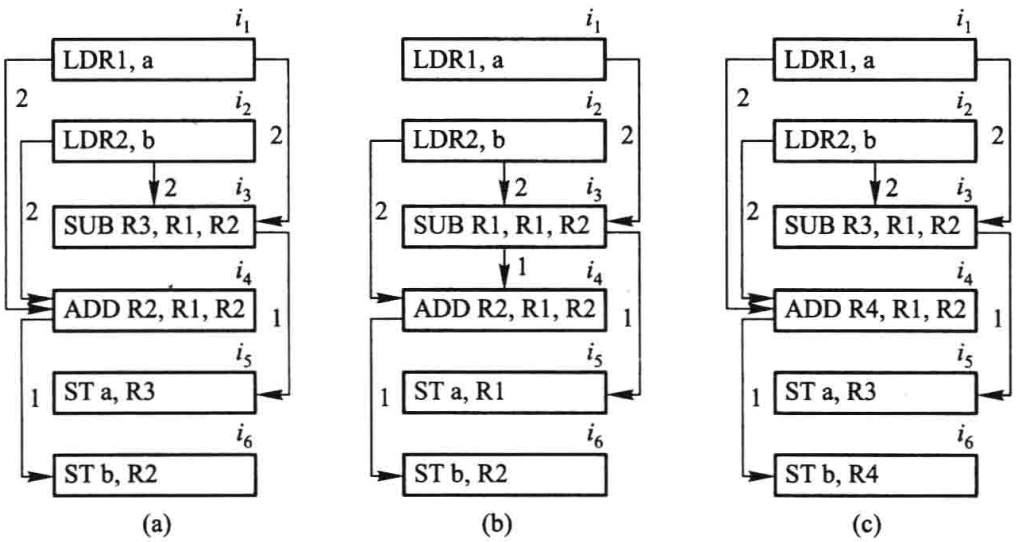


图 10.7　习题 10.4 的数据依赖图

10.5　假定机器有一个算术运算部件 ALU(用于 ADD 和 SUB 操作)和一个内存资源 MEM(用于 LD 和 ST 操作)。再假定,除了 LD 操作需要 2 个周期外,所有操作都只需要 1 个周期。像主教材上例 10.6 那样,在 LD 操作一个内存单元启动 1 周期后,操作同一个内存单元的 ST 可以启动。请为图 10.6 的各代码片段寻找最短调度。

答　图 10.8 是代码片段(a)的最短调度。由于读取指令的延迟是 2,因此不论两条读取指令和两条运算指令怎么调度,第 3 周期都没有可以调度的指令。

代码片段(b)和(c)的调度结果基本上一样的,在此从略。



图 10.8　表调度应用于图 10.6(a)代码片段的结果

10.6　根据下面的资源情况,重做习题 10.5。

(a) 机器有一个 ALU 资源和两个 MEM 资源。

(b) 机器有两个 ALU 资源和一个 MEM 资源。

(c) 机器有两个 ALU 资源和两个 MEM 资源。

答　图 10.9 仅给出在有两个 ALU 资源和两个 MEM 资源的情况下,表调度应用于图 10.6(a)代码片段的结果。其余的在此从略。由于读取指令的延迟是 2,不论怎么调度,第 2 周期都没有可以调度的指令。

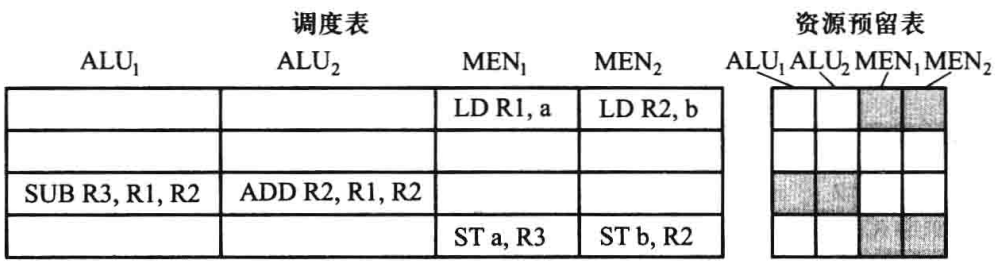


图 10.9　习题 10.6 的表调度结果的一个示例

10.7　假定机器模型和习题 10.5 的一样。

(a) 为下面的代码片段画出数据依赖图。

```
LD R1, a
ST b, R1
LD R2, c
ST c, R1
LD R1, d
ST d, R2
ST a, R1
```

(b) 在(a)的数据依赖图中,关键路径是哪几条?

答　(a) 数据依赖图见图 10.10。

(b) 关键路径有一条,包括结点 i_3,i_4,i_5和 i_7。

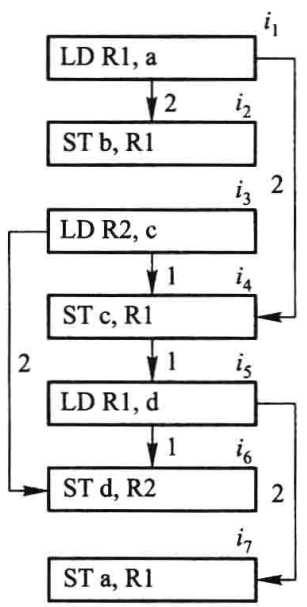


图 10.10　习题 10.7 的数据依赖图

10.8 考虑下面的代码片段

```
if (x == 0) a = b;
else a = c;
d = a;
```

假定机器使用主教材上例 10.6 的延迟模型，还假定机器一次可以同时执行任意两条指令。找出该程序片段可能的最短执行。

答 参照教材上例 10.8，假定变量 x，a，b，c 和 d 的地址不同，并且它们的地址分别保存在寄存器 R1 到 R5 中。那么该程序片段的最短执行见图 10.11。

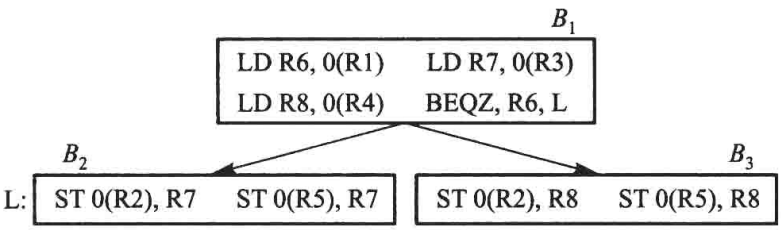


图 10.11 习题 10.8 的最短执行流图

分析 主教材上例 10.8 和上述答案都忽略了一件事。这三个基本块的代码不论按什么次序形成代码序列，总还需要一条无条件跳转指令，才能正确地离开该代码序列去执行后继代码。

***10.9** 使用主教材上例 10.10 的机器模型，为循环

```
for (i = 1; i < n; i++) {
    A[i] = B[i -1] + 1;
    B[i] = A[i -1] + 2;
}
```

找出最小启动间隔和对此间隔的一种调度。注意，迭代的计数通过寄存器的自动减 1 来完成，无须单独完成此计数的操作。

答 假定寄存器 R1 和 R2 分别保存数组 A 和 B 的开始地址，寄存器 R10 保存值 n-1，它在迭代外计算。对每次迭代的最佳紧凑调度如下，其中用到 MOV 指令，用于保存 A[i -1]的值。

```
    // R1 = &A; R2 = &B
    // R10 = n -1
    LD R4, 0(R2++)
    LD R3, 0(R1++)
L:  MOV R5, R3
    ADD R3, R4, #1
    ADD R4, R5, #2
```

```
        ST 0(R1++), R3
        ST 0(R2++), R4
        BL R10, L
```

软件流水化的代码每 3 周期开始一次迭代,这是最小启动间隔。流水化的代码如下,其中 MOV 指令当成读取操作看待。

```
        // R1 = &A; R2 = &B
        // R10 = n –1
        LD R4, 0(R2++)
        LD R3, 0(R1++)
        MOV R5, R3
                        ADD R3, R4, #1
                        ADD R4, R5, #2
L:MOV R5, R3                                ST 0(R1++), R3
                        ADD R3, R4, #1      ST 0(R2++), R4
                        ADD R4, R5, #2                          BL R10, L
                                            ST0(R1++), R3
                                            ST 0(R2++), R4
```

10.10 主教材上图 10.25 基于分块的矩阵乘法算法没有把矩阵 Z 初始化为零,请加上把 Z 初始化为零的步骤。

答 加上初始化 Z 的语句后,程序如下:

```
for (ii = 0; ii < n; ii = ii + b)
    for (jj = 0; jj < n; jj = jj + b) {
        for (i = ii; i < ii + b; i++)
            for (j = jj; j < jj + b; j++)
                Z[i][j] = 0.0;
        for (kk = 0; kk < n; kk = kk + b)
            for (i = ii; i < ii + b; i++)
                for (j = jj; j < jj + b; j++)
                    for (k = kk; k < kk + b; k++)
                        Z[i][j] = Z[i][j] + X[i][k]*Y[k][j];
    }
```

分析 把主教材上图 10.24 矩阵分块中的每个块看成只有一个元素,也就是把 b 看成等于 1,则上述程序本质上与主教材上图 10.22 的程序一致,从而不难理解上述程序中所加的对 Z 初始化的代码是正确的。

第 11 章　编译系统和运行时系统

一、主要知识点

主教材第 11 章的内容不直接面向编译技术本身，但是了解从源程序到可运行目标代码的实际翻译过程，可以启发在软件工程中更好地运用所学的编译知识，因此本章内容也值得学习。

本章的内容完全是介绍性的，没有需要重点掌握的部分，但解答本章的习题仍然需要灵活运用所学的编译知识。

二、习题精选与解析

11.1　如果 cfile 是一个 C 语言源程序（注意，该文件名没有后缀），在 x86/Linux 系统上，某版本的编译命令

```
cc cfile
```

的结果是输出是错误信息：

```
/usr/bin/ld: cfile: file format not recognized: treating as linker script
/usr/bin/ld: cfile: 1: parse error
collect2: ld returned 1 exit status
```

请解释为什么会是这样的报错信息。

答　在 UNIX 机器上，从 C 语言源程序得到其可执行目标程序，要经过如下所示的几步：

1. 编译器将后缀名为 c 的源程序编译成后缀名为 s 的汇编语言程序；
2. 汇编器将后缀名为 s 的汇编语言程序汇编成后缀名为 o 的可重定位机器代码；
3. 连接器将后缀名为 o 的可重定位机器代码装配成可执行的目标程序（可能连接了若干个库函数）。

编译命令 cc 是一个批命令，它根据文件的后缀名和命令选择项来选择上述三步中的若干步对文件进行处理。文件名 cfile 没有后缀，因此 cc 命令没有对它进行编译和汇编，直接将它交给最后一步的连接装配器（ld）。连接装配器发现该文件的格式不对，因此报告以上错误。

11.2　C 语言程序（存储为一个文件）

```
long gcd(p,q) long p,q; {
```

```
    if (p%q == 0)
        return q;
    else
        return gcd(q, p%q);
}
main() {
    printf("\n%ld\n",gcdx(4,12));
}
```

在 x86/Linux 系统上用某版本的 gcc 命令得到的编译结果如下：

```
In function 'main':
undefined reference to 'gcdx'
ld returned 1 exit status.
```

请问，这个 gcdx 没有定义，是在编译时发现的，还是在连接时发现的？试说明理由。

答 从上面的编译结果知道，gcdx 没有定义是在连接时发现的，因为 ld 是 Linux 上的连接程序。这只是直观的解释，具体理由如下。

C 语言是以文件为单位独立编译的。C 语言的函数可以在一个文件中定义，而在另一个文件中引用。因此在编译时若发现一个名字尚未定义，并不会立即报错，而是要等到连接找不到该名字的定义点时才报错。

11.3 一个 C 语言程序如下：

```
extern int a;
static int b;
int c;
main() {
    b = a;
}
```

它在 x86/Linux 系统上编译后生成的汇编代码如下（编译器版本见汇编代码最后一行），请说明编译时对 extern 变量的处理和外部变量的处理有什么区别。

```
    .file "extern.c"
    .version "01.01"
gcc2_compiled.:
.text
    .align 4
.globl main
    .type main,@function
```

```
main:
        pushl %ebp
        movl %esp,%ebp
        movl a,%eax
        movl %eax,b
.L1:
        leave
        ret
.Lfe1:
        .size main,.Lfe1-main
        .local  b
        .comm   b,4,4
        .comm   c,4,4
        .ident  "GCC:(GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

答 从汇编代码可以看出,外部变量 c 和静态外部变量 b 都在本汇编文件中分配存储单元,它们用 .comm 伪指令分配空间。而 extern 变量 a 在本汇编文件中没有分配存储单元,但本汇编文件中有对它的引用。在和其他文件连接时,若在其他文件中找不到这个 extern 变量的定义点,那么报告该 extern 变量未定义。

11.4 一个 C 程序的三个文件的内容如下:

```
head.h:
short int a = 10;

file1.c:
#include "head.h"
main() {
}

file2.c:
#include "head.h"
```

在 x86/Linux 系统上用某版本的命令

```
cc file1.c file2.c
```

编译,其结果报错的主要信息如下:

```
multiple definition of 'a'
```

试分析为什么会报这样的错误。

答 根据 C 语言有关文件包含的处理可以知道，在编译 file1. c 时，生成的目标文件 file1. o 中有变量 a 的定义；同样，在编译 file2. c 时，生成的目标文件 file2. o 中也有变量 a 的定义。当连接 file1. o 和 file2. o 时，显然会报告 a 有重复定义。

分析 在工程中，编程规约一般都要求在头文件中只有宏定义、数据类型定义和外部变量说明等这样一些“声明”信息，而不包含变量定义和函数定义等这样一些“定义”信息，但是要求另外单独用一个 C 文件对头文件中出现的外部变量加以定义。这样可以避免本题遇到的问题。

11.5 在工程中经常会碰到头文件被重复引用的情况，例如构成一个程序的部分文件的包含关系如下：

1. 文件 compiler. c 中有#include "parse. h"和#include "semantics. h"。
2. 文件 parse. h 中有#include "token. h"。
3. 文件 semantics. h 中也有#include "token. h" 。

这样，在编译 compiler. c 的时候，会出现重复引用 token. h 错误。如果遵循上一题指出的原则：头文件中只存放“声明”而不存放“定义”，从而避免出现重复定义，但是这样却避免不了因多次分析 token. h 而增加的编译时间开销。怎样用 C 语言的条件编译命令来解决这个问题？

答 可以用条件编译命令#ifndef 来解决这个问题。如果在文件 token. h 中有某个标识符的定义，例如#define debug 1，那么将文件 token. h 改成如下形式：

```
#ifndef debug
    …  /* token. h 原来的内容 */
#endif
```

经这样修改后可以避免对文件 token. h 作多次分析。

11.6 一些 C 程序设计的教材上指出：“在需要使用标准 I/O 库中的函数时，应在程序前使用#include <stdio. h>预编译命令，但在用 printf 和 scanf 函数时，则可以不要。”（备注：有些编译器给出警告错误）事实上，并非仅限于这两个函数。例如下面的 C 程序编译后运行时输出字符 A 并换行，它并没有预编译命令#include <stdio. h>。试解释为什么。

```
main( ) {
    putchar('A');
    putchar('\n');
}
```

答 一个 C 语言文件是否需要包含一个头文件，取决该 C 语言文件编译时，是否需要用到这个头文件中的宏定义、类型定义等，而不在于是否引用这个头文件中给出原型的外

部函数。这是因为 C 语言编译器对用传统语法形式给出的函数声明，并不进行实在参数和形式参数的个数和类型是否一致的检查（在第 5 章的习题中已经提到 C 语言函数声明的两种不同形式）。本习题的程序仅引用函数 putchar，因此是否包含 stdio. h 不会影响该程序的编译结果。

分析 在一个 C 语言文件中，如果在调用某函数前没有给出该函数的定义或声明，则 C 语言编译器将默认该函数的值类型为 int。例如下面是 C 程序文件 head. c 的内容：

```
main( ) {
    char *s = gets( );
    gets( );
}
```

它引用 gets 函数。gets 函数的返回值类型在 stdio. h 中被声明为 char *，由于没有包含 stdio. h，则 C 编译器将 gets 函数的返回值视为 int 类型。

在用 gcc 编译该程序文件时，会输出下面这样的警告：

```
head. c: In function 'main':
head. c:2: warning: initialization makes pointer from integer without a cast
```

11.7 把下面左边的文件 file1. c 提交给编译器，编译器没有报告任何错误。而把文件 file2. c 提交给编译器，错误报告如下：

```
file2. c: 2: error: conflicting types for 'func'
file2. c: 1: error: previous declaration of 'func'
```

试分析原因。（在这两个文件中，第 1 行都是函数 func 的原型，第 2 行都是函数 func 的定义，函数体为空）

file1. c

```
int func(double);
int func(f) float f; {}
```

file2. c

```
int func(double);
int func(float f) {}
```

答 文件 file1. c 中，函数 func 的定义采用传统的方式，形式参数 f 的类型被提升到 double。函数原型中该参数也声明成 double 类型，两者类型相同，因此没有错误。

而在文件 file2. c 中，函数 func 的定义采用现在提倡的形式，形式参数的类型就是 float，不会提升。它的类型和函数原型中声明的类型不相同，所以编译器会报告错误。

11.8 C 的一个源文件可以包含若干个函数，该源文件经编译可以生成一个目标文件；若干个目标文件可以构成一个函数库。如果一个用户程序引用库中的某个函数，那么，在连接装配时的做法是下面三种情况的哪一种？说明你的理由。

1. 将该库函数的目标代码连接到用户程序。
2. 将该库函数的目标代码所在的目标文件连接到用户程序。

3. 将该函数库全部连接到用户程序。

答 是第二种情况,将该库函数的目标代码所在的目标文件连接到用户程序。

分析 第一种方式似乎最合理,但是不要忘记,C 语言是以文件为编译单位的,生成的目标文件对应于源文件。一个源文件可能有外部变量、静态外部变量,可能有若干个函数,因此要想把其中一个函数的目标代码从对应的目标文件中分离出来是困难的,目标代码的格式也不是按这样的要求来设计的。

另一方面,一个函数库虽然可以由几个目标文件组成,但是它们只是简单地组成一个库(比方说,并没有解决这些目标文件之间的引用问题),因此从库中抽出一个目标文件并不困难。

11.9 cc 是 UNIX 系统上 C 语言编译命令,-l 是连接库函数的选择项。某程序员自己编写了两个函数库 libuser1. a 和 libuser2. a,当用命令

```
cc  test. c  -luser1. a  -luser2. a
```

编译时,报告有未定义的符号,而改用命令

```
cc  test. c  -luser2. a  -luser1. a
```

时,能得到可执行程序。试分析原因。(备注:库名中的 lib 在命令中省略。上面第一个命令和命令 cc test. c libuser1. a libuser2. a 的效果是一致的)

答 这是因为 test. c 的代码引用了在 libuser2. a 库中定义的函数或外部变量,而 libuser2. a 库中的代码引用了在 libuser1. a 库中定义的函数或外部变量。例如,可能是如下的情况:

test. c	libuser1. a	libuser2. a
引用 a	定义 b	定义 a
		引用 b

分析 在进行连接时,对于命令行指出的函数库,连接程序依次进行扫描。若在库中找到一个定义,它定义了某个尚未确定入口的引用,那么就从库中将该定义所在的目标文件连接到目标程序中。连接程序对函数库仅扫描一次,不会为了某个尚未确定入口的引用而反复扫描函数库。这样,函数库的次序有时是紧要的。

对于上面第一个命令,在扫描 libuser1. a 时,并不知道它的某个定义被 libuser2. a 的程序引用,因而没有连接到目标程序中,所以出现未定义的符号。

11.10 cc 是 UNIX 系统上 C 语言编译命令,-l 是连接库函数的选择项。两个程序员分别编写了函数库 libuser1. a 和 libuser2. a。当用命令

```
cc  test. c  -luser1. a  -luser2. a
```

编译时,报告有重复定义的符号。而改用命令

```
cc  test. c  -luser2. a  -luser1. a
```

时,能得到可执行程序。试分析原因。

答 简单地说,出现了下面这样的情况:

test. c	libuser1. a	libuser2. a
引用 a	定义 a	定义 b
引用 b		定义 a

并且在 libuser2. a 中,a 和 b 定义在同一个目标文件中。

分析 由于 test. c 引用 a 和 b,用第二种次序连接时,a 和 b 的定义在 libuser2. a 都能找到,所以不会再把 libuser1. a 中含 a 定义的目标文件连接进来。而用第一种次序连接时,先把 libuser1. a 中含 a 定义的目标文件连接进来,然后还需要把 libuser2. a 中含 b 定义的目标文件连接进来,引起把 libuser2. a 中 a 的定义也要连接进来,因为在 libuser2. a 中 a 和 b 的定义在同一个目标文件中。由此造成了连接进来 a 的重复定义。

这种情况在若干人一起为一个软件编程时可能发生。两个人使用各自的函数库,在单体测试时没有发现库函数重名,在连接成完整的程序时出现上面的情况。宁可让连接器发现这种重复定义,这样可以及时发现问题,而不希望通过测试去发现因为怕重名而少连接进来一个函数。

11.11 修改习题 11.2 中的程序,将 main 函数中的 gcdx 改为 gcd,并将修改后的程序保存在 gcd. c 中。对该程序采用以下两种方式进行编译、连接:

```
gcc -o gcd1 gcd. c
gcc -static -o gcd2 gcd. c
```

所产生的可执行目标文件 gcd1 和 gcd2 的大小并不相同,前者约 11 KB,后者却要接近 1 MB。请分析产生这种不同的原因,它们在执行时存在什么样的差异。

答 后一种情况(加 static 选择项)要求连接器建立一个完全连接的可执行目标文件,该目标文件可以直接装入内存运行,不需要任何进一步的连接。

前一种情况没有对库函数进行静态连接,即执行这个命令时,只是做了一部分连接准备工作,真正的连接要等到运行时进行,所以形成的目标文件较小。在这种方式下,程序运行时还要进行动态连接,因此运行时的开销大一些。

11.12 a 和 b 表示当前目录中的目标模块或静态库,a→ b 表示 a 依赖于 b,即 b 定义了被 a 引用的符号。对于以下各种情况,给出最小的命令行(即包含最少数目的目标文件和库参数),从而使静态连接器能解析所有的符号引用。

(a) p. o→ libx. a

(b) p. o→ libx. a→ liby. a

(c) p. o→ libx. a→ liby. a 并且 liby. a→ libx. a→ p. o

答 (a) gcc p. o -lx. a

(b) gcc p. o -lx. a -ly. a

(c) gcc p. o -lx. a -ly. a -lx. a

11.13 两个 C 语言文件 link1.c 和 link2.c 的内容分别如下：

```
int buf[1] ={100};
```

和

```
extern int *buf;
main(){
    printf("%d\n", *buf);
}
```

在 x86/Linux 系统上，经命令 cc link1.c link2.c 编译后，运行时产生如下的出错信息：

```
Segmentation fault (core dumped)
```

请说明原因。

答 buf 定义在文件 link1.c 中，使用在文件 link2.c 中。由于：

•目标文件连接时并不检查名字的类型，因此，虽然两个文件对 buf 的类型持不同的观点，但仍然能连接成目标程序；

•目标文件连接时，是让不同文件中的同一名字的地址相同。

因此，运行时，由于 buf 的内容是 100，取 *buf 的值就是取地址 100 的内容。该地址不在用户数据区内，因此报错。

分析 若把这两个文件的内容合并到一个文件中，则编译时肯定会报告类型错误。

11.14 两个 C 文件 long.c 和 short.c 的内容分别是

```
long i = 32768 *2;
```

和

```
extern short i;
main() { printf("%d\n", i); }
```

在 x86/Linux 系统上，用 cc long.c short.c 命令编译这两个文件，能否得到可执行目标程序？若能得到目标程序，运行时是否会报错？若不报错，运行结果输出的值是否为 65 536？若不等于 65 536，原因是什么？

答 编译是以文件为单位的，不会发现文件之间的类型错误。由于数据的类型信息没有附加在目标文件中，因此连接时也不会发现两个文件中变量 i 的类型不一致。

运行时不会报错，但输出结果不是 65 536，而是 0。这是由于 x86 机器对于整型数据来说，是低地址放整数的低位，高地址放整数的高位。在 short.c 文件中，short i 取的是 long.c 文件中 long i 的那两个低位字节，它们都是 0，因此输出结果是 0。

11.15 下面左右两边分别是两个 C 程序文件 file1.c 和 file2.c 的内容，用命令 cc file1.c file2.c 对这两个文件进行编译和连接。请回答：

(a) 编译器是否会报错？若你认为会，则说明理由。

(b) 若编译器不报错,连接器是否会报错? 若你认为会,则说明理由。

(c) 若上面两步都不报错,则运行时是否会报错? 若你认为会,则说明理由。

(d) 若上面三步都不报错,则运行输出的结果是什么? 说明理由。

```
char k = 2;
char j = 1;
```

```
#include <stdio.h>
extern short k;
main() {
    printf("%d\n", k);
}
```

答 (a) 编译器不会报错,因为两个文件是分别编译的,因而不会发现其中的类型错误。

(b) 连接器不会报错,因为可重定位代码中没有变量的类型信息。

(c) 运行时也不会报错。

(d) 若机器的特点是低地址放整数的低位,高地址放整数的高位,如 x86,则结果是 258,因为分配给变量 j 的字节正好作为变量 short k 的高位字节,否则结果是 513。

11.16 下面左右两边是一个 C 程序的两个源文件的内容:

```
#include <stdio.h>
char *p = "0123456789";
f() {
    printf("%s\n", p);
}
```

```
#include <stdio.h>
char p[10];
main() {
f(); p[1] = '1';f();
}
```

运行目标程序所得到的结果如下(编译器是 GCC:(GNU) 4.2.3 (Debian 4.2.3-5)):

```
0123456789
Segmentation fault
```

请描述该目标程序各数据区的内容,并回答为什么运行时会出现 Segmentation fault 错误。

答 由于 C 程序以文件为编译单位,因此编译器不会发现 p 在两个文件中的类型不一致。由于左边文件给出 p 的强符号定义,因此 p 的存储分配和初值以左边文件为准。该程序在只读数据区(rodata)和静态数据区(data)的内容如下(略去了无关的部分):

```
    .rodata
.LC0:.string    "0123456789"
    ...
    .data
.p  .long       .LC0
```

在第二次调用 f 前,由于对 p[1]的赋值改变了 p(按字符指针类型的观点)的值,导致它指向的地址不在程序数据区的范围内,引起 Segmentation fault 错误。

11.17 下面三列是三个 C 程序(其中第三个程序由两个文件构成)。它们在编译、连接或运行时是否会报错？若是,请指出编译器、连接器或运行时系统报告错误的原因。对于不报错的程序,除了要说明输出结果外,还要对比有错的程序,说明为什么没有错误。

```
#include <stdio.h>
void f(m) long m[2];{
    printf("%d, %d\n",
        m[0], m[1]);
}
typedef struct {
    long n[2];
} stype;
main() {
    stype k;
    k.n[0]=0; k.n[1]=1;
    f(k);
}
```

```
#include <stdio.h>
void f(m) long m[2]; {
    printf("%d, %d\n",
        m[0], m[1]);
}
main() {
    long m[2];
    m[0]=0; m[1]=1;
    f(m);
}
```

文件 link1.c:

```
long k[2]={0,1};
```

文件 link2.c:

```
#include <stdio.h>
typedef struct {
    long n[2];
} stype;
extern stype k;
main() {
    printf("%d, %d\n",
      k.n[0], k.n[1]);
}
```

答 三个程序在编译时和连接时都不会报错,虽然看上去有类型方面的问题。

第一个程序在运行时报错,因为 main 传给 f 的是两个长整型的值,而 f 认为参数是数组的地址。

第二个程序不会报错是显然的,因为 main 传给 f 的是数组的地址,而 f 也认为参数是数组的地址,两者一致。输出是 0,1。

第三个程序也不会报错,两个文件都认为 k 是整型数据的地址。link1.c 文件对应的可重定位目标模块从这个地址开始安排了两个长整型数据,link2.c 文件对应的可重定位目标模块从这个地址开始使用两个长整数,因此没有问题。输出是 0,1。

11.18 如果在 A 机器上有 C 语言编译器 CC_A,也有它的源码 S_A(用 C 语言写成)。如何利用它通过尽量少的工作来得到 B 机器的 C 语言编译器 CC_B?

答 工作步骤可以如下。

1. 修改源码 S_A的代码生成部分,让它产生 B 机器的代码,称修改后的结果程序为 S_B。那么,S_B是一个用 C 写的编译器,它将 C 语言程序编译成 B 机器的代码。

2. 将 S_B提交给 CC_A进行编译,得到一个在机器 A 上可执行的程序。该程序是 A 机器上可运行的一个编译器,它将 C 语言程序编译成 B 机器的代码。

3. 将 S_B提交给上述可执行程序进行编译,得到所需的编译器 CC_B。因为所得程序在 B 机器上可运行,它将 C 语言程序编译成 B 机器的代码,因此就是所要的 CC_B。

11.19 什么是静态连接库与动态连接库？它们各有什么优缺点？

答 静态连接库和动态连接库是两种由用户自己开发的、可以加入到最终用户（包括用户本人和其他使用该模块的人）应用程序中提供某一特定功能的函数和类的集合，也称为用户模块。使用用户模块，可以做到：

1. 使用者无须了解模块的内部实现；
2. 模块的源代码对使用者是透明的；
3. 避免模块的函数名、变量名与最终用户的程序发生冲突。

静态连接库提供了函数完整的目标代码，如果程序调用某静态连接库中的函数，则在进行连接时，连接程序将该静态连接库中的有关代码复制至运行文件中（见 11.8 题）。

动态连接库也包含了其所提供的函数的目标代码，但是在程序连接动态连接库中的函数时，连接程序并不将包含在动态连接库中的函数的目标代码复制至运行文件，而只是简单地记录函数的位置信息（即包含于哪个动态连接库中以及在动态连接库中的位置）。有了这些信息后，程序在执行时，即可找到该函数的目标代码。因为只是在执行时才真正地连接，因此称为动态连接。提供函数在动态连接库中位置的信息存放在一个独立的文件中，这个文件就是引入库。

由于静态连接库将目标代码连接到应用程序中，当程序运行时，如果两个程序调用了同一个静态库中的函数，内存中将出现该函数的多个副本。而动态连接库则更适合于多任务环境：当两个应用程序调用了同一动态连接库中的同一个函数时，内存中只保留该函数的一个副本，这样内存利用率更高。

利用动态连接库可以实现资源共享，像 Windows 下的串行口、并行口驱动程序都是动态连接库；另外，Windows 下的字体也是动态连接库。

静态连接库将目标代码连入应用程序中，因此使用静态连接库的应用程序可独立运行。而使用动态连接库时，随同应用程序还要提供动态连接库文件。

第 12 章　面向对象语言的编译

一、主要知识点

主教材第 12 章通过介绍 C++到 C 的翻译，说明面向对象语言编译的一些特征。

信息封装是面向对象语言的一个重要概念，但这是一些作用域规则，从实现的角度看比较容易，因此教材上未作过多介绍。

另一个重要的概念是继承性。继承性涉及对象的存储空间组织，它是本章的重点，其中重复继承是难点。

二、习题精选与解析

12.1　和主教材 12.2 节相比，下面的代码给出了图形对象类库的另一片断。根据教材 12.3.1 节的编译方案，确定该程序片断中定义的各个类的方法表和方法索引。

```
class ClosedGraphics : public GraphicalObj {
  public :
    // Surface area enclosed by object
    virtual double area (void);
};

class Ellipse : public ClosedGraphics {
    Point    _center;                    // Centre of the ellipse
    double   _x_radius, _y_radius;       // Radii of the ellipse
    double   _angle;                     // Rotation from x-axis

public:
  // Constructor
    Ellipse (Point &center, double x_radius, double y_radius, double angle = 0) {
        _center = center;
        _x_radius = x_radius;
        _y_radius = y_radius;
```

```
        _angle = angle;
}

    // Ellipse area --'ClosedGraphics :: area'
    double area (void) {return PI * _x_radius * _y_radius;}

    // Distance to a point --expensive!
    virtual double dist (Point &);

    // Center
    const Point & center (void) {return   _center;}

    // Translate --overwrites 'GraphicalObj :: translate'
    void translate (double x_offset, double y_offset) {
        _center. translate(x_offset, y_offset);
    }

    // Scale --overwrites 'GraphicalObj :: scale'
    void scale (double scale_factor) {
        _x_radius * = scale_factor;
        _y_radius * = scale_factor;
    }

    //...
};

class Circle : public Ellipse {
  public:
    // Constructor
    Circle (Point &center, double radius) {
        Ellipse (center, radius, radius);
    }

    // Distance to a point --- overwrites 'Ellipse :: dist'
    virtual double dist (Point &p) {
```

```
        double center_dist = _center.dist(p);
        if (center_dist <= radius) return 0;
        else return center_dist -radius;
    }
    //…
};
```

答 该程序片断中定义的各个类的方法表和方法索引见图 12.1。

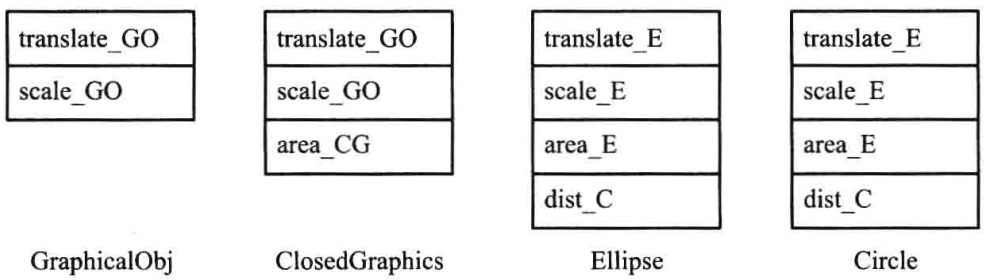


图 12.1 图形对象的不同子类的方法表

12.2 根据主教材上介绍的方法,C++中的对象声明语句应如何翻译成 C 语句? 例如图 11.1程序中的

```
Point _center;
```

应怎样翻译?

答 C++语言中类的对象声明不加翻译就成了 C 语言中相应结构体类型的变量声明,不管对象声明出现在程序中的什么地方。

分析 由于 C++语言中一个类的所有非静态属性构成 C 语言的一个结构体类型,并且类的名字就作为结构体类型的名字,因此上面的语句不做任何修改,_center 自然就成了 Point 结构体类型的变量。所要注意的是,还应该根据 Point 类中构造函数的参数置初值的情况,来给_center 适当地静态置初值。

12.3 按照主教材上介绍的方法,把下面 C++语言的函数翻译成 C 的函数。

```
void zoom (GraphicalObj &obj, double zoom_factor, Point &center) {
    obj.translate (-center.x, -center.y);        // 将中心点移至原点(0, 0)
    obj.scale (zoom_factor);                     // 缩放
}
```

答 按照主教材上介绍的方法并参考主教材上图 12.3,该函数的翻译结果如下:

```
void zoom (GraphicalObj &obj, double zoom_factor, Point &center) {
    obj.vptr[0] (obj,-center.x, -center.y); // 将中心点移至原点(0, 0)
    obj.vptr[1] (obj, zoom_factor);              // 缩放
}
```

12.4 根据独立的重复继承的编译方案，确定 12.1 题中 Circle 的方法表和方法索引，并且编译形式为 c.dist(p)的方法调用，其中 c 和 p 分别是类 Circle 和 Point 的对象。

答 在独立的重复继承编译方案下，Circle 的方法表和方法索引见图 12.2。方法调用 c.dist(p)被编译成

```
c.vptr[3].faddr(&c + c.vptr[3].offset, p)
```

faddr	offset
translate_E	0
scale_E	0
area_E	0
dist_C	0

图 12.2 Circle 的方法

分析 本例中类的继承关系是

GraphicalObj →ClosedGraphics →Ellipse →Circle

并不出现重复继承。因此，虽然是考虑独立的重复继承编译方案，但问题仍然简单，不会出现像主教材上图 12.9 那样的情况。不过，由于采用的是独立的重复继承编译方案，因此方法表中的每个方法有两个域，其中 faddr 是虚函数地址，offset 是调整视图的偏移。在这里，由于不用调整，因此 offset 都等于 0。

12.5 主教材上 12.2 节在介绍类的翻译时有这么一句话："将 C++语言中一个类的所有非静态属性构成一个 C 语言的结构体类型，取类的名字作为结构体类型的名字"。在学完本章后，你认为这句话需要修改吗？

答 需要修改，增加虚方法表指针作为第一个域。

12.6 在面向对象语言中，编译器在给每个对象分配空间的第一个域中存放虚方法表的指针。是否可以把虚方法表指针作为最后一个域而不是第一个域？请简要说明理由。

答 不行。按主教材上目前的分配方式，若作为最后一个域，则难以为该对象产生其超类的视图。

12.7 在面向对象语言中，编译器为每个类建立虚方法表，如主教材上图 12.3 和图 12.4 那样。请简要说明，为什么编译器给每个类仅建立虚方法表，而不是建立所有方法的方法表。

答 虚方法表是用来实现方法的动态绑定的。类中的虚方法可以被派生类重写，导致执行方式被派生类改变，因而出现动态绑定问题。而对于非虚方法，无论被其所在类的实例调用，还是被这个类的派生类的实例调用，方法的执行方式都不变，因而没有动态绑定问题。

12.8 主教材上 12.2 节的结尾介绍，类 Point 的方法 translate 翻译成函数 translate__5Pointdd：

```
void translate__5Pointdd(Point &this, double x_offset , double y_offset) {
    this.xc += x_offset;   this.yc += y _offset;
}
```

其中假设 C 语言也有引用调用方式(形式参数前面加字符&)。请按 C 语言实际只有值调用方式来考虑，写出方法 translate 翻译成的 C 函数，并说明调用点需要做的修改。

答 方法 translate 翻译成的 C 函数如下：

```
void translate__5Pointdd(Point *this, double x_offset , double y_offset) {
    this->xc += x_offset;   this->yc += y _offset;
}
```

在调用点，第一个实在参数前面应该加上取地址记号&。

12.9 请按主教材上图 12.4 的方式给出 12.2 节中类 Point 的对象表示。

答 见图 12.3。

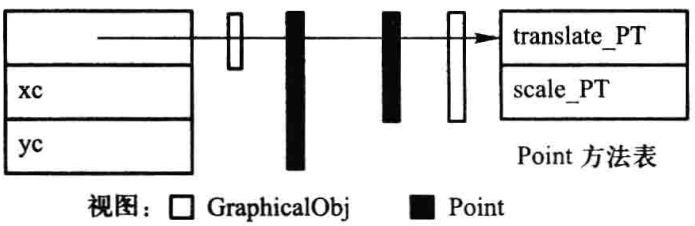


图 12.3 类 Point 的对象表示

第 13 章　函数式语言的编译

一、主要知识点

主教材第 13 章介绍了一种简单的函数式语言 SFP 以及 SFP 到一种抽象机语言 FAM 的编译方案。通过本章的学习可以了解函数式语言所涉及的 λ 演算的有关概念，还有它的抽象机的特点，特别是抽象机的堆的作用。从抽象机的堆所管理的对象可以知道，什么样对象的生存期同栈式管理的对象的生存期是不一样的，这是前面学习命令式语言式没有碰到过的。

由于国内学习和使用函数式编程语言的高校和研究机构很少，了解函数式语言的实现的人就更少。因此本章的习题解答给出几个简单的题目，仅用来帮助熟悉函数式语言的实现中那些有特点的内容。

二、习题精选与解析

13.1　下面是简单函数式编程语言 SFP 的一些表达式，确定它们的自由变量的集合和约束变量的集合：

(a) $(\lambda x.\ xy)\ (\lambda y.\ y)$

(b) $\lambda xy.\ z\ (\lambda z.\ z\ (\lambda x.\ y))$

(c) $(\lambda xy.\ xz\ (yz))\ (\lambda x.\ y(\lambda y.\ y))$

(d) $\lambda x.\ x$ + **letrec** $a == x$;

$x == f\ y$;

$y == z$

in $x + y + z$

答　(a) 自由变量集合：$\{y\}$，约束变量集合：$\{x, y\}$。

(b) 自由变量集合：$\{z\}$，约束变量集合：$\{y, z\}$。

(c) 自由变量集合：$\{y, z\}$，约束变量集合：$\{x, y\}$。

(d) 自由变量集合：$\{f, z\}$，约束变量集合：$\{x, y\}$。

13.2 把下列 SFP 表达式编译成 FAM 抽象机指令序列。

(a) $(\lambda x.\ x + 1)\ 3$

(b) **letrec** $F == \lambda xy\ .\ xy$;

$inc == \lambda x.\ x + 1$

in $F\ inc\ 5$

答 (a) 编译该表达式生成的指令序列如下:

	mark l_1;	V_code $((\lambda x.\ x + 1)\ 3)\ [\ ]\ 0$
	ldb 3;	C_code 3 [ ] 3
	mkbasic;	
	mkvec 0;	V_code$(\lambda x.\ x + 1)$ [ ] 4
	mkvec 0;	
	ldl l_2;	
	mkfunval;	
	ujmp l_3;	
l_2:	targ 1;	
	pushloc 1;	V_code$(x + 1)\ \{x \mapsto \langle \text{LOC}, -1 \rangle\}\ 0$
	getbasic;	
	ldb 1;	B_code $1\{x \mapsto \langle \text{LOC}, -1 \rangle\} 1$
	op_{bin};	
	mkbasic;	
	return 1;	
l_3:	apply;	
l_1:	stop	

(b) 编译该表达式生成的指令序列如下:

	alloc;	V_code (letrec ⋯ in ⋯) [ ] 0
	alloc;	
	mkvec 0;	C_code $(\lambda xy\ .\ xy)\ \{F \mapsto \langle \text{LOC}, 0 \rangle, inc \mapsto \langle \text{LOC}, 1 \rangle\}\ 2$
	mkvec 0;	
	ldll_1;	
	mkfunval;	
	ujmp l_2;	
l_1:	targ 2;	
	mark l_3;	V_code $(xy)\ \{x \mapsto \langle \text{LOC}, -1 \rangle, y \mapsto \langle \text{LOC}, -2 \rangle\}\ 0$
	pushloc 5;	C_code $y\{x \mapsto \langle \text{LOC}, -1 \rangle, y \mapsto \langle \text{LOC}, -2 \rangle\}\ 3$
	pushloc 5;	V_code $x\{x \mapsto \langle \text{LOC}, -1 \rangle, y \mapsto \langle \text{LOC}, -2 \rangle\}\ 4$

```
        eval
        apply;
l3:     return 2;
l2:     rewrite 2;
        mkvec 0;          C_code (λx. x + 1) {F ↦⟨LOC, 0⟩, inc ↦⟨LOC, 1⟩} 2
        mkvec 0;
        ldl l4;
        mkfunval;
        ujmp l5;
l4:     targ 1;
        pushloc 1;        V_code (x + 1) {x ↦⟨LOC, -1⟩} 0
        getbasic;
        ldb 1;            B_code 1{x ↦ ⟨LOC, -1⟩}1
        op_bin;
        mkbasic;
        return 1;
l5:     rewrite 1;
        mark l6;          V_code (Finc 5) {F ↦⟨LOC, 0⟩, inc ↦⟨LOC, 1⟩} 2
        ldb 5;            C_code 5{F ↦⟨LOC, 0⟩, inc ↦⟨LOC, 1⟩} 5
        pushloc 5;        C_code inc ↦{F ↦⟨LOC, 0⟩, inc ⟨LOC, 1⟩} 6
        pushloc 7;        V_code F{F ↦⟨LOC, 0⟩, inc ↦⟨LOC, 1⟩} 7
        eval;
        apply;
l6:     slide 2;
        stop
```

13.3 把表达式(λx.(λyz.(x+y)+z) 3) 4 5 翻译成 FAM 代码。

答 翻译结果如下：

```
        mark l1;          V_code((λx.( λyz.(x+y)+z) 3) 4 5) [ ] 0
        ldb 5;            C_code 5 [ ] 3
        mkbasic;
        ldb 4;            C_code 4 [ ] 4
        mkbasic;
        mkvec 0;          V_code (λx.( λyz.(x+y)+z) 3) [ ] 5
        mkvec 0;
```

```
        ldl l2;
        mkfunval;
        ujmp l3;
l2:     targ 1;
        mark l4;        V_code ((λyz.(x+y)+z) 3) {x ↦⟨LOC, -1⟩} 0
        ldb 3;          C_code 3 [ ] 3
        mkbasic;
        pushloc 5;      V_code (λyz.(x+y)+z) {x ↦⟨LOC, -1⟩} 4
        mkvec 1;
        mkvec 0;
        ldl l5;
        mkfunval;
        ujmp l6;
l5:     targ 2;
        pushglob 1;     V_code ((x+y)+z) {y ↦⟨LOC, -1⟩, z ↦⟨LOC, -2⟩, x ↦⟨GLOB,1⟩} 0
        eval;
        getbasic;
        pushloc 2;      B_code y{y ↦⟨LOC, -1⟩, z ↦⟨LOC, -2⟩, x ↦⟨GLOB, 1⟩} 1
        eval;
        getbasic;
        op_bin;
        pushloc 3;      B_code z{ y ↦⟨LOC, -1⟩, z ↦⟨LOC, -2⟩, x ↦⟨GLOB, 1⟩} 1
        eval;
        getbasic;
        op_bin;
        mkbasic;
        return 2;
l6:     apply;
l4:     return 1;
l3:     apply;
l1:     stop
```

13.4 表达式(λx.(λyz.(x + y) + z) 3) 4 5 和(λx.(λyz.(x + y) + z) 3 5) 4 有同样的结果。在抽象机 FAM 上,哪一个表达式对应的目标代码的执行效率高? 为什么?

答 第一个表达式在执行(λyz.(x+y)+z)3 时,出现了参数个数不足的情况,因此会出现这

样的情况:把 FUNVAL 对象压入栈顶,然后发现参数个数不足,再把它改成 FANVAL 对象并释放栈帧。

第二个表达式执行的是($\lambda yz.(x + y) + z$) 35,不会出现参数个数不足的情况,从而不会出现上面这样的反复。因此第二个表达式的执行效率比第一个表达式高。

郑重声明

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

反盗版举报电话　(010) 58581897　58582371　58581879
反盗版举报传真　(010) 82086060
反盗版举报邮箱　dd@ hep.com.cn
通信地址　北京市西城区德外大街 4 号　高等教育出版社法务部
邮政编码　100120

Document generated by Anna’s Archive around 2023-2024 as part of the DuXiu collection (https://annas-blog.org/duxiu-exclusive.html).

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):

{
  "filename": "MTM2NDU4NTEuemlw",
  "filename_decoded": "13645851.zip",
  "filesize": 30675053,
  "md5": "f98d453b23f0f70cc6be6fd888bf07d1",
  "header_md5": "6610afb0aec6431798cd514433bf4152",
  "sha1": "51b0063b1f0f39b9d279930c37f593b096f8341c",
  "sha256": "a4ed630a0454dfb6d917635ffbb7f911b3fb796699e505d2575a115bb106b968",
  "crc32": 2950194448,
  "zip_password": "",
  "uncompressed_size": 39256538,
  "pdg_dir_name": "",
  "pdg_main_pages_found": 201,
  "pdg_main_pages_max": 201,
  "total_pages": 207,
  "total_pixels": 1116328424,
  "pdf_generation_missing_pages": false
}