普通高等教育“十五”国家级规划教材

编 译 原 理

陈意云　张　昱

高 等 教 育 出 版 社

内 容 简 介

本书介绍编译器构造的一般原理和基本实现方法,主要内容包括词法分析、语法分析、语义分析、中间代码生成、代码优化和目标代码生成等。除了介绍命令式编程语言的编译技术外,本书还介绍面向对象语言和函数式编程语言的实现技术。本书还强调一些相关的理论知识,如形式语言和自动机理论、语法制导的定义和属性文法、类型论和类型系统等。

本书取材广泛新颖、图文并茂,注意理论联系实际。本书可作为高等学校计算机科学及相关专业的教材,也可供计算机软件工程技术人员参考使用。

前　言

本书介绍编译器构造的一般原理、基本设计方法和主要实现技术，可作为高等学校计算机科学及相关专业的教材。

虽然只有少数人从事构造或维护程序设计语言编译器的工作，但是编译原理和技术对高校学生和计算机软件工程技术人员来说仍是重要的基础知识之一。本书能使读者对程序设计语言的设计和实现有深刻的理解，对和程序设计语言有关的理论有所了解，对宏观上把握程序设计语言来说，能起一个奠基的作用。本书的学习还有助于读者快速理解、定位和解决在程序调试与运行中出现的问题。

对软件工程来说，编译器是一个很好的实例（基本设计、模块划分、基于事件驱动的编程等），本书所介绍的概念和技术能应用到一般的软件设计之中。

大多数程序员同时也是语言的设计者，虽然只是一些简单语言（如输入输出、脚本语言）的设计者，但学习本书仍有助于提高他们设计这些语言的水平。

编译技术在软件安全、程序理解和软件逆向工程等方面有着广泛的应用。

作为教材，本书有如下一些特点：

⑴ 在介绍语言实现技术的同时，强调一些相关的理论知识，如形式语言和自动机理论、语法制导的定义和属性文法、类型论和类型系统等。它们是计算机专业理论知识的一个重要部分，在本书中结合应用来介绍这些知识，有助于学生较快领会和掌握。

⑵ 在介绍编译器各逻辑阶段的实现时，强调形式化描述技术，并以语法制导定义作为翻译的主要描述工具。

⑶ 强调对编译原理和技术的宏观理解及全局把握，而不把读者的注意力分散到一些枝节的算法上，如计算开始符号集合和后继符号集合的算法、回填技术等。出于同样的目的，本书较详细地介绍了编译系统和运行系统。

⑷ 本书还介绍了面向对象语言和函数式语言的实现技术，有助于加深读者对语言实现技术的理解。书中带星号的章节，作为教学的可选部分。

⑸ 作为原理性教材，本书介绍基本的理论和方法，而不偏向于某种源语言或目标机器。

⑹ 我们鼓励读者用所学的知识去分析和解决实际问题，因此本书中有很多习题是从实际碰到的问题中抽象出来的。这些习题也能激发读者学习编译原理和技术的积极性。

⑺ 为了便于读者学习，本书配有习题解答（见参考文献[8]）。

本书的多数章节是参考了参考文献1和参考文献7编写的,部分习题取自参考文献8。在此向有关作者表示感谢。

本书第1章到第6章以及第12章主要由陈意云编写,第7章到第11章主要由张昱编写。作者的学生陈晖准备了10.2节的初稿,李筱青准备了10.3节的初稿,吴萍和项森也为第10章的编写做了很多技术工作。

中国科学院软件研究所研究员程虎先生审阅了全书,并提出了许多宝贵的意见,在此表示衷心的感谢。

由于作者水平有限,书中难免还存在一些缺点和错误,恳请广大读者批评指正。

作　者

于中国科学技术大学

2003年5月

目　录

第 1 章　编译器概述

从理论上说,构造专用计算机来直接执行某种高级语言写的程序是可能的。但是,实际上目前的计算机能执行的都是非常低级的机器语言。那么,一个基本的问题是:高级语言的程序最终是怎样在计算机上执行的。

能够完成从一种语言到另一种语言变换的软件称为翻译器,这两种语言分别叫做该翻译器的源语言和目标语言。编译器是一种翻译器,它的特点是目标语言比源语言低级。

本章通过描述编译器的各个组成部分来介绍编译这个课题。该课题涉及程序设计语言、机器结构、形式语言理论、类型论、算法和软件工程等方面的知识。

编译器的工作可以分成若干阶段,每个阶段把源程序从一种表示变换成另一种表示。编译过程的一种典型分解见图 1.1,图中的每个方框表示它的一个阶段。

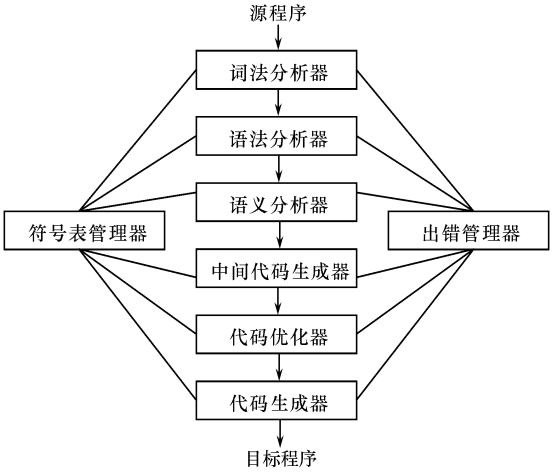


图 1.1　编译的阶段

本节以 Pascal 语言的赋值语句

$$position := initial + rate * 60 \tag{1.1}$$

的翻译(假定变量都是实型)为例,概要介绍编译的各个阶段。

1.1 词法分析

词法分析逐个读构成源程序的字符,把它们组成词法记号(*token*)流。赋值语句(1.1)的字符流在词法分析时被组成下面的词法记号流:

(1) 标识符(position)
(2) 赋值号(:=)
(3) 标识符(initial)
(4) 加号(+)
(5) 标识符(rate)
(6) 乘号(*)
(7) 数(60)

分隔记号的空格通常在词法分析时被删去。

词法单元 position、initial 和 rate 属于同样的记号,因此需要为某些记号增加一个属性来区分属于同一记号的不同词法单元。例如,发现 rate 这样的标识符时,词法分析器不仅产生一个记号,如 id,还把当前词法单元 rate 填入符号表,如果表中还没有它的话。id 这次出现的属性是符号表中 rate 条目的指针。

用 id_1、id_2 和 id_3 分别表示 position、initial 和 rate,以强调标识符的内部表示是有别于形成标识符的字符序列的。于是,语句(1.1)在词法分析后的表示是

$$id_1 := id_2 + id_3 * 60 \tag{1.2}$$

多字符算符 := 和数 60 也应该有它们的内部表示,本书将在第 2 章词法分析中讨论,为直观起见,这里直接用它们在源程序中的字符序列。

编译器的词法分析也叫做线性分析或扫描。

1.2 语法分析

语法分析(syntax analysis)简称为分析(parsing),它把词法记号流依照语言的语法结构按层次分组,以形成语法短语。因此语法分析也称为层次分析。源程序语法短语常用分析树表示,图 1.2 便是一例。

在表达式 initial + rate * 60 中,短语 rate * 60 是一个逻辑单位,因为按算术表达式的一般习惯,乘比加先完成;由于 initial + rate 后面是乘号,所以 initial + rate 不能组成一个短语。

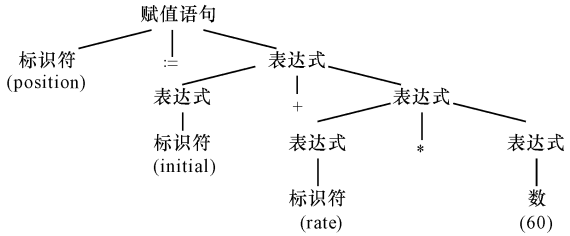


图1.2 position := initial+rate*60 的分析树

程序的层次结构通常由递归的规则表示,例如,可以用如下规则作为表达式定义的一部分:

(1) 任何一个标识符都是表达式;

(2) 任何一个数都是表达式;

(3) 如果 e_1 和 e_2 都是表达式,那么

$e_1 + e_2$

$e_1 * e_2$

(e_1)

也都是表达式。

规则 (1) 和 (2) 都是(非递归的)基本规则,而规则 (3) 是通过把算符作用于表达式来定义更复杂的表达式。这样,由规则 (1),initial 和 rate 都是表达式;由 (2),60 是表达式;由 (3),首先可推出 rate * 60 是表达式,然后 initial + rate * 60 也是表达式。

同样地,许多语言用类似如下的规则递归地定义语句:

(1) 如果 identifier 是标识符,expression 是表达式,那么

identifier := expression

是语句。

(2) 如果 expression 是表达式,statement 是语句,那么

while (expression) do statement

if (expression) then statement

也都是语句。

图1.2 的分析树描绘了 position := initial + rate * 60 的语法结构,这种语法结构更常见的内部表示由图1.3(a) 的语法树给出。语法树是分析树的浓缩表示,其中内部结点都是算符,内部结点的后代是它的运算对象。图1.3(b) 这样的树结构将在第4章4.2节讨论。在第4章的语法制导翻译中,将详细讨论编译器如何利用输入所含的层次结构来产生语法树。

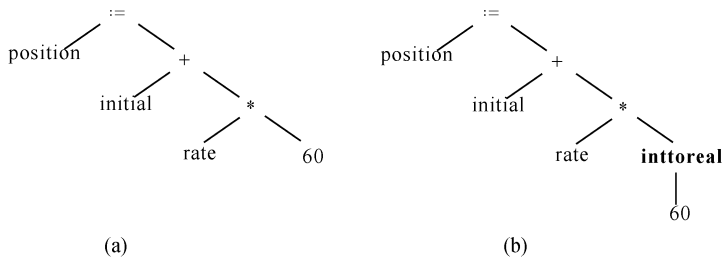


图1.3　语义分析插入了类型转换

1.3　语义分析

语义分析阶段检查程序的语义正确性,以保证程序各部分能有意义地结合在一起,并为以后的代码生成阶段收集类型信息。

语义分析的一个重要部分是类型检查,编译器检查每个算符的运算对象,看它们的类型是否适当。例如,当实数作为数组的下标时,许多语言的规定是要求编译器报告错误;当然也有些语言允许运算对象的类型隐式转换,例如二元的算术算符作用于一个整数和一个实数的时候。类型检查和语义分析在第5章讨论。

例1.1　在机器内部,整数的二进制表示和实数的二进制表示是有区别的,不论它们是否有相同的值。在图1.3中,所有的标识符都声明为实型,另外,由60本身可知它是整数。对图1.3(a)进行类型检查会发现*作用于实型变量rate和整数60,通常的办法是把整数转变为实数。可以建立一个额外的算符结点inttoreal(见图1.3(b)),它显式地把整数转变为实数。

编译器的前三个阶段对源程序分别进行不同的分析,以揭示源程序的基本数据和结构,决定它们的含义,建立源程序的中间表示。许多处理源程序的软件工具都要完成某类分析,下面给出几个例子。

(1) 格式打印程序　它以源程序为输入,以程序结构清晰可见的方式输出源程序。例如,注释可以以特殊的字体或颜色出现,语句可以按它们嵌套的层次阶梯式地显示出来。显然,格式打印程序需要对源程序进行词法分析和语法分析,但不需要对源程序进行语义分析,因为一种书写格式只和语法有关。

(2) 文档抽取程序　它抽取程序文件中的注释和函数首部等信息,形成一份程序文档。它需要对程序进行词法分析和语法分析,至少是部分的语法分析。要使得这样抽取的信息

形成一份有用的文档,注释很有讲究,否则抽取出来的文档没有什么用处。

⑶ 静态检查程序 静态检查程序读入程序,分析它,并试图不运行程序而发现一些潜在的错误。它的分析部分和第9章优化编译器的分析部分类似。例如,它可以检查出源程序的某些部分决不会执行,或者某个变量在赋值前可能被引用。此外,使用比第五章更精致的类型检查和类型推导技术,它还能捕捉程序的安全隐患,例如通过指针使用已经释放了的存储空间。

⑷ 解释器 纯解释器在执行源程序语句时,都需分析构成该语句的字符串,以便识别和执行它指定的计算。如果给定的语句仅执行一次,纯解释的方法是所有方法中代价最小的一种,例如,它常用于交互语言的"立即命令"。如果语句重复执行,较好途径是分析源程序的字符流仅一次,用一串更适于解释的符号序列或其他的形式来代替它。因此解释器往往也做某种程度的翻译,例如,对于赋值语句,解释器可能建立像图1.3⑵那样的树,在遍历树时执行结点的操作。在树根,它发现必须完成赋值,因此调用子例程来计算右部表达式的值,然后把值存到为标识符position分配的存储单元。计算树根右子树的子例程发现必须计算两个表达式的和,它递归地调用自己来计算表达式rate*60的值,然后再加上变量initial的值。

1.4 中间代码生成

语法分析和语义分析后,某些编译器产生源程序的显式中间表示,可以认为这种中间表示是一种抽象机的程序。中间表示必须具有两个性质:它易于产生并且易于翻译成目标程序。

中间表示有各种形式。在第7章,我们把中间表示看成三地址代码,它们像机器的汇编语言,这种机器的每个存储单元的作用类似于寄存器。三地址代码由三地址语句序列组成,每个三地址语句最多有三个操作数,语句(1.1)的三地址代码可以如下:

$$\begin{aligned} &temp1 := inttoreal(60)\\ &temp2 := id3 * temp1\\ &temp3 := id2 + temp2\\ &id1 := temp3 \end{aligned} \qquad (1.3)$$

这种中间形式有它的特点。首先,除了赋值算符外,每个语句至多只有一个算符,因此,在生成这些语句时,编译器必须决定运算完成的次序,语句(1.1)的乘优先于加。其次,编译器必须产生临时变量名,用以保留每个语句的计算结果。第三,某些三地址语句没有三个运算对象,例如语句(1.3)的第一个和最后一个语句。

本书在第7章叙述编译器时用的主要是中间表示。通常,除了计算表达式外,这些中间

表示还必须做其他事情,它们必须处理控制流结构和过程调用。第 4 章和第 7 章提供为程序设计语言典型结构产生中间代码的一些算法。

1.5 代码优化

代码优化阶段试图改进代码,以产生执行较快的机器代码。如果中间代码生成算法比较简单的话,它给代码优化留了很多机会。例如,产生中间代码的一个比较自然的算法是为语义分析后的树上的每个算符产生一条指令,因而得到语句 (1.3) 的中间代码。然而,还存在更好的算法,例如使用两条指令

```
temp1 := id3 * 60.0
id1 := id2 + temp1                                    (1.4)
```

也可以完成同样的计算。用简单的中间代码生成算法是可以的,因为产生较优代码这个问题可以在代码优化阶段得以解决,也就是,优化器会推断出,60 从整型表示变为实型表示可以在编译时完成,从而 inttoreal 操作可以删去。此外,temp3 只使用一次,即把它的值传给 id1,所以用 id1 代替 temp3 是妥当的,从而 (1.3) 的最后一个语句不必存在,这样就得到 (1.4) 的结果。

不同的编译器完成不同程度的优化,能完成大多数优化的编译器叫做“优化编译器”,但是编译的相当大一部分时间都消耗在这种优化上。简单的优化也可以使目标程序的运行时间大大缩短,而编译速度并没有降低太多。第 9 章将讨论优化问题。

1.6 代码生成

编译的最后一个阶段是目标代码生成,它生成可重定位的机器代码或汇编码。此阶段为源程序所用的每个变量选择存储单元,并且把中间代码翻译成等价的机器指令序列。此阶段的一个关键问题是寄存器分配。

例如,使用寄存器 R1 和 R2,(1.4) 的代码可以翻译成:

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1                                          (1.5)
ADDF R2, R1
MOVF R1, id1
```

每条指令的第一个和第二个操作数分别代表源和目的操作数。每条指令的F告诉我们指令处理浮点数。这段代码把地址id3的内容取入寄存器R2(我们暂且认为指令中id3代表对象id3的地址。变量的存储分配在第6章讨论),然后把它乘上实数60.0,#号代表60.0作为常数处理。第三条指令把id2取入寄存器R1,再把原先寄存器R2的值加上去,最后寄存器R1的值存入地址id1。这样,该段代码便实现了图1.4的赋值。代码生成在第8章讨论。

中间代码生成、代码优化和代码生成三个阶段合称为对源程序进行综合,它们从源程序的中间表示建立起和源程序等价的目标程序。

1.7 符号表管理

符号表管理和出错管理是编译过程中的两项重要工作,它们与词法分析、语法分析、语义分析、中间代码生成、代码优化和代码生成这六个阶段相互作用。下面我们简单介绍这两项工作。

编译器的一项重要工作是记录源程序中使用的标识符,并收集每个标识符的各种属性。这些属性提供标识符的存储分配、类型和作用域信息。如果是过程标识符,还有参数的个数和类型,参数传递方式和返回值类型(如果有的话)。

符号表是为每个标识符保存一个记录的数据结构,记录的域是标识符的属性。该数据结构允许我们迅速地找到一个标识符的记录,在此记录中存储和读取数据。符号表在第7章讨论。

词法分析器发现源程序的标识符时,把该标识符填入符号表。但是,一般来说。词法分析期间不能确定一个标识符的属性。例如,像

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

这样的Pascal声明,词法分析器读过position、initial和rate时,它们的类型还是未知的。

其余的阶段把标识符的信息填入符号表,然后以不同的方式使用这些信息。例如,语义分析和中间代码生成需要知道标识符的类型,才能检查源程序是否以合法的方式使用它们,才能为它们产生适当的操作。代码生成需要使用标识符存储分配信息以产生正确的指令。

1.8 错误诊断和报告

每个阶段都有可能发现源程序的错误。在发现错误后,该阶段必须处理此错误,使得编译可以继续进行,以便进一步发现源程序的其他错误。发现一个错误便停下来的编译器是

没有尽到责任的,除非它是集成在一个好的编程环境中,使得编译和编辑之间的切换相当方便和迅速。

语法分析和语义分析通常处理编译器能发现的绝大部分错误。词法分析阶段能发现的错误类型是,当前被扫描的字符串不能形成语言的词法记号。记号流违反语言的语法规则是由语法分析阶段诊断。语义分析时,编译器试图找出语法正确但对所含的操作来说是无意义的结构,如相加的两个标识符,其一是数组名,另一个是过程名。本书把每个阶段对错误的处理放在与该阶段有关的章节介绍。

语句 (1.1) 的编译过程总结在图 1.4 中。

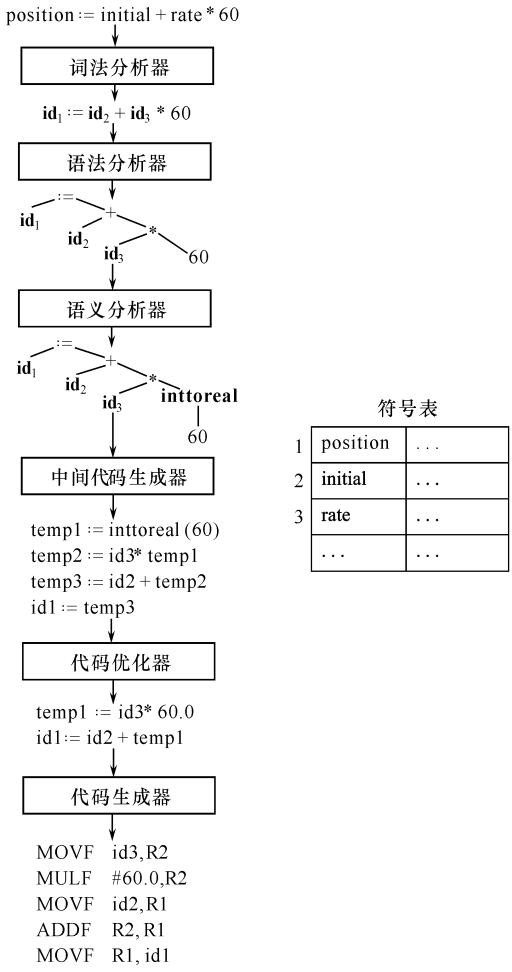


图 1.4　一个语句的翻译

1.9 阶段的分组

在实际的编译器中,若干阶段可以组合在一起,各阶段之间的中间表示也无需显式构造。

通常,所有的阶段被分成前端和后端两部分。前端只依赖于源语言,由几乎独立于目标机器的阶段或阶段的一部分组成。通常,它们是词法和语法分析、符号表建立、语义分析和中间代码生成,有些优化也可以在前端完成。前端还包括与这些阶段同时完成的错误处理。

后端是指编译器中依赖于目标机器的部分,它们一般独立于源语言,而与中间语言有关。后端包括代码优化、代码生成和伴随这些阶段的错误处理和符号表操作。

取一个编译器前端,重写它的后端以产生同一源语言在另一机器上的编译器已经是件普通的事情。如果这些后端是很仔细设计的,甚至不需要对它做很多的重新设计。

把几种不同的语言编译成同一种中间语言,让不同的前端使用同一后端,从而得到一台机器上的几个编译器,也是很吸引人的事。但是,由于不同语言的着眼点有区别,在这个方向上只取得了有限的成功。

编译的几个阶段常用一遍(pass)扫描来实现,一遍扫描包括读一个输入文件和写一个输出文件。在工程实践中,有很多不同的方式把编译器的阶段组成遍,因此我们更愿意围绕着阶段而不是围绕遍来组织对编译的讨论。

把几个阶段组成一遍,并且这些阶段的活动在该遍中交错进行是屡见不鲜的。例如,词法分析、语法分析、语义分析和中间代码生成可以组成一遍。如果这样,第一遍扫描可以直接从字符流翻译成中间代码。更具体一些,可以把语法分析器看成是“主导”的,它试图在所看见的记号流上找出语法结构。当它需要记号时,调用词法分析器取下一个记号。如果已看出一个语法结构,语法分析器则激活中间代码生成器,以完成语义分析和生成中间代码。

习 题 1

1.1 解释下列名词:

源语言 目标语言 翻译器 编译器 解释器

1.2 典型的编译器可以划分成哪几个主要的逻辑阶段,各阶段的主要功能是什么?

第2章　词 法 分 析

词法分析器的任务是把构成源程序的字符流翻译成词法记号流。构造词法分析器的一种简单办法是用状态转换图来描述源语言词法记号的结构,然后手工把这种状态转换图翻译成为识别词法记号的程序。用这种方式可以产生高效的词法分析器。

本章重点围绕词法分析器的自动生成展开,先介绍与之有关的正规式和有限自动机概念,以及词法分析器的自动生成方法,最后介绍一个词法分析器自动生成工具 Lex。

2.1　词法记号及属性

词法分析是编译的第一阶段,它的主要任务是读输入字符流,产生用于语法分析的词法记号序列。在第1章中提到,编译器一些阶段的活动会交错进行,图2.1给出了词法分析器和语法分析器的一种典型关系,即把词法分析器作为语法分析器的一个子程序来实现。当收到来自语法分析器的"取下一个记号"命令时,词法分析器读输入字符直到它能够确认下一个词法记号为止。

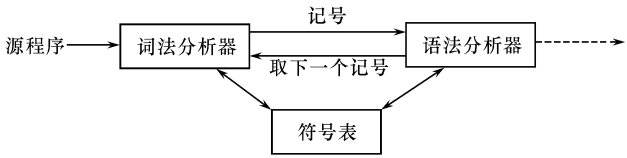


图2.1　词法分析器和语法分析器的相互作用

词法分析器是编译器中读源程序的部分,因而它还可以完成和用户接口的一些其他任务。其一是剥去源程序的注解和(由空格、制表或换行符等引起的)空白。另一任务是把来自编译器各个阶段的错误信息和源程序联系起来,例如,词法分析器记住当前处理的字符行的行号,从而源程序行号可以和错误信息联系到一起。在某些编译器中,复制源程序并把错误信息嵌在其中是词法分析器的责任。通常在词法记号流中已经没有行的概念,因此这样的事情一般由词法分析器来完成。如果源语言支持对宏定义的预处理,该功能可以在词法分析的时候实现。

2.1.1 词法记号、模式、词法单元

在谈论词法分析时,使用术语“词法记号”(简称记号)、“模式”和“词法单元”表示特定的含义。表2.1是使用它们的例子。一般来说,在输入的字符流中有很多字符串,它们的记号是一样的。这样的字符串集合由叫做模式的规则来描述,模式匹配对应集合的任一字符串。模式的形式描述将在下一节讨论。词法单元(lexeme),又称单词,是源程序的字符串,它由模式匹配为记号。例如,Pascal语句

```
var count :integer;
```

的子串count是记号“标识符”的一个词法单元。

本书用黑体字来表示记号,它们将作为源语言语法的终结符。

在大多数程序设计语言中,下列结构处理为记号:关键字、算符、标识符、常数、文字串(字符串)和标点符号。在上面的例子中,当字符串count出现在源程序中时,词法分析器将记号id返回给语法分析器。记号的返回由传递一个代表这个记号的整数来实现。

表2.1 记号的例子

词法记号	词法单元举例	模式的非形式描述
var	var	var
for	for	for
relation	<,<=,=,<>,>,>=	<或 <= 或=或<>或 > 或 >=
id	sum, count, D5	以字母开头的字母数字串
num	3.1416, 10, 2.08E12	任何数值常数
literal	segmentation error	引号 和 之间的任意字符串,但引号本身除外

一个模式是描述源程序中某个记号的词法单元集合的规则。表2.1中记号var的模式是字符串var,它是这个关键字的拼写。记号relation的模式是Pascal的6个关系算符的集合。为了更精确地描述更复杂的记号(如id和num),我们将使用2.2节介绍的正规式。

某些语言的一些规定给词法分析带来了困难。例如,对待空格,不同的语言有不同的规定。在早先的一些语言(如FORTRAN和Algol 68)中,空格无意义(文字串中的除外),而不是作为词法单元的分隔符,它可以随便加入,以改变程序的可读性。空格的这种约定增加了处理标识符记号的复杂性,典型例子是FORTRAN的DO语句,在语句

```
DO 8 I=3.75
```

中,词法分析器只有看见了小数点后,才能确定DO不是关键字,而DO8I是标识符。但是在表面上类似的语句

DO 8 I=3,75

中,DO是关键字,该语句共有7个记号:关键字DO,语句标号8,标识符I,算符(,常数3,逗号和常数75。在没有看见逗号之前,词法分析器不敢保证DO是关键字。空格的这种规定给词法分析器带来的困难是,需要向前阅读多个字符,才能回过头来确定一个记号。

另一个例子是,关键字是否保留。保留字是语言预先确定了含义的词法单元,程序员不可以对这样的词法单元重新声明它的含义,如Pascal语言的var和begin等称为保留字。

很多语言使用关键字概念,并且它们是保留的,因此和上面的保留字没有区别,如C语言和Java语言。但是FORTRAN语言的关键字不保留,如IF,当它作为语句的第一个词法单元时,很可能是关键字,因为这是该关键字出现的地方,但也不排除它是一个程序显式声明的标识符。若IF出现在语句的其他地方,它一定是程序显式或隐式声明的标识符。这就给词法分析带来很大困难,因为识别一个记号和该记号所处的上下文有关了。

顺便需要区分的是标准标识符概念,标准标识符也是预先确定了含义的标识符,程序也可以重新声明它的含义。在这个声明的作用域内,程序声明的含义起作用,而预先确定的含义消失,在其他地方都是预先确定的含义起作用,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。

2.1.2 词法记号的属性

从上一小节知道,Pascal的6个关系算符都属于记号relation。因为从程序的语法是否正确的角度看,使用哪个关系算符都一样。但是从翻译成目标代码来考虑,不同的关系算符,其翻译结果是不一样的。因此词法分析器需要给记号以属性,用属性来记住记号的附加信息,以便需要时使用它们。概括地说,记号影响语法分析的决策,属性影响记号的翻译。

例2.1 Pascal语句

position := initial + rate * 60

的记号和它们的属性值用二元组序列表示如下:

⟨id,指向符号表中position条目的指针⟩

⟨assign_op,⟩

⟨id,指向符号表中initial条目的指针⟩

⟨add_op,+⟩

⟨id,指向符号表中rate条目的指针⟩

⟨mul_op,*⟩

⟨num,整数值60⟩

注意,某些二元组没有属性值,它的第一个成分足以辨别词法单元,例如assign_op。因为+、-和or都可归入add_op,因此add_op在此需要第二元。在这个例子中,记号

num 给了一个整数值属性。编译器也可以把形成数的字符串存入数表,让记号 num 的属性值是指向这个条目的指针。

2.1.3 词法错误

词法分析几乎发现不了源程序的错误,因为词法分析器对源程序采取非常局部的观点。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。

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,它由词法分析器报错。词法分析器也就只能发现这样的错误。

最简单的错误恢复策略是"紧急方式"的恢复。它删掉输入指针当前指向的若干个字符(剩余输入的前缀),直到词法分析器能发现一个正确的记号为止。

另一种策略是进行错误修补尝试。最简单的办法是看一下剩余输入的前缀能否用下面的一个变换变成一个合法的词法单元:

(1) 删除一个多余的字符;

(2) 插入一个遗漏的字符;

(3) 用一个正确的字符代替一个不正确的字符;

(4) 交换两个相邻的字符。

这种策略基于这样的假设,大多数词法错误是多、漏或错了一个字符,或相邻的两个字符错位。这种假设通常是(但不总是)正确的。

2.2 词法记号的描述与识别

上一节提到,字符串集合由叫做模式的规则来描述。正规式是表示这些规则的一种重要方法,因此本节围绕正规式来介绍记号的描述与识别。在介绍正规式前,先给"语言"一个形式化的定义。

2.2.1 串和语言

术语字母表或字符类表示符号的有限集合,符号的典型例子有英文字母和标点符号。集合 {0, 1} 是二进制字母表,ASCII 是计算机字母表的一个例子。

字母表上的串是该字母表符号的有穷序列。串 *s* 的长度是出现在 *s* 中符号的个数，往往写做 $|s|$。例如 banana 是长度为 6 的串，空串是长度为 0 的特殊串，用 ε 表示。

术语语言表示字母表上的一个串集，属于该语言的串称为该语言的句子或字。这个定义相当宽，像 $\varnothing$（空集）和 $\{\varepsilon\}$（仅含空串的集合）这样的抽象语言也符合这个定义，所有语法正确的 Pascal 程序的集合和所有语法正确的英语句子集合也都分别符合此定义。当然，后两个集合更难描述。注意，这个定义并没有对语言中的串赋予任何意义，这个问题在第 4 章讨论。

如果 x 和 y 都是串，那么 x 和 y 的连接（写成 xy）是把 y 加到 x 后面形成的串。对连接运算而言，空串是一个恒等元素，也就是 $s\varepsilon = \varepsilon s = s$。

如果把连接看成“积”，那么可以定义串“指数”。我们定义 s^0 是 ε，定义 s^i 为 $s^{i-1}s\ (i>0)$。因为 εs 是 s 本身，所以 $s^2 = ss$，$s^3 = sss$，等等。

有一些重要的运算可以作用于语言。对词法分析而言，我们感兴趣的运算是和、连接和闭包，它们定义在表 2.2 中。我们把“指数”算符也用于语言，定义 L^0 是 $\{\varepsilon\}$，L^i 是 $L^{i-1}L$，即 L^i 是 L 连接它自己 i−1 次。

表 2.2 语言运算的定义

运算	定义
L 和 M 的和（写成 L∪M）	$L \cup M = \{s \mid s$ 属 L 或 s 属 M$\}$
L 和 M 的连接（写成 LM）	$LM = \{st \mid s$ 属 L 且 t 属 M$\}$
L 的闭包（写成 L^*）	$L^* = \bigcup_{i=0}^{\infty} L^i$，$L^*$ 表示零个或多个 L 连接的并集
L 的正闭包（写成 L^+）	$L^+ = \bigcup_{i=1}^{\infty} L^i$，$L^+$ 表示一个或多个 L 连接的并集

例 2.2 令 L 表示集合 {A, B, …, Z, a, b, …, z}，令 D 表示集合 {0, 1, …, 9}。下面是用表 2.2 定义的运算作用于 *L 和 D 所得到的新语言的例子。*

(1) $L \cup D$ *是字母和数字的集合；*

(2) LD 是所有一个字母后跟随一个数字的串的集合；

(3) L^6 是 6 个字母的串的集合；

(4) L^* 是所有字母串（包括 ε）的集合；

(5) $L(L \cup D)^*$ 是以字母开头的所有字母数字串的集合；

(6) D^+ 是不含空串的数字串的集合。

从这个例子可以看出，从基本集合开始，利用语言上的运算可以定义新的语言。下面将用更有利于计算机处理的形式来定义一类简单的语言。

2.2.2 正规式

正规式(又称正规表达式)是按照一组定义规则,由较简单的正规式构成的,每个正规式 *r* 表示一个语言 *L*(*r*)。定义规则说明 *L*(*r*)是怎样以各种方式从 *r* 的子正规式所表示的语言组合而成的。

下面是定义字母表 Σ 上正规式的规则,和每条规则相联的是被定义的正规式所表示的语言的描述。

(1) ε是正规式,它表示{ε};

(2) 如果 a 是Σ上的符号,那么 a 是正规式,它表示{a}。虽然都用同样的符号表示,但正规式 a 是不同于串 a 或符号 a 的,从上下文可以清楚地区别所谈到的 a 是正规式、串还是符号;

(3) 假定 r 和 s 都是正规式,它们分别表示语言 *L*(*r*)和 *L*(*s*),那么(*r*)|(*s*)、(*r*)(*s*)、(*r*)*和(*r*)都是正规式,分别表示语言 *L*(*r*)∪*L*(*s*)、*L*(*r*)*L*(*s*)、(*L*(*r*))*和 *L*(*r*)。

正规式表示的语言叫做正规集。

如果约定:

(1) 闭包运算(算符是*)有最高的优先级并且是左结合的运算

(2) 连接运算(两个正规表达式并列)的优先级次之且也是左结合的运算

(3) 或运算(算符是|)的优先级最低且仍然是左结合的运算

那么可以避免正规式中一些不必要的括号。例如,((a)(b)*)|(c)等价于 ab*|c。

例2.3 令字母表 Σ={a,b},那么:

(1) 正规式 a|b 表示集合{a,b}。

(2) 正规式(a|b)(a|b)表示{aa,ab,ba,bb},即由 a 和 b 构成的所有长度为2的串集。表示同样集合的另一正规式是 aa|ab|ba|bb。

(3) 正规式 a* 表示仅由字母 a 构成的所有串的集合,包括空串。

(4) 正规式(a|b)*表示由 a 和 b 构成的所有串的集合,包括空串。

如果两个正规式 r 和 s 表示同样的语言,则说 r 和 s 等价,写作 r=s。例如,(a|b)=(b|a)。

正规式遵守一些代数定律,它们可用于正规式的等价变换,表2.3列出了正规式 *r*、*s* 和 *t* 遵守的代数定律。

表 2.3 正规式的代数性质

公理	描述
$r\|s = s\|r$	|是可交换的
$r\|(s\|t) = (r\|s)\|t$	|是可结合的
$(rs)t = r(st)$	连接是可结合的
$r(s\|t) = rs\|rt$ $(s\|t)r = sr\|tr$	连接对|是可分配的
$\varepsilon r = r$ $r\varepsilon = r$	ε 是连接的恒等元素
$r^* = (r\|\varepsilon)^*$	* 和 ε 之间的关系
$r^{**} = r^*$	* 是幂等的

2.2.3 正规定义

可以对正规式命名,并用这些名字来引用相应的正规式,以求得表示上的简洁。这些名字也可以像符号一样,出现在正规式中。

如果 Σ 是基本符号的字母表,那么正规定义是形式为

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

…

$d_n \rightarrow r_n$

的定义序列,各个 d_i 的名字都不同,每个 r_i 都是 $\Sigma \cup \{d_1, d_2, \cdots, d_{i-1}\}$ 上的正规式。由于每个 r_i 只能含 Σ 上的符号和前面定义的名字,因而不会出现递归定义的情况。把这些名字用它们所表示的正规式来代替,就可以为任何 r_i 构造 Σ 上的正规式。

为了区别名字和符号,本书在正规定义中用黑体字表示名字。

例 2.4 Pascal 语言的标识符集合含所有以字母开头的字母数字串,下面是这个集合的正规定义。

letter → A | B | … | Z | a | b | … | z

digit → 0 | 1 | … | 9

id → letter (letter | digit) *

例 2.5 Pascal 的无符号数是 1946,11.28,63.6E8 和 1.999E−6 这样的串,下面是这样的串集的正规定义。

digit → 0 | 1 | … | 9

digits→digit digit*

optional_fraction→.digits|ε

optional_exponent→(E(+|-|ε)digits)|ε

num→digits optional_fraction optional_exponent

从这个定义可以知道,无符号数由整数部分、小数部分和指数部分三部分组成,其中小数部分和指数部分都是可能出现或可能不出现的。指数部分如果出现的话,是E及可能有的+或-号,再跟上一个或多个数字。注意小数点后面至少有一个数字,所以num能匹配20,但不能匹配2。

在正规式中,某些结构频繁出现,为方便起见,用缩写表示它们。

(1) 一个或多个实例　一元后缀算符"+"的意思是"一个或多个实例",即正规式a^+表示一个或多个a的所有串的集合。算符+和算符*有同样的优先级和结合性。代数恒等式$r^*=r^+|\varepsilon$和$r^+=rr^*$表达了这两个算符之间的关系。

(2) 零个或一个实例　一元后缀算符?的意思是"零个或一个实例",$r?$是$r|\varepsilon$的缩写。如果r是正规式,那么(r)?是表示语言L(r)∪{ε}的正规式。使用这两种缩写,可以用

num→digit+(.digit+)?(E(+|-)?digit+)?

来描述无符号数。

(3) 字符组　[abc](其中a、b和c是字母表的符号)表示正规式a|b|c。缩写字符组[a-z]表示正规式a|b|…|z。使用字符组,可以用正规式

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

描述标识符。

2.2.4 状态转换图

本书以下面正规定义描述的语言为例,讨论怎样识别记号。

例2.6　考虑下面的正规定义:

while→while

do→do

relop→<|<=|=|<>|>|>=

id→letter(letter|digit)*

num→digit+(.digit+)?(E(+|-)?digit+)?

其中letter和digit同前面所定义的一样。这是while语句中可能出现的部分记号的描述。词法分析器将识别保留字while和do,还有关系算符、标识符和数。假定词法单元之间必要时由空格(或制表符、换行符)分开,词法分析器通过把剩余输入的前缀和下面的正规定义ws相比较来完成忽略词法单元之间的空格。

delim→blank|tab|newline

ws→delim^{+}

如果剩余输入的前缀能由 ws 匹配,词法分析器不返回记号给分析器,它继续去寻找空格后面的记号,然后返回到分析器。

词法分析器输出的是(记号、属性)二元组序列,用表 2.4 给出各正规式描述的记号名称及各记号可能有的属性值。注意,正规式 ws 没有对应的记号,另外,关系算符的属性值由符号常量 LT、LE、EQ、NE、GT 和 GE 给出。

用状态转换图(简称转换图)作为构造词法分析器的第一步。状态转换图描绘语法分析器为得到下一个记号而调用词法分析器时,词法分析器所做的动作。

表 2.4 正规式、记号和属性

正规式	记号	属性值
ws	—	—
while	while	—
do	do	—
id	id	符号表条目的指针
num	num	数表条目的指针
relop	relop	LT,LE,EQ,NE,GT,GE

图 2.2 是识别记号 relop 的转换图,可以用这张图来解释有关转换图的概念。转换图上的圆圈叫做状态,状态由箭头连接,称为边,边上有指示输入字符的标记,标记通常是一个字符。若离开状态 *s* 的某个边上有标记 other,则它表示离开 *s* 的其他边所指示的字符以外的任意字符。这一节的转换图是确定的,即没有任何字符能够和离开一个状态的两条边上的标记都匹配。这个条件在后面会放宽。

有一个状态标记为开始状态,这是状态转换图的初启状态,开始识别记号时,控制进入开始状态。控制进入某个状态时,读输入串的下一个字符,如果离开这个状态的一条边上的标记和该输入字符匹配,控制就进入由这条边指向的状态,否则识别过程失败。某些状态有两个圆圈,它们是接受状态,控制进入这样的状态表示识别了一个记号。接受状态可以有动作,控制到达接受状态时执行它的动作。

对于图 2.2 的状态转换图,如果输入串是"<=…",那么控制从开始状态 0 到达接受状态 2,读出词法单元<=,执行动作 return(relop, LE)。

注意,如果到达接受状态 4,意味着<和另一个字符已被读过,由于这第二个字符不是关系算符<的一部分,因此必须把输入串上指示下一个字符的指针回移一个字符。用*表

示输入指针必须回移的状态。

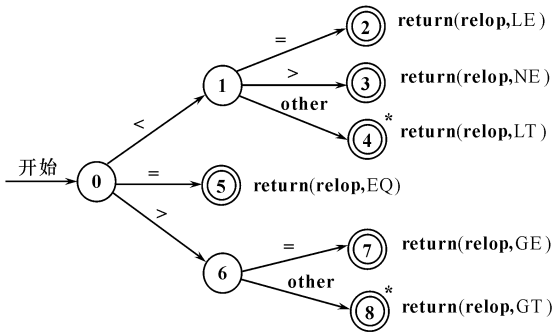


图2.2 关系算符的转换图

例2.7 图2.3是识别标识符的转换图,其中边上的标记 letter 和 digit 分别指字母集和数字集。边上的标记是一个字符集时,若输入字符是该字符集的成员,则称该标记和这个字符匹配。也可以用该转换图来识别保留字,因为保留字是特殊的标识符。当然,到达接受状态时,需要执行某段代码,以决定到达接受状态的词法单元是保留字还是标识符。

把保留字从标识符中分离出来的简单办法是建立一张保留字表。在扫描任何字符之前,把构成保留字的串 while 和 do 等都置入该表,把它们对应的记号也加入该表,以便识别这些串时返回。图2.3中接受状态旁边的返回语句使用 *install_id() 来获得要返回的记号* 和属性。过程 install_*id*() 首先查看保留字表,如果当前词法单元构成保留字,则返回相应的记号;否则该词法单元是标识符,该过程再查标识符表,如果在表中发现该词法单元则返回相应的条目指针,如果没有找到,则把该词法单元填入标识符表,并返回新建条目的指针。(很多编译器在语法分析阶段才将标识符填入标识符表,这时 id 的属性是它的拼写。)

如果要识别的保留字有所变化,无需修改转换图,只需给保留字表重新置初值。

为保留字单独构造转换图是可能的。对典型的程序设计语言,这么做会使词法分析器的状态数多达几百个。而用上面的方法,不到100个状态可能就够了。

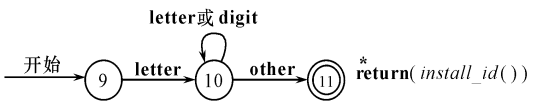


图2.3 标识符和保留字的转换图

例2.8 下面根据正规定义

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

为 Pascal 无符号数构造识别器。注意,这个定义中,小数部分(.digit$^+$)和指数部分(E(+|-)?digit$^+$)是可选的。图 2.4 是它的状态转换图。

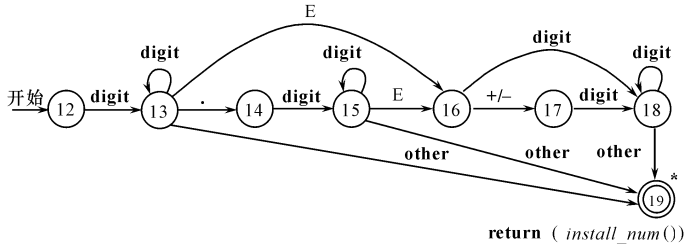


图 2.4 Pascal 无符号数转换图

到达接受状态的动作是调用过程 *install_num*(),它把词法单元置入数表,并返回建立的条目指针。词法分析器返回记号 num 和作为属性值的这个指针。

把图 2.2、图 2.3 和图 2.4 的三个开始状态 0、9 和 12 合并成一个开始状态,就可以把这三个转换图合并成一个转换图。

剩下的问题是空白。代表空白的 ws 的处理和上面讨论的代表各种记号的正规式的处理有所不同,因为在输入串中发现空白时,并没有任何东西返回给语法分析器。识别 ws 的转换图如图 2.5 所示。把这个转换图和其他几个转换图合并成一个转换图后,到达接受状态 22 的动作就是回到开始状态,识别下一个记号。

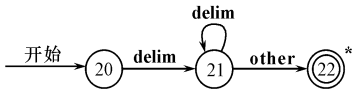


图 2.5 ws 的转换图

从正规式构造了转换图后,基于这样直观的转换图,编写识别这些正规式描述的记号的程序就不是件困难的事情了。当然,直接编写词法分析器并非一定要基于转换图概念,本节介绍转换图是为了便于大家接受有限自动机的概念。

2.3 有限自动机

语言的识别器是一个程序,它取串 x 作为输入,当 x 是语言的句子时,它回答“是”,否则回答“不是”。可以通过构造称为有限自动机的更一般的转换图,把正规式翻译成识别器。

有限自动机分成确定的和不确定的两种情况。“不确定”的含义是,存在这样的状态,对

于某个输入符号,它存在不止一种转换。

确定的和不确定的有限自动机都正好能识别正规集,也就是它们能识别的语言正好是正规式所能表达的语言。但是,它们之间存在着时空权衡问题:从确定的有限自动机得到的识别器,比从等价的不确定的有限自动机得到的识别器要快得多;但是,确定的有限自动机可能比等价的不确定的有限自动机占用更多的空间。由于把正规式变成不确定的自动机更直接一些,因此首先讨论这一类自动机。

本节和下一节的基本例子是正规式 $(a|b)^*ab$ 表示的语言。类似的语言在实际中也出现,例如,表示所有以.o结尾的文件名的正规式是 $(.|o|c)^*.o$ 的形式,其中c代表除.和o以外的任何字符。另一个例子是,C语言的注释是以/*开始和以*/结束的任意字符串,但它的任何前缀(本身除外)不以*/结尾。

2.3.1 不确定的有限自动机

不确定的有限自动机(简称NFA)是一个数学模型,它包括:

(1) 一个有限的状态集合 S;

(2) 一个输入符号的集合 Σ(输入符号字母表);

(3) 一个转换函数 $move: S\times(\Sigma\cup\{\varepsilon\})\to P(S)$(S的幂集),它把状态和符号(可以是 ε)两元组映射到状态的集合;

(4) 状态 s_0 是惟一的开始状态;

(5) 状态集合 F 是接受(或终止)状态集合,并且 $F\subseteq S$。

NFA可以用带标记的有向图表示,叫做状态转换图,结点表示状态,有标记的边代表转换函数。这种转换图和上一节所讲的略有区别,在这里,同样的符号可以标记出自一个状态的多条边。另外,边可以由输入符号标记,也可以由特殊符号 ε 标记。

可以识别语言 $(a|b)^*ab$ 的NFA的转换图如图2.6所示。这个NFA的状态集合是{0,1,2},输入符号表是{a,b},状态0是开始状态,接受状态2用双圈表示。

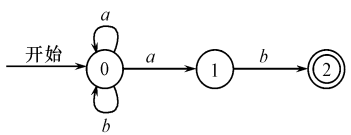


图2.6 识别 $(a|b)^*ab$ 的NFA

描述NFA时,可以用这种转换图表示;而在计算机上,NFA可以用不同的方法实现。最简单的办法是用转换表,每个状态一行,每个输入符号和 ε(如果需要的话)各占一列,表的第i行中符号a的条目是一个状态集合(说得更实际一些,是状态集合的指针),这是NFA

在输入为 *a*时,状态 *i*所能到达的状态集合。表2.5是对应图2.6的NFA的转换表。

表2.5 图2.6的NFA的转换表

状态	输入符号	
	a	*b*
0	{0,1}	{0}
1		{2}
2		

转换表的优点是可以快速访问给定状态和字符的状态集。它的缺点是,当输入字母表较大,并且大多数转换是空集时,占用了大量空间。显然,很容易把有限自动机的一种实现转变成另一种实现。

NFA接受输入串 *x*,当且仅当转换图中存在从开始状态到某个接受状态的路径,该路径各边上的标记拼成 *x*。图2.6的NFA接受输入串 *ab*, *aab*, *bab*, *aaab*,…。例如,从状态0开始,沿着标记为 a的边再到状态0,然后沿着标记分别为 a和b的边先后到达状态1和2构成的路径,接受 aab。对一个输入,可能有多条路径可以到达接受状态。

由NFA定义的语言是它接受的输入串集合,不难看出图2.6的NFA识别(也称接受) $(a|b)^*ab$。

例2.9 图2.7是识别 $aa^*|bb^*$的NFA,串 *aaa*由通过0,1,2,2和2的路径来接受,相应边上的标记分别是 *ε*, *a*, *a*和 *a*,它们拼成 *aaa*, *ε*在拼接中"消失"。

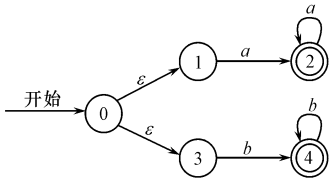


图2.7 识别 $aa^*|bb^*$的NFA

2.3.2 确定的有限自动机

确定的有限自动机(简称DFA)是不确定的有限自动机的特殊情况。其中:

(1) 任何状态都没有 *ε*转换,即没有任何状态可以不进行输入符号的匹配就直接进入下一个状态;

(2) 对任何状态 *s*和任何输入符号 *a*,最多只有一条标记为 *a*的边离开 *s*,即转换函数

move : $S \times \Sigma \to S$ 可以是一个部分函数。

确定的有限自动机从任何状态出发,对于任何输入符号,最多只有一个转换。如果用转换表表示 DFA 的转换函数,那么表中的每个栏目最多只有一个状态。结果是,很容易确定 DFA 是否接受某输入串,因为从开始状态起,最多只有一条到达某个终态的路径可由这个串标记。下面的算法表明怎样模拟 DFA 的行为。

算法 2.1 模拟 DFA。

输入 输入串 *x*,由文件结束符 eof 结尾。一个 DFA *D*,*其开始状态是* s_0,其接受状态集合是 *F*。

输出 如果 *D 接受 x*,则回答"yes",否则回答"no"。

方法 把图 2.8 的算法施加于输入串 *x*。*函数 move* (*s*, *c*) *给出一个状态,它是面临输*入符号 c,状态 s 的转换。函数 nextchar () 返回输入串 x 中的下一个字符。

```
s := s0;
c := nextchar();
while c ≠ eof do
    s := move(s, c);
    c := nextchar();
end;
if s属于 F then
    return yes
else return no;
```

图 2.8 模拟 DFA

例 2.10 图 2.9 的转换图表示一个 DFA,它和图 2.6 的 NFA 识别同样的语言 $(a|b)^*abb$。用这个 DFA 和输入串 *abab*,算法 2.1 沿着状态 0,1,2,1 和 2 移动,并返回"yes"。

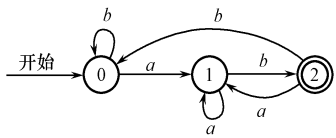


图 2.9 识别 $(a|b)^*abb$ 的 DFA

2.3.3 NFA 到 DFA 的变换

图 2.6 的 NFA 在状态 0,面对输入 *a* 时有两个转换,它可以进入状态 0 或 1。类似地,图

2.7的NFA在状态0,不接受任何输入(即面对 ε)也有两个转换。有些场合,还会出现既可以根据 ε 也可以根据一个实际输入符号进行转换的情况,这同样也会引起二义。这种转换函数多值的情况,使得很难用计算机程序模拟NFA。“接受”的定义仅仅是说必须存在从开始状态到某个接受状态的一条路径,该路径的标记正好拼成输入串。这样,在找到一条接受路径或确定没有这样的路径前,可能不得不同时考虑所有这些路径。

现在给出一个算法,它从NFA构造识别同样语言的DFA。这个算法通常称为子集构造法,它对于计算机程序模拟NFA来说也是有用的,一个和它密切相关的算法是下一章构造LR分析器的基础。

首先概述子集构造法的思想。在NFA的转换表里,每个条目是一个状态集;在DFA的转换表中,每个条目只有一个状态。从NFA构造等价的DFA的一般思想是让新构造的DFA的每个状态代表NFA的一个状态集,这个DFA用它的状态去记住该NFA在读输入符号后能到达的所有状态。也就是说,在读了输入 $a_1a_2\cdots a_n$ 后,这个DFA到达一个代表该NFA状态子集 T *的状态,这个子集* T 就是从该NFA的开始状态沿着那些标有 $a_1a_2\cdots a_n$ 的路径能到达的所有状态的集合。这样,这个DFA的状态数和该NFA的状态数是成指数变化的,但是实际上,这种最坏的情况很少发生。

算法2.2 从NFA构造DFA(子集构造法)。

输入 一个NFA N。

输出 一个接受同样语言的DFA D。

方法 为D构造转换表Dtran,表中的每个状态是N的状态集合,D“并行”地模拟N面对输入串的所有可能的移动。

用表2.6的运算来计算NFA状态集的变化(s 代表NFA的状态,T 代表NFA的状态集)。

在读第一个输入符号前,N *可以处于集合* ε-*closure*(s_0) 的任何状态,其中 s_0 是N的开始状态。假定集合T是从 s_0 出发,面临某个输入串所能到达的状态集合,令a是下一个输入符号,那么看见a时,N可以移动到集合move(T, a)中的任何状态。由于允许ε转换,看见a后,N可以处于 ε-*closure*(*move*(*T*, *a*)) 中的任何状态。

表2.6 对NFA状态的运算

运算	描述
ε-*closure*(*s*)	从NFA的状态 *s出发,只用* ε转换能到达的NFA状态集合
ε-*closure*(*T*)	NFA的状态集合 $\{s \mid s\in \varepsilon\text{-}closure(t)\ \&t\in T\}$
move(*T*, *a*)	NFA的状态集合 $\{s \mid s=move(t,a)\ \&t\in T\}$

按图 2.10 所示的算法构造 *D* 的状态集合 *Dstates* 和转换表 *Dtran*。*D* 的每个状态对应于 NFA 的一个状态集合,它是 *N* 读了某个符号串后所能到达的全部状态,包括 ε 转换后的所有状态。D 的开始状态是 ε－*closure*(*s*$_0$)。如果 D 的状态是至少含 N 的一个接受状态的状态集,那么它是 D 的一个接受状态。

```
初始,ε－closure(s0)是 Dstates仅有的状态,并且尚未标记;
while Dstates有尚未标记的状态 T do begin
    标记 T;
    for 每个输入符号 a do begin
        U := ε－closure(move(T, a));
        if U不在 Dstates中 then
            把 U作为尚未标记的状态加入 Dstates;
        Dtran[T, a] := U
    end
end
```

图 2.10 子集构造法

ε－*closure*(*T*)的计算是从给定的结点集合出发,在图上搜索可达结点的典型过程。*T* 看成结点集合,图只包含 NFA 的含 ε 标记的边。计算 ε－*closure* 的简单算法是用栈来保存那些边还没有完成 ε 转换检查的状态。图 2.11 描述了这样的过程。

```
把 T的所有状态压入栈;
ε－closure(T)的初值置为 T;
while 栈非空 do begin
    把栈顶元素 t弹出栈;
    for 每个状态 u(条件是从 t到 u的边上的标记为 a) do
        if u不在 ε－closure(T)中 do begin
            把 u加入 ε－closure(T);
            把 u压入栈
        end
end
```

图 2.11 ε－*closure*(*T*)的计算

例 2.11 图 2.12 是接受语言 (a|b)**ab* 的另一个 NFA *N*,之所以用它做例子,是因为下一节“机械地构造 NFA”算法也是以此为例。把算法 2.2 运用到 *N*,其等价的 DFA 的开始状态是 ε－*closure*(0),它是 *A*={0, 1, 2, 4, 7},因为它们正好是从状态 0 出发,经过标记都

是 ε 的路径所能到达的所有状态。由于路径可以没有边,所以 0 也是经这样的路径从 0 能到达的状态。

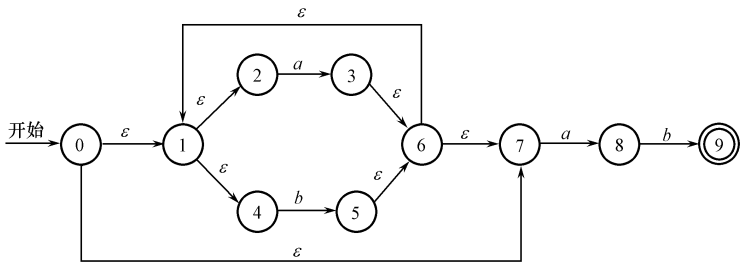


图 2.12 识别 (a|b)* (ab) 的 NFA

这里的输入字母表是 {a, b} 。根据图 2.10 的算法,首先标记 A,然后计算

ε− closure (move (A, a))

由于在 A= {0, 1, 2, 4, 7} 中,只有 2 和 7 有 a 转换,分别到 3 和 8,因此 move (A, a) = {3, 8} 。所以

ε− closure (move (A, a)) = ε− closure ({3, 8}) = {1, 2, 3, 4, 6, 7, 8}

后一步的结果是因为 ε− closure (3) = {1, 2, 3, 4, 6, 7} 并且 ε− closure (8) = {8} 。我们称这个集合为 B,于是, Dtran [A, a] = B。

在 A 中,只有状态 4 含 b 转换到 5,所以该 DFA 状态 A 的 b 转换到达

ε− closure (move (A, b)) = ε− closure ({5}) = {1, 2, 4, 5, 6, 7}

令该集合为 C,于是, Dtran [A, b] = C。

用新的没有标记的集合 B 和 C 继续这个过程,最终会达到这样一点:所有的集合 (即 DFA 的所有状态) 都已标记。因为 10 个状态的集合的不同子集只有 2^{10} 个,一个集合一旦标记就永远是标记的,所以终止是肯定的。对本例,实际构造出的 4 个不同状态集合是:

A= {0, 1, 2, 4, 7}

B= {1, 2, 3, 4, 6, 7, 8}

C= {1, 2, 4, 5, 6, 7}

D= {1, 2, 4, 5, 6, 7, 9}

状态 A 是开始状态,状态 D 是仅有的接受状态,完整的转换表 Dtran 如表 2.7 所示。

这个 DFA 的转换图见图 2.13。必须注意,图 2.9 的 DFA 也接受 (a|b)* ab,并且状态少一个。下一小节将讨论 DFA 的化简问题。

表 2.7 NFA 的转换表 *Dtran*

状态	输入符号	
	a	*b*
A	*B*	*C*
B	*B*	*D*
C	*B*	*C*
D	*B*	*C*

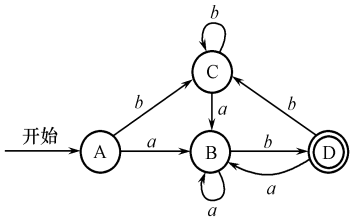


图 2.13 子集构造法用于图 2.12 的结果

2.3.4 DFA 的化简

理论上的一个重要结论是,每一个正规集都可以由一个状态数最少的 DFA 识别,这个 DFA 是惟一的(因状态名不同的同构情况除外)。本小节介绍如何把一个 DFA 化简到状态数最少,并且识别同样的语言。

我们的方法基于转换函数是全函数。如果一个 DFA 的转换函数不是全函数,可以引入一个"死状态" s_d,s_d 对所有输入符号都转换到 s_d 本身,如果 s 对符号 a 没有转换,那么加上从 s 到 s_d 的 a 转换。显然,加死状态后的 DFA 和原来的 DFA 等价。

对于 DFA *M*,*如果从状态 s 出发,在面临输入 w 时,最后停在某个接受状态,而从状态 t 出发,面临同样的输入时,它停在一个非接受状态*,或者反过来,串 w 可用来区别状态 s 和 t。如果找不到任何串来区别 s 和 t,那么 s 和 t 是不可区别的。例如,任何接受状态和非接受状态可用空串 ε 来区别。在图 2.13 中,A 和 B 由输入 b 来区别,因为对输入 b,A 走到非接受状态 C,而 B 走到接受状态 D。

极小化 DFA 状态数的算法就是把它的状态分成一些不相交的子集,每一子集的状态都是不可区别的,不同子集的状态都是可区别的。每个子集合并成一个状态。

最初,这个划分包括两个子集,即接受状态子集和非接受状态子集,因为它们可用空串来区别。然后,检查每一个子集,看其中的状态是否还可区别。对于一个状态子集,比如 A

$= \{s_1, s_2, \cdots, s_k\}$，和某个输入符号 a，检查 $s_1, s_2, \cdots, s_k$ 面临 a 的转换，如果这些转换所到的状态落入当前划分的两个或更多的状态子集中，那么 A 必须进一步划分，使得 A 的子集的 a 转换能落入当前划分的一个状态子集中。例如，若 s_1 和 s_2 的 a 转换分别到达 t_1 和 t_2，并且 t_1 和 t_2 在当前划分的不同子集中，那么 A 至少要分成两个子集，一个含 s_1，另一个含 s_2。注意，如果 t_1 和 t_2 是可由某个串 w 区别的，那么 s_1 和 s_2 一定是可由串 aw 区别的。

重复这个对当前的划分进一步细分的过程，直到没有任何一个子集再需细分为止。当说明为什么分在不同子集中的状态是可区别的时候，我们并没有说明最终留在一个子集中的状态是不能由任何输入串来区别的，这个证明留给有兴趣的读者。还有另一个问题也留给有兴趣的读者，这个问题是，从最终划分的每个子集中取一个状态，扔掉死状态和从开始状态不可到达的状态，所构造的 DFA 就是接受同样语言的状态数最少的 DFA。

算法 2.3 极小化 DFA 的状态数。

输入 一个 DFA *M*，它的状态集合是 *S*，输入符号集合是 Σ，转换函数是 $f: S \times \Sigma \to S$，开始状态是 s_0，接受状态集合是 *F*。

输出 一个 DFA *M′*，它和 *M* 接受同样的语言，且状态数最少。

方法 (1) 构造状态集合的初始划分 Π：分成两个子集，接受状态子集 F 和非接受状态子集 $S-F$。

(2) 应用下面的过程对 Π 构造新的划分 Π_{new}

for Π 中的每个子集 *G* do begin

把 *G* 划分成若干子集，*G* 的两个状态 *s* 和 *t* 在同一子集中，当且仅当对任意输入符号 a，s 和 t 的 a 转换是到 Π 的同一子集中。

在 Π_{new} 中，用 G 的划分代替 G。

end

(3) 如果 $\Pi_{new} = \Pi$，则让 $\Pi_{final} = \Pi$，再执行步骤 (4)，否则，令 $\Pi = \Pi_{new}$，转 (2)。

(4) 在 Π_{final} 的每个状态子集中选一个状态代表它，这些代表就是最简 DFA *M′* 的状态。如果 s 是这样的一个代表，在 DFA *M* 中，若 s 的 a 转换到 t，并且 t 所在子集的代表是 r（r 可能就是 t），那么，在 M′ 中，s 的 a 转换到 r。包含 s_0 的状态子集的代表是 M′ 的开始状态，M′ 的接受状态是那些原先属于 F 集合的代表。注意，Π_{final} 的每个子集或者仅含 F 中的状态，或者不含 F 中的状态。

(5) 如果 M′ 有死状态，则去掉它。从开始状态不可及的状态也删除。从任何其他状态到死状态的转换都成为无定义。

再次提请注意，使用这个算法时，其输入 DFA 的状态转换函数必须是全函数，否则有可能得到的新 DFA 和原来的 DFA 接受的不是同一个语言。前面讲的加死状态的目的就是把转换函数变成全函数。

例 2.12 重新考虑图 2.13 代表的 DFA。初始划分 Π 包括两个子集:接受状态子集 {D} 和非接受状态子集 {A, B, C}。为了构造 Π_{new},首先考虑 {D},因为这个子集只包含一个状态,它不能再划分,所以在 Π_{new} 中仍是 {D}。然后考虑 {A, B, C},对于输入 a,这些状态都转换到 B,但对于输入 b,A 和 C 都转换到状态子集 {A, B, C} 的一个成员,而 B 转换到 D,是另一个子集的成员。于是,在 Π_{new} 中,状态子集 {A, B, C} 必须分成两个新子集 {A, C} 和 {B},Π_{new} 成了 {A, C}、{B} 和 {D}。

再次扫描,只有 {A, C} 有划分的可能。但是对于输入 a 和 b。它们都是分别转换到 B 和 C,因而不必再划分。即这遍扫描后,$\Pi_{new} = \Pi$。*所以,* Π_{final} 是 {A, C}、{B} 和 {D}。

如要选择 A 作为 {A, C} 的代表,选择 B 和 D 作为其他单状态子集的代表,可以得到最简自动机。它的转换表如表 2.8 所示,状态 A 是开始状态,状态 D 是惟一的接受状态。

表 2.8 最简 DFA 的转换表

状态	输入符号	
	a	*b*
A	*B*	*A*
B	*B*	*D*
D	*B*	*A*

例如,在这最简的自动机中,D 的 b 转换到 A,因为在原来的自动机中,D 的 b 转换到 C,并且 A 是 C 所在子集的代表。类似的变化也发生在状态 A 面临输入 b 时。其余的都是从图 2.13 中复制过来。该图没有死状态,并且所有的状态都是从开始状态 A 可到达的。

2.4 从正规式到有限自动机

有很多办法可以从正规式建立识别器,每种办法都有它的长处和短处。本书介绍的是从正规式构造 NFA,然后用上节的子集构造法把 NFA 变成 DFA,并把它化简。本节将给出从正规式构造 NFA 的算法。

该算法有很多变种,这里提出一种容易实现的简单版本。该算法是语法制导的,它用正规式语法结构来制导构造过程。首先构造识别 ε 和字母表中一个符号的自动机,然后构造识别主算符为选择、连接或闭包的正规式的自动机。例如,对于正规式 r|s,*从 r 和 s 的* NFA 中归纳构造出它的 NFA。

在构造过程中,每步最多引入两个新的状态,所以为正规式构造的最终 NFA,状态数最

多是正规式中符号和算符总数的两倍。

算法 2.4 从正规式构造 NFA。

输入 字母表 Σ 上的正规式 r。

输出 接受 $L(r)$ 的 NFA N。

方法 首先分析 r,把它分解成子表达式,然后使用下面的规则 (1) 和 (2),为 r 中的每个基本符号 (ε 或字母表符号) 构造 NFA。基本符号对应正规式定义的 (1) 和 (2) 两部分。要注意,如果符号 a 在 r 中出现多次,那么要为它的每次出现构造 NFA。

然后,根据正规式 r 的语法结构,用下面的规则 (3) 归纳地组合这些 NFA,直到获得整个正规式的 NFA 为止。在构造过程中所产生的中间 NFA 有一些重要的性质:只有一个终态,没有边进入开始状态,也没有边离开终态。

(1) 对于 ε,构造如图 2.14 所示的 NFA,其中 i 是开始状态,f 是接受状态。很明显,这个 NFA 识别 $\{\varepsilon\}$。

(2) 对 Σ 中的每个符号 a,构造如图 2.15 所示的 NFA。同样,i 是开始状态,f 是接受状态。这个 NFA 识别 $\{a\}$。

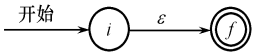


图 2.14 识别正规式 ε 的 NFA

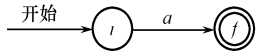


图 2.15 识别正规式 a 的 NFA

(3) 如果 N(s) 和 N(t) 分别是正规式 s 和 t 的 NFA,则:

(a) 对于正规式 $s|t$,构造合成的 NFA $N(s|t)$,结果如图 2.16 所示。这里 i 是新的开始状态,f 是新的接受状态。从 i 到 N(s) 和 N(t) 的开始状态有 ε 转换,从 N(s) 和 N(t) 的接受状态到 f 也有 ε 转换。N(s) 和 N(t) 的开始和接受状态不是 $N(s|t)$ 的开始和接受状态。这样,从 i 到 f 的任何路径必须排他地通过 N(s) 或 N(t)。这个合成的 NFA 识别 $L(s) \cup L(t)$。

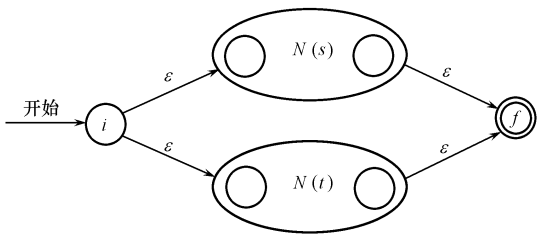


图 2.16 识别正规式 $s|t$ 的 NFA

(b) 对于正规式 st。构造合成的NFA *N(st)*,结果如图2.17所示。N(s)的开始状态成为合成后的NFA的开始状态,N(t)的接受状态成为合成后的NFA的接受状态,N(s)的接受状态和N(t)的开始状态合并,也就是N(t)开始状态的所有转换成为N(s)的接受状态的转换。合并后的这个状态不作为合成后的NFA的接受状态或开始状态。从i到f的路径必须首先经过N(s),然后经过N(t),所以这种路径上的标记拼成L(s)L(t)的串。因为没有边进入N(t)的开始状态或离开N(s)的接受状态,所以在i到f的路径中不存在N(t)回到*N(s)的现象,故合成的*NFA识别L(s)L(t)。

(c) 对于正规式s^*,构造合成的NFA *N*(s^*),结果如图2.18所示。同样,i和f分别是新的开始状态和接受状态。在这个合成的NFA中,可以沿着ε边直接从i到f,这代表ε属于(L(s))*,也可以从i经过N(s)一次或多次。显然,这个NFA识别(L(s))*。

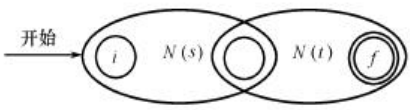


图2.17 识别正规式 *st的*NFA

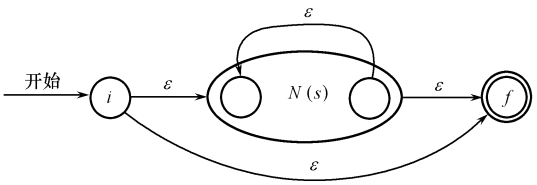


图2.18 识别正规式 s^* 的NFA

(d) 对于括起来的正规式(s),使用N(s)本身作为它的NFA。

对于每次构造的新状态都赋予不同的名字。这样,所有的状态都有不同的名字。

可以检验,算法2.4构造的每一步都产生识别对应语言的NFA。此外,产生的NFA有下列性质:

(1) N(r)的状态数最多是r中符号和算符总数的两倍。因为构造的每一步最多引入两个新的状态。

(2) N(r)只有一个接受状态,接受状态没有向外的转换。

(3) N(r)的每个状态有一个用Σ的符号标记的指向其他结点的转换,或者最多两个指向其他结点的ε转换。

例2.13 用算法2.4构造正规式 $r=(a|b)^*ab$ *的*NFA *N(r)*。*图*2.19是 *r的分析树。*对于成分r_1和r_2,构造它们的NFA,再用选择规则组合N(r_1)和N(r_2),得到$r_3=r_1|r_2$的NFA。(r_3)的NFA和r_3的一样,再构造(r_3)*的NFA。这样依次下去,最后得到 $r=(a|b)^*ab$ *的*NFA如图2.20所示,它和图2.12一致。

从手工构造NFA的角度看,算法2.4的缺点是引入了大量的 *ε转换,使得后面手工将*NFA确定化时,容易因疏忽而出错。因此在手工构造NFA时,应避免引入ε转换,例如图2.6识别 $(a|b)*ab$ *的*NFA就比图2.20的要简单得多。

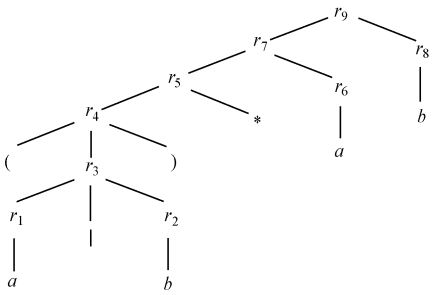


图2.19　(a|b)* ab的分解

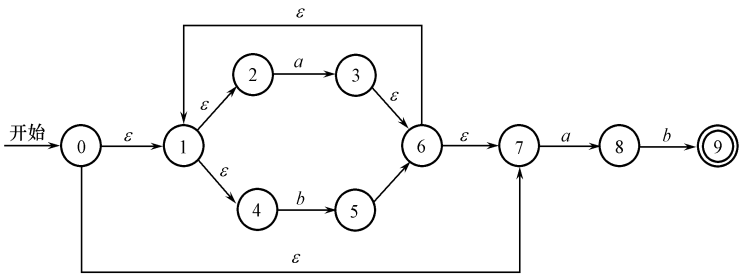


图2.20　识别 (a|b)* ab的NFA

2.5　词法分析器的生成器

本节描述一个特殊的工具——Lex,它从基于正规式的描述来构造词法分析器,并且已广泛用于描述各种语言的词法分析器。这个工具也称为Lex编译器,它的输入是用Lex语言编写的。讨论这个工具将使我们知道,基于正规式的模式说明是怎么和要求词法分析器完成的动作(例如,在符号表中增加新条目)组织在一起,从而形成词法分析器的规范。即使没有可用的Lex编译器,这样的规范也还是有用的,因为可以按2.2节的转换图技术手工地构造出词法分析器。

Lex通常按图2.21描绘的方式使用。首先,词法分析器的说明是用Lex语言建立于程序lex.l中,然后lex.l通过Lex编译器,产生C语言程序lex.yy.c。程序lex.yy.c包括从lex.l的正规式构造出的转换图(用表格形式表示)和使用这张转换图识别词法单元的标准子程序。在lex.l中,和正规式相关联的动作是用C语言的代码表示的,它们被直接搬入lex.yy.c。最后,lex.yy.c被编译成目标程序a.out,它就是把输入串变成记号序列的词法分

析器。

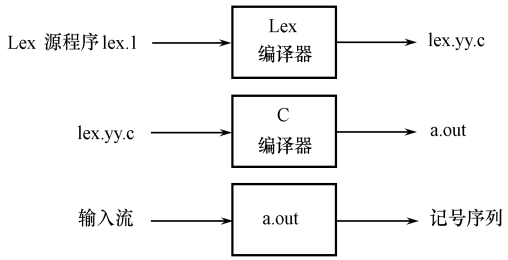


图 2.21 用 Lex 建立词法分析器

Lex 程序包括三个部分：

声明

%%

翻译规则

%%

辅助过程

声明部分包括变量声明、常量定义和正规定义。正规定义和 2.2 节中的方式类似，用于翻译规则中作为正规式。

Lex 程序的翻译规则是形如

p_1 {动作 1}

p_2 {动作 2}

… …

p_n {动作 n}

的语句。这里每个 p_i 是正规式，每个动作 i 是描述模式 p_i 匹配词法单元时词法分析器应执行的程序段。在 Lex 中，动作用 C 语言写，但是，一般来说它们可以用任何实现语言编写。

第三部分包括了动作所需要的辅助过程，这些过程也可以分别编译，然后在连接时装配在一起。

由 Lex 建立的词法分析器和语法分析器联系的方式是：词法分析器被语法分析器激活时，开始逐个字符地读它的剩余输入，直到它在剩余输入中发现能和正规式 p_i 匹配的最长前缀为止。然后执行动作 i。典型地，动作 i 将把控制返回语法分析器。如果不是这样，词法分析器就继续寻找下面的词法单元，直到有一个动作引起控制回到语法分析器为止。这种重复地搜索词法单元，直到显式地返回的方式，允许词法分析器方便地处理空白和注解。

词法分析器仅返回一个值（记号）给语法分析器，记号的属性值通过全程变量 yylval 传递。

例2.14 图2.22是识别表2.4中记号的Lex程序,此程序可体现出Lex的一些重要特征。

```
%{
  /* 常量LT, LE, EQ, NE, GT, GE, WHILE, DO, ID, NUMBER, RELOP的定义*/
%}
/* 正规定义 */
delim       [\t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?

%%
{ws}        {/* 没有动作,也不返回 */}
while       {return(WHILE);}
do          {return(DO);}
{id}        {yylval=install_id();return(ID);}
{number}    {yylval=install_num();return(NUMBER);}
"<"         {yylval=LT;return(RELOP);}
"<="        {yylval=LE;return(RELOP);}
"="         {yylval=EQ;return(RELOP);}
"<>"        {yylval=NE;return(RELOP);}
">"         {yylval=GT;return(RELOP);}
">="        {yylval=GE;return(RELOP);}
%%
install_id(){
    /* 把词法单元装入符号表并返回指针。
       yytext指向该词法单元的第一个字符,
       yyleng给出它的长度
    */
}
install_num(){
    /* 类似上面的过程,但词法单元不是标识符而是数 */
}
```

图2.22 识别表2.4记号的Lex程序

声明部分定义了一些在翻译规则中使用的常量,这些声明由特别的括号%{和%}包围。出现在括号中的任何东西都被直接抄写到词法分析器 lex.yy.c 中,不作为正规定义和翻译规则的一部分。在第三部分中的辅助过程也按同样方式处理。图 2.22 有两个过程 install_id 和 install_num,它们被抄入 lex.yy.c 中。

声明部分还包括一些正规定义,每个定义由一个名字和该名字指示的正规式组成。例如,第一个定义的名字是 delim,它代表字符类 [\t\n],也就是空格、制表(\t)和换行(\n)这三个字符中的任意一个。第二个是空白定义,由名字 ws 表示,空白是一个或多个分界符的序列。

在第五个定义 id 中使用了圆括号,它们是 Lex 的元符号,其含义是把一些正规式组成一个整体。同样地,竖线在 Lex 里表示"或者"。反斜线作为换码,让作为 Lex 元符号的字符有其自身原来的含义。数的正规定义中,十进制小数点由\.表示,因为 Lex 及许多处理正规式的 UNIX 程序中,点可以代表除了换行符以外的任何一个字符。在字符类 [+\-] 中,减号的前面放了反斜线,因为减号也是元符号,用于表示范围,如 [A-Z]。

还有别的方法可让作为元符号的字符表示原来的含义:把它们放在引号中。在翻译规则部分有这样的例子,6 个关系算符都由引号包围。

图中,第一个%%后面的翻译规则中,第一条规则表明,如果看见 ws,也就是空格、制表和换行符的串,没有任何动作发生,控制不返回分析器。词法分析器继续去识别记号,直至所识别记号的动作引起返回为止。注意,在 Lex 中,ws 必须由花括号包围,以区别由 5 个字母 ws 组成的模式,下面的 id 和 number 也是这样。

第二条规则表明,如果看见字母 while,则返回记号 WHILE,它是代表某个整数的常量,语法分析器把这个整数理解为 while。下面的一条规则以同样的方式处理 do。

在 id 的规则中,有关的动作含两个语句。第一个语句调用函数 install_id,该函数定义在第三部分,该函数的返回值赋给变量 yylval。变量 yylval 的定义出现在 Lex 的输出 lex.yy.c 中,它对分析器也是可用的,它的作用是保存返回记号的属性值,因为动作的第二个语句 return(ID) 只能返回记号的类别。

图中略去了 install_id 的详细代码,可以猜想,它在符号表中查找由模式 id 匹配的词法单元。注意,它和 2.2 节介绍的 install_id 的功能有区别,因为在这里保留字是用另外的正规式定义的。通过两个变量 yytext 和 yyleng,Lex 还使得词法单元的拼写对出现在第三部分的子程序是可用的,变量 yytext 是指针,它指向词法单元的开始字符,变量 yyleng 是整数,它指出词法单元的长度。

再下一条规则以类似方式处理数。最后 6 条规则都返回记号 relop,而用 yylval 的值来区别不同的关系符。

如果下一个要匹配的词法单元是 while,那么模式 while 和 {id} 都匹配这个词法单元,而

且这时它们都不能匹配更长的串,这时应该选哪一个呢?Lex 对这个冲突的解决是选择排在前面的模式。由于图 2.22 中保留字 while 的模式先于标识符的模式,因此该冲突的解决是选择排在前面的保留字。

如果要读的前两个字符是 <=,这时模式 < 匹配第一个字符,但它不是匹配输入中最长前缀的模式。由于 Lex 的策略是选择匹配某模式的最长前缀,这使解决 < 和 <= 的冲突变得容易,并且是按我们期望的方式选择 <= 作为下一个记号。

习 题 2

2.1 下列每种语言的输入字母表是什么?

(a) Pascal

(b) C

(c) Java

(d) Ada

2.2 在下面的各段程序中,按序列出所有的记号,并给每个记号以合理的属性值。

(a) Pascal

```
function max (i, j : integer) : integer;
 {return maximum of integer i and j}
begin
    if i > j then max := i
    else max := j
end;
```

(b) C

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

2.3 叙述由下列正规式描述的语言。

(e) 0 (0|1)* 0

(f) ((ε|0) 1)*

(g) (0|1)* 0 0|1) (0|1)

(h) 0* 10* 10* 10*

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

*2.4 为下列语言写正规定义：

(a) 包含5个元音的所有字母串，其中每个元音只出现一次且按顺序排列。

(b) 按词典序排列的所有字母串。

(c) C语言的注释，即以/* 开始和以 */结束的任意字符串，但它的任何前缀（本身除外）不以 */结尾。

(d) 相邻数字都不相同的所有数字串。

(e) 最多只有一处相邻数字相同的所有数字串。

(f) 由偶数个0和偶数个1构成的所有0和1的串。

(g) 由偶数个0和奇数个1构成的所有0和1的串。

(h) 所有不含子串011的0和1的串。

2.5 说明习题2.1中各种语言的数值常数的词法形式。

2.6 说明习题2.1中各种语言的标识符和关键字（或保留字）的词法形式。

2.7 用算法2.4为下列正规式构造非确定的有限自动机，给出它们处理输入串 *ababbab* 的状态转换序列。

(a) $(a|b)^*$

(b) $(a^*|b^*)^*$

(c) $((\varepsilon|a)b^*)^*$

(d) $(a|b)^*abb(a|b)^*$

2.8 用算法2.2把习题2.7的NFA变换成DFA。给出它们处理输入串 *ababbab* 的状态转换序列。

2.9 从表2.4中记号的转换图构造DFA。

2.10 C语言的注释是以/* 开始和以 */结束的任意字符串，但它的任何前缀（本身除外）不以 */结尾。画出接受这种注解的DFA的状态转换图。

2.11 我们可以从正规式的最简DFA同构来证明两个正规式等价。使用这种技术，证明下面的正规式等价。

(a) $(a|b)^*$

(b) $(a^*|b^*)^*$

(c) $((\varepsilon|a)b^*)^*$

2.12 为下列正规式构造最简的DFA。

(a) $(a|b)^*a(a|b)$

(b) $(a|b)^*a(a|b)(a|b)$

(c) $(a|b)^*a(a|b)(a|b)(a|b)$

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

2.14 构造一个 DFA,它接受 Σ = {0, 1} 上能被 5 整除的二进制数。

2.15 修改算法 2.4,使之尽可能少用 ε 转换,并保持所产生的 NFA 只有一个接受状态。

2.16 若 *L 是正规语言,证明下面的 L′语言也是正规语言。L′语言的定义是*

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

x^R 表示 x 的逆。

2.17 一个 C 语言编译器编译下面的函数时,报告 parse error 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(p,q)
long p,q;
{
    if (p%q == 0)
        /* then part */
        return q
    else
        /* else part */
    return gcd(q, p%q) ;
}
```

第3章　语 法 分 析

每种程序设计语言都有描述程序语法结构的规则。例如,Pascal 程序由程序块(又叫分程序)构成,程序块由语句组成,语句由表达式组成,表达式由记号组成,等等。这些规则可以用上下文无关文法或 BNF 范式(Backus－Naur Form)描述。

编译器常用的文法分析方法有自上而下和自下而上两种。正如它们的名字,自上而下分析器建立分析树是从根结点到叶结点,而自下而上分析器恰好反过来。它们的共同点是从左向右地扫描输入,每次一个符号。

最有效的自上而下和自下而上的分析法都只能处理上下文无关文法的子类。这些子类足以描述程序设计语言的大多数语法结构,其中 LL 文法的分析器通常用手工实现,而 LR 文法的分析器通常利用自动工具构造。

本章阐述编译器采用的典型语法分析方法。首先提出有关上下文无关文法的基本概念,然后介绍适合于手工实现的预测分析技术,最后给出自动工具用的 LR 分析算法。由于程序员准备的代码经常会出现一些语法错误,因此本章还扩展了所介绍的分析方法,使之能从常见的错误中恢复过来。

3.1　上下文无关文法

本节首先说明语法分析器(简称分析器)在编译器模型中的位置,然后介绍上下文无关文法。如图3.1所示,分析器读取词法分析器提供的记号流,检查它是否能由源语言的文法产生,输出分析树的某种表示。另外,我们希望该分析器能以易理解的形式报告任何语法错误,并从错误中恢复过来,使后面的分析能继续进行下去。

事实上,还有一些其他任务可能在分析时完成,例如把各种记号的信息收入符号表,完成类型检查和其他的语义检查,并产生中间代码。所有这些都包括在图3.1的"前端的其余部分"一框中,在下面三章将详细讨论它们。图3.1的虚线表示分析树是概念上的东西,并不一定真正生成。

3.1.1　上下文无关文法的定义

在第2章中用正规式定义了一些简单的语言,但是很多较复杂的语言不能用正规式表

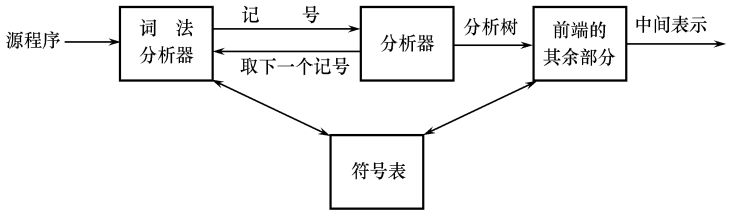


图3.1 分析器在编译器模型中的位置

达。例如,正规式不能用于描述配对或嵌套的结构,具体的例子有:由配对括号构成的串的集合不能用正规式描述,语句的嵌套结构也不能用正规式描述。还有,重复串也不能用正规式表示。例如,集合

$\{wcw \mid w$ 是 a 和 b 的串$\}$

不能用正规式描述。正规式只能表示给定结构的固定次数的重复或者没有指定次数的重复。

本节定义描述功能比正规式更强的上下文无关文法,并介绍一些与分析有关的术语。

形式上说,一个上下文无关文法 G 是一个四元组 (V_T, V_N, S, P),其中:

(1) V_T是一个非空有限集合,其元素称为终结符。在讨论程序设计语言的文法时,记号是终结符的同义词。

(2) V_N是一个非空有限集合,其元素称为非终结符,并有 $V_T \cap V_N = \varnothing$。在下面的例3.1中,expr 和 op 是非终结符。非终结符定义终结符串的集合,它们用来帮助定义由文法决定的语言。非终结符还强加层次结构于语言,这种层次结构对语法分析和翻译是有用的。

(3) S 是非终结符,称为开始符号,它定义的终结符串集就是文法定义的语言。

(4) P 是产生式的有限集合,每个产生式的形式是 $A \rightarrow \alpha$(有时用::=代替箭头),其中 $A \in V_N$,$\alpha \in (V_T \cup V_N)^*$。开始符号至少出现在某个产生式的左部。产生式指出了终结符和非终结符组成串的方式。

例3.1 文法 ({id, +, *, (,)}, {expr, op}, expr, P) 定义了有加、乘和一元减的算术表达式。P 由下列产生式组成:

$expr \rightarrow expr\ op\ expr$

$expr \rightarrow (expr)$

$expr \rightarrow -\ expr$

$expr \rightarrow$ id

$op \rightarrow +$

$op \rightarrow *$

为了表示上的简洁,在本书的剩余部分将采用下列约定来表示文法。

(1) 下列符号是终结符：

① 字母表中前面的小写字母，如 a,b,c；

② 黑体串，如 id 或 while；

③ 数字 0,1,…,9；

④ 标点符号，如括号，逗号等；

⑤ 运算符号，如+,-等。

(2) 下列符号是非终结符：

① 字母表中前面的大写字母，如 A,B,C；

② 字母 *S*，*并且它通常代表开始符号*；

③ 小写字母的名字，如 expr 和 stmt。

(3) 字母表中后面的大写字母，如 X,Y 和 Z，代表文法符号，即非终结符或终结符。

(4) 字母表中后面的小写字母，主要是 u，v，…，z，代表终结符号串。

(5) 小写希腊字母，例如 α,β 和 γ，代表文法的符号串。

(6) 如果 $A\rightarrow\alpha_1$，$A\rightarrow\alpha_2$，…，$A\rightarrow\alpha_k$ 是所有以 A 为左部的产生式（称它们为 A 产生式），则可以把它们写成 $A\rightarrow\alpha_1|\alpha_2|\cdots|\alpha_k$ 的形式，称 $\alpha_1,\alpha_2,\cdots,\alpha_k$ 是 A 的选择。

(7) 有了上面的这些约定，我们可以直接用产生式集合代替四元组来描述文法。此时，第一个产生式左部的符号是文法开始符号。

例 3.2 使用这些简写，例 3.1 的文法可以重新表示如下：

$$E\rightarrow EAE\mid(E)\mid-E\mid\mathrm{id}$$

$$A\rightarrow+\mid*$$

E 和 A 都是非终结符，其中 E 是开始符号，其余符号都是终结符。

3.1.2 推导

为描述文法定义的语言，需要使用推导的概念。推导的意思是，把产生式看成重写规则，把符号串中的非终结符用其产生式右部的串来代替。例如，对于下面的算术表达式文法

$$E\rightarrow E+E\mid E*E\mid(E)\mid-E\mid\mathrm{id} \tag{3.1}$$

产生式 $E\rightarrow E+E$ *意味着两个表达式相加仍然是表达式。这个产生式允许用* $E+E$ *代替* E 的任何出现，从简单的表达式产生更复杂一些的表达式。如果用 $E+E$ *代替单个* E，*这个*动作可以用式子

$$E\Rightarrow E+E$$

来描述，读做“E 推导出 $E+E$”。*产生式* $E\rightarrow(E)$ 表示 E 的任何出现可以用文法符号串 (E) 来代替，例如 $E*E\Rightarrow(E)*E$ *或* $E*E\Rightarrow E*(E)$。

从开始符号 E 开始，不断使用产生式，可以得到一个代换序列，如：

$E \Rightarrow E + E \Rightarrow id + E \Rightarrow id + id$

这个代换序列被称为从 E 到 id + id 的推导,这个推导表明了串 id + id 是表达式的实例。

抽象地说,如果 $A \to \gamma$ 是产生式,α 和 β 是文法的任意符号串,那么可以说 $\alpha A \beta \Rightarrow \alpha\gamma\beta$。如果 $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$,则说 α_1 推导出 α_n。符号 $\Rightarrow$ 表示"一步推导",符号 $\Rightarrow^*$ 可用于表示"零步或多步推导"。于是

(1) 对任何串有 $\alpha \Rightarrow^* \alpha$,并且

(2) 如果 $\alpha \Rightarrow^* \beta$,$\beta \Rightarrow \gamma$,那么 $\alpha \Rightarrow^* \gamma$。

类似地,我们用 $\Rightarrow^+$ 表示"一步或多步推导"。

对于开始符号为 S 的文法 G,可以用 $\Rightarrow^+$ 关系来定义 G 产生的语言 L(G),L(G) 的串仅包含 G 的终结符。我们说终结符号串 w 在 L(G) 中,当且仅当 $S \Rightarrow^+ w$,这时串 w 是语言 L(G) 的句子,也可以叫做文法 G 的句子。由上下文无关文法产生的语言叫做上下文无关语言。如果两个文法产生同样的语言,则称这两个文法等价。

如果 $S \Rightarrow^* \alpha$,α 可能含有非终结符,这时把 α 叫做 G 的句型。句子是只含终结符的句型。

例 3.3 串 $-(id + id)$ 是文法 (3.1) 的句子,因为存在着推导

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(id + E) \Rightarrow -(id + id) \tag{3.2}$$

出现在这个推导中的 E,$-E$,$-(E)$,…,$-(id + id)$ 都叫做这个文法的句型。

按推导长度进行归纳可以证明,对于文法 (3.1),其句子是由二元算符 + 和 *、一元算符 − 括号和运算对象 id 组成的算术表达式。反过来,按算术表达式长度进行归纳可以证明,这样的算术表达式都可以由此文法产生。于是文法 (3.1) 恰好产生这样的算术表达式集合。

如果在推导过程中出现的句型有两个或多个非终结符,那么就需要决定下一步推导代换哪个非终结符。例如,例 3.3 的推导在得到 $-(E + E)$ 后,可以如下进行:

$$-(E + E) \Rightarrow -(E + id) \Rightarrow -(id + id) \tag{3.3}$$

(3.3) 的每个非终结符代换时所用的右部和例 3.3 的一样,但有不同的代换次序。

为了理解某些分析器是怎样工作的,需要考虑每一步都是代换句型中最左边非终结符的推导,这样的推导叫做最左推导。若 $\alpha \Rightarrow \beta$ 是最左推导,可写成 $\alpha \Rightarrow_{lm} \beta$。推导 (3.2) 是最左推导,可以写成

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(id+E) \underset{lm}{\Rightarrow} -(id+id)$$

使用前面的约定,每步最左推导可写成 $wA\gamma \underset{lm}{\Rightarrow} w\delta\gamma$ 的形式,其中 w 只含终结符,$A \to \delta$ 是所用的产生式,γ 是文法的符号串。为了强调 α 最左地推导出 β,可写成 $\alpha \overset{*}{\underset{lm}{\Rightarrow}} \beta$。

类似地可以定义最右推导,即每步都代换最右边非终结符的推导,用 $\Rightarrow_{rm}$ 表示。最右推导又叫规范推导。

3.1.3 分析树

分析树是推导的图形表示。分析树的每个内部结点由非终结符标记,它的子结点由该非终结符的这次推导所用产生式的右部各符号从左到右依次标记。分析树的叶结点由非终结符或终结符标记,所有这些标记从左到右构成一个句型。例如,表达式 $-(id + id)$ 的最左推导的分析树(包括推导过程中的分析树)如图 3.2 所示。

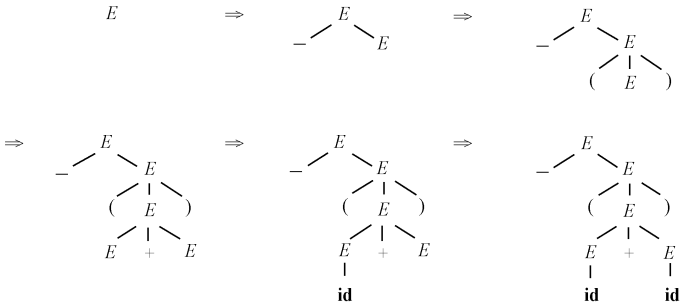


图 3.2 推导(3.2)的分析树

显然,表达式 $-(id + id)$ 的最左推导和最右推导的最终的分析树是一样的,也就是分析树忽略了不同的推导次序。不难看出,每棵分析树都有和它对应的最左推导和最右推导。

3.1.4 二义性

有的文法的一些句子存在不止一棵分析树,或者说这些句子存在不止一种最左(最右)推导。

例 3.4 考虑算术表达式文法(3.1),句子 $id * id + id$ 有两种不同的最左推导如下:

$$
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow id * E \\
&\Rightarrow id * E + E \\
&\Rightarrow id * id + E \\
&\Rightarrow id * id + id
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow E * E + E \\
&\Rightarrow id * E + E \\
&\Rightarrow id * id + E \\
&\Rightarrow id * id + id
\end{aligned}
$$

因而也有两棵不同的分析树,见图 3.3。

注意,图 3.3 右边的分析树反映了 + 和 * 通常的优先关系,而图 3.3 左边的分析树不是。也就是,习惯上 * 的优先级高于 +,因而表达式 $a * b + c$ 看成 $(a * b) + c$,而不是 $a * (b + c)$。

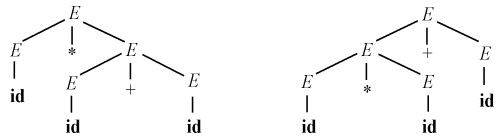


图3.3　id * id + id的两棵分析树

一个文法,如果存在某个句子有不止一棵分析树与之对应,那么称这个文法是二义的。也可以这么说,二义文法是存在某个句子有不止一个最左(最右)推导的文法。有些类型的分析器,它希望处理的文法是无二义的,否则它不能惟一确定对某个句子应选择哪棵分析树。出于某些需要,也可以构造允许二义文法的分析器,不过文法要带有消除二义性的规则,以便分析器扔掉不希望的分析树,为每个句子只留一棵分析树。

注意,文法二义并不代表语言一定是二义的。只有当产生一个语言的所有文法都是二义时,这个语言才称为二义的。

3.2　语言和文法

在语言(今后我们通常指程序设计语言)的设计和编译器的编写方面,文法都提供了很多的优点:

⑴ 文法为语言给出了精确的、易于理解的语法说明。

⑵ 对于某些文法类,可以自动产生高效的分析器。额外的好处是,分析器的自动构造过程可以揭示出语法的二义性和其他不属于该文法类的语法结构,这些问题在语言及其编译器的最初设计阶段很可能没有发现。

⑶ 设计得漂亮的文法可以给程序定义一些语法子结构,这些结构对于把源程序翻译成为正确的目标代码和错误诊断都是有用的。把以文法为基础的翻译描述变换成为相应程序的工具也是存在的。

⑷ 语言也是逐渐完善的,需要补充新的结构并完成新增的任务。如果存在以文法为基础的语言的实现,这些新结构的加入就更方便。

但是,必须注意,上下文无关文法只能描述程序设计语言的大部分语法而不是所有的语法。例如,对输入串的某些上下文有关的限制,如要求标识符的声明先于它们的使用,就不可能用上下文无关文法来描述。因此,语法分析后面的阶段必须分析语法分析器的输出,以保证输入串符合分析器无法检查的那些规则,这些事情通常在静态语义检查时完成。

本节首先考虑词法分析器和语法分析器的区别。由于每种分析方法只能处理某类文

法,因此有时需要改写文法,以便它对某种方法来说是可分析的。所以,本节还考虑一些文法的变换规则,以便产生适合于自上而下分析的文法。本节还讨论一些不能用上下文无关文法描述的语言结构,最后简单给出形式语言的小结。

3.2.1 正规式和上下文无关文法的比较

正规式可以描述的每种结构都能用上下文无关文法来描述。例如正规式 $(a|b)^*ab$ 和上下文无关文法

$$A_0 \to aA_0 \mid bA_0 \mid aA_1$$
$$A_1 \to bA_2$$
$$A_2 \to \varepsilon$$

描述同样的语言。

可以机械地把一个非确定的有限自动机变换成一个上下文无关文法,它产生的语言和这个自动机识别的语言相同。上述文法是从图 2.6 的 NFA 用下列规则构造的。首先确定终结符号集合,这是简单的。再为 NFA 的每个状态 i 引入非终结符 A_i,其中 A_0 是开始符号,因为 0 是开始状态。如果状态 i 有一个 a 转换到状态 j,引入产生式 $A_i \to aA_j$,如果是 ε 转换,则引入 $A_i \to A_j$。如果 i 是接受状态,再引入 $A_i \to \varepsilon$。

3.2.2 分离词法分析器的理由

既然正规集都是上下文无关语言,那么为什么要用正规式定义语言的词法?其理由如下:

⑴ 语言的词法规则非常简单,不必用功能更强的上下文无关文法描述它。

⑵ 对于词法记号,正规式给出的描述比上下文无关文法给出的描述更简洁且易于理解。

⑶ 从正规式自动构造出的词法分析器比从上下文无关文法构造出的更有效。

另一个问题是,为什么不把词法分析并入到语法分析中,直接从字符流进行语法分析,即把语言字母表上的字母作为语法分析的终结符。下面说明把词法分析从语法分析中分离出来的理由:

⑴ 简化设计是最重要的考虑。如果词法分析和语法分析合在一起,必须将语言的注解和空白的规则反映在文法中,这将使文法大大复杂。注解和空白由自己来处理的分析器,比注解和空格已由词法分析器删除的分析器要复杂得多。

⑵ 编译器的效率会改进。词法分析的分离可以简化词法分析器的设计,允许构造专门的和更有效的词法分析器。编译的相当一部分时间消耗在读源程序和把它分成一个个记号上,专门的读字符和处理记号的技术可以加快编译速度。

⑶ 编译器的可移植性加强。输入字符集的特殊性和其他与设备有关的不规则性可以

限制在词法分析器中,特殊的或非标准的符号的表示,如 Pascal 的↑,可以分离在词法分析器中处理。

(4) 把语言的语法结构分成词法和非词法两部分,为编译器前端的模块划分提供了方便的途径。

哪些应作为词法规则,哪些应作为语法规则,没有严格的准则。正规式是描述诸如标识符、常数和关键字等词法结构的最有力武器。上下文无关文法是描述括号配对、begin 和 end 配对、语句嵌套、表达式嵌套等结构的最有力武器,这些结构不可能用正规式来描述。

3.2.3 验证文法产生的语言

例3.5 考虑文法

$$S \rightarrow (S)S \mid \varepsilon \tag{3.4}$$

这个简单的文法产生所有配对的括号串,也只产生这样的串。为了明白这一点,首先证明 S 产生的每个句子都是配对的括号串,然后再证明任何配对括号串都可由 S 产生。前一个问题可以按推导步数进行归纳。对于归纳基础,可以看到,从 S 经一步推导能得到的终结符号串只有空串,它是配对的。

假定所有少于 n 步的推导都能产生配对的括号串,然后考虑 n 步的最左推导。这个推导必定是下面这种形式:

$$S \Rightarrow (S)S \overset{*}{\Rightarrow} (x)S \overset{*}{\Rightarrow} (x)y$$

由于从 S 到 x 和 y 的推导分别都少于 n 步,由归纳假设,x 和 y 都是配对括号串,所以串 (x)y *是配对括号串。*

下一步证明任何配对括号串都可由 S 产生,我们按串长进行归纳。对于归纳基础,空串是可以从 S 推导出的。

假定长度小于 $2n$ *的配对括号串都可以从 S 推导出来,然后考虑长度为* $2n$ $(n \geq 1)$ 的配对括号串 w。可以肯定,w 由左括号开始,令 (x) 是 w 的有相同个数的左括号和右括号的最短前缀,那么 w 可以写成 (x)y,其中 x 和 y 都是配对括号串,长度都小于 $2n$,*由归纳假设,* 它们都可以从 S 推导出来。这样,w 可以有如下的推导序列:

$$S \Rightarrow (S)S \overset{*}{\Rightarrow} (x)S \overset{*}{\Rightarrow} (x)y$$

从而证明了 w=(x)y 可以由 S 推导出来。

本课程并不要求掌握这样的证明技术。但是,当为一些例子语言设计文法时,如果能用这样的方式去思考问题的话,设计出正确文法的可能性就大得多。

3.2.4 适当的表达式文法

3.1 节构造的表达式文法有二义性,一个句子的不同分析树体现不同的算符优先关系

和算符结合性。下面构造非二义的有＋和＊运算的表达式文法，该文法和通常的算符优先关系和算符结合性对应。

设置两个非终结符 expr 和 term (expr 是开始符号)，用以表示不同层次的表达式和子表达式，再用非终结符 factor 来产生表达式的基本单位，基本单位有 id 和外加括号的表达式，即

factor→id | (*expr*)

再考虑两元算符＊，它们有较高的优先级，又是左结合的算符，因而产生式如下：

term→*term*＊ *factor* | *factor*

同样地，expr 产生由加法算符隔开的、左结合的 term 表，其产生式如下：

expr→ *expr*＋ *term* | *term*

这个表达式文法是无二义的。句子 id ＊ id ＊ id 和 id ＋ id ＊ id 的分析树如图 3.4 所示。

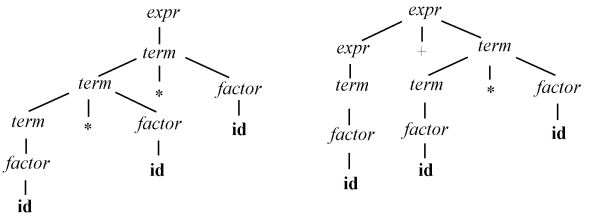


图 3.4 id ＊ id ＊ id 和 id ＋ id ＊ id 的分析树

上面两棵分析树所表现出的算符优先关系和结合性与通常的规定是一致的。可以看出，如果语言语义所规定的算符优先关系和结合性不是这样的话，我们的文法可能需要重新设计，否则所得到的分析树不能很方便地用于语义分析和中间代码生成等阶段。例如，如果规定＊和＋是右结合的运算，那么文法应该如下：

expr→ *term*＋ *expr* | *term*

term→ *factor*＊ *term* | *factor*

factor→id | (*expr*)

比较图 3.5 和图 3.4 的分析树，应该不难看出它们的区别。

3.2.5 消除二义性

从上一小节的例子可以看到，有些二义文法可以通过重写文法而消除二义性。再举一个例子，消除下面“悬空 else”文法的二义性：

stmt → if *expr* then *stmt*

| if *expr* then *stmt* else *stmt* (3.5)

| other

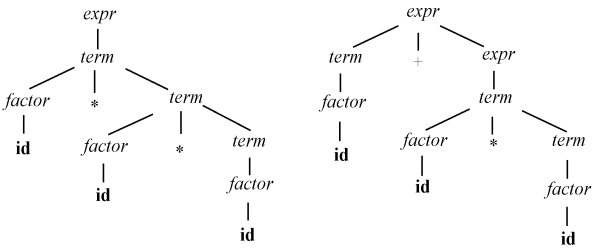


图3.5 文法修改后的分析树

这里的other代表任何其他语句。按照这个文法,形式为

if *expr* then if *expr* then *stmt* else *stmt* (3.6)

的复合条件语句有两个最左推导:

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

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

因此文法(3.5)是二义的。

所有含这种条件语句的语言都使用前一种最左推导,因为它和这些语言所采用的规则"每个else和前面最接近的还没有配对的then相配对"是一致的。这条规则可以直接体现在文法中,例如可以把文法(3.5)改写成下面无二义的文法。想法是这样,出现在then和else之间的语句必须是"配对"的,配对语句是指那些不含不配对语句的if-then-else语句,还有那些不是条件语句的语句。于是,可以使用文法

$$\begin{aligned} &stmt \rightarrow matched\_stmt \\ &\quad\ \mid unmatched\_stmt \\ &matched\_stmt \rightarrow \text{if } expr \text{ then } matched\_stmt \text{ else } matched\_stmt \\ &\quad\ \mid \text{other} \\ &unmatched\_stmt \rightarrow \text{if } expr \text{ then } stmt \\ &\quad\ \mid \text{if } expr \text{ then } matched\_stmt \text{ else } unmatched\_stmt \end{aligned} \tag{3.7}$$

该文法和文法(3.5)产生同样的串集,但是对于句型(3.6)只允许一种最左推导。注意,*unmatched_stmt的第二个产生式的右部* if *expr* then *matched_stmt* else *unmatched_stmt* 的*matched_stmt和unmatched_stmt是不能对调的,否则仍然是二义的。*

也许你会问,为什么各种程序设计语言都不用无二义的文法(3.7),而用二义文法(3.5)。这是因为,文法(3.7)失去了简洁性。定义语言语法的文法有二义性并不可怕,只要有消除二义性的规则就可以了。

3.2.6 消除左递归

一个文法是左递归的，如果它有非终结符 A，对某个串 α，存在推导 $A \Rightarrow^{+} A\alpha$。自上而下的分析方法不能用于左递归文法，因此需要消除左递归。形式为 $A \rightarrow A\alpha$ 的产生式引起的左递归称为直接左递归。

左递归产生式 $A \rightarrow A\alpha \mid \beta$，可以用非左递归的

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

来代替，它们没有改变从 A 推导出的串集。

例 3.6 考虑下面的算术表达式文法

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \mathrm{id}$$

消除 E 和 T 的直接左递归，可以得到

$$E \rightarrow TE'$$
$$E' \rightarrow + TE' \mid \varepsilon$$
$$T \rightarrow FT' \tag{3.8}$$
$$T' \rightarrow * FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid \mathrm{id}$$

不管有多少 A 产生式，都可以用下面的技术消除直接左递归。首先把 A 产生式组合在一起：

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

其中 β_i 都不以 A 开始，α_i 都非空，然后用

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \varepsilon$$

代替 A 产生式。这些产生式和前面的产生式产生一样的串集，但是不再有左递归。这个过程可删除直接左递归，但不能消除两步或多步推导形成的左递归。例如，考虑文法

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Sd \mid \varepsilon$$

其中非终结符 S 是左递归的，因为 $S \Rightarrow Aa \Rightarrow Sda$，但它不是直接左递归的。用 S 产生式代换 $A \rightarrow Sd$ 中的 S，可以得到下面的文法：

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Aad \mid bd \mid \varepsilon$$

删除其中的直接左递归,得到如下的文法:

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid A'$

$A' \rightarrow adA' \mid \varepsilon$

由此可见,写一个删除文法左递归的算法并不是件困难的事情。

3.2.7 提左因子

提左因子也是一种文法变换,它用于产生适合于自上而下分析的文法。在自上而下的分析中,当不清楚应该用非终结符 A 的哪个选择来替换它时,可以通过重写 A 产生式来推迟这种决定,推迟到看见足够多的输入,能帮助正确决定所需选择为止。

例如,条件语句有两个产生式:

$stmt \rightarrow$ if $expr$ then $stmt$ else $stmt$

| if $expr$ then $stmt$

当看见输入记号 if 时,不能马上确定用哪个产生式来扩展 stmt。

一般来说,如果 $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ 是 A 的两个产生式,输入串的前缀是从 α 推导出的非空串时,我们不知道是用 $\alpha\beta_1$ 还是用 $\alpha\beta_2$ 来扩展 A。但是可以通过先扩展 A 到 $\alpha A'$ 来推迟这个决定。然后,看完了从 α 推出的输入后,再扩展 A' 到 β_1 或 β_2。这就是提左因子,原来的产生式成为:

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$

例 3.7 对于悬空 else 的文法

$stmt \rightarrow$ if $expr$ then $stmt$ else $stmt$

| if $expr$ then $stmt$

| other

提左因子后的文法成为:

$stmt \rightarrow$ if $expr$ then $stmt$ $optional\_else\_part$

| other

$optional\_else\_part \rightarrow$ else $stmt$

| ε

这样,如果输入的第一个记号是 if,那么扩展 stmt 到 if $expr$ then $stmt$ $optional\_else\_part$,等到 if $expr$ then $stmt$ 都看见后,再决定扩展 $optional\_else\_part$ 到 else $stmt$ 还是到 ε。

3.2.8 非上下文无关的语言结构

在很多程序设计语言中,仅用上下文无关文法难以完成其中一些语法结构的说明。本节将给出一些这样的结构,并用简单的抽象语言来说明其中的困难。

例3.8 考虑抽象语言 $L_1 = \{wcw \mid w$ 属于 $(a|b)^*\}$。L_1 的句子的特点是,其前后是由 a 和 b 组成的相同的串,中间由 c 把它们隔开,例如 aabcaab。这个抽象语言是程序中标识符的声明应先于其引用的抽象,wcw 中的第一个 w 代表标识符 w 的声明,第二个代表它的引用。可以证明该语言不是上下文无关语言,但是这个证明超出了本书的范围。这个例子意味着 C 和 Pascal 都不是上下文无关语言,因为它们都要求标识符的声明先于引用,并且允许标识符任意长。

由于这一点,描述这些语言语法的文法只是用 id 这样的记号来代表所有的标识符,而在这些语言的编译器中,由语义分析阶段检查标识符的声明必须先于引用。

例3.9 语言 $L_2 = \{a^n b^m c^n d^m \mid n \geq 0, m \geq 0\}$ 不是上下文无关语言。L_2 是正规式 $a^* b^* c^* d^*$ 所表示的语言的子集,其要求是 a 和 c 的个数相等,b 和 d 的个数相等。它是过程声明的形参个数和过程引用的实参个数应该相同的问题的抽象,a^n 和 b^m 代表两个过程声明的形参表中分别有 n 和 m 个参数,c^n 和 d^m 分别代表这两个过程调用的实参表。

语言中过程声明和引用的语法并不涉及到参数的个数。例如 FORTRAN 的 CALL 语句可描述为:

$$stmt \rightarrow call\ id\ (expr\_list)$$
$$expr\_list \rightarrow expr\_list,\ expr \mid expr$$

实参和形参个数的一致性检查也是放在语义分析阶段完成。

例3.10 语言 $L_3 = \{a^n b^n c^n \mid n \geq 0\}$ 也不是上下文无关语言,它是 $L(a^* b^* c^*)$ 中 a、b 和 c 个数相等的串。它是早先排版描述的一个现象的抽象。

有趣的是,有些类似于 L_1,L_2 或 L_3 的语言却是上下文无关的。例如 $L_1' = \{wcw^R \mid w \in (a|b)^*\}$ 是上下文无关的,其中 w^R 代表逆序的 w,它可由下面的文法产生:

$$S \rightarrow aSa \mid bSb \mid c$$

语言 $L_2' = \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$ 是上下文无关的,它可由下面的文法产生:

$$S \rightarrow aSd \mid aAd$$
$$A \rightarrow bAc \mid bc$$

此外,$L_2'' = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\}$ 也是上下文无关的,文法是

$$S \rightarrow AB$$

$A \rightarrow aAb \mid ab$

$B \rightarrow cBd \mid cd$

最后，$L_3' = \{a^n b^n \mid n \geq 1\}$ 也是上下文无关的，文法是

$S \rightarrow aSb \mid ab$

值得注意的是，L_3' 是不能用正规式描述的语言的一个范例。我们可以证明这一点，假定 L_3' 可以由某个正规式描述，那么我们就可以构造一个 DFA D，它接受 L_3'。D 的状态数必定有限，设为 k，设 D 读完 ε，a，aa，…，a^k 分别到达状态 s_0，s_1，…，s_k，也就是 D 读 i 个 a 后到达状态 s_i。

因为 D 只有 k 个不同的状态，那么在序列 s_0，s_1，…，s_k 中至少有两个状态相同，例如是 s_i 和 s_j。从状态 s_i 出发，D 可以接受 i 个 b 到达一个接受状态 f，因为 $a^i b^i$ 属于 L_3'。这样，D 还存在着一条从开始状态 s_0 到 s_i 再到 f 的路径，该路径的标记为 $a^j b^i$，如图 3.6 所示。于是，D 也接受 $a^j b^i$，但它不在 L_3' 中，这和 D 接受的语言是 L_3' 的假设矛盾。

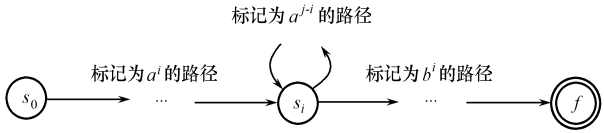


图 3.6 接受 $a^j b^i$ 和 $a^j b^i$ 的 DFA D

通俗地说，有限自动机不能计数，即有限自动机不能接受像 L_3' 这样的语言，它要求在接受 b 以前数出 a 的个数。类似地，上下文无关文法可以计两项的数，但不能计三项的数，即可用它定义 L_3'，但不能定义 L_3。

3.2.9 形式语言鸟瞰

是否可以用功能更强的文法描述上面那些不能用上下文无关文法描述的语言呢？回答是肯定的。

乔姆斯基 (Chomsky) 把文法分成四种类型，即 0 型、1 型、2 型和 3 型，0 型的描述能力强于 1 型，1 型的强于 2 型，2 型的强于 3 型。这几类文法的差别在于对产生式施加不同的限制。

我们说 $G = (V_T, V_N, S, P)$ 是 0 型文法，如果它的每个产生式

$\alpha \rightarrow \beta$

是这样的结构：$\alpha \in (V_N \cup V_T)^*$，且至少含一个非终结符，而 $\beta \in (V_N \cup V_T)^*$。

0 型文法也叫短语文法。一个非常重要的理论结果是，0 型文法的能力相当于图灵机。或者说，任何 0 型语言都是递归可枚举的，反之，递归可枚举集也必定是一个 0 型语言。

如果对0型文法加上以下第i条限制,就可以得到i型文法:

(1) G的任何产生式$\alpha\to\beta$都满足$|\alpha|\le|\beta|$(我们用$|x|$表示x中符号的个数)。只有$S\to\varepsilon$可以例外,但此时S不得出现在任何产生式的右部。

(2) G的任何产生式为$A\to\beta$的形式,$A\in V_N$,$\beta\in(V_N\cup V_T)^*$。

(3) G的任何产生式为$A\to aB$或$A\to a$的形式,$A,B\in V_N$,$a\in V_T$。

1型文法也叫上下文有关文法。这种文法意味着对非终结符的替换需要考虑上下文,并且一般不允许换成ε串。例如,若$\alpha A\beta\to\alpha\gamma\beta$是1型文法的产生式,且α和β不都为空,则非终结符A只有在α和β这样的上下文环境下才可以替换成γ。

2型文法也就是上下文无关文法,非终结符的替换可以不必考虑上下文。

3型文法等价于正规式,因而也叫正规文法。

前面提到的语言$L_3=\{a^nb^nc^n\mid n\ge1\}$可以用上下文有关文法来定义,其产生式如下:

$S\to aSBC$　　$bB\to bb$

$S\to aBC$　　$bC\to bc$

$CB\to BC$　　$cC\to cc$

$aB\to ab$

$a^nb^nc^n$的推导过程如下:

(1) $S\to aSBC$用$n-1$次得到$S]^*a^{n-1}S(BC)^{n-1}$;

(2) $S\to aBC$用1次得到$S]^+a^n(BC)^n$;

(3) 用产生式$CB\to BC$交换相邻的CB,得到$S]^+a^nB^nC^n$;

(4) $aB\to ab$用1次得到$S]^+a^nbB^{n-1}C^n$;

(5) $bB\to bb$用$n-1$次得到$S]^+a^nb^nC^n$;

(6) $bC\to bc$用1次得到$S]^+a^nb^ncC^{n-1}$;

(7) $cC\to cc$用$n-1$次得到$S]^+a^nb^nc^n$。

由此可见,上下文有关文法的能力强于上下文无关文法。

自乔姆斯基于1956年建立形式语言的描述以来,形式语言的理论发展得很快。这种理论对计算机科学有着深刻的影响,特别是对程序设计语言的设计、编译方法和计算复杂性等方面更有重大作用。有兴趣的读者可以阅读有关形式语言和自动机理论方面的书籍。

由于下面只涉及上下文无关文法,因而把它简称为文法。

3.3 自上而下分析

本节首先介绍自上而下分析的基本概念和一般方法,然后定义适合于自上而下分析的

LL(1)文法,再介绍一些实用的自上而下分析方法及分析表的自动生成,最后还要讨论自上而下的错误恢复。

3.3.1 自上而下分析的一般方法

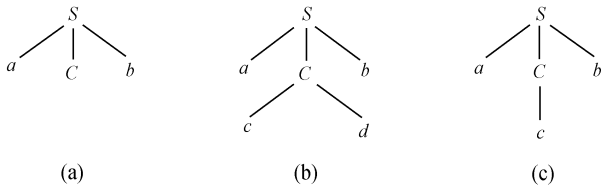
自上而下分析的宗旨是,对任何输入串,试图用一切可能的办法,从文法开始符号(根结点)出发,自上而下,从左到右地为输入串建立分析树。或者说,为输入串寻找最左推导。这种分析过程本质上是一种试探过程,是反复使用不同的产生式谋求匹配输入串的过程。

例3.11 若有文法

$S \rightarrow aCb$

$C \rightarrow cd \mid c$

为了自上而下地为输入串 w=*acb*建立分析树,首先建立只有标记为 S 的单个结点树,输入指针指向 w第一个符号 a。然后用 S 的第一个产生式来扩展该树,得到的树如图3.7(a)所示。



(a) (b) (c)

图3.7 自上而下分析的试探过程

最左边的叶子标记为 a,匹配 w的第一个符号。于是,推进输入指针到 w的第二个符号c,并考虑分析树上下一个叶子 C,它是非终结符。用 C的第一个选择来扩展C,得到图3.7(b)的树。现在第二个输入符号 c能匹配,再推进输入指针到 b,把它和分析树上的下一个叶子 d比较。因为 b和 d不匹配,回到 C,看它是否还有别的选择尚未尝试。

在回到 C时,必须重置输入指针于第二个符号,即第一次进入 C的位置。现在尝试 C的第二个选择,得到图3.7(c)的分析树。叶子 c匹配w的第二个符号,叶子 b匹配w的第三个符号。这样,得到了 w的分析树,从而宣告分析完全成功。

上述这种自上而下分析法存在困难和缺点。首先,如果存在非终结符 A,并且有

$A \Rightarrow^{+} Aa$

这样的左递归,那么文法将使上述自上而下分析过程陷入无限循环。因为当试图用 A去匹配输入串时会发现,在没有吃进任何输入符号的情况下,又得要求用下一个 A去进行新的匹配。因此,使用自上而下分析法时,文法应该没有左递归。

其次,当非终结符用某个选择匹配成功时,这种成功可能仅是暂时的。由于这种虚假现象,需要使用复杂的回溯技术。

第三,由于回溯,需要把已做的一些语义工作(指中间代码的生成和各种表格的簿记)推倒重来。这些事情既麻烦又费时间,所以最好设法消除回溯。

第四,回溯使得分析器很难报告输入串出错的确切位置。

最后,试探与回溯是一种穷尽一切可能的办法,效率低,代价高,它只有理论意义,在实践中的价值不大。

3.3.2 LL(1)文法

为构造不带回溯的自上而下分析算法,首先要消除文法的左递归,并找出克服回溯的充分必要条件。消除左递归的方法已介绍了,下面讨论如何克服回溯。

假设对文法的任何非终结符,当要用它去匹配输入串时,我们能够根据所面临的输入符号准确地指派它的一个选择去执行任务。这个准确是指:若此选择匹配成功,那么这种匹配决不是虚假的;若此选择无法完成匹配任务,则任何其他的选择也肯定无法完成。如果能做到这一点,那么回溯肯定能消除。

在讨论不得回溯的前提对文法有什么限定之前,先定义两个和文法有关的函数。一个文法的符号串 α 的开始符号集合 FIRST(α) 是

$$FIRST(\alpha) = \{a \mid \alpha \Rightarrow^* a\cdots, a \in V_T\}$$

特别是,$\alpha \Rightarrow^* \varepsilon$ 时,规定 $\varepsilon \in FIRST(\alpha)$。*如果非终结符 A 的所有选择的开始符号集合两两不相交*,即对 A 的任何两个不同的选择 α_i 和 α_j,有

$$FIRST(\alpha_i) \cap FIRST(\alpha_j) = \varnothing$$

那么,当要求 A 匹配输入串时,A 就能根据它所面临的第一个输入符号 a,准确地指派某一个选择前去执行任务。这个选择就是那个开始符号集合含 a 的 α。把一个文法改造成任何非终结符所有选择的开始符号集合两两不相交的办法是提取左因子,这种方法已经介绍过。

如果 ε 属于 A 的某个选择的开始符号集合,那么问题就比较复杂,需要定义文法非终结符的后继符号集合后才能解释。非终结符 A 的后继符号集合 FOLLOW(A) 是所有在句型中可以直接出现在 A 后面的终结符的集合,也就是

$$FOLLOW(A) = \{a \mid S \Rightarrow^* \cdots Aa\cdots, a \in V_T\}$$

此外,如果 A 是某个句型的最右符号,那么 \$ 属于 FOLLOW(A)。

例 3.12 考虑(3.8)的文法,把它重复如下:

$$E \to TE'$$

$$E' \to +TE' \mid \varepsilon$$

$T \rightarrow FT'$

$T' \rightarrow * FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

那么

$FIRST(E) = FIRST(T) = FIRST(F) = \{(, id\}$

$FIRST(E') = \{+, \varepsilon\}$

$FIRST(T') = \{*, \varepsilon\}$

有了上面这些 FIRST 集合后,就不难计算各产生式右部的 FIRST 集合了。例如,对于 $T \rightarrow FT'$, $FIRST(FT') = FIRST(F) = \{(, id\}$。

再看 FOLLOW 集合。这里要注意的是,如果有产生式 $A \rightarrow \alpha B$ 或 $A \rightarrow \alpha B\beta$ 且 $\beta \Rightarrow^* \varepsilon$,那么 FOLLOW(A) 的一切元素都要加入 FOLLOW(B) 中。

$FOLLOW(E) = FOLLOW(E') = \{), \$\}$

$FOLLOW(T) = FOLLOW(T') = \{+,), \$\}$

$FOLLOW(F) = \{+, *,), \$\}$

设计一个算法来计算开始符号集合和后继符号集合是件简单的事情。

下面回到 ε 属于 A 的某个选择 β 的开始符号集合这个问题。如果 a 属于 A 的另一个选择 α 的开始符号集合,并且 a 属于 FOLLOW(A),那么当面临 a 为 A 做选择时,选择 a 和 β 都是有理由的,其中选择后者的理由是让 β 推出空串,把这个 a 看成是 A 的后继符号。

这样,要想不出现回溯,需要文法的任何两个产生式 $A \rightarrow \alpha \mid \beta$ 都满足下面两个条件:

(1) $FIRST(\alpha) \cap FIRST(\beta) = \varnothing$;

(2) 若 $\beta \Rightarrow^* \varepsilon$,那么 $FIRST(\alpha) \cap FOLLOW(A) = \varnothing$。

把满足这两个条件的文法叫做 LL(1) 文法,其中的第一个 L 代表从左向右扫描输入,第二个 L 表示产生最左推导,1 代表在决定分析器的每步动作时向前看一个输入符号。除了没有公共左因子外,LL(1) 文法还有一些明显的性质,它不是二义的,也不含左递归。还有一些性质将在 3.3.5 节构造分析表时再说。很明显,例 3.8 的表达式文法是 LL(1) 的。

3.3.3 递归下降的预测分析

所谓预测分析是指能根据当前的输入符号为非终结符确定用哪一个选择,LL(1) 文法是满足这个要求的。递归下降的预测分析是指为每一个非终结符写一个分析过程,由于文法的定义是递归的,因此这些过程也是递归的。另外,在处理输入串时,首先执行的是对应开始符号的过程,然后根据产生式的右部出现的非终结符,依次调用相应的过程,这种逐步下降的过程调用序列隐含地定义了输入的分析树。

我们还是通过一个例子来说明如何构造递归下降的预测分析程序。

下面的文法产生 Pascal 类型的子集,用记号 dotdot 表示“..”以强调这个字符序列作为一个词法单元。

type → *simple*
| ↑ id
| array [simple] of *type*
simple → integer
| char
| num dotdot num

显然,该文法是 LL(1) 的。

图 3.8 是上面类型定义文法的递归下降预测分析器。这个分析器包括非终结符 type 和 simple 的过程以及附加的过程 match。使用 match 是为了简化 type 和 simple 的代码,如果它的参数匹配当前的符号,它就调用函数 nexttoken,取下一个记号,并改变变量 lookahead 的值。

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

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

图 3.8 预测分析器的代码

3.3.4 非递归的预测分析

如果显式地维持一个状态栈,而不是隐式地通过递归调用,那么可以构造非递归的预测分析器。预测分析的关键问题是决定取哪个产生式运用于非终结符,图 3.9 的非递归分析器通过查分析表来决定产生式。

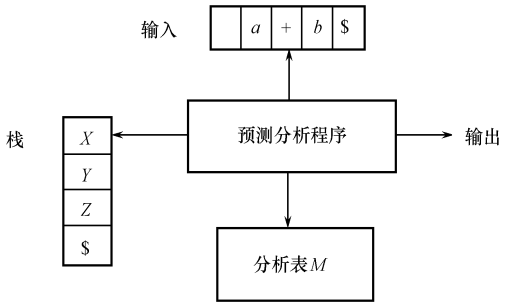


图 3.9 非递归的预测分析器的模型

表驱动的预测分析器有一个输入缓冲区、一个栈、一张分析表和一个输出流。输入缓冲区包含要分析的串,后面跟一个符号$,它是输入串的结束标记。栈中存放文法的符号串,栈底符号是$。初始时,栈中含文法的开始符,它在$的上面。分析表是一个两维数组 M[A, a], A是非终结符, a是终结符或$。

现在说明这个分析器的工作过程。预测分析程序根据当前的栈顶符号 X 和输入符号 a 决定分析器的动作,它有四种可能:

(1) 如果 X= a =$, 分析器宣告分析完全成功而停机。

(2) 如果 X= a≠$,分析器弹出栈顶符号 X,并推进输入指针,指向下一个符号。

(3) 如果 X 是终结符但不是 a,则分析器报告出错,调用错误恢复例程。

(4) 如果 X 是非终结符,程序访问分析表 M,若 M[X, a] 是 X 的产生式,例如,M[X,

$a]=\{X\to UW\}$,那么分析器用 WU 代替栈顶的 X,并让 U 在栈顶。作为输出,假定分析器在这里打印出所用的产生式,当然也可以执行其他代码。如果 $M[X,a]$ 指示出错,则分析器调用错误恢复例程。

算法 3.1 非递归的预测分析。

输入 串 w 和文法 G 的分析表 M。

输出 如果 w 属于 $L(G)$,则输出 w 的最左推导,否则报告错误。

方法 初始时分析器的格局是 $\$S$ 在栈里,其中 S 是开始符号并且在栈顶;$w\$$ 在输入缓冲区,图 3.10 是用预测分析表 M 对输入串进行分析的程序。

```
让 ip 指向 w$ 的第一个符号;
repeat
    令 X 等于栈顶符号,并且 a 等于 ip 指向的符号;
    if X 是终结符或$ then
        if X=a then
            把 X 从栈顶弹出并推进 ip
        else error()
    else if M[X,a]=X→Y1Y2…Yk then begin        /* X 是非终结符 */
        从栈中弹出 X;
        把 Yk,Yk-1,…,Y1 依次压入栈,Y1 在栈顶;
        输出产生式 X→Y1Y2…Yk
    end
    else error()
until X=$/* 栈空 */
```

图 3.10 预测分析程序

例 3.13 考虑文法 (3.8),该文法的预测分析表见表 3.1,表中空白表示出错,非空白指示一个产生式,用来替换栈顶的非终结符。

表 3.1 文法 (3.8) 的分析表

非终结符	输入符号					
	id	+	*	(	)	$
E	$E\to TE'$			$E\to TE'$		
E'		$E'\to +TE'$			$E'\to\varepsilon$	$E'\to\varepsilon$
T	$T\to FT'$			$T\to FT'$		
T'		$T'\to\varepsilon$	$T'\to *FT'$		$T'\to\varepsilon$	$T'\to\varepsilon$
F	$F\to$ id			$F\to(E)$		

如果输入是 id ＊ id ＋ id,分析过程中各部分的变化则如表 3.2。输入指针指在输入串最左边的符号。仔细观察分析器的动作可知,分析器跟踪的是输入的最左推导,也就是输出最左推导的那些产生式。已经扫描过的符号加上栈中的文法符号(从顶到底),构成最左推导的句型。

表 3.2 预测分析器接受输入 id ＊ id ＋ id 的动作

栈	输入	输出
$*E*	id ＊ id ＋ id $	
$*E′T*	id ＊ id ＋ id $	*E*→ *TE′*
$*E′T′F*	id ＊ id ＋ id $	*T*→ *FT′*
$*E′T′* id	id ＊ id ＋ id $	*F*→id
$*E′T′*	＊id ＋ id $	
$*E′T′F*＊	＊ id ＋ id $	*T′*→＊ *FT′*
$*E′T′F*	id ＋ id $	
$*E′T′* id	id ＋ id $	*F*→id
$*E′T′*	＋ id $	
$*E′*	＋ id $	*T′*→ ε
$*E′T*＋	＋ id $	*E′*→＋ *TE′*
$*E′T*	id $	
$*E′T′F*	id $	*T*→ *FT′*
$*E′T′*id	id $	*F*→id
$*E′T′*	$	
$*E′*	$	*T′*→ ε
$	$	*E′*→ ε

3.3.5 构造预测分析表

对于非递归的预测分析来说,剩下的问题是如何构造预测分析表。下面的算法为文法 *G* 构造预测分析表,这个算法的思想如下:如果 *A*→ α 是产生式且 a 在 FIRST(α) 中,那么当前输入符号为 a 时,分析器用 α 展开 A。惟一的复杂情况是 α] ＊ ε,在这种情况下,如果当前输入符号(包括$)在 FOLLOW(A) 中,仍应用 α 展开 A。

算法 3.2 构造预测分析表。

输入 文法 *G*。

输出 分析表 *M*。

方法 (1) 对文法的每个产生式 *A*→ α,执行 (2) 和 (3)。

(2) 对 FIRST(α)的每个终结符 a,把 A→α加入 M[A, a]。

(3) 如果 ε在 FIRST(α)中,对 FOLLOW(A)的每个终结符 b(包括$),把 A→α加入 M[A, b]。

(4) M 的其他没有定义的条目都是 error。

例 3.14 把算法 3.2 用于文法(3.8)。因为 FIRST(TE′) = FIRST(T) = {(, id},因此产生式 E→TE′使得 M[E, (]和 M[E, id]含产生式 E→TE′。

产生式 E′→+TE′使 M[E′, +]含产生式 E′→+TE′。因为 FOLLOW(E′) = {), $},产生式 E′→ε使 M[E′,)]和 M[E′, $]含产生式 E′→ε。

算法 3.2 作用于文法(3.8)产生的分析表见表 3.1。

算法 3.2 可用于任何文法 G 来产生分析表 M。然而对某些文法,M 可能含有一些多重定义的条目。例如 G 左递归或二义的话,M 至少含一个多重定义的条目。

例 3.15 以例 3.7 的条件语句为例,为方便起见,将文法重写如下:

$$\begin{aligned} &stmt \rightarrow \text{if } expr \text{ then } stmt\ e\_part \mid \text{other} \quad /* \ e\_part \text{ 指 optional\_else\_part } */ \\ &e\_part \rightarrow \text{else } stmt \mid \varepsilon \\ &E \rightarrow b \end{aligned} \tag{3.9}$$

它的分析表见表 3.3。

M[e_part, else]条目包括 e_part→else stmt 和 e_part→ε,因为 FOLLOW(e_part) = {else, $}。这个文法是二义的,即看见 else 时使用哪个产生式。可以只选择 e_part→else stmt 来解决这种二义性,这个选择刚好满足 else 和最接近的 then 配对这个约定。

表 3.3 文法(3.9)的预测分析表

非终结符	输入符号					
	other	b	else	if	then	$
stmt	stmt→other			stmt→if...		
e_part			e_part→else stmt e_part→ε			e_part→ε
expr		expr→b				

可以证明,一个文法的预测分析表没有多重定义的条目,当且仅当该文法是 LL(1)的。还可以证明,算法 3.2 为 LL(1)文法 G 产生的分析表能分析 L(G)的所有句子,也仅能分析 L(G)的句子。

剩下的一个问题是,当分析表有多重定义的条目时应该怎么办。一种办法是求助于文法变换,消除左递归和提取所有可能的左因子,以期得到的新文法的分析表没有多重定义的条目。遗憾的是,有些文法不论怎么变化也不能产生 LL(1)文法,文法(3.9)是一个这样的

例子,它的语言没有 LL(1) 文法。正如我们所看见的,让 M[e_part, else] = {e_part→else stmt},仍可以用预测分析器对(3.9)进行分析。一般来说,没有一个普遍适合的规则可用来删除多重定义的条目使其成为单值而不影响分析器识别的语言。

使用预测分析的主要困难在于为源语言写一个能构造出预测分析器的文法。虽然左递归的消除和提左因子是简单的,但它们使得结果文法难读并且不易用于翻译。3.5 节介绍的 *LR 分析器可以克服这些问题。*

3.3.6 预测分析的错误恢复

在讨论预测分析的错误恢复前,先对编译器的错误处理做一个概述。

如果编译器只处理正确的程序,它的设计和实现可以大大简化,但是程序员往往不是一次就能把程序写正确的,好的编译器应能帮助程序员识别和定位错误。虽然错误是那样容易发生,但是几乎所有程序设计语言的规范没有描述编译器应该怎样处理语法错误。

在设计编译器时,如果从一开始就计划错误处理,那么就可以简化编译器的结构,并且改进它对错误的响应。

大多数的程序错误可以简单加以归类:30%是标点符号错,20%是算符或运算对象错,15%是关键字错,剩下 5%是其他类型的错误。大多数的标点符号错是属于分号使用不正确。分号错误出现如此普遍的一个原因是各种语言的分号使用有很大区别,Pascal 语言把分号作为语句的分隔符,而 C 语言把分号作为语句的结束符。后一种使用较少出错。另一原因是分号除了上面的这个作用外,不再有其他的含义。算符错误的典型例子是忽略了 := 的冒号。关键字拼写错很少出现。

程序错误又可以根据性质来区分。例如,错误可能是:

(1) 词法错误,如标识符、关键字或算符的拼写错;

(2) 语法错误,如算术表达式的括号不配对;

(3) 语义错误,如算符作用于不相容的运算对象;

(4) 逻辑错误,如无穷的递归调用。

大多数错误的诊断和恢复集中在语法分析阶段,一个原因是大多数错误是语法错误,另一原因是现代分析方法的准确性,它们能以非常有效的方法诊断语法错误。在编译的时候,准确地诊断语义和逻辑错误是困难得多的事情。

分析器对错误处理的基本目标是:

(1) 清楚而准确地报告错误的出现;

(2) 迅速地从每个错误中恢复过来,以便诊断后面的错误;

(3) 它不应该使正确程序的处理速度降低太多。

这些目标的有效实现面临着挑战。幸好,常见的错误是简单的,直截了当地出错处理机

制一般就够用了。但是,有些时候,错误的实际位置远远先于发现它的位置,并且这种错误的准确性质也难以推断。在另一些困难的场合,错误处理程序甚至还需要猜想程序员的本意。

错误处理程序应怎样报告错误?至少,它应该报告源程序的错误被检测到的位置(它可能偏离错误的真正位置)。很多编译器采用的普遍办法是,印出违例的程序行,指出检测到错误的地方。如果能够知道实际错误很可能是什么,这时还会附带一个诊断信息,如"此处漏了分号"。

一旦查出错误,分析器应如何恢复?我们会结合各种语法分析方法介绍几种一般性的策略,但没有哪一种策略占明显优势。大多数情况下,检测到一个错误就放弃继续分析的分析器是不妥的,因为继续处理输入串还可以发现其他错误。通常,一些编译器试图恢复自己到某个状态,使输入串的分析可以继续进行,剩余程序中的错误能不断地被检测出来。

不妥当的恢复可能会引起讨厌的、以假乱真的伪错误大量涌现,这些错误不是出自程序员,而是错误恢复时改变了分析器的状态引起的。同样地,语法错误的恢复也可能会引入语义伪错误(它们由语义分析或代码生成阶段检查出来)。例如,错误恢复时,分析器可能跳过某个变量声明,如 zap,以后在碰到 zap 时,语法没有错,但由于符号表中无 zap 的条目,因此会产生"zap 没有定义"的信息。

一个保守的策略是,如果查到的后一个错误离前一个太近,就抑制这后一个错误的信息。也就是说,在发现一个语法错误后,编译器要求能正确地分析几个记号,然后才可以报告下一个错误。

看来,错误恢复的策略不得不进行折衷,考虑对常见错误类型的合理处理。另外,上述基本目标的第二点在一些集成编程环境中已不重要,因为编译和编辑之间的切换是非常迅速的。这样的编译器在发现第一个错误时便停下来,等待用户修改和重新提交编译。

非递归预测分析器的栈使得分析器希望和剩余输入匹配的符号变得明显,在下面的讨论中将引用分析栈中的符号。当栈顶的终结符和下一个输入符号不匹配,或者栈顶是非终结符 *A*,*输入符号是 a*,*而 M*[*A*, *a*]*是空白*,*预测分析器即发现一个错误*。

对于非递归的预测分析,采用紧急方式的错误恢复,这是最简单的方法,适用于大多数分析方法。发现错误时,分析器每次抛弃一个输入记号,直到输入记号属于某个指定的同步记号集合为止。同步记号一般是定界符,如分号或 end,它们在源程序中的作用是清楚的。当然,编译器的设计者必须选择适当的同步记号。和以后要介绍的其他方法相比,这种方法简单,不会陷入死循环。这种方法的缺点是常常跳过一段输入符号而不检查其中是否有其他错误,但是在一个语句很少出现多个错误的情况下,它还是可以胜任的。

紧急方式的错误恢复的效果依赖于同步记号集合的选择。这种集合的选择应该使得分析器能迅速地从实际可能发生的错误中恢复过来,一些提示如下:

(1) 至少可以把 FOLLOW(A) 的所有终结符放入非终结符 A 的同步记号集合。如果跳

过一些记号直到看见 FOLLOW(A) 的元素,再把 A 弹出栈,分析一般可以继续下去。

(2) 仅使用 FOLLOW(A) 作为 A 的同步集合是不够的。例如,分号在 C 语言中作为语句的结束符,那么作为语句开始符号的关键字很可能不出现在表达式非终符号的 FOLLOW 集合中。这样,仅按上面 (1) 来设定同步记号集合,作为赋值语句结束的分号的遗漏会引起下一语句的开始关键字被跳过。

语言的结构往往是层次的,如表达式出现在语句中,语句出现在程序块中等等。可以把高层结构的开始符号加到低层结构的同步记号集合中,例如,可以把语句开始的关键字加入到表达式非终结符的同步记号集合。

(3) 如果把 FIRST(A) 的终结符加入 A 的同步记号集合,恢复关于 A 的分析是可能的,只要 FIRST(A) 的终结符出现在输入中。

(4) 如果非终结符可以产生空串,且出错时栈顶是这样的非终结符,那么可以使用产生空串的产生式。这样做会延迟错误的发现,但不会遗漏,好处是可以减少错误恢复要考虑的非终结符数。

(5) 如果终结符在栈顶而不能匹配,简单的办法是,除了报告错误外,弹出此终结符,继续分析。效果上,这种方式等于把所有其他的记号作为该终结符的同步集合。

例 3.16 按照文法 (3.9) 分析表达式时,使用 FOLLOW 符号和 FIRST 符号作为同步记号是合情合理的。该文法的分析表 (表 3.1) 重复在表 3.4,并用 synch 来指示从非终结符的 *FOLLOW 集合中得到的同步记号。非终结符的 FOLLOW 集合由例* 3.12 得到。

表 3.4 同步记号加到表 3.1 的分析表上

非终结符	输入符号					
	id	+	*	(	)	$
E	$E\to TE'$			$E\to TE'$	synch	synch
E'		$E'\to +TE'$			$E'\to\varepsilon$	$E'\to\varepsilon$
T	$T\to FT'$	synch		$T\to FT'$	synch	synch
T'		$T'\to\varepsilon$	$T'\to *FT'$		$T'\to\varepsilon$	$T'\to\varepsilon$
F	$F\to$id	synch	synch	$F\to(E)$	synch	synch

表 3.4 的使用如下:如果分析器查找条目 M[A, a] 并发现它是空的,则跳过输入符号 *a,如果条目是* synch,则调用同步过程并把栈顶的非终结符弹出,恢复分析;如果栈顶的记号不匹配输入符号,则从栈顶弹出该记号,如上面所提到的那样。

表 3.4 的分析器和错误恢复机制面临有语法错误的输入 +id*+id 的行为见表 3.5。

表 3.5 由预测分析器产生的分析和错误恢复动作

栈	输入	输出
$E	+id * + id$	出错,跳过+
$E	id * + id$	id 属于 FIRST(E)
$E'T	id * + id$	
$E'T'F	id * + id$	
$E'T'id	id * + id$	
$E'T'	* + id$	
$E'T'F*	* + id$	
$E'T'F	+ id$	出错:"+"正好在 F 的同步记号集合中,无须跳过任何记号;F 被弹出
$E'T'	+ id$	
$E'	+ id$	
$E'T+	+ id$	
$E'T	id$	
$E'T'F	id$	
$E'T'id	id$	
$E'T'	$	
$E'	$	
$	$	

上面讨论的紧急方式恢复没有涉及错误信息这个重要问题。一般来说,错误信息必须由编译器的设计者提供。

3.4 自下而上分析

本节介绍自下而上分析的一般风格,称做移进-归约分析。编译器常用的移进-归约分析方法叫做 LR 分析,L 是指从左向右扫描输入,R 是指构造最右推导的逆。将在 3.5 节讨论 LR 分析。

3.4.1 归约

移进-归约分析为输入串构造分析树是从叶结点开始,朝着根结点逆序前进。可以把这个过程看成是把输入串归约成文法的开始符号。在每一步归约,一个子串和某个产生式的右部匹配,然后用该产生式的左部符号代替这个子串。如果每步都能恰当地选择子串,那

么它实际跟踪的是最右推导过程的逆过程。

例3.17 考虑文法

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

句子 abbcde 可以按如下步聚归约成 S。首先扫描 abbcde,寻找能够匹配某产生式右部的子串,子串 b 和 d 都可以。选择最左边的 b,用 A 代替(因为有 $A \rightarrow b$),得到 aAbcde。现在子串 Abc,b 和 d 分别都匹配一个产生式的右部,虽然 b 是匹配某产生式右部的最左子串,但用 A 代替子串 Abc (有 $A \rightarrow Abc$),得到 aAde。然后,因有 $B \rightarrow d$,那么用 B 代替 d,得 aABe,再用 S 代替此串。这样,归约序列:

abbcde

aAbcde

aAde

aABe

S

表示了 abbcde 到 S 的归约。

事实上,这些归约刻画出 abbcde 的最右推导过程

$S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcde$

的逆过程。

3.4.2 句柄

非形式地说,一个句型的句柄(handle)是和一个产生式右部匹配的子串,并且,把它归约成该产生式左部的非终结符代表了最右推导过程的逆过程的一步。在很多情况下,句型中能和某产生式 $A \rightarrow \beta$ 右部匹配的最左子串 β 就是句柄,但并非总是这样,有的时候用这个产生式归约产生的串不能归约到开始符号。在例3.17的第二个句型 aAbcde 中,如果用 A 代替 b,得到 aAAcde,那它就不能归约成 S。基于这一点,必须给句柄更精确的定义。

形式地说,右句型(最右推导可得的句型)γ 的句柄是一个产生式的右部 β 以及 γ 中的一个位置,在这个位置可找到串 β,用 A 代替 β(有产生式 $A \rightarrow \beta$)得到最右推导的前一个右句型。即,如果 $S \Rightarrow^{*}_{rm} \alpha Aw \Rightarrow_{rm} \alpha\beta w$,那么在 α 后的 β 是 $\alpha\beta w$ 的句柄。句柄右边的 w 仅含终结符。注意,如果文法二义,那么句柄可能不惟一,因为一个句子可能不只一个最右推导。只有文法无二义时,它的每个右句型才有惟一的句柄。

在上面的例中,abbcde 是右句型,它的句柄是 $A \rightarrow b$ 的右部 b,并且在位置2。同样,aAbcde 也是右句型,它的句柄是 Abc (有产生式 $A \rightarrow Abc$),并且也在位置2。

例 3.18 考虑文法

$$\begin{aligned} E &\to E + E \\ E &\to E * E \\ E &\to (E) \\ E &\to \mathrm{id} \end{aligned} \tag{3.10}$$

和最右推导

$$\begin{aligned} E &\underset{rm}{\Rightarrow} \underline{E * E} \\ &\underset{rm}{\Rightarrow} E * \underline{E + E} \\ &\underset{rm}{\Rightarrow} E * \underline{E + \mathrm{id}_3} \\ &\underset{rm}{\Rightarrow} E * \underline{\mathrm{id}_2} + \mathrm{id}_3 \\ &\underset{rm}{\Rightarrow} \underline{\mathrm{id}_1} * \mathrm{id}_2 + \mathrm{id}_3 \end{aligned}$$

为方便起见,给 id 以下标,并给每个右句型的句柄加底线。例如,id_1 是右句型 $\mathrm{id}_1 * \mathrm{id}_2 + \mathrm{id}_3$ 的句柄。注意,句柄右边的串仅含终结符。

因为文法 (3.10) 是二义的,那么存在着该句子的另一个最右推导:

$$\begin{aligned} E &\underset{rm}{\Rightarrow} \underline{E + E} \\ &\underset{rm}{\Rightarrow} E + \underline{\mathrm{id}_3} \\ &\underset{rm}{\Rightarrow} \underline{E * E} + \mathrm{id}_3 \\ &\underset{rm}{\Rightarrow} E * \underline{\mathrm{id}_2} + \mathrm{id}_3 \\ &\underset{rm}{\Rightarrow} \underline{\mathrm{id}_1} * \mathrm{id}_2 + \mathrm{id}_3 \end{aligned}$$

考虑右句型 $E * E + \mathrm{id}_3$,在这个推导中 $E * E$ *是句柄,而在上一个推导中* id_3 是句柄。

此例的两个最右推导类似于例 3.4 中两个最左推导。第一推导是 + 的优先级高于 *,而第二个反过来。

3.4.3 用栈实现移进—归约分析

如果使用移进—归约的方式分析句子,有两个问题必须解决。第一个是确定右句型中将要归约的子串;第二个是,如果这个子串是多个产生式的右部,如何确定选择哪一个产生式。在进入这些问题之前,首先看一下移进—归约分析器所使用数据结构的类型。

实现移进—归约分析的一种方便的办法是用栈保存文法符号,用输入缓冲区保存要分析的串 w,用$标记栈底,也用它标记输入串的右端。起初,栈是空的,串 w 在输入中,如下所示:

栈	输入
$	*w*$

分析器移动若干个(包括零个)输入符号入栈,直到句柄β在栈顶为止,再把β归约成恰当的产生式左部。分析器重复这个过程,直到它发现错误;或者直到栈中只含开始符号并且输入串为空:

栈	输入
$S	$

进入这个格局后,分析器停机并宣告分析完全成功。

例3.19 逐步看移进—归约分析器在分析输入串 $id_1 * id_2 * id_3$ 时的动作,文法是(3.10),使用例3.18的第一种最右推导过程的逆过程。动作序列见表3.6。注意,由于文法(3.10)对这个输入有两种最右推导,所以还存在分析器可取的另一个动作序列。

表3.6 移进—归约分析器对于输入 $id_1 * id_2 + id_3$ 的格局

栈	输入	动作
$	$id_1 * id_2 + id_3$	移进
$ id_1	$* id_2 + id_3$	按 E→id 归约
$ E	$* id_2 + id_3$	移进
$ $E*$	$id_2 + id_3$	移进
$ $E* id_2$	$+ id_3$	按 E→id 归约
$ $E* E$	$+ id_3$	移进
$ $E* E+$	id_3	移进
$ $E* E+ id_3$	$	按 E→id 归约
$ $E* E+ E$	$	按 E→E+E 归约
$ $E* E$	$	按 E→E*E 归约
$ E	$	接受

分析器的基本动作是移进和归约,实际可能的动作还有两种,即接受和报错。

(1) 移进动作 把下一个输入符号移进栈。

(2) 归约动作 分析器知道句柄的右端已在栈顶,然后它确定句柄的左端在栈中的位置,再决定用什么样的非终结符代替句柄。

(3) 接受动作 分析器宣告分析成功。

(4) 报错动作 分析器发现语法错误,调用错误恢复例程。

有一个重要的事实说明在移进—归约分析中栈的使用是合理的:句柄最终总是出现在栈顶而不是在栈的里面。从表3.6可以看出这个事实是明显的,在第一步归约前和每一步归约后,分析器必须移进若干个(包括零个)符号以使下一个句柄进栈,但它决不需要深入栈中查找句柄。由此可知,使用栈对实现移进—归约是特别方便的。我们还必须解释怎样选

取动作才可以使移进—归约分析器正常工作,马上要讨论的 LR 分析是一种这样的技术。

3.4.4 移进—归约分析的冲突

有些上下文无关文法不能使用移进—归约分析。这种文法的移进—归约分析器会到达这样的格局:它根据栈中所有的内容和下一个输入符号不能决定是移进还是归约(移进—归约冲突),或不能决定按哪一个产生式进行归约(归约—归约冲突)。现在给出一些语法结构的例子,它们属于这种文法。从技术上讲,这些文法不属于 3.5 节定义的 LR(k)类,称它们为非 LR 文法。LR(k)中的 k 代表向前看输入符号的个数,程序设计语言的文法都属于 LR(1)类,即向前看一个符号。

例 3.20 二义文法决不是 LR 的,例如考察悬空 else 文法(3.5)

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

如果移进—归约分析器处于格局

栈	输入
… if *expr* then *stmt*	else … $

我们不知道 if *expr* then *stmt* *是否为句柄,这是移进—归约冲突,所以这个文法不是* LR(1)。更一般地说,没有一种二义文法是 LR(k)的(对任何 k)。

不过,必须指出,移进—归约分析还是可以用来分析某些二义文法的,如上面的 if—then—else 文法。当为含条件语句的两个产生式的文法构造这样的分析器时,存在着上面所讲的冲突,如果采用优先于移进的策略来解决这个冲突,分析器的行为就自然了。3.6 节将讨论这种文法的分析器。

非 LR 出现的另一种情况是,知道了句柄,但根据栈里的内容和下一个输入符号不足以决定按哪个产生式归约。下面的例子说明这种情况。

例 3.21 假定词法分析器对任何标识符回送记号 id 而不管它是如何使用的,假如语言的过程调用是给出它们的名字和参数表,并且数组元素的引用也用同样的语法。因为数组引用的下标和过程调用的参数的翻译是不一样的,需要用不同的产生式来产生实参表和下标表。这样,文法可以有下面一些产生式:

(1) stmt → id (parameter_*list*)
(2) stmt → *expr* := *expr*
(3) parameter_*list* → parameter_*list*, parameter
(4) parameter_*list* → parameter
(5) parameter → id

(6) expr→id (expr_list)

(7) expr→id

(8) expr_list→expr_list, expr

(9) expr_list→expr

由 A(I, J)开始的语句经词法分析后,变为记号流 id (id, id) …进入分析器。把前三个记号移进栈后,分析器的格局是:

栈	输入
… id (id	, id) …

很明显,栈顶的 id 必须归约,但是按哪个产生式归约?如果 A是过程名,应按产生式 (6) 归约 如果 A是数组名,应按产生式 (7) 归约。但是栈中的信息不能告诉我们应按哪个归约,必须使用符号表中有关 A的信息。

解决这个问题的一种办法是把产生式 (1) 的记号 id 改为 procid,并且使用更聪明一点的词法分析器,它识别出作为过程名的标识符时,返回记号 procid。当然,这样做要求词法分析器在返回记号前访问符号表。

这样修改后,处理 A(I, J)时,分析器处于如下格局:

栈	输入
…procid (id	, id) …

或处于前面的格局。前者用产生式 (6) 归约,后者用产生式 (7) 归约。注意,栈中的第三个符号用来决定归约用的产生式,虽然它本身不包含在这个归约中。移进—归约分析可以深入到栈里取信息来指导分析。

3.5 LR 分析器

本节提出一种高效的、自下而上的语法分析技术,它能适用于一大类上下文无关文法的分析。这种技术叫做 LR(k) 分析技术,k 是指在决定分析动作时向前看的符号个数。(k) 省略时,表示 k 是 1。

在讨论 LR 分析算法后,我们提出构造 LR 分析表的三种技术。第一种方法叫做简单的 LR 方法 (简称 SLR),它最容易实现,但功能最弱。对某些文法,用另外两种方法能成功地产生分析表,但用它却失败。第二种方法称为规范的 LR 方法,它功能最强,但代价也最大。第三种方法叫做向前看的 LR 方法 (简称 LALR),它的功能和代价处于另外两者之间。LALR 方法可用于大多数程序设计语言的文法,它可以高效地实现。最后例举非 LR 的上下文无关结构。

3.5.1 LR分析算法

LR分析器的模型见图3.11,它包括输入、输出、栈、驱动程序和含动作和转移两部分的分析表。驱动程序对所有的LR分析方法都一样,不同的分析方法构造的分析表不同。分析程序每次从输入缓冲区读一个符号,它使用栈存储形式为 $s_0X_1s_1X_2s_2\cdots X_ms_m$ 的串,s_m 在栈顶。X_i 是文法符号,s_i 是叫做状态的符号,状态符号概括了栈中它下面部分所含的信息。栈顶的状态符号和当前的输入符号用来检索分析表,以决定移进—归约分析的动作。真正实现时,文法符号不必出现在栈里,不过在我们的讨论中总是包含它们以帮助解释LR分析的行为。

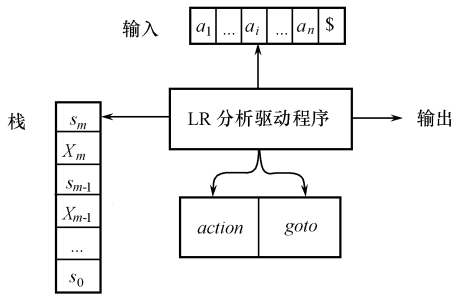


图3.11 LR分析器的模型

分析表由两部分组成,即动作函数 action 和转移函数 goto。LR分析驱动程序的行为是:它根据栈顶当前的状态 s_m 和当前的输入符号 a_i,访问 action[s_m, a],它可能的4个值如下:

(1) 移进 s,其中 s 是一个状态。

(2) 按文法产生式 $A\rightarrow\beta$ 归约。

(3) 接受。

(4) 出错。

函数 goto 取状态和文法符号作为变元,产生一个状态。我们将会明白,用SLR、规范的LR和LALR方法从文法G构造的分析表的goto函数都是识别G的活前缀的确定有限自动机的转换函数。这个DFA的开始状态是初始时置于LR分析器栈中的状态。

文法G的活前缀是它的右句型的前缀,该前缀不超过最右句柄的右端。由这个定义可知,在活前缀的右端加上一些终结符后可以使它成为右句型,因此只要输入串的已扫描部分可以归约成一个活前缀,那就意味着已经扫描的部分没有错误。LR分析栈中的文法符号总是构成活前缀。

LR 分析器的格局是二元组,它的第一个成分是栈的内容,第二个成分是尚未扫描的输入。

$$(s_0\ X_1\ s_1\ X_2\ s_2 \cdots X_m s_m,\ a_i\ a_{i+1} \cdots a_n\ \$)$$

这个格局代表右句型

$$X_1\ X_2 \cdots\ X_m a_i\ a_{i+1} \cdots\ a_n$$

从这里可以看出,它本质上和一般的移进—归约分析器一样,只有栈中的状态是新出现的。

分析器的下一个动作是用当前输入符号 a_i 和栈顶状态 s_m 访问分析表条目 action [s_m, a_i],4 种不同的移动引起的格局变化如下:

(1) 如果 action [s_m, a_i] =移进 *s*,*则分析器执行移进动作,进入格局*

$$(s_0\ X_1\ s_1\ X_2\ s_2 \cdots\ X_m s_m a_i s,\ a_{i+1} \cdots\ a_n\ \$)$$

即分析器把当前输入符号 a_i 和下一个状态 s 移进栈,a_{i+1}成为当前输入符号。

(2) 如果 action [s_m, a_i] =归约 $A \rightarrow \beta$,*则分析器执行归约动作,进入格局*

$$(s_0\ X_1 s_1\ X_2 s_2 \cdots X_{m-r} s_{m-r} A s,\ a_i\ a_{i+1} \cdots\ a_n\ \$)$$

其中 $s = goto[s_{m-r}, A]$,r 是 β 的长度。这里,分析器首先从栈中弹出 2*r* 个符号,*r* 个状态符号和 *r* 个文法符号,这些文法符号刚好匹配产生式右部 β,这时暴露出状态 s_{m-r}。然后把产生式左边的符号 A 和 goto [s_{m-r}, A] 状态 s 推入栈。在归约动作时,当前输入符号没有改变。

LR 分析器的输出由归约时执行与归约产生式有关的语义动作来产生,现在,我们暂且认为输出就是打印归约产生式。

(3) 如果 action [s_m, a_i] = 接受,则分析完成。

(4) 如果 action [s_m, a_i] = 出错,则分析器发现错误,调用错误恢复例程。

LR 分析算法总结在下面的算法中,所有 LR 分析器都按这个算法动作,惟一的区别是分析表的内容不一样。

算法 3.3 LR 分析算法。

输入 LR 分析表和输入串 w。

输出 若 *w 是句子,得到 w 的自下而上*分析,否则报错。

方法 最初,初始状态 s_0 在分析器的栈顶,w$ 在输入缓冲区,然后分析器执行图 3.12 的程序,直至碰到接受或出错动作。

```
让 ip 指向 w$的第一个符号;
repeat forever begin
    令 s 是栈顶的状态,a 是 ip 指向的符号;
    if action[s, a] =移进 s' then begin
        把 a 和 s'依次压入栈;
        推进 ip 指向一下输入符号
    end
    else if action[s, a] =归约 A→β then begin
        栈顶退掉 2 * |β|个符号;
        令 s'是现在的栈顶状态;
        把 A 和 goto[s', A] 压入栈;
        输出产生式 A→β
    end
    else if action[s, a] =接受 then
        return
    else error()
end
```

图 3.12 LR 分析程序

例3.22 表3.7给出一个算术表达式文法的LR分析表,该算术表达式的文法如下:

(1) E→*E* + *T*

(2) E→*T*

(3) T→*T* * *F*

(4) T→*F*

(5) F→(E)

(6) F→id

表中各类动作的含义是:

(1) si表示移进,把当前输入符号和状态i压进栈;

(2) rj表示按第j个产生式进行归约;

(3) acc表示接受;

(4) 空白表示出错。

表3.7 表达式文法的分析表

状态	动作						转移		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

注意,对于终结符a,状态转移动作可以在动作表的条目action[s, a]中找到,所以转移表中仅给出非终结符A的goto[s, A]。另外,我们还没有解释表3.7的条目都是怎么确定的,这个问题稍后讨论。

面对输入id * id + id,栈内容和输入内容的变化序列在表3.8中给出。例如,在第一行,LR分析器处于状态0,当前输入符号是id。表3.7的第0行和id列的动作是s5,它的含义是移进id,再把状态5压进栈,如第二行所示。然后,*成为当前输入符号,状态5面

对输入＊的动作是按 $F \rightarrow id$ 归约。这时两个符号(一个状态符号和一个文法符号)弹出栈，状态0显露出来。因为 goto[0，F]是3，因此把 F 和3推进栈，到达第三行所示的格局。剩余的动作也类似地决定。

表3.8 移进—归约分析器对于输入 $id_1 * id_2 + id_3$ 的格局

栈	输 入	动 作
0	id ＊ id ＋ id$	移进
0 id 5	＊ id ＋ id$	按 F→id 归约
0 *F* 3	＊ id ＋ id$	按 T→ *F* 归约
0 *T* 2	＊ id ＋ id$	移进
0 *T* 2 ＊7	id ＋ id$	移进
0 *T* 2 ＊7 id 5	＋ id$	按 F→id 归约
0 *T* 2 ＊7 *F* 10	＋ id$	按 T→ *T* ＊ *F* 归约
0 *T* 2	＋ id$	按 E→ *T* 归约
0 *E* 1	＋ id$	移进
0 *E* 1 ＋6	id$	移进
0 *E* 1 ＋6 id 5	$	按 F→id 归约
0 *E* 1 ＋6 *F* 3	$	按 T→ *F* 归约
0 *E* 1 ＋6 *T* 9	$	按 E→ *E* ＋ *T* 归约
0 *E* 1	$	接受

3.5.2 LR 文法和 LR 分析方法的特点

一个文法，如果能为它构造出所有条目都惟一的 LR 分析表，就说它是 LR 文法。存在非 LR 的上下文无关文法(将在3.5.6节讨论)，但程序设计语言的结构一般都是 LR 的。直观上说，如果句柄出现在栈顶时，自左向右扫描的移进—归约分析器能及时识别它，那么文法就是 LR 的，LR 分析器不需要扫描整个栈就可以知道句柄是否出现在栈顶，因为栈顶的状态符号包含了确定句柄所需要的一切信息。它是根据一个非常重要的事实：如果仅根据栈内的文法符号就可以识别句柄的话，那么存在一个有限自动机，它自底向上读栈中的文法符号就能确定栈顶是什么句柄(如果有的话)。LR 分析表的转移函数本质上就是这样的有限自动机。这个有限自动机并不需要随分析栈的每一步变化而自底向上读一次栈，因为如果这个识别句柄的有限自动机自底向上读栈中的文法符号的话，它最后到达的状态正是这时栈顶的状态符号所表示的状态，所以 LR 分析器可以从栈顶的状态确定它需要从栈中了解的一切。在学习了下面的分析表构造方法以后，读者才能对这段话有深刻的理解。

能够用来帮助 LR 分析器作出移进—归约决定的另一个信息源是剩余输入的前 k 个符

号,我们最感兴趣的是 k=0 或 k=1 的情况,并且仅讨论 k≤1 的情况。例如,表3.7仅向前看一个符号。最多向前看 k 个符号就可以决定动作的 LR 分析器所分析的文法叫做 LR(k)文法。

LR 分析器富有吸引力的原因是:

能够构造 LR 分析器来识别所有能用上下文无关文法写的程序设计语言的结构。

LR 分析方法是已知的最一般的无回溯的移进—归约方法,它能和其他移进—归约方法一样有效地实现。

LR 方法能分析的文法类是预测分析法能分析的文法类的真超集。

LR 分析器能及时发现语法错误,在自左向右扫描输入的前提下,它快到不能再快的程度。

这种方法的主要缺点是,对真正的程序设计语言文法,手工构造 LR 分析器的工作量太大,因而需要专门的工具——LR 分析器的生成器。幸好有很多这样的生成器可用,我们将讨论其中之一 Yacc 的设计和使用。有了这样的生成器,只要写出上下文无关文法,就可以用它自动产生该文法的分析器。如果文法二义或者有其他难以自左向右分析的结构,分析器的生成器能定位这些结构,并向编译器的设计者报告这些情况。

LL 文法和 LR 文法有明显的区别。对于 LR(k)文法,要求在看见了产生式右部推出的所有东西和 k 个向前看符号后,能够识别产生式右部的出现。这个要求远不如 LL(k)那样严峻,LL(k)文法要求在看见了右部推出的前 k 个符号后就识别所使用的产生式。所以 LR 文法比 LL 文法能描述更多的语言。

例3.23 现有句型 γlβbw,其规范推导是 $S \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma Abw \underset{rm}{\Rightarrow} \gamma l\beta bw$,*最后一步用的产生式是* $A \rightarrow l\beta$。LL(1)方法在看见右部 lβ 的第一个符号 l 时就必须决定用这个产生式,而 LR(1)方法在看见右部 lβ 的后继符号 b 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。

3.5.3 构造 SLR 分析表

本小节讨论怎样从文法构造 LR 分析表。我们将给出三种方法,它们的功能和实现的难易程度不同。第一种叫做简单的 LR 方法,简称 SLR 方法,它最弱但最易实现。根据这种方法构造的分析表叫做 SLR 分析表,使用 SLR 分析表的 LR 分析器叫做 SLR 分析器,能够为之构造 SLR 分析器的文法叫做 SLR 文法。另两种方法用向前看的信息来增强 SLR 文法,所以 SLR 方法是研究 LR 分析的理想起点。

文法 G 的 LR(0)项目(简称项目)是在右部的某个地方加点的产生式。如产生式 $A \rightarrow XYZ$ *对应有四个项目:*

$A \rightarrow \cdot XYZ$

$A \to X \cdot YZ$

$A \to XY \cdot Z$

$A \to XYZ \cdot$

产生式 $A \to \varepsilon$ 只有一个项目 $A \to \cdot$ 和它对应。直观地讲,项目表示在分析过程的某一点,已经看见了产生式的多大部分(点的左边部分)和下面希望看见的部分。例如,上面的第一个项目表示希望下一步从输入中看见由 XYZ 推出的串,第二个项目表示我们刚从输入中看见了由 X 推出的串,下面希望看见由 YZ 推出的串。也可以这么说,点的左边代表历史信息,而右边代表展望信息。

SLR 方法的主要思想是首先从文法构造识别活前缀的确定有限自动机。我们把项目按一定方法组成集合,这些集合对应 SLR 分析器的状态,也是这个 DFA 的状态。这样的一族 LR(0) 项目集,称做 LR(0) 项目集的规范族,它提供了构造 SLR 分析表的基础。为了构造文法的 LR(0) 项目集规范族,我们定义拓广文法和两个函数 closure 和 goto。

如果文法 G 的开始符号是 S,那么 G 的拓广文法 *G′是在 G 的基础上增加了新的开始符号* S′和产生式 $S' \to S$。*新产生式的目的是用来指示分析器什么时候应该停止分析和宣布接受输入。也就是当且仅当分析器执行归约* $S' \to S$ *时,意味着分析成功。*

下面先讲闭包函数 colsure。

如果 I 是文法 G 的一个项目集,那么 closure(I) 是用下面两条规则从 I 构造的项目集:

(1) 初始时,I 的每个项目都加入 closure(I)。

(2) 如果 $A \to \alpha \cdot B\beta$ *在 closure(I) 中,且* $B \to \gamma$ *是产生式,那么如果项目* $B \to \cdot \gamma$ *还不在 closure(I) 中的话,则把它加入。运用这条规则,直到没有更多的项目可加入 closure(I) 为止。*

直观上,$A \to \alpha \cdot B\beta$ *在 closure(I) 中表示在分析过程的某一点,下一步应该从输入中看见的可能是由* $B\beta$ 推出的串,因为 $B \to \gamma$ *是产生式,也可以认为首先应该从输入中看见由* γ *推出的子串,出于这个原因,把* $B \to \cdot \gamma$ *加入 closure(I)。*

例 3.24 考虑拓广的表达式文法

$$\begin{aligned} &E' \to E \\ &E \to E + T \mid T \\ &T \to T * F \mid F \\ &F \to (E) \mid \mathrm{id} \end{aligned} \tag{3.11}$$

如果 I 是项目集 $\{[E' \to \cdot E]\}$,*那么 closure(I) 含下列项目:*

$E' \to \cdot E$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$

这里,由规则(1),$E' \rightarrow \cdot E$ 被加入 closure(I)。因为 E 紧接在点的右边,由规则(2),把对应 E 产生式的、点在左端的项目 $E \rightarrow \cdot E + T$ 和 $E \rightarrow \cdot T$ 都加入。同样道理,把 $T \rightarrow \cdot T * F$ 和 $T \rightarrow \cdot F$ 都加入。最后,再把 $F \rightarrow \cdot (E)$ 和 $F \rightarrow \cdot id$ 也都加入。再没有其他项目可由规则(2)加入。

函数 closure 可以如图3.13那样计算。

```
function closure(I);
    begin
        J := I;
        repeat
            for J的每个项目 A→α·Bβ和G的每个
                    产生式 B→γ,若 B→·γ不在J中 do
                把 B→·γ加入J
        until 没有更多的项目可以加入 J;
        return J
    end
```

图3.13 closure的计算

从上面知道,如果 B 在点的右邻的一个项目加入 closure(I),那么与 B 的各产生式对应的并且点在左端的项目都加入这个闭包。事实上,并不需要列出由闭包运算加入的所有项目 $B \rightarrow \cdot \gamma$,而只要列出这样的非终结符 B 就足够了,我们可以把项目集的项目分成两类:

(1) 核心项目 它包括初始项目 $S' \rightarrow \cdot S$ 和所有那些点不在产生式右部的左端的项目。

(2) 非核心项目 它们的点在产生式右部的左端。

而且,每个所需要的项目集都可以由取核心项目集的闭包形成。当然,加入闭包的项目决不会是核心项目。这样,如果扔掉所有的非核心项目,那么可以用较少的存储空间来表示所需的项目集,因为非核心项目可以由闭包过程重新生成。

下面再看转移函数 goto(I, X),其中 I 是项目集,X 是文法符号。goto(I, X)的定义是,满足 $[A \rightarrow \alpha \cdot X\beta]$ 在 I 中所有项目 $[A \rightarrow \alpha X \cdot \beta]$ 的集合的闭包。直观上讲,如果 I 是对某个活前缀 γ 有效的项目集,那么 goto(I, X)是对活前缀 γX 有效的项目集。

例3.25 如果 I 是两个项目的集合 $\{[E' \rightarrow E \cdot], [E \rightarrow E \cdot + T]\}$,那么 goto(I, +)包括以下项目:

$E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

这是通过考察 I的项目,看+是否紧挨在点的右边来计算 *goto* (*I*, +) 的。 $E' \to E \cdot$ 不是这样的项目,但是 $E \to E \cdot + T$ 是。*把点移过*+得到 {[$E \to E + \cdot T$]} ,*然后求它的闭包*。

现在可以构造拓广文法 G′的 LR(0) 项目集的规范族了,算法如图3.14。

```
procedure items(G′);
  begin
    C := closure({[S′→·S]});
    repeat
      for 对 C的每个项目集I和每个文法符号 X,
            若 goto(I, X) 非空且不在 C中 do
          把 goto(I, X) 加入 C中
    until 没有更多的项目可以加入 C
  end
```

图 3.14 项目集的构造

例 3.26 例 3.24 文法 (3.11) 的 LR(0) 项目集规范族见图 3.15,这些项目集的 goto 函数以及接受活前缀的确定有限自动机 D 的转换图形式显示在图 3.16。

I_0 : $E' \to \cdot E$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

I_1 : $E' \to E \cdot$
$E \to E \cdot + T$

I_2 : $E \to T \cdot$
$T \to T \cdot * F$

I_3 : $F \to id \cdot$

I_4 : $E \to E + \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

I_5 : $T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

I_6 : $F \to (E \cdot)$
$E \to E \cdot + T$

I_3：$T \to F \cdot$

I_4：$F \to (\cdot E)$
$E \to \cdot E + T$
$E \to \cdot T$
$T \to \cdot T * F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathrm{id}$

I_5：$F \to \mathrm{id} \cdot$

I_6：$E \to E + \cdot T$
$T \to \cdot T * F$

I_7：$T \to T * \cdot F$
$F \to \cdot (E)$
$F \to \cdot \mathrm{id}$

I_8：$F \to (E \cdot)$
$E \to E \cdot + T$

I_9：$E \to E + T \cdot$
$T \to T \cdot * F$

I_{10}：$T \to T * F \cdot$

I_{11}：$F \to (E) \cdot$

图3.15 文法(3.11)的LR(0)项目集规范族

如果图3.16中*D的每个状态都是接受状态且I_0是初态*，那么D识别的正是文法(3.11)的活前缀。这决不是偶然的。可以证明，对每个文法G，项目集规范族的goto函数定义了一个DFA，它识别G的活前缀。

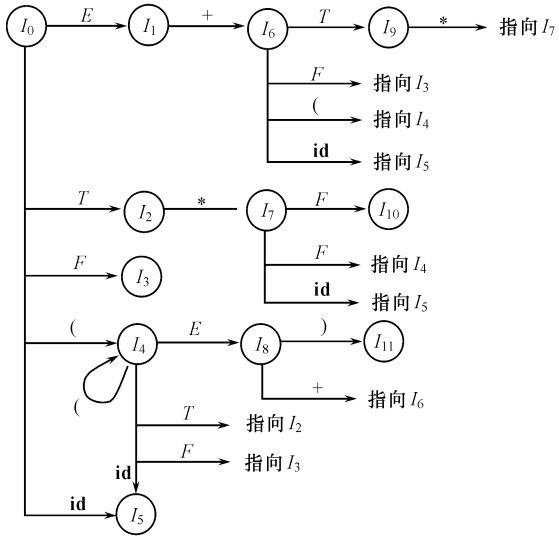


图3.16 接受活前缀的DFA*D的转换图*

事实上，也可以构造一个识别活前缀的NFA *N，它的状态是项目本身，从* $A \to \alpha \cdot X\beta$ *到* $A \to \alpha X \cdot \beta$ *有转换标记X，从* $A \to \alpha \cdot B\beta$ *到* $B \to \cdot \gamma$ *有转换标记ε。项目集(N的状态集)I的closure(I)正好是*2.3节定义的NFA状态集的ε闭包。若从N用子集构造法构造DFA，那么*goto(I, X)正好给出了在这个*DFA中从I根据符号X的转换。按照这种观点，图3.14的过程items(G′)正是子集构造法本身运用于从G′构造的这个NFA *N*。

如果 $S' \Rightarrow_{rm}^{*} \alpha Aw \Rightarrow_{rm}^{*} \alpha\beta_1\beta_2 w$，*则说项目* $A \rightarrow \beta_1 \cdot \beta_2$ 对活前缀 $\alpha\beta_1$ 是有效的。一般而言，一个项目可能对好几个活前缀都是有效的。对任何活前缀 $\alpha\beta_1$，从项目 $A \rightarrow \beta_1 \cdot \beta_2$ 有效这个事实可以知道，当 $\alpha\beta_1$ 在分析栈的栈顶时，分析器应该移进呢还是归约。如果 $\beta_2 \neq \varepsilon$，*那么它暗示*句柄还没有完全进栈，应该移进。如果 $\beta_2 = \varepsilon$，*那么* β_1 是句柄，应该用产生式 $A \rightarrow \beta_1$ 归约。

另一方面，一个活前缀可能有多个有效项目。一个活前缀 γ 的有效项目集就是从上述 DFA 的初态出发，沿着标记为 γ 的路径到达的那个项目集（状态），这是 LR 分析理论的一条基本定理。这样，当活前缀 γ 在栈顶时，其有效项目集（也就是栈顶的状态）概括了所有可以从栈中收集到的有用信息。我们不打算证明这个定理，而是在下面用例子来阐明。两个不同的有效项目可能要求分析器采取不同的动作，一些这样的冲突可以由向前看下一个输入符号解决，还有一些可以由下一小节的方法解决。当 LR 方法用于构造任意文法的分析表时，不能期望所有分析动作的冲突都是可以解决的。

例 3.27 再次考察文法（3.11），它的项目集和 goto 函数显示在图 3.15 和图 3.16 中。显然，串 $E + T*$ 是（3.11）的活前缀。在读完 $E + T*$ 后，图 3.16 的自动机处于状态 I_7，它包含项目

$$T \rightarrow T * \cdot F$$
$$F \rightarrow \cdot (E)$$
$$F \rightarrow \cdot \mathrm{id}$$

它们都对 $E + T*$ 有效。为了明白这一点，可以看下面 3 个最右推导：

$$E' \Rightarrow E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * \mathrm{id} \Rightarrow E + T * F * \mathrm{id}$$

$$E' \Rightarrow E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * (E)$$

$$E' \Rightarrow E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * \mathrm{id}$$

这 3 个推导分别展示了 $T \rightarrow T * \cdot F$、$F \rightarrow \cdot (E)$ 和 $F \rightarrow \cdot \mathrm{id}$ 的有效性。可以证明，不存在对 $E + T*$ 有效的其他项目了，这个证明留给感兴趣的读者。

以上完成了构造 SLR 分析表的第一步，即从文法构造识别活前缀的确定有限自动机。现在进行第二步，说明怎样从识别活前缀的确定有限自动机构造 SLR 分析表的动作函数和转移函数。该算法不可能为任何文法产生所有条目都惟一的分析表，但是它对许多程序设计语言的文法是成功的。给定文法 G，我们拓广 G，产生 G′，从 G′ 构造它的项目集规范族 *C，从 C 使用下面的算法构造分析表的动作函数 action 和转移函数 goto。这个算法需要知道*每个非终结符 A 的 FOLLOW(A)（见 3.3 节）。

算法 3.4 构造 SLR 分析表。

输入 拓广的文法 G'。

输出 *G′的* SLR 分析表的 action 函数(动作表)和 goto 函数(转移表)。

方法 (1) 构造 G′的 LR(0)项目集规范族 $C=\{I_0, I_1, \cdots, I_n\}$。

(2) 状态 i 从 I_i 构造。action 函数在状态 i 的值如下确定：

(a) 如果 $[A\rightarrow\alpha\cdot a\beta]$ *在* I_i 中，并且 $goto(I_i, a)=I_j$，那么置 action[i, a]为 sj，其含义是把 a 和状态 j 移进栈。这里，a 必须是终结符。

(b) 如果 $[A\rightarrow\alpha\cdot]$ 在 I_i 中，那么对 FOLLOW(A)中的所有 a，置 action[i, a]为 rj，j 是产生式 $A\rightarrow\alpha$ *的编号。这个动作的意思是按产生式* $A\rightarrow\alpha$ *归约。这里，A 不是 S′。*

(c) 如果 $[S'\rightarrow S\cdot]$ 在 I_i 中，那么置 action[i, $]为接受 acc。

如果由上面规则产生的动作有冲突，那么该文法就不是 SLR(1)的。在这种情况下，这个算法不产生分析器。

(3) 使用下面规则构造 goto 函数在状态 i 的值：

对所有的非终结符 A，如果 $goto(I_i, A)=I_j$，那么 goto[i, A]$=j$。

(4) 不能由规则(2)和(3)定义的条目都置为 error。

(5) 分析器的初始状态是包含 $[S'\rightarrow\cdot S]$ *的项目集对应的状态。*

由算法 3.4 决定的动作函数和转移函数组成的分析表叫做文法 *G 的* SLR(1)分析表，使用 G 的 SLR(1)分析表的 LR 分析器叫做 G 的 SLR(1)分析器，有 SLR(1)分析表的文法叫做 SLR(1)文法。通常省略 SLR 后面的(1)，因为我们不讨论向前看两个或多个符号的分析器。

例 3.28 为文法(3.11)构造 SLR 分析表，它的 LR(0)项目集规范族已在图 3.15 给出。首先考虑项目集 I_0：

$$E'\rightarrow\cdot E$$
$$E\rightarrow\cdot E+T$$
$$E\rightarrow\cdot T$$
$$T\rightarrow\cdot T*F$$
$$T\rightarrow\cdot F$$
$$F\rightarrow\cdot(E)$$
$$F\rightarrow\cdot id$$

项目 $F\rightarrow\cdot(E)$ 使得条目 action[0, (] $=s4$，项目 $F\rightarrow\cdot id$ 使得条目 action[0, id] $=s5$。I_0 的其他项目不产生动作。现在考虑项目集 I_1：

$$E'\rightarrow E\cdot$$
$$E\rightarrow E\cdot+T$$

第一项导致 action[1, $] $=acc$，*第二项使得* action[1, +] $=s6$。再考虑 I_2：

$$E\rightarrow T\cdot$$
$$T\rightarrow T\cdot*F$$

因为 FOLLOW(E) = {$, +,)}，因此第一项使得 action[2, $] = action[2, +] = action[2,)] = r2（因为 E→T 的序号为2）。第二项使得 action[2, *] = s7。以这种方法继续下去可得到表3.7的动作表和转移表。

每个 SLR(1) 文法都不是二义的，但是有很多非二义的文法（包括一些程序设计语言结构）都不是 SLR(1) 的，这说明 SLR(1) 文法的描述能力有限。

例3.29　看下面文法：

$$\begin{aligned} &S \rightarrow V = E \\ &S \rightarrow E \\ &V \rightarrow * E \\ &V \rightarrow \mathrm{id} \\ &E \rightarrow V \end{aligned} \tag{3.12}$$

可以把 V 和 E 想象成分别代表左值和右值，左值表示一个存储单元，右值是一个可存储的值。*表示“取单元内容”的算符。文法(3.12)的 LR(0) 项目集规范族如图3.17所示。

I_0：S′→·S
　　S→·V=E
　　S→·E
　　V→·* E
　　V→·id
　　E→·V

I_1：S′→S·

I_2：S→V·=E
　　E→V·

I_3：S→E·

I_4：V→*·E
　　E→·V
　　V→·*E
　　V→·id

I_5：V→id·

I_6：S→V=·E
　　E→·V
　　V→·* E
　　V→·id

I_7：V→* E·

I_8：E→V·

I_9：S→V=E·

图3.17　文法(3.12)的规范 LR(0) 族

考察项目集 I_2，这个集合的第一项目使得 action[2, =] 是 s6。因为 FOLLOW(E) 包含=（因为 S⇒V=E⇒*E=E）。第二项目使得 action[2, =] 为按 E→V 归约。这样，条目 action[2, =] 多重定义，因为既有移进条目又有归约条目在其中，状态2在输入符号是=时有移进—归约冲突。

文法(3.12)不是二义的。移进—归约冲突的出现说明了SLR分析器的构造方法没有强到记住足够多的上下文,以决定在看见了可归约到V的串并且面临输入=时,分析器应该取什么动作。下面要讨论的规范LR方法和LALR方法,将在更大的文法类上取得成功,包括文法(3.12)。

3.5.4 构造规范的LR分析表

现在给出从文法构造规范LR分析表的方法。回顾上面讲的SLR方法,如果I_i包含项目$[A\to\alpha\cdot]$且a在FOLLOW(A)中,那么状态i要求面临a时按$A\to\alpha$归约。*但是在有些场合下*,当状态i出现在栈顶、活前缀$\beta\alpha$在栈中并且a是当前输入符号时,用$A\to\alpha$*来归约却是不合适的*,因为在右句型中,a不可能跟随在βA的后面。

例3.30 再来看例3.29。在状态2有项目$E\to V\cdot$,它对应上面的$A\to\alpha\cdot$,=对应上面的*a,它在FOLLOW(E)中。于是*SLR分析器在状态2且面临输入=时,要求按$E\to V$归约。但是项目$S\to V\cdot=E$*也在状态*2中,因此分析器在状态2且面临=时也要求移进。然而例3.29的文法不存在以$E=\cdots$开始的右句型,因此分析器此时不应该把V归约成E。

让状态含有更多的信息,使之能够剔除上述那些无效归约是完全可能的。我们可以设想,必要时对状态进一步细分,使得LR分析器的每个状态能够确切地指出,当α后面跟哪些终结符时才容许把α归约为A。

重新定义项目,使之包含一个终结符作为第二个成分,这样就把更多的信息并入了状态。项目的一般形式也就成了$[A\to\alpha\cdot\beta,a]$,*其中*$A\to\alpha\beta$*是产生式*,a*是终结符号或*\$,这种项目叫做LR(1)项目,1是第二个成分的长度,这个成分叫做项目的搜索符。搜索符对β非空的项目$[A\to\alpha\cdot\beta,a]$*是不起作用的,但对形式为*$[A\to\alpha\cdot,a]$的项目,它表示只有在下一个输入符号是a时,才能要求按$A\to\alpha$*归约。这样,分析器只有在输入符号是a时才按*$A\to\alpha$*归约,其中*$[A\to\alpha\cdot,a]$在栈顶状态的LR(1)项目集中。这样的a的集合是FOLLOW(A)的子集,完全可能是真子集,如例3.30那样。

我们说LR(1)项目$[A\to\alpha\cdot\beta,a]$*对活前缀*γ*是*有效的,如果存在着推导$S\overset{*}{\underset{rm}{\Rightarrow}}\delta Aw\underset{rm}{\Rightarrow}\delta\alpha\beta w$,*其中*:

(1) $\gamma=\delta\alpha$;

(2) a是w的第一个符号,或者w是ε且a是\$。

例3.31 考虑文法

$$S\to BB$$
$$B\to bB\mid a$$

它有一个最右推导$S\overset{*}{\underset{rm}{\Rightarrow}}bbBBa\underset{rm}{\Rightarrow}bbbBBa$。*我们看到,项目*$[B\to b\cdot B,b]$*对活前缀*$\gamma=bbb$是有效的。根据上面定义,只需令$\delta=bb,A=B,w=ba,\alpha=b$*且*$\beta=B$*即可。*

再看该文法的另一个最右推导 $S \Rightarrow_{rm}^{*} BbB \Rightarrow_{rm} BbbB$，可以看出项目 $[B \rightarrow b \cdot B, \$]$ 对活前缀 Bbb 是有效的。

构造 LR(1) 项目集规范族的方法本质上和构造 LR(0) 项目集规范族的方法是一样的，只需要修改 closure 函数和 goto 函数。

为了懂得 closure 运算的新定义，考虑对活前缀 γ 有效的项目集中的项目 $[A \rightarrow \alpha \cdot B\beta, a]$。该文法必定存在一个最右推导 $S \Rightarrow_{rm}^{*} \delta Aax \Rightarrow_{rm} \delta\alpha B\beta ax$，其中 $\gamma = \delta\alpha$。假定 βax 推出终结符串 by，那么对每个形式为 $B \rightarrow \eta$ 的产生式，有推导 $S \Rightarrow_{rm}^{*} \gamma Bby \Rightarrow_{rm} \gamma\eta by$，于是 $[B \rightarrow \cdot \eta, b]$ 对 γ 有效。注意，b 可能是从 β 推出的第一个终结符，或者在推导 $\beta ax \Rightarrow^{*} by$ 中，β 推出 ε，b 就成了 a，总结这两种可能性，可以说 b 是 FIRST(βax) 中的任何终结符。注意，x 不可能含 by 的第一个终结符，所以 FIRST(βax) = FIRST(βa)。现在我们给出 LR(1) 项目集的构造算法。

算法 3.5 构造 LR(1) 项目集。

输入 拓广文法 G'。

输出 LR(1) 项目集，它们是对 G′的若干个活前缀有效的项目集。

方法 构造项目集的函数 closure 和 goto 及主例程 items，见图 3.18。

```
function closure (I) ;
begin
    repeat
        for I的每个项目 [A→α·Bβ, a]，G′中的每个产生式 B→γ
                和 FIRST(βa) 的每个终结符 b，如果 [B→·γ, b] 不在 I中 do
            把 [B→·γ, b] 加到 I;
    until 再没有项目可加到 I;
    return I
end;

function goto (I, X) ;
begin
    令 J是项目 [A→αX·β, a] 的集合，条件是 [A→α·Xβ, a] 在 I中;
    return closure (J)
end;

procedure items (G′) ;
begin
    C ≔ closure ({S′→·S, $}) ;
    repeat
```

for C的每个项目集I和每个文法符号X,若 goto(I, X)非空
且不在 C中 do
把 goto(I, X)加入 C中
until 再没有项目集可以加入 C中
end

图 3.18 构造文法 G′的 LR(1)项目集

例 3.32 拓广文法

$$S' \to S$$
$$S \to BB \tag{3.13}$$
$$B \to bB$$

的 LR(1)项目集规范族见图 3.19,goto函数已在图中给出。另外,[B→·bB, b/a]是两个项目[B→·bB, b]和[B→·bB, a]的缩写。

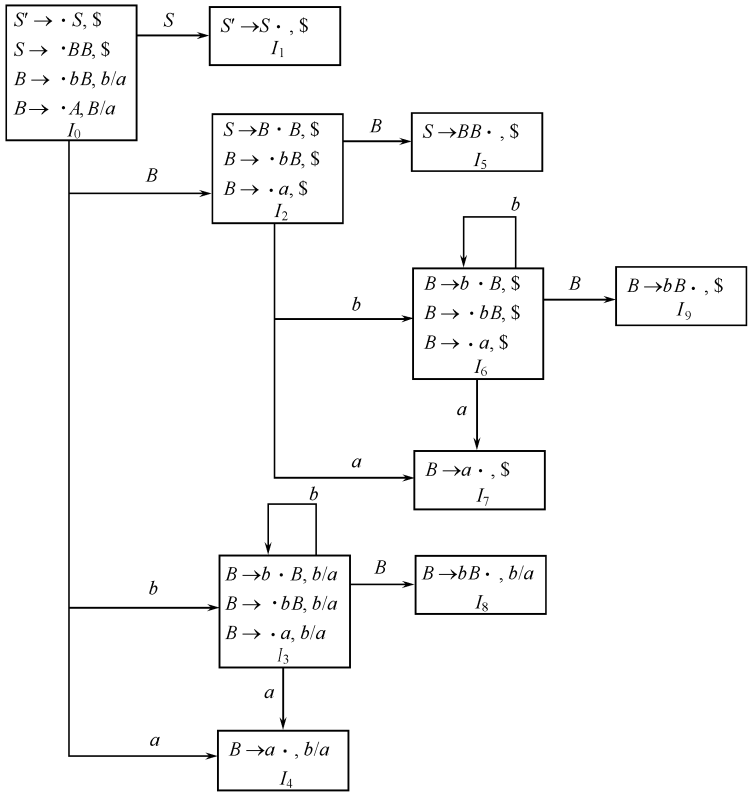


图 3.19 文法(3.13)的状态转移

下面给出从 LR (1) 的项目集构造 LR (1) 分析的动作函数和转移函数的规则。

算法 3.6 构造规范的 LR 分析表。

输入 拓广文法 G'。

输出 文法 G'*的规范* LR 分析的 action 函数和 goto 函数。

方法 (1) 构造 G'的 LR (1) 项目集规范族 $C=\{I_0, I_1, \cdots, I_n\}$。

(2) 从 I_i 构造分析器的状态 i，action 函数在状态 i 的值如下确定：

(a) 如果 $[A\to\alpha\cdot a\beta, b]$ *在* I_i 中，且 $goto(I_i, a)=I_j$，那么置 action [i, a] 为 sj，这里，a 一定是终结符。

(b) 如果 $[A\to\alpha\cdot, a]$ 在 I_i 中，且 $A\neq S'$，*那么置 action [i, a] 为 rj，j 是产生式* $A\to\alpha$ *的*序号。

(c) 如果 $[S'\to S\cdot, \$]$ 在 I_i 中，那么置 action [i, $] = acc。

如果用上面规则构造出现了冲突，那么文法就不是 LR (1) 的。算法对此文法失败。

(3) goto 函数在状态 i 的值如下确定：

如果 $goto(I_i, A)=I_j$，*那么* $goto[i, A]=j$。

(4) 用规则 (2) 和 (3) 未能定义的所有条目都置为 error。

(5) 分析器的初始状态是包含 $[S'\to\cdot S, \$]$ 的项目集对应的状态。

用算法 3.6 产生的动作函数和转移函数构成的表叫做规范的 LR (1) 分析表，使用这种表的 LR 分析器叫做规范的 LR (1) 分析器。如果动作函数没有多重定义的条目，那么这个文法叫做 LR (1) 文法。和前面一样，如果不会引起误解的话，省略“ (1) ”。

例 3.33 文法 (3.13) 的规范 LR 分析表见表 3.9，产生式 1，2 和 3 分别是 $S\to BB$，$B\to bB$ 和 $B\to a$。

表 3.9 文法 (3.13) 的规范 LR 分析表

状态	动作			转移	
	b	*a*	$	*S*	*B*
0	*s*3	*s*4		1	2
1			*acc*		
2	*s*6	*s*7			5
3	*s*3	*s*4			8
4	*r*3	*r*3			
5			*r*1		
6	*s*6	*s*7			9
7			*r*3		
8	*r*2	*r*2			
9			*r*2		

所有的SLR(1)文法都是LR(1)文法,但对于SLR(1)文法,规范的LR分析器可能比同一文法的SLR分析器有更多的状态。上面例子的文法是SLR文法,它的SLR分析器只有7个状态,而图3.19有10个状态。

对于例3.29,当构造它的规范LR(1)分析表时,可以看到不存在任何冲突。

3.5.5 构造LALR分析表

本小节介绍最后一种分析器的构造方法——LALR(lookahead LR)技术。实际的编译器经常使用这种方法,因为由它产生的分析表比规范LR的分析表要小得多,而对大多数一般的程序设计语言来说,其语法结构都能方便地由LALR文法表示。同样的结论对SLR文法几乎也是对的,但是有少数结构不能方便地由SLR技术处理(例3.29便是一个例子)。

就分析器的大小而言,SLR和LALR的分析表对同一个文法有同样多的状态,而规范LR分析表要大得多。例如,对Pascal这样的语言,SLR和LALR的分析表有几百个状态,而规范LR分析表有几千个状态。所以,使用SLR和LALR的分析表比使用规范LR分析表要经济得多。

再次考虑文法(3.13),它的LR(1)项目集见图3.19。取一对看起来类似的状态,如I_4和I_7,它们都只有一个项目,并且第一成分都是$B \rightarrow a\cdot$,I_4的搜索符是b或a,I_7的搜索符是\$。

我们来看一下分析器中I_4和I_7的不同作用。注意,文法(3.13)产生的是正规集b^*ab^*a。*当读输入bb…babb…ba时,分析器把第一组b和它后面的a移进栈,进入状态4*。随后,如果下一个输入符号是b或a的话,分析器按产生式$B \rightarrow a$*归约,因为b或a属于第2个*b^*a*的开始符号。如果下一个输入符号是*\$,那么整个输入实际是bb…*ba的形式,它不*属于这个语言,这时状态4能正确地指出错误。

在读过第二个a后,分析器进入状态7。这时分析器必须看见输入结束标记\$,否则输入串就不是b^*ab^*a*的形式。所以,合乎道理的做法是,面临*\$时,状态7应按$B \rightarrow a$*归约*,面临b或a时报告错误。

现在,我们把状态I_4和I_7合并为I_{47},并把它们的搜索符合起来,成为$[B \rightarrow a\cdot, b/a/\$]$。从$I_0$,$I_2$,$I_3$和$I_6$到达$I_4$或$I_7$的a转移现在都进入$I_{47}$,状态$I_{47}$的动作是不管面临任何符号都归约。修改后的分析器的行为本质上和原来的一样,但它会把某些情况下的a归约成B,例如输入是bba或babab时,而原来的分析器对这些情况是报错的。值得庆幸的是,这些错误最终还是会被逮住,而且是在移进下一个输入符号前被逮住。

更一般地说,我们寻找同心的LR(1)项目集,即略去搜索符后它们是相同的集合,并把这些同心集合并成一个项目集。例如在图3.19中,I_4和I_7,I_5和I_6以及I_8和I_9分别是同心的项目集。注意,一般而言,一个心是对应文法的一个LR(0)项目集,另外,LR(1)文法可

能会使多个项目集同心。

因为 goto(I, X)的心仅依赖于 I的心,被合并集合的 goto函数的结果集合也可以合并,所以项目集合并时带来的 goto函数修改不会引起问题。动作函数应做相应的修改,使得它能够反映合并前所有项目集的非出错动作。

对 LR(1)文法,如果把所有的同心集合并,有可能导致冲突。但是这种冲突不会是移进—归约冲突。因为,如果存在这种冲突,则意味着,面对当前的输入符号 a,有一项目[A→α·, a]要求采取归约动作,同时又有另一项目[B→β·aγ, b]要求把 a移进。这两个项目既然同处于合并之后的项目集中,则意味着在合并前,必有某个 c使得[A→α·, a]和[B→β·aγ, d]同处于合并前的某一集合中。然而,这又意味着,原来的 LR(1)项目集就已经存在着移进—归约冲突,从而文法不是 LR(1)的。因此,同心集的合并不会引起新的移进—归约冲突。

但是,同心集的合并有可能产生新的归约-归约冲突,如下面例子所示。

例3.34 考虑文法

$S' \rightarrow S$

$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$

$A \rightarrow c$

$B \rightarrow c$

它只产生四个串:acd、bcd、ace和 bce。构造该文法的 LR(1)项目集,可以看出没有冲突,它是 LR(1)文法。在它的项目集中,对活前缀 ac有效的项目集为{[A→c·, d], [B→c·, e]},对 bc有效的项目集为{[A→c·, e], [B→c·, d]},这两集合都不含冲突,它们是同心的。然而,它们合并后变成{[A→c·, d/e], [B→c·, d/e]},它产生归约-归约冲突,因为面临 c或 d时,不知道应该用哪个产生式进行归约。

下面,我们给出构造 LALR分析表的算法,其基本思想是,首先构造 LR(1)项目集规范族,如果它不存在冲突,则把同心集合并在一起,再按合并后的项目集构造分析表。这个方法可以作为描述 LALR(1)文法的基本定义。在实际使用中,由于构造完整的 LR(1)项目集规范族需要很多的空间和时间,因而需要另找算法。

算法3.7 一个简易但耗空间的 LALR分析表构造法。

输入 拓广文法 G'。

输出 G'的 LALR分析表的 action函数和 goto函数。

方法 (1) 构造 LR(1)项目集规范族 $C=\{I_0, I_1, \cdots, I_n\}$。

(2) 寻找 LR(1)项目集规范族中同心的项目集,用它们的并集代替它们。

(3) 令 $C'=\{J_0, J_1, \cdots, J_m\}$是合并后的 LR(1)项目集族。action函数在状态 i的值以与算法3.6同样的方式从 J_i构造。如果出现分析动作的冲突,则算法不能产生分析表,此文法不是 LALR(1)的。

(4) goto函数在状态i的值如下确定：如果 J_i 是若干个 LR(1) 项目集的并，即 $J_i = I_1 \cup I_2 \cup \cdots \cup I_m$，那么 goto($I_1$, X)，goto($I_2$, X)，…，goto($I_m$, X) 也同心，因为 I_1，I_2，…，I_m 都同心。记 J_k 为所有和 goto(I_i, X) 同心的项目集的并，那么，goto(i, X) = k。

由算法3.7产生的表叫做 *G的* LALR 分析表。如果没有分析动作的冲突，那么该文法叫做 LALR(1) 文法。在第(3)步构造的项目集族叫做 LALR(1) 项目集族。

例3.35 再次考虑文法(3.13)，它的转移图在图3.19。如我们已提到的那样，有3对项目集可以合并，I_3和 I_6 合并成

I_{36}：B→*b*·*B*, *b*/*a*/$
B→·*bB*, *b*/*a*/$
B→·*a*, *b*/*a*/$

I_4和 I_7 合并成

I_{47}：B→*a*·, b/a/$

I_8和 I_9 合并成

I_{89}：B→*bB*·, b/a/$

压缩项目集后的 LALR 动作函数和移转函数如表3.10所示。

表3.10 文法(3.13)的 LALR 分析表

状态	动作			转移	
	b	*a*	$	*S*	*B*
0	*s*36	*s*47		1	2
1			*acc*		
2	*s*36	*s*47			5
36	*s*36	*s*47			89
47	*r*3	*r*3	*r*3		
5			*r*1		
89	*r*2	*r*2	*r*2		

现在来看看转移函数是怎样计算的。考虑 goto(I_{36}, B)，在原来的 LR(1) 项目集中，*goto*(I_3, B) = I_8，而 I_8 现在是 I_{89}的一部分。因此，置 goto(I_{36}, B) = I_{89}。如果考虑 I_{36}的另一部分 I_6，可以得到同样的结论，即 goto(I_6, B) = I_9，而 I_9 现在是 I_{89}的一部分。又例如，考虑 goto(I_2, B)，它指出了在面对 b 执行了 I_2 的移进动作之后的转移方向。原来的 LR(1) 项目集中，goto(I_2, B) = I_5，因为 I_5 现在是 I_{36}的一部分，所以 goto(I_2, B) = I_{36}。于是，分析表中状态2面对 *b的条目是 s*36，其含意是移进 b，再把状态36压在栈顶。

当输入串为 b∗ *ab*∗ *a 时，不论是表*3.9的 LR 分析器还是表3.10的 LALR 分析器，都

给出了同样的移进—归约序列,其差别只是状态名不同而已。对于 LALR 文法,这种关系总是保持,即只要输入串正确。LR 和 LALR 分析始终形影相随。

当输入串有错误时,LALR 分析可能比 LR 分析多做了一些不必要的归约,但 LALR 分析决不会比 LR 分析移进更多的符号。即就准确地指出输入串的出错位置而言,LALR 分析和 LR 分析是等效的。例如,若输入串是 bba 时,LALR 分析的动作序列见表 3.11。

表 3.11 LALR 分析器对于输入 *bba* 的格局

栈	输入	动作
(1) 0	*bba*$	移进
(2) 0 *b*3 6	*ba*$	移进
(3) 0 *b*3 6 *b*3 6	*a*$	移进
(4) 0 *b*3 6 *b*3 6 *a*4 7	$	按 B→*a* 归约
(5) 0 *b*3 6 *b*3 6 *B*8 9	$	按 B→*bB* 归约
(6) 0 *b*3 6 *B*8 9	$	按 B→*bB* 归约
(7) 0 *B*2	$	报告错误

而如果用 LR 分析,分析器将把

0 *b*3 *b*3 *a*4

推进栈,并在状态 4 发现错误,因为状态 4 面临$的动作是“出错”。两者的区别是 LR 分析及时发现错误。

可以修改算法 3.7,避免在建立 LALR(1) 分析表的过程中构造完整的 LR(1) 项目集规范族,以得到高效的 LALR 分析表构造算法。有兴趣的读者可以查阅相关书籍。

例 3.34 的文法是规范的 LR(1) 的,但不是 LALR(1) 的。例 3.29 的文法是 LALR(1) 的,但不是 SLR(1) 的。这两个例子说明这三类文法的集合是不同的。

3.5.6 非 LR 的上下文无关结构

在 3.5.2 节曾经提到,若自左向右扫描的移进—归约分析器能及时识别出现在栈顶的句柄,那么相应的文法就是 LR 的。而二义文法一定不是 LR 的,那么是否存在非二义并且非 LR 的文法呢?可以通过例子来说明问题。

例 3.36 语言 $L=\{ww^R \mid w\ ((a|b)^*)\}$ 的文法

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

不是 LR 的。直观上说,对于这个语言的任何一个句子,如 abaaba,扫描前一半字符时应该压栈,扫描后一半字符时先做空归约,然后将剩余字符和栈中的字符通过归约进行比较,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。事实上,对于任意大的 k,总能找到一个句子,即使是向前看 k 个字符也无法判断是否应该做空归约了。因此该文法不是 LR(k)的。

例 3.37 为语言

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

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。

该语言的句子是 aa…abb…b 的形式,但后面 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,那么应该有形如 $S\rightarrow aSb$ 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 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。

如果将 $a^m b^n$ 的前 m 个 b 和 a^m 配对,如图 3.20 所示,那么按此特点写出的文法

$$S\rightarrow AB$$
$$A\rightarrow aAb \mid \varepsilon$$
$$B\rightarrow Bb \mid b$$

是 LR(1)文法。

如果将 $a^m b^n$ 的后 m 个 b 和 a^m 配对,如图 3.21 所示,那么按此特点写出的文法

$$S\rightarrow aSb \mid B$$
$$B\rightarrow Bb \mid b$$

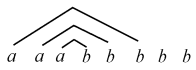


图 3.20 一种配对方式

图 3.21 另一种配对方式

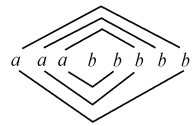


图 3.22 第三种配对方式

是非二义且非 LR(1)文法。因为在分析时,当所有的 a 进栈后,要将中间的若干个 b 归约成 B,使得剩下的 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栈顶的 b 归约成 B 还是移进下一个 b。

如果让 $a^m b^n$ 中的 a 和 b 有不止一种配对方式,如图 3.22 所示,那么它的文法就是二义的,如

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

3.6 二义文法的应用

任何二义文法决不是 LR 文法,因而不属于上节所讨论的任何一类文法,这是一条定理。但是,正如在这一节将要看到的,某些二义文法对说明和实现语言是有用的。像表达式这样的语言结构,二义文法提供的说明比任何其他非二义文法提供的都要简短些,更自然些。另外,为了便于对一些特殊情况进行优化,需要在文法中增加特殊情况产生式,以便把它们从一般结构中分离出来,这种产生式的加入会使文法二义,这是二义文法的另一应用。

必须强调,虽然使用的文法是二义的,但若在所有情况下都说明了消除二义的一些规则,以保证每个句子正好只有一棵分析树,那么整个语言的说明仍然是无二义的。

3.6.1 使用文法以外的信息来解决分析动作的冲突

我们考虑程序设计语言的表达式。下面有算符+和*的算术表达式文法是二义的,因为它没有指出算符+和*的结合性和优先级:

$$E \to E + E \mid E * E \mid (E) \mid \mathrm{id} \tag{3.14}$$

无二义的文法

$$\begin{aligned} &E \to E + T \mid T \\ &T \to T * F \mid F \\ &F \to (E) \mid \mathrm{id} \end{aligned} \tag{3.15}$$

产生同样的语言,但给+以较低的优先级,并让两个算符都是左结合的。有两点理由说明可能使用文法(3.14)而不是(3.15)。首先,如我们将要看到的那样,算符+和*的结合性和优先级可以方便地改变而无须修改文法(3.14),也不会改变分析器的状态数。其次,文法(3.15)的分析器要花一部分时间来完成产生式 $E \to T$ 和 $T \to F$ *的归约,而文法*(3.14)的分析器不会消耗时间在归约这样的单非产生式(右部只有一个非终结符产生式)上。

文法(3.14)用 $E' \to E$ *拓广后的* LR(0)项目集在图3.23。因为文法(3.14)二义,因此从这些项目集产生 LR 分析表时,肯定会出现分析动作的冲突,冲突出现在项目集 I_7 和 I_8 对应的状态。假如我们用 SLR 方法来构造动作表,I_7 产生的冲突在 $E \to E + E$ *引起的归约和*面临+和*的移进之间,因为+和*都在 *FOLLOW*(*E*)*中*。I_8 产生的冲突在 $E \to E * E$ *引*起归约和面临+和*的移进之间。事实上,用任何一种 LR 分析表的构造方法都会产生这些冲突。

I_0：E′→·E
E→·E+E
E→·E*E
E→·(E)
E→·id

I_1：E′→E·
E→E·+E
E→E·*E

I_2：E′→(·E)
E→·E+E
E→·E*E
E→·(E)
E→·id

I_3：E→id·

I_4：E→E+·E
E→·E+E
E→·E*E
E→·(E)
E→·id

I_5：E→E*·E
E→·E+E
E→·E*E
F→·(E)
F→·id

I_6：E→(E·)
E→E·+E
E→E·*E

I_7：E→E+E·
E→E·+E
E→E·*E

I_8：E→E*E·
E→E·+E
E→E·*E

I_9：E→(E)·

图 3.23 文法 (3.14) 拓广后的 LR(0) 项目集

这些冲突可以用有关+和*的优先级和结合性的信息来解决。考虑输入 id + id * id，基于图 3.23 的分析器在处理 id + id 后进入状态 7，形成如下格局：

栈	输入
0 E 1 + 4 E 7	* id $

如果*的优先级高于+，分析器应把*移进栈，准备归约*和它两侧的 id 到一个表达式。这正是该语言的 SLR 分析器（表 3.7）要做的。另一方面，如果+的优先级高于*，那么分析器应该归约。这样，根据+和*的优先关系就可以解决状态 7 的 *E→E+E 归约和面临*的移进之间的冲突。

如果输入是 id + id + id 的话，分析器处理完 id + id 后到达的格局和上面的惟一区别是下一个输入符号是+而不是*。在状态 7 面临+时仍有移进—归约冲突，现在是由算符+的结合性来解决冲突。如果+是左结合，正确的动作是按 E→E+E *归约，即第一个*+及其前后的 id 应看成一组。这个选择和例 3.24 文法的 SLR 分析器的动作是一致的。

总之,假如+是左结合的,那么在状态7面临+时应该按 $E \to E+E$ *归约;如果*×的优先级高于+,那么在状态7面临×时应该移进。可以类似地讨论状态8,最后得出如下结果 如果×是左结合且优先级高于+,那么不论面临+还是×,分析器在状态8的动作都是按 $E \to E \times E$ *归约*。

按这种方式处理,可以得到表3.12的LR分析表,产生式(1)到产生式(4)分别是 $E \to E+E$, $E \to E \times E$, $E \to (E)$ 和 $E \to$ id。有趣的是,类似的动作表可以从表3.7的SLR表中删去文法(3.15)的单非产生式 $E \to T$ *和* $T \to F$ *的归约得到。*(3.14)这样的二义文法可用类似的方法在构造LALR分析表和规范的LR分析表中处理。

表3.12 文法(3.14)的分析表

状态	动作						转移
	id	+	×	(	)	$	*E*
0	*s*3			*s*2			1
1		*s*4	*s*5			*acc*	
2	*s*3			*s*2			6
3		*r*4	*r*4		*r*4	*r*4	
4	*s*3			*s*2			7
5	*s*3			*s*2			8
6		*s*4	*s*5		*s*9		
7		*r*1	*s*5		*r*1	*r*1	
8		*r*2	*r*2		*r*2	*r*2	
9		*r*3	*r*3		*r*3	*r*3	

再考虑下面的条件语句文法:

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

可知该文法是二义的,因为它没有解决悬空else的二义性。可以肯定,构造LR分析表时,会碰到移进—归约冲突,即,当 if *expr* then *stmt 在栈顶,并且* else 是下一个输入符号时,究竟是将 if *expr* then *stmt 归约还是将* else 移进。根据语言关于else的配对规则,我们知道,对于这种移进—归约冲突,忽略归约,采用移进,即优先移进。

3.6.2 特殊情况产生式引起的二义性

如果需要引入额外的产生式来表示由其余的产生式产生的语法结构的一种特殊情况

时,文法会因加入了这额外的产生式而引起二义性,从而引起分析动作的冲突。我们先举一个这种文法的例子。

历史上,公式编排预处理器 EQN 中使用了特殊情况产生式,这是一个有趣的应用。在 EQN 中,描述数学表达式的文法用算符 sub 表示下角标并用算符 sup 表示上角标,这个文法的片断见(3.16)。花括号由预处理器用来表示复合表达式,c 作为表示任意正文串的记号。

(1) E→*E* sub *E* sup *E*
(2) E→*E* sub *E*
(3) E→*E* sup *E* (3.16)
(4) E→{E}
(5) E→*c*

文法(3.16)是二义的。该文法没有说明算符 sub 和 sup 的结合性和优先级。即使由它们引起的二义性解决了,比方规定这两个算符的优先级相同并且都是右结合的,该文法仍然是二义的。这是因为产生式(1)分离出了由产生式(2)和(3)产生的表达式的一种特例,即形式为 E sub *E* sup *E* 的表达式。*没有产生式*(1),该文法产生的语言是一样的。把这种形式的表达式处理为一种特殊情况的理由是,像 a sub *i* sup 2 这样的表达式应该排版成 a_i^2,而不是 a_i^2 或 a_2 的形式。只有加上特殊情况产生式后,EQN 才能够产生这样特殊的输出。

如果构造该文法的 LR 分析表,我们会发现,存在移进—归约冲突和归约－归约冲突。其中移进—归约冲突可以根据 sub 和 sup 这两个算符的优先级和结合性来解决。归约－归约冲突在产生式

E→*E* sub *E* sup *E*
E→*E* sup *E*

之间。即当 E sub *E* sup *E* *出现在栈顶时,我们是按前一个产生式将* E sub *E* sup *E* *归约,*还是按后一个产生式将 E sup *E* *归约。显然,应该优先特殊情况产生式,即按前一个产生式*归约。这样,和该特殊情况产生式相联的语义动作可以用更专门的措施来产生这样特定的输出。

写一个分离特殊情况语法结构的无二义文法是非常困难的。为了体会这是何等的困难,请读者为文法(3.16)构造等价的无二义文法,它分离形式为 E sub *E* sup *E* *的表达式*。

3.6.3 LR 分析的错误恢复

LR 分析器在访问动作表时若遇到出错条目,那么它就发现了错误。但是在访问转移表时它决不会遇到出错条目。只要已扫描的输入出现一个不正确的后继,LR 分析器便立即报告错误,决不会把不正确的后继移进栈。规范的 LR 分析器甚至在报告错误之前决不做任

何无效归约,但 SLR 和 LALR 分析器在报告错误前有可能执行几步这样的归约。

在 LR 分析中,可以如下实现紧急方式的错误恢复:从栈顶开始退栈,直至出现状态 s,它有预先确定的非终结符 A 的转移;然后抛弃若干个(可以是零个)输入符号,直至找到符号 a,它能合法地跟随 A;分析器再把 A 和状态 goto[s, A]压进栈,恢复正常分析。A 的选择可能不惟一,一般来说 A 应是代表较大程序结构的非终结符,如表达式、语句或程序块。例如,若 A 是非终结符 stmt,那么 a 可以是分号或 end。

这种错误恢复方法的实质是试图分离含错的语法短语。分析器认为由 A 推出的串含有一个错误,该串的一部分已经处理,这个处理的结果是若干状态已加到栈顶。这个串的其余部分仍在剩余输入中。分析器试图跳过这个串的其余部分,在剩余输入中找到一个符号,它能合法地跟随 A。通过从栈中移出一些状态,跳过若干输入符号,把 goto[s, A]推进栈,分析器装扮成已发现了 A 的一个实例,并恢复正常分析。

错误恢复的另一种方式叫做短语级恢复。当发现错误时,分析器对剩余输入作局部纠正,它用可以使分析器继续分析的串来代替剩余输入的前缀。典型的局部纠正是用分号代替逗号,删除多余的分号,或插入遗漏的分号等。编译器的设计者必须仔细选择替换的串,以免引起死循环。死循环是可能的,例如,总是在当前输入符号的前面插入一些东西。

这种替换可以纠正任何输入串,但是它的主要缺点是很难应付实际错误出现在诊断点以前的情况。

对 LR 分析来说,短语级恢复的实现由考察 LR 分析表的每一个错误条目并根据语言的使用情况,决定最可能进入该条目的输入错误,然后为该条目编一个适当的错误恢复过程。

可以在分析表动作域的每个空白条目填上一个指示器,它指向编译器设计者为之设计的错误处理例程。该例程的动作包括在输入中插入、删除或改变输入符号等。要注意,所做的选择不应引起 LR 分析器进入无限循环。保证至少有一输入符号被删除或最终被移进,在到达输入的末尾时保证栈最终会缩短的策略就足以防止这个问题。

例 3.38 考虑表达式文法

$$E \to E + E \mid E * E \mid (E) \mid \mathrm{id}$$

表 3.13 给出了这个文法的 LR 分析表,它在表 3.12 的基础上加了错误诊断和恢复。我们把某些错误条目改成了归约。这样修改会推迟错误的发现,多执行了一步或几步归约,但错误仍在移进下一个符号前被捕获。表 3.12 的其余空白条目已经改成了调用错误处理例程。

错误处理例程如下:

```
e1:/* 分析器处于状态 0,2,4 或 5 时,要求输入符号为运算对象首符,即 id 或左括号。
若遇到的是+,*或$时,调用此例程。*/
```

把一个假想的 id 压进栈,上面盖上状态 3(状态 0,2,4 和 5 面临 id 时的转移)。给出诊断信息“缺少运算对象”。

表 3.13 有错误处理例程的 LR 分析表

状态	动作						转移
	id	+	*	(	)	$	E
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	acc	
2	s3	e1	e1	s2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	s3	e1	e1	s2	e2	e1	7
5	s3	e1	e1	s2	e2	e1	8
6	e3	s4	s5	e3	s9	e4	
7	r1	r1	s5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

e2：/* 分析器处于状态 0,1,2,4 或 5,遇到右括号时调用此例程。*/

删除输入右括号。给出诊断信息“不配对的右括号”。

e3：/* 分析器处于状态 1 或 6,期望运算符,但遇到的却是 id 或左括号时,调用此例程。*/

把+压进栈,盖上状态 4。给出诊断信息“缺少算符”。

e4：/* 分析器处于状态 6,期望运算符或右括号,但遇到的却是$时,调用此例程。*/

把右括号压入栈,盖上状态 9。给出诊断信息“缺少右括号”。

读者可以用一个有语法错误的简短表达式为例,体会该错误恢复方法的效果。

3.7 分析器的生成器

本节说明分析器的生成器如何用来帮助构造编译器的前端。我们将 LALR 分析器的生成器 Yacc (Yet Another Compiler-Compiler) 作为讨论的基础,因为它实现了前两节讨论的许多概念,并且使用广泛。Yacc 是上世纪 70 年代初期分析器的生成器盛行时的产物,它已经被用来帮助实现了几百个编译器,现在它仍然是 UNIX 系统下的一个好工具。

3.7.1 分析器的生成器 Yacc

一个翻译器可用 Yacc 按图 3.24 表示的方式构造出来。首先,用 Yacc 语言将翻译器的说明建立于一个文件 (例如 translate.y) 中。UNIX 系统的命令

yacc translate.y

把文件translate.y翻译为C语言文件,叫做y.tab.c,它使用的是LALR方法。程序y.tab.c包含用C写的LALR分析器和其他用户准备的C语言例程。为了使LALR分析表少占空间,紧凑技术被用来压缩分析表的大小。

然后,再用命令

cc y.tab.c -ly

编译y.tab.c,其中的选择项ly表示使用LR分析器的库(名字ly随系统而定),编译的结果是目标程序a.out,该目标程序能完成上面的Yacc程序指定的翻译。如果还需要其他过程的话,它们可以和y.tab.c一起编译或装载,就和使用C程序一样。

Yacc源程序由三部分组成:

声明

%%

翻译规则

%%

用C语言编写的支持例程

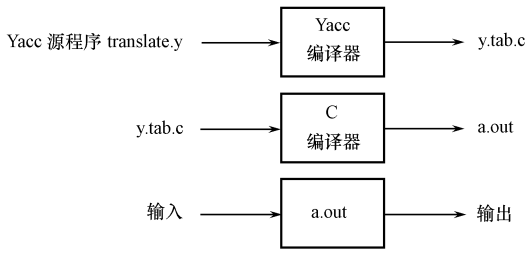


图3.24 用Yacc建立翻译器

例3.39 以构造一个简单的台式计算器为例,说明怎样准备Yacc源程序。该台式计算器读一个算术表达式,计算并打印它的值。构造该台式计算器从下面算术表达式的文法开始:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \text{digit}$

记号digit是0~9之间的单个数字。基于这个文法的Yacc台式计算器程序见图3.25。

Yacc程序的声明部分有可选择的两节。第一节处于分界符%{和%}之间,它是一些普通的C语言的声明,这里声明的常量和变量等由第二部分和第三部分的翻译规则或过程使用。图3.25中,这一节只有一个包含语句

include <ctype.h>

因为这个文件含有谓词isdigit。

声明部分的第二节是文法终结符(即词法记号)的声明,图3.25的语句

%token DIGIT

声明DIGIT是记号。这一节声明的记号可用于Yacc程序的第二部分和第三部分。

```
%{
#include <ctype.h>
%}
%token DIGIT
%%
line    : expr \n          {printf ( %d\n , $1) ;}
        ;
expr    : expr + term      {$$= $1 + $3 ;}
        | term
        ;
term    : term * factor    {$$= $1 * $3 ;}
        | factor
        ;
factor  : ( expr )         {$$= $2 ;}
        | DIGIT
        ;
%%
yylex () {
    int c ;
    c= getchar () ;
    if (isdigit (c)) {
        yylval= c- 0 ;
        return DIGIT ;
    }
    return c ;
}
```

图 3.25 简单台式计算器的 Yacc 说明

Yacc 程序的第二部分位于第一个%%后面,放置翻译规则,每条规则由一个文法产生式和有关的语义动作组成。产生式集合

左部→选择 1|选择 2|…|选择 *n*

在 Yacc 中写成

```
左部 :选择 1 {语义动作 1}
     | 选择 2 {语义动作 2}
     …
```

```
        | 选择 n {语义动作 n}
        ;
```

在 Yacc 产生式中,加单引号的字符 c 是由单个字符 c 组成的记号;没有引号的字母数字串,若也没有声明为记号,则是非终结符。右部的各个选择之间用竖线隔开,最后一个右部的后面用分号,表示该产生式集合结束。第一个左部非终结符是开始符号。

Yacc 的语义动作是 C 语句序列。在语义动作中,符号$$表示引用左部非终结符的属性值,而$*i* 表示引用右部第 *i* 个文法符号的属性值。每当归约一个产生式时,执行与之关联的语义动作,所以语义动作一般是从各$*i* 的值决定$$的值。在这个 Yacc 说明中,两个 *E* 产生式

$$E \rightarrow E + T \mid T$$

及和它们相关的语义动作写成

```
expr : expr + term      {$$ = $1 + $3;}
     | term
     ;
```

注意,在第一个产生式中,非终符 term 是右部的第三个文法符号,+ 是第二个文法符号。第一个产生式的语义动作是把右部 expr 的值和 term 的值相加,把结果赋给左部非终结符 expr 作为它的值。第二个产生式的语义动作描述省略,因为当右部只有一个文法符号时,语义动作缺省就是表示值的复写,即它的语义动作是 {$$ = $1;}。

注意,我们加了一个新的开始产生式

```
line : expr \n  {printf ("%d\n", $1);}
```

到这个 Yacc 程序。该产生式的意思是,这个台式计算器的输入是一个表达式后面跟一个换行字符。它的语义动作是打印表达式的十进制值并且换行。

Yacc 程序的第三部分是一些 C 语言写的支持例程。名字为 yylex() 的词法分析器必须提供(当然也可以用 Lex 来产生 yylex()),其他的过程,如错误恢复例程,需要的话,也可以加上。

词法分析器 yylex() 返回二元组(记号,属性)。返回的记号类别,如 DIGIT,必须在 Yacc 程序的第一部分声明。属性值必须通过 Yacc 定义的变量 yylval 传给分析器。

图 3.25 的词法分析器是非常粗糙的。它用 C 语言的函数 getchar() 每次读一个输入字符,如果是数字字符,取它的值存入变量 yylval,返回记号 DIGIT,否则把字符本身作为记号返回。若输入中有非法字符的话,它会引起分析器宣布一个错误而停机。

3.7.2 用 Yacc 处理二义文法

修改上节的 Yacc 程序,使台式计算器更加有用。首先,让台式计算器计算一列表达式,每行一个,也允许表达式之间有空白行。为做到这样,改第一个规则为

```
lines : lines expr \n       {printf ("%g\n", $2);}
```

```
|lines \n
|
;
```

按照 Yacc 的规定,第三行的空选择表示 ε。

其次,允许表达式使用多个数字组成的数,并将算符增加到包括+,-(一元和二元),*和/。这一回用二义文法

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid - E \mid \text{number}$$

来描述表达式。最终的 Yacc 程序见图 3.26。

```
%{
# include <ctype.h>
# include <stdio.h>
# define YYSTYPE double    /* 将 Yacc 栈定义为 double 类型 */
%}

%token NUMBER
%left + -
%left * /
%right UMINUS
%%
lines   : lines expr \n          {printf("%g\n",$2);}
        | lines \n
        | /*ε*/
        ;
expr    : expr + expr            {$$=$1+$3;}
        | expr - expr            {$$=$1-$3;}
        | expr * expr            {$$=$1*$3;}
        | expr / expr            {$$=$1/$3;}
        | (expr)                 {$$=$2;}
        | - expr %prec UMINUS    {$$=-$2;}
        | NUMBER
        ;
%%
yylex() {
    int c;
    while((c=getchar())==' ');
```

```
    if ((c == '.') || (isdigit(c))) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    return c;
}
```

图3.26 更高级的台式计算器的Yacc说明

因为图3.26的Yacc程序的文法是二义的,LALR算法将产生分析动作的冲突。Yacc会报告产生的分析动作冲突的数目。可以用编译选项-V获得产生的项目集和分析动作冲突的描述,这些信息在一个附加的文件y.output中,该文件还包含了LR分析表的可读表示,以及Yacc是怎样解决这些分析动作的冲突的。

当Yacc报告它发现分析动作冲突时,明智的做法是建立和查阅文件y.output,以明白为什么会出现分析动作的冲突和它们是否已经得到正确解决。

除非另有说明,否则Yacc按下面两条规则解决分析动作的冲突:

(1) 对于归约-归约冲突,选择在Yacc程序中最先出现的那个冲突产生式。按此规则,为了正确解决排版文法(3.16)的冲突,只要把产生式(1)放在产生式(3)前面就足够了。

(2) 对于移进-归约冲突,优先于移进。这条规则正确地解决了悬空else的移进-归约冲突。

由于这些缺省的规则并不总是编译器编写者所想要的,因而Yacc允许编译器编写者提供一些解决移进-归约冲突的说明。在声明部分,可以为终结符指定优先级和结合性。声明

```
%left '+' '-'
```

表示+和-有同样的优先级并且它们是左结合的。声明

```
%right '↑'
```

表示算符↑为右结合。还可以用声明限制两元算符为不可结合的(即该算符的两个相邻出现根本不被允许),如

```
%nonassoc '<'
```

终结符的优先级按它们在声明部分出现的次序而定,先出现的优先级低,同一声明中的终结符有相同的优先级。这样,图3.26的声明

```
%right UMINUS
```

使得UMINUS的优先级高于前面5个终结符。

Yacc解决移进-归约冲突时,首先参考这个冲突涉及的产生式和终结符的优先级和结

合性。通常,产生式的优先级和结合性同它最右边终结符的一致,在大多数情况下,这是合理的决策。

如果 Yacc 必须在移进输入符号 a 和按产生式 $A \rightarrow \alpha$ *归约这两个动作之间进行选择的*话,那么当这个产生式的优先级高于 a,或者优先级相同但产生式左结合时,取归约动作,否则选择移进。例如,给定产生式

$$E \rightarrow E + E \mid E * E$$

若搜索符是+,归约项目的产生式是 $E \rightarrow E + E$,*那么优先于归约,因为右部的*+和搜索符有同样的优先级,而+是左结合的。如果搜索符是*,那么取移进,因为搜索符的优先级高于这个产生式中+的优先级。

在那些最右终结符不能给产生式以适当优先级和结合性的情况下,我们可以给产生式附加标记

%prec 终结符

来强制,使得它的优先级和结合性同该标记终结符的一样,这个终结符可以仅仅是个占位符,就像图 3.26 的 UMINUS 那样,它不由词法分析器返回,仅用来决定一个产生式的优先级和结合性。图 3.26 中,声明

%right UMINUS

给记号 UMINUS 指定高于*和/的优先级。在翻译规则部分,标记

%prec UMINUS

在产生式

expr : - expr

的后面,它使得该产生式的一元减算符的优先级高于其他任何算符。

Yacc 不向编译器设计者报告用这种优先级和结合性能解决的移进—归约冲突。

3.7.3 Yacc 的错误恢复

前面介绍的短语级错误恢复方法显然不适用于 Yacc 这样分析器自动生成的情况。Yacc 用的是紧急方式的错误恢复思想。首先,它要求编译器设计者决定哪些"主要的"非终符将有错误恢复与它们相关联,这些非终结符的典型选择是用于产生表达式、语句、程序块和过程的那些非终结符。然后编译器设计者把形式为 $A \rightarrow$ error α *的*错误产生式加到文法上,其中 A 是主要非终结符,α 是文法符号串,也可能是空串,error 是 Yacc 保留字。Yacc 将从这样的说明产生分析器,把错误产生式当作普通产生式处理。

当 Yacc 产生的分析器遇到错误时,它用特别的方式来处理其项目集含错误产生式的状态。遇到错误时。Yacc 从栈中弹出状态,直到发现栈顶状态的项目集包含形为 $A \rightarrow \cdot$error α *的项目为止。然后分析器把虚构的终结符* error"移进"栈,好像它在输入中看见了这个终

结符。

当 α为ε时,立即进行对 A的归约并执行产生式A→error 的语义动作(它可能是报告错误信息并设置标记禁止生成目标代码)。然后分析器抛弃若干输入符号直到发现一个能回到正常处理的输入符号为止。

如果 α非空,Yacc 就在输入串上向前寻找能够归约为 α的子串。如果 α含的都是终结符,那么它在输入上寻找这样的串,把其中的符号移进栈,这时,error *α在分析器的栈顶。*随后分析器把 error *α归约为A,恢复正常分析。*

例如,出错产生式

stmt→error ;

要求分析器看见错误时跳过下一个分号,好像这个语句已经看见一样。

例 3.40 图3.27 的 Yacc 程序在图3.26 的基础上增加了错误产生式。错误产生式为

lines :error \n

当输入行有语法错误时,分析器从栈中弹出状态,直至碰到一个有移进 error 动作的状态。状态0是惟一的这种状态,因为它的项目包含

lines :error \n

状态0总是在栈底。分析器把终结符 error 移进栈,废弃输入符号,直至发现换行字符。然后分析器把换行符移进栈,把 error \n 归约成 lines,输出诊断信息"重新输入上一行"。专门的 Yacc 例程 yyerrok 用于使分析器回到正常操作方式。

```
%{
# include <ctype.h>
# include <stdio.h>
# define YYSTYPE double /* 将 Yacc 栈定义为 double 类型 */
%}

%token NUMBER
%left + -
%left * /
%right UMINUS
%%
lines   : lines expr \n          {printf ( %g\n , $2) }
        | lines \n
        | /* ε */
        | error \n {printf ( 重新输入上一行: ); yyerrok }
        ;
```

```
expr    : expr  +  expr              {$$ = $1  +  $3 ;}
        | expr  -  expr              {$$ = $1  -  $3 ;}
        | expr  *  expr              {$$ = $1  *  $3 ;}
        | expr  /  expr              {$$ = $1 / $3 ;}
        |  ( expr )                  {$$ = $2 ;}
        |  -  expr %prec UMINUS      {$$ = -$2 ;}
        | NUMBER
        ;
%%
yylex ()  {
    int c ;
    while ( (c = getchar () ) ==    ) ;
    if ( (c ==   . ) || (isdigit (c) ) )  {
        ungetc (c, stdin) ;
        scanf ( %lf , &yylval) ;
        return NUMBER ;
    }
    return c ;
}
```

图 3.27 有错误恢复的台式计算器

习 题 3

3.1 考虑文法

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

(a) 建立句子 $(a, (a, a))$ 和 $(a, ((a, a), (a, a)))$ 的分析树。

(b) 为 (a) 的两个句子构造最左推导。

(c) 为 (a) 的两个句子构造最右推导。

(d) 这个文法产生的语言是什么？

3.2 考虑文法

$S \rightarrow aSbS \mid bSaS \mid \varepsilon$

(a) 为句子 *abab* 构造两个不同的最左推导，以此说明该文法是二义的。

(b) 为 *abab* 构造对应的最右推导。

(c) 为 *abab* 构造对应的分析树。

(d) 这个文法产生的语言是什么？

3.3 下面的二义文法描述命题演算公式，为它写一个等价的非二义文法。

$S \rightarrow S$ and $S \mid S$ or $S \mid$ not $S \mid$ true $\mid$ false $\mid (S)$

3.4 文法

$R \rightarrow R \mid R \mid RR \mid R* \mid (R) \mid a \mid b$

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。

(a) 证明该文法产生字母表 {a, b} 上的所有正规式。

(b) 为该文法写一个等价的非二义文法。它给予算符 *、连接和 | 的优先级和结合性同 2.2 节中定义的一致。

(c) 按上面两个文法构造句子 $ab \mid b* a$ 的分析树。

3.5 下面的条件语句文法

$stmt \rightarrow$ if $expr$ then $stmt \mid matched\_stmt$

$matched\_stmt \rightarrow$ if $expr$ then $matched\_stmt$ else $stmt \mid other$

试图消除悬空 else 的二义性，请你证明该文法仍然是二义的。

3.6 为字母表 $\Sigma = \{a, b\}$ 上的下列每个语言设计一个文法，其中哪些语言是正规的？

(a) 每个 a 后面至少有一个 b 跟随的所有串。

(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) 不含 abb 作为子串的所有串。

* (e) 形式为 xy 且 $x \neq y$ 的所有串。

3.7 我们可以在文法产生式的右部使用类似正规式的算符。方括号可以用来表示产生式的可选部分，例如可以用

$stmt \rightarrow$ if $expr$ then $stmt$ [else $stmt$]

表示 else 子句是可选的。通常，$A \rightarrow \alpha [\beta] \gamma$ 等价于两个产生 $A \rightarrow \alpha\beta\gamma$ 和 $A \rightarrow \alpha\gamma$。

花括号用来表示短语可重复出现若干次（包括零次），例如

$stmt \rightarrow$ begin $stmt$ {; $stmt$} end

表示处于 begin 和 end 之间的由分号分隔的语句表。通常，$A \rightarrow \alpha \{\beta\} \gamma$ 等价于 $A \rightarrow \alpha\beta\gamma$ 和 $B \rightarrow \beta B \mid \varepsilon$。

概念上，[] 代表正规式 $B \mid \varepsilon$，{} 代表 β^*，现在我们把它们推广为允许文法符号的任何正规式出现在产生式的右部。

(a) 修改上面的 stmt 产生式，使得每个语句都以分号终止的语句表出现在产生式右部。

(b) 给出上下文无关的产生式，它和 $A \rightarrow B^* a (C \mid D)$ 产生同样的串集。

(c) 说明如何用一组有限的上下文无关产生式来代替产生式 $A \rightarrow \gamma$，其中 γ 是正规式。

3.8 (a) 消除习题 3.1 文法的左递归。

(b) 为 (a) 的文法构造预测分析器。

3.9 为习题 3.3 的文法构造预测分析器。

3.10 构造下面文法的 LL(1)分析表。

$D \rightarrow TL$

$T \rightarrow \text{int} \mid \text{real}$

$L \rightarrow \text{id}\ R$

$R \rightarrow ,\ \text{id}\ R \mid \varepsilon$

3.11 下面的文法是否为 LL(1)文法?说明理由。

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

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

*3.12 证明左递归的文法不是 LL(1)文法。

*3.13 证明 LL(1)文法不是二义的。

3.14 证明没有 ε 产生式的文法,只要每个非终结符的各个选择以不同的终结符开始,那么它就是 LL(1)的。

3.15 (a) 用习题 3.1 的文法构造(a,(a,a))的最右推导,说出每个右句型的句柄。

(b) 给出对应(a)的最右推导的移进—归约分析器的步骤。

(c) 对照(b)的移进—归约,给出自下而上构造分析树的步骤。

3.16 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。

3.17 考虑文法

$S \rightarrow AS \mid R$

$A \rightarrow SA \mid a$

(a) 构造这个文法的 LR(0)项目集规范族。

(b) 构造一个 NFA,它的状态是(a)的 LR(0)项目。证明从这个 NFA 用子集法构造的 DFA 和该文法的 LR(0)项目集规范族的转移图是一致的。

(c) 构造此文法的 SLR 分析表。

(d) 给出针对输入 bab 的 SLR 分析器的动作。

(e) 构造规范的 LR 分析表。

(f) 构造 LALR 分析表。

3.18 为习题 3.3 的文法构造 SLR 分析器。

3.19 考虑下面的文法

$E \rightarrow E + T \mid T$

$T \rightarrow TF \mid F$

$F \rightarrow F^* \mid a \mid b$

(a) 为此文法构造 SLR 分析表。

(b) 构造 LALR 分析表。

3.20 (a) 证明下面文法

$S \to AaAb \mid BbBa$

$A \to \varepsilon$

$B \to \varepsilon$

是 LL (1) 文法,但不是 SLR (1) 文法。

* (6) 证明所有 LL (1) 文法都是 LR (1) 文法。

3.21 证明下面文法

$S \to (X \mid [F)$

$X \to E) \mid F]$

$E \to A$

$F \to A$

$A \to \varepsilon$

是 LL (1) 文法,但不是 LALR (1) 文法。

3.22 证明下面文法

$S \to Aa \mid bAc \mid Bc \mid bda$

$A \to d$

是 LALR (1) 文法,但不是 SLR (1) 文法。

3.23 证明下面文法

$S \to X$

$X \to Ma \mid bMc \mid dc \mid bda$

$M \to d$

是 LALR (1) 文法,但不是 SLR (1) 文法。

3.24 说明每个 SLR (1) 文法都是 LALR (1) 文法。

3.25 证明下面文法

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

$A \to d$

$B \to d$

是 LR (1) 文法,但不是 LALR (1) 文法。

3.26 一个非 LR (1) 的文法如下:

$L \to M\,L\,b \mid a$

$M \to \varepsilon$

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

3.27 文法 G 的产生式如下:

$S \to I \mid R \qquad I \to d \mid Id \qquad R \to WpF$

$W \to Wd \mid \varepsilon \qquad F \to Fd \mid d$

(a) 令 d表示任意数字,p表示十进制小数点,那么非终结符 S, I, R, W和 F 在程序设计语言中分别表示什么?

(b) 该文法是 LR(1) 文法吗？为什么？

3.28 下面文法不是 LR(1) 的，对它略作修改，使之成为一个等价的 SLR(1) 文法。

PROGRAM→ begin DECLIST semicolon STATELIST end

DECLIST→ *d* semicolon DECLIST | *d*

STATELIST→ *s* semicolon STATELIST | *s*

3.29 描述文法

S→ *aSbS* | *aS* | ε

产生的语言，并为此语言写一个 LR(1) 文法。

3.30 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，给出那个有移进－归约冲突的规范的 LR(1) 项目集。

S→ *aAc* *S*→ *aAc*

A→ *Abb* | *b* *A*→ *bAb* | *b*

3.31 为语言

L = {$a^m b^n$ | $0 \le m \le 2n$}（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。

3.32 为语言

L = {*w* | *w* ∈ (*a* | *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。

3.33 为习题 3.4 的文法构造 SLR(1) 分析表，分析动作冲突的解决要保证正规式能以正常的方式分析。

3.34 由于文法二义引起的 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，是否也可以依据什么规则来消除这种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？

**3.35 为排版文法（3.16）构造一个等价的 LR 文法，它能把形式为 E sub *E* sup *E* 的表达式处理为一种特殊情况。

3.36 写一个 Yacc 程序，它把输入的算术表达式翻译成对应的后缀表达式输出。

3.37 写一个 Yacc“台式计算器”程序，它计算布尔表达式。

3.38 写一个 Yacc 程序，它取正规式作为输入，产生它的分析树作为输出。

3.39 对于例 3.16 和例 3.38 的预测分析器和 LR 分析器，追踪它们面临下面有错输入的动作。

(a) (id + (* id)

(b) * + id) + (id *

*3.40 为下面的文法构造有短语级错误恢复的 LR 分析器。

stmt→ if *e* then *stmt*

| if *e* then *stmt* else *stmt*

| while *e* do *stmt*

```
        | begin list end
        | s
stmt → list ; stmt
     | stmt
```

3.41 一个 C 语言的文件如下：

```
long gcd (p,q)
long p,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：

如果缺少第二行的逗号，编译器报告 parse error before q (line 2)。如果缺少第四行的右括号，编译器报告 parse error before return (line 5)。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）。

如果第四行的 if 误写成 fi，编译器仍报告 parse error before return (line 5)。此时是否违反了活前缀性质。

第 4 章　语法制导的翻译

在 3.7 节用 Yacc 写的例子中,我们看到一种有用的描述形式:语言结构的属性附加在代表语言结构的文法符号上,这些属性值由附加在文法产生式的语义动作来计算,而语义动作在归约对应的产生式时进行计算,由此得到结果。这种描述形式可用来描述编译器的语义分析,因此本章系统地研究这种称之为"语法制导下的语言翻译"的描述方法及其实现。它的语义动作(有时称为语义规则)的计算可以产生代码、把信息存入符号表、显示出错信息或完成其他工作。语义规则的计算结果就是我们所要的记号流的翻译。

本章讨论语义规则和产生式相联系的两种方式:语法制导的定义和翻译方案。语法制导定义是较抽象的翻译说明,它隐蔽了一些实现细节;而翻译方案陈述了一些实现细节,主要是指明了语义规则的计算次序。在第 5 章说明语义检查和第 7 章描述中间代码生成时,大量使用这两种方法。

本章还讨论语法制导定义和翻译方案的实现方法。概念上的方法是,首先分析输入的记号串,建立分析树,然后从分析树得到描述结点属性间依赖关系的有向图,从这个依赖图得到语义规则的计算次序,然后进行计算,最终得到翻译的结果。实际的实现并不需要按上面步骤逐步进行,本章将讨论几种不同限制下的实现方法。

4.1　语法制导的定义

语法制导的定义是上下文无关文法的推广,其中每个文法符号都有一个属性集合,它分成两个子集,分别叫做该文法符号的综合属性集合和继承属性集合。如果把分析树上的结点看成是保存对应文法符号的属性的记录,那么属性对应记录的域。属性可以表示任何东西:串、数、类型、内存单元,或其他想表示的东西。分析树结点的属性值由该结点所用产生式的语义规则定义。在语法制导定义中,把其中的文法称为基础文法。

本节介绍语法制导定义的形式及其概念上的实现模型。

4.1.1　语法制导定义的形式

在语法制导定义中,每个文法符号有一组属性,每个文法产生式 $A \rightarrow \alpha$ 有一组形式为 b

≔ $f(c_1, c_2, \cdots, c_n)$ 的语义规则,其中 f 是函数,b 和 $c_1, c_2, \cdots, c_k$ 是该产生式的文法符号的属性,并且:

⑴ 如果 b 是 A 的属性,$c_1, c_2, \cdots, c_k$ 是产生式右部文法符号的属性或 A 的其他属性,那么 b 叫做文法符号 A 的综合属性。

⑵ 如果 b 是产生式右部某个文法符号 X 的属性,$c_1, c_2, \cdots, c_k$ 是 A 的属性或右部文法符号的属性,那么 b 叫做文法符号 X 的继承属性。

在这两种情况下,我们都说属性 b 依赖于属性 $c_1, c_2, \cdots, c_k$。每个文法符号的综合属性集和继承属性集的交集应为空。一般来说,结点的综合属性的值是通过分析树中它子结点的属性值来计算 继承属性的值由结点的兄弟结点和父结点的属性值来计算。

语义规则函数通常是表达式。有时,语法制导定义中某些规则的目的是要产生副作用,如打印值或修改全程量,这样的语义规则写成过程调用或程序段。可以把它们想像成定义了产生式左部非终结符的一个虚拟综合属性,这个虚拟属性和符号 ≔ 在该规则中没有显式表示出来。

属性文法是指语义规则函数无副作用的语法制导定义。

例 4.1 图 4.1 的语法制导定义表示一个台式计算器程序。这个定义分别给非终结符 *E*、*T* 和 *F* 一个存放整数值的综合属性 *val*。对每个 *E*、*T* 和 *F* 产生式,语义规则从产生式右部非终结符的属性值 val(或 digit 的 *lexval*)计算产生式左部非终结符的属性值 *val*。

表 4.1 简单台式计算器的语法制导定义

产生式	语义规则
$L \rightarrow En$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val \coloneqq E_1.val + T.val$
$E \rightarrow T$	$E.val \coloneqq T.val$
$T \rightarrow T_1 * F$	$T.val \coloneqq T_1.val * F.val$
$T \rightarrow F$	$T.val \coloneqq F.val$
$F \rightarrow (E)$	$F.val \coloneqq E.val$
$F \rightarrow digit$	$F.val \coloneqq digit.lexval$

记号 digit 有综合属性 lexval,它的值由词法分析器提供。产生式 $L \rightarrow En$ 的语义动作是打印由 E 产生的算术表达式的值,我们把它看成定义了 L 的一个虚拟属性。这个台式计算器的 Yacc 说明在图 3.25 已给出,目的是用来说明 LR 分析时的翻译。

在语法制导定义中,终结符看成只有综合属性,因而定义中没有提供计算终结符属性的语义规则,终结符的属性值通常由词法分析器提供。开始符号没有任何继承属性。

4.1.2 综合属性

综合属性在实践中有广泛应用。仅仅使用综合属性的语法制导定义叫做 S 属性定义。对于 S 属性定义,分析树各结点属性的计算可以自下而上地完成:从叶结点到根,通过计算语义规则而得到结点的属性。4.2 节将描述 LR 分析器的生成器怎样实现以 LR 文法为基础的 S 属性定义。

每个结点的属性值都标注出来的分析树叫做注释分析树(annotated parse tree),计算各结点属性值的过程叫做分析树的注释或修饰。

例 4.2 例 4.1 的 S 属性定义说明一台式计算器。例如,若输入是表达式 8+5*2,并跟随一个换行符,那么该计算器打印值 18。图 4.1 是 8+5*2 n 的注释分析树,在树的根结点打印 18。

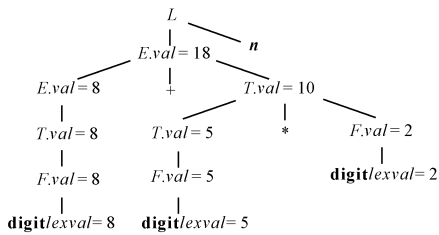


图 4.1 8+5*2n 的注释析树

为了明白属性值是怎么计算的,考虑最左边最底层的内部结点,它使用的产生式是 F→digit,对应的语义规则是 F.val ≒ digit.lexval。从该规则得到该结点的属性 F.val 为 8,因为它的子结点 digit 的 lexval 是 8。同样地,该结点的父结点的属性 T.val 也为 8。

再以产生式 E→E + T 的结点为例,这个结点的属性 E.val 由产生式 $E \rightarrow E_1 + T$ 和语义规则 E.val ≒ E_1.val + T.val 定义。当在这个结点运用该语义规则时,子结点 E_1 的 val 为 8,子结点 T 的 val 为 10,故在此结点求得 E.val 的值为 18。

4.1.3 继承属性

在分析树中,一结点的继承属性是由该结点的父结点和/或兄弟结点的属性来定义的。程序设计语言的一些语法结构的属性依赖于它们所在的上下文,此时使用继承属性是方便的。在下面的例子中,继承属性传递类型信息给一张声明表中的各个标识符。

例 4.3 在表 4.2 的语法制导定义中,非终结符 D 产生的声明由保留字 int 或 real 及一张标识符表组成。非终结符 T 有综合属性 type,它的值由声明中的保留字决定。产生式 D→TL 的语义规则置 L 的继承属性为声明中的类型。产生式 $L \rightarrow L_1$, id 的语义规则 L_1.in ≒ L.in 把继承属性 in 沿分析树向下传递给类型。L 产生式的规则调用过程 addtype,把类型信息加到符号表中各个标识符的条目(属性 entry 给出条目的入口)中。

表4.2 有继承属性的语法制导定义

产生式	语义规则
$D \rightarrow TL$	$L.in := T.type$
$T \rightarrow$ int	$T.type := integer$
$T \rightarrow$ real	$T.type := real$
$L \rightarrow L_1$, id	$L_1.in := L.in$;
	addtype (id.entry, L.in)
$L \rightarrow$ id	addtype (id.entry, L.in)

图4.2给出句子 int id_1, id_2, id_3 的注释分析树。这些属性值的确定首先是计算根的左子结点属性 T.type,然后在根的右子树自上而下地计算三个 L 结点的 L.in。在每个 L 结点,我们还调用过程 addtype,在符号表中将右子结点上标识符的类型记为整型。

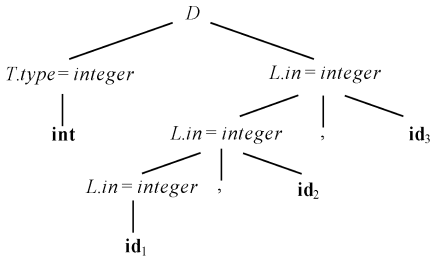


图4.2 int id_1, id_2, id_3 的注释分析树

使用继承属性的地方很多,例如,可以用继承属性来记住标识符是出现在赋值号的左边还是右边,以便决定是需要它的地址还是需要它的值。

重写语法制导定义使之仅使用综合属性总是可能的,但有时会使得重写后的文法失去了简洁和直观,反而不如使用带继承属性的语法制导定义自然。

4.1.4 属性依赖图

如果分析树一结点的属性 b 依赖某个结点的属性 c,那么属性 b 的语义规则的计算必须在属性 c 的语义规则的计算之后。分析树结点的属性间的互相依赖可以用一种叫做依赖图的有向图来描绘。

在构造分析树的依赖图之前,先为由过程调用组成的语义规则引入虚拟综合属性 b,使得每条语义规则都能写成 $b := f(c_1, c_2, \cdots, c_n)$ 的形式。依赖图的组成是这样的:分析树上每个结点的所有属性在依赖图上各有一个结点,如果属性 b 依赖于属性 c,那么从 c 的结点到 b 的结点有一条有向边。

例如,若 $S.s := f(A.a, B.b, C.c)$ 是产生式 $S \rightarrow ABC$ 的语义规则,它定义了依赖于属性 A.a,B.b 和 C.c 的综合属性 S.s。如果把这个产生式用于分析树,那么在依赖图上有 4 个结点 S.s,A.a,B.b 和 C.c,并且结点 A.a,B.b 和 C.c 分别有边到结点 S.s。

如果产生式 $S \rightarrow ABC$ 有语义规则 $A.a := f(S.s, B.b, C.c)$,那么在依赖图上,结点

S.s,*B.b* 和 *C.c* 分别有边到结点 *A.a*,因为 *A.a* 依赖于 *S.s*,*B.b* 和 *C.c*。

例 4.4 图 4.3 给出了图 4.2 分析树的依赖图。在图 4.3 中,虚线表示的是分析树;依赖图的结点用数表示,边用实线表示。从结点 4 的 *T.type* 到结点 5 的 *L.in* 有一条边,因为根据产生式 $D \rightarrow TL$ 的语义规则 *L.in* := *T.type*,*L.in* 依赖于 *T.type*。因为产生式 $L \rightarrow L_1, id$ 的语义规则 $L_1.in := L.in$ 导致 $L_1.in$ 依赖于 *L.in*,因此依赖图上有分别到达结点 7 和 9 的两条向下的边。L 产生式的语义规则 addtype(id.entry, L.in) 导致虚拟属性的建立,结

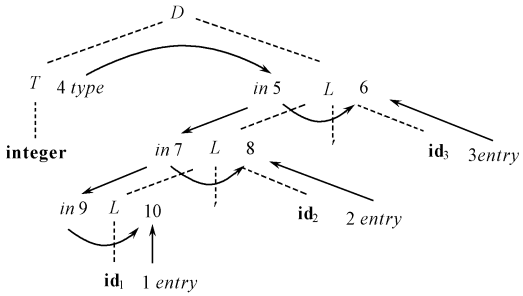


图 4.3 图 4.2 分析树的依赖图

点 6,8,10 是这样的虚拟属性结点。

4.1.5 属性计算次序

拓扑排序是有向无环图的结点的一种排序 $m_1, m_2, \cdots, m_k$,它使得边只会从这个次序中先出现的结点到后出现的结点,也就是若 $m_i \rightarrow m_j$ 是从 m_i 到 m_j 的边,那么在此排序中 m_i 先于 m_j。

显然,依赖图的任何拓扑排序都是分析树中结点属性计算的一个正确次序,即按拓扑排序进行计算的话,用语义规则 $b := f(c_1, c_2, \cdots, c_k)$ 计算 b 时,属性 $c_1, c_2, \cdots, c_k$ 已经计算过了。

这样,由语法制导定义说明的翻译可以准确地按下述步骤完成:

(1) 首先根据基础文法构造输入的分析树;

(2) 按上面讨论的方法构造属性依赖图;

(3) 对依赖图的结点进行拓扑排序,得到语义规则的计算次序;

(4) 按这个次序计算属性,得到输入串的翻译。

例 4.5 图 4.3 的依赖图中,每条边从序号较低的结点到序号较高的结点,因此结点 1,2,…,10 构成该图的一个拓扑排序。把依赖图中序号为 n 的结点的属性写成 a_n,那么从这个拓扑排序可以得下面的程序:

$a_4 = integer;$
$a_5 = a_4;$
$addtype(id_3.entry, a_5);$
$a_7 = a_5;$
$addtype(id_2.entry, a_7);$
$a_9 = a_7;$
$addtype(id_1.entry, a_9);$

这些语义规则的计算把类型 integer 存于符号表中每个标识符的条目中。

我们把语义规则的这种计算方法称为分析树方法。若依赖图有环,则这种方法失败。这种方法的缺点是编译速度很慢,因为它是在编译过程中决定属性的计算次序。显然,要想提高编译速度,必须考虑在编译前,即在构造编译器的时候就能把计算次序确定下来,免去编译时的构造依赖图和拓扑排序等工作。下面概述的两种方法分别是从手工构造编译器和自动生成编译器的角度考虑的改进。

在构造编译器时,用专门的工具或用手工对产生式的语义规则进行分析,对每个产生式,得到与它相联系的一组语义规则的计算次序。这样,把计算次序在编译前确定下来;编译时,分析树上结点属性的计算就按事先确定的次序进行。这种方法称为基于规则的方法,它适用于手工构造编译器。这种方法的缺点是,一些属性依赖关系复杂的语法制导定义很难事先确定属性计算次序,因此这种方法对语法制导定义的种类有限制。

如果事先确定了属性的计算策略,反过来要求编译器的设计者遵守我们限定的计算策略去写语义规则,那么这种方法称为忽略规则的方法(oblivious method)。例如,若我们的策略是边分析边计算,即在分析的同时完成属性计算,那么编译器的设计者在写语义规则时,必须考虑到分析树的结点是从左向右地生成等特点对属性计算带来的限制。这种方法适用于编译器的自动生成,它大大限制了能够实现的语法制导定义的种类,但是能够得到高效的编译器。

基于规则的方法和忽略规则的方法都不必在编译时显式构造依赖图,和分析树方法相比,它们使编译器的时空效率大大改进。从 4.2 节开始介绍的各种翻译方法都属于这两种方法。

4.2 S属性定义的自下而上计算

我们已初步了解了如何用语法制导定义来说明翻译,从本节开始,将带领读者一边熟悉语法制导定义,一边研究其实现。对任意语法制导定义都适用的翻译器是很难建立的,但对

语法制导定义的一些常用形式,它们的翻译器很容易构造。本节考虑 S 属性定义这一类语法制导定义。我们通过构造语法树的语法制导定义来熟悉 S属性定义。

4.2.1 语法树

语法树是分析树的浓缩表示(有些书把分析树称为语法树,而把这里的语法树称为抽象语法树),对表示语言结构是有用的。语法树作为中间表示允许把翻译从分析中分离出来,形成先分析后翻译的方式。即使是边分析边翻译,语法树作为一种概念上的中间表示,也是有用的。C编译器通常构造语法树。

在语法树中,算符和关键字不是作为叶结点,而是作为内部结点,这些内部结点对应分析树中这些叶结点的父结点。例如,对于产生式 $S \rightarrow$ if B then S_1 else S_2,可以把它的右部看成有一个算符 if-then-else 和3个运算对象 B, S_1 和 S_2,它的语法树见图 4.4(a)。

语法树中另一个简单的地方是单非产生式链可能消失,图 4.4(b)是表达式 8+5*2 的语法树。

语法制导翻译可以基于分析树,也可以基于语法树,方法是一样的。如同在分析树中那样,在语法树中也可以把属性附加到结点。

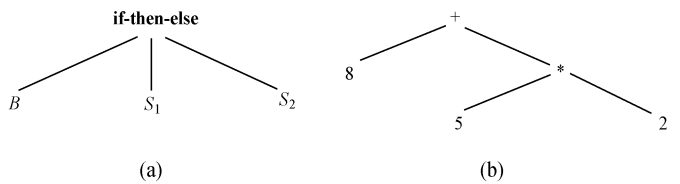


图 4.4 语法树例子

4.2.2 构造语法树的语法制导定义

本节给出构造表达式的语法树的语法制导定义。首先介绍一下数据结构,语法树的结点可以用有若干域的记录来实现。对于算符结点,一个域存放算符,该域作为该结点的标记,其余两个域含指向运算对象的指针。对于基本运算对象结点,一个域存放运算对象类别,另一个域存放其值。当用于翻译时,语法树的结点可能还有其他域来保存加在该结点的其他属性值(或属性值的指针)。

下面我们解释语义规则中用到的函数,这些函数用来建立语法树的叶结点和内部结点,每个函数都返回新建结点的指针。

(1) mkleaf(id, entry)。它建立标记为 id 的标识符结点,结点有一个域 entry,它是符号

表中该标识符条目的指针。

(2) mkleaf(num, val)。它建立标记为num的数结点,结点有一个域val,它是该数的值。

(3) mknode(op, left, right)。它用来建立标记为op的算符结点,结点有两个指针域,分别是left和right。

表4.3是为含+和*的表达式构造语法树的*S属性定义。它给文法的产生式添加语义*规则来安排函数mknode和mkleaf的调用,以便构造语法树。E,T和F的综合属性nptr用来记住函数调用返回的指针。

表4.3 构造表达式语法树的语法制导定义

产生式	语义规则
$E \rightarrow E_1 + T$	$E.nptr := mknode('+', E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr := T.nptr$
$T \rightarrow T_1 * F$	$T.nptr := mknode('*', T_1.nptr, F.nptr)$
$T \rightarrow F$	$T.nptr := F.nptr$
$F \rightarrow (E)$	$F.nptr := E.nptr$
$F \rightarrow id$	$F.nptr := mkleaf(id, id.entry)$
$F \rightarrow num$	$F.nptr := mkleaf(num, num.val)$

例4.6 图4.5给出了表达式*a*+5***b*的*注释分析树和执行语义规则所构造出的语法*树,分析树由带点的线表示。标有非终结符E,T和F的分析树结点,其综合属性nptr指向由该非终结符推出的表达式的语法树的根结点。

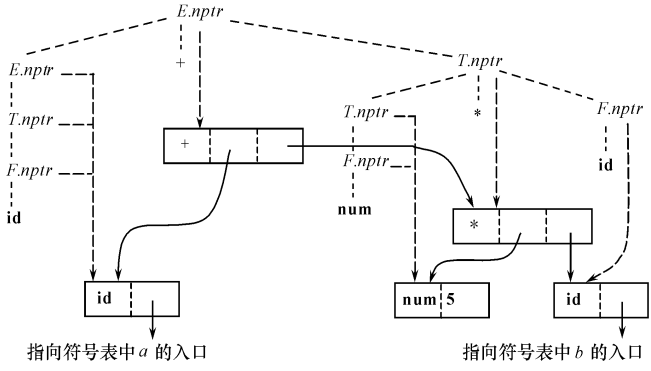


图4.5 *a*+5***b*的语法树的构造

产生式F→id和F→num的语义规则定义的属性F.nptr也是指针,它们分别指向代表

标识符和数的叶结点。属性 id.entry 和 num.val 是词法单元的值,词法分析器把它们随记号 id 和 num 一起返回。

在图 4.5 中,当表达式 E 只有一项,即使用产生式 E→*T* 时,*属性 E.nptr 和 T.nptr 的值*一样。当使用产生式 E→E_1+*T* 的语义规则 *E.nptr* ≔ *mknode*(+, E_1.*nptr*, *T.nptr*)时,先前的规则已把 E_1.*nptr 和 T.nptr 分别置为指向叶结点 a 和内部结点* 5∗ *b*。

处于图 4.5 的下部,由记录形成的树是构成输出的真正语法树,而虚线表示的树是分析树,它可能仅有象征意义。

根据表 4.3 的语法制导定义,对于输入 a+5∗ *b*,*实际执行的一系列函数调用如下*:

(1) p_1 ≔ *mkleaf*(id, entrya);

(2) p_2 ≔ *mkleaf*(num, 5);

(3) p_3 ≔ *mkleaf*(id, entryb);

(4) p_4 ≔ *mknode*(∗, p_2, p_3);

(5) p_5 ≔ *mknode*(+, p_1, p_4);

可以看出,写语法制导定义比写普通的程序更困难。对程序而言,整个程序的执行流程是显式地用语句描述的。而语法制导定义中语义规则的执行受输入的语法制导,当识别出输入串的一个语法结构时(可以看成发生一个事件),执行其相应的动作,因此这是一种事件驱动的程序设计。

4.2.3 *S* 属性的自下而上计算

综合属性可以由自下而上的分析器在分析输入时完成计算。分析器可以把文法符号的综合属性值放在它的栈里,每当归约时,根据出现在栈顶的产生式右部符号的属性来计算左部符号的综合属性。我们说明如何扩展分析器的栈使之能够保存综合属性。在 4.5 节将看到这种实现也支持一些继承属性。

S 属性定义的翻译器可以借助 LR 分析器的生成器来实现,例如 3.7 节讨论的 Yacc。根据 S 属性定义,分析器的生成器可以构造出翻译器,它在分析输入时计算属性。

自下而上分析器用栈来保存已分析子树的信息。可以在分析栈中增加一个域来保存综合属性值,图 4.6 给出了一个例子。假设拓展后的分析栈由状态数组 state 和值数组 val 实现,如图 4.6 所示的那样。state 的每个条目是 LR(1)分析表的指针或下标(注意,文法符号隐含在状态里,无需存放在栈中),但为直观起见,用文法符号来代替状态,这个符号就是 3.5 节所描述的分析栈中被状态盖住的那个符号。val 数组为文法符号存放的综合属性。如果 state 的第 i 个符号是 A,那么 val[i]保存对应这个 A 的分析树结点的属性值。

栈顶由指针 top 指示,并假定综合属性刚好在每步归约前计算。若产生式 A→*XYZ 的*

语义规则是 A.a ≒ *f*(*X.x*, *Y.y*, *Z.z*),那么在 *XYZ* 归约成 *A* 之前,属性 *Z.z* 的值在 *val*[top],*Y.y* 的值在 *val*[top − 1],X.x 的值在 val[top − 2]。如果某个符号没有属性,那么 val 数组对应条目没有定义。归约后,top 的值减 2,覆盖 A 的状态放进 *state*[top](即 *X* 原来的位置),综合属性的值 *A.a* 放进 val[top]。

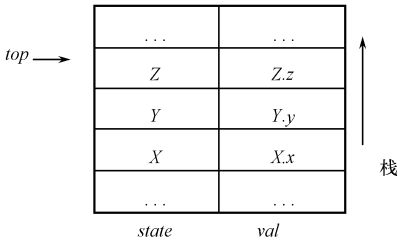


图 4.6　有综合属性域的分析栈

例 4.7　再次考虑表 4.1 的台式计算器的语法制导定义。图 4.1 的注释分析树上的综合属性可以由 *LR* 分析器在分析输入 8+5*2 n 期间计算。同前面一样,假定词法分析器提供属性 digit.lexval 的值。当分析器把 digit 移进栈时,记号 digit 置入 state[top],它的属性值放在 *val*[top]。

用 3.5 节的技术构造基础文法的 *LR* 分析器。为了计算属性,需要修改分析器,让它在归约前执行表 4.1 的语义动作,这些语义动作可以翻译成表 4.4 的栈操作代码段。这些代码段实际上就是用属性在 val 数组的位置代替表 4.1 语义规则的属性而得到的。

表 4.4　用 LR 分析器实现台式计算器

产生式	代码段
$L \rightarrow E$ n	*print*(val[top − 1])
$E \rightarrow E_1 + T$	val[top − 2] ≒ val[top − 2] + val[top]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	val[top − 2] ≒ val[top − 2] * val[top]
$T \rightarrow F$	
$F \rightarrow$ (E)	val[top − 2] ≒ val[top − 1]
$F \rightarrow$ digit	

表 4.4 中的代码段中,属性计算的结果都置入 val 数组中 top − 2 的位置,因为刚好对应产生式的右部都是 3 个符号。这里的 top 指归约前的栈顶,归约后它的值会根据右部符号的多少进行调整。

表 4.5 展示了面临输入 8+5*2n 时分析器的动作序列。每步动作后分析栈的 state 域和 val 域的内容都在表中给出,我们仍然用对应的文法符号代替状态。为直观起见,我们还用实际的输入数字代替记号 digit。

考虑看见第一个输入符号 8 时的动作序列。首先分析器把对应记号 digit 的状态移进

栈(状态由8表示,属性值8在 *val*域)。*第二步,分析器用产生式 F→*digit归约并完成语义规则 F.val = digit.lexval的计算。第三步,分析器按 T→*F归约,完成其语义规则的计算。*第四步,分析器按 E→*T归约,完成其语义规则的计算。注意,没有代码段和这*3个产生式相联,所以 val数组不改变,但每步归约后 val栈顶含归约产生式左部符号的属性。

表4.5 翻译器面临8+5*2n时的动作

输入	state	val	所用产生式
8+5*2n	—	—	
+5*2n	8	8	
+5*2n	*F*	8	*F*→digit
+5*2n	*T*	8	*T*→*F*
+5*2n	*E*	8	*E*→*T*
5*2n	*E*+	8 +	
*2n	*E*+5	8 + 5	
*2n	*E*+*F*	8 + 5	*F*→digit
*2n	*E*+*T*	8 + 5	*T*→*F*
2n	*E*+*T**	8 + 5 *	
n	*E*+*T**2	8 + 5 * 2	
n	*E*+*T**F*	8 + 5 * 2	*F*→digit
n	*E*+*T*	8 + 10	*T*→*T**F*
n	*E*	18	*E*→*E*+*T*
	*E*n	18—	
	L	18	*L*→*E*n

在上面的实现轮廓中,代码段是刚好在归约前执行。归约提供一种“挂钩”,任何代码段组成的动作可以悬挂在上面,即允许用户把代码和产生式联系起来,按此产生式归约时执行这些代码。

本小节的翻译方法受分析方法的限定,它只能用于S属性定义,而不考虑其他类的属性定义,因此它属于忽略规则的方法。

4.3 *L属性定义的自上而下计算*

从4.2节可知,将LR分析器进行拓展,很容易把S属性定义所要求的翻译嵌在分析过程中完成。那么,这种边分析边翻译的方式能否适用于有继承属性的情况。

当翻译在分析时发生,属性的计算次序一定受分析方法所限定的分析树结点建立次序的限制。不管是自上而下分析还是自下而上分析,一个共同的特点是,分析树的结点是自左向右生成。如果属性信息是自左向右流动,那么就有可能在分析的同时完成属性计算。下面按这种想法定义 L 属性定义,L 代表左(left),因为属性信息是从左向右流。

4.3.1 L 属性定义

语法制导定义是 L 属性的,如果每个产生式 $A\rightarrow X_1\ X_2\cdots\ X_n$ 的每条语义规则计算的属性是 A 的综合属性 或者是 X_j 的继承属性,$1\leq j\leq n$,但它仅依赖:

(1) 该产生式中 X_j 左边符号 X_1,X_2,…,X_{j-1} 的属性;

(2) A 的继承属性。

显然,S 属性定义属于 L 属性定义,因为限制 (1) 和 (2) 仅对继承属性进行限制。

例 4.3 有关变量类型声明的语法制导定义是一个 *L 属性定义,其中类型信息从左向右流*。为了说明问题,我们把这个语法制导定义重复写在这里。

表 4.6 有继承属性的语法制导定义

产生式	语义规则
$D\rightarrow TL$	$L.in := T.type$
$T\rightarrow$ int	$T.type := integer$
$T\rightarrow$ real	$T.type := real$
$L\rightarrow L_1$, id	$L_1.in := L.in$; *addtype* (id.entry, L.in)
$L\rightarrow$ id	*addtype* (id.entry, L.in)

在这个例子中,如果语义规则都在归约时才执行的话,肯定会有问题。例如产生式 $D\rightarrow TL$ *的语义规则是* $L.in := T.type$,*在分析由 L 推出的标识符表时,需要用* $L.in$ *的值*,等归约该产生式时才对 L.in 赋值的话肯定太晚了。这样,我们需要考虑语义规则的执行时机,以及反映语义规则执行时机的描述方法。下一小节介绍这种描述方法。

4.3.2 翻译方案

翻译方案和语法制导定义不同之处是它的语义动作(而不叫语义规则)放在括号 {} 内,并且可以插在产生式右部的任何地方。这是一种动作和分析交错的表示法,以表达动作的执行时刻。若 $A\rightarrow\alpha\{\cdots\}\beta$,那么 $\{\cdots\}$ 中语义动作的执行在 α 的推导(或向 α 的归约)结束以后,在 β 的推导(或向 β 的归约)开始之前。我们可以把 {} 之间的语义动作想像成一个文法

符号,在分析过程中对该符号进行推导(或归约)的时候,就是该语义动作执行的时候。

本章也使用翻译方案作为说明分析期间翻译的一种方法。

例4.8 下面是一个简单的翻译方案,它把有加和减的中缀表达式翻译成后缀表达式。

$$E \rightarrow T\ R$$

$$R \rightarrow \text{addop}\ T\ \{\text{print}(\text{addop.lexeme})\}\ R_1 \mid \varepsilon \qquad (4.1)$$

$$T \rightarrow \text{num}\ \{\text{print}(\text{num.val})\}$$

如果输入是8+5-2,该翻译方案的输出是85+2-。第2行的动作 *print(addop.lexeme)* 必须放在 T 和 R_1 之间,移到别的位置都会导致不正确的结果。

设计翻译方案时,必须保证动作在引用属性时其值已经可用,也就是保证动作不会引用还没有计算的属性值。

只有综合属性的情况最简单。此时,为每条语义规则建立一个赋值动作,把该动作放在对应产生式右部的末端,由此得到翻译方案。

如果同时有继承属性,必须仔细斟酌。下面是三条限制,这些限制的给出受L属性定义的启发:

(1) 产生式右部符号的继承属性必须在先于这个符号的动作中计算。

(2) 一个动作不能引用该动作右边符号的综合属性。

(3) 左部非终结符的综合属性只能在它所引用的所有属性都计算完后才能计算。计算该属性的动作通常放在产生式右部的末端。

对L属性语法制导定义,构造满足上面三个条件的翻译方案总是可能的。下面的例子说明这一点。它基于数学排版语言EQN。对于输入

```
E sub 1 .val
```

EQN编排的 E、1和.val的相对位置和相对大小见图4.7,下标1以较小的字体印刷,它的位置也低于E和.val。

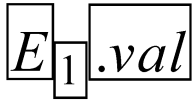


图4.7 编排单元的排版结果

例4.9 从表4.7的 *L属性定义,构造图*4.8的翻译方案。表中非终结符B代表公式编排单元,产生式 $B \rightarrow B_1\ B_2$ 代表两个单元的并置,$B \rightarrow B_1$ sub B_2 表示作为下标的第二个单元,其编排的尺寸比第一个单元小,并且位置较低。

继承属性ps表示点的大小,它会影响公式的高度。产生式 $B \rightarrow$ text 的规则用正文的正常高度乘以点的大小以得到正文的实际高度。text的属性h可以根据text的性质查表得到。在产生式 $B \rightarrow B_1 B_2$ 的语义规则中,通过复写规则,B_1 和 B_2 继承了B的点的大小。B的高度由综合属性ht表示,它取 B_1 和 B_2 高度的较大值。

在产生式 $B \rightarrow B_1$ sub B_2 的语义规则中,函数shrink将 B_2 的点缩小30%。*函数disp在计*算B的高度时,把单元 B_2 向下偏置。输出编排结果的语义动作在图中没有给出。

表 4.7 定义编排单元大小和高度的语法制导定义

产生式	语义规则
$S \rightarrow B$	$B.ps := 10$; $S.ht := B.ht$
$B \rightarrow B_1\ B_2$	$B_1.ps := B.ps$; $B_2.ps := B.ps$; $B.ht := max(B_1.ht, B_2.ht)$
$B \rightarrow B_1$ sub B_2	$B_1.ps := B.ps$; $B_2.ps := shrink(B.ps)$; $B.ht := disp(B_1.ht, B_2.ht)$
$B \rightarrow$ text	$B.ht := text.h * B.ps$

表 4.7 中,惟一的继承属性是非终结符 B 的 ps,定义 ps 的语义规则仅依赖产生式左部非终结符的继承属性,所以该定义是 L 属性定义的。

$S \rightarrow$ {B.ps := 10}
  B {S.ht := $B.ht$}

$B \rightarrow$ {$B_1.ps := B.ps$}
  B_1 {$B_2.ps := B.ps$}
  B_2 {$B.ht := max(B_1.ht, B_2.ht)$}
$B \rightarrow$ {$B_1.ps := B.ps$}
  B_1
  sub {$B_2.ps := shrink(B.ps)$}
  B_2 {$B.ht := disp(B_1.ht, B_2.ht)$}
$B \rightarrow$ text {$B.ht := text.h * B.ps$}

图 4.8 从表 4.7 构造的翻译方案

图 4.8 的翻译方案是根据上面的三点要求,通过把对应于表 4.7 语义规则的赋值插入产生式而得到的。为了可读性,把

$S \rightarrow$ {B.ps := 10} B {S.ht := $B.ht$}

写成

$S \rightarrow$ {B.ps := 10}
  B {S.ht := $B.ht$}

注意,给继承属性 $B_1.ps$ 和 $B_2.ps$ 赋值的动作刚好在产生式右部 B_1 和 B_2 的前面。

大多数算术运算符是左结合的,因此用左递归文法表示表达式是自然的。但是在构造预测翻译器时,必须消除文法中的左递归,而左递归的消除可能会引起继承属性的出现。下

面的例子说明这一点。

例 4.10 如果把表 4.3 构造语法树的语法制导定义变成翻译方案,那么 E 的产生式和语义动作成为

$E \to E_1 + T$ {E.nptr := *mknode*(+ , E_1.*nptr*, *T.nptr*)}

$E \to T$ {E.nptr := *T.nptr*}

从这个方案消除左递归时,变换后的翻译方案见图 4.9。F 的产生式和语义动作类似于表 4.3 最初定义中的那些产生式和语义动作。

$E \to T$ {R.i := *T.nptr*}
R {E.nptr := *R.s*}
$R \to +$
T {R_1.*i* := *mknode*(+ , R.i, T.nptr)}
R_1 {R.s := R_1.*s*}
$R \to \varepsilon$ {R.s := *R.i*}

$T \to F$ {W.i := *F.nptr*}
W {T.nptr := *W.s*}
$W \to *$
F {W_1.*i* := *mknode*(* , W.i, F.nptr)}
W_1 {W.s := W_1.*s*}
$W \to \varepsilon$ {W.s := *W.i*}

$F \to (E)$ {F.nptr := *E.nptr*}
$F \to$ id {F.nptr := *mkleaf*(id, id.entry)}
$F \to$ num {F.nptr := *mkleaf*(num, num.val)}

图 4.9 构造语法树的翻译方案

图 4.10 给出了图 4.9 的动作是怎样为 5 * *a* * *b* 构造语法树的。为使图简单起见,在图中略去了 E] *TR*] *T* 的部分。综合属性在文法符号的右边,继承属性在文法符号的左边。语法树的叶结点由与产生式 F→id 和 F→num 相联的动作构造,和例 4.6 的一样。在最左边的 F,属性 F.nptr 指向叶结点 a,指向 a 的指针由 T→ *FW* 右部的 W 的属性 W.*i* 继承。

当产生式 W→ * FW_1 用于根的右子结点时,W.i 指向 a 结点,F.nptr 指向结点 5。现在 mknode 作用于乘算符和这两个指针,构造出对应 a*5 的结点。

最后使用的产生式是 W→ ε 时,*W.i* 指向整个树的根。整个树通过 *W.*结点的 *s* 属性

返回(图4.10没有给出),直至它成为T.nptr的值。

原来左递归的文法没有继承属性,为什么把文法从左递归改成右递归后会出现继承属性?可以这样直观地解释:对于表4.3的文法,属性信息的流动方向和归约方向是一致的,因此一般来说只用综合属性就可以了;而图4.9翻译方案的属性信息(语法树结点指针)是从左向右流的,归约却是从右向左的,这种不一致性导致了继承属性的出现。

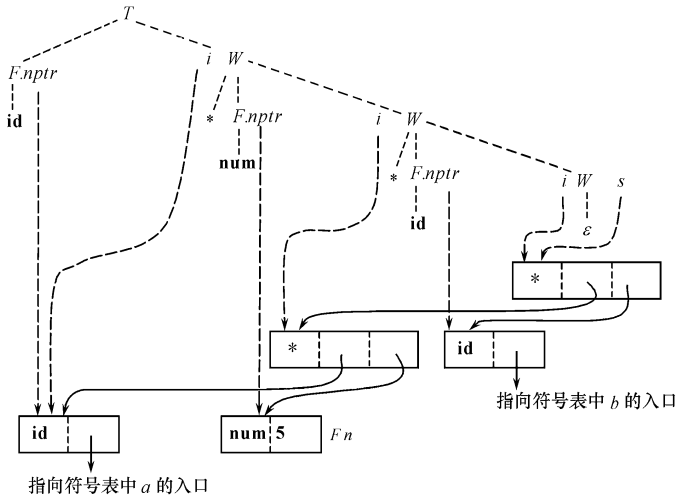


图4.10 *a*∗5∗*b*的语法树的构造

这两节给出了一些具体的语法制导定义和翻译方案。可以看出,和用程序设计语言编程序相比,编写语法制导定义和翻译方案是一项更有挑战性的工作,在下面几章中将会深刻体会到这一点。

4.3.3 预测翻译器的设计

本节讨论在预测分析的同时完成L属性定义。为明显看出动作和属性计算发生的次序,用翻译方案而不是语法制导定义。

下面的算法把预测分析器的构造方法推广到翻译方案的实现,该翻译方案的文法适于自上而下分析。

算法4.1 构造语法制导的预测翻译器。

输入 语法制导的翻译方案,其基础文法适于预测分析。

输出 语法制导翻译器的代码。

方法 修改预测分析器的构造技术。

(1) 为每个非终结符 A构造一个函数，A 的每个继承属性声明为该函数的一个形式参数，*A 的综合属性作为它的返回值 (可以是记录，或者是每个属性占一个域的记录的指针)*。为简单起见，假定每个非终结符正好只有一个综合属性。A的函数还为A产生式的其他每个文法符号的每个属性声明一个局部变量。

(2) A函数的代码的骨架是根据当前的输入决定使用什么产生式。

(3) 和每个产生式有关的代码按下面的方法生成。我们从左向右地考虑产生式右部的记号、非终结符和语义动作。

(a) 对于有综合属性 x 的记号 X，把 X的值保存在为 X.x 声明的变量中。然后产生匹配记号 X 的调用，并推进输入指针。

(b) 对于非终结符 B，产生赋值 $c := B(b_1, b_2, \cdots, b_k)$，它的右部是函数调用，其中 $b_1, b_2, \cdots, b_k$ 是对应 B继承属性的变量，c是代表 B 综合属性的变量。

(c) 对于每个语义动作，把代码复写到分析器，把对属性的引用改成对相应的局部变量的引用。

算法 4.1 在 4.5 节还要扩充，在分析树已构造好的情况下实现 L 属性定义。

例 4.11 图 4.9 的文法是 LL(1) 的，因而适合于自上而下分析。从该文法的非终符的属性，我们得到函数 E, R, T, W 和 F 的变元和结果类型如下。因为 E, T 和 F 没有继承属性，所以它们没有变元。

```
function E : ↑ syntax_tree_node;
function R (i : ↑ syntax_tree_node) : ↑ syntax_tree_node;
function T : ↑ syntax_tree_node;
function W (i : ↑ syntax_tree_node) : ↑ syntax_tree_node;
function F : ↑ syntax_tree_node;
```

R 的代码基于图 4.11 的分析过程。如果向前看符号是 +，那么使用产生式 R→+ TR，先用 match 过程去读 + 后面的下一个输入记号，然后调用过程 T 和 R。否则该产生式什么也不做，使用的是产生式 R→ε。

```
procedure R;
begin
    if lookahead = '+' then begin
        match('+'); T; R
    end
    else begin /* 什么也不做 */
    end
end
```

图 4.11 产生式 $R \rightarrow +TR \mid \varepsilon$ 的分析过程

图4.12的 *R* 函数含计算属性的代码,有这样几步:记号 + 的值 lexval 存于 addoplexeme,匹配+,调用 T,用 nptr 保存它的结果。变量 i1 对应继承属性 *R*1.*i*,s1 对应综合属性 R1.s。return 语句返回 s。其他函数可以效仿这个函数去构造。

```
function R (i : ↑ syntax_tree_node) : ↑ syntax_tree_node;
    var nptr, i1, s1, s : ↑ syntax_tree_node;
            addoplexeme : char;
    begin
        if lookahead = + then begin
                /* 产生式 R→+TR */
            addoplexeme := lexval;
            match( + );
            nptr := T;
            i1 := mknode(addoplexeme, i, nptr);
            s1 := R(i1);
            s := s1
        end
        else s := i; /* 产生式 R→ε */
        return s
    end;
```

图4.12 语法树的递归下降构造

本小节讨论的是 L 属性定义的自上而下翻译,属性计算次序预先约定,因而仍属于忽略规则的方法。

4.3.4 用综合属性代替继承属性

我们已经看出,S 属性定义比 L 属性定义易于理解。从例 4.10 知道,改变文法可能会引入继承属性。同样,改变文法有时可以避免使用继承属性,后者对于构造只允许综合属性的翻译方案极其有用。

例如,Pascal 的声明由标识符表加类型组成,如 m, n : integer。这种声明可以用下面的文法描述:

$D \rightarrow L : T$

$T \rightarrow$ integer | char

$L \rightarrow L$, id | id

在这个文法中,标识符表由 L 产生,但类型不是 L 的子树,我们不能靠仅使用综合属性

把类型和标识符联系起来。事实上,在第一个产生式中,由于非终结符 L 从它右边的 T 获得类型信息,所写的语法制导定义将不可能是 L 属性的,所以基于它的翻译也不可能在分析期间完成。

这个问题可以通过重新构造文法来解决,把类型作为标识符表的最后一个成分:

$D \rightarrow id\ L$

$L \rightarrow$, id L | : T

$T \rightarrow$ integer | char

现在,类型作为 L 的综合属性 L.type,当标识符由 L 产生时,它的类型可以进符号表,具体的翻译方案如下:

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

$L \rightarrow$, id L_1 {L.type $\doteq$ L_1. *type*; *addtype* (id.entry, L_1. *type*)}

$L \rightarrow$: T {L.type $\doteq$ *T.type*}

$T \rightarrow$ integer {T.type $\doteq$ *integer*}

$T \rightarrow$ real {T.type $\doteq$ *real*}

可以说,在 Pascal 的声明中,类型信息是从右向左流动,上面第一个文法是从左向右归约,因此不能在分析期间完成计算。而上面第二个文法是从右向左归约的,信息流向和归约方向的一致性使得属性计算可以在分析期间完成。

4.4 *L属性的自下而上计算*

本节提出在自下而上分析的框架中实现 L 属性定义的方法。它能实现任何基于 LL(1) 文法的 L 属性定义,也能实现许多(但不是所有的)基于 LR(1) 的 L 属性定义。该方法是 4.2.3 节自下而上翻译技术的推广。

4.4.1 删除翻译方案中嵌入的动作

4.2.3 节的自下而上翻译方法适用于 *S 属性定义,S 属性定义的语义动作都可以放在*产生式的右端。虽然 L 属性定义的继承属性计算需要嵌在产生式右部的不同地方,但是可以通过修改文法而把翻译方案中所有的嵌入动作都变换成只出现在产生式的右端,在嵌入动作只涉及虚拟属性时,这种变换尤其容易。

这种变换在文法中加入产生 ε 的标记非终结符,让每个嵌入动作由不同标记非终结符 *M 代表,并把该动作放在产生式 M→ε 的右端。例如,使用标记非终结符 M 和 N,*(4.1) 翻

译方案

$E \rightarrow T\ R$

$R \rightarrow +\ T\ \{print('+')\}\ R_1 \mid -\ T\ \{print('-')\}\ R_1 \mid \varepsilon$

$T \rightarrow num\ \{print(num.val)\}$

变换成

$E \rightarrow T\ R$

$R \rightarrow +\ T\ M\ R_1 \mid -\ T\ N\ R_1 \mid \varepsilon$

$T \rightarrow num\ \{print(num.val)\}$

$M \rightarrow \varepsilon\ \{print('+')\}$

$N \rightarrow \varepsilon\ \{print('-')\}$

这两个翻译方案中的文法接受同样的语言,并且对任何输入,它们执行的语义动作是一样的。变换后的翻译方案中,动作都在产生式的右边,所以它们可以在自下而上分析过程中归约产生式右部时完成。

4.4.2 分析栈上的继承属性

对产生式 $A \rightarrow XY$,自下而上分析器从它的栈顶移开 Y 和 X,并用 A 代替它们,完成产生式右部的归约。如果 X 有综合属性 $X.s$,4.2.3 节的实现把该属性放在分析器栈上与 X 对应的地方。

因为在 Y 以下的任何子树归约前,$X.s$ 的值已经在分析栈中,所以它的值可以被 Y 继承。如果继承属性由复写规则 $Y.i := X.s$ 定义,那么在需要使用 $Y.i$ 的地方,可以用 $X.s$ 的值来代替。在需要使用 $Y.i$ 的地方,如果能静态地确定 $X.s$ 在栈中的位置,那么这个想法很容易实现。

例 4.12 考虑由表 4.6 的语法制导定义改写的翻译方案:

$D \rightarrow T$	{L.in := T.type}
$\quad L$	
$T \rightarrow$ int	{T.type := integer}
$T \rightarrow$ real	{T.type := real}
$L \rightarrow$	{L1.in := L.in}
$\quad L1$, id	{addtype(id.entry, L.in)}
$L \rightarrow$ id	{addtype(id.entry, L.in)}

该翻译方案使用继承属性,并且继承属性由复写规则传递。下面我们以输入 int p, q, r 为例,看在需要 L.in 值的地方,能否静态地确定 T.type 在栈中的位置。

先用图 4.13 给出该输入的分析树及 type 和 in 属性之间的依赖关系,以增强直观性。

表4.8给出分析器面临该输入时的动作序列。为清楚起见,我们用对应的文法符号代替状态,用实际的标识符代替记号id。在表4.8中,每次归约L产生式右部时,T在栈中刚好处于该右部的下面,因此T.type在栈中的位置是可以静态地确定的。可以利用这个事实来访问属性*T.type*。

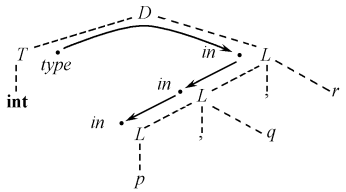


图4.13 在每个*L*结点上,*L.in* = *T.type*

按照4.2.3节假设,分析栈由一对数组state和val来实现。如果state[i]对应文法符号X,那么val[i]保存综合属性X.s。表4.9的实现利用了属性*T.type*在*val*栈中相对于栈顶的位置已知这个事实。

表4.8 分析器面临int p, q, r时的动作序列

状态	输入	所用产生式
–	int p, q, r	
int	p, q, r	
T	p, q, r	*T*→int
T p	, q, r	
T L	, q, r	*L*→id
T L,	q, r	
T L, q	, r	
T L	, r	*L*→*L*, id
T L,	r	
T L, r		
T L		*L*→*L*, id
D		*D*→*TL*

表4.9 *T.type*的值可用来代替*L.in*

产生式	代码段
D→*TL*	
T→int	*val*[*top*] = *integer*
T→real	*val*[*top*] = *real*
L→*L*$_1$, id	*addtype*(*val*[*top*], *val*[*top*−3])
L→id	addtype(val[top], val[top−1])

当使用产生式 L→id 时,id.entry 在 val 栈顶,T.type 刚好在它的下面。所以 addtype(val[top], val[top-1])等价于 addtype(id.entry, L.in)。同样,因为产生式 L→*L, id* 右部有3个符号,所以归约时 T.type 出现在 val[top-3]。其他和 L.in 有关的动作是复写规则,它们完全可以省略掉。

如果属性的位置不能预测,可以尝试增加标记非终结符,使属性的位置变得可以静态确定。下面通过一个例子来说明。

例 4.13 考虑下面的语法制导定义:

产生式	语义规则	
$S\rightarrow aAC$	$C.i \doteq A.s$	
$S\rightarrow bABC$	$C.i \doteq A.s$	(4.2)
$C\rightarrow c$	$C.s \doteq g(C.i)$	

通过复写规则,C.i 继承综合属性 A.s。由于在栈中,B 可能在、也可能不在 A 和 C 之间,因此在用产生式 C→*c* 归约时,*C.i* 的值(他就是 *A.s* 的值)可能在 val[top-2],也可能在 val[top-1]。

增加一个标记非终结符 M,插在(4.2)的第二个产生式右部的 C 之前,并增加了属性复写规则。修改后的语法制导定义如下:

产生式	语义规则
$S\rightarrow aAC$	$C.i \doteq A.s$
$S\rightarrow bABMC$	$M.i \doteq A.s;\ C.i \doteq M.s$
$C\rightarrow c$	$C.s \doteq g(C.i)$
$M\rightarrow \varepsilon$	$M.s \doteq M.i$

产生式 S→*bABMC* 和 M→ε 的语义规则保证 *C.i = M.s = M.i = A.s*,也就是对于第二个产生式 S→*bABMC*,*C.i* 间接地通过 *M.i* 和 *M.s* 继承 *A.s* 的值。这样,不管在 *aAC* 还是 bABMC 的情况下,在使用 C→*c* 时,*C.i* 的值都可以在 val[top-1] 找到。在使用 M→ε 时,M.i 的值可以在 val[top-2] 找到。

4.4.3 模拟继承属性的计算

如果继承属性并不直接等于某个综合属性,而是它的一个函数,是否能够利用上一小节的办法?回答是可以的。在这种情况下,可以用标记非终结符来模拟继承属性的计算。例如,考虑

产生式	语义规则
$S\rightarrow aAC$	$C.i \doteq f(A.s)$

$C \rightarrow c$ $\quad\quad$ $C.s := g(C.i)$

这里,定义 C.i 的规则不是简单的复写规则。如果不完成这个计算,则在栈中取不到 C.i 的值。解决办法是增加一个标记非终结符,把 f(A.s) 的计算移到对标记非终结符归约时进行,如下所示:

产生式	语义规则
$S \rightarrow aANC$	$N.i := A.s; C.i := N.s$
$N \rightarrow \varepsilon$	$N.s := f(N.i)$
$C \rightarrow c$	$C.s := g(C.i)$

当按 $N \rightarrow \varepsilon$ 归约时,N.i 的值可以在放 A.s 的地方(val[top])找到。当按 $C \rightarrow c$ 归约时,C.i 的值可以在对应于 N 的地方(val[top−1])找到。

例 4.14 对于表 4.7 的排版语言的例子,我们增加 3 个标记非终结符 *L,M 和 N,以保*证当 B 的子树归约时,所需的继承属性 B.ps 的值出现在分析栈中已知的位置。修改后的语法制导定义见表 4.10。

标记非终结符 L 的作用是完成 *$S \rightarrow LB$ 中的属性 B.ps 的初始化。$L \rightarrow \varepsilon$ 的语义规则 L.s* := 10 使得 B.ps 的初值 10 在栈中有存放的地方,以便后面引用。

标记非终结符 M 和 N(还有 L)的一个共同作用是,保证了在任何情况下向 B 归约时,*B.ps 的值总是正好在右部的下面。这个结论的证明留作一个练习。*

*N 的另一个作用是模拟继承属性的计算。在表 4.7 中,产生式 $B \rightarrow B_1$ sub B_2 的语义规*则 $B_2.ps := shrink(B.ps)$ *表明 $B_2.ps$ 不是简单地由复写规则定义的。产生式 $N \rightarrow \varepsilon$ 的动作*完成这个计算,并将计算结果保留在栈上。

表 4.10 由复写规则设置所有继承属性

产生式	语义规则
$S \rightarrow LB$	$B.ps := L.s; S.ht := B.ht$
$L \rightarrow \varepsilon$	$L.s := 10$
$B \rightarrow B_1\ MB_2$	$B_1.ps := B.ps; M.i := B.ps;$ $B_2.ps := M.s;$ $B.ht := max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.s := M.i$
$B \rightarrow B_1$ sub NB_2	$B_1.ps := B.ps; N.i := B.ps;$ $B_2.ps := N.s;$ $B.ht := disp(B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s := shrink(N.i)$
$B \rightarrow$ text	$B.ht := text.h * B.ps$

实现表 4.10 的语法制导定义的代码段列在表 4.11。在表 4.10 中,所有的继承属性都是由复写规则设置,因而只要知道它们在 val 栈中的位置,就可以获得它们的值。

表 4.11 表 4.10 的语法制导定义的实现

产生式	语义规则
$S \to LB$	$val[top-1] := val[top]$
$L \to \varepsilon$	$val[top+1] := 10$
$B \to B_1\ MB_2$	$val[top-2] := max(val[top-2], val[top])$
$M \to \varepsilon$	$val[top+1] := val[top-1]$
$B \to B_1\ sub\ NB_2$	$val[top-3] := disp(val[top-3], val[top])$
$N \to \varepsilon$	$val[top+1] := shrink(val[top-2])$
$B \to text$	$val[top] := val[top] * val[top-1]$

对于基于 LL(1) 文法的 L 属性定义,系统地引入标记非终结符,可以在 LR 分析期间完成属性计算。因为每个标记非终结符至多只有一个产生式,因此标记非终结符加入后文法仍然是 LL(1) 的。任何 LL(1) 文法一定是 LR(1) 文法,因此加标记非终结符后不会引起 LR 分析动作的冲突。

但是对于 LR(1) 文法,情况就不这么简单,标记非终结符的加入有可能使得文法不再是 LR(1) 的了。

例 4.15 文法 $L \to Lb \mid a$ 是 LR(1) 的,加了标记非终结符 M 后的文法

$$L \to MLb \mid a$$
$$M \to \varepsilon$$

不是 LR(1) 的。

该语言的句子是 $abb\cdots b$ 的形式。*根据这个文法,空归约的个数和 b 的个数一样多。*在面临 a 时,后面有多少个 b 是不知道的,因此一定有移进—归约冲突。所以这个加标记非终结符的文法不是 LR(1) 的。

算法 4.2 有继承属性的自下而上翻译。

输入 基础文法是 LL(1) 的 L 属性定义。

输出 在分析栈上计算所有属性值的分析器。

方法 把问题简化为每个非终结符 *A 只有一个继承属性 A.i,并且每个文法符号 X 都*有综合属性 X.s。如果 X 是终结符,那么它的综合属性就是词法分析器返回的记号属性。

对每个产生式 $A \to X_1 \cdots X_n$,引入 n 个新的标记非终结符 $M_1 \cdots M_n$,用 $A \to M_1 X_1 \cdots M_n X_n$ 代替该产生式。综合属性 $X_j.s$ 将进入分析栈的 val 数组内与 X_j 对应的条目中。继承属性

$X_j.i$,如果有的话,出现在同一数组与 M_j 对应的条目中。

在分析时,一个重要的不变性是,如果继承属性 $A.i$ 存在,则它的值可以在 val 数组中 M_1 的下面且紧贴 M_1 的地方找到。因为假设了开始符号没有继承属性,所以 A 是开始符号时也没有问题。即使有这样的属性,也可以把它设置在栈底的下面。根据自下而上分析步数的归纳,可以证明这种不变性。证明时需要注意这样的事实:X_j 的继承属性 $X_j.i$ 和标记非终结符 M_j 联系在一起,且 $X_j.i$ 是在向 X_j 归约之前的 M_j 的位置计算。

为了明白属性可以如所设想的那样自下而上计算,考察两种情况。首先,如果归约标记非终结符 Mj,我们知道它是属于哪个产生式 $A\rightarrow M_1X_1\cdots M_nX_n$ 的,因而知道为完成继承属性 $X_j.i$ 的计算所需的任何属性的位置。$A.i$ 在 val[top−2*j*+2] 中,$X_1.i$ 在 val[top−2*j*+3] 中,$X_1.s$ 在 val[top−2*j*+4] 中,$X_2.i$ 在 val[top−2*j*+5] 中,等等。这样,可以计算出 $X_j.i$,并把它存在 val[top+1] 中,归约后它成为新的栈顶。注意,限定文法到 LL(1) 是非常重要的,否则也许不能肯定 ε 应归约到哪个非终结符,因而无法知道应执行什么语义动作和所需属性的位置。LL(1) 文法加标记后仍然是 LL(1) 文法的证明留给读者。

第二种情况是归约非标记符号,比如说由产生式 $A\rightarrow M_1X_1\cdots M_nX_n$ 引起的归约。这时*$A.i$ 已经计算,在栈中下邻 A 将占据的位置,我们只需计算综合属性 $A.s$。计算 $A.s$ 所需*的其他属性都在栈中的已知位置,即进行这个归约时 X_j (*$1\leq j\leq n$*) *的位置。*

下面一些简化考虑是有用的:

(1) 如果 X_j 没有继承属性,我们不需要使用标记 M_j。当然,如果省略 M_j,属性在栈中预期的位置会改变,但是这种变化很容易并入分析器。

(2) 如果 $X_1.i$ *存在,它是由复写规则* $X_1.i\doteqdot A.i$ *计算,那么可以省略* M_1。因为由不变性可以知道,$A.i$ 定位在我们需要它的地方,即栈中 X_1 的下邻,因此这个值同时也可作为 $X_1.i$ *的值。*

本节讨论了如何修改翻译方案,以适应自下而上分析的限制,因此这是属于忽略规则的方法。

4.5 递归计算

除4.1节讨论的分析树方法以外,到现在为止,我们考虑的都是边分析边进行属性计算。这种方式使得访问结点的次序受到分析方法的限制,因而只能完成 L 属性计算。

对于非 L 属性定义,我们可以把分析和属性计算分开,先建立分析树,然后遍历分析树来计算属性。我们把4.3节的预测分析技术进行推广,仍然为每个非终结符构造一个函数,但是该函数不含语法分析部分,它是通过遍历分析树来完成属性计算的递归函数。该函数

以该非终结符的产生式和它的语义规则所决定的次序访问这个非终结符结点的子结点,但不一定是从左到右。

最后,我们还把这种方法推广到对分析树的扫描多于一次的翻译。

这种方法不受分析方法的限制,属性计算的次序可在构造编译器时通过分析语法制导的定义得出,因此是属于基于规则的方法。

4.5.1 自左向右遍历

先举一个 L 属性定义的例子。我们已经知道,通过为每个非终结符构造一个完成分析和翻译的递归函数,可以实现基于 LL(1) 文法的 L 属性定义。现在为了说明问题,我们让类似的递归函数在已经构造好的分析树上自左向右地遍历,由此完成属性计算。和算法 4.1 相比,这样的函数没有语法分析部分,另外增加了一个参数:分析树的结点。函数执行时,首先用 case 语句来查看该结点的产生式,知道它的子结点是什么,然后以这些子结点为参数,调用相应的函数。

例 4.16 考虑表 4.7 的确定编排单元大小和高度的语法制导定义。非终结符 B 有继承属性 ps 和综合属性 ht。按照上面思想构造出的 B 的函数在图 4.14 中,其中略去了变量和函数的类型声明。

函数 B 以结点 n 和对应该结点的继承属性 $B.ps$ 作为变元,返回对应该结点的 B.ht 的值。函数由 case 语句进行分支,区分 B 产生式的各种情况。对应每个产生式的代码计算与该产生式相关的语义规则。这些语义规则的使用次序是这样的:非终结符继承属性的计算必须先于对应该非终结符的函数的调用。

我们解释其中一个分支。在对应产生式 $B \rightarrow B_1$ sub B_2 的代码中,变量 ps,ps1 和 ps2 分别保存继承属性 $B.ps$,$B_1.ps$ 和 $B_2.ps$ 的值,变量 ht,ht1 和 ht2 分别保存综合属性 $B.ht$,$B_1.ht$ 和 $B_2.ht$ 的值。函数 child(m, i) 引用结点 m 的第 i 个子结点。因为 B_2 是结点 n 的第 3 个子结点的标记,所以 $B_2.ps$ 的值由函数调用 B(child(n,3), ps2) 确定。

```
function B(n, ps);
var ps1, ps2, ht1, ht2;
begin
    case 在结点 n 的产生式 of
      B→B1 B2 :
          ps1 := ps;
          ht1 := B(child(n,1), ps1);
          ps2 := ps;
          ht2 := B(child(n,2), ps2);
          return max(ht1, ht2);
      B→B1 sub B2 :
          ps1 := ps;
          ht1 := B(child(n,1), ps1);
          ps2 := shrink(ps);
          ht2 := B(child(n,3), ps2);
          return disp(ht1, ht2);
      B→text :
          return ps*text.h;
    default:
          error
    end
end;
```

图 4.14 表 4.7 的非终结符 B 的函数

4.5.2 其他遍历方法

其实,一旦分析树可用,就可以按任意次序访问结点的子结点。考虑例4.17的非*L*属性定义,非终结符A的一个产生式要求以从左到右的次序访问一个结点的子结点,而A的另一个子结点要求以从右到左的次序访问一个结点的子结点。由于分析树已经建立,访问子结点的次序已经不受分析树结点建立次序的影响。访问次序的安排主要还是考虑:一个结点的继承属性的计算应先于第一次对它的访问,一个结点的综合属性的计算应先于最后一次离开这个结点。

例4.17 表4.12文法的每个非终结符都有继承属性*i*和综合属性*s*,两个产生式的依赖图在图4.15给出,其中$A\rightarrow QR$的规则建立从右到左的依赖。

表4.12 非终结符A的产生式和语义规则

产生式	语义规则
$A\rightarrow LM$	$L.i:=l(A.i)$;$M.i:=m(L.s)$;$A.s:=f(M.s)$
$A\rightarrow QR$	$R.i:=r(A.i)$;$Q.i:=q(R.s)$;$A.s:=f(Q.s)$

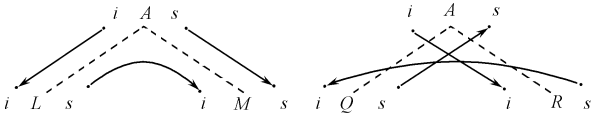


图4.15 非终结符*A*的两个产生式的依赖图

假定L,M,Q和R的函数已经建立,而非终结符A的函数展示在图4.16中。在这个函数中,变量根据非终结符的属性命名,如li和ls分别是对应L.i和L.s的变量。产生式$A\rightarrow LM$的代码按例4.16那样构造。产生式$A\rightarrow QR$的代码先访问R的子树,然后访问Q的子树。除了访问次序不同外,这两个产生式的代码没有什么区别。

```
function A(n, ai);
var li, ls, mi, ms, qi, qs, ri, rs;
begin
    case 在结点 n的产生式 of
     A→LM :          /* 从左到右的次序 */
        li := l(ai);
        ls := L(child(n,1), li);
```

```
        mi := m(ls) ;
        ms := M(child(n,2), mi) ;
        return f(ms) ;
    A→QR :                    /* 从右到左的次序 */
        ri := r(ai) ;
        rs := R(child(n,2), ri) ;
        qi := q(ls) ;
        qs := Q(child(n,1), qi) ;
        return f(qs) ;
    default:
        error
    end
end ;
```

图 4.16 图 4.15的依赖性决定了子结点的访问次序

4.5.3 多次遍历

前面介绍的方法可以延伸到需多次遍历分析树才能完成翻译的情况。这时为每个非终结符的每个综合属性建立一个函数(当然有些综合属性可以形成一组,放在一个函数中计算)。下面先用一个例子来说明一次遍历不能完成翻译的情况。

例 4.18 表 4.13的语法制导定义是标识符"重载"概念(见 5.6 节)的抽象。重载的标识符有一组可能的类型,由此引起表达式也有一组可能的类型,这时上下文信息用来从每个子表达式的可能类型中确定一个类型作为它的类型。可以先自下而上扫描,综合出可能类型集合,然后自上而下扫描,缩小可能类型集合到确定的惟一类型。

表 4.13 综合属性 s 和 t 不能一起计算

产生式	语义规则
$S \rightarrow E$	$E.i := g(E.s)$; $S.r := E.t$
$E \rightarrow E_1 E_2$	$E.s := fs(E_1.s, E_2.s)$; $E_1.i := f1(E.i)$; $E_2.i := f2(E.i)$; $E.t := ft(E_1.t, E_2.t)$
$E \rightarrow id$	$E.s := id.s$; $E.t := h(E.i)$

表 4.13的语义规则可以这样理解:综合属性 s 代表可能类型集合,继承属性 i 代表上下文信息,另一个综合属性 t 代表子表达式的类型或生成的代码,它不能和 s 在同一遍扫描

中完成计算。表 4.13 的各个产生式的依赖图见图 4.17。

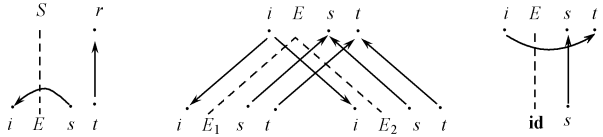


图 4.17 表 4.13 的各个产生式的依赖图

从各个产生式的依赖图还看不出属性的计算次序。但是如果把它们拼成图 4.18 的分析树的依赖图,就可以看出非终结符 E 的每个实例的属性必须以 E.s,E.i 和 E.t 的次序进行计算:首先自下而上扫描计算属性 s,然后自上而下扫描计算属性 i,最后自下而上扫描计算属性 t。

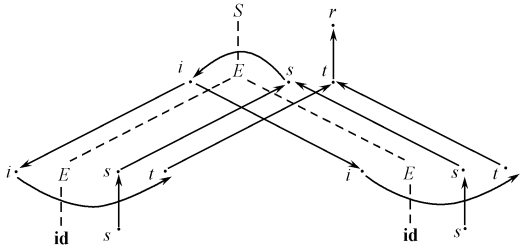


图 4.18 分析树的依赖图

在多次遍历的计算过程中,综合属性的函数以某些继承属性为参数。一般来说,如果综合属性 A.a 依赖于继承属性 A.b,那么 A.a 的函数以 A.b 为参数。我们还是举例来说明它们的使用。

例 4.19 图 4.19 的函数 *Es 和 Et 分别返回标记为 E 的结点 n 的一个综合属性。和* 4.5.2 节一样,非终结符的每个函数按所用的产生式分各种情况;不同的是,在每一种情况下执行的代码只计算表 4.13 中与该综合属性有关的语义规则。

从图 4.18 可以知道,分析树中 E 结点的属性 E.t 依赖于 E.i,因此继承属性 i 作为函数 Et 的参数。因为属性 E.s 不依赖于任何其他属性,所以函数 Es 没有对应属性值的参数。

多次遍历能适用的一类语法制导定义叫做“强无环”的定义。非形式地说,强无环是指任何分析树的属性依赖图都是无环的。对于该类定义,对一个非终结符的所有结点来说,它们的属性都可以按同样的(偏)序计算。

```
function Es(r) ;
begin
    case 在结点 n 的产生式 of
      E→E1 E2 :
        s1 := Es(child(n,1)) ;
        s2 := Es(child(n,2)) ;
        return fs(s1, s2) ;
      E→id :
        return id.s ;
    default:
        error
    end
end;
```

```
function Et(n, i) ;
begin
case 在结点 n 的产生式 of
      E→E1 E2 :
          i1 := fi1(i) ;
          t1 := Et(child(n,1), i1) ;
          i2 := fi2(i) ;
          t2 := Et(child(n,2), i2) ;
          return ft(t1, t2) ;
        E→id :
          return h(i) ;
      default:
          error
    end
end;
```

```
function Sr(n) ;
begin
    s := Es(child(n,1)) ;
    i := g(s) ;
    t := Et(child(n,1), i) ;
    return t
end;
```

图 4.19 表 4.13 中综合属性的函数

习 题 4

4.1 根据表 4.1 的语法制导定义,为输入表达式 5*(4*3+2) 构造注释分析树。

4.2 构造表达式((a*b) + (c)) 的分析树和语法树:

(a) 根据表 4.3 的语法制导定义。

(b) 根据图 4.9 的翻译方案。

4.3 为文法

S→(L) | a

L→L , S | 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。

4.4 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。

$E \rightarrow E + T \mid T$

$T \rightarrow \text{num.num} \mid \text{num}$

(a) 给出决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,既决定类型,又把表达式翻译成后缀表示。使用一元算符 intertoreal 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。

4.5 给出对表达式求导数的语法制导定义,表达式由+和*作用于变量 *x* 和常数组成,如 $x * 3 * x + x * x$,并假定没有任何化简,例如将 $3 * x$ 翻译成 $3 * 1 + 0 * x$。

*4.6 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。

4.7 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。

$S \rightarrow L.L \mid L$

$L \rightarrow LB \mid B$

$B \rightarrow 0 \mid 1$

(a) 用综合属性决定 S.val。

(b) 用语法制导定义决定 S.val。在该定义中,B 的惟一综合属性是 c,它给出由 B 产生的位对最终值的贡献。例如,101.101 的最前一位和最后一位对值5.625的贡献分别是 4 和 0.125。

4.8 重写例 4.3 的语法制导定义的基础文法,使得类型信息能够仅用综合属性就能完成。

4.9 由下列文法产生的表达式包括赋值表达式。

$S \rightarrow E$

$E \rightarrow E \doteqdot E \mid E + E \mid (E) \mid \text{id}$

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。

4.10 文法如下:

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

(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 2 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。

4.11 给出一个翻译方案,它检查同样的标识符不能在标识符表中出现两次。

4.12 程序的文法如下:

$P \rightarrow D$

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

(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。

4.13 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。

$E \rightarrow E_1 + T$	$E.nptr \doteq mknode(+, E_1.nptr, T.nptr)$
$E \rightarrow E_1 - T$	$E.nptr \doteq mknode(-, E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr \doteq T.nptr$
$T \rightarrow (E)$	$T.nptr \doteq E.nptr$
$T \rightarrow \text{id}$	$T.nptr \doteq mkleaf(\text{id}, \text{id.entry})$
$T \rightarrow \text{num}$	$T.nptr \doteq mkleaf(\text{num}, \text{num.val})$

4.14 给出例 4.10 中对应非终结符 *T 的翻译函数。*

4.15 *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:*

$D \rightarrow L\ :\ T$	$L.in \doteq T.type$
$L \rightarrow L_1, \text{id}$	$addtype(\text{id.entry}, L.in);\ L_1.in \doteq L.in$
$L \rightarrow \text{id}$	$addtype(\text{id.entry}, L.in)$
$T \rightarrow \text{integer}$	$T.type \doteq integer$
$T \rightarrow \text{real}$	$T.type \doteq real$

用先分析后翻译的方法,构造非终结符 D 和 L 的翻译函数。

*4.16 假定有 *L 属性定义,它的基础文法是* LL(1) 的,或者是一个能解决其二义性并且为之构造预测分析器的文法,证明可以把继承属性和综合属性放在由预测分析表驱动的自上而下分析器的分析栈中。

*4.17 证明在 LL(1) 文法的任何地方加上可区别的标记非终结符后,结果文法仍然是 LR(1) 的。

第5章　类型检查

编译器必须检查源程序是否满足源语言在语法和语义两个方面的约定。这种检查叫做静态检查(以区别在目标程序运行时的动态检查),它诊断和报告程序错误。静态检查的例子有:

(1) 类型检查

如果算符作用于不相容的运算对象,编译器会报告错误。例如数组变量和函数变量相加。

(2) 控制流检查

对于任何引起控制流离开一个结构的语句,程序中必须有该控制转移可以转到的地方。例如,C的break语句引起控制离开最小包围的while, for或switch语句,如果这样的包围语句不存在,则是一个错误。

(3) 惟一性检查

有些场合,对象必须正好被定义一次。例如,在Pascal语言中,标识符必须惟一地声明,case语句的分情形常量必须有区别,枚举类型的元素不能重复。

(4) 关联名字检查

有时,同样的名字必须出现两次或更多次。例如,在Ada语言中,循环或程序块可以有名字出现在它的开头和结尾。编译器必须检查两个地方是否使用同样的名字。

本章集中于类型检查。上面的列举表明,大多数其他静态检查是一些简单的工作,可以用上章所讲的技术实现。其中有些工作可以并入编译器的其他部分,例如,在把名字的信息填入符号表时,可以检查该名字声明的惟一性。许多Pascal编译器把静态检查和中间代码生成组织在分析的同时完成。对于更复杂的结构,如Ada中的一些结构,在分析和中间代码生成之间加一遍类型检查可能要方便一些,见图5.1。

图5.1 类型检查器的位置

5.1 类型在程序设计语言中的作用

5.1.1 引言

一个程序变量在程序执行期间的值可以设想为有一个范围,这个范围的一个界叫做该变量的类型。例如,类型 Boolean 的变量 x 在程序每次运行时的值只能是布尔值。如果 x 有类型 Boolean,那么布尔表达式 not (x) 在程序每次运行时都有意义。变量都被给定类型的语言叫做类型化语言 (typed language)。

语言若不限制变量值的范围,则被称作未类型化的语言 (untyped language),它们没有类型,或者说仅有一个包含所有值的泛类型。在这些语言中,一个运算可以作用到任意的运算对象,其结果可能是一个有意义的值、一个错误、一个异常或一个未做说明的结果。

类型化语言的类型系统 (type system) 是该语言的一个组成部分,它始终监视着程序中变量的类型,通常还包括所有表达式的类型。一个类型系统主要由一组定型规则 (typing rule) 构成,这组规则用来给各种语言构造 (程序、语句、表达式等) 指派类型。

为语言设计类型系统的目的是什么?简单讲,类型系统的根本目的是防止程序运行时出现执行错误 (execution error)。整个类型系统用来判断程序是否为良行为的 (well behaved)。只有那些顺从类型系统的程序才被认为是类型化语言的真正程序,其他的程序在它们运行前都应该被抛弃。

类型系统的这个非形式化的陈述激发了对类型系统的研究,但是需要进一步准确描述。首先,一个难以捉摸的问题是,什么情况叫做执行错误?我们将在后面讨论它。当用一种程序设计语言所能表示的所有程序运行时都没有执行错误出现时,我们说该语言是类型可靠的 (type sound)。显然,我们希望通过适当的分析来对程序设计语言的类型可靠性做出准确的回答。这种分析将基于语言的类型系统,这样,类型系统的研究也需要形式化的方法。

类型系统的形式化需要研究精确的表示法和定义,需要一些形式性质的详细证明。形式化的类型系统提供了概念上的工具,用它们可以评判语言定义的一些重要方面的适当性。非形式语言的描述通常不能把语言的类型结构详细规范到不会出现歧义实现的程度,因此经常会出现同一个语言的不同编译器实现了略有区别的类型系统,我们在 5.5 节会给出一个例子。而且,许多语言定义被发现不是类型可靠的,甚至经过类型检查 (type check) 后接受的程序也会崩溃。理想的状况是,形式化的类型系统应该是类型化程序设计语言的定义的一部分。这样,依靠精确的规范,证明语言是类型可靠的才有可能。

一种语言由一个实质存在的类型系统来类型化,而不在乎类型是否实际出现在程序的

语法中。一种语言,如果类型是语法的一部分,那么该语言是显式类型化的;否则是隐式类型化的。主流语言中不存在纯隐式类型化的语言,但是 ML 和 Haskell 这样的语言支持编写忽略类型信息的程序片段,这些语言的类型系统会自动地给这些程序片段指派类型。

本节非形式地给出定型、执行错误和其他有关概念的术语,讨论所期望的类型系统的性质和好处,并且概述怎样将类型系统形式化。本节所用的术语并非绝对标准,因为不同来源的这些术语原来就不一致。一般来说,当引用运行时的概念时,避免用类型和定型,例如用动态检查代替动态定型。

5.1.2 执行错误和安全语言

程序执行错误最明显的征兆是出现意想不到的软件错误,如非法指令错误、非法内存访问和除数为零错误。在许多计算机系统结构上,这样的错误使得计算立即停止。然而,还有一些难以捉摸的错误,它们引起数据遭破坏而不立即出现征兆。因此执行错误可以分成两类:一类引起计算立即停止;另一类当时未引起注意,而后引发难以预见的行为。前者叫做会被捕获的错误(trapped error),后者叫做不会被捕获的错误(untrapped error)。区别这两类错误是有用的。

不会被捕获的错误的一个例子是不适当地访问一个合法的地址,例如,在没有运行时界检查的情况下访问越过数组末端的数据。不会被捕获的错误的另一个例子是跳到一个错误的地址,该地址开始的内存正好代表一个指令序列,使得该错误可能会有一段时间未引起注意。

一个程序段是安全的,如果它不可能引起不会被捕获的错误出现。所有可能的程序段都是安全的语言叫做安全语言(safe language)。未类型化的语言也可以通过运行时的检查来增强安全性。类型化语言可以通过静态检查(当然也可以使用静态检查和运行检查混合的方式)来拒绝一切有潜在不安全的程序,以增强语言的安全性。

虽然安全性是程序一个至关重要的性质,但是对一种类型化语言来说,我们很少关心它正好仅仅排除不会被捕获的错误。通常,类型化语言的目标是在排除所有不会被捕获错误的同时,也排除很多会被捕获的错误。

对任何一种语言,可以指定所有可能执行错误集合的一个子集作为禁止错误(forbidden error)。禁止错误应该包括所有不会被捕获的错误,再加上会被捕获的错误的一个子集。一个程序段被认为是良行为的,如果它不会引起任何禁止错误出现。反之称为是有不良行为的(bad behavior)。特别要强调的是,一个良行为的程序段是安全的。如果一个语言所有合法的程序都是良行为的,就说该语言是类型可靠的。注意,这里把类型可靠概念从不出现任何执行错误放宽到不出现任何禁止错误,如果禁止错误集合正好只包括不会被捕获的错误,那么类型可靠的语言和安全的语言是同一个意思。把该类型可靠的语言又称为强检查的

(strongly checked)。

上面提到,未类型化语言可以通过彻底的运行时详细检查来排除所有的禁止错误。(例如,它们检查所有数组的界,检查所有的除法操作,当禁止错误出现时,产生可恢复正常的异常。)这些语言的检查过程称为动态检查,Lisp 语言是一个这样的例子。这些语言虽然既没有静态检查,也没有类型系统,但是它们是强检查的。

上面也提到,类型可靠的语言可以通过静态检查来拒绝有不良行为的程序,从而保证程序的良好行为(不能通过静态检查的程序不是合法的程序,严格说,它们不能称为程序)。语言的类型系统就是用来支持这样的检查,静态检查的过程叫做类型检查(type checking),完成这种检查的算法叫做类型检查器。因此,对于类型可靠的语言,若它是类型化语言,我们又称它是强类型化的(strongly typed)语言。能够通过类型检查器的程序称为良类型的(well typed)。这样,对于强类型化的语言来说,良类型的程序有下面几点性质:

不会出现不会被捕获的错误(即是安全的);

不会出现已列入禁止错误的会被捕获的错误;

有可能出现其他会被捕获的错误,避免它们是程序员的责任。

不能通过类型检查器的程序被称为有类型缺陷的(ill typed),它可以表示程序确实有不良行为,或者简单地表示不能保证程序是良行为的。

静态检查的语言有 ML 和 Pascal 等(注意,Pascal 有一些不安全的特征)。

静态检查语言通常也需要一些运行时的测试以保证安全性。例如,数组的界通常是动态测试的。

从上面的这些定义可以知道,良行为的程序是安全的,安全性是一个比良行为更基本和更重要的性质。类型系统的基本目标是通过排除程序运行中所有不会被捕获的错误来保证语言的安全性,但是,大多数类型系统设计成保证更一般的良行为性质。于是,通过区分良类型和有类型缺陷的程序,一个类型系统宣称的目标通常是保证所有程序的良行为。

现在实际使用的众多语言中,一些静态检查的类型化语言并不能保证安全性。因为它们的禁止错误集合并没有包含所有不会被捕获的错误。这些语言可以婉转地称做弱检查的(weakly checked)或弱类型化的(weakly typed),其含义是,部分非安全运算能被静态地检查出来,还有一些没有被检查出来。这类语言类型弱化的程度在一个很大的范围内变化。例如,Pascal 仅在使用无标志的变体记录类型和函数型参数时是不安全的,而 C 语言有很多不安全的并且被广泛使用的特征,例如指针算术运算和类型强制。很有趣的是,人们为 C 程序员总结的戒律中约一半是直接针对 C 的弱检查的。在 C 的弱检查引起的问题中,一些问题已经在 C++ 中得以缓和,更多一些问题在 Java 中已得到解决。Modula-3 支持一些不安全特征,但是仅在显式标记为不安全的模块中,以防止安全模块输入不安全的接口。ML 是一个类型化的安全语言。

大多数未类型化高级语言是安全的,如 Lisp。否则,在没有编译时和运行时检查来避免数据遭破坏的话,程序设计将是令人沮丧的事情。汇编语言属于令人讨厌的未类型化的不安全语言范畴。

在语言设计的历史上,安全性考虑不足是出于效率上的原因,最典型的是 C 语言。C 以处理低级构造的灵活性著称,它的设计为了保证编程的灵活性而牺牲了安全性,它鼓励在安全的边缘进行编程。这使得 C 的程序效率很高,但是也使得它们缺乏安全性。为了排除程序中的安全漏洞,需要大量的调式工作。

对于安全性不足的语言,可以通过运行时的检查来提高安全性,但是运行检查的代价有时是非常昂贵的。另外,即使是做了细致静态分析的语言,获得彻底的安全性也需要运行时的代价。例如,一般来说,数组界检查是不可能完全在编译时删除的。从另一个角度看,为获得安全性所花的代价是值得的。安全性使得程序出现执行错误就会停止执行,这可以减少调试的时间。而且安全性保证运行时结构的完整性,因而使得无用存储单元收集(garbage collection,俗称垃圾收集,见第 10 章)可以完成。而无用存储单元收集大大降低代码开发的时间和代码的规模,虽然它需要一点运行时的代价。

于是,从历史上看,安全和不安全语言之间的选择可能最终与开发时间和执行时间之间的权衡相关。

但是,随着经济和社会越来越依赖于信息技术,关键的信息基础构造——运输、通信、金融市场、能源分配和卫生保健等,都将非常危险地依赖于不是一个管理机构范围内的计算基础和资源。在我们越来越依赖于这些信息基础构造的同时,提高系统对恶意攻击和有漏洞软件破坏的抵抗能力显得越来越重要。例如,对于用 C 编写的系统来说,用缓冲区溢出来对它进行的攻击,已占目前各种攻击的 50%,另外还有基于 C 程序格式串的安全漏洞而进行的攻击等等。因此,当前在安全和不安全语言之间的选择还与对系统安全性的要求相关。

5.1.3 类型化语言的优点

程序设计语言是否应该有类型?该问题仍然是某些讨论的一个主题。从可维护性的观点看,很明显,弱检查的不安全的类型化语言优于安全的非类型化语言(例如,C 和 Lisp 的比较)。几乎不用怀疑,用非类型化语言写的产品代码维护起来非常困难。从工程的观点看,类型化语言有下面一些优点。

(1) 开发的实惠

有了类型系统可以较早发现错误,例如整数和串相加。若类型系统设计得很好,类型检查可以发现大部分日常的程序设计错误,剩下的错误也很容易调试,因为大部分错误已经被排除。

对于大规模的软件开发来说,接口和模块有方法学上的优点,类型信息在这里可以组

织到程序模块的接口中。程序员可以一起讨论要实现的接口,然后分头编写要实现的对应代码。这些代码之间的相互依赖最小,并且代码可以局部地重新安排而不用担心对全局造成影响。

程序中的类型信息还具有文档作用。程序员声明标识符和表达式的类型,也就是告诉了我们所期望的值的部分信息,这对阅读程序是很有用的。

⑵ 编译的实惠

程序模块可以相互独立地编译,例如 Modula-2 和 Ada 的模块,每个模块仅依赖于其他模块的接口。这样,大系统的编译可以更有效,因为改变一个模块并不会引起其他模块的重新编译,至少在接口稳定的情况下是这样。

⑶ 运行的实惠

在编译时收集类型信息,保证了在编译时就能知道数据占用空间的大小,因而可得到更有效的空间安排和访问方式,提高了目标代码的运行效率。例如,像 Pascal 的记录、C 的结构和对象,其域或成员的偏移可以根据它们的类型信息静态地确定。

另外,一般来说,精确的类型信息在编译时可以保证运行时的运算都作用到适当的对象并且不需要昂贵的运行时的测试,从而提高程序运行的效率。例如在 ML 中,精确的类型信息可以删除在指针脱引用 (dereference) 中的 *nil* 检查。

上面我们提到,类型信息具有文档作用,但是它和其他形式的程序评注不同。一般来说,关于程序行为的评注可以从非形式的注解一直到用于定理证明的形式规范。类型处在该范围的中间:它们比程序注解精确,比形式规范容易理解。另外,类型系统应该是透明的:程序员应该能够很容易预言一个程序是否可通过类型检查,如果它不能通过类型检查,那么其原因应该是明显的。

5.2 描述类型系统的语言

类型系统主要用来说明程序设计语言的定型规则,它独立于类型检查算法。这类似于形式文法可用来描述程序设计语言的语法,但这种描述独立于语法分析算法。

把类型系统从类型检查算法中分离出来是有用的:类型系统属于语言定义,而类型检查算法属于编译器。用类型系统 (而不是用编译器的算法) 很容易解释语言在类型方面的规定。而且,不同的编译器对同一个类型系统可能使用不同的类型检查算法。

从技术上说,定义一个类型系统,通常的设计目标是允许有效的类型检查算法。也有这样的类型系统,它只存在难以实施的类型检查算法,或者完全没有类型检查算法,我们的设计要避免这种情况。

类型系统的基本概念实质上可用于所有的计算范型(paradigms),包括函数式语言、命令式语言和并行语言等等。一些定型规则还可以不作修改就用于不同的范型,例如对函数来说,不管是函数式语言还是命令式语言,不管是值调用还是换名调用,其基本定型规则都一样。

形式化的类型系统是程序设计语言手册中非形式化的类型系统的一个数学描述。一旦类型系统形式化了,可以尝试证明语言是否类型可靠。如果这样的可靠性定理成立,可以说该语言的类型系统是可靠的,即一个类型化语言的所有程序都是良行为的和它的类型系统是可靠的,这两者表达的是同一件事。本书虽然不讨论类型系统可靠性的证明问题,但是还是尽量采用形式化的方法来描述类型系统。

5.2.1 定型断言

本小节引入描述类型系统的形式化方法。一个类型系统的描述从描述一组叫做断言(assertion)的形式化的论述开始。一个典型的断言具有形式

$\Gamma ? S$ *S的所有自由变量都声明在Γ中*

这里Γ是一个静态定型环境(static typing environment),例如形式为 $x_1:T_1, \ldots, x_n:T_n$ 的不同的变量和它们的类型的有序表,这相当于编译器的符号表。在对程序段进行类型检查的时候,定型规则总是针对它的一个静态类型环境来进行形式化。在Γ中声明的这组变量 $x_1, \ldots, x_n$ 用 $dom(\Gamma)$ 表示。变量个数为零的空环境用 $\varnothing$ 表示。S的形式随断言形式的不同而不同,但是S的所有自由变量必须在Γ中声明。

断言有三种形式,环境断言、语法断言和定型断言。例如,下面这个环境断言直接声称一个环境是合式的(well formed):

$\Gamma ?$ *Γ是合式的(即它是适当地构造的)*

语法断言用于规定类型表达式的语法。我们知道语言构造的类型可以用"类型表达式"来表示。基本类型是类型表达式,由类型构造器作用于类型表达式而形成的表达式也是类型表达式。例如:

$\Gamma ? Nat$

表示在静态定型环境Γ下,Nat是一个类型表达式。在程序设计语言的语法中,类型表达式也是用文法描述的(见下面图5.2),但是在设计类型系统时,通常都是用本节的方式来描述,其优点是能够用Γ来表示上下文有关的约束。

对我们现在的目的,最重要的断言是定型断言(typing assertion),它声称对于M的所有自由变量的一个静态定型环境Γ,M具有类型T。它的形式是:

$\Gamma ? M : T$ *M在Γ中具有类型T*

例如

? *true* : *Bool* *true具有类型 Bool*

x : *Nat* ? *x*+1 : Nat *x*+1的类型是 *Nat,只要 x具有类型 Nat*

任何一个断言可以看成是有效的(valid)(例如 Γ? *true* : *Bool*)或无效的(invalid)(例如 Γ? *true* : *Nat*)。*有效性可用来使良类型化程序的概念形式化。有效和无效断言之间的区*别可以用不同的方式表示,这里使用的是基于定型规则的风格,它便于论述和用来证明类型系统的引理和定理。而且,定型规则是高度模块化的:不同类型构造器的规则可以分开写。因此,定型规则相对容易阅读和理解。

5.2.2 定型规则

推理规则是在一组已知有效的断言的基础上,声称某个断言的有效性。它的一般形式是

(规则名) (注释) Γ1? *S*1, ..., Γ*n*? *Sn*] Γ? *S* (注释)

在每条推理规则中,]的左边是一组称为前提的断言,]的右边是一个称为结论的断言。当所有的前提都满足时,结论一定成立。前提个数也可能为零,这样的规则叫做公理。每条规则有一个名称。(由约定,名称的第一个词由结论断言确定。例如,形式为(Val...)的规则名用于这样的规则,其结论是一个值的定型断言。需要时,限制规则使用的条件,还有在规则中用的一些缩写,都可以列在规则名或规则的旁边。)需要注意的是,通常的描述方式是前提和结论用横线隔成上下两部分,为了排版的方便,在此采用记号]。

针对三种不同的断言形式,我们有三类推理规则。例如环境规则:

(Env)] ?

表示空环境是合式的,阐明这一点的基本规则没有前提,它是一个公理。

下面的语法规则本质上是说,在任何合适环境 Γ 下,Bool是一个类型。

(Type Bool) Γ?] Γ? *Bool*

推理规则的结论是定型断言的话,称之为定型规则。例如,下面的第一条规则是说,在任何合式环境下,任何一个自然数都是类型为 Nat的表达式。第二条规则是说,两个自然数表达式 M和 N相加的结果是一个自然数表达式 M+ *N,而且 M和 N的环境 Γ 继续作为 M* + *N的环境。*

(Val *n*) (*n* = 0, 1, ...) Γ?] Γ? *n* : *Nat*

(Val +) Γ? *M* : *Nat*, Γ? *N* : *Nat*] Γ? *M*+ *N* : *Nat*

一个语言的定型规则(以及它的环境规则和语法规则)的集合构成它的(形式化的)类型系统。具有类型系统的语言叫做类型化语言。从技术上讲,类型系统符合形式证明系统的一般框架,一组定型规则是用于执行一步步的演绎,类型系统中的这种演绎关系到程序的定型。

5.2.3 类型检查和类型推断

在一个类型系统中,通过推导得到的断言是该类型系统的有效断言(valid assertion),也就是说,一个有效断言是通过正确地应用类型规则可以得到的断言。形式化的演绎证明在其他课程中学习过,在此不做介绍,将在5.4节举例。

在一个类型系统中,如果存在一个类型 T 使得 Γ? *E :T是一个有效的断言,那么 E 对*环境 Γ 来说是良定型的,并且它的类型是 T。

用语法制导的方式,根据上下文有关的定型规则来判定程序构造是否为良类型的程序构造的过程,通常被称作类型检查。例如,因为 Γ? *E :T的任何推导必须根据 E 的结构,*因此可以用类型检查来判断一个语法构造 E 是否为良类型。若程序构造的形式是函数 f(x:t) = *E,则该函数头规定了 x 的类型。此时,在对 E 进行类型检查时,我们已经有了 x 的类型。但是,如果忽略这个类型,把该函数写成 f(x) = E,那么就不清楚该给 x 什么*类型,此时需要对 E 进行更加复杂的分析。于是,我们把类型信息不完全的情况下的定型判定问题叫做类型推断(type inference)问题。区别类型推断和类型检查是有用的,但是精确区分它们是困难的。

如果不存在 E 的任何推导,我们就说 E 是不可定型的,通常说它有类型错误或者类型缺陷。

程序构造的类型检查或类型推断问题,与所使用的类型系统密切相关。算法可能非常容易或非常困难,甚至不可能实现,取决于所使用的类型系统。另外,如果能够找到这样的算法,最快的算法可能非常有效,也可能是令人失望地慢。类型系统的实际效用依赖于是否存在好的类型检查或类型推断算法。

对于像 Pascal 这样的显式类型化命令式语言,类型检查问题还算容易解决。对于像 ML 这样的隐式类型化语言,类型推断问题就相当困难。其基本算法已很好地理解和广泛地使用,但是在实际中使用的该算法的版本相当复杂,并且仍然在研究中。

5.3 简单类型检查器的说明

本节将考察一个简单语言的一个具体的类型系统和它的一个语法制导的类型检查器。在此语言中,每个标识符的类型必须在它使用前先声明。这个类型检查器是一个翻译方案,它从子表达式的类型给出表达式的类型。该类型检查器能够处理数组、指针、语句和函数。

5.3.1 一个简单的语言

图5.2是该语言的文法,其中 *P* 表示程序,它由一段声明 *D* 和随后的一个语句 *S* 组成。

P→ *D* ; *S*
D→ *D* ; *D* | id : T
T→boolean | integer | array [num] of *T* | ↑ *T* | *T*′→ ′*T*
S→id ≒ *E* | if *E* then *S* | while *E* do *S* | *S* ; *S*
E→truth | num | id | *E* mod *E* | *E* [*E*] | *E*↑ | *E* (*E*)

图5.2 源语言文法

图5.2第3行是描述类型的产生式。经从 T 开始的语法推导得到的都是类型表达式,它们可以像算术表达式那样用树来描述。每个类型表达式都是该语言的一个类型,因此经从 T 开始的推导,可以得到该语言所有的类型。

有关图5.2文法的其他一些解释,穿插在下面各小节。由该文法生成的程序实例有

```
i : integer;
j : integer;
j ≒ i mod 2000
```

在该文法中,我们选择 mod 作为二元算符的代表。

5.3.2 类型系统

事实上,语言的设计和其类型系统的设计是同时进行、相互影响的。类型系统的设计包括三部分。首先,最简单的是环境规则的设计。表达式类型的确定和环境有关,我们首先给出同文法中的类型声明有关的两条环境规则。第一条规则是说空环境 是合适的,第二条规则是说,一个新的变量声明在环境中增加一个变量到类型的映射。

(Env)] ?
(Decl. Var) *Γ*? *T*, id ∉ *dom* (*Γ*)] *Γ*, id : T ?

类型系统设计的第二部分是设计类型表达式的语法,即合式类型表达式的定义。类型表达式的语法已经体现在5.3.1节的文法中。如果要用规则来表示的话,可以写成6条语法规则。

下面3条语法规则表示,在任何合式的环境下,Boolean、integer 和 void 都是类型表达式。其中 void 用在语句等情况,表示这样的语法结构没有类型,它和类型错误是两回事。

(Type Bool) *Γ*?] *Γ*? boolean

(Type Int) $\Gamma ?$] $\Gamma ?$ integer

(Type Void) $\Gamma ?$] $\Gamma ?$ void

下面3条语法规则说明构造类型的语法。指针、数组和函数都是类型构造器。

规则(Type Ref)说明,如果T是类型表达式,那么类型表达式↑*T*表示指向类型*T*的对象的指针类型。就像Pascal语言那样,类型声明中的前缀算符↑用于指针类型,表达式中的后缀算符↑用于脱引用。

规则(Type Array)说明,如果T是类型表达式,那么array [N] of *T*表示成分类型为*T*的数组类型。假定所有数组的下标都从1开始,那么该数组的下标集合是1..N。注意这里的类型表达式和文法中的区别,我们的文法为简单起见,没有整数出现在其中,而是用一个记号num代表了所有的整数,用num的属性来代表它每次出现时的值。

规则(Type Function)说,类型表达式 $T_1 \rightarrow T_2$ 表示定义域类型为 T_1 和值域类型为 T_2 的函数类型。为简单起见,我们的文法只考虑了一元函数。

(Type Ref) $\Gamma ?\ T$] $\Gamma ?\ \uparrow T$

(Type Array) $\Gamma ?\ T, \Gamma ?\ N : \text{integer}$] $\Gamma ?$ array [N] of T

(Type Function) $\Gamma ?\ T_1, \Gamma ?\ T_2$] $\Gamma ?\ T_1 \rightarrow T_2$

类型系统设计的第三部分是给各种形式的表达式和语句指派类型表达式,也就是设计定型规则,这是类型系统最重要的部分。下面若干条定型规则用来确定表达式的类型。

(Exp Truth) $\Gamma ?$] $\Gamma ?$ truth : boolean

(Exp Num) $\Gamma ?$] $\Gamma ?$ num : integer

(Exp Id) $\Gamma_1, \text{id} : T, \Gamma_2 ?$] $\Gamma_1, \text{id} : T, \Gamma_2 ?\ \text{id} : T$

(Exp Mod) $\Gamma ?\ E_1 : \text{integer},$
$\Gamma ?\ E_2 : \text{integer}$
] $\Gamma ?\ E_1 \bmod E_2 : \text{integer}$

(Exp Index) $\Gamma ?\ E_1 : \text{array [N] of } T,$
$\Gamma ?\ E_2 : \text{integer}$
] $\Gamma ?\ E_1 [E_2] : T$

(Exp Deref) $\Gamma ?\ E : \uparrow T$] $\Gamma ?\ E\uparrow : T$

(Exp FunCall) $\Gamma ?\ E_1 : T_1 \rightarrow T_2,$
$\Gamma ?\ E_2 : T_1$
] $\Gamma ?\ E_1 (E_2) : T_2$

规则(Exp Id)表示,如果环境中有id的类型说明,则id就是这个类型。对于下标变量引用,为了使规则简明起见,在规则中没有考虑下标表达式的越界问题。越界问题可以由运行时的数组界检查完成。

在我们的文法中,语句这样的语言结构没有类型。在类型系统设计中,可以给它们以特殊的基本类型 void,这样便于我们从程序 P 的属性判断它是否有类型错误。下面的规则用来确定语句和程序的类型。

(State Assign)	Γ? id : T, Γ? E : T	]	Γ? id := E : void
(State If)	Γ? E : boolean, Γ? S : void	]	Γ? if E then S : void
(State While)	Γ? E : boolean, Γ? S : void	]	Γ? while E do S : void
(State Seq)	Γ? S_1 : void, Γ? S_2 : void	]	Γ? S_1 ; S_2 : void
(Prog)	Γ? S : void	]	Γ? D ; S : void

5.3.3 类型检查

根据上一小节的类型系统来设计类型检查器,这个类型检查器的设计是直截了当的,我们用翻译方案来表示。

在下面的翻译方案中,文法符号的 type 属性给出它的类型,编译器的符号表对应类型系统中的环境 Γ。addtype 函数将 id 的类型添到符号表中,lookup 函数作用于 e 时表示取符号表中 e 指向的条目的类型。这里的类型表达式在表示上和类型系统中的略有区别,例如 ↑ T 写成 pointer (T) 。

因为类型错误不像语法错误那样直观,所以对类型检查器来说,发现错误时应做得更合理。除了报告错误的性质和位置,至少还希望类型检查器能从错误中恢复过来,以便检查剩余输入中的各种错误。这样,在类型检查器中我们增加一个特别的类型 type_error,用来表示有类型错误的语法结构。为简单起见,我们没有考虑如何报告类型错误。

先来看声明语句的语义动作。这些语义动作构造类型表达式,并将它们保存在 T 的综合属性 type 中。

$D \rightarrow D$; D	
$D \rightarrow$ id : T	{addtype (id.entry, T.type)}
$T \rightarrow$ boolean	{T.type := *boolean*}
$T \rightarrow$ integer	{T.type := *integer*}
$T \rightarrow \uparrow T_1$	{T.type := *pointer* (T1. *type*) }
$T \rightarrow$ array [num] of T_1	{T.type := *array* (num. val., T_1. *type*) }
$T \rightarrow T_1$ '→' T_2	{T.type := T1. *type* → T2. *type*}

在表达式的语义动作中,E 的综合属性 type 给出了 E 产生的表达式的类型表达式。如果碰到类型错误,该表达式的类型表达式是 type_error。如果表达式 E 的某个子表达式的类型是 type_error,那么 type_error 从这个子结构一直传到 E。

$E \rightarrow$ truth	{E.type := *boolean*}
$E \rightarrow$ num	{E.type := *integer*}
$E \rightarrow$ id	{E.type := *lookup*(id.entry)}
$E \rightarrow E_1$ mod E_2	{E.type := if E_1.*type* = *integer* and E_2.*type* = *integer* then *integer* else *type_error*}
$E \rightarrow E_1$ [E_2]	{E.type := if E_2.*type* = *integer* and E_1.*type* = *array*(*s*, *t*) then *t* else *type_error*}
$E \rightarrow E_1 \uparrow$	{E.type := if E_1.*type* = *pointer*(*t*) then *t* else *type_error*}
$E \rightarrow E_1$ (E_2)	{E.type := if E_2.type = *s* and E_1.*type* = $s \rightarrow t$ then *t* else *type_error*}

对于数组引用 E_1 [E_2],下标表达式必须有 integer 类型。此时,结果类型是从 E_1 的类型 array(s, t)中得到的元素类型 t。这里没有用数组的下标集合 s,因为数组界的检查在运行时完成。

对于函数调用 E_1 (E_2),E_1 的类型必须是从 E_2 的类型 s 到某个值域类型 t 的函数类型 $s \rightarrow t$,E_1 (E_2) 的类型是 t。在这里只考虑了一元函数,后面介绍了积类型后,把类型检查推广到多元函数是件容易的事情。

从语句和程序产生式的语义规则可以看出,在没有类型错误时,整个程序的类型是 void,否则是 *type_error*。

$S \rightarrow$ id := E	{S.type := if id.type = E.*type* then void else *type_error*}
$S \rightarrow$ if E then S_1	{S.type := if E.*type* = *boolean* then S_1.*type* else *type_error*}
$S \rightarrow$ while E do S_1	{S.type := if E.*type* = *boolean* then S_1.*type* else *type_error*}
$S \rightarrow S_1$; S_2	{S.type := if S_1.*type* = void and S_2.*type* = void then void else *type_error*}
$P \rightarrow D$; S	{P.type := if S.*type* = void then void else *type_error*}

5.3.4 类型转换

*考虑表达式 x+ i, 其中 x是实型, i是整型。因为在计算机中实数和整数有不同的表*示,并且对实数和整数有不同的机器指令,因此编译器可能首先把其中一个运算对象进行类型转换,以保证相加的两个对象有同样的类型。

语言的类型系统会指出哪些不同类型的运算是允许的,语言的实现会考虑是否要进行类型转换。例如,当整数表达式赋给实型变量时,应该把赋值号右边对象的类型转换成左边对象的类型。在表达式中,通常是把整数转换成实数,然后在一对实型对象上进行实数运算。编译器的类型检查器可用来在源程序的中间表示中插入这些转换操作。例如, x+ *i* 的后缀左可以是

x i inttoreal real+

这里,首先由 inttoreal 算符把 i 从整数转换成实数,然后由 real+ 将该转换结果和 x 进行实数加。

如果从一个类型转换到另一类型可以由编译器自动完成,这样的转换称为隐式的。例如,在 C 语言的算术表达式中,编译器把 ASCII 字符强制 (即隐式转换) 到 0 和 127 之间的整数。在许多语言中,要求隐式转换原则上不丢失信息,例如,整数可以转变为实数,但反过来不行。不过,当实数和整数用同样多比特表示时,还是有可能丢失信息的。

如果转换必须由程序员写出,那么这种转换叫做显式的。Ada 的所有转换都是显式的。显式转换就是一种函数调用,因此它们对类型检查器不提出新问题。

例 5.1 考虑把算术算符 op 作用于常数和标识符形成的表达式,如图 5.3 的文法那样。假定有实数和整数两个类型,必要时整数转换成实数。非终结符 E 的属性 type 可以是 *integer* 或 *real*,*类型检查在图* 5.3 中给出 (读者应该能写出相关的定型规则) 。

$E \rightarrow$ num	{E.type := *integer*}
$E \rightarrow$ num.num	{E.type := *real*}
$E \rightarrow$ id	{E.type := *lookup* (id.entry)}
$E \rightarrow E_1$ op E_2	{E.type := if E_1.type = *integer* and E_2.type = *integer* then *integer* else if E_1.type = *integer* and E_2.type = *real* then *real* else if E_1.type = *real* and E_2.type = *integer* then *real* else if E_1.type = *real* and E_2.type = *real* then *real* else *type_error*}

图 5.3 从整型到实型的类型检查

常数的隐式转换可以在编译时完成,并且常常可以使目标程序的运行时间明显减少。

*5.4 多态函数

普通的函数要求变元有惟一的类型,即它体中的语句只能在变元类型固定的情况下执行。多态函数允许变元有不同的类型,即它体中的语句可以在变元类型不同的情况下执行。术语"多态"也可用于以不同类型的变元执行的代码段,所以我们既谈多态函数,也谈多态算符。

内部定义的算符,如数组索引、函数作用和通过指针的间接访问通常都是多态的,因为它们并没有限于特定类型的数组、函数和指针。例如,C 语言的参考手册关于指针 & 的论述是:"如果运算对象的类型是'...',那么结果类型是指向'...'的指针"。因为任何类型都可以代替"...",所以 C 的算符 & 是多态的。

在 Ada 中,"类属"函数是多态的,但多态性在 Ada 中是受限制的。因为术语"类属"也用于重载的函数和函数变元的强制,所以我们避免使用这个术语。

本节讨论为有多态函数的语言设计类型检查器时出现的问题。为了处理多态性,我们拓广类型表达式集合,增加含类型变量的表达式。类型变量的引入又产生和类型表达式等价有关的一些有趣的算法问题。

5.4.1 为什么要使用多态函数

多态函数的吸引力在于它便于实现那些操作于某种数据结构,而不必顾及它成分类型的算法。例如,用多态函数很容易写出求表长度的程序而不必管表元的类型。

像 Pascal 这样的语言,它们要求给出函数参数类型的完整说明,所以确定整数链表长度的函数不能用于实数表。求整数表长度的 Pascal 代码见图 5.4。函数 length 顺着表的 next 链直至 nil 为止。虽然函数代码不以任何方式依赖表元信息的类型,但是 Pascal 要求在编写函数 length 时声明 info 域的类型。

在有多态函数的语言例如 ML 中,函数 length 可以编写成适用于任何类型的表,如图 5.5 所示。关键字 fun 表示 length 是函数。函数 null 和 tl 是预定义的,null 测试表是否为空,tl 返回拿开第一个表元后剩下的部分。用图 5.5 的定义,函数 length 的下列运用都产生值 3。

```
length ([sun, mon, tue]);
length ([10, 9, 8]);
```

在第一个运用中,length 作用于串表;在第二个运用中,它作用于整数表。

```
type link = ↑cell ;
     cell = record
         info : integer ;
         next : link
     end ;
function length (lptr : link) : integer ;
    var len : integer ;
    begin
       len := 0 ;
       while lptr <> nil do begin
            len := len+1 ;
            lptr := lptr↑.next
       end ;
       length := len
    end ;
```

图5.4 求表长度的Pascal程序

```
fun length (lptr) =
    if null (lptr) then 0
    else length (tl (lptr)) +1 ;
```

图5.5 求表长度的ML程序

要想完成多态函数的类型检查,首先要能给出它们的类型描述。像length函数,其参数表的表元类型可以任意,显然应该用变量表示。这样,在类型表达式中需要引入类型变量。允许使用类型变量并引入全称量词∀后,length的类型可以写成∀ $\alpha.\ list(\alpha) \rightarrow integer$。

5.4.2 类型变量

下面允许类型表达式包含变量,变量的值是类型表达式。本节的其余部分用希腊字母 $\alpha, \beta, \ldots\ldots$作为类型变量,用于类型表达式中。

类型变量除了可出现在多态函数的类型表达式中外,还可以用在其他方面,例如它便于我们讨论未知类型。在不要求标识符的声明先于使用的语言中,类型变量的一个重要运用是检查标识符使用的一致性。类型变量代表没有声明的标识符的类型,查看程序可以知道没有声明的程序变量的使用情况,如果它在某个语句作为整型使用,而在另一语句作为数组使用,那么可以报告使用不一致的错误。相反,如果变量总是作为整型使用,那么不仅能保证它使用的一致性,而且能推断出它的类型必定是整型。

例5.2 类型推断技术可用于C和Pascal这样的语言,在编译时补上程序中欠缺的类型信息。图5.6的代码段给出过程mlist,它有一个参数p,p本身是个过程。从过程mlist的第一行仅能知道p是一个过程,不能确定p的变元个数和它们的类型。C和Pascal的参考手册允许这种类型不完整的p声明。

过程 mlist 把参数 p 用于链表的每个表元。例如,p 可以用于给表元中整型变量置初值或打印其值。尽管 p 的变元类型没有指明,但是从表达式 p (lptr) 中 p 的使用可以推断出 p 的类型必须是

link→void

对于 mlist 的任何调用,如果过程参数不是这个类型,则是一个错误。过程可以看成是没有返回值的函数,所以它的结果类型是 void。

```
type link = ↑cell;
procedure mlist (lptr : link; procedure p) ;
begin
   while lptr < > nil do begin
      p (lptr) ;
      lptr := lptr↑.next
   end
end;
```

图 5.6 带过程参数 p 的过程 mlist

类型推断技术和类型检查技术有很多共同的地方。在这两种情况下都要处理含变量的类型表达式。类似于下例的推导可以由类型检查器用来推断变量表示的类型。

例 5.3 在下面假想的程序中,我们可以推断出多态函数 deref 的类型。函数 deref 和 Pascal 指针的脱引用算符↑有同样的作用。

```
function deref (p) ;
begin
        return p↑
end;
```

看见第一行

```
function deref (p) ;
```

时,对 p 的类型一无所知,因而用类型变量 β 代表它的类型。由定义,后缀算符↑作用于一个对象的指针,返回该对象。因为↑算符在表达式 p↑ 中用于 p,p 必定是指向未知类型 α 的指针,所以可以断定

$\beta = pointer(\alpha)$

其中 α 是另一个类型变量。而且,表达式 p↑ 有类型 α,因此函数 deref 的类型表达式可以写成

$\forall \alpha.pointer(\alpha) \rightarrow \alpha$

5.4.3 一个含多态函数的语言

到目前为止,多态函数是指它们可以用“不同类型”的变元执行。多态函数可以作用的类型集合的精确定义可以用符号P 表示,其含义是“对任何类型”。非形式地,有符号P 的类型表达式称为“多态的类型”。

用于检查多态函数的抽象语言由图 5.7 的文法产生。之所以称为抽象语言,是因为我们让非终结符 *T* 直接产生类型表达式,以简化语言和突出要讨论的问题。

P→ *D* ; *E*
D→ *D* ; *D* | id : Q
Q→P type_variable. Q | *T*
T→ *T*′→′*T*
| *T*× *T*
| unary_constructor (T)
| basic_type
| type_variable
| (T)
E→ *E* (*E*) | *E*, *E* | id

图 5.7 有多态函数的语言的文法

该文法对图 5.2 的文法做了一些简化。首先,程序是由类型声明和表达式(而不是语句)组成,这样可以免去考虑语句,因为语句部分的类型检查不会因加入多态类型而有什么变化。表达式也做了简化,略去了和问题无关的一些选择。用 basic_type 类型来表示像 boolean 和 integer 这样的基本类型,用 unary_constructor 表示一元构造器,它允许写出像 *pointer (integer) 和 list (integer) 形式的类型。*

我们增加了多态类型函数的声明,还增加了笛卡儿积类型的声明。积类型可以用来表示多元函数类型,产生式 E→ *E*, *E* 用以推导多个实参表达式的情况。*T*→ (T) 仅用来组合类型。

该文法生成的一个程序实例在图 5.8。

deref : P α. pointer (α) → α ;
q : *pointer (pointer (integer))* ;
deref (deref (q))

图 5.8 由图 5.7 文法产生的一个程序

再来看引入类型变量、笛卡儿积类型和多态函数后增加的推理规则。

下面的环境规则 (Env Var) 用来把一个类型变量加到静态定型环境中,这时,我们的环境不再仅仅是程序变量到类型的映射了。

(Env Var) $\Gamma ?\quad ,\ \alpha \notin dom(\Gamma)\ ]\ \Gamma, \alpha ?$

我们增加5条语法规则。规则 (Type Var) 用来从环境中得到一个类型变量,规则 (Type Product) 说明积类型的语法,规则 (Type Parenthesis) 表示加括号后的类型表达式仍然是类型表达式。(Type Forall) 和 (Type Fresh) 是为多态类型准备的规则,前者表示加全称量词以形成多态类型,后者表示去掉全称量词,用一个新的类型变量代替约束变量。记号 $[\alpha_i/\alpha]\ T$ 表示 *T 中自由出现的 α 用 α_i* 代换后的结果。

(Type Var) $\Gamma_1, \alpha, \Gamma_2 ?\ ]\ \Gamma_1, \alpha, \Gamma_2 ?\ \alpha$

(Type Product) $\Gamma ?\ T_1, \Gamma ?\ T_2\ ]\ \Gamma ?\ T_1 \times T_2$

(Type Parenthesis) $\Gamma ?\ T\ ]\ \Gamma ?\ (T)$

(Type Forall) $\Gamma, \alpha ?\ T\ ]\ \Gamma ?\ \forall\ \alpha \cdot T$

(Type Fresh) $\Gamma ?\ \forall\ \alpha .. T, \Gamma, \alpha_i ?\ ]\ \Gamma, \alpha_i ?\ [\alpha_i/\alpha]\ T$

定型规则

(Exp Pair) $\Gamma ?\ E_1 : T_1, \Gamma ?\ E_2 : T_2\ ]\ \Gamma ?\ E_1, E_2 : T_1 \times T_2$

比较直观,两个表达式 E_1 和 E_2 的类型分别是 T_1 和 T_2 的话,那么 E_1, E_2 的类型是 $T_1 \times T_2$。它用于多元函数的情况。

多态函数调用的定型规则非常复杂,为避免复杂起见,我们让一个类型代换 (类型表达式→类型表达式的函数) S直接出现在类型表达式中,S(T) 表示代换结果的类型表达式。

(Exp FunCall) $\Gamma ?\ E_1 : T_1 \to T_2,$

$\Gamma ?\ E_2 : T_3\ ]$

$\Gamma ?\ E_1(E_2) : S(T_2)$ (S是 T_1 和 T_3 的最一般的合一代换)

在上面这条定型规则中,对S的限制是,它是 T_1 和 T_3 的最一般的合一代换。下面先解释代换和合一等概念,然后再给出多态函数检查的翻译方案。

5.4.4 代换、实例和合一

类型表达式中的类型变量用其所代表的类型表达式替换,称之为代换。下面的递归函数 subst(t) 阐明了应用代换 S (类型变量到类型表达式的映射) 去替换表达式 t 中所有类型变量的概念,我们用函数类型构造器作为"典型"的构造器。

```
function subst (t : type_expression) : type_expression;
begin
```

if *t* 是基本类型 then return *t*
else if *t* 是类型变量 then return *S* (*t*)
else if *t* 是 $t_1 \rightarrow t_2$ then return *subst* (t_1) → *subst* (t_2)
end

为方便起见,仍用 S(t) 表示 subst 用于 t 后所得的类型表达式 (S 拓广为类型表达式到类型表达式的映射),这个结果 S(t) 叫做 t 的实例。如果代换 S 对变量 α 没有指定表达式,我们假定 S(α) 仍是 α,即 S 在这样的类型变量上面是恒等映射。

例 5.4 下面是用 *s* < *t* 表示 *s* 是 *t* 的实例。

pointer (*integer*) < *pointer* (α)
pointer (*real*) < *pointer* (α)
integer → *integer* < α → α
pointer (α) < β
α < β

但是,下面左边的类型表达式不是右边的实例,原因列在旁边:

integer	*real*	(代换不能用于基本类型)
integer → *real*	α → α	(α 的代换不一致)
integer → α	α → α	(α 的所有出现都应该代换)

如果存在某个代换 S 使得 S(t_1) = S(t_2),那么这两个表达式 t_1 和 t_2 能够合一 (unify)。实际上,我们感兴趣的是最一般的合一代换 (the most general unifier),它是对表达式中的变量限制最少的代换。更精确地说,表达式 t_1 和 t_2 最一般的合一代换是具有下列性质的代换:

(1) S(t_1) = S(t_2);

(2) 对任何其他满足 S′(t_1) = S′(t_2) 的代换 S′,代换 S′(t_1) 是 S(t_1) 的实例。

下面我们所说的合一都是指最一般的合一代换。

5.4.5 多态函数的类型检查

多态函数的类型检查在三个方面和 5.3 节的普通函数的类型检查有区别。在说明多态函数的类型检查前,用图 5.8 程序的表达式 deref (deref (q)) 来说明这些不同。这个表达式的语法树在图 5.9 给出。每个结点有两个标记:第一个指出结点代表的子表达式,第二个是指派给该表达式的类型表达式。下标 i 和 o 分别用来表示 deref 的里外两次不同的出现。

多态函数的类型检查和普通函数的类型检查的区别是:

(1) 表达式中同一多态函数的不同出现无须变元有相同类型。在表达式 deref$_o$(deref$_i$

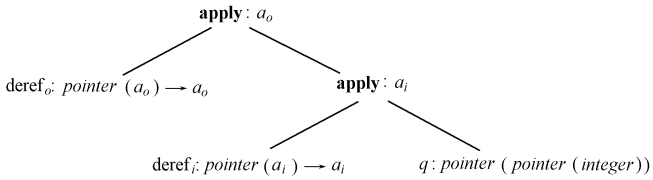


图 5.9 deref(deref(q)) 的带标记的语法树

(q)) 中,deref$_i$ 拿掉一层指针间接,所以 deref$_o$ 和 deref$_i$ 作用于不同类型的变元。这个性质的实现是基于 P α 的解释 (对任何类型 α)。deref 的每次出现对约束变量 α 代表什么类型有它自己的观点。所以对 deref 的每次出现指派一个类型表达式,这个类型表达式是用新的变量代替 deref 的类型表达式中的 α 并且移开全称量词 P 后形成的。在图 5.9 中,新的变量 α_o 和 α_i 分别用于里外两个 deref 的类型表达式中。

(2) 由于变量可以出现在类型表达式中,在某些场合,必须把类型相同的概念推广到类型合一。假如类型为 $s \rightarrow s'$ 的 E_1 作用于类型为 t 的 E_2,我们不是简单地确定 s 和 t 是否相同,而是要看它们能否合一。例如,图 5.9 里面那个标有 apply 的结点,如果 α_i 用 pointer (integer) 代替的话,那么有

$$pointer(\alpha_i) = pointer(pointer(integer))$$

即它们相同。

(3) 要有办法记录两个子表达式合一的结果。通常,类型变量可以出现在几个类型表达式中。如果 s 和 s' 的合一使得变量 α 代表类型 t,那么在类型检查的过程中 α 必须继续代表 t。例如,图 5.9 中,α_i 是 deref$_i$ 的值域类型,所以可用它代表 deref$_i$ (q) 的类型。deref$_i$ 定义域类型和 q 的类型的合一会影响里面那个 apply 结点的类型表达式。图 5.9 的另一个类型变量 α_o 代表 integer。

现在可以考虑多态函数的类型检查算法了。由图 5.7 文法产生的表达式的类型检查根据下面的操作定义。

(1) fresh (t)

它把类型表达式 t 中的约束变量用新的变量代替,返回指向替换后的类型表达式根结点的指针。在此过程中,t 中任何 P 符号都被删除。

(2) unify (m, n)

它合一由 m 和 n 指向的结点所代表的类型表达式。它有副作用:记住使类型表达式相同的代换。如果类型表达式 m 和 n 不能合一,整个类型检查过程失败。本书不打算介绍合一算法。

和 5.3 节的翻译方案不同的是,这里的类型表达式都用语法树来表示。类型表达式图

中的叶结点和内部结点可用类似于4.2节的 *mkleaf* 和 *mknode* 的操作来构造。

表达式类型检查的翻译方案在图5.10给出，我们略去了声明语句和其他无关语句的翻译方案。当由非终结符 T 和 Q 产生的类型表达式被扫描时，它产生语法树形式的类型表达式。在图5.10中，fresh 操作用新变量代替约束变量，并拿掉P 符号。和产生式 $E \rightarrow E_1(E_2)$ 有关的语义动作是置 E.type 为 E_1 和 E_2 类型的积。

函数作用 $E \rightarrow E_1(E_2)$ 的语义动作受到 E_1.*type* 和 E_2.*type* 都是类型变量情况的启发。例如，若 E_1.*type* = α 并且 E_2.*type* = β，那么 α 必须是函数，使得有某个未知类型 γ 满足 $\alpha = \beta \rightarrow \gamma$。在图5.10中，对应 γ 的新类型变量建立，并且 E_1.*type* 和 E_2.*type* → γ 合一。新的类型变量由 newtypevar 调用返回，为它安排的叶结点由 mkleaf 构造。要与 E_1.*type* 合一的代表该函数的结点由 mknode 构造。合一后，新的叶结点代表结果类型。

$E \rightarrow E_1(E_2)$	{p ≒ *mkleaf* (*newtypevar*) ; *unify* (E_1.*type*, *mknode* ('→', E_2.*type*, p)) ; *E*.*type* ≒ p}
$E \rightarrow E_1, E_2$	{E.type ≒ *mknode* ('×', E_1.*type*, E_2.*type*)}
$E \rightarrow$ id	{E.type ≒ *fresh* (*lookup* (id.entry))}

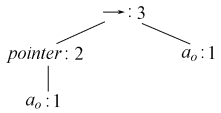
图5.10 检查多态函数的翻译方案

我们用一个简单的例子来详细说明图5.10的语义动作。表5.1中，给出了指派到每个子表达式的类型表达式，总结了这个算法的执行结果。在每一次函数作用时，unify 操作都有副作用，它记录类型变量的类型表达式。这样的副作用由表5.1的代换栏表示。

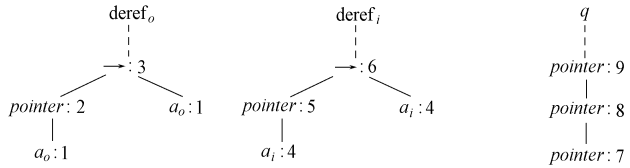
表5.1 自下而上确定类型的小结

表达式 类型	代换
q : *pointer* (*pointer* (*integer*))	
$deref_i$: pointer (α_i) → α	
$deref_i$ (q) : pointer (integer)	α = *pointer* (*integer*)
$deref_o$: pointer (α_o) → α_o	
$deref_o$ ($deref_i$ (q)) : integer	α_o = *integer*

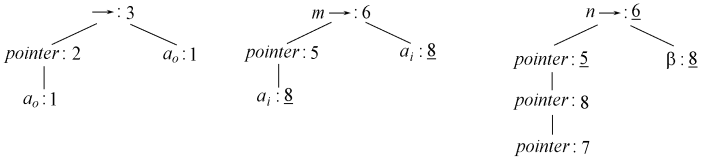
例5.5 图5.8程序的表达式 $deref_o$ ($deref_i$ (q)) 的类型检查从叶结点开始，自下而上处理。下标 o 和 i 仍然用来区别 deref 的不同出现。当考虑子表达 $deref_o$ 时，fresh 用新的类型变量 α_o 构造下列结点：



编号相同的结点表示它们经合一代换后的结果一样,即它们属于同一等价类。类型图中有关三个标识符的部分在下面给出。虚线表示编号为3,6和9的结点分别代表$deref_o$,$deref_i$和q。



函数作用$deref_i$ (q) 由构造从类型q到新类型变量β的函数结点n来检查。这个函数成功地和下面的结点m代表的$deref_i$合一。在结点m和n合一之前,每个结点有一个不同的编号。合一以后,等价的结点是下面有同样编号的结点,变化了的编号加了底线。



注意,α_i和pointer (integer) 的结点都标为8,即αi和这个类型表达式合一,如表5.1所示。再进行下去,α_o和integer合一。

下面一个例子把ML的多态函数的类型推断同5.3.3节和图5.10的类型检查联系起来。ML的函数定义的语法由

fun id_0 (id_1, . . ., id_k) = E;

给出。其中id_0代表函数名,id_1, . . ., id_k代表它的参数。为简单起见,假定表达式E的语法如图5.7所示。这个方法是例5.3方法的形式化,在例5.3中的类型推断用来决定deref的多态类型。现在,先为函数名和它的变元构造新的类型变量。内部函数通常有多态的类型,出现在这些类型中的任何类型变量都由全称量词∀约束。然后检查表达式id_0 (id_1, . . ., id_k) 的类型和E的类型是否匹配。匹配成功时,已得出这个函数的类型。最后,将所得类型的任何变量都用∀量词约束,就得到了这个函数的多态类型。

例5.5 再次写出图5.5的确定表长度的ML函数:

fun length (lptr) =

```
if null (lptr) then 0
else length (tl (lptr)) + 1;
```

为 length 和 lptr 分别引入类型变量 β 和 γ 来表示它们的类型。在形式化的证明中，可以发现 length (lptr) 的类型和函数体的类型能够匹配，并且得到 length 的类型是

$$\forall\ \alpha.\ list(\alpha) \rightarrow integer$$

更详细一些，建立图 5.11 的程序，以便把 5.3.3 节和图 5.10 的翻译方案用于它（或者说，把前面所给出的推理规则用于它）。该程序的声明把新类型变量 β 和 γ 同 length 和 lptr 联系起来，并把内部运算的类型显式化。按图 5.7 的风格，把多态函数 if 作用于三个运算对象，这三部分分别代表被测试的条件，then 部分和 else 部分的类型。这个声明指出 then 和 else 部分可以和任何类型匹配，它也是结果的类型。

```
length : β ;
lptr : γ ;
if : ∀ α. boolean × α × α → α ;
null : ∀ α. list (α) → boolean ;
tl : ∀ α. list (α) → list (α) ;
0 : integer ;
1 : integer ;
+ : integer × integer → integer ;
match : ∀ α. α × α → α ;
match (
    length (lptr) ,
    if (null (lptr) , 0, length (tl (lptr)) + 1)
)
```

图 5.11 类型声明及要检查的表达式

显然，length (lptr) 必须和函数体有同样的类型，这个检查用运算 match 表示。match 的使用是一种技术上的方便，它使得所有的检查都可以用图 5.8 风格的程序完成。

把图 5.10 的类型检查用于图 5.11 的结果见表 5.2。这里只给出了自下而上分析表达式 match (...) 的情况，略去了类型检查用于声明语句的情况。该表的第 3 列是所用的合一代换，第 4 列是得出该行定型断言所使用的规则。(Type Fresh) (或操作 *fresh*) 引入的用于多态的内部定义算符的新变量由 α 的下标区别。

从第 (3) 行可以知道，length 必须是从 γ *到某个未知类型* δ *的函数。然后，检查子表达式* null (lptr) 时，在第 (6) 行发现 γ 和 list (α_n) 合一，其中 α_n 是未知类型。在该点，length 的类型必须是

$$\forall\ \alpha_n.\ list\ (\alpha_n) \rightarrow \delta$$

最后,在第(15)行检查加时,δ和integer合一,为清楚起见,把"+"写在两个变元之间。当检查完毕时,类型变量α_n仍留在length的类型中。因为对类型α_n没有任何限制,所以该函数使用时任何类型可以代换它。让它成为约束变量,并写成

$$\forall\ \alpha_n.\ list\ (\alpha_n) \rightarrow integer$$

这就是length的类型。

表5.2 length的类型推导过程

行	定型断言	代换	规则
(1)	lptr : γ		(Exp Id)
(2)	length : β		(Exp Id)
(3)	length (lptr) : δ	$\beta=\gamma\rightarrow\delta$	(Exp FunCall)
(4)	lptr : γ		从(1)可得
(5)	null : $list\ (\alpha) \rightarrow boolean$		(Exp Id)和(Type Fresh)
(6)	null (lptr) : $boolean$	$\gamma = list\ (\alpha)$	(Exp FunCall)
(7)	0 : $integer$		(Exp Num)
(8)	lptr : $list\ (\alpha)$		从(1)可得
(9)	tl : $list\ (\alpha) \rightarrow list\ (\alpha)$		(Exp Id)和(Type Fresh)
(10)	tl (lptr) : $list\ (\alpha)$	$\alpha = \alpha_n$	(Exp FunCall)
(11)	length : $list\ (\alpha_n) \rightarrow \delta$		从(2)可得
(12)	length (tl (lptr)) : δ		(Exp FunCall)
(13)	1 : $integer$		(Exp Num)
(14)	+ : $integer \times integer \rightarrow integer$		(Exp Id)
(15)	length (tl (lptr)) + 1 : $integer$	$\delta = integer$	(Exp FunCall)
(16)	if : $boolean \times \alpha_i \times \alpha_i \rightarrow \alpha_i$		(Exp Id)和(Type Fresh)
(17)	if (...) : $integer$	$\alpha = integer$	(Exp FunCall)
(18)	match : $\alpha_m \times \alpha_m \rightarrow \alpha_m$		(Exp Id)和(Type Fresh)
(19)	match (...) : $integer$	$\alpha_m = integer$	(Exp FunCall)

5.5 类型表达式的等价

5.2节和5.3节中许多类型检查都有"如果两个类型表达式相同,那么返回某个类型,否则返回type_error"。*到目前为止,类型表达式相同的概念是清楚的,它就是*5.5.1节定义的类型结构等价的概念。

在有些程序设计语言中,类型表达式可以命名,并且这些名字可用于随后的类型表达式中。类型名字的使用会出现一些潜在的二义性,关键问题是类型表达式中的名字是代表它自己还是代表另一个类型表达式的缩写,由此引出区别于结构等价的名字等价概念。

本节讨论这些等价,这些讨论依据类型表达式的图形表示。在这种图形表示中,叶结点表示基本类型,内部结点表示类型构造器。如果名字看成类型表达式的缩写,递归定义的类型将会导致类型图中有环。

注意,类型名字的引入只是类型表达式的一个语法约定问题,它并不像引入类型构造器或类型变量那样能丰富我们所能表达的类型。

5.5.1 类型表达式的结构等价

如果类型表达式仅由类型构造器作用于基本类型组成,两个类型表达式等价的自然想法是结构等价,即两个表达式要么是同样的基本类型,要么是同样的类型构造器作用于结构等价的类型。也就是,两个类型表达式结构等价,当且仅当它们完全相同。例如,类型表达式 integer 仅等价于 integer,因为它们是同样的基本类型。类似地,pointer(integer) 仅等价于 *pointer(integer),因为它们是把同样的构造器 pointer 作用于等价的类型。*

在实际使用中,结构等价的概念常常需要修改以反映源语言的实际类型检查规则。例如,当数组作为参数传递时,我们可能不希望数组的界作为类型的一部分。

图 5.12 是测试结构等价的算法,假定仅有的类型构造器是数组、积、指针和函数。这个算法递归地比较类型表达式的结构而不检查环,所以它能用于类型表达式的树形表示。

```
(1)   function sequiv (s, t : boolean ;
(2)   begin
(3)       if s和t是相同的基本类型 then
(4)           return true
(5)       else if s = array(s1, s2) and t = array(t1, t2) then
(6)           return sequiv(s1, t1) and sequiv(s2, t2)
(7)       else if s = s1 × s2 and t = t1 × t2 then
(8)           return sequiv(s1, t1) and sequiv(s2, t2)
(9)       else if s = pointer(s1) and t = pointer(t1) then
(10)          return sequiv(s1, t1)
(11)      else if s = s1 → s2 and t = t1 → t2 then
(12)          return sequiv(s1, t1) and sequiv(s2, t2)
(13)      else return false
(14)  end
```

图 5.12 两个类型表达式 s 和 t 的结构等价测试

在

s = *array*(s_1, s_2)

t = *array*(t_1, t_2)

中,如果图 5.12 第(6)和第(8)行的数组等价测试重写为

```
else if s = array(s1, s2) and t = array(t1, t2) then
    return sequiv(s2, t2)
```

那么数组的界被忽略。

5.5.2 类型表达式的名字等价

在某些语言中,类型可以命名。例如,在 Pascal 的程序片段

```
type link = ↑cell;
var next :link;
    last :link;
    p :↑cell;
    q, r :↑cell;
```

中,标识符 link 声明为类型↑cell 的一个名字。现在问题出现了,变量 next、last、p、q 和 r 都有相同的类型吗?意想不到的是,答案依赖于实现。这是由于当初 Pascal 的报告没有明确定义术语"相同的类型"。

为了模仿这种情况,我们允许类型表达式命名,并且允许这些名字出现在类型表达式中原先只有基本类型出现的地方。例如,如果 cell 是类型表达式的名字,那么 *pointer*(cell)是类型表达式。目前,假定没有带环的类型表达式定义。

当名字允许出现在类型表达式中时,类型等价的两种不同概念出现了,它们取决于如何看待名字。在结构等价中,先把所有的类型名字由它们定义的类型表达式代换,完成代换后的两个类型表达式结构等价的话,那么原来的类型表达式结构等价。名字等价把每个类型名看成是一个可区别的类型,所以两个类型表达式名字等价当且仅当这两个类型表达式不做名字代换就结构等价。

例 5.6 下表给出了和声明(5.1)的各变量联系的类型表达式。

变量	类型表达式
next	link
last	link
p	*pointer*(cell)
q	*pointer*(cell)
r	*pointer*(cell)

在名字等价下,变量next和last有相同的类型,因为它们有同样的类型表达式。变量p,q和r也有同样的类型,但是p和next类型不相同,因为它们的类型表达式不同。在结构等价下,所有这5个变量都有同样的类型,因为link是类型表达式 *pointer*(cell) 的名字。

在不同语言中,标识符和类型通过声明联系的规则是不同的,在解释这些规则时,结构等价和名字等价是两个有用的概念。

例5.7 Pascal的许多实现用隐含的类型名和每个声明的标识符联系起来,每当变量声明中出现没有名字的类型表达式,那么就为它建立一个隐含的新类型名。于是,(5.1)的类型声明为p,q和r的类型表达式建立两个隐含的类型名。也就是,把这个声明看成是

```
type link = ↑cell;
     np = ↑cell;
     nqr = ↑cell;
var next : link;
    last : link;
    p : np;
    q : nqr;
    r : nqr;
```

这里引入了新的类型名np和nqr。在名字等价下,因为next和last用同样的类型名声明,因而它们有等价的类型。同样,q和r也看成有等价的类型,因为它们有同样的隐含类型名。但是p,q和next的类型不是等价的,因为它们都有不同的类型名。

5.5.3 记录类型

在介绍递归类型前,我们先介绍记录类型。

记录类型从某种意义上来说是它各域类型的积,记录和积之间的主要区别是记录的域被命名。把记录构造器 *record* 作用于域名和其类型的二元组序列,就形成记录类型表达式,由此我们可以写出与记录有关的定型规则如下:

(Type Record) (l_i 是有区别的)

$$\Gamma \vdash T_1, \ldots, \Gamma \vdash T_n \quad \Gamma \vdash \text{record}(l_1 : T_1, \ldots, l_n : T_n)$$

(Val Record) (l_i 是有区别的)

$$\Gamma \vdash M_1 : T_1, \ldots, \Gamma \vdash M_n : T_n$$
$$\Gamma \vdash \textit{record}(l_1 = M_1, \ldots, l_n = M_n) : \text{record}(l_1 : T_1, \ldots, l_n : T_n)$$

(Val Record Select)

$$\Gamma \vdash M : \text{record}(l_1 : T_1, \ldots, l_n : T_n) \quad \Gamma \vdash M.l_j : T_j \quad (j \in 1..n)$$

根据这些定型规则写一个翻译方案,以完成记录类型和记录域访问的类型检查是很容

易的。

例如,Pascal的程序片段

```
type row = record
             address : integer;
             lexeme : array [1..15] of char
           end;
```

声明类型名row代表类型表达式

```
record (address : integer, lexeme : array [15] of char)
```

5.5.4 类型表示中的环

链表和树这样的基本数据结构经常是递归定义的,例如,链表可能是空或者由一个表元和一个链表指针组成。这样的数据结构通常用记录实现,这种记录含指向同类型的记录的指针。在定义这样的记录类型时,类型名起着重要的作用。

考虑链表的每个表元含整型信息和指向下一个表元的指针的情况。链和表元的类型用Pascal声明如下:

```
type link = ↑cell ;
     cell = record
              info : integer ;
              next : link
            end;
```

类型名link根据cell定义,cell根据link定义,所以它们是递归定义的。

用 *pointer*(cell) 替换link,得到cell的类型表达式如图5.13(a)所示。如果愿意在类型图中引入环,递归定义的类型名可以替换掉,即用图5.13(b)的环的话,可以删除该类型图中 *record* 结点以下出现的cell。

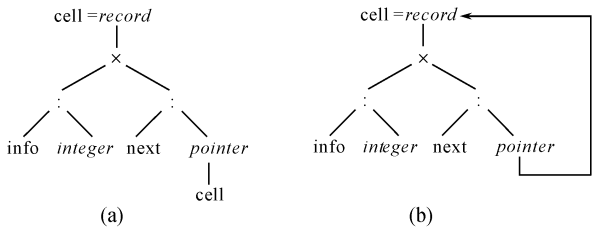


图5.13 递归定义的类型名

例5.8 C语言对除记录(结构)以外的所有类型使用结构等价,而记录类型用的是名字等价,以避免类型图中的环,因为环的出现会使结构等价的判断大大复杂。用C,cell的声明可以是

```
struct cell {
    int info;
    struct cell *next;
}
```

C用关键字struct而不是record,并且名字cell成为该记录类型的一部分。在效果上,C使用的是图5.13(e)的无环表示。

当碰到记录构造器时,结构等价的测试停止,被比较的类型或者由于它们有同样的命名记录类型而等价,或者它们不等价。

5.6 函数和算符的重载

和多态性容易混淆的一个概念是符号(包括函数名)的重载,尤其是有些没有多态性的语言,把类似于本书重载的概念定义为多态,更使得大家难以区分。重载(overload)符号是指该符号有多个含义,但在该符号的每个引用点,其含义可以依赖上下文来确定到惟一。在数学中,加法算符+是重载的,因为当A和B是整数、实数、复数或矩阵时,A+B*中的*+有不同的含义。在Ada语言中,括号()是重载的,表达式A(I)可能是数组A的第I个元素的引用,也可能是用变元I调用函数A。多态的符号只有一个含义,但是它允许参数类型在一定的范围内变化。

在重载符号的引用点,若其含义能确定到惟一就叫做重载的消除。例如,如果+可以表示整数加或实数加,那么在x+(i+j*中*,+的两个出现可以表示不同形式的加,它取决于*x,i和j的类型。重载的消除有时也称做*算符的辨别,因为它确定运算符号指称哪个运算。

在大多数语言中,算术算符是重载的。不过,它们的重载可以由仅看它们变元的类型而消除。它们惟一含义的确定类似于图5.3中$E \rightarrow E_1$ op E_2的语义规则,即E的类型可以由看E_1和E_2的类型来确定。

注意,重载的函数和符号的引入使得程序员可以用一个名字或符号表示多个不同类型的函数或运算,它也不像引入类型构造器或类型变量那样能丰富我们所能表达的类型。

5.6.1 子表达式的可能类型集合

并不总是只看函数的变元就可以消除重载。如下例所示,单看一个子表达式本身,它有

一个可能类型的集合,而不只是一个类型。在 Ada 中,上下文必须提供足够的信息来缩小这个集合到惟一类型。

例5.9 在 Ada 中,算符*的一个标准(即内部定义)解释是一对整数到一个整数的函数。加入下面这样的声明:

function * (*i*, *j*: integer) return complex;
function * (*x*, *y*: complex) return complex;

会使得*重载。在上述声明后,*可能的类型包括:

integer × *integer*→*integer*
integer × *integer*→*complex*
complex × *complex*→*complex*

如果2,3和5可能的类型仅是整型,在上述声明的环境下,子表达式3*5是整型或复型,到底是哪一个则取决于它的上下文。如果完整的表达式是2*(3*5),那么3*5必须是整型,因为*的两个变元要么都是整型,要么都得复型。相反,如果表达式是(3*5)*z并且z是复型,那么3*5必须是复型。

在5.3节,假定每个表达式有惟一的类型,所以函数作用的类型检查是:

$E \rightarrow E_1(E_2)$ {E.type ≔ if E_2. type = s and E_1. type = s→t then t
else type_error}

表5.3把这条规则自然地推广到有类型集合的情况。表5.3仅有的运算是函数作用,表达式中其他算符的类型检查仍类似于前面的检查。重载的标识符可能有几个声明,所以假定符号表的条目包含可能类型的集合,这个集合由 lookup 函数返回。开始非终结符 E′产生完整的表达式,它的作用将在下面澄清。

表5.3 确定表达式可能类型的集合

产生式	语义规则
$E' \rightarrow E$	E′.types ≔ E.types
$E \rightarrow id$	E.types ≔ lookup(id.entry)
$E \rightarrow E_1(E_2)$	E.types ≔ {t	E_2. types中存在一个s,使得 s→t属于 E_1. types}

如果用自然语言叙述的话,表5.3第3行的语义规则可以这么说:如果 s 是 E_2 的一个类型,并且 E_1 的一个类型能把 s 映射到t,那么 t 是 $E_1(E_2)$ 的一个类型。函数作用的类型不匹配会使集合 E.types 为空,可以用它作为通知类型错误的条件。

例5.10 除了解释表5.3的说明外,这个例子还暗示这种方法怎样贯彻到其他结构中。考虑表达式3*5,设算符*的声明如例5.9所示的那样。即根据上下文,*可以把一对整数

或复数映射到整数或复数。子表达式 3*5 的可能类型集合在图5.14 中,其中 i 和 c 分别是 *integer 和 complex 的缩写。*

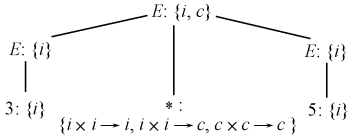


图 5.14　表达式 3*5 可能的类型集合

5.6.2　缩小可能类型的集合

Ada 要求完整的表达式有惟一的类型。根据上下文确定的表达式惟一类型,我们可以缩小每个子表达式的类型选择。如果这个过程不能使每个子表达式的类型确定到惟一,那么宣布该表达式有类型错误。

表5.4　缩小表达式的类型集合

产生式	语义规则
$E' \rightarrow E$	$E'.types \doteq E.types$ $E.unique \doteq$ if $E'.types = \{t\}$ then t else $type\_error$ $E'.code \doteq E.code$
$E \rightarrow id$	$E.types \doteq lookup(id.entry)$ $E.code \doteq gen(id.lexeme : E.unique)$
$E \rightarrow E_1(E_2)$	$E.types \doteq \{s \mid E_2.types$ 中存在一个 s,使得 $s \rightarrow s'$ 属于 $E_1.types\}$ $t \doteq E.unique$ $S \doteq \{s \mid s \in E_2.types$ and $s \rightarrow t \in E_1.types\}$ $E_2.unique \doteq$ if $S = \{s\}$ then s else $type\_error$ $E_1.unique \doteq$ if $S = \{s\}$ then $s \rightarrow t$ else $type\_error$ $E.code \doteq E_1.code \mid\mid E_2.code \mid\mid gen(apply : E.unique)$

表 5.4 的语法制导定义由表 5.3 的定义加上确定 *E 的继承属性 unique 的语义规则得*到。E 的综合属性 code 在下面讨论。

因为整个表达式由 E′ 产生,我们希望 E′.types 是单个类型的集合。这个类型由 E.*unique 继承。基本类型 type_error 仍是通知错误出现。*

如果函数作用 $E_1(E_2)$ 返回类型 t,那么可以找到类型 s, 它对变元 E_2 是可行的,同时,$s \rightarrow t$ *对该函数是可行的。表* 5.4 中语义规则的集合 S 用来检查具有这个性质的惟一类型 s。

表 5.4 的语法制导定义可以通过对表达式语法树的两次深度优先遍历来实现。在第一

次遍历时,属性 types 被自下而上综合。第二次扫描时,属性 unique 被自上而下传播,当从结点返回时,code 属性可以综合。code 属性是生成的中间代码,这里用的是后缀表示。在中间表示中,每个标识符和 apply 算符的实例都有一个类型。

习 题 5

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.
```

5.2 下面的 Pascal 程序在某些环境中运行时,报告访问错,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.
```

5.3 下面是一个 C 语言程序,虽然 main 函数中两次调用函数 gcd 的参数个数都不对,但是该程序能够通过编译和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)) ;
}
```

5.4 为下列类型写类型表达式：

(a) 指向实数的指针数组，数组的下标从1到100。

(b) 两维数组（即数组的数组），它的行下标从1到10，列下标从1到20。

(c) 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。

5.5 假如有下列C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写类型表达式。

5.6 下列文法定义字面常量表的表。符号的解释和图5.2文法的那些相同，增加了类型list，它表示类型T的元素表。

$P \rightarrow D\ ;\ E$

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

$T \rightarrow \text{list of}\ T \mid \text{char} \mid \text{integer}$

$E \rightarrow (L) \mid \text{literal} \mid \text{num} \mid \text{id}$

$L \rightarrow E,\ L \mid E$

写一个类似5.3节中的翻译方案，以确定表达式（E）和表（L）的类型。

5.7 把产生式

$E \rightarrow \text{nil}$

加入习题5.6的文法。含义是表达式可以是空表。修改习题5.6的答案，把nil看成空表，其元素可以是任何类型。

5.8 修改5.3节的翻译方案，发现错误时打印描述信息，并继续检查，好像所期望的类型已经看见。

5.9 修改5.3.3节的翻译方案，使之能处理：

(a) 有值语句。赋值语句的值是赋值号右边的表达式的值，条件语句或当语句的值是语句体的值，语句表的值是表中最后一个语句的值。

(b) 布尔表达式。加上逻辑算符and，or及not和关系算符的产生式。然后给出适当的翻译规则，它

们检查这些表达式的类型。

5.10 推广在5.3节给出的一元函数类型检查,使之能处理 *n* 元函数。

5.11 C语言是一种类型语言,但它不是强类型化语言,因为运行前的类型检查不能保证所接受的程序没有不会被捕获的错误。例如,编译时的类型检查一般不能保证运行时没有数组越界。请你再举一个例子说明C语言不是强类型化语言。

5.12 拓展5.3.3节的类型检查,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→record fields end
fields→fields;field | field
field→id :T
E→E.id
```

5.13 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程序所在的文件名是type.c,用SPARC/Solaris C编译器编译时,错误信息如下:

```
type.c 24: warning: passing argument 4 of qsort from incompatible pointer type
```

请对该程序略作修改,使得该警告错误能消失,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。SPARC/Solaris C比ANSI C的静态检查要严格一些。

```
#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) ;
}
```

5.14 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,返回这个对象的指针。

(b) 函数 *arrayderef*,它以一个数组为变元,数组的下标是整型,数组的元素是某类型的指针,返回一个数组,它的元素是变元数组的元素所指向的对象。

5.15 找出下列表达式的最一般的合一代换：

(a) (pointer (α)) × (β→γ)

(b) β× (γ→δ)

如果(b)的δ是α呢？

5.16 对下面的每对表达式,找出最一般的合一代换：

(a) $\alpha_1 \rightarrow (\alpha_2 \rightarrow \alpha_1)$

(b) array (β_1) → (pointer (β_1) → β_2)

(c) $\gamma_1 \rightarrow \gamma_2$

(d) $\delta_1 \rightarrow (\delta_1 \rightarrow \delta_2)$

5.17 效仿例5.5,推导下面map的多态类型：

map : P α.P β. ((α→β × *list* (α)) → *list* (β

map的ML定义是

```
fun map (f, l) =
    if null (l) then nil
    else cons (f(hd (l)), map (f, tl (l))) ;
```

在这个函数体中,内部定义的标识符的类型是：

```
null : P α. list (α) → boolean ;
nil : P α. list (α) ;
    cons : P α. (α× list (α)) → list (α) ;
    hd : P α. list (α) → α ;
    tl : P α list (α) → list (α) ;
```

5.18 假定类型名link和cell如5.5节那样定义,下面的表达式中,哪些结构等价？哪些名字等价？

(a) link

(b) *pointer* (cell)

(c) *pointer* (link)

(c) *pointer* (*record* ((info : *integer*) × (next : *pointer* (cell)))

5.19 推广表 5.4 的算法,使表达式有类型构造器 *array* 和 *pointer*。

5.20 使用例 5.9 的规则,确定下列哪些表达式有惟一类型(假定 z 是复数):

(a) 1*2*3

(b) 1*(z*2)

(c) (1*z)*z

5.21 在 C 语言的教材上,称 & 为地址运算符,&a 为变量 a 的地址。但是教材上没有说明表达式 &a 的类型是什么。另外,教材上说,数组名代表数组的首地址,但是也没有说明这个值的类型。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 的首地址,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,请根据编译报错信息和程序运行结果,判断出表达式a和 &a 的类型。若能明白它们的类型,那么它们的区别就清楚了,从而可以正确使用它们。

(1) 文件1:

```
typedef int A[10][20];
A a;

A* fun()
{
   return(a);
}
```

该函数在 Linux 上用 gcc 编译时,报告的类型错误如下:

第6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 Linux 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 Linux 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

第 6 章　运行时存储空间的组织和管理

在这一章,把过程和函数这样的程序单元统称为过程,运行时过程的一次执行称为过程的一次活动,过程的每次调用引起它的一个活动。过程的活动需要可执行代码和存放所需信息的存储空间,后者通常用一块连续的存储区来管理,称为活动记录。在考虑代码生成之前,需要把静态的程序正文和运行时的活动联系起来,考察名字和数据对象之间的关系。我们在本章不仅要讨论一个活动记录中的数据安排,还要讨论一个程序执行过程中所有活动记录的组织方式。

由于过程可以递归,在程序运行中某一时刻,可能有多个活跃着的活动与一个过程对应,因此递归过程中的一个变量可以表示目标机器上不同的数据对象,虽然某一时刻只有一个这样的对象是可访问的。这是一个影响存储分配策略的重要语言特征。影响存储分配策略的语言特征有如下一些:

⑴ 过程能否递归;
⑵ 当控制从过程的活动返回时,局部变量的值是否要保留;
⑶ 过程能否访问非局部变量;
⑷ 过程调用的参数传递方式;
⑸ 过程能否作为参数被传递;
⑹ 过程能否作为结果值传递;
⑺ 存储块能否在程序控制下动态地分配;
⑻ 存储块是否必须显式地释放。

语言编译器组织运行时的存储空间和把名字绑定到数据单元的方式,在很大程度上取决于对上述问题的回答,这也是本章的讨论中要重点解决的问题。

6 1　局部存储分配策略

本节讨论一个过程活动所需信息的存储分配,我们先回顾和这个存储分配有关的语言概念,然后介绍活动记录中的数据安排,最后介绍对过程中并列的程序块实行重叠分配的办法。

6.1.1 过程

过程定义是一个声明,它的最简单形式是将一个名字和一个语句联系起来。该名字是过程名,而这个语句是过程体。在大多数语言中,返回值的过程叫做函数,完整的程序也可以看作一个过程。

当过程名出现在调用语句中时,我们就说这个过程在该点被调用。过程调用就是执行被调用过程的过程体。过程调用也可以出现在表达式中,这时也叫做函数调用。

出现在过程定义中的某些名字是特殊的,它们被称为该过程的形式参数(或形参)(C 语言称它们为形式变元,FORTRAN 语言称它们为哑变元)。称为实在参数(或实参)的变元传递给被调用过程,它们取代过程体中的形式参数。建立实参和形参对应的方法在 6.4 节讨论。图 6.1 是一个完整的 Pascal 程序,除了主过程外,它还有 3 个过程嵌在主过程中。

过程体的每次执行叫做该过程的一个活动。过程 p 的一个活动的生存期是从过程体开始执行到执行结束的时间,包括消耗在执行被 p 调用的过程所需的时间,以及再由这样的过程调用过程所花的时间等等。一般而言,术语"生存期"涉及程序执行期间的连续的步序列。

6.1.2 名字的作用域和绑定

语言中的声明是把信息联系到名字的一种语法结构。名字的声明可以是显式的,例如 Pascal 的程序段:

```
var i:integer;
```

声明也可以是隐式的,例如 FORTRAN 程序中,若无其他声明的话,以字母 I 开始的变量名代表整型变量。

```
program sort(input,output);
  var a:array[0..10] of integer;

  procedure readarray;
    var i:integer;
    begin
      for i:=1 to 9 to read(a[i])
    end;

  function partition(y,z:integer):integer;
    var i,j,x,y:integer;
    begin
      ...
    end;

  procedure quicksort(m,n:integer);
    var i:integer;
    begin
      if(n>m) then begin
        i:=partition(m,n);
        quicksort(m,i-1);
        quicksort(i+1,n);
      end
    end;

  begin
    a[0]:=-9999;
    a[10]:=9999;
    readarray;
    quicksort(1,9)
  end.
```

图 6.1 读入整数并排序的 Pascal 程序

在程序的不同部分可能有同一名字的互相独立的声明,语言的作用域规则规定了该名字的哪个声明应用到程序正文中该名字的出现。图6.1的Pascal程序中,i在3个过程中都有声明,但它们的使用互相独立。

一个声明起作用的程序部分称为该声明的作用域。过程中出现的名字,如果是在该过程的一个声明的作用域内,那么这个出现称为局部于该过程的;否则叫做非局部的。局部和非局部的区分也适用于其他任何可包含声明的语法结构。作用域是名字声明的一个性质。为简单起见,用简称"名字x的作用域"来代替"用于名字x的这个出现的x声明的作用域"。

即使一个名字在程序中只声明一次,该名字在程序运行时也可能代表不同的数据对象。非正式的术语"数据对象"指的是保存值的存储单元。

在程序设计语言的语义中,由于上面所讲的原因,通常用环境和状态来表示变量名字到值的映射。术语环境表示将名字映射到存储单元的函数,术语状态表示将存储单元映射到它所保存的值的函数,如图6.2所示。我们也可以说,环境把名字映射到左值,而状态把左值映射到右值。状态和环境是有区别的,赋值改变状态,但不改变环境。

图6.2 从名字到值的两步映射

如果环境将名字x映射到存储单元s,我们就说x被绑定(binding)到s。术语"存储单元"是象征性的,因为如果x不是基本类型的话,那么x的存储单元s可能是一组存储字。

在学习本章时,读者应注意区别一些静态的概念和它们的动态对应物,表6.1中已经列出一些。尤其要注意,在某一时刻,递归过程可以有不止一个活动活跃着,递归过程的局部变量名字在该过程的不同活动中绑定到不同的存储单元。

表6.1 静态概念和动态概念的对应

静态概念	动态对应
过程的定义	过程的活动
名字的声明	名字的绑定
声明的作用域	绑定的生存期

6.1.3 活动记录

过程的一次执行所需要的信息用一块连续的存储区来管理,这块存储区叫做活动记录或帧(frame),它由图6.3的各个域组成。不同的语言,同一语言的不同编译器所使用的域可能是不同的,这些域在活动记录中的排放次序也可能是不同的,另外,寄存器往往可以取代它们中的一个或多个域。

活动记录的各个域的用途如下(从临时数据域开始):

图 6.3 一般的活动记录

⑴ 临时数据域。如计算表达式出现的中间结果,若寄存器不足以存放所有这些中间结果时,可以把它们存放在临时数据域中。

⑵ 局部数据域。保存局部于过程执行的数据,这个域的布局在下面讨论。

⑶ 机器状态域。保存刚好在过程调用前的机器状态信息,包括程序计数器的值和控制从这个过程返回时必须恢复的机器寄存器的值。

⑷ 访问链。Pascal 语言需要用访问链来访问非局部数据,我们在 6.3 节介绍。像 FORTRAN 和 C 这样的语言不需要访问链,因为全局数据保存在固定的地方。访问链也称为静态链。

⑸ 控制链。用来指向调用者的活动记录。控制链也称为动态链。

⑹ 参数域。用于存放调用过程提供的实在参数。为提高效率,实际上常常用寄存器传递参数。

⑺ 返回值域。用于存放被调用过程返回给调用过程的值。为提高效率,这个值也常常用寄存器返回。

每个域的长度都可以在过程调用时确定。事实上,几乎所有域的长度都可以在编译时确定。一个例外是,如果过程中有大小在过程激活时才能确定的局部数组时,那么只有运行到调用这个过程时才能确定局部数据域的大小,我们在 6.2 节讨论活动记录中可变长数据的分配方法。

活动记录其实并没有包含过程一次执行所需的全部信息,比方说非局部数据就不在活动记录中,6.2 节和 6.3 节中有介绍访问非局部数据的方法。另外,过程运行时生成的动态变量也不在活动记录中,对它们通常采用堆式分配。

6.1.4 局部数据的安排

假定运行时存储空间是连续字节区域,其中字节是可编址内存的最小单位。一个字节为 8 位,几个字节形成一个机器字。多字节对象存于连续的字节中,并以第一个字节的地址作为该对象的地址。

变量所需的存储空间可以由它的类型确定。一个基本数据对象,如字符、整数或实数,可以用几个连续字节保存。对于数组,在分配给它的存储区内,数组元素依次存放,以便于计算下标变量的地址。对于记录,在分配给它的存储区内,它的域通常按类型声明时出现的次序存放。

在编译时,一个过程所声明的局部变量,按这些变量声明时出现的次序,在局部数据域

中依次分配空间。这些局部数据的地址可以用相对于某个位置的相对地址来表示,例如相对于活动记录的开始点,或者相对于活动记录中部的某个特定单元。相对地址(或者叫偏移)就是指数据对象和这个位置的地址差,活动记录中其他域的访问也可以用相对于这个位置的相对地址来处理。

数据对象的存储安排还受目标机器寻址限制的影响。例如,整数加的指令可能要求整数对齐(alignment),即放在内存中满足一定条件的某个位置,例如被4整除的地址。例如10个字符的数组只需要10个字节,假如下面紧接着安排一个整数的话,编译器很可能跳过2个字节后再进行分配,以保证该整数的地址是4的倍数。由于考虑对齐而引起的无用空间叫做衬垫空白区。如果空间很宝贵,编译器可能紧凑安排数据,使得没有任何衬垫空白区出现,但是运行时可能要执行一些额外的指令来取出这些紧凑的数据,然后才能对它们进行操作。

6.1.5 程序块

程序块(block,又翻译成分程序)是本身含有局部变量声明的语句,程序块的概念起源于AlgoL语言,C语言的程序块的语法是

{声明 语句}

程序块的一个特点是它的嵌套结构,分界符标记程序块的开始和结束,C语言用括号"{"和"}"作为分界符,而AlgoL语言的传统是用begin和end做分界符。分界符保证程序块不是相互独立就是一个嵌在另一个里面,即不可能出现程序块 B_1 先于 B_2 开始,又先于 B_2 结束的情况。这种嵌套性有时又称作程序块结构。

程序块结构的声明作用域由下面的最接近的嵌套规则给出:

(1) 程序块 *B* 中声明的作用域包括 B_0。

(2) 如果名字 x 没有在 *B* 中声明,那么 *B* 中 x 的出现是在外围程序块 *B'* 的 x 声明的作用域中,且满足

(a) *B'* 有 x 的声明;

(b) *B'* 比其他任何含 x 声明的程序块更接近被嵌套的 B_0。

图6.4的程序中,每个声明给被声明变量置的初值是它所在的程序块的编号。这些声明的作用域见表6.2,B_0 中 b 的声明的作用域是 B_0-B_1,即不包括 B_1,因为 B_1 中也有 b 的声明,这样的间隙在声明的作用域中称为洞。

最接近的嵌套作用域规则反映在图6.4程序的输出中。控制流从正文中上邻程序块的点进入程序块,离开程序块时到达正文中它的下邻点。打印语句按照 B_2,B_3,B_1 和 B_0 的次序执行,即控制离开这些程序块的次序。在这些程序块中,a 和 b 的值分别为:

2 1
0 3

0 1
0 0

```
main(){
 /* begin of B0 */
        int a = 0;
        int b = 0;
        /* begin of B1 */
                int b = 1;
                /* begin of B2 */
                        int a = 2;
                        printf("%d %d\n",a,b);
                /* end of B2 */
                /* begin of B3 */
                        int b = 3;
                        printf("%d %d\n",a,b);
                /* end of B3 */
                printf("%d %d\n",a,b);
        /* end of B1 */
        printf("%d %d\n",a,b);
/* end of B0 */
```

图 6.4 C 程序的程序块

表 6.2 图 6.4 中各声明的作用域

声 明	作 用 域
int a = 0;	$B_0 - B_2$
int b = 0;	$B_0 - B_1$
int b = 1;	$B_1 - B_3$
int a = 2;	B_2
int b = 3;	B_3

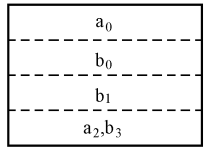


图 6.5 图 6.4 的程序所声明的变量的存储单元

如果在过程中有程序块,那么编译器在存储分配时要为程序块的变量声明留出所需的存储空间。对于图 6.4 的程序,可以按图 6.5 所示来分配存储空间,局部变量 a 和 b 的下标用来标识它们的声明所在的程序块。注意,a_2 和 b_3 可以重叠分配存储单元,因为它们所在的程序块不会同时活跃。

有一点需要注意,在确定所需存储空间时,假设程序运行时会走遍所有的控制路径,即为条件语句的then和else部分都会执行,循环语句的循环体也会执行。这样,虽然过程中有程序块,仍然能静态地确定活动记录中局部数据域的大小。

6.2 全局存储分配策略

上节介绍了单个活动记录,本节介绍程序运行时所需的各个活动记录在存储空间的分配策略。我们介绍3种分配策略,并描述过程的目标代码怎样访问绑定到局部名字的存储单元。这3种策略是:

(1) 静态分配策略 在编译时安排所有数据对象的存储单元。

(2) 栈式分配策略 按栈方式管理运行时的活动记录。

(3) 堆式分配策略 在运行时根据要求从堆数据区域分配存储空间和释放存储空间。

本节所讨论的存储分配策略可用于像FORTRAN,Pascal和C这样的语言。

6.2.1 运行时内存的划分

假定编译器从操作系统得到一块存储区,用于被编译程序的运行。运行时该存储区域可以划分成若干块,用以保存:

(1) 生成的目标代码;

(2) 数据对象和运行时所需的其他信息。

产生的目标代码长度在编译时即可确定,并且通常在运行时不会改变。一个过程有多个活动活跃时,这些活动共享代码段,虽然它们有不同的活动记录。这样,编译器可以把目标代码放在静态确定的区域中,可能是内存的低地址区。

一些数据对象的长度在编译时是可以知道的,并且它们的生存期是整个程序的运行时间,那么它们可以放在静态确定的数据区域中,如图6.6所示。尽可能对数据对象进行静态分配的一个理由是,这些对象的地址可以编译到目标代码中去,以提高运行时对这些数据对象的访问速度。例如,FORTRAN语言的所有数据(乃至所有的活动记录)都可以静态分配,因为FORTRAN语言不允许过程的递归调用,因此运行时每个过程只有一个活动记录。C语言程序的外部变量和程序中出现的常量都可以静态分配,虽然后者的生存期不一定是整个程序的运行时间。

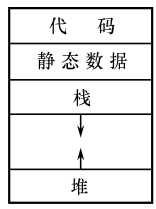


图6.6 运行时内存空间的划分

Pascal和C这样的语言,由于过程递归的次数一般不是静态可确定的,因此活动记录不

可能静态分配。另一方面,由于一个过程活动终止时其活动记录不再需要,又由于过程活动的生存期要么嵌套,要么无重叠,因此可以把当前活跃着的过程活动的活动记录组织成一个栈,这就是图 6.6 中所指的栈。这种存储分配策略将在 6.2.3 节讨论。

运行时内存的另一个单独区域叫做堆。Pascal 和 C 语言都允许数据在程序控制下分配和释放,这些数据可能比生成它们的过程活得更长,因此不能按照栈式分配的原则来安排它们,通常把它们分配在堆区。数据放在栈上比放在堆上的开销要小一些,这是由它们对数据的分配和释放方式决定的。

程序执行时,栈的长度和堆的长度都会改变,所以在图 6.6 中把它们分放在空闲区的两端,需要时向对方增长。

在第 10 章,我们还会对运行时内存空间的组织做更细一点的介绍。

6.2.2 静态分配

在静态分配中,名字在程序被编译时绑定到存储单元,不需要运行时的任何支持。因为运行时不会改变绑定,即这种绑定的生存期是程序的整个运行时间,因此一个过程每次被激活时,它的名字都绑定到同样的存储单元。这种性质允许变量的值在过程停止后仍然保持,因而当控制再次进入该过程时,局部变量的值和控制上一次离开时的一样。

对于静态分配来说,每个活动记录的大小是固定的,并且通常用相对于活动记录一端的偏移来表示数据的相对地址。编译器最后必须确定活动记录区域在目标程序中的位置,如相对于目标代码的位置。一旦这一点确定下来,每个活动记录的位置以及活动记录中每个名字的存储位置也就都固定了,所以编译时在目标代码中能填上所要操作的数据对象的地址。同时,过程调用时保存信息的地址在编译时也是已知的。

静态分配给语言带来一些限制:

(1) 递归过程不被允许,因为一个过程的所有活动使用同一个活动记录,也就是使用同样的局部名字的绑定。

(2) 数据对象的长度和它在内存中位置的限制,必须是在编译时可以知道的。

(3) 数据结构不能动态建立,因为没有运行时的存储分配机制。

FORTRAN 语言被设计成允许静态存储分配。FORTRAN 程序由主程序、子程序和函数组成,图 6.7 的 FORTRAN77 程序是一个示例。按图 6.6 的内存组织方式,该程序的代码和活动记录的安排见图 6.8。在 consume 的活动记录中,有局部变量 buffer,next 和 c 的空间。produce 中也有 next 声明,但这不会引起问题,因为它们分别局部于这两个过程,绑定在各自的活动记录中。

```
(1)     program consume
(2)         character*50 buffer
(3)         integer next
(4)         character c,produce
(5)         data next/1/,buffer/      /
(6)  6          c = produce()
(7)             buffer(next:next) = c
(8)             next = next + 1
(9)             if(c.ne.      ) goto 6
(10)            write(*,    (A)  ) buffer
(11)            end

(12)    character function produce()
(13)        character*80 buffer
(14)        integer next
(15)        save buffer,next
(16)        data next/81/
(17)            if(next.gt.80) then
(18)                read(*,    (A)  ) buffer
(19)                next = 1
(20)            end if
(21)            produce = buffer(next:next)
(22)            next = next + 1
(23)            end
```

图 6.7 一个 FORTRAN77 程序

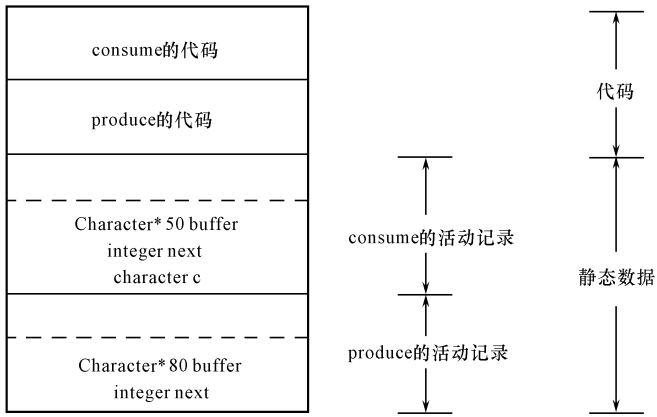


图 6.8 一个 FORTRAN77 程序局部变量的静态分配

例 6.1 在图 6.7 的程序中,过程 produce 的每次活动递交一个字符,主程序 consume 将每次调用 produce 得到的一个字符存入缓冲区,直至遇到空格时,将已读入的字符输出。

显然,该程序的结果取决于 produce 过程每次活动开始时局部变量的值。FORTRAN77 的 save 语句要求,过程的一个活动开始时,其局部变量的值必须与该过程上一次活动结束时的值一样。显然,本节的静态存储分配策略自动满足了 save 语句的要求。这些局部变量的初值用 data 语句指定。在静态存储分配策略下,置初值是在装入目标程序时完成,即程序开始运行前,data 语句指定的初值已被置入对应变量的存储单元。

该程序的输入是

FORTRAN Pascal C

的话,则输出是

FORTRAN

6.2.3 栈式分配

对于 FORTRAN、Pascal 和 C 这样的语言,我们知道,如果 *a 和 b* 是过程的活动,那么它们的生存期或者嵌套,或者无重叠。也就是说,如果在离开 *a 之前进入 b*,那么控制在离开 *b* 之后才能离开 a。这样,我们可以用树来描绘控制进入和离开活动的方式,这样的树称为活动树。在活动树中:

⑴ 每个结点代表某过程的一个活动;

⑵ 根结点代表主程序的活动;

⑶ 结点 *a 是结点 b 的父结点,当且仅当控制流从 a 的活动进入 b 的活动;*

⑷ 结点 a 处于结点 b 的左边,当且仅当 a 的生存期先于 b 的生存期。

因为结点和活动一一对应,所以当控制处于某结点代表的活动中时,我们就直接说控制在这个结点。

例 6.2 图 6.1 程序的活动树画在图 6.9 中,其中过程名都用它的第一个字母表示,括号中的数据是活动的实参。程序的控制流对应从活动树根开始的深度优先的遍历。

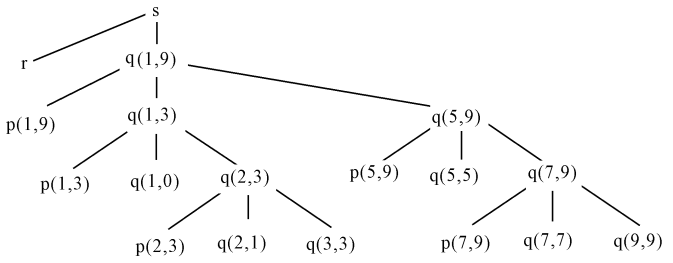


图 6.9 图 6.1 程序的活动树

例6.3 图6.10给出了控制进入q(2,3)代表的活动时,已经执行完毕和正在执行的活动。其中已经执行完毕的活动在虚线的下端,正在执行、尚未结束的活动在根结点到结点q(2,3)的这条实线路径上。尚未开始的活动在图中没有画出。

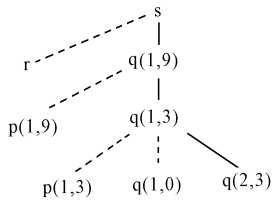


图6.10 控制栈包含实线上的结点

从图6.10可以看出,当前活跃着的过程活动可以保存在一个栈中。当活动开始时,把这个活动的结点压入栈中,当它结束时,把它的结点从栈中弹出。我们称这样的栈为控制栈。对于图6.10的情况,控制栈的内容从栈底到栈顶依次为:

s,q(1,9),q(1,3),q(2,3)

如果把控制栈中的信息拓广到包括过程活动所需的所有局部信息(即活动记录),控制栈就变成了活动记录栈,通常称为运行栈。当一个过程被调用时,它的一个新的活动记录被压入栈,局部变量被绑定到它的存储单元;当对应这次调用的活动终止时,该活动记录不再需要,被弹出栈,即局部变量的存储单元被释放,局部变量的值丢失。由于一个过程的每次调用都会引起一个新的活动记录进栈,所以过程每次活动时局部变量都被绑定到新的存储单元。

首先考虑所有活动记录的长度在编译时都是可知的栈式分配情况,而后再考虑编译时只能获得不完整长度信息的情况。

例6.4 图6.11表示当控制流通过如图6.9所示的活动树时,活动记录压入运行栈和从运行栈中弹出的情况,树上的虚线仍然引向已经结束的活动。程序执行开始时有过程s的活动,当控制到达s体中第一个调用时,激发过程r的一个活动,它的活动记录分配在栈顶。当控制从这个活动返回时,该活动记录从栈被释放,栈中仅剩下s的活动记录。在s的活动中,当控制到达以1和9为实参的过程q的调用时,q的这个活动的活动记录分配在栈顶。只要控制还在这个活动中,它的活动记录就在栈顶。

图6.11的最后两个瞬像之间有好几个活动出现。在最后一个瞬像中,活动p(1,3)和q(1,0)在q(1,3)的生存期里都已经开始和终止了,所以它们的活动记录也都已经入栈和退栈,留下q(1,3)的活动记录在栈顶。

对于活动记录的大小在编译时能确定的情况,我们能够确定局部数据在活动记录中的相对位置。假定运行时*top*寄存器指示运行栈的栈顶,那么过程的目标代码中局部名字x的地址可以写成$-dx(top)$,即栈顶活动记录中绑定到x的数据可以在$top-dx$的位置找到。当然,也可以用指向活动记录中某个固定点的寄存器,把它的值加(减)某个偏移来计算地址。

从前面的介绍可以知道,过程调用和过程返回都需要执行一些代码来管理活动记录栈,保存或恢复机器状态等。我们把在过程调用时执行的分配活动记录,把信息填入它的域中

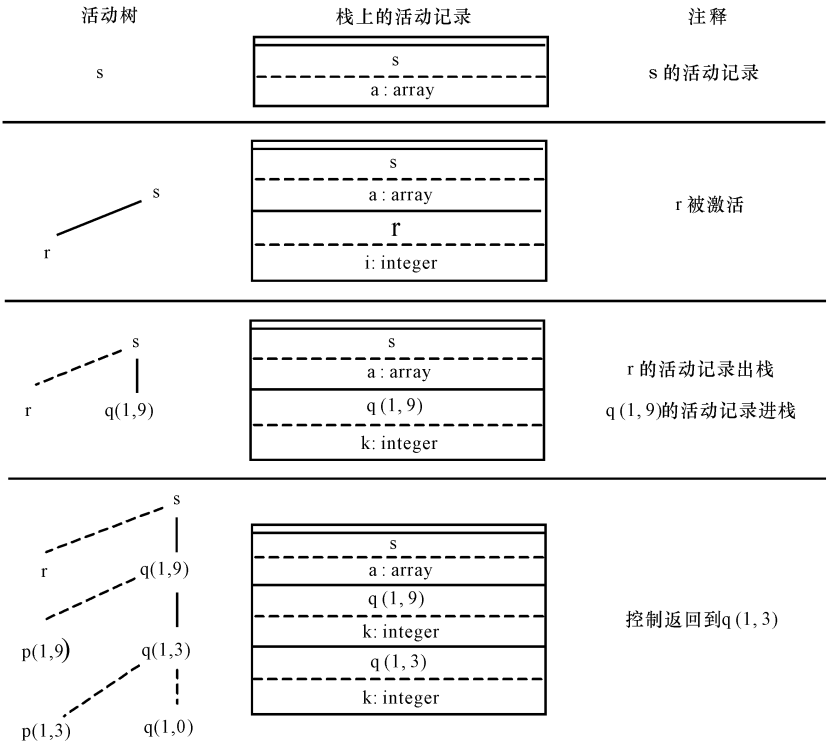


图 6.11 活动记录在栈中的分配 (栈向下长)

的代码称为过程调用序列 而把在过程返回时执行的恢复机器状态,释放活动记录,使调用过程能够继续执行的代码称为过程返回序列。

即使是同一种语言,过程调用序列、返回序列和活动记录中各域的排放次序,也会因实现而异。过程调用序列的代码常常分成两部分,分处于调用过程和被调用过程中。过程调用序列在这两个过程间的划分也不是惟一的。源语言、目标机器和操作系统强加的约束可能使得某种方法比另一种方法更合适。过程返回序列也是如此。

有助于设计过程调用序列、过程返回序列和活动记录的一个原则是,长度能较早确定的域放在活动记录的中间。在图 6.3 的一般活动记录中,控制链、访问链和机器状态域出现在中间。是否使用控制链和访问链,取决于语言及其编译器的设计,机器状态域需多少空间也取决于编译器的设计,因此这些域都可以在构造编译器时固定。如果对于每个活动,需要保存的机器状态信息的总数是完全相同的,那么可以用同样的代码来执行各个活动的保存和恢复。而且,当出现错误时,调试器很容易辨认栈的内容。

即使临时数据域的长度在编译时最终可以确定,但就编译器的前端而言,这个域的大小可能是未知的,因为代码生成或优化可能会缩减过程所需的临时数据区。在活动记录中,一般把临时数据域放在局部数据域的后面,它的长度的改变不会影响数据对象相对于中间域的位置。

因为每个调用都有自己的实参,因此调用者通常计算实参,并把它们传到被调用者的活动记录中。参数传递方式在6.4节讨论。在运行栈中,调用者的活动记录刚好处于被调用者的下面,如图6.12所示,因此把参数域和可能有的返回值域放在紧靠调用者活动记录的地方是有好处的。调用者可以根据对它自己活动记录末端的偏移来访问这些域,无须知道被调用者活动记录的整个安排。尤其是,对调用者来说,根本没有必要知道被调用者的局部数据或临时数据。这种参数安排的另一个好处是,可以允许变元个数可变的过程,如C语言的标准库函数printf,这个问题我们在后面讨论。

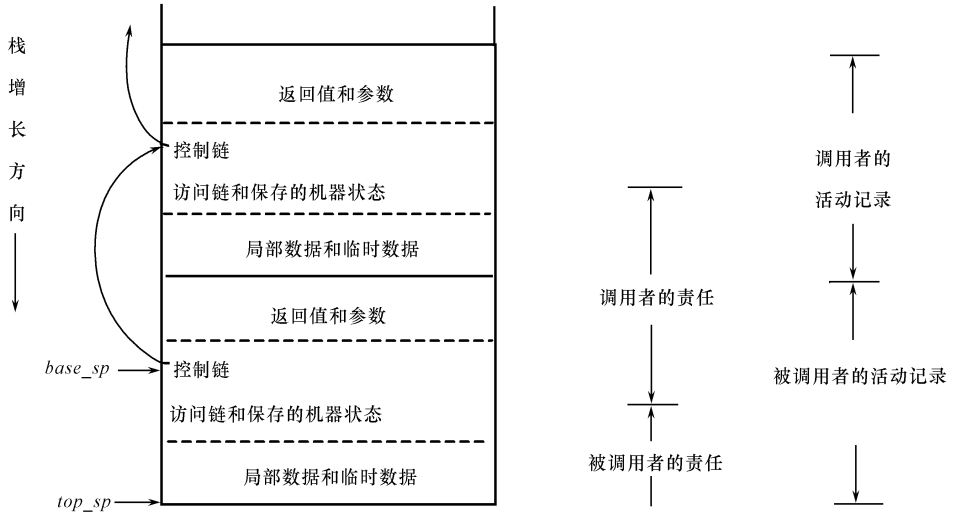


图6.12 调用者和被调用者之间的任务划分

下面给出一种调用序列和返回序列,它是受上面讨论的启发。在图6.12中,寄存器*top_sp指向栈顶活动记录的末端,另一个寄存器base_sp*指向栈顶活动记录中控制链所在的位置。假定过程p调用过程q。调用序列如下:

(1) p在栈上留出放返回值的空间,并计算实参,依次放入栈顶(也就是放到q的活动记录中),同时改变*top_sp*的值。

(2) p把返回地址和当前*base_sp*的值存入q的活动记录中,建立q的访问链,并增加*base_sp*的值。

⑶ q 保存寄存器的值和其他机器状态信息。

⑷ q 根据局部数据域和临时数据域的大小增加 *top_sp* 的值，初始化它的局部数据，并开始执行过程体，如图 6.12 所示。

返回序列如下：

⑴ q 把返回值置入邻近 p 的活动记录的地方。

⑵ q 对应上面步骤 ⑷，减小 *top_sp* 的值。

⑶ q 恢复寄存器（包括 *base_sp*）和机器状态，返回 p。

⑷ p 根据参数个数与类型和返回值类型调整 *top_sp*，然后取出返回值。

上面的调用序列和返回序列可用于过程的参数个数可变的情况（函数返回值改成用寄存器传递），例如 C 语言的标准库函数 printf。在一个程序中，调用者的实在参数的个数是清楚的，编译器产生将这些参数逆序进栈的代码，即将这些参数逆序填入被调用者活动记录的参数区，被调用函数（即 printf）虽然不知道参数区参数的个数，但是它能准确地知道第一个参数的位置（因为参数逆序进栈）。因此 printf 的实现首先取第一个参数——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、第三个参数等等。

例 6.5 一个 C 语言函数如下：

```
func()
long i;
{
    long j;
    j= i -1;
    func(j) ;
}
```

该函数在 X86/Linux 机器上编译生成的汇编代码（略去与问题无关的部分）及其注释如下：

```
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            —— 为局部变量 j 分配空间
    movl 8(%ebp),%edx       —— 取形参 i 的值到寄存器，i 的地址是 8(%ebp)
    decl %edx               —— i-1
    movl %edx,-4(%ebp)      —— i-1] j,分配给 j 的地址是-4(%ebp)
    movl -4(%ebp),%eax      —— 和下一条指令一起，完成将实参 j 的值压栈
    pushl %eax
    call func               —— 函数调用，将返回地址压栈，修改程序计数器
    add $4,%esp             —— 栈顶指针恢复到参数压栈前的位置
```

```
.L1:
    leave          —— 和下一条指令一起完成恢复老的基地址指针,
    ret            —— 将栈顶指针恢复到调用前参数压栈后的位置,
                   —— 并返回调用者
```

从上面的汇编代码,可以分析出该函数的一个活动记录的内容如图 6.13 所示。其中 *esp* 是栈顶指针寄存器(执行 movl 8(%ebp),%edx 指令时栈顶指针所指的位置就是图中所标明的位置),*ebp* 是基地址寄存器。

图 6.13 活动记录的内容及相关信息

在这个例子中,调用序列的代码是 pushl %eax、call func、pushl %ebp、movl %esp,%ebp 和 subl $4,%esp。返回序列的代码是 leave、ret 和 addl $4,%esp。

下面讨论活动记录的长度在编译时不能确定的情况。例如,图 6.14 所示的情况是局部数组的大小要等到过程激活时才能确定。过程 p 有 2 个不能静态确定大小的局部数组,在编译时,在活动记录中为这 2 个数组分别分配一个存放数组指针的单元。运行时,这些数组的大小能确定后,在栈顶为这些数组分配空间,并把起始地址置入存放数组指针的单元。这样,为这些数组分配的存储空间不是 p 的活动记录的一部分;另外,对这些数组的访问是通过活动记录中的数组指针间接进行的。

只要存储空间可以释放,就有可能出现悬空引用问题。引用某个已被释放的存储单元就叫做悬空引用(dangling reference)。使用悬空引用是一种逻辑错误,因为按大多数语言的语义,已被释放的存储单元的值是没有定义的。更糟糕的是,已被释放的存储单元可能随后被分配用来存放其他数据,因此有悬空引用错误的程序会出现难以理解的不会被捕获的错误。

例 6.6 在图 6.15 的 C 语言程序中,过程 dangle 返回一个指向绑定到局部名 j 的存储单元的指针。当控制从 dangle 返回到 main 时,dangle 的活动记录已经释放,并可能已另有安排。因为 main 中 q 的值是这个存储单元的地址,所以对 q 指向的对象的使用将是一种悬空引用。

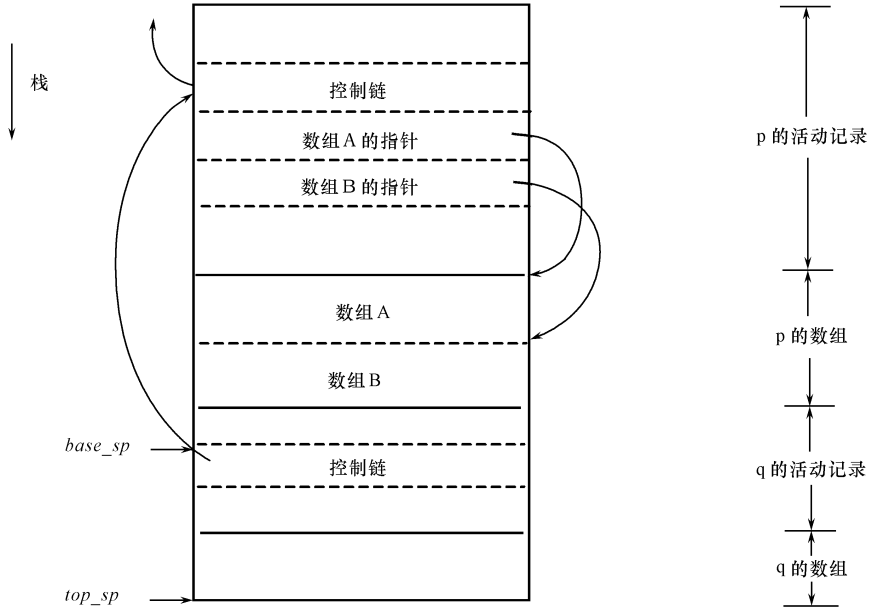


图 6.14　访问动态分配的数组

```
main(){
 {
       int*q;
       q = dangle(); 
}
 int(dangle(){
 {
       int j = 20;
       return &j;
}
```

图 6.15　q 指向已经释放了的存储单元

6.2.4　堆式分配

栈式分配策略在下列情况下行不通：

(1) 过程活动停止后,局部名字的值还必须维持。

⑵ 被调用者的活动比调用者的活动活得更长,此时活动树不能正确描绘程序的控制流。

对于上面这些情况,可以采用堆式分配策略。堆式分配把连续存储区域分成块,当活动记录或其他对象需要空间时,就为之分配一块。块的释放可以按任意次序进行。因此一段时间后,堆中可能包含交错的正在使用的和已经释放的块。

在一些程序设计语言中,过程活动停止后,部分局部变量的值还需要维持,例如 C 语言的静态局部变量(还有前面提到过的 FORTRAN 语言 save 变量)。可以把这些变量分配在静态数据区,对其余的变量仍用栈式分配,而无须把整个存储分配策略改成堆式分配,因为对堆的管理比对栈的管理困难得多。

不遵守栈式规则的还有 Pascal 语言和 C 语言的动态变量,它们由程序显式的过程调用来分配空间,由显式的过程调用来释放空间。这些变量的多少一般来说不是静态可决定的,我们通常把这些变量分配在堆区。同样,我们无须把整个存储分配策略改成堆式分配。因此 Pascal 语言和 C 语言既有运行栈,又有堆。

由程序员用显式的语句或表达式来释放动态变量所用空间是引起悬空引用的一个根源。现代的语言,如 Java,禁止程序员自己释放空间,而是用无用单元收集器来完成对无用单元的收集,我们在第 10 章中介绍。

在第 12 章有关函数式语言的实现中,我们会介绍另一些使用堆式分配的情况。

6.3 非局部名字的访问

本节讨论非局部名字的访问,虽然我们的讨论是基于活动记录的栈式分配,但同样的思想可用于堆式分配。

语言的作用域规则规定了如何处理非局部名字的访问。一种常用的规则叫做词法作用域或静态作用域规则,它仅根据程序正文静态地确定用于名字的声明。许多语言,如 Pascal,C 和 Ada,都使用静态作用域规则。我们首先考虑 C 语言那样的非局部名字。由于 C 语言不允许嵌套的过程声明,因此所有的非局部名字都可以静态地绑定到所分配的存储单元。

像 Pascal 这样的语言,它们允许过程的嵌套,并使用静态作用域,确定用于名字的声明需要根据过程的嵌套层次来决定。和 C 语言不同的是,Pascal 语言的非局部名字不一定就是全局的。运行时访问非局部名字的时候,首先要确定该非局部名字被绑定到的活动记录,我们讨论寻找这个活动记录的一种方法:利用访问链。

另一种规则叫做动态作用域规则,它是在运行时根据当前活跃着的过程活动来确定用于名字的声明。使用动态作用域的语言很少,如 Lisp,作为比较,将在 6.3.3 节讨论动态作

用域的实现。

6.3.1 无过程嵌套的静态作用域

由于 C 语言不允许过程嵌套,因此图 6.1 的排序程序改用 C 语言来写的话,其框架如图 6.16 所示。C 语言的程序由变量和函数的声明序列组成,如果在某个函数中对名字 a 有非局部引用,那么 a 必须作为外部变量,声明在所有函数的外面。函数外声明的作用域是该声明后的所有函数体,不过,若某个函数体中有该名字的重新声明的情况要除外。在图 6.16 中,readarray,partition 和 main 中对 a 的非局部引用都是引用第 (1) 行声明的数组。

```
(1) int a[11];
(2) readarray( ){...a...}
(3) int partition(y,z) int y,z; {...a...}
(4) quicksort(m,n) int m,n; {...}
(5) main( ){...a...}
```

图 6.16 用 C 语言写的排序程序

由于没有过程嵌套,6.2 节中局部名的栈式分配策略可以直接用于像 C 这样的静态作用域语言。声明在过程外面的所有名字都可以静态分配存储单元,它们的存储位置在编译时都可以知道。所以,过程体中的非局部引用可以直接使用静态确定的地址。任何其他的名字必定局部于栈顶的活动记录,可以通过 *base_sp* 指针来访问。过程嵌套会使这种方法失败,因为对非局部名字的访问需要深入到栈中访问数据,这个问题在 6.3.2 节讨论。

对非局部名字进行静态分配的一个重要好处是,程序中声明的过程可以作为参数来传递,也可以作为结果来返回 (C 语言传递和返回的是过程的指针)。这是因为在静态作用域和无嵌套过程的情况下,一个过程的任何非局部名字也是所有过程的非局部名字,它的静态地址可以被所有过程使用,而不用管这些过程是怎样被激活的。同样,如果过程作为结果返回,该被返回的过程中对非局部名字的引用,仍然是引用静态分配给这些名字的地址。

6.3.2 有过程嵌套的静态作用域

在 Pascal 语言的一个过程中,若该过程没有名字 a 的声明,则名字 a 在该过程的出现是在这样一个 a 声明的作用域中:从静态程序正文看,该声明外嵌在 a 的这个出现的外面,并且比其他这样的声明更接近 a 的这个出现。

为说明问题,把图 6.1 排序程序中的 partition 函数定义在 quicksort 过程的里面,并增加了 exchange 过程,成为图 6.17 的形式。修改后的排序程序中,过程定义的嵌套由下面的阶梯表示:

```
sort
    readarray
```

exchange
quicksort
partition

在图 6.17 中,第 (15) 行的 a 出现在函数 partition 中,最接近这个 a 的外嵌的 a 声明在第 (2) 行,它属于构成整个程序的过程。最接近的嵌套规则也可以用于过程名。第 (17) 行中由函数 partition 调用的过程 exchange 对 partition 来说是非局部的,应用这个规则,首先检查 exchange 是否定义在 quicksort 中;因为不是,所以在主程序 sort 中寻找它。

```
(1)  program sort(input,output);
(2)      var a:array[0..10] of integer;
(3)      x:integer;
(4)      procedure readarray;
(5)          var i:integer;
(6)          begin ... a ... end {readarray};
(7)      procedure exchange(i,j:integer);
(8)          begin
(9)              x:=a[i];a[i]:=a[j];a[j]:=x
(10)         end {exchange};
(11)     procedure quicksort(m,n:integer);
(12)         var k,v:integer;
(13)         function partition(y,z:integer):integer;
(14)             var i,j:integer;
(15)             begin ... a ...
(16)                   ... v ...
(17)                   ... exchange(i,j);...
(18)             end {partition};
(19)         begin ... end {quicksort};
(20)     begin ... end {sort}.
```

图 6.17 有过程嵌套的 Pascal 程序

在实现静态作用域时,需要过程的嵌套深度概念。设主程序的嵌套深度为 1,从一个过程进入一个被包围的过程时,嵌套深度加 1。因此第 11 行的 quicksort 过程的嵌套深度是 2,而第 13 行的 partition 过程的嵌套深度是 3。对名字的每次出现,把它的声明所在过程的嵌套深度作为该名字的嵌套深度。在 partition 的第 (15) 行到 (17) 行的 a,v 和 i 的嵌套深度分别

为 1,2 和 3。

过程嵌套的静态作用域的直接实现是在每个活动记录中增加一个叫做访问链的指针。如果过程 p 直接嵌在过程 q 中,那么过程 p 的活动记录的访问链直接指向最靠近的那个属于过程 q 的活动记录的访问链。

图 6.17 的程序运行时,运行栈的瞬像在图 6.18 给出。为了节约空间,图中只给出了过程名的第一个字母。sort 的活动的访问链为空,因为它不再有外围的过程。quicksort 的每个活动的访问链都指向 sort 的活动记录。图 6.18 (c) 中,partition (1,3) 活动记录的访问链指向最靠近的那个 quicksort 活动记录的访问链,即 quicksort (1,3) 活动记录的访问链。

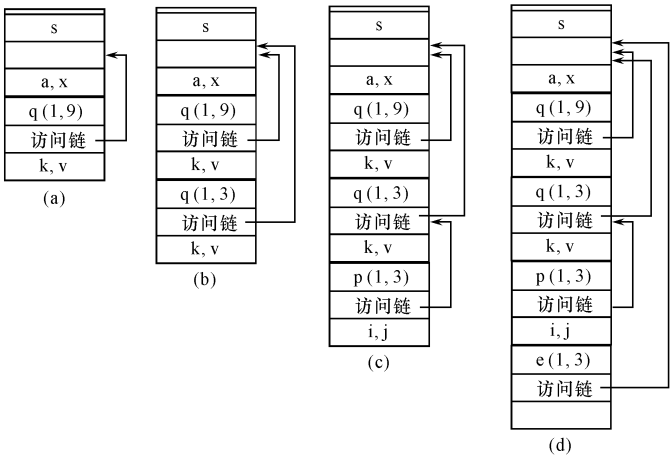


图 6.18　寻找非局部名字存储单元的访问链

假定过程 p 的嵌套深度为 n_p,它引用一个嵌套深度为 n_a 的变量 a,$n_a \leq n_p$,则 a 的存储单元可以如下找到:

(1) 当控制在 p 中时,p 的一个活动记录肯定在栈顶。首先从栈顶的活动记录开始,追踪访问链 $n_p - n_a$ 次 ($n_p - n_a$ 的值可以在编译时计算)。如果一个活动记录的访问链正好指向另一个活动记录的访问链,那么访问链的追踪用间接操作就可以完成。

(2) 追踪访问链 $n_p - n_a$ 次后,到达 a 的声明所在过程的活动记录。根据 6.1 节的讨论,它的存储单元是在相对该活动记录中某个位置的固定偏移处,这个偏移常常取为相对于访问链的偏移。

因此,过程 p 对变量 a 访问时,a 的地址由下面的二元组表示:

($n_p - n_a$,a 在活动记录中的偏移)

其中第一个分量给出追踪访问链的次数。

例如,在图 6.17 中的 (15) 和 (16) 行中,过程 partition 的嵌套深度为 3,它所引用的非局部

变量 a 和 v 的嵌套深度分别为 1 和 2。包含这些非局部变量存储单元的活动记录可以从 partition 的活动记录分别追踪访问链 3−1 = 2 次和 3−2 = 1 次而找到。

建立访问链的代码是过程调用序列的一部分。假定嵌套深度为 n_p 的过程 p 调用嵌套深度为 n_x 的过程 x,建立被调用过程访问链的代码取决于被调用过程是否嵌在调用过程的里面:

(1) $n_p < n_x$ 的情况。这表明被调用过程 x 比 p 嵌得更深,而且 x 肯定就声明在 p 中,否则 p 不能访问 x。图 6.18(a)中 sort 调用 quicksort 和图 6.18(c)中 quicksort 调用 partition 都属于这种情况。此时,被调用过程的访问链必须指向栈中刚好在它下面的调用过程的活动记录的访问链。

(2) $n_p \geq n_x$ 的情况。根据作用域规则,p 和 x 的嵌套深度分别为 1,2,...,n_x-1 的外围过程肯定相同。图 6.18(b)中 quicksort 调用本身和图 6.18(c)中 partition 调用 exchange 都属于这种情况。从调用过程追踪访问链 $n_p - n_x + 1$ 次,即到达了静态包围 x 和 p 的并且离它们最近的那个过程的最新活动记录。所到达的这个访问链就是被调用过程 x 的活动记录中的访问链应该指向的那个访问链。同样,$n_p - n_x + 1$ 的值可以在编译时计算。

当过程作为参数传递,尤其是被嵌套过程作为参数传递的情况,怎样在该过程被激活时建立它的访问链呢?我们以图 6.19 的 Pascal 程序为例说明之。在该程序的第(8)行,c 的过程体把 f 作为参数传递给 b。在 b 的体中,语句 writeln(h(2))激活 f,因为形参 h 代表的是 f,即 writeln 打印调用 f(2)的结果。可以看出,f 在 b 的体中被激活,但是从 b 的访问链难以建立 f 的访问链。

```
(1)   program param(input,output);
(2)       procedure b(function h(n:integer):
integer);
(3)           begin writeln(h(2)) end {b};
(4)       procedure c;
(5)           var m:integer;
(6)           function f(n:integer):integer;
(7)               begin f:=m+n end {f};
(8)           begin m:=0;b(f) end {c};
(9)       begin
(10)          c
(11)      end.
```

图 6.19 过程作为参数

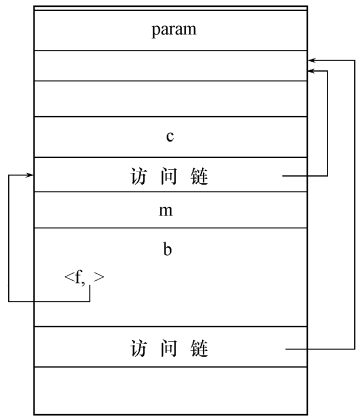


图 6.20 实过程 f 带着它的访问链一起传递

怎样解决这个问题呢？在过程作为参数传递时，必须把它的访问链和它一起传递。如图 6.20 所示，当过程 c 传递 f 时，它确定 f 的访问链，就好像 f 被调用一样，该链和 f 一起传递给 b。随后，当 f 在 b 中被激活时，该链用来建立 f 活动记录的访问链，而不按上面常规方法建立访问链。

对于 Pascal 语言来说，如果允许一个函数的返回值是函数，那么有可能出现作为返回值的函数执行时，需访问的非局部数据已经不复存在，习题 6.12 可以说明这一点。所以 Pascal 语言不允许函数类型作为函数的返回值类型。

前面已经提到，C 语言的函数声明不能嵌套，因此 C 的函数不论在什么情况下激活，在执行它的代码时要访问的数据分成两种情况：

(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 语言碰到的那种困难。

6.3.3 动态作用域

在动态作用域下，被调用过程的非局部名字到存储单元的绑定，同调用过程中该名字的绑定是一致的，即被调用过程的非局部名字 a 和它在调用过程中引用的是同样的存储单元。新的绑定仅为被调用过程的局部名字建立，这些名字在被调用过程的活动记录中占用存储单元。也可以这么说，一个名字的声明在运行时实施它的影响，直至在过程调用时遇到该名字的一个新的声明为止；此过程停止时，该影响恢复。

图 6.21 的程序可用来说明动态作用域的概念。第 (3) ～ (4) 行的过程 show 写非局部量 r 的值。按照 Pascal 的静态作用域规则，非局部量 r 在第 (2) 行声明的作用域中，所以程序的输出是：

```
0.250    0.250
0.250    0.250
```

如果是动态作用域，则它的输出是：

```
0.250    0.125
0.250    0.125
```

当主程序在第 (10) ～ (11) 行调用 show 时，写出 0.250，因为使用的是局部于主程序的 r。但是在第 (7) 行从 small 调用 show 时，输出 0.125，因为

```
(1)  program dynamic (input,output) ;
(2)      var r :real;
(3)      procedure show;
(4)          begin write (r:5:3) end;
(5)      procedure small;
(6)          var r: real;
(7)          begin r := 0.125; show end;
(8)      begin
(9)          r := 0.25;
(10)         show; small; writeln;
(11)         show; small; writeln
(12)     end.
```

图 6.21 用于说明作用域的一个程序

使用的是局部于small的变量。我们之所以在第(11)行重复第(10)的调用,目的是想说明,第(10)行的small调用结束后,第(6)行r声明的影响结束,第(2)行r声明的影响恢复。所以第(11)行再次调用show时,又输出0.250。

下面介绍两种实现动态作用域的方法。

(1) 深访问

概念上,如果访问链指向的活动记录和控制链指向的活动记录一样,那就实现了动态作用域。因此一种简单的实现是省略访问链,并用控制链搜索运行栈,寻找包含该非局部名字的第一个活动记录。深访问的意思是,搜索可能要深入运行栈中,搜索的深度取决于程序的输入,编译时决定不了。

(2) 浅访问

为程序中每个名字在静态分配的存储空间中保存它的当前值。当过程p的新活动出现时,p的局部名字n使用在静态数据区分配给n的存储单元。n的先前值可以保存在p的活动记录中,当p的活动结束时再恢复。

两种方法的权衡是:深访问对非局部名字需要较长的访问时间,但是它在活动的开始和结束处没有额外开销;浅访问则相反,它可以直接访问非局部名字,但在活动的开始和结束处需要时间来维护这些值。当函数作为参数传递和作为结果返回时,深访问可以较直截了当地实现。

6.4 参数传递

过程调用时,调用者和被调用者之间交换信息的办法通常是通过非局部名字和通过被调用过程的参数。本节讨论形参和实参联系的几种一般方法,它们是值调用、引用调用、复写-恢复和换名调用。了解语言(或编译器)的参数传递方法是很重要的,因为程序的结果依赖于所使用的方法。参数传递方法的区别基于实参是代表右值、左值还是实参本身的正文。

6.4.1 值调用

值调用是最简单的传递参数的方法。它计算实参,并把它的右值传给被调用过程。C语言和Java语言使用值调用,Pascal语言的值参数也是按这种方式传递,本章到目前为止的所有程序都是按这种方式传递参数的。值调用可以如下实现:

(1) 把形参当作所在过程的局部名看待,形参的存储单元在该过程的活动记录中。

(2) 调用过程计算实参,并把右值放入形参的存储单元中。

值调用的显著特征是,对形参的任何运算不会影响调用者的实参的值。

6.4.2 引用调用

引用调用(也叫做地址调用)的参数传递方式是,调用过程把实参存储单元的指针(即实参的左值)传给被调用过程,被调用过程对形参的任何运算就是对对应实参的运算。引用调用可以如下实现:

(1) 如果实参是有左值的名字或表达式,则把该左值放入形参的存储单元。如果实参是 a+b 或 2 这样没有左值的表达式,则把它的值计算到新的存储单元,然后传递这个单元的地址。

(2) 在被调用过程的目标代码中,任何对形参的引用都是通过传给该过程的指针来间接引用实参的。

引用调用的显著特征是,对形参的任何赋值都会影响调用者的实参。

Pascal 的 var 参数是按这种方式传递的,但要求实参是有左值的表达式。数组通常是按引用调用传递,以免传值会引起的大量数据传递。

6.4.3 复写-恢复调用

值调用和引用调用的混合叫做复写-恢复调用,也称为复写入和复写出,或值-结果调用。这种调用可以如下实现:

(1) 在控制流到被调用过程之前,调用过程计算实参,实参的右值和左值同时传给被调用过程。

(2) 在被调用过程中,像值调用那样使用实参的右值。

(3) 在被调用过程中,当控制返回调用过程时,根据传递来的实参的左值,将形参当前的值复写到实参存储单元。

这种方式和前两种方式的区别是:在值调用情况下,实参的值传给形参后,实参就不再受形参的影响;在引用调用的情况下,实参和形参是紧紧绑在一起的;在复写-恢复情况下,实参和形参也没有绑在一起,但在返回时,用形参的值来改变实参的值。

FORTRAN 的某些实现使用复写-恢复,其他实现采用引用调用。

如果被调用过程有不止一种方式访问调用者活动记录中的某个单元,则它可以用来区别引用调用和复写-传播调用。图 6.22 中第(6)行的调用 unsafe(a)建立的过程活动可以把 *a 作为非局部名字来访问,或者通过 x 访问 a。在引用调用方式下,对 x 和 a 的赋值都立即*影响 a,所以 a 的最终值是 0。但是在复写-传播方式下,实参 a 的值 1 复写到形参 *x,而 x* 的终值 2 在控制返回前复写给 *a,所以 a 的终值是* 2。

```
(1) program copyout(input,output) ;
(2)     var a:integer;
(3)     procedure unsafe(var x:integer) ;
(4)         begin x:=2;a:=0 end;
(5)     begin
(6)         a:=1;unsafe(a) ;writeln(a)
(7)     end.
```

图 6.22 不同的参数传递方式会使输出有区别

6.4.4 换名调用

换名调用出自一种计算模型——λ演算,它首先用于 Algol 60 语言。换名调用可以用 Algol 的拷贝规则来定义,具体如下:

(1) 把过程当作宏来对待,也就是在调用点,用被调用过程的体来替换调用者的调用,但是形参用对应的实参文字来代替。这种文字替换方式称为宏展开或内联展开。

(2) 被调用过程的局部名与调用过程的名字保持区别。可以认为在宏展开前,被调用过程的每个局部名字系统地被重新命名成可区别的名字。

(3) 为保持实参的完整性,实参可以由括号包围。

例 6.7 对于过程

```
procedure swap(var x,y:integer) ;
    var temp:integer;
    begin
        temp:=x;
        x:=y;
        y:=temp
    end
```

调用 swap(i,a[i])在换名调用下的实现好像它是

```
temp:=i;
i:=a[i] ;
a[i] :=temp
```

从这里可以看出换名调用和其他方式的重要区别。第 2 行引用的 a[i]和第 3 行被赋值的 a[i]可能是不同的数据单元,因为 i 的值在第 2 行被改变了。

换名调用比较复杂,我们在第 12 章介绍函数式语言的实现时再介绍它的具体实现。

虽然换名调用主要出于理论上的兴趣,但是概念上的内联展开暗示了可以缩短程序的

运行时间。建立过程的活动,包括活动记录的空间分配、机器状态的保存、链的建立和控制的转移等,都需要一定的代价。如果过程体较小,那么过程调用序列的代码可能会超过过程体的代码。此时把过程体的代码内联展开到调用者的代码中,即便程序的代码会稍长一些,效率也会提高。出于这样的考虑,C++ 和 Java 在 C 的基础上增加了内联函数。

习 题 6

6.1 使用 Pascal 的作用域规则,确定下面程序中用于名字 a 和 b 的每个出现的声明。该程序的输出是整数 1,2,3,4。

```
program a(input,output) ;
    procedure b(u,v,x,y:integer) ;
        var  a:record a,b:integer end;
             b:record b,a:integer end;
        begin
            with a do begin a := u; b := v end;
            with b do begin a := x; b := y end;
            writeln(a.a,a.b,b.a,b.b)
        end;
    begin
        b(1,2,3,4)
    end.
```

6.2 考虑下面的 C 语言程序:

```
main()
 {
     char *cp1,*cp2;

     cp1 = "12345";
     cp2 = "abcdefghij";
     strcpy(cp1,cp2);
     printf("cp1 = %s\ncp2 = %s\n",cp1,cp2);
}
```

该程序经以前的某些 C 编译器的编译,其目标程序的运行结果是:

```
cp1 = abcdefghij
cp2 = ghij
```

试分析,为什么 cp2 所指的串被修改了?

6.3 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 SPARC/Solaris 工作站上的运行结果如下:

```
Size of double,long,char = 8,4,1
Size of a,b = 24,16
```

结构体类型 a和 b的域都一样,仅次序不同,为什么它们需要的存储空间不一样?

6.4 下面给出一个 C 语言程序及其在 X86/Linux 操作系统上的编译结果。根据所生成的汇编程序来解释程序中 4 个变量的存储分配、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 语言程序生成的汇编代码:

```
        .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)"
```

6.5 假定使用：(a) 值调用；(b) 引用调用；(c) 值－结果调用；(d) 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 .
```

6.6 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 机器上的运行结果如下：

```
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 3 个形式参数的地址依次升高，而 3 个局部变量的地址依次降低。试说明为什么会有这个区别。注意，输出的数据是八进制的。

6.7 下面的 C 语言程序中，函数 printf 的调用仅含格式控制字符串 1 个参数，程序运行时输出 3 个整数。试从运行环境和 printf 的实现来分析，为什么此程序会有 3 个整数输出？

```
main ( )
 {
```

```
        printf( %d,%d,%d\n ) ;
    }
```

6.8 下面给出一个 C 语言程序及其在 SPARC/SUN 工作站上经某编译器编译后的运行结果。从运行结果看,函数 func 中 4 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 4 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 of i,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
```

6.9 一个 C 语言的函数

```
func(c,l)
char c;long l;
 {
    func(c,l) ;
}
```

在 X86/Linux 机器上编译生成的汇编代码如下:

```
        .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。)

6.10 从例 6.5 可以看到,C 程序执行时只用到了控制链,不需要使用访问链。为什么 Pascal 程序执行时需要使用访问链,而 C 程序不需要。

6.11 下面是求阶乘的 Pascal 程序。画出程序第 3 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.
```

6.12 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.
```

6.13 为什么 C 语言允许函数类型(的指针)作为函数的返回值类型,而 Pascal 语言却不允许?

6.14 一个 C 语言程序如下:

```
int n;

int f()
int g();
{
    int m;

    m = n;
    if(n == 0) return 1;
    else {
        n = n-1; return m* g();
    }
}

main()
{
    n = 5; printf("%d factorial is %d\n",n,f());
}
```

该程序的运行结果不是我们所期望的

```
5 factorial is 120
```

而是

0 factorial is 120

试说明原因。

6.15 下面程序在 SPARC/SUN 工作站上运行时陷入死循环,试说明原因。如果将第 7 行的 long *p 改成 short *p,并且将第 22 行 long k 改成 short k 后,loop 中的循环体执行一次便停止了。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;
}
```

6.16 一个 C 语言程序

```
main()
 {
      func();
      printf("Return from func\n");
```

```
}

func()
{
    char s[4];

    strcpy(s, "12345678");
    printf("%s\n",s);
}
```

在 X86/Linux 操作系统上的运行结果如下：

```
12345678
Return from func
Segmentation fault (core dumped)
```

试分析为什么会出现这样的运行错误。

6.17 一个过程作为实在参数传递，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 都是使用这种方式。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。

我们以下面的 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.
```

第 7 章　中间代码生成

在第 1 章已经介绍，编译器的前端把源程序翻译成中间表示，后端从中间代码产生目标代码，与目标语言有关的细节尽可能限制在后端。使用独立于机器的中间形式的好处是：

⑴ 再目标（retargeting）比较容易。把针对新机器的后端加到现成的前端上，可以得到另一种机器的编译器。

⑵ 独立于机器的代码优化器可用于这种中间表示。第 9 章将介绍这种代码优化。

因此，虽然可以把源程序直接翻译并生成目标代码，但编译器一般都采用中间语言。

本章将用第 4 章的语法制导定义方法来说明程序设计语言的结构怎样被翻译成中间形式。为简单起见，假定对源程序的分析和静态检查已经完成，如图 7.1 表示的那样。本章大多数语法制导定义可以用第 4 章的技术在自下而上或自上而下的分析期间实现，所以，如果愿意的话，中间代码生成可以在分析阶段完成。

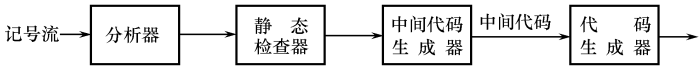


图 7.1　中间代码生成器的位置

7.1　中 间 语 言

4.2 节介绍的语法树是一种图形化的中间表示，本节再介绍几种常用的中间表示：后缀表示、其他图形表示和三地址代码。本章主要使用三地址代码。从程序设计语言的各种结构产生三地址代码的语义规则类似于产生语法树或后缀表示的那些规则。

7.1.1　后缀表示

表达式 E 的后缀表示可以如下递归定义：

⑴ 如果 E 是变量或常数，那么 E 的后缀表示就是 E 本身。

⑵ 如果 E 是形式为 $E_1 \mathrm{op} E_2$ 的表达式，其中 op 是任意的二元算符，那么 E 的后缀表示是 $E'_1 E'_2 \mathrm{op}$，其中 E'_1 和 E'_2 分别是 E_1 和 E_2 的后缀表示。

(3) 如果 E 是形式为 (E_1) 的表达式,那么 E_1 的后缀表示也是 E 的后缀表示。

后缀表示不需要括号,因为算符的位置及其运算对象的个数使得后缀表示仅有一种解释。例如,(8-4)+2 的后缀表示是 84-2+,而 8-(4+2) 的缀表示 842+-。

上面的定义很容易拓广到含一元算符的表达式。

后缀表示的最大优点是便于计算机处理表达式。利用一个栈,自左向右扫描表达式的后缀表示。每碰到运算对象,就把它压进栈;每碰到运算符,就从栈顶取出相应个数的运算对象进行计算,再将结果压进栈。最终的结果留在栈顶。

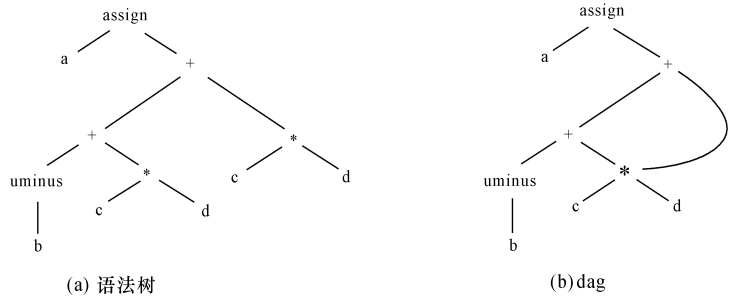
后缀表示也可以拓广到表示赋值语句和控制语句,但很难用栈来描述它的计算。

后缀表示又叫做逆波兰表示,它是波兰逻辑学家卢卡西维奇发明的。

7.1.2 图形表示

语法树是一种图形化的中间表示,它是分析树的浓缩表示,描绘了源程序在语义上的层次结构。后缀表示是语法树的一种线性表示,例如,赋值语句 a := (-b+c*d)+c*d 的语法树在图 7.2(a),对应的后缀表示为

*a b uminus c d * + c d * + assign*



(a) 语法树 (b) dag

图 7.2 a := (-b+c*d)+c*d 的图形表示

有向无环图(directed acyclic graph,简称 dag)也是一种中间表示。和语法树相比,它以更紧凑的方式给出同样的信息,因为公共子表达式标识出来了。见图 7.2(b),公共子表达式 c*d 不止一个父结点。在考虑代码优化时,有向无环图比语法树更适用。

表 7.1 的语法制导定义构造赋值语句的语法树,它是 4.2 节构造表达式语法树的语法制导定义的一个拓展,其中 *mknode(op, child)* 是构造一元运算结点的函数。这里用的是二义文法,假定算符的结合性和优先关系与通常的一样,虽然没有把它们加入文法。这个定义从输入 a := (-b+c*d)+c*d 构造出图 7.2(a) 的语法树(按照该定义,叶结点的形式应该像图 4.5 的叶结点那样)。

表 7.1 构造赋值语句语法树的语法制导定义

产生式	语义规则
$S \rightarrow \mathrm{id} := E$	$S.nptr := mknode(\ assign\ , mkleaf(\mathrm{id}, \mathrm{id}.entry), E.nptr)$
$E \rightarrow E_1 + E_2$	$E.nptr := mknode(\ +\ , E_1.nptr, E_2.nptr)$
$E \rightarrow E_1 * E_2$	$E.nptr := mknode(\ *\ , E_1.nptr, E_2.nptr)$
$E \rightarrow -E_1$	$E.nptr := mknode(\ uminus\ , E_1.nptr)$
$E \rightarrow (E_1)$	$E.nptr := E_1.nptr$
$F \rightarrow \mathrm{id}$	$E.nptr := mkleaf(\mathrm{id}, \mathrm{id}.entry)$

修改构造结点的函数,仍然用这个语法制导定义,可以构造出有向无环图。如果构造结点的函数首先检查是否已经有相同的结点存在,有就返回先前构造的这种结点的指针而不是构造新结点,那么就可以得到 dag。这个语法制导定义从输入 $a := (-b+c*d) + c*d$ 构造出的 dag 见图 7.2(b)。

7.1.3 三地址代码

三地址代码是一般形式为

$$x := y\ op\ z$$

的语句序列,其中 x、y 和 z 是名字、常数或编译器产生的临时变量,op 代表算符,如定点或浮点算术算符,或是对布尔类型数据操作的逻辑算符等。之所以叫做三地址代码,是因为每个语句通常包含三个地址,即两个运算对象的地址和一个结果的地址。因为每个语句的右边只有一个算符,所以源语言的表达式 $x+y*z$ *翻译成的三地址语句序列是*

$$t_1 := y * z$$

$$t_2 := x + t_1$$

其中 t_1 和 t_2 是编译器产生的临时名字。用临时名字保存中间结果使得较容易为三地址代码重新安排计算次序,在后缀表示上要改变计算次序是很困难的。

三地址代码是语法树或 dag 的一种线性表示,其中新增加的临时名字对应图的内部结点。图 7.2 的语法树和 dag 由图 7.3 的三地址语句序列表示。

三地址语句类似于汇编代码。语句可以有符号标号,而且可以有控制流语句。下面是本书常用的三地址语句:

(1) 形式为 $x := y\ op\ z$ *的赋值语句,其中 op 是二元算术或逻辑运算。*

(2) 形式为 $x := op\ y$ *的赋值语句,其中 op 是一元运算。一元运算主要包括一元减、逻辑否定、移位运算和类型转换运算。*

(3) 形式为 $x := y$ *的复写语句。*

$t_1 := -b$	$t_1 := -b$
$t_2 := c * d$	$t_2 := c * d$
$t_3 := t_1 + t_2$	$t_3 := t_1 + t_2$
$t_4 := c * d$	$t_4 := t_3 + t_2$
$t_5 := t_3 + t_4$	$a := t_4$
$a := t_5$	
(a) 语法树的代码	(b) dag 的代码

图 7.3 对应图 7.2 的树和 dag 的三地址代码

(4) 无条件转移 *goto L*。标号为 L 的语句是下一步将要执行的三地址语句。

(5) 形如 *if* x relop y *goto L* 的条件转移。这种指令用于 x 和 y 的关系运算（<、=和>=等）。如果关系成立，执行标号为 L 的语句；否则，与通常的顺序一样，执行 if *x* relop y goto L 的下一个三地址语句。

(6) param *x* 和 call p，n 用于过程调用，其中 n 表示实参个数。return y 用于过程返回，y 代表返回值，它也可以不出现。过程调用的典型使用是

```
param x1
param x2
...
param xn
call p, n
```

这些语句是过程调用 $p(x_1, x_2, \ldots, x_n)$ 的中间代码。

(7) 形式为 $x := y[i]$ 和 $x[i] := y$ 的索引赋值。第一个语句把 y 的存储单元以上第 i 个存储单元的值赋给 x。第二个语句把 y 的值赋给 x 的存储单元以上第 i 个存储单元。在这些指令中，x，y 和 i 都代表数据对象。

(8) 形式为 $x := \&y$，$x := *y$ 和 $*x := y$ 的地址和指针赋值。第一个语句把 y 的存储单元地址赋给 x，可以猜想 y 是名字（或临时变量），它表示像 a 和 A[i，j] 这样有左值的表达式，x 是指针变量或临时变量，即 x 的右值是某个对象的左值。在第二个语句中，y 是右值为存储单元地址的指针变量或临时变量，x 的右值等于那个存储单元的内容。最后，$*x := y$ 把 y 的右值置入 x 指向的存储单元。

选译适当的算符集合是中间代码设计的重要问题。很显然，它必须大到足以实现源语言的操作。较小的算符集合易于在新的目标机器上实现，但是，较小的算符集合可能迫使前端对源语言的某些运算产生较长的三地址语句序列，此时如果想产生好代码的话，优化器和代码生成器的工作会比较艰巨。

三地址语句是中间代码的抽象形式。在编译器中,三地址语句可以用记录来实现,这种记录有算符域和运算对象域,不同的编译器对这种记录的设计可能是不一样的。

7.2 声明语句

在分析过程或程序块的声明序列时,为局部名字建立符号表条目,并为它分配存储单元。这样,符号表中包含名字的类型和分配给它的存储单元的相对地址等信息,相对地址是对静态数据区基址的偏移或是对活动记录中某个基址的偏移。

前端分配地址时必须想到目标机器,目标机器的指令系统可能偏爱数据对象和它们地址的某种安排。在这里忽略数据对象的对齐问题。

7.2.1 过程中的声明

C、Java、Pascal 和 FORTRAN 这些语言的语法允许一个过程中的所有声明集中在一起处理。在这种情况下,可用全局变量,例如 offset,来记住下一个可用的相对地址。

在图 7.4 的翻译方案中,非终结符 P 产生形式为 id:T 的声明序列,再产生可执行语句 *S*,*本节先考虑声明序列。在检查第一个声明之前,offset 置为* 0。每次为一个名字分配存储单元时,它的偏移等于 offset 的当前值,同时 offset 增加由该名字指示的数据对象的宽度。

过程 enter(name,type,offset) 为名字 name 建立符号表条目,该名字的类型是 type,它在数据区的相对地址是 offset。用综合属性 type 和 width 表示非终结符的类型和宽度(该类型的对象所需的存储单元数)。属性 type 代表从基本类型 integer 和 real、应用类型构造器 *pointer 和 array 构造的类型表达式。如果类型表达式用图形表示,那么属性 type 可以是一个* 指针,指向代表类型表达式的结点。

在图 7.4 中,整数宽度是 4,实数宽度是 8,数组的宽度由每个元素的宽度乘以数组元素的个数而得到,每个指针的宽度假定为 4。在 Pascal 和 C 中,在看见指针所指对象的类型之前可以先看见指针,由于所有的指针通常都有同样的宽度,因此这里的存储分配是简单的。

7.2.2 作用域信息的保存

现在考虑像 Pascal 这样允许过程嵌套的语言。为简单起见,仅讨论无参过程,并且认为过程不会递归。所讨论语言的文法如下:

$$P \rightarrow D\ S$$
$$D \rightarrow D\ ;\ D \mid \text{id}\ :\ T \mid \text{proc id}\ ;\ D\ ;\ S \qquad (7.1)$$

在这样的语言里,局部于每个过程的名字仍然可以用图 7.4 的方式来分配相对地址,每个过程建立单独的符号表。这样,每个过程要有自己的符号表指针和自己的 offset。

$P \rightarrow$	{offset := 0}
$D\ S$	
$D \rightarrow D\ ;D$	
$D \rightarrow$ id : T	{enter(id.name, T.type, offset) ;
	offset := offset + T.width}
$T \rightarrow$ integer	{T.type := integer ;
	T.width := 4}
$T \rightarrow$ real	{T.type := real ;
	T.width := 8}
$T \rightarrow$ array [num] of T_1	{T.type := array(num.val, T_1.type) ;
	T.width := num.val * T_1.width}
$T \rightarrow \uparrow T_1$	{T.type := pointer(T_1.type) ;
	T.width := 4}

图 7.4 计算被声明名字的类型和相对地址

在一边扫描一边建立符号表和完成存储分配的情况下,当碰到过程嵌套时,对外围过程声明的处理需要暂时停止,等被嵌套过程处理完后再继续。可以用两个栈分别保存尚未处理完的过程的符号表指针和它们的 offset,这两个栈顶的元素分别是正在处理的过程的符号表指针和 offset。

按照这种方式,基于文法 (7.1) 的翻译方案见图 7.5,其中 T(类型) 产生式的语义动作和图 7.4 的一致,我们略去非终结符 S(语句) 的语义动作在 7.3 节开始介绍。图 7.5 的语义动作根据下面的操作定义:

(1) mktable(previous) 建立新的符号表,并返回新符号表的指针。变元 previous 指向先前建立的符号表,可以猜想,这是直接外围过程的符号表。指针 previous 放在新建符号表的首部,首部除了包括直接外围过程的符号表的指针外,还包含所有局部变量所需存储单元的总数等信息。

(2) enter(table, name, type, offset) 在 table 指向的符号表中为变量名 name 建立新条目。和前面一样,enter 把类型 type 和相对地址 offset 置于该条目的域中。如果要说得详细一些的话,本过程还需要完成检查名字是否重复定义等事情。

(3) addwidth(table, width) 把符号表 table 所有条目的累加宽度记录在该符号表的首部。

(4) enterproc(table, name, newtable) 在 table 指向的符号表中为过程名 name 建立新条目。

变元 newtable 指向过程 name 本身的符号表。

再对图 7.5 的语义动作做一些解释。对于过程声明 D→proc id ;N D_1 ;S,在扫描 D_1 之前,需要建立新的符号表,并让它指向直接外围过程的符号表,然后将 D_1 中声明的名字的条目建立在该新符号表中,同时根据新的 offset 进行存储分配。这些动作是通过标记非终结符 N 的动作完成的。在结束内嵌过程的扫描,执行 D→proc id ;N D_1 ;S 右部的动作时,由 D_1 产生的所有声明的宽度在 offset 栈的顶上,用 addwidth 把它记录在符号表中,然后 tblptr(符号表指针)栈和 offset 栈的顶元弹出,再把过程名 id 的条目建立在直接外围过程的符号表中。这些都结束后,继续分析外围过程的声明。

非终结符 M 的动作完成一些初始化的工作,它用操作 mktable(nil)建立最外层作用域的符号表,并用两个 push 操作来完成 tblptr 栈和 offset 栈的初始化。

$P \rightarrow MDS$	{addwidth(top(tblptr), top(offset));
	pop(tblptr); pop(offset)}
$M \rightarrow \varepsilon$	{t := mktable(nil);
	push(t, tblptr); push(0, offset)}
$D \rightarrow D_1 ; D_2$	
$D \rightarrow$ proc id ;N D_1 ;S	{t := top(tblptr);
	addwidth(t, top(offset));
	pop(tblptr); pop(offset);
	enterproc(top(tblptr), id.name, t)}
$D \rightarrow$ id :T	{enter(top(tblptr), id.name, T.type, top(offset));
	top(offset) := top(offset) + T.width}
$N \rightarrow \varepsilon$	{t := mktable(top(tblptr));
	push(t, tblptr); push(0, offset)}

图 7.5 处理嵌套过程中的声明

按图 7.5 的翻译方案,为图 6.17 的 Pascal 程序生成的符号表在图 7.6 给出。过程的符号表之间用双向链连接,从这个双向链可以知道过程的嵌套关系,例如,过程 readarray、exchange 和 quicksort 的符号表逆向指向直接外围过程 sort 的符号表,而 sort 的符号表中有 3 个指针指向这 3 个过程。

当碰到名字的引用性出现时,若在本过程的符号表中找不到这个名字,那么就需要到它的外围过程的符号表中去找,逆向指针可用于这个目的。若语言有预定义的标准标识符(程序员可以重新定义其含义),如 Pascal 语言的 integer, true 等,那么主过程符号表的逆向指针应该指向标准标识符的符号表,以便确定程序员没有定义的标识符是不是一个标准标识符。

我们实际上已经解释了 7.3 节用到的符号表查找函数 *lookup*。

如果允许过程递归的话，图 7.5 的翻译方案需要修改。在产生式 D→proc id ; ND_1 ; S 的语义动作中，该过程 id 是在把该过程体处理完后再进入符号表的，这样，在 S 中若有直接递归调用，在符号表查找该 id 时会报告没有定义。

若是有参过程，对构造符号表来说，形式参数的处理和其他局部名字的处理没有很大的区别，但是填在条目中的属性，如存储分配信息等，会有些区别。

如果是一遍扫描的编译器，每个过程被扫描后，它的目标代码已经生成，若无其他需要，该过程的符号表可以释放。这时编译器可以将符号表组织成一个栈，碰到一个过程声明时将该过程声明的名字进栈，该过程被扫描结束时将它的符号全部弹出栈。

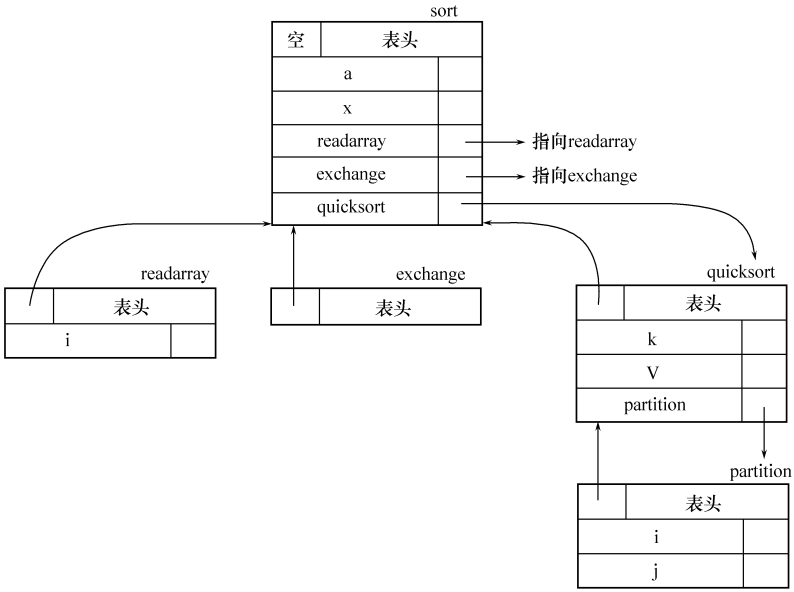


图 7.6 嵌套过程的符号表

7.2.3 记录的域名

在图 7.4 文法上增加下面一个产生式使得非终结符 *T* 除了可以产生基本类型、指针和数组外，还能产生记录类型：

T→record *D* end

在图 7.7 的翻译方案中，语义动作强调了记录类型中域的存储安排同过程的局部名字在活动记录中安排的相似性。因为过程定义不影响图 7.5 的宽度计算，因此把上面的产生式放

宽到允许过程定义出现在记录类型里面,这只是为了翻译方案的简洁易懂。

看见关键字 record 后,标记非终结符 L 的动作为域名建立新的符号表。该符号表的指针压进 tblptr 栈,相对地址 0 压入 *offset* 栈。产生式 $D \rightarrow id : T$ 的动作把域名 id 的信息加入记录类型的符号表。在记录类型的域分析完后,offset 栈顶保存记录类型中所有对象的总宽度。在图 7.7 中,end 后的动作把总宽度作为综合属性 T.width 返回。把构造器 record 作用于该记录的符号表指针可以得到 T.type。在 7.3 节,这个指针用来恢复记录类型中域名的名字、类型和相对地址。

$T \rightarrow$ record L D end　　{T.type ≔ *record*(*top*(tblptr)) ;
　　T.*width* ≔ *top*(*offset*) ;
　　pop(tblptr) ; *pop*(*offset*)}
$L \rightarrow \varepsilon$　　{t ≔ mktable(nil) ;
　　push(t, tblptr) ; *push*(0, offset)}

图 7.7　为记录中的域名建立符号表

记录类型定义和过程声明的处理还是有区别的。处理记录类型时并未真正为哪个变量分配了存储单元,只是决定了该类型的宽度和每个域在记录存储空间中的相对位置。在处理记录类型的变量声明时,才真正在活动记录中为该记录类型的变量分配存储单元。

7.3　赋值语句

在本节中,表达式的类型可以是整型、实型、数组和记录。为使翻译方案简洁,采用二义的表达式文法,并选择加运算作为二元运算的代表。作为赋值语句翻译成三地址指令的一部分,下面说明怎样从符号表中查找名字,怎样访问数组元素和访问记录的域。

7.3.1　符号表中的名字

在 7.1 节介绍中间语言时,为直观起见,让名字本身直接出现在中间语言中,但是应该把名字理解为它们在符号表中位置的指针。实际在处理源程序时,当在赋值语句中遇到名字的时候,需要在符号表中查找它的定义,获得它的属性,然后在生成的三地址代码中使用它在符号表中位置的指针。

在图 7.8 的翻译方案中,名字 id 的 name 属性代表组成该名字的字符序列,lookup(id.*name*)*用于根据名字的拼写检查符号表中是否存在该名字的条目。如果有,返回该条目的*

指针,否则 lookup 返回 nil,以表示没有找到。是否有过程嵌套,会影响 lookup 函数的设计,但不影响图 7.8 的翻译方案使用该函数来查符号表。

```
S→id := E     {p := lookup(id.name);
               if p ≠ nil then
                   emit(p, ':=', E.place)
               else error}
E→E1 + E2     {E.place := newtemp;
               emit(E.place, ':=', E1.place, '+', E2.place)}
E→- E1        {E.place := newtemp;
               emit(E.place, ':=', 'uminus', E1.place)}
E→(E1)        {E.place := E1.place}
E→id          {p := lookup(id.name);
               if p ≠ nil then
                   E.place := p
               else error}
```

图 7.8 为赋值语句产生三地址代码的翻译方案

*E 的属性 place 用来记住符号表条目的地址。函数 newtemp 用来产生一个新的临时变*量的名字,把该名字也存入符号表,并返回该条目的地址。过程 emit 将其参数写到输出文件上,就像 Pascal 语言的 writeln 一样。emit 的参数构成一个三地址语句。

如果碰到像 p.info 这样对记录的域的访问,如何查表找到所需的属性呢?首先,从过程的名字表中可以找到 p,它应该是记录类型的变量。根据 7.2.3 节我们可以知道,p 的类型是 record (tblptr),tblptr 是该记录类型的符号表,从该表中可以找到 info 的条目,从而我们可以得到所需的属性。

7.3.2 临时名字的重新使用

使用这样的假定:每当需要临时变量时,newtemp 产生新的临时名字。这在优化编译器中尤其有用,在第 9 章将会看到这样做是合理的。但是,大量临时变量会增加编译时符号表管理的负担,在非优化编译器中也会增加运行时临时数据占用的空间。可以根据临时变量的生存期特征,修改 newtemp 函数,使临时变量可以重新使用。

在语法制导的翻译期间,大量的临时变量由图 7.8 这样的语义动作产生。例如,$E \rightarrow E_1 + E_2$ 的动作产生的代码的一般形式为:

计算 E_1 到 t_1

计算 E_2 到 t_2

$t_3 \doteq t_1 + t_2$

从综合属性 E.place 的动作可以看出，t_1 和 t_2 在程序的其他地方并不引用。这些临时变量的生存期像配对括号那样嵌套或并列。事实上，计算 E_2 时用的所有临时变量的生存期都包含在 t_1 的生存期中。所以有可能修改 newtemp，使得它在过程的数据区域中用一个小数组来保存临时数据，好像它是一个栈一样。

为简单起见，假定仅处理整数。使用一个计数器 c，它的初值为 0。每当临时名字作为运算对象使用时，c 减少 1；每当要求产生新的临时名字时，使用 $\$c$，并把 c 加 1。注意，这个临时数据的“栈”在运行时并没有压栈和退栈的操作，虽然编译器可能碰巧会产生在“栈顶”存或取临时数据的指令。

例 7.1　考虑赋值语句

x ≐ a*b+c*d−e*f

表 7.2 给出了 *newtemp 被修改后，由图* 7.8 的语义动作产生的三地址语句序列。这张表还包含了每个语句生成中间代码后 c 的“当前值”。例如计算 $0−$1 时，c 的值减小到 0，所以用 $0 来保存结果。

有些场合，例如循环语句，临时变量的赋值和/或引用不止一次，这时临时变量不能按上述后进先出的方式指派名字。由于它们很少出现，可以对所有这样的临时值单独指派名字。

表 7.2　基于临时变量生成期特征的三地址代码

语句	*c* 的值
	0
$0 ≐ a*b	1
$1 ≐ c*d	2
$0 ≐ $0+$1	1
$1 ≐ e*f	2
$0 ≐ $0−$1	1
x ≐ $0	0

7.3.3　数组元素的地址计算

一个数组的所有元素通常按一定的次序存于连续的存储块中，这样可以迅速访问这些元素。一维数组的元素一般是顺序存放，如果每个数组元素的宽度是 w，那么一维数组 A 的第 i 个元素从地址

$$base + (i - low) \times w \tag{7.2}$$

开始,其中 low 是下标的下界,base 是分配给该数组的地址 (可能是活动记录中的相对地址),即 base 是 A [low] 的地址。

如果把表达式 (7.2) 重写成

$$i \times w + (base - low \times w)$$

那么可以在编译时完成该表达式后一部分的计算,从而减少了运行时的计算。

二维数组通常用两种形式之一存储:行为主 (一行接一行) 或列主为 (一列接一列)。FORTRAN 语言使用列为主形式,Pascal 语言和 C 语言都使用行为主形式。对于一个 2×3 的 A 数组,若是行为主,其元素的存放次序是:

A [1,1] ,A [1,2] ,A [1,3] ,A [2,1] ,A [2,2] ,A [2,3]

若是列为主,其元素的存放次序是:

A [1,1] ,A [2,1] ,A [1,2] ,A [2,2] ,A [1,3] ,A [2,3]

在行为主的情况下,因为 A [i,j] 等价于 A [i] [j] ,因此可以说是各个数组 A [i] 依次连续存放。

在行为主的两维数组情况下,A [i_1, i_2] 的地址可以由公式

$$base + ((i_1 - low_1) \times n_2 + (i_2 - low_2)) \times w$$

计算,其中 low_1 和 low_2 分别是这两维的下界,n_2 是第 2 维的大小。即,如果 $high_2$ 是 i_2 值的上界,那么 $n_2 = high_2 - low_2 + 1$。假定 i_1 和 i_2 是编译时不能知道的值,我们可以把上面表达式重写成

$$((i_1 \times n_2) + i_2) \times w + (base - ((low_1 \times n_2) + low_2) \times w) \tag{7.3}$$

同样,该表达式的后一项可以在编译时计算。

可以推广行为主或列为主的形式到多维数组。行为主形式的存储可以这样理解,当朝高地址扫描存储块时,最右边的下标变化最快,就像里程计上的数字。表达式 (7.3) 的推广使得 A [i_1, i_2, ..., i_k] 的地址表达式如下:

$$\begin{aligned}&(\ldots((i_1 \times n_2 + i_2) \times n_3 + i_3) \ldots) \times n_k + i_k) \times w \\ &\quad + base - ((\ldots((low_1 \times n_2 + low_2) \times n_3 + low_3) \ldots) \times n_k + low_k) \times w\end{aligned} \tag{7.4}$$

因为对所有的 j, $n_j = high_j - low_j + 1$ 是固定的,(7.4) 第 2 行的项可以在编译器分析数组声明时计算,存于符号表的 A 条目中。列为主的形式的布局相反,最左边的下标变化最快。

某些语言允许数组的大小在过程调用时动态指定,这种数组在运行栈上的分配在 6.2 节已经讨论了,其元素的访问公式和固定大小的数组相同,但由于上下界在编译时不知道,因此地址计算全部在运行时完成。

7.3.4 数组元素地址计算的翻译方案

根据 (7.4) 式知道,k 维数组引用 A [i_1, i_2, ..., i_k] 的三地址代码主要是完成

$$(\ldots((i_1 \times n_2 + i_2) \times n_3 + i_3) \ldots) \times n_k + i_k \tag{7.5}$$

的计算,然后乘以 w,再加上(7.4)式第2行的值。(7.5)式的值可以由递推计算

$$e_1 = i_1$$

$$e_m = e_{m-1} \times n_m + i_m \tag{7.6}$$

来完成一直到 m=*k* 为止。

根据(7.6)的递推计算,除了第一个下标表达式 i_1 外,对其他每一个下标表达式,需要产生乘和加两个三地址语句(除了下标表达式本身值计算的三地址语句外),因此在处理下标表达式时需要访问符号表中 A 的条目,以得到 A 各维的大小。

如果用有下列产生式

L→id [*Elist*] | id

Elist→ *Elist*, *E* | *E*

的非终结符 L 取代图7.8中id,那么数组引用可以出现在赋值语句中。如果仅用综合属性,在处理 *Elist*→ *E* 和 *Elist*→ *Elist*, *E* 时,访问不到符号表中 A 的条目,因为这是数组 id 的属性。加标记非终结符也解决不了。为了解决这个问题,将产生式重写为

L→ *Elist*] | id

Elist→ *Elist*, *E* | id [E

即数组名和最左边一个下标表达式连在一起。这个文法虽然不直观,但是从下面的翻译方案可以知道它能解决我们的问题。

使用 Elist 的综合属性 array 来传递符号表中数组名条目的指针,并使用 Elist.ndim 来记录已分析过的下标表达式的个数。函数 limit(array, j) 返回 n_j,它是该数组(array 指向它的符号表条目)第 j 维的大小。函数 base(array) 和 invariant(array) 分别从符号表条目中取(7.4)式第2行 *base* 的值和除 *base* 以外部分的值。*最后,Elist.place 指示临时变量,该变量*保存根据 Elist 的下标表达式计算的值。

左值 L 有两个属性,place 和 offset。当 L 是简单名字时,L.place 是该名字的符号表条目指针,L.offset 是 null,后者表示该左值是一个简单名字而不是数组元素引用。和图7.8一样,非终结符 E 有 place 属性,并且含义一样。

将语义动作加到下面文法上:

(1) S→ *L* = *E*

(2) E→ *E* + *E*

(3) E→ (E)

(4) E→ *L*

(5) L→ *Elist*]

(6) L→id

(7) Elist→ *Elist*, *E*

(8) Elist→id [E

和没有介绍数组元素引用时的表达式一样,三地址代码由调用过程 emit 产生。

如果 L 是简单名字,则产生正常的赋值;否则产生对由 L 的两个属性确定的存储单元的索引赋值:

(1) S→*L* = *E* {if *L*.*offset* = null then /* *L* 是简单变量 */
emit(*L*.*place*, " = ", E.*place*)
else
emit(*L*.*place*, "[", *L*.*offset*, "]", " = ", E.*place*)}

算术表达式的代码和图 7.8 的完全一样:

(2) E→E_1 + E_2 {E.*place* = *newtemp*;
emit(E.*place*, " = ", E_1.*place*, " + ", E_2.*place*)}

(3) E→(E_1) {E.*place* = E_1.*place*}

如果 E 产生数组元素 L,则需要 L 的右值,可用索引得到存储单元 L.*place*[L.*offset*] 的内容:

(4) E→*L* {if *L*.*offset* = null then /* *L* 是简单变量 */
E.*place* = *L*.*place*
else begin
E.*place* = *newtemp*;
emit(E.*place*, " = ", L.*place*, "[", L.*offset*, "]")
end}

L.*place* 和 *L*.*offset* 都是新的临时变量,前者对应 (7.4) 式的第二项;后者保存 w 乘以 Elist.*place* 的值,对应 (7.4) 的第一项:

(5) L→*Elist*] {L.*place* = *newtemp*;
emit(L.*place*, " = ", base(Elist.*array*), "(", invariant(Elist.*array*));
L.*offset* = *newtemp*;
emit(L.*offset*, " = ", Elist.*place*, "(", w)}

offset 为空时表示简单名字:

(6) L→*id* {L.*place* = id.*place*;
L.*offset* = null}

当看见下一个下标表达式时,使用递推公式 (7.6)。在下面的动作中,$Elist_1$.*place* 对应 (7.6) 的 e_{m-1},Elist.*place* 对应 e_m。如果 $Elist_1$ 有 m−1 个成分,那么产生式左边的 Elist 有 m 个成分:

(7) Elist→$Elist_1$, E {t = *newtemp*;
m = $Elist_1$.*ndim*+1;

emit(t, ‘≒’, Elist$_1$.place, ‘(’, limit(Elist$_1$.array, m))；
emit(t, ‘≒’, t, ‘+’, E.place)；
Elist.array ≒ Elist$_1$.array；
Elist.place ≒ t；
Elist.ndim ≒ m}

下面的 E.place 保存表达式 E 的值，也是(7.6)式 m=1 时的值：

(8) Elist→id ‘[’E {Elist.place ≒ E.place；
Elist.ndim ≒ 1；
Elist.array ≒ id.place}

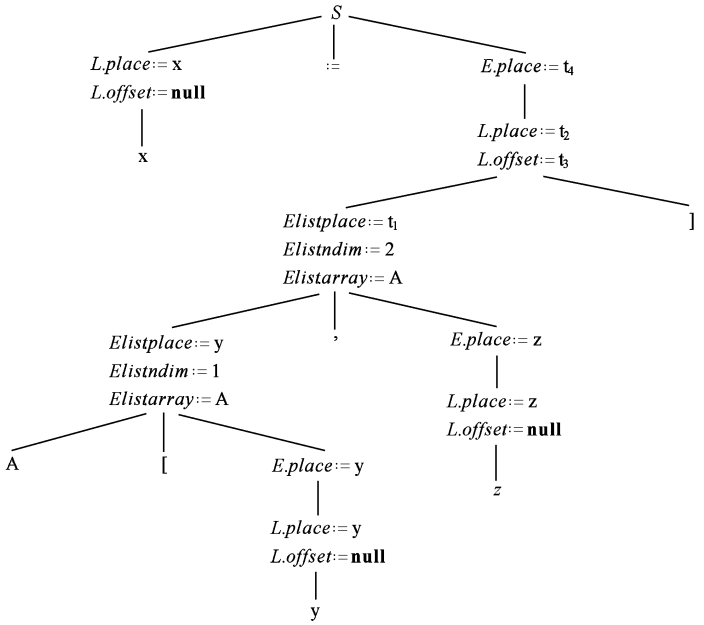


图 7.9 x ≒ A[y,z]的注释分析树

例 7.2 设 A 是 10×20 的数组，n_1=10 且 n_2=20，取 w=4。赋值语句 x ≒ A[y,z]的注释分析树如图 7.9 所示。该赋值语句翻译成下列三地语句序列

t_1 ≒ y×20
t_1 ≒ t_1+z
t_2 ≒A−84
t_3 ≒ 4×t_1

$t_4 := t_2[t_3]$

$x := t_4$

对每个变量,我们已用它的名字代替了 id.place。

7.3.5 类型转换

到目前为止的中间代码生成,都忽略了可能有的数据对象的类型转换操作。考虑上述赋值语句的文法,假定有整数和实数两个类型,必要时整数可转为实数。这里直接使用在 5.3 节已经熟悉的属性 *E.type*,它的值是 *real* 或 *integer*,并且还忽略类型错误的检查。

$E \rightarrow E + E$ 和大多数其他产生式的语义动作都可以修改成必要时产生 x := inttoreal y 的三地址语句,它的作用是把整数 y 转换成值相等的实数,再赋给 x,还必须给算符一个标记,用以标明这是定点还是浮点算术运算。产生式 $E \rightarrow E_1 + E_2$ 的完整语义动作列在图 7.10。

```
E.place := newtemp;
if E1.type = integer and E2.type = integer then begin
    emit(E.place, ':=', E1.place, 'int+', E2.place);
    E.type := integer
end
else if E1.type = real and E2.type = real then begin
    emit(E.place, ':=', E1.place, 'real+', E2.place);
    E.type := real
end
else if E1.type = integer and E2.type = real then begin
    u := newtemp;
    emit(u, ':=', 'inttoreal', E1.place);
    emit(E.place, ':=', u, 'real+', E2.place);
    E.type := real
end
else if E1.type = real and E2.type = integer then begin
    u := newtemp;
    emit(u, ':=', 'inttoreal', E2.place);
    emit(E.place, ':=', E1.place, 'real+', u);
    E.type := real
end
else
    E.type := type_error;
```

图 7.10 $E \rightarrow E_1 + E_2$ 的语义动作

例如,假定 x 和 y 的类型是 real,i 和 j 的类型是 integer,对于输入

x := y+i*j

根据图 7.10,输出的三地址语句序列是

t_1 := i int* j
t_2 := inttoreal t_1
t_3 := y real+ t_2
x := t_3

图 7.10 的语义动作为非终结符 E 使用了两个属性 E.place 和 E.type。

7.4 布尔表达式和控制流语句

在编程语言中,布尔表达式有两个基本目的,它们用于计算逻辑值,但更经常的是在控制流语句中用作条件表达式,如 if-then、if-then-else 或者 while-do 语句。

布尔表达式也可以像算术表达式那样递归地定义。下面是本节所用的布尔表达式文法,其中 relop 是关系算符,为简单起见,它的两个运算对象都只允许是变量。

$E \rightarrow E$ or $E \mid E$ and $E \mid$ not $E \mid (E) \mid$ id relop id $\mid$ true $\mid$ false

按习惯,我们认为 or 和 and 都是左结合的,or 的优先级最低,然后是 and,最后是 not。

在有些情况下,只要完成了布尔表达式的部分计算就可以知道表达式的结果,那么表达式剩下的部分是否还要计算?编程语言的语义决定布尔表达式是否需要全部计算。Pascal 语言要求所有部分都必须计算,而 C 语言允许部分计算。如果语言定义允许(或要求)部分布尔表达式不计算,那么,编译器可以优化布尔表达式的计算,使之只计算足以确定它值的那部分表达式。我们把这样的计算称为"短路"计算。因此,在 E_1 or E_2 这样的表达式中,E_1 或 E_2 可能不必计算(或不必完全计算)。但是如果 E_1 或 E_2 是有副作用的表达式(即含改变非局部变量的函数),那就可能会出现与预期不一致的结果。为避免不同的优化有不同的结果,有些语言对短路计算做了严格的语义规定,把 E_1 or E_2 定义成

if E_1 then true else E_2

把 E_1 and E_2 定义成

if E_1 then E_2 else false

表示布尔表达式的值有两种主要方法。第一种方法是把真和假数值化,使布尔表达式的计算类似于算术表达式的计算,常常用 1 表示真,用 0 表示假。当然还有许多其他的编码方式,例如,可以用非 0 表示真,用 0 表示假,或者用非负数表示真,用负数表示假。

实现布尔表达式的第二种方法是用控制流,即用程序中的位置来表示布尔表达式的值,

它适用于短路计算的情况,用这种方式实现控制流语句中的布尔表达式尤其方便。因为对于控制流语句来说,只要能根据布尔表达式的情况从正确的位置继续执行就可以了,对它的值无须再关心。

7.4.1 布尔表达式的翻译

首先考虑用 1 和 0 分别表示真和假,并且布尔表达式将从左到右地完全计算。显然,这时布尔表达式的中间代码生成和算术表达式的没有多少区别。例如:

a or b and not c

翻译成的三地址语句序列是

t_1 := not c

t_2 := b and t_1

t_3 := a or t_2

a<b 这样的关系表达式等价于条件语句 if a<b then 1 else 0,它被翻译成的三地址语句序列是(因为涉及转移指令,所以需要给语句以编号,这里从 100 开始编号是随意的):

100 if a<b goto 103

101 t := 0

102 goto 104

103 t := 1

104:

为布尔表达式产生三地址代码的翻译方案见图 7.11。在这个方案中,变量 nextstat 给出输出序列中下一个三地址语句的序号,emit 在产生每个三地址语句后将 nextstat 加 1。我们给 relop 以属性 op,用来确定该关系算符究竟代表哪个关系运算。

$E \rightarrow E_1$ or E_2	{E.place := *newtemp*; *emit*(E.place, :=, E_1.place, or, E_2.place)}
$E \rightarrow E_1$ and E_2	{E.place := *newtemp*; *emit*(E.place, :=, E_1.place, and, E_2.place)}
$E \rightarrow$ not E_1	{E.place := *newtemp*; *emit*(E.place, :=, not, E_1.place)}
$E \rightarrow (E_1)$	{E.place := E_1.place}
$E \rightarrow id_1$ relop id_2	{E.place := *newtemp*; *emit*(‘if’, id_1.place, relop.op, id_2.place, ‘goto’, nextstat+3); *emit*(E.place, :=, ‘0’);

	emit(goto ,nextstat+2) ;
	emit(*E*.*place*, ≒ , 1)}
E→true	{E.place ≒ *newtemp*;
	emit(*E*.*place*, ≒ , 1)}
E→false	{E.place ≒ *newtemp*;
	emit(*E*.*place*, ≒ , 0)}

图 7.11 用数值表示布尔值的翻译方案

例 7.3 图 7.11 的翻译方案为表达式 a<b or c<d and e<f 产生的三地址代码如图 7.12 所示。

100 :if a<b goto 103	107 t_2 ≒ 1
101 t_1 ≒ 0	108 :if e<f goto 111
102 :goto 104	109 t_3 ≒ 0
103 t_1 ≒ 1	110 :goto 112
104 :if c<d goto 107	111 t_3 ≒ 1
105 t_2 ≒ 0	112 t_4 ≒ t_2 and t_3
106 :goto 108	113 t_5 ≒ t_1 or t_4

图 7.12 a<b or c<d and e<f 的翻译

7.4.2 控制流语句的翻译

现在考虑 if−then,if−then−else,while−do 和顺序语句到三地址语句的翻译,这些语句由下面的文法产生:

S→if E then S_1
| if E then S_1 else S_2
| while E do S_1
| S_1 ;S_2

在这些产生式中,E 是布尔表达式。图 7.13 用图形表示出这四个语句的三地址代码的结构。图中 E.code 和 S.code 分别表示 E 和 S 的三地址代码,它们在图中的次序就表示了它们在整个代码中的次序。例如在 if−then 的结构图中,E.code 在 S.code 的上面,它表示 E 的三地址代码在 S 的三地址代码前面。

图 7.13 中的 *E.true*,*E.false*,*S.begin* 和 *S.next* 都是三地址语句的标号,它们都是继承属性。图 (a) 为 if−then 的结构图,由于 S_1.*code* 究竟有多少个语句在翻译 E 时是不知道的,

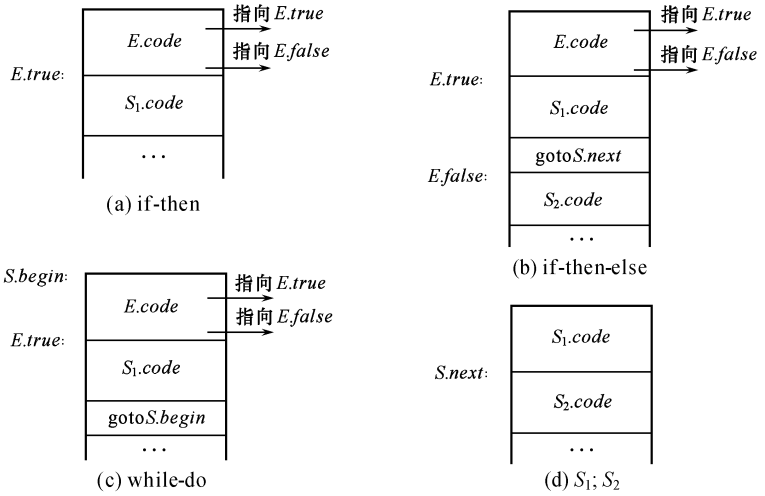


图 7.13 if－then,if－then－else,while－do 和顺序语句的代码

因此不能像上一小节那样,用 nextstat 加一个适当的常数来确定 S_1. *code 的第一个语句的编*号。在这里,给三地址语句以符号标号,函数 newlabel 每次调用时返加一个新的符号标号。

对布尔表达式 E,用两个标号 E.true 和 E.false 分别表示 E 为真和为假时控制流应该转向的标号,这两个属性由 E 的上下文决定。S.begin 是 S 第一个三地址语句的标号,S. *next 表示执行完 S 后应该执行的第一个三地址语句的标号,它们都由 S 的上下文决定。对*于 S_1 ;S_2 的情况,S_1 的 next 显然是 S_2 的第一条语句,如图 7.13 (c) 所示。对于其他语句,情况就不这么简单,见图 7.13 (b) 所示的 if－then－else 语句。在 S_1. *code 的后面有* goto *S. next,因为 S*1 执行结束意味着 S 执行结束,因此需要这个三地址语句,并且当 S_1 是赋值语句时肯定会执行这个语句。问题是 S_1 的 next 应该是什么,它可以是 goto *S. next 这个语句*的标号,但这意味着如果 S_1 中有三地址语句跳转到 S_1. *next 的话,那么紧接着再执行* goto *S. next。这种连续跳转显然应该避免并且可以避免,因此让 S*1. next 等于 S. next。*出于同样*的原因,S. next 标号不一定就是在紧挨着 S_2 的地方。

控制流语句的语法制导定义在表 7.3。和赋值语句的翻译方案不一样的是,用函数 gen 代替了过程 emit,gen 把形成的代码串作为函数值,而不是输出到文件上。还有,‖是作为串的连接算符。

表 7.3 没有给出布尔表达式的语法制导定义,7.4.1 节的翻译方案在此不适用,我们在下一小节另行给出。

表 7.3 控制流语句的语法制导定义

产生式	语义规则
$S \rightarrow$ if E then S_1	$E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.next;$ $S.code := E.code \parallel gen(E.true, ':') \parallel S_1.code$
$S \rightarrow$ if E then S_1 else S_2	$E.true := newlabel;$ $E.false := newlabel;$ $S_1.next := S.next;$ $S_2.next := S.next;$ $S.code := E.code \parallel gen(E.true, ':') \parallel S_1.code \parallel$ $gen('goto', S.next) \parallel gen(E.false, ':') \parallel S_2.code$
$S \rightarrow$ while E do S_1	$S.begin := newlabel;$ $E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.begin;$ $S.code := gen(S.begin, ':') \parallel E.code \parallel gen(E.true, ':') \parallel$ $S_1.code \parallel gen('goto', S.begin)$
$S \rightarrow S_1; S_2$	$S.code := S_1.code \parallel gen(S_1.next, ':') \parallel S_2.code$

7.4.3 布尔表达式的控制流翻译

现在讨论上一小节的 E.code,即为布尔表达式 E 产生中间代码。采用本节一开始提到的实现布尔表达式的第二种方式,即把 E 翻译成一串条件转移和无条件转移的三地址语句序列。

设计这个翻译的基本想法是这样,假定 E 是 a < b 的形式,*那么生成的代码形式为:*

if a < b goto *E.true*

goto *E.false*

设 E 是 E_1 or E_2 的形式。如果 E_1 为真,那么可以知道 E 本身为真,即 $E_1.true$ *和 E.true 一样。如果* E_1 为假,那么 E_2 必须计算,可以让 $E_1.false$ *是* E_2 代码的第一个语句的标号。E_2 的 true 和 false 分别与 E 的 true 和 false 一样。

类似的考虑可用于 E_1 and E_2 的翻译。形式为 not E_1 的表达式无需代码,只要交换 E 的 true 和 false 就得到 E_1 的 true 和 false。按这种方法为布尔表达式生成三地址代码的语法制导定义见表 7.4。

表 7.4 布尔表达式的语法制导定义

产生式	语义规则
$E \rightarrow E_1$ or E_2	$E_1.true := E.true$; $E_1.false := newlabel$; $E_2.true := E.true$; $E_2.false := E.false$; $E.code := E_1.code \parallel gen(E_1.false, ':') \parallel E_2.code$
$E \rightarrow E_1$ and E_2	$E_1.true := newlabel$; $E_1.false := E.false$; $E_2.true := E.true$; $E_2.false := E.false$; $E.code := E_1.code \parallel gen(E_1.true, ':') \parallel E_2.code$
$E \rightarrow$ not E_1	$E_1.true := E.false$; $E_1.false := E.true$; $E.code := E_1.code$
$E \rightarrow (E_1)$	$E_1.true := E.true$; $E_1.false := E.false$; $E.code := E_1.code$
$E \rightarrow id_1$ relop id_2	$E.code := gen('if', id_1.place, relop.op, id_2.place, 'goto', E.true) \parallel$ $gen('goto', E.false)$
$E \rightarrow$ true	$E.code := gen('goto', E.true)$
$E \rightarrow$ false	$E.code := gen('goto', E.false)$

例 7.4 再次考虑表达式

$$a < b \text{ or } c < d \text{ and } e < f$$

假定整个表达式的 true 和 false 已分别置为 L_{true} 和 L_{false}。那么用表 7.4 的定义可以得到下列代码：

```
    if a < b goto L_true
    goto L1
L1: if c < d goto L2
    goto L_false
L2: if e < f goto L_true
    goto L_false
```

生成的代码还不是最优的，因为删掉第二个语句不会改变代码的值。这种形式的冗余指令不难在以后的处理中删除。避免产生这些冗余转移的另一个办法是把形式为 $id_1 < id_2$

的关系表达式翻译成 if $id_1 > id_2$ goto *E.false*,即条件为真时执行正文上紧跟它的代码,而条件为假时跳转,简称为假转方式。

例 7.5 考虑语句

```
while a<b do
    if c<d then
        x := y+z
    else
        x := y-z
```

表 7.4 的语法制导定义,和赋值语句的翻译方案及控制流语句的语法制导定义合在一起,为该语句产生下列代码:

```
L1 : if a<b goto L2
     goto Lnext
L2 : if c<d goto L3
     goto L4
L3 : t1 := y+z
     x := t1
     goto L1
L4 : t2 := y-z
     x := t2
     goto L1
```

改变条件测试的方向可以删除前两个 goto 语句。

7.4.4 开关语句的翻译

在许多语言中都有“开关”语句或者“分情况”语句,但它们的语法和语义可能不同,开关语句的语法如下:

```
switch E
    begin
        case V1 : S1
        case V2 : S2
                ...
        case Vn-1 : Sn-1
        default : Sn
    end
```

这里有一个需要计算的开关表达式,后面有 n-1 个常量值,它们是该表达式可能取的

值。最后一个分支包含一个默认值,如果这 n−1 个常量值不能和该表达式匹配的话,那么该默认值总能匹配。匹配分支的执行结束就是该语句的执行结束。开关语句的执行流程是:

⑴ 计算开关表达式的值。

⑵ 分支测试,即在分情况表中找和该表达式值相同的常量值。如果没有这样的常量值,那么默认值和该表达式值匹配。

⑶ 执行相匹配分支的语句。

⑷ 跳转到该语句的后继语句。

步骤 ⑵ 的分支测试有多种实现方法。如果分支数不是很多,比方说少于 10 时,那么把开关语句翻译成下面的条件转移序列是合理的。

```
        t := E的代码
        if t≠ V1 goto L1
        S1 的代码
        goto next
L1 :    if t≠ V2 goto L2
        S2 的代码
        goto next
L2 :    ...
        ...
Ln-2 : if t≠ Vn-1 goto Ln-1
        Sn-1的代码
        goto next
Ln-1 : Sn 的代码
        next :
```

该方式的缺点是,很难对分支测试的代码进行特别处理。为提高效率,程序员应该把经常出现的分支放在前面。

另一种翻译方式是将分支测试的代码集中在一起,放在该语句代码的后部:

```
        t := E的代码
        goto test
L1 :    S1 的代码
        goto next
L2 :    S2 的代码
        goto next
        ...
```

```
L_{n-1}: S_{n-1}的代码
        goto next
L_n:    S_n 的代码
        goto next
test:   if t= V_1 goto L_1
        if t= V_2 goto L_2
        ...
        if t=V_{n-1} goto L_{n-1}
        goto L_n
        next:
```

把分支测试序列的代码放在代码的前部是不方便的,因为编译器不可能在看见每个 S_i 前就产生这样的代码。

为了便于识别从标号 test 开始的测试是开关语句的分支测试序列,以利于代码生成器对它进行特别处理,我们可以给中间代码增加一种 case 语句,将分支测试序列的中间代码改成下面的形式:

```
test: case V_1   L_1
      case V_2   L_2
      ...
      case V_{n-1}   L_{n-1}
      case t L_n
      next:
```

其中 case V L 和 if t= V goto L 的含义一样。

代码生成器实现这个测试序列的一种紧凑办法是建立一张二元组表,每个二元组由常量值和对应代码的入口地址组成。通过一个循环,将表达式的值和表中的各个值相比较,如果没有其他的值可匹配,最后的默认条目肯定匹配。

如果常量值的数目较多,那么用散列表效率会更高一些。

对一种经常出现的特殊情况,可以用更有效的办法实现 n 个分支。如果所有的常量值落在一个较小的区间,比方说 i_{min} 和 i_{max} 之间,并且 n 和 $i_{max}-i_{min}$ 的比值较大,那么可以构造标号数组,让与常量值 j 对应的标号放在偏移是 $j-i_{min}$ 的条目中,空白的条目全填上默认语句的标号。为完成分支测试和执行相匹配的语句,先计算开关表达式,获得它的值 m,检查 m 是否在 i_{min} 和 i_{max} 之间,若在其中,则间接转移到偏移为 $m-i_{min}$ 的条目所指的地址。

为把开关语句翻译成上面形式的代码,编译器看见保留字 switch 时,产生两个新的标号 test 和 next 以及新的临时变量 t,在分析表达式 E 时,产生计算 E 到 t 的代码,并产生跳转语句 goto test。然后为每个 case V_i :S_i 产生新建的标号 L_i,后面跟着 S_i 的代码,再后面是跳转

goto next。与此同时,将标号 L_i 加入符号表,并将这个符号表条目的指针和常量 V_i 放入专用于存储 V_i 和 L_i 对应关系的队列。当看见终止开关体的保留字 end 时,用这个队列中的信息产生分支测试的语句序列。

7.4.5 过程调用的翻译

第 6 章已详细介绍了过程调用的实现,包括存储空间的组织、过程调用序列和过程返回序列等,本小节将用文法

$S \rightarrow$ call id (Elist)

$Elist \rightarrow Elist, E$

$Elist \rightarrow E$

简单介绍过程调用语句的中间代码生成。

由前所述,过程调用的不同实现方式有不同的过程调用序列和返回序列,为了让中间代码能方便地用于不同的实现方式,我们特别为中间代码设计了一种 param 语句,专用于指示实在参数,就像开关语句的中间代码 case 指示分支测试一样。过程调用 id (E_1, E_2, ..., E_n) 的中间代码结构如下:

E_1.*place* ≒ E_1 的代码

E_2.*place* ≒ E_2 的代码

...

E_n.place ≒ E_n 的代码

param E_1.*place*

param E_2.*place*

...

param E_n.place

call id.place, n

根据上面的代码结构可知,在生成 E_i.*place* ≒ E_i 的代码后,需要保存 E_i.place 的值,队列是保存这些值的一种适当的数据结构。下面是采用这种数据结构的翻译方案:

$S \rightarrow$ call id (Elist) {为长度为 n 的队列中每个 E.place,执行 emit(param ,E.*place*) ;*emit*(call ,id.plase,n)}

$Elist \rightarrow Elist, E$

$Elist \rightarrow E$

在用 Yacc 等生成器来描述时,这个队列一般声明成一个全局的数据结构。

对于返回语句,产生它的中间代码是一件直截了当的事情。

习 题 7

7.1 把算术表达式 -(a+b) * (c+d) + (a+b+c) 翻译成：

(a) 语法树。

(b) 有向无环图。

(c) 后缀表示。

(d) 三地址代码。

7.2 把 C 程序

```
main()
 {
      int i;
      int a[10];
      while(i<=10)
        a[i]=0;
 }
```

的可执行语句翻译成：

(a) 语法树。

(b) 后缀表示。

(c) 三地址代码。

*7.3 证明 如果所有算符都是二元的，那么算符和运算对象的串是后缀表达式，当且仅当

(1) 算符个数正好比运算对象个数少一个；

(2) 在这个表达式的每个非空前缀中，算符数少于运算对象数。

7.4 修改图 7.4 中计算声明名字的类型和相对地址的翻译方案，允许名字表而不是单个名字出现在形式为 $D \rightarrow id:T$ 的声明中。

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

(a) 写出 *a*-(*b*+*c*) 的前缀形式。

*(b) 证明：所有的语义动作都是打印，并且所有的动作都出现在产生式右部末端的翻译方案不可能把中缀表达式翻译成前缀表达式。

(c) 给出把中缀表达式翻成前缀形式的语法制导定义。

7.6 编一个程序，实现表 7.4 的翻译布尔表达式到三地址代码的语法制导定义。

7.7 修改图 7.11 的语法制导定义，为栈机器产生代码。

7.8 表 7.4 的语法制导定义把 $E \rightarrow id_1 < id_2$ 翻译成一对语句

```
if id1 < id2 goto ...
goto ...
```

可以用一个语句

if id1 ≥ id2 goto ...

来代替,当 E 是真时执行后继代码。修改表 7.4 的语法制导定义,使之产生这种性质的代码。

7.9 下面的 C 语言程序

```
main()
 {
     int i,j;
     while((i||j)&&(j>5)) {
        i=j;
     }
}
```

在 X86/Linux 机器上编译生成的汇编代码如下:

```
        .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。

7.10 编一个程序,实现表 7.3 给出的控制流语句的语法制导定义。

7.11 用 7.3 节的翻译方案,把下列赋值语句翻译成三地址代码:

A[i,j] := B[i,j] + C[A[k,1]] + D[i+j]

7.12 C 语言的 for 语句有下列形式:

for (e1 ; e2 ; e3) stmt

它和

```
e1 ;
while (e2) do begin
    stmt ;
    e3
end
```

有同样的含义。构造一个语法制导定义,把 C 语言风格的 for 语句翻译成三地址代码。

7.13 Pascal 标准定义语句

for *v* := *initial* to *final* do *stmt*

和下面代码序列

```
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
```

有同样的含义

(a) 考虑下面的 Pascal 程序：

```
program forloop(input,output);
    var i,initial,final:integer;
    begin
        read(initial,final);
        for i ≔ initial to final do
            writeln(i)
    end.
```

当 initial=MAXINT−5 并且 *final*=MAXINT 时，该程序的行为是什么？其中 MAXINT 是目标机器的最大整数。

＊(b) 构造语法制导的定义，为 Pascal 的 for 语句产生正确的三地址代码。

7.14 对语句

```
for i ≔ 1 step 10−j until 10＊j do j ≔ j+1
```

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/1 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，都要执行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 语句属于这种情况，其表达式 e_2 和 e_3 是要重复计算的（见 7.12 题）。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。

第8章　代 码 生 成

编译器的最后一个阶段工作是代码生成,它取源程序的中间表示作为输入,产生等价的目标程序作为输出,如图８１所示。不管代码生成阶段的前面是否有代码优化阶段,本章提出的代码生成技术都是适用的。

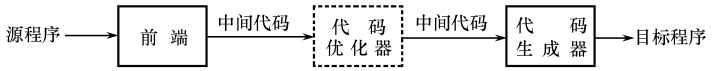


图 8.1　代码生成器的位置

对代码生成器的要求是严格的,首先它输出的目标代码必须是正确且高质量的,高质量的含义是目标代码应该有效地利用目标机器的资源,其次是代码生成器本身应该高效地运行。除此之外,易于实现、测试和维护也是重要的设计目标。

从理论上讲,产生最优代码问题是不可判定的,在实践中,有很多技术能够产生令人满意的好代码(虽不是最优的)。本章仅介绍一个简单的代码生成算法,以便让读者对代码生成技术有一个大致的了解。

8１　代码生成器设计中的问题

虽然代码生成器的具体细节依赖于目标机器和操作系统,但很多问题,如存储管理,指令选择,寄存器分配和计算次序选择是几乎所有的代码生成器都会碰到的问题。本节考察代码生成器设计中的一些公共问题,其中存储管理在第６章已经详细介绍。

代码生成器的输入包括由前端产生的中间表示和符号表信息,符号表信息用来决定中间表示中名字所代表的数据对象的运行地址。本章采用的中间表示是三地址代码。

8１１　目标程序

代码生成器的输出称为目标程序。像中间代码那样,目标程序的形式也可以多样:可执行目标模块(也称可执行目标程序)、可重定位目标模块或汇编语言模块。

可重定位目标模块是指代码装入内存的起始地址可以任意,代码中有一些重定位信息,

以适应重定位的要求。可执行目标模块装入内存的起始地址是固定的。

产生可执行目标模块作为输出的好处是,它可以放在内存的固定地方并且立即执行。这样,小程序可以迅速地编译和执行。历史上一些面向学生的编译器就产生可执行的目标程序。

采用可重定位的目标模块(也称可重定位的机器语言代码)时,需要连接器把一组可重定位目标模块连成一个可执行的目标程序。虽然产生可重定位目标模块必须增加额外的开销来进行连接,但带来的好处是灵活性。因为这种方式允许程序模块或子程序分别编译,允许从目标模块中调用其他先前编译好的程序模块。通常,可重定位目标模块中有重定位信息和连接信息,将在第10章详细介绍。

产生汇编语言程序作为输出使得代码生成的过程变得容易,因为可以产生符号指令并可利用汇编器的宏机制来帮助生成代码,所付出的代价是代码生成后的汇编工序。由于产生汇编码可免去编译器重复汇编器的工作,因此它也是一种合理的选择,尤其对于内存小而编译器必须分成几遍的情况来说更是这样。

为了可读性,本章用汇编代码作为目标语言。但需要强调的是,只要地址可以从符号表中的偏移和其他信息计算,那么产生名字的重定位地址或绝对地址,同产生它的符号地址一样容易。

8.1.2 指令选择

目标机器指令系统的性质决定了指令选择的难易程度,指令系统的统一性和完备性是重要的因素。例如,如果目标机器不能以统一的方式支持各种数据类型,那么每种例外的数据类型都需专门的处理。

指令的速度和机器特点是另一些重要的因素。如果不考虑目标程序的效率,指令的选择是直截了当的。对每一类三地址语句,可以设计所生成的目标代码的框架。例如,形式为 x ≒ y+z 的三地址语句,若 x,y 和 z 都是静态分配,那么它可以翻译成代码序列:

```
MOV  y,  R0  /*把y装入寄存器R0*/
ADD  z,  R0  /*z加到R0上*/
MOV  R0, x   /*把R0存入x中*/
```

遗憾的是,这种逐个语句地产生代码的方式常常得到质量低劣的代码。例如语句序列

```
a ≒ b+c
d ≒ a+e
```

将翻译成

```
MOV  b,  R0
ADD  c,  R0
```

```
MOV  R0, a
MOV  a,  R0
ADD  e,  R0
MOV  R0, d
```

显然,第四条指令多余。如果能够知道a以后不再使用的话,那么第三条也多余。

产生的代码质量取决于它的长度和执行速度。指令系统丰富的目标机器可能提供几种方式实现某一操作,由于不同实现方式的执行代价可能大不一样,因此对中间代码进行简单的翻译能产生正确的但效率可能难以接受的目标代码。例如,若目标机器有加1指令(INC),那么三地址语句a := a+1的高效实现是一条指令INC a,而不是下面的指令序列:

```
MOV  a,  R0
ADD  #1,R0
MOV  R0, a
```

指令的速度对设计好的代码序列是必需的,但是,精确的时间信息常常很难得到。决定哪个指令序列对给定的三地址结构是最优的,可能还要用到该结构出现的上下文知识。

8.1.3 寄存器分配

运算对象处于寄存器中的指令通常比运算对象处于内存的指令要短一些,执行也快一些。因此,充分利用寄存器对生成高质量的代码尤其重要。寄存器的使用可以分成两个子问题:

(1) 在寄存器分配期间,为程序的有关点选择驻留在寄存器中的一组变量。

(2) 在随后的寄存器指派阶段,挑选变量要驻留的具体寄存器。

选择最优的寄存器指派方案是困难的,它是NP完全问题。这个问题还会进一步复杂,因为目标机器的硬件和/或操作系统可能要求寄存器的使用遵守一些约定。

本章介绍的简单代码生成算法将寄存器分配和指派合在一起,统称为寄存器分配。

8.1.4 计算次序选择

计算的执行次序会影响目标代码的效率。例如,对一个表达式,某个计算次序可能会比其他次序需要较少的寄存器来保存中间结果。选择最佳计算次序也是一个NP完全问题。本章只讨论按照中间代码生成器产生的三地址语句的次序来产生目标代码,但是在习题中将给出一些体现计算次序选择的例子。

8.2 目标机器

熟悉目标机器和它的指令系统是设计好一个代码生成器的先决条件。遗憾的是,在代码生成的一般性讨论中,不能对目标机器描述到足够详细的程度,因而难以对一个完整的语言产生高效的代码。本章选择可作为几种微机代表的寄存器机器作为我们的目标计算机,所提出的一些技术可用于其他许多类机器。

8.2.1 目标机器的指令系统

所选的目标机器是字节寻址机器,4个字节组成1个字,有 n 个通用寄存器 R0, R1, ..., R(n-1)。它有形式为

op 源, 目的

的二地址指令,其中op是操作码,源和目的都是数据域。该机有如下的操作码:

MOV (源传到目的)
ADD (源加到目的)
SUB (目的减去源)

其他的指令等用到时再介绍。

由于源和目的这两个域没有长到足以保存内存地址,所以它们的某些位用来指明指令的下一个字或两个字包含运算对象或地址。于是,一条指令的源和目的由寄存器和内存单元与地址模式组合起来指明。在下面的描述中,*contents*(*a*) 表示由 *a* 代表的寄存器或内存单元的内容。

地址模式和它们的汇编语言形式及附加代价如下:

模式	形式	地址	附加代价
绝对地址	M	M	1
寄存器	R	R	0
变址	*c*(R)	*c*+*contents*(R)	1
间接寄存器	*R	*contents*(R)	0
间接变址	**c*(R)	*contents*(*c*+*contents*(R))	1

在下一小节将解释附加代价。

当用作源或目的时,内存单元M和寄存器R都代表本身。例如,指令

MOV R0, M

把寄存器 R0 的内容存入内存单元 M。

对寄存器 R 的值偏离 c 表示的一个地址写成 c(R)。例如,指令

MOV 4(R0), M

把值

contents(4+*contents*(R0))

存入内存单元 M。

后面两个间接模式由前缀*来指明。例如,指令

MOV *4(R0), M

把值

contents(*contents*(4+*contents*(R0)))

存入内存单元 M。

最后一个地址模式允许源是常数:

模式	形式	常数	附加代价
直接量	#*c*	*c*	1

例如,指令

MOV #1, R0

把 1 保存于寄存器 R0 中。

8.2.2 指令的代价

把指令代价取成 1,加上它的源和目的地址模式的附加代价(8.2.1 节地址模式表的最后一列),这个代价对应指令的长度(以字计算)。寄存器地址模式的代价是 0,而那些含内存单元或常数的地址模式的代价是 1,因为这样的运算对象必须和指令存放在一起。

如果空间是至关重要的,应该使指令的长度尽可能短。这样做有一个额外的重要好处,对大多数机器和大多数指令来说,从内存取指令的时间超过执行指令的时间,因而极小化指令的长度也使得指令的执行时间趋于最小。下面是几个例子:

(1) 指令 MOV R0,R1 把寄存器 R0 的内容复写到寄存器 R1,这条指令的代价是 1,因为它仅占 1 个字的内存。

(2) 指令 MOV R5,M 把寄存器 R5 的内容复写到内存单元 M。这条指令的代价是 2,因为内存单元的地址 M 存于指令的第 2 个字中。

(3) 指令 ADD #1,R3 把常数 1 加到寄存器 R3 的内容上。这条指令的代价是 2,因为常数 1 出现在指令的第 2 个字中。

(4) 指令 SUB 4(R0),*12(R1) 把值

contents(*contents*(12+ *contents*(R1)))－ *contents*(4+ *contents*(R0))

存入目的地址＊12(R1)。这条指令的代价是3,因为常量4和12存于指令的第2和第3个字中。

为这种机器产生代码的困难可以从下面的例子看出一些。考虑为形式是a := b+c的三址语句产生代码,其中a、b和c都是简单变量,静态分配内存单元,我们用它们的名字表示相应的单元。这个语句可以由许多不同的指令序列来实现,例如:

(1) MOV b,R0
ADD c,R0 代价=6
MOV R0,a

(2) MOV b,a
ADD c,a 代价=6

若R0,R1和R2分别含a,b和c的地址,则可以使用:

(3) MOV＊R1,＊R0
ADD＊R2,＊R0 代价=2

若R1和R2分别含b和c的值,并且b的值在这个赋值后不再需要,则可以使用:

(4) ADD R2,R1
MOV R1,a 代价=3

可以看出,为了替这个语句产生好的代码,必须有效地使用它的寻址能力。可能的话,尽量把名字的左值或右值保存在寄存器中,以便在不久的将来使用。

例8.1 图8.2是C语言赋值表达式a[i]=b+1的中间代码树,其中a[i],i和b都是long类型,并且a和i都是动态栈式分配的,b是静态分配的。为简单起见,假定机器是按字编址,并且long类型的值占1个字。和上一章的中间代码树不同的是,这种形式的中间代码树给代码生成提供了丰富的信息,它已体现了动态栈式分配变量的地址描述和左右值的区别。

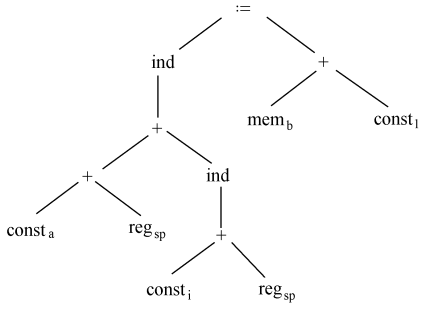


图8.2 赋值表达式 *a*[*i*] = *b*+1的中间代码树

该图中结点标记的解释如下:

(1) reg$_{sp}$表示栈的基址寄存器SP;

(2) 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;

(3) mem表示内存地址,mem$_b$表示分配给b的内存地址;

(4) 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_{+} 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。

用本节介绍的各种指令和一种叫做树重写的代码生成算法(本章不介绍),可为该中间代码树生成如下的代码序列:

```
MOV #a,R0
ADD SP,R0
ADD i(SP),R0
MOV b,R1
INC R1
MOV R1,*R0
```

8.3 基本块和流图

三地址语句序列的一种图形表示叫做流图。流图的结点代表一个顺序计算序列,边代表控制流。即使代码生成算法不明显构造流图,流图对于理解代码生成算法也是有用的。下一章将充分利用流图作为从中间代码收集信息的媒介。

8.3.1 基本块

基本块是一连续的语句序列,控制流从它的开始进入,并从它的末尾离开,没有停止或分支的可能性(末尾除外)。下面三地址语句序列形成一个基本块:

$$
\begin{aligned}
&t_1 := a*a\\
&t_2 := a*b\\
&t_3 := 2*t_2\\
&t_4 := t_1+t_3\\
&t_5 := b*b\\
&t_6 := t_4+t_5
\end{aligned}
\tag{8.1}
$$

我们说三地址语句 x := y+z 引用 y 和 z 并对 x 定值。如果一个名字的值在基本块的某

一点以后还要引用的话(包括在后继基本块的引用),则说这个名字在该点是活跃的。

下面的算法可用于把三地址语句序列分成基本块。

算法 8.1 划分基本块

输入 三地址语句序列。

输出 基本块列表,每个三地址语句仅在一个基本块中。

方法 (1) 首先确定所有的入口语句(基本块的第一个语句)。规则如下:

(a) 序列的第一个语句是入口语句。

(b) 能由条件转移语句或无条件转移语句转到的语句是入口语句。

(c) 紧跟在条件转移语句或无条件转移语句后面的语句是入口语句。

(2) 对于每个入口语句,它所在的基本块由它开始直到下一个入口语句(但不含该入口语句)或程序结束为止的所有语句组成。

例 8.2 考虑图 8.3 的源代码段,它计算两个长度为 20 的向量 *a* 和 *b* 的点积。在目标机器上完成这个计算的三地址语句序列见图 8.4。

```
begin
    prod := 0;
    i := 1;
    do begin
        prod := prod + a[i] * b[i];
        i := i+1
    end while i <= 20
end
```

图 8.3 计算点积的程序

```
(1)  prod := 0
(2)  i := 1
(3)  t1 := 4*i
(4)  t2 := a[t1]        /*计算 a[i]*/
(5)  t3 := 4*i
(6)  t4 := b[t3]        /*计算 b[i]*/
(7)  t5 := t2*t4
(8)  t6 := prod+t5
(9)  prod := t6
(10) t7 := i+1
(11) i := t7
(12) if i<=20 goto (3)
```

图 8.4 计算点积的三地址代码

把算法 8.1 作用到图 8.4 的三地址代码上来决定它的基本块。由规则(a),语句(1)是入口语句。由规则(b),语句(3)是入口语句,因为最后一个语句可以转到它。由规则(c),若语句(12)后面还有语句,则跟随语句(12)的语句是入口语句。这样,语句(1)和(2)构成一个基本块,其余的语句形成一个基本块。

8.3.2 基本块的变换

一个基本块计算一组表达式,这些表达式是在该基本块出口活跃的名字的值。如果两个基本块计算一组同样的表达式并且这些表达式的值分别代表同样的活跃名字的值,则它们是等价的。

有很多等价变换可用于基本块,这些变换对改进代码的质量是有用的。下一章将阐述全局代码优化怎样使用这样的变换来重新安排程序的计算次序,以缩减最终目标程序运行所需的时间或空间。这里介绍两类可用于基本块的局部变换,它们是保结构变换和代数变换。先假定基本块没有数组、指针和过程调用。

先介绍三种保结构变换。

⑴ 删除局部公共子表达式

考虑基本块:

$$a \doteq b+c$$
$$b \doteq a-d$$
$$c \doteq b+c$$
$$d \doteq a-d$$

第二个语句和第四个语句计算同样的表达式,即 b+c-d,因此该基本块可以变换成等价的基本块:

$$a \doteq b+c$$
$$b \doteq a-d$$
$$c \doteq b+c$$
$$d \doteq b$$

虽然第一个语句和第三个语句有同样的表达式出现在右部,但由于第二个语句重新定义 b,使得第一个语句和第三个语句中的 b 有不同的值,所以它们计算的是不相同的表达式。

⑵ 删除死代码

一个基本块的语句 $x \doteq y+z$ 给 x 定值,若 x 以后不再被引用,则称 x 为死变量。删除这样的语句是基本块的一种等价变换。

⑶ 交换相邻的独立语句

如果基本块有两个相邻的语句:

$$t_1 \doteq b+c$$
$$t_2 \doteq x+y$$

交换这两个语句而不影响基本块计算的表达式,当且仅当 x 和 y 都不是 t_1,b 和 c 都不

是 t_2。

下面讨论代数变换。有许多代数变换可用于把基本块计算的表达式集合变换成代数等价的表达式集合,其中有价值的是那些可以简化表达式或用较快运算代替较慢运算的变换。例如,像

$$x := x+0$$

或

$$x := x*1$$

这样的语句可以从基本块删除,这是基本块的一种等价变换。语句

$$x := y**2$$

的指数运算通常需要用函数调用实现。使用代数变换,该语句可以由快速、等价的语句

$$x := y*y$$

代替。

8.3.3 流图

可以把控制流信息加到基本块集合,形成一个有向图来表示程序,这样的有向图称为流图。流图的结点是基本块(简称块),有一个特殊的结点称为初始结点,它的入口语句是程序的第一个语句。如果在程序的某个执行序列中块 B_2 跟随块 B_1,那么块 B_1 到块 B_2 有一条有向边。也就是,如果:

(1) 从块 B_1 的最后一个语句有条件转移或无条件转移到块 B_2 的第一个语句,或者

(2) 按程序正文的次序块 B_2 直接跟随块 B_1,并且块 B_1 不是结束于无条件转移

那么块 B_1 到块 B_2 有一条有向边。我们说块 B_1 是块 B_2 的前驱,块 B_2 是块 B_1 的后继。

例8.3 图8.4程序的流图见图8.5。块 B_1 是初始结点。注意,最后一个语句,原来是条件转移到语句(3),现已由等价的转到 B_2 块开始的语句代替。

在流图中,什么是循环?怎样找出所有的循环?大多数时候,这些问题是容易回答的。例如,图8.5存在一个循环,它由块 B_2 组成。然而对更一般情况的回答是有点难以捉摸的,下一章将详细讨论它们。目前,只要知道循环是流图中满足下面条件的一簇结点就行了:

(1) 簇中所有结点是强连通的。即从循环中任一

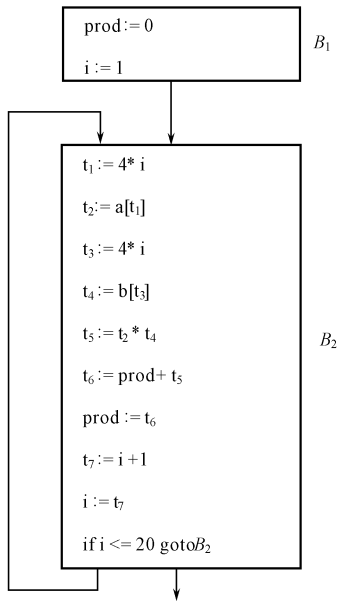


图8.5 图8.4程序的流图

个结点到另一个结点都有一条路径,路径上的所有结点都在这簇结点中。

(2) 这种结点簇有惟一的入口。从循环外的结点到达循环中任一个结点的惟一方式是首先通过入口。

不包含其他循环的循环叫做内循环。

8.3.4 下次引用信息

下面介绍如何收集基本块中名字的下次引用(next-use)信息,8.4节的简单代码生成器需要这样的信息来完成寄存器分配。这是因为,如果知道存于某个寄存器的一个名字的值以后不再需要了,那么该寄存器可以存放别的名字的值。

三地址语句中名字的引用定义如下:假定三地址语句 i 把 a 的值赋给 x,如果语句 j 用 x 作为运算对象,控制可以从 i 流到 j,并且这条路径当中没有 x 的其他赋值,那么称 j 引用 x 在 i 定的值。

我们希望为每个三地址语句 $x := y\ op\ z$ 决定 x、y 和 z 的下次引用信息。目前可不考虑它们在基本块外的引用情况,若需要这种信息,可用第9章的活跃变量分析技术获得。

决定下次引用信息的算法对每个基本块从最后一个语句反向扫描到第一个语句。在反向扫描一个基本块前,根据下一章的活跃变量分析,在符号表中把在出口活跃的所有变量都置上活跃标记,并置上没有下次引用(指在本块中没有下次引用)。而对于在出口不活跃的变量,都置上不活跃也没有下次引用。

假如反向扫描到达三地语句 $x := y\ op\ z$,*执行下面几步:*

(1) 从符号表中找到 x、y 和 z 的下次引用信息和活跃标记,并把它们加到语句 i 上。(如果 x 不活跃,可以删掉这个语句。)

(2) 在符号表中把 x 置成不活跃和没有下次引用。

(3) 在符号表中,置 y 和 z 活跃,并且置它们的下次引用信息为 i。

注意,(2)和(3)的次序不能颠倒,因为 x 可能就是 y 或 z。

如果三地址语句 *i* 是 $x := y$ 或 $x := op\ y$ 的形式,其步骤同上,但忽略 *z*。

利用下次引用信息,可以压缩临时变量需要的空间。一般地,如果两个临时变量的生存期不重叠的话,可以把它们压缩在同一单元中。因为几乎所有的临时变量都引用和定义在同一块中,因而下次引用信息就可用来紧缩临时变量。对于那些有穿越块的引用的临时变量,将在第9章数据流分析时讨论。

临时变量存储单元的分配可以这样进行:依次检查临时变量区域的单元,找到第一个不含活跃临时变量的单元,把它分配给待分配的临时变量;如果没有这样的单元,则在活动记录的临时变量区域加一个单元。在许多情况下,临时变量可以压缩到只使用寄存器而不需要内存单元,见8.4节的讨论。

例如,基本块(8.1)有6个临时变量,可压缩为只需要两个存储单元,它们是t_1和t_2。

$t_1 := a*a$

$t_2 := a*b$

$t_2 := 2*t_2$

$t_1 := t_1+t_2$

$t_2 := b*b$

$t_1 := t_1+t_2$

8.4 一个简单的代码生成器

本节的代码生成策略为三地址语句序列产生目标代码。它依次考虑每个语句,根据寄存器当前的使用情况,为其产生代码,并根据产生的代码修改寄存器的使用情况。

为简单起见,假定三地址语句的每种算符都有对应的目标机器算符,并且计算结果留在寄存器中尽可能长的时间,只有在下面两种情况下才把它存入内存:

(1) 如果此寄存器要用于其他计算;

(2) 正好在转移或标号语句之前。

条件(2)暗示在基本块的结尾,所有东西都必须存起来。必须这样做的原因是,离开一个基本块后,可能进入几个不同基本块中的一个,或者进入一个还可以从其他基本块进入的基本块。在这两种情况下,没有经过额外的努力,就认为基本块引用的某个数据在入口点一定处于某个寄存器中是不妥的。因此,为避免可能出现的错误,这个简单的代码生成算法在离开基本块时,存储所有的东西。

8.4.1 寄存器描述和地址描述

对三地址语句a := b+c,如果寄存器Ri含b,Rj含c,且b在此语句后不再活跃,即b不再引用,那么可以为它产生代价为1的代码ADD Rj,Ri,结果a在Ri中。如果Ri含b,但c在内存单元(为方便起见,仍叫做c),b仍然不再活跃,那么可以产生代价为2的代码

```
ADD c,Ri
```

或代价为3的代码序列

```
MOV c,Rj
ADD Rj,Ri
```

如果c的值以后还要用的话,第二种代码比较有吸引力,因为可以从寄存器Rj中取c

的值。还有很多的情况可考虑,取决于b和c当前在什么地方和b的值以后是否还要用。还必须考虑b和c的一个或两个都是常数的情况。如果+运算是可交换的话,考虑的情况还会增加。所以,可以看出,代码生成包含了对大量情况的考察,哪种情况占优势依赖于三地址语句出现的上下文。

从上面的例子可以看出,在代码生成过程中,需要跟踪寄存器的内容和名字的地址。本节的代码生成算法使用寄存器和名字的描述来跟踪寄存器的内容和名字的地址。

⑴ 寄存器描述记住每个寄存器当前存的是什么。假定初始时寄存器描述显示所有寄存器为空(如果寄存器分配穿越块边界,当然就不是这样简单了)。随着对基本块的代码生成逐步前进,在任何一点,每个寄存器保存若干个(包括零个)名字的值。寄存器的这些信息可以单独用一张寄存器表来描述。

⑵ 名字的地址描述记住运行时每个名字的当前值可以在哪个场所找到。这个场所可以是寄存器、栈单元、内存地址,甚至是它们的某个集合,因为复写时值仍然留在原来的地方。这些信息可以存于符号表中,在决定名字的访问方式时使用。

8.4.2 代码生成算法

代码生成算法取构成一个基本块的三地址语句序列作为输入,对每个三地址语句 $x := y\ op\ z$ 完成下列动作:

⑴ 调用函数 *getreg* 决定放 $y\ op\ z$ 计算结果的场所 L。L 通常是寄存器,也可能是内存单元。我们将简要描述 *getreg* 的算法。

⑵ 查看 y 的地址描述,确定 y 值当前的一个场所 y′。如果 y 当前值既在内存单元又在寄存器中,当然选择寄存器作为 y′,特别是 y 的值所在的寄存器正好是 L 的时候。如果 y 的值还不在 L 中,则产生指令 MOV y′,L,把 y 的值复写到 L 中。

⑶ 产生指令 op z′,L,其中 z′是 z 的当前场所之一。同上面一样,如果 z 值既在寄存器又在内存单元,就优先于前者。修改 x 的地址描述,以表示 x 在场所 L,如果 L 是寄存器,修改它的描述,以表示它含 x 的值。

⑷ 如果 y 和/或 z 的当前值不再引用,在块的出口也不活跃,并且还在寄存器中,那么修改寄存器描述,以表示在执行了 $x := y\ op\ z$ 以后,这些寄存器分别不再含 y 和/或 z 的值。

如果当前的三地址语句有一元算符,步骤同上面的类似,略去这些细节。一个重要的特例是三地址语句 $x := y$。如果 y 在寄存器中,只要改变寄存器和地址描述,记住 x 的值现在只能在存 y 值的寄存器中找到。如果 y 不再引用,并且在基本块出口不活跃,那么这个寄存器不再保存 y 的值。如果 y 的值仅在内存,原则上可以记下 x 的值在 y 的内存单元,但是这样会使算法复杂,因为以后若要改变 y 的值时必须先保存 x 的值。所以,如果 y 在内存,可用 *getreg* 来找到一个存放 y 的寄存器,并记住此寄存器是存 x 的场所。另一种办法是产生

指令 MOV y,x。尤其是 x 在块中不再引用时,这样做比较好。值得注意的是,如果用第 9 章的各种优化,尤其是复写传播算法,大多数(如果不是所有的)复写指令可以删去。

一旦处理完基本块的所有三地址语句,在基本块出口,用 MOV 指令把那些值尚不在它们内存单元的活跃名字的值存入它们的内存单元。为完成这一点,用寄存器描述来决定什么名字仍在寄存器中,用地址描述来决定这些名字的值是否不在它们的内存单元,用活跃变量信息来决定这些名字是否要存储起来。如果基本块之间的数据流分析没有计算活跃变量信息,只好认为所有用户定义的名字在基本块末尾都是活跃的。

8.4.3 寄存器选择函数

函数 *getreg* 返回保存语句 $x := y\ op\ z$ 的 x 值的场所 L。该代码生成算法的很多努力都消耗在实现这个函数上,以产生对 L 的较好选择。本小节讨论基于下次引用信息的一个简单易行的办法。

(1) 如果名字 y 在寄存器中,此寄存器不含其他名字的值(注意,别忘了 $x := y$ 这样的复写语句会使得寄存器同时保存两个或更多变量的值),并且在执行 $x := y\ op\ z$ 后 y 不再有下次引用,那么返回 y 的这个寄存器作为 L。

(2) (1)失败时,返回一个空闲寄存器,如果有的话。

(3) 当(2)不能成功时,如果 x 在块中有下次引用,或者 op 是必须用寄存器的算符,如变址,那么找一个已被占用的寄存器 R。如果 R 的值还没有保存到它应该在的内存单元 M,由 MOV R,M 把 R 的值存入内存单元 M,修改 M 的地址描述,返回 R。如果 R 保存着几个变量的值,那么对于每个需要存储的变量都产生 MOV 指令。怎样恰当地选择这个寄存器呢?可优先选择其数据在最远的将来使用,或者其数据同时在内存的寄存器。我们难以描述精确的选择方法,因为没有人能证明哪种选择方法是最佳的。

(4) 如果 x 在本基本块中不再引用,或者找不到适当的被占用寄存器,可选择 x 的内存单元作为 L。

更复杂的 *getreg* 函数在决定存放 x 值的寄存器时要考虑 x 随后使用情况和算符 op 的交换性。

例 8.4 赋值语句 $d := (a-b) + (a-c) + (a-c)$ 可以翻译成下面的三地址语句序列:

$$t_1 := a-b$$
$$t_2 := a-c$$
$$t_3 := t_1+t_2$$
$$d := t_3+t_2$$

假定只有 d 在基本块出口活跃。上面的代码生成算法为这个三地址语句序列产生如表 8.1

的代码序列。表中给出代码生成过程中相关的寄存器描述和地址描述,但是忽略了a,b和c的值总是在内存中这样一个事实。同时还假定t_1,t_2和t_3是临时变量,它们的值都不在内存,除非用MOV指令把它们存起来。

表8.1 目标代码序列

语句	生成的代码	寄存器描述	名字地址描述
		寄存器空	
$t_1 := a-b$	MOV a,R0 SUB b,R0	R0含t_1	t_1在R0中
$t_2 := a-c$	MOV a,R1 SUB c,R1	R0含t_1 R1含t_2	t_1在R0中 t_2在R1中
$t_3 := t_1+t_2$	ADD R1,R0	R0含t_3 R1含t_3	t_3在R0中 t_3在R1中
$d := t_3+t_3$	ADD R1,R0	R0含d	d在R0中
	MOV R0,d		d在R0和内存中

*getreg*的第一次调用返回R0作为计算t_1的场所。因为a不在R0,因此产生MOV a,R0和SUB b,R0的指令。修改寄存器描述表示R0含t_1。

代码生成以这种方式前进,直到最后一个三地址语句处理完。注意,这时R1为空,因为t_2不再引用。最后在基本块的结尾产生MOV R0,d,存储活跃变量d。

表8.1生成的代码的代价是12。可以把它缩减到11,在第一条指令后立即产生指令MOV R0,R1,删去指令MOV a,R1,但是这需要更复杂的代码生成算法。代价能减小的原因是从R1取到R0比从内存取到R0要低廉一些。

8.4.4 为变址和指针语句产生代码

变址与指针运算的三地址语句的处理和二元算符的处理相同。表8.2给出了为变址语句a := b[i]和a[i] := b产生的代码序列,假定b是静态分配的。

表8.2 变址语句的代码序列

语句	i在寄存器Ri中		i在内存Mi中		i在栈中	
	代码	代价	代码	代价	代码	代价
a := b[i]	MOV b(Ri),R	2	MOV Mi,R MOV b(R),R	4	MOV Si(A),R MOV b(R),R	4
a[i] := b	MOV b,a(Ri)	3	MOV Mi,R MOV b,a(R)	5	MOV Si(A),R MOV b,a(R)	5

i当前所在的场所决定代码序列。表中给出三种情况,分别是i在寄存器Ri中,i在内存单元Mi,还有i在栈中。对于后者,偏移为Si,且i所在的活动记录指针是寄存器A。寄存器R是调用函数getreg时返回的寄存器,对于第一个赋值,如果a在块中有下次引用,并且寄存器R是可用的,宁愿把a留在寄存器R中,对第二个语句还假定a是静态分配的。

表8.3给出了为指针语句a ≒ *p和*p ≒ a产生的代码序列。这里,p的当前位置决定代码序列。

表8.3 指针语句的代码序列

语句	p在寄存器Rp中		p在内存Mp中		p在栈中	
	代码	代价	代码	代价	代码	代价
a ≒ *p	MOV *Rp,a	2	MOV Mp,R MOV *R,R	3	MOV Sp(A),R MOV *R,R	3
*p ≒ a	MOV a,*Rp	2	MOV Mp,R MOV a,*R	4	MOV a,R MOV R,*Sp(A)	4

同上面一样,这里也给出了三种情况。寄存器R也是调用函数*getreg*返回的寄存器。第二个语句也假定a静态分配。

8.4.5 条件语句

机器实现条件转移有两种方式。一种方式是根据寄存器的值是否为下面6个条件之一进行分支:负、零、正、非负、非零和非正。在这样的机器上,像if x<y goto z这样的三地址语句可以这样实现:把x减y的值存入寄存器R,如果R的值为负,则跳到z。

第二种方式是用条件码来表示计算的结果或装入寄存器的值是负、零还是正。这种方法适用于大多数机器。通常,比较指令(在我们的机器上是CMP)有这样的性质,它设置条件码而不真正计算值。即,若x>y,那么CMP x,y置条件码为正,等等。条件转移指令根据指定的条件<,=,>,≤,≠或≥是否满足来决定是否转移。用指令CJ<=z表示如果条件码是负或者零则转到z。例如,if x<y goto z可以由

```
CMP    x, y
CJ<    z
```

来实现。

产生代码时,记住条件码的描述是有用的。这个描述告诉我们设置当前条件码的名字或比较的名字对。于是可以用

```
MOV    y, R0
```

```
ADD    z,  R0
MOV    R0, x
CJ<    z
```

来实现

```
x := y+z
if x<0 goto z
```

因为根据条件码描述可以知道在 ADD z,R0 之后,条件码是由 x 设置的。

习 题 8

8.1 为下列 C 语句产生 8.2 节目标机器的代码,假定所有的变量都是静态的,并假定有 3 个寄存器可用。

(a) x=1

(b) x=y

(c) x=x+1

(d) x=a+b*c

(e) x=a/(b+c)-d*(e+f)

8.2 重复习题 8.1,假定所有的变量都是自动的(分配在栈上)。

8.3 为下列 C 语句产生 8.2 节目标机器的代码,假定所有的变量都是静态的,并假定有 3 个寄存器可用。

(a) x=a[i]+1

(b) a[i]=b[c[i]]

(c) a[i]=a[i]+b[j]

8.4 使用 8.4 节的算法重做习题 8.1。

8.5 在 SPARC/SUNOS 上,经某编译器编译,下面程序的运行结果是 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样不同的结果。

```
int fact()
{
    static int i=5;

    if(i==0) {
        return(1);
    }
    else {
```

```
            i=i-1;
            return ((i+abs(1))*fact(i));
        }
    }

    main()
    {
        printf("factor of 5=%d\n",fact(5));
    }
```

8.6 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机器上编译后的运行结果是7,而在SPARC/SUNOS机器上编译后的运行结果是6。试分析运行结果不同的原因。

8.7 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机器上编译后生成的汇编代码如下,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,你能猜测出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)"
```

8.8 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 机器上编译后生成的汇编代码如下,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)"
```

8.9 一个C语言程序如下：

```
extern int a;
static int b;
int c;

main()
 {
    b=a;
}
```

它在 X86/Linux 机器上编译后生成的汇编代码如下,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)
```

*第9章　代码优化

由简单的编译算法产生的代码,经改进后可以运行得更快,或空间占得更少,或两者兼而有之。这种改进是通过程序变换来获得的,这样的程序变换称作优化。实施代码改进变换的编译器叫做优化编译器。

本章介绍独立于机器的优化,即不考虑任何目标机器性质的优化变换。依赖于机器的优化,例如寄存器分配,我们已在第8章讨论过了。

一般而言,程序的内循环(特别是最内循环)是重点要改进的地方,因为它们往往是程序中经常执行的部分。让这些部分尽可能地高效率,有可能使我们以最小的代价获得最大的利益。当然,编译器只能靠自己来对程序的热点在哪儿作出最好的猜测,这种猜测并不一定完全符合实际情况。程序流图中的循环由控制流分析过程来识别,这是本章讨论的重点之一。

要完成优化还需要收集程序中变量使用方式的信息,这由数据流分析来完成。在程序的不同点收集的这种信息可以用一组简单的方程联系起来。本章给出一些用数据流分析收集信息和在优化中有效地使用这些信息的算法。

9.1　优化的主要种类

本节介绍一些最有用的代码改进变换,实现这些变换的技术在下面几节给出。考察一个基本块的语句就可以完成的变换叫做局部变换,否则叫做全局变换,通常先完成局部变换。

在介绍各种变换之前,先给出代码改进变换的标准,并给出一个C程序的源程序、中间代码和流图,以此为例来介绍各种优化。本书不讨论过程间的优化,这里所说的程序是指单个过程。

9.1.1　代码改进变换的标准

由优化编译器提供的变换应该有下列几点性质:

首先,代码变换必须保持程序的含义,也就是优化不能改变程序对给定输入的输出,也不能引起在源程序版本中不会出现的错误,如除数为零。对于优化,编译器采取稳妥的策略,即宁可失去某些优化的机会,也不能采用可能改变程序行为的变换。

其次,变换减少程序的运行时间平均达到一个可度量的值。即,并不是每种变换都能成

功地改进每一个程序，偶尔，优化可能稍稍增加了个别程序的运行时间，所以我们强调一种变换对各种程序的平均影响。优化有可能使运行时间有可观的缩短，但是没有一个编译器能为程序找到最好的算法。代码的空间已经不像以往那么重要了，但是有时我们的兴趣还在缩小目标代码所需空间。

第三，变换所做的努力是值得的。编译器的编写者为实现代码改进变换所消耗的时间和精力，以及优化编译器优化阶段的开销，如果不能从目标程序的运行中得到补偿，那么这种改进变换是没有意义的。有些变换，只有在对源程序进行详尽的、往往是费时的分析后才能使用，因此很少把它们用于只运行几次的程序。例如，快速的、非优化的编译器可能对调试或者对运行几次就要扔掉的“学生作业”更有帮助。

本章用一个快速排序程序 quicksort 来说明各种代码改进变换的作用，图 9.1 是这个程序的 C 代码。不讨论该程序的算法方面，事实上，为了这个程序能正常工作，a[0] 和 a[max] 应分别是被排序的最小元素和最大元素。

用第 7 章的技术为图 9.1 的两个注释之间的程序段产生的中间代码在图 9.2。

```
void quicksort(m,n)
int m,n;
{
    int i,j;
    int v,x;
    if(n<=m) return;
    /* 程序段开始 */
    i=m-1; j=n; v=a[n];
    while(1){
        do i=i+1; while(a[i]<v);
        do j=j-1; while(a[j]>v);
        if(i>=j) break;
        x=a[i]; a[i]=a[j]; a[j]=x;
    }
    x=a[i]; a[i]=a[n]; a[n]=x;
    /* 程序段结束 */
    quicksort(m,j); quicksort(i+1,n);
}
```

图 9.1 快速排序的 C 代码

```
(1)  i := m-1
(2)  j := n
(3)  t1 := 4*n
(4)  v := a[t1]
(5)  i := i+1
(6)  t2 := 4*i
(7)  t3 := a[t2]
(8)  if t3<v goto (5)
(9)  j := j-1
(10) t4 := 4*j
(11) t5 := a[t4]
(12) if t5>v goto (9)
(13) if i>=j goto (23)
(14) t6 := 4*i
(15) x := a[t6]
(16) t7 := 4*i
(17) t8 := 4*j
(18) t9 := a[t8]
(19) a[t7] := t9
(20) t10 := 4*j
(21) a[t10] := x
(22) goto (5)
(23) t11 := 4*i
(24) x := a[t11]
(25) t12 := 4*i
(26) t13 := 4*n
(27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := 4*n
(30) a[t15] := x
```

图 9.2 图 9.1 部分程序的三地址代码

在代码优化器中，程序用流图表示，边表示控制流，结点表示基本块，就像 8.3 节讨论的

那样。

例9.1 图9.3是图9.2程序的流图,程序所有的条件转移和无条件转移在图9.3中都改成了转移到相应的基本块。

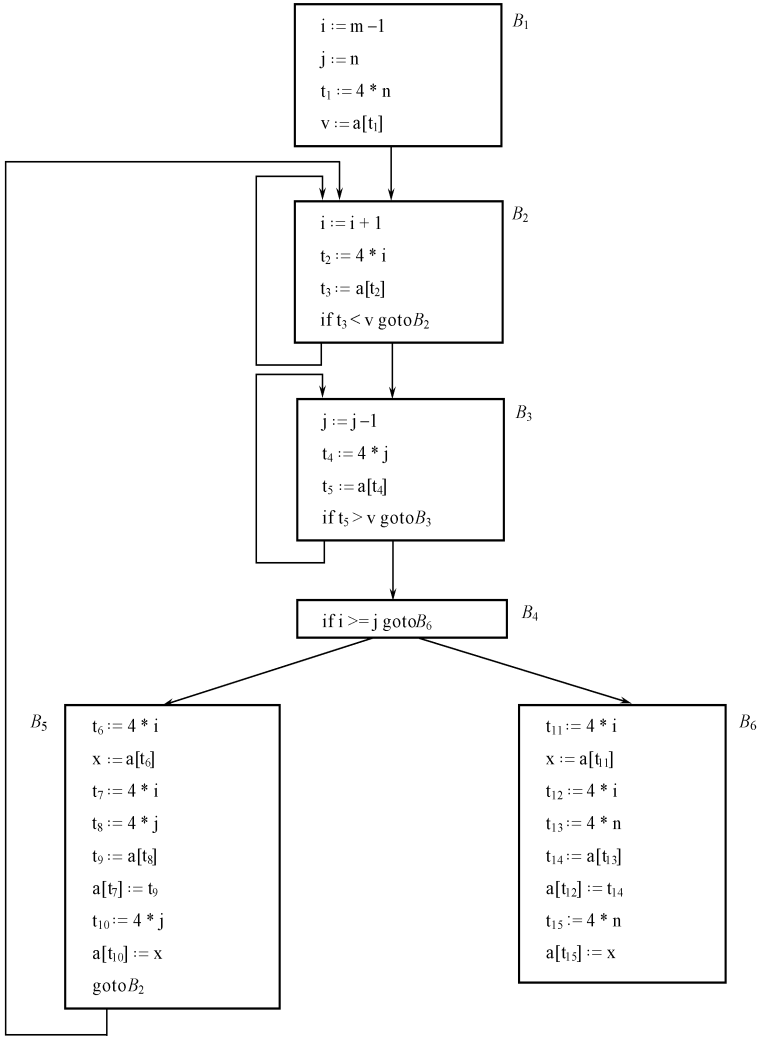


图9.3 图9.2程序的流图

在图9.3中,块 B_1 是初始结点。该图有三个循环 块 B_2 和 B_3 都单独构成循环 块 B_2,B_3,B_4 和 B_5 一起形成一个循环,该循环的首结点是块 B_2。循环的识别在9.2节介绍。

9.1.2 公共子表达式删除

如果表达式 E 先前已计算，并且从先前的计算到 E 的再次出现，E 中变量的值没有改变，那么 E 的这个再次出现称为公共子表达式。如果我们能够利用先前的计算结果，就可以避免表达式的重复计算。例如，在图 9.3 的基本块 B_5 中，对 t_7 和 t_{10} 赋值的语句分别有公共子表达式 4＊i 和 4＊j 出现在它们的右部。我们用 t_6 代替 t_7，用 t_8 代替 t_{10}，这些公共子表达式得以删除，删除后该基本块的代码如图 9.4 所示。以上我们是仅局限于基本块进行的公共子表达式的删除。

B_5
$t_6 := 4 * i$
$x := a[t_6]$
$t_8 := 4 * j$
$t_9 := a[t_8]$
$a[t_6] := t_9$
$a[t_8] := x$
goto B_2

图 9.4 删除局部公共子表达式后的基本块 B_5

例 9.2 图 9.3 流图中，B_5 和 B_6 块中全局公共子表达式和局部公共子表达式删除后的结果在图 9.5 给出。我们重点讨论 B_5 的变换。

删除了局部公共子表达式后，B_5 仍然计算 4＊i 和 4＊j，从全局看，它们仍然是公共子表达式。B_5 中三个语句

$$t_6 := 4*j; \quad t_9 := a[t_8]; \quad a[t_8] := x$$

可以由

$$t_9 := a[t_4]; \quad a[t_4] := x$$

代替。这是因为 t_4 在 B_3 中计算，在图 9.5 中可以看到，当控制从 B_3 中 4＊j 的计算传到 B_5 时，中间没有改变 j 的值，所以在 B_5 中需要 4＊j 时可以引用 t_4。

t_4 代替 t_8 后，B_5 的另一个公共子表达式变得清楚了。它是表达式 $a[t_4]$，对应于源代码中 a[j] 的值。不仅控制离开 B_3 和进入 B_5 时 j 的值没有变，而且 a[j] 的值也不变（a[j] 的值计算在临时变量 t_6 中），因为在这段区间中没有对 a 的元素赋值。这样，B_5 的语句

$$t_9 := a[t_4]; \quad a[t_6] := t_9$$

可以由 $a[t_6] = t_9$ 代替。

类似地，图 9.4 的 B_5 中对 x 赋的值和 B_2 中对 t_9 赋的值一样。删掉图 9.4 中对应到源代码表达式 a[i] 和 a[j] 的公共子表达式后，其结果是图 9.5 的 B_5。

图 9.5 的 B_6 也是完成了一串类似变换后的结果。图 9.5 的 B_1 和 B_6 中表达式 $a[t_1]$ 不能看作公共子表达式，虽然 t_1 在两个地方都使用，但是因为控制离开 B_1 进入 B_6 之前，它可以通过 B_3，B_5 有对 a 的赋值，因此在到达 B_6 时，$a[t_1]$ 的值可能和离开 B_1 时的值不一样，把 $a[t_1]$ 作为公共子表达式是不稳妥的。

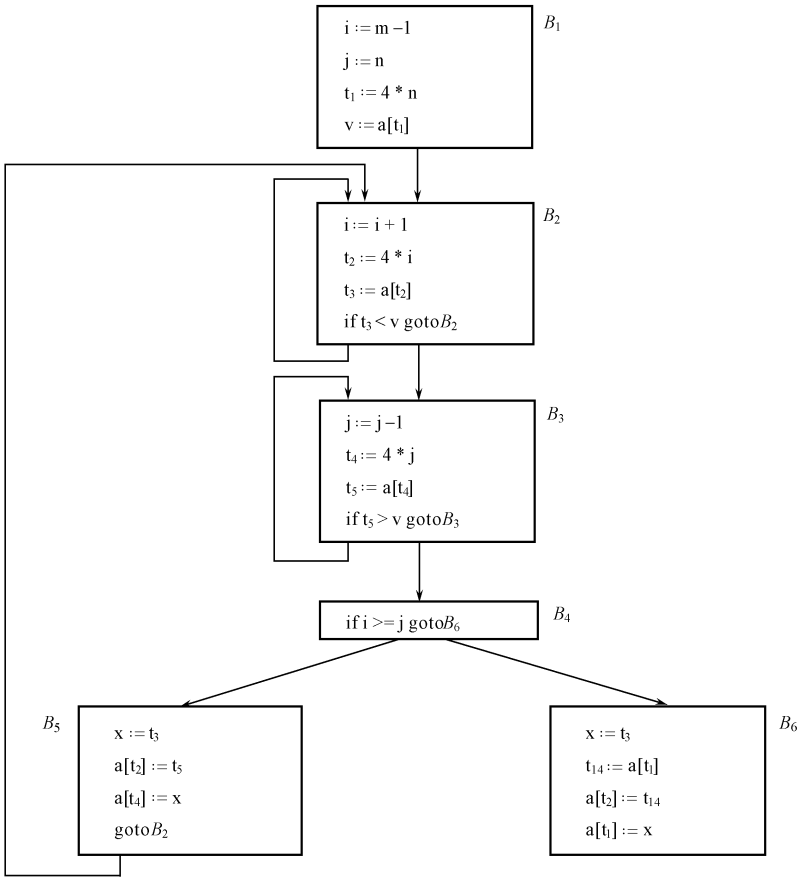


图 9.5 删除公共子表达式后的 B_5 和 B_6

9.1.3 复写传播

图 9.5 的 B_5 可以通过使用两种新的变换来删除 x 而进一步化简。一种变换是下一小节介绍的删除死代码；另一种变换和形式为 f := g 的赋值有关，这种赋值叫做复写语句，简称为复写。在例 9.2 中，若我们更深入地讨论的话，复写概念会较早一些提出，因为删除公共子表达式的算法会引进复写。其他一些算法也会引进复写。例如，当删除图 9.6 的公共子表达式 c := d+e 时，该算法使用新的变量 t 来保存 d+e 的值。因为控制到达 c := d+e 可能会在对 a 的赋值之后，也可能会在对 b 的赋值之后，因此用 c := a 或 c := b 来代替 c := d+e 都是不妥的。

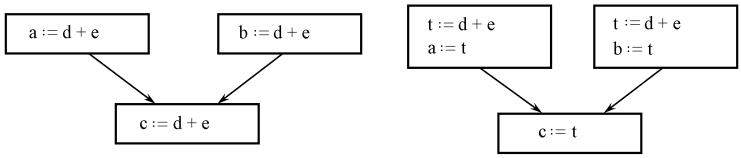


图9.6 删除局部公共子表达式期间引进复写

复写传播变换的做法是在复写语句 f := g 后，尽可能用 g 代表 f。例如，图 9.5 B_5 的赋值 x := t_3 是一个复写。把复写传播运用于 B_5 产生

$$\begin{aligned} & x := t_3 \\ & a[t_2] := t_5 \\ & a[t_4] := t_3 \\ & \text{goto } B_2 \end{aligned} \tag{9.1}$$

这看起来似乎没有改进，但我们将会看到，它增加了删除对 x 赋值的机会。

9.1.4 死代码删除

如果变量的值以后还要引用，则称它在程序的该点是活跃的，否则它在该点是死亡的。死代码或无用代码就是指计算的结果决不被引用的语句。虽然程序员不会故意引入死代码，但是前面的变换可能会引起死代码。例如我们可能在程序中增加一些

$$\text{if (debug) print ...} \tag{9.2}$$

语句来帮助测试或调试程序。当调试结束时，我们不是将它们从程序中删除，而是在程序的一开始将

debug := true

改成

debug := false

这样，从数据流分析可以推断出，程序每次到达这个语句时 debug 的值总是假。而且可以断定，不论程序实际取什么分支序列，该语句总是先于测试 (9.2) 的、对 debug 的最后一个赋值语句。当复写传播用 false 代替 debug 时，打印语句就成了死代码，可以从目标代码中删掉测试和打印。

更一般地，若在编译时能推断出一个表达式的所有运算对象都是常量，因而在编译时能完成这个计算，那么可以用该计算的结果代替这个表达式，这种变换叫做常量合并。复写传播可能会引入一些常量合并的机会。

复写传播的另一个优点是它常常使得复写语句成为死代码。例如，复写传播后再删除

死代码,可以删掉(9.1)中对 x 的赋值,把它变成

```
a[t5] := t6
a[t4] := t6
goto B2
```

这段代码是图9.5中 B_5 的进一步改进。

9.1.5 代码外提

本小节开始简单介绍一个非常重要而值得优化的地方,即循环,尤其是消耗程序运行大部分时间的内循环。如果减少了内循环的指令数,这时即使增加了外循环的指令数,程序的运行时间也可能缩短。循环优化的三种重要技术是:代码外提,它把代码移出循环;归纳变量删除,我们将用它从图9.5的内循环 B_2 和 B_3 中删掉 i 和 j;还有强度削弱,它用较快的操作代替较慢的操作,如用加代替乘。先介绍代码外提。

减少循环中代码总数的一种重要办法是代码外提。这种变换把循环不变计算,即运算结果独立于循环执行次数的表达式,放到循环的前面。例如,语句

```
while (i<=limit-2) ...
```

如果while的体不改变 limit 的值,那么 limit-2 是循环不变计算。代码外提的结果是

```
t=limit-2;
while (i<=t) ...
```

当然while的体也不能改变 t 的值。

在所给出的快速排序程序中,没有可以代码外提的地方。

9.1.6 强度削弱和归纳变量删除

考虑图9.7中由 B_3 构成的循环。j 和 t_4 的值步伐一致地变化,每次 j 的值减1,t_4 的值就减4,因为 4*j 赋给 t_4。这样的变量都叫做归纳变量。如果在循环中有两个或更多的归纳变量,也许只需要留下一个,而摆脱其余。这个操作由归纳变量删除过程来完成。对于图9.7(a) B_3 构成的循环,j 和 t_4 都不能摆脱,因为 t_4 在 B_3 中引用,j 在 B_4 中引用。然而可以用它们来说明强度削弱,而这个强度削弱又为删除归纳变量创造了机会。

例9.3 在图9.7(a)中,对内循环 B_3,若不考虑第一次进入,关系 $t_4=4*j$ 在 B_3 的入口一定保持,在 $j:=j-1$ 后,关系 $t_4=4*j+4$(即 $4*j=t_4-4$)也保持,那么 $t_4=4*j$ 可以用 $t_4=t_4-4$ 代替。现在要进行这个变换的惟一问题是第一次进 B_3 时 t_4 没有初值,所以在给 j 置初值的那个基本块的末尾给 t_4 置初值 4*j。在图9.7(b)中,这个语句放在块 B_1 的最后。

如果乘运算比加或减需要更多时间的话(许多机器都是这样),那么这种变换会加快目

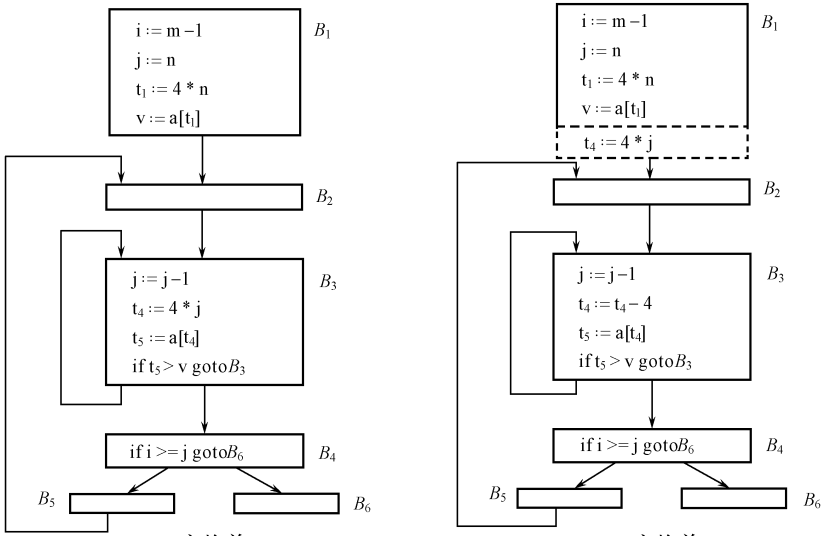


图9.7 强度削弱用于块 B_3 中的4＊j

标代码的速度。

9.4节将讨论怎样寻找归纳变量以及可以施加什么变换。下面我们再举一个删除归纳变量的例子来作为本节的结束，该例处理外循环 B_2，B_3，B_4 和 B_5 上下文中的i和j。

例9.4 把强度削减用于 B_2 和 B_3 的内循环后，i和j的作用仅在于决定 B_4 的测试结果。我们已知道j和 t_4 满足关系 $t_4=4*j$，i和 t_2 也满足关系 $t_2=4*i$，那么测试 $t_2>=t_4$ 等价于i>=j。一旦作出这种替换，B_2 的i和 B_3 的j就成了死变量，在这些块中对它们的赋值也就成了死代码，可以删除，这个结果在图9.8中给出。

前面的代码改进变换的效果是明显的。在图9.8中，B_2 和 B_3 的指令数都从图9.3最初流图的4条减为3条，B_5 从9条减到3条，B_6 从8条减到3条。虽然 B_1 从4条增加到6条，但是 B_1 在这段程序中仅执行一次，所以总的运行时间几乎不受 B_1 大小的影响。

9.1.7 优化编译器的组织

从上面的优化实例可以看到，要进行一项优化，需要掌握程序控制流和数据流方面很多信息，因此对中间代码进行控制流分析和数据流分析是代码优化阶段不可缺少的环节。本章的代码优化使用图9.9的组织形式，它由控制流分析、数据流分析和代码变换三部分组成。第8章讨论的代码生成器是从变换后的中间代码产生目标程序。

图9.9的组织形式有下列优点：

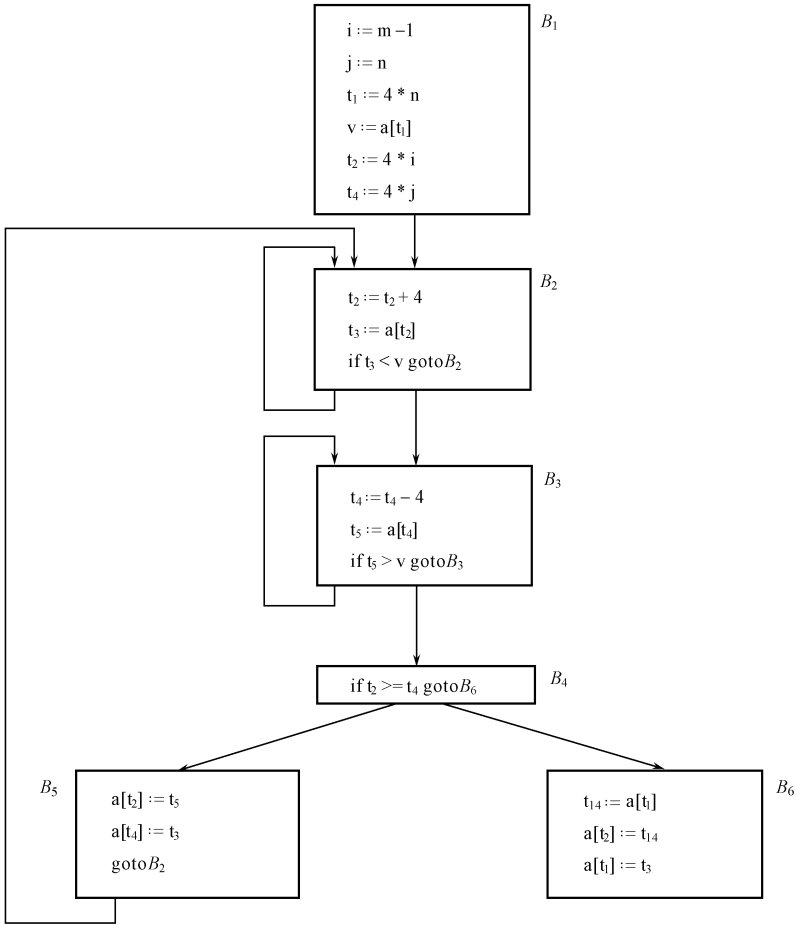


图 9.8 删除归纳变量后的流图

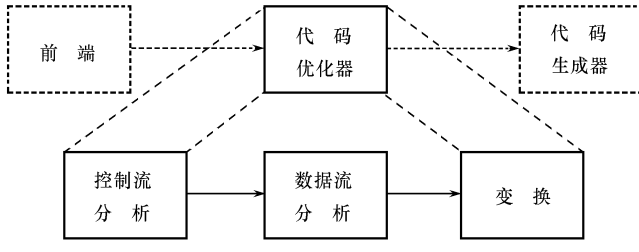


图 9.9 代码优化器的组织

⑴ 实现高级结构所需的操作在中间代码中是显式的,这就有可能优化它们。例如,a[i]的三地址计算在图9.2中是明显的,这样,像表达式4＊*i*的重复计算才可以删除。

有些代码改进变换要想在源语言级完成是不大可能的。例如,像Pascal和FORTRAN这样的语言,程序员只能按常规的方式引用数组元素*a*[*i*],即使程序员知道多次引用a[i]意味着它的地址会重复计算,他也没有办法改进它。

当然,像C这样的语言,这种变换可以由程序员在源程序级完成,因为数组元素的访问可以系统地重写成使用指针,以提高效率。这种重写类似于传统的FORTRAN优化器所做的工作。

⑵ 中间代码基本上独立于目标机器,所以,由一种机器的代码生成器改为另一种机器的代码生成器时,优化器不必作很多修改。

9.2 流图中的循环

要想优化产生较好的结果,必须考虑循环优化,因此需要定义流图中的循环由哪些结点构成。我们使用一个结点是另一个结点的必经结点的概念来定义自然循环和一类重要的流图——可归约流图。

9.2.1 必经结点

流图中结点*d*是结点*n*的必经结点,如果从初始结点起,每条到达n的路径都要经过*d*。写成*d* dom *n*。根据这个定义,每个结点是它本身的必经结点,循环的入口是循环中所有结点的必经结点。

例9.5 考虑图9.10的流图,它的初始结点是1。初始结点是所有结点的必经结点。结点2仅是它本身的必经结点,因为控制可沿着1→3开始的路径到达任何其他结点。结点3是除1和2以外的所有结点的必经结点。结点4是除了1,2和3以外的所有结点的必经结点,因为从1出发的所有路径必须由1→2→3→4或1→3→4开始。结点5和6仅是它们自己的必经结点,因为控制流可以走另一个结点而跳过这个结点。最后,7是7,8,9和10的必经结点;8是8,9和10的必经结点;9和10仅是本身的必经结点。

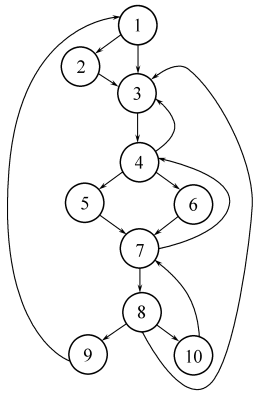


图9.10 流图

9.2.2 自然循环

必经结点信息的一重要运用是确定流图中适合于改进的循环。这样的循环有两个基本性质。

(1) 循环必须有惟一的入口点,叫做首结点,首结点是循环中所有结点的必经结点。

(2) 至少有一种办法重复循环,也就是至少有一条路径回到首结点。

寻找流图中所有循环的一个办法是找出流图所有的回边(如果有 a dom b,那么边 b→a 叫做回边)。

例 9.6 在图 9.10 中,4 *dom* 7,则 7→4 是回边。类似地,7 *dom* 10,10→7 是回边。其他的回边有 4→3,8→3 和 9→1。

给出一个回边 n→d,*我们定义这个边的*自然循环是 d 加上所有不经过 d 能到达 n 的结点。d 是这个循环的首结点。

例 9.7 回边 10→7 的自然循环由结点 7,8 和 10 组成,因为 8 和 10 是所有能够不经过 7 而到达 10 的结点。回边 9→1 的自然循环是整个流图(不要忘记路径 10→7→8→9)。

算法 9.1 构造回边的自然循环。

输入 流图 G 和回边 n→d。

输出 由回边 *n→d 确定的自然循环中所有结点*的集合 loop。

方法 由结点 *n 开始,考虑已置入 loop 的每个结点* m,m≠*d,以保证 m 的前驱也能置入 loop,这个算法*在图 9.11 中给出。loop 中的每个结点,除了 d 以外,一旦加入 stack,它的前驱就要被检查。注意,因为 d 是初始时置入循环,我们决不会考察它的前驱,因此仅找出那些不经过 d 可以到达 n 的结点。

```
procedure insert(m);
if m∉loop then begin
        loop := loop∪{m}
        把 m 压入 stack
end;
/*下面是主程序*/
stack := 空;
loop := {d};
insert(n);
while stack 非空 do begin
    弹出 stack 的顶元 m;
    for m 的每个前驱 p do insert
    (p)
end
```

图 9.11 构造自然循环的算法

如果把自然循环作为"循环",那么我们有一个实用的性质:两个循环要么不相交,要么一个完全包含(嵌入)在另一个里面,除非它们有相同的首结点。于是,暂时忽略有相同首结点的情况,若一个循环的结点集合是另一个循环的结点集合的子集,那么相对后一个循环而言,前一个循环是内循环。不再包含其他循环的循环则是最内循环。

当两个循环有相同的首结点,但并非一个循环的结点集合是另一个的子集,例如像图

9.12那样，我们很难说哪个是内循环。例如，若 B_1 结尾的测试是

if a=10 goto B_2

则循环 $\{B_0, B_1, B_3\}$ 可能是内循环。但是，如果不仔细检查代码，我们不能保证这一点。可能a大多数时候是10，那么在进入 B_3 之前会环绕循环 $\{B_0, B_1, B_3\}$ 很多次。所以，我们认为，当两个自然循环有相同的首结点，并且不是一个嵌在另一个里面时，可以把它们合并，看成一个循环。

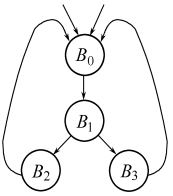
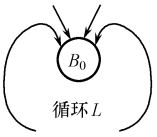


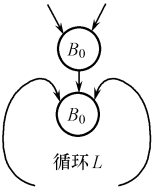
图 9.12 有相同首节点的两个循环

9.2.3 前置结点

某些变换要求移动语句到首结点的前面。于是，开始处理一个循环 L 时，创建一个新基本块，叫做前置结点。前置结点的惟一后继是 L 的首结点，并且原来从 L 外到达 L 首结点的边都改成进入该前置结点。从循环 L 里面到达首结点的边不改变。这种整理在图9.13给出。起初，前置结点为空，然后 L 的变换可能会放置一些语句到该结点中。



(a) 整理前，B_0 是首结点



(b) 整理后，增加前置结点 B_0'

图 9.13 引入前置结点

9.2.4 可归约流图

实际出现的流图常常落入下面定义的可归约流图类。结构化的控制流语句，如 if－then－else，while－do，continue 和 break 语句，它们的使用产生的程序流图总是可归约的。甚至事先没有结构化程序设计概念的程序员用 goto 语句编的程序，几乎也都是可归约的。

有好几种关于可归约流图的定义，我们采用的定义能显示出可归约流图的一个非常重要的性质：不存在从循环外向循环内的转移，进入循环只通过它的首结点。

一个流图 G 是可归约的，当且仅当可以把它的边分成两个不相交的子集，其中的边分别叫做正向边和回边，并且有下列性质：

⑴ 正向边子集形成有向无环图，在这个图中，每个结点可以从 G 的初始结点到达。

⑵ 回边子集仅由前面所讲的回边组成。

例9.8 图9.10的流图是可归约的。通常，如果知道了流图的 *dom* 关系，就可以找出

和去掉所有的回边。如果流图可归约,那么剩下的边必定都是正向边,所以检查流图是否可归约,只要检查所有正向边是否构成有向无环图便可以了。对于图9.10,如果拿开5条回边4→3,7→4,8→3,9→1和10→7,很容易看出剩下的图是无环的。

例9.9 我们看图9.14的流图,它的初始结点是1。该流图没有回边,因为2→3和3→2都不是回边。由于该图不是无环的,因此它不是可归约的。直观上,这个流图不可归约的原因是,可以从结点2和3两处进入由它们构成的环。这相当于该"循环"有两个首结点,它使得许多代码优化技术,如9.1节介绍的代码外提和归纳变量删除,都不能直接运用。

幸好,像图9.14这样的不可归约控制流结构在大多数程序里面几乎不出现,因而研究多于一个首结点的循环没有多大价值。有些语言只允许程序有可归约流图,另一些语言,只要不使用goto语句,也只会产生可归约流图。

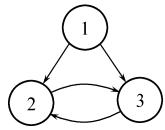


图9.14 非归约流图

对循环分析来说,可归约流图的关键性质是,我们非形式地称为循环的结点集合一定含一条回边。因此只要通过回边找出所有的自然循环,也就找出了所有的循环。

例9.10 回到图9.10,可以看出,最内循环是{7,8,10},它是回边10→7的自然循环。集合{4,5,6,7,8,10}是回边7→4的自然循环,注意,8和10可经10→7到达7。直观上看去,{4,5,6,7}是一个循环,这是错的,因为4和7都是入口点,违反了有关一个入口的限制。从另一角度说,没有理由认为控制会环绕结点集合{4,5,6,7}消耗较多的时间,完全有可能控制从7到8的次数多于从7到4的次数。把8和10包含在这个循环里,我们可确信已分离出程序频繁执行的一个区域。

应该认识到,对各分支作出执行频度的假设是危险的。例如,若把循环{7,8,10}的不变语句移出8或10,而事实上,控制沿边7→4比沿7→8更频繁。那么,我们实际上增加了被移动语句的执行次数。在9.4节我们将讨论避免这个问题的方法。

下一个较大的循环是{3,4,5,6,7,8,10},它是回边4→3和8→3的自然循环。和前面一样,如果把{3,4}看成循环则违反了关于一个入口点的要求。最后一个循环是回边9→1的自然循环,它是整个流图。

9.3 全局数据流分析介绍

为了优化代码,编译器需要把程序流图作为一个整体来收集信息,并把这些信息分配给流图的各个基本块。例如,从9.1节可以看到,使用全局公共子表达式的信息可以删除更多的冗余计算。

可以通过建立和解方程来收集数据流信息,这些方程联系程序不同点的信息。典型的方程形式为

$$out[B] = gen[B] \cup (in[B] - kill[B]) \tag{9.3}$$

这个方程的意思是,当控制流通过基本块 *B* 时,在 *B* 末尾得到的信息是在 *B* 中产生的信息,或者是进入 B 开始点并且没有被 B 注销的信息。这样的方程序叫做数据流方程。

怎样建立和解数据流方程依赖三个因素:

⑴ 产生和注销的概念依赖于所需要的信息,即根据数据流方程所要解决的问题。而且,对某些问题,不是沿着控制流前进并且由 *in*[B] 来定义 *out*[B],而是反向前进并由 *out*[B] 来定义 in[B]。

⑵ 因为数据沿控制路径流动,所以数据流分析受程序控制结构影响。

⑶ 过程调用、通过指针赋值、甚至对数组变量的赋值等的存在使得数据流分析大大复杂。我们不打算讨论这些较复杂的问题。

9.3.1 点和路径

在讨论代码优化时,需要定义程序的点和路径。在基本块中,两个相邻的语句之间为程序的一个点,第一个语句前和最后一个语句后各有一点,分别称为该块的开始点和结束点。例如,图 9.15 块 B_1 有四个点,第一点在所有赋值语句前,其余三个点是每个语句后各一个。

若以语句为点,当程序执行到该点时,该点对应的语句是执行了还是没有执行,是需要规定的。为清楚起见,以语句的前后为程序的点。另外,在讨论数据流分析时,我们可能又要引用语句,为便于引用,可以给语句加一个标号,如图 9.15 所示。

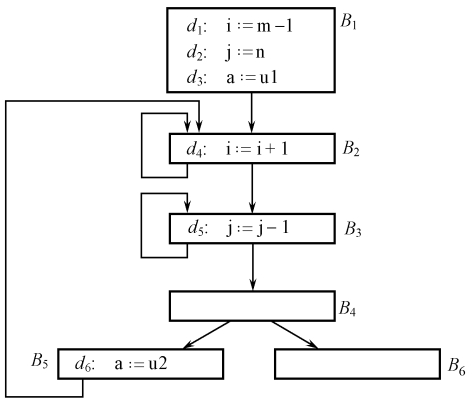


图 9.15 一个流图

从全局观点考虑所有块的所有点,从点 p_1 到点 p_n 的路径是点序列 $p_1, p_2, \ldots, p_n$,对 1 和 $n-1$ 间的每个 i,满足:

⑴ p_i 是先于一个语句的点,p_{i+1} 是同一块中位于该语句后的点,或者

⑵ p_i 是某块的结束点,p_{i+1} 是后继块的开始点。

例 9.11 在图 9.15 中,有一条路径从块 B_5 的开始点到 B_6 的开始点。它经过 B_5 的结束点,然后依次通过 B_2,B_3 和 B_4 的点,到达 B_6 的开始点。

9.3.2 到达－定值

变量 x 的定值是一个语句,它赋值或可能赋值给 x。最普通的定值是对 x 的赋值或读 x 的语句。这些语句真正修改 x 的值,可以称为 x 的确切定值。也还有一些语句,它们可能对 x 定值,称为 x 的可能定值。最常使用的 x 的可能定值形式有:

(1) 把 x 作为参数的过程调用(值参数除外),或者所调用的过程可以访问 x,因为 x 在该过程的作用域内。还要考虑"别名"的可能性,x 虽然不在该过程的作用域内,但 x 和另一个变量绑定在同一地址,这个变量作为参数传递或在该过程的作用域内。

(2) 通过引用 x 的指针来对 x 赋值。例如,如果 q 可能指向 x 的话,赋值 *q := y 是对 x 的可能定值。我们不打算讨论确定一个指针可能指向什么变量的方法。在缺少信息的情况下,可以假定通过一个指针的赋值是对每个变量的一个可能定值。

我们说变量被语句 *s* 引用,如果 s 需要它的右值的话。例如,b 和 c(a 不是)在语句 a := b+c 和 a[i] := c 中被引用。同样对于变量的引用,也有确切引用和可能引用之分。

我们说 x 的定值语句 *d 能*到达点 p,如果存在从紧跟 d 的点到达 p 的路径,并且在这条路径上没有 x 的确切定值。如果在这条路径上有 x 的确切定值,那么我们称语句 *d* 的 x 定值在这条路径上被注销。注意,只有 x 的确切定值可以注销 x 的其他定值,因为 x 的可能定值不一定是对 x 的赋值。这样,如果变量 x 的定值语句 *d 能到达点 p,那么在 p* 点 x 的最新值可能是在语句 *d* 赋给 x 的,仅仅是可能而已。换种说法,一条路径上 x 的确切定值和其后的 x 的可能定值都可以到达某个点。由此可知,到达－定值信息是不精确的,我们所说的到达某一点的定值集合是运行时能到达这一点的定值集合的超集。

例如,图 9.15 B_1 的定值 i := m−1 和 j := n 都能到达 B_2 的开始点。如果 B_4 和 B_5 中没有 j 的确切定值,B_3 的定值 j := j−1 的后面部分也是如此,那么定值 j := j−1 也可以到达 B_2。但是 B_3 中对 j 的赋值注销了定值 j := n,因此 j 的这个定值不能到达 B_4,B_5 或 B_6。

还有另外一个原因会影响到达－定值信息的精确性,假设流图的所有边都是会经过的,但实际上可能不是这样。例如,不管 a 和 b 是什么值,控制也不会到达下面程序段的赋值 *a* := 4

```
if a=b then a := 2
else if a=b then a=4
```

一般而言,决定流图中是否每条路径都会经过是一个不可判定问题。

虽然我们所定义的到达－定值是不精确的,但它是稳妥的。在设计代码改进变换时,面临任何怀疑,我们必须取稳妥的策略,虽然用稳妥策略会使我们失去某些实际是安全的变换。一个策略称为是稳妥的,如果它决不会导致程序所完成的计算发生改变。

为简单起见,从现在开始,我们集中在只有确切定值和确切引用的情况,分别直接称为

定值和引用。

下面考虑到达－定值的迭代算法。首先定义到达－定值的 *gen*[B]，*kill*[B]，*in*[B]和 *out*[B]。*如果 B 中的定值语句 d 能到达 B 的结束点，那么 d 在 gen*[B]中，*与它是否能到达 B 的开始点无关。可以说 gen*[B]是由 B 产生的定值。同样，*kill*[B]是指整个程序中决不*会到达 B 结束点的定值，也与它们是否能到达 B 的开始点无关。in[B]是能到达 B 的开始点的定值集合，out[B]是能到达 B 的结束点的定值集合，它们的计算都需要考虑整个程序的控制流。out[B]和 gen[B]是有区别的，后者是指那些不经过 B 外的路径就能到达 B 结束点的定值集合。

假定每个块的 gen 和 kill 都已经计算，可以建立两组方程，它们将 in 和 out 联系起来。第一组方程表示，in[B]是从 B 的所有前驱能到达的定值的并集。第二组方程就是前面提到的典型方程(9.3)。这两组方程是：

$$in[B] = \bigcup_{P\text{是}B\text{的前驱}} out[P]$$

$$out[B] = gen[B] \cup (in[B] - kill[B]) \qquad (9.4)$$

如果流图有 n 个基本块，从(9.4)我们得到 2*n*个方程。*对这*2*n*个方程，*可以迭代求解各*基本块的 in 和 out 集合。

算法 9.2 到达－定值的迭代求解。

输入 程序流图及各基本块 *B*的 *kill*[B]和 *gen*[B]。

输出 每个块 *B*的 *in*[B]和 *out*[B]。

方法 由所有 *B*的 *in*[B]＝ 开始迭代，一直迭代到所有 B 的 in(从而所有 B 的 out)都不再变化为止。该算法的轮廓在图 9.16。

```
    /*假定对所有 B, in[B] =  ,给 out[B]置初值*/
(1) for 每个基本块 B do out[B] := gen[B];
(2) change := true;        /*判断迭代是否继续的变量*/
(3) while change do begin
(4)       change := false;
(5)       for 每个基本块 B do begin
(6)             in[B] = ∪ out[P]   (P是B的前驱)
(7)             oldout := out[B];
(8)             out[B] := gen[B] ∪ (in[B] - kill[B])
(9)             if out[B] ≠ oldout then change := true
          end
    end
```

图 9.16 计算 *in*和 *out*的算法

直观上可以看出,算法 9.2 传播定值到尽可能远的地方,只要它们没有被注销。

该算法是终止的,因为在迭代过程中,任何 B 的 out[B] 集合不会减小。由于所有的定值集合都是有限的,最终总有一遍 while 循环的结果使得 change 仍为 false,算法终止。这个终止是安全的,因为 out 没有改变,则下一遍的 in 也不会改变。如果 in 不变,那么 out 也不变,所有以后各遍都不会有任何改变。

可以证明,while 循环次数的上界是流图中的结点数。直观理由是,如果某定值到达一点,那么沿着无环的路径它可能到达其他点,流图中结点数目是无环路径结点数的上界。每次 while 循环,定值至少沿着该路径传播到下一个结点。

例 9.12 图 9.17 是某程序的流图,我们把算法 9.2 用于这个流图。

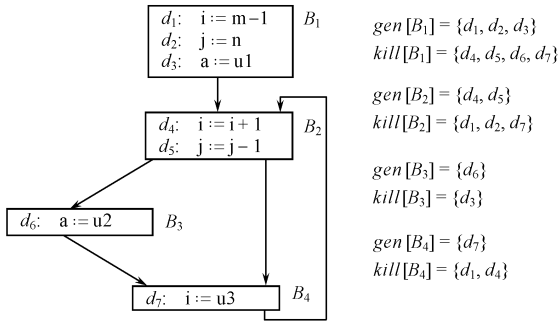


图 9.17 说明到达-定值的流图

我们用位向量表示定值集合,从左边开始的第 i 位表示定值 d_i。图 9.16 第一行循环为各个 *B 置初值 out[B] = gen[B], out[B] 的这些初值列在表* 9.1 中。为完整起见,每个 in[B] 的初值 也列在表中。假如 for 循环的执行次序是 B = B_1, B_2, B_3, B_4。对 B = B_1,因开始结点没有前驱,所以 in[B_1] 仍然是空,out[B_1] 仍然等于 gen[B_1],因而不把 change 置成 true。

然后考虑 B = B_2,在第 (6) 行计算

$$in[B_2] = out[B_3] \cup out[B_4] = 111\ 0000 + 000\ 0001 = 111\ 0001$$

和

$$out[B_2] = 000\ 1100 + (111\ 0001 - 110\ 0001) = 001\ 1100$$

这个计算总结在表 9.1 中。在第一遍扫描的结束,out[B_4] = 001 0111。它反映了这样的事实,d_7 被生成,并且 d_3, d_5 及 d_6 到达 B_4,但没有被 B_4 注销。

第二遍扫描后,in 和 out 集合不再有什么变化,所以在第三遍扫描后算法终止。

表9.1 *in*和*out*的计算

块 B	初 始		第一遍扫描		第二遍扫描	
	in[B]	*out*[B]	*in*[B]	*out*[B]	*in*[B]	*out*[B]
B_1	000 0000	111 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	000 1100	111 0001	001 1100	111 0111	001 1110
B_3	000 0000	000 0010	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	000 0001	001 1110	001 0111	001 1110	001 0111

许多数据流分析问题的方程在形式上同(9.4)的方程类似,下面的两个重要特征可用来区别这些方程:

(1) 到达－定值方程是正向的方程,所谓正向是指基本块的 out 集合根据 in 集合来计算。我们将会用到另一类数据流方程,它们是反向的,即由 out 集合来计算 in 集合。

(2) 表达前后基本块数据流信息联系的算符,我们称之为合流算符。在到达－定值情况下,到达一个基本块开始点的定值集合是到达其所有前驱基本块结束点的定值集合的并集,因此在这里求并集的算符是合流算符。与此相反,在考虑全局可用表达式问题时,求交集的算符是合流算符,因为一个表达式只有在块 B 的所有前驱块的结束点可用时,它才在块 B 的开始点可用。

通常,把到达－定值的信息存储为引用－定值链(或叫 ud 链)是方便的,它是指能够到达变量的某个引用的所有定值,可以把它们组织为每个变量一张表。如果块 B 中变量 a 的引用前没有 a 的定值,那么 a 的这个引用的 ud 链是 in[B]中 a 的定值集合。如果块 B 中 a 的引用前有 a 的定值,那么只有最后一个定值在 ud 链中,in[B]不在这个 ud 链中。

9.3.3 可用表达式

如果从初始结点到 p 的每条路径上(不必是无环)都计算 x+y,并且在最后一次这样的计算和 p 之间没有对 x 或 y 的赋值,那么我们称表达式 x+y 在点 p 可用。对可用表达式,我们说一个基本块注销表达式 x+y,如果它有对 x 或 y 的定值,并且随后没有重新计算 x+y。我们说一个基本块产生表达式 x+y,如果它计算 x+y,并且随后没有对 x 或 y 的定值。

注意,可用表达式注销和产生的概念和到达－定值的这些概念不完全一样,然而,它们遵守同样的规律。

可用表达式的基本应用是寻找公共子表达式。例如,在图 9.18 中,如果表达式 4＊i 在块 B_3 的开始点可用,那么块 B_3 的 4＊i 是公共子表达式。这有两种可能,一是块 B_2 没有对 i 定值,如图 9.18(a)所示。另一种如图 9.18(b)所示,块 B_2 中 i 的定值后面又重新计算 4＊i。

很容易计算基本块产生的可用表达式集合。在块的开始点,假定无可用表达式,然后从

头到尾扫描块中所有语句,如果在 p 点可用表达式集合是 A,q 是 p 的下一点,p 和 q 之间的语句是 x := y+z,那么 q 点的可用表达式集合由下列两步计算:

(1) 把表达式 y+z 加入 A 中。

(2) 删掉 A 中任何含 x 的表达式。

注意,这两步的次序不能调换,因为 x 也可能就是 y 或 z。到达块的结束点后,A 是此基本块产生的表达式集合。

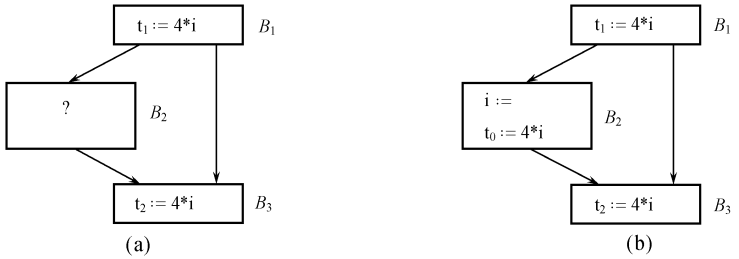


图 9.18 穿越块的公共子表达式

一个基本块注销的表达式集合包括所有的表达式 y+z,其中 y 或 z 在本块中定值,并且该块没有产生 y+z。

可用类似于计算到达-定值集合的方式寻找可用表达式。假定 U 是程序中出现在语句右部的所有表达式集合。对每个基本块 B,令 in[B] 是在块 B 开始点的可用表达式集合,令 out[B] 是在块 B 结束点的可用表达式集合,定义 e_gen[B] *是块 B 生成的可用表达式集合,定义* e_kill[B] *是 U 中(但不在 B 中)被块 B 注销的表达式集合。下列方程把未知的* in *和* out *同已知的* e_gen *和* e_kill *联系起来。*

$$\begin{aligned} out[B] &= e\_gen[B] \cup (in[B] - e\_kill[B]) \\ in[B] &= \bigcap_{P\text{是}B\text{的前驱}} out[P] \quad (B\text{ 不是初始块}) \\ in[B_1] &= \quad (B_1\text{ 是初始块}) \end{aligned} \tag{9.5}$$

方程 (9.5) 和 (9.4) 的到达-定值方程看起来似乎一样,但它们还是有区别的。第一个区别是初始块的 in 处理为特殊情况。这是因为程序从初始块开始执行时,没有任何东西可用。即使某些表达式沿着从程序的其他地方到达初始块的所有路径都可用也不行,因为若不强置 in[B_1] 为空,可能会错误地推断出某些表达式在程序开始执行前就可用。

第二个(也是更重要的)区别是,合流算符是交集运算而不是并集运算。因为只有当一个表达式在某块的所有前驱块的结束点都可用时,它才在该块的开始点可用。与此相反,一个定值只要能到达一个块的某个前驱的结束点,就可以到达该块的开始点。

∩而不是∪的使用使方程 (9.5) 的求解方式和方程 (9.4) 的有区别。虽然这两种方程的

解都不惟一，但对 (9.4) 而言，求的是对应于到达一定值的最小解。为了得到这个解，由假设没有任何东西可到达任何地方开始，然后逐步增大到这个解。按这种方式，我们决不会让定值 d 到达点 p，除非真能找到路径把 d 传播到 p。相反，对方程 (9.5)，我们想要的是最大可能解，所以从足够大的近似开始，然后逐步减小到所要的解。

我们从假定任何东西 (即集合 U) 在任何地方都可用开始，然后逐步删掉一些表达式。删除一个表达式的依据是，我们能够找到一条路径，在这条路上它们不可用。最后我们获得真正的可用表达式集合。

在可用表达式情况下，得到精确的可用表达式集合的一个子集是一种稳妥的办法，我们用方程 (9.5) 所能得到的正是这样的子集。这种子集之所以稳妥的原因是，我们利用这些信息是为了把先前计算的值代替可用表达式的计算，把一个可用表达式当成不可用仅仅禁止了我们进行这种优化。

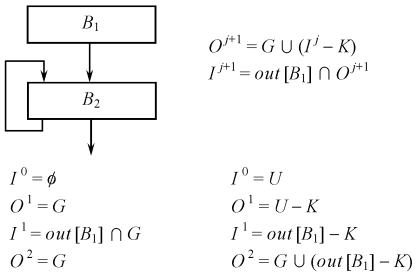


图 9.19 *in 集合的不同初值比较*

例 9.13 我们把注意力放在图 9.19 的基本块 B_2 来说明 in [B_2] 的初值对 out [B_2] 的影响，由此看到用 U 作为迭代初值的合理性。令 G 和 K 分别是 gen [B_2] 和 kill [B_2] 的缩写，块 B_2 的数据流方程是：

$$in[B_2] = out[B_1] \cap out[B_2]$$

$$out[B_2] = G \cup (in[B_2] - K)$$

我们用 I^j 和 O^j 分别表示 in [B_2] 和 out [B_2] 的第 j 次近似，将这些方程重写在图 9.19 中。图中还显示出，以 $I^0 =$ 开始，得到 $O^1 = O^2 = G$；而以 $I^0 = U$ 开始，得到较大的 O^2 集合。在这两种情况下，out [B_2] 都等于 O^2，因为它们都收敛在这一点。

直观上，以 $I^0 = U$ 开始，用

$$out[B_2] = G \cup (out[B_1] - K)$$

得到的解是我们想要的，因为它正确反映了这样一个事实，即 out [B_1] 中没有被块 B_2 注销的表达式在块 B_2 的结束点可用，如同块 B_2 产生的表达式那样。

算法 9.3 可用表达式的迭代求解。

输入 流图 G,每个块 B 的 e_kill[B] 和 e_gen[B] 已计算,初始块是 B_1。

输出 每个块 B 的 in[B] 集合。

方法 执行图 9.20 的算法,每步的解释和图 9.16 的类似。

```
in[B1] := ∅;
out[B1] := e_gen[B1];   /*对初始结点 B1,in 和 out 决不会改变*/
for B≠B1 do out[B] := U-e_kill[B]      /*初始估计值取最大*/
change := true;
while change do begin
    change := false;
    for B≠B1 do begin
        in[B] := ∩ out[P]   (P是B的前驱)
        oldout := out[B];
        out[B] := e_gen[B] ∪ (in[B] - e_kill[B]);
        if out[B] ≠ oldout then change := true
    end
end
```

图 9.20 可用表达式的计算

9.3.4 活跃变量分析

一些代码改进变换依赖从程序流图反方向计算得到的信息,现在考虑其中两种。在活跃变量分析中,对变量 x 和点 p,我们希望知道 x 的值在 p 点开始的路径上是否被引用,如果被引用,我们说 x 在 p 点活跃,否则称 x 在 p 点是死亡的。

我们已经知道,产生目标代码时,活跃变量信息有重要价值。一个保存在寄存器中的值,假定在一个基本块中有对它的引用,而在该块的末尾它是死亡的,那么离开该块时这个值不必存储。还有,如果所有寄存器都占用了,此时还需要另一寄存器,我们首先选择存放死亡值的寄存器,因为这个值不必存储。

我们定义 in[B] 是在块 B 开始点的活跃变量集合,定义 out[B] 是在块 B 结束点的活跃变量集合。def[B] 是块 B 中有定值且该定值前没有引用的变量集,use[B] 是块 B 中有引用且在该引用前没有定值的变量集。那么,联系 def 和 use 同未知的 in 和 out 的方程是:

$$\begin{aligned} in[B] &= use[B] \cup (out[B] - def[B]) \\ out[B] &= \bigcup_{S\text{是}B\text{的后继}} in[S] \end{aligned} \tag{9.6}$$

第一组方程指出,如果一个变量在某块中定值前有引用,或者在该块结束点活跃并且没

有被该块定值,那么它在该块开始点活跃。第二组方程指出,变量在块的结束点活跃,当且仅当它在该块的某个后继块开始点活跃。

应该注意方程(9.6)和(9.4)的到达－定值方程的联系,它们的 in 和 out 的作用互相交换了,use 和 def 分别代替了 gen 和 kill。如同(9.4)一样,方程(9.6)的解也不必惟一,我们要的也是最小解。用于求这个最小解的算法本质是算法9.2的反向版本。因为检查 in 是否改变的方法类似于算法9.2和算法9.3中检查 *out 是否改变的方法,因此我们在算法*9.4中省略了判断 while 循环终止的代码。

算法9.4　活跃变量分析。

输入　基本块的 *def 和 use 都已计算的流图。*

输出　*out[B],即在流图的每个块 B 出口*的活跃变量集合。

方法　执行图9.21的程序。

```
for 每个基本块 B do in[B] := ∅;
while 任何一个 in 有变化 do
    for 每个基本块 B do begin
        out[B] = ∪ in[S]   (S是B的后继)
        in[B] = use[B] ∪ (out[B] − def[B])
    end
```

图9.21　活跃变量的计算

和活跃变量分析以同样方式完成计算的还有定值－引用链(du 链)。du 链问题是计算对变量(如 x)定值的所有引用语句 s 的集合,s 满足从紧接着该定值的点 p 到 s 之前那一点的路径上没有 x 的定值。

像活跃变量那样,如果能够计算 out[B],即可到达块 B 结束点的引用集合,那么对块 B 中任意点 p,就能够计算可到达它的引用,只要扫描块 B 中 p 后面的部分即可。特别是,如果块中有变量 x 的定值,那么可以决定这个定值的 du 链,即这个定值所有的引用。其方法类似于计算 ud 链的方法,我们把它留给读者。

计算 du 链信息的方程很像(9.6),但 def 和 use 需要修改。在 use[B] 的地方,用块 B 的引用集合代替,即二元组(s,x)的集合,s 是块 B 的语句,它引用变量 x,并且 B 中 s 的前面没有 x 的定值。代替 def[B] 的是两元组(s,x)的集合,s 是引用 x 的语句,s 不在块 B 中,块 B 有 x 的定值。这些方程的求解方法明显类似于算法9.4,我们不再进一步讨论。

9.4　代码改进变换

9.1节介绍的完成代码改进变换的算法依赖于数据流信息,9.3节介绍了怎样收集这些信息。本节将介绍如何利用这些信息来完成公共子表达式删除、复写传播、循环不变计算外提和归纳变量删除。在编译器中,有些变换可以一起完成,但是这里提出的概念是基于单个的变换。

本节强调的是使用从程序获得的整体信息的全局变换。正如上一节所说的,全局数据流分析通常不关心基本块中的点,它代替不了局部变换,因此两者都必须实施。例如,当执行全局公共子表达式删除时,我们只关心一个表达式是否由某个基本块生成,而不关心它在该块中重新计算几次。

9.4.1 公共子表达式删除

上节讨论的可用表达式数据流向题可以判断一个表达式在流图的 p 点是否为公共子表达式。下面的算法将 9.1 节提出的公共子表达式删除的直观概念加以形式化。

算法 9.5 全局公共子表达式删除。

输入 有可用表达式信息的流图。

输出 修改后的流图。

方法 对每个形式为 x ≒ y+z 的语句 s(仍用 + 代表一般的算符),如果 y+z 在 s 所在块的开始点可用,在该块中 s 前没有 y 或 z 的定值,则执行下面的步骤:

(1) 为了寻找到达 s 所在块开始点的 y+z 的计算,我们顺着流图的边,从该块开始反向搜索,但是不穿过任何计算 y+z 的块。在每一条路径上,最后一次计算的 y+z 就是到达 s 的 y+z。

(2) 建立新变量 u。

(3) 把 (1) 中找到的每个语句 w ≒ y+z 用

u ≒ y+z

w ≒ u

代替。

(4) 用 x ≒ u 代替语句 s。

关于该算法的一些说明如下:

(1) 前述步骤 (1) 中寻找到达 s 的 y+z 的计算也可以形式化为一个数据流分析问题。但是,为所有的表达式 y+z 和所有的基本块求解这个问题是没有意义的,因为这样会收集了许多无关的信息。我们宁可在流图上搜索有关的语句和表达式。

(2) 算法 9.5 进行的修改并非都是改进。我们可能需要限制在步骤 (1) 发现的到达 s 的不同计算的个数,很可能限制到 1。不过,下面讨论的复写传播允许几个 y+z 的计算到达 s 时也能获得益处。

(3) 对于下面两组语句,算法 9.5 将漏掉 a*z 和 c*z 有相同的值这个事实。

a ≒ x+y 和 c ≒ x+y

b ≒ a*z d ≒ c*z

因为处理公共子表达式的这种简单方法仅考虑字面的表达式,而不是表达式计算的值。在

一遍扫描中找出这样的等价是可能的。用算法9.5进行多遍扫描也能找出它们，可以考虑重复该算法，直到没有新的变化为止。如果a和c是临时变量，它们在块外不再引用，那么，对临时变量作专门处理，可使公共子表达式(x+y)＊z被找出来，见下面的例子。

例9.14 假定在图9.22(a)的流图中对数组a没有赋值，我们可以稳妥地认为a[t2]和a[t6]是公共子表达式。问题是怎样删除这个公共子表达式。

图9.22(a)的公共子表达式4＊i在图9.22(b)中已被删除。发现a[t2]和a[t6]也是公共子表达式的一种办法是用复写传播把t2和t6都改成u，使得两个表达式都成为a[u]，再次使用算法9.5就能把它删除。注意，同样的新值u插入在图9.22(b)的两个块中，因此局部的复写传播足以把a[t2]和a[t6]都变成a[u]。

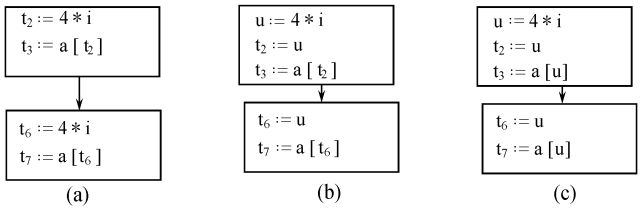


图9.22 发现a[t2]和a[t6]也是公共子表达

9.4.2 复写传播

算法9.5和后面要讨论的删除归纳变量的算法，都会引入形如x := y的复写语句。中间代码生成器也会直接生成复写语句，不过大部分都是对局部于基本块的临时变量赋值。如果能找出复写语句s：x := y中x定值的所有引用点，并用y代替x，那么可以删除这个复写语句，但它必须以每个x的引用u满足下列条件为前提：

(1) 语句*s是到达u的惟一的x定值*(即引用*u*的ud链只含*s*)。

(2) 从s到u的每条路径，包括穿过u若干次的路径(但没有第二次穿过s)上，没有对y的赋值。

条件(1)可用ud链信息来检查，但条件(2)呢？需要建立新的数据流分析方程来解决这个问题，其中*in*[*B*]是复写语句*s*：x := y的集合，就是从初始结点到块*B*的开始点的每条路径上都有这样的语句*s*，最后出现的*s*后面没有对x或y的赋值。集合*out*[*B*]可以相应地对块*B*的结束点定义。如果复写语句*s*：x := y出现在块*B*中，且块*B*中该语句的后面没有对x或y的定值，那么我们说*s在块B*中产生。如果x或y在块*B*中赋值，*并且s不在*块B中，我们说*s*：x := y在块*B*中注销。对x的赋值会注销x := y的概念类似于到达－定值，但是对y赋值也会注销它是这个问题特有的。注意，对x或y的赋值会注销x := y这个

事实的重要结果是,对 x 在左边的复写语句,*in*[*B*] 只可能含一个这样的语句。

令 U 代表程序中所有复写语句的集合,注意,不同位置的语句 x ≒ y 在 U 中是不同的。定义 *c_gen*[*B*] 是由块 *B* 产生的所有复写集合,*c_kill*[*B*] 是 *U* 中(但不在 *B* 中)所有被 *B* 注销的复写语句集合,那么下列方程联系这些定义:

$$
\begin{aligned}
&out[B] = c\_gen[B] \cup (in[B] - c\_kill[B]) \\
&in[B] = \bigcap_{P\text{是 }B\text{的前驱}} out[P] \qquad (B\text{不是初始块}) \\
&in[B_1] = \qquad\qquad (B_1\text{是初始块})
\end{aligned} \tag{9.7}
$$

如果 *c_kill* 和 *c_gen* 分别由 *e_kill* 和 *e_gen* 代替的话,那么方程(9.7)和方程(9.5)一样,所以,方程(9.7)可用算法(9.3)求解。我们给出一个例子揭示复写传播的某些细微差别。

例 9.15 考虑图 9.23 的流图。这里,*c_gen*[B_1] = {x ≒ y},*c_gen*[B_2] = {x ≒ z};*c_kill*[B_1] = {x ≒ z},*c_kill*[B_2] = {x ≒ y} 和 *c_kill*[B_3] = {x ≒ y}。

其他块的 *c_gen* 和 *c_kill* 是 ,由方程(9.7),in[B_1] 也是 。算法 9.3 一遍扫描判断出

$$in[B_2] = in[B_3] = out[B_1] = \{x \fallingdotseq y\}$$

类似地,out[B_2] = ,并且

$$out[B_3] = in[B_4] = out[B_4] = \{x \fallingdotseq z\}$$

最后,in[B_5] = *out*[B_2] ∩ *out*[B_4] =

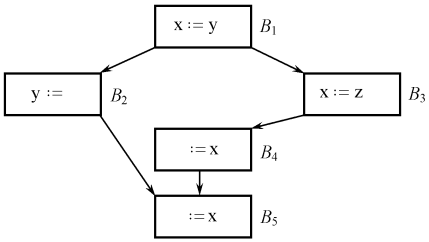


图 9.23 一个流图

我们看出,按算法 9.3 的意义,复写 x ≒ y 和 x ≒ z 都不能到达块 B_5 中 x 的引用,虽然按到达-定值的意义这些 x 都能到达 B_5。这两个复写几乎都不能传播,因为不能用 y(或 z)代替定值 x ≒ y(或 x ≒ z)能到达的所有 x 的引用,仅能做的是将块 B_4 中的 x 用 z 代替,但代码没有改进。

下面详细说明删除复写语句的算法。

算法 9.6 复写传播。

输入 流图 *G*,以及表示到达块 *B* 的定值的 ud 链,还有代表方程 9.7 的解 *in*[*B*],后者即沿着每条路径到达块 B 的复写 x ≒ y 的集合,在这些路径上 x ≒ y 的最后一个出现之后没有对 x 或 y 赋值。我们还需要能表达出每个定值的引用的 du 链。

输出 修改的流图。

方法 对每个复写 *s*:x ≒ y 执行下列步骤:

(1) 根据 du 链,找出该 x 定值能到达的那些 x 引用。

(2) 对(1)找到的每个 x 引用,确定 s 是否在 in[*B*] 中,块 B 是含这个 x 引用的基本块,

而且块 B 中该引用的前面没有 x 或 y 的定值。提醒一下,如果 s 在 in[B] 中,那么 s 是到达块 *B 的惟一* x 定值。

(3) 如果 s 满足 (2) 的条件,则删掉 s,且把 (1) 找出的所有对 x 的引用改成对 y 的引用。

9.4.3 寻找循环不变计算

用 ud 链来寻找循环不变计算,循环不变计算是指只要控制不离开循环,它的值就不改变。9.2 节已讨论过,循环由一组基本块组成,它的首结点是所有其他块的必经块,所以进入循环只能通过首结点。我们还要求从循环中的任何块至少有一路径回到首结点。

如果循环中有赋值 x ≒ y+z,而 y 和 z 所有可能的定值都在循环外面 (包括 y 和/或 z 是常数的特殊情况),那么 y+z 是循环不变计算。因为只要控制不离开循环,每次碰到的 y+z 的值都一样。所有这种赋值可以从 ud 链找到。

识别了 x ≒ y+z 计算的 x 值在循环中不变后,如果循环中有另一语句 v ≒ x+w,其中 w 也只能在循环外定值,那么 x+w 也是循环不变计算。根据这个想法,可以对循环多遍扫描,找出越来越多的循环不变计算。如果有 ud 和 du 链,甚至不需要重复扫描。定值 x ≒ y+z 的 du 链说明 x 的值在哪儿引用,我们只要检查循环中的这些 x 引用是否引用 x 的其他定值 (通过 ud 链)。若某个 x 引用并不引用 x 的其他定值,并且它所在表达式的其他运算对象也是循环不变的,那么该计算也是循环不变计算,同样可以移到循环的前置块中。

算法 9.7 寻找循环不变计算。

输入 由一组基本块构成的循环 *L,对循环 L 中的每个三地址语句有 ud 链可用。*

输出 从控制进入循环 *L 一直到离开 L,每次都计算同样值的三地址语句。*

方法 我们只给出该算法的非形式说明,讲清它的原理。

(1) 把运算对象都是常量 (或其所有的到达-定值都在循环 *L 外*) *的语句标记为"不变"* 语句。

(2) 重复 (3),直到某次重复没有新的语句可标记为"不变"为止。

(3) 给下面的语句标记"不变"——它们先前没有标记,并且所有的运算对象都是下列三种情况之一:

(a) 常量;

(b) 其所有的到达-定值都在循环外;

(c) 只有一个到达-定值,这个定值是循环中已标记为"不变"的语句。

9.4.4 代码外提

找出循环不变语句后,可对其中的一些语句实施代码外提的优化,即把这些语句移到循

环的前置结点。下面三个条件保证代码外提不会改变程序的计算。这些条件没有一个是非要不可的,之所以用这些条件是因为它们易于检查,并且可用到实际的程序中,放宽这些条件是可能的。

语句 s:x ≒ y+z 可以外提的条件是:

(1) 含 s的块是循环所有出口结点的必经结点,出口结点是指那些有后继结点不在循环中的结点。

(2) 循环中没有其他语句对 x 定值。如果 x 是只赋值一次的临时变量,这个条件肯定满足,不必检查。

(3) 除了 s 以外,x 的其他定值都不能到达循环中 x 的引用。如果 x 是临时变量,这个条件一般也满足。

下面三个例子引发上面这些条件。

例 9.16 不加限制地把循环不变计算移到循环外,可能会改变程序的计算,见图 9.24。这个例子引出条件 (1),因为只要不陷入无限循环,在所有出口的必经结点中的语句就一定会执行。

考察图 9.24 (a) 的流图,块 B_2,B_3 和 B_4 形成一个循环,块 B_2 是首结点,块 B_3 的语句 i ≒ 2 显然是循环不变计算。但是由于块 B_3 不是出口 B_4 的必经结点,如果把 i ≒ 2 外提到新的前置块 B_6,如图 9.24 (b) 所示,我们可能改变了块 B_5 中对 j 的赋值,因为块 B_3 可能不被执行。

例 9.17 如果循环中对 x 不止一次赋值,条件 (2) 是需要的。例如,图 9.25 的流图结构同于图 9.24 (a),我们也像图 9.24 (b) 那样建立前置结点 B_6。

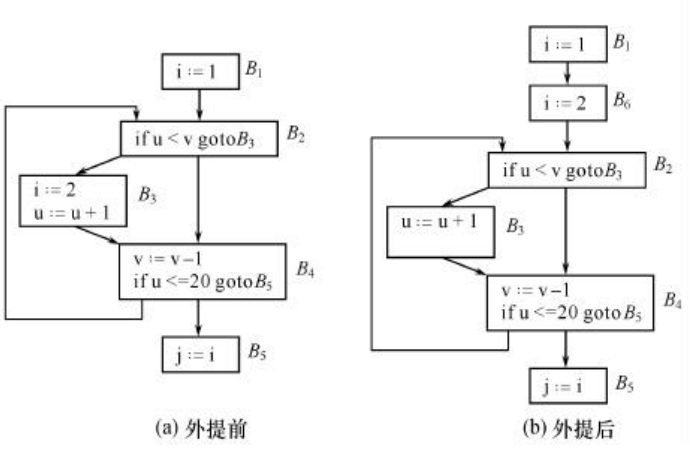


图 9.24 说明条件 (1) 的例子

因为 B_2 是图9.25惟一出口 B_4 的必经块，条件(1)不能阻挡我们把 i := 3 外提到前置块 B_3。我们做了这样的外提后，只要块 B_3 执行，i 的值就是2，这个值会到达块 B_5，即使执行的序列是 $B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2 \rightarrow B_4 \rightarrow B_5$ 时也是这样。可是，若没有外提，这个序列的执行使得 i := 3 到达块 B_5。

例9.18 现在考虑条件(3)。图9.26中，块 B_1 的 i := 1 和块 B_3 的 i := 2 都可以到达块 B_4 的 i 引用，所以不能外提 i := 2 到前置块，因为若 u>=v，到达 B_5 的 k 值会改变。

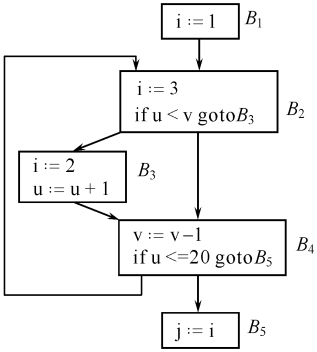


图9.25 说明条件(2)的例子

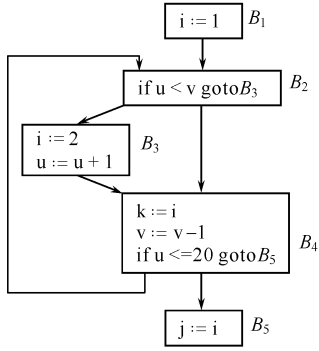


图9.26 说明条件(3)的例子

算法9.8 代码外提。

输入 循环 L，包括ud链信息和必经结点信息。

输出 该循环的一个修改版本，增加了前置块，可能有一些语句外提到前置块中。

方法 (1) 用算法9.7寻找循环不变语句 s：x := y+z。

(2) 对(1)找到的 x 的每个定值语句 s，检查是否满足：

(a) s所在块是 L 所有出口的必经块。

(b) x 在 L 的其他地方没有定值。

(c) L 中 x 的所有引用只有 s 的这个 x 定值才能到达。

(3) 按算法9.7找出的次序，把(1)找出的且满足(2)的三个条件的每个语句移到新的前置块。但是，若 s 的运算对象在 L 中定义（由算法9.7的(3)找出这种 s），那么只有在这种对象的定值语句外提到前置块后，才能外提 s。

上述算法的条件(2a)和(2b)保证在 s 计算的 x 值必定是 L 任何出口的 x 值，当把 s 外提到前置块时，s 仍然是到达 L 任何出口的 x 的定值。条件(2c)保证 L 中任何 x 的引用在外提前后都引用 s 计算的 x 值。

为了明白为什么变换不会增加程序的运行时间，我们只需注意条件(2a)，它保证控制每次进入循环 L 时，s 至少执行一次。代码外提后，它仅在前置块执行一次，控制进入 L 后，它

不再执行。

9.4.5 归纳变量删除

在循环中,若变量 x 值的每次改变都增加或减少某个固定的常量,那么 x 叫做循环的归纳变量,例如由 for i := 1 to 10 开头的循环中的变量 i。

归纳变量的一种常见情况是作为数组的下标,例如 i;其他一些归纳变量,如 t,它的值是 i 的线性函数,作为访问数组的实际偏移。i 经常仅用于测试循环的终止,这时我们可以摆脱 i,用对某个 t 的测试代替它。

如果循环中变量 i 只有形如 i := i± *c* 的赋值,*其中 c* 是常量,那么 i 是我们要找的循环的基本归纳变量。然后再找其他的归纳变量 j,它在循环中仅有一个定值,并且值是某个基本归纳变量的线性函数。

算法 9.9 寻找归纳变量。

输入 有到达-定值信息和循环不变信息(由算法 9.7 获得)的循环 *L*。

输出 一组归纳变量,联系到每个归纳变量 j 的是三元组 (*i*, *c*, *d*),其中 i 是基本归纳变量,*c 和 d* 是常量,在 j 的定值点,j 的值是 *c**i+*d*。我们说 j 属于 i 族。基本归纳变量 i 也属于它自己的族。

方法 (1) 扫描 *L* 的语句,找出所有基本归纳变量。在这里,我们使用循环不变计算的信息。对应每个基本归纳变量的是三元组 (*i*,1,0)。

(2) 寻找 *L* 中只有一个赋值的变量 k,它有下面的形式之一:

k := j* *b*, k := *b**j, k := j/ *b*, k := j± *b*, k := *b*±j

其中 *b* 是常数,j 是基本的或非基本的归纳变量。

如果 j 是基本的,那么 k 在 j 族中,k 的三元组依赖于定义它的指令,例如 k 由 k := j* *b* 定义,那么 k 的三元组是 (*j*, *b*, 0)。其余情况的三元组可类似地定义。

如果 j 不是基本归纳变量,它属于 i 族,那么我们附加的要求是:

(a) 在循环 *L* 中对 j 的惟一赋值和对 k 的赋值之间没有对 i 的赋值。

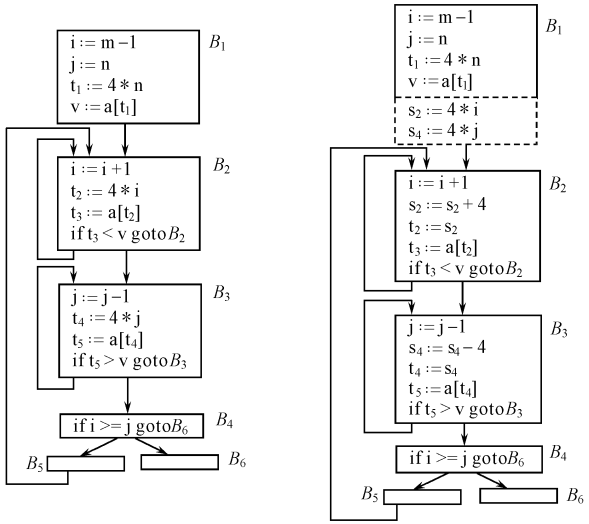
(b) 循环 *L* 外没有 j 的定值可到达 k 的这个定值点。

常见的情况是 k 和 j 属于同一块中的临时变量,它们比较容易检查。对于更一般的情况,如果我们分析 *L* 的流图来决定哪些块(因而哪些定值)位于路径上对 j 的赋值和对 k 的赋值之间,那么到达-定值信息将用于这种检查。

用 j 的三元组 (*i*, *c*, *d*) 和对 k 定值的语句来计算 k 的三元组。例如,定值 k := *b**j 使得 (*i*, *b** *c*, *b** *d*) 作为 k 的三元组。*b** *c* 和 *b** *d* 可以在这种分析时完成计算,因为 *b*, *c* 和 d 都是常数。

一旦找出一族归纳变量,可以修改计算归纳变量的语句,改为用加或减而不是乘。

例9.19　图9.27(a)中由块 B_2 构成的循环有基本归纳变量 i,i 族含 t_2,因为对 t_2 只有一个赋值,该赋值的右部是 4*i。于是 t_2 的三元组是(i,4,0)。同样地,在由块 B_3 构成的循环中,j 是仅有的基本归纳变量,t_4 属于 j 族,其三元组为(j,4,0)。



(a) 变换前　　(b) 变换后

图 9.27　强度削弱

还可以寻找由块 B_2 为首结点的外循环 B_2,B_3,B_4 和 B_5 的归纳变量,i 和 j 是这个较大循环的基本归纳变量 t_2 和 t_4 也是外循环的归纳变量,其三元组仍分别是(i,4,0)和(j,4,0)。

图9.27(b)的流图是把下面算法用到图9.27(a)后得到的。我们在下面讨论这个变换。

算法9.10　用于归纳变量的强度削弱。

输入　循环 L,还有到达-定值信息和由算法9.9计算的归纳变量族。

输出　修改后的循环。

方法　依次考虑基本归纳变量 i,对每个三元组为(i,c,d)的 i 族归纳变量 j:

(1) 建新变量 s,但如果变量 j_1 和 j_2 有同样的三元组,则仅建一个新变量用于两者。

(2) 用 j := s 代替 j 的赋值。

(3) 在 L 中每个赋值 i := i+n *的后面*(n是常数),紧接着它加上

　　s := s+c*n

其中的表达式 c*n *计算得到一个常数,因为* c *和* n 都是常数。把 s 放入 i 族,其三元组为(i,c,d)。

(4) 还必须保证在循环的入口处 s 的初值为 c*i+d,这个初始化可以放在前置块的末

尾,它由

```
s ≒ c*i        /*如果 c为1,则 s ≒ i*/
s ≒ s+d        /*如果 d为0则被省略*/
```

组成,注意 s 是 i 族的归纳变量。

例 9.20 从里向外考虑图 9.27(a)的循环。因为内循环 B_2 和 B_3 的处理非常类似,所以我们仅谈 B_3 的循环。在例 9.19 已知 j 是它的基本归纳变量,t_4 是它的另一个归纳变量,三元组为(j,4,0)。在算法 9.10 的(1),构造新变量 s_4。在(2),赋值语句 $t_4 \doteq 4*j$ 由 $t_4 \doteq s_4$ 代替。在步骤(4),赋值 $s_4 \doteq s_4-4$ 插在赋值 $j \doteq j-1$ 的后面,其中的 -4 是 -1 乘 4 得到。

因为块 B_1 可作为循环的前置块,我们把对 s_4 置初值放在块 B_1 的末尾,块 B_1 含 j 的定值。加上的指令放在块 B_1 中用虚线扩展的部分。

当考虑外循环时,流图如图 9.27(b)所示。变量 i,s_2,j 和 s_4 都是归纳变量。算法 9.10 的步骤(3)已把新建变量分别加入了 i 族和 j 族。为了完成 i 和 j 的删除,需要使用下一个算法。

强度削弱后将会发现有些归纳变量仅用于测试,可以用另外某个归纳变量的测试来代替该归纳变量的测试。例如,若 i 和 t 是归纳变量,t 的值总是 i 值的 4 倍,那么测试 i>=j 等价于 t>=4*j,替换后有可能删除 i。如果 t=-4*i,那么需要同时改变关系算符,因为 i>=j 等价于 t<=-4*j。在下面的算法中,我们考虑乘数为正的情况。算法推广到适用于负数的情况作为练习。

算法 9.11 归纳变量删除

输入 循环 L,带有到达-定值信息,循环不变计算信息(从算法 9.7)和活跃变量信息。

输出 修改后的流图。

方法 (1) 考虑仅用于计算同族中其他归纳变量并且用于条件分支的每个基本归纳变量 i。取 i 族的某个 j,优先取其三元组(i,c,d 中的 c 和 d 尽可能简单的 j(即优先于 c=1 和 d=0 的情况),把每个含 i 的测试改成对 j 的测试。下面我们假定 c 是正的。形式为 if i relop x goto B 的测试(其中 x 不是归纳变量),由

```
r ≒ c*x     /*如果 c等于1,则r ≒ x */
r ≒ r+ d    /*如果 d为0,则省略*/
if j relop r goto B
```

来代替,其中 r 是新的临时变量。if x relop i goto B 的处理类似。如果测试 if i_1 relop i_2 goto B 的 i_1 和 i_2 都是归纳变量,那么检查 i_1 和 i_2 是否都能被代替。最简单的情况是当我们有三元组为(i_1,c_1,d_1)的 j_1 和三元组为(i_2,c_2,d_2)的 j_2,还有 $c_1=c_2$ 且 $d_1=d_2$ 时,那么,i_1 relop i_2 等价于 j_1 relop j_2。在更复杂的情况下,测试的替换可能是没有价值的,因为我们可能要引入两步乘和一步加,而删除 i_1 和 i_2 只可能节省两步。

最后,当被删掉的归纳变量不再引用时,从循环中删去所有对它的赋值。

(2) 现在,考虑由算法9.10引入语句j:=s的每个归纳变量j。首先检查在引入的j:=s和任何j的引用之间有没有对s赋值。通常,j在它被定值的块中引用,因此这个检查可以简化;否则需要用到达-定值信息来实现这种检查。然后我们用引用s代替所有对j的引用,并删去语句j:=s。

例9.21 考虑图9.27(b)的流图,环绕块B_2的内循环包含两个归纳变量i和s_2,但是它们一个也不能删除,因为s_2作为数组a的下标,i作为外循环的测试。同样,环绕块B_3的循环包含归纳变量j和s_4,但是它们也都不能删除。

再让我们把算法9.11用于外循环。当算法9.10建立新变量s_2和s_4时,s_2在i族中,s_4在j族中,如例9.20所讨论的那样。考虑i族,i的惟一引用是在块B_4中用于测试循环终止,所以i是算法9.11步骤(1)选择的删除对象。块B_4的测试包含两个归纳变量i和j,幸好,i族和j族包含的归纳变量分别是s_2和s_4,它们的三元组有同样的常数,因为这两个三元组分别是(i,4,0)和(j,4,0)。于是,测试i>=j可由$s_2>=s_4$代替,使得i和j都被删除。算法9.11的步骤(2)运用复写传播于这些新建的变量,用s_2和s_4分别代替t_2和t_4。

在算法9.9和9.10中,我们允许用循环不变计算代替常量,因此归纳变量j的三元组(i,*c*,*d*)可能包含循环不变计算而不是常量。这些计算必须在循环*L*外的前置块完成。而且,由于中间代码要求每个语句至多一个算符,因此我们必须准备为这种表达式的计算产生中间代码语句。算法9.11的测试替换需要知道乘法常量*c*的符号。基于这一点,把注意力限制在*c*是已知常数的情况是合理的。

习 题 9

9.1 考虑图9.28的矩阵乘程序

```
'begin
    for i := 1 to ndb
        for j := 1 to ndb
            c[i,j] := 0;
    for i := 1 to ndb
        for j := 1 to ndb
            for k := 1 to ndo
                c[i,j] := c[i,j] + a[i,k] * b[k,j]
end
```

图9.28 矩阵乘程序

(a) 假定 a,b 和 c 都是静态分配的,在字节编址的内存中,每个字占 4 个字节,请为图 9.28 的程序产生三地址语句。

(b) 从三地址语句构造流图。

(c) 从每个基本块中删除公共子表达式。

(d) 找出流图中的循环。

(e) 把循环不变计算移出循环。

(f) 找出每个循环的归纳变量,可能的话,删除它们。

9.2 为习题 9.1(b) 的初始流图和 9.1(e) 的最终流图计算到达-定值和 ud 链。

9.3 对图 9.29 的流图,计算:

(a) ud 和 du 链。

(b) 每块末尾的活跃变量。

(c) 可用表达式。

9.4 图 9.29 中有常量合并的可能码?有的话,完成它。

9.5 图 9.29 中有公共子表达式码?有的话,删除它们。

9.6 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。

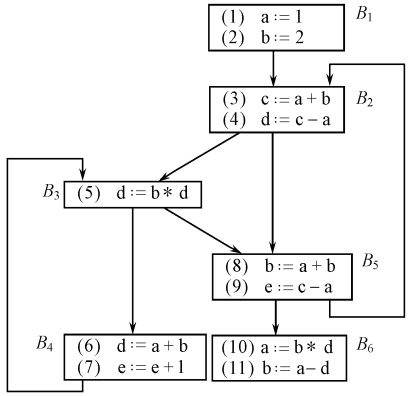


图 9.29 一个流图

9.7 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。

9.8 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 机器上经优化编译后,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（只需要说复写传播、删除公共子表达式等，不需要说怎样完成这些优化的）。

9.9 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。）

9.10 一个 C 语言程序

```
main()
 {
        long i,j;

        while(1) {
        if(j) {i=j;}
        }
}
```

的编译结果如下：

```
        .file "control.c"
        .version "01.01"
gcc2_compiled.:
.text
        .align 4
.globl main
        .type main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        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。

9.11 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, -gp, or -a is used. ...

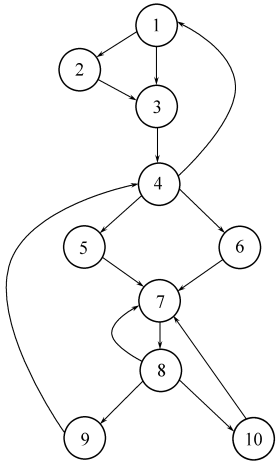
9.12 请利用代码优化的思想(代码外提和强度削弱等),改写下面C语言程序中的循环,得到优化后的C语言程序。

```
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;
        }
    }
}
```

9.13 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。

9.14 识别下面流图中的循环。



第 10 章　编译系统和运行系统

通常,除了编译器外,还需要一些其他工具的帮助,才能得到可执行的目标程序,这些工具包括预处理器、汇编器和连接器等。对于 FORTRAN、Pascal 和 C 来说,这些工具都较简单或明显。了解这些工具有助于我们掌握从源程序到可执行目标程序的实际处理过程,这些知识对于参与大型软件系统的开发是很有用的。本章首先介绍 C 语言的编译系统。

另外,目标代码运行时,还需要一些工具的支撑,如动态连接程序、无用单元收集程序等,这些工具的集合称为运行系统。本章还介绍 Java 语言的运行系统及其无用单元收集程序。

10.1　C 语言的编译系统

除了编译器外,还需要一些其他的工具来建立一个可执行的目标程序。本节以 GNU C 编译系统(简称 GCC 系统)为例,来说明程序设计语言编译系统的一般工作过程。

一个 C 源程序可以分成若干个模块,存储在不同的文件中。C 编译系统对这些源文件分别进行预处理、编译和汇编,形成可重定位的目标文件;然后再利用连接器将这些目标文件和必要的库文件连接成一个可执行的目标文件,即具有绝对地址的机器代码。这一过程可用图 10.1 描绘。

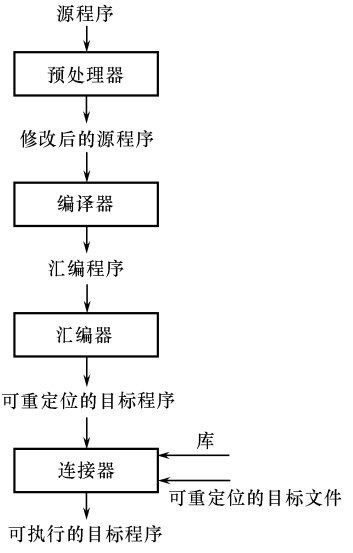


图 10.1　一个语言编译系统

大多数编译系统提供一个驱动程序来调用语言的预处理器、编译器、汇编器、连接器,以支持用户完成从源程序到可执行程序的翻译。在 GCC 系统中,驱动程序的名字是 gcc(或 cc)。

下面结合一个 C 语言的程序实例来讨论 GCC 系统的工作步骤。图 10.2 中的程序由两个文件 main.c 和 swap.c 组成,为便于引用中间的语句,我们增加了行号。在 UNIX(还有 Linux)环境下,键入如下命令可以得到该程序的可执行文件 swap:

gcc -v -o swap main.c swap.c

这里,使用选项-v可以输出该编译系统各步骤执行的命令和执行结果,选项-o紧跟着的字符串指示生成的可执行文件的名字。

main.c

```
(1)  #if 1
(2)  int buf[2];
(3)  #else
(4)  int buf[2] = {10,20};
(5)  #endif
(6)  void swap();
(7)  #define A buf[1]
(8)  int main()
(9)  {
(10) scanf("%d,%d",buf,buf+1);
(11) swap();
(12) printf("%d,%d",A,buf[1]);
(13) return 0;
(14) }
```

swap.c

```
(2)  extern int buf[2];
(3)  int *bufp0=buf;
(4)  int *bufp1;
(5)  void swap()
(6)  {
(7)  int temp;
(8)  bufp1=buf+1;
(9)  temp=*bufp0;
(10) *bufp0=*bufp1;
(11) *bufp1=temp;
(12) }
```

图10.2 main.c和swap.c组成的程序

10.1.1 预处理器

gcc首先调用预处理器cpp,将源程序文件翻译成一个ASCII中间文件,它是经修改后的源程序。图10.3是main.c经预处理后生成的中间文件main.i。

预处理器产生编译器的输入,它实现以下功能:

(1) 文件包含

预处理器可以把源程序文件中的包含声明(#include)扩展为程序正文。例如,当源程序文件中含有语句

#include <stdio.h>

时,预处理器会在系统标准路径下搜索stdio.h,再用文件stdio.h中的内容来代替这个语句。

(2) 宏展开

C程序中可以使用#define来定义宏,一个宏定义给出一段C代码的缩写。预处理器将源程序文件中出现的对宏的引用展开成相应的宏定义,这一过程称为宏展开。例如,main.c的第(7)行为宏A的定义,第(12)行中的A是对该宏的引用。在预处理后产生的main.i中,

宏 A 的定义转换成一个空行,对宏 A 的引用则展开成 buf[0]。

```
(1)  #1 main.c
(2)
(3)  int buf[2]
(4)
(5)
(6)
(7)  void swap();
(8)
(9)  int main(){
(10)  {
(11) scanf("%d,%d",buf,buf+1);
(12) swap();
(13) printf("%d,%d",buf[0],buf[1]);
(14) return 0;
(15) }
```

图 10.3 main.i 的内容

(3) 条件编译

预处理器根据#if 和#ifdef 等编译命令及其后的条件,将源程序中的某部分包含进来或排除在外。通常把排除在外的语句转换成空行。

显然,实现一个这样的预处理器并不困难。

有些语言的预处理器用于增强老的语言,使之包含现代的控制结构和数据类型。从增强语言到老语言的翻译由这样的预处理器完成。

10.1.2 汇编器

GCC 系统的编译器 cc1 产生汇编代码,main.i 被编译成的 ASCII 汇编文件 main.s,见图 10.4。这些汇编代码由汇编器进一步处理。

最简单的汇编器对输入进行两遍扫描。在第一遍,汇编器扫描输入,将表示存储单元的所有标识符都存入符号表,并分配地址。在第二遍,汇编器再次扫描输入,把每个操作码翻译成机器语言中代表那个操作的位串,并把代表存储单元的每个标识符翻译成符号表中为这个标识符分配的地址。

```
    .file  main.c
    .version    01.01
gcc2_compiled.:
.section .rodata
.LC0:
    .string  "%d, %d"                    —— scanf和 printf中使用的格式串
.text
    .align 4                             —— 按4字节对齐
.globl main
    .type main,@function                 —— 本模块定义的函数 main
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
    pushl $buf+4                         —— 实参 buf+1入栈
    pushl $buf                           —— 实参 buf入栈
    pushl $.LC0                          —— 格式串指针入栈
    call scanf                           —— 调用函数 scanf
    addl $12,%esp                        —— 栈顶指针恢复到参数压栈前的位置
    call swap                            —— 调用函数 swap
    movl buf+4,%eax                      —— 取实参 buf[1]到寄存器
    pushl %eax                           —— 实参 buf[1]入栈
    movl buf,%eax                        —— 取实参 buf[0]到寄存器
    pushl %eax                           —— 实参 buf[0]入栈
    pushl $.LC0                          —— 格式串指针入栈
    call printf                          —— 调用函数 printf
    addl $12,%esp                        —— 栈顶指针恢复到参数压栈前的位置
    xorl %eax,%eax
    jmp .L1
    .p2align 4,,7
.L1:
    leave
    ret
.Lfe1:
    .size main,.Lfe1-main
    .comm   buf,8,4                      —— 本模块定义的未初始化全局变量 buf
                                         —— 占8字节,按4字节对齐
    .ident   "GCC:(GNU) egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)"
```

图10.4 汇编文件main.s

如果一个汇编代码文件中有外部符号的引用,如果汇编器生成的是可重定位的目标文件,那么汇编器的工作将变得略微复杂。在 10.1.4 节,当知道了目标文件的格式后,要实现一个汇编器并不是一件困难的事情。另外,一遍扫描完成汇编代码到可重定位目标代码的翻译也是完全可能的。

在 GCC 编译系统中,要想得到源程序被编译成的汇编代码,只要加编译选项 -S 就可以。例如,用

```
gcc -S main.c
```

可以得到汇编文件 main.s。使用汇编器 as

```
as -o main.o main.s
```

可以将 main.s 汇编成可重定位目标文件 main.o。

10.1.3 连接器

我们已经知道,汇编器或编译器输出的机器代码称为目标模块或目标文件,它有两种形式:

(1) 可重定位的目标文件。它包含二进制代码和数据,可以和其他可重定位目标文件组装成一个可执行的目标文件。

(2) 可执行的目标文件。它包含二进制代码和数据,可以直接被复制到内存并被执行。

实际另外还有一种形式,即共享目标文件。它是一种特殊的可重定位目标文件。可以在装入程序或运行程序时,动态地装入共享目标文件到内存并将它和程序连接。

技术上,一个目标模块是一个字节序列,而一个目标文件则是一个以文件形式存储在外部存储器上的目标模块。不过,本书将不加区分地使用这两个术语。

连接是一个收集、组织程序所需的不同代码和数据的过程(它们可能在不同的目标模块中),以便程序能被装入内存并被执行。连接可以在将源代码翻译成机器代码的编译时候完成,也可以在程序装入内存并执行的装入时完成,甚至可以在程序运行时完成。

静态连接器负责将多个可重定位目标文件组成一个可执行目标文件(也可以组成一个可重定位目标文件);动态连接器则支持在内存中的可执行程序在执行时与共享目标文件进行动态的连接。有些系统将装入可执行程序时与共享目标文件进行的连接也称为动态连接。

如果这些目标文件是以有用的方式组在一起的,那么它们之间就会出现一些外部引用,即一个文件中的代码引用另一文件中的存储单元。这种引用可以是定义在一个文件而使用在另一个文件的数据单元,或者是入口点出现在一个文件而调用点出现在另一个文件的函数。

在连接器 ld 的上下文中,一个重定位模块 M 定义和引用的符号通常分成三类:

(1) 全局符号。是指那些在模块M中定义,可以被其他模块引用的符号。它包括模块M中定义的非static属性的函数和全局变量。

(2) 局部符号。是指那些在模块M中定义,且只能在本模块中引用的符号。它包括模块M中定义的有static属性的函数和全局变量。

(3) 外部符号。是指那些由模块M引用并由其他模块定义符号。

这样,连接器主要完成以下两个任务:

(1) 符号解析(symbol resolution)。连接器识别各个目标模块中定义和引用的符号,为每一个符号引用确定它所关联的一个同名符号的定义。

(2) 重定位。编译器和汇编器产生的代码节和数据节分别都是从零地址开始。连接器按如下方式来重定位这两节:将每一个符号定义关联到一个内存位置,然后修改所有对这些符号的引用,以使它们指向相关联的内存位置。

在10.1.4节明白了目标文件的格式后,实现连接器不是一件困难的事情。

10.1.4 目标文件的格式

目标文件格式随系统不同而不同。来自Bell实验室的第一个UNIX系统使用a.out格式,至今UNIX下的可执行文件仍被缺省地命名为a.out。System V UNIX的早期版本使用COFF(Common Object File Format)格式。Windows NT使用COFF的一个变体,称为PE(Portable Executable)格式。现代UNIX系统,如Linux、System V UNIX的后期版本、BSD UNIX变体和Sun Solaris,都使用UNIX的ELF(Executable and Linkable Format)格式。这里仅讨论UNIX使用的ELF文件格式。

图10.5为典型的ELF可重定位目标文件格式。ELF头(header)开始于一个16字节的序列,它描述了字的大小和产生此文件的系统的字节次序。ELF头的其余部分包含的信息用于连接器分析和解释目标文件,其中包括:ELF头的大小、目标文件的类型(可重定位、可执行或共享等)、机器类型(如IA32)、节头表(section header table)在本目标文件中的偏移、节头表中条目的大小和数量。

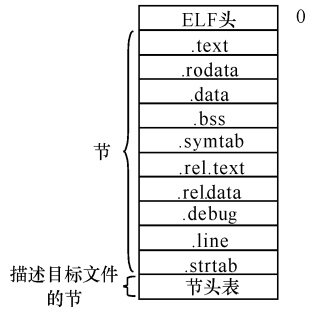


图10.5 ELF可重定位文件格式

节头表描述目标文件中各节的位置和大小。在ELF头和节头表之间是节本身。典型的ELF可重定位目标文件包含以下各节。

(1) .text:被编译程序的机器代码。

(2) .rodata:诸如printf语句中的格式串和switch语句的跳转表(见7.4.4节)等只读数

据。

⑶ .data:已初始化的全局变量,如 swap.c 中的 bufp0。

⑷ .bss:未初始化的全局变量,如 main.c 中的 buf 及 swap.c 中的 bufp1。该节在目标文件中不占实际的空间,只是一个占位符。目标文件格式区分已初始化和未初始化变量是为提高空间的利用率:在目标文件中,未初始化变量不必占用实际外部存储器的任何空间(历史上,bss 是 better save space 的缩写)。在有些汇编语言中,已经用.comm 代替了.bss。

⑸ .symtab:记录在该模块中定义和引用的函数和全局变量的信息的符号表。和编译器内部的符号表不同的是,.symtab 中不包含局部变量。

例 10.1 下面是用 readelf 工具显示的 main.o 的符号表的后 5 个条目。没有显示的前 9 个条目是供连接器内部使用的局部符号。

Num:	Value	Size	Type	Bind	Ot	Ndx	Name
9:	0	66	FUNC	GLOBAL	0	1	main
10:	4	8	OBJECT	GLOBAL	0	COM	buf
11:	0	0	NOTYPE	GLOBAL	0	UND	scanf
12:	0	0	NOTYPE	GLOBAL	0	UND	swap
13:	0	0	NOTYPE	GLOBAL	0	UND	printf

在符号表中,Value 域表示符号的地址:对于可重定位目标文件,它是相对于定义该对象的节的起始处的偏移;对于可执行目标文件,它是绝对的运行地址。Size 域表示该对象的字节数。Type 域表示符号的类型,一般是数据对象(OBJECT)或函数(FUNC)。Bind 域表明符号是局部的(LOCAL)、全局的(GLOBAL)还是外部的(EXTERN)。Ndx 域指示与该符号关联的节,它的值是节头中的索引号。除了节头表中记录的节外,还有 ABS、UND 和 COM 三个特殊的伪节。其中,ABS 指不需要重定位的符号;UND 指被该模块引用但没有在该模块中定义的符号;COM 是未初始化的数据对象。对于 COM 符号,Value 域表示值的对齐方式,Size 域表示其最小的字节数。

在该例中,第 9 个符号 main 是位于.text 节(Ndx=1)、偏移为 0 的 66 字节的函数;第 10 个符号 buf 是位于.bss 节、按 4 字节对齐的 8 字节对象;随后是引用的 3 个外部符号 scanf、swap 和 printf。

⑹ .rel.text:.text 节中需要修改的单元的位置列表。当连接器将该目标文件和其他目标文件连接时需要这些信息,它包括任何调用外部函数或引用全局变量的指令。

⑺ .rel.data:用于被本模块引用或定义的全局变量的重定位信息。通常,任何要初始化的全局变量,若它的初值为某全局变量或外部函数的地址,则它的值需要修改。

⑻ .debug:用于调试程序的调试符号表,它包含在源程序文件中定义的局部变量和类型定义、在源程序文件中定义和引用的全局变量,以及最初的源文件等条目。只有用 -g 选

项调用gcc才会出现此节。

(9) .line：源程序文件和.text节中的机器指令之间的行号映射。该节仅在用－g选项调用gcc时才会出现。

(10) .strtab：一组有空结束符的串构成的串表。它用于保存.symtab节和.debug节的符号表中的名字和节头表中节的名字。

10.1.5 符号解析

连接器需要将每个符号引用正确地与来自输入的可重定位模块的符号表中的一个符号定义相关联，从而确定各个符号引用的位置。在编译时，当遇到当前源文件没有定义的符号时，它假定该符号在其他某个模块中定义，并为该符号产生一条符号表条目（如main.o符号表中的swap符号），把它留给连接器处理。如果连接器在所有输入模块中都找不到被引用符号的定义，则打印错误消息并结束连接。

解析全局符号的引用还是有点棘手，因为同一个符号可能被多个目标模块定义。此时，连接器必须报告一个错误，或者按某种方式选择其中一个定义而放弃其余的定义。UNIX系统中采用的方法涉及到编译器、汇编器和连接器之间的合作，它会给粗心的程序员引入一些莫名其妙的错误。

在编译时，编译器向汇编器输出的全局符号区分为强和弱两种，汇编器隐式地将这些信息编码在可重定位目标文件的符号表中。函数和已初始化的全局变量为强符号；未初始化的全局变量为弱符号。对于图10.2的程序，main、bufp0和swap为强符号，buf和bufp1为弱符号。UNIX连接器使用如下规则处理多重定义的符号：

规则1　不允许有多重的强符号定义。

规则2　出现一个强符号定义和多个弱符号定义时，选择强符号的定义。

规则3　出现多个弱符号定义时，选择任意一个弱符号的定义。

比如，若将图10.2的swap.c的第1行“extern int buf[2];”改为“int buf[2] = {11, 21};”并存入swap1.c，再将main.c的第1行“#if 1”改为“#if 0”并存入main1.c，那么试图编译和连接main1.c和swap1.c时，连接器将报告错误信息，因为强符号buf被多次定义（规则1）。

若试图将swap1.c和原先的main.c进行编译和连接生成可执行文件时，由于main.c中定义的buf为弱符号，这时连接器将隐含地选择在swap1.c中定义的强符号buf，从而顺利地生成可执行文件（规则2）。

再将swap.c的第1行中的extern去掉并存入swap2.c中，然后编译和连接main.c和swap2.c，这时也能生成可执行文件（规则3）。

不过，若将swap2.c中buf的类型由int改为double并存入swap3.c，再编译和连接main.c和swap3.c时，虽能生成可执行文件（规则3），但会因为多重定义的两个弱符号的类型不相

容而给出警告信息。并且,由于这个类型不相容,生成的目标程序执行时,buf 数组元素的值和上一步得到的目标程序执行时的值不一样。

规则 2 和规则 3 的应用可能会产生一些难以理解的不会被捕获的错误,尤其是在多重定义的符号具有不同类型的时候。为避免这样的问题,程序员可以用适当的编译选项(如—— warn-common 选项)调用连接器,以获得解析多重全局符号定义的警告信息。

10.1.6 静态库

到目前为止,我们一直假设连接器读一组可重定位目标文件,将它们连接并输出一个可执行文件。事实上,所有的编译系统都提供一种机制,将相关的可重定位目标模块打包成一个文件,称为静态库。静态库可以作为连接器的输入被多次使用。当建立可执行文件时,连接器仅复制库中被应用程序引用的模块。

考虑 ANSI C,它定义了一个范围极广的标准 I/O、串操作和整型数学函数的集合,如 printf、strcpy、atoi 等。它们被编译打包在 libc.a 库中,可以供所有的 C 程序使用。ANSI C 还定义了浮点型的数学函数集,如 libm.a 库中的 sin 和 cos。现在思考编译器开发者不用静态库将这些函数提供给用户的几种不同方法。

一种方法是编译器认可对标准函数的调用并直接产生相关的代码。Pascal 语言使用这种方法,为程序员提供了少量的标准函数。对于 C 语言,这种方法是不可行的,因为 C 语言的标准定义了大量的标准函数,这样做会显著增加编译器的复杂性。当每次增加、删除或修改标准函数时,都需要发行新的编译器版本。不过对应用程序员来说,这种方法相当方便,因为这些标准函数总是可用的。

另一种方法是将 C 语言的所有标准函数放在一个可重定位目标文件中,如 libc.o,应用程序员可以将它连接并生成自己的可执行文件。这种方法的好处是标准函数的实现不再是编译器实现的一部分,而对程序员来说仍然相当方便。不过,一个严重的缺点是系统中每一个可执行文件都可能包含全部标准函数集的副本,这将极大地浪费外部存储器空间。更糟的是,每一个正在运行的程序在内存中都可能包含这些函数的副本,这将极大地消耗内存资源。另一个严重的缺点是,任何对标准函数的修改,不论是多么小的修改,都需要库开发者重新编译整个库的源文件,这种耗时的操作将使标准函数的开发和维护变得复杂。

可以通过为每个标准函数创建一个可重定位文件,并按照大家熟知的目录存储它们来部分地解决这些问题。可是这种方法要求应用程序员显式地将有关的目标模块连接到他们的可执行文件中,这一过程很容易出错并且耗时。

静态库可以用来解决上述各种方法存在的问题。可以把相关的函数分成若干源文件,分别编译,然后把生成的目标模块打包成一个静态库文件。应用程序可以通过在命令行中指定该库文件名来使用库中定义的任何函数。

在连接时,连接器将只复制被程序引用的目标模块,这就减少了可执行程序在外部存储器和内存中的大小。另一方面,应用程序员的编译命令只需要把用到的库文件的名字包含进来就可以了。事实上,gcc会自动地将libc.a等库传递给连接器,不需要应用程序员显式地指明。不过在直接用ld连接目标模块时,必须显式地给出被引用的所有库文件,否则连接器不能正确地完成连接。

在UNIX系统中,静态库是以一种特殊的文件格式,即档案文件(archive),存储在外部存储器中。一个档案文件是一个带档案头的被连接的目标文件的集合,档案头描述了每一成员目标文件的大小和位置。档案文件用.a后缀表示。例如,可以用两个命令将swap.c编译并打包成一个自己的库mylib.a:

```
gcc -c swap.c
ar rcs mylib.a swap.o
```

其中编译选项-c表示生成目标文件,不进行连接。

这时可以编译main.c并将它和mylib.a连接,建立可执行程序swap1:

```
gcc -static -o swap1 main.c /usr/lib/libc.a mylib.a
```

其中,-static参数要求连接器建立一个完全连接的可执行目标文件,它可以直接被装入内存运行,不需要任何进一步的连接。

图10.6总结了带有-static参数的连接器的活动。当连接器运行时,它确定由swap.o定义的swap符号被main.o引用,因此它将swap.o复制到可执行文件swap1中。而mylib.a中的其他成员模块(如果有的话),由于其定义的符号没有被引用,则不被连接器复制到可执行文件中。连接器同样复制来自libc.a的printf.o模块、scanf.o模块和一些来自C运行系统的其他模块。

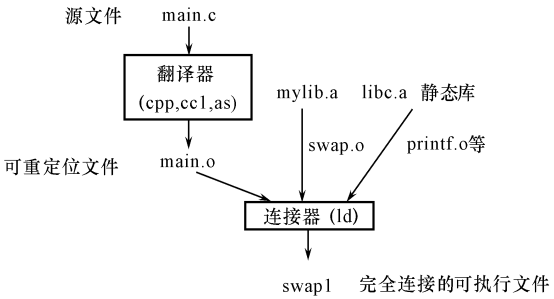


图10.6 和静态库连接

静态库一方面是一种有用的基本工具,另一方面它们也让程序员感到混乱。这是因为,UNIX连接器使用静态库来解析外部引用,在符号解析阶段,连接器按照可重定位的目标文

件和档案文件在 gcc 或 ld 命令行上出现的次序从左到右扫描它们。命令行中档案文件和目标文件的次序相当重要。如果定义一个符号的档案文件在命令行中出现在引用该符号的目标文件之前,则这种引用可能不被解析,导致连接将失败。

使用档案文件的一般规则是将它们放在命令行的最后。如果不同档案文件的成员是相互独立的,则由于没有一个档案文件的成员引用其他档案文件的成员定义的符号,这些档案文件可以按任何次序放在命令行的最后。如果这些档案文件之间相互不独立,则它们必须按一定的次序出现在命令行中,使得对每个被一个档案文件中的成员引用的外部符号 *s*,至少有定义 *s* 的一个档案文件在命令行中出现在对 *s* 的引用之后。

10.1.7 可执行目标文件及装入

图 10.7 概括了典型 ELF 可执行文件中的信息种类。它与可重定位目标文件格式类似。ELF 头描述文件的整体格式,它包含程序的入口点,即当程序运行时要执行的第一条指令的地址。对 .text、.rodata 和 .data 节,除了已经被重定位成最后运行时的内存地址以外,它们与可重定位目标文件中对应的节类似。.init 节定义一个称为 _init 的小函数,它由程序的初始化代码调用。由于可执行程序已被完全连接,因此它不需要与可重定位文件的 .rel.text、.rel.data 类似的可重定位节。

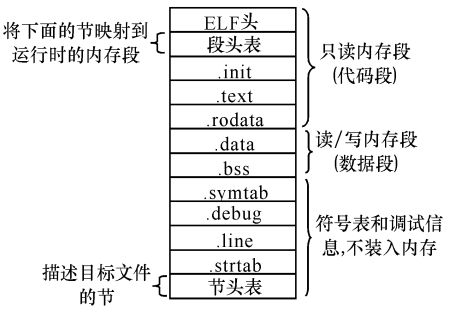


图 10.7 典型的 ELF 可执行目标文件

ELF 可执行文件被设计成易于装入内存,可执行文件中相邻的块被映射到相邻的内存段(*segment*)中。这种映射关系在段头表中描述。

为运行一个可执行程序 swap,可以在 UNIX shell 的命令行上敲入它的名字:

```
./swap
```

由于 swap 不是内部的 shell 命令,shell 假定 swap 是一个可执行目标文件,它通过调用叫做装载器的驻留在内存中的操作系统代码来运行 swap。任何 UNIX 程序都可以通过调用 execve 函数来调用装载器。装载器将可执行目标文件从外部存储器复制到内存,然后通过跳转到程序的入口点来运行程序。复制程序到内存中并运行程序的过程被称为装入。

每一个 UNIX 程序都有一个和图 10.8 类似的运行时内存映像。在 Linux 系统中,代码段一般总是从 0x08048000 地址开始。数据段从下一个按 4 KB 对齐的地址开始。运行时的堆跟在可读写段后,从第一个按 4 KB 对齐的地址开始,并通过调用 malloc 库函数向上增长。从 0x40000000 地址处开始的段被保留用于共享库。用户栈总是从 0xbfffffff 地址开始并向较

低的内存地址增长。位于栈之上,从0xc0000000地址开始的段被保留用于操作系统驻留内存部分的代码和数据,即操作系统内核。

当装载器运行时,它创建如图10.8的内存映像。在可执行目标文件的段头表的指导下,装载器将它的块复制到内存中的代码段和数据段。接着,装载器跳转到程序的入口点——它总是_start符号的地址。_start地址处的启动代码定义在目标文件crt1.o中,它对于所有的C程序都一样。_start完成用户程序运行前的准备,它调用.text和.init节中的初始化例程等后,调用应用程序的main函数,开始执行用户的C代码。

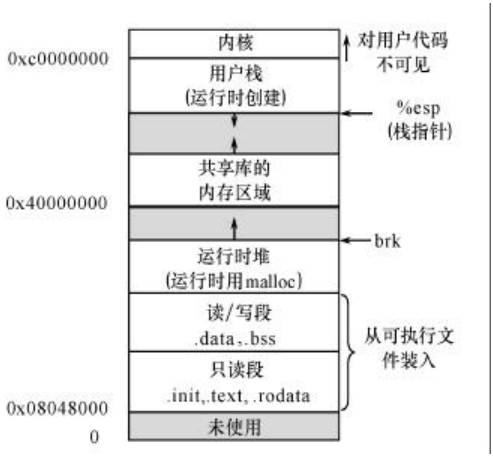


图10.8 Linux运行的内存映像

注意,这里描述的装入过程从概念上来说是正确的,只有在理解了进程、虚拟内存和内存分页等概念后,才能真正了解装入过程是如何工作的。

10.1.8 动态连接

与所有软件一样,静态库需要周期性地维护和更新。如果应用程序员想要使用库的最新版本,他们必须设法知道库已经改变,然后显式地将他们的程序和被更新的库重新连接。另一个问题是,几乎每一个C程序都使用如printf和scanf这样的标准I/O函数。在运行时,这些函数的代码被复制在每一个正在运行的进程的text段中。在一个运行有50～100个进程的典型系统上,这会显著地浪费内存系统资源。

共享库是弥补静态库的缺点所进行的一场变革。一个共享库,也称共享目标文件,它在运行时可以装到任意的内存位置并和内存中的程序连接。这一过程称为动态连接,它由动态连接器执行。在UNIX系统上,共享库一般以.so为后缀。Microsoft操作系统称共享库为动态连接库,一般以.dll为后缀。

共享库以两种不同的方式被共享。第一,在任何给定的文件系统中,对每个库,正好只存在一个 .so 文件,该 .so 文件中的代码和数据被所有引用该库的可执行目标文件所共享,这与静态库的内容被复制和嵌入到引用它的可执行目标文件中的方式相反。第二,运行时,共享库的 .text 节在内存中的一个副本可以被正在运行的不同进程共享。

图 10.9 总结了图 10.2 程序的动态连接过程,这里首先将源文件 swap.c 编译、连接成自己的共享库 mylib.so:

```
gcc -shared -fPIC -o mylib.so swap.c
```

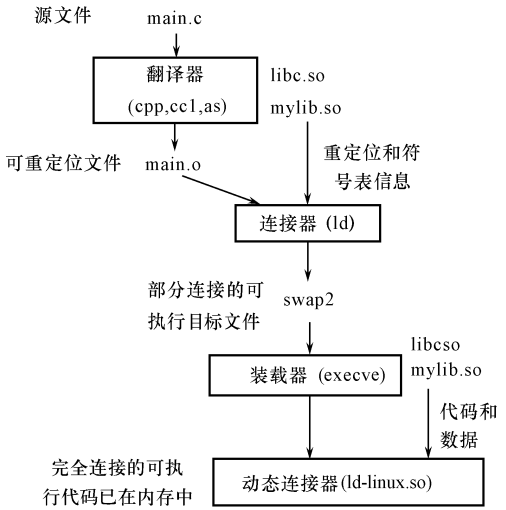


图 10.9 和动态库连接

-fPIC 选项指示编译器产生位置无关的代码(指无须连接器的修改而能装入到内存任意地址执行的代码),-shared 选项指示连接器创建一个共享的目标文件。

一旦创建了 mylib.so 共享库,就可以将它和 main.c 连接成可执行程序 swap2:

```
gcc -o swap2 main.c ./mylib.so
```

动态连接的基本思想是在可执行文件被创建时静态地做一部分连接准备工作,然后在程序被装入时动态地完成连接过程。需要注意的是,mylib.so 中的代码节和数据节并没有被复制到可执行目标文件 swap2 中。连接器只是复制一些重定位信息和符号表信息,它们用以在运行时解析对 mylib.so 中代码和数据的引用。

当装载器装入和运行可执行程序 swap2 时,它使用 10.1.7 节讨论的技术,装入已被部分地连接的可执行目标文件 swap2。swap2 包含有 .interp 节,该节包含动态连接器的路径名,如 Linux 系统下的 ld-linux.so,它本身是个共享对象。装载器通常装入和运行动态连接器,而

不是将控制直接传给应用程序。

动态连接器接着完成连接任务：

(1) 把libc.so的文本和数据装入内存并进行重定位，在IA32/Linux系统上，共享库被装载在起始地址为0x40000000的地方（见图10.8）。

(2) 把mylib.so的文本和数据装入内存并进行重定位。

(3) 重定位swap2中任何对libc.so或mylib.so定义的符号的引用。

最后，动态连接器将控制传递给应用程序。此后，共享库的位置被确定，并在该程序的执行期间不再改变。

上面介绍了共享库的一种装入方式，即在应用程序被装入时，动态连接器装入和连接共享库。还有另外一种方式，即应用程序在运行过程中通过显式的函数调用请求动态连接器装入和连接共享库。两种方式的装入和连接过程本质上没有什么区别，我们不再介绍。

10.1.9 处理目标文件的一些工具

在UNIX系统中，有很多工具可以用来帮助你理解和处理目标文件。尤其是，GNU的binutils包特别有用，可以运行在每一种UNIX平台上。下面是一些工具的名称和功能简介。

ar	创建静态库，插入、删除、罗列和提取成员。
strings	列出包含在目标文件中的所有可打印串。
strip	从一个目标文件中删除符号表信息。
nm	列出一个目标文件的符号表中定义的符号。
size	列出目标文件中各段的名字和大小。
readelf	显示目标文件的完整结构，包括编码在ELF头中的所有信息。它包括了size和nm的功能。
objdump	所有二进制工具之母，可以显示目标文件中的所有信息。其最有用的功能是反汇编.text节中的二进制指令。

UNIX系统还提供ldd程序用来处理共享库：

ldd	列出可执行目标文件在运行时需要的共享库。

10.2 Java语言的运行系统

一般的高级语言程序如果要在不同的平台上运行，至少需要编译成不同的目标代码。随着互联网的流行，不同类型的计算机通过网络共享数据和其他计算资源，人们希望同一版本的程序能够在多个不同的平台上运行。Java语言正是顺应这样的潮流而诞生的一种跨平

台的编程语言。

Java 虚拟机技术则是实现 Java 平台无关性特点的关键。Java 源程序被编译成与任何计算机系统结构都无关的中间代码——Java 虚拟机语言(简称 JVML),任何机器只要安装了 Java 运行系统就能够运行这种中间代码。Java 虚拟机是一种抽象的机器,在实际的计算机上通过软件模拟来实现,Java 运行系统就是 Java 虚拟机的一个实现。本书将不加区分地使用运行系统和虚拟机这两个概念。Java 虚拟机有自己的指令系统,也有自己的存储器区域。它屏蔽了不同操作系统和机器设备的区别,在运行的 JVML 程序和底层的硬件与操作系统之间建立了一个缓冲区。JVML 程序只需要与虚拟机交互,不需要关心底层的硬件和操作系统。

用虚拟机来实现一种程序设计语言的思想并不是 Java 首创的,在 Java 出现之前,UCSD Pascal 系统就已在一种商业产品 P-Code 机器中用到了这一思想。

10.2.1 Java 虚拟机语言简介

Java 虚拟机用以支持 JVML 这种中间语言,因此需要对 JVML 的一些概念和术语进行简要介绍,以更好地理解虚拟机。

Java 程序首先由 Java 编译器把它编译成字节码,也就是 JVML 程序,并被放置到后缀名为".class"的文件中。每个 class 文件是一个 8 位字节流,它包含一个 Java 类或接口的信息。由一张符号表和各方法的字节码序列以及其他辅助信息组成。下面我们介绍 class 文件中最重要的几个部分。

(1) 常量池(constant pool)

常量池包含了在该 class 文件结构及其子结构中引用的各种类型(字符串、浮点等)的常量,常量池的功能类似于传统程序设计语言中的符号表,但是它比通常的符号表包含更多的信息。

(2) 类成员信息

一个 Java 类的成员信息放在两个长度可变的表中:域信息表和方法信息表。

(3) JVML 指令序列

一条 JVML 指令由一个字节的操作码以及若干个供该操作使用的操作数构成。Java 虚拟机上同样也有运行数据区,每个线程都有一个运行栈,该栈保存局部变量和计算的中间结果,并参与方法的调用和返回。所有线程共享的堆用来动态分配对象。JVML 指令描述的就是在这样的抽象机上进行的操作。图 10.10 中展示了一个 JVML 程序的例子。

Java源程序中的方法：

```
int calculate (int i) {
    int j =2 ;
    return ((i+j) * (j-1)) ;
}
```

对应的字节码程序：

```
int calculate (int i)
iconst_2                    —— 在Java栈中压入常数2
istore_2                    —— 将栈顶常数存放到局部变量2(j)中
iload_1                     —— 将局部变量1(参数i)压入栈顶
iload_2                     —— 将局部变量2(参数j)压入栈顶
iadd                        —— 栈顶两整数相加,结果压栈
iload_2                     —— 将局部变量2(参数j)压入栈顶
iconst_1                    —— 在Java栈中压入常数1
isub                        —— 栈顶两整数相减,结果压栈
imul                        —— 栈顶两整数相乘,结果压栈
ireturn                     —— 将栈顶元素返回
```

图10.10 JVML程序示例

10.2.2 Java虚拟机

Java虚拟机一般由以下几个部分构成：类装载器(字节码验证器)、解释器或/和编译器、还有包括无用单元收集器(garbage collector)和线程控制模块在内的运行支持系统，另外还有一些标准类和应用接口的class文件库。其组成情况如图10.11所示。

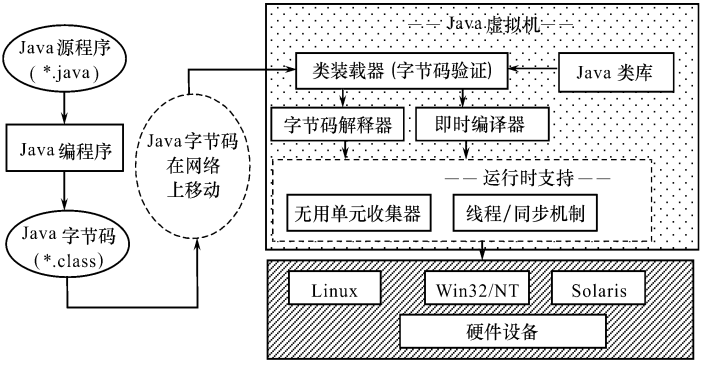


图10.11 Java的实现环境

首先运行 Java 编译器把 Java 应用程序编译成字节码程序。字节码程序可以在网络上传送,在另一台机器的 Java 虚拟机上运行。Java 虚拟机执行字节码的过程可以分为三步:代码的装入、代码的验证和代码的执行。

代码的装入由类装载器完成,类装载器负责装入程序运行时需要的所有代码,其中包括程序代码中用到的所有类。随后,被装入的代码由字节码验证器进行安全性检查,以确保代码不违反 Java 的安全性规则,字节码验证器还可以发现操作数栈溢出、非法数据类型转换等多种错误,通过验证之后,代码就可以提交运行了。

Java 字节码的运行有两种方式:解释执行方式和即时编译(just-in-time compilation)方式。解释执行的缺点是太过简单而且速度很慢,早期的 SUN JDK 解释器要比类似的 C++ 代码慢 5～30 倍。即时编译方式是由即时编译器先将字节码编译成本地机器代码之后再执行。即时编译的方法能够产生质量较高、执行速度较快的代码,但需要花费额外的编译时间。需求驱动是即时编译的另一特点,一个方法直到被调用时才将字节码翻译成机器代码,当一个方法被调用两次以上时,机器代码的执行效率便足以补偿编译耗费的时间。

Java 虚拟机还需要给字节码的执行提供其他运行时的支持。无用单元收集器用来管理所有线程共享的运行时的数据空间。在 Java 虚拟机的逻辑组件中,运行时数据空间包括 Java 栈、堆、方法区(method area)、常量池。其中 Java 栈是每个 Java 虚拟机线程私有的,与线程同时创建同时结束。而堆、方法区、常量池则是所有线程共享的。堆是从中分配所有类实例和数组的运行时的数据区。Java 对象从不被显式地回收,无用单元收集器自动地将程序中不再用到的单元回收。方法区类似于传统语言的代码存储区,如 UNIX 进程中的.text 段,它存储每个类的公用数据和方法代码。方法区逻辑上是堆的一部分,也由无用单元收集器来管理,但是一些简单的实现可以选择不回收它。常量池是各个 class 文件中常量池的运行时的表示,其内容包括从编译时已知的数值和文字到必须在运行时解析的方法和域引用。常量池是方法区的一部分。我们将会在 10.3 节中介绍无用单元收集技术。

Java 语言支持多线程,对于多线程的应用而言,线程调度和同步支持是多线程协同工作正确性的重要保证。在典型的商务应用中,同步操作占了相当大的部分,因此高效地实现同步也是高性能虚拟机的重要因素。对于线程的管理,虚拟机可以选择自己对程序中的多个线程进行调度,也可以采用本地绑定(native-binding)的方法,将每一个 Java 线程都映射为实际运行的操作系统上的线程,使用操作系统的调度器来实现对它们的高效支持。

10.2.3 即时编译器

由于 JVML 语言平台无关的特点,JVML 程序的装载、连接、编译过程与传统程序设计语言的编译过程相比更具有动态性,但是没有本质的区别,因此我们仅介绍即时编译器。

当一个类的某个方法第一次被调用时,虚拟机才激活即时编译器将它编译成机器代码,

编译器被称为“即时”也源于此。即时编译器以一个方法为单位进行编译,能够生成较高质量的代码,它生成的代码的执行速度可以达到解释执行的10倍。

即时编译器的出现使得Java程序的执行效率得到了很大提高,但是执行过程不得不等待编译结束,因此使得执行时间变长。在传统的静态编译中,编译时间可以忽略不计,因为经过一次编译得到的可执行文件,可以被多次执行。而对于即时编译器来说却并非如此,即时编译器在运行时编译字节码,编译时间是运行时间的一部分。为了缓解执行效率和编译开销的矛盾,很多虚拟机都会使用快速解释器和优化编译器的组合或者是简单编译器和复杂编译器的组合。下面我们用一个具体的实现来示例即时编译器是如何动态地、按需地编译一个方法的。

Intel开发的开放式运行平台(Open Runtime Platform,简称ORP)是一个研究动态编译和垃圾收集技术的开放资源研究性平台。它的即时编译器的动态性不仅表现在直到方法第一次调用才进行编译,而且还体现在它能够动态评估不同的代码而采用不同的编译策略。

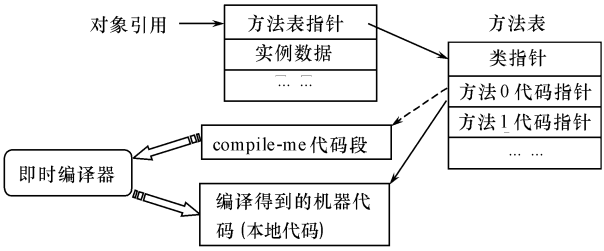


图10.12 即时编译方法

从图10.12可以看出,当一个类的方法表刚创建的时候,方法的代码指针并不是指向实际的方法代码,而是一段compile-me代码。顾名思义,compile-me代码的作用就是编译自己。所以当方法第一次被调用时,实际执行的是调用即时编译器来编译该方法的字节码。编译器完成工作之后,compile-me代码段将把方法表中的代码指针更新指向编译得到的本地代码,并且执行第一次调用。方法再次被调用时,执行的就是本地代码了。

ORP的编译器还实现了一个动态的重编译机制。这个重编译机制的关键在于对不同的代码自适应、有选择地使用不同的编译方案:对于那些执行频率低的“冷”代码,采用快速而较粗糙的编译器;而对执行频率高的“热”代码,则花更多时间对其做细致的编译。这个重编译机制的自适应性表现在它能够收集运行时的信息,判断代码是否是热点,而及时地调整编译策略。

图10.13展示了ORP的编译结构。这个编译结构有三个重要的组成部分:快速代码生成的编译器、优化编译器、统计信息。

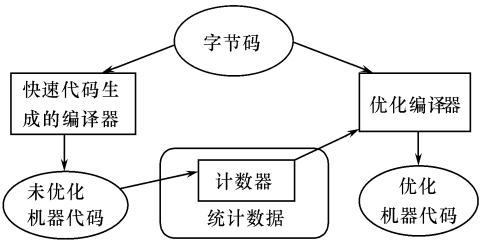


图 10.13 ORP 的重编译机制

快速代码生成编译器,在 ORP 中通常也被称为 O1 编译器。所有的方法在第一次被调用时都用这个编译器编译成机器代码。ORP 快速编译器不但能达到比较理想的编译速度,而且能够对机器代码进行轻量级的优化,产生的机器代码的执行速度要比解释执行快得多。O1 编译器的目标是快速地产生机器代码,并且维持一个合理的代码质量,它能够在对字节码进行两次遍历后产生机器代码

O1 编译器在机器代码中插入一些统计代码,用来收集统计信息,这些信息记录了方法或一段循环代码被调用的次数。这些统计信息包含进行重编译的触发条件。有了统计信息,就能够判断某段代码是否为"热"代码,当代码的调用次数达到某个阈值时,它将被优化编译器重新编译。优化编译器对方法重新编译时还将利用 O1 编译器收集的这些统计信息。

优化编译器也叫 O3 编译器,它对代码进行更为细致的优化,产生质量较好的目标代码。之后,当方法再次被调用时,就会执行优化版本的机器代码。O3 编译器为了产生高质量的目标代码,需要花更多时间在优化上。它采用传统的编译方法,为字节码建立一个中间表示,基于中间表示进行全局优化。

ORP 就是这样采用简单编译器和复杂编译器的组合,构造了一个平衡编译时间和代码质量的动态编译机制的。

*10.3 无用单元收集

理论上讲,无用单元(garbage)就是那些在程序的继续运行过程中不会再被使用的数据单元。被无用单元占据的内存空间应该被回收,以便给需要分配空间的变量使用,这样一个处理过程就称之为无用单元收集(garbage collection),俗称垃圾收集。该过程不需要程序员的干预,它可以自动地执行对内存的这种管理。但它可能需要来自编译器、操作系统和硬件方面的支持,并且由运行时系统来决定何时和怎样执行无用单元收集。

无用单元收集器(以下简称收集器)需要根据数据的活跃性来判断哪些是无用单元。由

于实际上并非总能判断出一个数据记录的值以后是否还需要,因此收集器所使用的活跃性分析采用稳妥的策略。该策略使得活跃性是通过根集(roots set)以及从根集开始的可达性来定义。按照这个策略所得到的可达记录并非都是真正活跃的,但是我们都保留它们;另一方面,我们要求编译器努力极小化可达记录中实际上并不活跃的记录的数目。这样,通常所指的无用单元是那些不可能从程序变量经指针链到达的堆分配记录。

函数式语言一般都用这种技术来回收内存空间。对于命令式语言 Java 来说,它与 C 和 C++ 语言不一样,对象和数组这样的数据都分配在堆上,在栈中有相应的指针指向它们,并且语言不向使用者提供释放空间的函数。这样,一旦这些指针在栈中被释放后,堆中相应的记录就很可能不可达,因此需要依靠收集器来回收它们。

本节我们简要介绍一些主要的无用单元收集方法,并且描述编译器和收集器之间的一些相互影响,包括编译器提供给收集器的支持和收集器提供给编译器的接口。

10.3.1 标记和清扫

标记和清扫收集方法首先标记堆上所有可达记录,然后回收未被标记的记录。

在进行无用单元收集时,全局可见的变量都被看作是活跃的,任何活跃着的过程的任何一个活动记录中的局部变量也都被看作是活跃的。这样,根集就包含了全局变量、活动记录栈中的局部变量和被活跃着的过程使用的寄存器。堆上活跃记录的集合也就是从根集开始的任何一条指针路径上的记录的集合,它们可以看成是一个有向图,其中记录是结点,指针是边,程序变量是根,因此任何图遍历算法,如深度优先算法,都可用于在这个图上标记所有的可达记录。这就是标记阶段。

任何未被标记的记录都是无用单元,应该被回收。回收过程称为清扫。从堆的首地址开始,逐个记录地考察整个堆,寻找未被标记的记录,把它们链成一个空闲链表。并且,这一阶段同时清除被标记的记录的标记,以备下一轮收集。当空闲链表中的空间不足以支持程序的存储分配请求时,就启动下一轮收集。

如果用户程序需要很多不同大小的记录,一个简单的空闲链表对于分配函数来说可能显得效率不高。因为当分配一个大小为 n 字节的记录时,它可能需要沿着这个链表寻找,直至找到一个大小合适的空闲块。我们可以通过建立有若干个空闲链表的一个数组来解决这个问题,例如 *freelist*[i] 就是所有大小为 2^i 的空闲块的链表。这样,如果程序要分配一个大小为 $n(2^{k-1}<n\leq 2^k-1)$ 的记录时,那就从 *freelist*[k] 中取一块。如果该链表为空,那就从 *freelist*[$k+1$] 中取一块,将其中一半分配给用户程序,剩余的一半链回到 *freelist*[k] 链表中。如果空闲块大小小于 2^{k+i} 的所有链表均为空,而 *freelist*[$k+i$] 链表不为空,则从 *freelist*[$k+i$] 链表中取一块,将其中的 2^k 大小分配给用户程序,而将剩余部分分割成若干块,分别插入到 *freelist*[k] ~ *freelist*[$k+i-1$] 链表中。如果这样的操作失败,那就调用收集器来补充

空闲链表。

传统的标记和清扫方法通常有两大问题。

首先是碎片(fragmentation)问题,它可以分为外部碎片和内部碎片两个方面。外部碎片是指当我们要分配一个 n 字节大小的记录时,发现有很多小于 n 字节的空闲块存在,但就是没有合适的空闲块可分配给这个记录。内部碎片就是说程序使用一个过大的记录而没有拆分它,导致没有被使用的内存处于该记录中。碎片的后果就是空闲块和活跃记录交织在一起,使得对大记录的分配非常困难。通过维护上面提到的一组空闲链表并合并相邻空闲块可以缓解这个问题,但问题并没有消失。

第二个问题涉及引用局部性(locality of reference)问题。既然记录都不会被移动,那么活跃记录在一次收集以后仍然在原位置,和空闲块相交织。然后,新记录使用这些空闲块,其结果是不同年代的记录交织在一起。这给引用局部性带来了消极的影响,因为这种交织有可能使得当前要使用的各个活跃记录被分散到很多的虚拟内存页中,这些页在内存中可能被频繁地换进换出。在有虚存或缓存的计算机系统中,良好的引用局部性是非常重要的。

10.3.2 引用计数

标记和清扫收集方法通过首先找出堆上所有可达记录来确定无用单元。我们也可以通过记住有多少指针指向每个记录来直接完成,这称为记录的引用计数,记录的引用计数存在该记录中。

在使用这项技术的系统中,编译器需要在每个出现指针存储的地方生成额外的指令,以调整一些引用计数器的值。比如记录 p 的引用存进 $x.f$ 中,那么 p 的引用计数值会加 1,而 $x.f$ 原来指向的记录的引用计数值会减 1。当一个记录的引用计数值为 0 的时候,就可以把该记录加入空闲链表,并且被回收记录本身的指针域都要一一检查,它们所指向的记录的引用计数值也都要减 1。

引用计数算法看起来非常简单,具有吸引力,但是除了碎片和引用局部性问题外,它还有两大问题:第一,它并不总是有效的;第二,很难提高效率。

(1) 引用计数的技术对于循环的数据结构会失效。如果一组记录中的指针形成了一个循环,那么,即使从根集已经不可能到达这些记录了,这些记录的引用计数也永远不可能减到零。而且事实上,这种循环在一般的程序中经常会产生。

(2) 引用计数的效率问题是它的代价。因为每当执行指针存储的时候,都需要执行额外的指令来调整一些引用计数器的值。

上述问题使得引用计数技术近年来已失去了吸引力。对于大部分高性能的通用系统来说,引用计数收集已经被跟踪型收集代替,这种收集器是在遍历可达记录图时,将活跃记录和无用单元区分开来。10.3.1 节的标记和清扫收集方法就是一种跟踪型收集方法。

但是引用计数方法本身还是有很多有意义的优点。例如,它回收迅速;另外,即使大部分堆空间都被占据的时候,它的性能仍然不受影响,很多其他的收集器这时会需要更多的空间用于交换以提高效率;还有,它可以被直截了当地做成完全渐增的和实时的。将来也许还会发现这种技术的一些其他用途,也许在混合型的收集器中使用,也许通过特殊的硬件可以提高它的性能。但是,一般来说,引用计数方法不会作为传统的单处理器上一种主要的无用单元收集技术。

10.3.3 拷贝收集

这个算法跟10.3.1节中的标记和清除算法一样,它也遍历可达记录所组成的有向图,只不过它在遍历的同时进行清扫,并且这种清扫主要是拷贝活跃记录。

通常,这种方法将整个堆空间分成大小相等的两块,每块都是连续的区域,一块叫做*from_space*,另一块称做*to_space*。在程序运行时,只有*from_space*是可用的。当运行程序要求内存时,就在*from_space*中向上线性地分配内存。当运行程序要求的内存分配超过了*from_space*中空闲区域的大小时,运行程序暂时被停下来,拷贝收集器被激活,用来回收空间。

拷贝收集器根据某种遍历算法将所有的活跃记录遍历一次,同时把它们从*from_space*拷贝到*to_space*,并且在*to_space*中这些记录被紧缩到一边。一旦拷贝完成,*to_space*和*from_space*的角色相互交换,然后程序继续运行。图10.14给出了收集前后的一个示例。

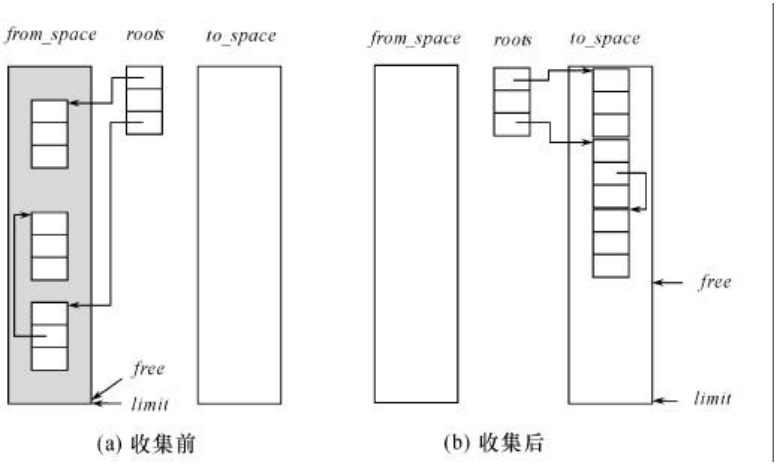


图10.14 拷贝收集

我们把这个算法讨论得稍微深入一点。对该算法来说,最基本的操作是指针的转移(forwarding)操作,就是把一个指向*from_space*的指针*p*,修改成指向*to_space*中合适的位

置,算法见图 10.15。有 3 种情况需要分别对待:

⑴ 如果指针 *p* 指向一个已经被拷贝到 *to_space* 的 *from_space* 记录,那么该记录的第一个域,也就是 *p.f*1 是一个特殊的转移指针,它指示该记录拷贝在什么地方。因此,现在只需要将它返回就可以了。

⑵ 如果指针 *p* 指向的记录还没有被拷贝,那么就将它拷贝到 *to_space* 中 *free* 所指位置,并把这个转移指针赋给 *p.f*1,也就是让 *p.f*1 指向该记录在 *to_space* 中的位置。

⑶ 如果 *p* 不是指针,或者如果 *p* 指向 *from_space* 之外的区域 (如指向收集区域以外的地方,或者指向 *to_space*),那么对 *p* 的转移操作将什么也不做。

```
function forward(p) ;
    begin
        if p 指向 from_space then
            if p.f1 指向 to_space then
                return p.f1
            else
                begin
                    for p 的每个域 fi do free.fi := p.fi ;
                    p.f1 := free ;
                    free := free+记录 p 的大小 ;
                    return p.f1
                end
        else return p
    end
```

图 10.15 转移指针的算法

从理论上来说,只要给足够的内存,拷贝收集器可以得到很高的效率。另外,它还可以将活跃数据紧缩在一起,碎片情况消失,引用局部性得到改善。当然,在实际使用中,这些好处会受到很多限制。首先,这个算法需要的空间是实际需要空间的两倍。其次,拷贝大记录的代价太大。还有,如果遍历采用宽度优先搜索的话,引用局部性的改善不充分。但总的来说,拷贝算法的前景是很好的,它被广泛用作其他无用单元收集算法的基础,如分代算法、渐增式算法。

10.3.4 分代收集

分代收集的原理基于对这样一个现象的观察:很多程序的运行过程中,新建记录很可能

很快死去,不会出现对它的拷贝;反过来,一个记录在几次收集后还可到达的话,那么很可能还会活跃到许多次收集后。因此收集器应该将精力集中到“年轻”的数据上,因为这里有相对高的无用单元比率。

我们将堆分成代(generation),最年轻的记录在 G_0 代,G_i ($i>0$) 代中的记录比 G_{i-1} 代中的任何记录都要老。越年轻的代越被频繁地收集。

对 G_0 代进行收集时,就是从根集开始,使用拷贝方法或者使用标记和清扫方法。但是,这时的根集不仅仅是程序变量,它还包括 G_1, G_2, ...中指向 G_0 的指针。如果这样的指针太多的话,那么处理根的过程将比遍历 G_0 的可达记录的时间还要长。幸好,老记录指向年轻得多的记录的情况极少出现。通常是较新的记录指向老记录。

为了避免确定 G_0 的根集时在 G_1, G_2, ...中查找,我们需要编译器提供一些支持,让编译过的程序运行时能记住哪些地方有老记录到新记录的指针。这样的方法有好几种:

(1) 记忆集合 编译器产生指令,在每个形式为 b.f := a 的修改存储单元的指令后,将 b 放到被修改记录的一个向量(记忆集合)中。然后在收集时,收集器扫描记忆集合,寻找指进 G_0 的老指针 b.f。

(2) 卡标记 该方法将存储区域划分为大小为 2^k 的逻辑“卡”,一个记录可以占据一块卡的一部分,或者从一块卡的中间开始,继续到下一块。每当地址 b 的内容被更新时,包含这个地址的卡就被标记。这里有一个字节数组用来做标记,字节索引可以通过将地址 b 右移 k 位获得。

(3) 页标记 该方法类似于卡标记方法,如果 2^k 正好是页的大小,那么可以用操作系统的虚拟内存管理机制来标记页,而不需要编译器生成额外的指令。

记忆集合的优点在于精确,因为它包含的是被更新的指针的地址,而卡标记方法的优点在于简单。对于每条更新指令的维护,记忆集合的方法可能需要 10 条额外的指令,而卡标记最少情况下只需要 2 条额外指令就够了,它所需要的开销远远小于记忆集合方法。如果将两种方法结合在一起使用,会带来更高的效率。

当收集开始时,记忆集合告诉我们,老一代的哪些记录(或者卡、页)可能包含指进 G0 的指针。这些指针是 G0 根集的一部分。

通常,G_i ($i>0$) 是按指数地大于 G_{i-1},比如 G_0 是 0.5 兆字节大小,那么 G_1 应该是 2 兆字节大小,G_2 就应该是 8 兆字节大小。假设 G_0 是当前要收集的代,它里面的活跃记录将拷贝到 G_1 中。经过几次对 G_0 的收集之后,G_1 可能会聚集了相当数量的无用单元需要被收集。因为 G_0 可能包含了很多指进 G_1 的指针,所以最好将 G_0 和 G_1 一起收集。和前面一样,这时需要扫描记忆集合,以获得包含在 G_2, G_3 ..... 中的那部分根。这样,各代都可能被收集,当然,越老的代收集频率越低。

10.3.5 渐增式收集

虽然收集时间的总和只占整个程序运行时间很小的百分比,但是收集器仍然有可能偶尔将运行程序中断相对长的时间。例如分代算法提到 G_0, G_1……的大小呈指数变化,当对较老或最老的代进行收集时,可能就需要较多的时间。对于交互式程序和实时程序来说,这一点是难以接受的。而渐增式收集器或者并发收集器将程序运行和无用单元收集交错进行,避免了出现这种长时间的中断。

在渐增式算法中,只有当运行程序请求时,收集器才开始工作;在并行算法中,收集器可以在运行程序执行的任何指令之间进行工作。渐增式算法和并发算法面对的问题虽然类似,但是显然后者更加复杂和困难。

这些收集算法的困难在于,当收集器在做遍历以得到一个可达记录图时,改变者并没有停止修改可达记录图。因此,收集算法必须有某种方法来跟踪可达记录图的变化。面对这些变化,也许需要重新计算可达记录图中的某些部分。已经有很多这方面的技术,我们在此不作介绍。这些技术都需要编译器生成额外的指令提供支持,有些方法也可以利用虚拟内存的硬件来实现。

10.3.6 编译器与收集器之间的相互影响

在收集器的设计中,对收集器的底层有下面这些基本要求:

⑴ 收集器必须能确定堆上分配的记录大小,这样它们才能被拷贝;

⑵ 收集器必须能定位包含在堆记录里的指针,这样它们才能被跟踪和更新;

⑶ 收集器必须能定位所有在全局变量中的指针;

⑷ 收集器必须能在程序中任何一个可以进行收集的地方找到所有在活动记录栈中和寄存器中的指针;

⑸ 收集器必须能找到所有由指针运算所产生的值指向的记录;

⑹ 收集器必须能在记录被移动时更新所有涉及到的指针值。涉及到的指针包括指向它的指针,以及由指向它的指针经计算而得到的指针。

这些要求都必须通过编译器的支持才能得到满足,因为很多所需信息只有在编译时能够获得。这是编译器必须对收集器提供支持的一些基本方面。上述 6 点中,对最后两点的理解需要阅读下面的导出指针部分。

概括来说,对于使用无用单元收集技术的语言,它的编译器与收集器之间的相互影响一般有下面这些:编译器生成(可能需要用收集器提供给它的接口函数)用来分配记录的代码;编译器为每个收集周期提供根集元素的存储位置描述;编译器向收集器提供堆上数据记录

的布局描述等等。

另外,对于渐增式收集和分代收集,编译器还必须生成一些额外的代码,如分代收集中所讲的建立记忆集合的指令。

快速分配

某些编程语言和某些程序可以快速分配堆记录,同时也快速产生无用单元。对于函数式语言来说尤其如此,因为它不鼓励更新旧的数据。

可以想像到,最快的分配记录和产生无用单元的速度是每遇到一条存储指令就分配一个字,这是因为一个堆分配记录的每个字通常都需要被初始化。经验数据表明,不管采用什么样的编程语言或者编写什么样的程序,每7条指令中就有一条是存储指令。因此,我们可以认为每条运行指令要进行1/7个字的分配。

假设收集的代价可以通过调整分代收集的代的数目变得很小,那么仍然需要可观的代价用于创建堆记录。若想最小化这样的代价,应该使用拷贝收集方法,因为它供分配的空间是连续空间,而不是空闲链表。若 free 指针指示下一个空闲位置,limit 指针指示空闲区域的末端(见图 10.14),要分配一个 N 字节大小的记录,一般是调用分配函数,下面列出其执行步骤:

(1) 分配函数的调用序列;

(2) 测试 free + N < limit? (如果失败,则调用收集器)

(3) result ← free;

(4) 将 free 开始的 N 个字节都清空;

(5) free ← free + N;

(6) 分配函数的返回序列:

(a) 将 result 移到对后面的计算有用的某个地方;

(b) 用一些有用的值将该记录初始化。

通过将分配函数在每个分配记录的地方进行内联展开,步骤 (1) 和 (6) 可以删除。步骤 (3) 经常可以删除,将它合并到步骤 (a) 即可,步骤 (4) 也可以删除,用步骤 (b) 就可以了。步骤 (a) 和 (b) 之所以编号不同,是因为它们应该被看成是有用计算中的一部分,而不属于分配开销。

步骤 (2) 和 (5) 不能删除,但是如果在同一基本块中不止一个分配,则它们可以被多个分配操作共享。将 free 和 limit 放在寄存器中,步骤 (2) 和 (5) 总共用3条指令就可以完成。

通过这种合并技术,分配一个记录并且最终回收它的开销大约只需要4条指令就可以了。

堆上数据布局的描述

收集器必须能够处理各种类型的数据,这些类型包括链表、树、程序声明的类型等等。它必须能够知道每种类型的域的数目,以及其中哪些是指针域。

对于静态类型化的语言,如 Pascal,或者面向对象的语言,如 Java,确定堆记录最简单的方法就是让每个记录的第一个字指向一个特殊的类型或者类的描述符记录。该描述符记录会告知该记录的大小以及每个指针域的位置。

这样,对于静态类型化语言,每个记录有一个字(描述符指针)的开销用于收集。但是对于面向对象语言,每个对象本来就需要这样一个描述符指针用来实现对方法的动态查找,因此每个对象不存在用于收集的额外开销。

类型或类的描述符必须由编译器的语义分析阶段所得的静态类型信息来生成。描述符指针将是运行时系统的 alloc 函数的参数。

除了描述堆上每个记录外,编译器还必须为收集器确定每个包含指针的临时变量和局部变量,不管它们是在一个寄存器中还是在一个活动记录中。因为每条指令都可能改变活跃临时变量的集合,因此指针映像(指活跃指针的集合)在程序的每个点都可能不同。于是,为简单起见,编译器一般只在收集工作可以启动的地方描述指针映像。这些地方是 alloc 函数的调用点;还有,因为任何函数调用可能调用一个会调用 alloc 的函数,因此在每个函数调用点必须描述指针映像。

指针映像最好是用返回地址作为键值,即一个处于地址 a 处的函数调用最好用紧跟在它后面的返回地址来索引,因为返回地址是收集器在相邻活动记录中能看到的值。对于在该调用后立即活跃的每个指针,指针映像告诉我们存放该指针的寄存器或该指针在活动记录中的地址。

在收集开始时,为了获得所有的根,收集器从栈顶开始,对所有的活动记录逐个地往下遍历栈。其中每个返回地址都是描述下一个活动记录的指针映像入口的关键字。在每个活动记录中,收集器从来自这个活动记录的指针去进行标记(如果是标记和清扫方法的话)或者转移(如果是拷贝收集的话)。

对于被调用者承担保存责任的寄存器需要做特殊处理。假设函数 f 调用 g,g 又调用 h。*h 知道在它的活动记录内保存了哪些被调用者承担保存责任的寄存器,并且在它的指针*映像中提到这个事实;但是 h 不知道它所保存的这些寄存器中哪些寄存器含有指针。因此*g 的指针映像必须描述,在它保存的由被调用者承担保存责任的寄存器中,在 h 的调用点*哪些含有指针。

导出指针

程序的指针有时可能指在一个堆记录的中间，或者指在这个记录的前面或后面。例如表达式 a[i−2000] 经编译后被当作 M[a−2000+*i* 来计算：

$t_1 := a - 2000$

$t_2 := t_1 + i$

$t_3 := M[t_2]$

如果表达式 *a*[*i*−2000] 出现在一个循环内部，那么代码优化可能会将 $t_1 := a - 2000$ 作为循环不变计算提升到该循环外面。如果该循环包括了一个 *alloc* 调用，并且一个收集发生时 t_1 还是活跃的，那么，收集器是否会感到困惑？因为指针 t_1 没有指向一个记录的开头，甚至出现更坏的情况，它指向一个不相关的记录。

我们称 t_1 是从基指针 a 导出的指针。指针映像必须能识别导出指针，并且知道每个导出指针的基指针是哪个指针。于是，当收集器将 a 重新分配到地址 a′时，它必须把 t_1 调整到 $t_1 := t_1 + a' - a$。

当然，这意味着，只要 t_1 是活跃的，a 就必须保持活跃。我们考虑图 10.16 左边的一个函数体，其执行语句就是一个循环，该函数体的实现在图 10.16 的右边，假定数组分配在堆上。如果不存在 a 的其他使用，那么变量 a 在对 t_1 的赋值后不再活跃。但是和返回地址 L_2 相关的指针映像将不能适当地解释 t_1。于是，对于编译器的活跃变量分析来说，一个导出指针的活跃隐含地要求保持它的基指针活跃。

```
var a: array [1..100] of integer := 0;
    i: integer;
begin
    for i := 2001 to 2100 do
        f(a[i−2000])
end
```

```
    r1 := 100
    r2 := 0
    call alloc
    a := r1
    t1 := a−2000
    i := 2001
L1: r1 := M[t1+i]
    call f
L2: if i≤2100 goto L1
```

图 10.16 一个函数体及它的实现

习 题 10

10.1 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 机器上,命令

```
cc cfile
```

的结果是错误信息

```
/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错误信息。

10.2 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 机器上用 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。

10.3 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 机器上用命令

```
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。

10.4 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。"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);
}
```

10.5 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,在连接时的做法是下面三种情况的哪一种,说明你的理由。

(a) 将该库函数的目标代码连到用户程序;

(b) 将该库函数的目标代码所在的目标文件连到用户程序;

(c) 将该函数库全部连到用户程序。

10.6 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(库名必须以 lib 为前缀),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。

10.7 现将图 10.2 的 swap.c 编译成可重定位目标文件 swap.o。对于在 swap.o 中定义或引用的符号,请说明它是否在 swap.o 的.symtab 节中有相应的符号表条目?如果有,指出该符号被定义的模块(swap.o

或 main.o)、符号类型 (local、global 或 extern),以及所在节 (.text、.data 或 .bss)。

符号	在 swap.o 的 .symtab 节中有条目?	符号类型	由哪个模块定义	所在节
buf				
bufp0				
bufp1				
swap				
temp				

10.8 修改习题 10.2 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 11K,后者却要接近 1M。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。

10.9 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。

(1) p.o→libx.a

(2) p.o→libx.a→liby.a

(3) p.o→libx.a→liby.a 并且 liby.a→libx.a→p.o

10.10 假设 main 调用函数 f,被调用者承担保存责任的寄存器含的值都是 0。然后,f 保存它将要使用的被调用者承担保存责任的寄存器;并在其中一部分中存放指针,在另一部分中存放整数,对剩下的没有动作;再下一步调用函数 g。现在 g 也保存一些被调用者承担保存责任的寄存器,将一些指针和整数放到其中,然后调用 alloc,它启动无用单元收集工作。

(1) 阐明函数 f 和 g 的指针映像。

(2) 写出无用单元收集器为了恢复所有指针的确切位置而需要的步骤。

*第 11 章　面向对象语言的编译

软件系统的规模越来越大,并且日趋复杂,以更有效和更透明的方法来开发这样的系统的呼声与日俱增。最终的目标是从已做好的标准构件去构造软件系统,就像现在构造硬件系统那样。模块化、模块的可重用性、模块的可扩充性和抽象性是朝向这个目标的一些尝试,而面向对象语言在这些方面提供了一种新的可能性。现在,面向对象已被看成管理复杂软件系统的一种重要风范。

本章概述面向对象语言的重要概念和实现技术。为突出一些重要概念的实现技术,我们以C++语言为例,介绍如何将C++程序翻译成 C 程序;实际的编译器大都把C++程序直接翻译成低级语言程序,而不是把 C 语言作为中间语言并利用 C 语言编译器。

11.1　面向对象语言的概念

面向对象语言可以看成是命令式语言,除了变量、数组、结构和函数等熟知的概念外,它还引入一些新概念。本章只讨论其中一些概念。

11.1.1　对象和对象类

命令式语言的主要模块化单元是过程(包括函数)。在数据的复杂性与处理的复杂性相比显得微不足道时,这是适宜的抽象和模块化。但是,当任务的描述和有效的解决方案需要使用复杂的数据结构时,单用函数进行模块化是不够的。有效处理这些任务的合适抽象级别应该是允许把数据结构和操作这些数据结构的相关函数封装在一个单元中。

面向对象语言的最基本的概念是对象。一个对象由一组属性和操作于这组属性的过程组成,属性到值的映射称为对象的状态,过程也叫做方法。属性和方法共同形成了对象的特征。于是,对象封装了数据及其上的操作。面向对象语言最重要的基本操作是激活对象 *o* 的方法 m,写成 o.m。在这儿,对象扮演着主要角色,而方法是对象的成分并且从属于对象。

为了说明面向对象的使用,以二维图形对象为例。圆、椭圆、矩形、三角形、多边形、点、线和折线等都是图形对象。图形对象组合在一起又形成新的类型:复合图形对象。每一类图形对象有它自己的特性,但是各类图形对象也具有一些公共的性质。于是,对各类图形对

象都有用的构件箱至少应包含复制、平移、删除和缩放等操作。

以复制为例来说明面向对象技术的优点。对于像复制这样的操作,被操作对象的准确类型是无关紧要的,因而不必给出。因为,复制一个复合对象可描述为:复制该复合对象的所有子对象,再把这些复制品组成一个新的复合对象。在面向过程的实现中,为了能够复制所有类型的图形对象,必须这样定义复制函数:它首先确定被复制对象的类型,然后,调用相应类型的特定复制函数。该函数的一个缺点是,图形对象的所有类型以及它们的复制函数通过该函数及该函数解释的表示图形对象的数据结构彼此关联起来。结果是,一个程序包含了所有这些复制函数,即使图形对象的许多类型在这个程序中实际上并不需要。这样,该程序的规模没有必要地变大了。另一个问题是扩充变得复杂起来。例如,若需要把另一个基本类型加入到构件箱,那么上述复制函数必须扩充,所解释的数据结构也必须修改。在最坏的情况下,构件箱的所有程序块都不得不重新编译,当这些程序依赖于由该复制函数解释的公共数据结构时,就会发生这种情况。

而在面向对象的实现中,每个对象有它自己的复制函数,激活该函数并不需要知道准确的对象类型。新的基本类型也可以直接加进去,构件箱中的原来部分可以维持不变,不必重新编译就可继续使用。对于大家关心的模块性和扩充性,面向对象的方法给出了相当可观的改进。

面向对象语言拓广了 Pascal 和 C 等命令式语言的类型概念,增加了对象类(简称类)的概念。一个对象类规范了该类中对象的属性和方法,包括它们的类型和原型(参数和返回值类型)。一个对象要想属于该类,它必须含有这些特征,当然还可以含有其他一些特征。某些面向对象的语言,例如 Eiffel,允许对方法进一步规范,例如规定方法的前置条件和后置条件,这是提供一种限制方法含义(语义)的手段。

对象类形成了面向对象语言的模块单元。对于上面描述的图形对象,可以为不同类型的图形对象定义不同的对象类,用对象类库来实现构件箱。

同一类中的不同对象,它们的属性值可能不一样,因此它们必须有自己的存放属性的存储单元;但是,它们的方法是一样的,因此它们可以共享方法的代码。这一原则指导 11.2 节开始介绍的C++ 程序到 C 程序的翻译。

下面将把术语“类”和“类型”看成是同义的。

11.1.2　继承

继承定义为类 A 的所有特征并入到新的类 B,B 可以进一步定义自己的一些其他特征,在一定条件下还可以重写或覆盖(overwrite)从 *A* 继承来的方法。某些语言允许重新命名继承来的特征,以避免名字冲突,或者允许在新的上下文中使用更有意义的名字。

如果类 *B 继承类 A,那么类 B 叫做类 A* 的派生类,而类 *A 叫做类 B* 的基类。

继承是面向对象语言最重要的概念之一。继承的层次性允许把类库结构化和引入不同级别的抽象。仍以图形对象的例子来说明这一点。图 11.1 给出了图形类库中继承层次的一部分。在该图中,用椭圆表示对象类,椭圆中包括类名和该类的部分方法;继承由箭头表示。例如,图形对象类 GraphicalObj 有方法 translate 和 scale,那么所有的图形对象都可以被平移(translate)和缩放(scale)。封闭图形类 ClosedGraphics 和折线类 PolyLine 从类 GraphicalObj 继承了这些方法。在类 PolyLine 中,这些被继承的方法被覆盖,并且引入计算周长的方法 length。类 ClosedGraphics 引入新方法 area,它计算封闭图形对象的面积。多边形类 PolyGon 同时继承类 ClosedGraphics 和类 PolyLine,其中 area 被覆盖。最后,矩形类 Rectangle 继承类 PolyGon 并且覆盖 area。

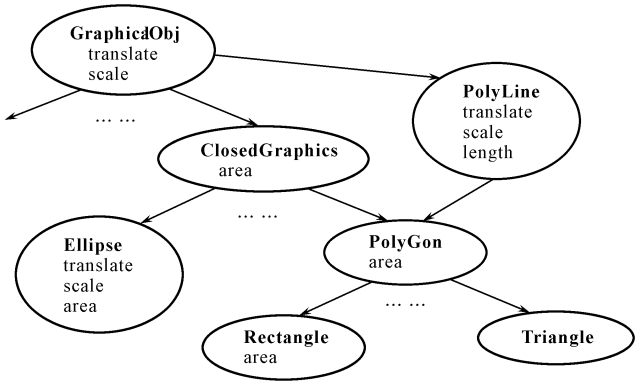


图 11.1 图形对象的继承层次结构

虽然方法 translate 和 scale 在类 GraphicalObj 中引入,但是它们却不能在这儿定义。因为这些方法只能结合具体的图形对象来定义,例如椭圆类 Ellipse 和折线类 PolyLine。不过,这些方法在类 GraphicalObj 中引入,意味着每个图形对象必须有这些方法,虽然这儿并不提供它们的实现。包含未定义方法的类叫做抽象类,它没有自己的任何对象实例。

类似地,类 ClosedGraphics 引入方法 area,但该方法也不能在这个类中定义。area 首先在类 Ellipse 和类 PolyGon 中定义。但是,对一般的多边形而言,面积计算是复杂的,它取决于该多边形能划分成多少个三角形以及这些三角形的面积和。另一方面,矩形的面积计算是简单的。如果矩形经常使用,最好在类 Rectangle 中以更有效的方法实现 area,如引入矩形的边长属性,利用它们直接计算面积。对于类 PolyLine 的所有方法,类 PolyGon 最好接管所有这些方法,不作任何覆盖。

由上所述,继承概念提供了这样的可能性:用简单的办法可以重用部分现有实现,或扩充它们,还可以重写个别方法以适应局部的特殊需求和环境。

此外,任何事物,如果它能用较高级别的抽象表述,那么它就有较广的应用可能性,因而有较大程度的可重用性。所以应尽可能使用高的抽象级别。另一方面,有时必须转移到更具体的级别,使得在解决某些特定问题时有更多的结构可用。假定图形对象的一种变换可以由一串平移、缩放和其他一些图形对象的操作描述,那么该变换可以用一个函数实现,该函数对任何类型的图形对象都可用。如果无抽象类并且编译器要进行类型检查,那么需要为每个类写一个这样的函数,即使这些函数的实现具有相同的内容。

于是,类型化的面向对象语言考虑其类型系统中的继承层次结构。如果类 B 继承类 A,那么指派给 B 的类型是指派给 A 的类型的子类型。子类型的每个对象自动地是它超类型的一个元素。一个继承类是被它继承的类的子类。这样就有下列结果:

⑴ 子类型规则

当某个类型的一个对象在某个输入位置(函数的输入参数、赋值的右部)被需要或作为函数的返回值时,其任何子类型的对象允许出现在这些地方。

类 *B* 的一个对象,若它不同时是 *B* 的某个真子类的对象,那么称该对象是 *B* 的一个真对象,称 *B* 是该对象的运行时类型。这样,每个对象有惟一且确定的运行时类型,它是该对象所属的最小类型。此外,一个对象也是它运行时类型的每个超类型的元素。

凭借子类型规则,面向对象语言的方法可以接受运行时类型不同(因而结构也不同)的对象,例如在参数位置就可以这样。这是多态性的一种形式。

继承的子类型规则,以及继承类可以覆盖被继承的方法,引起了一个有意思的结果。它对编译器来说是重要的。例如有一个函数 *f*,它允许类 ClosedGraphics 的对象作为参数,并假定 *f* 调用该参数的 area 方法。因为类 ClosedGraphics 不能定义 area 方法,因此调用的实参肯定是类 ClosedGraphics 的子类的对象,而不是它本身的真对象。下面是一般性的规则。

⑵ 方法选择规则

如果类 *B 继承类 A* 并且重写了方法 m,那么对类 *B* 的对象 b 来说,即使它作为类 *A* 的对象使用,也必须使用在类 *B* 中定义的方法 m。

该规则给编译器提出一个问题:编译器必须会产生调用一个方法的代码,但在编译时它很可能确定不了究竟要调用哪一个方法。对上面的例子来说,在 *f* 的代码生成中,编译器不知道应该把名字 area 绑定到哪个类的 area 方法。一般来说,这个绑定只有在运行时当实参已经明确的情况才能完成,这样的绑定叫做动态绑定。因此,我们也可以把方法选择规则表述如下:

⑶ 动态绑定规则

当对象 o 的一个方法可能被子类重新定义时,如果编译器不能确定 o 的运行时类型,那么必须对该方法进行动态绑定。

本节的主要部分讨论继承的有效实现。

11.1.3 信息封装

大多数面向对象语言提供了一种机制,它可用来将类的特征分成私有的(private)和公共的(public)。私有特征完全不可见,或者至少在某个上下文中不可见。某些面向对象语言用不同的上下文区分作用域,如"在一个类中"、"在派生类中"、"在友元类中"等等。语言构造或一般规则可以指明上下文的可见、可读、可写或可调用特征。

由编译器来实现这些作用域规则是简单而又明显的。因此,虽然信息封装是非常重要的,但是下面不讨论它的实现。

现在,我们把到目前为止讨论的概念小结如下:

(1) 面向对象语言引入了新的模块化单元:对象类。对象类可以封装数据及在这些数据上的操作。

(2) 继承概念对构造现有模块(对象类)的派生模块是极其有用的。

(3) 面向对象语言的类型系统使用继承概念:继承类是基类的子类型,它们的对象可以用在基类的对象所允许出现的任何地方。

(4) 继承层次结构把不同级别的抽象引入程序。即在程序或系统的不同点使用不同级别的抽象。

(5) 抽象类型可用于规格说明,通过逐步继承而求精,直至最终的实现。也就是说,它提供从规格说明经过逐步设计到各种实现的无缝转变。

本章将以面向对象语言C++和Eiffel为例。

11.2 方法的编译

本节用具体的例子来说明方法的编译。首先,用C++实现前一节图形对象的类层次结构的一部分。

先定义一般的图形对象类GraphicalObj如下:

```
class GraphicalObj {
    virtual void translate (double x_offset, double y_offset);
    virtual void scale (double factor);
    ...     //可能还有一些其他方法
};
```

其中方法translate和scale声明为虚方法。在C++中,这是要求它的派生类重写此方法。

类GraphicalObj的一个特别重要的子类是点类Point。几乎每一种具体的图形对象类的

实现都以某种方式使用点。类 Point 的定义如下：

```
class Point :public GraphicalObj {
    double xc, yc;
  public :
    void translate (double x_offset, double y_offset) {
        xc += x_offset;
        yc += y_offset;
    }
    void scale (double factor) {
        xc *= factor;
        yc *= factor;
    }
    Point(double x0 = 0, double y0 = 0) {xc = x0; yc = y0;}
    void set(double x0, double y0) {xc = x0; yc = y0;}
    double x(void) {return xc;}
    double y(void) {return yc;}
    double dist(Point&) ;
};
```

一个点有 x 和 y 坐标 xc 和 yc，它们表示点在二维空间中的位置。xc 和 yc 是点的私有数据，只能在该类的方法中访问（或者由友元函数访问）。在类 Point 中定义的方法是公共的，它可以由知道一个点的任何场合使用。例如，如果 p 是一个点，那么 p.x() 激活 p 的方法 x 并且返回 p 的 x 坐标；p.translate(1, 2) 通过改变 p 的坐标，使它在 x 轴方向上移动一个单位，在 y 轴方向上移动两个单位。

一般而言，对一个对象 o，它的带有参数 arg1, arg2..... 的方法 m 由 o.m(arg1, arg2.....) 激活。

将一个C++ 语言的类翻译成 C 语言的程序段，主要工作有如下几点（由继承引出的问题放在 11.3 节考虑）。

⑴ 将C++ 语言中一个类的所有非静态属性构成一个 C 语言的结构类型，取类的名字作为结构类型的名字。

⑵ 类的静态属性是该类的所有对象所共有的，应当翻译成 C 中的全局变量，但是需要改一个名字。根据下面 ⑷ 的 (a)，读者应该明白如何改名。

⑶ C++ 语言中类的对象声明不加翻译就成了 C 语言中相应结构类型的变量声明，不管对象声明出现在程序中的什么地方。

⑷ 在解释类的非静态方法的翻译之前，我们先做个约定。我们知道，在C++ 语言中，

函数的参数传递有值调用和引用调用两种方式,当形式参数名前加字符 & 时,表示该参数是引用调用。但是 C 语言的参数传递只有值调用。为简洁起见,我们下面假定 C 也有引用调用方式,也用字符 & 表示这种方式,以避免在解释如何翻译引用调用上花笔墨。

将C++语言中类的非静态方法翻译成 C 语言的函数,对应的方法和函数的区别如下:

(a) 由于类声明在C++语言中形成一层作用域,类中方法声明的作用域就是该类;而在 C 语言中,函数声明的作用域至少是所在的文件。为了避免不同类的同名方法在 C 程序中变成同名函数,函数的名字必须在原来方法名的基础上修改,比较容易的做法是把类名字加上去。考虑到方法的重载,参数类型也编码到函数名中,才能保证不会有名字冲突。

(b) 和方法声明相比,函数声明增加一个形参,作为它的第一个形参,该形参的类型就是对应该类的结构类型,该形参的名字通常取 this,传递方式是引用调用。

(c) 和方法调用相比,在函数体中出现的函数调用也要增加一个实参,作为它的第一个实参。若原来是调用本对象的方法,那么新增的实参就是 this;若是调用其他对象的方法,则新增实参是该对象对应的结构变量。

(d) 在方法中对本对象的非静态属性的访问,改成对 this 相应域的访问。在方法中对其他对象的非静态属性的访问不必修改,直接就成了对对应结构变量的相应域的访问。若是对静态属性的访问,则翻译成对 C 的全局变量的访问。

(e) 类的静态方法在定义和调用时,与该类的特定对象无关,因此在翻译时,无须增加表示当前操作对象的参数,只需要按 (d) 中的 (a) 将方法名改成函数名即可。

必须注意一点,对C++程序的语法和语义分析在这个翻译之前进行,因此生成 C 程序时,C++语言的可见性规则已经检查过;C 的可见性规则虽然不一样,这时也没有什么影响了。

例 11.1 如果 m 是类 C 的一个非静态方法,它的原型是"返回值类型 m(形参表)",那么等价于 m 的函数 fm 的原型是(下面给出的语法是非标准的):

返回值类型 fm(C &this, 形参表)

C &this 表示第一个形参的类型和它的名字,传递方式是引用调用。若 m 中有对当前对象的非静态属性 k 的访问,有对 m 本身的递归调用,有对某个对象 o 的非静态方法 n 的调用 o.n,有对 o 的非静态属性 k 的访问 o.k,那么,它们的翻译结果见表 11.1。

类 Point 的方法 x 翻译成等价的函数 x__5Point,其定义是:

```
double x__5Point (Point this) {return this.xc;}
```

方法调用 p.x() 翻译成 x__5Point(p)。

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;
```

```
}
```

表 11.1　类 C 的方法 m 被翻译成函数 fm

	方法	函数
原型	返回类型 m(形参表)	返回类型 fm(C &this,形参表)
调用	m(实参表) o.n(实参表)	fm(this,实参表) fn(o,实参表)
属性访问	k o.k	this.k o.k

方法的名字本身没有作为实现函数的名字,而是扩展成包含所属类的类名的编码和参数类型的编码。把类名编码加到实现函数的名字中是必要的,类名的编码保证了生成的函数名是惟一的。考虑到方法的重载,在C++ 的实现中,参数类型也编码到函数名中。例如,名字 translate__5Pointdd 有下列 5 个部分:

⑴ 方法名 translate;

⑵ 分隔符__;

⑶ 类名的编码 5Point,领头的数表示后面多少个字符属于类名;

⑷ double 类型的编码 d(第一个参数的类型);

⑸ double 类型的编码 d(第二个参数的类型)。

从这一节,我们可以看出把C++ 这样的面向对象语言翻译成 C 语言的可能性。

11.3　继承的编译方案

继承是面向对象语言引入的最重要概念。本节讨论它的实现。

如果类 *B 直接或间接继承类 A,那么类 A 是类 B 的超类,类 B 的对象可以用在几乎所*有类 A 的对象可用的地方。出于效率的考虑,编译器要求类的对象具有某种灵活的结构,因为为了使类 B 的对象可以作为类 A 的对象使用,编译器必须能以一种有效的方式产生类 B 的对象的 A 视图。在这种视图中,编译器关于类 A 的对象结构的假设必须满足。

我们知道,类 A 的虚方法可以在类 B 中被重写。11.1.2 节给出的动态绑定规则要求,如果编译器不能直接确定类 A 的对象 o 的运行时类型,那么该方法应该动态绑定。例如,如果 o 的运行时类型是 *B,那么应该使用 B 中的方法,而不是 A 的方法。在这样的情况下,*编译器必须做出一些准备,使得在程序运行时,被激活方法所期望的视图(即 B 视图)能够

有效地从 A 视图产生。

许多面向对象语言,尤其是一些老的面向对象语言,仅支持单一继承。11.3.1 节先讨论编译单一继承的一种合适方案。11.3.2 节将转向编译重复继承这个更加复杂的问题。

11.3.1 单一继承的编译方案

对于只有单一继承的语言来说,每个类最多从一个类继承。这种语言的继承层次结构是树或森林。

我们从一个例子开始,图 11.2 的程序描述类 PolyLine 和它的派生类 Rectangle:

```
#include graphicalobj.h          /* imported GraphicalObj */
#include list.h                  /* imported lists */
#include point.h                 /* imported Point */
class PolyLine :public GraphicalObj {
    list <Point> points;
  public:
    void translate (double x_offset, double y_offset) ;
    virtual void scale (double factor) ;
    virtual double length (void) ;
} ;

#include polyline.h
class Rectangle :public PolyLine {
    double side1_length, double side2_length;
  public :
    Rectangle (double s1_len, double s2_len, double x_angle = 0) ;
    void scale (double factor) ;
    double length (void) ;
} ;
```

图 11.2 类 PolyLine 和类 Rectangle

出于效率的原因,在矩形类 Rectangle 的定义中引入属性 side1_length 和 side2_length 存储矩形两个邻边的长度,这样允许我们得到一个效率较高的 length 的重新定义。scale 也必须重新定义,因为缩放操作会改变边的长度。此外,translate 可以直接由 PolyLine 的相应定义接管,因为平移操作不改变边的长度。

仍需解释编译器是怎样有效地实现动态绑定的。在上述例子中,PolyLine 的 scale 方法

不能用于缩放矩形,因为它不知道新加的矩形边长属性也由缩放操作改变,因此必须使用 Rectangle 的 scale 方法。我们知道,类 Rectangle 的对象可以作为方法的参数传递,只要超类 PolyLine 的对象在这些地方被允许,例如,它可以用于函数 zoom:

```
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
}
```

该函数首先平移一个图形对象,使得中心点 center 落到原点,然后根据值 zoom_factor 缩放该对象。

如果函数 zoom 作用于矩形,那么 zoom 的体必须调用 Rectangle 的缩放函数,而不是 PolyLine 甚至 GraphicalObj 的缩放函数。然而,zoom 可能在类 Rectangle 被定义前已经被编译并存于库中。也就是说,当编译 zoom 时,编译器不知道运行时在 zoom 体应该激活哪个方法。而且,在 zoom 的不同调用点,应该激活的方法可能还是不一样的。因此,编译器不可能把 scale 绑定到一个具体的方法,而不得不由 zoom 在执行时将 scale 动态地绑定到一个方法。

编译器可以用下面的方案来有效地处理动态绑定。对每个类,编译器建立一个方法表,该表包含了定义在该类中并且必须动态绑定的方法。在 C++ 中,这样的方法表叫做虚函数表,它们包含一个类或它的超类中所有定义为 virtual 的方法的入口。11.2 节提到,每个对象在 C 程序中有对应的结构,现在我们为这样的结构增加一个域,作为第一个域,该域是这样的方法表的指针。编译器把方法名绑定到方法表的索引。当运行时调用某方法时,存储在方法表中相应索引下的函数被激活。继承类的方法表按如下方式产生:首先拷贝其基类的方法表,在这个拷贝中,被重新定义的方法由新的定义覆盖;然后,新引入的方法被追加到这张表上。这就保证了基类中定义的方法名在新类中具有相同的方法表索引。

如果 B 是一个类,A 是 B 的超类,那么类 B 的对象 b 的 A 视图包含两部分:b 的前一部分域域和 b 引用的方法表的前一部分。其中属于 A 视图的 b 的前一部分域由方法表的指针和从 A 继承的属性组成,属于 A 视图的方法表部分包括在 A 及其超类中引入的方法的索引。b 的所有视图由一个指向 b 的指针以同样的方式表示。

我们将用图 11.3 和图 11.4 的例子来解释该编译方案。

图 11.3 给出了 GraphicalObj、PolyLine 和 Rectangle 的方法表。PolyLine 的方法表是从 GraphicalObj 的方法表派生出来的。首先,由 PolyLine 重新定义的方法 translate_PL 和 scale_PL 取代 GraphicalObj 中的相应方法。然后新定义的虚方法 length_PL 被加入。依次地,Rectangle 的方法表从 PolyLine 中的方法表派生。在 Rectangle 中重新定义的方法取代 PolyLine 的相应方法。没有重新定义的方法 translate_PL 仍然保留。编译器把 translate、scale 和 length 分别绑定到方法表索引 0、1 和 2。

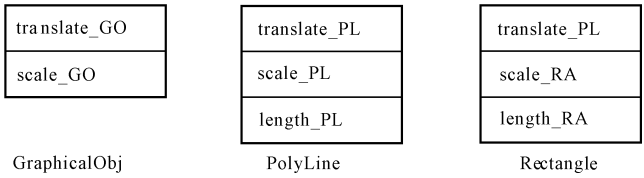


图 11.3 图形对象的不同子类的方法表

图 11.4 给出了 Rectangle 的对象表示。除了自身的状态外,每个这样的对象包含指向类 Rectangle 方法表的指针。

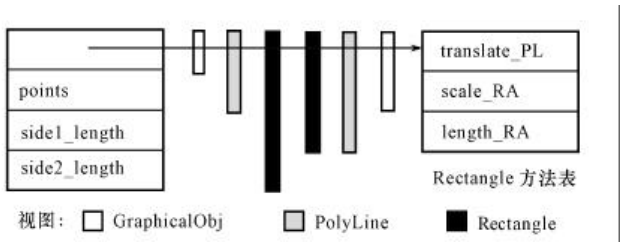


图 11.4 Rectangle 的对象表示

用动态绑定来实现单一继承,每个对象需要一个指针的额外存储空间开销。另外,和每个类关联的方法表需要存储空间。动态绑定引起了运行时方法调用的时间增加,因为通过指针找到方法表以及检索要被激活的方法的位置需要消耗时间。

11.3.2 重复继承的编译方案

单一继承的编译方案比较容易,而重复继承对语言定义和编译器设计来说,都具有很大的挑战性。

在有重复继承的语言中,一个类可以从多个类继承,即一个类可以有多个直接的超类。因此继承层次结构不再是树,而是有向无环图。

重复继承的使用支持这样的编程风格:基本功能用一些很小的基类实现,通过继承用这些基类去构造更加复杂的类。重复继承也可用于抽象类,例如,可以把一个抽象类给出的规范和作为它的实现的一个具体类集成起来。比方,对于一个预先定义了大小的栈,这样的类可以通过从抽象类 Stack 和具体类 Array 的继承关系来实现。

所有会碰到的问题和可能的解决办法都可以通过双重继承来阐明。本节假定类 C 同时从类 B1 和 B2 派生。下面两点引出了语言定义中的问题,在某种程度上也引出了编译器设计的问题:

⑴ B1 和 B2 间的冲突与矛盾。例如，如果在两个基类中，方法或属性使用相同的名字，那么继承将引起冲突。

⑵ 重复继承。例如，若 B1 和 B2 都直接从 A 继承，则 C 将从 A 重复继承。我们将看到，这会引起非常有趣的冲突局面。

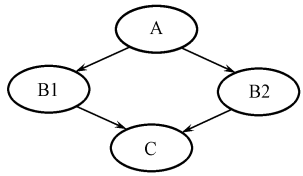


图 11.5　重复继承

问题 ⑴ 基本上是语言定义问题。下面一些解决办法可以独立或组合地应用于语言设计中：

⑴ 将 B1 定义为主要后代，冲突解决优先于 B1。这种办法主要用于解释执行的 Lisp 的面向对象的扩充上。它通过预先定义的次序查找继承层次结构来动态地绑定名字。在我们的例中，先 C，然后 B1，最后 B2。以首先找到的为准。

这种办法并非没有危险，因为可能的冲突并非都是明显的。因此，当上述解决冲突的策略被使用时，并非所有的冲突对程序开发者来说都总是清楚的。

⑵ 语言允许重新命名被继承的特征，因而允许程序员通过显式的干预来解决可能的冲突。使用这种方式的有 Eiffel 语言。

⑶ 语言提供别的显式手段来解决冲突。例如，如果 B1 和 B2 中名字 n 的定义有矛盾，那么 B1：：n 或 B2：：n 将无二义地指明应该使用 B1 还是 B2 中的 n 定义。使用这种方式的有 C++ 语言。

对于这些解决办法，实现起来并无什么困难，只涉及到编译器符号表的组织和管理问题。这里不再继续讨论。

可是对前面的问题 ⑵ 而言，存在两种截然相反的解决方法：

⑴ 被重复继承的类可以有多个实例（见图 11.6）；

⑵ 被重复继承的类只能有一个实例（见图 11.7）。

这两种方法有各自的优缺点。

下面把图形对象的类库作为第一个例子。如图 11.1 所示，类 PolyGon 同时从类 ClosedGraphics 和类 PolyLine 继承，因为多边形是闭折线。这样，类 PolyGon 从两种路径继承类 GraphicalObj。但是，这并不意味着在多边形对象中应该包含两个独立的类 GraphicalObj 的对象作为它的子对象，即像图 11.6 那样。我们扩充前面的例子来解释这一点。在考虑对象移动的可视化时，为了提高效率，通常不是对象本身在移动，而是它的边界矩形在移动。基于这个背景，我们说明每个图形对象必须包含一个指向其边界矩形的指针。这个指针作为一个新的属性加到类 GraphicalObj 中。如果被重复继承的类在派生类对象中有多个实例作为其子对象，那么多边形对象中就包含了两个不同的指针，它们都指向该对象的边界矩形，并且它们的一致性必须维护。幸好，在这个例子中可以保持这两个指针不变，它们总是指向同一个边界矩形。当主对象上的操作引起这个矩形变化时，这两个指针仍然不变。如

果不是边界矩形的指针,而是矩形的顶点坐标加入到类 GraphicalObj 中,就会导致必须维护 GraphicalObj 类在派生类对象中的多个子对象的矩形顶点值的一致性。在这种情况下,我们需要在类 PolyGon 中重新定义从类 PolyLine 继承的方法,以保证从类 ClosedGraphics 继承的 GraphicalObj 的拷贝与从 PolyLine 继承的类 GraphicalObj 的拷贝保持一致。此外,在 PolyLine 或其子类定义的方法也需要维持这两个拷贝的一致。这将变得很复杂。于是我们看出,在重复继承中,当被重复继承的类具有多个实例时,对图形对象这个例子是不合适的,只能在某些情况下可用。

图 11.6 重复继承的多个实例

图 11.7 重复继承的单个实例

再考虑一个例子,在该例中,被重复继承的类的多个实例正好给出了所需的结果。在 GNU C++ 类库中,包含两个用于统计计算的类:SampleStatistics 和 SampleHistogram。类 SampleStatistics 包含确定测量序列平均值、方差和标准偏差的功能。类 SampleHistogram 包含确定直方图的功能。假定我们必须定义这样一个类,它为一个温度测量序列确定平均值和方差,并且为一个气压测量序列作直方图。一种明显的办法是从类 SampleHistogram 和类 SampleStatistics 继承,并且重新命名它们的方法(C++ 不允许重新命名方法,但 Eiffel 是允许的),这样,名字可表示它们是与温度测量还是气压测量有关。在 C++ 的类库中,类 SampleHistogram 定义为从类 SampleStatistics 继承的。于是我们的气压/温度类从 SampleStatistics 继承了两次。在这个例子中,公共继承有多个实例是关键的,否则用于统计计算的两个测量序列将不能被分离。

如前所述,在有些场合下需要公共继承有多个实例,而在另一些场合只需要公共继承的单个实例。在某种情况下,甚至会有这样的需求:对重复继承的某些特征需要单个实例,而对另一些特征则需要多个实例。Eiffel 提供这种灵活性。

C++ 语言包含了对上述两种方法的支持。以图 11.5 所示的重复继承为例,在 C++ 中,当我们将这些类定义为:

```
class A {...};
class B1 :public A {...};
```

```
class B2 :public A {...} ;
class C :public B1, public B2 {...} ;
```

这时,不论是类 B1 或类 B2 都内含一个类 A 的拷贝,这样在类 C 的对象布局中将包含两个独立的类 A 子对象。在C++中,一般直接称这种继承为重复继承。

若将类 B1 和类 B2 设定为从类 A 虚拟继承,再让类 C 从 类 B1 和类 B2 继承,即

```
class A {...} ;
class B1 :public virtual A {...} ;
class B2 :public virtual A {...} ;
class C :public B1, public B2 {...} ;
```

这时,在 C 的对象布局中将只包含一个类 A 子对象。

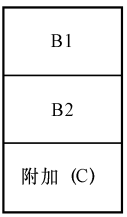


图 11.8　独立的重复继承时的对象结构 (程序视图)

下面考虑仅允许被重复继承的类有多个实例的编译方案。这些方案比同时还允许单个实例的编译方案要简单些,产生的代码效率也高些。我们讨论独立的重复继承的编译方案。

在独立的重复继承情况下,来自基类的继承是相互独立的。相应地,继承类 C 的对象包含基类 B1 和 B2 的完整拷贝,如图 11.8 所示。

如图 11.6 所示,重复继承在下述情况导致冲突和二义:

(1) 当多实例的特征被用于访问、调用和覆盖的时候。

(2) 当类 C 的对象的 A 视图被建立时。因为类 C 的对象包含多个类 A 的子对象。

可见性规则可以在某些情况下帮助避免这些困难。例如,C++允许一个类对它的继承者隐藏它自己的继承性。比如,B1 可以私有地从 A 继承而 C 并不知道这一点,此时 B1 不是 A 的子类,并且 C 中不会由于 A 的多个实例而出现二义。在可见性规则不足以消除二义的地方,需要引入额外的语言构造:在C++中,受限算符“::”可以用于 B1::f 的形式以表示特性 f 属于 B1。此外,可以使用类型转换,如显式地把 C 的对象转换成 B1 的对象。

现在把单一继承的编译方案扩充到这种独立的双重继承场合。在单一继承的情况下(见图 11.4),为了有效地实现方法的动态绑定,在每个对象中加入了一个指针作为该类的第一个成分,该指针指向方法表。这里仍然保持这种方式。

注意,对于类 C 的每个超类 B,编译器必须能够产生类 C 对象的 B 视图。因为 B1 子对象处在 C 对象的开头,因而对 B1 仍可以使用单一继承的办法,即 C 对象的 B1 视图是 C 视图的开头部分(对象成分和方法表都这样)。但是,不能用 C 视图的开头部分作为 B2 视图,因为一个对象的 B2 视图必须有一个方法表指针作为它的第一个成分,该方法表的内容是依据 B2 和 C 确定的;跟随该指针的是 B2 的属性值。这就导致了下面的方法:在 C 对象中,在 B2 属性值的前面加上指向另一方法表的指针;这个方法表由 B2 方法表的拷贝经 C 中重新定义的方法覆盖而产生。于是,B2 视图由一个 B2 引用表示,它指向 B2 子对象的开始。B2

子对象的第一个成分是指向 B2 子对象方法表的指针。对于超类 A 的每个实例,编译器知道 C 对象中对应子对象的偏移。编译器通过把这个偏移加到 C 引用而产生相应的 A 引用。得到的结构显示在图 11.9 中。

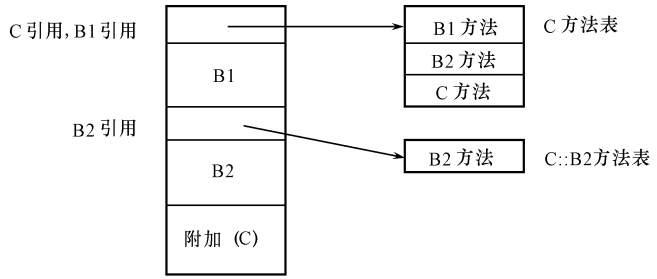


图 11.9 独立的重复继承的对象结构(实现视图)

下面通过例子来理解这种偏移在对象视图切换中的应用。比如,对于语句

```
B2 pb2 = new C;
```

则新创建的 C 对象的地址在赋给 pb2 时必须调整,以指向其 B2 子对象,即提供 C 对象的 B2 视图。遍译时将会产生以下代码:

```
C *temp = new C;
B2 *pb2 = temp ? temp+sizeof(B1) : 0;
```

使得 pb2 是 B2 引用,以便通过它使用 B2 子对象的特征。

当程序员要删除 pb2 所指的对象时,如:

```
delete pb2;
```

如果指针 pb2 所指对象的运行时类型是 C,那么 pb2 的值必须从 B2 引用调整到 C 引用,即调整到 C 视图。然而,由于 pb2 所指对象的运行时类型一般不是静态可确定的,因此上述调整的偏移量一般不是静态可确定的。

更复杂的情况是,C 方法(指在 C 中定义的方法)总是期望得到它为之激活的对象的 C 视图。如果这样的一个方法是覆盖超类 A(更精确地说,超类 A 的一个实例,因为 C 可以含 A 的几个实例)的方法表的一个方法入口,那么在该方法激活时,可能只有 A 的视图是可用的。运行时,我们必须能够从它计算出所需的 C 视图。如果 A 和 C 都是已知的,那么这很容易做:因为视图可由相应的引用表示,并且对于每个 C 对象,A 引用和 C 引用之间的差 d 是一个常数,即在 C 对象中 A 子对象的偏移,因此 C 引用可以从 A 引用减去 d 计算出来。可惜的是,在该方法调用(它可以出现在 B1 或 B2 某方法的代码中)被编译和运行时,C 可能都是未知的。

基于这些原因,编译器把用于确定所需视图的偏移存放在方法表中下邻该方法指针的

地方。即方法表中的每个入口有两个成分:方法指针和偏移,使得该方法所期望的视图可以从该方法激活时的可用视图中生成。

于是第 i 个虚函数的调用操作,由

(＊pb2->vptr[i])(pb2);

改变为:

(＊pb2->vptr[i].fadd)(pb2+pb2->vptr[i].offset);

其中 fadd 是虚函数的地址,offset 是调整视图的偏移。

这种做法的缺点是,不管是否需要用 offset 来调整,所有的虚函数调用都必须这么操作。

图 11.9 描述的办法有一个虽小但很重要的毛病:类 C 的两个方法表包含 B2 方法表的两份(修改过的)拷贝。结果是,存放类 C 的方法表所需的存储空间可能会随 C 定义的复杂度增加而呈指数增长(对于类定义的复杂度,我们指的是类的展开定义(*unfolded definition*)的大小,而类的展开定义是通过把类定义中的基类定义用该基类的展开定义代替而得到的)。

为了避免呈指数增长,必须只使用 B2 方法表的一份拷贝。为做到这一点,把 C 方法表以分布方式存储,如图 11.10 所示。该图隐蔽了一个有意义的细节:B1 和 B2 方法表的拷贝也可以按分布的方式存储。

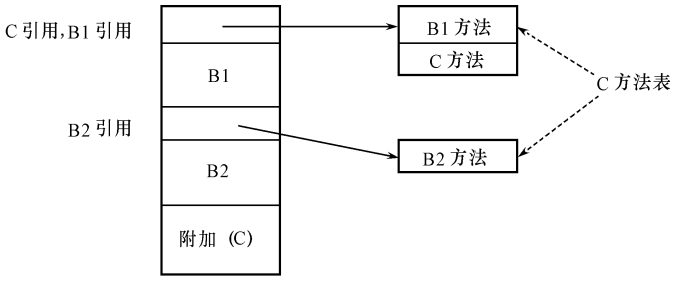


图 11.10　独立的重复继承的对象结构(实际实现)

单个实例的重复继承,以及其他一些形式的重复继承,它们的实现都比较复杂,我们不在此介绍。

习　题　11

11.1　图 11.11 给出图形对象类库的另一片断。根据 11.3.1 节的编译方案,确定该图中定义的类的方法表和方法索引。

11.2 根据独立的重复继承的编译方案,确定图 11.11 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。

```
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;
    }
    // ...
};
```

图 11.11　类 ClosedGraphics、Ellipse 和 Circle

11.3　C++ 中的对象声明语句应如何翻译成 C 语句，如图 11.11 程序中的

```
Point _center;
```

应翻译成什么？

11.4　表 11.1 给出了函数调用的翻译。但是这种翻译方式不能用于虚函数的情况。试说明 11.3.1 节中的虚函数调用 obj.scale (zoom_factor) 应该翻译成什么形式的 C 语句。

* 第 12 章　函数式语言的编译

函数式程序设计语言起源于 Lisp,因此可以追溯到 20 世纪 60 年代初期。到了 20 世纪 70 年代末期,当新的概念和实现方法出现时,这类语言才摆脱了 Lisp 语言的统治,成为一类比较成熟的语言,其代表有 Miranda 和 ML 等。

在函数式语言中,函数是构造程序的基本成分,并且语言还提供一些机制用于构造更为复杂的函数。纯函数式语言禁止使用赋值语句,从而不会产生副作用,其优点是具有引用透明性,有助于程序的等式变换和推理。

函数式程序设计的主要任务在于定义(或构造)函数,以求解所提出的问题。所定义的作为主程序的函数可以包含一些辅助函数(可看作子程序)。计算机按照所定义函数的相应表达式,根据计算规则逐步计算,最后得到所需的结果。表达式中可能包含函数名,计算时可将其相应的定义作为归约规则。

本章介绍一种简单的函数式程序设计语言 SFP 及其实现。

12.1　函数式程序设计语言简介

本节介绍一种简单的函数式程序设计语言 SFP,用它来解释编译函数式程序设计语言的一些原理。我们的兴趣在于定义一个适当的抽象机和为该机器产生代码,而不在于像类型检查这样一些编译事务。例如,多态性概念是编译器的其他部分支持的,因而不把它放入 SFP。另外,我们也不考虑函数定义中包含分情形和模板的情况。

12.1.1　语言构造

现在描述 SFP 的语言构造。为此,假定 SFP 有一些基值类型和这些类型上的运算。这些基值类型是什么对下面的讨论并不重要,但它们必须包括布尔类型。另外,假定这些类型的每个值都只需占用抽象机的一个存储单元,以简化的讨论。SFP 的表达式通过使用内部定义的算符和函数抽象及函数应用,可以直接从基值和变量归纳地构造。SFP 的语法论域总结在表 12.1 中,它的语法总结的图 12.1 中(函数式语言的语法通常用“=”而不是“→”来描述)。

表 12.1　SFP 的语法论域

元素名字	语法论域	
b	B	基值集合，例如布尔值、整数、字符、...
op_{bin}	Op_{bin}	基值上的二元算符集合，例如＋，－，＝，≠，and，or，...
op_{un}	Op_{un}	基值上的一元算符集合，例如－，not，...
v	V	变量集合
e	E	表达式集合

对表 12.1 中的每个论域，我们给它一个“类型化”的名字，这样便于在图 12.1 中引用这些集合上的元素。图 12.1 中的 SFP 大部分构造是清楚的，但是函数抽象、函数应用和联立递归定义需作解释。

构造 λv. e 定义一个一元函数，其中 v 是形式参数，e 是定义表达式。函数 λv. e 应用到表达式 e′写成 (λv. e) e′。根据该语言的语义和实现，它的效果是用表达式 e′(或它的值) 代替形式参数 v 的所有自由出现，或者说把 v 约束到 e′(或 e′的值) 。

$$
\begin{aligned}
e = {} & b \mid v \mid (op_{un}\ e) \mid (e_1\ op_{bin}\ e_2) \\
& \mid (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \\
& \mid (e_1\ e_2) && \text{函数应用} \\
& \mid (\lambda v.e) && \text{函数抽象} \\
& \mid (\text{letrec } v_1 == e_1; && \text{联立递归定义} \\
& \qquad\quad v_2 == e_2; \\
& \qquad\quad \cdots \\
& \qquad\quad v_n == e_n; \\
& \quad \text{in } e)
\end{aligned}
$$

图 12.1　*SFP 的语法*

为了少用括号，我们规定函数应用有最高优先级并且左结合；算术和逻辑算符有通常的优先级。在一个表达式中，λ 抽象选择最大可能的语法表达式作为 λv. e 的体 e，即 e 延伸到表达式的结尾或碰到第一个不能配对的右括号为止。

SFP 的语法允许函数定义和函数应用的嵌套。n 元函数可以用 $f == \lambda v_1.\ \lambda v_2.\ \ldots.\ \lambda v_n.e$ 来定义，一个函数可以用 $fe_1 \ldots e_m$ 的方式应用到若干个变元。为了书写的简便和执行的效率，通常把 n 元函数写成 $\lambda v_1 \ldots v_n.e$ 的形式，并把 $fe_1 \ldots e_m$ 实现为一次函数应用，而不是 m 次应用。

为了保证函数应用的语义十分清楚，有两点必须说明。第一点是参数传递机制，即对于

$e_1 e_2$,传给 e_1 的是什么。第二点是在解释自由变量时,用静态约束还是动态约束,即第6章所讲的静态作用域还是动态作用域。

12.1.2 参数传递机制

在第6章,介绍过值调用、换名调用和引用调用。在函数式语言中,后者没有意义,因为对变量只使用名字和值,不使用地址。在函数式语言中,对于表达式 $e_1 e_2$,参数传递机制确定表达式 e_2 本身还是它的值被传递。我们讨论各种可能性,有三种情况需要区分:

⑴ 值调用

先计算 e_2,然后把它的值传给 e_1。其优点是 e_2 只计算一次。其缺点是即使 e_2 的值不用,它也得计算,这时如果 e_2 的计算不终止的话,将是灾难性的。

⑵ 换名调用

从 e_1 的计算开始,每当需要 e_2 时,计算它的值。于是 e_2 以没有被计算的形式传到 e_1 中相应形式参数出现的地方。其优点是 e_2 只在真正被需要时才计算,因而它比值调用有较好的终止性。其缺点是 e_2 可能计算多次,而每次计算的值都一样。

⑶ 按需调用

又称惰性计算。从 e_1 的计算开始,当第一次需要 e_2 时,计算它的值,也就计算这一次。因此,第一次访问时引起 e_2 的计算,其他的访问用第一次访问时计算的值。这种方式结合了前两种方式的优点。

在 *SFP* 中,对用户定义的函数用按需调用的方式传递参数。

虽然参数传递机制和约束规则并非完全独立,但是在讨论参数传递机制时,我们还是暂不考虑静态约束还是动态约束。

下面通过几个例子来以熟悉 SFP 语言。

例 12.1 表达式

```
letrec x == 2;
       f == λy. x+y;
       F == λg x. g2
in F f1
```

如果是静态约束,f 体中的自由变量 x 引用定义 x == 2,此时,F f1 的值是 4。若是动态约束,在 f 体中访问 x 的值之前把 x 约束到 1,因此结果是 3。

考虑静态约束的实现问题。在例 12.1 中,存在 x、f 和 F 的约束,特别是 x 约束到 2。这样的一列约束通常叫做环境。为了使一个变元(相当于命令语言中的实参表达式概念)中自由变量的正确环境在这些自由变量的每一个出现处都可用,该环境 u 和变元 e 一起传递。

这样形成的二元组 (e,u) 叫做闭包。闭包 (e,u) 中的环境 u 用来保证 e 中的自由变量会被正确地解释。(当然,在值调用场合,有时也需要形成闭包,主要是在有自由变量的函数作为参数传递的时候。)

表达式 letrec $v_1 == e_1 ; \ldots ; v_n == e_n$ in e_0 把 n 个新名字 $v_1, \ldots, v_n$ 填入环境,它们的作用域是 $G = e_0 e_1 \ldots e_n$。可以这样直观地解释,每当这些名字在这个作用域范围内碰到并且需要它们的值时,定义这些名字的等式的右部被使用。

SFP 虽然很简单,但仍可以被用来编程。

例 12.2　表达式

```
letrec fac == λn. if n = 0 then 1 else n * fac (n−1) ;
       fib == λn. if n = 0 or n = 1 then 1 else fib (n−1) + fib (n−2) ;
       one == λn.1
in fib (fac 2) + one (fac 2 −2))
```

SFP 展示了函数式语言的一个重要特征,即高阶函数。函数可以作为函数的变元,也可以作为函数的结果。

例 12.3　表达式

```
letrec F == λx y. x y ;                    函数作为变元
       inc == λx. x+1
in F inc 5                                 值是 6

letrec comp == λf. λg. λx. f (g x) ;       函数作为变元和结果
       F == λx. ... ;
       G == λz. ... ;
       h == comp F G
in h (. .) + F (. .) + G (. .)
```

每一个 n 元函数可以作为一个高阶函数使用。当它作用于 m (m< *n*) *个变元时,其变元个数不足,其结果是一个 (n− m) 元的函数。*

12.1.3　变量的自由出现和约束出现

在命令语言中,变量和类型等的声明以及形式参数的声明引入新的名字。在 SFP 中,名字定义在 letrec 等式的左部和函数定义 $\lambda v_1 \ldots v_n . e$ 的 $\lambda v_1 \ldots v_n$ 中。把前者称为等式定义的名字,后者称为 λ 定义的名字。就函数式语言的语义和它的编译来考虑,在一个表达式中,一个 (全局) 变量的自由出现与它的定义性出现之间的联系是重要的。下面我们先归纳定义一个表达式 e 中自由变量的集合 freevar (e) ,然后定义约束变量的集合。

定义 12.1 表达式中自由出现的变量集合。

$freevar(b) = \varnothing$ 只由基值构成的表达式无自由变量

$freevar(v) = \{v\}$ 只由一个变量构成的表达式含一个自由变量,即它本身

$freevar(op_{un}\ e) = freevar(e)$

$freevar(e_1\ op_{bin}\ e_2) = freevar(e_1) \cup freevar(e_2)$

$freevar(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) = freevar(e_1) \cup freevar(e_2) \cup freevar(e_3)$

$freevar(e_1\ e_2) = freevar(e_1) \cup freevar(e_2)$

$freevar(\lambda v_1 \ldots v_n.e) = freevar(e) - \{v_1, \ldots, v_n\}$

e 中的自由变量 $v_1 \ldots v_n$ 在整个表达式中是受约束的

$freevar(\text{letrec } v_1 == e_1; \ldots; v_n == e_n \text{ in } e_0)$

$$= \bigcup_{i=0}^{n} freevar(e_i) - \{v_1, \ldots, v_n\}$$ (同上)

如果 $x \in freevar(e)$,我们说 x 在 e 中有自由出现。

类似地,可以定义约束出现的变量集合。

定义 12.2 表达式中约束出现的变量集合。

$bdvar(b) = \varnothing$

$bdvar(v) = \varnothing$

$bdvar(op_{un}\ e) = bdvar(e)$

$bdvar(e_1\ op_{bin}\ e_2) = bdvar(e_1) \cup bdvar(e_2)$

$bdvar(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) = bdvar(e_1) \cup bdvar(e_2) \cup bdvar(e_3)$

$bdvar(e_1\ e_2) = bdvar(e_1) \cup bdvar(e_2)$

$bdvar(\lambda v_1 \ldots v_n.e) = bdvar(e) \cup (\{v_1, \ldots, v_n\} \cap freevar(e))$

$bdvar(\text{letrec } v_1 == e_1; \ldots; v_n == e_n \text{ in } e_0)$

$$= (\bigcup_{i=0}^{n} bdvar(e_i)) \cup (\{v_1, \ldots, v_n\} \cap freevar(e_0, \ldots, e_n))$$

如果 $x \in bdvar(e)$,我们说 x 在 e 中有约束出现。

例 12.4 表达式

$$e = (\lambda x\ y.\ (\lambda z.\ x + z)\ (y + 2))\ x$$

中的自由变量和约束变量分别为:

$$freevar(e) = \{x, 2\}$$

$$bdvar(e) = \{x, y, 2\}$$

可以看出,在一个表达式中,一个变量可以既有自由出现,又有约束出现。但是,一个变量的某个具体出现只能是自由的或约束的。

定义 12.3 自由出现和约束出现。

变量 x 的一个出现是自由的，如果这个出现既不是 λ...x... .e 的项 e 的子项，也不是 letrec $v_1 == e_1; \ldots; v_n == e_n$ in e 并且 $x = v_j, e = e_i (0 \le i \le n, 1 \le j \le n)$ 的项 e 的子项。否则 x 的这个出现称为是约束的。

在例 12.4 中，x+z 中的 z 是约束的，而 y+z 中的 z 是自由的。

在 SFP 中采用静态约束，换句话说，在一个表达式或子表达式中，变量的自由出现联系到该变量的第一个外围 letrec 定义或 λ 抽象（即命令式语言的静态作用域）。

12.2　函数式语言的编译简介

在 12.3 节中，我们将介绍一个抽象机 FAM，它的机器语言是 SFP 语言的目标语言。SFP 程序，更一般地说，用按需调用语义和静态约束的函数式语言的程序可以编译成 FAM 的机器语言。该机器有一个栈，生存期符合栈式管理的所有变量都管理在该栈中；它还有一个堆，所有其他的变量存在其中。

本节首先用一连串的例子来启发后面从 SFP 程序到 FAM 程序的编译描述。然后讨论环境和约束。

12.2.1　几个受启发的例子

我们从简单的例子开始，逐步引入到较复杂的例子。一个 SFP 程序最外的表达式叫做程序表达式，换句话说，该表达式的计算给出程序的结果。

例 12.5　程序表达式是

$e = 1+2$

我们希望 e 的编译结果被执行时，它应该把 e 的值留在 FAM 上可访问的存储空间（栈或堆）中。基值类型的结果是可以保留在栈中的，但是，如果结果是一个函数，它将不能存放在栈上。因此，当希望所有的程序表达式都以同样的方式编译，独立于程序表达式的结果类型时，我们把程序表达式的结果统一存放在堆中，在栈顶用一个指针指向堆中的这个位置。

例 12.6　表达式

letrec $x == 1/y; y == 0; z == x$ in $1+2$

该程序表达式必须这样编译，其结果 3 存放在堆中，并且栈顶的指针指向堆中这个位置。该表达式包含变量。在函数式语言中，变量代表值。就实现的效率而言，快速访问这些值是重要的。在编译（而不是解释）实现中，尽可能将可直接访问的存储单元分配给这些变量。在 SFP 实现中，由 letrec 或函数抽象引入的变量将在 FAM 的栈上分配单元。

x、y 和 z 的等式应该这样编译：产生的指令序列并不直接计算这些等式的右部（待将来

真正需要这些值的时候再计算)。另一方面,x、y 和 z 分别约束到这些右部的信息必须在将来需要 x、y 或 z 的时候是可用的。于是,生成的指令序列必须构造 x、y 和 z 的闭包,并分别将指向这三个闭包的三个指针存放在栈中。因为表达式 1+2 的计算不需要访问 *x、y 或 z* 的值,因此这三个闭包根本不计算。

再引入两点改进。y 的等式无须构造闭包,因为它的右部不含自由变量,我们只要让指针指向一个对象,该对象由值 0 和标明该对象是基值的标记组成。另一个改进是让 z 和 x 约束到同一个闭包。

可以看出,上下文清楚地决定了一个表达式应该怎样编译。例 12.6 中的表达式 1+2 必须这样编译:生成的代码产生它的值;而表达式 1/y 的代码必须为它产生闭包。

例 12.7 表达式

```
if (if 1≠2 then true else false) then 3 else 4
```

如果仍希望结果在堆中,那么表达式 3 和 4 都必须按这种方式编译,即它们的值都应在堆中。但是对于条件表达式(if 1≠2 then true else false),FAM 的假转指令 jfalse 希望在栈顶测试它的值,因此,该条件表达式是这样编译的,即生成的指令序列把它的值留在栈上。同样,表达式 true 和 false 也必须这样编译,即它们的值在栈顶。

例 12.8 表达式

```
letrec f == λyz. if z = 0 then 1 else 1/y;
       x == 5
in f 1 (x+1)
```

这个程序可用来研究两个问题:关于函数变元的闭包的构造和以函数为值的表达式的构造。我们先从后者开始。从前面的例子知道,为联立等式产生的代码必须为等式的右部形成闭包,并为左部存储一个指向该闭包的指针。对于像 x == 5 这种等式的优化,我们已经碰到过了。现在来考虑右部是函数定义的情况。当为 λyz. if z = 0 then 1 else 1/y 构造闭包时,我们把它进一步做成 FUNVAL 对象,因为 f 的函数应用代码的第一步必须保证 f 是可应用的函数。FUNVAL 对象和该闭包的区别仅在于它还包含存放变元指针的存储空间。构造闭包和转变成 FUNVAL 对象这两步是由该等式的代码直接完成的。

怎样编译 f 的变元 1 和 *x*+1?FAM 的按需调用语义使得变元必须以闭包的形式传递,这样才可能保证它们的值仅在需要时计算。因此,必须为 f 的变元 x+1 产生闭包。它包含一个指令序列和一个约束向量,该指令序列的执行给出该变元的值,该向量包含一些指针,它们指向出现在该变元中的自由变量的值(或值的表示)。现在的这个向量仅含指向 x 的闭包的指针。f 的等式和 x 的等式的指令序列是在构造 x+1 闭包之前执行的,得到了关于 *f* 的 FUNVAL 对象和对应 x 的基对象,这两个堆对象的指针在栈中,它们被赋给 f 和 x。当构造 x+1 的闭包时,x 的闭包的指针可以从栈中拷贝到该闭包的约束向量中。对于 f 的变

元 1,由前面知道,我们不必创建一个闭包,因为它不含自由变量。

例 12.9　表达式

```
letrec x == 2+1;
       f == λab. g a+ h b;
       g == λx. ...
       h == λy. ...
in f x x
```

该程序表明 SFP 程序的(以闭包或值形式的)表达式的指针可以拷贝任意多份,作为函数变元和全局变量的值等等。因此在 SFP 的实现中,总是值的指针和闭包的指针而不是它们的本身在传递,并且将它们存于约束向量和栈帧(相当于第 6 章的活动记录)中。因为我们拷贝表达式的指针而不是表达式(更确切说是它的闭包)本身,因此每个表达式只有一个实例存在,在 letrec 定义中的对应变量的所有出现都有一个指针指向它。表达式对应变量的首次使用引起闭包的计算是面向对应变量的所有出现。以后的出现就可以直接访问这个值而无须重新计算。

例 12.10　表达式

```
letrec f == letrec x == 2;
            in λy. x+ y
in f5
```

该程序可用来说明命令式语言和函数式语言在局部变量生存期上的区别。在 Pascal 语言中,除了那些由 new 过程在堆上创建的对象外,所有在过程激活时建立的对象都有相同的生存期,它们在过程终止时消失。

对于 SFP 函数,情况不再是这样。在上面的程序中,为了把 f 作用于 5,需要计算由最内的 letrec 构造出的函数。这里,x 是局部于这个 letrec 的。若最内的这个 letrec 已经计算,栈式管理会忘掉属于这个 letrec 的一切东西,包括局部变量。这个例子说明高阶函数的出现需要延长局部变量的生存期,这意味着局部变量必须存放在堆上,组装到 FUNVAL 对象中。

12.2.2　编译函数

12.2.1 节的例子已清楚地表明,同一个表达式在不同的上下文中会编译成不同的指令序列。在前面的例子中遇到了 4 种上下文,我们用不同的字母表示它们:P (program)、B (basic)、V (value) 和 C (closure):

(1) P:编译完整的程序表达式。结果在堆中,栈顶有一指针指向它。它总是最外上下文,编译是在这种上下文中开始的。

⑵ B:结果必须是基值并且存在栈上。例如,处理条件表达式中的条件时,这种上下文会出现。

⑶ V:结果在堆中,栈顶有一指针指向它。这是计算的正常情况。但是对于基值类型的表达式,作为结果的基值先放在栈上,然后将它做成一个堆对象。

⑷ C:结果必须是被编译的表达式的闭包。函数的变元和递归等式的右部总是这种情况。

4种上下文对应4个编译函数P_code、B_code、V_code和C_code,这些函数为与它们相关的上下文产生指令序列。

12.2.3 环境与约束

SFP有两类名字定义:由λ定义的名字,它出现在一个λ抽象的名字序列中;由等式定义的名字,它出现在letrec表达式的一个等式的左部。从例12.5到例12.10可知,一个名字的定义性出现总是关联到一个闭包。当一个等式被编译时,其左部的名字总是关联到其右部的闭包,而λ抽象中的约束名字是在函数应用时关联到该次应用的变元的闭包。

名字的引用性出现应该这样编译:它们获得相关联的定义性出现的值。SFP按需调用语义是这样实现的:第一次碰到一个名字的引用性出现时,其对应的定义性出现的闭包被计算并且该闭包被计算结果覆盖,以后再碰到该名字的引用性出现时就直接取这个值。FAM的指令eval可处理这两种情况。

SFP的编译器遵循通常的哲理,它使用编译时静态可用的信息去有效地管理运行时的动态对象。让我们来看SFP程序的哪些信息是静态的,因而在编译时可用。表12.2列出从程序中可读到的关于函数的静态信息(称为原始静态信息)和从它们可以导出的信息。其中前两种情况用来确定在函数应用时创建的栈帧中的寻址,最后一种情况的信息允许在约束向量中相对寻址。

表12.2 原始信息和导出信息

原始信息	导出信息
变元个数	形参地址(相对于第一个形参的地址)
在函数体中任何一点都可访问的由等式定义的局部名字的集合	这组等式定义的局部名字的地址(相对于第一个等式定义的局部名字的地址)
函数体中全局名字的集合	全局名字的在约束向量中的下标

于是,对于SFP程序中变量的引用性出现,编译器知道它对于直接围绕它的letrec或函数定义是局部(约束)的呢还是全局(自由)的,并且知道哪个相对地址或约束向量中的哪个

下标指派给它。这种静态信息包含在对应的编译环境中(见表 12.3)。

表 12.3　位置和编译环境

名字	论域	备注
p	$P = \{LOC, GLOB\} \times integer$	位置(相对地址或下标)
β	$B = (V \rightarrow P)$	编译环境

在 Pascal 编译器中,地址环境中包含变量的相对地址和它的嵌套深度。在 SFP 编译器中,对应每个变量,编译环境包含它的位置和有关它是自由变量还是约束变量的信息。

当 SFP 程序被编译时,什么时候编译环境会改变?很显然,当函数抽象的体或 letrec 中的表达式开始编译时,新引入的局部变量必须被加入编译环境。

当生成构造闭包或 FUNVAL 对象的代码时,必须把这时的全局变量构造成它们的运行环境。

表达式中的局部变量在 FAM 栈帧上分配存储单元,使用栈帧的相对地址。全局变量存储单元的指针分配在约束向量中,它们的下标指明在运行时指向它们的值的指针在约束向量的什么地方。

12.3　抽象机的系统结构

从本节开始,我们逐步描述 FAM 的系统结构和指令集合,以及把 SFP 编译到 FAM 的编译方案。

FAM 的存储器包含一个程序存储区 PS,每个单位含一条 FAM 指令。某些指令含一个运算对象。程序计数器 PC 总是保留当前指令的地址。在正常情况下,FAM 重复执行下列三步:

(1) 取当前指令;

(2) PC 增 1;

(3) 解释当前指令。

当碰到 stop 指令或发生错误时,FAM 停机。

另外,存储器中还包括一个栈 ST 和一个堆 HP,它们的空间不受限制。我们还假定 FAM 有一个堆管理器,它能在执行 new 过程时分配空间,并能自动回收不再使用的空间。

12.3.1 抽象机的栈

FAM 的栈和第 6 章介绍的活动记录栈类似。SP 是栈顶指针寄存器,它指示已被占用的最高地址。下列 4 种对象都只占栈中一个单元:

⑴ 基值,如整数、布尔值等;

⑵ 栈地址;

⑶ 堆地址;

⑷ 程序指令的地址。

栈被分成若干帧,栈帧在函数应用和计算闭包时建立。前者与 Pascal 的过程激活时建立活动记录是一致的。后者在 Pascal 中无对应的概念,它是按需调用语义要求延迟变元计算引起的,我们稍后会讨论。图 12.2 显示出了两种情况的栈帧结构。

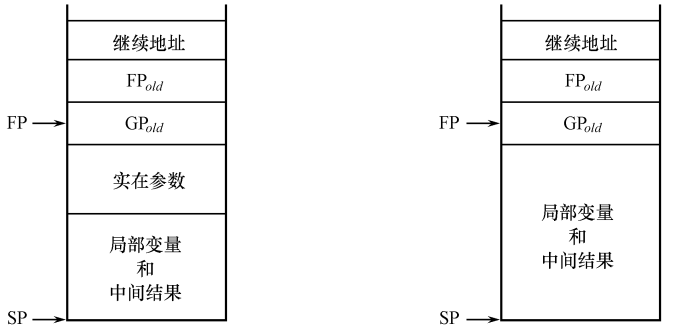


图 12.2 栈帧结构

闭包计算时没有实在参数单元(指向实在参数的指针单元),除此以外,两种情况下的栈帧有同样的结构。栈帧指针寄存器 FP 指示三个有序单元中地址最高的那一个。第一个单元是当前活动(函数应用或闭包计算)结束后继续执行的指令地址,第二个单元是老 FP 的值,第三个单元是所有全局变量构成的向量的指针 GP。FP 和 GP 分别相当于第 6 章所介绍的动态链和静态链,但区别还是有的。在第 6 章 GP 是链的开始,通过它可以找到所有的非局部变量,在这儿 GP 指向一个向量,该向量包含指向一个函数或表达式所有自由变量值的指针。

这种栈帧结构暗示了建立和释放栈帧可用一组固定的指令,见图 12.3,它们可以集成在 FAM 的 mark 和 eval 或 return 指令中。

```
ST[SP+1] := 继续地址;          PC := ST[FP-2];
ST[SP+2] := FP;                GP := ST[FP];
ST[SP+3] := GP;                SP := FP-2;
SP := SP+3;                    FP := ST[FP-1];
FP := SP;
```

(a) 建立栈帧　　(b) 释放栈帧

图 12.3　栈帧的建立与释放指令

12.3.2　抽象机的堆

FAM 的堆所存储的是其生存期和栈管理方式不相容的对象。堆对象有一个标记，用于指示对象的性质。4 种标记 BASIC、FUNVAL、CLOSURE 和 VECTOR 分别表示下列对象。

(1) BASIC：存放基值的单元 b。

(2) FUNVAL：对象表示一个函数结果。它是一个三元组 (cf, fap, fgp)，其中指针 cf 指向程序区中函数体开始的地方，指针 fap 指向函数变元向量，指针 fgp 是函数各全局变量值的指针所组成的向量的指针。这两个向量也存在堆中。FUNVAL 对象是在函数定义的翻译结果被处理时构造的。在这里，变元向量当然是空的，因为现在还没有变元。当 n 元函数应用于 $m\ (m<n)$ 个变元时，FUNVAL 对象的变元向量就变成非空。如前面所提到的，变元不足的应用结果仍然是函数，它仍由 FUNVAL 对象表示，该对象是在把这些变元装进先前的 FUNVAL 对象后得到的。

(3) CLOSURE：对象是一个闭包。它表示一个延迟的计算，因为按需调用的参数传递是把变元的代码和及其全局变量的值组成一个闭包，并且仅在需要时才计算闭包。该对象有两个成分，即代码的指针 cp 和全局变量值的指针向量的指针 gp，后面我们会看到变量的所有定义性出现都被赋了一个闭包。

(4) VECTOR：对象是堆对象指针的向量。该向量存放函数变元的指针，或者存放 FUNVAL 对象或 CLOSURE 对象的全局变量的指针。

标记选择子 tag 用来确定堆对象的标记。函数 size 用来确定 VECTOR 对象的向量长度。

某些 FAM 的指令根据标记来解释堆对象的内容，当它不能作用于堆对象（由于对象类型不对）时会报告错误。有些指令从最高栈单元的内容构造这样的对象，并且把新创建的这个堆对象的指针放在栈顶。这些指令列在表 12.4 中。

12.3.3　名字的寻址

对于 SFP 来说，栈管理和名字的寻址不同于 Pascal 等语言。函数定义的编译必须考虑

函数应用允许变元不足和变元过剩的情况。当编译函数应用 $e\,e_1 \ldots e_m$ 时,编译器并非总是能知道被应用函数的变元个数。例如,e 可以是外围高阶函数的一个形参。

我们考虑编译函数应用 $e\,e_1 \ldots e_m$ 的可能方法。很清楚,为函数应用产生的指令序列必须产生 m 个闭包和一个 FUNVAL 对象。它们的指针必须放在这个函数应用的栈帧中,这些指针在栈帧中有两种安排方式,如表 12.4 所示。

表 12.4 建立堆对象的指令

指令	含义	备注
mkbasic	ST[SP] := new(BASIC:ST[SP])	建立基本堆对象
mkfunval	ST[SP−2] := new(FUNVAL:ST[SP], ST[SP−1], ST[SP−2]); SP := SP−2	建立函数堆对象
mkclos	ST[SP−1] := new(CLOSURE:ST[SP], ST[SP−1]); SP := SP−1	建立闭包
mkvec *n*	ST[SP−n+1] := new(VECTOR:ST[SP−*n*+1], ST[SP−*n*+2], ..., ST[SP]); SP := SP−*n*+1	建立有 *n* 个分量的向量
alloc	SP := SP+1; ST[SP] := new(CLOSURE:NIL, NIL);	建立空闭包

现在讨论这两种情况。图 12.4(a) 的存储安排允许变元和形式参数的寻址相对于 FP 的内容。但是,因为编译函数定义时 *m* 的值是不知道的,而局部变量在栈帧中是邻近这些变元的,因此局部变量的寻址就有困难。此外,如果函数应用提供过多的变元时,也会遇到类似的问题。

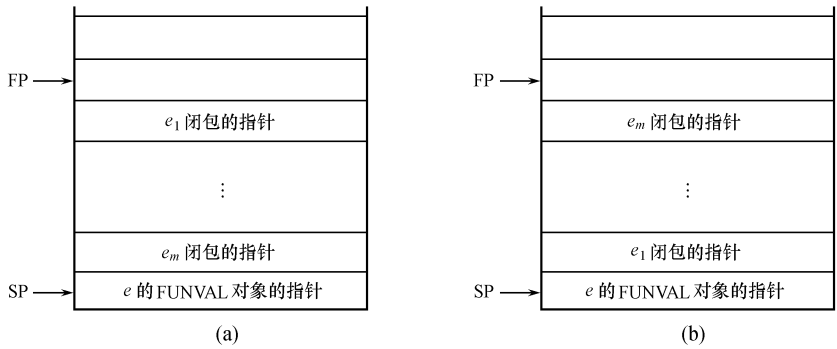


图 12.4 变元在栈帧中的可能安排

图 12.4(b) 的存储安排不允许相对于 FP 的内容来为形式参数寻址。但是形式参数和局部变量可以相对于 SP 的内容寻址。SP 在建立新的局部变量和中间结果时是变化的，因此选择某个动态地址作为形式参数和局部变量相对寻址的基地址。这个地址可选择为正好在 e_1 闭包的指针单元的下面，我们把它叫做 sp_0。这样，函数 $\lambda v_1 \ldots v_n.e$ 的形式参数 $v_1, \ldots, v_n$ 的相对地址必定是 $-1, -2, \ldots, -n$，*而局部变量的地址必定是* $0, 1, 2, \ldots$。

下面的情况出现在一个函数体的处理的开始（由后面描述的 apply 指令产生）。PC 寄存器置到函数体指令区的开始位置，指针 GP 指向恰当的全局约束。栈包含了 3 个有序的单元，它们是继续地址、老的 FP 和老的 GP，还包含 $e_m, \ldots, e_1$ 的闭包的指针单元。SP 指向 e_1 的闭包的指针单元。e_1 的闭包的指针之上的地址是 sp_0，如图 12.5 所示。

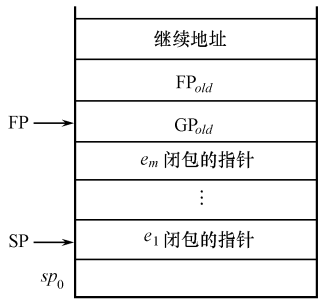


图 12.5　处理函数体之前的栈布局

如果处理函数体，由 letrec 表达式引入的局部变量必须在栈上分配空间，SP 的值相应地增加。还好，SP 的当前值和 sp_0 的值之间的差，对于函数体每一点来说是静态可确定的，因为已出现的新局部变量和中间结果的数目是已知的。这个差值在编译时保存在参数 sl 中，传给前面提到的 4 个编译函数。换句话说，如果编译到函数体中的一个点 a，*sl 的当前值为* sl_a，运行时执行到这一点的 SP 值为 sp_a，那么下列地址关系式成立：

$$sp_a = sp_0 + sl_a - 1 \tag{12.1}$$

因此，生成的指令可以使用编译时确定的值 sl_a 和运行时 SP 的值 sp_a 来计算运行时的值 sp_0。形式参数可以用负的相对地址寻址，而局部变量可以用非负的相对地址寻址。

12.3.4　约束的建立

前面已讨论过，对于每个函数定义以及每个表达式，自由变量集合都是静态可知的。这个集合的元素可按任意次序列表，表中的位置便可作为自由变量的相对地址。这些相对地址在运行时用于访问自由（全局）变量的值。这些值的地址存放在一个向量中，该向量的指针存放在堆中作为 FUNVAL 或 CLOSURE 对象的一部分；在计算闭包或函数应用时，该指针复写到 GP，即运算时可以通过 GP 去寻找该向量的元素。到目前为止，我们还不清楚这些全局变量的值的指针是怎样进入这样的向量的。但有一点是清楚的，当整个程序表达式被编译时，还有被编译的程序开始执行时，自由变量集合和对应的值向量都为空。

假定函数对象或闭包被构造。在这两种情况下，全局变量的值的指针向量必须和它们一起组装。因为 SFP 已规定为静态约束，所以，刚处理的外围函数的形式参数、局部变量和全局变量都应该看成该函数定义或表达式的全局变量。在 12.3.3 节已知道形式参数和局

部变量在运行时可正确寻址。如果归纳地假定能通过 GP 访问当前的全局变量,那么就可拷贝所有新的全局变量的值的指针到栈中,形成一个向量,并把它作为一个堆对象的一部分。

12.4 指令集和编译

通过对问题的分析和直观介绍,现在可以一步步地描述编译和所有的 FAM 指令。前面已提到过,共有 4 个编译函数 P_code、B_code、V_code 和 C_code,它们与所期望的生成代码执行结果的性质相对应。这些 code 函数有三个参数:被编译的表达式 e,变量环境 β 和栈标高 sl。前面已经提到过,sl 定义了被生成的代码执行前 SP 寄存器的值和地址 sp_0 的差,形式参数和局部变量可相对于 sp_0 寻址。

12.4.1 表达式

SFP 程序表达式 e 的编译总是从 P_code 函数的一个应用开始:

$$P\_code\ e = V\_code\ e\ [\ ]\ 0;\ stop$$

因为 e 不能含任何自由变量,所以环境为空。因为栈未作任何填充,所以栈标高 sl 是 0。stop 指令停止机器的执行。

现在编译简单表达式,即仅由基值、算符和 if 构成的表达式。我们考虑需要一个基值作为执行结果的编译,编译函数 B_code 适用于这种情况。

由 B_code 生成的指令列在表 11.5。我们假定对 SFP 的每个一元运算符 op_{un} 和二元运算符 op_{bin},在 FAM 中都有对应的机器指令 op_{un} 和 op_{bin},于是

$$B\_code\ b\ \beta\ sl = ldc\ b$$

$$B\_code\ (e_1\ op_{bin}\ e_2)\ \beta\ sl = B\_code\ e_1\ \beta\ sl;\ B\_code\ e_2\ \beta\ (sl+1);\ op_{bin}$$

$$B\_code\ (op_{un}\ e)\ \beta\ sl = B\_code\ e\ \beta\ sl;\ op_{un}$$

$$B\_code\ (\text{if } e_1 \text{ then } e_2 \text{ else } e_3)\ \beta\ sl = B\_code\ e_1\ \beta\ sl;\ false\ l_1;\ B\_code\ e_2\ \beta\ sl;\ ujmp\ l_2;$$

```
                                    l1 : B_code e1 βsl;
                                    l2 :
B_code e βsl = V_code e βsl ;
               getbasic
```

表 12.5　基值、运算和分支指令

指令	含义	备注
ldb *b*	SP ≒ SP+1; ST[SP] ≒ *b*	装入基值
getbasic	if HP[ST[SP]].*tag* ≠ BASIC then *error* fi ; ST[SP] ≒ HP[ST[SP]].*b*	从堆中往栈上装入基值
op_{bin}	ST[SP−1] ≒ ST[SP−1] op_{bin} ST[SP] ; SP ≒ SP−1	二元运算
op_{un}	ST[SP] ≒ op_{un} ST[SP]	一元运算
false *l*	if ST[SP] = *false* then PC ≒ *l* fi; SP ≒ SP−1	条件分支
ujmp *l*	PC ≒ *l*	无条件分支
ldL *l*	SP ≒ SP+1; ST[SP] ≒ *l*	把标号放入栈中

函数 V_code 的对应情况与上面类似，增加了把作为结果的基值放入堆中并用栈顶指针指向它的指令：

```
V_code b βsl = B_code e βsl;
               mkbasic
V_code (e1 op_bin e2) βsl = B_code (e1 op_bin e2) βsl;
               mkbasic
V_code (op_un e) βsl = B_code (op_un e) βsl;
               mkbasic
V_code (if e1 then e2 else e3) βsl = B_code e1 βsl;
                                     false l1;
                                     V_code e2 βsl;
                                     ujmp l2;
                                   l1 : V_code e3 βsl;
                                   l2 :
```

注意上面条件表达式的编译，e_1 是用 B_code 函数编译的，而 e_2 和 e_3 是用 V_code 函数编译的，因为 e_1 的值需要放在栈上，而 e_2 和 e_3 的值需要放到堆中。

12.4.2 变量的引用性出现

当上下文为 V 时，变量的引用性出现必须编译成访问其值，而当上下文为 C 时，变量的引用性出现必须编译成访问其闭包。在前一种情况下，如果值还没有计算，必须首先从一个闭包计算这个值。

```
V_code vβsl = getvar vβsl;
              eval
C_code vβsl = getvar vβsl
```

必须深入考虑代码生成函数 getvar：

```
getvar vβsl = let (p, i) = β(v)
              in if p = LOC then pushloc sl − i
                            else pushglob i
              fi
```

它为局部变量和形式参数产生 pushloc 指令，为全局变量产生 pushglob 指令。这些指令的定义在表 12.6 给出。

表 12.6 变量值的入栈指令

指令	含义	备注
pushloc *j*	SP ≒ SP+1; ST[SP] ≒ ST[SP−*j*]	把形式参数或局部变量的值的指针压入栈
pushglob *j*	SP ≒ SP+1; ST[SP] ≒ HP[GP].*v*[*j*]	把全局变量的值的指针压入栈

现在考虑对于变量 v 的调用 getvar $v\beta sl_a$，其中 v 是形式参数或局部变量。环境 β 把它约束到二元组 (LOC, i)，其中 i 是非负数 (局部变量) 或负数 (形式参数)。于是 getvar 生成一条指令 pushloc $sl_a - i$。

假定在指令 pushloc 执行前地址关系式 (12.1) 对于参数 sl_a 和 SP 的状态 sp_a 成立，即 $sp_a = sp_0 + sl_a - 1$。执行 pushloc $sl_a - i$ 的效果是：

$$ST[sp] \fallingdotseq ST[sp-(sl_a-i)]$$

其中 $sp = sp_a + 1$，因为 sp 在存储访问前增 1。但是我们仍然有

$$sp-(sl_a-i) = sp_a+1-sl_a+i = (sp_0+sl_a-1)+1-sl_a+i = sp_0+i$$

于是形式参数和局部变量都能正确地装入。

就访问全局变量而言,假定编译时的环境 β为一个向量中的所有全局变量定义一个下标,并假定在运行时寄存器 GP 指向一个向量,该向量填充了根据这种下标定义安排的这些全局变量的指针。于是 pushglob 指令的效果就是从该向量中把一个全局变量的指针压入栈。

12.4.3　函数定义

函数定义可以在两种上下文(值上下文 V 和闭包上下文 C)中编译。前面已介绍过,生成的代码应该构造闭包,但是为了提高函数应用的效率,生成的代码立即建立 FUNVAL 对象。我们回忆一下,这样的对象有三个成分:函数代码的起始地址,变元指针向量(初始时为空)的指针和约束向量的指针。这些指针必须在这个定义点赋值。给全局变量所置的相对地址,加上形式参数的相对地址,就构成了开始编译函数体时的环境:

```
V_code (λv1 ... vn. e) βsl = C_code (λv1 ... vn. e) βsl
C_code (λv1 ... vn. e) βsl =
        pushfree fr βsl;          拷贝全局变量的值的指针
        mkvec g;
        mkvec 0;                  空的变元向量
        ldL l1;                   函数代码的地址
        mkfunval;
        ujmp l2;
    l1: targ n;                   测试变元个数
        V_code e ([v'i | (LOC, −i)]_{i=1}^{n} [v'j | (GLOB, j)]_{j=1}^{g}) 0;
        return n;
    l2:
```

其中

```
fr = [v'1, ..., v'g] = list (freevar (λv1 ... vn. e))
pushfree [v1, ..., vg] βsl = getvar v1 βsl;
                             getvar v2 β (sl+1);
                             ...
                             getvar vg β (sl+g−1)
```

需要对上述代码作些解释。总的来说,由 C_code 函数生成的指令序列用于构造一个 FUNVAL 对象。但与此同时,C_code 函数也必须编译函数定义,产生函数的代码。在上面

的编译方案中,从标号 l_1 到 return n 部分是函数的代码。其余部分用于生成 FUNVAL 对象,并把该函数代码的起始地址 l_1 放到这个 FUNVAL 对象中。ujmp 指令用于跳过这段函数代码。

必须注意函数全局变量的处理。它们是静态可知的,这些变量的集合是用函数 freevar 构造。函数 list 从这个集合构造它的(无重复的)成员表。全局变量的次序也决定了它们相应的值指针的次序。pushfree $[v_1, \ldots, v_g]$ β sl 生成建立该向量的指令序列。它用 getvar 把变量 $v_1, \ldots, v_g$ 的值的指针压入栈,getvar 用环境 β 中的信息进行寻址。函数体的编译开始时,参数 sl 的值为 0,在函数体的代码开始执行前,栈的布局如图 12.5 所示。于是,地址关系式(12.1)在函数体执行前保持,因为我们有

$$sp_a = sp_0 + 0 - 1$$

因为每个 getvar 产生一条指令,它的执行使 SP 增 1,因此 getvar 的参数 sl 也增 1,模拟运行时的 SP 增 1,这就保证了地址关系式(12.1)成立,因此局部变量和形式参数可以正确地寻址。

12.4.4 函数应用

为函数应用所生成的指令序列必须保证:当进入函数执行时,栈的布局必须保持,该布局是编译函数定义时所假定的。在进入前的栈布局由图 12.5 说明。

V_code $(e e_1 \ldots e_m)$ β sl = mark l;
$e \neq e'e''$ C_code e_m β $(sl+3)$;
...
C_code e_1 β $(sl+m+2)$;
V_code e β $(sl+m+3)$;
apply;
l:

mark l 为这个应用建立一个新的栈帧,继续地址 l 和 FP 与 GP 的当前值都被保留。然后,生成的指令序列为变元在堆上建立闭包,并把闭包的指针压入栈中。注意,mark 的执行使 SP 加 3。如果地址关系式在编译和在执行的开始分别都保持,那么它在 e_m 编译前和相应的代码执行前也都保持。因为每个闭包的指针都需要一个存储单元,因此编译每个 e_i $(m \geq i \geq 1)$ 时都让参数 sl 加 1,保证了地址关系式在该编译方案自始至终都保持。表 12.7 是 mark 指令的定义。mark 指令的效果由图 12.6 说明。

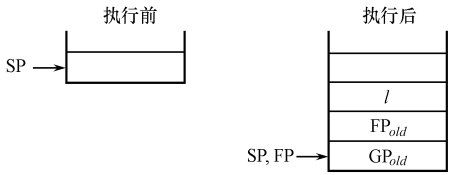


图 12.6 mark 指令执行前后情况

表 12.7　建立一个栈帧

指令	含义	备注
mark *l*	ST [SP+1] ≒ l; ST [SP+2] ≒ FP; ST [SP+3] ≒ GP; SP ≒ SP+3; FP ≒ SP	建立栈帧,保存有关地址

表 12.8 列出了 apply 指令的定义。在变元都从 FUNVAL 对象装入到栈中并且约束指针置好后,它跳到函数体指令的开始点。它执行后栈的布局如图 12.7 所示。

表 12.8　函数应用

指令	含义	备注
apply	if HP [ST [SP]] . *tag* ≠ FUNVAL then error fi; let (FUNVAL : *cf*, *fap*, *fgp*) = HP [ST [SP]] in PC ≒ *cf*; 　GP ≒ *fgp*; 　SP ≒ SP−1; 　for *i* ≒ 1 to size (HP [*fap*] . v) do 　　SP ≒ SP+1; 　　ST [SP] ≒ HP [fap] . v [*i*] 　od tel	函数应用 指向约束 从 FUNVAL 对象中把变元装入栈

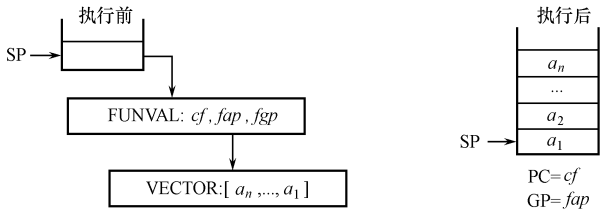


图 12.7　apply 批令执行前后情况

就实现函数的整个任务考虑,我们已经讨论了下列子任务:

(1) 从函数定义生成 FUNVAL 对象,并且在正确的环境下编译函数体。

(2) 用 mark 指令为函数应用建立一个栈帧,随后的存储单元用于存放变元指针。

(3) 用 apply 指令,在建立了 FUNVAL 对象的正确约束后,函数代码开始执行。

此外,还有围绕着函数体的代码的两条指令需要介绍,即 targ 和 return 指令。targ 指令的定义列在表 12.9 中,解释在图 12.8 中。targ 测试提供给函数的变量是否不足:如果不足的话,它把现存的变元组装到 FUNVAL 对象中,并且释放栈帧。

表 12.9 如果变元个数不足,形成 FUNVAL 对象

指令	含义	备注
targ *n*	if SP−FP < *n* then *h* := ST[FP−2] ; ST[FP−2] := new (FUNVAL:PC−1, new (VECTOR: [ST[FP+1], ST[FP+2], ..., ST[SP]]), GP) ; GP := ST[FP] ; SP := FP−2; FP := ST[FP−1] ; PC := *h* fi	变元个数不足

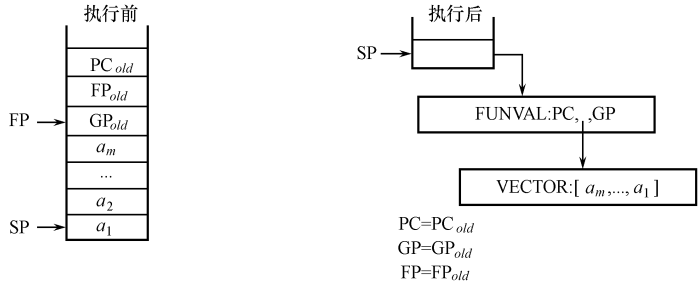


图 12.8 targ 指令发现变元不足 (n> *m*),建立 FUNVAL 对象

return 指令的责任是函数应用和闭包计算结尾的处理。它的参数定义了由函数应用或闭包计算消费的变元个数。它对应两种情况。在第一种情况中,栈帧包含的变元指针个数和被应用函数所需要的一样多,此时,函数结果拷贝到适当的地方,并释放当前栈帧。在第二种情况下,栈帧包含的变元指针个数多于被应用函数所需要的个数,此时的函数应用消费适当个数的变元,其结果是一个函数,再应用到剩余的变元,这些变元的指针仍在栈上。表达式 (λx. (λyz. x+ *y*+ *z*)3) 45 的执行会出现第二种情况,读者可以自行分析一下。

return 指令的定义在表 12.10,它的效果说明在图 12.9 中。

表 12.10　函数应用和闭包计算结尾的处理

指令	含义	备注

return *n*	if SP ≒ FP+1+ *n*
	then
	　PC ≒ ST [FP−2] ;
	　GP ≒ ST [FP] ;
	　ST [FP−2] ≒ ST [SP] ;
	　SP ≒ FP−2 ;
	　FP ≒ ST [FP−1]
	else
	　if HP [ST [SP]] . *tag*≠ FUNVAL then error fi ;
	　let (FUNVAL : *cf*, *fap*, *fgp*) = HP [ST [SP]]
	　in PC ≒ *cf* ;
	　　GP ≒ fgp ;
	　　SP ≒ SP − *n*−1 ;
	　　for *i* ≒ 1 to size (HP [*fap*] . *v*) do
	　　　SP ≒ SP+1 ;
	　　　ST [SP] ≒ HP [*fap*] . *v* [*i*]
	　　od
	　tel
	fi
		结束
		继续地址
		
		结果
		
		变元过多，函数结果应用到剩余变元
		
		消费了 *n* 个变元

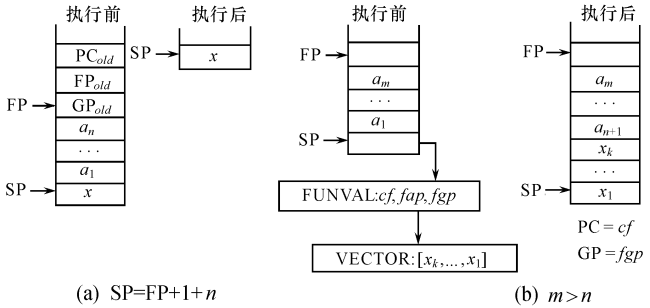


图 12.9　return 指令的两种情况

12.4.5　构造和计算闭包

用 C_code 函数编译表达式，其生成的代码执行时会为该表达式建立一个闭包。堆上的这个 CLOSURE 对象由两个指针组成，分别指向表达式代码和一个向量，该向量的每个元

素指向一个全局变量的值。因为这些值都已给出,因此一个闭包是对它的外围没有其他需求的对象。和函数的处理类似,该表达式的真正编译是在 V 上下文和一个新环境下,该环境只知道全局变量和值为零的参数 sl_0。和函数应用一样,闭包的计算需要建立一个栈帧,在该栈帧中,局部变量可以按前面讨论过的方案寻址。

和编译函数定义一样,编译表达式得到的代码结构包含一个较外的块(建立闭包)和一个较内的块(表达式计算的代码)。

```
C_code e β sl = pushfree fr β sl;        将全局变量的值压栈
                mkvec g;                 把它们做成一个向量
                ldl l1;
                mkclos;
                ujmp l2;
             l1: V_code e [v1, ..., vg] (GLOB, g) g-1 0;
                update;
             l2:
```

其中

$$fr = [v_1, \ldots, v_g] = \text{list}(\text{freevar}(e))$$

利用简单的优化,基本表达式可以处理得更有效。在这种情况下,不必显式地构造闭包。

$$C\_code\ b\ \beta\ sl = V\_code\ b\ \beta\ sl$$

当生成的闭包的值被需要时,执行为 e 生成的代码。这由 eval 指令来完成,如果它在栈顶得到的是一个闭包的指针,那它就建立一个栈帧来计算该闭包。正如上下文 V 所刻画的,这个计算把结果留在堆中,并且在栈中该闭包的指针上面用一个指针指向堆中的这个结果。update 指令(表 12.11 和图 12.10)用这个结果去覆盖该闭包对象。这代表了 FAM 按需调用语义,因为以后再访问时无需重新计算,直接用第一次的计算结果即可。

表 12.11 update 指令

指令	含义	备注
update	HP[ST[SP−4]] := HP[ST[SP]]; PC := ST[FP−2]; GP := ST[FP]; SP := FP−3; FP := ST[FP−1]	覆盖闭包

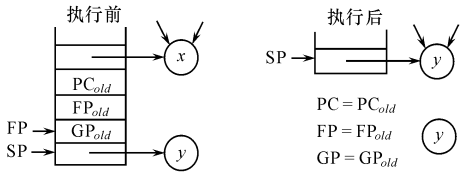


图 12.10　update 指令的效果

图 12.11 说明 eval 指令在栈上的效果，它的定义列在表 12.12 中。

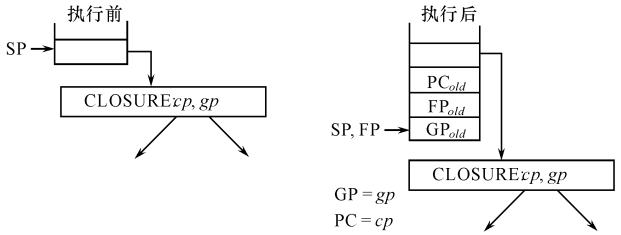


图 12.11　eval 指令的效果

表 12.12　在计算闭包时由 eval 完成栈帧的建立

指令	含义	备注
eval	if HP[ST[SP]].*tag* = CLOSURE then 　ST[SP + 1] ≒ PC; 　ST[SP + 2] ≒ FP; 　ST[SP + 3] ≒ GP; 　GP ≒ HP[ST[SP]].*gp*; 　PC ≒ HP[ST[SP]].*cp*; 　SP ≒ SP + 3; 　FP ≒ SP fi	计算闭包 约束指针 表达式代码的起始地址

12.4.6　letrec 表达式和局部变量

对于 letrec 表达式 letrec $v_1 == e_1; \ldots; v_n == e_n$ in e_0，当在 V 上下文中编译它时，必须产生下列指令和环境：

(1) 为这 n 个表达式 $e_1, \ldots e_n$ 建立闭包的指令序列必须产生。

(2) 计算 e_0 的指令序列必须生成。

(3) 根据全局变量以及 $v_1, \ldots, v_n$,为 $e_1, \ldots, e_n$ 建立同样的环境。

它们由下列编译方案完成：

$V\_code\ (letrec\ v_1 == e_1; \ldots; v_n == e_n\ in\ e_0)\ \beta\ sl =$ repeat n alloc;
$C\_code\ e_1\ \beta'\ sl'$;
rewrite n;
$C\_code\ e_2\ \beta'\ sl'$;
rewrite $n-1$;
...
$C\_code\ e_n\ \beta'\ sl'$;
rewrite 1;
$V\_code\ e_0\ \beta'\ sl'$;
slide n

其中

$\beta' = \beta\ [v_i \mapsto (LOC, sl+i-1)]_{i=1}^{n}$;

$sl' = sl+n$;

repeat $n\ c =$ if $n = 0$ then nocode
else c;
repeat $(n-1)\ c$
fi

逐步来考察这个方案。首先生成 n alloc 指令,它在堆上建立 n 个空对象,并把它们的指针压在栈上。然后为每一个表达式 e_j 产生指令序列

$C\_code\ e_j\ (\beta\ [v_i \mapsto (LOC, sl+i-1)]_{i=1}^{n})\ (sl+n)$;

rewrite $(n-j+1)$

该序列为 e_j 建立闭包,然后覆盖对应的空闭包对象。覆盖指令 rewrite m 的含义见表 12.13。因此,空闭包对象必须已经存在,建立闭包的代码才可以使用这些空对象的指针。但是它不访问这些空对象的成分。

表 12.13 rewrite 指令和 slide 指令

指令	含义	备注
rewrite m	$HP[ST[SP-m]] \doteq HP[ST[SP]]$; $SP \doteq SP-1$	覆盖堆对象
slide m	$ST[SP-m] \doteq ST[SP]$; $SP \doteq SP-m$	拷贝结果

在这儿,对于定义的次序必须有个假定。我们对变量的 C_code 方案已作了改进,用变量的约束作为返回而不是建立一个闭包。若不对定义排序,对于例 12.11,这将是灾难性的,因为这儿空闭包仍将用 pushloc 访问。为了避免这一点,我们对定义进行排序,使得像 *a* == *b* 这样的变量命名总是处于 b 的定义的后面。循环重新命名是没有意义的,因此理所当然被编译器拒绝。

例 12.11　表达式

letrec *a* == *b*;
　　b == 0
in ...;

剩下还必须确定地址关系式 (12.1) 在这种情况下是否仍然保持,如果它在 letrec 编译前和生成的代码执行前分别保持的话。假设 sl 对应的开始值是 sl_0。*n* 次 alloc 使 SP 的值增加 n,结果是,e_1 编译时的参数是 sl_0+n。rewrite 释放栈的最高存储单元,因此其他表达式 *e*, ..., e_n, e_0 的编译仍然有正确的 sl 参数。

局部变量 v_i 分配的地址是 sl_0+i-1。于是 v_1 的地址是 sl_0,v_2 的地址是 sl_0+1,等等。如果该 letrec 是函数体中第一个 letrec,那么 v_1 的相对地址是 0,v_2 的相对地址是 1,等等。因此,在函数 getvar 中用 pushloc 指令的局部变量寻址也是正确的。

习　题　12

12.1　为下列函数写出 SFP 表达式:

(a) 求两个数 *a* 和 *b* 的最大公约数的函数 *gcd*。

(b) 检查一个自然数是否为完全数的函数 *isperfect*。如果一个数等于它真因子的和,那么这个数就是完全数。例如,6 是完全数,因为 6 = 1+2+3。

12.2　确定下面 SFP 表达式中自由变量的集合和约束变量的集合:

(λx. x) (λy. y)
λxy. z (λz. z (λx. y))
(λxy. xz (yz)) (λx. y (λy. y))
λx. x+letrec *a* == *x*;
　　x == *fy*;
　　y == *z*
　　in *x*+ *y*+ *z*

12.3　把下列 SFP 表达式编译成 FAM 抽象机指令序列:

(a) (λx. x+1) 3

(6) letrec $F == \lambda x.\ xy$;
$inc == \lambda x.\ x+1$
in $F\ inc\ 5$

并说明在生成的 FAM 指令执行过程中,栈和堆是怎样变化的。

12.4 把下列 SFP 表达式编译成 FAM 代码:

letrec $fac == \lambda n.$ if $n = 0$ then 1 else $n * fac\ (dec\ n)$;
$dec == \lambda n.\ n-1$
in $fac\ 4$

12.5 考虑如下形式的 SFP 程序(最外表达式是 letrec 表达式):

letrec $v_1 == e_1$;
...
$v_n == e_n$;
in e

变量 v_i 可以按它们在栈开始点的绝对地址来寻址。引入一类称为 ABS 的变量,并且相应地修改编译方案。我们可以进行什么样的优化?

12.6 在这个习题中,我们优化下列形式的递归函数:

$f == \lambda v_1 \ldots v_n.\ \ldots (f e_1 \ldots e_n) \ \ldots$
in ...

如果该递归调用 $(f e_1 \ldots e_n)$ 既不是一个算符的运算对象,又不是函数变元,那么称它为尾递归。在这种情况下,函数值的计算是通过对该函数的这个递归调用来完成的。通常,运行时碰到函数调用,一个新的栈帧必须建立。然而,对于尾递归,当前栈帧是空或者从递归调用返回后不再使用。于是,我们不必建立一个新的栈帧,而是在当前栈帧中计算递归调用。此时,老变元的指针直接由那些新变元的指针代替。

定义相应的指令,并且修改编译方案,使它能识别尾递归调用并且用较有效的方式处理它们。(提示:扩充那些代码函数,增加一个参数,该参数包含有关被编译表达式上下文的信息。)

12.7 修改 FUNVAL 对象的结构以及函数定义和函数应用的代码方案,使得在函数变元不足的情况下,变元的解包和重新打包可以避免。

参 考 文 献

1 Aho A, Sethi R, Ullman J D. Compilers: principles, techniques, and tools. 2nd edition. Addison Wesley, 1986

2 Wilhelm R, Maurer D. Compiler design. Addison Wesley

3 Appel A W. Modern compiler implementation in Java. Cambridge University Press, 1998

4 Appel A W. Modern compiler implementation in C. Cambridge University Press, 1998

5 Steven S Muchnick. Advanced compiler design and implementation. Morgan Kaufmann Publishers, 1997

6 John R Levine. Linkers and loaders. Morgan Kaufmann Publishers, 1999

7 陈意云. 编译原理和技术. 第2版. 中国科学技术大学出版社, 1997

8 陈意云, 张昱. 编译原理习题精选. 合肥: 中国科学技术大学出版社, 2002