21 世纪高等学校计算机规划教材

编 译 原 理

——编译程序构造与实践教程

张幸儿　戴新宇　编著

人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

编译原理 : 编译程序构造与实践教程 / 张幸儿, 戴新宁编著. -- 北京 : 人民邮电出版社, 2010.4
21世纪高等学校计算机规划教材
ISBN 978-7-115-21512-3

Ⅰ. ①编… Ⅱ. ①张… ②戴… Ⅲ. ①编译程序－程序设计－高等学校－教材 Ⅳ. ①TP314

中国版本图书馆CIP数据核字(2009)第191479号

内 容 提 要

本书系统而简洁地介绍编译程序的构造原理，内容主要包括：概论、编译程序构造的基础知识、词法分析、语法分析、语义分析与目标代码生成、中间表示代码与代码优化、程序错误的检查与校正、目标代码的运行，以及虚拟机目标程序的解释程序的编制。各章开始于本章导读，各章末有本章小结、复习思考题以及习题。本书突出实践性，在编译程序构造的各个环节中，提供了具体可行的实现方法和技巧，供读者参考。

本书可作为计算机及相关专业的编译原理课程教材，也可作为计算机软件技术人员、研究生及广大计算机爱好者的参考用书。

21 世纪高等学校计算机规划教材

编译原理——编译程序构造与实践教程

◆ 编　　著　张幸儿　戴新宇
责任编辑　武恩玉

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街 14 号
邮编　100061　　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京华正印刷有限公司印刷

◆ 开本：787×1092　1/16
印张：18.25
字数：477 千字　　2010 年 4 月第 1 版
印数：1 – 3 000 册　　2010 年 4 月北京第 1 次印刷

ISBN 978-7-115-21512-3

定价：31.00 元

读者服务热线：(010)67170985　印装质量热线：(010)67129223
反盗版热线：(010)67171154

出版者的话

计算机应用能力已经成为社会各行业最重要的工作要求之一，而计算机教材质量的好坏会直接影响人才素质的培养。目前，计算机教材出版市场百花争艳，品种急剧增多，要从林林总总的教材中挑选一本适合课程设置要求、满足教学实际需要的教材，难度越来越大。

人民邮电出版社作为一家以计算机、通信、电子信息类图书与教材出版为主的科技教育类出版社，在计算机教材领域已经出版了多套计算机系列教材。在各套系列教材中涌现出了一批被广大一线授课教师选用、深受广大师生好评的优秀教材。老师们希望我社能有更多的优秀教材集中地呈现在老师和读者面前，为此我社组织了这套“21世纪高等学校计算机规划教材-精品系列”。

“21世纪高等学校计算机规划教材-精品系列”具有下列特点。

（1）前期调研充分，适合实际教学需要。本套教材主要面向普通本科院校的学生编写，在内容深度、系统结构、案例选择、编写方法等方面进行了深入细致的调研，目的是在教材编写之前充分了解实际教学的需要。

（2）编写目标明确，读者对象针对性强。每一本教材在编写之前都明确了该教材的读者对象和适用范围，即明确面向的读者是计算机专业、非计算机理工类专业还是文科类专业的学生，尽量符合目前普通高等教学计算机课程的教学计划、教学大纲以及发展趋势。

（3）精选作者，保证质量。本套教材的作者，既有来自院校的一线授课老师，也有来自IT企业、科研机构等单位的资深技术人员。通过他们的合作使老师丰富的实际教学经验与技术人员丰富的实践工程经验相融合，为广大师生编写出适合目前教学实际需求、满足学校新时期人才培养模式的高质量教材。

（4）一纲多本，适应面宽。在本套教材中，我们根据目前教学的实际情况，做到“一纲多本”，即根据院校已学课程和后续课程的不同开设情况，为同一科目提供不同类型的教材。

（5）突出能力培养，适应人才市场要求。本套教材贴近市场对于计算机人才的能力要求，注重理论技术与实际应用的结合，注重实际操作和实践动手能力的培养，为学生快速适应企业实际需求做好准备。

（6）配套服务完善，共促提高。对于每一本教材，我们在教材出版的同时，都将提供完备的PPT课件，并根据需要提供书中的源程序代码、习题答案、教学大纲等内容，部分教材还将在作者的配合下，提供疑难解答、教学交流等服务。

在本套教材的策划组织过程中，我们获得了来自清华大学、北京大学、人民大学、浙江大学、南京大学、吉林大学、武汉大学、哈尔滨工业大学、东南大学、四川大学、上海交通大学、西安交通大学、电子科技大学、西安电子科技大学、北京邮电大学、北京林业大学等院校老师的大力支持和帮助，同时获得了来自信息产业部电信研究院、联想、华为、中兴、同方、爱立信、摩托罗拉等企业和科研单位的领导和技术人员的积极配合。在此，人民邮电出版社向他们表示衷心的感谢。

我们相信，“21世纪高等学校计算机规划教材-精品系列”一定能够为我国高等院校计算机课程教学做出应有的贡献。同时，对于工作欠缺和不妥之处，欢迎老师和读者提出宝贵的意见和建议。

前　言

编译原理是计算机专业的一门重要课程，本课程开设的目的是讲解编译程序的构造原理。由于编译程序是把高级程序设计语言源程序翻译成等价的低级语言目标程序，其处理对象是符号序列，因此可以说编译程序是符号处理的工具，只要是与符号处理相关的领域，都将可能需要应用编译原理中的基本原理。例如，编译原理中所讨论的正则表达式与状态转换图就应用在很多领域。编译原理中的代码优化，尽管是在源程序的内部中间表示一级上进行的，但优化的思想可以为软件开发人员，特别是计算机程序设计爱好者所借鉴。

编译原理课程的特点是理论性强，因为它与形式语言理论紧密相关，可以说是在形式语言理论基础上讨论编译原理，这使得编译原理成为计算机专业最为困难的课程之一。其实，实践性应是编译原理课程更为关注的问题，也就是说，掌握编译程序构造原理的目的是实践编译程序构造。本书着力降低理论难度，围绕编译程序的编制开展讨论，以加强读者的实践应用能力。

本书有如下一些特点。

（1）以 C 语言为背景。紧密结合 C 语言，尽可能处处以 C 语言相关内容为例子进行讨论，使得讲解更有针对性，读者也更易于理解和接受。

（2）突出实践性。对编译过程的每个方面进行论述时，都有计算机实现的讨论，例如文法的存储表示、推导和语法分析树的生成、词法分析、识别程序句型分析，以及翻译方案的实现，都有相当的篇幅。最后还讨论了符号模拟执行虚拟机目标代码的解释程序的编制，给出可运行的解释程序。

（3）明确 C 语言语法成分目标代码结构与其他语言的区别。例如函数调用中，实际参数的计算次序与赋值语句左部变量地址的计算次序等，全书有多个 C 语言程序作为例子，包括应用指针与结构，以及建立结点链的 C 程序编写等，使读者对 C 语言有更深入的认识。

（4）加强讲解编程知识。本书中，除了作为例子的一些经典算法，如排序程序与计算多项式的霍纳方案外，不乏编程的方法和技巧。例如，文法的存储表示、LR 分析表的计算机存储表示与特征变量的引进以及置初值等，以启发读者应用 C 语言编程的能力。

学习是吸收新知识的过程，更重要的是培养和提高分析问题和解决问题能力的过程。一个常见的思维方法是：以简单的实例，通过对它的观察分析，得到解决问题的思路。另一种惯用的思维方法是：从已给的输入（Input）和要得到的输出（Output）出发，设法进行处理（Process），以得到解决方案，这就是 IPO。读者可以体会本书中各种方法和技巧的应用。

总的来说，本书力图以简洁易懂的文字阐述主要的基本概念，用朴实的实例展示实用的方法，按直观的思维方法启发寻找问题的解答。

本书在教学内容安排上，不追求面面俱到，突出从词法分析、语法分析、语义

分析和目标代码生成这条主线，因此词法分析部分突出词法分析程序的实现，语法分析部分突出LR 分析技术，而语义分析部分突出与 LR 分析技术结合的翻译方案。正则表达式、状态转换图与有穷状态自动机、算符优先分析技术等内容可根据读者的情况安排选用。作者建议，把更多的注意力放到计算机实践中去，以培养和提高程序编写能力和计算机实践能力。建议上机实践安排下列几个题目：文法的计算机存储表示、词法分析程序的实现、LR 识别程序句型分析与赋值语句目标代码生成，还可以实现符号模拟执行虚拟机目标代码的解释程序。

由于时间仓促，书中难免有错误与不足之处，欢迎读者批评指正。作者邮箱：

zhangxr0@sina.com　或　zhang_xinger@hotmail.com

最后要感谢武恩玉编辑，是她的大力支持和为本书所做的大量工作，使作者能有本书以飨广大读者，也感谢我的家人，是他（她）们的大力支持，使本书能够顺利完成。

作者

于南京大学计算机技术与科学系

2009 年 6 月 25 日

目　录

第 1 章 概论

本 章 导 读

编译原理课程讨论什么？编译程序又是什么？本章将讨论编译程序概况与编译程序的构造，并简单说明编译程序的实现，使读者对本课程有一个初步而较为全面的认识，充分认识到编译程序与高级程序设计语言的紧密联系，从而领会编译程序的作用、功能和组成，并理解为什么有这样的功能和组成。

1.1 编译程序概况

编译原理课程讨论的是编译程序构造原理。如编译程序是什么？为什么需要编译程序？编译程序又有什么样的结构，应如何去构造？这些都是编译原理课程讨论的内容。

编译程序是一种系统软件，它的功能是把高级程序设计语言源程序翻译成等价的低级语言目标程序。编译程序通常由前端与后端组成，前端进行分析，即词法分析、语法分析与语义分析；后端进行综合，即对经分析生成的内部中间表示生成目标程序，或者对内部中间表示进行优化后再生成目标程序。为了理解编译程序为什么有这样的构造，需要先了解高级程序设计语言及程序的概念。

1.1.1 编译程序的引进

首先理解程序的概念是什么？怎样书写和执行程序？

日常生活中人们有着对程序的一般理解：一系列依次执行的命令。例如开大会，主持人宣布“大会开始、主席发言、×××代表发言、……”，这是大会的一般程序，它由人来执行。现在假定要从键盘上键入 3 个整数，求其平均值，分别用汉语和英语来表达如下。

汉语	英语
提示：“请键入 a、b 与 c 的值”。	Prompt: “Type the values of a, b and c, please”.
键入 a、b 与 c 的值。	Type the values of a, b and c.
求 a、b 与 c 的平均值 ave。	Calculate the average ave of a, b and c.
打印输出平均值 ave。	Print the value of ave.

若用 C 程序设计语言来表达这个过程，则可写出如下代码：

```
printf("Type the values of a, b and c, please:");
```

```
scanf("%d,%d,%d", &a, &b, &c);
ave=( a+b+c) / 3;
printf("ave=%f\n", ave);
```

显然上面前两种表达方式都是容易理解的，对于第 3 种表达方式，熟悉 C 语言的读者也能理解。这是因为所有这些表达方式在书写上都是正确的，即单词和符号都是正确的，顺序安排符合语法，含义也表达清晰。但这些仅是片段，不是完整的表述，尤其是上述第 3 种不是一个完整的程序。

一个 C 语言程序必须遵循 C 语言程序书写规定，按照特定的格式来书写。例如，下述程序才是可运行的，假定该程序保存在文件 ave3.c 中。

```
/*  ave3.c  */
main( )
{ int a,b,c;  float ave;
   printf("Type values of a, b and c, please:");
   scanf("%d,%d,%d", &a, &b, &c);
   ave=(a+b+c)/3.0;
   printf("ave=%f\n", ave);
}
```

现在打开文件 ave3.c，能得到平均值 ave 吗？显然不行。为了得到预期的结果，通常做法如下：运行 Turbo C 2.0，在相应界面主菜单的 File 选项下打开文件 ave3.c，然后按 Ctrl+F9 组合键来运行该文件上的 C 语言程序 ave3.c。

这里的 Turbo C 2.0 就是编译程序，更确切地可以说是编译系统，它把编辑、编译、运行与调试集成于一体。当打开相应文件夹，查看与 ave3.c 相关的文件时，可以发现除了文件 ave3.c 外，还有文件 ave3.obj 与 ave3.exe 等。如果直接运行 ave3.exe，可达到与按 Ctrl+F9 组合键执行 ave3.c 相同的效果。这表明生成了与 ave3.c 等价的可执行程序 ave3.exe。每次执行 ave3.exe，为 a、b 与 c 键入不同的值，就可以得到不同的平均值，如图 1-1 所示。

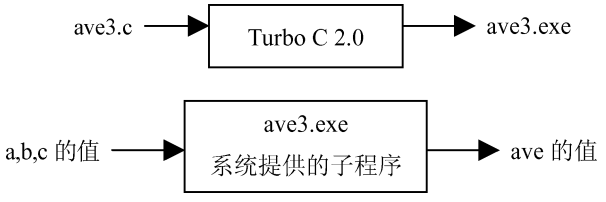


图 1-1

其中，系统提供的子程序在本书以后的部分将称为运行子程序。

通过执行 ave3.c 这样一个特例，可以考察一般情形。

任何语言都有一组记号，并由这些记号组成语法成分的规定，C 语言也不例外。一个 C 语言程序，必须按照 C 语言的拼写约定正确地写出一切符号，并由这些符号正确地组成各个语法成分，最终写出一个完整的程序。然而即使它书写正确，还是不能直接运行，因为它仅是一个符号序列，或者归根结底是一个由字母、数字、括号和标点符号等组成的字符序列。这是不能为计算机所直接接受、理解和执行的，计算机能直接接受、理解和执行的仅是二进位串形式的机器语言指令。

一般情况下，高级程序设计语言程序的执行有两种方式，即解释方式与翻译方式。

当计算机按解释方式执行高级程序设计语言程序时，有一个解释程序，由它对组成程序的各个语句，逐个语句地进行分析，模拟产生相应的运行效果，最终获得所期望的结果。解释方式的

优点是可以边运行边修改，一旦发现程序中有错误，可以立即改正，不足之处是功效较低。例如，当执行如下的循环语句：

```
s=0;
for( k=1; k<=n; k++)
   s=s+k;
```

每次重复执行循环体，便需要重新分析 for 语句，因此 for 语句总共要重复分析 n 次。

当计算机按翻译方式执行高级程序设计语言程序时，并不是直接执行高级程序设计语言程序，而是有一个翻译程序，它把高级程序设计语言程序翻译成等价的低级语言程序，然后执行所生成的低级语言程序，获得所期望的效果，此翻译程序称为**编译程序**。高级程序设计语言程序称为**源程序**，而等价的低级语言程序称为**目标程序**。相应的语言分别称为源语言和目标语言。把源程序翻译成为等价的目标程序的过程也称编译。编译方式执行高级程序设计语言程序的过程如图 1-2 所示。

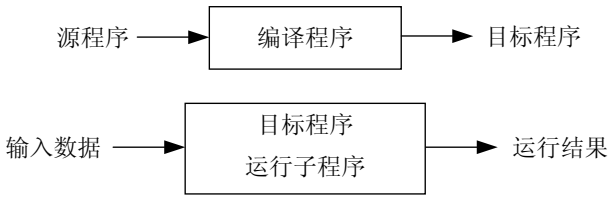


图 1-2

编译方式执行高级程序设计语言程序的优点很明显：功效高。源程序只需编译一次，目标程序可运行任意多次。当编译程序进行目标程序优化工作，改进目标程序质量时，更可大大提高功效；不足之处是修改源程序较为困难，必须对照目标程序中出现的错误，在源程序中找出相应的位置，从而才能进行修改。值得庆幸的是，当前的众多编译程序扩充为集成式的开发环境，其中提供符号调试功能，可以在源程序一级进行调试。当然，作为编译程序的开发人员，就不仅要实现编译程序本身，还得考虑符号调试程序功能的实现。

编译原理课程讨论的是编译程序的构造原理,重点讨论构造编译程序本身的实现技术和方法。希望读者在学习完本课程后，了解编译程序有怎样的结构，理解为什么有这样的结构，掌握编译实现的技术，了解在应用各种编译技术时的应用条件与可能存在的问题。

1.1.2　编译程序与高级程序设计语言的联系

由上述可见，编译程序是执行高级程序设计语言程序的一种手段。作为编译程序的输入，源程序是用某种高级程序设计语言编写的，必须遵循这种语言的书写规定。以 C 语言为例，一个 C 程序由一组函数定义组成，函数定义之外，还可能有一些全局量定义。为简化起见，仅讨论函数定义的情况。

假定有如下的 C 程序：

```
(1)        void main( )
(2)        {  int x, y, max;
(3)           printf("Input values of x and y:");
(4)           scanf("%d,%d", &x, &y);
(5)           if(x>y) max=x; else max=y;
(6)           printf("max=%d\n", max);
           }
```

读者一定熟悉，标有(1)的行是函数首部，其中指明函数名、函数值的类型与参数及其类型。函数体用一对大括号括住，一般包含说明部分与控制部分。显然(2)是说明部分，(3)到(6)是控制部

分。为了描述函数定义如何书写，可以用口语陈述，例如：

“函数定义由函数首部后跟以函数体组成；函数首部则首先是函数值类型（可能省略）后跟以函数名，再跟以用小括号对括住的参数表列（允许没有参数表列）；函数体由用大括号对括住的说明部分和控制部分组成，且说明部分在前，控制部分在后。说明部分与控制部分可能缺省其中一个，甚至两者都缺省。”

显然这样陈述十分累赘，可以改用下列图示的形式描述，如图 1-3 所示。

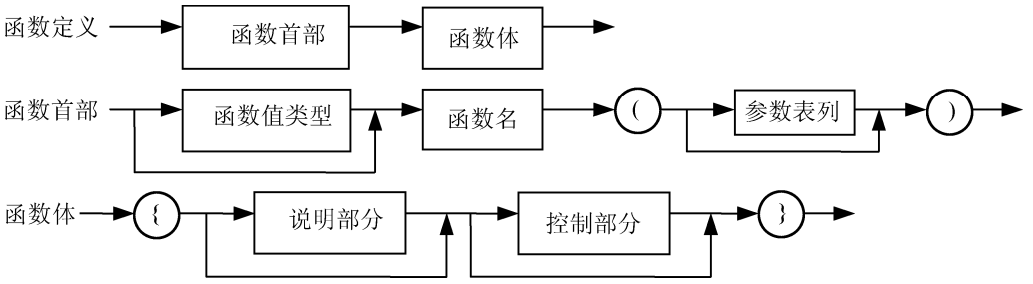


图 1-3

图 1-3 这样的图称为语法图，用图的形式规定了一个 C 语言函数定义的结构，也即书写规则。其特点是直观、形象，一眼便明了一个函数定义应该怎样书写。但其不足是篇幅太大，且是平面图形，难以由计算机处理，特别是不利于编译程序自动构造。

本书将采用称为 **BNF 表示法**的形式来描述书写规则。例如，函数定义可描述如下：

<函数定义>::=<函数首部> <函数体>

<函数首部>::=<函数值类型><函数名>(<参数表列>)

| <函数值类型><函数名>( )

| <函数名>(<参数表列>)

| <函数名>( )

<函数体>::={ <说明部分> <控制部分> } | { <说明部分> } | {<控制部分> }| { }

其中，记号“::=”解释为“定义为”，记号“|”解释为“或”。记号“<”与“>”括住的内容称为语法实体（为简化，可以不用“<”与“>”括住），它们实际上是对程序中某部分所取的语法名称，以便于进行讨论与分析，并不会出现在程序中。上述的描述可解释为：

“函数定义定义为：函数首部，后跟以函数体；函数首部定义为：函数值类型后跟以函数名，再跟以用小括号对括住的参数表列，或者定义为：函数值类型后跟以函数名，再跟以小括号对，或者定义为：…，或者定义为：…。”

显然这与前面的口语描述及语法图描述都是一致的，但它们能以一种确切的、无歧义的简洁方式描述完整的意思。这种写在一行上的记号更便于计算机处理。

概括起来，程序设计语言的书写规则可以用 3 种方式描述：

- 口语
- 语法图
- BNF 表示法

本书将主要采用类似 BNF 表示法的形式来描述语法。

当进一步分析程序的组成时，或者说，分析函数体的组成时，可以进一步引进语句、表达式和量等概念。一般地，一个 C 语言程序可以有如下层次。

C 程序—函数定义—说明部分、控制部分—语句—表达式—量—符号—基本符号

基本符号的例子：void　m　a　i　n　(　)　{　int　x　;　if　else　=　}

符号的例子：void　main　(　)　{　int　x　max　if　else　=　}

量的例子：x　y　max

表达式的例子：x　max=x

语句的例子：max=x;　scanf(″%d, %d″, &x, &y);

说明部分的例子：int x, y, max;

控制部分的例子：if(x>y) max=x; else max=y; printf(″max=%d\n″, max);

其中，基本符号是组成程序的最基本成分。例如，括号(与)是基本符号，关键字 if 与 else 也是基本符号，这些符号都作为一个整体，具有特定的含义，是不可分割的。但对于 C 语言或其他语言，字母与数字也是基本符号，这类符号并不一定具有含义，仅作为符号或符号的组成部分时，才有含义，即表示整个符号的含义。例如，字母 x，作为基本符号，没有含义，若作为标识符，就有了含义。组成 max 的 m、a 与 x 情况一样，单独作为基本符号时并无含义，整个符号 max 才有含义，即是标识符。

概括起来，基本符号可以有含义，也可以没有含义。C 语言的基本符号包括：字母、数字与界限符，界限符则包括关键字、括号、运算符与其他一些专用符号。由这些基本符号可以组成 C 语言标识符、常量、标号与界限符等。

一个程序可以看成是由基本符号组成的，但归根结底是由字符组成的。例如，关键字 if 由字母 i 与字母 f 组成。这就决定了一个编译程序首先必须识别各个基本符号或符号，然后识别出由各个符号组成的语法成分，最后识别出整个程序，进一步依据各个语法成分的含义，生成目标程序。

一个高级程序设计语言通常涉及 4 个方面，即语法、语义、语用与语境。

① 语法指明程序设计语言程序的书写规则。由基本符号组成符号的拼写规则称为词法规则，由符号组成语法成分的规则称为语法规则。词法规则与语法规则全体统称为语法。如前所述，语法可用口语、语法图和 BNF 表示法 3 种形式描述。

② 语义指明按语法规则构成的各个语法成分的含义，尤其是程序的含义。程序设计语言的语义决定了语法成分的含义，即程序执行的效果。语义通常用口语方式来描述。例如，用 BNF 表示法描述的 C 语言赋值语句：

<赋值语句>::=<左部变量>=<表达式> ;

其语义用口语方式描述如下。

“赋值语句的语义是：首先计算右部表达式的值，然后把该值赋给左部变量。必要时先对右部表达式的值进行类型转换，转换到左部变量的类型。”

口语定义方式的不足是明显的：不严格、不精确，并可能存在歧义，不利于保证或验证程序的正确性，特别是自动生成程序。计算机科学中的一个分支：形式语义学，研究的是如何形式地定义一个程序设计语言的语义。但此已超出本书的讨论范围，且这远未实用化，难以高效实现编译。本书不讨论这一范畴的内容，只是采用折衷的语法制导翻译技术，即引进所谓的属性文法来描述语义。

③ 语用一般用来表示语言中的符号及其使用者之间的关系。例如，程序中的注解指明某变量的物理意义与用途等，这就是语用的体现。

④ 语境指明理解和实现程序设计语言的环境，包括编译环境与实现环境。不同的语境明显地影响着语言的实现，例如，C 语言中，整型量通常占 2 个存储字节，这意味着一个整型量最大值

为 $2^{16}-1=32\,767$，当值 32 767 再加 1 时将得到结果− 32 768，这显然是错误的。但如果让一个整型量占 4 个字节，情况就完全不一样，32 767+1，这时不会发生错误。在应用某程序设计语言编写程序时，注意语境问题也是必要的。

1.1.3 编译原理课程的教学内容、教学目标和要求

从前面的讨论可知，编译原理课程讨论的是编译程序的构造原理，而编译程序是为执行高级程序设计语言程序而开发的系统软件，只有深刻理解编译程序与高级程序设计语言之间的紧密联系，才能真正理解编译程序的作用与结构。编译原理课程将讨论下列 3 方面内容：

- 高级程序设计语言的相关知识；
- 编译程序实现的基础知识；
- 编译程序构造原理。

希望读者在学习完本课程后，了解编译程序的作用是什么，功能是什么；理解编译程序的结构如何，为什么有这样的结构；掌握构造编译程序的原理、技术和方法，了解应用时的应用条件，可能会发生什么问题和如何解决这些问题等。

本书结合 C 语言进行讨论，强调了 C 语言有别于其他语言的特殊之处，期望读者通过本书的学习，加深对 C 语言的理解。

本书不仅要求读者掌握相关的概念，更希望读者注意实践性，期望读者应用书中讨论的相关技术与方法，在计算机上实践从实践中加深对概念的理解，尤其是通过实践，培养和提高自己研制程序的能力与计算机实践能力。

1.2 编译程序的构造

1.2.1 编译程序的功能

编译程序是一款系统软件，它实现把高级程序设计语言源程序翻译成等价的低级语言目标程序。等价的含义就是：当源程序正确时，运行目标程序，可达到相同的效果；如果源程序中存在错误，则目标程序有相应的反映。

由于源程序归根结底是一个符号序列，往往把编译程序看作是符号处理的工具。具体说，读者需要了解编译程序需要完成哪些工作？

编译程序与高级程序设计语言紧密相关。从前面的讨论不难了解编译程序必须依次进行的工作如下。

① **词法分析**。编译程序首先对源程序从左到右逐个字符地进行扫描（读入），识别开各个具有独立意义的最小语法单位（即符号或单词，如标识符、常量和关键字等），并把它们转换成称为属性字的内部中间表示，同时进行一些力所能及的其他工作，例如删除注释与非 C 语言成分的字符（如换行字符与空格字符等）。请注意，对于 C 语言来说，由于包含预处理功能，即文件包含（include）命令、宏定义（define）命令，以及条件编译命令，在进行了预处理之后才正式进行词法分析。预处理工作无实质性的困难，并且不是编译程序的“份内”工作，在此不加讨论。

完成词法分析的部分称为**词法分析程序**，也称为**扫描程序**。概括起来，词法分析程序的功能是：扫描（读入）源程序，识别开各个符号，并把它们转换为相应的内部中间表示（属性字），以

及进行删除注释与非语言成分的字符等力所能及的工作。必要时可能进行类似于 C 语言预处理的工作。

② **语法分析**。词法分析时识别开各个符号之后，由语法分析部分根据程序设计语言的语法规则，识别出各个语法成分，最终识别出完整的程序。在识别各类语法成分的同时，也就检查了语法的正确性。当识别出是语法上正确的程序时，生成相应的内部中间表示（通常是语法分析树或其他内部中间表示），如果存在错误，则给出相应的报错信息。

完成语法分析的部分称为**语法分析程序**，或称为**识别程序**。概括起来，语法分析程序的功能是识别出各个语法成分，生成相应的内部中间表示，同时进行语法正确性的检查。

③ **语义分析**。编译程序继语法分析之后进行语义分析，即基于语法分析时输出的内部中间表示，依据各个语法成分的含义进行语义分析。由于一个程序通常由数据结构和控制结构两部分组成，必然对这两部分进行语义分析。对于数据结构，语义分析部分进行的语义分析工作是确定类型和类型检查，确切地说，检查标识符是否有定义，确定标识符所对应数据对象的数据类型等属性，检查运算的合法性及运算分量数据类型的一致性；对于控制结构，根据程序设计语言所规定的语义，对它们进行相应的语义处理，这时可以生成相应的目标代码。例如，对于一个加法运算，当检查了两个运算分量都有定义，它们都能进行加法运算（运算是合法的），且两个运算分量的类型一致（相容）时，可以生成进行加法的目标代码。不言而喻，执行语义分析的同时，还进行一些语义检查，当然这只是静态语义检查，即在编译时刻所能进行的语义检查，例如，检查是否从循环外通过控制转移语句把控制转入循环体。在运行时刻才能进行的语义检查称为动态语义检查，如检查数组元素下标是否越界，以及指针变量是否有初值等，自然不在语义分析时刻进行。为了改进目标程序质量，语义分析时可能不生成目标代码，而是生成另外一种内部中间表示，或称中间表示代码。代码优化阶段就是基于这种中间表示代码进行优化，然后再从优化了的中间表示代码生成目标代码。语义分析工作通常由语义子程序完成。

完成语义分析的部分称为**语义分析程序**。概括起来，语义分析程序的功能是确定类型、类型检查、识别含义与相应语义处理，以及其他一些静态语义检查等。

④ **代码优化**。代码优化指的是为改进目标程序质量而在编译时刻进行的各项优化工作。代码优化通常基于语义分析部分生成的中间表示代码进行，把它变换成功能相同、但功效更高的优化了的中间表示代码。本书的中间表示代码主要是四元式序列。代码优化有与机器无关的优化和与机器有关的优化之分。本书重点讨论与机器无关的优化。除了对中间表示代码进行优化外，也可以对目标代码进行优化，这种优化通常称为窥孔优化。窥孔优化是机器语言级上仅在一个很小的范围内进行的、不太复杂因而代价不是很高的一类优化。

当前的很多高级程序设计语言的编译程序都进行代码优化，C 语言编译程序是突出的优化典型。

完成代码优化的部分称为**代码优化程序**。概括起来，代码优化程序的功能是：在编译时刻基于中间表示代码进行目标程序质量的改进。

⑤ **目标程序生成**。如前所述，语义分析时可以直接生成目标程序，这里的目标程序生成指的是基于优化了的中间表示代码（也即四元式序列）生成目标程序。目标程序的生成与运行目标程序的计算机密切相关。为了学习的目的，本书避免把大量的时间花在对具体计算机细节的了解上，将基于一种所谓的虚拟机来讨论目标程序的生成。

1.2.2　编译程序的组成

由上述可见，编译程序在对一个源程序进行翻译时，将经历词法分析、语法分析、语义

分析、代码优化和目标程序生成等阶段，因此通常把编译程序看作由两个部分组成，即前端与后端。**前端**进行分析，即词法分析、语法分析与语义分析；**后端**进行综合，即代码优化和目标程序生成。前端基本上与计算机无关，而后端，特别是目标程序生成，一般与运行目标程序的计算机密切相关。

对应于编译程序要完成的各项功能，分别有词法分析程序（扫描程序）、语法分析程序（识别程序）、语义分析程序、代码优化程序与目标程序生成程序，把这些程序有机地结合起来，从而实现对高级程序设计语言源程序的编译。显然，编译程序实际上是一个结构庞杂的程序系统，因此往往称为编译系统。但是在实际实现时，并非一定按上述 5 个方面依次实现编译。编译系统的开发人员往往根据实际情况，例如编译系统性能指标设计、参与人员的配备与交付日期的缓急等，把整个编译程序分成若干阶段来开发。每一阶段都以上一阶段的输出作为本阶段的输入进行相应的处理（例外是，第一阶段时是源程序）。每一阶段生成等价的内部中间表示作为输出，这输出要有利于下一阶段的处理。

每个阶段从头到尾读入整个输入并进行处理的过程称为**遍**（或趟）。一个编译程序由几遍完成编译便称为几遍编译程序。如果对源程序从头到尾扫描一次便完成词法分析、语法分析、语义分析、代码优化与目标程序生成等全部工作，就称这样的编译程序为一遍编译程序。一遍编译程序对于功能不太强的小语言还可以，但一般的情况是对各个基本工作进行适当的组合，分成若干遍。例如，词法分析作为第一遍，语法分析与语义分析作为第二遍，代码优化与目标程序生成作为第三遍，这样的编译程序便是三遍编译程序。三遍编译程序的工作示意图，如图 1-4 所示。

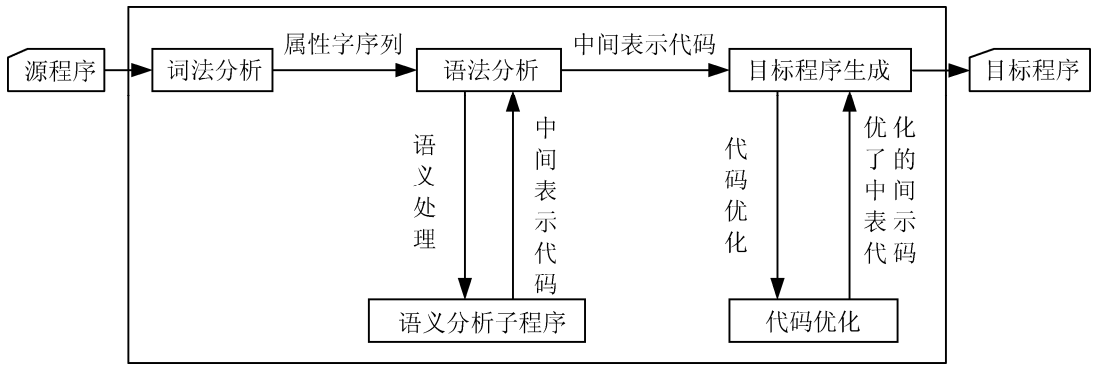


图 1-4

这里要提醒的是：为了目标程序能运行，编译系统开发人员必须研制相应的一些系统配套程序，即运行子程序。只有在运行子程序的支持下，目标程序才能运行，并获得期望的效果。关于运行子程序将在第 9 章 9.4 节中讨论。

1.2.3 编译程序的种类

不同的程序设计语言有各自的编译程序，即使是同一个程序设计语言，也可以按照不同的用途、设计的性能指标，以及开发人员配备情况，研制不同的编译程序。一般，按照用途与侧重面，大致有如下几类编译程序。

① 诊断型编译程序。这类编译程序专门设计来帮助开发与调试程序。当进行编译时，诊断型编译程序不仅可以查出、甚至可以自动校正一些小错误（例如，左右括号不配对、缺少分号等小

错误），而且还可以加强对运行时刻才能查出的错误的诊断（例如，除以零、数组元素的下标越界与指针变量误用等）。当编译时，在目标程序中生成可以查出这类运行时刻错误，并因此放弃程序执行的代码。如果目标程序中过多地包含了这类检查错误的代码，就会导致运行效率较低。因此，合适的做法是：在调试阶段，采用诊断型编译程序。当程序已完成或接近完成时，采用“产品型”编译程序，不包含诊断用代码，从而提高功效。

② 优化型编译程序。这类编译程序有较强的代码优化功能，能生成高功效的目标程序，运行速度快，存储空间少。但鉴于速度和空间往往是相互制约的两个方面，一般为达到速度快，则需要较多的空间，反之亦然。一个典型的例子是：对于使用频繁的变量（如循环控制变量），让它们的值存放在寄存器中，减少存取时间，从而提高功效。但在调用函数时需保护和恢复这些寄存器，又使函数的调用有额外开销。因此，这种优化往往是某种折衷或某种侧重面上的优化。C 语言编译程序是典型的优化型编译程序，给出若干优化级，可以由用户自己选择，使用户以合理的代价获得所期望的优化效果。

③ 可重定位目标型编译程序。一般来说，为一个特定的程序设计语言开发的编译程序，它所生成的目标程序在运行编译程序的特定目标机上运行。要在其他型号计算机上运行此程序设计语言程序，必须另行开发编译程序。鉴于编译程序前端一般与计算机无关，后端生成目标代码时才与特定计算机有关，因此可以这样来开发编译程序，即保留与计算机无关的部分，仅重写编译程序中与目标计算机相关的部分。这样的编译程序称为可重定位型编译程序，对于它，只需重写与某计算机有关部分，便可生成在该计算机上运行的目标程序。

④ 交叉型编译程序。如果在一种型号上运行的编译程序，它生成的却是在另一种特定型号计算机上运行的目标程序，这类编译程序称为交叉型编译程序。

⑤ 应用并行技术的编译程序。对于 C 语言等顺序型程序设计语言，相应的编译程序生成的都是顺序地执行的目标程序。随着计算机技术的发展，多处理器计算机甚至多计算机系统，例如并行系统和分布式系统，得到了较大的发展，支持并行和通信的高级程序设计语言应运而生，相应的编译程序将处理这样的语言，包括处理并行和通信成分，实现共享变量、消息传递和同步等。即使对于顺序程序设计语言，在多处理器计算机支持下，编译程序也探索应用并行处理技术，也就是说，在顺序程序中，自动寻找并行性，即在原有的顺序程序中寻找并行执行的可能性。例如，当对数组的一切元素赋以相同的初值、可相互无关地独立计算两个矩阵的乘积（矩阵）时，就可能并行地处理，这样便可极大地提高程序运行效率。应用并行技术的编译程序实现技术一般是高级课程的内容，超出本书的讨论范畴。

1.3　编译程序的实现

1.3.1　编译程序实现要点

编译程序是把高级程序设计语言源程序翻译成等价的低级语言目标程序的程序。这表明编译程序同源语言与目标语言紧密相关。因此在实践编译程序实现时，首先应明确源语言和目标语言，然后考虑编译程序的实现问题。

从前面的讨论可知，编译过程需完成分析和综合两方面工作，分成若干个部分，不能仓促编写编译程序。作为一个软件系统的开发，宜采用软件工程学原理来实现编译程序的构造，结合编

译程序实际，可列出编译程序实现要点如下：

- 确定待开发编译程序的设计指标；
- 确定源语言与目标语言；
- 总体设计；
  - 确定遍数及各遍的功能；
  - 确定各遍之间的接口、输入输出；
- 设计调试方案与调试实例；
- 确定实现语言与开发平台。

在各遍的实现中，实现要点如下：

- 确定本遍功能；
- 确定编译实现技术；
- 设计相关的数据结构和控制流程图；
- 编写程序。

1.3.2 样本语言的轮廓

编译程序实践的目的是了解编译全过程，掌握编译实现的基本技术和方法。学习过程中不可能实现一个功能齐全、内涵丰富的程序设计语言。本书以下列 C 型语言为样本语言。注意，为了以简洁、清晰而又无歧义的方式描述，采用 BNF 表示法。

1. 基本符号

基本符号::=字母|数字|界限符

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

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

数字::=0|1|2|3|4|5|6|7|8|9

界限符::=运算符 | 括号 | 关键字 | 其他专用符号

运算符::=+ | - | * | / | % | < | <= | > | >= | == | != | & | && | || | ! | =

括号::= (|) | [|] | { | }

关键字::=void | int | float | char | struct | if | else | while | do | for | return

其他专用符号::=, | ;

2. 符号

符号::=标识符 | 无正负号整数 | 字符串 | 界限符 | 标号

3. 语言成分

程序结构::=全局量说明 函数定义部分 | 函数定义部分

函数定义部分::= 函数定义部分 函数定义 | 函数定义

函数定义::=函数值类型 函数名 (参数表列)函数体

| 函数名 (参数表列)函数体

函数体::={ 说明部分 控制部分 }

说明部分::=常量定义 变量说明部分

变量说明部分::=变量说明部分 变量说明 | 变量说明

类型符::=int | float | char

构造类型::=数组类型 | 结构类型 | 指针类型

控制部分::=语句表列
语句表列::=语句表列 语句 | 语句
语句::=赋值语句 | 选择结构 | 迭代结构 | 函数调用语句
赋值语句::=左部变量=表达式
选择结构::=if 语句 | switch 语句
if 语句::=if (表达式) 语句 else 语句 | if (表达式)语句
迭代结构::=while 语句 | do-while 语句 | for 语句
while 语句::=while (表达式) 语句
函数调用语句::=函数名(实际参数表列)

这里所列的并非完整的语法定义，仅让读者了解语言的轮廓。有的是大家所熟知的，不详细列出，有的可以随时在讨论需要时补充，例如，数组、结构和指针等类型变量说明。为了简化讨论，有时也会做出相应的改动。

1.3.3　开发环境

当实现编译程序时，宜采用高级程序设计语言书写和实现编译程序，这样便于阅读检查，从而易于改正错误。例如，用 C 语言实现编译程序，以 Turbo C 2.0 作为开发平台，可以先写出最基本部分，逐步扩充待实现语言的功能，直到实现所确定的整个语言。为了便于编译时的操作，可以按如下方式进行：选择可视化的 VC++（例如 Borland VC++6.0 版本）作为开发平台，即利用它开发编译程序运行时的操作界面，而各部分的主要功能，如词法分析与语法分析等用 C 语言编写。这样将使编译程序的运行有良好的接口界面，又便于编写并调试程序。

本 章 小 结

本章是本书的概论，讨论了编译程序的作用、构造，以及实现，并概括了编译程序的实现要点。

高级语言程序归根结底，是一个符号序列，不能由计算机直接执行。执行高级语言程序的方式可以是解释方式和翻译方式。解释方式，是由解释程序对高级程序设计语言程序逐个语句地模拟执行，产生相应的效果；翻译方式，则是由翻译程序把高级程序设计语言源程序翻译成等价的低级语言目标程序，然后执行目标程序，这个翻译程序就是编译程序。编译程序与高级程序设计语言紧密相关，基于高级程序设计语言的特征，编译程序必然要进行下列工作：词法分析、语法分析与语义分析等，然后进行综合，综合成目标程序，可能之前还会进行代码优化。因此编译程序由前端和后端两部分组成，前端进行分析，后端进行综合。

编译程序是系统软件，也可以认为是符号处理的工具。

按照用途和侧重面，存在多种不同类型的编译程序，然而当前编译程序往往把编辑、编译、调试与运行集于一体，因此可以说，编译程序是高级程序设计语言集成支持环境，是软件开发平台。

本章给出了样本语言轮廓，以便读者了解将涉及的语言的大致轮廓。

通过本章也可大致了解编译原理课程的学习内容、学习目标和要求。

相关概念有源语言、目标语言、BNF 表示法、遍。

复习思考题

1．编译程序是什么？为什么需要有编译程序？
2．编译程序的构造如何？为什么有这样的构造？
3．你使用过哪些编译程序（系统）？你怎样理解所用的编译程序？

第 2 章 编译程序构造的基础知识

本 章 导 读

第 1 章中讨论了编译程序与高级程序设计语言之间的联系，了解到源程序是一个符号序列（符号串）。本章讨论与编译程序构造相关的基础知识，内容包括：符号串与符号串集合、文法与语言，以及句型分析。现在，把程序看作语言的句子，重点讨论如何从识别符号生成句子、反过来又如何识别一个输入符号串是某个给定文法的句子。请注意相应的重要概念，如推导与归约、短语与简单短语、句型与句子，以及二义性等。为了在对源程序翻译过程中，每个阶段都能采用高功效的翻译技术，引进 Chomsky 文法类与相应的语言类，请读者重点关注正则文法与上下文无关文法。关于句型分析，按语法分析树构造的方向，分析技术分自顶向下与自底向上两大类，请关注这两类分析技术各自的基本思想与要解决的基本问题。本章最后，讨论了语法分析树的计算机生成，希望读者熟练掌握文法的存储表示与语法分析树的计算机存储表示及生成要点。

一个高级程序设计语言用来书写程序，通常说，所有可能的程序构成该程序设计语言。如前所述，一个程序归根结底是基本符号序列，或说是字符序列。不言而喻，并不是把一个程序设计语言的基本符号随意组合就能构成一个程序的，可以这样叙述，程序由程序设计语言的基本符号按一定的规则构成，或者说，程序是按一定规则构成的符号序列。

如何构造一个符号序列，它必定是书写上正确的程序？如何由编译程序自动检查一个符号序列确定是书写上正确的程序？这事实上涉及到编译程序的实现技术。下面把程序设计语言作为特定的符号语言，从一般的符号语言角度进行讨论，把程序看作是一个**符号串**。

2.1 符号串与符号串集合

一个程序设计语言中，一切不同的基本符号组成的集合称为**字母表**。一般地，字母表是非空有穷的符号集合。例如，C 语言的字母表包含了一切英文字母、数字与界限符，它至少包含一个符号，而且符号个数必定是有穷的。不同的程序设计语言有不同的字母表。PASCAL 语言的字母表中包含有保留字（关键字）begin、end、integer 与 real 等就明显区别于 C 语言。

为了讨论的简便，字母表中的元素总是统称为符号，且可以有如下形式的表示：字母表 B={0，1}、字母表 Σ={a，b，c}与字母表 E={a，b，+，*，(，)}等。由于字母表是有穷的符号集合，一般在字母表的集合表示中明确写出一切符号。

字母表中的符号组成符号串，**符号串**是由字母表上的符号组成的有穷序列。

【例 2.1】 字母表 B={0，1}上的符号串：0、1、00、01、10、11、001、…，字母表 E={a，

b，+，*，(，)}上的符号串：a、ab、a+b、*ab 与 a+b*(a+b)、…。

符号串由字母表上的符号组成，当把字母表中的若干符号接连放在一起时便组成了符号串。符号串中包含的符号个数称为符号串的长度。例如，a+b 的长度为 3，表示为|a+b|=3，即用两竖表示长度。尽管符号串的长度可以是任意的，但总是有限个数的，甚至可以仅包含 0 个符号。包含 0 个符号，即不包含任何符号的符号串称为**空符号串**，简称空串，通常用 ε 表示。

今后将经常仅关注一个符号串中的一部分，因此引进子符号串的概念。

一个符号串 u 中的相邻若干个符号称为符号串的子符号串，这个子符号串中至少要包含一个符号。更确切地如下叙述：

设有非空符号串 u=xvy，其中符号串 v≠ε，则称符号串 v 为符号串 u 的**子符号串**。显然，|u|≥|v|>0。

【例 2.2】 设有符号串 x=a+b*(a+b)，则 a、a+、b*、*(a 与(a+b)等都是符号串 x 的子符号串。

为了关注一个符号串 u 中的某一部分，可以仅列出这部分，而把其余部分用省略形式表示，例如写成 u=v…，表示对符号串 u 中打头的子符号串 v 感兴趣，写成 u=…v，表示对符号串 u 中结尾的子符号串 v 感兴趣，而写成 u=…v…，表示感兴趣的是子符号 v 出现在符号串 u 中。对于仅包含一个符号 T 的子符号串，自然地写成：

u=T…　　u=…T…　　u=…T

如何生成一个字母表上的符号串？不言而喻，字母表上的任一符号都可看成字母表上长度为 1 的符号串。一个包含多个符号的符号串可以看成是由包含符号数较少的符号串通过一定的运算而生成的。

引进下列两种符号串运算：联结（或称并置）与方幂。

① 符号串的联结（并置）。设某个字母表上有两个符号串 u 与 v，当把 v 的各个符号相继连接在 u 的符号之后，所得到的符号串称为符号串 u 与符号串 v 的联结（并置），记为 uv。

【例 2.3】 字母表 B={0，1}上符号串 u=010 与 v=10 的联结 uv=01010，这里|u|=|010| =3，|v|=|10|=2，|uv|=|01010|=5，显然|uv|=|u|+|v|=5。

由于|ε|=0，显然，εu=uε=u。

② 符号串的方幂。把某符号串相继地重复联结若干次时，便得到该符号串的方幂。设 u 是某字母表上的符号串，把 u 自身联结 n 次，即，v=uu…u（n 个 u），称 v 为符号串 u 的 n 次方幂，沿用一般数学中的记号，记为 $v=u^n$。

【例 2.4】 设字母表{0,1}上有符号串 x=101，则 $x^2=xx=101101$，$x^3=xxx=101101101$。

当 n=0 时，$u^0=\varepsilon$，$|u^0|=0$。

显然，$u^n=u^{n-1}u=uu^{n-1}$，且$|u^n|=n|u|$。当|u|=1 时，$|u^n|=n|u|=n$。

当把一些符号串放在一起时便构成符号串集合，一般的概念如下。

符号串集合：如果一个集合 A 中的一切元素都是某字母表 Σ 上的符号串，则称 A 为字母表 Σ 上的符号串集合。

符号串集合通常用大写字母 A、B、C、…表示。例如用 A 表示字母表 B={0，1}上的一切符号串组成的集合；用 K 表示字母表 E={a，b，c，…，x，y，z}上的一切关键字组成的符号串集合等。符号串集合作为一种集合，可以沿用一般集合的表示法，这有 3 种，即枚举法、省略法与描述法。

① 枚举法。用枚举法表示符号串集合时，明确写出（枚举）符号串集合中的一切元素（符号串）。例如，符号串集合{1,11,111,1111}，这显然适用于元素个数有穷且个数不多时。

② 省略法。当不可能或不便于穷尽符号串集合的一切元素时，可采用省略法，例如

{1,11,111,1111,…，111111111}或者{1,11,111,1111,…}。一般很容易看出省略未写出的元素是什么。

③ 描述法。当明确给出符号串集合中一切元素必须满足的条件而写出元素的一般形式，这时的表示方法称为描述法。例如，上面省略法表示的两个集合用描述法表示时分别是{1^i|1≤i≤9}与{1^i|i≥1}。一般地写作{x|x 满足的条件}。注意，在这种表示方式中，1^i 表示符号串 1 的 i 次方幂，即符号串 1 自身联结 i 次，不涉及含义。也可以表示为{x | x 全由 1 组成，且 1≤|x|≤9}与{x|x 全由 1 组成，且|x|≥1}。

符号串集合可以是有穷的，也可以是无穷的。当不包含任何符号串时，符号串集合是空集，沿用通常的空集表示法，表示为∅。

字母表本身可以看成是字母表上的符号串集合，只是该符号串集合中的所有符号串长度都是 1。因此当把 C 语言基本符号集看作是字母表时，C 语言是该字母表上的一个符号串集合。

在符号串运算的基础上可以引进符号串集合的运算，然后引进字母表的闭包和正闭包的概念。

符号串集合作为一种集合，可以进行各种集合运算，例如，集合并、集合交与集合补等。这里引进集合的乘积和方幂运算。

两个符号串集合 A 与 B 的乘积 AB 定义为：

$$AB=\{xy\mid x\in A，且\ y\in B\}$$

由定义可见，两个符号串集合的乘积是一个符号串集合，其中的所有元素都是由符号串集合 A 和 B 中的所有符号串两两联结而成。

【例 2.5】 设 A＝{a，b，c}，B={0，1}，则 A 与 B 的乘积 AB＝｛a0，a1，b0，b1，c0，c1｝。当 A 中有 m 个元素，而 B 中有 n 个元素时，则 AB 中有 m×n 个元素。

由于对于任何符号串 u，εu=uε=u，有{ε}A=A{ε}=A，但∅A=A∅=∅。

当一个符号串集合 A 自身联结若干次，便得到符号串集合 A 的方幂，有：

$$A^0=\{\varepsilon\}\quad A^1=A\quad A^n=A^{n-1}A=AA^{n-1}\ (n>0)$$

一般符号串集合的方幂太过一般，感兴趣的是字母表的方幂。设字母表 Σ，

$$\Sigma^0=\{\varepsilon\}\quad \Sigma^1=\Sigma\quad \Sigma^n=\Sigma^{n-1}\Sigma=\Sigma\Sigma^{n-1}\ (n>0)$$

由于当 x∈Σ 时，|x|=1，因此当 x∈Σ^n 时，|x|=n （n≥0）。

设有字母表 A，对它作方幂 A^0、A^1、A^2、…、A^n、…。显然，所得符号串集合中的元素（符号串）的长度分别为 0、1、2、…、n、…。

令 $A^*=A^0\cup A^1\cup A^2\cup\cdots\cup A^n\cup\cdots$，则称 A^*为字母表 A 的**闭包**。不言而喻，字母表的闭包中包含了字母表上一切长度 n（n≥0）的符号串。

当不允许上述符号串集合中包含空串，即，长度不允许为 0，令

$$A^+=A^1\cup A^2\cup\cdots\cup A^n\cup\cdots$$

称 A^+为字母表 A 的**正闭包**。

显然，$A^*=A^0\cup A^+$，$A^+=AA^*=A^*A$。A^*与 A^+的区别仅在于一个空串ε。

【例 2.6】 设字母表 B={0,1}，则 $B^+=\{0,1\}^+=\{0,1\}^1\cup\{0,1\}^2\cup\cdots\cup\{0,1\}^n\cup\cdots=$ {0,1,00,01,10,11,001,010,…},即，一切二进制数组成的集合。

设 Σ 为 C 语言字母表，Σ^+为 C 语言字母表上一切符号组成的所有符号串的集合。一个 C 语言程序是字母表 Σ 上的符号按一定规则组成的符号串，C 语言是所有 C 语言程序的集合，因此，C 语言是其字母表 Σ 的正闭包 Σ^+的子集，且为真子集。现在的问题归结为：如何生成字母表上构成程序的符号串。为此引进文法的概念。

2.2 文法与语言

2.2.1 文法及其应用

一个语言按照语法规则来确定其程序的书写。现在称这些规则为**重写规则**，规则的全体称为文法。

1. **重写规则**

重写规则采用类似 BNF 表示法的形式表示。一般写作 U::=u，其中，U 是一个符号，称为重写规则左部；u 是有穷非空符号串，称为重写规则的右部。

【例 2.7】 重写规则的例子。

<字母>::=a　　<字母>::=b　　<字母>::=c

如前所述，记号“::=”解释为“定义为”，因此，上面的可读作“<字母>定义为 a，<字母>定义为 b，<字母>定义为 c”。为简洁起见，缩写为：

<字母>::=a | b | c

其中的记号“|”解释为“或”，因此读作：<字母>定义为 a 或（定义为）b 或（定义为）c。类似地写出：

<数字>::=0 | 1 | 2 | 3

读作：<数字>定义为 0 或 1 或 2 或 3。

<标识符>::=<字母> | <标识符> <字母数字>

<字母数字>::=<字母> | <数字>

读作：<标识符>定义为<字母>或者定义为<标识符>后跟以<字母数字>，而<字母数字>定义为<字母>或<数字>。

这样，利用记号“|”，把若干个重写规则合并成了一个重写规则，称一个规则的右部有多个选择，事实上，一个选择对应于一个重写规则。

在不引起混淆的情况下，重写规则就简称为规则。规则左部的符号称为**非终结符号**，规则中不是非终结符号的符号称为**终结符号**。对于定义程序设计语言的重写规则而言，事实上，终结符号是可能出现在程序中的符号，而非终结符号则是为分析语法成分而引进的名称，它们并不会出现在程序中，记号“<”与“>”用来括住非终结符号，但这不是必要的，例如可以写作：字母::=a | b | c。

在讨论程序设计语言的定义时，有时采用扩充的 BNF 表示法，这里也类似地引进扩充表示法，主要使用 3 类括号作为元语言符号，其用途如下。

① 大括号对{与}：作为元语言符号，用来指明所括住的内容重复若干次。

【例 2.8】 <标识符>::=<字母>$\{<字母数字>\}_0^5$

这表示标识符以字母打头，后跟以 0 个到 5 个的字母数字，即字母后最多跟以 5 个字母或数字，这样就限制了标识符的最大长度为 6。如果不采用扩充表示法，上面的规则必须写成如下形式：

<标识符>::=<字母>|<字母><字母数字>|<字母><字母数字><字母数字>
|<字母><字母数字><字母数字><字母数字>
|<字母><字母数字><字母数字><字母数字><字母数字>
|<字母><字母数字><字母数字><字母数字><字母数字><字母数字>

这远没有扩充表示法简洁。当在大括号对外不指明重复次数时，则表示可出现任意多次，例如：

<标识符>::=<字母>{<字母数字>}

表示标识符可以任意长。

② 中括号对[与]：作为元语言符号，用来指明所括住的内容可能出现也可能不出现。

【例 2.9】　<函数体>::=“{”[<说明部分>][<控制部分>]“}”

这刻划了函数体的大括号对内，由说明部分后跟以控制部分组成，但这两者可能出现也可能不出现，甚至两者都不出现，显然前面的定义与此完全相同，但这里的定义形式更简洁。

注意，此扩充了的规则中，大括号用双撇号括住，这是因为它们都是可以出现在程序中的符号，并非元语言符号。如果不用双撇号括住，便理解为元语言符号，将允许说明部分与控制部分都可以出现任意多次，甚至函数体内将出现如下的结构：

说明部分　控制部分　说明部分　控制部分

这是不允许的，C 语言规定说明部分必须集中在一起。

③ 小括号对（与）：作为元语言符号，用来进行提公因子，即一个规则右部有若干个选择，且其中有公共的因子时，可把公因子提到小括号对外，例如，设有下列规则：

<表达式>::=<表达式>+<项>|<表达式>-<项>

或简单地写为：

```
E::=E+T|E-T
```

可提公因子而改为：

```
E::=E(+T|-T)    或    E::=E(+|-)T
```

又如：

```
S::=if (E) S else S | if (E) S
```

可提因子为：

```
S::=if (E) S (else S | ε )
```

其中ε是空串。

2. 文法

一组重写规则组成的集合称为文法，一般的概念如下：

文法 G[Z]是非空有穷的重写规则集合，其中 G 是文法名，而 Z 称为文法的**识别符号**。

【例 2.10】　关于简单变量说明的文法 G2.1[<变量说明>]可写出如下。

1)<变量说明>::=<类型符><变量名>;
2)<类型符>::=int
3)<类型符>::=float
4)<类型符>::=char
5)<变量名>::=a
6)<变量名>::=b
7)<变量名>::=c
8)<变量名>::=d

该文法名为 G2.1，识别符号是<变量说明>。该文法的一切非终结符号是<变量说明>、<类型符>与<变量名>，而一切终结符号是 int、float、char、a、b、c 与 d。显然，<变量说明>之类非终结符号都不会出现在变量说明中，只有 int、a 与 d 之类终结符号能出现在变量说明中。

一般地，文法的第一个重写规则的左部是识别符号，这样在写文法名时，可不写出识别符号。例如，可仅写出 G2.1。为强调识别符号时，或者没有把识别符号作为第一个重写规则的左部时，文法名后应指明识别符号，如 G2.1[<变量说明>]。识别符号刻划了文法的特征，例如，G2.1[<变

量说明>]指明该文法与变量说明相关，G[<程序>]则与程序相关。

给定一个文法，也就是给定了组成该文法的一组重写规则，因而确定了一切非终结符号组成的集合与一切终结符号组成的集合，两者分别记为 V_N 与 V_T。例如，文法 G2.1 [<变量说明>]的 V_N={<变量说明>, <类型符>, <变量名>}, V_T={int,float,char,a,b,c,d}。显然，$V_N \cap V_T = \varnothing$。$V = V_N \cup V_T$ 称为文法的**词汇表**。词汇表包含了可以出现在文法规则中的一切符号。

概括起来，一个文法 G 的 4 个要素是：识别符号 Z、重写规则集 P、非终结符号集 V_N 与终结符号集 V_T。

除非特别指明，本书中将一直沿用记号 V_N、V_T 与 V，分别表示非终结符号集、终结符号集与词汇表。

3. 应用文法生成句子

一个程序必须遵循程序设计语言的语法规则来书写，现在，就是按文法重写规则来书写程序。如何保证这样书写出来的一定是正确的程序？尤其关心的是如何自动判别所给的一个符号串是书写上正确的程序？通过应用文法生成程序的例子，将不难理解如何解决这两个问题。鉴于定义程序的文法将较为庞杂，仅以前面的关于变量说明的文法为例进行说明。

基于文法 G2.1[<变量说明>]生成变量说明的思路如下：首先从识别符号<变量说明>出发，以识别符号为当前符号串，把该符号串替换为相应规则的右部符号串，即该规则是以识别符号为左部的。然后以所得的新符号串作为当前符号串，把该符号串中的最左非终结符号，替换为以其为左部的相应规则的右部符号串，以所得的新符号串作为当前符号串，类似地重复替换其中的最左的非终结符号，直到最终得到的当前符号串全由终结符号组成为止。

【例 2.11】 试以文法 G2.1 [<变量说明>]为例说明如何生成变量说明。

<变量说明> => <类型符><变量名>; 应用规则 1 进行替换
=> int <变量名>; 应用规则 2 进行替换
=> int a; 应用规则 5 进行替换

或者， <变量说明> => <类型符><变量名>; 应用规则 1 进行替换
=> float <变量名>; 应用规则 3 进行替换
=> float c; 应用规则 7 进行替换

至此，得到两个变量说明：int a;与 float c;，它们不言而喻在书写上是正确的。显然，应用不同的规则进行替换，就可以得到不同的变量说明。

由于识别符号可以是各种各样的，既可以是<变量说明>，也可以是<表达式>，还可以是<程序>，从识别符号生成的符号串相应地分别是变量说明、表达式或程序。因此，概括地引进一般的概念：句子。

从识别符号出发，通过重复应用重写规则，对非终结符号进行替换而生成的终结符号串称为文法的**句子**。

归纳起来，应用文法生成句子的步骤如下：

步骤 1 从识别符号出发，以识别符号作为当前符号串；

步骤 2 把当前符号串中最左的非终结符号，替换为以该非终结符号为左部的重写规则的右部符号串；

步骤 3 以替换所得的新符号串作为当前符号串，重复步骤 2，直到当前符号串中再无非终结符号可被替换而结束。

最终所得的就是句子，它全由终结符号组成。

由于总是从识别符号开始进行替换，有时把识别符号称为开始符号。

这样生成的变量说明可以有多少个？显然仅 3×4=12 个。现在对文法 G2.1[<变量说明>]做一些修改。

G2.2[<变量说明>]：

<变量说明>::=<类型符><变量名>；　　（1）
<类型符>::=int　　（2）
<类型符>::=float　　（3）
<类型符>::=char　　（4）
<变量名>::=<字母>　　（5）
<变量名>::=<变量名><字母>　　（6）
<字母>::=a　　（7）
<字母>::=b　　（8）
<字母>::=c　　（9）
<字母>::=d　　（10）

可以如下地生成变量说明 int ab; 。

<变量说明>=><类型符><变量名>；　　应用规则（1）
=> int <变量名>；　　应用规则（2）
=> int <变量名> <字母>；　　应用规则（6）
=> int <字母> <字母>；　　应用规则（5）
=> int a <字母>；　　应用规则（7）
=> int a b；　　应用规则（8）

不难发现，如果在第 4 步时不是应用规则 5，而是应用规则 6，则将生成 int <变量名> <字母> <字母>，最终将生成由 3 个字母组成的变量名。每重复应用规则 6 一次，变量名中就将多一个字母，这表明将可以生成无限多个变量说明。

尽管文法中仅包含有限多个重写规则，但可以生成无限多个句子（终结符号串）。因此说，文法是以有穷的方式描述（构造）潜在地无穷的符号串集合的手段。

要说明的是，上述步骤中总是对当前符号串中最左的非终结符号进行替换，事实上，也可以总是对最右的非终结符号进行替换,甚至对当前符号串中的任何一个非终结符号进行替换都可以，建议按有规律的方式进行替换。

上述进行替换的过程称为推导或生成句子的过程，其中每一步称为直接推导。

如果对于文法 G，有两个符号串 v 和 w，v，w∈V^+，能写出 v = xUy 与 w = xuy，其中，x、y∈V^*，且 U:: = u 是 G 中的规则，则称符号串 v 直接推导到或直接产生符号串 w，记作 v =>w，或者称 w 是 v 的**直接推导**，也可以称符号串 w **直接归约**到符号串 v。

由于其中 x、y∈V^*，可以有 x=y=ε，因此，对文法 G 的任何规则 U::=u，都可以有 U=>u。

可以这样来理解直接推导的概念，对于文法 G 的一个符号串 v=xUy，把其中的某个非终结符号 U 替换为相应规则 U::=u 的右部符号串 u，就是进行了一次直接推导，即

xUy=>xuy

反过来，把符号串 w=xuy 中的一个子符号串 u，替换为以 u 为右部符号串的规则的左部非终结符号 U，就是进行了一次直接归约，即

xUy=>xuy

如果对于符号串 v 与 w 存在一个直接推导序列 v=>u_1=>u_2=>…=>u_{n-1}=>w,则称符号串 v **推导**到（或产生）符号串 w，或称 w **归约**到 v，记作 v=>+ w，称此直接推导序列是长度为 n 的推导，

且称符号串 w 是相对于符号串 w 的一个**字**。

例如，从文法 G2.1 可有下列推导(归约)：

<类型符><变量名>;=>+ int <变量名>; 长度为 1
<类型符><变量名>;=>+ int a; 长度为 2
<变量说明>=>+ int <变量名>; 长度为 2
<变量说明>=>+ int a; 长度为 3

如果对于符号串 v 和 w，有

$v \Rightarrow^+ w$ 或 $v = w$

则称符号串 v **广义推导**到符号串 w，或称 w **广义归约**到符号串 v，记作

$v \Rightarrow^* w$

显然，直接推导=>的长度为 1，推导=>+的长度≥1，而广义推导=>*的长度≥0。

一般地，任意一个符号串都可以对它进行推导来产生新的符号串，只需把其中的某个非终结符号替换为以其为左部的规则的右部符号串，但实际上关心的是从文法的识别符号推导生成的符号串，这种符号串称为**句型**。

设 G[Z]是一个文法。如果符号串 x 是从识别符号 Z 推导所得，即

$Z \Rightarrow^* x \quad x \in (V_N \cup V_T)^+$

则称该符号串 x 是文法 G 的句型。如果一个句型 x 全部由终结符号组成，即

$Z \Rightarrow^* x \quad x \in V_T^+$

则称该句型 x 是文法 G 的句子。

因此，从识别符号出发，应用文法的重写规则进行推导产生的符号串都是相应文法的句型。识别符号可看作文法的最简单的句型，句子则是全由终结符号组成的句型。例如，例 2.11 中<变量说明>、<类型符><变量名>；与 int <变量名>；、int a；等都是句型，其中仅 int a；为句子。

当为 C 语言构造一个文法 G[<程序>]时，一个 C 语言程序将是该文法 G[<程序>]的句子，不言而喻，其中是不会包含<函数定义>之类非终结符号的，必定全由作为终结符号的基本符号组成。

假定 v=xUy 是文法 G 的一个句型，G 中有重写规则 U::=u，则 w=xuy 也是文法 G 的一个句型，这是因为 v=xUy 是句型，所以 Z=>* xUy，应用规则 U::=u 进行直接推导，所以 Z=>* xUy=>xuy，即 Z=>* xuy。这里 w=xuy 中的子符号串 u 称为 w 中相对于 U 的**简单短语**。类似地，如果符号串 v=xUy 是文法 G 的一个句型，且有 U=>+ u，则 w=xuy 也是文法 G 的一个句型。这时，w=xuy 中的子符号串 u 称为 w 中相对于 U 的**短语**。

当从句型 w=xuy 出发来理解其中的简单短语或短语时，必须理解 v=xUy 与 w=xuy 的关系。由于 U=>+ u 或 U=>u，

$Z \Rightarrow^* xUy \Rightarrow^+ xuy$ 或 $Z \Rightarrow^* xUy \Rightarrow xuy$

因此 u 是 w=xuy 中可（直接）归约的子符号串，当其（直接）归约后得到的新符号串 xUy 仍然是句型。可（直接）归约与（直接）归约后仍为句型，是 u 为（简单）短语必须满足的两个条件，试以下例加以说明。

设有文法 G2.3[<标识符>]：

<标识符>::=<字母>|<标识符><字母>|<标识符><数字>
<字母>::=a|b|c <数字>::=1|2|3|4

符号串<字母><字母><字母>显然是文法 G2.3 的句型。有规则<标识符>::=<字母>，所以，

<字母><标识符><字母>=><字母><字母><字母>

但<字母><标识符><字母>并不是文法 G2.3 的句型，因此第二个<字母>不是句型<字母><字母><字母>中的简单短语。事实上，左边第一个<字母>才是简单短语：

<标识符>=>* <标识符><字母><字母>=><字母><字母><字母>

如果不考虑归约后所得新符号串仍是句型，则任何句型中的某个子符号串，只要与某个规则的右部符号串相同时，该子符号串将是该句型的简单短语，这显然是不正确的。

除了仅由识别符号组成的句型外，一个文法的任何句型中都必定包含一个简单短语或短语，且可能包含不止一个的简单短语或短语。例如，文法 G2.3[<标识符>]的句型 a<字母>c 中包含简单短语 a 与 c，包含短语 a、a<字母>与 c。

简单短语：

<标识符>=>* <字母><字母>c，<标识符>=>* a<字母>c，且<字母>::=a

<标识符>=>* a<字母><字母>，<标识符>=>* a<字母>c，且<字母>::=c

短语：

<标识符>=>* <标识符>c，

　　<标识符>=>* a<字母>c，　且<标识符>=>+ a<字母>　（长度为 3）

<标识符>=>* <标识符><字母>c，

　　<标识符>=>* a<字母>c，　且<标识符>=>+ a　　（长度为 2）

<标识符>=>* a<字母><字母>，

　　<标识符>=>* a<字母>c，　且<字母>=>+ c　　（长度为 1）

一个句型中的最左简单短语称为该句型的**句柄**。例如，句型 a<字母>c 中的句柄是 a。任何一个句型的句柄总是存在且唯一的。例外是仅由识别符号组成的句型。

简单短语、短语与句柄的概念，在分析技术中有重要的作用。

一个有限多个规则的文法，为何能生成无限多个的句子？如同应用文法 G2.2[<变量说明>]时所见，在对<变量名>进行替换时，每应用规则 6 一次，变量名中就将多增加一个字母，应用任意多次，便可生成任意长的变量名，从而生成任意的变量说明。规则 6 形式的规则称为递归定义的规则，简称递归规则。

关于递归，有规则递归与文法递归两大类。一般概念如下。

① **规则递归**。若规则形如 U::=U…，则称文法关于非终结符号 U 规则左递归；若规则形如 U::=…U，则称文法关于非终结符号 U 规则右递归；若规则形如 U::=…U…，则称文法关于非终结符号 U 规则（一般）递归。

② **文法递归**。文法关于非终结符号 U，若存 在 U=>+ U…，则称文法关于非终结符号 U 文法左递归；若存在 U=>+ …U，则称文法关于非终结符号 U 文法右递归；若存在 U=>+ …U…，则称文法关于非终结符号 U 文法（一般）递归。

重点关注的是左递归。不论是规则左递归，还是文法左递归，左递归对分析技术的应用将产生重大影响。今后将讨论左递归的消去问题。

4. 句子生成的计算机实现

如前所述，文法的句子是从识别符号出发进行推导而生成的。推导是一个直接推导序列。鼓励采用系统的方式进行推导，为此引进最左推导与最右推导的概念。

一个推导，其中的每一步直接推导，总是把相应句型中最左的非终结符号，替换为以其为左部的重写规则的右部符号串，称为**最左推导**。

一个推导，其中的每一步直接推导，总是把相应句型中最右的非终结符号，替换为以其为左部的重写规则的右部符号串，称为**最右推导**。

这里考虑如何利用计算机实现应用推导生成文法的句子。

【例 2.12】 设文法 G2.4[E]：

```
E::=E+T|E-T|T   T::=T*F|T/F|F   F::=(E)|i
```

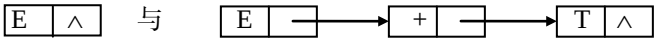
试以句子 i+i/i 的生成为例讨论实现问题。

不难构造如下的推导（最左推导）：

```
E=>E+T=>T+T=>F+T=>i+T=>i+T/F=>i+F/F=>i+i/F=>i+i/i
```

最终所推导得到的终结符号串便是句子。

首先考虑相关的数据结构。推导中每一个句型都是一个符号串，但其长度可变，宜用链表结构，例如，对于 E 与 E+T 等引进如下所示的链表。



链表中每个结点是符号结点，由两部分组成，即，

文法符号	后继符号结点指针

为了把一个直接推导所涉及的两个句型相联，引进另一类结点，即，链接结点，形如：

句型首符号结点指针	下一句型链头指针

因此，对于上述推导可有如图 2-1 所示的链表结构。

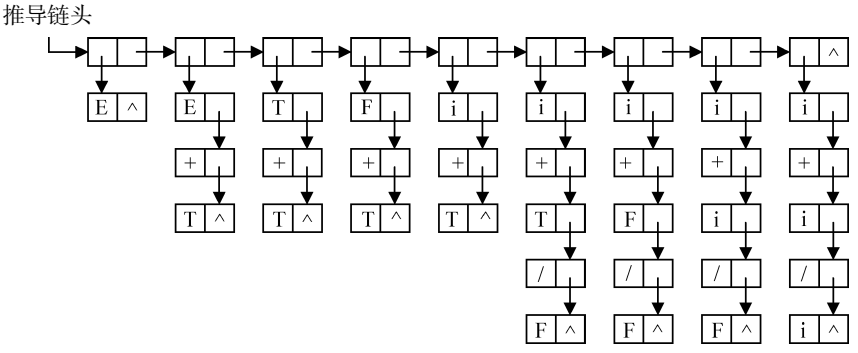


图 2-1

显然图 2-1 中的每一“列”（垂直链），各结点相应的符号组成一个句型。这样的链表结构十分容易用 C 语言实现，例如上述两类结点可用 C 语言结构类型定义如下：

```
typedef struct 符号结点
{ symbol  文法符号;
  struct  符号结点 *后继符号结点指针;
} 符号结点类型;
```

其中，symbol 是符号类型，若一个符号由多个字符组成时，可定义为：

```
typedef char symbol[MaxLength];
```

若文法符号仅由单个字符组成时，可简化为 char 类型，即，定义为：

```
typedef char symbol;
```

为了简化起见，用序号代替符号本身，即非终结符号用在 V_N 中的序号代替，而终结符号用在 V_T 中的序号代替。由于两类符号都从 1 开始，为在规则右部中能区别开，可把非终结符号的序号加上易识别的一个数值，如 100。例如，V_N={E,T,F},E 的序号为 1+100=101，T 的序号为 2+100=102。而 V_T={+,-,*,/,(,),i}，+的序号为 1，-的序号为 2 等。这样，符号结点的数据结构可定义如下：

```
typedef struct 符号结点
{  int    文法符号序号;
```

```
    struct  符号结点 *后继符号结点指针;
} 符号结点类型;
```

而链接结点可采用结构类型定义如下：

```
typedef struct 链接结点
{  符号结点类型  *句型首符号结点指针;
   struct 链接结点 *下一句型链头指针;
}  链接结点类型;
```

整个推导只需引进变量“推导链头”head 如下：

```
    链接结点类型   *head;
```

由 head 指向推导链的首结点。

当显示输出推导时，只需依次把各“列”结点中的符号依次输出，在各列符号串之间输出箭头“=>”，便得到推导。

现在讨论如何建立一个直接推导。由于推导可按最左推导、最右推导与一般推导 3 种方式建立，先考虑最左推导的情况。

假定推导构造过程中已产生某一个句型，即已构造相应的一列结点，这时只需从相应的句型首符号结点指针开始，向下顺次查看各个结点上的文法符号，找到的第一个非终结符号就是最左的非终结符号，按此找到以它为左部的各个规则，用其中一个规则的右部符号串构造一个结点链，用此链代替该非终结符号结点，便得到直接推导所得的新符号串，从而得到一“列”新句型。

由于一个文法的非终结符号可能是若干个重写规则的左部，选择哪一个规则才能生成所给定的句子，在讨论分析技术之前只能在人的干预下确定，因此必须采用交互方式实现。

从建立识别符号结点开始，逐步建立推导链的过程用流程图表示如图 2-2 所示。

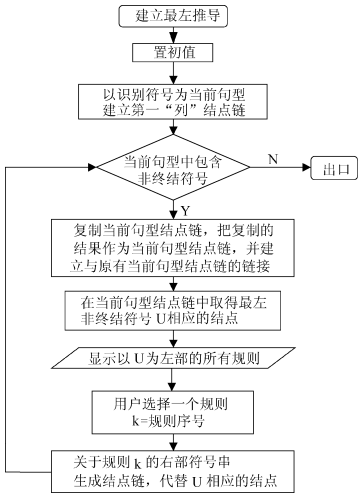


图 2-2

相应的 C 语言伪代码程序可写出如下：

```
/* 建立最左推导 */
链接结点类型 *建立最左推导(  )
{  置初值;
   以识别符号为当前句型，建立第一"列"结点链;
   /*实现如下:
```

```
    head=(链接结点类型*)malloc(sizeof(链接结点类型));
    当前句型链指针=head;
    当前句型首结点指针=(符号结点类型*)malloc(sizeof(符号结点类型));
    当前句型首结点指针→文法符号序号=识别符号序号+100;
    当前句型首结点指针→后继符号结点指针=NULL;
    head→句型首符号结点指针=当前句型首结点指针;
  */
  while(当前句型首结点指针指向的当前句型中包含非终结符号)
  {  old当前句型链指针=当前句型链指针;
     当前句型链指针=复制当前句型链(当前句型链指针);
     old当前句型链指针→下一句型链头指针=当前句型链指针;
     p=当前句型链指针→句型首符号结点指针;
     U前位置指针=NULL;
     while(p && p→文法符号序号<100)
     {  U前位置指针=p;
        p=p→后继符号结点指针;
     }
     U=VN[p→文法符号序号-100];
     找出以U为左部的一切重写规则, 显示规则及其序号;
     printf("选择一个规则序号:");
     scanf("%d", &k);
     关于规则k的右部符号串生成结点链, 链首指针为first, 链尾指针为last;
     把上述结点链链入U位置指针指向的结点处;
     /* 实现如下:
        last→后继符号结点指针=p→后继符号结点指针;
        if(U前位置指针!=NULL)
            U前位置指针→句型首符号结点指针=first;
        else
             当前句型链指针→后继符号结点指针=first;
     */
  }
  return head;
}
```

其中，判别当前句型中是否包含非终结符号的C型语言伪代码程序（函数定义）可设计如下：

```
int判当前句型包含非终结符号(符号结点类型 *当前句型首结点指针)
{  q=当前句型首结点指针;
   while(q && q→文法符号序号<100)
     q=q→后继符号结点指针;
   return (q!=NULL);
}
```

其他需实现的是复制当前句型与为规则右部符号串生成结点链。

复制当前句型的C型语言伪代码程序（函数定义）可设计如下：

```
链接结点类型 *复制当前句型链(链接结点类型 *老句型链指针)
{  oldLinkP=老句型链指针;
   oldNodeP=老句型链指针→句型首符号结点指针;
```

```
    newLinkP=(链接结点类型 *)malloc(sizeof(链接结点类型));
    newNodeP=(符号结点类型 *)malloc(sizeof(符号结点类型));
    newLinkP→句型首符号结点指针=newNodeP;
    newNodeP→文法符号序号=oldNodeP→文法符号序号;
    oldNodeP=oldNodeP→后继符号结点指针;
    while(oldNodeP)
    {  p=(符号结点类型 *)malloc(sizeof(符号结点类型));
       p→文法符号序号=oldNodeP→文法符号序号;
       newNodeP→后继符号结点指针=p;
       newNodeP=p;
       oldNodeP=oldNodeP→后继符号结点指针;
    }
    newNodeP→后继符号结点指针=NULL;
    return  newLinkP;
}
```

显然，实现的要点是文法的机内存储表示。一种简单的存储表示是采用数组结构存储文法，数组中每个元素对应于一个重写规则，可设计如下：

```
typedef struct
{  symbol  左部非终结符号;
   symbol  右部符号串[MaxLength];
   int     右部长度;
}  重写规则类型;
```

由于文法符号可以是多个符号，如同前面那样，简洁地用序号代替符号，即当符号 $T \in V_N$ 时，用 T 在 V_N 中的序号（加上易识别的数值，如 100）代替，而 $T \in V_T$ 时，用 T 在 V_T 中的序号代替。因此，重写规则类型可修改设计如下：

```
typedef struct
{  int  左部符号序号;
   int  右部符号串[MaxLength];
   int  右部长度;
}  重写规则类型;
```

从而文法可定义为如下的数据结构：

```
    重写规则类型  文法[MaxRuleNum];
```

例如，文法 G2.4[E]：

```
    E::=E+T|E-T|T   T::=T*F|T/F|F    F::=(E)|i
```

可以用以下方式通过 C 语言赋初值方式存放在计算机内：

```
重写规则类型  文法 G2.4[MaxRuleNum]=
{ {101,{101,1,102},3}, {101,{101,2,102},3}, {101,{102},1},
  {102,{102,3,103},3}, {102,{102，4,103},3}, {102,{103},1},
  {103,{5，101,6}, 3},  {103，{7}，1}
};
```

鉴于 C 语言数组元素的下标从 0 开始，较为合适的做法是避免使用 0 号元素，可改写如下：

```
重写规则类型  文法 G2.4[MaxRuleNum]=
{ {0},
```

```
  {101,{0,101,1,102},3}, {101,{0,101,2,102},3}, {101,{0,102},1},
  {102,{0,102,3,103},3}, {102,{0,102,4,103},3}, {102,{0,103},1},
  {103,{0,5,101,6}, 3},  {103,{0,7},1}
};
```

显然，显示输出以按其展开的非终结符号作为左部的规则很容易实现。同样，为相应右部符号串建立符号结点链也很容易实现，这时与复制句型的区别仅在于：不再从符号结点链出发构造新的符号结点链，而是从规则右部符号串（数组）出发构造新的符号结点链。

2.2.2 语言的概念

一个程序设计语言一般被看作是用此语言写出的一切程序的集合。当把程序设计语言看作一般的字母表上的符号组成的符号串集合时，一个程序是相应程序设计语言的句子，而一个程序设计语言是相应文法的一切句子的集合。设有文法 G[Z]，则由该文法描述的语言表示为 L(G[Z])。因此，一般地引进语言的概念如下：

$L(G[Z])=\{x \mid Z=>^* x, \text{且} x\in V_T^+\}$

其中 Z=>* x，且 $x\in V_T^+$，即 x 是文法 G[Z]的句子，因此，**语言**是相应文法的一切句子组成的集合。不能由识别符号推导得到的终结符号串不是语言的句子，对于程序设计语言也就不是程序设计语言的程序。

一个文法所能产生的一切句子组成一个语言，因此不对文法的句子与语言的句子严加区别。

语言可以是有穷的，也可能是无穷的，即语言中包含无限多个句子。从前面的讨论知道，当文法关于某非终结符号递归时，将产生无限多个句子。一个程序设计语言可以用来写出无穷多个的程序，也就是因为相应文法存在递归性，即规则递归与文法递归。例如，对于 C 语言相应文法中，必然包含下列规则：

<语句>::=<条件语句>

<条件语句>::=if(<表达式>)<语句>else<语句>

以及

<表达式>::=<表达式><加法运算符><项>

<项>::=<项><乘法运算符><因子>

<因子>::=(<表达式>)

如果一个语言是无穷的，则描述（生成）该语言的文法必定是递归定义的。

不论文法是递归定义的，还是非递归定义的，给定一个文法，也就唯一地确定了由它描述的语言；但若给定的是语言，则可用不同的文法来产生该语言的全部句子。

【例 2.13】 设有文法 G2.5 [E]：

```
E::=T+E|T-E|T    T::=F*T|F/T|F    F::=(E)|i
```

则该文法 G2.5[E]与文法 G2.4[E]所确定的语言是相同的。

【例 2.14】 设有语言 $L=\{ab^i|i\geqslant 0\}$，可把 L 看成是标识符的集合，a 为字母，b 为字母或数字。可为它构造文法 G2.6[A]：

```
A::=a|Ab
```

也可为它构造文法 G2.6′[A]：

```
A::=a|aB   B::=Bb|b
```

文法 G2.6[A]与 G2.6′[A]确定相同的语言 L。

若两个不同的文法确定的语言是相同的，则称这两个文法是**等价的文法**。

当给定一个语言，可为它构造若干个不同的、但都是等价的文法，它们都能产生该语言，那么，其中的哪一个文法更能贴切地描述该语言呢？这里所谓的贴切又按什么标准？

为此，可以对文法与语言分类，当某文法的文法类与相应的语言类相一致时，便可认为是最贴切的，这里引进 Chomsky 文法类与语言类。

2.2.3　文法与语言的分类

1．Chomsky 文法类与语言类

首先介绍 Chomsky 文法的概念。如前所述，一个文法有 4 要素，即非终结符号集、终结符号集、重写规则集与识别符号，可以把这 4 要素抽象成所谓的 Chomsky 文法。

Chomsky 文法是一个四元组(V_N,V_T,P,Z)，其中，V_N 是非终结符号集，V_T 是终结符号集，P 是重写规则集，而 Z 是识别符号。

显然，$V_N \cap V_T = \varnothing$。V 表示字汇表，即 $V = V_N \cup V_T$。

下面按重写规则的定义形式引进 4 类 Chomsky 文法。

（1）(Chomsky)**0 型文法**

文法中 P 的每个重写规则都具有下列形式：

u::=v

其中，$u \in V^+$，$v \in V^*$。(Chomsky)0 型文法又称**短语结构文法**，缩写为 PSG。

（Chomsky）0 型文法或短语结构文法的相应语言称为 **0 型语言**或**短语结构语言**。

当从识别符号出发生成句子而应用规则 u::=v 时，可在当前句型中把子符号串 u 重写（替换）为符号串 v。由于|v|≥0，v 可能为空串ε，而 $u \in V^+$，必定|u|≥1，即规则左部决不可能是空串，但可能包含不止一个的符号，换言之，终结符号也可能出现在规则左部。

（2）(Chomsky)**1 型文法**

文法中 P 的每个重写规则都具有下列形式：

xUy::=xuy

其中，$x,y \in V^*$,$U \in V_N$,$u \in V^+$。(Chomsky)1 型文法又称**上下文有关文法**，缩写为 CSG。

(Chomsky)1 型文法或上下文有关文法的相应语言称为 **1 型语言**或**上下文有关语言**。

当关于 1 型文法，从识别符号出发生成句子而应用规则 xUy::=xuy 时，仅当 U 处于上下文 x 与 y 之中时，在当前句型中可把 U 重写（替换）为符号串 u，因此说，该类文法是上下文有关的。

C 语言中上下文有关的重写规则例子可以有：

<函数名>‘(’<实参表列>‘)’::=<标识符>‘(’<实参表列>‘)’

与

<数组名>‘[’<下标表达式>‘]’::=<标识符>‘[’<下标表达式>‘]’

可见标识符仅在特定的上下文中才被直接归约成函数名或数组名等。

注意，其中的小括号与中括号都用撇号括住，这是因为在这里，它们是可以出现在程序中的符号，并不是元语言符号。

（3）(Chomsky)**2 型文法**

文法中 P 的每个重写规则都具有下列形式：

U::=u

其中，$U \in V_N$，$u \in V^+$。(Chomsky)2 型文法又称**上下文无关文法**，缩写为 CFG。

(Chomsky)2 型文法或上下文无关文法的相应语言称为 **2 型语言**或**上下文无关语言**。

(Chomsky)2 型文法与 1 型文法的显著区别在于：当从识别符号出发生成句子而应用规则 U::=u 时，不论上下文是什么，都可把当前句型中的非终结符号 U 重写（替换）为符号串 u，因此该类文法是上下文无关的。先前所讨论的文法事实上都是上下文无关文法。

请注意，规则右部符号串 $u\in V^+$，而不是 $u\in V^*$。这就是说，u 不能是空串。然而在某些情况下，可能引进规则ε，即规则右部为空串的规则。例如，

<如果语句>::=if(<表达式>)<语句><else 部分>

<else 部分>::=else<语句>

<else 部分>::=ε

其中，第三个规则是ε规则，$\varepsilon\in V^*$，这样，$u\in V^*$。由于在一个文法中增加或删除ε规则所产生的语言与原有文法产生的语言最多可能相差一个ε句子，即空串ε，而对一个非 0 型语言，往其中添加或从其中删去ε句子，并不改变其语言类，因此往往把 2 型文法规则 U::=u 扩展为 $u\in V^*$。当进行文法的等价变换时，往往使 $u\in V^+$扩展为 $u\in V^*$。例如上述<如果语句>的规则可看成是对

<如果语句>::=if(<表达式>)<语句>else<语句>

与

<如果语句>::= if(<表达式>)<语句>

进行等价变换的结果。

如上所述，C 语言应采用上下文有关文法来描述，但事实上往往采用上下文无关文法来描述，而与上下文有关的信息通过其他手段来表达，例如语义处理等。

（4）(Chomsky)**3 型文法**

文法中 P 的每个重写规则都具有下列形式：

U::=T 或 U::=WT

其中，$T\in V_T$，U、$W\in V_N$。3 型文法又称**正则文法**，缩写为 RG。

(Chomsky)3 型文法或正则文法的相应语言称为(Chomsky)**3 型语言或正则语言**。

当关于 3 型文法，从识别符号出发生成句子时，单个非终结符号只能重写（替换）为单个终结符号或单个非终结符号后跟以单个终结符号。当是后一种情况时非终结符号总是在终结符号的左边，因此，这种形式的正则文法可称为左线性文法。有时，正则文法的规则形式如下：

U::=T 或 U::=TW

其中，非终结符号 W 在终结符号 T 的右边。这种形式的正则文法可以称为右线性文法。本书采用左线性文法形式的正则文法。要注意的是，如果既有形如 U::=WT 的规则，又有形如 U::=TW 的规则，则这样的文法既不是左线性的，也不是右线性的，即不是正则文法，事实上是 2 型文法或上下文无关文法了。

从上面的讨论可知，Chomsky 文法是按照对规则的定义形式逐步加限制而得到，因此，一个 3 型文法可看成是 2 型文法，也可看成是 1 型文法，甚至可看成是 0 型文法，2 型文法可看成是 1 型文法，也可看成是 0 型文法；反之，0 型文法有一些可以是 1 型文法，有一些不能是 1 型文法，更不是 2 型文法，特别是有一些 2 型文法，可以是 3 型文法，而有一些不能是 3 型文法。

本书重点讨论上下文无关文法与正则文法。

2. Chomsky 文法类与程序设计语言

讨论 Chomsky 文法分类法的目的，是把它应用于程序设计语言及其编译实现，以便采用最合适的分析技术，提高实现效率。

如前所述，C 语言本质上应采用上下文有关文法来描述。例如，标识符还可作为标号，但不是任何标识符都是标号，只在特定上下文中才是标号，因此引进下列规则是合适的：

<标号>:<语句>::=<标识符>:<语句>

goto <标号>::=goto <标识符>

但对于上下文有关文法类，除了其上下文无关的子集外，尚无成熟的高效的分析技术可以应用，通常以上下文无关文法描述程序设计语言。

如前所述，程序是基本符号集上的符号按一定规则构造的符号串，由符号组成各类语法成分。符号由词法分析阶段识别，符号的拼写规则事实上可由正则文法来刻划。例如，标识符与无正负号整数可用下列形式的规则定义：

<标识符>::=字母|<标识符>字母 | <标识符>数字

其中的字母与数字分别代表具体的字母与数字。显然这是正则文法规则的形式。对于 C 语言运算符!=可定义如下：

<!=运算符>::=<惊叹号>=

<惊叹号>::=!

即使对于关键字，它们由字母组成，也可按正则文法规则形式定义。例如，对于关键字 else 有：

```
<else_1>::=e
<else_2>::=<else_1>l
<else_3>::=<else_2>s
<else>::=<else_3>e
```

显然，<else>=>+ else。

概括地说，词法分析是基于正则文法进行的。采用关于正则文法的分析技术，可以高效地实现词法分析，进一步可以实现词法分析程序的自动生成。

一般的语法成分则采用上下文无关文法描述，例如对于 C 语言程序可引进如下的重写规则：

<C 程序>::=<全局变量说明><函数定义序列>

<函数定义序列>::=<函数定义序列><函数定义> | <函数定义>

这里作了简化，把一切全局变量说明全部集中在 C 程序的最前面，至于必须要有一个且仅一个 main 函数定义，将由语义来规定。

概括起来，语法分析是基于上下文无关文法进行的，采用关于上下文无关文法的分析技术，可以高效地实现语法分析，尤其可以实现语法分析程序的自动生成。

综上所述，与词法有关的规则用正则文法描述，与局部语法有关的规则用上下文无关文法描述，而与全局语法有关的部分则用上下文有关文法来描述。但为了高效地实现，通常对与语法有关的部分用上下文无关文法来描述。因此编译实现时，与正则文法相关的词法分析部分采用正则分析技术，而语法分析部分采用上下文无关分析技术。本书后面将基于正则文法讨论词法分析问题，基于上下文无关文法讨论语法分析问题。

2.3 句 型 分 析

2.3.1 句型分析与语法分析树

1. 语法分析树

对于某个文法，当为一个输入符号串构造出一个推导时，表明这个符号串是该文法的一个句子。是否能为一个给定的输入符号串构造出推导，这就取决于它是否该文法的句子。对于一个给定的程序设计语言，判别某个输入符号串是否是该程序设计语言的程序，即语法上无错误，从编译实现的角度，也就是判别它是否描述该程序设计语言的文法的句子。

对于某个文法，识别一个输入符号串是否其句子的过程称为**句型分析**。对于程序设计语言，句型分析，也就是识别一个输入符号串是否是该程序设计语言的无语法错误的程序。

为了有利于句型分析，引进一种辅助工具——**语法分析树**，简称**语法树**。

语法分析树借用英语等课程中帮助理解句子结构的语法分解图。这种语法分解图中把句子分解成各个组成部分，以便分析句子的语法结构，从而帮助理解句子的结构。例如对句子“Nanjing is a beautiful city”，可有语法分解图如图 2-3 所示。

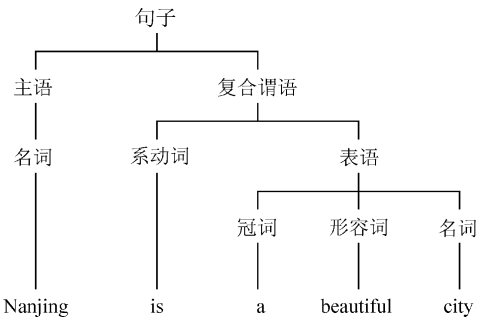


图 2-3

从该语法分解图可明显看出：Nanjing 是名词作主语，is 是系动词，与后面的表语构成复合谓语，这里的表语是 a beautiful city。可以引进相应的文法规则如下：

<句子>::=<主语><复合谓语>
<主语>::=<名词>
<名词>::=Nanjing | Hangzhou | Beijing | city
<复合谓语>::=<系动词><表语>
<系动词>::=is | was
<表语>::=<冠词><形容词><名词>
<冠词>::=a | an | the
<形容词>::=beautiful | great | wonderful

对照图 2-3，这种图解表示与所定义的文法规则完全一致，但可以给人直观而完整的印象，是理解语法结构的极好工具。现在沿用语法分解图到编译实现，称为语法分析树，一般画成如图 2-4 所示，为了简化，可以不画出小圆。

从图形上看，语法分析树像一棵倒过来的树，即根在上，叶在下。

下面以简化的算术表达式为例讨论语法分析树的构造。

设有文法 G2.7[E]：

```
E::=E+T|T   T::=T*F|F   F::=(E)|i
```

对于输入符号串 i+i*i 可有下列推导：

```
E=>E+T=>T+T=>F+T=>i+T=>i+T*F=>i+F*F=>i+i*F=>i+i*i
```

可画出相应的语法分析树如图 2-5 所示。为简化起见，该图中没有为结点画出小圆，事实上往往不画小圆。

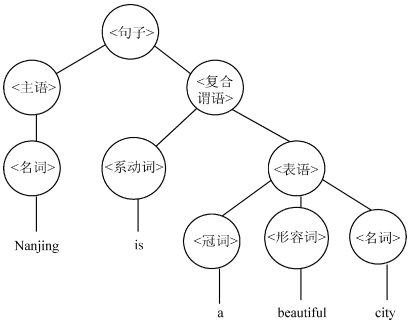


图 2-4

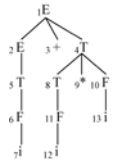


图 2-5

语法分析树的结构详细介绍如下。

首先，语法分析树由结点和边组成。结点用小圆表示，写在小圆内的是结点所对应的文法符号，一个结点对应一个文法符号，例如，E 与+；两个结点之间的连线称为边，边反映语法概念之间的上下层关系和同层关系。

其次，关于语法分析树有如下几个概念。

① 根结点：指的是没有从上向下进入它的边，而只有从它向下发出的边的结点。根结点是唯一的，一般总是对应于识别符号。

② 分支：从某结点向下发出的边，连同边上的结点构成分支。分支的名字是发出该分支的结点所相应的文法符号。分支的各个结点称为分支结点。分支结点所相应的文法符号，按从左到右顺序而形成的符号串称为**分支结点符号串**。分支名字结点是各分支结点的父结点，而分支结点是分支名字结点的子结点。同一分支上的结点彼此是兄弟结点。一般约定：最左的结点最“年长”，最右边的最“年轻”。如图 2-5 所示，编号为 4 的结点 T（称结点 4）向下发出的三个边及边上的结点（编号分别为 8、9 与 10）构成一个分支，结点 4 是这 3 个结点的父结点，这 3 个结点是结点 4 的子结点，编号为 8 的结点最“年长”，编号为 10 的结点最“年轻”。其他分支有：从结点 1 向下发出的 3 个边连同边上的结点（编号分别为 2、3 与 4）构成的分支，结点 6 向下发出的边及边上的结点 7 构成的分支等。

③ 末端分支：这是指语法分析树中所有分支结点都是末端结点的分支。末端结点是没有向下发出的边的结点，例如，编号为 7、12 与 13 的结点都是末端结点，因此，编号为 6、11 与 10 向下发出的分支都是末端分支。注意：末端结点上的文法符号不一定是终结符号，非终结符号也可能是末端结点符号。例如，图 2-6 中所示的语法分析树便是如此。

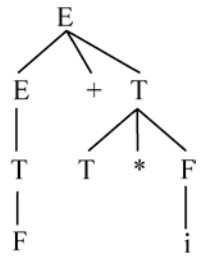


图 2-6

④ 子树：语法分析树的某个结点，连同从它向下发出的部分（如果有）称为该语法分析树的子树，该结点称为子树的根结点。例如，图 2-5 中编号为 4、8、9、10、11、12 与 13 的结点及相关的边构成一个子树，子树根结点是结点 4。

一个语法分析树的最大子树就是它本身。

⑤ 末端结点符号串：一个语法分析树（或其子树）的一切末端结点所对应的文法符号，从左到右地排列时构成的符号串称为该语法分析树（或其子树）的**末端结点符号串**。今后学习语法分

析树的构造方法时不难看出，语法分析树的末端结点符号串一定是相应文法的句型，当末端结点符号串全由终结符号组成时便是句子。

2. 从推导构造语法分析树

【例 2.15】 从推导构造语法分析树的例子。

今以文法 G2. 7[E]:

```
E::=E+T|T   T::=T*F|F   F::=(E)|i
```

关于输入符号串 i+i*i 的推导：

```
E=>E+T=>T+T=>F+T=>i+T=>i+T*F=>i+F*F=>i+i*F=>i+i*i
```

为例，构造语法分析树，其构造过程如下：

步骤 1　以识别符号 E 为根结点，由第一个直接推导 E=>E+T，把 E 替换为 E+T，因此以 E 为分支名字结点，从它向下画一个分支，即 3 条边连同 3 个分支结点 E、+与 T。

步骤 2　对应于下一个直接推导 E+T=>T+T，把 E 替换为 T，以 E 为分支名字结点，从它向下画一个分支，这时是一个边连同边上的结点 T。

步骤 3　类似地，每次对应于一个直接推导，从被替换的非终结符号所对应的结点出发向下画分支，相应的分支结点符号串正是替换该非终结符号的符号串。如此重复步骤 3，画出相应于直接推导的分支，直到推导结束。最后得到的语法分析树即为所求。

概括起来，从推导构造语法分析树的过程如下：以识别符号为根结点，从它开始关于每一个直接推导画一个分支，这一分支的名字是直接推导左边被替换的非终结符号，而分支的分支结点符号串是直接推导右边所替换的符号串。

从上述构造过程可知，对于每个推导必定存在一个语法分析树，也可看出，分支结点符号串是相应句型中相对于分支名字结点符号的简单短语。例如，当构造到编号为 4 的结点 T 作为分支名字结点时，相应分支是末端分支，相应句型是 i+T*F，分支结点符号串是 T*F，它是当前句型 i+T*F 中相对于 T（编号为 4 的结点符号）的简单短语：

```
E=>* i+T=>* i+T*F, 其中  T::=T*F
```

记住这点将可以方便地利用语法分析树在相应的句型中找出简单短语。

从构造过程也可看到，子树的末端结点符号串是相应句型中相对于子树根结点符号的短语。例如，以编号为 4 的结点作为子树根结点的子树，它的末端结点符号串 i*i 就是相应句型 i+i*i 中相对于子树根结点 T（结点编号为 4）的短语：

```
E=>* i+T=>* i+i*i ,    其中  T=>+ i*i
```

或者，在构造编号为 13 的结点 i 之前的语法分析树中，子树末端结点符号串 i*F 是相应句型 i+i*F 中相对于子树根结点 T（编号为 4 的结点符号）的短语：

```
E=>* i+T=>* i+i*F,    其中  T=>+ i*F
```

记住这点将可以方便地利用语法分析树在相应句型中找出短语。

由于语法分析树是关于某文法句型的推导而构造的，因此称为某文法句型之推导的语法分析树，通常简称为关于某文法句型的语法分析树。例如，G2.7[E]的句型 i+i*F 的语法分析树。

【例 2.16】 试为文法 G2.7[E]的句型 F+T*i 构造语法分析树，并从中找出一切简单短语和短语。

可为该句型构造推导如下：

```
E=>E+T=>E+T*F=>E+T*i=> T+T*i=>F+T*i
```

按照前述从推导构造语法分析树的步骤，首先以识别符号建立根结点，然后关于每个直接推

导依次建立分支，最后可得到所需的语法分析树如图 2-6 所示。找出其中的一切末端分支，即末端分支的末端结点符号串，这时有两个，因此有两个简单短语，即最左的 F 与 i。找出一切子树，即子树的末端结点符号串，这时有 4 个，因此，短语是 4 个，即最左的 F、T*i、i 以及整个符号串 F+T*i。

3. 从语法分析树构造推导

从推导构造语法分析树时，关于每个直接推导依次建立分支，不难想到，从语法分析树构造推导的过程是其逆过程，即不断剪去分支，同时建立直接推导的过程。还是以 i+i*i 的语法分析树为例进行说明。

假定已有如图 2-5 所示的语法分析树。首先找出最左的末端分支，其末端分支结点符号串为 i（编号为 7），进行剪支。这样得到如图 2-7 所示的语法分析树，其末端结点符号串是 F+i*i，构造最左的直接推导：

```
F+i*i=>i+i*i
```

从剪支后的语法分析树中再找出最左的末端分支，这时的末端分支的分支结点符号串是 F（编号为 6），剪去此末端分支，同时建立直接推导：

```
T+i*i=>F+i*i
```

因此有：

```
T+i*i=>F+i*i=>i+i*i
```

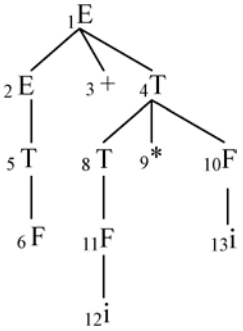


图 2-7

如此继续，每次对新得到的语法分析树，找出最左末端分支，在剪去该分支的同时，建立相应的直接推导，直到无分支可剪，构造推导的过程结束。这时可得到如下的推导：

```
E=>E+T=>E+T*F=>E+T*i=>E+F*i=>E+i*i=>T+i*i=>F+i*i=>i+i*i
```

显然，这是最右推导，它在构造过程中由每次剪去当前语法分析树的最左末端分支而生成。如果由此推导来建立语法分析树，将发现在构造语法分析树过程中，每步建立的分支是最右末端分支。如果在从语法分析树构造推导时，每步剪去的是最右末端分支，将得到前面例 2.14 所给的推导：

```
E=>E+T=>T+T=>F+T=>i+T=>i+T*F=>i+F*F=>i+i*F=>i+i*i
```

不言而喻，这是最左推导。

2.3.2　二义性

概括地说，如果给定一个语法分析树，按照不同的次序剪去分支，可为它构造若干不同的推导。当给定一个推导，则可构造唯一的语法分析树。现在问题是，当给定一个句子时，构造的语法分析树是否唯一？试看下例。

【例 2.17】 二义性的例子。

设有文法 G2.8[E]：

```
E::=E+E | E*E | (E) | i
```

对于其句子 i+i*i，可以有下列推导：

```
E=>E+E=>i+E=>i+E*E=>i+i*E=>i+i*i
```

但还可以为它构造下列推导：

```
E=>E*E=>E+E*E=>i+E*E=>i+i*E=>i+i*i
```

为这两个推导可分别构造语法分析树如图 2-8（a）与图 2-8（b）所示。

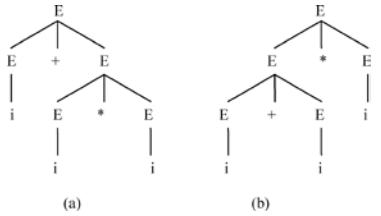


图 2-8

显然这是两个不同的语法分析树，这表明文法的一个句子，可能为它构造两个不同的语法分析树。分解不唯一，意味着含义解释的不唯一。如果三个 i 的值分别为 1、2 与 3，+是加法，*是乘法，则对于图 2-8（a），最终计算结果是 7，而对于图 2-8（b）， 最终计算结果将是 9。这种现象称为二义性。

如果对于某文法的同一个句子，存在两个不同的语法分析树，则称该句子是**二义性**的句子，包含二义性句子的文法称为**二义性文法**。

清楚的是，一个程序设计语言的文法必须是无二义性文法，否则用该程序设计语言所写的程序之运行效果将不唯一，而可能造成错误，一般都希望是无二义性的文法。如果是二义性文法，便应该设法消去此二义性。分析上述文法不难看出，产生二义性的原因是信息不足，既没有明确规定运算符+与*的优先级，也没有对相同优先级的运算符规定是左结合的还是右结合的。一般程序设计语言的文法较为复杂，难以保证其必定是无二义性的，便可能在语义说明中提供运算符的优先级与结合性等信息。合适的做法是对文法的规则作适当修改，让运算符的优先级与结合性等在规则中反映出来。事实上，文法 G2.7[E]:

```
E::=E+T|T   T::=T*F|F   F::=(E)|i
```

是无二义性的，也就是因为其中反映了运算符+与*的优先级及结合性。具体地说，T*F 的（直接）归约先于 E+T 的（直接）归约，也就是运算符*先于运算符+进行运算，简而言之，先归约先运算。左结合性也从图 2-9 可明显看出。

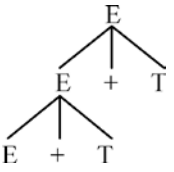


图 2-9

从上可见，一个语言，可以为它设计二义性的文法，也可以为它设计无二义性的文法。因此一般来说，讨论语言的二义性是无意义的，重要的是对同一个语言，为它构造无二义性文法。如果构造的是二义性文法，应设法在不改变语言的前提下，把二义性文法等价变换成无二义性文法。文法等价变换的概念将在后面讨论。注意，这里文法二义性的概念是语法范畴的概念，并不涉及含义。

2.3.3 分析技术及其分类

当给定某文法的一个句型的推导，可以为它构造相应的语法分析树，反之，给定一个语法分析树，也可以从它构造出相应的推导。语法分析树以其清晰的表示描述了句子的语法结构，清楚地反映出规则的应用，也显示了文法中各语法成分的层次，因此语法分析树是进行句子结构分析的有效工具。

问题在于：要能关于一个输入符号串画出语法分析树或得到推导，此输入符号串必须是文法的句子。然而，通常要构造出推导或语法分析树才能知道给定的输入符号串是否是文法的句子。对于一个长的符号串，这不是一件易事。因此必要的是引进一些分析技术，自动进行句型分析。

句型分析，就是识别一个输入符号串是否是某给定文法的句子的过程，这是推导或语法分析树的构造过程。即当给定一个输入符号串时，试图按照文法的规则，关于该输入符号串构造推导或语法分析树，从而识别出它是所给定文法的句子。

实现句型分析的程序称为**分析程序**，分析程序以输入符号串（即终结符号串）为输入，当识别出是文法的句子时给出推导或语法分析树作为输出，否则给出报错信息作为输出。分析程序进行句型的识别，所以又称**识别程序**。识别程序基于由分析技术确定的识别算法而编写。

分析技术分为两大类，即**自顶向下分析技术**与**自底向上分析技术**，这是因为语法分析树可以按从上向下与从下向上两种方式来构造。不论是哪一类分析技术，鉴于程序总是从左到右地书写与阅读，通常总是从左到右地识别输入符号串，即首先扫描并处理输入符号串最左边的第一个符号，然后从左到右逐个符号地分析其他各个符号。

这里简略说明两类分析技术的基本思想及各自要解决的问题。

1．自顶向下分析技术

自顶向下分析技术从上向下地逐步构造语法分析树，由于根结点在最上面，因此首先构造根结点。

自顶向下分析技术的基本思想是：从识别符号开始，试图从它推导出与输入符号串相同的终结符号串。因此，自顶向下识别过程是一个不断建立直接推导的过程。

从语法分析树看，自顶向下识别过程以识别符号为根结点，试图从它开始向下构造语法分析树，使得该语法分析树的末端结点符号串与输入符号串相同。如果是这样，则识别出输入符号串是文法的句子，如果所构造语法分析树的末端结点符号串不能与输入符号串相匹配，则表明该输入符号串不是相应文法的句子。

显然，采用自顶向下分析技术构造语法分析树的过程是一个不断向下构造分支的过程。

【例 2.18】　设有文法 G2.9[S]：

```
S::=V=E   V::=i   E::=F+F|F   F::=(E)|i
```

试采用自顶向下分析技术识别输入符号串 i=i+(i)是否是文法 G2.9 的句子。

以自顶向下分析技术为输入符号串构造语法分析树的步骤如下。

以识别符号 S 为根结点，向下构造语法分析树。首先构造直接推导 S=>V=E，且以 S 为分支名字结点，按规则 S::=V=E 向下构造分支，分支结点符号串为 V=E，如图 2-10 左边第二个语法分析树所示。这时语法分析树的末端结点符号串 V=E 是当前句型，在这当前句型中最左非终结符号是 V，相应地以结点 V 为分支名字结点，按规则 V::=i 构造直接推导：V=E => i=E，同时构造分支，分支结点符号串是 i。这时语法分析树的末端结点符号串是 i=E，也即是当前句型。如此继续，每步总是在当前句型 xUy 中对最左的非终结符号 U 找到规则 U::=u，构造直接推导：xUy => xuy，并以语法分析树中相应于该 U 的结点为分支名字结点，向下构造分支，分支结点符号串是 u。直到最后语法分析树的末端结点符号串与输入符号串 i=i+(i)相一致。构造语法分析树的全过程如图 2-10 所示。

可见输入符号串 i=i+(i)是文法 G2.9[S]的句子。这时得到推导：

```
S=>V=E=>i=E=>i=F+F=>i=i+F=>i=i+(E)=>i=i+(F)=>i=i+(i)
```

并得到如图 2-10（h）所示的语法分析树。

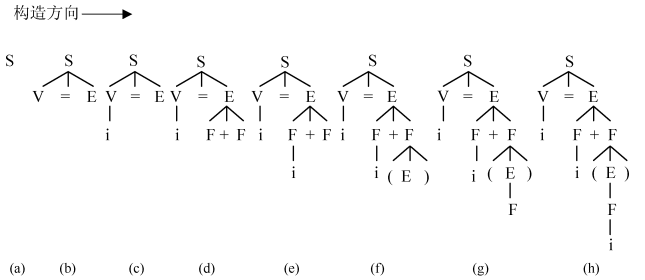


图 2-10

上述构造过程是一个不断进行直接推导或添加分支的过程，每步总是对相应句型中最左的非终结符号进行直接推导，或从相应结点向下构造分支。但一般来说进行句型分析可以只构造推导或只构造语法分析树。当构造语法分析树时，总是以当前所构造语法分析树的末端结点符号串中对应于最左非终结符号的结点为分支名字结点，向下构造分支。这对应于最左推导。

而对应于最右推导，构造语法分析树的过程显然是类似的。

现在的问题是：当对于某个非终结符号 U，存在若干个以它为左部的规则时，选择哪一个规则？例如，当已构造了图 2-10（d）所示的语法分析树时，关于 F 存在规则 F::=(E)与 F::=i，应该如何选择规则？显然关于左边的 F，选规则 F::=i，而关于右边的 F 必须选规则 F::=(E)。如果反过来，关于左边的 F 选 F::=(E)，而右边的 F 选 F::=i，则将不能为输入符号串 i=i+(i)构造出语法分析树，其末端结点符号串将与它不同。对于图 2-10（g）中所示语法分析树有类似情况。在这构造过程中，之所以能成功，关键在于人为的干预，由人选择了正确的重写规则。事实上，句型分析的目的，是自动机械地高效实现识别输入符号串是否是某文法的句子，因此可以把自顶向下分析技术要解决的基本问题归结如下。

在句型分析过程的某步时，当前句型 xUy 按其中的非终结符号 U 展开（直接推导）时，关于 U 存在若干个规则：

$$U::=u_1|u_2|\cdots|u_k \quad (k\geqslant 1)$$

如何确定规则 $U::=u_i(1\leqslant i\leqslant k)$?

2. 自底向上分析技术

自底向上分析技术试图从下向上地逐步构造语法分析树，由于根结点在最上面，因此首先构造末端结点符号串。

自底向上分析技术的基本思想是：从输入符号串开始，试图从它归约到识别符号，这是一个不断建立直接归约的过程。从语法分析树看，自底向上识别过程是从输入符号串构造相应的末端结点开始，试图从它们向上构造语法分析树，使得最终语法分析树的根结点正好是识别符号。如果是这样，则识别出输入符号串是文法的句子，如果不能构造出这样的语法分析树，则输入符号串不是文法的句子。

【例 2.19】 试按自底向上分析技术识别输入符号串 i=i+(i)是否文法 G2.9 [S]的句子。

以自底向上分析技术为输入符号串 i=i+(i)构造语法分析树的步骤如下。

以输入符号串 i=i+(i)作为末端结点符号串，构造相应的结点，如图 2-11（a）所示。如果是句子，则 i=i+(i)也是文法的句型，可以找出其中的最左简单短语（句柄）i，以相应的结点作为分支结点，按规则 V::=i 构造分支及分支名字结点 V，如图 2-11（b）所示。这时得到当前句型 V=i+(i)，同时可得到：V=i+(i)=>i=i+(i)。在此当前句型中找出最左简单短语 i，以相应的结点作为分支结点，

按规则 F::=i 构造分支及分支名字结点 F，如图 2-11（c）所示，这时得到当前句型 V=F+(i)，也可同时得到直接推导 V=F+(i)=>V=i+(i)。类似地，每步从当前句型中找出最左的简单短语 u，以相应的结点作为分支结点构造分支及分支名字结点 U，同时可得到一步直接推导。如此继续，直到当前句型仅为识别符号，相应结点正好是根结点，这时识别出输入符号串是文法的句子。整个句型分析过程如图 2-11 所示。

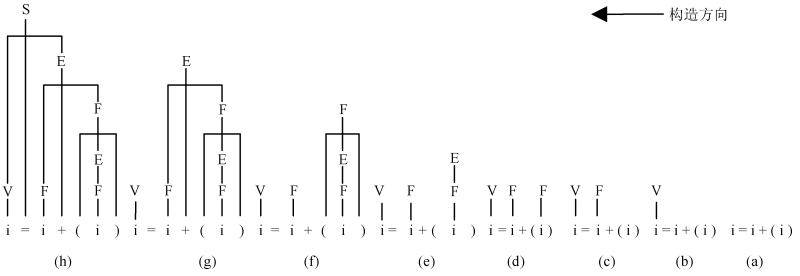


图 2-11

这时构造的归约是：

```
S=>V=E=>V=F+F=>V=F+(E)=>V=F+(F)=>V=F+(i)=V=i+(i)=>i=i+(i)
```

在此构造过程中，每一步找的是最左简单短语，由它作为分支结点符号串来构造分支与分支名字结点，这是一个最左归约过程。也可以找最右简单短语，将对应于最右归约。

现在的问题是：在某识别过程中，如果相应句型中的最左简单短语能归约到不同的非终结符号，则该归约到哪一个？例如，直接归约 i 时，可以有规则：V::=i 与 F::=i。对最左的 i 直接归约时，必须归约到 V，对于最左第二个 i 直接归约时，必须归约到 F。不然，便不能为输入符号串 i=i+(i)构造语法分析树，因此，概括起来，自底向上分析技术要解决的基本问题是：

按照自底向上分析技术识别句型时，若当前句型为 xuy，关于 u 存在若干个规则：U_1::=u，U_2::=u，…，U_k::=u，则如何确定 u？又把 u 按哪个规则 U_i::=u 直接归约($1 \leqslant i \leqslant k$)？

如果某种方法能用来帮助进行句型分析，但不能解决上述基本问题，则这种方法不能称为自底向上分析技术。作为分析技术，必须能同时解决上述两个基本问题。

请读者注意，关于输入符号串的构造，不论是推导，还是语法分析树，都不能反映出它们是以自顶向下方式，还是以自底向上方式构造的。要表明采用的是哪种分析技术，最简洁方式是用序号标明构造的先后顺序。

下面引进句型分析中的一个重要概念：规范分析。

在按自底向上分析技术句型分析，进行直接归约时，归约的总是当前句型中最左的简单短语。这时直接归约到某个非终结符号后，该非终结符号的右边总是不可能包含有其他任何非终结符号。这意味着输入符号串从左到右逐个符号地进行句型分析时，当左边的部分形成一个（最左的）简单短语时，便将归约，而还未形成简单短语时，便将继续向右扫描下一个符号，因此这种做法十分自然，与人的处理习惯很相符合。关于此引进规范分析的概念。

如果某直接推导 xUy=>xuy 中，y 不包含非终结符号，则称该直接推导是**规范**的。如果某推导 v=>+ w 中，每一步直接推导都是规范直接推导，则称该推导是**规范推导**，即 v 规范推导到 w，或者称是**规范归约**，即 w 规范归约到 v。

显然，规范推导是最右推导，规范归约是最左归约。不难发现，对于一个句子，它的最右推导正好与最左归约相同，而最左推导与最右归约相同。

注意，任何句子都有一个规范推导，但并非每个句型都有规范推导。例如，V=i+(F)是文法G2.9[S]的一个句型，对于它，显然不存在规范推导。

可由规范推导产生的句型称为**规范句型**。

例如，符号串 V=F+(i)是文法 G2.9[S]的规范句型。

不论是规范推导还是规范归约，统称为**规范分析**。

对于程序设计语言的编译实现来说，规范分析，尤其是规范归约，是一个十分重要的概念。当从左到右逐个符号地扫描输入符号串时，将发现简单短语而进行归约，然后继续扫描，这实际上就是边扫描边归约。这时的简单短语必定是最左的，即是句柄，它归约成的非终结符号的右边只能是输入符号，不可能有非终结符号，这就是最左归约，也就是规范归约。

可以说，凡是句子，关于它，必定存在规范分析（规范归约），这是自底向上分析技术可行的依据。

2.4 语法分析树的计算机生成

在前面 2.2.1 文法及其应用一节中讨论了文法句子的计算机生成，这实际上是推导的计算机生成，本节讨论如何由计算机生成语法分析树的问题。

为此，首先结合例子考察语法分析树应有怎样的数据结构，然后再考虑怎样构造。

【例 2.20】 设文法 G2.4[E]：

```
E::=E+T|E-T|T   T::=T*F|T/F|F   F::=(E)|i
```

试以输入符号串 i-i*i 为例，讨论语法分析树的生成。

如前所述，可为输入符号串 i+i*i 构造下列推导：

```
E=>E-T=>T-T=>F-T=>i-T=>i-T*F=>i-F*F=>i-i*F=>i-i*i
```

。

由此推导有相应的语法分析树如图 2-12 所示。

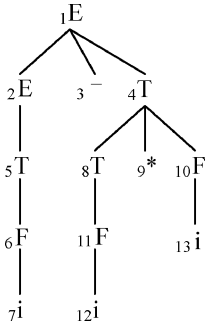


图 2-12

语法分析树由结点与边组成，每个结点对应于一个文法符号。为了确定语法分析树的结构，显然，只需确定每个结点在语法分析树中的位置，包括各个结点之间的相互关系。例如，确定了每个结点的父结点、左（紧邻）兄结点与（最）右子结点，也就确定了整个语法分析树。因此，语法分析树的结点的数据结构可设计如下：

```
typedef struct
{  int  结点序号;     int  文法符号序号;
   int  父结点序号;
   int  左兄结点序号; int 右子结点序号;
} 结点类型;
```

语法分析树的数据结构可设计如下:

```
结点类型  语法分析树[MaxNodeNum];
```

另外设置一个变量，记录语法分析树中结点的总数。

这样，对于图 2-12 中的语法分析树可如下表所示：

结点序号	文法符号	父结点序号	左兄结点序号	右子结点序号
1	E	0	0	4

2	E	1	0	5
3	-	1	2	0
4	T	1	3	10
5	T	2	0	6
6	F	5	0	7
⋮				

如果按照此表所提供各结点的信息来画出语法分析树，得到的结果将与图 2-12 的完全一样。

现在可以给出构造语法分析树的步骤如下：

步骤 1　以识别符号 Z 建立根结点，序号为 1，且以这仅包含一个非终结符号的句型 Z 作为当前句型。

步骤 2　从当前句型中找出最左的非终结符号 U，显示以 U 为左部的一切重写规则，根据所给的输入符号串，选择其中的一个规则 $U::=X_1X_2\cdots X_m$，以 U 为分支名字结点，以 $X_1X_2\cdots X_m$ 作为分支结点符号串，构造分支，建立父子兄弟结点关系。

步骤 3　重复步骤 2，直到当前句型中不再包含非终结符号，语法分析树构造结束，最终的语法分析树为所求。

显然，如果按照上述的步骤，很难知道当前的句型是什么，为了方便得到当前句型，从而找出最左非终结符号，也为了显示输出与语法分析树相应的推导，可以设置一个数组 D，用它来存放每一步的句型。假定在某步的当前句型是 $Y_1Y_2\cdots Y_{j-1}Y_jY_{j+1}\cdots Y_n$，存放在 D[k]中，其中最左的非终结符号是 Y_j=U，且选择的关于 U 的规则是：

$U::=X_1X_2\ldots X_m$

则在 D[k+1]中存放的将是下列句型：

$Y_1Y_2\ldots Y_{j-1}X_1X_2\ldots X_mY_{j+1}\cdots Y_n$

这时可如下地得到 D[k+1]，

① 把 D[k][0]～D[k][j-2]的内容 $Y_1Y_2\ldots Y_{j-1}$ 复制到 D[k+1][0]～D[k+1][j-2]处；

② 把 $X_1X_2\ldots X_m$ 复制到 D[k+1][j-1]～D[k+1][j-1+m-1]处；

③ 把 $Y_{j+1}\cdots Y_n$ 从 D[k][j]～D[k][n-1]复制到 D[k+1][j-1+m]～D[k+1][n+m-1]处。

下次在当前句型中找最左的非终结符号时，只需在 D[k+1]中，从 j=0 开始，把 j 逐次加 1，找出是非终结符号的 D[k+1][j]。

上述构造步骤可以修改如下：

步骤 1　以识别符号 Z 建立根结点，序号 N 为 1，且让 D[0]="Z"，k=0。以 D[k]作为当前句型。

步骤 2　从当前句型 $D[k]=Y_1Y_2\cdots Y_{j-1}Y_jY_{j+1}\cdots Y_n$ 中找出最左的非终结符号 Y_j=U，显示 以 U 为左部的一切重写规则，根据所给的输入符号串，选择其中的一个规则 $U::=X_1X_2\cdots X_m$，以 U 为分支名字结点，以 $X_1X_2\cdots X_m$ 作为分支结点符号串，构造分支，建立父子兄弟结点关系。同时，把新的句型 $Y_1Y_2\cdots Y_{j-1}X_1X_2\cdots X_mY_{j+1}\cdots Y_n$ 存放在 D[k+1]中，作为当前句型。

步骤 3　重复步骤 2，直到当前句型中不再包含非终结符号，语法分析树构造结束，最终的语法分析树为所求。

步骤 2 中构造分支结点，并建立父子兄弟结点关系，可以如下实现：

```
父结点序号=N;
```

```
左兄结点序号=0;
for(j=1; j=m; j++)  /* m 是规则右部长度 */
{ N=N+1;
   语法分析树[N].结点序号=N;
   语法分析树[N].文法符号序号=Xj 的序号;
   语法分析树[N].父结点序号=父结点序号;
   语法分析树[N].左兄结点序号=左兄结点序号;
   语法分析树[N].右子结点序号=0;
   左兄结点序号=N;
}
```

由于引进数组 D，输出推导是十分容易的事。建议读者把上述的构造步骤综合整理，得出完整的程序，上机实践语法分析树的计算机生成。

本章小结

本章讨论与编译程序构造相关的基础知识：文法与语言，以及句型分析。

两个基本问题：一是如何生成正确的程序；二是如何自动检查是正确的程序。围绕这两个方面引进相关的概念。

从符号串角度看程序，引进符号串与符号串集合，相关的重要概念是字母表的闭包与正闭包，读者必须熟练掌握闭包与正闭包的表示法；从文法角度看程序，程序是文法的句子，句子是从文法识别符号出发推导出来的终结符号串，因此引进重写规则、文法与语言，相关的重要概念包括：句子、句型、推导、归约、短语、简单短语、句柄。读者必须熟练掌握句子的生成，掌握最左推导与最右推导。文法由有限多个重写规则组成，但文法相应的语言一般是无限的，这是由于文法递归定义所引起的。递归有规则递归和文法递归。左递归影响到分析技术的应用，读者请注意左递归的问题。

一个文法的基本要素包括 4 方面，即非终结符号集 V_N、终结符号集 V_T、重写规则集 P 与识别符号 S，由此引进 Chomsky 文法(V_N，V_T，P，S)，并分为 4 类 Chomsky 文法类与语言类。一个语言，定义它的(Chomsky)文法类应与它的(Chomsky)语言类相一致，以便更好地应用相应的分析技术。与编译程序实现相关的主要是上下文无关文法与正则文法，它们分别是讨论语法分析和词法分析的基础，语义分析将在扩充的上下文无关文法基础上进行讨论。这两类文法必须熟练掌握。

句型分析是识别一个输入符号串是否某文法的句子的过程，句型分析的好工具是语法分析树。读者必须熟练掌握语法分析树的构造，了解语法分析树的各种用途。从语法分析树角度，句型分析技术分两大类，即自顶向下与自底向上分析技术，它们都有各自必须解决的基本问题。这里强调的是自动地、没有人为干预地进行句型分析，也就是说，作为一种分析技术，它必须要能解决相应的基本问题。相关的重要概念包括二义性与二义性文法、规范分析、规范句型。

本章讨论了句子生成的计算机实现，包括推导与语法分析树的实现。推导生成时的要点是结点链的建立，而语法分析树生成时的要点是结点的数据结构的设计，以及句型的存储表示，读者需熟练掌握。

复习思考题

1．为什么引进字母表的闭包与正闭包的概念？请试以 A={0，1，2}的 A^+与 A^*解释它们的直观含义。

2．为什么有限多个重写规则的文法能生成无限多个的句子？

3．引进 Chomsky 文法类与语言类有什么作用？

4．为什么 C 等高级程序设计语言必须用上下文无关文法描述？

5．语法分析树有哪些用途？请举例说明。

6．二义性对句型分析有什么影响？如何消去？

7．自顶向下分析技术与自底向上分析技术各自需解决什么基本问题？

习　　题

1．试以各种不同的表示方式给出 1/3 的一切近似值的集合表示。注意：不考虑含义，仅形式表示。

2．试给出{A，B，C}$^+$ {0，1，2，3}*中长度分别为 1、2、3、4 与 5 的元素。

3．试为下列文法写出 V_T与 V_N。

```
G[Q]：Q::=id              Q::=C
      id::=id L | id d | L  C::=C d | d
      L::= a | b | c        d::= 1 | 2| 3 | 4
```

4．设文法 G[E]：

```
E::=E+T|E-T|T   T::=T*F|T/F|F   F::=(E)|i
```

试分别为输入符号串 i/i、i*i-i 与 i*(i-i)+i 构造最左推导。

5．试关于题 4 中文法 G[E]，分别为输入符号串 i/i、i*i-i 与 i*(i-i)+i 构造最右推导。

6．试关于题 4 中文法 G[E]，在句型 E-T/F*F+i 中找出一切简单短语与短语。

7．设结点有如下结构：{整数值,下一结点指针}，要求建立 3 个结点的链，链头由指针 p 指向。试写出相应的 C 语言程序。

8．试为题 4 中的文法 G[E]写出 Chomsky 文法，指明这是哪类 Chomsky 文法。

9．试从下列推导

```
E=>E+T=>T+T=>F+T=>i+T=>i+T*F=>i+T/F*F
 =>i+F/F*F=>i+i/F*F=>i+i/i*F=>i+i/i*i
```

构造语法分析树。

10．设关于文法 G[S]：

```
S::=V=E   V::=i   E::=F+F|F   F::=(E)|i
```

有语法分析树如图 2-13 所示，试从它构造相应的推导。

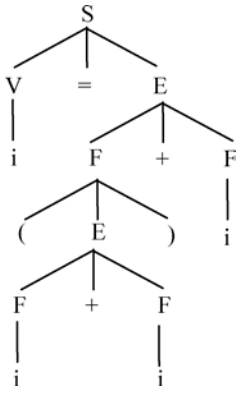


图 2-13

11．用 C 语言置初值方式给出题 10 中的文法 G[S]，其 V_N={ S, V, E, F }, V_T={ =, i, +, (,) }。要求：给出文法的数据结构定义。

第 3 章 词法分析

本 章 导 读

当从第 2 章了解了编译程序构造的基础知识后，可以对源程序进行翻译，其第一阶段是词法分析，词法分析由词法分析程序（或称扫描程序）完成。本章讨论词法分析，内容包括：概况、词法分析程序的手工实现，以及词法分析程序的自动生成。请读者重点关注词法分析程序的实现要点，包括属性字的设计与标识符的处理。词法分析程序由通用扫描算法和相关的表组成，自动生成，也就是表的自动生成。必要的是引进描述特定文法的符号之结构的手段，相应地引进正则表达式，进一步引进状态转换图和有穷状态自动机。请关注正则文法同正则表达式、状态转换图与有穷状态自动机相互之间的联系。

3.1 概　　况

编译程序的功能是把高级程序设计语言源程序翻译成等价的低级语言目标程序，而源程序本身是一个基本符号序列或字符序列，因此，首先必须进行词法分析，识别开各个符号（或单词）。编译程序中完成词法分析工作的部分称为词法分析程序，或称**扫描程序**。如前所述，词法分析程序的功能主要有 3 个：

- 扫描（读入）源程序字符序列；
- 识别开具有独立含义的最小语法单位：符号（单词），并把识别出的符号处理成等价的内部中间表示（属性字）；
- 进行一些简单而又有利于下一阶段（语法分析）之处理的工作，如删除注解和空格等无效字符，可能还会进行某些预处理。例如，对于 C 语言程序的 3 类预处理命令：文件包含（include）命令、宏定义（define）命令与条件编译命令，都应在语法分析之前处理，因此在正式进行词法分析之前处理完这 3 类预处理命令。

词法分析程序的输入是源程序字符序列，而输出是属性字序列。词法分析时仅识别开符号，并不识别由符号组成什么语法成分。换言之，词法分析程序仅关注符号的拼写规则，并不关心由符号组成语法成分的语法规则，因此处理的是字符序列，结合词法分析的具体情况，今后将称字符序列为字符串。如前所述，词法分析将以正则文法为基础文法，在正则文法的基础上讨论词法分析。

C 语言符号包括标识符、常量、字符串、标号及界限符等。界限符包括关键字、运算符、括号及其他一些专用符号。为了便于讨论与实践，这里仅以 C 语言的子集为例进行讨论。假定仅包

含下列符号：

标识符

无正负号整数

关键字：void　int　float　char　struct　if　else　while　do　for　return

运算符：+　-　*　/　%　&　&&　||　!　<　<=　>　>=　==　!=　=

括号：　(　)　[　]　{　}

其他：　,　;

标识符与无正负号整数分别定义如下：

<标识符>::=<标识符>字母 |<标识符>数字|字母

<无正负号整数>::=<无正负号整数>数字 | 数字

其中的字母与数字各表示特定的字母（a,b,c…）与数字（0,1,2…）。

似乎定义&&与||等双字符运算符的重写规则形式将不是正则文法的，事实上，例如对于&&可以如下定义：

<逻辑与>::=<&符号>&

<&符号>::=&

这是正则文法规则的定义形式，其他各类双字符运算符可以类似地进行定义。关于关键字可以如下定义，例如，对于关键字 int：

<int-1>::=i

<int-2>::=<int-1>n

<int 符号>::=<int-2>t

其他关键字可类似地定义出来。

可见所有符号的定义都具有下列两种形式：

U::=T　　与　　U::=WT

其中，U、W∈V_N，T∈V_T。因此可用正则文法规则定义各类符号。

把符号的定义与语法成分的定义分开，而不是在分析语法结构的同时进行词法分析，也即独立地用正则文法技术来实现词法分析，这样处理可以使编译程序的结构清晰、条理化，并且便于语法分析的实现，有利于教学，尤其可以突出词法分析与语法分析各自的实现原理，构造更为有效的词法分析程序，高效地实现词法分析，也可以进一步考虑词法分析程序的自动构造。

词法分析程序的实现可以有手工实现方式和自动构造方式两种，下面分别加以讨论。

3.2　词法分析程序的手工实现

3.2.1　实现要点

词法分析程序实现的功能是把源程序字符串变换成等价的内部中间表示，即属性字序列，其输入是源程序字符串，输出是属性字序列。按照软件工程学原理，一般软件系统的开发需经历分析（问题分析、需求分析）、开发（概要设计、详细设计、编程调试等）与运行（运行与维护等）等阶段。对于词法分析程序的具体开发来说，按照其实际情况，可以从简。实现词法分析程序的

大致步骤如下。

- 确定程序设计语言（或其子集），明确相应的字符集、符号及其构造。
- 总体设计。
  - 设计各类数据结构，包括输入（源程序字符串）与输出（属性字序列）的数据结构，以及实现中相关的各类数据结构，重点是属性字的设计。
  - 功能模块分解，明确各个模块（子程序）的功能与相互之间的接口，画出总控流程图及各个子程序的控制流程图。
- 最终编程并设计调试实例，进行调试，完成词法分析程序的研制。

关于第一个方面，即词法分析时所处理的对象：字符集与符号已在前面列出，实现的要点是属性字的设计与标识符的处理。

3.2.2 属性字的设计

词法分析程序的输出是属性字序列，这是一种内部中间表示，它可以有各种不同的具体内部表示形式。对其设计要求有两方面：一是不同的符号都有唯一的表示，能彼此区分开；二是便于后继编译阶段的处理。

由于源程序中的符号之长度各不相同，除了单字符符号（如+与*等），还有双字符符号（如<=与&&等）和关键字（如 if 与 else 等），尤其是标识符的长度可以是任意的。为了有利于以后语法分析阶段的工作，属性字都采用统一的长度，且是固定的长度。为了能相互识别开各个不同的符号，并有利于以后阶段的处理，属性字一般由两部分组成：一部分是符号类，指明一个符号属于哪一类；另一部分是符号值，指明符号本身是什么。属性字的结构如下所示：

符号类	符号值

下面具体讨论属性字的设计。

属性字由符号类和符号值两部分组成：符号类通常用整数编码表示，一个符号类对应于一个编码值；符号值则是符号本身。显然对于关键字、括号与运算符等符号，一个符号类中仅有一个符号，即一个符号类便唯一地定义了一个符号值，即符号本身，因此称这些符号是特定符号。至于标识符，它对于语法分析来说，在所处理的上下文无关文法中仅是一个终结符号，无需考虑是哪一个具体的标识符，因此，标识符统归为一个符号类，即标识符类。至于到底是哪一个标识符，则由属性字的符号值指明，例如，可以是 main，也可以是 x 或 temp 等。一个标识符类可以对应于任意多个符号值，即对应具体的标识符，因此，标识符类是一类非特定符号类。由于标识符可以是任意长度的，为了能符合属性字等长且定长的要求，属性字的符号值部分往往不直接给出标识符本身，而是给出标识符在标识符表中所登录条目的序号或指针。标识符表中登录标识符及相关的属性字，当词法分析程序扫描到一个标识符时，把它的相关条目登录在标识符表中，以所登录的条目的序号或指针作为相应属性字的符号值，通过此序号或指针可以很方便地存取该标识符。关于无正负号整数，情况类似，只是相关属性字的属性值是无正负号整数在常量表中的序号。常量表用来登录一切常量，可能在相关的条目中包含常量的类型（int 与 float 等）信息。

概括起来，属性字中符号类识别开不同类的符号，符号值识别开同类中的不同符号。

表 3.1 中给出了各符号类的编码。

表 3.1

符　号　类	编　码	助 记 忆 符	符　号　类	编　码	助 记 忆 符
无定义	0	$UND	(	19	$LPAR
标识符	1	$ID	)	20	$RPAR
整数	2	$NUM	[	21	$LS
+	3	$PLUS	]	22	$RS
-	4	$MINUS	{	23	$LB
*	5	$STAR	}	24	$RB
/	6	$SLASH	,	25	$COMMA
%	7	$MOD	;	26	$SEMICOLON
<	8	$LT	void	27	$VOID
<=	9	$LE	int	28	$INT
>	10	$GT	float	29	$FLOAT
>=	11	$GE	char	30	$CHAR
==	12	$EQ	struct	31	$STRUCT
!=	13	$NEQ	if	32	$IF
&	14	$ADDR	else	33	$ELSE
&&	15	$AND	while	34	$WHILE
||	16	$OR	do	35	$DO
!	17	$NOT	for	36	$FOR
=	18	$ASSIGN	return	37	$RETURN

为了易读起见，引进了助记忆符。显然，特定符号类的符号也有长度不等的情况，如*与 return，在实际实现时，属性字的符号值必须保持等长且定长。往往引进一个表登录这些符号，让整数值，即这些特定符号在表中条目的序号来代替符号本身。在本书的实现中，引进两个表，一个称为符号机内表示表，另一个是关键字表。关键字表中列出了所处理语言中的一切关键字。符号机内表示表中列出了除关键字外的所有特定符号，包括单字符的符号与双字符的符号。之所以关于关键字另外设定关键字表，这是因为它们全由字母组成，结构类似于标识符，当识别出由字母组成的符号时，首先查看关键字表，这样便可区别开标识符与关键字。

【例 3.1】　设有程序

```
void main ( )
{  int i, j, d;
   if (i>j) d=i-j; else d=j-i;
   d=d/4;
}
```

试写出相应的属性字序列。

对照表 3.1，可写出该程序的相应属性字序列如表 3.2 所示。为直观起见，在属性字符号值中写出符号本身，且作为比较，同时写出符号序列。

表 3.2

输 入 符 号	属性字序列	输 入 符 号	属性字序列
void	27, "void"	=	18, "="
main	1, "main"	i	1, "i"
(	19, "("	-	4, "-"
)	20, ")"	j	1, "j"
{	23, "{"	;	26, ";"
int	28, "int"	else	33, "else"
i	1, "i"	d	1, "d"
,	25, ","	=	18, "="
j	1, "j"	j	1, "j"
,	25, ","	-	4, "-"
d	1, "d"	i	1, "i"
;	26, ";"	;	26, ";"
if	32, "if"	d	1, "d"
(	19, "("	=	18, "="
i	1, "i"	d	1, "d"
>	10, ">"	/	6, "/"
j	1, "j"	4	2, "4"
)	20, ")"	;	26, ";"
d	1, "d"	}	24, "}"

从例 3.1 可见，词法分析程序在扫描（读入）源程序时，删除了空格字符与换行字符等无效字符，单纯地识别开各个符号，在输出的属性字序列中，明显地包含有说明部分的相应属性字。标识符与相应类型的关联将在语义分析阶段进行。

然而，为了提高功效，在属性字中包含更多的属性信息，可以如下地改进属性字的设计：

- 属性字分为特定符号类与非特定符号类两大类，给出区别的标志；
- 属性字中包含符号的更多信息，如包含运算符优先级等信息。

语法分析程序对特定符号类与非特定符号类两大类符号的处理有所不同，在属性字符号类中包含大类的信息将简化处理，可以在属性字符号类中给出大类标志，1 表示特定符号类，0 表示非特定符号类，即特定符号类符号的属性字一般形式如下：

1	符号类	符号值

而非特定符号类符号的属性字有下列的一般形式：

0	符号类	符号值

符号的更多信息可以如下引进，区别是说明符（如 int 等）还是运算符（如+等），对于运算符，可以加入运算符优先级信息等。假定一个属性字的长度统一为 32 位二进位，对于特定符号类符号的属性字可有如下构造。

$\varepsilon_0\varepsilon_1\varepsilon_2\varepsilon_3\varepsilon_4\varepsilon_5\varepsilon_6\varepsilon_7\cdots\varepsilon_{15}\varepsilon_{16}\cdots\varepsilon_{31}$

其中，ε_0=1：特定符号

$$\varepsilon_1=\begin{cases}1 & \text{说明符}\\ 0 & \text{非说明符}\end{cases}\qquad \varepsilon_2=\begin{cases}1 & \text{运算符}\\ 0 & \text{非运算符}\end{cases}$$

$\varepsilon_3\varepsilon_4\varepsilon_5\varepsilon_6$：运算符优先级信息

$\varepsilon_7 \cdots \varepsilon_{15}$：符号类编码

$\varepsilon_{16} \cdots \varepsilon_{31}$：符号值

运算符优先级定义如下：

8：&　！　+　-（单目）

7：*　/　%

6：+　-　（双目）

5：<　<=　>　>=

4：==　!=

3：&&

2：‖

1：=

采用改进的属性字结构后，例 3.1 中的程序的属性字序列如表 3.3 所示。

表 3.3

符　号　类					符　号　值	符　号　类					符　号　值
1	0	0	0	27	"void"	1	0	1	1	18	"="
0	0	0	0	1	"main"	0	0	0	0	1	"i"
1	0	0	0	19	"("	1	0	1	6	4	"-"
1	0	0	0	20	")"	0	0	0	0	1	"j"
1	0	0	0	23	"{"	1	0	0	0	26	";"
1	1	0	0	28	"int"	1	0	0	0	33	"else"
0	0	0	0	1	"i"	0	0	0	0	1	"d"
1	0	0	0	25	","	1	0	1	1	18	"="
0	0	0	0	1	"j"	0	0	0	0	1	"j"
1	0	0	0	25	","	1	0	1	6	4	"-"
0	0	0	0	1	"d"	0	0	0	0	1	"i"
1	0	0	0	26	";"	1	0	0	0	26	";"
1	0	0	0	32	"if"	0	0	0	0	1	"d"
1	0	0	0	19	"("	1	0	1	1	18	"="
0	0	0	0	1	"i"	0	0	0	0	1	"d"
1	0	1	5	10	">"	1	0	1	7	6	"/"
0	0	0	0	1	"j"	0	0	0	0	2	"4"
1	0	0	0	20	")"	1	0	0	0	26	";"
0	0	0	0	1	"d"	1	0	0	0	24	"}"

3.2.3　标识符的处理

在一个源程序中出现最频繁、对程序的影响最大的显然是标识符，编译程序的处理重点是标识符。对于词法分析程序而言，标识符的处理也是十分关键的问题。尽管词法分析程序并不处理源程序中的说明部分，也就是说，并不立即让标识符与它们所代表数据对象的数据类型等属性联系起来，但必须考虑到标识符的如下情况。

① 出现在源程序的不同语法位置上有不同的处理方式。例如，出现在说明部分与出现在控制部分的处理不同，出现在函数定义内和出现在函数定义外的处理不同，等等。

② 同一个标识符出现在不同的上下文中将可能代表不同的对象，例如，在某处代表一个函数

名，而在另一处可能代表简单变量等。

一般说来，对于标识符的处理需要考虑下列 3 个问题：区分标识符的定义性出现与使用性出现、确定标识符的作用域，以及确定标识符属性字中的符号值。

1. 区分标识符的定义性出现与使用性出现

C 语言规定任何标识符必须显式定义，这样就使编译程序能了解一个标识符将代表什么数据对象，而且一般都必须遵循“定义在前，使用在后”的原则。不言而喻，只要是代表同一个数据对象的标识符，它们不论出现在源程序的任何位置上，都应该有相同的属性字。这意味着，第一次扫描到一个标识符时，为该标识符构造属性字，而在其后扫描到该标识符的出现时，不应为它再次构造属性字，而应沿用（即复制）之前所构造的属性字。标识符的第一次出现往往是在源程序的说明部分中，也就是定义性出现，而其后各次通常都是在控制部分中，即，使用性出现。

因此，必须区分标识符的定义性出现与使用性出现。

当标识符出现在源程序的说明部分中，首次被说明而与类型等属性相关联时，这时标识符是定义性出现，不是定义性出现的标识符都是使用性出现。正确的源程序中，一个标识符的定义性出现必定只发生一次。

标识符的定义性出现一般是在说明部分中，但并不是说标识符出现在说明部分中一定是定义性出现。例如，C 语言的下列说明语句：

```
struct date
{  int year;  int month;  int day;
};
struct person
{  char name[8];  char sex;
   struct date birthday;
} man, woman;
```

第二次出现的标识符 date 出现在说明部分中，但不是定义性出现。一般说，标识符作为类型名出现时都不是定义性出现。

当区分开标识符的定义性出现与使用性出现后，词法分析程序通常如下处理：当标识符定义性出现时，把标识符连同为它构造的属性字一起登录入标识符表，当使用性出现时，在标识符表中查得该标识符的条目，复制相应的属性字内容到属性字序列中。

标识符表的结构通过下例了解。

【例 3.2】 标识符表结构的例子。

设有 C 语言程序如下。

```
int max3(int a, int b ,int c)
{  int max;
   if(a>b) max=a; else max=b;
   if(c>max) max=c;
   return max;
}
main( )
{  int x,y,z,max;
   printf("Input values of x,y and z: ");
   scanf("%d%d%d", &x,&y,&z);
   max=max3(x,y,z);
   printf("max(%d,%d,%d)=%d\n", x,y,z,max);
}
```

词法分析程序对该程序词法分析的结果，构造标识符表如表 3.4 所示。其中每一个条目由标识符及相应属性字组成。在词法分析程序扫描到这些标识符的定义性出现时登录入标识符表，在

标识符使用性出现时查此标识符表，找到相应条目并复制属性字到属性字序列中。

表 3.4

max3
00001　"max3"
A
00001　"a"
B
00001　"b"
c
00001　"c"
Max
00001　"max"
X
00001　"x"
Y
00001　"y"
Z
00001　"z"

注意，在此标识符表中未写出标识符 main、printf 与 scanf 等。事实上，它们是系统定义的标识符，它们与其他的系统定义标识符一起，先于程序编写人员定义的标识符（max3 等）被登录在标识符表中。一般来说，标识符定义性出现时，在登录相应条目时若查到在标识符表中已登录有相应条目，表明该标识符重定义；而在使用性出现时，在标识符表中没有查到相应条目，表明该标识符无定义。

上述程序中 max3 的函数定义的属性字序列如表 3.5 所示。

表 3.5

符　号　类	符　号　值	符　号　类	符　号　值
1 1 0 0　28	"int"	1 0 1 1　18	"="
0 0 0 0　1	"max3"	0 0 0 0　1	"a"
1 0 0 0　19	"("	1 0 0 0　26	";"
1 1 0 0　28	"int"	1 0 0 0　33	"else"
0 0 0 0　1	"a"	0 0 0 0　1	"max"
1 0 0 0　25	","	1 0 1 1　18	"="
1 1 0 0　28	"int"	0 0 0 0　1	"b"
0 0 0 0　1	"b"	1 0 0 0　26	";"
1 0 0 0　25	","	1 0 0 0　32	"if"
1 1 0 0　28	"int"	1 0 0 0　19	"("
0 0 0 0　1	"c"	0 0 0 0　1	"c"
1 0 0 0　20	")"	1 0 1 5　10	">"
1 0 0 0　23	"{"	0 0 0 0　1	"max"
1 1 0 0　28	"int"	1 0 0 0　20	")"
0 0 0 0　1	"max"	0 0 0 0　1	"max"
1 0 0 0　26	";"	1 0 1 1　18	"="
1 0 0 0　32	"if"	0 0 0 0　1	"c"
1 0 0 0　19	"("	1 0 0 0　26	";"
0 0 0 0　1	"a"	1 0 0 0　37	"return"
1 0 1 5　10	">"	0 0 0 0　1	"max"
0 0 0 0　1	"b"	1 0 0 0　26	";"
1 0 0 0　20	")"	1 0 0 0　24	"}"
0 0 0 0　1	"max"		

在表 3.5 中，符号值部分依然是为易读起见给出的符号本身。如前所述，属性字是等长且为定长的，往往用整数序号来代替符号本身。由于标识符所代表对象不同，例如有的是简单变量，有的是整型常量，还有的是函数名等，在具体实现时，符号值是不同表中条目的序号。例如，简单变量一般是标识符表序号，对于常量，是常量表序号，而对于函数标识符，是函数信息表序号，等等。

关于上例中 main 函数定义的属性字序列，请读者自行写出。读者不禁会问：main 函数定义中的标识符 max 的属性字是否与 max3 函数定义中的 max 的属性字相同？初看起来，两个函数定义中的 max 都是 int 型的简单变量，属性字没有区别。但显然，这两者代表不同的数据对象，它们的属性字应有所区别，必须区别开这两个代表不同数据对象的标识符。为此，在下面讨论标识符的作用域问题。

2. 确定标识符的作用域

例 3.2 中的标识符 max 出现在 max3 与 main 两个函数定义中，尽管都是 max，事实上在运行时，在两个函数定义内代表两个不同的数据对象。在刚进入 main 函数定义时，变量说明 int x, y, z,max; 使得其中的 max 所代表的变量存在，当执行语句 max=max3(x,y,z)时调用函数 max3，控制进入 max3 函数定义内的函数体，这时由说明语句 int max；定义的 max 也同时存在，用来暂存待返回 main 函数的三个参数最大值。一旦返回，此处的 max 便不复存在。显然两个函数定义内的 max，它们虽然都是 int 型的变量，实际上是两个不同的数据对象。

从概念上说，这两个 max 有不同的定义域，都是在所在定义域内的局部量。什么是标识符的作用域？标识符在程序中，总是在一定的范围内与某类型属性相关联，代表具有所关联数据类型属性的数据对象，这个范围就是标识符的作用域。一般来说，在函数定义中被说明，即与某类型属性相关联的变量，其作用域就是该函数定义（或说函数体）。在该函数定义外部，即使有同名的标识符，它也只代表另一个数据对象。

标识符的**作用域**是标识符与某种类型属性相关联的有效范围。同一个标识符出现在不同的作用域中将代表不同的数据对象，试看下例。

【例 3.3】 标识符作用域的例子。

```
/* Scope for identifier */
int m=3;
int fun(int a, int b)
{  int m=4, x=5;
   return (a+m)*(x-b)-m;
}
main( )
{  int x=1, y=1;
   printf("fun(%d,%d)/%d=%d\n", x, y, m, fun(x, y)/m);
}
```

该程序中标识符 m 与 x 各被定义了两次，被赋以值 3 的 m 是全局量，其作用域是除了 fun 函数定义之外的整个程序，因为它在其内被重新定义，而被赋以值 4 的 m 是 fun 函数定义内的局部量，作用域就是 fun 的函数体。标识符 x 分别在 fun 与 main 的函数定义内被定义，作用域分别是这两个函数定义的函数体，分别置初值为 5 与 1。该程序的运行结果是显示输出：

```
fun(1,1)/3=5
```

请自行验证。

确定标识符的作用域，其作用在于：当词法分析程序扫描到某个标识符的使用性出现时，通过确定其作用域，能确定该标识符所代表的数据对象，从而确定其具有的类型等属性。

如何确定使用性出现的标识符的作用域？例如，如何确定例 3.3 中程序第 5 行中的 m 是值为 4 的 m，而第 9 行中的 m 是值为 3 的 m？词法分析程序必须能自动机械地、不受人的干预地作出判定。

首先明确：确定标识符作用域的原则是什么。为便于叙述，引进分程序的概念。分程序是包含说明部分的复合语句。函数定义的函数体可看成是一个分程序。如果在 for 语句的循环体中引进变量说明，则循环体也就成了一个分程序。因此，C 语言中也存在嵌套的分程序结构。关于嵌套分程序结构，标识符的作用域可如下确定。

- 一个标识符的作用域，是对其定义的分程序；
- 外层分程序中定义的标识符，可以自动在内层分程序中继承，只要在内层分程序中没有对该标识符重新定义。

这仅根据程序正文决定与标识符相关联的类型等属性，因此称为静态作用域。为了确切地确定一个使用性出现的标识符的作用域，采用静态作用域法则，并按下列“最接近的嵌套”约定来确定其作用域：

- 分程序 B 说明部分定义的标识符，它的作用域包括 B；
- 如果一个标识符 x 不是在分程序 B 中定义的，则 B 中使用性出现的 x 的作用域，是 B 的外围中这样的分程序，它包含对 x 的定义，却比其他包含对 x 的定义的分程序更接近被嵌套的 B。

为了实现正确地确定标识符的作用域，简单可行的办法是利用后进先出栈。确定标识符的作用域的基本思想如下：设立一个标识符栈，初始时下推入一个条目，其内容是区别于标识符条目的所谓间隔条目。当进入一个分程序的说明部分，把该说明部分中定义的标识符逐个下推入标识符栈，建立相应的条目，这时所建立的所有条目称为（作用域）当前层。当进入该分程序的控制部分时，对使用性出现的标识符，在当前层中，从所建立的最后标识符条目自顶向底查找。如果查到，则得到相应的定义，如果查不到，则越过间隔条目，继续自顶向底在外层中继续查找，直到查到。如果最终都没有查到，则表明该标识符没有定义。

这时有两种情况：

① 在达到当前分程序的结束符号（}）之前又进入一层分程序，则下推入一个间隔条目，把新进入的分程序之说明部分中定义性出现的标识符下推入标识符栈，建立相应的条目，这一层作为当前层。

② 当扫描程序扫描到结束当前层的分程序结束符号（}）时，上退去当前层的所有条目，直到间隔条目，并上退去此间隔条目，外层作为新的当前层。当达到整个程序结束处时，标识符栈成为空栈。

【例 3.4】　扫描过程中标识符栈构造过程的例子。

设有 C 语言程序如下。

```
float x=0;
void f                          (1)
(int q)
{ float p;                      (2)
  …x…q…p…
}                               (3)
main ( )
{ int x=1, i=2, p=3;            (4)
   …x…i…p…                      (5)
   for (i=1; i<=10; i++)
   { char p;  float x;  int t;  (6)
     …x…i…p…t…
   }                            (7)
     …x…i…p…t…
}
```

标识符栈构造过程示意图如图 3-1 所示。

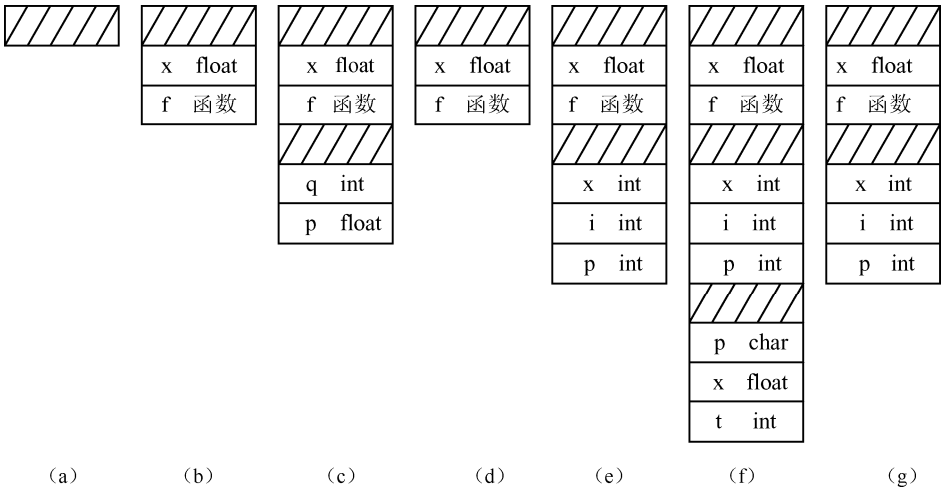


图 3-1

这里对照图 3-1 简略说明扫描过程。扫描开始时，标识符栈如图 3-1（a）所示仅为空栈，这时下推入了一个区别于标识符条目的间隔条目。扫描到了(1)这一行时，标识符栈如图 3-1（b）所示。当扫描到了(2)这一行时，已进入新的一层作用域，标识符栈如图 3-1（c）所示。下一行中的标识符 q 与 p 将在当前层作用域中查到，分别为 int 与 float 型，而 x 则在外层作用域中查到，类型为 float。当扫描到(3)这一行时，已退出当前层，标识符栈如图 3-1（d）所示，它与（b）相同。当扫描到了(4)这一行时，进入新的一层作用域，标识符栈又进入新的一层，如图 3-1（e）所示，从图可见，这仍是第二层。这是显然的，因为 f 与 main 这两个函数定义是并列的，退出第二层的 f 函数定义，而进入第二层的 main 函数定义。扫描到(5)这一行时，标识符 x、i 与 p 都可以在当前层中查到，都是 int 型。扫描到(6)这一行时，又进入新的一层作用域，标识符栈如图 3-1（f）所示，下一行中的标识符 x、p 与 t 在当前层查到，类型分别为 float、char 与 int，而标识符 i 则在外层查到，为 int 型。当扫描到(7)这一行时，退出一层，标识符栈如图 3-1（g）所示，它与图 3-1（e）相同。因此，扫描到下一行时，查到标识符 x、i 与 p 都为 int 型，标识符 t 则为无定义。当扫描了程序最后一行的“}”时，再自顶向底退去当前层直到间隔条目，并退去此间隔条目，因此，标识符栈又如同图 3-1（b）所示。由于整个程序已扫描完毕，再退去当前层而在标识符栈仅有间隔条目，即空栈。扫描工作全部结束。

从上述实例可以看出，利用后进先出栈可以正确地确定标识符的作用域。但这种标识符栈仅在此词法分析阶段引进，不保留在以后使用，对于标识符，应能从其属性字很方便地获取相关联的类型等属性信息，为此可以如下地处理。

当标识符定义性出现时，登录标识符表，建立相应条目，另外按前述作用域确定算法，还下推入标识符栈，建立相应条目，其中包括标识符本身及标识符表相应条目的序号，通过标识符表序号可以把标识符正确地定位在所在的作用域。这时既建立标识符栈还建立标识符表，简单的办法是把标识符栈与标识符表这两者结合起来，或者说，用这样的标识符栈代替标识符表，这栈只下推入标识符条目，而决不上退去标识符条目。由于原先上退去的标识符作用域层，现在不再退去，当在当前层作用域中查找，没有查到而自顶向底继续查找时，便得跳过并行（同层次）的作用域层后再继续向底查找，因而，查找算法将变得复杂一些。不言而喻，这时必要的是在标识符条目中必须包含

作用域层次的信息。这样的标识符表可保留在整个编译期间，甚至在运行时刻使用。

3. 确定标识符属性字的符号值

不同的符号应有不同的属性字，对于特定符号类，一个符号类也就是一个符号，即有唯一的属性字。对于非特定符号类，同一个符号类可以有多个不同的符号，确切地说，对于标识符，同为标识符类，却可以有多个不同的标识符，为了相互区别，必须利用符号值，通过符号值区别开不同的标识符。如标识符的作用域一节中所讨论的，为了确定标识符与类型等属性的正确关联，需要利用标识符表的序号。对于属性字序列中使用性出现的标识符，在它的属性字中，符号值置为该标识符在标识符表中所登录条目的序号。

尽管进行词法分析时不实际处理程序中的说明部分，仅把说明部分也处理成相应的属性字序列，但以标识符在标识符表中所登录条目的序号作为属性字符号值，在语义分析阶段可以十分方便地取到相应的类型等属性信息。

上述是关于标识符的属性字符号值的论述，无正负号常量的情况类似，在常量的属性字中，符号值可以是常量在常量表中所登录条目的序号。

要说明的是，程序中的说明部分是没有相应的目标代码的。构造类型和函数定义一般包含较多的信息，例如数组不仅涉及数组（元素）的类型，还涉及数组的维数与各维的上下界等。这些信息不可能在一个标识符表条目中全部反映出来，通常需要引进相应的信息表，如数组信息表与函数信息表等。对于这些情况，合适的处理方法是使用相应信息表的序号。

当引进这些信息表时，可以按如下方式确定非特定符号的属性字符号值：

- 对于基本类型的简单变量，以标识符表序号或编号作为属性字符号值。
- 对于常量，以常量在常量表中的条目序号作为属性字符号值。
- 对于其他各类标识符，以相应信息表序号作为属性字符号值。

如果在属性字符号值中再添入作用域嵌套层次或所属作用域编号，这样确定的符号值可确切地区别开各个不同的符号。

3.2.4　词法分析程序的设计和编写

掌握了词法分析程序所处理源程序中能出现的一切符号，即源语言的字符集和符号集，并设计了属性字的结构，也讨论了标识符之处理的三个问题之后，可以着手进行词法分析程序的设计与实现。首先设计词法分析的总控流程图以及相关子程序的功能，然后讨论相关数据结构的设计，最后讨论实现问题。

1. 总控流程图

词法分析程序的处理思想很简单，即扫描（读入）源程序字符串，逐个取到符号，并生成相应的属性字，直到源程序结束。取符号实质上就是识别符号。词法分析程序的总控流程图如图 3-2 所示。显然，核心是取符号子程序，其控制流程图如图 3-3 所示。

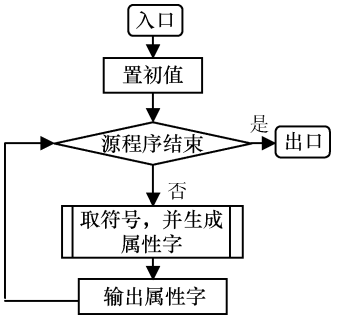


图 3-2

取符号子程序的处理思想很简单：当取到一个字母字符时，将识别出是标识符或关键字，因此逐个取到后继字母或数字字符，直到非标识符组成字符，这时查关键字表，或者查到，从而识别出关键字，生成相应属性字，或者未查到，因此是标识符。第一次是定义性出现，构造标识符表（登录标识符条目），其余各次是使用性出现，则查标识符表，把查得的属性字复制入属性字序列。

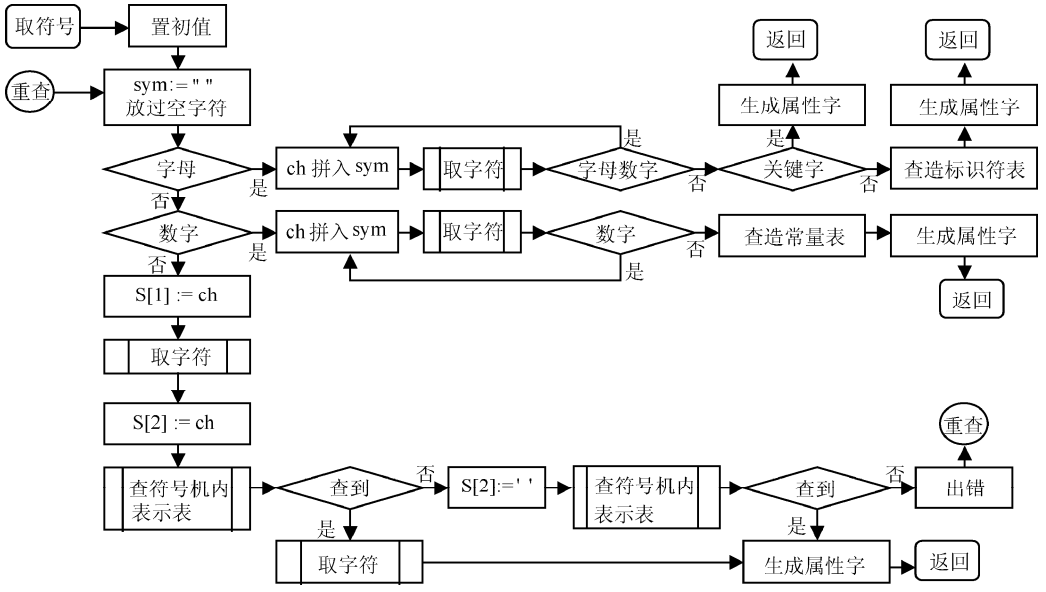


图 3-3

当取到一个数字字符时，将识别出无正负号整数，因此逐个取到后继各数字字符，直到非数字字符。这时查常量表，如果没查到，则构造常量表（登录常量表条目）。不论查到与否，都生成相应的属性字。

对于关键字，如前所述，如同标识符那样处理。对于其余的符号，不论是双字符的还是单字符的，都按双字符符号处理：查符号机内表示表，生成相应的属性字。这时先对相邻两个字符查表，如果查到，则识别出双字符符号，如果查不到，则让第一个字符后跟一个空格字符去查此表，如果查到，即为单字符符号。

请注意，在查到是双字符符号时，将进行取字符，而在其他各种情况识别出一个符号时，并不进行取字符。这是因为：在其他所有情况，都已取到当前所识别符号的后继第一个字符，无需取字符，仅在识别出双字符符号时未取到而需取字符。为统一起见，对取符号子程序作如下约定。

- 进入取符号子程序时已取到当前符号的第一个字符；
- 离开取符号子程序时已取到当前符号之紧接后继字符。

C 语言程序中通常可能包含注释，注释由词法分析程序删除，为此，请读者自行考虑修改图 3-3 中所示控制流程图。

从取符号子程序的控制流程图可见，涉及的子程序包括取字符子程序、查关键字表子程序、查造标识符表子程序、查造常量表子程序以及查符号机内表示表子程序等。属性字可以用直接生成方式生成，不一定用调用子程序的形式生成。

下面简略介绍这些子程序的功能。

① 取字符子程序。该子程序的功能是：从源程序文件中取一个字符，返回给取符号子程序。源程序文件一般是磁盘文件或软盘文件，为了提高效率，减少对文件的频繁存取，往往设立一个适当大小的输入缓冲区，把源程序从源程序文件中读到输入缓冲区，每次读入缓冲区大小那样多个的字符。每当调用取字符子程序，便从输入缓冲区中读出（扫描）字符。一旦缓冲区中的字符全部读出，则再次从源程序文件读入到输入缓冲区。需注意几点：

- 源程序是文本文件，一般按行首缩进格式书写和键入，合适的做法是引进行计数器与列计

数器，以便当发现错误时，输出的报错信息能指明出错的位置，并显示所在行的源程序。

- 空格字符与换行字符的处理必须注意，这两类字符虽不是 C 语言字符集字符，但它们都标志一个符号结束，有结束一个符号的作用，不能删除。对于换行字符，在取到它时，行计数器值加一，并且应该把它改为空格字符返回到取符号子程序。
- 其他控制字符都是无效字符，一律删除。如果有非法字符，可以简单地删除，但一般发出报错信息，同时指明错误出现的位置。

如果可能在输入缓冲区中已取到最后一个字符，而还未识别出一个符号，例如，输入缓冲区取到最后一个字符时，else 符号仅扫描到 s，这时应采用双缓冲区技术，即两个缓冲区依次轮流读入源程序字符串，当一个输入缓冲区已扫描完时，把源程序读入到另一缓冲区，以避免发生上述情况。对于源程序不太长，可一次性全部读入的情况，可不考虑双缓冲区技术。

请注意，当已读入源程序中全部字符时，取字符子程序应把源程序已结束的信息反馈给取符号子程序，以便结束词法分析工作。

词法分析程序所处理的源程序中可能出现的一切合法字符列出在字符表中。可以说，字符表中的一切字符构成文法的字母表。读入源程序一个字符，便查该字符表，检查是否是合法字符。这样的做法当然可以，但较花时间。因为如果识别出合法符号，组成的字符必然合法，不能构成符号的字符必定不合法，可以不查字符表。

② 查造标识符表子程序。该子程序的功能是：当标识符定义性出现时，在标识符表中登录标识符与相应属性字信息，即构造标识符表条目。当标识符是使用性出现时，在标识符表中查找该标识符的相应条目，取到相应的属性字，这时必须结合前面所讨论的关于标识符作用域的确定。一般地说，定义性出现是第一次出现，且在说明部分；使用性出现在控制部分。如果判别正扫描到的标识符是处于说明部分还是控制部分，就可以发现标识符重定义或无定义的错误。但那样将给词法分析程序的实现带来一定的复杂性。为简化起见，初学者可以假定源程序是语法上正确的，每个标识符总有显式定义，且定义在前，使用在后。因此可以简化如下：扫描到一个标识符便查标识符表，如果查到，回送相应的属性字信息，否则假定标识符是定义性出现而造标识符表，即构造相应标识符条目。不论哪种情况，必须考虑标识符的作用域问题。

③ 查造常量表子程序。该子程序的功能是：扫描到一个常量时，如果是第一次，就把该常量登录到常量表中，否则查常量表。不论是否第一次，总是回送常量表序号作为属性字中的符号值。常量表条目中可以给出常量类型的信息，这是易于实现的，且可以避免重新判别类型的麻烦。

④ 查关键字表子程序。该子程序的功能是：判定正扫描到的一列字母字符是否组成关键字，如果是，就构造关键字的属性字，否则作为一般的标识符处理。由于关键字的构造类似于标识符，这样处理将有利于功效的提高，对于编译程序的开发来说，也非常容易实现。

⑤ 查符号机内表示表子程序。该子程序的功能是：识别单字符与双字符的专用符号，符号机内表示表中的符号均按双字符符号形式存放。当是单字符符号时，第二个字符置为空格字符，同时在此表内保存各符号的相应属性字，当查到是某一符号时，回送相应属性字。

一般来说，在编写程序前，应该先画出控制流程示意图，以明确思路，然后再编写程序。上述各子程序，除了查造标识符表子程序因其涉及标识符作用域等而稍复杂外，其余都较简单，可直接编程，必要的是为这些表格设计数据结构，这问题在后面讨论。

在总体设计中，也应考虑到源程序的出错处理问题。应该要求一次编译能查出尽可能多的错

误，甚至全部错误，以便程序编写人员进行改正。词法分析中发现的源程序错误是拼写错误（包括键入错误），比较简单，一般一处仅一个字符错误，不会连续多个字符错。实现词法分析程序的出错处理时，可以指明错误出现在源程序中的行列位置，同时，可以作出修正的尝试，例如，删除出错的字符后继续，请参看第 8 章相关章节。

当用 C 语言实现时，不言而喻，一个子程序对应于一个函数定义。

2. 数据结构的设计

从上面的讨论可见，词法分析程序涉及的数据结构包括输入缓冲区与属性字序列，其他还包括字符表、符号机内表示表、关键字表、标识符表、常量表以及标识符栈。下面以 C 型语言进行设计。

输入缓冲区的数据结构简单地是一个字符数组，这不再详述。

属性字序列是一个数组，其元素是属性字。属性字的数据结构可设计如下，其中为便于阅读，符号值部分的值是符号本身。

```
typedef struct 属性字
{ int 大类;
  符号类类型  符号类;
  符号值类型  符号值;
} 属性字类型;
```

其中符号类类型可设计如下：

```
typedef struct 符号类
{  int 符号大类; /* 1：特定符号类；0：非特定符号类 */
   int 说明符标志; /* 1：说明符；0：非说明符  */
   int 运算符标志; /* 1：运算符；0：非运算符  */
   int 运算符优先级;
   int 符号类编码;
} 符号类类型;
```

符号值类型可以根据具体属性确定，例如，为了易读，可以是符号本身，这时是字符数组类型，也可以是代表序号的整型值，利用这个序号很容易得到符号本身。

属性字序列可以用数组类型设计如下：

```
   属性字类型  属性字序列[MaxAttriWordNum];
```

其中，MaxAttriWordNum 是可允许的属性字序列最大长度。

标识符表条目的数据结构可基于属性字类型设计如下：

```
typedef struct
{  char 标识符[MaxIdLength];
   /*当符号值为符号本身时此成员变量可删 */
   属性字类型  标识符属性字;
   int  条目序号;  /* 当符号值为序号时，此成员变量可删 */
} 标识符表条目类型;
```

标识符表的数据结构可设计如下：

```
标识符表条目类型  标识符表[MaxIDTLength];
```

标识符栈的数据结构可设计如下：

```
struct
{  int 标识符栈顶;  /* 一般取名为 top  */
   栈条目类型  栈内容[MaxStackDepth];
```

```
} 标识符栈;
```

其中，栈条目类型可设计如下：

```
typedef struct
{  char 标识符[MaxIdLength];
   int 标识符表序号;
} 栈条目类型;
```

符号机内表示表仅包含单双字符符号类，符号本身可包含在属性字中，但考虑到属性字为序号的情况，符号机内表示表的数据结构可设计如下：

```
struct
{  符号类类型  符号类;
   char  符号[3];
} 符号机内表示表[MaxSymbolNum];
```

如前所述，为查表方便，单双字符符号统一成双字符符号，这时对于单字符符号，第二个字符置为空格字符。可以以（C 语言）赋初值方式设置好该表，例如，

```
struct
{  符号类类型  符号类;
   char 符号[3];
}  符号机内表示表[MaxSymbolNum]=
{ {{1,0,1,6,3},"+ "},   {{1,0,1,6,4},"- "}, {{1,0,1,7,5},"* "}, { {1,0,1,7,6},"/ "},
   {{1,0,1,7,7},"% "},  {{1,0,1,5,8},"< "},  {{1,0,1,5,9},"<="},  {{1,0,1,5,10},"> "},
   {{1,0,1,5,11},">="}, {{1,0,1,4,12},"=="},{{1,0,1,4,13},"!="}, …,
   {{1,0,0,0,23},"{ "},  {{1,0,0,0,24},"} "},  {{1,0,0,0,25},", "},  { {1,0,0,0,26},"; "}
};
```

关键字表类似于符号机内表示表，可以以（C 语言）赋初值方式设置如下：

```
struct
{  符号类类型  符号类;
   char 符号[MaxLength];
}  关键字表[MaxKeyWordNum]=
{ {{1,0,0,0,27},"void"},  {{1,1,0,0,28},"int"},     {{1,1,0,0,29},"float"},
   {{1,1,0,0,30},"char"},  {{1,1,0,0,31},"struct"}, {{1,0,0,0,32},"if"},
   {{1,0,0,0,33},"else"},  {{1,0,0,0,34},"while"},  {{1,0,0,0,35},"do"},
   {{1,0,0,0,36},"for"},   {{1,0,0,0,37},"return"}
};
```

3. 各子程序的实现

查符号机内表示表与查关键字表，由于两表有类似的结构，查这两个表的相应程序的控制流程图类似，也比较简单，与通常的查表并无很大的不同。

标识符表虽然有类似的结构，但因标识符的处理涉及区别定义性出现与使用性出现，以及确定作用域的问题，所以查造标识符表子程序，通常除了给出待查标识符作为参数外，还给出一个标志参数，指明是定义性出现还是使用性出现。如果仅一层作用域，可以假定源程序是正确的，以第一次出现作为定义性出现，其余各次作为使用性出现，这样可不识别标识符的所在语法位置。在一般的情况下，必须根据上下文，识别标识符在程序中的语法位置，粗略地说，识别是说明部分还是控制部分。这将使词法分析程序的实现变得复杂一些。在确定标识符的作用域时，必须使用后进先出的标识符栈，这样，既有标识符表，又有标识符栈。标识符表可处理成只下推入标识符栈条目，而决不上退标识符栈条目的标识符栈（最终的内容），因此可以这样实现：当把一个标识符的相应条目下推入标识符栈时，同时在标识符表中登录一个条目，以此条目的序号作为属性字中的属性值。当标识符栈退栈时，并不

在标识符表中删除相应条目，这样最终的标识符表便是所需的。

常量表的结构类似于除标识符表外的其余各表，查造常量表的实现是简单的，只需第一次（未查到）时登录，其他各次都是取得所查常量在常量表中的序号。

读者可以以例 3.1 与例 3.3 中的 C 语言程序作为实例，测试自己所研制的词法分析程序工作的正确性。

3.3 词法分析程序的自动生成

3.3.1 词法分析程序自动生成的基本思想

词法分析程序的功能，概括地说，就是进行词法分析。词法分析程序的自动生成，也就是说，自动生成或构造一个程序，由此程序来实现词法分析。

各个不同的程序设计语言有各自的字符集、各自的符号类，也可以有各种不同的机内表示。事实上，尽管各种程序设计语言不同，但各类符号的结构，在不同的程序设计语言中都是大同小异的。如果为两个程序设计语言分别独立实现词法分析程序的自动生成，显然是事倍功半。例如，对于标识符，其结构可概括如下：

标识符首字符　标识符组成字符 … 标识符组成字符

对于 PASCAL 语言，标识符首字符是（大小写）字母字符，而标识符组成字符是字母字符与数字字符。对于 C 语言，标识符首字符除了字母字符外，还可以是下划线“_”，标识符组成字符也可以是下划线“_”。不论哪一种语言，只要扫描到的字符是标识符首字符，便意味着识别出一个标识符（当然应排除关键字的情况），只要此后扫描到的字符是标识符组成字符，便把此字符拼入已识别的标识符部分中，当扫描到的不再是标识符的组成字符时，便识别出一个标识符。

其他各类符号的组成大致有如下几类结构，即，

无正负号整数：　数字字符…数字字符

单字符界限符：　界限符

双字符界线符：　界限符 界限符

对于 FORTRAN 之类语言，还有另一类结构的符号，例如・LG・与・EQ・等。由于 C 语言中不存在这样的符号，对这类符号不拟讨论。

因此可以为各个不同的程序设计语言设计一个通用扫描算法，实现各个不同程序设计语言的词法分析，只需为一个程序设计语言给出标识符的结构，即标识符首字符表、标识符组成字符表，以及一切界限符及单字符与双字符符号表等的信息，还包括各类符号的编码表信息。

现在问题是如何用一种统一的方式来给出为实现词法分析所需要的信息？一般做法是引进所谓的关于源语言的符号说明书。另外，设计一个构造程序，以符号说明书作为输入，从中取到字符集、符号构造及机内表示等信息，生成一些表，如字符表、符号机内表示表与关键字表等，供通用扫描算法使用。

自动生成词法分析程序的程序称为词法分析程序的构造程序。按照上述处理思想，自动生成词法分析程序的构造程序实质上是自动生成通用扫描算法所需的各种表。图 3-4 展示了为词法分析程序自动生成所需的构造程序与通用扫描算法及程序设计语言的符号说明书之间的联系。

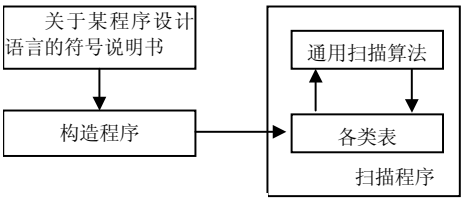


图 3-4

符号说明书是语言相关的，更改符号说明书，也就可以改变词法分析程序所能识别的符号集。一旦从符号说明书生成所需的各个表，构造程序便再无存在的必要。

下面是符号说明书的一个样例。

【例 3.5】　某程序设计语言的符号说明书的例子。

```
BEGIN  SCANNER
    DIGIT  0  1  2  3
    IDBEG  A  B  C  D  E  F  G
    IDCHAR  a  b  c  d  e  f  g  0  1  2  3  4  5  6  7  8  9
    INVISICHAR  X20
    DELIM  +  -  *  /  %  <  <=  >  >=  ==  !=  &  &&  ||  !  =
    DELIM  (  )  [  ]  {  }  ,  ;
    RES   3  +  -  *  /  %
    RES   8  <  <=  >  >=  ==  !=
    RES   14  &  &&  ||  !  =
    RES   19  (  )  [  ]  {  }  ,  ;
    KEY   27  void  int  float  char  struct
    KEY   32  if  else  while  do  for  return
END
```

这个符号说明书中指明了标识符的组成字符与无正负号整数的组成字符，并给出了所有关键字与其他各类符号，而且提供符号的编码，即+、-、*、/与%的编码从 3 开始，依此加 1；&、&&、||、!与=的编码则从 14 开始，依次加 1；而关键字的编码则从 27 开始，直到 37 为止。

不言而喻，标识符的编码与无正负号整数的编码是隐指的，即前者为 1，后者为 2。且这两类为非特定符号类，其他都是特定符号类。

注意符号说明书中，INVISICHAR 意指“不可见字符”，X20 是空格字符的 ASCII 编码（十六进制），即相应字符是空格字符。之所以特别强调空格字符，是因为它不是不起作用的字符，它标志一个符号的结束。

显然这个符号说明书十分简单，它用来产生的属性字中仅包含符号编码。如果需要指明哪些是说明符，哪些是运算符，运算符的优先级又如何，则可以在符号说明书中添加相关内容。例如，

```
    PRIORI  8  &  !
    PRIORI  7  *  /  %
    PRIORI  6  +  -
    PRIORI  5  <  <=  >  >=
    PRIORI  4  ==  !=
```

等等。

读者可能会问：一般的无正负号实数如何处理？更一般的符号又如何处理？

上述符号说明书只能描述处理标识符、无正负号整数、关键字与单双字符界限符这几类，灵活性受到限制，使程序设计语言的表达能力受到限制。为此引进所谓的正则表达式，进一步引进相关的一些概念，如状态转换图等。

3.3.2 正则表达式

1. 正则表达式的概念

正则表达式是描述程序设计语言符号组成的好工具。例如，让 D 表示数字，L 表示字母，并采用第 2 章中所引进的扩充表示法，D{D}描述无正负号整数的组成，而 L{L|D}描述标识符的组成，只需记住：用“{”与“}”表示由其括住的内容可以出现任意多次。这十分容易理解，数字后面跟以若干个数字，当然是（无正负号）整数，字母后跟以若干个字母或数字，当然是标识符。D{D}与 L{L|D}就是正则表达式的实例。有了正则表达式，可以用它来描述无正负号实数如下：

```
e=D{D}.D{D}[E[+|-]D{D}]  |  D{D}E[+|-]D{D}
```

其中字母 E 表示：其后跟的数字是 10 的幂次，该正则表达式 e 描述了两类实数，第一类是有小数点、可能没有指数部分的实数，第二类是没有小数点的实数，但这时必需有指数部分。

下面来说明一般情况下如何在某个字母表上构造**正则表达式**。

假定有字母表 Σ，

- 若 a 是 Σ 中的一个元素，a∈Σ，则 a 是 Σ 上的正则表达式；
- 若 e、e_1 与 e_2 都是 Σ 上的正则表达式，则(e)、e_1e_2、$e_1|e_2$ 与{e}都是 Σ 上的正则表达式。

按照严格的定义，空串ε与空集∅也都是 Σ 上的正则表达式。正则表达式∅的引进完全是为了理论讨论上的完备性。

从上述可知，字母表中的元素都是正则表达式，这种是原子正则表达式，因为它们是不可再分解的。一般的正则表达式可以从原子正则表达式或较小的正则表达式通过加括号或一些运算来构造，这些运算包括连接（产生 e_1e_2）、联合（产生 $e_1|e_2$）与闭包（产生{e}）等。

【例 3.6】 设 Σ_1={0,1}， 则(0|1){0|1}是 Σ_1 上的正则表达式。

【例 3.7】 设 Σ_2={A,B,0,1}，则(A|B){A|B|0|1}是 Σ_2 上的正则表达式。

正则表达式可以按照上述方式构造，但它有什么具体意义？它代表字符串集合。例如例 3.6 中 Σ_1 上的正则表达式(0|1){0|1}，它是通过联合、加括号、闭包与连接等运算构造的，代表 Σ_1 上的字符串集合 B={0,1,00,01,10,11,000,001,…}，即以 0 或 1 打头、后跟以任意多个 0 或 1 的字符串所组成的集合，也即二进制数集合。例 3.7 中 Σ_2 上的正则表达式(A|B){A|B|0|1}，它也是通过联合、加括号、闭包与连接等运算构造的，代表 Σ_2 上的字符串集合 ID={A,B,AA,AB,A0,A1,BA,BB,B0,B1,AAA,AAB,…}，即，以字母 A 或 B 打头，后跟以任意多个 A、B、0 或 1 的字符串所组成的集合，也即字母仅 A 与 B、数字仅 0 与 1 的标识符集合。

正则表达式所代表的集合称为正则表达式的值，这个集合是一个**正则集**。

关于正则表达式的值一般计算方法如下。

字母表 Σ 上的正则表达式 e 的值是字母表 Σ 上的正则集，记作|e|，计算规则如下：

```
|ε|={ε}                          |∅|=∅
|a|={a}                          |(e)|=|e|
|e1e2|=|e1||e2|={xy | x∈|e1|, y∈|e2|}
|e1|e2|=|e1|∪|e2|={x|x∈|e1|或 x∈|e2|}
|{e}|=|e|*    即|e|的闭包
```

正则表达式∅的值是空集，即对应于不包含任何字符串的正则集，引进的目的仅仅是为了理论上的完备性。

请读者对照上述计算规则，自行计算|(0|1){0|1}|与|(A|B){A|B|0|1}|。

假设某字母表上的两个正则表达式 e_1 与 e_2 的值相等，即它们表示的正则集相同，则称它们是等价的，记作 $e_1=e_2$。

对照扩充表示法，文法重写规则 S::=(0|1){0|1}，它等价于：

```
S::=0|1          S::=S0|S1
```

显然从 S 所能推导得到的字符串所组成的集合，与前面的字符串集合 B 相同。而重写规则 I::=(A|B){A|B|0|1}，它等价于：

```
I::=I(A|B|0|1)    I::=A|B
```

显然从 I 所能推导得到的字符串所组成的集合，与前面的字符串集合 ID 相同。因此正则表达式与扩充表示法的正则文法重写规则的右部相当，但它更明确直观地描述了一类符号的结构。

与由正则文法定义的语言相比较，用正则表达式更容易理解定义的是什么符号集合，这类符号如何构造，特别是可以利用某个正则表达式自动构造词法分析程序，它所识别的正是该正则表达式所代表的字符串集合中的字符串，从而减轻实现词法分析程序时工作的单调乏味程度。

事实上，正则表达式在信息处理，特别是信息检索中有广泛的应用。正则表达式是模式匹配与模式识别的重要工具。

2. 正则表达式与词法分析程序的实现

前面所给词法分析程序仅能处理 4 大类符号，即标识符、无正负号整数、单字符专用符号与双字符专用符号，显然这是远远不够的。自动构造的词法分析程序应能识别程序设计语言中的任何符号，包括可能即将扩充的符号。这意指，程序设计语言的符号说明书中应能描述任何希望包含的符号。正则表达式是描述符号的结构的好工具，因此，可以把正则表达式应用于程序设计语言的符号说明书。

如何应用正则表达式来实现词法分析程序的自动生成?

应用正则表达式实现词法分析程序自动生成的思路大致如下。

- 让正则表达式对应于一个状态转换图或有穷状态自动机；
- 从状态转换图或有穷状态自动机对应到词法分析程序。

（1）从正则表达式到状态转换图

状态转换图是由结点与弧组成的有向图，结点对应于状态，弧对应于状态的转换。从正则表达式生成状态转换图，思路如下：首先建立一个有向图，它只有开始状态结点与终止状态结点，两结点用一弧相连，弧上标记就是所给的正则表达式。然后应用如图 3-5 所示的 3 个转换规则对所给出的正则表达式进行转换，在还可以应用转换规则时，继续应用转换规则进行转换，直到一切弧上的标记都是原子正则表达式，不能再应用转换规则时结束。

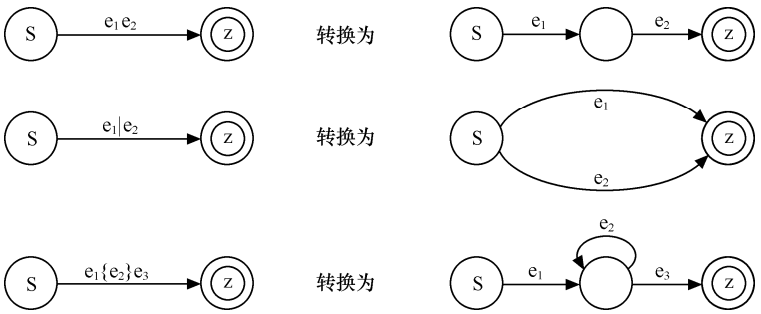


图 3-5

下面以具体例子加以说明。

【例 3.8】 从正则表达式构造状态转换图的例子。

设有正则表达式 L{L|D}|D{D}，试从它构造相应的状态转换图，构造步骤如图 3-6 所示。

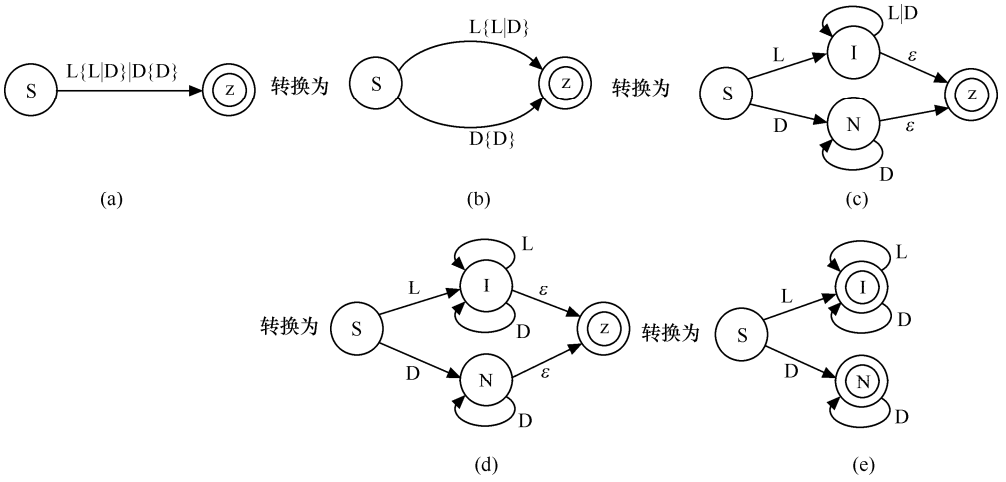


图 3-6

最终所得为所求的状态转换图。为什么称这种图为状态转换图，它有什么作用？对照图 3-6（e）中的状态转换图进行解释。

在这个有向图中，S 为开始状态，I 与 N 都为终止状态，用双圆标志。从结点 S 到结点 I 的弧上有一标记，即字母 L，L 是英文单词 Letter 的首字母，所以 L 表示字母字符，这表示当在状态 S 时扫描（读入）的字符是字母字符，便从状态 S 行进到状态 I，I 是英文单词 Identifier（标识符）的首字母，这表示：扫描到了字母字符，便进入标识符状态，将识别出一个标识符。从结点 I 到自身的两个弧，弧上标记分别为 L 与 D，L 的意义同前，D 是英文单词 Digit（数字）的首字母，这表示：在状态 I 时扫描到字母字符或数字字符时，继续保持在标识符状态。除非扫描了非字母或数字字符才结束。

从结点 S 到结点 N 有一个弧，弧上的标记为 D，这表示：扫描到了数字字符，便将从状态 S 行进到状态 N，N 是英文单词 Number 的首字符，这表示：开始状态 S 时扫描到数字字符，便进入整数状态，将识别出一个（无正负号）整数。从结点 N 到自身的弧上标记为 D，表示在状态 N，扫描到的字符是数字字符时，将保持在整数状态，除非扫描到了非数字字符。

这一有向图显然用于识别标识符与整数，这种有向图称为状态转换图，它刻画了从一个状态在一个输入字符下转换（行进）到下一个状态的情况。**状态转换图**是为识别正则文法的句子而专门设计的有向图。例如图 3-6（e）是识别标识符与（无正负号）整数这两类正则文法句子的。这里的正则文法句子，不言而喻指的是程序设计语言的符号。当用一个正则表达式来描述一个程序设计语言的一切符号时，例如：

```
L{L|D}|D{D}|+|-|*|/|%|<|<=|…
```

就能通过相应的状态转换图识别所列的一切符号。

状态转换图是用来识别正则文法句子的有向图。问题是：它如何与正则文法相联系？一般情况下如何构造？又如何用来识别正则文法的句子？

下面先讨论如何从一般的正则文法构造状态转换图，然后讨论用状态转换图识别正则文法句子的过程。

为正则文法构造状态转换图的步骤如下：

步骤 1　首先引进一个符号，它是不属于所给文法的新符号，为它构造结点，作为开始状态结点；

步骤 2　为所有非终结符号构造相应的状态结点；

步骤 3　对于形如 U::=T 的每个重写规则，作一条从开始状态结点 S 到状态结点 U 的弧，弧上标记为 T。对于形如 U::=WT 的每个重写规则，作一条从状态结点 W 到状态结点 U 的弧，弧上标记为 T；

步骤 4　以识别符号对应的状态作为终止状态。

【例 3.9】　假定有正则文法 G[K]：

```
K::=If|Se   I::=i   S::=Ls   L::=El   E::=e
```

对照上述步骤，可以为该正则文法构造如图 3-7 所示的状态转换图。因为 S 是文法的符号，在英文字母表中从 A 开始选择不是文法符号的符号，因此选择字母 A 作为开始状态名。

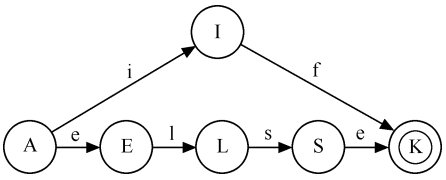
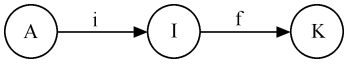


图 3-7

从开始状态 A 出发，到终止状态 K 有两条路径，即，A→I→K 与 A→E→L→S→K。这两条路径上的标记形成的标记序列分别是 if 与 else。识别过程可解释如下。

从开始状态 A 出发，以 A 为当前状态，当输入字符是 i 时，则沿着以字符 i 为标记的弧行进，达到状态 I，以此 I 为当前状态，输入字符是 f 时，沿着以 f 为标记的弧行进，到达下一状态 K，这状态 K 是终止状态，因此识别出关键字 if。如果第一个输入字符是 e，则将沿着以 e 为标记的弧，从开始状态 A 行进达到下一状态 E，以状态 E 为当前状态，依下一输入字符 l 而继续沿以 l 为标记的弧行进达到状态 L，当输入字符相继为 s 与 e，则将从状态 L 达到状态 S，最终达到终止状态 K，这时识别出关键字 else，这时可图示如下：



或简写成：(A)i→(I)f→(K)。类似地，有，

```
(A)e→(E)l→(L)s→(S)e→(K)
```

可见从开始状态结点出发到达终止状态时，弧序列上的标记序列依次形成的字符串正是被识别的字符串。

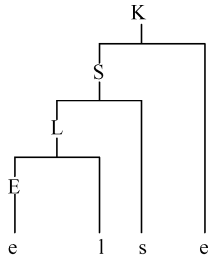
用状态转换图识别正则文法句子的过程称为**运行状态转换图**。

为什么可利用状态转换图识别正则文法的句子？

实质上，运行状态转换图的过程，就是对输入字符串进行归约的过程，或说是进行自底向上分析的过程。再看上述构造步骤，对于重写规则 U::=T，将有从开始状态到状态 U 的弧，弧上标记为 T。当开始状态为当前状态，而当前输入字符是 T 时，将从开始状态转换（行进）到状态 U，这实质上是把 T 直接归约为 U。而对于规则 U::=WT，从状态结点 W 到状态结点 U 有一弧，弧上标记为 T，当前状态为 W，当前输入字符是 T 时，就将从状态 W 转换到状态 U，这实质上是把 WT 直接归约到 U。下面以 else 的识别为例，用图表示，如图 3-8 所示。

步骤	当前状态	输入的其余部分
1	A	else
2	E	lse
3	L	se
4	S	e
5	K	

(a)



(b)

图 3-8

对照关于 else 的推导（归约）就十分明显了：

```
K=>Se=>Lse=>Else=>else
```

设输入字符串是 w，概括起来，一般的识别过程如下：

步骤 1　从开始状态出发，以它为当前状态，并从输入字符串 w 的最左第一个字符开始，重复步骤 2 直到达到输入字符串 w 的右端为止；

步骤 2　扫描 w 的下一字符（当前字符），在当前状态中射出的各个弧中，找出以该字符为标记的弧，沿此弧行进，以所达到的状态作为当前状态。

这时存在两种可能：一种可能是重复步骤 2 到某次时，找不到一个弧，它的标记与当前输入字符相同，这时无法再行进下去，因而达不到输入字符串 w 的右端，因此 w 不是相应正则文法的句子；另一种可能是每次重复步骤 2 时，都能找到一个弧，它的标记与当前字符相同，因此能达到输入字符串 w 的右端，这时，从开始到结束，整个弧序列上的标记依次连成的字符串正是要识别的输入字符串。因此要识别出是相应正则文法的句子，则最终达到的当前状态必须是终止状态。

当识别一个输入字符串 w 时，如果能从状态转换图的开始状态出发，沿着以 w 为标记序列的弧序列行进达到 w 的右端，则 w 是句子的充分必要条件是：最终达到的状态是终止状态。

从推导（归约）的角度，运行状态转换图的步骤可概括如下：首先把输入字符串的第一个字符进行直接归约，所归约成的非终结符号就是：以开始状态为当前状态，沿着第一个输入字符为标记的弧行进，所达到的状态相应的非终结符号，以后每一步，总是把当前状态对应的非终结符号及当前字符直接归约为转换到的状态相应的非终结符号，直到达到终止状态，也即归约到识别符号。

（2）确定有穷状态自动机 DFA

状态转换图是一种有向图，这种图形表示当然是一种非形式的表示，不便于由计算机自动处理，尤其是不利于词法分析程序的自动生成。状态转换图往往对应于所谓的有穷状态自动机，缩写为 FA。这里讨论 DFA，即确定有穷状态自动机的概念。

一个状态转换图，不难看出有如下 5 部分：状态集、字母表、状态转换、开始状态与终止状态（集）。对此 5 个部分进行抽象，便得到 DFA 的概念。

确定有穷状态自动机 DFA 是五元组（K,Σ,M,S,F），其中，

K 是非空有穷的状态集合；

Σ 是非空有穷的输入字母表；

M 是从 K×Σ→K 的映像；

S 是开始状态，S∈K；

F 是非空的终止状态集，F⊆K。

M 表示一个状态与一个输入字母表元素将映像到一个状态，具体说，如果 M(R,T)=Q，则表

示当前状态是 R 时，将在当前输入字符 T 下，映像到状态 Q，即 Q 成为下一当前状态。

【例 3.10】　从图 3-7 中的状态转换图可构造确定有穷状态自动机如下：

DFA D=({A,I,K}, {i,f,e,l,s,e},M,A,{K})

其中，M：M(A,i)=I　M(I,f)=K

M(A,e)=E　M(E,l)=L　M(L,s)=S　M(S,e)=K

从正则文法出发，构造状态转换图或有穷状态自动机时，识别符号对应于终止状态，因此终止状态仅一个。但一般情况下，终止状态可以不止一个，也即定义中的 F 是终止状态集。因此，即使终止状态仅一个，FA 中的 F 也必须写作{K}。

【例 3.11】　设有正则文法 G3.2[Z]：

```
Z::=Aa|Bb    A::=a|Ba    B::=Ab|b
```

可为其构造状态转换图如图 3-9 所示。

相应的 DFA　D=(K,{a,b},M,S,{Z})，其中：

```
K={S,A,B,Z}
M:  M(S, a)=A   M(S, b)=B
    M(A, a)=Z   M(A, b)=B
    M(B, a)=A   M(B, b)=Z
```

类似于运行状态转换图，也可以运行 DFA。假定有输入字符串 ababaa，关于 ababaa 运行上述 DFA　D，从开始状态 S 出发，对 ababaa 不断进行状态转换的步骤可写出如下：

```
M(S, ababaa)=M(M(S, a), babaa)=M(M(A, b), abaa)=M(M(B, a),baa)
            =M(M(A, b), aa)=M(M(B, a),a)=M(A, a)=Z
```

显然这一映像的过程，与运行状态转换图来识别句子的过程是一样的。因为 Z 是终止状态，识别出输入字符串 ababaa 是正则文法 G3.2[Z]的句子。

在一般情况下，若对于某 DFA　D，S 是开始状态，t 是输入字符串，有 M(S, t)=R, R∈F，则称输入字符串 t 可为该 DFA 所接受。

由于映像 M 现在是对输入字符串进行映像，实际上对定义中的映像 M 进行了扩充，即从 K×Σ→K 扩充成了 K×Σ^*→K。

（3）不确定有穷状态自动机 NFA

为引进不确定有穷状态自动机，先看一个例子。

【例 3.12】　设有正则文法 G3.3[Z]：

```
Z::=a|Ia|I0|0|N0
I::=a|Ia|I0        N::=0|N0
```

可为它构造状态转换图如图 3-10 所示。

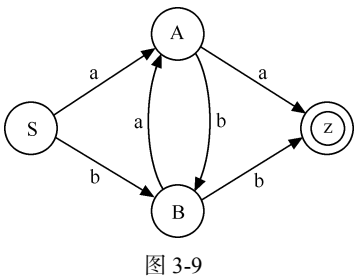


图 3-9

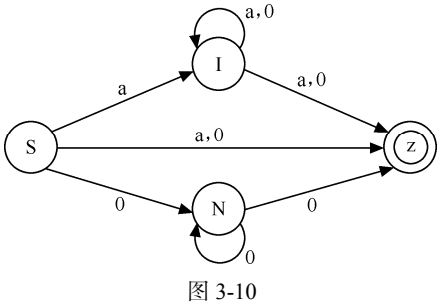


图 3-10

假定输入字符串为 a0a00，关于它运行该状态转换图时将发现：如果当前状态是 S 或 I 时，输入字

符是 a 或 0，不能确定下一当前状态是哪一个，因为从这两个状态结点都有相同标记的弧射出，必须进行人工干预，选择合适的状态作为下一个当前状态。例如，S 为当前状态时，输入字符为 a，可能的下一状态有 I 与 Z，选择状态 I。I 为当前状态时，输入字符为 0，可能的下一状态有 I 与 Z，这时依然选择状态 I 作为下一当前状态。如此继续，最终识别出输入字符串 a0a00 是上述正则文法 G3.3[Z]的句子。

这种状态转换不唯一的状态转换图，所对应的有穷状态自动机称为**不确定有穷状态自动机**，通常定义如下。

不确定有穷状态自动机 NFA　N=(K,Σ,M,S,F)，其中：

K：非空有穷的状态集；

Σ：非空有穷的输入字母表；

M：从 K×Σ 到 K 的子集所组成集合的映像；

S：开始状态集，S⊆K；

F：非空的终止状态集，F⊆K。

注意，这里的 S 是开始状态集，也就是说，开始状态可以不止 1 个。

【例 3.13】　构造上述文法 G3.3 [Z]的 FA。

从状态转换图看，状态转换不唯一，是因为有若干相同标记的弧从同一个状态结点射出。这种情况发生的原因是因为正则文法中，存在有两个或更多个重写规则的右部相同。因此，同一个输入字符串，可以按若干个不同重写规则，直接归约到不同的非终结符号。可见，要为某个正则文法构造 FA，可以首先考察重写规则有无相同的右部，若存在相同的右部，便是 NFA。从正则文法 G3.3[Z]看，多个重写规则有相同的右部，因此必定是 NFA，可构造如下。

```
NFA  N=({S,I,N,Z}, {a, 0}, M, {S}, {Z})
```

其中，M：　M(S, a)={I, Z}　　M(S, 0)={N, Z}

M(I, a)={I, Z}　　M(I, 0)={ I, Z}

M(N, a)={　}　　M(N,0)={N, Z}

M(Z, a)={　}　　M(Z,0)={　}

NFA 与 DFA 的区别在于两方面：一是开始状态，DFA 为唯一，而 NFA 不唯一。当然，从正则文法出发构造相应的 FA，即使是 NFA，开始状态必定也唯一，但也必须使用集合表示，即{S}；二是映像，DFA 是单值的映像，即 K×Σ 到 K，从当前状态转换到的下一当前状态是唯一的，而 NFA 是多值的映像，即 K×Σ 到 K 的子集所组成的集合的映像，从当前状态转换到的下一当前状态不唯一。对于 Q∈K，T∈Σ，一般地可以有：

$M(Q, T)=\{Q_1, Q_2, \cdots, Q_m\}$　$(Q_i \in K, i=1,2,\ldots,m)$

从书写形式上看，即使 R 发出的弧中无标记为 T，也必须写出 M(R,T)={ }，而 DFA 中不必写出。

类似于 DFA，也可以通过运行 NFA 来识别一个输入字符串是否是相应正则文法的句子。但由于可能 $M(R,T)=\{Q_1,Q_2,\cdots,Q_m\}$，$M(R,Tt)=M(M(R,T),t)=M(Q_1,t)\cup M(Q_2,t)\cup\ldots\cup M(Q_m,t)$，其中 $T\in\Sigma$，$t\in\Sigma^*$。

关于输入字符串 a0a00，运行上述 NFA 的过程如表 3.6 所示。

表 3.6

步　骤	当 前 状 态	输入的其余部分	可能的后继	选　择
1	S	a0a00	I, Z	I
2	I	0a00	I, Z	I
3	I	a00	I, Z	I
4	I	00	I, Z	I
5	I	0	I, Z	Z

通常用表 3.6 所示的形式描述运行 NFA 的过程。

【例 3.14】　运行 NFA 的例子。

设有文法 G3.4[Z]：

Z::=Z0|T1|0|1　　T::=Z0|0

由于存在有右部相同的规则，相应的 FA 为 NFA，可构造如下。

NFA　N=({S,T,Z}, {0,1}, M, {S},{Z})

其中，M：　　M(S, 0)={T, Z}　　M(S, 1)={Z}

M(T, 0)={　}　　M(T, 1)={Z}

M(Z, 0)={T, Z}　　M(Z, 1)={　}

假定输入字符串是 1010010，运行上述 NFA　N 的过程如表 3.7 所示。

表 3.7

步　骤	当 前 状 态	输入的其余部分	可能的选择	选　择
1	S	1010010	Z	Z
2	Z	010010	T,Z	T
3	T	10010	Z	Z
4	Z	0010	T,Z	Z
5	Z	010	T,Z	T
6	T	10	Z	Z
7	Z	0	T,Z	Z

从表 3.7 可见，从开始状态 S 出发，对输入字符串 1010010 逐个字符地进行状态转换，最终达到输入字符串右端，且达到的状态 Z 是终止状态，因此，1010010 可为该 NFA 所接受，也即是相应文法 G3.4[Z]的句子。

上述运行过程中，如果当前状态是 Z，输入字符是 0 时，可能的后继有 T 与 Z 两个状态，在步骤 2 时选择 T，而在步骤 4 时选择 Z。如果不这样选择，显然将导致不同的结论。而之所以能进行正确地选择，是因为人为的干预。如果没有人为的干预，尽管知道当前状态与当前输入字符，相当于知道直接归约的简单短语,但不能确定直接归约到对应于下一当前状态的哪个非终结符号。解决这个困难的一个办法便是确定化，把 NFA 确定化为等价的 DFA。

（4）NFA 确定化

NFA 区别于 DFA 主要是两方面：一是开始状态不唯一；二是映像（状态转换）不唯一。与正则文法相联系，主要是映像的不唯一。换句话说，对于 NFA，总是存在一些状态，对于它们有如下的映像：

$M(R, T)=\{Q_1, Q_2, \cdots, Q_m\}$

即当前状态为 R 而输入字符为 T 时，将转换到的状态是 Q_1，也可能是状态 Q_2，…，还可能是 Q_m。为了使 NFA 成为等价的 DFA，把状态集合$\{Q_1, Q_2, \cdots, Q_m\}$就看成一个状态，用$[Q_1, Q_2, \cdots, Q_m]$表示这单个状态，或者说，这单个状态$[Q_1, Q_2, \cdots, Q_m]$代表所有可能状态中的一个状态，所转换到的状态总是在其中。在这个思想指导下，形成了从 NFA 构造等价的 DFA 的思路。

设有 NFA　N=(K,Σ,M,S,F)，则相应的 DFA　N′构造如下：

DFA N′=(K',Σ,M',S',F')

其中：

K′是非空有穷的状态集，其每个状态形如$[Q_i,Q_j,\cdots,Q_k]$，各个 Q∈K；

M′如下定义：若 $M(R_1,T)\cup M(R_2,T)\cup\cdots\cup M(R_i,T)=\{Q_1,Q_2,\cdots,Q_j\}$，则 $M'([R_1,R_2,\cdots,R_i], T)=$

$[Q_1,Q_2,\cdots,Q_j]$。若 $M(R_1,T)\cup M(R_2,T)\cup\cdots\cup M(R_i,T)$缩写为 $M(\{R_1,R_2,\cdots,R_i\},T)$，则可以写出：

若 $M(\{R_1,R_2,\cdots,R_i\},T)=\{Q_1,Q_2,\cdots,Q_j\}$，则 $M([R_1,R_2,\cdots,R_i],T)=[Q_1,Q_2,\cdots,Q_j]$；

$S'=[S_1，S_2，\cdots，S_n]$，　这里 $S=\{S_1,S_2,\cdots,S_n\}$；

$F'=\{[Q_i,Q_j,\cdots,Q_k]\mid[Q_i,Q_j,\cdots,Q_k]\in K'$，且$[Q_i,Q_j,\cdots,Q_k]\cap F\neq\varnothing\}$；

例如，由 $M(T,0)=\{\ \}$，$M(Z,0)=\{T,Z\}$，有 $M(T,0)\cup M(Z,0)=\{\ \}\cup\{T,Z\}=\{T,Z\}$，或表示为：$M(\{T,Z\},0)=\{T,Z\}$，因此，$M([T,Z],0)=[T,Z]$。

请注意，新状态名必须按某种规范顺序写出，例如，按字母表顺序写出，对于{T, Z}将写出[TZ]，而不应写成[ZT]。

$F=\{Z\}$，$\{T,Z\}\cap F=\{Z\}\neq\varnothing$，所以，$[T,Z]\in F'$。

为了此 DFA 中的一切状态都是有效的，即能从开始状态出发，沿着一条弧序列行进达到它，且能从它出发，沿着一条弧序列行进达到终止状态，用列表方式，从开始状态 S′开始来构造各个新状态。下面以例 3.14 中的 NFA 确定化为例进行说明，这时可构造如表 3.8 所示。

表 3.8

转换为 状态 \ 输入	0	1
[S]	[TZ]	[Z]
[TZ]	{TZ}	[Z]
[Z]	[TZ]	

因此，DFA N'=({[S], [Z], [TZ]}, {0,1}, M', [S], {[Z], [TZ]})

其中，M'：M'([S], 0)=[TZ]　　M'([S], 1)=[Z]

M'([TZ], 0)=[TZ]　　M'([TZ], 1)=[Z]

M'([Z], 0)=[TZ]

读者可自行对于输入字符串 1010010 运行该 DFA　N'，显然，可为该 DFA 所接受，也即是相应文法的句子。

说明：如果不按上述列表法进行 NFA 的确定化，则将需要关于 K 的幂集来构造 K'，这样，K'中将包含 2^n-1 个状态，这里 n 是 K 中的状态数。例如，例 3.14 中 NFA N 的状态数=3，将有：

K'={[S],[T],[Z],[ST],[SZ],[TZ],[STZ]}

且，M'：M'([S], 0)=[T,Z]　　M'([S], 1)=[Z]

M'([T], 1)=[Z]

M'([Z], 0)=[T,Z]

M'([ST], 0)=[T,Z]　　M'([ST], 1)=[Z]

M'([SZ], 0)=[T,Z]　　M'([SZ], 1)=[Z]

M'([TZ], 0)=[T,Z]　　M'([TZ], 1)=[Z]

M'([STZ], 0)=[T,Z]　　M'([STZ],1)=[Z]

如果画出 DFA　N'的状态转换图，将发现状态[T]、[ST]、[SZ]与[STZ]这 4 个状态都是不能从开始状态[S]出发，沿任何弧序列行进达到的状态，因此应从 K'中删除这些状态，最终所得的正与前面所写的相同。一般情况下，如果按 K 的幂集来构造 K'，K 中状态数为 n，则 K'中的状态数为 2^n-1。对于上例，n=3，K'中的状态数为 7，而实际上仅为 3。用列表法对 NFA 确定化，显示出很大的优

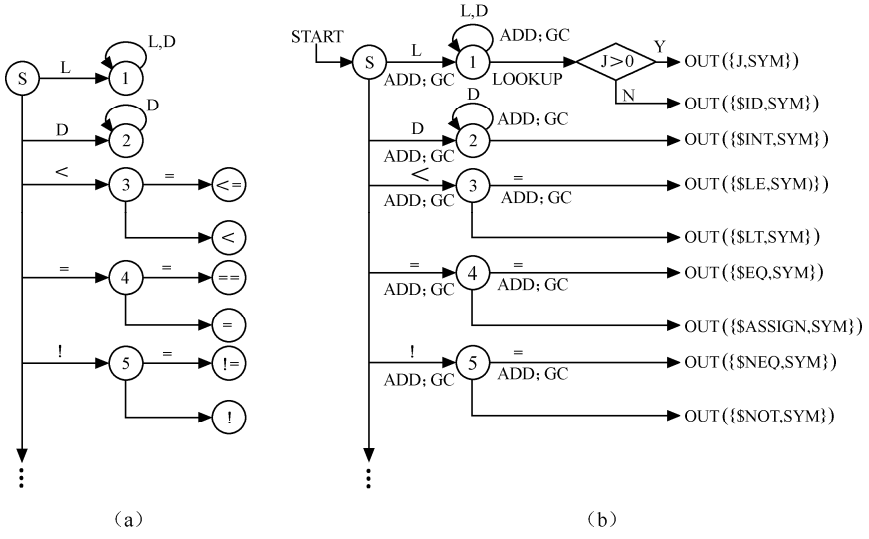
点，它既简便，又能保证正确。

表 3.8 显示了各个状态在输入字符下的转换，因此往往称为**状态转换矩阵**。显然状态转换矩阵很容易用 C 语言二维数组来实现。

（5）状态转换图与程序实现

前面讨论了可以从正则表达式引导出状态转换图或有穷状态自动机，而运行状态转换图可以识别正则文法的句子，也就是说，可以识别一个程序设计语言的各个符号。运行状态转换图只是概念上的，实际上，很容易把状态转换图与程序对应起来。

假定有状态转换图如图 3-11 所示。



（a）　　（b）

图 3-11

图 3-11（a）中的状态转换图，对前面所定义的有所扩充，也即两个状态结点之间的弧上的标记，不仅可以是字母表上的字符，还可以是一个字符类，即字母类 L 代表任一字母字符，数字类 D 代表任一数字字符。还有的弧上不指明任何标记，代表“其他字符”。每次运行状态转换图达到终止状态时，标志识别出一个符号。除了终止状态，在其他各状态时都隐指对读入的当前输入字符进行判别，即判别所读入的输入字符是否该标记所指明的字符。当加入语义动作，例如取字符、把字符拼入正取到的部分符号中等，并考虑到关键字的处理时，可把图 3-11（a）中的状态转换图扩充成为图 3-11（b）所示的状态转换图。因此一个状态到另一个状态的转换将可以对应到一个 if 语句，显然，一个结点自身到自身的状态转换，可对应到一个循环结构。例如，关于图 3-11（b）中的状态转换图，可对应于如下的 C 型语言程序：

```
START: ch=getchar( );
if(isletter(ch))
{  sym=sym || ch;  ch=getchar( );
   STATEI: while(isletter(ch) || isdigit(ch))
           {  sym=sym || ch;  ch=getchar( ); }
           J=lookup (sym);
           if (J>0) OUT ({J, sym});
           else OUT ({$ID, sym});
}
else if (isdigit(ch))
{  sym=sym || ch;  ch=getchar( );
```

```
    STATEN: while (isdigit (ch))
            {  sym=sym || ch;  ch=getchar( ); }
            OUT ({$INT, sym});
}
else if (ch=='<')
{  sym=sym || ch;  ch=getchar( );
   if(ch=='=')
    {  sym=sym || ch;  ch=getchar( );  OUT({$LE, sym}); }
    else OUT ({$LT, sym});
}
else if (ch=='=')
{  sym=sym || ch;  ch=getchar( );
   if (ch=='=')
    {  sym=sym || ch;  ch=getchar( );  OUT({$EQ, sym}); }
   else  OUT ({$AGGIGN, sym});
}
else if (ch=='!')
{  sym=sym || ch;  ch=getchar( );
   if(ch=='=')
    {  sym=sym || ch;  ch=getchar( );  OUT({$NEQ, sym}); }
   else  OUT ({$NOT, sym});
}
else …
```

综上所述，利用正则表达式来描述程序设计语言的符号，可以增强语言的符号说明书的灵活表达能力，通过正则表达式到状态转换图的转换，再到程序的实现，可以实现词法分析程序的自动生成。

本章小结

本章讨论编译程序的第一阶段：词法分析，内容包括概况与词法分析程序的实现。

词法分析程序的基本功能是：读入源程序字符序列，识别开源程序中的各个符号，把它们处理成相应的属性字。词法分析程序的手工实现，要点是属性字的设计与标识符的处理。标识符的处理主要涉及3个方面，即区分标识符的定义性出现与使用性出现、确定标识符的作用域以及确定属性字的符号值。词法分析程序总控程序很容易实现，重点是取符号子程序与相关表格的数据结构的设计。

本章还讨论了词法分析程序的自动生成，其基本思想是：词法分析程序由通用扫描算法基于各类表完成词法分析，自动生成词法分析程序，也就是自动生成各个表。为了生成程序设计语言各自的表，给出特定程序设计语言的符号说明书，其中指明实现词法分析所需各个表所涉及的信息，例如一切字符、符号及符号的机内表示等。为了描述符号，引进正则表达式的概念。从正则表达式转换到专门为识别正则文法句子而设计的状态转换图或相应的有穷状态自动机，再到程序的实现。正则表达式形象地描述了符号的组成。一类符号，只要能用正则表达式描述，词法分析程序就能识别它。正则表达式的引进大大改善了描述符号的灵活性和多样性。事实上，正则表达式在模式匹配和模式识别等方面，有着广泛的应用。建议读者关注这部分内容。

词法分析阶段的基础文法是正则文法。

相关概念有状态转换矩阵、NFA 的确定化。

复习思考题

1．为什么词法分析基于正则文法，而不是基于上下文无关文法?

2．如何设计属性字，使得更有利于下阶段语法分析工作?

3．词法分析时，对标识符的处理包括哪些方面?

4．试简述词法分析程序自动生成的基本思想。

5．为什么说正则表达式是描述符号组成的好工具?

习　　题

1．试为下列函数定义写出相应的属性字序列，编码参照表 3.1。

```
int max2(int a, int b)
{ int max;
   if(a>b) max=a; else max=b;
   return max;
}
```

2．试为下列函数定义写出相应的改进的属性字序列，编码参照表 3.1。

```
int fun(int a, float b)
{ int m=4;  float x=5;
  return (a+m)*(x-b)-m;
}
```

3．设有关键字表由赋初值方式给出 (参看 3.2.4 节)。试写出关于某个关键字查找关键字表的 C 语言程序。要求：所查关键字从键盘输入，查到时输出关键字表序号及相应属性字，所写程序可在计算机上运行。

4．C 语言程序中可包含注解，它从符号“/*”开始，结束于符号“*/”，之间的部分都属于注解。试写出删除注解的程序段。

5．试写出实数的正则表达式描述，其中既有小数点，又有用 e 表示的指数部分。

6．试画出下列文法 G[Z]的状态转换图。

G[Z]：　Z::=Zb|Aa|Bb　　A::=Ab|a　　B::=Ba|b

7．试为文法 G[Z]构造相应的 FA。

G[Z]：　Z::=Zb|Aa|Bb　　A::=Ab|a　　B::=Ba|b

8．试为文法 G[Z]：

Z::= Z0|Z1|V0|U1　　M::= M0|M1|0|1

U::= M1|1　　V::= M0|0

构造 FA。提示：先判别是 DFA，还是 NFA。

9．试把下列 NFA 确定化：

NFA N=({q1,q2,q3},{a,b},M, {q1},{q3})

其中，M：　M(q1,a)={ q1,q2 }　　M(q1,b)={ q1 }

M(q2,a)={ q2,q3 }　　M(q2,b)={ q2 }

M(q3,a)={ q1,q3 }　　M(q3,b)={　}

第4章 语法分析——自顶向下分析技术

本章导读

词法分析之后，对源程序的翻译便进入语法分析阶段。语法分析由语法分析程序（或称识别程序）完成。分析技术分自顶向下与自底向上两大类。本章讨论自顶向下分析技术，内容包括：概况及两类无回溯的自顶向下分析技术，即递归下降分析技术与预测分析技术。请注意无回溯自顶向下分析技术的应用条件：无左递归性与无回溯性。读者应熟练掌握消去左递归的文法等价变换。关于分析技术，重点关注递归下降识别程序的构造，熟练掌握预测识别程序的句型分析。预测识别程序在预测分析表和分析栈的配合下完成句型分析，预测分析表又称 LL 分析表，理解预测分析表的构造思路，将能更深刻理解预测分析技术的实现思想。本章最后讨论了预测识别程序句型分析的计算机实现。请注意概念与实践上的差异，特别是实践上的方法与技巧。

4.1 自顶向下分析技术概况

当编译程序对一个源程序进行了词法分析之后，便进入语法分析阶段，也就是说，在识别开源程序中的各个符号之后，将继续识别这些连续的符号构成什么语法成分，最终识别出在语法上正确的程序。编译程序中进行语法分析的部分称为**语法分析程序**。语法分析程序在识别过程中，将生成相应的内部中间表示，为编译程序的下一阶段语义分析工作作好准备。这内部中间表示可以是语法分析树，也可以是推导或其他形式的中间表示。当存在错误时，语法分析部分将给出报错信息。

语法分析树的构造存在两种方式：自顶向下方式与自底向上方式，因此语法分析技术相应地也分两大类，即自顶向下分析技术与自底向上分析技术，本章重点讨论自顶向下分析技术。

4.1.1 讨论前提

语法分析阶段是以词法分析阶段的输出作为输入，也就是说，语法分析程序的输入是内部中间表示形式的符号序列，即属性字序列。每个属性字等长且定长，其中包含了最基本的信息，如是特定符号还是非特定符号，对于运算符可能还包含优先级信息等。非终结符号对应于语法实体（成分），然而，对于标识符与整数这些在词法分析时是非终结符号的符号，由于不再考虑标识符是怎样由字母和数字组成的，也不再考虑整数是怎样由数字组成的，而是作为一个不可再分割的整体，在语法分析时，标识符与整数都作为终结符号处理。对于符号的表示法，与字符的表示法

自然是不同的。例如，加号‘+’作为字符，表示为'+'，然而作为符号，则仅写作+。请读者注意这种表示法上的区别。只是为讨论简便起见，并不考虑符号的内部中间表示形式，还是以符号形式作为讨论对象，显然这样并不失一般性。

词法符号通常以正则文法形式的规则来描述，因此词法分析以正则文法为基础。由于一般的语法成分不可能由正则文法定义，语法分析将基于上下文无关文法进行，这是由程序设计语言本身的特征决定的。例如，"if(a>b)max=a; else max=b; "之类的条件语句，不可能由正则文法描述。由小括号“(”与“)”括住的表达式结构，同样不能由正则文法形式的规则来定义。事实上，语法分析树和推导等概念本身就是建立在上下文无关文法基础上的。

通常源程序是从左到右、从上到下地书写阅读的，语法分析自然也是从左到右、从上到下逐个符号地进行。也就是说，语法分析是自输入符号串的左边第一个符号开始，从左到右逐个符号地读入和进行分析。

概括起来，自顶向下分析技术的讨论前提有 3 条：即输入是符号序列，基于上下文无关文法进行语法分析，分析过程从左到右地进行。

4.1.2　自顶向下分析技术要解决的基本问题

语法分析程序的输入是中间表示形式的符号序列，即属性字序列。按自顶向下分析技术进行语法分析时，从推导的角度看，是从识别符号出发，试图建立一个推导，使推导得到的符号串与输入符号串相同。在推导过程中，每一步直接推导产生的句型中，有一个非终结符号，它被展开（替换）为以它为左部的规则之右部符号串；从语法分析树角度看，是从识别符号出发，以识别符号为根结点，试图从上向下构造语法分析树，使该语法分析树的末端结点符号串正好与输入符号串相同，如果正好相同，识别出输入符号串是相应文法的句子。在这语法分析树的构造过程中，不断从上向下添加分支，进一步说，每个分析步中，在正构造的语法分析树中，对应于被展开（替换）的非终结符号的末端结点处，向下构造一个分支。

作为一种自顶向下分析技术，如同第 2 章所讨论的，必须解决一个基本问题，即分析过程中，按某非终结符号 U 展开时，若关于 U 存在规则 $U::=u_1|u_2|\cdots|u_k$，如何确定用哪个 $u_i(1\leqslant i\leqslant k)$替换 U?

不言而喻，作为分析技术，重要的是如何自动、机械地解决上述基本问题而完成语法分析。

4.1.3　自顶向下分析技术的实现思想与应用条件

1. 自顶向下分析技术的一般实现思想

先考虑最简单最基本的一种自顶向下分析技术，其基本思想是：在自顶向下分析过程中的每一步，将按 U 为目标进行展开，而关于 U 存在规则：

$$U::=u_1|u_2|\cdots|u_k$$

时，首先用 $U::=u_1$ 尝试把 U 展开成 u_1，如果可行则继续，否则依次逐个尝试把 U 展开成 u_2 或 u_3、…、或 u_k，直到某个 u_i $(1\leqslant i\leqslant k)$可行。下面举例说明。

【例 4.1】　设有文法 G4.1[E]：

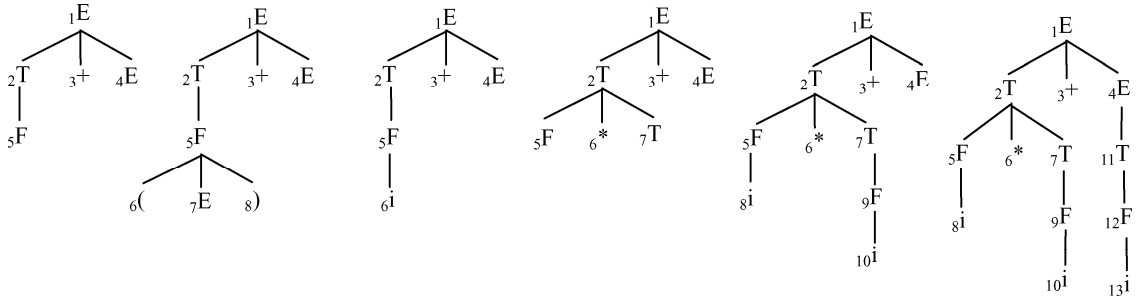
```
E::=T+E|T    T::=F|F*T    F::=(E)|i
```

假定输入符号串为 i*i+i。设置一个输入指针，指向正将被扫描的符号，即最左的第一个输入符号 i，其语法分析树构造步骤如下。

步骤 1　首先以识别符号 E 为根结点，序号为 1。以 E 为目标，试图向下构造分支。这时对以 E 为左部的规则，选取按第一个选择 T+E 展开，并建立分支，分支结点的序号依次是 2、3 与

4，期望其分支结点符号串 T+E 能覆盖当前输入指针所指向处（i）开始的部分输入。

步骤 2　由于分支结点符号串 T+E 中最左边第一个符号 T 是非终结符号，又期望以其为目标进行展开，能覆盖输入指针所指向处开始的部分输入。因此又以 T 为目标，按关于它的规则，逐个选取右部的各个选择进行尝试。首先选取第一个选择 F 作为目标，进行展开并建立分支，仅有分支结点 F，序号为 5，期望该分支结点 F 能覆盖当前输入指针所指向处（i）开始的部分输入，这时如图 4-1（a）所示。同样，由于 F 是非终结符号，又以其为目标，按关于它的规则的第一个选择(E)展开，建立分支，分支结点符号串是(E)，分支结点的序号依次是 6、7 与 8，期望该分支结点符号串(E)能覆盖当前输入指针所指向处(i)开始的部分输入，如图 4-1（b）所示。由于当前输入指针指向的输入符号是 i，而这时最左的分支结点符号是(，不匹配，因此，此第一个选择(E)是错误的，撤销所建立的分支结点序号为 6、7 与 8 的分支，改选第二个选择 i，按此选择展开，建立分支，分支结点只有 i，序号为 6，期望该分支结点 i 能覆盖当前输入指针所指向处（i）开始的部分输入，如图 4-1（c）所示。显然匹配，建立分支成功，输入指针前进到指向下一个输入符号*。



（a）　（b）　（c）　（d）　（e）　（f）

图 4-1

步骤 3　由于以 F 为目标展开成功，表明按规则 E::=T+E 进行展开时，T 已匹配，下一步将让 T+E 中的下一个符号+去与当前输入指针所指向处（*）开始的部分输入匹配。这时不言而喻，+与输入的下一个符号*不匹配，表明 T 的这一选择事实上是错误的，因此，撤销所建立的两个分支（分支结点序号分别为 5 与 6），改选 T 的第二个选择 F*T，建立分支，如图 4-1（d）所示，并把输入指针回退到原先与 T 去匹配的输入符号位置（i）。这时又因 F*T 中最左边第一个符号是 F，如同步骤 2 中所述，先选取第一个选择(E)，然后再改选为第二个选择 i，按此选择展开，建立分支，期望其分支结点符号串 i 能覆盖当前输入指针所指向处（i）的部分输入，这时匹配，建立分支成功。输入指针前进到指向下一个输入符号*。

步骤 4　F 匹配成功，表明以 T 为目标，按规则 T::=F*T 进行展开时，F 已匹配，下一步将让 F*T 中的下一个符号*去与当前输入指针所指向处（*）开始的部分输入匹配。当然能匹配，输入指针前进到指向下一输入符号 i。这时以 F*T 中的第三个符号 T 为目标，如同前面步骤中那样，关于 T 的规则，逐个尝试关于它的规则之右部的各个选择。这时经历关于 T 的选择与关于 F 的各个选择，建立又撤销所建立分支，如同步骤 2 中所述，最后建立了如图 4-1（e）所示的语法分析树，输入指针前进到指向下一输入符号+。

步骤 5　由于分支结点符号串 F*T 中三个结点均已匹配成功，因此以 E 为目标、按规则 E::=T+E 展开时的右部第一个符号 T 已匹配成功，将期望由右部第二个符号+去与输入指针所指向处（+）的输入符号串部分匹配，这时显然匹配，输入指针前进到指向符号 i。将以右部第三个符号 E 为目标，又将如同前述步骤那样试图建立分支，只是这时在选取第一个选择 T+E 时将出错，

而按第二个选择 T 进行展开。最终得到如图 4-1（f）所示的语法分析树。

2. 问题及其解决

从上述语法分析树的构造过程可以看出以下两点。

① 在每一步分析中，当以某个非终结符号 U 为目标，按相应规则展开，构造分支时，总是关于以 U 为左部的规则，从右部第一个选择开始，逐个尝试，试图构造分支。

② 构造过程中，发现当前所选取的选择中，某个符号与输入指针指向的符号不匹配时，立即剪去所作分支，甚至可能发现先前认为是正确地构造的分支实际上也是不正确的，需要把已构造的分支剪去，按规则的下一个选择重新构造分支。

在上述构造过程中，当对于某一个非终结符号的某个选择匹配失败，必须剪去所构造的失败分支，并把输入指针进行回退，回退到试图与当前失败的选择去进行匹配的那些符号的开始位置，然后选取另一个选择再行尝试。当用某个非终结符号的某个选择去进行匹配而失败时，剪去失败的分支，并回头查看输入符号，以便尝试与其他的选择相匹配，这种过程称为**回溯**。因此，这一分析过程是一个面向目标的、试探的、回溯的过程，称为**带回溯的**自顶向下分析技术。

带回溯的自顶向下分析技术存在两大问题，即左递归与低效。

先考虑左递归的问题。假设把文法 G4.1[E]改写如下：

```
E::=E+T |T   T::=F | T*F  F::=(E) | i
```

首先以识别符号 E 为根结点，然后试图以它为目标进行展开。由于每次总是按第一个选择进行展开，不难画出语法分析树如图 4-2 所示。这表明始终按规则 E::=E+T 进行展开，将永无止境，因此这种左递归的存在将使得带回溯的自顶向下分析技术不可行。必要的是进行消去左递归的文法等价变换，这在下一节中讨论。

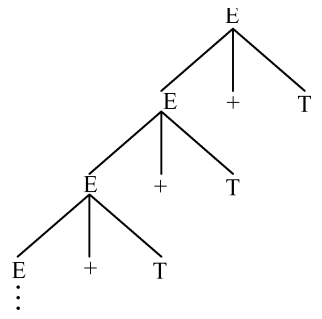


图 4-2

低效问题主要源自回溯。尽管先前已经匹配了，但对后来的不匹配寻根溯源，发现早先的匹配并不正确。这时就得把早先所构造的相应分支都剪去，并把输入指针回退到相应的位置，重新按规则的其他选择进行匹配。这种回溯的次数相当多，大大影响了功效。因此合适的做法是设法排除回溯，应用无回溯的自顶向下分析技术。

从上可见，左递归是应用自顶向下分析技术的最大障碍，必须消除文法的左递归，如果要求功效高，还应该消去回溯。

4.1.4　消去左递归的文法等价变换

文法等价变换是对文法进行的一种变换，使变换后所得文法与原有文法生成的语言相同。消去左递归的文法等价变换，也就是使变换后的文法不再包含左递归的文法等价变换。文法的左递归，可以是规则左递归，也可以是文法左递归。存在规则左递归，表明文法中至少包含一个非终结符号 U，关于它存在形如 U::=U…的规则；存在文法左递归，则表明文法中至少存在一个非终结符号 U，关于它存在 U=>+ U…，这意指存在一列规则：$U::=U_i\cdots$、$U_i::=U_j\cdots$、…、$U_k::=U_l\cdots$、$U_l::=U\cdots$。

一个左递归的文法将导致应用自顶向下分析技术失败，因此在应用自顶向下分析技术时必须首先消去左递归，必须对文法进行消去左递归的文法等价变换。

1. 消去规则左递归的文法等价变换

消去规则左递归的思路，简单地是把文法等价变换为右递归的。例如，对于文法 G4.1[E]，把规则 E::=E+T|T 改为 E::=T+E|T。这种变换从所生成的句子集合看是等价的，也就是生成相同的句

子集合；但从含义上看，还是有一定的区别。对于其他更为一般的文法不是这样换个顺序就能达到目的的。下面讨论更为一般的消去规则左递归的文法等价变换方法，仍以 E::=E+T 为例。

应用规则 E::=E+T，不难得到下列推导：

E=>E+T=>E+T+T=>…=>E+T+T…+T+T=>T+T+T…+T+T

即 E=>+ T+T+T…+T+T。引进新的非终结符号 E'，让 E'=>+ +T+T…+T+T。可引进规则 E::=TE'。由于其中+T 出现的次数可以任意，显然可引进规则 E'::=+TE'。每多应用该规则一次，便多推导出一个+T。所以，

E=>TE'=>T+TE'=>T+T+TE'=>…

为了推导出 T：E=>TE'=>T，显然必须引进规则 E'::=ε。概括起来，由 E::=E+T|T 等价变换成了：

E::=TE'　E'::=+TE'|ε

对于 T::=F|T*F，类似地进行处理。因此文法 G4.1[E]的消去了左递归的等价文法是 G4.1'[E]：

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

一般情况下，设有规则 U::=Ux|y，则由 U=>Ux=>Uxx=>…=>yxx…x，引进 U::=yU'与 U'::=xU'|ε。更一般地，若 U::=Ux_1|Ux_2|…|Ux_m|y_1|y_2|…|y_n，其中每个 y 的首符号不是非终结符号 U，可如下地进行消去规则左递归的文法等价变换。首先提因子，得

U::=U(x_1|x_2|…|x_m)|(y_1|y_2|…|y_n)

然后等价变换成：

U::=(y_1|y_2|…|y_n)U'　U'::=(x_1|x_2|…|x_m)U'|ε

注意，一般总是让ε是最后的选择。

【例 4.2】 设有关于数组元素的下标表达式的重写规则如下：

<下标表达式>::=<下标表达式>,<表达式>|<表达式>

这是规则左递归，x=,<表达式>，y=<表达式>，因此，等价变换成：

<下标表达式>::=<表达式><下标表达式'>

<下标表达式'>::=,<表达式><下标表达式'>|ε

2. 消去文法左递归的文法等价变换

当文法中仅存在规则左递归时，可采用前面所讨论的、变换为规则右递归的文法等价变换，消去文法中的规则左递归。但对于文法左递归，这无能为力。一般情况下难以消去文法左递归，仅在某些特殊情况下，存在消去文法左递归的算法。下面给出的算法仅当文法不包含回路（即形如 U=>+ U 的推导），也不包含ε规则时可实现文法左递归的消去。

首先说明消去文法左递归的思路。前面已提到，包含文法左递归的原因是因为存在一组规则：U::=U_i…、U_i::=U_j…、…、U_k::=U_l…与 U_l::=U…，从而有 U=>+ U…。现在对一切非终结符号排序：U_1、U_2、…、U_n，经等价变换后使得一切非终结符号为右部首个符号的规则都形如：U_i::=U_j… (j>i)，因此，使得 U_{i_1}=>U_{i_2}…=>…=>U_{i_m} …=>… ，最终推导所得的首个非终结符号，因为 i_1<i_2<…<i_m<…，决不会是 U_{i_1}，因而不再有文法左递归。具体消去文法左递归的等价变换算法步骤如下。

步骤 1　把文法的一切非终结符号以某种顺序排序成 U_1、U_2、…、U_n。

步骤 2　以步骤 1 中的顺序执行下列程序（C 型语言）。

```
for(i=1; i<=n; i++)
{   for(j=1; j<=i—1; j++)
    {  把形如 Ui::=Ujr 的规则改写成 Ui::=xj1r|xj2r|…|xjkr;
       其中 Uj::=xj1|xj2|…|xjk 是对于 Uj 的一切规则;
```

```
    }
    消去关于 Ui 的规则左递归;
}
```

步骤 3　从步骤 2 得到的文法中，删除无用规则，给出最终的消去了文法左递归的等价文法。

当关于某个实例，应用上述消去文法左递归的算法时，显然所得文法中若有形如 $U_i::=U_j\cdots$ 的规则，U_i、$U_j\in V_N$ 时，必定 j>i。具体说，对于 $U_1::=U_k\cdots$，必定 k>1，只能是 k=2,3,…；对于 $U_2::=U_k\cdots$，必定 k>2，只能是 k=3,4,…；对于 $U_{n-1}::=U_k\cdots$，必定是 k>n-1，只能是 k=n；而对于 $U_n::=X\cdots$，X 不能是非终结符号，只能是 $X\in V_T$。

说明　进行了消去文法左递归的等价变换后，文法中也不再包含规则左递归。

下面以具有一般性的例子来说明上述算法的应用。

【例 4.3】 设有文法 G4.2[Z]：

```
Z::=Za|Sbc|dS    S::=Zef|gSh
```

试消去其文法左递归。

该文法关于 Z 是规则左递归，但还关于 Z 与 S 是文法左递归，因此应用消去文法左递归的算法。

步骤 1　对文法非终结符号排序：U_1=Z，U_2=S（n=2）。

步骤 2　执行循环：

i=1, j=1：j>i-1，不执行 j 循环，但关于 U_1=Z 存在规则左递归，对 U_1 进行消去规则左递归的等价变换，改写成右递归形式如下：

```
Z::=(Sbc | dS)Z'    Z'::=aZ' | ε
```

i=2, j=1：有规则 S::=Zef，形如 $U_2::=U_1r$，且存在规则 Z::=(Sbc|dS)Z'，形如 $U_1::=x_{11}$，改写成 $U_2::=x_{11}r$ 的形式，连同 S::=gSh，有

```
S::= (Sbc|dS)Z'ef | gSh
```

经整理后有：

```
S::=SbcZ'ef | dSZ'ef | gSh
```

i=2, j=2：j>i-1，关于 j 的循环结束，消去关于 U_2=S 的规则左递归，应用规则左递归消去的文法等价变换，改写成右递归形式：

```
S::= ( dSZ'ef | gSh)S'      S'::= bcZ'efS' | ε
```

步骤 3　最后得到消去了左递归的等价文法 G4.2'[Z]：

```
Z::= (Sbc | dS)Z'          Z'::=aZ' | ε
S::= ( dSZ'ef | gSh)S'     S'::= bcZ'efS' | ε
```

如果改变非终结符号的排序顺序，例如，U_1=S，U_2=Z，则所得到的消去了左递归的文法在形式上会有所不同，但它们都是等价的，即生成的语言是相同的，因此如何排序无关紧要。

【例 4.4】　设有文法 G4.3[Z]：

```
Z::=Sa | Tb | cZ    S::=Tde | Zf | Sg    T::=Sh | jTk
```

试消去文法左递归。

显然该文法关于非终结符号 Z、S 与 T 为文法左递归，应用消去文法左递归的算法。

步骤 1　对文法非终结符号排序：U_1=Z、U_2=S、U_3=T。

步骤 2　执行循环：

i=1, j=1：j>i-1，不执行关于 j 的循环，且关于 U_1=Z 不存在规则左递归，不进行消去关于 Z

的规则左递归的等价变换。

i=2, j=1：有规则 S::=Zf，形如 U_2::=U_1r，且 Z::=Sa | Tb |cZ，形如 U_1::=x_{11} | x_{12} | x_{13}，连同 S::=Tde | Sg 有

```
S::=( Sa | Tb | cZ) f | Tde | Sg 或 S::=S(af|g) |T(bf | de)| cZf
```

i=2, j=2：j>i-1，关于 j 的循环结束，对 U_2=S 消去规则左递归，得到

```
S::=( T(bf | de)| cZf) S'    S'::=(af|g)S' | ε
```

i=3, j=1：无形如 U_3::=U_1r 的规则，不进行代入。

i=3, j=2：有规则 T::=Sh，形如 U_3::=U_2r，且规则 S::=(T(bf | de)| cZf) S'，形如 U_2::=x_{21}，进行代入，连同 T::=jTk 有

```
T::=( T(bf | de)| cZf)S'h | jTk
```

或整理得：T::=T(bf | de)S'h | cZfS'h | jTk。

i=3, j=3：j>i-1，关于 j 的循环结束，对 U_3=T 消去规则左递归，得到

```
T::=( cZfS'h | jTk)T'     T'::=(bf | de)S'hT' | ε
```

步骤 3　最后得到消去了左递归的等价文法 G4.3'[Z]如下：

```
Z::=Sa | Tb |cZ
S::=( T(bf | de)| cZf) S'    S'::=(af|g)S' | ε
T::=( cZfS'h | jTk)T'        T'::=(bf | de)S'hT' | ε
```

从上面两例可以看到：

- 必须在关于 j 的循环结束时才进行关于 U_i 的规则左递归的消去，否则将引起错误；
- 在关于 j 的循环结束时便消去关于 U_i 的规则左递归，因此在消去文法左递归之后，也不存在规则左递归，不必要另行进行消去规则左递归的文法等价变换。

概括起来，消去左递归的文法等价变换的规范步骤如下：

步骤 1　判别是规则左递归还是文法左递归，确定相应的算法；

步骤 2　按照所确定算法进行消去左递归的文法等价变换；

步骤 3　给出最终的消去了左递归的等价文法。

3. 实现的考虑

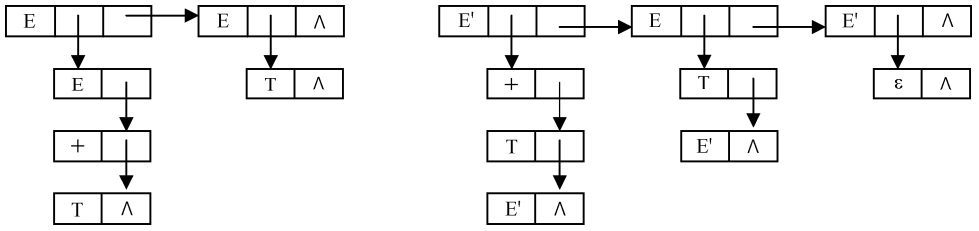
现在考虑消去左递归的文法等价变换由计算机程序自动来实现，这时必须考虑以下两个要点：

- 识别文法是否存在左递归，存在时，判断是规则左递归还是文法左递归；
- 文法在计算机内的存放（表示法）。

识别左递归的思路如下：关于每个非终结符号建立一个结点，如果存在规则 U_i::=U_j…，U_i、U_j∈V_N，则从结点 U_i 到结点 U_j 作一链接（弧）。当对一切规则建立了这样的链接后，对每个非终结符号 U 查看：是否存在一个链，从结点 U 出发，可沿着此链回到该结点 U。若存在，则表明文法是左递归的。如果所有这样的链长度都不超过 1，则是规则左递归的；否则，只要至少有一个链的长度大于 1，就是文法左递归的。

当具体用 C 语言或其他高级程序设计语言实现时，应引进怎样的数据结构才更有效地实现此左递归的判别，请读者自行考虑。

至于文法在计算机内的存放问题，考察规则左递归的消去：从 U::=Ux 与 U::=y 变换成 U::=yU' 与 U'::=xU' | ε，也就是说，U::=y 成为 U::=yU'，而 U::=Ux 成为 U'::=xU'|ε。这里 x 与 y 都可以是任意的符号串，因此可以引进链结构。例如，E::=E+T 与 E::=T 变换成：E::=TE'与 E'::=+TE'|ε，变换前后可有如图 4-3（a）与（b）所示的链结构。



（a）　　（b）

图 4-3

从该图可见有两类结点的数据结构如下所示。

规则链结点：

左部符号序号	右部首符号结点指针	下一规则指针

右部符号结点：

右部符号序号	后继符号结点指针

从图 4-3 中十分容易看出变换规律：对规则链中左递归的规则，即右部首符号与规则左部相同的规则链处，删除右部首符号结点（E），而在右部符号链的末结点处链接一个结点，其符号是新添加的非终结符号（E'），同时把左部符号改为新添加的非终结符号。而对于非左递归的规则，即右部首符号与左部符号不同的规则链处，在右部符号链的末结点（T）处链接一个结点，其符号是新添加的非终结符号（E'），最后，在规则链之末结点处链接一个ε规则（E'::=ε）。

当输出时，很容易得到消去了左递归的文法规则：

```
E'::=+TE'        E::=TE'          E'::=ε
```

通常为了简便起见，符号名仅由单个字符组成，但新引进的非终结符号为 E'，成为双字符符号，为保持全是单字符符号，可以从字母表的字母 A 开始，选择不是文法符号的英文字母作为新添加的非终结符号名。例如，对于本例，可用字母 A 代表 E'：

```
A::=+TA          E::=TA          A::=ε
```

尽管消去了左递归后，可以应用自顶向下分析技术进行语法分析，但由于回溯的存在，功效比较低下。为了提高功效，合适的做法是采用无回溯的自顶向下分析技术。

4.2　无回溯的自顶向下分析技术

4.2.1　应用条件

前面已看到在应用自顶向下分析技术时，如果文法关于某非终结符号具有左递归性，将导致应用失败。由于是面向目标而试探地进行分析的，不可避免地在分析过程中有回溯，导致效率甚低。因此如果既要可行又要高功效，应该采用无回溯的自顶向下分析技术，显然要求文法满足下列两个条件，即，

① 无左递归性。文法中关于任何非终结符号 U，都不具有规则左递归和文法左递归，即不存在形如 U::=U…的规则，也不存在 U=>+ U…。

② 无回溯性。产生回溯的原因是因为对某个非终结符号 U 展开时，可能存在有规则

$U::=u_1|u_2|\ldots|u_k$，对于其中两个选择 u_i 与 u_j，有 $u_i=>*\ T_i\ldots$与 $u_j=>*\ T_j\ldots$，$T_i, T_j\in V_T$，且 $T_i=T_j=T$，因此在尝试 $U::=u_i$ 失败之后又要尝试 $U::=u_j$，从而产生回溯。这如同在到某处去的某路口，可能有两条同名的路可继续前进，这就要尝试，便可能走回头路，也即回溯。如果 $T_i=T$, $T_j\neq T$，在选择 u_i 失败之后就不会再去关于 T 进行尝试。假定一个文法的任何 $U\in V_N$，存在 $U::=u_1|u_2|\ldots|u_k$，若 $u_i=>*\ T_i\ldots$与 $u_j=>*\ T_j\ldots$，T_i、$T_j\in V_T$, $i\neq j$，就有 $T_i\neq T_j$, 则称该文法**无回溯性**。

从无回溯性条件可知，当按某个选择 u_i 进行展开时，要么成功要么失败，再没有回溯的可能。

当一个文法关于某个非终结符号规则左递归时，必定存在回溯性，例如，E::=E+T|T，情况便是如此，通过消去左递归的文法等价变换，得到 E::=TE'与 E'::=+TE'，消去了左递归，同时也消去了回溯性。但反过来有回溯性不一定是左递归的，例如下列规则：

<if 语句>::=if(<表达式>)<语句> else <语句>
| if(<表达式>)<语句>

显然不是左递归的，但有回溯性，可通过提因子和引进新非终结符号这样的等价变换来消去回溯性。例如，

<if 语句>::=if(<表达式>)<语句>(else <语句> | ε)

因此可变换为，　<if 语句>::=if(<表达式>)<语句><else 部分>
<else 部分>::=else <语句> | ε

一般情况下，若有 U::=uv|uw，则 U::=u(v|w)，从而变换为 U::=uU'与 U'::=v|w，消去了回溯性。

当一个文法满足既无左递归性又无回溯性这两个条件时，就可能对相应的语言应用无回溯的自顶向下分析技术。

4.2.2　递归下降分析技术

1. 实现思想

考虑文法 G4.1'[E]：

```
E::=TE'        E'::=+TE'|ε   T::=FT'
T'::=*FT'|ε    F::=(E)|i
```

假定输入符号串为 i*i+i，可为它构造语法分析树如图 4-4 所示。

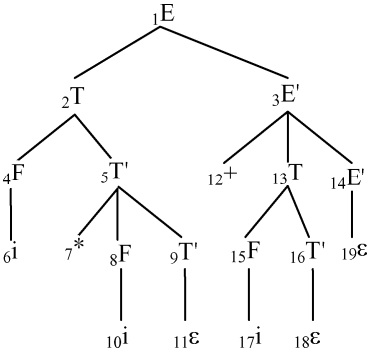


图 4-4

分析该语法树，按自顶向下分析技术，首先以识别符号 E 为根结点，试图以它为目标，按规则 E::=TE'展开。从图 4-4 可见，以序号为 2 的 T 为根结点的子树，它的末端结点符号串为 i*i，即可以覆盖输入符号串的左半部 i*i，显然有 $T=>*\ i*i$；而以序号为 3 的 E'为根结点的子树，它的末端结点符号串为+i，即可以覆盖输入符号串的右半部+i，显然有 $E'=>*\ +i$，两者结合起来正是输入符号串 i*i+i，也即以序号为 1 的 E 为根结点的（子）树，它的末端结点符号串覆盖整个输入符号串 i*i+i。事实上，$E=>TE'=>*\ i*iE'=>*\ i*i+i$。i*i 是相应句型中相对于 T 的短语，而+i 是相应句型中相对于 E′的短语。以序号为 4 的 F 为根结点的子树，它的末端结点符号串为 i，以序号为 5 的 T'为根结点的子树，它的末端结点符号串为*i 等，情况都类似。因此考虑关于各个非终结符号构造一些子程序（函数），它们各自处理在句型中相对于该非终结符号的短语，这便构成了所谓的递归下降分析技术。

应用递归下降分析技术实现句型分析的程序称为**递归下降识别程序**。

递归下降分析技术的基本实现思想是：让递归下降识别程序由一组子程序组成，这组子程序中的每一个对应于一个非终结符号，每个子程序处理的正是分析过程中，当前句型中相对于相应非终结符号的短语。不言而喻，相应于识别符号的子程序处理的是相对于识别符号的短语，也就是句子。由于文法往往是递归定义的，以便产生无限的符号串集合，因此，这些子程序往往也是递归定义的（递归函数定义）。

对于文法G4.1'[E],关于识别符号E的子程序,它以E为目标,期望按E进行展开,由于E::=TE',期望展开为 TE'，因此先后调用关于 T 与 E'的子程序，分别以 T 与 E'为目标，期望先后按 T 与 E'进行展开。因此递归下降分析技术是面向目标的，这个目标就是子程序所相应的非终结符号，它也是预测的，预测在相应句型中能找到这个相对于该非终结符号的短语。

2. 递归下降识别程序的构造

从递归下降识别程序的基本实现思想，不难看到各个子程序的构造思路。下面给出递归下降识别程序构造的例子。

【例 4.5】 试为文法 G4.1′[E]：

```
E::=TE'    E'::=+TE'    E'::=ε
T::=FT'    T'::=*FT'    T'::=ε    F::=(E)    F::=i
```

构造递归下降识别程序，其程序控制流程图如图 4-5 所示。

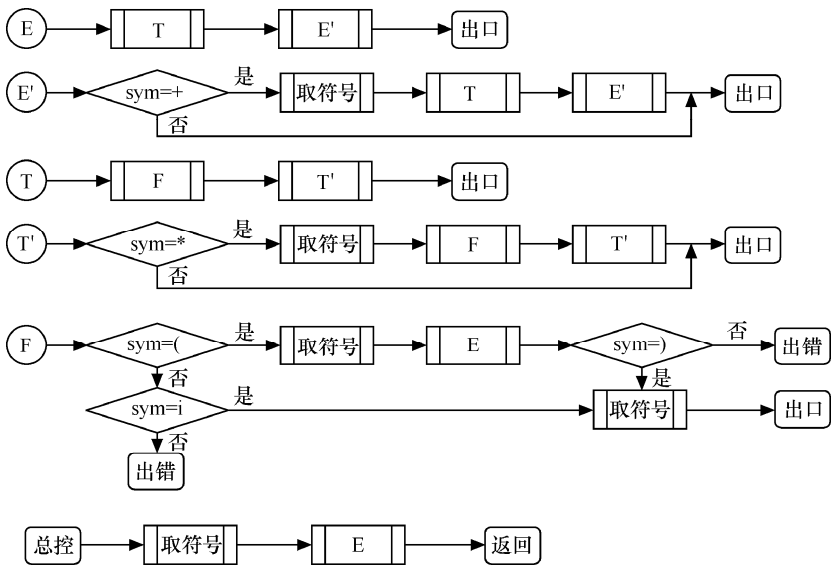


图 4-5

注意，在构造各子程序的控制流程图的同时，必需给出总控程序的流程图。

当略去说明部分等而仅写出控制部分时，可写出递归下降识别程序（框架）如下（C 型语言）。

```
/* 关于文法 G4.1'的递归下降识别程序 */
void  GetSymbol( ) { … }
void Error( ){ … }
void E( )
{  T( );  E1( );  }
void E1( )
{  if(sym==+)
   { GetSymbol( );  T( );  E1( );  }
```

```
}
void T( )
{  F( );  T1( );  }
void T1( )
{  if(sym==*)
   { GetSymbol( );  F( );  T1( );  }
}
void F( )
{  if(sym==i} GetSymbol( );
   else if(sym==( )
        {  GetSymbol( );  E( );
           if(sym==)) GetSymbol( );
           else Error( );
        }  else  Error( );
}
void main( )
{ GetSymbol( );  E( );  }
```

其中，GetSymbol 的功能是取符号，所取符号存放在全局变量 sym 中；Error 是出错处理函数，可以输出出错符号及它在输入符号串中的位置，然后结束分析或者进行相应的处理。

考察所写识别程序，不难体会递归下降分析技术名称的由来：一方面，E 调用 T，T 调用 F，F 又调用 E，因此 E 是（间接）调用本身的，T 与 F 有类似情况，总的来说，一般情况下，组成识别程序的各个子程序是（直接或间接）递归定义的；另一方面，E 代表表达式（Expression），T 代表项（Term），而 F 代表因子（Factor），从语言成分上看，E、T 与 F 是有层次的，E 层次最高，F 最低，E 调用 T、T 调用 F，层次下降了。了解这两个方面，对递归下降分析技术将有更好的理解。

构造递归下降识别程序时，请注意ε规则的处理与子程序构造约定这两个方面。

① ε规则的处理。ε规则的处理可以对照对 E'或 T'与 F 的处理。在关于 F 的子程序中，由于 F 仅有两种情形：或为(E)，或为 i，因此按 F 展开所产生的符号串中第一个符号不是(就是 i。如果既不是(，也不是 i，就表明输入有错。一般情况下，设 U::=Vv | Ww，其中 V、W∈V_N。设当前输入符号是 T，则仅当 V=>+ T…时与 Vv 匹配。如果不是 V=>+ T…，而是 W=>+ T…，则 T 与 Ww 匹配。如果既非 V=>＋T…，也非 W=>＋T…，则表明输入有错。但如果还存在规则 U::=ε，虽然既非 V=>＋T…，也非 W=>＋T…，但认为当前输入符号 T 与ε匹配，不会出现错误。例如对于 E'：E'::= +TE'|ε，一种可能是按第一个选择展开为+TE'，因此进入 E'时的输入符号应是+，但若不是+，则任何输入符号都可以与ε匹配，因此不会发生错误，在关于 E′的子程序中不调用 Error 函数。

② 子程序构造约定。具体约定如下：当进行句型分析时，凡进入某个关于非终结符号 U 的子程序时，都已取到相对于 U 的短语的第一个符号，凡在离开子程序时都已取到相对于 U 的短语的后继第一个符号。这从例 4.5 中的递归下降识别程序明显可见：每次调用某个子程序之前，必先取符号，特别是每当一个终结符号与输入符号（如+或*）匹配时便调用取符号函数 GetSymbol，然后调用相应的子程序（如 T 或 F）。当调用子程序 F 时，识别是 i 或)后取符号，正是相对于 F 的短语之后的第一个符号。

一般情况下，为某文法构造递归下降识别程序，其规范步骤如下。

步骤 1　首先检查该文法是否满足应用递归下降分析技术的条件，即无左递归性和无回溯性。如果不满足，则需进行文法等价变换使之符合应用条件；

步骤 2　对符合条件的文法构造递归下降识别程序的各个子程序，包括写出识别程序的总控程序。

【例 4.6】　试为文法 G4.4[C]：

```
C::=C;S              C::=S
S::=if(E)S else S    S::=if(E)S   S::=A
```

构造递归下降识别程序。

步骤 1　检查文法 G4.4[C]满足应用条件否。显然它既有左递归性又有回溯性，先进行文法等价变换，得到 G4.4'[C]：

```
C::=SC'              C'::=;SC'|ε
S::=if(E)SS' | A     S'::=else S | ε
```

经检查，新文法既无左递归性，也无回溯性。

步骤 2　为新文法 G4.4'[C]构造递归下降识别程序的各个子程序，相应的程序控制流程图如图 4-6 所示。

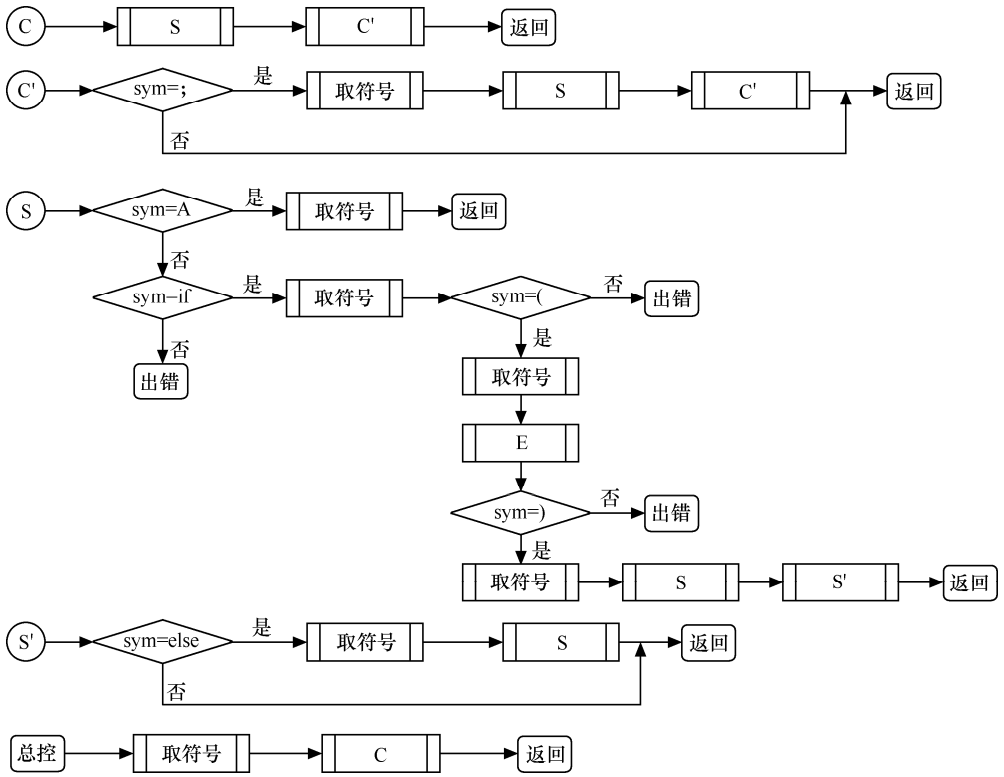


图 4-6

用 C 型语言所写的递归下降识别程序（框架）如下。

```
/* 文法 G4.4 的递归下降识别程序  */
void GetSymbol( ){ …  }
void Error( ){ … }
void C( )
{  S( );  C1( );  }
void C1( )
{  if(sym==;)
   { GetSymbol( );  S( );  C1( );  }
}
void S( )
{  if(sym==A) GetSymbol( );
```

```
    else if(sym==if)
         {  GetSymbol( );
            if(sym==( )
            {  GetSymbol( );  E( );
               if(sym==) )
               { GetSymbol( );  S( );  S1( );  }
               else Error( );
            }  else Error( );
         }  else Error( );
}
void S1( )
{  if(sym==else)
   { GetSymbol( );  S( );  }
}
void main( )
{  GetSymbol( );  C( );  }
```

为了进一步说明递归下降分析技术的基本实现思想，以例 4.7 加以说明。

【例 4.7】 设有文法 G4.5[S]：

```
S::=A|C   A::=i=E   C::=if(E)S
```

试为其构造递归下降识别程序。

步骤 1　首先检查文法 G4.5[S]是否满足递归下降分析技术的应用条件。易见既无左递归性，也无回溯性，满足应用条件。

步骤 2　构造递归下降识别程序的各个子程序如下。

```
void GetSymbol( ){ … }
void Error( ){ … }
void S( )
{   if(sym==i)       /*  A=>*  i…   */
     A( );
    else if(sym==if)   /*  C=>*  if…  */
     C( );
    else
     Error( );
}
void  A( )
{    if(sym==i)
     {  GetSymbol( );
        if(sym== =)
        {  GetSymbol( );
           if(sym==E)  GetSymbol( );
           else  Error( );
        }  else  Error( );
     }  else  Error( );
}
void C( )
{   if(sym==if)
    {  GetSymbol( );
       if(sym==( )
       {  GetSymbol( );
          if(sym==E)
          { GetSymbol( );
            if(sym==) )
            {  GetSymbol( );  S( ); }
```

```
            else Error( );
          } else Error( );
       }  else Error( );
    } else Error( );
}
void main( )
{  GetSymbol( );  S( );  }
```

请注意关于 S 的子程序（函数）。因为 A=>* i…，当 sym 是 i 时调用 A，而 C=>* if…。当 sym 是 if 时调用 C。由于在检查了 sym 后调用，在 A 与 C 函数定义中可不必再检查 sym 是 i 还是 if，改进的问题请读者自行考虑。

3. 应用递归下降分析技术句型分析

句型分析可以通过递归下降识别程序实现。具体过程是：当进入识别程序总控程序（主函数）时，读入一个符号后，调用相应于识别符号的子程序（函数），进行相对于识别符号的短语（句子）的识别，进而调用下一层次的各个子程序（函数）。这种函数调用与返回的过程难以用文字表达，如果输入符号串较长时情况更为突出。下面以图解表示说明分析过程，读者可以通过它进一步理解递归下降分析技术的实现思想。

图 4-7 展示了应用递归下降分析技术识别输入符号串 i*i 是否文法 G4.1′[E]的句子的过程，最终执行完函数 E 而成功结束，识别出 i*i 是文法 G4.1′的句子。读者可以以输入符号串 i*i+i 为例自行进行识别，画出识别过程的图解表示。

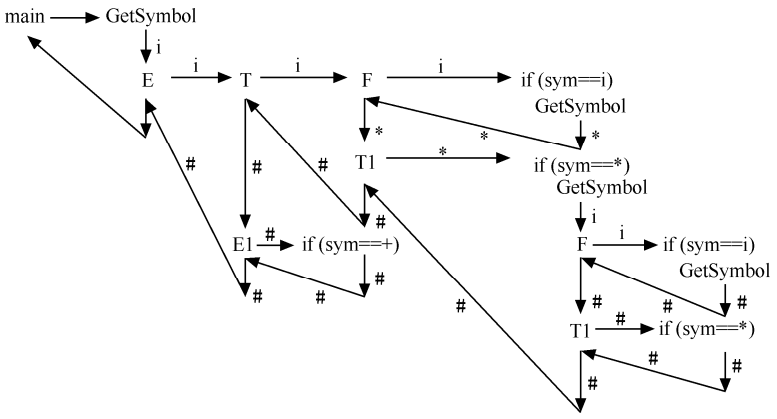


图 4-7

当识别出输入符号串是相应文法的句子时，需输出语法分析树。如何构造并输出语法分析树呢？可以在进入关于某非终结符号 U 的子程序时为 U 生成相应的结点，对于终结符号 T，则在子程序中当 T 与输入符号匹配时为 T 构造结点。这样，对函数定义可作如下的修改，例如对于 E 与 E1 可改写成：

```
void E( )
{  MakeNode(E);  T( );  E1( );  }
void E1( )
{  MakeNode(E1);
   if(sym==+)
   {  MakeNode(+);  GetSymbol( );  T( );  E1( ); }
   else  MakeNode(ε);
}
```

其他的子程序可类似地进行修改。细心的读者可能已想到仅生成结点是不够的，还必须确定结点

之间的关系，即确定结点之间的父子关系与兄弟关系。可以通过一些典型例子的分析来明确构造语法分析树的算法，这里不作详细讨论。提示：对各子程序（函数定义）增加参数，即父结点序号与左兄结点序号作为参数。例如，函数 E 与 E1（C 型语言）可设计如下：

```
int E(int father, int brother)
{ N=MakeNode(E, father, brother);  /* 建立结点 E，序号为 N */
  Nb=T(N,0);            /* T 的父结点序号为 N，左兄结点序号为 0(无) */
  Ns=E1(N, Nb);         /* E1 的父结点序号为 N，左兄结点序号为 Nb */
  Node[N].son=Ns;       /* E 的子结点 E1 序号为 Ns  */
  return N;             /* 返回 E 的结点序号 N */
}
int E1(int father, int brother)
{ N=MakeNode(E1, father, brother);
  if (sym==+)
  { N1=MakeNode(+, N, 0);  GetSymbol( );
    Nb=T(N, N1);
    Ns=E1(N, Nb);
    Node[N].son=Ns;
  } else
  { Ns=MakeNode(ε, N, 0);  /* 关于规则右部ε也需生成结点  */
    Node[N].son=Ns;
  }
  return N;
}
```

其他子程序（函数定义）可类似地设计。

从上述语法分析树结点的建立过程可见，当按递归下降分析技术构造语法分析树时，结点序号有两种：按推导生成的序号和按深度优先遍历顺序的实际建立顺序序号，最终的语法分析树中结点的序号都是按实际建立顺序的序号。在按预测分析技术构造语法分析树时，这种情况将看得更清楚。

从前面的讨论可见，递归下降分析技术有如下优点：

① 实现思想简单易理解，对照文法规则，可方便地画出相应的程序控制流程图或写出程序。

② 能灵活地在各个子程序中添加语义处理工作，例如生成语法分析树。

③ 能用 C 型语言写出递归下降识别程序，递归子程序的实现功效高，则相应的识别程序功效也高。

在早期，递归下降分析技术因其简单和切实可行，曾被很多编译程序所采用。

4.2.3 预测分析技术

1. 预测分析技术的实现思想

虽然递归下降分析技术简单易行，通常可由高级程序设计语言来实现，但由于组成它的各个子程序通常是递归定义的，一方面需要所采用的程序设计语言支持子程序（函数定义），特别是需要支持递归定义；另一方面，其实现功效取决于程序设计语言的函数设施。一般地说，函数的调用需要额外的开销，影响了实现功效。现在考虑其实现与程序设计语言无关的分析技术，这就是预测分析技术。

预测分析技术是一种无回溯的自顶向下分析技术，它使用一个称为**预测分析表**的分析表和一个分析栈联合进行控制，实现句型分析。这个预测分析表用来指明在句型分析过程中，当正待按其展开的非终结符号，遇上当前输入符号时按哪个规则展开。换句话说，以该非终结符号为目标，

预期在该当前输入符号时按所指明规则展开。例如，对于文法 G4.1'[E]：

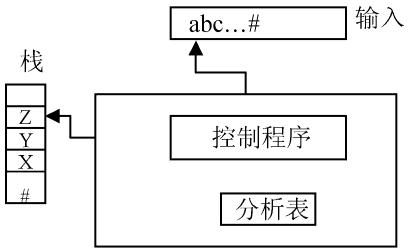
```
E::=TE'      E'::=+TE'     E'::=ε      T::=FT'
T'::=*FT'    T'::=ε        F::=(E)     F::= i
```

其预测分析表如表 4.1 所示。从表中可见当分析栈顶非终结符号 E 与当前输入符号 i 匹配时，规则为 E::=TE'，即正按 E 为目标展开时，输入符号为 i，则按规则 E::=TE'展开。若栈顶为非终结符号 F 时，若输入符号是 i，则按规则 F::=i 展开，若输入符号是(，则按规则 F::=(E)展开。分析表中元素为空白时，表示出错，即分析栈顶非终结符号与当前输入符号的匹配是错误的。

表 4.1

规则 / 输入 / 栈顶	+	*	(	)	i	#
E			E::=TE'		E::=TE'	
E'	E'::=+TE'			E'::=ε		E'::=ε
T			T::=FT'		T::=FT'	
T'	T'::=ε	T'::=*FT'		T'::=ε		T'::=ε
F			F::=(E)		F::=i	

分析栈用来存放句型分析过程中动态产生的文法符号序列，具体说，开始时把识别符号下推入分析栈中，以后每当分析栈顶是非终结符号时，以它为目标，按它进行展开。这时由该非终结符号与当前输入符号共同确定预测分析表元素，即按其展开的规则。确定规则后，从分析栈中上退去栈顶元素，然后按逆序把所确定的规则的右部各个符号下推入分析栈。因此分析栈记录了分析活动经历。



预测识别程序

图 4-8

应用预测分析技术实现句型分析的程序称为**预测识别程序**。预测识别程序的模型如图 4-8 所示。

预测识别程序由控制程序及分析表组成，它以要识别的符号串作为输入，以语法分析树或推导作为输出。

请注意，输入符号串的右端添加有一个符号#，标志输入符号串的结束，此外，预测分析技术还要求在输入符号串之前也有符号#，标志输入符号串的左端，因此，符号#称为左右端标志符号，它不是文法符号，是由识别程序自动添加的。预测识别程序的控制程序在赋初值时，下推入识别符号之前便下推入符号#。

2. 应用预测分析技术句型分析

预测分析技术通过使用预测分析表和分析栈联合进行控制，实现句型分析。下面关于文法 G4.1'[E]：

```
E::=TE'   E'::=+TE'|ε    T::=FT'    T'::=*FT'|ε     F::=(E)| i
```

以对符号串 i+i*i 进行句型分析为例进行说明，其预测分析表见表 4.1。

初始时，首先把左右端标志符号#下推入分析栈，然后把识别符号 E 下推入分析栈，E 成为分析栈顶符号，分析栈如图 4-9（a）所示，这时输入指针指向左端第一个输入符号 i。

当分析栈顶为非终结符号 E 时，表明以其为目标，将按以其为左部的规则的右部展开。这时查该非终结符号与输入指针指向的符号（当前输入符号）配对所确定的预测分析表元素。如果分析表元素为空白，表明分析栈顶非终结符号与输入符号不匹配，出现错误，输入符号串不是相应

文法的句子。现在 E 与 i 是匹配的，分析表元素为规则 E::=TE'，按此规则展开 E，这时上退去分析栈顶符号 E，把规则右部符号串 TE′按逆序下推入分析栈，这时分析栈如图 4-9（b）所示。

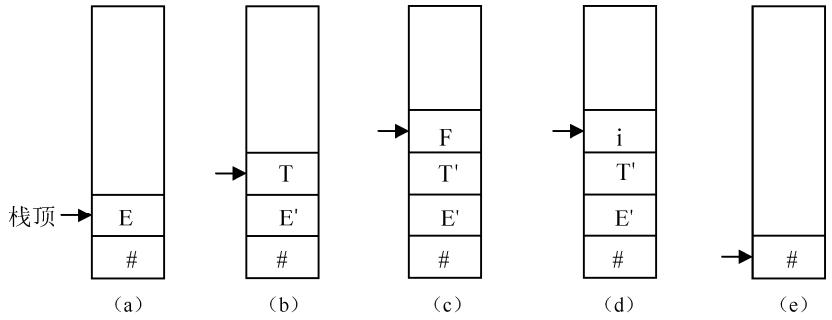


图 4-9

继续按分析栈顶非终结符号 T 与当前输入符号 i 配对确定的预测分析表元素，即按规则 T::=FT'展开 T。现在，上退去分析栈顶符号 T，把规则 T::=FT'的右部符号串 FT'按逆序下推入分析栈，分析栈如图 4-9（c）所示。类似地，当按分析栈顶的非终结符号 F 展开时，分析栈如图 4-9（d）所示。

当分析栈顶为终结符号时，表明当前输入符号应与此栈顶符号相同；若不相同，表明不正确，输入符号串不是相应文法的句子，如果相同，则上退去分析栈顶符号，输入指针指向下一个输入符号。

如此继续，最终达到输入符号串的右端，输入指针指向右端标志符号#，而分析栈中也仅有元素#，如图 4-9（e）所示，这时分析结束，识别出输入符号串 i+i*i 是相应文法的句子。

通常用表列方式给出分析全过程，如表 4.2 所示。

表 4.2

步　骤	栈	输　入	输　出
0	#E	i+i*i#	E::=TE'
1	#E'T	i+i*i#	T::=FT'
2	#E'T'F	i+i*i#	F::=i
3	#E'T'i	i+i*i#	
4	#E'T'	+i*i#	T'::=ε
5	#E'	+i*i#	E'::=+TE'
6	#E'T+	+i*i#	
7	#E'T	i*i#	T::=FT'
8	#E'T'F	i*i#	F::=i
9	#E'T'i	i*i#	
10	#E'T'	*i#	T'::=*FT'
11	#E'T'F*	*i#	
12	#E'T'F	i#	F::=I
13	#E'T'i	i#	
14	#E'T'	#	T'::=ε
15	#E'	#	E'::=ε
16	#	#	

从上述分析过程可概括为：在任何分析时刻，分析栈顶符号 X 与当前输入符号 T 共同决定预测识别程序所应执行的分析动作。预测识别程序可能执行 4 种动作：

① 如果 $X\in V_T$，且 X=T=#，则分析成功而结束。

② 如果 $X\in V_T$，X=T≠#，则从分析栈中上退去 X，并把输入指针推进指向下一个输入符号。

③ 如果 $X\in V_N$，且分析表 A 的元素 A[X][T]=“$X::=x_1x_2\cdots x_k$”，则把分析栈顶的符号 X 上退去，把 $x_1x_2\cdots x_k$ 依次下推入分析栈（注意：按照逆序把 $x_1x_2\cdots x_k$ 下推入分析栈，使 x_1 在分析栈顶）。

④ 在其他情况下，即 $X\in V_T$，且 X≠T，或 $X\in V_N$，且 A[X][T]=ERROR（空白元素），识别失败，调用出错处理程序，使之能继续下去，或结束分析。

预测识别程序的控制程序（C 型语言）可写出如下，其中 S 为分析栈。

```
/* 预测识别程序控制程序 */
void main( )
{  push(#);  push (Z); /* 依次把#与识别符号 Z 下推入分析栈 */
   T=下一输入符号;
   Flag=true;  /* Flag 标志继续否，true: 1，false: 0 */
   while (Flag)
   {  X=top(S);  /* 置 X 为分析栈顶元素 */
      if(X∈VT∪{#})
         if(X==T)
           if(T==#)  Flag=false;
           else
           {  pop(S); /* 上退去分析栈顶元素 */
              T=下一输入符号;
           }
         else
           ERROR ( );
      else  /* X∈VN */
         if(A[X][T]=="X::=x1x2…xk")
         { pop(S);  push (xkxk-1…x1);
            /* 上退去分析栈顶元素后把规则右部按逆序下推入分析栈 */
            打印输出规则 X::=x1x2…xk;
         } else  ERROR( );
   } /* while */
   STOP;  /* 分析成功而结束  */
}
```

3. 语法分析树的生成

语法分析阶段为下一阶段语义分析提供输入，也就是说，当分析成功，识别出输入符号串是相应文法的句子时，应向语义分析阶段提供内部中间表示，即语法分析树或推导等。这里讨论如何在语法分析过程中生成语法分析树，从实例分析着手。

假定文法 G4.1′[E]的句子 i*i 有如下的推导：

```
E=>TE'=>FT'E'=>iT'E'=>i*FT'E'=>i*iT'E'=>i*iE'=>i*i
```

如果不考虑分析技术，单纯从上述推导建立语法分析树，将如图 4-10（a）所示。当由预测识别程序句型分析生成语法分析树时，如果按其展开的规则的右部中有不止一个符号，将不同时为这些符号建立结点，而是按深度优先遍历的顺序建立结点，相应语法分析树如图 4-10（b）所示。

为了能正确生成语法分析树，引进两种排序序号，即结点序号与实际建立顺序。前者是按推导过程建立的顺序,它在按规则右部展开时建立；后者是实际建立结点的顺序，是在分析过程中实际建立一个结点时确定的顺序。对照前面讨论的分析过程，不难发现：只需当按一个符号展开时，为其建立实际的结点即可，也就是说，当一个符号出现在分析栈顶时为该符号实际建立结点，确定顺序。例如，当分析栈中内容为#E'T 时，符号 T 在分析栈顶，其结点序号为 2，且因按其展开，实际建立顺序也为 2，但由于 T 展开为 FT'，分析栈变成#E'T'F，为符号 E'建立的结点便不可能实际建立顺序为 3，当 E'出现在分析栈顶，为它实际建立结点时，实际建立顺序为 11。以实际建立结点的顺序作为结点的序号时，相应的语法分析树才是语法分析的结果。

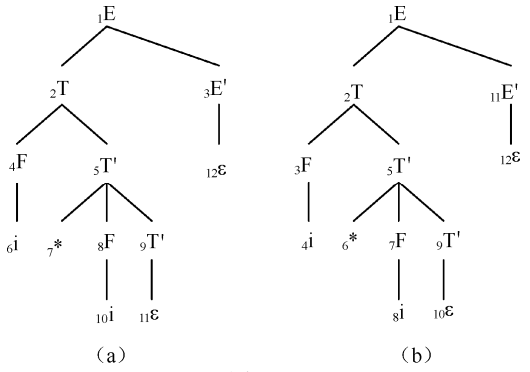


图 4-10

概括起来，预测识别程序进行句型分析建立语法分析树的过程如下。

① 设立分析栈，初始时，把左端标志符号#与识别符号 Z 依次下推入分析栈。设立输入指针指向当前输入符号，初始时当前输入符号为左端第一个符号。为 Z 建立结点，其结点序号为 1。

② 每个分析步，当分析栈顶为非终结符号 U，当前输入符号为 T，查看分析表元素 A[U][T]：

- 若 A[U][T]= “U::=$x_1x_2\cdots x_k$”，则为规则右部符号串从左到右依次建立结点，并与结点 U 建立父子与兄弟关系。同时为结点 U 确定实际建立顺序，上退去分析栈顶元素 U，把规则右部符号串 $x_1x_2\cdots x_k$ 依逆序下推入分析栈，使 x_1 在分析栈顶。
- 若 A[U][T]=ERROR（空白元素），则出错，表明输入符号串不是相应文法的句子。

③ 当分析栈顶为终结符号 T，则与当前输入符号比较：

- 若相同，则确定该终结符号 T 相应结点的实际建立顺序，并上退去分析栈顶符号 T，同时把输入指针指向下一个输入符号。
- 若不相同，则表明出错，分析失败，输入符号串不是相应文法的句子。

④ 当分析栈顶符号是左端标志符号#，

- 若当前输入符号是右端标志符号#，则分析成功，识别出是句子而结束；
- 若当前输入符号不是右端标志符号#，则分析失败，输入符号串不是句子。

请注意ε规则，从表 4.2 可见，当按 U::=ε规则展开时，并不把ε下推入分析栈，但在生成语法分析树时，必须为ε生成结点，并建立与父结点的父子关系。

从上述语法分析树的构造过程可见，当利用预测分析技术构造语法分析树时，将按深度优先遍历的顺序建立结点，而最终的语法分析树应是按实际建立顺序建立的。因此，可以说，按自顶向下分析技术构造的语法分析树是按深度优先遍历顺序生成的。

请读者结合语法分析树的建立，自行修改前面所给的预测识别程序。

如果要在分析过程中生成推导，该如何处理？可以从实例出发来分析，例如，对于 i*i 的推导：

```
E=>TE'=>FT'E'=>iT'E'=>i*FT'E'=>i*iT'E'=>i*iE'=>i*i
```

对照分析栈内容，一个符号如果下面加下画线，表示它是被从分析栈中上退去的。

```
E=>TE'=>FT'E'=>iT'E'=> iT'E'=>i*FT'E'=>i*FT'E'
 =>i*iT'E'=>i*iT'E'=>i*iE'=>i*i
```

显而易见，只需把被上退去的终结符号存入一个暂存区中，它与当时分析栈中的内容连在一起可构成一个句型，因此，依次把暂存区中内容与分析栈中内容连接后输出，可得到整个推导。请读者按此思路自行尝试上机实践。

4．预测分析表的构造

前面的讨论表明：预测分析技术实现的关键是预测分析表。预测分析表的构造方法其实很简单。正如前面所看到的，当对 E 展开时，有分析表元素 A[E][i]= “E::=TE′” 与 A[E][()= “E::=TE'”，这是因为 E=>TE'=>FT'E'=>(…与 E=>TE'=>FT'E'=>i…，即 E=>* (… 与 E=>* i…。

一般地，若有规则 U::=u，U=>u=>* T…，T∈V_T，把从 u 所推导得到的一切可能的符号串的打头的终结符号所组成的集合，称为 u 的 First 集合，用符号表示就是

```
First(u)={ T | u=>*  T…, T∈V_T }
```

如果 u=>* ε，则让ε∈First(u)。例如，First(E)= First(T)= First(F)={(, i)，First(E')={+, ε}，First(T')={*, ε}。

一般地，若存在 U::=u，有 T∈First(u)，T∈V_T，则有 A[U][T]= “U::=u”。

对于ε规则，例如，E'::=ε，将存在分析表元素 A[E'][T]= “E'::=ε”。这时 T=？。输入符号不可能是ε，什么输入符号 T 与 E'配对，才能让 E'=>ε? 试看图 4-11 中所示的关于 i+i 的语法分析树。关于 T'应用了规则 T'::=ε，把 T'展开为ε，这时的输入符号将是+，是把ε直接归约为 T'时 T'的后继终结符号。另一处应用规则 T'::=ε把 T'展开成ε，这时的输入符号将是识别程序自动添加的右端标志符号#，此#也是在ε直接归约为 T'时 T'的后继符号。关于 E'::=ε的应用，情况类似。为了更清楚地看清 T'的后继输入符号，可写出推导如下：

```
E=>* iT'+iεε=>iε+iεε
```

其中，ε是空符号串，实际上该推导是：E=>* iT'+i=>i+i。+是句型 iT'+i 中紧随 T'之后的终结符号。一般地可画出如图 4-12 所示的语法分析树，T 是句型中紧随 U 之后的终结符号，当 U 是句型中的最右符号时，其后跟的将是右端标志符号#。

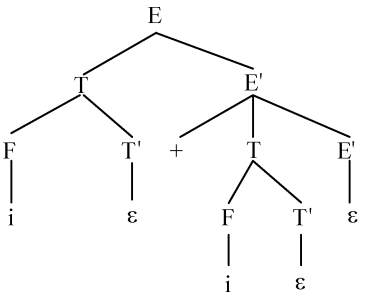


图 4-11

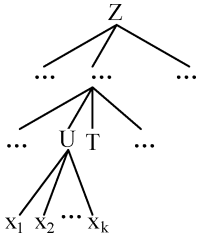


图 4-12

在包含一个非终结符号 U 的一切句型中，紧随 U 之后的一切可能的终结符号（可能还有右端标志符号#）所组成的集合，称为 U 的 Follow 集合，用符号表示如下：

```
Follow(U)={T|Z=>* …UT…, T∈V_T∪{#} }
```

例如，+∈Follow(E')和#∈Follow(E')，事实上，Follow(E') =Follow(T')={+,),# }。

一般地，对于 U::=u，若ε∈First(u)，则对于 Follow(U)中的一切终结符号 T 或#，置 A[U][T]= “U::=u” 或 A[U][#]= “U::=u”。

构造预测分析表的算法步骤可概括如下。

步骤 1　对文法的每个规则 U::=u，执行步骤 2 与步骤 3。

步骤 2　对每个 T∈V_T，若 T∈First(u)，则置 A[U][T]=“U::=u”。

步骤 3　若ε∈First(u)，则对每个 T∈Follow(U)，置 A[U][T]=“U::=u”；若#∈Follow(U)，则置 A[U][#]=“U::=u”。

步骤 4　分析表中未定义的元素全置为 ERROR（用空白表示）。

最后结束时所得的表 A 就是所构造的预测分析表。对于文法 G4.1'[E]的预测分析表正是应用上述算法所得。

【例 4.8】　设有文法 G4.6 [S]：

```
S::=A|C          A::=V=E    C::=if(E)S
E::=E+V|E-V|V    V::=i
```

试为其构造预测分析表。

首先检查文法 G4.6[S]是否满足应用预测分析技术的条件，即既无左递归性，也无回溯性。显然，由于存在规则 E::=E+V|E–V|V 不满足应用条件，进行消去左递归的文法等价变换，得到文法 G4.6'[S]：

```
S::=A     S::=C      A::=V=E     C::=if(E)S    V::=i
E::=VE'   E'::=+VE'  E'::=-VE'   E'::=ε
```

构造 First 集合和 Follow 集合如下。

```
First(V=E)={i}          First(if(E)S)={if}    First(i)={i}
First(VE')={i}          First(+VE')={+}       First(-VE')={-}
First(A)  ={i}          First(C)={if}         First(S)={i, if}
First(V)  ={i}          First(E)={i}          First(E')={+,-,ε}
Follow(S)={#}           Follow(A)={#}         Follow(C)={#}
Follow(V)={=,},+,-,#}   Follow(E)={ }, #}          'Follow(E')={ ), #}
```

最后填分析表，如表 4.3 所示。

表 4.3

规则 / 输入 / 栈顶	=	if	(	)	i	+	–	#
S		S::=C			S::=A			
A					A::=V=E			
C		C::=if(E)S						
V					V::=i			
E					E::=VE'			
E'				E'::=ε		E'::=+VE'	E'::= –VE'	E'::=ε

不需要一开始就计算全部的 Follow 集合，只需对 u=>*ε的规则 U::=u 的左部 U 计算 Follow 集合，需要时再对相关的其他非终结符号计算 Follow 集合。例如，本例中，仅 E'::=ε，只需要计算 Follow(E')，但它与 E 相关，因此再计算 Follow(E)。为了有助于读者计算 First 集合与 Follow 集合，给出它们的具体计算步骤，初始时一切 First 集合与 Follow 集合均为空集。

First 集合的计算步骤如下。

步骤 1　若 X∈V_T，则 First(X)={X}。

步骤 2　若 X∈V_N，并存在规则 X::=T…，T∈V_T，则把 T 添加入 First(X)中。若存在 X::=ε，则把ε也添加入 First(X)中。

步骤 3　设存在规则 $X::=x_1x_2\cdots x_k$，若 x_1、x_2、…、$x_{j-1}\in V_N$，且$\varepsilon\in First(x_i)(i=1,2,\cdots,j-1)$，即 $x_1x_2\cdots x_{j-1}=>^*\varepsilon$，则把 $First(x_i)$ $(i=1,2,\cdots,j-1)$中的一切非ε符号添加到 $First(x_1x_2\cdots x_j)$中，若 $\varepsilon\in First(x_i)(i=1,2,\cdots,k)$，则把ε加入 $First(x_1x_2\cdots x_k)$。

Follow 集合的计算步骤如下。

步骤 1　把左右端标志符号#加入 Follow(Z)，Z 是识别符号。

步骤 2　若存在规则 $U::=xVy$，$V\in V_N$，则 First(y)中除ε外的一切符号都属于 Follow(V)。

步骤 3　若存在规则 U::=xV 或 U::=xVy，其中 $V\in V_N$，且$\varepsilon\in First(y)$，即 $y=>^*\varepsilon$，则 Follow(U)中的一切符号都属于 Follow(V)。

如前所述，在构造 Follow 集合时，可以首先找到存在规则 U::=ε的 U，对它计算 Follow(U)，然后计算与 U 相关的非终结符号的 Follow 集合，最终计算出一切必要的 Follow 集合。

应用预测分析技术进行句型分析之所以可行，是因为预测分析表中的一切元素都是确定的，使得分析栈顶符号与当前输入符号配对所确定的分析动作是唯一的。但实际上存在一些情况，分析表元素的值不唯一，看下面的例子。

【例 4.9】 设有文法 G4.7[S]：

```
S::=if(E)SS' | v=E   S'::=else S | ε   V::=i   E::=i
```

试为其构造预测分析表。

首先，判别该文法是否符合预测分析技术的应用条件，即无左递归性和无回溯性。显然 G4.7 是满足的，无需进行文法等价变换。

其次，计算 First 集合和 Follow 集合。

```
First(S)={if, i}      First(S')={else, ε }
First(V)={i}          First(E)={i}
```

因为仅有 S'::=ε，故计算 Follow(S')，它与 Follow(S)相关，计算这两者：

```
Follow(S') =Follow(S) ={else, #}
```

最后按构造算法填预测分析表 A，结果如表 4.4 所示。

表 4.4

规则 \ 符号 / 栈顶	if	(	)	=	else	i	#
S	S::=if(E)SS′					S::=V=E	
S'					S'::=else S S'::=ε		S'::=ε
V						V::=i	
E						E::=i	

表 4.4 中 A[S'][else]包含两个值，即 S'::=else S 与 S'::=ε。前者因为规则 S'::=else S，else∈First(else S)，后者因为规则 S'::=ε，且 else∈ Follow(S')。

对于其分析表元素均为唯一的文法引进一个重要的文法类概念：LL(1)文法类。

如果对于某文法，其预测分析表的值都是唯一的，则该文法称为 **LL(1)文法**。

一个 LL(1)文法是无二义性的文法，它所定义的语言正是利用它的预测分析表所能识别的所有句子。

例如文法 G4.6[S]是 LL(1)文法，是无二义性的。文法 G4.7[S]不是 LL（1）文法，因为它是二义性的文法。例如，对于符号串 if(i) if(i) i=i else i=i，可为它构造如图 4-13（a）和（b）所示的两

个不同的语法分析树。

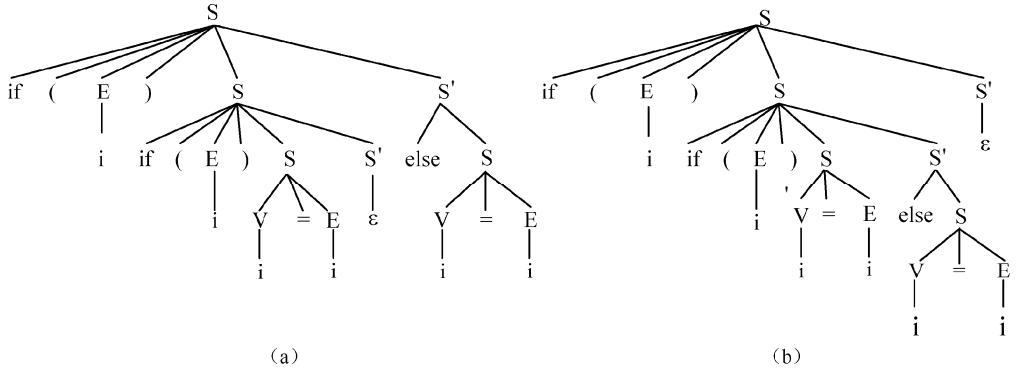


图 4-13

事实上，文法 G4.7[S]可改写为：

```
S::=if(E)S else S | if(E)S | V=E    V::=i    E::=i
```

相当于 C 语言的 if 语句与赋值语句，对于 if 语句必须给出 else 与相应 if 匹配的规定，否则就难以确定 else 与哪一个 if 匹配，例如下列语句：

```
if (i) if (i) i=i; else i=i;
```

如果没有相关规定，那么其中的 else 可与两个 if 中的任一个匹配，因此文法具有二义性。为了避免这种二义性，一般规定：嵌套 if 语句中，else 与其前面（左边）最近的、尚未与 else 匹配的 if 匹配。对于此例，else 就是与右边的 if 匹配。

为了在构造预测分析表之前，就判别一个文法是否是 LL(1)文法，给出文法是 LL(1)文法的充分必要条件如下：一个文法 G 是 LL(1)文法，当且仅当对于 G 中的每个非终结符号 U，存在以 U 为左部的多个规则时，其中任何两个重写规则 U::=v 与 U::=w，下面 3 个条件成立：

① First(v)∩First(w)=∅。

② v=>* ε与 w=>* ε不能同时成立。

③ 若 w=>* ε，则 First(v)∩Follow(U)=∅。

显然文法 G4.7[S]之所以发生预测分析表元素不唯一，是因为对于 S'::=else S| ε，有 w=ε ,w=>* ε，First(else S)∩Follow(S')={else}≠∅，文法 G4.7[S]不是 LL(1)文法。

4.3 预测识别程序句型分析的计算机实现

要在计算机上实现预测分析技术，也就是运行预测识别程序进行句型分析，不言而喻，必要的是编写预测识别程序控制程序，并设置好预测分析表。按照前面的讨论，不难实现预测识别程序控制程序，关键是解决两方面的问题，即预测分析表的存放与语法分析树的构造。

4.3.1 预测分析表的存储表示

预测分析表在概念上与实践上反差较大，必须解决预测分析表在计算机内的存放问题。

预测分析表是一个表格，行是将出现在分析栈顶的非终结符号，列是输入符号，而表中元素是相应的文法规则，显然，它将用二维数组来实现。对于各种程序设计语言，包括 C 语言，数组元素的下标应是整型值。为了能用二维数组实现预测分析表，让序号来代替符号，即非终结符号

U 用 U 在 V_N 中的序号代替，输入符号 T 即终结符号，用 T 在 V_T 中的序号代替。至于该二维数组元素的值，也没必要是构成文法规则的符号串，用文法规则序号便可以。需要注意的是：一切终结符号和非终结符号的序号都从 1 开始，它们的排序及一切文法规则的排序必须与预测分析表相一致。为区别终结符号与非终结符号，如前所述，只需把非终结符号的序号加上易识别的数值，如 100，同时设置左右端标志符号#的序号为 0。

预测分析表的数据结构可以设计如下（C 型语言）：

```
int 预测分析表[MaxVnNum][MaxVtNum];
```

其中 MaxVnNum 与 MaxVtNum 分别是可允许的非终结符号最大个数与终结符号最大个数。例如，当 V_N={E，E′，T，T′，F}，E、E′、T、T′与 F 的序号依次是 1、2、3、4 与 5，V_T={+,*,(,),i}，+、*、(、)与 i 的序号依次是 1、2、3、4 与 5，并让输入符号串的左右端标志符号#的序号为 0 时，利用 C 语言的置初值设施，对照表 4.1，可以如下地给出文法 G4.1′[E]的预测分析表 A：

```
int  A[5+1][6]=
{  {0},
   { 0,0,0,1,0,1},{ 3,2,0,0,3,0 },{0,0,0,4,0,4},
   {6,6,5,0,6,0},{ 0,0,0,7,0,8 }
};
```

预测分析表元素的值是文法规则序号，合适的是用数组结构存放文法规则，可以设计如下：

```
typedef struct
{  int  左部符号序号;
   int  右部符号串[MaxLength];  int  右部长度;
}  文法规则类型;
文法规则类型  文法[MaxRuleNum];
```

当一切非终结符号与终结符号如同前面一样地排序时，利用 C 语言置初值方式，可给出文法 G4.1′[E]如下：

```
文法规则类型  文法 G4.1′[E]=
{  {0},
   { 101,{ 0,103,102 },  2},  /* 1: E::=TE′ */
   { 102,{ 0,1,103,102 },3 },/* 2: E′::=+TE′ */
   { 102,{ 0 },           0}, /* 3: E′::=ε */
   { 103,{ 0,105,104 },  2 }, /* 4: T::=FT′ */
   { 104,{ 0,2,105,104 },3},  /* 5: T′::=*FT′ */
   { 104,{ 0 },          0 }, /* 6: T::=ε */
   { 105,{ 0,3,101,4 }, 3},   /* 7: F::=(E) */
   { 105,{ 0,5 },       1 }   /* 8: F::=i */
};
```

提醒：C 语言数组元素的下标从 0 开始，而序号一般从 1 开始，因此对应于 0 号规则的内容是{0}，即全 0，每个规则右部 0 号元素置为 0，例如，规则 E::=TE′对应于{ 101,{0,103,102 },2}，而规则 E′::=ε 对应于{102,{0},0}，其中非终结符号的序号加了易识别的值，即 100。

4.3.2　语法分析树的构造及输出

当用 C 语言实现预测识别程序时，必要的是设计语法分析树的数据结构。语法分析树由结点组成，要确定结点在语法分析树中的位置，只需确定它的父结点、左兄结点与右子结点。如同第 2 章中所讨论的，可以把语法分析树的数据结构设计为结点数组（C 型语言）如下：

```
语法分析树结点类型  语法分析树[MaxNodeNum];
```

其中语法分析树结点类型的数据结构设计如下：

```
typedef  struct
{  int 结点序号;
   int 文法符号序号; int 父结点序号;
   int 左兄结点序号; int 右子结点序号;
} 语法分析树结点类型;
```

但是，如前所述，在应用自顶向下分析技术生成语法分析树时，有两种序号，即按推导生成的结点序号与按预测分析技术的实际建立顺序序号，最终的结果应是实际建立顺序序号的语法分析树，因此除了语法分析树外，还引进过渡语法分析树，其结点的数据结构设计如下：

```
typedef  struct
{  int 结点序号;
   int 文法符号序号; int 父结点序号;
   int 左兄结点序号; int 右子结点序号;
   int 实际建立顺序序号;
} 过渡语法分析树结点类型;
```

过渡语法分析树的数据结构设计如下：

```
过渡语法分析树结点类型  过渡语法分析树[MaxNodeNum];
```

其中文法符号序号和前面一样处理，即终结符号的序号是它在 V_T 中的序号，而非终结符号的序号是它在 V_N 中的序号再加上一个易于识别的值（如 100），MaxNodeNum 是可允许的最大结点个数。

过渡语法分析树中既包含了按推导建立的结点序号，也包含了结点的实际建立顺序序号，由这种对应关系可以构造所需的，仅包含实际建立顺序的语法分析树。

显然语法分析树的打印显示输出按下列表格形式是十分容易实现的。

结点序号	文法符号	父结点序号	左兄结点序号	子结点序号
1	E	0	0	11
2	T	1	0	5
3	F	2	0	4
4	i	3	0	0
5	T′	2	3	9
⋮				

为了检查预测分析过程的正确性，可以打印显示分析过程如下所示。

步骤	栈	输入	输出
0	#E	i*i#	E::=TE′
1	#E′T	i*i#	T::=FT′
2	#E′T′F	i*i#	F::=i
3	#E′T′i	i*i#	
4	#E′T′	*i#	T′::=*FT′
5	#E′T′F*	i #	
⋮			

显然为显示输出，只需在每一分析步记录当时的栈内容、输入指针指向的当时位置以及按其展开的规则序号。

分析栈的数据结构可设计如下：

```
struct
```

```
{  int  分析栈内容[MaxDepth]; /* 元素是结点序号 */
   int  top;
} 分析栈;
```

关于分析栈的 push、top 与 pop 等操作是容易实现的。

建议读者自行在计算机上实现预测识别程序句型分析，能生成语法分析树，并显示输出分析过程。

本 章 小 结

本章讨论自顶向下分析技术，内容包括：概况（讨论前提、要解决的基本问题与应用条件），两类无回溯的自顶向下分析技术，即递归下降分析技术和预测分析技术，特别讨论了结合分析技术进行句型分析时，构造推导和语法分析树的的计算机实现问题。

依据推导来构造语法分析树，以及按自顶向下分析技术来构造语法分析树，结点建立的顺序是不同的，按自顶向下分析技术，结点按深度优先遍历顺序生成。当文法是左递归的，将不可能应用自顶向下分析技术，因此必须进行消去左递归的文法等价变换，一般情况下应用不带回溯的自顶向下分析技术，必须满足无左递归性与无回溯性。在应用自顶向下分析技术之前，必须首先检查文法是否满足这两个应用条件。读者应熟练掌握文法左递归消去的文法等价变换。

递归下降分析技术的基本思想是：为文法的每个非终结符号设计一个子程序，它处理相应句型中相对于该非终结符号的短语。一方面是递归，因为文法递归定义，所以子程序通常是递归定义的；另一方面是下降，因为子程序调用而语法概念层次下降。递归下降识别程序的构造比较简单，不论是控制流程图形式，还是 C 型语言程序形式，都与文法有直接的对照关系。正因为其简单易理解，在早期，很多编译程序采用了递归下降分析技术（所谓的递归子程序方法）。

预测分析技术的基本思想是：用一个分析表和一个分析栈联合进行控制，从而实现句型分析。两个要点，一是应用预测识别程序进行句型分析；二是预测分析表的构造。读者应熟练掌握预测识别程序进行句型分析。构造预测分析表时要注意的是 ε 规则的处理。不言而喻，预测分析表的元素必须都是唯一的，这时的预测分析表称为 LL(1)分析表，相应文法称为 LL(1)文法。

本章相关的概念有文法等价与文法等价变换、回溯、First 集合、Follow 集合等。

本章讨论了句型分析时构造推导与语法分析树的计算机实现问题，要点是相应数据结构的设计，尤其是预测分析表的计算机存储表示。在这里，一个重要的思维方法是：通过对实例的分析来明确解决问题的思路。例如，关于某文法的句子，给出相应的语法分析树与识别过程，观察分析栈的变化，明确解决的思路。对于推导，情况类似。建议读者充分利用 C 语言的置初值设置。

复习思考题

1．自顶向下分析技术的讨论前提是什么？

2．自顶向下分析技术应解决的基本问题是什么？怎样理解这个基本问题？

3．以自顶向下分析技术构造语法分析树时，为什么说是按照深度优先遍历顺序生成结点？

4．简述递归下降识别程序的实现思想。为什么说在递归下降识别程序组成中，为某个非终结

符号设计的子程序，处理的是相应句型中相对于该非终结符号的短语。

5．简述预测识别程序的实现思想。预测识别程序句型分析如何体现自顶向下分析技术？

习　题

1．设文法 G4.8[S]：S::=if E T | A　　T::=then S L　　A::=V=E

L::=else S　　V::=i　　E::=i | E+E

试为输入符号串 if i then i=i+i else i=i 按自顶向下分析技术构造语法分析树。要求：标明结点的实际建立顺序。

2．试对文法 G4.9[S]：

S::=Sab|Tc|h　　T::=dS|Tef|Sg

进行消去左递归文法等价变换。

3．试对文法 G4.10[S]：

U::=Uab|Vc|d　　V::=Uef|eV|Wg　　W::=Uhj|Vk|m

进行消去左递归文法等价变换。提示：先行判别是什么左递归。

4．试为题 1 中的文法 G4.8[S]构造递归下降识别程序。

5．试为文法 G4.11[C]：

C::={ L }　　L::=S | L;S

S::=A | I　　A::=V=E

I::=if E S　　E::=V|E+V　　V::=i|(E)

构造递归下降识别程序。要求：采用 C 型语言。

6．试应用预测识别程序对输入符号串 i = i – i + i 与 if(i)i = i + i – i 进行句型分析（预测分析表见表 4.3 ）。

7．试为题 1 中文法 G4.8[S]构造预测分析表。

第 5 章 语法分析——自底向上分析技术

本 章 导 读

第 4 章讨论的是自顶向下分析技术，本章讨论自底向上分析技术，内容包括：概况、LR 分析技术与其他的自底向上分析技术（算符优先分析技术）。读者应理解自底向上分析技术的 3 个讨论前提，熟练掌握移入—归约法。类似于自顶向下的预测识别程序，LR 识别程序是在 LR 分析表与分析栈的配合下完成句型分析。读者应熟练掌握应用 LR 识别程序进行句型分析，理解状态的概念与作用。LR 分析表有 3 种构造方法。本章重点讨论 SLR（1）分析表构造方法，读者应理解构造思路及相关的概念，如 LR(0)项集规范族与活前缀等，并掌握应用 LR 识别程序句型分析时，语法分析树的构造和计算机实现的方法与技巧，关于算符优先分析技术，主要理解应用该分析技术句型分析的思路及相关的概念，如算符优先关系、质短语与优先函数等。

5.1 自底向上分析技术概况

上一章讨论了自顶向下分析技术，它是以识别符号为根结点，试图从上向下构造语法分析树，使末端结点符号串正是输入符号串。本章讨论自底向上分析技术，就是从输入符号串出发，以它们为末端结点符号串，试图从下向上构造语法分析树，使根结点正是识别符号，如果构造成功，则可以识别出输入符号串是相应文法的句子，可以让相应的语法分析树作为输出，如果失败，则给出报错信息。按自顶向下分析技术，每一个分析步中，对句型中的一个非终结符号进行展开，即从相应结点向下构造分支，因此是一个推导的过程；而按自底向上分析技术，每一个分析步中对相应句型中的一个简单短语进行直接归约，也就是以此简单短语作为相应分支的分支结点符号串，建立分支及相应的分支名字结点，因此是一个归约的过程。

基于自底向上分析技术对输入符号串进行句型分析的算法，称为**自底向上识别算法**，基于自底向上识别算法的语法分析程序称为**自底向上识别程序**。目前熟知的分析技术有简单优先分析技术、算符优先分析技术与 LR 分析技术，相应地有简单优先识别程序、算符优先识别程序与 LR 识别程序。本章简略介绍算符优先分析技术，重点讨论 LR 分析技术。

如同对自顶向下分析技术的讨论，首先明确讨论的前提及自底向上分析技术要解决的基本问题，另外还明确基本实现方法。

5.1.1 讨论前提

自底向上分析技术的讨论前提有 3 个，即处理对象是符号串、基础文法是压缩了的上下文无

关文法，以及从左到右地进行规范分析。

① 处理对象。自底向上识别程序的处理对象是符号串。与自顶向下识别程序一样，标识符、整数与关键字等是终结符号，而不再是非终结符号，也即不再去考虑这些符号的结构如何、由什么组成等。语法分析的输入是词法分析的输出，是一种内部中间表示，即属性字序列，但为简单起见，也不失一般性，仍以符号串形式进行讨论，不考虑符号的表示细节。

② 基础文法。语法分析讨论的是相继的各个符号如何组成语法成分，其依据是文法规则。程序设计语言由它自身的特征所决定，它必定是以上下文无关文法来描述的。因此自底向上分析技术是基于上下文无关文法的。与自顶向下分析技术不同的是，自底向上分析技术是基于压缩了的上下文无关文法，也就是说，文法中任何一个规则 U::=u 都满足条件：U 出现在句型中，且 u 能推导到终结符号串，任何一个规则都能应用于句子的生成。

③ 分析方式。由于程序通常总是从上到下、从左到右地书写的，执行时一般也是按此书写的正常顺序逐个语句地执行。因此与自顶向下分析技术一样，自底向上分析技术也是从左到右逐个符号地执行。但是自底向上分析技术进行的是规范归约，确切地说，在每个分析步中被归约的简单短语的右部不包含非终结符号，因此分析过程产生的整个归约过程是规范的。这是由自底向上分析技术边扫描边归约的特征所决定的。

最后，回顾在第 2 章中所讨论的自底向上分析技术要解决的基本问题：在分析过程的每一分析步中必须解决。

① 如何找出进行直接归约的简单短语？

② 把所找出的简单短语直接归约到哪一个非终结符号？

某些分析技术归约的可能是特别引进的某种短语。

这里要强调的是：一种方法如果是分析技术，便必须能解决所要解决的基本问题。如果不能由这种方法解决基本问题，那就还只是一个实现方法，而不成为分析技术。

5.1.2 基本实现方法

在讨论特定的自底向上分析技术之前，首先讨论各种自底向上分析技术共同采用的基本实现方法：**移入—归约法**。下面以例子说明移入—归约法。

【例 5.1】 设有文法 G5.1[E]：

E::=E + E | E*E | (E) | i

假定输入符号串是 i+i*i，它显然是该文法的句子，可为它构造下列推导：

E = >E + E = >E + E*E = >E + E*i = >E + i*i=>i + i*i

相应的分析过程可列出如下，其中为了指明直接归约的 i 是哪一个，对 i 加了下标。

句型	句柄	归约用规则
$i_1 + i_2*i_3$	i_1	E:: = i
$E + i_2*i_3$	i_2	E:: = i
$E + E*i_3$	i_3	E:: = i
E + E*E	E*E	E:: = E*E
E + E	E+E	E:: = E + E
E		

当应用移入—归约法实现句型分析时，引进一个分析栈来存放输入符号以及直接归约成的非终结符号，分析过程大致如下。

步骤 1　初始时分析栈中下推入左端标志符号#。

步骤 2　查看分析栈顶的部分符号串是否形成句柄，即当前句型中的最左简单短语。

① 如果还没有形成句柄，把当前输入符号下推入分析栈（移入），输入指针推进指向下一个输入符号。

② 如果已形成句柄，则对它进行直接归约，把它替换（归约）为相应的非终结符号。

步骤 3　重复步骤 2，直到达到输入符号串的右端，即直到当前输入符号是右端标志符号#。当分析栈中仅包含两个元素，即左端标志符号#与识别符号，则成功，识别出句子，这时的分析动作称为接受。

用移入—归约法识别句子的过程如表 5.1 所示。

表 5.1

步　骤	分　析　栈	输　入	动　作	规　则
1	#	i + i*i#	移入	
2	#i	+i*i#	归约	E:: = i
3	#E	+i*i#	移入	
4	#E+	i*i#	移入	
5	#E + i	*i#	归约	E:: = i
6	#E+E	*i#	移入	
7	#E + E*	i#	移入	
8	#E + E*i	#	归约	E:: = i
9	#E + E*E	#	归约	E:: = E*E
10	#E + E	#	归约	E:: = E + E
11	#E	#	接受	

从表 5.1 可见，连同报错动作，移入—归约法共采取 4 个动作，即移入、归约、接受与报错。

① 移入。当分析栈顶的部分符号串未形成句柄时，读入一个输入符号，并把它下推入分析栈。

② 归约。当分析栈顶的部分符号串形成句柄时，对此句柄进行直接归约。注意：分析栈顶的符号是该句柄的右端符号。这时上退去分析栈顶形成句柄的部分符号串，然后把所归约成的非终结符号下推入分析栈。

③ 接受。当分析栈顶仅有两个元素，即左端标志符号#与识别符号，而输入也已达到右端标志符号#，这时说明分析成功，识别出输入符号串是相应文法的句子，执行接受动作，结束分析。

④ 报错。当识别程序察觉存在错误，因而输入符号串不是句子时，无法继续分析，可由出错处理子程序进行处理或停止分析。

表 5.1 充分体现了移入—归约法的具体执行过程，即分析过程中每一步，若分析栈顶的部分符号串还未形成句柄时，便把输入符号下推入分析栈（移入）；一旦分析栈顶的部分符号串形成句柄时便进行归约，如此继续，直到接受或报错，因此称为**移入—归约法**。

但是，移入—归约法本身不能确定它所采取的分析动作是移入还是归约，换句话说，移入—归约法本身不能确定分析栈顶的部分符号串是否已形成句柄。例如，表 5.1 中步骤 6，当时分析栈

中内容为#E+E，栈顶部分可以认为已形成句柄，但实际上并不作为句柄，而是采取移入动作，把输入符号*下推入栈，之所以会采取这样的动作，是因为人为的干预；至于归约到哪个非终结符号，也必须在人为的干预下确定。因此移入—归约法本身不能解决自底向上分析技术必须解决的两个基本问题，这两个基本问题将由后面讨论的分析技术来解决，移入—归约法只是各种自底向上分析技术共同采用的一种基本实现方法。

显然，移入—归约法这种基本实现方法，必须与基本实现工具—分析栈相配合才能实现句型分析。

5.2 LR(1)分析技术

在自顶向下的分析技术中，预测分析技术使预测识别程序在预测分析表的控制下进行句型分析，由分析栈顶元素与当前输入符号配对确定分析表元素，从而确定分析动作，避免了对子程序的调用，尤其是避免了对子程序的递归调用，使识别程序功效大大提高。本节讨论的也是基于分析表，由分析栈顶元素和当前输入符号配对确定分析表元素，从而确定分析动作的分析技术，但这是自底向上分析技术，这就是 LR(1)分析技术。

5.2.1 LR(1)分析技术与 LR(1)文法

1. LR(1)分析技术的基本思想

LR(k)分析技术的设想是：从左到右地扫描（读入）输入符号串中的符号，并且向前看确定个数（k 个）的符号，一点不回溯地识别给定的符号串。在对一个输入符号串进行句型分析时，每个分析步所采取的分析动作，由句型中的可能句柄左部的符号串及右部的向前看的 k 个符号所唯一地确定。

LR(k)分析技术基于 LR(k)文法。

一个文法是 LR(k)文法的条件是：当且仅当在句型识别过程中，任一句柄由其左部的符号串及其右部的 k 个符号所唯一地确定。

LR(k)文法的另一种说法是：句型识别过程中，分析动作总是由句柄左部的符号串及其右部向前看的 k 个符号所唯一地确定。因此，如果有两个不同的句型：

$$\alpha=x_1x_2\cdots x_nx_{n+1}\cdots x_{n+k}y_1y_2\cdots y_u$$

$$\beta=x_1x_2\cdots x_nx_{n+1}\cdots x_{n+k}z_1z_2\cdots z_v$$

其中，$u,v\geq 0$，且 x_{n+1}、…、x_{n+k}、y_1、y_2、…、y_u 与 z_1、z_2、…、z_v 都不是非终结符号。如果句型α中的句柄是 $x_{n-r}\cdots x_n$，按规则 $U::=x_{n-r}\cdots x_n$ 直接归约为 U（$0\leq r<n$），则句型β中的句柄也必定是 $x_{n-r}\cdots x_n$，它也将按规则 $U::=x_{n-r}\cdots x_n$ 直接归约为 U。

基于 LR(k)分析技术的识别程序称为 LR(k)识别程序。LR 识别程序示意图如图 5-1 所示。图中的输入符号串结束于$\#^k$，这是当扫描到输入符号串右端时向前看的 k 个符号，它由识别程序自动添入。分析栈的结构如图 5-2 所示，分析栈元素的结构如下图所示：

状态 S	文法符号 X

或

状态 S	文法符号序号 N

状态 S 可以反映分析过程的当前状况以及期望的向前看信息。整个分析栈的内容往往写成 $S_0X_1S_1X_2S_2\cdots X_mS_m$(其中省略左端标志符号#)，它记录了分析过程。尽管实现思想是：在进行每个

分析步的分析动作时，要由句型中可能句柄左部的符号串及向前看的 k 个符号所决定，事实上无需回溯去查看可能句柄左部的符号串，这就是因为状态的存在。

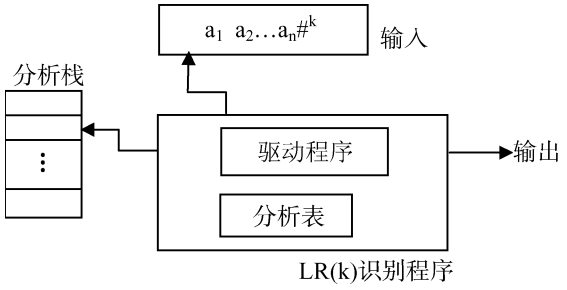


图 5-1

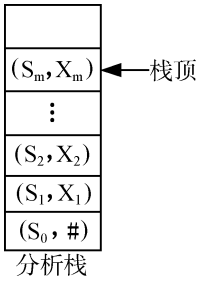


图 5-2

为什么状态能反映分析过程的当前状况以及期望的向前看信息？这是因为状态对应于 LR(k)项集，即由 LR(k)项组成的集合。圆点表示法的 LR(k)项形如：

$$[U_p \rightarrow X_{p1}X_{p2}\ldots X_{pj}.X_{p\,j+1}\ldots X_{pn_p};\alpha]$$

或者用脚标表示法表示成：

$$[p, j; \alpha]$$

其中 p 是重写规则 $U_p :: = X_{p1}X_{p2}\ldots X_{pj}X_{pj+1}\ldots X_{pn_p}$ 的序号，j 指明按规则 p 进行归约时已处理到的规则右部位置，α是向前看信息，即向前看的 k 个符号。

本书将仅讨论 k=1 的情况，即 LR(1)分析技术与 LR(1)文法。

例如，文法 G5.2[E]：

E:: = E + T　　E:: = T　　T:: = T*F　　T:: = F　　F:: = (E)　　F:: = i

它是 LR(1)文法，可为它构造 LR(1)分析表如表 5.2 所示。

表 5.2

状态	ACTION						GOTO		
	+	*	(	)	i	#	E	T	F
0			S4		S5		1	2	3
1	S6					acc			
2	r2	S7		r2		r2			
3	r4	r4		r4		r4			
4			S4		S5		8	2	3
5	r 6	r6		r6		r6			
6			S4		S5			9	3
7			S4		S5				10
8	S6			S11					
9	r 1	S7		r1		r1			
10	r 3	r3		r3		r3			
11	r 5	r5		r5		r5			

从表 5.2 可见，LR(1)分析表由 ACTION 部分与 GOTO 部分组成。ACTION 部分指明当分析栈顶的状态与当前输入符号匹配时执行的动作，S 表示移入（Shift），r 表示归约（Reduce），acc 表示接受（Accept），空白元素表示出错。例如，按表 5.2 中的 LR 分析表，分析栈顶状态是

0 时，若输入符号是 i，则执行动作 S5，这是移入动作，即把状态 5 与当前输入符号 i 一起下推入分析栈；若这时的输入符号是(，则执行动作 S4，这也是移入动作，把状态 4 与当前输入符号(一起下推入分析栈。当分析栈顶的状态是 2 时，若输入符号是 + 、)或#时执行动作 r2，即按规则 2（E:: = T）对分析栈顶构成句柄的符号串 T 进行直接归约；若此时的输入符号是*，则执行动作 S7，把状态 7 与当前输入符号*下推入分析栈。当分析栈顶状态是 1，而输入符号是右端标志符号#时，执行动作 acc，即接受，这时因识别出输入符号串是句子而结束分析。如果分析栈顶的状态是 1，而输入符号是*或 i 等时，确定的元素是 ERROR（空白），表明出错，将给出报错信息，并作出相应处理。

GOTO 部分指明进行归约时状态的转换，确切地说，当分析栈顶的部分符号串构成句柄时，把构成句柄的符号串上退去，并把此时分析栈顶的状态与所归约成的非终结符号匹配，所确定 GOTO 部分的元素作为下一当前状态，这时把该状态连同所归约成的非终结符号一起下推入分析栈，作为分析栈新栈顶元素。例如，当分析栈中的内容是 0E1 + 6T9*7F10，而输入符号是#时，分析栈顶的部分符号串 T*F 构成句柄，这时按表 5.2 中的 LR 分析表 ACTION 部分，由状态 10 和输入符号#确定的动作是 r3，即按规则 3：T:: = T*F 归约，把 T*F 归约成 T，因此上退去分析栈顶的部分符号串 T*F（连同相应的状态），分析栈顶状态现在是 6，归约成的非终结符号是 T，从 GOTO 部分查得相应元素是 9，因此把状态 9 与非终结符号 T 一起下推入分析栈，分析栈内容成为 0E1 + 6T9。

从分析表可见，任何一个状态与任何一个文法符号（包括右端标志符号#）相匹配时，唯一地决定了所应执行的动作或状态的转换。一个 LR(1)文法，其 LR(1)分析表必定是无冲突的，LR(1)分析表无冲突的文法才是 LR(1)文法。

2. 应用 LR 分析技术句型分析

下面讨论 LR(1)分析技术如何应用 LR(1)分析表进行句型分析。

类似于 LL(1)识别程序，LR(1)识别程序对一个输入符号串进行句型分析时，在每一个分析步执行的动作，都是由分析栈顶元素与当前输入符号相匹配所确定的分析表元素指明的动作，所不同的是，现在分析栈元素指的是状态，确切地说，是由分析栈顶元素中的状态与当前输入符号匹配、确定分析表 ACTION 部分的元素，即 ACTION[分析栈顶元素中状态][当前输入符号]确定所执行的动作，完成移入或归约动作，进行状态的转换，直到最后执行接受或报错动作。

用数组数据结构实现分析栈，栈顶计数器设为 top，应用 LR 分析技术进行句型分析的识别过程大致如下。

步骤 1　置初值（包括 top = 0 等），把{S_0,#}下推入栈，其中 S_0 是初始状态。

步骤 2　把 R 置为下一（当前）输入符号。

步骤 3　执行动作 ACTION[分析栈[top].状态][R]，有以下几种可能：

- 动作是 Si，即移入，把状态 i 与输入符号 R 一起下推入分析栈，重复步骤 2。
- 动作是 rj，即归约，按规则 j 进行归约，把分析栈顶形成句柄的部分符号串（连同状态）上退去，以这时的分析栈顶状态与所归约成的非终结符号匹配，查 GOTO 部分，得到将转换成的状态，把此状态与所归约成的非终结符号一起下推入分析栈，重复步骤 3。
- 动作是 acc，即接受，识别出输入符号串是相应文法的句子而结束分析。
- 动作是 ERROR（空白元素），给出报错信息，进行出错处理。

LR 识别程序控制流程示意图见图 5-3。

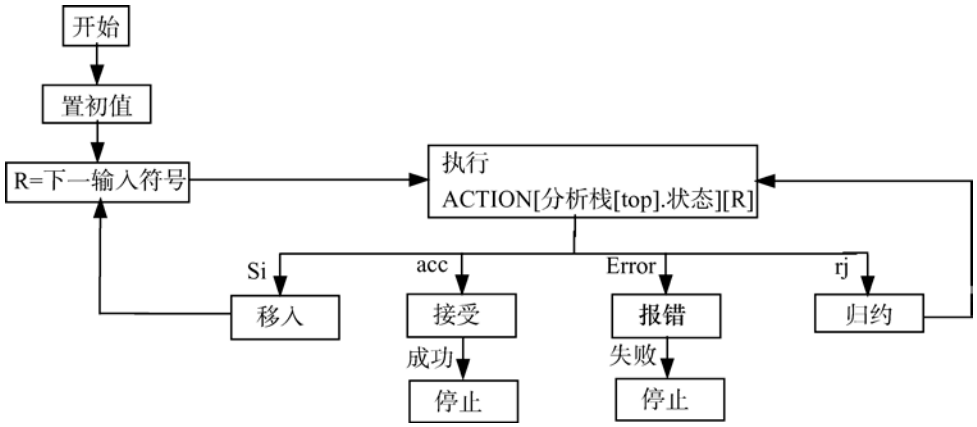


图 5-3

LR 识别程序（驱动程序）可写出如下（C 型语言）。

```
/* LR 识别程序驱动程序*/
void main( )
{ 置初值;/* 包括 top=0; */
  top=top+1;
  分析栈[top]={0, #}; /* (初始状态 0，左端标志符号#) */
  tagS=true;  /* tagS: 继续分析标志*/
  while (tagS)
  { R=下一输入符号;
    tagR=true; /* tagR: 归约时继续分析标志 */
    while (tagR)
    {  switch(ACTION[分析栈[top].状态][R])
       {  case  SHIFT:   /* 移入：Si  */
            top=top+1;
            分析栈[top]={ i, R}; /* (状态 i, 输入符号 R)*/
            break;
          case  ACCEPT:  /*  接受：acc  */
            输出:“识别出句子”;
            tagS=false;
            break;
          case  ERROR:   /* 报错: 空白元素  */
            输出:报错信息; 调用出错处理子程序;
            tagS=false;
            break;
          case  REDUCE:  /* 归约：rj  */
            上退去分析栈顶构成句柄的相应若干个元素;
            U=规则 j 的左部非终结符号的序号;
            S= GOTO[分析栈[top].状态][U];
            top=top+1;  分析栈[top]={S，U};
            continue;
        }
       tagR=false;
    }
  }
```

}

不言而喻，这里的 LR 识别程序（驱动程序）控制流程示意图与 LR(1)识别程序（驱动程序）对任何一个 LR(1)文法都适用，都可以利用这同一个 LR(1)识别程序驱动程序进行句型分析。不同的 LR(1)识别程序区别仅在于 LR(1)分析表不同。

【例 5.2】 试关于文法 G5.2[E]应用 LR(1)分析技术对输入符号串 i*(i+i)进行句型识别。

文法 G5.2[E]的 LR 分析表见表 5.2，以表的形式写出对 i*(i+i)的识别过程如表 5.3 所示。

表 5.3

步　骤	栈	输　入	动作	说　明
0	0	i*(i + i)#	S5	移入 i
1	0i5	*(i + i)#	r6	归约 i　F:: = i
2	0F3	*(i + i)#	r4	归约 F　T:: = F
3	0T2	*(i + i)#	S7	移入*
4	0T2*7	(i + i)#	S4	移入(
5	0T2*7(4	i + i)#	S5	移入 i
6	0T2*7(4i5	+i)#	r6	归约 i　F:: = i
7	0T2*7(4F3	+i)#	r4	归约 F　T:: = F
8	0T2*7(4T2	+i)#	r2	归约 T　E:: = T
9	0T2*7(4E8	+i)#	S6	移入+
10	0T2*7(4E8 + 6	i)#	S5	移入 i
11	0T2*7(4E8 + 6i5	)#	r6	归约 i　F:: = i
12	0T2*7(4E8 + 6F3	)#	r4	归约 F　T:: = F
13	0T2*7(4E8 + 6T9	)#	r1	归约 E + T　E::=E + T
14	0T2*7(4E8	)#	S11	移入)
15	0T2*7(4E8)11	#	r5	归约(E)　F:: = (E)
16	0T2*7F10	#	r3	归约 T*F　T:: = T*F
17	0T2	#	r2	归约 T　E:: = T
18	0E1	#	acc	接受

因此，符号串 i*(i + i)是文法 G5.2[E]的句子。

为了使读者更容易掌握 LR(1)识别程序的分析过程，对表 5.3 中前几个步骤加以说明。

初始时（步骤 0），分析栈中下推入初始状态 0 与符号#，但通常省略不写出符号#。输入为符号串 i*(i + i)#，其中的#由识别程序自动添入。由分析表知，ACTION[0][i] = S5，因此执行移入动作，把状态 5 与输入符号 i 一起下推入分析栈，此时分析栈成为 0i5，输入的其余部分是*(i + i)#。这时（步骤 1）分析栈顶状态是 5，输入符号是*，查表 ACTION[5][*] = r6，按规则 6：F:: = i 归约，把分析栈顶构成句柄的部分符号串 i 直接归约成 F，这时是归约动作，上退去分析栈顶构成句柄的部分符号串 i（连同状态），分析栈顶状态又是 0，而所归约成的非终结符号是 F，由 GOTO[0][F] = 3，状态转换为 3，把状态 3 与所归约成的非终结符号 F 一起下推入分析栈，分析栈成为 0F3（步骤 2）。这时重复取到 ACTION 部分的元素，即 ACTION[3][*] = r4，为归约动作，按规则 4：T:: = F 归约，把分析栈顶构成句柄的部分符号串 F 归约成 T。类似地，把分析栈顶构成句柄的部分符号串 F（连同状态）上退去，分析栈顶状态是 0，所归约成的非终结符号是 T，由 GOTO 部分查得 GOTO[0][T] = 2，因此，把状态 2 与所归约成的非终结符号 T 一起下推入分析栈，

分析栈内容成为 0T2（步骤 3），输入符号串未处理部分仍是*(i + i)#。如此继续，直到步骤 18 时分析栈内容是 0E1，分析栈顶状态是 1，这时的输入符号是#，由 ACTION[1][#] = acc，接受动作，因此识别出输入符号串 i*(i + i)是文法 G5.2 的句子。

从例 5.2 可见，

① 每个分析步所执行的动作，完全由分析表中 ACTION 部分的元素决定，而该元素由分析栈顶的状态与当前输入符号匹配所确定。分析栈中的文法符号并不起作用，把文法符号放在分析栈中，主要是为了易读性好，有助于了解 LR 识别程序的识别过程。

② 在执行归约动作时才涉及文法规则，这时需要的仅是左部非终结符号与规则右部符号串的长度，即上退去分析栈顶构成句柄的部分符号串时，只需了解规则右部的长度，按长度上退去分析栈顶相应个数的元素便可。

③ 如果略去分析栈中元素的状态部分，分析栈中的符号串（从分析栈底到栈顶）与正待扫描的输入符号串并置在一起将构成一个句型，例如，步骤 2 时的 i 与*(i + i)，步骤 6 时的 T*(i 与 + i)，而且这种句型还是规范句型，因为句柄总是在分析栈的栈顶部分形成，句柄右端的符号是分析栈的栈顶符号，在分析栈顶符号的右端不可能出现非终结符号。

通常引进活前缀与构型两个概念。

① **构型**。分析栈从栈底到栈顶的内容，连同输入符号串当前未处理部分（其余部分），中间用一个竖线隔开，称为构型。例如，从表 5.3 可知，初始状态 S_0，输入符号串是 i*(i + i)，则步骤 0 时的构型是 S_0|i*(i + i)#，步骤 1 时的构型是 S_0iS_5|*(i + i)#，而识别出句子时的构型，即步骤 18 时的构型，是 S_0ES_1|#。

② 规范句型的**活前缀**。一个规范句型中这样一个前缀子符号串，它不包含句柄之后的任何符号，称为该规范句型的活前缀。例如，表 5.3 中步骤 1 时分析栈从底到顶的符号串（不计状态）是 i，这是规范句型 i*(i + i)的活前缀，步骤 5 时分析栈从底到顶的符号串 T*(是规范句型 T*(i + i)的活前缀，又如步骤 12 时，分析栈从底到顶的符号串 T*(E + F 是规范句型 T*(E + F)的活前缀。

概括起来，构型刻画了句型分析过程中一个分析步的状况，而分析栈中从底到顶的符号串总是相应规范句型的活前缀，在其右部添加若干个输入符号时将可能构成规范句型。如果分析栈中从底到顶的符号串不构成相应规范句型的活前缀，表明输入符号串不是相应文法的句子。

下面讨论在应用 LR(1)分析技术句型分析时生成语法分析树的问题。

3. 应用 LR(1)分析技术句型分析时生成语法分析树

语法分析不仅要识别一个输入符号串是否是某文法的句子，而且要生成相应的内部中间表示作为输出，提供给下一阶段（语义分析）作为输入。这种中间表示可以是语法分析树，也可以是推导等。这里讨论如何建立语法分析树。

如前所述，语法分析树的结构由结点之间的关系确定，任一结点的位置只需由父结点、左兄结点与右子结点确定，因此语法分析树的数据结构可设计如下：

```
typedef struct
{  int  结点序号;      int  文法符号序号;
   int  父结点序号;    int  左兄结点序号;   int  右子结点序号;
} 语法分析树结点类型;
语法分析树结点类型  语法分析树[MaxNodeNum];
```

如前所述，LR(1)分析技术是自底向上分析技术，因此，在每一分析步时，以所找到句柄中包含的符号串作为分支结点符号串建立分支，并建立分支名字结点。确切地说，为句柄中包含的符号串相

应的结点建立兄弟关系，另外建立一个结点作为分支名字结点，它与上述那些兄弟结点建立父子关系。

为建立语法分析树，LR(1)识别程序可修改如下。需注意的是，为包含更多的信息，把分析栈元素中的文法符号序号改为结点序号，即分析栈元素的数据结构修改成如下所示：

状态	结点序号

用数组结构实现分析栈，栈顶计数器为 top，可写出 LR 识别程序（驱动程序）如下：

```
/* 建立语法分析树的 LR 识别程序驱动程序*/
void main ( )
{  置初值; /* 栈计数器 top=0，结点序号 N=0 等  */
   为输入符号串中各个符号依次建立结点;
   top=top+1;  分析栈[top]={初始状态 0, 结点序号 0};
   tagS=true; /* tagS: 继续分析标志 */
   while(tagS)
   { R=下一输入符号;
     tagR=true; /* 归约时继续分析标志 */
     while(tagR)
     {  switch(ACTION[分析栈[top].状态][R])
        {  case SHIFT:      /* 移入：Si */
              top=top+1; 分析栈[top]={i, 符号 R 所相应的结点序号};
              break;
           case  ACCEPT:  /* 接受：acc */
               输出:“识别出句子”;
               tagS=false;
               break;
           case  ERROR:  /* 报错：空白元素 */
               输出报错信息;调用出错处理子程序;
               tagS=false;
               break;
           case  REDUCE:  /* 归约：rj */
               m=规则 j 的右部长度;
               if(m==0)
               { /*规则右部为ε，为ε建立结点；*/
                 N=N+1; 语法分析树[N]={N,ε的序号 0,0,0,0};
                 top=top+1; 分析栈[top]={ 0,  N}; /* 状态序号可为任意 */
                 m=1;
               }
               右子结点序号=分析栈[top].结点序号;
               U=规则 j 的左部非终结符号的序号; /* 加 100 */
               N=N+1; 语法分析树[N]={N,U,0,0,右子结点序号 };
               左兄结点序号=0;
               for(j=1; j<=m; j++)
               { 分支结点序号=分析栈[top–(m–j)].结点序号;
                 语法分析树[分支结点序号].父结点序号=N;
                 语法分析树[分支结点序号].左兄结点序号=左兄结点序号;
                 左兄结点序号=分支结点序号;
               }
               top=top–m;
               NewState=GOTO[分析栈[top].状态][U–100];
```

```
                top=top+1;  分析栈[top]={ NewState, N};
                continue;
            } /*switch */
            tagR=false;
          }  /* while(tagR) */
            }  /* while(tagS) */
            输出语法分析树;
}
```

C 语言中结构类型变量的赋值方式。赋值语句：

```
分析栈[top]={ 初始状态 0, 结点序号 0};
```

实际上需由下列两个赋值语句实现：

```
分析栈[top].状态=初始状态 0;
分析栈[top].结点序号=结点序号 0;
```

其他的类似。

4. LR(1)分析表的构造

如同 LL(1)分析技术的关键是 LL(1)分析表的构造，LR(1)分析技术的关键是 LR(1)分析表的构造。

为了构造 LR(1)分析表，首先需理解状态是什么、状态转换的含义又是什么。在 LR(1)分析技术中，状态对应于 LR(1)项所组成的集合，即 LR(1)项集。如前所述，LR(1)项呈以下形式：

$$[U_p \rightarrow X_{p1}X_{p2}\cdots X_{pj}\cdot X_{pj+1}\cdots X_{pn_p};\ a]$$

它表示正按规则 p: U_p:: = $X_{p1}X_{p2}\cdots X_{pn_p}$ 进行归约，当前已处理到 X_{pj}，即尚未被扫描的输入符号串部分要（广义）归约到 X_{pj+1}，当输入符号串已归约到 U_p 时，则 U_p 在句型中的后继符号将是符号 a。显然，这个符号 a 不是任意的，而是必须可以在句型中紧随 U_p 之后出现的终结符号。例如，一个句型中的句柄是 E + T 时，在 E + T 被归约成 E 后，句型中紧随 E 之后出现的符号只能是 + 、)与#，不能是*、(或 i 或其他。因此，可以有[E→E. + T; a]、[E→E + .T; a]与[E→E + T.; a]等，其中的 a 是 + 、)或#。图 5-4 展示了这 3 种情况。

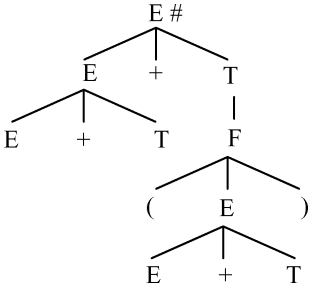


图 5-4

现在看 LR(1)项[E→E + .T;a]，如果分析过程处于这一情况，即按规则 1:E:: = E + T 归约时，输入符号串的左边部分已归约到 E + ，当前待扫描的输入符号串部分正待归约到非终结符号 T。由于 T:: = T*F 与 T:: = F，不言而喻，要归约到 T 必须先归约到 T*F 或 F。而如果要归约到 F，又必须先(广义)归约到(E)或 i，因此 LR(1)项[T→.T*F; a]与[T→.F; a]都同[E→E + .T; a]紧密相关。而对于其中的[T→.T*F; a]必须把尚待扫描的输入符号串部分归约成 T*F，即先归约成 T，这时句型中紧随 T 的符号是*，于是有[T→.T*F; *]。类似地，相关的又有[T→.F; *]，为归约成 F，又有[F→.(E); *]与[F→.i; *]等。把所有这样的相关 LR(1)项放入一个集合中，即 LR(1)项集。

让 LR(1)项集对应于状态，状态之间的转换实质上对应于项集的后继关系，从而引进后继项的概念。

如果有一个 LR(1)项：[U→u.Av; a]，其中 A∈V_N∪V_T，则其后继项是[U→uA.v; a]，后继项是把项中圆点右移一个符号位置所得的项。从句型识别的角度看，其含义很明显，即当扫描过的输入符号部分已与规则右部的 u 相匹配（即归约成 u）时，应期望后继的那些输入符号与 u 的后继符号 A 相匹配，

因此[U→u.Av; a]所在项集对应的状态，将转换到其后继项所在项集（即后继项集）对应的状态。

如何确切地构造各个 LR(1)项集？通常的做法是，从初始项出发构造初始项集，从初始项集开始逐个构造后继项集，再从后继项集构造后继项集，最终构造出一切 LR(1)项集。让各个项集对应于状态，后继关系对应于状态转换关系，然后填 LR(1)分析表，即得所求。

LR(1)分析技术应用来进行句型分析时，每个分析步都要固定向前看 1 个符号，因此 LR(1)项集个数比较多，后继关系也比较复杂，导致 LR(1)分析表的构造工作量庞大，在早期难以使 LR(1)分析技术实用化。

为此从两个方面进行改进，一是自动生成 LR(k)分析表，另一是简化分析表构造方法。当前除了初始提出的规范的 LR(k)分析表构造方法外，还有简单 LR(k)（SLR(k)）方法与前视 LR(k)（LALR(k)）方法。还是考虑 k = 1 的情况。

SLR(1)方法的基本思想是：如果不向前看一个符号能确定分析动作，就不向前看任何符号，如果不能确定，再向前看一个符号。因此，基于 LR(0)项集。

LR(0)项，因为不向前看任何符号，表示为：U→u.Av。例如，对于规则 E:: = E + T，可以有如下 4 个 LR(0)项：

E→.E + T　　E→E. + T　　E→E + .T　　E→E + T.

圆点在右端的，如 E→E+T.，称为完备项，其他的是不完备项；圆点之后是终结符号的，如 E→E. + T，称为移入项；圆点之后是非终结符号的，如 E→E + .T，称为待约项。通常为了使识别符号 Z 仅作为一个规则的左部，引进新的符号 Z′作为识别符号，并引进规则 Z′::=Z#。Z′→Z#.称为接受项，其他的完备项称为归约项。因此 LR(0)项中，完备项包括接受项与归约项，不完备项包括移入项与待约项。

对于文法 G5.2[E]，初始项是 Z→.E#，与其相关的一切项组成初始项集 I_0，显然，

I_0={ Z→.E#, E→.E+T, E→.T, T→.T*F, T→.F, F→.(E), F→.i }

从 I_0 中的项来产生后继项集，圆点之后的符号是 E，称为 E_后继，圆点之后的符号是 T，称为 T_后继，类似地还有 F_后继与(_后继等。确切地说，I_0 的 E_后继是一个项集，它包含 I_0 中圆点之后符号为 E 的项的后继项及一切相关项。因此，I_0 的 E_后继 = {Z→E.#, E→E. + T}，这时因圆点后是符号#与 + ，都不是非终结符号，它们并无相关的项。类似地，T_后继 = { E→T., T→T.*F}，其他还有 F_后继、(_后继与 i_后继。圆点在规则右部符号串右端的后继称为#_归约后继。例如上面 T_后继的#_归约后继，即#E:: = T 归约后继。为了用一种简便而明确的方式表达，采用表的形式，如表 5.4 所示，其中的闭包指的就是从后继项引出的相关各个项。

表 5.4

项集编号	项集名	项		后继关系
0	初始项	基本项集	Z→.E#	E → 1
		闭包集合	E→.E+T E→.T T→.T*F T→.F F→.(E) F→.i	T → 2 F → 3 (→ 4 i → 5
1	E_后继	基本项集	Z→E.#	# → 12

			E→E.+T	+→ 6
2	T_后继	基本项集	E→T. T→T.*F	#E::=T→ 13 *→ 7

续表

项集编号	项集名	项		后继关系
3	F_后继	基本项集	T→F.	#T::=F→ 13
4	(_后继	基本项集	F→(.E)	E→ 8
		闭包集合	E→.E+T E→.T T→.T*F	T→ 2
			T→.F	F→ 3
			F→.(E)	(→ 4
			F→.i	i→ 5
5	i_后继	基本项集	F→i.	#F::=i→ 13
6	+_后继	基本项集	E→E+.T	T→ 9
		闭包集合	T→.T*F	
			T→.F	F→ 3
			F→.(E)	(→ 4
			F→.i	i→ 5
7	*_后继	基本项集	T→T*.F	F→ 10
		闭包集合	F→.(E)	(→ 4
			F→.i	i→ 5
8	E_后继	基本项集	F→(E.)	)→ 11
			E→E. + T	+→ 6
9	T_后继	基本项集	E→E + T.	#E::=E+T→ 13
			T→T.*F	*→ 7
10	F_后继	基本项集	T→T*F.	#T::=T*F→ 13
11	) _后继	基本项集	F→(E).	#F::=(E)→ 13
12	#_后继	基本项集	Z→E#.	#Z::=E#→ 13

从表 5.4 可见，当一个项集中包含项 U→u.Xv 时它便有 X_后继，这个 X_后继是一个项集，它由基本项集与闭包集合组成，其中的基本项集中的项都形如 U→uX.v，即圆点右移一个符号位置形成的后继项。当 v 中的第一个符号是非终结符号，例如 W，且 W:: = w 是文法规则时，将有相关的项：W→.w，这些相关的项又有相关的项，直到最后不再有相关的项加入，便得到闭包集合。例如，项集 0，因为项 F→.(E)，而有(_后继，(_后继的基本项集包含项 F→(.E)，由于 E 是非终结符号，且 E:: = E + T 与 E:: = T，有相关的项 E→.E + T 与 E→.T，由于圆点之后的 T 是非终结符号，又有相关的项：T→.T*F 与 T→.F，

类似地，又有F→.(E)与F→.i，这时圆点之后是终结符号(与i，再也没有相关的项。得到闭包集合{E→.E＋T, E→.T, T→.T*F, T→.F, F→.(E), F→.i }。事实上，基本项集中的项也是闭包集合的元素。

一般地项集闭包定义如下：

Closure (I)=I∪{W→.w | U→u.Wv ∈ Closure (I), 且 W::=w 是文法规则}

因此，一个 X_后继事实上是其基本项集中项的闭包。例如，项集 I_0 的(_后继是项集 4，即

Closure ({F→(.E)})={ F→(.E), E→.E+T, E→.T, T→.T*F, T→.F, F→.(E), F→.i }

请注意，项集 0 的 T_后继与项集 6 的 T_后继不是相同的项集。如果两个项集相同，则它们所包含的项必须完全相同，即个数与内容都相同。然而，项集 0 的 T_后继中包含了项 E→T.，而项集 6 的 T_后继中包含了项 E→E＋T.，尽管都包含 2 个项，且其中一个项都是 T→T.*F，但两者还是不同的，因此前者为项集 2，后者为项集 9。类似地。项集 0 的(_后继不能与项集 0 相同，而是项集 4。

另外请注意，项集 1 的#_后继是由 Z→E#.所组成的项集 12。当分析过程处于此项集相应的状态时，输入符号必定是右端标志符号#，这时识别出输入符号串是相应文法的句子。然而，当分析过程处于项集 1 相应的状态，而输入符号是#，已识别出句子，因此无需项集 12。但为保持项集序号的连续性，把项集 1 的#_后继的项集序号定为 12，而不是 6。尽管重写规则众多，但#归约_后继仅一个，且它并不对应于任何状态，不反映在 LR 分析表中，因此，为#_归约后继取最大的编号 13。

从表 5.4 可见，项集为 0、1、2、…、11 与 12，共 13 个，这 13 个项集构成的集合称为 LR(0)项集规范族。一般地，LR(0)项集规范族的构造步骤如下：

步骤 1　以初始项集 S_0=Closure({Z′→.Z#})作为文法 G 的初始 LR(0)项集，其中 Z′是包含规则 Z′::=Z#的增广文法的识别符号。

步骤 2　为文法 G 的 LR(0)初始项集构造一切后继项集，得到文法 G 的 LR(0)项集。

步骤 3　为已构造的 LR(0)项集构造一切后继项集，生成 G 的新的 LR(0)项集。

步骤 4　重复步骤 3，直到再无新的 LR(0)项集产生。

简而言之，从文法的初始项出发构造初始项集，从初始项集构造后继项集，再为这些后继项集构造新的后继项集，直到再无新的项集生成。最终所得的全部 LR(0)项集构成文法 G 的 LR(0)项集规范族。

通常令 LR(0)项集规范族中的每一个项集对应于一个状态，而项集之间的后继关系对应于状态之间的状态转换关系，这样 LR(0)项集规范族将对应于称为特征有穷状态机（CFSM）的自动机，其开始状态对应于初始项集，终止状态对应于#_归约后继项集。例如，文法 G5.2 [E]的特征有穷状态机如图 5-5 所示。

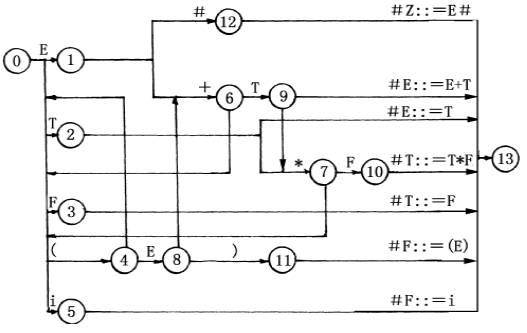


图 5-5

说明如下：当从开始状态出发，沿着弧序列行进到任何一个状态，相应的弧上标记所组成的

符号串是一个规范句型的活前缀。例如，(0)E→(1) + →(6)(→(4)E→(8) + →(6)T→(9)* →(7)，弧上标记符号串是 E + (E + T*，它是一个活前缀，在其后再添加一些适当的输入符号，例如 i)，就可构成一个规范句型：E + (E + T*i)。

请注意图 5-5 中的状态 2 与状态 9。其他的状态所对应的 LR(0)项集中包含的项集，或者全是不完备项（移入项或待约项），或者全是归约项，仅状态 2 与状态 9 对应的 LR(0)项集中既包含完备项又包含不完备项，即既有移入项又有归约项。这表明其他各状态下，与输入符号匹配时将执行确定的分析动作：或移入或归约。只有这两个状态，不能确定是移入还是归约，因此状态 2 与状态 9 称为不适定状态。

SLR(1)方法的基本思想也就是以简单的方法解决这种不适定状态问题，即向前看 1 个符号，从而决定是移入还是归约。

如果一个文法 G 的 CFSM 中不包含不适定状态，则该文法 G 是 LR(0)文法。这表示在任何状态下，不需要向前看任何符号便可确定执行什么分析动作。当包含不适定状态时，对于这些不适定状态，需要向前看一个符号以确定是移入还是归约。如果所有不适定状态都只需向前看一个符号就能唯一地确定分析动作，则该文法 G 是 SLR(1)文法。

为了判定对于某个不适定状态，向前看一个符号能否唯一地确定分析动作，引进简单向前看 1 集合。一般地，不适定状态的状态转换有两类：对应于移入项 U→u.Xv 的 X_转换（文法符号 X 下的状态转换）；对应于归约项 U→u.的#U::＝u 转换（按规则 U::＝u 归约的状态转换）。以状态 9 为例，它相应的 LR(0)项集是{E→E + T., T→T.*F}，它包含归约项 E→E + T.与移入项 T→T.*F，因此，状态 9 有两类状态转换，即#E::=E+T 转换与*_转换。是移入还是归约，决定于向前看的一个符号 a，若 a 属于 X_转换的简单向前看 1 集合，则进行移入，否则，a 属于#U::=u 转换的简单向前看 1 集合，则进行归约。

X_转换的简单向前看 1 集合是{X}，而#U::＝u 转换的简单向前看 1 集合是 $F_T^1(U)$ = Follow(U)。如前所述，Follow(U)是如下的集合：

$$\text{Follow}(U)=\{T \mid Z=>^* \ \ldots UT\ldots,\ T\in V_T\}$$

即 Follow(U)中的元素是出现在包含 U 的句型中，紧随 U 之后的终结符号。对于状态 9，X_转换是*_转换，简单向前看 1 集合是{*}，而#U::＝u 转换是#E::＝E + T 转换，$F_T^1(E)$ = { + ,),#}。因此，向前看的 1 个符号是*时执行移入动作，向前看的 1 个符号是 + 、)或#时执行归约动作。可以引进过渡状态 7′与 13′。当向前看的 1 个符号是*时过渡到状态 7′，然后读入输入符号（*），状态转换到状态 7；当向前看的 1 个符号是 + 、)或#时过渡到状态 13′，然后读入输入符号（ + 、)或#），状态转换到状态 13。示意图如图 5-6 所示。

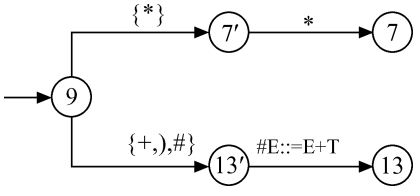


图 5-6

由此可见，如果一个不适定状态向前看 1 个符号就能唯一地决定分析动作，那么关于它的各个 X_转换与#U::=u 转换的简单向前看 1 集合必须互不相交。一般地，SLR(1)文法的定义可给出如下：

如果一个上下文无关文法 G 是 SLR(1)文法，当且仅当其 CFSM 每个不适定状态的各个 X_转换和#U::＝u 转换的简单向前看 1 集合互不相交。

一个文法的 CFSM 中存在不适定状态时，若不适定状态的 X_转换与#U::=u 转换的简单向前看 1 集合相交，则向前看一个符号时不能确定执行什么分析动作，就必须至少再向前看 1 个符号（即至少向前看 2 个符号）才能确定分析动作，这就不是 SLR(1)文法。

现在可以归纳出 SLR(1)分析表构造步骤如下。

步骤 1　从原有文法构造增广文法，即增加新的符号 Z′作为识别符号，并添加规则 Z′::=Z#。

步骤 2　以如表 5.4 所示的表的形式构造 LR(0)项集规范族，令项集序号作为状态名。

步骤 3　按下列规则填 SLR(1)分析表，考虑项集编号为 i 的项集：

① 对于 X_后继，$X \in V_N$，置 GOTO[i][X] = j，这里 j 是该 X_后继的项集序号。

② 对于 X_后继，$X \in V_T$，置 ACTION[i][X] = Sj，这里 j 是该 X_后继的项集序号。

③ 对于#U:: = u 后继，置 ACTION[i][a] = rj，这里 j 是规则 U:: = u 的序号，而 $a \in F_T^1(U)$。

④ 对于初始项 Z′→.Z#的 Z_后继的#_后继，置 ACTION[k][#] = acc，这里，k 是 Z′→.Z#的 Z_后继的项集序号。

⑤ 凡不能由上述 4 个规则确定的分析表元素，全置为报错标志（空白元素）。

如果由上述步骤构造的分析表 ACTION 部分与 GOTO 部分中的所有元素都仅一个值，即不发生冲突，因而分析表中每一个元素决定唯一的分析动作，则此分析表称为 SLR(1)分析表，具有 SLR(1)分析表的文法称为 SLR(1)文法。

表 5.2 中的分析表就是按此步骤从表 5.4 所得。

【例 5.3】　设有文法 G5.3[S]：

1：　S:: = if(E)S　　2: S:: = V = E　　3: V: = i

4：　E:: = (E)　　5: E:: = i

试为其构造 SLR 分析表。

构造 SLR 分析表的步骤如下。

步骤 1　首先构造增广文法，引进新的符号 Z 作为识别符号，添加规则 Z::=E#。

步骤 2　以表的形式构造 LR(0)项集规范族，如表 5.5 所示。

表 5.5

项集编号	项集名	项		后继关系
0	初始项	基本项集	Z→.S#	S → 1
		闭包集合	S→.if(E)S	if → 2
			S→.V=E	V → 3
			V→.i	i → 4
1	S_后继	基本项集	Z→S.#	# → 15
2	if_后继	基本项集	S→if.(E)S	(→ 5
3	V_后继	基本项集	S→V.=E	= → 6
4	i_后继	基本项集	V→i.	#V::=i → 16
5	(_后继	基本项集	S→if(.E)S	E → 7
		闭包集合	E→.(E)	(→ 8

			E→.i	i → 9

续表

项集编号	项集名	项		后继关系
6	=_后继	基本项集	S→V=.E	E → 10
		闭包集合	E→.(E)	(→ 8
			E→.i	i → 9
7	E_后继	基本项集	S→if(E.)S	) → 11
8	(_后继	基本项集	E→(.E)	E → 12
		闭包集合	E→.(E)	(→ 8
			E→.i	i → 9
9	i_后继	基本项集	E→i.	#E::=i → 16
10	E_后继	基本项集	S→V=E.	#S::=V=E → 16
11	) _后继	基本项集	S→if(E).S	S → 13
		闭包集合	S→.if(E)S	if → 2
			S→.V=E	V → 3
			V→.i	i → 4
12	E_后继	基本项集	E→(E.)	) → 14
13	S_后继	基本项集	S→if(E)S.	#S::=if(E)S → 16
14	)_后继	基本项集	E→(E).	#E::=(E) → 16

步骤 3　按 5 个规则填分析表。对于文法 G5.3[S]，Follow(S) = {#}、Follow(V) = { = }与 Follow(E)={), #}。最终所得分析表如表 5.6 所示。

表 5.6

状态	ACTION						GOTO		
	if	(	)	=	i	#	S	V	E
0	S2				S4		1	3	
1						acc			
2		S5							
3				S6					

4				r3					
5		S8			S9				7

续表

状态	ACTION						GOTO		
	if	(	)	=	i	#	S	V	E
6		S8			S9				10
7			S11						
8		S8			S9				12
9			r5			r5			
10						r2			
11	S2				S4		13	3	
12			S14						
13						r1			
14			r4			r4			

由于在状态 1 时，当输入符号是#，已识别出输入符号串是文法的句子，因此同前面一样，删去项集 15 所对应的状态 15。该分析表之所有元素均为唯一，因此是 LR(1)分析表，事实上，从表 5.5 可见，任一 LR(0)项集中，或者全是移入项或者全是归约项，则相应的 CFSM 中将不包含不适定状态，从而该分析表事实上是 LR(0)分析表，相应文法是 LR(0)文法。

请注意，上述是直接从表来构造 SLR 分析表。如果从 CFSM 出发构造分析表，则分析表构造步骤如下。

步骤1　从原有文法构造增广文法，即增加新的符号 Z'作为识别符号，并添加规则 Z′::=Z#。

步骤 2　构造 CFSM。首先构造 LR(0)项集规范族 $C=\{S_0,S_1,S_2,\cdots,S_n\}$，并构造 GO 函数。这里 GO 函数给出了一个项集关于某个文法符号的后继项集。例如对于文法 G5.2 [E]的表 5.4，项集 0 有 E_后继，其项集序号是 1，则有 $GO(S_0,E)=S_1$，反之，$GO(S_2,*)=S_7$，指明项集 2 有*_后继，其项集序号是 7。根据表 5.4 可以构造一切 GO 函数。让项集对应于状态，且项集序号作为状态名，GO 函数描述了状态的转换，因此给定 LR(0)项集规范族和 GO 函数，实质上已经构造了 CFSM。以含有项 Z′→.Z#的项集 S_i 所对应的状态作为初始状态。通常初始状态名 i=0，对应于#归约_后继的状态为终止状态。

步骤 3　填 SLR(1)分析表。分析表中的状态就是步骤 2 中所得到的状态。分析表中的 ACTION 部分和 GOTO 部分按下列规则填入。

① 如果 $GO(S_i,U)=S_j,U\in V_N$,则置 GOTO[i][U] = j，表示当分析栈顶状态是 i，归约成的非终结符号是 U 时，状态转换为 j。

② 如果移入项 U→u.Tv∈S_i，$T\in V_T$，且 $GO(S_i,T) = S_j$，则置 ACTION[i][T] = Sj，表示把状态 j 与文法符号 T 一起移入（下推入）分析栈。

③ 如果归约项 U→u.∈S_i，且 U:: = u 是增广文法 G 序号为 j 的规则，则对任何 $T\in F_T^1(U)$，置 ACTION[i][T] = rj，表示按序号为 j 的规则进行直接归约。

④ 如果 Z′→Z.# ∈S_i，则置 ACTION[i][#] = acc，表示接受动作，这是因为 S_i 的#_后继是接受项

Z′→Z#.。

⑤ 凡是不能由上述 4 个规则确定的分析表（ACTION 部分与 GOTO 部分）元素，全置为报错标志（空白）。

5. 为二义性文法构造 LR 分析表

有一点要强调的是：LR 文法是无二义性的，SLR 文法也一样。那么为某个语言设计一个上下文无关文法时，该文法是否一定是无二义性的呢？当然未必。那么为二义性文法能否构造 SLR(1)分析表？

【例 5.4】　为二义性文法构造 SLR(1)分析表的例子。

设有文法 G5.4[E]：

```
E::=E+E   E::=E*E   E::=(E)   E::=i
```

该文法显然是二义性的，例如，对于其句子 i+i*i 可构造两个不同的语法分析树，如图 5-7 所示。

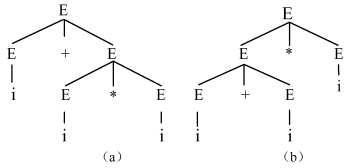


图 5-7

按 SLR(1)分析技术构造分析表如下。

步骤 1　增广文法 G5.4 [E]为 G5.4′[Z]：

Z:: = E#　　E:: = E + E　　E:: = E*E

E:: = (E)　　E:: = i

步骤 2　以表的形式构造 LR(0)项集规范族，如表 5.7 所示。

表 5.7

项集编号	项集名	项		后继关系
0	初始项集	基本项集	Z→.E#	E→1
		闭包集合	E→.E+E	
			E→.E*E	
			E→.(E)	(→2
			E→.i	i→3
1	E_后继	基本项集	Z→E.#	#→10
			E→E.+E	+→4
			E→E.*E	*→5
2	(_后继	基本项集	E→(.E)	E→6
		闭包集合	E→.E+E	
			E→.E*E	
			E→.(E)	(→2
			E→.i	i→3

续表

项集编号	项集名	项		后继关系
3	i_后继	基本项集	E→ i.	#E::=i → 11
4	+_后继	基本项集	E→E+.E	E → 7
		闭包集合	E→.E+E E→.E*E E→.(E) E→.i	(→ 2 i → 3
5	*_后继	基本项集	E→E*.E	E → 8
		闭包集合	E→.E+E E→.E*E E→.(E) E→.i	(→ 2 i → 3
6	E_后继	基本项集	E→(E.) E→E.+E E→E.*E	) → 9 + → 4 * → 5
7	E_后继	基本项集	E→E+E. E→E.+E E→E.*E	? → ?
8	E_后继	基本项集	E→E*E. E→E.+E E→E.*E	? → ?
9	)_后继	基本项集	E→(E).	#E::=(E) → 11

从表 5.7 可见，存在序号为 7 与 8 的两个项集，既包含归约项又包含移入项，因此相应的 CFSM 中将存在不适定状态。对于项集 7 相应的不适定状态将有+_转换、*_转换与#E::=E+E 转换，向前看 1 集合依次是{ + }、{*}与 $F_T^1(E)$。为此计算 $F_T^1(E)$ = Follow(E) = { + ,*,) ,#}，这表明项集 7 相应的不适定状态的+_转换、*_转换与#E:: = E + E 转换的简单向前看集合不是互不相交的。在表 5.7 中项集 7 的后继关系中用问号表示。当分析过程中，分析栈顶为相应于项集 7 的状态时，若输入符号为+或*将难以确定执行移入动作还是归约动作，因此文法 G5.4[E]不是 SLR(1)文法。项集 8 情况类似。

那么，能否为文法 G5.4[E]构造 SLR(1)分析表？分析文法 G5.4[E]，可以发现存在二义性的原因在于信息不足，确切地说，缺少运算符优先级与结合性的语义信息。因此在填分析表确定分析动作时，应用运算符优先级与结合性的语义信息。假定*比+优先级高，即*先运算，+后运算，从归约的角度看，*先于+归约，且*与+都具有左结合性。

从项集 7 是项集 4 的 E_后继，可推断出：当分析栈顶是对应于项集 7 的状态时，分析栈顶的部分符号串是 E + E，相应句型将是 E + EX…。若输入符号是 X = + ，则句型为 E + E + …，显然由左结合，应进行归约，不应该移入。如果输入符号是 X=*，则句型是 E+E*…，显然由运算符优先级，必须移入而不是对 E+E 归约，又如果输入符号是 X=)或#，它们属于 $F_T^1(E)$，应按规则 E:: = E + E 归约。

因此概括起来，当分析过程中，分析栈顶状态是项集 7 相应的状态时，输入符号是*时，执行移入动作，而在其他输入符号(+ ,)与#) 时执行归约动作。而对于项集 8，它包含项 E→E*E.，是归约项，在分析过程中，当分析栈顶是项集 8 相应的状态时，分析栈顶部分是 E*E，句型呈 E*E…形，而另两个项是 E→E. + E 与 E→E.*E，都是移入项，但从运算符的优先级与结合性看，不论是符号 + 还是*，都应执行归约动作，即按规则 E::=E*E 归约。因此分析栈是项集 8 相应的状态时，不论输入符号是什么(+ ，*，)与#)，总是执行归约动作。概括起来，通过步骤 3 填分析表如表 5-8 所示。

表 5.8

状态	ACTION						GOTO
	+	*	(	)	i	#	E
0			S2		S3		1
1	S4	S5				acc	
2			S2		S3		6
3	r4	r4		r4		r4	
4			S2		S3		7
5			S2		S3		8
6	S4	S5		S9			
7	r1	S5		r1		r1	
8	r2	r2		r2		r2	
9	r3	r3		r3		r3	

5.2.2　LR(1)识别程序的计算机实现

如前所述，LR 识别程序由驱动程序和分析表组成。驱动程序是总控程序，这在应用 LR(1)分析技术生成语法分析树一节中已经给出其 C 型语言程序。不论是哪一个文法，只要是 LR(1)文法，且给出了 LR(1)分析表，便可应用该驱动程序进行句型分析，因此实现的要点是分析表、分析栈与语法分析树的实现。

分析表由 ACTION 部分和 GOTO 部分组成。ACTION 部分是由状态与文法符号匹配决定一个分析动作，类似于 LL(1)分析表，ACTION 部分也可用一个二维数组实现：状态是第一维，文法符号是第二维，它们都可以用序号表示，即对应于整型值。分析动作是下列 4 类：

①“Si”表示移入动作，把状态 i 与输入符号一起下推入分析栈。

②“rj”表示归约动作，按序号为 j 的文法规则对分析栈顶构成句柄的符号串部分进行直接归约。

③ “acc”表示接受，识别出输入符号串是相应文法的句子。

④ 空白元素表示在当前分析步时，分析栈顶的状态与当前输入符号匹配是错误的，因此输入符号串不是相应文法的句子。

这些分析表元素都是字符串。如果 ACTION 部分的元素定义为字符串，为了识别是哪种分析动作、哪个状态或哪个文法规则等，较为麻烦、简单的方法是用数值代替，也就是说，

“Si”：用正整值表示，即 + i(i>0)。例如，“S2”对应于正整值 + 2。

“rj”：用负整值表示，即 – j(j>0)。例如，“r3”对应于负整值 – 3。

“acc”：用特殊整值表示，如 999。

空白元素：用数值 0 表示。

这样 LR 识别程序只需判别分析表元素是正整值、负整值、特殊值还是 0，便可确定分析动作是移入、归约、接受还是报错。例如，如果分析表元素是+5，便是 S5，是移入；如果是–6，便是 r6，是归约；如果是 999，便是 acc，接受，识别出是句子，而当分析表元素是 0，便是报错。这样处理，很容易取到移入动作时的状态序号、归约时的文法规则序号等，也很容易用 C 语言实

现。例如，可为分析表 ACTION 部分如下地定义数据结构：

```
int  ACTION [MaxStateNum][MaxVtNum];
```

其中，MaxStateNum 与 MaxVtNum 分别是可允许的最大状态（state）个数与最大终结符号个数。

进一步可以利用 C 语言赋初值设置，如下地给出文法 G5.2 [E]的分析表 ACTION 部分（见表 5.2），其中 MaxStateNum=12 与 MaxVtNum=6。

```
int  ACTION [12][6]=
{   { 0,  0,  0,  4,  0,  5},      {999,   6,   0,   0,    0,  0},
    {-2, -2,  7,  0, -2, 0},       {-4,   -4,  -4,   0,   -4,  0},
    {   0,  0,  0,  4,  0,   5},   {-6,   -6,  -6,   0,   -6, 0},
    {   0,  0,  0,  4,  0,   5},   { 0,  0,   0,    4,    0, 5},

    {   0,  6,  0,  0,  11,  0},   { -1,  -1,   7,   0,   -1, 0},

    { -3, -3,  -3, 0, -3,  0},     { -5,  -5,  -5,  0,   -5,  0}
};
```

分析表的 GOTO 部分类似地用二维数组实现：第一维是状态序号，第二维是非终结符号序号，而数组元素是状态的序号，例如，对于文法 G5.2[E]的分析表 GOTO 部分（见表 5.2），可给出如下：

```
int  GOTO [12][4]=
{   { 0,  1, 2, 3 },  { 0,  0, 0,  0 }, { 0,  0, 0, 0 },
    { 0,  0, 0, 0 },  { 0,  8, 2,  3 }, { 0,  0, 0, 0 },
    { 0,  0, 9, 3 },  { 0,  0, 0, 10 }, { 0,  0, 0, 0 },
    { 0,  0, 0, 0 },  { 0,  0, 0,   0 },{ 0,  0, 0, 0 }
};
```

这里 MaxStateNum = 12，MaxVnNum = 4。注意，这里非终结符号与终结符号的排序与存放文法时确定的排序必须一致，也就是说，对于文法 G5.2[E]，必须 V_N = {E, T, F}，而 V_T = { + , *, (,), i}。左右端标志符号#的序号是 0。由于 C 语言数组元素的下标从 0 开始，而非终结符号的序号从 1 开始，因此赋初值时，添加了第一个数组元素 0，例如，GOTO[0]={0,1,2,3}，因此，MaxVnNum 是 4，而不是 3。

下面考虑分析栈的实现。分析栈的元素由状态与文法符号组成，状态是序号，即整型值，而文法符号也可以用整型值表示，即用文法符号在相应集合中的序号表示，只是对于非终结符号必须序号 + 100（或其他易识别的值），以区别于终结符号。然而在语法分析树中，同一个文法符号可以作为语法分析树上多个结点上的符号，例如，关于文法 G5.2[E]，为输入符号串 i*(i + i)构造的语法分析树，如图 5-8 所示，其中文法符号是 i 的结点有 3 个，而文法符号分别是 T 与 F 的结点各有 4 个。为了标明分析中的文法符号是哪一个结点的文法符号，用结点序号代替文法符号。这样，从结点序号同样可以知道文法符号是什么，不仅如此，还可以知道该结点的父结点、左兄结点与右子结点。

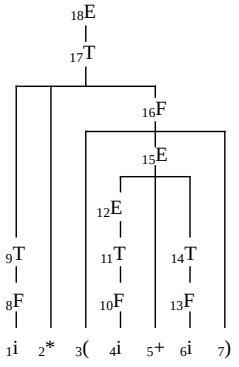


图 5-8

分析栈元素的数据结构用 C 语言可设计如下：

```
typedef struct
{ int 状态序号;     int   结点序号;
} 分析栈元素类型;
```

分析栈用数组实现，用 C 语言可设计如下：

```
分析栈元素类型　分析栈[MaxDepth];
```

这里 MaxDepth 是可允许的最大分析栈深度，另外引进一个分析栈顶指针 top，它实质上是分析栈元素个数计数器，是一个整型变量，它的值指示分析栈中栈顶位置。

关于语法分析树的数据结构很简单，一个结点的结构示意图如下：

结点序号	文法符号	父结点	左兄结点	右子结点

这是一种结构类型，当每个域都用序号即整型值表示时，用 C 语言可设计如下：

```
typedef struct
{  int 结点序号;      int 文法符号序号;
   int 父结点序号;    int 左兄结点序号;  int  右子结点序号;
}语法分析树结点类型;
```

用数组实现语法分析树，用 C 语言可设计如下：

```
语法分析树结点类型　语法分析树[MaxNodeNum];
```

这里 MaxNodeNum 是语法分析树中可允许的最大结点数，类似地引进一个结点个数计数器，以指明实际建立的结点个数。

为了检验语法分析树构造正确与否，可以按下列格式输出语法分析树：

结点序号　文法符号　父结点序号　左兄结点序号　右子结点序号

例如，对于图 5-8 中的语法分析树可输出如表 5.9 所示。

表 5.9

结点序号	文法符号	父结点序号	左兄结点序号	右子结点序号
1	i	8	0	0
2	*	17	9	0
3	(	16	0	0
4	i	10	0	0
5	+	15	12	0
6	i	13	0	0
7	)	16	15	0
8	F	9	0	1
9	T	17	0	8
10	F	11	0	4
11	T	12	0	10
12	E	15	0	11
13	F	14	0	6
14	T	15	5	13
15	E	16	3	14
16	F	17	2	7
17	T	18	0	16
18	E	0	0	17

其中输出的是文法符号本身而不是序号，这是很易实现的。假定有：

```
char  VN [4 ]={ ' ', 'E', 'T', 'F' },
      VT [6 ]={ ' ', '+', '*', '(', ')', 'i'};
```

设正输出序号为 k 的结点上之信息，用下列程序片段即可输出文法符号本身：

```
S=语法分析树[k].文法符号序号;
if(S<100)
  Symbol=VT[S];
else
```

```
  Symbol=VN[S-100];
```

如果是多字符的符号，例如文法 G5.3[S]，可修改如下：

```
     char VN[MaxVnNum][MaxLength]= { " ", "S", "V", "E"},
     VT [MaxVtNum][MaxLength]={ " ","if", "(", ")", "i", "+"};
 S=语法分析树[k].文法符号序号;
 if(S<100)
    strcpy (Symbol, VT[S]);
 else
    strcpy (Symbol, VN[S-100]);
```

对照语法分析树表列形式的输出，画出各个结点及反映相互关系的边，就可以得到语法分析树，通过它来检查语法分析树构造的正确性，从而也检查了语法分析的正确性。

5.2.3 识别程序自动构造

1. 识别程序自动构造的基本思想

如前所述，LR(1)识别程序由驱动程序和分析表组成，而对于任何 LR(1)文法，可共同使用同一个驱动程序，区别仅在于 LR(1)分析表。因此，基于 LR(1)分析技术自动构造识别程序，实质上是 LR(1)分析表的自动构造。可以设计并实现一个 LR 分析表自动生成程序，它以上下文无关文法为输入，以 LR 分析表为输出。

如果一个文法是无二义性文法，进一步说，是 SLR(1)文法，则必定可为它生成 SLR(1)分析表。但如果不是 SLR(1)文法，便不可能为它构造 SLR(1)分析表，因为分析表中会存在一些 ACTION 元素不唯一，这时可利用一些语义信息，例如运算符优先级与结合性，设法来解决。因此不是 SLR(1)文法，也可能为它构造 SLR(1)分析表。这里以 SLR(1)分析表的自动构造为例，给出控制流程示意图如图 5-9 所示。

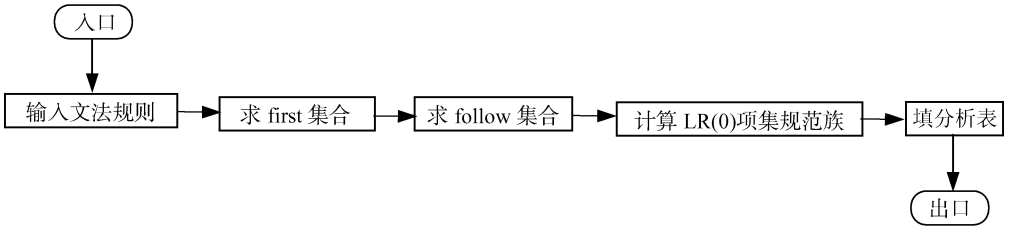


图 5-9

这个示意图看似简单，但用 C 语言实现工作量较大，尽管利用 LR(0)的脚标表示法可方便地引进项在计算机内的表示法，然而对于项集的表示，特别是项集是否相同的判别，实现起来难度较大。

有兴趣自行实现分析表自动构造的读者，可以关心文法、项、项集、状态转换函数（项集后继关系）及向前看集合在计算机内的表示及构造，自行设计相应的数据结构及构造算法，重点考虑相同项集的判别算法。

当前广泛应用着一个识别程序自动生成系统，即 YACC。YACC 是识别程序自动生成工具中的典型代表，自 20 世纪 70 年代问世以来，几经改进，目前已广泛应用于许多编译系统的开发。

YACC 以 LALR 类文法为输入，生成 LALR 分析表。与 SLR 分析表的构造情况类似，YACC 不一定正好应用于 LALR(1)文法，分析表构造过程中也可能发生冲突，YACC 给出了解决冲突的一些办法与设施，使得分析表元素能唯一地确定。

这里不拟详细讨论 YACC，仅讨论 LALR(1)分析表构造及 LALR(1)文法的一些基本情况。

2. LALR(1)分析表构造方法

【例 5.5】　设有文法 G5.5[S]:

1: S::=V=E　　2: S::=E　　3: V::=*E　　4: V::=i　　5: E::=V

易见该文法是无二义性的文法，试为其构造 SLR(1)分析表。

步骤 1　对文法 G5.5[S]进行增广，以新符号 Z 为识别符号，增加规则 Z:: = S#。

步骤 2　以表的形式构造 LR(0)项集规范族如表 5.10 所示。

表 5.10

项集编号	项集名	项		后继关系
0	初始项集	基本项集	Z→.S#	S→1
		闭包集合	S→.V=E S→.E V→.*E V→.i E→.V	V→2 E→3 *→4 i→5
1	S_后继	基本项集	Z→S.#	#→10
2	V_后继	基本项集	S→V.=E E→V.	=→6 #E::=V→11
3	E_后继	基本项集	S→E.	#S::=E→11
4	*_后继	基本项集	V→*.E	E→7
		闭包集合	E→.V V→.*E V→.i	V→8 *→4 i→5
5	i_后继	基本项集	V→i.	#v::=i→11
6	=_后继	基本项集	S→V=.E	E→9
		闭包集合	E→.V V→.*E V→.i	V→8 *→4 i→5
7	E_后继	基本项集	V→*E.	#V::=*E→11
8	V_后继	基本项集	E→V.	#E::=V→11
9	E_后继	基本项集	S→V=E.	#S::=V=E→11

10	#_后继	基本项集	Z→S#.	#Z::=S# ⟶ 11

从表 5.10 可见，显然存在项集 2，既包含移入项又包含归约项，它将对应于 CFSM 中的不适定状态。这时存在 = _转换与#E::=V 转换。显然两者的简单向前看 1 集合分别是{=}与 F_T^1(E) = Follow(E) = { =，#}，它们相交，有共同的元素：符号=，因此，发生移入—归约冲突。文法 G5.5 不是 SLR(1)文法。

可以利用例 5.4 中的方法处理这一情况，使得能构造 SLR(1)分析表，具体处理过程如下。

项集 2 是项集 0 的 V_后继，对于其中的项 E→V.是项 E→.V 的后继项，而项 E→.V 是在项 S→.E 所在的闭包集合（项集 0）中，这表明把 V 归约为 E，是要进一步归约为 S。当分析栈顶是项集 2 对应的状态 2 时，分析栈顶符号是 V，这时如果输入符号是=，便不能归约，只能是移入，仅当符号#时归约。因此，可以构造 SLR(1)分析表如表 5.11 所示。

表 5.11

状态	ACTION				GOTO		
	=	*	i	#	S	V	E
0		S4	S5		1	2	3
1				acc			
2	S6			r5			
3				r2			
4		S4	S5			8	7
5	r4			r4			
6		S4	S5			8	9
7	r3			r3			
8	r5			r5			
9				r1			

由于在状态 1 输入符号为#时已可识别出句子，删去了项集 10 所对应的状态。尽管构造出了 SLR(1)分析表，但文法 G5.5[S]并不是 SLR(1)文法。事实上，文法 G5.5[S]是 LALR(1)文法。下面讨论 LALR 方法构造分析表的基本思想。

LALR 是构造 LR 分析表的一种方法，主要是为解决规范 LR 分析表构造工作量太大的问题而提出的。构造 LALR(1)分析表的基本思想是：基于 LR(1)项集规范族，通过对其中所谓的同心项集进行合并，而使分析表既保持 LR(1)项中向前看符号的信息，又使状态数减少到与 SLR(1) 分析表的一样多。

现在考虑 LR(1)项集规范族的概念及其构造。LR(0)项集规范族是从初始 LR(0)项 Z′→.Z#出发，构造初始项集，然后构造初始项集的后继项集，再从后继项集构造新的后继项集，最终得到全部项集而构成 LR(0)项集规范族。LR(1)项集规范族与 LR(0)项集规范族类似，区别仅在于：把 LR(0)项改为 LR(1)项，LR(1)项形如[U→u.v; a]，其中 U→u.v 是 LR(0)项，$a \in V_T \cup \{\#\}$，是向前看的 1 个符号，通常称为搜索符。一个搜索符仅当是归约项时才起作用，即对于 v=ε的[U→u.; a]所对应的状态，只有在输入符号是 a 时才能按规则 U:: = u 归约，在移入项或待约项中的搜索符都不起作用。

当构造初始项集或后继项集时，同样地涉及闭包项集的计算。只是现在固定向前看 1 个符号，试以例说明。

关于文法 G5.5[S]，为输入符号串*i=*i 可构造语法分析树如图 5-10 所示。初始项为[Z→.S；#]。注意增广的规则为：Z:: = S，右部末端没有符号#。要归约到 S，需先归约到 V=E，应有[S→.V=E; #]，这样，需要先归约到 V，如图所示，应用规则 V:: = *E，而 V 之后（右边）为符号 = ，因此有[V→.*E; =]。也可能按规则 V::=i 归约，因此又有[V→.i; =]。由此可见，

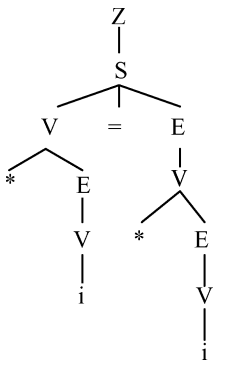


图 5-10

向前看的 1 个符号将是这样得到的一个终结符号或#,即由相应句型中所归约成的非终结符号之右部紧随的符号串所推导出来的第一个终结符号或#。

初始项集 I_0 是初始 LR(1)项的闭包集合，有：

I_0=Closure ({[Z→.S; #]}) =

{ [Z→.S; #], [S→.V=E; #], [S→.E; #], [V→.*E; =], [V→.i; =],

[E→.V; #], [V→.*E; #], [V→.i; #] }

为书写简单起见，把有相同 LR(0)部分的 LR(1)项合并，不同搜索符之间用一竖隔开，这样，I_0 写成了：

I_0={[Z→.S; #],[S→.V=E; #],[S→.E; #],[V→.*E; =|#], [V→.i; =|#], [E→.V; #] }

为直观起见，类似地可用表的形式构造各个 LR(1)项集如表 5.12 所示。

表 5.12

项集编号	项集名	项		后继关系
0	初始项集	基本项集	[Z→.S; #]	S→1
		闭包集合	[S→.V=E; #] [S→.E; #] [V→.*E; =] [V→.i; =] [E→.V; #] [V→.*E; #] [V→.i; #]	V→2 E→3 *→4 i→5
1	S_后继	基本项集	[Z→S.; #]	#Z::=S→14
2	V_后继	基本集合	[S→V.=E; #] [E→V.; #]	=→6 #E::=V→15
3	E_后继	基本项集	[S→E.; #]	#S::=E→15
4	*_后继	基本项集	[V→*.E; =|#]	E→7
		闭包集合	[E→.V; =|#] [V→.*E; =|#] [V→.i; =|#]	V→8 *→4 i→5
5	i_后继	基本项集	[V→i.; =|#]	#V::=i→15
6	=_后继	基本项集	[S→V=.E; #]	E→9
		闭包集合	[E→.V; #] [V→.*E; #] [V→.i; #]	V→10 *→11 i→12
7	E_后继	基本项集	[V→*E.; =|#]	#V::=*E→15

8	V_后继	基本项集	[E→V.; =|#]	#E::=V → 15
9	E_后继	基本项集	[S→V=E.; #]	#S::=V=E → 15

续表

项集编号	项集名	项		后继关系
10	V_后继	基本项集	[E→V.; #].	#E::=V → 15
11	*_后继	基本项集	[V→*.E; #]	E → 13
		闭包集合	[E→.V; #]	V → 10
			[V→.*E; #]	* → 11
			[V→.i; #]	i → 12
12	i_后继	基本项集	[V→i.; #]	#V::=i → 15
13	E_后继	基本项集	[V→*E.; #]	#V::=*E → 15

由该表填分析表，比填 SLR(1)分析表更方便得多，因为 LR(1)项中已指明了向前看符号，例如，对于项集 2，显然填移入项 ACTION[2][=] = S6，填归约项 ACTION[2][#] = r5，无需再计算 F_T^1 集合。只是这里有 14 个项集，将对应于 14 个状态，而用 SLR 方法，仅对应 10 个状态。

LALR(1)分析表构造方法的基本思想是合并同心项集，即如果不计扫描符，两个 LR(1)项集完全相同，则这两个 LR(1)项集就是同心项集，就应该合并。例如，项集 4 与项集 11 是同心项集，因为它们不计扫描符的 LR(0)部分完全相同。假定合并后的项集序号为 4′。类似地，项集 5 与项集 12 是同心项集，可合并为项集 5′，项集 7 与项集 13 是同心项集，可合并为项集 7′，项集 8 与项集 10 是同心项集，可合并为项集 8′。这样，项集数减少到 10，这正是同一文法的 SLR(1)分析表的状态数。利用合并了同心项集的 LR(1)项集规范族，可以类似地构造分析表，即 LALR(1)分析表。

从构造 LALR(1)分析表 的方法可见，LALR 方法既有与 SLR(1)方法一样的优点，即生成的分析表状态数少，又有与规范 LR(1)构造分析表方法一样的优点，即利用了向前看信息，因而应用范围更广，因此它介于规范 LR(1)与 SLR(1)之间。当前语法分析程序自动生成系统 YACC 便是基于 LALR 分析表构造方法的。

5.3 其他的自底向上分析技术

自底向上分析技术除了 LR 分析技术，还有优先分析技术。优先分析技术根据相邻两个符号进行归约的先后次序，引进优先关系来实现句型分析。优先分析技术又分简单优先分析技术与算符优先分析技术两类，这里概要介绍算符优先分析技术。

5.3.1 算符优先分析技术概况

通常进行算术运算时，熟知的法则是：先乘除，后加减。如果有式子 3 + 5*7，则计算顺序是 3 + （5*7）。如果是 7 + 5*3，计算顺序还是：7 + （5*3），并不因为加号前的是 7，比 5 与 3 大，就先计算加法。这表明计算次序与运算分量无关。从文法角度看，先计算，也就是先归约，因此

联想到表达式文法 G5.6[E]：

E::=E+T|E−T|T　T::=T*F|T/F|F　F::=(E)|i

在进行句型分析时，能否也仅仅由“运算符”来决定归约的次序？

现在的问题是：什么是运算符，什么是运算分量？算符优先分析技术把非终结符号看作是运算分量，终结符号看作是运算符。通常的算术表达式中，不可能出现相邻的两个运算分量，从文法的角度，一个句型中不可能出现相邻的两个非终结符号，这时文法中的任何规则的右部，必然不出现相邻的两个非终结符号，这种文法称为算符文法，即算符文法中的重写规则必定都不呈以下形式：

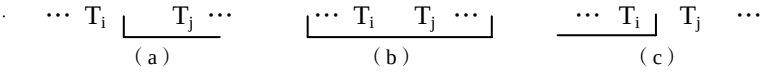
$U::=\cdots U_iU_j\cdots$　　$U_i, U_j \in V_N$

这表明，必须在算符文法的基础上讨论算符优先分析技术。由于表达式中，不可能有相邻的两个运算分量，相对照，算符文法的句型必定呈以下形式：

$[N_1]T_1[N_2]T_2[N_3]\cdots[N_{m-1}]T_{m-1}[N_m]T_m[N_{m+1}]$

其中，$N_k \in V_N$, $T_k \in V_T$（k = 1,2,⋯,m），括号对［与］表示括住的 N 可能出现，也可能不出现。

在进行句型分析时，相邻的两个终结符号 T_i 与 T_j（可能之间有一个非终结符号 N），进行归约时必定仅如下所示 3 种情况：



（a）中 T_j 先归约，是被归约短语中的头终结符号，T_i 不在被归约短语中；（b）中 T_i 与 T_j 同时在被归约短语中；（c）中 T_i 先归约，是被归约短语的尾终结符号，而 T_j 不在被归约短语中。对这 3 种情况分别引进 3 个所谓的算符优先关系：$\lessdot$、$\doteq$ 与 $\gtrdot$，这里的圆代表 Operator（算符）的首字母，分别有：$T_i \lessdot T_j$、$T_i \doteq T_j$ 与 $T_i \gtrdot T_j$。

关于文法 G5.6 [E]画出一些语法分析树如图 5-11 所示。

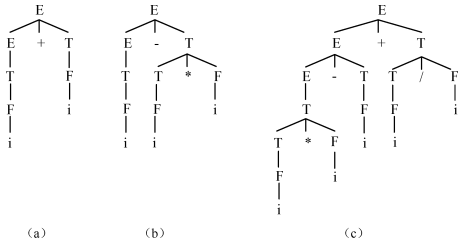


图 5-11

从这些语法分析树可找出算符优先关系，例如，从（a）可有 $i \gtrdot +$与$+ \lessdot i$，从（b）有 $i \gtrdot -$、$- \lessdot i$ 与 $i \gtrdot *$等。从（c）又有$- \gtrdot +$与$+ \lessdot /$等。当构造足够多的、有代表性的语法分析树时，可找出所有算符优先关系。但构造语法分析树找优先关系，可能重复，更可能遗漏。

为了能系统地求出一切算符优先关系，可以按文法规则直接计算。例如，从形如：

$U::=\cdots T_iT_j\cdots$

或　　$U::=\cdots T_iWT_j\cdots$

的规则直接找出 $\doteq$ 关系：$T_i \doteq T_j$。对于 $\lessdot$，寻找形如：

$U::=\cdots T_iV\cdots$

的规则，其中 $V=>+ T_j\cdots$，或 $V=>+ WT_j\cdots$，$V,W \in V_N$，$T_i \lessdot T_j$。对于 $\gtrdot$，寻找形如

$U::= \cdots VT_j\cdots$

的规则，其中 $V=>+ \cdots T_i$，或 $V=>+ \cdots T_iW$，$V,W \in V_N$，$T_i \gtrdot T_j$。

通常把所有的算符优先关系放在所谓的算符优先矩阵中，如表 5.13 所示。从表中可见各对终结符号之间的算符优先关系，如（ ⋖ +与* ⋗ ）等。

表 5.13

	+	−	*	/	(	)	i
+	⋗	⋗	⋖	⋖	⋖	⋗	⋖
−	⋗	⋗	⋖	⋖	⋖	⋗	⋖
*	⋗	⋗	⋗	⋗	⋖	⋗	⋖
/	⋗	⋗	⋗	⋗	⋖	⋗	⋖
(	⋖	⋖	⋖	⋖	⋖	=	⋖
)	⋗	⋗	⋗	⋗		⋗	
i	⋗	⋗	⋗	⋗		⋗	

一个算符文法，如果它的算符优先矩阵中一切元素都是唯一的，则称此算符文法是**算符优先文法**。

概括起来，能应用算符优先分析技术的文法，必须是算符优先矩阵的元素都唯一的算符文法。算符优先文法中任何重写规则的右部不包含相邻的两个非终结符号，它的一切句型中，也一定不包含相邻的两个非终结符号。

当应用算符优先分析技术，对输入符号串句型分析时，分析过程如下：首先从左到右找出被归约短语的尾符号，其右部的符号不在被归约短语中，然后反过来，从右到左找出被归约短语的头符号，其左部的符号不在被归约短语中。这样，从头符号到尾符号止的符号串组成被归约的短语。对于输入符号串的左端符号与右端符号，将与左右端标志符号 # 比较。#的优先级总是最低。例如，对 i+i*i 的分析过程如下：

步骤	句型	关系	被归约短语	归约成符号
1	i+i*i	# ⋖ i ⋗ + i*i	i	F
2	F+i*i	# ⋖ + ⋖ i ⋗ *i	i	F
3	F+F*i	# ⋖ + ⋖ * ⋖ i ⋗ #	i	F
4	F+F*F	# ⋖ + ⋖ * ⋗ #	F*F	T
5	F+T	# ⋖ + ⋗ #	F+T	E
6	E			

最后，句型中只有一个非终结符号，识别出输入符号串 i+i*i 是文法 G5.6[E]的句子，结束分析。按照算符优先分析技术，每个分析步归约的并不是最左简单短语（即句柄），而是 F*F 与 F + T 这样的短语，而且归约成的非终结符号，也是可以任意的，例如，步骤 4 时可以让它归约成 E，而步骤 5 时让它归约成 F 。这是因为非终结符号对分析过程不起任何作用。

把被归约短语称为质短语。**质短语**是句型中这样的短语：其中至少包含一个终结符号，且不包含任何其他的质短语。因此，可以把按算符优先分析技术句型分析的过程概括为：在分析过程中，首先从左到右找出质短语的尾终结符号，然后从右到左找出质短语的头终结符号。

5.3.2 应用算符优先分析技术句型分析

实现算符优先分析技术句型分析的识别程序称为算符优先识别程序。在句型分析时，类似于 LR 识别程序，应用移入—归约法，也引进一个分析栈，当在分析栈中尚未形成质短语时，把输入符号下推入分析栈，当在分析栈顶的部分符号形成质短语时，进行归约，直到达到输入符号串的右端标志符号 # ，栈中仅左端标志符号 # 和一个非终结符号，这时分析成功而结束。例如，给出对输入符号串 i + i*i 的识别过程，如表 5.14 所示。

表 5.14

步骤	栈	关系	下一符号	其余输入符号	动作	最左质短语
1	#	⋖	i	+i*i#	移入	
2	#i	⋗	+	+i*i#	归约	i
3	#F	⋖	+	i*i#	移入	
4	#F+	⋖	i	*i#	移入	
5	#F + i	⋗	*	i#	归约	i
6	#F + F	⋖	*	i#	移入	
7	#F + F*	⋖	i	#	移入	
8	#F + F*i	⋗	#		归约	i
9	#F + F*F	⋗	#		归约	F*F
10	#F + T	⋗	#		归约	F+T
11	#E		#		成功	

很明显，上述分析过程典型地应用了移入—归约法。要注意的是，分析栈顶的符号可能是非终结符号，它在分析过程中不起任何作用，应把它放过。对照上述识别过程，不难用 C 语言来实现算符优先识别程序。关于识别程序句型分析过程的控制流程图请读者自行思考。

5.3.3　优先函数

实际应用算符优先分析技术时，往往用数值取代优先关系，其基本思想是：引进两个优先函数 f 和 g，对于 T_i R T_j，其中 R 是三种算符优先关系之一，把 T_i 映射到 $N_i = f(T_i)$，而把 T_j 映射到 $N_j = g(T_j)$，通过比较 N_i 与 N_j 来取代 T_i R T_j。当引进优先函数后，便不再需要算符优先矩阵。确切地说，关于算符优先文法 G 的算符优先矩阵 P，引进两个优先函数 f 和 g，如果它们满足下列条件：

① 对于 $T_i \lessdot T_j$，有 $f(T_i) < g(T_j)$

② 对于 $T_i \doteq T_j$，有 $f(T_i) = g(T_j)$

③ 对于 $T_i \gtrdot T_j$，有 $f(T_i) > g(T_j)$

则函数 f 和 g 称为算符优先文法 G 的算符优先矩阵 P 的**双线性优先函数**，简称**优先函数**。

由于文法符号是离散的，函数 f 与 g 必定是离散的，只能用表列表示，例如，对于文法 G5.6[E] 的算符优先矩阵如表 5.13 所示，可构造优先函数 f 与 g 如表 5.15 所示。

表 5.15

	+	−	*	/	(	)	i
f	4	4	9	9	2	9	9
g	3	3	7	7	10	2	11

从表 5.15 可见，f(+) = 4，g(−) = 3，有 f(+)>g(−)，对应于 + ⋗ − ；f(−) = 4，g(*) = 7，f(−)<g(*)，对应于 − ⋖ *；又 f(( ) = g( )) = 2，对应于(≐)。用表 5.13 中的算符优先矩阵验证 f 与 g 的值，显然每一对终结符号之间的算符优先关系全都满足，f、g 是要求的双线性优先函数。

从算符优先矩阵构造双线性优先函数的方法有 3 种，即逐次加一法、Bell 有向图法与 Martin 算法。这里不作详细介绍，有兴趣的读者可以自行参看相关资料。

在第 7 章讨论逆波兰表示这种中间表示代码时，从通常的中缀表示的表达式直接生成目标代码时所采用的方法，实际上是算符优先分析技术。早期应用较普遍的运算符优先数法实质上也就是算符优先分析技术。

本章小结

本章讨论自底向上分析技术，内容包括概况（讨论前提、输入输出、要解决的基本问题与基本实现方法）以及 LR(1)分析技术，另外简单介绍了算符优先分析技术。

自底向上句型分析，从推导的角度，是从输入符号串出发，不断进行归约的过程；从语法分析树角度，是以输入符号串作为末端结点符号串，自下向上构造语法分析树的过程。

关于 LR 分析技术，要点有两个：一是应用 LR(1)识别程序句型分析；二是分析表的构造。

应用 LR 分析技术句型分析时，采用了移入—归约法，这是边输入边归约的规范归约过程，由于它不能解决自底向上分析技术必须解决的基本问题，所以仅是自底向上分析技术共同采用的一种基本实现方法。LR 识别程序类似于 LL 分析技术，是在分析表与分析栈联合进行控制下实现句型分析，读者应熟练掌握应用 LR 识别程序进行句型分析，并理解分析栈中状态的作用及为什么能有这样的作用。LR 分析技术通常有 3 种构造 LR 分析表的方法，即规范 LR 方法、简单 LR 方法（SLR）与前 视 LR 方法（LALR）。SLR 分析表最容易构造，工作量较小，但应用面不如 LALR 广，LALR 构造方法目前较多地应用于识别程序的自动生成。SLR 分析表宜于采用表的形式构造。

一个 LR 识别程序由驱动程序和分析表组成，LR(1)驱动程序对于所有 LR(1)文法都适用。LR 识别程序自动构造，实质上是 LR 分析表的自动生成。LR 文法是无二义性的文法，但对于二义性文法，识别程序自动生成程序应有能力处理，换句话说，即使对于二义性文法，它们不是 LR 文法，也能为它们生成 LR 分析表。相关概念包括初始项集、后继项集、LR(0)项集规范族、状态、CFSM、不适定状态、构型、活前缀、LR(1)项集规范族、向前看集合、搜索符。

在应用 LR(1)分析技术句型分析时也有生成语法分析树的问题。分析过程中，每当进行归约时，为所归约成的非终结符号建立一个结点，同时建立结点之间的父子兄弟关系。读者可以比较与自顶向下预测（LL(1)）识别程序生成语法分析树的区别。

另一类自底向上分析技术是算符优先分析技术，它也应用移入—归约法进行归约，但与 LR 分析技术的区别在于：一是引进仅建立于终结符号集之上的算符优先关系来控制句型分析过程；二是归约的是质短语，而不是句柄（最左简单短语）。在实际应用中，往往以优先函数之值的比较控制分析过程。早期应用较普遍的运算符优先数法实质上也就是算符优先分析技术。相关概念有算符优先文法，算符优先矩阵。

复习思考题

1. 自底向上分析技术的讨论前提是什么？
2. 自底向上分析技术应解决的基本问题是什么？怎样理解这基本问题？
3. 移入—归约法是一种分析技术吗？请说明理由。
4. 把 LR(0)项区分为移入项、归约项、待约项与接受项，有何意义？
5. 如何理解活前缀？
6. 请简述基于 LR 分析技术的识别程序自动生成的基本思想。
7. 请比较算符优先分析技术与 LR 分析技术的异同。

习　　题

1．设有某文法的一个重写规则：S:: = if E S
试为它写出一切 LR(0)项，指明各 LR(0)项的种类，如移入项等。

2．试关于文法 G5.2[E]：

E::=E+T　　E::=T　　T::=T*F　　T::=F　　F::=(E)　　F::=i

的输入符号串 i+i+i、(i+i)*i 与 i*i+i*(i+i)，以下表的形式写出句型识别过程，分析表如表 5.2 所示。

步骤	栈	输入	动作	说明

3．试关于文法 G5.4[E]：

E::=E + E　　E::=E*E　　E::=(E)　　E:: = i

的输入符号串 i + i + i 与(i + i)*i，以下表的形式写出句型识别过程，分析表如表 5.8 所示。

步骤	栈	输入	动作	说明

4．设有文法 G5.5[S]:

S::=V=E　　S::=E　　V::=*E　　V::=i　　E::=V

的句子*i = *i，由 LR 识别程序构造的语法分析树如图 5-12 所示，请为各结点标明建立的顺序。

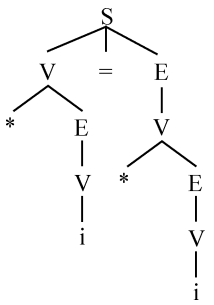


图 5-12

5．试以 C 语言置初值方式给出文法 G5.5[S]：

S::=V=E　　S::=E　　V::=*E　　V::=i　　E::=V

的 LR(1)分析表之 ACTION 部分与 GOTO 部分，分析表如表 5.11 所示。

6. 设为题 3 中文法 G5.4 的句子 i + i*(i)构造的语法分析树如下表所示，请画出相应语法分析树。

结点序号	文法符号	父结点序号	左兄结点序号	右子结点序号
1	i	8	0	0
2	+	10	8	0
3	i	9	0	0
4	*	13	10	0
5	(	12	0	0
6	i	11	0	0
7	)	12	11	0
8	E	10	0	1
9	E	10	2	3
10	E	13	0	9
11	E	12	5	6
12	E	13	4	7
13	E	0	0	12

7．试为文法 G5.6[E]：

1：　E:: = E + T　　2：　E::=E − T　　3：　E:: = T
4：　T:: = T*F　　5：　T:: = T/F　　6：　T:: = F
7：　F:: = (E)　　8：　F:: = i

构造 SLR(1)分析表。

第 6 章 语义分析与目标代码生成

本 章 导 读

在语法分析之后便进入语义分析和目标代码生成阶段。本章讨论的内容包括：概况、说明部分的翻译、类型检查与目标代码的生成等。

概况中讨论了语义分析程序的功能、采用的翻译技术与基本的类型描述手段。语义分析的功能有 4 个，应理解：为什么是这些。本章采用语法制导翻译技术进行翻译，要理解为什么说是语法制导翻译？这翻译技术基于属性文法，属性文法有语法制导定义与翻译方案两类，读者应能读懂这两类属性文法，并掌握相关的概念，如注释分析树与依赖图等。类型表达式是一种与具体语言无关的类型表示方式，读者应熟练掌握为 C 语言的数据对象写出相关联的类型表达式。

对说明部分的翻译，并不生成目标代码，而是把与标识符相关联的类型等属性信息填入相关的表内。而对控制部分的翻译，将生成目标代码，这时需考虑若干相关方面，最关键的是语言结构与目标代码的对应关系。只需根据语言结构的相应语义，明确执行步骤，很容易设计目标代码结构，进一步设计生成目标代码的翻译方案，希望读者理解并熟悉这种解决问题的思路。读者应能结合控制结构的翻译方案，熟练地为 C 语言控制结构写出相应的虚拟机目标代码。希望读者通过本章的讨论，注意到 C 语言的特殊点，对 C 语言有更深入的理解。

翻译方案或语法制导定义，为语义分析与目标代码生成的实现制定了规范，但本身并不能被直接执行，需要设计相应的语义子程序来实现。本章最后，讨论了翻译方案的实现，实现要点有 3 个，即属性值的存取、属性值的计算以及目标代码的生成与存储。建议读者仔细对照，自行进行翻译方案的实现。

6.1 概　　况

6.1.1 语义分析的概念

一个高级程序设计语言程序，一般总是在特定的应用领域下，为实现某种功能而写出的，不做任何事的程序是无意义的。换句话说，写一个程序的目的是达到某种效果，这种效果可看成是程序的含义。因此，编写一个程序不仅要关心程序在书写上的正确性，还必须关心程序在含义上的正确性。在含义正确的基础上，生成正确的目标程序。

程序含义正确性的分析，由编译程序的语义分析部分来完成。语义分析是编译程序继词法分析与语法分析之后的分析工作。通过语义分析，分析源程序中各个语法成分的含义，在理解含义

的基础上，为生成相应的目标程序做好准备，或者直接生成目标程序。编译程序中完成语义分析的部分称为**语义分析程序**。

一个程序的含义包括哪些方面？众所周知，程序是计算任务的处理对象与处理规则的描述。处理对象指的是数据对象，而处理规则指的是控制部分，所以一个程序的含义自然涉及数据结构和控制结构两部分的含义。

【例 6.1】 试按照霍纳方案计算多项式 $a_9x^9+a_8x^8+\cdots+a_2x^2+a_1x+a_0$ 的值，写出相应的程序。

霍纳方案的计算公式如下所示：

$$a_nx^n+a_{n-1}x^{n-1}+\cdots+a_2x^2+a_1x+a_0=(((\cdots(a_nx+a_{n-1})x+\cdots)x+a_2)x+a_1)x+a_0$$

可写出下列 C 语言程序：

```
main( )
{  float  x, S, A[10];
   int  k;
   S=A[9];
   for( k=8;  k>=0;  k=k-1)
      S=S*x+A[k];
}
```

为简单起见，其中省去了输入输出语句，即输入变量 x 与数组 A 的值及输出计算结果 S。该程序的数据对象涉及变量 x 与存放多项式系数 a_0、a_1、…与 a_9 之值的数组 A，所计算多项式的值保存在变量 S 中，整型变量 k 显然是计数器。按照霍纳方案，可用上述循环实现多项式值的计算。

数据对象有不同的类型，例如 S 是 float 型，而 k 是 int 型，这可以通过说明部分规定。说明部分让标识符和相应数据对象的类型等属性相关联，使标识符代表具有相应数据类型属性的数据对象。数据类型不同，它们在计算机内的存储表示也不同。编译程序通过说明部分确定每个标识符代表什么数据类型的数据对象，为它们安排相应的存储区域与相应的存储表示，从而能关于控制部分生成相应的目标指令。

当关于控制部分生成目标指令时，一个必不可少的工作是类型检查，即检查运算的合法性和运算分量类型的一致性。例如，取数组元素时，下标值必须是 int 类型的，如果 A[k]中的 k 不是 int 型显然就错了。一般来说，对于一个双目运算 x　op　y，op 必须是允许对 x 与 y 进行的运算，且 x 与 y 的类型是相同的，在算术运算情况下，至少是相容的。例如，float 类型的变量不允许进行整除求余运算（%），但 float 型变量允许与 char 型变量进行算术运算等。

概括地说，关于数据结构，语义分析要进行确定类型与类型检查两方面的工作，即确定说明部分中的标识符所代表数据对象的类型，以及检查控制部分中运算的合法性与运算分量类型的一致性（相容性）。

不言而喻，在确定标识符类型时，必须考虑到标识符的作用域问题，因为同一个标识符在程序中不同的位置上可以代表完全不同的数据对象。

控制结构的语义是由语言规定的，与说明部分一样，难以用语法规则那种形式来描述。因此语义通常用口语描述，例如，赋值语句 V = e;的语义用口语描述如下：把赋值语句右部表达式 e 的值赋给左部变量 V，如果 e 与 V 的值类型不一致，则先把 e 的值转换到 V 的类型。赋值语句 S=A[9];的语义是：把 A[9]的值赋给变量 S。上述 for 循环语句按 C 语言语义规定，将是 k 从 8 开始执行循环体 S=S*x+A[k]，以后每次 k 值减 1 重复执行循环体，直到 k 的值＜0 为止。当进行语义分析时，编译程序识别所处理语法成分的含义，并进行相应的语义处理。

概括起来，语义分析部分的基本功能有如下 4 个方面。

1. 确定类型

语义分析程序处理源程序中的说明部分时，确定标识符所关联数据对象的数据类型，同时把类型等属性信息填入（或添加入）标识符表相应条目中。对说明部分的处理并不产生目标代码，也没有中间表示代码。请注意，由于经过语义分析，标识符的属性信息被记入标识符表，这时的标识符表扩充成了符号表，因此后面将不再使用标识符表，而改用符号表这一术语。

2. 类型检查

语义分析程序处理源程序中的控制部分时，按照语言的类型规则，对正处理的运算符及运算分量进行类型检查，检查运算的合法性以及运算分量类型的一致性（相容性），必要时进行相应的类型转换。例如，float 型变量与 int 型变量进行加法运算，则需把 int 型变量转换到 float 型后，再与原有的 float 型变量进行加法运算。

3. 识别含义并作相应的语义处理

语义分析程序根据相关语言的语义定义，确认（识别）源程序控制部分中各语法成分的含义，作相应的处理，例如可以对可执行语句直接生成目标代码，也可以生成中间表示代码，以便作进一步的目标代码优化。

4. 其他一些静态语义检查

语义分析程序可以进行一些控制流检查。通常，一些语言（包括 C 语言）不允许从循环体外把控制转移入循环体内，也不允许通过控制转移语句把控制转入条件语句的内嵌语句内。语义分析程序进行的这些语义检查是在编译时刻进行的，因此称为静态语义检查。这样可以避免因这些错误而引起的运行结果不正确。

在本章内仅考虑语义分析程序生成目标程序的情况，因此，语义分析程序以语法分析部分的输出作为输入，即以语法分析树或其他等价的内部中间表示作为输入，其输出是相应的目标程序。

如同语法分析部分不是以词法分析的输出属性字序列作为输入，而是以符号序列作为输入一样，语义分析程序也以符号序列作为输入，并不考虑符号实际的内部表示。特别指出的是：语义分析部分往往与语法分析相结合，在进行语法分析的同时进行语义分析，因此语义分析部分与语法分析部分将有共同的输入。

当讨论词法分析和语法分析时，往往强调其基础文法，即词法分析基于正则文法进行讨论，而语法分析基于上下文无关文法进行讨论。语义分析基于什么文法进行讨论?

词法分析与语法分析基于相应的文法，按照不同的分析技术，构造相应的识别算法，相关的理论和技术已成熟而典型。然而，语义分析大不一样。语义难以用形式化的方式来描述，尽管国内外的专家学者在这方面开展了大量工作，甚至形成了相应的学科：形式语义学，但在实用化方面，并没有为大众所公认和广泛接受的、可行的描述方式和分析技术，各种技术都显得杂乱无章和支离破碎。近年来，在语义分析与目标程序生成上广泛采用了语法制导翻译技术，在语义分析与目标程序生成的形式描述上有了一定的进展，可以说为语义分析与目标程序生成制定了规范。

6.1.2 属性与属性文法

语法制导翻译技术的基本思想是：把语义与语法分开处理，但又在语法分析的同时进行相应的语义分析工作。也就是说，在一个上下文无关文法中给出语法规则（重写规则）的同时，给出语义规则或语义动作，这些语义规则或语义动作给出了语义计算规则的描述。这样，把原有的上

下文无关文法扩充成了所谓的属性文法，语义分析将在属性文法的基础上进行。

要了解什么是属性文法，必须首先了解什么是属性。

一个程序设计语言用某个上下文无关文法来描述，如前所述，引进了非终结符号与终结符号。非终结符号代表语法实体，终结符号是源程序中可以出现的符号。例如，文法 G6.1[E]：

```
E::=E+T    E::=T    T::=T*F    T::=F    F::=(E)    F::=i
```

语法实体 E、T 与 F 事实上相应于表达式、项与因子。终结符号 + 与*对应于算术运算符，(与)是用来改变运算顺序的括号，而 i 则表示标识符，代表变量。显然，E、T、F 与 i 都有相应的类型和值。类型与值是这些文法符号的属性，因此对文法符号可以引进值与类型这样一些属性，这些属性的值便是文法符号的语义值。可以为这些属性取符号名，例如 type（类型）与 val（值）等，沿用结构成员的表示法，可以写出 E.type、T.type 与 F.val 等。当生成目标指令时，需为表达式与变量分配存放值的存储空间，以便引用，可以引进属性 place，表示为 E.place 与 F.place 等。一个标识符 i 的信息总是记录在符号表中，在其相应条目中记录有种类、类型以及分配的存储地址等，为取得其条目，引进属性 entry，例如，i.entry 代表标识符 i 在符号表中条目的指针或序号。总之，只要是语义分析和目标代码生成（或其他翻译工作）的需要，可以为文法符号引进各种各样的属性，这些属性可以是类型、值与存储地址，甚至可以是符号表条目指针与语句标号等。

确定了文法符号的属性，如何计算这些属性值？这由语义规则或语义动作给出属性值计算规则的描述。例如，对于重写规则 E::=E+T，可有如下的语义规则：

$$E.val:=E_1.val+T.val$$

这表示重写规则左部 E 的值 val 是由右部的 E 与 T 的值 val 求和得到的。

同一个文法符号在一个规则中出现多次时，为了区别开是哪一次出现，采用了脚标表示方式，如 E_1。为了避免赋值号与等于号混淆，赋值号用:=表示。

与文法符号属性相关的信息，即属性值，是在语法分析过程中，从所生成语法分析树相应结点的环境中推导出来的，进一步说，是对文法规则附以语义规则，通过对语义规则的计算求得属性值。这表明必须对文法中所有相关重写规则写出相应的语义规则。

当对于某个压缩了的上下文无关文法，让每个文法符号联系于一组属性，且让该文法中的重写规则附以语义规则时，便称该上下文无关文法为**属性文法**。由于使用属性文法时把语法规则与语义规则分开，但在使用语法规则进行语法分析生成语法分析树的同时，又使用这些语义规则来制导翻译，最终生成目标程序，所以称为**语法制导翻译**。语法制导翻译就是在语法规则的制导下，通过对语义规则的计算，实现对输入符号串的翻译。

一个属性文法往往涉及一个压缩了的上下文无关文法、一组有穷个数的属性与一组有穷个数的属性计算规则。还可以有一组有穷个数的条件，指明关于某个重写规则的属性计算规则所应满足的条件。为简单起见，本书不考虑条件。

属性文法有两种书写形式，分别称为语法制导定义与翻译方案。

语法制导定义的例子如下。

【例 6.2】　计算并打印算术表达式之值的语法制导定义。

假定给定了计算算术表达式之值的文法 G6.2[L]:

```
L::=E      E::=E+T    E::=E-T    E::=T
T::=T*F    T::=T/F    T::=F      F::=(E)    F::=N
```

其中 N 表示整数。可为它设计计算并打印算术表达式之值的语法制导定义如表 6.1 所示。

表 6.1

重 写 规 则	语 义 规 则
L::=E	print(E.val)
E::=E_1+T	E.val:=E_1.val+T.val
E::=E_1-T	E.val:=E_1.val-T.val
E::=T	E.val:=T.val
T::=T_1*F	T.val:=T_1.val*F.val
T::=T_1/F	T.val:=T_1.val/F.val
T::=F	T.val:=F.val
F::=(E)	F.val:=E.val
F::=N	F.val:=N.lexval

表 6.1 中的赋值号: = 沿用了 PASCAL 语言中的表示法，以强调等号 = 只表示等于号，而不是赋值号。除了语法制导定义，翻译方案也采用这种表示法。表 6.1 中为文法符号 N 引进了属性 lexval，属性 N.lexval 的值是由词法分析程序提供的，当语义分析时，N 已有输入值作为其属性 lexval 的值。例如，如果输入的是整数值 5，则 N.lexval 值是 5。引进的其他属性，如 E.val 与 T.val 等，则按语义规则计算而得。最终结果由第一个重写规则 L::=E 相应的语义规则 print(E.val)打印输出。

为了展示分析过程中属性值之计算，假定输入符号串为 5-3*1，画出相应的语法分析树如图 6-1（a）所示，并标记出各结点上文法符号的属性之值，如图 6-1（b）所示。

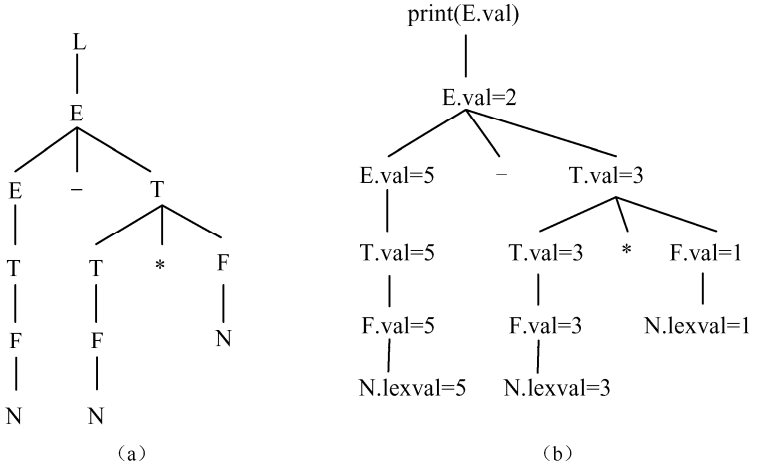


图 6-1

对照语义规则，从图 6-1 清晰可见，各属性值的计算规则是，由子结点的属性值来计算父结点的属性值。例如，由 N.lexval=5 得到 F.val=5，由 F.val=5 又得到 T.val=5，等等。最终，E.val=2 也正是根据 E.val=E_1.val-T.val，从子结点 E.val=5 与 T.val=3 而得父结点 E.val=5−3=2。

这种在各个结点上标记出属性值的语法分析树称为**注释语法树**，而计算语法分析树各个结点上的属性值的过程称为对语法分析树注释。

作为另一个例子，试设计处理程序设计语言说明语句的语法制导定义。

【例 6.3】 处理程序设计语言说明语句的语法制导定义的设计。

这里的处理指的是：把说明语句中标识符的类型属性添加入符号表相应条目中。说明语句的

```
G6.3[D]:  D::=T  L     T::=int    T::=float    L::=L,id     L::=id
```

相应的语法制导定义可设计如表 6.2 所示。

表 6.2

重 写 规 则	语 义 规 则
D::=T L	L.in:=T.type
T::=int	T.type:=integer
T::=float	T.type:=real
$L::=L_1$,id	L_1.in:=L.in；addtype(id.entry, L.in)
L::=id	addtype(id.entry, L.in)

重写规则 D::=T L 的相应语义规则是：L.in:=T.type，这表示属性 L.in 的值从 T.type 而来，而 T.type:=integer 表示：当输入的是 int 时，T.type 的值是 integer（整）型，而 T.type:=real 表示：当输入的是 float 时，T.type 的值是 real（实）型。函数调用 addtype(id.entry,L.in)的功能是把属性 L.in 的值（类型属性值 integer 或 real）添加入由 id.entry 指明的符号表条目中，也就是，向 id.entry 指明的 id 的符号表条目中添加入类型信息。当输入符号串是 float　i,j,k 时，可有如图 6-2（a）所示的语法分析树及如图 6-2（b）所示的注释分析树。

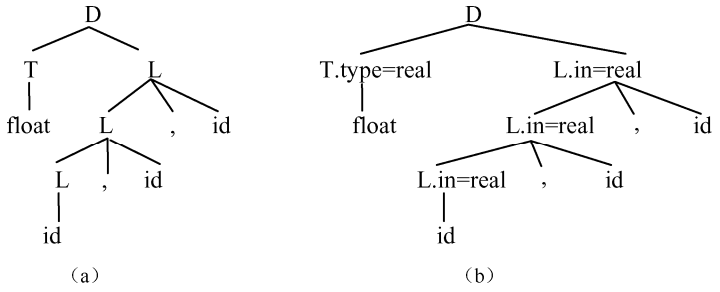


图 6-2

对照图 6-1（b）与图 6-2（b），考虑属性的计算方式。在图 6-1（b）中，一切结点的属性值都是根据它们子结点的属性值计算而得。从语义规则的形式看，都是根据重写规则右部符号的属性值来计算左部符号的属性值；而图 6-2（b）中，除结点 T 上的属性值是由其子结点上的属性值决定外，其他结点上的属性值都是由兄弟结点与父结点上的属性值来计算。例如，语法分析树第 2 层上结点 L 的属性 L.in 由其左兄结点 T 的属性 T.type 而来，第 3 层上结点 L 的属性 L.in 由其父结点的属性 L.in 而来。从重写规则看，其右部符号的属性由右部其他符号和左部符号的属性值来计算。这表明存在两类不同的属性，分别称为综合属性与继承属性。

① **综合属性**。如果语法分析树一个结点的属性，它的值是通过对它子结点的属性之值进行计算而求得，称该属性为综合属性。图 6-1（b）中属性 E.val、T.val 与 F.val 等都是综合属性。图 6-2（b）中语法分析树结点 T 的属性 T.type 也是综合属性。

② **继承属性**。如果语法分析树一个结点的属性，它的值是通过对它兄弟结点和/或它父结点的属性之值进行计算而求得，称该属性为继承属性。图 6-2（b）中几个结点 L 的属性 L.in 都是继承属性。

判断一个文法符号的属性是综合属性还是继承属性，显然只需在重写规则中看这个符号是在规则的左部还是在规则的右部。如果正在计算的一个属性是规则左部符号的属性，则这属性是综合属性，如果正在计算的一个属性是规则右部符号的属性，则这属性是继承属性。那么表 6.1 中的属性 N.lexval 与表 6.2 中的属性 id.entry 是哪类属性？这两个属性都是在词法分析阶段时确定的，

称为内在属性。内在属性通常是终结符号的属性，其值由词法分析时提供，往往把内在属性看作特殊的综合属性。有一类语义规则，即表 6.1 中的 print(E.val)与表 6.2 中的 addtype(id.entry,L.in)，它们是否是属性？如果是属性，又是哪类属性？显然这种函数调用形式的语义规则与用赋值语句计算属性不一致，它们一般是为完成某项功能而引进的。一般情况下，一个语义规则中所涉及的文法符号必须是在相应重写规则中出现的，对于在重写规则中不出现的符号，是不能在相应语义规则中计算它们的属性值的。函数调用形式的语义规则使得有可能引用不出现在相应重写规则中的文法符号的属性值，甚至可以改变某些非局部量的值，换句话说，就是可以产生副作用，一般把改变非局部变量值的函数调用称为有副作用。为了形式上一致起见，可以设想有一个虚拟属性 image，对它进行赋值。例如：

```
L.image:=print(E.val)
```

与 `L.image:=addtype(id.entry, L.in)`。

这样，一切语义规则都取相同的形式，显然，虚拟属性也作为综合属性。

请注意，一般来说，对于属性文法，其语义规则中的函数调用都不产生副作用，而语法制导定义中的语义规则允许产生副作用。语法制导定义可看成是对属性文法的扩充。

概括起来，文法符号有两类属性，即综合属性和继承属性。非终结符号不可能有内在属性，终结符号可以有内在属性，内在属性可看作特殊的综合属性，可以把函数调用形式的语义规则看作是对虚拟属性赋值，这个虚拟属性也是综合属性。

从上面的讨论可以看到，即使不画出语法分析树，也可以识别一个文法符号的属性是综合属性还是继承属性。就是说，主要看计算这个属性的符号是在重写规则的左部还是在右部。

一般地，对于一个文法符号 X 的属性 X.a 的计算呈以下形式：

$X.a:=f(X_i.b, X_j.c, \ldots, X_k.d)$

其中 X_i.b、X_j.c、…与 X_k.d 等都是文法符号的属性，例如，E.val:=E_1.val+T.val。这展示了属性 X.a 对属性 X_i.b、X_j.c、…与 X_k.d 的依赖关系，只有已经计算了属性 X_i.b、X_j.c、…与 X_k.d，才能计算属性 X.a。例如 E.val 必须在计算了 E_1.val 与 T.val 之后才能计算。因此，对一个输入符号串进行语义分析，计算各个属性时，必须按照这种相互的依赖关系而进行。为直观起见，可在语法分析树上对各个属性编号，并用有向边来指明属性计算的先后次序，箭尾处的属性先计算，箭头处的属性后计算。对于图 6-2（a）中的语法分析树可以画出如图 6-3 所示的有向图。

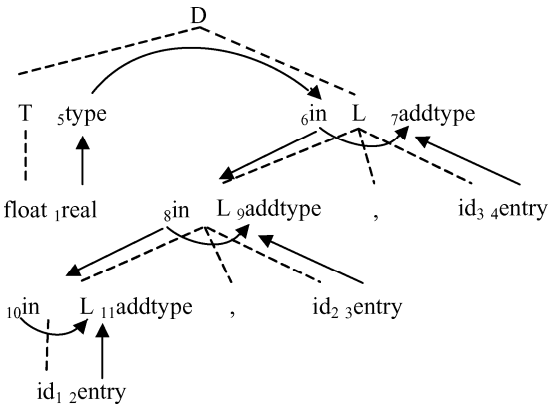


图 6-3

用来表达属性间相互关系的有向图称为**依赖图**。图 6-3 所示的有向图就是一个依赖图。利用

依赖图可以机械地推出语义规则的计算次序。

一般地，由语法制导定义描述的翻译可按下列步骤完成。

步骤 1　按基础文法（上下文无关文法）构造输入符号串的语法分析树。

步骤 2　从语法分析树构造依赖图。

步骤 3　对依赖图进行拓扑排序，得到语义规则的计算次序。

步骤 4　按上述计算次序计算语义规则，最终完成输入符号串的翻译。

图 6-3 中对各个属性进行了编号，它们计算的先后次序是否是拓扑排序？有向边指明了属性间的依赖关系，也就表示指明了计算的先后次序。从图可见，有向边都是从低序号结点指向高序号结点，即低序号结点在前，高序号结点在后，因此是一个拓扑排序，能够按语义规则计算一切属性。由这拓扑排序可以对应到如下的 C 型语言程序：

```
a5=real;                    /* T.type:=real */
a6=a5;                      /* L.in:=T.type */
addtype(id3.entry, a6);     /* addtype(id.entry, L.in)  */
a8=a6;                      /* L1.in:=L.in */
addtype(id2.entry, a8);     /* addtype(id.entry, L.in)  */
a10=a8;                     /* L1.in:=L.in */
addtype(id1.entry, a10);    /* addtype(id.entry, L.in)  */
```

显然，这些语义规则的计算结果将把类型 real 填入符号表中标识符 k、j 与 i 的相应条目中。

关于拓扑排序的概念，读者可以参考数据结构课程或其他相关的教材或参考资料。要说明的是，尽管可以有各种确定拓扑排序的算法，拓扑排序会不唯一，但效果是一样的。除非有向图中包含回路，不能确定是拓扑排序。只要不包含回路，依赖图作为一个有向图，总可确定拓扑排序而完成对属性值的计算。

请读者自行画出关于文法 G6.2[L]的输入符号串 5-3*1 的依赖图，自行对各个属性编号，写出拓扑排序。计算各个属性的一个拓扑排序所相应的属性值，计算次序如下：

```
N1.lexval:=5;   F.val:=N1.lexval;   T.val:=F.val;      E.val:=T.val;
N2.lexval:=3;   F.val:=N2.lexval;   T.val:=F.val;
N3.lexval:=1;   F.val:=N3.lexval;   T.val:=T1.val*F.val;
E.val:=E1.val-T.val;                print(E.val);
```

显然，最终的翻译结果是打印结果值 2。

前面讨论的是：基于根据输入符号串的语法分析树构造的依赖图，进行拓扑排序，从而确定语义规则的计算次序，这种方法称为语法分析树方法。除了采用语法分析树方法来确定语义规则的计算次序外，还可以采用基于规则的方法和忽略规则的方法来确定语义规则的计算次序。

按照基于规则的方法，在构造编译程序之前，事先通过对文法重写规则的语义规则进行分析，确定与每个重写规则相关联的语义规则的计算次序。这可以用手工完成，也可以用专门的工具自动完成。

采用忽略规则的方法时，语义规则的计算次序将由语法分析技术所确定，而与语义规则无关。

对 3 种方法进行比较可以看出：语法分析树方法由于在翻译时要构造依赖图，工作量大、功效低，不宜采用，而基于规则的方法虽然无需构造依赖图，但在事先需对重写规则相关的语义规则进行分析，确定语义规则的计算次序，也有相当的工作量，也不宜采用。事实上，本书将以忽略规则的方法为重点，也即结合语法分析技术，在分析输入符号串的同时完成语义规则的计算，边分析边计算属性，其顺序与语义规则书写顺序无关。请读者注意，语法分析技术有两大类，即自顶向下与自底向上分析技术，尤其是结合自底向上的 LR 分析技术实现语义规则的计算，由于

访问语法分析树结点的次序受分析方法的限制，这样确定的计算次序，将限制能实现的语法制导定义的种类。

下面介绍两类语法制导定义：S—属性定义与 L—属性定义。

对于一个语法制导定义，如果在其中仅使用综合属性，则称该语法制导定义是 **S—属性定义**。这里的 S 是英文单词 Synthesis（综合）的首字母，强调了 S—属性定义中的属性全都是综合属性。这表示语法分析树上一切结点的属性值都可由相应子结点的属性值来计算。因此，基于 S—属性定义生成的语法分析树可用自底向上方式进行注释，注释过程中，从叶结点到根结点的方向，按照语义规则求出每个结点的属性值。概括地说可结合 LR 分析技术来实现 S—属性定义。

由于是 S—属性定义，除了叶结点外的一切结点的属性都由子结点的属性值计算，从叶结点开始，可以计算出 S—属性定义确定的一切属性值。可以说，S—属性定义的语法分析树是可以注释的。

一个语法制导定义，在满足下列条件时称为 **L—属性定义**，即其中每个重写规则 $U::=X_1X_2\cdots X_n$ 的相应语义规则中，每个属性，或者是综合属性，或者是 $X_j(j=1,2,\cdots,n)$的满足这样条件的继承属性，即它们仅依赖于：

① 该重写规则中 X_j 左边的符号 $X_i(i=1, 2,\cdots, j-1)$的属性。

② U 的继承属性。

按照这个定义，显然 $X_j(j=1,2,\cdots,n)$的继承属性都是由其所在重写规则左边的符号 $X_i(i=1,2,\cdots,j-1)$的属性与左部 U 的继承属性来计算，因此可以理解此“L—属性定义”中的“L”，代表继承属性从左向右计算的“左”，也可以理解为“左边”的“左”（Left 的首字母）。如果关于某个输入符号串，画出包含分支结点符号串 $X_1X_2\cdots X_n$ 的语法分析树，如图 6-4 所示，回顾应用自顶向下语法分析技术构造语法分析树的过程，结点的建立顺序是从上到下，从左到右。换句话说，应用自顶向下分析技术进行语法分析时，按深度优先遍历顺序构造语法分析树。按照 L—属性定义计算 X_j 的继承属性时，结点 X_1、X_2、…、X_{j-1} 都已建立，且这些结点相关的属性也都已计算好。语法分析树上的每一个结点的继承属性所依赖的属性必定都已经计算好。L—属性定义的特征是：其属性总是可按深度优先遍历顺序计算。深度优先遍历的语法分析树结点属性的计算可用下列 C 型语言算法实现：

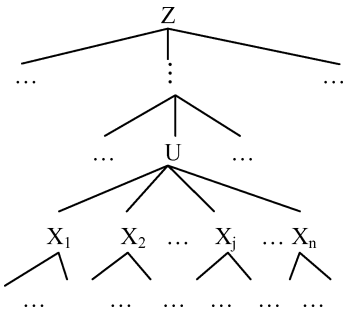


图 6-4

```
void DepthFirstVisit(结点类型 n)
{  for(结点 n 的从左到右的每个子结点 m)
   { 计算结点 m 的继承属性;
     DepthFirstVisit (m);
   }
   计算结点 m 的各个综合属性;
}
```

不是所有的语法制导定义都是 S—属性定义，因为有的语法制导定义中，需要引进继承属性，不会全部都是综合属性，L—属性定义就不是 S—属性定义。当然，S—属性定义一定是 L—属性定义，因为 S—属性定义中不涉及继承属性，自然满足相应条件。也存在一些语法制导定义不是 L—属性定义的，例如，假如一个语法制导定义中包含如下的重写规则与相应的语义

规则：

重写规则　　语义规则

X::=YZ　　Y.i:=f(Z.s);　Z.i:=g(Y.s);　X.s:=h(Y.i)

其中属性名 i 表示继承属性，属性名 s 表示综合属性，则这个语法制导定义一定不是 L—属性定义，原因是明显的：对于 Y.i:=f(Z.s)中文法符号 Y 的继承属性 Y.i，由相应重写规则中它右边的符号 Z 的属性来计算。

6.1.3　语法制导定义与翻译方案的设计

1．语法制导定义与翻译方案的例子

前面已讨论了基于语法制导定义进行翻译的问题，已经知道确定语义规则计算顺序的方法有 3 种。不论是哪一种，语法制导定义的设计人员都不必去考虑语义规则的计算次序。隐蔽了一些实现细节，在书写时不指明翻译时语义规则的计算次序，这是语法制导定义的一个特征。因此在设计语法制导定义时，不必考虑实现细节，尤其是不必考虑语义规则的计算次序，完全只需按照语言的语义去设计语法制导定义。例如，关于算术表达式文法 G6.4[E]：

E::=E+T|T　　T::=T*F|F　　F::=i　　F::=num　　F::=num.num

为检查运算的运算分量类型是否相同，可写出如下的语法制导定义，如表 6.3 所示。

表 6.3

重 写 规 则	语 义 规 则
$E::=E_1+T$	E.type:=if E_1.type=T.type then E_1.type else type_error
E::=T	E.type:=T.type
$T::=T_1*F$	T.type:=if T_1.type=F.type then T_1.type else type_error
T::=F	T.type:=F.type
F::=i	F.type:=gettype(i.entry)
F::=num	F.type:=integer
F::=num.num	F.type:=real

其中 num 表示整数（数字字符串），相应类型为 integer（整）型，num.num 是用一个小数点相连的两个整数，代表带小数点的实数，相应类型为 real（实）型。函数调用 gettype(i.entry)的功能是从符号表相应条目处取得 i 的类型属性，这条目由 i.entry 指明。表 6.3 中第一个语义规则表示：如果加法运算的两个运算分量 E（即 E_1）的类型 E.type(即 E_1.type)与 T 的类型 T.type 相同，则以 E_1.type 作为 E 的类型 E.type 的值，否则两个运算分量类型不相同，表达式 E+T 的类型为错误，取值 type_error。显然所有语义规则的作用与含义是一目了然的。

这里讨论属性文法的另一种形式：翻译方案。

【例 6.4】　试设计从中缀表达式生成后缀表达式的翻译方案。

通常的表达式是中缀表示的，即双目运算符写在两个运算分量之间，例如 a+b 与 a*(b+c)。后缀表达式中，双目运算符写在两个运算分量之后，例如，a+b 的后缀表达式是 ab+，而 a*(b+c)的后缀表达式是 abc+*。后缀表达式的特征是无括号，不再需要由括号来指明或改变运算的执行次序。

如何从中缀表达式生成相应的后缀表达式？例如有 x　op　y，其中 op 是双目运算符，要得到后缀表达式 x　y　op。显然，在处理 x 之后输出 x，处理 op 时不输出 op，而是在处理并输出 y 之后再输出 op。可以设计翻译方案如表 6.4 所示。

表 6.4

重 写 规 则	语 义 动 作
$E::=E_1+T$	{ print(+) }
E::=T	
$T::=T_1*F$	{ print(*) }
T::=F	
F::=(E)	
F::=N	{ print(N.lexval) }

翻译方案的书写表示与语法制导定义的书写表示有两个明显的区别：一是翻译方案由重写规则和语义动作两部分组成，语义动作写在大括号对{与}内；另一个区别，更为重要的是，翻译方案语义动作可以安置在重写规则右部的任意位置上，这就规定了语义动作的执行次序。因此，翻译方案涉及实现细节。必须注意的是，在计算某个属性时，它所依赖的一切属性必须已计算过，可以引用这些属性值。这表示翻译方案必须遵循深度优先遍历计算次序。

为了看清计算一个属性值时，是否已计算了该属性所依赖的一切属性值，可以对语法分析树加以扩充，即把语义动作设想为终结符号，并作为相应重写规则左部非终结符号对应结点的子结点，加到语法分析树上。为有所区别，可使用虚线连接。例如，假定有输入符号串 5*(3+1)，按表 6.4 中的翻译方案可画出语法分析树如图 6-5 所示。输出的将是 5 3 1 + *，即 5*(3+1)的后缀表示。

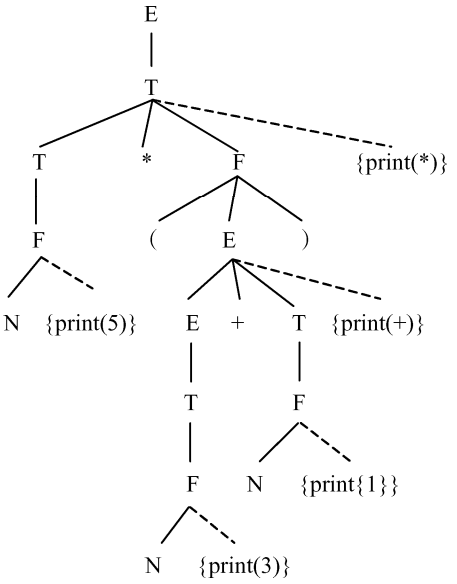


图 6-5

翻译方案的另一个例子如表 6.5 所示。

表 6.5

重 写 规 则	语 义 动 作
L::=E	{print(E.val) }
$E::=E_1+T$	$\{E.val:=E_1.val+T.val\}$
$E::=E_1-T$	$\{E.val:=E_1.val-T.val\}$
E::=T	{E.val:= T.val }
$T::=T_1*F$	$\{T.val:=T_1.val*F.val\}$
$T::=T_1/F$	$\{T.val:=T_1.val/F.val\}$
T::=F	{T.val:=F.val }
F::=(E)	{F.val:=E.val }
F::=N	{F.val:=N.lexval }

表 6.4 与表 6.5 中翻译方案的语义动作都在重写规则右部的右端，事实上，可以在重写规则右部的任意位置上。下面讨论如何设计翻译方案。

对照表 6.5 与表 6.1，它们都是关于文法 G6.2[L]设计的。表 6.5 中的翻译方案与表 6.1 中的语法制导定义，仅在形式上有所区别，即一个称为语义规则，另一个称为语义动作，且语义规则没有大括号对括住，语义动作用大括号对括住。本质上，更重要的区别是：表 6.1 中的语法制导定义不涉及实现细节，而表 6.5 中的翻译方案则指明了语义动作的执行时刻，只是本例中，语义动作正好全部在重写规则右部的右端，可以看成从表 6.1 改写成表 6.5。

显然，如果一个语法制导定义是 S—属性定义，即其中涉及的属性全是综合属性，只需把每个语义规则改写成语义动作（用大括号对括住），并把该语义动作放在相应规则右部的右端，便能得到翻译方案。例如，对于重写规则与语义规则：

$E::=E_1+T$　　E.val:=E_1.val+T.val

将改写成：

$E::=E_1+T$　　{ E.val:=E_1.val+T.val }

一般情况下，语法制导定义中将既涉及综合属性，又涉及继承属性。为了保证在计算一个属性的值之前，它所依赖的一切属性之值都已计算好，即遵循深度优先遍历顺序计算约定，属性的计算应满足下列 3 个要求：

① 重写规则右部符号的继承属性必须在先于这个符号的语义动作中计算。

② 一个语义动作不能引用该语义动作右部的符号的综合属性。

③ 对于重写规则左部非终结符号的综合属性，只能在它所引用的所有属性都计算完之后再计算，计算该属性的语义动作通常放在重写规则右部的右端。

假如有如下的翻译方案：

$S::=A_1A_2$　　{ A_1.in:=1;　A_2.in:=2 }

A::=a　　{ print(A.in) }

输入符号串为 aa，当把语义动作看作终结符号，画出如图 6-5 所示那样的语法分析树时，可以立即看出该翻译方案的错误。请读者自行改正之前，画出相应的语法分析树。

上述有错误的翻译方案正是不满足第一个条件。

对于 L—属性定义，为其构造满足上述 3 个要求的翻译方案总是可能的。

下面讨论如何结合语法分析技术来设计翻译方案。

2. 基于自底向上语法分析技术的翻译方案设计

按自底向上语法分析技术，是从输入符号串出发，以它们为末端结点符号串，试图向上构造语法分析树，使得根结点正好是识别符号。在每一分析步时，都是以当前正构造部分中，某一构成句柄（或其他被归约的短语）的结点符号串作为分支结点符号串，向上构造分支与分支名字结点，而综合属性是从一个分支的分支结点上的属性值来计算其父结点的属性值。显然，当一个语法制导定义是一个 S—属性定义的，即所涉及属性全是综合属性时，非常适合采用自底向上分析技术，由 S—属性定义改写而成的翻译方案情况一样。表 6.5 中展示的翻译方案就是这样。

翻译方案是涉及实现细节的一种描述形式。为了提高功效，可以结合自底向上的 LR 分析技术改写表 6.5。

改写思路如下：应用 LR 分析技术实现自底向上分析时，引进了分析栈。当分析栈顶的部分符号串未构成句柄时，把当前输入符号连同分析表元素确定的状态下推入分析栈；当构成句柄时，则进行归约，把分析栈顶形成句柄的部分上退去，并把所归约成的非终结符号连同从分析表 GOTO 部分取得的状态下推入分析栈。这表明，只需扩充分析栈，使栈中元素不仅包含状态与文法符号，还包含文法符号的属性值。在归约，即处理完重写规则右部的全部符号时，可以知道关于符号及相应属性值的信息。例如，假设分析栈名 S，由数组实现，分析栈顶由计数器变量 top 指向。当分析栈 S 的内容如图 6-6 所示时，栈顶形成句柄的符号串为 T*F，显然，T.val 在 S[top-2]中，而 F.val 在 S[top]中。因

状态	文法符号	属性值	
	F	val=1	←top
	*		
	T	val=3	
	-		
	E	val=5	
	#		

图 6-6

此，语义动作

```
T.val:=T1.val*F.val
```

可实现如下：

```
S[ntop].val:=S[top-2].val*S[top].val
```

其中 ntop=top−2，即上退去形成句柄的分析栈顶部分，且下推入新元素后，分析栈新栈顶的位置。一般情况下，ntop=top−r+1，r 是按其归约的重写规则的右部长度。

表 6.5 可改写成表 6.6 如下。

表 6.6

重 写 规 则	语 义 动 作
L::=E	{ print(S[top].val) }
$E::=E_1+T$	{ S[ntop].val:=S[top-2].val+S[top].val }
$E::=E_1-T$	{ S[ntop].val:=S[top-2].val-S[top].val }
E::=T	
$T::=T_1*F$	{ S[ntop].val:=S[top-2].val*S[top].val }
$T::=T_1/F$	{ S[ntop].val:=S[top-2].val/S[top].val }
T::=F	
F::=(E)	{ S[ntop].val:=S[top-1].val }
F::=N	{ S[ntop].val:=N.lexval }

按此翻译方案，语义动作全在重写规则右部的右端，因此都在执行归约动作时执行。假定输入符号串是 5-3*1，分析过程可以如表 6.7 所示。为突出属性值的计算，而对状态不感兴趣，表 6.7 中，分析栈元素用二元组（文法符号，属性值）表示。如果一个文法符号无属性值，属性值用“—”表示，如（+, —）。为简化起见，表中不指明每一分析步执行什么动作，然而执行的是移入、归约，还是接受，是显而易见的。

表 6.7

步 骤	分 析 栈	输 入	重 写 规 则
0	(#,—)	5−3*1#	
1	(#,—)(N,5)	−3*1#	F::=N
2	(#,—)(F,5)	−3*1#	T::=N
3	(#,—)(T,5)	−3*1#	E::=T
4	(#,—)(E,5)	−3*1#	
5	(#,—)(E,5)(−,—)	3*1#	
6	(#,—)(E,5)(−,—)(N,3)	*1#	F::=N
7	(#,—)(E,5)(−,—)(F,3)	*1#	T::=F
8	(#,—)(E,5)(−,—)(T,3)	*1#	
9	(#,—)(E,5)(−,—)(T,3)(*,—)	1#	
10	(#,—)(E,5)(−,—)(T,3)(*,—)(N,1)	#	F::=N
11	(#,—)(E,5)(−,—)(T,3)(*,—)(F,1)	#	T::=T*F
12	(#,—)(E,5)(−,—)(T,3)	#	E::=E−T
13	(#,—)(E,2)	#	L::=E
14	(#,—)(L, print(2))	#	

显然对于 S—属性定义，用这种改写的方式设计翻译方案是合适而方便的，因为 S—属性定义中包含的全是综合属性。但对于既包含综合属性又包含继承属性的语法制导定义，情况复杂得多，要根据具体情况改写。但不论如何，属性的计算应遵循深度优先遍历约定，满足前面所列 3 个要

求。对于 L—属性定义，容易使改写成的翻译方案满足这 3 个要求。

现在考虑将关于文法 G6.3[D]的语法制导定义改写成翻译方案的情况。

关于文法 G6.3[D]的语法制导定义如表 6.2 所示，这是一个 L—属性定义，可以按前述 3 要求把它改写成如下的翻译方案：

```
D::=T { L.in:=T.type } L
T::=int {T.type:=integer }
T::=float { T.type:=real }
L::={ L1.in:=L.in } L1,id { addtype(id.entry, L.in) }
L::=id { addtype(id.entry, L.in ) }
```

如果把语义动作看作终结符号，在语法分析树中也建立相应结点，边用虚线表示，则对于输入符号串 float i, j, k 可画出如图 6-7 所示的语法分析树。

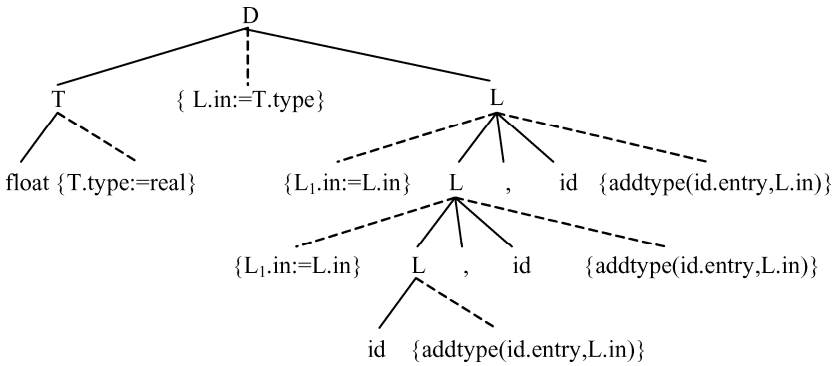


图 6-7

显然，按自底向上语法分析技术，语法分析树结点的建立顺序与属性值计算的拓扑排序将不一致，例如，在关于标识符 id=i 的函数调用 addtype(id.entry,L.in)时将有 L.in 无值。需要按语法分析技术建立语法分析树，然后再根据依赖图确定的拓扑排序计算各个属性值。合适的做法是类似于把表 6.5 改写成表 6.6 的情况，利用从分析栈取得属性值来改写。

先写出关于输入符号串 float i, j, k 的分析过程如表 6.8 所示。

表 6.8

步　骤	分　析　栈	输　入	重 写 规 则
0	(#,—)	float i,j,k#	
1	(#,—)(float,—)	i,j,k #	T::=float
2	(#,—)(T,real)	i,j,k #	
3	(#,—)(T,real)(id,i)	,j,k #	L::=id
4	(#,—)(T,real)(L,real)	,j,k #	
5	(#,—)(T,real)(L,real)(,,—)	j,k #	
6	(#,—)(T,real)(L,real)(,,—)(id, j)	,k #	L::=L, id
7	(#,—)(T,real)(L,real)	,k #	
8	(#,—)(T,real)(L,real)(,,—)	k #	
9	(#,—)(T,real)(L,real)(,,—)(id, k)	#	L::=L, id
10	(#,—)(T,real)(L,real)	#	D::=T L
11	(#,—)(D,—)	#	

其中，为简单起见，用标识符名代替符号表条目指针，例如，用 i 代替 id.entry。从表 6.8 中步骤 2 可见，按规则 T::=float 归约时，可直接把分析栈顶的内容(float,—)改成(T,real)；从步骤 4 可见，从 id 归约到 L 时，可从分析栈 S 的次顶元素 S[top-1]处取到类型 real，即 L.in；从步骤 6 与步骤 9 处可见，可从分析栈中 S[top-3]处取得类型 real，即 L_1.in=L.in。因此，可把表 6.2 中的语法制导定义改写成如下的翻译方案：

```
D::=T L
T::=int   { S[top].val:=integer }
T::=float { S[top].val:=real }
L::=L,id  { addtype(S[top].val, S[top-3].val);  S[ntop].val:=S[top-3].val }
L::=id    { addtype(S[top].val, S[top-1].val)}
```

其中，ntop=top−r+1，r 是按其归约的重写规则的右部长度。

这样省略了属性值 L.in 的计算，功效更高，但是与实现细节联系太紧密。另一种办法是按自底向上分析时，分析到每个标识符而还未取到类型属性的情况下，引进一个队列来暂存标识符的信息，直到处理完输入符号串时，把类型属性值添加到队列中所保存的一切标识符的相应符号表条目中，即写出下列翻译方案：

```
D::=N T L  { L.in:=T.type;
             while NOT EmptyQueue(Q)) do
             begin DeleteQueue(Q, q); addtype(q, L.in) end
           }
T::=int    { T.type:=integer }
T::=float  { T.type:=real }
L::=L,id   { EnterQueue(Q, id.entry) }
L::=id     { EnterQueue(Q, id.entry) }
N::=ε      { CreateQueue(Q) }
```

其中非终结符号 N 的引进，目的是为了建立新的队列。函数调用 CreateQueue(Q)的功能是创建新队列 Q（空队列），函数调用 EnterQueue(Q, x)的功能是把数据元素 x 存入队列 Q，函数调用 DeleteQueue(Q, y)的功能是把队列 Q 头一端的数据元素（取）出队列，并存入 y 中，而函数调用 EmptyQueue（Q）的功能是判别队列 Q 是否为空队列，如果是，回送 true（真），否则回送 false（假）。当按规则 L::=L,id 或 L::=id 归约时，将把 id.entry 存入队列 Q，而按规则 D::=NTL 归约时，把队列 Q 中的 id.entry 逐个取出，并在相应符号表条目中添加入类型信息 L.in。

3. 基于自顶向下语法分析技术翻译方案的设计

按自顶向下语法分析技术，是从识别符号出发，以它为根结点，试图向下构造语法分析树，使得语法分析树的末端结点符号串正好与输入符号串相同。这时识别出该输入符号串是相应文法的句子。一个翻译方案与自顶向下分析技术相结合，不言而喻，它所相应的上下文无关文法必须满足自顶向下分析技术的应用条件，即无左递归性与无回溯性。如果不满足应用条件，就必须先进行文法等价变换。例如，如果要为文法 G6.5[E]：

```
E::=E+T    E::=E-T    E::=T
T::=T*F    T::=T/F    T::=F    F::=(E)   F::=N
```

设计基于自顶向下分析技术的翻译方案，显然表 6.5 所示的翻译方案是不可行的。必须先进行文法等价变换得到文法 G6.5'[E]:

```
E::=TE'    E'::=+TE'   E'::=-TE'   E'::=ε
T::=FT'    T'::=*FT'   T'::=/FT'   T'::=ε
F::=(E)    F::=N
```

为了设计翻译方案，先看实例，通过实例观察分析设计思路。假定输入符号串为 3*2，可为其构造语法分析树如图 6-8（a）所示。

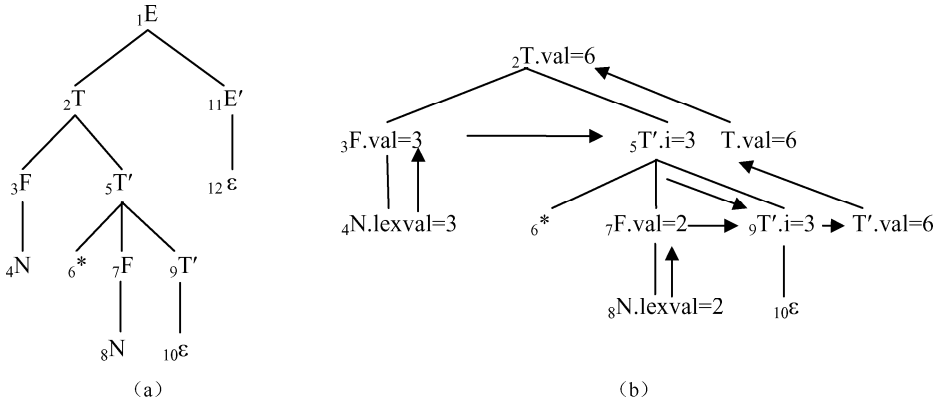


图 6-8

其中为各结点标记了建立的先后顺序。显然，为 F 引进的属性 val，其值源自 N 的属性 N.lexval，可有重写规则与语义动作如下：

```
F::=N  { F.val:=N.lexval }
```

因此，结点 3 上 F.val=3，结点 7 上 F.val=2。如何从这两个 F.val 计算 3*2？相关的重写规则是 T'::=*FT'，对左部的 T′引进属性 val。这时相关的结点序号分别是 5（T'）、7（F）与 9（T'）。

问题是如何把结点 3 处的 F.val 参与到此运算中，乘法运算结果又如何能传递到结点 1 处 E 的属性 E.val？一种合理的设想是把结点 3 上的 F.val 传递给结点 5 上 T′的属性，把此属性值与结点 7 上的属性值 F.val 进行乘法运算，结果存入结点 9 的某个属性，此后通过该属性把该运算结果上传到结点 5，再上传到结点 2 上 T 的属性，最终上传到结点 1 上的 E 的属性 E.val。根据这样的分析，可引进继承属性 T'.i，它是由左兄结点 F 的属性传递而来，或由左兄结点 F 的属性与父结点 T′的继承属性 T'.i 进行运算而得。为了上传，引进综合属性 T'.val。属性计算示意图如图 6-8（b）所示。可为 T′::=*FT′与 T′::=ε写出如下的语义动作：

```
T'::=*
     F { T'1.i:=T'.i*F.val }
     T'1{ T'.val:=T'1.val }
T'::= ε { T'.val:=T'.i }
```

为 T::=FT′写出如下的语义动作：

```
T::=F {T'.i:=F.val }
    T' { T.val:=T'.val }
```

类似地，考察输入符号串 7-6 的语法分析树，分析如何实现 7-6 的计算。最终可以为文法 G6.5′设计如下的翻译方案。

```
E::=T { E'.i:=T.val }
    E' {E.val:=E'.val }
E'::= +
      T  { E'1.i:= E'.i +T.val }
      E'1 { E'.val:=E'1.val }
E'::= -
      T { E'1.i:= E'.i -T.val }
      E'1 { E'.val:=E'1.val }
E'::= ε { E'.val:=E'.i }
T::=F { T'.i:=F.val }
    T' {T.val:=T'.val }
T'::= *
      F  { T'1.i:= T'.i *F.val }
```

```
     T1' { T'.val:=T'1.val }
T'::=/
     F  { T'1.i:= T'.i /F.val }
     T1' { T'.val:=T'1.val }
T'::= ε { T'.val:=T'.i }
F::=(
    E { F.val:=E.val }
    )
F::=N { F.val:=N.lexval }
```

按此翻译方案，可为输入符号串 7-3*2 构造指明属性依赖关系的注释分析树，如图 6-9 所示。

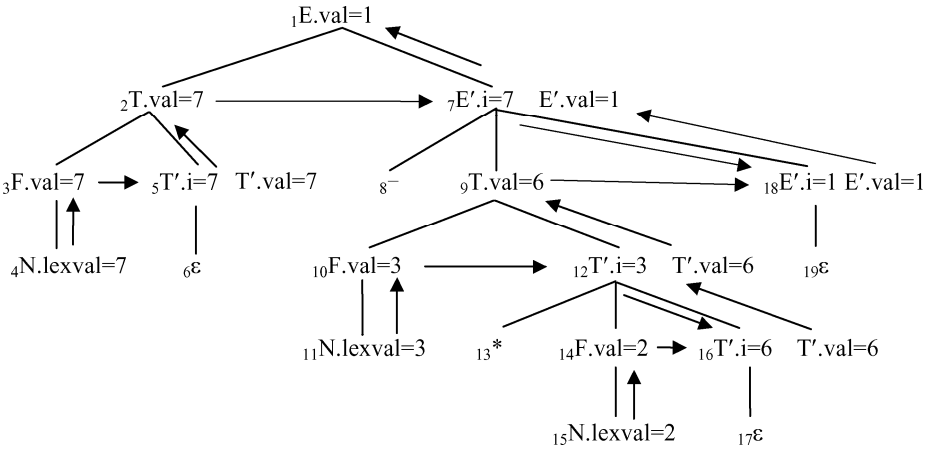


图 6-9

从对实际例子设计过程的观察与分析，可以看到继承属性 T'.i 与综合属性 T'.val 的作用，从而加深对这两类属性的理解。从这例子也可看到一个有效的解决问题的思维方法：从简单的实例出发，进行观察与分析，明确要解决的问题及解决思路，就可得到求解问题的答案。

对于一般的为左递归文法设计基于自顶向下分析技术的翻译方案，读者也可以对照下列的一般处理思想进行设计：

```
U::=U1 y  { U.s:=f(U1.s, y.a) }
U::=x     { U.s:=g(x. b) }
```

其中每个文法符号有综合属性，而 f 和 g 是任意的函数，则消去左递归后的翻译方案如下：

```
U::=x        { U'.i:=g(x. b)  }
     U'      { U.s:=U'.s }
U'::=y       { U'1.i:=f(U'.i, y.a) }
      U'1    { U'.s:=U'1.s }
U'::=ε       { U'.s:=U′.i }
```

例如，对于

```
E::=E1+T  { E.val:=E1.val+T.val }
E::=T     { E.val:=T.val }
```

其中，U=E，U.s=E.val，y=T，y.a=T.val，而 x=T，x.b=T.val，且 f(U_1.s, y. a)=E_1.val+T.val，而 g(x. b)=T.val，因此，按上述处理思想，有

```
E::=T  { E'.i:=T.val }
     E'  { E.val:=E′.val }
E'::=+
      T {E'1.i:=E'.i+T.val }
      E'1 { E'.val:=E'1.val }
E'::=ε  { E'.val:=E'.i }
```

对照为文法 G6.5′[E]设计的翻译方案，显然是相同的。

下面讨论基于属性文法实现语义分析功能。

6.1.4　类型表达式

一个高级程序设计语言程序包含数据结构和控制结构。如前所述，关于数据结构的语义分析功能有两个：确定类型与类型检查。为了进行这两项分析工作，有一个如何表达类型的问题。

众所周知，不同的程序设计语言其数据结构有不同的表示方式。例如，对于 C 语言，结构类型变量

```
struct date
{ int Day;  int Month;  int Year;
} d;
```

定义 d 为结构类型的变量，它有 3 个成员，即 Day、Month 与 Year，均为 int 型。然而，对于 PASCAL 语言，同样的结构类型变量 d 定义如下：

```
VAR  d: RECORD
          VAR  Day: integer;  Month: integer;  Year: integer
        END;
```

又如 PASCAC 语言中如下的定义：

```
VAR  A: ARRAY[—1..5, 2..4] OF integer;
```

定义 A 为一个整型二维数组，它的第一维有 7 个元素，下标是从—1 到 5，而第二维有 3 个元素，下标是从 2 到 4。然而对于 C 语言只能如下定义：

```
int  A[7][3];
```

虽然也定义一个整型二维数组，且两维元素的个数也分别是 7 与 3，但各维下标的下界只能从 0 开始，因此 PASCAL 语言有更大的灵活性。

属性文法作为实现语义分析与目标程序生成等部分的规范，关于数据类型的表示方式应与具体的程序设计语言无关。例如，结构类型的主要部分是成员名与成员变量类型，对于上述结构类型可以引进下列表示：

```
record ((Day×integer)×(Month×integer)×(Year×integer))
```

这种表示形式称为类型表达式。

类型表达式是属性文法中表达类型的、与具体程序设计语言无关的一种表示。它应能表达大部分程序设计语言中共同具有的数据类型，且有适当的扩充功能。一般来说，程序设计语言的数据类型有基本类型和构造类型。基本类型如整型、实型、逻辑型与字符型；构造类型则有数组、结构与指针等类型，为此引进类型构造符，以适应各种构造类型。

类型表达式定义如下。

① 基本类型表达式，它们是整型、实型、逻辑型与字符型，分别表示为 integer、real、boolean 与 char。另外有基本类型 type_error 和 void，分别表示错误类型和“无值”类型或回避类型的类型。

② 对类型表达式命名的类型是类型表达式。

③ 类型构造符作用于类型表达式的结果是类型表达式。类型构造符包括：

- 数组类型构造符 array。设 T 是类型表达式，则 array(I, T)是类型表达式，它对应于一个一维数组类型，数组元素类型是 T，而 I 是下标值集合，通常具有形式 low..up，指明所取值的范围是从 low 到 up，有时简化，仅给出 up，low 的默认值是 1 或 0。
- 记录类型构造符 record。设 N_1、N_2、…、N_n 是标识符，而 T_1、T_2、…、T_n 是类型表达式，则 record($(N_1\times T_1)\times(N_2\times T_2)\times\cdots\times(N_n\times T_n)$)是类型表达式，它对应于 C 语言结构类型，该结构有 n 个

成员，每个成员名是 N_i，成员变量的类型是 T_i 相关联的类型（i=1,2,…,n）。

- 指针类型构造符 pointer。设 T 是类型表达式，则 pointer(T)是类型表达式，它对应于一个指针类型，所指向对象的类型是 T。
- 函数类型构造符→。设 D_1、D_2、…、D_n 与 R 都是类型表达式，则 $D_1\times D_2\times\cdots\times D_n\rightarrow R$ 是类型表达式，它对应于从定义域 $D_1\times D_2\times\cdots\times D_n$ 到值域 R 的映射。

对于类型表达式，这里要说明两点：

① 类型表达式定义中的符号×表示卡氏积运算符，也可看作类型构造符。若 T_1 与 T_2 都是类型表达式，则 $T_1\times T_2$ 是类型表达式，对应于卡氏积。但在记录类型构造符中，允许 T_1 是标识符，即 N×T。

② 把函数也看作是一种类型，由于函数的形式参数名字无关紧要，仅对函数各个参数的类型构造卡氏积类型表达式。

函数类型表达式 $D_1\times D_2\times\cdots\times D_n\rightarrow R$ 与通常关于函数的数学表示类似。

一般情况下，类型表达式中可以包含变量，该变量的值是类型表达式，这对应于类型重载的概念。鉴于 C 语言中没有类型重载的设施，且为了简化，让函数参数的类型全是唯一确定的，不允许出现类型变量。

前面的翻译方案中已采用了类型表达式的概念，即 integer 与 real。例如，{T.type:=integer } 与{ T.type:=real }。

为了理解类型表达式的概念，能熟练地写出类型表达式，以 C 语言为背景，给出对照实例。

【例 6.5】 数组类型表达式的例子。

设有 C 语言数组变量说明如下：

```
int A [8];
```

它定义 A 为一维整型数组，共有 8 个元素。则与 A 相关联的类型表达式是 array(0..7, integer)。这是因为 C 语言数组元素的下标从 0 开始，8 个元素，所以下标是从 0 到 7。

【例 6.6】 记录类型表达式的例子。

设有 C 语言结构类型与结构类型数组变量说明如下：

```
typedef struct
{ int No;  /* 学号 */   char  name[8];   /* 姓名 */
  float ave_score;  /* 平均成绩 */
} StudentType;
StudentType StudentScore [40];
```

它定义一个结构类型 StudentType。由此类型标识符定义一个一维结构数组 StudentScore，它共有 40 个元素，每个元素都具有类型 StudentType。StudentType 所关联的类型表达式是：

```
record ((No×integer)×(name×array(0..7, char))×(ave_score×real))
```

与 StudentScore 相关联的类型表达式是：

```
 array (0..39, StudentType)
```

【例 6.7】 指针类型表达式的例子。

设有 C 语言变量说明：

```
StudentType  *sp;
```

其中，StudentType 如同例 6.6 中定义，此变量说明定义 sp 为一个指针变量，它所指向的对象具有类型 StudentType，则与 sp 相关联的类型表达式是：

```
pointer (StudentType)
```

【例 6.8】　函数类型表达式的例子。

按照 C 语言关于函数定义的语法定义，用户自定义的函数定义的一般形式如下：

```
T  f (T1  p1, T2  p2, … , Tn  pn)
{       …       }
```

作为映像，它作用于类型分别为 T_1、T_2、…、T_n 的 n 个参数，映像到类型为 T 的值。函数定义中重要的是参数的类型，形式参数名是什么无关紧要。因此，相关联的类型表达式是：$T_1 \times T_2 \times \cdots \times T_n \rightarrow T$。例如，为查找学号为 No 的学生之平均成绩 ave_score，可有 C 语言函数定义如下：

```
float SearchScore (int No, StudentType *sp)
{   …   }
```

其中 sp 是学生成绩记录表的指针，SearchScore 相关联的类型表达式是：

```
integer×pointer(StudentType) →real
```

对于系统提供的库函数，也可以写出相应的类型表达式。例如，判断一个字符是否字母字符的函数 isalpha 与求幂次的函数 pow，请读者自行写出与它们相关联的类型表达式。

从上面的讨论可见，类型表达式是一种表达类型、与具体程序设计语言无关的表示，它或者是与基本类型相关联的类型表达式，或者是由类型构造符作用于其他类型表达式而形成的类型表达式。

判断两个类型表达式是否相等的问题称为类型等价问题。通常关于类型有名字等价与结构等价两个概念。对于两个类型表达式名字等价，顾名思义，它们是相同的类型，或者是相同的类型构造符作用于相同的类型表达式，简单地说，它们在书写上是相同的。

两个类型表达式结构等价，指的是：当是基本类型表达式时，它们或者相同，或者是为相同的基本类型取的名；当是构造类型表达式时，则是相同的类型构造符分别作用于结构等价的类型表达式。

假定有下列 C 语言类型定义与变量说明：

```
typedef StudentType *ps1type;
typedef StudentType *ps2type;
typedef ps1type npstype;
ps1type p1;   ps2type p2;   npstype np;
```

请读者写出 ps1type、ps2type 与 npstype 及 p1、p2 与 np 所关联的类型表达式。试自行讨论 p1、p2 与 np 的类型等价性情况。

6.2　说明部分的翻译

C 语言程序的基本单位是函数定义，对 C 语言程序的翻译，核心是对函数定义的翻译。函数定义由函数首部与函数体组成。

【例 6.9】　试从 Num 个学生的成绩表中找出成绩最高的学生，返回其序号，且保存该最高成绩。

可以写出 C 程序如下：

```
int HighestScore(int Num, StudentType S[ ], float *score)      /* 函数首部 */
{  float Highest;  int k, n;                                   /* 说明部分 */
   Highest=S[1].ave_score;   n= S[1].No;                       /* 控制部分 */
   for(k=2; k<=Num; k++)
    if(S[k].ave_score>Highest)
```

```
        {  Highest=S[k].ave_score;
           n= S[k].No;
        }
      *score=Highest;
      return n;
   }
```

其中 StudentType 的定义见前面。

函数首部中提供函数参数与函数值的信息。语义分析之后，把这些信息存放在所谓的函数信息表中。上述函数 HighestScore 在函数信息表中将包含如下信息，即参数个数（3）、3 个参数的类型信息（int 型、StudentType 型数组与 float 型指针），以及函数值类型 int。

函数体通常由说明部分和控制部分组成。关于控制部分，编译程序将为其生成目标代码，这在后面讨论。关于说明部分，并不生成目标代码。说明部分的作用是把标识符所代表的对象与相应的类型等属性相关联，就是把标识符与类型等属性信息相关联，或者说，确定标识符的类型。语义分析的一个功能是确定类型，实质上是处理说明部分。例如，说明部分中的“float Highest;”与“int k, n;”，确定标识符 Highest 所相应的数据对象为 float 型，而标识符 k 与 n 所相应的数据对象为 int 型。编译程序了解了这些标识符所相应数据对象的类型，在控制部分中就可以为这些数据对象生成相应的操作码。例如，k<=Num 是 int 型的两个量相比较，而 S[k].ave_score>Highest 是 float 型的两个量相比较。假若让一个 int 型变量（比如说 n）与一个 float 型变量 Highest 比较，就必须把 int 型变量的类型先转换为 float 型，然后再与 float 型变量 Highest 进行比较。

本节讨论说明部分的翻译。说明部分通常包含有常量定义与变量说明等。事实上，函数定义也可看成是说明部分的一个组成部分，因为说明部分都是描述性的、被动性的，不像赋值语句、条件语句或者函数调用语句“主动”去执行一些操作命令。对说明部分处理的思路通常是：把相关的信息存放入一些表中。对于常量与简单变量，把类型信息添加入标识符在符号表的相应条目中，数组类型则在数组信息表中保存关于数组维数、各维上下界以及数组元素类型信息等，下面分别加以讨论。

6.2.1 常量定义的翻译

典型的 C 语言并没有常量定义机制，常量定义的功能由宏定义#define 来实现。例如，下列两个宏定义：

```
#define  pi  3.1416
#define  N  40
```

分别“定义”pi 与 N 为 3.1416 与 40。当程序中出现如下的语句：

```
L=2*pi*r
```

时，其中的 pi 代表常量 3.1416。C 语言编译程序实际处理该宏定义是在翻译之前，即词法分析之前进行预处理，把源程序中作为标识符的 pi 的每次出现，都替换为字符串“3.1416”。标识符 N 的情况相同。因此在编译后运行时的源程序中是查不到标识符 pi 与 N 的。

目前的 Turbo C2.0 版本中已扩充了 C 语言，可以应用常量定义，例如，C 语言程序中可以有如下的常量定义：

```
const float pi=3.1416; int N=40;  int M=N+5;
```

pi、N 与 M 将如同一般的标识符出现在源程序中。编译程序对常量标识符的处理思路如下：把常量定义中等号之后的常量值登录在常量表中，如果该常量值已登录过，则无需再登录。不论常量是否已登录，都回送该常量在常量表中的序号，然后为等号左边的标识符登录符号表新条目，在

该条目中填入常量标志、相应类型与常量表序号。如果如下定义新的常量标识符：

```
const int M=N;
```

情况类似，只需把 N 的常量表序号作为 M 的常量表序号，因为它们代表同一个常量。

由于常量本身可由它的书写形式确定类型，例如 3.1416 是实型，而 40 是整型，为简化起见，在常量定义中不指明类型，且在常量定义中，等号右边仅有常量或常量标识符。可为常量定义设计翻译方案如下：

```
ConstDef::=const CDT;
CDT::=CDT CD
CDT::=CD
CD::=id=num;
  { num.ord:=SearchConstT(num.lexval);
    id.ord:=num.ord;   id.kind:=CONSTANT;
    id.type:=LookType (num.lexval);
    add (id.entry, id.kind, id.type, id.ord);
  }
CD::=id=id1;
  { id.ord:=id1.ord;
    id.kind:=id1.kind;  id.type:=id1.type;
    add (id.entry, id.kind, id.type, id.ord);
  }
```

其中，函数调用 SearchConstT(value)的功能是：在常量表中查 value，如果未查到，把 value 录入常量表，不论查到与否，回送 value 在常量表中的序号，函数 LookType(value)的功能是取得 value 的类型；函数调用 add(entry, kind, type, order)的功能是：把种类 kind、类型 type 与序号 order 添加入由 entry 指明的符号表条目中，为了表达种类是常量，可以引进枚举值 CONSTANT，表示一个标识符的种类是常量。

常量的类型可以是整型、实型与字符型等，这由函数 LookType 来判断和取得。通常，编译程序把字符型（和字符串型）常量与算术型常量分开存放，C 语言就是这样处理。

常量的处理显然与词法分析部分紧密相关。请读者自行考虑：如何把词法分析部分对常量的处理与语义分析中对常量定义的处理相互配合，以提高功效。

不言而喻，通常对常量定义的翻译，实现了确定常量标识符的类型。

请读者自行关于下列常量定义：

```
const pi=3.1416;N=40;M=N;
```

验证上述翻译方案的正确性。

6.2.2　变量说明的翻译

变量说明的作用是向编译程序提供信息，告知各个标识符与什么类型等属性相关联，使其在控制部分中能生成正确的相应目标指令。变量说明的翻译实质上是确定类型，即把相关联的类型等属性信息填入或添加入符号表相应条目中。

以 C 语言为例，C 语言的变量说明包括简单变量说明与构造类型变量说明。前者有 int 型、float 型与 char 型，后者包括数组类型、结构类型与指针类型等。

简单变量可以有类型，构造类型的变量也可以有类型，例如，数组的元素可以具有 int 型或 float 型，甚至构造类型，因此对于变量的信息将包含种类与特征。对于 C 语言，一般来说，种类

可以是常量、简单变量、数组、结构与指针等，特征则相对于变量而言，指的是类型，基本类型可以是整型、实型与字符型，还可以是枚举型。

考虑到一切标识符的情况，为了在语义分析时能在符号表中保存各种必要的信息，其条目中也需要包含上述的种类与特征。这样，关于标识符的属性字结构，可设计得符号类部分呈以下形式：

0	种类	特征

其中，种类指明标识符所代表对象的种类，对于 C 语言可以是常量、简单变量、数组、结构、联合、文件、标号、指针、函数与类型，另外可以特别指明是主函数；特征对于不同的对象有不同的内容，对于常量与变量等，可以指明类型，基本类型是整型、实型与字符型，还可以是用户自定义的枚举型，对于函数，它可以指明是有值或无值、有参与无参，特征还可以指明是否形参，等等。

由于种类没超过 15 种，可以用 1～15 范围内的整数值代表种类，而为了处理特征的方便，每个特征用一个二进制数表示：1 表示是，0 表示否。

一个属性字通常取整个机器字或其倍数长，如词法分析部分中所述，属性字的统一长度为 32 位二进制数，其中左边 16 位是符号类，右边 16 位是符号值。考虑到上述种类与特征的引进，一个标识符的属性字可以如下设置：

$$\varepsilon_0\varepsilon_1...\varepsilon_4\varepsilon_5...\varepsilon_{15}\varepsilon_{16}...\varepsilon_{31}$$

其中，ε_0=0：表示非特定符号类。

ε_1…ε_4：表示种类，其值从 1 到 15，1 对应于常量，2 对应于变量，其后依次对应于数组、结构、联合、文件、标号、指针、函数与类型，令主函数特别对应于 15。

ε_5…ε_{15}：表示特征，ε_5=1 是整型（ε_5=0，非整型），ε_6=1 是实型，ε_7=1 是字符型，ε_8=1 是枚举型，ε_9=1 是成员变量，ε_{10}=1 是函数有值，ε_{11}=1 是形参，ε_{12}=1 是函数有参数，对于系统定义的标识符，ε_{15}=1。必要时，可以把ε_{13}设置为函数是否递归定义的标志。

显然，特征部分的各位二进制数可以进行组合。例如，作为实型形参的标识符，其属性字中将有ε_1…ε_4=0010，即 2，ε_6=1，ε_{11}=1。

对于例 6.9 中的变量说明：float Highest;，语义分析时，关于标识符 Highest 的属性字中种类=2（变量），特征中ε_6=1，其属性字如下：

0	0010	01000000000	Highest 在符号表中的序号

类似地，关于标识符 k 的属性字如下：

0	0010	10000000000	k 在符号表中的序号

对于例 6.9 中的函数首部：

```
int  HighestScore (int Num, StudentType S[ ], float *score)
```

关于标识符 Num 的属性字如下：

0	0010	10000010000	Num 在符号表中的序号

表达了 Num 是一个整型形参。关于 HighestScore 的属性字将如下：

0	1001	10000101000	HighestScore 在符号表中的序号

该属性字表达了 HighestScore 是函数名，该函数有形参，且有函数值，值为整型。

基于上述考虑，关于符号表条目引进数据结构，可以利用 C 语言的位运算。但为简单起见，

可以如下设计符号表条目类型：

```
typedef struct
{  int 特定符号标志位;                /* 0:非特定符号 */
   int 种类;                          /* 取值 1～15 */
   struct
   {  int 整型标志位;                 int  实型标志位;
      int 字符型标志位;               int  枚举型标志位;
      int 成员变量标志位;             int  函数有值标志位;
      int 形参标志位;                 int  函数有参数标志位;
      int 函数递归定义标志位;         int  待定位;
      int 系统定义标志位;
   } 特征;
} 符号表条目类型;
```

用数组实现符号表，可以设计如下：

```
符号表条目类型  符号表[MaxEntryNum];
```

假定标识符 HighestScore 在符号表中的条目序号是 N，则把确定类型时所得信息填入符号表时，只需执行下列 C 语言赋值语句（初值全为 0）：

```
符号表[N].特定符号标志位=0;          /* 非特定符号   */
符号表[N].种类=9;                    /* 函数 */
符号表[N].特征.整型标志位=1;         /* 函数值是整型 */
符号表[N].特征.函数有值标志位=1;     /* 函数有值     */
符号表[N].特征.函数有参数标志位=1;   /* 函数有参数   */
⋮
```

这些赋值工作可以由函数 AddType 或 Add 实现。

基于前面的讨论，下面讨论变量说明部分翻译方案的设计。

关于结构类型在后面讨论。不失一般性，现在按一个变量说明中仅有一个标识符，且数组仅一维，还假定下界恒为 1 的简化情况进行讨论。以下列文法 G6.6 [VariableDeclPart]为例设计翻译方案。

```
G6.6[VariableDeclPart]:
    VariableDeclPart::=D;
    D::=D; D
    D::=T  id | T  id[num]
    T::=int | float | char | T*
```

对每个变量说明 T　id 的处理思想如下。

首先确定类型，由此类型确定相应变量所占用存储区域大小，并确定当前分配的存储地址；然后为标识符 id 创建符号表条目，并在该条目中填入相应种类（变量标志）和特征（类型）及所分配的存储地址。

这里要说明几点。首先对于存储地址，在编译时刻不可能确定目标程序（包括数据区域）在存储器中的绝对地址，通常都是相对地址，即相对于所分配存储区域基址（首址）的位移量。可以设置一个变量，记录当前正在分配的相对地址，当把该相对地址加上刚为变量分配的存储大小时，得到的是为下一个变量分配的相对地址。按照 C 语言，字符型变量占 1 个字节的存储区域，整型变量占 2 个字节，而实型变量占 4 个字节。

其次对于构造类型，指针的处理大致相同，且指针变量占 4 个字节，而数组类型有所不同，

填入符号表中的类型是数组类型,其所占存储区域的大小要由数组元素个数和元素类型两者确定。关于数组的维数、各维上下界及元素类型等信息一般存放于数组信息表中。

词法分析时登录有标识符的信息，即在标识符表中登录了标识符本身及相应属性字，其中不包含关于该标识符相关联的类型等属性信息，进行语义分析时，确定标识符所代表数据对象的类型等属性信息。因此，这时标识符表扩充成了符号表，除了包含标识符本身的拼写信息外，还包含前面所讨论的种类与特征等信息，现在还包含相对地址的信息。

现在可为简化情况的变量说明部分设计翻译方案如表 6.9 所示。

表 6.9

重 写 规 则	语 义 动 作
VariableDeclPart::= D;	{offset:=0 }
D::=D;D	{ Enter(id.name, VARIABLE, T.type, offset); offset:=offset+T.width }
D::=T id	
D::=T id[num]	{ Enter(id.name, VARIABLE, array(num.lexval, T.type), offset); offset:=offset+ num. lexval*T.width }
T::=int	{ T.type:=integer; T.width:=2 }
T::=float	{ T.type:=real; T.width:=4 }
T::=char	{ T.type:=char; T.width:=1 }
$T::=T_1$ *	{ T.type:=pointer(T_1.type); T.width:=4 }

其中，VARIABLE 是枚举值，表明种类是变量。函数调用 Enter(name, kind, type, offset)的功能是：为名字是 name 的标识符在符号表中创建一个条目，除了一些基本内容（例如置“非特定符号标志位”等）外，把标识符名 name、种类 kind、类型 type 与相对地址（位移量）offset 填入该条目中。offset 用来存放当前的位移量，其初值为 0，这由语义动作 offset:=0 实现。细心的读者可能已发现，该语义动作书写在重写规则右部第一个符号 D 之前。一般来说，希望语义动作出现在重写规则右部的末端。为此，可使用ε规则，把原先的改写如下：

```
VariableDeclPart::=M D;
M::=ε                { offset:=0 }
```

读者可以以下列变量说明为例，验证上述翻译方案的正确性：

```
float Highest;  int k;  float *score;
char name [8];
```

上述翻译方案中仅定义一维数组，如果允许二维或更高维，则存在按行存放与按列存放的问题。这样，数组元素的地址将按不同的算法计算。C 语言将按行存放方式存放数组元素。关于这方面，将在本章后面关于赋值语句的翻译中讨论。

6.2.3 函数定义的翻译

C 语言程序由函数定义组成，每个函数定义实现相对独立且完整的功能。函数定义由函数首部和函数体组成。如同词法分析中所讨论的，函数体是一个作用域，函数体说明部分中的标识符，所对应的变量是函数体中的局部变量。不同的函数定义中的函数体是不同的作用域，并为每一个作用域开辟一个单独的数据区域，局部变量都存放在该数据区域中，并以相对于该数据区域首址的位移量标记局部变量的存放位置。

概括起来，函数定义的翻译要点是：解决标识符的作用域问题。

如同在词法分析一章所讨论的，标识符作用域问题的解决办法是利用栈。每当进入一个作用域，也就是说，扫描到一个函数定义的开始时，便为其安排一个单独的符号表，把该符号表首地址的指针下推入符号表指针栈 tableptr，当扫描完一个函数定义时，从符号表指针栈 tableptr 中上退去该指针，这样，符号表指针栈 tableptr 的栈顶，总是指向当前函数定义的符号表首地址。每处理函数定义中的一个说明语句，便在符号表指针栈栈顶指向的符号表内建立新条目。由于符号表的大小是可变的，可以用链表实现符号表，链表中每个结点对应于一个符号表条目。由于每个函数定义都有自己的单独的数据区域，为了记住当前正在处理的函数定义中下一个可用的相对地址，引进另一个栈：位移量栈 offset，其栈顶总是当前函数定义中的下一个可用的相对地址。

这里还需考虑全局变量的问题。C 语言中除了函数定义外，还可以有在函数定义外或函数定义之间的变量说明，它们当然不是函数定义的局部变量，而是整个 C 语言程序的全局变量。为简化起见，也不失一般性，让一切全局变量的说明集中在所有函数定义的前面。

关于简化了的 C 语言程序的重写规则可以设计如下：

```
P::=G L
G::=D; | ε
D::=D;D
D::=T  id | T id[num]
T::=int | float | char | T*
L::=L F | F
F::= id ( ) { D; S }
```

其中，F 代表函数定义，D 代表变量说明部分，S 代表控制语句部分。基于前面的讨论，设计翻译方案如表 6.10 所示。

表 6.10

重 写 规 则	语 义 动 作
P::=M G L	{ AddWidth(top(tableptr),top(offset)); pop(tableptr); pop(offset) }
M::=ε	{ t:=MakeTable(NULL); push (t, tableptr); push(0,offset) }
G::=D;	
G::=ε	
D::=D; D	
D::=T id	{ EnterId(top(tableptr), id.name, VARIABLE, T.type, top(offset)) ; top(offset):= top(offset)+T.width }
D::=T id[num]	{ EnterId(top(tableptr), id.name, VARIABLE, array(num.lexval, T.type), top(offset)); top(offset):=top(offset)+T.width*num.lexval }
T::=int	{ T.type:=integer; T.width:=2 }
T::=float	{ T.type:=real; T.width:=4 }
T::=char	{ T.type:=char; T.width:=1 }
$T::=T_1$ *	{ $T.type:=pointer(T_1.type)$; T.width:=4 }
L::=L F	
L::=F	
F::=N id( ) { D;S}	{ t:=top(tableptr); AddWidth(t, top(offset)); pop(tableptr); pop(offset); EnterFunc(top(tableptr), id.name, FUNC, t) }
N::=ε	{ t:=MakeTable(top(tableptr)); push (t, tabletptr); push(0,offset) }

其中重写规则 M::=ε的引进，目的是为了使得语义动作都放在重写规则右部的右端，并能执行相应的语义动作。关于 N::=ε，情况类似。函数调用 AddWidth（table,width）的功能是：把 table 指向的符号表一切条目中局部变量所占的存储区域大小，记录在该符号表的首部 header 中，其中 width 的值正是位移量栈 offset 栈顶元素的值，即下一个可用的相对地址，也就是已分配的存储区域大小。函数调用 MakeTable(outside)的功能是：创建新的符号表，并返回新符号表指针，参数 outside 是指向先前已建立的符号表指针，也就是最外层作用域的符号表或紧外层作用域的符号表指针，它的值将保存在新创建的符号表的首部 header 中。符号表首部 header 中除了上述指针外，还可以置有作用域嵌套深度的信息及函数定义的序号等。函数调用 EnterId(table, name,VARIABLE, type, offset)的功能是：在由 table 指向的符号表中为名字 name 建立新条目，在该条目中填入种类 kind（表示变量的枚举值 VARIABLE）、类型 type 和相对地址 offset。函数调用 EnterFunc(table, name, FUNC, NewTable)的功能是：为名字为 name 的函数在由 table 指向的符号表中创建新条目，该条目中有一个域，它的值是 NewTable，这是指向函数 name 的符号表的指针，其中的 FUNC 是枚举值，指明种类是函数。在一般情况下，可以判别函数是否有返回值与是否有参数，而置相应的标志位。top 与 push 是关于栈的操作，功能分别是取栈顶元素与下推入栈。

【例 6.10】 创建符号表的例子。

设有 C 语言排序程序 sort 如下。

```
/* sort: 对全局数组 A 的元素进行排序，排序结果在 B 中 */
#define  N  20
int A[N]={ 34,4,2,78,34,6,9,90,10,21,31,5,76,79,63,23,16,99,83,69};
void swap (int *a, int *b)
{  int temp;
   temp=*a; *a=*b; *b=temp;
}
void sort (int A[ ], int n )
{  int i, j, k;
   for (i=0; i<n-1; i++)
   {  k=i;
      for (j=i+1; j<=n-1; j++)
          if(A[j]<A[k]) k=j;
      if (k!=i)
          swap (&A[k], &A[i]);
   }
}
void main ( )
{  int B[N]; int k;
   for (k=0; k<N; k++)
      B[k]=A[k];
   sort (B, N);
   for (k=0; k<N; k++)
      printf ("%3d", B[k]);
   printf ("\n");
}
```

该程序把数组 A 的元素复制到数组 B 中，然后对 B 中元素进行排序。它由 3 个函数定义组成，有 3 个作用域，连同全局变量所属作用域，共有 4 个作用域，因此有 4 个符号表。为了符合上述重写规则，把说明部分作相应修改，例如把“int i,j,k;”改写成“int i; int j; int k;”等。简单起见，把形式参数看作函数的局部变量。根据表 6.10 中的翻译方案所生成的符号表及它们之间的相互联系，如图 6-10 所示。

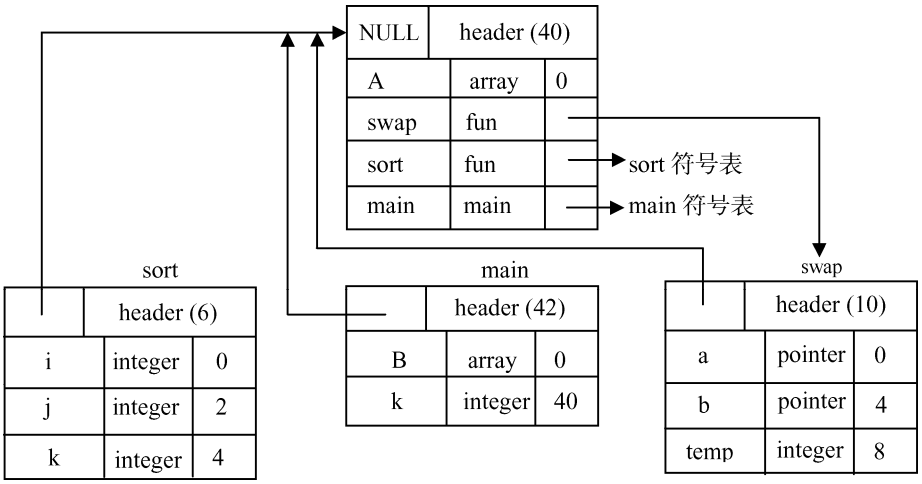


图 6-10

对上述程序的处理过程大致如下所述。

初始时，由重写规则 M::=ε的语义动作已建立了符号表，即全局变量符号表，且把指向该符号表的指针值下推入符号表指针栈 tableptr，把初始位移量 0 下推入位移量栈 offset。当处理全局变量说明“int A[N];”时，按相应重写规则 D::=T id[num]的相应语义动作，将在 top(tableptr)指向的符号表中为标识符 id (A)建立新条目，该条目中填入标识符名 A、种类 VARIABLE（变量）、数组类型（连同元素个数及类型的信息）与所分配的相对地址，此相对地址由 offset 栈的栈顶 top(offset)值确定。

为准备下一分配的相对地址，把 top(offset)的值增加数组元素个数×元素大小，即 num.lexval×T.width=20×2=40。考虑当处理 swap 的函数定义中的变量说明“int temp;”时的情况。这时已按重写规则 N::=ε的相应语义动作，由函数调用 MakeTable(top(tableptr))创建一个新的符号表，这是函数 swap 的符号表，该符号表的首部中包含一个指针，指向一个符号表，即指向由 top(tableptr)指向的符号表，也就是全局变量符号表。把指向 swap 符号表的指针值下推入符号表指针栈 tableptr。现在符号表指针栈顶所指向的符号表是 swap 的符号表。因此，处理“int temp;”时，按重写规则 D::=T id 的相应语义动作，将在由 top(tableptr)指向的 swap 的符号表中为标识符 temp 创建一个新条目，在该条目中填入标识符名 temp、种类 VARIABLE、类型 T.type 及相对地址 top(offset)。T.type 的值是 integer，它由重写规则 T::=int 的相应语义动作 T.type:=integer 设置。相对地址是 offset 栈顶元素中的值，即 top(offset)，这时是 0。只是把形式参数 a 与 b 看作是函数的局部变量，如同一般的局部变量那样处理，因此，swap 的符号表如同图 6-10 中所示，temp 的相对地址是 8。由于 int 型的存储字节数为 2，把 offset 栈顶元素的值加 2，成为 10。

关于控制部分的翻译在后面讨论，略过对控制部分 S 的翻译，处理完 swap 的函数定义时，将执行相应的语义动作：t:=top(tableptr)，使 t 的值是一个指针，该指针指向的是符号表指针栈栈顶元素所指向的符号表，即 swap 的符号表。函数调用 AddWidth(t, top(offset))的执行，使 t 所指向的符号表的首部 header 中填入 offset 栈顶元素的值，现在是 10，正是函数 swap 中的局部变量所占存储区域的总字节数。执行函数调用 pop(tableptr)与 pop(offset)，使得符号表指针栈栈顶元素再次指向全局变量符号表，而位移量栈 offset 的栈顶元素的值为下一个全局变量的相对地址，或说已处理全局变量所占存储区域的总字节数。函数调用 EnterFunc(top(tableptr), id.name, FUNC, t)的执行将为函数 swap 在当前符号表，即由 top(tableptr)指向的全局变量符号表中建立一个新条目。

该条目中填入函数名 swap、种类 FUNC(函数)与指向 swap 符号表的指针。swap 符号表与全局变量符号表之间的联系，在图 6-10 中展示无遗。

对于 sort 的函数定义与主函数定义的翻译，请读者自行完成。最终可有如图 6-10 所示的符号表示意图。

通过上例的讨论，可以明显体会到栈的作用。由于栈的先进后出性，对变量说明进行翻译时，总能在标识符的作用域所相应的符号表中为标识符建立相应的条目。

概括起来，在进行函数定义的翻译时要设置两个栈，一个是符号表指针栈，其栈顶元素所指向的符号表总是当前正处理的函数作用域；另一个是位移量栈，位移量栈栈顶元素的最终值，正好是相应函数作用域中全部局部变量所占存储区域的总字节数。当进入函数定义时，为它创建一个符号表，把指向它的指针与相对地址 0 分别下推入符号表指针栈 tableptr 与位移量栈 offset。当处理各个变量说明时，在 top(tableptr)指向的符号表中为标识符创建条目，该条目中设置好如下各域：标识符名、种类、类型与相对地址。该相对地址从 top(offset)取得，top(offset)的值增加了刚处理的标识符所对应数据对象的长度后，便准备好了下一个相对地址，处理完说明部分时的 top(offset)，正是为整个函数定义作用域中的一切局部变量所分配存储区域的大小。函数体中的控制部分也翻译完时，整个函数定义处理完毕。这时，在为该函数设置的符号表首部中，填入所分配局部变量数据存储区域大小，上退符号表指针栈 tableptr 与位移量栈 offset，然后在当前作用域的符号表，即上退后 top(tableptr)所指向的符号表中，建立关于函数名的条目，在该条目中设置好该函数名及其他必要的信息域，特别是设置好指向该函数作用域符号表的指针。

对于包含形式参数和可能返回函数值的函数定义，可以对表 6.10 进行扩充，以得到更一般的翻译方案，例如，引进关于形式参数表的重写规则及相应的语义动作如下：

```
B::=B, f        /* B代表形式参数表  */
B::=f           /* f代表形式参数指明  */
f::=T id {EnterId(top(tableptr), id.name, FORMALPARAM, T.type, top(offset));
          top (offset):=top(offset)+T.width  }
```

请读者自行思考如何扩充到形式参数是数组与指针类型的情况，以及如何扩充，使得在符号表条目中添加函数值的类型的信息。

6.2.4 结构类型的翻译

当一个数据对象是数组类型时，它所包含的一切元素都是具有相同类型的，然而结构类型与数组类型有很大的不同，对于结构类型的变量，有两点特别之处：

- 由不同类型的成员变量组成，因此各个成员变量所占存储区域大小不同；
- 必须通过成员名对成员变量进行存取，而不同的结构类型可以有相同的成员名。

鉴于上述两点，一个结构类型变量 R 的成员变量 m，必须用 R.m 这种表示法来表示对 R 的成员 m 访问。通常，为结构变量内各成员变量所分配存储区域地址，是相对于为该结构类型变量所分配存储区域的首地址的，即为一个结构类型变量分配的存储区域是一个整体，一个结构类型看作一个作用域，因此为结构类型建立自己的符号表。对于结构类型的翻译方案可设计如下。

```
T::=struct J {D}  {  T.type:=record(top(tableptr)); T.width:=top(offset);
                     pop(tableptr); pop(offset)  }
J::=ε {t:=MakeTable(NULL); push(t, tableptr);  push(0, offset) }
```

其中，非终结符号 D 与前面定义的 D 有相同的意义，非终结符号 J 的引进，目的是使得能执行为结构类型建立新符号表的语义动作。重写规则 J::=ε的语义动作包含 t:=MakeTable (NULL)，是因为把结构类型看作一个作用域，独立地翻译。如果作为整个程序或函数定义中的一部分，有外层作用域，应改为 t:=MakeTable(top(tableptr))。

建议读者自行以下列结构类型变量说明为例，考察翻译过程：

```
struct { int No; char name[8];  float ave_score;
       } StudentScore;
```

尽管出现了作用域多层嵌套的情况，但由于使用了栈，不会出现任何问题。只是类型构造符 record，在这里不是作用于类型表达式，而是作用于符号表指针，通过该符号表指针访问所指向的符号表，从而取得结构类型中各个成员变量的名字及类型。

当把前面讨论的常量定义、变量说明、结构类型及函数定义的翻译方案综合在一起，可实现对说明部分的翻译。这一综合工作请读者自行完成。

下面讨论对控制部分的翻译。

6.3　类 型 检 查

对说明部分的处理，实质上是语义分析的类型确定，把标识符与类型等属性相关联，具体地说，是把类型等属性记录在相应标识符的符号表条目中。对控制部分的语义分析主要是进行语义处理，也就是依据所识别出的语法结构的语义，进行相应的处理，这时可以生成目标代码，也可以为代码优化而生成中间表示代码。不论是哪种处理方式，为了生成正确的可运行目标代码，必要的是进行类型检查，即检查运算符的合法性和运算分量类型的一致性（相容性）。

本节讨论类型检查的问题。

类型检查可以在编译时刻进行，也可以在目标程序运行时刻进行。目标程序运行时刻进行的类型检查称为动态类型检查。例如，检查数组元素的下标是否越界，即检查是否超出某维的上下界范围，如果数组变量说明“float　A[30];”把 A 定义为数组，那么，数组元素 A[i]中 i 的值只能在 0 到 29 之间（包括 0 与 29），否则便越界。如果对每个数组元素都进行下标越界检查，显然将大大降低运行功效，一般来说，动态类型检查都将导致目标程序运行功效降低，因此往往仅是诊断型编译程序所生成的目标程序进行动态类型检查。

编译时刻进行的类型检查称为静态类型检查。不言而喻，应尽可能完善与充实静态类型检查功能，尽可能保证目标程序运行时不会有类型错误。一个程序设计语言，如果其编译程序能保证：它所接受的程序在运行时不发生类型错误，这种程序设计语言便被称为是**强类型**的。

为了更好地进行类型检查，需要一种与程序设计语言无关的表示法，这就是类型表达式。如前所述，类型表达式有基本类型的类型表达式 integer、real、boolean 与 char 等，以及类型构造符作用于类型表达式产生的类型表达式。读者请注意另外两个类型表达式：void 与 type_error。前者表示无类型或回避类型的类型，后者是错误类型。

6.3.1　表达式的类型检查

表达式通常由运算符、运算分量及括号对组成，其中运算分量可以是常量、简单变量、具有基本类型的数组元素、指针所指向的基本类型变量以及函数调用等。表达式的类型检查，主要检

查表达式中运算的合法性与运算分量类型的一致性。通过类型检查，确定表达式的类型。试以下列表达式文法 G6.7[E]：

```
E::=literal | num | num.num | id
E::=E+E | E%E |E[E] | *E
```

为例，设计类型检查的翻译方案。

文法 G6.7 中，literal 表示字面上的量，即字符，而 num 是不包含小数点的数字字符序列，也即整数；num.num 表示包含小数点的数字字符序列，即带小数点的实数；%是 C 语言中的整除求余运算符。单目运算符*是去指针运算符，*p 将取到指针变量 p 所指向的数据对象。因此可以设计翻译方案如表 6.11 所示。

表 6.11

重 写 规 则	语 义 动 作
E::=literal	{E.type:=char }
E::=num	{E.type:=integer }
E::=num.num	{E.type:=real }
E::=id	{E.type:=GetType(id.entry))
$E::=E_1+E_2$	{E.type:=if E_1.type=E_2.type then E_1.type else type_error }
$E::=E_1\%E_2$	{E.type:=if E_1.type=integer and E_2.type=integer then integer else type_error }
$E::=*E_1$	{E.type:=if E_1.type=pointer(T) then T else type_error }

其中函数调用 GetType(p)的功能是：返回符号表中由 p 所指向条目所包含的类型信息。

一般来说，运算符不仅是加法 + 与整除求余%，还可以是更一般的双目运算符：–、*、/、&&与||等。可用 op 来表示一般的双目运算符（除了整除求余）。类型检查要求双目运算的两个运算分量类型相同，但对于算术运算 +、–、*与/不要求类型严格相同（一致），可以是类型相容，即可以类型不同，只需都是算术型的。对于算术运算 op，可设计语义动作如下：

```
E::=E1 op E2 { if E1.type=integer and E2.type=integer
               then integer else
               if E1.type=integer and E2.type=real
               then real else
               if E1.type=real and E2.type=integer
               then real else
               if E1.type=real and E2.type=real
               then real else type_error
             }
```

这样，尽管运算符 op 的两个运算分量类型并不相同，但由于是算术运算，int 型与 float 型相容，并不发生错误，仅在其他情况时才发生错误，取得类型 type_error。

当两个运算分量只是类型相容时，尽管不发生错误，但生成目标代码时，必须进行类型的转换，也即生成类型转换的目标代码，使两个运算分量有相同的类型，再进行运算。

这种由编译程序自动完成的隐式类型转换称为类型的强制，出于对程序运行结果的考虑，应该要求类型强制转换不会丢失信息。如果从实型转换为整型，一般将导致小数部分丢失。C 语言的处理是：从“低级”类型转换到“高级”类型，即从字符型转换到整型，从整型转换到实型，这样一般不会丢失信息。程序书写者可以在书写程序时显式指明类型的转换。C 语言中引进了类型转换运算符机制。例如，i+(int)x，其中 i 是 int 型，x 是 float 型，(int)是类型转换运算符，即把

x 转换成 int 型后与 i 作加法，当然也可以写(float)i+x，(float)是类型转换运算符，把 i 转换成 float 型后与 x 作加法。

不论是显式类型转换写成 x + (float)i，还是隐式地自动类型转换写成 x+i，都以类型转换的时间为代价。因此从功效的观点，对实型变量 x 赋以值 1，不要写作“x=1”，合适的写法是写作“x=1.0”。

【例 6.11】　表达式类型检查的例子。

设 i 与 j 为 int 型变量，x 为 float 型变量，则表达式 i*x+j 的类型检查步骤如下。因 t=i*x 中 E_1=i 为 int 型，其类型表达式为 integer，而 E_2=x 为 float 型，类型表达式为 real，因此 i*x，即 t 的类型表达式为 real，t+j 中 j 为 int 型，类型表达式为 integer，因此整个表达式 i*x+j 的类型表达式为 real。

6.3.2　语句的类型检查

语句的类型指的是语句执行之后结果值的数据类型。大多数常用的程序设计语言不要求语句的结果值，C 语言便是如此，因此让语句得到的结果值类型是 void，即无值类型或回避类型。如果发现语句中存在类型错误，则让语句得到类型 type_error，即错误类型。

尽管语句不涉及类型，组成语句的表达式却是有类型的，例如赋值语句的赋值号左部变量与右部表达式有类型、条件语句中的条件（表达式）也有类型，等等。语句的类型检查其实是对组成语句的表达式的类型检查。C 语言的控制语句除了基本的赋值语句外，还有条件语句、3 类循环语句以及复合语句。

这里仅列出与赋值语句、if 语句、while 语句、do-while 语句与复合语句相关的类型检查翻译方案如表 6.12 所示。

表 6.12

重 写 规 则	语 义 动 作
S::=id=E	{S.type:=if id.type=E.type then void else type_error}
S::=if(E)S_1 else S_2	{if E.type=boolean then if S_1.type=void and S_2.type=void then S.type:=void else S.type:=type_error else S.type:=type_error }
S::=if(E)S_1	{S.type:=if E.type=boolean then S_1.type else type_error }
S::=while(E)S_1	{S.type:=if E.type=boolean then S_1.type else type_error }
S::=do S_1 while(E)	{S.type:=if E.type=boolean then S_1.type else type_error }
S::=S_1;S_2	{S.type:=if S_1.type=void and S_2.type=void then void else type_error }

上述翻译方案的含义很直观。例如，第 2 个到第 5 个重写规则的语义动作明确指明：如果表达式 E 不是 boolean（逻辑）型，整个语句的类型就是错误的类型，即 type_error。E 的类型正确的情况下，只有内嵌语句类型正确，各个语句的类型才正确，取值 void。复合语句的语义动作指明，只有组成复合语句的一切内嵌语句的类型都正确，复合语句的类型才正确。至于第一个赋值语句的语义动作指明，必须赋值号左边与右边的类型相同，赋值语句的类型才正确。由于算术类型可仅相容，无须完全相同，可以如同前面讨论的那样，对语义动作作相应修改，当赋值号左边

与右边表达式类型相容，生成目标代码时将把右部表达式的类型强制（转换）到左部变量的类型。

关于 C 语言中的 switch 语句与 for 语句，类似地设计类型检查翻译方案。可以给出如下的重写规则：

```
S::=switch(E){CasePart; default:S }
CasePart::=CasePart; CaseS | CaseS
CaseS::=case C: S
S::=for(E; E; E) S
```

请读者自行思考给出类型检查的相应语义动作。

【例 6.12】 语句类型检查的例子。

设变量 i 与 j 的类型为 int，变量 c 的类型为 char，变量 x 的类型为 float。设有语句

```
S=if (i<j) x=c;
```

由于 i 与 j 均为 int 型，类型表达式为 integer，i<j 的类型表达式为 boolean，这样 S 的类型将取决于 x=c 的类型。与 x 相关联的类型表达式为 real，而与 c 相关联的类型表达式为 char，c 语言允许把字符型强制（转换）到实型，于是，赋值语句 x=c 相关联的类型表达式为 void。因此整个 if 语句的类型为 void。

关于函数的类型检查，指的是函数调用时类型的检查。函数在定义时规定了函数值的类型与函数参数的类型，函数调用时实际参数与形式参数必须在个数、顺序与类型上一致，其要点是函数调用时实际参数的类型必须与形式参数的类型一致。为了简单起见，假定函数只有一个参数，且以类型表达式的形式给出函数定义的重写规则，以避免程序设计语言的具体表示：

```
T::=T→T
```

这样把函数看作一种类型，在翻译方案中可添加相应的重写规则与语义动作如下：

```
T::=T1→T2 { T.type:=T1.type→T2.type }
```

关于函数调用的重写规则可写出如下：

```
E::=id (E)
```

关于函数调用的类型检查将有如下的重写规则与语义动作：

```
E::=id(E1) { E.type:=if id.type=s→t and E1.type=s then t else type_error }
```

这是显而易见的，由于函数 id 是类型 s 到类型 t 的映像，如果参数 E_1 的类型是 s 时，结果类型当然是 t，否则将是错误的类型 type_error。

【例 6.13】 函数类型检查的例子。

系统函数 itoa 的功能是把整数值转换成一个字符串，即与整数值相对应的数字字符串。例如当 k 的值是 123 时，函数调用 itoa(k)执行的结果返回数字字符串“123”。因此，函数 itoa 的类型可写作 integer→array(1..6, char)。这里假定：作为参数的整数值最大是 6 位数。这样，id=itoa, s=integer，而 t=array(1..6, char)。当 E_1=k 的类型为 int 时，相应类型表达式为 s=integer，返回结果的类型表达式必然是 t=array(1..6, char)。

6.4 目标代码的生成

从源程序结构来说，说明部分翻译之后，将进行控制部分的翻译。从语义分析角度看，确定类型与类型检查之后，接下来的工作是语义处理，或直接生成目标代码，或生成中间表示代码以备优化。本节讨论直接生成目标代码的问题。

目标代码生成部分一般称为**代码生成程序**，代码生成程序的输入是源程序的中间表示，这种

中间表示，或者是语法分析树，或者是下一章中讨论的中间表示代码。不论是哪一种中间表示的输入，为简便起见，都看作是符号序列，如同语法分析程序的输入属性字序列那样，而不考虑具体表示。在生成目标代码时源程序已经历了词法分析与语法分析，因此此时的输入是正确的，也就是说，不仅符号拼写正确，而且语法书写上也是正确的，不仅如此，由于已经历了语义分析阶段的类型检查，也已发现并校正了明显的静态语义错误。总之，当生成目标代码时，代码生成程序的输入是正确的。不言而喻，经代码生成程序的处理，将输出与源程序等价的低级语言目标程序。

为了实现目标代码生成，必须解决若干相关问题。

6.4.1　与目标代码生成相关的若干要点

编译程序的功能是把高级程序设计语言源程序翻译成等价的低级语言目标程序。要检查一个编译程序的正确性，最好的方式就是运行目标程序，查看是否达到预期的运行效果。不言而喻，为了能运行，目标程序的生成是一个重点，它不仅在书写等方面要正确，还必须注意如下几个方面：

- 目标指令均有效、可行；
- 为数据进行存储分配；
- 合理充分地使用寄存器；
- 为目标程序的运行作好准备。

这些方面涉及编译程序的总体设计，也涉及具体实现。下面讨论在目标代码生成中一些带有普遍性的要点。

1. 目标机目标语言的确定

不言而喻，目标程序与特定的计算机紧密相关，不同型号的计算机有着不同的特性，也配备有不同的低级语言：机器语言或汇编语言。因此首先必须确定目标机目标代码的形式，从而确定可选用的目标指令集与目标程序结构。

目标代码通常有如下的 3 种形式。

（1）目标机器代码形式。这种目标代码是特定目标机上的特定机器指令组成的目标代码，与目标机紧密相关。为了生成目标代码，必须了解目标机的指令系统，包括指令格式、指令种类与寻址方式，还必须了解寄存器的用法。

目标指令中可以是取绝对地址形式与目标模块结构形式两种。取绝对地址形式时，目标代码将装入存储器指定位置，也为数据指定特定的存储位置。因此一经装入，目标代码可立即运行。对于小程序而言，可以快速编译和运行，例如对于面向学生的编译程序可以采用这种形式处理。但在当前多任务运行的环境中，这种方式不可重定位，带来极大的不便，甚至不可行。常用的编译程序往往采用目标模块结构形式，例如，C 语言编译程序便是如此。如果读者打开存放编译结果的文件夹，就可以发现与源程序相应的目标模块，也就是以 obj 为扩展名的文件。C 语言编译程序的系统库程序文件以 lib 为其扩展名，其实也是可重定位的模块。目标模块结构用可浮动的相对地址编址，当需要运行时，由连接程序把目标模块重定位后连接在一起，形成一个绝对地址的可运行程序。采用这种方式的好处是灵活，可以分别编译，未编译好的目标模块才需要编译，目标模块可调用已编译好的其他目标模块。

（2）汇编语言程序形式。目标机器代码形式有快速编译与运行的优点，但是它要求透彻了解目标机器的指令系统等各个方面的具体特征细节，代码生成工作复杂繁琐，因此选择汇编语言程序形式。

汇编语言是一种符号语言，符号语言目标代码与机器语言目标代码相比较，具有易读性好的特点，且可以把运行时的存储分配等工作交由汇编程序完成，尤其是可以利用宏机制来简化代码

生成工作，例如关于输入输出目标代码的生成将大大简化。采用汇编语言程序形式，使代码生成的过程变得容易很多。

采用汇编语言程序形式的不足是：这时的目标代码需要先由汇编程序对它进行汇编，然后才能运行，而汇编语言依然是与机器相关的，不同的计算机系统其汇编语言差异甚大，并不适合学校教学的目的。

（3）虚拟机目标代码形式。为了避免需要花费较多时间和精力去了解特定计算机的机器语言或汇编语言，使得能有更多的时间去更好地掌握编译程序的实现要点，可以采用虚拟机目标代码形式。

顾名思义，虚拟机是一种臆想的计算机，按照教学要求，它可以具有所需要的各种特性，而与具体计算机无任何联系，其优点是可以灵活地随时扩充目标指令的种类。这样既可便于了解和掌握目标代码生成的基本原理，又可随时按教学需要进行扩充。可以把花在了解特定计算机性能或汇编语言书写约定上的大量时间节省下来，更好地理解目标代码生成的基本原理。具体虚拟机目标代码的例子如下。

设有C语言赋值语句：

```
L=2*pi*r;
```

操作数以符号名表示，相应的虚拟机目标代码如下：

```
MOV  #2,  r0
MPY  pi,  r0
MPY  r,   r0
MOV  r0,  L
```

设有C语言if语句：

```
if (a>b) max=a; else max=b;
```

相应的虚拟机目标代码如下：

```
MOV   a,   r0
CMP   r0,  b
CJ>   *+8
GOTO  *+16
MOV   a,   r1
MOV   r1,  max
GOTO  *+12
MOV   b,   r1
MOV   r1,  max
```

综上所述，本书为教学的目的选择虚拟机目标代码形式。与采用虚拟机目标代码形式相关的问题是对这种形式目标代码的检查，可以通过阅读虚拟机目标代码程序检查正确性，更好的方式是运行目标代码程序。这种运行目标代码程序的系统应由编译程序开发人员开发，本书第10章将介绍符号模拟执行虚拟机目标代码的解释程序的设计与实现，供读者参考。建议读者参照本书所学知识自行开发。

2. 语言结构目标代码的确定

编译程序把高级程序设计语言源程序翻译成等价的低级语言目标程序，组成源程序的每个语言结构被翻译成相应的目标机目标指令。因此实现编译程序的要点是：建立源程序语言结构与目标代码结构之间的对应关系。具体地说，为源程序中各类语言结构，依据其语义确定相应的目标机目标代码结构。显然，能否建立这样的对应关系直接关系到编译程序实现的成败。如果建立不起这种对应关系，便无法实现翻译，只要能建立这样的对应关系，便能正确地实现语义、实现翻译。不言而喻，目标机指令系统的特性决定了目标机指令选择的难易程度，指令系统的一致性和完备性对于对应关系的建立有着重要影响。如果目标机能以一致的方式支持各种数据类型，那么

对应关系的建立要容易得多。如果指令系统内涵丰富，目标机可以有若干选择来对应某一语言结构，可以选择效率高、速度快的一种。如前所述，由于本书考虑的是虚拟机作为目标机，因此总可以设计合适的虚拟机目标指令来建立与语言结构的对应关系。

3. 运行时刻存储管理

一个程序由数据部分和控制部分组成。数据部分涉及控制部分所处理的数据，控制部分实现对数据的处理。实现控制部分的目标代码存储在计算机存储器内，数据当然也存储在存储器内。事实上变量名可看作是为存储字取的名字，通过此名字来对存储字存取。这表明运行时刻，编译程序要为程序中的量（常量与变量）分配存储区域，一是为量指定地址；二是使其占用相应大小的存储字字节数。如前所述，采用相对地址形式，即相对于为一个作用域中数据所分配的存储区域的首地址的位移量，这相对地址及所占存储区域大小，在语义分析阶段确定类型时确定，可以从变量名（标识符）在符号表的相应条目中获得。

当考虑运行时刻的存储管理、分配存储区域时，应该考虑字节边界对齐问题。对于字节编址的计算机，对不同类型的量所分配存储区域的起始编址，必须符合边界要求。例如，字符型量占一个字节，可安排在任意的存储位置上，但整型量占两个字节，为其分配的存储区域的起始地址必须是偶数地址，不然将引起错误，对于实型量,情况类似。为了简化处理，C 语言编译程序把字符型（包括字符串型）量集中存放在一个存储区域中，字符型外的其他数据集中在一起。为了讨论简单起见，本书不讨论边界对齐问题。

尽管给定了各个变量的存储地址，运行时刻何时为某变量实际分配存储区域，何时释放所分配存储区域，并非全部相同，换言之，有一个运行时刻的存储管理问题。关于运行时刻存储分配，有静态分配与动态分配两大类存储分配策略。这些将在后面讨论。

4. 寄存器分配

计算机运算器的高速与存储器的低速形成一个强烈的反差，寄存器的引进缓解了这种状况。寄存器可用来保存中间计算结果，而且操作数在寄存器中的指令，一般比操作数在存储器中的指令要短些，执行速度要快些。计算机与存储器之间一般不直接打交道，因此，充分而合理地利用寄存器，对生成高效的目标代码十分重要。

编译程序实现时，应该注意到寄存器的使用约定。例如，对于 IBM370 型计算机，整数乘和整数除要求使用寄存器对，即偶序数和顺序下一个奇序数的寄存器。对于乘法，被乘数在偶/奇寄存器对的偶寄存器，乘数在该偶/奇寄存器对的奇寄存器中，而对于除法，被除数占据一个偶/奇寄存器对，除法之后，偶寄存器保存余数，奇寄存器保存商，等等。

除了遵守寄存器的使用约定、正确使用寄存器外，为了充分合理地使用寄存器，避免因寄存器个数有限带来的影响，应考虑程序运行到某处时，将有哪些变量的值驻留在哪些寄存器中，即哪些变量占用哪些寄存器，以及确定下一步寄存器的使用。

本书不拟涉及关于寄存器分配的太多细节，仅请注意下列两点：

- 尽可能使用寄存器，尽可能长时间地保留在寄存器；
- 应考虑可用寄存器的个数，例如，1 个、2 个，还是 16 个等。

假定有下列 C 语言赋值语句：

```
x=(a*b+c/d)/(d-e/f)+sqrt(g* (h+k));
```

如果可用寄存器为 3 个、2 个与 1 个，有何区别？请读者自行思考。

5. 求值顺序的选择

对一个表达式求值时，利用括号、运算符优先级及结合性，可确定运算的先后顺序。然而对于

a*b+c*d 这样的表达式，a*b 与 c*d 还是有哪一个先计算的选择。自然的要求是，适当地改变表达式中各运算的执行顺序以确定一个最佳求值顺序，使得需要来保存中间运算结果的寄存器个数较少。

不同的编译程序有不同的处理。一个典型的例子是：C 语言表达式(k++)*(k++)*(k++)当 k 的值是 3（或其他任何值）时的求值，在不同的编译程序环境下运行，有不同的计算次序，因而有不同的计算结果。另一个典型的例子是：函数调用中实际参数的计算次序，不同语言的编译程序采用了不同的计算次序。请注意，C 语言函数调用中的实际参数以从右到左的次序计算。例如，输出语句：

```
printf ("i++=%d, (j++)*2+i++=%d, j++=%d/n", i++, (j++)*2+i++, j++);
```

当 i=1 和 j=2 时的输出结果是：

```
i++=2, (j++)*2+i++=7, j++=2
```

而不是

```
i++=1, (j++)*2+i++=6, j++=3
```

这表明计算次序与书写顺序正好相反。

要选择最佳求值顺序是困难的。本书不讨论求值顺序问题。一般情况下，简单地按源程序书写顺序或内部中间表示生成的顺序生成目标代码。

6. 代码生成程序的设计

如前所述，编译程序有诊断型与优化型等不同类型。不同的编译程序也有不同的设计指标，例如，可重定目标的、寄存器优化使用与诊断型的，等等。一个代码生成程序可以采用各种不同的生成算法来实现目标代码生成。对所有代码生成程序而言，最重要的是生成正确的代码，同时易于实现、测试与维护也是代码生成程序的重要设计目标。

6.4.2 虚拟机

如前所述，为了避免了解具体计算机的细节特征，目标机目标代码采用一种臆想的虚拟机目标代码形式，可按照教学或学习中对目标代码生成的需要而设置虚拟机。下面给出虚拟机的基本情况。

假定虚拟机目标机按字节编址，4 个字节组成一个字，并有 n 个通用寄存器 R_0、R_1、…、R_{n-1} 可用。虚拟机指令为如下所示的二地址指令：

```
操作码    源    目的
```

其中，操作码指明进行何种操作，是符号表示形式的，例如，加法运算的操作码是 ADD，传送指令的操作码是 MOV，等等。源和目的是符号表示形式的数据域，对应于操作数（运算分量）。显然这种方式类似于通常的汇编语言指令形式。通常一个字的 4 个字节中存放一条指令，操作数占一个字节，一般一个存储字地址占两个字节，因此同一条指令中，不可能源和目的都是存储地址。这表明，指令中的源和目的必须由寄存器与存储地址及寻址方式组合起来指明。这里的虚拟机指令系统的寻址方式类似于汇编语言，如表 6.13 所示。

表 6.13

寻 址 方 式	形　　式	地　　址
绝对地址	M	M
寄存器	R	R
变址	D(R)	D+R 的内容
间接寄存器	*R	R 的内容
间接变址	*D(R)	地址为 D+R 之内容的存储字的内容
立即数	#C	常数 C

一个指令系统一般涉及指令格式、操作码种类与寻址方式。寻址方式如前所述，操作码种类假定包含下列运算和操作：反号 NEG、加法 ADD、减法 SUB、乘法 MPY、除法 DIV、比较 CMP、按关系 relop 为 true（真）转 CJrelop、整型转实型 ITOF、无条件控制转移 GOTO、返回 RETURN 与函数调用 CALL 等。因此可以引进下列指令：

NEG	T	T 的内容反号⇒T
ADD	S,T	T 的内容+S 的内容⇒T
SUB	S,T	T 的内容–S 的内容⇒T
MPY	S,T	T 的内容×S 的内容⇒T
DIV	S,T	T 的内容/S 的内容⇒T
CMP	S,T	S 的内容与 T 的内容比较
CJrelop	T	按 relop 为 true(真)转向 T （其中 relop 可以是<, ≤, >, ≥, ≠与＝）
ITOF	S,T	S 的内容从整型转换到实型⇒T
GOTO	T	无条件控制转移到 T
RETURN		函数调用结束后返回
CALL	F,N	以 N 个参数调用无值函数 F
CALL	F,N,T	以 N 个参数调用有值函数 F，T 中为返回值

其中控制转移指令中所转向的目标可以取各种形式，例如可以是相对寻址形式，形如*+N 与*–N，表示所在指令首址加或减 N 个字节，也可以是指令序号，这时序号用小括号对括住，还可以是对指令加的标号，以指明指令的位置。

6.4.3　控制语句的翻译

控制语句翻译的要点是：建立控制语句与相应目标代码的对应关系，具体地说，是依据各个控制语句的书写规则，尤其是依据控制语句的语义，确定实现一个控制语句的目标代码结构。

建立对应关系的思路是：由控制语句的语义，明确控制语句的执行步骤，从而确定目标代码的结构。下面以 C 语言控制结构为例讨论控制语句的翻译。

1．赋值语句的翻译

（1）赋值语句的语法定义形式

C 语言赋值语句的一般书写形式如下：

V=E;

其中 V 是左部变量，可以是简单变量、数组元素，也可以是结构的成员变量等。E 是右部表达式，表达式可以具有各种类型，包括整型、实型、字符型与逻辑型等。为简化起见，这里仅讨论右部是算术表达式的情况，且略去赋值语句末了的分号。

（2）赋值语句的语义描述

赋值语句的语义以口语形式可描述如下：把右部表达式 E 的值赋给左部变量 V。左部变量 V 与右部表达式 E 的类型要求相同，如果不相同，把右部表达式 E 的值的类型转换成左部变量 V 的类型，然后再赋值给左部变量 V。

（3）赋值语句的执行

赋值语句中的变量，可能是简单变量，也可能是数组元素或其他变量，当左部是数组元素时，需要计算变量的存储地址。因为 C 语言中自增自减运算与赋值表达式的存在，左部变量的地址，在右部表达式 E 计算之前计算还是计算之后再计算，可能产生不同的效果。例如，

```
j=2;  A[j]=(j++)*2;
```

如果左部变量的地址先计算，这两个语句执行的结果是把 4 赋值给 A[2]，但如果后计算，则将把值 4 赋值给 A[3]，又如，

```
j=2;  A[j]=(j=1)*2;
```

如果左部变量的地址先计算，执行结果是把值 2 赋值给 A[2]，否则是把值 2 赋值给 A[1]。因为当前的 C 语言编译程序，左部变量的地址都在右部表达式计算之后再计算，因此赋值语句的执行步骤如下：

步骤 1　计算右部表达式 E 的值；

步骤 2　如果 E 值的类型与左部变量 V 的类型不同，则进行类型转换，把 E 的值转换到 V 的类型；

步骤 3　计算左部变量 V 的存储地址；

步骤 4　把右部表达式 E 的值（可能经转换后的）赋值给左部变量 V。

（4）赋值语句目标代码结构设计

基于上述执行步骤，可设计赋值语句的目标代码结构如下：

- 计算右部表达式 E 的值的目标代码；
- 必要时对 E 的值进行强制类型转换的目标代码；
- 计算左部变量 V 的存储地址的目标代码；
- 把（类型转换过的）E 的值赋值给左部变量 V 的目标代码。

不言而喻，当左部变量 V 是简单变量时，无需生成计算存储地址的目标代码。

【例 6.14】　赋值语句目标代码的例子。设 x、y、z、a、b、c 与 d 都是简单变量，都用它们的名字表示相应的存储地址，对于赋值语句 x=y 可以有下列目标代码：

```
MOV  y,   R0
MOV  R0,  x
```

由于一个目标指令中不能包含有两个存储地址，因此必须通过寄存器传送，不能仅通过下列一个目标指令来实现：

```
MOV  y,   x
```

那将是错误的。

设另有赋值语句 z=a*b−c/d，将有目标代码如下：

```
MOV  a,   R0          a⇒R0
MPY  b,   R0          a*b⇒R0
MOV  c,   R1          c⇒R1
DIV  d,   R1          c/d⇒R1
MOV  R0,  R2          R0⇒R2，即 a*b⇒R2
SUB  R1,  R2          R2-R1⇒R2，即，a*b-c*d⇒R2
MOV  R2,  z           R2⇒z，  即 a*b-c/d⇒z
```

（5）赋值语句翻译方案的设计

考虑赋值语句左部变量仅是简单变量，且右部表达式中仅允许算术运算的简单情况。翻译方案设计思想如下。

首先考虑引进的属性，显然有两种：一个是代码，一个是值存放地址，因此关于非终结符号 E 引进 code 与 place 两个属性，而为非终结符号 S 引进属性 code。关于 E::=E+E 的语义动作设计如下。为标明是哪一个 E，写作 $E::=E_1+E_2$。首先为表达式 E_1+E_2 的计算结果 E 给出存储地址 t。由于寄存器的个数有限，通常以临时变量作为过渡，即以 t 1 、t2、…作为存放中间结果的地址。用 newtemp 表示取得下一个临时变量名，初始时为 t1，下一个为 t2 等，因此 E.place:=newtemp，

为 E 分配临时变量 t1，然后把左运算分量 E_1 的值从 E_1.place 传送入 t1，再把右运算分量 E_2 的值从 E_2.place 加到此 t1 中。因此，连同计算左右两个运算分量的目标代码，语义动作设计如下：

```
E.place:=newtemp;
E.code:=E1.code ‖ E2.code
       ‖ gencode("MOV", E1.place, E.place)
       ‖ gencode("ADD", E2.place, E.place)
```

其中，运算符||表示并置运算符，其操作是把两个操作数（字符串）并置。函数调用 gencode(op, source, destination)的功能是生成如下形式的目标指令：

```
op   source,  destination
```

如果是 a+b，则有目标代码如下：

```
MOV  a,   t1        /* MOV  E1.place,  E.place */
ADD  b,   t1        /* ADD  E2.place,  E.place */
```

其中，a 与 b 相应的重写规则为 E::=id，关于 E 并不生成目标代码，而存放 E 的值的地址 E.place 是从 a 与 b 的符号表相应条目中取得，即从对函数 lookup 的调用取得。函数调用 lookup(name)的功能是：在符号表中查找关于 name 的条目，回送该条目的指针。因此语义动作可设计如下：

```
p:=lookup(id.name);
if p≠NULL then            /*  肯定能查到，可不判别  */
begin
  E.place:=p→place; E.code:=" "
end
else error;
```

其中，id 的值的存储地址由 p→place 指明。

关于 S::=id=E 的语义动作的设计，先看例子。如果有赋值语句 x=a+b，将对应于目标代码：

```
MOV  a,   t1
ADD  b,   t1
MOV  t1,  x
```

存放 E=a+b 的地址是 E.place=t1，应有目标代码 MOV　t1，　x。因此一般情形下，把右部表达式 E 的值赋值给左部变量 id 的目标指令是：

```
MOV  E.place,  p→place
```

此处的 p 同样通过函数调用 lookup(id.name)取得。生成赋值语句目标代码 S.code 的语义动作部分简单地是下列语句：

```
S.code:=E.code || gencode("MOV", E.place, p→place)
```

但若考虑到赋值语句 x=y 的情况，这时 E=y，关于它不需要生成目标代码，即 E.code=" "。用标识符代表存储地址，将有

```
MOV   y,   x
```

显然这是不正确的，必须先把 y 传送入一个临时变量（寄存器），然后再从它传送入左部变量 x，为此需判别 E.code 是否为空，如果是，则引进一个临时变量，生成相应的传送指令，并以此临时变量作为 E 的存储地址。因此关于 S::=id=E 可有下列语义动作：

```
p:=lookup(id.name);
if E.code=" " then
begin
   t:=newtemp;
   E.code:=gencode("MOV", E.place, t);
   E.place:=t
end;
S.code:=E.code || gencode("MOV", E.place, p→place)
```

关于 E::=E−E 与 E::=E*E 等的语义动作可类似地设计，最终可有如下的翻译方案。

```
S::=id=E { p:=lookup(id.Name);
           if E.code=" " then
           begin
             t:=newtemp;
             E.code:=gencode("MOV", E.place, t);
             E.place:=t
           end;
           S.code:=E.code || gencode("MOV", E.place, p→place)
         }
E::=E1+E2 { E.place:=newtemp;
            E.code:=E1.code || E2.code
                   || gencode("MOV", E1.place, E.place)
                   || gencode("ADD", E2.place, E.place)  }
E::=E1-E2 { E.place:=newtemp;
            E.code:=E1.code || E2.code
                  || gencode("MOV", E1.place, E.place)
                  || gencode("SUB", E2.place, E.place)  }
E::=E1*E2 { E.place:=newtemp;
            E.code:=E1.code || E2.code
                  || gencode("MOV", E1.place, E.place)
                  || gencode("MPY", E2.place, E.place)  }
E::=E1/E2 { E.place:=newtemp;
            E.code:=E1.code||E2.code
                  || gencode("MOV", E1.place, E.place)
                  || gencode("DIV", E2.place, E.place)  }
E::=-E1 { E.place:=newtemp;
          E.code:=E1.code
                || gencode("MOV", E1.place, E.place)
                || gencode("NEG", E.place)              }
E::=(E1) { E.place:=E1.place;  E.code:=E1.code }
E::=id {  p:=lookup(id.Name);
          E.place:=p→place;  E.code:=" "
        }
```

按上述翻译方案，赋值语句 x=a*b−c/d 将有如下的目标代码：

```
MOV  a,   t1       a=>t1
MPY  b,   t1       a*b=>t1
MOV  c,   t2       c=>t2
DIV  d,   t2       c/d=>t2
MOV  t1,  t3       a*b=>t3
SUB  t2,  t3       a*b-c/d=>t3
MOV  t3,  x        t3=>x
```

由 newtemp 产生的临时变量名可以代表寄存器。假定每当需要时就可由 newtemp 产生一个新的临时变量名。如果寄存器个数有限，将涉及存储分配与寄存器分配问题。简单起见，假定寄存器个数不限。

（6）相关问题

对于赋值语句的翻译方案，应该考虑下列两个方面，即类型强制与变量种类的扩充。

① 类型强制。类型强制指的是把一种类型强制转换为另一种特定的类型。例如，赋值语句的右部表达式的类型与左部变量的类型不相同时，需把右部表达式值的类型强制转换成左部变量的类型。如果表达式中一个运算的两个运算分量类型不相同，“低级”类型就必须被强制转换成“高

级”类型，这是为了能正确地执行目标指令所必须的。众所周知，一个量的类型决定了它在存储中的表示（存放），一个算术运算的两个运算分量必须在存储中有相同的表示才能进行运算。因此，关于赋值语句重写规则 S::=id=E 的语义动作中必须判别赋值号两边的类型是否相同。当不相同时，需把右部表达式的类型转换成左部变量的类型，可修改如下。

```
S::=id=E  {  p:=lookup(id.name);
             if E.code=" " then
             begin
               t:=newtemp;
               E.code:=gencode("MOV", E.place, t);
               E.place:=t
             end;
             if p→type=integer and E.type=integer
             then
               S.code:=E.code || gencode("MOV", E.place, p→place)
             else if p→type=integer and E.type=real
             then begin
                    u:=newtemp;
                    S.code:=E.code
                          || gencode("FTOI", E.place, u)
                          || gencode("MOV", u, p→place)
             end
             else if …
             ⋮
          }
```

其中 FTOI 是扩充的操作码，其功能是进行从实型到整型的转换。省略部分未列出的内容是相比较的其他情况。右部表达式中类型强制的翻译方案请读者自行讨论。

② 从简单变量到数组元素的扩充。数组元素的处理与简单变量相比较，主要是必须进行数组元素地址的计算，目标代码中必须包含目标指令来计算数组元素地址。相应地，要有从该地址取得数组元素值的目标指令以及把值赋值到该地址所指明数组元素的目标指令。先看例子，

设有数组变量说明：

```
int A[10];
```

说明 A 是一个有 10 个元素的一维整型数组，假定其元素的下标从 1 开始，即其元素是 A[1]、A[2]、…、A[10]，现在计算数组元素 A[j]的存储地址。

计算数组元素的存储地址时，首先假定任何数组的全部元素存储在连续的存储区域内。因此对于第一个元素的下标是 1 的一维数组 A，元素 A[j]的存储地址是：

```
A[1]的地址+(j-1)×sizeof(int)
=(A[1]的地址-1×sizeof(int))+j×sizeof(int)=A[0]的地址+j×sizeof(int)。
```

即元素 A[j]的存储地址一般计算公式是：

```
A[0]的地址+j×sizeof(int)
```

这里的 A[0]是数组 A 并不存在的虚拟元素，其存储地址称为零地址。例如 A[3]的地址是 A[0]的地址+3×sizeof(int) =A[0]的地址+6。如果 A[1]的起始存储地址是 1002，则 A[3]的存储地址是 1006（如图 6-11 所示）。说明：对于 C 语言，数组下标从 0 开始，上述计算公式也适用，区别在于，这时 A[0]不是虚拟元素。

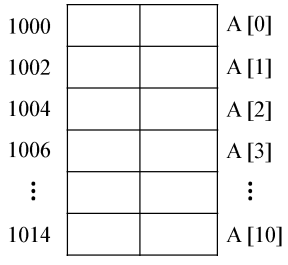


图 6-11

计算 A[j]的目标代码如下所示：

```
MOV   j,              t1
MPY   2,              t1  /*sizeof(int)=2 */
ADD   A[0]的存储地址, t1
```

地址计算结果在 t1 中。由于 A[j]的地址在 t1 中，把 A[j]的值存储在 t2 中的目标指令必须利用变址寻址方式，即 0(t1)。例如，对于 x=A[j]，将有目标指令：

```
MOV   0(t1),          t2
MOV   t2,             x
```

对于 A[j]=y，将有目标指令：

```
MOV   y,             t2
MOV   t2,            0(t1)
```

不言而喻，这里的 t1 和 t2 都应该是寄存器。

对于二维或更高维数组的元素之地址的计算，情况类似，只需计算出所给数组元素关于数组存储区域的首地址（即第一个元素的开始存储地址）的位移量，把它加上首地址便得到该数组元素地址。以二维数组为例，给出位移量计算公式如下。

设有数组变量说明“T　A[num_1][num_2];”，数组元素按行存放。假定各维下标下界从 1 开始，则数组元素 A[k][m]的存储地址的计算公式如下：

$$\begin{aligned} A[k][m]\text{的地址} &= A[1][1]\text{的地址} + ((k-1)\times num_2 + (m-1)) \times sizeof(T) \\ &= (A[1][1]\text{的地址} - (num_2+1) \times sizeof(T)) + (k \times num_2 + m) \times sizeof(T) \\ &= A[0][0]\text{的地址} + (k \times num_2 + m) \times sizeof(T) \end{aligned}$$

这里同样地 A[0][0]是并不存在的虚拟元素，其存储地址称为零地址。在运行时刻已进行了存储分配，数组 A 的零地址是已知的地址常量，A[k][m]相对于虚拟元素 A[0][0] 的起始地址的位移量是(k×num_2+m)×sizeof(T)。例如对于以变量说明 int B[3][4];定义的数组 B，其元素 B[k][m]的存储地址是 B[0][0]的地址+(k×4+m)×sizeof(int)=B[0][0]的地址+(k×4+m)×2。为计算数组元素 B[k][m]的地址的目标代码将是：

```
MOV   k,              t1
MPY   #4,             t1       /* 第二维上界 */
ADD   m,              t1
MPY   #2,             t1       /* sizeof(int)=2 */
ADD   B[0][0]的地址,  t1       /* B[k][m]的存储地址=>t1 */
```

假定赋值语句是 A[j]=B[k][m]，可有如下的目标代码：

```
MOV   k,              t1
MPY   #4,             t1
ADD   m,              t1
MPY   #2,             t1
ADD   B[0][0]的地址,  t1
MOV   0(t1),          t2       /*B[k][m]的值在 t2 中*/
MOV   j,              t3
MPY   #2,             t3
ADD   A[0]的地址,     t3       /*A[j]的地址在 t3 中*/
MOV   t2,             0(t3)
```

对于 m 维数组变量说明“T　A[n_1][n_2]…[n_m]”定义的数组 A 的元素 A[k_1][k_2]…[k_m]的存储地址计算公式如下：

$$\begin{aligned} A[1][1]\cdots[1]\text{的地址} + ((k_1-1)\times n_2 \times n_3 \times \cdots \times n_m + (k_2-1)\times n_3 \times n_4 \times \cdots \times n_m \\ + \cdots + (k_{m-1}-1)\times n_m + (k_m-1)) \times sizeof(T) \end{aligned}$$

$$= A[1][1]\ldots[1]\text{的地址} - ((\ldots((n_2+1)\times n_3+1)\ldots)\times n_m+1)\times \text{sizeof(T)}$$
$$+ ((\ldots((k_1\times n_2+k_2)\times n_3+k_3)\ldots)\times n_{m-1}+k_m)\times \text{sizeof(T)}$$
$$= A[0][0]\ldots[0]\text{的地址} - C + d$$

其中，$C=((\ldots((n_2+1)\times n_3+1)\ldots)\times n_m+1)\times \text{sizeof(T)}$是一个常数，可事先计算好。事实上 A[1][1]…[1]的地址-C 正是 A[0][0]…[0]的地址，即数组 A 的零地址，在运行时刻是已知的地址常量，而 $d=((\ldots((k_1\times n_2+k_2)\times n_3+k_3)\ldots)\times n_{m-1}+k_m)\times \text{sizeof(T)}$，通常按递推方式计算，是数组元素 $A[k_1][k_2]\ldots[k_m]$ 相对于虚拟元素 A[0][0]…[0]的存储地址的位移量。因此，

$A[k_1][k_2]\ldots[k_m]$的地址=数组 A 的零地址+d

如前所述，对于 C 语言，数组元素的下标从 0 开始，上述计算公式也适用，只是 A[0][0]…[0]不是虚拟元素。

现在考察数组元素地址计算的翻译方案的设计。

对于增加了数组元素的重写规则可写出如下：

```
V::=id | id[Elist]
Elist::=Elist][E | E
```

从语法分析角度看，这样的重写规则描述了数组元素的书写规则，但从翻译方案角度，根据这样的重写规则，难以设计出能满足深度优先遍历顺序计算属性的翻译方案。例如，处理下标表达式 Elist 与 E 时，无法得到关于相应数组的存储区域首地址（从数组名）的信息，也难以得到维数和上界的信息。为此改写如下：

```
V::=id | Elist]
Elist::=Elist][E | id[E
```

这样使得符号表中数组名条目的指针可以作为 Elist 的综合属性 array 而传递。考虑属性的引进。

非终结符号 V 有属性 place，表示其存储地址，V.place 将是执行目标代码后求得的数组元素存储地址。此外还引进属性 Atag，当 V.Atag 取值 0，表示 V 不是数组元素，取值 1，则是数组元素。

现在关于赋值语句的文法可给出如下：

```
G6.8[S]:
   S::=V=E          E::=id          V::=id          V::=Elist]
   Elist::=Elist][E   Elist::=id[E
```

这里表达式的处理不是重点，简化为仅单个变量。对照前面的讨论，为非终结符号 Elist 引进了属性 array，Elist.array 指明关于数组的信息，包括数组元素类型 T、各维上界及零地址等。另引进属性 dim，其值是相应的维数。可设计如下的翻译方案。

```
a)S::=V=E {  if V.Atag=0 then
             begin
               if E.code=" " then
               begin
                 t:=newtemp;
                 E.code:=gencode("MOV", E.place, t);
                 E.place:=t
               end;
               S.code:=E.code||gencode("MOV", E.place,V.place)
             end else
               S.code:=E.code||V.code
                   ||gencode("MOV", E.place, "0("||V.place||")")  }
b)E::=V {  if V.Atag=0 then
           begin
```

```
            E.place:=V.place;  E.code:=" "
          end else
          begin
            E.place:=newtemp;
            E.code:=V.code
                 ||gencode("MOV", "0("||V.place||")", E.place)
          end  }
c)V::=Elist] {  V.place:=newtemp;  V.Atag:=1;
              V.code:=Elist.code
                   || gencode("MOV", Elist.place, V.place)
                   || gencode("MPY", sizeof(T)存放处,V.place)
                   || gencode("ADD", Elist.array的零地址存放处,V.place) }
d)V::=id  {  p:=lookup(id.name);
           V.place:=p→place; V.Atag:=0    }
e)Elist::=Elist1][E
      {  t:=newtemp;
         k:=Elist1.dim+1;
         Elist.code:=Elist1.code || E.code
                   ||gencode("MOV", Elist1.place, t)
                   ||gencode("MPY", Elist1.array的第k维上界nk存放处,t)
                   ||gencode("ADD", E.place, t) ;
         Elist.array:=Elist1.array;   Elist.place:=t;
         Elist.dim:=k                                          }
```

其中第k维上界是关于$Elist_1$.array的，当处理到这里的语义动作时，已处理了下标表达式的前k－1维下标。

```
f)Elist::=id[E {  p:=lookup(id.name);
                  Elist.place:=E.place;     Elist.dim:=1;
                  Elist.code:=E.code;       Elist.array:=p   }
```

请注意，翻译方案中出现有“sizeof(T)存放处”与“零地址存放处”等不确切词语，这是因为：关于数组的信息如何存放并没有进行详细讨论，在编译时刻生成代码时，虽然可以取得这些信息，但都难以确切指明。不过，这样处理并不影响对问题的讨论。

【例6.15】 设有数组说明：

```
int  A[10], B[3][4];
```

试为赋值语句A[j]=B[k][m]应用上述翻译方案生成目标代码。

上述翻译方案应用于本例的赋值语句，可以生成下列目标代码：

```
MOV  k,                          t2
MPY  B的第二维上界4的存放处,     t2
ADD  m,                          t2
MOV  t2,                         t3
MPY  sizeof(int)的存放处,        t3
ADD  B的零地址存放处,            t3
MOV  0(t3),                      t4
MOV  j,                          t1
MPY  sizeof(int)的存放处,        t1
ADD  A的零地址存放处,            t1
MOV  t4,                         0(t1)
```

其中运算分量都用符号名（标识符）表示以易于阅读。要提醒的是，所引用的临时变量一般代表寄存器。为了简单起见，未对寄存器个数加限制。请读者自行比较按翻译方案生成的目标代码与

先前的代码之间的区别。为进一步加深对处理数组元素的翻译方案的理解，给出上述赋值语句的注释分析树如图 6-12 所示。

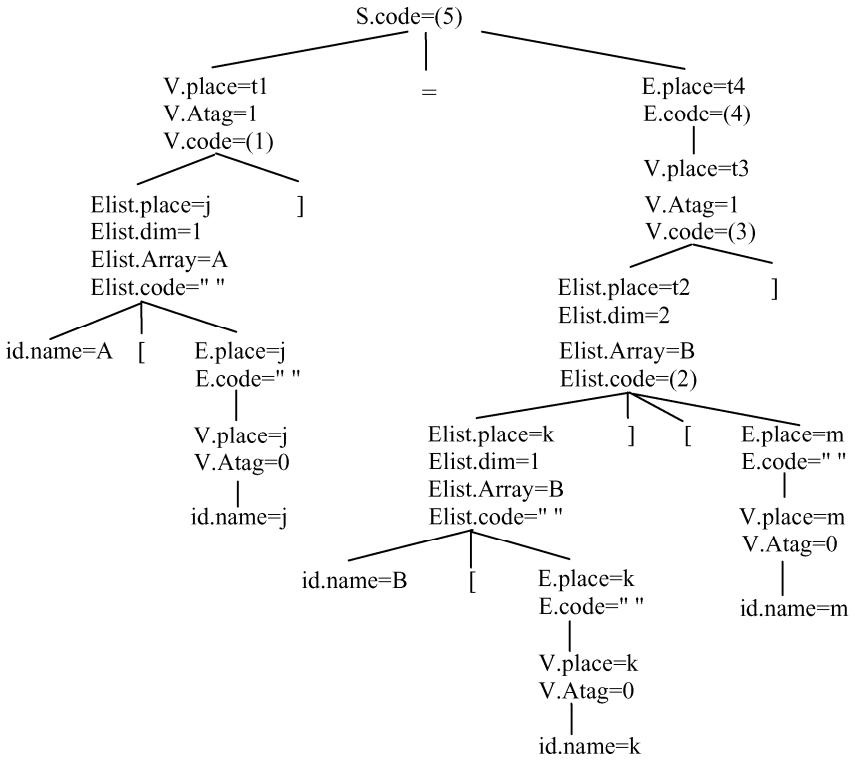


图 6-12

图 6-12 中代号（1）～（5）分别表示下列内容：

```
(1) =       MOV  j,                    t1
          ||MPY  sizeof(int)存放处,    t1
          ||ADD  A 的零地址存放处,     t1
(2) =       MOV  k,                    t2
          ||MPY  B 的第二维上界存放处, t2
          ||ADD  m,                    t2
(3) = (2) ||MOV  t2,                   t3
          ||MPY  sizeof(int)存放处,    t3
          ||ADD  B 的零地址存放处,     t3
(4) = (3) ||MOV  0(t3),                t4
(5) = (4) ||(1)|| MOV   t4,            0(t1)
```

细心的读者会发现：先为赋值语句左部变量安排临时变量（t1），后为右部表达式安排临时变量，这是因为属性的计算是按深度优先遍历顺序进行的，但在生成的目标代码中，左部变量存储地址的计算还是在右部表达式计算之后。

变量除了简单变量和数组元素外，还可以是结构的成员变量。对成员变量的翻译，要点是取到成员变量的类型和相对地址。读者必定记得存取结构的成员变量时，必须取 R.m 的形式，其中 R 为结构类型变量，m 为其成员变量，这表明，取到成员变量名 m 时已了解该成员变量 m 所属的结构类型变量 R 的信息。每个结构类型都有自己的符号表，每个成员变量的类型和相对于结构类

型变量存储区域首地址的相对地址等，就记录在该符号表的成员名目录中，因此，查找成员变量名，取到相应类型及相对地址的工作可以很方便地实现，甚至只需使用前面的 lookup 函数便可。

另一种变量是指针所指向的对象变量，关于这类变量的翻译问题，请读者自行考虑。只需结合一些实例，建立源语言结构与目标代码结构的对应关系，不难设计相应的语义动作。

这里要提醒的是，作为完整的赋值语句翻译方案，实际实现一个编译程序时，赋值语句的翻译应扩充表达式，更应考虑与赋值语句的上下文（包括函数调用）的翻译结合起来。确切地说，应与函数定义的翻译结合起来，因为赋值语句是函数定义的控制部分的一个组成。不言而喻，其他控制结构有类似的情况。

2. 选择结构的翻译

C 语言中选择结构有两类，即两者择一的 if 语句和多路选择的 switch 语句。本节重点讨论 if 语句的翻译。

（1）if 语句的翻译

① if 语句的语法定义形式。C 语言 if 语句的一般书写形式如下：

```
if (E) S else S
```

或

```
if (E) S
```

其中 E 是作为条件的表达式（逻辑表达式），而 S 是单个内嵌语句。

② if 语句的语义描述。if 语句的语义用口语形式可描述如下：根据逻辑表达式 E 的值是 true（非零）还是 false（零），确定应执行的内嵌语句 S，即当是 true 时执行紧跟 E 之后的内嵌语句 S（称为真部语句），然后跳过后面的 else 部分，执行 if 语句的后继语句，否则是 false，在第一种情形，跳过真部语句，执行 else 之后的内嵌语句 S（称为假部语句），然后执行 if 语句的后继语句。在第二种情形，则跳过紧随 E 之后的真部语句，立即执行 if 语句的后继语句。

③ if 语句的执行。根据 if 语句的语义，可知 if 语句的执行过程。以第一种情形为例，列出执行步骤如下。

步骤 1　计算表达式 E 的值；

步骤 2　判别 E 的值是否是 true（非零），如果是，则执行真部语句，然后无条件控制转移去执行 if 语句的后继语句；

步骤 3　否则，E 的值为 false（零），跳过真部语句，而执行 else 之后的假部语句，然后执行 if 语句的后继语句。

④ if 语句目标代码结构设计。基于上述执行步骤，不难确定 if 语句的目标代码结构。为了更直观起见，先画出目标代码结构示意图如图 6-13 所示，其中引进了 3 个标号：E.true、E.false 和 S.next，分别标志真部语句目标代码的起始位置、假部语句目标代码的起始位置与 if 语句后继语句入口处位置。S_T 与 S_F 分别表示真部语句和假部语句。相对照，if 语句的目标代码结构可设计如下：

```
            计算表达式 E 的值的目标代码
            当 E 之值为 false 时控制转向 S_F 入口处的目标代码
E.true:     真部语句 S_T 的目标代码
            无条件控制转向后继语句入口处 S.next 的目标代码
E.false:    假部语句 S_F 的目标代码
S.next:     后继语句的目标代码
```

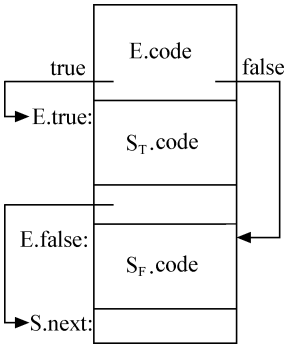


图 6-13

上述目标代码结构中判别的是条件不成立，即 E 的值为 false，这样会引起不便，即对于关系表达式，判别<时需改为≥，判别!=（不等）时需改为==（相等），等等。合适的是修改目标代码结构如下：

```
          计算表达式 E 的值的目标代码
          判别 E 的值的目标代码
          当 E 的值为 true 时控制转向 E.true 的目标代码
          无条件控制转向 E.false 的目标代码
E.true:   真部语句 S_T 的目标代码
          无条件控制转向 S.next 的目标代码
E.false:  假部语句 S_F 的目标代码
S.next:   后继语句的目标代码
```

【例 6.16】　if 语句目标代码的例子。设有 if 语句

```
if(a>b) max=a; else max=b;
```

其中 a、b 与 max 都是整型变量，为该 if 语句将生成目标代码如下：

```
      MOV   a,    R0    a⇒R0
      CMP   R0,   b     a 与 b 比较
      CJ>   L1          a>b 时转 L1
      GOTO  L2          a≤b 时转 L2
L1:   MOV   a,    R1    a⇒R1
      MOV   R1,   max   R1⇒max
      GOTO  L3
L2:   MOV   b,    R1    b⇒R1
      MOV   R1,   max   R1⇒max
L3:
```

⑤ if 语句翻译方案的设计。根据 if 语句目标代码结构设计，特别是参照图 6-13 中的示意图，可以很方便地写出翻译方案。类似于表达式的情况，为生成新的临时变量名，引进 newtemp，这里为生成新的标号，引进 newlabel。两种 if 语句的翻译方案可设计如下：

```
S::=if(E) S1 else S2
   { E.true:=newlabel;    E.false:=newlabel;   S.next:=newlabel;
     S1.next:=S.next;     S2.next:=S.next;
     S.code:=E.code
           || gencode("CMP", E.place, "#1")
           || gencode("CJ=", E.true)
           || gencode("GOTO", E.false)
           || gencode(E.true,":") || S1.code
           || gencode("GOTO", S.next)
           || gencode(E.false,":") || S2.code
           || gencode(S.next,":")              }
S::=if(E)S1
  { E.true:=newlabel;   S.next:=newlabel;
    E.false:=S.next;    S1.next:=S.next;
    S.code:=E.code
         || gencode("CMP", E.place, "#1")
         || gencode("CJ=", E.true)
         || gencode("GOTO", E.false)
         || gencode(E.true, ":") || S1.code
         || gencode(S.next, ":")              }
```

C 语言中用非零和零分别表示逻辑表达式值 true 和 false，这里用 1 表示非零，因此把 E.place 与 1 进行比较。另外，通过把 newlabel 赋值给 S.next，S.next 得到一个新的标号，因此，S.next 可以处理作为综合属性。如果把 S.next 处理作为继承属性，则它的值由上下文环境确定，即执行 if 语句的语义动作之前，已由 if 语句的外围上下文中给 S.next 一个标号值，不再在此语义动作内赋值。

⑥ 与 if 语句翻译相关的问题。除了上述问题外，与 if 语句翻译相关的问题还包括 if 语句中表达式 E 的翻译以及 if 语句目标代码中的回填问题。

（a）对于 if 语句中表达式 E 的翻译。if 语句中作为条件的表达式 E，本质上是一个逻辑表达式。一般来说，除了关系运算符外还将包含逻辑运算符，如逻辑与（&&）、逻辑或（||）与逻辑非（!）等，因此必须扩充相应的虚拟机操作码，例如引进 AND、OR 与 NOT，分别对应于逻辑与、逻辑或与逻辑非等。关于这些，很容易写出相应表达式的翻译：

```
E::=E1 && E2 {  E.place:=newtemp;
                E.code:=E1.code || E2.code
                     || gencode("MOV", E1.place, E.place)
                     || gencode("AND", E2.place, E.place)  }
E::=E1 || E2 {  E.place:=newtemp;
                E.code:=E1.code || E2.code
                     || gencode("MOV", E1.place, E.place)
                     || gencode("OR",  E2.place, E.place)  }
E::=!E1 {  E.place:=newtemp;
           E.code:=E1.code
                  || gencode("MOV", E1.place, E.place)
                  || gencode("NOT", E.place,  E.place)  }
```

当关系运算符出现在逻辑表达式中时，显然与逻辑运算符的情况不一样。例如，a>b，等价于由条件运算符构成的下列表达式：

```
(a>b)? 1:0
```

因此令 relop 表示任意一种关系运算符，可以有：

```
E::=id1 relop id2
{ E.place:=newtemp;  t:=newtemp;
  E.code:=gencode("MOV", id1.place, t)
       ||gencode("CMP", t, id2.place) || gencode("CJ"||relop, "*+8")
       ||gencode("GOTO", "*+12") || gencode("MOV", "#1", E.place)
       ||gencode("GOTO", "*+8") || gencode("MOV", "#0", E.place)     }
```

其中使用了相对地址形式“*+8”等，因为一条指令 4 个字节，“GOTO *+8”表示控制转向该指令下面的第 2 条指令。关于 true 与 false 类似地有：

```
E::=true  { E.place:=newtemp;
            E.code:=gencode("MOV", "#1", E.place)  }
E::=false { E.place:=newtemp;
            E.code:=gencode("MOV", "#0", E.place)  }
```

其中，"CJ" ‖ relop 表示把字符串“CJ”与字符串 relop 并置。例如，当 relop 是>时,"CJ"‖">"便是"CJ>"，表示当>时控制转移。

假定有逻辑表达式 k>1 && k<10，可以有如下的目标代码：

```
(1) MOV   k,    t2          (9)  CMP     t4,   #10
(2) CMP   t2,   #1          (10) CJ<     *+8
(3) CJ>   *+8               (11) GOTO    *+12
(4) GOTO  *+12              (12) MOV     #1,   t3
```

```
(5) MOV   #1,    t1          (13) GOTO    *+8
(6) GOTO  *+8                (14) MOV     #0,   t3
(7) MOV   #0,    t1          (15) MOV     t1,   t5
(8) MOV   k,     t4          (16) AND     t3,   t5
```

上述目标代码最终在临时变量 t5 中得到该逻辑表达式的值 true（1）或 false（0），当在外围上下文中使用该值时，将使用下列 3 条目标指令：

```
CMP       t5,          #1
CJ=       true（=1）时执行处
GOTO      false（=0）时执行处
```

考虑到 C 语言中逻辑表达式的短路处理，即当进行了部分计算已经能确定整个逻辑表达式的值时，可以不再对其余部分进行计算，从而提高功效。具体地说，当计算 A && B 时，若 A 为 false，则 A && B 必为 false，无需计算 B。只有当 A 的值为 true 时，才需计算 B 的值。类似地，计算 A || B，若 A 为 true，则 A || B 必为 true，无需计算 B，只有当 A 的值为 false 时，才需计算 B 的值。例如逻辑表达式 i++>2 ‖ --j<i，当 i=2 与 j=2 时，执行结果为 true（1），且 i=3 与 j=1，而当 i=3 与 j=2 时，执行结果也为 true（1），但此时 i=4 与 j=2。这表明：无需对整个逻辑表达式进行计算，便可得到整个逻辑表达式的值。可以通过控制转移到的位置来体现逻辑表达式的值。

为此引进非终结符号 E 的两个属性：E.true 与 E.false，属性值都是标号，分别表示逻辑表达式 E 的值为 true 与 false 时控制转移到的目标指令位置。这样，对于 a>b 形式的表达式 E，可以解释为：

```
if(a>b) goto E.true;
goto   E.false;
```

概括起来，逻辑表达式的翻译有两种处理方式，第一种方式是计算整个逻辑表达式的值，其值是 true（1）或 false（0），使用该值时将通过与 true（1）的比较，把控制转移到 E.true 或 E.false 处。第二种方式不再计算逻辑表达式的值，而是以控制转移到的位置来体现其值。表达式按第二种处理方式的翻译方案可设计如表 6.14 所示。

表 6.14

重 写 规 则	语 义 动 作
$E::=E_1$ && 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 || gencode(E_1.true, ":") || E_2.code }
$E::=E_1$ || 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 || gencode(E_1.false,":") || E_2.code }
$E::=!E_1$	{ E_1.true:=E.false; E_1.false:=E.true; E.code:=E_1.code; }
$E::=(E_1)$	{ E_1.true:=E.true; E_1.false:=E.false; E.code:=E_1.code; }
$E::=id_1$ relop id_2	{ t:=newtemp; E.code:=gencode("MOV", id_1.place, t) ||gencode("CMP", t, id_2.place) ||gencode("CJ"||relop, E.true) ||gencode("GOTO", E.false) }
E::=true	{ E.code:=gencode("GOTO", E.true) }
E::=false	{ E.code:=gencode("GOTO", E.false) }

表 6.14 的翻译方案中，在涉及逻辑运算的表达式 E 的语义动作所生成的目标代码 E.code 中，

没有显式写出判别表达式的值是否是 true 的目标指令，这是因为在每个运算分量的表达式 E_1 与 E_2 中，包含了判别表达式的值而控制转向 E.true 或 E.false 的目标指令。E.true 与 E.false 在外围表达式或语句中给定。

这些语义动作的设计是明显的。如果画出目标代码结构示意图将更清楚地了解设计的正确性，例如，对于 E::=E && E 的目标代码结构示意图如图 6-14 所示，其中需要引进一个新的标号 E_1.true，标志 E_2.code 的入口处。

现在对于表达式 k>1 && k<10, 可写出相应的目标代码如下：

```
      MOV     k,    t1
      CMP     t1,   #1
      CJ>     L1            /*E1.true=L1*/
      GOTO    E1.false      /*即 E.false*/
L1:   MOV     k,    t2
      CMP     t2,   #10
      CJ<     E2.true       /*即 E.true */
      GOTO    E2.false      /*即 E.false*/
```

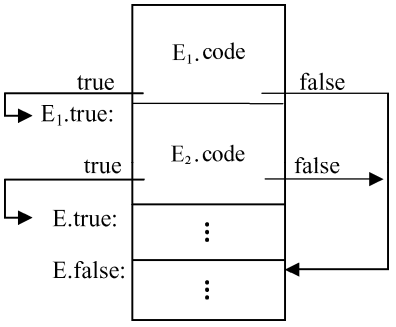


图 6-14

当按第二种处理方式设计翻译方案时，便无需引进操作码 AND、OR 与 NOT。

显然，这样的目标代码结构比第一种处理方式要好。问题是 E.true 与 E.false 的值何时确定？如果在处理完一个 if 语句后才执行相应的语义动作，例如：

```
S::=if(E) S else S { E.true:=newlabel; … }
```

当处理相应于 E 的语义动作时必然还未给定 E.true 与 E.false。因此需要修改前面所给的翻译方案如下：

```
S::= if       { E.true:=newlabel;   E.false:=newlabel;
                S.next:=newlabel                        }
     (E) S1 else S2 { S1.next:=S.next;   S2.next:=S.next;
                      S.code:=…                         }
S::= if       {  E.true:=newlabel;   S.next:=newlabel;
                 E.false:=S.next                        }
     (E) S1   {  S1.next:=S.next;    S.code:=…          }
```

（b）if 语句目标代码中的回填。翻译 if 语句时，在作为条件的表达式 E 后，要生成判别条件为 true 或 false，从而转向真部语句或假部语句的控制转移目标指令。例如，关于 if 语句：

```
if (a<b) min=a;  else  min=b;
```

的目标代码如下：

```
MOV   a,    R0
CMP   R0,   b
CJ<   *+8                   (1)
GOTO  *+16                  (2)
MOV   a,    R1
MOV   R1,   min
GOTO  *+12                  (3)
MOV   b,    R1
MOV   R1,   min
```

当生成(1)处的目标代码时，知道将控制转移到其后第二条目标指令，因此相对地址是*+8。但对于(2)与(3)处的目标指令，在生成时还没有处理其后的内嵌语句，不知道关于它们的目标指令数，并不了解应把控制转向到何处。

为了确定控制转向的去处，一种办法是采用两遍扫描，即在下一遍扫描时再确定并填入控制转移去处，这将带来不必要的麻烦，降低功效；另一种办法是采用标号，用标号标志控制转移的去处，这是把确定控制转移去处的工作交给运行目标程序的系统而回避了这一问题。

这里采用不用再次扫描便能立即生成可执行目标代码的方法，即地址回填。

地址回填的基本思想如下。设置一个转移指令表，当生成控制转移指令而还不知道控制转移到的指令位置时，先仅生成指令坯：如"GOTO　_"，这时，把需确定控制转移到的指令位置的目标指令的序号（或者能标志该目标指令位置的其他信息），记录在指令转移表中。当确定了控制转移到的指令位置时，按转移指令表中的目标指令序号(或其他信息）把所确定的指令位置回填入目标指令中。

为了把所确定的指令位置正确地回填入目标指令，为非终结符号 E 引进属性 truelist 与 falselist，分别用来保存按 E.true 转移和按 E.false 转移的转移指令表的指针。对于非终结符号 S，类似地引进属性 nextlist，用来保存待回填后继语句位置的目标指令序号的转移指令表的指针。

为了建立转移指令表，引进函数 makelist 与 mergelist。函数调用 makelist (i)的功能是建立仅包含目标指令序号 i 的新转移指令表，回送该表的指针。函数调用 mergelist(p1,p2)的功能是把指针 p1 和 p2 所指向的两个转移指令表合并成一个，回送合并后的转移指令表的指针。另外引进回填函数 backpatch。函数调用 backpatch(p, i)的功能是把指令序号 i 所指向的指令地址回填到目标指令中，这些目标指令的序号是在指针 p 所指向的转移指令表中指明的。

至此，可以设计关于 if 语句与逻辑表达式实现回填的翻译方案如下：

```
S::=if(E) M1 S1 N else M2 S2
  { backpatch(E.truelist, M1.pos); backpatch(E.falselist, M2.pos);
    S.nextlist:=mergelist(S1.nextlist, S2.nextlist);
    S.nextlist:=mergelist(S.nextlist, N.nextlist)
  }
N::=ε { N.Nextlist:=makelist(nextpos); emit("GOTO", "_")  }
M::=ε { M.pos:=nextpos }
S::=if(E) M S1
  { backpatch(E.truelist, M.pos);
    S.nextlist:=mergelist(E.falselist, S1.nextlist)  }
E::=E1 && M E2 { backpatch(E1.truelist, M.pos);
                 E.truelist:=E2.truelist;
                 E.falselist:=mergelist(E1.falselist, E2.falselist)  }
E::=E1 || M E2 {  backpatch(E1.falselist, M.pos);
                  E.truelist:= mergelist(E1.truelist, E2.truselist);
                  E.falselist:=E2.falselist)  }
E::=!E1  { E.truelist:=E1.falselist;  E.falselist:=E1.truelist  }
E::=(E1)  { E.truelist:=E1.truelist;  E.falselist:=E1.falselist  }
E::=id1 relop id2 {  E.truelist:=makelist(nextpos+2);
                     E.falselist:=makelist(nextpos+3);
                     t:=newtemp;
                     emit("MOV", id1.place, t);
                     emit("CMP", t, id2.place);
                     emit("CJ"||relop, "_");
                     emit("GOTO", "_")          }
E::=true  { E.truelist:=makelist(nextpos)      }
E::=false  { E.falselist:=makelist(nextpos)    }
```

请注意，该翻译方案中引进了非终结符号 M，引进的目的是为了通过它的属性 pos 记住下一条目标指令的序号。nextpos 产生下一目标指令的序号，初值是 0，每应用 nextpos 一次，它的值

便增加 1。对于非终结符号 N，它正紧跟真部语句 S_1，通过相应的语义动作，生成跳过假部语句的、无条件控制转向 if 语句的后继语句的 GOTO 指令：GOTO S_1.next。由于 S_1.next 还不能确定，它是待回填的，这条 GOTO 指令的序号记录在 N.nextlist 中。

通过上述翻译方案，可以实现对 if 语句，包括逻辑表达式的翻译，其中的控制转移地址都实现了回填。必须注意，现在的目标指令通过函数 emit 生成并输出，涉及的内嵌语句是赋值语句时，其翻译方案需作适当的修改，即也通过 emit 函数生成并输出目标指令。例如，可修改如下：

```
S::=id=E {  p:=lookup(id.name);
            if 未生成关于 E 的目标指令 then
            begin
               t:=newtemp;
               emit("MOV", E.place, t);
               E.place:=t
            end;
            emit("MOV", E.place, p→place);
            S.nextlist:={  }
         }
```

其中，判别是否生成关于 E 的目标指令，把判别 E.code=" "改成了口语叙述“未生成关于 E 的目标指令”，这是因为已无属性 E.code，合理的处理方式是引进属性 E.hascode，当 E.hascode=1 时，表示生成过目标指令，当 E.hascode=0 时，表示未生成过目标指令。为此，关于表达式 E 的翻译部分需作相应的修改，请读者自行完成。下面给出例子说明回填过程。

【例 6.17】 回填的例子。设有 if 语句：

```
if (a>b || c<d && e==f) x=g; else x=h+j;
```

应用上述实现回填的翻译方案。为便于叙述，令 E_1=a>b、E_2=c<d、E_3=e==f、E_4=E_2 && E_3 与 E_5=E_1||E_4。对照翻译方案，E_3 之前的 M 记为 M_3，E_4 之前的 M 记为 M_4。当翻译过程处理到 a>b 时，E_1.truelist={3}，E_1.falselist={4}。关于 E_1 所生成的目标指令是：

```
1: MOV   a,    t1
2: CMP   t1,   b
3: CJ>   _
4: GOTO  _
```

当处理到 c<d 时，又生成关于 E_2 的目标指令如下：

```
5: MOV   c,    t2
6: CMP   t2,   d
7: CJ<   _
8: GOTO  _
```

这时运算符||后相应的 M 的 pos 是 M_4.pos=5，E_2.truelist={7}，E_2.falselist={8}。当处理到 e==f 时，相应于&&之后的 M 是 M_3.pos=9，而 E_3.truelist={11}，E_3.falselist={12}，生成下列目标指令：

```
9:  MOV   e,   t3
10: CMP   t3,   f
11: CJ=   _
12: GOTO  _
```

这时进行对表达式 E_4 的目标指令的回填，把 M_3.pos＝ 9 填入 E_2.truelist 中包含的指令序号相应指令中。即生成目标指令

```
7:  CJ<   (9)
```

并令 E_4.truelist=E_3.truelist，即{11}，把 E_2.falselist 与 E_3.falselist 合并作为 E_4.falselist，因此，E_4.falselist={8,12}。这样，整个逻辑表达式处理完，将执行表达式 E_5 相应的语义动作，按 E_1.falselist 中所包含的指令序号相应的指令进行回填，把 M_4.pos=5 回填入目标指令 4 中：

```
4: GOTO   (5)
```

并把 E_1.truelist 与 E_4.truelist 合并作为 E_5.truelist。因此，E_5.truelist={3, 11}，而 E_5.falselist= E_4.falselist，因此，E_5.falselist={8,12}。

当处理真部语句 x=g 时，M_1.pos=13，并由赋值语句相应的语义动作生成目标指令如下：

```
13: MOV  g,    t4
14: MOV  t4,   x
```

N 的相应语义动作的执行结果生成新的转移指令表 N.nextlist={15}，并生成目标指令:

```
15: GOTO  _
```

处理假部语句 x=h+j 时，由非终结符号 M 相应的语义动作，置 M_2.pos=16，并由赋值语句相应的语义动作生成下列目标指令：

```
16: MOV   h,    t5
17: ADD   j,    t5
18: MOV   t5,   x
```

最后由 if 语句相应的语义动作完成按 E.truelist 与 E.falselist 的回填。把 M_1.pos=13 回填入序号为 3 与 11 的目标指令中。

```
3:  CJ>   (13)
11: CJ=   (13)
```

把 M_2.pos=16 回填入序号为 8 与 12 的目标指令中：

```
8:  GOTO  (16)
12: GOTO  (16)
```

然后执行：S.nextlist:=mergelist(S_1.nextlist, S_2.nextlist);

S.nextlist:=mergelist(S.nextlist, N.nextlist);

使 S.nextlist={15}。

综上所述，最终生成的目标代码如下：

```
1: MOV   a,      t1
2: CMP   t1,     b
3: CJ>   (13)
4: GOTO  (5)
5: MOV   c,      t2
6: CMP   t2,     d
7: CJ<   (9)
8: GOTO  (16)
9: MOV   e,      t3
10: CMP  t3,     f
11: CJ=  (13)
12: GOTO (16)
13: MOV  g,      t4
14: MOV  t4,     x
15: GOTO _
```

```
16: MOV   h,     t5
17: ADD   j,     t5
18: MOV   t5,    x
```

读者必定发现序号为 15 的目标指令中尚未回填入指令的地址，这是因为该指令序号 15 已保存在 S.nextlist 指明的转移指令表中，将由该 if 语句的外围语句实现按 S.nextlist 进行回填。一般来说，应填入序号 19。可以对该翻译方案稍作修改，以立即完成回填。请读者自行思考。提示：引进重写规则 P::=S。

为加深理解，给出上述 if 语句和逻辑表达式的注释分析树如图 6-15 所示。

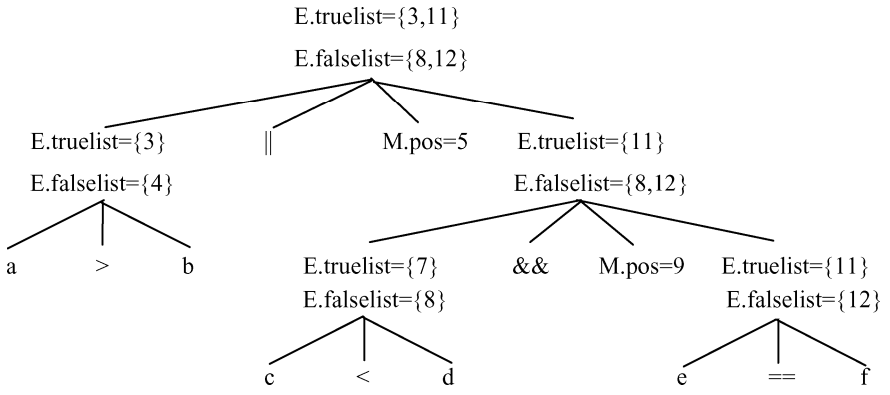


图 6-15

（2）switch 语句的翻译

① switch 语句的语法定义形式。C 语言中 switch 语句的一般书写形式如下：

```
switch(E)
{  case C1: S1
   case C2: S2

   case Cn-1: Sn-1
   default: Sn
}
```

其中，C_j(j=1,2,…,n−1)可以是一般的常量表达式，这里仅考虑常量的情况，称为情况常量。

② switch 语句的语义描述。switch 语句以口语形式可描述如下：根据表达式 E 的值确定执行其中的某个语句 S_j(j=1,2,…,n−1)。确切地说，如果 E 的值等于某个情况常量值 C_j，则执行相应的语句 S_j(j=1,2,…,n−1)。如果没有一个值相等，则执行与缺省值 default 相应的语句 S_n，然后执行 switch 语句的后继语句。

switch 语句与 if 语句的区别是：它是一种多路分支结构。一般来说，它并不是与 C_1、C_2、…、C_{n-1} 逐个地比较，从而确定与 C_j 相应的语句 S_j，即它并不等价于多重嵌套 if 语句。应是给定一个 E 的值，便能“开关”到相应的语句 S_j，示意图如图 6-16（a）所示，而非图 6-16（b）所示。

C 语言 switch 语句的语义规定，按 E 的值确定情况常量 C_j，执行相应的语句 S_j 后，将继续执行下去，直到执行到 break 语句或 S_n 之后，才执行完毕该 switch 语句。

③ switch 语句的执行。根据 C 语言 switch 语句的语义，switch 语句的执行步骤如下：

步骤 1　计算表达式 E 的值；

步骤 2　在 switch 语句内寻找和 E 的值相匹配的情况常量值。如果找不到，则 default 和 E 的值相匹配；

步骤 3　执行和该匹配值相应的语句。

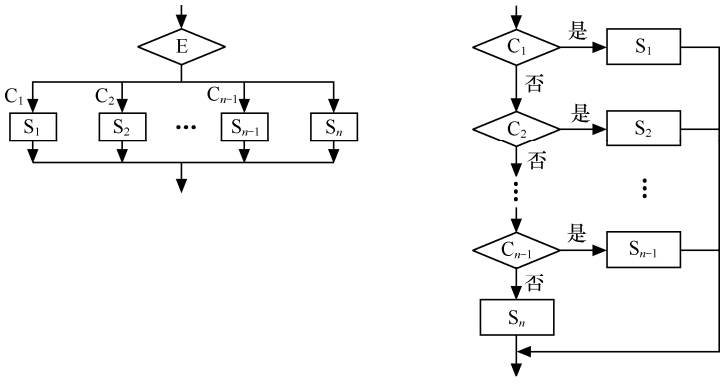


图 6-16

④ switch 语句目标代码结构设计。根据 switch 语句的语义，switch 语句执行时，并不是把 E 的值与 C_1、C_2、…逐个地比较，以确定相应的语句 S。在一般情况下，可通过构造散列表或情况地址表，从而能够从表达式 E 的值，直接取到某一内嵌语句 S_j(j=1,2,…,n−1)的首地址，而把控制立即转移到相应语句的目标代码处。

这里给出一种简单而易于实现的方案。假定情况常量的值依次对应于 1 、2 、…、n−1，default 对应于 0，引进一个称为情况地址表的指针数组 A，A[k]的值正是值 k 所对应情况常量相应语句的目标代码的入口地址（或第一条目标指令序号）。因此将执行语句 goto　A[k]，从而控制转移去执行语句 S_k。

可有如下的目标代码结构：

```
        计算 E 的值并存于 k 的目标代码
        goto  A[k]
A[1]:   语句 S1 的目标代码
        转向后继语句入口处 next 的目标代码  /*C 语言时不生成*/
A[2]:   语句 S2 的目标代码
        转向后继语句入口处 next 的目标代码  /*C 语言时不生成*/
         ⋮
A[n-1]:语句 Sn-1 的目标代码
        转向后继语句入口处 next 的目标代码  /*C 语言时不生成*/
A[0]:   语句 Sn 的目标代码
next:   switch 语句后继语句的目标代码
```

请注意 switch 语句与 if 语句的区别，特别是注意 C 语言 switch 语句的语义与 PASCAL 语言等的不同。

⑤ switch 语句翻译方案的设计。请读者自行完成 switch 语句翻译方案的设计。只要比较 switch 语句的语法规则以及为其设计的目标代码结构两者之间的对应关系，不难写出相应的翻译方案，要点是如何使情况常量值依次对应于 1、2、…、n−1，以及如何建立情况地址表，可以考虑简化的情况。

3. 迭代结构的翻译

迭代结构是一种重复执行同一个程序段的控制结构。C 语言中迭代结构有 3 类，即 while 语句、do-while 语句与 for 语句。这里着重讨论 while 语句的翻译。

（1）while 语句的语法定义形式。C 语言中 while 语句的一般书写形式如下：

```
while (E) S
```

其中 E 是表达式，一般是逻辑型的，S 是单个内嵌语句，是循坏体。

（2）while 语句的语义描述

while 语句的语义按其书写，可以很直观地以口语形式描述如下：当逻辑表达式 E 的值为 true（非零）时，重复执行内嵌语句 S，直到 E 的值为 false（零）时，不再重复执行内嵌语句而执行 while 语句的后继语句。

（3）while 语句的执行

根据 while 语句的语义，可以很方便地确定其执行步骤如下：

步骤 1　计算表达式 E 的值；

步骤 2　判别 E 的值，如果是 true，执行循环体语句 S，然后控制返回步骤 1；

步骤 3　当 E 的值是 false 时，不再执行循环体语句 S，而执行 while 语句的后继语句。

（4）while 语句目标代码结构设计

为了确定 while 语句的目标代码结构，可以先基于上述执行步骤，画出目标代码结构示意图如图 6-17 所示，这样可以更直观地理解。其中引进了 3 个标号：begin、E.true 与 next，分别标志 while 语句的入口位置、循环体目标代码的起始位置与后继语句的起始位置。相对照，while 语句的目标代码结构可设计如下：

```
begin: 计算表达式 E 的值的目标代码
       当 E 之值为 true 时控制转向循环体的目标代码
       当 E 之值为 false 时控制转向后继语句的目标代码
       循环体语句 S 的目标代码
       转向标号 begin 处的目标代码
next:  后继语句的目标代码
```

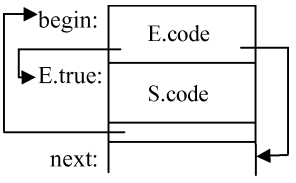


图 6-17

【例 6.18】　while 语句目标代码的例子。设有 while 语句：

```
S=0; j=1;
while (j<10)
{ S=S+j; j=j+1; }
```

将有如下的目标代码：

```
          MOV   #0,     S
          MOV   #1,     j
begin:    MOV   j,      t1
          CMP   t1,     #10
          CJ<   E.true
          GOTO  next
E.true:   MOV   S,      t2
          ADD   j,      t2
          MOV   t2,     S
          MOV   j,      t3
          ADD   #1,     t3
          MOV   t3,     j
          GOTO  begin
next:
```

（5）while 语句翻译方案的设计

从上述目标代码结构设计中可见，需引进 3 个标号，即 begin、E.true 与 next。它们作为相应非终结符号的属性，且类似地通过 newlabel 来生成。可以设计 while 语句的翻译方案如下：

```
S::=while(E) S1 {  S.begin:=newlabel;
                   E.true:=newlabel;  S.next:=newlabel;
                   E.false:=S.next;   S1.next:=S.begin;
```

```
S.code:=gencode(S.begin,":")||E.code
       ||gencode(E.true,":")||S1.code
       ||gencode("GOTO",S1.next)
       ||gencode(S.next,":")          }
```

表达式 E 是一个逻辑表达式，关于它的翻译方案，请参看前面关于 if 语句的翻译。要提醒的是，关于 E 的目标代码中包含了控制转向 E.true 与 E.false 的部分，显然 E.false 的值就是 S.next。

【例 6.19】　while 语句目标代码的例子。设有 while 语句：

```
k=0;
while(k>0)
{ t=k;  k=k+1; }
```

应用上述翻译方案可以有如下的目标代码：

```
   MOV   #0,   k
L1:MOV   k,    t1
   CMP   t1,   #0
   CJ>   L2
   GOTO  L3
L2:MOV    k,   t2
   MOV   t2,   t
   MOV   k,    t3
   ADD   #1,   t3
   MOV   t3,   k
   GOTO  L1
L3:
```

其中的临时变量 t1、t2 与 t3 一般将对应于寄存器。

由于 while 语句中涉及条件控制转移，类似于 if 语句，同样应考虑逻辑表达式的翻译与回填两个问题。逻辑表达式的翻译沿用前面的讨论，关于回填，可修改成如下的实现回填的翻译方案：

```
S::=while M1 (E) M2 S1
{  backpatch (S1.nextlist, M1.pos);
   backpatch (E.truelist, M2.pos);
   S.nextlist:=E.falselist;
   emit("GOTO", M1.pos)        }
```

引进 M 的作用是明显的，即用来记录目标指令的位置，有

```
M::=ε { M.pos:=nextpos }
```

while 语句的内嵌循环体语句从语法书写上看，只能是单个语句，然而从例 6.19 看到，循环体中往往包含多个语句，为此把循环体中的所有语句用大括号对括住，从而成为一个复合语句。当然，复合语句中可以包含任意的控制语句，也就是说，又可能是 while 语句。从上述翻译方案来看，语义动作中仅对内嵌语句 S 与表达式 E 的目标代码进行了回填，并没有对 while 语句的回填，必须在外围语句中应用所生成的 S.nextlist 来对 while 语句回填。为了对复合语句中的 while 语句以及 if 语句等的目标代码进行回填，设计关于复合语句实现回填的翻译方案如表 6.15 所示。

表 6.15

重 写 规 则	语 义 动 作
S::={ L }	{S.nextlist:=L.nextlist }
L::= L1; N 　　S	{backpatch(L1.nextlist, N.pos) } {L.nextlist:=S.nextlist }
L::=S	{L.nextlist:=S.nextlist }
N::=ε	{N.pos:=nextpos }

请读者自行以实例检查该翻译方案的正确性。

关于 do-while 语句的翻译，参照关于 while 语句的讨论，不难设计翻译方案，请读者自行进行。至于 for 语句的翻译，这里不拟讨论，读者可以关于 for 语句的简化情况进行讨论。例如，假定 for 语句形如：for(v=E_1; V<=E_2; V=V+1) S, 考虑其规范展式：

```
        v=E1;
loop:   if(v>E2) goto FINISH;
        S
        v=v+1;
        goto loop;
FINISH:
```

不难为其设计相应的翻译方案。

4. 函数调用语句的翻译

（1）函数调用与活动记录

函数是高级程序设计语言中必不可少的语言成分，它通常完成相对独立而完整的功能，使得语言的描述能力大大加强。函数设施使得能在程序的不同位置上，以不同的参数执行同一个程序段（函数体）。在 C 语言中通过函数定义来定义用户自定义的函数，其中的函数体给出了该函数的实现细节。控制部分中通过函数调用去执行相应的函数体中的控制部分。函数定义在说明部分的翻译中讨论，此处着重讨论函数调用语句的翻译。

【例 6.20】 函数定义与函数调用的例子。设有自定义函数 max2，其函数定义如下：

```
int max2 (int x, int y)
{  int t;
   if(x>y) t=x; else t=y;
   return  t;
}
```

今求 a、b 与 c 中的最大值，置于 max 中，则有下列语句：

```
max=max2 (a, b);
max=max2 (max, c);
```

在函数调用时，明显的工作包括下列几方面：

- 建立函数调用时的实际参数与函数定义中的形式参数之间的对应关系；
- 把控制转向函数体入口地址处，运行函数体中的控制部分；
- 回送函数值；
- 把控制返回到函数调用处，继续运行。

因此对于 max2(a, b)，首先建立实际参数与形式参数之间的对应关系，即把 a 作为 x 的值，把 b 作为 y 的值，然后控制转入 max2 的函数体，执行 if 语句，此后，回送 t 的值，返回到调用处后赋值给 max。调用程序与被调用程序之间包括了两个方面的工作，即数据联系和控制联系。正确的数据联系包括：寄存器在调用前后不被破坏、实际参数正确地传递给形式参数、为被调用函数局部量分配存储区域、被调用函数的非局部量能被正确地引用、当是有值函数时正确回送函数值，等等。为了保证正确地控制联系又方便处理，对程序运行期间的控制流作如下假定：

- 程序的执行由一些连续的步骤组成，在任何一个执行步骤时，控制总是处于正运行程序中某处，这表示控制流是连贯的；
- 每次函数的调用总是从函数体目标代码入口地址处开始，当被调用函数执行结束时，返回到紧随函数调用处的位置。

由于 C 语言允许函数递归定义，例如：

```
int fac( int n)
{ int t;
```

```
  if (n==0)
    t=1;
  else
    t=n*fac (n-1);
  return  t;
}
```

在以参数 n 执行函数体时，又将以参数 n−1 调用 fac 本身，调用过程如图 6-18 所示。

当以参数 n 调用 fac 时，将为形参 n 与局部量 t 分配存储区域，譬如说，存储地址分别为 N 与 T。如果此 N 与 T 是固定的，当其函数体内以参数 n−1 调用 fac 时，以参数 n 调用 fac 的执行还未结束，又把 n−1 的值传送入 N 中，而不能保留 n 值，导致不能正确地实现 fac 的功能。这表明，每次调用函数，需要为被调用函数分配新的存储区域。通常引进称为活动记录的概念。

活动记录是为管理函数的一次执行（活动）中所需信息而设置的连续存储区域。一个函数被调用时，涉及实际参数、局部量、非局部量以及需保护的寄存器等。如图 6-19 所示，通常活动记录由 7 个域组成。

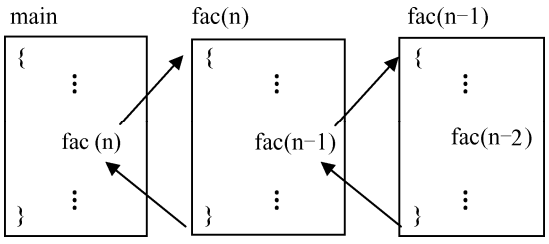


图 6-18

图 6-19

- 返回值域存放被调用（有值）函数回送给调用函数的函数值；
- 实际参数域用于存放调用函数提供给被调用函数的实际参数的值；
- 控制链域的值是一个指针，用来指向调用函数的活动记录，从而可以取得调用函数的相关信息；
- 访问链域的值是一个指针，用于引用存放于其他活动记录的非局部数据；
- 机器状态域用来保存临进入被调用函数时的机器状态信息，包括程序计数器的值和控制从这个被调用函数返回时必须恢复的一切寄存器的值；
- 局部数据域用来存放函数当前的一次执行时的局部数据，如 fac 函数的局部量 t；
- 临时数据域用来存放函数执行时产生的中间结果数据，例如计算表达式时产生的中间结果值。

C 语言等程序设计语言的通常做法是设置一个运行栈，在函数被调用时，为该被调用函数下推入一个活动记录。当被调用函数执行结束而把控制返回到调用函数时，把该活动记录从运行栈中上退去。

通常设指针变量 tops 指向运行栈的栈顶，即 tops 指向下一个活动记录的开始位置。为了方便被调用函数访问自己的局部数据域与临时数据域，通常另设一个指针 top_sp，指向活动记录中机器状态域的末端位置，这样，函数中的局部变量相对于局部数据域首地址的位移量就都是正整值。top_sp 的值存放在活动记录的访问链域中。

活动记录中每个域的大小随被调用函数的不同而不同，然而对于 C 语言等不允许大小可变的实际参数的语言，在编译时刻已可确定几乎所有域的大小。

综上所述，为了实现调用时的衔接，调用函数与被调用函数分别完成下列事项。

调用程序实现调用前需完成的事项：

- 计算实际参数的值；
- 在运行栈中下推入被调用函数的活动记录，把返回地址和自己的活动记录的指针存入被调用函数的活动记录相应域中。

被调用函数完成下列事项：

- 把寄存器的值和其他状态信息保存在机器状态域中；
- 对局部数据初始化，提醒注意的是：C 语言编译程序仅对静态的局部数据才进行初始化，因此，C 语言程序中的 Auto 局部变量在运行开始时刻无定义(无值)；
- 执行函数体。

当从被调用函数返回到调用函数时，将完成下列事项：

- 被调用函数把返回值存放入活动记录中；
- 被调用函数使用活动记录中相应域的信息恢复 top_sp 和其他寄存器，并按所保存的返回地址把控制转移到调用函数；
- 调用函数把返回值复制入自己的活动记录相应位置中。

基于上述分析，设计函数调用的目标代码结构及翻译方案。

（2）函数调用语句翻译方案的设计

① 函数调用语句的语法定义形式。对于 C 语言，函数调用有两种情况，即一种情况是作为表达式的一个组成部分出现在程序中，例如，sin(x)与 max2(a, b)等；另一种情况是作为语句出现在程序中，例如，swap(&x, &y)与 printf("max=%d\n", max)。为简化讨论，这里着重考虑函数调用作为语句的情况。

C 语言函数调用语句的一般书写规则如下：

```
id(Elist); 或 id( );
```

分别对应于有参数与无参数的情形，其中 id 是函数名，Elist 是实际参数表。为便于识别出函数调用语句，令函数调用语句的一般书写规则如下：

```
CALL id (Elist);
```

② 函数调用语句的语义描述。函数调用语句的语义可以用口语描述如下：建立实际参数与形式参数之间的对应关系，然后执行被调用函数的函数体，该函数体执行结束后返回到调用处。其中建立形实参数对应关系，就是调用函数计算各个实际参数值，把它们传送入被调用函数的活动记录的相应域中。

③ 函数调用语句的执行。联系前面对函数调用语句的讨论及函数调用语句的语义，一个函数调用语句的执行步骤大致如下：

步骤 1　在运行栈中下推入被调用函数的活动记录。

步骤 2　计算函数调用语句参数表中各个实际参数的值，并把它们传送入活动记录相应域的存储位置中（实际参数域）。

步骤 3　设置环境指针，使被调用函数能访问非局部数据（访问链域）。

步骤 4　保护临调用时的机器状态，包括寄存器值与返回地址（机器状态域）。

步骤 5　对局部数据初始化（局部数据域）。注意，C 语言仅对静态的局部数据才进行初始化，不对非静态的局部数据进行初始化。

步骤 6　把控制转移到被调用函数的入口处，即执行被调用函数的函数体。

概括起来，函数调用时，先设置好活动记录，然后去执行被调用函数。

④ 函数调用语句目标代码结构设计。从上述可见，为调用一个函数，需要做大量的工作，生成相当数量的目标指令，且各种被调用函数有各自的活动记录结构，特别是很难用简洁的形式来表达此目标代码结构，因此通常用一种简单的方式来实现，即通过被调用函数的函数体的入口子程序，由它实现调用函数时的各项工作；相应地被调用函数的函数体还有一个出口子程序，由它完成从被调用函数返回到调用函数时所需完成的各项工作。函数调用语句的目标代码可简单设计如下。

```
计算实际参数 P1 的目标代码
PARAM   P1
计算实际参数 P2 的目标代码
PARAM   P2
⋮
计算实际参数 Pm 的目标代码
PARAM   Pm
CALL    P,    m
```

其中 P 是函数名，m 是参数个数，参数分别是 P_1、P_2、…、P_m，符号 PARAM 指明其后是实际参数。

⑤ 函数调用语句翻译方案的设计。根据上述目标代码的设计，不难给出函数调用语句的翻译方案如下：

```
S::=CALL id(Elist)
    { S.code:=Elist.code || gencode("CALL", id.place, Elist.number) }
Elist::=Elist1, E
    { Elist.code:=E.code || Elist1.code || gencode("PARAM", E.place);
      Elist.number:= Elist.number +1                                  }
Elist::=E
    { Elist.code:=E.code || gencode("PARAM", E.place);
      Elist.number:=1                                                 }
```

其中，id.place 是函数 id 的入口地址。

【例 6.21】　设有函数调用语句。

```
printf ("a+b=%d, c*d=%d, e-f=%d\n", a+b, c*d, e-f);
```

按上述翻译方案将生成下列目标代码：

```
MOV   e,     t3
SUB   f,     t3
MOV   c,     t2
MPY   d,     t2
MOV   a,     t1
ADD   b,     t1
PARAM      "a+b=%d, c*d=%d, e-f=%d\n"
PARAM      t1
PARAM      t2
PARAM      t3
CALL       printf,  4
```

按照上述翻译方案所生成的目标代码中，参数计算部分将集中在一起，而参数（PARAM）语句又集中在一起。

请注意，关于 Elist 的重写规则的语义动作中目标代码 Elist.code 的生成，E.code 在前，Elist.code 在后，这样便使得计算实际参数的目标代码的顺序正好与书写顺序相反。这是因为 C 语言规定函数参数的计算，按从右到左的顺序进行。假定按书写顺序计算实际参数，只需改为：

```
Elist.code:=Elist1.code || E.code || gencode ("PARAM", E.place)
```

对于有值函数的调用，它与无值函数的调用区别在于：作为表达式的一个组成部分，且有函

数值返回，为保存并回送函数的返回值，可以在目标指令中增加一个参数，如

```
CALL  P,  m,  t
```

其中，t 的值为函数返回值。这时可以在生成目标指令的语句中改为：

```
gencode ("CALL", id.place, Elist.number, id.value)
```

其中 id.value 是存放函数返回值的临时变量，可由下列赋值语句赋值：

```
id.value:=newtemp;
```

如果是有值函数调用，将作为表达式的一个组成部分，对于它的翻译方案应如何设计？请读者自行考虑。显然应引进重写规则：

```
F::=id(Elist)
```

⑥ 与函数调用语句翻译相关的问题。与函数调用语句翻译相关的问题大致是如下几个：参数的传递、非局部数据的存取与运行时刻支持程序包。

（a）参数的传递。函数调用时必须建立形实参数的对应，也就是说，把实际参数的值传递给相应的形式参数。这种对应有几种方式，即值调用方式与地址调用方式，还能是换名调用方式。

所谓值调用方式，也就是实际参数的值与形式参数建立对应关系，而地址调用方式则是实际参数的存储地址与形式参数相联系。对于换名调用方式的实际参数，它是一个名字（字符串），以它替换形式参数，函数体内出现的形式参数都被替换为那个名字（字符串）。这在早期的程序设计语言中出现过。

值调用是最简单的参数传递方式。如果一个形式参数被指明是值调用的，则在函数调用时，把实际参数的值，传递入被调用函数的活动记录中相应形式参数的存储区域中，引用形式参数便是引用实际参数的值，即使形式参数在函数体执行过程中值被改变了，相应的实际参数也不会被改变值。最典型的例子是函数 swap。假定 swap 定义如下：

```
void swap( int a, int b)
{ int t;
  t=a;  a=b;   b=t;
}
```

设有函数调用语句

```
swap(i, j);
```

尽管在 swap 的函数体内交换了形式参数 a 与 b 的值，但在返回后，相应实际参数 i 与 j 的值并未被交换。这就是因为形式参数 a 与 b 都是值调用方式的参数，在建立形实参数对应关系时，仅分别得到实际参数 i 与 j 的值。

如果是地址调用方式，则把实际参数的存储地址，传递入被调用函数的活动记录中相应形式参数的存储区域中，因此，形式参数将得到实际参数的地址，通过该地址，可以改变相应存储区域的内容，从而改变调用程序中一些量的值。例如把上述 swap 函数的定义修改如下：

```
void swap( int  *a,  int  *b)
{ int t;
  t=*a;  *a=*b;  *b=t;
}
```

现在的调用语句是：

```
swap( u, v);
```

而 u 与 v 如下地定义：

```
int i=3, j=5;   int *u=&i, *v=&j;
```

使 u 与 v 的值分别是 i 与 j 的存储地址，则对 swap 的上述调用结果将改变 i 与 j 的值。但由于作为参数的 u 与 v，其值是不能被改变的存储地址，即实际参数本身并不能改变，因此，通常认为

C 语言仅值调用方式一种。

值调用方式的形实参数对应，可通过如下方式实现：

- 把值调用方式的形式参数作为函数的局部变量，在所属函数的活动记录的相应域中为其分配存储区域；
- 把实际参数的值传送入该值调用参数的相应活动记录存储区域中。

在实际参数的值是一个存储地址时，实际上实现了地址调用参数。

（b）非局部数据的存取。非局部变量是调用函数与被调用函数之间交换信息的另一种常用手段，这时需要利用活动记录中的访问链。访问链中给出了调用函数的活动记录的 top_sp 指针，通过该指针可以取接调用函数中的局部量（被调用函数的非局部量）。但如果要对调用函数的调用函数中的，甚至更深层次的调用函数中的变量进行存取，必须逐层取得调用函数的调用函数的活动记录中的访问链，这样的效率是十分低下的。为了提高功效，可以引进如下形式的二元组：

$$(n_p - n_q, \text{a 在相应活动记录中的位移量})$$

它表示调用深度为 n_p 的函数 p 中，所引用调用深度为 n_q 的函数中的变量 a 的地址，把它保存于符号表中，这种二元组在编译时刻设定。

可以更快地访问非局部量的方法是使用显示表。显示表 d 是一个活动记录指针数组，调用深度为 k 的非局部量 a，在显示表元素 d[k]所指明的活动记录中。显示表随着函数的活动而变化，其最大元素个数决定了函数的最大调用深度。由于无需顺着访问链追踪，可以直接从显示表元素取得所需的访问链，因此，使用显示表是更快的方式。

（c）运行时刻支持程序包。由于函数调用的频繁，函数调用的实现对程序运行的功效有重大影响。如前所述，函数的调用是通过函数入口子程序与出口子程序来实现，这些子程序关系到函数参数的传递、函数的调用与函数的返回。这些子程序作为目标代码运行的支持，是运行时刻支持程序包不可缺少的一个组成部分。要特别强调的是，没有运行时刻支持程序包的支持，目标代码是不可能运行的。C 语言编译程序的运行时刻支持程序包中的运行子程序，通常以库函数形式存放在扩展名为.lib 的文件中。

输入输出语句是任何程序设计语言的重要而必不可少的组成部分。输入输出语句的实现与目标代码运行的功效紧密相关。但通常输入输出的实现有一定的难度，因为输入输出与计算机硬件细节紧密相关。一般来说，对应于一个输入输出语句的目标代码是一列冗长的目标机目标指令。对于 C 语言来说情况一样。考虑到 C 语言中的输入输出语句同样是一种特殊的函数调用语句，为了简单起见，可以不单独考虑输入输出语句的目标代码生成问题。仅给出虚拟机目标指令操作码，如 print 与 read 等。更简单的方式是，保持 C 语言中的符号，如 printf 与 scanf 等，由系统子程序来实现这些操作码的功能。

6.5　翻译方案的实现

综合前面的讨论，了解了基本的赋值语句及各类控制语句，包括各类选择结构、迭代结构与函数调用语句的翻译，包括翻译方案的设计及相关问题的讨论。确实，属性文法，尤其是翻译方案，是一种描述语义的好的表示手段。

然而读者必然有一个疑问，如何去实现翻译方案？

语法分析时，重写规则描述了语言结构的书写规则，通过识别程序，实现对源程序（中间表

示）进行语法分析。但是，翻译方案中的语义动作相当于用一种高级程序设计语言书写的程序，它并不能直接运行，因为它们也只是符号串，本质上与 C 语言程序一样。进一步说，这些语义动作或语义规则也必须与高级程序设计语言源程序一样，经历词法分析、语法分析和语义分析等，然后再生成目标程序。这样将需要一个更为复杂的翻译系统，显然行不通。

简单而可行的办法是：为每个语义动作或语义规则，设计相应的语义子程序，在生成语法分析树的同时，或基于语法分析树与依赖图，通过调用这些语义子程序进行属性值的计算，最终实现翻译。

本节着重以赋值语句目标代码生成为例，说明语义子程序的实现思路。

6.5.1 实现要点

为突出重点，简化赋值语句的语法规则。设有文法 G6.9[A]：

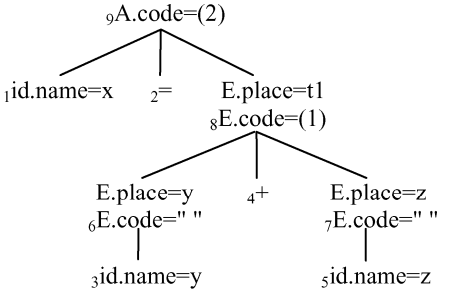
```
A::=id=E    E::=E+E|E*E|-E|(E)|id
```

假定经过语法分析与语义分析，书写上无错，且每个标识符 id 都有定义，可在符号表中查到，同时目标指令中用标识符名替代符号表条目指针，不引进函数 lookup，可写出如下的翻译方案：

```
A::=id=E {  if E.code=" " then
            begin
               t:=newtemp;
               E.code:=gencode ("MOV", E.place, t);
               E.place:=t
            end;
            A.code:=E.code || gencode ("MOV", E.place, id.name)
         }
E::=E1+E2 {  E.place:=newtemp;
             E.code:=E1.code || E2.code
                   || gencode ("MOV", E1.place, E.place)
                   || gencode ("ADD", E2.place, E.place)    }
E::=E1*E2 {  E.place:=newtemp;
             E.code:=E1.code || E2.code
                   || gencode ("MOV", E1.place, E.place)
                   || gencode ("MPY", E2.place, E.place)    }
E::=-E1 {  E.place:=newtemp;
           E.code:=E1.code
                 || gencode ("MOV", E1.place, E.place)
                 || gencode ("NEG", E.place)                }
E::=(E1) { E.place:=E1.place;  E.code:=E1.code               }
E::=id  { E.place:=id.name;  E.code:=" "     }
```

假定输入符号串是 x=y+z，相应的自底向上 LR 分析技术生成的注释分析树如图 6-20 所示。该赋值语句的目标代码如下：

```
MOV  y,   t1
ADD  z,   t1
MOV  t1,  x
```



其中，(1)= "MOV y, t1" || "ADD z, t1"
(2)= (1) || "MOV t1, x"

图 6-20

现在通过对照翻译方案与注释分析树来概括实现要点。考虑下面几个方面：属性值的存取、属性值的计算与目标代码的存储。

1. 属性值的存取

首先考虑图 6-20 中的注释分析树。注释分析树是一种语法分析树，在其结点上给出

了属性值。从图中可见，同一个非终结符号可以有多个属性，同时，同一个属性可以出现在多个结点上。例如，非终结符号 E 有属性 place 与 code，分别存放存储地址与目标代码，而注释分析树上结点 6、7 与 8 都有 E.place 与 E.code 两个属性。因此设置属性信息表，并扩展语法分析树结点的数据结构，把属性在属性信息表中的序号存放在语法分析树相应文法符号的结点中，通过该序号存取属性。

属性信息表用来登录各个文法符号的属性，包括文法符号名、属性名、属性类型与值，以及依赖属性等，还指明该属性所属结点的序号。用 C 语言可设计数据结构如下：

```
struct  依赖属性类型
{  int   属性序号;
   struct 依赖属性类型  *下一依赖属性;
};
```

其中属性序号是属性在属性信息表中条目的序号，通过序号来存取属性信息表条目。

```
struct 属性表条目类型
{  int  属性序号;   int 文法符号序号;
    char 属性名[MaxNameLength];
    char 属性值种类; /*可以是'I','F','S','M'等*/
    union 属性值联合类型  属性值;
    struct 依赖属性类型  *依赖属性链;
    int 结点序号;
};
```

其中属性值种类根据属性的具体情况确定，例如，整数值是‘I’，存储地址是‘S’（字符串），代码则为‘M’。因为属性值有各种不同种类，所以一般情况下，须引进属性值联合类型。如果属性值仅一种类型，便无需引进。

```
struct
{ struct 属性表条目类型  属性表条目[MaxAttributeNum];
  int   属性表条目数;
} 属性信息表;
```

为了今后程序书写简洁起见，可把属性信息表设计为一个数组：

```
struct 属性表条目类型  属性信息表[MaxAttributeNum];
```

另设一个属性信息表条目计数器变量。

注释分析树的数据结构可设计如下：

```
struct 属性结点类型
{ int  属性序号;
   struct 属性结点类型  *下一属性结点;
} ;
struct 注释树结点类型
{ int 结点序号;    int 文法符号序号;
   int 父结点序号; int 左兄结点序号; int  右子结点序号;
   struct 属性结点类型 *属性信息链;
} ;
struct
{ struct 注释树结点类型 注释树结点[MaxNodeNum];
   int   注释树结点数;
} 注释分析树;
```

同样为了今后程序书写简单，把注释分析树设计为一个数组：

```
struct 注释树结点类型  注释分析树[MaxNodeNum];
```

另外引进一个整型变量，记录注释分析树中结点个数。

当把属性序号保存在某结点中时，可通过此序号来引用相应属性值。

【例 6.22】 关于文法 G6.9[A]及其相应的翻译方案，可设计属性信息表的数据结构如下：

```
struct DependAttributeT    /*依赖属性类型*/
{  int  AttributeNo;
   struct DependAttributeT  *NextDependAttri;
};
struct  AttributeItemT     /*属性表条目类型*/
{   int AttributeNo;       int  SymbolNo;
    char  AttributeName[8];
    char  AttributeKind;    /*存储地址'S'和目标代码'M'等*/
    union  AttributeType  AttributeValue;
    struct  DependAttributeT  *DependAttriLink;
    int NodeNo;
};
struct  AttributeItemT  AttributeInfoTable[MaxItemNum];/*属性信息表*/
int NumOfAttribute;
```

其中 union 类型的 AttributeType 的数据结构可设计如下：

```
union  AttributeType
{  char  Addr[10];     /*存储地址名，即字符串*/
   struct 目标指令类型  *目标指令链;
};
```

关于目标指令类型的设计，见后面的讨论。

2. 属性值的计算

计算一个属性值，可以通过执行语义子程序来实现。编写语义子程序时通常要考虑 3 个方面，即取到依赖属性值并建立依赖属性链、计算属性值并在属性信息表中建立相应条目，以及把属性值保存到注释分析树中。

这里以重写规则 E::=E+E 及相应的语义动作：

```
E.place:=newtemp;
E.code:=E1.code || E2.code
       || gencode ("MOV", E1.place, E.place)
       || gencode ("ADD", E2.place, E.place)
```

为例来说明。

在计算 E 的属性 code，即 E.code 时，它将依赖于 4 个属性：E_1.code、E_2.code、E_1.place 与 E_2.place，因此需取到这 4 个属性。先看如何取到 E_1.code。思路如下：首先确定该文法符号 E（即 E_1）在注释分析树中相应结点的序号 NE1，然后从结点序号 NE1 及属性名“code”查到属性信息表条目，最后便取得相应属性 E_1.code 的值。

结点序号 NE1 如何得到？这需要结合语法分析技术。因为是按自底向上的 LR 分析技术，在执行语义动作时已达到重写规则 E::=E+E 右部的右端，而进行归约时，显然这时分析栈示意图如图 6-21 所示。栈顶指针 tops 指向的是 E（即 E_2），tops-2 指向的 E 是 E_1，因此有赋值语句：

```
NE1=分析栈[tops-2].结点符号;
```

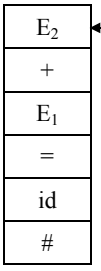


图 6-21

属性 E_1.code 如何得到？现在有了结点序号 NE1 与属性名 code，可以引进一个函数 search，由它在属性信息表中查到属性 E_1.code。函数 search 的原型如下：

```
int search(int 结点序号,char 属性名[ ]);
```

函数调用 search(NE1, "code")执行后将返回 E_1.code 在属性信息表中的序号，存于 NE1code 中，因此有赋值语句：

```
NE1code=search (NE1, "code");
```

函数 search 的实现是容易的，请读者自行完成。

属性信息表[NE1code].属性值便是 E_1.code 的值。

作为例子，再考虑 E_2.place 的取得：

```
NE2=分析栈[tops].结点序号;
NE2place=search (NE2, "place");
```

属性信息表[NE2place].属性值便是 E_2.place 的值。

假定已取得 NE1code、NE2code、NE1place 与 NE2place 之值，执行下列赋值语句：

```
E.code:=E1.code || E2.code
       || gencode ("MOV", E1.place, E.place)
       || gencode ("ADD", E2.place, E.place)
```

便求得 E.code 值，让它在 Ecode 中。下面要在属性信息表中建立相应条目。

对照数据结构，相应依赖属性链如图 6-22 所示，其中让 PE1code 指向这一依赖属性链。

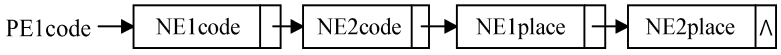


图 6-22

令函数调用 DPointer(N, p)的功能是生成依赖属性结点，即生成属性序号为 N，其下一依赖属性结点是指针 p 所指向的结点，返回值是所生成结点的指针。可写出如下的赋值语句：

```
PE2place=DPointer (NE2place, NULL);
PE1place=DPointer (NE1place, PE2place);
PE2code=DPointer (NE2code, PE1place);
PE1code=DPointer (NE1code, PE2code);
```

其中的 DPointer 函数是容易用 C 语言实现的，事实上，只需下列函数定义：

```
struct 依赖属性类型 * DPointer(int N, struct 依赖属性类型 *p)
{ struct 依赖属性类型 *q;
  q=(struct 依赖属性类型 *)malloc(sizeof(struct 依赖属性类型));
  q→属性序号=N;
  q→下一依赖属性=p;
  return q;
}
```

如上所述，属性值 E.code 在 Ecode 中，假定当前结点序号是 N，当前属性信息表条目计数器是 A，则由下列语句把 E.code 登录入属性信息表。

```
A=A+1;
属性信息表[A]={A, 'E', "code", 'M', Ecode, PE1code, N};
```

当然用 C 语言实现时，应分别对结构变量的成员变量赋值，即

```
属性信息表[A].属性序号=A;
属性信息表[A].文法符号='E'; /*实际实现时用序号代替*/
strcpy(属性信息表[A].属性名, "code");
```

```
属性信息表[A].属性值种类='M';
属性信息表[A].属性值=Ecode;
属性信息表[A].依赖属性链=PE1code;
属性信息表[A].结点序号=N;
```

再以例子说明 E.place:=newtemp 的实现。

一个临时变量名取 t1、t2 等形式。如何使 E.place 取得这样一个临时变量名？

每次使用 newtemp，便安排下一个临时变量，即序号加 1。因此引进一个计数器 Ntemp，初值为 0。需把 Ntemp 转换为字符串类型，并置在字母 t 之后，但并不是简单地执行下列赋值语句：

```
E.place="t" || itoa(Ntemp);
```

其中 itoa 是从整型到字符串型的类型转换函数。事实上，不仅仅有：

```
A=A+1; 属性信息表[A].属性值="t" || itoa(Ntemp);
```

应是赋值语句：

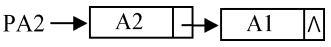
```
A=A+1; 属性信息表[A]={A,'E',"place",'S',"t"||itoa(Ntemp),NULL,N};
```

因为不依赖于任何其他属性，所以依赖属性链为 NULL。该语句用 C 语言实现时，同样需要分别对结构成员变量赋值来实现，并用字符串相关的函数实现对字符串的操作。

把属性值保存入注释分析树中的工作，一个是建立属性信息链，另一个就是存入注释分析树。假定当前正建立的注释分析树结点序号是 N，则由下列赋值语句建立属性信息链：

```
PA1=APointer (A1, NULL);
PA2=APointer (A2, PA1);
```

其中假定 E.place 与 E.code 在属性信息表中相应条目的序号分别是 A1 与 A2。图示如下：



其中的 APointer 函数类似于 DPointer，可以用 C 语言函数定义实现如下：

```
struct 属性结点类型 * APointer(int N, struct 属性结点类型 *p)
{ struct 属性结点类型 *q;
  q=(struct 属性结点类型 *)malloc(sizeof(struct 属性结点类型));
  q→属性序号=N;
  q→下一属性结点=p;
  return q;
}
```

使用下列赋值语句把该属性信息链存入注释分析树结点中：

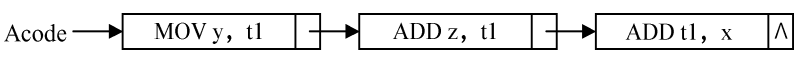
```
注释分析树[N].属性信息链=PA2;
```

3. 目标代码的生成与存储

前面求得 E.code 的值存放在 Ecode 中，但并没有明确如何实际求得，本节解决这个问题。

如前所述，目标代码作为文法符号的属性，例如，E.code 与 A.code 等，当生成之后存储在属性信息表条目中。目标代码通常是一系列目标指令，从书写上看，是一个非常长的字符串，因此必须为目标代码设计合适的数据结构。

目标代码是目标指令序列，其大小是随机变化的。合适的处理方法是，引进目标指令链，链上每个结点对应于一个目标指令，例如：



不难为目标指令链设计如下的数据结构：

```
struct 目标指令类型
{ char 虚拟机指令[MaxLength];
   struct 目标指令类型 *下一目标指令;
};
struct 目标指令类型 *目标指令链;
```

基于该数据结构，考察如何进行目标代码生成。还是以相应于 E::=E+E 的下列语义动作为例进行说明。

```
E.code:=E1.code||E2.code
       || gencode ("MOV",  E1.place, E.place)
       || gencode ("ADD",  E2.place, E.place)
```

显然，现在符号||不再是字符串并置运算符，而是代表链接，把两个目标指令链链接成一个目标指令链，即把右运算分量中的目标指令链链接到左运算分量中的目标指令链之后。首先考虑 E.code:=E_1.code||E_2.code。一般的处理过程是先把 E_1.code 传送入 E.code 中，然后再把 E_2.code 链接到 E.code 中最后一条指令之后。因此引进两个函数：CopyCode 与 MergeCode，函数调用 CopyCode(pcode)的功能是：复制指针 pcode 所指向的目标指令链，返回的是指针，它指向所复制目标指令链第一条目标指令；函数调用 MergeCode(pcode1, pcode2)的功能是：把 pcode1 所指向的目标指令链，链接到指针 pcode2 所指向目标指令链中最后一个目标指令之后，返回一个指针，它指向链接后的目标指令链中第一个目标指令。

函数调用 gencode(op, S, T)的功能是生成一条目标指令（字符串），回送指向该目标指令（字符串）的指针。

假定已取到 E_1.code、E_2.code、E_1.place 与 E_2.place 在属性信息表中条目的序号分别为 NE1code、NE2code、NE1place 与 NE2place，则可设计如下的实现程序如下：

```
Ecode=CopyCode(属性信息表[NE1code].属性值);
Ecode=MergeCode(属性信息表[NE2code].属性值, Ecode);
strcpy(E2place,属性信息表[NE2place].属性值);
strcpy(Eplace,属性信息表[A].属性值);
  /*由执行 E.place:=newtemp 填入的属性值*/
Addp=IPointer(gencode("ADD", E2place, Eplace), NULL);
strcpy(E1place,属性信息表[NE1place].属性值);
Movp=IPointer(gencode("MOV", E1place,Eplace),Addp);
Ecode=MergeCode(Movp, Ecode);
```

其中，IPointer 的函数定义可类似地设计如下：

```
struct 目标指令类型 *IPointer(char c[ ], struct 目标指令类型 *p)
{  struct 目标指令类型 *q;
   q=(struct 目标指令类型 *)malloc(sizeof(struct 目标指令类型));
   strcpy(q→虚拟机指令, c);
   q→下一目标指令=p;
   return q;
}
```

6.5.2　语义子程序及其执行

基于前面的讨论，可为赋值语句文法 G6.9[A]的翻译方案写出语义子程序如下。这里采用了 C 型语言书写，并为各语义子程序编号，取名为_1、_2 等。其中，U 是当前正被归约成的重写规则

左部非终结符号（实际实现时用序号代替），N 是正生成的注释分析树结点的序号，即为文法符号 U 生成的结点的序号。这是结合自底向上的 LR 分析技术所决定的。

```
 void  _1( )  /* A::=id=E */
 { NE=分析栈[tops].结点序号;NEcode=search(NE, "code");
   NEplace=search (NE, "place");
   Ecode=属性信息表[NEcode].属性值;
   if(Ecode==NULL)  /*判别 E.code=" "否*/
   { Ntemp=Ntemp+1; /* newtemp */
     strcpy(t, "r");  strcat(t, itoa(Ntemp));
     strcpy(Eplace,属性信息表[NEplace].属性值);
     Movp=IPointer (gencode("MOV", Eplace, t), NULL);
     属性信息表[NEcode].属性值=Movp;
     strcpy(属性信息表[NEplace].属性值, t);
   }
   Ni=分析栈[tops-2].结点序号;Niplace=search(Ni, "name");
   strcpy(Eplace,属性信息表[NEplace].属性值);
   strcpy(iplace,属性信息表[Niplace].属性值);
   Acode=CopyCode (Ecode);
   Movp=IPointer (gencode ("MOV", Eplace, iplace), NULL);
   Acode=MergeCode (Movp, Acode);
   PEcode=DPointer (NEcode, NULL);
   PEplace=DPointer (NEplace, PEcode);
   Piplace=DPointer (Niplace, PEplace);
   A=A+1;
   属性信息表[A]={A, U,"code",'M',Acode,Piplace,N};
   注释分析树[N].属性信息链=APointer(A,NULL);
}
void _2( )   /* E::=E+E */
{ Ntemp=Ntemp+1;  /* newtemp */
   strcpy(t, "r");  strcat(t,itoa(Ntemp));
   A=A+1;
   属性信息表[A]={A,U,"place",'S',t,NULL,N}; /*Eplace*/
   NE1=分析栈[tops-2].结点序号;NE1code=search(NE1, "code");
   NE2=分析栈[tops].结点序号;  NE2code=search(NE2, "code");
  Ecode=CopyCode(属性信息表[NE1code].属性值);
  Ecode=MergeCode(属性信息表[NE2code].属性值, Ecode);
  strcpy(Eplace,属性信息表[A].属性值);
  NE1place=search(NE1,"place");
  strcpy(E1place,属性信息表[NE1place].属性值);
  NE2place=search(NE2,"place");
  strcpy(E2place,属性信息表[NE2place].属性值);
  Addp=IPointer (gencode ("ADD", E2place, Eplace), NULL);
  Movp=IPointer (gencode ("MOV", E1place, Eplace), Addp);
  Ecode=MergeCode (Movp, Ecode);
  PE2place=DPointer (NE2place, NULL);
  PE1place=DPointer (NE1place, PE2place);
  PE2code=DPointer (NE2code, PE1place);
  PE1code=DPointer (NE1code, PE2code);
  PA1=APointer(A,NULL); /*E.place 的属性信息表条目序号为 A */
  A=A+1;
```

```
   属性信息表[A]={A,U,"code",'M',Ecode,PE1code,N};
   注释分析树[N].属性信息链=APointer(A, NULL);
}
void _3( )   /* E::=E*E */
{ Ntemp=Ntemp+1;  /* newtemp */
   strcpy(t,"r");  strcat(t,itoa(Ntemp));
   A=A+1;
   属性信息表[A]={A,U,"place",'S',t,NULL,N};
   NE1=分析栈[tops-2].结点序号;NE1code=search(NE1,"code");
   NE2=分析栈[tops].结点序号;  NE2code=search(NE2,"code");
   Ecode=CopyCode(属性信息表[NE1code].属性值);
   Ecode=MergeCode(属性信息表[NE2code].属性值, Ecode);
   strcpy(Eplace,属性信息表[A].属性值);
   NE1place=search (NE1,"place");
   strcpy(E1place,属性信息表[NE1place].属性值);
   NE2place=search (NE2,"place");
   strcpy(E2place,属性信息表[NE2place].属性值);
   Mpyp=IPointer(gencode("MPY", E2place, Eplace), NULL);
   Movp=IPointer(gencode("MOV", E1place, Eplace), Mpyp);
   Ecode=MergeCode (Movp, Ecode);
   PE2place=DPointer (NE2place, NULL);
   PE1place=DPointer (NE1place, PE2place);
   PE2code=DPointer (NE2code, PE1place);
   PE1code=DPointer (NE1code, PE2code);
   PA1=APointer(A,NULL);  /*E.place 的属性信息表条目序号为 A */
   A=A+1;
   属性信息表[A]={A,U,"code",'M',Ecode,PE1code,N};
   注释分析树[N].属性信息链=APointer(A,NULL);
}
void _4( )   /* E::=-E */
{ Ntemp=Ntemp+1;  /* newtemp */
   strcpy(t,"r");  strcat(t, itoa(Ntemp));
   A=A+1;
   属性信息表[A]={A,U,"place",'S',t,NULL,N};
   NE1=分析栈[tops].结点序号;NE1code=search(NE1, "code");
   Ecode=CopyCode(属性信息表[NE1code].属性值);
   strcpy(Eplace,属性信息表[A].属性值);
   NE1place=search (NE1, "place");
   strcpy(E1place,属性信息表[NE1place].属性值);
   Negp=IPointer (gencode ("NEG", Eplace), NULL);
   Movp=IPointer (gencode ("MOV", E1place, Eplace), Negp);
   Ecode=MergeCode (Movp, Ecode);
   PE1place=DPointer (NE1place, NULL);
   PE1code=DPointer (NE1code, PE1place);
   PA1=APointer (A, NULL); /*E.place 的属性信息表条目序号为 A */
   A=A+1;
   属性信息表[A]={A,U,"code",'M',Ecode,PE1code,N};
   注释分析树[N].属性信息链=APointer(A, PA1);
}
void _5( )   /* E::=(E) */
```

```
{ NE1=分析栈[tops-1].结点序号;NE1place=search(NE1,"place");
   PE1place=DPointer (NE1place, NULL);
   strcpy(E1place,属性信息表[NE1place].属性值);
   A=A+1;
   属性信息表[A]={A,U,"place",'S', E1place, PE1place,N};
   PA1=APointer (A, NULL); /*E.place 的属性信息表条目序号为 A */
   NE1code=search (NE1,"code");
   PE1code=DPointer (NE1code, NULL);
   strcpy(E1code,属性信息表[NE1code].属性值);
   A=A+1;
   属性信息表[A]={A,U,"code",'M', E1code, PE1code,N};
   注释分析树[N].属性信息链=APointer(A, PA1);
}
void _6( )   /* E::=id */
{ Ni=分析栈[tops].结点序号;Niplace=search(Ni,"name");
   Piplace=DPointer (Niplace, NULL);
   strcpy(place,属性信息表[Niplace].属性值);
   A=A+1;
   属性信息表[A]={A,U,"place",'S',iplace,Piplace,N};
   PA1=APointer (A,NULL); /*E.place 的属性信息表条目序号为 A */
   A=A+1;
   属性信息表[A]={A,U,"code",'M',NULL,NULL,N};
   注释分析树[N].属性信息链=APointer(A,PA1);
}
```

读者把这些语义子程序与关于文法 G6.9 的翻译方案相对照，可以理解应如何编写语义子程序。

现在最后一个要解决的问题是：何时执行语义子程序？

从翻译方案看，所有的语义动作都在重写规则右部的右端，也就是说，按自底向上 LR 分析技术，当处理完重写规则右部，进行归约时执行语义动作，因此可以明确：当在语法分析过程中，执行归约动作时调用相应的语义子程序。赋值语句目标代码生成的程序控制流程示意图如图 6-23 所示，其中（b）是归约时的控制流程示意图。

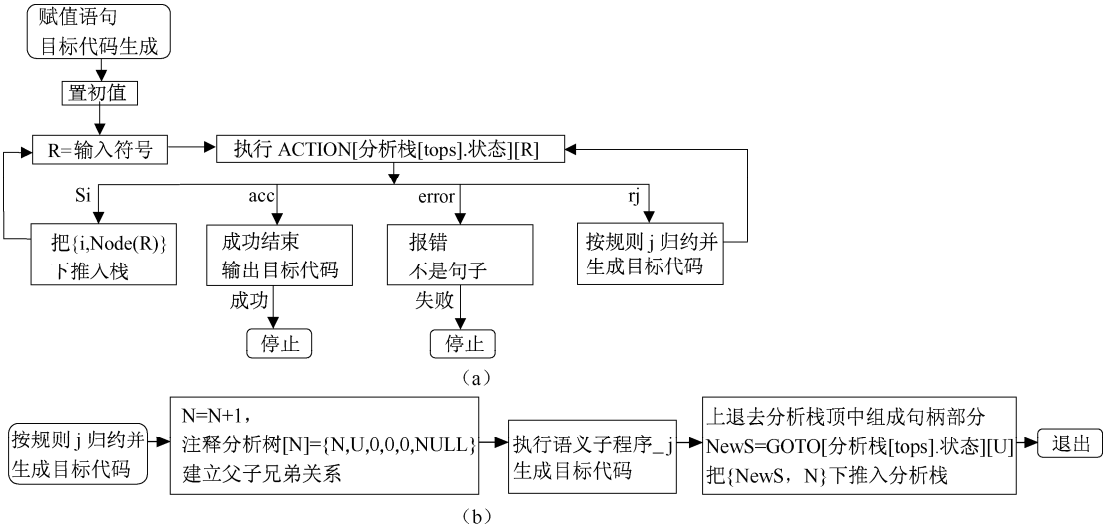


图 6-23

请读者注意，语义子程序名_1、_2、…中的编号是与相应重写规则的序号一致的。当按规则 j 归约时便能调用语义子程序_j（j=1,2,…）。在语法识别程序的实现中讨论过 LR 分析表的存储表示，即用数值表示分析表元素，例如，S2 用+2 表示，r6 用−6 表示，因此可以如下地通过 switch 语句快速调用语义子程序。

```
j=ACTION[分析栈[tops].状态][R];
if(j<0)   /* 分析动作是 rj */
{  j=-j;
   switch(j)
   {  case 1:  _1( );  break;
      case 2:  _2( );  break;
      ⋮
   }
   ⋮
}
```

本 章 小 结

本章讨论语义分析与目标代码生成，包括概况、说明部分的翻译与控制部分的翻译。特别讨论了翻译方案的计算机实现。

一个程序涉及数据结构和控制结构两部分，因此语义分析应包括对数据结构和控制结构两部分的语义分析，功能包括类型确定、类型检查、识别含义并作相应的语义处理，以及其他一些静态语义检查。

为了和具体程序设计语言无关地进行语义分析，引进类型表达式的概念，使得能用一种一致的、与具体语言无关的表示来描述类型。读者应熟练掌握类型表达式的书写。

对于词法和语法，能用形式表示法和使用典型而成熟的识别算法实现词法分析和语法分析，而一个程序设计语言的语义分析往往显得支离破碎，为此采用所谓的语法制导翻译技术进行语义分析和目标代码的生成。语法制导翻译技术的基本思想是：为文法符号引进属性，在给出语法规则的同时，给出计算属性的语义规则，这样，在语法规则制导下，通过对语义规则的计算，实现对源程序的翻译。

由于文法中每个符号都对应于一个语法成分，如对应于变量或对应于表达式，等等，具有各自的自身特性，例如类型与值，可以为文法符号引进一些属性，这样便形成了属性文法。属性有继承属性和综合属性两类。属性文法是扩充有属性和属性计算规则的、压缩了的上下文无关文法。属性文法有语法制导定义与翻译方案两种形式。翻译方案是涉及实现细节的，即语义动作可安排在语法规则右部的任何位置。必须记住：计算一个属性时，它所依赖的所有属性都必须已计算好。相关的概念有注释语法树、依赖图、S—属性定义、L—属性定义等。

语法制导定义和翻译方案可看作是语义分析和目标代码生成的规范，读者应能熟练地阅读它们。

对说明部分的翻译，并不生成目标代码，只是把相关的信息填入符号表中，这是容易实现的。控制部分的翻译是翻译的重点，控制部分在翻译之后生成相应的目标代码。为了实现目标代码生成，必须考虑相关的若干要点。最关键的是：基于语义，明确语法成分与目标代码的对应关系。一个最直观的思考方式是：首先明确一个控制语句的语义和执行步骤，从而确定目标代码的结构，

按照此目标代码结构，对应地设计翻译方案。相关概念有地址回填、转移指令表、活动记录。

翻译方案中的语义动作，本质上是一个与源程序一样的符号串，并不能直接执行。一个简单的实现办法是根据翻译方案，设计相应的语义子程序。从简单的实例出发，进行观察与分析，明确要解决的问题及解决思路，就可得到求解问题的答案。对照赋值语句的翻译方案与注释分析树，可以概括出编写语义子程序时的 3 个要点，即属性值的存取、属性值的计算与目标代码的存储。

复习思考题

1．语义分析的功能有哪些？每个功能处理的是程序中哪部分？

2．属性文法中的属性是什么？为什么可以为文法符号引进属性？

3．注释分析树与语法分析树的区别是什么？

4．设计翻译方案时，应注意哪些方面？

5．类型表达式与程序设计语言中的类型有何联系，又有何区别？

6．翻译说明部分时，如何确定标识符的作用域？

7．目标代码生成中，必须考虑哪些方面？其中最关键的是什么？

8．C 语言控制结构的目标代码结构设计中，有哪些是需要特别注意的？

9．函数调用时，需要完成大量的工作，如何使得既能完成这些工作，目标代码生成的工作又较简单？

习　　题

1．设文法 G6.2[L]：

```
L::=E      E::=E+T    E::=E-T    E::=T
T::=T*F   T::=T/F     T::=F      F::=(E)    F::=N
```

其中 N 表示整数。试为输入符号串 9/3+7 构造注释分析树，语法制导定义见表 6.1。

2．设文法 G6.3[D]：

```
D::=T  L      T::=int    T::=float    L::=L,id     L::=id
```

试为输入符号串 int a,c,d 构造注释分析树，语法制导定义见表 6.2。

3．试根据 6.1.3 的 3 中为左递归文法设计基于自顶向下分析技术的翻译方案，为下列文法 G6.10[B]构造翻译方案。

```
G6.10[B]:B::=B||L | L          L::=L && R  |  R
         R::=E relop E  | V     E::=N
         V::=true | false
```

其中，relop 是关系运算符，代表运算符<、<=、>、>=、==、!=。

4．试写出下列 C 语言变量说明，并写出相关联的类型表达式。

（1）一个一维整型数组变量，数组元素个数是 12；

（2）指向实型数据的指针数组，该数组元素个数是 30；

（3）一个函数，实参是整型值，返回值是一个指针，它指向由一个字符值和一个实型值组成的结构变量。

5．设有 C 语言说明如下：

```
typedef struct NodeT
{ int info; struct NodeT *next;
} NodeType;
NodeType *Np;
NodeType Tree[100];
```

试写出分别与 Np 和 Tree 相关联的类型表达式。

6．设有文法 G[E]：

```
E::=E+T|E-T|T    T::=T*F|T/F|F  F::=(E) | num | num.num | literal
```

其中 literal 表示字面上的值，即字符，num 表示数字字符串，即整数。试写出确定表达式的类型，同时进行类型检查的语法制导定义。

7．试根据翻译方案，为赋值语句 x=a*(b+c)−(e−f)/d 写出虚拟机目标代码。

8．试根据翻译方案，依两种方式为 if 语句 if(x>10||x<5) y=x+5; else y=10−x; 写出虚拟机目标代码。

（1）求出逻辑值的方式；

（2）以控制转移的位置反映逻辑值的方式。

9．试根据翻译方案，写出下列程序片断的虚拟机目标代码。

```
s=0;  j=1;
while(j<=20)
{ s=s+j;  j=j+1;}
printf("s=%d",s);
```

10．试为 x=a+b 所生成的目标代码：

```
MOV  a,   R0
ADD  b,   R0
MOV  R0,  x
```

生成目标指令链。由指针 head 指向其第一个指令的结点。

11．试写出实现 E.place:=newtemp 的 C 语言程序。假定这是第一次登录属性信息表。

12．试写出实现 E.code:=″ ″的 C 语言程序。目标指令的数据结构参见 6.5 翻译方案的实现。

第 7 章 中间表示代码与代码优化

本 章 导 读

第 6 章讨论了语义分析与目标代码生成。一般地说，编译程序只能按共性来生成目标代码，因此，与手工编写的程序相比较，目标程序的质量是甚为低下的，有必要进行代码优化。通常，代码优化是编译时刻在中间表示代码一级上进行的。本章讨论代码优化问题，内容包括：概况、源程序的中间表示代码、基本块的代码优化、与循环有关的优化、全局优化的实现思想，以及窥孔优化。源程序的中间表示代码可以有抽象语法树、逆波兰表示、三元式序列与四元式序列。本章重点讨论在四元式序列基础上进行的、与机器无关的优化。读者应能熟练地为 C 语言程序写出相应的四元式序列，并从四元式序列生成目标代码；熟悉基本块的优化种类、并了解实现手段：dag；熟悉与循环有关的优化种类，了解实现与循环有关优化的思路，并理解三类数据流方程的含义与作用；了解窥孔优化的语言级及优化种类。建议读者在首先保证正确的前提下，把代码优化的思想应用于 C 语言程序设计中。

7.1 概　　况

7.1.1 代码优化与代码优化程序

本章讨论目标代码优化问题，先看下列例子。

【例 7.1】 设有 if 语句

```
if(a+b>0) x=(a+b)*2;  else x=(a+b)/2;
```

按第 6 章相应翻译方案将生成下列目标代码：

```
(1) MOV   a,   R0        (10) MOV   R2,   x
(2) ADD   b,   R0        (11) GOTO  (17)
(3) CMP   R0,  #0        (12) MOV   a,    R3
(4) CJ>   (6)            (13) ADD   b,    R3
(5) GOTO  (12)           (14) MOV   R3,   R4
(6) MOV   a,    R1       (15) DIV   #2,   R4
(7) ADD   b,    R1       (16) MOV   R4,   x
(8) MOV   R1,   R2       (17)
(9) MPY   #2,   R2
```

如果用手工编写，将会有如下的代码：

```
(1) MOV   a,   R0        (5) MPY   #2,   R0
(2) ADD   b,   R0        (6) GOTO  (8)
```

```
(3) CMP    R0, #0       (7) DIV    #2,   R0
(4) CJ≤    (7)          (8) MOV    R0,   x
```

两者相比，前者的目标指令数多达 16 条，竟是后者的两倍，且临时变量（寄存器）有 5 个。这是因为编译程序只能按照程序设计语言的一般情况，根据翻译方案中的规定生成目标代码，不能根据源程序的具体上下文生成高功效的目标代码。因此编译程序生成的目标代码，与手工编写的程序相比较，效率远为低下。尽管计算机发展非常迅速，运算速度与存储容量与早期已不可同日而语，但随着应用领域的日趋广阔，尤其是人工智能、图像处理与数据库处理等方面软件的发展，往往依然受到计算机容量和速度之限制的困扰。因此，在保证程序正确性的前提下，依然在努力改进，提高程序的功效。注意到一个程序中往往存在着某些关键部分，典型的就是循环结构，它们在整个程序总长中仅占一小部分，但运行时间却可能占整个程序运行时间的绝大部分。对这样的关键部分，功效的改进将对整个运行的效率有重大影响。

对程序中的关键部分进行改进，提高程序运行功效是可行的。通常进行改进、提高功效的途径有 4 个，即改进算法、利用系统提供的库程序、源程序级程序变换以及编译时刻代码优化。本章专门讨论编译时刻代码优化问题。

代码优化指的是编译时刻为改进目标程序质量而进行的各项工作。代码优化的目的是改进目标程序质量，包括减少存储空间与缩短运行时间。由于时空是一对相互冲突与相互制约的因素，往往是采取折中或者依据实际情况而偏重某一方面。要保证得到的目标程序最优是困难的，一般不说是最优化，而仅是代码优化。

目标代码通常与计算机指令紧密相关，且难以从目标代码获得有利于优化的信息，分析工作复杂而代价高，因此不直接对所生成的目标代码进行代码优化，通常的思路是：在语义分析时仅生成中间表示代码，对中间表示代码进行优化，从优化了的中间表示代码生成目标代码，然后再对目标代码进行小范围的优化，即所谓的窥孔优化。

编译程序中完成代码优化工作的部分称为**代码优化程序**。代码优化程序的输入是语义分析阶段生成的某种中间表示代码，输出是通过优化而改进了的中间表示代码。中间表示代码可以有抽象语法树、逆波兰表示、四元式序列与三元式序列等。本章讨论的将是在四元式序列基础上进行的代码优化，关于四元式序列将在后面讨论。

7.1.2　代码优化的分类

代码优化工作可以从 3 个角度，即与机器相关性、优化范围及优化语言级来分类。

1. 与机器相关性

按照与机器相关性的程度，代码优化可分为与机器相关的优化和与机器无关的优化。

与机器相关的优化，一般有涉及寄存器的优化、多处理器的优化、特殊指令的优化及消除无用指令的优化等。这几类优化，不言而喻，与具体机器的特性紧密相关。例如，可用寄存器的总数、使用寄存器的具体规定、单处理器还是多处理器、多处理器时处理器又有几个，等等。为了教学需要，仅考虑虚拟机目标代码，回避了了解与熟悉特定计算机特性等问题，因此不去讨论与机器相关的优化，无需在讨论优化时去关心目标机的细节特性。对于与机器相关的优化，仅引进窥孔优化的概念。

2. 优化范围

从优化范围的角度，代码优化分为局部优化与全局优化两类。

局部优化是指仅在称为基本块的四元式序列的局部范围内进行的代码优化。因为仅在一个基

本块范围内进行优化，所以不考虑基本块之间的相互联系。当在考察若干个基本块之间的相互联系与影响的基础上进行优化，就将不再是局部的，而是全局的优化。不论是局部优化还是全局优化，讨论的将都是与机器无关的优化。

局部优化时，可对基本块进行下列优化，即合并常量计算、消除公共子表达式、削减计算强度与删除无用代码等 4 类。

全局优化时，重点是进行与循环相关的优化。与循环相关优化的种类包括：循环不变表达式外提、归纳变量删除与计算强度削减等。

不言而喻，全局优化是在对基本块优化的基础上进行的，进行全局优化时，也将进行对基本块的局部优化。

3. 优化语言级

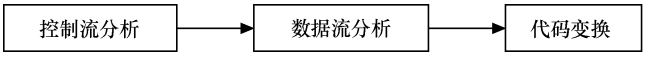
如前所述，优化是针对中间表示代码，主要是对四元式序列进行的，与计算机具体特性无关，更易了解代码优化的思路。这样也比从目标语言代码更易识别出可进行优化的情况。

除了中间表示代码级优化，也可以对目标代码进行优化，但进行全局优化的代价将是高昂的，因此这时仅考虑较小范围内的局部优化，通常称为窥孔优化。这样无需花费太大代价就可取得较好的效果。

在源程序一级上进行等价变换来改进目标代码质量方面，在国内外开展了相当的工作。但这并不是本章所讨论的代码优化，不拟讨论。只是为理解优化的思路，有时也会通过源程序例子来说明。

概括起来，本章讨论的重点是在中间表示代码级上的、与机器无关的局部优化和全局优化。

代码优化程序通常由 3 个部分组成，即控制流分析、数据流分析与代码变换，示意图如下所示。



（1）控制流分析

如前所述，源程序中的循环结构，在翻译之后生成的目标代码中，由条件判断和控制转移的组合来实现，而循环结构是代码优化的重点内容，因此必须识别出循环结构。这时不是简单地识别关键字 while、do 与 for 等，在目标代码中它们已不复存在。为此，必须采用一定的措施来识别出循环结构，这就是控制流分析。控制流分析的任务就是基于流图识别出循环结构。

（2）数据流分析

这部分进行数据流信息的收集。这些信息是反映程序中变量值的获得和使用情况的数据流信息，包括到达一定值、活跃变量与可用表达式等。为此，建立一些相应的数据流方程组，通过解这些方程组来获得这些信息，基于从数据流分析所获得的流图中基本块之间的数据流信息，进行循环优化和各种全局优化。

（3）代码变换

代码变换部分对中间表示代码逐个进行分析，在基本块内进行等价变换，从而实现循环的优化和各种全局优化。

从变换后最终生成的、优化了的中间表示代码生成目标代码。

7.2 源程序的中间表示代码

当编译程序分为几遍来完成源程序的翻译时，每遍都将产生一种内部中间表示，直到最终生

成目标代码，例如，词法分析后的属性字序列，语法分析后的语法分析树等都是内部中间表示。本节讨论的内部中间表示是指语义分析产生的中间表示代码。实际实现中，按语法制导的翻译方案来生成中间表示代码，可看成是语法分析与语义分析相结合的产物。

中间表示代码可以是抽象语法树、逆波兰表示、四元式序列与三元式序列，重点讨论四元式序列。

7.2.1　四元式序列

1. 表示法约定

一个双目运算符对两个运算分量进行操作，产生一个结果，因此，引进的中间表示代码可以具有下列形式：

运算符　运算分量　运算分量　结果

其中运算分量和结果可以是变量或由编译程序引进的临时变量，运算分量还可以是常量。这种形式的中间表示代码称为**四元式**。例如，相应于表达式 x+y*z 的四元式序列是：

运　算　符	运 算 分 量	运 算 分 量	结　　果
*	y	z	t1
+	x	t1	t2

其中 t1 和 t2 是编译程序引进的临时变量，显然用表格的形式是更清楚的。

对应于语言中的各种语法成分，除了双目运算的四元式外，还允许有对应于单目运算、关系运算、控制转移与函数调用等各种形式的四元式，它们分别约定如下：

对于单目运算 op　x，如单目减、逻辑非与类型转换等，其四元式呈以下形式：

op	x		t

其中不包含第二个运算分量，运算结果在临时变量 t 中。

对于赋值语句 x=y，其四元式呈以下形式：

:=	y		x

其中赋值号沿用符号:=。

为了能间接赋值，把 y 的值传送入以 x 的值为存储地址的存储字中，引进间接赋值运算&:=，相应的四元式呈以下形式：

&:=	y		x

对于按关系运算 relop 比较，按真转的四元式呈以下形式：

relop	x	y	L

其中，relop 可以是<、≤、>、≥、=与≠，其含义是：当 x relop y 为 true 时，控制转向序号为 L 的四元式，否则执行紧接该四元式的下一个四元式。显然，这相当于下列 C 语言语句：

```
if(x relop y) goto L;
```

对于转向语句 goto L，其四元式呈以下形式：

GOL	L		

四元式允许有标号，该标号代表四元式序号。如果无条件控制转移到序号为 L 的四元式处，

相应四元式呈以下形式：

GO	L		

为简单起见，关于 GOL 与 GO 的四元式中标号与序号兼用。

对于函数调用 $p(x_1,x_2,\ldots,x_m)$，分别关于参数和函数调用，约定四元式呈以下形式：

PARAM	x_1		
PARAM	x_2		
……			
PARAM	x_m		
CALL	p	m	

（无值情况）

或

CALL	p	m	t

（有值情况，值在 t 中）

对于有说明部分的复合语句，即分程序，为标志其开始与结束，约定四元式如下：

BLOCK			
ENDBLOCK			

对于数组元素的四元式表示约定，以例子说明。设有赋值语句：

```
A[j]=B[k+m];
```

相应的四元式序列如下：

(1)	+	k	m	t2
(2)	=[ ]	B	t2	t3
(3)	[ ]=	A	j	t1
(4)	&:=	t3		t1

其中，=[ ]表示取数组元素值操作，而[ ]=表示取数组元素地址操作。不言而喻，当数组元素出现在赋值语句左边时将取地址，用[ ]=，其他情况取值，用=[ ]。四元式中的数组名（如 B）表示为数组（B）分配的存储区域的首地址。四元式（2）中 t3 的值是 B[t2]，即 B[k+m]的值，因此 t2 的值是 B[k+m]相对于数组 B 的存储区域首地址的位移量。显然，一般情况下还需考虑到数组元素所占存储区域的大小，此位移量还需乘上数组元素所占存储字节数。例如，假定都是整型数组，元素大小为 2，将把上述四元式序列修改如下：

```
(1)  +     k    m    t3          (4)  *     j   2    t1
(2)  *     t3   2    t4          (5)  [ ]=  A   t1   t2
(3)  =[ ]  B    t4   t5          (6)  &:=   t5       t2
```

请读者注意临时变量的编号。在前面的四元式序列中，四元式（1）中是 t2，在四元式（3）中才是 t1，这是由翻译方案所决定的，四元式序列的生成必须结合翻译方案。

2. 四元式序列的例子

【例 7.2】 设有字符数组 S，包含 n 个字符，将其内容置为逆序的 C 程序如下：

```
j=0;
while(j < n/2)
{ c=S[j];   S[j]=S[(n-1)-j];  S[(n-1)-j]=c;
  j=j+1;
}
```

如前目标代码生成的翻译方案所见，该循环语句将展开成下列语句：

```
j=0;
loop: if(j<n/2)
      { c=S[j];   S[j]=S[(n-1)-j];  S[(n-1)-j]=c;
        j=j+1;
        goto  loop;
      }
```

数组元素为字符，仅占 1 个存储字节，相应的四元式序列如下：

```
(1)   :=     0        j          (11)   &:=    t6          t3
(2)   /      n    2   t1         (12)   -      n    1      t7
(3)   <      j    t1  (5)        (13)   -      t7   j      t8
(4)   GO     (19)                (14)   [ ]=   S    t8     t9
(5)   =[ ]   S    j   t2         (15)   &:=    c           t9
(6)   :=     t2       c         (16)   +      j    1      t10
(7)   -      n    1   t4         (17)   :=     t10         j
(8)   -      t4   j   t5         (18)   GO     (2)
(9)   =[ ]   S    t5  t6         (19)
(10)  [ ]=   S    j   t3
```

其中对 if 语句中逻辑表达式的处理，请参看 if 语句的翻译部分，E 的值为 true 时，控制转向 E.true，值为 false 时，控制转向 E.false。

7.2.2　生成四元式序列的翻译方案的设计

为程序设计语言的语言成分生成四元式序列的翻译方案的设计，其思路与生成目标代码是一样的，即先确定语言成分与四元式序列的对应关系，然后对照对应关系来设计翻译方案。以赋值语句文法 G7.1[A]：

```
A::=id=E    E::=E+E | E*E | -E |(E) | id
```

为例说明。

对于重写规则 E::=E+E，简单地生成四元式：

```
+   E左.place   E右.place    E.place
```

这里为左部变量 E 引进临时变量作为 E.place。因此可以有

```
E.place:=newtemp;
```

newtemp 表示当前下一个临时变量，而 $E_{左}$与 $E_{右}$是加法运算的左右运算分量。对于 A::=id=E，简单地生成四元式：

```
:=    E.place              id.place
```

其中，id.place 需从符号表相应条目中取到，因此有：

```
p:=lookup (id.name);
```

id.place 就是 p→place。类似地考虑其他重写规则的情况。

如前目标代码生成时的处理方式，这时有两种存储四元式序列的方式：一种方式是把四元式序列作为属性存储，类似于 E.code，引进 E.quad（Quadruple 四元式），这时引进函数 genquad，由函数调用 genquad(op, x, y, z)来生成下列四元式：

```
op   x   y   z
```

另一种方式是如同 emit 那样立即输出四元式，引进 emitquad，由函数调用 emitquad(op, x, y, z)输出下列四元式：

```
op   x   y   z
```

这里采用第二种方式设计文法 G7.1[A]的翻译方案如下：

```
A::=id=E { p:=lookup(id.name);
           emitquad (":=", E.place, " ", p→place) }
E::=E1+E2 { E.place:=newtemp;
            emitquad ("+", E1.place, E2.place, E.place) }
E::=E1*E2 { E.place:=newtemp;
            emitquad ("*", E1.place, E2.place, E.place) }
E::=-E1 { E.place:=newtemp;
          emitquad ("-", E1.place, "  ", E.place) }
E::=(E1) { E.place:=E1.place }
E::=id { p:=lookup(id.name);  E.place:=p→place) }
```

进一步可为 if 语句设计实现回填的生成四元式序列的翻译方案如下：

```
S::=if(E)M1 S1 N else M2 S2
{ backpatch (E.truelist, M1.pos); backpatch(E.falselist, M2.pos);
  S.nextlist:=mergelist (S1.nextlist, S2.nextlist);
  S.nextlist:=mergelist (S.nextlist, N.nextlist)    }
N::=ε{ N.nextlist:=makelist(nextpos+1);  emitquad("GOTO","_")  }
M::=ε{ M.pos:=nextpos+1 }
S::=if(E)M S1
{ backpatch (E.truelist, M.pos);
  S.nextlist:=mergelist (E.falselist, S1.nextlist)    }
E::=E1 && M E2
{ backpatch (E1.truelist, M.pos);
  E.truelist:=E2.truelist;
  E.falselist:=mergelist (E1.falselist, E2.falselist)    }
E::=E1 || M E2
{ backpatch (E1.falselist, M.pos);
  E.truelist:=mergelist (E1.truelist, E2.truelist);
  E.falselist:=E2.falselist                              }
E::=!E1
{ E.truelist:= E1.falselist;  E.falselist:=E2.truelist   }
E::=id1 relop id2
{ E.truelist:=makelist(nextpos+1);
  E.falselist:=makelist(nextpos+2);
  emitquad (relop, id1.place, id2.place, "_");
  emitquad ("GOTO", "_")                          }
E::=true { E.truelist:=makelist(nextpos+1); emitquad("GOTO","_") }
E::=false { E.falselist:=makelist(nextpos+1); emitquad("GOTO","_") }
```

其中，nextpos 的值在每次生成四元式时加 1，总是正待生成的四元式的序号。

与生成目标代码的翻译方案相比较，显然这是更为简单的。如果让标识符名表示所分配存储位置，则无需引进函数 lookup，这时可把 E.place:=p→place 改写为：

```
E.place:=id.name;
```

【例 7.3】 基于上述翻译方案，可为下列 if 语句：

```
if(a>b && c<d || e==f) x=m; else x=n;
```

生成下列四元式序列：

```
(1)  >   a   b   (3)          (6)  GO  (9)
(2)  GO  (5)                  (7)  :=   m       x
(3)  <   c   d   (7)          (8)  GO  _
(4)  GO  (5)                  (9)  :=   n       x
(5)  =   e   f   (7)
```

四元式(8)中控制转向的去处尚未回填。这时整个 if 语句 S 的属性 S.nextlist={8}，而 nextpos=9，将由外围控制结构进行回填，把下一四元式序号 nextpos+1=10 回填入四元式(8)中。另外请注意，为了处理 if 语句的内嵌赋值语句，需添加重写规则 S::=A。

显然对照目标代码生成的翻译方案，不难为 while 语句等迭代结构设计生成四元式序列的翻译方案，请读者自行完成。

7.2.3　从四元式序列生成目标代码

当四元式序列中包含的一切运算符都有相应的目标机操作码时，从四元式序列生成目标代码的工作容易实现。基于虚拟机目标机，这项工作变得更为方便。现在重点关注的是运算分量与计算结果的存取问题，核心是寄存器，因为运算分量是否在寄存器中，以后是否还会被使用等，对目标指令的生成有一定的影响。例如，假定要为四元式：

```
+   x   y   z
```

生成目标代码。生成的目标代码可有下列几种情况：

- `ADD   Rj,   Rk`

这里假定 x 与 y 的值分别在寄存器 R_k 与 R_j 中，且此四元式后不再引用 x，计算结果 z 在 R_k 中。

- `MOV   Rk,   Rm`
  `ADD   Rj,   Rm`

这里假定 x 与 y 的值分别在寄存器 R_k 与 R_j 中保持不变，且有寄存器 R_m 可用，计算结果 z 在 R_m 中。

- `MOV   x,   Rm`
  `ADD   y,   Rm`

这里假定 x 与 y 的值都不在寄存器中，有寄存器 R_m 可用，计算结果 z 在寄存器 R_m 中。

如果要把 z 的值保存在存储器字中，需有目标指令：

```
MOV   Rm,   z
```

这里仅讨论简单的目标代码生成算法，它基于基本块，下面进一步看基本块的概念。

1．基本块

基本块是一个连续的四元式序列，控制从其第一个四元式进入，从其最后一个四元式离开，其中没有停止，也不包含分支。

例如，下列四元式序列是一个基本块：

```
*    a    a    t1
*    b    b    t2
+    t1   t2   t3
*    c    c    t4
+    t3   t4   t5
/    t5   3    t6
```

其中第一个四元式称为**入口四元式**，最后一个称为**出口四元式**。

一个四元式序列通常由若干个基本块组成。划分基本块的思路是：

- 第一个四元式是入口四元式；
- 由条件或无条件控制转移四元式转移到的四元式是入口四元式；
- 紧随在条件或无条件控制转移四元式之后的四元式是入口四元式。

这样，从一个入口四元式到下一个入口四元式之前或到出口四元式为止的一切四元式构成一个基本块。

【例 7.4】 设有按霍纳方案计算多项式 $y=a_{10}x^{10}+a_9x^9+\cdots+a_2x^2+a_1x+a_0$ 值的 C 语言程序如下：

```
i=10;   s=0;
while (i>=0)
{ s=s*x+A[i];
  i=i-1;
}
y=s;
```

其中各个系数存放在数组 A 中，未考虑数组元素存储大小的相应四元式序列如下：

```
(1)  :=    10          i           (7)  +     t1     t2     t3
(2)  :=    0           s           (8)  :=    t3           s
(3)  ≥     i     0     (5)         (9)  -     i      1     t4
(4)  GO    (12)                    (10) :=    t4           i
(5)  *     s     x     t1          (11) GO    (3)
(6)  =[ ]  A     i     t2          (12) :=    s            y
```

其中入口四元式的序号分别是 1、3、4、5 与 12，因此，该四元式序列由 5 个基本块组成，它们分别包含序号为 1～2、3、4、5～11 与 12 的四元式。

2. 目标代码生成算法

以虚拟机为目标机，四元式序列中涉及的一切运算符都可以有相应的目标指令操作码。如前所述，重点是运算分量与计算结果的存取，核心是寄存器的使用。为简单起见，可以假定寄存器个数没有限制，在生成目标指令时，尽可能使用寄存器，而且尽可能长时间地把计算结果保留在寄存器中，这样将有利于功效的提高。

为了管理寄存器的使用，充分利用寄存器，需要跟踪寄存器的内容与每个变量的当前值和存储处所，因此引进寄存器描述符和地址描述符。

寄存器描述符用来记录每个寄存器当前保存的是什么变量的值。初始时，寄存器描述符显示一切寄存器都为无值，即空的。每当生成一个目标指令，便模拟执行该目标指令，在寄存器描述符中，记录哪个寄存器中存放了哪个变量的值。不言而喻，一个寄存器在一个特定时刻可能无值，可能存放一个变量的值，也可能存放多个变量的值。

地址描述符用来记录每个变量的当前值存放在何处。变量的当前值，可能存放在存储字中，也可能存放在寄存器中，还可能既在存储字中，又在寄存器中。这些信息可以保存在符号表中，以便在生成目标指令时确定对变量的访问方式，也即确定寻址方式。

下面以赋值语句为例说明。

设有下列赋值语句

```
x=a/(b+c)-d;
```

其相应四元式序列如下：

```
+    b    c    t1
/    a    t1   t2
-    t2   d    t3
:=   t3        x
```

假定 a、b、c 与 d 的值在存储字中，t1、t2 与 t3 是临时变量，通常将为上述四元式序列生成如下的目标代码：

```
MOV   b,    R0          为 t1 分配寄存器 R0
ADD   c,    R0          b+c=>R0
MOV   a,    R1          a=>R1
DIV   R0,   R1          为 t2 分配寄存器 R1
SUB   d,    R1          为 t3 分配寄存器 R1
MOV   R1,   x           R1=>x
```

可以用表的形式来表达寄存器描述符与地址描述符如表 7.1 所示。

表 7.1

四元式序列	生成的目标代码	寄存器描述符	地址描述符
(1) + b c t1	MOV b, R0 ADD c, R0	R0 包含 t1	t1 在 R0 中
(2) / a t1 t2	MOV a, R1 DIV R0, R1	R0 包含 t1 R1 包含 t2	t1 在 R0 中 t2 在 R1 中
(3) − t2 d t3	SUB d, R1	R0 包含 t1 R1 包含 t3	t1 在 R0 中 t3 在 R1 中
(4):= t3 x	MOV R1, x	R1 包含 t3, x	t3 在 R1 中 x 在 R1 和存储字中

为了尽可能使用寄存器，需要有一个实现算法，因此引进函数 GetRegister（寄存器 register），其功能是：确定四元式 op x y z 执行后存放 z 的值的处所 L。函数 GetRegister 确定 L 的思路如下：

- 如果 x 的值在寄存器中，此寄存器仅包含 x 的值，且在执行运算 x op y 后也不再引用此 x 的值，则以此寄存器作为 L（四元式(3)的情况）。这时修改地址描述符，指明 x 的值不再在 L 中；
- 不是上述情况，且寄存器有空闲的，则以某个寄存器作为 L，这时对寄存器描述符要作相应的修改（四元式(1)的情况）；
- 如果因寄存器个数有限而找不到寄存器，但又必须使用寄存器，则必须在被使用的寄存器中腾出一个寄存器来，即先用 MOV 指令把一个寄存器的值保存入存储字中，然后使用该寄存器。不言而喻，为在众多的被使用寄存器中选出此寄存器，使得是最合适的，需要收集更多的信息；
- 如果选择空闲寄存器有麻烦，而所有的基本块内又不再引用 z 的值，则可选择 z 的存储字 M 作为 L。

显然，函数 GetRegister 设计的好坏，影响着对 L 的选择的好坏。

基于寄存器描述符和地址描述符及 GetRegister 函数，可以给出从基本块四元式序列生成目标代码的简单生成算法如下。

输入：构成一个基本块的四元式序列。

输出：相应的目标代码。

对每个四元式 op x y z 执行下列步骤：

步骤 1　调用函数 GetRegister，确定存放 x op y 的计算结果 z 的处所 L；

步骤 2　查看地址描述符，确定 x 的当前值的一个存放处所 x′。如果 x 的当前值既在存储字中，又在寄存器中，则选择寄存器作为 x′。如果 x 的值并不在由步骤 1 确定的 L 中，则生成目标指令：

```
MOV x', L
```

把 x 的值传送到 L。

步骤 3　生成目标指令 op y′, L，其中 y′是 y 的当前值所在处所之一，同样尽可能选寄存器。修改地址描述符，指明 z 的值存放在 L 中。当 L 是寄存器时，同时修改寄存器描述符，指明该寄存器 L 包含 z 的值。

步骤 4　如果 x 和/或 y 的当前值在寄存器中，并且之后不再被引用，甚至在基本块的最后一

个四元式之后也如此，则修改寄存器描述符，指明在执行该四元式 op x y z 后，这些寄存器不再包含 x 和/或 y 的值。

因为上述目标代码生成算法是仅关于一个基本块的，在基本块的出口四元式之后，是否还引用该基本块内的变量是无法确定的，必须结合数据流分析，因此，假定基本块中用户定义的一切变量在该基本块出口四元式之后还要被引用。对于值不在存储字中，而仅在寄存器中的变量，必须用 MOV 指令把它们保存在存储字中。

一个考虑周密而质量较好的目标代码生成算法，显然需要收集更多的信息，而这往往是全局性的，这就需要由代码优化程序的代码变换部分完成。

除了双目运算四元式 op x y z 外，还有单目运算四元式 op x z，这与上述算法步骤类似。至于赋值四元式:= x y，在生成目标代码时，注意不要生成 MOV x, y 这样的目标指令，其中 x 与 y 的值都在存储字中，这时需判别 x 的值是否在寄存器中。如果是，譬如说在寄存器 R 中，则只需生成 MOV R, y，这时只需对寄存器描述符和地址描述符作出相应修改。如果 x 的值在存储字中，可以调用 GetRegister 函数，取到存放 x 当前值的寄存器 R，生成下列目标指令：

```
MOV  x,   R
MOV  R,   y
```

这时对寄存器描述符与地址描述符作相应的修改。

关于关系运算符四元式与控制转移四元式等，及其他各类四元式，不难实现目标指令的生成，要点是确定标号与四元式序号所相应的目标指令位置，应用回填技术，引进转移指令表进行回填。

3. 关于数组元素的四元式的目标代码

如同 C 语言等大多数程序设计语言，必须考虑数组的情况。这里考虑关于数组元素的四元式如何生成目标代码。以例子进行说明。设有赋值语句 x=A[k]与 B[j]=y，为实现赋值，必须找到 A[k]与 B[j]的存储地址。从 A[k]的存储地址取到 A[k]的值，传送入 x，而把 y 的值传送入 B[j]的存储地址所指明的存储字中。数组元素的地址由相对于为数组分配的存储区域的首地址的位移量 d 来确定，因此，绝对地址与寄存器两种寻址方式是不够的，必须应用变址和间接寻址方式。例如 A[k]，首地址 A 的值是不变的，可把由 k 确定的位移量存放在寄存器 R_k 中，$A(R_k)$确定的便是 A[k]的存储地址。对于赋值语句 x=A[k]的四元式：

```
(1)  =[ ]   A   k   t1
(2)  :=     t1      x
```

相应的目标代码将是：

```
MOV  A(Rk),   R
MOV   R,      x
```

其中假定了 k 的值在寄存器 R_k 中。如果 k 的值在存储字 M_k 中，则需有从 M_k 传输到 R_k 的目标指令：

```
MOV      Mk,    Rk
```

对于赋值语句 B[j]=y 的四元式：

```
[ ]=    B   j   t1
&:=     y       t1
```

应先把 B[j]的存储地址传送入一个寄存器 R，然后按间接寻址方式，把 y 的值传送入以 R 的值为地址的存储字中。因此，相应的目标代码是：

```
ADDR  B(Rj),   R
MOV    y,      *R
```

其中 j 的值在寄存器 R_j 中，且引进了取地址指令：

```
ADDR   d,    R
```

其功能是把 d 的存储地址传送入寄存器 R 中。

对于数组元素，需要计算相对于数组存储区域首地址的位移量，一是还需要考虑数组元素的大小，二是要考虑二维或更高维数组的情况。

例如设 A 为一维整型数组，B 为一维实型数组。对于 x=A[k]的四元式序列为：

```
 *     k    2    t1
=[ ]   A    t1   t2
:=     t2        x
```

其中，2 是整型数据所占存储字节数，相应的目标代码为：

```
MOV   k,       R0
MPY   #2,      R0
MOV   A(R0),   R1
MOV   R1,      x
```

对于 B[j] = y 的四元式序列为：

```
*     j     4     t1
[ ]=  B     t1    t2
&:=   y           t2
```

其中，4 是实型数据所占存储字节数，相应的目标代码为：

```
MOV   j,       R0
MPY   #4,      R0
ADDR  B(R0),   R1
MOV   y,       *R1
```

下面考虑二维数组情况。假定有说明语句：

```
int  C[3][4];
```

则 z=C[j][k]的四元式序列如下：

```
*     j     4     t1          4 是数组第二维的元素个数
+     t1    k     t2
*     t2    2     t3          2 是 int 型数据所占存储字节数
=[ ]  C     t3    t4
:=    t4          z
```

其中 t3 的值是(j×4+k)×sizeof(int)，即相对于 C 的存储区域首地址的位移量。相应的目标代码是：

```
(1) MOV   j,    R0          (4) MPY   #2,     R0
(2) MPY   #4,   R0          (5) MOV   C(R0),  R1
(3) ADD   k,    R0          (6) MOV   R1,     z
```

7.2.4　其他的中间表示代码

除了四元式序列这一种中间表示代码外，还有三元式序列、抽象语法树与逆波兰表示等中间表示代码，下面作简单介绍。

1. 三元式序列

三元式序列类似于四元式序列，其区别在于：三元式中不包含运算结果，即每个三元式呈以下形式：

运算符　运算分量　运算分量

其中，运算分量除了是变量与常量外，还可能是三元式序号，例如，对于赋值语句 x=(a+b)/c-d*e 的相应三元式序列如下：

	运算符	运算分量	运算分量
(1)	+	a	b
(2)	/	(1)	c

```
(3)      *          d          e
(4)      -          (2)        (3)
(5)      :=         (4)        x
```

其中，运算分量（1）、（2）与（3）等是三元式的序号。由于没有运算结果的位置，赋值三元式如上面的三元式（5）所示，把三元式（4）中计算的值赋给 x。三元式与四元式的其他区别，主要是控制转移三元式，以下列例子说明。

可为下列 if 语句：

```
if(a>b && c<d || e==f) x=m; else x=n;
```

生成下列三元式序列：

```
(1)  >    a    b          (7)  GOF  (10)   (6)
(2)  GOF  (6)  (1)        (8)   :=    m     x
(3)  <    c    d          (9)  GO   (11)
(4)  GOF  (6)  (3)        (10)  :=   n      x
(5)  GO   (8)             (11)
(6)  =    e    f
```

可见三元式与四元式在表示法上的区别主要是两点：

- 运算分量位置上可以出现三元式序号；
- 有按假转控制转移三元式 GOF，当第二个运算分量的值为 false 时，控制转向由第一个运算分量指明的三元式。

可以类似于四元式序列，设计生成三元式序列的翻译方案，并从三元式序列生成目标代码。

2. 逆波兰表示

通常，一个表达式中，双目运算符总是出现在两个运算分量的中间，称为中缀表示。逆波兰表示是双目运算符出现在两个运算分量后面的一种表示法，它的特点是无括号，例如，a+b 的逆波兰表示是 ab+，而(a−b)*c+d/(e+f)的逆波兰表示是 ab−c*def+/+。

对于逆波兰表示来说，重要的是把这种表示法扩充到语言各种语法成分，可如下地进行扩充。

对于赋值语句 v=e，它的逆波兰表示是：v'e'=，其中，v'与 e'分别是 v 与 e 的逆波兰表示。例如，x=(a+b)*c 的逆波兰表示是 ab+c*x=。

对于 if 语句：if(E) S_1 else S_2，它的逆波兰表示是：E' N_1 GOF S_1' N_2 GO S_2'

其中，E'、S_1'与 S_2'分别是 E、S_1 与 S_2 的逆波兰表示，而 N_1 与 N_2 是逆波兰表示中符号的序号。GOF 与 GO 是单目运算符，N_1 GOF 是逆波兰表示，即按假转到序号为 N_1 的符号位置处。N_2 GO 是无条件控制转移到序号为 N_2 的符号位置处。例如：

```
if(a>b) max=a; else max=b;
```

的逆波兰表示是：

```
a b > 11 GOF max a = 14 GO max b =
```

对于其他各种控制结构，如 while 语句等可以根据其语义，或按其规范展式，写出相应的逆波兰表示。请读者自行完成。

在第 4 章递归下降分析技术一节中，已讨论过由递归下降识别程序生成逆波兰表示，在第 6 章 6.1.3 节也讨论了设计从中缀表达式生成后缀表达式，即逆波兰表示的翻译方案。

从逆波兰表示生成目标代码是十分方便的。例如，通常的中缀表达式

```
x=(a+b* (c-6))/d+e
```

有如下的逆波兰表示：

```
x a b c 6 - * + d / e + =
```

从这逆波兰表示生成目标代码，基本处理思想如下：

设置一个运算分量栈，用来存放还不能生成目标指令的运算分量。

初始时，运算分量栈为空。然后从左到右逐个符号地扫描逆波兰表示，如果是运算分量，把它下推入运算分量栈，如果是运算符，则根据运算符是几目运算符，从运算分量栈取出相应个数的运算分量，生成目标指令，这时把所取运算分量从运算分量栈退去，并把存放相应运算结果的寄存器下推入运算分量栈，再继续扫描逆波兰表示，直到处理完整个逆波兰表示。

对于上述逆波兰表示生成目标指令的过程如图 7-1 所示。

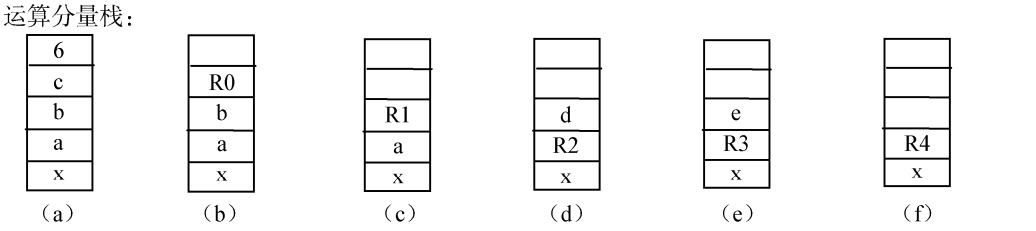


图 7-1

当扫描到运算符−、*与+时，运算分量栈分别如图 7-1（a）、（b）与（c）所示。在生成运算符+的目标指令后，共生成 6 个目标指令。当扫描到 d 时，把它下推入运算分量栈，如图 7-1（d）所示。继续扫描到运算符+时，运算分量栈如图 7-1（e）所示，这时又生成 2 个目标指令：

```
MOV  R3,  R4
ADD  e,   R4
```

最终扫描到赋值运算符=时，运算分量栈中仅 x 与 R4 两个元素，如图 7-1（f）所示，这时生成传输指令：

```
MOV  R4,  x
```

总计生成 11 个目标指令。上退去栈顶元素 R4，运算分量栈仅有作为最终结果的 x。

通常引进运算符栈，利用它从中缀表达式生成逆波兰表示，具体过程大致如下。

初始时，把左端标志符号#下推入运算符栈，#看作优先级最低的运算符。从左到右扫描中缀表达式，如果是运算分量时，直接输出，如果是运算符时，把它与运算符栈顶的运算符进行优先级比较。如果当前运算符优先级高，把它下推入运算符栈，如果运算符栈顶运算符的优先级高，则把运算符从运算符栈退去，并输出。继续把当前运算符与运算符栈顶运算符比较优先级，直到当前运算符优先级高，把它下推入运算符栈。如此继续，直到扫描完整个中缀表达式，这时整个输出便是所求的逆波兰表示。

把运算分量栈与运算符栈结合起来，可以从中缀表达式直接生成目标指令。

请读者自行用实例实践上述构造法。事实上，当从表达式扩充到语言各个语法成分时，这是一种分析技术，即算符优先分析技术。

3. 抽象语法树

先前讨论的语法分析树与具体的文法紧密相关。这里所谓的抽象语法树，是与程序设计语言的具体表示无关的一种表示。

例如，PASCAL 语言中，if 语句的书写规则是：

```
if 语句::=if 表达式 then 语句 else 语句
```

而 C 语言 if 语句的书写规则是：

```
if 语句::=if（表达式）语句 else 语句
```

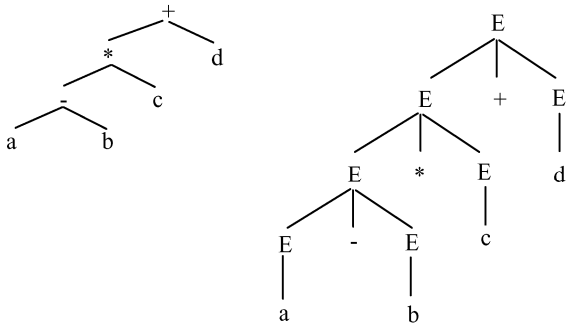
事实上，if 语句的本质部分是 3 个，即

```
表达式  语句  语句
```

因此如下地定义 if 语句：

```
if 语句  表达式  语句  语句
```

这就是抽象语法规则。所有的抽象语法规则构成抽象语法，相应的语法分析树称为抽象语法树。例如，表达式(a−b)*c+d 的抽象语法树将如图 7-2（a）所示。



（a）　　（b）

图 7-2

显然，它比如图 7-2（b）所示的通常的语法分析树结点数少得多。如果引进非终结符号 T 与 F 等，语法分析树上的结点比相应抽象语法树的还将更多。

如何设计抽象语法树的计算机存储表示，并应用抽象语法树作为中间表示代码生成目标程序的问题，不多加讨论。有兴趣的读者可以参考相关的资料。

7.3　基本块的代码优化

7.3.1　基本块优化的种类

基本块的优化是局限于一个基本块内的优化，是与循环无关的局部优化，包括下列几类优化：合并常量计算、消除公共子表达式、削减计算强度与删除无用代码等。读者理解了各类优化的思路，就能熟练应用。

1．合并常量计算

合并常量计算的思路是：在编译时刻能进行的计算就在编译时刻完成。最典型的例子是计算圆周长的公式 l=2πr。这时，一般写出下列赋值语句：

```
l=2*3.1416*r;
```

如果利用符号常量，例如由宏定义引进 pi：

```
#define  pi   3.1416
```

写出：　l=2*pi*r;

显然，不论哪种情况，2π 的值可以在编译时刻事先计算好。因此，可以由编译程序优化为下

列语句：

```
l=6.2832*r;
```

记住：代码优化是对四元式序列进行的，因此下列四元式序列：

```
(1)  *   2   3.1416   t1        (1)  *   2   pi   t1
(2)  *   t1    r      t2   或   (2)  *   t1   r    t2
(3)  :=  t2            l        (3)  :=  t2        l
```

被优化成：

```
(1)  *   6.2832   r   t2
(2)  :=  t2            l
```

由于在编译时刻已计算好 2 与 3.1416 的乘积，运行时刻无需再计算，改进了目标代码质量，提高了功效。

一个表达式，当它全由常量组成，可以在编译时刻计算出它的值，且用此值去代替这个表达式，这种优化称为**合并常量计算**优化。注意如果是如下的四元式：

```
*   2   r       t1
*   t1  3.1416  t2
```

是不能合并常量计算的，只有当一个四元式的两个运算分量都是常量或计算出的常量值时，才能进行合并常量计算优化。另外请注意包含函数调用的情况，如 2*sqrt(2)，不属于合并常量计算优化范围。

概括起来，合并常量计算优化，是在编译时刻把已可计算的常量表达式的值计算好，以避免在运行时刻去计算。

2. 消除公共子表达式

消除公共子表达式优化的思路是：如果一个表达式在它出现之前已计算过，那么就用以前所计算的值代替这个表达式。当然要能代替，必须该表达式中涉及的变量的值都未改变，看下面例子。这种后面出现的、可被代替的表达式称为**公共子表达式**。

【例 7.5】　设有表达式：

```
x*y+z+2* (x*y+z)/(x*y)
```

其中 x*y 出现 3 次，x*y+z 出现 2 次。显然，由于 x、y 与 z 的值都未改变，第一次出现时计算所得的值可以去代替后面的出现。现在从四元式序列出发，考虑消除公共子表达式的优化。

```
(1)  *   x   y   t1          (5)  *   2   t4   t5
(2)  +   t1  z   t2          (6)  *   x   y    t6
(3)  *   x   y   t3          (7)  /   t5  t6   t7
(4)  +   t3  z   t4          (8)  +   t2  t7   t8
```

（3）中的 x*y 显然是公共子表达式，因为（1）中已计算过，且其中的 x 与 y 值未被改变，因此无需执行（3），可用 t1 代替 t3，（4）成为：

```
(4')  +   t1   z   t4
```

这时（4'）中的表达式 t1+z，已在（2）中计算过，且 t1 与 z 值未被改变，因此也是公共子表达式，可不必执行（4'），且用 t2 代替 t4。四元式（6）的情况类似，无需执行，可用 t1 代替 t6，把（7）改写成：

```
(7')  /   t5   t1   t7
```

最终原有基本块优化成如下的基本块：

```
(1)  *     x     y     t1
(2)  +     t1    z     t2
(3)  *     2     t2    t5
(4)  /     t5    t1    t7
(5)  +     t2    t7    t8
```

从 8 个四元式减少到 5 个四元式，其功效提高是明显的。

注意，消除公共子表达式优化，同样是对四元式序列进行的优化。四元式序列是在语义分析后生成的，像(x*y+z+1)*u–(2*x*y+z)/v 这样的表达式不可能进行消除公共子表达式的优化。请读者自行写出相应的四元式序列来验证。另外，也请注意：公共子表达式必须是第二次或以后的出现，且其中的变量都不被改变值。例如下列基本块：

```
(1)  +   a   b   c
(2)  –   c   d   b
(3)  +   a   b   e
(4)  –   c   d   f
```

其中，（4）中的表达式 c–d 是公共子表达式，但（3）中的表达式 a+b 不是，因为其中的 b 在（2）中被改变了值。

概括起来，消除公共子表达式优化，是把表达式中的一切子表达式，用先前所计算的值代替而不再重复计算，从而实现优化的目的。

3. 削减计算强度

削减计算强度优化的思路是利用代数恒等变换，用运算速度快的计算代替运算速度慢的计算。例如，用加法代替乘法，用移位代替除法，以及用乘法代替乘幂等，因为乘法速度快于乘幂，加法速度快于乘法，等等。由于必须是恒等变换，并非任何的情况都可以进行计算强度削减，事实上仅如下的少数几种情况：

- 用 x×x 代替 x^2；
- 用 x+x 代替 2x；
- 用 x×0.5 代替 x/2 等。

例如，求解一元二次方程式 $ax^2+bx+c=0$ 时的一个解：

$$x_1=\frac{-b+\sqrt{b^2-4ac}}{2a}$$

用**表示乘幂运算符时可以写出下列赋值语句：

```
x1=(–b+sqrt(b**2–4*a*c))/(2*a);
```

相应的四元式序列如下：

```
(1) NEG    b        t1          (7) CALL   sqrt   1    t6
(2)  **    b    2   t2          (8)  +     t1     t6   t7
(3)  *     4    a   t3          (9)  *     2      a    t8
(4)  *     t3   c   t4          (10) /     t7     t8   t9
(5)  –     t2   t4  t5          (11)  :=   t9          x1
(6) PARAM  t5
```

当应用削减计算强度优化后的四元式序列如下：

```
(1) NEG    b        t1          (7) CALL   sqrt   1    t6
(2)  *     b    b   t2          (8)  +     t1     t6   t7
(3)  *     4    a   t3          (9)  +     a      a    t8
(4)  *     t3   c   t4          (10) /     t7     t8   t9
(5)  -     t2   t4  t5          (11) :=    t9          x1
(6) PARAM  t5
```

该四元式序列相当于下列赋值语句：

```
x1=(–b+sqrt(b*b–4*a*c))/(a+a);
```

概括起来，削减计算强度优化是由编译程序利用代数恒等变换，把运算速度较快的计算去代替运算速度较慢的计算。

4．删除无用代码

删除无用代码优化的思路是：把四元式序列中，对此后的计算不发生任何影响的四元式删除。一般地说，程序中所包含的语句都是为实现某功能而写出的，如果不起作用，是多余的，而且容易出错。例如，

```
x=0;   y=1;   x+y;   y=x+2;
```

其中的语句 y=1;与 x+y;显然不起作用，都是多余的，是无用代码，应删除。要查出无用代码是困难的，本节中的删除无用代码指的是：所计算的值决不被引用的四元式，特别是在进行了消除公共子表达式优化后产生的复写四元式。下面从例 7.5 中的四元式序列看复写四元式是如何产生的。

通常，从例 7.5 中的四元式序列，进行消除公共子表达式优化后，得到的四元式序列如下：

```
(1)  *    x    y    t1          (5)  :=   t2        t4
(2)  :=   t1        t3          (6)  *    2    t2   t5
(3)  :=   t1        t6          (7)  /    t5   t1   t7
(4)  +    t1   z    t2          (8)  +    t2   t7   t8
```

其中，有两种情况：一是把 t1 的值赋值给 t3 与 t6，把 t2 的值赋值给 t4，二是用 t1 去代替 t3 与 t6，用 t2 去代替 t4。（2）、（3）与（5）中的赋值四元式称为复写四元式，而用 t1 代替 t3 与 t6 等称为复写传播。

当进行了复写传播之后，被复写的临时变量 t3 与 t4 等将不再被引用，相应的复写四元式便成为无用代码而被删除。例 7.5 中最后所得的基本块，事实上是进行消除公共子表达式优化，并进行删除无用代码优化之后的结果。

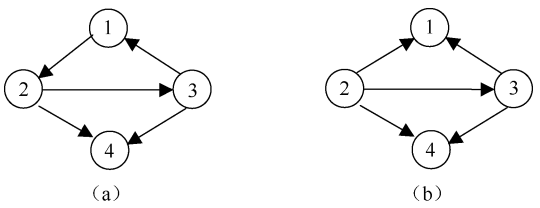
概括起来，删除无用代码优化主要是指删除由复写传播变换而形成的复写四元式等无用代码。

7.3.2　基本块优化的实现

关于基本块的优化，包括合并常量计算、消除公共子表达式、削减计算强度与删除无用代码。为了实现基本块的优化，引进所谓的无环路有向图（Directed Acyclic Graphic），缩写为 dag。引进 dag 的目的是消除公共子表达式。当从基本块四元式序列构造了 dag，再从 dag 还原到四元式序列时，便进行了删除公共子表达式的优化。也可以利用 dag 来进行合并常量计算与削减计算强度等优化。

1．基本块的 dag 的构造法

有向图是一种由结点和有向边组成的图形表示，dag 是不包含任何环路的有向图。例如图 7-3（a）中的有向图不是 dag，因为其中有环路：①→②→③→①。图 7-3（b）中的有向图是 dag，其中不包含环路。



（a）　　（b）

图 7-3

为基本块构造的 **dag** 是结点上将有下列标记的 dag：

- 叶结点用变量名（标识符）或常数作为其唯一的标记，当标记是标识符时，代表变量的初值，给它加下标 0；

- 内部结点用运算符标记，它表示所计算的值；
- 各结点可能附加一个或若干个标识符，附加于同一个结点上的若干个标识符，相应的变量具有相同的值。

因此，任何结点都有标记，对于叶结点，用标识符或常数作为唯一标记，内结点，用运算符作为标记，且各个结点可能有一个附加标识符表，同一个结点的附加标识符表中的各个标识符有相同的值。

【例 7.6】 基本块 dag 的例子。设有基本块四元式序列如下：

```
(1)  +   a   b   c
(2)  -   c   d   b
(3)  +   a   b   e
(4)  -   c   d   f
```

为其构造的 dag 如图 7-4（a）所示。

图 7-4（b）是图 7-4（a）中 dag 的结点信息表，也可以说是 dag 的存储表示示意图，它记录了 dag 中结点的建立顺序及结点之间的相互联系。当从 dag 还原成四元式序列时，需要利用此结点信息表来得到正确的原有顺序。

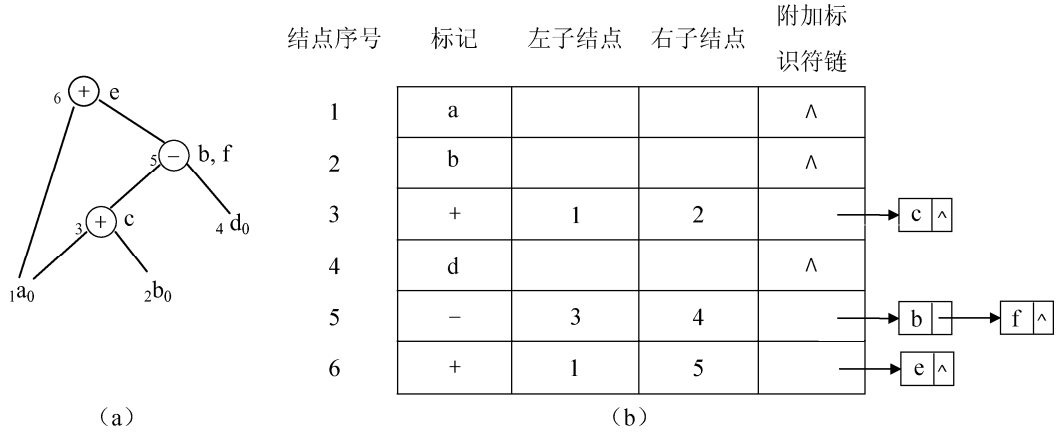


图 7-4

虽然是有向图，但图中画出的并不是有向边，事实上有一个隐含的约定：从下向上画边与结点，因此箭头总是朝上的。

【例 7.7】 为例 7.5 中的表达式 x*y+z+2*(x*y+z)/(x*y)的四元式序列构造 dag。

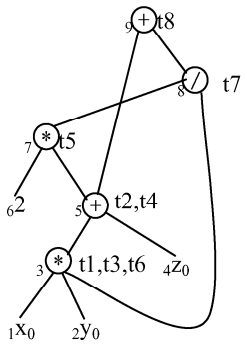
先写出其四元式序列如下：

```
(1)  *   x   y   t1          (5)  *   2   t4   t5
(2)  +   t1  z   t2          (6)  *   x   y    t6
(3)  *   x   y   t3          (7)  /   t5  t6   t7
(4)  +   t3  z   t4          (8)  +   t2  t7   t8
```

这是一个基本块，为它构造的 dag 如图 7-5（a）所示。

dag 为什么能进行消除公共子表达式的优化，考察 dag 的构造过程。其思路如下：顺次逐个四元式地进行分析，对于四元式 op x y z，寻找这样的内结点，它的标记为 op，左右子结点分别为 x 与 y 的当前值所在结点。如果找到，则为该结点添加附加标识符 z，且该结点为 z 的当前值所在结点；否则，根据具体情况分别进行处理。为记住一个变量的当前值所在结点，引进函

数 Node，Node(x)的值就是具有标识符 x 当前值的结点的序号，此序号可记录在该标识符的符号表相应条目中，以便取得当前值或相应结点的序号。



（a）

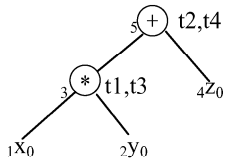
结点序号	标记	左子结点	右子结点	附加标识符链
1	x_0			^
2	y_0			^
3	*	1	2	→ t1 → t3 → t6 ^
4	z_0			^
5	+	3	4	→ t2 → t4 ^
6	2			^
7	*	6	5	→ t5 ^
8	/	7	3	→ t7 ^
9	+	5	8	→ t8 ^

（b）

图 7-5

现在考虑如何构造 dag。四元式（1）是第一个四元式，尚未建立任何结点，因此为 x 与 y 建立结点，设序号分别为 N1 和 N2，标记分别为 x_0 与 y_0，显然 Node(x)=N1 与 Node(y)=N2。分别以 N1 与 N2 作为左右子结点，建立父结点 N3，它以运算符*作为标记，并且以 t1 为附加标识符，因此 Node(t1)=N3，t1 的当前值在结点 N3。对于四元式（2），寻找这样的内结点，它以运算符＋为标记，分别以 Node(t1)与 Node(z)为左右子结点。由于仅存在结点 Node(t1)，建立结点 N4，以 z 为标记，Node(z)=N4，并建立结点 N5，它以运算符+为标记，且分别以 Node(t1)与 N4 为左右子结点，它以 t2 为附加标识符，Node(t2)=N5。

现在考察四元式（3），类似地寻找以*为标记，分别以 Node(x)与 Node(y)为其左右子结点的结点。容易发现这正是结点 N3，无需建立新结点，仅把标识符 t3 添加入结点 N3 的附加标识符表中，所以 Node(t3)=N3。类似地，对于四元式（4），可找到以+为标记，分别以 Node(t3)与 Node(z)为左右子结点的结点 N5，因此不建立新结点，仅把标识符 t4 添加入 Node(t2)=N5 的附加标识符表中，Node(t4)=N5。这时的部分 dag 与结点信息表如图 7-6 所示。



（a）

结点序号	标记	左子结点	右子结点	附加标识符链
1	x_0			^
2	y_0			^
3	*	1	2	→ t1 → t3 ^
4	z_0			^
5	+	3	4	→ t2 → t4 ^

（b）

图 7-6

继续顺次逐个分析该基本块中的其他四元式，最终将建立如图 7-5 中所示的 dag 及结点信息表。

在一般情况下，构造 dag 的过程大致如下。

初始时无任何结点，对于任何标识符，Node 都是无定义的。顺次逐个地对基本块中的每个四元式，如下进行分析处理并建立 dag。

对于当前四元式 op　x　y　z，寻找 Node(x)，如果还没定义，则为 x 建立结点（叶结点），标记为 x，且让 Node(x)=当前结点序号 N。然后再寻找 Node(y)，未找到时，类似地建立新结点，Node(y)=当前结点序号 N。再寻找以 op 为标记，分别以 Node(x)与 Node(y)为左右子结点的结点。如果找不到，则建立新结点，它以 op 为标记，分别以 Node(x)与 Node(y)为左右子结点。让找到的或新建立的结点的序号统一为 N，从原有 Node(z)的附加标识符表中删除标识符 z，把 z 添加到结点 N 的附加标识符表中，且让 Node(z)=N。这种双目运算符四元式的处理便完成了。

对于单目运算四元式 op　x　　z，情况类似，区别仅在于无需寻找第 2 个运算分量 y 的 Node(y)。对于形如:=　x　　z 的赋值四元式，由于 z 与 x 将有相同的值，只需把 z 添加入 N_x=Node(x)的附加标识符表中，并置 Node(z)=N_x。

2. 从 dag 重建四元式序列

为基本块四元式序列建立了 dag 后，可以从 dag 重新生成基本块四元式序列。这时根据 dag 中结点生成的顺序，也可以说，按结点信息表，依次逐个结点地分析，还原成四元式序列，还原规则如下。

规则 1　若是叶结点，且附加标识符表为空，则不生成四元式。

规则 2　若是叶结点，标记为 x，且附加标识符为 z，则生成四元式:= x　　z。

规则 3　若是内结点，附加标识符是 z，则根据其标记 op 及子结点个数生成相应的四元式。假定 x 与 y 是子结点的标记（叶结点时）或附加标识符（内结点时），

- 标记 op，不是=[ ]或[ ]=，也不是 relop，有左右子结点，则生成下列四元式：

```
op  x  y  z
```

- 标记 op 是=[ ]或[ ]=，则生成下列四元式：

```
=[ ]  x  y  z 或  [ ]=  x  y  z
```

- 标记 op 是 relop，则生成下列四元式：

```
relop  x  y  z
```

其中 z 是四元式序号。原有四元式序号可能已被改变，此四元式序号应作相应改变。

- 若内结点只有一个子结点，则生成下列四元式：

```
op  x    z
```

规则 4　若是内结点，但无附加标识符，则为该结点添加一个局部于本基本块的临时性附加标识符，然后按规则 3 生成相应四元式。

规则 5　结点的附加标识符表中包含多个标识符 z_1、z_2、…、z_m 时，

- 若结点是叶结点，标记是 z，则生成如下的一列赋值四元式：

```
:=   z      z1
:=   z      z2
…
:=   z      zm
```

- 若结点是内结点，附加标识符表中第一个标识符是 z_1，则生成如下的一列赋值四元式：

```
:=   z1     z2
:=   z1     z3
…
:=   z1     zm
```

当应用上述规则于例 7.7 中生成的 dag 时，可重新生成如下的四元式序列：

```
(1)  *   x    y    t1          (5)  :=   t2        t4
(2)  :=  t1       t3          (6)  *    2    t2   t5
(3)  :=  t1       t6          (7)  /    t5   t1   t7
(4)  +   t1   z   t2          (8)  +    t2   t7   t8
```

该例中已进行消除公共子表达式优化，由于引进复写四元式并进行复写传播，还可以进行删除无用代码的优化，可消除四元式（2）、（3）与（5）。

如果比较例 7.6 中的基本块四元式序列，不难发现为四元式（3）建立结点时，由于其中的 b 的当前值所在结点序号为 Node(b)=5，而不是 Node(b_0)=2，将为 a+b 建立结点 6，在从 dag 重新生成四元式序列时，不会为四元式（3）生成四元式：

```
:=  c       e
```

表明四元式（3）中的 a+b 不是公共子表达式。

显然，通过这两个例子可以领会到 dag 为什么能进行消除公共子表达式的优化。

3. 应用 dag 于基本块的其他优化

应用 dag 除了可以进行消除公共子表达式的优化及删除无用代码优化外，还可以进行合并计算常量与计算强度削减两类优化。

（1）合并常量计算优化

当在构造 dag 的过程中，发现一个以 op 为标记的结点，其左右子结点上的标记 x 与 y 都是给定的常量或计算出的常量值时，可以进行合并常量计算优化，这时只需计算出 x op y 的值，把此结点更改为叶结点，且把原标记改为 x op y 的值。例如，如图 7-7 所示，下列两个四元式可优化为一个四元式：

```
*    2    3.1416    t1          =>   *    6.2832    r    t2
*    t1   r         t2
```

（2）削减计算强度优化

这类优化基于代数恒等式进行等价变换，用运算速度快的计算代替运算速度慢的计算。类似于合并常量计算优化，只需在构造 dag 过程中识别出可以进行削减计算强度优化的情况，便可进行这类优化。例如，识别出内结点的标记 op 是*，而两个子结点中有一个的标记是常量 2，便可把 2*x 优化成 x+x。其他情况类似。图 7-8 展示了用加法代替乘法的削减计算强度优化时的 dag 的变化情况。

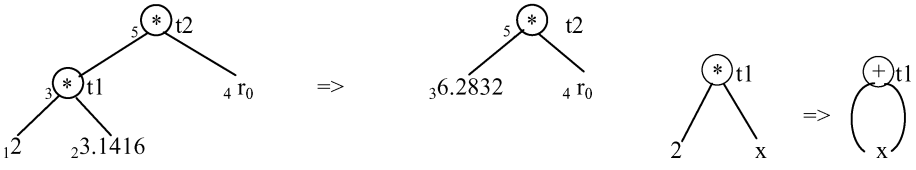


图 7-7　　图 7-8

从 dag 的构造过程可见，除了可进行消除公共子表达式、削减计算强度、合并常量计算与删除无用代码等优化外，还可以获得这样一些四元式的信息：对于它们在本基本块内所计算的值，在基本块外仍可引用。

先看一个简单的例子：

```
(1)  +   a   b   x
(2)  —   a   b   x
```

假定对于四元式（1），建立标记为加号+的结点，序号为 N1，其附加标识符表为{x}，Node(x)=N1。对于四元式（2），建立标记为减号—的结点，序号为 N2，这时从结点 Node(x)的附加标识符表中删除标识符 x，把 x 添入结点 N2 的附加标识符表中，且让 Node(x)=N2。显然四元式（1）处计算的值已不能在该基本块外被引用，结点 N1 的附加标识符表为空集。一般情况下，为某四元式 op　x　y　z 构造的结点是 N，则有 Node(z)=N，如果在整个基本块的 dag 构造结束时，Node(z)仍然是 N，即 z 在结点 N 的附加标识符表中，则该四元式所计算之值在本基本块外才是可以被引用的。

概括起来，通过构造基本块的 dag，可以进行下列优化：

- 消除公共子表达式；
- 合并常量计算；
- 削减计算强度；
- 删除无用代码。

还可以获得可用于全局优化的下列信息：

- 本基本块内引用其值的标识符；
- 在本基本块外能引用它所计算的值的四元式。

因此，dag 确实是实现基本块优化的有效工具。

只是读者要注意数组元素赋值的处理，必须保证优化前后的四元式序列是等价的，否则将造成错误。试看下列基本块的情况：

```
(1)  =[ ]   A   j   t1
(2)  [ ]=   A   k   t2
(3)  &:=    y       t2
(4)  =[ ]   A   j   t3
```

为其构造 dag。由于四元式（4）中的 A 与 j 均未被改变，按构造 dag 的规则，将识别出（4）中的是公共子表达式，从 dag 重新生成四元式序列时，将生成：

```
(1)  =[ ]   A   j   t1
(2)  :=     t1      t3
(3)  [ ]=   A   k   t2
(4)  &:=    y       t2
```

这看起来是对的，但当运行时，若 j=k，但 y≠A[j]，则 t3 之值在“优化”前后的计算结果将不同，“优化”前是 y，而“优化”后是 A[j]。优化前后必须保持等价，不等价的便不成其为优化。因此对 op 为=[ ]的四元式生成结点时应补充规则。请读者自行思考。

7.4　与循环有关的优化

一个程序运行的功效，往往为程序中仅占长度很少比例的关键部分所左右。这个关键部分就是循环。循环仅占整个程序长度的很小比例，然而运行时间却可占整个运行时间的很大部分，因此对循环的优化是提高程序功效的关键。减少循环结构内的指令数，即使增加了循环外的指令数，也会使程序的运行时间大大减少。

7.4.1　循环优化的种类

对循环的优化包括：循环不变表达式外提、归纳变量删除与计算强度削减，而计算强度削减

又包括归纳变量计算强度削减与数组元素地址计算强度削减。

1．循环不变表达式外提

如果在循环内有一个表达式，不论重复执行多少次循环体，每次重复，它的值都不变，这样的表达式便是**循环不变表达式**。不言而喻，这种表达式的值只需计算一次。循环不变表达式外提优化的思路便是：把这种表达式外提到循环外计算，然后在循环体内引用所计算的这个值。

【例 7.8】 假定要打印一组三角形的面积值，条件是给定三角形的两边 a 与 b 的长度，夹角 C 则是 10°、20°、…，直到 180°。三角形面积计算公式为 S=½a×b×sin(C)。为此可写出 C 程序片段如下：

```
for (k=1; k<=18; k=k+1)
{  S=0.5*a*b*sin(10*(pi/180)*k);
   printf("a=%f,b=%f,夹角是%d°时三角形面积=%8.2f\n", a, b, k*10, S);
}
```

其中，pi 是常量 3.1416。上述源程序中循环内，0.5*a*b 与 10*(pi/180)的值，显然不论是哪次重复执行循环体，总是相同的，因此可以把对它的计算外提到循环前：

```
c1=0.5*a*b;  c2=10* (pi/180);
for (k=1; k<=18; k=k+1)
{  S=c1*sin(c2*k);
   printf("a=%f,b=%f,夹角是%d°时三角形面积=%8.2f\n",a,b,k*10, S);
}
```

这样，对 0.5*a*b 与 10*(pi/180)的计算各从 18 次减少到 1 次。如果说，让 a 和 b 值也变化，例如：

```
for(i=1; i<=m; i=i+1)
{  for(j=1; j<=n; j=j+1)
   { c1=0.5*a[i]*b[j];  c2=10* (pi/180);
     for(k=1; k<=18; k=k+1)
      {  S=c1*sin(c2*k);
         printf("a=%f,b=%f,夹角是%d°时三角形面积=%8.2f\n",a[i],b[j],k*10, S);
      }
   }
}
```

应用循环不变表达式外提优化的思想，又可改写成：

```
c2=10* (pi/180);
for (i=1; i<=m; i=i+1)
{ c1i=0.5*a[i];
  for (j=1; j<=n; j=j+1)
  { c1=c1i*b[j];
    for(k=1; k<=18; k=k+1)
    {  S=c1*sin(c2*k);
       printf("a=%f,b=%f,夹角是%d°时三角形面积=%8.2f\n",a[i],b[j],k*10, S);
    }
  }
}
```

功效的提高更为明显：k 循环中每个语句的执行次数是 18mn。10*(pi/180)的计算次数从 18mn 次减少到了 1 次，0.5*a[i]的计算次数从 18mn 次减少到了 m 次，而 c1i*b[j]的计算次数从 18mn 次减少了到 mn 次。

这是循环不变表达式外提优化吗？请注意，这不是！这里的优化都是指的编译时刻对四元式序列进行的改进。上述讨论仅是从源程序级来理解循环不变表达式外提优化的思想。

首先，必须基于关于语义分析的翻译方案得出四元式序列，在四元式序列中识别出循环不变表达式，从而进行外提的优化。

尽管前面没有给出关于 while 语句生成中间表示代码的翻译方案，但联系到目标代码生成的翻译方案，可以看到，while 语句的关键字 while 已不再出现，while 语句代之以由 if 语句和 goto 语句的组合。从源程序级看，while 语句：while(E)S 展开成了下列等价结构：

```
Loop: if (E)
      { S;  goto  Loop; }
```

一般地，do-while 语句：do S while(E); 展开成了下列等价结构：

```
Loop: S
      if (E) goto Loop;
```

对于 for 语句：for(E1;E2;E3) S，类似地它按规范方式展开成：

```
      E1;
  Loop: if(!E2) goto Finish;
      S
      E3;
      goto Loop;
  Finish:
```

可见，E1 仅计算一次，E2 与 E3 每次重复都需计算。

关于固定的 a 与 b 的值，计算三角形面积的上述程序可规范展开如下：

```
        k=1;
  Loop: if(!(k<=18)) goto Finish;
        S=0.5*a*b*sin (10*(pi/180)*k);
        printf("a=%f,b=%f,夹角是%d°时三角形面积=%8.2f\n", a,b,k*10, S);
        k=k+1;
        goto Loop;
 Finish:
```

可写出相应的四元式序列如下：

```
(1)  :=       1       k            (13) :=      t7           S
(2)  <=       k   18  (5)          (14) *       k       10   t8
(3)  GO       (4)                  (15) PARAM   "   a=%f,    b=%f,…\n"
(4)  GO       (24)                 (16) PARAM   a
(5)  *        0.5 a   t1           (17) PARAM   b
(6)  *        t1  b   t2           (18) PARAM   t8
(7)  /        pi  180 t3           (19) PARAM   S
(8)  *        10  t3  t4           (20) CALL    printf  5
(9)  *        t4  k   t5           (21) +       k       1    t9
(10) PARAM    t5                   (22) :=      t9           k
(11) CALL     sin 1   t6           (23) GO      (2)
(12) *        t2  t6  t7           (24)
```

易见，四元式（5）、（6）、（7）与（8）计算的是循环不变表达式，可外提到循环之外，而且可对四元式（7）进行合并常量计算优化。

注意，如果把计算 S 的表达式改写成：

```
S=a*b/2*sin(k*10* (pi/180));
```

还能进行循环不变表达式外提优化吗?

概括起来，循环不变表达式外提优化是：把循环中不论哪次重复循环体，值都不变的表达式（循环不变表达式）外提到循环外，以减少计算次数。

一般地说，进行循环不变表达式外提的优化必须解决如下几个问题：

- 如何识别循环中的不变表达式?

- 把循环不变表达式外提到何处?
- 什么条件下可以外提循环不变表达式?

这些问题将在后面讨论。

2. 归纳变量删除

为了解什么是归纳变量，先看例子。

【例 7.9】　假定要计算首项为 3 与等差为 3 的等差数列前 30 项之和 S。不难写出如下的 C 程序：

```
S=0;
for (k=1; k<=30; k=k+1)
{  j=3*k;   S=S+j;   }
```

一般地设计中间表示代码生成的翻译方案,将使其规范展开成相当于下列程序的四元式序列：

```
      S=0;  k=1;
Loop: if(!(k<=30)) goto Finish;
      j=3*k;
      S=S+j;
      k=k+1;
      goto Loop;
Finish:
```

考察其中的变量 j 与 k，变量 k 的值从 1 开始，每次重复执行循环体，k 值增加 1，直到 30 为止；相应地，变量 j 的值，每次重复执行循环体，j 值增加 3。这表明变量 j 和 k 的共同特点是：每次重复执行循环体，它们都增加一个常量值。这种在循环中每次重复执行循环体，都增加或减少某个固定常量值的变量称为**循环的归纳变量**。因此，j 和 k 是上述 k 循环的归纳变量。

如果在一个循环中有两个或更多个的归纳变量，则归纳变量的个数往往可以设法减少，甚至减少到仅一个。

考察上例中归纳变量 j 和 k 各自的作用。显然，j 是每次重复执行循环时加入到和数 S 中的当前项，而 k 是计数器，记录是第几次重复，或说加入到和数 S 中的是第几项。如果能使 j 每次重复执行循环得到相应的值，即 j 每次加 3，且通过 j 的最终值 90 来控制循环的继续与否，变量 k 就可删除。从源程序级上看，相当于：

```
      S=0;  j=0;
Loop: if(!(j<90)) goto Finish;
      j=j+3;
      S=S+j;
      goto Loop;
Finish:
```

这是从源程序级理解删除归纳变量的概念。现在来考察相应的四元式序列：

```
(1)  :=   0        S            (7)  :=   t1       j
(2)  :=   1        k            (8)  +    S    j    t2
(3)  <=   k   30   (6)          (9)  :=   t2       S
(4)  GO   (5)                   (10) +    k    1    t3
(5)  GO   (13)                  (11) :=   t3       k
(6)  *    3    k    t1          (12) GO   (3)
```

从四元式序列看，t1 与 j 是归纳变量，每次重复执行循环，其值固定增加 3，可以有：

```
(6')  +   t1   3   t1
```

因此，t1 的初值是 0，其终值将是 87。由于 k 将被删除，（2）代之以：

```
(2') :=    0       t1
```

最终可写出四元式序列如下：

```
(1)  :=   0        S            (6)  +    t1    3    t1
```

```
(2)  :=   0        t1          (7)  +    S    t1    t2
(3)  <=   t1   87  (6)         (8)  :=   t2        S
(4)  GO   (5)                  (9)  GO   (3)
(5)  GO   (10)
```

显然，删除了变量 k 的四元式序列与删除之前的四元式序列等价。

概括起来，归纳变量删除优化是减少循环的归纳变量个数的优化。

在进行归纳变量删除优化时，要点是：

- 找出要删除的归纳变量；
- 确定要进行的等价变换。

3. 计算强度削减

前面看到，关于基本块的计算强度削减优化是利用代数恒等式，用运算速度快的计算代替运算速度慢的计算。关于循环，也有计算强度削减的优化，例如，例 7.9 中，四元式（6）对 t1 的计算 3*k 被改成了 t1+3，从乘法运算改进成了加法运算。这是与循环的归纳变量有关的优化，即归纳变量的计算强度削减。

由于归纳变量的值在每次重复执行循环体时总是增加或减少一个固定的常量值，一般来说，总是可以对归纳变量进行计算强度削减优化。

第二种与循环有关的计算强度削减优化是数组元素地址计算时的计算强度削减。如前所述，当假定各维的下界都是 1 时，一个数组元素的地址往往可以由下式来计算：

```
base-C+d
```

对于二维数组元素 $A[j_1][j_2]$，base–C 是数组 A 的虚拟元素 A[0][0]的存储地址，而 $d=(j_1\times n_2+j_2)\times sizeof(T)$，其中 n_2 是第二维元素个数，是常量。T 是数组元素的类型，简单起见，令 sizeof(T)=1，因此，循环中数组元素地址计算要能进行计算强度削减优化，必需 $d=j_1\times n_2+j_2$ 满足一定的条件。

通常，二维数组元素出现在二重循环体内，例如，

```
for(v1=v10; v1<=v1f; v1=v1+1)
   for(v2=v20; v2<=v2f; v2=v2+1)
    {   … A[j1][j2]…  }
```

显然只需 j_1 与 j_2 都是关于 v1 与 v2 的线性式，即

$$j_1=C_{10}+C_{11}v1+C_{12}v2$$
$$j_2=C_{20}+C_{21}v1+C_{22}v2$$

其中 C_{k0}、C_{k1} 与 C_{k2}（k=1,2）都是常量，对 d 的计算便可进行计算强度削减优化，即用加法代替乘法，因为在内循环时，v1 固定，v2 每次加 1，d 的值增加一个常量值，当在外循环时，v1 每次加 1，进入内循环时 d 的初值也是增加一个常量值。这表明，对于 C 语言这一类数组各维上下界都是常量的语言来说，只需数组元素的下标是循环控制变量如下的线性式：

$$j=C_0+C_1v1+C_2v2$$

其中 C 都是常量，便可进行数组元素地址计算强度削减优化。这请读者自行验证。

概括起来，关于循环的计算强度削减优化有两类，一类是归纳变量的计算强度削减优化，另一类是数组元素地址计算的计算强度削减优化。不论是哪一类，在实现优化时，必须对相关变量的初值及递加的增量进行计算并作出相应的处理。

要提醒的是，对循环的优化是对中间表示代码，即四元式序列进行的。

7.4.2 循环优化的基础

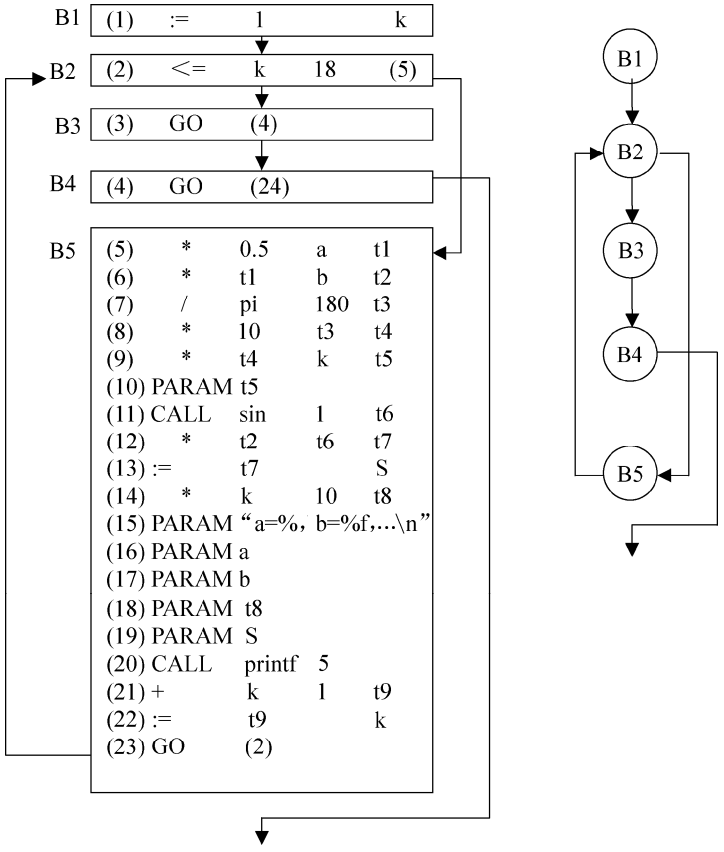
1. 流图与循环结构的识别

经过语义分析后，中间表示代码中标志循环结构的关键字 while、do 与 for 等已不复存在。为

了进行循环结构的优化，首先必须识别出循环结构。为了清晰地表达四元式序列中各基本块之间的控制联系，引进流图的概念。

流图是把控制流信息加到基本块集合所形成的有向图。现在流图中的结点是基本块，结点之间的有向边代表控制流。对于一个结点，如果相应的基本块中第一个四元式是四元式序列的第一个四元式，称该结点是流图的首结点。

【例 7.10】 为例 7.8 中计算三角形面积值的 C 程序所生成的四元式序列中，入口四元式是（1）、（2）、（3）、（4）与（5），共有 5 个基本块 B1、B2、…与 B5，分别包含四元式（1）、（2）、（3）、（4）与（5）～（23）。可画出相应的流图如图 7-9（a）所示，或简洁地画成如图 7-9（b）所示。



（a）　　（b）

图 7-9

各个基本块的控制流向，并不像一般程序设计语言中的 if 语句那样指明控制转移条件，这需要由出口四元式确定。

为了从流图识别出循环结构，需要了解循环的两个基本性质，即

- 循环必具有唯一的入口点，称为首结点，首结点是循环中所有结点的必经结点；
- 对于循环中任何一个结点，必定至少存在一条路径回到首结点。

这表明：循环内任何两个结点之间，必定存在路径把它们相连，且该路径上的一切结点都属于该循环。从首结点进入循环，沿着某路径到达某一结点后，必定能从该结点沿着一路径回到首结点。如果找不到这样的回到首结点的路径，就将构不成一个循环。

下面通过必经结点与回边的概念来识别循环结构。

必经结点指的是：如果从流图中某初始结点出发，每条到达结点 n 的路径都要经过结点 m，则称结点 m 是结点 n 的必经结点，表示为 m dom n（dom 是 domain 的前 3 个字母）。若 m=n，则 m dom m，即任何结点都是它自己的必经结点；若 m≠n，则 m 是 n 的前驱结点，而 n 是 m 的后继结点。这时若 m 只通过一个有向边达到 n，则 m 是 n 的直接前驱结点，n 是 m 的直接后继结点。

回边指的是：假定流图中存在两个结点 M 与 N，有 M dom N，则边 N→M 称为流图的回边。如果关于回边 N→M 找出不经过结点 M 而能达到结点 N 的一切结点，连同结点 M 与 N，显然构成一个循环，其首结点是 M，称此循环是关于回边 N→M 的**自然循环**。

一个自然循环由构成它的所有结点的集合来表示，识别循环结构，也就是识别流图中的回边及组成回边的相应自然循环的一切结点。

【例 7.11】 自然循环的例子。设有流图如图 7-10 所示。为了寻找自然循环，需要找出回边 N→M。通常控制流箭头从上向下，回边一般地说是从下向上的，这样，从下向上的有向边 N→M 就是回边。这时必须注意应有 M dom N。显然图 7-10 中有回边 4→3、7→4、10→7、8→3 与 9→1，因为相应地有 3 dom 4、4 dom 7、7 dom10、3 dom 8 与 1 dom 9。

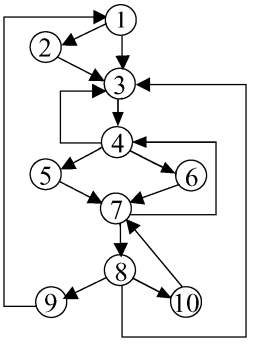


图 7-10

考察与回边 4→3 相关的自然循环。是{3,4}吗？请注意，循环有唯一的首结点，即入口点。现在不仅结点 3 是入口点，结点 4 也是入口点，因为有 7→4。相应的自然循环应是{3,4,5,6,7,8,10}，其首结点是结点 3。类似地，与回边 9→1 相应的自然循环是{1,2,3,4,5,6,7,8,9,10}，首结点是 1。请读者自行考虑相应于其他回边的自然循环。

在流图中寻找自然循环的思路是：找出满足条件 M dom N 的回边 N→M 后，找出不经过结点 M 而能达到 N 的一切结点，即 N 的一切前驱结点，连同 M 与 N，便构成关于回边 M→N 的自然循环。当找出一切回边，便可找出全部自然循环。

把自然循环作为循环来处理，然而循环结构是复杂的，可以是平行循环结构，也可以是嵌套循环结构。但不论如何，要进入循环，必须通过循环首结点。如 C 语言等大多数语言都禁止从循环外控制转移进入循环内。一个流图，当删去其中的回边后便构成无环路有向图，这种流图称为可归约流图。从首结点进入循环是可归约流图的一个重要性质。应用结构式程序控制结构所写出的结构化程序的流图总是可归约流图。

2. 数据流分析

识别出循环，就为实现对循环的优化提供了机会。这时可进行循环不变表达式外提、归纳变量删除与计算强度消减等优化。除了基本块优化及与循环相关的优化外，还可以在更大范围内，即全局范围内进行消去全局的公共子表达式与删除无用代码等优化。

如前所述，将通过数据流分析来收集进行这几类优化所需的信息流信息。本小节先讨论相关的 3 类数据流方程。

（1）到达—定值数据流方程

一个表达式要是循环不变表达式，表示该表达式中涉及的变量值，在循环内都不改变，也就是说，这些变量的值是在循环外获得值，仅在循环内引用。变量获得值的方式通常可以有如下几种，即通过赋值语句赋以值，通过输入语句输入值，或者通过函数形实参数传递取得值，等等。

不论哪一种方式，都称为对变量的定值。为分析程序中所有变量的定值与引用之间的关系，引进一类所谓的到达—定值数据流方程，相应的概念如下。

① 点：用来指明流图的结点基本块中的位置，包括第一个四元式之前的点（称入口点）、每两个相邻四元式之间的点和最后一个四元式之后的点（称出口点）。

② 定值：使变量获得值的四元式。例如，四元式 d: op　x　y　z，使 z 获得值，该四元式 d 的位置称为 z 的定值点，通常就称 z 有定值 d。

③ 引用点：引用某变量 z 的四元式的位置称为变量 z 的引用点。

④ 到达—定值：假定变量 z 有定值 d，且存在某条路径可以到达某点 p。如果 d 之前某点处对变量 z 有最新定值，则点 p 处便不能引用 z 在 d 处的定值，称 z 的定值 d 被注销，因而 d 处的定值不能到达点 p，否则 d 处的定值能达到点 p。

⑤ 引用—定值链：设变量 x 有一引用点 u，变量 x 能到达点 u 的一切定值的集合称为变量 x 在引用点 u 处的**引用—定值链**，或称为 ud 链（u 与 d 各是 use 与 define 的首字母）。

例如，例 7.8 中，k 的定值（1）随循环体的重复执行而被注销，k 的引用点（9）与（21）处的 ud 链是{22}。因此（9）与（21）中的 k 不是循环不变量。而关于 a 与 b 的定值在此四元式序列内并未反映出来，可以看成是第一个四元式前定值，这表明 a 与 b 并未在循环内定值，因此是循环不变量。因此（5）～（8）相应于循环不变表达式。

概括起来，要实现循环不变表达式外提优化的关键是：求得循环内一切变量的引用点处的 ud 链。首先计算基本块入口点处各个变量的定值集合，然后求得到达基本块中某点时的变量的定值集合。为此引进**到达—定值数据流方程**如下：

$$IN[B] = \bigcup_{p \in P[B]} OUT[p]$$

$$OUT[B]=GEN[B]\cup(IN[B]-KILL[B])$$

其中 P[B]表示基本块 B 的一切直接前驱结点基本块集合，对方程中的符号解释如下。

IN[B]表示基本块 B 入口点处各个变量的定值集合，则在基本块 B 中某点 p 处的 ud 链可按下列方式求得:

- 若 B 中点 p 之前有 x 的定值 d，且该定值能到达点 p，则点 p 处 x 的 ud 链是{d}。
- 若 B 中点 p 之前没有 x 的定值，则 IN[B]关于变量 x 的每个定值都能到达点 p，即点 p 处变量 x 的 ud 链是 IN[B]中关于 x 的定值集合。

OUT[B]表示基本块 B 出口点处各个变量的定值集合。显然，基本块出口点处的 OUT 集合是其直接后继结点基本块入口点处的 IN 集合。能到达基本块出口点处的定值必定是如下两种情况之一：即一是在基本块内定值，二是在基本块外定值，但在基本块内未被注销而达到出口点处。

GEN[B]是各个变量在基本块 B 内定值并能达到 B 的出口点处的所有定值集合，而 KILL[B]是各个变量在基本块 B 内重新定值，因而在基本块 B 内被注销的定值集合。由于 GEN[B]与 KILL[B]可从流图结点基本块求出而作为已知量，因此上述方程组是关于变量 IN[B]与 OUT[B]的联立方程组。假定一个给定的流图中包含 n 个结点基本块，则这是 2n 个变量（IN[B]与 OUT[B]各 n 个）的 2n 个方程的联立方程组。

为了理解此方程组，给出下例进行说明。

【例 7.12】 设有流图如图 7-11 所示。它由 4 个基本块组成，分别为 B_1、B_2、B_3 与 B_4。总计定值四元式有 8 个，分别记为 d_1、d_2、…、d_8。各基本块的 GEN 与 KILL 集合分别为：

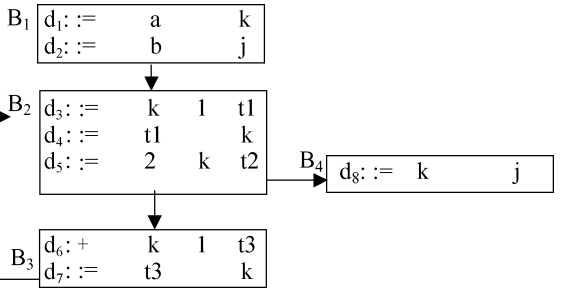


图 7-11

$GEN[B_1]=\{d_1,d_2\}$　　$KILL[B_1]=\{d_3,d_4,d_5,d_6,d_7,d_8\}$

$GEN[B_2]=\{d_3,d_4,d_5\}$　　$KILL[B_2]=\{d_1,d_7\}$

$GEN[B_3]=\{d_6,d_7\}$　　$KILL[B_3]=\{d_1,d_4\}$

$GEN[B_4]=\{d_8\}$　　$KILL[B_4]=\{d_2\}$

到达—定值数据流方程组通常采用迭代算法求解，相应的 C 型语言程序可给出如下：

```
for(j=1; j<=n; j=j+1)
{  IN[Bj]={ };
   OUT[Bj]=GEN[Bj];
}
change=true;   /* 1 */
while (change)
{  change=false;  /*  0  */
   for (j=1; j<=n; j=j+1)
   {  IN[Bj]=  ∪  OUT[p];
            p∈P[Bj]
      oldOUT=OUT[Bj];
      OUT[Bj]=GEN[Bj]∪(IN[Bj]-KILL[Bj]);
      if (OUT[Bj]!=oldOUT) change=true;
   }
}
```

其中引进了变量 change，标志最新一次迭代时，OUT[B]是否改变。若 OUT[B]不再改变，因此 IN[B]也不再改变，结束 while 循环，得到所求的解。应用上述算法，可得到最终结果如下：

$IN[B_1]=\{\ \}$　　$OUT[B_1]=\{d_1,d_2\}$

$IN[B_2]=\{d_1,d_2,d_3,d_5,d_6,d_7\}$　　$OUT[B_2]=\{d_2,d_3,d_4,d_5,d_6\}$

$IN[B_3]=\{d_2,d_3,d_4,d_5,d_6\}$　　$OUT[B_3]=\{d_2,d_3,d_5,d_6,d_7\}$

$IN[B_4]=\{d_2,d_3,d_4,d_5,d_6\}$　　$OUT[B_4]=\{d_3,d_4,d_5,d_6,d_8\}$

不言而喻，上述算法中的集合运算，需要设计相应子程序来实现。

由此结果可见，基本块结点 B2 与 B3 组成的循环中不包含循环不变量，也即不能进行循环不变表达式外提的优化。

（2）活跃变量数据流方程

在对基本块进行删除无用代码优化时，删除的是因消除公共子表达式而引进的复写四元式。然而，当在一个循环范围内，甚至整个程序范围内，在删除基本块内看来是无用代码的一些代码时，必须考虑在其他的基本块内，是否可能要引用这些代码的计算结果。

一般情况下，基本块内的值未被引用的计算，还必须考虑这个计算在该基本块外的引用。如果该计算在该基本块外将被引用，尽管在基本块内是“无用”的，也不能被删除。如果所计算的一个值，存放在寄存器中，而在基本块外还将被引用，该值便必须存储。概括地说，必须收集变量在各基本块中被引用的信息，为此引进活跃变量的概念。

如果对于变量 x 和流图上的某个点 p，存在一条从点 p 开始的路径，在此路径上引用变量 x 的值，则称变量 x 在点 p 是**活跃变量**，否则称变量 x 在点 p 不活跃。

判别变量在基本块出口点处是否活跃的工作称为活跃变量分析。

进行活跃变量分析的必要性是明显的。如果变量 x 在点 p 处定值，然而之后并没有在所在基本块或循环内被引用，并且在该基本块或循环出口点处之后是不活跃的，显然该变量 x 在点 p 的定值是无用代码，可以删除。

为了计算在基本块入口点与出口点处的活跃变量集合，类似地引进 L_IN 集合与 L_OUT 集合，这里 L 是单词 Live（活跃）的首字母。

L_IN[B]是在基本块 B 入口点处的活跃变量集合，L_OUT[B]是在基本块 B 出口点处的活跃变量集合。L_USE[B]是在基本块 B 中引用，但引用之前未曾在 B 中被定值的变量集合，而 L_DEF[B]是在基本块 B 内定值，且在定值前未曾在 B 中引用的变量集合。一个基本块入口点处活跃的变量，必定是在该基本块内未被重新定值而仅引用的变量，如果重新定值，则在入口点处必不活跃。如果某基本块出口点处，一个变量是活跃的，这个变量必定在其后继结点基本块中被引用，也即在其后继结点基本块入口点处是活跃的。换句话说，在后继结点基本块入口点是活跃的变量，在它所在的结点基本块出口点处必定是活跃的。因此建立**活跃变量数据流方程**如下。

$$L\_IN[B]=L\_USE[B]\cup(L\_OUT[B]-L\_DEF[B])$$

$$L\_OUT[B]=\bigcup_{s\in S[B]} L\_IN[s]$$

其中 S[B]表示基本块结点 B 的一切直接后继基本块结点集合。

显然 L_USE[B]与 L_DEF[B]是可从流图求得的已知量，当流图由 n 个基本块结点组成时，上述方程组也是 2n 个变量（L_IN[B]与 L_OUT[B]各 n 个）、2n 个方程的联立方程组。同样地采用迭代方法求解。相应算法步骤的 C 型语言程序如下：

```
for (j=1; j<=n; j=j+1) L_IN [Bj]={  };
change=true;    /*  1  */
while (change)
{ change=false;   /*  0  */
   for(j=1; j<=n; j=j+1)
   { L_OUT[Bj]= ∪  L_IN[s];
              s∈S[Bj]
     oldIN=L_IN[Bj];
     L_IN[Bj]=L_USE[Bj]∪(L_OUT[Bj]-L_DEF[Bj]);
     if(L_IN[Bj]!=oldIN) change=true;
   }
}
```

【例 7.13】　对于例 7.12 中的流图（见图 7-11），求解活跃变量数据流方程。从流图可知：

$L\_USE[B_1]=\{a, b\}$　　$L\_DEF[B_1]=\{k, j\}$

$L\_USE[B_2]=\{k,t1\}$　　$L\_DEF[B_2]=\{t1,k,t2\}$

$L\_USE[B_3]=\{k,t3\}$　　$L\_DEF[B_3]=\{t3,k\}$

$L\_USE[B_4]=\{k\}$　　$L\_DEF[B_4]=\{j\}$

最终求解结果是：

$L\_IN[B_1]=\{a,b,t1\}$　　$L\_OUT[B_1]=\{k,t1,t3\}$

$L\_IN[B_2]=\{k,t1,t3\}$　　$L\_OUT[B_2]=\{k,t1,t3\}$

$L\_IN[B_3]=\{k,t1,t3\}$　　$L\_OUT[B_3]=\{k,t1,t3\}$

$L\_IN[B_4]=\{k\}$　　$L\_OUT[B_4]=\{\ \}$

（3）可用表达式数据流方程

一个基本块内若存在有公共子表达式，这些公共子表达式的出现，可以代之以先前所计算的值。在更大范围内，也可能存在有类似的情况，也就是说，如果一个基本块内出现的某个表达式，先前已在其他基本块内出现过，并且计算过值，而且从那次计算到这次出现，表达式所涉及的变量值都未被改变，那么先前所计算的表达式的值可代替这次出现的表达式。只是这里把先前出现的表达式称为可用表达式。确切地说，可用表达式的概念如下：

可用表达式指的是：如果从一个流图的首结点到点 p 的每条路径上，都有对表达式 x op y 的计算，并且在最后一个这样的计算和点 p 之间，没有对 x 和 y 的重新定值，则称表达式 x op y 在点 p 是可用的。

如果一个基本块 B 对 x 与 y 定值，并且以后没有重新计算 x　op　y，则称基本块 B 注销表达式 x　op　y，这表明先前的 x　op　y 是不可用的。如果基本块对 x　op　y 进行计算，而且以后没有对 x 或/与 y 重新定值，则称基本块 B 产生表达式 x　op　y，只要不对 x 与 y 重新定值，那么这个 x　op　y 之值在此后都是可用的。

显然，如果表达式 x　op　y 在点 p 处是可用的，则在点 p 处出现 x　op　y 时，它就可以作为公共子表达式而被消除。

【例 7.14】　为下列 C 语言程序片段：

```
if(a+b>c) x=(a+b)*2; else x=(a+b)/2;
z=x+1;
```

生成的四元式序列之流图如图 7-12 所示。

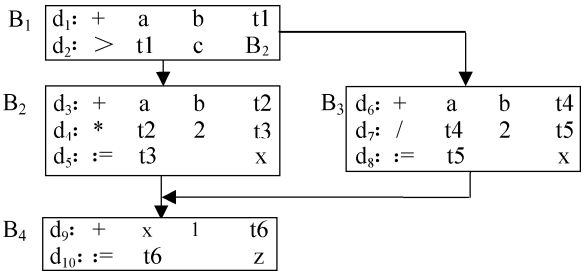


图 7-12

其中，基本块 B_1 中计算 a+b，在基本块 B_2 与 B_3 中也都有 a+b 的计算，由于在 B_1 中已计算 a+b，并且未改变 a 与 b 的值，在 B_2 与 B_3 之入口点处 a+b 是可用的，因此可以消除公共子表达式 a+b，在基本块 B_4 的入口点处显然 a+b 仍是可用的。如果基本块 B_4 中有对 a 与/或 b 的定值，从而注销表达式 a+b，则 B_4 入口点处表达式 a+b 将是不可用的。请注意，基本块 B_1 中的控制转移四元式，用基本块序号来指明所转向的位置，这样的方式，在进行优化后四元式序列有变化的情况下是更合适的，显然这样将不受四元式个数变化的影响。

如何识别一个基本块中的可用表达式？假定在基本块的入口点处没有可用表达式。对基本块四元式序列中的四元式逐个地分析，如果点 p 处的可用表达式集合是 A，点 q 是点 p 的下一个点，点 p 与点 q 之间的四元式是 op x y z，而且没有对 x 与 y 的定值，则点 q 处的可用表达式集合如下：

- 可用表达式集合 A 中所有的可用表达式，加上表达式 x op y；
- 删去上述可用表达式中任何含 z 的表达式。

这时点 q 处除了原先 A 中的可用表达式外，又多了一个可用表达式 x op y，但由于对 z 定值，有新值 x op y，含 z 的表达式自然成为不可用的。

为了寻找一切基本块的可用表达式，类似地引进 A_IN 集合与 A_OUT 集合，这里 A 是单词 Available（可用的）的首字母。**可用表达式数据流方程**如下：

```
A_OUT[B]=(A_IN[B]-A_KILL[B])∪A_GEN[B]
A_IN[B]= ∩  A_OUT[p]      B≠B1
       p∈P[B]
A_IN[B1]={  }             B1是首结点基本块
```

其中 A_IN[B]是基本块 B 入口点处的可用表达式集合，A_OUT[B]是基本块 B 出口点处的可用表达式集合，A_GEN[B]是基本块 B 中产生的可用表达式的集合，A_KILL[B]是基本块 B 中被注销的表达式集合。显然，A_GEN[B]与 A_KILL[B]可由流图直接给出。例如图 7-12 中的流图，有

```
A_GEN[B1]={a+b}           A_KILL[B1]={ }
A_GEN[B2]={a+b, t2*2}     A_KILL[B2]={ }
A_GEN[B3]={a+b, t4/2}     A_KILL[B3]={ }
A_GEN[B4]={x+1}           A_KILL[B4]={ }
```

因此类似地，当流图中包含 n 个基本块时，也是关于 2n 个变量（A_IN[B]与 A_OUT[B]各 n 个）的 2n 个方程的联立方程组。

对于可用表达式数据流方程，同样地采用迭代方法求解。C 型语言表达的算法步骤如下：

```
A_IN[B1]={ };
A_OUT[B1]=A_GEN[B1];
for (j=2; j<=n; j=j+1)
  A_OUT[Bj]=U-A_KILL[Bj];
change=true;   /*  1  */
while (change)
{ change=false;  /*  0  */
   for (j=2; j<=n; j=j+1)
   { A_IN[Bj]= ∩  A_OUT[p];
             p∈P[Bj]
     oldOUT=A_OUT [Bj];
     A_OUT [Bj]=A_GEN[Bj]∪(A_IN[Bj]-A_KILL[Bj]);
     if(A_OUT[Bj]!=oldOUT) change=true;
   }
}
```

其中 U 表示所谓的全集，即四元式序列中所有表达式 x op y（即 op x y）的集合，这是因为以足够大的近似开始，逐步减少，就可期望得到最大可能解。

7.4.3　循环优化的实现

当经过数据流分析，特别是基于 3 类数据流方程的求解，收集了流图中各基本块之间的数据流信息，为实现与循环相关的优化，进一步实现全局优化，打下了基础。下面简单讨论关于循环的 3 类优化的实现，以及全局公共子表达式消除与无用代码（复写四元式）的删除。

与循环相关的优化包括循环不变表达式外提、归纳变量删除与计算强度削减。

1．循环不变表达式外提

为了实现循环不变表达式外提优化，需要解决 3 个问题：即如何识别出循环不变表达式、把不变表达式外提到何处，以及在什么条件下才能外提。

（1）循环不变表达式的识别

当一个表达式的值不随循环体的重复执行而改变时，它就是循环不变表达式，这表示，表达式中涉及的一切变量的定值全在循环外。这时可以利用引用—定值链，即 ud 链来确定引用点处定

值的位置，从而识别变量的定值是否全在循环外。如果一个变量的定值全在循环外，它是循环不变量，如果一个表达式中的变量全是循环不变量，则是循环不变表达式。

ud 链由求解到达—定值数据流方程而得到。当已建立 ud 链时，可采用加标记算法。如果一个四元式 op x y z，其中的 x 与 y，或者是常量，或者是所有定值都在循环外的循环不变量，便对此四元式加“不变”标记，从而识别出一切循环不变表达式。

（2）循环不变表达式外提到何处

对于循环不变表达式，它的值在循环内不随重复执行而改变，这仅表示只需计算一次，之后在循环内引用该值，但并不是说不需计算其值。按照程序设计的“先定义，后使用”法则，循环不变表达式应在进入循环内引用之前计算好。因此，循环不变表达式外提到循环首结点之前最合适，就是建立一个前置结点，在其中计算外提的循环不变表达式的值。要注意的是，应该是等价变换，必须保持原有的控制联系不变。确切地说，循环内原先到首结点的有向边不变，原先从循环外引向循环首结点的有向边，修改为循环外引向前置结点的有向边。前置结点到循环首结点有一条有向边，如图 7-13 所示。

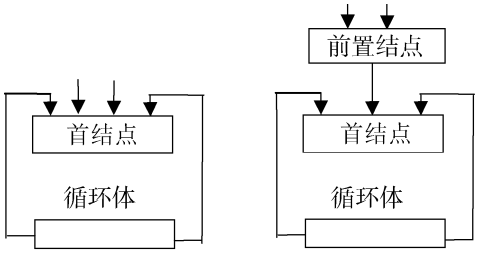


图 7-13

这样外提前后的流图是等价的。但问题是，并不是任何循环不变表达式都能外提到前置结点，事实上必须满足一定的条件。

（3）循环不变表达式外提的条件

假定四元式 Q: op x y z 是循环不变计算，即 x op y 是循环不变表达式，它能外提的条件如下：

条件 1　四元式 Q 所在基本块结点是循环所有出口点的必经结点，这里循环出口点是有后继结点不在循环内的结点。

条件 2　循环中 z 没有其他定值点。如果 z 是在编译时刻由编译程序引进的，则此条件必定满足而无需检查。

条件 3　循环中 z 的引用点仅由 Q 到达。如果 z 是临时变量，则此条件一般也满足。

下面图 7-14 中给出不满足条件的 3 种情况。请读者自行分析哪些是循环不变表达式，如果把它们外提，会产生什么后果，从而理解之所以不能外提的原因。

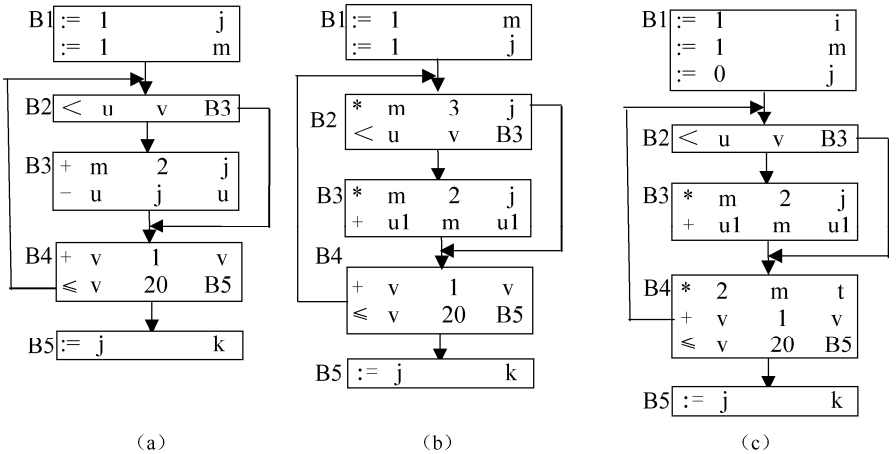


图 7-14

概括起来，当已计算 ud 链及必经结点信息时，循环不变表达式外提的步骤如下：

步骤 1　寻找循环中的一切不变计算四元式。

步骤 2　对所找到的每个不变计算四元式 Q: op　x　y　z 检查是否满足可外提的 3 个条件。

步骤 3　按照找出循环不变计算四元式的顺序把其中满足上述 3 个条件的不变计算四元式外提到循环的前置结点基本块中。不言而喻，如果 Q 中的运算分量 x 与 y 在循环内定值，则只有在 x 与 y 的定值四元式外提到前置结点基本块之后，才能把 Q 外提到前置结点基本块。

2. 归纳变量计算强度削减与归纳变量删除

归纳变量是随每次重复执行循环体而增加或减少一个常值的变量。要进行与归纳变量相关的优化，必须首先识别出哪些是归纳变量。循环中每次重复执行循环体，仅增加或减少一个常值的归纳变量是最简单的归纳变量，例如，循环控制变量 i 往往是如此，可以有 i=i+c 或 i=i–c，其中 c 是常量，因此称为基本归纳变量。一般的归纳变量往往是基本归纳变量的线性函数。例如，$j = c_1 \times i + c_2$，其中 c_1 与 c_2 都是常量。如果变量 j 是基本归纳变量 i 的线性函数，称 j 为 i 族归纳变量。

寻找归纳变量的思路如下。

步骤 1　扫描构成循环的四元式，找出所有基本归纳变量。

步骤 2　寻找循环中只有一个赋值（定值）的变量 k，对它的定值有如下形式之一：

```
* j c k      * c j k      / j c k
+ j c k      + c j k      - j c k      - c j k
```

其中 c 是常量，j 是归纳变量。当 j 是基本归纳变量时，k 是 j 族归纳变量。如果 j 不是基本归纳变量，而是 i 族归纳变量，即有 $j=c_1 \times i+c_2$，其中 c_1 与 c_2 都是常量，显然，必定有 $k=c_1' \times i+c_2'$，其中 c_1' 与 c_2' 也都是常量，且可在分析时刻计算好，即 k 也是 i 族归纳变量。这时必须满足两个要求：

- 在循环中对 j 的唯一赋值和对 k 的赋值之间没有对 i 的赋值。
- 没有 j 在循环外的定值可以到达 k 的这一定值点。

当 k 与 j 都是属于同一个基本块中的临时变量时，上述两条是容易检查的，对于更一般的情况，需要利用到达—定值信息，进行对上面两个要求的检查。

总的说来，为寻找归纳变量，必须在数据流分析的基础上，已获得到达—定值信息和循环不变计算四元式的信息。为了实现归纳变量相关的优化，可以利用三元组来表示，例如，假定 j 是 i 族归纳变量，$j=c_1 \times i+c_2$，可表示为(i,c_1,c_2)。基本归纳变量 i 可表示为(i, 1,0)。

由于基本归纳变量 i，每次重复执行循环体增加值 c，则 i 族归纳变量 $j=c_1 \times i+c_2$ 每次重复执行循环体将增加值 $c_1 \times c$。归纳变量计算强度削减的优化可由下列步骤实现：

步骤 1　建立新变量 t，如果 i 族归纳变量 j_1 与 j_2 有相同的三元组，则仅需建立一个新变量，两者共用。

步骤 2　用四元式:=　t　　j 对 j 赋值。

步骤 3　在循环中每个基本归纳变量 i 之定值 i:=i+c 之后，增加 t:=t+n（n 等于 $c_1 \times c$ 的值），更确切地说，四元式+　i　c　i 或+　i　c　t 与:=　t　　i 的紧后面，添加四元式+　t　n　t。把 t 归入 i 族，它的三元组是(i,c_1,c_2)。

步骤 4　在前置结点基本块的最后，对于 i 族归纳变量 t 添加定值 $t:=c_1 \times i+c_2$，即添加下列两个四元式：

```
*   c1   i    t
+   t    c2   t
```

如果 c_1 为 1 或 c_2 为 0，作相应的简化。

从上面的步骤可以看到：当进行归纳变量计算强度削减优化后，一些归纳变量在循环内的计

算已仅每次重复时增加或减少一个常量值，与基本归纳变量不再有直接的联系。当某个基本归纳变量在循环中的作用仅用于测试，以控制循环执行次数，便可能利用对另外某个归纳变量的测试来替代，而该基本归纳变量再无存在的必要，这正如同例 7.9 所见，k 是基本归纳变量，而 j=3*k 是 k 族归纳变量，k 主要用于控制循环控制次数，一旦对 j 进行归纳变量计算强度削减优化后，对所引进的 t1 值的测试可用来控制循环执行次数时，基本归纳变量 k 已再无存在的必要，因此可删除。

现在考虑如何把对基本归纳变量 i 的测试用对 i 族归纳变量 j 的测试代替。假定 j 的三元组是 (i,c_1,c_2)，即 $j=c_1\times i+c_2$，而对 i 的测试是 relop　i　x　B，其中 x 不是归纳变量。显而易见，把 i 与 x 的比较可代之以 j 与 $c_1\times x+c_2$ 的比较，即把 relop　i　x　B 用下列四元式来代替：

```
    *     c1    x     r    /* 若 c1=1，则改为:=  x       r */
    +     r     c2    r    /* 若 c2=0，则省略此四元式 */
relop     j     r     B
```

其中 r 是新的临时变量。不言而喻，应从 j 的三元组中选取尽可能简单的，如 $c_1=1$ 与 $c_2=0$。对于 relop　x　i　B，类似地处理。

对于 relop　i_1　i_2　B 这类测试，其中 i_1 与 i_2 都是基本归纳变量，检查 i_1 与 i_2 能否被替代。宜于替代的，仅是最简单情况，即 j_1 与 j_2 分别是 i_1 族与 i_2 族归纳变量，且三元组分别是(i_1,c_1,c_2)与(i_2,c_1,c_2)，这时 relop　i_1　i_2　B 等价于 relop　j_1　j_2　B，更一般的复杂情况，并不值得进行替换测试。

当删除一些归纳变量后，由于不再引用这些归纳变量，可删除循环内对它们的定值。另外还需考虑删除归纳变量后的其他一些影响，例如，归纳变量计算强度削减优化时，用复写四元式:=　t　j 代替了对 j 的赋值，现在假定删除了归纳变量 j，便应该令对 t 的引用代替对 j 的引用，并删除复写四元式:=　t　　j。当然，必须检查在复写四元式:=　t　　j 和任何对 j 的引用之间没有对 t 的定值（赋值）。在更一般的情况，在对 j 定值的基本块外引用 j 就需要到达—定值信息，并对流图进行分析来实现这种检查。

要注意的是归纳变量所增加值的正负号。把四元式 relop　i　x　B 替换为 relop　j　r　B，是因为 i 与 j 所增加的值同号。如果一个是正值，另一个是负值，则需要改变 relop，例如，如果 $j=-4\times i$，i 每次增加 1，与测试 $I\leqslant 20$ 等价的将是 $j\geqslant -80$，即需改变关系运算符 relop。

这里又看到了到达—定值数据流方程的作用。不论是归纳变量计算强度削减，还是归纳变量删除，进行这两类优化的前提是已有到达—定值信息，并且找出了循环中的归纳变量。对于归纳变量删除优化，除了循环不变计算信息外，还需要活跃变量信息。

7.5　全局优化的实现思想

这里讨论的全局优化着重于公共子表达式消除与复写四元式删除两类优化。

1．公共子表达式消除

这里的公共子表达式不再仅是在一个基本块内的，而是构成循环的各个基本块内的，甚至更大范围的。这时，为在一个基本块中的公共子表达式利用其他基本快内已计算的值，显然要利用可用表达式的信息，它可通过求解可用表达式数据流方程获得。

消除全局公共子表达式的实现思想如下。

对于流图的一切基本块中形如 op　x　y　z 的四元式 Q：如果在 Q 所在基本块的入口点处 x　op　y 可用，且在该基本块中，Q 之前没有对 x 和 y 的定值，则顺着流图的有向边从该基本块开

始，反向搜索，不穿过任何计算 x　op　y 的基本块，找出到达 Q 所在基本块的 x　op　y 的计算，即四元式 R: op　x　y　t，因此 R 是到达 Q 的最后一个计算 x　op　y。这时建立新变量 u，把四元式 R 用下列两个四元式替代：

```
op  x  y   u
:=  u      t
```

这时 x　op　y 的值在 u 中，因此四元式:=　u　　z 代替四元式 Q。

如果在流图的两个基本块中存在如下的两对四元式：

```
+  x  y  a          +  x  y  c
*  a  z  b          *  c  z  d
```

其间没有对 x、y 与 z 的定值。那么除了 x+y 是公共子表达式外，是否还有其他公共子表达式? 从计算的值来看，b 与 d 都有值(x+y)*z，d 的计算可用 b 来代替。但从字面来看，是识别不出的。为了能识别这类公共子表达式，可以采用迭代方法。首先从+　x　y　a 代替为：

```
+   x   y   u
:=  u       a
```

把*　a　z　b 识别作*　u　z　b，这样，类似地四元式*　c　z　d 也可识别作*　u　z　d，从而识别出公共子表达式 u * z。

2. 复写四元式的删除

当对基本块进行消除公共子表达式优化时，所产生的复写四元式，往往因复写传播而成为无用代码，可以删除。当进行全局公共子表达式消除时，类似地将产生这样的复写四元式。由于通过复写四元式定值的变量，大多是局部于基本块的临时变量，因此可以考虑删去这些复写四元式。

假定有复写四元式 d: :=　x　y，它对变量 y 定值。为了删除此复写四元式，找出其后对 y 的一切引用点，用 x 去代替 y，则可删除此复写四元式 d。但这时对 y 的每个引用 u 必须满足下列两个条件：

① 此定值 d 是到达 u 的唯一的对 y 的定值，即 u 处的 ud 链仅包含 d。

② 从 d 到 u 的每条路径上，包括穿过 u 若干次的（但不第二次穿过 d 的）路径上没有对 x 的定值。

为检查这两个条件的满足与否，实现复写传播而删除复写四元式，同样需要利用 ud 链等信息，即应用到达—定值数据流方程。另外，还可以关于复写四元式，引进类似于可用表达式的数据流方程。这里不拟详加讨论。

7.6　窥 孔 优 化

前面讨论的优化，不论是基本块的优化，还是与循环相关的优化，都是基于源程序的中间表示代码的，具体地说，是基于四元式序列进行的。即使进行了所有这些优化，所生成目标代码中，依然可能存在有可能优化的结构。例如例 7.8 中的：

```
(3)  GO  (4)
(4)  GO  (24)
```

与例 7.9 中的：

```
(4)  GO  (5)
(5)  GO  (10)
```

相应的目标代码明显可以进一步优化，即在目标代码级，分别得到相当于：

```
(3)  GO  (24)
```

与　　(4)　GO　(10)

的优化。

之所以存在这种情况，是因为目标代码与中间表示代码一样，只能按一般情况（共性）来生成。因此当必要且可能时，在无需花费太大代价的前提下，可以对机器级目标代码进行进一步的优化，许多简单的优化变换便可以大大改进目标代码运行的时间和空间功效。由于这是在很小范围内进行的优化，这类优化称为窥孔优化。

窥孔优化是在机器语言目标代码级上进行的局部优化，它仅考虑目标代码中很短的指令序列，只要有可能，就把它优化为更短或更快的指令序列。

窥孔优化可以有下列几类优化：

- 冗余指令删除；
- 控制流优化；
- 代数化简；
- 特殊指令的使用。

为了易理解、更直观，往往以虚拟机目标指令形式，甚至源程序语句作为例子来讨论。显然这并不失一般性。

7.6.1　冗余指令删除

1. 多余的存取指令的删除

假如源程序中有语句：

```
a=b+c;  d=a-e;
```

所生成的目标代码将如下所示：

```
(1) MOV  b,  R0      (4) MOV  a,  R0
(2) ADD  c,  R0      (5) SUB  e,  R0
(3) MOV  R0, a       (6) MOV  R0, d
```

指令（3）执行的结果，b+c 的值在 a 中，同时也在寄存器 R0 中。很明显（4）中把 a 的值传输入寄存器 R0 完全是多余的。只要没有控制转移指令从其他指令把控制转移到指令（4），指令（4）完全可以删除。由于这两个语句相应的四元式在同一个基本块中，不会发生控制转移到（4）的情况，删除（4）是安全的。

2. 死代码删除

死代码是程序中控制决不会达到的代码。例如紧随无条件控制转移指令之后的指令，如果没有控制转移指令把控制转移到这里，这些指令就将是死代码。由于死代码对程序的执行毫无作用，应该被删除。

一般地说，程序中不会有这样的死代码。如果存在，只表明程序书写中可能有错误。但对于 C 语言，由于提供有条件编译这样的预处理设施，运行时刻的目标代码中便可能包含死代码。例如，在调试时，C 语言程序中可能加入如下的程序片段：

```
#define  debug  1
⋮
if(debug==1)打印调试信息的语句;
⋮
```

相应的目标代码在调试状态下运行时，由于符号 debug 被替换为 1，因此条件 debug==1 恒真，打印调试信息。然而在正常运行状态时，debug 将如下定义：

```
#define  debug  0
```

因此上述条件事实上是条件 0==1，恒假，决不执行打印调试信息的语句。这表明，这些打印调试

信息的语句的相应代码是死代码，可以把它们删除。

7.6.2　控制流优化

控制流优化指的是：在目标代码中减少不必要的控制转移，甚至删除不必要的控制转移指令，以提高程序的运行效率。

最简单的情形是如下所示的控制转移：

```
L0: goto  L1;
⋮
L1:goto  L2;
```

显然，可把 goto L1 改为 goto L2，从 L0 直接控制转移到 L2。如果不存在其他到 L1 的控制转移，那么指令 L1: goto L2 也可以删除。

其他可进行控制流优化的情况如下所示。

条件转移到转移：

```
if (b) goto L1;
⋮
L1:  goto L2;
```

转移到条件转移：

```
L0:goto L1;
⋮
L1:if (b) goto L2;
L3:
```

这两类情况的控制流优化，请读者自行讨论。

7.6.3　代数化简

窥孔优化时的代数化简也是利用代数恒等式进行等价变换，达到优化的效果。只是窥孔优化是在很小的局部范围内，以甚少的代价实现的机器语言级的优化，因此仅利用类似于 x=x+0 与 x=x×1 之类简单的恒等式进行代数化简，删除相应的指令。当利用恒等式 $x^2=x\times x$，用 x×x 代替 x^2 时，将比调用计算幂次的子程序效率更高。

7.6.4　特殊指令的使用

当目标机的指令系统内涵丰富时，为实现同一个功能，很可能有若干个选择，如果能使用简洁快速的特殊指令，将更有利于运行效率的提高。例如，对于源程序语句 i=i+1，通常生成下列三条目标指令：

```
MOV  i,   R0
ADD  #1,  R0
MOV  R0,  i
```

如果目标机目标指令系统包含增 1 指令 INC 这样的特殊指令时，对于上述语句只需生成一条指令：

```
INC  i
```

类似地，对于 i=i−1 这样的赋值语句，可以只生成一条减一指令这样的特殊指令来实现。

当目标机指令系统中包含实现某类特定操作的特殊指令时，可以进行窥孔优化，找出允许使用这些特殊指令的情况，并设法用这些特定指令来实现相应的操作。显然，由于使用了特殊指令，充分利用了目标机指令系统的特性而可以大大减少目标代码的执行时间。

概括起来，窥孔优化包括上述 4 类优化，其特点是每次改进都可能会引起新的改进机会。为了达到最大的改进，一般需要对目标代码重复扫描进行窥孔优化。

本章小结

本章讨论中间表示代码与代码优化，包括概况、中间表示代码、基本块的代码优化、与循环有关的优化、全局优化的实现思想与窥孔优化。

对源程序的翻译总是只能按照一般情况，或说共性来考虑，与手工编程相比，编译程序生成的目标程序质量低劣很多，需要进行代码优化。代码优化指的是编译时刻为改进目标程序质量而进行的一系列工作，这是在中间表示代码基础上进行的。中间表示代码一般有四元式序列、三元式序列、逆波兰表示与抽象语法树。本章重点讨论四元式序列基础上的代码优化。应注意关于四元式的书写约定，能熟练地为 C 语言程序写出四元式序列。

代码优化可以从不同角度来分类，本章主要讨论与机器无关的、中间表示代码级的优化。在基本块的基础上进行与循环有关的优化，进一步进行全局优化。关于基本块的优化有 4 类，即合并常量计算、消除公共子表达式、削减计算强度与删除无用代码。基本块的优化利用所谓的无环路有向图 dag 来实现。注意，在构造 dag 时必须要构造相应的结点信息表。

与循环相关的优化包括：循环不变表达式外提、归纳变量删除与计算强度削减，而计算强度削减又包括：归纳变量计算强度削减与数组元素地址计算强度削减。由于经语义分析生成中间表示代码之后，标志循环结构的关键字 while、do 与 for 已不复存在，因此必须进行控制流分析，识别出循环结构，并且进行数据流分析，收集基本块之间的数据流信息。为此建立 3 类数据流方程，即到达—定值数据流方程、活跃变量数据流方程与可用表达式数据流方程。读者应理解这 3 类数据流方程的作用与含义。

相关概念有寄存器描述符、地址描述符、引用—定值链（ud 链）。

复习思考题

1．通常有哪些改进程序质量的方法？本章的代码优化指的是什么？
2．为什么要划分基本块？基本块的特征是什么？
3．为什么可以利用 dag 进行基本块的优化？
4．如何在中间表示代码中识别出循环结构？
5．如何判别一个变量是否是循环的不变量？
6．3 类数据流方程的含义是什么，对循环优化有何作用？

习　　题

1．试写出赋值语句 x=a*(b+c)−(e−f)/d 的四元式序列。
2．试写出 if 语句 if(x>10||x<5) y=x+5; else y=10−x;的四元式序列。
3．试写出下列 while 语句的四元式序列：

```
while(j>0 && A[0]<A[j])
{ A[j+d]=A[j]; j=j-d; }
```

4．为下列四元式序列生成虚拟机目标代码。

```
(1)  :=    10         i      (7)  +     t1     t2     t3
(2)  :=    0          s      (8)  :=    t3            s
(3)  ⩾     i    0     (5)    (9)  -     i      1      t4
(4)  GO    (12)             (10) :=    t4            i
(5)  *     s    x     t1     (11) GO    (3)
(6)  =[ ]  A    i     t2     (12) :=    s             y
```

5．为下列基本块构造 dag。

```
(1)  :=    0          prod        (7)  *     t2      t4     t5
(2)  :=    1          i           (8)  +     prod    t5     t6
(3)  *     4     i    t1          (9)  :=    t6             prod
(4)  =[ ]  A     t1   t2          (10) +     i       1      t7
(5)  *     4     i    t3          (11) :=    t7             i
(6)  =[ ]  B     t3   t4          (12) ⩽     i       20     (3)
```

并从 dag 还原成四元式序列。说明所进行的优化。

6．假定需要计算 20mn 个圆柱体的重量，有 m 个不同密度、n 个不同半径和 20 个不同高度，它们的值分别存放在数组 p、数组 r 和数组 h 中，计算重量的公式是 $V=\pi r^2hp$，计算结果不保存，立即输出。可写出 C 程序如下。

```
for(i=1; i<=m; i=i+1)
{  for(j=1; j<=n; j=j+1)
   { for(k=1; k<=20; k=k+1)
     {  W=pi*r[j]*r[j]*h[k]*p[i];
        printf("h=%f,r=%f,p=%d 时的重量=%f\n",h[k],r[j],p[i],W);
     }
   }
}
```

试应用循环不变表达式外提优化的思想，进行改写，使得计算速度最快。

7．设有实型数组 A，试求其最大元素，把相应序号存放于 NoMax 中。可写出 C 语言程序如下：

```
max=A[1];  NoMax=1; j=1;
while(j<30)
{  j=j+1;
   if( max<A[j]) { NoMax=j;  max=A[j]; }
}
```

每个实型数据占 4 个存储字节，相应四元式序列如下。

```
(1)  *     4    1    t1          (11)  =[ ]    A     t4    t5
(2)  =[ ]  A    t1   t2          (12)  <        max   t5    (14)
(3)  :=    t2        max         (13)  GO       (18)
(4)  :=    1         NoMax       (14)  :=       j           NoMax
(5)  :=    1         j           (15)  *        4     j     t6
(6)  <     j     30  (8)         (16)  =[ ]     A     t6    t7
(7)  GO    (19)                  (17)  :=       t7          max
(8)  +     j     1   t3          (18)  GO       (6)
(9)  :=    t3        j           (19)
(10) *     4     j   t4
```

试进行删除归纳变量优化，写出优化了的四元式序列。

8．设有下列 C 语言程序

```
#define  m  30
main( )
{  int i, n,t;    float  A[100];
   n=50;  k=1;
   for( i=k;  i<=k+m-1;  i=i+1)
   {  t=A[i];  A[i]=A[i+n];  A[i+n]=t;  }
}
```

试写出相应的四元式序列，进行所有可能的优化，假定每个实型数据占 4 个存储字节。写出最终优化的四元式序列。

第8章 程序错误的检查与校正

本章导读

编译程序对源程序进行翻译，把它翻译成等价的低级语言目标程序，但源程序中难免存在错误，编译程序应能处理源程序中存在的错误，使得在一次编译中能发现尽可能多的错误，甚至全部错误。本章讨论源程序中错误的检查与校正问题，内容包括：概述、词法错误的复原与校正、语法错误的复原与校正以及语义错误。所提及的词法错误与语法错误的校正思路，可供读者在设计出错处理子程序时参考。作者极力推荐使用静态模拟追踪法与动态调试工具，作为语义错误的检查措施。

8.1 概　　述

8.1.1 程序错误检查的必要性

如前所述，一个程序设计语言被看作一般的语言时，程序将被看作它的句子。这表明程序中没有错误，既没有词法错误，也没有语法错误等，因此最终能生成目标代码，运行后获得预期的效果。

但由于所要解决问题的复杂性（包括问题规模与算法等）和程序员素质等因素，程序中往往包含错误。由于对系统环境不够了解，甚至键入失误，也会引进一些错误。因此程序中存在错误是难免的，特别是中大型软件的程序，错误情况尤其严重。如果在编译时刻发现源程序有一个错误，立即停止编译，结果将令人难以容忍。试回想早期由计算机房管理人员统一管理程序上机运行的情景！所有程序穿孔在纸带或卡片上，排队上机。好不容易排到了时间，上机编译运行了，却因一个小错误必须修改后重新排队等待。改好后说不定又因一个小错误，又必须再次重新排队等待。即使在今天个人计算机发展甚为迅速的年代，如果因一个小错误就必须重新编译，也会十分不便。

概括起来，一个编译程序必须能处理源程序中的错误，使得对错误进行处理后，能继续进行编译，以便一次编译就能找出源程序中尽可能多的错误，甚至全部错误。错误的检查与处理应是编译程序设计中的一个重要方面。

8.1.2 错误的种类

一个源程序是一个基本符号序列或字符序列，由基本符号或字符拼写成符号，再由符号按语法规则构成语法成分，最终构成程序。按照语言的语义，编译程序把源程序翻译成等价的目标程序。因此一个源程序中一般有如下几类错误：词法错误、语法错误和语义错误。只是要注意，还

可能有违反环境限制的错误。

① 词法错误：编译程序在词法分析阶段发现的源程序错误。例如，关键字拼写错、标点符号错与非法字符错等。

② 语法错误：编译程序在语法分析阶段发现的源程序错误，这时，书写不符合语法成分的语法规则。例如，括号不配对、if 语句中的条件表达式未用小括号对括住，以及 else 没有与之匹配的 if 等。变量未说明或被重定义，也可看作语法（全局语法）错误。

③ 语义错误：源程序中的语义错误有静态语义错误与动态语义错误两类。在编译时刻能发现的语义错误是静态语义错误，例如，语义分析时，发现一个运算的两个运算分量类型不相容，或者对某些运算分量进行不允许的运算等，也可能是试图从循环外把控制转入循环内。动态语义错误是在目标程序运行期间才能发现的语义错误，这类错误在编译时刻没有任何反映，但运行时运行不能正常结束，或验证运行结果时发现是不正确的。动态语义错误往往是逻辑上的，即算法上的错误。例如，把 a + b 错写成 a − b，或把(b*b − 4*a*c)/(2*a)写成了(b*b − 4*a*c)/2*a，导致不能正确地反映算法，又如把 if(x = = 0)写成了 if(x = 0)，导致控制条件不正确，甚至可能编程所依据的算法本身就存在错误。

④ 违反环境限制的错误：尽管一个程序设计语言可以有丰富的表达能力，用来书写各种应用程序，然而由于编译程序的实现问题，一个使用中的编译程序往往对它所能接受的源程序施加某些限制。例如，某些编译程序对标识符的长度、整数的最大取值范围，甚至对数组的最大维数与 if 语句的最大嵌套层数都有一定的限制。违反环境限制错误的最典型例子是：C 语言中整型变量 i 的值是 32 767 时再加 1，导致−32 768 的结果，这是因为 C 语言规定 int 型量的最大值是 32 767。

对于一个好的编译程序来说，应具有较强的查错和改错的能力。

查错，就是编译程序能在编译时刻及时发现源程序中的错误，并能以简明的方式向程序书写人员报告错误的性质和错误所在的确切位置。正确的处理方式不应是发现一个错误就立即停止编译，而是应该在一次编译期间，发现源程序中尽可能多的错误，以便在尽短时间内改正。虽然希望对于目标程序运行时刻才能发现的动态错误，也提供相应的错误信息，但一般说来，仅诊断型编译程序才这样处理。

改错，是指编译程序在其翻译过程中发现源程序的错误时，能对源程序作出适当的修正，也即校正。显然，由于源程序书写的灵活性、错误发生的随机性与可能校正的多样性，自动校正错误的难度相当大。为了正确地校正，必须了解错误的性质，确切地对错误定位，甚至了解程序的意图。即使是最简单的词法错误，也必须根据错误所在的上下文，试探性地作出修改，不能简单地校正，不过自动校正的信息可作为程序书写人员进行校正时的参考。

总的说来，如果一个编译程序能在一次编译时刻，查出源程序中几乎所有的错误，指出错误的性质，给出错误所在的确切位置，并能提供尝试自动改正的信息，那么对于迅速改正源程序错误将有非常大的帮助。本章讨论的程序错误的检查与校正，涉及的主要是静态错误，即词法错误、语法错误、非逻辑或算法上的语义错误，以及违反环境限制的错误。

8.1.3　相关的基本概念

如果一个源程序没有错误，编译程序将顺利地对它进行词法分析、语法分析、语义分析及目标代码生成，其中还可能进行目标代码的优化。但一个编译程序不应该只能处理正确的源程序，而对存在的错误束手无策，遇到错误时便停止编译。一个好的实用的编译程序，在编译一个源程序时，必须能处理源程序中的错误，使得发现一个错误时不是结束编译，而是继续下去，以便一次编译查出尽可能多的错误，甚至全部错误。

在编译过程中，发现源程序中的错误时，采取一定的措施，使得能继续进行编译，这称为**错**

误复原。

当源程序中发现某个错误时，往往可能由此错误导致编译程序向源程序书写人员发出更多的错误信息，而所发出的更多的出错信息往往是不真实的，这种信息称为株连信息。例如，假定源程序中包含一个函数调用 f(R.m)，其中，m 是结构变量 R 的成员变量，但在键入时错键入成了 f(R, m)。编译时，处理到符号“)”时，将发出如下 3 个报错信息：“m 无定义”、”参数的类型不匹配”与“参数多一个”。显然这 3 个报错信息是因为把圆点“.”错成了逗号“,”而引起的株连信息，是不真实的。有时可能因为源程序中的一个错误而引出一连串株连信息，应该遏止这种株连信息，使得尽可能少。

如果发现上述错误时，取得关于函数 f 的参数信息（1 个），以及标识符 R 的类型信息（结构类型），并向前看到右括号，确定参数的个数，这样，甚至可以作出正确的修改，把“,”改成“.”。这样便遏止了株连信息。

再看一个遏止株连信息的例子。假定源程序中出现数组元素 A[j][k]，编译程序发现标识符 A 不是数组名，扫描到符号“[”时将发出报错信息：符号[错。此后显然将发出一连串错误的株连信息，例如不合法的间接错。原因可能是未对 A 进行数组说明。为了遏止株连信息，一种简便的处理方式是：让“万能”标识符 U 代替标识符 A，从而让 A 可与任何的数据类型相关联。这时在符号表相应条目中加上相应的标志。以后根据上下文所取得的信息，往符号表中填入相应的属性。标识符 U 或 A 将是关联于数组类型，且为二维数组。这样，此后再次在源程序中扫描到形如 A[e_1][e_2] 的成分时，将不再发出报错信息。

如果同一个错误出现在源程序中多处时，就将形成重复信息。例如，上例中，不对标识符 A 进行如上处理时便是这样。如果一个标识符未在函数定义内说明，则在该函数定义内的语句部分中，每次引用时都将发出报错信息：标识符×××无定义。显然只需报错一次就可以。这只需在扫描到函数定义的末了处发出报错信息：“××函数定义中，标识符×××无定义”。这就简明得多。

在一般情况下，为了遏止重复信息，可以事先建立一个出错信息表，在其中给出一切可能的出错信息和编号，在编译时刻建立一个出错信息集合，其元素呈（编号, 关联信息）形式。每当发现一个错误，便把相应的（编号, 关联信息）添加入该出错信息集合中，相同的出错信息显然仅出现一次。当编译结束时，按某种次序输出出错信息集合中的出错信息，便无重复信息。

概括起来，在错误复原时，应做到：

- 遏止株连信息；
- 遏止重复信息。

8.2 词法错误的复原与校正

8.2.1 词法错误的种类

词法分析程序的基本功能是：读入源程序字符序列，识别开具有独立意义的最小语法单位（单词或符号），并把它们转换成属性字形式的内部中间表示。因此词法分析时发现的是词法错误，大多数是单词拼写错误，可能是因为书写错误，也可能是因为键入错误。所以，假定不会有连续几个字符的错误，在这假定下，词法错误有如下几类：

- 拼错一个字符，例如，default 错成 defoult；
- 遗漏一个字符，例如，switch 错成 swich；

- 多拼一个字符，例如，struct 错成 structe；
- 相邻两个字符颠倒了次序，例如，return 错成了 retrun。

对于错误复原来说，需解决下列问题：

- 错误的查出；
- 错误的定位；
- 错误的局部化；
- 重复错误信息的遏止。

词法分析是在正则文法基础上进行的，各类单词通常都可以用正则表达式描述。在识别单词时，通常采用最长子串匹配策略。如果输入字符序列中，尚未扫描部分的任何前缀都不能与所有的词形相匹配，则表明存在错误，应调用出错处理子程序进行处理。

8.2.2　词法错误的校正

由于假定词法分析时不存在连续几个字符都错，词法错误仅拼错、遗漏、多拼一个字符，以及相邻两个字符颠倒次序这 4 类，因此对词法错误的校正相应地有如下 4 类：

- 替换一个字符；
- 插入一个字符；
- 删除一个字符；
- 交换相邻两个字符。

因为词法分析时，还没有收集到足够的信息，发现错误便立即校正将为时过早，只是在某些情况，例如下面所述的，可以予以校正。

① 知道下一步应处理的是关键字，而尚未扫描的输入字符序列部分中，当前所扫视的任何前缀都不能构成关键字，则查关键字表，从其中选择与当前所扫描的输入字符序列前缀最接近的关键字去代替这个前缀。例如，switch(i){ casc…，其中 casc 是错误的，且不能在关键字表中查到，最接近的是 case，用它去代替 casc。

② 如果某个标识符拼写错了，而不能在符号表中查到相应条目，可用符号表中与它最接近的标识符去代替它。例如，如果有函数调用 sim(x)，但在符号表中查不到标识符 sim，则可以用最接近的 sin 去代替 sim。

③ 其他拼写错误的某些情况，可以用与上面类似的方法来校正。例如，数组元素的数组标识符，因拼写错误而不能在符号表中查到时，可以用符号表中可查到的、说明为数组且有相同维数的、拼写最接近的数组标识符去代替。又如，控制转移语句的转移目标（标号）因拼写错误而无定义时，可在符号表中找到拼写最接近的标号标识符去代替等。

一般地说，不论是用替换、插入、删除，还是交换的方法去校正，都是试探进行的，可以以最可能成功的那种修改作为对错误的校正。不言而喻，必须输出校正信息，供程序书写人员校正时参考。

8.3　语法错误的复原与校正

8.3.1　语法错误的复原

对于语法错误的复原，与词法错误复原的情况一样，需解决下列问题：

- 错误的查出；
- 错误的定位；
- 错误的局部化；
- 重复错误信息的遏止。

语法分析基于上下文无关文法进行，与词法错误的查出相比较，语法错误可以由基于某种分析技术的识别程序自动查出。由于不同的分析技术发现错误的手段和方式不同，有的分析技术可对所发现的错误确切地定位，有的则不能。例如，对于 LR(1)分析技术，是在当前分析栈栈顶状态与当前输入符号匹配所对应的分析表元素为空时发现错误，因此可以对错误确切定位，然而存在有一些分析技术（如优先分析技术）是不能确切地定位的。但为了语法错误的复原，不论哪种情况，在发现语法错误时，都必须对错误确切地定位，采取一定的措施，使语法分析能继续进行下去。

对语法错误复原，最简单的处理是放过相应的语法结构，例如放过到一个语句的后继符号。这种过于简单的处理，往往可能使得失去发现更多语法错误的机会。设法尝试校正所发现的语法错误，是更可取的处理方式，这样除了减少错过发现错误的可能外，还因为即使不能保证成功改正错误，但提供的校正信息可供程序书写人员参考。

8.3.2 语法错误的校正

语法分析技术有自顶向下与自底向上两大类，不同的语法分析技术决定了语法错误的校正也有不同的处理方式。

1．自顶向下分析中语法错误的校正

假定在按自顶向下分析技术分析过程的某一时刻，源程序符号序列可写成 w_1Aw_2 的形式，其中 w_1 是源程序中已扫描部分，A 是当前扫描符号，w_2 是输入符号序列的其余部分。如果扫描到 A 时发现错误，分析程序又无法确定下一个合法的分析动作，这表明，自顶向下分析过程中，已构造的语法分析树可覆盖 w_1，但不能继续构造语法分析树去覆盖 A 与输入符号串的其余部分 w_2，这时的校正措施一般可有如下 3 种。

① 删去 A 再继续进行下去；

② 插入终结符号串 x 成为 w_1xAw_2，从 xAw_2 的首符号开始继续进行分析；

③ 修改 w_1，例如删去 w_1 尾部的若干个符号、替换 w_1 尾部的若干个符号或者在 w_1 的后面插入若干个符号。

第 3 种措施显然是不可取的，因为 w_1 已处理过，不可能再次重新扫描，尤其是如果对已处理部分进行修改，往往必须要更改语义信息，实现上较为困难。

下面以 LL(1)分析技术为例，说明如何进行校正。可以在 LL(1)分析表表示出错的空白元素处，根据所预期的输入符号，事先设计好错误处理子程序进行语法错误的校正。例如，假定源程序中一个赋值语句错写成如下所示：

```
i=i+);
```

当处理到符号“)”时，假定分析栈栈顶元素为 T，而输入符号为)，分析表 A 的元素 A[T][)]=“ ”，发现错误。可令 A[T][)]对应于出错处理子程序 E_j，其功能是删除当前输入符号，并发出报错信息，因此这时发出报错信息：“符号）错，删除”。下一当前输入符号为“;”，依然有 A[T][;]=“ ”，这时可让 A[T][;]对应于出错处理子程序 E_k，其功能是：插入标识符 i，并发出报错信息：“缺少运算分量，插入标识符 i”，因此这时发出报错信息：“缺少运算分量，插入标识符 i”。这样 A[T][i]=

“T:: = FT′”，将正常地继续语法分析下去。显然把上述赋值语句 i = i +);校正为 i=i+i;是合理的选择。

一般地，当按自顶向下的 LL(1)分析技术构造语法分析树时，对照语法分析树和预期展开成的符号串，让分析表中每个空白元素对应于一个出错处理子程序，采用上述修改措施①与②，可以尝试进行语法错误的校正。

2．自底向上分析中语法错误的校正

当采用自底向上分析技术时，LR(1)分析表的 ACTION 部分指明了当前分析栈顶的状态与当前输入符号匹配所应执行的动作。如果它是空白元素，则表明存在错误，即当前输入符号是不正确的。为了对语法错误校正，可以类似于 LL(1)分析技术，令 ACTION 部分每个空白元素对应于一个错误处理子程序，根据出错情况，在各个出错处理子程序中作出相应处理。

这里仍以赋值语句 i=i+);为例来讨论。

假定扫描到上述语句中的符号+时进入状态 S_m。显然，将有 ACTION[S_m][)]=“　”（出错）。现在令该元素对应于出错处理子程序 E_j，其功能是：删去当前输入符号，并发出报错信息：“删除不合法的输入符号：…”。因此，执行动作 ACTION[S_m][)]将调用出错处理子程序 E_j，删去当前输入符号)，并发出报错信息：“删除不合法的输入符号：)”。然后将扫描下一个符号继续分析。这时的输入符号是“;”，依然有 ACTION[S_m][;] = “　”（出错）。现在可令它对应于出错处理子程序 E_k，其功能是：插入标识符 i，并发出报错信息：“缺少运算分量，插入标识符 i”。插入标识符 i 后，将执行动作：ACTION[S_m][i] = Sn，把(i, n)下推入分析栈，此后 LR(1)识别程序将继续正常进行分析。因此，上述错误的赋值语句也校正成了 i = i + i;。

综上所述，两大类分析技术都可以事先设计好出错处理子程序，在处理到分析表中的出错元素时，便调用相应的出错处理子程序。当程序书写人员得到相应的报错信息，便可以在校正时参考。

8.4　语 义 错 误

8.4.1　语义错误的种类

如前所述，语义错误有两类：一类是编译时刻可以发现的静态语义错误，另一类是在运行时刻才能发现的动态语义错误。

1．静态语义错误

静态语义错误源自于数据类型，也可能源自于控制结构。

运算符不合法和运算分量类型不相容，是典型的数据类型引起的静态语义错误。例如，C 语言中实型量进行整除求余运算（%），以及实型量与字符串（字符数组）型量进行算术运算都是不正确的，这样的由数据类型引起的静态语义错误可以在语义分析时查出。

语义分析程序还进行控制流方面的某些静态语义检查。例如试图从循环外把控制转移到循环内，以及由无条件控制转移语句把控制转移到 if 语句的内嵌语句内，等等。

显然，静态语义错误是容易准确地定位和确定错误性质的，然而，正确的校正必须由程序书写人员自行完成。

2．动态语义错误

在运行目标代码时才能发现的源程序错误是动态语义错误，最常见的有如下几类：

- 除以零；

- 存取未置值或值为 NULL 的指针变量所指向对象的成员；
- 数组元素的下标表达式的值越界；
- 变量未被赋值便被引用；
- 显示输出的运行结果与预期的不一致。

前两种情况往往导致运行非正常终止而得不到任何结果，然而后 3 种情况可以正常终止，但是得不到预期的效果。

一般地说，程序设计错误（包括键入错误引起的错误）和逻辑错误将引起动态语义错误，这时的错误往往表现为运行结果与预期的不一致。如果是在软件开发的设计阶段或需求分析阶段没有正确确定问题的数学模型或算法不正确，就必须重新考虑数学模型或算法，才能实现对这些错误的校正。

程序设计的错误也会导致程序不反映算法，使运行结果与预期的不一致，也可能导致程序运行的夭折。特别是与指针变量有关的语义错误，可能对低地址存储区域赋值，甚至可能造成巨大破坏。试看下面的 C 程序片段。

```
p=q;
while (p→next!=NULL)
{  …
   p=p→next;  p→inf=…;
  …
}
```

这程序片段看来正确，但实际上包含着重大错误。

在这里判别了 p→next 是否为 NULL。不为 NULL 时可以执行 while 循环中的两个赋值语句：p=p→next; 与 p→inf=…;，但如果 q 的值是 NULL，或者并没有为 q 所指向的对象分配存储空间，这两个赋值语句将可能带来致命的错误。只有 p 不是空指针，才可能对其值不是 NULL 的成员变量 next 所指向的对象赋值。应该把 while 循环的重复条件：

```
p→next!=NULL
```

改写为：

```
p!=NULL && p→next!=NULL
```

之前当然应对 q 置初值 NULL，或通过下列赋值语句：

```
q=(…)malloc (sizeof (…));
```

为 q 所指向的对象分配存储空间。

8.4.2 语义错误检查措施

由于语义通常是非形式地定义的，而且语义往往涉及算法，因此语义错误往往难以采用系统而有效的方法来发现和校正。

语义分析时采用语法制导的翻译，从而使得能系统地描述：如何应用语法制导定义或翻译方案实现类型相关的静态语义检查和关于控制流的某些静态语义检查。

对于变量未赋值就被引用的语义错误，可以在语义分析时查看变量是否被赋值而避免。

为了发现其他更一般的动态语义错误，通常采用下列两种方式：应用静态模拟追踪法与使用动态调试工具。

1. 静态模拟追踪法

静态模拟追踪法，顾名思义，是静态模拟执行程序，对变量的值进行追踪检查。进一步说，是由人模拟计算机，相继模拟执行程序中各个语句，沿着所模拟的执行路径，记录下程序中各个变量值的变化，最终检查结果的正确性。下面给出一个简单的例子。

【例 8.1】　设有下列程序片段：

```
x=x+y;  x=x-y;  y=x-y;
```

它实现什么功能呢？现在用静态模拟追踪法来确认其功能。

假定初始时刻 t_0 时，x 与 y 的值分别是 x_0 与 y_0。让相继执行 3 个语句的时刻分别为 t_1、t_2 与 t_3，则变量追踪的情况如表 8.1 所示。

表 8.1

	t_0	t_1	t_2	t_3
x	x_0	x_0+y_0	x_0+y_0	y_0
y	y_0	y_0	x_0	x_0

由表 8.1 可见，模拟结果是 x 与 y 的值相互交换了。即上述程序片段的功能是交换 x 与 y 的值。

2. 利用动态调试工具

静态模拟追踪法是由人静态模拟计算机的运行，进行变量值的追踪，同时也进行控制路径的追踪。采用这种办法可以查出程序中相当大比例的错误。然而，明显的不足是工作量太大，效率不高，尤其当变量较多，控制结构又较复杂时，这个不足更为突出。

利用软件工具在计算机上动态地调试检查，将大大地减轻人的负担。

编译程序可看成这样的软件工具，但如前所述，仅能查出编译时刻能发现的那些错误。为了发现动态语义错误，需要利用动态调试程序（debugger）。

动态调试程序通常包含下列功能：设置断点、查看变量值、步进（手工地逐个执行语句）与连续执行等。例如，如果有下列求和程序：

```
(1)   int  s=0, i=1;
(2)   while(i<=1000)
(3)   {  s=s+i;
(4)      i=i+1;
(5)   }
```

发现结果不正确。为了发现其中的错误，就必须利用动态调试手段。

以后每次都步进执行(3)和(4)处的语句，查看什么时候结果变得不正确。这样将花费太多的时间，可在程序中加入少量语句，例如，如下所示：

```
(1)  int  s=0, i=1;
(2)  while (i<=1000)
(3)  { s=s+i;
(4)    if(s<=0)
(5)      printf("s=%d\n", s);
(6)    i=i+1;
(7)  }
```

这时在(5)处设置断点，当程序运行到(5)处时由于 s 的值小于等于 0 而暂停，查看 i 的值，将找出 s 值错误的原因。

如果要检查数组元素 A[j]的下标表达式的值是否越界，类似地只需把断点设置在要查看的 A[j]处，程序运行到此处时暂停，显示查看 j 的值就可以了。读者可能会问：运行的是机器语言目标程序，并不是源程序，怎样能看懂目标程序？现在的动态调试程序是符号级的，即是符号的动态调试程序。当前流行的编译程序，如 Turbo C2.0、Borland C++ Builder 6.0 与 Visual C++6.0 等，都配置有符号调试程序，而且都是把编辑、编译、调试与运行集成于一体的程序设计语言支持系统，为编译与调试高级程序设计语言程序提供了极大的方便，极大地提高了程序生产率。

当然，即使有动态调试程序等来帮助查出程序中的错误，但程序中的错误的发现与校正仍然取决于人的经验。例如，对错误的性质和出现位置作出判断，以及选择断点位置等都需要经验。

尽管动态符号调试程序十分有用，但对于编译程序研制人员来说，符号调试程序的研制是整个编译程序研制项目中的一个组成部分，应由编译程序研制人员完成。本书重点是编译程序对源程序进行翻译的本体部分，对动态符号调试程序的实现问题不加讨论。

本章小结

源程序中难免存在错误，必要的是，一次编译就能找出源程序中尽可能多的错误，甚至全部错误。一个源程序中一般有如下几类错误：词法错误、语法错误、语义错误与违反环境限制的错误。

本章讨论源程序中错误的检查和校正。查出错误时对错误定位，并指明错误的性质，而且能继续编译下去，这就是错误的复原，这时要注意遏止株连信息和重复信息。困难的是错误的校正。一般来说，语法错误可以结合分析技术来查出，并由出错处理子程序设法校正。语义错误一方面需采用静态模拟追踪法由人查出源程序中的大部分错误，另一方面是在计算机上利用符号调试程序进行动态调试，这样更有利于发现并校正动态语义错误。熟练掌握动态调试，将能帮助尽快找出程序中所有错误。

相关概念有错误复原、错误校正。

复习思考题

1．源程序中一般存在哪几类错误？

2．一个编译程序，应怎样处理编译过程中发现的源程序错误？

3．什么是静态模拟追踪法？你使用过吗？有什么体会？

4．什么是动态调试？主要有哪些操作？怎么样才能充分利用调试功能？

习　　题

试在计算机上进行动态调试，发现下列程序在何时发生错误：

```
S=0;
for( k=1,j=1; j<=500; j=j+1)
{ S=S+k; k=k+3; }
```

第 9 章 目标代码的运行

本 章 导 读

尽管为源程序生成了目标程序，甚至是优化了的目标程序，但仍然不能运行。必须为目标程序准备好一切方面才可能运行。本章讨论的就是几个相关方面，即，运行时刻的存储管理、符号表及运行时刻支持系统。由于程序中存在三类不同的量，运行时刻的存储管理包括静态和动态两大类存储分配策略，动态存储分配策略又分栈式与堆式两类存储分配策略，请读者注意这几类存储分配策略的语言背景，了解由谁来进行存储分配。

符号表存在于整个编译期间，甚至存在于运行时刻，它的使用与存取功效直接影响到编译效率，需要关心符号表应有怎样的结构，用什么样的数据结构来实现。

运行时刻支持系统由运行子程序组成，没有运行子程序的支持，目标程序便不可能运行。请读者注意有哪几类运行子程序，它们有何作用。

9.1 概　　述

一个源程序在经历了词法分析、语法分析与语义分析，可能还进行了目标代码优化后，生成了目标代码。目标代码如果能运行并获得预期效果，必须考虑如下几个方面，即正确地生成了目标代码，以及具备了支持目标代码运行的一切条件。

为了正确地生成目标代码，必须考虑到目标程序运行时可能涉及的一切方面，必须把程序静态的正文与目标程序运行时的动态活动联系起来。除了语法成分目标代码的总体设计外，还必须考虑与运行紧密相关的运行时刻存储管理与寄存器分配，也就是说，在运行时刻，为变量分配存储处所，以便对变量进行存取。

对于寄存器分配，前面在一个简单的目标代码生成算法中，利用函数 GetRegister 进行了基本块的寄存器分配。C 语言中提供有存储类别 register，允许编程人员指明寄存器变量。C 语言编译程序通常把频繁使用的局部变量，例如循环控制变量指定为寄存器变量，不为它们分配存储字，而是分配寄存器，从而提高运行效率。显然，寄存器变量的引进并未带来实质性的问题，为篇幅关系，不拟对寄存器分配问题作进一步讨论。下面重点讨论运行时刻的存储管理问题，即运行时刻存储分配策略问题。

不论是运行时刻的存储管理，还是寄存器分配，都是源程序中的名字与数据对象之间的联系。这与符号表紧密相关，不仅在编译过程中要频繁访问符号表，而且在运行时刻，为了提供标识符的有关信息，也需要访问符号表。因此对符号表的组织管理也是编译程序实现必须考虑的重要方面。

即使生成的目标代码是正确的，考虑到了各个方面，它仍是不能运行的。如前所述， 目标程序必须在运行子程序的支持下才能运行。概括起来，本章将讨论下列几个相关的方面：

- 运行时刻存储管理；
- 符号表的组织与管理；
- 运行时刻支持系统。

9.2 运行时刻的存储管理

9.2.1 变量情况分析

程序中变量的值存储在存储器中，它们通常由变量的名字（标识符）来存取。运行时刻的存储管理，也就是对变量的存储分配与释放（收回）的管理。下面以例子来说明存储管理的几种情况。

【例 9.1】 设有 C 语言程序如下。

```
typedef struct NodeT
{ int data;  struct NodeT  * next;
} NodeType;
NodeType  *P;                                          (1)
void CreateLink ( )
{ NodeType *p1, *q;                                    (2)
  int j=1;
  q=(NodeType*)malloc (sizeof (NodeType));             (3)
  p1=q;
  while (j)
  { printf ("Input an integer value:");
    scanf ("%d", &j);
    if (j)
    { p1→next=(NodeType *)malloc(sizeof(NodeType));    (4)
      p1=p1→next;  p1→data=j;
    }
  }
  p1→next=NULL;  p=q→next;
  free (q);                                            (5)
} /* CreateLink */
void display ( NodeType *q)
{ NodeType *p1;                                        (6)
  p1=q;
  while (p1)
  { printf ("%4d", p1→data);  p1=p1→next; }
}
main ( )
{ CreateLink ( );  display(p); }
```

该程序的功能是边输入数据（整数值），边创建一个不带链头结点的单链表，它由指针变量 p 指向，然后逐个结点地输出所输入的数据值。请注意几种不同的变量：

- 全局变量 p（在(1)处定义）;
- 局部变量 p1 与 q（分别在(2)与(6)处定义）;
- 指针变量 q 与 p1 所指向的各个无名变量（分别在(3)与(4)处创建）。

对于全局变量 p，由于在编译时刻已经知道它的存在，且知道它的大小，因此在编译时刻可以为它分配存储区域，对全局变量的存取，也就是对它们所分配存储区域内容的存取。

对于局部变量 p1 与 q，如同第 6 章中所述，它们仅在函数被调用期间，即在函数活动的生存期才存在，具体地说，当函数被调用时，为它建立活动记录，并下推入运行栈中，为局部量分配的存储区域，就在活动记录的局部数据域。当函数执行结束而返回调用程序时，由于被调用函数的活动记录被上退去，不再为局部变量分配存储区域。这表明，p1 与 q 随所在函数被调用而存在，随所在函数调用结束而消亡。

对于指针变量 q 所指向的无名变量，它由执行下列语句而创建：

```
q=(NodeType *)malloc(sizeof(NodeType));
```

这无名变量在刚调用函数 CreateLink 时还不存在，直到执行上述语句后才被创建。但在执行了(5)处的函数调用 free(q)后，它被释放而不再存在，也就是说，在 CreateLink 函数的执行还没有结束时，q 所指向的无名变量已不再存在。然而(4)处的语句：

```
p1→next=(NodeType *)malloc(sizeof(NodeType));
```

只要 j 的值不是零（假值）就将被执行而创建一个无名变量，直到 j 的值为零时才不再创建无名变量。显然由于没有调用 free 函数，这些无名变量将一直存在，即使对 CreateLink 函数的调用执行结束时还是如此。因此这些无名变量是不能预先知道其存在与否，是随机地创建、随机地消亡的。

对上述 3 类变量，看到是完全不同的 3 类，第 1 类（p）是与整个程序同时存亡的全局变量，第 2 类（p1 与 q）是与函数调用同时存亡的函数局部变量，而第 3 类则是不能预先确定其存在和消亡的无名变量。在存储区中，第 1 类安排在静态存储区中，第 2 类安排在栈中，而第 3 类安排在堆中。

运行一个目标程序时，存储区域划分的示意图如图 9-1 所示。其中静态数据区中的，是其存在与大小在编译时刻已知、并可在编译时刻为其分配存储的数据，对于 C 语言来说，全局变量与静态局部变量被分配在静态数据区。栈就是运行栈。每当调用一个函数，便把相应的活动记录下推入运行栈，活动记录中包含了局部数据域。每当从被调用函数返回到调用程序时，把被调用函数的活动记录从运行栈中上退去。堆中的存储区域分配给由调用 malloc 等函数而创建的无名变量。

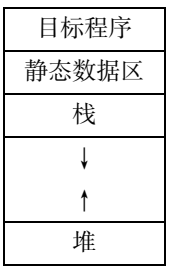


图 9-1

由于栈与堆的长度不是固定不变的，因此如图 9-1 所示，把栈与堆分别安排在存储区域的两端。通常栈向下增长，堆向上增长。目标程序在低地址一端，数据在高地址一端，当栈向下增长时存储地址将增大。

通常总是把数据安排在连续的存储区域中，数据对象由其类型决定所占存储区域的大小。一个基本类型的数据对象存放在若干个连续的字节中，为存取方便，对于数组或结构等构造类型的数据也存放在连续的存储区域中。由于存储器按字节编址，不同基本类型的数据有不同的编址要求，例如，每个整型数据必须以偶序号地址为首地址。在生成实际可运行的目标程序时，为数据对象分配存储时必须考虑边界对齐问题，使存储地址编址符合要求。为了处理简便，C 语言编译程序通常把字符型（包括字符串）数据集中存放在一起。

概括地说，从图 9-1 可见，存储区中有 3 个不同的数据区域，即静态数据区、栈和堆。第一个数据区在编译时刻由编译程序确定，然而后两个数据区（栈和堆）是在目标程序运行时刻动态地确定的，因此存储分配策略有静态存储分配策略与动态存储分配策略两大类。

9.2.2 静态存储分配

对于程序中的全局变量，例如例 9.1 中的全局变量 p，在编译时刻，编译程序已确定它的存在，并知道它的大小，可以由编译程序在编译时刻为它分配存储。这种在编译时刻进行的存储分配称为**静态存储分配**。

C 语言中的外部变量是全局变量，采用静态存储分配方式。用存储类别关键字 static 指明的静态外部变量与静态局部变量仅涉及作用域，本质上还是采用静态存储分配方式，安排在静态数据区。C 语言的编译程序在目标程序运行前，对静态存储分配的变量进行赋初值。

如前所述，C 语言这样的程序设计语言是不能仅仅采用静态存储分配策略的。函数调用设施，以及 malloc 与 free 等存储分配相关函数的存在，要求动态地创建数据对象，尤其是需要支持递归函数定义的实现，即使对于递归定义的函数的函数体内的同一个标识符，在运行时不同的活动中，也必须为相应的局部变量分配不同的存储区域。概括地说，必须采用动态存储分配策略。动态存储分配策略又分栈式存储分配策略与堆式存储分配策略两类。

9.2.3 栈式存储分配

如前所述，每当调用一个函数，便为该被调用函数建立一个活动记录，把它下推入运行栈，为局部变量分配存储的局部数据域包含于其中。每当执行结束，从被调用函数把控制返回到调用程序时，从运行栈中上退去该被调用函数的活动记录，从而撤销局部变量的存储分配。

如何确定局部变量的存储位置？如前所述，函数中局部变量的存储位置由相对于存储区域首地址的位移量来确定。这个首地址通常是由指针变量 top_sp 来指向，具体地说，top_sp 指向活动记录中机器状态域的末端，也即局部变量域的紧前位置，因此每个局部变量的存储地址可以表示为 $dx(R_{top\_sp})$，这里寄存器 R_{top_sp} 存放 top_sp 的值，而 dx 是局部变量 x 相对于 top_sp 所指向存储位置的位移量，为 x 分配的存储地址是 top_sp+dx。

由于每个函数的活动记录的长度在编译时刻是确定不变的，每当调用一个函数，只需把运行栈顶指针 top，即活动记录指针的值，增加被调用函数的活动记录的长度，在从被调用函数返回时，把 top 的值减少被调用函数的活动记录的长度，同时对 top_sp 的值作相应的修改。

对于函数中访问的非局部数据，它们或者是全局数据，或者是其他函数的局部数据，必定已分配了存储，或者在静态数据区，或者在相应函数的活动记录的局部数据域，只需通过访问链或显示表访问它们就可。由于栈式存储分配的局部变量的存在与否决定于函数调用，因此编译程序不可能在编译时刻对它们赋初值。在对它们赋值之前，它们的值是随机地不确定的。

读者可能会问：由谁在何处完成栈式存储分配，也即活动记录的分配与释放？

回顾语义分析目标代码生成一章，由调用序列实现活动记录的存储分配，由返回序列实现活动记录的撤销。进一步说，是由函数入口子程序实现对活动记录分配存储，由函数出口子程序实现对活动记录撤销存储。函数入口/出口子程序都是运行子程序。这进一步说明，目标程序的运行必须得到运行子程序的支持。

9.2.4 堆式存储分配

从例 9.1 的讨论中看到，由指针变量 q 与 p1 指向的无名变量的生命期，是与相关函数的生命期不一致的，可能更短，也可能更长，不能为这些无名变量在活动记录内分配存储，也就是说，不能采用栈式存储分配策略。一般地说，只要出现下列情况中的任何一种情况，就必须采用堆式

存储分配。

- 数据对象随机地创建、随机地撤销；
- 当函数的活动结束时，局部变量的值还得保存下来；
- 被调用函数活动的生命期比调用函数活动的生命期更长。

对于 C 语言这样的程序设计语言，需要采用堆式存储分配策略，主要表现在：存在一些与存储分配相关的函数，如 malloc 与 free 等，它们的存在使得可以随机地创建和随机地撤销无名变量。

在目标程序运行时，每当调用函数 malloc，便在如图 9-1 所示的堆区中分配一个连续存储块，并把该存储块首地址的指针强制转换成相应类型后回送给调用程序。这样便创建了一个无名的数据对象，为它在堆中分配了一个存储块。一旦产生了这个数据对象，它就一直存在，与它所相关的函数的生命期无关，仅当调用函数 free 撤销对它的存储分配，它才会消亡。

显然，由于堆式存储分配时随机地分配存储，随机地释放存储，经过一段时间后，堆区中将交错地出现正在使用的和已经释放的存储块。为了对堆区进行管理，需要研制堆管理程序（通常称为废料收集程序），当然，堆管理程序的运行需要在空间上和时间上额外开销。早期，国内外对堆管理算法的设计开展了相当多的工作，这里不拟讨论，有兴趣的读者可自行查阅相关的参考资料。

在谈及堆式存储分配问题时，有必要提醒读者在程序设计时应注意的一个问题，那就是悬空引用。

悬空引用是指对已释放的存储的引用。例如，如果已执行了函数调用 free(q)，却依然去执行下列语句：

```
printf("%d", q→inf);
```

q→inf 就是悬空引用。悬空引用是一种逻辑错误，因为对于 C 语言之类的大多数程序设计语言，被释放的存储的内容是没有定义的。当所释放的存储随后又被分配给其他数据对象时，这种悬空引用就可能引起难以理解的错误。

在避免悬空引用的同时，也应该注意不要产生无用单元而造成信息的丢失。所谓无用单元，指的是动态分配时引起的不可达到的存储字，例如，在释放单链表中的某个结点时，没有把该结点的后继结点的位置保存入指针变量中，导致后继结点无法被访问而成为无用单元。无用单元的产生往往造成程序的逻辑错误，应该避免。

概括起来，对于 C 语言，因源程序中的全局变量和静态变量的存在而采用静态存储分配策略，因函数调用设施的存在而关于局部变量采用动态的栈式存储分配策略，因存储分配函数的存在而采用动态的堆式存储分配策略。

9.3　符　号　表

9.3.1　符号表的组织

词法分析阶段引进的标识符表，它们用来登录源程序中出现的一切标识符，同时提供标识符的属性字信息，其中也可能包含标识符的作用域信息。从标识符在标识符表中登录的序号可以唯一地确定标识符。随着编译工作的进展，尤其在语义分析时，与标识符相关联的信息不断增加，标识符表扩充成了符号表。

符号表是标识符表的扩充，它不仅登录标识符，而且还登录不断增加的属性等信息。在源程

序中，每当遇到标识符便要查符号表，如果出现新的标识符或者已登录标识符的新信息，便要进行登录，可以说，符号表存在于编译的全过程，甚至在目标程序运行时，为了得到标识符等信息，还可能需要符号表。

由于在编译过程中，频繁访问符号表，合理地组织符号表，并选用高效率的查填表方式，对提高编译程序的工作效率有很大影响。

1．符号表的结构

符号表由一系列条目组成，源程序中出现的每一个标识符在符号表中有一个相应的条目。符号表的结构和条目的内容，因程序设计语言编译程序实现的不同而不同，即使同一个编译程序实现中条目的内容，也可能因标识符所关联对象的不同而不同。例如，数组数据对象所涉及的是维数、各维元素个数及数组元素的类型等信息，而对于指针数据对象，涉及的是它所指向对象的类型等信息，等等。

一般地说，符号表的条目由两部分组成，即名字栏与信息栏。

符号表条目的名字栏登录完整的标识符。由于标识符的长度可以不同，通常限定最大长度，例如 32 个字符。但为了避免浪费存储空间，可以另外设立存放标识符的字符数组，其中每个标识符后有结束标志。在名字栏中，仅给出标识符在该字符数组中的起始位置（序号），如例 9.1 中的标识符 NodeT、data、next 与 NodeType 等，如图 9-2 所示。

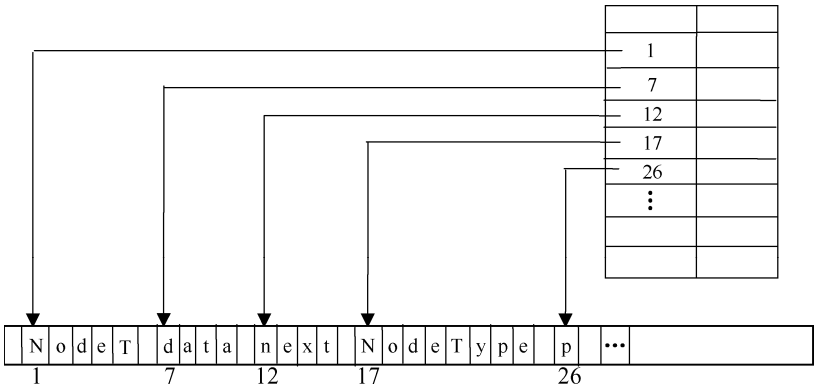


图 9-2

当然，也可以在标识符后不给出结束标志，而在符号表条目的名字栏中给出标识符的起始位置与长度。

这样，符号表条目的名字栏有一致的大小，且在大多数标识符不是太长的情况下，可以节省存储。

符号表条目的信息栏中登录与标识符相关联对象的各种信息，对于 C 语言可以包括关联对象的种类（常量、变量、数组、结构、指针与函数等）、类型等属性（整型、实型、字符型与枚举型等）和作用域信息，以及为相关联数据对象所分配的存储区域的相对地址。信息栏中的内容是在各个编译阶段逐步加入的。例如，与标识符相关联的类型及相对地址（位移量）是在语义分析阶段确定类型时填入符号表信息栏中的。

由于符号表条目的内容，根据标识符相关联的对象的类型不同而有所不同，符号表一个条目难以容纳全部的内容，为保持符号表条目结构的统一，有一致的大小，往往把与标识符相关联的构造类型等信息保存在符号表外的某处，例如数组信息表等，这些内容见下一小节的讨论。

2. 非基本类型属性的处理

非基本类型也即构造类型。C 语言中有数组类型、结构类型与指针类型等。不同的构造类型涉及不同的信息。为能从符号表一个条目取到一个构造类型的全部信息，往往把与标识符相关联的、非基本类型的信息保存在符号表外，而把相应的指针或序号填入符号表条目的信息栏内。

（1）数组类型

假定标识符 A 由说明语句“T　A[u];”说明为一维数组，它所关联的类型表达式是 array(u, T)。数组也可以是更高维的，例如：

```
T   A[u1][u2]…[un];
```

这表明，数组涉及维数、各维元素个数与数组元素类型。不言而喻，这些信息不可能全部容纳在一个条目的信息栏中。为此，通常引进所谓的数组信息表，其示意图如图 9-3 所示。数组信息表的一个条目包含维数 n、各维元素个数 d（一般情况是下界 l 与上界 u），以及数组元素类型 T 的信息。为了方便运行时刻对数组元素地址的计算，往往还把计算数组元素地址涉及的数组 A 的首地址 base 与常量 C 包含在数组信息表中。这里 base 与 C 如下地参与在数组元素 $A[j_1][j_2]\cdots[j_n]$ 的存储地址 D 的计算公式中：D=base-C+d（参见第 6 章 6.4.3）。对于 C 语言，固定地有下界 $_i$=0，上界 $_i=u_i$-1 与个数 $_i=u_i$，(i=1,2,…,n)，且 C=0，base 是 A[0][0]…[0]的存储地址。

维数		
下界 $_1$	上界 $_1$	个数 $_1$
下界 $_2$	上界 $_2$	个数 $_2$
	⋮	
下界 $_n$	上界 $_n$	个数 $_n$
首址 base		
常量 C		
数组元素类型		
⋮		

图 9-3

（2）结构类型

对于结构类型，需要取得其成员变量的信息。这些成员变量的个数是不确定的，且类型可以是任意的，特别是，又可以是结构类型或数组类型等。不言而喻，符号表一个条目是不可能容纳下所有这些信息的。

然而由于结构类型定义的特殊性，通常不必对结构类型引进结构信息表。实际上，任何成员变量标识符在符号表中都有一个相应的条目，且总是第一个成员的条目在相应结构类型（变量）的符号表条目的紧后，或最后一个成员变量的条目在相应结构类型（变量）的符号表条目的紧前，标记为成员的相继若干个条目都属于同一个结构类型。例如假定有结构变量说明如下：

```
struct
{ int class;  int number;  float score;
} c1,c2;
```

相应符号表的示意图如图 9-4 所示，其中结构变量 c1 与 c2 的条目中，信息栏的类型属性指向它的第一个成员变量的条目。

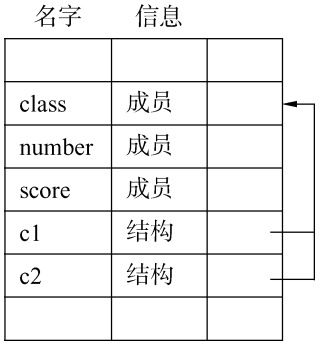


图 9-4

图 9-4 中的符号表表明：结构类型变量说明可以如同其他变量说明一样处理，并没有让结构类型作为单独的作用域，引进新的符号表，而是引进成员属性信息。在这里对于结构类型变量说明的处理方式如同一般的说明部分，因此对先前关于结构类型的翻译，应作相应的修改，请读者自行思考完成。

一个结构变量的符号表条目中，信息栏的类型属性值是结构类型第一个成员在符号表中的条目的序号或位置指针。

（3）指针类型

C 语言中，指针类型可以是指针类型标识符，也可以是由“所指向对象类型*”形式定义的新指针类型。关于指针类型变量的符号表条目，重要的是从它的信息栏能取得所指向对象的类型。

这是容易做到的，只需让指针变量的符号表条目的信息栏中，类型属性值是对象类型在符号表中条目的序号，或者是一个指针，它指向对象类型的条目。

当然，指针类型或指针类型变量的符号表条目信息栏中，种类必须是指针。

概括地说，一个指针类型变量的符号表条目中，信息栏关联对象种类是指针，属性则是所指向对象的类型的符号表条目序号或位置指针。

3. 作用域信息

源程序说明部分把标识符与具有某种类型属性的数据对象相关联，不同的标识符对应于不同的数据对象。然而同一个标识符在不同的程序位置上被说明时，将代表不同的数据对象，这就是标识符的作用域。这表示，当一个标识符使用性出现而去查符号表时，必须遵循作用域法则，才能确定正确的相应符号表条目。

在语法分析时，为每个函数定义建立一个符号表，一个程序中包含多个函数定义，将存在多个符号表，而符号表将存在于整个编译期间，甚至在运行时刻还为了提供标识符信息而继续保存。多个符号表将造成管理上的不便，合理的处理方式是对整个源程序仅建立一个符号表。这时需解决的是：如何保留作用域信息？这可考虑词法分析一章中讨论的确定作用域的算法。按这个算法，进入一个新的作用域时，设置标志进入新作用域的间隔条目，然后登录这一作用域内的局部变量的条目。当退出该作用域时，将上退去这作用域中的一切条目，并上退去上述间隔条目。当最后扫描完整个源程序时，整个符号表也成为了一个空表。这样处理的不足之处是不能把符号表保留到后面各个阶段。

可以作如下修改，即把符号表与确定作用域的栈结合起来。具体做法如下：为每一个函数给定一个唯一的序号，称函数号，所有全局变量相应的作用域也有一个序号，譬如 0。在每个函数作用域内标识符的符号表条目中添加相应的函数号，这即作用域信息。另外，当进入一个作用域而在栈中增加条目时，也往符号表中增加条目，但在出一个作用域而把栈中条目上退去时，在符号表中相应条目并不上退去，相反，增加一个包含有紧外层作用域嵌套层次的间隔条目，最终这样生成的符号表就是包含作用域信息的符号表。当然，需要对查造符号表算法作适当修改。

对于 C 语言，采纳的是静态作用域法则，按最近的嵌套作用域原则确定作用域。如果一个标识符在一个作用域内未被说明，则相继在紧外层作用域中寻找其说明，找到有关说明的作用域便是该标识符的作用域。

4. 存储分配的信息

为了方便地生成目标指令，在符号表条目的信息栏中，应包含为数据对象所分配存储的信息。

对于静态存储分配的数据对象，如果目标代码是汇编语言指令，则存储分配的工作可以让汇编程序去完成。编译程序要做的工作是：在生成了源程序的汇编语言目标代码后，扫描符号表，为每个数据对象（标识符）生成汇编语言的数据定义，并添加到所生成的目标代码上。当编译程序生成的是机器语言目标代码时，一般采用相对地址，由编译程序计算为每个数据对象分配的存储字相对于某个基址的位移量。例如，以前面所述的静态存储区的第一个存储字的地址作为基址。

对于按栈式动态存储分配策略分配在栈区的数据对象，编译程序不直接为它们进行存储分配。然而如同在说明部分的翻译中所讨论的，对于这些数据对象，语法制导定义或翻译方案已为它们确定了相对于活动记录中局部数据域首地址的位移量，只需在运行时刻调用函数时建立活动记录，确定 top_sp 的值就行，它就是局部数据域的首地址。

对于按堆式动态存储分配策略分配在堆区的数据对象，编译程序不为它们进行存储分配。分配存储的工作由运行时刻支持程序包中的运行子程序来完成，具体地说，就是由存储分配函数

malloc 与 free 等所对应的运行子程序来完成。编译程序为这些子程序提供数据类型及其所占字节数的信息，以便分配合适的存储区域和对它们正确地访问。当然，作为编译程序的组成部分，这些存储分配函数应由编译程序开发人员来实现。

一般地说，在说明部分的翻译中所讨论的翻译方案已进行了上述工作，只是基址只能在运行时刻确定。

9.3.2　符号表的数据结构

1．线性符号表

符号表最简单和最易实现的数据结构是线性表。线性表的元素，即符号表的条目是一个由名字栏和信息栏组成的结构。如前所述，名字栏可以是存放标识符的字符串，也可以是一个二元组，指明标识符在另一个标识符字符数组中的起始位置序号及标识符长度。信息栏中则登录标识符所关联的数据对象的种类、类型等属性信息以及作用域和存储分配信息等。

对符号表的操作主要是两种，即在符号表中为标识符建立条目和按标识符查找条目。可以设置一个变量 available 来指明符号表中下一个可用位置，即指向符号表中下一个条目的位置。当扫描到一个说明语句时，为定义性出现的标识符建立一个符号表新条目，登录入相应的各种信息。然后令 available 增加一个符号表条目的长度，指向下一个可用位置。当扫描到使用性出现的标识符时，进行查找符号表操作。

按照静态的最接近嵌套作用域法则，从 available 指向的位置开始，向上查找所给定标识符的条目。如果所给标识符在有相同作用域标记的条目中查到，则这个标识符是在当前作用域中定义的。如果没有找到，继续向上查找，即在当前作用域的紧外层（也就是最接近的）作用域中查找。如果还未查到，则再向上继续查找，当查到时，必定是在最接近的作用域中定义的。如果直到符号表的第一个条目，仍未查到所给标识符的条目，则所给标识符是无定义的，应给出报错信息。

线性表可以用数组来实现，也可以用链表来实现。考虑用数组实现的情况。假定符号表中共有 n 个条目，并假定符号表中所有条目的标识符出现的概率都相等，则关于某个标识符查找符号表所需的平均比较次数是(n+1)/2，即关于某标识符查找符号表时的比较次数，与符号表的长度成正比。如果考虑到建立新条目时都需向上查看是否已登录过，建立 n 个条目所需的平均比较次数是 $\frac{1}{4}$ n(n+1)，因此为建立 n 个条目，并在建立 n 个条目后查找 m 次所需的平均比较次数共计：

$$\frac{1}{4}\,n(n+1)+\frac{1}{2}\,(n+1)m=\frac{1}{4}\,(n+1)(n+2m)$$

设一次比较所需的机器时间为 t，总计用时 $\frac{1}{4}$ (n+1)(n+2m)t，当 n 与 m 较大时，所花费的时间将十分可观。

2．散列符号表

为了提高查找符号表的效率，可以考虑采用散列技术来实现符号表。

应用散列技术实现符号表的思想如下。引进一个包含 n 个元素的散列表，每个元素是指向一个条目的指针。这时设计一个散列函数 h，它把标识符 id 映射到散列表元素的序号 k：h(id)=k，序号为 k 的条目，指向的正是散列符号表中标识符 id 的条目。

应用散列技术建立符号表条目的过程如下：对于定义性出现的标识符 id，计算 h(id)，得到序号 m=h(id)，为标识符 id 建立条目，令散列表中序号为 m 的元素是指向所建立条目的指针。

按照设计思想，应该一个散列表元素指向一个条目，但很可能 $h(id_1)=m$，又有 $h(id_2)=m$，即一个序号对应于 2 个条目。这样便不得不把 id_1 与 id_2 的条目链接起来，成为一个条目链，如图 9-5 所示。

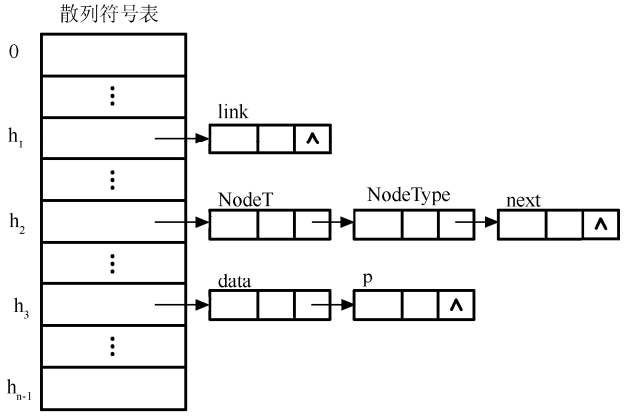


图 9-5

如果 $id_1 \neq id_2$，但 $h(id_1)=h(id_2)$，这种情况称为冲突。解决冲突可以有很多办法，这里采用了把发生冲突的条目链接起来。在查符号表时，在所指向的条目链上，逐个结点比较，找出所要查找的条目。但总起来说，应用散列技术时，必须使散列函数计算速度快，而且尽可能避免发生冲突，即，使函数值 h(id)尽可能遍历 0 到 n−1 之间的每个值。

当查符号表时，关于标识符 id，计算 h(id)，得到序号 m=h(id)，查看序号为 m 的散列表条目指向的符号表条目是否是 id 的条目。如果是，则找到，如果不是，则从链指针达到条目链中的下一个条目，查看是否是标识符 id 的条目。如此继续，直到查到或达到条目链尾而未查到。如果达到链尾时还未查到，则发出报错信息：“id 无定义”的同时，并为所查标识符 id 建立相应的结点，把它链入条目链尾，成为新链尾。

散列符号表的元素个数是固定的，但每个元素所指向的条目链的长度是可变的，因此，条目总数不是固定的，这种散列称为**开散列**。

使用散列技术，查找效率显然比线性表的效率高很多。假定散列表元素是 M 个，最多可指向 M 个条目链，而条目总共为 N 个，则条目链的平均长度是 N/M，一次查找的平均比较次数为⌊N/M⌋（⌊N/M⌋表示对 N/M 取整）。如果令 N/M 是一个较小的常数，例如 2，那么，只要散列函数的计算是快速的，效率提高是十分明显的。

9.4 运行时刻支持系统

一般地说，从源程序翻译得到的等价目标程序是一些目标模块，需要通过连接装配程序对这些模块进行连接装配，装配成完全确定了装入的存储地址的可执行机器语言程序。但是，即使连接装配后，目标程序依然不能独立地运行。显然，必须有函数入口子程序与函数出口子程序的支持，才能运行。一句话，目标程序必须有运行子程序的配合才能运行。

运行子程序是为支持目标程序运行而开发的一系列子程序，它们的全体组成所谓的运行时刻支持程序包，或称运行时刻支持系统。

编译程序所包括的运行子程序，随背景程序设计语言的不同而不同，也随编译程序实现的不

同而不同，这完全取决于程序设计语言的表达能力和编译程序的设计。总的说来，有两大类运行子程序，即面向用户的与对用户隐匿的。

面向用户的运行子程序，指的是程序书写人员（用户）知道其存在，并能使用的运行子程序。例如，程序中经常涉及计算初等函数，如 sin 与 cos，与这些函数相应的实现就是面向用户的运行子程序。这类子程序往往以系统提供的函数形式而展现在用户面前，调用这些系统函数也就是执行相应的运行子程序。用户无需自己去实现这些函数，而且以库子程序形式提供的这些运行子程序通常有较高的质量。

对用户隐匿的运行子程序，指的是程序书写人员（用户）并不了解其存在，且不能在源程序中显式应用的运行子程序。如前所述的函数入口/出口子程序就是对用户隐匿的，又如数组元素地址计算的运行子程序也是。对用户隐匿的运行子程序是为配合目标程序运行而设计的，没有它们，目标程序不能运行。但用户不需要了解它们的细节，甚至可以根本不知道它们的存在。

对于编译程序研制人员来说，不论是面向用户的还是对用户隐匿的，运行子程序的实现都是编译程序本身开发中的一个不可或缺的组成部分。

本 章 小 结

要使源程序翻译成的等价的目标代码能够运行，得到期望的效果，生成目标代码前必须考虑到运行时可能涉及的一切方面。但是，即使生成的目标代码是正确的，考虑到了各个方面，它仍是不能运行的，还必须具备支持目标代码运行的一切条件。本章讨论相关的几个主要方面，即运行时刻存储管理、符号表的组织与管理、运行时刻支持系统。

本章通过对 C 语言程序实例的分析，概括了 3 类变量，即全局变量、与函数调用紧密相关的局部变量以及由存储分配函数创建的无名变量，相应地有静态存储分配、栈式存储分配与堆式存储分配，后两类是动态存储分配。读者必须清楚地认识这 3 类存储分配策略的语言背景。

符号表是标识符表的扩充，符号表存在于编译全过程，合理地组织符号表，并选用高效率的查填表方式，对提高编译程序的工作效率有很大影响。符号表的组织涉及符号表的结构设计、非基本类型属性的处理、作用域的信息，以及存储分配的信息。通常采用线性符号表实现符号表，但采用散列符号表，只需散列函数的计算速度快，而且散列函数值能均匀分布在整个区间，减少冲突的可能，对于符号表的查找效率提高，将十分可观。

运行子程序是为支持目标程序运行而开发的一系列子程序，它们的全体组成所谓的运行时刻支持程序包，或称运行时刻支持系统。目标程序必须在运行子程序的配合支持下才能运行。运行子程序分两大类，即面向用户的与对用户隐匿的。面向用户的运行子程序，是系统提供的可为用户使用的子程序，例如计算初等函数 sin 与 cos 等的库子程序。对用户隐匿的运行子程序，例如函数入口子程序与出口子程序，这是开发实际可行的编译程序时必须考虑的方面。

相关概念有悬空引用、数组信息表、开散列。

复习思考题

1．运行一个 C 语言程序时，如何实现 3 类存储分配管理策略？

2．悬空引用是怎样引起的？应怎样避免？

3．符号表中，如何登录构造类型变量条目？

4．请简单说明你对程序设计语言运行时刻支持环境的理解。建议结合某个具体的程序设计语言。

习　　题

1．设有 C 语言函数定义如下，试说明该函数定义的功能，并指明 3 类不同种类的变量分别是哪些。

```
typedef struct CommandT
{  char code[30];
   struct CommandT  * next;
}  CommandType;
CommandType *head;

CommandType * CopyCode(CommandType *source)
{  CommandType *p, *q, *first;
   p=source;
   first=(CommandType *) malloc (sizeof( CommandType));
   strcpy( first→code, p→code);
   p=p→next;
   q=first;
   while(p)
   {  q→next=(CommandType *) malloc (sizeof( CommandType));
      q=q→next;
      strcpy( q→code, p→code);
      p=p→next;
   }
   q→next=NULL;
   return first;
}
main( )
{  CommandType *CommandP;
     …
   head=CopyCode( CommandP);
     …
}
```

2．试创建无名变量。设有语法分析树结点类型定义如下：

```
struct NodeT
{  int No;   int  info;
   struct NodeT  *next;
} ;
struct  NodeT  *head;
```

试创建由 3 个结点组成的结点链，由 head 指向第一个结点，其中，序号 No 依次为 1、2 与 3，信息 info 依次为 101、1 与 102。

第 10 章
虚拟机目标程序的解释程序的研制

本 章 导 读

前面讨论了从源程序到目标程序的整个翻译过程，然而生成的目标程序是虚拟机目标指令序列，不可能在现有的计算机上运行，为了运行，或说体验所生成目标程序，需要有一个符号模拟执行虚拟机目标程序的解释程序，本章讨论这样的解释程序的研制，内容包括：虚拟机指令操作码种类、设计要点和数据结构设计，并给出了可运行的解释程序，期望读者参考该解释程序的研制，自行设计并实现功能更为完善的解释程序。

当把源程序翻译成了等价的目标程序，不言而喻，希望能了解目标程序的正确性，尤其是希望能运行而得到期望的结果。如果目标程序是机器语言目标程序或汇编语言程序，可以在目标机上运行，然而如果是虚拟机目标程序，该如何运行呢?

一个行之有效的办法是自己动手，用 C 语言研制一个符号模拟执行虚拟机目标程序的解释程序，也就是说，并不把虚拟机目标程序转换为机器语言程序，而是研制一个解释程序，它逐条指令地模拟执行虚拟机目标程序。

下面讨论这一解释程序的设计和实现，供读者参考。

10.1 虚拟机指令操作码种类

首先明确可允许的虚拟机指令系统。虚拟机指令系统的指令格式与寻址方式参见第 6 章 6.4.2 虚拟机一节。为明确起见，分类列出操作码与指令形式如下。

输入输出类：

```
READ  x
PRINT x  或 PRINT #N
```

其中，N 是数值。

传输类：

```
MOV  x,  y
```

算术运算类：

```
op   x,  y
```

其中 op 可以是 ADD（加法）、SUB（减法）、MPY（乘法）与 DIV(除法)。

比较类：

```
CMP  x,  y
```

无条件控制转移类：

```
GOTO （N）
```

其中 N 是指令序号。

条件控制转移类：

```
CJrelop （N）
```

其中 relop 可以是<、<=、>、>=、==与!=，N 是指令序号。

其中操作数可以是简单变量、寄存器与常量 3 类，当是常量时取#N 形式，N 是常数值。当是控制转移指令时，控制转移到的指令序号用小括号对括住。数据类型仅整型。

假定有 3 个 C 语言程序片段如下，其中为了简化输入输出，把 C 语言输入输出语句形式简化，故称为 C 型语言程序。

【例 10.1】 设有十分简单的 C 型语言程序片段如下：

```
a=2; b=3; x=a+b; print x;
```

相应的虚拟机目标程序如下：

```
MOV  #2,  a
MOV  #3,  b
MOV  a,   R0
ADD  b,   R0
MOV  R0,  x
PRINT   x
```

【例 10.2】 设有 C 型语言程序片段如下：

```
read a;   read b;
if(a>b) x=(a+b)*2;  else x=(a-b)/2;
print x;
```

相应的虚拟机目标程序如下：

```
(1) READ   a                  (9)  MPY    #2,   R1
(2) READ   b                  (10) MOV    R1,   x
(3) MOV    a,   R0            (11) GOTO   (16)
(4) CMP    R0,  b             (12) MOV    a,    R2
(5) CJ>    (7)                (13) SUB    b,    R2
(6) GOTO   (12)               (14) DIV    #2,   R2
(7) MOV    a,   R1            (15) MOV    R2,   x
(8) ADD    b,   R1            (16) PRINT  x
```

【例 10.3】 设有 C 型语言程序片段如下：

```
S=0;  i=1;
while(i<=100)
{ S=S+i;
  i=i+1;
}
print  S;
```

相应的虚拟机目标程序如下：

```
(1) MOV    #0,    S           (8)  ADD    i,    R1
(2) MOV    #1,    i           (9)  MOV    R1,   S
(3) MOV    i,     R0          (10) MOV    i,    R2
(4) CMP    R0,    #100        (11) ADD    #1,   R2
(5) CJ<=   (7)                (12) MOV    R2,   i
(6) GOTO   (14)               (13) GOTO   (3)
(7) MOV    S,     R1          (14) PRINT  S
```

显然，这 3 个实例涉及了 C 语言最基本的 3 类控制成分：赋值语句、选择结构（if 语句）与

迭代结构（while 语句），且相应目标程序中的指令覆盖了虚拟机的各类指令与操作数类型。

为了研制符号模拟执行虚拟机目标程序的解释程序，首先必须明确实现思路。

实现思路十分简单。在读入目标程序指令序列后，逐条取出目标指令，确定其操作数，然后根据指令操作码模拟执行当前指令，直到目标程序模拟执行结束。控制流程示意图如图 10-1 所示。

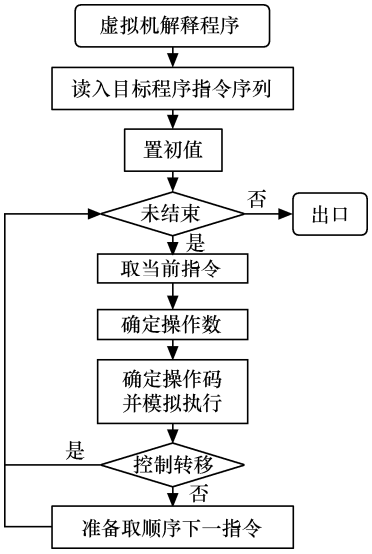


图 10-1

10.2　设 计 要 点

当实现符号模拟执行虚拟机目标程序的解释程序时，必须考虑下列几个要点：操作数的处理（包括对标识符的处理）、控制转移指令的处理以及输入输出的处理，进一步考虑操作码的确定与模拟执行。

10.2.1　操作数的处理

操作数中最主要的是变量。模拟执行虚拟机目标程序时，一个最为关键的问题是如何对变量存取值？例如，下列的虚拟机指令序列：

```
MOV   #1,   i
MOV    i,   R0
ADD   #1,   R0
MOV   R0,   i
```

并不是把这些虚拟机指令转换为相应的计算机指令，然后执行，也不是把这些虚拟机指令插入 C 语言程序中，而是通过符号模拟执行的方式来“执行”这些指令。

按照通常 C 语言程序中变量的概念，变量等同于存储字，通过变量名（标识符）来存取相应的存储字。也就是说，为变量分配存储字，把计算所得的值传送入该存储字。当引用时，通过变量名得到存储地址，从相应存储字中取出值。现在可以建立一个虚拟存储区，为变量在虚拟存储区中分配存储字。这个虚拟存储区实际上用数组来实现。为了管理方便，令该虚拟存储区元素呈

以下形式：

变量名	值

以标识符在虚拟存储区中的“存储地址”，即相应元素的序号来存取值。

当确定操作数时，发现操作数是一个变量（标识符），便查虚拟存储区，没有查到，则把它登录入虚拟存储区，不论是新登录，还是查到已登录，都得到虚拟存储区的存储地址，即一个序号，并按此序号存取值。例如，

```
MOV  #1,   i
MOV  i,    R0
```

假定 i 是第一个出现的标识符，也是第一次出现，这时序号为 1，因此把标识符 i 登录在序号为 1 的虚拟存储区元素的变量名部分。模拟执行第一条指令，把值 1 送入 i 在虚拟存储区的相应元素中，即序号为 1 的元素的值部分。模拟执行第二条指令，由于查到 i 的序号为 1，从序号为 1 的存储字中取出值，即 1，把此值送入寄存器 R0 中。

寄存器通常以 R0 与 R1 等形式表示，以字母 R 打头，为简便起见，可按标识符处理。

一条虚拟机指令一般包含两个操作数，由于总是仅从第一个操作数取值，而不可能把值赋给第一个操作数，因此不论是变量、寄存器，还是常量，都取到它的值，把它存放在解释程序安排的固定的内部变量中。计算结果将存入第二个操作数，因此需要得到第二个操作数的存储地址，即在虚拟存储区中相应元素的序号。

10.2.2 控制转移指令的处理

为了记录当前正模拟执行的指令，设置一个程序计数器，记录正模拟执行的指令序号。通常从模拟执行第一条指令开始，置初值时程序计数器中设置为 1。模拟执行完一条指令，程序计数器的值加 1，便将执行顺序下一条指令。但若模拟执行的是控制转移指令，则应在程序计数器中置好将转移到的指令序号。例如，模拟执行指令 GOTO (12)，将把值 12 置于程序计数器中，下一条被模拟执行的指令就是序号为 12 的指令。

下面考虑条件控制转移指令的模拟执行。

设有虚拟机指令序列如下：

```
(1) CMP    a,     b
(2) CJrelop  (N1)
(3) GOTO   (N2)
```

其中 relop 是< 、<=或!=等关系运算符。由于 a relop b 等价于(a－b) relop 0。例如，a>b 等价于 a－b>0，因此在指令(1)处计算 a－b，在指令(2)处判别是否(a－b) relop 0。如果是，把 N1 值传送入程序计数器，否则程序计数器的值顺次加 1，模拟执行指令(3)，把 N2 值传送入程序计数器。这样便非常简单而有效地实现了条件控制转移。

可以设置一个变量，例如 ConditionValue，来存放 CMP 指令所模拟执行的减法的值，以便模拟执行 CJrelop 指令时使用。

模拟执行控制转移指令，例如指令“CJ> (7)”，只需执行下列语句：

```
if(ConditionValue>0)
  程序计数器=opr1V;
```

这里，opr1V 中存放第一个操作数的值，这时是 7。一般地，执行：

```
if(ConditionValue relop 0)
  程序计数器=opr1V;
```

要提醒的是，模拟执行控制转移指令时，置好程序计数器的值之后，便不应再去让程序计数器的值加 1。

10.2.3　操作码的确定与模拟执行

为了确定操作码，只需设立一个操作码表，在其中登录一切可允许的操作码。例如，如表 10.1 所示。关于当前模拟执行的虚拟机指令的操作码 op，查找此操作码表，可得到一个序号 OpNo。执行下列 switch 语句可快速模拟执行虚拟机指令：

表 10.1

0	
1	READ
2	MOV
3	GOTO
4	PRINT
5	ADD
6	SUB
7	MPY
8	DIV
9	CMP
10	CJ<
11	CJ<=
12	CJ>
13	CJ>=
14	CJ==
15	CJ!=
16	

```
switch(OpNo)
{ case 1:  /* READ */
    ⋮
  case 2:  /* MOV */
    ⋮
  case 15: /* CJ!=*/
    ⋮
}
```

下面以算术运算为例，进一步说明如何模拟执行的处理思想。

操作数 1 可能是变量、寄存器，也可能是形如#N 的常量，而操作数 2 可能是变量或寄存器。不论哪一种情况，确定算术运算的两个操作数后，都把值分别放入变量 opr1V 与 opr2V 中。因此在关于 OpNo 的 switch 语句中，有如下形式的 C 语言程序片段：

```
case 5: case 6: case 7: case 8:
  switch(OpNo)
  { case 5:  /* ADD */
      v=opr2V+opr1V;  break;
    case 6:  /* SUB */
      v=opr2V-opr1V;  break;
    case 7:  /* MPY */
      v=opr2V*opr1V;  break;
    case 8:  /* DIV */
      v=opr2V/opr1V;  break;
  }
  把 v 送入到第二个操作数所相应的虚拟存储区存储字中;
  break;
```

10.2.4　输入输出指令的处理

一个程序没有输入输出，将不能检查它的执行是否正确。为此引进简单的输入与输出指令，形如：

```
READ   x
PRINT  x
```

输入时的操作数必是变量（标识符），假定它在虚拟存储区中相应存储字的序号为 N，则只需执行下列 C 语言语句：

```
printf("请输入%s 的值: ", 虚拟存储区[N].变量名);
scanf("%d", &v);
虚拟存储区[N].值=v;
```

为所显示的变量名，输入一个值。为显示变量名，更简单的是执行下列语句：

```
printf("请输入%s 的值: ", opr1);
```

因为第一个操作数的符号名总是存放在 opr1 中。

输出时情况类似，确定了变量（标识符）在虚拟存储区中相应存储字的序号，假定为 N，只需执行下列 C 语言语句：

```
printf("%s=%d\n", 虚拟存储区[N].变量名,虚拟存储区[N].值);
```

这样在看到输出值的同时，可以看到是什么变量的值。

如果允许输出常量值，仅作微小修改便可。

10.3 数据结构设计

一个应用程序，在总体设计时，必要的是：在考虑控制结构的同时考虑数据结构。综上所述，符号模拟执行虚拟机目标程序的解释程序的数据结构有如下一些：目标程序、虚拟存储区、操作码表以及程序计数器。

关于目标程序的数据结构可设计如下：

```
char 目标程序[最大程序长度][最大指令长度];
```

为了在调试时不因小错误或键入错误而频繁地重新键入目标程序，可利用 C 语言置初值设施，例如，

```
char 目标程序[最大程序长度][最大指令长度]
   ={ "\0",
      "READ  a", "READ  b", "MOV  a,  R0", …, "PRINT x",
      "\0"
    };
```

让指令序号从 1 开始，相应地有记录虚拟机指令总数的变量。

关于虚拟存储区的数据结构，可设计为一个结构数组如下：

```
struct
{ char 变量名[8];  int  值;
} 虚拟存储区[最大虚拟区长度];
```

相应地有一个记录虚拟存储区长度的变量，记录标识符的个数。

操作码表的数据结构可设计为一个字符串数组，且用置初值方式给定如下：

```
char 操作码表[ ][8]
 ={ "\0",
    "READ",  "MOV", …, "CJ!=",
    "\0"
  };
```

为了简化处理，可以引进下列变量：

```
char op[8], opr1[8], opr2[8];   /* 一条虚拟机指令的分解 */
int  Opr1Pos, Opr2Pos;          /* 变量操作数在虚拟存储区中的存储字(序号) */
int  opr1V, opr2V;              /* 进行运算的两个操作数的值 */
```

10.4 符号模拟执行虚拟机目标程序的解释程序

下面给出符号模拟执行虚拟机目标程序的解释程序，它可以在 Turbo C 2.0 环境下运行，其中

一些数据的取名与前面所述的有些不同，但并不影响阅读。

所给程序仅供参考，建议读者先自行设计和实现这一解释程序，也建议读者进行改进，例如对虚拟机目标程序中错误的处理、数据类型的扩充、寻址方式的扩充，以及操作种类的进一步充实与完善等。

尽管这一解释程序的实现比较简单，没有明确结合词法分析与语法分析等，但读者可以从这程序的设计与实现中，体会模拟执行一个符号程序的实现思想和技术。

符号模拟执行虚拟机目标程序的解释程序如下。

```
int  NumofCommands, CurrentCommandNo/*程序计数器*/ ;
char op[8], opr1[8], opr2[8];  /* 当前虚拟机指令的分解 */
struct
{ char name[8]; int value; }
ValueT[30];              /* 变量表，即虚拟存储器 */
int  LengthofValueT;         /* 变量个数 */

char OPT[20][8]=          /* 操作码表 */
{ "0",
  "READ", "MOV", "GOTO", "PRINT",
  "ADD",  "SUB", "MPY",  "DIV",
  "CMP",  "CJ<", "CJ<=", "CJ>", "CJ>=", "CJ==", "CJ!=",
  "0"
};

char Commands[50][20]  /* 目标程序，最多 50 条指令，指令 20 个字符长*/
={ "\0",
   "MOV   #0,   S",
   "MOV   #1,   i",
   "MOV   i,    R0",
   "CMP   R0,   #100",
   "CJ<=  (7)",
   "GOTO  (14)",
   "MOV   S,    R0",
   "ADD   i,    R0",
   "MOV   R0,   S",
   "MOV   i,    R1",
   "ADD   #1,   R1",
   "MOV   R1,   i",
   "GOTO  (3)",
   "PRINT  S",
   "PRINT  #7777",
   "\0"
 };
/* 第二个实例 */
/* ={ "\0",
      "MOV   #2,   a",
      "MOV   #3,   b",
      "MOV   a,    R0",
      "ADD   b,    R0",
      "MOV   R0,   x",
      "PRINT  x",
      "\0"
   };
```

```
*/
/* 第三个实例 */
/*  ={ "\0",
       "READ   a",
       "READ   b",
       "MOV    a,    R0",
       "CMP    R0,   b",
       "CJ>    (7)",
       "GOTO   (12)",
       "MOV    a,    R1",
       "ADD    b,    R1",
       "MPY    #2,   R1",
       "MOV    R1,   x",
       "GOTO   (16)",
       "MOV    a,    R1",
       "SUB    b,    R1",
       "DIV    #2,   R1",
       "MOV    R1,   x",
       "PRINT  x",
       "\0"
        };
*/

main( )
{  int opr1V, opr2V, v, ConditionValue;  /* 模拟执行时的操作数值*/
   int Opr1Pos, Opr2Pos;  /* 两个操作数的存储字（序号） */
   int OpNo,i,j,k,m,n;
   char  Const[8];

   i=1;
   while(Commands[i][0]) i++;
   NumofCommands=i-1;

   CurrentCommandNo=1;

   while(CurrentCommandNo<=NumofCommands)
   { i=CurrentCommandNo;
     /* 取得当前指令，在 op,opr1 与 opr2 中 */
     j=0; k=0;
     while(Commands[i][k]!=' ')
       op[j++]=Commands[i][k++];
     op[j]='\0';
     while(Commands[i][k]==' ') k++;
     j=0;
     while(Commands[i][k]!='\0' && Commands[i][k]!=',')
       opr1[j++]=Commands[i][k++];
     opr1[j]='\0';   k++;
     while(Commands[i][k]!='\0' && Commands[i][k]==' ') k++;
     j=0;
     while(Commands[i][k]!='\0' && Commands[i][k])
       opr2[j++]=Commands[i][k++];
     opr2[j]='\0';
     /* 处理两个操作数 */
     if(opr1[0]=='#')
     { n=0; j=1;
```

```
    while(opr1[j])
       Const[n++]=opr1[j++];
    Const[n]='\0';
    opr1V=atoi(Const);
  } else
  if(opr1[0]=='(')
  { n=0; j=1;
    while(opr1[j]!=')')
      Const[n++]=opr1[j++];
    Const[n]='\0';
    opr1V=atoi(Const);
  } else
  { j=1;
    while(j<=LengthofValueT && strcmp(ValueT[j].name, opr1)) j++;
    if(j>LengthofValueT)
    { strcpy(ValueT[j].name,opr1);
      LengthofValueT++;
    }
    opr1V=ValueT[j].value;
    Opr1Pos=j;
}
if(opr2[0]=='#')
{ n=0; j=1;
  while(opr2[j])
     Const[n++]=opr2[j++];
  Const[n]='\0';
  opr2V=atoi(Const);
} else
{ j=1;
  while(j<=LengthofValueT && strcmp(ValueT[j].name, opr2)) j++;
  if(j>LengthofValueT)
  { strcpy(ValueT[j].name,opr2);
    LengthofValueT++;
  }
  Opr2Pos=j;
}
/* 分析并模拟执行指令 */
/* 确定 OP */
OpNo=1;
while(OpNo<=15 && strcmp(OPT[OpNo], op))
  OpNo++;
/*  分析操作码,并模拟执行  */
switch(OpNo)
{ case 1:  /* READ */
      printf("Input a value:");  scanf("%d", &v);
       ValueT[Opr1Pos].value=v;
      break;
  case 2:  /* MOV */
      ValueT[Opr2Pos].value=opr1V;
      break;
  case 3:  /* GOTO */
      CurrentCommandNo=opr1V;
      continue;
  case 4:  /* PRINT */
     if(opr1[0]!='#')
```

```
                printf("%s=%d\n", ValueT[Opr1Pos].name, opr1V);
            else
                printf("%d\n",  opr1V);
            break;
        case 5: case 6: case 7: case 8:
            if(opr2[0]!='#')
               opr2V=ValueT[Opr2Pos].value;
            switch(OpNo)
            { case 5:  /* ADD  */
                v=opr2V+opr1V;
                break;
              case 6:  /*  SUB  */
                v=opr2V-opr1V;
                break;
              case 7:  /*  MPY  */
                v=opr2V*opr1V;
                break;
              case 8:  /* DIV  */
                v=opr2V/opr1V;
                break;
            }
            ValueT[Opr2Pos].value=v;
            break;
        case 9:  /* CMP  */
           if(opr2[0]!='#')
             opr2V=ValueT[Opr2Pos].value;
           ConditionValue=opr1V-opr2V;
           break;
        case 10:  /* CJ<  */
           if(ConditionValue<0)
           { CurrentCommandNo=opr1V;  continue;  }
           break;
        case 11:  /* CJ<= */
           if(ConditionValue<=0)
           { CurrentCommandNo=opr1V;  continue;  }
           break;
        case 12:  /* CJ> */
           if(ConditionValue>0)
           { CurrentCommandNo=opr1V;  continue;  }
           break;
        case 13:  /* CJ>= */
           if(ConditionValue>=0)
           { CurrentCommandNo=opr1V;  continue;  }
           break;
        case 14:  /* CJ== */
           if(ConditionValue==0)
           { CurrentCommandNo=opr1V;  continue;  }
           break;
        case 15:  /* CJ!= */
          if(ConditionValue!=0)
          { CurrentCommandNo=opr1V;  continue;  }
          break;
      }
      CurrentCommandNo++;
    }
  }
```

本 章 小 结

当生成了虚拟机目标程序，如何检查目标程序的正确性？行之有效的办法是研制一个符号模拟执行虚拟机目标程序的解释程序。本章讨论了解释程序的实现思路，并讨论了实现要点，即操作数的处理（包括对标识符的处理）、控制转移指令的处理以及输入输出的处理，进一步考虑操作码的确定与模拟执行。通过本章的学习，将可理解一般情况下，执行符号程序的实现思路和处理方法。

复习思考题

1．在模拟执行一个符号程序时，如何处理操作数，特别是标识符？
2．在模拟执行一个符号程序时，如何处理控制转移指令？
3．如何快速模拟执行虚拟机符号指令？
4．本章所给的解释程序，在哪些方面可以改进？如何改进？

参考文献

[1] 张幸儿. 计算机编译原理. 第 3 版. 北京：科学出版社，2008.

[2] 张幸儿. 计算机编译原理——编译程序构造实践. 第 2 版. 北京：科学出版社，2009.

[3] 张幸儿. 编译原理 编译程序构造与实践. 北京：机械工业出版社，2008.