

编译原理讲义

王宝库 张俐 朱靖波 王会珍 马安香 肖桐
东北大学计算机学院编译原理教学团队

V1.0

2/21/2023

Copyright © 2001-2023

东北大学计算机学院编译原理教学团队

本讲义遵循以下开源协议

Licensed under the Creative Commons AttributionNonCommercial 4.0 Unported License (the "License"). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc/4.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "as is" basis, without warranties or conditions of any kind, either express or implied. See the License for the specific language governing permissions and limitations under the License.

本讲义的主要内容来自：

1. 王宝库、张俐的《编译原理》课件
2. 朱靖波、肖桐的《编译原理》教学视频
3. 教学团队的教学笔记、与学生互动的内容

特别感谢参与本讲义编写、排版、校对的人员：马安香、王会珍、阮俊豪、常开妍、范瑀纯、黄鹏程、寇凯淇、周航，刘沛灼、赫洱锋、蒙龙、张靖男、黄灿安

前言

编译原理是计算机及相关专业的专业基础课程之一。作为一个长期从事编译原理教学和实践的团队，我们很高兴能有机会把我们的教学内容进行分享，供同学们参考。本讲义最初由王宝库老师编写，之后经过张俐老师多次修改，形成了相对完整的版本。此后，被东北大学多位教师在教学中采用，并在此基础上开展了大量的实践。本讲义尽可能用通俗易懂的方式介绍编译器设计所涉及的理论、方法、算法和实现技巧，适合初学者使用。但是，程序设计语言和编译技术的发展十分迅速，本讲义的部分内容也会与时俱进，不断更新，欢迎大家提出宝贵意见！

肖桐、朱靖波

目录

1	编译程序的基本概念	1
1.1	什么是编译程序?	2
1.2	编译程序结构	3
1.3	编译程序的实现机制	5
1.4	编译过程实例	5
1.5	本章小结	8
2	形式语言基础	9
2.1	形式语言是符号串集合	9
2.1.1	符号串(集合)的运算	10
2.1.2	符号串集合的文法描述	12
2.2	形式语言由文法定义的	14
2.2.1	什么是文法?	14
2.2.2	文法是怎样定义语言的?	14
2.3	主要语法成分的定义	16
2.3.1	文法的运算问题	16
2.3.2	句型、句子和语法树	18
2.3.3	短语、简单短语和句柄	20
2.4	两种特性文法	21
2.4.1	递归文法	22
2.4.2	二义性文法	22
2.5	文法的等价变换	23
2.5.1	文法的等价性	23

2.5.2	文法变换方法	23
2.5.3	文法变换方法 1	24
2.6	形式语言的分类	28
3	自动机基础	29
3.1	正规语言及其描述方法	29
3.1.1	正规语言的正规式表示法	30
3.1.2	正规语言的有限自动机表示法	30
3.2	有限自动机的定义与分类	33
3.2.1	有限自动机的定义	33
3.2.2	有限自动机怎样描述语言	33
3.2.3	有限自动机的两种表现形式	35
3.2.4	有限自动机的分类	38
3.3	有限自动机的等价变换	38
3.3.1	有限自动机的确定化	39
3.3.2	有限自动机的最小化	43
3.4	正规语言描述方法间的相互转换	50
3.5	有限状态自动机的实现问题	54
3.5.1	控制程序设计	55
3.5.2	变换表存贮结构设计	56
4	词法分析	59
4.1	词法分析的基本概念	59
4.1.1	单词的分类与识别	59
4.1.2	单词的机内表示	61
4.2	词法分析程序的设计	62
4.2.1	词法分析程序功能划分	62
4.2.2	一个简单词法分析器的实现	63
4.2.3	词法分析示例	66
4.3	算数常数处理机设计	67
4.3.1	识别器设计	68

4.3.2	翻译器设计	68
5	语法分析	73
5.1	语法分析的基本概念	73
5.2	递归子程序法	75
5.2.1	递归子程序法示例	76
5.2.2	递归子程序构造方法	77
5.2.3	递归子程序法适用范围	79
5.3	LL(1) 文法定义	80
5.4	LL(1) 分析法的完整流程	82
5.4.1	抽象的流程表示	87
5.5	LL(1) 文法及其判定	87
5.5.1	首符号集合、后继符集合与选择符集合	88
5.5.2	LL(1) 文法及其判定	90
5.6	LL(1) 分析器设计 (实现)	91
5.6.1	LL(1) 分析表的构造	91
5.7	LR() 分析法的介绍	93
5.7.1	LR() 分析法的“统治地位”	93
5.7.2	LR() 分析法的定义	94
5.8	LR(0) 分析器设计	97
5.8.1	LR(0) 文法及其判定	98
5.8.2	LR(0) 分析表构造	98
5.8.3	LR(0) 控制程序设计	99
5.9	项目集和可归约前缀图	99
5.9.1	扩展文法	100
5.9.2	由扩展文法构造可归约前缀图 (句柄识别器)	100
5.9.3	由可归约前缀图构造 LR(0) 分析表	101
5.9.4	LR(0) 分析法过程示例	101
5.9.5	LR(0) 分析法实例	102
5.10	LR(0) 分析法的扩展	103

5.11	SLR(1) 分析法的扩展	105
5.11.1	扩展文法	106
5.11.2	构造可归约前缀图	106
5.12	简单优先分析法基本概念	108
5.12.1	什么是简单优先分析法	108
5.12.2	简单优先分析过程示例	108
5.12.3	文法符号之间的优先关系	109
5.13	简单优先分析器设计	113
5.13.1	简单优先文法及其判定	114
5.13.2	简单优先分析矩阵分析表构造	114
5.13.3	简单优先控制程序设计	115
5.14	算符优先分析	115
5.14.1	算符文法	115
5.14.2	头符号集合和尾符号集合	116
5.14.3	算符优先关系定义	116
5.14.4	算符优先文法	116
6	符号表	119
6.1	符号表的地位和作用	119
6.1.1	符号表的定义	119
6.1.2	标识符的四种语义信息	120
6.1.3	符号表的基本功能	120
6.2	符号表的组织与管理	121
6.2.1	符号表的工作原理	121
6.2.2	符号表的查询、访问方式	121
6.2.3	符号表的维护、管理方式	121
6.3	符号表的结构设计	122
6.3.1	符号表总表 (SYNBL)	123
6.3.2	类型表 (TYPEL)	124
6.3.3	数组表 (AINFL)	124

6.3.4	结构表 (RINFL)	125
6.3.5	函数表 (PFINFL)——过程或函数语义信息	125
6.3.6	其他表 (...)	126
6.4	符号表的构造过程示例	126
6.5	运行时刻存储分配	131
6.5.1	标识符值单元分配	131
6.5.2	活动记录	132
6.5.3	简单的栈式存储分配	134
6.5.4	嵌套过程语言的栈式存储分配	137
7	中间代码生成	145
7.1	导入	145
7.1.1	为什么要研究语义分析	145
7.1.2	为什么需要生成中间代码	145
7.2	中间代码生成	146
7.2.1	常用的中间代码形式	146
7.2.2	各种语法成分的中间代码设计	147
7.3	中间代码翻译算法	155
7.3.1	属性文法	155
7.3.2	语法制导翻译技术	156
7.3.3	四元式翻译文法设计扩展 1	159
7.4	中间代码翻译的实现	163
7.4.1	递归子程序翻译法	163
7.4.2	LL(1) 翻译法	165
7.4.3	LR() 翻译法	166
7.4.4	算符优先翻译法	168
7.4.5	翻译文法的变换问题	169
8	优化处理	171
8.1	优化的分类	171
8.1.1	与机器无关的优化	171

8.1.2	与机器有关的优化	171
8.2	常见的几种局部优化方法	172
8.2.1	常值表达式节省（常数合并）	172
8.2.2	公共表达式节省（删除多余运算）	172
8.2.3	删除无用赋值	172
8.2.4	不变表达式外提（循环优化之一：把循环不变运算提到循环外）	172
8.2.5	消减运算强度（循环优化之二：把运算强度大的运算换算成强度小的运算）	173
8.3	局部优化算法探讨	173
8.3.1	基本块划分算法	173
8.3.2	局部优化示例	175
8.4	基于 DAG 的局部优化方法	176
8.4.1	四元式序列的 DAG 表示	177
8.4.2	基于 DAG 的局部优化算法	180
9	目标代码及其生成	185
9.1	目标代码生成的基本问题	185
9.1.1	目标代码选择	185
9.1.2	变量的活跃信息	189
9.1.3	寄存器的分配问题	192
9.1.4	目标代码生成问题	195
9.2	目标代码生成算法设计	198
9.2.1	目标代码生成要点和生成环境	202
9.2.2	表达式四元式目标代码生成算法	203
9.2.3	赋值四元式目标代码生成算法	203
9.2.4	条件语句四元式目标代码生成算法	204
9.2.5	循环语句四元式目标代码生成算法	206
9.3	一个简单代码生成器的实现	208

第 1 章 编译程序的基本概念

计算机语言的层次结构如图 1.1 所示：上层是大家编写的高级程序语言比如：C 语言、C++ 或 java 语言。中间会有一层汇编语言，不同目标机的汇编语言不一样，最底层机器执行的指令，汇编语言离机器指令已经非常非常近了，但它还不是真正的机器指令，你可以把机器指令理解成 0、1 的码，编译程序需要把高级语言转化成汇编语言，汇编程序再把汇编语言翻译成真正的目标指令。编译器所做的就是红色的部分。如果不经过汇编语言也可以直接把高级语言翻译为机器语言。或者大家经常也会遇到解释性程序，解释程序和编译程序做一样的工作，也是把高级语言翻译成目标机的指令，只不过解释程序是逐条翻译，写一句就执行一句，而编译程序是整体翻译。解释性语言比如：python、shell、perl。同样我们也可以把 java 程序转化为 python 的程序，这个叫做转换程序。当然也存在反汇编程序和反编译程序。

为了更好的解释交叉汇编程序，我们举个例子。比如，当需要在 window 环境下生成在 Linux 环境下可执行的程序。正常的做法是，如果需要的 window 环境下的可执行程序，就将源代码在 window 环境下编译然后生成可执行程序在 windows 环境下执行，如果需要的是 Linux 环境下的可执行程序，就将源代码在 Linux 环境下编译然后生成可执行程序在 Linux 环境下执行。由于可执行程序是跟操作系统以及具体的硬件和底层的指令是相关的。而所谓的交叉汇编程序是指，能够在 windows 环境下生成一个在 Linux 环境下可执行的程序，但是这个可执行程序在 windows 环境下是无法执行的。在某些特殊的场景，是需要这么去做的。当一个真实所需的环境搭建起来代价比较高，我们可以在一个简单容易搭建的环境下，去生成一个特定环境下的可执行程序。

计算机中语言的层次体系

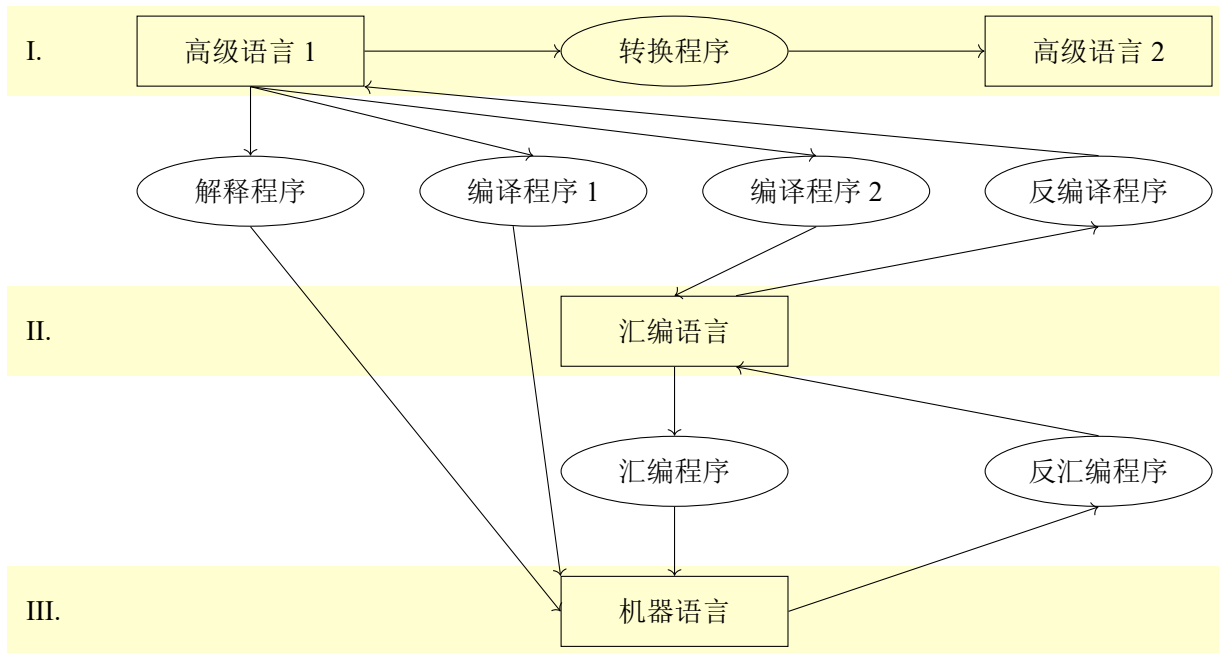


图 1.1: 计算机中语言的层次体系

1.1 什么是编译程序？

编译程序（compiler）是一种翻译程序，它特指把某种高级程序设计语言翻译成具体计算机上的低级程序设计语言。

高级语言执行过程有两个阶段：第一个阶段叫做编译的阶段：从源语言经过编译程序生成目标语言。这个目标语言就是要生成的语言，如果我们要把汇编语言做为目标语言的话，那么汇编语言就是编译的结果，编译器做的工作就是把高级语言转化为汇编语言。拿到一段程序后，运行程序时可能与外界有数据的交互，最终通过这些交互会把程序的结果输出出来。这门课程给大家介绍的是编译程序这部分。

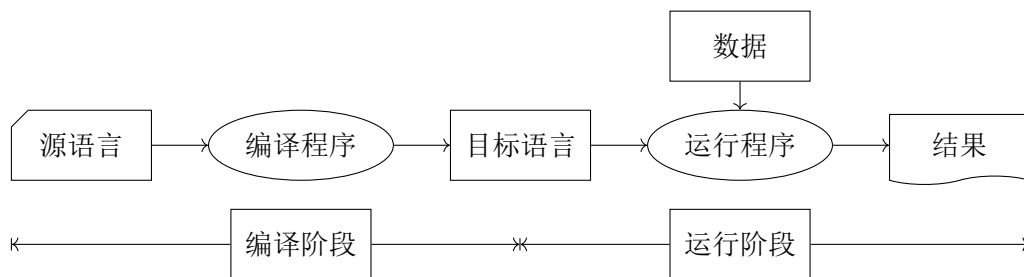


图 1.2: 高级语言的执行过程

※ 什么是解释程序？

解释程序（interpreter）也是一种翻译程序，将某高级语翻译成具体计算机上的低级程序设计语言。

解释程序会逐条读取代码，读取之后就会把代码翻译成目标程序，直接进行执行，这个过程会反复执行，所以在执行 perl 的时候不会产生目标代码。

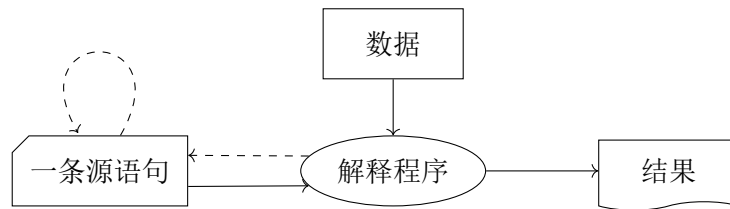


图 1.3: 解释程序的执行过程

编译程序和解释程序有两个重要的区别：

1. 前者有目标程序而后者无目标程序，但是解释性程序执行比较慢；
2. 前者运行效率高而后者便于人机对话，这种解释性程序一句一句执行便于调试。

编译程序和解释程序的目都是为了帮助用户执行程序，实现二者的算法是类似的。

1.2 编译程序结构

编译程序包括词法分析、语法分析、语义分析、优化处理和代码生成五个阶段，如图 1.4 所示。

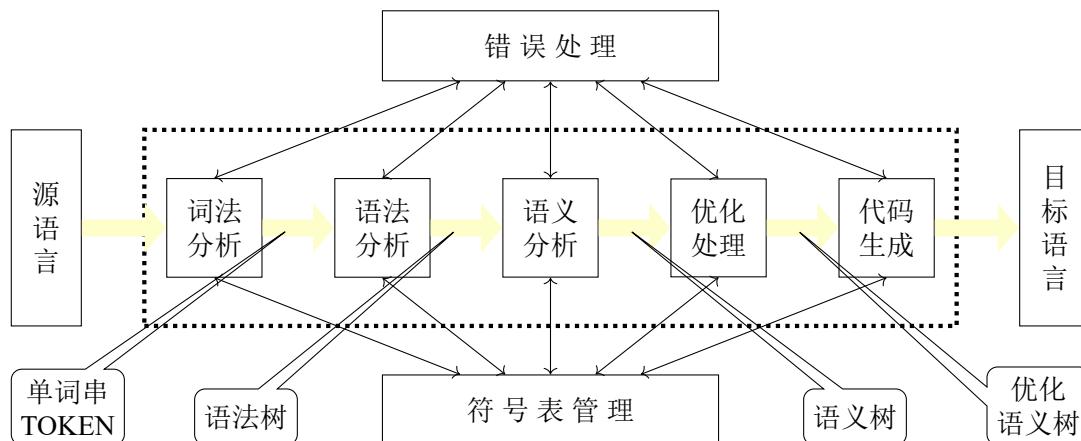


图 1.4: 编译程序总体结构框图

上图中左侧表示输入的源程序，右侧表示经过编译处理得到的目标程序。比如输入 C 语言源程序，经过编译处理，输出该源程序对应的汇编语言或者是可执行的机器代码。其中会包含若干个步骤分别是：词法分析、语法分析、语义分析、优化处理、代码生成。这五个步骤构成了编译器最核心的部件。首先拿到源语言会进行词法分析生成单词 $TOKEN$ 串，词法分析的作用是识别在编译中什么是一个词；拿到了 $TOKEN$ 串后生成语法树，语法分析是最重要的内容，语法分析的作用是告诉我们一句话的结构是什么，比如中文的主谓宾定状补；之后我们对语法分析的结果进行语义分析，语义分析告诉我们每一句高级语言的句子是什么意思，举个例子： $a = b$ 代表的是把 b 的值赋值给 a 这个变量，据此我们可以得到中间代码；中间代码并不是最高效的，中间步骤可能会存在冗余，我们要执行优化处理得到优化后的代码；最终优化后的代码会生成我们的目标代码，也就是我们所谓的汇编语言程序或者机器的指令。

这些步骤的执行中有两个问题需要注意：第一个问题：待编译的程序出错了怎么办？如何告诉我们错误在哪？错误处理模块就是负责这部分工作，所有编译程序里都有错误处理模块。第二个问题：符号表管理，我要知道这个程序里有哪些函数，每个函数有哪些形参和局部变量，每个变量都是什么类型，存在什么位置等？比如在 C 语言中变量声明语句 *inta*，我们是如何知道 *a* 被定义为一个整形的变量？C 语言是典型的强数据类型的语言，这种语言需要在你使用变量之前对变量的类型进行定义，而且这个类型在 C 语言中是不允许修改的，但像 *python* 就是弱数据类型的语言，对变量的类型不需要定义。这些变量及其语义信息（如变量的数据类型和存储位置等）都存放在符号表里。

※ 编译程序与机器翻译的类比：

我们用自然语言的处理来类比一下计算机的编译过程。

机器翻译是指计算机把一种自然语言翻译成另一种自然语言。下面通过汉译英示例分析机器翻译的过程。

例 1.1 我们用树叶和颜料能够制作美丽的图画

1. 词法分析：首先分词, 图中 r 代表代词,p 代表介词,n 代表名词。

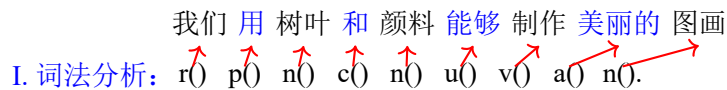


图 1.5: 一个简单的翻译过程例子

2. 句法分析：句子的结构用树来表示，称之为句法树，描述了整个句子的结构。

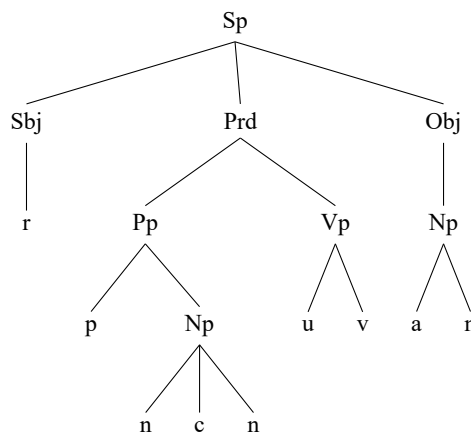


图 1.6: 语法树

3. 语义分析：除了这个结构之外，我们还要知道这个句子表达什么动作执行了什么事情。

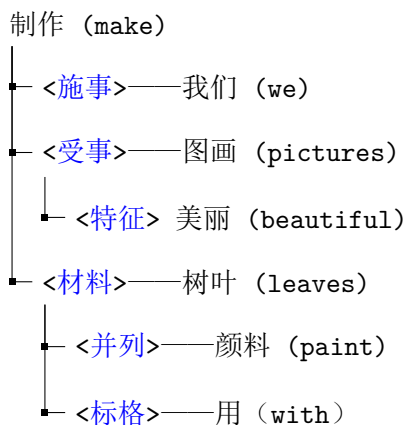


图 1.7: 语义网

4. 优化处理:

5. 目标生成: We can make beautiful pictures with leaves and paint.

在上述英译汉的各个阶段中, 我们需要一些词典和知识库的支持, 例如汉英词典等。

1.3 编译程序的实现机制

类比机器翻译, 我们是不是也可以做高级程序设计语言的翻译呢? 在做编译程序的时候有一个概念叫遍。

定义 1.1 遍: 编译程序对源程序或等价程序从头至尾扫描的次数。

一般来说我们现在使用的编译器是两遍的编译器:

第一遍: 词法分析、语法分析和语义分析;

第二遍: 目标代码生成和目标代码优化。

当一遍中包含若干阶段时, 各阶段的工作是穿插进行的。由于语法分析在整个编译器中起到核心的作用, 我们会提到语法制导的翻译或语法制导的编译。所谓语法制导的编译, 用语法分析来做引导的一种编译器。例如: 使语法分析器处于核心位置。当语法分析需要下一个单词时, 就调用扫描器, 识别一个单词并生成相应 Token; 一旦识别出一个语法单位时, 就调用中间语言生成器, 完成相应的语义分析并产生中间代码。

1.4 编译过程实例

Listing 1.1: Pascal 程序片段

```

1   Var $a, b$: integer;
2       ...
3       B := a + 2 * 5
  
```

编译过程如下:

1. 词法分析：识别单词并分类

单词类码

- (1) 关键字 (k) —var, integer;
- (2) 标识符 (i) —a, b;
- (3) 常数 (c) —2, 5;
- (4) 界符 (p) —, ; := +

图 1.8: 词法分析

2. 语法分析：组词成句及语法错误检查

例： $b := a + 2 * 5$ 的分析过程如下所示：

算数表达式的层次结构

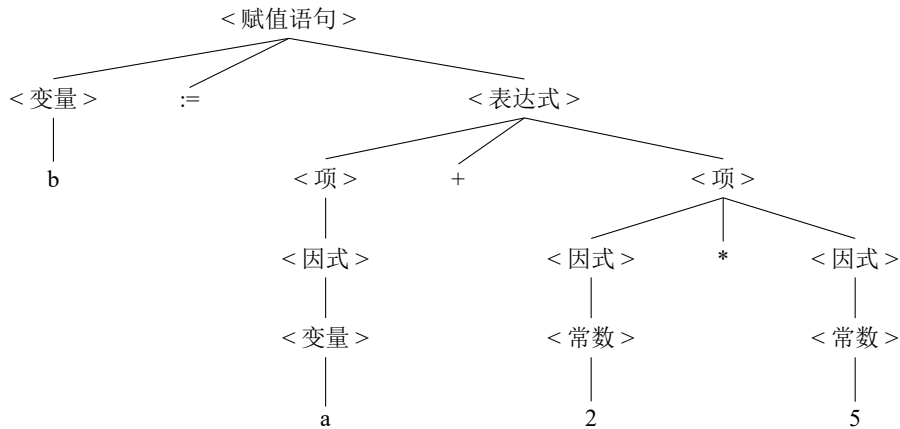


图 1.9: 赋值语句 $b:=a+2*5$ 的语法树

画出这个句子的语法树；第一步知道这个句子是一个赋值语句，第二步知道把表达式赋值给变量，表达式由两部分组成，第一项是一个由变量 a 构成的因式，第二项是由常数 2 因式和常数 5 因式构成。这些就构成了整个赋值语句。算数表达式的层次结构为：常数、因式、项。

上述句法分析告诉我们这个句子是什么样的结构

3. 语义分析：分析各种语法成分的语义特征

构建标识符的语义词典——符号表:

符号表

名字	类型	种类	地址
a	i	v	
b	i	v	

```
var a,b:integer;
...
b:=a+2*5;
```

数据区

a 的值
b 的值

图 1.10: 构建符号表

我们要得到句子的语义动作信息，这个动作是什么？ a 是什么？ b 是什么？ a 变量的值放在物理内存的哪个位置？符号表会告诉我们这些事情。

如: $b:=a+2*5$

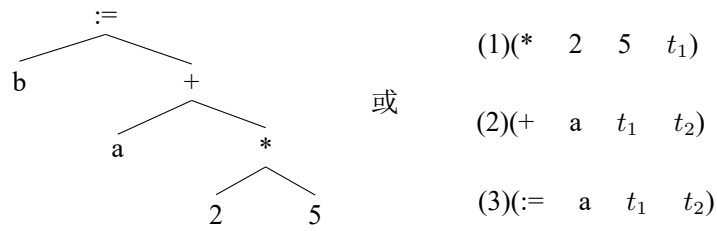


图 1.11: 语法树

之后我们会把上述赋值语句写成这种树的形式，我们就知道了这个式子的意思是把 2 和 5 相乘再与 a 相加最后赋值给 b 。当然我们也可以把这个树写成四元式的形式，第一元表示算符，第二元和第三元是运算的对象，最后一元是运算的结果。

4. 优化：提高目标程序质量的工作

例: $b := a + 2 * 5$

经常数合并，可分别获得优化后的中间代码：

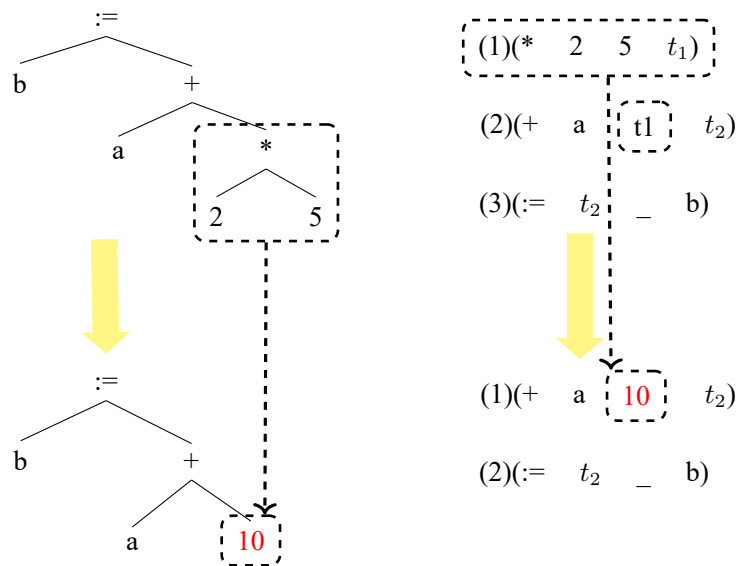


图 1.12: 优化后的中间代码

对计算机来说不必反复计算 $2*5$ 这个过程，经过优化四元式由三条变为两条，这个是运行前的优化，而寄存器优化是负责运行时的优化。

1. 目标代码生成：产生计算机可识别的语言

通常，是把中间语言转换成目标代码，把之前的四元式生成汇编语言

(1)(+ a 10 t₂)

(2)(:= t₂ _ b)



(1)LD R , a

(2)ADD R , 10

(3)ST R , b

图 1.13: 生成汇编语言

1.5 本章小结

本章介绍了编译原理的课程目标，即设计编译器以实现高级程序语言的理解、分析和处理，同时为解决其他复杂工程问题提供一种新思路。给出编译程序的定义及其逻辑结构，并分析了编译程序实现机制，最后通过实例分析编译过程。

第2章 形式语言基础

计算机处理语言，首先应考虑语言的形式化、规范化，使其具有可计算性和可操作性，这就是形式语言理论研究的问题。

形式语言诞生于1956年，由chomsky创立。语言研究涉及三个方面：语法、语义和语用；我们侧重语法的研究。

※ 形式语言的基本观点是：语言是符号串之集合！（语言就是一句一句话说）

※ 形式语言理论研究的基本问题是：研究符号串集合的表示方法、结构特性以及运算规律。

2.1 形式语言是符号串集合

定义 2.1 形式语言：字母表上的符号，按一定的规则组成的所有符号串集合。其中的每个符号串称为句子。

定义 2.2 字母表：元素（符号）的非空有限集合；（构成字符串的最基本符号不能再切割了）。

定义 2.3 符号串：符号的有限序列。

在讨论符号串时，注意是基于某个字母表的，例如对于0, 1构成的字母表，其上的符号串可以是010, 0101110, ……

定义 2.4 符号串集合：有限个或者无限个符号串组成的集合。

定义 2.5 规则：以某种形式表达的在一定范围内共同遵守的规章和制度。这里指符号串的组成规则。

形式语言概念示例：

例 2.1 $L_1 = \{00, 01, 10, 11\}$;

字母表有 $\Sigma_1 = \{0, 1\}$;

句子有：00,01,10,11

例 2.2 $L_2 = \{ab^m c, b^n \mid m > 0, n \geq 0\}$;

字母表有 $\Sigma_2 = \{a, b, c\}$;

句型 1: $ab^m c$,

有句子：abc, abbc, abbbc, …

句型 2: b^n ,

有句子: $\varepsilon, b, bb, bbb, \dots$

注意:

1. 句型不代表某一个具体的串, 代表一类串;
2. m 是大于 0 的, n 是大于等于 0 的;
3. $b^0 = \varepsilon$ (空符号串), $b^1 = b, b^2 = bb, b^3 = bbb, \dots$;
4. L_1 为有限语言, L_2 为无限语言, 包括无限多个符号串。

2.1.1 符号串(集合)的运算

符号串的运算:

定义 2.6 连接: $\alpha \cdot \beta = \alpha\beta$ 如 $a \cdot b = ab$

连接: 设有 x, y 两个符号串, 它们的连接操作是把它们连在一起, 结果是 xy 。举个例子, $x = ab, y = c$, 则连接后的符号串为 $xy = abc$ 。

特例: 如果一个字符串和一个空字符串连接的话, 其结果为自己本身, 即: $x\varepsilon = \varepsilon x = x$

定义 2.7 或: $\alpha | \beta = \alpha$ (或者 β)

定义 2.8 方幂: $\alpha^n = \prod_{i=1}^n \alpha = \alpha\alpha^{n-1} = \alpha^{n-1}\alpha$

$\alpha^0 = \varepsilon$ (空符号串)

$\alpha^1 = \alpha; \alpha^2 = \alpha\alpha; \dots$

方幂: 设 x 是一个符号串, 则 x 的方幂 $x^2 = xx, x^3 = x^2x, \dots, x^n = x^{n-1}x$ 。

特例: $x^1 = x, x^0 = \varepsilon$

定义 2.9 闭包: α 的正闭包: $\alpha^+ = \alpha^1 | \alpha^2 | \dots | \alpha^n | \dots$

α 的星闭包: $\alpha^* = \alpha^0 | \alpha^1 | \alpha^2 | \dots | \alpha^n | \dots$

闭包: 设 x 是一个符号串, 则 x 的闭包 $x^* = x^0 | x^1 | x^2 | x^3 | \dots | x^n | \dots$ 其中: ‘|’ 是元符号(描述符号), 意义为: 或者是, 即: x^* 是一种泛指。

※ 符号串运算示例

例 2.3 1. $abc \cdot de = abcde$

2. $ab | cd = ab$ (或者 cd)

3. $(a | b)^1 = (a | b) = a | b$

$(a | b)^2 = (a | b)(a | b) = aa | ab | ba | bb$

$(a | b)^* = (a | b)^0 | (a | b)^1 | (a | b)^2 | \dots$

即: $(a | b)^* = (a | b)^n, n \geq 0$

a 或 b 的星闭包表示由 a 或 b 所构成的串（包含空串），如果不包含空串则写成下面的形式。

$$(a | b)^+ = (a | b)^n, n > 0$$

符号串集合的运算

设 A 和 B 为两个符号串集合，则：

定义 2.10 乘积： $AB = \{xy | x \in A \text{ 且 } y \in B\}$

例如： $U = \{a, b\} V = \{c\}$ 则 $UV = \{ac, bc\}$

特例： $\{\varepsilon\} \cdot U = U \cdot \{\varepsilon\} = U \emptyset \cdot U = U \cdot \emptyset = \emptyset$

定义 2.11 和： $A \cup B = A + B = \{x | x \in A \text{ 或 } x \in B\}$

找到 A 或者 B 中的元素取并集

定义 2.12 方幂： $A^n = \prod_n A = AA^{n-1} = A^{n-1}A$

$$A^0 = \{\varepsilon\}$$

$$A^1 = A; A^2 = AA; A^3 = AAA; \dots$$

方幂是若干个集合进行乘积，所谓方幂就是 $x^n = x^{n-1}x$ 。

特例： $x^1 = x, x^0 = \varepsilon$

定义 2.13 闭包： A 的正闭包： $A^+ = A^1 \cup A^2 \cup \dots \cup A^n \cup \dots$

$$A \text{ 的星闭包: } A^* = A^0 \cup A^1 \cup A^2 \cup \dots \cup A^n \cup \dots$$

把 A 的一次幂到 n 次幂求并集叫做集合 A 的正闭包，如果考虑 A 的 0 次幂，包含空串的集合称为 A 的星闭包。

符号串集合运算示例：

例 2.4 设 $A = \{a, b\}, B = \{c, d\}$

$$\text{则 } A + B = \{a, b, c, d\}$$

$$\text{则 } AB = \{xy | x \in A, y \in B\} = \{ac, ad, bc, bd\}$$

例 2.5 设 $A = \{a\}$

$$\text{则 } A^* = A^0 \cup A^1 \cup A^2 \cup \dots \cup A^n \cup \dots$$

$$= \{\varepsilon\} + \{a\} + \{aa\} + \{aaa\} + \dots$$

$$= \{\varepsilon, a, aa, aaa, \dots\}$$

$$= \{a^n | n \geq 0\}$$

假设 A 只包含一个符号串 a ，通过并集运算得到一个包含空串和只由 a 构成的符号串的集合，它可以看作字母 a 的一个语言。

我们把 A 扩展为包含 a 和 b 的集合，那 A 的星闭包是什么呢？

例 2.6 设 $A = \{a, b\}$, $A^* = ?$

$$\because A^* = A^0 \cup A^1 \cup A^2 \cup \dots \cup A^n \cup \dots$$

$$A^0 = \{\varepsilon\}$$

$$A^1 = A = \{a, b\}$$

$$A^2 = A \cdot A = \{a, b\} \cdot \{a, b\} = \{aa, ab, ba, bb\}$$

$$A^3 = A \cdot A^2 = \{a, b\} \cdot \{aa, ab, ba, bb\} = \{aaa, aab, aba, abb, baa, bab, bba, bbb\}$$

由上述分析可知, A 的星闭包由 a 和 b 构成的所有符号串所组成的集合, 包括空串。

推论: 若 A 为任一字母表, 则 A^* 就是该字母表上的所有符号串 (包括空串) 的集合。

如果把 a 和 b 看成字母表中的两个元素, A 的星闭包就是以 a 和 b 这两个符号构成的语言, 当然这里面是包含空串的。 A 的星闭包或者 A 的加闭包实际上是在定义语言, 给一个字母表, 这个操作就可以把字母表所对应的所有符号串都生成出来。

2.1.2 符号串集合的文法描述

前面已经讨论了语言就是符号串的集合, 也介绍了集合的基本操作。但是如果给你一个语言, 不管是机器语言还是 C++ 语言, 我们如何去描述他呢? 这是一个比较深刻的问题。我们需要一种能刻画客观世界规律性的描述方式。从我们计算机现在的手段来讲, 一般用一些可数学化描述的方式把客观事物描述出来。比如算法或者数据结构, 包括前面讲的形式语言自动机, 归根结底还是一种数学化的描述。我们现在说符号串的集合表示一种语言, 那么我们如何去描述这种语言呢? 其中一种非常重要的工具就叫文法。这个概念很重要! 计算机中很多问题都可以用文法来描述, 甚至很多你用的事务都可以用文法来描述, 比如统计符号串, 脚本语言中正则表达式的匹配, 实际上都是一种文法的体现。那么什么是文法呢?

例 2.7 $L = \{ab^n c \mid n \geq 0\}$

字母表: $\Sigma = \{a, b, c\}$

展开: $L = \{ac, abc, abbc, abbbc, \dots\}$

假如我们现在有这样一个语言, 这个语言包含字符串的形式是 a 、 b 的 n 次幂和 c 的乘积 ($n \geq 0$), 他表示的是字母表上 abc 这三个字母所构成的一个语言, 这个语言所包括的符号串模式是, 一个 a 和一个 c , 中间夹了零个或若干个 b , 对于这样一个语言如果我们用文法来描述他的话, 应该是什么样子呢? 我们用如下几条文法规则来描述上面语言, 文法规则也叫产生式规则, 这些规则就构成了上述语言生成的规则。

文法规则

$$S \rightarrow aAc$$

$$A \rightarrow bA \mid \varepsilon$$

其中 S 和 A 可以理解为一些抽象的变量, 如果把 S 看成句子的话, A 就是句子里的一个短语, 后面将 S 和 A 定义为非终结符; a 、 b 、 c 是字母表中的字母, 这里用 ε 表示空符号串, 箭头表示推导或产生的意思, 左部产生右部, $|$ 表示或的关系。第二组规则也可以写成两条规则: $A \rightarrow bA, A \rightarrow \varepsilon$ 。

怎样利用上述文法规则表示语言 L ?

从开始符号出发, 对符号串中的定义对象 (非终结符), 采用推导的方法 (用其规则右部替换左部) 产生新的符号串, 如此进行, 直到新符号串中不再出现定义的对象为止, 则最终的符号串就是一个句子。

给定下述文法规则, 我们分析其所定义的语言:

文法规则

$$S \rightarrow aAc$$

$$A \rightarrow bA \mid \varepsilon$$

句子产生过程 (\Rightarrow 推导算符):

1. $S \Rightarrow aAc \Rightarrow a\varepsilon c = ac$
2. $S \Rightarrow aAc \Rightarrow abAc \Rightarrow ab\varepsilon c = abc$
3. $S \Rightarrow aAc \Rightarrow abAc \Rightarrow abbAc \Rightarrow abbc$

...

$$\therefore S \stackrel{\pm}{\Rightarrow} ab^n c, n \geq 0$$

(1) 假定 S 是起始符号, S 推出 aAc , 再把 A 替换为 ε , 得到了 $a\varepsilon c$, 最终得到了 ac

(2) 假定 S 是起始符号, S 推出 aAc , 把 A 替换为 bA , 再把 A 替换为 ε , 得到了 $ab\varepsilon c$, 最终得到了 abc

(3) 假定 S 是起始符号, S 推出 aAc , 把 A 替换为 bA , 把 A 替换为 bA , 再把 A 替换为 ε , 得到了 $abb\varepsilon c$, 最终得到了 $abbc$

以此类推, 可以得到 $S \stackrel{\pm}{\Rightarrow} ab^n c, n \geq 0$, \Rightarrow 表示只经过一次推导, $+$ 表示至少经过一次推导, $\stackrel{\pm}{\Rightarrow}$ 代表连续推导。

2.2 形式语言由文法定义的

2.2.1 什么是文法?

我们这门课重点讨论上下文无关文法，上下文无关文法可以理解为规则均是由一个非终结符构成的左部推导出右部，这样的文法都可以称之为上下文无关文法。

定义 2.14 文法(grammar)是规则的有限集,其中的上下文无关文法可以定义为四元组: $G(Z) = (V_N, V_T, Z, P)$

V_N : 非终结符集(定义的对象集,如:语法成分等);后面语法树中的中间节点都是非终结符,从计算的角度看,可以把非终结符看作变量,可以被替换的

V_T : 终结符集(字母表)

Z : 开始符号(研究范畴中,最大的定义对象);整个推导开始的状态

P : 规则集(又称产生式集);规则集包含若干个产生式,每个产生式左部由单一的非终结符构成,右部是由终结符和非终结符构成的任意符号串,形如: $A \rightarrow \alpha$ 或者 $A \rightarrow \alpha | \beta$ 。

其中,描述符号: \rightarrow (定义为), $|$ (或者是),文法符号: $Z, A \in V_N, \alpha, \beta \in (V_N + V_T)^*$

Z 和 A 都属于非终结符集; $V_N + V_T$ 表示这个集合中的元素要么是终结符要么是非终结符, $V_N + V_T$ 的星闭包代表以这个集合里面元素构成的所有符号串,即把 $V_N + V_T$ 集合作为字母表所构成的所有符号串。

2.2.2 文法是怎样定义语言的?

定义 2.15 设有文法 $G(Z)$, $L(G)$ 为 G 所定义的语言,则 $L(G) = \{x | Z \stackrel{\pm}{\Rightarrow} x, x \in V_T^*\}$

有了文法我们如何定义语言呢?一个语言就是由这个文法所生成的字符串构成的。 $G(Z)$ 是一个文法, $L(G)$ 为 $G(Z)$ 所定义的语言,他是一个由若干个元素构成的集合,元素是从文法其实符号 Z 经过加推导出的由终结符构成的符号串 x 。这个定义包含两个层面,第一,起始符是文法开始符号 Z , 推导一定是由 Z 开始的;第二,语言中的符号串 x 一定是终结符串,不能含有非终结符,如果含有非终结符还要继续推导,直到这个串中没有非终结符,这样才能形成最终的语言。

即:一个文法所定义的语言,是由该文法开始符号推导出的所有仅含终结符的符号串之集合。其中的每个符号串皆称为句子。

定义 2.16 标识符:字母开头的字母、数字序列。

例 2.8 标识符的文法。

标识符在我们书写程序的时候会使用,声明一个变量,或者写一个函数就会用到标识符,标识符是不可以以数字开头的。现在标识符的定义也不仅仅包含字母和数字,下划线也可以作为标识符的一部分。

标识符文法的非终结符包括两个部分: I (标识符), A (标识符尾)

终结符包括两个部分： ℓ (字母), d (数字)

起始符: I

规则集: $I \rightarrow \ell A \mid \ell$

$A \rightarrow \ell A \mid dA \mid \varepsilon$

不难看出, 上述文法定义的符号串是一个以字母开头的字母和数字构成的序列。

例 2.9 无符号整数文法

能否写出一个无符号整数的文法? 可以写成这样的形式 $N \rightarrow dN \mid d$, 其中 d 表示 0-9 的数字。这个文法很特殊, N 既出现在了文法的左部又出现在了文法的右部, 这就构成了递归的形式, 递归是描述无限语言非常有效的方式。无符号整数的元素非常多, 用这种方式就可以描述无限个无符号的整数。

上述无符号整数文法规则是否存在问题? 请读者自行分析。

※ 标识符文法所定义的语言求解:

标识符文法是如何表达标识符的呢? 如果想看这个文法表示的是什么内容, 那要知道这个文法生成的句子(符号串)是什么样子。单独从文法规则是看不出来的, 我们需要进行一些简单的运算。

上面构造的标识符文法属于正规文法(定义在后)类, 正确性检验较容易; 下面给出一个算法:

$$\left. \begin{array}{l} \text{令: } I = \ell A \mid \ell \\ A = \ell A \mid dA \mid \varepsilon \end{array} \right\} \Rightarrow \text{正规方程式}$$

※ 求解 I 值步骤:

这个等式不是代数上的等式, 而是符号串的替换: I 可以用 ℓA 或 ℓ 替换。我们要求解 I 是什么, 从起始符号 A 开始, 它可以得到 ℓA 或 dA 或 ε , 那 A 是什么呢?

$A = (\ell \mid d)A, A = (\ell \mid d)^2A, \dots, A = (\ell \mid d)^n A$ 。 $A = \ell A \mid dA$ 对应 $A = (\ell \mid d)A$, 这个等式左边右边都有变量, 那右边的变量我们可以用同样的 A 进行替换。我们就可以生成 $A = (\ell \mid d)^2 A$, 同样也可以生成 $A = (\ell \mid d)^n A$, 只有当 A 推出 ε 时停止, 已经全是终结符串了, 最终得到 $A = (\ell \mid d)^n A, n \geq 0$, 当 $n = 0$ 时包含了 A 推出 ε 的情况。再把 A 代入 $I = \ell A \mid \ell$ 中得到了 $I = \ell(\ell \mid d)^n, n \geq 0$, 这是标识符串, 起始是字母, 后面是由字母和数字构成的符号串。

例 2.10 简单算术表达式文法 (这个文法务必记牢)

这个文法的非终结符包含 E(算术表达式)、T(项)、F(因式)

终结符包含 i (变量或常数), $+$, $-$, $*$, $/$, $(,)$

起始符是 E, 由 E 开始推导

规则

规则: $E \rightarrow T \mid E + T \mid E - T$
 $T \rightarrow F \mid T * F \mid T / F$
 $F \rightarrow i \mid (E)$

此文法定义了算术表达式的层次嵌套结构:

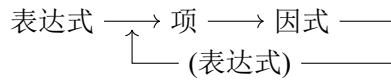


图 2.1: 算术表达式的层次嵌套结构

定义了优先级, 在推导的时候先推 +、-, 后推 *、/, 从运算的角度, 因式先做, 实际上对应的是规约的过程, 当然这个因式可以包括表达式, 可以无限嵌套下去。

算术表达式文法应用示例:

证明 $i * (i + i - i)$ 是文法 $G(E)$ 的一个句子 (即合法的算术表达式):

根据语言定义式2.15, 合法的算术表达式是指:

$E \stackrel{\pm}{\Rightarrow} i * (i + i - i)$ 成立吗?

首先 i 可能代表的是一个数字、标识符、变量, 不管 i 代表的是什么, 我们把 i 当成一个终结符来看, 我们想证明这个符号串是否是表达式文法定义的句子。

语言由起始符开始经过若干次推导得到的终结字符串, 这样的串构成的集合称之为语言, 反过来说, 如果我们要证明这个串是语言中的一个句子, 那就看能不能从文法起始符经过加推导得到这个符号串。

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * (E) \Rightarrow T * (E - T) \Rightarrow T * (E + T - T) \Rightarrow i * (E + T - T) \Rightarrow \dots \Rightarrow i * (i + i - i)$$

经过上述推导, 可以看出符号串 $i * (i + i - i)$ 是表达式文法定义的句子。

$\therefore E \stackrel{\pm}{\Rightarrow} i * (i + i - i)$

2.3 主要语法成分的定义

2.3.1 文法的运算问题

文法有一些基本的运算。文法有两种基本运算: 推导, 规约。

设 $x, y \in (V_N + V_T)^*$, $A \rightarrow \alpha \in P$

假设 x 和 y 是由终结符和非终结符构成的符号串, $A \rightarrow \alpha$ 是一条规则。

定义 2.17 直接推导 (\Rightarrow): $xAy \Rightarrow x\alpha y$

这个过程称之为使用上述规则完成的一个推导, 将符号串中的一个非终结符 A 替换成所对

应规则的右部 α ，这个过程称之为直接推导。 \Rightarrow 称为直接推导算符。

即：指用产生式的右部符号串替换左部非终结符。

定义 2.18 加推导 ($\xRightarrow{+}$)：一个符号串经过上述直接推导若干次得到的一个新的串，简记为 $\alpha \xRightarrow{+} \beta$ ， $\xRightarrow{+}$ 称为加推导算符。加推导需要至少一次推导。

即：指一步或一步以上的直接推导运算。

定义 2.19 星推导 ($\xRightarrow{*}$)， α 经过若干次推导得到 β ，也可以一步都不推导。 α 可以星推导得到 α ，但是 α 不能加推导得到 α 。

即：指零步或零步以上的直接推导运算。

定义 2.20 直接规约 (\Rightarrow)： $x\alpha y \Rightarrow xAy$

推导对应的是自上而下的分析，归约对应自下而上的分析。可以把规约看作是推导的逆运算， \Rightarrow 称之为直接归约算符。

即：直接归约是直接推导的逆运算，用产生式的左部非终结符替换右部符号串。

定义 2.21 加归约 ($\xRightarrow{+}$) 经过若干次归约，若干次左部替换右部的操作，加归约规定至少归约一次， $\xRightarrow{+}$ 为加归约算符。

即：指一步或一步以上的直接归约运算。

定义 2.22 星归约 ($\xRightarrow{*}$)，若干次的直接归约，可以是零次。

即：指零步或零步以上的直接归约运算。

定义 2.23 最左推导 ($\xRightarrow{+}_l$)——每次推导皆最左非终结符优先。

定义 2.24 最左归约 ($\xRightarrow{+}_l$)——每次规约皆最左可归约串优先。

为什么会定义最左推导和最左归约，最左推导和一般的推到相比有什么好处？最左归约和一般的归约相比有什么好处？读者可以思考一下。

例 2.11 给定一个符号串 $i + i * i$ ：

算数表达式文法

$$G(E) : E \rightarrow T \mid E + T \mid E - T \mid T * F \mid T / F \mid i \mid (E)$$

最左推导（从开始符号出发）过程：

$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow i + T \Rightarrow i + T * F \Rightarrow i + F * F \Rightarrow i + i * F \Rightarrow i + i * i$$

$$\therefore E \xRightarrow{+}_l i + i * i$$

最左归约（从符号串出发）过程：

$$i + i * i \Rightarrow F + i * i \Rightarrow T + i * i \Rightarrow E + i * i \Rightarrow E + F * i \Rightarrow E + T * i \Rightarrow E + T * F \Rightarrow E + T \Rightarrow E$$

$$\therefore i + i * i \xRightarrow{+}_l E$$

既然推导和归约是相逆的过程，那最左推导和最左归约是不是一个互逆的过程？实际上最左推导和最左归约并不互逆。

2.3.2 句型、句子和语法树

设有文法 $G(Z) = (V_N, V_T, Z, P)$ ，则：

定义 2.25 句型：若有 $Z \xRightarrow{*} \alpha$ ，则 α 是句型

α 既可以包含终结符也可以包含非终结符。

定义 2.26 句子：若有 $Z \xRightarrow{*} \alpha$ ，且 $\alpha \in V_T^*$ ，则 α 是句子。即句子是由开始符号加推导出的任一终结符号串。

句子必须都由终结符构成。

定义 2.27 语法树：句型（句子）产生过程的一种树结构表示。

树根——开始符号；树叶——给定的句型（句子）。

树对应了整个推导的过程，反过来说，在上下文无关文法的体系下，一个语法树就对应了一个推导，而且在一些限定条件下，他会唯一对应一个推导。

语法树的构造算法：

1. 置树根为开始符号；
2. 若采用了推导产生式： $A \rightarrow x_1x_2 \dots x_n$ ，则有子树：

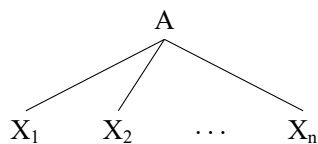


图 2.2: 推导产生式的子树

3. 重复步骤 (2)，直到再没有推导产生式为止

根节点是文法的起始符号，产生式的左部作为子树的树根，产生式的右部作为子树的树叶，重复上述过程，直到没有非终结符可以生成子树了，最终就生成了一个句子。把树的叶子节点称之为这个推导所对应的句型（句子）。

※ 如此构造的语法树，其全体树叶（自左至右）恰好是给定的句型（句子）。

※ 句型、句子和语法树示例：

例 2.12 1. 证明 $(T/F + F) * i$ 是一个句型（表达式型）

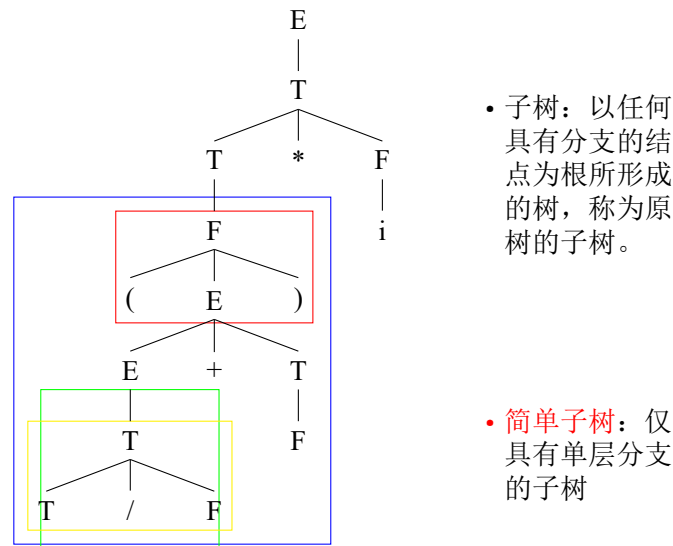
2. 画出该句型的语法树。

算数表达式文法

$$E \rightarrow T \mid E + T \mid E - T \mid T \rightarrow F \mid T * F \mid T / F \mid T \rightarrow i \mid (E)$$

1. 证明 $(T/F + F) * i$ 是一个句型（表达式型）
2. 画出该句型的语法树。

证明： $*$ 句型 $(T/F + F) * i$ 的语法树构造：



- 子树：以任何具有分支的结点为根所形成的树，称为原树的子树。
- 简单子树：仅具有单层分支的子树

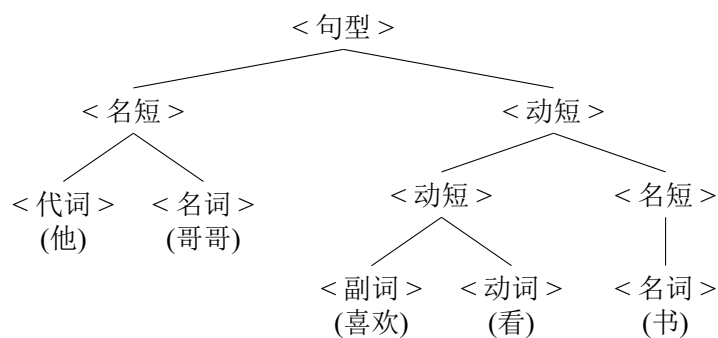
图 2.3: 语法树构造

$$E \Rightarrow T \Rightarrow T * F \Rightarrow F * F \Rightarrow (E) * F \Rightarrow (E + T) * F \Rightarrow (T + T) * F \Rightarrow (T/F + T) * F \Rightarrow (T/F + F) * F \Rightarrow (T/F + F) * i$$

- 定义 2.28** 子树：以任何具有分支的结点为根所形成的树，称为原树的子树。
- 定义 2.29** 简单子树：仅具有单层分支的子树。

需要注意的是，2.3中红色圈出的部分不是子树，而是树片段，子树的叶子必须是原树的叶子。蓝色和绿色圈出的部分也不是简单子树，黄色圈出的子树才是简单子树。

例 2.13 图 2.4 为一个中文句型的语法树：



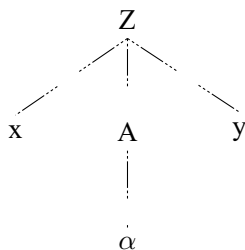
- **短语** – 他哥哥 <名短>, 喜欢看 <动短>, 书 <名词>, 喜欢看书 <动短>, 他哥哥喜欢看书 <句子>
- **简单短语** – 他哥哥, 喜欢看, 书
- **句柄** – 他哥哥 (最左边的简单短语!)

图 2.4: 中文句型的语法树

为什么要定义句柄? 请读者思考这个问题。

2.3.3 短语、简单短语和句柄

* 设文法 $G(Z)$, $x y$ 是一个句型, 则:

图 2.5: $G(Z)$ 的语法树

定义 2.30 短语——若 $Z \stackrel{\pm}{\Rightarrow} xAy \stackrel{\pm}{\Rightarrow} x\alpha y$, 则 α 是句型 $x\alpha y$ 关于 A 的短语 (A 是 α 的名字);

* 任一子树的**树叶全体**(具有共同祖先的叶节点符号串) 皆为**短语**;

定义 2.31 简单短语——若 $Z \stackrel{\pm}{\Rightarrow} xAy \Rightarrow x\alpha y$, 则 α 是句型 $x\alpha y$ 关于 A 的短语 (A 是 α 的名字);

简单短语是通过一步推导出来的 * 任一子树的**树叶全体**(具有共同父亲的叶节点符号串) 皆为**简单短语**;

定义 2.32 句柄——一个句型的最左简单短语称为该句型的**句柄**

例 2.14 图 2.5 为一个算术表达式 (型) 的语法树:

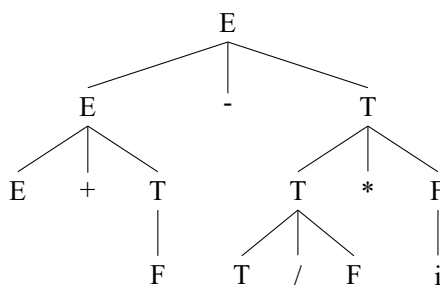


图 2.6: 算术表达式 (型) 的语法树

句型: $E + F - T / F * i$

短语: $E + F - T / F * i, E + F, F, T / F * i, T / F, i$

简单短语: $F, T / F, i$

句柄: F

* 一类典型的综合例题:

例 2.15 $G(S) : S \rightarrow aAcBe; A \rightarrow Ab|b; B \rightarrow d$

* 给定符号串 $\alpha: aAbcde$

1. 证明 $\alpha = aAbcde$ 是一个句型;
 2. 画出句型 α 的语法树;
 3. 指出 α 中的短语、简单短语和句柄
1. 证明 α 是由起始符号可以推导出的一个符号串就可以了

(a) $S \Rightarrow aAcBe \Rightarrow aAbcBe \Rightarrow aAbcde$

(b) 语法树如右图:

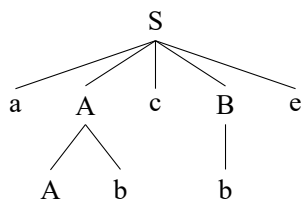
(c) 短语、简单短语和句柄:

短语: $aAbcde, Ab, d$

简单短语: Ab, d

句柄: Ab

2. 每使用一条规则就把他所对应的子树片段画出来, 一层一层画, 就可以把这个树画出来

图 2.7: $G(S)$ 的语法树

2.4 两种特性文法

设有文法: $G(Z) = (V_N, V_T, Z, P)$

2.4.1 递归文法

定义 2.33 设 $A \in V_N, x, y \in (V_N + V_T)^*$, 则; 若 $A \xRightarrow{\pm} xAy$; 称文法具有**递归性**;

A 是一个非终结符, 若 A 经过若干次推导可以得到 xAy , x 和 y 是由终结符和非终结符构成的集合上的符号串, 则称 A 所在的文法具有递归性。 A 经过若干次推导, 推导得到的中间结果里面仍然含有 A , 就是 A 又生成了 A 。如果在上下文无关文法中, A 可以无限的推导, 这样我们得到了无限生成字符串的手段。

特别: 若 $A \rightarrow A\alpha$, 称文法具有**直接左递归性**;

$A \rightarrow \alpha A$, 称文法具有**直接右递归性**。

如果文法中存在这样的产生式, 其左部和右部都含有非终结符 A , 并且 A 在产生式右部的最左端, 称这样的文法叫直接左递归文法。同理, 如果产生式在左部和右部同时含有非终结符 A , 并且 A 在产生式右部的最右端, 称这样的文法叫直接右递归文法。

如:

直接左递归文法

$G1(S) : S \rightarrow Sb|a$

直接右递归文法

$G2(S) : S \rightarrow bS|a$

$G1(S)$ 和 $G2(S)$ 是正则文法

* 递归文法是定义无限语言的工具 (递归文法定义的语言有无限个句子)!!

2.4.2 二义性文法

定义 2.34 若文法中存在这样的句型, 它具有两棵不同的语法树, 则称该文法是二义性文法。

一个文法中含有一个句型, 这个句型具有两棵及以上不同的语法树, 即可以用不同的语法树来生成同一个句型, 我们就称这个文法是二义性文法。

例 2.16 算术表达式的另一种文法:

算术表达式的另一种文法

$G'(E) : E \rightarrow E + E | E - E | E * E | E / E | (E) \mid i; i(\text{变量或常数})$

这个算不算是一个算术表达式的文法?

\therefore 句型 $i + i * i$ 有两棵不同的语法树 (右图):

左边的语法树先计算乘号, 右边的语法树先计算加号。上述文法有没有考虑加减乘除的优先级? 该文法没有考虑, 这导致对于同一个表达式会有不同的分析。

如果用【例 2.10】给出的算术表达式文法, 就不存在二义性。

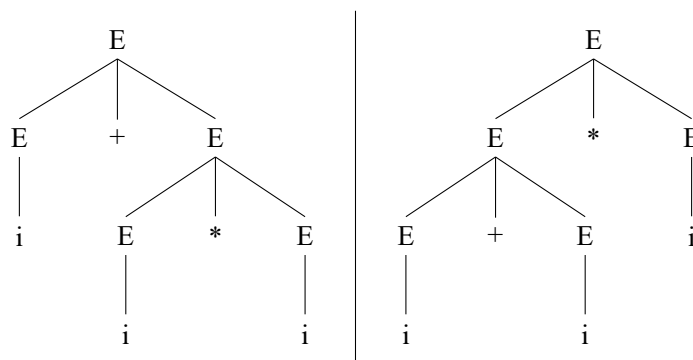


图 2.8: 不同的语法树

$\therefore G'(E)$ 是二义性文法。二义性文法会引起歧义，应尽量避免之！

2.5 文法的等价变换

2.5.1 文法的等价性

变换的依据是文法的等价性

定义 2.35 设 G_1 、 G_2 是两个文法，若 $L(G_1)=L(G_2)$ ，则称 G_1 与 G_2 等价，记作 $G_1 \equiv G_2$ 。

如果文法 G_1 和 G_2 生成的语言是完全相同的，那么 G_1 和 G_2 这两个文法就是等价的。
即：文法的等价性是指他们所定义的语言是一样的

例 2.17 G_1 和 G_2 分别对应两个文法， G_1 是一个直接右递归文法， G_2 是一个直接左递归文法

文法规则

$G_1 : S \rightarrow aS|a;$

$G_2 : S \rightarrow Sa|a$

$L(G_1) = \{a, aa, aaa, \dots\} = \{a^n \mid n \geq 1\}$

$L(G_2) = \{a, aa, aaa, \dots\} = \{a^n \mid n \geq 1\}$

$\therefore L(G_1) = L(G_2)$ 即 $G_1 \equiv G_2$

【注】 一个语言，其描述文法并不唯一。

一个语言可以对应无数组文法描述，从这个角度说，如果一个文法描述了一个语言，那么我们是否有其他文法可以描述同样的语言呢？我们可以找到另外一个文法，这个文法没有歧义。

2.5.2 文法变换方法

在实际工作中，人们总是希望定义一种语言的文法尽可能地简单；另外，某些常用的语法分析技术也会对文法提出一定的要求或限制。为了适应上述要求，有时需要对文法进行必要的改写一通常称为文法变换，当然改写后的文法要与原文法等价。（文法变化具有技巧性）

这里重点介绍三类变换：

1. 删除无用的产生式（文法的化简）；
2. 删除 ϵ 产生式；（文法的右部是一个空串，我们要把他删除掉）
3. 常用的三种文法变换方法：（是我们一些常见的文法变换方法，这种方法比我们前面公式化要简单很多）
 - ①必选项法；
 - ②可选项法；
 - ③重复可选项法。
 这三种方法会贯穿我们后面语法分析的部分。

2.5.3 文法变换方法 1

I 文法的化简

文法化简是指消除如下无用产生式：

1. 删除 $A \rightarrow A$ 形式的产生式 (自定义)；
2. 删除不能从其推导出终结字符串的产生式 (不终结)；
3. 删除在推导中永不使用的产生式 (不可用)。

刚开始写文法的时候很难写出没有冗余的文法，当存在上述情况时如何处理？对于自定义产生式很容易找到，下面说一下后两种情况的处理方法。

※ 删除不终结产生式的算法：

1. 构造能推导出终结字符串的非终结符集 V_{VT} ：
 - (a) 若有 $A \rightarrow \alpha$ 且 $\alpha \in V_T^*$ ；则令 $A \in V_{VT}$ ；
 - (b) 若有 $B \rightarrow \beta$ 且 $\beta \in (V_T + V_{VT})^*$ ；则令 $B \in V_{VT}$ ；
 - (c) 重复步骤① ②，直到 V_{VT} 不再扩大为止。
2. 删除不在 V_{VT} 中的所有非终结符 (连同其产生式)。

想消除掉不终结的产生式，但是我们不容易求不终结的产生式，那我们可不可以求出哪些产生式是终结的。这个算法定义了 V_{VT} 。

①对于 $A \rightarrow \alpha$ 且 $\alpha \in V_T^*$ ，A 可以推出终结符，则把 A 放入 V_{VT} ，把产生式右部是终结字符串的变量放入 V_{VT} 中

② $B \rightarrow \beta$ 这个产生式的右部是由终结符或 V_{VT} 中元素构成的符号串，我们就把 B 看作可终结的，把 B 放入 V_{VT} 中

③这个过程反复不断的执行，直到 V_{VT} 不再扩大为止，如果这个符号不在 V_{VT} 中，我们就称 C 为不终结产生式的左部，我们把 C 和包含 C 的所有产生式删除。

※ 删除不可用产生式的算法：

1. 构造可用的非终结符集 V_{US}
 - (a) 首先令 $Z \in V_{US}$ ；(Z 为文法开始符号)

- (b) 若有 $Z \Rightarrow \dots A \dots$, 则令 $A \in V_{Us}$;
- (c) 重复步骤②, 直到 V_{VT} 不再扩大为止。

2. 删除不在 V_{VT} 中的所有非终结符 (连同其产生式)。

不可用产生式就是由起始符号不能推导出来的非终结符所对应的产生式, 我们称这样的产生式为不可用产生式。

找不可用产生式和不终结的产生式, 他们的思路是一模一样的, 一个是从终结符开始找, 不可用是从起始符号开始找, 只是方向有区别, 算法是一样的。

首先定义一个集合 V_{VT} , 文法的开始符号都放在 V_{VT} 中, 如果 Z 经过若干次推导, 将推导出的符号串中的非终结符放入 V_{VT} 中, 重复执行这样的过程直到 V_{VT} 不再扩展。最后, 删除不在 V_{VT} 中的所有非终结符连同其产生式。

运用上面三条规则来做一下例 2.16。

例 2.18 化简下述文法:

语法规则

$G(S) : S \rightarrow Be|Ec$
 $A \rightarrow Ae|e|A$
 $B \rightarrow Ce|Af$
 $C \rightarrow Cf; D \rightarrow f; G \rightarrow b$

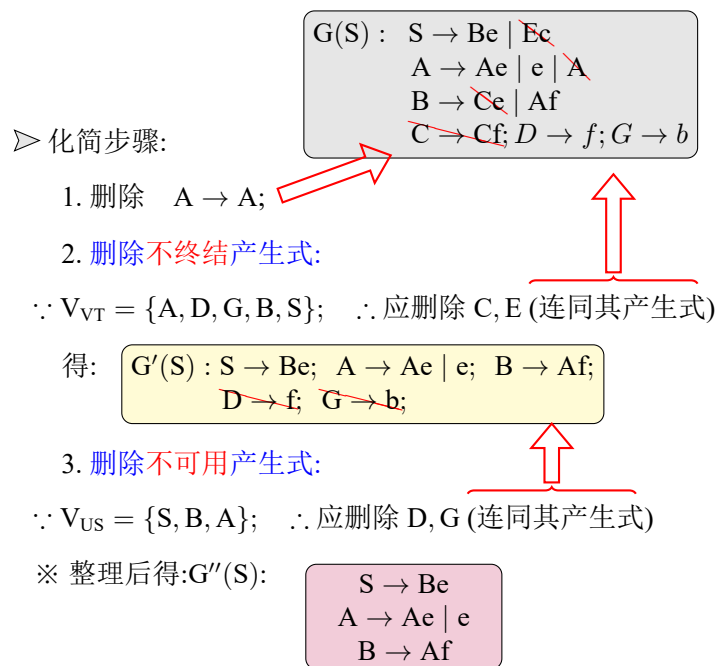


图 2.9: 文法化简示意图

2. 哪些非终结符对应的是不可终结的符号串? CE

3. 起始符号到达不了的非终结符 DG

II 删除 ϵ 产生式

* 假定文法 $G(Z); \epsilon \equiv L(G)$ 【算法】

1. 首先构造可以推出空串的非终结符集: V_ϵ

- ① 若有 $A \rightarrow \epsilon$; 则令 $A \in V_\epsilon$
- ② 若有 $B \rightarrow A_1 \dots A_n$ 且全部 $A_i \in V_\epsilon$; 则令 $B \in V_\epsilon$;
- ③ 重复步骤① ②, 直到 V_ϵ 不再扩大为止。

2. 删除 $G(Z)$ 中的 $A \rightarrow \epsilon$ 形成的产生式;

3. 依次改写 $G(Z)$ 中的产生式 $A \rightarrow X_1 X_2 \dots X_n$: 若有 $X_i \in V_\epsilon$, 则用 $(X_i | \epsilon)$ 替换之 (一个分裂为两个);

* 若有 j 个 $X_i \in V_\epsilon$, 则一个产生式将分裂为 2^j 个!

V_ϵ 中的非终结符, 经过若干次推导可以推导出空串

- ① 首先找直接能推出空串的产生式, 把他的左部加入 V_ϵ
- ② 假设产生式右部都是能推导出空串的符号, 即 $A_1 \dots A_n$ 且全部 $A_i \in V_\epsilon$, 那么产生式左部所对应的非终结符放入 V_ϵ 中
- ③ 重复上述过程, 直到 V_ϵ 不再扩大为止。

消除文法中含有空符号串的产生式, 找到文法中所有 $A \rightarrow \epsilon$, 再看 A 在哪些产生式中被使用了, 如果这个产生式右部含有的符号恰巧是 V_ϵ 中的, 就做如下的替换: 把 X_i 写成 X_i 或。

* 若有 j 个 $X_i \in V_\epsilon$, 则一个产生式将分裂为 2^j 个!

例 2.19 $G(S) : S \rightarrow aAbc|bS \quad A \rightarrow dABe|\epsilon; B \rightarrow A|b$

我们要求任何一个表达式的右部都不能含有空串

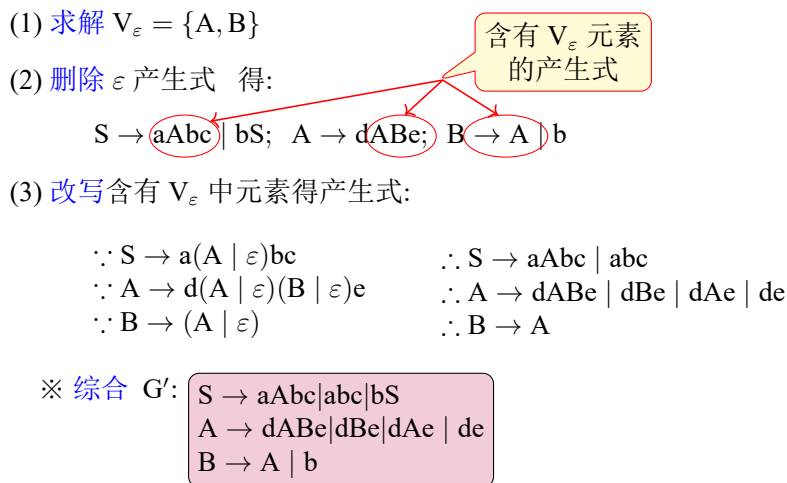


图 2.10: 文法改写示意图

1. 我们要检查这些规则的右部, 哪些元素是属于 V_ϵ 的
2. 先把第一条中的 A 替换为 $(A|\epsilon)$, 这种组合有两种再把第二条中的 A 和 B 替换为 $(A|\epsilon)$ 和 $(B|\epsilon)$, 这种组合有四种最后得到 $G'(S)$ 和 $G(S)$ 是等价的, 并且 $G'(S)$ 不包含产生 ϵ 的产生式

III 常用的三种文法变换方法:

文法的写作上有很多方便的形式来帮助我们的一些文法表达的更简便。

* 基本思想: 扩展文法, 引进新的描述符号:

() 圆括号; [] 方括号; { } 花括号

① 必选项法 (圆括号法) 令 $(\alpha | \beta) = \alpha$ 或者 β

例如: 有 $A \rightarrow a\alpha|a\beta$

可变换成: $A \rightarrow a(\alpha|\beta)$

也可: $A^A \rightarrow A'$; $A' \rightarrow \alpha | \beta$

【注】 此法又称提公因子法, 利用此法可以使文法:

具有相同左部的各产生式首符号不同!

这两个文法规则是等价的, 只不过我们把右部的 a 提出来了。

把圆括号的部分替换为 A' , 就可以得到另外两个产生式, 那为什么要这么做呢? 读者可以思考一下。

② 可选项法 (方括号法)

令 $[\alpha] = \alpha$ 或者 ε 该方法表示方括号中的内容可选也可不选

例如: $S \rightarrow \alpha|\alpha\beta$

可变换成: $S \rightarrow \alpha[\beta]$

也可 $S \rightarrow \alpha S'$; $S' \rightarrow \beta | \varepsilon$

可以把方括号中的部分用 S' 替换

$S \rightarrow \text{if}(B)S$

$S \rightarrow \text{if}(B)S \quad \text{else} \quad S$

B 为条件语句

可变换成: $S \rightarrow \text{if}(B)S[\text{else}S]$

或: $S \rightarrow \text{if}(B)SS'$; $S' \rightarrow \text{else}S | \varepsilon$

把后面 $\text{else} S$ 的部分用方括号圈起来, 表达的是可选也可以不选

③ 重复可选项法 (花括号法)

令 $\{\alpha\} = \varepsilon$ 或 α 或 $\alpha\alpha$ 或 $\alpha\alpha\alpha \dots$

花括号代表无数种情况, 表示由 α 构成的符号串包括空串

例如: $S \rightarrow A\beta|\alpha$ 这个文法所生成的语言是一个什么样子的语言呢?

这个文法只有 A 使用 α 进行推导的时候才能生成终结符串, 如果 A 推导出 $A\beta$ 那这个推导就无法终止, $A \rightarrow A\beta \rightarrow A\beta^2 \rightarrow A\beta^3 \rightarrow \dots A\beta^n = \alpha\beta^n, n \geq 0$ 可变换为: $A \rightarrow \alpha\{\beta\}$

也可 $A \rightarrow \alpha A'$; $A' \rightarrow \beta A' | \varepsilon$

* 验证:

\therefore 通过递推方法, 可得:

$A \Rightarrow A\beta \Rightarrow A\beta\beta \Rightarrow A\beta\beta\beta \Rightarrow \dots \Rightarrow \alpha\beta^*$

\therefore 有 $A \rightarrow \alpha\{\beta\}$; 或 $A \rightarrow \alpha A'$; $A' \rightarrow \beta A' | \varepsilon$

【注】 此方法常用来消除文法的直接左递归!

同样我们也可以用 A' 来替换花括号中的内容, 粉色和绿色的部分分别代表花括号的方式和文法改造的方式。

白色部分的文法具有左递归，绿色部分的文法没有直接左递归，但是有递归的部分，是直接右递归。左递归和右递归在我们文法分析方法中导致的方法是完全不一样的。在后面文法分析一章会发现直接左递归文法很难分析。举个例子：比如用自上而下的分析，如果文法有直接左递归性，则采用最左推导时会导致死循环，所以在很多场景中我们要消除直接左递归。

2.6 形式语言的分类

Chomsky 把形式语言分为四类，分别由四类文法定义；四类文法的区别在于产生式的形式不同。

1. 0 型语言由 0 型文法定义

- 产生式的形式为： $\alpha \rightarrow \beta$ ，又称无限制文法。

2. 1 型语言由 1 型文法定义

- 产生式的形式为： $xAy \rightarrow x\beta y$ ，又又称上下文有关文法。

3. 2 型语言由 2 型文法定义

- 产生式的形式为： $A \rightarrow \beta$ ，又称上下文无关文法。

4. 3 型语言由 3 型文法定义

- 产生式的形式为： $A \rightarrow aB, A \rightarrow a, A \rightarrow \varepsilon$ ，又称正规文法。

【注】 四类语言为包含关系，且有 $L_0 \supset L_1 \supset L_2 \supset L_3$ ，编译处理中主要应用后两种文法。

在这门课中，用到的事 2 型语言和 3 型语言，上下文无关文法对应语法分析的部分，正规文法对应词法分析的部分。

第3章 自动机基础

3.1 正规语言及其描述方法

若文法中每个产生式都限制为 $A \rightarrow aB$ 或 $A \rightarrow a$, 其中 A 和 B 均为单个非终结符 (此文法为右线性文法, 左线性文法形如 $A \rightarrow Ba$ 或 $A \rightarrow a$), 则称这类文法为正规文法。

定义 3.1 正规语言是指由正规文法定义的语言。

正规语言有三种等价的表示方法:

1. 正规文法
2. 正规式
3. 有限自动机

正规文法就是正规语言的一种描述形式, 正规式可以简单理解为正则表达式, 是用计算机语言来描述正规语言的一种方式, 这三种都是等价的。

例 3.1 正规文法 $G(Z) : A \rightarrow A|\epsilon$

$$\because A \Rightarrow \epsilon; A \Rightarrow aA \Rightarrow aaA \Rightarrow aaaA \Rightarrow \dots \Rightarrow a^n, n \geq 0$$

正规语言 $\therefore L(G) = \{a^n \mid n \geq 0\}$

例 3.2 $L1 = \{a^m b^n \mid m \geq 0, n \geq 1\}$, 正规语言?

\therefore 可由正规文法定义:

$G1(S) : S \rightarrow aS | bA; A \rightarrow bA | \epsilon$

$\therefore L1$ 是正规语言

如果一种语言是正规语言, 那就可以找到一个正规文法去描述该语言。

例 3.3 $L2 = \{(ab)^n \mid n \geq 1\}$, 正规语言? \therefore 可由正规文法定义:

$G2(S) : S \rightarrow aA; A \rightarrow bB; B \rightarrow aA | \epsilon$

$\therefore L2$ 是正规语言

例 3.4 $L3 = \{a^n b^n \mid n \geq 0\}$, 正规语言?

∴ 不能由正规文法定义 (正规文法无法描述 a 和 b 数量上不相等!):
 ∴ 不是正规语言

用什么样的语言可以描述

例 3.5 这个文法? 这个模型对应的语言需要用下推机来描述。

如果上下文无关文法不含有 ϵ , 则该文法可以被转化为正规文法。

3.1.1 正规语言的正规式表示法

* 正规式: 是指由字母表中的符号, 通过以下三种运算 (也可以使用圆括号) 连接起来构成的表达式 e : 连接 (\cdot) 或 ($|$) 闭包 ($+$, $*$)

正规式	正规式的值
$ab.cde$	$abcde$
$ab cde$	ab, cde
a^+	$a, aa, aaa, \dots, a^n, \dots$
a^*	$\epsilon, a, aa, aaa, \dots, a^n, \dots$

图 3.1: 正规式举例

* 设 $val(e), L(e)$ 分别表示正规式 e 的值和语言, 则: $L(e) = \{x | x = val(e)\}$, 即正规式表示的语言是该正规式所有值的集合 (正规集)。

* 正规式表示正规语言示例:

例 3.6 (1) $e_1 = a^*b^+$

$$L(e_1) = \{a^m b^n \mid m \geq 0, n \geq 1\};$$

$$= \{b, ab, bb, aaab, aab, abb, aabb, \dots\}$$

(2) $e_2 = (ab)^+$

$$L(e_2) = \{(ab)^n \mid n \geq 1\},$$

$$= \{ab, abab, ababab, abababab, \dots\}$$

(3) $e_3 = ab^*c \mid b^*$

$$L(e_3) = \{ab^n c, \quad b^n \mid n \geq 0\}$$

$$= \{ac, abc, abbc, abbbc, \dots; \epsilon, b, bb, bbb, \dots\}$$

3.1.2 正规语言的有限自动机表示法

有限自动机 (FA) 是正规语言的描述方法之一, 先来看三个示例。

$L_1 = \{a^m b^n \mid m \geq 0, n \geq 1\}$, L_1 对应的有限状态自动机为

FA2:

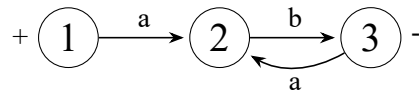


图 3.2: L1 对应的有限状态自动机

$L2 = \{(ab)^n \mid n \geq 1\}$, L2 对应的有限状态自动机为

FA2:

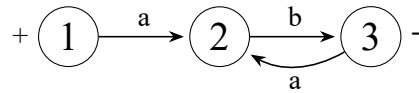


图 3.3: L2 对应的有限状态自动机

$L3 = \{ab^nc, b^n \mid n \geq 0\}$, L3 对应的有限状态自动机为

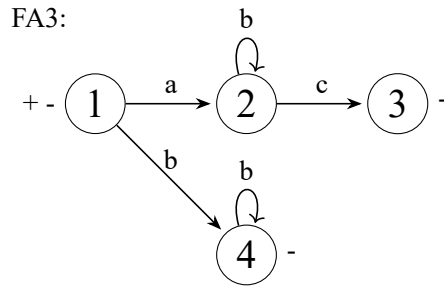


图 3.4: L3 对应的有限状态自动机

※ 初看起来，有限自动机是带标记的有向图！①②③④ 是图上的节点，称之为状态，带箭头的线可以理解为图上的边，这个边是有方向的，称之为有向边，+ 和- 分别对应起始状态和结束状态

我们让大家有一个直观的印象，每一个正规语言都可以用一个有限状态自动机来描述。

※ 有限自动机表示法说明：

$L3 = \{ab^nc, b^n \mid n \geq 0\}$ 有限状态自动机由什么组成呢？

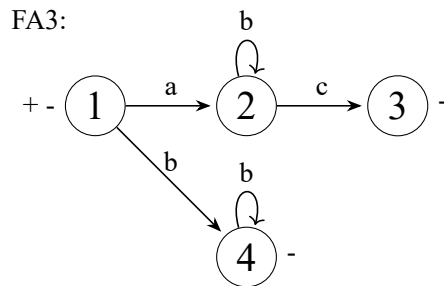


图 3.5: FA3 对应自动机

我们分析一下 FA3

【图符说明】：

{1,2,3,4}-状态集;

其中:+(开始状态);-(结束状态)

{a,b,c}-字母表;

$\delta(1, a) = 2$ -变换 (或 $\textcircled{1} \xrightarrow{a} \textcircled{2}$);...

(表示 1 状态遇符号 a 变到 2 状态...);

图 3.6: 符号说明

①②③④ 是状态，状态表示的是事物在某一时刻的表示，+ 和-分别表示开始状态和结束状态，自动机的运行一定要从开始状态开始，最终到达结束状态，① 代表起始状态和结束状态可以是同一个状态。

字母表可以理解为文法中的字母表，自动机需要读入的最基本的元素， δ 表示状态转移，对应有向图中带箭头的边， $\delta(1, a) = 2$ 表达状态 1 读入 a 后要跳转到状态 2。

【运行机制】

一个 FA，若存在一条从某开始状态 i 到某结束状态 j 的通路，且这条通路上所有边的标记连成的符号串为 α ，则 α 就是一个句子；所有这样的 α 的集合，就是该 FA 表示的语言。

如果一个自动机的起始状态和终止状态写在了一起，则表示这个自动机是能接受空串的。

从自动机的应用上来说，基本的功能是判断一个符号串能否被一个文法所接收，换句话说来说，若能找到一个路径恰巧对应这个符号串，则表示该符号串能被接收，如果找不到这样的路径，则表示这个符号串不能被自动机所接收。

* 正规语言的三种表示法综合示例：

例 3.7 $L = \{ab^n c, b^n \mid n \geq 0\}, \sigma = \{a, b, c\}$

1. 正规文法描述:

$G(S) : S \rightarrow aA|bB|\epsilon, A \rightarrow bA|c, B \rightarrow bB|\epsilon$

2. 正规式描述: $e = ab^*c \mid b^*$

3. 有限自动机描述:

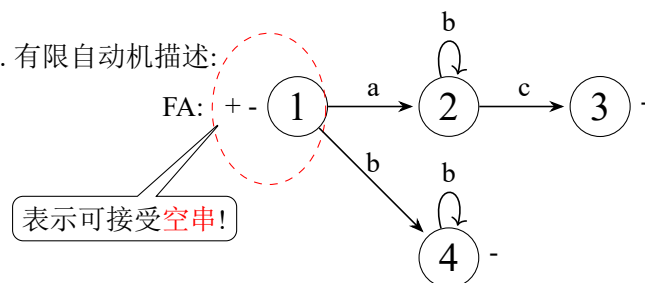


图 3.7: 综合示例

【注】凡是能由上述三种方法表示的语言，一定是正规语言；反之，凡是不能由上述三种方法表示的语言，一定不是正规语言。

3.2 有限自动机的定义与分类

3.2.1 有限自动机的定义

定义 3.2 有限自动机是一种数学模型，用于描述正规语言，可定义为五元组: $FA = (Q, \Sigma, S, F, \delta)$
上下文无关文法是几元组？四元组：非终结符集，终结符集，起始符号，产生式集合。有限自动机是五元组，

Q (有限状态集);

Σ (字母表);

S (开始状态集, $S \subseteq Q$);

F (结束状态集, $F \subseteq Q$);

δ : 变换 (二元函数):

状态 ①②③④⑤ 用状态 Q 来表示, S 是 Q 的子集, 变换 $\delta(i, a) = j$ 或 $\textcircled{i} \xrightarrow{a} \textcircled{j}, i, j \in Q, a \in \Sigma + \{\epsilon\}$
表示状态 i 遇符号 a , 变换为状态 j 。空串在应用的角度来说不好, 给计算机处理带来很大的麻烦, 但是有限自动机是可以读入空串的。

3.2.2 有限自动机怎样描述语言

令 $L(FA)$ 为自动机 FA 所描述的正规语言; 则 $L(FA) = \{x \mid (i \xrightarrow{x} j), x \in \Sigma^*, i \in S, j \in F\}$
一个有限状态自动机 FA 所对应的语言写作 $L(FA)$ 。从起始状态 i 到结束状态 j 的路径上所有符号所拼接起来构成的符号串便是该自动机能接受的一个符号串, 所有这样符号串构成的集合便是 FA 所对应的语言, 记作 $L(FA)$ 。 $i \xrightarrow{x} j$ 表示从起始状态 i 到终止状态 j , 其中每一次跳转分别读入的符号为 a_1, a_2 一直到 a_n , 写成这样的形式设 $x = a_1 a_2 \dots a_n$, $a_i \in \Sigma + \{\epsilon\}$ 则有 $i \xrightarrow{a_1} i_1 \xrightarrow{a_2} i_2 \dots \xrightarrow{a_n} j$, 每次读入的字符构成了一个符号串 $a_1 a_2 \dots a_n$, 记为 x , 则符号串 x 是 $L(FA)$ 中的一个句子, 所有句子构成的集合就是该有限自动机对应的语言。

FA 存在一条从某初始状态 i 到某结束状态 j 的连续变换, 且每次变换 (\Rightarrow) 的边标记连成的符号串为 x , 则称 x 被 FA 接受, 否则拒绝。

所以有限状态自动机最重要的一个功能是判断一个词串是否能被接收, 词法分析便是利用有限状态自动机来识别单词串。

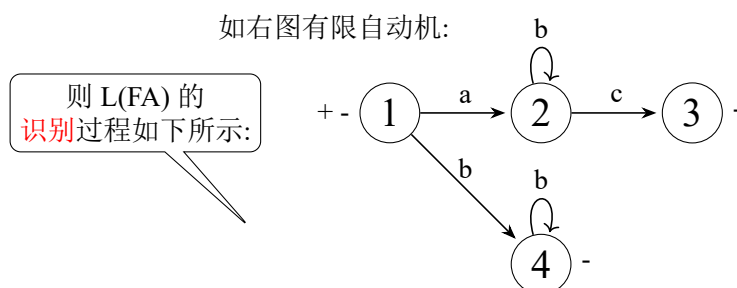


图 3.8: 有限自动机

*L(FA) 的生成 (或识别) 过程示例:

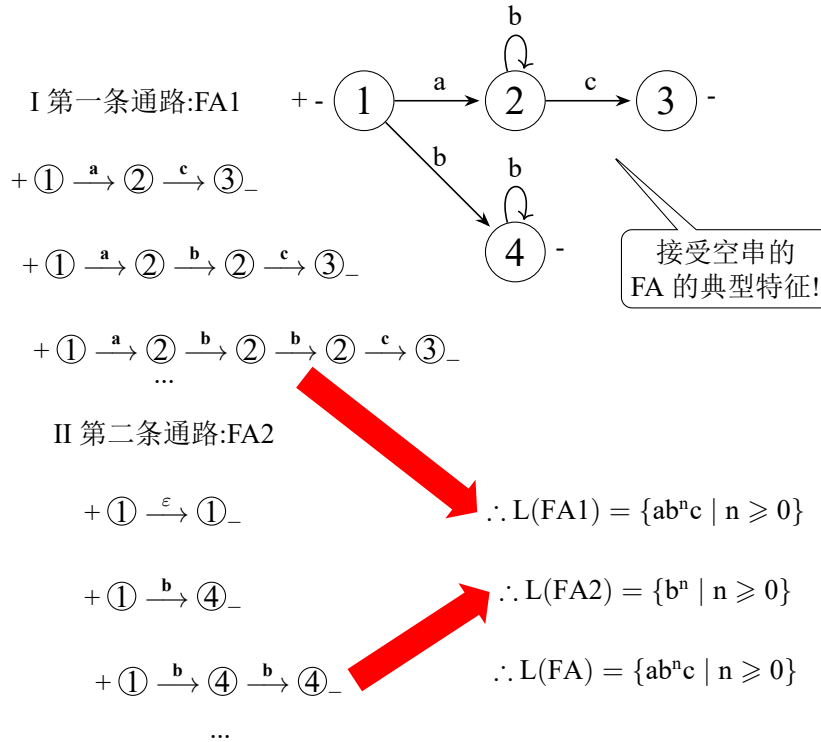


图 3.9: L(FA) 生成 (或识别) 过程示例

起始状态 ① 读入 a 跳转到 ② 号状态, ② 号状态读入一个 c 到 ③ 号状态, ③ 号状态是一个结束态, 因此符号串 ac 是该语言的一个句子。

起始状态 ① 读入 a 跳转到 ② 号状态, ② 号状态读入一个 b 仍然跳转到 ② 状态, ② 状态读入 c 跳转到 ③ 号状态, ③ 状态是个结束态, 因此符号串 abc 是该语言的一个句子。

起始状态 ① 读入 a 跳转到 ② 号状态, ② 号状态读入一个 b 仍然跳转到 ② 状态, ② 号状态又读入一个 b 仍然跳转到 ② 状态, ② 状态读入 c 跳转到 ③ 号状态, ③ 状态是个结束态, 因此符号串 abbc 是该语言的一个句子

综上, 上半部分自动机 FA1 对应的语言可表示为 $\therefore L(FA1) = \{ab^nc \mid n \geq 0\}$

II. 第二条通路: FA2

起始状态 ① 直接读入空串就到达了结束态。

起始状态 ① 读入 b 到达结束态 ④。

起始状态 ① 读入 b 到达状态 ④, 再读入 b 又到了 ④ 状态, 这个时候是结束态。

以此类推, 下半部分的自动机 FA2

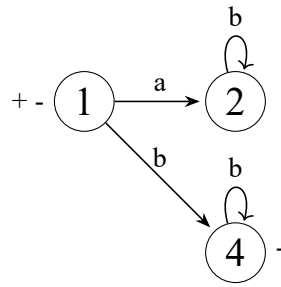


图 3.10: FA 自动机

表达的是 $\therefore L(FA2) = \{b^n \mid n \geq 0\}$

因此整个自动机所对应的语言就是 FA1 和 FA2 对应的语言合并在一起, 即 $\therefore L(FA) = \{ab^n c \mid n \geq 0\}$

3.2.3 有限自动机的两种表现形式

例 3.8 有限自动机: $FA = (Q, \Sigma, S, F, \delta)$

其中: $Q=1,2,3,4, \Sigma=a,b,c, S=1, F=3,4$

$\delta: \delta(1, a) = 2; \delta(1, b) = 4; \delta(2, b) = 2;$

$\delta(2, c) = 3; \delta(2, \epsilon) = 3; \epsilon(4, b) = 4;$

我们可以把有限自动机写成状态图或变换表的模式
FA 的两种表现形式:

(1) 状态图:

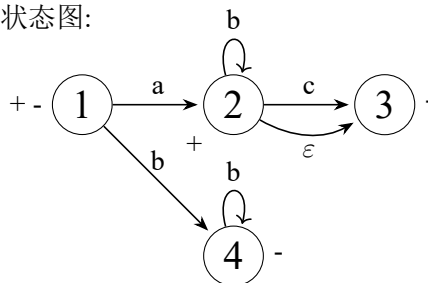


图 3.11: FA 状态图

这个图并不容易让计算机去识别

(2) 变换表:

	a	b	c	ϵ	
1	2	4			+
2		2	3	3	+
3					-
4		4			-

开始状态
结束状态

图 3.12: FA 变换表

这是计算机存储中最常用的方式, 可以把状态转移看成一个表的形式, 表中每一行对应一

个状态，所以这个表中写了四个状态，每一个列分别对应一个字母，行和列的交叉点表示对应的状态读入相应的字母时应跳转到的状态。

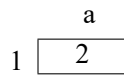


图 3.13: 状态 1

表达 ① 状态读入 a 跳转到 ② 状态，即 $\delta(1, a) = 2$

起始状态和结束状态分别用 “+” 和 “-” 表示。* 有限自动机的构造示例 1

例 3.9 $A = \{ab^n c, (ab)^n \mid n \geq 0\}$ FA 的构造:

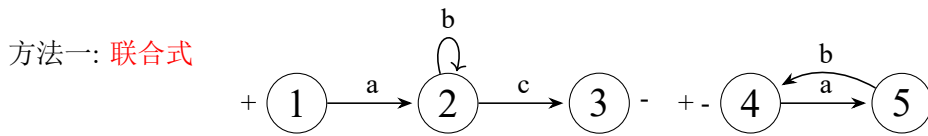


图 3.14: 方法一

对于符号串 $ab^n c$ 可构造出这样一个自动机，开始状态 ① 读入 a 进入 ② 状态，使用循环方式表示 ② 状态可读入无限个 b, ② 状态若读入 c 进入 ③ 状态。同样道理第二个串 $(ab)^n$ 可以写成这样的自动机，开始状态 4 读入 a 进入 5 状态，5 状态读入 b 进入结束态 4。注意 4 状态既是起始态又是结束态，表示该自动机可接收一个空串。

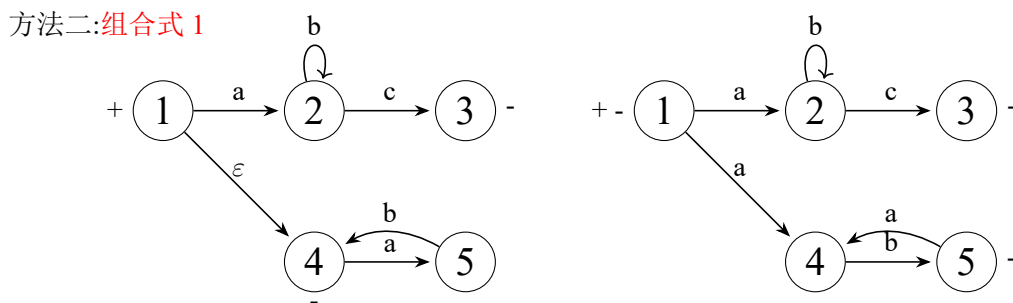


图 3.15: 方法二

④ 状态是一个起始状态，但是在方法二中 ① 状态通过跳转到了 ④ 状态，所以 ④ 状态可以不标注起始状态。

消除 ① 状态到 ④ 状态的 ϵ 跳转得到

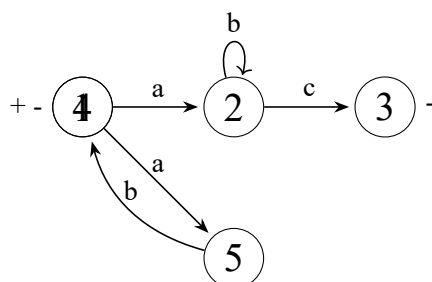


图 3.16: 跳转结果

该自动机与联合式或组合式的自动机是否等价？答案：不等价
 正确的自动机如图所示

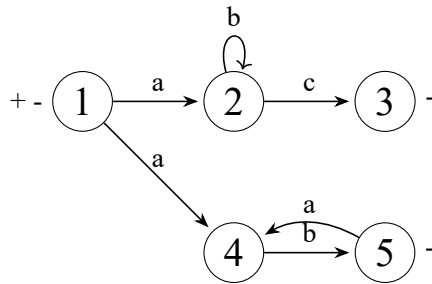


图 3.17: 正确结果

接下来比较各种自动机表示方法的优劣：方法一：开始状态不唯一, 不好用! 方法二：带有 ϵ 边, 还是不好用! 另外, 变换函数不单值, 如 $\delta(1, a) = (2|4)$ 也不好!

方法三: 确定式

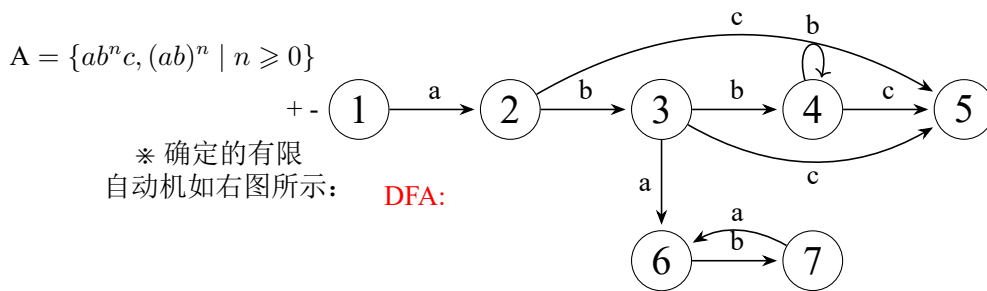


图 3.18: 方法三

状态唯一, 不带有 ϵ 边, 变换函数单值的有限自动机。

最后我们再举一个现在中的例子：奇偶校验

例 3.10 奇偶校验

奇校验

Input: 由 0, 1 组成的字符串

Output: 奇数个 1, 则输出 ok, 否则拒绝

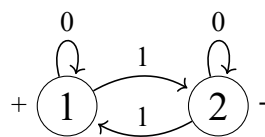


图 3.19: 奇校验

类似的, 偶校验的自动机如下:

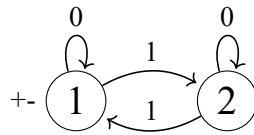


图 3.20: 偶校验

3.2.4 有限自动机的分类

1. 确定的有限自动机 (DFA)

特征：①开始状态唯一；②变换函数单值；③不带 ε 边。

2. 非确定的自动机 (NFA) – 不能全部具备上述特征者！

(a) 带有 ε 边的非确定的有限自动机，记为 ε NFA；

(b) 不带有 ε 边的非确定的有限自动机，记为 $\bar{\varepsilon}$ NFA；

※ 有限自动机的分类示例

例 3.11 试分别指出下述有限自动机的分类情况

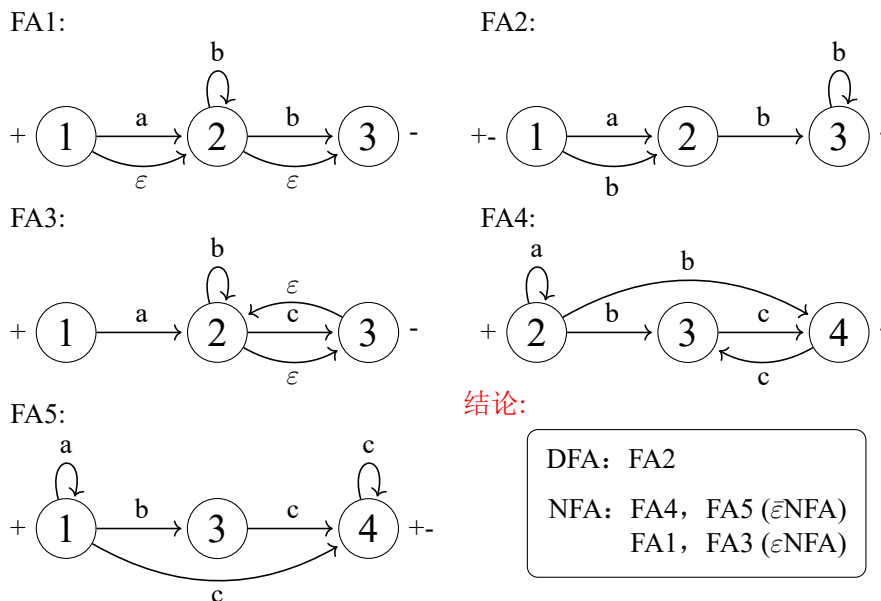


图 3.21: 有限自动机的分类情况

上述五个有限状态自动机中哪些是确定型的？FA2 为 DFA。

FA4 变换函数不单值，FA5 起始状态不唯一，FA1 和 FA3 带有 ε 边。

3.3 有限自动机的等价变换

能不能找到确定型的有限状态自动机，从理论上已经证明任何有限状态自动机都可以转化为确定型的有限状态自动机。有限自动机的等价变换，是指把 FA1 变换为 FA2，且有 $L(\text{FA1}) = L(\text{FA2})$ 。主要包含以下两个内容：

1. 有限自动机的确定化 (NFA => DFA) ;

找到一个确定型有限状态自动机, 同时其接收的语言和原始的自动机接收的语言是一摸一样的, 这个过程叫确定化。非确定机 (NFA) 较易构造, 但不易于使用。相反, 确定机 (DFA) 较难构造, 但使用上更加方便。

* 任何一非确定机 NFA, 皆可通过有效算法把其转换为等价的确定自动机 DFA。

2. 有限自动机的化简 (最小化);

找到一个自动机和原自动机是等价的, 但其状态数是最少的。

* 对任何一确定机 DFA1, 皆可通过有效算法将其转换为等价的确定机 DFA2, 且 DFA2 的状态最少。

3.3.1 有限自动机的确定化

第一步先把一个含 ϵ 边的非确定有限状态自动机转化为不含 ϵ 边的非确定有限状态自动机, 第二步把不含 ϵ 边的非确定有限状态自动机确定化。

1. 消除 ϵ 边算法 (ϵ NFA \Rightarrow $\bar{\epsilon}$ NFA 或 DFA)

1. 若存在 ϵ 闭路, 则把 ϵ 闭路上各节点合而为一:

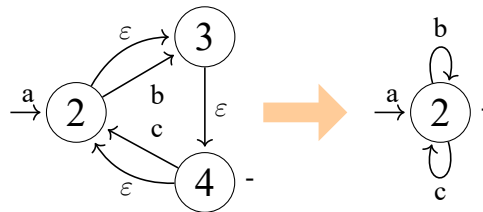


图 3.22: ϵ 闭路上各节点合而为一

2. 标记隐含的开始态和结束态:

开始态的 ϵ 通路上的节点: +

结束态逆向 ϵ 通路上的节点: -

从起始状态开始找到所能到达的 ϵ 通路上的所有节点, 将其标注为起始状态。比如起始状态①, 通过 ϵ 跳转到达的任何节点都标注为起始状态。

从结束态逆向标注, 如果一个节点通过 ϵ 跳转能到达结束态, 则认为这个节点是结束态逆向通路上的节点, 将其标记为结束态。

例如:

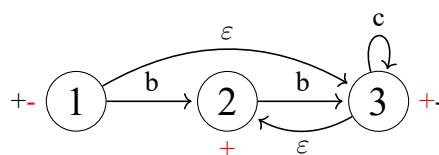


图 3.23: 标记隐含的开始态和结束态

起始状态①通过 ϵ 跳转到③状态，③状态通过 ϵ 跳转到②状态，因此③状态和②状态都是起始状态 ϵ 通路上的节点，将状态②和状态③标记为起始状态。同样道理，③状态是结束态，①状态通过 ϵ 跳转可以到达③状态，因此将①状态标记为结束状态。

3. 对剩余的 ϵ 边，逆向逐一进行：

①删除一个 ϵ 边；同时

②引进新边：凡由原 ϵ 边终点出发的边，也要由其始点发出。

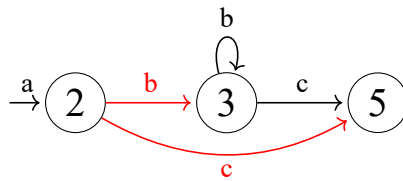


图 3.24: 删除 ϵ 边，同时引进新边

4. 重复步骤 (3)，直到再无 ϵ 边为止。例如：

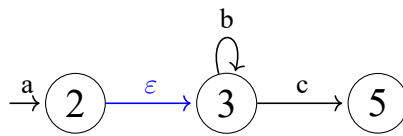


图 3.25: 重复步骤 (3)，直到再无 ϵ 边

删除状态②到状态③的 ϵ 边，③状态能到达③状态和⑤状态，所以②状态读入 b 也得到达③状态，②状态读入 c 也能到达⑤状态。

例 3.12 消除 ϵ 边算法示例：

已知 ϵ NFA 如下，求 $L(\epsilon$ NFA) = ?

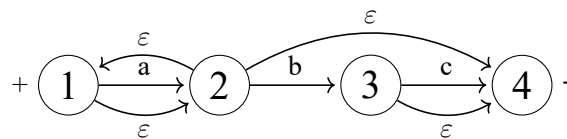


图 3.26: 消除 ϵ 边算法示例

1. ϵ 闭路上的节点等价，(① \equiv ②)，可合二为一；得到

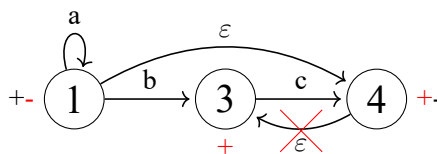


图 3.27: 节点等价

2. 标记隐含的开始态和结束态：+④，+③，-①

从开始状态找 ϵ 通路，把通路上的节点都标为开始状态，①状态读入 ϵ 到达④状态，④状态读入 ϵ 到达③状态。从结束态开始，找 ϵ 逆向通路上的节点，将其标记为结束态，①状态通过 ϵ 跳转到达结束态④状态，所以将①状态标记为结束态。

3. 逆序逐一删除 ϵ 边，同时引进新边：

先删除④状态到③状态的 ϵ 边

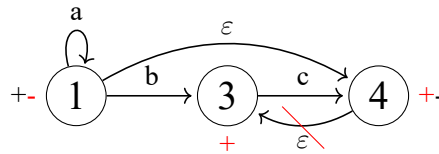


图 3.28: 删除④到③的 ϵ 边

然后删除①到④的 ϵ 边

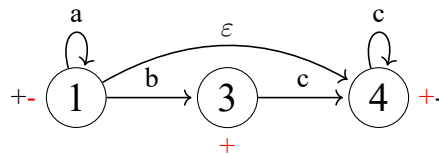


图 3.29: ①到④的 ϵ 边

③能到达④，那④就能到达④（即③能读入 c 到④，那④就能读入 c 到④）

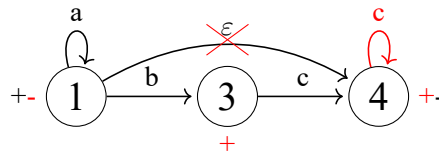


图 3.30: ④读入 c 到④

删除状态①到状态④的 ϵ 边， \square 状态能到达④状态，所以①状态读入 c 也能到达④状态

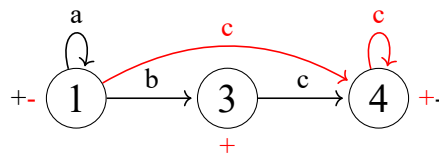


图 3.31: $\bar{\epsilon}$ NFA

$$\therefore L(\bar{\epsilon}\text{NFA}) = \{a^m, a^m b c^n, a^m c^n \mid m \geq 0, n \geq 1\}$$

2. $\bar{\epsilon}$ NFA 的确定化算法 ($\bar{\epsilon}\text{NFA} \Rightarrow \text{DFA}$)

1. 构造 DFA 的变换表 (框架):

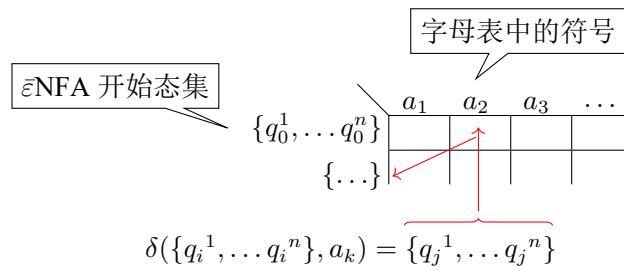


图 3.32: DFA 变换表

2. 按 ϵ NFA 的变换函数实施变换： $\delta(\{q_i^1, \dots, q_i^n\}, a_k) = \{q_j^1, \dots, q_j^n\}$
3. 若 $\{q_i^1, \dots, q_i^n\}$ 未作位状态进行标记，则作新行标记；
4. 重复步骤 (2)(3)，直到不再出现新状态集为止；
5. 标记 DFA 的开始态和结束态：
 - 第一行 $\{q_i^1, \dots, q_i^n\}$ ，（右侧）标记+；
 - 凡是状态行中含有 ϵ NFA 的结束状态者，（右侧）标记-。

【注】必要时，新产生的 DFA 可用状态图表示。

自动机所有的起始状态构成一个集合 $\{q_i^1, \dots, q_i^n\}$ ，该集合作为确定自动机 DFA 的起始状态。DFA 变换表的行由原始自动机中状态或状态集合表示 $\{q_i^1, \dots, q_i^n\}$ ，列是字母表上的符号。从 DFA 起始状态集合开始，把集合中任意一个状态读入相应的字母所到达的状态合成一个集合（到达的状态集合），作为 DFA 变换表中相应的表项，注意该表项可能是一个单一状态，也可能是一个状态集合。只要表项中状态或状态集合没有出现在 DFA 变换表的行标记中，则将其作为一个新的行标记，直到没有新的行标记出现为止。经过上述处理后，便可得到原始自动机对应的确定化有限自动机，这个确定化有限自动机的状态可能是是原始自动机中的状态或状态的集合。最后，标注开始态和结束态，新自动机的开始态只有一个，即 $\{q_i^1, \dots, q_i^n\}$ ；如果新自动机中行标记包含原始自动机的结束态，就将其都标记为结束态。

※ ϵ NFA 确定化示例

例 3.13

联合自动机 NFA:

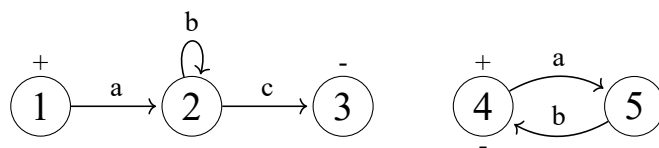


图 3.33: 联合自动机 NFA

不是确定型的自动机

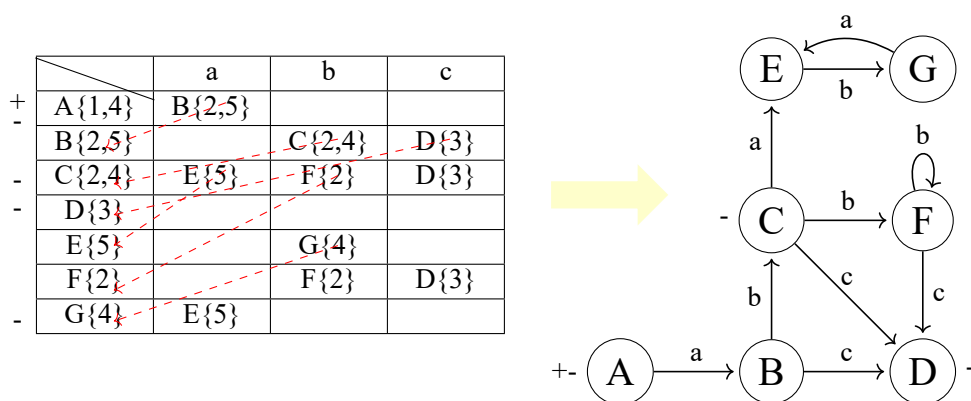


图 3.34: 确定化过程

【注】A,B,C,⋯状态集的代码

表列是字母表，行对应的是状态，原始自动机的起始状态①④作为确定自动机的起始状态，读入 a 到达②⑤

B{2,5} 读入 b 到达 C{2,4} 读入 c 到达 D{3};

C{2,4} 读入 a 到达 E{5} 读入 b 到达 F{2} 读入 c 到达 D{3};

E{5} 读入 b 到达 G{4};

F{2} 读入 b 到 F{2} 读入 c 到 D{3};

G{4} 读入 a 到 E{5};

只要新自动机中行标记包含原始自动机的结束态，就将其都标记为结束态。

根据确定自动机的变换表，便可构造出该确定自动机对应的状态图，不难看出，这两个自动机是等价的。

3.3.2 有限自动机的最小化

有限自动机的最小化，又称有限自动机的化简，是指对给定的确定机 DFA1，构造另一个确定机 DFA2，使得 $L(DFA1) = L(DFA2)$ ，且 DFA2 的状态最少。

有限自动机最小化算法，是指构造满足下述条件的确定有限自动机 (称为最小机):

1. 删除无用状态;
2. 合并等价状态。

第一步是删除无用状态

定义 3.3 无用状态是指由开始态达不到的状态 (不可达) 或者由其出发不能到达结束态的状态 (不终结)。

无用状态是指两种状态，一种是不可达的状态，一种是不终结的状态。这两种状态是不会参与到自动机的运行里的，参与了也没有用，所以这两种状态是不需要的，可以删除掉。第二步是合并等价状态。

如果两个状态是等价的，这两个状态是可以在最小化的操作中被合并的。那什么是等价状态呢？

定义 3.4 等价状态是指这样的两个状态，若分别将其看作开始态，二者接收的符号串集合相同。

下面我们来看一个例子

例 3.14 无用状态

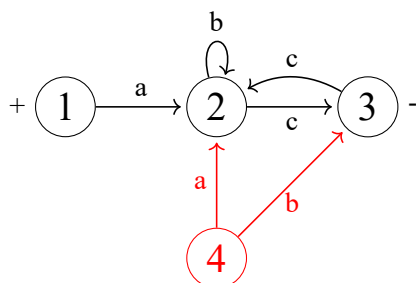


图 3.35: 不可达状态

状态④只有出边没有入边，所以该状态是一个不可达状态。

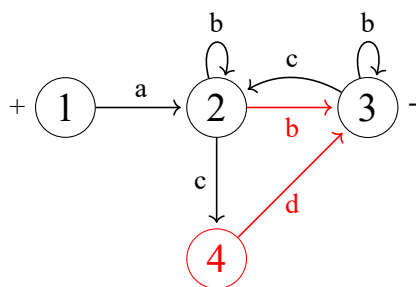


图 3.36: 不终结状态

状态③只有入边，没有出边，所以该状态是一个不终结状态。

不管是不可达也好，还是不终结也好，这两个状态都不是自动机需要的状态。

我们再来看下一个例子：

例 3.15 合并等价节点

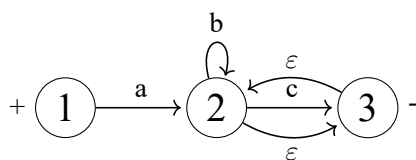


图 3.37: 合并等价节点

状态②和状态③这两个状态可以通过读入空串来互相跳转，②通过读入空串跳转到③，③通过读入空串跳转到②。之前说过，这是一个带空边的闭环，所以在这个闭环上的所有节点都是等价的，因此可以把它们合二为一。

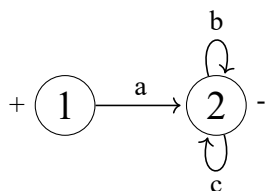


图 3.38: 保留非空跳转

可以看到，合并之后的②状态可以通过读 b 跳转到②，可以通过读 c 跳转到②，这是因为②③合并之后，②和③上的非空跳转仍然需要保留。

接下来看下一个例子，如何去判断等价性。上面例子中的带空边闭环是比较好判断的。

例 3.16 如何判断等价性：

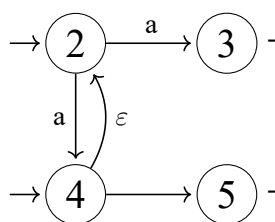


图 3.39: 判断等价性

首先判断④和⑤是否是等价的，我们可以从定义出发：

等价状态是指这样的两个状态，若分别将其看作开始态，二者接收的符号串集合相同。如果同样把④和⑤看做开始态，可以看到④状态可以接收空边到②，②再接收 a 到④，④还可以接收 b 到结束态⑤，即④作为开始态可以接收符号串 ab ，但是⑤作为开始态并不可以接收符号串 ab 。所以④和⑤是不等价的。

第二个问题：③和⑤是否是等价的？

状态③和⑤都是终止态，这两个状态都不能接收任何符号串，所以状态③和状态⑤是等价的。

第三个问题，②和③是否是等价？

显然②状态能够读入 a 到③状态，而③状态不能接收任何符号串，所以这两个状态是不等价的。

最后一个问题，②和④是否是等价？

④状态能够接收 b 到结束态⑤，而②状态不能接收 b ，所以这两个状态也不是等价的。

【算法】

接下来我们看一下如何去执行这两步：删除无用状态以及合并等价状态。首先看删除无用状态的算法。删除无用状态我们在这给了两个算法，第一个是删除不可达状态。

※ **【删除不可达状态】** 构造可达状态集 Q_{AR}

1. 设 q_0 为开始态, 则令 $q_0 \in Q_{AR}$;
2. 若 $q_i \in Q_{AR}$ 且有 $\delta(q_i, a) = q_j$ 则令 $q_j \in Q_{AR}$;
3. 重复执行 (2), 直到 Q_{AR} 不再增大为止。
4. 从状态集 Q 中, 删除不在 Q_{AR} 中的所有状态。

首先要理解什么是不可达状态? 就是无法到达的状态。那么为什么不可达呢? 这个不太好描述, 我们可以反过来想想, 什么是可达的状态。可达状态的定义非常简单, 从起始状态开始, 只要能找到一条路径到达一个状态, 那么这个状态就是可达状态。从这个角度说, 是否可以找出一个可达状态的集合呢? 这个算法就告诉我们怎么去构造可达状态集合。

首先从起始状态开始, 不断的进行搜索, 将所有可达的状态都加入到集合当中, 最后当这个集合不再增加的时候, 就得到了可达状态集合, 从状态集中把这些状态删除, 就得到了不可达状态的集合。

同样的道理, 不终结的状态怎么判断呢? 类似的, 不好判断一个状态是否是不终结状态, 我们可以去想想什么节点能够终结? 那就是什么节点能够到达终止状态, 如果可以找出所有能够到达终止状态的节点, 那么其余状态便是不终结状态。

※ 【删除不终结状态】构造可终结状态集 Q_{FN}

1. 设 q_i 为结束态, 则令 $q_i \in Q_{FN}$;
2. 若 $q_j \in Q_{FN}$ 且有 $\delta(q_i, a) = q_j$ 则令 $q_i \in Q_{FN}$;
3. 重复执行 (2), 直到 Q_{FN} 不再增大为止。
4. 从状态集 Q 中, 删除不在 Q_{FN} 中的所有状态。

接下来看一下如何判断等价状态。首先, 两个状态 i, j 等价, 当且仅当满足下面两个条件:

1. 必须同是结束态, 或同不是结束态;
2. 对所有字母表上符号, 状态 i, j 必变换到等价状态。

第一个条件, 两个状态必须同是结束态, 或者同不是结束态, 如果这两个状态里有一个是结束态, 有一个不是结束态, 显然结束态的那个节点可以读空直接停止了, 而非结束态的不一定。

第二个条件, 对于字母表上的所有符号, 两个状态分别读入同一个字符跳转到的两个状态都应该是等价状态。例如, i 状态读入 a 跳转到 I' , j 状态读入 a 跳转到 J' , 如果 I' 和 J' 等价, 那么 i, j 也是等价的。

例 3.17 判断等价状态:

把下述自动机最小化

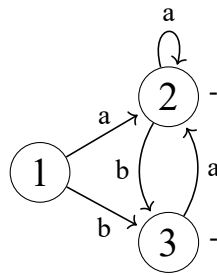


图 3.40: 判断等价状态

1. 初分成两个不等价子集: $Q_1 = \{1, 2\}$, $Q_2 = \{3\}$

将状态分为两个不等价子集, 根据等价条件的第一个条件, 因为①, ②状态都是结束态, 而③状态不是结束态, 可以划分成两个不等价子集 1, 2 和 3。这一步只能说①, ②有可能是等价, 但是①跟 3 或者②跟③肯定不是等价的。

接下来第二步, 根据等价条件的第二个条件, 如果两个状态读入相同的字符, 都能跳转到等价状态, 那么这两个状态是等价的。状态①, ②同时读入 a, b 都能跳转到同一个状态, 因此①, ②等价。

2. 还能分成不等价子集吗?

$$\because \delta(\{1, 2\}, a) = 2$$

$$\text{又 } \delta(\{1, 2\}, b) = 3$$

$$\therefore 1 \equiv 2$$

所以可以将①, ②状态合并在一起, 得到了下面的自动机

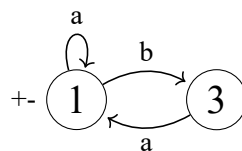


图 3.41: 将①, ②状态合并

我们可以将上述步骤总结成一个算法:

【合并等价状态算法】 – 划分不等价状态集

1. 初始, 把状态集 Q 化分成两个不等价子集:

Q_1 (结束状态集), Q_2 (非结束状态集);

2. 把每个 Q_i 再划分成不同的子集, 条件是:

同一 Q_i 中两个状态 i, j , 若对字母表中的某个符号, 变换到已划分的不同的状态集中, 则 i, j 应分离, 如: $\delta(i, a) \in Q_m, \delta(j, a) \in Q_n$ 且 $m \neq n$

3. 重复步骤 (2), 直到再不能划分为止;

4. 合并最终划分的每个子集中的状态 (合而为一)。

例 3.18 * 有限自动机化简示例:

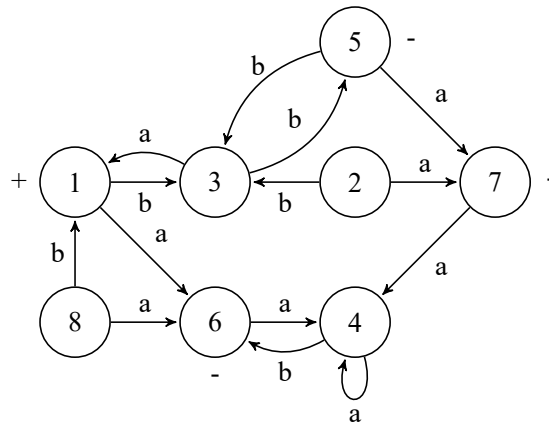


图 3.42: DFA

第一步：删除无用状态

动态构造 DFA 变换表，即从开始状态 1 出发，把变换后的状态填入表项，并同时作为新行标记；如此下去，直到不再出现新状态为止。未出现的状态，就是无用的状态。

		a	b
+	1	6	3
	3	1	5
-	6	4	
-	5	7	3
	4	4	6
-	7	4	

图 3.43: 构造 DFA 变换表

例??对应的 DFA 变换表如上图所示，该表中状态便是所有的可达状态，状态 2 和 8 不在里面，所以状态 2 和 8 是不可达状态。在这里我们分析的是不可达状态，不终结状态可以采用类似的方法来分析，最终删除的不可达和不终结状态是 2 状态和 8 状态。

第二步：合并等价状态

		a	b
+	1	6	3
	3	1	5
-	6	4	
-	5	7	3
	4	4	6
-	7	4	

图 3.44: DFA 变换表

1. 先将原始的状态划分为两个状态集，一个是结束状态集，另一个是非结束状态集。从变换表可以看出来结束态是 5, 6, 7，所以把 5, 6, 7 划分为一个状态集，剩余的 1, 3, 4 是非结束状态集。

令 $Q_{NE} = \{1, 3, 4, 5, 6, 7\}$

2. 第二步逐一去访问当前等价状态集合中的任意两个状态，如果它们读入同一个字符

后跳转到的状态不在同一个等价状态集合里，则把它们分裂到不同的等价集中。

取 {3,4}:

$$\because \delta(1, a) = 6, \delta(3, 4, a) = \{1, 4\}$$

$$\therefore \text{划分成 } Q_1 = \{1\}, Q_2 = \{3, 4\}$$

$$\text{即 } Q_{NE} = \{\{1\}, \{3, 4\}, \{5, 6, 7\}\}$$

首先看 1, 3, 4 状态集，由于状态 1 遇到 a 跳转到状态 6，3, 4 遇到 a 跳转到状态 1,4，其中状态 6 和 1, 4 在不同的等价集合中，即 6 在 5, 6, 7 中，3, 4 在 1, 3, 4 中，所以把 1 和 3, 4 分裂成不同的等价状态集。在这一步，我们可知状态 1 和状态 3，状态 4 肯定不等价，但是状态 3，状态 4 是否等价我们还不知道，还得继续去执行算法。

3. 下一步我们看 3, 4，来确认状态 3 和状态 4 是否等价。

由于状态 3 遇到 a 变成状态 1，状态 4 遇到 a 变成状态 4，由于状态 1, 状态 4 在不同的等价划分里，所以状态 3 和状态 4 肯定不是等价状态。因此，得到了新的等价集合 3 和 4。

4. 第四步继续对 5, 6, 7 执行算法。

取 {5,6,7}: 同理，可划分成 $Q_1 = \{5\}, Q_2 = \{6, 7\}$

最后: $Q_{NE} = \{\{1\}, \{3\}, \{4\}, \{5\}, \{6, 7\}\}$

需要注意的是，划分等价状态集算法适用于不含空边的自动机。若自动机带有空边，可以先用之前学过的算法消空边，接下来再执行划分等价状态集算法。

5. 最后得到了状态 6 和状态 7 是等价状态，因此只需要将 6, 7 合并，如何合并呢？只需要将 6 替换 7，得到的结果如下所示：

	a	b
+	1	3
	3	5
-	6	
-	5	3
	4	6

图 3.45: 合并等价状态

6. 最后将转化表变为自动机。

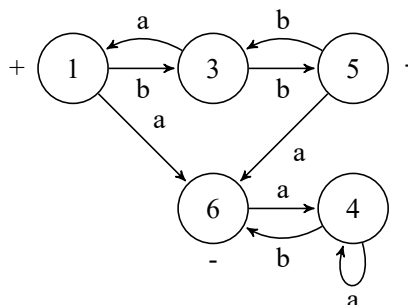


图 3.46: 最小的 DFA

3.4 正规语言描述方法间的相互转换

正规语言有三种等价的表示方法：

1. 正规文法
2. 正规式
3. 有限自动机

任何一个表示方法都能转化为另外两个，我们先看看正规文法和确定有限状态自动机之间的转换方法。

设 $G(Z) = (V_N, V_T, Z, P)$, $DFA = (Q, \Sigma, s, F, \delta)$ 正规文法是一个四元组，确定有限状态自动机是一个五元组，可以发现二者之间是有对应关系的，如下图所示：

正规文法	DFA
V_N (非终结符集)	Q (状态集)
V_T (终结符集)	Σ (字符集)
Z (开始符号)	S (开始状态)
$A \rightarrow aB$	$\delta(A, a) = B$
$A \rightarrow a$	$\delta(A, a) = B$ (结束态)
$A \rightarrow \varepsilon$	A (结束态)

图 3.47: 正规文法与 DFA 之间的对应关系

例 3.19 自动机 \Rightarrow 正规文法：

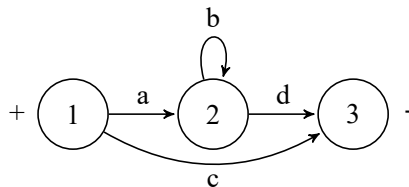


图 3.48: DFA

令 $Z=①$, $A=②$, $B=③$, 则有正规文法

$$G(Z) : Z \rightarrow aA \mid cB, A \rightarrow bA \mid dB, B \rightarrow \varepsilon$$

根据上面的对应关系，可以基于自动机写出正规文法。

相反，我们也能够将正规文法转为自动机：

例 3.20 正规文法 \Rightarrow 自动机，并求 $L(G)$ ：

语法规则

$$G(Z) : Z \rightarrow aZ \mid bA \mid \varepsilon, A \rightarrow bA \mid d$$

首先第一步， $A \rightarrow d$ ，根据上面的对应规则，我们先做一点点变换：

将 $A \rightarrow d$ 变换为 $A \rightarrow dB, B \rightarrow \varepsilon$ 。

这一步的目的是什么？是希望让 B 作为终止态，显性体现在自动机里，经过上述变换后得到一个新的等价文法，唯一的区别是新文法中引入了一个非终结符 B 。

$\therefore G'(Z)$ 与 $G(Z)$ 等价：

$$Z \rightarrow aZ \mid bA \mid \varepsilon, A \rightarrow bA \mid dB, B \rightarrow \varepsilon$$

之后令状态①表示 Z ，状态②表示 A ，状态③表示 B ，画出自动机。

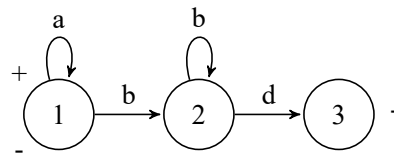


图 3.49: DFA

这个文法对应的语言为：

$$L(G) = \{\varepsilon, a^m b^n d \mid m \geq 0, n > 0\}$$

接下来，我们来看一下正规式和有限自动机状态机的转换。

转换的机制如下：

设 e 为正规式， $DFA = (Q, \Sigma, s, F, \delta)$ ，转换机制：

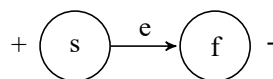


图 3.50: 转换机制

对于正规式 e ，可以看作是一个符号串，我们希望有一个自动机，能够从初始状态读入这个符号串，到达终止状态，这就完成了转化的过程。

定义 3.5 符号串转换为 DFA 的过程称为分解，DFA 转换为符号串的过程称为合成。

以下是转换的规则，图中正方向称为分解过程，负方向称为合成过程：

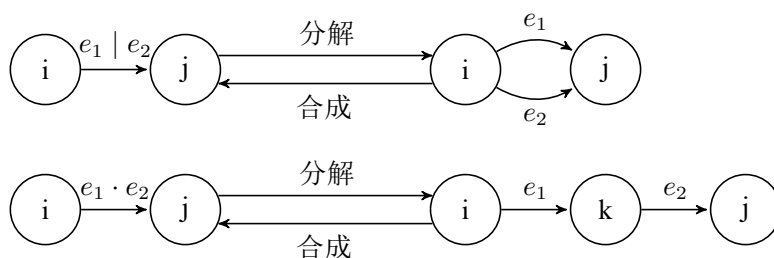


图 3.51: 转换规则

而对于闭包型正规式，有如下 5 种转换方式：

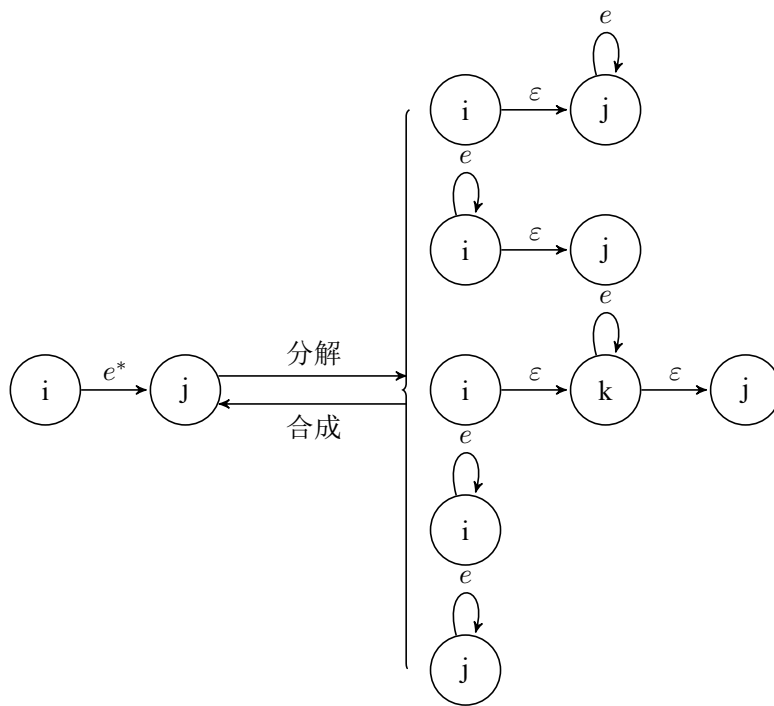


图 3.52: 闭包型正规式转换规则

例 3.21 正规式 \Rightarrow 自动机：

设 $e = a^*b \mid bc^*$

首先根据上述的转化规则转化为一个初始的自动机：

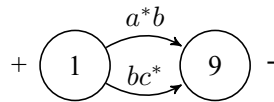


图 3.53: 初始自动机

经过一次分解后自动机变为：

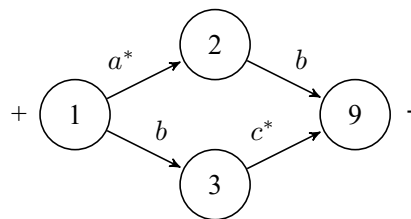


图 3.54: 一次分解后自动机

对闭包型正规式分解后变为：

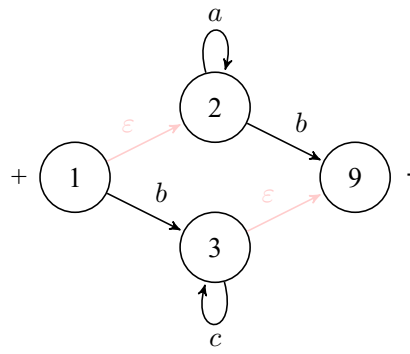


图 3.55: 完全分解后自动机

之后需要对这个带空边的自动机进行确定化，怎么进行确定化呢？先标记起始状态和终止状态，从起始状态开始，经过空边到达的所有节点都标记为起始状态，同样，所有通过空边到达终止状态的节点都标记为终止状态。然后逆向消除空边，第一次消除空边后变成这样：

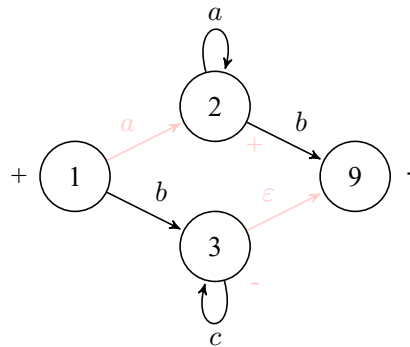


图 3.56: 消除一条空边的自动机

消除第二个空边变成下面这样：

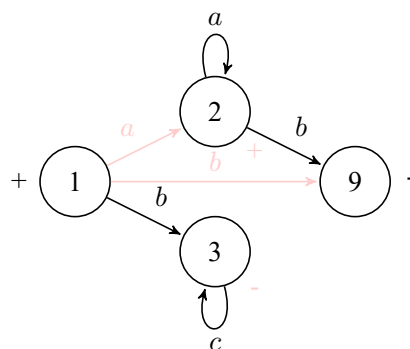


图 3.57: 消除第二条空边的自动机

由于有①②两个起始状态，我们需要进一步进行确定化，用自动机确定化算法，构造出如下表所示的确定自动机的变换表。

	a	b	c
A{1,2}	B{2}	C{3,9}	
B{2}	B{2}	D{9}	
C{3,9}			E{3}
D{9}			
E{3}			E{3}

图 3.58: 自动机变换表

最终得到了确定化后的自动机:

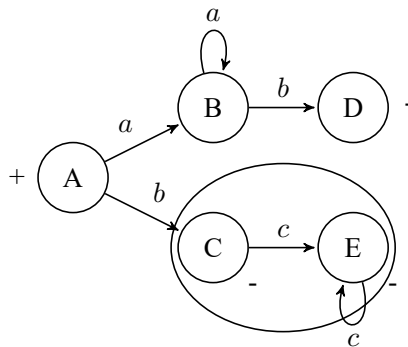


图 3.59: 确定化后的状态机

由变换表3.58可知，C、E 状态等价，最小化后的状态机如下图所示:

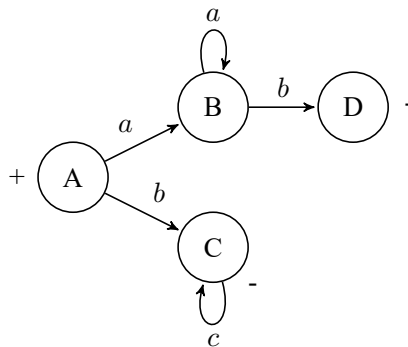


图 3.60: 最小化后的自动机

3.5 有限状态自动机的实现问题

在实际应用时，怎么去设计一个自动机呢？自动机的设计有以下两个说明：

1. 假定自动机只作为识别器，即对待识别的符号串仅回答：是（接受）或否（拒绝）。自动机其实就是这么一个装置，对任意的一个字符串，能接受就是 yes，不能就是 no。
2. 为了便于处理，可令 \forall 作为待识别的符号串的泛指后继符。

我们将这个符号作为符号串的后继符，这个后继符怎么去理解呢？可以理解成把一个字符串的结尾单独用一个符号来表示。

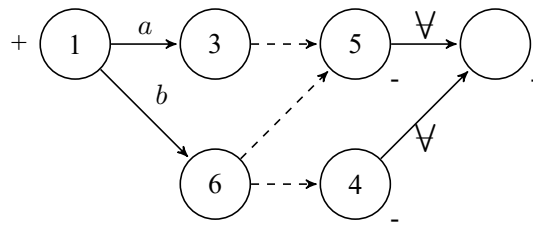


图 3.61: 扩展后的自动机

扩展之后的自动机如上图所示，只有唯一的开始态和唯一的结束态。当然我们可以将自动机表示成一个变换表，其中 no 代表不能接受的后继字符。

	a	b	...	∇
+	1	3	6	...
...
-	4	no	no	...
-	5	no	no	...
				ok

图 3.62: 自动机变换表

3.5.1 控制程序设计

基于上面的说明，给出一个自动机的控制程序流程图：

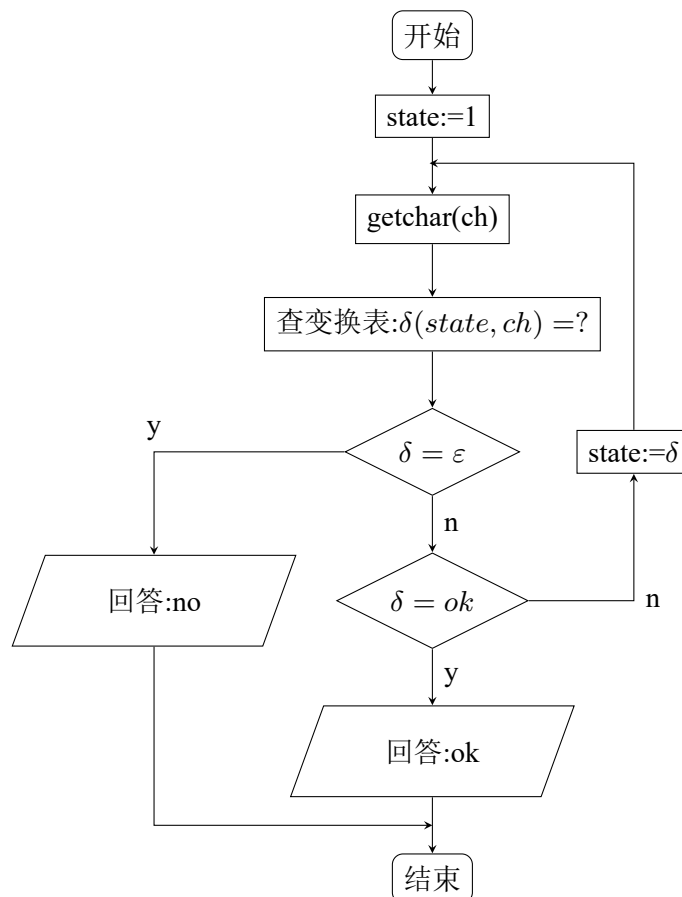


图 3.63: 自动机控制程序流程图

3.5.2 变换表存储结构设计

下面将解决状态转移表即变换表存储的问题，可以有很多存储状态转移表的方式：

1. 二维数组，其下标是 (状态，输入符号)；

※ 为了适应不同编码语言的需要，状态和输入符号可采取相应的编码形式；通常，使用连续的正整数：0,1,2,3,⋯。

二维数组的方式优点是简洁、访问快，提供下标就能直接访问。但是缺点是太占空间。

2. 压缩变换表，方法是把每个状态行作为子表，状态为索引，并把错误的输入符号合并在一起，如下图所示，图中 √（其他）代表错误符号，左侧蓝色索引表中的序号代表数字，右侧红色表为变换子表：

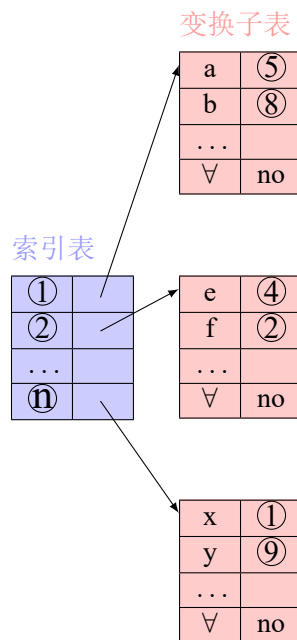


图 3.64: 压缩变换表

我们一般用第二种方法，索引表的形式。

例 3.22 ※ 有限自动机实现示例：

有限自动机 DFA:

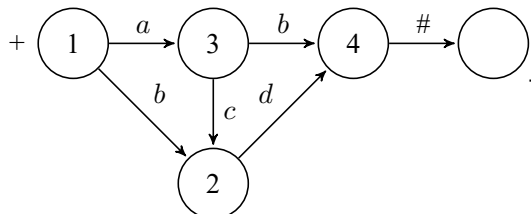


图 3.65: DFA

根据这个有限状态自动机，我们可以用上述的压缩变换表的形式存储自动机状态转移表。

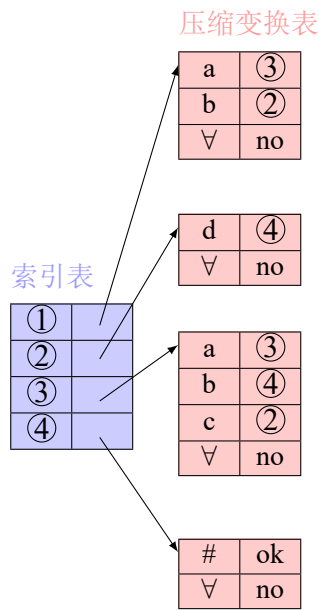


图 3.66: 自动机状态转移表

最后，模拟一下应用压缩变换表的自动机的识别字符串的过程：

state	ch	剩余	变换	备注
1	a	acd#	3	
3	a	cd#	3	
3	c	d#	2	
2	d	#	4	
4	#		ok	接受

图 3.67: 字符串识别过程

第4章 词法分析

编译器分为多个组件。从大的层面上分，分成编译器的前端和编译器的后端两个部分。这里的前后端和 Web 开发不同，编译前端指的是词法分析、语法分析和中间代码生成，即转化为中间代码及之前的内容都称为前端；编译后端指的是和目标机相关的部分，包括中间代码优化和目标代码生成与优化。

前端内容的核心是理解、分析输入的程序，转化成机器内部可以表示、执行的代码。其中第一步，称之为词法分析。词法分析，就是将输入的源程序转化为单词序列。在一个程序里，包括在人类语言里，单词是处理的不可分割的最小单元，即后续的处理以单词为单位进行。在后面语法分析和语义分析中，处理对象均为单词序列。

词法分析器又称扫描器，具体包括两个任务：

- ① 识别单词——从用户的源程序中把单词分离出来；
- ② 翻译单词——把单词转换成机内表示，便于后续处理。

词法分析流程如图4.1所示。

这一章的内容包括词法分析的基本概念、词法分析程序的设计与实现、算术常数处理机的设计与实现。本章对应的思维导图如图4.2所示。

4.1 词法分析的基本概念

4.1.1 单词的分类与识别

先简单讲一下概念，什么是单词。在自然语言，比如在中文里，中国是个单词，太阳是个单词。那在计算机高级程序语言中，什么是单词呢，这个的定义不是特别直接，从我们感性的

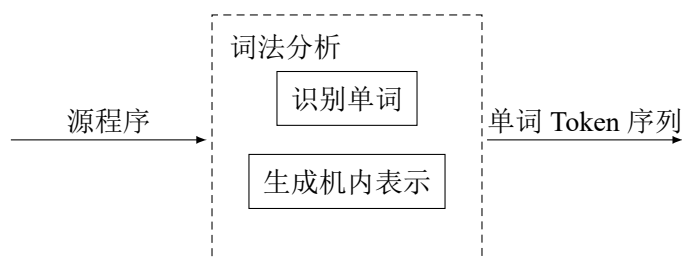


图 4.1: 词法分析流程图

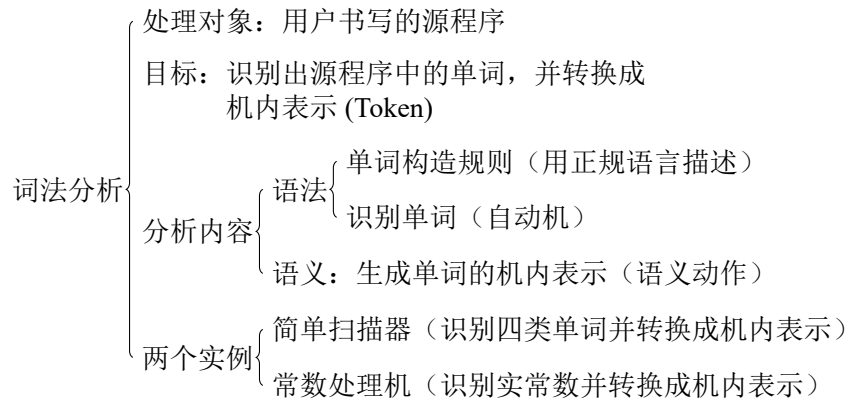


图 4.2: 词法分析思维导图

认识出发，回顾学过的高级程序语言 C，C++ 等，if 是个单词，while 是个单词。可以看出无论是自然语言还是程序语言，**单词是承载语义信息的最小语法单位。**

单词按语法功能可分为标识符、常数、关键字和界符。常数包括算数常数，逻辑常数以及字符串常数等，关键字指系统内部定义的，具有固定意义的，如在 C 语言中的 if、else、while 这些。界符则包括标点、算符，其中算符包括单字符算符、双字符算符。具体地，我们按单词的语法功能，将单词分为：

- ① 标识符——用户给一些变量起的名字；
- ② 常量——以自身形态面对用户和系统；
- ③ 关键字——系统内部定义，具有固定的意义，通常用来区分语法单元；
- ④ 界符——
 - (a) 单字符界符——+ - * / , ; : = < ...
 - (b) 双字符界符——:= <= <> >= == /* */ ...
 - (c) 其他

识别单词的基础是分析单词的构词规则。从单词的构词规则上看，如果字母开头，大概率是关键字或标识符，如果是数字开头，很有可能是数值型常量，如果以 ‘(单引号) 或 “(双引号) 开头，则可能是字符型常量或字符串常量，如果是其他一些没有涵盖到的，如 + - */ 这类的，那它可能是界符或者其他形式的定义。不难发现，我们可以根据上述这些构词规则，区分出不同类型的单词。综上，**单词的识别主要包括以下四种情况：**(构词规则)

- ① 字母开头——关键字或标识符；
- ② 数字开头——数值型常量；
- ③ ‘或 “开头——字符型常量或字符串常量；
- ④ 其它——多数以自身形态识别之，如 +, :=, ...。

进一步，如何区分关键字和标识符？在早期，有的编译程序，强迫用户在输入关键字时，必须用某种特殊符号将其括起来；如：begin→#begin#，或书写时用黑体字，这种方式不友好。现在大多数编译程序使用“保留字”，即标识符不能用关键字。系统预先构建关键字表，拼好的字符串，先查关键字表，查到了，视为关键字，否则，视为标识符。

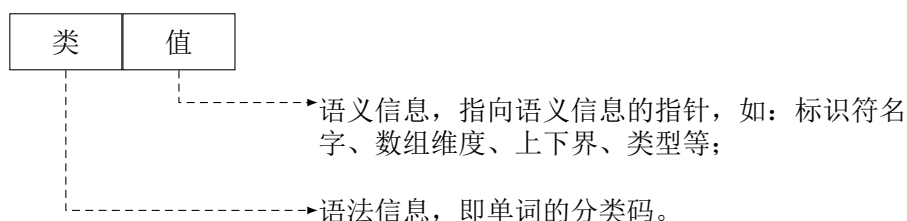


图 4.3: 单词 Token 表示法

4.1.2 单词的机内表示

在计算机中如何表示单词，怎么从编译器的角度看这些单词呢？计算机中通常使用一些结构化、非常简单、易于存储和读取的方式。单词在机器中的表示并不复杂，有以下一些基本的要求。

第一，要求长短统一。比如：对于变量名 A 和 ABC，都是标识符，在编译器内对于这两个字符串，是怎么保存定位呢？假设不考虑结束字符，A 长度 1 个字符，ABC 长度 3 个字符，显然，不同的变量名，需要的存储长度不同。从计算机存储，数据结构组织的角度来看，变长的存储方式过于繁琐，一般来说，我们希望存储的形式是长度统一的，可以用一个固定的结构存储。

第二，要求语法、语义信息分开。这在当前阶段可能不太好理解，简单说来，语法和语义在一个单词的表示中是不一样的。在自然语言中，比如说，“打车”的“打”，“打”从自然语言角度看，语法上表示是一个动作，一个动词，而语义上则要根据“打”的具体使用来定，像“打车”的语义是招手拦车的一个事，如果是“打气”，指的是加油打气，如果是“打饭”，也不是“打车”里的意思，是指去食堂盛饭。在计算机中，每个单词也存在着语法和语义的功能。

由此可以看出，程序设计语言的单词不但类别不同，而且长短不一，机内表示可以使：

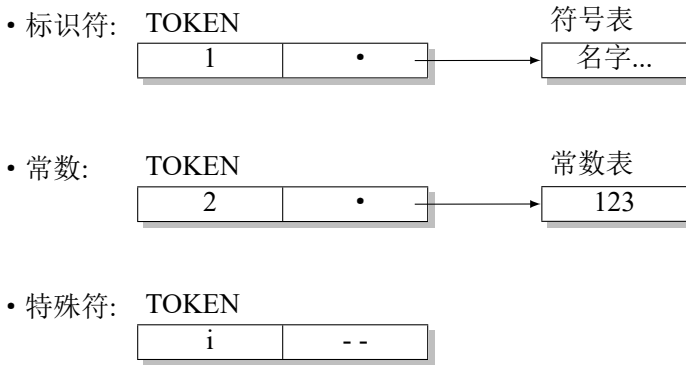
- ① 长短统一；
- ② 语法、语义信息分开。

在词法分析器中，单词 Token 采用二元组的表示形式，一个是类，表示语法信息，也即分类码，一个是值，表示语义信息，实际上存储的不是具体值，而是一个指向符号表的指针，符号表中记录上下界、类型等信息，这部分内容会在后面进行详述。

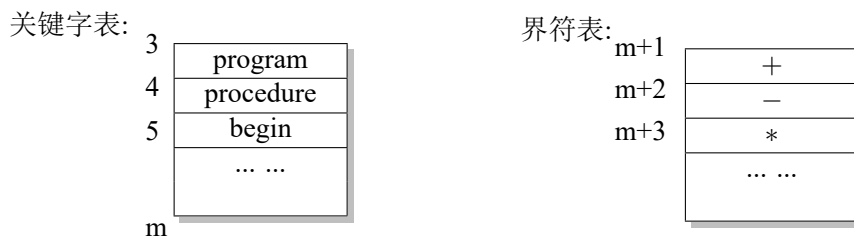
我们先介绍 Token 的表示，如图4.3所示。

假如是一个标识符，对应的 Token，类码写 i，第二项的指针指向标识符表，标识符表其实是符号表内容的一部分，存储标识符的内容。常数对应的 Token 也是两部分构成，第一项是它的类码 c，第二项指向存储常数内容的表，我们称之为常数表。考虑到前面长度统一的要求，例如双精度和单精度表示长度不同，字符型和字符串型长度也不同，所以常数存储在常数表，而不是直接放在 Token 表示的第二项中。关键字对应的 Token，类码用 k 表示，第二项指向关键字表。类似地，对于界符对应的 Token，类码用 p 表示，第二项指向界符表。我们发现，不同 Token 的表示形式，除了类码外结构基本一致，其中指针分别指向对应的表。类码并不是严格规定的，可以自行设计，这里我们习惯用缩写进行定义，而在程序设计时，可以按照 1, 2, 3, ……进行定义。

根据上述分析，给出各类单词对应的 Token 表示的详细设计方案：



其中：i——从 3 开始编码，一个单词一个码！习惯上，关键字先编，其余后编。



4.2 词法分析程序的设计

4.2.1 词法分析程序功能划分

词法分析器的实现，一般有两种方式，第一种方式是独立一遍的扫描器，是将整段程序，输入扫描器，得到 Token 序列，也就是单词串，如图4.4所示。

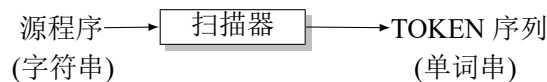


图 4.4: 独立一遍的扫描器

第二种方式是考虑下游系统，将扫描器作为语法分析器的子程序，如图4.5所示。此种方式在收到语法分析器发出的取单词指令后，从源程序中取一个单词并执行，然后将得到的 Token 返回给语法分析器，语法分析器会不断执行这个过程，最终得到整个分析的结果。换言之，Token 序列并没有在单步执行扫描器后，显性表示，而是在语法分析过程中逐步生成，我们把这种方式称为语法制导。在这种方式下，词法分析器不是独立的，而是作为依附于语法分析的一个装置，或一个配件。

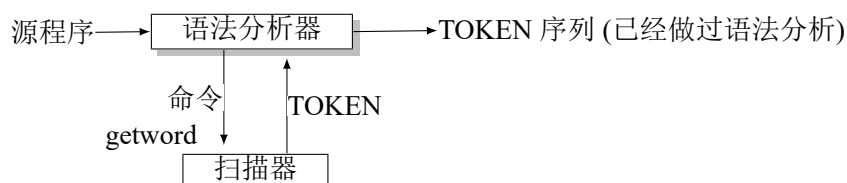


图 4.5: 作为语法分析器的子程序的扫描器

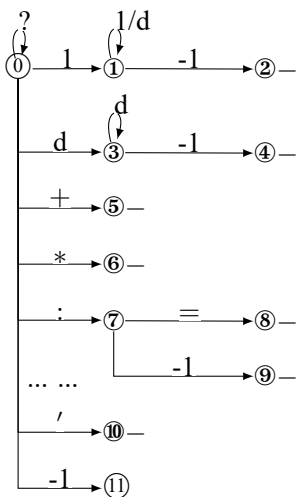


图 4.6: 类 Pascal 语言的 Token 扫描器

单词	编码
program	3
procedure	4
begin	5
end	6
while	7
do	8
+	9
*	10
:	11
:=	12
=	13
,	14
;	15

图 4.7: 关键字和界符表

下面比较这两种方法的优缺点，第一种方式实现简单，词法分析和语法分析相互分离，耦合小，易于调试；第二种方式通过语法驱动，适合语法分析的方式，耦合大。语法驱动的方式读入一次源程序，可同时完成词法分析和语法分析，效率要高于第一种方式。

4.2.2 一个简单词法分析器的实现

词法分析器的核心由两部分构成，第一个是识别器，顾名思义，识别源程序中一个一个单词。第二部分是翻译器，根据识别器所识别出的对象，完成从单词串到单词的 Token 串的翻译。

分析单词的构词规则不难发现，单词可以用正规语言描述，字母和数字等是单词正规语言对应的终结符集，而符合构词规则的标识符、关键字和常数等便是该语言定义的符号串。例如，标识符对应的正规语言为 $L_1 = \{l|d\}^n, n \geq 0\}$ ，其中 l 表示字母， d 表示数字。有限状态自动机是识别正规语言的主要方式，因此构建识别器的核心是构造能够识别源程序中各种单词的有限状态自动机。

翻译器是将识别器识别出的单词转换为统一的机内表示 Token。单词 Token 包括分类码和单词语义信息两部分，因此翻译器的目标便是分析出单词的类别及其语义信息，值得注意的是，在词法分析中，我们获取单词的语义信息主要是单词名字，如标识符名字，其它语义信息将在后续符号表和中间代码生成环节中逐渐扩充。明确了翻译器的目标（即任务），我们可通过为自动机的状态节点设计相应的语义动作函数，来生成单词的分类码和语义信息，即单词 Token。

首先，我们设计一个简单的类 Pascal 语言的 Token 扫描器，如图 4.6 所示，该扫描器的目标是识别四类单词并转换成单词 Token。

下面先介绍自动机描述中的一些符号表示。第一，用“1”表示字母，用“d”表示数字，图中的“.”、“-”都是界符。事实上，整个符号表包括字母、数字和其他一些符号。我们假设用 # 表示程序结束，一旦遇到 #，表示程序结束。

第二，“?”表示空格、回车和换行，这三者构成我们常说的一些格式的变换。自动机的这一部分保证了会把和内容无关的空格、回车和换行过滤掉，不影响后面识别的结果。

第三，带有“-”号标记的状态是结束态，表示已经识别出一个单词。

第四，……表示省略了其他界符的处理。

接下来，我们从不同路径看一下上述的自动机实现的功能。

自动机中第一条路径，这一部分自动机识别出关键字和标识符。对于关键字和标识符，我们要求起始字符是字母，不能以数字等作为起始。我们现在使用的绝大多数编程语言都是这样约定的，目的是为了**避免歧义**。

自动机中第二条路径，没有小数点，识别出来的是无符号整数。

其余路径用于识别 +、*、:= 等界符。

给定一种语言以及该语言定义，就可以相应画出识别 Token 的自动机。

一个简单的词法分析装置如下，给定一个源程序字符串，通过识别器识别得到一串单词，通过为相应状态节点设计**语义动作**函数，可在识别出单词的同时把单词转化为机内表示 Token，也就是确定单词的类码以及语义。

在图4.6所示的 Token 扫描器中，**状态 2** 表明自动机已识别出一个以字母开头后面跟着字母和数字组成的单词，但需要进一步确定该单词是关键字还是标识符。因此，**状态 2** 的语义动作可表述为，查找关键字表，判断是否为系统保留字，如果是，就返回这个关键字所对应的 Token；如果没有查到，则在符号表中查找当前标识符，看是否为已经定义的标识符，若找到则说明该标识符已经定义过，若没有则在符号表中增加这个新的标识符，并返回相应的 Token 表示。**状态 2** 对应的语义动作伪代码如下（即单词转换为 Token 的伪代码）：

```

1   IDorKeyToken (strTOKEN)           //strTOKEN中保存的是已识←
    别出的单词
2   begin
3       code:=Reserve(strTOKEN);       //查关键字表，查到时返回←
        其编码，否则，返回0
4       if (code=0)                   //未查到，因此strTOKEN中←
        的串为标识符
5           begin
6               value:=InsertID(strTOKEN); //将strTOKEN插←
                入符号表，返回当前位置
7               return(1,value);        //生成一个标识←
                符TOKEN
8           end
9       else                             //查到了，因此strTOKEN中←
        的串为关键字
10          return(code,_);            //生成一个关键字TOKEN
11  end

```

其中：关于两个语义动作函数的说明如下，

- Reserve(): 查 strTOKEN 中的字符串是否在关键字表中，查到了，返回其编码，否则，返回 0;
- InsertID(): 用 strTOKEN 中的标识符查符号表，查到了，返回位置指针，否则，将此标识符插入到符号表中，并返回当前位置指针。

状态 4 表明自动机已识别出一个常数单词，图4.6给出的常数识别较为简单，只能识别是否为无符号整数。实际上还可识别出带小数点表示的浮点数，浮点数还可以通过科学计数法，用 e 指数的形式表示，形如 6.26×10^{12} 。不难发现，采用科学计数法表示的浮点数其实是一个用字符串形式表示的常数，生成 Token 时需要将其转换为数值型常数，此种情况下，常数处理对应的也是一个自动机，常数处理是将一个字符串表达的常数，转化为计算机内部真正的表示，一个数值型常数，再填入常数表，得到的常数 Token 就会指向这个表。此处，先给出4.6中**状态 4**对应的语义动作伪代码，关于字符型常数转换为数值型常数的处理，将在下一节做详细介绍。

```

1   ConstToken(strTOKEN);           //strTOKEN中保存的是已←
   识别出的单词
2   begin
3       value:=InsertConst(strTOKEN); //将strTOKEN插入常数←
   表，返回当前位置
4       return(2,value);           //生成一个常数TOKEN
5   end

```

其中：关于语义动作函数的说明如下，

- InsertConst(): 用 strTOKEN 中的常数查常数表，查到了，返回位置指针，否则，将此常数插入到常数表中，并返回当前位置指针。

状态 5、6、8、9、10 用于识别界符，相应的语义动作为：查找界符表，如果查得到，确定是界符，返回相应的 Token，如果查不到，则返回一个错误，表示输入程序出错。日常程序编译时报无法识别的错误，可能就是这样的原因。**状态 5、6、8、9、10 的语义动作伪代码如下：**

```

1   PuncToken(strTOKEN)           //strTOKEN中保存的是已识别出←
   的单词
2   begin
3       code:=Bound(strTOKEN);     //查界符表，查到时返回其编←
   码，否则，返回0
4       if (code=0)                //未查到，因此是非法字符
5           ProcError();           //错误处理
6       else                        //查到了，因此是界符
7           return(code,_);        //生成一个界符TOKEN
8   end

```

其中：关于语义动作函数的说明如下，

- **Bound()**: 查 strTOKEN 中的字符串是否在界符表中，查到了，返回其编码，否则，返回 0。

基于上述分析，给出完整的词法分析器伪代码，详见本章附录 1。

综上，可以看出，设计一个简单的词法分析器主要包括三个步骤：

1. 分析单词的构词规则，明确单词的形式化描述方法。高级程序语言中单词可用正规语言描述；
2. 构造有限状态自动机，用于识别单词；
3. 根据词法分析的目标（将单词转换为机内表示），设计语义动作，并插入自动机的相应状态处。

4.2.3 词法分析示例

这里给出一个 Pascal 程序片段，下面给出采用上一小节介绍的词法分析方法，生成该程序片段对应的 Token 序列的详细过程。

Pascal 程序片段如下：

```

1      x1:=x1+1;
2      begin
3          yy:=5;
4          zz:=yy*x1;
5      end;
6      while (x1=20) do
7          x1:=yy;

```

首先，对于这样一段程序，我们需要关键字表，含有程序预定义好、具有特殊语义的一些关键字，如表中给出的 **Begin**, **End** 等，我们可以对关键字进行编码。需要界符表，包括识别出来的界符，如 `::`、`:=` 等，同样可以进行编码。还需要标识符表，保存变量名，到后续符号表的学习，会进一步了解变量类型，物理地址等。还有常数表，保存程序中出现的常数。其中，关键字表和界符表是在编译程序之前由系统设定好的，而标识符表和常数表则是在处理用户输入程序过程中动态变化的。

下面，根据这段程序进行分析。给定源程序后，通过图 4.6 所示的扫描器，也就是词法分析部分，对它进行处理。开始的时候，假设指针指向整个源程序的第一个符号 `x`，然后来看整个流程。在我们取到 `x` 后，先通过识别器，来识别这个符号，当前起始状态是 0，读入 `x`，进入 1 状态，再读入数字 1，也就是图中在 1 状态时读入表示数字的 `d`，仍保持在 1 状态，后面是一个空格，不是字母或数字，则进入状态 2，当前单词识别结束，得到一个识别好的字符串 x_1 ，在状态 2，执行对应的语义动作 `IDorKeyToken (strTOKEN)`，以判断该单词类别并生成相应的 Token。语义动作 `IDorKeyToken (strTOKEN)` 的执行过程可描述为，先查图 4.7 所示的关键字和界符表，

符号表 I	常数表 C
x_1	1
yy	5
zz	20

图 4.8: 符号表和常数表

表中没有名为 x_1 的关键字，接着判断是否为标识符表里的内容。这里需要注意，在刚得到 x_1 ，进行查表的时候，标识符表里是没有 x_1 信息的，需要在标识符表中填写 x_1 的信息，由此完成这个 Token 的识别，最终得到的结果是 (1,I1)，其中 1 表示 x_1 的类型是标识符，I1 表示指向标识符表第一项。

继续执行，识别得到界符=，进入状态 8，并执行状态 8 对应的语义动作 PuncToken(strTOKEN)。通过查图 4.7 所示的关键字和界符表，确定是界符，对应的代码为 12，因此得到该界符对应的 Token 信息 (12,_)。

再向下执行，可以再识别得到一个 x_1 ，进入状态 2，执行对应的语义动作 IDorKeyToken(strTOKEN)。这个 x_1 的 Token 表示结果是 (1,I1)，因为在识别过程中，进行查表的时候，标识符表里已经有 x_1 ，会直接返回表里存在的索引，不需要填表。以此类推，可以得到 + 的 token 表示是 (9,_)。得到常数 1 的 token 表示过程中，需要填常数表，得到 (2,C1) 的表示。以此类推，最终得到本节一开始给出的程序片段对应的标识符表、常数表如图 4.8:

上述 Pascal 程序片段对应的 Token 序列:

(1, I1), (12, _), (1, I1), (9, _), (2, C1), (15, _),
 (5, _),
 (1, I2), (12, _), (2, C2), (15, _),
 (1, I3), (12, _), (1, I2), (10, _), (1, I1), (15, _),
 (6, _), (15, _),
 (7, _), (1, I1), (13, _), (2, C3), (8, _),
 (1, I1), (15, _), (1, I2), (15, _)

4.3 算数常数处理机设计

从上一节可以看出，我们在识别单词有限状态自动机中，加入语义动作函数，便可得到一个简单的词法分析器。给定源程序，经过词法分析器分析后便可得到一个 Token 序列。但是需要注意的是，在简单词法分析器中对于实常数的处理，只考虑了最简单的形式，即无符号整数。如果源程序中的实常数是用字符串来描述的，比如将 6.26×10^{12} 赋值给 a 的语句， 6.26×10^{12} 在源程序中是一个字符串，而在机器处理时，需要将字符型实常数 6.26×10^{12} 转换为数值型实常数 6.26×10^{12} ，这里面临的问题是如何将字符型的量转化为数值型的量，即生成字符型实常数的机内表示 Token。

同样的，字符型实常数可以用正规语言来描述，因此，字符型实常数的识别仍可采用有限

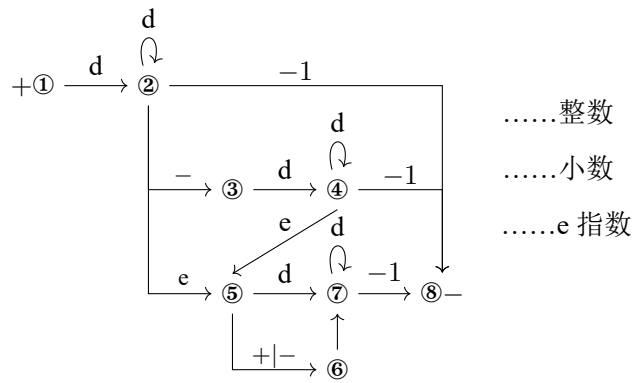


图 4.9: 识别实常数的自动机

状态自动机来识别，同时在自动机中加入负责数值转换的语义动作，形成常数处理机，从而完成字符型实常数到数值型实常数的转换。

4.3.1 识别器设计

一个实常数需要满足如下的形式，该形式不是唯一的表示形式，其中 e 指数部分是可选的。小数点前是整数部分，小数点后是小数部分。 e 前面称之为尾数部分， e 后面称之为指数部分。

Pascal 常数的文法：画图 例： 128.67×10^{-12}

接下来，设计一个识别实常数的自动机，如图4.9所示。首先它的起始状态是 1， d 表示数字，“-1”泛指后继符，2 状态通过不断读入数字 d ，来识别整数部分。我们约定至少要有有一个数字，如果一个数字都没有，就只能停留在 1 状态，读入一个数字 d ，就会到达 2 状态。对于 3 状态和 4 状态，这一路径用来识别小数，到 2 状态时是整数，读入一个小数点，跳转到 3 状态，此时知道这是一个小数，3 状态只能跳转到 4 状态，4 状态不断读入数字 d ，作为小数部分，直到读到其他字符。如果不是 e ，那么当前小数识别结束，如果是 e ，就到达 5 状态，当然 5 状态也可以从 2 状态到达。表示指数的 e 出现在一个整数或者小数后边，是科学计数法的表示。对于 5 状态，或者走上面的路径，或者走下面的路径，下面的路径带加减号，表示幂指数的正负，如果不带加减号，读入数字 d ，就默认指数部分是正数。到了 7 状态不断读入数字，作为指数部分，最后到达结束状态 8。

其中：“ d ”表示数字，“-1”泛指单词的后继符。

4.3.2 翻译器设计

从上一节介绍的简单扫描器可知，自动机可以识别出源程序中的单词，通过翻译器即相应的语义动作可完成单词到机内表示 Token 的转换。在常数处理机中也是同样的，通过图4.9所示的自动机可以识别出字符型表示的实常数，翻译器将其转换为统一的机内表示 Token。

单词 Token 包括分类码和单词语义信息两部分，因此翻译器的目标便是分析出单词的类别及其语义信息，在常数处理中，语义信息是指字符型实常数对应的数值型实常数。明确了翻译器的目标（即任务），我们可通过为自动机的状态节点设计相应的语义动作函数，来生成单词的

分类码和语义信息，即单词 Token。

对于这一设计，我们约定，一个数由尾数和指数两部分组成，N 表示尾数部分数值，当读尾数部分时，读入一个数字，就将之前的尾数部分乘以 10，再加上当前读入 d 的数值；P 表示指数部分数值，和尾数部分计算方法类似，读入一个新的数字 d，就把当前的指数乘以 10 并加上 d 的数值，比如读入 8， $0*10$ 再加上 8，就是 8，接着如果跟着 2，把 8 乘 10 再加上 2，就是 82。表示 N 时没有考虑小数点位置，我们还需要记录小数位数，用 m 表示小数位数，读入一位小数，就把 m 加 1。用 e 表示指数部分的符号变量，做一个简单的取值，正的用 1 表示，负的用 -1 表示。综合以上，我们还能知道一个变量是整型或实型，用 t 表示是整型或实型，约定整数为 0，浮点数为 1。最终可得结果变量的计算公式： $NUM := N * 10^{e*P-m}$ 。

通过上述分析可知，为了计算常数值，引入了如下变量：

- N 表示尾数部分数值；
- P 表示指数部分数值；
- m 表示小数位数；
- e 表示指数部分的符号变量，指数为正用 1 表示，为负用 -1 表示；
- t 表示是整型或实型，约定整数为 0，浮点数为 1。

为了完成转化的任务，我们在图 4.9 所示自动机的状态节点处设计语义动作函数，并假设在 i 状态结点处的语义动作为 q_i 。

1. q_1 : 初始化, $N:=P:=m:=t:=0$; $e:=1$ (表示默认是正数); $NUM:=0$;
2. q_2 : 读入整数部分, $N:=10*N+(d)$, 其中 (d) 表示将字符 d 转换为数字 d;
3. q_3 : 读入小数点, 表示为小数, $t:=1$;
4. q_4 : 读入小数部分 $N:=10*N+(d)$, $m:=m+1$;
5. q_5 : 读入表示指数的 e, $t:=1$;
6. q_6 : 读入指数部分符号, 如果读入负号, 则 $e:=-1$;
7. q_7 : 读入指数部分, $P:=10*P+(d)$;
8. q_8 : 拼实常数, 存入常数表并生成 Token, $NUM := N * 10^{e*P-m}$, $Token=(2, C)$, 其中 C 为指向常数表中 NUM 的指针。

有了上述 8 个语义动作，我们执行前面的自动机，便能把字符串型的数值常量识别出来并自动翻译成一个计算机能够识别表示的数字。常数处理机的实现包括两部分，状态转换表设计和主控程序设计。根据图 4.9 所示自动机可以很容易得到状态转换表，如图 4.10(a) 所示。常数处理机主控程序则是在通用自动机主控程序上加入语义动作即可，即转换到某一状态时，要执行该状态对应的语义动作，如图 4.10(b) 所示，其中 Semfun(state) 表示执行状态 state 对应的语义动作。

接下来，通过一个具体的例子描述常数处理机的执行过程。假设待处理的字符串为 8.67e-12#，该字符串的处理过程如图 4.11 所示。

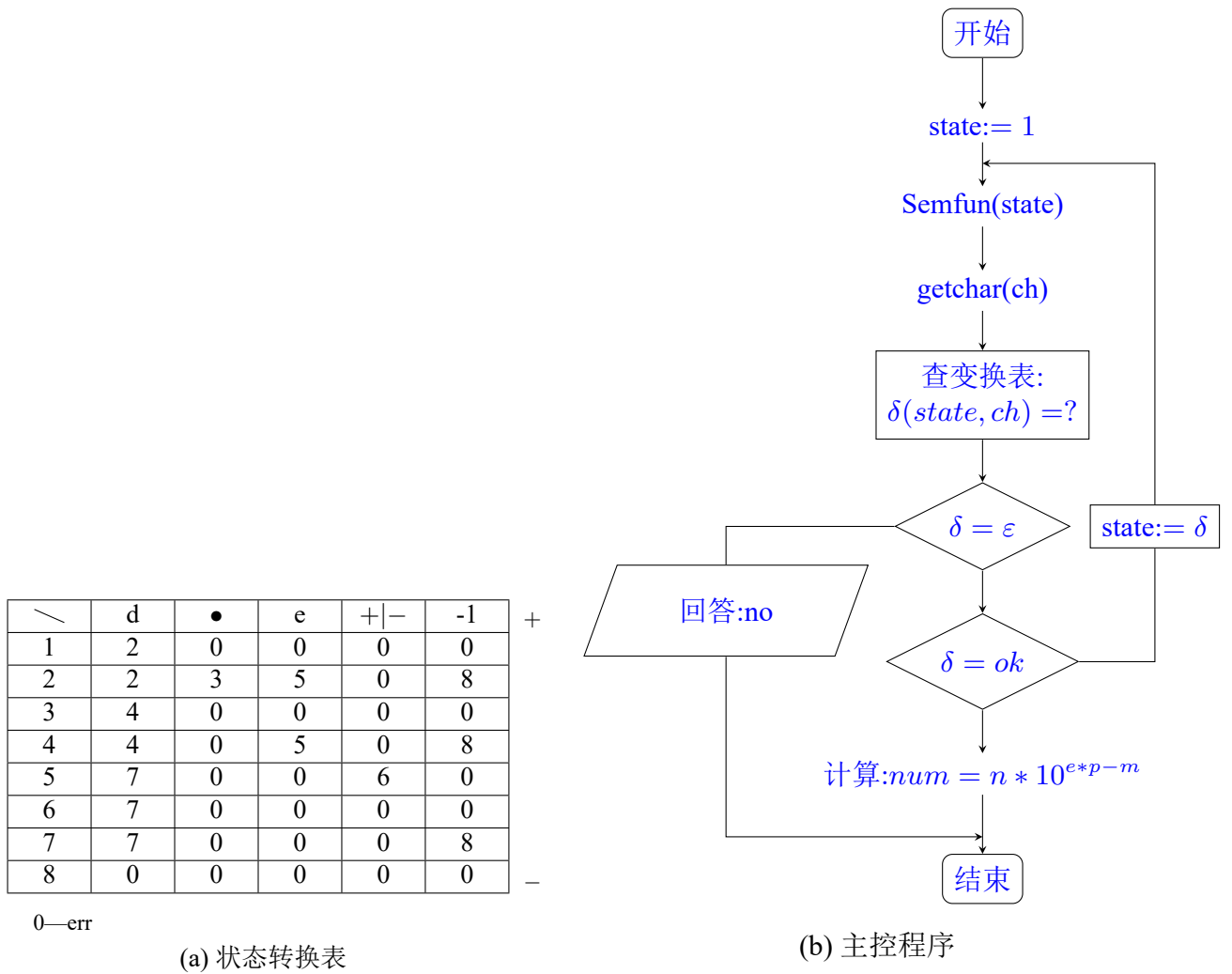


图 4.10: 常数处理机状态表和主控程序

状态	翻译	n	m	p	e	t	符号	变换
1	q(1)	0	0	0	1	0	8	2
2	q(2)	8	0	0	1	0	.	3
3	q(3)	8	0	0	1	1	6	4
4	q(4)	86	1	0	1	1	7	4
4	q(4)	867	2	0	1	1	e	5
5	q(5)	867	2	0	1	1	-	6
6	q(6)	867	2	0	-1	1	1	7
7	q(7)	867	2	1	-1	1	2	7
7	q(7)	867	2	12	-1	1	#	ok

图 4.11: 常数处理过程示例

起始状态是 1 状态，执行 1 状态的语义动作 q_1 ，进行初始化。读入数字 8，查状态表得知要变换到 2 状态。接下来，执行 2 状态的语义动作 q_2 ，更新 N。指针向后移动，读入小数点，2 状态遇到小数点，经过查表，得知到达 3 状态。到达 3 状态，执行 3 状态的语义动作 q_3 ，令 $t=1$ ，表示这是带小数的，接下来读入数字 6，3 状态读 6，到 4 状态。在 4 状态，执行 4 状态的语义动作 q_4 ，处理尾数结果，并计算小数位数，根据计算公式，更新 n 、 m 、 p 、 e 、 t 的值，接下来，4 状态读入 7，仍然是 4 状态。这一步比较容易，同样更新 n 、 m 、 p 、 e 、 t 的值，再读入 e ，4 状态读入 e ，到达 5 状态。执行 5 状态的语义动作 q_5 ，令 $t=1$ ，读入“-”，进入 6 状态。执行 6 状态的语义动作 q_6 ，令 $e=-1$ 。接下来读入数字 1，进入 7 状态。执行 7 状态的语义动作 q_7 ，处理指数部分，根据公式更新 p 的值，读入数字 2，仍在 7 状态。同样执行 7 状态的语义动作 q_7 ，更新 p 的值，最终读入 #，进入状态 8。执行语义动作 q_8 ，得到数值转换结果 $NUM=867 * 10^{-14}$ 和机内表示 Token。

附录 1

完整的简单词法分析器伪代码如下：

```

1 int code, value; //TOKEN 结构
2 strTOKEN := “ ”; // 字符数组，存放构成单词←
   符号的字符串
3 ch := GetChar(); // 读当前字符到 ch
4 ch := GetBC(); // 读一个非空字符到 ch
5 if (IsLetter(ch)) // IsLetter(ch)---判断←
   ch 是否为字母
6   begin // ch 为字母，以下拼标识符←
   或关键字
7   while (IsLetter(ch) or IsDigit(ch)) // 拼单词
8     begin
9     Concat(); // 将 ch 中字符连接到 strTOKEN←
   中
10    ch := GetChar();
11    end
12    Retract(); // Retract()---当前符号位置减 1
13    code := Reserve(); // 查关键字表，查到时返回其编←
   码，否则，返回 0
14    if (code = 0) // 未查到，因此 strTOKEN 中的串为标←
   标识符
15      begin
16      value := InsertID(strTOKEN); // 将 strTOKEN 插入符号表，返←
   回当前位置
17      return(1, value); // 生成一个标识符 TOKEN

```

```

18     end
19     else //查到了，因此strTOKEN中的串为关←
        键字
20     return(code, _); //生成一个关键字TOKEN
21     end
22 else if (IsDigit(ch)) //ch中不是字母，因此IsDigit(ch)←
    ---判断ch是否为数字
23     begin //ch为数字，以下拼数字
24     while (IsDigit(ch))
25         begin
26         Concat();
27         ch:=GetChar();
28         end
29     Retract();
30     value:=InsertConst(strTOKEN); //将strTOKEN插入常数表，←
        返回当前位置
31     return(2,value); //生成一个常数TOKEN
32     end
33 else //判断ch是否是界符
34     begin
35     Concat();
36     if (ch=':')
37         begin
38         ch:=GetChar();
39         if (ch='=')
40             Concat();
41         else
42             Retract();
43         end
44 code:=Bound(); //查界符表，查到时返回其编码，否←
        则，返回0
45 if (code=0) //未查到，因此是非法字符
46     ProcError(); //错误处理
47 else //查到了，因此是界符
48     return(code, _); //生成一个界符TOKEN
49 end.

```

第5章 语法分析

语法分析是指对给定的符号串，判定其是否是某文法的句子。通俗讲，就是判定用户源程序中是否有语法错误。给定一个用户源程序，通过第四章词法分析方法能够得到源程序对应的单词 Token 串（即符号串，每一个单词 Token 可看作一个符号），语法分析的目标是分析该单词 Token 串是否是某高级程序设计语言文法的句子，即单词 Token 串是否满足某高级程序设计语言文法。同时，在语法分析的过程中，我们还可以得到用户源程序（单词 Token 串）的结构，即语法树。

基于文法的语法分析，主要有两类方法，一种是自顶向下的，即基于文法推导的方法，一种是自底向上的，即基于文法进行规约的方法。这两种方法中包含多种算法，在本书中，将介绍自顶向下算法包括递归下降子程序法以及 LL(1) 方法，自底向上算法包括 LR() 分析法以及简单优先分析法。

这一章的内容包括语法分析的基本概念，以及常用语法分析方法的设计与实现。

5.1 语法分析的基本概念

给定一个符号串 α 和一个文法 $G(Z)$ ，语法分析用于判定 α 是否是文法 $G(Z)$ 的句子。基于文法的语法分析，主要有两种方法，一种是自顶向下的，即基于文法推导的方法，一种是自底向上的，即基于文法进行规约的方法。语法分析定义如下：

定义 5.1 语法分析是指对给定的符号串 α ，判定其是否是某文法 $G(Z)$ 的句子。即

1. 自顶向下法（推导法）

自顶向下法是从文法的开始符号出发，自顶向下构造语法树，不断地使用文法规则进行推导，即用文法规则的右部替换左部的非终结符，最终试图使树叶全体恰好是给定的符号串 α ，即对给定的符号串 α ，判定其是否存在 $Z \xrightarrow{+} \alpha$ ；其中： Z 是开始符号。我们约定，当一个句型中有多个非终结符可以进行推导替换时，优先替换最左非终结符，即习惯上采用“最左推导法”。

例 5.1 给定文法 $G(E)$ ，给定一个符号串： $\alpha = a*(b+c)$ ，判断 α 是否是文法的合法句子？

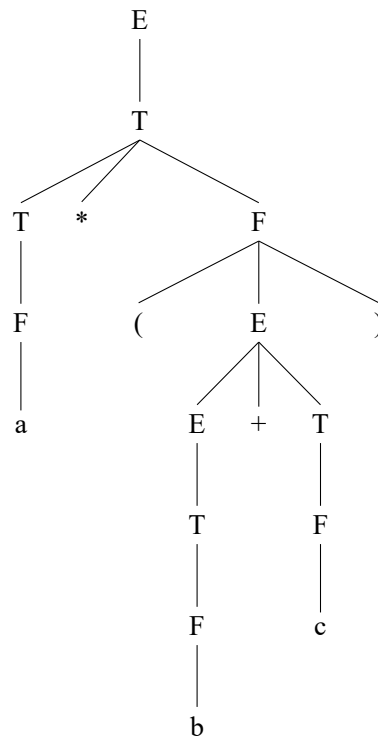


图 5.1: 基于推导的语法树

文法定义

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

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

$$F \rightarrow i \mid (E)$$

采用自顶向下的语法分析思路如下:

1. 分析过程: $E \Rightarrow T \Rightarrow T * F \Rightarrow F * F \Rightarrow a * F \Rightarrow a * (E) \Rightarrow a * (E + T) \Rightarrow a * (T + T) \Rightarrow a * (F + T) \Rightarrow a * (b + T) \Rightarrow a * (b + F) \Rightarrow a * (b + c)$

2. 构造语法树

从上述分析过程不难看出, 若分析的第一步不是选择 $E \rightarrow T$, 而是选择 $E \rightarrow E+T$, 则将导致分析失败。可见, 对于自顶向下的算法, 主要解决的是当同一个非终结符有多个产生式时, 如何选择正确的产生式进行推导, 即自顶向下分析的关键技术是如何确定具有相同左部的产生式之侯选者。

2. 自底向上法 (归约法)

自底向上法是从给定的符号串 α 出发, 自底向上构造语法树, 不断地使用文法规则进行规约, 即用文法规则的左部非终结符替换右部的符号串, 最终试图使树根是且仅是文法的开始符号, 即对给定的符号串 α , 判定其是否存在 $\alpha \xrightarrow{+} Z$; 其中: Z 是开始符号。我们约定, 当一个句型中有多个简单短语可以进行规约时, 优先规约最左简单短语, 即习惯上采用“最左规约法”。

对于例 5.1, 我们采用自底向上的语法分析思路如下:

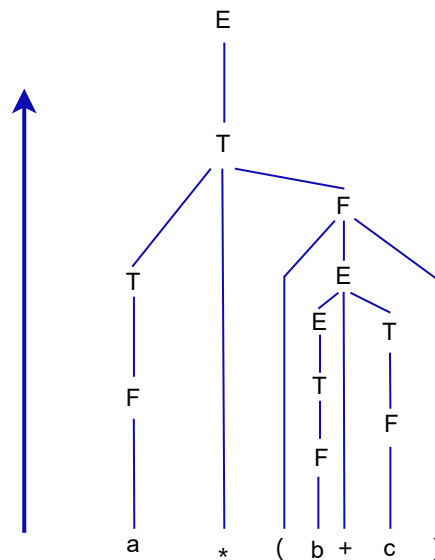


图 5.2: 基于规约的语法树

1. 分析过程: $a * (b + c) \Rightarrow F * (b + c) \Rightarrow T * (b + c) \Rightarrow T * (F + c) \Rightarrow T * (T + c) \Rightarrow T * (E + c) \Rightarrow T * (E + F) \Rightarrow T * (E + T) \Rightarrow T * (E) \Rightarrow T * F \Rightarrow T \Rightarrow E$
2. 构造语法树: 从树叶向树根方向构造语法树。

从上述分析过程不难看出, 若分析的第三步不是选择 $b \rightarrow F$, 而是选择 $T \rightarrow E$, 则将导致分析失败。可见, 规约也是相同的道理, 无论是哪一种自底向上的算法, 需要关注它是如何通过当前的句型, 来判断句柄, 即自底向上分析的关键技术是如何确定当前句型的句柄。

在语法分析过程中, 如果没有任何外部支持, 很难进行规则的选择判断, 而一旦判断错误, 就会连锁导致后面的推导失败。最简单的方法, 就是利用回溯方法, 所谓回溯方法, 就是把所有产生式规则都平等对待, 随机选一个, 走不通再换其他规则, 理论上来说一定能找到正确的推导过程, 但是效率较低。所以, 我们后续讲到的语法分析方法, 不论是基于推导的还是基于规约的方法, 本质上都是为了提高效率。如果文法不存在二义性, 正确的分析过程是唯一的, 我们希望找到一种策略, 或者一些知识来指导系统, 不进行回溯, 以提高语法分析效率。

5.2 递归子程序法

递归子程序法, 又名递归下降子程序法, 是一种自顶向下分析方法, 也就是一种基于推导的方法。从开始符号出发, 对每一个非终结符, 构造一个子程序, 用以识别该非终结符所定义的符号串。递归子程序法的重点在于“递归”以及“子程序”。所谓“递归”, 就是程序不断地直接调用或间接调用自己。文法的递归性, 使得通过有限的产生式, 能够表达无限的语句集合。若文法是递归的, 则子程序也递归, 故称递归子程序。

递归下降子程序法比起回溯方法, 能够准确判断每一步推导选用的产生式, 没有试错后的回溯, 效率有所提升, 但由于递归有大量进栈弹栈操作, 效率还有进一步提升的空间。该方法的优点在于简单且易于实现。

递归子程序法的关键在于如何选择正确的候选式来完成推导的过程。递归子程序法的基本

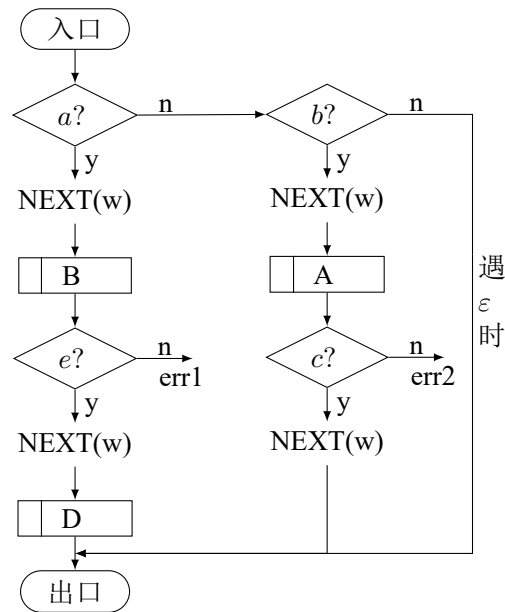


图 5.3: 例 5.3 对应的递归子程序

原理是为文法中每一个非终结符构造一个子程序，来识别产生式右部；子程序以产生式左部非终结符命名，以产生式右部构造子程序内容。

5.2.1 递归子程序法示例

递归子程序法的核心思想是为文法中每个非终结符构造一个子程序，子程序以产生式左部非终结符命名，以产生式右部构造子程序内容。文法中会有很多相同左部的产生式，通常我们将同一个非终结符的产生式，都放到同一个子程序中。

下面通过具体例子来分析递归子程序法的原理。

例 5.2 给定文法： $A \rightarrow aBeD (1) \mid bAc (2) \mid \epsilon (3)$ ，非终结符 A 对应的递归子程序如图 5.3 所示。

其中，NEXT(w) 读的是下一个待处理的单词，即 token 串 (token 序列) 中下一个单词。

在语法分析过程中，我们用当前符，表示正在处理的字符。如图 5.5 所示，产生式 (1) 中第一个字母是 a，判断当前符是否为 a，如果不是，则出错，如果是，则读入下一个字符，再进入子程序 B 中，对于 A 这个子程序来说，它并不会去判断 B 里面的内容，而是直接调用 B 的子程序，B 的子程序负责匹配 B 生成的串。从 B 子程序返回后，判断当前符是否为 e，与前面类似，如果不是，则出错，如果是，再读入下一个字符，并进入子程序 D，从 D 返回后，即可到达出口。

图 5.5 所示子程序入口处读入的当前符不是 a，则判断是不是 b。如果是 b，就再取一个字符，然后调用 A 的子程序。A 处理完之后，判断是不是 c。如果不是 c，则报错；如果是 c，那第二个分支执行完毕。

图 5.5 所示子程序入口处读入的当前符既不是 a，又不是 b，是不是应该报错呢？假设是 f，在这里 f 不一定是 A 推出来的，那 f 为什么能进入到这个程序里呢？可能是这样的情况，例如文法中还有一条产生式 $C \rightarrow Af$ ，可以看出 f 是由调用 A 的子程序 C 生成的；A 推出来的是空，因

为是 C 调用 A，所以 C 要处理 A 后面的 f。因此当前符既不是 a，又不是 b 时，可执行子程序中第三条分支，返回到调用子程序 A 的程序中，由其进行后续处理。

通过上述分析不难看出，同一个非终结符对应的多个产生式放到同一个子程序中，每一个产生式对应子程序中一个分支。

读者可能会对何时读入下一字符存在疑问，这里给出对于接口问题的约定：

1. 进入子程序前，当前符已准备好，图中进入子程序 B 前，准备好了当前符。
2. 子程序出口前，当前符已读进来，图中判断当前符是否为 e，这里的当前符在子程序 B 中已读入。

接口的约定保证了调用任何一个子程序前，当前符是有效的，即已经读入，但没有被其他程序使用，而当匹配成功后，一定要读入下一字符，更新当前符。如果缺少对于接口的约定，整个程序都会混乱。

接下来，给出使用图 5.5 所示子程序进行语法分析的示例。

例 5.3 给定用户输入符号串 $\alpha=abed$ ，采用递归子程序法判断符号串 α 是否能由例 5.3 中文法推出。

分析：用 ω 表示当前符。递归子程序入口处首先判断当前符是否为 a，而此时当前符 $\omega=a$ ，发现匹配成功，如果当前符不是 a，则会进行后续判断。匹配成功后，读入下一个字符，更新当前符 $\omega=b$ ，因为产生式中接下去的字符 B 是非终结符，非终结符 B 有自身对应的子程序，因此直接进行调用该子程序，B 子程序中具体执行步骤，这里不用考虑。B 处理完后，会自动更新当前符 $\omega=e$ ，与产生式中对应进行匹配，如果匹配失败，就出错，发现匹配成功，读入下一个字符，并更新当前符 $\omega=d$ 。产生式中接下去的字符 D 是非终结符，因此直接进行调用 D 对应的子程序，如果能从 D 中正确地返回，就到达出口。

5.2.2 递归子程序构造方法

递归子程序法为每一个非终结符构建一个子程序，为构建一个完整程序，我们还需要一个主程序。给定文法 $G(E)$ ，其中 E 为文法初始符。上一节中关于子程序入口，约定进入每一个子程序前，都需要准备好当前符，而利用 E 产生式构建的子程序作为主程序，会出现缺少当前符问题，因此我们无法将文法初始符的子程序作为主程序。为解决这一问题，我们会增加一个新的产生式 $Z \rightarrow E$ ，这个产生式很简单，左边是一个新的符号，右边是原本的文法初始符。把这个新产生式构造的子程序作为主程序，帮助第一个子程序，读入当前符。给定一个文法 $G(E)$ ，其中 E 为文法初始符。递归子程序构造包括以下几个步骤：

1. 扩展文法：定义一个新的产生式 $Z \rightarrow E$ 。把这个新产生式构造的子程序作为主程序，用于将待分析符号串的第一个字符读进来。
2. 子程序内容设计：
 - (a) 遇到终结符：判断当前符与该终结符是否一致，如果一致，则读下一个字符；
 - (b) 遇到非终结符：调用非终结符的子程序，在非终结符返回之后不需要再读一个字符，

因为返回的时候，已经把下一个待处理的字符读进来了；

(c) 遇到空串：直接连接出口。

下面举例说明递归子程序构造方法。

例 5.4 给定文法 $G(S): S \rightarrow aAb|bS, A \rightarrow cd|\epsilon$. 试构造该文法的递归子程序。

首先，构造主程序。增加一个新的产生式 $Z \rightarrow S$ ，把这个新产生式构造的子程序作为主程序，帮助第一个子程序 S ，读入当前符。然后调用原来文法初始符 S 的子程序，返回后，判断当前符是否为结束符 $\#$ ，如果不是则出错，如果是则分析成功，程序结束。主程序如下图所示。

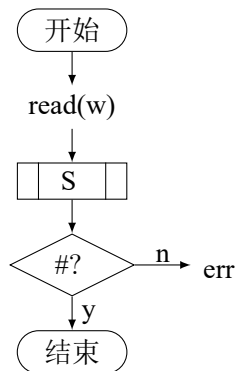


图 5.4: 例 5.4 的主程序

非终结符 S 的子程序，如图 5.7 所示。首先要判断当前符是不是 a 。如果是 a ，就再读一个字符（单词），这里调用 A 的子程序， A 执行完之后，无需 $next(w)$ 函数读下一个字符，因为 A 的出口会把下一个单词读进来。接着判断 b ，如果是 b ，则要把下一个字符读进来。 S 子程序入口处判断当前符不是 a ，则判断是不是 b ，如果是则读下一个字符，然后调用 S 。调用完 S 后，不需要再读字符，直接出来就可以。如果既不是 a ，也不是 b ，就会报错。

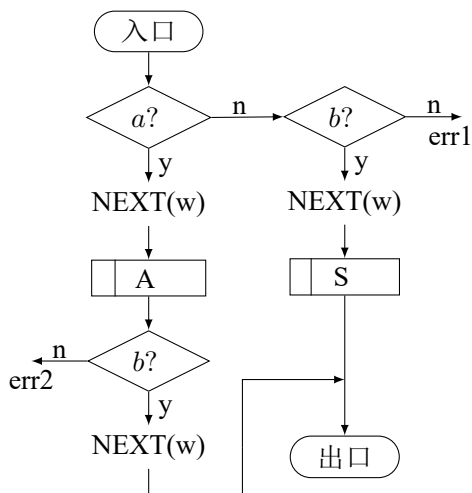


图 5.5: 例 5.4 的子程序 S

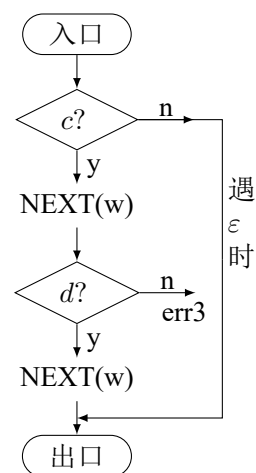


图 5.6: 例 5.4 的子程序 A

非终结符 A 的子程序，如图 5.6 所示。先判断当前符是否为 c ，如果是就读下一个字符，然后判断 d 。如果不是 d 就报错，如果是 d 就接着读，接着就是出口。子程序 A 入口处当前符如果不是 c ，就会直接从 A 的子程序出去，然后在 S 的子程序里接着判断是否是 b 。

5.2.3 递归子程序法适用范围

递归子程序是不是适用于所有的语法分析问题呢？我们分析两种情况。

1. 如果两个具体有相同左部产生式的第一个字符相同，例如 $S \rightarrow bA|bB$ ，两个产生式首符号均是 b ，如果当前符是 b ，此种情况递归子程序无法判断应该选择哪一个产生式。可能的解决方案有：看下一个字符，来判断选择合适的产生式。但是在递归子程序法中，不能往后看一个字符，它只有通过当前符来进行判断。

因此，同一个非终结符的不同产生式，如果它们的首符号相同，则无法设计相应的递归子程序。

2. 如果文法中有这样一条产生式规则 $A \rightarrow Ab$ ，进了 A 子程序之后就调用 A ，无限调用永远不停止，陷入死循环状态。从这里我们可以看出，如果产生式存在直接左递归，就会进入死循环，因此运用递归子程序法的文法中，不允许直接左递归产生式存在。

综上，给定一个文法，如果不存在直接左递归，并且具有相同左部的各产生式，首符号不同，则该文法就可以采用递归子程序的方法进行语法分析。

例 5.5 给定文法 $G(E)$ ，试构造该文法的递归子程序。

$G(E)$

$$\begin{aligned} E &\rightarrow T | E\omega_0T \\ T &\rightarrow F | TE\omega_1F \\ F &\rightarrow i | (E) \end{aligned}$$

其中产生式 $E \rightarrow E+T$ 和 $T \rightarrow T * F$ 中存在直接左递归，不能直接运用递归子程序法，需要先消除直接左递归。进行消除时，必须保证变换前后是等价的。我们通过文法变换消除左递归，结果如文法 $G'(E)$ 所示。

$G'(E)$

$$\begin{aligned} E &\rightarrow T \omega_0 T \\ T &\rightarrow F \omega_1 F \\ F &\rightarrow i(E) \end{aligned}$$

其中 “ ω ” 和 “ ω ” 是一种特殊的符号，并不是终结符，它表示括号里的内容是可选的，可以是 0 个，可以是 1 个或多个。

转换成上述形式后，不存在直接左递归，下面利用递归子程序法进行语法分析。首先构造主程序。增加一个新的产生式 $Z \rightarrow S$ ，把这个新产生式构造的子程序作为主程序。主程序同图 5.6。

非终结符 E 的子程序如图 5.8 所示。从入口进入后，先调用 T 的子程序，因为后边里的内容是可选的，所以从 T 的子程序返回后，先判断当前符是否为 ω_0 ，如果匹配成功，就读入下一字符，调用 T 的子程序，因为里的内容可能有多个，所以从 T 的子程序返回后，回到判断当前符是否为 ω_0 ，当前符不是 ω_0 时，就认为后面不属于该产生式，到达出口。我们发现当前符不

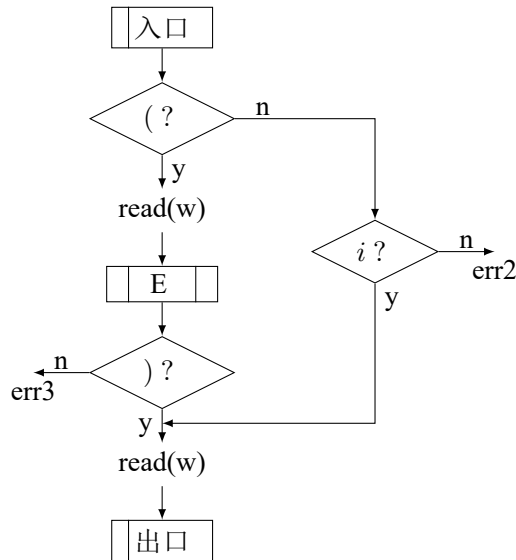


图 5.9: 例 5.5 的子程序 F

是 ω_0 时, 可能不是正常结束, 而是出现错误, 但在递归子程序法中, 在出口前的位置, 无法判断是否出错, 而是让后面的程序来判断是否出错, 这是一种滞后报错。

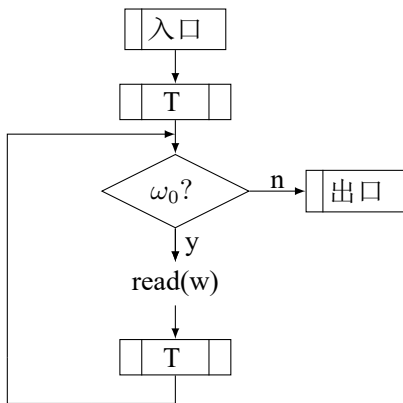


图 5.7: 例 5.5 的子程序 E

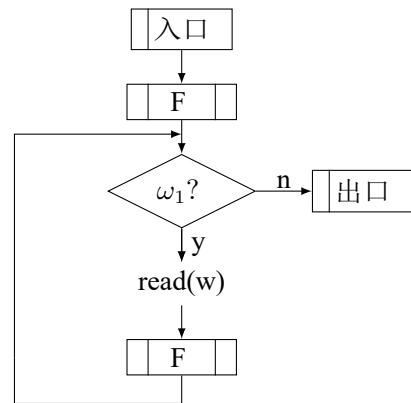


图 5.8: 例 5.5 的子程序 T

非终结符 T 和 F 的子程序, 分别如图 5.9 和 5.10 所示, 具体构造过程, 此处不再赘述。

递归子程序法在推导过程中, 面对同一个非终结符多个产生式, 无需回溯, 根据当前状态便可自动准确判断应用哪条产生式, 其关键在于对文法进行了约束, 即同一个非终结符的多个产生式的首符号不相同, 且文法不存在直接左递归。递归子程序法简单有效, 但它对文法的约束较强, 对大量文法的语法分析难以有效。另外, 由于递归子程序法其递归性的存在, 不断的递归调用, 导致整个方法的效率下降。因此, 虽然递归子程序法的思路清晰, 实现简单, 在实际应用中, 也往往转化为非递归的方法, 从而提高整体效率。

5.3 LL(1) 文法定义

上一节中我们提到过, 语法分析方法通常分成两大类, 分别是自顶向下法 (推导法) 和自底向上法 (归约法)。递归子程序是其中的一种自顶向下的方法。

下面要介绍的是第二种自上而下的方法，也是一种基于推导的方法，叫 LL(1) 分析法。为什么叫 LL(1) 分析法？这两个 ‘L’ 都是有意义的：第一个 ‘L’ 指的是自左向右扫描，即算法（程序）对字符串进行分析的时候是自左向右扫描的；第二个 ‘L’ 指的是最左推导，即算法要求推导最左边可推导的非终结符。括号里的 “1” 指的是在推导过程中，只查看当前的一个符号。为了简单的理解上述内容，我们给出一个例子。

例 5.6 如有以下文法

文法定义

$$\begin{aligned} S &\rightarrow \text{主谓宾} \\ \text{主} &\rightarrow \text{我} \mid \text{你} \mid \text{它} \mid \text{小牛} \\ \text{谓} &\rightarrow \text{研究} \mid \text{学习} \\ \text{宾} &\rightarrow \text{主} \mid \text{自然语言技术} \end{aligned}$$

现在需要检查目标语句“小牛研究自然语言技术”是否归属于这个文法，我们从开始符号 S 出发得到分析短语“主谓宾”。对目标短语从左往右逐个扫描的分析策略是第一个 ‘L’，对分析短语自左向右推导就是第二个 ‘L’ (具体的说，就是当分析短语存在多个非终结符的时候，我们总是从最左侧的非终结符开始推导)。

1. $S \rightarrow \text{主谓宾}$
2. 从目标语句中读入“小”字，确定主 \rightarrow 小牛
3. 从目标语句中读入“牛”字，匹配
4. 从目标语句中读入“研”字，确定谓 \rightarrow 研究
5. 从目标语句中读入“究”字，匹配
6. 从目标语句中读入“自”字，确定宾 \rightarrow 自然语言技术
.... 接下来就全是匹配过程

在上面的分析流程中，我们确定推导式子，只需要读入一个字符即可，这就是 LL(1) 的 1 的含义。既然有 LL(1)，那么也可以存在让算法观察更多字符来决定选用什么式子进行推导的方法，也就是 LL(2)、LL(3) 等等。观察更多的字符，对相同左部的产生式右部有相同的前缀会有更多的容忍（允许产生式右部有相同的前缀），比如例5.6的‘主’如果还能推导出“小刚”，那我们需要再多看一个字符才能决定主语要推出什么右部。

LL(1) 分析法还有一种别的称呼，叫预测分析法。“预测”指的是根据当前的状态，能够知道接下来使用的是哪条产生式。刚才已经提到，LL(1) 分析法与递归子程序一样，都是自上而下的，而且是确定性文法的分析方法。对于 LL(1) 分析法，有三个基本的要点：

※ 三个基本要点

1. 利用一个分析表，登记如何选择产生式的知识；
2. 利用一个分析栈，记录分析过程；
3. 此分析法要求文法必须是 LL(1) 文法。

更详细的补充说明上述三点：

1. LL(1) 分析法必须要有一个分析表，这和递归子程序不一样。分析表的用途是什么？分析表登记了一些如何选择产生式的知识，即在某一个时刻状态下，应该用哪个产生式。
2. LL(1) 分析法里面有一个分析栈，是在执行分析方法的时候用的，目的是记录算法分析的过程，就是要把分析的状态，通过一个栈的形式记录下来。
3. LL(1) 分析法的前置条件是，其所分析的文法必须是 LL(1) 文法。回顾一下 LL(1) 文法的两个条件：第一，左部相同的产生式，其右部产生式的首字母不能相同；第二，不能有左递归。.... 接下来就全是匹配过程

5.4 LL(1) 分析法的完整流程

例 5.7 一个具体例子的演示

先对 LL(1) 文法进行感性的认识。首先，把产生式进行编号。

G(Z):

$Z \rightarrow dAZ$ ① | bAc ②
 $A \rightarrow aA$ ③ | ε ④

四个产生式都进行了编号，其中 ①、② 两个产生式的左部都是 Z，③、④ 两个产生式的左部都是 A。对于这些产生式，可以得到一个分析表。这个分析表，每一行都对应了一个非终结符，每一列都对应了一个终结符，以及结束的标志 ‘#’。

	a	b	c	d	#
Z		②		①	
A	③	④	④	④	

这样的—个表称为分析表，通过一个终结符和一个非终结符，我们就能查到一个表项。表项表示什么？比如，②表示的意思是，当非终结符 Z，看到了终结符 ‘b’，就要使用产生式②进行推导。其它项以此类推。

有了分析表之后，就可以用 LL(1) 分析法进行分析。例如对符号串 $\alpha = "abc\#"$ 进行分析。分析时需要有一个栈结构，记录当前的符号是什么，剩余的符号是什么，操作是什么。

对符号串: $\alpha = bac\#$ 的分析过程:

栈	当前符号	剩余序列	栈操作

图 5.10: 对符号串 $\alpha = "abc\#"$ 的分析过程表

首先, 要把 ‘#’ 压栈, 这是规定。之后还要把起始符号 Z 压到栈里, 因为推导得从起始符号推导。此时, 待处理的第一个符号是 ‘b’, 后面的 ‘a’、‘c’、‘#’ 是还没处理的剩余序列。这个时候该如何执行分析方法? 栈顶符号是一个非终结符, 就用非终结符 Z 和当前符号 ‘b’ 到分析表里进行查找, 对应的是产生式②这一项, 因此就要用产生式②进行推导。

*对符号串: $\alpha = bac\#$ 的分析过程:

栈	当前符号	剩余序列	栈操作
#Z	b	a c #	选择 $Z \rightarrow bAc$ ②

查分析表

图 5.11: 根据栈顶元素和当前符号选择操作 2

下一个操作是让 Z 出栈, 然后把 Z 的右部 “bAc” 逆序压栈, 即 ‘c’、‘A’、‘b’ 依次进栈。因为栈是先进后出的结构, 这样 ‘b’ 就能在栈顶, 可以先匹配, ‘A’ 次之, ‘c’ 最后。现在栈顶符号和当前符号都是 ‘b’, 也是终结符, 因此操作就是匹配 ‘b’。如图 5.12 所示。

*对符号串: $\alpha = bac \#$ 的分析过程:

栈	当前符号	剩余序列	栈操作
# Z	b	a c #	选择 $Z \rightarrow bAc$ ②
# c A b	b	a c #	匹配 b

逆序压栈

查分析表

图 5.12: 根据当前符号和栈顶元素进行匹配

匹配 ‘b’ 之后, 就将 ‘b’ 出栈, 栈顶符号变成 ‘A’, 当前符号是 ‘a’, 剩余序列是 ‘c#’。‘A’ 是一个非终结符, 当前符号是 ‘a’, 到分析表里查, 得到的对应项是③, 所以应该用产生式③进行推导。如图 5.13所示。

*对符号串: $\alpha = bac \#$ 的分析过程:

栈	当前符号	剩余序列	栈操作
# Z	b	a c #	选择 $Z \rightarrow bAc$ ②
# c A b	b	a c #	匹配 b
# c A	a	c #	选择 $A \rightarrow aA$ ③

逆序压栈

查分析表

图 5.13: 根据栈顶元素和当前符号选择产生式

‘A’ 推导完后, 就把 ‘A’ 弹出栈, 再把产生式③的右部逆序压栈。这时候栈顶是 ‘a’, 当前符号也是 ‘a’, 于是匹配 ‘a’。如图 5.14所示。



图 5.14: 根据当前符号和栈顶元素进行匹配

匹配完 ‘a’ 后，弹出栈顶符号的同时，当前符号读取下一个，即 ‘c’，剩余序列只剩一个 ‘#’。对于栈顶符号 ‘A’ 和当前符号 ‘c’，查表得④。如图 5.15 所示。



图 5.15: 根据当前符号和栈顶元素选择产生式

这时候选择是空，操作是一样的，把栈顶的非终结符弹出去，右部逆序压栈，空的逆序压栈即没有元素压栈。当前符号是 ‘c’，所以匹配 ‘c’。

匹配了 ‘c’ 之后把栈顶符号弹出去，这时候就剩下 ‘#’，当前符号也处理完了，只剩下 ‘#’，说明匹配结果是正确的。

总结一下以上 LL(1) 分析过程:

1. 首先，给文法规则的产生式进行编号，得到一个分析表，分析表的每一行代表一个非终结符，每一列代表一个终结符或表示结束的 ‘#’。
2. 接下来是对符号串 $\alpha = "bac\#"$ 的分析，先把 ‘#’ 压进栈，然后压入一个起始非终结符，之后就根据栈顶符号和当前符号去查分析表找到要做的操作，弹出栈顶符号并将右部逆序压栈。
3. 如果栈顶符号和当前符号是相同的终结符，就进行匹配，然后把匹配的元素弹出栈，更新要识别的当前符号。以此类推，直到栈顶符号和当前符号都是 ‘#’，即分析正确。



图 5.16: 最终的匹配流程图

这里有两个问题，分别是：

1. 选择推导产生式后，为什么要逆序压栈？
2. 当栈顶为 A, 当前单词为 c 时，为什么选择 $A \rightarrow \epsilon$ ？

这个地方我不太确定，试着给出这两个问题的答案：

1. 我们选择一个产生式是通过分析表，而分析表的形成主要是通过相同左部，分析它右部各子产生式最左端不相同的前缀字符形成的。LL1 文法从左侧开始扫描目标语句，每次读入一个字符，正好可以和选中的右部子产生式的第一个左侧字符匹配，所以可以发现每次选取非 ϵ 的产生式时下一步一定是一个匹配流程。又因为栈是 FILO 结构，所以逆序压栈才能保证产生式的左端在栈顶。
2. 栈顶是非终结符号且当前元素不是其产生式的最左字符，说明这个非终结符处理不了这个情况，只能交给栈中在其下的终结符进行匹配或者非终结符进行生成再匹配。这个问题就好像你想要一杯牛奶结果店里只有咖啡，店员不可能收了你的钱给你一杯咖啡，只会让你去找另一个有牛奶的店买牛奶一个道理。

5.4.1 抽象的流程表示

✱ 设有文法 $G(Z)$, # 栈底标记和结束标记;

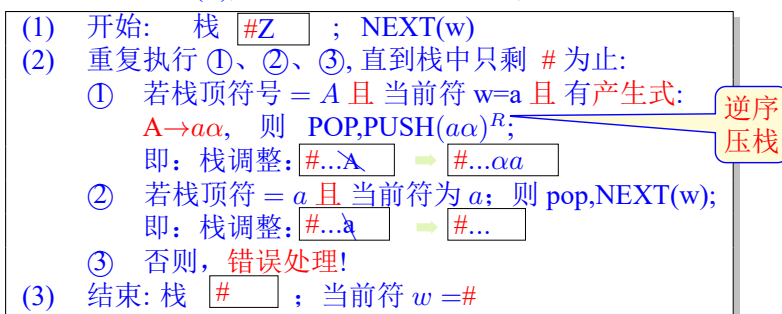


图 5.17: LL1 流程

Algorithm: LL(1) 分析流程 (G, W)

Input: 文法集合 G 待匹配目标字符串 W

Output: 一个状态, 接受字符串或者错误

```

1 先把'#' 和开始符号压入栈中, 从目标字符串左侧中读入一个字符。初始化栈结构 Stack,
   初始化分析表 T。
2 for  $w \in W$  and  $w \neq \#$  do
3   if  $Stack.top()$  is Non-Terminator then
4     generate=lookup(T)
5     Stack.push(generate.reverse())
6     Stack.pop()
7   else if  $Stack.top() == w$  then
8     Stack.pop()
9   else
10    return "Error"
11  end
12 end
13 if  $Stack.pop() \neq \#$  then
14   return "Error"
15 return "Accept"

```

开始如果栈顶是非终结符, 根据当前元素查找分析表选择产生式, 弹出栈顶非终结符, 逆序将产生式右部压栈。如果栈顶是终结符, 且与当前元素相同则弹出栈顶元素即可。否则报错。循环这个过程直到栈顶和当前元素都是'#' 时, 认为可以接受, 否则报错。

5.5 LL(1) 文法及其判定

前面我们说过 LL(1) 分析依赖于特定的文法特征, 大家可以回忆一下前面所提到的三个基本要点的第三点。同时我们还没有给出一个很规范的流程, 怎么得到分析表, 这依赖于我们即

将提出的三个概念。为了更清晰的表达这个概念，我们借助一个具体的文法展开

文法定义

$$G(Z) = (V_N, V_T, Z, P)$$

$A \rightarrow \alpha \in P$, 这里的 α 是一个串，可以是终结符也可以是非终结符组成。

5.5.1 首符号集合、后继符集合与选择符集合

定义 5.2 首符号集合

$$first(\alpha) = \{t | \alpha \xRightarrow{*} t\dots, t \in V_t\}$$

定义在 α 上，这里 α 是一个串，是一个产生式的右部。first 集的定义就是，所有 α 能推导出来的串，这些串的串首的终结符所构成的集合称之为 α 的 first 集，也叫首符号集合。通俗一点讲就是，把 α 能推导出来的第一个终结符的集合，称之为 α 的首符号集合。

定义 5.3 后继符集合

$$follow(A) = \{t | Z \xRightarrow{*} \dots At\dots, t \in V_t\}$$

注意，follow 集和 first 集定义的范围不一样，first 集是在一个串，也就是产生式右部上定义的，follow 集是在左部，也就是非终结符上定义的。A 的 follow 集表示的是 A 这个非终结符，在推导的过程中，后面能跟哪些终结符。

定义 5.4 选择符集合

$$select(A \rightarrow \alpha) = \begin{cases} first(\alpha), \alpha \not\xRightarrow{*} \varepsilon \\ first(\alpha) \cup follow(A), \alpha \xRightarrow{*} \varepsilon \end{cases}$$

要形成分析表，只需要观察选择集。选择集定义的范围是在产生式上，比前面 first 集合在产生式右部、follow 集在产生式左部的范围更大一些。当整个文法面临一个非终结符和待生成的终结符，选择集告诉我们我们应该选择哪一条产生式进行推导。

一个右侧不能推出空的表达式，他的选择集就是他的 first 集合，这个理由非常明显，既然右侧不能推出空，最后一定会形成一个只有非终结符的串，那这个非空串的串首字符显然就是这个非终结符在碰到它时应选择什么产生式的凭证。

一个右侧能推出空的产生式，他的选择集多了一个产生式左部的 follow 集合，这也很好理解，因为当非终结符为空的时候，他不会影响那一个紧挨着他的终结符的推导过程，这也是我们前面所说的去牛奶店买牛奶而不是去咖啡店买牛奶（当然也许咖啡店也卖牛奶）。

注:

1. $\alpha \xRightarrow{*} \varepsilon$ (α 可空), $\alpha \not\xRightarrow{*} \varepsilon$ (α 不可空);
2. 若 $\alpha = \varepsilon$ 则 $\text{first}(\alpha) = \{ \}$ 是空集;
3. 设 # 为输入串的结束符, 则 $\# \in \text{follow}(Z)$;

这三条注解其实是偏定义的东西, 很难说明为什么要这么做。就第一条来说, 能推出空他自己就可以是空, 不能推出空他自己不能是空, 说这么多就是 **禁止无中生有**, 科学也得遵循哲学指导不是。

第二条定义也非常自然, 空了啥玩意都推不出来了, 那还有啥终结符呢。这个式子就是告诉我们 **一分耕耘, 一分收获**, 你今天不播种, 那肯定没有收成。理解成物质守恒也行, 咋好记你就咋记。

唯独需要小心的是第三条, 他很容易被忘记, 你可以认为每个文法都隐含着这样一条产生式 $Z \Rightarrow Z\#$ 。Z 是开始符号, # 是结束符号, 用咱中国人的话说那就是 **有始有终**, 建议同学们在科学学习中贯穿哲学文化思想。

例 5.8 一个分析表的构建实例

经过这么多的定义, 得拿起武器开始干活了。大家也学过编程, 这个像分段函数一样的选择集定义, 在编程上就是个 if-else 的分支结构。所以我们拿到一个产生式的时候, 得先去做个判断, 判断产生式的右侧能不能为空, 我们不需要很严谨的把整个产生式都推导到全是终结符, 一旦出现一个终结符, 就说明右侧不为空了, 因为终结符不能继续推导, 这个就是 **赚来的钱, 交的朋友都可能会离开你**, 他们是非终结符; **学到的知识是终结符, 它永远空不了**。

我们从这个文法中说明怎么形成分析表

G(Z):

$Z \rightarrow dAZ$ ① | bA ②
 $A \rightarrow aA$ ③ | ε ④

①显然不为空, 因为出现了非终结符 d; ②显然不为空, 因为出现了非终结符 b; ③同理不为空; ④为空。

不为空的①②③, select 集合就是 first 集, 分别是 d、b、a。

能为空的④, select 集合是 first 集与 follow 集的并集, 但是因为他不是能为空, 而是本身是空, 根据上面的“物质守恒”原则, 他的 first 集也是空, 他的 follow 集合就是 d 和 b 以及 #, 这个地方可能有点难, 我给你们推荐个经验, 先从别的产生式里面找到这个非终结符, 如果非终结符后头跟的是终结符, 直接塞进 follow 集, 如果跟的是非终结符, 把这个后头的非终结符的 first 集塞进 follow 集合即可。所以 A 这个非终结符, 出现在了①②中。在①中, 其后跟的是非终结符, 把 Z 的 first 集合也就是 {d,b} 加入 A 的 follow 集合。其次是 '#', 这个根据我们前面提到

的“有始有终”原则，所有的开始符号后面都隐含跟着一个'#'号，所以②中可以认为是 $Z \rightarrow Z\# \rightarrow bA\#$ ，于是 A 的 follow 集合就还有 # 号。

到这个地方，select 集其实就是分析表的等价表示，如果你还没想明白怎么从 select 集构建分析表，你需要下去花点功夫，当然也许睡一觉起来就想明白了。这个分析表的结果在上面 LL1 的分析例子中给出来了。

5.5.2 LL(1) 文法及其判定

定义 5.5 LL(1) 文法

LL(1) 文法是指文法中，具有相同左部的产生式，其 select 集合没有交集

这个文法是递归子程序法和 LL(1) 分析法的使用前提。因为一旦相同左部的产生式的 select 集有交集，我们就不知道在面对一个元素的时候需要选择哪一条产生式，也就是出现了歧义。

我们举一个新的例子，

G(Z):

$Z \rightarrow Zb$ ① | a ②

$$\left. \begin{array}{l} \therefore \text{select}(\textcircled{1}) = a \\ \text{select}(\textcircled{2}) = a \end{array} \right\} \text{Selection set intersection}$$

这两个产生式的选择集合相交了，就不属于 LL(1) 文法了。这里也能看到，在递归子程序里，不能有左递归，一旦有左递归，选择集合一定会相交。具有左递归的文法，一定不是 LL(1) 文法！那有左递归，该怎么办呢？我们可以尝试把左递归改写成右递归形式。

再扩展一下，左递归有两种，一种是直接左递归，一种是间接左递归，上面的例子是直接左递归，这两种左递归都需要消除。一个通用的算法，如下

Algorithm 2: 消除左递归算法

```

1 以某种顺序排列非终结符  $A_1, A_2, \dots, A_n$ ;
2 for int  $i = 1; i \leq n; i++$  do
3   | for int  $j = 1; j \leq i-1; j++$  do
4   |   | 将每个形如  $A_i \rightarrow A_j \gamma$  的产生式替换为产生式组  $A_i \rightarrow \xi_1 \gamma \mid \xi_2 \gamma \mid \dots \mid \xi_k \gamma$ ，其中， $A_j \rightarrow a_1$ 
   |   |   |  $a_2 \mid \dots \mid a_k$  是所有的当前  $A_j$  产生式
5   | end
6 end
7 消除关于  $A_i$  产生式中的直接左递归性 }

```

转换为右递归文法后，select 集不相交了

$G'(Z)$:

$$Z \rightarrow aA \text{ ①}$$

$$A \rightarrow bA \text{ ②} \mid \varepsilon \text{ ③}$$

$$\left. \begin{array}{l} \text{select}(\text{②}) = b \\ \text{select}(\text{③}) = \# \end{array} \right\}$$

$\therefore G'(Z)$ 是 LL(1) 文法, 可以使用 LL(1) 分析。

5.6 LL(1) 分析器设计 (实现)

分析器由两部分构成, 第一部分是分析资源, 在 LL(1) 分析里面就是分析表, 第二部分是程序, 程序通过读分析表来完成分析过程。简单的说, 就是构建分析表, 和利用分析表解析待识别串。

LL(1) 分析表 + LL(1) 控制程序

5.6.1 LL(1) 分析表的构造

LL(1) 分析表是存储文法选择集合的知识表。表列是终结符, 行是非终结符, 每个元素是产生式序号。展示一下 C++ 版本的一些粗糙思路供同学们参考。

```

1  struct Element{
2      int number; //产生式序号
3  }
4  //这两个Map需要一段初始化程序, 根据你选择的输入方式决定, 本
    质上就是一段从字符串里分离  $\rightarrow$  左右部分的处理。选择使用map是因为需要一个从char到int的映射, 才方便从
    表里存取表项, 很自然的, 我们还需要维护两个int类型的index
    变量, 来为每一个独立的char进行一个index++的初始化。这段
    地方留给同学们自己想想。
5  Map<char, int> terminator;
6  Map<char, int> Nterminator;
7  Element L[terminator.size()][Nterminator.size()];
8  //数据结构定义到这里了仍然存在一些小问题, 那就是我们该怎么
    记录产生式, 不然我们拿到了序号也没有意义, 当然我们可以不
    存序号, 将element类里改成string类型, 直接存下产生式的右
    部, 还有很多选择, 我就不赘述了, 这个地方同学们后期做编译
    原理课设会有很多机会去实现的。

```

这个地方的代码是我根据 PPT 的意思写的，不太符合工程实现的感觉。你也可以按你喜欢，觉得更工程的方法来完成这个步骤。如果你已经把数据结构的定义和初始化想好了，那我们可以开始下一个问题，也就是分析表的初始化。这个需要我们去求 first 和 follow 集合，还是挺有挑战性的。我是这么想的，如果对编程不熟悉的同学，可以考虑手动输入每个产生式的 first 和 follow 集合；如果编程能力比较强的同学，可以尝试自动从产生式里解析出来 first 和 follow 集合，因为后期课设的产生式非常多非常丰富，如果纯靠脑袋去想会很累，当然最重要的是很不酷。

我给出一点我的想法，希望是启发，不是约束。我尽可能避开具体的代码，只给出注释。

```

1    map<char, set<char> firstSet;
2    void genFirstSet(){
3        //先初始化终结符的firstset，当然是自身。
4        for(){
5            //遍历终结符集合，逐个插入对应的firstSet中
6        }
7
8        //这里是难度重头戏，需要遍历产生式，为非终结符生成←
           firstSet，我建议流程是这样的。
9        // 这里应该有一个循环
10       while(){
11           // 取出产生式的左部，去firstSet里面拿到其对应的←
           firstSet，可能是空，也可能是不完整的firstSet，也←
           可能是完整的。
12           //判断这个产生式的右部是不是终结符或者是空，且是否←
           存在这个左部的firstSet中，不在就加入
13           //否则如果是非终结符，判断右部第一个字符能不能是←
           空，如果能，就把他的firstSet复制过来，继续往后←
           看，这里应有一个循环。
14           //如果右侧第一个非终结符不空，那直接复制firstSet就←
           好
15           //这里还有个小困难，就是如何结束这个循环，我建议是←
           当没有改变时逃出，通过维护一个bool变量，在有改变←
           的分支里不断变换他的值，在循环一开始改动他的值。
16       }
17   }
```

first 集合的自动生成大概如上所示，一个定义的不错的结构，会让你的思路事半功倍，如果做完 first 集合生成，相信 follow 集合也不会难倒你. 加油.

接着是生成 select 集合，这个地方根据上面的定义做就好。在做完这一切，得到分析表之后，你可以看一下

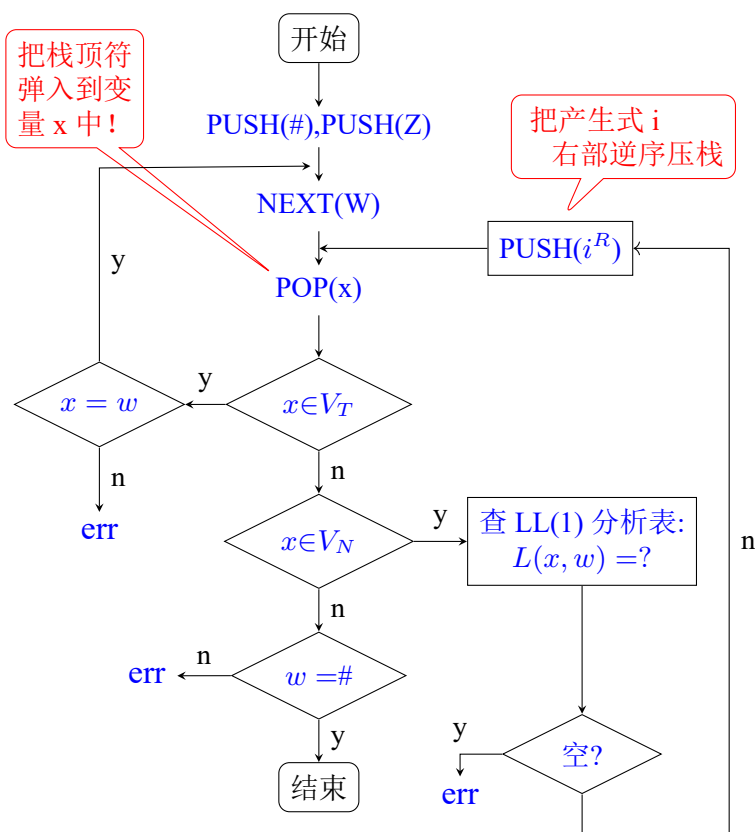


图 5.18: 流程图

我们回头再梳理一遍要做的事情

1. 消除左递归
2. 生成 firstSet
3. 生成 followSet
4. 生成 selectSet
5. 生成分析表
6. 借助控制程序进行分析

5.7 LR() 分析法的介绍

5.7.1 LR() 分析法的“统治地位”

LR 分析是当前最一般的分析方法。因为它对文法的**限制最少**，现今能用上下文无关文法描述的程序设计语言一般均可用 LR 方法进行有效的分析。而且在分析的效率上也不比诸如不带回溯的自顶向下分析、一般的“移进归约”以及算符优先等分析方法逊色。

上下文无关文法的简要回顾

上下文无关文法就是说这个文法中所有的产生式左边只有一个非终结符，很多人可能对上下文这个概念一知半解，我举一个反例。 $aZc \rightarrow abZbb$ ，当我们需要匹配‘Z’的时候，需要确保这个‘Z’的两侧有正确的上文‘a’和下文‘c’，这个就是上下文有关文法。

LR(1) 分析是我们重点关注的内容，与以前的分析方法依赖的文法包含关系如图 5.19 所示。需要说明，LR(0) 和 LL(1) 没有包含关系，但是有交集。

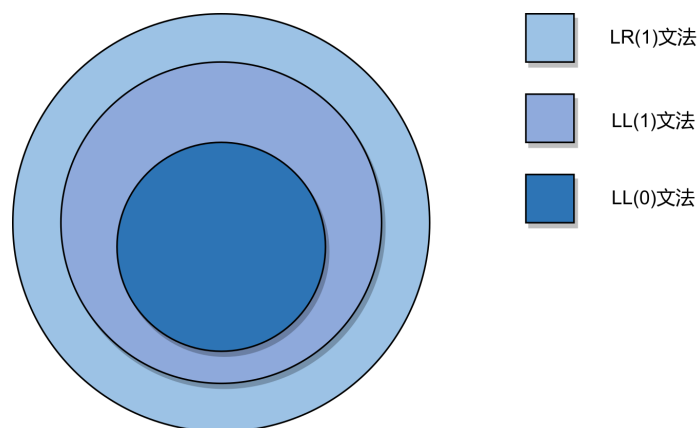


图 5.19: 常见的三种文法的包含关系

一些课外的扩展知识研究中提到一个由 LR(K) 文法产生的语言均可以等价的由某一 LR(1) 文法产生，这也是我们为什么在课程只研究 $k \leq 1$ 的情况，因为对于程序编译基本够用了，感兴趣的同学可以自行查找证明资料。我在编纂这个的过程也找到了一个对 LL 分析和 LR 分析进行对比的 blog¹，也可供同学们做课外读物之用。

5.7.2 LR() 分析法的定义

LR() 分析法是指从左到右扫描、最左归约 (LR) 之意；它属于自底向上分析方法。推导和归约是一对互逆的过程，推导是从非终结符产生零到多个终结符的过程，归约则是从多个终结符逆推回开始符号的过程。相信在上一节 LL(1) 分析法刚刚接触过扫描和推导的读者，应该不会对这个表述很陌生。

在本节学习中，我们需要认识 LR(0)，掌握 LR(1) 分析，括号中的数字是需要看几个当前元素才能决定归约产生式的意思，和 LL(1) 中的‘1’没有什么不同。

为了让同学们更深入的理解最左归约和最左推导这个 LR 和 LL 分析法唯一的不同，我们会给出一个例子。但是在此之前，我需要先介绍两个概念。

¹<https://blog.reverberate.org/2013/07/ll-and-lr-parsing-demystified.html>

略显晦涩的定义

短语: 如果有 $S \xRightarrow{*} \alpha A \delta$ 且 $A \xRightarrow{*} \beta$, 则称 β 是句型 $\alpha A \delta$ 关于非终结符 A 的短语。特别的, 如果 $A \Rightarrow \beta$, 则称 β 是句型 $\alpha A \delta$ 关于非终结符 A 的**直接短语**。

句柄: 一个句型的最左直接短语称为该句型的句柄。

最左归约的任务就是要在扫描特定数量的元素之后, 确定当前句型的**句柄**。

接着到具体的例子上, 展示一个 LR() 分析过程, 注意不是 LR(1)

文法定义

$$G(Z) = (V_N, V_T, Z, P)$$

$$Z \rightarrow aBA\delta, A \rightarrow bc|c, B \rightarrow bB|c$$

符号串 $abccd$ 最左归约过程:

1. 读入 a , 没法归约, 此时存着 a
2. 读入 b , 没法归约, 此时存着 ab
3. 读入 c , 此时存着 abc , 其中的 c 可以被归约成 B , 也就是 $B \leftarrow c$, 归约完后存着 abB 。这里是一个难点, 会有同学思考为什么不是 $A \leftarrow bc$, 从而只存着 aA 。
4. 读入 c , 此时存着 $abBc$, $B \leftarrow bB$, 归约完后存着 aBc
5. 读入 d , 此时存着 $aBcd$, $A \leftarrow c$, 归约完后存着
6. 最后 $Z \leftarrow aBA\delta$, 归约结束

整个流程是

$$1. \underline{a}bc\underline{c}d \rightarrow ab\underline{B}cd \rightarrow aB\underline{c}d \rightarrow aB\underline{A}\delta \rightarrow Z$$

$$2. \underline{a}bc\underline{c}d \rightarrow aA\underline{c}d \rightarrow aABd \text{ 无法进行下去了}$$

我们可以解释一下第二条路径为什么行不通, 上面的短语定义有一个前提, 那就是开始符号能推出这个串, 但是 $aAcd$ 并不能被开始符号 Z 推出。从语法树角度来说, 一个短语应该是从语法树的叶节点开始, 囊括一个完整的子树, 如果从这个角度思考, b 和 c 同为叶节点, 但是他们之间并没有通路, 所以不能算是一个完整的子树, 既然不是短语, 那就不会是句柄了。图 5.20 中展示了按序号顺序构建语法树的过程。

(2) 符号串 $abccd$ 的语法树:

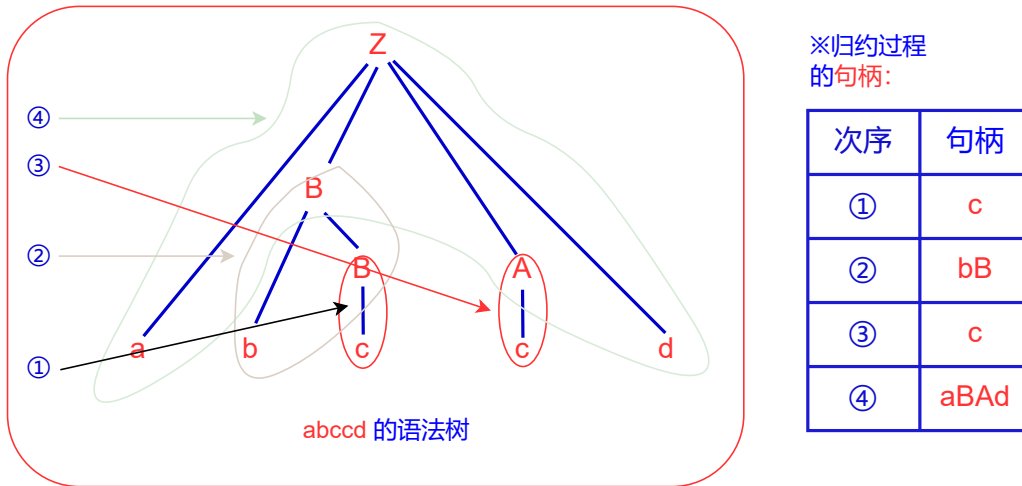


图 5.20: 最左归约形成的语法树

我们用规整的数据结构来表示上面的最左归约过程

(3) 利用分析栈记录分析过程:

设待分析的符号串: $abccd\#$

分析栈	w	剩余串	句柄产生式	操作
#	a	b c c d #		移进, NEXT
# a	b	c c d #		移进, NEXT
# a b	c	c d #		移进, NEXT
# a b <u>c</u>	c	d #	$B \rightarrow c$	归约,
# a b <u>B</u>	c	d #	$B \rightarrow bB$	归约,
# a B	c	d #		移进, NEXT
# a B <u>c</u>	d	#	$A \rightarrow c$	归约,
# a B A	d	#		移进, NEXT
# a <u>B A d</u>	#		$Z \rightarrow aBAd$	归约
# Z	#			OK

图 5.21: 分析栈记录的分析过程

句柄识别器的构造

在具体程序中, 我们需要找到更简单的方法来辨别目前保存的串中是否有句柄。我们通过标注产生式右部的文法符号位置的扩展文法实现 (同时还用一个新的开始符号替换了旧的开始符号)。

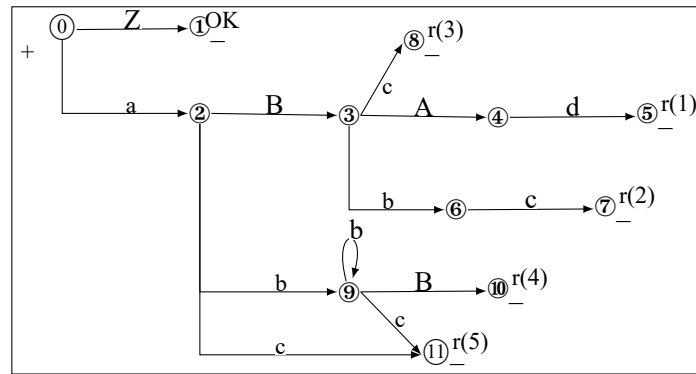


图 5.22: 自动机

文法定义

扩展前 $Z \rightarrow aBAd, A \rightarrow bc|c, B \rightarrow bB|c$

扩展后

$Z' \rightarrow Z_1 \textcircled{0}, Z \rightarrow a_2B_3A_4d_5 \textcircled{1}, A \rightarrow b_6c_7 \textcircled{2}|c_8 \textcircled{3}, B \rightarrow b_9B_{10} \textcircled{4}|c_{11} \textcircled{5}$

这么做的好处在于，同一个 c ， c_8 和 c_{11} 就不会在归约时产生歧义，从不同位置归约就不一样了。我们把他们的位置当做状态，构建自动机来识别句柄。

具体的做法如下，我们假设开始状态都是 0，有了状态，自动机还需要状态转移方程 $\delta(state.token) = next_state$ 才完整。构建完成的自动机如图 5.22 所示。其中移进状态是 0、2、3、4、6、9，在这个状态下需要继续读入字符，才能判断下一步行为。归约状态是 5、7、8、10、11，在这个状态下可以进行归约操作。接受状态是 1，OK 表示归约结束。我们用刚刚识别 $abccd\#$ 的过程举个例子，以 b_9B_{10} 来说，当到达归约状态时弹出 b_9B_{10} ，回到 2 状态，2 状态见到 B 就到了 3 状态，别的就没什么不好理解的地方了。

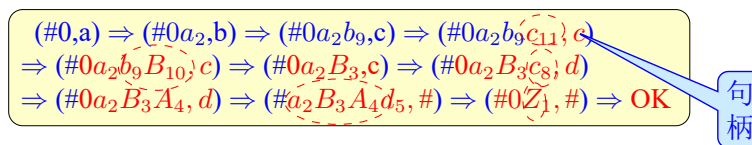


图 5.23: 句柄自动机运用后的识别图

句柄识别器又称“活前缀图”：意思是在最左归约过程中，识别了句柄，实际上也就识别了以句柄为后缀的该句型（规范句型）的前部符号串。

5.8 LR(0) 分析器设计

有了自动机（句柄识别器）后，就能做自底向上的分析，就可以构建一个 LR(0) 分析器。LR(0) 分析器跟 LL(1) 分析器非常像，由两部分构成：第一部分，分析表，即自动机，涵盖了识别句柄也就是活前缀图的信息；第二部分是控制程序，程序要读取分析表，同时根据用户输入的词串进行分析。再强调一下，LR(0) 中的 0，是指不必查看当前符号就可确认句柄的意思。

LR(0) 分析法要求文法应是 LR(0) 文法。

LR(0) 分析表 \oplus LR(0) 控制器

图 5.24: LR(0) 分析器的基本组成

5.8.1 LR(0) 文法及其判定

满足下述特点的文法称为 LR(0) 文法：

1. 句柄识别器中，移进和归约不冲突；即移进和归约不同时发生。
2. 归约时不查看当前符号。

5.8.2 LR(0) 分析表构造

LR(0) 分析表是 LR(0) 分析法的知识表，是句柄识别器的一种机内表示形式：纵轴表示既有终结符和 ‘#’，又有非终结符，因为 LR(0) 分析里状态的转移之间读入的符号，既可能是终结符，也可能是非终结符。横轴表示的是状态编码，代表对状态的编号，0 表示起始。

状态编码	终结符 + #			非终结符	
	a	...	#	Z	...
0					
1	ak		OK	Zk	
⋮					
n	r(j)	...	r(j)		

图 5.25: LR(0) 分析表

接下来要根据句柄识别器，填写表格里的表项。如果当前状态是 i ，读入 x ，到达 k ，即 $\delta(i, x) = k$ 就把 xk 填到相应的表项里， x 既可以是终结符也可以是非终结符， k 表示的是到达的目标状态。如果是归约态，则把这一行全写成归约，不管后面出现什么符号，都要归约。如果在 1 状态看到 “#”，就是 ok。用形式化的语言描述如下：

【算法】

- (1) 扩展文法，构造句柄识别器；
- (2) 根据句柄识别器，填写 LR(0) 分析表：
 - ① 若 $\delta(i, x) = k, x \in (V_N + V_T)$ ，则 $R(i, x) := xk$ ；
 - ② 若状态 i 标记有 $(-, r(j))$ ，则对任何 $a \in (V_T + \#)$ ， $R(i, a) := r(j)$ ；
 - ③ $R(1, \#) := OK$ 。

图 5.26: LR(0) 分析表构建算法描述

在下面这个文法上实操一下

文法定义

$Z' \rightarrow Z_1$ ①, $Z \rightarrow a_2B_3A_4d_5$ ①, $A \rightarrow b_6c_7$ ②| c_8 ③, $B \rightarrow b_9B_{10}$ ④| c_{11} ⑤

他的自动机图5.22我们已经在上面给出过，其分析表按算法填完如下图：

∖	a	b	c	d	#	Z	A	B
0	a2					Z1		
1					OK			
2		b9	c11					B3
3		b6	c8				A4	
4				d5				
5	r(1)	r(1)	r(1)	r(1)	r(1)			
6				c7				
7	r(2)	r(2)	r(2)	r(2)	r(2)			
8	r(3)	r(3)	r(3)	r(3)	r(3)			
9		b9	c11					B10
10	r(4)	r(4)	r(4)	r(4)	r(4)			
11	r(5)	r(5)	r(5)	r(5)	r(5)			

图 5.27: 分析表实例

5.8.3 LR(0) 控制程序设计

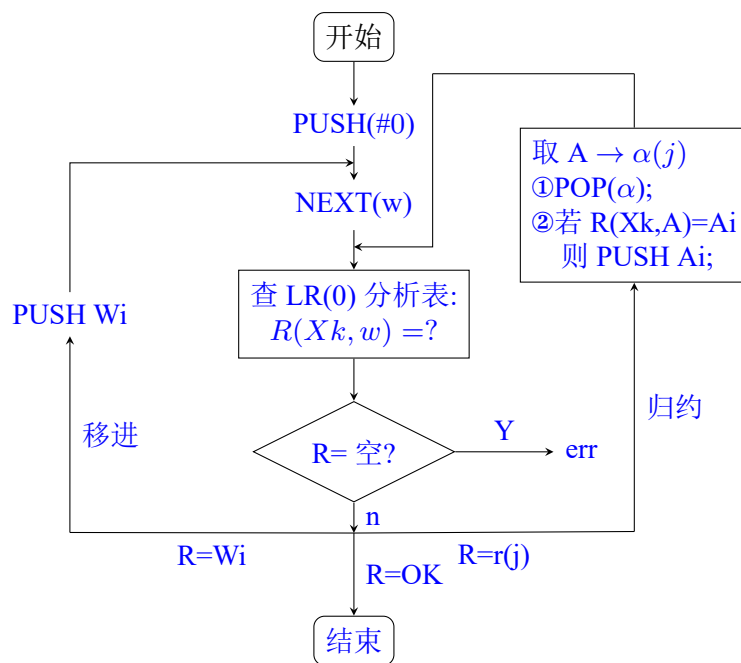


图 5.28: 控制程序流程图

5.9 项目集和可归约前缀图

下面讲另一种句柄识别器的构造方法，即用项目集的方式构造。

5.9.1 扩展文法

假设有一个文法 $G(Z)$ ，要构造它对应的句柄识别器，有以下的步骤：先将原始文法加入位置信息和替换开始符号，形成扩展文法。加入 $Z' \rightarrow Z$ ，引入 Z' ，目的是让程序能够启动。这个技巧在递归子程序也讲过，起始的主程序要读入一个字符，保证进入子程序之前已经读了一个字符。接着对所有产生式右部的所有符号进行编号，包括终结符和非终结符，编号没有特别的要求，可以任意编号，编号只代表状态。

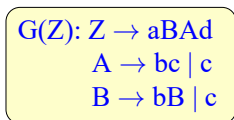


图 5.29: 原始文法

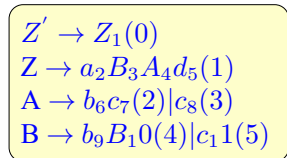


图 5.30: 扩展文法

5.9.2 由扩展文法构造可归约前缀图（句柄识别器）

首先把 $Z' \rightarrow \cdot Z$ 放到项目集里， \cdot 后面是非终结符 Z ，把 Z 的产生式 $Z \rightarrow \cdot aBAd$ 写到下面，形成 I_0 状态。

I_0 状态把 \cdot 后移一位，可以读入 Z 或者 a ，读入 Z 形成 $Z' \rightarrow Z \cdot$ ，也就是 I_1 状态，我们把这个状态称作 OK 状态，代表解析正常结束，是一个结束态。

对于 I_0 状态还可以读入 a ，形成 I_2 状态，接下来希望看到 B 。因此，我们把 B 的产生式附着在 I_2 后面，形成了 I_2 状态。

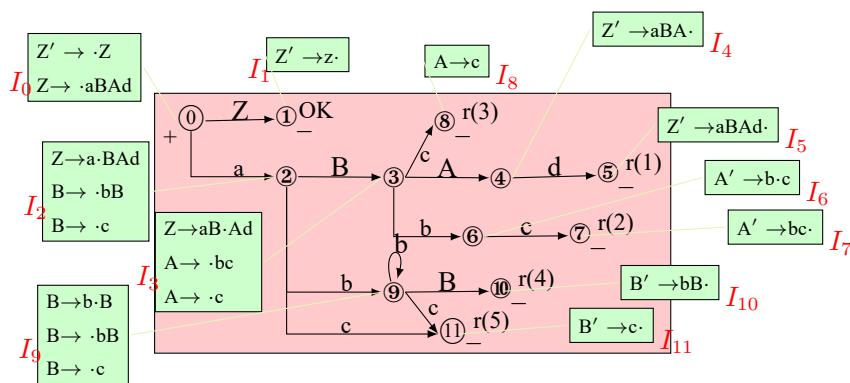


图 5.31: 项目集图

对于 I_2 状态，现在可以读入 B ，形成 I_3 状态，遇见 B 之后希望看到 A ，所以把 A 的表达式也接在 I_3 后面，形成了 I_3 状态。

我们通过四个状态的形成的例子，表达了前缀图的形成原则， \cdot 后紧跟着的非终结符或终结符是我们希望预见的符号，所以如果是非终结符，我们需要把他的产生式加入新的状态中，因为是归约，是从句柄慢慢回到开始符号，句柄在句法树的底端都是终结符。如果是终结符，直接后移预见符号即可。最后形成

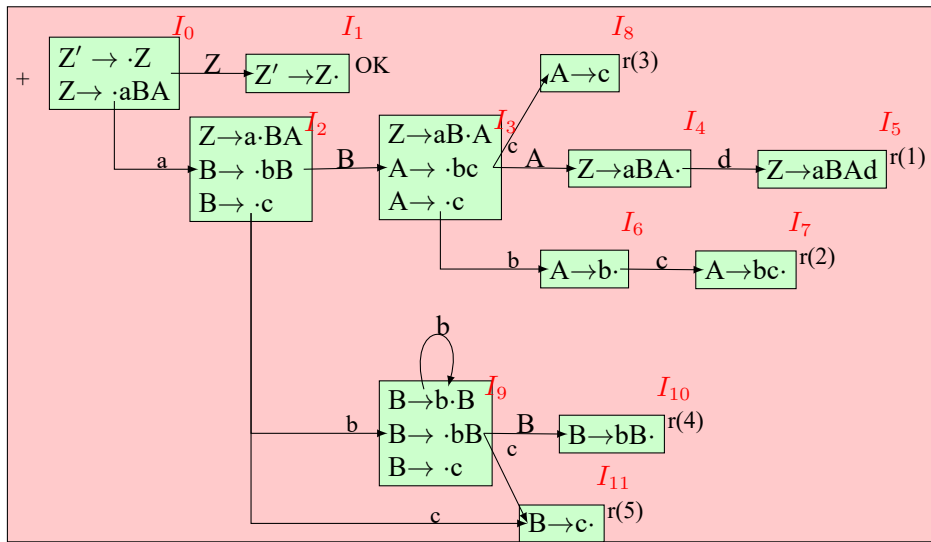


图 5.32: 由前缀图形成的自动机

5.9.3 由可归约前缀图构造 LR(0) 分析表

得到这样一个自动机后，就可以构造 LR(0) 分析表。0 状态看到 Z 到 1 状态，就把“Z1”写到 0 行 Z 列所在的表项。1 状态看到 #，代表结束，即 OK。5 状态稍微麻烦一点，I5 表示对产生式 (1) 进行归约，就把这一行全写成“r(1)”，表示不看当前字符，全部进行归约。其它同理。

	a	b	c	d	#	Z	A	B
0	a2					Z1		
1					OK			
2		b9	c11					B3
3		b6	c8				A4	
4				d5				
5	r(1)	r(1)	r(1)	r(1)	r(1)			
6				c7				
7	r(2)	r(2)	r(2)	r(2)	r(2)			
8	r(3)	r(3)	r(3)	r(3)	r(3)			
9		b9	c11					B10
10	r(4)	r(4)	r(4)	r(4)	r(4)			
11	r(5)	r(5)	r(5)	r(5)	r(5)			

图 5.33: LR(0) 分析表

5.9.4 LR(0) 分析法过程示例

LR(0) 分析是一个移进归约算法，有一个栈，只有两个操作，一是往栈里压入一个符号，二是对栈顶符号进行归约。通过自动机，就能知道用哪个操作。首先，先压入“#0”，表示起始状态，然后读入一个符号到 w，注意 w 是待处理的下一个符号，还未压入栈中。剩余串为“bccd#”。0 状态看到 a，通过查分析表，到 2 状态，做移进操作 PUSH(a2)，a 是待处理的符号，2 是当前状态编号。a 被压入栈了，当前 w 为空，所以读入下一个符号 b，剩余串为“ccd#”。2 状态看到 b，查表，到 9 状态，则 PUSH(b9)，并读入下一个字符 c，剩余串为“cd#”。当前状态是 9 状态，看到 c，到 11 状态，所以 PUSH(c11)。11 状态是一个归约态，要用产生式 (5) B->c 进行

归约，即把栈顶的 c11 归约成 B。c 被归约后，弹出栈，当前栈顶是 9 号状态，看到被归约后的 B，查表得 B10，写到栈顶。现在到了 10 状态，仍然是归约态，用产生式 (4) $B \rightarrow bB$ 归约，即把 “b9B10” 弹出，在 2 状态看到被归约出的 B，查表得 B3，到 3 状态。3 状态看到 c，查表得 c8，做移进操作， $PUSH(c8)$ 。接下来的分析同理，这里不一一赘述。

分析栈	w	剩余串	操作
#0	a	bccd#	PUSH(a2),NEXT(w)
#0 a2	b	ccd#	PUSH(b9),NEXT(w)
#0 a2 b9	c	cd#	PUSH(c11),NEXT(w)
#0 a2 b9 c11	c	d#	REDUCE(5)
#0 a2 b9 B10	c	d#	REDUCE(4)
#0 a2 B3	c	d#	PUSH(c8),NEXT(w)
#0 a2 B3 c8	d	#	REDUCE(3)
#0 a2 B3 A4	d	#	PUSH(d5),NEXT(w)
#0 a2 B3 A4 d5	#		REDUCE(1)
#0 Z1	#		OK

图 5.34: LR(0) 分析过程

5.9.5 LR(0) 分析法实例

做一个小练习，对于黄色框的文法，构造可归约前缀图。

*构造下述文法的 LR(0) 分析表:

$G(Z): Z \rightarrow aAb, A \rightarrow cA|d$

- (1) 扩展文法: $G'(Z')$ $Z' \rightarrow Z_1(0)$
 (2) 构造句柄识别器: $Z \rightarrow a_2A_3b_4(1)$
 (3) 构造可归约前缀图: $A \rightarrow c_5A_6(2)|d_7(3)$

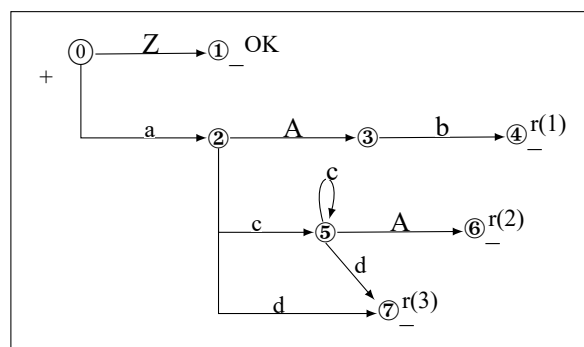


图 5.35: 句柄识别器

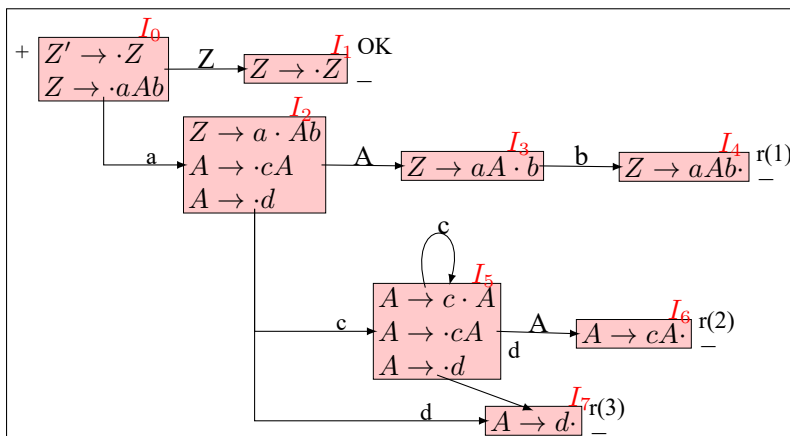


图 5.36: 可归约前缀图

	a	b	c	d	#	Z	A
0	a2					Z1	
1					OK		
2			c5	d7			A3
3		b4					
4	r(1)	r(1)	r(1)	r(1)	r(1)		
5			c5	d7			A6
6	r(2)	r(2)	r(2)	r(2)	r(2)		
7	r(3)	r(3)	r(3)	r(3)	r(3)		

图 5.37: LR(0) 分析表

5.10 LR(0) 分析法的扩展

LR(0) 分析要求文法是 LR(0) 文法，LR(0) 文法有两个最基本的要求。第一，对于任何一个状态，不能产生移进/归约冲突，即不能既有移进操作又有归约操作。也不能有归约/归约冲突，即一个终止状态有两个产生式可以归约，这样在实现系统的时候是无法执行的，因为不知道是归约还是移进。

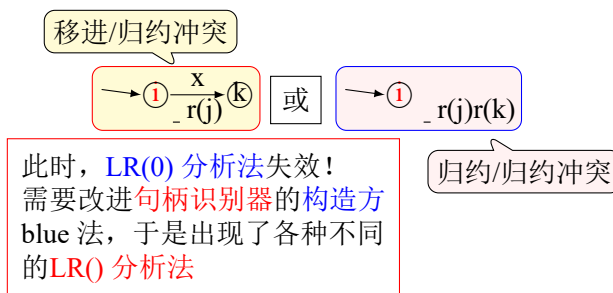


图 5.38: 句柄识别器的冲突

此时，LR(0) 分析法失效，因为产生了歧义，需要改造句柄识别器的构造方法。第一种改造方法，把 LR(0) 分析表进行简单的扩展，称为 SLR(1) 分析法。如何把 LR(0) 简单地扩展一下，变成 SLR(1) 方法呢？从字面意思理解，S 表示简单，没有太复杂的操作，括号里的 1 表示要看待处理字符 w 才知道要做什么操作。

看一个两个非终结符 Z、A 的文法例子

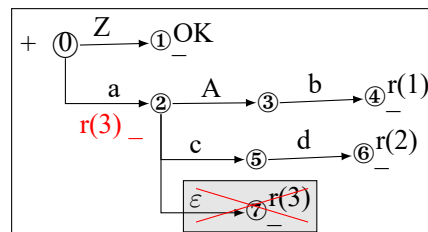
文法定义

$$Z \rightarrow aAb, A \rightarrow cd|\varepsilon,$$

1. 扩展文法，构造句柄识别器

写成如黄色框所示的扩展形式，画成自动机。

$G'(Z') :$	$Z' \rightarrow Z_1(0);$ $Z \rightarrow a_2A_3b_4$ $A \rightarrow c_5d_6(2) \varepsilon_7(3)$
------------	---



这个自动机跟以前的自动机相比，多了一个空 (ε)，2 状态看到空串到 7 状态，是一个归约产生式 (3) 的状态。从 2 状态到 7 状态不需要读符号，进行确定化之后，可以把 7 状态删除，将 7 状态进行归约的操作放到 2 状态执行，即 2 状态是一个终止态，要归约产生式 (3)。这时候就产生了移进/归约冲突，2 状态既可以移进 c，也可以归约产生式 (3)。这样当到 2 状态的时候，就不知道该到 3 状态或 5 状态，还是直接归约产生式 (3)。

$\therefore \textcircled{2}: \{c5, r(3), A3\}$
 \therefore 称为 移进/归约冲突!

图 5.39: 移进/归约冲突

这里的解决方法有两种思路，一是变换文法，但是在 LR(0) 分析里变换文法并不方便，因为变换文法后的自动机并不能直接看出来。二是改变方法，在 2 状态的时候，一个关键的决策是如何知道要归约产生式 (3)? 在 LR(0) 的时候，不需要看待处理的符号 w 再进行归约。反过来说，如果能够看 w，就能知道是不是要进行归约。这里思考一下，w 是什么？是归约完的符号后面跟的符号，比如，w 是 a，栈顶如果能归约成 A 的话，A 后面马上就能跟 a，所以 w 是归约的左部（非终结符）的 follow 集的元素。在 2 状态的时候，如果想用规则 (3) 进行归约，则要求下一个符号是规则 (3) 左部对应的 follow 集，即 b。如果在 2 状态要归约产生式 (3) 到 A，下一个符号只能是 b。2 状态能移进的符号只有 c，到 5 状态。反过来说，b 代表要进行归约操作，c 代表要移进下一个符号。Follow 集跟移进操作的集合不一样。

* 通过查看“当前单词”，是否可以解决？为此：
 求： $follow(A) = b$ 看到： $\{b\} \cap \{c\} = \Phi$ ；
 \therefore 若 $w=c$ 则 $c5$ ；若 $w=b$ 则 $r(3)$ 。

图 5.40: 解决方法

2. 扩展句柄识别器，构造 SLR(1) 分析表

把这个过程用项目集的方式写出来。图中红色的部分 $A \rightarrow \cdot \epsilon$ ，可直接写出归约。对于 I_2 状态，有两种操作，看到 b 则归约，看到 c 则移进。这种方式称为 SLR(1) 文法，在 2 状态需要看 w 才能决定要不要归约，其它状态不需要看。

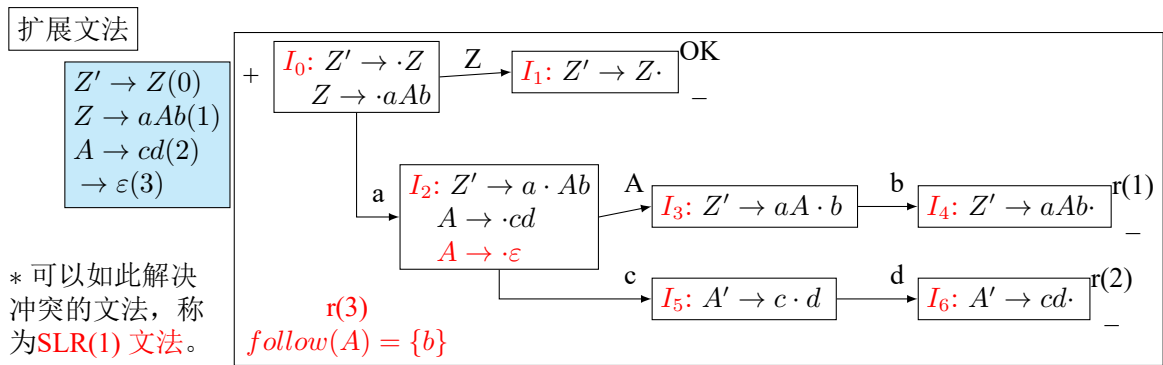


图 5.41: 可归约前缀图

它的分析表比较好写，唯一要注意的是 2 状态，之前的分析表是一整行都要写归约，因为不需要看 w 。在 SLR(1) 里，需要看 w ，当 w 是 b 的时候，归约产生式 (3)，是 c 的时候移进。注意这里与 LR(0) 分析表的区别，当是归约态是，LR(0) 里一整行都要归约，而 SLR(1) 不是。其它部分与 LR(0) 一样。

$$G(S) : S \rightarrow aAd|aec|bAc|bed; A \rightarrow e$$

图 5.42: 分析表

5.11 SLR(1) 分析法的扩展

尽管查看当前单词，SLR(1) 方法仍然对许多文法的 LR(0) 项目集中存在的状态冲突不能解决。比如，产生移进/归约冲突，follow 集里的元素包括移进的元素，这时候就不能说要归约该元素，因为有可能要移进。如果集合带有交集，就要对 SLR(1) 进一步进行扩展。看一个例子。

SLR(1) 分析表

	a	b	c	d	#	Z	A
0	a2					Z1	
1					OK		
2		r(3)	c(5)				A3
3		b4					
4	r(1)	r(1)	r(1)	r(1)	r(1)		
5				d6			
6	r(2)	r(2)	r(2)	r(2)	r(2)		

注意与 LR(0) 分析表的区别！

5.11.1 扩展文法

$G'(S')$:	$S' \rightarrow S(0)$; $S \rightarrow aAd(1) \mid aec(2)$ $S \rightarrow bAc(3) \mid bed(4)$ $A \rightarrow e(5)$
------------	---

5.11.2 构造可归约前缀图

首先是 S' 到 S ，能归约出 S 的产生式有四个：(1)、(2)、(3)、(4)，都写下来。对于 I_0 状态，可以跳转到 3 个状态。第一种情况，可以看到 S ，到 OK 状态，即结束态。第二种情况，看到 a ，希望看到 a 的是 $S \rightarrow \cdot aAd, S \rightarrow \cdot acc$ ，把这两条产生式的 \cdot 往后移一位，分别表示看到 a 后希望看到 Ad ，以及看到 a 后希望看到 ec 。 A 是非终结符，所以把 A 的产生式写下来。对于 I_2 状态，可能看到两种情况， A 或 e 。如果看到 A ，就到 I_3 状态，再看到 d ，则到结束态 I_4 。如果看到 e ，把 $S \rightarrow a \cdot ec, A \rightarrow \cdot e$ 的 \cdot 往后移一位，得到状态 I_5 。 I_5 这两个产生式含义不一样， $S \rightarrow ae \cdot c$ 表示希望看到 c ， $A \rightarrow e \cdot$ 表示要进行归约，归约的是产生式 (5)。再看到 c 就到 I_6 状态。同理，在 I_0 状态看到 b ，也能得到类似的结果。

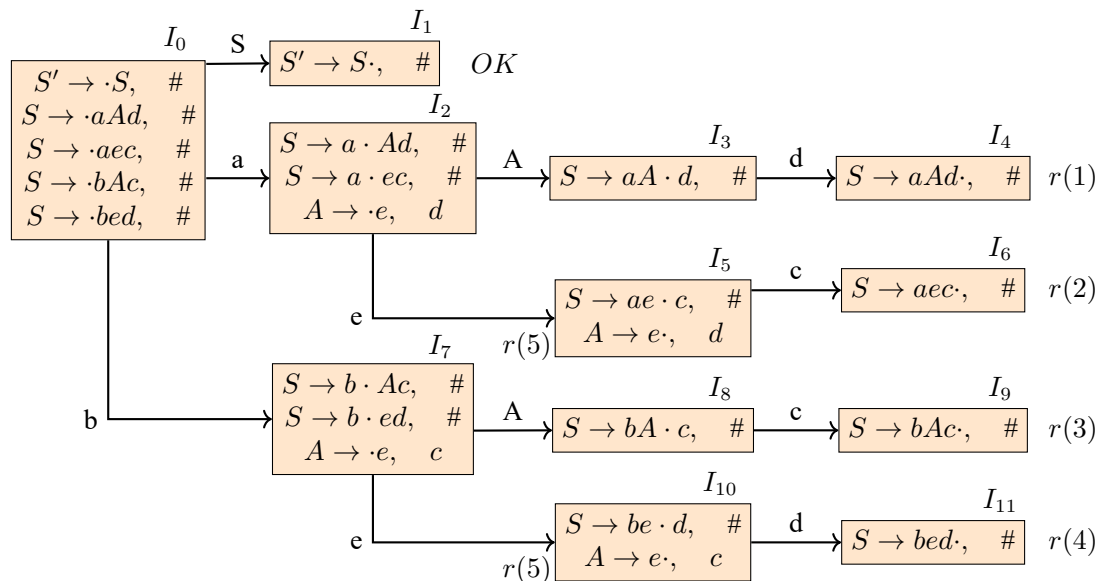


图 5.43: 可归约前缀图

这里出现的问题是，在 I_5 和 I_{10} 状态分别出现了冲突，既有归约又有移进。看归约产生式的左部的 $follow$ 集是否跟移进操作相交，比如这里 $follow(A)=c,d$ ，即 $\{c\} \cap \{c,d\} \neq \emptyset, \{d\} \cap \{c,d\} \neq \emptyset$ ，不能用 $SLR(1)$ ，查看“当前单词”不能解决冲突。

解决思路：对有冲突的状态，仍然是查看‘当前单词’，但要看的更精确一点，以确定动作。(*) 构造 $LR(1)$ 可归约前缀图还是用项目集的方式，但是在产生式后面加一个符号，这个符号表示如果这个产生式被归约了，它后面应该出现什么符号，比如说 I_0 状态第一条产生式 $S' \rightarrow \cdot S$ ，如果 S 被归约成了 S' ，那么 S' 后面应该跟着 $\#$ 。注意这里跟 $follow$ 集不一样。如果 S 被推导完了，即 $S' \rightarrow S \cdot$ ，那么后面也跟着 $\#$ ，如 I_1 状态。在 I_0 状态时，读入 a ，产生式 $S \rightarrow \cdot aAd$

的 \cdot 往后移一位, 得 $S \rightarrow a \cdot Ad$, 后面的符号也是 $\#$ 。产生式 $S \rightarrow aec$ 同理。对于 I_2 项目集, $S \rightarrow a \cdot Ad$ 中的 A 怎么得到? 只有产生式 (5) 能归约出 A 。如果 e 出现了, 归约出 A , A 后面应该跟什么符号? A 后面只有跟 d , 才能做推导, 即 $A \rightarrow \cdot e, d$ 。在 I_2 看到 e 之后, 到 I_5 , 并没有产生移进/归约冲突, 因为 w 如果是 d , 就归约 $A \rightarrow e \cdot$, 如果不是, 就移进 c 。虽然 c 在 A 的 follow 集里, 但不代表在 I_5 状态里看到 c 就要进行归约, 因为进行归约的前置条件是, 在 I_2 状态时 A 后面跟一个 d , 因此, 只有在 I_5 状态里 $A \rightarrow e \cdot$ 这个推导完后, 必须跟 d , 才需要进行归约。

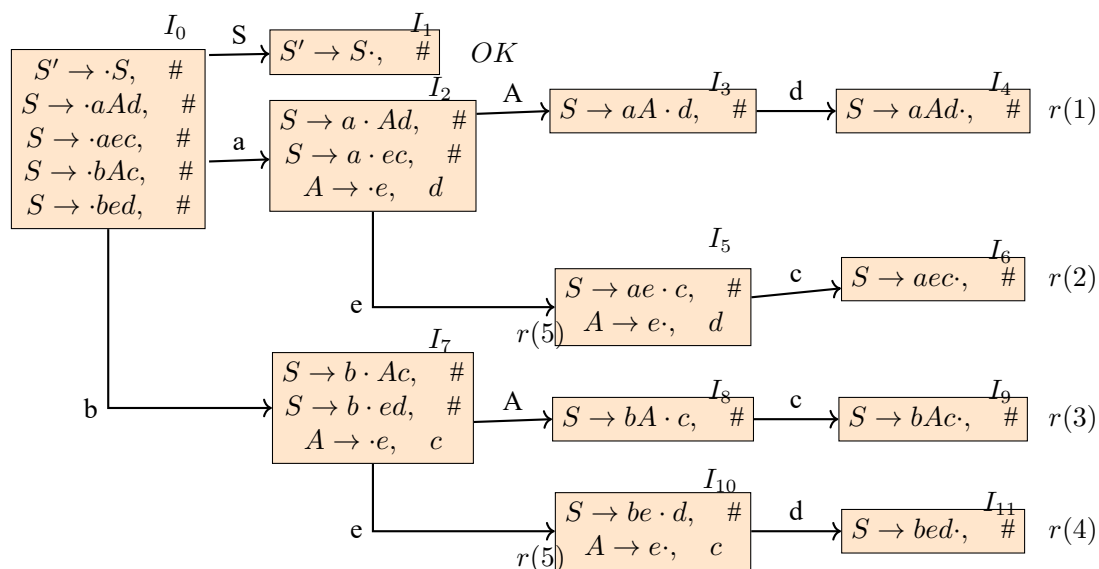
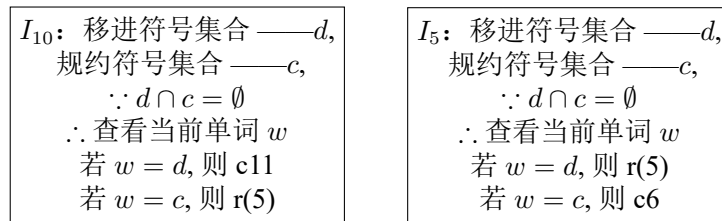


图 5.44: 完整的可归约前缀图

解决办法:

当项目 I_j (例如 I_2) 中同时含有 $A \rightarrow \alpha \cdot B\beta$ 和 $B \rightarrow \cdot Y$ 时, 则向前看一个符号 a , 只有 $a \in First(\beta)$ 时, 才能进行 $B \rightarrow Y$ 归约。

3. 构造 LR(1) 分析表

在每个有可能产生冲突的地方进行标注, 在 LR(1) 里所有归约的地方, 都要用一个符号来记录, 看到 w 的哪个元素进行归约。红色里表达的意思是, 我出现这个符号的时候需要进行归约。比较特殊的是 I_5 和 I_{10} 状态, 分别看到 d 和 c 需要进行归约, 否则就是 I_5 读入 c 到 I_6 , 或 I_{10} 读入 d 到 I_{11} 。# 表示归约的条件, 看到 # 就归约。LR(1) 并没有像 LR(0) 那样, 如果归约一行全是 r , 要多看一个 w 。基于这样一个想法, 就很容易把 LR(1) 分析表写出来, 在 4 状态的时候, 标准的 LR(0) 在一行全都要归约, 但 LR(1) 要看 w , 只有 w 是 # 的时候才需要归约。5 状态比较特殊, 看到 c 的时候到 6 状态, 看到 d 的时候归约产生式 (5)。这里 LR(1) 一行不只有归约, 还有移进, 和 SLR(1) 一样。判断是移进还是归约, 看后面可能跟的元素, 在这个集合里就归约, 不在就移进。

5.12 简单优先分析法基本概念

5.12.1 什么是简单优先分析法

简单优先分析法是一种从左到右扫描、最左归约分析法。简单优先和 LR(0) 分析的策略是一样的，都是从左到右扫描，最左归约。它属于自底向上的分析方法。到现在已经了解了两种自下而上的分析方法，LR(0) 分析和简单优先分析。分析的目的是进行归约，要找句柄。简单优先分析法找句柄的方法与 LR(0) 分析不一样，LR(0) 是用句柄识别器，做一个自动机，进行归约。简单优先利用文法符号之间的优先关系来确定待归约的句柄，来确定当前句型的句柄。简单优先分析法的基本要点有三：

(1) 利用一个分析表，登记选择句柄产生式的知识；

(2) 利用一个分析栈，记录分析过程；

(3) “分析算法”依次读取单词，并进行如下操作：当栈顶出现句柄是，归约之，否则移进。所以简单优先分析也是一个移进归约的方法。

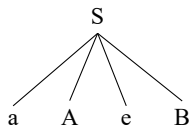
跟 LR(0) 分析不同的是判断句柄出现的方法，LR(0) 分析时根据句柄识别器的状态，简单优先分析要根据分析表里的一些规则或关系。

5.12.2 简单优先分析过程示例

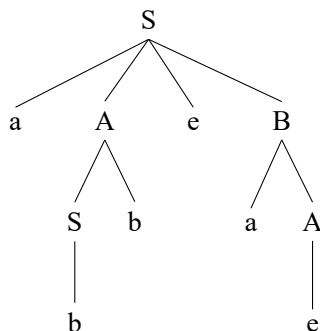
设有文法：

G(S): $S \rightarrow aAeB|b$
 $A \rightarrow Sb|e$
 $B \rightarrow aA$

现分析符号串 $\alpha = abbeae\#$ ，求分析树。首先，从 S 开始，由产生式 $S \rightarrow aAeB$ ，可推导出



这里有两个非终结符，A 和 B，分别进行进一步的推导。A 推导出 S 和 b，S 进一步推导出 b；B 推导推导出 a 和 A，A 再推导出 e。



这棵树对应了文法推导的过程，如果是自下而上就是归约的过程。

利用分析栈记录行分析过程

首先，往栈里放一个 #，当前待处理的符号 w 是 a，栈顶不能归约，所以移进 a，w 读取下一个符号 b。还是没有句柄，继续移进 b，w 读 b，注意此时栈里是第一个 b，w 是第二个 b。此时出现句柄，b 要归约成 S。w 里还是 b，栈顶没有句柄了，移进，读取下一个符号 e。这时又出现句柄 Sb，归约。依次类推，不断进行归约操作，得到最终结果。

设 待分析的符号串:abbeac#

分析栈	w	剩余串	句柄产生式	操作
#	a	b b e a e #		移进, NEXT
# a	b	b e a e #		移进, NEXT
# a b	b	e a e #	S → b	归约
# a S	b	e a e #		移进, NEXT
# a S b	e	a e #	A → S b	归约
# a A	e	a e #		移进, NEXT
# a A e	a	e #		移进, NEXT
# a A e a	e	#		移进, NEXT
# a A e a e	#		A → e	归约
# a A e a A	#		B → a A	归约
# a A e B	#		S → a A e B	归约
# S	#			OK

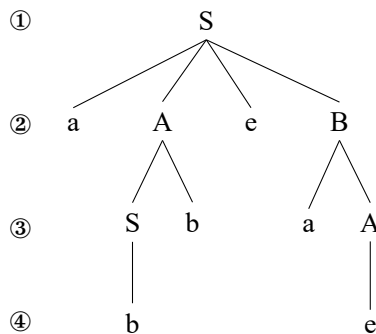
句柄

这个过程跟 LR() 分析的过程很相似，唯一区别在求句柄的地方。何时栈顶出现句柄？怎样求当前句柄产生式？

5.12.3 文法符号之间的优先关系

归约过程中如何确认句柄？

是否是句柄，还要看其所在符号串中的位置。句柄要在产生式的右部，产生式右部与树有什么样的关系呢？首先，树是有层次的，底下的符号要比上面的符号先归约。如下图，第④层要比第③层先归约，第③层比第②层先归约，第②层比第①层先归约。



从语法树上，找出优先关系（指相邻符号之间）如下：

①同时归约者为相等关系，记作 ≡

②左后归约者为小于关系，记作 $< \cdot$ 。优先关系的小于号，左侧晚于右侧归约。

③左先归约者为大于关系，记作 $\cdot >$

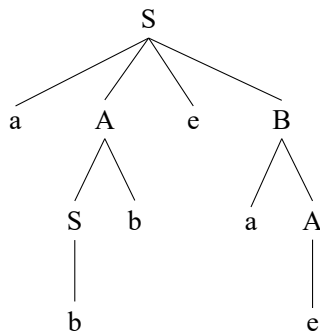
这些符号的左和右的含义并不是代数上的等号、小于号和大于号的含义，左和右分别表示连续出现的两个符号左边那个符号和右边那个符号。对于例子 $b \cdot > b$ ，前一个 b 表示的是连续出现两个符号左边那个符号是 b 的情况，后一个 b 表示连续出现两个符号右边那个是 b 的情况。表示如果连续出现两个 b ，左边的 b 要早于右边的 b 先归约。

假如知道了任意两个符号的大于、等于、小于关系，就能把关系符号填到一个串任意两个符号中间，特殊的一点是这里面最左边和最右边的 $\#$ 是约定，表示字符串的起始和结束。最左边和最右边的 $\#$ 比所有符号都小。

< a < b > b > e < a < e >

图 5.45: 待分析符号串的优先关系

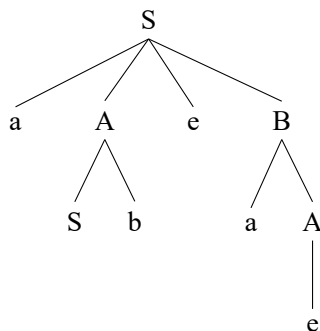
中间的画横线的部分， b 被夹在中间，这里的 b 对应语法树左下角的 b 。 b 与左侧的 a 是 $<$ 的关系，与右侧的 b 是 $>$ 的关系，意思是比左边（右边）先规约，即它应该先规约。



归约完后， b 变成了 S ，即

< a < S = b > e < a < e >

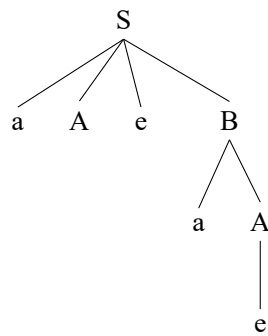
语法树变成：



左下角的 a 与 S 的关系是， S 比 a 要先归约， S 与 b 的关系是 \equiv ，同时归约， b 要先于上面的 e 归约，所以 S 和 b 要被归约，归约成 A ，即

< a = A = e < a < e >

得到的树如图：



此时被归约的是划横线的 e，归约成 A，接着再把 a、A、e、B 归约。

< a = A = e < a = A > #
< a = A = e = B >

总结：一个句型的句柄，位于第一次（自左至右）出现在 < 和 > 之间的符号串，且要求该符号串之间的关系是 \equiv 。

优先关系的定义

文法 G(S):
 $S \rightarrow aAeB|b$
 $A \rightarrow Sb|e$
 $B \rightarrow aA$

从文法中获取优先关系。设 s_i, s_j 是两个文法符号。

①如果产生式右部连续出现的符号是 s_i 和 s_j ，那 s_i 和 s_j 相等，即 $s_i \equiv s_j$ 。

②如果产生式右部推导出符号 s_i ， s_i 后面跟着非终结符 W，W 经过若干次推导，推导出的符号是 s_j ，即 s_j 要被归约成 W，就说 s_j 是早于 s_i 归约的，即 $s_i < \dots_j$ 。比如， $S \rightarrow aAeB|d$ 中，A 推导出的第一个符号要早于 a 归约。

③如果一个非终结符 V 右边跟着符号 s_j ，V 推导出的最后一个符号 s_i ， s_i 要早于 s_j 归约，即 $s_i > s_j$

- ① $S_i = S_j$, 当且仅当有 $U \rightarrow \dots s_i, s_j \dots$;
如: $a=A, A=e, e=B, S=b$;
- ② $S_i < S_j$, 当且仅当有 $U \rightarrow \dots s_i W \dots$, 且 $W \xrightarrow{+} s_j \dots$;
如: $a < e, a < S, a < a, a < b, e < a \dots$
- ③ $S_i > S_j$, 当且仅当有 $U \rightarrow \dots V s_j \dots$, 且 $V \xrightarrow{+} \dots s_i$;
如: $b > b, B > b, A > b, e > b$ 。

图 5.46

头符号集合和尾符号集合

头符号集合指一个非终结符通过推导出现的第一个符号是什么。头符号集合不要求集合里的元素都是终结符，所有的符号都可以推导出。尾符号集合指非终结符推导出的串的最后一个符号是什么。这个定义和 select 集、first 集、follow 集是不一样的。在 LL(1) 分析的时候，select 集要求的是推导出来的终结符，头符号集合和尾符号集合没有这个要求，所有的符号都行。

设 $A \in V_N, s_i, s_j$ 是两个文法符号；则：

$$\begin{aligned} \text{FIRSTVT}(A) &= \{s_i | A \Rightarrow s_i \dots\} \\ \text{LASTVT}(A) &= \{s_j | A \Rightarrow \dots s_j\} \end{aligned}$$

图 5.47: 头符号集合和尾符号集合的形式化表述

例 5.9

文法 $G(S)$: $S \rightarrow aAeB | b, A \rightarrow Sb | e, B \rightarrow aA$

图 5.48: 文法定义

求头符号和尾符号集合

根据 $S \rightarrow aAeB | b$ ，S 的头符号集合是 a, b。B 和 b 属于 S 的尾符号集合，B 是一个非终结符，B 的尾符号集合也属于 S 的尾符号集合。根据 $B \rightarrow aA$ ，B 的尾符号集合是 A。A 也是一个非终结符，A 的尾符号集合也属于 S 的尾符号集合，A 的尾符号集合是 b, e。综上，S 的尾符号集合是 B, b, A, e。

根据 $A \rightarrow Sb | e$ ，S 和 e 属于 A 的头符号集合。S 是非终结符，S 的头符号集合 a, b 属于 A 的头符号集合。A 的尾符号集合是 b, e。

同理，B 的头符号集合是 a，尾符号集合是 A, b, e。

	S	A	B
FIRSTVT	a,b	S,e,a,b	a
LASTVT	B,b,A,e	b,e	A,b,e

优先矩阵

得到了头符号集合与尾符号集合之后，就可以求优先关系。优先关系体现在优先矩阵里，优先矩阵准确描述了任意两个符号之间的优先关系。纵轴表示关系符号左边的符号，横轴表示右边的符号。比如图中 S 所在行，A 所在列的表项，填的是左边出现 S，紧接着右边出现 A，它们两个之间的优先关系。优先关系矩阵的横轴和纵轴是一样的，而且所有终结符和非终结符都要写上。

	S	A	B	a	b	e
S						
A						
B						
a						
b						
e						

图 5.49: 优先矩阵

如何填该表？根据优先关系的定义，产生式连续出现两个符号，就是 \equiv ；如果一个终结符，一个非终结符，非终结符的头符号集合和左边的终结符构成 $<$ 关系，非终结符的尾符号集合和右边的终结符构成 $>$ 关系。

具体看一下。首先看产生式 $S \rightarrow aAeB|b$ ，推出一个符号就不用考虑了，因为至少要两个符号。先看 aA ，是 \equiv 关系，注意这里填的是 a 所在行， A 所在列，表示 a 在左边， A 在右边，而不是 A 所在行， a 所在列。 A 是一个非终结符， a 跟在非终结符 A 后面，那 a 要晚于 A 的头符号集合归约，填上 $<$ 。

接着往后看 Ae ， A 和 e 是 \equiv 关系，往 A 所在行， e 所在列的表项填上 \equiv 。左边是非终结符 A ，右边是终结符 e ， A 的尾符号集合要早于右边出现的 e 归约。 A 的尾符号集合是 B ， e ，所以 B 和 e 要早于 e 归约。千万不要把左右两个符号的顺序弄反了， Ae 的 e 出现在右侧，所以这个 e 要按照纵列去查，而 A 的尾符号集合要按表中左侧的行去查。

这里有个问题，为什么 $e > e$ ？如果左边出现 e ，右边同样出现 e ，表示左边的 e 先归约，这里不是指两个相同符号的代数运算，表达的是优先关系。

同理，其它符号的关系也可以填上。

(2) 优先矩阵

	S	A	B	a	b	e
S					=	
A					>	=
B					>	.
a	<	=		<	<	<
b		.			>	>
e			=	<	>	>

表项 = 空
表示两个
符号不可
能相邻。

图 5.50: 填好的优先矩阵

表中空的地方表示确定不了两个符号之间的优先关系，这个方法用不了。

5.13 简单优先分析器设计

有了优先矩阵表，就可以实现简单优先分析器。

简单优先分析器的基本组成:
 优先矩阵分析表 ⊕ 优先分析控制器
 *分析中只查看当前符号就可确认句柄;
 *优先分析法要求文法应是简单优先文法。

简单优先虽然简单，但有适用的范围。

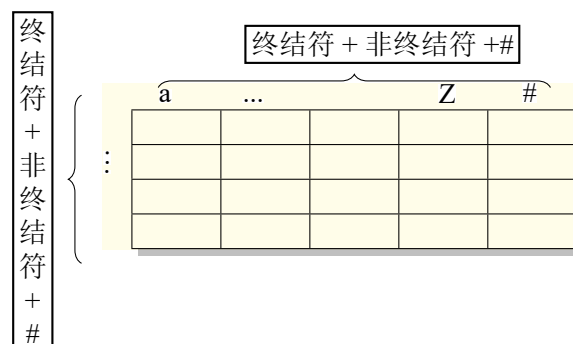
5.13.1 简单优先文法及其判定

满足下述特点的文法称为简单优先文法：①文法产生式没有相同的右部。不能同时有 A 推出 α 以及 B 推出 α 。在归约的时候，看到的是一个赋值串，有一个句柄要归约，但在简单优先分析里不知道要归约成什么。要想知道用什么归约，只能到归约机里查，查到哪个产生式就用哪个产生式，如果查到两个产生式右部是相同的，就可以用两个产生式进行归约，就会不知道用哪一条来归约。

②文法符号之间至多有一种优先关系。

5.13.2 简单优先分析矩阵分析表构造

首先，制作表格如下



算法：①如果产生式右部连续出现两个符号，填 \equiv

②如果产生式右部出现了一个非终结符 W，它和它左边的符号 s_i 就构成了 $<$ 关系，所有 s_i 小于 W 的头符号集合的元素。

③如果产生式右部出现的非终结符 V，V 后面跟着一个符号 s_j ，V 的尾符号集合的元素和 s_j 构成了 $>$ 关系。

- 设 $W, V \in V_N$; s_i, s_j 是两个文法符号;
- ① $S_i = S_j$,
当且仅当有 $U \rightarrow \dots s_i, s_j \dots$;
 - ② $S_i < S_j$,
当且仅当有 $U \rightarrow \dots s_i W \dots$, 且 $s_j \in FIRSTVT(W)$;
 - ③ $S_i > S_j$,
当且仅当有 $U \rightarrow \dots V s_j \dots$, 且 $s_i \in LASTVT(V)$ □

图 5.51: 简单优先分析矩阵分析表构造的形式化描述

5.13.3 简单优先控制程序设计

先把 # 压栈，再读一个符号 w ，然后查表，看当前栈顶符号和 w 的优先关系。当前栈顶符号 X_k 是左边的符号， w 是右边的符号，查表里是否为空，如果为空表示出错，如果不为空就看是否是，表示的意思是 X_k 早于 w 归约，有可能出现句柄。如果不是，就判断是否要结束，即只剩“#S#”，如果还没结束，就接着移进。如果是，就要归约，找离最近的的中间部分，用 $s_i s_{i+1} \dots s_j$ 表示，把 $s_i s_{i+1} \dots s_j$ 弹出，用相应左部的符号替换。这里注意，归约完后，不能移进， w 只是帮助判断栈顶是否出现句柄，还未处理（压栈）。

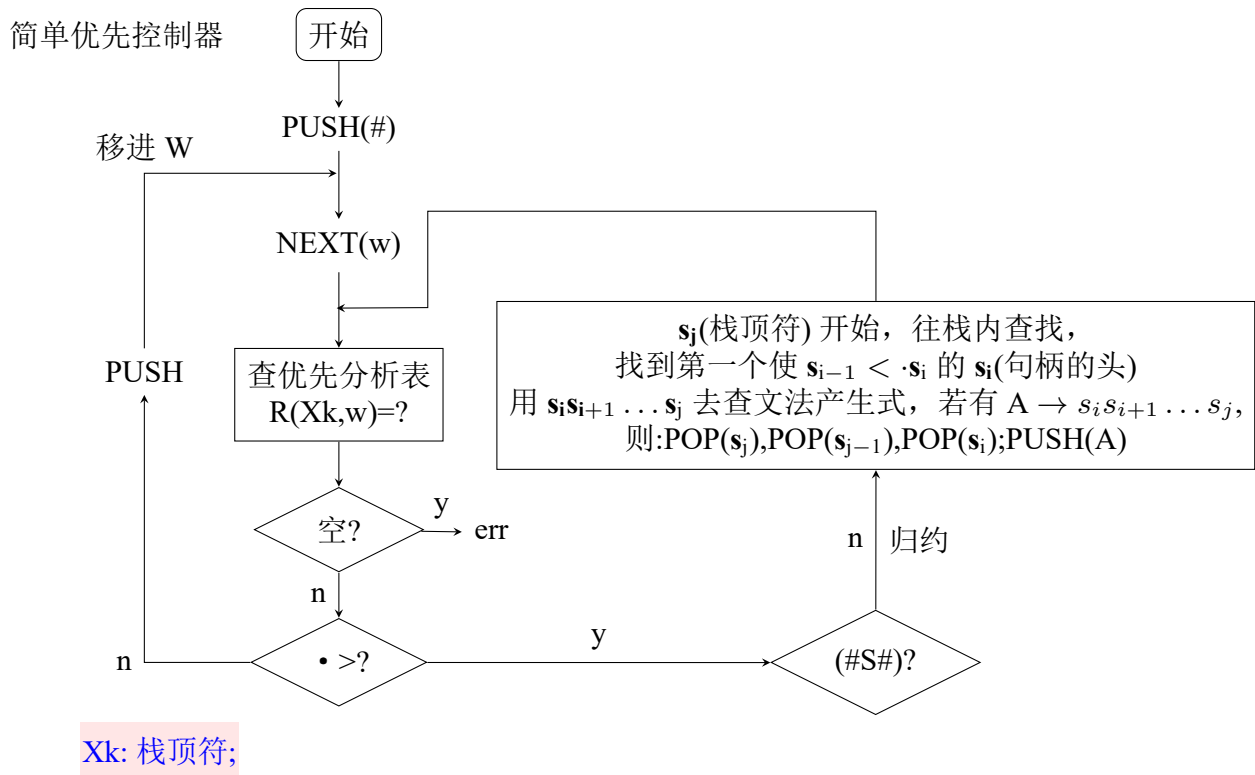


图 5.52: 简单优先控制程序

5.14 算符优先分析

5.14.1 算符文法

算符优先要求文法的右部不能连续出现两个非终结符，比如右部不能出现 A 后面跟着 B ，这样就不能叫算符优先。如果文法里产生式右部没有连续出现的非终结符，就有可能是算符优先文法。

设有一文法 G ，如果 G 中没有形如 $A \rightarrow \dots QR \dots$ 的产生式，其中 Q 和 R 为非终结符，则称该文法为算符违法（OG 文法）。

5.14.2 头符号集合和尾符号集合

如何定义算符优先？我们重新定义一下头符号集合和尾符号集合。头符号集合不只包含 P 推导出来的串首终结符，而且包含推导出来的串首是非终结符的终结符，例如， $A \rightarrow aB$, $A \Rightarrow abB$, $A \Rightarrow Bac$ ，这三个产生式的 a 都属于头符号集合，换句话说，不需要考虑非终结符，来确定头符号集合。尾符号集合同理，不考虑非终结符。

设 $a \in V_T$, $P, R \in V_N$ ，则：

$$\begin{aligned} \text{FIRSTVT}(P) &= \{a \mid P \rightarrow a \dots \square P \rightarrow Ra \dots\}, \\ \text{LASTVT}(P) &= \{a \mid P \rightarrow \dots a \square P \rightarrow \dots aR\}. \end{aligned}$$

图 5.53: 头符号集合和尾符号集合

5.14.3 算符优先关系定义

跟上面一样，两个符号连续出现就是 \equiv 。如果出现了像 aQb 这种情况， a 和 b 的关系也是 \equiv 。注意，在算符优先文法里，不考虑非终结符，可以看成 a 和 b 是连续的。

< 除了要考虑 $p \rightarrow \dots aR \dots$ ，还要考虑头符号集合的定义和尾符号集合的定义。

设 $a \in V_T$, $P, Q, R \in V_N$ ，

$$\begin{aligned} \textcircled{1} a &= b, && \text{当且仅当有 } P \rightarrow \dots ab \dots \text{ 或 } P \rightarrow \dots aQb \dots; \\ \textcircled{2} a &< b, && \text{当且仅当有 } P \rightarrow \dots aR \dots, \text{ 且 } R \xrightarrow{+} b \dots \text{ 或 } R \xrightarrow{+} Qb \dots; \\ \textcircled{3} a &> b, && \text{当且仅当有 } P \rightarrow \dots Rb \dots, \text{ 且 } R \xrightarrow{+} \dots a \text{ 或 } R \xrightarrow{+} \dots aQ; \end{aligned}$$

图 5.54: 算符优先关系定义

5.14.4 算符优先文法

如果算符文法 G 中的任何一对终结符 a 和 b 之间，仅满足上述一种关系，则 G 就是一个算符优先文法 (OPG)。

例 5.10 求文法 $G(E)$ 是简单优先文法吗？先写出头符号集合和尾符号集合，然后把优先矩阵写出来。

文法 $G(E)$: $E \rightarrow T \mid E+T$, $T \rightarrow F \mid T * F$, $F \rightarrow i \mid (E)$

	E	T	F
FIRST	E, T, F, i, (T, F, i, (i, (
LASTVT	T, F, i,)	F, i,)	i,)

图 5.55: 头符号集合和尾符号集合

	E	T	F	+	*	i	()
E				=				=
T				>	=			>
F				>	>			>
+			<			<	<	
*			=			<	<	
i				>	>			>
(<	<	<			<	<	
)	=			>	>			>

文法符号之间优先关系不唯一!

图 5.56: 优先矩阵

简单优先文法有两个条件，不能有相同右部的产生式，任意两个符号之间的关系只能有一种。矩阵有不唯一的关系，所以文法 $G(E)$ 不是简单优先文法。

第6章 符号表

第五章所讲的语法分析在整个编译器里处于一个承上启下的作用。所谓“承上”的“上”，是指前端的词法分析，将源程序中的单词识别并翻译为 Token 序列。语法分析作为过渡阶段，基于 Token 序列，得到句子的语法树结构。所谓“启下”的“下”，则是指语义分析，根据语法的定义进一步生成中间语言，甚至目标代码。

“语义”在语言学界没有统一的概念。从系统开发角度来讲，在计算机编译程序的过程中，语义是指在基于语法分析结果，执行语义动作。比如，假设目标是将程序翻译为目标代码，语义动作就是生成目标代码的动作；假设目标是统计源代码里 Token 的数量，语义就是统计数量。换句话说，语义是由开发者（编译器设计者）来定义的。接下来需要对语义进行表示、访问，其中一个非常重要的手段就是符号表，即第六章的内容。

第六章的内容包括符号表的地位和作用、符号表的组织与管理、符号表的结构设计、符号表的构造过程，以及运行时刻存储分配。

6.1 符号表的地位和作用

6.1.1 符号表的定义

符号表是**标识符的动态语义词典**，属于编译中语义分析的知识库。它的核心目的是组织标识符的查询。其中，符号表面向的对象是标识符，变量、函数名都是标识符。

动态语义词典的定义如下：

- **动态**：指标识符表随着编译过程的执行，会发生变化。（也有一些语言在受限的情况下已经预编译好，可以直接运行，则使用静态标识符表）。

例如：递归调用。递归的深度不能预知，故递归操作本身是动态的，跟递归相关的标识符内容也是动态的。

- **语义**：指描述标识符所有的信息。

例如：标识符可能是整型变量或浮点型变量，整型或浮点型就是标识符的一种语义。

- **词典**：指编译器查询的依据，符号表的目的是让编译器能够快速准确地找到想要查询的标准。

6.1.2 标识符的四种语义信息

1. **名字**：标识符源码，用作查询关键字。即一个符号，能用来指示标识符，用于唯一标识标识符的身份，便于查询。
2. **类型**：该标识符的数据类型及其相关信息。
3. **种类**：该标识符在源程序中的语义角色。
4. **地址**：与值单元相关的一些信息。地址保存了标识符所对应变量的内容，“值单元”就是存储的地方。

6.1.3 符号表的基本功能

- **定义和重定义检查。**

定义：可以修改符号表去定义一个变量，例如定义“int A”，要将 A 填写到符号表里，否则后面不能使用 A。

重定义检查：例如在 C 语言的一个函数里面，不能有两个“int A”，否则会报错“重定义”，报错的原因是在查符号表时发现 A 已经被定义过了。

- **类型匹配校验**

脚本语言分为强数据类型和弱数据类型。C 语言、C++ 等语言是强数据类型，变量类型要先定义好，不能把一个字符串类型赋值给一个整型，编译器在查询符号表的过程中，检查到整型和字符串不匹配，会报错。而 Python 是弱数据类型，即一个变量不需要定义类型，可以任意赋值，把字符串类型赋值给整型也没有问题。

- **数据的越界和溢出检查**

符号表会限制数组的界限，例如生成了一个维度是 10 的静态数组，如果要访问第 11 号元素，编译器将会报错“访问非法”。

- **值单元存储分配信息**

符号表会定义在哪个位置能找到这个元素。

- **函数、过程的参数传递与校验**

举个例子，怎么知道两个函数匹不匹配？调用函数的时候写的方式是否合法？以及类似的问题，都可以通过符号表来解决。

符号表实际上是一个逻辑上的概念，并不是物理上制作一个符号表。它贯穿了编译过程中的很多部分，从某种意义上讲，没有符号表，编译器就缺乏了存储的知识，因为编译程序中标识符的定义全部保存在符号表里。

6.2 符号表的组织与管理

6.2.1 符号表的工作原理

符号表的工作有两种操作，一是写，二是读。符号表存储的是标识符对象语义信息，

- 在遇到声明语句，即定义性标识符时，执行写操作，顺着 Token 指针，将语义信息写入表中，如图6.1。

(i, \rightarrow) → 该标识符符号表项

图 6.1: 符号表操作

可以理解为 (i, \rightarrow) 是一个 Token，前面是语法单元，后面是语义信息。将这个 Token 填到符号表里，从而通过 Token 指针指向的内容，记录标识符的相关语义信息。

- 在遇到操作语句，即应用性标识符时，执行读操作，顺着 Token 指针，读符号表的相应项。例如定义 `int A`，有 `C=A+B`，把 A 加 B 的值赋值给 C。目标是查符号表找到 A 符号所在的位置，用 A 所在的位置协助访问 A 的信息。所以同样要把 Token 的指针指向 A 所对应的符号表这一项。

6.2.2 符号表的查询、访问方式

符号表存储的是所有与用户定义相关的语义信息，在整个编译过程中，对符号表的读写非常频繁，符号表采用的数据结构好坏将直接影响最后编译器的性能。我们学过的数据结构线性表、顺序表、索引表和散列表，都可以采用。具体数据结构的使用不是符号表要介绍的主要内容，本章主要介绍的是符号表承载的语义信息，利用什么样的结构承载这些语义信息。因此本节提出符号表的查询、访问方式，是为了引起大家注意，在具体使用时还需要具体问题具体分析。

6.2.3 符号表的维护、管理方式

不同的程序定义符号表的方式不同，在本书中，符号表可以相对片面的理解为：一个源文件有若干个函数组成，每个函数对应一个符号表，此外还有一个全局的公用符号表。

符号表的组织方式或管理方式一般是针对语言来说的，实际实现时，往往取决于所属语言的程序结构。就 C 语言来说，可以在内存设置一定长度的符号表区，对应内存的一段数据，并建立适当的索引机制，访问相应的符号表，符号表区的形式如图6.2所示：

当前函数的所有符号信息，都保存在“现行函数符号表”。如果这个函数被其他函数调用，它上一层函数的符号信息就保存在“...”里面，以此类推，第一个函数叫 FUNCTION 1，保存在“FUNCTION 1 符号表”。所有符号表的最外层是公用符号表，称为全局符号表。

符号表实际上就是按照这种层次化的方式去组织，把下面部分称为局部符号表区，把上面部分称为全局符号表区。可以采用不同的索引机制去索引，保证能访问到所有符号表。

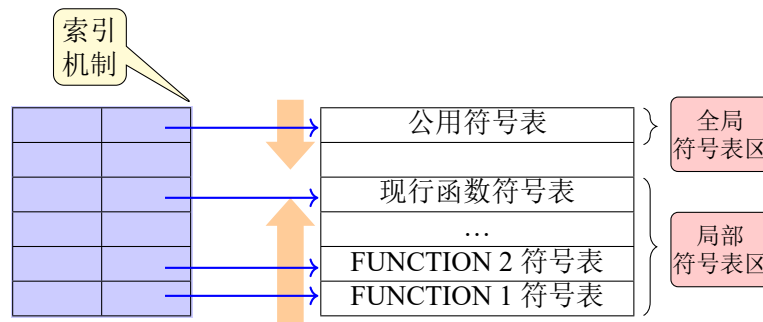


图 6.2: 符号表区形式

6.3 符号表的结构设计

例 6.1 有下列过程函数:

```
FUNCTION exp (x : REAL; VAR y : INTEGER) : REAL;
CONST pai = 3.14;
TYPE arr = ARRAY[1..5, 1..10] OF INTEGER;
VAR a: arr; b, a: real;
BEGIN ...; a[2,5] := 100; b := z + 6; ...END;
```

1. 程序说明:

这是一段简单的 Pascal 语言程序，定义了一个名为 `exp`，返回值为实型的函数，包含两个形参 `x` 和 `y`，`x` 是实数（浮点数）类型，是**赋值形参**，`y` 是整型，是**换名形参**，由关键字 `VAR` 声明。函数的代码段从 `BEGIN` 开始，一直到 `END`。在函数声明和代码段中间的内容，是一系列声明，包括常量标识符 `pai=3.14`，类型标识符定义整型数组 `arr`，两个变量标识符 `a` 和 `b`，`VAR` 是定义变量的关键字，后面的是变量名。

2. 符号表要回答的问题:

- 需要进行符号表的标识符
`exp` (函数，附带信息：类型、参数情况和入口地址...), `pai` (常量), `arr` (类型), `a` (下标变量), `b` (简单变量), ...
- 怎样检查出：`a` 重定义、`z` 无定义以及下标变量？`a[2,5]` 的值地址在何处？ ...

※ 符号表的体系结构设计

由于标识符的种类不同，导致语义属性也不尽相同。下面提供一个符号表的体系结构如图 6.3，观察符号表是怎样组织的：

符号表 (SYNBL) 由词法分析里学到的 `Token`，留有一个指针指向它。符号表有四个属性，可以看成是一个表格。

- 名字 (NAME): 符号表的名字。

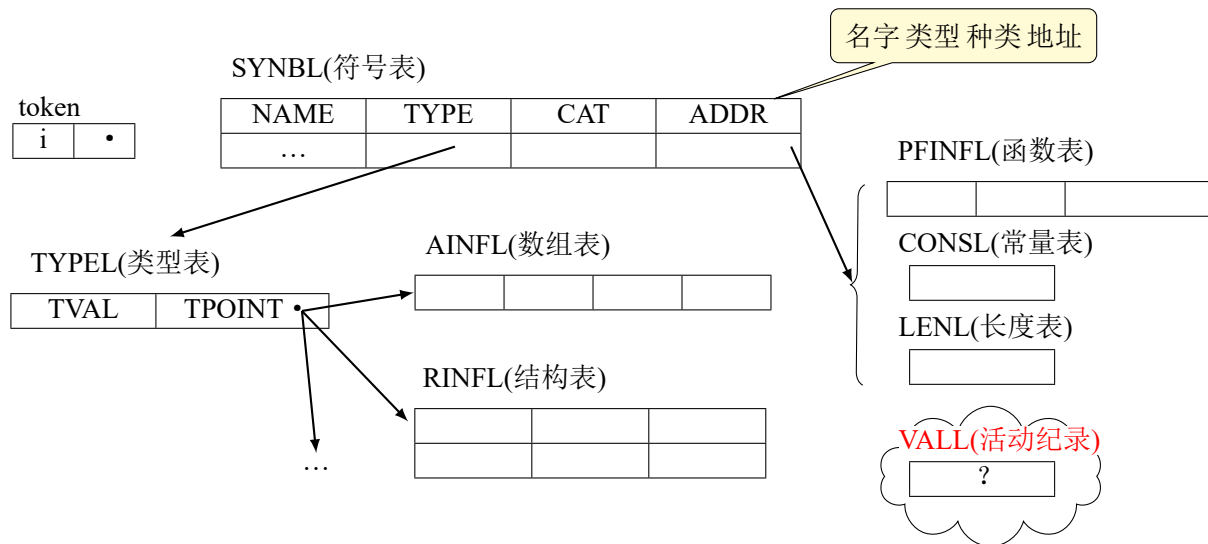


图 6.3: 符号表体系结构

- 类型 (TYPE): 符号的数学类型。

int、float、char 都是数学类型，类型仍然是一个指针，指向一个叫“类型表”的结构。也就是说，符号表是使用一个额外的表来描述类型的。类型表既能表示常见的数据类型，还能记录其它的类型（通过指针指向其他表）。例如“数组表 (ALNFL)”，用来描述数组，在 Pascal 语言里，数组是一个常见的类型。数组大小，上界和下界，数组的每个元素是什么，都可以通过数组表来定义。此外，类型表还可以指向“结构表 (RINFL)”，C 语言里的 `structure` 类型即结构体类型，也可以用类型表来描述。类型表可以指向丰富的类型，所有的数据类型都可以通过类型表来定义。

- 种类 (CAT): 变量的种类，按值传递（变量）或按地址传递（函数名）。
- 地址 (ADDR): 值单元的描述，可简单理解为标识符所存在的物理地址。

地址指向的是内容，例如假设种类是个函数，这时地址指向的是函数表 (PFINFL)，来描述函数的信息；假设定义的是一个常量 `pai`，指向的就是常量表 (CONSL)；还可以指向长度表 (LENL)，记录这个类型多大，占几个字节；最重要的，可能会指向**活动记录 (VALL)**，活动记录和函数的执行是同时进行的，函数执行过程中会生成相应的活动记录。所有的变量，真正保存的地址就是活动记录里关于这个变量描述的内容，换句话说，变量的物理存储保存在活动记录里。

下面分别具体介绍符号表各个部分的内容。

6.3.1 符号表总表 (SYNBL)

总表结构包括四项内容：

NAME	TYPE	CAT	ADDR
------	------	-----	------

图 6.4: 符号表总表 (SYNBL) 结构

- NAME (名字): 标识符源码（或内部码）。

- TYPE (类型): 指针, 指向类型表相应项。
- CAT (种类): 种类编码。
f(函数), c(常量), t(类型), d(域名), v(常规变量), vn(换名形参, 即地址传递, 只需拷贝存储地址), vf(赋值形参, 即值传递, 需要将实参拷贝一份到形参存储位置)。
- ADDR (地址): 指针, 根据标识符的种类不同, 分别指向函数表 PFINFL, 常数表 CONSL, 长度表 LENL, 活动记录 VALL, ...

下面分别具体介绍符号表各个部分的内容。

6.3.2 类型表 (TYPEL)

类型表结构包括两项内容:

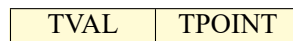


图 6.5: 符号表类型表 (TYPEL) 结构

- TVAL (类码): 表示类型的编码。
 1. 静态数据类型: 包括 i(整型)、r(实数型/浮点型)、c(字符型)、b(布尔型), 以及编译器简单预定义好的其他静态数据类型。
 2. 复杂数据类型, 例如 a(数组型) 或 d(结构体型), 结构体有几个域, 每个域是什么, 是程序员自己写的, 编译器不知道, 只能在看到源程序的时候去分析它。从编译的角度来看, 它是一个不确定的动态结构, 但从执行的角度来看, 它又是确定的 (这里不需要用静态和动态来区分)。
- TPOINT (指针): 根据数据类型不同, 指向不同的信息表项。指针进一步描述数据的类型。
 1. 基本数据类型 (i, r, c, b) —— nul(空指针)。数据类型是预定义好的, 不需要进一步描述, 指针部分不需要指向任何单元。
 2. 数组类型 (a) —— 指向数组表。编译器不知道具体地址, 需要根据用户输入的源程序才知道。如果定义了一个数组, 指针可能要指向的是数组表。
 3. 结构类型 (d) —— 指向结构表。同理数组类型。

6.3.3 数组表 (AINFL)

数组表结构包括四项内容:

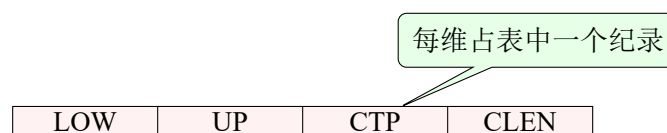


图 6.6: 符号表数组表 (AINFL) 结构

- LOW (数组的下界): C 语言自动设置为 0。
- UP (数组的上界): 用户定义的最大访问范围。
- CTP (成分类型指针): 指针, 指向该维数组成分类型 (在类型表中的信息)。
- CLEN (成分类型长度): 成分类型的数据所占值单元的个数 (假定: 值单元个数依字长为单位计算)。

6.3.4 结构表 (RINFL)

一个结构体会包括若干项, 称之为域, 每个域占表中的一个记录, 指示结构体的每一项都是什么类型。结构表结构包括三项内容:

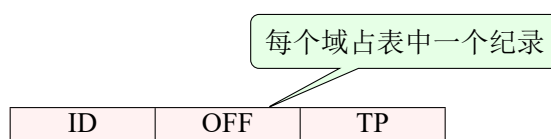


图 6.7: 符号表结构表 (RINFL) 结构

- ID (结构的域名): 每个域的名字。
- OFF (区距): 是 id_k 的值单元首地址相对于所在记录值区头位置。

计算公式如下:

$$\begin{aligned}
 \text{约定: } off_1 &= 0, \\
 off_2 &= off_1 + \text{LEN}(tp_1), \dots \\
 off_n &= off_{n-1} + \text{LEN}(tp_{n-1})
 \end{aligned}$$

公式说明: 第 1 个域的偏移是 0, 即区距是 0。第 2 个域的区域将第 1 个域所对应的区距加上第 1 个域所对应变量的长度。以此类推, 第 n 个域的区域就是域 n-1 的起始地址再加上第 n-1 个域对应变量的长度, 称为第 n 个域的偏移。

- TP (域成分类型指针): 指针, 指向 id_k 域成分类型 (在类型表中的信息)。

思考: 为什么类型要用一个复杂的类型表去表示?

1. 重用: 定义数组类型之后, 这个类型可以定义很多变量, 不需要每定义一次变量就把类型重新定义一遍。
2. 复杂数据类型, 需要对类型本身进行复杂地描述。如果全是预定义类型, 一个编码就足够, 但是复杂的数据类型用简单的类型无法描述, 在设计数据结构时, 就不能统一用一个表来做, 要分出一个数据类型来单独描述。这也是一种编程的常规的习惯, 由于描述的对象很复杂, 不是一个静态的东西, 因此需要一个动态的内容去描述。

6.3.5 函数表 (PFINFL)——过程或函数语义信息

- LEVEL (层次号): 该过函静态层次嵌套号, 用来表示函数的位置 (并非递归的层次)。

LEVEL	OFF	FN	ENTRY	PARAM	...
-------	-----	----	-------	-------	-----

图 6.8: 符号表函数表 (PFINFL) 结构

- **OFF (区距)**: 该过函自身数据区起始单元相对该过函值区区头位置。
- **FN (参数个数)**: 该过函的形式参数的个数 (可以没有)。
- **PARAM (参数表)**: 指针, 指向形参表 (描述每个参数的内容)。形参是函数非常重要的语义信息, 且数量可能较多, 因此构建形参表, 并以指针形式由 **PARAM** 指向形参表。
- **ENTRY (入口地址)**: 该函数目标程序首地址 (运行时填写)。

对 **LEVEL**、**OFF** 进行说明: 函数包括数据对象以及对对象进行的操作, 这两部分内容都需要载入内存, 操作部分不属于本章讨论范围。下面针对数据对象部分, 进一步讨论。

编译器处理一个函数的所有数据对象时, 载入内存一定是连续存储的。设计的函数是静态的, 函数被载入内存一次就产生一次活动, 在这次活动中处理的数据对象所在区域, 被存储在活动记录中。活动记录会存储形参变量、局部变量、临时变量。

以 $x = x + 10$ 为例, 编译器将该语句分解为计算和赋值两步, 开始计算得到 $x + 10$ 的结果, 即为临时变量, 存放中间结果。除变量数据之外, 活动记录还需要记录管理数据, 如函数嵌套调用时的返回地址存储, 断点保存等。管理数据从区头开始存放, 占据一定空间, 变量数据区起始单元相对区头的位置, 记为区距 **OFF**。层次号 **LEVEL** 用于处理变量作用域问题, 以 C 语言程序段为例, 主函数中定义了 x, y , 中间某一段过程中调用的 x 来自该段程序中定义的 x , 调用的 y 来自前面定义的 y , 我们可以通过层次号区分不同作用域的变量。如图所示, 将前面的 $\text{int } x, y$; 所在作用域定为 L 层, 则中间段就属于 L+1 层, 从逻辑上来说, 可以将 x 进行区分, 开始定义的 x 是第 L 层的, 中间段的 x 是 L+1 层的。在 C 语言中, 函数不可嵌套定义, 而在 pascal 语言中, 可以在函数内部定义子函数, 在这种情况下, **LEVEL** 的标识尤为重要。**LEVEL** 表示该过程或函数静态层次嵌套号, 根据写好的程序进行判断, 不随运行时刻而改变。

6.3.6 其他表 (...)

- **常量表 (CONSL)**: 存放相应常量的初值, 仅有一个域。对于字符常量、字符串常量、实型常量、整型常量等这些不同类型的常量, 分别列表。
- **长度表 (LENL)**: 存放相应数据类型所占值单元个数, 仅有一个域。
- **活动记录表 (VALL)**: 一个函数 (或过程) 虚拟的值单元存储分配表; 是函数 (或过程) 所定义的数据, 在运行时刻的内存存储映像。

6.4 符号表的构造过程示例

例 6.2 根据函数构造符号表:

FUNCTION exp (*x* : REAL; VAR *y* : INTEGER) : REAL;

CONST pai = 3.14;

TYPE arr = ARRAY[1..5, 1..10] OF INTEGER;

VAR *a*: arr; *b*, *a*: real;

BEGIN ...; *a*[2,5] := 100; *b* := *z* + 6; ...END;

填表过程如图6.9:

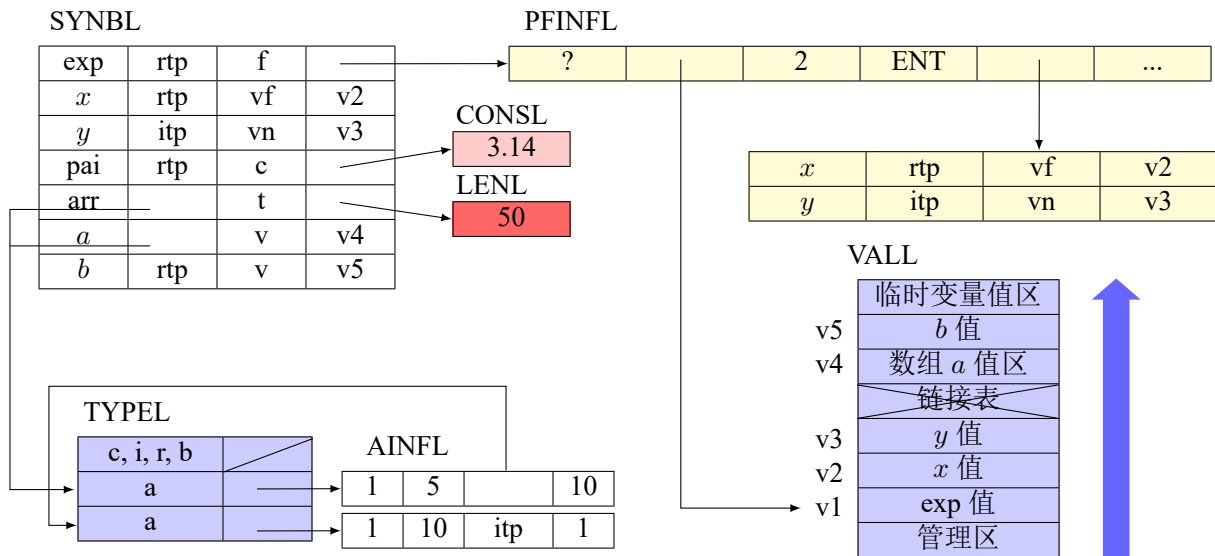


图 6.9: 符号表填写过程

1. exp (函数)

- 填符号表 SYNBL: NAME 域填函数名字 **exp**, TYP 域填函数的返回值类型 **rtp** (实数型), CAT 域填写函数的种类 **f** (函数), ADDR 域指向函数表。

说明: rtp 是指针, 指向 TYPEL 中的 r 项, 此处是为了节省画图空间, 简化为这样。

- 填函数表 PFINFL: 没有完整的程序不知道函数表的层次号, 编译时才知道, 所以暂时不填 LEVEL 域, OFF 域指向 **exp** 返回值所存的内容, 即 v1 的地址, FN 域填函数的变量个数 **2**, ENTRY 域填函数入口地址 **ENT**, PARAM 域指向形参表, 暂时不填。

2. x (变量)

- 填符号表 SYNBL: 函数有两个变量, 在符号表中继续填 **x** 的相关内容, NAME 域填变量名字 **x**, TYP 域填变量类型 **rtp** (实数型, 指向类型表的 r 项), CAT 域填写变量种类 **vf** (赋值形参, 按值传递的参数)。x 值存在活动记录 VALL 中 **exp** 值的下一个位置 (VALL 中的地址从下至上依次增大), x 的起始地址记为 **v2**, 将其填在 ADDR 域内 (实际上 v2 是一个指针, 指向 VALL 中 v2 的内容, 此处为了记录简洁, 没有画出指针)。
- 填形参表 PARAM: 根据 **x** 的名字、类型、种类、地址, 填写 NAME、TYP、CAT、ADDR 四项内容。

思考：为什么在符号表中填完 x ，还要在参数表中再填一遍？两项内容是完全相同的。

3. y (变量)

- 填符号表 SYNBL: 在符号表中继续填 y 的相关内容, NAME 域填变量名字 y , TYP 域填变量类型 itp (整型, 指向类型表的 i 项), CAT 域填写变量种类 vn (换名形参, 按地址传递的参数)。 y 值存在活动记录 VALL 中 x 值的下一个位置, 起始地址记为 $v3$, 将其填在 ADDR 域内。

思考： y 是按地址传递的参数, 那么 VALL 中变量 y 中存的内容是什么? 是 y 所指向的值还是 y 的地址?

答：保存的是 y 的地址, 这里有一个重要的概念——解引用。按地址传递的好处是可以直接修改原始变量的内容; 坏处是多了一次解引用, 解释地址指向的内容, 多了一次跳转, 导致速度变慢。

- 填形参表 PARAM: 根据 y 的名字、类型、种类、地址, 填写 NAME、TYP、CAT、ADDR 四项内容。

4. pai (常量)

- 填符号表 SYNBL: 在符号表中继续填 pai 的相关内容, NAME 域填常量名字 pai, TYP 域填常量类型 rtp (实数型, 指向类型表的 r 项), CAT 域填写常量种类 c 。 ADDR 域指向常量表 CONSL。
- 填常量表 CONSL: 填入 3.14。

5. arr (数组类型)

- 填符号表 SYNBL: 在符号表中继续填 arr 的相关内容, NAME 域填名字 arr, TYP 域指向数组的定义——类型表。
- 填类型表 TYPEL: 没有数组的定义, 要新增。 TVAL 域填类码 a , TPOINT 域指向数组表。
- 填数组表 ALNFL: LOW 域填数组的下界 1 (Pascal 语言从 1 开始), UP 域填数组的上界 5, CTP 域指向类型表, 表示每个单元的类型, 仍然是数组类型 a (嵌套)。由于该数组类型没有被定义过, 所以类型表再次新增一行。 CLEN 域填值单元的长度 10 (填完 ALNFL 之后得到, 根据数组范围可计算整个数组长度为 50)。
- 填类型表 TYPEL: TVAL 域填类码 a (与上一个数组类型的定义不同), TPOINT 域指向数组表。
- 填数组表 ALNFL: LOW 域填数组的下界 1, UP 域填数组的上界 10, CTP 域指向类型表, 此处填 itp (整型, 指向类型表的 i 项), CLEN 域填值单元的长度 1。此时, 反推上一个数组单元的长度为 10。
- 填符号表 SYNBL: CAT 域填写数组种类 t (类型, 可以用作定义其他变量的数据类型)。 ADDR 域指向长度表。

- 填长度表 CONSL: 填入 50。

6. *a* (变量)

填符号表 SYNBL: 在符号表中继续填 *a* 的相关内容, NAME 域填变量名字 *a*, TYP 域指向类型表中 arr 定义的数组类型 a, CAT 域填写变量种类 v。 *a* 值存在活动记录 VALL 中, 起始地址记为 v4, 将其填在 ADDR 域内 (占 50 个单元)。

VALL 中链接表的作用: 静态数据类型直接放在底部, 复杂数据类型编译器不知道内容, 需要指令分析所以放在链接表上面。

7. *b* (变量)

填符号表 SYNBL: 在符号表中继续填 *b* 的相关内容, NAME 域填变量名字 *b*, TYP 域填变量类型 rtp (实数型, 指向类型表的 r 项), CAT 域填写变量种类 v。 *b* 值存在活动记录 VALL 中, 起始地址记为 v5, 将其填在 ADDR 域内。

※ **强调:** 如果种类是类型, 指向的是长度表, 因为要知道这个类型占多少空间; 如果种类是变量, 只需要知道它的物理地址, 指向地址。

例 6.3 根据类型说明填写符号表

```
TYPE arr = ARRAY [1..10] OF ARRAY [1..5] OF INTEGER;
```

设: 实型占 8 个存储单元, 整型占 4 个单元, 布尔型和字符型占 1 个单元。

i, *r*, *c*, *b* 是不同的项, 它们的指针都为空。该例主要介绍二维数组如何存储, 填表过程如图 6.10:

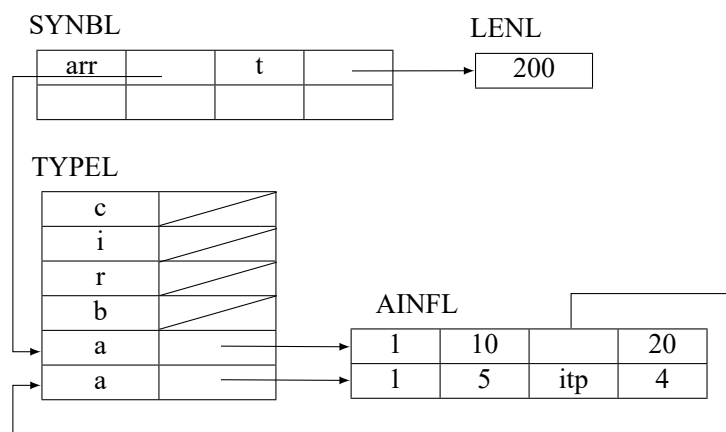


图 6.10: 符号表填写过程

- 填符号表 SYNBL: NAME 域填数组名字 arr, TYP 域指向类型表, arr 是一个二维数组 (数组嵌套数组)。
- 填类型表 TYPEL: TVAL 域填类码 a, TPOINT 域指向数组表。
- 填数组表 ALNFL: LOW 域填数组的下界 1, UP 域填数组的上界 10, CTP 域指向类型表, 仍然是数组类型 a (嵌套), 类型表再次新增一行。CLLEN 域填值单元的长度 20 (填完 ALNFL 之后得到, 根据数组范围可计算整个数组长度为 200)。

- 填类型表 TYPEL: TVAL 域填新的类码 a , TPOINT 域指向数组表。
- 填数组表 ALNFL: LOW 域填数组的下界 1, UP 域填数组的上界 5, CTP 域填 itp (整型, 指向类型表的 i 项), CLEN 域填值单元的长度 4。此时, 反推上一个数组单元的长度为 20。
- 填符号表 SYNBL: CAT 域填写数组种类 t (类型, 可以用作定义其他变量的数据类型)。ADDR 域指向长度表。
- 填长度表 CONSL: 填入 200。

例 6.4 根据类型说明填写符号表

TYPE rec = RECORD

u : INTEGER;

v : ARRAY[1..10] OF BOOLEAN;

r : RECORD x, y : REAL END

END;

设: 实型占 8 个存储单元, 整型占 4 个单元, 布尔型和字符型占 1 个单元。

一个记录可以理解为结构体, 由 3 部分组成: 第一个域 u —— 整型; 第二个域 v —— 10 维的布尔类型数组; 第三个域 r —— 结构体, 由两个域 x 和 y 构成, 都是实数类型。

复杂的结构在符号表里的表示过程如图 6.11:

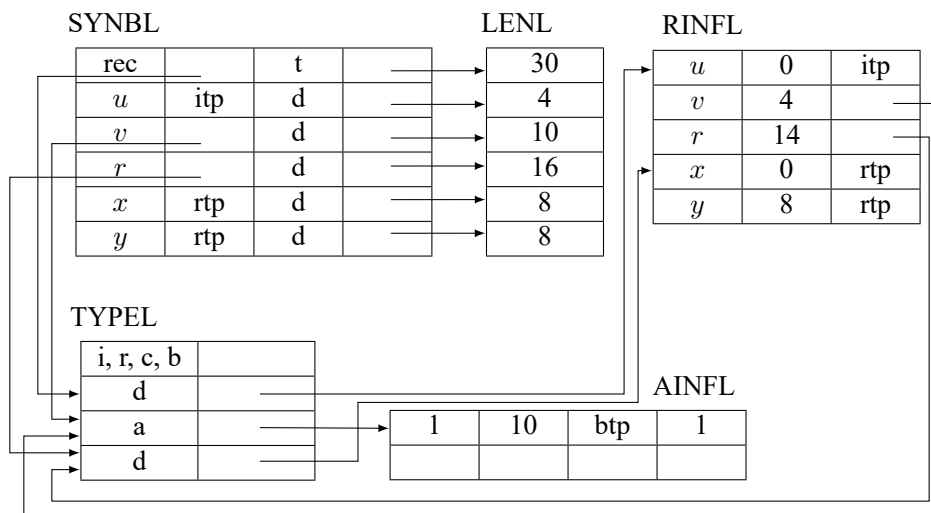


图 6.11: 符号表填写过程

- 填符号表 SYNBL: NAME 域填记录名字 rec , TYP 域指向类型表, 新增一个结构体类型 d (跳转到 TYPEL), CAT 域填写结构体种类 t , ADDR 域指向长度表。
- 填类型表 TYPEL: 若 TVAL 域填类码 d , TPOINT 域指向结构表 (跳转到 RLNFL); 若 TVAL 域填类码 a , TPOINT 域指向数组表 (跳转到 ALNFL)。
- 填结构表 RLNFL:

1. 第一条记录: ID 域填第一个结构体的第一个域名 u , OFF 域填区距 0 (因为是第一个域), TP 域填 itp (整型, 指向类型表的 i 项)。同时在符号表中描述 u , NAME 域填 u , TYP 域填 itp , CAT 域填结构类型 d , ADDR 域指向的长度表填 4。
 2. 第二条记录: ID 域填第一个结构体的第二个域名 v , OFF 域填区距 4 (第一个域的区距 + 第一个域的长度), TP 域指向类型表, 新增一个数组类型 a (跳转到 TYPEL)。同时在符号表中描述 v , NAME 域填 v , TYP 域指向类型表中新添加的数组类型 a , CAT 域填结构类型 d , ADDR 域指向的长度表填 10。
 3. 第三条记录: ID 域填第一个结构体的第三个域名 r (仍是一个结构体), OFF 域填区距 14 (第二个域的区距 + 第二个域的长度), TP 域指向类型表, 新增一个结构类型 d (跳转到 TYPEL)。同时在符号表中描述 r , NAME 域填 r , TYP 域指向类型表中新添加的数组类型 d , CAT 域填结构类型 d , ADDR 域指向长度表。
 4. 第四条记录: ID 域填第二个结构体的第一个域名 x , OFF 域填区距 0 (因为 x 是新纪录的第一个域), TP 域填 rtp (实数型, 指向类型表的 r 项)。同时在符号表中描述 x , NAME 域填 x , TYP 域填 rtp , CAT 域填结构类型 d , ADDR 域指向的长度表填 8。
 5. 第五条记录: ID 域填第二个结构体的第一个域名 y , OFF 域填区距 8 (第一个域的区距 + 第一个域的长度), TP 域填 rtp (实数型, 指向类型表的 r 项)。同时在符号表中描述 y , NAME 域填 y , TYP 域填 rtp , CAT 域填结构类型 d , ADDR 域指向的长度表填 8。
- 填数组表 ALNFL: LOW 域填数组的下界 1, UP 域填数组的上界 10, CTP 域填 btp (布尔型, 指向类型表的 b 项), CLEN 域填值单元的长度 1。此时, 反推数组的长度为 10。
 - 填长度表 CONSL: 完善 r 的长度 = $16(x + y)$, rec 的长度 = $30(u + v + r)$ 。

6.5 运行时刻存储分配

本节介绍标识符标识符变量的地址分配与对它们的访问, 主要讲解类码为 v 的标识符地址分配问题。

6.5.1 标识符值单元分配

一个变量保存的位置, 在 C++ 语言里用 `new` 就可以得到, 但是需要注意, 程序运行是有环境的, 不能单独写一个 `new` 语句来运行, 要放在一段程序里, 具体来说是在一个函数里去执行。这个问题衍生出一个思考, 分配 `new` 的时候, 是在一个什么样的条件下去分配? 在函数内部可以做 `new` 操作, 外部也行, 因为 `main` 函数也是一个函数。

对于值单元的分配, 一般有两种策略。

1. 静态分配:

在编译阶段即可完成真实的地址分配。在编译时对所有数据对象分配固定的存储单元, 且在运行时始终保持不变。

- 优点：程序编译之后、执行之前，就知道每个变量所存的位置和大小。静态分配不需要计算变量的存储位置，可以提高程序执行的效率。
- 缺点：如程序递归调用时，不能做到静态分配。递归的层数由用户输入的变量决定，递归函数每个变量的存储位置不能确定，这种情况下采用动态分配。

2. 动态分配:

在**运行时刻**进行的值单元分配，即动态地决定变量所存储的位置和大小，在编译时只能进行相对地址分配。

- 栈式动态分配：栈常用于先进后出的操作。
- 堆式动态分配：堆常用于维护一个有序的表，排序、top-k 等可以用堆加速操作过程。

注：值单元分配是以过程函数为单位的，每个过程函数有其自身的活动记录。“过程”在 Pascal 语言里可以简单地理解为没有返回值的函数。

6.5.2 活动记录

首先介绍三个基本概念：

1. 过程：一个可执行模块，过程或函数，通常完成特定的功能。一个过程在编译器里指一段有独立功能的程序，可以完成一个函数或者一个 Pascal 过程，也可以称为一个函数。
2. 活动：过函的一次执行。每执行一次过程函数体，则产生该过函的一个活动。
3. 活动记录：一个有结构的连续存储块。用来存储过函一次执行中所需要的的信息。所谓结构，是指定义了管理数据、形参、局部变量等的位置结构，可视作一种数据结构。

三者关系：过程是一个抽象的概念，不对应函数具体的执行；活动是这个抽象概念的一次具体实现过程；活动记录是伴随着活动被定义的概念，一个活动记录用来记录一个活动所需要或产生的信息。

注意：

- 活动记录并不是针对某一个函数，而是针对这个函数的一次执行。更准确地说，是运行时所产生的一个记录，而不运行就没有活动或活动记录，只有过程。
- 活动记录仅是一种存储映像，编译程序所进行的运行时刻存储分配是在符号表中进行的，符号表中分配的变量就存储在活动记录里。
- 如果不支持可变数据结构（如动态数组，需要动态存储），活动记录的体积是可以在编译时确定的。

活动记录结构如图6.12所示：

1. 连接数据区

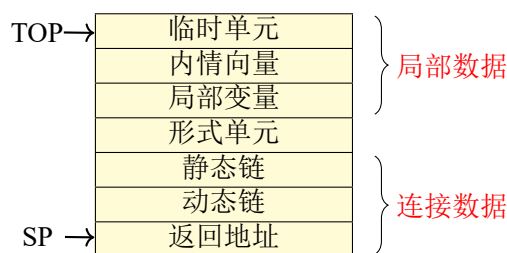


图 6.12: 活动记录结构

- 返回地址：保存断点地址，返回主控程序时继续执行的位置。（断点地址：函数被调用时的返回地址，是汇编语言执行的地址，而不是活动记录的返回地址。）
- 动态链：指向调用该过程的主调程序的活动记录的指针。（直接外层：与层次有关，如果是在第 3 层调用，直接外层就是第 2 层。）
- 静态链：指向静态直接外层活动记录的指针。

2. 形式单元

用来存放实参的值或地址。

3. 局部数据区

用来存放局部变量、内情向量、临时单元。内情向量存放计算过程中的一些相关参数；临时单元存放运算过程中的中间结果（在生成中间代码时生成的非用户定义的临时变量）。

4. 栈指针

- SP：指向现行过程活动记录的起点，即第一个单元。
- TOP：指向（已占用）栈顶单元，即活动记录的最后一个单元。

用 SP 和 TOP 就可以确定活动记录所在的物理存储的区间。

例 6.5 动态链、静态链

动态是指运行过程中，而静态是指编写程序时。

- 动态链与程序调用有关。

M 程序调用 N 子程序的活动记录栈结构如下图，当 M 程序载入内存时，两个指针指向 M 活动记录的起始位置和终止位置，定义了当前可操作的数据区。当 N 子程序载入内存时，N 活动记录进入活动记录栈，这两个指针移动到指向 N 活动记录的起始位置和终止位置，表示当前运行函数为 n。这两个指针限定了当前可操作的数据区，即当前运行函数的数据区。

当 N 运行结束返回 M 时，N 活动记录弹栈，两个指针又重新指向 M 活动记录的起始位置和终止位置。为了获知 M 活动记录的位置，在 n 活动记录中开辟一个域，当 N 活动记录压栈时，M 程序将自身活动记录首地址放入该域中，指针指向 M 活动记录首地址，该指针就是动态链。

- 静态链与程序设计相关。

Pascal 语言允许嵌套定义函数，例如在函数 P 中嵌套定义了函数 M 和函数 N，在函数 M 中又嵌套定义了函数 Q。换言之，Q 子程序可以访问 M 的数据，也可以访问 P 的数据，但是不能访问 N 的数据，而 M 可以调用 N，N 也可以调用 M。此处涉及函数作用域的问题，内层的函数可以访问外层的、嵌套的外层的数据。例如在函数 P 中定义了变量 x ，在函数 Q 中存在 $x = 10;$ 语句，在活动记录中结构如下图所示。函数 P 的活动记录压栈，包含 x 的信息，不断调用，将 Q 压栈，此时动态链指向 Q 活动记录的起始位置和终止位置，但是 x 不在 Q 的活动记录内，无法根据动态链找到 x 的位置。Q 访问 x 需要获知 P 活动记录的首地址，再根据变量 x 的区距找到 x 。函数 Q 保存静态定义的外层 M，再外层 P 等所有外层在内存中最新活动记录的首地址信息，就可以进行上述访问操作，这就是静态链。

此类变量作用域的问题，与静态设计相关，需要通过静态链解决。静态链可以指向静态直接外层活动记录的首地址，也可以通过多步跳转访问各个外层函数的数据。

6.5.3 简单的栈式存储分配

以 C 语言为例：没有分程序结构，过程定义不允许嵌套，但允许过程的递归调用。

例 6.6 C 语言过程调用关系：Main() -> Q() -> R()

1. C 语言程序的存储组织

活动记录栈状态如图 6.13，从下往上，地址由小到大。由于是栈式存储分配，后调用的函数存储在栈顶。当 R() 执行结束后弹栈，再执行 Q()，以此类推。下图中当前运行函数为 R，可操作数据区由指针 SP 和 TOP 进行限定。SP 和 TOP 分别指向当前活动记录的首尾地址，即 R 的第一个单元和最后一个单元。

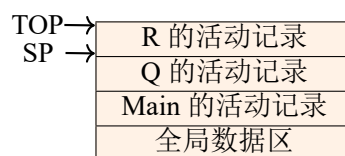


图 6.13: C 语言调用过程 VALL-1

2. C 的活动记录

如图 6.14 所示，将参数个数加入活动记录中，该形式支持被调函数完成实参到形参的传递。Old SP 指向上一个调用函数的过程所对应活动记录的首地址，由动态链完成。C 语言中函数不可嵌套定义，因此活动记录中只有动态链，没有静态链。

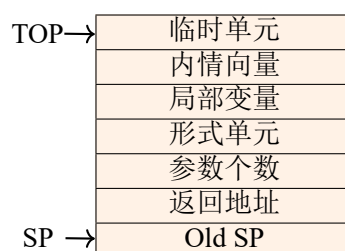


图 6.14: C 语言调用过程 VALL-2

3. C 语言的过程调用与返回

(1) 过程调用

- 过程调用的四元式序列

(param, entry(t_1), __, __)

.....

(param, entry(t_n), __, __)

(call, entry(P), n, __)

四元式的第一个元素是操作的运算符或函数，第二、三个元素是操作对象，第四个元素是结果；(param, entry(t_1), __, __) (param, entry(t_n), __, __) 表示对变量 t_1 到 t_n ，用 param 进行操作，得到的结果保存在对应四元式的最后一个位置 __；(call, entry(P), n, __) 表示调用 P 函数。

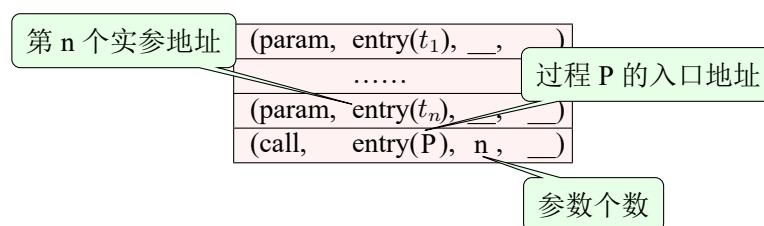


图 6.15: 调用函数过程四元式序列

图6.15 四元式序列描述调用函数的过程。 t_1 到 t_n 表示函数的实参地址，entry(P) 表示调用函数的入口地址，参数个数是 n 。

- 对应的目标指令

四元式: (param, entry(t_i), __, __)

作用: 现有主调过程的活动记录，此时还未执行到函数 P 对应的活动记录。构建一个过程 P 所对应的活动记录，(param, entry(t_i), __, __) 将 t_i 写入子程序 P 活动记录中的形参区对应的位置。

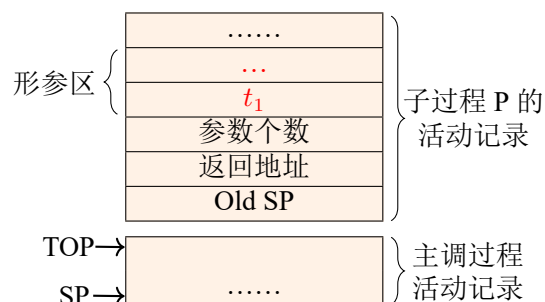


图 6.16: C 语言调用过程 VALL-3

对应指令:

$(i + 3)[TOP] := \text{entry}(t_i).\text{Addr}$ //将 t_i 地址填到活动记录的形参区去

以上语句表示: 在 TOP 地址之上第 $i + 3$ 个单元，保存变量 t_i 的地址。 $(i + 3)$ 表示在 TOP 地址之上增加 $i + 3$ 个偏移量，因为形参区的第一个单元与 TOP 之间相差 3 个单元，所以 $i + 3$ 可以索引到当前所要访问的形参编号。

四元式: (call, entry(P), n, __)

作用: 表示要执行过程 P, 但此时 P 活动记录里的一些必要信息还没有填写。(call, entry(P), n, __) 填写 Old SP 和参数个数 n。

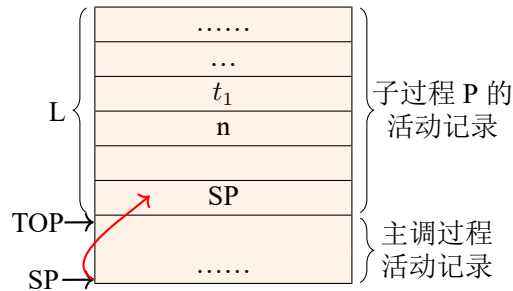


图 6.17: C 语言调用过程 VALL-4

对应指令:

```
1[TOP] := SP      //保护现行 SP
3[TOP] := n       //传递参数个数
JSP    P
```

以上语句表示: Old SP 将上一个过程即主调过程的 SP (指向主调过程的首地址), 填写在 TOP 地址之上 1 个偏移量的位置。参数个数 n 填写在 TOP 地址之上 3 个偏移量的位置。最后跳转到 P, 指的是汇编语言里真正要执行的函数过程的首地址, 不是活动记录的首地址。

- 子过程 P 需完成自己的工作: 定义自己的活动记录

此时, SP 和 TOP 还未指向当前要执行的子过程 P 的活动记录, 需要分别指向子过程 P 的起始地址和终止地址, 还需要填写 P 执行之后的返回地址。(假设 P 活动记录长度为 L)

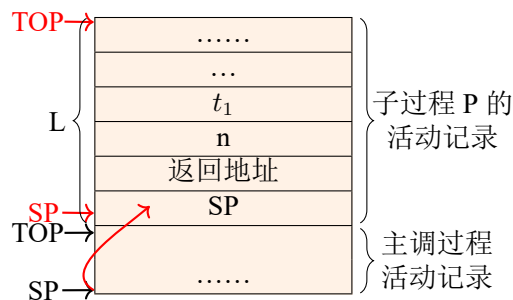


图 6.18: C 语言调用过程 VALL-5

对应指令:

```
SP := TOP + 1      //定义过程 P 的 SP
1[SP] := 返回地址  //保护返回地址
TOP := TOP + L     //定义新 TOP
```

(2) 过程返回

- 过程返回的四元式: (ret, __, __, __)

ret 指令表示返回。

- 对应的目标指令：

此时，子过程 P 执行结束，返回主调过程，需要将 TOP 和 SP 重新指向主调过程的起始和终止位置：TOP 移动到主调过程的最后一个单元，SP 的前一个单元；SP 移动到 P 活动记录中 Old SP 记录的地址，即主调过程活动记录的 SP。最后根据子过程 P 的返回地址，跳转回调用位置的下一条语句。（注：返回地址表示函数真实执行的位置，活动记录中记录函数里所存储的量的位置。）

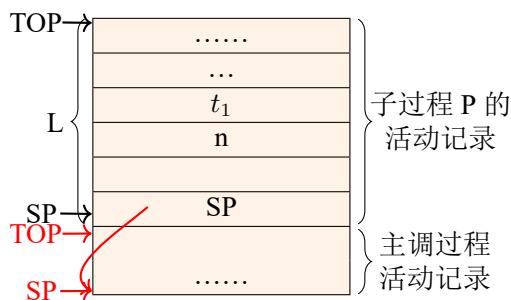


图 6.19: C 语言调用过程 VALL-6

对应指令：

```

TOP := SP - 1      //恢复 TOP
SP := 0[SP]       //恢复 SP
X := 2[TOP]       //取返回地址，X 为某一变址器
UJ    0[X]        //按 X 中的返回地址实行变址转移

```

6.5.4 嵌套过程语言的栈式存储分配

1. 标识符的作用域

(1) 过程嵌套的一个关键问题：标识符的作用域问题

标识符的作用范围往往与它所处的过程相关，也就是说，同一个标识符，在不同的程序段里，代表不同的对象，具有不同的性质，因此要分配不同的存储空间。

(2) 标识符的有效范围：服从最小作用域原理

- 在外层未定义，而在内层定义的，服从内层定义；
- 在外层已定义，而在内层未定义的，服从外层定义；
- 在外层已定义，而在内层也定义的，在外层服从外层定义，在内层服从内层定义（就近原则）。

2. 活动记录

(1) 问题的提出：

过程 Q 可能会引用到它的任意外层过程的最新活动记录中的某些数据，该如何存储？

(2) 解决问题的思想：

为了在活动记录中查找这些非局部名字所对应的存储空间，过程 Q 运行时必须设法跟踪它的所有外层过程的最新活动记录的地址。

(3) 解决方案:

活动记录中增加**静态链**如图6.20的最新活动记录的首地址。

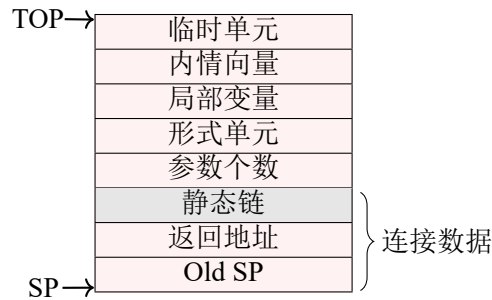


图 6.20: 添加静态链的活动记录

3. 嵌套层次显示表 (display) 和活动记录结构

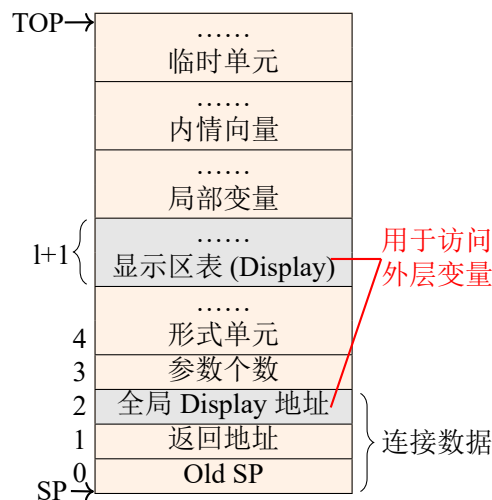


图 6.21: 添加嵌套层次显示表的活动记录

(1) 连接数据区: 0 2

- Old SP——主调过程的活动记录首地址
- 全局 display 地址——主调过程的显示区表首址，用于访问当前活动记录所有外层的活动记录信息

(2) 参数个数: 3

(3) 形参值单元区: 入口为 4

- 换名形参 (vn)——分配 2 个单元 (地址传递)
- 赋值形参 (vf)——按相应类型长度分配

(4) 显示区表 (display): 指向外层活动记录的指针, 占 I+1 个单元

I 为层次号, 包含直接外层嵌套的 I 个过程的活动记录的首地址, 再加上本过程的活动记录首地址

(5) 局部变量区: 入口为 $\text{off} + I + 2$

- off 为形参区最后一个值单元地址
- 局部变量值单元按相应类型长度分配地址
- 类型标识符、常量标识符等不分配值单元；常量放在常数表，跟函数表没有关系

(6) 临时变量区：

编译系统定义的变量，按局部变量值单元分配原则分配地址

4. Display 表的建立

设过程调用关系为 $Q() \rightarrow R()$ ，且 $R()$ 的层次号为 I ，则 Q 与 R 的 display 表的关系如图6.22:

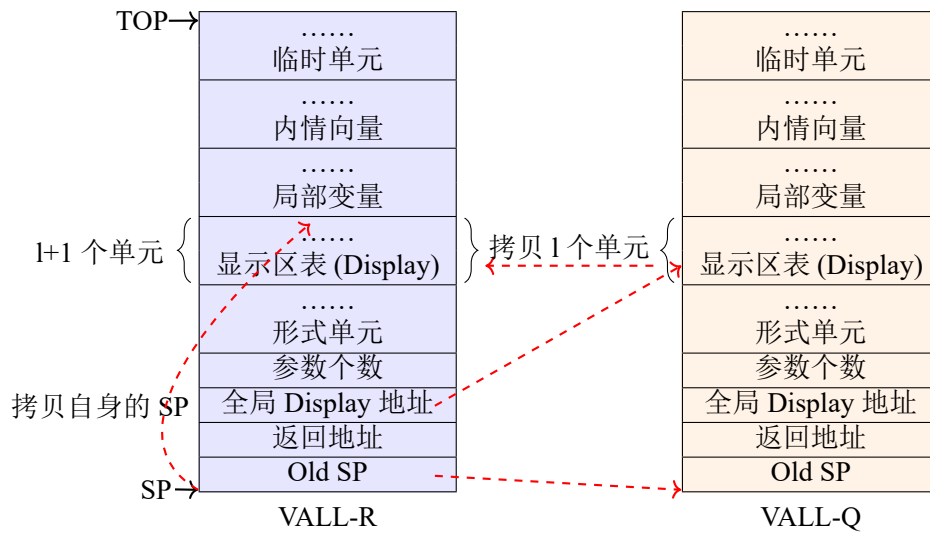


图 6.22: Q 与 R 的 display 表关系

R 的活动记录：

- Old SP: 指向 Q 活动记录的首地址。
- 全局 Display 地址: 指向 Q 的显示区表首地址。让当前活动记录能访问到所有外层的活动记录，外层的活动记录存储在 Q 的活动记录的显示区表里。
- 显示区表 (Display): 长度为 $I+1$ 个单元。从 Q 的显示区表拷贝 I 个单元（第 I 层拷贝 I 个），并将自身活动记录首地址写入显示区表的第 $I+1$ 个单元。

例 6.7 设有 Pascal 程序片段如图6.23:

这是 Pascal 中一个完整程序形式，函数（或过程）是用 program（或 procedure）定义的。P 是主程序，除了定义两个变量外，还定义了两个子程序，分别是 Q 和 S，Q 又定义了自己的内部函数为 R。因此该程序片段中，P 是 0 层，Q 和 S 是 1 层，R 是 2 层。

主控程序的代码在最后的 begin 到 end，调用了函数 S，S 代码段中，调用了函数 Q，Q 代码段中调用了函数 R。因此，整个过程的调用关系为 $P \rightarrow S \rightarrow Q \rightarrow R$ 。

根据调用关系，我们可以判断活动记录栈的大致形式如下。下面对 P、S、Q、R 的活动记录进一步说明，模拟运行时的活动记录。图6.24是整个调用过程的内存映像。

- P 的活动记录：

P 为 0 层，由于 P 是系统调用的，Old SP 为 0，返回地址的值运行时存放断点地址，全局 Display 地址为 0，参数个数为 0，Display 表长度为 1，存放自身活动记录首地址 0。根据函数定义填写局部变量为 a 和 x ，用 “ $a-(0,5)$ ” 表示变量 a 的层次号为 0，偏移量为 5。

- S 的活动记录:

P 调用 S，S 的活动记录载入内存。S 的 Old SP 指向 P 的活动记录首地址 0，返回地址内容运行时进行填写，全局 Display 指向 P 的 Display 表首地址 4，参数个数为 0。S 是 1 层，Display 表长度为 2，先拷贝 P 的 Display 表 0，再写入自身活动记录首地址 13，根据函数定义填写局部变量 c 和 i 。

- Q 的活动记录:

S 调用 Q，Q 的活动记录载入内存。Q 的 Old SP 指向 S 的活动记录首地址 13，返回地址内容运行时进行填写，全局 Display 指向 S 的 Display 表首地址 17，参数个数为 1，接着存放形参 b 。Q 也是 1 层，Display 表长度为 2，拷贝 S 的 Display 表中前 1 个单元内容 0，再写入自身活动记录首地址 27，根据函数定义填写局部变量 i 。

- R 的活动记录:

Q 调用 R，R 的活动记录载入内存。R 的 Old SP 指向 Q 的活动记录首地址 27，返回地址内容运行时进行填写，全局 Display 指向 Q 的 Display 表首地址 35，参数个数为 2，依据参数类型进行存放形参 u 和 v 。R 是 2 层，Display 表长度为 3，拷贝 Q 的 Display 表中前 2 个单元 0 和 27，再放入自身活动记录首地址 41，然后根据函数定义填写局部变量 c 和 d 。

5. 值单元的地址分配值单元分配是依据活动记录的结构，在符号表中进行的。

例 6.8 Pascal 程序片段如下，P1 所在层 level=2，试给出符号表组织及值单元分配情况。图6.25左侧是符号表的内容，右侧紫色框是活动记录。

设:(1) 实型占 8 个存储单元，整型占 4 个单元，布尔型和字符型占 1 个单元

(2) 换名形参 vn 分配 2 个单元，赋值形参 vf 按相应类型长度分配

```
PROCEDURE P1(  $x$ : REAL; VAR  $y$ : BOOLEAN );
```

```
CONST  $\text{pai} = 3.14$ ;
```

```
TYPE arr = ARRAY [1..10] OF INTEGER;
```

```
VAR  $m$ : INTEGER;
```

```
 $a$ : arr;
```

```
 $l$ : REAL;
```

```
FUNCTION F1(  $z$ : REAL;  $k$ : INTEGER ): REAL;
```

```
BEGIN .....END;
```

```
.....;
```

```
BEGIN
```

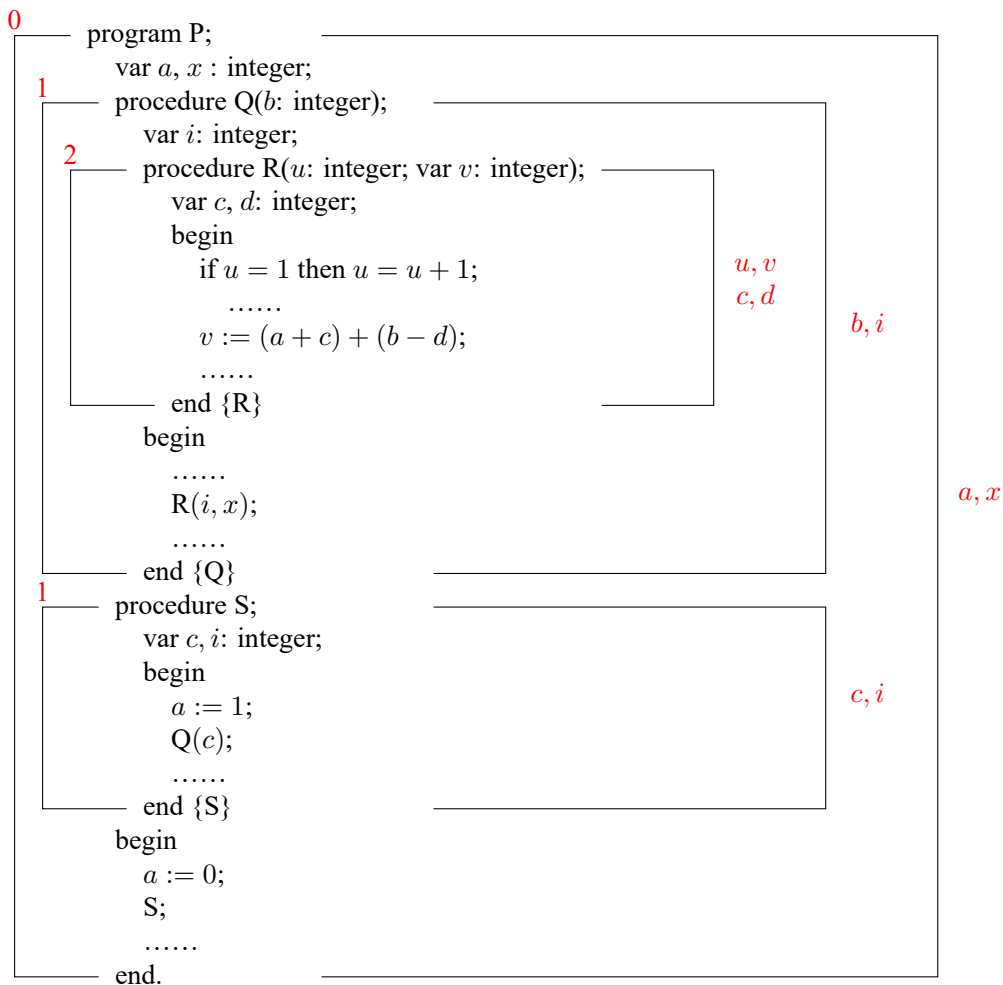


图 6.23: Pascal 程序片段

.....;

END;

该程序片段定义了一个过程 P1，层次号为 3。包括 1 个常量标识符，1 个类型标识符，3 个局部变量，1 个内部函数。P1 定义了 F1，F1 的层次号为 4。

符号表组织及值单元分配情况过程：

(1) P1 (过程)

- 填符号表 SYNBL: NAME 域填过程名字 P1, TYP 域没有返回值不填, CAT 域填写函数的种类 p (过程), ADDR 域指向函数表。
- 填函数表 PFINFL: LEVEL 域填函数的层次号 3, OFF 域填区距 3, FN 域填函数的变量个数 2, ENTRY 域填函数入口物理地址 Entry, PARAM 域指向形参表。
- 填形参表 PARAM: 根据 x 和 y 的名字、类型、种类、地址, 填写 NAME、TYP、CAT、ADDR 四项内容。其中 ADDR 内容在填完 VALL 之后, 填入变量的层次号和偏移, x 对应 (3,4), y 对应 (3,12)。填好形参表之后, 在符号表中填入变量 x 和 y 的相关信息 (与形参表中内容一致)。

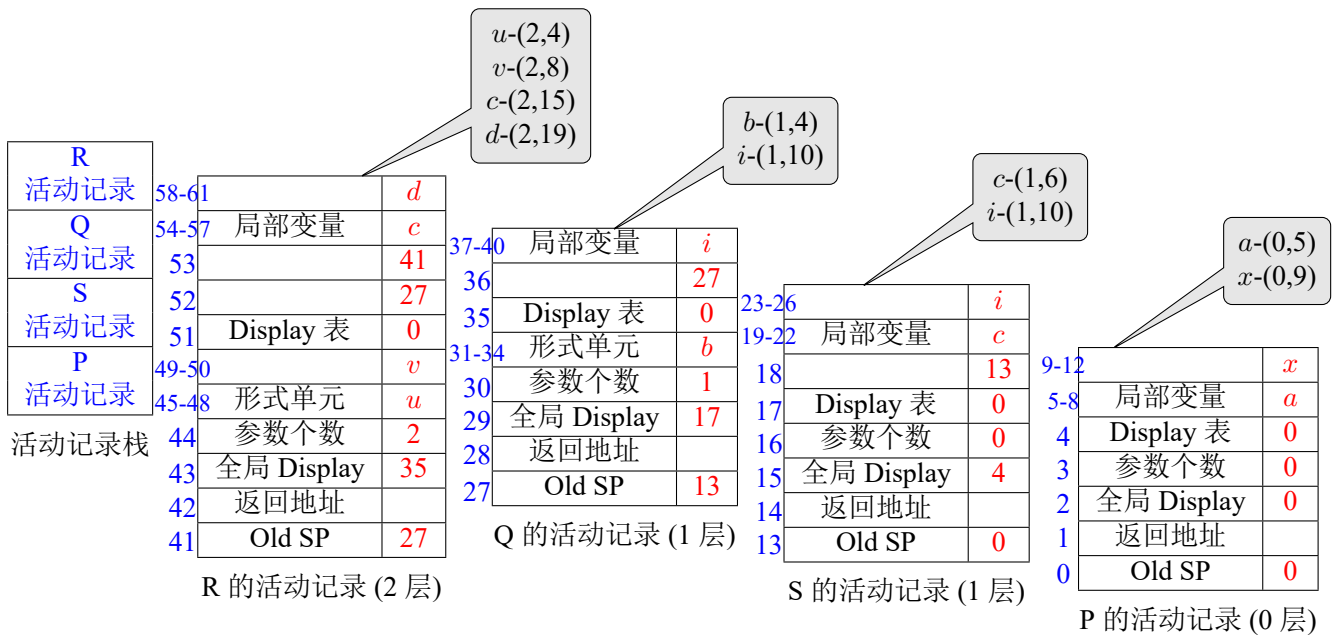


图 6.24: 活动记录栈调用过程

- 填活动记录 VALL: 参数个数为 2; 形式单元填 x , 实型占 8 个单元, 偏移量为 4-11, 填 y , 布尔型占 1 个单元, 偏移量为 12-13。Display 表长度为 4。

(2) pai (常量)

- 填符号表 SYNBL: 在符号表中继续填 pai 的相关内容, NAME 域填常量名字 pai, TYP 域填常量类型 rtp (实数型, 指向类型表的 r 项), CAT 域填写常量种类 c。ADDR 域指向常量表 CONSL。
- 填常量表 CONSL: 填入 3.14。

(3) arr (数组类型)

- 填符号表 SYNBL: 在符号表中继续填 arr 的相关内容, NAME 域填名字 arr, TYP 域指向数组的定义——类型表。
- 填类型表 TYPEL: 没有数组的定义, 要新增。TVAL 域填类码 a, TPOINT 域指向数组表。
- 填数组表 ALNFL: LOW 域填数组的下界 1, UP 域填数组的上界 10, CTP 域填数组元素类型 itp (整数型, 指向类型表的 i 项)。CLEN 域填值单元的长度 4。
- 填符号表 SYNBL: CAT 域填写数组种类 t (类型, 可以用作定义其他变量的数据类型)。ADDR 域指向长度表。
- 填长度表 CONSL: 填入 40。不填活动记录 VALL, 因为 arr 是类型不是变量。

(4) m (局部变量)

- 填符号表 SYNBL: 在符号表中继续填 m 的相关内容, NAME 域填变量名字 m, TYP 域填 itp, CAT 域填写变量种类 v, ADDR 域填活动记录中变量的层次号 (偏移量) 即 (3,18)。

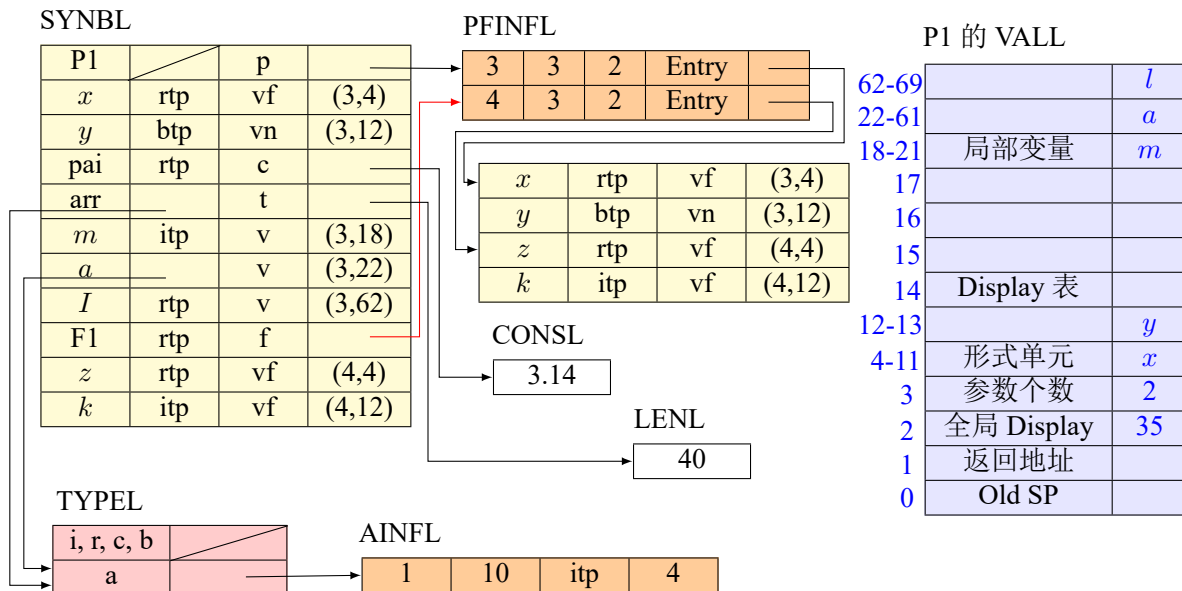


图 6.25: 符号表 + 活动记录填写过程

- 填活动记录 VALL: 在 Display 表上方填局部变量 m , 整型占 4 个单元, 偏移量为 18-21。

(5) a (局部变量)

- 填符号表 SYNBL: 在符号表中继续填 a 的相关内容, NAME 域填变量名字 a , TYP 域指向类型表中 arr 定义的数组类型 a , CAT 域填写变量种类 v , ADDR 域填活动记录中变量的层次号 (偏移量) 即 (3,22)。
- 填活动记录 VALL: 局部变量 m 上填 a , 数组类型占 40 个单元, 偏移量为 22-61。

(6) I (局部变量)

- 填符号表 SYNBL: 在符号表中继续填 I 的相关内容, NAME 域填变量名字 I , TYP 域填 rtp , CAT 域填写变量种类 v , ADDR 域填活动记录中变量的层次号 (偏移量) 即 (3,62)。
- 填活动记录 VALL: 局部变量 a 上填 I , 实型占 8 个单元, 偏移量为 62-69。

(7) F1 (函数)

- 填符号表 SYNBL: NAME 域填过程名字 F1, TYP 域填函数返回值类型 rtp , CAT 域填写函数的种类 f (函数), ADDR 域指向函数表。
- 填函数表 PFINFL: LEVEL 域填函数的层次号 4, OFF 域填区距 3, FN 域填函数的变量个数 2, ENTRY 域填函数入口物理地址 Entry, PARAM 域指向形参表。
- 填形参表 PARAM: 根据 z 和 k 的名字、类型、种类、地址, 填写 NAME、TYP、CAT、ADDR 四项内容。其中 ADDR 内容在填完 VALL 之后, 填入变量的层次号和偏移, z 对应 (4,4), y 对应 (4,12)。填好形参表之后, 在符号表中填入变量 z 和 k 的相关信息 (与形参表中内容一致)。

- 填活动记录 VALL: 参数个数为 2; 形式单元填 z , 实型占 8 个单元, 偏移量为 4-11, 填 k , 整型占 4 个单元, 偏移量为 12-15。Display 表长度为 5。

第7章 中间代码生成

7.1 导入

7.1.1 为什么要研究语义分析

词法分析是基于有限状态自动机，扫描识别字符串序列中的语素（关键字、标识符、运算符、常数、界符等），为下一部的语法分析做基础。

语法分析是基于自底向上或自顶向下的分析方法，检查词法分析处理过后的语素集合是否能被该编程语言产生式匹配的过程，这个过程会隐式的生成语法树。在语法分析里，我们可以检查代码更大粒度的正确性。

符号表被很多人认为是语义分析的一部分（剩下的一块拼图就是本章的中间代码生成）。在平时编译阶段，对于类型检查，如将整型当作数组类型使用；控制流，如 C 语言中大括号的使用是否前后呼应；重复定义，如不能在一个函数中重复定义两次同一个变量；作用域，如全局变量和局部变量的使用等，这些问题，通过语义分析可以找出。

自然语言的语义与程序语言的语义是不一样的，严格意义上讲，自然语言的语义会非常丰富，而程序语言的语义相对固定。语义分析的目的是利用语法分析的结果，把源语言转换成机器可读的形式，称为中间代码，这个过程也称为语义分析。生成中间代码的过程，要去定义一些语义动作，这些语义动作就是语法分析每一个局部的结果，后面会有例子。

7.1.2 为什么需要生成中间代码

在程序编译过程中，可执行代码与具体的操作系统、硬件等相关，而生成的中间代码与操作系统、硬件设施无关，具有较好的移植性，方便了编译器系统的开发。另一方面，也便于进行优化操作，如程序中的 $1+2$ ，在经过优化后直接变成 3 ，这样的话，就节省了一步，如果是在循环或实时系统中，这样的优化就十分关键

另一个原因就是直接把源语言转换成汇编指令，这种方式的好处是效率高，缺点是无法把中间过程进行分解，系统复杂，不易调试。常用手段是两小步的方法，先把源语言转换成中间代码，然后再把中间代码转换成目标语言。

概括性的总结一下这一小节：

1. 易于修改，对于复杂的系统来讲，一般用模块化进行开发，把大任务或复杂问题简化，这里有一个重要的概念——耦合，我们希望模块间的耦合度不要太大，加入中间代码也是降低模块间耦合度的一种方式，使用中间代码这种特殊的表达形式来作为中间衔接，而不是程序接口这种简单复杂的形式，易于开发、调试、维护。
2. 跨平台，易于迁移。
3. 首先生成中间代码给了我们更多的优化算法介入的空间，因为越抽象底层的代码优化起来难度更大，先生成可转为汇编的中间语言，让我们既朝底层更进一步，却也保留了一些优化的可能性，毕竟汇编确实看起来让人觉得有些繁琐。

7.2 中间代码生成

7.2.1 常用的中间代码形式

中间代码可以理解成一个源语言句子表示成机器容易理解的形式，这种形式可以很容易地转化成目标语言。

设有赋值语句： $x = a * b + c$ ，则有以下中间语言：

(1) 逆波兰式： $ab * c + =$

在编写计算器程序时，使用的就是逆波兰式的形式。以 $a+b$ 为例，我们平时写的 $a+b$ 这种形式，是中缀式，而逆波兰式形式得到的是 $ab+$ ，又称后缀式，将操作项放在前面，运算符放在后面。

(2) 四元式

(1) (* a b t1) (2) (+ t1 c t2) (3) (= t2 _ x)	或	(1) t1 = a * b (2) t2 = t1 + c (3) x = t2
---	---	---

四元式表示一元或二元运算，可以顺序地把程序执行的过程表达出来。第一条四元式表示，用 $*$ 对 a 和 b 进行操作，结果赋给 $t1$ 。第二条表示， $t1$ 与 c 相加，结果赋给 $t2$ 。第三条表示，把 $t2$ 的值赋给 x 。这个过程与赋值语句 $x = a * b + c$ 是一样的。四元式也可以简单地写成二元运算的形式，左边和右边框表示的含义是一样的，一般写成左边这种方式，比较规整。

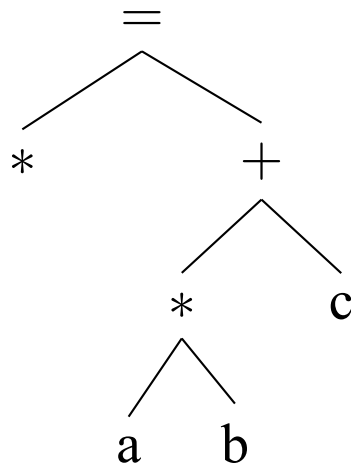
四元式形式具有更强的表达能力和可读性，因此在后续中间代码生成中使用更多。

(3) 三元式

三元式和四元式的区别在于，四元式的最后一项叫结果变量，而三元式没有，是用①来表示计算的结果。

①	(*	a	b)
②	(+	①	c)
③	(=	②	x)

(4) 语义树



把 $a*b$ 进行计算，结果在与 c 相加，最终把结果赋给 x 。

这四种方式是常用的形式，还有很多其它方式，这里不一一列举。这四种方式都是非常有效的中间代码表示形式，把程序语言表示成一种便于计算机计算的方式。这四种方式也各有优缺点，简单总结一下：

逆波兰式简单；四元式清楚；
三元式节省；语义树直观；

7.2.2 各种语法成分的中间代码设计

逆波兰式

给定一个表达式，可以把它写成逆波兰式的形式。逆波兰式把运算符放在操作数的后面比如：

$$a + b \Rightarrow ab+$$

表达式的逆波兰式设计

设 $pos(E)$ 为表达式 E 的逆波兰式；则：

$$\textcircled{1} pos(E1\omega E2) = pos(E1)pos(E2)\omega$$

$$\textcircled{2} pos((E)) = pos(E)$$

$$\textcircled{3} pos(i) = i$$

其中， ω 是运算符， i 是运算对象（变量或常量）

【注】（1）上述三个定义式，为逆波兰翻译法则；（2）定义式中的 ω 为 $E1\omega E2$ 中最后运算的算符！例如，式 $a+b*c$ 中， $E1$ 相当于 a ， $E2$ 相当于 $b*c$ ， ω 为 $+$ 。

例 7.1 表达式逆波兰式翻译示例

$x * (a + b/d) < (-e + 5)$, 求该表达式的逆波兰式 $pos()$?

因为表达式中 $<$ 的优先级最低, 所以 ω 为 $<$, E_1 为 $x*(a+b/d)$, E_2 为 $(-e+5)$, 所以得 $pos(x*(a+b/d))pos((-e+5))<$ 。 $pos(x*(a+b/d))$ 中 $*$ 最后运算, 得 $x pos((a+b/d))*$; $pos((-e+5))$ 根据式②, 得 $pos(-e+5)$ 。 $pos((a+b/d))$ 中 $+$ 最后算, 得 $a pos(b/d)+$, $pos(b/d)$ 再得到 $bd/$ 。 $pos(-e+5)$ 里最后算的是 $+$, 得 $pos(-e)5+$, 在这里把 $-e$ 的定义取负, 是一个单目运算, 于是 $pos(-e)$ 得 $e-$ 。最终得到如下所示逆波兰式。

$$\begin{aligned}
 & \therefore pos(x * (a + b/d) < (-e + 5)) \\
 & = pos(x * (a + b/d)) pos(-e + 5) < \\
 & = x pos((a + b/d)) * pos(-e + 5) < \\
 & = x a pos((b/d)) + * pos(-e + 5) < \\
 & = x a bd/ + * pos(-e + 5) < \\
 & = x a bd/ + * pos(-e)5 + < \\
 & = x a bd/ + * e - 5 + < \\
 & \therefore pos() = xabd/ + * e - 5 + <
 \end{aligned}$$

注

单目运算符的逆波兰式, $pos(-e) = e-$, 请大家和双目运算符加以区分

例 7.2 其它语法成分的逆波兰式设计示例

定义 7.1 单目运算 “-” 的逆波兰式

$$pos(-E) \Rightarrow pos(E)- \text{ 或 } 0 pos(E) -$$

定义 7.2 赋值语句的逆波兰式

$$v=E \Rightarrow v pos(E) =$$

例 7.3 $x=(a+b)/-e*5$, $pos()$?

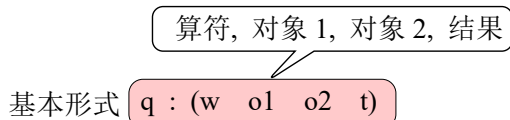
首先, $=$ 优先级最低, 得 $x pos((a+b)/-e*5)=$ 。 $pos((a+b)/-e*5)$ 里最后操作的是 $*$, 得 $pos((a+b)/-e)pos(5)*$ 。接下来的过程省略, 结果如图所示

$$\begin{aligned} & \therefore pos(x = (a + b) / - e * 5) \\ & = x pos((a + b) / - e * 5) = \\ & = x pos((a + b) / - e) pos(5) * = \\ & = x pos(a + b) pos(-e) / 5 * = \\ & = xab + e - / 5 * = \\ & \therefore pos() = xab + e - / 5 * = \end{aligned}$$

四元式

逆波兰式更多是用来计算表达式，其它比如说 if 语句、while 语句不太容易用逆波兰式，因为不是简单的代数运算。还有很多标志符没有操作数，语义很复杂，这时候编译器更常用四元式。

1、表达式的四元式设计：



四元式如 q 所示，包括四个部分：算符、对象 1、对象 2 和结果，分别用 ω 、 $o1$ 、 $o2$ 和 t 表示。 $o1$ 和 $o2$ 用 ω 进行计算，结果存到 t 。

设 $quat(E)$ ， $res(E)$ 分别为表达式 E 的四元式和结果变量。

$$\begin{aligned} \textcircled{1} \quad & quat(E_1 \omega E_2) = \begin{array}{l} quat(E_1) \\ quat(E_2) \\ q : (\omega \quad res(E_1) \quad res(E_2) \quad t) \end{array} \\ \textcircled{2} \quad & quat((E)) = quat(E) \\ \textcircled{3} \quad & quat(i) = \text{空}, \quad res(i) = i \end{aligned}$$

其中： ω (运算符); i 运算对象 (变量或常数)

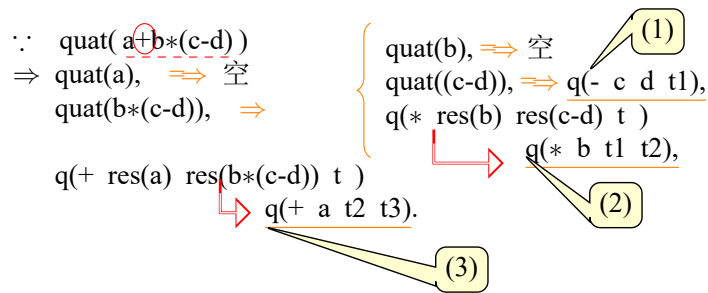
规则①，求 $E1\omega E2$ 的四元式 $quat(E1\omega E2)$ ，先写出 $E1$ 、 $E2$ 的四元式 $quat(E1)$ 、 $quat(E2)$ ，结果分别是 $res(E1)$ 、 $res(E2)$ 。这两个结果经过 ω 计算，最终放到一个变量 ti ， ti 就是整个表达式的结果 $res(E1\omega E2)$ 。

【注】

- (1) 方框内的三个定义式，为四元式翻译法则；
- (2) 定义式中的 ω 为 $E1\omega E2$ 中最后运算的算符！

例 7.4 表达式四元式翻译示例：

$$a+b*(c-d)$$



ω 是 +, E1 的四元式是 $\text{quat}(a)$, E2 的四元式是 $\text{quat}(b*(c-d))$, 最后一条语句是 $q(+ \text{res}(a) \text{res}(b*(c-d)) t)$, E1 的四元式是 $\text{res}(a)$, E2 的四元式结果是 $\text{res}(b*(c-d))$, 把它们做加法运算, 将结果存到变量 t 。

$\text{quat}(a)$ 是空, $\text{quat}(b*(c-d))$ 需要根据准则继续求。 ω 是 *, $\text{quat}(b)$ 是空, $\text{quat}((c-d))$ 是 $q(- c d t1)$, 对于 $q(* \text{res}(b) \text{res}(c-d) t)$, $\text{res}(b)$ 是 b , $\text{res}(c-d)$ 是 $t1$, 将它们进行计算存在 $t2$, 有 $q(* b t1 t2)$ 。

对于 $q(+ \text{res}(a) \text{res}(b*(c-d)) t)$, $\text{res}(a)$ 是 a , $\text{res}(b*(c-d))$ 是 $t2$, 把结果放在 $t3$, 有 $q(+ a t2 t3)$ 。

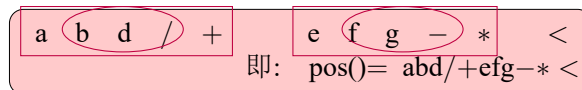
按最终结果的生成顺序, 可得:

- | |
|-----------------|
| (1) (- c d t1) |
| (2) (* b t1 t2) |
| (3) (+ a t2 t3) |

例 7.5 表达式逆波兰式和四元式最简翻译算法示例:

$(a+b/d) < e*(f-g)$, $\text{pos}()$?, Quat ?

逆波兰式生成要点: 运算对象顺序不变, 运算符紧跟运算对象之后! 则:



四元式生成要点: 按照运算法则, 依次生成四元式! 则:

- | |
|------------------|
| (1) (/ b d t1) |
| (2) (+ a t1 t2) |
| (3) (- f g t3) |
| (4) (* e t3 t4) |
| (5) (< t2 t4 t5) |

四元式实际上和逆波兰式对应, 先算 b/d , 结果和 a 相加, 然后算 $(f-g)$, 结果和 e 相乘, 最终把这两个结果 $t2$ 和 $t4$ 用 $<$ 比较, 比较的结果存到 $t5$ 。

2、赋值语句的四元式设计

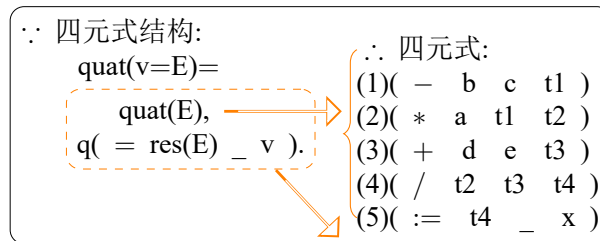
设有赋值语句: $v=E$

$$v = E \Rightarrow \begin{matrix} \text{quat}(E) \\ \text{q}(:= \text{res}(E) _ v) \end{matrix}$$

先算表达式右侧的 E，再进行赋值。

例 7.6 $x=a*(b-c)/(d+e)$

按照上面的规则，先求 $\text{quat}(E)$ ，把结果赋值给 v。Quat(E) 按计算的顺序写下来，最后用 t4 代替 $\text{res}(E)$ 。结果如下图所示：



3、转向语句与语句标号的四元式设计

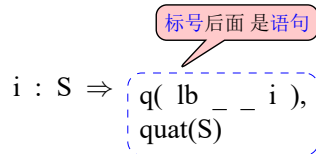
语句标号为转向语句提供转入语句标识，二者用标号相关联。

(1) 设有转向语句： $\text{goto } i$

则有 $\text{goto } i \Rightarrow \text{q}(\text{gt} _ _ i)$

其中 gt 表示 goto

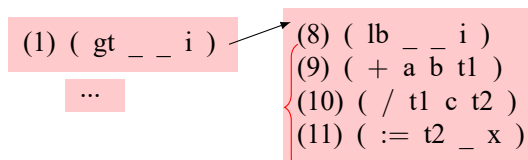
(2) 设有标号语句： $i: S$



其中，i 表示标号，lb 表示 label，S 是后面的语句。

例 7.7 $\text{goto } i; \dots i: x=(a+b)/c;$

则有四元式序列



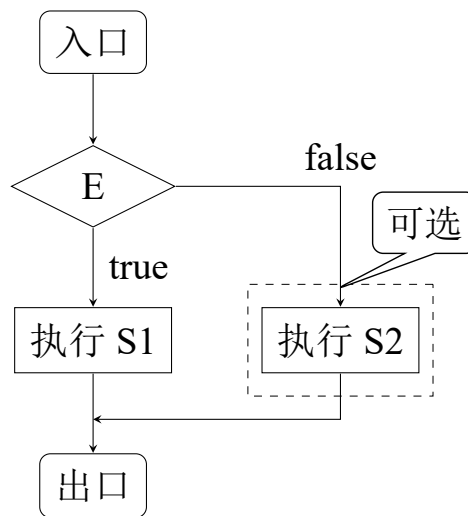
4、条件语句的四元式设计

(1) 文法：

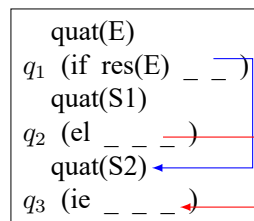
$$S \rightarrow \text{if}(E) S1 [\text{else } S2]$$

可选

(2) 语义结构：从入口进，判断 E，为真则执行 S1，为假则执行 S2，S2 路径是可选的，最终都到达出口。



(3) 四元式结构：



【注】

q₁: 当 res(E)=False, 则转向 S2 入口四元式;

q₂: 无条件转向出口四元式;

q₃: 条件语句出口四元式。

首先执行的是 quat(E), 计算表达式 E 的结果, 存到 res(E)。

$$q_1(\text{if res}(E) _ _)$$

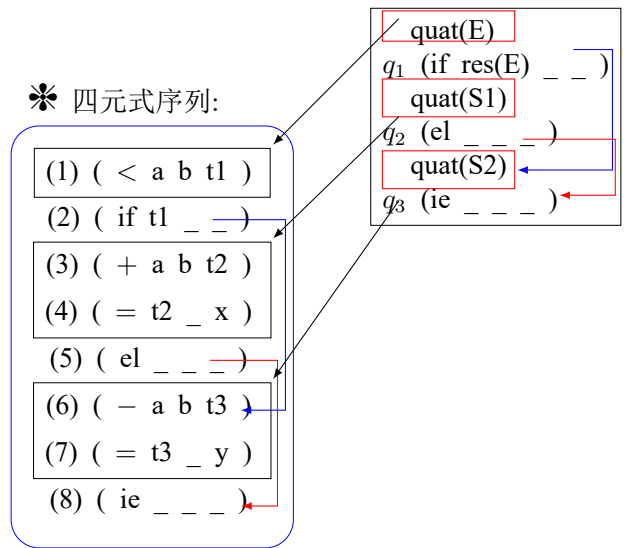
表示判断 res(E) 是否为假, 若为假就跳转到 quat(S2), 若为真就顺序执行, 执行 quat(S1), 执行完后跳转到出口。

例 7.8 条件语句四元式翻译示例:

if (a<b) x=a+b;

else y=a-b;

四元式序列:

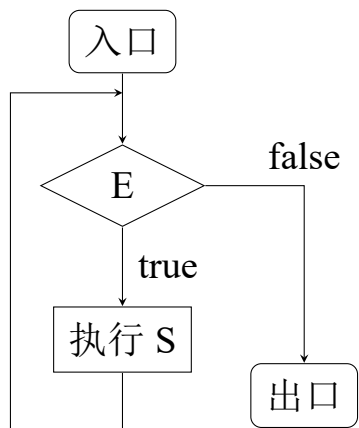


【注】如果语句中没有 else y=a-b 部分，则四元式结构中和四元式序列中的相应部分也不存在了！

5、循环语句的四元式设计：

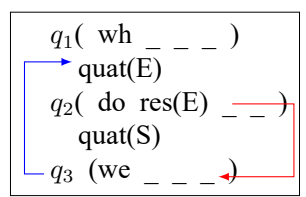
(1) 文法：S → while (E) S

(2) 语义结构：



逻辑是：while 语句先判断 E，若 E 为真就执行 S，再判断 E，还为真则继续执行 S，直到 E 为假则跳到出口。

(3) 四元式结构：



一开始定义 q1，它是一个标识，表示 while 的入口。接下来计算 quat(E)，然后判断 res(E) 是否为假，为假则跳出循环，为真则执行 S，再通过语句 q3 返回计算 quat(E)。

【注】

q1: while 语句的入口四元式 (提供转向 E 参照);

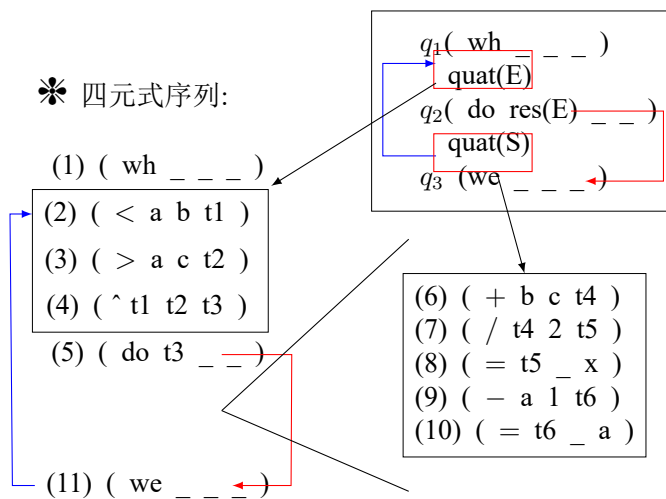
q2: 当 res(E)=False 转向出口四元式;

q3: while 尾 (兼循环转向 E) 四元式。

例 7.9 循环语句四元式翻译示例:

while (a<b^>c)

{x=(b+c)/2; a=a-1; }

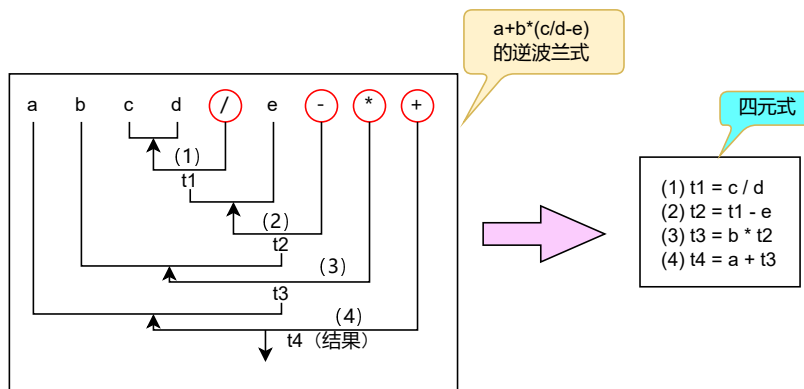


表达式 $a < b \wedge a > c$ 用四元式 (2)(3)(4) 来表示, 结果存到 t3。S 用四元式 (6)(7)(8)(9)(10) 表示, (6)(7)(8) 对应 $x=(b+c)/2$, (9)(10) 对应 $a=a-1$ 。最后是 q3, 起到跳转的作用, 跳转到四元式 (2)。

例 7.10 逆波兰式 (四元式) 计算过程示例:

【算法】 设置一个“栈”, 每当 NEXT(w), 重复执行:

- (1) 若 w= 运算对象, 则压入栈中暂存: PUSH(w);
- (2) 若 w= 运算符, 则弹出栈顶对象计算之, 并把结果压栈。



$a+b*(c/d-e)$ 的逆波兰式是 $abcd/e-*+$, 自左向右扫描, 看到第一个运算符/, 就把 c 和 d 进行运算, 对应 t1, 写成四元式 $t1=c/d$ 。继续往后扫描, 看到第二个运算符 -, 就把栈顶的两个元

素 e 和 t_1 做减法，结果存在 t_2 。以此类推，可以得到 t_3 、 t_4 。

7.3 中间代码翻译算法

7.3.1 属性文法

定义 7.3 属性文法是上下文无关文法在语义上的扩展，可定义为如下三元组：

$A=(G, V, E)$ ；其中， G （文法）， V （属性集）， E （属性规则集）。

说明：

1. 属性：代表与文法符号相关的信息，这里主要指语义信息（类型、种类、值和值地址...）；文法产生式中的每个文法符号都附有若干个这样的属性。比如，描述一个人，身高、头发、颜色、出生地、年龄，都可以称为人的属性。文法符号也可以用若干个属性来描述，比如一个变量，整型/浮点型、值都是属性。
2. 属性可以进行计算和传递，属性规则就是在同一产生式中，相互关联的属性求值规则。比如，统计一个班级的总分，就是把所有同学的分数相加，如果把分数看成每一个人的属性，把所有分数相加就相当于对属性进行计算。
3. 属性分两类（按属性求值规则区分）：综合属性：其值由子女属性值来计算（自底向上求值）；继承属性：其值由父兄属性值来计算（自顶向下求值）。比如，对于继承属性，一般而言，姓氏是从父亲那“继承”的，该属性由父亲的属性决定。对于综合属性，父亲的属性需要用孩子的属性计算，以之前班级的例子为例，班级是学生一个更高层次的抽象，对于班级的成绩属性，需要把所有学生的成绩相加求平均来求。

例 7.11 属性文法构造示例：

算数表达式的属性文法：

设： $X.val$ 为文法符号 X 的属性值；

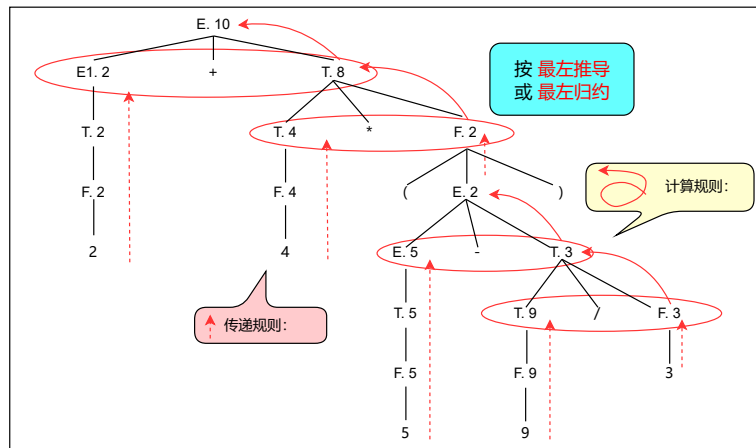
下述属性文法用于算术表达式的求值运算：

$$\begin{array}{ll}
 E \rightarrow E^1 + T ; & || E.val := E^1.val + T.val \\
 E \rightarrow E^1 - T ; & || E.val := E^1.val - T.val \\
 E \rightarrow T ; & || E.val := T.val \\
 T \rightarrow T^1 * F ; & || T.val := T^1.val * F.val \\
 T \rightarrow T^1 / F ; & || T.val := T^1.val / F.val \\
 T \rightarrow F ; & || T.val := F.val \\
 F \rightarrow (E) ; & || F.val := E.val \\
 F \rightarrow i ; & || F.val := i.val
 \end{array}$$

表的左侧是原始的文法，右侧是属性文法。对于 $E \rightarrow E^1 + T$ ， E^1 与 E 是不同的，其它加了上标 1 的符号同理。它对应的属性文法是右侧，将 E^1 的值加上 T 的值的的结果赋给 E 。-、*、/ 都类似。可以看出，该 $X.val$ 属性是综合属性。对于表达式， E 相当于树的根， E^1 、 T 相当于树的叶子，父亲的属性值是由孩子的属性值决定的。

属性计算过程示例：

根据算术表达式属性文法及其相应的属性求值规则， $2+4*(5-9/3)$ 的属性语法树：



如何算这个属性？基于最左推导或最左归约，传递规则是从下往上，比如 2 传给 F，F 传给 T，T 传给 E^1 ，然后还有相对复杂的计算，最后得到最上面的 E ，也就是整个表达式对应的文法符号的值为 10。这个过程和之前描述的属性文法是一致的，父亲的属性值需要通过孩子进行计算，也就是说，如果得到了属性的值，就可以得到整个表达式的计算结果。

7.3.2 语法制导翻译技术

定义 7.4 语法制导 (syntax_directed)

是指根据语言的形式文法对输入序列进行分析、翻译处理；核心技术是构造翻译文法——在源文法产生式中插入语义动作符号（翻译子程序），借以指明属性文法中的属性求值时机和顺序。

语法制导翻译指在语法分析过程中，通过加入语义动作，来完成中间代码的生成，生成的过程称为翻译。语法制导强调所有定义、计算、建模，全都是基于语法的，在本编译原理教程中，语法是核心，决定程序的结构或系统的架构，把词法和中间代码生成连接起来。

翻译文法可以看成是一种属性文法，带有属性的计算。语法制导的翻译，没有一个独立的程序来得到中间代码，中间代码的生成是贯穿在语法分析过程中的，语法分析执行完了，中间代码也就生成出来了。语法制导翻译技术由两部分构成：

语法分析技术 + 翻译文法构造技术

为叙述简便，除非临时详细指明，标识符 X 的属性一律视为符号表项指针，同时用 X 表示 $X.point$ 。接下来的大部分例子，将以算术表达式文法为例，探讨翻译文法的构造。

逆波兰式翻译文法构造示例：

例 7.12 算术表达式逆波兰式翻译文法

$$\begin{array}{l}
 G'(E) \\
 E \rightarrow T \mid E+T+ \mid E-T\{-\} \\
 T \rightarrow F \mid T*F* \mid T/F\{/ \} \\
 F \rightarrow ii \mid (E)
 \end{array}$$

该文法和简单的算术表达式稍有区别，大括号里面的是语义动作，去掉及里面的内容，剩下的内容就是标准的算术表达式文法。 a 的含义是输出 (a)。

符号串 $a*(b+c)$ 的分析（翻译）过程（推导法）：

$$\begin{array}{l}
 E \Rightarrow T \Rightarrow T * F\{*\} \Rightarrow F * F\{*\} \Rightarrow a\{a\} * F\{*\} \Rightarrow a\{a\} * (E)\{*\} \\
 \Rightarrow a\{a\} * (E + T\{+\})\{*\} \Rightarrow a\{i\} * (T + T\{+\})\{*\} \\
 \Rightarrow a\{a\} * (b\{b\} + c\{c\}\{+\})\{*\}
 \end{array}$$

导出序列

用推导的方式，首先， E 用 T 来推导，先推导 $*$ 乘法。然后是 $T*F*$ ，比原始文法多了 $*$ ，但还是按照文法内容完整推导出来。接下来用 $T \rightarrow F$ 把 T 推导成 F ，得 $F * F\{*\}$ 。再将第一个 F 推导成 $a\{a\}$ ，第二个 F 推导成 (E) ， E 就是 $b+c$ 。 (E) 再推导成 $(E + T\{+\})$ ，需要带上 $+$ 。接下来的步骤类似。

分解导出序列：

去掉动作符号： $a * (b + c)$源表达式;

去掉文法符号： $abc + *$逆波兰式;

大括号 $\{\}$ 是定义的一种动作，可以看作程序执行的一段代码，并不是文法真实的符号，代表的是程序要执行的一个操作。通过在语法中加入语义动作，在使用推导的方法进行语法分析后，就能得到一个动作序列，结果就是输出逆波兰式，达到生成中间代码的目的。

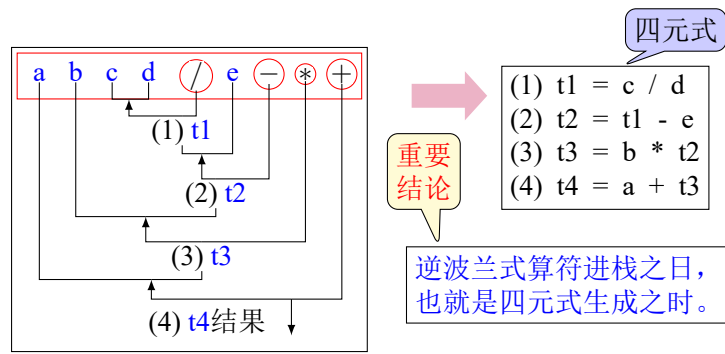
这个例子说明，通过改造文法，不需要额外地设计新的复杂程序，就能生成逆波兰式。

四元式翻译文法构造示例 1：

1、从逆波兰式的翻译到四元式的翻译：

假定： $\{a\}$ 为 $PUSH(a)$ ，即把 a 压入栈（语义栈）中。

则算术表达式 $a+b*(c/d-e)$ 的逆波兰式（生成）计算顺序：



往栈里依次压入元素 a、b、c、d，当遇到 / 时，就用栈顶的两个符号和 / 一起生成四元式 $t1=c/d$ 。同样，当遇到第二个运算符 - 时，就用栈顶符号 t1 和 e 生成第二个四元式。四元式 (3)(4) 同理。

{a} 代表了压栈操作，当遇到运算符，就用栈顶符号生成四元式，并把结果变量压回栈里。

四元式翻译文法构造示例 2:

2、算术表达式四元式翻译文法设计

定义:

SEM(m)——语义栈（属性传递、赋值场所），可以理解为一个指向符号表的指针，记录了所有关于这个标识符的信息；

QT[q]——四元式区，存储生成的四元式；

文法定义 $G''(E)$

$$E \rightarrow T \mid E + T\{GEQ(+)\} \mid E - T\{GEQ(-)\}$$

$$T \rightarrow F \mid T * F\{GEQ(*)\} \mid T / F\{GEQ(/)\}$$

$$F \rightarrow i\{PUSH(i)\} \mid (E)$$

其中:

(1) PUSH(i)——压栈函数（把当前 i 压入语义栈）；

(2) GEQ(ω)——表达式四元式生成函数:

① $t := NEWT; \{ \text{申请临时变量函数}; \}$

② $SEND(\omega \ SEM[m-1], SEM[m], t)$

③ $POP; POP; PUSH(t)$

生成一个四元式送 QT[q]

语义栈次栈顶、栈顶

有三个步骤：一，申请变量 t；二，生成四元式，用 ω 进行计算，操作数是栈顶的两个元素，SEM[m-1] 是次栈顶元素，SEM[m] 是栈顶元素，把结果存在 t；三，分别是弹栈、弹栈、压栈，即把栈顶的两个元素出栈，再把结果压入栈。

动作函数（序列）执行过程示例：

算术表达式 $a*(b/c-d)$ 的逆波兰式 $abc/d-*$ ，可得四元式动作序列：

动作序列	SEM[m]	QT[q]
PUSH(a) \rightarrow	a	
PUSH(b) \rightarrow	a b	
PUSH(c) \rightarrow	a b c	
GEQ(/) \rightarrow	a b c	(1) (/ b c t1)
PUSH(d) \rightarrow	a t1 d	
GEQ(-) \rightarrow	a t1 d	(2) (- t1 d t2)
GEQ(*) \rightarrow	a t2	(3) (* a t2 t3)
	t3	

PUSH(a)、PUSH(b)、PUSH(c) 分别压栈，GEQ(/) 用栈顶两个元素 b 和 c 生成四元式，弹出 b 和 c 并把结果 t1 压栈。之后同理，最终得到结果 t3。

7.3.3 四元式翻译文法设计扩展 1

1、赋值语句四元式翻译文法：

$S \rightarrow v \text{ PUSH}(v) = \text{EASSI}(=)$;

赋值语句的表达式 $v=E$ ，它的翻译文法步骤是通过加入两个语义动作 PUSH(v) 和 ASSI(=)，其中，ASSI(=)——赋值函数，把栈顶元素赋值给次栈顶元素，然后把栈顶和次栈顶元素弹出，如下图。

(1) SEND(:= SEM[m], __, SEM[m - 1]);
 (2) POP; POP;

2、标号、转向语句四元式翻译文法：

$S \rightarrow i \text{ PUSH}(i) : \text{LABEL}(i) S$;

$S \rightarrow \text{goto } i \text{ GOTO}(i)$;

其中，LABEL(i)——标号函数，跳转到语义栈栈顶的元素所对应的那行代码，然后将栈顶元素弹出，如 (1)SEND(lb __, __, SEM[m]); (2)pop; , SEM[m] 是 i; GOTO(i)——转向函数，跳转到 i，如 (1)SEND(gt __, __, i);

3、条件语句四元式翻译文法：

$S \rightarrow \text{if}(R) \{ \text{IF}(if) S; [\text{else} \{ \text{EL}(el) \} S] \{ \text{IE}(ie) \} \text{IF}(if)$ ——if 函数，用 if 来判断栈顶符号是否为假，SEM[m] 存的是 R 的结果；如图

```
(1)SEND(if SEM[m],_,_);
(2)POP;
```

EL(el)——else 函数，跳出 if；如图

```
(1)SEND(el _,_,_);
```

IE(i)——ifend 函数，指示 if 的结束；如图

```
(1)SEND(ie _,_,_);
```

循环语句四元式翻译文法：

$S \rightarrow \text{while Wh}() (R) \text{ DO}(\text{do}) S \text{ WE}(\text{we});$

WH(wh)——循环头函数，标识 while 的开始；如图

```
(1)SEND(wh _,_,_);
```

DO(el)——DO 函数，条件判断，判断 SEM[m] 栈顶元素的值是否为假，若为假则跳到函数尾，不为假则继续执行；如图

```
(1)SEND(do SEM[m],_,_);
(2)POP;
```

WE(we)——循环尾函数，表示条件的结束，还要再一次判断 R 的值是否为假，跳回到 while 的入口；如图

```
(1)SEND(we _,_,_);
```

【注】文法中的 R 为关系表达式（见条件语句四元式翻译文法）

4、说明语句四元式翻译文法：

(1)

原文法:

$$D \rightarrow id : T; D \mid id : T$$

$$T \rightarrow interger \mid real \mid array[num] \text{ of } T \mid \uparrow T$$

对该文法进行改造。

(2)

文法转换:

$$D \rightarrow id : T; D'$$

$$D' \rightarrow ; id : TD' \mid \varepsilon$$

$$T \rightarrow interger \mid real \mid array[num] \text{ of } T \mid \uparrow T$$

(3)

翻译文法:

$$D \rightarrow \{INI()\} id \{NAME()\} : T \{ENT()\} D'$$

$$D' \rightarrow ; id \{NAME()\} : T \{ENT()\} D' \mid \varepsilon$$

$$T \rightarrow interger \{TYP(i)\} \mid$$

$$\rightarrow real \{TYP(r)\} \mid$$

$$\rightarrow array[num] \text{ of } T \{TYP(a)\} \mid$$

$$\rightarrow \uparrow T \{TYP(\uparrow T)\}$$

INI()——初始化函数，在文法的开始，只能有一次。D 的产生式完成的是初始化的操作，而 D' 不需要，这也是文法变换的一个目的，把第一个变换提取出来。

$$(1) offset := 0;$$

NAME()——标识符，名字处理函数，得到标识符对应的地址，存在该属性值；

$$(1) id.name := entry(id);$$

ENT()——填写符号表函数，根据名字、类型、偏移，把变量存入活动记录相应位置，同时 offset 要加上变量的长度；

```
(1) enter(id.name, T.type, offset);
(2) offset := offset + T.width;
```

TYP(x)——标识符类型处理函数，用 x 表示 T 的类型，x 的 width 表示 T 的 width；

```
T.type := x; T.width := x.width;
```

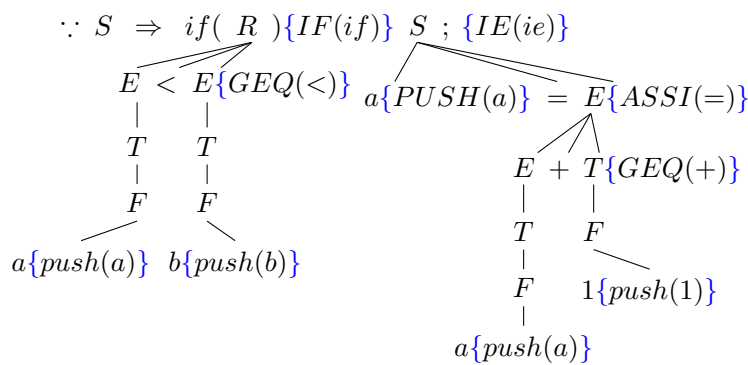
NUM()——标识符类型处理函数，将变量的值存到对应的 val 属性；

```
num.val := val(num);
```

翻译文法应用验证示例

例 7.13 已知条件语句 if(a<0) a=a+1;

(1) 通过语法制导分析（属性推导树），写出动作序列：



所以 if(a<0) a=a+1 四元式翻译的动作序列依次是：push(a), push(b), GEQ(<), IF(if), PUSH(a), push(a), push(1), GEQ(+), ASSI(=), IE(ie)

(2) 根据上述条件语句四元式翻译的动作序列，if(a<0) a=a+1 的四元式生成过程：

动作序列	SEM[m]	QT[q]
PUSH(a) →	a	
PUSH(b) →	a b	
GEQ(<) →	a b	(1) (< a b t1)
IF(if) →	t1	(2) (if t1 __)
PUSH(a) →	a	
PUSH(a) →	a a	
PUSH(1) →	a a 1	
GEQ(+) →	a a 1	(3) (+ a 1 t2)
ASSI(=) →	a t2	(4) (:= t2 _ a)
IE(ie) →		(3) (ie _ _ _)

7.4 中间代码翻译的实现

语法制导翻译的结构，可描述为：

扩展的语法分析器 \oplus 翻译文法

语法分析器用文法对序列进行分析。翻译文法的设计，即在文法中加入语义动作，执行语法分析的过程同时执行翻译的过程。

1、自顶向下的翻译文法的要求：

- (1) 源文法应满足自顶向下分析要求（如 LL(1) 文法）；
- (2) 属性是自顶向下可求值的（属性计算不发生冲突）；
- (3) 动作符号可插入到产生式右部任何位置。

2、自底向上翻译文法的要求：

- (1) 源文法应满足自底向上分析要求（如 LR() 文法）；
- (2) 属性是自底向上可求值的（属性计算不发生冲突）；
- (3) 动作符号只能位于产生式的最右端。

7.4.1 递归子程序翻译法

例 7.14 算术表达式四元式翻译器的设计 1：

1、设置

语义栈：SEM[m]——暂存运算对象的属性值。

四元式区：QT[q]——四元式生成结果的存储区。

2、翻译文法设计

G1(E):

$$E \rightarrow T\{+T\{GEQ(+)} \mid -T\{GEQ(-)}\}$$

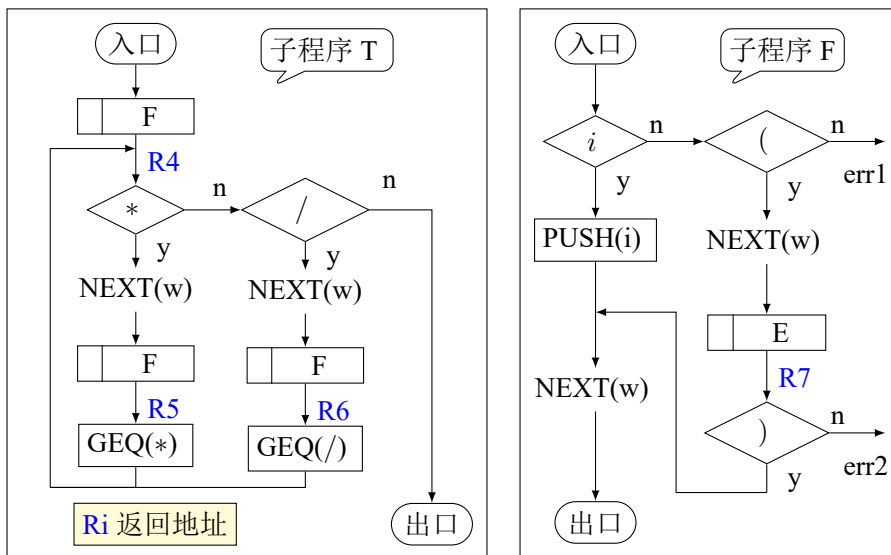
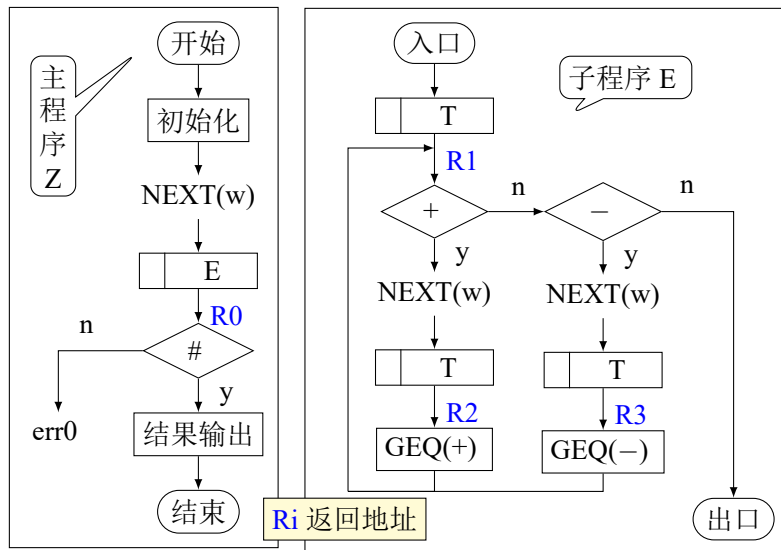
$$T \rightarrow F\{*F\{GEQ(*)} \mid /T\{GEQ(/)}\}$$

$$F \rightarrow i\{PUSH(i)} \mid (E)$$

3、递归子程序的扩展

用递归子程序来分析文法，求四元式序列。要求：

- ①动作符号在哪，就在哪执行之；
 - ②主控程序 (Z → E) 功能：初始化；结果输出。
- 算术表达式翻译文法的递归子程序：



算术表达式四元式翻译过程：

递归子程序调用算法为：入口时，把返回地址压入返回地址栈并进入；出口时，把返回地址弹出返回地址栈并返回。

设待翻译的表达式为 $a*(b/c)\#$ ，执行递归子程序的过程如下：

首先，压入 Z，然后读取 $w=a$ ，调用子程序 E，进入返回地址 R0。在子程序 E 中，先压入 T，而在子程序中，又读入 F，返回 R4。在子程序 F，先判断是不是 i，是 i 则执行 PUSH(i)，把 i 压入到语义栈，然后再读取 w，跳出子程序 F，回到 R4。接下来的过程不再赘述，最终分析结束的时候，会生成两个四元式，如表所示。

递归子程序栈	返回地址栈	w	SEM[m]	QT[q]
Z E T F	R ₀ R ₁ (R ₄)	a	a	
Z E T	R ₀ R ₁	*	a	
Z E T (F)	R ₀ R ₁ R ₅	(a	
Z E T F E T F	R ₀ R ₁ R ₅ R ₇ R ₁ (R ₄)	b	a b	
Z E T F E (T)	R ₀ R ₁ R ₅ R ₇ R ₁	\	a b	
Z E T F E T F	R ₀ R ₁ R ₅ R ₇ R ₁ (R ₆)	c	a b c	
Z E T F E (T)	R ₀ R ₁ R ₅ R ₇ (R ₁))	a b c	(1)/(b,c,t1)
Z E T F (E)	R ₀ R ₁ R ₅ (R ₇))	a t1	
Z E T (F)	R ₀ R ₁ R ₅)	a t1	
Z E T	R ₀ (R ₁)	#	a t1	(2)(*a,t1,t2)
Z E	(R ₀)	#	t2	
Z		#	t2	
Ok!				

7.4.2 LL(1) 翻译法

例 7.15 算术表达式四元式翻译器的设计 2:

1. 设置语法栈: SYN[n];
语义栈: SEM[m];
四元式区: QT[q];
2. 翻译文法设计将例7.14的文法改造成右递归的形式

右递归改造文法

$$E \rightarrow TE' \quad (1)$$

$$E' \rightarrow +T\{GEQ(+)\} E' \quad (2) \mid -T\{GEQ(-)\} E' \quad (3) \mid \varepsilon \quad (4)$$

$$T \rightarrow FT' \quad (5)$$

$$T' \rightarrow *F\{GEQ(*)\} T' \quad (6) \mid /F\{GEQ(/)\} T' \quad (7) \mid \varepsilon \quad (8)$$

$$F \rightarrow i\{PUSH(i)\} \quad (9) \mid (E) \quad (10)$$

3、LL(1) 分析器的扩展

- (1) 当产生式（逆序）压栈是，动作符号也不例外；
- (2) 当动作符号位于栈顶时，执行之。

算术表达式四元式翻译过程：

首先得到 LL(1) 分析表：

∖	i	+	-	*	/	()	#
E	①					①		
E'		②	③				④	④
T	⑤					⑤		
T'		⑧	⑧	⑥	⑦		⑧	⑧
F	⑨					⑩		

图 7.1: LL(1) 分析表

设待翻译的表达式为 $a+b*c\#$ ，分析的过程如下：

栈底是 # 和 E，x 是栈顶符号 E，w 是当前要处理的符号 a。E 看到 a，是产生式①，所以 push(E)，把 E 产生式的右部逆序压栈，把 T 放到 x。T 遇到 a，把产生式⑤逆序压栈，此时栈顶符号是 F，F 遇到 a，把产生式⑨逆序压栈。压栈时，把 PUSH(i) 当成一个元素，即把 PUSH(a)、a 依次压栈。栈顶是 a，w 也是 a，做匹配操作，读取下一个符号 +。栈顶符号是 PUSH(a)，执行该语义动作，把 a 压入语义栈 SEM[m]，SYN[n] 的栈顶元素变为 T'。把 T' 存到 x，T' 遇到 +，把产生式⑧逆序压栈，产生式⑧是空，所以不用压栈。接着是 E' 遇到 +，产生式②逆序压栈，把栈顶符号 + 存到 x，匹配 +。读取下一个符号 b，T 遇到 b，逆序压栈产生式⑤。接下来的过程不再赘述，最终 # 和 # 匹配，生成两个四元式，如下表。

SYN[n]	x	w	操作	SEM[m]	QT[q]
# E	E	a	PUSH① ^R		
#E' T	T	a	PUSH⑤ ^R		
#E' T' F	F	a	PUSH⑨ ^R		
#E' T' PUSH(a)	a	a	NEXT(W)		
#E' T' PUSH(a) →		+		a	
#E' T	T'	+	PUSH⑧ ^R	a	
# E	E'	+	PUSH② ^R	a	
#E' GEQ(+) T	+	+	NEXT(W)	a	
#E' GEQ(+) T	T	b	PUSH⑤ ^R	a	
#E' GEQ(+) T' F	F	b	PUSH⑨ ^R	a	

SYN[n]	x	w	操作	SEM[m]	QT[q]
#E' GEQ(+) T' PUSH(b)	b	b	NEXT(w)	a	
#E' GEQ(+) T' PUSH(b) →		*		ab	
#E' GEQ(+) T'	T'	*	PUSH⑥ ^R	ab	
#E' GEQ(+) T' GEQ(*) F	*	*	NEXT(w)	ab	
#E' GEQ(+) T' GEQ(*) F	F	c	PUSH⑨ ^R	ab	
#E' GEQ(+) T' GEQ(*)					
PUSH(c)	c	c	NEXT(w)	ab	
#E' GEQ(+) T' GEQ(*)					
PUSH(c) →		#		abc	
#E' GEQ(+) T' GEQ(*) →		#		a b c	(1)(*b,c,t1)
#E' GEQ(+) T'	T'	#		at1	
#E' GEQ(+) →		#		a t1	(2)(+a,t1,t2)
# E'	E'	#		t2	
#		#	ok		

这个过程和语法分析是类似的，不同的是加入了一些语义动作，当栈顶是语义动作时，就执行相应的动作。

7.4.3 LR() 翻译法

例 7.16 算术表达式四元式翻译器的设计 3:

1、设置

语法栈: SYN[n];

语义栈：SEM[m];

四元式区：QT[q];

2、

翻译文法设计（带有状态编码）

$$\begin{aligned}
 Z &\rightarrow E_1 (0) \\
 E &\rightarrow E_2 +_3 T_4\{GEQ(+)\} (1) | T_5 (2) \\
 T &\rightarrow T_6 *_7 F_8\{GEQ(*)\} (3) | F_9 (4) \\
 F &\rightarrow i_{10}\{PUSH(i)\} (5) | (_{11} E_{12})_{13} (6)
 \end{aligned}$$

注意，自底向上翻译文法的语义动作只能在产生式的最右端。

3、LR() 分析表的扩展

LR() 分析表中的 r(i) 执行下述两种操作：

- ①首先执行动作符号（翻译函数）；
- ②然后执行归约操作（按产生式 i 归约）。

SLR(1) 分析表的构造：

\	i	+	*	()	#	E	T	F
0	i10			(11			E1,2	T5,6	F9
1,2		+3				OK			
3	i10			(11				T4,6	F9
4,6		r(1)	*7		r(1)	r(1)			
5,6		r(2)	*7		r(2)	r(2)			
7	i10			(11					F8
8	r(3)	r(3)	r(3)	r(3)	r(3)				
9	r(4)	r(4)	r(4)	r(4)	r(4)				
10	r(5)	r(5)	r(5)	r(5)	r(5)				
11	i10			(11			E1,2	T5,6	F9
12,2		+3)13				
13	r(6)	r(6)	r(6)	r(6)	r(6)				

算术表达式四元式翻译过程：设待翻译的表达式为 $a*b+c\#$ ，分析过程如下：首先把 0 状态压栈，0 状态看到 a 到 10 状态，压入 a10。10 状态看到 #，执行 r(5)，归约产生式 (5)。归约的时候，先执行 PUSH(a)，把 a 压入语义栈，再归约 a，压入 F9。9 状态看到 *，归约产生式 (4)，归约成 T。0 状态看到 T，是 T5,6 状态，5,6 状态是一起的，是确定化之后得到的。T5,6 状态看到 * 到 7 状态，随后的过程与前面类似，这里不再赘述，最终得到整个分析过程。

SYN[n]	w	动作符号	SEM[m]	QT[q]	
# ₀	0	a			
# ₀ <u>a</u> ₁₀	10	*	PUSH(a)	a	
# ₀ <u>F</u> ₉	9	*		a	
# ₀ <u>T</u> _{5,6}	5,6	*		a	
# ₀ <u>T</u> _{5,6} * ₇	7	b		a	
# ₀ <u>T</u> _{5,6} * ₇ <u>b</u> ₁₀	10	+	PUSH(b)	ab	
# ₀ <u>T</u> _{5,6} * ₇ <u>F</u> ₈	8	+	GEQ(*)	a b	(1) (* a b t1)
# ₀ <u>T</u> _{5,6}	5,6	+		t1	
# ₀ <u>E</u> _{1,2}	1,2	+		t1	
# ₀ <u>E</u> _{1,2} + ₃	3	c		t1	
# ₀ <u>E</u> _{1,2} + ₃ <u>c</u> ₁₀	10	#	PUSH(c)	t1c	
# ₀ <u>E</u> _{1,2} + ₃ <u>F</u> ₉	9	#		t1c	
# ₀ <u>E</u> _{1,2} + ₃ <u>T</u> _{4,6}	4,6	#	GEQ(+)	t1 c	(2)(+ t1 c t2)
# ₀ <u>E</u> _{1,2}	1,2	#			结束

7.4.4 算符优先翻译法

例 7.17 算术表达式四元式翻译器的设计 4:

1、设置

语法栈: SYN[n];

语义栈: SEM[m];

四元式区: QT[q];

2、翻译文法设计

翻译文法:

$$E \rightarrow E + T\{GEQ(+)\} | T$$

$$T \rightarrow T * F\{GEQ(*)\} | F$$

$$F \rightarrow i\{PUSH(i)\} | E$$

1、算符优先分析器的扩展

(1) 归约时, 先执行语义动作, 后归约;

(2) 归约时, 只要产生式右部的终结符排列和栈顶 $\langle \cdot \dots \cdot \rangle$ 中的终结符匹配上即可;

(3) 归约后, 得到的非终结符不必入栈。

(4) 算术表达式四元式翻译过程:

\	+	*	i	()	#
+	>	<	<	<	>	>
*	>	>	<	<	>	>
i	>	>			>	>
(<	<	<	<	=	>
)	>	>			>	>
#	<	<	<	<	<	OK

图 7.2: 优先级图

设待翻译的表达式为 $a+b*c\#$ ，分析过程为：首先压入 $\#$ ，如果关系是 $<$ ，就移进，如果是 $>$ 就归约，用归约，先执行 $PUSH(a)$ ，再归约成 F 。剩下过程不再赘述，只需注意归约时要加上语义动作。

SYN[n]	关系	W	操作	SEM[m]	QT[q]
#	<	a	移进, 读		
# < a	>	+	规约, 不读	a	
#	<	b	移进, 读		
# < +	<	b	移进, 读		
# < + < b	>	*	规约, 不读	a b	
# < +	<	*	移进, 读		
# < + < *	<	c	移进, 读		
# < + < * < c	>	#	规约, 不读	a b c	
# < + < *	>	#	规约, 不读	a t1	(* , b, c, t1)
# < *	>	#	规约, 不读	t2	(+ , a, t1, t2)
#		#	OK		

7.4.5 翻译文法的变换问题

自底向上中间代码翻译，要求翻译文法的动作符号必须位于产生式的最右端。不满足此条件的属性翻译文法，需要通过文法变换来解决。

1. 赋值语句的属性翻译文法：

$G(S)$:

$$S \rightarrow vPUSH(v) = EASSI(=);$$

改造文法，令 $Sa \rightarrow vPUSH(v)$,

则有：

$G'(S)$:

$$S \rightarrow Sa = EASSI(=);$$

$$S \rightarrow vPUSH(v);$$

2. 标号、转向语句属性翻译文法：

G(S):

$$S \rightarrow i \text{ PUSH}(i) : \text{LABEL}(i) S;$$

$$S \rightarrow \text{goto } i \text{ GOTO}(i);$$

令 $S_i \rightarrow i \text{ PUSH}(i)$

$SI \rightarrow S_i : \text{LABEL}(i)$ 则有:

G'(S):

$$S \rightarrow SI S;$$

$$SI \rightarrow S_i : \text{LABEL}(i)$$

$$S_i \rightarrow i \text{ PUSH}(i)$$

3. 分支语句的翻译文法

G(S):

$$S \rightarrow \text{if}(R) \text{ IF}(\text{if})S;$$

$$[\text{else } \text{El}(\text{el})S] \text{ IE}(\text{ie})$$

令 $S_{if} \rightarrow \text{if}(R) \text{ IF}(\text{if})$

则有:

G'(S):

$$S \rightarrow S_{if} S; [\text{Sel } S] \text{ IE}(\text{ie})$$

$$S \rightarrow \text{if}(R) \text{ IF}(\text{if})$$

4. 循环语句的翻译文法

G(S):

$$S \rightarrow \text{whileWH}() (R) \text{ DO}(\text{do}) S \text{ WE}(\text{we})$$

令 $S_{wh} \rightarrow \text{whileWH}() S_{do} \rightarrow (R) \text{ DO}(\text{do})$

则有:

G'(S):

$$S \rightarrow S_{wh} S_{do} \text{ SWE}(\text{we})$$

$$S_{wh} \rightarrow \text{whileWH}()$$

第8章 优化处理

在实际开发编译器时，对于前端技术，近年来一直比较稳定，以语法制导、形式语言自动机为基础，更多强调的是语法制导技术的翻译与分析，到中间代码生成为止，编译器基础的算法已经介绍完毕。对于后端技术，发展十分迅猛，是现代编译器工程实现中相当重要的部分。后端是一个从中间代码到目标代码的生成系统，由两部分构成：一是**优化处理**，给定中间代码，加快代码执行效率，减小空间占用；二是**目标代码生成**，在优化处理的基础上，生成精简高效的目标代码。前端决定了编译器能否实现，后端则决定了编译器是否可用。

本章主要介绍优化处理，优化是编译器中一个重要环节，优化的设计在整个编译器设计过程中占据相当大的比重，目的是**产生更高效的目标代码**。优化处理中包括大量数据结构、算法的优化，本书不做详细展开，本章主要介绍优化的基本思想和几种优化方法。

8.1 优化的分类

优化处理可以在不同层面上进行，可分为在源代码或中间代码级上进行的与机器无关的优化，以及在目标代码级上进行的与机器有关的优化。

8.1.1 与机器无关的优化

不采用与目标机相关的知识，如果优化在各个机器上都可以进行，就是与机器无关的优化。由于每一条中间代码指令都是一个单操作，在中间代码级上优化比较容易。在源代码级的优化相对困难，可以在编写程序时实现。

- **全局优化**：针对整个源程序，能达到更好的优化效果，实现难度更大。
- **局部优化**：除全局优化以外，处理局部的函数和程序块，更易实现。

8.1.2 与机器有关的优化

需要考虑目标机的性质。目标代码生成是从四元式生成目标代码，如果不考虑优化，是一一对一的映射过程，是有模板的映射，而考虑优化则需要处理大量问题。

- **寄存器分配优化**：现代 CPU 的设计，包括操作系统和编译器，都要考虑目标机关于存储器

的分配。寄存器访问速度快，运算效率高，但资源有限，设计相关算法合理分配寄存器能够大大优化效率。

- **消除无用代码优化：**例如一些无用跳转，可以依据目标机进行。

以上这两类优化处理方法中，主要介绍局部优化（第八章）和寄存器分配优化（第九章）。

8.2 常见的几种局部优化方法

8.2.1 常值表达式节省（常数合并）

如： $a = 5 + 3; b = a + 1; \dots$

等号右边的表达式 $5 + 3$ 和 $a + 1$ 都是常值表达式， a 和 b 的值不需要在程序运行时进行计算，编译器会将其优化为 $a = 8; b = 9;$ 的形式，这样的优化被称为**常值表达式节省**。这种方法看似简单，却十分关键，节省了大量无用计算，若没有这种技术，现代计算机慢 10 倍，编写的程序中需要进行大量常数合并，是一种常见的优化形式。

注：若 $a = 5 + 3; \dots; a = x \dots; a = a + 1; a = 5 + 3$ 中的 a 是常数，后 a 经过赋值， $a = a + 1$ 中的 a 不再是常数， $a + 1$ 不是常值表达式。

8.2.2 公共表达式节省（删除多余运算）

如： $a = b * d + 1; e = b * d - 2; \dots$

等号右边的两个表达式中都有 $b * d$ ，且 b 和 d 的值在使用前后没有被修改，则 $b * d$ 是公共表达式，不用重复计算，可以优化为 $t = b * d; a = t + 1; e = t - 2;$ 的形式，减少了一次乘法，增加了一次赋值，乘法的计算代价高于赋值，这样的形式达到了优化的目的。

注：若 $b = b * d + 1; e = b * d - 2;$ 就不能再使用这样的方法，因为第一个赋值表达式中，修改了 b 的值，公共表达式要求表达式中的变量不变，所以此处的 $b * d$ 不是公共表达式。

8.2.3 删除无用赋值

如： $a = b + c; x = d - e; y = b; a = e - h/5;$

看似没有问题，仔细观察在 a 的两次赋值之间，没有对 a 的应用，则前一次对 a 的赋值 $a = b + c$ 为无用赋值。可以将第一个式子删除，优化为 $x = d - e; y = b; a = e - h/5;$ 的形式。

8.2.4 不变表达式外提（循环优化之一：把循环不变运算提到循环外）

如： $i = 1; \text{while}(i < 100) \ x = (k + a)/i; i ++;$

假设循环体中 k 和 a 的值都没有改变，则每一次循环中 $k + a$ 的值都不变，将其称为循环不变表达式。如果每次循环中，都计算一遍 $k + a$ ，循环次数非常多时，计算代价将非常高。因

此，可以优化为 $i = 1; t = k + a; \text{while}(i < 100) \ x = t/i; \dots; i ++;$ 的形式。

不变表达式外提也是一种常见的优化方法。如何通过算法实现找出不变表达式并外提，也是一个需要解决的问题。

8.2.5 消减运算强度（循环优化之二：把运算强度大的运算换算成强度小的运算）

如： $i = 1; \text{while}(i < 100) \ t = 4 * i; b = a \uparrow 2; i ++;$

幂运算强度大于乘除取模运算，乘除取模运算强度大于加减赋值运算。 $a \uparrow 2$ 为 a 的平方，循环变量 i 每次加 1， $4 * i$ 的结果为 4, 8, 12, ... 的等差数列，每次乘法的效果和每次加 4 的效果一致。因此，可以优化为 $i = 1; t = 4 * i; \text{while}(i < 100) \ t = t + 4; b = a * a; \dots; i ++;$ 的形式。

8.3 局部优化算法探讨

通过以上例子，我们对优化有了一定感性的认识，接下来进一步系统地进行探讨。在编译器设计中，局部优化算法是以**基本块**为单位进行的，**基本块**也是目标代码生成的基本单位。

定义 8.1 基本块

程序中一段顺序执行的语句序列，其中只有一个入口和一个出口。

根据定义中的“顺序执行”、“一个入口，一个出口”，自然联想到，基本块的划分与条件判断以及跳转语句有关。执行一个基本块内的程序，只能从第一条顺序执行到最后一条，不存在其他入口或出口。满足这样要求的程序段称为一个独立的基本块。根据定义，只要找到基本块的入口和出口，就能找到基本块。

8.3.1 基本块划分算法

1. 找出基本块的入口语句：（以下为两种判断条件，满足其一即可）

- 程序的第一个语句或转向语句转移到的语句，四元式中的转向语句包括 goto、then、else (无条件跳)、do (循环体中)。
- 紧跟在转向语句后面的语句。

2. 对每一入口语句，构造其所属的基本块：

- 从该入口语句到另一入口语句之间的语句序列；
- 从该入口语句到另一转移语句（或停止语句）之间的语句序列。

例 8.1 条件语句四元式如下，给出基本块划分：


```

gt(E)
(then, res(E), __, __)
gt(S1)
(else, __, __, __)
gt(S2)
(iffend, __, __, __)

```

解：语句，无外乎逻辑以及逻辑下的具体操作。其中 $gt(S_1)$ 和 $gt(S_2)$ 均为完整语义块，若不考虑嵌套条件语句， $gt(S_1)$ 和 $gt(S_2)$ 均为顺序执行。因此给出的四元式中，包含 $gt(E)$ ($then, res(E), _, _$), $gt(S_1)$ ($else, _, _, _$) 和 $gt(S_2)$ ($iffend, _, _, _$) 三个基本块，入口语句为第一个语句或转移语句的后一句，一直到另一转移语句结束。

例 8.2 设有源程序片段，给出基本块划分：

```

x = 1;
a : r = x * 5;
if(x < 10)
x = x + 1;
goto a;
r = 0;

```

解： $goto$ 语句在编程中不建议大家使用，但有助于理解编译器。第一步，写出对应的四元式序列如图8.1，其中的赋值语句是以单操作形式，最左侧为结果单元。 $goto$ 语句对应的四元式，定义操作符为 gt ，表示直接跳转。 lb 四元式表示跳转的目标，因此将它作为基本块的开始。 if 四元式等价于前面介绍的 ($then, res(E), _, _$)。 ie 四元式是转向语句转移到的语句，作为基本块的开始。

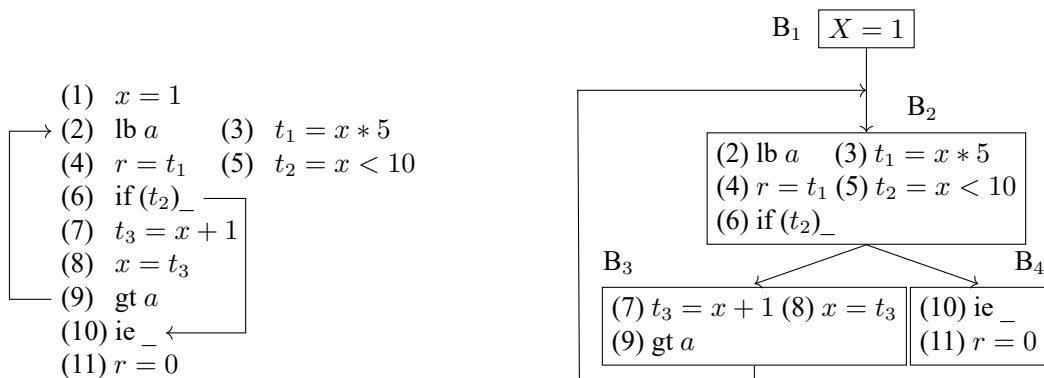


图 8.1: 对应的四元式序列

图 8.2: 以基本块为节点的程序流图

第二步，划分基本块。根据算法，找出所有入口，两入口之间的部分即为基本块。观察四元式序列，第 (1) 个四元式是程序入口，是基本块的入口语句；第 (2) 个四元式是第 (9) 个四元式跳转到的语句，可以作为基本块的入口；第 (3) 到第 (5) 个四元式，根据定义，都不是基本块入口；第 (6) 个四元式，是跳转语句，不是基本块入口，但跳转语句的下一条，即第 (7) 个四元

式是基本块入口；第(8)个四元式不是基本块入口；第(9)个四元式是跳转语句，不是基本块入口，下一个四元式(10)是基本块入口。从而划分得到如下四个基本块，如图8.2。

8.3.2 局部优化示例

例 8.3 基本块内设有语句片段：

$$B = 5; A = 2 * 3.14 / (R + r);$$

$$B = 2 * 3.14 / (R + r) * (R - r);$$

解：这是一个基本块，优化后结果如下。第一个式子删除对 B 的无用赋值；第二个式子可进行常值表达式节省， $2 * 3.14$ 可优化为 6.28；第二个式子和第三个式子中都出现了 $2 * 3.14 / (R + r)$ ，可以进行公共表达式节省，第二个式子中替换为 A。

$$A = 6.28 / (R + r);$$

$$B = A * (R - r);$$

上述是人工优化的思考过程，下面系统地进行优化。

- 第一步，根据原语句片段，生成图8.3左侧四元式序列。
 - 第二步，逐个观察四元式。
 - 第(1)个四元式，没有可优化的。
 - 第(2)个四元式，是常值表达式，可以优化为 $t_1 = 6.28$ ，保存该值，不生成四元式。
 - 第(3)个四元式， $R + r$ 必须计算，不能优化。
 - 第(4)个四元式，将 t_1 替换为 6.28。
 - 第(5)个四元式不能优化。
 - 第(6)个四元式，与第(2)个四元式相同，优化为 $t_4 = 6.28$ 并保存。
 - 第(7)个四元式，与第(3)个四元式运算、运算对象相同，是公共表达式，保留 $t_5 = t_2$ 的关系。
 - 第(8)个四元式， t_4 用 6.28 替代， t_5 用 t_2 替代，与第(4)个四元式有公共表达式，保留 $t_6 = t_3$ 的关系。
 - 第(9)个四元式不能优化。
 - 第(10)个四元式中的 t_6 用 t_3 代替。
 - 第(11)个四元式赋值给 B。
- 得到结果如图8.3右侧，相较原四元式序列，省去了四条。
- 第三步：继续观察四元式。B 的两次赋值之间（第(2)到第(10)个四元式中），B 没有引用，因此删除第(1)个四元式中的无用赋值 B。

最终得到优化后 6 个四元式序列，优化过程如图8.3所示。

根据例8.3总结优化的基本算法设计：

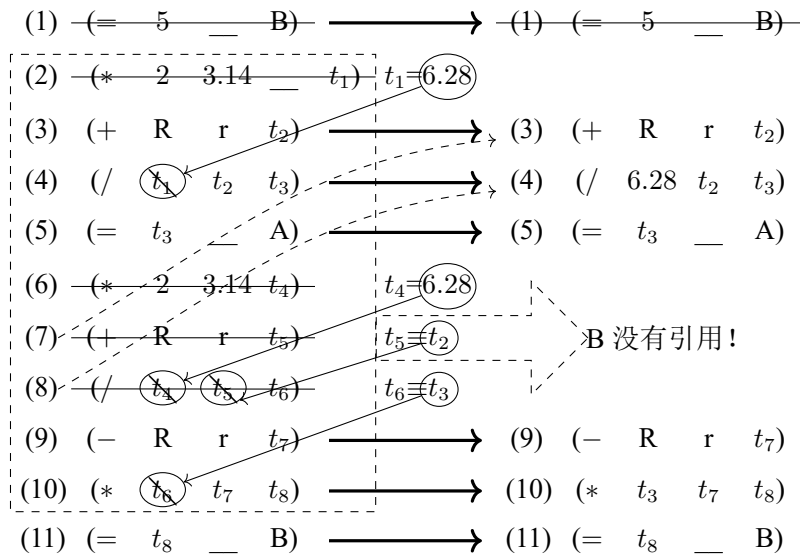


图 8.3: 四元式序列的优化过程

1. 对于常值表达式节省 (例中 (2)(6))
 - (1) 先进行常值计算;
 - (2) 取常值的变量以常值代之;
 2. 公共表达式节省 (例中 (7)(8))
 - (1) 找公共表达式, 建立结果变量等价关系;
 - (2) 等价变量以老变量代替新变量;
 3. 删除无用赋值 (例中 (1))
 - (1) 确认一个变量两个赋值点间无引用点;
 - (2) 则前一赋值点为无用赋值;
-

接下来, 进一步构造算法解决这些问题。

8.4 基于 DAG 的局部优化方法

DAG (Directed Acyclic Graph) 是指无环有向图, 或称有向无环图 (如图8.4), 对于计算机相关的系统开发及研究十分重要, 这里用来对基本块内的四元式序列进行优化。

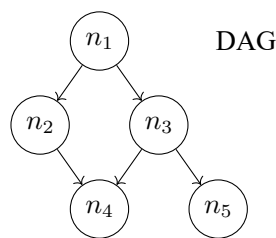


图 8.4: 有向无环图

8.4.1 四元式序列的 DAG 表示

如何把线性的四元式映射到图结构呢？下面图8.5给出 DAG 的结点内容及其表达。图中包含结点，因为是有向图，又包含前驱和后继。四元式中包含一个运算符，两个运算对象，一个结果单元。有向无环图的要素如何与四元式的要素对应起来，这是要研究的第一个问题。

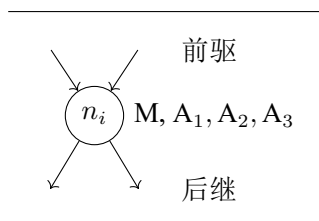


图 8.5: 结点的前驱和后继

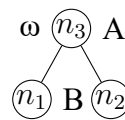
1. DAG 的结点内容及其表示

- n_i : 结点的编码;
- ω : 运算符, 置于结点左侧, 将该结点称为 ω 运算结点, 代表 ω 运算。 ω 运算的运算对象为其后继节点, 结果单元标记在结点右侧;
- M: 主标记, 使用时作为代表优先被使用, 如果是叶子结点, 表示变量和常数的初值;
- A_i : 附加标记, 为运算结果变量, 为便于描述等价关系, 可设置多个, $i = 1, 2, 3, \dots$ 。如例8.3中 t_2 和 t_5 都表示 $R + r$ 的结果, 可以 t_2 为主标记, t_5 为附加标记, 从而便于描述等价关系。

右图中值为 ω 运算结果超过一个变量, M, A_1, A_2, A_3 都取 ω 运算结果值, 它们都应放在结点右侧。在 M, A_1, A_2, A_3 中选择 M 作为主标记, 主标记之外的运算结果变量称为附加标记, A_1, A_2, A_3 均与 M 等价。

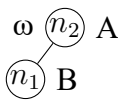
2. 四元式的 DAG 表示

- 赋值: 赋值四元式 ($=B _ A$) 或 $A=B$, 将 B 的值赋给 A, 表示 A 和 B 是等价的。DAG 表示为 $(n_i)B|A$, 省去了赋值运算符, B 为主标记, A 为附加标记。
- 双目运算: 双目运算四元式 ($\omega B C A$) 或 $A = B\omega C$, 将 $B\omega C$ 赋值给 A, 其中 ω 可以



是算术运算、关系运算、逻辑运算等。DAG 表示为 $(n_1)B(n_2)C$, ω 运算符放在结点 n_3 左侧, 结果单元 A 放在结点 n_3 右侧, 两个运算对象 B 和 C 放在下面结点 n_1 和 n_2 的右侧, 作为 n_3 的后继结点, 通过 ω 运算赋值给 A。

- 单目运算: 有单目运算四元式 ($\omega B _ A$) 或 $A = \omega B$ 。与双目运算类似, 从原来的二



叉变成“一叉”, DAG 表示为 $(n_1)B$ 。结点 n_1 表示 B, 通过 ω 运算赋值给 A。

- **下标变量赋值运算**：有下标变量赋值运算四元式 $A=B[C]$ ，DAG 表示为 。A 有两个后继 B 和 C，[] 表示按变量 C 为偏移量，取变量 B 对应数组的元素，结果

用 A 表示。将变量赋值给数组元素 $B[C]=D$ ，DAG 表示为 。这种情况比较特殊，[] 操作对象不是 B、C、D 中任何一个，而是以 B 指针开始，以 C 对应的变量为偏移，并以 D 赋值，因此结点 n_4 右侧没有出现结果单元，而是用这个结构表示赋值的过程。

- **转向**：有转向四元式 $(\omega [B] _ A)$ ，B 为可选单元，DAG 表示如下：第一个表示无条件跳转到 A ，第二个根据 B 进行跳转

有了这样的表示形式，就能够完成对四元式序列的 DAG 表示。

例 8.4 求下述语句片段的 DAG 表示：

$B = 5; A = 2 * 3.14 * (R + r); B = 2 * 3.14 * (R + r) / (R - r);$

解：

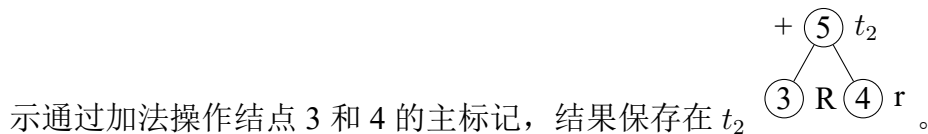
- 第一步，生成对应的四元式序列。
 - (1) $B = 5$ (2) $t_1 = 2 * 3.14$ (3) $t_2 = R + r$
 - (4) $t_3 = t_1 * t_2$ (5) $A = t_3$ (6) $t_4 = 2 * 3.14$
 - (7) $t_5 = R + r$ (8) $t_6 = t_4 * t_5$ (9) $t_7 = R - r$
 - (10) $t_8 = t_6 / t_7$ (11) $B = t_8$

- 第二步，逐个扫描四元式，构建优化 DAG 图。

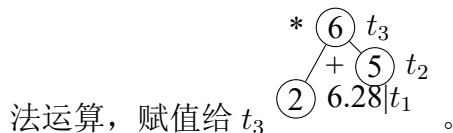
第 (1) 个赋值四元式，创建结点 1 ，主标记为 5，附加标记为 B，表示 $B = 5$ 。

第 (2) 个常值运算四元式，将 $2*3.14$ 化简为 6.28，创建结点 2 ，与结点 1 类似，主标记为 6.28，附加标记为 t_1 。

第 (3) 个四元式，R 和 r 在 DAG 中均未出现，创建结点 3、4 表示 R 和 r，再创建结点 5，表



第 (4) 个四元式， t_1 在结点 2， t_2 在结点 5，创建结点 6，操作结点 2 和 5 的主标记进行乘

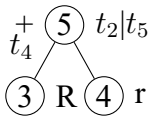


第 (5) 个四元式，将 t_3 赋值给 A， t_3 在 DAG 中已存在，因此 A 作为结点 6 的附加标

记 * ⑥ $t_3|A$ 。注意，因为主标记 t_3 为临时变量，而 A 为用户定义变量，这里要将 A 和 t_3 进行互换，表示为 * ⑥ $A|t_3$ 。

第 (6) 个四元式，将 6.28 赋值给 t_4 ，6.28 在 DAG 中已存在，因此 t_4 作为结点 2 的附加标记 ⑥ $6.28|t_1, t_4$ 。

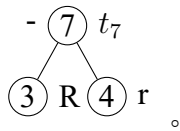
第 (7) 个四元式，将 $R + r$ 赋值给 t_5 ， R 和 r 以及 $R + r$ 的结果在 DAG 中也已存在，因此将



t_5 作为结点 5 的附加标记 ⑤ $t_2|t_5$ 。

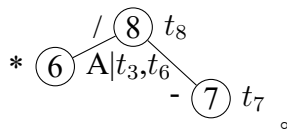
第 (8) 个四元式， t_4 在结点 2， t_5 在结点 5，结点 2 和结点 5 的乘法结果在 DAG 中已存在，因此将 t_6 作为结点 6 的附加标记 $A|t_3, t_6$ 。

第 (9) 个四元式， R 在结点 3， r 在结点 4， $R - r$ 的结果不存在，因此创建结点 7， $R - r$ 的



结果保存在 t_7 。

第 (10) 个四元式， t_6 在结点 6， t_7 在结点 7，创建结点 8，表示结点 6 的主标记和结点 7 的



主标记相除，结果保存在 t_8 。

第 (11) 个四元式，将 t_8 赋值给 B ，在结点 1 中删去 B ① $5|B$ ，将 B 作为结点 8 的附加标记 $B|t_8$ ，同样需要调换 B 和 t_8 的顺序 $B|t_8$ 。

得到最终的 DAG 表示如下：

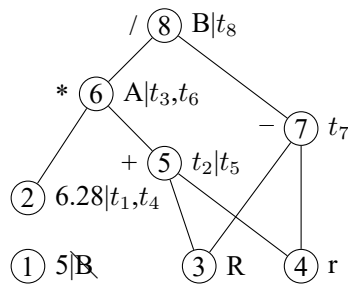


图 8.6: 最终的 DAG 表示

- 第三步，根据优化的 DAG 重组四元式。

结点 1，没有后继结点，即没有运算四元式，附加标记的 B 被删去，若存在附加标记，表示存在赋值四元式。

结点 2，没有后继结点，没有运算四元式，但存在等价关系，理论上生成 $t_1 = 6.28$ 和 $t_4 = 6.28$ 两个赋值四元式，但 t_1 和 t_4 是临时变量，是编译器定义的，与用户无关，临时变量不会在后续被使用，且 t_1 、 t_4 与 6.28 等价，因此可以用 6.28 代替 t_1 和 t_4 ，附加标记中放的都是临时变量，这时的赋值四元式可以省去。

结点 3 和结点 4，没有后继结点，没有附加标记，不需要进行操作。

结点 5，存在后继结点，生成运算四元式 $t_2 = R + r$ ，附加标记上的临时变量不用生成赋值语句，因此具有等价关系的 t_5 被省去。

结点 6，存在后继结点，生成运算四元式 $A = 6.28 * t_2$ ，附加标记 t_3 、 t_6 均为临时变量，省去。

结点 7，存在后继结点，生成运算四元式 $t_7 = R - r$ 。

结点 8，存在后继结点，生成运算四元式 $B = A/t_7$ 。

最终得到重组四元式如下，四元式数量从原来的 11 减少为 4。

$$(1) t_2 = R + r$$

$$(2) A = 6.28 * t_2$$

$$(3) t_7 = R - r$$

$$(4) B = A/t_7$$

介绍到这里，第二步中对于结点 6 和结点 8，为什么要进行位置交换？其实目的是减少四元式数量，优化中间代码。当结点右部存在若干个等价的值（包括常值、变量），在这若干个中选择一个作为主标记，优先关系为：常值（第一级）、用户定义变量（第二级）、临时变量（第三级）。如果附加标记中都是临时变量，赋值四元式可以省去；如果附加标记中存在用户定义变量，赋值四元式不可省去，因为在后续可能会被用户使用。结点 1 中 B 在附加标记上，且被重新赋值，原来的 B 可被主标记值替代，因此被省去，若 B 在主标记上，则不可省去。

8.4.2 基于 DAG 的局部优化算法

例 8.4 是一个典型的 DAG 优化算法实例，下面对基于 DAG 的局部优化算法进行系统总结。

1. 构造基本块内优化的 DAG

假设：① n_i 为已知结点号， n 为新结点号；② 访问各节点信息时，按结点号逆序进行；具体原因后面在例题中说明。

• 开始：

① DAG 置空；依次读取一四元式 $A = B \omega C$ 。

② 分别定义 B, C 结点，若已定义过，可以直接使用。

(1) 若赋值四元式 $A = B$

① 把 A 附加于 B 上，即 $\textcircled{n_1} \dots B \dots A$ ，表示 A 和 B 等价。

② 若 A 在 n_2 已定义过，且 A 在附加标记时，需要删去，即 $\textcircled{n_2} \dots | \dots A$ ；A 在主标记时，则无需删去。

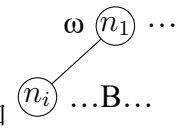
(2) 若常值表达式 $A = C_1 \omega C_2$ 或 $A = C$

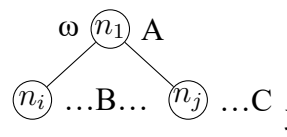
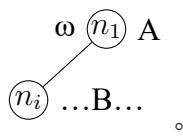
① 计算常值 $C = C_1 \omega C_2$ 。

②若 C 在 n_1 已定义过, 则 $\omega(n_1) C | \dots, A$, 如上例中结点 2 的主标记为 6.28, 表示 $t_4 = 6.28$ 时, 只需将 t_4 作为这个结点的附加标记; 否则申请新结点, 且 $\omega(n) C | A$, C 为主标记, A 为附加标记。

③若 A 在 n_2 已定义过, 且 A 在附加标记, 删去, 即 $\omega(n_2) \dots | \dots A \dots$ 。

(3) 若其他表达式 $A = B \omega C$ 或 $A = \omega B$

①若在 n_1 存在公共表达式 $B \omega C$ 或 ωB , 分别表示为 $\omega(n_1) \dots$  和 $\omega(n_1) \dots$ , 则把 A 附加在 n_1 上, 即 $\omega(n_1) \dots, A$ 。

②若不存在公共表达式, 则申请新结点 n,  或 。

③若 A 在 n_2 已定义过, 且 A 在附加标记, 则删除, 即 $\omega(n_2) \dots | \dots A \dots$; 若 A 为主标记, 则免删。

• 结束: 调整结果单元结点的主标记、附加标记的顺序。

★ 主标记优先顺序为: 常量, 非临时变量, 临时变量。

2. 根据基本块内优化的 DAG, 重组四元式

假设: ①临时变量的作用域是基本块内; ②非临时变量的作用域也可以是基本块外。

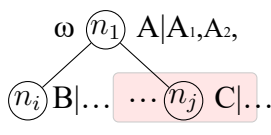
两条假设的目的是在最终优化后的代码中, 必须保证所有非临时变量的值是最新的, 临时变量的值若在后续没有使用, 就不需要赋值。

• 开始: 按结点编码顺序, 依次读取每一结点 n_1 信息:

(1) 若 n_1 为带有附加标记的叶结点, 即 $\omega(n_1) B | A_1, A_2, \dots$, 表示 A_i 与 B 等价

①若 A_i 为非临时变量, 则生成 $q_1: A_i = B (i = 1, 2, \dots)$ 。

②若 A_i 是临时变量, 则不需要生成。临时变量在基本块外不使用, 而生成四元式时使用的是主标记 B。

(2) 若 n_1 为带有附加标记的非叶结点, 即 。

①生成 $q_1: A = B \omega C$ 或 $A = \omega B$, 用主标记进行计算。

②若 A_i 为非临时变量, 则生成 $q_2: A_i = A (i = 1, 2, \dots)$ 。保证非临时变量在基本块出口时, 值是正确的。若 A_i 是临时变量, 则不需要生成四元式。

★ 注意: 以主标记参加运算。

例 8.5 求下述语句片段的 DAG 表示:

$$A = 2 * 3 + B / C$$

$$B = 2 * 3 + B / C$$

$$C = 2 * 3 + B / C$$

解:

- 第一步, 生成四元式序列 (不能“自动”优化)

(1) (*, 2, 3, t_1)

(2) (/, B, C, t_2)

(3) (+, t_1 , t_2 , t_3)

(4) (=, t_3 , __, A)

(5) (*, 2, 3, t_4)

(6) (/, B, C, t_5)

(7) (+, t_4 , t_5 , t_6)

(8) (=, t_6 , __, B)

(9) (*, 2, 3, t_7)

(10) (/, B, C, t_8)

(11) (+, t_7 , t_8 , t_9)

(12) (=, t_9 , __, C)

- 第二步, 构造优化的 DAG

依次读取四元式, 根据算法构造 DAG。需要注意, 访问各结点信息时, 按结点号逆序进行。例如读取第 10 个四元式后, 从结点 5 开始找最新的 B 和 C, 从而建立新的结点, 旧的 B 参与运算, 且是主标记, 要保留。前两个表达式中的 B/C 是公共表达式, 而第三个式子中不是, B 的值已经被更新。类似地, 读取各个四元式, 依据算法构建 DAG, 最终优化后的 DAG 图如图 8.7。

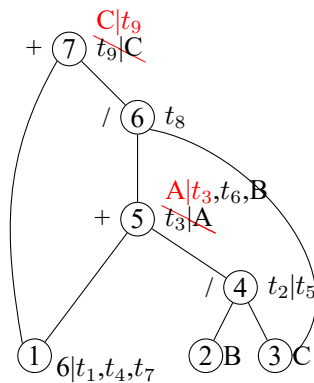


图 8.7: 最终优化后的 DAG 表示

- 第三步, 优化后的四元式序列。按结点编码顺序, 依次读取每一结点 n_1 信息, 生成相应四元式序列如下。

- (1) (/, B, C, t_2)
- (2) (+, 6, t_2 , A)
- (3) (=, A, __, B)
- (4) (/, A, C, t_8)
- (5) (+, 6, t_8 , C)

第9章 目标代码及其生成

什么是目标代码？目标代码在编译器中起到什么作用呢？

目标代码是指在机器上可以运行的代码，在本章中，可以视作汇编指令。目标代码生成，是编译的最后一个阶段，其功能可表示如下图9.1。

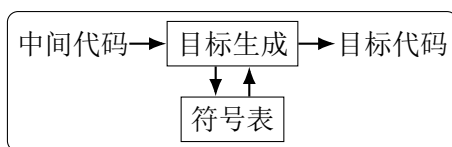


图 9.1: 目标代码功能

目标代码的来源是中间代码，也有从源程序直接生成目标代码的编译器（解释器），本书中介绍的是从中间代码进行转化的目标代码。为了简化教学内容，在生成目标代码时忽略了值单元地址，因此符号表甚少出现，但在实际生成编译器时，符号表对于生成目标代码十分重要。

其中，**中间代码**的种类是多样的，包括第7章中介绍的逆波兰式、三元式、四元式、语义树等；**目标代码**包括机器语言，汇编语言等，高级语言也可以作为目标代码，如：代码移植。为了便于讲解，本章中的目标代码是汇编语言的一个虚拟指令集，并不能直接运行，但原理是一致的；**符号表**包括变量的语义词典等，在生成目标代码时，会访问符号表来获取变量地址等信息。

9.1 目标代码生成的基本问题

9.1.1 目标代码选择

目标代码生成任务的目的是生成在机器上可以执行的指令。首先要考虑选择什么作为目标代码？

大多数编译程序不产生绝对地址的机器代码，而是以**汇编语言程序**作为输出，因为汇编代码与机器指令是相互对应的，以汇编语言程序作为输出可以使代码生成阶段变得容易。此外，**指令集的选择**以及**指令的执行速度问题**都是重要因素。为了使算法具有通用性，这里采用的是类似于8086的虚拟机及其指令系统，选择原因如下：

- (1) 直接生成机器代码，难度较大，且对于不同的架构，机器代码也有所区别；
- (2) 希望选择一种相对抽象但又能够理解的指令，汇编语言就具备这一属性；

(3) 不希望过于复杂, 复杂可能会影响程序执行效率, 即使有较好的可读性, 细节上也不便于进行操作。

★ 虚拟机及其指令系统:

1. 虚拟机寄存器: R_0, R_1, \dots, R_{n-1}

现代 CPU 的计算, 严格意义上来说, 都是在寄存器中完成, CPU 不能直接操作内存或外存上的数据, 只能操作寄存器中的内容, 因此内存或外存上的数据, 只有转到寄存器中, CPU 才能对其进行操作。

2. 虚拟机指令系统:

(1) 指令的基本形式: $op\ R_i, R_k/M$ 。

其中 op 表示操作码, 可以理解为指令的一种方式, 如加减乘除; M 表示变量的内存地址; R_i/R_k 表示寄存器地址。

“ $op\ R_i, R_k$ ” 的含义是 “ $R_i := (R_i)\ op\ (R_k)$ ”, 表示 R_i 寄存器里的内容, 和 R_k 寄存器里的内容, 通过 op 运算, 结果存在 R_i 寄存器中;

注意:

① R_i 寄存器里的内容参与运算, 且结果保存在 R_i 中, 即 R_i 中原来的值被结果覆盖了。

② 如果 op 为单目运算, 含义是 “ $R_i := op\ (R_k/M)$ ”, 表示通过 op 运算, 操作 R_k 寄存器里的内容或 M 地址指向的内容, 结果存在 R_i 寄存器中。

(2) 常用的指令:

① 取数据、存数据: 8086 中存、取指令均为 $move$, 这里为了做区分, 取数据用 LD , 存数据用 ST 。

$LD\ R_i, R_k/M$ —— 表示从 R_k/M 取到的数据存入 R_i 中, 写作 $R_i := (R_k/M)$

$ST\ R_i, R_k/M$ —— 表示将 R_i 中的内容存入 R_k/M 中, 写作 $R_k/M := (R_i)$

② 转向操作: FJ 表示假跳, TJ 表示真跳, JMP 表示无条件跳转。

$FJ\ R_i, M$ —— 表示判断 R_i 中内容是否为假, 为假, 则跳转到 M 地址指向的代码

$TJ\ R_i, M$ —— 表示判断 R_i 中内容是否为真, 为真, 则跳转到 M 地址指向的代码

$JMP\ _, M$ —— 表示无条件跳转到 M 地址指向的代码

③ 算术运算: 与 8086 指令类似, 包含 ADD (加)、 SUB (减)、 MUL (乘)、 DIV (除)

$ADD\ R_i, R_k/M$ —— $R_i := (R_i) + (R_k/M)$

$SUB\ R_i, R_k/M$ —— $R_i := (R_i) - (R_k/M)$

$MUL\ R_i, R_k/M$ —— $R_i := (R_i) * (R_k/M)$

$DIV\ R_i, R_k/M$ —— $R_i := (R_i)/(R_k/M)$

此外, 还有很多操作码 op , 在具体实现时, 可以自行定义。

④ 逻辑运算

$LT (<), GT (>), EQ (==), LE (<=), GE (>=), NE (!=)$

$AND (& \&), OR (||), NO (!)$

介绍完指令集，接下来介绍目标代码生成。四元式序列是不需要进一步解析的单操作，目标指令集确定时，从中间代码到目标代码，实际上是一个**模板**翻译的过程。

高级程序设计语言，即使程序非常复杂，但指令的种类是有限的，四元式也是如此，包括运算四元式、赋值四元式、跳转四元式等。

(1) **运算四元式**。若有四元式 (ω, a, b, t) ，表示 a 和 b 进行 ω 操作，结果单元放到 t 中。依据前面介绍的指令集，替换成目标代码时，运算对象放到寄存器中再进行运算。生成三条目标代码如下：

- ① LD R, a
- ② $\tilde{\omega}$ R, b
- ③ ST R, t

第一条表示将 a 取到寄存器 R 中；

第二条表示寄存器中内容与 b 发生 ω 运算，结果在 R 中；

第三条表示将 R 中内容保存到 t 中。对于加法，只需将 ω 替换为 ADD。显然，如果不考虑代码优化，这是一个模板替换的过程。

(2) **赋值四元式**。若有四元式 $(=, a, _, b)$ ，对应的目标代码如下。

- ① LD R, a
- ② ST R, b

因为指令集中没有内存单元存到内存单元的操作，所以①第一条先将 a 取到寄存器 R 中；②第二条将寄存器 R 的内容保存到 b 中。

例 9.1 据下列四元式翻译目标代码：

- (1) $(+ \ a \ b \ t_1)$
- (2) $(- \ t_1 \ d \ t_2)$

解法 1：若完全依据**模板**，将生成如下 6 条目标指令。

- ① LD R, a
- ② ADD R, b
- ③ ST R, t_1
- ④ LD R, t_1
- ⑤ SUB R, d
- ⑥ ST R, t_2

下一步考虑能否进行优化。这 6 条中，有两条做了无用功，第 3 条将寄存器 R 中内容保存到 t_1 ，紧接着第 4 条将 t_1 中的内容取到寄存器 R 中，这两条可以省去。相当于第 1 条四元式生成的目标代码中省去了模板的最后一条，第 2 条四元式生成的目标代码省去了模板中的第一条，不难发现运算四元式对应模板中第二条必不能省去，另两条可根据情况进行删减，如当一条四元式的结果单元，是下一条四元式的第一运算对象时。但不能简单地省去，否则若后续还有用到 t_1 的操作，将无法进行。如果不省 ST R, t_1 ， $a + b$ 的结果存入了 t_1 ，如果后面用到 t_1 ，也可以取到相应值。 t_1 后续是否会被使用，决定了是否要将 t_1 存入内存单元。应用该模板，最好的

优化结果是只保留第二条，最差的结果是三条都保留。

为了精简代码，四元式结果变量值不急于存储。生成目标代码时，可以先生成前两条，第三条目标代码滞后生成。例如在生成 $(+ a b t_1)$ 对应的目标代码时，先生成前两条，再根据后一条四元式决定是否生成最后一条四元式。

解法 2：第 (1) 个四元式表示将 a 和 b 相加，结果放在 t_1 。 a 和 b 是变量，符号表中记录的变量地址是内存地址，CPU 运算需要在寄存器上操作，因此需要先将运算对象 a 取到寄存器 R_0 中，从而通过 ADD 指令对 R_0 和 b 变量对应的内容进行计算，结果放在 R_0 中。

第 (2) 个四元式表示 t_1 减去 d ，结果放在 t_2 。经过上一步计算， t_1 存在 R_0 中，可以直接使 R_0 中的内容减去 d 变量对应内容，即 SUB R_0, d, R_0 寄存器中内容为 t_2 。

- ① LD R_0, a
- ② ADD R_0, b
- ③ SUB R_0, d

例 9.2 根据下列四元式翻译目标代码：

- (1) $(+ a b t_1)$
- (2) $(- c d t_2)$
- (3) $(* t_1 t_2 t_3)$

解：依照例 9.1 解法 1 的分析，先生成 $(+ a b t_1)$ 对应的前两条目标代码，第三条 ST 滞后生成。处理下一条四元式，两个操作数与 t_1 均不相关，一种处理方式是将 R_0 寄存器中的内容保存到 t_1 中，生成 ST 指令，然后用 R_0 寄存器做 $c - d$ 的运算，这种方式应用于单寄存器；若有多个寄存器可以选择，可以选 R_1 寄存器进行第二条四元式的运算，同样的，模板中的第三条指令滞后生成。再看第三条四元式，来决定前两条四元式的目标代码中的 ST 指令是否生成，发现第三条四元式用到了前两条四元式的结果单元，前两条四元式的 ST 指令均可省去，直接生成乘法操作的目标代码。生成目标代码如下：

- ① LD R_0, a
- ② ADD R_0, b
- ③ LD R_1, c
- ④ SUB R_1, d
- ⑤ MUL R_0, R_1

【讨论】 在真正算法实现时，还需要解决一些问题：

(1) 为了精简代码，四元式结果变量值并不急于存储。上例中没有将 t_1 、 t_2 、 t_3 的值存入内存中，而是放在寄存器中。

(2) 例 9.1 中的 t_1 的值，系统如何知道是在寄存器 R_0 中？

(3) 例 9.2 存在寄存器分配问题，显然，若 t_2 仍然占用寄存器 R_0 ，则 t_1 值将被覆盖。例子中假设有多个寄存器，如果是单寄存器，需要先保存 R_0 中的内容，再进行下一步操作。

9.1.2 变量的活跃信息

为了解决上一节的问题，介绍一个重要概念——**活跃信息**，定义的是一个变量在一个基本块或一段程序内被使用的情况。具体来说需要引入变量的定义点和应用点，来判断一个变量是否活跃。

例 9.2 中的 t_1 ，如果在接下来还会被用到，那它就是活跃的，可能需要执行 ST 指令；如果 t_1 在接下来都不会再被用到，那它就是不活跃的，它所在的寄存器 R_0 就可以让给其他变量。

1. 变量的定义点和应用点设有四元式： $q(\omega \ B \ C \ A)$

应用点和定义点是相对一个四元式而言的，变量 B 和 C 是操作数，在四元式 q 中被使用，则称 B 和 C 在四元式 q 处有应用点 (q)；A 是结果单元，在四元式 q 中对 A 进行赋值，即给 A 一个新的定义，则称 A 在四元式 q 处有定义点 (q)。

利用定义点和应用点的概念，确定一个变量是否为活跃变量。

2. **活跃变量与非活跃变量** **活跃变量**：一个变量从某时刻 (q) 起，到下一个定义点止，期间若有应用点，则称该变量在 q 是活跃的 (y)，否则称该变量在 q 是非活跃的 (n)。活跃和非活跃是相对某一时刻而言，脱离时刻概念，讨论活跃和非活跃也就没有意义。

注意，我们是在一个基本块内讨论变量的活跃信息的，基本块既是优化的基本单位，也是目标代码生成的基本单位，还是求取活跃信息的基本单位。为了方便处理，假定：

- (1) 临时变量在基本块出口后是非活跃的 (n)；在基本块结束后，临时变量不会再被使用。
- (2) 非临时变量在基本块出口后是活跃的 (y)；在基本块结束后，非临时变量会再被使用。

这样的约定，可能不是最高效的，但却是最安全的。不会将活跃变量误认为非活跃变量，而没有及时更新信息，产生错误。

以一个赋值语句为例，中间进行算术运算时，产生的临时变量到赋值完成后都不会再被使用。下面通过具体例子说明变量的活跃信息求解过程。

例 9.3 求下述基本块内变量的活跃信息： $x = (a + b)/(a * (a + b)); i = a + b$ ；解：令 $A(I)$ 中的 I 为变量 A 在某点的活跃信息 (y/n)。

第一步，写出四元式序列：

- ① $(+ \ a \ b \ t_1)$
- ② $(+ \ a \ b \ t_2)$
- ③ $(* \ a \ t_2 \ t_3)$
- ④ $(/ \ t_1 \ t_3 \ t_4)$
- ⑤ $(= \ t_4 \ _ \ x)$
- ⑥ $(+ \ a \ b \ t_5)$
- ⑦ $(= \ t_5 \ _ \ i)$

第二步，进行 DAG 优化，得到优化后的四元式：

- ① (+ a b t₁)
- ② (* a t₁ t₃)
- ③ (/ t₁ t₃ x)
- ④ (= t₁ _ i)

第三步，填写活跃信息。

对于第一个四元式中变量 a ，根据定义，看 a 到下一个定义点之间是否有应用点。发现没有下一个定义点，又因为 a 是用户定义变量，在基本块出口，第四个四元式执行完后是活跃的，认为第四个四元式后还有变量 a 的定义点，且到这个定义点之间还有 a 的应用点。在第四个四元式之后使用变量 a ，变量 a 是活跃的，在第一个到第四个四元式之间的变量 a 当然也是活跃的，活跃信息填 (y)。

对于第一个四元式中变量 b ，虽然在第二个四元式到第四个四元式之间没有使用，但是根据前面对于变量 a 的分析，变量 b 是用户定义变量，在基本块出口时是活跃的，因此活跃信息填 (y)。

对于第一个四元式中变量 t_1 ，在第一个四元式中被定义，在第二个四元式中被使用， t_1 是活跃的，活跃信息填 (y)。

同理，根据变量在下一定义点前是否被使用，并结合基本块出口处的活跃信息，附有活跃信息的四元式如下。其中对于第三个四元式中变量 t_3 ，在第四个四元式中没有 t_3 出现， t_3 在这个基本块内不能再被使用，在第四个四元式之后，因为 t_3 是临时变量，基本块出口后是非活跃的，即 t_3 在第四个四元式之后不会再被使用，因此第三个四元式中变量 t_3 是非活跃的，活跃信息填 (n)。同理第四个四元式中变量 t_1 也是非活跃的。

- (1) (+ a(y) b(y) t₁(y))
- (2) (* a(y) t₁(y) t₃(y))
- (3) (/ t₁(y) t₃(n) x(y))
- (4) (= t₁(n) _ i(y))

3. 基本块内活跃信息求解的算法 ★ 数据结构支持:

(1) 在符号表上增设一个信息项 (L) 用以记录活跃信息，结构如下。活跃信息是动态的，与时刻相关，信息项辅助填写活跃信息。

name	...	L
------	-----	---

图 9.2: 记录活跃信息的信息项

(2) 四元式中变量 X 的附加信息项 $X(L)$ ，取值 $L=n/y$ ，表示不活跃/活跃；

★ 算法:

(1) 初值：基本块内各变量 $SYMBL[X(L)]$ 分别填写：若 X 为非临时变量，则置 $X(y)$ ，否则置 $X(n)$ 。 y/n 表示活跃/不活跃，目的是初始化为基本块出口的状态。

(2) 逆序扫描基本块内各四元式（设为 $q: (\omega B C A)$ ）:

执行:

- ① $QT[q: A(L)] := SYMBL[A(L)]$; 对于结果单元 A , 读取符号表中变量 A 的附加信息项, 作为四元式中结果单元 A 的活跃信息。
- ② $SYMBL[A(L)] := (n)$; 将符号表中变量 A 的附加信息项置为 n (非活跃)。
- ③ $QT[q: B, C(L)] := SYMBL[B, C(L)]$; 对于运算对象 B 、 C , 同样读取符号表中对应变量的附加信息项, 作为四元式中 B 、 C 的活跃信息。
- ④ $SYMBL[B, C(L)] := (y)$; 将符号表中 B 、 C 的附加信息项置为 y (活跃)。

以此类推, 逆序扫描基本块内各个四元式, 直到基本块的第一个四元式。虽然活跃信息的定义是一个正序的定义, 但算法填写活跃信息, 是逆序填写, 从基本块出口状态入手, 反向进行。如果出现定义点, 则定义点之前活跃信息为 n , 如果出现应用点, 则应用点前不到定义点的时刻, 活跃信息为 y 。

下面通过活跃信息生成过程示例进一步说明上述算法。

例 9.4 根据基本块内四元式序列, 填写活跃信息:

解:

第一步, 初始化符号表中的附加信息项如图9.3。对非临时变量, 初始化为 y , 对临时变量初始化为 n 。

基本块内下述四元式序列如下:

$QT[q:]$

$q:(\omega \ B(L) \ C(L) \ A(L))$	
(1)(+ a() b() t ₁ ())	
(2)(- c() d() t ₂ ())	
(3)(* t ₁ () t ₂ () t ₃ ())	
(4)(- a() t ₃ () t ₄ ())	
(5)(/ t ₁ () 2 t ₅ ())	
(6)(+ t ₄ () t ₅ () x())	

SYMBL[X(L)]	
	L
a	y
b	y
c	y
d	y
t ₁	n
t ₂	n
t ₃	n
t ₄	n
t ₅	n
x	y

图 9.3: 初始化附加信息项

第二步, 逆序扫描四元式, 填写活跃信息。

第 (6) 个四元式, 对于结果单元 x , 从符号表中取得 x 的附加信息项内容, 作为 x 的活跃信息, 并将符号表中 x 的附加信息项置为 n , 这是 x 的定义点, 在定义点之前且没有应用点, x 是非活跃的。类似地, 填写 t_4 、 t_5 的活跃信息, 先从符号表中取得附加信息项内容作为对应变量的活跃信息, 然后将符号表中的附加信息项置为 y , t_4 、 t_5 被使用, 则在之前也是活跃的。

第 (5) 个四元式, 对于结果单元 t_5 , 从符号表中取得 t_5 的附加信息项内容, 作为的它活跃信息, 并将符号表中 t_5 的附加信息项置为 n 。对于 t_1 , 活跃信息填从符号表中取得的附加信息项内容, 并将符号表中修改为 y 。

第(4)个四元式, 对于结果单元 t_4 , 取符号表中的附加信息项, 作为活跃信息, 并将符号表中的附加信息项置为 n 。对于运算对象 a 和 t_3 , 取符号表中的附加信息项, 作为活跃信息, 并将符号表中的附加信息项置为 y 。

类似地, 得到活跃信息结果如下图9.4。

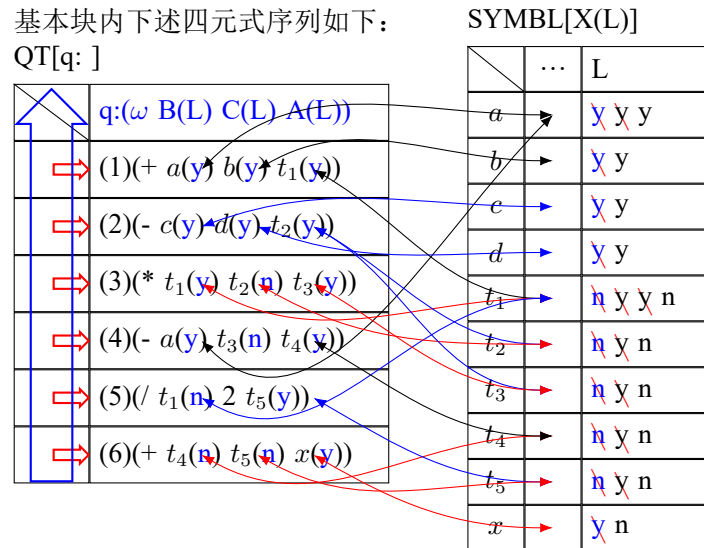


图 9.4: 活跃信息填表结果

用户定义变量大多数情况下是活跃的, 但如果对同一变量两次赋值之间, 没有进行使用, 用户定义变量就会是非活跃的。

9.1.3 寄存器的分配问题

目标代码生成部分, 目标指令的生成是根据模板进行, 不是重难点, 重点和难点在于优化。优化中一个很重要的内容, 就是对寄存器使用的优化, 即寄存器分配问题。寄存器在整个计算机体系, 在编译器设计中十分重要, 因为寄存器操作快且指令短, 从寄存器中取数据远快于从内存或外存中取数据。

系统需要知道当前寄存器中存储的变量, 或者说一个变量是否在寄存器中, 在哪一个寄存器中。

1. 设置描述表 $RDL(R_0, R_1, \dots, R_n)$ 用以记录寄存器的当前状态, 如 $RDL.R_1 = x$, 即指明当前变量 x 值在寄存器 R_1 中。
2. 寄存器分配三原则: 设当前四元式: $q: A = B \omega C$, 为 A 分配寄存器有以下三种情况:
 - (1) 主动释放。如果 B 已经在寄存器 R_i 中, 则选择 R_i ; 若 ω 可交换, 也可考虑 C 所在寄存器。
 - ① 若 B 活跃, 即在下个定义点前存在应用点, 则要保存 B 的值, 方法是: 若有空闲寄存器 R_j , 则要将 R_i 中的 B 保存到 R_j 中, 生成指令 $ST R_i, R_j$; 否则暂时存到内存中, 生成指令 $ST R_i, B$ 。若 B 不活跃, 就不用保存 B 的值, 不生成相关指令。
 - ② 修改描述表: 删除 B , 填写 A 。

(2) **选空闲者**。从空闲寄存器中选一 R_i ；并把 A 填入描述表。

(3) **强迫释放**。剥夺一 R_i ；处理办法同规则 (1)。

以 $(\omega B C A)$ 为例说明上述三条分配原则。进行 ω 运算，需要先指定寄存器 R，运算结束后，寄存器 R 中的值就是结果单元 A 的值。先确定一个寄存器作为运算寄存器，即给 A 分配一个寄存器。寄存器的选择有三种情况：

(1) B 在 R_i 中，根据运算四元式生成的三条目标代码，第一条是将 B 取到寄存器中，此时 B 已经在寄存器中，这三条中的第一条即可省去，这种情况称为主动释放，B 主动将寄存器 R_i 释放给 A。如果 B 是活跃的，要先保存 B，然后再进行 ω 运算，若有其他空闲寄存器 R_j ，就生成指令 $ST R_i, R_j$ ，否则生成 $ST R_i, B$ ；如果 B 不活跃，直接进行 ω 运算。计算完成后， R_i 寄存器中保存的是 A 的值，因此修改描述表 $R_i = A$ 。若对于乘法和加法，第一运算对象和第二运算对象可交换时，还可以考虑主动释放寄存器 C 给 A。

(2) 如果 B 不在寄存器中，乘法加法运算中 C 也不在寄存器中，则在空闲寄存器中选一个分配给 A，并修改描述表。

(3) 如果 B、C 都不在寄存器中，也没有空闲寄存器，就“抢”一个寄存器，处理办法同第一种情况。怎么“抢”也有多种方法：随机抢、优先抢最不活跃的等。

例 9.5 设有三个寄存器： R_0, R_1, R_2 。根据给定的四元式及活跃信息，生成目标代码：

解：顺序处理四元式，根据寄存器分配三原则、活跃变量概念，写出目标代码。

设有三个寄存器：**R0,R1,R2**

支持：寄存器分配原则；活跃变量概念。

QT[q]	OBJ[p]	RDL		
		R0	R1	R2
(1)(+ a(y) b(y) t ₁ (y))				
(2)(- c(y) d(y) t ₂ (y))				
(3)(* t ₁ (y) t ₂ (n) x(y))				
(4)(= a(y) _ y(n))				
(5)(/ a(n) x(n) x(y))				
(6)(- a(n) x(n) x(y))				
(7)(= x(y) _ a(y))				
(8) ...				

图 9.5: 寄存器分配表

第 (1) 个四元式，给结果单元 t_1 分配寄存器，初始状态下三个寄存器均为空，选择寄存器 R_0 ，生成前两条目标代码为①LD R_0, a ②ADD R_0, b ，模板中第三条指令滞后生成。修改描述表 RDL，此时 R_0 保存 t_1 ， R_1 、 R_2 空闲。

第 (2) 个四元式，给结果单元 t_2 分配寄存器，运算对象 c 和 d 都不在寄存器中，根据分配三原则“选空闲者”，将空闲寄存器 R_1 分配给 t_2 ，生成目标代码③LD R_1, c ④SUB R_1, d ，模板第

三条指令滞后生成。修改描述表 RDL，此时 R_0 保存 t_1 ， R_1 保存 t_2 ， R_2 空闲。

第 (3) 个四元式， t_1 和 t_2 相乘，由于 t_1 在寄存器 R_0 中，根据分配三原则“主动释放”，给 x 分配寄存器 R_0 ，由于 t_1 是活跃的，需要保存，且有空闲寄存器 R_2 ，将 t_1 保存到 R_2 ，之后完成计算，生成目标代码⑤ST R_0, R_2 ⑥MUL R_0, R_1 ，模板第三条仍然滞后生成。修改描述表 RDL，此时 R_0 保存 x ， R_1 保存 t_2 ， R_2 保存 t_1 。

第 (4) 个四元式，给 y 分配寄存器， a 不在寄存器中，此时也没有空闲寄存器，根据分配三原则“强迫释放”，因 t_2 不活跃，则剥夺 t_2 所在寄存器 R_1 分配给 y ，因 t_2 不活跃，不必生成 ST 指令，生成目标代码⑦LD R_1, a ，模板第二条滞后生成。修改描述表 RDL，此时 R_0 保存 x ， R_1 保存 y ， R_2 保存 t_1 。

第 (5) 个四元式， a 除以 x 放在 x 中，尽管前一句将 a 放在 R_1 ，但根据寄存器描述表， R_1 寄存器的标志为 y ，认为 R_1 中没有 a 。又因除法运算是不可交换的，尽管 x 在寄存器中，但不能使用 x 所在寄存器。根据分配三原则“强迫释放”，因 y 不活跃，则剥夺 y 所在寄存器 R_1 分配给 x ，因 y 不活跃，不生成 ST 指令，生成目标代码⑧LD R_1, a ⑨DIV R_1, R_0 ，模板第三条仍然滞后生成。⑦和⑧相同，后续可以进一步优化，但根据算法逻辑，会生成这样的结果。修改描述表 RDL，此时 R_0 空闲， R_1 保存 x ， R_2 保存 t_1 。

第 (6) 个四元式，给 y 分配寄存器，运算对象 x 在寄存器 R_1 中，根据分配三原则“主动释放”，将 R_1 分配给 y ，由于 x 是活跃的，需要保存，且有空闲寄存器 R_0 ，将 x 保存到 R_0 ，然后进行运算，生成目标代码⑩ST R_1, R_0 ⑪SUB R_1, t_1 ，模板第三条仍然滞后生成。修改描述表 RDL，此时 R_0 保存 x ， R_1 保存 y ， R_2 保存 t_1 。

第 (7) 个四元式，给 a 分配寄存器，运算对象 x 在寄存器 R_0 中，根据分配三原则“主动释放”，将 R_0 分配给 a ，由于 x 是活跃的，需要保存，且 R_2 寄存器中的 t_1 不活跃，将 x 保存到 R_2 中，生成目标代码⑫ST R_0, R_2 ，模板第二条滞后生成。修改描述表 RDL，此时 R_0 保存 a ， R_1 保存 y ， R_2 保存 x 。

填表结果如下图9.6:

QT[q]	OBJ[p]	R_0	R_1	R_2
➔(1)(+ $a(y)$ $b(y)$ $t_1(y)$)	①LD R_0, a ②ADD R_0, b	t_1		
➔(2)(- $c(y)$ $d(y)$ $t_2(y)$)	③LD R_1, c ④SUB R_1, d		t_2	
➔(3)(* $t_1(y)$ $t_2(n)$ $x(y)$)	⑤ST R_0, R_2 ⑥MUL R_0, R_1	x		t_1
➔(4)(= $a(y)$ $_y(n)$)	⑦LD R_1, a		y	
➔(5)(/ $a(n)$ $x(n)$ $x(y)$)	⑧LD R_1, a ⑨DIV R_1, R_0		x	
➔(6)(- $a(n)$ $x(n)$ $x(y)$)	⑩ST R_1, R_0 ⑪SUB R_1, t_1	x	y	
➔(7)(= $x(y)$ $_a(y)$)	⑫ST R_0, R_2	a		x
➔(8) ...				

图 9.6: 寄存器分配结果

在寄存器分配过程中，需要注意：

- (1) 一个变量在同一时刻只能占有一个寄存器；
- (2) 在基本块出口时，寄存器中的活跃变量应保存其值。

另外，寄存器分配时，是先主动释放，还是先分配空闲寄存器，这是一种算法设计，主要考虑目标代码的效率，与设计的四元式有关，可以自行调整。

如果各个寄存器中的变量都活跃，则选择一个寄存器强迫释放，并将寄存器中的变量保存到内存中，后面使用该变量时，再将其读到寄存器中。这样的访存操作，也是导致程序执行慢的主要原因。

9.1.4 目标代码生成问题

目标代码生成是以**基本块**为单位的，在生成目标代码时要注意如下三个问题：

- (1) 基本块开始时所有寄存器应是空闲的；结束基本块时应释放所占用的寄存器。
- (2) 一个变量被定值（被赋值）时，要分配一个寄存器 R_i 保留其值，并且要填写相应的描述表 $RDL.R_i$ 。
- (3) 为了生成高效的目标代码，生成算法中要引用寄存器分配三原则和变量的活跃信息。

定义**数据结构**如下：

- $QT[q]$ ——四元式区：存放四元式。
- $OBJ[p]$ ——目标区：存放目标代码。
- $RDL(R_0, R_1, \dots, R_n)$ ——寄存器状态描述表
- $SEM(m)$ ——语义栈（用于信息暂存）：主要用于跳转指令生成目标代码时，与中间代码生成时的语义栈截然不同。

注意，这里没有考虑符号表。

具体讲解单寄存器下，一些常用四元式目标代码生成过程，包括表达式、条件语句和循环语句。

例 9.6 单寄存器 (R) 下表达式目标代码生成：

设： R 表示寄存器； RDL 表示描述表； SEM 表示语义栈。

解：对于表达式生成目标代码，都是顺序生成，用不到 SEM 结构。根据题意，仅有一个寄存器 R ，起始状态为空。主动释放、选空闲者、强迫释放三原则对于单寄存器的情况，没有很大意义，但仍沿用这一说法。

第 (1) 个四元式，给结果单元 t_1 分配寄存器 R ，生成目标代码① $LD R, a$ ② $ADD R, b$ 。修改描述表 RDL ， R 保存 $a + b$ 的结果 t_1 。

B	QT[q]	OBJ[p]	RDL	SEM
	(1)(+ a(y) b(y) t ₁ (y))			
	(2)(- c(y) d(y) t ₂ (y))			
	(3)(* t ₁ (y) t ₂ (n) t ₃ (y))			
	(4)(- a(y) t ₃ (n) t ₄ (y))			
	(5)(/ t ₁ (n) 2 t ₅ (y))			
	(6)(+ t ₄ (n) t ₅ (n) x(y))			
	...			

图 9.7: 寄存器分配表

第 (2) 个四元式, 给结果单元 t_2 分配寄存器, 运算对象 c 不在寄存器 R 中, 由于是单寄存器, 因此“强迫释放”, 剥夺 t_1 所在寄存器 R 分配给 t_2 , 且 t_1 活跃, 需要保存到内存, 生成 ST 指令。然后将 c 读到 R , 进行 $c - d$ 运算, 生成目标代码③ $ST R, t_1$ ④ $LD R, c$ ⑤ $SUB R, d$ 。修改描述表 RDL , R 保存 $c - d$ 的结果 t_2 。

第 (3) 个四元式, 给结果单元 t_3 分配寄存器, 运算对象 t_1 不在寄存器 R 中, 乘法运算可交换, 运算对象 t_2 在寄存器 R 中, 进行“主动释放”, 且 t_2 不活跃, 不需要保存到内存, 生成目标代码⑥ $MUL R, t_1$ 。修改描述表 RDL , R 保存 $t_1 * t_2$ 的结果 t_3 。

第 (4) 个四元式, 虽然 t_3 在寄存器中, 但由于减法运算不可交换, 因此“强迫释放”, 先读入 a 再减 t_3 : 将 t_3 保存到内存 (减法中使用), 生成 ST 指令。然后将 a 读到 R , 进行 $a - t_3$ 运算, 生成目标代码⑦ $ST R, t_3$ ⑧ $LD R, a$ ⑨ $SUB R, t_3$ 。修改描述表 RDL , R 保存 $a - t_3$ 的结果 t_4 。(此处 t_3 标注非活跃的原因, 是在读 a 之前保护现场, 而非运算完成后释放 t_3)

第 (5) 个四元式, 给结果单元 t_5 分配寄存器, 运算对象 t_1 不在寄存器 R 中, “强迫释放”, 剥夺 t_4 所在寄存器 R 分配给 t_5 , 且 t_4 活跃, 需要保存到内存, 生成 ST 指令。然后将 t_1 读到 R , 进行 $t_1 / 2$ 运算, 生成目标代码⑩ $ST R, t_4$ ⑪ $LD R, t_1$ ⑫ $DIV R, 2$ 。修改描述表 RDL , R 保存 $t_1 / 2$ 的结果 t_5 。

第 (6) 个四元式, 与 (3) 类似, 给 x 分配寄存器, 加法运算可交换, 运算对象 t_5 在寄存器 R 中, “主动释放”, 生成目标代码⑬ $ADD R, t_4$ 。

填表结果如下图9.8:

表达式的目标代码是顺序生成的, 比较容易理解, 接下来介绍带跳转语句的目标代码生成。

例 9.7 单寄存器 (R) 下, 条件语句目标代码生成: 给定程序段如下, 生成目标代码。

$\text{if}(a > b)x = (a + b) * c; \text{else } x = 5 - a * b$

解: SEM 栈用于保存待返填地址, 具体实现可采用栈或队列的结构, 本书中采用栈的形式进行说明。

B	QT[q]	OBJ[p]	RDL	SEM
⇒	(1)(+ a(y) b(y) t ₁ (y))	①LD R, a ②ADD R, b	t₁	→
⇒	(2)(- c(y) d(y) t ₂ (y))	③ST R, t ₁ ④LD R, c ⑤SUB R, d	t₂	
⇒	(3)(* t ₁ (y) t ₂ (n) t ₃ (y))	⑥MUL R, t ₁	t₃	
⇒	(4)(- a(y) t ₃ (n) t ₄ (y))	⑦ST R, t ₃ ⑧LD R, a ⑨SUB R, t ₃	t₄	
⇒	(5)(/ t ₁ (n) 2 t ₅ (y))	⑩ST R, t ₄ ⑪LD R, t ₁ ⑫DIV R, 2	t₅	
⇒	(6)(+ t ₄ (n) t ₅ (n) x(y))	⑬ADD R, t ₄	x	
	...			

图 9.8: 寄存器分配结果

第一步，生成四元式序列，划分基本块并标注活跃信息。基本块划分时，跳转到的语句，及跳转的下一条语句都是基本块的开始，并要求程序入口也是基本块的开始。在基本块入口和出口时，要保证寄存器是空闲的。

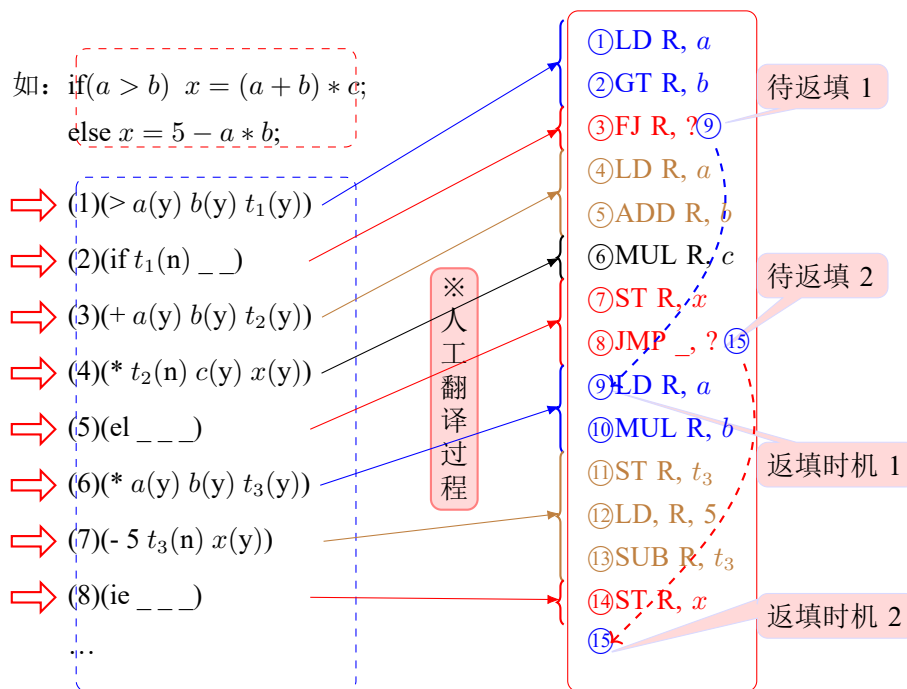


图 9.9: 例 9.7 人工翻译过程

第二步，人工翻译目标代码。根据题意，只有一个寄存器 R，初始状态为空。

第 (1) 个四元式，给 t₁ 分配寄存器，生成目标代码①LD R, a ②GT R, b，GT 为 > 的操作码，表示比较 R 中内容是否大于 b。修改描述表 RDL，R 保存 t₁。

第 (2) 个四元式，if 四元式当 t₁ 为假时，跳转到第 (6) 个四元式的第一条目标代码，但在生成 if 四元式的目标代码时，跳转位置未知，生成假跳目标代码③FJ R, ?, ? 表示待返填。跳转位置要等第 (6) 个四元式目标代码生成后，才能填写，因此将带问号的目标代码编号③保存到 SEM

中。第(2)个四元式是一个基本块的出口, 需要将所有寄存器内容释放, t_1 不活跃, 不需要保存到内存。修改描述表 RDL, R 空闲。

第(3)个四元式, 给 t_2 分配寄存器, 将 a 读入寄存器 R, 进行 $a+b$ 运算, 生成目标代码④LD R, a ⑤ADD R, b 。修改描述表 RDL, R 保存 t_2 。

第(4)个四元式, 运算对象 t_2 在寄存器中, 乘法运算可交换, 进行“主动释放”, 且 t_2 不活跃, 不需要保存到内存, 直接进行 t_2*c 运算, 生成目标代码⑥MUL R, c 。修改描述表 RDL, R 保存 x 。

第(5)个四元式, else 四元式是基本块出口, 需要清空寄存器。根据约定, 用户定义变量在基本块出口是活跃的, 需要保存到内存, 生成目标代码⑦ST R, x 。同时该四元式表示无条件跳转, 跳转到 ie 四元式的下一条, 跳转位置未知, 生成目标代码⑧JMP, ?, 将带问号的目标代码编号⑧保存到 SEM 中, ? 待返填。

第(6)个四元式, 生成的第一条目标代码编号为⑨, 注意第(2)个四元式的跳转位置为第(6)个四元式的第一条目标代码, 所以将⑨填入第一个返填位置: 从 SEM 队列中找到编码③, 然后回填到编码为③的目标代码的? 处。之后生成第(6)个四元式的目标代码⑨LD R, a ⑩MUL R, b 。修改描述表 RDL, R 保存 t_3 。

第(7)个四元式, 与前一例类似, 减法运算不可交换, 第二运算对象 t_3 需要保存到内存供减法中使用, 生成目标代码⑪ST R, t_3 ⑫LD R, 5 ⑬SUB R, t_3 。修改描述表 RDL, R 保存 x 。

第(8)个四元式, 作为基本块出口, 要释放寄存器, 用户定义变量 x 保存到内存, 生成目标代码⑭ST R, x 。下一条目标代码编号为 15, 填入第二个返填位置: 从 SEM 队列中找到编码⑧, 然后回填到编码为⑧的目标代码的? 处。

注意, 要及时处理跳转地址返填。在实践中, 先返填, 后编写跳转指令。如生成第五个四元式对应的目标代码⑦、⑧时, 先不生成⑧的指令, 先将下一条目标代码的编号⑨返填, 再编写跳转指令⑧。因为编写跳转指令后, 要将编号压栈, 栈顶不是③, 是⑧, 返填时就会出现问题。

填表结果如下图9.10:

上述例题中, 都是通过人工的方式翻译目标代码, 接下来系统介绍生成目标代码。

9.2 目标代码生成算法设计

通过两道例题总结算法。

例 9.8 给定如下基本块, 写出四元式序列, 进行 DAG 优化, 完成单寄存器目标代码生成:

$$a = 10 + 5$$

$$x = 10 + 5 - y$$

$$y = a * x$$

解: 第一步, 生成四元式序列如下:

如: $\text{if}(a > b) x = (a + b) * c; \text{else } x = 5 - a * b;$

※ 计算机目标代码生成过程示例:

B	QUAT[q]	OBJ[p]	RDL	SEM
→	(1)($> a(y) b(y) t_1(y)$)	①LD R, a ②GT R, b	t1	→
←	(2)($\text{if } t_1(n) _ _$)	③FJ R, ? ④		3
→	(3)($+ a(y) b(y) t_2(y)$)	④LD R, a ⑤ADD R, b	t2	
	(4)($* t_2(n) c(y) x(y)$)	⑥MUL R, c	x	
→	(5)($\text{el } _ _ _$)	⑦ST R, x ⑧JMP $_ _ _ ?$ ⑮		8
←	(6)($* a(y) b(y) t_3(y)$)	⑨LD R, a ⑩MUL R, b	t3	
	(7)($- 5 t_3(y) x(y)$)	⑪ST R, t3 ⑫LD R, 5 ⑬SUB R, t3	x	
←	(8)($\text{ie } _ _ _$)	⑭ST R, x		
	...	⑮		

图 9.10: 例 9.7 填表结果

- ① (+, 10, 5, t₁)
- ② (=, t₁, $_ _$, a)
- ③ (+, 10, 5, t₂)
- ④ (-, t₂, y, t₃)
- ⑤ (=, t₃, $_ _$, x)
- ⑥ (*, a, x, t₄)
- ⑦ (=, t₄, $_ _$, y)

第二步, 画 DAG 图如图9.11:

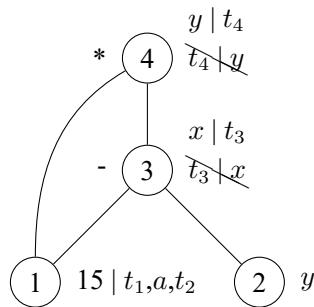


图 9.11: 例 9.8 DAG 图

第三步, 得到优化后的四元式序列如下。括号中待填的是活跃信息。

- ① (=, 15, $_ _$, a())
- ② ($_ _$, 15, y(), x())
- ③ (*, 15, x(), y())

第四步, 生成目标代码。(1) 标活跃信息:

给变量 a, x, y 标活跃信息，由于均为用户定义变量，初值均为 y ，逆序填活跃信息。对最后一个四元式，填结果单元 y 的活跃信息，先从符号表主表的信息项读取当前状态 y ，在符号表中返填 n ；填运算对象 x 的活跃信息，先从符号表主表的信息项读取当前状态 y ，在符号表中返填 y 。类似地，完成活跃信息的填写，结果如下：

- ① ($=$, 15, $_$, $a(y)$)
- ② ($_$, 15, $y(n)$, $x(y)$)
- ③ ($*$, 15, $x(y)$, $y(y)$)

a		n
x		n
y		y

(2) 目标指令生成：

第一个四元式，是赋值四元式，生成模板为：LD R, ...; ST R, ... 给 a 分配寄存器，生成目标指令：LD R, 15 模板第二条滞后生成。

第二个四元式，是运算四元式，生成模板为 LD R, ...; \tilde{w} R, ...; ST R, ...。给 x 分配寄存器，运算对象不在寄存器中，由于 a 是活跃变量，需要保存到内存，生成 ST 指令，生成目标指令 ST R, a ; LD R_m 15; SUB R, y ，模板第三条滞后生成。

第三个四元式，给 y 分配寄存器，寄存器中的值为第二运算对象，乘法运算可交换，模板第一条可省去， x 主动释放寄存器，由于 x 是活跃变量，需要保存到内存，生成 ST 指令，生成目标指令为 ST R, x ; MUL R, 15。一个基本块结束，释放寄存器，由于 y 是活跃变量，需要保存到内存，生成目标指令 ST R, y 。结果如下：

目标指令生成

- ① LD R, 15
- ② ST R, a
- ③ LD R, 15
- ④ SUB R, y
- ⑤ ST R, x
- ⑥ MUL R, 15
- ⑦ ST R, y

例 9.9 给定如下基本块，写出四元式序列，进行 DAG 优化，完成单寄存器目标代码生成：

$$y = 1$$

$$x = 10 + 5 - y$$

$$y = a * x$$

$$b = a + b$$

第一步，生成四元式序列如下：

- ① ($=$, 1, $_$, y)
- ② ($+$, 10, 5, t_1)
- ③ ($-$, t_1 , y , t_2)
- ④ ($=$, t_2 , $_$, x)
- ⑤ ($*$, a , x , t_3)
- ⑥ ($=$, t_3 , $_$, y)
- ⑦ ($+$, a , b , t_4)
- ⑧ ($=$, t_4 , $_$, b)

第二步，画 DAG 图如下图9.12:

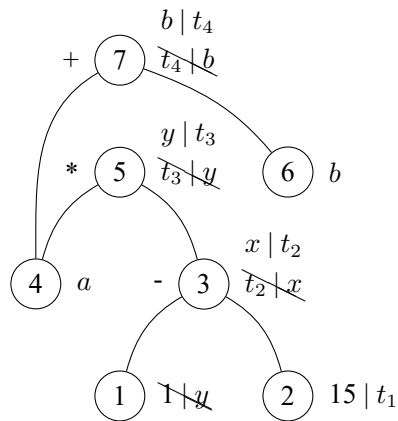


图 9.12: 例 9.9 DAG 图

第三步，得到优化后的四元式序列如下，括号中待填的是活跃信息。

- ① ($-$, 15, 1, $x()$)
- ② ($*$, $a()$, $x()$, $y()$)
- ③ ($+$, $a()$, $b()$, $b()$)

第四步，生成目标代码。(标活跃信息，目标指令生成)

- ① ($-$, 15, 1, $x(y)$)
- ② ($*$, $a(y)$, $x(y)$, $y(y)$)
- ③ ($+$, $a(y)$, $b(n)$, $b(y)$)

x | n
 y | n
 a | y
 b | y

目标指令生成

- ① LD R, 15
- ② SUB R, 1
- ③ ST R, x
- ④ MUL R, a
- ⑤ ST R, y
- ⑥ LD R, a
- ⑦ ADD R, b
- ⑧ ST R, b

总结算法之前，先要明确目标代码的生成要点和生成环境。

9.2.1 目标代码生成要点和生成环境

★ 生成要点

1. 目标代码生成是在一个基本块上进行的：

- (1) 入口：寄存器空闲；
- (2) 出口：释放寄存器。如9.7中编号 7 和 14 的目标代码。

2. 目标代码生成是以一个一个四元式为单位进行的：

- (1) 表达式、赋值四元式：首先对结果变量申请寄存器；然后编写目标指令；并修改描述表。
- (2) 转向四元式：首先保存占用寄存器的活跃变量值（转向四元式是基本块出口）；然后再编写跳转指令；同时记住待返填地址。先保存活跃变量，还是先编写跳转指令，可以自己设计。

注意：释放寄存器时，编写存储指令，保存占有寄存器的活跃变量值，通常发生在如下两个时刻：

- (1) 为结果变量申请寄存器时。强迫释放时，要先保存寄存器内活跃变量。
- (2) 基本块出口时。出口需要释放寄存器。

★ 生成环境

1. 虚拟机：单一寄存器 R；指令形式 $p: (op R, M)$ ，M 表示变量内存地址

含义： $R := (R) op (M)$ 或 $R := op (M)$

2. 表、区和栈：QY[q]——四元式区（附有变量的活跃信息）；

OBJ[p]——目标代码区；

SEM[m]——语义栈（登记待返填的目标地址）；

SYMBL[i]——符号表；

RDL[R]——寄存器描述表；

当 $RDL=0$ 时，表示寄存器 R 空闲；当 $RDL=X$ 时，表示寄存器 R 被变量 X 占用。

3. **变量和函数**: $CODE(op, R, M; \dots)$ —— (送代码函数) 把目标代码送目标区;
 $BACK(p_i, p_k)$ —— (返填函数) 把地址 p_k 返填到地址 p_i 中, p_k 表示跳转到的地址, p_i 表示待填地址。

9.2.2 表达式四元式目标代码生成算法

设当前扫描的四元式 $q: (\omega \ B \ C \ A)$, 是操作四元式, 对应包含三条目标指令的模板。

(1) 为结果单元 A 预分配寄存器, 编写目标指令:

① 当 $RDL == 0$, 则 $CODE(LD \ R, B; \tilde{\omega} \ R, C)$ 。 $\tilde{\omega}$ 表示算符 ω 对应的操作码。

② 当 $RDL == B$, 则

若 $B(y)$, 则 $CODE(ST \ R, B; \tilde{\omega} \ R, C)$;

否则 $CODE(\tilde{\omega} \ R, C)$ 。

③ 当 $RDL == C$, 且 ω 可交换, 则

若 $C(y)$, 则 $CODE(ST \ R, C; \tilde{\omega} \ R, B)$;

否则 $CODE(\tilde{\omega} \ R, B)$ 。

④ 当 $RDL == D$ (上述三种情况之外) 则

若 $D(y)$, 则 $CODE(ST \ R, D; LD \ R, B; \tilde{\omega} \ R, C)$;

否则 $CODE(LD \ R, B; \tilde{\omega} \ R, C)$

(2) 变量 A 登录到描述表中: $RDL := A$;

9.2.3 赋值四元式目标代码生成算法

设当前四元式 $q: (= \ B \ _ \ A)$

(1) 为 A 预分配寄存器, 编写目标指令:

① 当 $RDL == 0$, 则 $CODE(LD \ R, B)$; 此时 R 寄存器的标志是 A。

② 当 $RDL == B$, 则

若 $B(y)$, 则 $CODE(ST \ R, B)$ 。

此时 B 在寄存器里, 但不一定在内存里, 所以需要判断 B 是否活跃。若 B 活跃, 寄存器被清空时要保证 B 写回内存; 若 B 不活跃, 则不需要保存, 直接修改描述表即可。

③ 当 $RDL == D$ ($D \neq B, D \neq A$) 则

若 $D(y)$, 则 $CODE(ST \ R, D; LD \ R, B)$;

否则 $CODE(LD \ R, B)$ 。

(2) 变量 A 登录到描述表中: $RDL := A$;

9.2.4 条件语句四元式目标代码生成算法

有了表达式四元式和赋值四元式目标代码生成算法，接下来介绍条件语句和循环语句目标代码生成的算法。if 语句、while 语句等从语义上实现的是一种逻辑，对于不同条件，有不同的处理，这类逻辑在 if 语句中，主要关注 if 四元式、el 四元式和 ie 四元式。为了简化说明，假设不存在 if 嵌套，其他四元式均顺序执行，为表达式四元式或赋值四元式，前面已经介绍过目标代码生成的算法。接下来主要要解决的问题是，对于 if 四元式、el 四元式和 ie 四元式，如何生成目标指令。

回顾例9.7，if 四元式在中间代码级别语义为假跳，即当 t_1 为假时，跳转到 el 四元式的下一句。目标代码的语义与中间代码等价，if 语句的目标指令语义同样对应假跳。再看 el 四元式，中间代码语义为无条件跳，对应目标指令的无条件跳。对 ie 四元式，在中间代码级别是作为一个跳转标志，没有语义，因此没有对应的目标指令生成。

如：if($a > b$) $x = (a + b) * c$; else $x = 5 - a * b$;

※ 计算机目标代码生成过程示例：

B	QUAT[q]	OBJ[p]	RDL	SEM
→	(1)($> a(y) b(y) t_1(y)$)	①LD R, a ②GT R, b	t_1	→
←	(2)(if $t_1(n) _ _$)	③FJ R, ? ④		③
→	(3)($+ a(y) b(y) t_2(y)$)	④LD R, a ⑤ADD R, b	t_2	
	(4)($* t_2(n) c(y) x(y)$)	⑥MUL R, c	x	
→	(5)(el $_ _$)	⑦ST R, x ⑧JMP $_$, ? ⑨		⑧
←	(6)($* a(y) b(y) t_3(y)$)	⑨LD R, a ⑩MUL R, b	t_3	
	(7)($- 5 t_3(y) x(y)$)	⑪ST R, t_3 ⑫LD R, 5 ⑬SUB R, t_3	x	
←	(8)(ie $_ _$)	⑭ST R, x		
	...	⑮		

实践中，先返填，后编跳转指令！

图 9.13: 目标代码生成示例

根据上例，总结得到条件语句四元式目标代码生成算法，SEM 采用栈结构，先返填，后编写跳转指令。

★ 条件语句的四元式结构：

设条件语句：if(E) S1; else S2;

则四元式结构如图9.14:

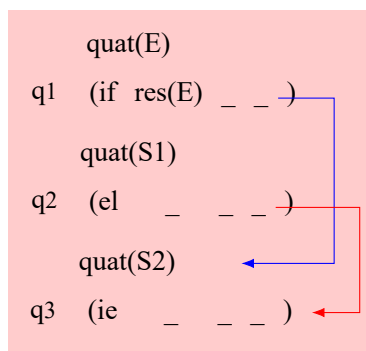


图 9.14: 四元式结构

注意：q2(el _ _ _) 的下面就是 S2 语句的入口。

★ 算法设计：

1. 设当前四元式 q: (if B _ _ _)

(1) 若 $RDL == 0$ (寄存器为空)，则 $CODE(LD R, B ; FJ R, _)$; $PUSH(p)$;

p 为待返填的目标代码 (FJ R, _) 的编号 (地址)。

(2) 若 $RDL == B$,

① 若 $B(y)$ ，则 $CODE(ST R, B ; FJ R, _)$ ；否则 $CODE(FJ R, _)$;

if 语句为基本块出口，要释放寄存器，B 在寄存器 R 中，若 B 活跃，要保存到内存。

② $PUSH(p)$; $RDL := 0$;

(3) 若 $RDL == D$ ($D \neq B$)

① 若 $D(y)$ ，则 $CODE(ST R, D ; LD R, B ; FJ R, _)$ ；否则 $CODE(LD R, B ; FJ R, _)$;

② $PUSH(p)$; $RDL := 0$;

2. 设当前四元式 q: (el _ _ _)

else 语句有两个功能：①它的下一条语句是 if 语句跳转到的位置；②它本身是跳转语句，跳转到条件语句的出口位置。且 else 语句是一个基本块的出口，需要释放寄存器。

(1) 当 $RDL == X$ 且 $X(y)$ ，则 $CODE(ST R, X ;)$; $RDL := 0$

(2) 返填转向地址： $POP(p')$; $BACK(p', p+2)$ ；POP 函数表示 SEM 栈顶地址弹到 p'，BACK 函数表示把地址 p+2 填入 p' 中。

9.7中 SEM 若使用栈结构，对于第 (5) 个四元式，未生成对应目标代码 □ 时，执行 $POP(p')$ 函数，根据 SEM 栈顶元素 p' (编号③)，找到对应的目标指令。下一步执行 $BACK(p', p+2)$ ，p' 是待返填的目标指令编号，对于条件语句，在编号为 p 的目标指令之后，还有编号为 p+1 的无条件跳转目标指令 $JMP ?$ ，跳转的目标位置在 JMP 指令之后，即编号为 p+2，因此将 p+2 返填回 p'。此处，p 为当前指令编号⑦，p+2 为⑨，即将⑨返填回③对应的目标代码。⑧目标代码还未生成，先将⑨返填回③。

(3) 编写转向指令， $CODE(JMP _, _)$; $PUSH(p)$

3. 设当前四元式 q : (ie ___)

if end 语句为基本块出口, 需要释放寄存器。

(1) 当 $RDL == X$ 且 $X(y)$, 则 $CODE(ST R, X); RDL := 0$

(2) 返填转向地址: $POP(p'); BACK(p', p+1)$ 。

9.7中, 对于第 (8) 个四元式, 已经生成编号 14 的目标代码, p 为编号 14, 执行 $POP(p')$, 当前 SEM 栈顶元素 p' 为⑧。下一步执行 $BACK(p', p+1)$, 因为 (ie ___) 生成目标指令中没有 JMP 指令, 因此跳转目标位置的编码为 $p+1$ 为编号 15, 将编号 15 返填回⑧对应的目标代码。

9.2.5 循环语句四元式目标代码生成算法

有了条件语句四元式目标代码生成的基础, 循环语句就容易理解了。

例 9.10 写出给定语句的目标代码:

```
while (a > b) x = (a + b) * c;
```

解:

第一步, 写出对应四元式序列, 划分基本块, 并标注活跃信息。

第二步, 生成目标代码。

第 (1) 个四元式, 每次循环结束后, 都要跳转到该位置, 再进行条件判断, 这个语句本身不生成相应的目标代码, 但是它的下一条语句是循环的开始位置, 所以要将这个语句的编号①压入 SEM 栈中, 为了后面跳转到下一次循环时, 找到跳转位置。

第 (2) 个四元式, 与前面例题类似, 生成目标代码①LD R, a ②GT R, b。

第 (3) 个四元式, do 语句判断 t_1 为假时, 进行跳转, 又因该四元式为基本块出口, 需要释放寄存器, t_1 非活跃, 不需要进行保存, 生成目标指令③FJ R, ?, 跳转到第 (6) 个四元式生成的第一条目标代码, 现在目标代码编号未知, 将③压入 SEM 栈, ? 待返填。

第 (4)、(5) 个四元式, 与前面例题类似, 分别生成目标代码④LD R, ? ⑤ADD R, b 和⑥MUL R, c。

第 (6) 个四元式, 是基本块出口, 需要释放寄存器, 保存寄存器 R 中的活跃变量 x 到内存中, 生成目标代码⑦ST R, x; 同时也是跳转语句, 跳转到循环体的开始。下一条目标代码编号为⑧, 返填回 SEM 栈顶③对应的目标代码。之后生成目标代码⑧JMP __, ?, 跳转到循环的开始, ? 待返填。此时 SEM 栈顶为①, 填⑧对应的目标代码中。

注意, 在实践中, 先返填, 后编写跳转指令, 即在生成⑧时, 先将⑧返填回 SEM 栈顶元素③中, 再将①填入⑧中。

如: $\text{if}(a > b) \ x = (a + b) * c; \ \text{else} \ x = 5 - a * b;$

★ 算法设计

※ 计算机目标代码生成过程示例:

B	QUAT[q]	OBJ[p]	RDL	SEM
→	(1)(wh __ __)			→
	(2)(> a(y) b(y) t ₁ (y))	①LD R, a ②GT R, b	t ₁	①
←	(3)(do t ₁ (n) __ __)	③FJ R, ? ⑨		③
→	(4)(+ a(y) b(y) t ₂ (y))	④LD R, a ⑤ADD R, b	t ₂	
	(5)(* t ₂ (n) c(y) x(y))	⑥MUL R, c	x	
←	(6)(we __ __)	⑦ST R, x ⑧JMP __, ? ①		
		⑨		

实践中，先返填，后编跳转指令！

图 9.15: 目标代码生成示例

1. 设当前四元式 q: (wh __ __), 这是一个基本块的入口, 也是一个循环的入口。
 PUSH(p); 将当前要生成的目标指令编号压栈。
2. 设当前四元式 q: (do B __ __), do 语句判断 B 是否为假, 为假则跳转。
 - (1) 当 RDL==0, 则 CODE(LD R, B ; FJ R, __); PUSH(p) 将指令编号或地址压入 SEM 栈, 等待返填。
 - (2) 当 RDL==B, 则
 - ① 若 B(y), 则 CODE(ST R, B ; FJ R, __); 否则 CODE(FJ R, __);
 - ② PUSH(p); RDL:=0;
 在 (1) 中未改变寄存器状态, 始终为空, (2) 中寄存器状态原为 B, 需要释放寄存器。(3)
 - 当 RDL==D (D != B), 则
 - ① 若 D(y), 则 CODE(ST R, D ; LD R, B ; FJ R, __); 否则 CODE(LD R, B ; FJ R, __);
 - ② PUSH(p); RDL:=0;
3. 设当前四元式 q: (we __ __), 作用 1: 标记结束; 作用 2: 跳转到 while 开始。
 - (1) 若 RDL==X 且 X(y), 则 CODE(ST R, X ;);
 - (2) RDL:=0;
 - (3) 返填转向地址: POP(p'); BACK(p', p+2)。

9.10, 在第 (6) 个四元式中, 已经生成目标代码⑦, 当前 p 为编号⑦, 执行 POP(p'), 根据 SEM 栈顶元素 p' (编号③), 找到对应的目标指令。下一步执行 BACK(p', p+2), p' 是待返填的目标指令编号③, 在编号为 p 的目标指令之后, 还有编号为 p+1 的无条件跳转目标指令 JMP ?, 跳转的目标位置在 JMP 指令之后, 即编号为 p+2, 因此将 p+2 返填回 p'。p+2 为⑨, 将⑨返填回③对应的目标代码。

(4) POP(p');

9.10中, 将 SEM 栈顶地址①弹入 p'。

(5) CODE(JMP __,p');

9.10中, 跳转到 p', 将①填入⑧对应的目标代码中。

9.3 一个简单代码生成器的实现

下面给出的简单代码生成器的主控程序流程图9.16。

首先进行预处理, 划分基本块。接下来, 对一个基本块进行处理, 生成变量活跃信息, 顺序遍历四元式, 编写目标指令, 直到到达基本块出口, 释放寄存器, 并跳转到取下一基本块, 继续执行上述操作, 直到取不到下一基本块, 表示处理结束。主控程序不唯一, 可以自行设计。

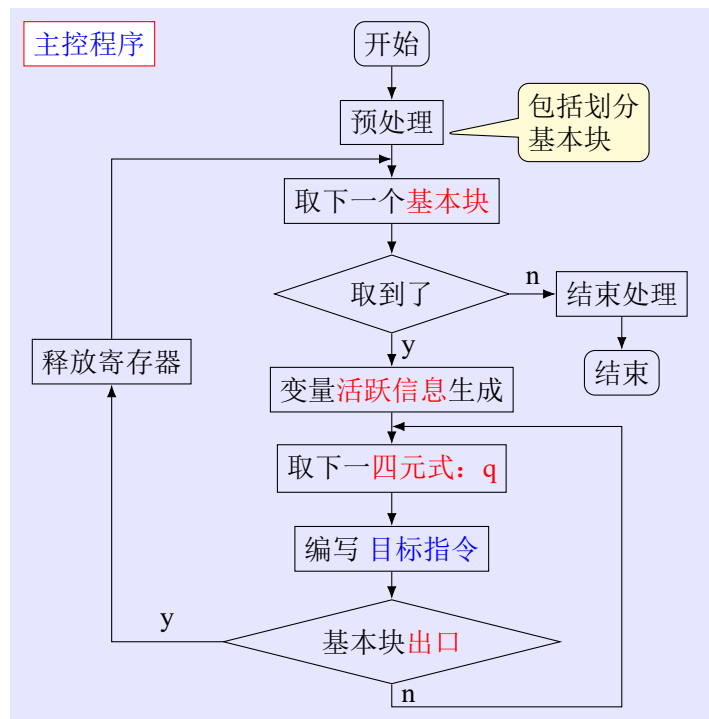


图 9.16: 主控程序流程图

下面介绍一个表达式与赋值四元式生成目标代码的例子, 这些语句都是顺序执行, 比较简单。

例 9.11 设有赋值语句 (四元式序列): $x = a * (a + b - d)$; $a = (a + b) / 2$; $y = 5$;

解:

第一步, 生成四元式序列, 这里只有一个基本块, 标注活跃信息。

第二步, 顺序遍历四元式, 编写目标指令。

第 (1) 个四元式, 给 t_1 分配寄存器, 生成目标代码①LD R, a ②ADD R, b。

第 (2) 个四元式, 给 t_2 分配寄存器, 运算对象 t_1 在寄存器中, 且 t_1 活跃, 需要保存到内存,

生成目标代码③ST R, t_1 ④SUB R, d 。

第(3)个四元式, 乘法运算可交换, 运算对象 t_2 在寄存器中, 且 t_2 非活跃, 直接进行计算, 生成目标代码⑤MUL R, a 。

第(4)个四元式, 给 a 分配寄存器, 运算对象 t_1 不在寄存器中, 原本寄存器中的 x 活跃, 需要保存到内存, 生成目标代码⑥ST R, x ⑦LD R, t_1 ⑧DIV R, 2。

第(5)个四元式, 给 y 分配寄存器, 寄存器中的 a 活跃, 需要保存到内存, 生成目标代码⑨ST R, a ⑩LD R, 5。到达基本块出口, 释放寄存器, 且 y 活跃, 需要保存到内存, 生成目标代码⑪ST R, y 。

第(3)个四元式中的 a 是非活跃的, 活跃的定义是从当前到下一个定义点之间有应用点, 而变量 a , 第(4)个四元式是它的定义点, 当前在第(3)个四元式, 中间没有其他应用点, 因此它是非活跃的。

填表结果如下图9.17:

B	QT[q]	OBJ[p]	RDL	SEM
	(1)(+ $a(y)$ $b(y)$ $t_1(y)$)	①LD R, a ②ADD R, b	t_1	
	(2)(- $t_1(y)$ $d(y)$ $t_2(y)$)	③ST R, t_1 ④SUB R, d	t_2	
	(3)(* $a(n)$ $t_2(n)$ $x(y)$)	⑤MUL R, a	x	
	(4)(/ $t_1(n)$ 2 $a(y)$)	⑥ST R, x ⑦LD R, t_1	a	
		⑧DIV R, 2		
	(5)(:= 5 _ $y(y)$)	⑨ST R, a ⑩LD R, 5	y	
	基本块出口	⑪ST R, y		

图 9.17: 例 9.11 寄存器分配填表结果

例 9.12 给定如下语句, 写出四元式序列, 进行 DAG 优化, 完成单寄存器目标代码生成。

if ($a > b$) $x = (a + b)/(c - d) + (a + b)$;

第一步, 生成四元式序列如下:

- ① ($>$, a , b , t_1)
- ② (if, t_1 , __, __)
- ③ (+, a , b , t_2)
- ④ (-, c , d , t_3)
- ⑤ (/, t_2 , t_3 , t_4)
- ⑥ (+, a , b , t_5)
- ⑦ (+, t_4 , t_5 , t_6)
- ⑧ (=, t_6 , __, x)
- ⑨ (end, __, __, __)

第二步，划分基本块，对第二个基本块，画 DAG 图如下9.18:

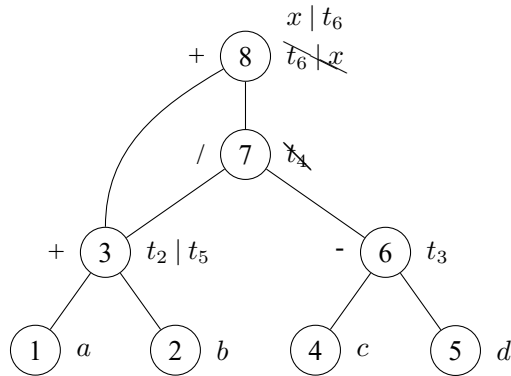


图 9.18: 例 9.12 DAG 图

第三步，写出优化后四元式如下图。括号中是活跃信息。

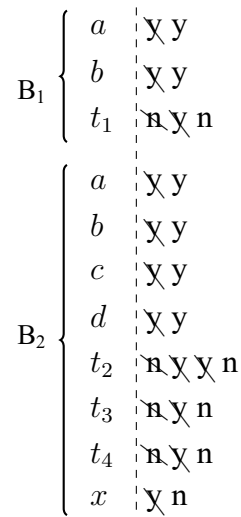


图 9.19: 例 9.12 活跃信息

- ① (>, a(y), b(y), t₁(y))
- ② (if, t₁(n), __, __)
- ③ (+, a(y), b(y), t₂(y))
- ④ (-, c(y), d(y), t₃(y))
- ⑤ (/, t₂(y), t₃(n), t₄(y))
- ⑥ (+, a(n), b(n), t₅(y))
- ⑦ (end, __, __, __)

第四步，生成目标指令。

目标指令

- ① LD R, a
- ② GT R, b
- ③ FJ R, ?
- ④ LD R, a
- ⑤ ADD R, b
- ⑥ ST R, t_2
- ⑦ LD R, c
- ⑧ SUB R, d
- ⑨ ST R, t_3
- ⑩ LD R, t_2
- ⑪ IV R, t_3
- ⑫ ADD R, t_2
- ⑬ ST R, x