第二章 PL/0 编译程序的实现

【课前思考】

复习第 1 章介绍的一个高级程序设计语言编译程序的功能和实现的步骤。编译程序就是一个语言的翻译程序，通常是把一种高级程序设计语言（称源语言）书写的程序翻译成另一种等价功能语言（称目标语言）的程序。换句话说，编译是指把一种用源语言表示的算法转换到另一种等价的用目标语言表示的算法。编译程序实现的必要步骤有词法、语法、语义分析和代码生成。此外必需有符号表管理程序和出错处理程序。本章介绍的 PL/0 编译程序的实现是用 PASCAL 语言书写的

【学习目标】

本章目的：以 PL/0 语言编译程序为实例,学习编译程序实现的基本步骤和相关技术，对编译程序的构造和实现得到一些感性认识和建立起整体概念，为后面的原理学习打下基础。

◇ 了解并掌握用语法图和扩充的巴科斯-瑙尔范式(EBNF)对 PL/0 语言的形式描述。

◇ 了解并掌握 PL/0 语言编译程序构造和实现的基本技术和步骤。

◇ 了解并掌握 PL/0 语言编译程序的目标程序在运行时数据空间的组织管理。

【学习指南】

◇ 要求读者阅读 PL/0 语言编译程序文本，了解一个编译程序构造的必要步骤和实现技术。一个编译程序的实现比较复杂，读懂一个典型的程序从设计思想到实现技术也有一定难度，特别是入门开始需要耐心。一但读懂，不仅了解编译程序的实现方法和技术，还可学到许多编程技巧和好的编程风格。

◇ 阅读 PL/0 语言编译程序文本时，应从整体结构开始逐步细化，弄清楚每个过程的功能和实现方法及过程之间的相互关系。

◇ 建议用一个 PL/0 源程序的例子为导引作为阅读 PL/0 语言编译程序文本的入门，然后再逐步全面读懂。

◇ 通过对 PL/0 语言编译程序某些指定功能的扩充，加深对编译程序构造步骤和实现技术的理解，并能在实践中应用。

【难 重 点】

重点：

◇ 弄清源语言(PL/0)、目标语言(类 pcode)与实现语言（C）这 3 个语言之间的关系和作用。

◇ 掌握用语法图和扩充的巴科斯-瑙尔范式(EBNF)对一个高级程序设计语言的形式描述。

◇ 了解 PL/0 语言编译程序的语法分析技术采用的是自顶向下递归子程序法。

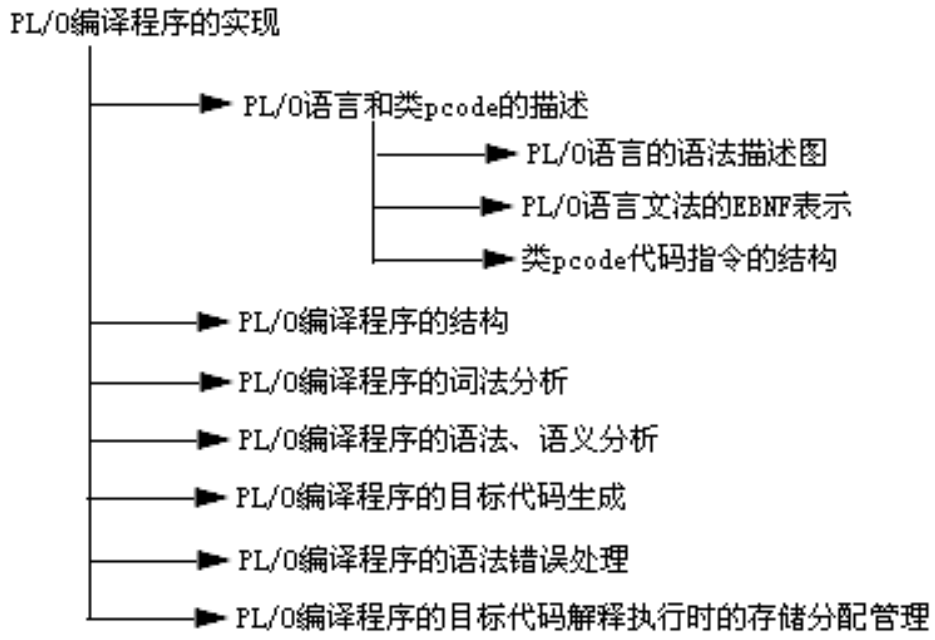
◇ 掌握 PL/0 语言编译程序的整体结构和实现步骤，并弄清词法分析、语法分析、语义分析、代码生成及符号表管理每个过程的功能和相互联系。

◇ 掌握 PL/0 语言编译程序的目标程序在运行时采用栈式动态存储管理的实现技术。

难点：

◇ 符号表管理起着编译期间和目标程序运行时信息联系的纽带，符号表的建立并不困难，但信息之间的关系往往需要反复学习才能理解。

【知识结构】



为了使读者在系统地学习本教材以下各章节之前，对编译程序的构造得到一些感性认识和初步了解，本章推荐了世界著名计算机科学家 N.Wirth 编写的"PL/0 语言的编译程序"，并对其实现过程作了概括的分析说明，作为读者阅读 PL/0 语言编译程序文本的提示。对 PL/0 语言文法的表示只给出语法图和扩充的巴科斯-瑙尔范式(EBNF)的描述形式，不作文法理论上的讨论。由于 PL/0 语言功能简单、结构清晰、可读性强，而又具备了一般高级语言的必须部分，因而 PL/0 语言的编译程序能充分体现一个高级语言编译程序实现的基本技术和步骤。因此，"PL/0 语言编译程序"是一个非常合适的小型编译程序的教学模型。所以，阅读"PL/0 语言编译程序"文本后，可帮助读者对编译程序的实现建立起整体概念。

2.1 PL/0 语言和类 pcode 的描述

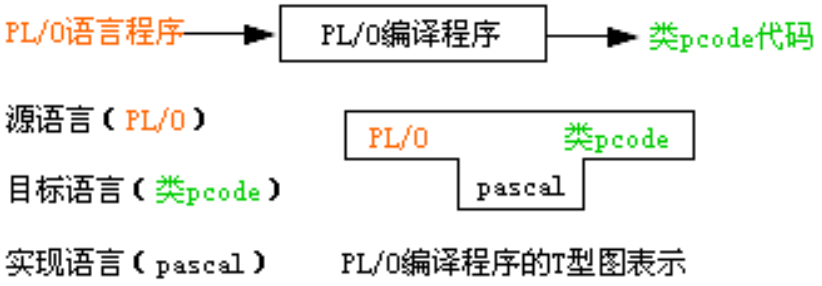
在上一章中已介绍过，编译程序的功能是把一种高级程序设计语言的程序翻译成某种等价功能的低级语言的程序。PL/0 语言的编译程序就是要把 PL/0 语言程序翻译成为一种称为类 pcode 的假想的栈式计算机汇编语言程序。这种汇编语言与机器无关，若需要在某一机器上实现 PL/0 语言程序，只需用机器上配置的任何语言对类 pcode 语言程序进行解释执行。那么 PL/0 语言是什么样的语言？类 pcode 语言又是什么样的结构？它们之间是如何映射的？只有在明确了这些问题之后，才能确定如何构造这个翻译程序。

就像一个翻译要把汉语翻译成英语，必须对汉语和英语的单词、语法结构都很精通，才有可能翻译得准确无误。另外，编译程序既然是为了完成这种功能的一个程序，就存在用什么语言来编写这个程序的问题。这些问题在本节将逐步介绍。

现对 PL/0 语言编译程序的功能用“T”型图（“T”型图将在第 13 章详细介绍）表示，并用图形概括描述 PL/0 编译程序的结构框架。

本节将用语法图和扩充的巴科斯-瑙尔范式（BACKUS-NAUR FORM）(EBNF)两种形式给出 PL/0 语言的语法描述。对类 pcode 代码给出指令的结构形式和含义。

巴科斯-瑙尔范式（BACKUS-NAUR FORM）是根据美国的 John W.Backus 与丹麦的 Peter Naur 命名的。



2.1.1 PL/0 语言的语法描述图

用语法图描述语法规则的优点是直观、易读。在图 2.1 的语法图中用椭圆和圆圈中的英文字表示终结符，用长方形内的中文字表示非终结符。所谓终结符，是构成语言文法的单词，是语法成分的最小单位，而每个非终结符也是一个语法成分。

但非终结符可由其它文法符号定义，终结符不能由其它文法符号定义。例如，程序是由非终结符'分程序'和终结符"."组成的串定义的。由于对某些非终结符可以递归定义，如图 2.1(b)、2.1 (c)、2.1 (e)中：分程序、表达式和语句。第一个非终结符 '程序'为文法的开始符号。

注意在书写语言程序时非终结符并不出现，程序是由终结符串构成。

图 2.1(a) 程序语法描述图

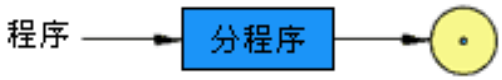


图 2.1(b) 分程序语法描述图

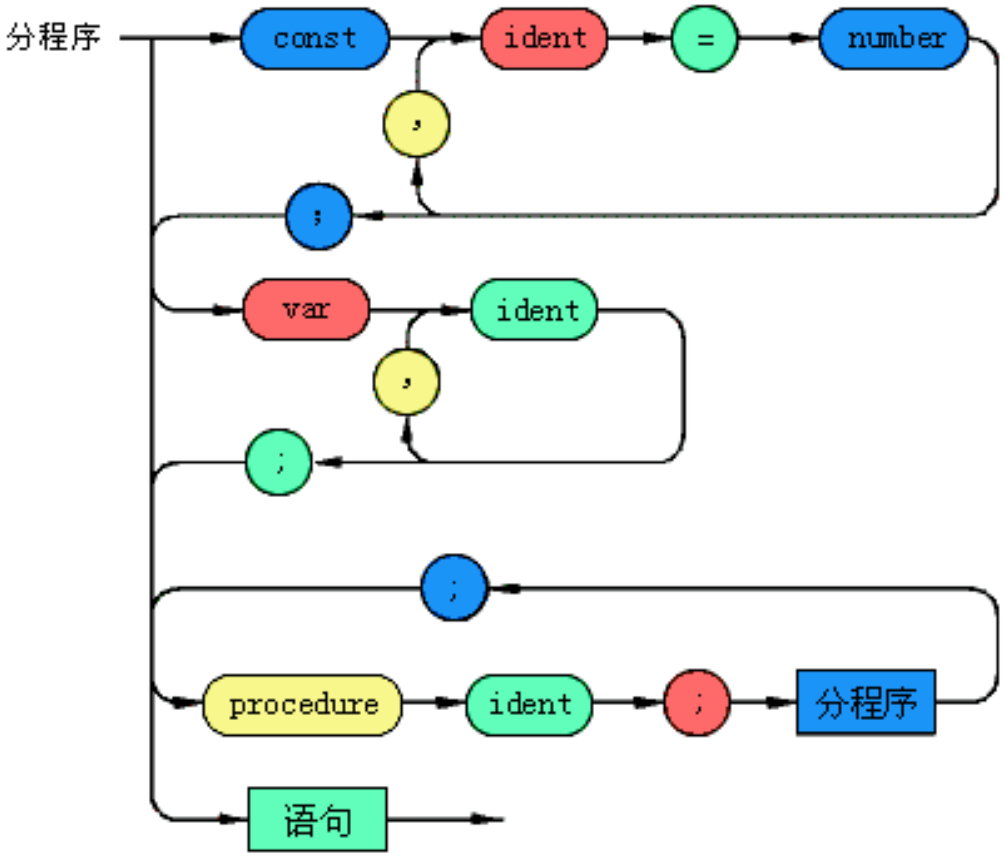


图 2.1(c) 语句语法描述图

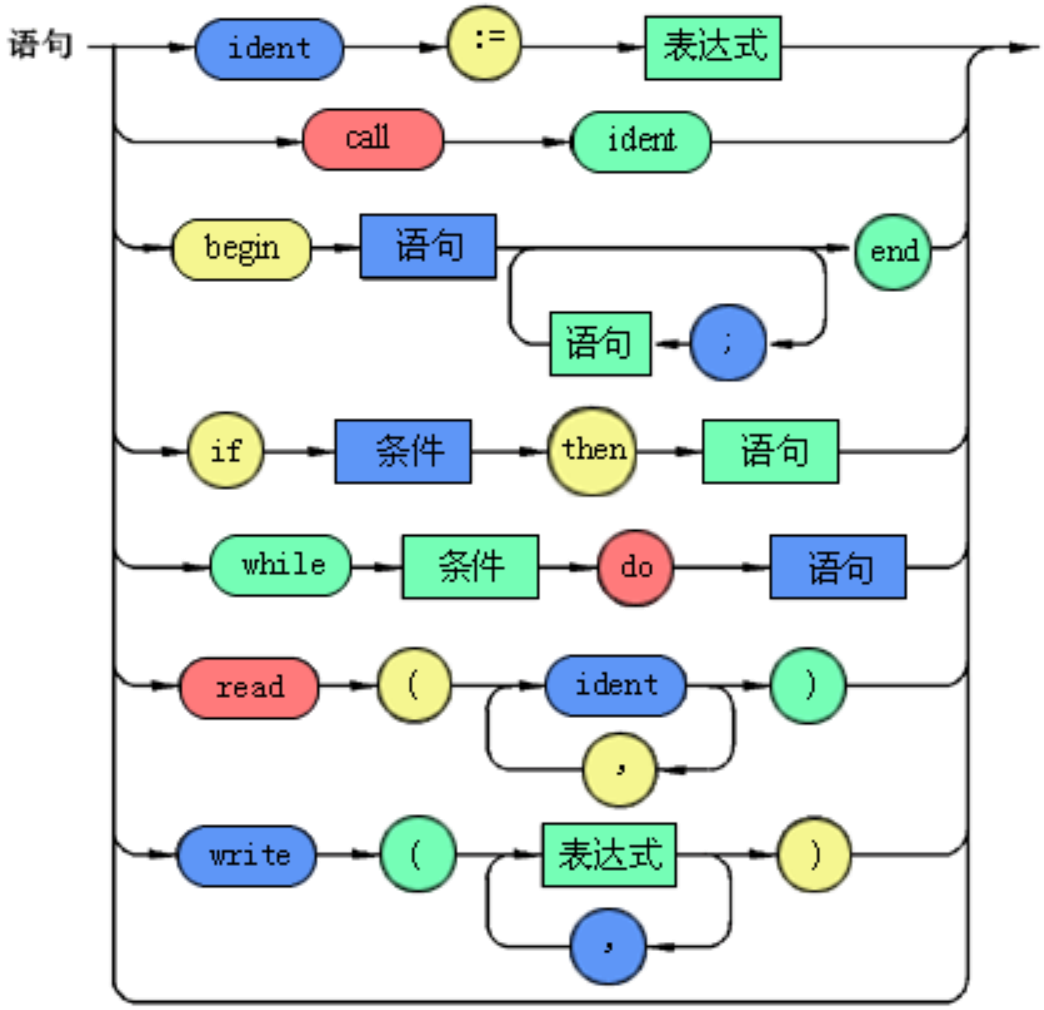


图 2.1(d) 条件语法描述图

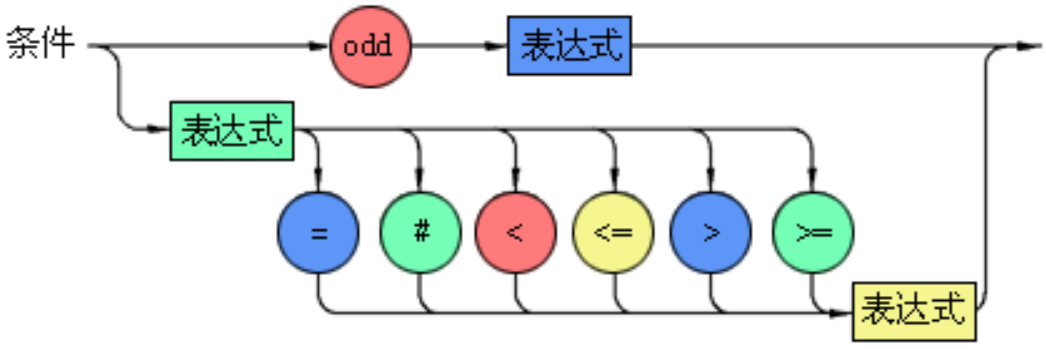


图 2.1(e) 表达式语法描述图

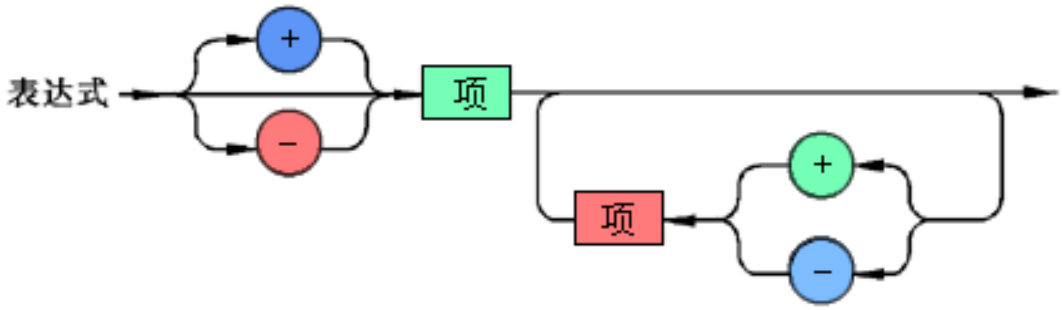


图 2.1(f) 项语法描述图

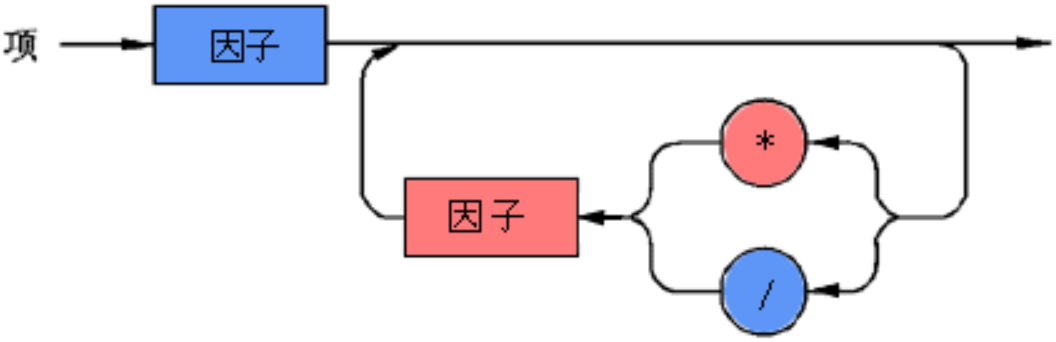
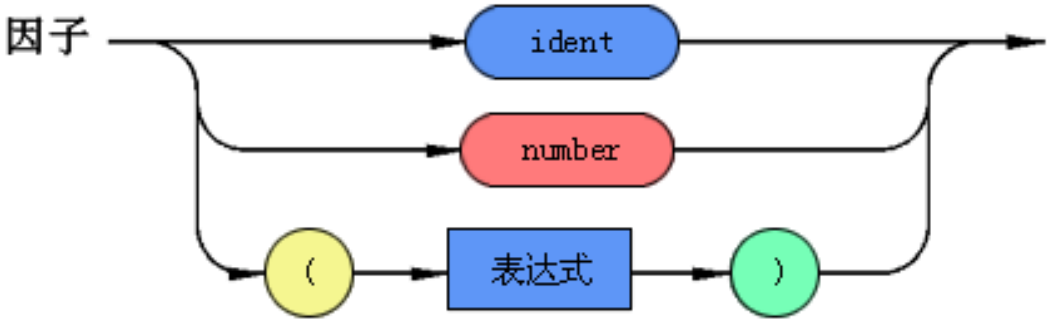


图 2.1(g) 因子语法描述图



画语法图时要注意箭头方向和弧度，语法图中应该无直角，如图 2.1 给出的 PL/0 语法描述图。沿语法图分析时不能走尖狐线。

2.1.2 PL/0 语言文法的 EBNF 表示

EBNF 表示的符号说明。

〈 〉：用左右尖括号括起来的中文字表示语法构造成分，或称语法单位，为非终结符。

::= ：该符号的左部由右部定义，可读作'定义为'。

| ：表示'或'，为左部可由多个右部定义。

{ } ：花括号表示其内的语法成分可以重复。在不加上下界时可重复 0 到任意次数，有上下界时为可重复次数的限制。

如：{*}表示*重复任意次，$\{*\}_8^3$ 表示*重复 3-8 次。

[ ] ：方括号表示其内的成分为任选项。

( ) ：表示圆括号内的成分优先。

例：用 EBNF 描述<整数>文法的定义 ：

<整数>∷=[+|-]<数字>{<数字>}

<数字>∷=0|1|2|3|4|5|6|7|8|9

或更好的写法

<整数>∷=[+|-]<非零数字>{<数字>}|0

<非零数字>∷=1|2|3|4|5|6|7|8|9

<数字>∷=0|<非零数字>

PL/0 语言文法的 EBNF 表示为：

〈程序〉∷=〈分程序〉.

〈分程序〉∷=[〈常量说明部分〉][〈变量说明部分〉][〈过程说明部分〉]〈语句〉

〈常量说明部分〉∷=CONST〈常量定义〉 {，〈常量定义〉}；

〈常量定义〉∷=〈标识符〉=〈无符号整数〉

〈无符号整数〉∷=〈数字〉{〈数字〉}

〈变量说明部分〉∷=VAR〈标识符〉{，〈标识符〉}；

〈标识符〉∷=〈字母〉{〈字母〉|〈数字〉}

〈过程说明部分〉∷=〈过程首部〉〈分程序〉{；〈过程说明部分〉}；

〈过程首部〉∷=PROCEDURE〈标识符〉；

〈语句〉∷=〈赋值语句〉|〈条件语句〉|〈当型循环语句〉|
〈过程调用语句〉|〈读语句〉|〈写语句〉|〈复合语句〉|〈空〉

〈赋值语句〉∷=〈标识符〉：=〈表达式〉

〈复合语句〉∷=BEGIN〈语句〉{；〈语句〉}END

〈条件〉∷=〈表达式〉〈关系运算符〉〈表达式〉|ODD〈表达式〉

〈表达式〉∷=[+|-]〈项〉{〈加法运算符〉〈项〉}

〈项〉∷=〈因子〉{〈乘法运算符〉〈因子〉}

〈因子〉∷=〈标识符〉|〈无符号整数〉|'('〈表达式〉')'

〈加法运算符〉∷=+|-

〈乘法运算符〉∷=*|/

〈关系运算符〉∷=#|=|＜|＜=|＞|＞=

〈条件语句〉∷=IF〈条件〉THEN〈语句〉

〈过程调用语句〉∷=CALL〈标识符〉

〈当型循环语句〉∷=WHILE〈条件〉DO〈语句〉

〈读语句〉∷=READ'('〈标识符〉{，〈标识符〉}')'

〈写语句〉∷=WRITE'('〈表达式〉{，〈表达式〉}')'

〈字母〉∷=a|b|…|X|Y|Z

〈数字〉∷=0|1|2|…|8|9

用语法图描述语言的语法与 EBNF 描述相比较：

语法图描述： 直观，易读，易写。

EBNF 表示：形式化强，易机器识别。

无论用语法图或 EBNF 作为描述程序设计语言语法的工具，对描述的语法可以检查哪些符号序列是所给语言的合法程序，一个程序设计语言如 C 或 JAVA，用户可用它写出成千上

万个不同的程序，但 C 或 JAVA 它们各自的语法只有一个，这就使得无穷的句子集可用有穷的文法（语法）描述。

练习：下面给出一个 PL/0 语言的程序，请学员对应 PL/0 语言的语法描述图与 EBNF ，检查该程序的语法是否正确。

```
CONST A=10;
  VAR B,C;
  PROCEDURE P;
    VAR D;
    PROCEDURE Q;
      VAR X;
      BEGIN
        READ(X);
        D:= D* C +X; WRITE(D);
        WHILE X#0 DO CALL P;
      END;
    BEGIN
      READ(D, C);
      CALL Q;
    END;
  BEGIN
    CALL P;
  END.
```

PL/0 编译程序的使用限制

◇ 标识符的有效长度是 10

◇ 数字最多为 14 位

◇ 过程最多可嵌套三层，可递归调用

◇ 标识符的作用域同 PASCAL，内层可引用包围它的外层定义的标识符（如：变量名，过程名，常量名）

2.1.3 类 pcode 代码指令的结构

PL/0 编译程序所产生的目标代码是一个假想栈式计算机的汇编语言，可称为类 PCODE 指令代码，它不依赖任何具体计算机，其指令集极为简单，指令格式也很单纯，其格式如下：

f	l	a

其中 f 代表功能码，l 表示层次差，也就是变量或过程被引用的分程序与说明该变量或过程的分程序之间的层次差。a 的含意对不同的指令有所区别，对存取指令表示位移量，而对其它的指令则分别有不同的含义，见下面对每条指令的解释说明。

目标指令有 8 条：

① LIT：将常量值取到运行栈顶。a 域为常数值。

② LOD：将变量放到栈顶。a 域为变量在所说明层中的相对位置,l 为调用层与说明层的层差值。

③ STO：将栈顶的内容送入某变量单元中。a,l 域的含意同 LOD 指令。

④ CAL：调用过程的指令。a 为被调用过程的目标程序入口地址，l 为层差。

⑤ INT：为被调用的过程(或主程序)在运行栈中开辟数据区。a 域为开辟的单元个数。

⑥ JMP：无条件转移指令，a 为转向地址。

⑦ JPC：条件转移指令，当栈顶的布尔值为非真时，转向 a 域的地址，否则顺序执行。

⑧ OPR：关系运算和算术运算指令。将栈顶和次栈顶的内容进行运算，结果存放在次栈顶，此外还可以是读写等特殊功能的指令，具体操作由 a 域值给出。（详见解释执行程序）。

类 pcode 代码指令的详细解释（指令功能表）

认识目标代码类 pcode

目标代码类 pcode 是一种假想栈式计算机的汇编语言。

指令格式：

f	l	a

f 功能码

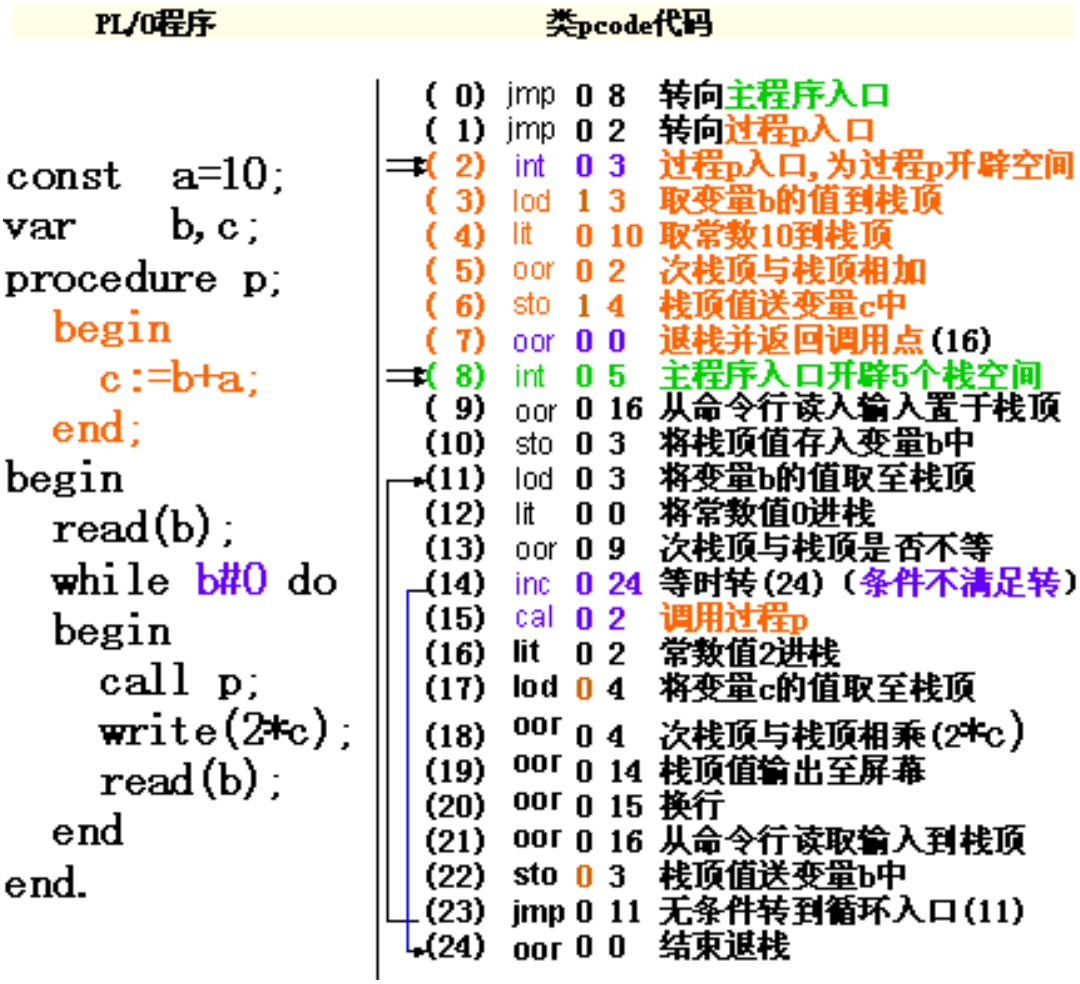
l 层次差 （标识符引用层减去定义层）

a 根据不同的指令有所区别

指令功能表	
LIT 0 a	将常数值取到栈顶，a 为常数值
LOD l a	将变量值取到栈顶，a 为偏移量，l 为层差
STO l a	将栈顶内容送入某变量单元中，a 为偏移量，l 为层差
CAL l a	调用过程，a 为过程地址，l 为层差
INT 0 a	在运行栈中为被调用的过程开辟 a 个单元的数据区
JMP 0 a	无条件跳转至 a 地址
JPC 0 a	条件跳转，当栈顶布尔值非真则跳转至 a 地址，否则顺序执行
OPR 0 0	过程调用结束后,返回调用点并退栈
OPR 0 1	栈顶元素取反
OPR 0 2	次栈顶与栈顶相加，退两个栈元素，结果值进栈
OPR 0 3	次栈顶减去栈顶，退两个栈元素，结果值进栈
OPR 0 4	次栈顶乘以栈顶，退两个栈元素，结果值进栈
OPR 0 5	次栈顶除以栈顶，退两个栈元素，结果值进栈
OPR 0 6	栈顶元素的奇偶判断，结果值在栈顶
OPR 0 7	
OPR 0 8	次栈顶与栈顶是否相等，退两个栈元素，结果值进栈
OPR 0 9	次栈顶与栈顶是否不等，退两个栈元素，结果值进栈
OPR 0 10	次栈顶是否小于栈顶，退两个栈元素，结果值进栈
OPR 0 11	次栈顶是否大于等于栈顶，退两个栈元素，结果值进栈
OPR 0 12	次栈顶是否大于栈顶，退两个栈元素，结果值进栈
OPR 0 13	次栈顶是否小于等于栈顶，退两个栈元素，结果值进栈

OPR 0 14	栈顶值输出至屏幕
OPR 0 15	屏幕输出换行
OPR 0 16	从命令行读入一个输入置于栈顶

一个 PL/0 程序与目标代码类 pcode 指令的映射例子:



2.2 PL/0 编译程序的结构

由 2.1 节可知，PL/0 语言可看成是 PASCAL 语言的子集，它的编译程序是一个编译解释执行系统。PL/0 的目标程序为假想栈式计算机的汇编语言，与具体计算机无关。PL/0 的编译程序和目标程序的解释执行程序都是用 PASCAL 语言书写的，因此 PL/0 语言可在配备 PASCAL 语言的任何机器上实现。其编译过程采用一趟扫描方式，以语法分析程序为核心，词法分析程序和代码生成程序都作为一个独立的过程，当语法分析需要读单词时就调用词法分析程序，而当语法分析正确需生成相应的目标代码时，则调用代码生成程序。此外，用表格管理程序建立变量、常量和过程标识符的说明与引用之间的信息联系。用出错处理程序对词法和语法分析遇到的错误给出在源程序中出错的位置和错误性质。当源程序编译正确时，PL/0 编译程序自动调用解释执行程序，对目标代码进行解释执行，并按用户程序要求输入数据和输出运行结果。其编译和解释执行的结构图如图 2.2(a)和 2.2(b)所示。

PL/0 的编译程序和目标程序的解释执行程序都是用 PASCAL 语言书写的，因此 PL/0 语言可在配置有 PASCAL 语言的任何机器上实现。读者也可用其它语言改写 PL/0 编译程序，也可以用另一种语言编写目标代码类 pcode 的解释执行程序。

PL/0 编译程序的编译过程是按源程序顺序进行分析的，常量变量说明部分不产生目标代码。

图 2.2(a) PL/0 编译程序的结构图

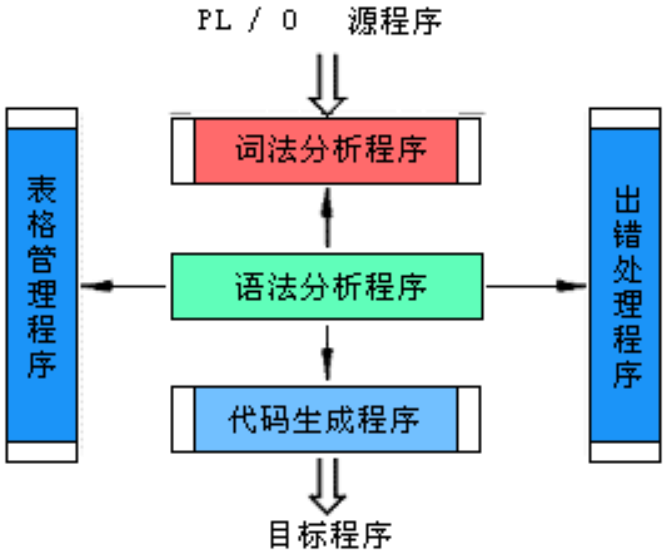
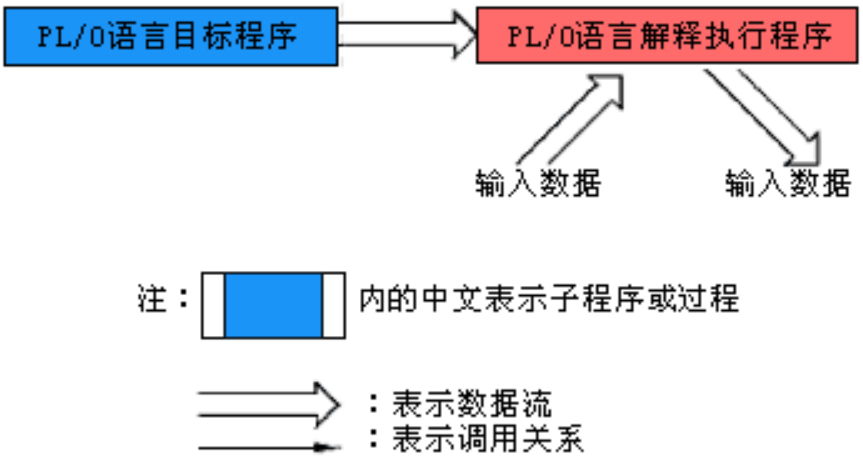
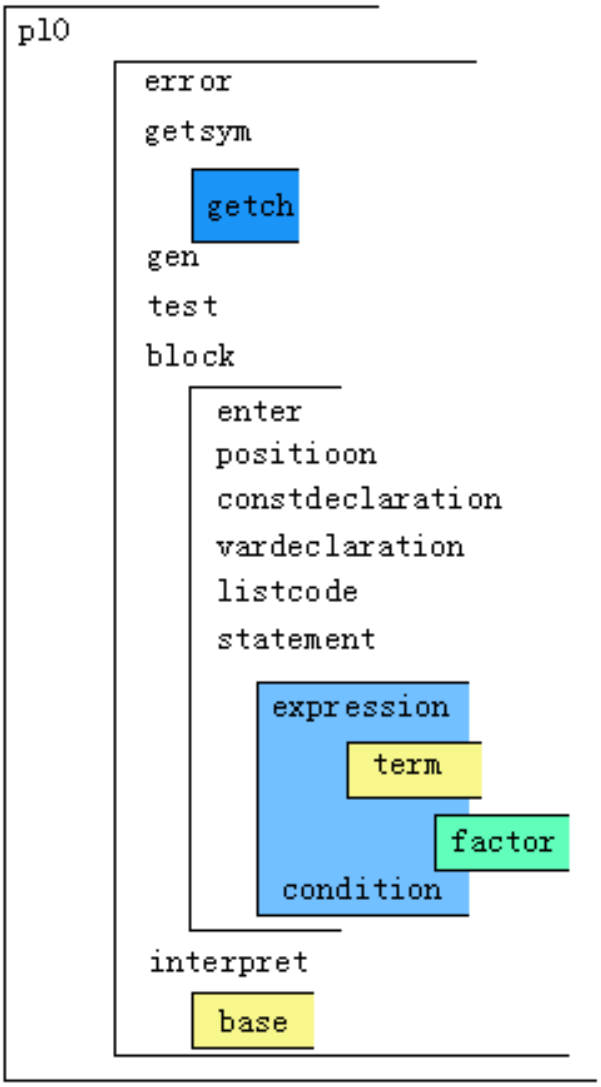


图 2.2(b) PL/0 的解释执行结构



PL/0 编译程序是用 PASCAL 语言书写的，整个编译程序(包括主程序)是由 18 个嵌套及并列的过程或函数组成，下面分别简要给出这些函数的功能及它们的层次结构。这些过程或函数的嵌套定义层次结构如图 2.3 所示。

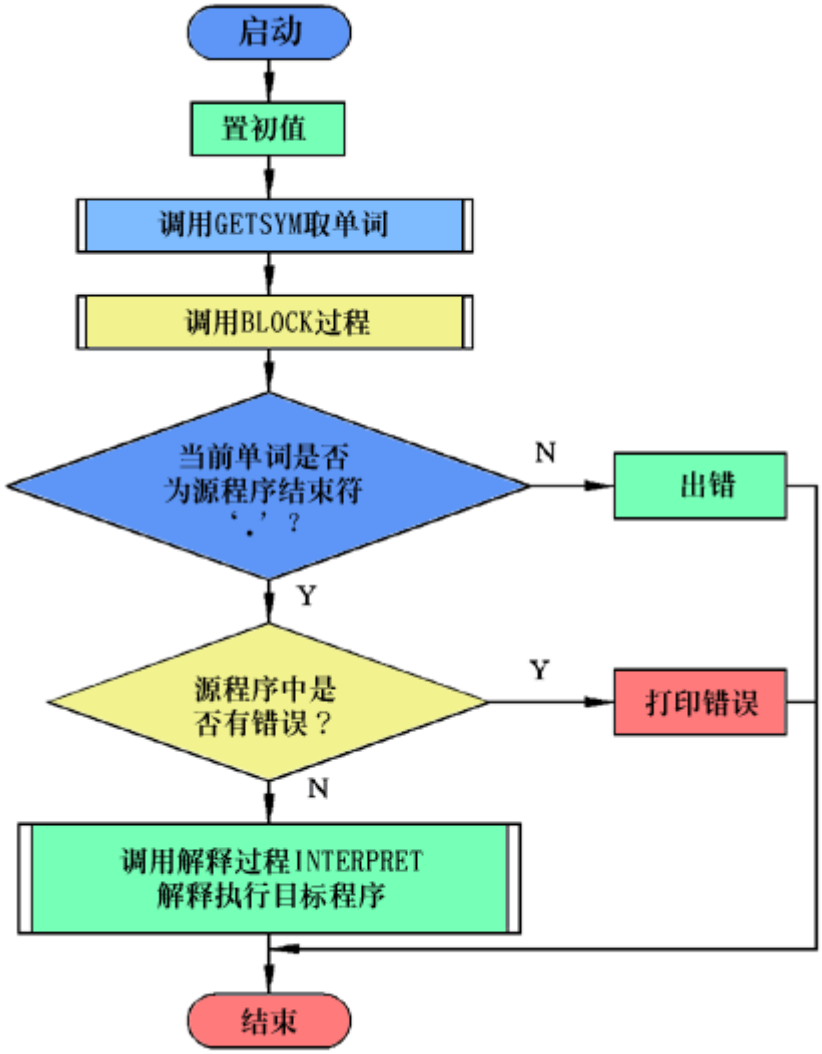
图 2.3 PL/0 编译程序过程与函数定义层次结构图



由于 PL/0 编译程序采用一趟扫描方法，所以语法分析过程 BLOCK 是整个编译过程的核心。下面我们将在图 2.4 中先给出编译程序的总体流程，以弄清 BLOCK 过程在整个编译程序中的作用。在流程图 2.4 中可以看出，主程序置初值后先调用读单词过程 GETSYM 取一个单词，然后再调用语法分析过程 BLOCK，直到遇源程序的结束符"．"为止。

语法分析过程 BLOCK 是整个编译过程的核心，是指开始由主程序调用 GETSYM 取一个单词，再调用语法分析过程 BLOCK， BLOCK 由当前单词根据语法规则再调用其它过程，如说明处理、代码生成或出错处理等过程进行分析，当分析完一个单词后，BLOCK 再调用 GETSYM 取下一个单词，一直重复到当前单词为结束符"．"表明源程序已分析结束。若未取到结束符"．"，而源程序已没有输入符号，这时表明源程序有错误，无法再继续分析。

图 2.4 PL/0 编译程序总体流程图



2.3 PL/0 编译程序的词法分析

PL/0 的词法分析程序 GETSYM(图 2.5)是一个独立的过程，其功能是为语法分析提供单词用的，是语法分析的基础，它把输入的字符串形式的源程序分割成一个个单词符号。为此 PL/0 编译程序设置了**三个全程量的公用单元**如下：

SYM：存放每个单词的类别，用内部编码形式表示。

ID：存放用户所定义的标识符的值。即标识符字符串的机内表示。

NUM：存放用户定义的数。

单词的种类有五种。

基本字：也可称为保留字或关键字，如 BEGIN，END，IF，THEN 等。

运算符：如：+、-、*、／、：=、#、＞=、＜=等。

标识符：用户定义的变量名、常数名、过程名。

常数：如：10，25，100 等整数。

界符：如：'，'、'.'、'；'、'('、')'等。

如果我们把基本字、运算符、界符称为语言固有的单词，而对标识符、常数称为用户定义的单词。那么经词法分析程序分出的单词，对语言固有的单词只给出类别存放在 SYM 中，而对用户定义的单词(标识符或常数)既给类别又给值，其类别放在 SYM 中，值放在 ID 或 NUM

中，全部单词种类由编译程序定义的纯量类型 SYMBOL 给出，也可称为语法的词汇表。如下面提到的 IFSYM，THENSYM，IDENT，NUMBER 均属 SYMBOL 中的元素。

词法分析过程图 2.5 的 GETSYM 框图对应程序见 PL/0 编译程序文本中 procedure getsym，其中对标识符和关键字（保留字）的识别方式为：

当识别到字母开头的字母数字串时，先查关键字表。若查不到则为标识符，查到则为关键字。PL/0 编译程序文本中主程序开始对关键字表置初值如下：

关键字表为：

word[1]:='begin '； word[2]:='call '；

...

word[13]:='write '；

每个数组元素的字符长度为 10，不满 10 个字符时，以空格补满。

查到时找到关键字相应的内部表示为：

Wsym[1]:=beginsym; wsym[2]:=callsym；

…

wsym[13]:=writesym；

PL/0 编译程序文本中开始对类型的定义中给出单词定义为：

type symbol=(nul,ident,number,plus,…,varsym,procsym)；

定义单词是纯量/枚举类型，又定义了 3 个全程量为：

sym:symbol;

id:alfa;

num:integer;

因此词法分析程序 GETSYM 将完成下列任务：

(1) 滤空格：空格在词法分析时是一种不可缺少的界符，而在语法分析时则是无用的，所以必须滤掉。

(2) 识别保留字：设有一张保留字表。对每个字母打头的字母、数字字符串要查此表。若查着则为保留字，将对应的类别放在 SYM 中。如 IF 对应值 IFSYM，THEN 对应值为 THENSYM。若查不着，则认为是用户定义的标识符。

(3) 识别标识符：对用户定义的标识符将 IDENT 放在 SYM 中，标识符本身的值放在 ID 中。

(4) 拼数：当所取单词是数字时，将数的类别 NUMBER 放在 SYM 中，数值本身的值存放在 NUM 中。

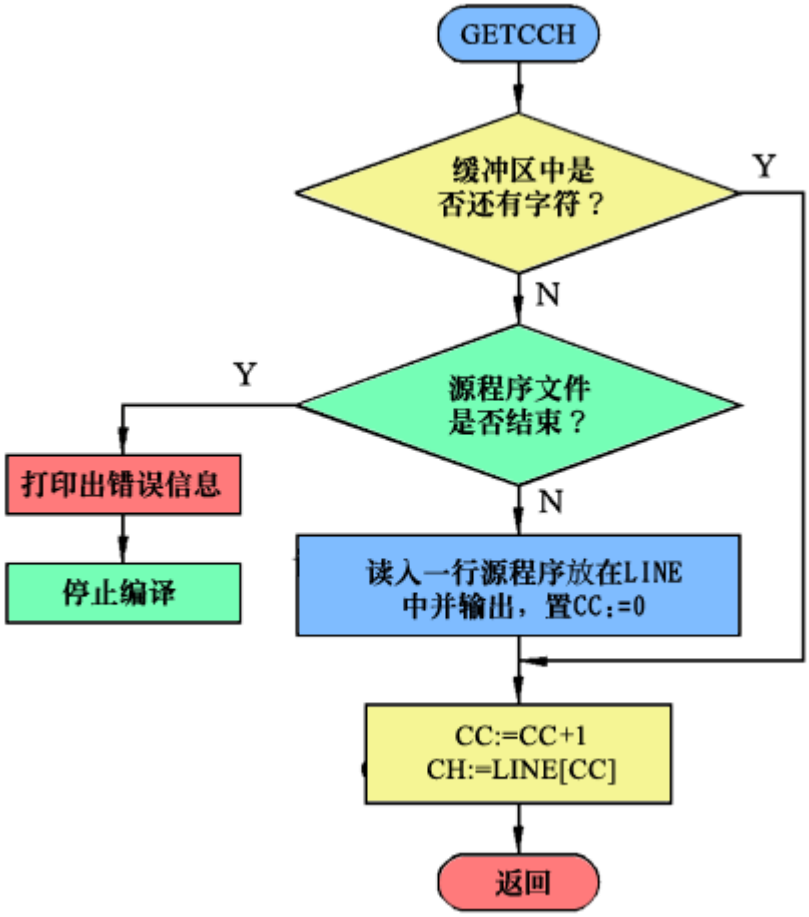
(5) 拼复合词：对两个字符组成的算符

如：＞=、：=、＜= 等单词，识别后将类别送 SYM 中。

(6) 输出源程序：为边读入字符边输出(可输出在文件中)。

由于一个单词往往是由一个或几个字符组成的，所以在词法分析过程 GETSYM 中又定义了一个取字符过程 GETCH(见图 2.6)，由词法分析需要取字符时调用。

图 2.6 取字符过程 GETCH



GETCH 所用单元说明：

CH： 存放当前读取的字符，初值为空。

LINE： 为一维数组，其数组元素是字符，界对为 1：80。用于读入一行字符的缓冲区。

LL 和 CC 为计数器，初值为 0。

GETSYM 流程图的工作单元说明：

A：一维数组，数组元素为字符，界对[1：10]。

ID：同 A。

WORD：保留字表，一维数组，数组元素是以字符为元素的一维数组。界对为[1：13]。查表方式采用二分法。

单个字符对应的单词表的建立是，首先在主程序中定义了下标为字符的数组 ssym，数组 ssym 的元素为单词，主程序开始对下标为字符的所有数组元素置初值为 nul，对 PL/0 语言用到的单个字符为单词的，将其字符作为数组下标的元素置初值为对应的单词。如：

ssym['+']:=plus; ssym['-']:=minus;

…

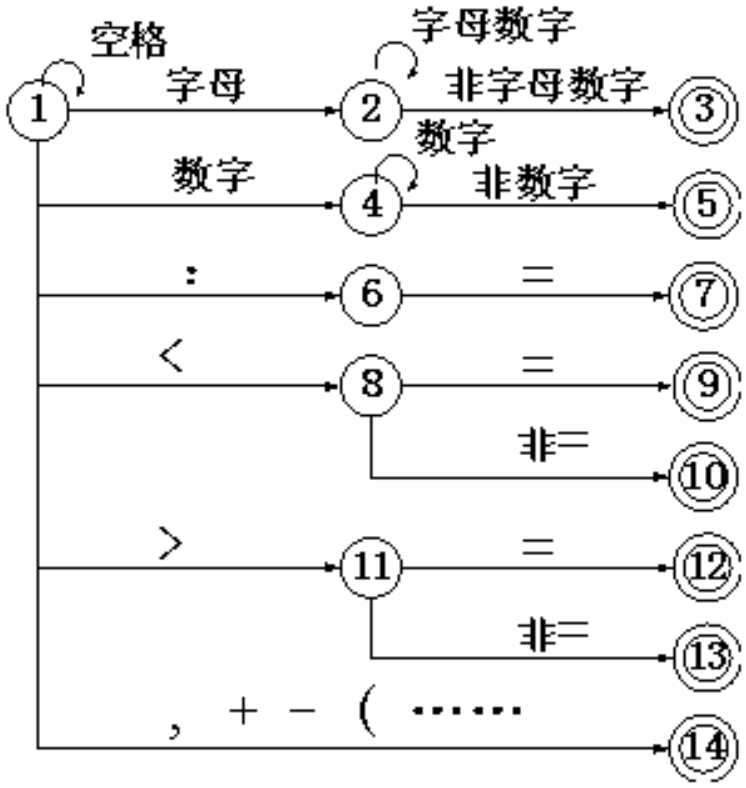
ssym[';']:=semicolon;

使用状态转换图实现词法分析程序的设计方法

词法分析程序的设计---使用状态转换图实现

○表示状态，对应每个状态编一段程序，每个状态调用取字符程序，根据当前字符转到不同的状态，并做相应操作。

◎表示终态，已识别出一个单词。



2.4 PL/0 编译程序的语法分析

PL/0 编译程序语法、语义分析是整个编译程序设计与实现的核心部分,要求学员努力学习掌握实现技术和方法。现分别说明语法分析实现的主要思想方法和语义分析的实现。

语法分析的任务是识别由词法分析给出的单词符号序列在结构上是否符合给定的文法规则。PL/0 语言的文法规则已在 2.1 节中给出。本节将以语法图描述的语法形式为依据，给出语法分析过程的直观思想。

PL/0 编译程序的语法分析采用了自顶向下的递归子程序法。

什么是自顶向下的语法分析？

可形象地对该程序自顶向下构造一棵倒挂着的语法分析树，其构造方法是：

（1） 由开始符号非终结符'程序'作为分析树的根结点，由非终结符'程序'规则的右部为子结点。

（2） 对分析树中的每个非终结符结点，选择它规则的一个右部为子结点构造分析树的下一层。

（3） 重复（2）直到分析树中的末端结点只有终结符。

（4） 若分析树中的末端结点从左到右连接的终结符号串刚好是输入的程序终结符号串，则说明所给程序在语法上是正确的。

可用下面简单的 PL/0 程序为例构造其语法分析树

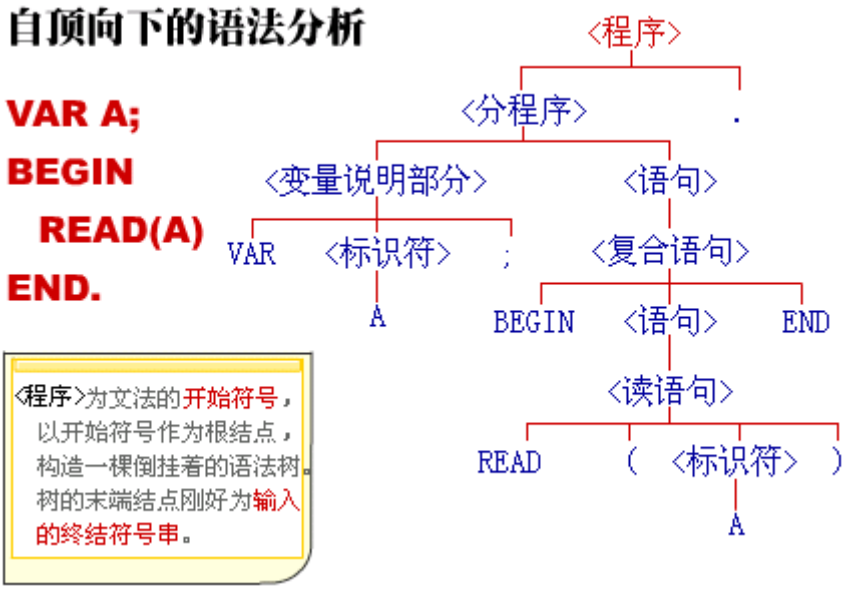
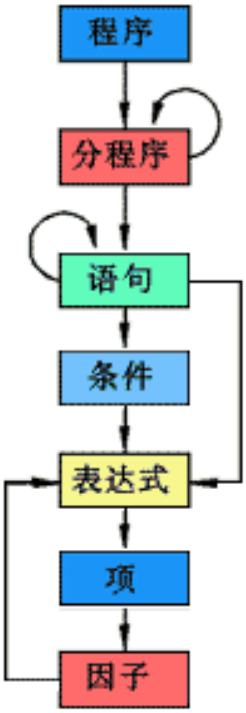


图 2.7 PL/0 语法调用关系图



粗略地说：自顶向下的递归子程序法就是对应每个非终结符语法单元，编一个独立的处理过程(或子程序)。语法分析从读入第一个单词开始由非终结符'程序'即开始符号出发，沿语法描述图箭头所指出的方向进行分析。当遇到非终结符时，则调用相应的处理过程，从语法描述图看也就进入了一个语法单元，再沿当前所进入的语法描述图的箭头方向进行分析，当

遇到描述图中是终结符时，则判断当前读入的单词是否与图中的终结符相匹配，若匹配，则执行相应的语义程序(就是翻译程序)。再读取下一个单词继续分析。遇到分支点时将当前的单词与分支点上的多个终结符逐个相比较，若都不匹配时，可能是进入下一非终结符语法单位或是出错。

如果一个 PL/0 语言的单词序列在整个语法分析中，都能逐个得到匹配，直到程序结束符'.'，这时就说所输入的程序是正确的。对于正确的语法分析做相应的语义翻译，最终得出目标程序。

以上所说语法分析过程非常直观粗浅，实际上应用递归子程序法构造语法分析程序时，对文法有一定的要求和限制，这个问题我们将在第 5 章详细讨论。

此外，从 PL/0 的语法描述图中可以清楚地看到，当对 PL/0 语言进行语法分析时，各个非终结符语法单元所对应的分析过程之间必须存在相互调用的关系。这种调用关系可用图 2.7 表示。也可称为 PL/0 语法的依赖图，在图中箭头所指向的程序单元表示存在调用关系，从图中不难看出这些子程序在语法分析时被直接或间接递归调用。

由图 2.7 PL/0 语法调用关系图可以看出对分程序和语句为直接递归调用，对表达式为间接递归调用。

例：如何用递归子程序法实现表达式的语法分析

现用 2.1 中给出的表达式语法图进行语法分析，语法图如下：

图 2.1(e) 表达式语法描述图

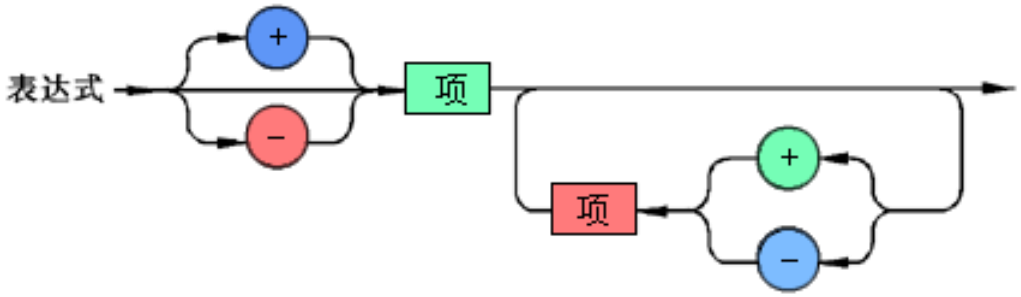


图 2.1(f) 项语法描述图

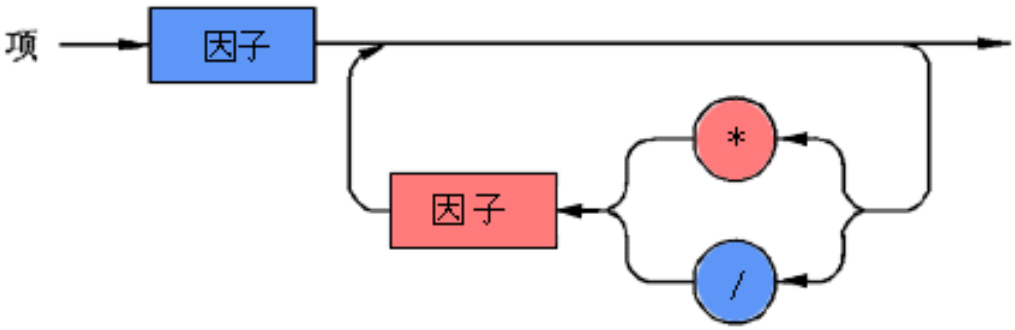
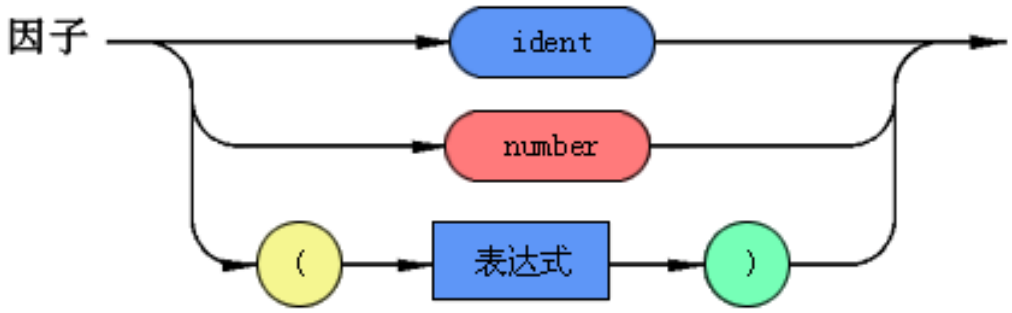


图 2.1(g) 因子语法描述图



表达式的 EBNF

〈表达式〉::=[+|-]〈项〉{（+|-）〈项〉}

〈项〉::=〈因子〉{（*|/）〈因子〉}

〈因子〉::=〈标识符〉|〈无符号整数〉|‘（’〈表达式〉‘）’

〈表达式〉的递归子程序实现

```
procedure expr;
   begin
      if sym in [ plus, minus ] then
         begin
            getsym; term;
         end
      else term;
      while sym in [plus, minus] do
         begin
            getsym; term;
         end
   end;
```

〈项〉的递归子程序实现

```
procedure term;
begin
   factor;
   while sym in [ times, slash ] do
   begin
      getsym; factor;
   end
end;
```

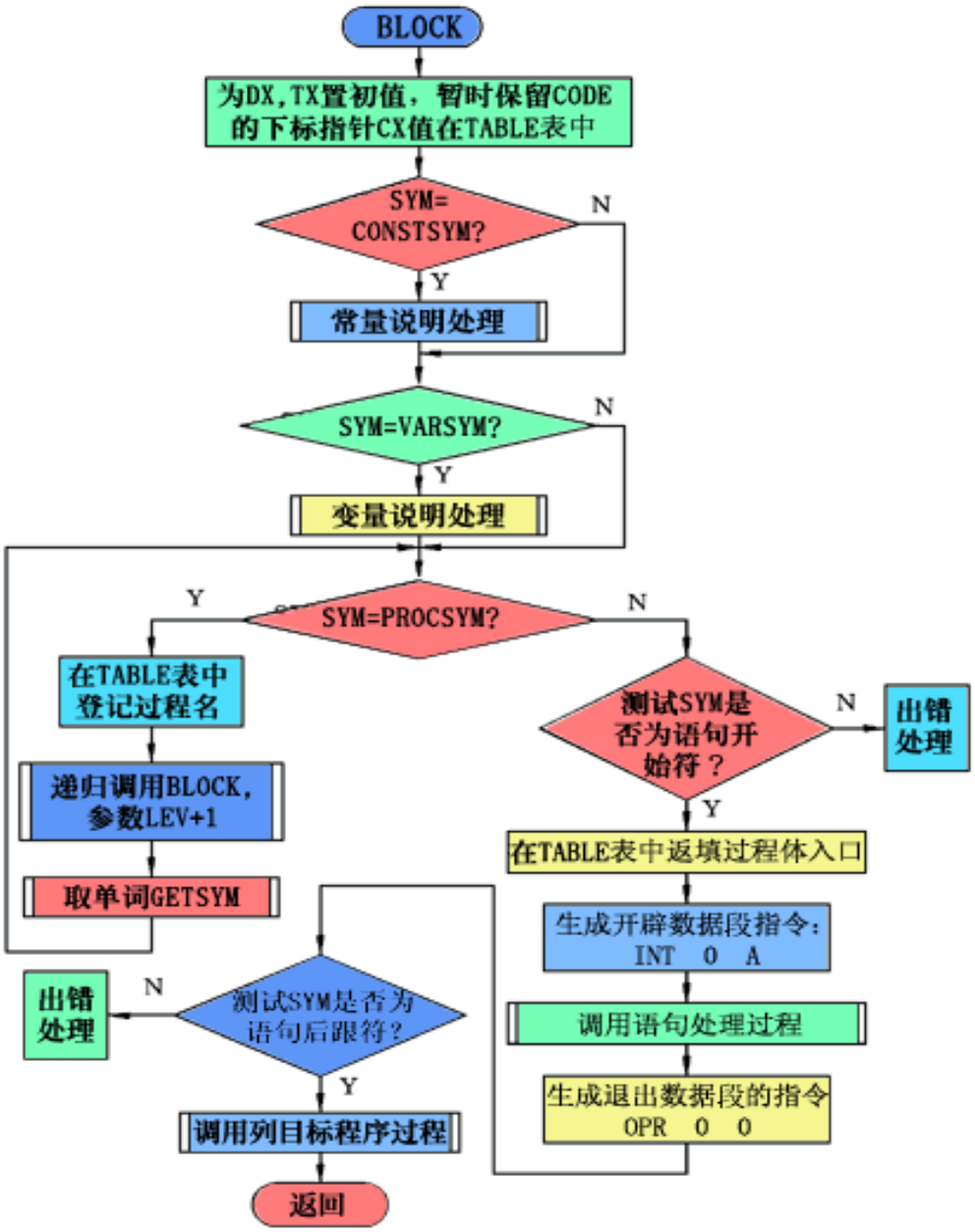
〈因子〉的递归子程序实现

```
procedure factor;
begin
   if sym <> ident then
      begin
         if sym <> number then
            begin
               if sym = ‘(‘ then
                  begin
```

```
                getsym;
                expr;
                if sym = ‘)’ then
                   getsym
                else error
             end
          else error
       end
    end
end;
```

语法分析程序除总控外主要有两大部分的功能，即对说明部分的处理和对程序体部分的处理，也就是在语法单元中的分程序功能。在 PL/0 编译程序中对应的过程为 BLOCK，其流程图如图 2.8 所示。

图 2.8 程序 BLOCK 过程的流程图



PL/0 编译程序语法、语义分析的的核心程序是 BLOCK 过程，在 BLOCK 过程内又定义了许多嵌套及并列的过程。

在过程 BLOCK 内对说明部分及程序体部分的分析说明如下：

(1) 说明部分的分析

由于 PL/0 语言允许过程调用语句，且允许过程嵌套定义。因此每个过程应有过程首部以定义局部于它自己过程的常量、变量和过程标识符，也称局部量。每个过程所定义的局部量只能供它自己和它自己定义的内过程引用。对于同一层并列过程的调用关系是先定义的可以被后定义的引用，反之则不行。

说明部分的处理任务就是对每个过程的说明对象造名字表，填写所在层次、标识符的属性和分配的相对位置等。标识符的属性不同时，所需要填写的信息也有所不同。登录信息是调用 ENTER 过程完成的。

说明部分的处理对主程序看成是第 0 层过程，主程序定义的过程为第 1 层，随着嵌套的深度增加而层次数加大。PL/0 允许最大层次为 3。

所造名字表放在全程量一维数组 TABLE 表中。TX 为索引表的指针，表中的每个元素为记录型数据。LEV 给出层次，DX 给出每层局部量当前已分配到的相对位置，可称地址指示器，每说明完一个变量后 DX 指示器加 1。

PL/0 编译程序文本中对名字表定义有：

说明类型的定义：

type object= (constant, variable, procedur)

（定义为纯量/枚举类型）

名字表的定义：

table:array[0..txmax] of record

name:alfa;

case kind:object of

constant:(val:integer);

variable:procedur:(level,adr,size:integer);

end;

例如：一个过程的说明部分为：

CONST A=35,B=49;

VAR C，D，E;

PROCEDURE P;

VAR G, ...

对常量，变量和过程说明分析后，在 TABLE 表中的信息如表 2.2 所示。

在说明处理后 TABLE 表中的信息对于过程名的 ADR 域，是在过程体的目标代码生成后再反填过程体的入口地址。例中在处理 P 过程的说明时对 LEV 就增加 1。在 P 过程中的变量名的层次为 LEV+1 后的值。对过程还有一项数据 SIZE，是记录该过程体运行时所需的数据空间。至于在造表和查表的过程中，如何保证每个过程的局部量不被它的外层引用，请读者阅读完 PL/0 编译程序后自己总结。

表 2.2 TABLE 表中的信息

	NAME	KIND	LEVEL/VAL	ADR	SIZE
TXo→	A	CONSTANT	35	DX	
	B	CONSTANT	49	DX+1	4

	C D E P	VARIABLE VARIABLE VARIABL EPROCEDUR	LEV LEV LEV LEV	DX+2 过程 p 的入口（待填）	
TX→	G …	VARIABLE …	LEV+1 …	ADR：DX …	

TABLE 表的表头索引 TX 和层次单元 LEV 都以 BLOCK 的参数形式出现。在主程序调用 BLOCK 时实参值都为 0。每个过程中变量的相对起始位置在 BLOCK 内置初值 DX：=3。

例如：对变量定义的语法处理

语法：<变量说明部分>: := var <标识符>{， <标识符>}；

程序：

```
if sym=varsym then
   begin
      getsym;
      repeat
         vardeclaration;(*变量说明处理*)
         while sym=comma do
            begin
               getsym;
               vardeclaration
            end;
         if sym=semicolon then
            getsym
         else error(5)
      until sym<>ident;
   end;
```

变量说明处理程序：

```
Procedure vardeclaration;
   begin
      if sym = ident then
         begin
            enter(variable);( *调用过程 enter 造名字表*）
            getsym
         end
      else error(4)
   end(*vardeclaration*);
```

(2) 过程体的分析

程序的主体是由语句构成的。处理完过程的说明后就处理由语句组成的过程体，从语法上要对语句逐句分析。当语法正确时就生成相应语句功能的目标代码。当遇到标识符的引用时就调用 POSITION 函数查 TABLE 表，看是否有过正确的定义，若已有，则从表中取相应的有关信息，供代码的生成用。若无定义则出错。

例：READ 语句的语法语义分析处理

语句的语法:<读语句>::=READ'('<标识符>{，<标识符>}')'；

程序：

```
if sym=readsym then
   begin
      getsym;
      if sym<>lparen then error(34)
      else
      repeat
         getsym;
         if sym=ident then i:=position(id)  （* 查找标识符表 *）
         else i:=0;
         if i=0 then error(35)
         else
            with table[i] do  （* 查到了标识符产生两条目标代码 *）
               begin
                  gen(opr,0,16);
                  gen(sto,lev-level,adr)（*其中 Lev 为引用层，level 为定义层 *)
               end;
            getsym
      until sym<>comma;
      if sym<>rparen then
      begin
         error(33);
         while not(sym in fsys)  （*出错处理*）
         do getsym
      end
   else getsym  （*正常出口*）
end
```

2.5 PL/0 编译程序的目标代码结构和代码生成

编译程序的目标代码是在分析程序体时生成的，在处理说明部分时并不生成目标代码，而当分析程序体中的每个语句时，当语法正确则调用目标代码生成过程以生成与 PL/0 语句等价功能的目标代码，直到编译正常结束。

PL/0 语言的代码生成是由过程 GEN 完成的。GEN 过程有三个参数，分别代表目标代码的功能码、层差和位移量(对不同的指令含意不同)。生成的代码顺序放在数组 CODE 中。CODE 为一维数组，数组元素为记录型数据。每一个记录就是一条目标指令。CX 为指令的指针，由 0 开始顺序增加。实际上目标代码的顺序是内层过程的排在前边，主程序的目标代码在最后。下面我们给出一个 PL/0 源程序和对应的目标程序的清单。

```
Run p10
Input file? TEST
List object code? Y
    const a=10；
```

```
    var b,c;
    procedure p;
      begin
        c:= b+a
      end;
```

The object code of procedure p:

```
 1          int         0          3
 3          lod         1          3
 4          lit         0          10
 5          opr         0          2
 6          sto         1          4
 7          opr         0          0
```

```
   begin
      read(b);
      while b＃0 do
        begin
          call p; write(2*c); read(b)
        end
    end.
```

The object code of main program:

```
 8          int         0          5
 9          opr         0          16
 10         sto         0          3
 11         lod         0          3
 12         lit         0          0
 13         opr         0          9
 14         jpc         0          24
 15         cal         0          2
 16         lit         0          2
 17         lod         0          4
 18         opr         0          4
 19         opr         0          14
 20         opr         0          15
 21         opr         0          16
 22         sto         0          3
 23         jmp         0          11
 24         opr         0          0
```

```
start p10
? 2   24
? 4   28
? 0
end p10
```

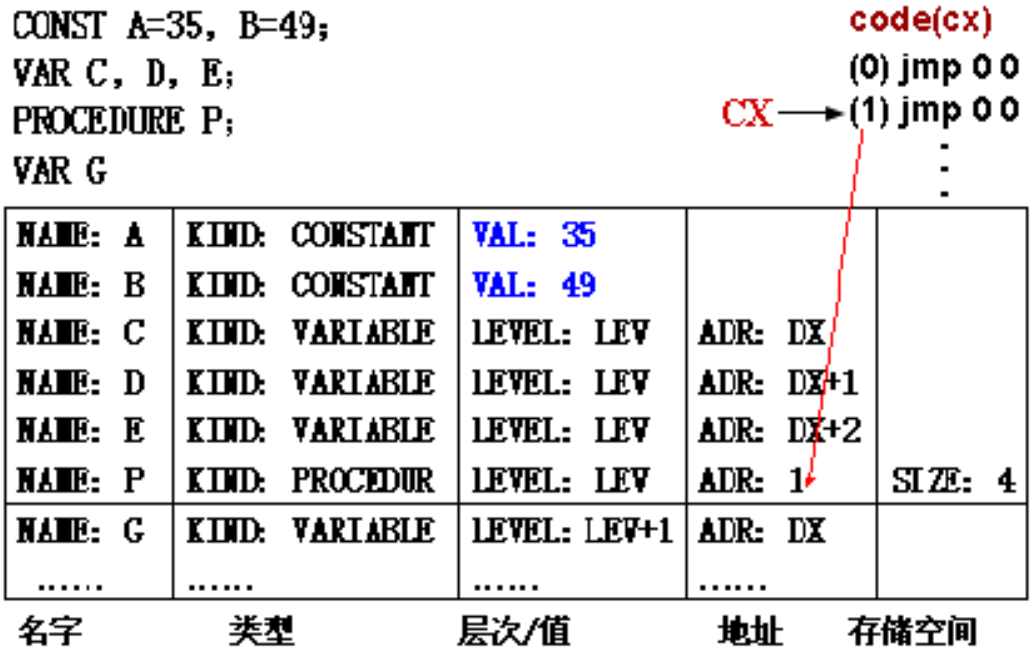
PL/0 编译程序的目标代码生成是由 GEN 过程完成的，当语法分析正确则调用目标代码生成过程以生成与 PL/0 语句等价功能的目标代码，直到编译正常结束。 除了过程说明部分外，

变量和常量的说明都不产生目标代码。在 block 入口处生成一条(jmp,0,0)指令，作为过程体入口指令，该指令的第 3 区域的'0'需分析到过程体入口时才返填。目标代码生成时所用到的变量地址和层差等信息是由名字表 table 提供的，而名字表的信息是在说明时填写的。在代码生成时查名字表，这就是表格管理的作用。这些信息之间的连接关系学员必须弄清。下面对一些重要程序段给予扼要的解释。（gen 过程的实现很简单不再解释）

对分程序体人口的处理（见程序文本 block 的过程体）

```
begin (*block*)
  dx:=3;
  tx0:=tx;  (*保留当前 table 表指针值,实际为过程名在 table 表中的位置*)
  table[tx].adr:=cx;(*保留当前 code 指针值到过程名的 adr 域*)
  gen(jmp,0,0);
```

记录过程在 code 的入口到 table 中的 adr 域如下表所示：



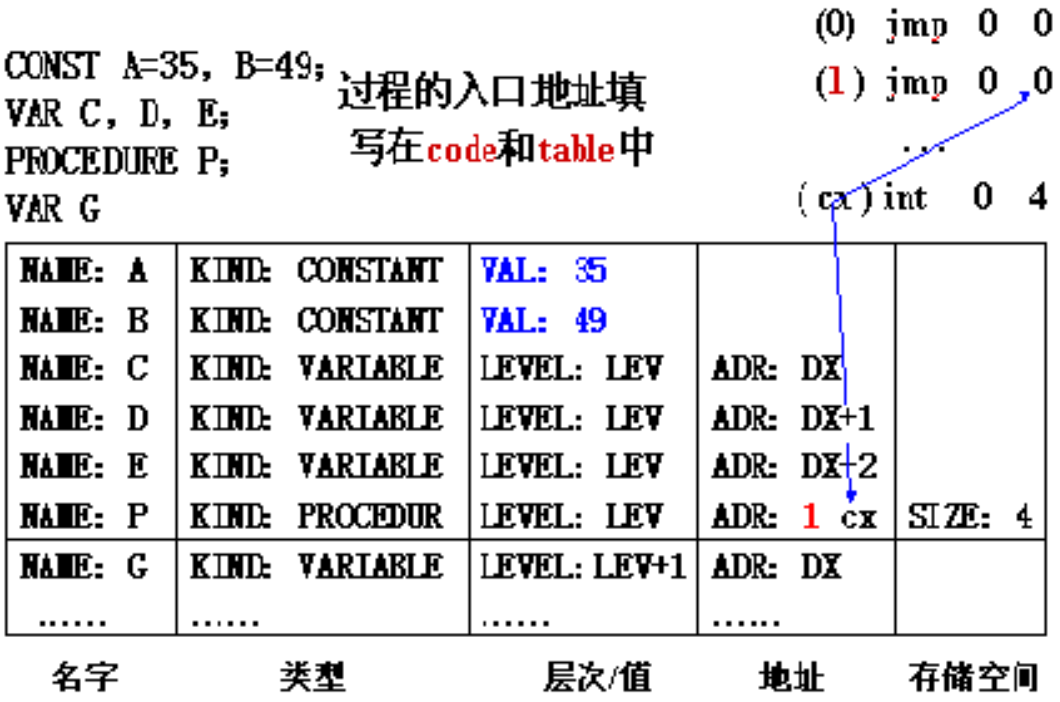
（*生成转向过程体入口的指令，该指令的地址为 cx 已保留在过程名的 adr 域，真正的过程体入口地址，等生成过程体入口的指令时，再由 table[tx].adr 中取出 cx 将过程体入口返填到 cx 所指目标代码，即:（jmp,0,0）的第 3 区域，同时填到 table[tx].adr 中*)

过程体入口时的处理

```
code[table[tx0].adr].a:=cx;（cx 为过程入口地址，“回填”写在 code 中）
with table[tx0] do
  begin
    adr:=cx;  （过程的入口填写在 table 表的过程名中）
    size:=dx;  （过程需要的空间填写在 table 中）
  end;
cxo:=cx;  （保留过程在 code 中的入口地址在输出目标代码时用）
gen(int,0,dx);（生成过程入口指令）
```

请特别注意 dx、 tx、 cx 的作用和如何处理信息之间的连接关系。

table 表格管理



2.6 PL/0 编译程序的语法错误处理

编写一个程序，往往难于一次成功，常常会出现各种类型的错误。一般有语法错、语义错及运行错。出错的原因是多方面的，这就给错误处理带来不少困难。就语法错来说，任何一个编译程序在进行语法分析遇到错误时，总不会就此停止工作，而是希望能准确地指出出错位置和错误性质并尽可能进行校正，以便使编译程序能继续工作。但对所有的错误都做到这样的要求是很困难的，主要困难在校正上，因为编译程序不能完全确定程序人员的意图。例如在一个表达式中，圆括号不配对时，就不能确定应补在何处。有时由于校正得不对反而会影响到后边，导致出现误判错误的情况。因此编译程序只能采取一些措施，对源程序中的错误尽量查出，加以修改，以便提高调试速度。

PL/0 编译程序对语法错误的处理采用两种办法：

(1) 对于一些易于校正的错误，如丢了逗号、分号等，则指出出错位置，并加以校正。校正的方式就是补上逗号或分号。

(2) 对某些错误编译程序难于确定校正的措施，为了使当前的错误不致影响整个程序的崩溃，把错误尽量局限在一个局部的语法单位中。这样就需跳过一些后面输入的单词符号，直到读入一个能使编译程序恢复正常语法分析工作的单词为止。具体做法是：当语法分析进入以某些关键字(保留字)或终结符集合为开始符的语法单元时，通常在它的入口和出口处，调用一个测试程序 TEST(见图 2.9)。例如：语句的开始符是 begin，if，while，call，read，write；说明的开始符是 var，const，procedure；因子的开始符是"("，ident，number。当语法分析进入这样的语法单元前，可用测试程序检查当前单词符号是否属于它们开始符号的集合，若不是则出错。

请读者对照图 2.1 各语法描述图直观地找出每个非终结符语法单元的开始符号集合，与表 2.3 进行比较，验证对开始符号集合理解的正确性。对于一个文法符号的开始符号集合的形式定义将在第 5 章详细介绍。现给出 PL/0 文法部分非终结符语法单元的开始符号和后继符号的集合。

另外由于 PL/0 编译程序采用自顶向下的分析方法，一个语法单元分析程序调用别的语法单元的分析程序时，以参数形式(文本中以 FSYS 定义为单词符号集合作为形参)给出被调用的语法分析程序出口时合法的后继单词符号集合(如表 2.3 所给出)，在出口处也调用测试程序。若当前单词符号是属于所给集合，则语法分析正常进行，否则出错。单词符号集合 FSYS 参数是可传递的，随着调用语法分析程序层次的深入，FSYS 的集合逐步补充合法单词符。

图 2.9 TEST 测试过程流程图

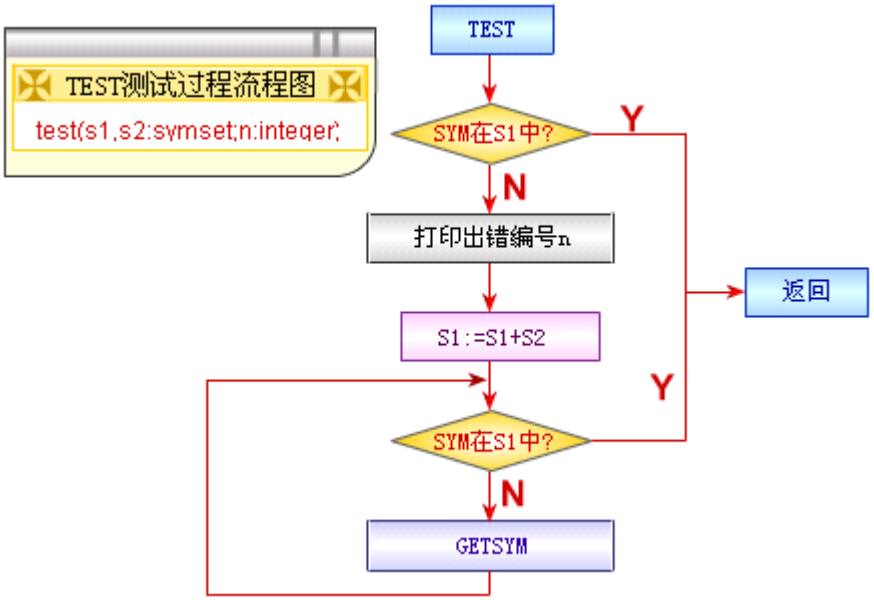
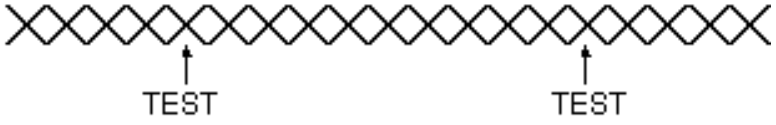


表 2.3 PL/0 文法非终结符的开始符号与后继符号集合表

非终结符名	开始符号集合	后继符号集合
分程序	const var procedure ident if call begin while read write	. ;
语句	ident call begin if while read write	. ; end
条件	odd + - (ident number	then do
表达式	+ - (ident number	. ;) rop end then do
项	ident number (	. ;) rop + - end then do
因子	ident number (	. ; + - * / end then do

*注： 表 2.3 中'rop'表示关系运算符集合,如=，＃，＜，＜=，＞，＞=。

PL/0 编译程序采用对语法分析程序入口和出口处都调用测试程序检查当前单词符号进入和退出该语法单位的合法性，可用下图形象地描述。



在进入某个语法单位时，调用 TEST，检查当前符号是否属于该语法单位的开始符号集合。若不属于，则滤去开始符号和后继符号集合外的所有符号。

在语法单位分析结束时，调用 TEST，检查当前符号是否属于调用该语法单位时应有的后继符号集合。若不属于，则滤去后继符号和开始符号集合外的所有符号。

TEST 测试过程有三个参数，它们的含意是：

① S1：当语法分析进入或退出某一语法单元时当前单词符号应属于的集合，它可能是一个语法单元开始符号的集合或后继符号的集合。

② S2：在某一出错状态时，可恢复语法分析继续正常工作的补充单词符号集合。因为当语法分析出错时，即当前单词符号不在 S1 集合中，为了继续编译，需跳过后边输入的一些单词符号，直到当前输入的单词符号是属于 S1 或 S2 集合的元素。

③ n:整型数，出错信息编号。

为了进一步明确 S1，S2 集合的含意，现以因子(FACTOR)的语法分析程序为例，在过程 FACTOR 的入口处调用了一次 TEST 过程，它的实参 S1 是因子开始符号的集合(文本中的 FACBEGSYS)。S2 是每个过程的形参 FSYS 调用时实参的传递值。当编译程序第一次调用 BLOCK 时，FSYS 实参为[·]与说明开始符和语句开始符集合的和。以后随着调用语法分析程序层次的深入逐步增加。如调用语句时增加了[；]和[endsym],在表达式语法分析中调用项时又增加了[+]和[-]，而在项中调用因子时又增加了[*]和[/]，这样在进入因子分析程序时即使当前符号不是因子开始符，出错后只要跳过一定的符号，遇到当时输入的单词符号在 FSYS 中或在因子开始符号集合中，均可继续正常进行语法分析。而在因子过程的出口处也调用了测试程序，不过这时 S1 和 S2 实参恰恰相反。说明当时的 FSYS 集合的单词符号都是因子正常出口时允许的单词符号，而因子的开始符号为可恢复正常语法分析的补充单词符号。

从 PL/0 编译程序文本中因子过程的处理片段说明上述问题。

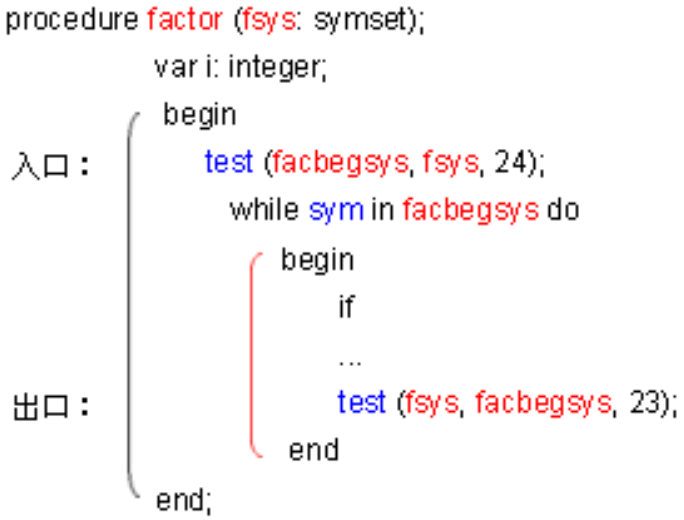
（1）PL/0 编译程序文本中给出关于某些语法单位开始符号集合的定义为：

```
symset=set of symbol;  （见 PL/0 文本类型说明部分）
declbegsys, statbegsys, facbegsys:symset;
declbegsys:=[constsym,varsym,procsym];（见 PL/0 文本主程序置初值部分）
statbegsys:=[beginsym,callsym,ifsym,whilesym,readsym,writesym];
facbegsys:=[ident,number,lparen];
```

（2）后继符号集合 fsys 作为参数传递（见 PL/0 文本相应过程的说明部分）

```
procedure test(s1,s2:symset; n:integer);
procedure block(lev,tx:integer; fsys:symset);
procedure statement(fsys:symset);
procedure expression (fsys:symset);
procedure term (fsys:symset);
procedure factor (fsys:symset);
```

（3）因子过程的处理片段（见 PL/0 文本的 factor 过程）



（4）由于后继符号集合 fsys 作为参数传递，随着调用语法分析程序层次的深入后继符号集合逐步增加，但对调用同一个过程所需增加的后跟符与调用位置有关。例如： 在 write 语句和 factor 中调用 expression(fsys);所增加的后继符号不完全相同。

· write 语句的语法：<写语句> ::= write(<exp>{,<exp>});

处理在（ ）内调用 expression 时在 fsys 中应增加 rparen,comma。

expression([rparen,comma]+fsys);

· factor 的语法：<因子>::=...|'(' exp ')

在处理（ ）内调用 expression 时在 fsys 中应增加 rparen。

expression([rparen]+fsys);

然而 PL/0 编译程序对测试程序 TEST 的调用有一定的灵活性。对语义错误，如标识符未加说明就引用，或虽经说明，但引用与说明的属性不一致。这时只给出错误信息和出错位置，编译工作可继续进行。而对运行错，如溢出，越界等，只能在运行时发现，由于 PL/0 编译程序的功能限制无法指出运行错在源程序中的错误位置。

这节最后我们给出 PL/0 语言的出错信息表。

出错编号	出错原因
1	常数说明中的"="写成"：="。
2	常数说明中的"="后应是数字。
3	常数说明中的标识符后应是"="。
4	const ,var, procedure 后应为标识符。
5	漏掉了'，'或'；'。
6	过程说明后的符号不正确(应是语句开始符，或过程定义符)。
7	应是语句开始符。
8	程序体内语句部分的后跟符不正确。
9	程序结尾丢了句号'.'。
10	语句之间漏了'；'。
11	标识符未说明。
12	赋值语句中，赋值号左部标识符属性应是变量。
13	赋值语句左部标识符后应是赋值号'：='。
14	call 后应为标识符。
15	call 后标识符属性应为过程。
16	条件语句中丢了'then'。

17 丢了'end"或'；'。
18 while 型循环语句中丢了'do'。
19 语句后的符号不正确。
20 应为关系运算符。
21 表达式内标识符属性不能是过程。
22 表达式中漏掉右括号')'。
23 因子后的非法符号。
24 表达式的开始符不能是此符号。
31 数越界。
32 read 语句括号中的标识符不是变量。

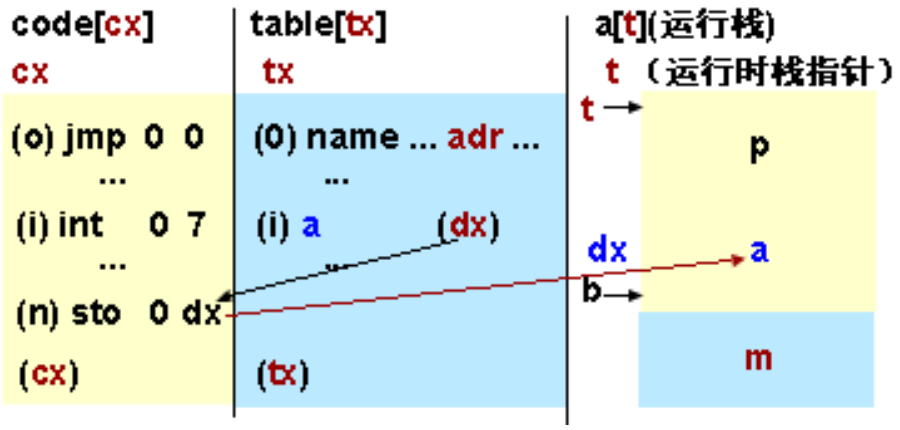
2.7 PL/0 编译程序的目标代码解释执行时的存储分配

当源程序经过语法分析，如果未发现错误时，由编译程序调用解释程序，对存放在 CODE 中的目标代码 CODE[0]开始进行解释执行。当编译结束后，记录源程序中标识符的 TABLE 表已没有作用。

因为计算每个变量在运行栈中相对本过程基地址的偏移量 dx 的值，放在 table 表中的 adr 域，生成目标代码时再从 adr 域中取出基地址的偏移量 ，放在 code 中的 a 域。

因此数据空间只需以数组 CODE 存放的只读目标程序和运行时的数据栈 S。S 是由解释程序定义的一维整型数组。由于 PL/0 语言的目标程序是一种假想的栈式计算机的汇编语言，仍用 PASCAL 语言解释执行。解释执行时的数据空间 S 为栈式计算机的存储空间。遵循后进先出规则，对每个过程(包括主程序)当被调用时，才分配数据空间，退出过程时，则所分配的数据空间被释放。

变量在 code [cx] 、table[tx]和 s[t]之间的信息联系



解释程序还定义了 4 个寄存器。

(1) I：指令寄存器。存放着当前正在解释的一条目标指令。

(2) P：程序地址寄存器。指向下一条要执行的目标程序的地址(相当目标程序 CODE 数组的下标)。

(3) T：栈顶寄存器。由于每个过程当它被运行时，给它分配的数据空间(下边称数据段)可分成两部分。

静态部分：包括变量存放区和三个联系单元(联系单元的作用见后)。

动态部分：作为临时工作单元和累加器用。需要时随时分配，用完后立即释放。栈顶寄存器 T 指出了当前栈中最新分配的单元(T 也是数组 S 的下标)。

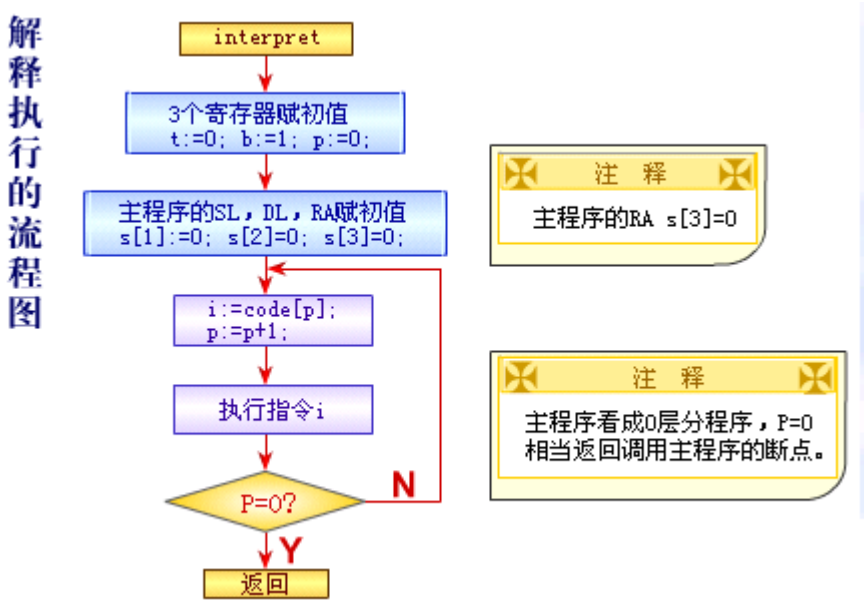
(4) B：基址寄存器。指向每个过程被调用时，在数据区 S 中给它分配的数据段起始地址，也称基地址。

为了实现对每个过程调用时给它分配数据段，也就是对即将运行的过程所需数据段进栈；过程运行结束后释放数据段，即该数据段退栈；以及嵌套过程之间对标识符引用的寻址问题。每个过程被调用时，在栈顶分配三个联系单元，这三个单元存放的内容分别为：

(1) SL：静态链：它是指向定义该过程的直接外过程(或主程序)运行时最新数据段的基地址。

(2) DL：动态链：它是指向调用该过程时正在运行过程的数据段基地址。

(3) RA：返回地址：记录调用该过程时目标程序的断点，即当时的程序地址寄存器 P 的值。也就是调用过程指令的下一条指令的地址。



例：若有程序结构为：

```
…
 procedure A:
    …
       procedure B:
          …
             procedure C:
                程序体 C
                   …
                   call B:


                   …
           程序体 B
              …
              call C:
              …
    程序体 A
       …
       call B:
```

…

主程序

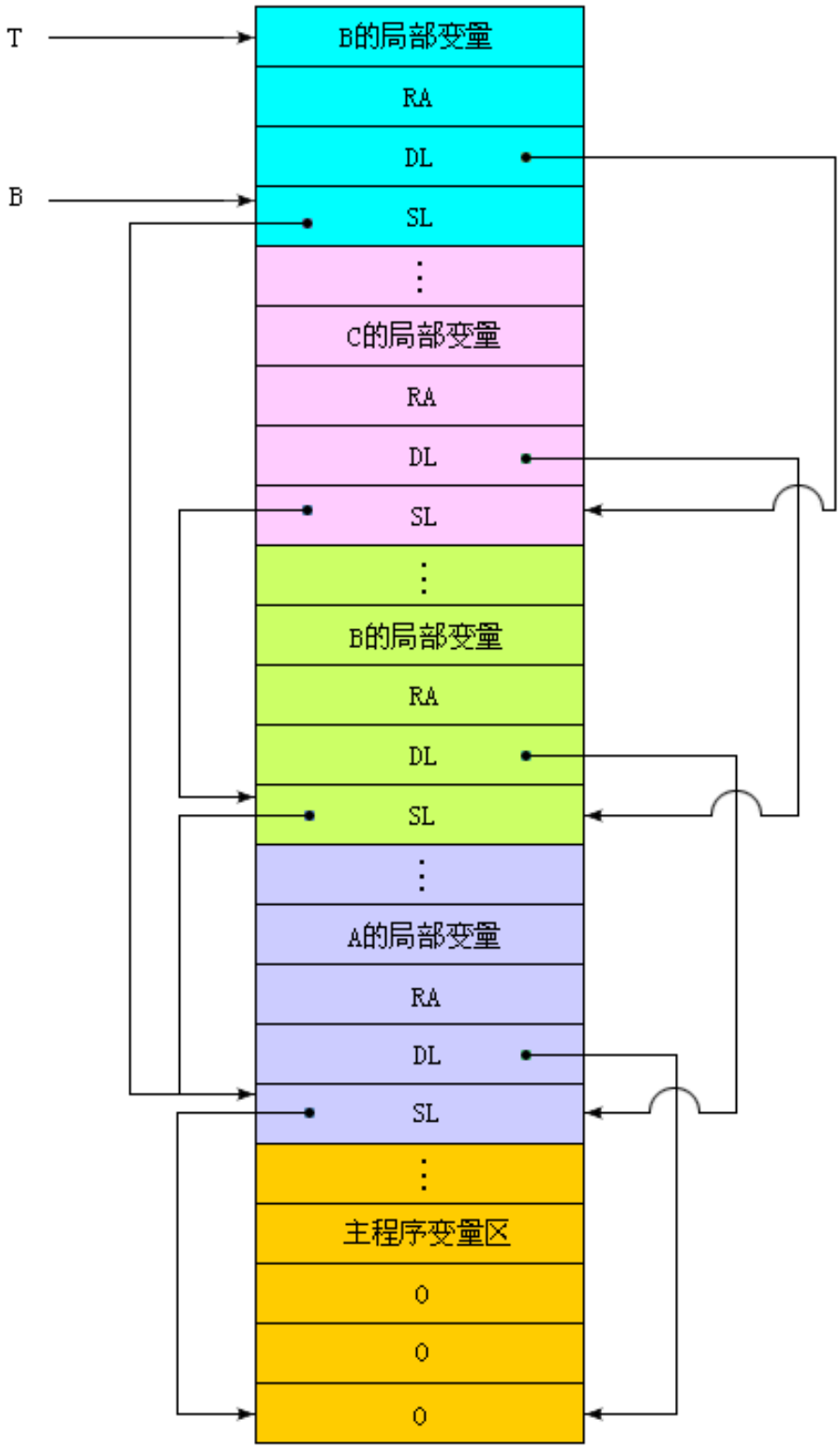
…

call A:

…

下面举例说明解释执行时数据区的变化过程，图 2.10 给出示意图。

图 2.10 运行时数据栈 S 的变化状态



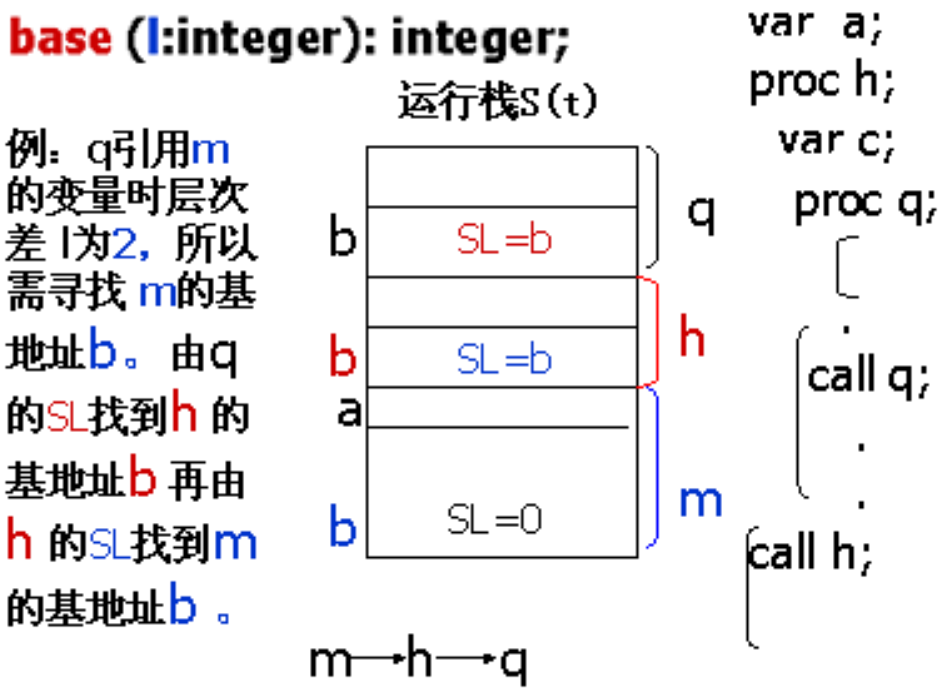
在图 2.10 中我们可以看到当例中程序执行进入到 C 过程后，在 C 过程中又调用 B 过程时，数据区栈中的状况，这时过程 B 的静态链是指向过程 A 的基地址，而不是指向过程 C 的基地址。因为过程 B 是由过程 A 定义的，它的名字在 A 层的名字表中，当在 C 过程中调用 B 过程时，层次差为 2，所以这时应沿 C 过程数据的静态链，跳过两个数据段的基地址值，才是当前被调用的 B 过程的静态链之值。这里也可看出不管 B 过程在何时被调用，它的数据段静态链总是指向定义它的 A 过程的最新数据段基地址。

在过程调用时如何寻找数据段静态链是由函数 base(l:integer)完成的（见 PL/0 文本的 interpret 过程）

函数 base(l:integer)程序

```
function base(l:integer): integer;
  var b1:integer;
    begin b1:=b; (*find base l level down*)
      while l>0 do
        begin
          b1:=s[b1]; l:=l-1;
        end;
      base:=b1
    end (*base*);
```

函数 base(l:integer)执行的示意图



具体的过程调用和结束，对上述寄存器及三个联系单元的填写和恢复由下列目标指令完成。

(1) INT 0 A

每个过程目标程序的入口都有这样一条指令，用以完成开辟数据段的工作。A 域的值指出数据段的大小，即为局部变量个数+3(联系单元个数为 3)。由编译程序的代码生成给出。开辟数据段的结果是改变栈顶寄存器 T 的值，即 T：=T+A；。

(2) OPR 0 0

是每个过程出口处的一条目标指令。用以完成该过程运行结束后释放数据段的工作，即退栈工作。恢复调用该过程前正在运行的过程(或主程序)的数据段基地址寄存器的值，和栈顶寄存器 T 的值，并将返回地址送到指令地址寄存器 P 中，以使调用前的程序从断点开始继续执行。

(3) CAL L A；

为调用过程的目标指令，其中

L： 为层次差，它是寻找静态链的依据。在解释程序中由 BASE(L)函数来计算，L 为参数，实参为所给层差。

A： 为被调用过程的目标程序入口。

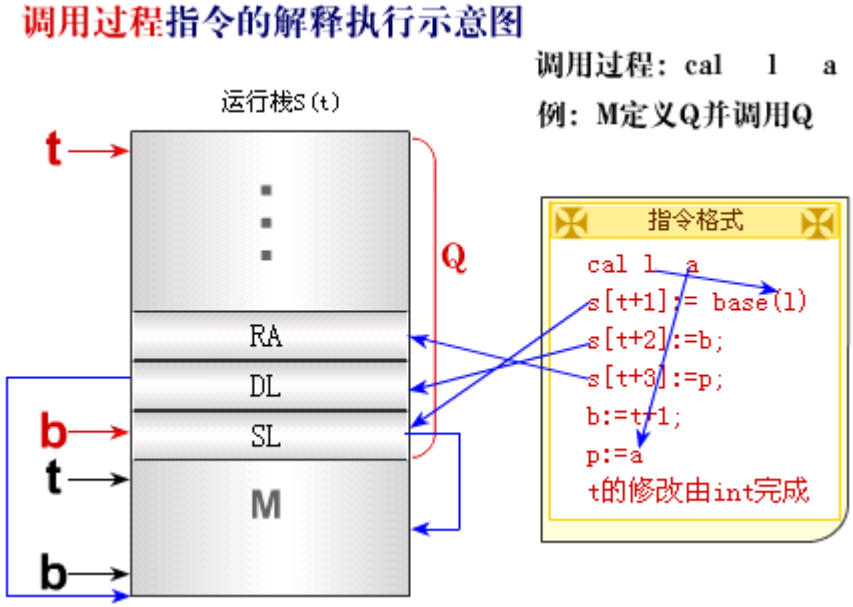
CAL 指令还完成填写静态链、动态链、返回地址，给出被调用过程的基地址值，送入基址寄存器 B 中，目标程序的入口地址 A 的值送指令地址寄存器 P 中，使指令从 A 开始执行。

几条特殊指令的解释执行：（见 PL/0 文本的 interpret 过程中相关操作的解释执行）

调用过程指令格式 ： cal l a

```
cal: begin (*generat new block mark*)
            s[t+1]:=base(l); 填写静态链
            s[t+2]:=b; 填写动态链
            s[t+3]:=p; 填写返回地址
            b:=t+1; 被调用过程的基地址
            p:=a 过程入口地址 a 送 p
        end;
```

调用过程指令的解释执行示意图



过程入口指令格式 ： int 0 a 在栈顶开辟 a 个单元（进栈）

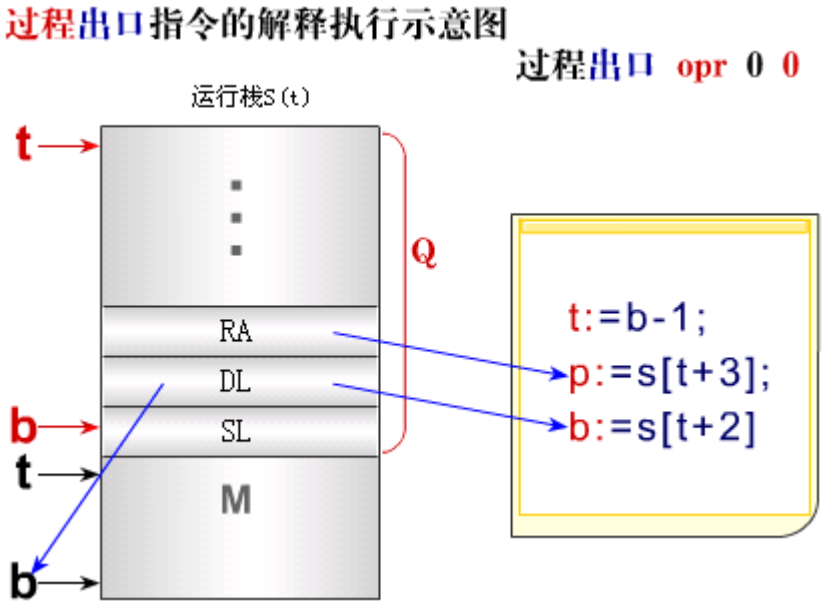
int: t:=t+a; （t 是当前栈顶值）

过程出口指令格式 ： opr 0 0 释放数据段（退栈）

```
opr: case a of (*operator*)
            0: begin (*return*)
            t:=b-1; 恢复调用前栈顶
            p:=s[t+3]; 送返回地址到 p
            b:=s[t+2] 恢复调用前基地址
```

end;

过程出口指令的解释执行示意图



最后为了使读者弄清 PL/0 编译程序各阶段的任务；源程序和目标程序的等价功能；解释执行目标程序时数据栈的变化情况，建议读者参看第 2.5 节中的例子，在阅读 PL/0 编译程序文本时，可按例子对照学习。

另外由于 PL/0 编译程序是用 PASCAL 语言编写的(若文件名为 PL0.PAS)，所以要对 PL/0 语言的源程序进行编译，如在 PC 机上，首先必须对 PL0.PAS 进行编译、汇编、连接得到 PL0.EXE 文件。运行 PL0.EXE 文件才是启动 PL/0 的编译程序。因此执行命令。

RUN PL0<回车>启动 PL/0 编译程序，输出一些询问信息，需用户回答。

输出信息	回答信息
INPUT FILE?	PL/0 源程序文件名<回车>
LIST OBJECT CODE?	Y 或 N<回车>

目标程序输出的次序是，最内层的过程体在最前边，主程序体在最后。源程序清单中的序号，是该语句在目标程序中对应的起始位置。但需指出例题中序号为 0，1 指令的内容为：

0 jmp 0 8　　8 为主程序入口

1 jmp 0 2　　2 为过程 P 的入口

【本章小结】

本章是一个编译程序的实例，通过认真阅读 PL/0 语言编译程序文本，加深理解一般编译程序构造的整体结构和实现的步骤，对词法、语法、语义分析、代码生成及符号表管理每个过程的功能和相互联系及实现技术。联系实际做好作业和实验，巩固知识。

实验要求:

◇ 扩充条件语句

〈条件语句〉::=IF〈条件〉THEN〈语句〉[ELSE〈语句〉]

◇ 扩充重复语句

〈重复语句〉::=REPEAT〈语句〉{；〈语句〉}UNTIL〈条件〉

◇ 对 PL/0 语言扩充整形数组（有条件的同学尽量做此题）。

读者自己设计语法并给出用语法图和 EBNF 描述

也可用其它语言改写 PL/0 编译程序

如用 C 语言或 Java 语言改写 PL/0 编译程序；

实验报告内容:

⑴ 对扩充部分用语法图和 EBNF 描述；

⑵ 对原 PL/0 语言编译程序文本中程序变动部分的说明；

⑶ 所用测试用例包括正确的测试用例和错误的测试用例；

⑷ 实验体会和建议。

建议：为了引起读者对阅读 PL/0 编译程序文本的兴趣，建议用本节教材文本中的 PL/0 源程序和目标程序的对应关系为例，阅读 PL/0 编译程序文本，并总结词法、语法、语义分析和目标代码生成之间的关系。其方法是：

⊙ 先用 PL/0 语言的语法图和 EBNF 检查例子的语言是否合乎语法规则。

⊙ 阅读 PL/0 源程序文本。方法是从主程序开始，例中遇到什么语法成分就转到相应的语法处理过程。

⊙ 遇到语义处理造名字表或生成目标代码时，用纸记录下它们操作的结果，以及名字表 table 和目标代码 code 的信息变化，并分析这些变化的目的。例如，对过程入口的拉链返填方法。

⊙ 遇到看不懂的地方再回头看教材文本和注释中的说明和解释。

⊙ 检查你所得到的代码是否与例中的相同，若不同则找出原因。