编译原理大作业实习

国防科学技术大学计算机学院

目　录

第一节　概　述

一、目的与要求

与第十六章编译原理（技术）基础实习不同，本章旨在加强学生对编译过程整体认识，而不是个别阶段实习。

二、组织与实施

我们以一个经简化、但关键内容又不缺少的高级语言 PL 语言为背景，实习编译程序的基本构造方法。PL 语言是 PASCAL 语言的一个子集，它的设计主要参考了 N. Wirth 等人设计实现的 PL/0 和 PASCAL—S 语言。该语言不太大，但能充分展示高级语言的最基本成分。

本实习的实验可采用以下两种方法：

（1）对 PL 编译程序进行扩充，通过扩充加深对编译程序的理解。扩充的语言成分根据可用机时和学生程度确定。这种方式可由每个学生单独进行。

（2）以本章所介绍的编译程序构造方法为依据，由学生独立完成一个能实现编译功能的简单编译程序设计。这种方式适宜采用小组进行实验，每个学生完成一部分工作，最后连接起来。

实习时间为每人 20~40 学时。学习工具宜采用 PASCAL 语言或 C 语言。

三、PL 语言及其编译程序简介

PL 编译程序用 PASCAL 语言编写，其源语言为 PL 语言，目标语言为 PC 宏汇编语言。

1．PL 语言简介

PL 语言是 PASCAL 语言的一个子集，它包含程序设计所需的最基本语言成分：

（1）常量、类型、变量和过程等说明；

（2）过程说明允许嵌套，过程调用允许递归，过程参数可以是值参数和变量参数；

（3）3 种标准类型：整型、字符型、布尔型和一种自定义数组类型；

（4）赋值语句、条件语句、while 语句、过程调用语句、读语句和写语句。

下面用 BNF 给出 PL 语言的文法。

<程序>::=“problem”《标识符》 “:”<程序体>“.”

<程序体>::= [“const”<常量说明>{<常量说明>}]

[“type”<类型说明>{<类型说明>}]
[“var”<变量说明>{<变量说明>}]
[<过程说明>{<过程说明>}]
“begin”<语句>{“;”<语句>}“end”
<常量说明>::=<标识符>“=”<常量>“;”
<类型说明>::=<标识符>“=”<类型>“;”
<类型>::=<类型标识符>|<数组类型>
<数组类型>::=“array” “[”<常量>“. .”<常量>
{“，”<常量>“. .”<常量>}“]”“of”<类型>
<变量说明>::=<标识符>{“，”<标识符>}“:”<类型>
<过程说明>::=“procedure”<标识符>[<参数表>]“;”<程序体>“;”
<参数表>::=“(”[“var”]<标识符>{“,”<标识符>} “:”<类型标识符>
{“;”[“var”]< 标识符>{“,”<标识符>} “:”<类型标识符>} “)“
<语句>::=<赋值语句>|<条件语句>|<循环语句>
|<过程语句>|<复合语句>|<空语句>
<赋值语句>::=<变量>“=”<表达式>
<条件语句>::=“if”<表达式>“then”<语句>“else”<语句>
<循环语句>::=“while”<表达式>“do”<语句>
<过程语句>::=<过程标识符>[“(”<表达式>{“，”<表达式>“}”]
<复合语句>::=“begin”<语句>{“;”<语句>}“end”
<空语句>::=
<表达式>::=<简单表达式>{<关系运算符><简单表达式>}
<关系运算符>::=“=”|“<”|“>”|“<=”|“>=”|“<>”
<简单表达式>::=[“+”|“—”]<项>{<加法运算符><项>}
<加法运算符>::=“+”|“—”|“or”
<项>::=<因子>{<乘法运算符><因子>}
<乘法运算符>::=“*”|“/”|“mod”|“and”
<因子>::=<无符号常量>|<变量>|“(”<表达式>“)”|“not”<因子>
<变量>::=<变量标识符>[“[”<表达式>{“.”<表达式>}“]”]
<常量>::=[“+”|“—”](<常量标识符>|<无符号整数>)|字符
<无符号常量>::=<常量标识符>|<无符号整数>|字符
<标识符>::=字母{字母、数字}
<无符号整数>::=数字{数字}

2．PL 编译结构

PL 编译程序是一个两遍扫描的编译程序，其结构如图 17.1.1 所示。

第一遍扫描包括词法分析、语法分析、语义分析。最后生成一种中间语言代码形式。采用比较流行且容易实现的递归下降分析法，整个程序以语法分析为核心;词法分析作为一个比较独立的子程序，等到语法分析再需要单词符号时调用;根据语法制导翻译的思想在语法分析过程中每当分析出一个语法单位就执行相应的语义动作，并产生相应的中间代码。

第二遍扫描为目标代码生成，它把中间代码逐条翻译为 PC 宏汇编语言代码。

本章给出了 PL 编译程序的原程序清单，它是在 PC 系列机上用 Turbo-PASCAL5.0 实现的，除极少数语句不同外，很容易移植到其他机器环境中。

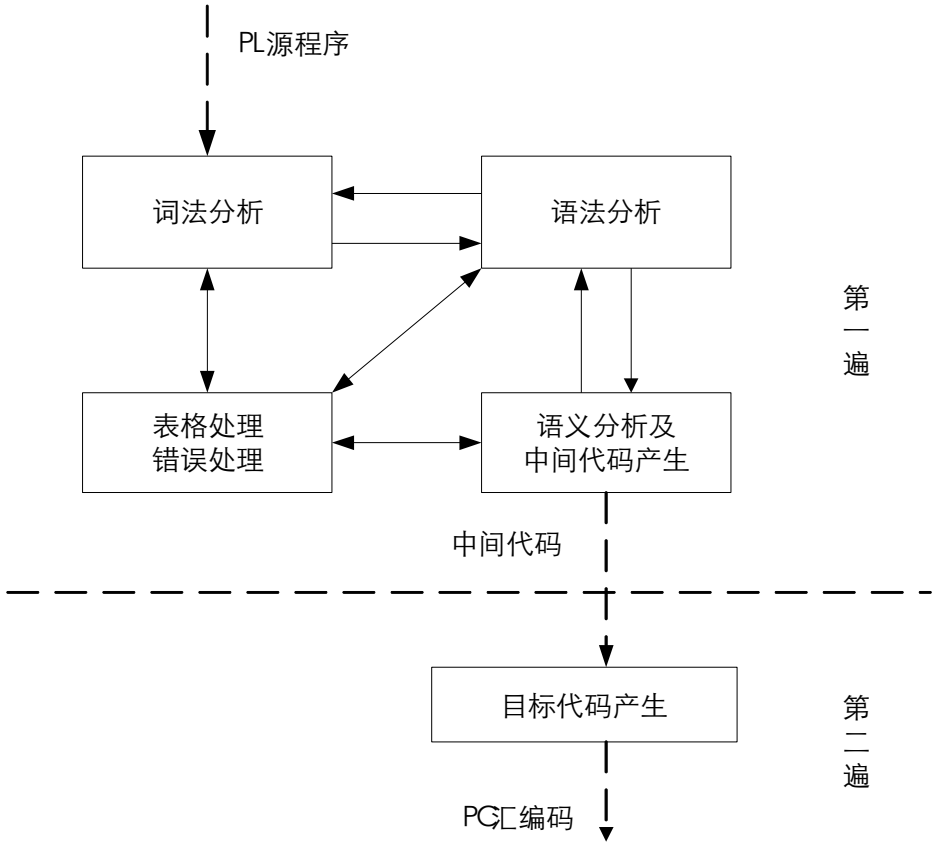


图 17.2.1 PL 编译结构

四、实习题

在阅读和理解 PL 编译程序之后，根据教学要求和条件对 PL 语言及编译程序进行扩充。

1.扩充语句。例如，增加 for 语句、repeat 语句、case 语句等。

2.扩充数据类型，例如实型、记录类型、子界类型、指针类型、文件类型。

3.增加文件的说明和引用。

4.增加其他语言成分。

第二节　PL 词法分析

一、 词法分析的基本实现方法

词法分析读入源程序，并从源程序的字符串中识别出一个个的单词符号。我们将词法分析器设计成一个独立的子程序，供语法分析需要时调用。每调用一次，它将提交给语法分析一个二元组

（单词种别，单词的自身值）

因此，在设计词法分析器时，首先应对单词种别进行分类和编码。PL 语言的单词种别在编译程序中用枚举值表示，以提高可读性。单词符号与其种别表示的对应关系如表 17.2.1 所示。

单词符号的识别方法是，自左至右逐个字符地扫描源程序。找出其中具有独立意义的字符串（即单词符号），并把它提交给语法分析程序。

表 17.2.1 PL 语言单词符号及其种别值

单词符号	种别枚举值	单词符号	种别枚举值
标识符	IDENT	]	RbRACK
整常量	INTCOM	(	LPAREN
字符常量	CHARCON	)	RPAREN
]+	PLUS	,	COMMA
-	MINUS	;	SEMICOLON
*	TIMES	.	PERIOD
/	DIVSYM	:=	BECOME
=	EQL	:	COLON
<>	NEO	begin	BEGINSYM
<	LSS	end	ENDSYM
<=	LEQ	if	IFSYM
>	GTR	then	THENSYM
>=	GEQ	else	ELSESYM
of	OFSYM	while	WHILESYM
array	ARRAYSYM	do	DOSYM
program	PROGRAMSYM	call	CALLSYM
mod	MODSYM	const	CONSTSYM
and	ANDSYM	type	TYPESYM
or	ORSYM	var	VARSYM
not	NOTSYM	Procedure	PROCSYM
[	LbRACK		

在设计词法分析程序时，我们通常先画出识别单词符号的状态转换图（确定的有限自

动机），然后可以很方便地把状态转换图用程序实现。识别 PL 语言单词符号的状态转换图如图 17.2.1 所示。

由该状态转换图编写出 P L 编译程序中词法分析子程序 GETSYM.其工作过程如下:

(1)跳过源程序中的空格字符;

(2)从源程序字符序列中识别出单词符号,并把该单词符号相应的种别枚举值送入全局变量 SYM 中;

(3)如果取来的单词为标识符,则把它存入全局变量 ID 中.为了区分保留字和标识符,设置了一张保留字表 WORD,用二分法查找保留字表,识别诸如 if、end 等保留字;

(4)如果取来的单词为无符号整常数,则将构成该整数的数字字符串转换为内部二进制整数值存入全局变量 NUM 中;

(5)如果取来的单词为字符常量,则将该字符的 ASCII 序号存入全局变量 NUM 中。

为了扫描输入的字符序列，GETSYM 使用一个过程 GETCH，其任务是取一个字符。

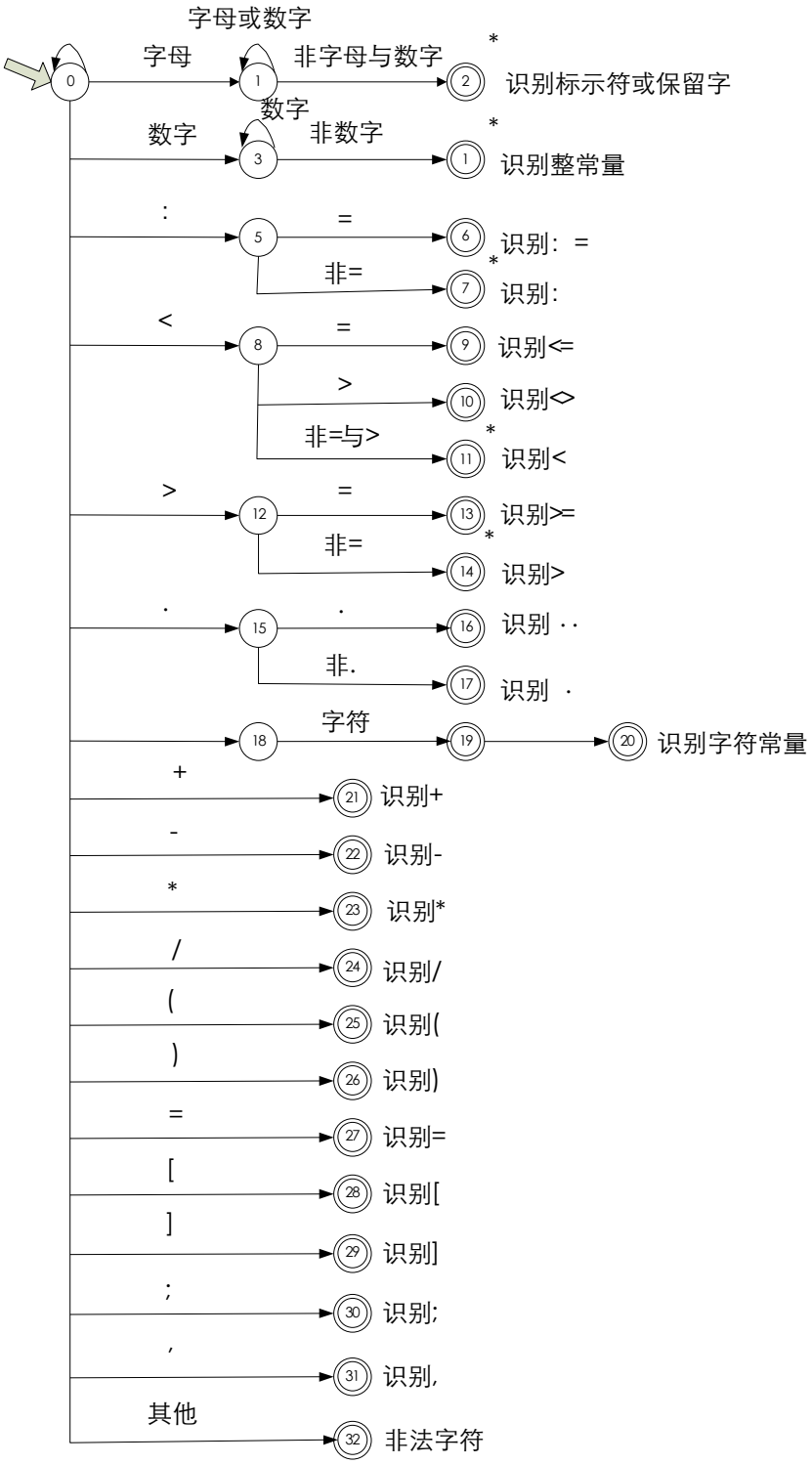


图 17.2.1 识别 PL 单词的状态图

GETCH 将输入源文件（.PLS）的每一行源程序先读入缓冲区 LINE 中，然后再从 LINE 中取出字符。程序中用变量 LL 记录当前源程序行的长度，用变量 CC 对当前所取字符在该行的位置进行计数。GETCH 除取来下一个字符外，还完成：

（1） 识别并越过行结束标记。

（2） 把源文件读入的源程序同时输出到列表文件（.LST）上，以形成被编译程序的列表。

（3） 在输出每一行源程序的开始处印出编译生成的中间代码指令相应行号。

二、实习提要

（一）目的与要求

通过扩充（或设计）.调试一个词法分析程序，，加深对词法分析原理的理解，掌握词法分析法。

要求认真阅读 PL 编译程序中词法分析程序 GETSYM，弄清词法分析的基本方法和实现技术;根据需要，确定单词种别及识别它们的策略;确保词法分析工作的正确性，为语法分析打下良好的基础。

（二）实习题

词法分析工作相对来说比较简单，为了扩充 PL 语言编译程序，要对其词法分析子程序 GETSYM 稍加改动或扩充。

1 扩充词法子程序 GETSYM，使 PL 语言增加更多的语句，增加函数定义和引用。

这时，应把诸如 for、repeat、until、case、function 等保留字插入保留字中适当位置。

2. 修改 GETSYM 使之适应扩充数据类型的需要。这时

（1）如果要增加实数类型且允许实型常数采用科学表示法（如 1.32E+3），则在识别它们时，应正确地把它们转化为实数数值表示。

（2）如果允许 PL 语言程序中使用字符串常量（如‘ABCD’），则应为字符串常量安排合适的存贮方式。

3. 可以在 GETSYM 中增加处理注解的功能，注解形式可为{……}或|*……*|。

4．PL 语言表示符号与保留字只允许小写字母，如果想不区分大小写，请对 GETSYM 进行修改。

第三节　PL 语法分析

一、语法分析程序的构造

语法分析的任务是分析语言的句子结构。对于词法分析其所产生的单词符号串，语法分析器将根据文法的产生式规则，识别该符号串是否为文法的句子（或程序）、

语法分析的方法有许多，我们采用比较实用、容易掌握的递归下降分析法。递归下降分析方法的基本思想是直接由产生式构造语法分析程序。对文法中的每个非终结符，构造一个与之对应的分析子程序，把这些子程序组合在一起，就构成了给定语言的语法分析程序。由于文法定义是递归的，因此，相应的过程也是递归的。

为了构造不带回溯的递归下降分析程序，要求语言的文法满足下面的 LL（1）条件。

对文法中的每个非终结符 A，设它的所有的产生式为

$$A \longrightarrow a_1 | a_2 | ... | a_n$$

则要求：

（1）A 的各个产生式的头符号集合必须两两不相交，即

$$FIRST(a_i) \cap FIRST(a_j) = \varnothing \quad (i=j,\text{对 } i,j=1,2,3...n)$$

（2）对每个能推导出空串的非终结符 A，其头符号集与其后继符号集不相交，即便

FIRST（A）∩ FOLLOW（A）= Ø

对于不满足 LL（1）条件的文法，必须先对文法进行改造，使之满足 LL（1）条件，才能构造其递归下降分析程序。改写文法的方法通常有消除左递归和提取公共因子。

对满足 LL（1）条件的文法，我们可根据文法中每个非终结符相应的产生式形式构造其分析子程序。假设对每个语法项 T，其相应的分析程序中的语句为 S（T），则：

（1）形如　T1T2…T n 的语法项，其相应的语句为复合语句

begin S(T1); S(T2);…S(Tn) end

（2）形如　T1 | T2}　…T n 的选择形式的语法项，其相应的语句为 if 语句或 case 语句，如

```
    case sym of
FIRST(T1): S(T1);
FIRST(T2): S(T2);
     - - -
FIRST(Tn): S(Tn)
end
```

（3）行如{T}的重复形式的语法项，其相应的语句为

while sym in FIRST（t）do S（T）

（4）对每个非终结符 A，其相应的语句为调用 A 所对应的分析子程序

（5）对单个终结符 a，其对应的语句为

if sym =a then getsym

else error

对上述模式，具体编程时，可根据需要灵活掌握。

在构造 PL 语言的递归下降分析程序之前，对上节中所给的 PL 语言的文法，我们求出非终结符的头符号集和后继符号集附表 17.3.1 所示。

对照表 17.3.1 验证 LL（1）条件，我们发现，对于<语句>的产生式，由于

FIRST（<赋值语句>）∩ FIRST（<过程语句>）={ident} ≠ Ø

不满足 LL（1）条件，因为赋值语句与过程语句都已标识符开头。

解决这个问题有 3 种方法：

表 17.3.1 PL 的头符号集以及后继符号集

非终结符	FIRST 符号集	FOLLOW 符号集
<程序体>	const type var procedure begin	. ;
<常量说明>	ident	ident type var procedure begin
<类型说明>	ident	ident var procedure begin
非终结符	FIRST 符号集	FOLLOW 符号集
<变量说明>	ident	ident procedure begin
<过程说明>	procedure	procedure begin
<类型>	ident array	;
<参数表>	(	;
<语句>	ident if while begin	; end else
<赋值语句>	ident	; end else
<条件语句>	if	; end else
<循环语句>	while	; end else
<复合语句>	begin	; end else
<表达式>	ident int (+ - not	; ,)] then do end else
<简单表达式>	ident intoon (+ - not	; ,)] then do + - or rop else
<项>	ident intoon (not	; ,)] then do + - or rop end else
<因子>	ident intoon (not	; ,)] then do +- or and rop * / mod end else
<常量>	+ - ident intoon	; • •,

（1） 最简单的方法是引入保留字 call,要求过程语句以 call 开头，即把<过程语句>的产生式改为：

<过程语句>::=“call”<过程标识符>[“（”<表达式> {“,” <表达式>}“）”]

（2） 通过提左公共因子改造文法，例如，把有关<语句>的产生改为：

<语句>::=<条件语句>｜<循环语句>｜<复合语句>｜<空语句>｜<赋值或过程语句>

<赋值或过程语句其余部分>::=[“[” <表达式>{“,” <表达式>“[C] “:= ” <表达式>｜[“（”<表达式> {“,” <表达式>}“）”]

（3） 通过查找符号表中标识符的属性，确定正在分析的语句是赋值语句还是过程语句，从而进入不同的分析程序。

PL 的编译程序采用了第一种方法。读者实习是可改成其他方法。

对于修改后的 PL 语言文法，我们完全可以验证它们满足 LL（1）条件。因此，我们可以根据文法构造其语法分析程序。为了便于阅读程序，我们给出了 PL 分析程序的主要子程序的嵌套关系如图 17.3.1 所示。

```
PROGRAM                          {主程序}
    BLOCK                        {处理程序题}
        CONSTANT                 {处理常量}
        TYP                      {处理类型 }
        CONSTDECLARATION         {处理常量说明}
        TYPEDECLATARION          {处理类型说明}
        VARDECLARATION           {处理变量说明}
        PARAMENTERLIST           {处理参数表}
        PROCDECLARATIN           {处理过程说明}
        STATEMENT                {处理语句}
            EXPRESSION           {处理表达式}
                SIMPLEEXPRESSION     {处理简单表达式}
                        TERM         {处理项}
                            FACTOR   {处理因子}

                ASSIGNMENT           {处理赋值语句}
                IFSTATEMENT          {处理条件语句}
                WHILESTATEMENT       {处理循环语句}
                COMPOUND             {处理复合语句}
                CALL                 {处理过程调用语句}
```

图 17.3.1 PL 分析子程序嵌套关系

二、错误处理

一个好的编译程序不仅能识别和翻译语法正确的程序，还应该能检查出错误并对错误进行有效处理。通常，错误处理工作应做到以下几点。

（1）能检查出源程序中的各类错误并准确指出错误位置及性质。

（2）能通过一次编译程序将源程序中的错误尽可能多地找出来。

（3）尽可能把错误限制在一个局部的范围内，减少错误影响程序其他部分的分析和检查。

（4）尽可能校正错误。

PL 编译程序由过程 ERROR 完成报错工作·打印错误信息及出错位置。其错误信息表如表 17.3.2 所示，表中包括语法错误和语义错误。

表 17.3.2 错误信息表

0	非法字符
1	1 名字表溢出，源程序中定义的名字最多 100 个
2	数组信息表溢出
3	程序体信息表溢出
4	过程定义的嵌套层次不能超过 7
5	中间代码太长
10	标识符无定义
11	标识符多重定义
12	应为常量或常量标识符
13	非法符号、本符号在此出现是非法的
14	数组下标说明错误
15	应为 program
16	应为“[”
17	应为“of”
18	类型定义出错
19	应是类型标识符
21	形参说明应以 var 或标识符开头
22	应为标识符
23	应是“;”
24	应是“：”
26	应是“=”
28	应是“]”
29	实参与形参个数不等
30	应为变量
31	实参与形参类型不一致
32	应是“（”
33	应为“：=”
34	if 或 while 后面的表达式必须为布尔型
35	应为“then”
36	应为“end”
37	应为“do”
38	应为“‥”
40	型不一致
41	过程名或类型名不能出现在表达式中

43　　操作数类型错
44　　数组元素引用时，下标类型错
45　　下标个数不正确
46　　字符常数错误
47　　整型常数太大
48　　应为”’”
51　　应用过程名

当发现源程序中有语法错误时，说明语法分析与源程序中当前符号以不匹配。这时，如果想继续进行语法分析，就需设法改变这种不匹配状态。一种有效的做法是：当发现错误时，就适时跳过后面的一段输入符号，直到下一个可以正确地后随当前正在分析的语法结构的符号为止。这就是说我们至少可使分析跳到当前正分析的语法结构的后继符号集合中。然而，如果在任何情况下都要跳到输入符号中下一个这种后继符号出现的地方，可能会漏检过多的符号。因此，除了后继符号集外，用来终止跳读的符号集中还应加上那些明显能使程序继续开始分析的保留字或其他终止符号。

在 PL 编译程序中，我们为每个分析过程提供了一个参数 FSYS,它指明正在分析的语法结构可能的后继符号和上述明显的不能忽略而跳过的符号的集合。我们先传递给 FSYS 一定的初值，然后随着语法分析过程的一步步深入逐步补充别的符号。

上面所述的测试与跳读工作由过程 TEST 实现

```
PROCEDURE TEST(S1,S2:SYSMSET;N:INTEGER)
BEGIN
  IF NOT (SYM IN S1) THEN
    BEGIN
      ERROR(N); S1:=S1+S2;
      WHILE NOT (SYM IN S1) DO GETSYM
    END
END; {TEST}
```

该过程有 3 个参数，SI 为非法符号集合，如果当前符号不在此集合内，则报告一个错误;S2 为另加的停止符号的集合，这些符号的出现无疑是错误的，但它们绝不应忽略而跳过;整数 n 表示错误信息的的序号。

TEST 过程主要出现在下面两种场合。一是在对语法单位分析的出口处，使得当本语法单位中出现错误是，编译程序能从适当的后续符号开始继续后续语法单位的分析。二是当进入一个语法单位的分析程序时，利用 TEST 来测试当前读入的符号是否属于该语法单位的开始符号集合。

对源程序中的语法错误进行校正往往是非常困难的。Pl 编译程序实现了一些简单的错误校正工作。如，把赋值号 “:=”写成 “=”，在 if 语句中，把 if…then…else 写成 if…do…,等等，这些错误显然是程序员的笔误造成的，编译程序予以校正。

下面我们举一个简单例子说明错误的源程序编译后的报错信息列表。

```
0   program sort;
0    var i,j,temp:integer;
1        a:array[1..10]of integer;
1    begin
```

```
2      i:=1;
5      while i<10 do
9        begin
9          j:=i+1
12          while j<=10 do
****                  23
18            begin
18                      if a[i]>a[j]then
35                    begin
35                  temp :=a[i];
46                        a[ i ] : = a[ j ]
* * * *                 28
* * * *                 13
62                  a[ j ] : = temp
69                    end
71                        j : = j + 1
74            end ;
77         i = i + 1
* * * *                  33
80     end
```

三、表格处理

编译过程中需不断地汇集和反复查证出在源程序中的各种名字及其属性，这些信息通常记录在若干张表格中。借助这些表格编译程序可以有效地进行名字的作用域分析、类型一致性检查、辅助代码生成、进行地址分配等。

1. 表格定义

PL 编译程序使用的主要表格有:名字表（nametab）、程序体表（btab）、层次显示表（display）

数组信息表(atab)、中间代码表(code)。

（1）名字表 nametab

名字表 nametab 的作用是登记程序中出现的各种名字及其属性，表格形式如下:

其中:

name,名字标识符，取前 10 个字符。

kind,名字种类，可以使常量（constant）、变量（variable）、类型（tape）、过程（procedure）。

lev，名字所在程序体静态层次。规定主程序的层次为 1，主程序中定义的过程层次为 2，以此类推。

name	kind	lev	typ	normal	ref	adr/val/size	link

0								
1								
tx→								

typ,名字的类型，类型有整型（ints）、字符型（chars）、布尔型（bool）、数组（arrays）、对无类型的名字填入 notyp.

ref,当名字为数组类型名或数组变量名时，ref 指向该数组在数组信息表（atab）中登陆的位置;当名字为过程名时，ref 指向该过程在程序体表（btab）中的相应位置;其他情况 ref 为 0.

normal,一个布尔量，用于标明名字是否为变量型参名。当名字为变量型参名时填入 false,其他情况填入 ture 或不填。

adr，当名字为变量名时（包括形参），填入该变量（或形参）在相应活动记录中分配存贮单位的相对地址;对于过程名、填入它们相应代码的入口地址。

val,当名字为常量名时，填入它们的相应值。

size,当名字为类型名时，填入该类型数据所存贮单位的数目。

link,指向同一程序体中定义的上一个名字在 nametab 中的位置。每个程序体在 nametab 中登记的第一个名字的 link 域为 0.

（2）程序体表 btab 和层次显示表 display

为了便于对源程序中定义的名字的作用域进行分析，PL 编译程序建立了一张程序体表 btab，用于记录各程序体的总信息。另外，由于 PL 的程序结构允许嵌套，因此，编译程序还设立了一个层次显示表 display，描述正在处理的各嵌套层。程序体表对名字表进行管理，层次显示表对程序体表进行管理。

程序体表 btab 的形式如下:

	lastpar	**last**	**psize**	**vsize**
0				
1				
⋮				
bx→				

其中:

lastpar,指向本程序体中最后一个形参在 nametab 中的位置。

last，指向本程序体中最后一个名字在 nametab 中的位置。

psize，本程序体所有形参所需体积、包括连接数据所占空间。

vsize，本程序体所有局部数据所需空间大小。

display 表的形式如下

0	
1	
⋮	
level→	

display 表的每一项指向相应层的程序体在 btab 的位置。

为了便于理解，我们用一个例子说明 namenb、btab、dipplay 的关系。假设某个程序的结构如下。

```
program P:
  var a,b:integer;
  procedure P1(i1, j1 : integer);
     var c, d : integer
      ┆
  end;
    procedure P2(i2, j2 : integer);
   var a, c : integer
   procedure P21;
     var b1,b2:Boolean;
      ┆
   end;
    ┆
      end;
┆
    end.
```

当编译程序处理完过程 P21 的说明部分时，表格内容如图 17.3.2 所示。为了简单，图中 btab 表我们只列了 lastpar 和 last 栏的内容，nametab 表只列了 name 和 link 栏的内容。

从图 17.3.2 我们可以看到，PL 语言的预定义标准标识符（如 char,false 等）可以认为是在 0 层程序中定义的，它们由过程 enterpreid 在编译开始时填入表中。

（3）数组信息表 atab

它用于记录每个数组的详细信息，表格形式如下。

	inxtyp	eltyp	elref	low	high	elsize	size
1							
2							
┆							
ax→							

其中:

inxtyp,数组的下表类型。

eltyp,数组元素类型。

elref,当元素类型为数组时，它指向该元素数组信息在 atab 表中的位置，其他情况为 0。

low、high,分别表示数组的上下限。

elsize、size，分别表示数组元素的体积和数组本身的体积。

(4)中间代码表 code

code 用于存放编译程序所产生的每条中间代码。

2. 表格处理

对符号表格的操作主要有查表、填表等。一般而言，当编译程序处理到对名字的说明语句要进行填报工作，当处理到对名字的引用时要进行查表工作。

（1）填表操作。对名字填表时，为了判断名字是否重复定义，在填表之前首先要查表。根据 PASCAL 语言的作用域规则(因为 PL 是 PASCAL 的子集)，一个名字可以与外界程序体中定义的名字具有统一标识符，因此只检查本层程序体对名字是否已有定义。根据当前

程序体层次 level 沿着 display 表和 btab 表找到本层的最后一个名字的位置,再沿着 link 域逐个比较。如果表中已有相同标识符则报错，否则把名字填入表中。

PL 编译程序中，名字的填表操作主要有 ENTER 过程完成，程序体表和数组信息表每一项的建立分别由 ENTERBLOCK 和 ENTERARRAY 完成，中间代码的填表由 GEN 过程完成。

（2）查表操作。对名字进行查表时，由于名字是先定义后使用，因此，若查不到，则表示该标识符无定义。根据作用域规则，如果在本程序体中查不到该名字，则要在其直接外层程序体中继续查找，等等如此，一直查找到 0 层中定义的预定义名字。

PL 编译程序中，对名字的查找操作由函数 POSITION 完成，若查到则 POSITION 回送名字在 nametab 中的位置，若查不到则回送 0.。

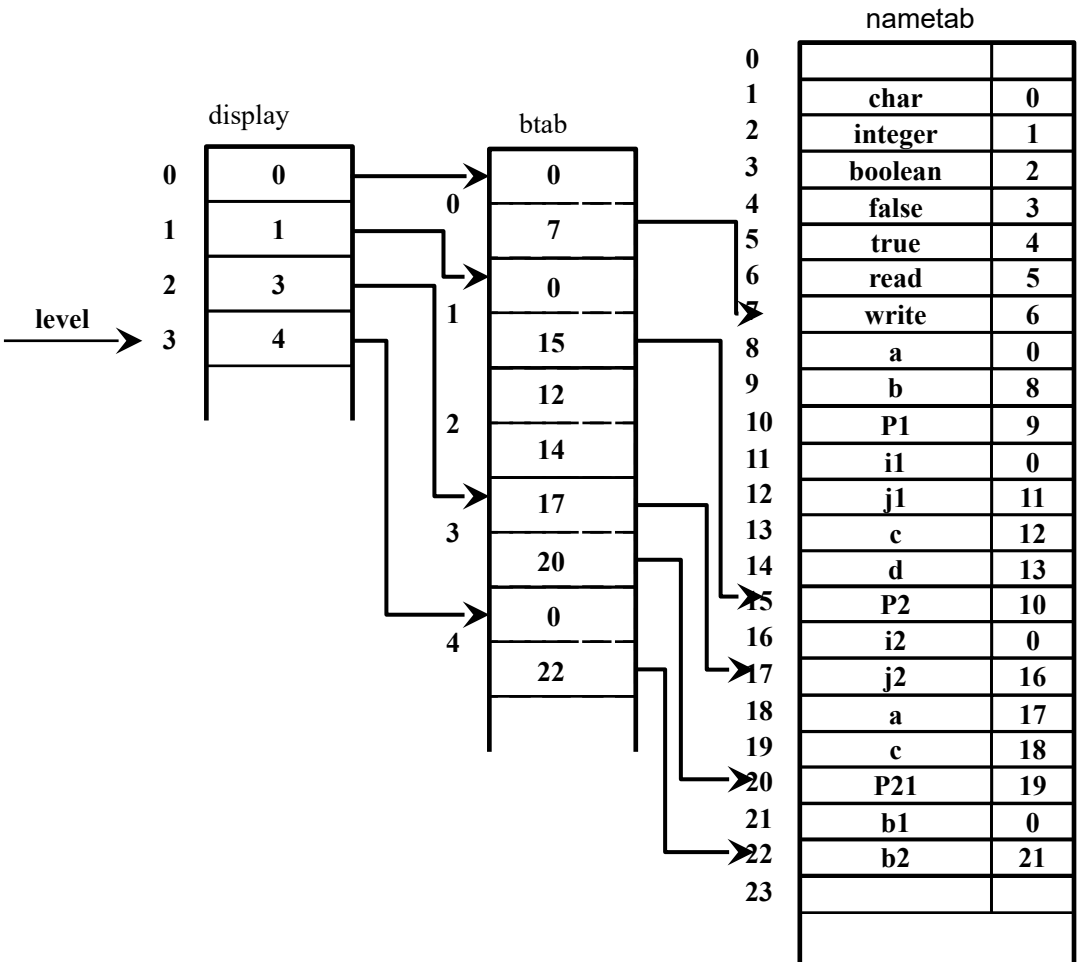


图 17.3.2 符号表内容示例

四、实习提要

（一）目的与要求

通过扩充（或设计）、调试一个实际的语法分析程序、理解语法分析原理与方法，掌握递归下降分析程序的构造。理解和掌握错误处理方法即符号表的组织方式。

为了实现在扩充的语言成分的语法分析、首先应对 PL 语言文法进行扩充，并使文法满足 LL（1）条件，然后按照文法构造语法分析程序。要增加对扩充语言成分的错误处理的功能。根据扩充的语言成分的需要、合理调整和扩充符号表格及其内容。

（二）实习题

1．扩充语句部分,如增加 for 语句 repeat 语句 case 语句等。

这时，要对文法中<语句>的定义进行扩充，然后写出相应的分析子程序。

2．增加函数的说明和使用。

函数的处理与过程的处理类似，因此很多地方可供用分析程序。主要差别在于，寒暑说明是必须指明函数值的类型，寒暑的引用通常出现在表达式中。

3．扩充数据类型，如扩充实型、记录类型等。

若要扩充记录类型，则应注意，记录中得域名可与其它名字相同，但同一记录中各域的名字不能重名，因此，记录中域名的填查要做特殊处理。

4．其它语言成分扩充。

第四节　PL 语义分析及中间代码产生

编译程序识别出每个语法单位后，就可对他们进行语义分析和翻译。通常，编译程序不把高级语言程序直接翻译成待定机器上的目标代码，而是先翻译为中间语言形式。这样更易于编译程序的移植和优化。除了分析程序的“含义”和进行翻译外，语义分析工作还包括名字的作用域确定及类型一致性检查等。名字的作用域确定实际上在上节的表格处理一段中涉及到。

PL 编译成序采用的中间语言为 PL 抽象机代码。

一、PL 抽象计算机

PL 抽象计算机是一台虚构的计算机，它的结构和指令的设计充分考虑了 PL 语言的特点。它不依赖于任何特定的计算机，但很容易翻译为各种具体的计算机代码。

PL 抽象机包括一个存储器和 4 个寄存。

存储器可分为两个部分。第一部分为代码区 code,用于存放 PL 代码指令。第二部分为数据区 S，用于存放程序运行时所需的各种数据，数据区 S 组织为一种栈式结构。

PL 抽象机的 4 个寄存器为：

i,指令寄存器

PC,指令计数器

bp,基址寄存器

top,栈顶寄存器

存储器与寄存器的关系如图 17.4.1 所示。

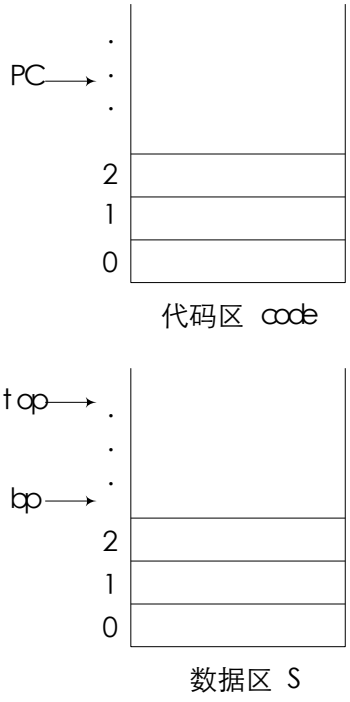


图 17.4.1 存储器和寄存器的关系

PL 抽象机是一台栈式计算机，它没有传统的累加器，所有对数据的运算均在栈顶进行。例如，加法指令是把栈顶两个单元的内容相加，并把结果留在次栈顶，条件转移指令根据栈顶单元的 内容决定是否转移，等等。

PL 抽象机每条指令的格式为：

opcod	l	a

其中，opcod 为操作码，l 为第一操作数，经常用于表示操作对象定义所在的程序体层次;a 为第二操作数，经常表示操作对象在相应程序体的活动记录中的相对地址。

PL 抽象机指令集及各指令的功能见表 17.4.1.PL 抽象机及其指令通过解释程序 INTERPRET 实现。读者可通过阅读 INTERPRET 详细了解各条指令的功能。

表 17.4.1 指令操作码

f	l	a	功能	f	l	a	功能
LIT	0	A	装入常量 A	FNDP	0	0	程序结束
LITI	0	A	装入常量 A(用于计算数组元素地址)	ANDS	0	0	与
LOD	L	A	装入变量值(L 为层次，A 为相对地址)	ORS	0	0	或
ILOD	L	A	间接装入	NOTS	0	0	非
LODA	L	A	装入变量地址(L 为层次，A 为相对地址	IMOD	0	0	求模
LODT	0	0	装入栈顶值为地址的单元内容	MUS	0	0	求负
LODB	0	A	块装入(A 为块长)	ADD	0	0	加
STO	0	0	将栈顶值存入次栈顶值所指单元	ADD1	0	0	加(用于计数组元素地址)
CPYB	0	A	块传送(A 为块长)	SUB	0	0	减
JMP	0	A	无条件转(A 为转移地址)	MULT	0	0	乘
JPC	0	A	栈顶值为 0 时转(A 为转移地址)	IDIV	0	0	除
RED	0	A	读指令(A 为 0 或 1 分别表示读整型或字符型)	EQ	0	0	等于
WRT	0	A	写指令(A 为 0 或 1 分别表示写整型或字符型)	NE	0	0	不等于
CAL	L	A	转子（L 为层次，A 为过程入口）	LS	0	0	小于
RETP	0	0	过程返回	LE	0	0	小于等于
UDIS	L	A	调整 display(L 为被调过程层，A 为调用过程层)	GT	0	0	大于
OPAC	0	0	打开活动记录	GE	0	0	大于等于

ENT P	L	A	进入过程(L 为层次，A 为局部空间大小)	

PL 编译程序产生的中间代码程序（.PLD）可由 INTERPRET 解释执行。

二、运行时的数据空间组织

PL 语言允许过程的递归调用，因此采用栈式存贮分配方法进行存贮管理。每次调用一个过程时，就在运行数据栈上为该过程分配所 需的数据空间;每当过程结束时，就是放这一部分空间。这段数据空间成为过程的活动记录。PL 编译程序为每个过程安排的活动记录的形式如图 17.4.2 所示。

其中：

返回地址，存放调用过程的下一条指令地址。

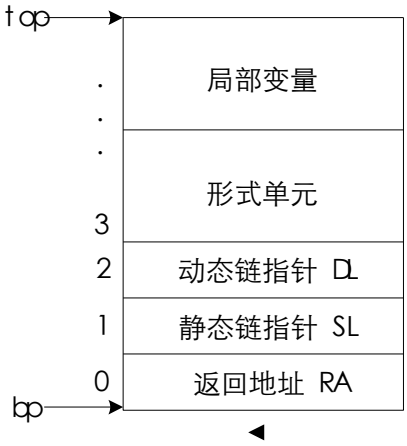


图 17.4.2 过程的活动记录

静态链指针，指向本过程的直接外层过程的活动记录在运行栈中的起始地址。

动态链指针，指向本过程的调用过程的活动记录在运行栈中的起始地址。

形式单元，存放实参的值（对值参数）或地址（对变量参数）。

局部变量，为过程中所定义的变量分配的空间。

PL 编译程序给每个变量分配的地址都是在其活动记录中的相对地址，因此，变量在运行栈中的绝对地址可由活动记录的基址加上变量的相对地址算出。

由于 PL 语言允许过程嵌套定义，内层过程运行时可能引用外层过程中所定义的非局部变量。因此，必须知道当前运行过程及其各外层过程的活动记录的基址。为此我们在程序运行时建立了一个运行时的嵌套层次显示表 display，它记录了正在执行的过程及其所有外层过程的活动记录的基址。请读者注意编译时的 display 和运行时的 display 的区别。

```
program p;
  var x,y : integer;
    ...
  procedure p1;
    var i,j : integer;
      ...
    procedure p11(a,b : integer);
      ...
      begin
        ...
      end;
    begin
      ...
      call p11(i,j);
      ...
    end;
  procedure p2
    var s,t : integer;
      ...
    procedure p21;
      ...
      begin
        ...
      end;

    begin
      ...
      call p1;
      ...
    end;

  begin
    ...
    call p2;
    ...
  end.
```

图 17.4.3 示例程序结构

作为一个例子，我们假设某程序的形式如图 17.4.3 所示。如果程序执行时的调用次序为 P→P2→P1→P11 正在执行时，运行栈 S 及 display 表的情况如图 17.4.4 所示。

通过 display 表，对任何变量 x，设其定义层次为 1，相对地址为 a，可算出其绝对地址 D 为： D=display（1）+2

三、代码生成

PL 编译程序采用与法制导方法进行语义分析和中间代码的翻译。语义分析和中间代码生成工作嵌入在各语法分析子程序中，随着语法分析的步步进展，每当识别出一个个语法成分时。相应的语义动作和翻译工作也同时进行。

1．说明部分的处理

PL 语言允许有常量说明、类型说明、变量说明和过程说明。对于说明语句，编译程序不须产生代码（过程体除外），所要做的予以工作主要是把说明的名字（标识符）及其属性填入到有关表格。

（1）常量说明

每个常量说明的形式为

<标识符>“□”<常量>

处理完每个常量说明后，编译程序要把常量标识符以及它代表的常量的类型（typ）和值（val）填入名字表（nametab）中。为此，处理<常量>的过程 CONSTANT 带有一个变量参数 C，用于返回常量的类型和值，C 为一个记录变量，其类型为：

```
TYPE CONSTREC = RECORD
                  TP：TYPES;
                  I：INTEGER
                END
```

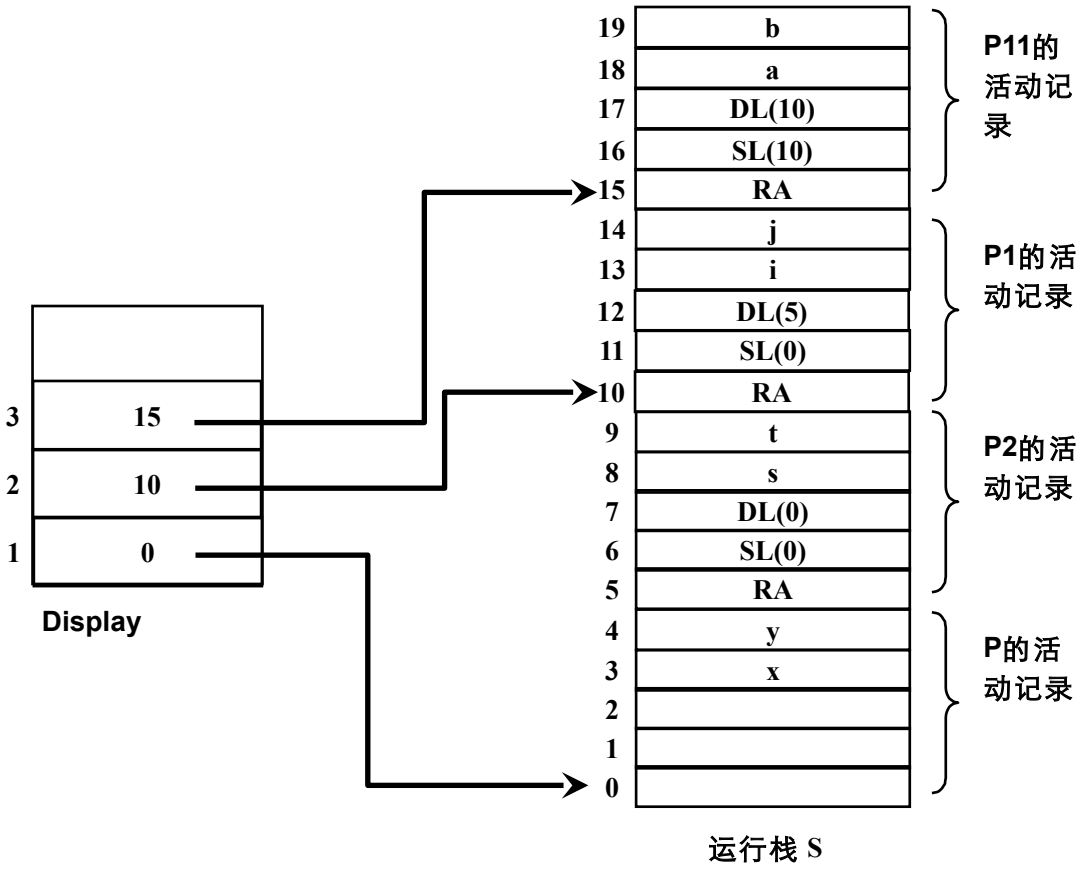


图 17.4.4 运行栈示例

其中，TP 用于表示常量的类型;I 表示常量的值，若常量为整型，则 I 为该常量值，若常量为字符型，则 I 存放该常量的 ASCII 码，若常量为布尔型，则 i 存放 0(false)或 1（true）。

（2）类型说明

PL 语言每个类型说明的定义为

<类型说明>::=<标识符>“=”<类型>

<类型>::=<类型标识符>|<数组类型>

编译程序中过程 TYP 专门处理（类型）。对于预定义的基本类型或程序中已定义的类型标识符，可直接查 nametab 表，找到该类型的相应信息。对于数组类型说明，则由过程 ARRAYTYP 进行处理，收集数组信息并填入数组信息表 atab 中。

值得注意的是，PL 编译程序把多维数组处理为数组的数组。如对下面类型说明

a=array[1..10,1..10]of integer;

编译后，有关表格栏目信息如图 17.4.5 所示。

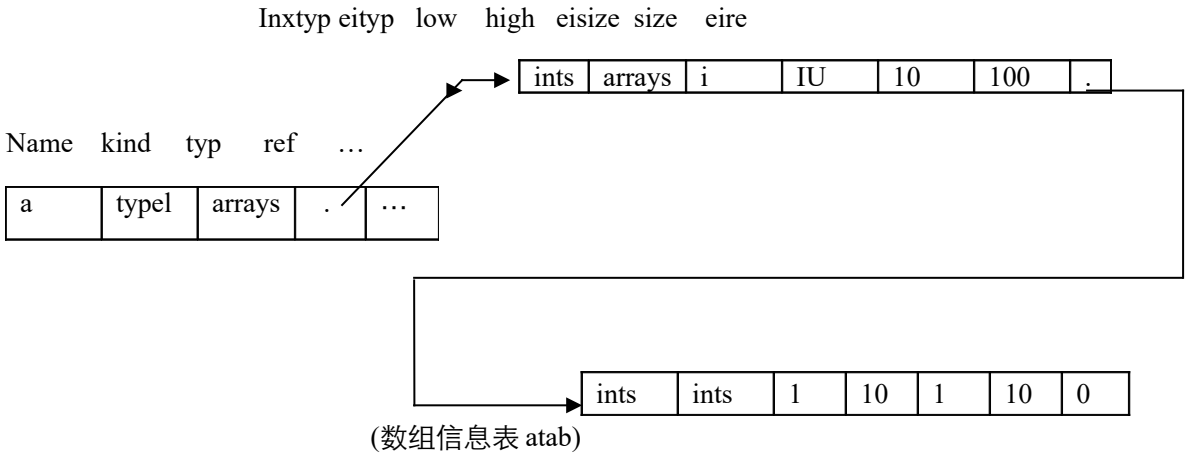


图 17.4.5 数组信息示例

（3）变量说明

变量说明的形式为

<标识符>{“，”<标识符>}“:”<类型>“;”

对于变量标识符，除了要确定变量的类型(这一点处理上与类型说明类似)之外,还要根据该类型数据所占空间的大小（size）为变量分配贮存空间,并赋予变量一个地址。由于 PL 语言采用栈式存贮分配方式，所以为每个变量分配的地址是其所在过程活动记录的相对地址。

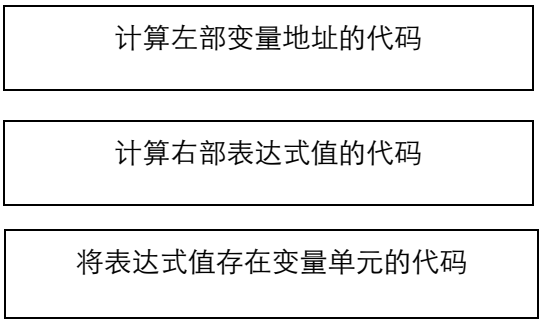
PL 编译程序使用 DX 作为相对地址分配的计数器。在处理每个程序体（block）的 开始 DX 被初始化为 3，即每个过程的活动记录的开头应留出 3 个单元，作为存放返回地址、静态链指针、动态链指针的空间。

2．表达句与赋值语句的代码

赋值语句的形式为

<变量>“=”<表达式>

其相应的目标代码结构为



例如，设 x、y、z 都是整型变量，则对

x=y+z

生成的代码序列为：

loda lev(x) adr(x);装入 x 的地址

lod lev(y) adr(y);装入 y 的值

lod lev(z) adr(z) ; 装入 z 的值

add 0 0 ; y+z

sto 0 0 ;将 y+z 的值存入 x 中

其中，lev(x)表示 x 的定义层次，adr(x)的相对地址，等等。

在对表达式、赋值语句等进行处理时要对数据的类型的一致性进行检查。为了确定类型，我们在过程 EXPRESSION、SIMPLEEXPRESSION、TERM、FACTOR 中均设置了一个变量形参 x，过他返回类型。

对应于变量形参，由于形式单元中存放的是实参的地址，因此，当对变量形参的值进行访问时，应采用间接访问。PL 编译程序通过名字表中的 normal 栏强调某个变量是否为变量形参。

对于下标变量（即数组元素）的访问必须首先计算数组元素的地址，设数组 a 为

var a:array[l1..h1,..,ln..hn]of integer

则数组元素 a[e1,e2,…,en]的地址为

$$\text{A 的首地址}+\sum_{k=1}^{n}(e\ \text{k-eisizek})$$

其中 eisize 可以通过查找数组信息 atab 获得。数组元素的代码产生实现请阅读过程 ARRAYELEMENT。

3. 控制语句的代码

PL 语言有两个控制语句。if 语句和 while 语句。对控制语句进行翻译时，为了确定转移地址，通过须采用“回填”技术。

（1） if 语句

if 语句的一般形式为

if<表达式 e> then <语句 S> 或

if<表达式 e> then <语句 S1> else <语句 S2>

PL 编译程序把他们分别翻译为图 17.4.6 所示的结构。

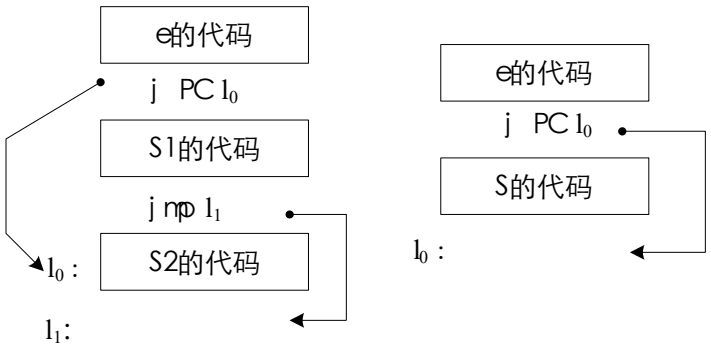


图 17.4.6 if 语句的代码结构

从目标代码结构看出，当生成 jpc（条件为假时转移）和 jmp（无条件转移）指令时，转移地址 1`1 尚不知道。对不带 else 部分的确 if 语句，只有在生成 S 的代码后才知道;对带 else 部分的 if 语句，只有分别在生成 jmp 指令和 S2 的代码之后才能知道。因此，在生成指令 jpc 和 jmp 时，应记下当前指令的位置，便在适当时刻回填的转移地址。读者从过 IFSTATEMENT 可以看出产生 if 语句代码的实现细节。

（2）while 语句

while 语句的形式为

while<表达式 e>do<语句 S>

其目标代码结构如图 17.4.7 所示。

while 语句的代码生成与 if 语句一样存在回填问题，其实现细节请阅读过程 WHILESTATEMENT。

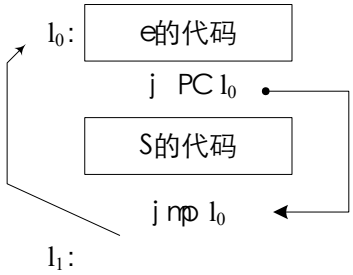


图 17.4.7 while 语句的代码结构

4．过程调用、过程进入和过程返回

编译程序在实现过程时应考虑：在过程调用时，首先应把实参的信息(值或地址)传递给被调过程，其次应保留过程执行完毕后的返回地址，最后完成转子工作;进入过程之后，首先要建立必要的连接数据并分配过程活动记录所需空间;过程执行完毕后，除了按返回地址返回调用程序外，还要释放过程所占的活动记录的空间。

（1）过程调用

对每条形如

call P(e_1 , e_2, … ,e_n)

的过程调用语句，PL 编译程序产生的代码结构如图 17.4.8 所示。

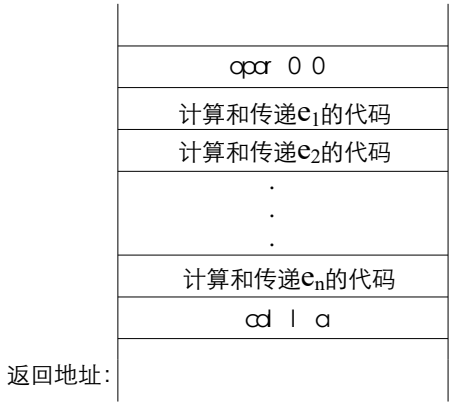


图 17.4.8 过程调用的代码结构

其中，opac 指令的作用是“打开”被调用过程的活动记录空间，为参数的计算和传递作准备，cal la 的作用是保留返回地址及静态链指针和动态链指针的值，把程序控制转移到被调过程。1 和 a 分别是被调过程的层次和入口地址。

对于每个实参 e_i（$1 \leqslant i \leqslant n$）,如 e_i对应的形参为值参数，则 e 可以使一个表达式，传递给形参的是表达式的值，因此 e_i对应的代码就是计算这个表达式的代码，结果值在运行栈

的栈顶（即 e_i 对应的形式单元）;如果 e_i 对应的形参为变量参数，则 e_i 只能是一个变量（包括下标变量），因此 e_i 对应的代码就是装入（或计算）这个变量的地址的代码。

（2）过程进入和过程返回

PL 编译程序对每个过程的执行部分产生如图 17.4.9 所示的代码结构。其中，entp 为过程进入指令，它的作用是建立过程所需活动记录空间；retp 为过程返回指令，它的作用是释放过程所占活动记录的空间、恢复调用前的运行栈状态、根据返回地址返回到调用程序。

图 17.4.9 过程体的代码结构

图 17.4.10 中，我们给出了过程调用、过程进入、过程进入、过程返回各个时刻，运行栈的变化情况。

现在我们来考察运行时 display 表的变化情况。当过程被调用时，随着调用的层层深入，个过程活动记录的基地址被填入了 display 表中，如果是外层过程调用内层过程的话，当过程返回时 display 表的值仍保持调用前的情况;但是，如果是内层过程调用外层过程，则情况就不同。

以前面的图 17.4.3 所示的程序结构为例。假设程序执行时的调用次序为 P→P2→P21→P1，则当 P21 调用 P1 之前，运行栈 display 表如图 17.4.11（a）所示，当进入 P1 之后，运行栈以及 display 表如图 17.4.11（b）所示。因此，当从 Pl 返回到 P21 的调用处时，要完成恢复调用前 display 的工作，这项工作由指令 udis 完成，该指令的功能是沿着静态键重建 display 表。

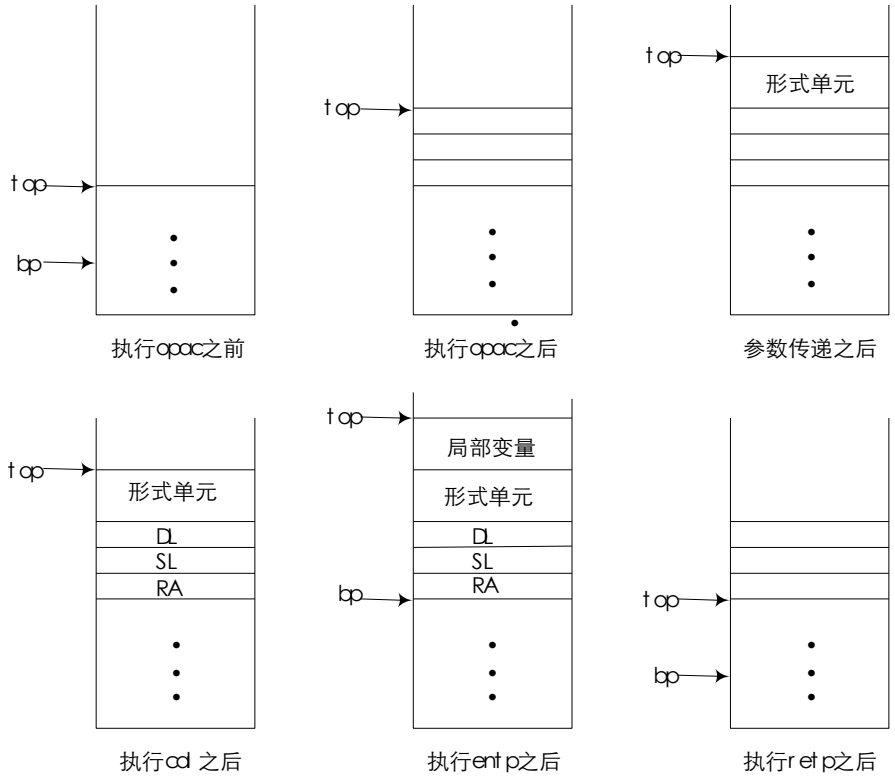


图 17.4.10 过程调用前后运行栈变化情况

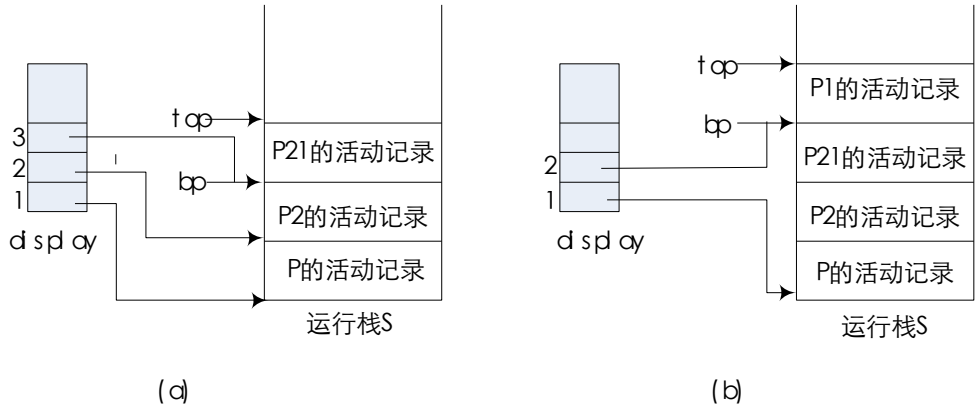


图 17.4.11 运行时 display 变化

5 例

下面，给出一个完整程序的中间代码。改程序读入一个数 n，求 n 的结成。采用递归过程，其 PL 语言程序如下：

```
program pp;
 var n,p;integer;
 procedure p1(n:integer;var p;integer);
  begin
   if n<=1 then p=1
        else begin
        call p1(n-1,p);
        p=n * p
       end
  end;
 begin
  call read(n);
  call p1(n,p);
  call write(p)
 end.
```

编译后产生的中间代码为

```
0  JMP    0  24
1  JMP    0   2
2  ENTP   2   5
3  LOD    2   3
4  LIT    0   1
5  LE     0   0
6  JPC    0   11
7  LOD    2   4
8  LIT    0   1
9  STO    0   0
10 JMP    0   23
11    OPAC      0      0
```

12	LOD	2	3
13	LIT	0	1
14	SUB	0	0
15	LOD	2	4
16	CAL	1	2
17	ADIS	1	2
18	LOD	2	4
19	LOD	2	3
20	ILOD	2	4
21	MULT	0	0
22	STO	0	0
23	RETP	0	0
24	ENTP	1	5
25	LODA	1	3
26	RED	0	0
27	OPAC	0	0
28	LOD	1	3
29	LODA	1	4
30	CAL	1	2
31	LOD	1	4
32	WRT	0	0
33	ENDP	0	0

四、 实习提要

（一）目的与要求

通过一个实际的语义分析和中间代码产生的实现，理解和掌握自顶向下分析中语法制导翻译的基本思想与实现方法。要求对扩充的语言成分严格的按照其语义生成的相应的中间代码。

（二）实习题

1.增加 repeat、for、case 等语句

为了实现这些语句到 PL 抽象机代码的翻译，首先对每种语句，根据语句的语义确定其代码结构，然后把代码产生工作嵌入到相应分析程序的适当位置。实现这些语句时，可根据需要扩充 PL 抽象机指令集，引入适合这些语句的新指令。

2.扩充类型

扩充类型时，应增加相应的类型一致性检查。

如果要扩充实型，则应实现整型数和实型数混合运用时把整型转换为实型的功能。为了存贮实型数，运行

top→ top→

⋮

3
2
1
0

bp→

局部变量
形式单元
动态链指针
静态链指针
返回地址
函数值

17.4.12 活动记录

栈 S 中的元素可用变体记录表示。

实现记录时，应考虑记录域的访问和地址计算。

3. 扩充函数

函数体的处理与过程体类似，不同的是：○函数返回时要回送函数值;○函数的引用通常与变量一样出现在表达式中。

为了处理函数的返回值，可以在子程序（过程、函数）的活动记录中增加“函数值”一项，如图 17.4.12 所示。

对于过程，含数值一项控者不用。对应于函数返回的指令，硬是函数只保留在运行栈栈顶。

第五节　汇编代码生成

编译程序中，所谓目标代码生成就是生成特定机器上的低级语言代码。PL 抽象机代码是一种不依赖于特定机器的中间代码，可把它翻译为具体的计算机代码。

一、PC 机及其汇编语言简介

为了实现到特定机器语言的翻译，必须对该机器的特点（如体系结构、指令系统、存储管理、寄存器、寻址方式等等）有充分的了解，从而充分利用机器的特征，产生高效的代码。下面对 IBM PC 及兼容机的汇编语言有关内容作一简单介绍，其细节可参阅[6]。

1.寄存器

IBM PC 及其兼容机的寄存器可分为 4 组。

（1）通用寄存组。它包括 AX、BX、CX、DX 等四个 16 位寄存器。这 4 个寄存器对大多数操作可以互换使用，但它们又有各自的特殊用处。AX 最适合作为累加器。BX 通常用作基址寄存器。CX 称为计数寄存器，当用于循环次数计数时，CX 可自动减 1。DX 为数据寄存器，当进行乘除运算时规定数据的高位必须在 DX 中。

（2） 指示器和变址寄存器组。它包括 4 个 16 位寄存器，它们是堆栈指示器 SP、基址指示器 BP、串源变址寄存器 SI、串目的寄存器 DI。SP 和段寄存器 SS 结合在内存中建立堆栈，并从栈顶存取栈中的数据，而 BP 和 SS 结合则不通过栈顶存取堆中的数据。通常 SI、DI 可用于变址寻址。

（3）段寄存器组。它包括 4 个 16 位的段寄存器 CS、DS、SS 和 ES，用来标识当前代码段、数据段、堆栈段和附加段，其主要功能是支持对内存的不同段进行读写。

（4）指令计数器 IP 和标志寄存器 FLAG。指令计数器 IP 总是含有当前代码段的偏移量，依此偏移量可从代码段中取出指令以便执行。标志寄存器 FLAG 中含若干标志位和控制标准位，状态标志位寄存着指令执行结果的状态信息，控制标志位则对 CPU 的某些操作进行干预。

2. 存贮器与堆栈

PC 机有 20 条地址线，且存贮器是以字节为单位的，故直接寻址能力可达 1MB（2^{20} 字节），地址从 00000H 到 FFFFFH。但 PC 对地址的运算只能是 16 位的，与地址有关的寄存器 IP、SP、BP、SI、DI 等也是 16 位的。为了把 16 位地址变换为 20 位地址，PC 对内存采用一种分段技术。通常，段可分为 4 类：代码段，主要用来存放程序;堆栈段，专用来作为堆栈;数据段和附加段，均用来存放程序的各种数据。访问内存时，物理地址有寄存器中存放的段地址和偏移地址决定。

PC 机的堆栈是内存中开辟的一端固定一端活动的存贮空间。对战指示器 SP 始终指向栈顶，通过 SP 实现从栈顶存取数据。堆栈之内的数据也可以通过基址指示器 BP 进行存取，这种情况下则不是通过站定存放数据。与 PL 抽象机运行数据栈不同的是，PC 机栈的伸展方向是从大地址向小地址。

3. 寻址方式

PC 机的指令寻址方式有 8 种：立即寻址、寄存器寻址、直接寻址、寄存器间接寻址、基址寻址、变址寻址、基址加变址寻址、相对寻址。对于指令系统、汇编程序结构等等，在此不作细述。

二、PC 汇编代码产生

为了简单起见，目标代码程序的运行数据栈的组织和程序执行方式仍像在 PL 抽象机中一样。图 17.5.1 画出了目标代码运行时运行栈的情况。其中，SP 为栈顶指针，BP 指向现行运行过程的活动记录的起始地址。

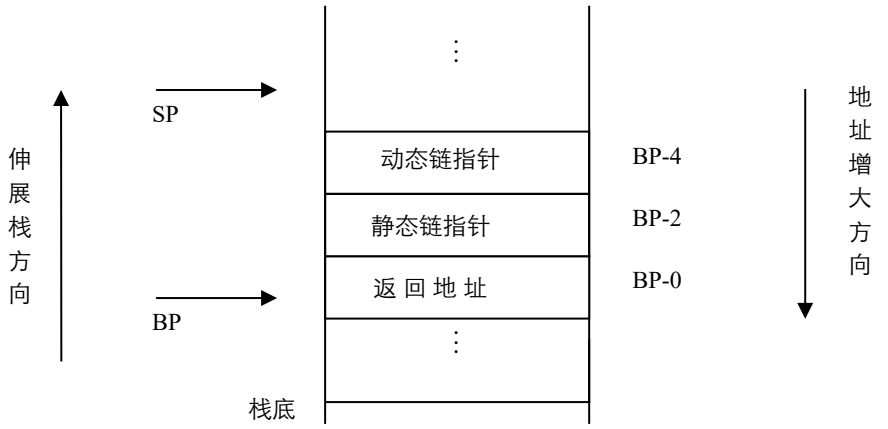


图 17.5.1　目标程序运行栈

运行栈在程序中定义为

```
-STACK SEGMENT PARA STACK’STACK’
DW 2048 DUP（0）
-STACK ENDS
```

在数据中定义了两项数据：

```
DLSP DW 8 DUP（？）
OLDTOP DW
```

DLSP 用于存放运行时的 display 内容。OLDTOP 为临时工作单元。指向过程调用前的栈顶位置。

汇编代码生成程序实际上是一个翻译程序，它把 PL 抽象机代码逐条翻译为 PC 汇编代码，程序结构类似于解释程序 INTERPRET。代码产生程序的细节请阅读 TRANSTOASM 程序。

在产生目标代码时，必须确定转移指令的转移地址，为此，我们再前一遍扫描中建立了一个转移地址表 JMPADRTAB，目标代码生成时要读入这个表。

由于 PC 堆栈是负增长的（见图 17.5.1），与 PC 抽象机运行栈正好相反，另外，PC 机是以字节编址的、每个数据占两个字节。因此，在计算地址时应采取相应变化。为了便于产生目标代码中计算数组元素地址的代码，我们引入了两条中间代码指令 LIT1 和 ADD1,它们在 PC 抽象机中与 LIT 和 ADD 完全一样，但翻译为目标代码时有所不同。

为了产生完整的汇编程序，汇编程序的头尾分别由 ASMHEAD 和 ASMTALL 产生。我们把两个标准过程 read 和 write 定义为外部过程，当涉及到读写时，必须把目标代码与这两个过程连接之后才能运行。

三、实习提要

（一）目的与要求

通过设计、调试一个实际的代码生成程序，理解和掌握目标代码生成的思想和方法。在设计目标代码生成程序时，首先要对目标机器有足够的了解，这样不仅能够生成正确的目标程序，还能充分利用机器的特点，生成更加高效的代码。

（二）实习题

1.将前段所实现的语言成分正确地翻译成汇编代码。

2.本章中没有给出的两个标准过程 read 和 write 的汇编代码，请读者完成之。由 read 读入的数据保留在寄存器 DX 中，要写的数据由寄存器 BX 传递给 write。

3.产生更加优化的目标代码。