编译原理实践教程

序言

《编译原理和技术》的课程实践至少有两种可能的安排。其一，为配合编译课程教学，而安排多次小型实践，分别支持编译程序的各个阶段。其二，针对某一规模适中的语言来设计和实现一个相对完整、独立编译器。

《编译原理实践教程》作为《编译原理和技术》课程的延伸，其目的是让大家动手设计和实现某一规模适中的语言的编译器，该编译器不仅涉及编译程序的各个阶段，而且也强调了编译的总体设计、各个阶段的接口安排等等。

通过上机实践，来设计这个相对完整的编译器，一方面可以使学生增加对编译程序的整体认识和了解——巩固《编译原理和技术》课程所学知识，另一方面，通过上机练习，学生也可以学到很多程序调试技巧和设计大型程序一般的原则，如模块接口的协调，数据结构的合理选择等等。

为了使学生能尽早动手实践，我们建议把实践分成三部分，首先阅读本教程第一部分，在这部分就 PL/0 语言的语法及其编译程序的各个阶段作了简单介绍，以便对 PL/0 编译程序有个初步的印象。其次要认真阅读理解第三部分所给出的 PL/0 编译器源程序，使上一阶段的初步印象得以加深、具体化。最后按照第二部分的实验要求扩充 PL/0 语言的功能并加以实现。

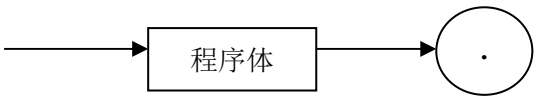
第一部分 PL/0 语言及其编译器

1. PL/0 语言介绍

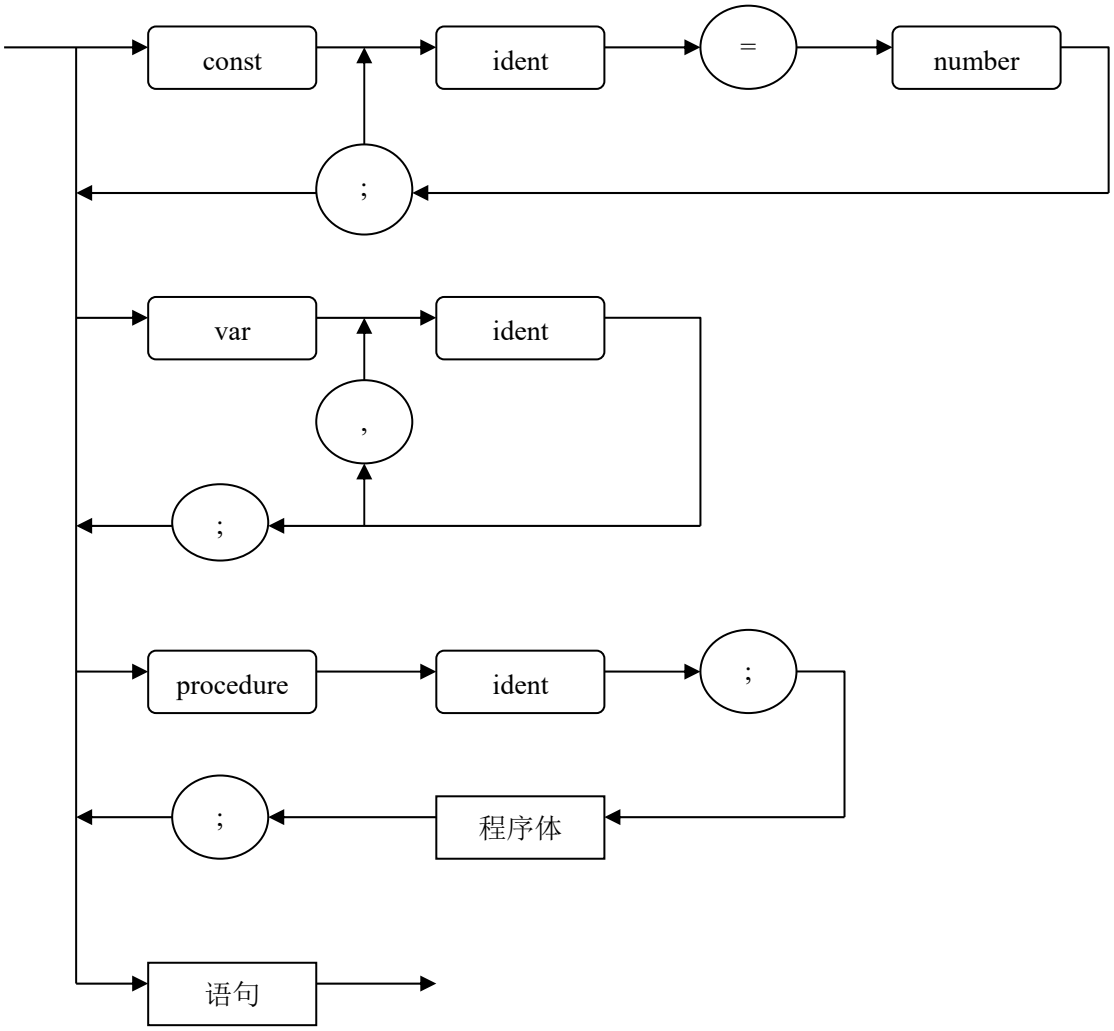
PL/0 程序设计语言是一个较简单的语言，它以赋值语句为基础，构造概念有顺序、条件和重复（循环）三种。PL/0 有子程序概念，包括过程定义（可以嵌套）与调用且有局部变量说明。PL/0 中唯一的数据类型是整型，可以用来说明该类型的常量和变量。当然 PL/0 也具有通常的算术运算和关系运算。具体的 PL/0 语法图如下。

1.1 PL/0 语言的语法图

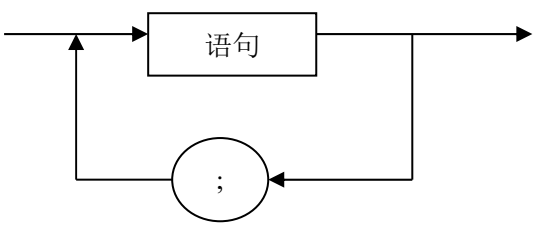
程序



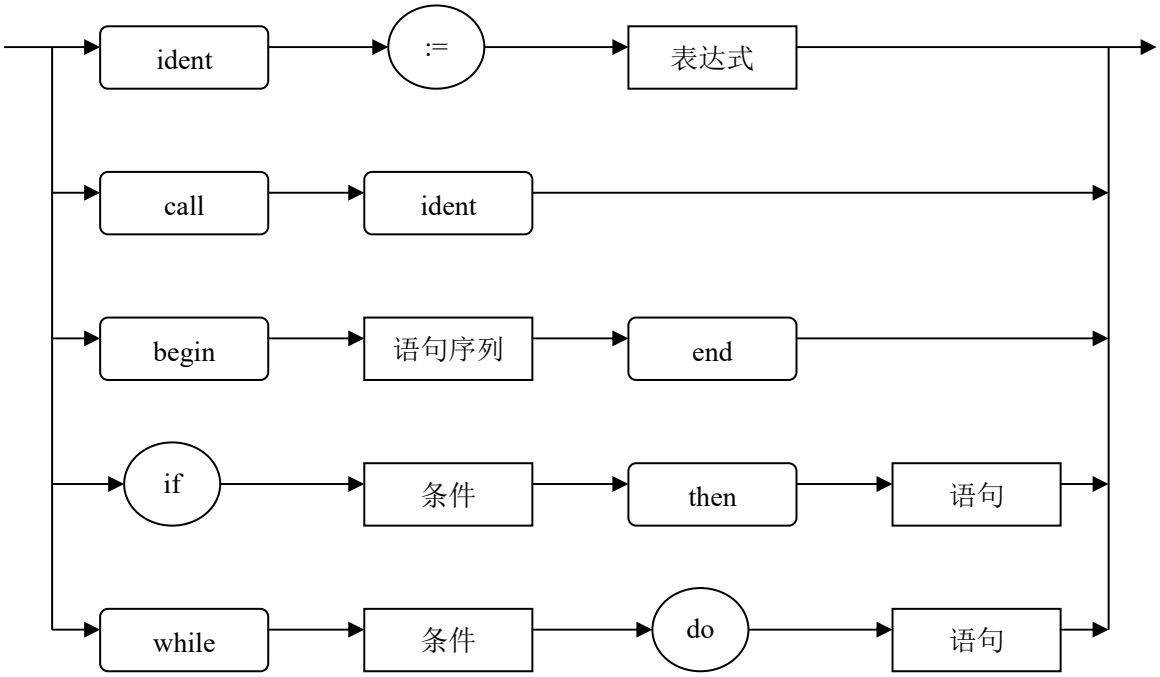
程序体



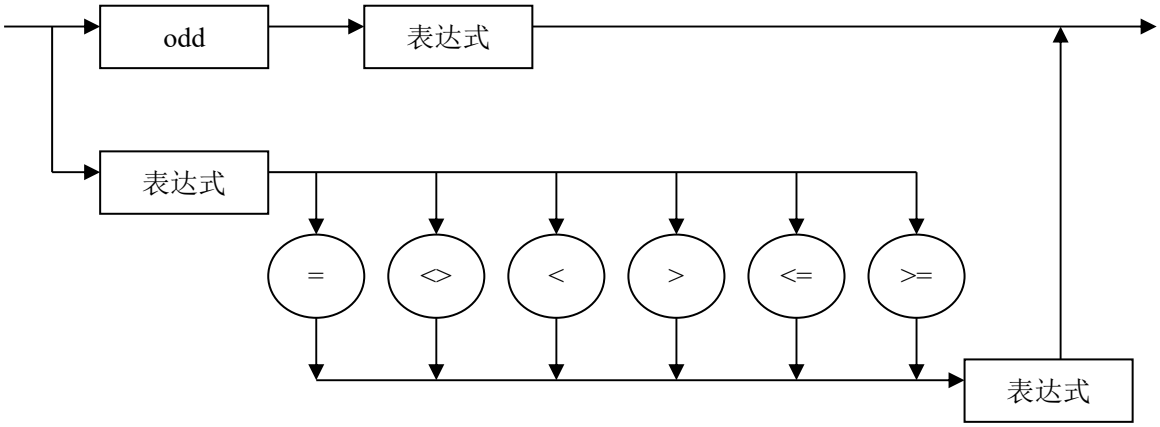
语句序列



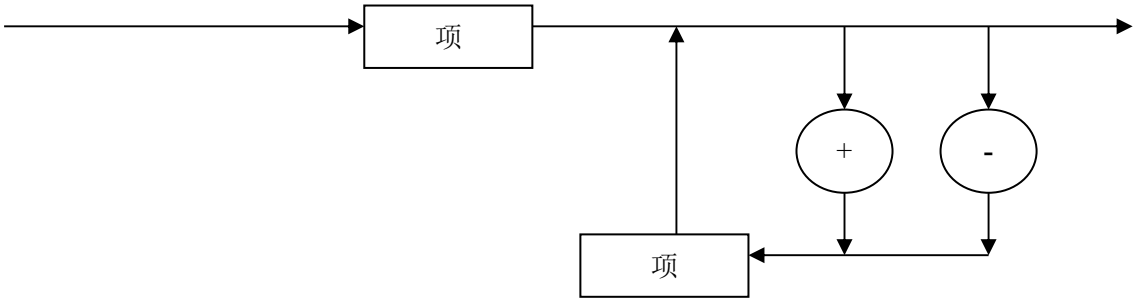
语句


ident
:=
表达式
call
ident
begin
语句序列
end
if
条件
then
语句
while
条件
do
语句

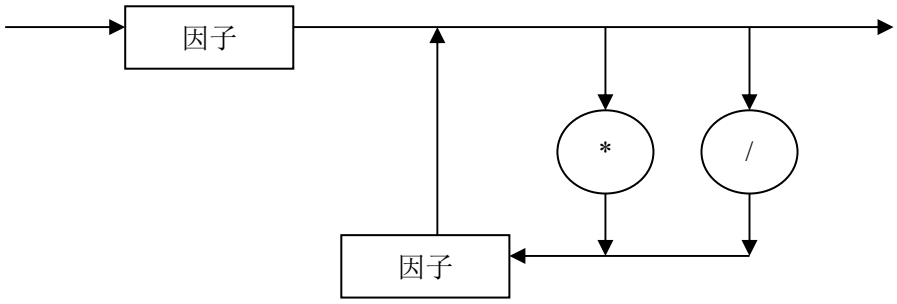
条件


odd
表达式
表达式
=
<>
<
>
<=
>=
表达式

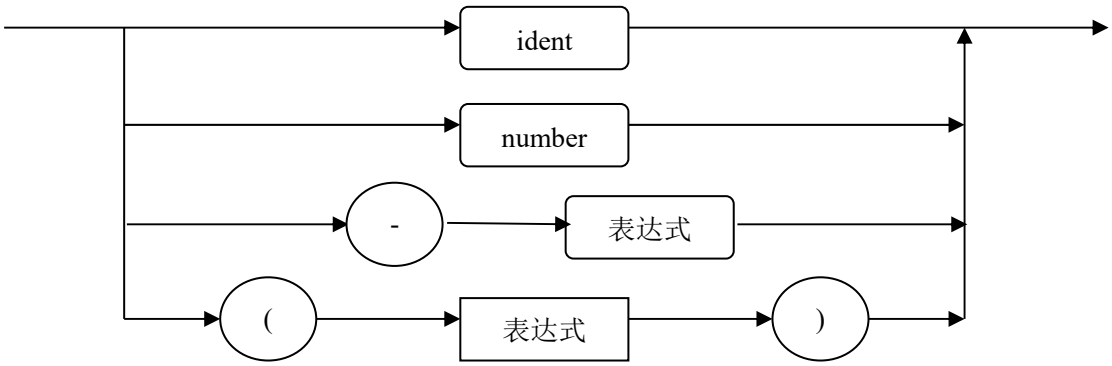
表达式


项
+
-
项

项



因子



2. PL/0 语言编译器

本书所提供的 PL/0 语言编译器的基本工作流程如图 1-1 所示：

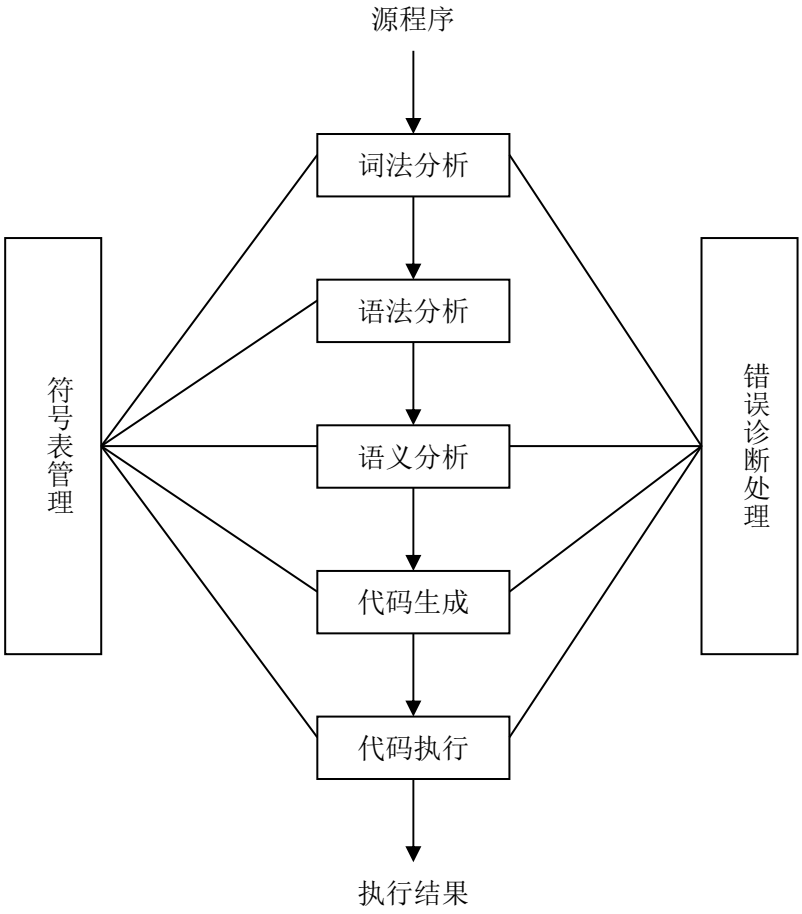


图 1-1 PL/0 编译器基本工作流程

2.1 词法分析

PL/0 的语言的词法分析器将要完成以下工作：

（1） 跳过分隔符（如空格，回车，制表符）；

（2） 识别诸如 begin，end，if，while 等保留字；

（3） 识别非保留字的一般标识符，此标识符值（字符序列）赋给全局量 id，而全局量 sym 赋值为 SYM_IDENTIFIER。

（4） 识别数字序列，当前值赋给全局量 NUM，sym 则置为 SYM_NUMBER；

（5） 识别:=，<=，>=之类的特殊符号，全局量 sym 则分别被赋值为 SYM_BECOMES，SYM_LEQ，SYM_GTR 等。

相关过程（函数）有 getsym()，getch()，其中 getch()为获取单个字符的过程，除此之外，它还完成：

（1） 识别且跳过行结束符；

（2） 将输入源文件复写到输出文件；

（3） 产生一份程序列表，输出相应行号或指令计数器的值。

2.2 语法分析

我们采用递归下降的方法来设计 PL/0 编译器。以下我们给出该语言的 FIRST 和 FOLLOW 集合。

非终结符（S）	FIRST(S)	FOLLOW(S)
程序体	const var procedure ident call if begin while	. ;
语句	ident call begin if while	. ; end
条件	odd + - (ident number	then do
表达式	+ - (ident number	. ;) R end then do
项	ident number (	. ;) R + - end then do
因子	ident number (	. ;) R + - * / end then do

注：表中 R 代表六个关系运算符。

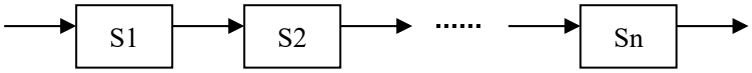
不难证明，PL/0 语言属于 LL（1）文法。（证明从略。）

以下是我们给出如何结合语法图编写（递归下降）语法分析程序的一般方法。假定图 S 所对应的程序段为 T（S），则：

（1） 用合适的替换将语法约化成尽可能少的单个图；

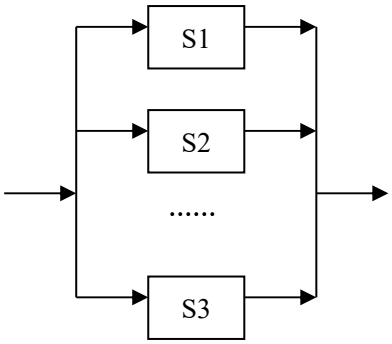
（2） 将每一个图按下面的规则（3）-（7）翻译成一个过程说明；

（3） 顺序图对应复合语句：



对应：begin T(S1); T(S2); ...; T(Sn) end

（4） 选择：

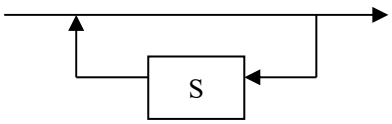


对应：case 语句或者条件语句：

```
case ch of                  if ch in L1 then T(S1) else
  L1: T(S1);               if ch in L2 then T(S2) else
  L2: T(S2);  或           ...
  ...                        if ch in Ln then T(Sn) else
  Ln: T(Sn);               error
```

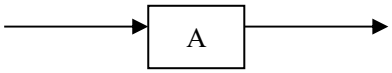
其中 Li∈FIRST（Si），ch 为当前输入符号。（下同）

（5） 循环



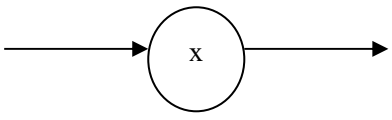
对应：while ch in L do T(S)

（6） 表示另一个图 A 的图：



对应：过程调用 A。

（7） 表示终结符的单元图：



对应：if ch == x then read(ch) else error

相关过程有：

block(), constdeclaration(), vardeclaration(), statement(), condition(), expression(), term(), factor()等。

它们之间依赖关系如图 1-2：

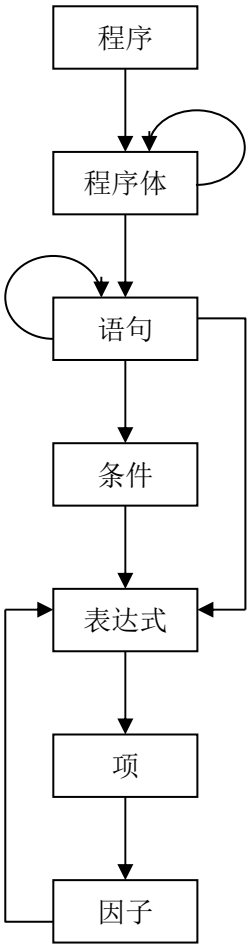


图 1-2 语法分析过程依赖关系

2.3 语义分析

PL/0 的语义分析主要进行以下检查：

（1） 是否存在标识符先引用未声明的情况；

（2） 是否存在已声明的标识符的错误引用；

（3） 是否存在一般标识符的多重声明。

2.4 代码生成

PL/0 编译程序不仅完成通常的词法分析、语法分析，而且还产生中间代码和“目标”代码。最终我们要“运行”该目标码。为了使我们的编译程序保持适当简单的水平，不致陷入与本课程无关的实际机器的特有性质的考虑中去，我们假想有台适合 PL/0 程序运行的计算机，我们称之为 PL/0 处理机。PL/0 处理机顺序解释生成的目标代码，我们称之为解释程序。注意：这里的假设与我们的编译概念并不矛盾，在本课程中我们写的只是一个示范性的编译程序，它的后端无法

完整地实现，因而只能在一个解释性的环境下予以模拟。从另一个角度上讲，把解释程序就看成是 PL/0 机硬件，把解释执行看成是 PL/0 的硬件执行，那么我们所做的工作：由 PL/0 源语言程序到 PL/0 机器指令的变换，就是一个完整的编译程序。

PL/0 处理机有两类存贮，目标代码放在一个固定的存贮数组 code 中，而所需数据组织成一个栈形式存放。

PL/0 处理机的指令集根据 PL/0 语言的要求而设计，它包括以下的指令：

（1）LIT　　/* 将常数置于栈顶 */

（2）LOD　　/* 将变量值置于栈顶 */

（3）STO　　/* 将栈顶的值赋与某变量 */

（4）CAL　　/* 用于过程调用的指令 */

（5）INT　　/* 在数据栈中分配存贮空间 */

（6）JMP, JPC　/* 用于 if, while 语句的条件或无条件控制转移指令 */

（7）OPR　　/* 一组算术或逻辑运算指令 */

上述指令的格式由三部分组成：

F	L	A

其中，f, l, a 的含义见下表：

F	L	a
INT	———	常　　量
LIT	———	常　　量
LOD	层次差	数据地址
STO	层次差	数据地址
CAL	层次差	程序地址
JMP	———	程序地址
JPC	———	程序地址
OPR	———	运算类别

表 2-1 PL/0 处理机指令

上表中，层次差为变量名或过程名引用和声明之间的静态层次差别，程序地址为目标数组 code 的下标，数据地址为变量在局部存贮中的相对地址。

PL/0 的编译程序为每一条 PL/0 源程序的可执行语句生成**后缀式**目标代码。这种代码生成方式对于表达式、赋值语句、过程调用等的翻译较简单。

如赋值语句 X := Y op Z（op 为某个运算符），将被翻译成下面的目标代码序列：（设指令计数从第 100 号开始）

No.	f	L	a
100	LOD	Level_diff_Y	Addr_Y
101	LOD	Level_diff_Z	Addr_Z
102	OPR	————————	op
103	STO	Level_diff_X	Addr_X

而对 if 和 while 语句稍繁琐一点，因为此时要生成一些跳转指令，而跳转的目标地址大都是未知的。为解决这一问题，我们在 PL/0 编译程序中采用了**回填技术**，即产生跳转目标地址不明确的指令时，先保留这些指令的地址（code 数组的下标），等到目标地址明确后再回过来将该跳转指令的目标地址补上，使其成为完整的指令。下表是 if、while 语句目标代码生成的模式。（L1，L2 是代码地址）

if C then S	While C do S
条件 C 的目标代码 JPC -- L1 语句 S 的目标代码 L1: ...	L1: 条件 C 的目标代码 JPC - L2 语句 S 的目标代码 JMP L1 L2: ...

表 2-2 if-while 语句目标代码生成模式

相关过程（函数）有：gen()，其任务是把三个参数 f、l、a 组装成一条目标指令并存放于 code 数组中，增加 CX 的值，CX 表示下一条即将生成的目标指令的地址。

2.5 代码执行

为了简单起见，我们假设有一个 PL/0 处理机，它能够解释执行 PL/0 编译程序所生成的目标代码。这个 PL/0 处理机有两类存贮、一个指令寄存器和三个地址寄存器组成。程序（目标代码）存贮称为 code，由编译程序装入，在目标代码执行过程中保持不变，因此它可被看成是“只读”存贮器。数据存贮 S 组织成为一个栈，所有的算术运算均对栈顶元和次栈顶元进行（一元运算仅作用于栈顶元），并用结果值代替原来的运算对象。栈顶元的地址（下标）记在栈顶寄存器 T 中，指令寄存器 I 包含着当前正在解释执行的指令，程序地址寄存器 P 指向下一条将取出的指令。

PL/0 的每一个过程可能包含着局部变量，因为这些过程可以被递归地调用，故在实际调用前，无法为这些局部变量分配存贮地址。各个过程的数据区在存贮栈 S 内顺序叠起来，每个过程，除用户定义的变量外，还摇篮有它自己的内部信息，即调用它的程序段地址（返回地址）和它的调用者的数据区地址。在过程终止后，为了恢复原来程序的执行，这两个地址都是必须的。我们可将这两个内部值作为位于该过程数据区的内部式隐式局部变量。我们把它们分别称为返回地址（return address）RA 和动态链（dynamic link）DL。动态链的头，即最新分配的数据区的地址，保存在某地址寄存器 B 内。

因为实际的存贮分配是运行（解释）时进行的，编译程序不能为其生成的代码提供绝对地址，它只能确定变量在数据区内的位置，因此它只能提供相对地址。为了正确地存取数据，解释程序需将某个修正量加到相应的数据区的基地址上去。若变量是局部于当前正在解释的过程，则此基地址由寄存器 B 给出，否则，就需要顺着数据区的链逐层上去找。然而遗憾的是，编译程序只能知道存取路线

的表态长度，同时动态链保存的则是过程活动的动态历史，而这两条存取路线并不总是一样。

例如，假定有过程 A，B，C，其中过程 C 的说明局部于过程 B，而过程 B 的说明局部于过程 A，程序运行时，过程 A 调用过程 B，过程 B 则调用过程 C，过程 C 又调用过程 B，如下图所示：

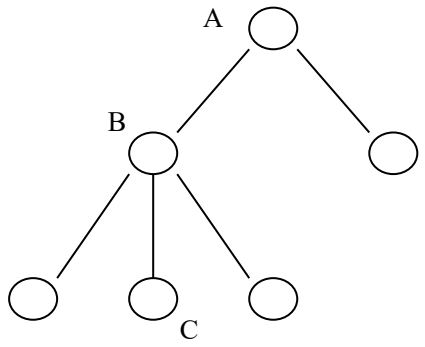
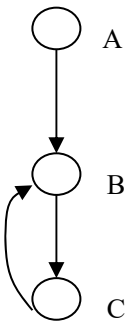
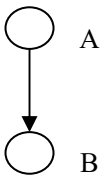


图 2-1　过程说明嵌套图

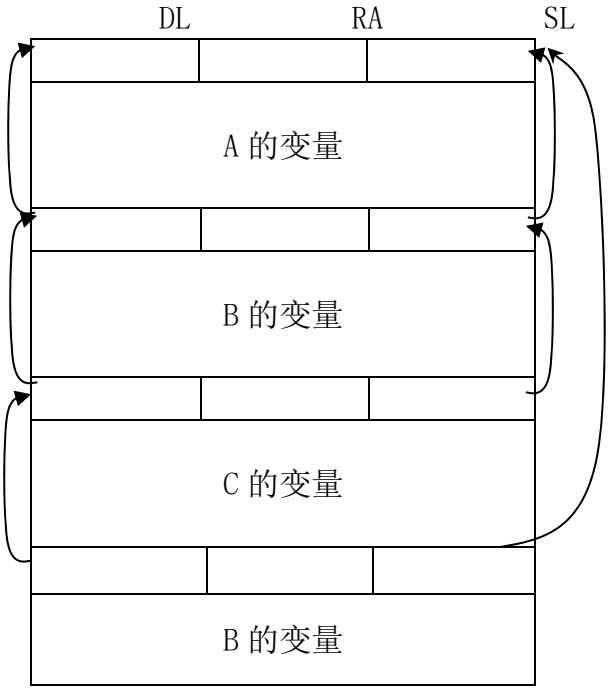


过程调用图



表示 A 调用 B

从静态的角度我们可以说 A 是在第一层说明的，B 是在第二层说明的，C 则是在第三层说明的。若在 B 中存取 A 中说明的变量 a，由于编译程序只知道 A，B 间的静态层差为 1，如果这时沿着动态链下降一步，将导致对 C 的局部变量的操作。为防止这种情况发生，有必要设置第二条链，它以编译程序能明了的方式将各个数据区连接起来。我们称之为静态链（static link）SL。这样，编译程序所生成的代码地址是一对数，指示着静态层差和数据区的相对修正量。下面我们给出的是过程 A、B 和 C 运行时刻的数据区图示：



有了以上认识，我们就不难明白 PL/0 源程序的目标代码是如何被解释执行的。以语句 X ：= Y op Z 为例，（该语句的目标代码序列我们已在 2.4 节给出），PL/0 处理机解释该指令的“步骤”如下：

```
step 1,
  S[++T] ← S[base(level_diff_Y) + addr_Y];
    // 将变量 Y 的值放在栈顶
step 2,
  S[++T] ← S[base(level_diff_Z) + addr_Z];
    // 将变量 Z 的值放在栈顶，此栈顶元为变量 Y 的值
step 3,
  T--;
    // 栈顶指针指向次栈顶元，即存放结果的单元
step 4,
  S[T] ← S[T] op S[T + 1];
    // 变量 Y 和变量 Z 之间进行“op”操作
step 5,
  S[base(level_diff_X) + addr_X] ← S[T];
    // 将栈顶的值存放到变量 X 所在的单元
step 6,
  T--;
    // 栈顶指针减一
```

相关过程：base()，interpret()。其中 base()的功能是根据层次差并从当前数据区沿着静态链查找，以便获取变量实际所在的数据区其地址；interpret()则完成各种指令的执行工作。

2.6 错误诊断处理

一个编译程序，在多数情况下，所接受的源程序正文都是有错误的。发现错误，并给出合适的诊断信息且继续编译下去从而发现更多的错误，对于编译程序而言是完全必要的。一个好的编译器，其特征在于：

- ◆ 任何输入序列都不会引起编译程序的崩溃。
- ◆ 一切按语言定义为非法的结构，都能被发现和标志出来。
- ◆ 经常出现的错误，程序员的粗心或误解造成的错误能被正确地诊断出来，而不致引起进一步的株连错误。

根据这样的要求，我们为 PL/0 编译程序制定了以下两条规则：

（1） 关键字规则；程序员在写程序时，可能会因为粗心而漏掉语句的分隔符——“;”，但他决不会漏掉算术运算符“+”，对于编译程序而言，不论是分隔符号类的符号还是关键字符号类的符号，它们都具有同等重要的地位。基于这样的特点，我们可以采用不易出错的部分来作为恢复正常步调的标记。每当遇到错误时，分析程序跳过后面的某些部分，直到出现所期望的符号为止。对于程序设计语言来说，这种符号

（称为同步符号）的最好选择就是关键字。PL/0 的每一种构造语句以 begin、if 或 while 开头；每种说明则以 var、const 或 procedure 开头。每遇到错误时，编译程序便可跳过一段程序，直到遇到这类符号为止，而继续编译。

（2） 镇定规则；自顶向下分析的特点在于目标对分成一些子目标，分程序则用别的分析程序来处理其子目标。镇定规则是说一个分析程序发现了错误，它不应该消极地停止前进，仅仅向调用它的程序报告发生的错误；而应该自己继续向前扫描，找到似乎可以使正常的分析得以恢复的地方。这一规则在程序设计上的含义就是任一分析程序除了正常终止外，没有其它出口。

对于镇定规则，一个可能的严格解释为：一旦发现非法结构，即跳过后面的输入正文，直到下一个可以正确地跟随当前正在分析的句子结构的符号为止。这意味着每一分析程序需知道其当前活动结点的后继符号集合。

为了找到这个后继符号集合，我们给对应语法图的每一个分析过程提供一个显式参数，set，它指明可能的后继集合。不过在任何条件下，如果都跳到输入正文中下一个这种后继符号出现的地方，未免太短视了。程序中所含的错误可能只不过是漏掉了一个符号（如“;”）而已，由此而忽略去源程序的符号集合中，再凑加一些关键字，它们用于标记那些不容忽略的结构的开始符，因此，作为参数传递给分析过程的那些符号就不仅是后继符号了。

对于这样的符号集，我们采用这样的计算策略：先用一些明显的关键符号给它赋初值，然后随着分析子目标的层次深入，逐步补充别的合法符号。为了灵活起见，我们引入 test 子程序来实现所说的验证工作。

test 过程有三个参数：

（1） 可允许的下一个符号集合 S1，如果当前符号不在此集合中，当即得到一个错误号；

（2） 另加的停止符号集合 S2，有些符号的出现，虽然无疑是错的，但它们绝对不应被忽略而跳过；

（3） 整数 n，表示有关错误的诊断号：

```
void test(symset s1, symset s2, int n)
{
   symset s;

   if (! inset(sym, s1))
   {
       error(n);
       s = uniteset(s1, s2);
       while(! inset(sym, s))
           getsym();
       destroyset(s);
   }
}
```

我们前面提出的方案，具有这样的性质：试图通过略过输入正文中的一个或多个符号来恢复分析的正常步调。在错误仅为漏掉一个符号所引起的情况下，它都是不适宜的策略。经验表明，这类错误基本上限于那种仅有语法作用，而不代

表动作的符号（如“;”）。把一些关键字加到后继符号集合中去可使分析程序不再盲目地跳过后面的符号，好象漏掉的已经补上去一样。下面程序段就是 PL/0 分析程序中复合语句分析的一小段。它的效果等于关键字前插入漏掉的分号。statbegsys 集合是“语句”这个结构的首符号集。

```
if (sym == SYM_BEGIN)
    {   getsym();
        set1 = createset(SYM_SEMICOLON, SYM_END, SYM_NULL);
        set = uniteset(set1, fsys);
        statement(set);
        while (sym == SYM_SEMICOLON || inset(sym, statbegsys))
        {
            if (sym == SYM_SEMICOLON)
            {
                getsym();
            }
            else
            {
                error(10);
            }
            statement(set);
        } // while
        destroyset(set1);
        destroyset(set);
        if (sym == SYM_END)
        {
            getsym();
        }
        else
        {
            error(17); // ';' or 'end' expected.
        }
    }
```

相关过程：test(), inset(), createset, uniteset(), error().

2.7 符号表管理

为了组成一条指令，编译程序必须知道其操作码及其参数（数或地址）。这些值是由编译程序本身联系到相应标识符上去的。这种联系是在处理常数、变量和过程说明完成的。为此，标识符表应包含每一标识符所联系的属性；如果标识符被说明为常数，其属性值为常数值；如果标识符被说明成变量，其属性就是由层次和修正量（偏移量）组成的地址；如果标识符被说明为过程，其属性就是过程的入口地址及层次。

常数的值由程序正文提供，编译的任务就是确定存放该值的地址。我们选择

顺序分配变量和代码的方法；每遇到一个变量说明，就将数据单元的下标加一（PL/0 机中，每个变量占一个存贮单元）。开始编译一个过程时，要对数据单元的下标 dx 赋初值，表示新开辟一个数据区。dx 的初值为 3，因为每个数据区包含三个内部变量 RA，DL 和 SL。

相关过程：enter()，该函数用于向符号表添加新的符号，并确定标识符的有关属性。

2.8 其他

本教程所提供的 PL/0 编译程序包括词法分析、语法分析、错误诊断、代码生成、解释执行等几部分。关于这几个程序，我们做如下说明：

（1） 每一个分程序（过程）被编译结束后，将列出该部分 PL/0 程序代码。这个工作由过程 listcode()完成。注意，每个分程序（过程）的第一条指令未被列出。该指令是跳转指令。其作用是绕过该分程序的说明部分所产生的代码（含过程说明所产生的代码）。

（2） 解释程序作为 PL/0 编译程序的一个过程，若被编译的源代码没有错误，则编译结束时调用这个过程。

（3） PL/0 语言没有输出语句。解释程序按执行次序，每遇到对变量的赋值就输出其值。

第二部分 上机实践要求

“编译原理与技术”的上机实验要求你对 PL/0 语言及其编译器进行扩充和修改。每个扩充或修改方式可得到不同的分数，满分为 100 分。

对现有的 PL/0 编译程序所做修改，分为基础修改【必做】和提高扩展【选做】。

- 基础修改，实现（1）-（5），（小计 70 分）。

（1） 添加注释（5 分）
块注释由/*和*/包含，不允许嵌套。
行注释由//开始直到行结束符（回车）。

（2） 扩展 PL/0 中“条件”:（15 分）
这种修改包括：
（i） 增加逻辑运算符 **&&** 、|| 和 **!**
（ii） 把 PL/0 语言中的“条件”概念一般化为 C 语言那样(表达式值非零即为“真”)。
（iii） “条件”的短路计算。

（3） 添加数组（10 分）
在 PL/0 中允许有数组变量声明/对数组元素赋值/在表达式中引用数组元素等。可以有多维数组，数组的维度范围设为常量：
dimDeclaration → [const] dimDeclaration
dimDeclaration → ε
const → **ident** | **number**

为简单起见，PL/0 中基本数据类型就是整型。
这样，在变量声明部分可以出现:

```
var i,j,k; //三个变量均为整型变量
```

也可以出现：

```
var  i,j,k[10][10];
```

//i,j 为普通整型变量，而 k 为二维整型数组

（4） 参数传递。实现**传值调用**，如传递常量值，或普通变量/数组元素的值。并进行简单的语义检查(如实参和形参个数/类型的对应等)。(15 分）
例如，过程（头）定义：procedure func(**p**,**q**,**r**)…在此过程定义中，有三个形式参数 p,q,r。

（5） 添加语句实现:（25 分）
else/**elif** 子句/**exit** 语句（退出当前执行的 PL/0 程序）（5 分）
return 语句及**返回值**的实现（10 分）
for 语句实现。例如：for(i:=0;i<10;i:=i+1)…。语法/语义参照 C 语言，也可设计++，--等算符。(10 分）

- 提高扩展，从（6）-（13）中**任选若干项加以实现**（小计 30 分）

（6） 给 PL/0 添加内置函数 **random** 和 **print**。例如，函数 random 的调用形式可以是 random()或者 random(100)，前者返回一个任意的随机自然数，而后者则返回一个小于 100 的随机自然数。输出函数 print，调用形式可以是，print()或者 print(i,j)，前者可以用来换行，后者则输出变量 i,j 的值。（10 分）

（7） 给 PL/0 添加内置函数 **CALLSTACK**。该函数可以按照调用的先后次序，输出在运行时栈中存放的正在执行的各个过程/函数的活动记录相关信息（如程序计数器，参数值等）。（10 分）

（8） 更多的 **C 风格的运算表达式**实现。语法/语义参照 C 语言。（10 分）
例如，max := i > j **?** i **:** j; // ? : 运算
例如，min := i **<<** 2;//左移操作
再例如：i := **j := k := 100**;//赋值表达式

（9） **实现传地址调用**。（10 分）
例如，过程（头）定义：procedure func(**p,&q,r[2][3]**)... 在此过程定义中，有三个形式参数 p,q,r。又设变量定义：var i,j,k[2][3];则过程调用语句 func(i,j,k),将把变量 i 的值，变量 j 的地址，数组 k 的首地址传入。

（10）**过程作为参数传递的实现。（15 分）**
例如：过程（头）定义：procedure func(**p,q,r(i,j)**)... 在此过程定义中，有三个形式参数 p,q,r，其中 r 是一个有两个整型形参的过程型参数。而在 func 过程体中，过程参数 r 的调用形式可以是 r(p,q)或 r(1,2)等等。

（11）**goto** 语句/**break** 语句（跳出包含它的最内层循环）/**continue** 语句（继续执行包含它的最内层循环）的实现。语法/语义参照 C 语言。（15 分）

（12）**do while** 语句/**switch** 语句的实现。语法/语义参照 C 语言。（15 分）

（13）**加强的 PL/0 变量的定义/初始化及其实现。（20 分）**
例如，var i,**j = 10**,k;将定义整型变量 i,j,k,其中 j 还有初值 10。
例如，var **a[]** = {1,2,3,4,5};将定义变量 a 是一个长度为 5 的一维数组。而 var **b[][2]**={{0,1},{2,3},{4,5}};将定义变量 b 是一个 3*2 已初始化的整型数组。
变量的定义声明/初始化语句可以出现在复合语句（compound statement）中。语法/语义参照 C 语言。

为了实现以上功能，可任意增加 PL/0 处理机的指令。但要注意指令的简单与合理。

完成上机作业后，必须提交**设计文档**，包括：

（1） 修改后的 PL/0 语言文本。包含词法分析（正规式），语法分析（BNF）。

（2） 有关修改后的 PL/0 编译器的说明。详细说明你的编译器是如何编译新的 PL/0 语言程序的。指出你的程序中最精彩的部分，以及你为什么这样做，你是如何控制和恢复语义错误的。如果重做一遍，你又会有哪些新的改进？

第三部分 PL/0 语言编译器源程序

1. 一个例子

1.1 PL/0 语言源程序

下面我们给出一个 PL/0 语言写的二数相乘、求最大公约数的算法：

```
const m = 7, n = 85;
var x, y, z, q, r;

procedure multiply;
var a, b;
begin
    a := x; b := y; z := 0;
    while b > 0 do
    begin
        if odd b then z := z + a;
        a := 2 * a; b := b / 2;
    end
end;

procedure gcd;
var f, g;
begin
    f := x;
    g := y;
    while f <> g do
    begin
        if f < g then g := g - f;
        if g < f then f := f - g;
    end
end;
begin
    x := m; y := n; call multiply;
    x := 34; y := 36; call gcd;
end.
```

1.2 生成的代码（片段）

前面我们给出了 PL/0 语言写的一段程序，其中乘法过程经过编译程序产生以下代码：

```
 2  INT     0   5   --  allocate storage
 3  LOD 1   3   --  x
 4  STO     0   3   --  a          } a := x
 5  LOD 1   4   --  y
 6  STO     0   4   --  b          } b := y
 7  LIT     0   0   --  0
 8  STO     1   5   --  z          } z := 0
 9  LOD 0   4   --  b
10  LIT     0   0   --  0          } b > 0
11  OPR     0   12  --  >
12  JPC     0   29  --  if b <= 0 then goto 29
13  LOD 0   4   --  b
14  OPR     0   6   --  odd        } odd(b)
15  JPC     0   20  --  if not(odd(b)) goto 20
16  LOD 1   5   --  z
17  LOD 0   3   --  a
18  OPR     0   2   --  +          } z := z + a
19  STO     1   5   --  z
20  LIT     0   2   --  2
21  LOD 0   3   --  a
22  OPR     0   4   --  *          } a := 2 * a
23  STO     0   3   --  a
24  LOD 0   4   --  b
25  LIT     0   2   --  2
26  OPR     0   2   --  /          } b := b / 2
27  STO     0   4   --  b
28  JMP     0   9   --  goto 9
29  OPR     0   0   --  return
```

（if 包含 odd(b) 至 z := z + a；while 包含 b > 0 至 goto 9）

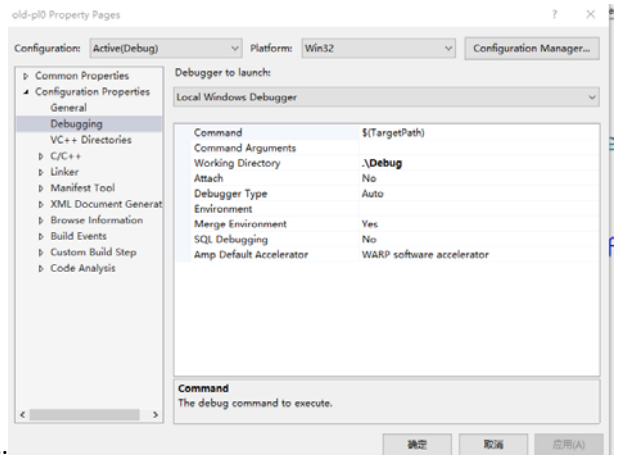
上述代码采用助记符形式，“--”后面是为了便于理解而额外加上的注释，大括号右边为左部代码序列对应的源程序中的语句或表达式。

2. PL/0 语言编译器源程序

PL/0 语言编译器源程序包括如下 C 程序文件，PL0.h、PL0.c、set.h 和 set.c。（略）

3. PL/0 程序 VS/linux 编译环境设置

- 在 Visual Studio 2013 界面上，点击 PROJECT 菜单下最后一项 XXX Properties…

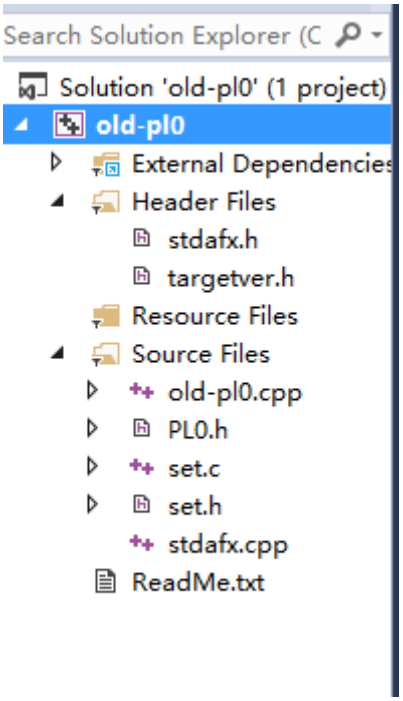


设置 Debugging 下的 Working Directory 为.\Debug
将你的 pl0 测试例子拷贝到你的 project 目录下的 debug 中，那里同时也将存放生成的 pl0 执行文件（调试模式下）。这样，你即可以在 VC 中调试并读入 PL0 测试例子，也可以在命令行下执行所生成的 pl0 编译器并读入测试例子。

- 在 pl0 源程序开头处添加：

#pragma warning(disable:4996)
增强 VC 编译兼容性。

- 将 PL0.h/set.c/set.h 等加入 VC 工程，如下图所示。



- 在 linux 等环境下，将所有程序拷贝到同一个目录下，再用 gcc 编译：
  gcc -o plcc pl0.c
  运行： ./plcc
  此外，还有一点很重要的，将你准备的 pl0 测试程序，最好转换为 unix 下文本格式。