


大题套路

 by kingno

四、综合题

有以下 PL/0 程序作为编译输入文件

- 1) 在编译过程中为什么要建立符号表?
- 2) PL/0 语言编译器中的符号表的数据结构是怎样的?
- 3) 假设有如下 PL/0 源程序, 编译器分析到 `i:=a` 这行时, 符号表中的信息是怎样的?
- 4) 请写出生成的 PCode 代码
- 5) 画出代码程序执行完 `if x<(a+1) then b:=x` 指令后, 数据栈的布局示意图

```
const m=5;
var a,b;
procedure g;
  procedure h;
    var i,x;
    begin
      i:=a;
      x:=(m+i)/(m-i);
      if x<(a+1) then b:=x;
    end
  begin
    a:=a+1;
    call h;
  end;
begin
  a:=2;
  call g;
end.
```

1. 分层

拿到源程序后，先对其进行分层

<code>const m=5;</code>	0
<code>var a,b;</code>	
<code>procedure g;</code>	
<code> procedure h;</code>	1
<code> var i,x;</code>	
<code> begin</code>	
<code> i:=a;</code>	2
<code> x:=(m+i)/(m-i);</code>	
<code> if x<(a+1) then b:=x;</code>	
<code> end</code>	
<code> begin</code>	
<code> a:=a+1;</code>	1
<code> call h;</code>	
<code> end;</code>	
<code>begin</code>	
<code> a:=2;</code>	0
<code> call g;</code>	
<code>end.</code>	

- `procedure` 之后的一行到 `end` 之间算新的一层
- `begin` 到 `end` 之间不算新的一层
- 这个分层的数字（0、1、2），用于表示变量、常量、过程名的level，在之后会用到

2. 填符号表

name	kind	value/level	address	size
m	const	5		
a	var	0	3	
b	var	0	4	
g	pro	0		3
h	pro	1		5
i	var	2	3	
x	var	2	4	

- name: 填写变量、常量、过程名
- kind: 变量填 **var**，常量填 **const**，过程名填 **pro**
- value/level: 变量填level，常量填value（也就是常量的值），过程名填level。level是第一步中对应的层，比如变量 **a** 在第 0 层，就填 0
- address

- 常量: 不用填
- 变量: 变量的address是指**相对地址**。假设这个变量是**该层**的第 **i** 个变量，则它的地址为 **2+i**。比如，变量 **a** 是第 0 层的第一个变量，它的 address 就是 3

为什么不从0、1、2开始？因为变量是存在栈上的，0、1、2已经被静态链（SL）、动态链（DL）、返回地址（RA）占据了

- 过程名: 先空着，等写完 PCode 再回填
- size: 只有过程需要填写size，size=3+局部变量的个数（3就是SL、DL、RA）

（虽然我们可以肉眼观察出过程的size，但计算机事先是不知道的，它需要在运行时确定。这对应着第3小题）

3. 写 PCode

如何直接对着 PL/0 源程序直接翻译写出 PCode? 套路:

① 自上而下对被说明的变量名、过程名及程序段按嵌套深度分级,并写出变量的位移地址。

② 过程开始前,写 1 个“`jmp 0,0`”指令,若连续嵌套说明几个过程,则再写几个“`jmp 0,0`”指令。

③ 每个过程的开始,用一条“`int 0,m`”指令($m = 3 + \text{局部变量数}$)为过程和过程变量预留存储单元。

④ 一个过程结束,用一条“`opr 0,0`”指令返回调用程序。

⑤ 遇常量(不论常数还是常量标识符)用一条“`lit 0,val`”指令($val = \text{常量值}$)。

⑥ 遇变量用一条“`lod lev-level,adr`”指令,其中,level 是变量说明的级;lev 是语句所在程序段的级;adr 是变量的位移地址。

⑦ 表达式、关系式、以及各种语句的翻译规则见前面各节,不再重述。

现在仍以例 11-9 的 PL/0 程序为例,说明怎样按上述算法规则直接翻译写出生成的代码程序。

具体过程

1. 先自上而下对被说明的变量名、过程名及程序段按嵌套深度分级,并写出变量的位移地址

这就是第一步 分层

2. 过程开始前,写 1 个 `jmp 0,0` 指令,若连续嵌套说明几个过程,则再写几个 `jmp 0,0` 指令

对于这段源程序,有主函数、g、h三个过程,所以写三个

```
0  jmp 0,0
1  jmp 0,0
2  jmp 0,0
```

3. 每个过程的开始,用一条 `int 0,m` 指令 ($m=3+\text{局部变量数}$) 为过程和过程变量预留存储单元

我们首先翻译 `h` (为什么? 我也不知道), `h` 有2个局部变量 `i,x`, 所以填入 `int 0,5`

```
0  jmp 0,0
1  jmp 0,0
2  jmp 0,0
3  int 0,5
```

4. 一个过程结束,用一条 `opr 0,0` 指令返回调用程序

套路,没什么好说的

5. 遇到常量（不论常数还是常量）用一条 `lit 0,val` 指令

比如，过程 `h` 中会使用到常量 `m`，我们就在用到 `m` 的位置插入 `lit 0,5`

6. 遇到变量用一条 `lod lev-level, adr` 指令。其中，`level` 是符号表中的 `level`，`lev` 是当前语句所在的 `level`，`adr` 是符号表中的 `address`

比如，过程 `h` 中有一句 `i:=a`，`a` 是第0层的，所以翻译过来就是 `lod 2-0,3`

7. 各种语句（if、while 等）的翻译看书

完整的 PCode 如下

```
0    jmp 0,30    # 套路，30是全部指令写完后确定的
1    jmp 0,23    # 套路，23是全部指令写完后确定的
2    jmp 0,3     # 套路，3 是全部指令写完后确定的
          -----过程h开始-----
3    int 0,5     # 预留 3+2 个空间
4    lod 2,3     # 把 a 存到栈顶
5    sto 0,3     # 把栈顶 a 的值赋给 i
6    lit 0,5     # 把 m 存到栈顶
7    lod 0,3     # 把 i 存到栈顶
8    opr 0,2     # 把 m+i 存到 m 的位置，并让栈顶指针 t 指向 m+i
9    lit 0,5     # 把 m 存到栈顶(覆盖了第7条指令的i)
10   lod 0,3     # 把 i 存到栈顶
11   opr 0,3     # 把 m-i 存到 m 的位置，并让栈顶指针 t 指向 m-i
12   opr 0,5     # 把 (m+i)/(m-i) 存到 (m+i) 的位置，你知道此时 t 指向哪里吗？
13   sto 0,4     # 把栈顶 (m+i)/(m-i) 存到 x 中
14   lod 0,4     # 把 x 的值存到栈顶
15   lod 2,3     # 把 a 的值存到栈顶
16   lit 0,1     # 把 1 存到栈顶
17   opr 0,2     # 把 a+1 存到 a 的位置
18   opr 0,10    # 把 x<(a+1) 的比较结果存到栈顶
19   jmc 0,22    # 不满足条件分支，结束
20   lod 0,4     # 把 x 的值存到栈顶
21   sto 2,4     # 把栈顶 x 的值存到 b 中
22   opr 0,0     # 套路
          -----过程h结束-----
23   int 0,3     # 预留 3+0 个空间
24   lod 1,3
25   lit 0,1
26   opr 0,2
          -----过程g结束-----
27   sto 1,3
28   cal 0,3     # 0是g的静态链，3是h第一条指令的地址
29   opr 0,0
30   int 0,5
31   lit 0,2
32   sto 0,3
33   cal 0,23    # 主函数的静态链是0，23是g的第一条指令的地址
34   opr 0,0
          -----主过程结束-----
```

写完 PCode 之后，记得回去填写符号表

4. 画数据栈

在画数据栈之前，你需要理解 `opr 0,a` 指令执行时，栈顶指针 `t` 是怎么移动的。见书本 P137

- 对于每个过程，SL、DL、RA 都是必须画的
- 栈中的数据会在运行时变化，比如 `lit`，`lod` 等指令都会引起数据的变化