

# 编译第五次作业答案

2024 年 12 月 30 日

## 1 PL0 编译器运行时及中间代码生成练习（见 pl0\_ex1.pdf）

### (a) LODA/STOA/LEA 指令的语义代码

```
1 // (LODA, 0, 0): 以当前栈顶单元的内容为“地址偏移”来读取相应单元的值，并
   将该值存储到原先的栈顶单元中
2 case LODA:
3     stack[top]=stack[stack[top]];
4     break;
5
6 // (STOA, 0, 0): 将位于栈顶单元的内容，存入到次栈顶单元内容所代表的栈单元
   里，然后弹出栈顶和次栈顶。
7 case STOA:
8     stack[stack[top-1]]=stack[top];
9     // printf可省略
10    top=top-2;
11    break;
12
13 // (LEA, l, a): 获取名字变量在“运行时栈-stack”上“地址偏移”。
14 case LEA:
15     stack[++top]=base(stack, b, i.l)+i.a;
16     break;
```

### (b)

#### 1. a 和 p 的类型表达式

`int* a[10]`: a 是一个大小为 10，元素为指向 `int` 类型的指针的数组。类型表达式为 `array(10,pointer(int))`

`int* ((*p)[10])[10]`: p 是一个指向包含 10 个元素的数组的指针，其中每个元素又是一个指向包含 10 个指向 `int` 类型的指针的数组的指针。类型表达式为

`pointer(array(10,pointer(array(10,pointer(int))))))`

1. `(*p)[10]`: p 是一个指向包含 10 个元素 1 的数组的指针，数组中的每个元素 1 是一个指向元素 2 的指针。

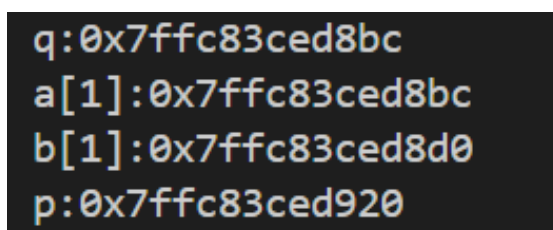
2. `int* ((*p)[10])[10]`: 每个元素 2 是一个指向包含 10 个 `int*` 的数组的指针。

**2. 根据 PL0 编译环境设定，上述程序中分配的总变量空间是多少？各个变量在活动记录中“地址偏移”是多少？**

假设 int 类型占用 4 字节，指针类型占用 4 字节  
int i: 4 字节  
int\* q: 4 字节  
int\* a[10]: 4\*10=40 字节  
int\* (\*b[10])[10]: 数组 b 包含 10 个指针，每个指针指向一个包含十个元素的数组。b 的空间只分配指针的空间，因此占用空间为 40 字节  
int\* (\*(p)[10])[10]: p 是一个指向包含 10 个元素的数组的指针，因此占用空间 4 字节。  
总变量空间为 4+4+40+40+4=92 字节  
变量 i 地址偏移为 0，q 为 4，a 为 8，b 为 48，p 为 88，单位为字节。

**3. 两条输出语句中不同的表达式各自仅包含唯一的名字变量 p。根据你的 C 语言知识，补全这两处输出语句中的源代码。**

`*(p[0][1])[1]` 和 `*(p[1])[1]`  
p[0] 表示对 p 解引用并访问第一个元素（即 (\*p)[0]），这个元素是一个指向包含 10 个 int\* 元素的数组的指针。  
\*p 是对 p 的解引用，得到 p 指向的数组，这个数组包含 10 个元素，每个元素都是指向数组的指针，而这个数组又包含 10 个 int\* 指针。  
a[1] = q: 将 a[1] 指向 i（即 q = &i）。  
b[1] = &a: 将 b[1] 指向 a（即 b[1] 是指向 a 的指针）。  
p = &b: 将 p 指向 b（即 p 是指向 b 的指针）。  
p[0] 是 (\*p)[0]，即 b[0]，它是指向 a 的指针。  
(\*p[0][1]) 是解引用 b[0][1]，得到 a[1]，也就是 q，指向 i。  
(\*p[0][1])[1] 是解引用 q，然后访问 i 的第 1 个元素。  
\*(p[1]) 是 b[1]，即指向 a 的指针。  
(\*(p[1])) 是解引用 b[1]，得到 a。  
(\*(p[1])[1]) 是访问 a[1]，即 q，然后解引用 q，得到 i 的值。



```
q:0x7ffc83ced8bc
a[1]:0x7ffc83ced8bc
b[1]:0x7ffc83ced8d0
p:0x7ffc83ced920
```

图 1: 中间变量的值 (不同机器结果可能不同)

**4. 给出一个上述下划线处源代码对应的 PL0 代码**

```
1    // i = 100
2    LIT 0 100
```

```

3      STO 0 0      // 将 100 存入 i
4
5      // a[1] = q
6      LIT 0 0      // 加载 i 的地址
7      ADD 0 0      // 获取 i 的地址
8      STO 0 1      // 将 i 的地址赋给 q
9
10     LIT 0 1      // 获取指针 q
11     STO 0 2      // 将 q 存入 a[1]
12
13     // p = &b
14     LIT 0 2      // 获取 a 的地址
15     STO 0 3      // 将 a 的地址存入 b[1]
16
17     LIT 0 3      // 获取 b 的地址
18     STO 0 4      // 将 b 的地址存入 p
19
20     // 对应表达式 *(*p[0][1])[1]
21     LIT 0 4      // p 的地址为 4, p[0] 即 b[0] (地址为 4)
22     LOD 0 4      // 获取 p[0], 即 b[0] 地址
23     LOD 0 1      // 获取 p[0][1], 即 a[1] (地址为 1)
24     LOD 0 2      // 解引用 a[1], 即 q 的值 (地址为 q)
25     LOD 0 3      // 访问 q[1], 即 q 的第二个元素
26
27     // 对应表达式 *(*(*p)[1])[1]
28     LIT 0 4      // p 的地址为 4, (*p) 即 b[0] (地址为 4)
29     LOD 0 4      // 获取 (*p), 即 b[0] 地址
30     LOD 0 1      // 获取 (*p)[1], 即 a[1] (地址为 1)
31     LOD 0 2      // 解引用 a[1], 即 q 的值 (地址为 q)
32     LOD 0 3      // 访问 q[1], 即 q 的第二个元素

```

**(c) 对于函数调用：func(r, r, \*r) 分别给出计算三个实参的“值”到 stack 栈顶的 PL0 代码。假设 r 的地址偏移为 3。**

引用变量 r 实际上存储的是一个指针的地址，而这个指针指向一个 int 类型的值。

int func(int \*i, int\* &j, int k)

r(int \*i): 指针地址

LOD 0, 3; 加载 r 的地址偏移为 3 的值到栈顶

r(int\* &j): 对指针的引用，传入指针地址即可

LOD 0, 3; 加载 r 的地址偏移为 3 的值到栈顶

\*r(int k): 指针地址保存的值，需要两次解引用

LOD 0, 3; 加载 r 的地址偏移为 3 的值 (r 的地址) 到栈顶

LOD 0, 0; 间接加载 r 所指向地址的值到栈顶

## 2 C runtime 练习 (见 runtime.pdf)

### 第一题

#### (a) 输出

输出: 36313032 2016

char c[5] 和 int i 共用同一段内存 (最小大小为 sizeof(int), 假设为 4 字节)。在小端模式下 (i386 默认是小端模式), 低地址存储低字节, 高地址存储高字节。

data.c[0] = '2'; → 字符'2' 的 ASCII 值是 0x32, 存入最低地址。

data.c[1] = '0'; → ASCII 值是 0x30。

data.c[2] = '1'; → ASCII 值是 0x31。

data.c[3] = '6'; → ASCII 值是 0x36。

data.c[4] = '\0'; → 字符串终止符 0x00。

联合体的 int i 被覆盖, 存储为 0x36313032 (小端存储, 顺序为 0x32, 0x30, 0x31, 0x36)

#### (b) 补全代码

```
1      .section .rodata
2  .LC0:
3      .string "%x,%s\n"
4      .text
5  .globl main
6      .type main,@function
7  main:
8      pushl %ebp
9      movl %esp, %ebp
10     subl $40, %esp
11     andl $-16, %esp // 按16字节对齐(空间不变或增加), 方便一些指令并行
                        操作
12     movl $0, %eax
13     subl %eax, %esp
14     movb $50, -24(%ebp)
15     movb $48, -23(%ebp)
16     movb $49, -22(%ebp)
17     movb $54, -21(%ebp)
18     movb $0, -20(%ebp)
19     leal -24(%ebp), %eax
20     movl %eax, -28(%ebp)
21     sub $4, %esp // 由addl $16, %esp可知, 有操作使得%esp减少16, 即有数
                        据入栈。printf的两个参数和返回地址入栈会占用12字节空间, 因此这
                        里还需要占用4字节空间, 可以通过%esp减四实现。
22     pushl -28(%ebp) // 参数入栈
23     pushl -24(%ebp) // 参数入栈
24     pushl $.LC0 // 返回地址入栈
25     call printf
26     addl $16, %esp // 恢复栈之前状态, 清除掉printf函数参数的空间
27     movl $0, %eax // return 0;
```

```

28     leave
29     ret

```

## 第二题

### (a) 补全代码

```

1 //当 N=2 时, 生成的汇编代码片段
2     .file "test1.c"
3     .text
4     .globl f
5     .type f,@function
6 f:
7     pushl %ebp
8     movl %esp, %ebp
9     movl $100, 8(%ebp)
10    movl $16, 12(%ebp)
11    movb $65, 17(%ebp) // z[0]:16(%ebp) z[1]:17(%ebp)
12    movl 20(%ebp), %eax // 字节对齐, z[2]补到了4个字节
13    pushl 12(%eax) // 逆序入栈
14    pushl 8(%eax)
15    pushl 4(%eax)
16    pushl (%eax)
17    call f
18    addl $16, %esp
19    leave
20    ret

```

```

1 //当 N=11 时, 生成的汇编代码片段
2     .file "test1.c"
3     .text
4     .globl f
5     .type f,@function
6 f:
7     pushl %ebp
8     movl %esp, %ebp
9     pushl %edi
10    pushl %esi
11    movl $100, 8(%ebp)
12    movl $24, 12(%ebp) // N=11时大小为4+4+11+1+4=24
13    movb $65, 17(%ebp) // z[1]的地址偏移不变
14    subl $8, %esp
15    movl 28(%ebp), %eax //字节对齐
16    subl $24, %esp
17    movl %esp, %edi
18    movl %eax, %esi
19    cld
20    movl $6, %eax // 解释见下
21    movl %eax, %ecx

```

```

22     rep
23     movsl
24     call f
25     addl $32, %esp
26     leal -8(%ebp), %esp
27     popl %esi
28     popl %edi
29     leave
30     ret
31 // rep movsl 为数据传送指令，即，由源地址 esi 开始的 ecx 个字的数据传
    送到由 edi 指示的目的地址。

```

movl \$6, %eax: 在汇编中，函数 f 需要将 p.next 所指向的结构体内容复制到新的栈空间中，复制的字节数为 sizeof(DOT) 的大小。由于 rep movsl 是按字（4 字节）为单位复制的，因此需要将总字节数除以 4，得出需要复制的 long 数量。

$$long\_num = \frac{sizeof(DOT)}{long} = \frac{24}{4} = 6$$

(b) 从运行时环境看，addl \$16, %esp 和 leal -8(%ebp), %esp 这两条汇编指令的作用是什么？

清除函数 f 占用的空间

(c) 结合上述两种汇编代码，简述编译器在按值传递结构变量时的处理方式。

1. 逆序的栈传递方式
2. 数据多时采用数据传送指令。

### 第三题

(a) 补全代码

```

1 //第三题函数 g 的汇编代码片段
2 .globl g
3     .type g,@function
4 g:
5     pushl %ebp
6     movl %esp, %ebp
7     movl 8(%ebp), %eax // %eax保存参数位置，即**p的地址
8     movl (%eax), %eax // 第一次解引用，%eax保存*p的地址
9     addl $1, (%eax) // 第二次解引用，%eax保存p的地址，(%eax)保存p的
    值
10    movl 8(%ebp), %eax
11    addl $4, (%eax) // **p为int类型，数据类型大小为四个字节。因此(*p
    )++实际上会增加一个int类型指针的大小，即四个字节
12    leave
13    ret

```

```

1      .file "p.c"
2      .text
3      .globl main
4      .type main,@function
5  main:
6      pushl %ebp
7      movl %esp, %ebp
8      subl $72, %esp
9      andl $-16, %esp
10     movl $0, %eax
11     subl %eax, %esp
12     leal -56(%ebp), %eax
13     movl %eax, -64(%ebp)
14     movl $0, -60(%ebp)
15  .L2:
16     cmpl -60(%ebp), $9
17     // 通过下面两条跳转指令可知, 小于等于的时候跳转L5, 否则跳转L3. 因此
        这里应该是与9比较大小。再通过L5的leal和incl两条指令可知, i保存
        在%ebp-60中
18     jle .L5
19     jmp .L3
20  .L5:
21     movl -64(%ebp), %edx
22     movl -60(%ebp), %eax
23     movl %eax, (%edx)
24     subl $12, %esp
25     leal -64(%ebp), %eax
26     pushl %eax
27     call g
28     addl $16, %esp // subl $12, %esp和pushl %eax使得%esp减少了十六, 这
        里加回来清除占用的空间
29     leal -60(%ebp), %eax // %eax保存地址%ebp-60
30     incl (%eax) // %eax保存地址对应的值加一, 实际上是i++
31     jmp .L2 // 跳转回L2判断循环是否结束
32  .L3:
33     movl $0, %eax
34     leave
35     ret
36  //第三题函数 main 的汇编代码片段

```

(b) main 函数中 for 循环结束时, 数组 line 各元素值是多少?

1 2 3 4 5 6 7 8 9 10

#### 第四题

(a) 补全代码

```

1  main:
2      pushl %ebp
3      movl %esp, %ebp
4      subl $24, %esp
5      andl $-16, %esp
6      movl $0, %eax
7      subl %eax, %esp
8      movl $0, -20(%ebp)
9      movl $0, -16(%ebp)
10     movl $1, -12(%ebp)
11     movl $2, -12(%ebp)
12     movl $3, -8(%ebp)
13     movl $0, %eax
14     leave
15     ret

```

### (b) 描述所用编译器对 C 分程序所声明变量的存储分配策略

1. 分配在栈上，地址由根据局部变量的先后顺序由低到高
2. 退出作用域的变量空间会被重用

## 第五题

### (a) 指出波浪线处的汇编代码的作用

这段代码使用 movsl (Move String Long) 指令与 rep (Repeat) 前缀复制数据。具体来说，它将.LC0 标签处的数组（即初始化的数组 a）复制到栈上。

leal -40(%ebp), %edi 设置目的地址（栈上的数组 a 的位置），movl \$.LC0, %esi 设置源地址（原始数组），movl \$6, %ecx 和 movl %eax, %ecx 设置复制的元素数量。

### (b) 补全代码

```

1  .LC0:
2      .long 0
3      .long 1
4      .long 2
5      .long 3
6      .long 4
7      .long 5
8  .LC1:
9      .string "%d\n"
10     .text
11     .globl main
12     .type main, @function
13 main:
14     pushl %ebp
15     movl %esp, %ebp
16     pushl %edi

```



```

17     pushl %esi
18     subl $48, %esp
19     andl $-16, %esp
20     movl $0, %eax
21     subl %eax, %esp
22     leal -40(%ebp), %edi
23     movl $.LC0, %esi
24     cld
25     movl $6, %eax
26     movl %eax, %ecx
27     rep
28     movsl
29     movl $6, -44(%ebp) // i=6
30     movl $7, -48(%ebp) // j=7
31     leal -40(%ebp), %eax
32     addl $24, %eax
33     // 计算@a+1, 注意@a是数组a的首地址, 类型是int(*)[6], 即指向包含6个
        int的数组的指针。@a+1的意思是跳过整个数组a, 它指向a之后的地
        址。
34     movl %eax, -52(%ebp)
35     subl $8, %esp // esp减少8
36     movl -52(%ebp), %eax
37     subl $4, %eax // p为int*类型变量, p-1中的1是一个int*, 因此大小为四
        字节
38     pushl (%eax) //esp减少4
39     pushl $.LC1 //esp减少4
40     call printf
41     addl $16, %esp // 上面共减少8+4+4=16
42     movl $0, %eax
43     leal -8(%ebp), %esp // 下面两条指令分别将esi和edi从栈中pop出, 注意
        到main最开始将edi和esi入栈了。为了恢复esi, edi, 需要更新栈指针
44     popl %esi
45     popl %edi
46     leave
47     ret

```

## 第六题

### (a) 运行结果

10

### (b) 相关图示

假设在运行栈中, 每个活动记录 (AR) 包含以下信息:

1. 返回地址
2. 静态链接 (指向封闭作用域的 AR)
3. 参数

#### 4. 局部变量

当 f(17, dummy) 最深嵌套调用时，栈将包含以下活动记录：

```
|-----|
| AR for f(10, f@level=11) |
| Return Addr      |
| Static Link -> f@level=11 |
| level = 10      |
| arg = f@level=11 |
|-----|
| AR for f(11, f@level=12) |
| Return Addr      |
| Static Link -> f@level=12 |
| level = 11      |
| arg = f@level=12 |
|-----|
|      ...      |
|-----|
| AR for f(17, dummy) |
| Return Addr      |
| Static Link -> staticLink |
| level = 17      |
| arg = dummy      |
|-----|
| AR for staticLink |
| Return Addr      |
| Static Link      |
|-----|
```

图 2: 图示

### 3 Compiler-Explorer-FPC-pascal-Code-demo-static-link.pas

(1)

```
1 procedure grandpa(); // 嵌套深度: 0
2   var g : integer; // 嵌套深度: 1
3   procedure uncle(var u,v :integer;w:integer); // 嵌套深度: 1
4
5   begin
6     g := 10; // 嵌套深度: 2
7   end;
8
```

```

9      procedure father(p_f : integer); // 嵌套深度: 1
10         var f : integer; // 嵌套深度: 2
11
12         procedure son(p_s,p_t : integer); // 嵌套深度: 2
13            var s : integer; // 嵌套深度: 3
14            begin
15               s := f; // 嵌套深度: 3
16               f := g; // 嵌套深度: 3
17               g := s; // 嵌套深度: 3
18               uncle(s,f,g); // 嵌套深度: 3
19            end;
20         begin
21            son(20,30); // 嵌套深度: 2
22         end;
23 begin
24     father(40); // 嵌套深度: 1
25 end;
26
27 end.

```

名字	类别	定义点嵌套深度 Def	引用点嵌套深度 Ref	层次差 =Ref-Def
grandpa	函数	0	0	0
g	变量	1	2/3	1/2
uncle	函数	1	3	2
u,v,w	变量	2	2	0
father	函数	1	1	0
p_f	变量	2	2	0
f	变量	2	3	1
son	函数	2	2	0
p_s,p_t	变量	3	3	0
s	变量	3	3	0

(2)

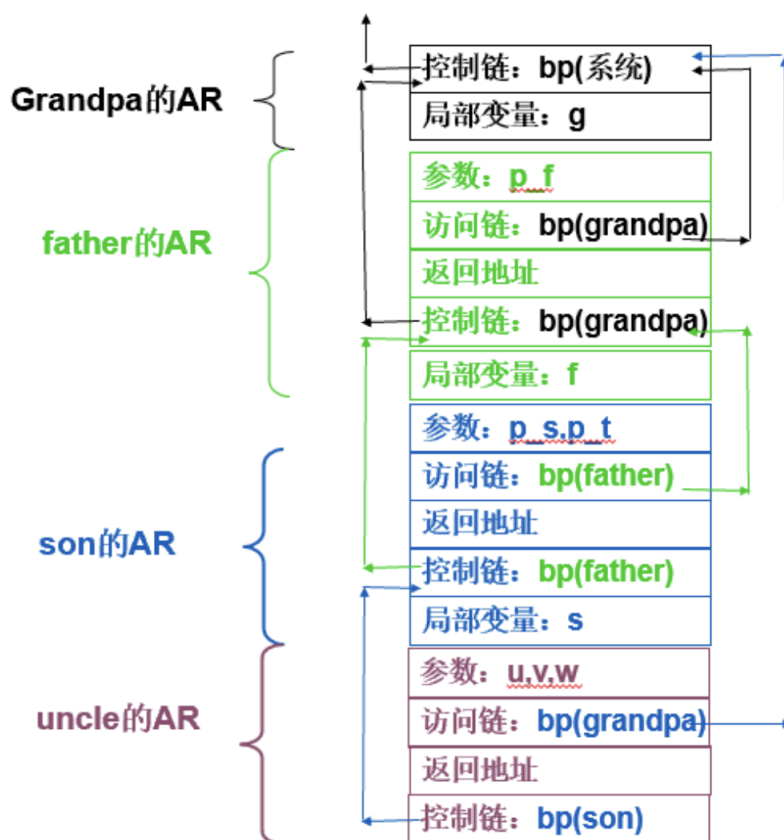


图 3: Enter Caption

(3)

```

1 son(smallint, smallint):
2  建立过程框架:
3      pushq    %rbp
4      movq     %rsp, %rbp
5      leaq     -32(%rsp), %rsp
6  这段代码用于建立son过程的栈帧。首先保存调用者的基指针%rbp，然后将当前的
   栈指针%rsp设置为新的基指针。
7  传递参数和局部变量的初始化:
8      movq     %rdi, -24(%rbp)
9      movw     %si, -8(%rbp)
10     movw     %dx, -16(%rbp)
11 这部分代码将参数从寄存器传递到栈上，以便在过程内部使用。
12 局部变量赋值和处理:
13     movq     -24(%rbp), %rax
14     movw     -20(%rax), %ax
15     movw     %ax, -28(%rbp)
16     movq     -24(%rbp), %rdx
17     movq     -24(%rbp), %rax
18     movq     -16(%rax), %rax
19     movw     -4(%rax), %ax

```

```

20     movw    %ax,-20(%rdx)
21     movq    -24(%rbp),%rdx
22     movq    -16(%rdx),%rdx
23     movw    -28(%rbp),%ax
24     movw    %ax,-4(%rdx)
25 这段代码进行局部变量s、f和g之间的值交换，并准备调用uncle过程。
26 调用uncle过程：
27     movq    -24(%rbp),%rax
28     movq    -16(%rax),%rax
29     movswl  -4(%rax),%ecx
30     movq    -24(%rbp),%rax
31     leaq    -20(%rax),%rdx
32     leaq    -28(%rbp),%rsi
33     movq    -24(%rbp),%rdi
34     movq    -16(%rdi),%rdi
35     call    uncle(smallint,smallint,smallint)
36 这部分代码设置了调用uncle过程所需的参数，并执行调用。
37 结束过程：
38     movq    %rbp,%rsp
39     popq    %rbp
40     ret
41 最后，恢复基指针%rbp，释放栈帧，并返回到调用者。

```

## 4 Compiler Explorer Pascal Editor #1 Code (4).pas

(1)

层数	函数名	返回地址	静态链	动态链	参数	局部变量
5	s	调用 s 后的下一条指令	r	q	17	z=17
4	q	调用 q 后的下一条指令	main	p	s	
3	p	调用 p 后的下一条指令	main	r	q,s	
2	r	调用 r 后的下一条指令	main	main		i=10
1	main					

表 1: 函数调用信息

(2)

```

1  \\传递参数s:
2     movq    $p$()$_$.s(longint),%rax
3     movq    %rax,-24(%rbp)
4     movq    %rbp,-16(%rbp)
5     movq    -24(%rbp),%rdx
6     movq    -16(%rbp),%rcx
7  \\传递参数q:

```

```
8      movq    $q(),%rax
9      movq    %rax,-24(%rbp)
10     movq    $0,-16(%rbp)
11     movq    -24(%rbp),%rdi
12     movq    -16(%rbp),%rsi
13     call    p(crcd993699d)
```

**(3)**

27