H10 中间/汇编代码生成

7.9 下面的C语言程序

```
main() {
    int i,j;
    while ( (i || j) && (j > 5) ) {
        i = j;
    }
}
```

在x86/Linux系统上编译生成的汇编代码如下（编译器版本见汇编代码最后一行）：

```
        .file   "ex7-9.c"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        pushq   %rbp
        movq    %rsp, %rbp
        jmp     .L2
.L5:
        movl    -4(%rbp), %eax
        movl    %eax, -8(%rbp)
.L2:
        cmpl    $0, -8(%rbp)
        jne     .L3
        cmpl    $0, -4(%rbp)
        je      .L4
.L3:
        cmpl    $5, -4(%rbp)
        jg      .L5
.L4:
        movl    $0, %eax
        popq    %rbp
        ret
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~16.04) 7.5.0"
```

在该汇编代码中有关的指令后加注释，将源程序中的操作和生成的汇编代码对应起来，以判断确实是用短路计算方式来完成布尔表达式计算的。

```
.LFB0:
        pushq   %rbp
        movq    %rsp, %rbp
        jmp     .L2
.L5:
        movl    -4(%rbp), %eax
        movl    %eax, -8(%rbp)
.L2:
        cmpl    $0, -8(%rbp)     比较0和保存到-8(%rbp)的值的大小，即比较0和i
```

```
        jne     .L3             若二者不相等（即i!=0，（i||j）为true），则跳转到L3
        cmpl    $0, -4(%rbp)    比较0和保存到-4(%rbp)的值的大小，即比较0和j
        je      .L4             若二者相等，此时i、j均为0，则跳转到L4
.L3:
        cmpl    $5, -4(%rbp)    比较5和保存到-4(%rbp)的值的大小
        jg      .L5             若-4(%rbp)大于5的值（即j>5），则跳转到L5
.L4:
        movl    $0, %eax
        popq    %rbp
        ret
```

7.10 下面是一个C语言程序和在x86/Linux系统上编译（版本较低的GCC编译器，并且未使用优化）该程序得到的汇编代码（为便于理解，略去了和讨论本问题无关的部分，并改动了一个地方）。

```
main() {
    long i,j;
    if ( j )
        i++;
    else
        while ( i ) j++;
}
```

编译产生的汇编代码如下：

```
main:
    pushl %ebp
    movl %esp,%ebp
    subl $8,%esp
    cmpl $0,-8(%ebp)
    je .L2
    incl -4(%ebp)
    jmp .L3
.L2:
.L4:
    cmpl $0,-4(%ebp)
    jne .L6
    jmp .L5
.L6:
    incl -8(%ebp)
    jmp .L4
.L5:
.L3:
.L1:
    leave
    ret
```

(a) 为什么会出现一条指令前有多个标号的情况，如.L2和.L4，还有.L5、.L3和.L1？从控制流语句的中间代码结构加以解释。

条件和循环语句的中间代码如下：

```
if Expr1 then stmt1 else stmt2
Expr1的代码
为假转至L2
stmt1的代码
L2：stmt2的代码
L3：while Expr2 do stmt
L4:Expr2的代码
    为真转至L6
    转至L5
L6:stmt的代码;
   转至L4
L5:
```

在此程序中，while语句即为if语句中的stmt2，将这两个合并就出现了多个标号的情况

(b) 每个函数都有这样的标号.L1，它可能的作用是什么，为什么本函数没有引用该标号的地方？

.L1标号定义的是返回调用者所需要执行的指令，而本函数内没有return语句，故不需要返回给调用者。

(c) 如果用较新的gcc版本（如gcc7.5.0）进行编译，产生的汇编代码如下。请说明L3~L5的含义，为什么没有L1和L2标号，分析可能的原因。

```
        .file   "ex7-10.c"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        pushq   %rbp
        movq    %rsp, %rbp
        cmpq    $0, -16(%rbp)
        je      .L4
        addq    $1, -8(%rbp)
        jmp     .L3
.L5:
        addq    $1, -16(%rbp)
.L4:
        cmpq    $0, -8(%rbp)
        jne     .L5
.L3:
        movl    $0, %eax
        popq    %rbp
        ret
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~16.04) 7.5.0"
```

L3~L5的含义：

```
.L5:
        addq    $1, -16(%rbp)    将1加到-16(%rbp)，即执行 i=i+1
.L4:
        cmpq    $0, -8(%rbp)     比较0和保存到-8(%rbp)的值的大小
        jne     .L5              若不相等，则跳转至L5
.L3:
        movl    $0, %eax         把返回值保存至寄存器eax
        popq    %rbp             将main函数的栈帧地址恢复到%rbp
        ret                      返回
```

用较新版本的 gcc 编译，若没有必要，编译器不会产生一条指令前有多个标号的情况。

7.17 C语言和Java语言的数组声明和数组元素引用的语法形式同7.3.节讨论的不一样，例如float A[10][20]和A[i+1][j-1]，并且每一维的下界都是0。若适应这种情况的赋值语句的文法如下：

```
S --> L := E
E --> E + E | (E ) | L
L --> L [E ] | id
```

（a）重新设计7.3.2节数组元素的地址计算公式，以方便编译器产生数组元素地址计算的中间代码。不要忘记每一维的下界都是0。

A[i_1 , i_2 , … , i_k]的地址表达式为 base + $i_1 \times w_1 + i_2 \times w_2 + \ldots + i_k \times w_k$

其中 w_j 指 A [i_1] [i_2] … [i_j] 的宽度

（b）重新设计数组元素地址计算的翻译方案。只需写出产生式 `L --> L [E ] | id` 的翻译方案，但要能和7.3.3节中产生式 `S --> L := E` 和 `E --> E + E | (E ) | L` 的翻译方案衔接。若翻译方案中引入新的函数调用，要解释这些函数的含义。

```
L -> L1[E]      {   L.ndim = L1.ndim + 1;
                    L.array = L1.array;
                    w = getwidth(L.array , L.ndim);
                    if L.ndim == 1  then
                        L.place = newtemp();
                        emit( L.offset , '=' , E.place , '*' , w );
                    else
                        t = newtemp();
                        L.offset = newtemp();
                        emit( t , '=' , E.place , '*' ,w );
                        emit( L.offset , '=' , L1.offset , '+' , t )
                }
L -> id         {   L.offset = 0;
                    L.ndim = 0;
                    L.place = id.place;
                    L.array = id.place;
                }
```

其中，getwidth(L.array , L.ndim) 指从 L.array 所指条目中取第 L.ndim 维的元素所需要的字节