中 国 科 学 技 术 大 学

2007－2008 学年第二学期考试试卷（A）

考试科目：编译原理和技术　　得分：_____

学生所在系：_____　姓名：_____　学号：_____

1、（10 分）用正规式表示字母表{a, b}上 a 不会相邻的所有句子的集合，并给出接受该语言的最简 DFA。

2、（15 分）（1）为下面文法构造规范 LR(1)分析表，画出像教材上图 3.19 这样的状态转换图就可以了。

$S \to V = E \mid E$

$V \to * E \mid \mathbf{id}$

$E \to V$

（2）上述状态转换图有同心项目集吗？若有，合并同心项目集后是否会出现动作冲突？

3、（10 分）为字母表{0, 1}上的回文数集合写一个 LR 文法；若你认为该语言不存在 LR 文法，则说明理由。（注：一个数字串，从左向右读和从右向左读都一样时，称它为回文数。）

4、（10 分）下面的翻译方案计算 0 和 1 的串的值（解释为二进制的正整数）。

$B \to B_1\ 0$ { $B.val = B_1.val \times 2$ }

$B \to B_1\ 1$ { $B.val = B_1.val \times 2 + 1$ }

$B \to 1$ { $B.val = 1$ }

重写该翻译方案，使得它基础文法没有左递归，并且为整个输入串计算的 *B.val* 和原来的一样。

5、（10 分）若布尔表达式有异或 **xor**（exclusive-or）运算（异或运算的结果为真，当且仅当它的一个运算对象为真），请按照教材上表 7.4 的风格给出异或运算的语义规则。（备注：该题目设计得不好。）

6、（5 分）在面向对象语言中，编译器给每个类建立虚方法表，如教材上图 11.3 和图 11.4 那样。请简要说明，为什么编译器给每个类仅建立虚方法表，而不是建立所有方法的方法表。

7、（15 分）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教材上 7.3.3 节和 7.3.4 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：

$S \to L := E$

$E \to E + E \mid (E) \mid L$

$L \to L\ [E\ ] \mid \mathbf{id}$

（1）重新设计教材上 7.3.3 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。

（2）重新设计数组元素地址计算的翻译方案。只需写出产生式 $L \rightarrow L\,[E\,]$ | **id** 的翻译方案，但要能和 7.3.4 节中产生式 $S \rightarrow L := E$ 和 $E \rightarrow E + E \mid (E\,) \mid L$ 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。

8、（15 分）一个 C 语言的文件如下：

```
func(long i, long j, long k) {
    k = (i + j) – (i – j – f(k));
}
```

经 GCC 3.4.6 编译器编译得到的汇编代码分两列在下面给出：

```
        .file   “call.c”                |       movl    16(%ebp), %eax
        .text                           |       movl    %eax, (%esp)
.globl func                             |       call    f
        .type       func, @function     |       subl    %eax, %ebx
func:                                   |       movl    %ebx, %eax
        pushl   %ebp                    |       subl    %eax, %esi
        movl    $esp, %ebp              |       movl    %esi, %eax
        pushl   %esi                    |       movl    %eax, 16(%ebp)
        pushl   %ebx                    |       popl    %ebx
        movl    12(%ebp), %eax          |       popl    %esi
        movl    8(%ebp), %esi           |       popl    %ebp
        addl    %eax, %esi              |       ret
        movl    12(%ebp), %edx          |       .size   func, .-func
        movl    8(%ebp), %eax           |       .section    .note.GNU-stack,””,@progbits
        movl    %eax, %ebx              |       .ident  “GCC: (GNU)   3.4.6”
        subl    %edx, %ebx
```

请根据上述汇编代码，并参考教材上例 6.5 和第 6 章到第 9 章习题中的汇编代码进行总结：为适应函数调用引起的跨函数执行，该编译器在寄存器的值的保护方面有些什么约定？

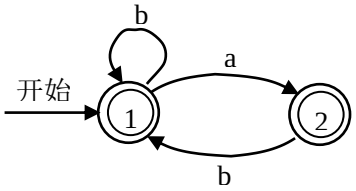
9、（10 分）下面左边的函数被 GCC: (GNU) 3.3.5 (Debian 1:3.3.5-13)优化成右边的代码。若你认为该优化结果不对或该优化不合理，则阐述你的理由。若你认为该优化结果是对的，请说明实施了哪些优化，并解释循环优化结果的合理性。

```
f(a,b,c,d,x,y,z)                    |       f:
int a,b,c,d,x,y,z;                  |           pushl     %ebp
{                                   |           movl      %esp, %ebp
    while(a<b) {                    |           movl      8(%ebp), %edx
        if (c<d)                    |           movl      12(%ebp), %eax
            x = x+z;                |           cmpl      %eax, %edx
        else                        |           jge   .L9
            x = y-z;                |       .L7:
    }                               |           jl    .L7
}                                   |       .L9:
                                    |           popl%ebp
                                    |           ret
```

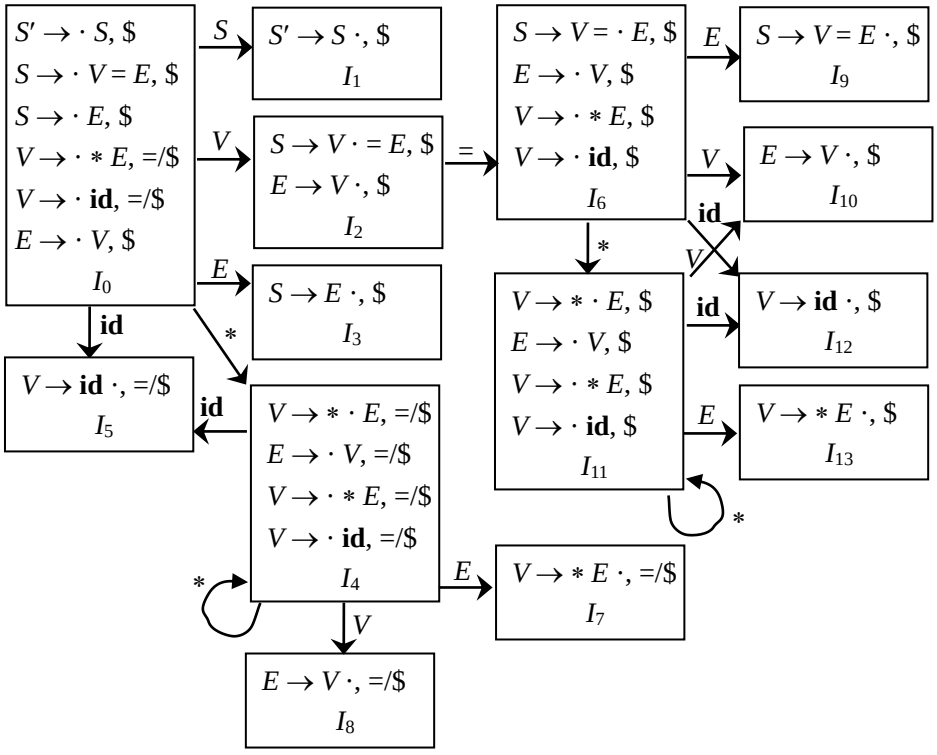
2007－2008 学年第二学期

编译原理和技术参考答案（A）

1、该语言的正规式是 $b^*(abb^*)^*(a|\varepsilon)$。接受该语言的最简 DFA 如下：



2、（1）状态转换图如下：



（2）合并同心项目集（I_4 和 I_{11}，I_5 和 I_{12}，I_7 和 I_{13}，I_8 和 I_{10}）后没有动作冲突。

3、该语言不存在 LR 文法。LR 分析是一种移进–归约分析，在扫描回文数的前一半字符时总体上是移进；而扫描后一半字符时，总体上是通过归约来比较后一半是否为前一半的逆。由于对任意的 k，不能保证最多向前看 k 个字符就能判断到达句子的中间位置，因此不能避免移进–归约冲突的出现，故不可能存在 LR 文法。

4、翻译方案如下：

$B \to 1$ { $A.i = 1$ } A { $B.val = A.s$ }

$A \to 1$ { $A_1.i = A.i \times 2 + 1$ } A_1 { $A.s = A_1.s$ }

$A \to 0$ { $A_1.i = A.i \times 2$ } A_1 { $A.s = A_1.s$ }

$A \to \varepsilon$ { $A.s = A.i$ }

5、例 7.4 是布尔表达式的控制流翻译。E_1 **xor** E_2 若生成控制流代码，则 E_2 部分的代码必须出现两次，并且两次的 $E_2.true$ 和 $E_2.false$ 正好相反，因此难以用语法制导的定义直接表达。简单的解决办法是将 E_1 **xor** E_2 等价变换成(E_1 **and not** E_2) **or** (**not** E_1 **and** E_2)，再进行语法制导的翻译。

按题目要求做的话，需要一个记号来表达代码串中的两个标号互换，见下面。

$E \to E_1$ **xor** E_2　　$E_1.true = newlabel$;　　$E_1.false = newlabel$;

$E_2.true = E.false$;　　$E_2.false = E.true$;

$E.code = E_1.code \parallel gen(E_1.true, ":") \parallel E_2.code; \parallel$

$gen(E_1.false, ":") \parallel E_2.code[E.false \leftrightarrow E.true]$;

6、虚方法表是用来实现方法的动态绑定的。类中的虚方法可以被派生类重写，导致执行方式被派生类改变，因而出现动态绑定问题。而对于非虚的方法，无论被其所在类的实例调用，还是被这个类的派生类的实例调用，方法的执行方式不变，因而没有动态绑定问题。

7、（1）以二维数组为例，$A[i_1][i_2]$的地址计算公式是

$$base + i_1 \times w_1 + i_2 \times w_2$$

其中 w_1 是存放一行元素需要的字节数，而 w_2 是存放行中一个元素需要的字节数。对于 k 维数组来说，$A[i_1][i_2]\ldots[k]$的地址计算公式是

$$base + i_1 \times w_1 + i_2 \times w_2 + \ldots + i_k \times w_k$$

其中 w_j $(1 \le j \le k)$是二维情况的 w_1 和 w_2 的推广。

（2）翻译方案如下，其中函数 *getwidth*(*p*, *k*)表示到符号表中去取该维一个元素需要的字节数（其中 *p* 是数组名在符号表中指针，*k* 表示第几维）。

$L \rightarrow L_1\ [E\ ]$　{*L.array* = L_1.*array*; *L.ndim* = L_1.*ndim*+1;

w = *getwidth*(*L.array*, *L.ndim*);

if (*L.ndim* == 1) **begin**

L.offset = *newtemp*;

emit (*L.offset*, '=', *E.place*, '*', *w*);

end else begin

t = *newtemp*; *L.offset* = *newtemp*;

emit (*t*, '=', *E.place*, '*', *w*);

emit (*L.offset*, '=', L_1.*offset*, '+', *t*);

end}

$L \rightarrow$ **id**　　{*L.place* = **id**.*place*; *L.offset* = **null**; *L.ndim* = 0; *L.array* = **id**.*place*;}

8、（1）esp 是由调用者和被调用者共同维护的栈顶地址寄存器。

（2）ebp 是活动记录的基地址寄存器，各函数在入口和出口对它原来的值进行保存和恢复。即 ebp 是由被调用者保护的寄存器。

（3）一个函数代码若使用 esi 和 ebx 这样的寄存器来保存表达式计算的中间结果，则在入口和出口对它原来的值进行保存和恢复。即 esi 和 ebx 是由被调用者保护的寄存器。

（4）一个函数使用 eax 和 edx 这样的寄存器时，不进行上述方式的保存与恢复。即 eax 和 edx 是由调用者保护的寄存器。

（5）函数的返回值存于 eax 中。

为每个函数产生目标代码时应该按上述准则来分配和使用寄存器。

9、该优化结果是对的。它主要使用了代码外提、无用赋值删除和其它无用代码删除技术。对于该函数中的循环来说：

（1）由于 c<d 是循环不变计算，条件语句每次执行选择的是同一个分支。由于其它地方没有 x 的引用，很容易判断条件语句的两个分支的 x = x+z 和 x = y-z 都是无用赋值，可以删除。进一步，条件判断 c<d 也可以删除，导致循环体为空。

（2）循环控制的判断 a<b 也是循环不变计算，外提为 movl 8(%ebp), %edx; movl 12(%ebp), %eax; cmpl %eax, %edx 三条指令。

（3）循环部分剩余的代码变成基于标志位判断的两个分支，jge .L9 表示循环结束，jl .L7 表示继续执行循环。