清华大学本科生考试试题专用纸

考试课程 《编译原理》 （A 卷） 2016 年 1 月 5 日

学号：_____ 姓名：_____ 班级：_____

（注：解答可以写在答题纸上，也可以写在试卷上；交卷时二者均需上交。）

一.（11分）必做实验相关简答题（所得分数直接加到总评成绩）

二.（10分）

1.（5分）Decaf/Mind 实验框架使用多级符号表组织方式，为每个作用域单独建立一个符号表，仅记录当前作用域中声明的标识符，同时建立一个作用域栈来管理整个程序在语义分析过程中开作用域的动态变化情况。

对于如下 Decaf/Mind 程序：

```
class Stack {
    int sp;
    int[] elems;

    void Init() {
        …               /*该语句序列中不含声明语句*/
    }

    void Push(int i) {
        …               /*该语句序列中不含声明语句*/          (A)
    }

    int Pop() {
        int val;
        …               /*该语句序列中不含声明语句*/
    }

    int NumElems() {
        return sp;
    }

    static void main() {
        class Stack s;
        …               /*该语句序列中不含声明语句*/          (B)
    }
}

class Main {
    static void main() {
        Stack.main();                                       (C)
    }
}
```

回答以下问题：

1）当分析至（A）时，作用域栈的栈顶和次栈顶作用域中分别包含哪些符号？（2分）

2）当分析至（B）时，作用域栈的栈顶和次栈顶作用域中分别包含哪些符号？（2分）

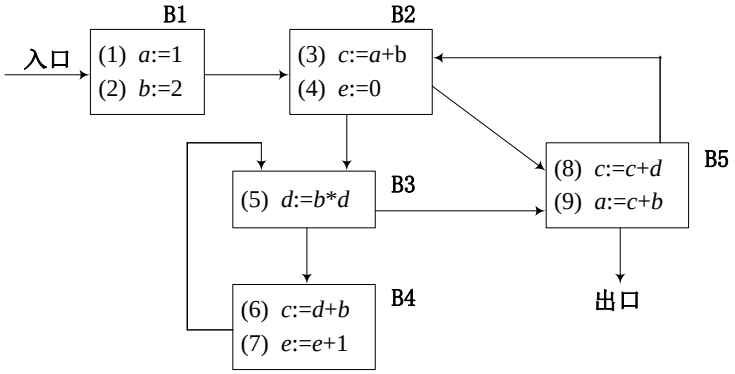
3）当分析至（C）时，作用域栈中有几个作用域？ （1分）

2.（5分） 以下是某简单语言的一段代码。语言中不包含数据类型的声明，所有变量的类型默认为整型。语句块的括号为‘begin’和‘end’组合；赋值号为 ‘:=’。每一个过程声明对应一个静态作用域。该语言支持嵌套的过程声明，但只能定义无参过程，且没有返回值。

```
（1） var x;
（2） procedure   p ;
（3）       var x;
（4）       procedure   r ;
（5）             var y;
（6）             begin
  .                    ……       /*不含任何 call 语句和声明语句*/
（l1）                 call q;
  .              end;
  .         begin
（l2）              call r ;
  .                ……       /*不含任何 call 语句和声明语句*/
  .         end ;
  .  procedure   q ;
  .         var y;
  .         begin
  .               ……       /*不含任何 call 语句和声明语句*/
（l3）              call p ;
  .         end ;
  .  begin
  .         ……       /*不含任何 call 语句和声明语句*/
  .         call p;
  .  end .
```

若过程活动记录中的控制信息包括了静态链 SL 和动态链 DL。当过程 q 被第二次激活时，运行栈上共有几个活动记录？当前位于栈顶的活动记录中静态链 SL 和动态链 DL 分别指向什么位置？（注：指出是第几个活动记录的起始位置即可，假设主过程对应于第1个活动记录，之后被激活的活动记录分别是第2个、第3个、..）。若程序的执行遵循静态作用域规则，则在当前访问的变量 x 应当属于哪个活动记录？若程序的执行遵循动态作用域规则，则在当前访问的变量 x 应当属于哪个活动记录？

三 **(16分)** 下图是包含 5 个基本块（略去了其中的跳转语句或停语句）的流图，其中 B1 为入口基本块：



1. (2分) 指出在该流图中对应于回边 B5→B2 的自然循环（即这些循环中包含哪些基本块）。

2. (4分) 已知基本块B2和B3出口处的**到达-定值**（reaching definitions）信息分别为

   Out(B2) = {1,2,3,4, 5,9}　　和　　Out(B3) = {1,2,3,4, 5,6,7,9}

   试计算 Out(B5)=？

3. (2分) 指出该流图范围内，变量 *c* 在（8）的 UD 链。

4. (2分) 指出该流图范围内，变量 *c* 在（8）的 DU 链。

5. (3分) 对于该流图，假定已求解出**活跃变量**（live variables）信息，如下表所示：

	LiveUse	**DEF**	**LiveIn**	**LiveOut**
B1	∅	{a,b}	{d}	{ a,b,d}
B2	{a,b}	{c,e}	{a,b,d}	{ b,c,d,e}
B3	{b,d}	∅	{b,c,d,e}	{b,c,d,e}
B4	{b,d,e}	{c}	{b,d,e}	{ b,c,d,e}
B5	{c,d}	{a}	{ b,c,d}	{ a,b,d}

   试计算在定值点 (1)，(6) 和 (8) 之后的活跃变量分别有哪些？

6. (3分) 对于该流图，给出相应的寄存器相干图。若采用课堂介绍的启发式算法，要保证图着色过程中不会出现将寄存器泄漏到内存中的情形，那么可供分配的物理寄存器的最小数目是多少？

四 (12 分) 给定文法 *G*[*S*]： $S \to R$

$R \to aR \mid bRaR \mid \varepsilon$

1. (5分) 针对文法*G*[*S*]，下表给出各产生式右部文法符号串的 *First* 集合，各产生式左部非终结符的 *Follow* 集合，以及各产生式的预测集合*PS*。试填充其中空白表项的内容：

G 中的规则 r	*First* (*rhs*(r))	*Follow* (*lhs*(r))	*PS* (r)
$S \to R$		#	
$R \to aR$	a		a
$R \to bRaR$	b	此处不填	b
$R \to \varepsilon$		此处不填	

表中的 *rhs(r)* 表示产生式 r 右部的文法符号串，*lhs(r)* 表示产生式 r 左部的非终结符。

2. (4分) 以下是文法 $G[S]$ 的预测分析表，一些表项的内容未给出，试补充完成：

	a	b	#
S	$S \to R$	$S \to R$	$S \to R$
R			

3. (3分) 说出 $G[S]$ 是否 *LL*(1) 文法。为什么？

注意：第1题是否正确会影响到后两题的回答，因此首先要保证第1题正确。

五 (10 分) 给定文法 $G[S]$： $S \to R$
$R \to aR \mid bRaR \mid c$

各产生式的预测集合分别为 $PS(S \to R)=\{a, b, c\}$
$PS(R \to aR)=\{a\}$
$PS(R \to bRaR)=\{b\}$
$PS(R \to c)=\{c\}$

显然，$G[S]$为 LL(1) 文法。

如下是以$G[S]$为基础文法的一个 L 翻译模式：

（1）$S \to \{ R.i := 0 \}\ R\ \{ \text{print } (R.d) \}$
（2）$R \to a\ \{ R_1.i := R.i + 1 \}\ R_1\ \{ R.d := R_1.d \}$
（3）$R \to b\ \{ R_1.i := R.i \}\ R_1\ \ a\ \{ R_2.i := 1 \}\ R_2\ \{ R.d := R_1.d + R_2.d - 1 \}$
（4）$R \to c\ \{ R.d := R.i \}$

1. (3分) 对于合法输入串 *abbacaacaabcac*，该翻译模式的语义计算结果是什么？（回答print的执行结果即可，这里print为普通显示语句）

2. (7分) 针对该翻译模式构造一个自上而下的递归下降（预测）翻译程序（伪码）

```
void ParseS()                    // 主函数
{
    switch (lookahead)  {              // lookahead 为下一个输入符号
        case 'a','b', 'c':
            .....                  // 填充此处          (2 分)
            break;
        default:
```

```
                printf("syntax error \n")
                exit(0);
        }
}

int ParseR(int i)
{
        switch (lookahead)  {              // lookahead 为下一个输入符号
                case ′ a‘:
                        .....              //  填充此处          （2 分）
                        break;
                case ‘b’:
                        .....              //  填充此处          （3 分）
                        break;
                case ‘c’:
                        MatchToken(‘c’) ;
                        Rd:= i;
                        break;
                default:
                        printf("syntax error \n")
                        exit(0);
        }
        return Rd;
}
```

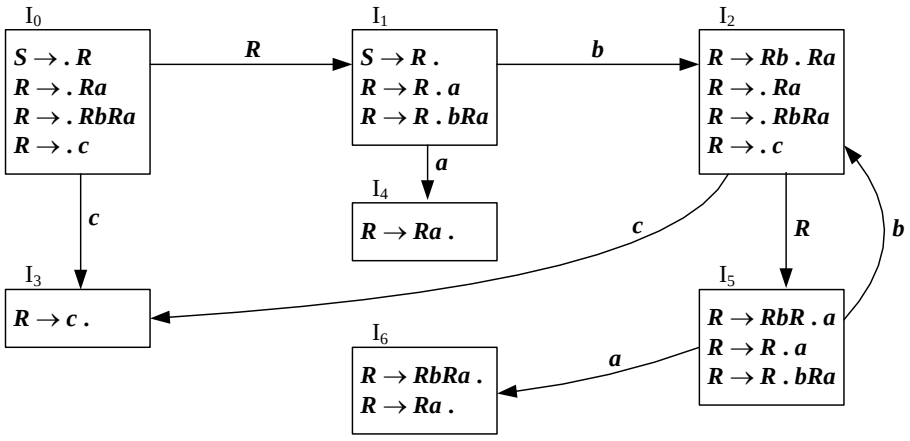
试填充完成函数 ParseS 和 ParseR 中未给出的部分。

六 (24 分)

给定如下文法 $G[R]$：

(1) $R \rightarrow Ra$
(2) $R \rightarrow RbRa$
(3) $R \rightarrow c$

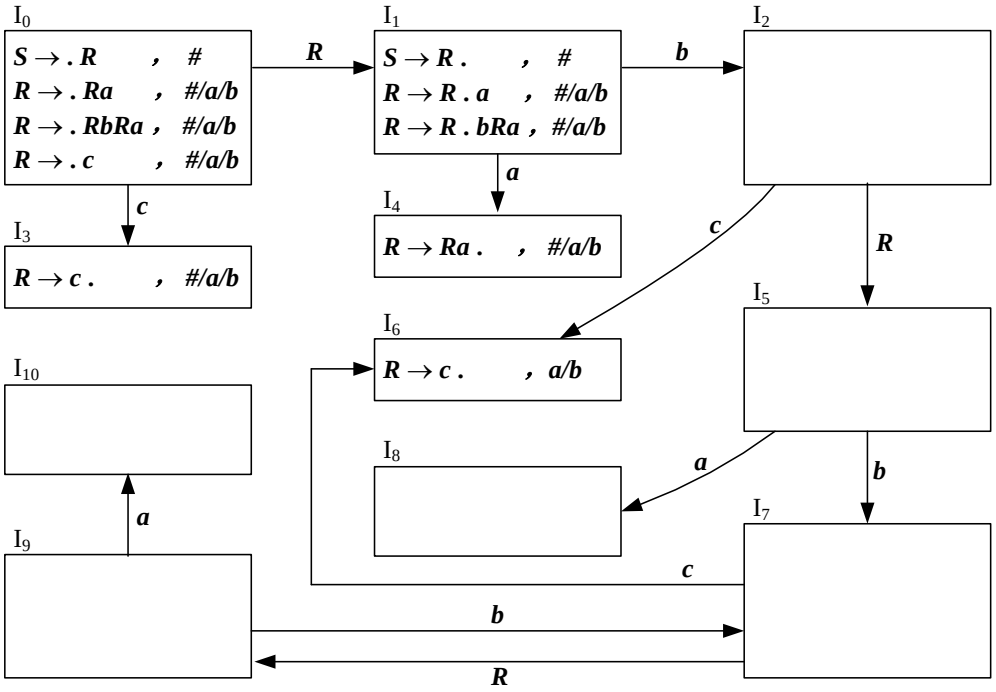
为文法 G[R] 增加产生式 $S \rightarrow R$，得到增广文法 $G'[S]$，下图是相应的LR(0)自动机：



1. (2分) 指出LR(0)自动机中的全部冲突状态及其冲突类型，以说明文法G[R]不是LR(0)文法。

2. (2分) 文法G[R]也不是SLR(1)文法。为什么？

3. (6分) 下图表示相应于增广文法 $G'[S]$ 的 LR(1) 自动机，部分状态所对应的项目集未给出，试补齐之（即分别给出状态 I_2，I_5，I_7，I_8，I_9， 和 I_{10} 对应的项目集。



4. (2分) 指出LR(1)自动机中的全部冲突状态，这说明文法 $G[R]$ 也不是 LR(1) 文法。

5. (6分) 若规定某种匹配原则，则可以得到如下SLR(1)分析表：

状态	ACTION				GOTO
	a	*b*	*c*	#	*R*
0			s3		1
1	s4	s2		acc	
2			s3		5
3	r3	r3		r3	
4	r1	r1		r1	
5	s6	s2			
6	r2	r2		r2	

若规定同样的匹配原则，则也可以解决LR(1)自动机中的全部冲突状态。以下是在遵循这一匹配原则的情况下构造的 LR(1) 分析表，其中状态 2，5，7，8，9 和 10 对应的行未给出，试补齐之：

状态	ACTION				GOTO
	a	*b*	*c*	#	*R*
0			s3		1
1	s4	s2		acc	
2					
3	r3	r3		r3	
4	r1	r1		r1	
5					
6	r3	r3			
7					
8					
9					
10					

6. (6分) 对于文法 *G*[*R*] 中正确的句子，基于上述两个分析表均可以成功进行*LR*分析。然而，对于不属于文法 *G*[*R*] 中的句子，两种分析过程发现错误的速度有所不同，即发现错误时所经过的移进/归约总步数有差异。

   试问在一般情况下，基于 SLR(1) 和 LR(1) 两种分析表的分析过程，哪一个发现错误的速度更快些？ （2分）

   试给出一个不能被 G[R] 接受的输入串，其长度不超过6，使得两种分析过程发现错误的步数不相同。 （4分）

七 (10 分)

给定如下文法 *G*[*S*]：

（1）$S \rightarrow R$
（2）$R \rightarrow Ra$
（3）$R \rightarrow R\ a\ R\ b$
（4）$R \rightarrow \varepsilon$

如下是以*G*[*S*]为基础文法的一个 *L* 翻译模式：

（1）$S \rightarrow \{\ R.i := 0\ \}\ R\ \{\ \text{print}\ (R.d)\ \}$
（2）$R \rightarrow \{\ R_1.i := R.i + 1\ \}\ R_1\ a\ \{\ R.d := R_1.d\ \}$
（3）$R \rightarrow \{\ R_1.i := R.i\ \}\ R_1\ a\ \{\ R_2.i := 1\ \}\ R_2\ b\ \{\ R.d := R_1.d + R_2.d - 1\}$
（4）$R \rightarrow \varepsilon\ \{\ R.d := R.i\ \}$

引入非终结符 *M*，*N* 和 *P* 及其 ε 产生式，并在上述翻译模式的基础上添加如下三条语义计算规则：

（5）$M \rightarrow \varepsilon$ $\{\ M.s := 0\ \}$
（6）$N \rightarrow \varepsilon$ $\{\ N.s := N.i + 1\ \}$
（7）$P \rightarrow \varepsilon$ $\{\ P.s := 1\ \}$

1. (4分) 试用 *M*，*N* 和 *P* 替代嵌在产生式（1）~（3）中间的非复写语义动作，添加适当的复写规则，使得嵌在产生式中间的语义动作集中仅含复写规则，并使得在自底向上的语义计算过程中，文法符号的所有继承属性均可以通过归约前已出现在分析栈中的确定的综合属性进行访问。

2. (6分) 如果在 LR 分析过程中根据这一修改后新的翻译模式进行自下而上翻译，试写出在按每个产生式归约时语义处理的一个代码片断（设语义栈由向量 *val* 表示，归约前栈顶位置为 *top*，终结符不对应语义值，而每个非终结符的综合属性都只对应一个语义值，可用 *val*[*i*].*s* 或 *val*[*i*].*d* 表示；不用考虑对 *top* 的维护）。

八 (18分)

1. (6分) 以下是一个L-翻译模式片断，描述了某小语言中语句相关的一种类型检查工作：

$P \rightarrow D\ ;\ S$ { *P.type* := if *D.type* = *ok* and *S.type* = *ok* then *ok* else *type_error* }
$S \rightarrow$ if E then S_1 { *S.type* := if *E.type*=*bool* then S_1.*type* else *type_error* }
$S \rightarrow$ while E do S_1 { *S.type* := if *E.type*=*bool* then S_1.*type* else *type_error* }
$S \rightarrow S_1\ ;\ S_2$ { *S.type* := if S_1.*type* = *ok* and S_2.*type* = *ok* then *ok* else *type_error* }

其中，type 属性以及类型表达式 *ok*，*type_error*，*bool* 等的含义与讲稿中一致。

（此外，假设在语法制导处理的分析过程中遇到的冲突问题均可以按照某种原则处理，这里不必考虑基础文法是否 *LR* 文法。）

下面叙述本小题的要求：

若在基础文法中增加关于“带条件的继续循环”语句的产生式

$S \rightarrow$ continue when E

语句 continue when E 只能出现在某个循环语句内，即至少有一个包围它的 while；同时 *E* 必须为 *bool* 类型。

试在上述翻译模式片段基础上增加相应的的语义处理内容（要求是 *L*-翻译模式），以实现针对“带条件的继续循环”语句的这一类型检查任务。（提示：可以引入 *S* 的一个继承属性）

2. (6分) 以下是一个*L*-翻译模式片断，可以产生控制语句的 *TAC* 语句序列：

$P \rightarrow D$; { *S.next* := *newlabel* } S { *gen*(*S.next* ‘:’) }
$S \rightarrow$ if { *E.true* := *newlabel*; *E.false* := *S.next* } E then
{ S_1.*next* := *S.next* } S_1 { *S.code* := *E.code* || *gen*(*E. true* ‘:’) || S_1.*code* }
$S \rightarrow$ while { *E.true* := *newlabel*; *E.false* := *S.next* } E do
{ S_1.*next* := *newlabel* } S_1
{ *S.code* := *gen*(S_1.*next* ‘:’) || *E.code*|| *gen*(*E. true* ‘:’) ||
S_1.*code* || *gen*(‘*goto*’ S_1.*next*) }
$S \rightarrow$ { S_1.*next* := *newlabel* } S_1 ;
{ S_2.*next* := *S.next* } S_2
{ *S.code* := S_1.*code* || *gen*(S_1.*next* ‘:’) || S_2.*code* }

其中，属性 *S .code* , *E .code* , *S .next* , *E.true* , *E.false*, 语义函数 *newlabel* , *gen( )* 以及所涉及到的 *TAC* 语句与讲稿中一致：继承属性 *E.true* 和 *E.false* 分别代表 *E* 为真和假时控制要转移到的程序位置，即标号；综合属性 *E.code* 表示对 *E* 进行求值的 *TAC* 语句序列；综合属性 *S.code* 表示对应于 *S* 的 *TAC* 语句序列；继承属性 *S.next* 代表退出 *S* 时控制要转移到的语句标号。语义函数 *gen* 的结果是生成一条 *TAC* 语

句；“||”表示 *TAC* 语句序列的拼接。

（此外，假设在语法制导处理的分析过程中遇到的冲突问题均可以按照某种原则处理，这里不必考虑基础文法是否 LR 文法。）

下面叙述本小题的要求：

若在基础文法中增加关于“带条件的继续循环”语句的产生式

$$S \rightarrow \text{continue when } E$$

语句 continue when *E* 只能出现在某个循环语句内，即至少有一个包围它的 while 语句，其执行语义为：当 *E* 为真时，跳出直接包含该语句的 while 循环体，并回到 while 循环的开始处判断循环条件重新执行该循环；当 *E* 为假时，不做任何事情。

试在上述 *L*-翻译模式片段基础上增加针对 continue when *E* 语句的语义处理内容（不改变 *L*-翻译模式的特征，不考虑“是否出现在while 语句内”的语义检查工作）。

注：可设计引入新的属性或删除旧的属性，必要时给出解释。

3. (6分) 以下是一个 *S*-翻译模式/属性文法片断，可以产生控制语句的 *TAC* 语句序列：

$P \rightarrow D\ ;\ S\ M$ { *backpatch*(*S.nextlist, M.gotostm*) }

$S \rightarrow \text{if } E \text{ then } M_1\ S_1$ { *backpatch*(*E.truelist, M*$_1$*.gotostm*) ;
S.nextlist := *merge*(*E.falselist, S*$_1$*.nextlist*) }

$S \rightarrow \text{while } M_1\ E \text{ do } M_2\ S_1$ { *backpatch*(*S*$_1$*.nextlist, M*$_1$*.gotostm*) ;
backpatch(*E.truelist, M*$_2$*.gotostm*) ;
S.nextlist := *E.falselist*;
emit(‘*goto*’, *M*$_1$*.gotostm*) }

$S \rightarrow S_1\ ;\ M\ S_2$ { *backpatch*(*S*$_1$*.nextlist, M*$_1$*.gotostm*) ;
S.nextlist := *S*$_2$*.nextlist* }

$M \rightarrow \varepsilon$ { *M.gotostm* := *nextstm* }

$N \rightarrow \varepsilon$ { *N.nextlist* := *makelist*(*nextstm*); *emit*(‘*goto* _’) }

其中，所用到的属性和语义函数与讲稿中一致：综合属性 *E.truelist*（真链），表示一系列跳转语句的地址，这些跳转语句的目标语句标号是体现布尔表达式 *E* 为“真”的标号；综合属性 *E.falselist*（假链），表示一系列跳转语句的地址，这些跳转语句的目标语句标号是体现布尔表达式 *E* 为“假”的标号；综合属性 *S. nextlist*（*next*链），表示一系列跳转语句的地址，这些跳转语句的目标语句标号是在执行序列中紧跟在 *S* 之后的下条 *TAC* 语句的标号；语义函数*makelist*(*i*)，用于创建只有一个结点 *i* 的表，对应于一条跳转语句的地址；语义函数*merge*(p_1, p_2)，表示连接两个链表 p_1 和 p_2，返回结果链表；语义函数*backpatch*(*p, i*)，表示将链表 *p* 中每个元素所指向的跳转语句的标号置为*i*；语义函数*nextstm*，将返回下一条*TAC* 语句的地址；语义函数*emit*(…)，将输出一条*TAC* 语句，并使 *nextstm* 加1。

（和前面一样，假设在语法制导处理的分析过程中遇到的冲突问题均可以按照某种原则，这里不必考虑基础文法是否 *LR* 文法。）

下面叙述本小题的要求：

若在基础文法中增加关于“带条件的继续循环”语句的产生式

$$S \rightarrow \text{continue when } E$$

语句 continue when *E* 只能出现在某个循环语句内，即至少有一个包围它的 while 语句，其执行语义为：当 *E* 为真时，跳出直接包含该语句的 while 循环体，并回到

while 循环的开始处判断循环条件重新执行该循环；当 E 为假时，不做任何事情。

试在上述 *S*-翻译模式片段基础上增加针对 continue when E 语句的语义处理内容（不改变 *S*-翻译模式的特征，不考虑“是否出现在while 语句内”的语义检查工作）。

注：可设计引入新的属性或删除旧的属性，必要时给出解释。