清华大学本科生考试试题专用纸

考试课程 《编译原理》 （B 卷） 2014 年 1 月 7 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；交卷时二者都需要上交。）

一.（12分）Decaf/Mind 实验相关简答题（所得分数直接加到总评成绩）

二.（15分）

下面是一段 Decaf/Mind 程序：

```
class Fruit
{
    int price;
    string name;
    void init(int p, string s)  {price=p; name=s;}
    void print(){
        Print(“The price of ",color," ",name," is ", price,"\n");
    }
}
class Apple extends Fruit
{
    string color;
    void setcolor(string c)  {color=c;}
    void print(){
        Print(“The price of ",color," ",name," is ", price,"\n");
    }
}
class Main {
{
    static void main() {
        class Apple a;
        a=new Apple();
        a.setcolor("red");
        a.init(100,"apple");
        a.print();
    }
}
```

1. （6分）Decaf/Mind 编译器使用一个作用域栈来记录当前的开作用域。试结合本课程主体实验内容，回答下列问题：

   （1）在分析到 class Apple 中 setcolor 方法内的语句 color = c 时，当前作用域中有哪些符号？次栈顶作用域中有哪些符号？此时的color属于哪个作用域？ （4分）

   （2）根据目前课程实验的框架，若将Fruit和Apple的代码改一下前后顺序，二者之间

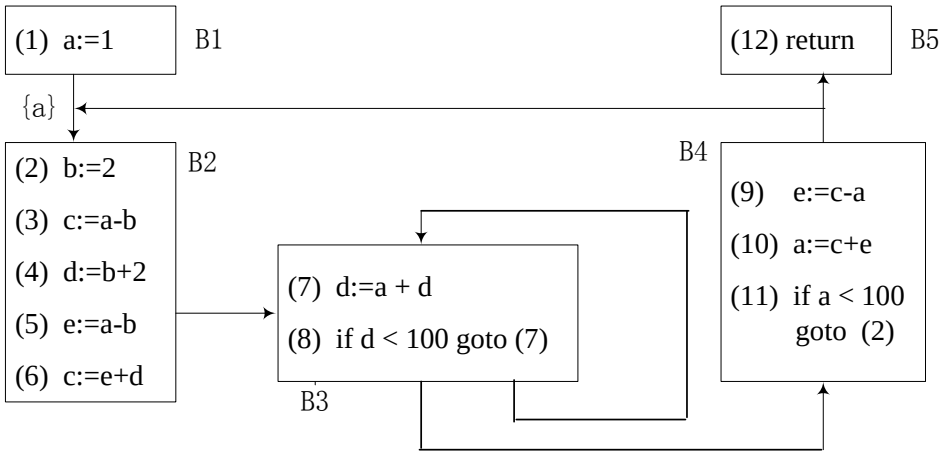
extends关系的处理会不会有影响？若是将符号表的建立调整到第一阶段语法分析的同时来进行，那么Fruit和Apple的顺序会不会影响到extends关系的处理？　　（2分）

2. （9分）结合本课程主体实验中Decaf/Mind 编译器的运行时组织，回答下列问题：

（1）在进行过各类初始化后开始执行主函数，执行完语句 a = New Apple 时，与执行前相比栈区和堆区的数据信息有什么变化？概要叙述这些信息的具体内容（不必给出每项内容的相对位置和大小等信息）。　　（6分）

（2）在执行 a.init(100, “apple”) 时，如何找到方法init的代码位置？　（3分）

三 **(10分)** 下图是包含 5 个基本块的流图，其中 B1 为入口基本块，B5 为出口基本块：



1. (2分) 指出在该流图中的所有回边，并指出对应这些回边的自然循环（即这些循环中包含哪些基本块）。

2. (4分) 对于上图所给出的流图，采用迭代求解数据流方程的方法对**活跃变量信息**进行分析。假设 B1 的 LiveIn 和 B5 的 LiveOut 信息为 ∅，则迭代结束时的结果如下图所示。其中，基本块 B2 一行对应的信息未给出，请补齐之。

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

3. (2分) 指出该流图范围内，变量 d 在（4）的 DU 链。

4. (2分) 指出该流图范围内，变量 c 在（10）的 UD 链。

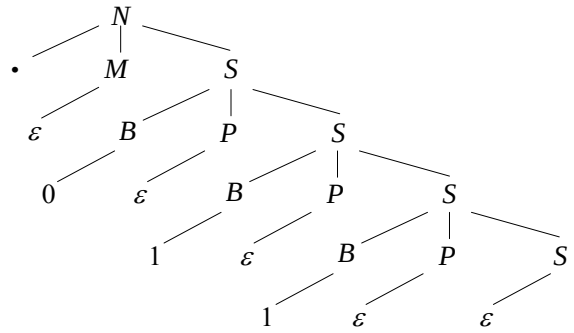
四 (22 分) 给定 LL(1) 文法 $G[N]$:

（1）$N \rightarrow .\ M\ S$
（2）$S \rightarrow B\ P\ S_1$
（3）$S \rightarrow \varepsilon$
（4）$B \rightarrow 0$
（5）$B \rightarrow 1$
（6）$M \rightarrow \varepsilon$
（7）$P \rightarrow \varepsilon$

如下是以G[*N*]为基础文法的一个*L* 翻译模式，其所有继承属性均可用综合属性模拟:

（1）$N \rightarrow .\ M$ { *S.f* : = *M.s* } *S* { *print*(*S.v*) }
（2）$S \rightarrow$ { *B.f* : =*S.f* } *B* { *P.i* :=*S.f* } *P* { S_1*.f* := *P.s* } S_1 { *S.v* := *B.v* + S_1*.v* }
（3）$S \rightarrow \varepsilon$ { *S.v* := 0 }
（4）$B \rightarrow 0$ { *B.v* := 0 }
（5）$B \rightarrow 1$ { *B.v* := 2^(-*B. f*)}
（6）$M \rightarrow \varepsilon$ { *M.s* :=1}
（7）$P \rightarrow \varepsilon$ { *P.s* := *P.i* +1 }

1. (3分) 输入串 **.011** 的一棵语法分析树如下图所示。通过对该语法分析树进行标注，可得到一棵带标注的语法分析树。试指出在这一带标注的语法分析树中所有4个S符号对应的属性值。（注意：每个S符号有2个属性，*S.f* 和 *S.v*）



2. (5分) 试分别指出句型 **.***MBS* 和 **.***MBPBPS* 的所有短语, 直接短语。如果这些句型同时也是右句型，那么还要给出其句柄 . 请将结果填入下表中:

句型	短语	直接短语	句柄
.*MBS*			
.*MBPBPS*			

3. (6分) 如果在 LR 分析过程中根据该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].*v* 或 val[i].s 表示；不用考虑对 top 的维护）。

4. (8分) 如下是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

```
void ParseN()                    // 主函数
```

```
{
    MatchToken('.');          // 匹配'.'，直接使用讲稿中的 MatchToken 函数，下同
    Ms : = ParseM( );
    Sf : = Ms;
    Sv : = ParseS(Sf);
    print(Sv);
}

float ParseS(int f )
{
    ……
}

float ParseB(int f)
{
    switch (lookahead)  {              // lookahead 为下一个输入符号
        case ′0':
            MatchToken('0') ;
            Bv := 0;
            break;
        case '1':
            MatchToken('1') ;
            Bv := 2^(- f )
            break;
        default:
            printf("syntax error \n")
            exit(0);
    }
    return Bv;
}

int ParseM( )
{
    Ms :=1;
    return Ms;
}

int ParseP(int i )
{
    ……
}
```

试填充完成函数 ParseS 和 ParseP 的整个函数体。

五 (13 分) 给定如下文法 $G[S]$：

$S \rightarrow \underline{\texttt{if}}\ S\ \underline{\texttt{else}}\ S$
$S \rightarrow \underline{\texttt{if}}\ S$
$S \rightarrow \texttt{a}$

提取左公因子后，并用 i 表示 if，用 e 表示 else，得到新文法$G'[S]$：

$S \rightarrow \texttt{i}\ S\ A$
$A \rightarrow \texttt{e}\ S \mid \varepsilon$
$S \rightarrow \texttt{a}$

1. (5分) 针对文法*G'*[*S*]，试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*，即完成下表：

G 中的规则 *r*	*First* (*rhs*(*r*))	*Follow* (*lhs*(*r*))	*PS* (*r*)
S → i *S A*			
A → e *S*			
A → ε		此处不填	
S → a		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. (3分) 给出文法*G'*[*S*]的预测分析表，即完成下表：

	i	e	a	#
S				
A				

3. (1分) 从上述预测分析表可看出*G'*[*S*]不是 *LL*(1) 文法，请说出为什么？

4. (4分) 虽然*G'*[*S*]不是 *LL*(1) 文法，但通过强制优先使用能匹配输入中符号 else 的产生式，则还是有可能使用 *LL*(1) 分析方法。例如，对输入串iiaea#的一个表驱动自上而下分析步骤为：

步骤	下推栈	余留符号串	下一步动作
1	#*S*	iiaea#	应用产生式 *S*→ i*SA*
2	# *AS*i	iiaea#	匹配栈顶和当前输入符号
3	# *AS*	iaea#	应用产生式 *S* → i*SA*
4	# *AAS*i	iaea#	匹配栈顶和当前输入符号
5	# *AAS*	aea#	应用产生式 *S* → a
6	# *AA* a	aea#	匹配栈顶和当前输入符号
7			
8			
9			
10			
11	# *A*	#	应用产生式 *A* → ε
12	#	#	返回，分析成功

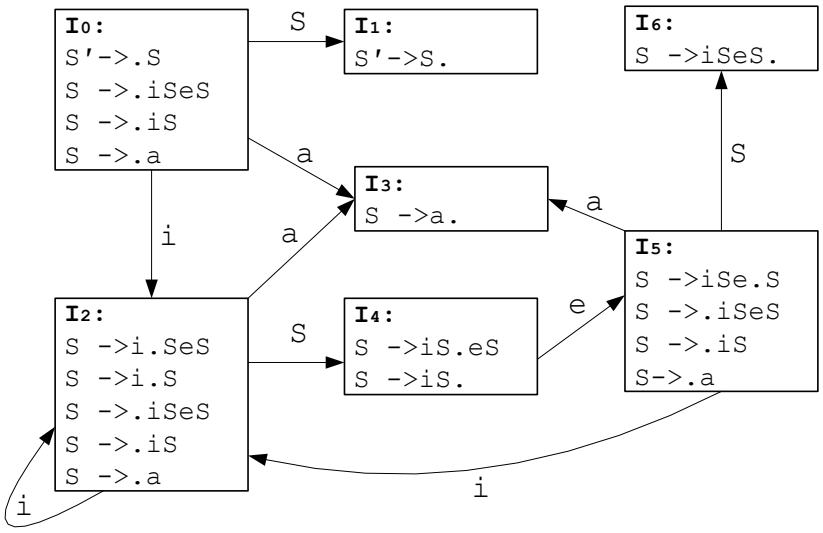
这个分析步骤中的第7~10步未给出，试补齐之。

六 (22分)

给定如下文法 $G[S]$：

(1) $S \rightarrow$ if S else S
(2) $S \rightarrow$ if S
(3) $S \rightarrow$ a

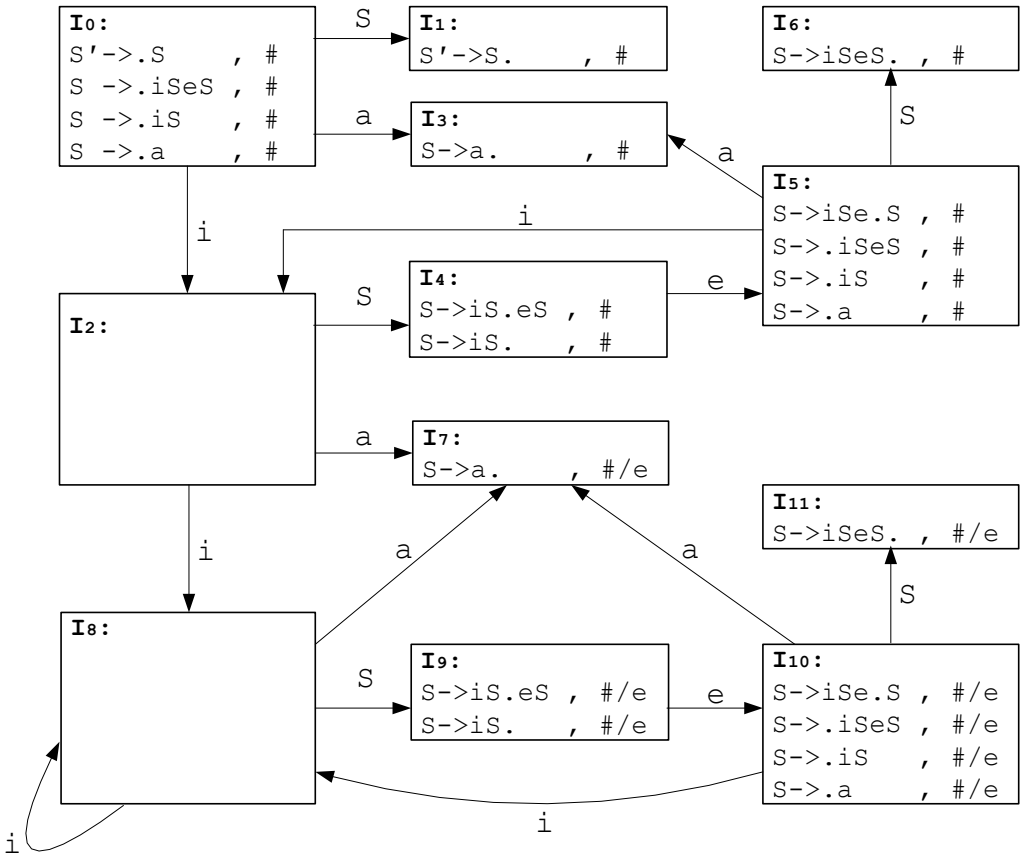
为文法 $G[S]$ 增加产生式 $S' \rightarrow S$，得到增广文法 $G'[S']$，下图是相应的 LR(0) 自动机（i 表示 if，e 表示 else）：



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$G[S]$不是LR(0)文法。

2. (3分) 文法$G[S]$也不是SLR(1)文法。为什么？

3. (4分) 下图表示文法$G[S]$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 和 I_8）。



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

5. (6分) 若规定最近匹配原则，即 else 优先匹配左边靠近它的未匹配的if，则可以解决上述2个自动机中的状态冲突。下图表示文法G[S]在规定这一规则情况下的SLR(1)分析表，状态 4~6 对应的行未给出，试补齐之 。

状态	ACTION				GOTO
	i	e	a	#	S
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4					
5					
6					

下图表示文法G[S] 在规定这一规则情况下的LR(1)分析表，状态 4，7 和 9 对应的行未给出，试补齐之 。

状态	ACTION				GOTO
	i	e	a	#	S
0	s2		s3		1
1				acc	
2	s8		s7		4
3				r3	
4					
5	s2		s3		6
6				r1	
7					
8	s8		s7		9
9					
10	s8		s7		11
11		r1		r1	

6. (5分) 对于文法G[S]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[S]中的句子，两种分析过程发现错误的速度不同，即发现错误时所经过的移进/归约总步数有差异。试给出一个长度不超过10的句子（即所包含的终结符个数不超过10），使得两种分析过程发现错误的速度不同。哪一个更快？对于你给的例子，两种分析过程分别到达哪个状态会发现错误？

七 (18分)

1. （4分）以下是与课程简单语言中表达式相关的翻译模式片断，其作用是计算表达式相关语法单位的类型信息，同时检查表达式中运算数类型与给定运算是否兼容：

$E \rightarrow$ true	{ *E.type* := *bool* }
$E \rightarrow$ false	{ *E.type* := *bool* }
$E \rightarrow$ int	{ *E.type* := *int* }
$E \rightarrow$ real	{ *E.type* := *real* }
$E \rightarrow$ id	{ *E.type* := if *lookup_type*(*id.name*) = nil then *type_error* else *lookup_type*(*id.name* }
$E \rightarrow E_1$ op E_2	{ *E.type* := if E_1*.type=real* and E_2*.type=real* then *real* else if E_1*.type=int* and E_2*.type=int* then *int* else *type_error* }
$E \rightarrow E_1$ rop E_2	{ *E.type* := if E_1*.type=real* and E_2*.type=real* then *bool* else if E_1*.type=int* and E_2*.type=int* then *bool* else *type_error* }

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

（此外，假设在语法制导处理过程中遇到的冲突问题均可以按照某种原则处理，比如规定运算的优先级与结合性，等等，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条件表达式的产生式

$$E \to E\ ?\ E : E$$

条件表达式 $e_1 ? e_2 : e_3$ 的类型规则要求：e_1 为布尔表达式，e_2 和 e_3 为类型相同的算术表达式，表达式的结果类型即为这个类型。

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 *S*-翻译模式），以实现针对条件表达式的类型检查任务。

2. （6分）以下是一个*S*-翻译模式片断，可以产生相应于赋值语句、算术表达式和布尔表达式的*TAC* 语句序列：

$S \to \underline{\text{id}} := A$	{ *S.code* := *A.code* || *gen*(id .*place* ':=' *A.place*) }
$A \to \underline{\text{id}}$	{ *A.place* := id .*place* ; *A.code* := "" }
$A \to \underline{\text{int}}$	{ *A.place* := *newtemp*; *A.code* := *gen* (*A.place* ':=' int .*val*) }
$A \to A_1 + A_2$	{ *A.place* := *newtemp*; *A.code* := A_1.*code* || A_2.*code* || *gen* (*A.place* ':=' A_1.*place* '+' A_2.*place*) }
$A \to (A_1)$	{ *A.place* := A_1.*place* ; *A.code* := A_1.*code* }
$E \to E_1 \vee E_2$	{ *E.place* := *newtemp*; *E.code* := E_1.*code* || E_2.*code* || *gen* (*E.place* ':=' E_1.*place* 'or' E_2.*place*) }
$E \to \neg E_1$	{ *E.place* := *newtemp*; *E.code* := E_1.*code* || *gen* (*E.place* ':=' 'not' E_1.*place*) }
$E \to (E_1)$	{ *E.place* := E_1.*place* ; *E.code* := E_1.*code* }
$E \to \underline{\text{id}}_1\ \underline{\text{rop}}\ \underline{\text{id}}_2$	{ *E.place* := *newtemp*; *E.code* := *gen* ('if ' id1.*place* rop.op id2.*place* 'goto' *nextstat*+3) || *gen* (*E.place* ':=' '0') || *gen* ('goto' *nextstat*+2) || *gen* (*E.place* ':=' '1') }
$E \to$ true	{ *E.place* := *newtemp*; *E.code* := *gen* (*E.place* ':=' '1') }
$E \to$ false	{ *E.place* := *newtemp*; *E.code* := *gen* (*E.place* ':=' '0') }

其中，各属性以及语义函数的含义与讲稿中一致：

id .*place* 表示相应的名字对应的存储位置；int .*val* 表示常量值；*A*生成算术表达式，综合属性*A.place* 表示存放 *A* 的值的存储位置；*E*生成布尔表达式，综合属性 *E.place* 表示存放 *E* 的值的存储位置；综合属性 *A.code*，*E.code* 和*S.code* 分别表示相应计算的*TAC*语句序列。

语义函数 *gen* 的结果是生成一条 *TAC* 语句；语义函数 *newtempt* 的作用是在符号表中新建一个从未使用过的名字，并返回该名字的存储位置；|| 是*TAC* 语句序列之间的链接运算；语义函数*nextstat* 返回输出代码序列中下一条 *TAC* 语句的下标。

（和前面一样，假设在语法制导处理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条件表达式的产生式

$$A \to E\ ?\ A : A$$

条件表达式 $e_1 ? e_2 : e_3$ 的含义与C语言中的条件表达式类似：布尔表达式e_1的求值结果为真，则取算术表达式e_2的求值结果，否则取算术表达式e_3的求值结果。

试在上述 *S*-翻译模式片段基础上增加针对条件表达式的语义处理内容（不改变 *S*-翻译模式的特征）。

提示：*E.place* 处取值为 "1"表示布尔表达式取"真"值，"0"表示取"假"值。

注：设计时如果需要引入其他新的属性或删除旧的属性，请给出相应的解释。

3. （8分）以下是一个*L*-翻译模式片断，可产生赋值语句、算术表达式、布尔表达式以及控制语句的*TAC* 语句序列：

$S \to \underline{\text{id}} := A$ { *S.code* := *A.code* || *gen*(id .*place* ':=' *A.place*) }
$A \to \underline{\text{id}}$ { *A.place* := id .*place* ; *A.code* := "" }
$A \to \underline{\text{int}}$ { *A.place* := *newtemp*; *A.code* := *gen* (*A.place* ':=' int .*val*) }
$A \to A_1 + A_2$ { *A.place* := *newtemp*;
A.code := A_1.*code* || A_2.*code* ||
gen (*A.place* ':=' A_1.*place* '+' A_2.*place*) }
$A \to (A_1)$ { *A.place* := A_1.*place* ; *A.code* := A_1.*code* }
$E \to$ { E_1.*true* := *E.true*; E_1.*false* := *newlabel* } $E_1 \vee$
{ E_2.*true* := *E.true*; E_2.*false* := *E.false* } E_2
{ *E.code* := E_1 .*code* || *gen* (E_1.*false* ':') || E_2 .*code* }
$E \to$ { E_1.*false* := *E.false*; E_1.*true* := *newlabel* } $E_1 \wedge$
{ E_2.*false* := *E.false*; E_2.*true* := *E.true* } E_2
{ *E.code* := E_1 .*code* || *gen* (E_1.*true* ':') || E_2 .*code* }
$E \to \neg$ { E_1.*true* := *E.false*; E_1.*false* := *E.true* } E_1 { *E.code* := E_1.*code* }
$E \to \underline{\text{id}}_1$ rop $\underline{\text{id}}_2$ { *E.code* := *gen* ('if ' id1.*place* rop.op id2.*place* 'goto' *E.true*) ||
gen ('goto' *E. false*) }
$E \to$ true { *E.code* := *gen* ('goto' *E.true*) }
$E \to$ false { *E.code* := *gen* ('goto' *E. false*) }
$S \to$ if { *E.true* := *newlable*; *E.false* := *newlable* } *E* then
{ S_1.*next* := *S.next* } S_1 else { S_2.*next* := *S.next* } S_2
{ *S.code* := *E.code* || *gen*(*E. true* ':') || S_1.*code* ||
gen('*goto*' *S.next*) || *gen*(*E.false* ':') || S_2.*code* }
$S \to$ { S_1.*next* := *newlable* } S_1 ; { S_2.*next* := *S.next* } S_2
{ *S.code* := S_1.*code* || *gen*(S_1.*next* ':') || S_2.*code* }

其中，各属性以及语义函数的含义与讲稿中一致：

id .*place* 表示相应的名字对应的存储位置；int .*val* 表示常量值；*A*生成算术表达式，综合属性*A.place* 表示存放 *A* 的值的存储位置；*E*生成布尔表达式，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 *A.code*，*E.code* 和*S.code* 分别表示相应计算的*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；语义函数 *newtempt* 的作用是在符号表中新建一个从未使用过的名字，并返回该名字的存储位置；|| 是*TAC* 语句序列之间的链接运算。

（和前面一样，假设在语法制导处理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条件表达式的产生式

$$A \to E \,?\, A : A$$

条件表达式 $e_1 ? e_2 : e_3$ 的含义与C语言中的条件表达式类似：布尔表达式e_1的求值结果为真，则取算术表达式e_2的求值结果，否则取算术表达式e_3的求值结果。

试在上述 *L*-翻译模式片段基础上增加针对条件表达式的语义处理内容（不改变 *L*-

翻译模式的特征）。

提示：可考虑引入继承属性*A.next*。

注：设计时如果需要引入其他新的属性或删除旧的属性，请给出相应的解释。