清华大学本科生考试试题专用纸

考试课程 《编译原理》 （A 卷） 2018 年 1 月 9 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.（12分）必做实验相关简答题（所得分数直接加到总评成绩）

1.（3分）PA1（A）实验

本学期实验中，我们新增了框架中没有的若干新的语言特性，其中包括：
（1）支持整复数类型，复数表示形式为a+bj，其中a为实部，bj为虚部，a、b均为整数。
（2）支持switch-case表达式，一般形式如：

```
switch (表达式){
    case 常量1：表达式1;
    case 常量2：表达式2;
    …
    case 常量n：表达式n;
    default：表达式n+1;
}
```

参考语法：
Expr ::= **case** (Expr) {AcaseExpr* DefaultExpr} | …
AcaseExpr ::= Constant : Expr ;
DefaultExpr ::= **default** : Expr;

以下是截取了与上述语言特性相关的部分词法和语法分析的代码片断，，试补全其中（1）、（2）和（3）的内容。

/*lexer.l*/

```
DIGIT                = ([0-9])
INTEGER              = ({DIGIT}+)
IMGCONST             =        （1）          /*识别复数虚部*/
…
"case"       { return keyword(      （2）      ); }
"default"    { return keyword(Parser.DEFAULT); }
…
```

/* Parser.y*/

```
Expr : CaseExpr
        |   Constant     /*常量表达式*/
        | /*  略  */
        ;
CaseExpr : CASE   '(' Expr ')' '{' ACaseExprList DefaultExpr '}'
                  { $$.expr = new Tree.CaseExpr($3.expr, $6.acaselist,
  (ACaseExpr)$7.expr, $1.loc);
                   };
ACaseExprList : ACaseExprList ACaseExpr
```

```
                {
                        $$.acaselist.add((ACaseExpr)$2.expr);
                }
                |    /* empty */
                {
                    $$ = new SemValue();
                     $$.acaselist = new ArrayList<ACaseExpr>();
                };
ACaseExpr :   Constant ':' Expr ';'
                {
               $$.expr = new Tree.ACaseExpr($2.expr, $4.expr, $3.loc);
};
DefaultExpr : _____（3）_____
                {
                    $$.expr = new Tree.ACaseExpr(null, $3.expr, $2.loc);
                };
```

参考解答/评分方案：

（1）({INTEGER}"j")

（2）Parser.CASE

（3）DEFAULT ':' Expr ';'

2.（3分）PA1（B）实验

一个命题逻辑表达式Expr的文法为

Expr ::= Expr *and* Expr | Expr *or* Expr | *not* Expr | *atom*

其中*and*, *or*, *not*, *atom*均为终结符，二元运算符*or*的优先级最低，*and*的优先级比*or*高，一元运算符*not*的优先级最高。运算符*or*与*and*均为左结合。在PA-1B的框架下，我们可以用如下文法来表达（语义动作已经略去）

```
Expr : Expr1 { … }
Expr1 : Expr2 ExprT1 { … }
ExprT1 : Oper1  ____(a)____  ____(b)____  { … }
Expr2 : Expr3 ExprT2 { … }
ExprT2 : Oper2  ____(c)____  ____(d)____  { … }
Expr3 : Oper3 Expr3 | atom { … }
```

那么，(a) 处应填写的非终结符为_____，(d) 处应填写的非终结符为_____，Oper2表示的运算符为_____。

参考解答/评分方案：

Expr2；ExprT2；*and*。

每空 1 分。

完整文法为

```
Expr : Expr1 { … }
Expr1 : Expr2 ExprT1 { … }
```

ExprT1 : *or* Expr2 ExprT1 { … }
Expr2 : Expr3 ExprT2 { … }
ExprT2 : *and* Expr3 ExprT2 { … }
Expr3 : *not* Expr3 | atom { … }

3.（3分）PA2实验

对表达式类型检查时如果出现语义错误则将该表达式的类型值设为Error，Error类型的表达式出现在其他表达式中应不再导致新的语义报错（运算表达式除外，本题不涉及）。新增的super关键字只支持函数调用，不允许在static中调用。以下代码会有四个地方报错，其中一个错误已经指出，请补充剩下三个错误，并标上对应的行号（列号可忽略）。

补充的错误信息请从下面给出的错误信息中选择（若错误信息中包含IDENT和TYPE等，请替换成代码中对应的字符串，TYPE包含int, string, bool, void, complex, student ,ERROR等）：

1）`super.member_var is not supported`
2）`can not use super in static function`
3）`field 'IDENT' not found in 'class : TYPE'`
4）`'IDENT' is not a method in class 'class : TYPE'`
5）`incompatible operands: TYPE = TYPE`

```
 1 class student {
 2         int a;
 3         int BeMan() {
 4                 a = super.age1;
 5         }
 6 }
 7
 8 class Main {
 9     static void main(){
10         int a;
11         complex ca;
12         ca = case(a){
13             0: a+1;
14             3: b;
15             default: 10;};
16
17         a = super.a;
18         class student ab;
19         ab.BeMan();
20     }
21 }
```

请输出对应的错误代码，按行排序。

*** Error at 14: undeclared variable 'b’

参考解答/评分方案：

*** Error at (4,19): super.member_var is not supported
*** Error at (12,6): incompatible operands: complex = int

*** Error at (14,7): undeclared variable 'b'
*** Error at (17,7): can not use super in static function

4.（3分）PA3实验

请在下图右侧横线上补全_Compile的VTABLE以及TAC代码，使它和左侧的Decaf代码相对应。

```
class A {
    int a;
    void setA(int i){
        a = i;
    }
    void print(){
        Print(" a=",a);
    }
}

class B extends A{
    int b;
    void setB(int i, int j){
        a = i;
        b = j;
    }
    void print(){
        Print(" b=",b);
    }
}


class Main {
   static void main() {
    class B b;
    b = new B();
    b.setB(22,23);
   }
}
```

```
VTABLE(_A) {
    <empty>
    A
    _A.setA;
    _A.print;
}
VTABLE(_B) {
    _________
    B
    _________
    _________
    _________
}
VTABLE(_Main) {
    <empty>
    Main
}
FUNCTION(_A_New) {
memo ''
_A_New:
    _T7 = ____
    parm _T7
    _T8 =  call _Alloc
    _T9 = 0
    *(_T8 + 4) = _T9
    _T10 = VTBL <_A>
    *(_T8 + 0) = _T10
    return _T8
} (略去部分函数的 TAC)

FUNCTION(main) {
memo ''
main:
    _T26 =  call _B_New
    _T25 = _T26
    _T27 = 22
    _T28 = 23
    parm _T25
    parm _T27
    parm _T28
    _T29 = *(_T25 + 0)
    _T30 = *(_T29 + ____)
    call _T30
}
```

参考解答/评分方案：

_A
_A.setA ;
_B.print ;
_B.setB ;
8

16

二.（12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（假设占用一个存储单元）。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。过程活动记录中的控制信息包括静态链 SL，动态链 DL，以及返回地址 RA。程序的执行遵循静态作用域规则。

```
（1）  var a0,b0;
（2）  procedure fun1 ;
（3）          var a1,b1;
（4）          procedure fun2 ;
（5）                  var a2;
（6）                  begin
（7）                          a2:= a1*a0-b1;
（8）                          if(a2<0) then call fun3;
   .                           ……     /*不含任何 call 语句和声明语句*/
   .                   end;
   .          begin
                       a1:= a0 - b0;
                       b1:=a0 + b0;
（x）                  If   a1 < a0   then   call fun2 ;
   .                   ……      /*不含任何 call 语句和声明语句*/
   .          end ;
   .  procedure fun3 ;
   .          var a3,b3;
   .          begin
                       a3:=a0*b0 ;
                       b3:=a0/b0 ;
（y）                  if(a3 <> b3) call fun1 ;
 .                     ……     /*不含任何 call 语句和声明语句*/
   .          end ;
   .  begin
   .          a0 := 1;
   .          b0 := 2;
   .          call fun3;
   .          ……     /*不含任何 call 语句和声明语句*/
   .  end .
```

1. （6分）若实现该语言时符号表的组织采用多符号表结构，即每个静态作用域均对应一个符号表。在静态语义分析中，为了体现作用域信息，须要维护一个作用域栈，假设栈中的元素是指向某个作用域的指针。试指出：在分析至语句（7）时，当前作用域栈中包含的作用域的指针有多少个？分别是哪几个作用域？其中，次栈顶所指向的作用域中包含哪些符号？

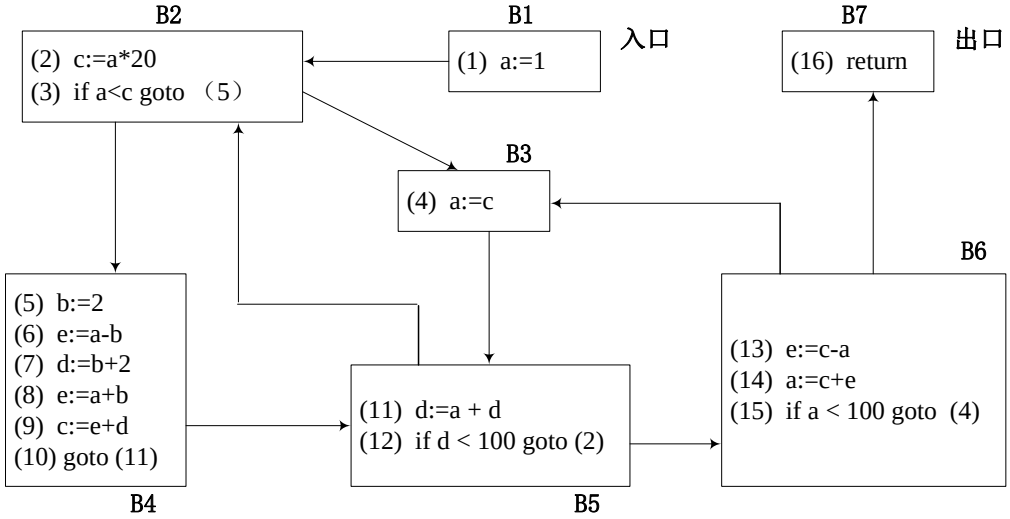
2. （6分）当过程fun1被第二次激活时，运行栈上共有几个活动记录？分别是那些过程的活动记录？当前位于栈顶和次栈顶的活动记录中静态链 SL 和动态链 DL 分别指向什么位置？（注：指出是哪个活动记录的起始位置即可）

参考解答/评分方案：

1. （6分）在分析至语句（7）时，当前作用域栈中包含指向3个作用域的指针（1分）；这3个作用域分别是：过程fun2，过程fun1以及主过程作用域的指针（3分）；次栈顶所指向的作用域是过程fun1声明的作用域，包含的符号有：a1和fun2（2分）。

1. （6分）当过程fun1被第二次激活时，运行栈上共有6个活动记录（1分）；包括 main，fun3,fun1,fun2,fun3, fun1（1分）。 当前位于栈顶的活动记录的静态链 SL指向运行栈的起始位置（1分），即主过程的活动纪录起始位置；动态链 DL指向过程fun3第二个实例所对应的活动记录的起始位置（1分）。当前位于次栈顶的活动记录的静态链 SL指向运行栈的起始位置，即主过程的活动纪录起始位置（1分）；动态链 DL指向过程fun2第一个实例所对应的活动记录的起始位置（1分）。

三 **(13分)** 下图是包含 7 个基本块的流图，其中 B1 为入口基本块，B7 为出口基本块：



1. (3分) 指出在该流图中存在的回边，以及该回边所对应的自然循环（即指出循环中所包含的基本块）。

2. (4分) 已知基本块 B2 和 B6 入口处的活跃变量（live variables）信息分别为

   `LiveIn(B2) = {a,d}` 和 `LiveIn(B6) = {a,c,d}`

   试计算 `LiveIn(B5) =?` 并指出在基本块 B4 内第 (7) 条语句之前处的活跃变量信息。

3. (3分) 已知基本块 B2 出口处的到达-定值（reaching definitions）信息为

   Out(B2) = {1, 2, 4, 5, 8, 11, 13}

   试指出在基本块 B4 内第 (8) 条语句使用变量 a 的 UD 链。

4. (3分) 试从基本块 B4 的 DAG 图，导出一个算术表达式，用来表示结点 c 的计算结果。要求该表达式中的运算数仅包含 DAG 图的叶子结点。（注：基本块入口处活跃变量所对应的叶子结点可表示为a_0, b_0,c_0, …）

参考解答/评分方案：

如下两张表仅供出题和评阅时参考（答题时并非必需）：

	LiveUse	DEF	LiveIn	LiveOut
B1	∅	{a}	{d}	{a,d}
B2	{a}	{c}	{a,d}	{a,c,d}
B3	{c}	{a}	{c,d}	{a,c,d}
B4	{a}	{b,c,d,e}	{a}	{a,c,d}
B5	{a,d}	∅	{a,c,d}	{a,c,d}
B6	{a,c}	{e}	{a,c,d}	{c,d}
B7	∅	∅	∅	∅

	GEN	KILL	IN	OUT
B1	{1}	∅	∅	{1}
B2	{2}	{9}	{1, 2, 4, 5, 8, 9, 11, 13}	{1, 2, 4, 5, 8, 11, 13}
B3	{4}	{1, 14}	{1, 2, 4, 5, 8, 9, 11, 13, 14}	{2, 4, 5, 8, 9, 11, 13}
B4	{5, 7, 8, 9}	{2, 11,13}	{1, 2, 4, 5, 8, 11, 13}	{1, 4, 5, 7, 8, 9}
B5	{11}	{7}	{1, 2, 4, 5, 7, 8, 9, 11, 13}	{1, 2, 4, 5, 8, 9, 11, 13}
B6	{13, 14}	{1,4,8}	{1, 2, 4, 5, 8, 9, 11, 13}	{2, 5, 9, 11, 13, 14}
B7	∅	∅	{2, 5, 9, 11, 13, 14}	{2, 5, 9, 11, 13, 14}

1. (3分) 该流图中存在唯一的回边 B5→B2，该回边所对应的自然循环包含基本块B2，B3，B4，B5和B6。 （答对回边1分；5个基本块全对2分，回答了B3，B4，B6可得1分；多答回边，扣1分）

2. (4分) LiveIn(B5) = {a,c,d}。在基本块 B4 内第 (7) 条语句之前处的活跃变量信息为{a, b}。 （各2分）

3. (3分) 在基本块 B4 内第 (8) 条语句使用变量 a 的 UD 链为{1, 4}。 （3分）

4. (3分) $(a_0+2)+4$ （计算过程等效于$(a_0+2)+4$，3分；相当于a_0+2+4，未指明计算次序，2分；回答a_0+6，给1分）

四 (18 分) 给定某类表达式文法 *G*[*E*]：

$E \rightarrow + E\ R \mid -\ E\ R \mid \underline{positive}\ R$
$R \rightarrow * E\ R \mid \varepsilon$

其中，+ 和 – 分别代表一元正和一元负运算，* 代表普通的二元乘法运算，*positive* 为代表正整数（非0）的单词。

1. (3分) 针对文法 *G*[*E*]，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（共3处）的内容：

G[*E*]的规则 *r*	*First* (*rhs*(*r*))	*Follow* (*lhs*(*r*))	*PS* (*r*)
$E \rightarrow + E\ R$	+		+
$E \rightarrow - E\ R$	-	此处不填	-
$E \rightarrow \underline{positive}\ R$	*positive*	此处不填	*positive*
$R \rightarrow * E\ R$	*		*
$R \rightarrow \varepsilon$	ε	此处不填	

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

2. (2分) *G*[*E*] 不是 *LL*(1) 文法，试解释为什么?

3. (6分) 虽然*G*[*E*] 不是 *LL*(1) 文法，但可以采用一种强制措施，使得常规的 *LL*(1) 分析算法仍然可用。针对含5个单词的输入串 *+ - 20 * 18* ，以下基于这一措施以及上述各产生式的预测集合（或预测分析表）的一个表驱动 *LL*(1) 分析过程：

步骤	下推栈	余留符号串	下一步动作
1	*# E*	*+ - 20 * 18 #*	应用产生式 $E \rightarrow + E\ R$
2	*# R E +*	*+ - 20 * 18 #*	匹配栈顶和当前输入符号
3	*# R E*	*- 20 * 18 #*	应用产生式 $E \rightarrow - E\ R$
4	*# R R E -*	*- 20 * 18 #*	匹配栈顶和当前输入符号
5	*# R R E*	*20 * 18 #*	应用产生式 $E \rightarrow \underline{positive}\ R$
6	*# R R R positive*	*20 * 18 #*	匹配栈顶和当前输入符号
7	*# R R R*	** 18 #*	
8			
9			
10			
11			
12	*# R R R*	*#*	应用产生式 $R \rightarrow \varepsilon$
13	*# R R*	*#*	应用产生式 $R \rightarrow \varepsilon$
14	*# R*	*#*	应用产生式 $R \rightarrow \varepsilon$
15	#	*#*	结束

试填写上述分析过程中第7步时使用的产生式，以及第 8~11 步的分析过程，共计13处空白；并指出采用了什么样的强制措施。

4. (7分) 如下是以*G*[*E*]为基础改造的一个 *L* 翻译模式：

$E \rightarrow + E_1$ { $R.i := E_1.sign$ } R { $E.sign := R.s$ }
$E \rightarrow - E_1$ { $R.i :=$ if $E_1.sign$ = P then N else P } R { $E.sign := R.s$ }
$E \rightarrow \underline{positive}$ { $R.i :=$ P } R { $E.sign := R.s$ }
$R \rightarrow * E$ { $R_1.i := R.i$ } R_1 {$R.s :=$ if $E.sign = R_1.s$ then P else N }
$R \rightarrow \varepsilon$ {$R.s := R.i$}

其中，每个子表达式 *E* 的符号（即值大于零或小于零）记录在属性 *E*.sign 中（属性值分别用 P 或 N 表示，其类型为枚举型 SIGN = enum {P, N}）。

试针对该 *L* 翻译模式构造一个自上而下的递归下降（预测）翻译程序（注：采用了上一小题中所述的强制措施）：

```
SIGN ParseE()                    // 主函数
{
    switch (lookahead)  {                // lookahead 为下一个输入符号
        case '+':
            MatchToken('+');
            e1_sign := ParseE();
            _____①_____ ;
            rs := ParseR(ri);
            e_sign := rs;
            break;
        case '-':
            MatchToken('-');
            e1_sign := ParseE();
            ri := if e1_sign=P then N else P;
            rs := ParseR(ri);
            e_sign := rs;
            break;
        case positive:
            MatchToken(positive);
            rs := ParseR(P);
            _____②_____ ;
            break;
        default:
            printf("syntax error \n")
            exit(0);
    }
    return e_sign;
}

SIGN ParseR(SIGN i)
{
    switch (lookahead)  {                // lookahead 为下一个输入符号
        case  ___③___ :
            MatchToken('*');
            e_sign := ParseE();
            _____④_____ ;
            r1s := ParseR(r1i);
            _____⑤_____ ;
            break;
        case  ___⑥___ :
            _____⑦_____ ;
```

```
                    break;
                default:
                    printf("syntax error \n")
                    exit(0);
            }
            return rs;
        }
```

其中使用了与课程中所给的 MatchToken 函数。

该翻译程序中 ①~⑦ 的部分未给出，试填写之。

参考解答/评分方案：

1. (3分) 共3处，每处1分。

G[*E*]的规则 *r*	*First* (*rhs*(*r*))	*Follow* (*lhs*(*r*))	*PS* (*r*)
$E \rightarrow + E R$	+	# *	+
$E \rightarrow - E R$	-	# *	-
$E \rightarrow \underline{positive}\ R$	*positive*	# *	*positive*
$R \rightarrow * E R$	*	# *	*
$R \rightarrow \varepsilon$	ε	# *	# *

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. (2分) *G*[*E*] 不是 *LL*(1) 文法，试解释为什么?

因为 *PS* ($R \rightarrow * E R$) 与 *PS* ($R \rightarrow \varepsilon$) 相交不为空。

3. (6分)

步骤	下推栈	余留符号串	下一步动作
1	# *E*	+ - *20* * *18* #	应用产生式 $E \rightarrow + E R$
2	# *R E* +	+ - *20* * *18* #	匹配栈顶和当前输入符号
3	# *R E*	- *20* * *18* #	应用产生式 $E \rightarrow - E R$
4	# *R R E* -	- *20* * *18* #	匹配栈顶和当前输入符号
5	# *R R E*	*20* * *18* #	应用产生式 $E \rightarrow \underline{positive}\ R$
6	# *R R R positive*	*20* * *18* #	匹配栈顶和当前输入符号
7	# *R R R*	* *18* #	应用产生式 $R \rightarrow * E R$
8	# *R R R E* *	* *18* #	匹配栈顶和当前输入符号
9	# *R R R E*	*18* #	应用产生式 $E \rightarrow \underline{positive}\ R$
10	# *R R R R positive*	*18* #	匹配栈顶和当前输入符号
11	# *R R R R*	#	应用产生式 $R \rightarrow \varepsilon$

12	# *R R R*	#	应用产生式 $R \rightarrow \varepsilon$
13	# *R R*	#	应用产生式 $R \rightarrow \varepsilon$
14	# *R*	#	应用产生式 $R \rightarrow \varepsilon$
15	#	#	结束

采用的强制措施：如果在下一输入符号为 *（未到结束符 #）时，不使用 $E \rightarrow \varepsilon$，而是使用 $R \rightarrow * E R$。（1 分）

表中的 7~11 行，每行 1 分。

4. （7分） 每空1分

```
void ParseE()                    // 主函数
{
    switch (lookahead)  {                // lookahead 为下一个输入符号
        case ‘+’:
            MatchToken(‘+’);
            e1_sign := ParseE();
            ri := e1_sign;
            rs := ParseR(ri);
            e_sign := rs;
            break;
        case ‘-’:
            MatchToken(‘-’);
            e1_sign := ParseE();
            ri := if e1_sign=P then N else P;
            rs := ParseR(ri);
            e_sign := rs;
            break;
        case positive:
            MatchToken(positive);
            rs := ParseR(P);
            e_sign := rs;
            break;
        default:
            printf("syntax error \n")
            exit(0);
    }
    return e_sign;
}

SIGN ParseR(SIGN i)
{
    switch (lookahead)  {                // lookahead 为下一个输入符号
        case ‘*’:
            MatchToken(‘*’);
            e_sign := ParseE();
            r1i := i;
            r1s := ParseR(r1i);
            rs := if e_sign= r1s then P else N;
            break;
```

```
                case #:
                    rs := i;
                    break;
                default:
                    printf("syntax error \n")
                    exit(0);
            }
            return rs;
        }
```

其中使用了与课程中所给的 MatchToken 函数。

五 (25 分) 设语言 $L = \{ a^n b^m \mid m > n \geq 0 \}$，下列 $G_1[S]$ 和 $G_2[S]$ 是该语言的两个无二义文法（对应两种不同的配对方法）：

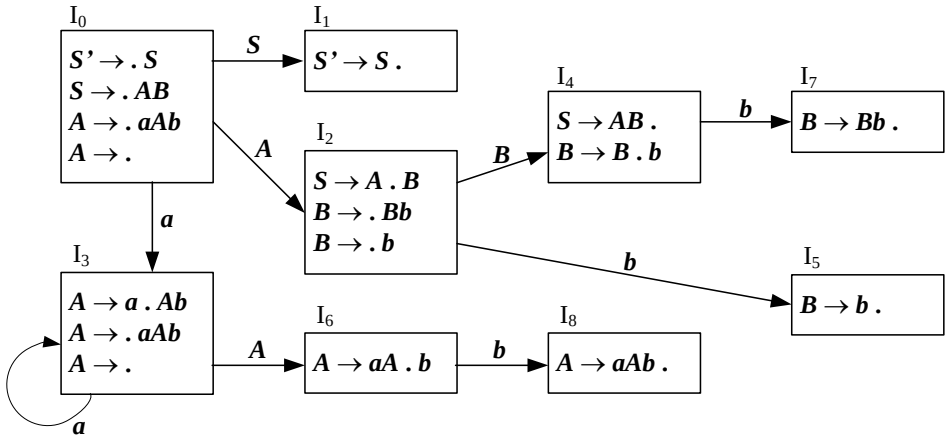
$G_1[S]$ ：

(1) $S \rightarrow AB$
(2) $A \rightarrow aAb$
(3) $A \rightarrow \varepsilon$
(4) $B \rightarrow Bb$
(5) $B \rightarrow b$

$G_2[S]$ ：

(1) $S \rightarrow aSb$
(2) $S \rightarrow B$
(3) $B \rightarrow Bb$
(4) $B \rightarrow b$

1. (4分) 下图是相应于 $G_1[S]$ 的 LR(0) 自动机：



文法 $G_1[S]$ 不是LR(0) 文法。试指出 $G_1[S]$ 的 LR(0) 自动机中存在哪些冲突的状态？并指出这些状态的冲突类别，即是移进-归约冲突还是归约-归约冲突？

2. (4分) 文法 $G_1[S]$ 是 SLR(1) 文法。试解释为什么？

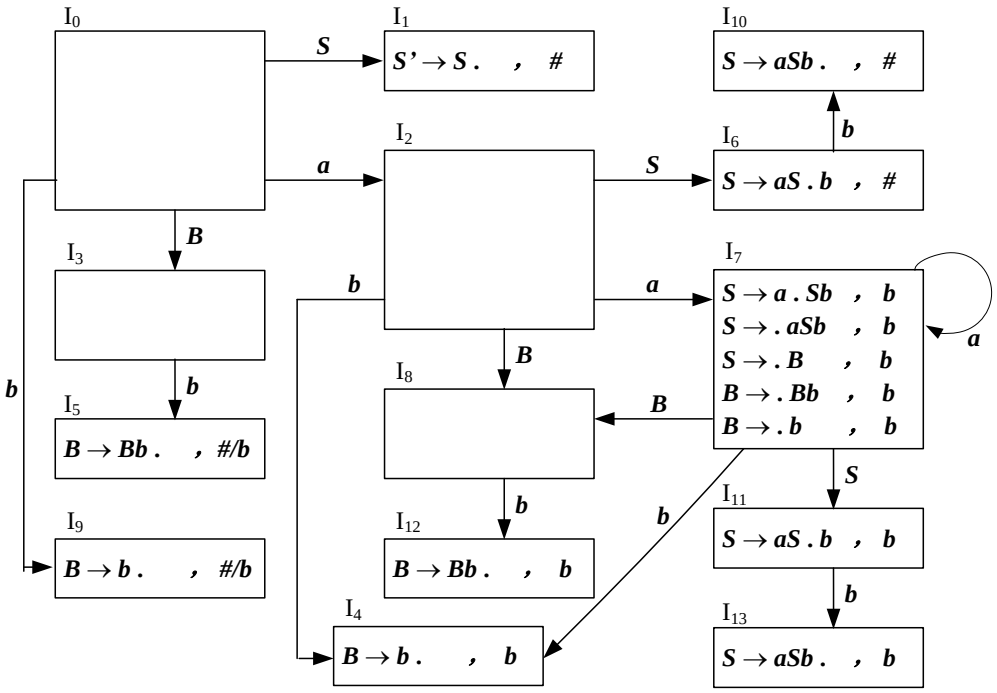
3. (4分) 下图表示 $G_1[S]$ 的 LR(0) 分析表和 SLR(1) 分析表中状态 I_3 和 I_4 两行所对应的内容，上半部分是 LR(0) 分析表，下半部分是 SLR(1) 分析表，但表中的ACTION部分没有给出，试补齐之。

	状态	ACTION			GOTO		
		a	*b*	#	*S*	*A*	*B*
LR(0)分析表	3					6	
	4						
SLR(1)分析表	3					6	
	4						

4. (4分) 在针对 $G_1[S]$ 的SLR(1)分析过程的某个时刻，符号栈的栈顶是 *A*，且栈中包含 *a*，则此时所期待的句柄有哪些？而在另一个时刻，符号栈的栈顶是 *A*，但栈中不包含 *a*，则此刻所期待的句柄有哪些？

5. (2分) 在针对 $G_1[S]$ 的SLR(1)分析过程中，若输入消耗掉相同数目的 *a* 和 *b* 时发生错误，则最不可能的报错信息是 _____？ 请在“末尾少 *a*”，“末尾少 *b*”，“末尾多 *a*”和“末尾多 *b*”中选择。

6. (4分) 下图是相应于 $G_2[S]$ 的 LR(1) 自动机，但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_0，I_2，I_3 和 I_8 对应的项目集。



7. (3分) $G_2[S]$ 是否LR(1)文法？若不是，则指出 $G_2[S]$ 的 LR(1) 自动机中哪些状态有冲突？

参考解答/评分方案：

1. (4分) $G_1[S]$ 的 LR(0) 自动机中存在3个冲突的状态：I_0，I_3 和 I_4 （3分）；均为移进

-归约冲（1分）。

2. (4分) FOLLOW(S)={ #}，FOLLOW(A)={b}。（1分）

   因为 FOLLOW(A) 中不含 a，所以 I_0 和 I_3 的移进-归约冲突可解决（2分）；又因为，FOLLOW(S) 中不含 b，所以 I_4 的移进-归约冲突可解决（1分）。

3. (4分) 完整的分析表如下：

	状态	ACTION			GOTO		
		a	*b*	#	*S*	*A*	*B*
LR(0)分析表	3	s3,r3	r3	r3		6	
	4	r1	s7,r1	r1			
SLR(1)分析表	3	s3	r3			6	
	4		s7	r1			

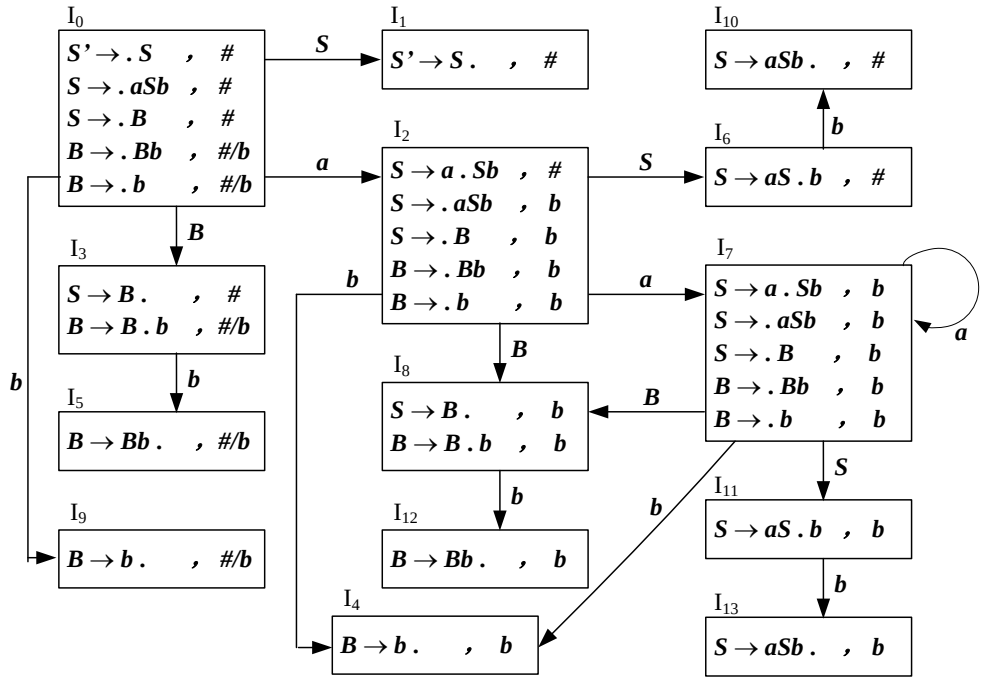
（LR(0) 每行1分，有冲突的位置正确0.5分，其余0.5分）

（SLR(1) 每行1分，两个位置各0.5分，其它位置有填充扣0.5分，无负分）

4. (4分) 在针对 $G_1[S]$ 的SLR(1)分析过程的某个时刻，符号栈的栈顶是 *A*，且栈中包含 *a*，则此时所期待的句柄只有 *aAb*（2分）。而在另一个时刻，符号栈的栈顶是 *A*，但栈中不包含 *a*，则此刻所期待的句柄有3个：*Bb*、*b* 和 *AB*（2分，答案中含 ε，扣一半分。每一问中，多答或少答酌情扣分）。

5. (2分) 在针对 $G_1[S]$ 的SLR(1)分析过程中，若输入消耗掉相同数目的 *a* 和 *b* 时发生错误，则最不可能的报错信息是 _____？ 请在“末尾少 *a*”，“末尾少 *b*”，“末尾多 *a*”和“末尾多 *b*”中选择。 “末尾少 *a*”

6. (4分) 缺4个状态，每个状态1分。



7. (3分)

不是（1分），冲突的状态仅一个：I_8（2分）。

六 (14 分)

给定如下文法 $G[S]$：

（1）$S \to P$
（2）$P \to P\ P\ \wedge$
（3）$P \to P\ P\ \vee$
（4）$P \to P\ \neg$
（5）$P \to \underline{id}$

其中，$\wedge$、$\vee$、$\neg$ 分别代表逻辑与、或、非等运算符单词，$\underline{id}$ 代表标识符单词。如下是以 $G[S]$ 为基础文法的一个 L 翻译模式：

（1）$S \to \{\ P.i := 0\ \}\ P\ \{\ \text{print}\ (P.s)\ \}$
（2）$P \to \{\ P_1.i := P.i\ \}\ P_1\ \{\ P_2.i := P_1.s\ \}\ P_2\ \wedge\ \{\ P.s := P_2.s + 2\ \}$
（3）$P \to \{\ P_1.i := P.i\ \}\ P_1\ \{\ P_2.i := P_1.s\ \}\ P_2\ \vee\ \{\ P.s := P_2.s + 1\}$
（4）$P \to \{\ P_1.i := P.i\ \}\ P_1\ \neg\ \{\ P.s := P.i + P_1.s\ \}$
（5）$P \to \underline{id}\ \{\ P.s := 0\ \}$

1. (3分) 对于合法输入串 $a\ b\ \neg\ \wedge\ a\ b\ c\ \wedge\ \vee\ a\ \neg\ c\ b\ \wedge\ \vee\ \vee\ \vee$，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. (2分) 引入非终结符 M 及其 ε 产生式，并在上述翻译模式的基础上添加如下语义计算规则：

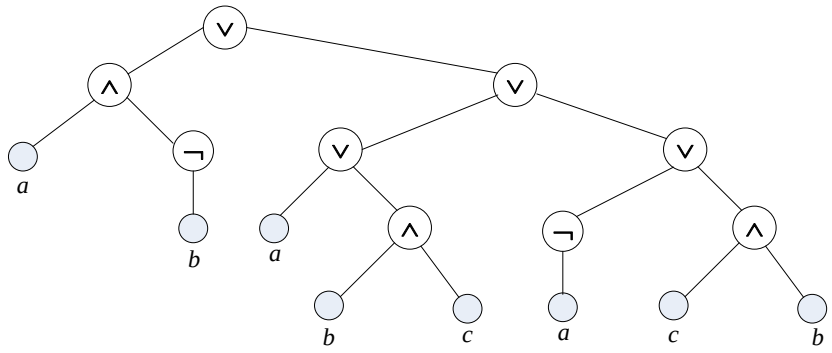
（6）$M \to \varepsilon\quad \{\ M.s := 0\ \}$

试用 M 替代嵌在上述翻译模式产生式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。（如有必要，可增加新的非终结符和相应的产生式）

3. (4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，本题中可用 *val*[*i*].*s* 表示；不用考虑对 *top* 的维护）。

4. (5分) 文法 *G*[*S*] 可用于识别后缀形式（逆波兰式）的命题表达式。输入串 $a\ b\ \neg\ \wedge\ a\ b\ c\ \wedge\ \vee\ a\ \neg\ c\ b\ \wedge\ \vee\ \vee\ \vee$ 对应于中缀式 $a \wedge \neg\, b \vee (\,(\,a \vee b \wedge c\,) \vee (\,\neg\, a \vee (\,c \wedge b\,)\,)\,)$，以下是该命题表达式对应的表达式树：



假设在一个简单的基于寄存器的机器 M 上进行表达式求值，除了load/store指令用于寄存器值的装入和保存外，其余操作均由下列格式的指令完成：

OP　reg0, reg1, reg2

OP　reg0, reg1

其中，reg0, reg1, reg2处可以是任意的寄存器，OP 为运算符。运行这些指令时，对reg1和reg2的值做二元运算，或者对reg1的值做一元运算，结果存入reg0。对于load/store指令，假设其格式为：

LD　reg, mem　　/* 取内存或立即数 mem 的值到寄存器 reg */

ST　reg, mem　　/* 存寄存器 reg 的值到内存量 mem */

我们假设M机器指令中，逻辑运算 ∧、∨、¬ 分别用助记符 AND、OR、NOT 表示。

试说明，为上述表达式树生成机器 M 指令序列时，需要寄存器书目的最小值 n =? 假设这些寄存器分别用助记符R_0，R_1，...，和R_{n-1} 表示，试采用课程中所介绍的方法生成该命题表达式的目标代码（仅含指令AND、OR、NOT、LD和ST，以及仅用寄存器 R_0，R_1，...，和R_{n-1}）。（给出算法执行结果即可，不必进行目标代码优化）

参考解答/评分方案：

1. 对于输入串 $a\ b\ \neg\ \wedge\ a\ b\ c\ \wedge\ \vee\ a\ \neg\ c\ b\ \wedge\ \vee\ \vee\ \vee$，该翻译模式的语义计算结果是10. *3分*

2. (2分)

（1）$S \to M$ { $P.i := M.s$ } P { print ($P.s$) }　　*2分*

（2）$P \to$ { $P_1.i := P.i$ } P_1 { $P_2.i := P_1.s$ } $P_2 \wedge$ { $P.s := P_2.s + 2$ }

（3）$P \to$ { $P_1.i := P.i$ } P_1 { $P_2.i := P_1.s$ } $P_2 \vee$ { $P.s := P_2.s + 1$}

（4）$P \to$ { $P_1.i := P.i$ } $P_1 \neg$ { $P.s := P.i + P_1.s$ }

（5）$P \rightarrow \underline{id}$ { $P.s := 0$ }
（6）$M \rightarrow \varepsilon$ { $M.s := 0$ }

3. (4分)

（1）$S \rightarrow M\ P$	{ print ($val[top].s$) }	*0.5分*
（2）$P \rightarrow P_1\ P_2\ \wedge$	{ $val[top\text{-}2].s := val[top\text{-}1].s + 2$ }	*0.5分*
（3）$P \rightarrow P_1\ P_2\ \vee$	{ $val[top\text{-}2].s := val[top\text{-}1].s + 1$}	*0.5分*
（4）$P \rightarrow P_1\ \neg$	{ $val[top\text{-}1].s := val[top\text{-}2].s + val[top\text{-}1].s$ }	*1分*
（5）$P \rightarrow \underline{id}$	{ $val[top].s := 0$ }	*0.5分*
（6）$M \rightarrow \varepsilon$	{ $val[top+1].s := 0$ }	*1分*

4. (5分)

n =3。 （2分）

假设这些寄存器分别用 R_0，R_1，和R_2 表示，生成该命题表达式的目标代码如下：
（3分，等效的代码，或经过某些优化，均可）

```
LD      R0, b
LD      R1, c
AND     R0, R0, R1
LD      R1, a
OR      R0, R1, R0
LD      R1, c
LD      R2, b
AND     R1, R1, R2
LD      R2, a
NOT     R2, R2
OR      R1, R2, R1
OR      R0, R1, R0
LD      R1, a
LD      R2, b
NOT     R2, R2
AND     R1, R1, R2
OR      R0, R1, R0
```

七 (18分)

1. 以下是一个S-翻译模式片断，描述了某小语言部分特性的类型检查工作：

$P \rightarrow D\ ;\ S$ { $P.type$:= if $D.type = ok$ and $S.type = ok$ then ok else $type\_error$ }
$S \rightarrow S'$ { $S.type := S'.type$ }
$S \rightarrow$ **if** E **then** S_1 { $S.type$:= if $E.type=bool$ then $S_1.type$ else $type\_error$ }
$S \rightarrow$ **while** E **begin** S_1 **end** { $S.type$:= if $E.type=bool$ then $S_1.type$ else $type\_error$ }
$S \rightarrow$ **for** (S'_1; E ; S'_2) **begin** S_1 **end**
{ $S.type$:= if $E.type=bool$ and $S'_1.type = ok$ and $S'_2.type = ok$
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 $type\_error$ }
$S' \rightarrow \underline{id} := E'$ { $S'.type$:= if $lookup\_type$ ($\underline{id}.entry$) = $E'.type$ then ok else $type\_error$ }
$D \rightarrow \ldots$ /*省略与声明语句相关的全部规则*/
$E \rightarrow \ldots$ /*省略与布尔表达式相关的全部规则*/
$E' \rightarrow \ldots$ /*省略与算术表达式相关的全部规则*/

其中，type 属性以及类型表达式 ok，$type\_error$，$bool$,以及所涉及到的语义函数（如

lookup_type）等的含义与讲稿中一致；加黑的单词为保留字；声明语句、布尔表达式以及算术表达式相关的部分已全部略去。

（此外，假设在语法制导语义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，我们不考虑基础文法是否 *LR* 文法。）

下面叙述本小题的要求：

若在基础文法中增加如下产生式，用于定义针对for-循环的“跳过本次循环体”语句：

$S \rightarrow$ **leap**

语句 leap 只能出现在for-循环语句内部，但同时不能直接出现在某个while-循环语句的内部。以下代码片断中，列举了几个合法与不合法使用leap语句的例子：

```
…              /*不含循环语句*/
leap;          /*此处的leap是不合法的*/
…              /*不含循环语句*/
for (i:=1; i<100; i:=i+1)
begin
    …
    leap;      /*此处的leap是合法的*/
    …
    if i=50 then leap;      /*此处的leap是合法的*/
    …
    while cond
    begin
        …
        leap;      /*此处的leap是不合法的*/
        …
        for (…)
        begin
            leap;      /*此处的leap是合法的*/
        end
        …
    end

end
…
```

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 leap 语句的类型检查（合法性检查）任务。

（提示：可以引入 *S* 的一个继承属性）

注：未修改的规则可省略不写。每条规则中，可仅给出与变化内容相关的规则。

参考解答/评分方案：　　*6分　每条规则1分*

仅列出相关的规则：

$P \rightarrow D$; { $S.leap := 0$ } S
　　{ $P.type$:= if $D.type = ok$ and $S.type = ok$ then ok else $type\_error$ }
$S \rightarrow$ **if** E **then** { $S_1.leap := S.leap$ } S_1
　　{ $S.type$:= if $E.type=bool$ then $S_1.type$ else $type\_error$ }
$S \rightarrow$ **while** E **begin** { $S_1.leap := 0$ } S_1 **end**
　　{ $S.type$:= if $E.type=bool$ then $S_1.type$ else $type\_error$ }

$S \rightarrow S \rightarrow$ **for** (S'_1; E ; S'_2) **begin** { $S_1.leap := 1$ } S_1 **end**
{ $S.type$:= if $E.type$=*bool* and $S'_1.type = ok$ and $S'_2.type = ok$
then $S_3.type$ else *type_error* }

$S \rightarrow$ { $S_1.leap := S.leap$ } S_1 ; { $S_2.leap := S.leap$ } S_2
{ $S.type$:= if $S_1.type = ok$ and $S_2.type = ok$ then *ok* else *type_error* }

$S \rightarrow$ **leap** { $S.type$:= if $S.leap = 1$ then *ok* else *type_error* }

注：未修改的规则可省略不写。

2. 以下是一个*L*-翻译模式片断，可以产生相应的 *TAC* 语句序列：

$P \rightarrow D$; { $S.next := newlabel$ } S { $gen(S.next$ ':') }
$S \rightarrow$ { $S'.next := S.next$ } S' { $S.code := S'.code$ }
$S \rightarrow$ **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
begin { $S_1.next := newlabel$ } S_1 **end**
{ $S.code := gen(S_1.next$ ':') || $E.code$|| $gen(E.true$ ':') ||
$S_1.code$ || $gen($'goto' $S_1.next)$ }
$S \rightarrow$ **for** ({ $S'_1.next := newlabel$ } S'_1;
{ $E.true := newlabel$; $E.false := S.next$ } E ;
{ $S'_2.next := S'_1.next$ } S'_2)
begin { $S_1.next := newlabel$ } S_1 **end** {
$S.code := S'_1.code$ || $gen(S'_1.next$ ':') || $E.code$ ||
$gen(E.true$ ':') || $S_1.code$ || $gen(S_1.next$ ':') ||
$S'_2.code$ || $gen($'goto' $S'_1.next)$ }
$S \rightarrow$ { $S_1.next := newlabel$ } S_1 ;
{ $S_2.next := S.next$ } S_2
{ $S.code := S_1.code$ || $gen(S_1.next$ ':') || $S_2.code$ }
$S' \rightarrow$ id := E' { $S'.code := E'.code$ || $gen($id $.place$ ':=' $E'.place)$ }
$D \rightarrow$ … /*省略与声明语句相关的全部规则*/
$E \rightarrow$ … /*省略与布尔表达式相关的全部规则*/
$E' \rightarrow$ … /*省略与算术表达式相关的全部规则*/

其中，属性 $S.code$, $S'.code$, $E.code$, $S.next$, $S'.next$, $E.true$, $E.false$, 语义函数 *newlabel* , *gen( )* 以及所涉及到的 *TAC* 语句与讲稿中一致。语义函数*newtemp* 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；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。所有符号的 *place* 综合属性也均与讲稿中一致。

（此外，和前面一样，假设在语法制导语义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，我们不考虑基础文法是否 *LR* 文法。）

下面叙述本小题的要求：

若在基础文法中增加如下产生式，用于定义针对for-循环的“跳过本次循环体”语句：

$S \rightarrow$ **leap**

如第1小题中所述，语句 leap 只能出现在for-循环语句内部，但同时不能直接出现在某个while-循环语句的内部，其执行语义为：设 for (S'_1; E ; S'_2) begin ... end 是直接包围该 leap 语句的 for-循环，在执行 leap 语句后，控制将跳过该循环体内部剩余的语句，跳转到下一次循环（先执行S'_2）。

试在上述 *L*-翻译模式片段基础上增加针对 leap 语句的语义处理内容（不改变 *L*-翻译模式的特征，不考虑第1小题中已完成的语义检查工作）。

注：可引入新的属性或删除旧的属性，必要时给出解释。未修改的规则可省略不写。

参考解答/评分方案： *6分　每条规则1分*

仅列出相关的规则：

$P \rightarrow D$; { $S.next$:= $newlabel$; $S.leap$:= $newlabel$ } S { $gen(S.next$ ':') }
$S \rightarrow$ { $S'.next$:= $S.next$ } S' { $S.code$:= $S'.code$ }
$S \rightarrow$ **if** { $E.true$:= $newlabel$; $E.false$:= $S.next$ } E **then**
{ $S_1.next$:= $S.next$; $S_1.leap$:= $S.leap$ } S_1
{ $S.code$:= $E.code$ || $gen(E.true$ ':') || $S_1.code$ }
$S \rightarrow$ **while** { $E.true$:= $newlabel$; $E.false$:= $S.next$ } E
begin { $S_1.next$:= $newlabel$; $S_1.leap$:= $newlabel$ } S_1 **end**
{ $S.code$:= $gen(S_1.next$ ':') || $E.code$|| $gen(E.true$ ':') ||
$S_1.code$ || $gen($'goto' $S_1.next)$ }
$S \rightarrow$ **for** ({ $S'_1.next$:= $newlabel$ } S'_1;
{ $E.true$:= $newlabel$; $E.false$:= $S.next$ } E ;
{ $S'_2.next$:= $S'_1.next$ } S'_2)
begin { $S_1.next$:= $newlabel$; $S_1.leap$:= $S_1.next$ } S_1 **end**
{ $S.code$:= $S'_1.code$ || $gen(S'_1.next$ ':') || $E.code$ ||
$gen(E.true$ ':') || $S_1.code$ || $gen(S_1.next$ ':') ||
$S'_2.code$ || $gen($'goto' $S'_1.next)$ }
$S \rightarrow$ { $S_1.next$:= $newlabel$; $S_1.leap$:= $S.leap$ } S_1 ;
{ $S_2.next$:= $S.next$; $S_2.leap$:= $S.leap$ } S_2
{ $S.code$:= $S_1.code$ || $gen(S_1.next$ ':') || $S_2.code$ }
$S \rightarrow$ **leap** { $S.code$:= $gen($'goto' $S.leap)$ }

注：未修改的规则可省略不写。每条规则中，可仅给出与变化内容相关的规则。

3. 以下是一个 *S*-翻译模式片断，可以产生相应的 *TAC* 语句序列：

$P \rightarrow D$; S M { $backpatch(S.nextlist, M.gotostm)$ }
$S \rightarrow S'$ { $S.nextlist$:= "" }
$S \rightarrow$ **if** E **then** M_1 S_1 N **else** M_2 S_2 { $backpatch(E.truelist, M_1.gotostm)$;
$backpatch(E.falselist, M_2.gotostm)$;
$S.nextlist$:= $merge(S_1.nextlist, merge(N.nextlist, S_2.nextlist))$) }
$S \rightarrow$ **while** M_1 E **begin** M_2 S_1 N **end** { $backpatch(S_1.nextlist, M_1.gotostm)$;
$backpatch(E.truelist, M_2.gotostm)$;
$backpatch(N.nextlist, M_1.gotostm)$;
$S.nextlist$:= $E.falselist$; }
$S \rightarrow$ **for** (S'_1; M_1 E ; M_2 S'_2 N_1) **begin** M_3 S_1 N_2 **end**
{ $backpatch(E.truelist, M_3.gotostm)$;
$backpatch(S_1.nextlist, M_2.gotostm)$;
$backpatch(N_1.nextlist, M_1.gotostm)$;
$backpatch(N_2.nextlist, M_2.gotostm)$;
$S.nextlist$:= $E.falselist$ }
$S \rightarrow S_1$; M S_2 { $backpatch(S_1.nextlist, M.gotostm)$;
$S.nextlist$:= $S_2.nextlist$ }
$S' \rightarrow$ id := E' { $emit$ (id $.place$ ':=' $E'.place$) }
$M \rightarrow \varepsilon$ { $M.gotostm$:= $nextstm$ }
$N \rightarrow \varepsilon$ { $N.nextlist$:= $makelist(nextstm)$; $emit($'goto _') }
$D \rightarrow \ldots$ /*省略与声明语句相关的全部规则*/
$E \rightarrow \ldots$ /*省略与布尔表达式相关的全部规则*/
$E' \rightarrow \ldots$ /*省略与算术表达式相关的全部规则*/

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。同第2小题，所有符号的 *place* 综合属性也均与讲稿中一致。

（和前面一样，假设在语法制导语义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如下产生式，用于定义针对for-循环的“跳过本次循环体”语句：

$S \rightarrow$ **leap**

如第1小题中所述，语句 leap 只能出现在for-循环语句内部，但同时不能直接出现在某个while-循环语句的内部，其执行语义为：设 for (S'_1; E ; S'_2) begin ... end 是直接包围该 leap 语句的 for-循环，在执行 leap 语句后，控制将跳过该循环体内部剩余的语句，跳转到下一次循环（先执行S'_2）。

试在上述 *L*-翻译模式片段基础上增加针对 leap 语句的语义处理内容（不改变 *L*-翻译模式的特征，不考虑第1小题中已完成的语义检查工作）。

试在上述 *S*-翻译模式片段基础上增加针对 leap 语句的语义处理内容（不改变 *S*-翻译模式的特征，不考虑第1小题中已完成的语义检查工作）。

注：可引入新的属性或删除旧的属性，必要时给出解释。未修改的规则可省略不写。

参考解答/评分方案：　　*6分*

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

$S \rightarrow S'$　　{ *S.nextlist* := “”; *S.leaplist* := “” }　　(0.5分)

$S \rightarrow$ **if** E **then** $M_1\ S_1\ N$ **else** $M_2\ S_2$　　{ *backpatch*(*E.truelist*, M_1.*gotostm*) ;
backpatch(*E.falselist*, M_2.*gotostm*) ;
S.nextlist := *merge*(S_1.*nextlist*, *merge*(*N.nextlist*, S_2.*nextlist*)) ;
S.leaplist := *merge*(S_1.*leaplist*, S_2.*leaplist*) }　　(1分)

$S \rightarrow$ **while** $M_1\ E$ **begin** $M_2\ S_1\ N$ **end**　　{ *backpatch*(S_1.*nextlist*, M_1.*gotostm*) ;
backpatch(*E.truelist*, M_2.*gotostm*) ;
backpatch(*N.nextlist*, M_1.*gotostm*) ;
S.nextlist := *E.falselist*; *S.leaplist* := “” }　　(1分)

$S \rightarrow$ **for** (S'_1; $M_1\ E$; $M_2\ S'_2\ N_1$) **begin** $M_3\ S_1\ N_2$ **end**
{ *backpatch*(*E.truelist*, M_3.*gotostm*) ;
backpatch(S_1.*nextlist*, M_2.*gotostm*) ;
backpatch(N_1.*nextlist*, M_1.*gotostm*) ;
backpatch(N_2.*nextlist*, M_2.*gotostm*) ;
backpatch(S_1.*leaplist*, M_2.*gotostm*) ;　　(1分)
S.nextlist := *E.falselist* ;
S.leaplist := “” }　　(1分)

$S \rightarrow S_1$; $M\ S_2$　　{ *backpatch*(S_1.*nextlist*, M_1.*gotostm*) ;
S.nextlist := S_2.*nextlist* ;

S.leaplist := *merge*(S_1.*leaplist*, S_2.*leaplist*) }　　(0.5分)

$S \rightarrow$ **leap**　　{ *S.leaplist* := *makelist* (*nextstm*) ; *S. nextlist* := “”;
emit (‘goto _’) }　　(1分)

注：未修改的规则可省略不写。每条规则中，可仅给出与变化内容相关的规则。