清华大学本科生考试试题专用纸

考试课程 《编译原理》 （A 卷） 2019 年 1 月 17 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，其中一项为：支持对象复制语句。形如 scopy(id, E)，scopy 为新增关键字，参考语法：

Stmt ::= OCStmt ; | …
OCStmt ::= **scopy** (identifier, 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*/

```
// 识别关键字的规则
"void"          { return keyword(Parser.VOID);      }
"scopy"         { return keyword(    ( 1 )    );    }
. . .
```

/* Parser.y*/

```
%token IDENTIFIER SCOPY
…

Stmt      :   OCStmt ';'    /* object copy statement *
              | /*略*/
              ;

OCStmt    :   SCOPY '('  (2)  ',' Expr ')'
              {
                  $$.stmt = new SCopy($3.ident,  ( 3 ) , $1.loc);
              }
              ;
```

/*tree.java*/

```
public static class SCopy extends Tree {
        public String dst;
        public Expr src;

        public SCopy(String dst, Expr src, Location loc) {
            super(SCOPY, loc);
            this.dst = dst;
            this.src = src;
```

```
        }

        /*略*/
    }
```

2.（3分）PA1（B）实验

一个类型表达式的文法为

Type ::= Type * Type | Type → Type | (Type) | baseType

其中*，→，(，)，baseType 均为终结符，二元操作符→的优先级最低，*的优先级比→高，( )优先级最高。二元操作符→为右结合，*为左结合。在 PA-1B 的框架下，我们可以用如下文法来表达（部分语义动作已略去）：

```
Type : Type1 { $$.type = $1.type; }
Type1 : Type2 TypeT1
{
    $$.tvec = new Vector<TypeNode>();
    if ($2.tvec != null) {
        $$.tvec.addAll($2.tvec);
    }
    $$.tvec.add($1.type);
    $$.type = $$.tvec.get(0);
    for (int i = 1; i < $$.tvec.size(); i++) {
        $$.type =    a    ;
    }
}
TypeT1 : '→' Type2 TypeT1
{
    $$.tvec = new Vector<TypeNode>();
     b    ;
} | /*empty*/ { }
Type2: Type3 TypeT2 { … }
TypeT2: '*' Type3 TypeT2   { … } | /*empty*/ { }
Type3 : '('   c   ')' { … } | baseType { … }
```

其中，SemValue 的type成员记录了非终结符对应的语法树结点，tvec为辅助向量/数组。

（1）辅助函数

TypeNode MakeArrow(TypeNode *left*, TypeNode *right*)

返回一个语法树结点，表示*left* → *right*。那么，*a*处应填的代码片段是下列哪一项？

A. MakeArrow($$.type, $$.tvec.get(i))

B. MakeArrow($$.tvec.get(i), $$.type)

（2）*b*处应填的代码片段是下列哪一项？

A. $$.tvec.add($2.type); **if** ($3.tvec != **null**) { $$.tvec.addAll($3.tvec); }

B. **if** ($3.tvec != **null**) { $$.tvec.addAll($3.tvec); } $$.tvec.add($2.type)

（3）*c*处应填的非终结符是_____。

3.（3分）PA2实验

在 PA2 中，当检测到错误的时候，程序应该尽可能继续分析，同时需要避免连锁报错。例如在 checkBinaryOP 函数中，**为了能够尽可能继续分析，该函数会依据操作符推测二元操作的返回类型。为了避免连锁报错，当操作数出现ERROR类型时将不会报错。**

在 PA2 中，为了防止var 作为右值出现，我们新增加了基础类型 UNKNOWN，同时定义 UNKNOWN 与任何类型运算都不兼容。

请按照上述原则，填写下列程序片段补充下列错误输出(**注意：请按照上述提示输出而不是现有 decaf 框架输出**)。

```
1  int a;
2  int b;
3  string s;
4  a = var c + b + s;
5  a = s + var d + var f;
```

```
*** Error at (4): incompatible operands: unknown + int
*** Error at (4): incompatible operands: _____ + _____
*** Error at (5): incompatible operands: _____ + _____
*** Error at (5): incompatible operands: _____ + _____
```

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Decaf代码相对应，并写出程序的数据结果。

```
class A {
    int a;
     int last;
    void setA(int i){
         last = a-1;
        a = i;
    }
    void print(){
        Print(" a=",a);
    }
}

class B {
    class A a1;
    class A a2;
     int b;
    void init(){
        a1 = new A();
        a2 = new A();
```

```
VTABLE(_A) {
    <empty>
    A
    _A.setA;
    _A.print;
}

VTABLE(_B) {
    <empty>
    B
    _B.init;
    _B.setB;
    _B.print;
}

VTABLE(_Main) {
    <empty>
    Main
}
```

```
        b = 0;
    }
    void setB(int i, int j, int k){
        a2 = new A();
        a1.setA(i);
        a2.setA(j);
        b = k;
    }
    void print(){
        a1.print();
        a2.print();
        Print(" b=",b);
        Print("\n");
    }
}

class Main {
    static void main() {
        class B a;
        class B b;
        b = new B();
        b.setB(1,1,1);
        scopy(b,a);
        a.setB(2,2,2);
        a.print();
        b.print();
        b.setB(3,3,3);
        a.print();
        b.print();
    }
}
```

```
FUNCTION(_A_New) {
memo ''
_A_New:
    _T9 = _______________
    parm _T9
    _T10 =  call _Alloc
    _T11 = 0
    *(_T10 + 4) = _T11
    *(_T10 + 8) = _T11
    _T12 = _______________
    *(_T10 + 0) = _T12
    return _______________
}

FUNCTION(_B_New) {
memo ''
_B_New:
    _T13 = _______________
    parm _T13
    _T14 =  call _Alloc
    _T15 = 0
    *(_T14 + 4) = _T15
    *(_T14 + 8) = _T15
    *(_T14 + 12) = _T15
……
(下方代码省略)
```

程序输出结果为：

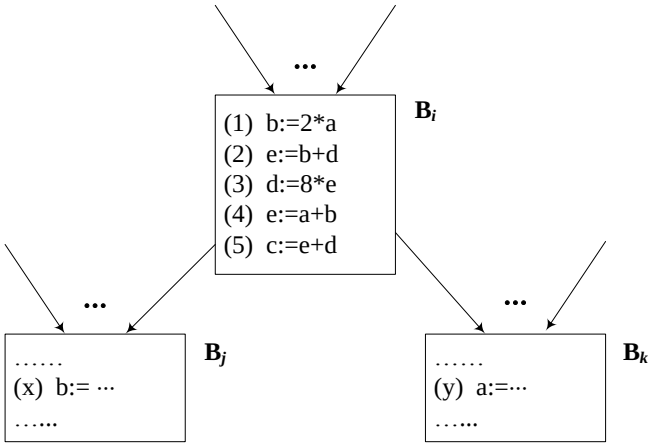
```
a=2 a=2 b=2
_______________
_______________
a=3 a=3 b=3
```

二.（18分）简答题:

1. **(2分)** 按照本课程介绍的消除一般左递归算法消除下面文法$G[S]$中的左递归（要求依非终结符的排序 S、P、Q 执行该算法）：

$$S \to Pa \mid \varepsilon$$
$$P \to Qb \mid \varepsilon$$
$$Q \to Sc \mid \varepsilon$$

2. **(6分)** 下图是某个流图的局部示意，基本块 $\mathbf{B}_i$ 中包含 5 条TAC语句，$\mathbf{B}_i$ 有两个后继基本块 $\mathbf{B}_j$ 和 $\mathbf{B}_k$。



假设：(1) 基本块 $\mathbf{B}_j$ 和 $\mathbf{B}_k$ 入口处的**活跃变量**（live variables）信息分别为 **LiveIn**($\mathbf{B}_j$) = { u, a, d } 和 **LiveIn**($\mathbf{B}_k$) = { b, c, v}；（2）基本块 $\mathbf{B}_i$ 入口处的**到达-定值**（reaching definitions）信息可表示为 **In**($\mathbf{B}_i$) ={ 3, 4, 5, x, y}，其中的各定值点对应上图中的语句标号。

（a）试计算基本块 $\mathbf{B}_i$ 入口处活跃变量信息 **LiveIn** ($\mathbf{B}_i$)=？ (2分)

（b）分别指出在流图范围内，$\mathbf{B}_i$ 内第 (4) 条语句中使用变量 a 和 b 的 UD 链。

（c）分别指出在流图范围内，$\mathbf{B}_i$ 内定值点 (2) 和 (4) 的 DU 链。 (2分)

3. (10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‘:=’，不等号为‘<>’。每一个过程声明对应一个静态作用域（假定采用多遍扫描机制，在静态语义检查之前每个作用域中的所有表项均已生成）。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。程序的执行默认遵循静态作用域规则。

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

```
                    a3 := a0*b0 ;
（y）               if(a2 <> a3) call fun1 ;
.                   ……    /*不含任何 call 语句和声明语句*/
.           end ;
.   begin
.           a0 := 1;
.           b0 := 2;
.           a2 := a0/b0 ;
.           call fun3;
.           ……    /*不含任何 call 语句和声明语句*/
.   end .
```

（a）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。试指出：在分析至语句（x）时，当前开作用域有几个？分别包含哪些符号？在分析至语句（y）时，所访问的a2是在哪行语句声明的？（4分）

（b）当过程fun2被第二次激活时，运行栈上共有几个活动记录？依次是哪些过程的活动记录？当前位于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） （ 4分）

（c）若程序的执行改为遵循动态作用域规则， 当过程fun2被第二次激活时，运行栈上共有几个活动记录？依次是那些过程的活动记录？ （ 2分）

三 **(18 分)** 给定命题表达式文法 $G[S]$：

$$S \to P$$
$$P \to \wedge\ P\ P \mid \vee\ P\ P \mid \neg\ P \mid \underline{id}$$

其中，$\wedge$、$\vee$、$\neg$ 分别代表命题逻辑与、或、非等运算符单词，$\underline{id}$ 代表标识符单词。

1. **(2分)** $G[S]$ 是 $LL(1)$ 文法，为什么?
2. **(5分)** 在 $G[S]$ 中将产生式 $P \to \underline{id}$ 替换为产生式 $P \to \varepsilon$，得到新文法 $G'[S]$。试计算在 $G'[S]$ 中，Follow(S) =？ PS($P \to \varepsilon$) =？并回答 $G'[S]$ 是否 $LL(1)$ 文法？为什么？
3. **(5分)** 基于$G[S]$的预测分析表和一个分析栈，课程中介绍了一种表驱动的 $LL(1)$ 分析过程。假设有输入符号串：$\vee \vee a \wedge b\ c \vee \neg\ a \wedge c\ b\ \#$。试问，在分析过程中，分析栈中最多会出现几个$S$？几个$P$？若因误操作使输入串多了一个符号，变为 $\vee \vee a \wedge b\ c\ c \vee \neg\ a \wedge c\ b$，当分析过程中发生错误时，关于报错信息，你认为最不可能的选择是（4选1）：（1）缺运算数；（2）多运算数；（3）缺运算符；（4）多运算符。
4. **(6分)** 如下是以 $G[S]$ 为基础文法的一个 L 翻译模式：

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

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：

```
void ParseS ( )                    // 主函数
{
```

```
        pi := 0;
        ps := ParseP(pi);
        print (ps);
}

int ParseP ( int i )                    // 主函数
{
    switch (lookahead)  {                    // lookahead 为下一个输入符号
        case '∧':
            MatchToken ('∧');
            p1i := ____①____;
            p1s := ParseP (p1i);
            p2i := p1s;
            p2s := ParseP (p2i);
            ps := p2s + 1;
            break;
        case '∨':
            MatchToken ('∨');
            p1i := ____②____;
            p1s := ParseP (p1i);
            ____③____;
            p2s := ParseP (p2i);
            ____④____;
            break;
        case '¬':
            MatchToken ('¬');
            ____⑤____;
            p1s := ParseP (p1i);
            ____⑥____;
            break;
        case id:
            MatchToken(id);
            ps := 0;
            break;
        default:
            printf("syntax error \n")
            exit(0);
    }
    return ps;
}
```

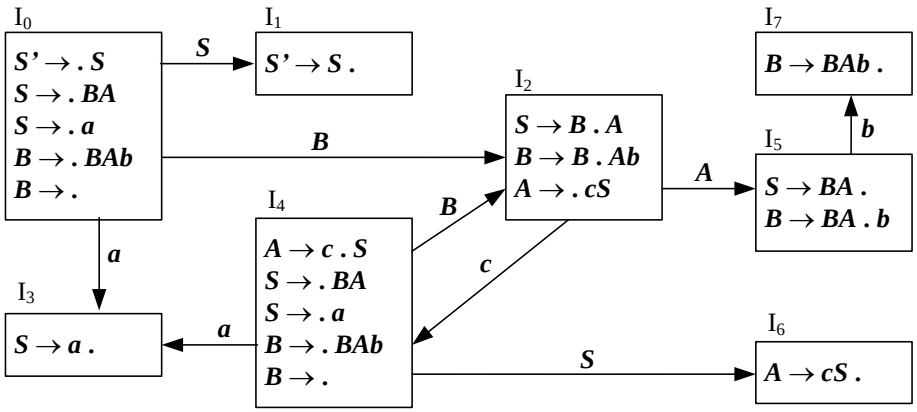
其中使用了与课程中所给的 MatchToken 函数。

该翻译程序中 ①~⑥ 的部分未给出，试填写之。

四 (25 分) 给定下列文法 $G[S]$：

（1）$S \to B\ A$
（2）$S \to a$
（3）$B \to B\ A\ b$
（4）$B \to \varepsilon$
（5）$A \to c\ S$

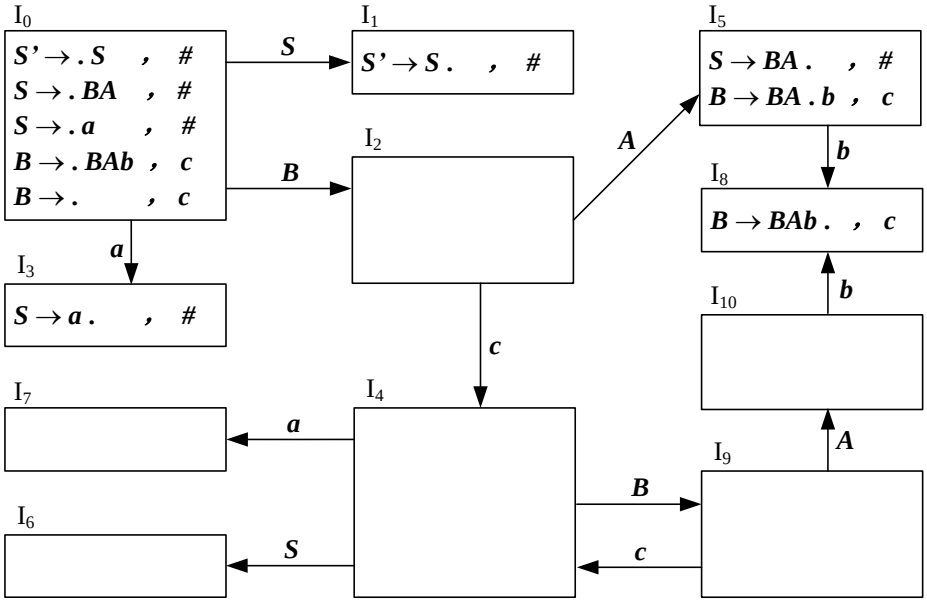
1. **(4分)** 下图是相应于 $G[S]$ 的 LR(0) 自动机：



文法 $G[S]$ 不是LR(0) 文法。试指出 $G[S]$ 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 $G[S]$ 也不是 SLR(1) 文法。试解释为什么？

3. **(4分)** 下图是相应于 $G[S]$ 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_2，I_4，I_6，I_7，I_9和 I_{10} 对应的项目



4. **(2分)** $G[S]$ 是否LR(1)文法？若不是，则指出 $G[S]$ 的 LR(1) 自动机中有冲突的状态。

5. **(4分)** 下图表示 $G[S]$ 的 LR(1) 分析表和 SLR(1) 分析表中有冲突状态的行所对应的内容，上半部分是 LR(1) 分析表，下半部分是 SLR(1) 分析表，但表中的状态号和表项中的内容没有给出，试补齐之。（注意：仅考虑LR(1) 和 SLR(1) 分析表，不考虑 LR(0) 分析表；仅需列出分析表中有冲突的状态所对应的行）

	冲突的状态	ACTION				GOTO		
		a	*b*	*c*	#	*S*	*A*	*B*
LR(1) 分析表								
SLR(1)分析表								

6. **(4分)** 可以通过限定措施来解决 LR(1) 分析表和 SLR(1) 分析表中冲突的状态。请给出你解决冲突的方案，并根据所给方案，分别修改题5中有冲突的表项。

7. **(3分)** 根据题6中修改后新的 LR(1) 分析表或 SLR(1) 分析表，分别进行 LR 分析。对于一个有语法错误的输入符号串，一般哪一种分析方法更快一些？所谓“更快”是指发现错误时经过的步数较少，而每一“步”是指进行一次“移进”或者完成一次“归约”的动作。请给出一个会发生语法错误的输入符号串（长度不超过5），用以支持你的结论。

五 (12 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）$P \to \underline{id}$

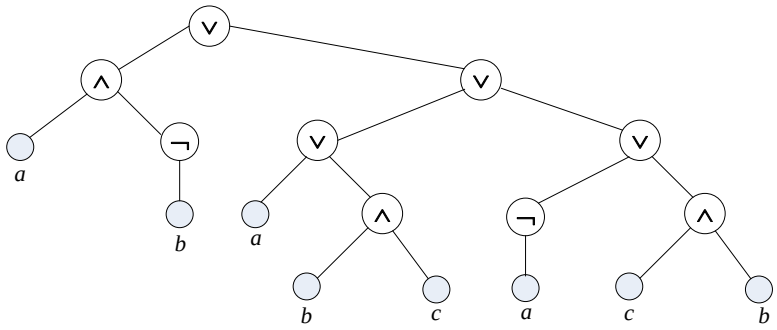
其中，$\wedge$、$\vee$、$\neg$ 分别代表命题逻辑与、或、非等运算符单词，$\underline{id}$ 代表标识符单词。如下是以 $G[S]$ 为基础文法的一个 S 翻译模式：

（1）$S \to P$ { *print* ($P.s$) }
（2）$P \to P_1\ P_2\ \wedge$ { $P.s := f\ (P_1.s, P_2.s)$ }
（3）$P \to P_1\ P_2 \vee$ { $P.s := f\ (P_1.s, P_2.s)$ }
（4）$P \to P_1\ \neg$ { $P.s := g\ (P_1.s)$ }
（5）$P \to \underline{id}$ { $P.s := 1$ }

其中，*print* 为显示函数，f 和 g 为其他语义函数。

1. **(3分)** 如果在 LR 分析过程中根据这一翻译模式进行自下而上语义计算，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）。

2. **(4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：



如果上述 *S* 翻译模式中 *print* (*P.s*) 的含义是显示由 *P* 所识别后缀命题表达式所对应的表达式树根节点对应的Ershov 数（ Ershov number），请给出语义函数 *f* 和 *g* 的具体定义。

3. (5分) 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，为上一小题图中所示的表达式树生成机器 M 指令序列时，需要寄存器数目的最小值 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）

六 (7 分) 以下是一个 *S*-翻译模式片断，可以产生相应的 *TAC* 语句序列：

$S \rightarrow$ if E then $M\ S_1$	{ *backpatch*(*E.truelist, M.gotostm*) ; *S.nextlist* := *merge*(*E.falselist, S*$_1$*.nextlist*) }
$S \rightarrow S_1$; $M\ S_2$	{ *backpatch*(*S*$_1$*.nextlist, M.gotostm*) ; *S.nextlist* := *S*$_2$*.nextlist* }
$S \rightarrow$ id := A	{ *emit* (id *.place* ':=' *A.place*); *S.nextlist* := *''*; }
$A \rightarrow$ id	{ *A.place* := id *.place* }
$A \rightarrow$ int	{ *A.place* := *newtemp*; *emit* (*A.place* ':=' int *.val*) }
$E \rightarrow E_1 \wedge M\ E_2$	{ *backpatch*(*E*$_1$*.truelist, M.gotostm*); *E.falselist* := *merge*(*E*$_1$*.falselist, E*$_2$*.falselist*) ; *E.truelist* := *E*$_2$*.truelist* }
$E \rightarrow (E_1)$	{ *E.truelist* := *E*$_1$*.truelist*; *E.falselist* := *E*$_1$*.falselist* }
$E \rightarrow$ id$_1$ rop id$_2$	{ *E.truelist* := *makelist* (*nextstm*); *E.falselist* := *makelist* (*nextstm*+1); *emit* ('if ' id1.*place* rop.op id2.*place* 'goto _'); *emit* ('goto _') }
$M \rightarrow \varepsilon$	{ *M.gotostm* := *nextstm* }

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。所有符号的 *place* 综合属性也均与讲稿中一致。

（注：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。）

若以语句序列“if *(a<b* ∧ *c<*d *)* then *x*:=2；*y*:=3”为输入，试给出基于这个翻译模式的翻译结果。翻译结果的每条语句按序依次用连续的整数编号（设所产生的第一条语句的编号为0）。

注：这里，我们特别限定：基础文法中，语句复合（“；”）的优先级低于条件语句构造子（“if ...then…”）。

七 (8分)

以下是语法制导生成 *TAC* 语句的一个 *L*-属性文法（同第四次书面作业中的属性文法，建议仅在需要参考某些部分时再去细读）：

S → *if E then* S_1
{ *E .case* := ***false*** ;
E .label:= *S .next* ;
S_1*.next* := *S .next* ;
S .code := *E .code* || S_1 *.code* || *gen(S .next ':')*
}

S → *if E then* S_1 *else* S_2
{ *E .case* := ***false*** ;
E .label:= *newlabel;*
S_1 *.next* := *S .next* ;
S_2 *.next* := *S .next* ;
S .code := *E .code* || S_1 *.code* || *gen('goto'S.next)* || *gen(E .label ':')*
|| S_2 *.code* || *gen(S .next ':')*
}

S → *while E do* S_1
{ *E .case* := ***false*** ;
E .label:= *S .next* ;
S_1 *.next* := *newlabel* ;
S .code := *gen*(S_1 *.next ':')* || *E .code* || S_1 *.code* || *gen('goto'*S_1 *.next)* || *gen(S .next ':')*
}

S → S_1; S_2
{ S_1 *.next* := *newlabel* ;
S_2 *.next* := *S .next* ;
S .code := S_1 *.code* || S_2 *.code*
}

E → E_1 *or* E_2
{ E_2 *.label* := *E .label* ;

E_2 .case := E .case ;
E_1 .case := **true** ;
if E .case {
E_1 .label := E .label;
E .code := E_1 .code || E_2 .code }
else {
E_1 .label := newlabel ;
E .code := E_1 .code || E_2 .code|| gen(E_1 .label ':') }
}

$E \rightarrow E_1$ and E_2
{ E_2 .label := E .label ;
E_2 .case := E .case ;
E_1 .case := **false** ;
if E .case {
E_1 .label := newlabel ;
E .code := E_1 .code || E_2 .code|| gen(E_1 .label ':') }
else {
E_1 .label := E .label;
E .code := E_1 .code || E_2 .code }
}

$E \rightarrow$ not E_1
{ E_1 .label := E .label;
E_1 .case := **not** E .case;
E .code := E_1 .code
}

$E \rightarrow (E_1)$
{ E_1 .label := E .label;
E_1 .case := E .case;
E .code := E_1 .code
}

$E \rightarrow \underline{id_1}$ rop $\underline{id_2}$
{
if E .case {
E .code := gen('if' $\underline{id_1}$.place rop.op $\underline{id_2}$.place 'goto' E .label) }
else {
E .code := gen('if' $\underline{id_1}$.place rop.not-op $\underline{id_2}$.place 'goto' E .label) }
}
// 这里，rop.not-op 是 rop.op 的补运算，例=和≠，< 和 ≥ ，> 和 ≤ 互为补运算

$E \rightarrow$ true
{
if E .case {
E .code := gen('goto' E .label) }
}

$E \rightarrow$ false
{
if **not** E .case {
E .code := gen('goto' E .label) }
}

其中，属性 *S.code, E.code, S.next*，语义函数 *newlabel, gen*，以及所涉及到的 TAC 语句与讲稿中一致，"||"表示 TAC 语句序列的拼接；如下是对属性 *E.case* 和 *E.label* 的简要说明：

E.case：取逻辑值 ***true*** 和 ***false***之一（***not*** 是相应的“非”逻辑运算）

E.label：布尔表达式 *E* 的求值结果为 *E.case* 时，应该转去的语句标号

此外，假设在语法制导处理过程中遇到的二义性问题可以按照某种原则处理（比如规定优先级，else 匹配之前最近的 if，运算的结合性，等等），这里不必考虑基础文法的二义性。

若在基础文法中增加产生式 $E \to \Delta\,(\,E, E\,, E\,)$，试参考上述布尔表达式的处理方法，给出相应的语义处理部分。其中“Δ”代表一个三元逻辑运算符，其语义可用其它逻辑运算定义为 $\Delta\,(\,E_1\,, E_2\,, E_3\,) \equiv E_1$ and E_2 and (not E_3)。

八 **(12分)** 考虑一个简单的栈式虚拟机。该虚拟机维护一个存放整数的栈，并支持如下3条指令：

- `Push` *n*：把整数 *n* 压栈；
- `Plus`：弹出栈顶元素 n_1 和次栈顶元素 n_2，计算 $n_1 + n_2$ 的值，把结果压栈；
- `Minus`：弹出栈顶元素 n_1 和次栈顶元素 n_2，计算 $n_1 - n_2$ 的值，把结果压栈。

一条或多条指令构成一个指令序列。初始状态下，虚拟机的栈为空。

给定一个仅包含加法和减法的算术表达式语言：

$$A \to A + A \mid A - A \mid (\,A\,) \mid \underline{\text{int}}$$

终结符 int 表示一个整数，用 int.val 取得语法符号对应的语义值。

任何一个算术表达式都可以翻译为一个指令序列，使得该虚拟机执行完此指令序列后，栈中仅含一个元素，且它恰好为表达式的值。简单起见，我们用“||”来拼接两个指令序列。例如，算术表达式 1 + 2 – 3可翻译成指令序列

`Push` 3 || `Push` 2 || `Push` 1 || `Plus` || `Minus`

执行完成后，栈顶元素为0。

1. **(4分)** 上述翻译过程可描述成如下S-翻译模式，其中综合属性 *A*.instr表示*A*对应的指令序列：

$A \to A_1 + A_2$ { *A*.instr := ... }
$A \to A_1 - A_2$ { *A*.instr := ... }
$A \to (\,A_1\,)$ { *A*.instr := A_1.instr }
$A \to \underline{\text{int}}$ { *A*.instr := `Push` int.val }

请补全其中两处空缺的部分。

给上述虚拟机新增一个变量表，支持读取和写入变量对应的整数值。新增如下指令：

- `Load` *x*：从表中读取变量 *x* 对应的值并压栈；
- `Store` *x*：把栈顶元素作为变量 *x* 的值写入表，并弹出栈顶元素；
- `Cmp`：若栈顶元素大于或等于 0，则修改栈顶元素为 1；否则，修改栈顶元素为 0；
- `Cond`：若栈顶元素非 0，则弹出栈顶元素；否则，弹出栈顶元素和次栈顶元素后，压入整数 0。

考虑一个仅支持赋值语句的简单语言*L*：

$$S \to \underline{\text{id}} := E \mid S\ ;\ S$$
$$E \to A \mid E\ \underline{\text{if}}\ B$$
$$A \to \ldots \mid \underline{\text{id}}$$
$$B \to A > A \mid B\ \&\ B \mid !B \mid \underline{\text{true}} \mid \underline{\text{false}}$$

终结符 id 表示一个变量，用 id.val 取得语法符号对应的语义值。算术表达式新增id，用来读取变量 id 的值。赋值语句 id := *E* 表示将表达式*E*的值写入变量 id。条件表达式 *E* if *B* 的语义为：若布尔/关系表达式 *B* 求值为真，则该表达式的值为 *E* 的值，否则为 0。布尔/关系表达式中，>为大于，&为逻辑与，!为逻辑非，true为真，false为假。

设 *P* 为 *L* 语言的一个程序，若 *P* 中所有被读取的变量，在读取之前都已经被赋过值，那么称 *P* 为合法程序。任何一个 *L* 语言的合法程序都可以翻译为一个指令序列，使得该虚拟机执行完此指令序列后，对任意程序中出现的变量，表中所存储的值等于程序执行后的实际值。

2. (8分) 上述翻译过程可描述成如下S-翻译模式（与(1)中相同的部分已省略），综合属性 *E*.instr, *B*.instr, *S*.instr 分别表示 *E*, *B*, *S* 对应的指令序列：

$E \to E_1\ \underline{\text{if}}\ B$	{ *E*.instr := ... }
$A \to \underline{\text{id}}$	{ *A*.instr := `Load` id.val }
$B \to A_1 > A_2$	{ *B*.instr := ... }
$B \to B_1\ \&\ B_2$	{ *B*.instr := ... }
$B \to !B_1$	{ *B*.instr := ... }
$B \to \underline{\text{true}}$	{ *B*.instr := `Push` 1 }
$B \to \underline{\text{false}}$	{ *B*.instr := `Push` 0 }
$S \to \underline{\text{id}} := E$	{ *S*.instr := *E*.instr || `Store` id.val }
$S \to S_1\ ;\ S_2$	{ *S*.instr := S_1.instr || S_2.instr }

请补全其中四处空缺的部分。提示：在正确的实现中，任何表达式 *E*, *A*, *B* 翻译成的指令序列必须满足：虚拟机在初始状态下执行完此指令序列后，栈中仅含一个元素，且为表达式的值。