（若发现问题，请及时告知）

第 7 讲书面作业包括两部分。第一部分为 Lecture07.pdf 中课后作业题目中的第 4，5，和 10 题。第二部分为以下题目：

A1　参考 2.3.4 节采用短路代码进行布尔表达式翻译的 *L*-翻译模式片断及所用到的语义函数。若在基础文法中增加产生式 E → E ? E : E，试给出相应该产生式的语义动作集合。其语义可用其它逻辑运算定义为 P ? Q : T ≡ P and Q and (not T)。

参考解答：

$E \rightarrow$ { E_1.false := E.false; E_1.true := *newlabel* ;} E_1 ?
{ E_2. false:= E.false ; E_2. true:= *newlabel* ; } E_2 :
{ E_3.true := E.false ; E_3.false := E.true ; } E_3
{ *E.code := E_1.code* || *gen* (E_1.true ‘:’) || E_2.*code*
|| *gen* (E_2.true ‘:’) || E_3.*code* }

A2　参考2.3.6节采用拉链与代码回填技术进行布尔表达式和控制语句翻译的*S*-翻译模式片断及所用到的语义函数，重复题 A1 的工作。

参考解答：

$E \rightarrow E_1$? M_1 E_2 : M_2 E_3 { *backpatch(E_1.truelist,M_1.gotostm) ;*
backpatch(E_2.truelist,M_2.gotostm) ;
E.truelist := E_3 . falselist ;
E.falselist := merge(merge(E_1.falselist, E_2.falselist), E_3 .
truelist) ;
}

A3 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：

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

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) 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) 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 := ... }
$B \to$ <u>true</u> { B.instr := `Push` 1 }
$B \to$ <u>false</u> { B.instr := `Push` 0 }
$S \to$ <u>id</u> := E { S.instr := E.instr || `Store` <u>id</u>.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。

参考解答

(1)

$A \to A_1 + A_2$ { A.instr := A_2.instr || A_1.instr || `Plus` }
A_1 与 A_2 可交换
$A \to A_1 - A_2$ { A.instr := A_2.instr || A_1.instr || `Minus` }

(2)

题干给出 true、false 规则说明 0 表示假，1 表示真。

$E \to E_1$ <u>if</u> B { E.instr := E_1.instr || B.instr || `Cond` }
$B \to A_1 > A_2$ { B.instr := A_2.instr || A_1.instr || `Minus` || `Cmp` || A_2.instr || A_1.instr || `Minus` || `Cond` }
等价于 ($A_1 - A_2 \geq 0$) if ($A_1 - A_2$!= 0) else 0
$B \to B_1$ & B_2 { B.instr := B_2.instr || B_1.instr || `Cond` }
等价于 B_2 if B_1，B_1 与 B_2 可交换
$B \to !B_1$ { B.instr := `Push` 1 || `Push` 1 || B_1.instr || `Minus` || `Cond` }
等价于 1 if ($B_1 - 1$!= 0) else 0

A4

（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。

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。额外地，我们提醒：*lookup_type* 对于未声明变量，返回 *nil* 而非 *type_error*。

本题中，我们不考虑语法分析的二义性，即可以认为给出的文法是已经语法分析好的抽

象语法。

$P \rightarrow D\ ;\ S$ { *P.type := if D.type = ok and S.type = ok then ok else type_error* }

$S \rightarrow$ ***if*** E ***then*** S_1 ***end*** { *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* }

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

$S \rightarrow id := E$ { *S.type := if lookup_type (id.entry) = E.type then ok else type_error* }

$D \rightarrow$ ... { *省略与声明语句相关的全部规则* }

$E \rightarrow$ ... { *省略与整数/布尔表达式相关的全部规则* }

下面叙述本小题的要求：

在基础文法中增加对“批量赋值”(bulk assignment) 的支持，原来的 *id* **:=** *E* 赋值语句变为批量赋值语句：

id **(,** *id***)*** **:=** *E* **(,** *E***)***

其中 * 表示 Kleene 星闭包（即符号串出现零次或多次）。

批量赋值语句是合法的，当且仅当 := 左侧每个 *id* 均在 *D* 中出现（使用 *lookup_type* 检查），并且右侧每个 *E* 都通过类型检查。而且还要求，:= 左边 *id* 数目和右边 *E* 数目相等，且对应元素类型（*id* 的声明类型和 *E* 的类型）相同。

试在上述 S-翻译模式片段的基础上，新增对批量赋值语句进行类型检查的片段，填写以下空缺的部分。（提示：需要增加一个综合属性）

$L \rightarrow id$ { ① }

$L \rightarrow id\ ,\ L_1$ { ② }

$R \rightarrow E$ { ③ }

$R \rightarrow E\ ,\ R_1$ { ④ }

$S \rightarrow$ ⑤ { ⑥ }

除了课程所讲以及上文例子中有的属性动作 / 语义函数外，你还可以在属性动作中使用如下操作：

- *[x]* 初始化一个只有 *x* 一个元素的列表，类似讲义中的 *makelist*。
- 加号表示列表拼接。例如 *[x] + [z, w]* 的结果是 *[x, z, w]*，类似讲义中的 *merge*。
- *list1 = list2* 判断列表相等。仅当长度相等，且对应位置的元素都相等的时候，两个列表才相等。
- 列表的长度使用 *len(list)* 获得。
- 列表的第 *i* 个元素使用 *list[i]* 获得，注意：列表下标从 0 开始。

请不要使用其他操作。

(2) 以下是一个 L-翻译模式片断，可以对原语言的程序产生相应的 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 ***end***

{ $S.code := E.code$ || $gen(E.true$ ":") || $S_1.code$ }

$S \rightarrow$ ***while*** { $E.true := newlabel$; $E.false := S.next$ } E ***begin***

{ $S_1.next := newlabel$ } S_1 ***end***

{ $S.code := gen(S1.next$ ":") || $E.code$ ||

$gen(E.true$ ":") || $S_1.code$ || $gen($"goto" $S_1.next)$ }

$S \rightarrow$ { $S_1.next := newlabel$ } S_1 ***;***

{ $S_2.next := S.next$ } S_2

{ $S.code := S_1.code$ || $gen(S_1.next$ ":") || $S_2.code$ }

$S \rightarrow id$ ***:=*** 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* }

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。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。

下面叙述本小题的要求：

考虑增加批量赋值的 TAC 生成功能。我们要求，批量赋值的几个赋值是一次性完成的。例如 *a, b := b, a* 可以交换 *a* 和 *b* 的值。

试在上述 L-翻译模式片段的基础上，增加针对批量赋值的 TAC 代码生成片段，填写以下空缺的部分。要求不改变 L-翻译模式的特征。假设输入已经通过第 1 小题的类型检查。

$L \rightarrow id$ { ① }

$L \rightarrow id$, L_1 { ② }

$R \rightarrow E$ { ③ }

$R \rightarrow E$, R_1 { ④ }

S → ⑤ { ⑥ }

除了第 1 小题中允许的操作外，你可以在属性动作中：

- 使用形如 “*for i = A to B do 循环体 end*” 的循环，其中 *i* 的值可以取到循环上下界 *A, B*。如果 *B < A* 那么循环等于空语句（但 *B=A* 的时候不是）。
- 用 "" 表示空 TAC 串。

请不要使用其他操作。

(3) 我们不妨允许批量赋值语句中，:= 左侧的变量个数比右侧的表达式个数多，如 *a, b, c := 1*。对于多出来的变量（称之为“多余变量”），我们按照它们的类型是 *bool* 还是 *int* 给一个默认值。

因此我们加入新的保留字 ***default***，并且加入语法 $S \rightarrow$ ***begin*** S_1 ***end default*** E_1, E_2 表示 S_1 中所有的“多余变量”，如果是 *bool* 类型，那么默认值是 E_1，如果是 *int* 类型，那么默认值是 E_2；假设没有其他类型。E_1, E_2 中可以有变量，但它们的值在 ***begin*** 的时候就被求得（按照先 E_1 后 E_2 的顺序），无论 ***begin*** ... ***end*** 中对 E_1, E_2 使用的变量如何修改，E_1 和 E_2 的值都不变。

例如，假设变量 x，y，z 均为 int 类型，而变量 a，b，c 均为 bool 类型。下面代码

begin *b* ***:= true ;*** *x, y, a* ***:=*** *1* ***end default*** *a && b, 0*

其中有两个多余变量：*y* 和 *a*。这段代码等价为

e1 ***:=*** *a && b* ***;*** *e2* ***:=*** *0* ***;*** *b* ***:= true ;*** *x, y, a* ***:=*** *1, e2, e1*

提示：因为多余变量的值要到后来看到 ***default*** 才知道，所以在批量赋值时我们还不知道它们的默认值。因此我们需要采用“代码回填”技术，将额外变量的 *place* 先留空，等看到 ***default*** 中再填空。使用 *gen(id.place “ := _”)* 表示（注意是 *gen* 而非 *emit*）：*id* 是额外变量，但它的 *place* (即 “_”) 还不确定，具体是什么由在后面看到 ***default*** 的时候回填。使用 *backpatch(list，place)* 表示回填的过程。其中 *list* 列表包含一系列如上待回填的 *“ := _”*，而 *place* 回填的值。例如，*backpatch([id.place]，E.place)* 将会回填 *gen(id.place “ := _”)* 中的空白，将它变成 *gen(id.place “ := ” E.place)*。

下面叙述本小题的要求：

增加针对批量赋值默认值的 TAC 代码生成片段，填写以下空缺的部分。要求不改变 L-翻译模式的特征。假设输入已经通过第 1 小题的类型检查，且保证 E_1 类型是 *bool*，E_2 类型是 *int*。

你可以使用前面两个小题中对 *L* 和 *R* 定义的所有属性。本题答案可能会和第 2 小题答案重复，为方便，请将所有需要“在此处复制第 2 小题的对应的那个空”之处用“**…**”表示。

P → *D* ***;*** { *S.next := newlabel ; S.eint := []; S.ebool := []* }

S { *gen(S.next “:”)* }

$S \rightarrow$ ***if*** { *E.true := newlabel ; E.false := S.next* } *E* ***then***

{ S_1*.next := S.next* } S_1 ***end***

{ *S.code := E.code || gen(E.true ":") ||* S_1*.code ;*

S.eint := S_1*.eint ; S.ebool :=* S_1*.ebool* }

$S \rightarrow$ ***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.eint := S_1*.eint ; S.ebool :=* S_1*.ebool* }

$S \rightarrow$ { 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.eint := S_1*.eint +* S_2*.eint ; S.ebool :=* S_1*.ebool +* S_2*.ebool* }

$L \rightarrow$ *id* { **…** }

$L \rightarrow$ *id,* L_1 { **…** }

$R \rightarrow E$ { **…** }

$R \rightarrow E,$ R_1 { **…** }

$S \rightarrow$ **…** { ①　}

$S \rightarrow$ ***begin*** S_1 ***end default*** E_1, E_2 { ②　}

$D \rightarrow$ ... { *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* }

参考答案：

(1)

... → ... {...}

L → id { L.types := [lookup_type(id.entry)] }

L → id , L1 { L.types := [lookup_type(id.entry)] + L1.types }

R → E { R.types := E.type }

R → E , R1 { R.types := [E.type] + R1.types }

S → L := R { S.type := if L.types = R.types then ok else type_error }

(2)

... → ... {...}

L → id { L.places := [id.place] }

L → id , L1 { L.places := [id.place] + L1.places }

R → E { R.places := [E.place]; R.codes := [E.code] }

R → E , R1 { R.places := [E.place] + R1.places; R.codes := [E.code] + R1.codes }

S → L := R { S.code := "" ;

for i = 0 to len(R.codes)-1 do

S.code := S.code || R.codes[i] end ;

for i = 0 to len(R.places)-1 do

S.code := S.code || gen(L.places[i] ":=" R.places[i]) end }

(3)

S → ... { ... ;

for i = len(R.places) to len(L.places)-1 do

S.code := S.code || gen(L.places[i] ":= _" ;

if L.types[i] = bool then S.ebool := [L.places[i]] + S.ebool

else S.eint := [L.places[i]] + S.eint end; }

S → begin S1 end default E1, E2 { S.code := E1.code || E2.code || S1.code; backpatch(S1.ebool, E1.place); backpatch(S1.eint, E2.place); }

以下是 Lecture07 文档中的题目

---

4. 参考 2.3.5 节进行控制语句（不含 break）翻译的 *L*-翻译模式片断及所用到的语义函数。设在该翻译模式基础上增加下列两条产生式：

$S \to \{ S'.next := S.next \}\ S'\ \{ S.code := S'.code \}$

$S' \to id := E'\ \{ S.code := E.code \| gen(\underline{id}.place\ ':='\ E.place) \}$

若在基础文法中增加对应 for-循环语句的产生式 $S \to for\ (S'; E; S')\ S$，试给出相应该产生式的语义动作集合。

注：for-循环语句的控制语义类似 C 语言中的for-循环语句。

参考解答：

$S \rightarrow$ *for* ({ S'_1 . *next* := *newlabel* } S'_1 ;
{ *E* . *true* := *newlabel* ; *E* . *false* := *S* .*next* } *E* ;
{ S'_2. *next* := S'_1 . *next* } S'_2)
{ S_1 . *next* := *newlabel* } S_1
{ *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*)
}

5. 参考 2.3.6 节采用拉链与代码回填技术进行布尔表达式和控制语句（不含 break）翻译的 *S*-翻译模式片断及所用到的语义函数。设在该翻译模式基础上增加下列两条产生式及相应的语义动作集合：

$S \rightarrow S'$ { *S.nextlist* := *S'.nextlist* }
$S' \rightarrow$ *id* := *E'* { *S'.nextlist* := ''; *emit* (*id* .*place* ':=' *E'* . *place*) }

其中，*E'* 是生成算术表达式的非终结符（对应 2.3.1 中的 *E*）。若在基础文法中增加对应 for-循环语句的产生式 $S \rightarrow$ *for* (*S'*; *E* ; *S'*) *S*，试给出相应该产生式的语义动作集合。

注：for-循环语句的控制语义类似 C 语言中的for-循环语句。

参考解答：

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

10. 以下是语法制导生成 *TAC* 语句的一个 *L*-属性文法：

$S \rightarrow$ *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* ':')
}

$S \rightarrow$ *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 → 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 → 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。）

（*a*）若在基础文法中增加产生式 $E \rightarrow E \uparrow E$，其中"↑"代表"与非"逻辑运算符，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。

注："与非"逻辑运算的语义可用其它逻辑运算定义为 $P \uparrow Q \equiv \text{not}\ (P \text{ and } Q)$

（*b*）若在基础文法中增加产生式 $S \rightarrow$ repeat S until E，试参考上述控制语句的处理方法，

给出相应的的语义处理部分。

注：repeat <循环体> until <布尔表达式> 至少执行<循环体>一次，直到<布尔表达式>成真时结束循环

参考解答：

（*a*）

$E \rightarrow E_1 \uparrow E_2$

{ E_2 .*label* := E .*label* ;
  E_2 .*case* := ***not*** E .*case* ;
  E_1 .*case* := ***false*** ;
  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* ':') }
}

（*b*）

$S \rightarrow$ repeat S_1 until E

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