（若发现问题，请及时告知）

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，试给出相应该产生式的语义动作集合。其中，“↑”代表“与非”逻辑算符，其语义可用其它逻辑运算定义为 P ↑ Q ≡ not (P and Q)。

参考解答：

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

A2 参考2.3.5节进行语句翻译的*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 S → repeat S until E，试给出相应该产生式的语义动作集合。

注：控制语句 repeat <循环体> until <布尔表达式> 的语义为：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：

$S \rightarrow$ repeat { S_1.next := *newlabel* ; } S_1 until
{ E. true := S.next ; E. false := *newlabel* ; } E
{ S.code := gen(E. false ‘:’) || S_1.code || gen(S_1.next ‘:’) || E.code
}

A3 设有开关语句

switch A of
case d_1 : S_1 ;
case d_2 : S_2 ;
.....
case d_n : S_n ;
default S_{n+1}
end

假设其具有如下执行语义：

(1) 对算术表达式 A 进行求值；

(2) 若 A 的取值为 d_1，则执行 S_1，转 (3)；

否则，若 A 的取值为 d_2，则执行 S_2，转 (3)；

.....

否则，若 A 的取值为 d_n，则执行 S_n，转 (3)；

否则，执行 S_{n+1}，转 (3)；

(3) 结束该开关语句的执行。

若在基础文法中增加关于开关语句的下列产生式

$S \rightarrow switch\ A\ of\ L\ end$
$L \rightarrow case\ V : S\ ;\ L$
$L \rightarrow default\ S$
$V \rightarrow d$

其中，终结符 d 代表常量，其属性值可由词法分析得到，以 $d.lexval$ 表示。A 是生成算术表达式的非终结符（对应2.3.1中的 E）。

试参考 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，给出相应的语义处理部分（不改变 L-翻译模式的特征）。

注：可设计增加新的属性，必要时给出解释。

参考解答：

$S \rightarrow switch\ A\ of\ \{\ L.a := A.place\ ;\ L.next := S.next\ \}\ L\ end$
$\{\ S.code := A.code \ ||\ L.code\ \}$

$L \rightarrow case\ \{\ V.failed := newlabel\ \}\ V : \{\ S.next := L.next\ \}\ S\ ;$

$\{\ L_1.a := L.a\ ;\ L_1.next := L.next\ \}\quad L_1$
$\{\ L.code := gen\ (\text{'if'}\ L.a\ \text{'!='}\ V.value\ \text{'goto'}\ V.failed\ )$
$\quad || S.code \ ||\ gen\ (\text{'goto'}\ L.next\ ) \ ||\ gen(V.failed\ \text{':'}) \ ||\ L_1.code\ \}$

$L \rightarrow default\ \{\ S.next := L.next\ \}\ S$
$\{\ L.code := S.code\ \}$

$V \rightarrow d\ \{\ V.value := d.lexval\ ;\}$

A4 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 $\underline{\text{int}}$ 表示一个整数，用 $\underline{\text{int}}$.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：

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

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

$E \rightarrow E_1$ if B { E.instr := E_1.instr || B.instr || `Cond` }
$B \rightarrow A_1 > A_2$ { B.instr := A_2.instr || A_1.instr || `Minus` || `Cmp` || A_2.instr || A_1.instr || `Minus` || `Cond` }
等价于 ($A_1 - A_2 \geqslant 0$) if ($A_1 - A_2$!= 0) else 0
$B \rightarrow B_1$ & 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
$B \rightarrow !B_1$ { B.instr := `Push` 1 || `Push` 1 || B_1.instr || `Minus` || `Cond` }
等价于 1 if ($B_1 - 1$!= 0) else 0

A5 以下是语法制导生成 *TAC* 语句的一个 *L*-属性文法：

S → *if E then* S_1
{ *E .case := **false** ;*
E .label:= S .next ;
S_1*.next := S .next ;*
S_1 *.loop := newlabel ;*
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_1 .loop := newlabel ;
S_2 .loop := newlabel ;
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_1 .loop := newlabel ;
S .code := gen(S_1 . loop ':') || E .code || S_1 .code || gen('goto' S_1 . loop) || gen(S .next ':')
}

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

$E \rightarrow$ 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。）

（*a*）若在基础文法中增加产生式 $E \to E \uparrow E$，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 → 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*）

$E \to E_1 \uparrow E_2$

{ 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*）

$S \to$ repeat S_1 until *E*

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

（注：其中的 *gen*(S_1 .*loop* ‘:’) 其实未起作用，也可以去掉）

A6

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：

$P \to D$; S { *P.type* := if *D.type* = *ok* and *S.type* = *ok* then *ok* else *type_error* }
$S \to S'$ { *S.type* := *S'.type* }
$S \to$ **if** *E* **then** S_1 { *S.type* := if *E.type=bool* then S_1.*type* else *type_error* }
$S \to$ **while** *E* **begin** S_1 **end** { *S.type* := if *E.type=bool* then S_1.*type* else *type_error* }
$S \to$ **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* }

$S' \rightarrow$ id := E' { *S'.type* := if *lookup_type* (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:

仅列出相关的规则：

$P \to D$; { $S.leap := 0$ } S
{ $P.type$:= if $D.type = ok$ and $S.type = ok$ then ok else $type\_error$ }

$S \to$ **if** E **then** { $S_1.leap := S.leap$ } S_1
{ $S.type$:= if $E.type=bool$ then $S_1.type$ else $type\_error$ }

$S \to$ **while** E **begin** { $S_1.leap := 0$ } S_1 **end**
{ $S.type$:= if $E.type=bool$ then $S_1.type$ else $type\_error$ }

$S \to S \to$ **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 $type\_error$ }

$S \to$ { $S_1.leap := S.leap$ } S_1 ; { $S_2.leap := S.leap$ } S_2
{ $S.type$:= if $S_1.type = ok$ and $S_2.type = ok$ then ok else $type\_error$ }

$S \to$ **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：

$P \to D$; { $S.next := newlabel$ } S { $gen(S.next$ ':' $)$ }

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

$S \to$ **if** { $E.true := newlabel$; $E.false := S.next$ } E **then**
{ $S_1.next := S.next$ } S_1 { $S.code := E.code$ || $gen(E.\ true$ ':' $)$ || $S_1.code$ }

$S \to$ **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' $S_1.next)$ }

$S \to$ **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' $S'_1.next)$ }

$S \to$ { $S_1.next := newlabel$ } S_1 ;
{ $S_2.next := S.next$ } S_2

{ *S.code* := S_1.*code* || *gen*(S_1.*next* ':') || S_2.*code* }

S' → id := E' { S'.*code* := E'.*code* || *gen*(id .*place* ':=' E'.*place*) }

D → … /*省略与声明语句相关的全部规则*/

E → … /*省略与布尔表达式相关的全部规则*/

E' → … /*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 → **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:

仅列出相关的规则：

P → D ; { *S.next* := *newlabel*; *S.leap* := *newlabel* } S { *gen*(*S.next* ':') }

S → { S'.*next* := *S.next* } S' { *S.code* := S'.*code* }

S → **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 → **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 → **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。