（若发现问题，请及时告知 ）

1　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 $E \rightarrow E \uparrow E$，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 $P \uparrow Q \equiv \text{not}(P \text{ and } Q)$。

参考解答：

$E \rightarrow$ { E_1.false := E.true ; E_1.true := *newlabel* ; } $E_1 \uparrow$
{ E_2.false := E.true ; E_2.true := E.false ; } E_2
{ *E.code := E*$_1$*.code* || *gen* (*E*$_1$*.true* ‘:’) || *E*$_2$*.code* }

2　参考 2.3.5 节进行控制语句翻译的 *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 $S \rightarrow$ 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
}

3　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 1 和题 2 的工作。

参考解答：

$E \rightarrow E_1 \uparrow M\ E_2$　{ *backpatch(E*$_1$*.truelist,M.gotostm) ;*
E.truelist := merge(E$_1$*.falselist, E*$_2$*.falselist) ;*
E.falselist := E$_2$*.truelist*
}

$S \rightarrow$ repeat $M_1\ S_1$ until $M_2\ E$
{ *backpatch(S*$_1$*.nextlist, M*$_2$*.gotostm) ;*
backpatch(E.falselist, M$_1$*.gotostm) ;*
S.nextlist := E.truelist;
}

6. 参考 2.3.6 节采用拉链与代码回填技术进行布尔表达式翻译的 *S*-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

$E \rightarrow \Delta\ (\ E, E\ ,\ E\ )$

其中“Δ”代表一个三元逻辑运算符。逻辑运算 $\Delta\ (\ E_1\ ,\ E_2\ ,\ E_3)$ 的语义可由下表定义：

E_1	E_2	E_3	$\Delta\ (\ E_1\ ,\ E_2\ ,\ E_3\ )$
false	false	false	false
false	false	true	false
false	true	false	true
false	true	true	false
true	false	false	false
true	false	true	false
true	true	false	false
true	true	true	false

试给出相应该产生式的语义处理部分（必要时增加文法符号，类似上述属性文法中的符号 *M* 和 *N*），不改变 S-属性文法（翻译模式）的特征。

参考解答：

$E \rightarrow \Delta\ (\ E_1\ ,\ M_1\ E_2\ ,\ M_2\ E_3)$

{ *backpatch*(E_1 . *falselist*, M_1 .*gotostm*) ;

backpatch(E_2 . *truelist*, M_2 .*gotostm*) ;

E.truelist := E_3 . *falselist* ;

E.falselist :=*merge*(*merge*(E_1.*truelist*, E_2.*falselist*), E_3 . *truelist* }

8. 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：

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

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

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

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

12. （a）以下是与语句及过程声明相关的类型检查的一个翻译模式片断：

$P \rightarrow D\ ;\ S$ { $P.type :=$ if $D.type = ok$ and $S.type = ok$ then ok else $type\_error$ }
$S \rightarrow$ if E then S_1 { $S.type :=$ if $E.type = bool$ then $S_1.type$ else $type\_error$ }
$S \rightarrow$ if E then S_1 else S_2 { $S.type :=$ if $E.type = bool$ and $S_1.type = ok$ and $S_2.type = ok$

then *ok* else *type_error* }

$S \rightarrow$ while E then S_1 { *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* and S_2.*type* = *ok* then *ok* else *type_error* }

$F \rightarrow F_1$; id (V) S { *addtype*(id.*entry*, *fun* (*V.type*));
F.type := if F_1.*type* = *ok* and *S.type* = *ok* then *ok* else *type_error* }

其中，type 属性以及类型表达式 *ok*，*type_error*，*bool* 等的含义与1.2.2中一致。

若在基础文法中增加关于 *continue* 语句的产生式

$S \rightarrow$ *continue*

continue 语句只能出现在某个循环语句内，即至少有一个包围它的 while 语句。

试在该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 *continue* 语句的这一类型检查任务。（提示：可以引入 *S* 的一个继承属性）

参考解答：

$P \rightarrow D$; { *S. continue* := 0 } S
{ *P.type* := if *D.type* = *ok* and *S.type* = *ok* then *ok* else *type_error* }

$S \rightarrow$ if E then { S_1.*continue* := *S.continue* } S_1
{ *S.type* := if *E.type=bool* then S_1.*type* else *type_error* }

$S \rightarrow$ if E then { S_1.*continue* := *S.continue* } S_1 else { S_2.*continue* := *S.continue* } S_2
{ *S.type* := if *E.type=bool* and S_1.*type* = *ok* and S_2.*type* = *ok*
then *ok* else *type_error* }

$S \rightarrow$ while E then { S_1.*continue* := 1 } S_1
{ *S.type* := if *E.type=bool* then S_1.*type* else *type_error* }

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

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

$F \rightarrow F_1$; id (V) { *S.continue* := 0 } S
{ *addtype*(id.*entry*, *fun* (*V.type*));
F.type := if F_1.*type* = *ok* and *S.type* = *ok* then *ok* else *type_error* }

（b）以下是一个*L*-翻译模式片断，可以产生控制语句的 *TAC* 语句序列：

$P \rightarrow D$; { *S.next* := *newlable* } S { *gen*(*S.next* ':') }

$S \rightarrow$ if { *E.true* := *newlable*; *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* := *newlable*; *E.false* := *S.next* } E do
{ S_1.*next* := *newlable* } S_1
{ *S.code* := *gen*(S_1.*next* ':') || *E.code*|| *gen*(*E. true* ':') ||
S_1.*code* || *gen*('*goto*' S_1.*next*) }

$S \rightarrow$ { 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* }

其中的属性及语义函数与2.3.5中一致。

若在基础文法中增加关于 *continue* 语句的产生式

$S \rightarrow$ *continue*

试在该*L*-翻译模式片段基础上增加针对 *continue* 语句的语义处理内容（不改变*L*-翻译模式的特征）。

注：可设计引入新的属性或删除旧的属性，必要时给出解释。

参考解答：

$P \rightarrow D$; { *S.next* := *newlable*; *S.continue* := *newlable* } S { *gen*(*S.next* ‘:’) }
$S \rightarrow$ if { *E.true* := *newlable*; *E.false* := *S.next* } E then
{ S_1.*next* := *S.next*; S_1 . *continue* := *S* . *continue* } S_1
{ *S.code* := *E.code* || *gen*(*E. true* ‘:’) || S_1.*code* }
$S \rightarrow$ while { *E.true* := *newlable*; *E.false* := *S.next* } E do
{ S_1.*next* := *newlable*; S_1 . *continue* := S_1.*next* } S_1
{ *S.code* := *gen*(S_1.*next* ‘:’) || *E.code*|| *gen*(*E. true* ‘:’) ||
S_1.*code* || *gen*(‘*goto*’ S_1.*next*) }
$S \rightarrow$ { S_1.*next* := *newlable*; S_1 . *continue* := *S* . *continue* } S_1 ;
{ S_2.*next* := *S.next*; S_2 . *continue* := *S* . *continue* } S_2
{ *S.code* := S_1.*code* || *gen*(S_1.*next* ‘:’) || S_2.*code* }
$S \rightarrow$ *continue*
{ *S .code := gen(‘goto’S* . *continue)* }

（c）以下是一个 *S*-翻译模式/属性文法片断，可以产生控制语句（注意：条件语句和上题不同）的 *TAC* 语句序列：

$P \rightarrow D$; S M { *backpatch*(*S.nextlist*, *M.gotostm*) }
$S \rightarrow$ 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 then M_2 S_1 { *backpatch*(S_1.*nextlist*, M_1.*gotostm*) ;
backpatch(*E.truelist*, M_2.*gotostm*) ;
S.nextlist := *E.falselist*;
emit(‘*goto*’, M_1.*gotostm*) }
$S \rightarrow S_1$; M S_2 { *backpatch*(S_1.*nextlist*, M_1.*gotostm*) ;
S.nextlist := S_2.*nextlist* }
$M \rightarrow \varepsilon$ { *M.gotostm* := *nextstm* }
$N \rightarrow \varepsilon$ { *N.nextlist* := *makelist*(*nextstm*); *emit*(‘*goto* _’) }

其中的属性及语义函数与2.3.6中一致。

若在基础文法中增加关于 *continue* 语句的产生式

$$S \rightarrow continue$$

试在该 *S*-翻译模式片段基础上增加针对 *continue* 语句的语义处理内容（不改变 *S*-翻译模式的特征）。

注：可设计引入新的属性或删除旧的属性，必要时给出解释。

参考解答：

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

$S \rightarrow$ 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. continuelist := *merge*(S_1. *continuelist*, S_2. *continuelist*) }

$S \rightarrow$ while $M_1\ E$ then $M_2\ S_1$ { *backpatch*(S_1.*nextlist*, M_1.*gotostm*) ;
backpatch(*E.truelist*, M_2.*gotostm*) ;
backpatch(S_1. *continuelist*, M_1.*gotostm*);
S. continuelist := *“”;*
emit(‘*goto*’, M_1.*gotostm*) }

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

$S \rightarrow$ *continue;* { *S. continuelist* := *makelist* (*nextstm*) ; *S. nextlist* := *“”;*
emit (‘goto _’) }

$M \rightarrow \varepsilon$ { *M.gotostm* := *nextstm* }

$N \rightarrow \varepsilon$ { *N.nextlist* := *makelist*(*nextstm*); *emit*(‘*goto* _’) }